文档中心

Version v5.6, 2026-09-17

关于IvorySQL 1
欢迎 3
发行说明 3
快速上手 6
.1. 快速开始 6
安装部署 9
.1. 安装指南 9
.2. 集群搭建 14
Oracle兼容功能 19
.1. 1、大小写转换 19
.2. 2、LIKE操作符 25
.3. 3、匿名块 27
.4. 4、函数与存储过程 28
.5. 5、内置数据类型与内置函数 40
.6. 6、端口与IP 62
.7. 7、XML函数 63
.8. 8、sequence 68
.9. 9、包 69
.10. 10、不可见列 72
.11. 11、RowID 74
.12. 12、OUT 参数 76
.13. 13、%TYPE、%ROWTYPE 79
.14. 14、NLS 参数 83
.15. 15、Force View 86
.16. 16、嵌套子函数 87
.17. 17、sys_guid 函数 89
.18. 18、空字符串转null 89
.19. 19、CALL INTO 90
1. 实时查询分析 96
容器化与云服务 98
容器化指南 98
云服务平台指南 152
IvorySQL生态 175
.1. 芯片架构适配 175
.2. 操作系统适配 175
生态组件适配 177
监控运维 203

.1. 日常监控 203
.2. 日常维护 244
.3. 运维管理 266
数据迁移 306
.1. 迁移指南 306
IvorySQL开发者 322
社区贡献指南 322
.1. 开发者指南 326
IvorySQL架构设计 402
参考指南 447
.1. 工具参考 447
.2. asciidoc语法快速参考 552
PG参数参考手册 560
PG函数参考手册 672
常见问题解答 767
.1. FAQ 767

关于IvorySQL

IvorySQL简介

概述

IvorySQL是一款以PostgreSQL为基础进行开发，并且兼容Oracle的开源数据库。

IvorySQL社区始终承诺与PostgreSQL数据库保持100%兼容，并且可以直接替换最新版本的PostgreSQL。

IvorySQL增加了一个名为 `ivorysql.compatible_mode` 的GUC参数用以控制IvorySQL的兼容模式，该参数有 `oracle` 和 `pg` 两种值。在初始化数据目录的时候，通过指定 `-m` 参数来指定数据目录的兼容模式，`-m pg` 则数据目录为PostgreSQL模式，该模式下 `ivorysql.compatible_mode` 参数将会失效，`-m oracle` 或者不指定 `-m` 参数则数据目录为兼容Oracle模式，该模式下 `ivorysql.compatible_mode` 参数初始值为 `oracle` 并且不支持部分PostgreSQL的语法，通过 `set ivorysql.compatible_mode to pg` 就可以使得数据库100%支持PostgreSQL的语法及功能。

IvorySQL的亮点之一是PL/iSQL过程语言，它支持Oracle的PL/SQL语法。同时，IvorySQL通过增加与内核绑定的插件 ivorysql_ora 来实现兼容Oracle的功能，目前实现的功能包括内置函数、数据类型、系统视图、merge以及GUC参数的增加，未来将会继续以绑定内核的插件的形式来实现新的兼容功能。

IvorySQL项目是在Apache 2.0许可证下发布的，社区鼓励且欢迎所有类型的贡献和参与。

产品目标和范围

我们致力于遵守开源方式的原则，我们坚信建立一个健康和包容的社区。我们坚持认为，好的想法可以来自任何地方。虽然IvorySQL目前的版本主要关注Oracle兼容性功能，但未来的路线图和功能集将由社区以开源方式确定。

核心特性

IvorySQL基于PostgreSQL数据库开发，与Oracle数据库兼容，具有强大的兼容性。适用于PostgreSQL数据库和Oracle数据库场景。

竞争优势

- 核心开源：IvorySQL的核心代码包括兼容功能全部在开源协议下公开，没有厂商的限制。并应用于瀚高股份数据库公司实例，且拥有一个活跃的开发者社区。
- 兼容Oracle：可以将Oracle数据库迁移到IvorySQL。
- 可定制化：只需下载代码，并按照你的想法自定义。
- 简单易用：对系统管理员来说，IvorySQL 大幅降低了管理和维护的代价。对开发者来说，IvorySQL提供了简单的接口、极简的解决方案和与第三方工具的无缝集成。对数据分析专家来说，IvorySQL 提供了便捷的数据访问能力。
- 瀚高支持：由领先的PostgreSQL数据库提供商瀚高股份提供支持。

技术生态

IvorySQL基于PostgreSQL，具有完整的SQL、坚如磐石的可靠性和庞大的生态系统。

核心应用场景

IvorySQL数据库的主要应用场景：

- 企业数据库

> 如 ERP、交易系统、财务系统涉及资金、客户等信息，数据不能丢失且业务逻辑复杂，选择 IvorySQL
> 作为数据底层存储，一是可以帮助您在数据一致性前提下提供高可用性，二是可以用简单的编程实现复杂的业务逻辑。

- 含 LBS 的应用

> 大型游戏、O2O 等应用需要支持世界地图、附近的商家，两个点的距离等能力，PostGIS
> 增加了对地理对象的支持，允许您以 SQL
> 运行位置查询，而不需要复杂的编码，帮助您更轻松理顺逻辑，更便捷的实现
> LBS，提高用户粘性。

- 数据仓库和大数据

> IvorySQL
> 更多数据类型和强大的计算能力，能够帮助您更简单搭建数据库仓库或大数据分析平台，为企业运营加分。

- 建站或 App

> IvorySQL 良好的性能和强大的功能，可以有效的提高网站性能，降低开发难度。

- 数据库迁移

> 如果需要将Oracle数据库迁移到PostgreSQL数据库，可以直接使用IvorySQL数据库进行迁移。

主要、基本功能

IvorySQL是一个功能强大的开源对象关系数据库管理系统(ORDBMS)。
用于安全地存储数据，支持最佳做法，并允许在处理请求时检索它们。除此之外，还兼容了Oracle的语法，适用于使用Oracle的场景。

与Oracle的兼容性

- 大小写转换
- LIKE操作符
- 匿名块
- 函数与存储过程
- 内置数据类型与内置函数
- 端口与IP
- XML函数
- sequence
- 包
- 不可见列
- RowID

- OUT 参数
- %TYPE、%ROWTYPE
- NLS 参数
- Force View
- 嵌套子函数
- sys_guid 函数
- 空字符串转null
- CALL INTO

欢迎

IvorySQL 是唯一一款基于 PostgreSQL 研发的兼容 Oracle 的数据库。

开始

从 Github 下载源码开始。

发布

请前往 IvorySQL 发布页面。

关于 IvorySQL

IvorySQL 项目是瀚高软件提出的一个开源项目，旨在将 Oracle 兼容性功能添加到流行的 PostgreSQL 数据库中。

IvorySQL 开源并且可以免费使用，如果您有任何建议请联系 support@ivorysql.org

文档下载

IvorySQL v5.6 pdf 文档

发行说明

版本概览

[发布日期：2026 年 9 月 17 日]

IvorySQL 5.6 将 PostgreSQL 内核升级至 18.6，并针对 Oracle 兼容性、内存安全、客户端接口和扩展构建提供多项修复。有关更新的完整列表，请访问我们的 文档站点。

上游更新

由于在发布准备过程中发现了回归问题，PostgreSQL 18.5 未发布。

- PostgreSQL 18.6
  1. 通过新增的服务器参数 `output_plugin_libraries`，将逻辑解码输出插件限制为该参数指定的集合。
  2. 修复 contrib/pgcrypto 的 PGP 加密，使其能够检测不支持的加密算法。
  3. 修复 psql：即使脚本中的 `COPY … FROM STDIN` 命令在发送 `PGRES_COPY_IN` 前失败，也能跳过该命令后的内联数据。
  4. 在此前未执行检查的位置检查数据类型的 `USAGE` 权限。
  5. 角色变更后，使依赖角色的缓存计划失效。

6. 确保并行 GIN 索引构建能够正确更新表的 `pg_class.reltuples` 值。

更多详情，请参阅 PostgreSQL 18.6 发布说明。

增强内容

- 升级至 PG 18.6 内核：Feature #1719
  PostgreSQL 内核已升级至 18.6 版本。
- AI 辅助贡献指南：Feature #1473
  制定相关指南，涵盖许可证合规、人工审查与签署责任、AI 工具归属标注、AI 生成代码占比披露，以及负责任地参与代码审查。
- IvorySQL 5.6 在线体验：Feature #12
  现已提供交互式浏览器环境，用户可实时体验与评估 IvorySQL 5.6，无需安装。
- 全平台打包
  为 X86、ARM、MIPS 和龙芯架构提供多平台安装介质。
- 容器化部署支持
  支持通过 Docker Compose、Podman、Swarm、Helm 以及 IvorySQL 5.6 Operator 快速部署单机或高可用的 IvorySQL 5.6 集群。
- 发布 IvorySQL Cloud 5.6（数据库全生命周期可视化管理平台）
  提供全托管的可视化控制平台，负责 IvorySQL 5.6 数据库订阅、端到端的全生命周期编排，并集成周边生态服务。
- 扩展
  支持 34 个扩展和工具：AGE、ddlx、pg_ai_query、pg_bigm、pg_bulkload、pg_cron、pg_curl、pg_hint_plan、pg_jieba、pg_partman、pg_profile、pg_readonly、pg_repack、pg_show_plans、pg_stat_monitor、pg_textsearch、pgagent、pgaudit、pgBackRest、pgBouncer、PgDog、pgnodemx、PGroonga、pgRouting、pgsql-http、pgtt、pgvector、plpgsql_check、PostGIS、redis_fdw、set_user、system_stats、wal2json、zhparser。

缺陷修复

- 修复 Oracle 兼容的 interval 乘除法中可能发生的溢出和 NaN 处理不当问题：PR #1211
- 禁止通过 ALTER USER/ROLE/DATABASE SET 持久化 ivorysql.compatible_mode，并封堵 SET … TO DEFAULT 和针对该参数执行 RESET 的绕过方式：PR #1372
- 修复表达式求值期间，带包限定符的变量引用被解析为无关局部变量的问题：PR #1381
- 修复使用 GCC 14 及更高版本时的构建失败问题：PR #1385
- 修复序列中相互冲突或重复的 SHARD 选项校验问题：PR #1401
- 修复第三方 PostgreSQL 扩展针对 IvorySQL 构建时的失败问题：PR #1448
- 使 ivorysql.enable_case_switch 在 SHOW ALL 中可见，并可通过配置文件进行配置：PR #1450
- 修复 DBMS_UTILITY 回溯和调用栈输出中缺少包名及模式限定错误的问题：PR #1471
- 修复 REGEXP_COUNT、REGEXP_SUBSTR、REGEXP_INSTR 和 REGEXP_REPLACE 中换行符处理错误以及 m/n 选项行为异常的问题：PR #1489
- 修复 Ivyresult 提前返回路径上的内存泄漏问题：PR #1556
- 修复 psql 命名绑定查询转义和 \\bind 处理中发生的堆缓冲区越界问题：PR #1569
- 修复 XML 函数使用过长的命名空间前缀或 URL 时发生的堆缓冲区溢出问题：PR #1827

源码仓库

IvorySQL 的开发工作主要在以下四个代码仓库中进行：

- IvorySQL 数据库源码：https://github.com/IvorySQL/IvorySQL
- IvorySQL 官网：https://github.com/IvorySQL/Ivory-www
- IvorySQL 文档：https://github.com/IvorySQL/IvorySQL_docs
- IvorySQL Docker：https://github.com/IvorySQL/docker_library

贡献者名单

以下人员（按字母顺序）作为补丁作者、提交者、审阅者、测试者或问题报告者，为本次发布做出了贡献。

- 123123213weqw
- Aias00
- ai-yang
- btlqql
- Cédric Villemain
- db-nikhil
- digoal
- enkilee
- goeason-world
- Grant Zhou
- lispking
- messere1
- tian1220A
- xuhuicai
- yaozichen2025
- Yasir Hussain Shah
- yyqdbngt
- Z95-cmyk
- 高雪玉
- 韩建桥
- 矫顺田
- 梁翔宇
- 刘晓辉
- 吕新杰
- 牛世继
- 潘振浩
- 石卓妍
- 隋戈
- 陶郑
- 严少安
- 赵法威
- 张哲

完整变更记录

快速上手

.1. 快速开始

环境要求

- 硬件要求

配置参数	最低配置	推荐配置
CPU	4核	16核
内存	4GB	64GB
存储	800MB，机械硬盘	5GB以上，SSD或NvMe
网络	千兆网络	万兆网络

- 软件要求

  IvorySQL数据库目前支持的操作系统包括但不限于CentOS 8.X、CentOS Stream 9以及Ubuntu系统。

快速安装

> 快速开始示例所使用的操作系统为CentOS Stream 9。

从yum源安装IvorySQL数据库

- 安装前准备

  安装前请先创建一个用户，并赋予其root权限，安装和使用均以该用户执行。这里以ivorysql用户为例。如何创建sudo用户

- 下载安装

  创建或编辑IvorySQL yum源配置文件/etc/yum.repos.d/ivorysql.repo

```
vim /etc/yum.repos.d/ivorysql.repo
[ivorysql5]
name=IvorySQL Server 5 $releasever - $basearch
baseurl=https://yum.highgo.com/dists/ivorysql-rpms/5/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=0
```

保存退出后，安装IvorySQL5

```
$ sudo dnf install -y ivorysql5-5.6
```

> 正确安装后，数据库将被安装在/usr/ivory-5/路径下的IvorySQL-version(如:IvorySQL-{ivorysql-version})文件夹内

执行以下命令为ivorysql用户赋权：

```
$ sudo chown -R ivorysql:ivorysql /usr/ivory-5
```

- 配置环境变量

  将以下配置写入~/.bash_profile文件并使用source命令该文件使环境变量生效：

```
PATH=/usr/ivory-5/bin:$PATH
export PATH
PGDATA=/usr/ivory-5/data
export PGDATA
```

```
$ source ~/.bash_profile
```

- 数据库初始化

```
$ initdb -D /usr/ivory-5/data
```

> 其中-D参数用来指定数据库的数据目录。更多参数使用方法，请使用initdb --help命令获取。

- 启动数据库服务

```
$ pg_ctl -D /usr/ivory-5/data -l ivory.log start
```

其中-D参数用来指定数据库的数据目录，如果[_3-1:::配置环境变量] 配置了PGDATA，则该参数可以省略。-l参数用来指定日志文件。更多参数使用方法，请使用pg_ctl --help命令获取。

查看确认数据库启动成功：

```
$ ps -ef | grep postgres
ivorysql  3214     1  0 20:35 ?        00:00:00 /usr/ivory-5/bin/postgres -D
/usr/ivory-5/data
ivorysql  3215  3214  0 20:35 ?        00:00:00 postgres: checkpointer
ivorysql  3216  3214  0 20:35 ?        00:00:00 postgres: background writer
ivorysql  3218  3214  0 20:35 ?        00:00:00 postgres: walwriter
ivorysql  3219  3214  0 20:35 ?        00:00:00 postgres: autovacuum launcher
ivorysql  3220  3214  0 20:35 ?        00:00:00 postgres: logical replication launcher
ivorysql  3238  1551  0 20:35 pts/0    00:00:00 grep --color=auto postgres
```

docker方式运行

- 从Docker Hub上获取IvorySQL镜像

```
$ docker pull ivorysql/ivorysql:5.6-ubi8
```

国内用户也可使用 IvorySQL 社区的 Harbor 镜像服务加速拉取，例如：`docker pull registry.highgo.com/ivorysql/ivorysql:5.6-ubi8`。

- 运行IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/ivorysql:5.6-ubi8
```

- 查看IvorySQL容器运行是否成功

```
$ docker ps | grep ivorysql
CONTAINER ID   IMAGE               COMMAND                  CREATED          STATUS
PORTS                               NAMES
6faa2d0ed705   ivorysql:5.6-ubi8   "docker-entrypoint.s…"   50 seconds ago   Up 49
seconds   5866/tcp, 0.0.0.0:5434->5432/tcp   ivorysql
```

数据库连接

psql连接数据库：

```
$ psql -d YOUR_DATABASE
psql (18.6)
Type "help" for help.

ivorysql=#
```

```
  其中-
d参数用来指定想要连接到的数据库名称。IvorySQL默认使用ivorysql数据库，但较低版本的Ivory
SQL首次使用时需用户先连接postgres数据库，然后自己创建ivorysql数据库。较高版本的IvorySQ
L则已为用户创建好ivorysql数据库，可以直接连接。

  更多参数使用方法，请使用psql --help命令获取。
```

Docker运行IvorySQL时，需要添加额外参数，如 psql -d ivorysql -U ivorysql -h 127.0.0.1 -p 5434

现在可以开始使用IvorySQL啦！就是这么简单！

想要获得更多安装方式，请参考安装指南

安装部署

.1. 安装指南

概述

IvorySQL安装方式包括以下5种：

- yum源安装
- docker安装
- rpm安装
- 源码安装
- deb安装

本章将详细介绍每种方式的安装、运行及卸载过程。想要更快获得IvorySQL，请参阅快速开始。

同样，安装前请先创建一个用户，并赋予其root权限，安装、使用和卸载均以该用户执行。这里以ivorysql用户为例。

yum源安装

创建或编辑IvorySQL yum源配置文件/etc/yum.repos.d/ivorysql.repo

```
vim /etc/yum.repos.d/ivorysql.repo
[ivorysql5]
name=IvorySQL Server 5 $releasever - $basearch
baseurl=https://yum.highgo.com/dists/ivorysql-rpms/5/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=0
```

保存退出后，安装IvorySQL5

```
$ sudo dnf install -y ivorysql5-5.6
```

docker安装

- 从Docker Hub上获取IvorySQL镜像

```
$ docker pull ivorysql/ivorysql:5.6-ubi8
```

> 国内用户也可使用 IvorySQL 社区的 Harbor 镜像服务加速拉取，将上述地址替换为 `registry.highgo.com/ivorysql/ivorysql:5.6-ubi8` 即可。

- 运行IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/ivorysql:5.6-ubi8
```

-e参数说明

参数名	是否必填	描述
IVORYSQL_USER	否	数据库用户，默认 ivorysql
IVORYSQL_PASSWORD	是	数据库用户密码
IVORYSQL_DB	否	数据库名称，默认 ivorysql
POSTGRES_HOST_AUTH_METHOD	否	修改主机身份验证方式，参考值：md5
POSTGRES_INITDB_ARGS	否	为initdb是添加额外参数，参考值："--data-checksums"
PGDATA	否	将数据目录定义到其他路径文件夹下（例如子目录），默认/var/lib/ivorysql/data
POSTGRES_INITDB_WALDIR	否	定义IvorySQL transaction文件夹路径，默认在数据目录（PGDATA）的子目录中。

1. 不推荐将POSTGRES_HOST_AUTH_METHOD参数填写为trust，这样将会使IVORYSQL_PASSWORD设置失效。
2. 如果POSTGRES_HOST_AUTH_METHOD参数设置为scram-sha-256，同时需将POSTGRES_INITDB_ARGS设置为—auth-host=scram-sha-256，才会使数据库正确进行初始化。

rpm安装

- 安装依赖

```
$ sudo dnf install -y lz4 libicu libxslt python3
```

- 获取rpm包

```
$ sudo wget
https://github.com/IvorySQL/IvorySQL/releases/download/IvorySQL_5.6/IvorySQL-5.6-
9d890e9-20251120.x86_64.rpm
```

- 安装rpm包

  使用以下命令安装所有rpm包：

```
$ sudo yum --disablerepo=* localinstall *.rpm
```

数据库将被安装在/usr/ivory-5/路径下。

源码安装

- 安装依赖

```
# CentOS
```

```
$ sudo dnf install -y bison readline-devel zlib-devel openssl-devel uuid-devel
$ sudo dnf groupinstall -y 'Development Tools'

# Ubuntu
$ sudo apt install -y bison libreadline-dev zlib1g-dev libssl-dev
$ sudo apt install -y build-essential
$ sudo apt install -y pkg-config uuid-dev
```

- 获取IvorySQL源代码

```
$ git clone https://github.com/IvorySQL/IvorySQL.git
$ cd IvorySQL
$ git checkout -b IVORY_REL_5_STABLE origin/IVORY_REL_5_STABLE
```

- 配置

  在IvorySQL目录下，执行以下命令进行配置，请使用—prefix指定安装目录:

```
$ ./configure --prefix=/usr/local/ivorysql/ivorysql-5
```

- 编译

  执行以下命令进行编译:

```
$ make
```

> 编译完毕，安装之前可先执行make check或make all-check-world测试刚刚编译的结果。

- 安装

  执行以下命令安装，数据库将被安装在上述由—prefix指定的路径下:

```
$ sudo make install
```

deb安装

- 安装依赖

```
$ sudo apt -y install pkg-config libreadline-dev libicu-dev libldap2-dev uuid-dev tcl-dev libperl-dev python3-dev bison flex openssl libssl-dev libpam-dev libxml2-dev libxslt-dev libossp-uuid-dev libselinux-dev gettext
```

- 获取deb包

```
$ sudo wget
```

```
https://github.com/IvorySQL/IvorySQL/releases/download/IvorySQL_5.6/IvorySQL-5.6-
a50789d-20250304.amd64.deb
```

- 安装deb包

```
$ sudo dpkg -i IvorySQL-5.6-a50789d-20250304.amd64.deb
```

数据库将被安装在/usr/ivory-5/路径下。

启动数据库

参考yum源安装、rpm安装、源码安装、deb安装的用户，需要手动启动数据库。

- 赋权

  执行以下命令为安装用户赋权，示例用户为ivorysql，安装目录为/usr/ivory-5/：

```
$ sudo chown -R ivorysql:ivorysql /usr/ivory-5/
```

- 配置环境变量

  将以下配置写入用户的~/.bash_profile文件并使用source命令该文件使环境变量生效：

```
PATH=/usr/ivory-5/bin:$PATH
export PATH
PGDATA=/usr/ivory-5/data
export PGDATA
```

```
$ source ~/.bash_profile
```

- 数据库初始化

```
$ mkdir /usr/ivory-5/data
$ initdb -D /usr/ivory-5/data
```

```
其中-D参数用来指定数据库的数据目录。更多参数使用方法，请使用initdb --help命令获取。
```

- 启动数据库服务

```
$ pg_ctl -D /usr/ivory-5/data -l ivory.log start
```

其中-D参数用来指定数据库的数据目录，如果[_4-1:::配置环境变量] 配置了PGDATA，则该参数可以省略。-l参数用来指定日志文件。更多参数使用方法，请使用pg_ctl --help命令获取。

查看确认数据库启动成功：

```
$ ps -ef | grep postgres
ivorysql  130427       1  0 02:45 ?        00:00:00 /usr/ivory-5/bin/postgres -D
/usr/ivory-5/data
ivorysql  130428  130427  0 02:45 ?        00:00:00 postgres: checkpointer
ivorysql  130429  130427  0 02:45 ?        00:00:00 postgres: background writer
ivorysql  130431  130427  0 02:45 ?        00:00:00 postgres: walwriter
ivorysql  130432  130427  0 02:45 ?        00:00:00 postgres: autovacuum launcher
ivorysql  130433  130427  0 02:45 ?        00:00:00 postgres: logical replication
launcher
ivorysql  130445  130274  0 02:45 pts/1    00:00:00 grep --color=auto postgres
```

数据库连接

psql连接数据库：

```
$ psql -d YOUR_DATABASE
psql (18.6)
Type "help" for help.

ivorysql=#
```

```
  其中-
d参数用来指定想要连接到的数据库名称。IvorySQL默认使用ivorysql数据库，但较低版本的Ivory
SQL首次使用时需用户先连接postgres数据库，然后自己创建ivorysql数据库。较高版本的IvorySQ
L则已为用户创建好ivorysql数据库，可以直接连接。

  更多参数使用方法，请使用psql --help命令获取。
```

Docker运行IvorySQL时，需要添加额外参数，参考：psql -d ivorysql -U ivorysql -h 127.0.0.1 -p 5434

卸载IvorySQL

使用任何一种方法卸载前请先停止数据库服务并做好数据备份。

yum源安装的卸载

执行以下命令依次卸载：

```
$ sudo dnf remove -y ivorysql5-5.0
```

docker安装的卸载

执行以下命令，使IvorySQL容器停止运行，并删除IvorySQL容器和镜像：

```
$ docker stop ivorysql
$ docker rm ivorysql
$ docker rmi ivorysql/ivorysql:5.6-ubi8
```

rpm安装的卸载

执行以下命令卸载并清理文件夹：

```
$ sudo yum remove --disablerepo=* ivorysql5\*
$ sudo rm -rf /usr/ivory-5
```

源码安装的卸载

执行以下命令卸载数据库并清理文件夹：

```
$ sudo make uninstall
$ make clean
$ sudo rm -rf /usr/ivory-5
```

deb安装的卸载

执行以下命令卸载数据库并清理文件夹：

```
$ sudo dpkg -P IvorySQL-5.6
$ sudo rm -rf /usr/ivory-5
```

.2. 集群搭建

主节点

安装并启动数据库

yum源快速安装数据库，请参考从yum源安装ivorysql数据库。

想要获取更多安装方式，请参考安装指南。

> 主节点数据库需要安装并启动

关闭防火墙

集群中各节点均需关闭防火墙才能正常通信。

```
$ sudo systemctl stop firewalld
```

配置文件

为了从主节点向备节点搭建流复制，主节点需要对data目录下的postgresql.conf和pg_hba.conf两个文件进行配置。

- postgresql.conf

  将以下配置追加到postgresql.conf文件末尾：

```
listen_addresses = '*'
max_connections = 100
wal_level = replica
max_wal_senders = 5
hot_standby = on
```

- pg_hba.conf

  将以下配置追加到pg_hba.conf文件末尾：

```
host all all 0.0.0.0/0 trust
host replication all 0.0.0.0/0 trust
```

> 示例中的pg_hba的配置，仅作为demo用来测试，这种配置会导致数据库密码失效，请根据环境实际情况进行配置

重启主节点数据库服务

```
$ pg_ctl restart
```

备节点

安装数据库

Yum源快速安装数据库，请参考从yum源安装ivorysql数据库。

想要获取更多安装方式，请参考安装指南。

> 备节点数据库只需要安装，不需要启动

关闭防火墙

集群中各节点均需关闭防火墙才能正常通信。

```
$ sudo systemctl stop firewalld
```

搭建流复制

在备节点上执行以下命令，创建一个主节点的基础备份，即搭建流复制：

```
$ sudo pg_basebackup -F p -P -X fetch -R -h <primary_ip> -p <primary_port> -U ivorysql
-D /usr/local/ivorysql/ivorysql-5/data
```

- -h为主节点ip；

- -p为主节点数据库端口号，默认为5432；
- -U为数据库用户；
- -D为要创建的备节点数据库数据目录。

其他参数使用方法，请使用pg_basebackup --help命令获取。

配置环境变量

将以下配置写入~/.bash_profile文件：

```
PATH=/usr/local/ivorysql/ivorysql-5/bin:$PATH
export PATH
PGDATA=/usr/local/ivorysql/ivorysql-5/data
export PGDATA
```

source该文件使环境变量生效：

```
$ source ~/.bash_profile
```

启动备节点数据库服务

```
$ pg_ctl -D /usr/local/ivorysql/ivorysql-5/data start
```

集群的使用

查看集群状态

在主节点上执行以下命令可以看到walsender：

```
$ ps -ef |grep postgres
...
ivorysql 11176  8067  0 21:54 ?        00:00:00 postgres: walsender ivorysql
192.168.31.102(53416) streaming 0/7000060...
```

而备节点则可看到walreceiver：

```
$ ps -ef | grep postgres
...
ivorysql  6567  6139  0 21:54 ?        00:00:00 postgres: walreceiver streaming
0/7000060
...
```

在主节点上psql连接数据库，并查看集群状态：

```
$ psql -d ivorysql
```

```
psql (18.6)
Type "help" for help.

ivorysql=# select * from pg_stat_replication;
  pid  | usesysid | usename  | application_name |  client_addr   | client_hostname |
client_port |         backend_start         | backend_
xmin |   state   | sent_lsn  | write_lsn | flush_lsn | replay_lsn | write_lag |
flush_lag | replay_lag | sync_priority | sync_state |
    reply_time
-------+----------+----------+------------------+----------------+-----------------+
-------------+-------------------------------+---------
-----+-----------+-----------+-----------+-----------+------------+-----------+
-----------+------------+---------------+------------+------
-------------------------
 11176 |       10 | ivorysql | walreceiver      | 192.168.31.102 |                 |
53416 | 2024-12-18 21:54:52.041847-05 |
     | streaming | 0/7000148 | 0/7000148 | 0/7000148 | 0/7000148  |           |
|            |             0 | async      | 2024-
12-18 22:52:07.325111-05
(1 row)
```

这里192.168.31.102为备节点的ip，async表示数据同步方式为异步流复制。 === 使用集群
集群中所有的写操作均在主节点执行，读操作则主备节点都可以执行。主节点的数据通过流复制同步到备节点。主节点写操作的结果在任何一个备节点都能够查询到。
例如，在主节点创建一个新的数据库test，并在主节点进行查询：

```
$ psql -d ivorysql
psql (18.6)
Type "help" for help.

ivorysql=# create database test;
CREATE DATABASE
ivorysql=# \l
                                                    List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+
------------+-----------+-----------------------
 ivorysql  | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           |
 template0 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
           |          |          |                 |             |             |
|           | ivorysql=CTc/ivorysql
 template1 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
```

```
           |          |          |                |             |             |
|           | ivorysql=CTc/ivorysql
 test      | ivorysql | UTF8     | libc           | en_US.UTF-8 | en_US.UTF-8 |
|           |
(4 rows)
```

在备节点查询:

```
$ psql -d ivorysql
psql (18.6)
Type "help" for help.

ivorysql=# \l
                                                    List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU
Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+
------------+-----------+-----------------------
 ivorysql  | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           |
 template0 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
           |          |          |                |             |             |
|           | ivorysql=CTc/ivorysql
 template1 | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           | =c/ivorysql          +
           |          |          |                |             |             |
|           | ivorysql=CTc/ivorysql
 test      | ivorysql | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
|           |
(4 rows)
```

Oracle兼容功能

.1. 1、大小写转换

目的

- 为了满足PG和Oracle的引用标识符大小写兼容，ivorysql设计了三种引用标识符的大小写转换模式。通过guc参数“identifier_case_switch”选择转换模式;

功能

大小写转换的三种模式（默认为interchange）

- 如果 guc参数“identifier_case_switch”值为“normal”

```
1). 保持双引号所引用的标识符中的字母大小写不变。
```

- 如果 guc参数“identifier_case_switch”值为“interchange”：

```
1). 如果双引号所引用的标识符中的字母全部为大写，则将大写转换为小写。
```

```
2). 如果双引号所引用的标识符中的字母全部为小写，则将小写转换为大写。
```

```
3). 如果用双引号引起来的标识符中的字母是大小写混合的，则保持标识符不变。
```

- 如果 guc参数“identifier_case_switch”值为“lowercase”

```
1). 如果双引号所引用的标识符中的字母全部为大写，则将大写转换为小写。
```

```
2). 如果用双引号引起来的标识符中的字母是大小写混合的，则保持标识符不变。
```

初始化数据库集簇时

- 在initdb程序中加入 -C选项设置大小写转换模式，-C对应的值为：

```
"normal"   ------ "0"同义
```

```
"interchange" ------ "1"同义
```

```
"lowercase"  ------ "2"同义
```

在初始化数据库集簇的过程中，将大小写转换模式保存到data目录的global/pg_control文件中。

经典用例

normal

```
ivorysql=# SET ivorysql.compatible_mode to oracle;
SET

ivorysql=# SET ivorysql.enable_case_switch = true;
SET

ivorysql=# SET ivorysql.identifier_case_switch = normal;
SET

ivorysql=# CREATE TABLE "NORMAL_1"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "Normal_2"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "normal_3"(c1 int, c2 int);
CREATE TABLE

ivorysql=# select * from "NORMAL_1";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Normal_1";
ERROR:  relation "Normal_1" does not exist
LINE 1: select * from "Normal_1";

ivorysql=# select * from "normal_1";
ERROR:  relation "normal" does not exist
LINE 1: select * from "normal";

ivorysql=# select * from NORMAL_1;
ERROR:  relation "normal_1" does not exist
LINE 1: select * from NORMAL_1;
```

```
ivorysql=# select * from "Normal_2";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "NORMAL_2";
ERROR:  relation "NORMAL_2" does not exist
LINE 1: select * from "NORMAL_2";

ivorysql=# select * from "normal_2";
ERROR:  relation "normal_2" does not exist
LINE 1: select * from "normal_2";

ivorysql=# select * from Normal_2;
ERROR:  relation "normal_2" does not exist
LINE 1: select * from Normal_2;

ivorysql=# select * from "normal_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "NORMAL_3";
ERROR:  relation "NORMAL_3" does not exist
LINE 1: select * from "NORMAL_3";

ivorysql=# select * from "Normal_3";
ERROR:  relation "Normal_3" does not exist
LINE 1: select * from "Normal_3";

ivorysql=# drop table "NORMAL_1";
ERROR:  relation "NORMAL_1" does not exist
ivorysql=# drop table "Normal_2";
DROP TABLE
ivorysql=# drop table "normal_3";
DROP TABLE
```

interchange

```
ivorysql=# SET ivorysql.compatible_mode to oracle;
SET
```

```
ivorysql=# SET ivorysql.enable_case_switch = true;
SET

ivorysql=# SET ivorysql.identifier_case_switch = interchange;
SET

ivorysql=# CREATE TABLE "INTER_CHANGE_1"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "Inter_Change_2"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "inter_change_3"(c1 int, c2 int);
CREATE TABLE

ivorysql=# select * from "INTER_CHANGE_1";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Inter_Change_1";
ERROR:  relation "Inter_Change_1" does not exist
LINE 1: select * from "Inter_Change_1";

ivorysql=# select * from "inter_change_1";
ERROR:  relation "INTER_CHANGE_1" does not exist
LINE 1: select * from "inter_change_1";

ivorysql=# select * from INTER_CHANGE_1;
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Inter_Change_2";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "INTER_CHANGE_2";
ERROR:  relation "inter_change_2" does not exist
LINE 1: select * from "INTER_CHANGE_2";
```

```
ivorysql=# select * from "inter_change_2";
ERROR:  relation "INTER_CHANGE_2" does not exist
LINE 1: select * from "inter_change_2";

ivorysql=# select * from Inter_Change_2;
ERROR:  relation "inter_change_2" does not exist
LINE 1: select * from Inter_Change_2;

ivorysql=# select * from "inter_change_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "INTER_CHANGE_3";
ERROR:  relation "inter_change_3" does not exist
LINE 1: select * from "INTER_CHANGE_3";

ivorysql=# select * from "Inter_Change_3";
ERROR:  relation "Inter_Change_3" does not exist
LINE 1: select * from "Inter_Change_3";

ivorysql=# select * from inter_change_3;
ERROR:  relation "inter_change_3" does not exist
LINE 1: select * from "INTER_CHANGE_3";

ivorysql=# drop table "INTER_CHANGE_1";
DROP TABLE
ivorysql=# drop table "Inter_Change_2";
DROP TABLE
ivorysql=# drop table "inter_change_3";
DROP TABLE
```

lowercase

```
ivorysql=# SET ivorysql.compatible_mode to oracle;
SET

ivorysql=# SET ivorysql.enable_case_switch = true;
SET

ivorysql=# SET ivorysql.identifier_case_switch = lowercase;
SET
```

```
ivorysql=# CREATE TABLE "LOWER_CASE_1"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "Lower_Case_2"(c1 int, c2 int);
CREATE TABLE

ivorysql=# CREATE TABLE "lower_case_3"(c1 int, c2 int);
CREATE TABLE

ivorysql=# select * from "LOWER_CASE_1";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Lower_Case_1";
ERROR:  relation "Lower_Case_1" does not exist
LINE 1: select * from "Lower_Case_1";

ivorysql=# select * from "lower_case_1";
 c1 | c2
----+----
(0 行记录)


ivorysql=# select * from LOWER_CASE_1;
 c1 | c2
----+----
(0 行记录)


ivorysql=# select * from "Lower_Case_2";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "LOWER_CASE_2";
ERROR:  relation "lower_case_2" does not exist
LINE 1: select * from "LOWER_CASE_2";

ivorysql=# select * from "lower_case_2";
ERROR:  relation "lower_case_2" does not exist
```

```
LINE 1: select * from "lower_case_2";

ivorysql=# select * from Lower_Case_2;
ERROR:  relation "lower_case_2" does not exist
LINE 1: select * from Lower_Case_2;

ivorysql=# select * from "lower_case_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "LOWER_CASE_3";
 c1 | c2
----+----
(0 rows)

ivorysql=# select * from "Lower_Case_3";
ERROR:  relation "Lower_Case_3" does not exist
LINE 1: select * from "Lower_Case_3";

ivorysql=# select * from LOWER_CASE_3;
 c1 | c2
----+----
(0 行记录)

ivorysql=# drop table "NORMAL_1";
DROP TABLE
ivorysql=# drop table "Normal_2";
DROP TABLE
ivorysql=# drop table "normal_3";
DROP TABLE
```

.2. 2、LIKE操作符

目的

- 本文档意在为使用 like 模糊查询的人员提供一个深入了解兼容Oracle的模糊查询like实现的过程，是like兼容的实现文档。

功能说明

数据库名称	like模糊查询
oracle	oracle的字符串类型是varchar2，支持对数字、日期、字符串字段类型的列用Like关键字配合通配符来实现模糊查询

IvorySQL	IvorySQL的字符串基本类型是text，所以like是以text为基础上，其他IvorySQL的类型能隐式转换成text，这样不用创建operator就能自动转换

测试用例

```
create table t_ora_like (id int ,str1 varchar(8), date1 timestamp with time zone,
date2 time with time zone, num int, str2 varchar(8));
insert into t_ora_like (id ,str1 ,date1 ,date2) values (123456,'test1','2022-09-26
16:39:20','2022-09-26 16:39:20');
insert into t_ora_like (id ,str1 ,date1 ,date2) values (123457,'test2','2022-09-26
16:40:20','2022-09-26 16:40:20');
insert into t_ora_like (id ,str1 ,date1 ,date2) values (223456,'test3','2022-09-26
16:41:20','2022-09-26 16:41:20');
insert into t_ora_like (id ,str1 ,date1 ,date2) values (123458,'test4','2022-09-26
16:42:20','2022-09-26 16:42:20');

select * from t_ora_like where str1 like 'test%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 223456 | test3 | 2022-09-26 16:41:20.000000 +08:00 | 16:41:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(4 rows)

select * from t_ora_like where date1 like '2022%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 223456 | test3 | 2022-09-26 16:41:20.000000 +08:00 | 16:41:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(4 rows)

select * from t_ora_like where date2 like '16%';
   id   | str1  |               date1               |    date2    | num | str2
--------+-------+-----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 223456 | test3 | 2022-09-26 16:41:20.000000 +08:00 | 16:41:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(4 rows)
```

```
select * from t_ora_like where id like '123%';
   id   | str1  |              date1               |    date2    | num | str2
--------+-------+----------------------------------+-------------+-----+------
 123456 | test1 | 2022-09-26 16:39:20.000000 +08:00 | 16:39:20+08 |     |
 123457 | test2 | 2022-09-26 16:40:20.000000 +08:00 | 16:40:20+08 |     |
 123458 | test4 | 2022-09-26 16:42:20.000000 +08:00 | 16:42:20+08 |     |
(3 rows)

select * from t_ora_like where id like null;
 id | str1 | date1 | date2 | num | str2
----+------+-------+-------+-----+------
(0 rows)
```

.3. 3、匿名块

目的

- 本文档是PLSQL匿名块（anonymous block）兼容Oracle语法功能的设计文档，目的是可以在IvorySQL中兼容Oracle的匿名块语句。

功能说明

- 匿名块是能够动态地创建和执行过程代码的PLSQL结构，而不需要以持久化的方式将代码作为数据库对象储存在系统目录中。本次实现中IvorySQL主要兼容的是PLSQL匿名块的语法格式，我们主要处理的部分包括客户端工具psql、主服务器和PSQL端支持。

测试用例

```
declare
i integer := 10;
begin
 raise notice '%', i;
end;
/
NOTICE:  10
```

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE grade
    WHEN 'A' THEN raise notice 'Excellent';
    WHEN 'B' THEN raise notice 'Very Good';
  END CASE;
EXCEPTION
```

```
  WHEN CASE_NOT_FOUND THEN
    raise notice 'No such grade';
END;
/
NOTICE:  Very Good
```

.4. 4、函数与存储过程

目的

PostgreSQL支持函数(FUNCTION)和存储过程(PROCEDURE)，但是在语法上和Oracle有差异，为了让Oracle的PLSQL语句可以在IvorySQL上执行，也就是“语法兼容”，IvorySQL采取这样的解决方案：如果Oracle的子句在IvorySQL中存在相同功能的子句，则直接映射为对应的IvorySQL子句，否则只实现其语法，不实现其功能。

兼容支持

具体的Oracle兼容支持包括以下几个方面：

CREATE FUNCTION 语法支持 EDITIONABLE/NONEDITIONABLE

```
CREATE or replace EDITIONABLE FUNCTION ora_func RETURN integer AS
BEGIN
    RETURN 1;
END;
/

CREATE or replace NONEDITIONABLE FUNCTION ora_func RETURN integer IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 RETURN/IS 关键字，无需指定语言

```
CREATE or replace EDITIONABLE FUNCTION ora_func RETURN integer IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法函数没有参数，函数名后面不带()

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = METADATA
```

```
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 参数个数增加，最大值可在configure时指定

```
configure --help
    --with-max-funarg=MAXFUNARG
```

CREATE FUNCTION 语法中 END; 在psql中以/结束

```
CREATE or replace EDITIONABLE FUNCTION ora_func RETURN integer IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法变量声明前面没有 DECLARE 关键字

```
CREATE OR REPLACE
FUNCTION ora_func (num1 IN int, num2 IN int)
RETURN int
AS
 num3 int :=10;
 num4 int :=10;
 num5 int;
BEGIN
 num3 := num1 + num2;
 num4 := num1 * num2;
 num5 := num3 * num4;
RETURN num5;
END;
/
CREATE FUNCTION

select ora_func(5,9)from dual;
 ora_func
----------
      630
(1 row)
```

CREATE FUNCTION 语法支持 OUT 参数 NOCOPY 功能

```
CREATE OR REPLACE FUNCTION test_nocopy(a IN int, b OUT NOCOPY int, c IN OUT NOCOPY
int)
RETURN int
IS
BEGIN
   b := a;
   c := a;
   return 1;
END;
/
```

CREATE FUNCTION 语法支持 sharing 子句

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = METADATA
IS
BEGIN
    RETURN 1;
END;
/

CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 invoker_rights (AUTHID) 子句，默认权限改成 DR（DEFINER）

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID CURRENT_USER
IS
BEGIN
    RETURN 1;
END;
/

CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
```

```
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 ACCESSIBLE BY

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER ACCESSIBLE BY ( B )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER ACCESSIBLE BY ( A.B )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER ACCESSIBLE BY ( FUNCTION A.B )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D, PACKAGE E,
TRIGGER F, TYPE G )
IS
```

```
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 DEFAULT COLLATION

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 deterministic 子句，功能与 IvorySQL 的 IMMUTABLE 相同

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 parallel_enable 子句，功能与 IvorySQL 的 PARALLEL SAFE 相同

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE
IS
BEGIN
    RETURN 1;
END;
```

```
/
```

CREATE FUNCTION 语法支持 result_cache 子句

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ()
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 aggregate 子句

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING pg_catalog.int4
IS
BEGIN
    RETURN 1;
END;
/
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING int
IS
BEGIN
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 pipelined 子句

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING int
PIPELINED
IS
BEGIN
```

```
    RETURN 1;
END;
/
```

CREATE FUNCTION 语法支持 sql_macro 子句

```
CREATE or replace FUNCTION ora_func RETURN integer
SHARING = NONE AUTHID DEFINER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
DEFAULT COLLATION USING_NLS_COMP
DETERMINISTIC
PARALLEL_ENABLE ( PARTITION A BY RANGE ( B, C ) CLUSTER A BY ( E,F ) )
RESULT_CACHE RELIES_ON ( data_source1, data_source2)
AGGREGATE USING int
PIPELINED TABLE POLYMORPHIC USING pg_catalog.int4
SQL_MACRO
IS
BEGIN
    RETURN 1;
END;
/
```

ALTER FUNCTION 语法兼容

```
alter function public.test_func noneditionable;
alter function test_func compile;
alter function test_func compile debug;
alter function test_func compile debug sd = mv;
alter function test_func compile debug reuse settings;
```

CREATE PROCEDURE 语法支持 EDITIONABLE / NONEDITIONABLE

```
CREATE OR REPLACE EDITIONABLE PROCEDURE ora_procedure
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/

CREATE OR REPLACE NONEDITIONABLE PROCEDURE ora_procedure
IS
```

```
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

CREATE PROCEDURE 语法函数没有参数，函数名后面不带()

```
CREATE OR REPLACE EDITIONABLE PROCEDURE ora_procedure
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

CREATE PROCEDURE 语法中 END; 在psql中以 / 结束

```
CREATE OR REPLACE PROCEDURE ora_procedure
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

CREATE PROCEDURE 语法支持 sharing 子句

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = NONE
IS
        p integer := 20;
begin
        raise notice '%', p;
```

```
end;
/
```

CREATE PROCEDURE 语法支持 DEFAULT COLLATION 子句

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

CREATE PROCEDURE 语法支持 invoker_rights (AUTHID) 子句

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID DEFINER
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

CREATE PROCEDURE 语法支持 ACCESSIBLE BY 子句

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
```

```
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( B )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( A.B )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( FUNCTION A.B )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

```
CREATE OR REPLACE PROCEDURE ora_procedure
SHARING = METADATA
DEFAULT COLLATION USING_NLS_COMP
AUTHID CURRENT_USER
ACCESSIBLE BY ( FUNCTION A.B, PROCEDURE C.D, PACKAGE E, TRIGGER F, TYPE G )
IS
        p integer := 20;
begin
        raise notice '%', p;
end;
/
```

ALTER PROCEDURE 语法兼容

```
alter procedure test_proc editionable;
alter procedure public.test_proc noneditionable;
alter procedure test_proc compile;
alter procedure test_proc compile debug;
alter procedure test_proc compile debug sd = mv;
alter procedure test_proc compile debug reuse settings;
```

存储过程和函数没有参数，SELECT调用支持不带 ()

```
create or replace function f_noparentheses
return int is
begin
return 11;
end;
/
select f_noparentheses from dual;

create or replace procedure protest
as
begin
raise notice 'protest';
end;
/
CALL protest();
```

函数和存储过程相关的视图

```
相关视图位于文件 contrib/ivorysql_ora/src/sysview/sysview--1.0.sql
中，包括DBA_PROCEDURES，ALL_PROCEDURES，USER_PROCEDURES，DBA_SOURCE，ALL_SOURCE，USER_
```

```
SOURCE，DBA_ARGUMENTS，ALL_ARGUMENTS，USER_ARGUMENTS等。
```

支持—和/**/两种注释方法

pg_dump备份SQL脚本文件的备份格式时，在函数的定义最后增加一个斜线 /

.5. 5、内置数据类型与内置函数

内置数据类型

char
varchar
varchar2
number
binary_float
binary_double
date
timestamp
timestamp with time zone
timestamp with local time zone
interval year to month
interval day to second
raw
long

内置函数类型

sysdate
systimestamp
add_months
last_day
next_day
months_between
current_date
current_timestamp
new_time
tz_offset
trunc
instr
instrb
substr
substrb
trim
ltrim

rtrim
length
lengthb
rawtohex
replace
regexp_replace
regexp_substr
regexp_instr
regexp_like
to_number
to_char
to_date
to_timestamp
to_timestamp_tz
to_yminterval
to_dsinterval
numtodsinterval
numtoyminterval
localtimestamp
from_tz
sys_extract_utc
sessiontimezone
hextoraw
uid
USERENV
asciistr
to_multi_byte
to_single_byte
compose
decompose
sys_context

内置函数说明

sysdate 函数

功能：查看对应的日期与时间，测试用例如下： 查询当前系统的日期：

```
select sysdate() from dual;
  sysdate
------------
 2023-07-06
(1 row)
```

查询往前推1天的日期：

```
select sysdate()-1 from dual;
  ?column?
------------
 2023-07-05
(1 row)
```

systimestamp 函数

功能：返回本机数据库上当前系统日期和时间（包括微秒和时区），测试用例如下：
查询当前日期的日期和时间：

```
select systimestamp() from dual;
  systimestamp
-----------------------------------
 2023-07-06 10:18:31.674322 +08:00
(1 row)
```

add_months 函数

功能：函数将一个月数(n)添加一个日期，并返回相隔n月的同一天，支持参数：date, number；测试用例如下： 查询当前日期（七月六日）的下个月的同一天：

```
select add_months(sysdate(),1) from dual;
 add_months
------------
 2023-08-06
(1 row)
```

查询当前日期的上个月的同一天：

```
select add_months(sysdate(),-1) from dual;
 add_months
------------
 2023-06-06
(1 row)
```

last_day 函数

功能：返回指定日期所在月份的最后一天，支持参数：date,测试用例如下： 查询当天所在月份的最后一天：

```
select last_day(sysdate())from dual;
  last_day
------------
```

```
 2023-07-31
(1 row)
```

查询某一天所在月份的最后一天：

```
select last_day(to_date('2019-09-01'))from dual;
  last_day
------------
 2019-09-30
(1 row)
```

`next_day` 函数

功能：返回指定日期的下一个日期。支持参数:date, integer /date ,text,
说明：当函数中第二个参数传的星期数比现有星期数小时，会返回下一个星期的日期；当函数中第二个参数所传的日期比现有星期数大，会返回本周相应星期日期。测试用例如下： 查询当前日期的下一天：

```
select next_day(sysdate(),1) from dual;
 next_day
------------
 2023-07-07
(1 row)
```

查询当前日期的下个星期五：

```
select next_day(sysdate(),'FRIDAY') from dual;
  next_day
------------
 2023-07-07
(1 row)
```

`months_between` 函数

功能：返回日期类型的date1和date2之间相差的月份，支持参数：date,date，说明：如果date1晚于date2，返回正数；如果date1早于date2返回负数；如果date1和date2是某月里的同一天，返回结果为整数；如果不是同一天，会在每月31天的基础上返回带有小数部分的结果。测试用例如下：
查询不同月份同一天之间相差的月份：

```
select months_between(to_date('2023-07-06'),to_date('2023-08-06')) from dual;
 months_between
----------------
             -1
(1 row)
```

查询不同月份不同日期之间相差的月份：

```
select months_between(to_date('2023-07-06'),to_date('2023-08-05')) from dual;
   months_between
--------------------
 -0.967741935483871
(1 row)
```

current_date 函数

功能：返回当前时区的当前日期，测试用例如下： 查询当前时区的当前日期：

```
select current_date from dual;
 current_date
--------------
 2023-07-06
(1 row)
```

current_timestamp 函数

功能：返回当前时区的当前日期与当前时间，包含当前时区信息。支持参数：integer,
说明：返回的时间可调整精度。测试用例如下： 查询当前时区的当前日期与时间：

```
select current_timestamp from dual;
      current_timestamp
-----------------------------------
 2023-07-06 10:27:01.440600 +08:00
(1 row)
```

查询当前时区的当前日期与时间(精度调整为前三位小数)：

```
select current_timestamp(3) from dual;
      current_timestamp
-----------------------------------
 2023-07-06 10:27:14.182000 +08:00
(1 row)
```

new_time 函数

功能：返回某个时间在某时区所对应的在另一个时区的日期，支持参数：date, text, text ,测试用例如下：
返回当前日期在另一个时区对应的日期：

```
select sysdate() bj_time,new_time(sysdate(),'PDT','GMT')los_angles from dual;
  bj_time    | los_angles
------------+------------
 2023-07-06 | 2023-07-06
```

```
(1 row)
```

`tz_offset` 函数

功能：返回给定时区与标准时区的偏移量，支持参数：text,测试用例如下：
返回给定时区与标准时区偏移量：

```
select tz_offset('US/Eastern') from dual;
 tz_offset
-----------
 -04:00
(1 row)
```

`trunc` 函数

功能：可以截取日期，得到想要的数值，如年，月，日，时，分，支持参数：date/date,text,测试用例如下：
截取当前日期：

```
select trunc(sysdate()) from dual;
 trunc
------------
 2023-07-06
(1 row)
```

截取年，返回值只有年是正确的，月和日不是准确值：

```
select trunc(sysdate(),'yyyy') from dual;
   trunc
------------
 2023-01-01
(1 row)
```

截取月，返回值只有月是正确的，年和日不是准确值：

```
select trunc(sysdate(),'mm') from dual;
   trunc
------------
 2023-07-01
(1 row)
```

`instr` 函数

功能：字符串查找函数，用于判断源字符串中是否包含目标字符串，并返回匹配位置。支持参数形式如下：

- `instr(string, str)`

- `instr(string, str, start_position, nth_appearance)`

以下为测试用例：

默认情况下返回第一次匹配的位置：

```
SELECT INSTR('database administration', 'data') FROM DUAL;
 instr
-------
     1
(1 row)
```

指定起始位置和匹配序号：

```
SELECT INSTR('database administration', 'i', 1, 2) FROM DUAL;
 instr
-------
    15
(1 row)
```

支持从尾部开始的反向搜索：

```
SELECT INSTR('mississippi river', 's', -5, 2) FROM DUAL;
 instr
-------
     6
(1 row)
```

当未匹配到目标字符串时返回0：

```
SELECT INSTR('database administration', 'z') FROM DUAL;
 instr
-------
     0
(1 row)
```

`instr` 还能像 `like` 一样用于模糊查询：

```
select * from tableName where instr(name,'helloworld')>0;
```

函数实现说明：

- 查找一个字符串在另一个字符串中的位置，采用经典的暴力匹配算法。
- 复用了 `text_instring` 函数。
- 通过 `isByte` 参数判断当前数据库编码是否为多字节，并区分单字节与多字节处理逻辑。

- 根据 `position` 的正负确定正向或反向搜索，并设置循环的起止位置和步长。
- 在循环中逐字节移动并比较，与模式字符串匹配成功即返回相应位置。

`instrb` 函数

功能：字符串查找函数，返回字符串的位置，支持参数： varchar2, text, number DEFAULT 1, number DEFAULT 1,以下为测试用例： 返回CORPORATE FLOOR中默认第一次出现OR时字符串的位置：

```
SELECT INSTRB('CORPORATE FLOOR','OR') "Instring in bytes" FROM DUAL;
 Instring in bytes
-------------------
                  2
(1 row)
```

返回corporate floor中从第五个字符开始查询，第二次出现or时字符串的位置：

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes" FROM DUAL;
 Instring in bytes
-------------------
                 14
(1 row)
```

`substr` 函数

功能：截取字符串函数，以字符为单位截取，支持参数：text, integer, 测试用例如下：
截取’今天天气很好’中从第五个字符开始，往后的字符串：

```
SELECT SUBSTR('今天天气很好',5) "Substring with bytes" FROM DUAL;

 Substring with bytes
----------------------
 很好
(1 row)
```

`substrb` 函数

功能：截取字符串函数，以字节为单位截取，支持参数：varchar2, number/varchar2, number，number,测试用例如下： 截取’今天天气很好’中从第五个字节开始，往后的字符串：

```
SELECT SUBSTRB('今天天气很好',5) "Substring with bytes" FROM DUAL;
 Substring with bytes
----------------------
 天气很好
(1 row)
```

截取’今天天气很好’中从第五个字节开始，第八个字节结束的字符串：

```
SELECT SUBSTRB('今天天气很好',5, 8) "Substring with bytes" FROM DUAL;
 Substring with bytes
----------------------
   天气
(1 row)
```

`trim` 函数

功能：去除指定字符串的左右空格或对应数据，支持参数：varchar2 /varchar2，varchar2，测试用例如下：去除' aaa bbb ccc '的左右空格：

```
select trim('   aaa bbb ccc   ')trim from dual;
  trim
-------------
 aaa bbb ccc
(1 row)
```

去除’aaa bbb ccc’中的aaa：

```
select trim('aaa bbb ccc','aaa')trim from dual;
  trim
----------
  bbb ccc
(1 row)
```

`ltrim` 函数

功能：去除指定字符串的左侧空格或对应数据，支持参数：varchar2 /varchar2,varchar2,测试用例如下：去除' abcdefg '的左侧空格：

```
select ltrim('   abcdefg   ')ltrim from dual;
    ltrim
------------
 abcdefg
(1 row)
```

从’abcdefg’左侧开始遍历，一旦存在某字符出现在’fegab’中就去除，不存在则返回结果：

```
select ltrim('abcdefg','fegab')ltrim from dual;
 ltrim
-------
 cdefg
(1 row)
```

`rtrim` 函数

功能：去除指定字符串的右侧空格，测试用例如下： 去除' abcdefg '的右侧空格：

```
select rtrim('   abcdefg   ')rtrim from dual;
     rtrim
----------------
    abcdefg
(1 row)
```

从’abcdefg’右侧开始遍历，一旦存在某字符出现在’fegab’中就去除，不存在则返回结果：

```
select rtrim('abcdefg','fegab')rtrim from dual;
 rtrim
-------
 abcd
(1 row)
```

`length` 函数

功能：求取指定字符串字符的长度，支持参数：char/integer/varchar2测试用例如下： 查询223的字符长度：

```
select length(223) from dual;
 length
--------
      3
(1 row)
```

查询'223’的字符长度：

```
select length('223') from dual;
 length
--------
      3
(1 row)
```

查询’ivorysql数据库’的字符长度：

```
select length('ivorysql数据库') from dual;
 length
--------
     11
(1 row)
```

lengthb 函数

功能：求取指定字符串字节的长度，支持参数：char/bytea/varchar2测试用例如下：
查询’ivorysql’的字节长度:

```
select lengthb('ivorysq'::char) from dual;
 lengthb
---------
       1
(1 row)
```

查询'0x2C’的字节长度:

```
select lengthb('0x2C'::bytea) from dual;
 lengthb
---------
       4
(1 row)
```

查询’ivorysql数据库’的字节长度:

```
select lengthb('ivorysql数据库'::varchar2) from dual;
 lengthb
---------
      17
(1 row)
```

replace 函数

功能：替换指定字符串中的字符或删除字符，支持参数：text, text, text/varchar2, varchar2, varchar2 DEFAULT NULL::varchar2, 测试用例如下： 替换’jack and jue’中的’j’为’bl':

```
select replace('jack and jue','j','bl') from dual;
    replace
----------------
 black and blue
(1 row)
```

删除’jack and jue’中的’j':

```
select replace('jack and jue','j') from dual;
  replace
------------
 ack and ue
```

```
(1 row)
```

regexp_replace 函数

此函数为`replace`函数的扩展。功能：用于通过正则表达式来进行匹配替换。支持参数：text, text, text /text, text, text, integer/varchar2, varchar2/varchar2, varchar2 varchar2, 测试用例如下：
将匹配到的数字替换为*#：

```
select regexp_replace('01234abcd56789','[0-9]','*#')from  dual;
      regexp_replace
--------------------------
 *#*#*#*#*#abcd*#*#*#*#*#
(1 row)
```

从第二个数开始将匹配到的数字替换为*#：

```
select regexp_replace('01234abcd56789','[0-9]','*#',2)from  dual;
    regexp_replace
-------------------------
 0*#*#*#*#abcd*#*#*#*#*#
```

删除'01234abcd56789’中的'01'：

```
select regexp_replace('01234abcd56789','01')from dual;
 regexp_replace
----------------
 234abcd56789
(1 row)
```

用’xxx’替换01234abcd56789’中的012：

```
select regexp_replace('01234abcd56789','012','xxx')from dual;
 regexp_replace
----------------
 xxx34abcd56789
(1 row)
```

regexp_substr 函数

功能：拾取符合正则表达式描述的字符子串，支持参数：text, text,integer /text, text, integer, integer/ text, text, integer, integer, text /varchar2 ,varchar2,测试用例如下： 查询'012ab34’中从第一个数开始的012字串：

```
select regexp_substr('012ab34', '012',1) from dual;
 regexp_substr
---------------
```

```
 012
(1 row)
```

查询'012ab34’中从第一个数第一组开始的012字串：

```
select regexp_substr('012ab34', '012',1,1) from  dual;
 regexp_substr
---------------
 012
(1 row)
```

查询'012a012Ab34’中从第一个数第一组开始不区分大小写的012字串：

```
select regexp_substr('012a012Ab34', '012A',1,1,'i') from dual;
 regexp_substr
---------------
 012a
(1 row)
```

查询'012a012Ab34’中从第一个数第一组开始区分大小写的012字串：

```
select regexp_substr('012a012Ab34', '012A',1,1,'c') from dual;
 regexp_substr
---------------
 012A
(1 row)
```

查询’数据库’中 '数据’子串：

```
select regexp_substr('数据库', '数据') from dual;
 regexp_substr
---------------
 数据
(1 row)
```

`regexp_instr` 函数

功能：用于标定符合正则表达式的字符子串的开始位置，支持参数：text, text,integer /text, text, integer, integer/ text, text, integer, integer, text/text, text, integer, integer, text, integer/ varchar2, varchar2,测试用例如下： 查询’abcaBcabc’中从第一个字符开始，出现abc子串的位置：

```
SELECT regexp_instr('abcaBcabc', 'abc', 1);
 regexp_instr
--------------
```

```
 1
(1 row)
```

查询’abcaBcabc’中从第一个字符开始，第三次出现abc子串的位置：

```
SELECT regexp_instr('abcaBcabc', 'abc', 1, 3);
 regexp_instr
--------------
 7
(1 row)
```

查询’abcabcabc’中从第一个字符开始，第二次出现abc子串后发生的位置：

```
SELECT regexp_instr('abcaBcabc', 'abc', 1, 2,1);
 regexp_instr
--------------
 7
(1 row)
```

查询’abcaBcabc’中从第一个字符开始，第一次出现abc子串后发生的位置（区分大小写）：

```
SELECT regexp_instr('abcaBcabc', 'abc',1,2,1,'c');
 regexp_instr
--------------
 7
(1 row)
```

查询’数据库’中’库’出现的位置：

```
SELECT regexp_instr('数据库', '库');
 regexp_instr
--------------
            3
(1 row)
```

`regexp_like` 函数

功能：与like类似，用于模糊查询。支持参数：varchar2, varchar2 /varchar2, varchar2 varchar2,
首先创建一个regexp_like表用于测试用例查询：

```
create table t_regexp_like
(
  id varchar(4),
```

```
  value varchar(10)

);
insert into t_regexp_like values ('1','1234560');
insert into t_regexp_like values ('2','1234560');
insert into t_regexp_like values ('3','1b3b560');
insert into t_regexp_like values ('4','abc');
insert into t_regexp_like values ('5','abcde');
insert into t_regexp_like values ('6','ADREasx');
insert into t_regexp_like values ('7','123  45');
insert into t_regexp_like values ('8','adc  de');
insert into t_regexp_like values ('9','adc,.de');
insert into t_regexp_like values ('10','abcbvbnb');
insert into t_regexp_like values ('11','11114560');
```

测试用例如下： 查询t_regexp_like表中带有abc的列：

```
select * from t_regexp_like where regexp_like(value,'abc');
 id |  value
----+----------
 4  | abc
 5  | abcde
 10 | abcbvbnb
(3 rows)
```

查询t_regexp_like表中带有ABC的列(不区分大小写)：

```
select * from t_regexp_like where regexp_like(value,'ABC','i');
 id |  value
----+----------
 4  | abc
 5  | abcde
 10 | abcbvbnb
(3 rows)
```

`to_number` 函数

功能：是将一些处理过的按一定格式编排过的字符串变回数值型的格式，支持参数：text/text,text测试用例如下： 将字符串'-34,338,492’ 转换为数值型格式：

```
SELECT to_number('34,338,492', '99,999,999') from dual;
 to_number
-----------
```

```
 -34338492
(1 row)
```

将字符串'5.01-'转换为数值型格式：

```
SELECT to_number('5.01-', '9.99S');


 to_number
-----------
 -5.01
(1 row)
```

`to_char` 函数

功能：将数字或日期转换为字符类型，支持参数：date/date,text/timestamp/timestamp,text测试用例如下：
将当前系统日期转换为字符格式：

```
select to_char(sysdate()) from dual;
  to_char
------------
 2023-07-10
(1 row)
```

将当前系统日期转换为月份/日期/年字符格式：

```
select to_char(sysdate(),'mm/dd/yyyy') from dual;
  to_char
------------
 07/10/2023
(1 row)
```

将当前日期的timestamp格式转换为字符格式：

```
SELECT to_char(sysdate()::timestamp);
         to_char
----------------------------
 2023-07-10 09:46:44.000000
```

将当前日期的timestamp格式转换为月份/日期/年字符格式：

```
SELECT to_char(sysdate()::timestamp,'MM-YYYY-DD');
  to_char
------------
```

```
 07-2023-10
(1 row)
```

`to_date` 函数

功能：将字符类型转换为日期类型，支持参数：text/text,text测试用例如下：
将'2023/07/06’ 转换为日期类型：

```
select to_date('20230706') from dual;
  to_date
------------
 2023-07-06
(1 row)
```

将'-44-02-01’ 转换为日期类型：

```
SELECT to_date('-44,0201','YYYY-MM-DD');
 to_date
------------
 0044-02-01
(1 row)
```

`to_timestamp` 函数

功能：可以存储年、月、日、小时、分钟、秒，同时还可以存储秒的小数部分。支持参数：text/text,text测试用例如下： 查询'2018-11-02 12:34:56.025’ 以日期形式输出：

```
SELECT to_timestamp('20181102.12.34.56.025');
        to_timestamp
----------------------------
 2018-11-02 12:34:56.025000
(1 row)
```

查询’2011,12,18 11:38’ 以日期形式输出：

```
SELECT to_timestamp('2011,12,18 11:38 ', 'YYYY-MM-DD HH24:MI:SS');
       to_timestamp
----------------------------
 2011-12-18 11:38:00.000000
(1 row)
```

`to_timestamp_tz` 函数

功能：根据时间查询，时间字符串有T,Z并有毫秒，时区。测试用例如下： 查询'2016-10-9 14:10:10.123000’ 以日期形式输出：

```
 SELECT to_timestamp_tz('2016-10-9 14:10:10.123000') FROM DUAL;
          to_timestamp_tz
------------------------------------
 2016-10-09 14:10:10.123000 +08:00
(1 row)
```

查询'10-9-2016 14:10:10.123000 +8:30’ 以日期形式输出：

```
 SELECT to_timestamp_tz('10-9-2016 14:10:10.123000 +8:30', 'DD-MM-YYYY HH24:MI:SS.FF
TZH:TZM') FROM DUAL;
          to_timestamp_tz
------------------------------------
 2016-09-10 13:40:10.123000 +08:00
(1 row)
```

`to_yminterval` 函数

功能：将一个字符串类型转化为年和月的时间差类型，支持参数:text, 测试用例如下：
查询'20110101’ 以后两个年零八个月后的日期：

```
select to_date('20110101','yyyymmdd')+to_yminterval('02-08') from dual;
  ?column?
------------
 2013-09-01
(1 row)
```

`to_dsinterval` 函数

功能：将一个日期加上一定的小时或者天数变成另外一个日期,支持参数:text,测试用例如下：
查询当前系统时间加上9个半小时后的日期（当前为2023-07-06，18：00）：

```
select sysdate()+to_dsinterval('0 09:30:00')as newdate from dual;
 newdate
------------
 2023-07-07
(1 row)
```

`numtodsinterval` 函数

功能：将数字转换成时间间隔类型的数据。支持参数:double precision, text测试用例如下：
转换100.00个小时为时间间隔类型数据：

```
SELECT NUMTODSINTERVAL(100.00, 'hour');
        numtodsinterval
--------------------------------
```

```
 +000000004 04:00:00.000000000
(1 row)
```

转换100分钟为时间间隔类型数据：

```
SELECT NUMTODSINTERVAL(100, 'minute');
      numtodsinterval
-------------------------------
 +000000000 01:40:00.000000000
(1 row)
```

`numtoyminterval` 函数

功能：将数字转换成日期间隔类型的数据。 支持参数：double precision,text,测试用例如下：
转换1.00，year为日期间隔：

```
SELECT NUMTOYMINTERVAL(1.00,'year');
 numtoyminterval
-----------------
 +000000001-00
(1 row)
```

转换1，mouth为日期间隔：

```
SELECT NUMTOYMINTERVAL(1,'month');
 numtoyminterval
-----------------
 +000000000-01
(1 row)
```

`localtimestamp` 函数

功能：返回会话中的日期和时间，支持参数：integer, 函数中增加参数为精度，测试用例如下：
返回当前会话中的日期和时间：

```
select localtimestamp from dual;
       localtimestamp
----------------------------
 2023-07-07 09:18:15.896472
(1 row)
```

返回当前会话中的日期和时间（精度为1）：

```
select localtimestamp(1) from dual;
```

```
     localtimestamp
----------------------------
 2023-07-07 09:18:16.100000
(1 row)
```

`from_tz` 函数

功能：将时间从一个时区转换为另一个时区，支持参数;timestamp, text ,测试用例如下： 将'2000-03-28 08:00:00', '3:00’ 转换为当前时区：

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00') FROM DUAL;
              from_tz
-----------------------------------
 2000-03-28 13:00:00.000000 +08:00
(1 row)
```

`sys_extract_utc` 函数

功能：将一个timestamptz转换为UTC时区时间。支持参数:timestamp with time zone 测试用例如下：查询转换timestamp '2000-03-28 11:30:00.00 -8:00’ 为UTC时区后的时间：

```
select sys_extract_utc(timestamp '2000-03-28 11:30:00.00 -8:00') from dual;
      sys_extract_utc
----------------------------
 2000-03-28 19:30:00.000000
(1 row)
```

`sessiontimezone` 函数

功能：查看时区详细信息，测试用例如下： 查看当前时区的详细信息：

```
select sessiontimezone() from dual;
 sessiontimezone
-----------------
 Asia/Shanghai
(1 row)
```

修改timezone后，查看时区详细信息：

```
set timezone = 'Asia/Hong_Kong';
SET
select sessiontimezone() from dual;
 sessiontimezone
-----------------
 Asia/Hong_Kong
```

```
(1 row)
```

`hextoraw` 函数

功能：将字符串表示的二进制数值转换为一个raw数值。支持参数：text,测试用例如下：
将字符串’abcdef’转换为raw数值：

```
select hextoraw('abcdef')from dual;
 hextoraw
----------
 \xabcdef
(1 row)
```

`uid` 函数

功能：获取数据库的实例名。测试用例如下： 获取当前数据库的实例名：

```
select uid() from dual;
 uid
-----
  10
(1 row)
```

`USERENV` 函数

功能：返回当前用户环境的信息，测试用例如下： 查看当前用户是否是dba，如果是返回true：

```
select userenv('isdba')from dual;
 get_isdba
-----------
 TRUE
(1 row)
```

查看会话标志：

```
select userenv('sessionid')from dual;
 get_sessionid
---------------
 1
(1 row)
```

`ASCIISTR` 函数

功能：传入字符串，返回对应的ASCII字符，测试用例如下： 只有ASCII字符：

```
 select asciistr('Hello, World!') from dual;
   asciistr
---------------
 Hello, World!
(1 row)
```

非ASCII字符:

```
 select asciistr('你好') from dual;
  asciistr
------------
 \4F60\597D
```

同时包含ASCII字符和非ASCII字符:

```
 select asciistr('ABÄCDE') from dual;
  asciistr
------------
 AB\00C4CDE
(1 row)
```

`TO_MULTI_BYTE` 函数

功能：将字符串中的半角字符转换为全角字符： 输入半角字符，转换为全角字符:

```
select to_multi_byte('1.2'::text) ;
 to_multi_byte
---------------
 １．２
```

`TO_SINGLE_BYTE` 函数

功能：将字符串中的半角字符转换为全角字符 输入全角字符，转换为半角字符:

```
select to_single_byte('１．２');
 to_single_byte
----------------
 1.2
```

`COMPOSE` 函数

功能：将基本字符和组合标记组合一个复合Unicode字符: 输入基本字符a和组合标记768, 返回法语à

```
select compose('a'||chr(768)) from dual;
```

```
 compose
---------
 à
(1 row)
```

DECOMPOSE 函数

功能：将复合Unicode字符（如带有重音或特殊符号的字符）分解为其基本字符和组合标记
输入法语é,返回基本字符e和组合标记301：

```
select asciistr(decompose('é')) from dual;
 asciistr
----------
 e\0301
```

SYS_CONTEXT 函数

功能：返回当前时刻与给定上下文关联的参数的值，可以在SQL和PLSQL语言中使用。

```
ivorysql=# select sys_context('USERENV','DB_NAME');
 sys_context
-------------
 ivorysql
(1 row)
```

.6. 6、端口与IP

目的

- 为了将Oracle端口、IP与PG的端口IP进行区分。现需要增加对ORAPORT和ORAHOST的处理;

功能

- 新增ivoryhost：需要在连接时新增参数ivoryhost即可指定，其功能类似 host;
- 新增ivoryport：相较于host，port的功能相对复杂一些。其中涉及到可以在configure阶段配置，连接阶段指定端口;

测试用例：

```
./configure --with-oraport=5555
./initdb ....
./pg_ctl -D ../data start

./pg_ctl -o "-p 5433 -o 1522" -D ../data
```

.7. 7、XML函数

目的

在Oracle中，常会出现带有XML函数的SQL代码，IvorySQL在PostgreSQL的基础上，实现与Oracle XML函数的高度兼容，确保了从Oracle迁移到IvorySQL后的数据格式和结构的一致性。这种兼容性意味着用户无需对现有的XML处理逻辑进行大规模修改，从而保证了数据的完整性和准确性。此外，IvorySQL的这种跨平台的兼容特性，也降低了因格式差异带来的额外用户维护和升级成本，使得数据处理和管理更加高效、可靠和灵活。

> XML（eXtended Markup Language可扩展标记语言）是一种基于文本的，用于结构化任何可标记文档的格式语言。它是一种轻便的，可扩展的，标准的且简单易懂的保存数据的语言。

实现原理

IvorySQL在实现与Oracle 12c中11个常用XML SQL函数的兼容性时，与PostgreSQL保持了一致，底层处理函数同样采用了libxml2库提供的接口。这些XML函数作为ivorysql_ora插件的一个子插件提供，确保了与PostgreSQL数据库在XML处理方面的兼容性和一致性。

由于Oracle的xml函数要求某些参数类型是 XMLType，比如下面这个existsnode()函数：

原型： EXISTSNODE(XMLType_instance, XPath_string [, namespace_string])

示例： SELECT existsnode(XMLType('<a>d'), '/a') from dual;

因此为了兼容，添加了一个 XMLType 数据类型，作用是把用户提供的字符串转换成内部的xmltype类型，让用户的SQL语句从Oracle迁移到IvorySQL不用做改动

另外，为了避免与PostgreSQL中已有的同名关键字“extract”产生混淆，IvorySQL将原有的同名关键字重命名为“PGEXTRACT”，以确保函数调用的清晰性和准确性。

在实现这11个Oracle兼容的XML函数时，IvorySQL采用了两种不同的实现方式。其中，除了UPDATEXML函数外，其他10个函数均通过SQL函数的方式进行实现。由于UPDATEXML函数的参数数量是不确定的，因此采用了表达式方式来实现，这要求为其编写专门的语法分析代码和执行器代码，以确保其功能的正确性和灵活性。

兼容函数如下

序号	函数名	功能简介
1	extract(XML)	该函数用于返回XML节点路径下的相应内容， 其中参数XMLType_instance用于指定XMLType实例，Xpath_string用于指定XML节点路径
2	extractvalue	该函数提供对XML内容的检索功能，extractvalue只能返回一个节点的一个值。
3	existsnode	该函数用于检查XML内容是否符合指定的路径表达式。
4	deletexml	该函数用于删除指定路径的XML节点

5	appendchildxml	该函数用于在XML对象中插入子节点。 它接受一个XML对象和一个XML片段作为参数，并将XML片段作为子节点插入到XML对象中
6	updatexml	该函数用于更新特定XML相应的节点路径的内容
7	insertxmlbefore	该函数用于在XML中的指定路径前插入子节点
8	insertxmlafter	该函数用于在XML中的指定路径后插入子节点
9	insertchildxml	该函数用于向指定的XML路径中插入子节点
10	insertchildxmlbefore	该函数用于向指定的XML路径之前插入子节点
11	insertchildxmlafter	该函数用于向指定的XML路径之后插入子节点
12	xmlisvalid	该函数用于检查XML格式是否正确

XML函数使用示例

准备表与数据

```
ivorysql=# set ivorysql.compatible_mode to oracle;
SET
ivorysql=# create table inaf(a int, b xmltype);
CREATE TABLE
ivorysql=# insert into inaf values(1,xmltype('<a><b>100</b></a>'));
INSERT 0 1
ivorysql=# insert into inaf values(2, '');
INSERT 0 1
ivorysql=# select * from inaf;
 a |         b
---+-------------------
 1 | <a><b>100</b></a>
 2 |
(2 rows)
ivorysql=# create table xmltest(id int, data XMLType);
CREATE TABLE
ivorysql=# insert into xmltest values(1, '<value>one</value>');
INSERT 0 1
ivorysql=# insert into xmltest values(2, '<value>two</value>');
INSERT 0 1
ivorysql=# select * from xmltest;
 id |        data
----+--------------------
```

```
  1 | <value>one</value>
  2 | <value>two</value>
(2 rows)
ivorysql=# create table xmlnstest(id int, data xmltype);
CREATE TABLE
ivorysql=# INSERT INTO xmlnstest VALUES(1, xmltype('<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:typ="http://www.def.com"
xmlns:web="http://www.abc.com"><soapenv:Body><web:BBB><typ:EEE>41</typ:EEE><typ:FFF>42
</typ:FFF></web:BBB></soapenv:Body></soapenv:Envelope>'));
INSERT 0 1
```

extract(XML)

```
ivorysql# SELECT extract(XMLType('<AA><ID>1</ID></AA>'), '/AA/ID') from dual;
  extract
------------
 <ID>1</ID>
(1 row)
```

extractvalue

```
ivorysql# SELECT extractvalue(XMLType('<a><b>100</b></a>'),'/a/b') from dual;
 extractvalue
--------------
 100
(1 row)
```

existsnode

```
ivorysql=# SELECT existsnode(XMLType('<a><b>d</b></a>'), '/a/b') from dual;
 existsnode
------------
          1
(1 row)
```

deletexml

```
ivorysql=# SELECT
deletexml(XMLType('<test><value>oldnode</value><value>oldnode</value></test>'),
'/test/value') from dual;
 deletexml
-----------
```

```
 <test/>
(1 row)
```

appendchildxml

```
ivorysql=# SELECT
appendchildxml(XMLType('<test><value></value><value></value></test>'),  '/test/value',
XMLTYPE('<name>newnode</name>')) from dual;
      appendchildxml
--------------------------
 <test>                  +
   <value>               +
     <name>newnode</name>+
   </value>              +
   <value>               +
     <name>newnode</name>+
   </value>              +
 </test>
(1 row)
```

updatexml

```
ivorysql=# SELECT updatexml(xmltype('<value>one</value>'), '/value',
xmltype('<newvalue>1111</newvalue>')) FROM dual;
         updatexml
---------------------------
 <newvalue>1111</newvalue>
(1 row)
```

insertxmlbefore

```
ivorysql=# SELECT insertxmlbefore(XMLType('<a>222<b>100</b><b>200</b></a>'), '/a/b',
XMLTYPE('<c>88</c>')) from dual;
                 insertxmlbefore
--------------------------------------------------
 <a>222<c>88</c><b>100</b><c>88</c><b>200</b></a>
(1 row)
```

insertxmlafter

```
ivorysql=# SELECT
insertxmlafter(XMLType('<a><b>100</b></a>'),'/a/b',XMLType('<c>88</c>')) from dual;
 insertxmlafter
```

```
----------------
 <a>           +
   <b>100</b>  +
   <c>88</c>   +
 </a>
(1 row)
```

insertchildxml

```
ivorysql=# SELECT insertchildxml(XMLType('<a>one<b></b>three<b></b></a>'), '//b',
'name', XMLTYPE('<name>newnode</name>')) from dual;
                            insertchildxml
----------------------------------------------------------------------
 <a>one<b><name>newnode</name></b>three<b><name>newnode</name></b></a>
(1 row)
```

insertchildxmlbefore

```
ivorysql=# SELECT insertchildxmlbefore(XMLType('<a><b>100</b></a>'), '/a', 'b',
XMLType('<c>88</c>')) from dual;
 insertchildxmlbefore
----------------------
 <a>                 +
   <c>88</c>         +
   <b>100</b>        +
 </a>
(1 row)
```

insertchildxmlafter

```
ivorysql=# SELECT insertchildxmlafter(XMLType('<a><b>100</b></a>'), '/a', 'b',
XMLType('<c>88</c>')) from dual;
 insertchildxmlafter
---------------------
 <a>                +
   <b>100</b>       +
   <c>88</c>        +
 </a>
(1 row)
```

xmlisvalid

```
ivorysql=# SELECT xmlisvalid(XMLTYPE('<a>'));
xmlisvalid
------------
f
(1 row)

ivorysql=# SELECT xmlisvalid(XMLTYPE('<a/>'));
xmlisvalid
------------
t
(1 row)
```

.8. 8、sequence

目的

- 本文档是IvorySQL兼容Oracle sequence文档，目的是可以在IvorySQL中使用Oracle的sequence语句。

功能说明

- 序列是一个数据库对象，与表和视图类似，它表示可以由全局数据库命名空间中的任何表和视图使用的整数序列。可以使用NEXTVAL和CURRVAL访问序列值。序列可以是升序或降序

测试用例

```
ivorysql=# CREATE sequence seq;
CREATE SEQUENCE
ivorysql=# SELECT seq.NEXTVAL FROM DUAL;
 nextval
---------
       1
(1 row)
ivorysql=# ALTER sequence seq restart start with 10;
ALTER SEQUENCE
ivorysql=# SELECT seq.NEXTVAL FROM DUAL;
 nextval
---------
       10
(1 row)
ivorysql=# DROP SEQUENCE seq;
DROP SEQUENCE
```

.9. 9、包

目的

IvorySQL提供了兼容Oracle自定义包的功能。包（package）是一个封装的相关程序对象集合，这些对象存储在数据库中。程序对象包括过程、函数、变量、常量、游标和异常。

本文档旨在为使用人员提供一个深入了解自定义包实现的过程。

功能说明

IvorySQL提供了兼容Oracle自定义包的功能，包括包和包体的创建、修改和描述，在PostgreSQL交互终端（psql）中增加了与包相关的命令，新增了 \dk 命令。

创建包规范

使用CREATE OR REPLACE PACKAGE语句创建或替换一个包的规范。包是相关的存储过程、函数、和其他程序对象的集合，在数据库中作为一个单元存储。包规范声明这些对象，在包体中定义这些对象。

在自己的模式下创建或替换一个包规范，必须有CREATE PROCEDURE系统权限，如果在其他用户的模式下创建或替换一个包，则需要有CREATE ANY PROCEDURE系统权限。

创建包体

使用CREATE OR REPLACE PACKAGE BODY语句创建或替换一个包体。创建包体需要有与创建包规范一样的权限，要求包体与包规范在同一个模式下，且包规范必须存在，该语句定义包规范中声明的对象。

当包规范中有cursor或子过程时，那么必须拥有一个包体去定义它，否则包体是可选的。

更新包

使用ALTER PACKAGE语句来改变包的属性。执行ALTER PACKAGE语句的权限要求：必须是包的所有者，或者具有ALTER ANY PROCEDURE权限才能修改其他用户的包。

删除包和包体

DROP PACKAGE语句删除数据库中的一个存储包。这个语句会同时删除包体和包规范。DROP PACKAGE BODY则只删除包体。

不能使用此语句删除包中的一个单独对象。权限要求：包必须在用户模式下，或用户有DROP ANY PROCEDURE系统权限。

丢弃包

DISCARD PACKAGE功能是为了兼容PostgreSQL的DISCARD功能而做的。

在psql中使用 \dk[+] 查看包与包体定义

psql中支持 \dk[+] 命令查看包与包体的定义信息。

命令	描述
\dk[+]	列出当前能看到的包信息
\dk[+] xxx	列出xxx包的包规范和包体内容

测试用例

创建包规范

```
ivorysql=# create or replace package pkg is
ivorysql-#   var1 integer;
ivorysql-#   var2 integer;
ivorysql-#   function test_f(id integer) return integer;
ivorysql-#   procedure test_p(id integer);
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE
```

创建包体

```
ivorysql=# create or replace package body pkg is
ivorysql-#   var3 integer;
ivorysql-#   function test_f(id integer) return integer is
ivorysql-#   begin
ivorysql-#     dbms_output.put_line('pkg test_f');
ivorysql-#     return id;
ivorysql-#   end;
ivorysql-#   procedure test_p(id integer) is
ivorysql-#   begin
ivorysql-#     dbms_output.put_line('pkg proc');
ivorysql-#   end;
ivorysql-#   --privite function
ivorysql-#   function test_piv1(id integer) return integer is
ivorysql-#   begin
ivorysql-#      return id;
ivorysql-#   end;
ivorysql-#   --privite procedure
ivorysql-#   procedure test_piv2(id integer) is
ivorysql-#   begin
ivorysql-#     dbms_output.put_line('privite proc');
ivorysql-#   end;
ivorysql-# begin
ivorysql-#    var1 := 1;
ivorysql-#    var2 := 2;
ivorysql-#    var3 := 4;
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE BODY
```

更新包

```
ivorysql=# alter package pkg noneditionable;
ALTER PACKAGE
```

删除包和包体

```
ivorysql=# Drop package pkg;
DROP PACKAGE

ivorysql=# Drop package body pkg;
DROP PACKAGE BODY
```

丢弃包

```
ivorysql=# discard package;
DISCARD PACKAGES
```

在psql中使用 \dk[+] 查看包与包体定义

```
ivorysql=# \dk
       List of packages
 Schema |   Name   |  Owner
--------+----------+----------
 public | pkg      | ivorysql
 public | test_pkg | ivorysql
(2 rows)

ivorysql=# \dk pkg
     List of packages
 Schema | Name |  Owner
--------+------+----------
 public | pkg  | ivorysql
(1 row)

ivorysql=# \dk pkg1
Did not find any package named "pkg1".

ivorysql=# \dk+

List of packages
 Schema |   Name   |  Owner   | Security | Editionable | Use Collation |
Specification                 |                          Package Body
```

```
--------+----------+----------+----------+-------------+---------------+
-----------------------------------------------+
-----------------------------------------------------
---------
 public | pkg      | ivorysql | definer  | Editionable | default       | var1 integer;
+|
        |          |          |          |             |               | var2 integer;
+|
        |          |          |          |             |               | function
test_f(id integer) return integer;  +|
        |          |          |          |             |               | procedure
test_p(id integer);               +|
        |          |          |          |             |               | end
|
 public | test_pkg | ivorysql | definer  | Editionable | default       | var1 integer;
+| FUNCTION test_f(id integer) RETURN integer IS
        +
        |          |          |          |             |               |   FUNCTION
test_f(id integer) RETURN integer;+|   BEGIN
        +
        |          |          |          |             |               | end
|     dbms_output.put_line('invoke function test_pkg.t
est_f');+
        |          |          |          |             |               |
| RETURN 23;
        +
        |          |          |          |             |               |
|   end;
        +
        |          |          |          |             |               |
| BEGIN
        +
        |          |          |          |             |               |
|   var1 := 23;
        +
        |          |          |          |             |               |
| end
(2 rows)
```

.10. 10、不可见列

目的

本功能的引入是为了兼容Oracle的不可见列功能，将使数据库设计更加简洁，数据管理的灵活性得到提高，用户可以更好地控制列的可见性。

在应用迁移过程中，不可见列也很有用。使新列不可见意味着它们不会被任何现有应用程序看到，但仍然可以被任何新应用程序引用，从而使应用程序的在线迁移更加简单。

功能说明

不可见列功能使用户能够在执行诸如SELECT *
等操作时隐藏特定列。可以使用不可见列对表进行更改，而不会干扰使用该表的应用程序。对表的任何通用访问都不会显示不可见的列，必须通过列名显式引用不可见列，查询和其他操作才能访问不可见列。

在Oracle模式下，psql的扩展描述命令（\d+）增加了对不可见列的支持。

测试用例

创建包含不可见列的表

```
ivorysql=# CREATE TABLE mytable (a INT, b INT INVISIBLE, c INT);
CREATE TABLE
```

向不可见列插入值

只有在INSERT语句中显式指定不可见列时，才能向不可见列插入值。向包含不可见列的表插入值时不能省略列列表，否则会报错。

```
ivorysql=# INSERT INTO mytable (a, b, c) VALUES (1,2,3);
INSERT 0 1

ivorysql=# INSERT INTO mytable VALUES (4,5,6);
ERROR:  INSERT has more expressions than target columns
LINE 1: INSERT INTO mytable  VALUES (4,5,6);
```

显示不可见列的输出

使用SELECT语句查询时，如果要显示不可见列信息，必须显式指定该不可见列，否则将不显示该不可见列。

```
ivorysql=# select * from mytable;
 a | c
---+---
 1 | 3
(1 row)

ivorysql=# select a,b,c from mytable;
 a | b | c
---+---+---
 1 | 2 | 3
(1 row)
```

在Oracle模式下psql扩展描述命令（\d+）对不可见列的支持

```
ivorysql=# \d+ mytable
                                                    Table "public.mytable"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
 b      | pg_catalog.int4 |           |          |         | invisible | plain   |
|              |
 c      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Access method: heap
```

限制

- 目前不支持modify column，将在未来版本实现；
- 在Oracle中，不可见列和列顺序有特别的注意事项。当一个表包含一个或多个不可见列时，这些不可见列不会包含在表的列顺序中。目前我们尚未对这块做处理。

.11. 11、RowID

目的

IvorySQL提供了兼容Oracle RowID的功能。RowID是一种伪列，在创建表时由数据库自动生成，对于数据库中的每一行，RowID伪列返回该行的地址。

RowID 应当具备以下特性：

1. 逻辑地标识每一行，且值唯一
2. 可以通过ROWID快速查询和修改表的其他列，自身不能被插入和修改
3. 用户可以控制是否开启此功能

功能开启

IvorySQL提供了多种方式来开启RowID功能。

通过GUC参数开启

在IvorySQL 的兼容Oracle 模式下，可以通过 set ivorysql.default_with_rowids to on 来开启RowID，这个参数默认值为off。打开后创建的表，默认就带有了RowID列，可以通过 \d+ 表名 来查看。

```
ivorysql=# show ivorysql.default_with_rowids;
 ivorysql.default_with_rowids
------------------------------
 off
```

```
(1 row)
```

```
ivorysql=# create table t(a int);
CREATE TABLE
ivorysql=# \d+ t
                                                    Table "public.t"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Access method: heap
```

通过建表语句中增加 WITH ROWID 选项开启

用户可以选择在需要的表上带有这个选项，没有WITH ROWID选项，将会创建一个普通的表。

```
ivorysql=# create table t2(a int) with rowid;
CREATE TABLE
ivorysql=# \d+ t2
                                                    Table "public.t2"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
Compression | Stats target | Description
--------+-----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Indexes:
    "t2_16432_rowid_idx" btree (rowid)
Access method: heap
Has ROWID: yes
```

通过对现有表ALTER TABLE … SET WITH ROWID开启

这种方式允许当一个普通表需要使用 ROWID 时，通过ATLER 命令去添加ROWID 列。 也可以通过ALTER TABLE … SET WITHOUT ROWID命令去除RowID。

```
ivorysql=# create table t3(a int);
CREATE TABLE
ivorysql=# alter table t3 set with rowid;
ALTER TABLE
ivorysql=# \d+ t3;
                                                    Table "public.t3"
 Column |      Type       | Collation | Nullable | Default | Invisible | Storage |
```

```
Compression | Stats target | Description
--------+----------------+-----------+----------+---------+-----------+---------+
-------------+--------------+-------------
 a      | pg_catalog.int4 |           |          |         |           | plain   |
|              |
Access method: heap
Has ROWID: yes
```

.12. 12、OUT 参数

目的

IvorySQL提供了兼容Oracle的out参数功能，包括含有out参数的函数与过程、匿名块支持out参数、libpq支持out参数。

本文档旨在为使用人员介绍out参数的功能。

功能说明

IvorySQL提供了兼容Oracle的out参数功能，包括如下内容。

含有out参数的函数

创建语法如下：

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...]
] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...
```

其中argmode可以是IN，OUT，INOUT（IN
OUT），VARIADIC。如果不指定默认是IN。argmode也可以写在argname的后面。

不同于原生PG，兼容Oracle的out参数和返回值数据类型没有关联。IN
OUT模式和OUT模式都不能有默认值。如果有out参数，则函数体中必须有RETURN语句。

匿名块支持out参数

支持冒号占位符形式的绑定变量，例如: 1，:name。

新增DO+USING语法： DO [LANGUAGE lang_name] code [USING IN | OUT | IN OUT, …]

支持在libpq中按位置和按参数名字绑定变量，提供系统函数get_parameter_descr，该函数根据SQL语句，
返回变量名字与位置的关系。

libpq中调用含out参数的函数

libpq接口端提供准备、绑定、执行函数，这些函数与OCI相应函数类似。

使用流程为： 使用IvyHandleAlloc分配语句句柄和错误句柄。 调用IvyStmtPrepare准备语句。
调用IvyBindByPos或IvyBindByName 绑定参数。 调用IvyStmtExecute 执行，可重复执行。
调用IvyFreeHandle 释放语句句柄和错误句柄。

另外还实现了Ivyconnectdb，Ivystatus，Ivyexec，IvyresultStatus，IvyCreatePreparedStatement，IvybindOutParameterByPos，IvyexecPreparedStatement，IvyexecPreparedStatement2，Ivynfields，Ivyntuples，Ivyclear等一系列接口函数。

在源代码中 src/interfaces/libpq/ivytest 目录里可以找到示例程序。

测试用例

含有out参数的函数

1. out参数和返回值数据类型没有关联

```
ivorysql=# create or replace function test_return_out(id integer,price out
integer,name out varchar) return varchar
ivorysql-# as
ivorysql-# begin
ivorysql-#   price := 20000;
ivorysql-#   name := 'test a char out';
ivorysql-#   return 'welcome to QingDao';
ivorysql-# end;
ivorysql-# /
CREATE FUNCTION
```

2. IN OUT模式和OUT模式都不能有默认值

```
ivorysql=# create or replace function test_return_inout(id integer,price in out
integer default 100,name out varchar) return varchar
```

```
ivorysql-# as
ivorysql-# begin
ivorysql-#   price := 20000 + price;
ivorysql-#   name := 'this is a test';
ivorysql-# return 'welcome to QingDao';
ivorysql-# end;
ivorysql-# /
ERROR:  IN OUT formal parameters may have no default expressions
```

3. 如果有out参数，并且函数返回类型不是void，则函数体中必须有RETURN语句

```
```
--if function's return type is non-void, the function body must has RETURN
statement
--if there is no RETURN statement, the function can be created, but when it is
called,
--an error is raised
ivorysql=# create or replace function f2(id integer,price out integer) return
varchar
ivorysql-# as
ivorysql-# begin
ivorysql-#   price := 2;
ivorysql-# end;
ivorysql-# /
CREATE FUNCTION
ivorysql=# declare
ivorysql-#   a varchar(20);
ivorysql-#   b int;
ivorysql-# begin
ivorysql-#   a := f2(1, b);
ivorysql-# end;
ivorysql-# /
ERROR: Function returned without value
CONTEXT: PL/iSQL function f2(pg_catalog.int4,pg_catalog.int4) line 0 at RETURN
PL/iSQL function inline_code_block line 5 at assignment
```
```

匿名块支持out参数

1. 支持冒号占位符形式的绑定变量，新增DO+USING语法

```
```
ivorysql=# do $$
```
```

```
ivorysql$# declare
ivorysql$#   a int;
ivorysql$# begin
ivorysql$#   :x := 1;
ivorysql$#   :y := 2;
ivorysql$# end; $$ using out, out;
 $1 | $2
----+----
  1 |  2
(1 row)
```

2. 系统函数 get_parameter_descr

```
ivorysql=# select * from get_parameter_description('insert into t values(:x,:y)');
 name  | position
-------+----------
 false |        0
 :x    |        1
 :y    |        2
(3 rows)
```

.13. 13、%TYPE、%ROWTYPE

目的

IvorySQL提供了兼容Oracle的plsql数据类型功能，包括%TYPE、%ROWTYPE。

本文档旨在为使用人员介绍%TYPE、%ROWTYPE的功能。

功能说明

IvorySQL提供了兼容Oracle的%TYPE、%ROWTYPE功能，包括如下内容。

引用发生改变，%TYPE或%ROWTYPE声明的变量也相应改变。

创建表以及函数。

```
CREATE TABLE t1(id int, name varchar(20));

--function's parameter datatype is tablename.columnname%TYPE
CREATE OR REPLACE FUNCTION fun1(v t1.id%TYPE) RETURN varchar AS
BEGIN
  RETURN v;
```

```
END;
/
```

函数状态是valid，并且执行函数能够成功。

```
SELECT prostatus FROM pg_proc WHERE proname like 'fun1'; --v
 prostatus
-----------
 v
(1 row)

SELECT fun1(1) FROM dual;
 fun1
------
 1
(1 row)
```

修改引用的先前定义的声明，函数状态变为invalid，但是函数能够执行成功。

```
ALTER TABLE t1 ALTER COLUMN id TYPE varchar(20);

SELECT prostatus FROM pg_proc WHERE proname like 'fun1'; --n
 prostatus
-----------
 n
(1 row)

--after changing the column id type from int to varchar, call the function again
SELECT fun1('a') FROM dual;  --successfully
 fun1
------
 a
(1 row)
```

重新编译函数，状态重新变成valid。

```
ALTER FUNCTION fun1 COMPILE;
SELECT prostatus FROM pg_proc WHERE proname like 'fun1'; --v
 prostatus
-----------
 v
(1 row)
```

%TYPE声明的变量继承引用变量的约束

示例:

```
--the following testcase will fail
DECLARE
  name     VARCHAR(25) NOT NULL := 'Niu';
  surname  name%TYPE ;
BEGIN
  raise notice 'name=%' ,name;
  raise notice 'surname=%' ,surname;
END;
/

ERROR:  variable "surname" must have a default value, since it's declared NOT NULL
LINE 3:   surname  name%TYPE ;
          ^
```

表名%ROWTYPE或视图名%ROWTYPE作为函数或存储过程的参数类型和函数返回值类型

示例:

```
CREATE TABLE employees(first_name varchar(20) not null,
last_name varchar(20) not null,
phone_number varchar(50));

INSERT INTO employees VALUES ('Steven','Niu','1-650-555-1234');

CREATE OR REPLACE PROCEDURE p0(v employees%ROWTYPE) AS
BEGIN
  raise notice 'v.first_name = %, v.last_name = %, v.phone_number = %',
    v.first_name, v.last_name, v.phone_number;
END;
/

DECLARE
  a employees%ROWTYPE;
BEGIN
  select * into a from employees ;
  raise notice 'a=%', a;
  call p0(a);
END;
/
```

```
NOTICE:  a=(Steven,Niu,1-650-555-1234)
NOTICE:  v.first_name = Steven, v.last_name = Niu, v.phone_number = 1-650-555-1234

\df p0
                        List of functions
 Schema | Name | Result data type |  Argument data types   | Type
--------+------+-----------------+------------------------+------
 public | p0   |                 | IN v employees%ROWTYPE | proc
(1 row)
```

INSERT语句增强

INSERT语句增强支持把一个%ROWTYPE声明的变量插入表中。

语法为：

```
INSERT INTO table_name VALUES row_variable ;
```

示例：

```
CREATE TABLE t1(id int, name varchar(20));

DECLARE
  v1 t1%ROWTYPE;
BEGIN
  FOR i IN 1 .. 5 LOOP
    v1.id := i;
    v1.name := 'a' || i;
    INSERT INTO t1 VALUES v1;
  END LOOP;
END;
/


SELECT * FROM t1;
 id | name
----+------
  1 | a1
  2 | a2
  3 | a3
  4 | a4
  5 | a5
(5 rows)
```

UPDATE语句增强

示例：

```
CREATE TABLE t1(id int, name varchar(20));

DELETE FROM t1;

DECLARE
  v1 t1%ROWTYPE;
  v2 t1%ROWTYPE;
BEGIN
  v1.id := 11;
  v1.name := 'abc';
  INSERT INTO t1 VALUES v1;
  v2.id := 22;
  v2.name := 'new';
  UPDATE t1 SET ROW = v2;
END;
/

SELECT * FROM t1;
 id | name
----+------
 22 | new
(1 row)
```

.14. 14、NLS 参数

目的

NLS 是 National Language Support（国家语言支持）的缩写，指的是 Oracle 提供的本地化支持功能。

IvorySQL提供兼容Oracle的NLS参数功能。

功能说明

包含如下参数：

参数名称	功能描述
ivorysql.datetime_ignore_nls_mask	表示日期格式是否忽略NLS参数影响，默认为0。
nls_length_semantics	兼容Oracle的同名参数，表示char/varchar/varchar2的类型修饰符的大小单位是字节还是字符。
nls_date_format	表示默认的日期格式，可以通过show命令查看，默认为‘YYYY-MM-DD’。
nls_timestamp_format	兼容Oracle的同名参数，控制带时间的日期格式。

nls_timestamp_tz_format	兼容Oracle的同名参数，控制带时区的日期时间格式。
nls_territory	兼容Oracle的同名参数，指定数据库的默认区域。
nls_iso_currency	兼容Oracle的同名参数，指定国家和区域对应的唯一货币符。
nls_currency	兼容Oracle的同名参数，指定显示本地货币的符号，对应数字字符串格式中占位符L。

NLS 日期掩码设置

示例:

```
ivorysql=# set ivorysql.datetime_ignore_nls_mask = 0;
SET
ivorysql=# select '2025-10-15 11:00:00.102030 CST'::oratimestamp ;
ERROR:  datetime format picture ends before converting entire input string
LINE 1: select '2025-10-15 11:00:00.102030 CST'::oratimestamp ;
               ^
ivorysql=# set ivorysql.datetime_ignore_nls_mask = 2;
SET
ivorysql=# select '2025-10-15 11:00:00.102030 CST'::oratimestamp ;
        oratimestamp
----------------------------
 2025-10-15 11:00:00.102030
(1 row)
```

禁用NLS日期/时间戳参数

示例:

```
ivorysql=# select '2025-10-15 11:00:00.102030 '::oratimestamp ;
        oratimestamp
----------------------------
 2025-10-15 11:00:00.102030
(1 row)

ivorysql=# set nls_timestamp_format="pg";
SET
ivorysql=# select '2025-10-15 11:00:00.102030 '::oratimestamp ;
ERROR:  date format not recognized
LINE 1: select '2025-10-15 11:00:00.102030 '::oratimestamp ;
               ^
```

设置nls_length_semantics

IvorySQL使用nls_length_semantics参数的值来决定长度类型，有byte和char两种值，默认为byte。 示例:

```
vorysql=# alter session set nls_length_semantics = char;
SET
ivorysql=# create table character_tb(char_c char(6), char_b varchar2(6 byte), char_v
varchar(6));
CREATE TABLE
ivorysql=# insert into character_tb values('中文测试数据', '123456', '成功');
INSERT 0 1
ivorysql=# select * from character_tb ;
    char_c     | char_b | char_v
--------------+--------+--------
 中文测试数据 | 123456 | 成功
(1 row)

ivorysql=# select length(char_b), length(char_c), length(char_v) from character_tb;
 length | length | length
--------+--------+--------
      6 |      6 |      2
(1 row)
```

设置NLS货币符号

示例:

```
ivorysql=# show ivorysql.identifier_case_switch;
 ivorysql.identifier_case_switch
---------------------------------
 interchange
(1 row)

ivorysql=# set nls_currency to "CHINA";
SET
ivorysql=# show nls_currency;
 nls_currency
--------------
 CHINA
(1 row)

ivorysql=# set nls_currency to "China";
SET
ivorysql=# show nls_currency;
 nls_currency
--------------
 China
```

```
(1 row)
```

nls_currency和nls_iso_currency的行为，受兼容oracle大小写特性 `identifier_case_switch` 影响

.15. 15、 Force View

目的

- 本文档解释 IvorySQL 中 Force View 的用途，帮助用户在依赖对象尚未就绪时仍可创建强制视图占位并保持与 Oracle 行为一致。
- Force View 可支持在依赖表尚未准备好的情况下进行视图迁移，在依赖满足后通过自动或显式编译快速恢复为普通视图。

功能说明

- `CREATE [OR REPLACE] FORCE VIEW`：即使查询引用了不存在的表或函数，也会创建视图占位对象并保存原始 SQL，提示“View created with compilation errors”。
- 自动编译：访问 Force View 时会尝试重新编译；成功后变为普通视图，失败则报错 `view "<schema>.<name>" has errors`。
- 依赖失效回退：当普通视图因依赖对象被删除或结构变更而失效时，系统会自动将其转化为 Force View，保留最后一次有效定义以便后续恢复。

测试用例

创建缺失依赖的 Force View

```
-- 引用不存在的基础表，验证可成功创建占位
CREATE FORCE VIEW fv_customer AS
SELECT c_id, c_name FROM missing_customer;
-- 期望输出: WARNING: View created with compilation errors
```

自动编译并恢复为普通视图

```
-- 先补齐依赖对象
CREATE TABLE missing_customer(
  c_id   int primary key,
  c_name text
);
INSERT INTO missing_customer VALUES (1, 'Alice');

-- 访问 Force View 时自动尝试编译
SELECT * FROM fv_customer;
-- 成功时视图转换为普通视图并返回数据
```

显式编译与失败回退

```
-- 再次让视图失效：重建定义为 Force View
CREATE OR REPLACE FORCE VIEW fv_customer AS
SELECT c_id, upper(c_name) AS c_name FROM missing_customer_v2;

-- 显式编译：依赖仍缺少，保持 Force View 状态并输出 WARNING
ALTER VIEW fv_customer COMPILE;
-- 期望输出：WARNING: View altered with compilation errors

-- 补齐依赖后再次编译
CREATE TABLE missing_customer_v2(
  c_id   int,
  c_name text
);
ALTER VIEW fv_customer COMPILE;
-- 期望输出：ALTER VIEW 成功，视图恢复为普通视图
```

.16. 16、嵌套子函数

目的

- 在 IvorySQL 中使用 Oracle 风格嵌套子函数。

功能说明

- 支持在匿名块、函数或过程内部声明与调用子函数/子过程，作用域限定在父块内。
- 子函数可读取及更新父级变量，也可定义自身局部变量；父级无法直接访问子函数内部状态。
- 支持重载解析机制，按参数个数、类型或命名区分同名子程序。

测试用例

```
DO $$
DECLARE
  v_result integer;
  FUNCTION inner_square(p_value integer) RETURN integer IS
  BEGIN
    RAISE NOTICE 'inner_square called';
    RETURN p_value * p_value;
  END;
BEGIN
  v_result := inner_square(10);
  RAISE NOTICE 'result=%', v_result;
END;
```

```
$$ LANGUAGE plisql;
```

```
DO $$
DECLARE
  v_base_multiplier  integer := 20;
  v_audit_counter    integer := 0;
  v_result integer;
  FUNCTION inner_square(p_value integer) RETURN integer IS
  BEGIN
    RAISE NOTICE 'inner_square called';
    v_audit_counter := v_audit_counter + 1;
    RETURN v_base_multiplier * p_value;
  END;
BEGIN
  v_result := inner_square(10);
  RAISE NOTICE 'result=%', v_result;
  RAISE NOTICE 'v_audit_counter=%', v_audit_counter;
END;
$$ LANGUAGE plisql;
```

```
-- Polymorphic nested function specializing on argument type
DO $$
DECLARE
  v_last_notice text := 'none';

  FUNCTION describe_value(p_input anyelement) RETURN text IS
  BEGIN
    v_last_notice := format('polymorphic dispatch with %s', pg_typeof(p_input));
    RETURN v_last_notice;
  END;

  FUNCTION describe_value(p_input anyarray, p_element anyelement) RETURN text IS
  BEGIN
    v_last_notice := format('array dispatch with %s', pg_typeof(p_input)::text);
    RETURN v_last_notice;
  END;
BEGIN
  RAISE NOTICE '%', describe_value(100);
  RAISE NOTICE '%', describe_value('IvorySQL'::text); -- explicit cast avoids
ambiguous literal
  RAISE NOTICE '%', describe_value(ARRAY[1,2,3], NULL::int); -- extra arg guides
anyarray resolution
```

```
  RAISE NOTICE 'last notice=%', v_last_notice;
END;
$$ LANGUAGE plisql;
```

.17. 17、sys_guid 函数

目的

IvorySQL的sys_guid()
是一个强大的随机数产生函数，它产生并返回一个由16个字节组成的数据库级别唯一的标识符(原始值)。

使用示例

```
ivorysql=# select sys_guid() from dual;
              sys_guid
------------------------------------
 \x3ed9426c8a093442a38bea09a74f44a1
(1 row)
```

sys_guid函数在建表时可以作为主键

```
create table student
(
student_id raw(16) default sys_guid() primary key,
student_name varchar2(100) not null
);
```

新增数据时自动填充主键

```
insert into student(student_name) values ('Steven');
```

.18. 18、空字符串转null

目的

- 在 IvorySQL 的 Oracle
  兼容模式下,支持将空字符串转换成NULL进行存储,提供与Oracle数据库一致的行为。

功能说明

- Oracle兼容模式下支持将空字符串转换成NULL进行存储。
- 通过参数 `ivorysql.enable_emptystring_to_null` 控制该特性,默认值为 `on`。
- 当该参数开启时,插入空字符串会自动转换为NULL值存储。
- 可以通过 `IS NULL` 条件查询到转换后的NULL值。

测试用例

```
-- 创建测试表
ivorysql=# create table abc (id int);
CREATE TABLE

-- 查看空字符串转NULL参数状态
ivorysql=# show ivorysql.enable_emptystring_to_null;
 ivorysql.enable_emptystring_to_NULL
-------------------------------------
 on
(1 row)

-- 插入空字符串
ivorysql=# insert into abc values('');
INSERT 0 1

-- 查询表数据,显示为NULL
ivorysql=# select * from abc;
 id
\----

(1 row)

-- 使用IS NULL条件查询
ivorysql=# select * from abc where id is null;
 id
\----

(1 row)
```

.19. 19、CALL INTO

目的

- IvorySQL中的CALL语句支持调用单独的函数存储过程，也可以是在包或对象类型中的函数和存储过程。CALL调用函数增加INTO子句语法，插入的目标是一个host variable。

功能说明

- CALL支持调用单独的或定义在包中函数和存储过程。
- CALL调用函数时，增加INTO子句语法，插入的目标是一个host variable。
- CALL调用无参或全取默认值的函数/存储过程不能省略空括号。
- CALL在调用函数和存储过程时的参数或INTO子句中支持引用绑定变量。

- CALL语句中OUT参数对应的绑定变量支持对精度和数据类型的验证。
- 输出绑定变量不允许被重复绑定。

测试用例

call into调用函数，并插入host variable

```
-- 创建函数
ivorysql=# create or replace function f_defs(a number default 1314)
ivorysql-# return number
ivorysql-# is
ivorysql-# begin
ivorysql-# raise notice '%', a;
ivorysql-# return a;
ivorysql-# end;
ivorysql-# /
CREATE FUNCTION
-- 声明绑定变量
ivorysql=# variable x number
-- 调用函数并获取返回值
ivorysql=# call f_defs() into :x;
NOTICE: 1314

Call completed.

ivorysql=# print x
  X
------
 1314
```

call调用包中的存储过程

```
ivorysql=# create table tb1(c1 int);
CREATE TABLE
-- 创建包规范
ivorysql=# create or replace package pkg is
ivorysql-# var1 integer;
ivorysql-# procedure test_p ;
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE
-- 创建包体
ivorysql=# create or replace package body pkg is
```

```
ivorysql-# procedure test_p is
ivorysql-# begin
ivorysql-# insert into tb1 values(1);
ivorysql-# end;
ivorysql-# begin
ivorysql-# var1 := 2;
ivorysql-# end;
ivorysql-# /
CREATE PACKAGE BODY
-- 调用包中的过程
ivorysql=# call pkg.test_p();
CALL
ivorysql=# select * from tb1;
 c1
-----
  1
(1 row)
```

无参和默认参数调用

```
-- 创建带默认参数函数
ivorysql=# CREATE OR REPLACE FUNCTION default_arg_func(p_num NUMBER DEFAULT 100)
RETURN NUMBER AS
ivorysql-# BEGIN
ivorysql-#   RETURN p_num + 5;
ivorysql-# END;
ivorysql-# /
CREATE FUNCTION
-- 正确调用默认参数函数（必须带括号）
ivorysql=# VARIABLE default_result NUMBER;
ivorysql=# CALL default_arg_func() INTO :default_result; -- 使用默认值100

Call completed.

ivorysql=# PRINT default_result;
 DEFAULT_RESULT
----------------
 105

-- 带参数调用
ivorysql=# CALL default_arg_func(200) INTO :default_result;

Call completed.
```

```
ivorysql=# PRINT default_result;
 DEFAULT_RESULT
----------------
 205
```

在参数或INTO子句中引用绑定变量

```
-- 创建独立函数 stand_alone_func
CREATE OR REPLACE FUNCTION stand_alone_func(p_input NUMBER)
RETURN NUMBER
IS
BEGIN
    RETURN p_input * 2;
END;
/
-- 设置输入绑定变量
ivorysql=# VARIABLE input_num NUMBER = 7;
-- 使用绑定变量作为参数调用函数
ivorysql=# VARIABLE func_result NUMBER;
ivorysql=# CALL stand_alone_func(:input_num) INTO :func_result;

Call completed.

ivorysql=# PRINT func_result;
 FUNC_RESULT
-------------
 14
```

CALL语句中OUT参数支持精度和数据类型验证

```
-- 创建带OUT参数的过程
ivorysql=# CREATE OR REPLACE PROCEDURE out_param_proc(
ivorysql(#   p_in IN VARCHAR2,
ivorysql(#   p_out OUT VARCHAR2,
ivorysql(#   p_num_out OUT NUMBER
ivorysql(# ) AS
ivorysql-# BEGIN
ivorysql-#   p_out := p_in || ' processed';
ivorysql-#   p_num_out := LENGTH(p_in);
ivorysql-# END;
ivorysql-# /
```

```
CREATE PROCEDURE
-- 测试OUT参数类型匹配
ivorysql=# VARIABLE out_var VARCHAR2(50);
ivorysql=# VARIABLE num_var NUMBER;
ivorysql=# CALL out_param_proc('Test input', :out_var, :num_var);

Call completed.

ivorysql=# PRINT out_var;
       OUT_VAR
----------------------
 Test input processed

ivorysql=# PRINT num_var;
 NUM_VAR
---------
 10

-- 测试OUT参数精度不足（会截断）
ivorysql=# VARIABLE short_out VARCHAR2(5);
ivorysql=# CALL out_param_proc('Long input string', :short_out, :num_var);

Call completed.

ivorysql=# PRINT short_out;
 SHORT_OUT
-----------
 Long

-- 测试类型不匹配（会报错）
ivorysql=# VARIABLE wrong_type NUMBER;
ivorysql=# CALL out_param_proc('Test', :wrong_type, :num_var);
ERROR:  invalid input syntax for type numeric: "Test processed"
```

输出绑定变量不允许重复绑定

```
-- 准备绑定变量
ivorysql=# VARIABLE dup_var VARCHAR2(100);
-- 尝试重复绑定（会报错）
ivorysql=# CALL out_param_func('Test', :dup_var) INTO :dup_var;
ERROR:  output parameter cannot be a duplicate bind
-- 正确做法：使用不同的绑定变量
```

```
ivorysql=# VARIABLE out1 VARCHAR2(100);
ivorysql=# CALL out_param_proc('Correct usage', :out1, :num_var);

Call completed.

ivorysql=# PRINT out1;
          OUT1
-------------------------
 Correct usage processed
```

Chapter 1. 实时查询分析

概述

ivy_mooncake 是 pg_mooncake 的 IvorySQL 发行版。它是一个 Postgres 扩展，可在 Apache Iceberg 中为 Postgres 表创建列存镜像，从而支持亚秒级新鲜度的快速分析（OLAP）查询。它让 IvorySQL 在事务表之上直接获得实时分析处理能力：

- 实时摄取：由 moonlink 驱动，支持流式与批量的 INSERT/UPDATE/DELETE。
- 快速分析：由 DuckDB 加速，在 ClickBench 上排名前 10。
- Postgres 原生：可像查询普通 Postgres 表一样查询列存表。
- Iceberg 原生：数据可被其他查询引擎直接访问。

文档下载

- ivy_mooncake 技术白皮书（V1.0 beta1）
- ivy_mooncake 用户使用手册（V1.0 beta1）

安装

使用 Docker 运行（预览版）

提供了预装 `pg_duckdb` 与 `pg_mooncake` 的 IvorySQL 预览镜像：

```
docker run --name ivy_mooncake \
  -e IVORYSQL_PASSWORD=password \
  -p 5432:5432 -p 1521:1521 \
  -v ivy_mooncake_data:/var/lib/ivorysql/data \
  -v ivy_mooncake_warehouse:/tmp/moonlink_iceberg \
  registry.highgo.com/mooncake/ivy_mooncake:0.1
```

快速开始

- 创建 `pg_mooncake` 扩展

  ```
  CREATE EXTENSION pg_mooncake CASCADE;
  ```

- 创建普通 Postgres 表 `trades`

  ```
  CREATE TABLE trades(
    id bigint PRIMARY KEY,
    symbol text,
    time timestamp,
    price real
  );
  ```

- 创建与 trades 保持同步的列存镜像 trades_iceberg

```
CALL mooncake.create_table('trades_iceberg', 'trades');
```

- 向 trades 插入数据

```
INSERT INTO trades VALUES
  (1,  'AMD', '2024-06-05 10:00:00', 119),
  (2, 'AMZN', '2024-06-05 10:05:00', 207),
  (3, 'AAPL', '2024-06-05 10:10:00', 203),
  (4, 'AMZN', '2024-06-05 10:15:00', 210);
```

- 查询 trades_iceberg，它反映 trades 的最新状态

```
SELECT avg(price) FROM trades_iceberg WHERE symbol = 'AMZN';
```

容器化与云服务

容器化指南

..1. K8S部署

单机容器

进入k8s集群的master节点上，创建名为ivorysql的namespace

```
[root@k8s-master ~]# kubectl create ns ivorysql
```

下载最新docker_library代码

```
[root@k8s-master ~]# git clone https://github.com/IvorySQL/docker_library.git
```

进入单机目录

```
[root@k8s-master ~]# cd docker_library/k8s-cluster/single
[root@k8s-master single]# vim statefulset.yaml
#根据个人环境自行修改statefulset中的pvc信息及数据库密码
```

使用statefulset.yaml创建一个单机pod

```
[root@k8s-master single]# kubectl apply -f statefulset.yaml
service/ivorysql-svc created
statefulset.apps/ivorysql created
```

等待pod创建成功

```
[root@k8s-master single]# kubectl get all -n ivorysql
NAME             READY   STATUS              RESTARTS   AGE
pod/ivorysql-0   0/1     ContainerCreating   0          47s

NAME                   TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/ivorysql-svc   NodePort   10.108.178.236   <none>
5432:32106/TCP,1521:31887/TCP   47s

NAME                        READY   AGE
statefulset.apps/ivorysql   0/1     47s
[root@k8s-master single]# kubectl get all -n ivorysql
NAME             READY   STATUS    RESTARTS   AGE
pod/ivorysql-0   1/1     Running   0          2m39s
```

```
NAME                   TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/ivorysql-svc   NodePort   10.108.178.236   <none>
5432:32106/TCP,1521:31887/TCP   2m39s

NAME                        READY   AGE
statefulset.apps/ivorysql   1/1     2m39s
```

psql连接IvorySQL的PG端口

```
[root@k8s-master single]# psql -U ivorysql -p 32106 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
-----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)

ivorysql=# exit
```

psql连接IvorySQL的Oracle端口

```
[root@k8s-master single]# psql -U ivorysql -p 31887 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
-----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
```

```
 ivorysql.compatible_mode
-------------------------
 oracle
(1 row)
```

卸载单机容器

```
[root@k8s-master single]# kubectl delete -f statefulset.yaml
```

高可用集群

进入k8s集群的master节点上，创建名为ivorysql的namespace

```
[root@k8s-master ~]# kubectl create ns ivorysql
```

下载最新docker_library代码

```
[root@k8s-master ~]# git clone https://github.com/IvorySQL/docker_library.git
```

进入高可用集群目录

```
[root@k8s-master ~]# cd docker_library/k8s-cluster/ha-cluster/helm_charts
[root@k8s-master single]# vim values.yaml
#根据个人环境自行修改values.yaml中的pvc信息及集群规模等信息，数据库密码查看templates/s
ecret.yaml并自行修改。
```

使用 Helm 命令部署高可用集群

```
[root@k8s-master helm_charts]# helm install ivorysql-ha-cluster -n ivorysql .
NAME: ivorysql-ha-cluster
LAST DEPLOYED: Wed Sep 10 09:45:36 2025
NAMESPACE: ivorysql
STATUS: deployed
REVISION: 1
TEST SUITE: None
[root@k8s-master helm_charts]# kubectl get all -n ivorysql
NAME                        READY   STATUS    RESTARTS   AGE
pod/ivorysql-patroni-hac-0  1/1     Running   0          42s
pod/ivorysql-patroni-hac-1  0/1     Running   0          18s

NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)                        AGE
service/ivorysql-patroni-hac        NodePort    10.96.119.203    <none>
5432:32391/TCP,1521:32477/TCP   42s
```

```
service/ivorysql-patroni-hac-config   ClusterIP   None            <none>
<none>                          42s
service/ivorysql-patroni-hac-pods     ClusterIP   None            <none>
<none>                          42s
service/ivorysql-patroni-hac-repl     NodePort    10.100.122.0    <none>
5432:30111/TCP,1521:32654/TCP   42s

NAME                                    READY   AGE
statefulset.apps/ivorysql-patroni-hac   1/3     42s
```

等待所有 Pod 进入“Running”（运行中）状态，即表示集群已部署成功。

```
[root@k8s-master helm_charts]# kubectl get all -n ivorysql
NAME                        READY   STATUS    RESTARTS   AGE
pod/ivorysql-patroni-hac-0  1/1     Running   0          88s
pod/ivorysql-patroni-hac-1  1/1     Running   0          64s
pod/ivorysql-patroni-hac-2  1/1     Running   0          41s

NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)                         AGE
service/ivorysql-patroni-hac        NodePort    10.96.119.203    <none>
5432:32391/TCP,1521:32477/TCP   88s
service/ivorysql-patroni-hac-config   ClusterIP   None            <none>
<none>                          88s
service/ivorysql-patroni-hac-pods     ClusterIP   None            <none>
<none>                          88s
service/ivorysql-patroni-hac-repl     NodePort    10.100.122.0    <none>
5432:30111/TCP,1521:32654/TCP   88s
NAME                                    READY   AGE
statefulset.apps/ivorysql-patroni-hac   3/3     88s
```

使用psql连接集群主节点的PG、Oracle端口

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 32391 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
```

```
-------------------
 f
(1 row)

ivorysql=# exit
```

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 32477 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 f
(1 row)

ivorysql=#
```

使用psql连接集群备节点的PG、Oracle端口

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 30111 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
(1 row)

ivorysql=# exit
```

```
[root@k8s-master helm_charts]# psql -U ivorysql -p 32654 -h 127.0.0.1 -d ivorysql
Password for user ivorysql:

ivorysql=#  show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
(1 row)

ivorysql=#
```

卸载高可用集群

```
[root@k8s-master helm_charts]# helm uninstall ivorysql-ha-cluster -n ivorysql
```

删除PVC

```
[root@k8s-master helm_charts]# kubectl delete pvc ivyhac-config-ivorysql-patroni-hac-0
-n ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc ivyhac-config-ivorysql-patroni-hac-1
-n ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc ivyhac-config-ivorysql-patroni-hac-2
-n ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc pgdata-ivorysql-patroni-hac-0 -n
ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc pgdata-ivorysql-patroni-hac-1 -n
ivorysql
[root@k8s-master helm_charts]# kubectl delete pvc pgdata-ivorysql-patroni-hac-2 -n
ivorysql
```

..2. Operator部署

Operator安装

1. Fork ivory-operator 仓库 并克隆到本地：

```
YOUR_GITHUB_UN="<your GitHub username>"
git clone --depth 1 "git@github.com:${YOUR_GITHUB_UN}/ivory-operator.git"
```

```
cd ivory-operator
```

2. 执行以下命令完成安装：

```
kubectl apply -k examples/kustomize/install/namespace
kubectl apply --server-side -k examples/kustomize/install/default
```

说明

在本教程中，我们将基于 `examples/kustomize/ivory` 目录中的示例进行构建。

当引用 YAML 清单中的嵌套对象时，我们将使用类似 `kubectl explain` 的 `.` 格式。例如，对于以下 YAML 文件：

```
spec:
  hippos:
    appetite: huge
```

我们会用 `spec.hippos.appetite` 来表示最深层的元素。

`kubectl explain` 是一个非常有用的命令。你可以使用它来查看 `ivorycluster.ivory-operator.ivorysql.org` 自定义资源定义（CRD）的结构：

```
kubectl explain ivorycluster
```

创建一个 Ivory 集群

创建

创建一个 Ivory 集群非常简单。使用 `examples/kustomize/ivory` 目录中的示例，只需运行：

```
kubectl apply -k examples/kustomize/ivory
```

IVYO 将在 `ivory-operator` 命名空间中创建一个名为 `hippo` 的简单 Ivory 集群。你可以通过以下命令跟踪 Ivory 集群的状态：

```
kubectl -n ivory-operator describe ivoryclusters.ivory-operator.ivorysql.org hippo
```

你也可以使用以下命令跟踪 Ivory Pod 的状态：

```
kubectl -n ivory-operator get pods \
  --selector=ivory-operator.ivorysql.org/cluster=hippo,ivory-operator.ivorysql.org/instance
```

发生了什么？

IVYO 根据 `examples/kustomize/ivory` 目录中的 Kustomize 清单信息创建了 Ivory 集群。让我们通过查看 `examples/kustomize/ivory/ivory.yaml` 文件来更好地理解发生了什么：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

当我们运行 `kubectl apply` 命令时，实际上是在 Kubernetes 中创建了一个 `ivorycluster` 自定义资源。IVYO 检测到新增的 `ivorycluster` 资源后，开始创建在 Kubernetes 中运行 Ivory 所需的所有对象！

还发生了什么？IVYO 从 `metadata.name` 读取值，为 Ivory 集群命名为 `hippo`。此外，IVYO 通过查看 `spec.image` 和 `spec.backups.pgbackrest.image` 的值，分别确定了 Ivory 和 pgBackRest 使用的容器镜像。`spec.postgresVersion` 的值也很重要，它帮助 IVYO 跟踪你使用的 Ivory 主版本。

IVYO 通过清单中的 `spec.instances` 部分知道要创建多少个 Ivory 实例。虽然 `name` 是可选的，但我们选择将其命名为 `instance1`。我们也可以在集群初始化期间创建多个副本和实例，但稍后在我们讨论 如何扩展并创建高可用 Ivory 集群 时会详细介绍。

`ivorycluster` 自定义资源中非常重要的一部分是 `dataVolumeClaimSpec` 部分。它描述了 Ivory 实例将使用的存储，建模自 Persistent Volume Claim。如果你没有提供 `spec.instances.dataVolumeClaimSpec.storageClassName`，则将使用 Kubernetes 环境中的默认存储类。

作为创建 Ivory 集群的一部分，我们还指定了备份存档的信息。IVYO 使用 pgBackRest，这是一个开源的备份与恢复工具，专为处理 TB 级备份而设计。在集群初始化期间，我们可以指定备份和归档（https://www.postgresql.org/docs/current/wal-intro.html[预写日志或 WAL]）的存储位置。我们将在本教程的 灾难恢复 部分更深入地讨论 `ivorycluster` 规范的这一部分，并了解如何将备份存储在 Amazon S3、Google GCS 和 Azure Blob Storage 中。

IvorySQL / pgBackRest Pod 处于 `Pending` 状态

最常见的原因是 PVC 未绑定。请确保你在任何 `volumeClaimSpec` 中正确设置了存储选项。你可以随时更新设置并使用 `kubectl apply` 重新应用更改。

还要确保你有足够的持久卷可用：你的 Kubernetes 管理员可能需要配置更多持久卷。

如果你使用的是 OpenShift，可能需要将 `spec.openshift` 设置为 `true`。

连接到 Ivory 集群

创建 Ivory 集群是一回事，连接到它又是另一回事。让我们看看 IVYO 如何让连接 Ivory 集群变得简单！

背景：Service、Secret 与 TLS

IVYO 会创建一系列 Kubernetes Service，为访问 Ivory 数据库提供稳定的端点。这些端点让应用程序能够始终如一地连接到数据。要查看可用的 Service，可执行：

```
kubectl -n ivory-operator get svc --selector=ivory-operator.ivorysql.org/cluster=hippo
```

输出示例：

```
NAME              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
hippo-ha          ClusterIP   10.103.73.92   <none>        5432/TCP   3h14m
hippo-ha-config   ClusterIP   None           <none>        <none>     3h14m
hippo-pods        ClusterIP   None           <none>        <none>     3h14m
hippo-primary     ClusterIP   None           <none>        5432/TCP   3h14m
hippo-replicas    ClusterIP   10.98.110.215  <none>        5432/TCP   3h14m
```

大多数 Service 用于集群内部管理，无需关注。连接数据库时，只需关注名为 `hippo-primary` 的 Service。得益于 IVYO，你甚至无需手动指定它——这些信息已被写入 Secret！

集群初始化时，IVYO 会引导创建一个数据库和用户，供应用程序使用。相关信息保存在名为 `<clusterName>-pguser-<userName>` 的 Secret 中。对于 `hippo` 集群，该 Secret 名为 `hippo-pguser-hippo`，包含以下键值：

- `user`：用户账户名
- `password`：用户密码
- `dbname`：用户默认可访问的数据库名
- `host`：数据库主机名（指向主实例的 Service）
- `port`：数据库监听端口
- `uri`：PostgreSQL 连接 URI，含完整登录信息
- `jdbc-uri`：PostgreSQL JDBC 连接 URI，供 JDBC 驱动使用

所有连接均通过 TLS 进行。IVYO 自带证书中心（CA），支持使用 Ivory 的 `verify-full` SSL 模式，防止窃听与中间人攻击。你也可以稍后使用自定义 CA，详见 自定义集群 章节。

修改 Service 类型、NodePort 值与元数据

默认情况下，IVYO 部署的 Service 类型为 `ClusterIP`。根据暴露数据库的方式，你可能需要更改为其他 Service 类型 或指定 NodePort 值。

可通过以下字段调整 IVYO 管理的 Service：

- `spec.service`：控制连接主库的 Service
- `spec.userInterface.pgAdmin.service`：控制 pgAdmin 管理工具的 Service

例如，将主库 Service 改为 `NodePort` 并指定端口、注解与标签：

```
spec:
  service:
    metadata:
      annotations:
        my-annotation: value1
      labels:
        my-label: value2
    type: NodePort
    nodePort: 32000
```

重新应用后，再次查看 Service：

```
kubectl -n ivory-operator get svc --selector=ivory-operator.ivorysql.org/cluster=hippo
```

输出示例：

```
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
hippo-ha          NodePort    10.105.57.191   <none>        5432:32000/TCP   48s
hippo-ha-config   ClusterIP   None            <none>        <none>           48s
hippo-pods        ClusterIP   None            <none>        <none>           48s
hippo-primary     ClusterIP   None            <none>        5432/TCP         48s
hippo-replicas    ClusterIP   10.106.18.99    <none>        5432/TCP         48s
```

查看 `hippo-ha` 的详细信息，顶部将显示自定义注解与标签已生效：

```
Name:               hippo-ha
Namespace:          ivory-operator
Labels:             my-label=value2
                    ivory-operator.ivorysql.org/cluster=hippo
                    ivory-operator.ivorysql.org/patroni=hippo-ha
Annotations:        my-annotation: value1
```

> 使用默认 `ClusterIP` 类型时禁止设置 `nodePort`；该值必须在合法范围内且未被占用。此处提供的注解与标签优先级最高。若通过外部暴露 Service 并依赖 TLS 验证，需使用 IVYO 的 自定义 TLS 功能。

连接应用程序

本教程以 Keycloak（开源身份管理应用）为例。Keycloak 可部署在 Kubernetes 上，并使用 Ivory 作为数据库。以下示例清单将 Keycloak 连接到已运行的 `hippo` 集群：

```
kubectl apply --filename=- <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: ivory-operator
  labels:
    app.kubernetes.io/name: keycloak
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: keycloak
  template:
    metadata:
      labels:
        app.kubernetes.io/name: keycloak
    spec:
      containers:
      - image: quay.io/keycloak/keycloak:latest
        args: ["start-dev"]
        name: keycloak
        env:
        - name: DB_VENDOR
          value: "ivory"
        - name: DB_ADDR
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
        - name: DB_PORT
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
        - name: DB_DATABASE
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
        - name: DB_USER
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
        - name: DB_PASSWORD
          valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
        - name: KEYCLOAK_ADMIN
          value: "admin"
        - name: KEYCLOAK_ADMIN_PASSWORD
          value: "admin"
        - name: KC_PROXY
          value: "edge"
        ports:
        - name: http
```

```
          containerPort: 8080
        - name: https
          containerPort: 8443
        readinessProbe:
          httpGet:
            path: /realms/master
            port: 8080
      restartPolicy: Always
EOF
```

注意环境变量部分：

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: host } }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: port } }
- name: DB_DATABASE
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: dbname } }
- name: DB_USER
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: user } }
- name: DB_PASSWORD
  valueFrom: { secretKeyRef: { name: hippo-pguser-hippo, key: password } }
```

所有连接信息均来自 `hippo-pguser-hippo` Secret，无需硬编码或明文传递凭据，即可直接在 GitOps 流水线中使用！

高可用（HA）

Ivory 以稳定著称，通常“开箱即用”。但在 Kubernetes 这样的分布式环境中，仍可能遇到多种影响可用的事件：

- 数据库磁盘或其余硬件故障
- 网络不可达
- 节点操作系统崩溃
- 数据文件损坏
- 整个数据中心失联
- Kubernetes 自身组件（如 Service）被误删

此外，常规运维（小版本升级、安全补丁、硬件更换）也会带来计划内停机。

好消息：IVYO 已针对上述场景提供保护。要让集群真正高可用，首先把它扩容。

为 Ivory 集群添加副本

IVYO 支持两种快速获得 HA 集群的方式：

1. 调大 `spec.instances.replicas`
2. 在 `spec.instances` 新增条目

本文采用方法 1，将副本数设为 2。对应 YAML 片段：

```yaml
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: "{{< param imageIvorySQL >}}"
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2          # <—— 新增副本
      dataVolumeClaimSpec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: "{{< param imagePGBackrest >}}"
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes: ["ReadWriteOnce"]
            resources:
              requests:
                storage: 1Gi
```

应用后稍等片刻，新副本自动初始化。通过以下命令可实时查看实例 Pod：

```bash
kubectl -n ivory-operator get pods \
  -l ivory-operator.ivorysql.org/cluster=hippo,\
    ivory-operator.ivorysql.org/instance-set
```

验证集群自愈能力

测试 1 —— 删除主库 Service

上一篇《连接集群》提到，应用默认通过 `hippo-primary` Service 读写。我们人为删除它：

```bash
kubectl -n ivory-operator delete svc hippo-primary
```

立刻再查询 Service 列表：

```
kubectl -n ivory-operator get svc \
  -l ivory-operator.ivorysql.org/cluster=hippo
```

可见 `hippo-primary` 已被 IVYO 秒级重建。多数应用凭借重连逻辑几乎无感知。

测试 2 —— 删除主库 StatefulSet

首先找到当前主库 Pod 对应的 StatefulSet 名字：

```
PRIMARY_STS=$(kubectl -n ivory-operator get sts \
  -l ivory-operator.ivorysql.org/cluster=hippo,\
    ivory-operator.ivorysql.org/role=master \
  -o jsonpath='{.items[0].metadata.name}')
echo $PRIMARY_STS
```

假设输出为 `hippo-instance1-zj5s`，直接删除：

```
kubectl -n ivory-operator delete sts "$PRIMARY_STS"
```

再次查看 StatefulSet：

```
kubectl -n ivory-operator get sts \
  -l ivory-operator.ivorysql.org/cluster=hippo
```

IVYO 会立即重建被删对象，并自动将原副本重新加入集群。同时，另一实例已被提升为新主：

```
kubectl -n ivory-operator get pods \
  -l ivory-operator.ivorysql.org/role=master \
  -o jsonpath='{.items[0].metadata.labels.ivory-operator\.ivorysql\.org/instance}'
```

即使 IVYO 进程短暂离线，Patroni 仍能独立完成故障切换，确保应用读写不中断。

同步复制（Synchronous Replication）

IvorySQL 支持同步复制，可进一步降低事务丢失风险。只需在集群里增加：

```
spec:
  patroni:
    dynamicConfiguration:
      synchronous_mode: true
```

如需强制所有提交都同步到至少一个副本，可再加：

```
      synchronous_mode_strict: true
```

Patroni 默认“可用性优先”，当同步副本全部失效时会退化为异步；若业务要求绝对同步，请启用 `synchronous_mode_strict`，此时无可用同步副本将拒绝写入。

亲和性（Affinity）与反亲和性

Pod 反亲和

- `preferredDuringSchedulingIgnoredDuringExecution` —— 尽力分散，资源不足时允许同节点
- `requiredDuringSchedulingIgnoredDuringExecution` —— 强制分散，找不到空闲节点则 Pending

示例 —— 强制让同一 `instance-set` 的 Pod 落在不同节点：

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - topologyKey: kubernetes.io/hostname
      labelSelector:
        matchLabels:
          ivory-operator.ivorysql.org/cluster: hippo
          ivory-operator.ivorysql.org/instance-set: instance1
```

节点亲和

将数据库实例固定在带 `workload-role=db` 标签的节点：

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: workload-role
          operator: In
          values: ["db"]
```

Pod 拓扑分布约束（Topology Spread Constraints）

相比反亲和的“0 或 1”限制，拓扑分布约束可按比例打散，粒度更细。字段模板：

```
topologySpreadConstraints:
- maxSkew: <整数>
  topologyKey: <标签键>
  whenUnsatisfiable: <DoNotSchedule | ScheduleAnyway>
  labelSelector: <对象>
```

示例 —— 5 个实例 Pod 在 3 节点间尽量均衡：

```
  instances:
    - name: instance1
      replicas: 5
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              ivory-operator.ivorysql.org/instance-set: instance1
```

备份仓库主机（repo-host）也可同理配置，确保多集群场景下仓库 Pod 分散。

在线调整 Ivory 集群规格

业务蒸蒸日上，流量激增，需要给 Ivory 集群扩容，却又担心 resize 造成停机？ IVYO 提供滚动升级机制，能在零感知或毫秒级中断内完成 CPU、内存、磁盘等所有规格的在线调整。继续阅读前，请确保已按上一章《高可用》部署了 HA 集群（至少 2 副本）。

垂直调整 CPU / 内存

IVYO 把资源声明分散在多处，保持统一语义（与 Kubernetes 原生 `resources` 字段一致），并支持 QoS 类别设置：

`spec.instances.resources` └ Ivory 主容器、init 容器、数据迁移 Job 的 CPU / 内存

`spec.instances.sidecars.replicaCertCopy.resources` └ 副本证书复制 sidecar

`spec.backups.pgbackrest.repoHost.resources` └ pgBackRest 仓库主机及对应 init / 迁移 Job

`spec.backups.pgbackrest.sidecars.*.resources` `spec.backups.pgbackrest.jobs.resources`
`spec.backups.pgbackrest.restore.resources` `spec.dataSource.ivorycluster.resources`

示例：把 `hippo` 每个实例上限调整为 2 CPU、4 GiB 内存

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  instances:
    - name: instance1
      replicas: 2
      resources:          # <—— 新增或修改
        limits:
          cpu: "2"
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes: ["ReadWriteOnce"]
```

```
        resources:
          requests:
            storage: 1Gi
```

```
kubectl apply -k examples/kustomize/ivory
----
```

实时观察滚动过程（逐 Pod 重建）：

```
watch kubectl -n ivory-operator get pods \
  -l ivory-operator.ivorysql.org/cluster=hippo,\
    ivory-operator.ivorysql.org/instance \
  -o custom-columns=NAME:.metadata.name,ROLE:.metadata.labels.ivory-operator\.
ivorysql\.org/role,PHASE:.status.phase
----
```

流程解析： 1. 先升级所有 副本 实例 → 新 Pod 就绪后旧 Pod 才删除 2. 执行受控主从切换（switchover）→ 应用仅感受到毫秒级重连 3. 最后升级原主库 → 再次选主完成

在线扩容 PVC（磁盘）

场景 A – StorageClass 允许扩容

要求： - 底层 StorageClass 的 `allowVolumeExpansion=true` - 只能增不能减

需要调大的字段：

- `spec.instances.dataVolumeClaimSpec.resources.requests.storage` （数据目录）
- `spec.backups.pgbackrest.repos[*].volume.volumeClaimSpec…` （备份仓库）

示例：数据盘 1 GiB → 10 GiB，备份盘 1 GiB → 20 GiB

```
spec:
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        resources:
          requests:
            storage: 10Gi    # 1→10
  backups:
    pgbackrest:
      repos:
        - name: repo1
          volume:
            volumeClaimSpec:
              resources:
```

```
                requests:
                  storage: 20Gi    # 1→20
```

```
kubectl apply -k examples/kustomize/ivory
----
```

IVYO 会按 副本优先、主库最后 的顺序触发底层 `pvc.spec.resources.requests.storage` 修改，Kubelet 与存储插件完成文件系统在线扩容，Pod 无需重建，业务无感知。

场景 B – StorageClass 禁止 扩容

部分公有云早期 StorageClass 或本地盘 CSI 驱动未开启扩容，仍可通过 “新增大容量实例集 → 切换 → 删除老实例集” 完成“曲线”扩容。

步骤示例：

1. 保留原 `instance1`（1 GiB），新增 `instance2`（10 GiB）

```yaml
spec:
  instances:
    - name: instance1
      replicas: 2
      dataVolumeClaimSpec:
        resources:
          requests:
            storage: 1Gi
    - name: instance2          # 新实例集
      replicas: 2
      dataVolumeClaimSpec:
        resources:
          requests:
            storage: 10Gi
```

1. 等待 `instance2` 副本同步追上主库
2. 提交仅含 `instance2` 的清单，IVYO 将自动：
   - 把 `instance2` 某一副本提升为新主
   - 删除 `instance1` 所有 Pod & PVC
   - 完成“数据迁移”

结果：业务未中断，磁盘已换成 10 GiB，老 PVC 被释放，费用停止计费。
反向操作即可“缩容”磁盘（先建小盘实例集 → 切换 → 删大盘）。

常见问题

Pod 无法调度

- 节点剩余资源不满足 `requests` → 扩容节点或降低 requests
- PVC 申请过大 / StorageClass 不存在 → 检查存储类及配额

PVC 大小未变

确认 StorageClass：

```
kubectl get sc -o custom-columns=NAME:.metadata.name,ALLOW
Expansion:.allowVolumeExpansion
----
```

若返回 `false` 或空值，请：

- 换用支持扩容的 StorageClass，或
- 使用上文“场景 B”实例集替换方案

自定义 Ivory 配置

管理 Ivory 集群中多个实例的诀窍之一是确保所有配置更改都能传播到每个实例。这正是 IVYO 的用武之地：当您为集群进行 Ivory 配置更改时，IVYO 会将其应用到所有 Ivory 实例。

例如，在上一步中，我们分别添加了 CPU 和内存限制为 `2.0` 和 `4Gi`。让我们调整一些 Ivory 设置以更好地利用我们的新资源。我们可以在 `spec.patroni.dynamicConfiguration` 部分中进行此操作。以下是一个更新后的示例清单，其中调整了几个设置：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - name: instance1
      replicas: 2
      resources:
        limits:
          cpu: 2.0
          memory: 4Gi
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
```

```
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
  patroni:
    dynamicConfiguration:
      postgresql:
        parameters:
          max_parallel_workers: 2
          max_worker_processes: 2
          shared_buffers: 1GB
          work_mem: 2MB
```

特别是，我们在 `spec` 中添加了以下内容：

```
patroni:
  dynamicConfiguration:
    postgresql:
      parameters:
        max_parallel_workers: 2
        max_worker_processes: 2
        shared_buffers: 1GB
        work_mem: 2MB
```

使用以下命令将这些更新应用到您的 Ivory 集群：

```
kubectl apply -k examples/kustomize/ivory
```

IVYO 将应用这些设置，并在必要时重新启动每个 Ivory 实例。您可以使用 Ivory 的 `SHOW` 命令验证更改是否已生效，例如：

```
SHOW work_mem;
```

应该会产生类似以下的结果：

```
 work_mem
----------
 2MB
```

自定义 TLS

IVYO 中的所有连接都使用 TLS 加密组件之间的通信。IVYO 设置了一个 PKI 和证书颁发机构 (CA)，允许您创建可验证的端点。但是，您可能希望根据组织要求引入不同的 TLS 基础设施。好消息是：IVYO 允许您这样做！

如何自定义 TLS

IVYO 有几个不同的 TLS 端点可以自定义，包括 Ivory 集群的端点以及控制 Ivory 实例之间如何进行身份验证的端点。让我们看看如何通过定义以下内容来自定义 TLS：

- 一个 `spec.customTLSSecret`，用于标识集群并加密通信；以及
- 一个 `spec.customReplicationTLSSecret`，用于复制身份验证。

要自定义 Ivory 集群的 TLS，您需要在 Ivory 集群的命名空间中创建两个 Secret。其中一个 Secret 将是 `customTLSSecret`，另一个将是 `customReplicationTLSSecret`。这两个 Secret 都包含要使用的 TLS 密钥（`tls.key`）、TLS 证书（`tls.crt`）和 CA 证书（`ca.crt`）。

注意：如果提供了 `spec.customTLSSecret`，则必须也提供 `spec.customReplicationTLSSecret`，并且两者都必须包含相同的 `ca.crt`。

自定义 TLS 和自定义复制 TLS Secret 应包含以下字段（如果您无法控制 Secret 的 `data` 中的字段名称，请参见下面的解决方法）：

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

例如，如果您本地计算机上存储有名为 `ca.crt`、`hippo.key` 和 `hippo.crt` 的文件，您可以运行以下命令从这些文件创建 Secret：

```
kubectl create secret generic -n ivory-operator hippo-cluster.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=hippo.key \
  --from-file=tls.crt=hippo.crt
```

创建 Secret 后，您可以在 `ivorycluster.ivory-operator.ivorysql.org` 自定义资源中指定自定义 TLS Secret。例如，如果您创建了 `hippo-cluster.tls` Secret 和 `hippo-replication.tls` Secret，您可以将它们添加到您的 Ivory 集群中：

```
spec:
  customTLSSecret:
    name: hippo-cluster.tls
  customReplicationTLSSecret:
    name: hippo-replication.tls
```

如果您无法控制 Secret 中的键值对，您可以创建一个映射来告诉 Ivory Operator 哪个键保存了预期的值。它看起来类似于以下内容：

```
spec:
```

```
  customTLSSecret:
    name: hippo.tls
    items:
      - key: <tls.crt key in the referenced hippo.tls Secret>
        path: tls.crt
      - key: <tls.key key in the referenced hippo.tls Secret>
        path: tls.key
      - key: <ca.crt key in the referenced hippo.tls Secret>
        path: ca.crt
```

例如，如果 `hippo.tls` Secret 中的 `tls.crt` 位于名为 `hippo-tls.crt` 的键中，`tls.key` 位于名为 `hippo-tls.key` 的键中，`ca.crt` 位于名为 `hippo-ca.crt` 的键中，那么您的映射将如下所示：

```
spec:
  customTLSSecret:
    name: hippo.tls
    items:
      - key: hippo-tls.crt
        path: tls.crt
      - key: hippo-tls.key
        path: tls.key
      - key: hippo-ca.crt
        path: ca.crt
```

注意：尽管自定义 TLS 和自定义复制 TLS Secret 共享相同的 `ca.crt`，但它们不共享相同的 `tls.crt`：

- 您的 `spec.customTLSSecret` TLS 证书应具有与主服务名称匹配的通用名称 (CN) 设置。这是集群名称后缀为 `-primary` 的名称。例如，对于我们的 `hippo` 集群，这将是 `hippo-primary`。
- 您的 `spec.customReplicationTLSSecret` TLS 证书应具有与预设复制用户 `_ivoryrepl` 匹配的通用名称 (CN) 设置。

与其他更改一样，您可以使用 `kubectl apply` 推出 TLS 自定义设置。

标签

有几种方法可以将您自己的自定义 Kubernetes 标签 添加到您的 Ivory 集群。

- 集群：您可以通过编辑自定义资源的 `spec.metadata.labels` 部分将标签应用于集群中的任何 IVYO 托管对象。
- Ivory：您可以通过编辑 `spec.instances.metadata.labels` 将标签应用于 Ivory 实例集及其对象。
- pgBackRest：您可以通过编辑 `ivoryclusters.spec.backups.pgbackrest.metadata.labels` 将标签应用于 pgBackRest 及其对象。

注解

有几种方法可以将您自己的自定义 Kubernetes 注解 添加到您的 Ivory 集群。

- 集群：您可以通过编辑自定义资源的 `spec.metadata.annotations` 部分将注解应用于集群中的任何 IVYO 托管对象。
- Ivory：您可以通过编辑 `spec.instances.metadata.annotations` 将注解应用于 Ivory 实例集及其对象。

- pgBackRest：您可以通过编辑 `spec.backups.pgbackrest.metadata.annotations` 将注解应用于 pgBackRest 及其对象。

Pod 优先级类

IVYO 允许您使用 pod 优先级类 通过设置 Ivory 集群上的 `priorityClassName` 字段来指示 pod 的相对重要性。这可以通过以下方式完成：

- 实例：优先级是按实例集定义的，并应用于该实例集中的所有 Pod，方法是编辑自定义资源的 `spec.instances.priorityClassName` 部分。
- 专用仓库主机：在规范的 repoHost 部分下定义的优先级通过编辑自定义资源的 `spec.backups.pgbackrest.repoHost.priorityClassName` 部分应用于专用仓库主机。
- 备份（手动和计划）：优先级在 `spec.backups.pgbackrest.jobs.priorityClassName` 部分下定义，并将该优先级应用于所有 pgBackRest 备份作业（手动和计划）。
- 还原（数据源或就地）：通过编辑自定义资源的 `spec.dataSource.ivorycluster.priorityClassName` 部分为“数据源”还原或就地还原定义优先级。
- 数据迁移：规范中第一个实例集（数组位置 0）定义的优先级用于 PGDATA 和 WAL 迁移作业。pgBackRest 仓库迁移作业将使用应用于 repoHost 的优先级类。

独立的 WAL PVC

IvorySQL 通过将更改存储在其https://www.postgresql.org/docs/current/wal-intro.html[预写日志 (WAL)] 中来提交事务。由于访问和使用 WAL 文件的方式通常与数据文件不同，并且在高性能情况下，可能需要将 WAL 文件放在单独的存储卷上。使用 IVYO，可以通过在您的 ivorycluster 规范中为您所需的实例添加 `walVolumeClaimSpec` 块来实现，无论是在创建集群时还是之后的任何时间：

```
spec:
  instances:
    - name: instance
      walVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
```

稍后可以通过从实例中移除 `walVolumeClaimSpec` 部分来移除此卷。请注意，在更改 WAL 目录时，会小心避免丢失任何 WAL 文件。只有在先前配置的卷上不再有任何 WAL 文件时，IVYO 才会删除 PVC。

数据库初始化 SQL

IVYO 可以在集群创建和初始化过程中为您运行 SQL。IVYO 使用 psql 客户端运行 SQL，因此您可以使用元命令连接到不同的数据库、更改错误处理或设置和使用变量。其功能在 psql 文档 中有所描述。

初始化 SQL ConfigMap

Ivory 集群规范接受对包含您的 init SQL 文件的 ConfigMap 的引用。更新您的集群规范以包括 ConfigMap 名称 `spec.databaseInitSQL.name` 和您的 SQL 文件的数据键 `spec.databaseInitSQL.key`。例如，如果您使用以下命令创建 ConfigMap：

```
kubectl -n ivory-operator create configmap hippo-init-sql --from
```

```
-file=init.sql=/path/to/init.sql
```

您可以将以下部分添加到您的 ivorycluster 规范中：

```
spec:
  databaseInitSQL:
    key: init.sql
    name: hippo-init-sql
```

ConfigMap 必须与您的 Ivory 集群位于同一命名空间中。

在您将 ConfigMap 引用添加到您的规范后，使用 `kubectl apply -k examples/kustomize/ivory` 应用更改。IVYO 将创建您的 `hippo` 集群，并在集群启动后运行您的初始化 SQL。您可以通过检查 Ivory 集群上的 `databaseInitSQL` 状态来验证您的 SQL 是否已运行。在状态设置期间，您的 init SQL 不会再次运行。您可以使用 `kubectl describe` 命令检查集群状态：

```
kubectl -n ivory-operator describe ivoryclusters.ivory-operator.ivorysql.org hippo
```

在某些情况下，由于 Kubernetes 处理 ivorycluster 状态的方式，IVYO 可能会多次运行您的 SQL 命令。请确保您在 init SQL 中定义的命令是幂等的。

现在 `databaseInitSQL` 已在您的集群状态中定义，请验证数据库对象是否已按预期创建。验证后，我们建议从您的规范中移除 `spec.databaseInitSQL` 字段。从规范中移除该字段也将从集群状态中移除 `databaseInitSQL`。

PSQL 用法

IVYO 使用 psql 交互式终端在您的数据库中执行 SQL 语句。语句使用标准输入和文件名标志传递（例如 `psql -f -`）。

SQL 语句以超级用户身份在默认维护数据库中执行。这意味着您可以完全控制创建数据库对象、扩展或运行您可能需要的任何 SQL 语句。

与用户和数据库管理集成

如果您正在创建用户或数据库，请参阅 用户/数据库管理 文档。通过规范的用户管理部分创建的数据库可以在您的初始化 sql 中引用。例如，如果定义了数据库 `zoo`：

```
spec:
  users:
    - name: hippo
      databases:
        - "zoo"
```

您可以通过将以下 `psql` 元命令添加到您的 SQL 来连接到 `zoo`：

```
\c zoo
create table t_zoo as select s, md5(random()::text) from generate_Series(1,5) s;
```

事务支持

默认情况下，`psql` 会在每个 SQL 命令完成时提交它。要将多个命令组合成一个单独的 事务，请使用 `BEGIN` 和 `COMMIT` 命令。

```
BEGIN;
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
COMMIT;
```

PSQL 退出代码和数据库 Init SQL 状态

`psql` 的退出代码将决定何时设置 `databaseInitSQL` 状态。当 `psql` 返回 `0` 时，状态将被设置，并且不会再次运行 SQL。当 `psql` 返回错误退出代码时，状态将不会被设置。IVYO 将继续尝试执行 SQL，作为其协调循环的一部分，直到 `psql` 正常返回。如果 `psql` 以失败退出，您将需要编辑 ConfigMap 中的文件，以确保您的 SQL 语句将导致成功的 `psql` 返回。对 ConfigMap 进行实时更改的最简单方法是使用以下 `kubectl edit` 命令：

```
kubectl -n <cluster-namespace> edit configmap hippo-init-sql
```

请务必将所有更改传回您的本地文件。另一个选项是在本地文件中进行更改，并使用 `kubectl --dry-run` 创建模板，并将输出通过管道传输到 `kubectl apply`：

```
kubectl create configmap hippo-init-sql --from-file=init.sql=/path/to/init.sql --dry
-run=client -o yaml | kubectl apply -f -
```

> 如果您编辑了 ConfigMap 但更改没有显示出来，您可能正在等待 IVYO 协调您的集群。一段时间后，IVYO 将自动协调集群，或者您可以通过对集群应用任何更改来触发协调（例如，使用 `kubectl apply -k examples/kustomize/ivory`）。

为了确保 `psql` 在您的 SQL 命令失败时返回失败退出代码，请在您的 SQL 文件中设置 `ON_ERROR_STOP` 变量：

```
\set ON_ERROR_STOP
\echo Any error will lead to exit code 3
create table t_random as select s, md5(random()::text) from generate_Series(1,5) s;
```

用户/数据库管理

IVYO 内置了一些即用型便利功能，用于管理 Ivory 集群中的用户和数据库。然而，您可能有需要创建额外用户、调整用户权限或向集群添加额外数据库的需求。

有关 IVYO 中用户和数据库管理工作原理的详细信息，请参阅架构指南中的 用户管理 部分。

创建新用户

您可以通过在 `ivorycluster` 自定义资源中添加以下片段来创建新用户。让我们将其添加到我们的 `hippo` 数据库中：

```
spec:
  users:
```

```
    - name: rhino
```

现在您可以应用更改，并看到新用户已创建。请注意以下事项：

- 该用户只能连接到默认的 `ivory` 数据库。
- 用户不会将任何连接凭据填充到 `hippo-pguser-rhino` Secret 中。
- 该用户是未特权的。

让我们创建一个名为 `zoo` 的新数据库，我们将允许 `rhino` 用户访问该数据库：

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
```

检查 `hippo-pguser-rhino` Secret。您现在应该看到 `dbname` 和 `uri` 字段已填充！

我们可以通过使用 Ivory 提供的标准 角色属性 并将它们添加到 `spec.users.options` 来设置角色权限。假设我们希望 rhino 成为超级用户（在授予 Ivory 超级用户权限时要小心！）。您可以将以下内容添加到规范中：

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "SUPERUSER"
```

就这样：我们创建了一个名为 `rhino` 的 Ivory 用户，该用户具有超级用户权限，并且可以访问 `rhino` 数据库（尽管超级用户可以访问所有数据库！）。

调整权限

假设您想从 `rhino` 中撤销超级用户权限。您可以通过以下方式执行此操作：

```
spec:
  users:
    - name: rhino
      databases:
        - zoo
      options: "NOSUPERUSER"
```

如果您想添加多个权限，您可以在 `options` 中用空格分隔每个权限，例如：

```
spec:
  users:
```

```
    - name: rhino
      databases:
        - zoo
      options: "CREATEDB CREATEROLE"
```

管理 ivory 用户

默认情况下，IVYO 不允许您访问 ivory 用户。但是，您可以通过执行以下操作来访问此帐户：

```
spec:
  users:
    - name: ivory
```

这将创建一个模式为 <clusterName>-pguser-ivory 的 Secret，其中包含 ivory 帐户的凭据。对于我们的 hippo 集群，这将是 hippo-pguser-ivory。

删除用户

IVYO
不会自动删除用户：将用户从规范中移除后，它将仍然存在于您的集群中。要删除用户及其所有对象，作为超级用户，您需要在用户拥有对象的每个数据库中运行 DROP OWNED，并在您的 Ivory 集群中运行 DROP ROLE。

例如，对于上面的 rhino 用户，您将运行以下命令：

```
DROP OWNED BY rhino;
DROP ROLE rhino;
```

请注意，您可能需要根据对象所有权结构运行 DROP OWNED BY rhino CASCADE; —— 请非常小心此命令！

删除数据库

IVYO
不会自动删除数据库：从规范中移除数据库的所有实例后，它将仍然存在于您的集群中。要完全删除数据库，您必须以 Ivory 超级用户身份运行 DROP DATABASE 命令。

例如，要删除 zoo 数据库，您将执行以下命令：

```
DROP DATABASE zoo;
```

灾难恢复与克隆

也许有人不小心删除了 users
表。也许你想把生产数据库克隆到降级环境。也许你想演练灾难恢复系统（这很重要！）。

无论哪种情况，了解如何使用 IVYO
执行“恢复”操作以便从特定时间点恢复数据，或出于其他目的克隆数据库都很重要。

我们来看看如何执行不同类型的恢复操作。首先，让我们了解自定义资源上的核心恢复属性。

恢复属性

IVYO 提供了从现有 ivorycluster 或远程云数据源（如 S3、GCS
等）恢复的能力。有关更多信息，请参阅 从 S3 / GCS / Azure Blob 存储中存储的备份克隆

> 部分。
>
> 请注意，您不能同时使用本地 ivorycluster 数据源和远程云数据源；如果同时填写了 `dataSource.ivorycluster` 和 `dataSource.pgbackrest` 字段，本地 ivorycluster 数据源将优先。

自定义资源上有几个重要属性需要了解，这些都是恢复过程中的关键。所有这些属性都分组在自定义资源的 spec.dataSource.ivorycluster 部分中。

请查看下表，了解每个属性在设置恢复操作时的工作原理。

- `spec.dataSource.ivorycluster.clusterName`：您要从中恢复的集群的名称。这对应于另一个 `ivorycluster` 自定义资源的 `metadata.name` 属性。
- `spec.dataSource.ivorycluster.clusterNamespace`：您要从中恢复的集群的命名空间。当集群存在于不同的命名空间时使用。
- `spec.dataSource.ivorycluster.repoName`：用于恢复的 `spec.dataSource.ivorycluster.clusterName` 中的 pgBackRest 仓库的名称。可以是 `repo1`、`repo2`、`repo3` 或 `repo4` 之一。仓库必须存在于另一个集群中。
- `spec.dataSource.ivorycluster.options`：IVYO 允许的任何额外 pgBackRest 恢复选项或常规选项。例如，您可能希望设置 `--process-max` 以帮助提高大型数据库的性能；但您将无法设置 `--target-action`，因为该选项目前被禁止。（如果存在 `--target`，IVYO 总是将其设置为 `promote`，否则将其留空。）
- `spec.dataSource.ivorycluster.resources`：设置恢复作业的 资源限制和请求 可以确保其高效运行。
- `spec.dataSource.ivorycluster.affinity`：自定义 Kubernetes 亲和性规则约束恢复作业，使其仅在某些节点上运行。
- `spec.dataSource.ivorycluster.tolerations`：自定义 Kubernetes 容忍度 允许恢复作业在 污点 节点上运行。

让我们通过一些示例来了解如何克隆和恢复我们的数据库。

克隆 Ivory 集群

让我们创建一个我们之前创建的 `hippo` 集群的克隆。我们知道我们的集群名为 `hippo`（基于其 `metadata.name`），并且我们只有一个名为 `repo1` 的备份仓库。

让我们称我们的新集群为 `elephant`。我们可以使用如下所示的清单创建 `hippo` 集群的克隆：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: elephant
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
```

```
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

请注意规范中的这一部分：

```
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
```

这部分告诉 IVYO 将 `elephant` 集群创建为 `hippo` 集群的独立副本。

以上就是克隆 Ivory 集群所需的全部操作！IVYO 将在新的持久卷声明 (PVC)
上创建数据副本，并致力于将集群初始化到规范。很简单！

执行时间点恢复 (PITR)

有人删除了用户表吗？您可能希望执行时间点恢复 (PITR)
以将数据库恢复到更改发生之前的状态。幸运的是，IVYO 可以帮助您做到这一点。

您可以使用为 IVYO 的灾难恢复功能提供支持的备份管理工具 pgBackRest 的 restore 命令来设置
PITR。您需要在 `spec.dataSource.ivorycluster.options` 上设置一些选项来执行 PITR。这些选项包括：

- `--type=time`：这告诉 pgBackRest 执行 PITR。
- `--target`：执行 PITR 的目标位置。恢复目标的一个示例是 `2021-06-09 14:15:11-04`。此处指定的时区为 -04，即东部夏令时。有关其他时区选项，请参阅 pgBackRest 文档。
- `--set`（可选）：选择从哪个备份开始 PITR。

开始前的一些快速说明：

- 要执行 PITR，您必须有一个在 PITR 时间之前完成的备份。换句话说，您不能对没有备份的时间执行 PITR！
- 所有相关的 WAL 文件必须成功推送，以便恢复正确完成。
- 确保选择包含所需备份的正确仓库名称！

考虑到这一点，让我们使用上面的 `elephant` 示例。假设我们要执行到 `2021-06-09 14:15:11-04` 的时间点恢复 (PITR)，我们可以使用以下清单：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: elephant
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
      repoName: repo1
      options:
      - --type=time
      - --target="2021-06-09 14:15:11-04"
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

需要注意的部分是：

```
spec:
  dataSource:
    ivoryCluster:
      clusterName: hippo
```

```
      repoName: repo1
      options:
      - --type=time
      - --target="2021-06-09 14:15:11-04"
```

请注意我们如何放入选项以指定进行 PITR 的位置。

使用上述清单，IVYO 将继续创建一个恢复其数据直到 `2021-06-09 14:15:11-04` 的新 Ivory 集群。此时，集群被提升，您可以从该特定时间点开始访问您的数据库！

执行就地时间点恢复 (PITR)

与上面描述的 PITR 恢复类似，您可能希望执行类似的回退到更改发生之前的状态，但不创建另一个 IvorySQL 集群。幸运的是，IVYO 也可以帮助您做到这一点。

您可以使用为 IVYO 的灾难恢复功能提供支持的备份管理工具 pgBackRest 的 restore 命令来设置 PITR。您需要在 `spec.backups.pgbackrest.restore.options` 上设置一些选项来执行 PITR。这些选项包括：

- `--type=time`：这告诉 pgBackRest 执行 PITR。
- `--target`：执行 PITR 的目标位置。恢复目标的一个示例是 `2021-06-09 14:15:11-04`。
- `--set`（可选）：选择从哪个备份开始 PITR。

开始前的一些快速说明：

- 要执行 PITR，您必须有一个在 PITR 时间之前完成的备份。换句话说，您不能对没有备份的时间执行 PITR！
- 所有相关的 WAL 文件必须成功推送，以便恢复正确完成。
- 确保选择包含所需备份的正确仓库名称！

要执行就地恢复，用户首先需要填写规范的恢复部分，如下所示：

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
        options:
        - --type=time
        - --target="2021-06-09 14:15:11-04"
```

然后，要触发恢复，您需要使用以下命令注释 ivorycluster：

```
kubectl annotate -n ivory-operator ivorycluster hippo --overwrite \
  ivory-operator.ivorysql.org/pgbackrest-restore=id1
```

恢复完成后，可以禁用就地恢复：

```
spec:
```

```
  backups:
    pgbackrest:
      restore:
        enabled: false
```

请注意我们如何放入选项以指定进行 PITR 的位置。

使用上述清单，IVYO 将继续重新创建您的 Ivory 集群，以恢复其数据直到 `2021-06-09 14:15:11-04`。此时，集群被提升，您可以从该特定时间点开始访问您的数据库！

恢复单个数据库

出于性能原因或为了将选定数据库移动到没有足够空间来恢复整个集群备份的计算机上，您可能需要从备份中恢复特定数据库。

> pgBackRest 支持这种情况，但请务必确保这是您想要的。以这种方式恢复将从备份中恢复请求的数据库并使其可访问，但备份中的所有其他数据库在恢复后将无法访问。
>
> 例如，如果您的备份包含数据库 `test1`、`test2` 和 `test3`，并且您请求恢复 `test2`，则恢复完成后，`test1` 和 `test3` 数据库将无法访问。请查看 pgBackRest 文档中关于 恢复单个数据库的限制。

您可以使用类似于以下规范的规范从备份中恢复单个数据库：

```
spec:
  backups:
    pgbackrest:
      restore:
        enabled: true
        repoName: repo1
        options:
        - --db-include=hippo
```

其中 `--db-include=hippo` 将仅恢复 `hippo` 数据库的内容。

备用集群

高级高可用性和灾难恢复策略涉及将您的数据库集群分布在数据中心之间，以帮助最大化正常运行时间。IVYO 提供了使用外部存储系统或 IvorySQL 流复制部署可以跨越多个 Kubernetes 集群的 ivorycluster 的方法。https://github.com/IvorySQL/ivory-operator/blob/master/docs/content/architecture/disaster-recovery.md[灾难恢复架构] 文档中提供了 IVYO 备用集群的高级概述。

创建备用集群

本教程部分将描述如何创建三种不同类型的备用集群，一种使用外部存储系统，一种直接从主集群流式传输数据，一种利用外部存储和流式传输。这些示例集群可以在同一个 Kubernetes 集群中使用单个 IVYO 实例创建，也可以通过正确的存储和网络配置分布在不同的 Kubernetes 集群和 IVYO 实例中。

基于仓库的备用集群

基于仓库的备用集群将从存储在外部存储中的 pgBackRest 仓库中恢复。主集群应使用基于云的 备份配置 创建。以下清单定义了一个 ivorycluster，其中 `standby.enabled` 设置为 true，并且 `repoName` 配置为指向主集群中配置的 `s3` 仓库：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  standby:
    enabled: true
    repoName: repo1
```

流式备用集群

流式备用集群依赖于通过网络到主集群的经过身份验证的连接。主集群应可通过网络访问并允许 TLS 身份验证（默认启用 TLS）。在以下清单中，我们将 `standby.enabled` 设置为 `true`，并提供了指向主集群的 `host` 和 `port`。我们还定义了 `customTLSSecret` 和 `customReplicationTLSSecret` 以提供允许备用集群向主集群进行身份验证的证书。对于这种类型的备用集群，您必须使用 自定义 TLS：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  backups:
    pgbackrest:
      repos:
      - name: repo1
```

```
        volume:
          volumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  customTLSSecret:
    name: cluster-cert
  customReplicationTLSSecret:
    name: replication-cert
  standby:
    enabled: true
    host: "192.0.2.2"
    port: 5432
```

具有外部仓库的流式备用集群

另一个选项是使用从主集群流式传输的外部 pgBackRest
仓库创建备用集群。通过此设置，如果流式复制落后，备用集群将继续从 pgBackRest
仓库恢复。在此清单中，我们启用了前两个示例中的设置：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo-standby
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec: { accessModes: [ReadWriteOnce], resources: { requests: {
storage: 1Gi } } }
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  customTLSSecret:
    name: cluster-cert
  customReplicationTLSSecret:
    name: replication-cert
  standby:
    enabled: true
    repoName: repo1
```

```
    host: "192.0.2.2"
    port: 5432
```

提升备用集群

在某些时候，您会希望提升备用集群以开始接受读取和写入。这具有将 WAL（事务归档）推送到 pgBackRest 仓库的净效应，因此我们需要确保我们不会意外创建脑裂场景。如果两个主实例尝试写入同一个仓库，则可能会发生脑裂。如果主集群仍处于活动状态，请确保在尝试提升备用集群之前 关闭 主集群。

一旦主集群处于非活动状态，我们可以通过移除或禁用其 `spec.standby` 部分来提升备用集群：

```
spec:
  standby:
    enabled: false
```

此更改触发将备用领导者提升为 IvorySQL 主实例，并且集群开始接受写入。

从 S3 / GCS / Azure Blob 存储中存储的备份克隆 {#cloud-based-data-source}

您可以从存储在 AWS S3（或使用 S3 协议的存储系统）、GCS 或 Azure Blob 存储中的备份克隆 Ivory 集群，而无需活动的 Ivory 集群！方法与从现有 ivorycluster 克隆类似。如果您希望为人们提供数据集但将其压缩在更便宜的存储上，这很有用。

出于本示例的目的，假设您创建了一个名为 `hippo` 的 Ivory 集群，其备份存储在 S3 中，看起来类似于以下内容：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: hippo
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
```

```
        repo1-path: /pgbackrest/ivory-operator/hippo/repo1
      manual:
        repoName: repo1
        options:
         - --type=full
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
```

确保 `ivyo-s3-creds` 中的凭据与您的 S3 凭据匹配。有关 使用 S3 部署 Ivory 集群进行备份 的更多详细信息，请参阅教程的 备份 部分。

为了从活跃集群创建新集群时获得最佳性能，请确保对前一个集群进行了最近的完整备份。上面的清单设置为进行完整备份。假设 `hippo` 是在 `ivory-operator` 命名空间中创建的，您可以使用以下命令触发完整备份：

```
kubectl annotate -n ivory-operator ivorycluster hippo --overwrite \
  ivory-operator.ivorysql.org/pgbackrest-backup="$( date '+%F_%H:%M:%S' )"
```

等待备份完成。完成后，您可以删除 Ivory 集群。

现在，让我们将 `hippo` 备份中的数据克隆到一个名为 `elephant` 的新集群中。您可以使用类似于以下的清单：

```
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: elephant
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/hippo/repo1
      repo:
        name: repo1
        s3:
          bucket: "my-bucket"
```

```
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: {{< param imagePGBackrest >}}
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/elephant/repo1
      repos:
      - name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
```

在此清单中需要注意以下几点。首先，请注意我们新的 ivorycluster 中的 `spec.dataSource.pgbackrest` 对象与旧的 ivorycluster 中的 `spec.backups.pgbackrest` 对象非常相似，但略有不同。主要区别是：

1. 从基于云的数据源恢复时不需要镜像
2. 从基于云的数据源恢复时，`stanza` 是必填字段
3. `backups.pgbackrest` 有一个 `repos` 字段，这是一个数组
4. `dataSource.pgbackrest` 有一个 `repo` 字段，这是一个单一对象

还要注意相似之处：

1. 我们正在为两者重用密钥（因为新的恢复 pod 需要具有与原始备份 pod 相同的凭据）
2. `repo` 对象是相同的
3. `global` 对象是相同的

这是因为 `elephant` ivorycluster 的新恢复 pod 将需要重用最初设置 `hippo` ivorycluster 时使用的配置和凭据。

在此示例中，我们正在创建一个新的集群，该集群也备份到同一个 S3 存储桶；只有 `spec.backups.pgbackrest.global` 字段已更改为指向不同的路径。这将确保新的 `elephant` 集群将预填充来自 `hippo` 备份的数据，但将备份到自己的文件夹，确保原始备份仓库得到适当保留。

部署此清单以创建 `elephant` Ivory 集群。观察它启动并运行：

```
kubectl -n ivory-operator describe ivorycluster elephant
```

当它准备就绪时，您将看到预期实例的数量与就绪实例的数量相匹配，例如：

```
Instances:
  Name:               00
  Ready Replicas:     1
  Replicas:           1
  Updated Replicas:   1
```

前面的示例展示了如何使用现有的 S3 仓库预填充 ivorycluster，同时使用新的 S3 仓库进行备份。但是使用基于云的数据源的 ivorycluster 也可以使用本地仓库。

例如，假设一个名为 `rhino` 的 ivorycluster 旨在从原始的 `hippo` ivorycluster 预填充，清单将如下所示：

```yaml
apiVersion: ivory-operator.ivorysql.org/v1beta1
kind: IvoryCluster
metadata:
  name: rhino
spec:
  image: {{< param imageIvorySQL >}}
  postgresVersion: {{< param postgresVersion >}}
  dataSource:
    pgbackrest:
      stanza: db
      configuration:
      - secret:
          name: ivyo-s3-creds
      global:
        repo1-path: /pgbackrest/ivory-operator/hippo/repo1
      repo:
        name: repo1
        s3:
          bucket: "my-bucket"
          endpoint: "s3.ca-central-1.amazonaws.com"
          region: "ca-central-1"
  instances:
    - dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
```

```
      image: {{< param imagePGBackrest >}}
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

监控

虽然拥有 高可用性 和 灾难恢复 系统可以在您的 IvorySQL 集群出现问题时提供帮助，但监控可以帮助您预防问题的发生。此外，监控可以帮助您诊断和解决可能导致性能下降的问题，而不是停机。

让我们看看 IVYO 如何允许您在集群中启用监控。

添加 Exporter Sidecar

让我们看看如何使用 Postgres Operator 示例 仓库中的 `kustomize/ivory` 示例将 IvorySQL Exporter sidecar 添加到您的集群中。

监控工具是使用自定义资源的 `spec.monitoring` 部分添加的。目前，唯一支持的监控工具是使用 pgMonitor 配置的 IvorySQL Exporter。

在 `kustomize/ivory/ivory.yaml` 文件中，将以下 YAML 添加到规范中：

```
monitoring:
  pgmonitor:
    exporter:
      image: {{< param imagePostgresExporter >}}
```

保存您的更改并运行：

```
kubectl apply -k kustomize/ivory
```

IVYO 将检测到更改并将 Exporter sidecar 添加到集群中存在的所有 Ivory Pod 中。IVYO 还将完成允许 Exporter 连接到数据库并使用 IVYO 监控 堆栈收集指标的工作。

为 Exporter 配置 TLS 加密

IVYO 允许您配置 exporter sidecar 以使用 TLS 加密。如果您通过 exporter 规范提供自定义 TLS Secret：

```
  monitoring:
    pgmonitor:
      exporter:
        customTLSSecret:
```

```
          name: hippo.tls
```

与 IVYO 可以配置的其他自定义 TLS Secret 一样，Secret 需要在与您的 PostgresCluster 相同的命名空间中创建。它还应该包含启用加密所需的 TLS 密钥 (`tls.key`) 和 TLS 证书 (`tls.crt`)。

```
data:
  tls.crt: <value>
  tls.key: <value>
```

为 exporter 配置 TLS 后，您将需要更新您的 Prometheus 部署以使用 TLS，并且与 exporter 的连接将被加密。查看 Prometheus 文档，了解有关为 Prometheus 配置 TLS 的更多信息。

访问指标

在您的集群中启用 IvorySQL Exporter 后，请按照 IVYO 监控 中概述的步骤安装监控堆栈。这将允许您在 Kubernetes 中部署 pgMonitor 配置的 Prometheus、https://grafana.com/[Grafana] 和 Alertmanager 监控工具。这些工具将默认设置为连接到您的 Ivory Pod 上的 Exporter 容器。

配置监控

虽然默认的 Kustomize 安装应该在大多数 Kubernetes 环境中工作，但可能需要根据您的特定需求进一步自定义项目。

例如，默认情况下，`fsGroup` 设置为 `26`，用于为组成 IVYO 监控堆栈的各种部署定义的 `securityContext`：

```
securityContext:
  fsGroup: 26
```

在大多数 Kubernetes 环境中，此设置是必需的，以确保容器内的进程具有写入组成 IVYO 监控堆栈的每个 Pod 挂载的任何卷所需的权限。但是，在 OpenShift 环境中安装时（更具体地说，当使用 `restricted` 安全上下文约束时），应删除 `fsGroup` 设置，因为 OpenShift 将自动处理在 Pod 的 `securityContext` 中设置适当的 `fsGroup`。

此外，在同一部分中，可能还需要根据您的特定存储配置修改 `supplmentalGroups` 设置：

```
securityContext:
  supplementalGroups : 65534
```

因此，应修改和/或修补（例如，使用额外的覆盖）`kustomize/monitoring` 下的以下文件，以确保 `securityContext` 为您的 Kubernetes 环境正确定义：

- `deploy-alertmanager.yaml`
- `deploy-grafana.yaml`
- `deploy-prometheus.yaml`

为了修改 IVYO 监控安装程序创建的各种存储资源（即 PersistentVolumeClaims）的配置，还可以修改 `kustomize/monitoring/pvcs.yaml` 文件。

此外，还可以通过修改以下配置资源来进一步自定义组成 IVYO 监控堆栈的各种组件（Grafana、Prometheus 和/或 AlertManager）的配置：

- `alertmanager-config.yaml`

- `alertmanager-rules-config.yaml`
- `grafana-datasources.yaml`
- `prometheus-config.yaml`

最后，请注意，可以通过修改 `kustomize/monitoring/grafana-secret.yaml` 文件中的 Grafana Secret 来更新 Grafana 的默认用户名和密码。

安装

一旦 Kustomize 项目根据您的特定需求进行了修改，就可以使用 `kubectl` 和 Kustomize 安装 IVYO 监控：

```
kubectl apply -k kustomize/monitoring
```

卸载

同样，一旦安装了 IVYO 监控，就可以使用 `kubectl` 和 Kustomize 卸载它：

```
kubectl delete -k kustomize/monitoring
```

连接池

连接池有助于扩展和维护应用程序与数据库之间的整体可用性。IVYO 通过支持 PgBouncer 连接池和状态管理器来促进这一点。

让我们看看我们如何添加连接池并将其连接到我们的应用程序！

添加连接池

让我们看看如何使用 Ivory Operator 仓库示例文件夹中的 `kustomize/keycloak` 示例添加连接池。

连接池是使用自定义资源的 `spec.proxy` 部分添加的。目前，唯一支持的连接池是 PgBouncer。

添加 PgBouncer 连接池的唯一必需属性是设置 `spec.proxy.pgBouncer.image` 属性。在 `kustomize/keycloak/ivory.yaml` 文件中，将以下 YAML 添加到规范中：

```
proxy:
  pgBouncer:
    image: {{< param imageIvoryPGBouncer >}}
```

（您也可以在 `kustomize/examples/high-availability` 示例中找到此示例）。

保存您的更改并运行：

```
kubectl apply -k kustomize/keycloak
```

IVYO 将检测到更改并创建一个新的 PgBouncer Deployment！

设置起来相当容易，所以现在让我们看看如何将我们的应用程序连接到连接池。

连接到连接池

当连接池部署到集群时，IVYO 会将附加信息添加到用户 Secret 中，以允许应用程序直接连接到连接池。回想一下，在此示例中，我们的用户 Secret 称为 `keycloakdb-`

pguser-keycloakdb。描述用户 Secret：

```
kubectl -n ivory-operator describe secrets keycloakdb-pguser-keycloakdb
```

您应该看到此 Secret 中包含几个新属性，允许您通过连接池连接到您的 Ivory 实例：

- pgbouncer-host：PgBouncer 连接池的主机名。这引用了 PgBouncer 连接池的 Service。
- pgbouncer-port：PgBouncer 连接池正在侦听的端口。
- pgbouncer-uri：一个 PostgreSQL 连接 URI，提供通过 PgBouncer 连接池登录到 Ivory 数据库的所有信息。
- pgbouncer-jdbc-uri：一个 PostgreSQL JDBC 连接 URI，提供通过使用 JDBC 驱动程序的 PgBouncer 连接池登录到 Ivory 数据库的所有信息。请注意，默认情况下，连接字符串禁用 JDBC 管理预处理事务以实现 与 PgBouncer 一起使用的最佳方式。

打开 kustomize/keycloak/keycloak.yaml 中的文件。更新 DB_ADDR 和 DB_PORT 值如下：

```
- name: DB_ADDR
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-host
} }
- name: DB_PORT
  valueFrom: { secretKeyRef: { name: keycloakdb-pguser-keycloakdb, key: pgbouncer-port
} }
```

这会更改 Keycloak 的配置，使其现在通过连接池连接。

应用更改：

```
kubectl apply -k kustomize/keycloak
```

Kubernetes 将检测到更改并开始部署新的 Keycloak Pod。完成后，Keycloak 现在将通过 PgBouncer 连接池连接到 Ivory！

TLS

IVYO 通过 TLS 部署每个集群和组件。这包括 PgBouncer 连接池。如果您使用自己的 自定义 TLS 设置，则需要在 spec.proxy.pgBouncer.customTLSSecret 中为 PgBouncer 提供 Secret 引用。

PgBouncer 的 TLS 证书应具有与 PgBouncer Service 名称匹配的通用名称 (CN)。这是集群的名称，后缀为 -pgbouncer。例如，对于我们的 hippo 集群，这将是 hippo-pgbouncer。对于 keycloakdb 示例，它将是 keycloakdb-pgBouncer。

要自定义 PgBouncer 的 TLS，您需要在您的 Ivory 集群的命名空间中创建一个 Secret，其中包含要使用的 TLS 密钥 (tls.key)、TLS 证书 (tls.crt) 和 CA 证书 (ca.crt)。Secret 应包含以下值：

```
data:
  ca.crt: <value>
  tls.crt: <value>
  tls.key: <value>
```

例如，如果您本地计算机上存储有名为 ca.crt、keycloakdb-pgBouncer.key 和 keycloakdb-pgBouncer.crt

的文件，则可以运行以下命令：

```
kubectl create secret generic -n ivory-operator keycloakdb-pgBouncer.tls \
  --from-file=ca.crt=ca.crt \
  --from-file=tls.key=keycloakdb-pgBouncer.key \
  --from-file=tls.crt=keycloakdb-pgBouncer.crt
```

您可以在您的 `ivorycluster.ivory-operator.ivorysql.org` 自定义资源中的 `spec.proxy.pgBouncer.customTLSSecret.name` 字段中指定自定义 TLS Secret，例如：

```
spec:
  proxy:
    pgBouncer:
      customTLSSecret:
        name: keycloakdb-pgBouncer.tls
```

自定义

PgBouncer 连接池是高度可定制的，从配置和 Kubernetes 部署角度来看都是如此。让我们探索一些您可以进行的自定义！

配置

可以通过 `spec.proxy.pgBouncer.config` 自定义 PgBouncer 配置。进行配置更改后，IVYO 会将它们推出到任何 PgBouncer 实例，并自动发出“重新加载”。

您可以通过以下几种方式自定义配置：

- `spec.proxy.pgBouncer.config.global`：接受键值对，这些更改全局应用于 PgBouncer。
- `spec.proxy.pgBouncer.config.databases`：接受键值对，这些键值对代表 PgBouncer 数据库定义。
- `spec.proxy.pgBouncer.config.users`：接受键值对，这些键值对代表 应用于特定用户的连接设置。
- `spec.proxy.pgBouncer.config.files`：接受文件列表，这些文件挂载在 `/etc/pgbouncer` 目录中，并在使用 PgBouncer 的 include 指令 考虑任何其他选项之前加载。

例如，要将连接池模式设置为 `transaction`，您需要设置以下配置：

```
spec:
  proxy:
    pgBouncer:
      config:
        global:
          pool_mode: transaction
```

有关 PgBouncer 配置 的参考，请参阅：

https://www.pgbouncer.org/config.html

副本

默认情况下，IVYO 部署一个 PgBouncer 实例。您可能希望运行多个 PgBouncer 实例以具有一定的冗余级别，尽管您仍然希望注意有多少连接将连接到您的 Ivory 数据库！

您可以通过 `spec.proxy.pgBouncer.replicas` 属性管理部署的 PgBouncer 实例的数量。

资源

您可以通过 `spec.proxy.pgBouncer.resources` 属性管理分配给 PgBouncer 实例的 CPU 和内存资源。`spec.proxy.pgBouncer.resources` 的布局应该很熟悉：它遵循设置 容器资源 的标准 Kubernetes 结构。

例如，假设我们想为 PgBouncer 实例设置一些 CPU 和内存限制。我们可以添加以下配置：

```
spec:
  proxy:
    pgBouncer:
      resources:
        limits:
          cpu: 200m
          memory: 128Mi
```

由于 IVYO 使用 Deployment 部署 PgBouncer 实例，因此这些更改会使用滚动更新推出，以最大程度地减少应用程序与 Ivory 实例之间的中断！

注释 / 标签

您可以通过 `spec.proxy.pgBouncer.metadata.annotations` 和 `spec.proxy.pgBouncer.metadata.labels` 属性分别为您的 PgBouncer 实例应用自定义注释和标签。请注意，对这两个属性中的任何一个的任何更改都将优先于您添加的任何其他自定义标签。

Pod 反亲和性 / Pod 亲和性 / 节点亲和性

您可以通过 `spec.proxy.pgBouncer.affinity` 属性控制 pod 反亲和性、pod 亲和性和节点亲和性，特别是：

- `spec.proxy.pgBouncer.affinity.nodeAffinity`：控制 PgBouncer 实例的节点亲和性。
- `spec.proxy.pgBouncer.affinity.podAffinity`：控制 PgBouncer 实例的 Pod 亲和性。
- `spec.proxy.pgBouncer.affinity.podAntiAffinity`：控制 PgBouncer 实例的 Pod 反亲和性。

以上每个都遵循 设置亲和性的标准 Kubernetes 规范。

例如，要为 `kustomize/keycloak` 示例设置首选 Pod 反亲和性规则，您需要向配置中添加以下内容：

```
spec:
  proxy:
    pgBouncer:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            podAffinityTerm:
```

```
            labelSelector:
              matchLabels:
                ivory-operator.ivorysql.org/cluster: keycloakdb
                ivory-operator.ivorysql.org/role: pgbouncer
            topologyKey: kubernetes.io/hostname
```

容忍度

您可以通过设置 容忍度 通过 `spec.proxy.pgBouncer.tolerations` 将 PgBouncer 实例部署到具有污点的节点。此属性遵循 Kubernetes 标准容忍度布局。

例如，如果有一组具有 `role=connection-poolers:NoSchedule` 污点的节点，您希望将 PgBouncer 实例调度到这些节点，您可以应用以下配置：

```
spec:
  proxy:
    pgBouncer:
      tolerations:
      - effect: NoSchedule
        key: role
        operator: Equal
        value: connection-poolers
```

请注意，设置容忍度并不一定意味着 PgBouncer 实例将被分配给具有这些污点的节点。容忍度充当密钥：它们允许您访问节点。如果您希望确保您的 PgBouncer 实例部署到特定节点，您需要将设置容忍度与节点亲和性相结合。

Pod 分布约束

除了使用亲和性、反亲和性和容忍度之外，您还可以通过 `spec.proxy.pgBouncer.topologySpreadConstraints` 设置 拓扑分布约束。此属性遵循 Kubernetes 标准拓扑分布约束布局。

例如，由于我们的每个 pgBouncer Pod 都将设置标准的 `ivory-operator.ivorysql.org/role: pgbouncer` 标签，我们可以在确定 `maxSkew` 时使用此标签。在下面的示例中，由于我们有 3 个节点，`maxSkew` 为 1，并且我们将 `whenUnsatisfiable` 设置为 `ScheduleAnyway`，我们理想情况下应该在每个节点上看到 1 个 Pod，但如果其他约束阻止这种情况发生，我们的 Pod 可以分布得不那么均匀。

```
  proxy:
    pgBouncer:
      replicas: 3
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: my-node-label
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              ivory-operator.ivorysql.org/role: pgbouncer
```

如果您希望确保您的 PgBouncer 实例分布得更均匀（或根本不部署），您需要将 `whenUnsatisfiable` 更新为 `DoNotSchedule`。

管理任务

手动重启 IvorySQL

有时您可能需要手动重启 IvorySQL。这可以通过向集群的 `spec.metadata.annotations` 部分添加或更新自定义注释来完成。IVYO 将检测到更改并执行 滚动重启。

例如，如果您在命名空间 `ivory-operator` 中有一个名为 `hippo` 的集群，您只需要使用以下命令修补 hippo ivorycluster：

```
kubectl patch ivorycluster/hippo -n ivory-operator --type merge \
  --patch '{"spec":{"metadata":{"annotations":{"restarted":"'"$(date)"'"}}}}'
```

观察您的 hippo 集群：您将看到滚动更新已触发，重启已开始。

关闭

您可以通过将 `spec.shutdown` 属性设置为 `true` 来关闭 Ivory 集群。您可以通过编辑清单来执行此操作，或者在 `hippo` 集群的情况下，执行如下命令：

```
kubectl patch ivorycluster/hippo -n ivory-operator --type merge \
  --patch '{"spec":{"shutdown": true}}'
```

这样做的结果是，此集群的所有 Kubernetes 工作负载都缩放为 0。您可以使用以下命令验证这一点：

```
kubectl get deploy,sts,cronjob --selector=ivory-operator.ivorysql.org/cluster=hippo -n
ivory-operator

NAME                              READY   AGE
statefulset.apps/hippo-00-lwgx    0/0     1h

NAME                              SCHEDULE   SUSPEND   ACTIVE
cronjob.batch/hippo-repo1-full    @daily     True      0
```

要将已关闭的 Ivory 集群重新打开，您可以将 `spec.shutdown` 设置为 `false`。

暂停协调和部署

您可以通过将 `spec.paused` 属性设置为 `true` 来暂停 Ivory 集群协调过程。您可以通过编辑清单来执行此操作，或者在 `hippo` 集群的情况下，执行如下命令：

```
kubectl patch ivorycluster/hippo -n ivory-operator --type merge \
  --patch '{"spec":{"paused": true}}'
```

暂停集群将暂停对集群当前状态的任何更改，直到协调恢复。这允许您完全控制何时将 ivorycluster spec 的更改部署到 Ivory 集群。在暂停期间，除了 “Progressing” 条件外，不会更新任何状态。

要恢复 Ivory 集群的协调，您可以将 `spec.paused` 设置为 `false` 或从清单中删除该设置。

轮换 TLS 证书

应尽可能频繁地使凭据失效并替换（轮换）它们，以最大限度地降低其被滥用的风险。与密码不同，每个 TLS 证书都有一个过期日期，因此替换它们是不可避免的。

实际上，IVYO 会在证书过期日期 之前 自动轮换其管理的客户端证书。将在其工作持续时间的 2/3 之后生成新的客户端证书；因此，例如，IVYO 创建的证书在 12 个月后过期，将在大约第 8 个月时被 IVYO 替换。这样做是为了让您不必担心遇到过期证书的问题或服务中断。

触发证书轮换

如果您想轮换单个客户端证书，您可以通过从其证书 Secret 中删除 `tls.key` 字段来重新生成现有集群的证书。

是时候轮换您的 IVYO 根证书了吗？您只需要删除 `ivyo-root-cacert` secret。IVYO 将无缝地重新生成并推出它，确保您的应用程序继续与 Ivory 集群通信，而无需更新任何配置或处理任何停机时间。

```
kubectl delete secret ivyo-root-cacert
```

> IVYO 仅更新包含生成的根证书的 secret。它不会触及自定义证书。

轮换自定义 TLS 证书

当您使用自己的 TLS 证书与 IVYO 时，您有责任适当地替换它们。方法如下。

IVYO 会自动检测并加载对 IvorySQL 服务器和复制 Secret 内容的更改，而不会停机。您或您的证书管理器只需要替换 `spec.customTLSSecret` 引用的 Secret 中的值。

如果您将 `spec.customTLSSecret` 更改为引用新的 Secret 或新的字段，IVYO 将执行 滚动重启。

> 更改 IvorySQL 证书颁发机构时，请确保同时更新 `customReplicationTLSSecret`。

更改主节点

有时您可能希望更改 HA 集群中的主节点。这可以通过使用 ivorycluster spec 的 `patroni.switchover` 部分来完成。它允许您在 ivoryclusters 中启用切换，将特定实例作为新的主节点，并在您的 ivorycluster 进入不良状态时运行故障转移。

让我们完成执行切换的过程！

首先，您需要更新您的 spec 以准备您的集群以更改主节点。编辑您的 spec 以具有以下字段：

```yaml
spec:
  patroni:
    switchover:
      enabled: true
```

应用此更改后，IVYO 将寻找触发器以在您的集群中执行切换。您将通过将 `ivory-operator.ivorysql.org/trigger-switchover` 注释添加到您的自定义资源来触发切换。设置此注释的最佳方法是使用时间戳，这样您就知道何时启动了更改。

例如，对于我们的 `hippo` 集群，我们可以运行以下命令来触发切换：

```
kubectl annotate -n ivory-operator ivorycluster hippo \
  ivory-operator.ivorysql.org/trigger-switchover="$(date)"
```

如果您想执行另一次切换，您可以重新运行注释命令并添加 `--overwrite` 标志：

```
kubectl annotate -n ivory-operator ivorycluster hippo --overwrite \
  ivory-operator.ivorysql.org/trigger-switchover="$(date)"
```

IVYO 将检测到此注释并使用 Patroni API 请求更改当前主节点！

随着 Patroni 的工作，您的数据库实例 Pod 上的角色将开始更改。新的主节点将具有 `master` 角色标签，旧的主节点将更新为 `replica`。

切换的状态将使用 `status.patroni.switchover` 字段进行跟踪。这将设置为您在触发器注释中定义的值。如果您使用时间戳作为注释，这是确定何时请求切换的另一种方法。

在实例 Pod 标签已更新且 `status.patroni.switchover` 已设置后，主节点已在您的集群上更改！

更改主节点后，我们建议您通过将 `spec.patroni.switchover.enabled` 设置为 false 或完全从您的 spec 中删除该字段来禁用切换。如果该字段被删除，相应的状态也将从 ivorycluster 中删除。

定位实例

切换主节点时，您可以选择的另一个选项是提供目标实例作为新的主节点。在执行切换时，此目标实例将用作候选节点。`spec.patroni.switchover.targetInstance` 字段接受您要切换到的实例的名称。

此名称可以在几个不同的地方找到；一个是 StatefulSet 的名称，另一个是数据库 Pod 上的 `ivory-operator.ivorysql.org/instance` 标签。以下命令可以帮助您确定谁是当前主节点以及用作 `targetInstance` 的名称：

```
$ kubectl get pods -l ivory-operator.ivorysql.org/cluster=hippo \
    -L ivory-operator.ivorysql.org/instance \
    -L ivory-operator.ivorysql.org/role -n ivory-operator

NAME                      READY   STATUS      RESTARTS   AGE     INSTANCE
ROLE
hippo-instance1-jdb5-0    3/3     Running     0          2m47s   hippo-instance1-jdb5
master
hippo-instance1-wm5p-0    3/3     Running     0          2m47s   hippo-instance1-wm5p
replica
```

在我们的示例集群中，`hippo-instance1-jdb5` 当前是主节点，这意味着我们希望在切换中定位 `hippo-instance1-wm5p`。现在您知道哪个实例当前是主节点以及如何找到您的 `targetInstance`，让我们更新您的集群 spec：

```
spec:
  patroni:
```

```
    switchover:
      enabled: true
      targetInstance: hippo-instance1-wm5p
```

应用此更改后，您将再次需要通过注释 ivorycluster 来触发切换（请参见上面的命令）。您可以通过检查 Pod 角色标签和 `status.patroni.switchover` 来验证切换是否已完成。

故障转移

最后，当您的集群进入不健康状态时，我们可以选择进行故障转移。完成此操作所需的唯一 spec 更改是将 `spec.patroni.switchover.type` 字段更新为 `Failover` 类型。需要注意的是，执行故障转移时需要 `targetInstance`。基于上面的示例集群，假设 `hippo-instance1-wm5p` 仍然是副本，我们可以更新 spec：

```
spec:
  patroni:
    switchover:
      enabled: true
      targetInstance: hippo-instance1-wm5p
      type: Failover
```

应用此 spec 更改后，您的 ivorycluster 将准备好执行故障转移。同样，您需要通过注释 ivorycluster 来触发切换（请参见上面的命令），并验证 Pod 角色标签和 `status.patroni.switchover` 是否已相应更新。

> 切换过程中遇到的错误可能会使您的集群处于不良状态。如果您遇到问题，请在 operator 日志中找到问题，您可以更新 spec 以修复问题并应用更改。应用更改后，IVYO 将尝试再次执行切换。

删除Ivory集群

总有一个时刻，您需要删除自己的集群。如果您一直在跟着示例操作，只需运行下面一条命令即可删除 Ivory 集群：

```
kubectl delete -k examples/kustomize/ivory
```

IVYO 会清理与该集群相关的所有对象。

关于数据保留：这取决于您 PVC 的 回收策略。如需了解 Kubernetes 如何管理数据保留，请参阅 Kubernetes 官方文档关于卷回收的说明。

..3. Docker & Podman部署

> 除 Docker Hub 外，IvorySQL 社区还为国内用户提供 Harbor 镜像服务。将本章节中的镜像地址前缀替换为 `registry.highgo.com/` 即可使用，例如 `registry.highgo.com/ivorysql/ivorysql:5.6-ubi8`。

docker方式运行

- 从Docker Hub上获取IvorySQL镜像

```
$ docker pull ivorysql/ivorysql:5.6-ubi8
```

- 运行IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/ivorysql:5.6-ubi8
```

- 查看IvorySQL容器运行是否成功

```
$ docker ps | grep ivorysql
CONTAINER ID   IMAGE               COMMAND                   CREATED          STATUS
PORTS                               NAMES
6faa2d0ed705   ivorysql:5.6-ubi8   "docker-entrypoint.s…"   50 seconds ago   Up 49
seconds   5866/tcp, 0.0.0.0:5434->5432/tcp   ivorysql
```

podman方式运行

- 从Docker Hub上获取IvorySQL镜像

```
[highgo@manager-node1 ~]$ podman pull ivorysql/ivorysql:5.6-ubi8
⊠ docker.io/ivorysql/ivorysql:5.6-ubi8
Trying to pull docker.io/ivorysql/ivorysql:5.6-ubi8...
Getting image source signatures
Copying blob 5885448c5c88 done   |
Copying blob 6c502b378234 done   |
Copying blob 8b4f2b90d6b6 done   |
Copying blob 9b000f2935f6 done   |
Copying blob 806f782da874 done   |
Copying blob e4c51845a9eb done   |
Copying blob dcb1e9a04275 done   |
Copying blob 285a279173f8 done   |
Copying blob 1f6f247b9ae0 done   |
Copying blob 3cc81bed8614 done   |
Copying blob 863c87bf25eb done   |
Copying blob 4f4fb700ef54 done   |
Copying config 88e1bbeda8 done   |
Writing manifest to image destination
88e1bbeda81c51d88e12cbd2b19730498f1343d1c64bb3dddc8ffcb08a1f965f
```

- 运行IvorySQL

```
$ podman run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=123456 -d
ivorysql/ivorysql:5.6-ubi8
```

- 查看IvorySQL容器运行是否成功

```
[highgo@manager-node1 ~]$ podman ps | grep ivorysql
CONTAINER ID            IMAGE                        COMMAND          CREATED
STATUS          PORTS                                NAMES
368dee58d5ef  docker.io/ivorysql/ivorysql:5.6-ubi8  postgres    20 seconds ago  Up 20
seconds  0.0.0.0:5434->5432/tcp, 1521/tcp, 5866/tcp  ivorysql

[highgo@manager-node1 ~]$ podman exec -it ivorysql /bin/bash
[root@8cc631eb413d /]#
ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=#
```

..4. Docker Swarm & Docker Compose部署

准备三个网络互通的服务器，并搭建swarm集群。 测试集群名称及对应ip地址如下：

manager-node1: 192.168.21.205

manager-node2: 192.168.21.164

manager-node3: 192.168.21.51

```
[root@manager-node1 docker-swarm]# docker node ls
ID                            HOSTNAME        STATUS    AVAILABILITY   MANAGER STATUS
ENGINE VERSION
y9d9wd9t2ncy4t9bvw6bg9sjs *   manager-node1   Ready     Active         Reachable
26.1.4
iv17o6m9t9e06vd9iu1o6damd     manager-node2   Ready     Active         Leader
25.0.4
vjnax76qj812mlvut6cv4qotl     manager-node3   Ready     Active         Reachable
24.0.6
```

docker swarm搭建IvorySQL HA Cluster

下载源码

```
[root@manager-node1 ~]# git clone https://github.com/IvorySQL/docker_library.git
[root@manager-node1 ~]# cd docker_library/docker-cluster/docker-swarm
```

部署一个三节点的etcd

```
[root@manager-node1 docker-swarm]# docker stack deploy -c docker-swarm-etcd.yml
ivoryhac-etcd
Creating network ivoryhac-etcd_etcd-net
Creating service ivoryhac-etcd_etcd3
Creating service ivoryhac-etcd_etcd1
Creating service ivoryhac-etcd_etcd2
[root@manager-node1 docker-swarm]# docker service ls
ID             NAME                  MODE         REPLICAS   IMAGE
PORTS
1jst0mva8o5n   ivoryhac-etcd_etcd1   replicated   1/1
quay.io/coreos/etcd:v3.5.8   *:2379-2380->2379-2380/tcp
sosag5017cis   ivoryhac-etcd_etcd2   replicated   1/1
quay.io/coreos/etcd:v3.5.8
8twpgkzo2mnx   ivoryhac-etcd_etcd3   replicated   1/1
quay.io/coreos/etcd:v3.5.8
```

可自定义数据库外挂目录，修改docker-swarm-ivypatroni.yml中的volumes，修改完成后修改目录权限及属主；示例如下

```
mkdir -p /home/ivorysql/{data,patroni}
chown -R 1000:1000 /home/ivorysql/{data,patroni}
chmod 700 /home/ivorysql/{data,patroni}
```

部署IvorySQL高可用集群

```
[root@manager-node1 docker-swarm]# docker stack deploy -c docker-swarm-ivypatroni.yml
ivoryhac-patroni
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating service ivoryhac-patroni_ivypatroni1
Creating service ivoryhac-patroni_ivypatroni2
[root@manager-node1 docker-swarm]# docker service ls
ID             NAME                        MODE         REPLICAS   IMAGE
PORTS
1jst0mva8o5n   ivoryhac-etcd_etcd1         replicated   1/1
quay.io/coreos/etcd:v3.5.8                         *:2379-2380->2379-2380/tcp
sosag5017cis   ivoryhac-etcd_etcd2         replicated   1/1
quay.io/coreos/etcd:v3.5.8
8twpgkzo2mnx   ivoryhac-etcd_etcd3         replicated   1/1
quay.io/coreos/etcd:v3.5.8
uzdvjq5j2gwt   ivorysql-hac_ivypatroni1   replicated   1/1        ivorysql/docker-
swarm-ha-cluster:5.0-4.0.6-ubi8   *:1521->1521/tcp, *:5866->5866/tcp
fr0m9chu3ce8   ivorysql-hac_ivypatroni2   replicated   1/1        ivorysql/docker-
swarm-ha-cluster:5.0-4.0.6-ubi8   *:1522->1521/tcp, *:5867->5866/tcp
```

psql连接数据库的Oracle端口及PG端口

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p1521 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# exit
```

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p5432 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)
```

卸载IvorySQL集群

```
[root@manager-node1 ~] docker stack rm ivoryhac-patroni
[root@manager-node1 ~] docker stack rm ivoryhac-etcd
```

docker compose搭建IvorySQL HA Cluster

下载源码

```
[root@manager-node1 ~]# git clone https://github.com/IvorySQL/docker_library.git
[root@manager-node1 ~]# cd docker_library/docker-cluster/docker-compose
```

拷贝文件至其他服务器

将etcd和ivypatroni的docker-compose文件，分别拷贝到其他服务器上。

如测试服务器:

192.168.21.205 存放etcd1+ivorypatroni1，

192.168.21.164 存放etcd2+ivorypatroni2，

192.168.21.51 存放etcd3+ivorypatroni3

部署一个三节点的etcd，以node1为例

```
[root@manager-node1 docker-compose]# docker-compose -f ./docker-compose-etcd1.yml up
-d
[+] Running 1/1
 ⊠ Container etcd  Started
0.1s
```

部署IvorySQL高可用集群

在每个节点上，部署ivyhac服务 以node1为例

```
[root@manager-node1 docker-compose]# docker-compose -f ./docker-compose-
ivypatroni_1.yml up -d
[+] Running 1/1
 ⊠ Container ivyhac1  Started
0.1s
[root@manager-node1 docker-compose]# docker ps
CONTAINER ID   IMAGE                                                COMMAND
CREATED          STATUS          PORTS     NAMES
736c0d188bdd   ivorysql/docker-compose-ha-cluster:5.0-4.0.6-ubi8   "/bin/sh /docker-
ent…"   18 seconds ago   Up 17 seconds             ivyhac1
9d8e04e4f819   quay.io/coreos/etcd:v3.5.8
"/usr/local/bin/etcd"    24 minutes ago   Up 24 minutes             etcd
```

此时，一主两备集群搭建完成 psql连接数据库的Oracle端口及PG端口

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p1521 -U ivorysql -d ivorysql
Password for user ivorysql:
```

```
ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 oracle
(1 row)

ivorysql=# exit
```

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p5432 -U ivorysql -d ivorysql
Password for user ivorysql:

ivorysql=# select version();
                                                        version
-----------------------------------------------------------------------------------------
----------------------------------
 PostgreSQL 18.6 (IvorySQL 5.6) on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0
20210514 (Red Hat 8.5.0-28), 64-bit
(1 row)

ivorysql=# show ivorysql.compatible_mode;
 ivorysql.compatible_mode
--------------------------
 pg
(1 row)
```

卸载IvorySQL集群 以node1为例

```
[root@manager-node1 ~] docker-compose -f ./docker-compose-ivypatroni_1.yml down
[root@manager-node1 ~] docker-compose -f ./docker-compose-etcd1.yml down
```

云服务平台指南

..1. IvorySQL Cloud安装

IvorySQL
Cloud平台是一个综合性的解决方案，它集成了IvorySQL数据库以及周边生态，以提供全面的数据库和资源

管理功能。安装前需要上github编译安装好

前端：https://github.com/IvorySQL/ivory-cloud-web

后端：https://github.com/IvorySQL/ivory-cloud

搭建好K8S集群，并在集群master节点上安装ivory-operator

https://github.com/IvorySQL/ivory-operator/tree/IVYO_REL_5_STABLE

IvorySQL Cloud平台安装

IvorySQL Cloud平台，目前支持Linux系统下的安装，以下为各部分对应的安装包：

组件	安装包
前端	dist
后端	cloudnative-1.0-SNAPSHOT.jar
K8S集群	1. docker.io/ivorysql/ivory-operator:v5.6 2. docker.io/ivorysql/pgadmin:ubi8-9.9-5.0-1 3. docker.io/ivorysql/pgbackrest:ubi8-2.56.0-5.0-1 4. docker.io/ivorysql/postgres-exporter:ubi8-0.17.0-5.0-1 5. docker.io/ivorysql/ivorysql:ubi8-5.0-5.0-1

另外，云服务平台还需要用户安装以下组件：

- 后端数据库：负责存储和管理所有与云资源、用户信息、权限控制、计费信息等相关的数据。需要使用PG系列数据库，如PostgreSQL、瀚高数据库、IvorySQL等。
- NGINX：支持云服务平台的web服务。

安装前准备

安装前，所有服务器都必须完成以下准备工作。并且将IvorySQL Cloud平台部署在K8S（1.23）的服务器上，且该K8S集群需要有默认的storage class.

关闭防火墙

所有服务器关闭防火墙，以保证他们之间的网络互通。

```
systemctl stop firewalld.service

systemctl disable firewalld.service
```

后端部署

后端数据库

IvorySQL Cloud平台的后端数据库需用户自行安装，请参考IvorySQL官网。

后端服务程序

编译后端服务程序

```
# 克隆代码
```

```
git clone https://github.com/IvorySQL/ivory-cloud.git

# 进入到项目根目录

cd ivory-cloud
```

请确保ivory-cloud\cloudnative\src\main\resources\monitor文件夹，及其所有的子文件夹下的以.sh结尾的文件是unix格式的，如果不是，请执行dos2unix命令转换成unix格式。

```
dos2unix cloudnative\src\main\resources\monitor\*

# 编译

mvn clean

mvn package -D maven.test.skip=true

打包完成后，可以在 ivory-cloud/cloudnative/target下找到cloudnative-1.0-SNAPSHOT.jar文件
```

部署程序

```
在K8S服务器上执行如下操作：

# 创建目录

mkdir -p /home/ivory

# 将上一步编译好的文件ivory-cloud/cloudnative/target/cloudnative-1.0-SNAPSHOT.jar上传至上述目录

# 配置文件

## 创建配置目录

mkdir -p /home/ivory/config

## 上传配置文件

将源代码ivory-cloud/cloudnative/src/main/resources目录里的如下三个配置文件上传至 /home/ivory/config
```

```
application.yaml

application-native.yaml

spring_pro_logback.xml

## 修改配置文件，请将url、username、password修改为<<backend-
db,后端数据库章节>>安装的数据库的信息。

## /home/ivory/config/application-native.yaml

datasource:

druid:

db-type: com.alibaba.druid.pool.DruidDataSource

driver-class-name: org.postgresql.Driver

url: jdbc:postgresql://127.0.0.1:5432/ivorysql

username: ivorysql

password: "ivory@123"
```

启动后端服务程序

```
# 安装jdk1.8

yum install -y java-1.8.0-openjdk.x86_64

[root@cloud ivory]# pwd

/home/ivory/

[root@cloud ivory]# nohup java -jar cloudnative-1.0-SNAPSHOT.jar > log_native 2>&1 &

[root@cloud ivory]# ps -ef |grep java

root 77494 1 0 10月09 ? 00:03:07 java -jar cloudnative-1.0-SNAPSHOT.jar
```

前端部署

编译前端服务程序

```
## 获取代码

git clone https://github.com/IvorySQL/ivory-cloud-web.git

## 进入项目根目录

cd ivorysql-cloud-web

## 安装依赖

npm install

## 编译打包

npm run build:prod
```

修改目录和文件权限

```
# 创建目录

[root@cloud opt]# mkdir -p /opt/cloud/web

# 将前端构建后的dist文件夹置于/opt/cloud/web

# 授权

[root@cloud web]# chmod 755 /opt/cloud/web/dist

[root@cloud web]# chmod -R 777 /opt/cloud/web/dist
```

修改config.js

修改文件

```
[root@cloud dist]# pwd

/home/cloud/web/dist

[root@cloud dist]# vi config.js
```

```
var PLATFROM_CONFIG = {};

// ip请更换为当前服务器地址

PLATFROM_CONFIG.baseUrl = "http://192.168.31.43:8081/cloudapi/api/v1"

//true: need to show "注册" on login page

//false: don't show "注册" on login page

globalShowRegister = true

//是否隐藏云原生数据库， true: 隐藏云原生数据库; false:显示云原生数据库

disableNative = false

// 数据库类型

dbtype = "IvorySQL"

dbversion = "5.6"
```

安装部署nginx

IvorySQL Cloud平台服务器需要安装nginx，以支持云服务平台的web服务。nginx需要用户自行安装，这里提供一种安装方法作为参考。

下载nginx安装包

```
[root@cloud web]# wget https://nginx.org/download/nginx-1.20.1.tar.gz

[root@cloud web]# ls -lrt

总用量 3924

-rwxrwxr-x. 1 root root 1061461 5月 25 2021 nginx-1.20.1.tar.gz

-rwxrwxr-x. 1 root root 2943732 10月 9 16:43 dist.tar.gz

drwxrwxrwx. 4 root root 103 10月 21 13:20 dist
```

安装相关依赖

```
[root@host30 cloud]# yum -y install pcre-devel

[root@host30 cloud]# yum -y install openssl openssl-devel
```

编译安装nginx

nginx会被安装在configure时由—prefix指定的目录下，例如这里的/opt/cloud/nginx：

```
## 解压缩nginx-1.20.1.tar.gz安装包

[root@cloud web]# tar -zxvf nginx-1.20.1.tar.gz

## 解压后生成nginx-1.20.1文件夹

[root@cloud web]# ls -lrt

总用量 3924

-rwxrwxr-x. 1 root root 1061461 5月 25 2021 nginx-1.20.1.tar.gz

-rwxrwxr-x. 1 root root 2943732 10月 9 16:43 dist.tar.gz

drwxrwxr-x. 9 1001 1001 186 10月 9 16:53 nginx-1.20.1

drwxrwxrwx. 4 root root 103 10月 21 13:20 dist

## 配置导向

[root@cloud web]# cd nginx-1.20.1

[root@cloud nginx-1.20.1]# ./configure --prefix=/opt/cloud/nginx --with
-http_ssl_module

## 编译安装

[root@cloud nginx-1.20.1]# make

[root@cloud nginx-1.20.1]# make install
```

修改配置文件nginx.conf

配置文件在/opt/cloud/nginx路径下，可以按照github上readme对nginx.conf进行对比修改。ip请配置为当前服务器的ip。

```
server {

listen 9104;

server_name 192.168.31.43;

location / {

root /opt/cloud/web/dist;

index index.html index.htm;

}

error_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}
```

启动nginx

```
[root@cloud sbin]# pwd

/opt/cloud/nginx/sbin

[root@cloud sbin]# ./nginx -c /opt/cloud/nginx/conf/nginx.conf

[root@cloud sbin]# ps -ef | grep nginx

root 2179 131037 0 09:46 pts/1 00:00:00 grep --color=auto nginx

root 55047 1 0 10月21 ? 00:00:00 nginx: master process ./nginx -c
/opt/cloud/nginx/conf/nginx.conf

nobody 55048 55047 0 10月21 ? 00:00:00 nginx: worker process
```

operator部署

请自行搭建K8S，此处描述为在K8S集群上安装ivory-operator和load镜像。

安装ivory-operator

参见

https://github.com/IvorySQL/ivory-operator/tree/IVYO_REL_5_STABLE

网站上的readme

load镜像

如果服务器可以直接访问到docker hub,可以跳过该章节。否则需要在所有的K8S集群节点提前load
如下docker镜像

国内用户也可使用 IvorySQL 社区的 Harbor 镜像服务拉取以下镜像，将 `docker.io/` 替换为 `registry.highgo.com/` 即可。

```
docker.io/ivorysql/pgadmin:ubi8-9.9-5.0-1

docker.io/ivorysql/pgbackrest:ubi8-2.56.0-5.0-1

docker.io/ivorysql/pgbouncer:ubi8-1.23.0-5.0-1

docker.io/ivorysql/postgres-exporter:ubi8-0.17.0-5.0-1

docker.io/ivorysql/ivorysql:ubi8-5.0-5.0-1

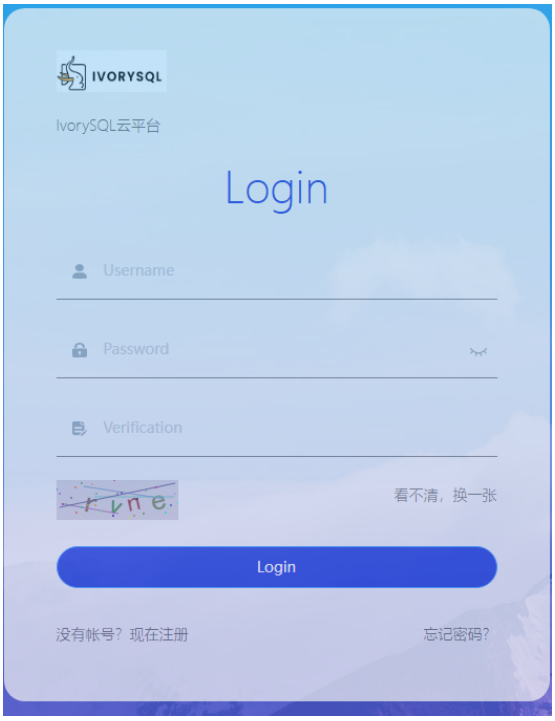
docker.io/prom/prometheus:v2.33.5

docker.io/prom/alertmanager:v0.22.2

docker.io/grafana/grafana:8.5.10
```

..2. IvorySQL Cloud使用

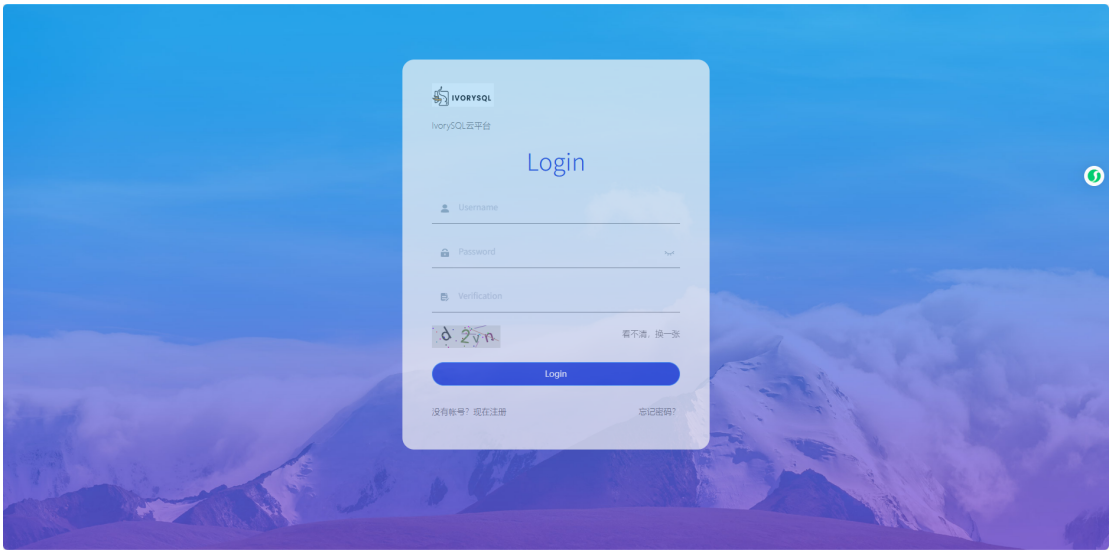
IvorySQL
Cloud是一个基于Web的服务平台，用户可以在任意电脑上通过浏览器进行访问，安装云服务平台的服务器IP为192.168.31.43，然后在浏览器输入http://192.168.31.43:9104/（9104为前端nginx.conf.default中设置的端口）即可进入登录页面页面：



登录和退出

用户登录

进入登录页面，根据提示输入信息即可访问IvorySQL云服务平台：



退出登录

点击页面右上角的头像，会显示当前登录用户名和“Log Out”。点击“Log Out”退出当前登录页面。点击用户名则保留当前登录页面：

管理员功能

添加集群

1. 平台登录admin用户后，点击左侧菜单栏的“K8S集群管理”按钮，进入集群管理页。

2. 点击页面左上角“增加Kubernetes集群”按钮，输入集群信息后提交。

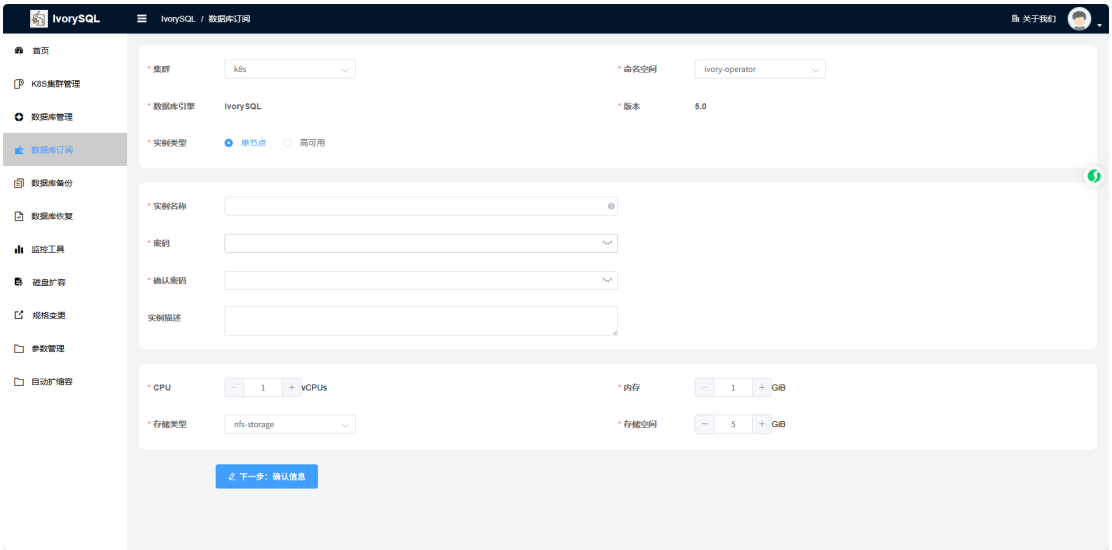
管理集群

在集群管理页面，可以查看已添加集群的详细信息，同时对集群信息进行编辑和删除操作。

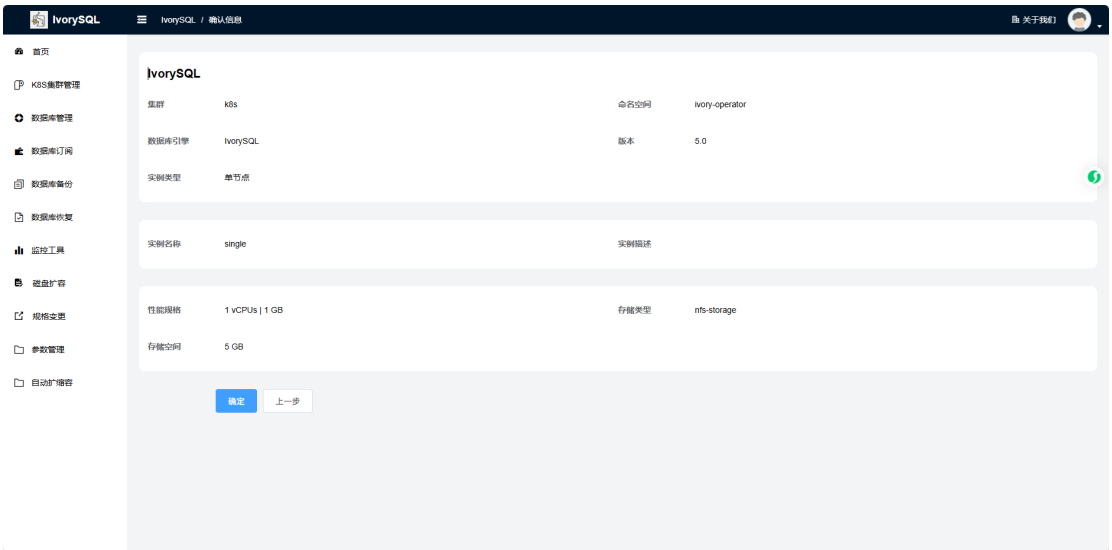
demo用户功能

数据库订阅

1. 使用demo用户登录平台。
2. 点击左侧菜单栏"数据库订阅"按钮，填写需要申请的数据库的各个参数，然后点击"下一步：确认信息"。



3. 检查信息后，点击"确定"



4. "确定"后，会自动跳转到"数据库管理"页面，查看订阅任务

数据库管理

显示云服务平台管理的数据库信息。

数据库重启

1. demo用户登录平台。
2. 点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"重启"选项。

3. 检查信息后，点"确认"

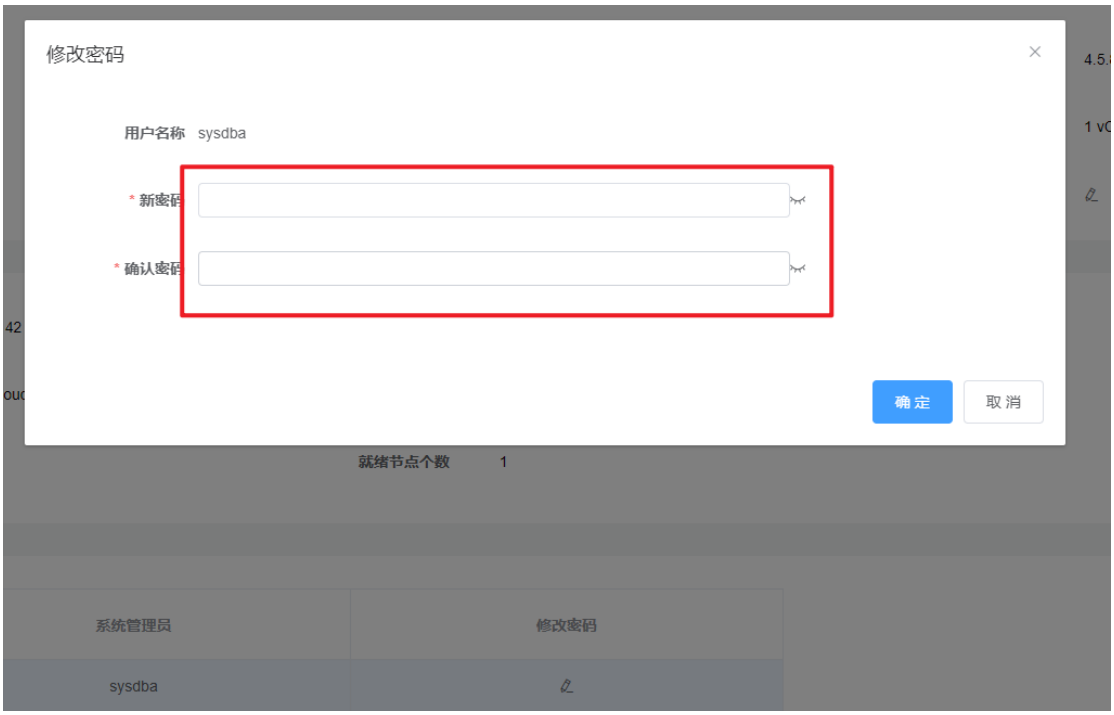
修改密码

1. demo用户登录平台。
2. 点击左侧导航栏"数据库管理"，选中数据库，点击"实例ID"列。

3. 进入数据库详情页面，点击"修改密码"列的图标。

系统管理员	修改密码
sysdba	
syssso	
syssao	

4. 输入新的数据库密码，点击"确定"按钮。



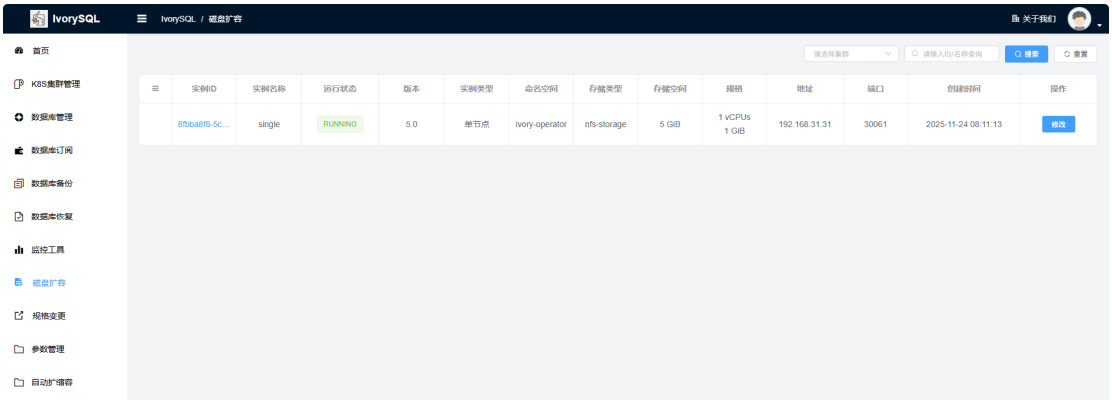
删除实例

1. demo用户登录平台。
2. 点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"删除实例"选项。

3. 检查信息后，点击"确定"按钮。

磁盘扩容

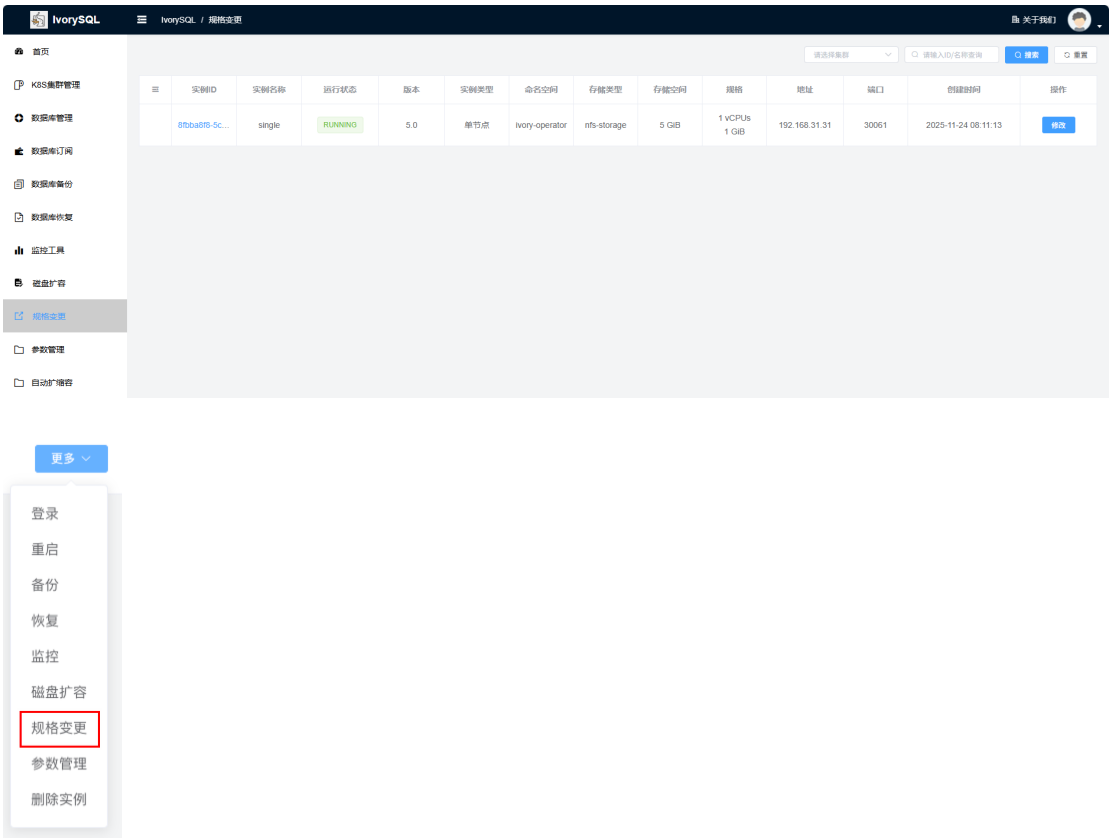
1. 此功能需用户自行配置相关插件，如topolvm
2. demo用户登录平台。
3. 点击左侧导航栏"磁盘扩容"，选中数据库，点击"操作"列的"修改"按钮；或者点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"磁盘扩容"选项。



4. 检查并输入磁盘扩容的大小后，点击"确定"按钮。

规格变更

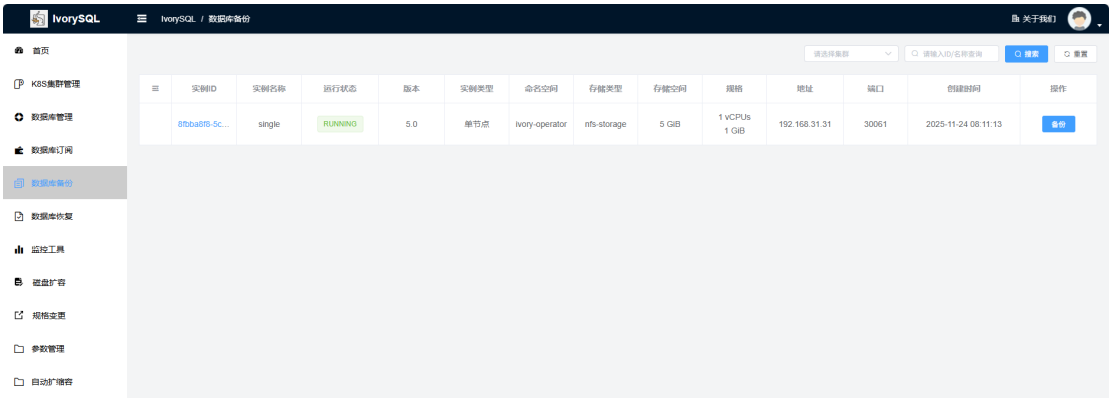
1. demo用户登录平台。
2. 点击左侧导航栏"规格变更"，选中数据库，点击"操作"列的"修改"按钮；或者点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"规格变更"选项。



3. 检查信息并输入规格变更的大小后，点击"确定"按钮。

数据库备份

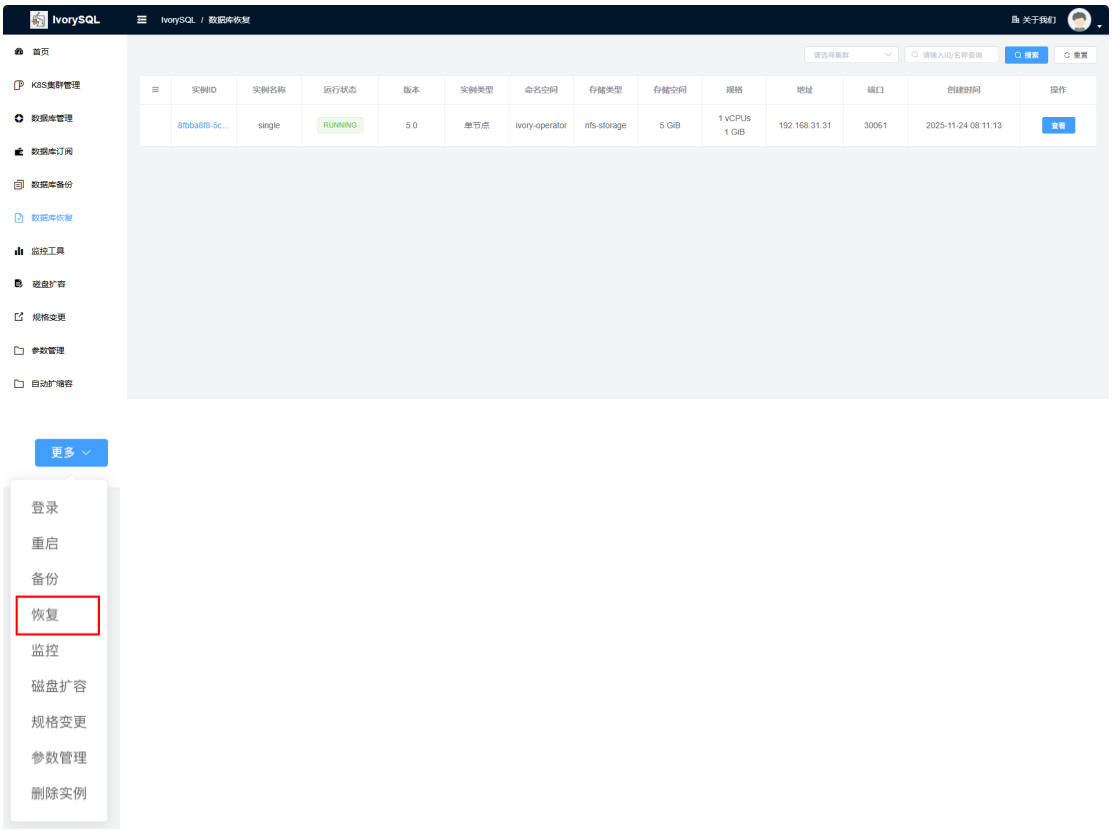
1. demo用户登录平台。
2. 点击左侧导航栏"数据库备份"，选中数据库，点击"操作"列的"备份"；或者点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"备份"选项。



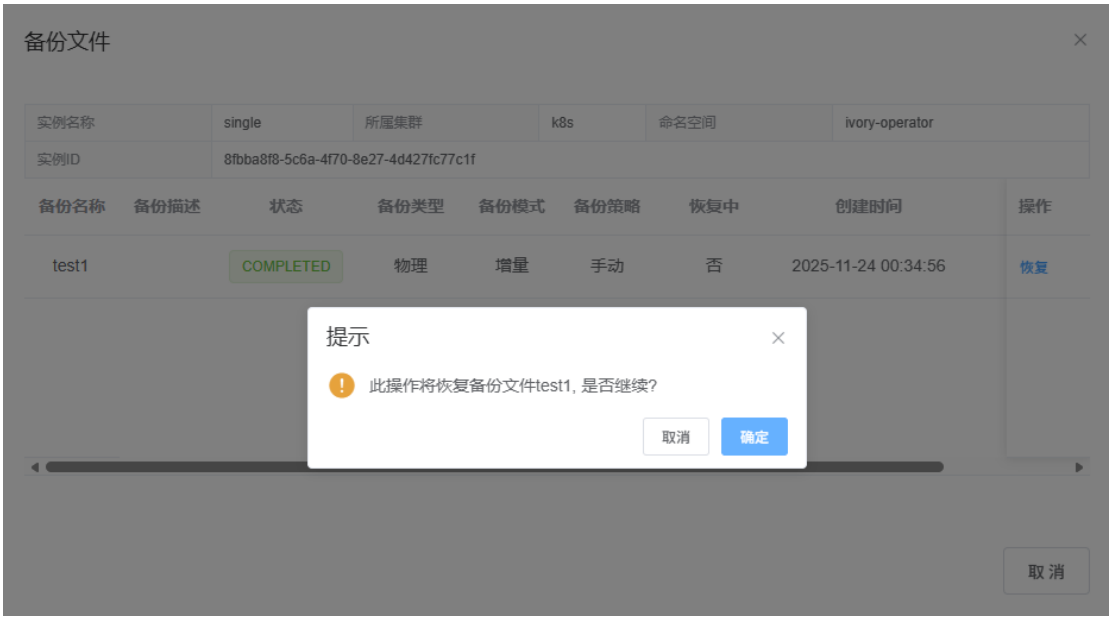
3. 检查信息并输入备份名称后，点击"确定"按钮。

数据库恢复

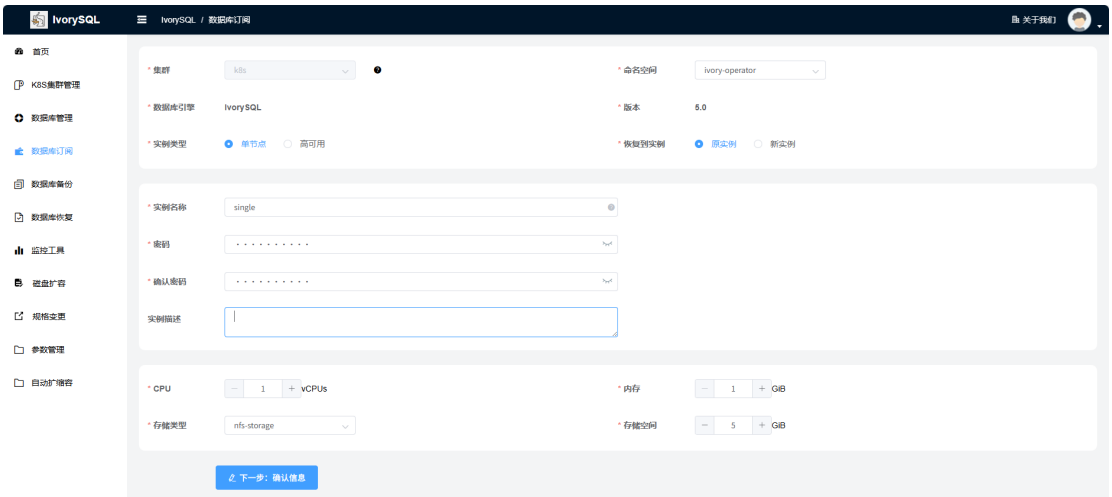
1. demo用户登录平台。
2. 点击左侧导航栏"数据库恢复"，选中数据库，点击"操作"列的"查看"按钮；或者点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"恢复"选项。



3. 选择需要备份的文件后，点"操作"列的"恢复"



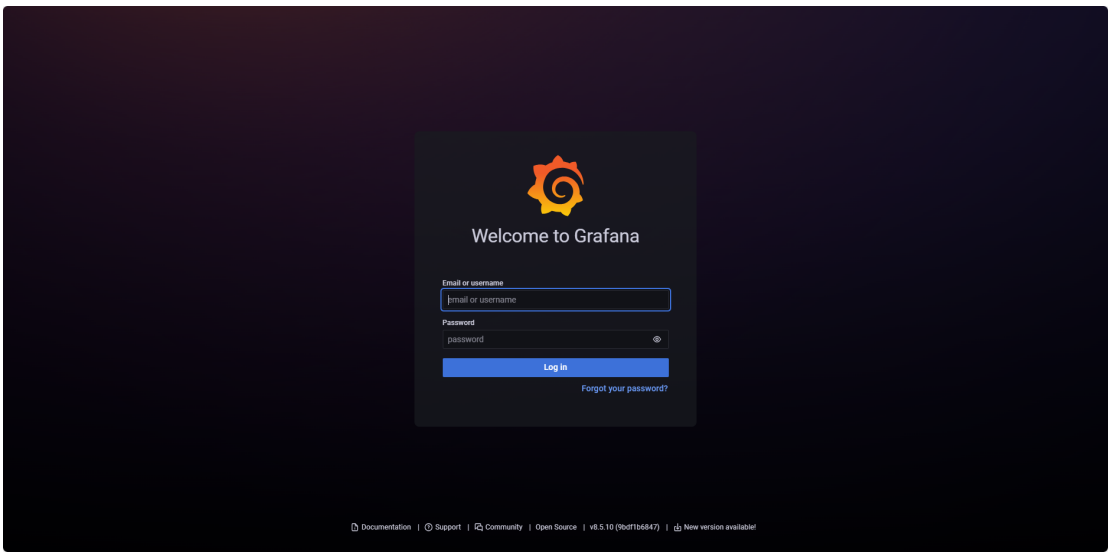
4. 输入需要恢复的数据库信息，密码为备份前的数据库密码



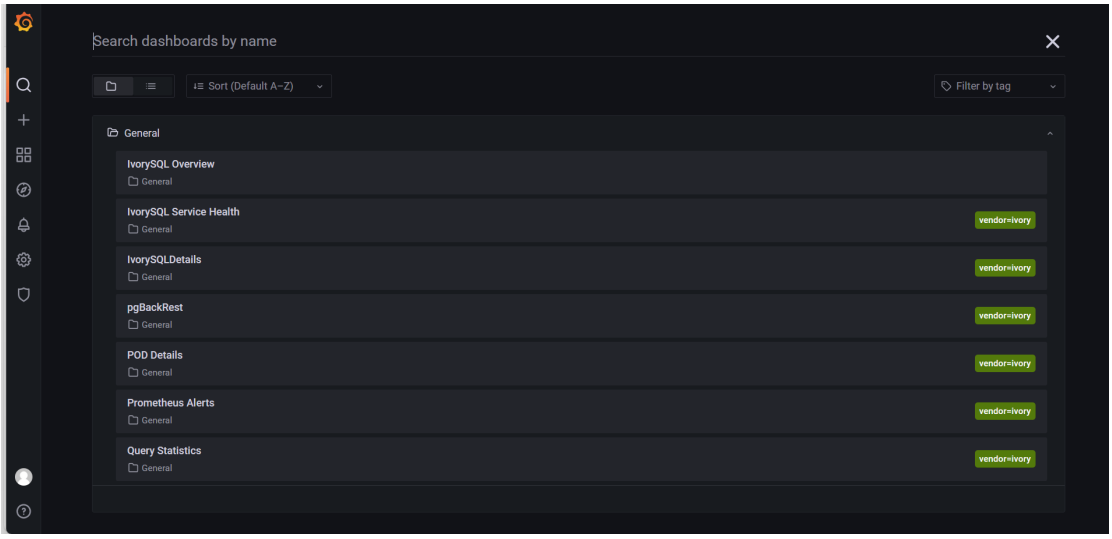
5. 后续参考"4.1数据库订阅"的步骤。

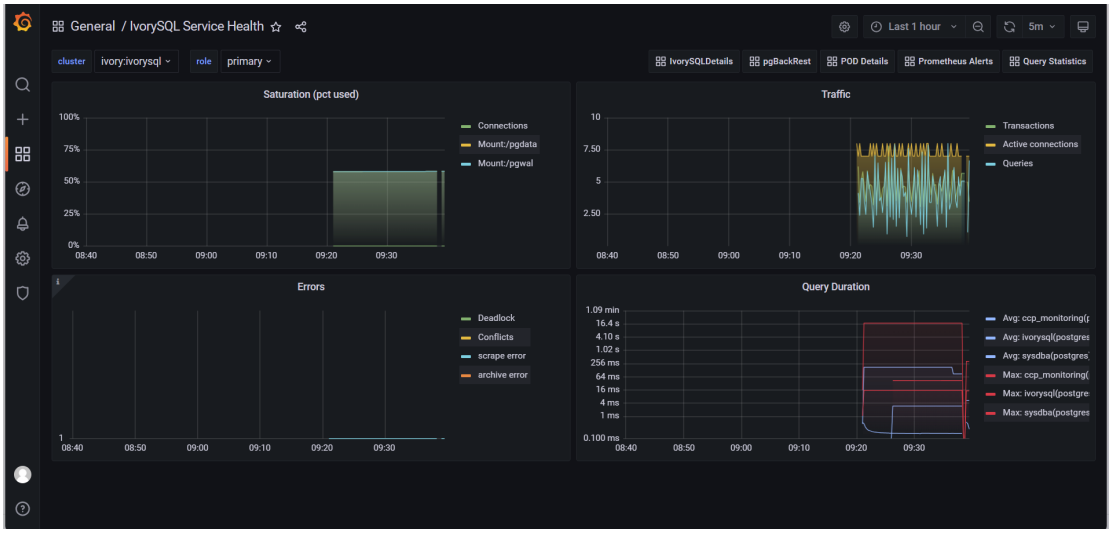
1. demo用户登录平台。
2. 点击左侧导航栏"监控工具"，选择数据库所在的集群；或者点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，在下拉选项中选择"监控"选项。

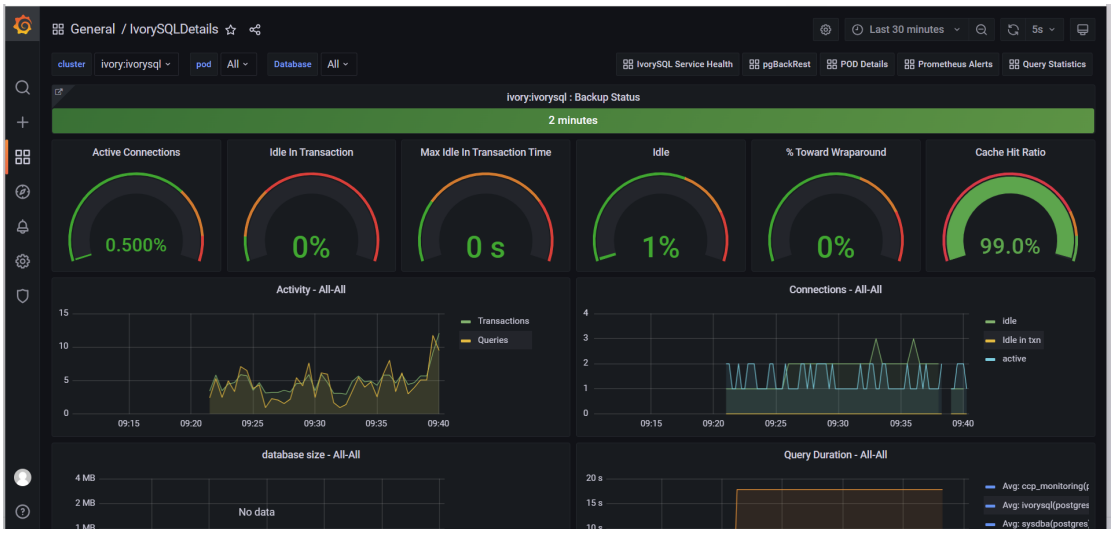
3. 等待监控创建完毕后，再次重复（2）中操作，跳转新页面，输入用户名：admin，密码admin，点击log in登录

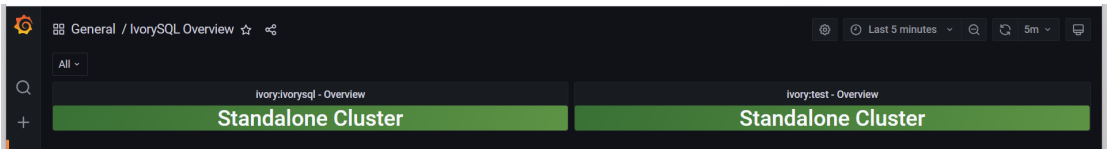


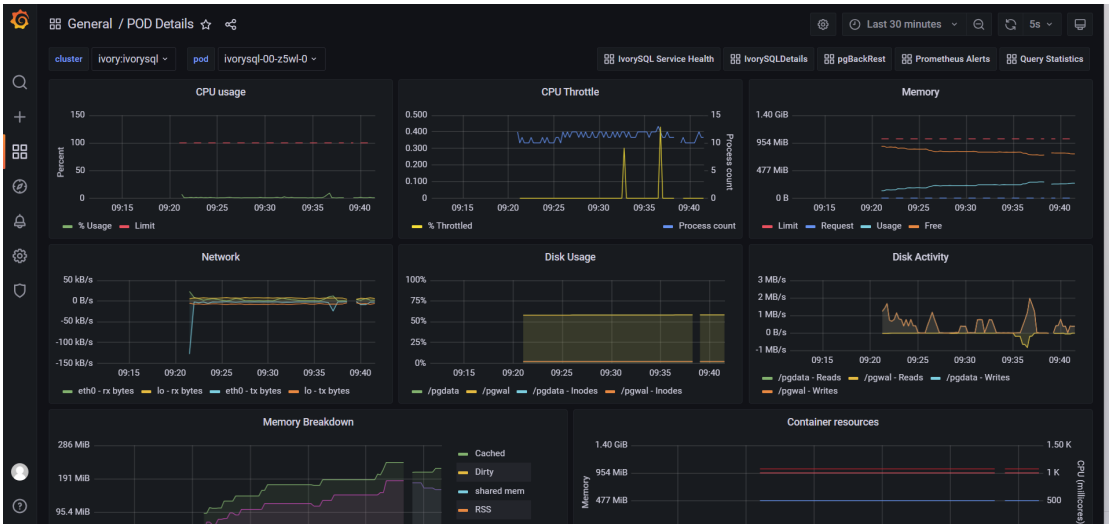
4. 选择放大镜图标，点击查看监控指标。

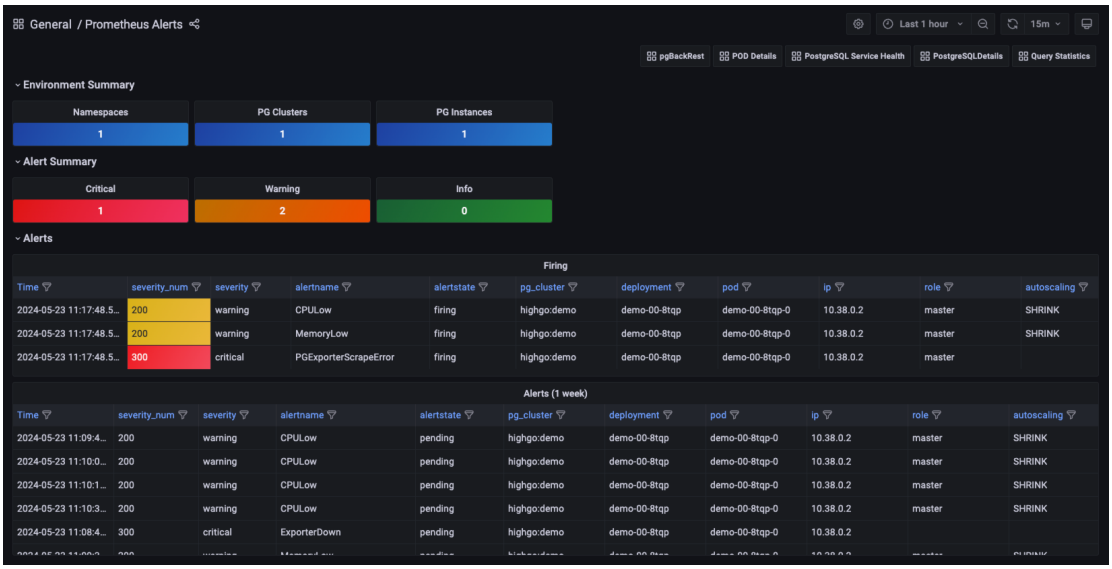

Search dashboards by name
Sort (Default A-Z)
Filter by tag
General
IvorySQL Overview
General
IvorySQL Service Health
General
vendor=ivory
IvorySQLDetails
General
vendor=ivory
pgBackRest
General
vendor=ivory
POD Details
General
vendor=ivory
Prometheus Alerts
General
vendor=ivory
Query Statistics
General
vendor=ivory


General / IvorySQL Service Health
Last 1 hour
5m
cluster
ivory:ivorysql
role
primary
IvorySQLDetails
pgBackRest
POD Details
Prometheus Alerts
Query Statistics
Saturation (pct used)
Connections
Mount:/pgdata
Mount:/pgwal
Traffic
Transactions
Active connections
Queries
Errors
Deadlock
Conflicts
scrape error
archive error
Query Duration


General / IvorySQLDetails
Last 30 minutes
5s
cluster
ivory:ivorysql
pod
All
Database
All
IvorySQL Service Health
pgBackRest
POD Details
Prometheus Alerts
Query Statistics
ivory:ivorysql : Backup Status
2 minutes
Active Connections
0.500%
Idle In Transaction
0%
Max Idle In Transaction Time
0 s
Idle
1%
% Toward Wraparound
0%
Cache Hit Ratio
99.0%
Activity - All-All
Transactions
Queries
Connections - All-All
idle
Idle in txn
active
database size - All-All
No data
Query Duration - All-All


General / IvorySQL Overview
Last 5 minutes
5m
All
ivory:ivorysql - Overview
Standalone Cluster
ivory:test - Overview
Standalone Cluster

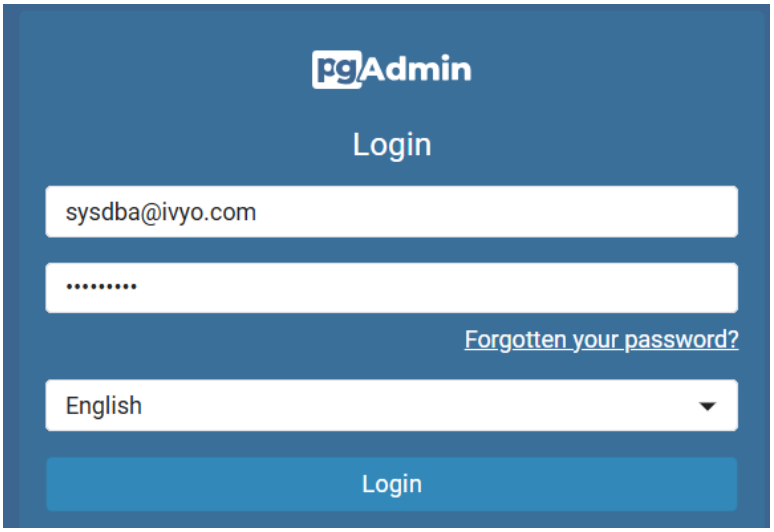




可视化登录工具

1. demo用户登录平台。
2. 点击左侧导航栏"数据库管理"，选中数据库，点击"操作"列的"更多"按钮，选择"登录"选项。

3. 跳转新页面后，输入数据库用户名 sysdba@ivyo.com 及数据库密码，点击"Login"按钮。



4. 连接数据库，即可进行操作。

IvorySQL生态

.1. 芯片架构适配

IvorySQL适配认证如下CPU架构：

序号	架构名称	适配品牌	全平台介质包下载
1	x86_64	Intel、AMD、兆芯、海光	deb, rpm
2	aarch64	飞腾、鲲鹏	deb, rpm
3	mips64el	龙芯3000、龙芯4000	deb, rpm
4	loongarch64	龙芯5000	deb, rpm

.2. 操作系统适配

IvorySQL适配认证如下操作系统：

序号	操作系统名称	操作系统简介	证书查看
1	银河麒麟高级服务器操作系统V11	银河麒麟高级服务器操作系统V11是麒麟软件依托多年技术研发积淀与丰富应用实践经验，为企业级关键业务量身打造的新一代服务器操作系统。产品以高可靠、高可用、高安全、高性能、高扩展为核心优势，在深度融合AI技术的基础上，更以自主创新突破构建起高速网络协议 MPTCP、热补丁管理、智能故障诊断、场景化优化及安全容器镜像仓库等关键技术体系，为党政机关信息化建设、重点行业数字化转型及国家重大工程实施，筑牢安全可信的支撑底座。	 瀚高数据库 KYLINSOFT 麒麟软件 麒麟软件适配认证 瀚高基础软件股份有限公司 IvorySQL 4.5 与 麒麟软件有限公司 银河麒麟高级服务器操作系统V11 在Intel、AMD、海光、兆芯平台 完成兼容性测试，能够达到通用兼容性要求及性能、 可靠性要求，满足用户的关键性应用需求 特此颁发适配认证 瀚高基础软件股份有限公司 麒麟软件有限公司 适配认证专用章 在线查验证书 证书编号：20250919S-034 2025年09月19日 "麒心伙伴"计划

2	openKylin 2.0 SP1	OpenAtom openKylin（简称“openKylin”）是由开放原子开源基金会孵化及运营的开源项目，由基础软硬件企业、非营利性组织、社团组织、高等院校、科研机构和个人开发者共同创立，旨在以“为世界提供与人工智能技术深度融合的开源操作系统”为社区愿景，在开源、自愿、平等、协作的基础上，共同打造全球领先的智能桌面开源操作系统根社区，推动Linux开源技术及其软硬件生态繁荣发展。	
3	OpenAnolis OS (龙蜥操作系统) 23	龙蜥操作系统 Anolis OS 23 是龙蜥社区（OpenAnolis）基于开源生态发展合作倡议，面向上游原生社区独立选型，持续演进并保障兼容性和稳定性的操作系统。Anolis OS 23 基于 Linux Kernel 6.6 LTS 的企业级操作系统，依托 ANCK 6.6 内核深度优化，全面支持海光、飞腾、龙芯（LoongArch）、兆芯等国产芯片及通用 x86_64/ARM64 架构。其针对虚拟化、安全特性及性能优化进行专项增强，并通过分层架构设计与智能调优工具，实现软硬协同性能最大化。同时原生支持 AI 生态组件，提供安全的 AI 容器镜像，加速模型开发与推理部署。在开发工具链方面，集成 GCC 12.3+/LLVM 17、Python 3.11、Rust 等，支持多平台高效开发。桌面生态方面兼容 GNOME、DDE 桌面环境，并通过集成玲珑包管理器满足了多样化场景需求实现生态扩展。Anolis OS 23 支持各类常见应用和国产化应用，助力企业实现高效、安全、可靠的数字化转型。	
4	deppin (深度操作系统) 20	深度操作系统是一个致力于为全球用户提供美观易用、安全可靠的Linux发行版。深度操作系统 20正式版（1002）采取统一的设计风格，从桌面环境和应用进行重新设计，带来焕然一新的视觉感受。底层仓库升级到Debian 10.5，系统安装采用双内核机制（Kernel 5.4、Kernel 5.7），全面提升系统稳定性和兼容性。全新设计的启动器菜单、指纹识别、系统安全增强等，系统部分预装应用升级到最新版本，只为给你更好体验。	

生态组件适配

概述

IvorySQL生态插件适配列表

IvorySQL 作为一款兼容 Oracle 且基于 PostgreSQL 的高级开源数据库，具备强大的扩展能力，支持丰富的生态系统插件。这些插件可以帮助用户在不同场景下增强数据库功能，包括地理信息处理、向量检索、全文搜索、数据定义提取和路径规划等。以下是当前 IvorySQL 官方兼容和支持的主要插件列表：

序号	插件名称	版本	功能描述	适用场景
1	postgis	3.5.4	为 IvorySQL 提供地理空间数据支持，包括空间索引、空间函数和地理对象存储	地理信息系统（GIS）、地图服务、位置数据分析
2	pgvector	0.8.1	支持向量相似性搜索，可用于存储和检索高维向量数据	AI 应用、图像检索、推荐系统、语义搜索
3	pgddl (DDL Extractor)	0.31	提取数据库中的 DDL（数据定义语言）语句，便于版本管理和迁移	数据库版本控制、CI/CD 集成、结构比对与同步
4	pg_cron	1.6.0	提供数据库内部的定时任务调度功能，支持定期执行SQL语句	数据清理、定期统计、自动化维护任务
5	pgsql-http	1.7.0	允许在SQL中发起HTTP请求，与外部Web服务进行交互	数据采集、API集成、微服务调用
6	plpgsql_check	2.8	提供PL/pgSQL代码的静态分析功能，可在开发阶段发现潜在错误	存储过程开发、代码质量检查、调试优化
7	pgroonga	4.0.4	提供非英语语言全文搜索功能，满足高性能应用的需求	中日韩等语言的全文搜索功能
8	pgaudit	18.0	提供细粒度的审计功能，记录数据库操作日志，便于安全审计和合规性检查	数据库安全审计、合规性检查、审计报告生成
9	pgrouting	3.8.0	提供地理空间数据的路由计算功能，支持多种算法和数据格式	地理空间分析、路径规划、物流优化
10	system_stats	3.2	提供用于访问系统级统计信息的函数	系统监控
11	pgtt	4.5	创建、管理与使用Oracle风格临时表	业务开发
12	pgnodemx	1.7	通过 SQL 提供 Linux 节点、进程、文件系统、cgroup 和 Kubernetes 指标	数据库主机监控、容器可观测性和容量分析

这些插件均经过 IvorySQL 团队的测试和适配，确保在 IvorySQL 环境下稳定运行。用户可以根据业务需求选择合适的插件，进一步提升数据库系统的能力和灵活性。

我们也将持续扩展和丰富 IvorySQL 的插件生态，欢迎社区开发者提交新的插件适配建议或代码贡献。如需了解更多每个插件的详细使用方法和最新兼容版本，请参阅各插件对应的文档章节。

..1. postgis

概述

IvorySQL原生100%兼容PostgreSQL,因此，PostGIS可以完美适配IvorySQL。

安装

根据开发环境，用户可从 PostGIS安装 页面选择适合自己的方式进行安装PostGIS。

源码安装

除PostGIS社区提供的安装方式以外，IvorySQL社区也提供了源码安装方式，源码安装环境为 Ubuntu 24.04(x86_64)。

> 请确保环境中已安装了IvorySQL5.6及以上版本

- 安装依赖

```
sudo apt install \
    docbook-xsl-ns \
    gettext \
    libgdal-dev \
    libgeos-dev \
    libjson-c-dev \
    libproj-dev \
    libprotobuf-c-dev \
    libsfcgal-dev \
    libxml2-dev \
    libxml2-utils \
    protobuf-c-compiler \
    xsltproc
```

- 安装 PostGIS

```
$ wget https://download.osgeo.org/postgis/source/postgis-3.5.4.tar.gz
$ tar xvf postgis-3.5.4.tar.gz
$ cd postgis-3.5.4
$ ./configure  --with-pgconfig=/path/to/pg_config   eg: /usr/ivory-
5/bin/pg_config，如果ivorysql安装目录在/usr/ivory-5.
$ make
$ sudo make install
```

> 如出现PGXS报错, 请根据环境中IvorySQL安装路径, 修改—with-pgconfig的参数值。

创建Extension并确认PostGIS版本

psql 连接到数据库，执行如下命令：

```
ivorysql=# CREATE extension postgis;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'postgis';
  name   | default_version | installed_version |                          comment
---------+-----------------+-------------------+
-----------------------------------------------------------
 postgis | 3.5.4           | 3.5.4             | PostGIS geometry and geography
spatial types and functions
(1 row)
```

使用

关于PostGIS的使用，请参阅 PostGIS3.5官方文档

..2. pgvector

概述

向量数据库是生成式人工智能(GenAI)的关键组成部分。pgvector作为PostgreSQL的重要扩展，不仅能够支持高达16000维度的向量计算，还提供了强大的向量操作和索引功能，使得PostgreSQL能够直接转化为高效的向量数据库。由于IvorySQL基于PostgreSQL研发，这使得它具备了与pgvector扩展无缝集成的能力，从而为用户提供了更广泛的数据处理和分析选项。在Oracle兼容模式下，pgvector扩展同样可用，这为Oracle用户使用向量数据库提供了极大的便利，使其能够轻松地迁移和管理数据，实现更高效的业务操作。

原理介绍

IVFFLAT和HNSW是PGVector的两个索引算法

IVFFLAT

IVFFLAT的工作原理是将相似的向量聚类为区域，并建立一个倒排索引，将每个区域映射到其向量。这使得查询可以集中在数据的一个子集上，从而实现快速搜索。通过调整列表和探针参数，ivfflat
可以平衡数据集的速度和准确性，使 PostgreSQL
有能力对复杂数据进行快速的语义相似性搜索。通过简单的查询，应用程序可以在数百万个高维向量中找到与查询向量最近的邻居。对于自然语言处理、信息检索等，ivfflat 是一个比较好的解决方案 在建立 ivfflat
索引时，你需要决定索引中包含多少个 list。每个 list 代表一个 "中心"；这些中心通过 k-means
算法计算而来。一旦确定了所有中心，ivfflat
就会确定每个向量最靠近哪个中心，并将其添加到索引中。当需要查询向量数据时，你可以决定要检查多少个中心，这由 ivfflat.probes 参数决定。这就是 ANN
性能/召回率的结果：访问的中心越多，结果就越精确，但这是以牺牲性能为代价的。

HNSW

HNSW (Hierarchical Navigating Small World)
是一种基于图的索引算法，它由多层的邻近图组成，因此称为分层的 NSW
方法。它会为一张图按规则建成多层导航图，并让越上层的图越稀疏，结点间的距离越远；越下层的图越稠密，结点间的距离越近。HNSW
算法是一种经典的空间换时间的算法，它的搜索质量和搜索速度都比较高，但是它的内存开销也比较大，因为不仅需要将所有的向量都存储在内存中。还需要维护一个图的结构，也同样需要存储。

安装

> 环境中已经安装了IvorySQL5及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

- 设置PG_CONFIG环境变量

```
export PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config
```

- 拉取pg_vector源码

```
git clone --branch v0.8.1 https://github.com/pgvector/pgvector.git
```

- 安装 pgvector

```
cd pgvector


sudo --preserve-env=PG_CONFIG make
sudo --preserve-env=PG_CONFIG make install
```

- 创建pgvector扩展

```
[ivorysql@localhost ivorysql-5]$ psql
psql (18.6)
Type "help" for help.


ivorysql=# create extension vector;
CREATE EXTENSION
```

至此，pgvector扩展安装已完成。 更多用例，请参考 pgvector文档

Oracle兼容性

在IvorySQL Oracle兼容模式下，pgvector扩展同样可以正确运行

> 建议用户使用1521端口进行测试， psql -p 1521

数据类型

```
ivorysql=# CREATE TABLE items5 (id bigserial PRIMARY KEY, name varchar2(20), num
number(20), embedding bit(3));
CREATE TABLE
ivorysql=# INSERT INTO items5 (name, num, embedding) VALUES ('1st oracle data',0,
'000'), ('2nd oracle data', 111, '111');
INSERT 0 2
ivorysql=# SELECT * FROM items5 ORDER BY bit_count(embedding # '101') LIMIT 5;
 id |      name       | num | embedding
----+-----------------+-----+-----------
  2 | 2nd oracle data | 111 | 111
```

```
  1 | 1st oracle data | 0   | 000
```

匿名块

```
ivorysql=# declare
i vector(3) := '[1,2,3]';
begin
raise notice '%', i;
end;
ivorysql-# /
NOTICE:  [1,2,3]
DO
```

存储过程（PROCEDURE）

```
ivorysql=# CREATE OR REPLACE PROCEDURE ora_procedure()
AS
p vector(3) := '[4,5,6]';
begin
raise notice '%', p;
end;
/
CREATE PROCEDURE
ivorysql=# call ora_procedure();
NOTICE:  [4,5,6]
CALL
```

函数（FUNCTION）

```
ivorysql=# CREATE OR REPLACE FUNCTION AddVector(a vector(3), b vector(3))
RETURN vector(3)
IS
BEGIN
RETURN a + b;
END;
/
CREATE FUNCTION
ivorysql=# SELECT AddVector('[1,2,3]','[4,5,6]') FROM DUAL;
 addvector
----------------
 [5,7,9]
(1 row)
```

..3. pgddl(DDL Extractor)

概述

pgddl 是一个专为 PostgreSQL 数据库设计的 SQL 函数扩展，它能够直接从数据库系统目录中生成清晰、格式化的 SQL DDL (数据定义语言) 脚本，例如 CREATE TABLE 或 ALTER FUNCTION。它解决了 PostgreSQL 原生缺乏类似 SHOW CREATE TABLE 命令的问题，让用户无需借助外部工具（如 pg_dump）即可在纯 SQL 环境中轻松获取对象的创建语句。

该扩展通过一组简单的 SQL 函数提供了一套完整的解决方案，其优势包括：仅需使用 SQL 查询即可操作、支持通过 WHERE 子句灵活筛选对象、并能智能处理对象之间的依赖关系，生成包含 Drop 和 Create 步骤的完整脚本。这使得它特别适用于数据库变更管理、升级脚本编写和结构审计等场景。

需要注意的是，ddlx 仍在发展中，可能尚未覆盖所有 PostgreSQL 对象类型和高级选项。生成的脚本建议始终在非生产环境中先行检查和测试，以确保其正确性与安全性。

安装

IvorySQL的安装包里已经集成了pgddl插件，如果使用安装包安装的IvorySQL，通常不需要再手动安装pgddl即可使用。其它安装方式可以参考下面的源码安装步骤。

> 源码安装环境为 Ubuntu 24.04(x86_64)，环境中已经安装了IvorySQL5及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

从https://github.com/lacanoid/pgddl 下载pgddl v0.31代码。

```
cd pgddl
# 设置PG_CONFIG环境变量值为pg_config路径，eg: /usr/local/ivorysql/ivorysql-5/bin/pg_config
make PG_CONFIG=/path/to/pg_config
make PG_CONFIG=/path/to/pg_config install
```

创建Extension并确认ddlx版本

psql 连接到数据库，执行如下命令：

```
ivorysql=# CREATE extension ddlx;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'ddlx';
 name | default_version | installed_version |         comment
------+-----------------+-------------------+-------------------------
 ddlx | 0.31            | 0.31              | DDL eXtractor functions
(1 row)
```

使用

关于pgddl的使用，请参阅 ddlx官方文档

..4. pg_cron

概述

在 PostgreSQL 中运行周期性任务，例如执行
VACUUM或删除旧数据，是一种常见需求。实现这一点的简单方法是配置
cron或其他外部守护进程，使其定期连接到数据库并运行命令。然而，随着数据库越来越多地作为托管服务或独立容器运行，配置和运行一个单独的守护进程通常变得不切实际。此外，很难让您的
cron任务感知故障转移，或者跨集群节点调度任务。

pg_cron 是 PostgreSQL 的开源定时任务扩展，允许直接在数据库内部设置 cron
风格的任务调度，用于自动化数据维护任务(清理，聚合),
数据库健康检查，执行存储过程和自定义函数等操作。它将cron任务存储在表中，周期性任务会随着
PostgreSQL 服务器自动进行故障转移。详情可以参见 pg_cron文档。

安装配置

> 源码安装环境为 Ubuntu
> 24.04(x86_64)，环境中已经安装了IvorySQL5及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

```
# 拉取pg_cron源码
git clone https://github.com/citusdata/pg_cron.git
cd pg_cron
# 将pg_config的路径设置到PATH环境变量里，eg:
export PATH=/usr/local/ivorysql/ivorysql-5/bin/:$PATH
make
make install
```

配置文件 (ivorysql.conf)

```
# 共享预加载扩展
shared_preload_libraries = 'pg_cron'

# 指定任务元数据存储库（默认当前库）
cron.database_name = 'ivorysql'

# 允许的最大并发任务数
cron.max_running_jobs = 5
```

重启服务

```
pg_ctl restart -D ./data -l logfile
```

创建Extension并确认pg_cron版本

psql 连接到数据库，执行如下命令：

```
ivorysql=# CREATE extension pg_cron;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'pg_cron';
   name   | default_version | installed_version |         comment
---------+-----------------+-------------------+---------------------------
 pg_cron | 1.6             |                   |Job scheduler for PostgreSQL
(1 row)
```

核心功能使用

创建定时任务

```
SELECT cron.schedule(
    'nightly-data-cleanup',        -- 任务名称（唯一标识）
    '0 3 * * *',                   -- cron表达式（每天UTC 3:00）
    $$DELETE FROM logs
      WHERE created_at < now() - interval '30 days'$$  -- 执行SQL
);
```

cron表达式速查表：

示例	含义
'0 * * * *'	每小时整点执行
'*/15 * * * *'	每15分钟执行
'0 9 * * 1-5'	工作日早9点执行
'0 1 1 * *'	每月1日凌晨1点执行

pg_cron还允许使用 '$'表示月份的最后一天。

任务管理

```
# 查看所有任务
SELECT * FROM cron.job;
```

```
ivorysql=# select jobid, jobname, schedule, command from cron.job order by jobid;
 jobid |        jobname         |  schedule  |                                  command
-------+------------------------+------------+-------------------------------------------------------------------------------
     3 | delete-job-run-details | 0 12 * * * | DELETE FROM cron.job_run_details WHERE end_time < now() - interval '1 hour'
     4 | last-day-of-month-job1 | 0 11 $ * * | vacuum
     5 |                        | 30 seconds | select 1
(3 rows)
```

```
# 查看任务执行历史
SELECT * FROM cron.job_run_details ORDER BY start_time DESC LIMIT 10;
```

```
ivorysql=# select jobid, job_pid, database, command, status, start_time, end_time from cron.job_run_details order by start_time desc limit 10;
 jobid | job_pid | database | command  |  status   |          start_time           |           end_time
-------+---------+----------+----------+-----------+-------------------------------+-------------------------------
     5 |  222442 | ivorysql | select 1 | succeeded | 2025-08-21 18:25:38.04872+08  | 2025-08-21 18:25:38.049301+08
     5 |  222436 | ivorysql | select 1 | succeeded | 2025-08-21 18:25:08.043366+08 | 2025-08-21 18:25:08.044139+08
     5 |  222426 | ivorysql | select 1 | succeeded | 2025-08-21 18:24:38.038019+08 | 2025-08-21 18:24:38.038593+08
     5 |  222420 | ivorysql | select 1 | succeeded | 2025-08-21 18:24:08.033158+08 | 2025-08-21 18:24:08.033693+08
     5 |  222411 | ivorysql | select 1 | succeeded | 2025-08-21 18:23:38.028386+08 | 2025-08-21 18:23:38.028805+08
     5 |  222407 | ivorysql | select 1 | succeeded | 2025-08-21 18:23:08.023131+08 | 2025-08-21 18:23:08.023676+08
     5 |  222400 | ivorysql | select 1 | succeeded | 2025-08-21 18:22:38.017147+08 | 2025-08-21 18:22:38.01764+08
(7 rows)
```

```
# 删除任务
SELECT cron.unschedule('nightly-data-cleanup');


# 暂停任务（更新状态）
UPDATE cron.job SET active = false WHERE jobname = 'delete-job-run-details';
```

..5. pgsql-http

概述

pgsql-http 是一个为 PostgreSQL 数据库设计的开源扩展，它允许用户直接在数据库内部发起 HTTP 请求，扮演了一个内置 Web 客户端的角色。该扩展的核心目的是打通数据库与外部 Web 服务之间的壁垒，使得通过简单的 SQL 函数调用即可与外部 Web 服务、API 端点进行交互，无需依赖外部应用程序或中间件。

借助此扩展，开发者可以在 SQL 查询、触发器或存储过程中直接获取网络数据（GET）、提交数据（POST/PUT）、更新（PATCH）或删除（DELETE）远程资源。它提供了丰富的功能，包括设置请求头、自动处理 URL 编码、发送 JSON 数据以及解析响应状态、头部和内容，极大简化了将外部数据集成到数据库操作中的流程。

其典型应用场景包括：实时获取外部数据（如汇率、天气信息）并存入表；在数据变更时通过触发器自动通知微服务；对数据库中的数据进行清洗后直接提交至外部 API 等。它为构建以数据库为中心的集成应用提供了强大而灵活的解决方案。

安装

IvorySQL的安装包里已经集成了pgsql-http插件，如果使用安装包安装的IvorySQL，通常不需要再手动安装pgsql-http即可使用。其它安装方式可以参考下面的源码安装步骤。

> 源码安装环境为 Ubuntu 24.04(x86_64)，环境中已经安装了IvorySQL5及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

- 安装依赖

对libcurl有依赖，libcurl的开发文件（例如 libcurl4-openssl-dev）需要提前安装上

```
#安装依赖
sudo apt install libcurl4-openssl-dev
```

- 编译安装

从https://github.com/pramsey/pgsql-http/releases/tag/v1.7.0 下载 1.7.0的源码包 pgsql-http-1.7.0.tar.gz

```
tar xvf pgsql-http-1.7.0.tar.gz
cd pgsql-http-1.7.0
# 确保pg_config在PATH里可以访问，eg: /usr/local/ivorysql/ivorysql-5/bin/pg_config
make
sudo make install
```

创建Extension并确认http版本

psql 连接到数据库，执行如下命令：

```
ivorysql=# CREATE extension http;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'http';
   name    | default_version | installed_version |       comment
-----------+-----------------+-------------------+
-----------------------------------------------------------------------
 http      | 1.7             |  1.7              | HTTP client for PostgreSQL, allows
web page retrieval inside the database.
(1 row)
```

使用

关于pgsql-http的使用，请参阅 pgsql-http官方文档

..6. plpgsql_check

概述

在 PostgreSQL
数据库开发过程中，编写存储过程和函数时往往难以在运行前发现代码中的语法错误、类型不匹配、未定义变量等潜在问题。传统方式需要等到函数实际执行时才能发现这些错误，这不仅增加了调试成本，也可能在生产环境中引发意外故障。

plpgsql_check 是一个专为 PostgreSQL 的 PL/pgSQL
语言设计的静态代码分析工具（Linter），它能够在不实际执行函数的情况下，对存储过程和函数的源代码进行深度检查。该工具可以提前发现语法错误、类型不匹配、未使用的变量、性能问题、安全隐患等多种代码质量问题，帮助开发者在开发阶段就确保代码的正确性和健壮性。详情可以参见
plpgsql_check官方文档。

安装

> 源码安装环境为 Ubuntu
> 24.04(x86_64)，环境中已经安装了IvorySQL5及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

```
# 从https://github.com/okbob/plpgsql_check/releases/tag/v2.8.3 下载 2.8.3的源码包
plpgsql_check-2.8.3.tar.gz
```

```
tar xvf plpgsql_check-2.8.3.tar.gz
cd plpgsql_check-2.8.3
# 将pg_config的路径设置到PATH环境变量里，eg:
export PATH=/usr/local/ivorysql/ivorysql-5/bin/:$PATH
make USE_PGXS=1 clean
make USE_PGXS=1 all
sudo make USE_PGXS=1 install
```

创建Extension并确认plpgsql_check版本

psql 连接到数据库，执行如下命令:

```
ivorysql=# CREATE EXTENSION plpgsql_check;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'plpgsql_check';
     name      | default_version | installed_version |                      comment
---------------+-----------------+-------------------+
-----------------------------------------------------
 plpgsql_check | 2.8             | 2.8               | extended check for plpgsql
functions
(1 row)
```

使用

检查单个函数

```
-- 创建一个示例函数
CREATE OR REPLACE FUNCTION test_function(p_id integer)
RETURNS text AS $$
DECLARE
    v_name text;
    v_unused integer;  -- 未使用的变量
BEGIN
    SELECT name INTO v_name FROM users WHERE id = p_id;
    RETURN v_naem;  -- 拼写错误
END;
$$ LANGUAGE plpgsql;

-- 使用 plpgsql_check 检查函数
SELECT * FROM plpgsql_check_function('test_function(integer)');
```

检查结果示例:

```
                    plpgsql_check_function
----------------------------------------------------------------
 error:42601:7:assignment:target variable "v_naem" is undefined
 warning:00000:4:DECLARE:unused variable "v_unused"
(2 rows)
```

更多详细使用方法和高级特性，请参阅 plpgsql_check官方文档。

..7. pgroonga

概述

PostgreSQL 内置了全文搜索功能，但在处理大规模数据、复杂查询以及非英语语言（特别是中日韩等 CJK 语言） 时，其功能和性能可能无法满足高性能应用的需求。

PGroonga 应运而生，它是一个 PostgreSQL 的扩展插件，将 Groonga 这款高性能的全功能全文搜索引擎与 PostgreSQL 数据库深度融合。Groonga
本身是一个优秀的开源搜索引擎，以其极致的速度和丰富的功能著称，尤其擅长处理多语言文本。PGroonga 的使命就是将 Groonga 的强大能力无缝地带入 PostgreSQL 的世界，为用户提供远超原生全文搜索的体验。

安装

> IvorySQL的安装包里已经集成了PGroonga插件，如果使用安装包安装的IvorySQL，通常不需要再手动安装PGroonga，直接跳过安装步骤即可。

用户可从 PGroonga package安装 页面选择适合自己的方式进行PGroonga安装。

IvorySQL社区提供了源码安装步骤，下面以PGroonga v4.0.4为例进行演示。

安装依赖

环境

```
操作系统: Ubuntu 24.04
CPU架构: x86_64
IvorySQL: v5.6
```

安装 msgpack-c

编译PGroonga，有个配置选项：HAVE_MSGPACK=1，它是用于支持WAL，使用这个选项需要安装msgpack-c 1.4.1或者更高版本。

```
sudo apt install libmsgpack-dev
```

安装 Groonga

确保Groonga >= 14.0.0

```
sudo apt install groonga libgroonga-dev
```

验证Groonga安装成功

```
highgo@ubuntu:~/work/IvorySQL/inst$ groonga --version
  Groonga 15.1.7 [Linux,x86_64,utf8,match-escalation-threshold=0,nfkc,mecab,message-pack,mruby,onigmo,zlib,lz4,zstandard,epoll,apache-arrow,xxhash,blosc,h3,simdjson,llama.cpp]
```

编译安装PGroonga

下载解压源码

```
wget https://packages.groonga.org/source/pgroonga/pgroonga-4.0.4.tar.gz
tar xvf pgroonga-4.0.4.tar.gz
cd pgroonga-4.0.4
```

编译

运行make前确保`pg_config`命令的路径在`PATH`环境变量里，例如IvorySQL安装的路径是`~/work/IvorySQL/inst`，则环境变量设置如下：

```
export PGHOME=~/work/IvorySQL/inst
export PGDATA=$PGHOME/data
export PATH=$PGHOME/bin:$PATH
```

然后执行以下命令进行编译和安装：

```
make HAVE_MSGPACK=1
make install
```

创建Extension并确认PGroonga版本

psql 连接到数据库的pg模式下，执行命令：

```
postgres=# CREATE extension pgroonga;
CREATE EXTENSION
postgres=# SELECT * FROM pg_available_extensions WHERE name = 'pgroonga';
   name   | default_version | installed_version |
comment
----------+-----------------+-------------------+
-------------------------------------------------------------------------------
 pgroonga | 4.0.4           | 4.0.4             | Super fast and all languages
supported full text search index based on Groonga
(1 row)

postgres=# select version();
                                   version
```

```
-----------------------------------------------------------------------------
 PostgreSQL (IvorySQL 5.6) 18.6 on x86_64-linux, compiled by gcc-13.3.0, 64-bit
(1 row)
```

编码兼容性

PGroonga 依赖 Groonga 支持的字符编码，其中不包含 GB18030。因此，在 GB18030 编码的数据库中创建 PGroonga 索引时，索引可能没有生成词元，并导致查询无法返回匹配结果。

使用 PGroonga 的业务应创建 UTF-8 编码的数据库：

```
CREATE DATABASE search_db
  WITH ENCODING 'UTF8'
       LC_COLLATE 'C'
       LC_CTYPE 'C'
       TEMPLATE template0;
```

修改 Groonga 的运行时编码或重建索引不会改变数据库编码。已有数据需要迁移或重新导入 UTF-8 数据库，再创建 PGroonga 索引。

使用

关于PGroonga的使用，请参阅 PGroonga官方文档

..8. pgaudit

概述

PgAudit 是一个审计扩展，可以为 DDL、DML、DCL 等关键操作生成可追溯的日志记录。通过审计日志，数据库管理员能够满足合规要求、及时发现异常行为，并在出现问题时快速定位责任主体与影响范围。

功能特点

- 全面审计能力：记录 `SELECT`、`INSERT`、`UPDATE`、`DELETE`、DDL 命令以及权限变更等操作，帮助构建完整的操作轨迹。
- 灵活的审计维度：支持全局审计、角色审计与对象审计，可按用户、角色、模式或具体操作类型进行精细化配置。
- 平滑集成：复用 PostgreSQL 标准日志子系统，可与 `syslog`、`logrotate` 等工具联动，兼容现有日志采集与分析方案。
- 合规支撑：提供结构化审计日志，便于生成符合金融、政企等行业规范的审计报告。
- 安全增强：通过记录和分析数据库活动，及时发现未授权访问、异常 DML 或潜在数据泄露风险。
- 运维优化：辅助回放操作行为、定位性能瓶颈，支撑 SQL 优化与问题排查。

安装部署

环境准备

- 已安装的 IvorySQL 数据库。
- 编译工具链：`gcc`、`make`、`tar` 等。
- 数据库管理员权限，用于修改 `ivorysql.conf` 并重启数据库实例。

编译安装 PgAudit

以 PgAudit 18.0 为例：

```
wget https://github.com/pgaudit/pgaudit/archive/refs/tags/18.0.tar.gz
tar -xf 18.0.tar.gz
cd pgaudit-18.0
make install USE_PGXS=1 PG_CONFIG=$PGHOME/bin/pg_config
```

上述命令依赖环境变量 `PGHOME` 指向安装好的 IvorySQL 主目录。安装成功后，`pgaudit.so` 会被放置到 IvorySQL 的扩展目录中。

注册扩展前的基础配置

修改 `ivorysql.conf`，启用插件并设置常用参数：

```
shared_preload_libraries = 'pgaudit'   # 需实例重启生效
pgaudit.log = 'read, write, ddl'       # 审计范围示例，可按需调整
```

重启数据库实例，使共享库配置生效。

创建扩展并验证

```
CREATE EXTENSION IF NOT EXISTS pgaudit;
SELECT name,
       default_version,
       installed_version,
       comment
  FROM pg_available_extensions
 WHERE name = 'pgaudit';
```

若返回的 `installed_version` 与期望版本一致，说明扩展安装成功。

使用

执行如下sql示例：

```
CREATE TABLE audit_demo(id serial PRIMARY KEY, info text);
INSERT INTO audit_demo(info) VALUES ('pgaudit test');
SELECT * FROM audit_demo;
UPDATE audit_demo SET info = 'pgaudit update' WHERE id = 1;
DELETE FROM audit_demo WHERE id = 1;
```

在数据库服务器上查看审计日志：

```
tail -f $PGDATA/log/*.log | grep 'AUDIT:'
```

```
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,CREATE
SEQUENCE,SEQUENCE,public.audit_demo_id_seq,"CREATE TABLE audit_demo(id serial PRIMARY
KEY, info text)",<not logged>
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,CREATE
TABLE,TABLE,public.audit_demo,"CREATE TABLE audit_demo(id serial PRIMARY KEY, info
text)",<not logged>
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,CREATE
INDEX,INDEX,public.audit_demo_pkey,"CREATE TABLE audit_demo(id serial PRIMARY KEY,
info text)",<not logged>
2025-10-31 15:56:32.113 CST [11451] LOG:  AUDIT: SESSION,1,1,DDL,ALTER
SEQUENCE,SEQUENCE,public.audit_demo_id_seq,"CREATE TABLE audit_demo(id serial PRIMARY
KEY, info text)",<not logged>
2025-10-31 15:56:32.117 CST [11451] LOG:  AUDIT: SESSION,2,1,WRITE,INSERT,,,INSERT
INTO audit_demo(info) VALUES ('pgaudit test'),<not logged>
2025-10-31 15:56:32.121 CST [11451] LOG:  AUDIT: SESSION,3,1,READ,SELECT,,,SELECT *
FROM audit_demo,<not logged>
2025-10-31 15:56:32.122 CST [11451] LOG:  AUDIT: SESSION,4,1,WRITE,UPDATE,,,UPDATE
audit_demo SET info = 'pgaudit update' WHERE id = 1,<not logged>
2025-10-31 15:56:32.127 CST [11451] LOG:  AUDIT: SESSION,5,1,WRITE,DELETE,,,DELETE
FROM audit_demo WHERE id = 1,<not logged>
```

若想记录参数的值，打开`pgaudit.log_parameter = 'on'`，效果如下：

```
ivorysql=# SHOW pgaudit.log_parameter;
pgaudit.log_parameter
-----------------------
on
(1 row)
```

..9. pgrouting

概述

pgRouting 是一个基于 PostgreSQL/PostGIS 数据库构建的开源地理空间路由扩展库。它为数据库赋予了强大的网络分析功能，使其能够处理复杂的路径规划与图论计算问题，例如计算两点之间的最短路径、执行旅行推销员（TSP）分析或计算服务区范围等。它将路由算法直接嵌入到数据库中，从而避免了在应用层进行复杂的数据传输与计算。

该扩展的核心优势在于能够利用 PostgreSQL 强大的数据管理能力和 PostGIS 丰富的空间函数，直接在数据库内部对空间网络数据执行高效计算。这不仅简化了应用程序的开发流程，还通过减少数据移动大幅提升了大规模网络分析的性能。

pgRouting 广泛应用于物流配送、交通导航、网络分析、城市规划及供应链管理等多个领域。其开源特性吸引了全球开发者持续的贡献与完善，使其成为空间数据库领域进行路径分析和网络求解的首选工具之一。

安装

> 环境中已经安装了IvorySQL5.6及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

- 安装依赖

对perl有依赖，perl一般在装IvorySQL时已经装上了，这里不用再装。 CMake版本要求 >= 3.12, Boost版本 >= 1.56

```
#安装依赖
sudo apt install cmake libboost-all-dev
```

- 编译安装

```
wget https://github.com/pgRouting/pgrouting/releases/download/v3.8.0/pgrouting-
3.8.0.tar.gz
tar xvf pgrouting-3.8.0.tar.gz
cd pgrouting-3.8.0
mkdir build
cd build
cmake .. -DPOSTGRESQL_PG_CONFIG=/path/to/pg_config  # eg:
/usr/local/ivorysql/ivorysql-5/bin/pg_config
make
sudo make install
```

创建Extension并确认ddlx版本

psql 连接到数据库，执行如下命令：

```
ivorysql=# CREATE extension pgrouting;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'pgrouting';
   name    | default_version | installed_version |       comment
-----------+-----------------+-------------------+---------------------
 pgrouting | 3.8.0           |                   | pgRouting Extension
(1 row)
```

使用

关于pgRouting的使用，请参阅 pgRouting官方文档

..10. system_stats

概述

system_stats 是一个 PostgreSQL 扩展，提供用于访问系统级统计信息的函数，可用于监控场景。

安装

源码安装环境为 Ubuntu 24.04(x86_64)，环境中已经安装了IvorySQL5及以上版本，安装路径为/usr/local/ivorysql/ivorysql-5

源码安装

```
# 从 https://github.com/EnterpriseDB/system_stats/releases/tag/v3.2 下载 3.2 的源码包
v3.2.zip
unzip v3.2.zip
cd system_stats-3.2

# 编译安装插件
make PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config
make PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config install
```

创建插件并确认版本

psql 连接到数据库，执行如下命令：

```
ivorysql=# CREATE EXTENSION system_stats;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'system_stats';
     name     | default_version | installed_version |                    comment
--------------+-----------------+-------------------+
-----------------------------------------------
 system_stats | 3.0             | 3.0               | EnterpriseDB system statistics
for PostgreSQL
(1 row)
```

使用

```
ivorysql=> select pg_sys_os_info() from dual;
                              pg_sys_os_info
-----------------------------------------------------------------------
 """"Ubuntu 24.04.1 LTS""                                              +
 ","Linux 6.14.0-29-generic",Ubuntu,"(none)",5568,292,532,x86_64,,864122)
(1 row)




ivorysql=> select pg_sys_cpu_info() from dual;
                                                          pg_sys_cpu_info
```

```
-----------------------------------------------------------------------------------
-----------------------------------------------
 (GenuineIntel,"GenuineIntel model 154 family 6","12th Gen Intel(R) Core(TM) i7-
1260P",,0,2,1,x86_64,2496000000,,,,48,32,1280,18432)
(1 row)


ivorysql=> select pg_sys_cpu_usage_info() from dual;
 pg_sys_cpu_usage_info
-----------------------
 (0,0,0,100,0,0,0,,,,)
(1 row)


ivorysql=> select pg_sys_memory_info() from dual;
                                pg_sys_memory_info
-------------------------------------------------------------------------------
 (4055482368,3911159808,144322560,4055887872,233725952,3822161920,2638602240,,,,,)
(1 row)


ivorysql=> select pg_sys_io_analysis_info() from dual;
                    pg_sys_io_analysis_info
---------------------------------------------------------------
 (loop0,15,0,21504,0,1,0)
 (loop1,1362,0,27916288,0,680,0)
 (loop2,175,0,3354624,0,131,0)
 (loop3,141,0,1827840,0,83,0)
 (loop4,918,0,32752640,0,212,0)
 (loop5,156,0,2617344,0,67,0)
 (loop6,3707,0,131992576,0,828,0)
 (loop7,243,0,4321280,0,95,0)
 (fd0,0,0,0,0,0,0)
 (sda,639286,1580583,26547332096,240138948608,402702,2744653)
 (sda1,561,0,22966272,0,61,0)
 (sda2,636820,1580583,26491053056,240138948608,402316,2744653)
 (sr0,87,0,3219456,0,45,0)
 (sr1,0,0,0,0,0,0)
 (loop8,228,0,2491392,0,120,0)
 (loop10,2333,0,36571136,0,1628,0)
 (loop9,3730,0,66532352,0,3973,0)
 (loop11,87,0,709632,0,50,0)
 (loop12,156,0,1948672,0,52,0)
```

```
 (loop14,2579,0,97446912,0,2083,0)
 (loop13,30,0,102400,0,14,0)
 (loop15,82,0,1189888,0,117,0)
 (loop16,1357,0,28335104,0,500,0)
 (loop17,110,0,2187264,0,93,0)
 (loop18,129,0,2355200,0,28,0)
 (loop19,110,0,2202624,0,55,0)
(26 rows)


ivorysql=> select pg_sys_disk_info() from dual;
                                pg_sys_disk_info
--------------------------------------------------------------------------------
 (/,/dev/sda2,,,ext4,105086115840,50903396352,48797392896,6553600,619735,5933865)
(1 row)


ivorysql=> select pg_sys_load_avg_info() from dual;
 pg_sys_load_avg_info
----------------------
 (0,0,0,)
(1 row)


ivorysql=> select pg_sys_process_info() from dual;
 pg_sys_process_info
---------------------
 (294,1,201,0,0)
(1 row)


ivorysql=> select pg_sys_network_info() from dual;
                            pg_sys_network_info
-----------------------------------------------------------------------------
 (lo,127.0.0.1,1071575,5643,0,0,1071575,5643,0,0,0)
 (ens33,192.168.198.128,619877515,1490961,0,161,5540250106,4345449,0,0,1000)
(2 rows)


ivorysql=> select pg_sys_cpu_memory_by_process() from dual;
                     pg_sys_cpu_memory_by_process
----------------------------------------------------------------------
```

```
 (1,"(systemd)",864793,0,0.34,13840384)
 (2,"(kthreadd)",864793,0,0,0)
 (3,"(pool_workqueue_release)",864793,0,0,0)
 (4,"(kworker/R-rcu_gp)",864793,0,0,0)
......
```

..11. pgtt

概述

pgtt
是一个创建、管理与使用Oracle风格临时表的PostgreSQL插件。这个插件的目标是在PG内核实现临时表之前提供类似特性。

安装

> 源码安装环境为 Ubuntu 24.04(x86_64)，环境中已经安装了IvorySQL，安装路径为/path-to/ivorysql

源码安装

```
# 下载pgtt源代码
wget https://github.com/darold/pgtt/archive/refs/tags/v4.5.tar.gz
tar zxvf v4.5.tar.gz
cd pgtt-4.5

# 编译安装
make PG_CONFIF=/path-to/ivorysql/bin/pg_config
make PG_CONFIF=/path-to/ivorysql/bin/pg_config install
```

创建Extension

psql 连接到数据库，执行如下命令：

```
-- create the extension
CREATE EXTENSION pgtt;
```

载入插件

```
创建插件后还需要载入才能使用
可以使用以下三种方式之一进行载入：

1. 在配置文件中修改 session_preload_libraries
session_preload_libraries = 'pgtt'

2. 在database级别上启用
```

```
ALTER DATABASE mydb SET session_preload_libraries = 'pgtt';

3. 在session中载入
LOAD 'pgtt';

成功载入插件后可以看到 pgtt.enabled 的值为 on
ivorysql=# show pgtt.enabled;
 pgtt.enabled
--------------
 on
(1 row)

search_path 中增加了 pgtt_schema
ivorysql=# SHOW search_path;
         search_path
------------------------------
 "$user", public, pgtt_schema
(1 row)
```

使用

```
创建临时表:
使用这条语句会产生一条警告信息，但是可以忽略。

ivorysql=# CREATE GLOBAL TEMPORARY TABLE test_gtt_table (
    id integer,
    lbl text
) ON COMMIT PRESERVE ROWS;
WARNING:  GLOBAL is deprecated in temporary table creation
LINE 1: CREATE GLOBAL TEMPORARY TABLE test_gtt_table (
               ^
CREATE TABLE

如果不希望产生这条警告信息，可以使用注释符号:
CREATE /*GLOBAL*/ TEMPORARY TABLE test_gtt_table (
    id integer,
    lbl text
) ON COMMIT PRESERVE ROWS;



创建临时表也可以使用 LIKE 子句:
ivorysql=# CREATE /*GLOBAL*/ TEMPORARY TABLE test_gtt_table AS SELECT * FROM
source_table WITH DATA;
```

```
CREATE TABLE AS

在表列上创建索引:
ivorysql=# CREATE INDEX ON test_gtt_table (id);
CREATE INDEX

删除临时表:
ivorysql=# drop table test_gtt_table;
DROP TABLE

添加约束(除了外键):
ivorysql=# CREATE /*GLOBAL*/ TEMPORARY TABLE t2 (
    c1 serial PRIMARY KEY,
    c2 VARCHAR (50) UNIQUE NOT NULL,
    c3 boolean DEFAULT false
);
CREATE TABLE

创建带有外键的表是不允许的:
ivorysql=# CREATE /*GLOBAL*/ TEMPORARY TABLE t1 (c1 integer, FOREIGN KEY (c1)
REFERENCES source (id));
ERROR:  attempt to create referential integrity constraint on global temporary table

ivorysql=# ALTER TABLE t2 ADD FOREIGN KEY (c1) REFERENCES source (id);
ERROR:  attempt to create referential integrity constraint on global temporary table
```

更多详细使用方法和高级特性，请参阅 https://github.com/darold/pgtt 。

..12. pgnodemx

概述

`pgnodemx` 通过 SQL 提供 Linux 操作系统、进程、文件系统、cgroup 和 Kubernetes Downward API 指标。监控程序可以复用数据库连接采集节点及 IvorySQL 进程指标，无需额外部署独立的 exporter 端点。

本文在 Ubuntu 22.04 x86_64 环境中使用 IvorySQL 5.4（PostgreSQL 18.4）和 pgnodemx 1.7 完成验证。

已验证的兼容能力

能力	状态	验证结果
PGXS 编译安装	支持	pgnodemx 1.7 使用 IvorySQL 5.4 头文件和库成功编译
Linux `/proc` 指标	支持	内存、磁盘、进程、网络、负载和挂载信息函数均能返回数据
cgroup 检测	支持，存在限制	`cgroup_mode()` 能识别主机布局；上游 pgnodemx 1.7 不支持读取 hybrid 模式指标
Oracle 兼容会话	支持	切换兼容模式后，版本、内存和磁盘指标查询均成功

能力	状态	验证结果
Kubernetes Downward API	取决于环境	配置的 Downward API 卷存在时可用

前提条件

- IvorySQL 5.4 已安装服务端开发头文件和 PGXS。
- 目标系统为 Linux，并向 IvorySQL 服务进程提供所需的 `/proc` 和 cgroup 文件。
- 已安装 C 编译器和 GNU make。
- pgnodemx 必须使用与 IvorySQL 相同的 OpenSSL 头文件和库进行编译。

> 不要在 IvorySQL 服务进程中混用不同 OpenSSL 主版本。如果加入 pgnodemx 后启动时报 OpenSSL 未定义符号，应在与 IvorySQL 匹配的构建环境中重新编译 pgnodemx。将另一个不兼容的 OpenSSL 库直接链接进扩展不是安全的解决方案。

编译安装

```
git clone --branch v1.7 --depth 1 \
  https://github.com/CrunchyData/pgnodemx.git
cd pgnodemx

make USE_PGXS=1 \
  PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config
sudo make USE_PGXS=1 \
  PG_CONFIG=/usr/local/ivorysql/ivorysql-5/bin/pg_config install
```

编译前确认选中的 `pg_config` 属于 IvorySQL 5.4：

```
/usr/local/ivorysql/ivorysql-5/bin/pg_config --version
```

配置预加载

在 `ivorysql.conf` 中将 pgnodemx 追加到已有的 `shared_preload_libraries`，不要删除 IvorySQL 原有预加载库。

```
shared_preload_libraries = 'liboracle_parser, ivorysql_ora, gb18030_2022, pgnodemx'
```

非 Kubernetes 主机可以关闭 Downward API 访问，避免无意义的启动警告：

```
pgnodemx.kdapi_enabled = off
```

其他相关默认配置如下：

```
pgnodemx.cgroup_enabled = on
pgnodemx.containerized = off
pgnodemx.cgrouproot = '/sys/fs/cgroup'
```

```
pgnodemx.kdapi_path = '/etc/podinfo'
```

修改 shared_preload_libraries 后重启 IvorySQL，并在每个需要提供指标的数据库中创建扩展：

```
CREATE EXTENSION pgnodemx;


SELECT extversion
FROM pg_extension
WHERE extname = 'pgnodemx';
```

预期扩展版本为 1.7。

授予监控权限

pgnodemx 要求调用者属于预定义的 pg_monitor
角色。应将该角色授予专用监控账户，不要让监控程序使用超级用户连接。

```
CREATE ROLE node_monitor LOGIN PASSWORD 'replace-with-a-strong-password';
GRANT pg_monitor TO node_monitor;
GRANT CONNECT ON DATABASE monitoring TO node_monitor;
```

同时应通过 pg_hba.conf、网络访问控制和 TLS 配置限制只有监控系统可以连接。

查询指标

扩展和运行环境信息

```
SELECT pgnodemx_version();
SELECT exec_path();
SELECT openssl_version();
SELECT cgroup_mode();
```

内存、CPU、磁盘和网络信息

```
SELECT * FROM proc_meminfo()
WHERE key IN ('MemTotal', 'MemAvailable');


SELECT * FROM proc_loadavg();
SELECT * FROM proc_cputime();
SELECT * FROM proc_diskstats();
SELECT * FROM proc_network_stats();
```

IvorySQL 进程信息

```
SELECT * FROM proc_pid_cmdline();
```

```
SELECT * FROM proc_pid_io();
SELECT * FROM proc_pid_stat();
```

这些函数读取 IvorySQL
进程可见的操作系统视图。在容器中，查询结果反映的是容器可访问的命名空间和挂载文件系统。

Oracle 兼容模式

会话切换模式后扩展仍然可用：

```
SET ivorysql.compatible_mode = oracle;


SELECT pgnodemx_version() FROM dual;
SELECT val FROM proc_meminfo() WHERE key = 'MemTotal';
SELECT count(*) FROM proc_diskstats();
```

以上三条查询均已在 IvorySQL 5.4 验证通过。

限制与安全建议

- pgnodemx 1.7 支持 cgroup v1（`legacy`）和 cgroup v2（`unified`），不支持混合的 `hybrid` 布局。cgroup 指标不可用时，`/proc` 系列函数仍可使用。
- `pgnodemx.kdapi_path` 不存在时，Kubernetes 函数返回 NULL 或空结果；非 Kubernetes 环境应关闭此功能。
- 主机文件可能包含敏感的进程、挂载和环境变量信息，只应向可信监控角色授予 `pg_monitor`。
- 指标采集会增加 SQL 和文件读取负载，应设置合理轮询间隔，并只查询必要的函数和字段。
- 该扩展仅适用于 Linux，并依赖内核、容器运行时和安全策略暴露的文件。

完整函数列表请参阅 pgnodemx 1.7 官方文档。

监控运维

.1. 日常监控

监控数据活动

标准Unix工具

在大部分 Unix
平台上，IvorySQL会修改由`ps`报告的命令标题，这样个体服务器进程可以被标识。一个显示样例

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: background
writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres:
checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum
launcher
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats
collector
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug
127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl
regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl
regression [local] idle in transaction
```

（`ps`的调用方式随不同的平台而变，但是显示的细节都差不多。这个例子来自于一个最近的 Linux
系统）。列在这里的第一个进程是主服务器进程。为它显示的命令参数是当它被启动时使用的那些。接下来的五个进程是由主进程自动启动的后台工作者进程（如果你已经设置系统为不启动统计收集器，“统计收集器”进程将不会出现；同样“自动清理发动”进程也可以被禁用）。剩余的每一个进程都是一个处理一个客户端连接的服务器进程。每个这种进程都会把它的命令行显示设置为这种形式

```
postgres: user database host activity
```

在该客户端连接的生命期中，用户、数据库以及（客户端）主机项保持不变，但是活动指示器会改变。活动可以是`闲置`（即等待一个客户端命令）、**在事务中闲置**（在一个`BEGIN`块里等待客户端）或者一个命令类型名，例如`SELECT`。还有，如果服务器进程正在等待一个其它会话持有的锁，
`等待中`会被追加到上述信息中。在上面的例子中，我们可以推断：进程 15606 正在等待进程 15610
完成其事务并且因此释放一些锁（进程 15610
必定是阻塞者，因为没有其他活动会话。在更复杂的情况中，可能需要查看pg_locks系统视图来决定谁阻塞了谁）。

如果配置了cluster_name，则集簇的名字 也将会显示在`ps`的输出中：

```
$ psql -c 'SHOW cluster_name'
 cluster_name
```

```
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1:
background writer
...
```

如果你已经关闭了update_process_title，那么活动指示器将不会被更新，进程标题仅在新进程被启动的时候设置一次。 在某些平台上这样做可以为每个命令节省可观的开销，但在其它平台上却不明显。

> 提示
>
> Solaris需要特别的处理。你必需使用`/usr/ucb/ps`而不是`/bin/ps`。
> 你还必需使用两个`w`标志，而不是一个。另外，你对`postgres`命令的最初调用必须用一个比服务器进程提供的短的`ps`状态显示。如果你没有满足全部三个要求，每个服务器进程的`ps`输出将是原始的`postgres`命令行。

统计收集器

IvorySQL的 统计收集器 是一个支持收集和报告服务器活动信息的子系统。
目前，这个收集器可以对表和索引的访问计数，计数可以按磁盘块和个体行来进行。它还跟踪每个表中的总行数、每个表的清理和分析动作的信息。它也统计调用用户定义函数的次数以及在每次调用中花费的总时间。

IvorySQL也支持报告有关系统正在干什么的
动态信息，例如当前正在被其他服务器进程执行的命令以及系统中存在哪些其他连接。
这个功能是独立于收集器进程存在的。

统计收集配置

因为统计收集给查询执行增加了一些负荷，系统可以被配置为收集或不收集信息。这由配置参数控制，它们通常在`postgresql.conf`中设置。

参数track_activities允许监控当前被任意服务器进程执行的命令。

参数track_counts控制是否收集关于表和索引访问的统计信息。

参数track_functions启用对用户定义函数使用的跟踪。

参数track_io_timing启用对块读写次数的监控。

通常这些参数被设置在`postgresql.conf`中，这样它们会应用于所有服务器进程，但是可以在单个会话中使用SET命令打开或关闭它们（为了阻止普通用户对管理员隐藏他们的活动，只有超级用户被允许使用`SET`来改变这些参数）。

统计收集器通过临时文件将收集到的信息传送给其他IvorySQL进程。这些文件被存储在名字由stats_temp_directory参数指定的目录中，默认是`pg_stat_tmp`。为了得到更好的性能，`stats_temp_directory`可以被指向一个基于 RAM 的文件系统来降低物理 I/O
需求。当服务器被干净地关闭时，一份统计数据的永久拷贝被存储在`pg_stat`子目录中，这样在服务器重启后统计信息能被保持。当在服务器启动时执行恢复时（例如立即关闭、服务器崩溃以及时间点恢复之后），所有统计计数器会被重置。

表 1 中列出了一些预定义视图 可以用来显示系统的当前状态。 表 2 中列出了另一些视图可以显示统计收集的结果。你也可以使用底层统计函数来建立自定义的视图。

在使用统计信息监控收集到的数据时，你必须了解这些信息并非是实时更新的。每个独立的服务器进程只在进入闲置状态之前才向收集器传送新的统计计数；因此正在进行的查询或事务并不影响显示出来的总数。同样，收集器本身也最多每`PGSTAT_STAT_INTERVAL`毫秒（缺省为500ms，除非在编译服务器的时候修改过）发送一次新的报告。因此显示的信息总是落后于实际活动。但是由`track_activities`收集的当前查询信息总是最新的。

另一个重点是当一个服务器进程被要求显示任何这些统计信息时，它首先取得收集器进程最近发出的报告并且接着为所有统计视图和函数使用这个快照，直到它的当前事务的结尾。因此只要你继续当前事务，统计数据将会一直显示静态信息。相似地，当任何关于所有会话的当前查询的信息在一个事务中第一次被请求时，这样的信息将被收集。并且在整个事务期间将显示相同的信息。这是一种特性而非缺陷，因为它允许你在该统计信息上执行多个查询并且关联结果而不用担心那些数字会在你不知情的情况下改变。但是如果你希望用每个查询都看到新结果，要确保在任何事务块之外做那些查询。或者，你可以调用`pg_stat_clear_snapshot`()，那将丢弃当前事务的统计快照（如果有）。下一次对统计性信息的使用将导致获取一个新的快照。

一个事务也可以在视图`pg_stat_xact_all_tables`、**pg_stat_xact_sys_tables**、`pg_stat_xact_user_tables`和`pg_stat_xact_user_functions`中看到它自己的统计信息（还没有被传送给收集器）。这些数字并不像上面所述的那样行动，相反它们在事务期间持续被更新。

表 1中显示的动态统计视图中的一些信息是有安全限制的。
普通用户只能看到关于他们自己的会话的所有信息（属于他们是成员的角色的会话）。
在关于其他会话的行中，许多列将为空。
但是，请注意，一个会话的存在和它的一般属性，例如会话用户和数据库，对所有用户都是可见的。
超级用户和内置角色`pg_read_all_stats`的成员可以看到所有会话的所有信息。

表1.动态统计视图

视图名称	描述
pg_stat_activity	每个服务器进程一行，显示与那个进程的当前活动相关的信息，例如状态和当前查询。
pg_stat_replication	每一个 WAL 发送进程一行，显示有关到该发送进程连接的后备服务器的复制的统计信息。
pg_stat_wal_receiver	只有一行，显示来自 WAL 接收器所连接服务器的有关该接收器的统计信息。
pg_stat_subscription	每个订阅至少一行，显示有关该订阅的工作者的信息。
pg_stat_ssl	每个连接（常规的或者复制）一行，显示在这个连接上使用的SSL的信息。
pg_stat_gssapi	每个连接（常规和复制）有一行，显示关于GSSAPI验证和加密的信息。
pg_stat_progress_analyze	每个运行`ANALYZE`的后端(包括自动清理工作者进程)的行，显示当前进度。
pg_stat_progress_create_index	每个后台运行`CREATE INDEX`或`REINDEX`的后端都有一行，显示当前的进度。
pg_stat_progress_vacuum	每个运行着`VACUUM`的后端（包括autovacuum工作者进程）一行，显示当前的进度。
pg_stat_progress_cluster	每个运行着`CLUSTER`或`VACUUM FULL`的后端一行，显示当前进度。
pg_stat_progress_basebackup	每一个WAL发送者进程的行显示一个基础备份，显示当前进度。

表2.已收集统计信息的视图

视图名称	描述
pg_stat_archiver	只有一行，显示有关 WAL 归档进程活动的统计信息。
pg_stat_bgwriter	只有一行，显示有关后台写进程的活动的统计信息。
pg_stat_database	每个数据库一行，显示数据库范围的统计信息。
pg_stat_database_conflicts	每个数据库一行，显示数据库范围的统计信息， 这些信息的内容是关于由于与后备服务器的恢复过程发生冲突而被取消的查询。
pg_stat_all_tables	当前数据库中每个表一行，显示有关访问指定表的统计信息。
pg_stat_sys_tables	和`pg_stat_all_tables`一样，但只显示系统表。
pg_stat_user_tables	和`pg_stat_all_tables`一样，但只显示用户表。
pg_stat_xact_all_tables	和`pg_stat_all_tables`相似，但计数动作只在当前事务内发生，用于生存和死亡行数量的列以及清理和分析动作在此视图中不出现。
pg_stat_xact_sys_tables	和`pg_stat_xact_all_tables`一样，但只显示系统表。
pg_stat_xact_user_tables	和`pg_stat_xact_all_tables`一样，但只显示用户表。
pg_stat_all_indexes	当前数据库中的每个索引一行，显示：表OID、索引OID、模式名、表名、索引名、 使用了该索引的索引扫描总数、索引扫描返回的索引记录数、使用该索引的简 单索引扫描抓取的活表(livetable)中数据行数。 当前数据库中的每个索引一行，显示与访问指定索引有关的统计信息。
pg_stat_sys_indexes	和`pg_stat_all_indexes`一样，但只显示系统表上的索引。
pg_stat_user_indexes	和`pg_stat_all_indexes`一样，但只显示用户表上的索引。
pg_statio_all_tables	当前数据库中每个表一行(包括TOAST表)，显示：表OID、模式名、表名、 从该表中读取的磁盘块总数、缓冲区命中次数、该表上所有索引的磁盘块读取总数、 该表上所有索引的缓冲区命中总数、在该表的辅助TOAST表(如果存在)上的磁盘块读取总数、 在该表的辅助TOAST表(如果存在)上的缓冲区命中总数、TOAST表的索引的磁盘块读 取总数、TOAST表的索引的缓冲区命中总数。 当前数据库中的每个表一行，显示有关在指定表上 I/O 的统计信息。
pg_statio_sys_tables	和`pg_statio_all_tables`一样，但只显示系统表。
pg_statio_user_tables	和`pg_statio_all_tables`一样，但只显示用户表。
pg_statio_all_indexes	当前数据库中每个索引一行，显示：表OID、索引OID、模式名、 表名、索引名、该索引的磁盘块读取总数、该索引的缓冲区命中总数。 当前数据库中的每个索引一行，显示与指定索引上的 I/O 有关的统计信息。
pg_statio_sys_indexes	和`pg_statio_all_indexes`一样，但只显示系统表上的索引。

pg_statio_user_indexes	和`pg_statio_all_indexes`一样，但只显示用户表上的索引。
pg_statio_all_sequences	当前数据库中每个序列对象一行，显示：序列OID、模式名、序列名、序列的磁盘读取总数、序列的缓冲区命中总数。 当前数据库中的每个序列一行，显示与指定序列上的 I/O 有关的统计信息。
pg_statio_sys_sequences	和`pg_statio_all_sequences`一样，但只显示系统序列（目前没有定义系统序列，因此这个视图总是为空）。
pg_statio_user_sequences	和`pg_statio_all_sequences`一样，但只显示用户序列。
pg_stat_user_functions	对于所有跟踪功能，函数的OID，模式，名称，数量通话总时间，和自我的时间。自我时间是 在函数本身所花费的时间量，总时间包括 它调用函数所花费的时间。时间值以毫秒为单位。 每一个被跟踪的函数一行，显示与执行该函数有关的统计信息。
pg_stat_xact_user_functions	和`pg_stat_user_functions`相似，但是只统计在当前事务期间的调用
pg_stat_slru	每个SLRU一行, 显示操作的统计信息。

针对每个索引的统计信息对于判断哪个索引正被使用以及它们的效果特别有用。

`pg_statio_`系列视图主要用于判断缓冲区的效果。当实际磁盘读取数远小于缓冲区命中时，这个缓冲能满足大部分读请求而无需进行内核调用。但是，这些统计信息并没有给出所有的事情：由于IvorySQL处理磁盘 I/O 的方式，不在IvorySQL缓冲区中的数据库仍然驻留在内核的 I/O
缓存中，并且因此可以被再次读取而不需要物理磁盘读取。我们建议希望了解IvorySQL I/O
行为更多细节的用户将IvorySQL统计收集器和操作系统中允许观察内核处理 I/O 的工具一起使用。

pg_stat_activity

`pg_stat_activity`视图每个服务器进程将有一行，显示与该进程当前活动相关的信息。

表3.pg_stat_activity 视图

列	类型	描述
datid	oid	这个后端连接到的数据库的OID
datname	name	这个后端连接到的数据库的名称
pid	integer	这个后端的进程 ID
leader_pid	integer	并行组组长的进程ID，如果该进程是并行查询工作者。如果该进程是一个并行组的组长或不参与并行查询，则为`NULL`。
usesysid	oid	登录到这个后端的用户的 OID
usename	name	登录到这个后端的用户的 名称
application_name	text	连接到这个后端的应用的名称
client_addr	inet	连接到这个后端的客户端的 IP 地址。如果这个字段为空，它表示客户端通过服务器机器上的一个 Unix 套接字连接或者这是一个内部进程，如自动清理。

client_hostname	text	已连接的客户端的主机名，由 client_addr 的反向 DNS 查找报告。 这个字段将只对 IP 连接非空，并且只有log_hostname被启用时才会非空。
client_port	integer	客户端用于与此后端通信的TCP端口号，如果使用Unix套接字，则为`-1`。如果该字段为空，它表示这是一个内部服务器进程。
backend_start	timestamp with time zone	这个进程被启动的时间。对客户端后端来说，这就是客户端连接到服务器的时间。
xact_start	timestamp with time zone	这个进程的当前事务被启动的时间，如果没有活动事务则为空。如果当前查询是它的第一个事务，这一列等于 query_start 列。
query_start	timestamp with time zone	当前活动查询被开始的时间，如果 state 不是 active，则为上一个查询开始的时间
state_change	timestamp with time zone	state 上一次被改变的时间
wait_event_type	text	后端等待的事件类型，如果有的话;否则NULL。
wait_event	text	如果后端当前正在等待，则等待事件名称，否则为NULL。
state	text	这个后端的当前总体状态。可能的值为：active: 后端正在执行一个查询。idle: 后端正在等待一个新的客户端命令。idle in transaction: 后端在一个事务中，但是当前没有正在执行一个查询。idle in transaction (aborted): 这个状态与 idle in transaction 相似，除了在该事务中的一个语句导致了一个错误。fastpath function call: 后端正在执行一个 fast-path 函数。disabled: 如果在这个后端中track_activities被禁用，则报告这个状态。
backend_xid	xid	这个后端的顶层事务标识符，如果存在。
backend_xmin	xid	当前后端的 xmin 范围。
query	text	这个后端最近查询的文本。如果 state 为 active，这个字段显示当前正在执行的查询。在所有其他状态下，它显示上一个被执行的查询。默认情况下，查询文本会被截断至1024个字节，这个值可以通过参数track_activity_query_size更改。

backend_type	text	当前后端的类型。可能的类型为autovacuum launcher, autovacuum worker, logical replication launcher, logical replication worker, parallel worker, background writer, client backend, checkpointer, startup, walreceiver, walsender and walwriter. 除此以外，由扩展注册的后台Worker可能有额外的类型。

wait_event`和`state`列是独立的。如果一个后端处于`active`状态，它可能是也可能不是某个事件上的`waiting。如果状态是`active`并且`wait_event`为非空，它意味着一个查询正在被执行，但是它被阻塞在系统中某处。

表4.等待事件类型

等待事件类型	描述
Activity	服务器进程空闲。此事件类型表示在其主处理循环中等待活动的进程。 `wait_event`将识别特定的等待点。
BufferPin	服务器进程正在等待对数据缓冲的独占访问。 如果另一个进程持有一个打开的游标，该游标最后一次从相关缓冲区读取数据，则缓冲区销等待可能是漫长的。
Client	服务器进程正在等待连接到用户应用程序的套接字上的活动。 因此，服务器预计发生一些独立于其内部进程的事情。`wait_event`将识别特定的等待点。
Extension	服务器进程正在等待扩展模块定义的某个条件。
IO	服务器进程正在等待一个I/O操作完成。`wait_event`将识别特定的等待点。
IPC	服务器进程正在等待与另一个服务器进程进行交互。 `wait_event`将识别特定的等待点。
Lock	服务器进程正在等待一个重量级锁。重量级锁，也称为锁管理器锁或简单锁，主要保护表等SQL可见对象。 然而，它们也用于确保某些内部操作的互斥，例如关系扩展。`wait_event`将识别等待的锁的类型。
LWLock	服务器进程正在等待一个轻量级锁。大多数这样的锁保护共享内存中的特定数据结构。 `wait_event`将包含标识轻量级锁用途的名称。 (有些锁有特定的名称；其他锁是一组锁的一部分，每个锁具有类似的目的。)
Timeout	服务器进程正在等待超时过期。`wait_event`将识别特定的等待点。

表5.`Activity`类型的等待事件

Activity 等待事件	描述
ArchiverMain	在归档进程的主循环中等待。
AutoVacuumMain	在自动清理启动过程的主循环中等待。
BgWriterHibernate	在后台写进程中等待，休眠状态。
BgWriterMain	在后台写进程主循环中等待。

CheckpointerMain	在校验指针进程的主循环中等待。
LogicalApplyMain	在逻辑复制应用进程的主循环中等待。
LogicalLauncherMain	在逻辑复制启动器进程的主循环中等待。
PgStatMain	在统计收集器进程的主循环中等待。
RecoveryWalStream	流恢复期间，在启动进程主循环等待WAL到达。
SysLoggerMain	在syslogger进程的主循环中等待。
WalReceiverMain	在WAL接收器进程的主循环中等待。
WalSenderMain	在WAL发送者进程的主循环中等待。
WalWriterMain	在WAL写入进程的主循环中等待。

表6.`BufferPin`类型的等待事件

BufferPin 等待事件	描述
BufferPin	等待获得缓冲区上的独占销。

表7.`Client`类型的等待事件

Client 等待事件	描述
ClientRead	等待从客户端读取数据。
ClientWrite	等待写入数据到客户端。
GSSOpenServer	在建立GSSAPI会话时等待从客户端读取数据。
LibPQWalReceiverConnect	在WAL接收器等待与远程服务器建立连接。
LibPQWalReceiverReceive	在WAL接收器中等待从远程服务器接收数据。
SSLOpenServer	在尝试连接时等待SSL。
WalReceiverWaitStart	等待启动进程发送用于流复制的初始数据。
WalSenderWaitForWAL	在WAL发送器进程中等待WAL被刷新。
WalSenderWriteData	在WAL发送器进程中处理WAL接收器的回复时，等待任何活动。

表8.`Extension`类型的等待事件

Extension 等待事件	描述
Extension	在扩展中等待。

表9.`IO`类型的等待事件

IO 等待事件	描述
BufFileRead	等待从缓冲文件中读取。
BufFileWrite	等待对缓冲文件的写入。
ControlFileRead	等待读取`pg_control`文件。
ControlFileSync	等待`pg_control`文件到达持久存储。
ControlFileSyncUpdate	等待更新`pg_control`文件以达到持久存储。
ControlFileWrite	等待写入`pg_control`文件。
ControlFileWriteUpdate	等待写入更新`pg_control`文件。
CopyFileRead	在文件复制操作期间等待读取。
CopyFileWrite	在文件拷贝操作期间等待写入。

DSMFillZeroWrite	等待用零填充动态共享内存备份(backing)文件。
DataFileExtend	等待关系数据文件被扩展。
DataFileFlush	等待关系数据文件达到持久存储。
DataFileImmediateSync	等待关系数据文件到持久存储的立即同步。
DataFilePrefetch	等待关系数据文件的异步预取。
DataFileRead	等待对关系数据文件的读取。
DataFileSync	等待对关系数据文件的更改达到持久存储。
DataFileTruncate	等待关系数据文件被截断。
DataFileWrite	等待对关系数据文件的写入。
LockFileAddToDataDirRead	在向数据目录锁文件中添加一行时等待读取。
LockFileAddToDataDirSync	等待数据到达持久存储，同时向数据目录锁文件添加一行。
LockFileAddToDataDirWrite	在向数据目录锁文件中添加一行时等待写操作。
LockFileCreateRead	创建数据目录锁文件时等待读取。
LockFileCreateSync	在创建数据目录锁文件时等待数据到达持久存储。
LockFileCreateWrite	在创建数据目录锁文件时等待写操作。
LockFileReCheckDataDirRead	在重新检查数据目录锁文件期间等待读取。
LogicalRewriteCheckpointSync	等待逻辑重写映射到在检查点到达持久存储。
LogicalRewriteMappingSync	在逻辑重写期间等待映射数据到达持久存储
LogicalRewriteMappingWrite	在逻辑重写期间等待映射数据的写入。
LogicalRewriteSync	等待逻辑重写映射到达持久存储。
LogicalRewriteTruncate	等待在逻辑重写期间截断映射数据。
LogicalRewriteWrite	等待逻辑重写映射的写入。
RelationMapRead	等待关系映射文件的读取。
RelationMapSync	等待关系映射文件到达持久存储。
RelationMapWrite	等待对关系映射文件的写入。
ReorderBufferRead	在重新排序缓冲区管理期间等待读取。
ReorderBufferWrite	在重新排序缓冲区管理期间等待写操作。
ReorderLogicalMappingRead	在重新排序缓冲区管理期间等待读取逻辑映射。
ReplicationSlotRead	等待从复制槽位控制文件读取。
ReplicationSlotRestoreSync	等待复制槽控制文件到达持久存储，同时将其恢复到内存中。
ReplicationSlotSync	等待复制槽控制文件到达持久存储。
ReplicationSlotWrite	等待对复制槽控制文件的写入。
SLRUFlushSync	在检查点或数据库关闭期间等待SLRU数据到达持久存储。
SLRURead	等待读取SLRU页面。
SLRUSync	在写页面后等待SLRU数据到达持久存储。
SLRUWrite	等待SLRU页面的写入。
SnapbuildRead	等待读取序列化的历史目录快照。
SnapbuildSync	等待序列化历史目录快照到达持久存储。
SnapbuildWrite	等待串行历史目录快照的写入。

TimelineHistoryFileSync	等待通过流复制接收的时间线历史文件到达持久存储。
TimelineHistoryFileWrite	等待通过流复制接收的时间线历史文件的写入。
TimelineHistoryRead	等待读取时间线历史文件。
TimelineHistorySync	等待新创建的时间线历史文件到达持久存储。
TimelineHistoryWrite	等待写入新创建的时间线历史文件。
TwophaseFileRead	等待读取两阶段状态文件。
TwophaseFileSync	等待两阶段状态文件到达持久存储。
TwophaseFileWrite	等待对两阶段状态文件的写入。
WALBootstrapSync	在引导过程中等待WAL达到持久存储。
WALBootstrapWrite	在引导过程中等待WAL页面的写入。
WALCopyRead	通过复制一个已有WAL段来创建一个新的WAL段时等待读取。
WALCopySync	等待通过复制一个已有WAL段到持久存储来创建一个新的WAL段。
WALCopyWrite	通过复制一个已有WAL段来创建一个新的WAL段时等待写入。
WALInitSync	等待一个新初始化的WAL文件到持久存储。
WALInitWrite	在初始化一个新的WAL文件时等待写入。
WALRead	等待WAL文件的读取。
WALSenderTimelineHistoryRead	在walsender时间线命令期间等待从时间线历史文件读取。
WALSync	等待WAL文件到达持久存储。
WALSyncMethodAssign	等待数据到达持久存储，同时分配一个新的WAL同步方法。
WALWrite	等待写入WAL文件。

表10.`IPC`类型的等待事件

IPC 等待事件	描述
BackupWaitWalArchive	等待备份所需的WAL文件成功存档。
BgWorkerShutdown	等待后台工作者关闭。
BgWorkerStartup	等待后台工作者启动。
BtreePage	正等待继续并行B-树扫描所需的页号变得可用。
CheckpointDone	等待检查点完成。
CheckpointStart	等待检查点开始。
ExecuteGather	在执行`Gather`计划节点时，等待子进程的活动。
HashBatchAllocate	等待一个选定的并行哈希参与者分配哈希表。
HashBatchElect	等待选择一个并行哈希参与者来分配哈希表。
HashBatchLoad	等待其他并行哈希参与者完成哈希表的加载。
HashBuildAllocate	等待一个选定的并行哈希参与者分配初始哈希表。
HashBuildElect	等待选择一个并行哈希参与者来分配初始哈希表。
HashBuildHashInner	等待其他并行哈希参与者完成内部关系的散列。
HashBuildHashOuter	等待其他Parallel 哈希参与者完成对外部关系的分区。

HashGrowBatchesAllocate	等待选定的并行哈希参与者分配更多批处理。
HashGrowBatchesDecide	等待选择一个并行哈希参与者来决定未来的批处理增长。
HashGrowBatchesElect	等待选择一个Parallel 哈希参与者来分配更多批处理。
HashGrowBatchesFinish	等待一个选定的并行哈希参与者决定未来的批量增长。
HashGrowBatchesRepartition	等待一个选定的并行哈希参与者决定未来的批处理增长。
HashGrowBucketsAllocate	等待选定的并行哈希参与者完成更多bucket的分配。
HashGrowBucketsElect	等待选择一个并行哈希参与者来分配更多的buckets。
HashGrowBucketsReinsert	等待其他Parallel 哈希参与者完成将元组插入到新buckets中。
LogicalSyncData	等待逻辑复制远程服务器发送用于初始表同步的数据。
LogicalSyncStateChange	等待逻辑复制远程服务器更改状态。
MessageQueueInternal	等待另一个进程附加到共享消息队列。
MessageQueuePutMessage	等待将协议消息写入共享消息队列。
MessageQueueReceive	等待从共享消息队列接收字节。
MessageQueueSend	等待将字节发送到共享消息队列。
ParallelBitmapScan	等待并行位图扫描被初始化。
ParallelCreateIndexScan	等待并行`CREATE INDEX`工作者完成堆扫描。
ParallelFinish	等待并行工作人员完成计算。
ProcArrayGroupUpdate	等待组领导在并行操作结束时清除事务ID。
ProcSignalBarrier	等待屏障事件被所有后端处理。
Promote	等待备用系统提升。
RecoveryConflictSnapshot	等待vacuum清理的恢复冲突解决。
RecoveryConflictTablespace	等待恢复冲突解决删除表空间。
RecoveryPause	等待恢复继续进行。
ReplicationOriginDrop	等待复制源变为非活动状态，以便可以删除它。
ReplicationSlotDrop	等待复制槽变为非活动状态，以便可以删除它。
SafeSnapshot	等待获取`READ ONLY DEFERRABLE`事务的有效快照。
SyncRep	在同步复制期间等待远程服务器的确认。
XactGroupUpdate	等待分组组长在并行操作结束时更新事务状态。

表11.`Lock`类型的等待事件

Lock 等待事件	描述
advisory	等待获得一个建议用户锁。
extend	等待扩展一个关系。
frozenid	等待升级 pg_database.datfrozenxid 和 pg_database.datminmxid.
object	等待获取非关系数据库对象上的锁。
page	等待获取一个关系页面上的锁。

relation	等待获得一个关系的锁。
spectoken	等待获取推测的插入锁。
transactionid	等待事务完成。
tuple	等待获取元组上的锁。
userlock	等待获取用户锁。
virtualxid	等待获取虚拟事务ID锁。

表12.`LWLock`类型的等待事件

LWLock 等待事件	描述
AddinShmemInit	等待管理共享内存中的扩展空间分配。
AutoFile	等待更新`postgresql.auto.conf`文件。
Autovacuum	等待读取或更新自动清理工作者的当前状态。
AutovacuumSchedule	等待确保选择为自动清理的表仍然需要清理。
BackgroundWorker	等待读取或更新后台工作者状态。
BtreeVacuum	等待读取或更新b-树索引的清理相关信息。
BufferContent	等待访问内存中的数据页。
BufferIO	等待数据页上的I/O。
BufferMapping	等待将数据块与缓冲池中的缓冲区关联。
Checkpoint	等待开始一个检查点。
CheckpointerComm	等待管理fsync请求。
CommitTs	等待读取或更新事务提交时间戳的最后一个值集。
CommitTsBuffer	在提交时间戳SLRU缓冲区上等待I/O。
CommitTsSLRU	等待访问提交时间戳SLRU缓存。
ControlFile	等待读取或更新`pg_control`文件或创建一个新的WAL文件。
DynamicSharedMemoryControl	等待读取或更新动态共享内存分配信息。
LockFastPath	等待读取或更新进程的快速路径锁信息。
LockManager	等待读取或更新关于“heavyweight”锁。
LogicalRepWorker	等待读取或更新逻辑复制工作器的状态。
MultiXactGen	等待读取或更新共享的multixact状态。
MultiXactMemberBuffer	在multixact成员SLRU缓冲区上等待I/O。
MultiXactMemberSLRU	等待访问multixact成员SLRU缓存。
MultiXactOffsetBuffer	在multixact 偏移 SLRU缓冲区上等待I/O。
MultiXactOffsetSLRU	等待访问multixact 偏移 SLRU缓存。
MultiXactTruncation	等待读取或截断multixact信息。
NotifyBuffer	在`NOTIFY` 消息 SLRU缓冲区上等待I/O。
NotifyQueue	等待读取或更新`NOTIFY` 消息。
NotifyQueueTail	等待`NOTIFY`消息存储上的更新限制。
NotifySLRU	等待访问`NOTIFY`消息SLRU缓存。
OidGen	等待分配一个新的OID。
OldSnapshotTimeMap	等待读取或更新旧的快照控制信息。
ParallelAppend	在并行附加计划执行期间等待选择下一个子计划。

ParallelHashJoin	在并行哈希连接计划执行期间等待同步工作器。
ParallelQueryDSA	等待并行查询动态共享内存分配。
PerSessionDSA	等待并行查询动态共享内存分配。
PerSessionRecordType	等待访问有关复合类型的并行查询信息。
PerSessionRecordTypmod	等待访问有关标识匿名记录类型的类型修饰符的并行查询信息。
PerXactPredicateList	在并行查询期间等待访问当前可序列化事务持有的谓词锁列表。
PredicateLockManager	等待访问可序列化事务使用的谓词锁信息。
ProcArray	等待访问每个进程共享的数据结构(通常情况，是获取快照或报告会话的事务ID)。
RelationMapping	等待读取或更新`pg_filenode.map`文件(用于跟踪某些系统目录的文件节点分配)。
RelCacheInit	等待读取或更新`pg_internal.init`关系缓存初始化文件。
ReplicationOrigin	等待创建、删除或使用复制源。
ReplicationOriginState	等待读取或更新一个复制源的进度。
ReplicationSlotAllocation	等待分配或释放复制槽。
ReplicationSlotControl	等待读取或更新复制槽状态。
ReplicationSlotIO	在复制槽位上等待I/O。
SerialBuffer	在可串行事务冲突的SLRU缓冲区上等待I/O。
SerializableFinishedList	等待访问已完成的可序列化事务列表。
SerializablePredicateList	等待访问可序列化事务持有的谓词锁列表。
SerializableXactHash	等待读取或更新关于可序列化事务的信息。
SerialSLRU	等待访问可序列化事务冲突SLRU缓存。
SharedTidBitmap	在并行位图索引扫描期间等待访问共享的TID位图。
SharedTupleStore	在并行查询期间等待访问共享元组存储。
ShmemIndex	等待在共享内存中找到或分配空间。
SInvalRead	等待从共享目录失效队列中检索消息。
SInvalWrite	等待向共享编目失效队列添加消息。
SubtransBuffer	在子事务SLRU缓冲区上等待I/O。
SubtransSLRU	等待访问子事务SLRU缓存。
SyncRep	等待读取或更新有关同步复制状态的信息。
SyncScan	等待选择同步表扫描的起始位置。
TablespaceCreate	等待创建或删除表空间。
TwoPhaseState	等待读取或更新已准备事务的状态。
WALBufMapping	等待在WAL缓冲区中替换一个页面。
WALInsert	等待将WAL数据插入内存缓冲区。
WALWrite	等待WAL缓冲区写入磁盘。
WrapLimitsVacuum	等待更新事务id和multixact消费的限制。
XactBuffer	在事务状态的SLRU缓冲区上等待I/O。
XactSLRU	等待访问事务状态的SLRU缓存。
XactTruncation	等待执行`pg_xact_status`或更新它可用的最早的事务ID。

XidGen	等待分配新的事务ID。

表13.`Timeout`类型的等待事件

Timeout 等待事件	描述
BaseBackupThrottle	当有限流活动时在基础备份期间等待。
PgSleep	由于调用`pg_sleep`或同类函数而等待。
RecoveryApplyDelay	由于延迟设置，在恢复期间等待应用WAL。
RecoveryRetrieveRetryInterval	当WAL数据无法从任何来源(pg_wal，存档或流)获得时，在恢复期间等待。
VacuumDelay	在一个基于代价的清理延迟点。

下面的例子展示了如何查看等待事件：

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event is NOT
NULL;
 pid  | wait_event_type | wait_event
------+-----------------+------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArray
(2 rows)
```

pg_stat_replication

`pg_stat_replication`视图将在每个WAL发送方进程中包含一行，显示关于复制到发送方连接的后备服务器的统计信息。 只有直接连接的备用设备被列出;没有关于下游后备服务器的信息。

表14.pg_stat_replication 视图

列	类型	描述
pid	integer	一个 WAL 发送进程的进程 ID
usesysid	oid	登录到这个 WAL 发送进程的用户的 OID
usename	name	登录到这个 WAL 发送进程的用户的名称
application_name	text	连接到这个 WAL 发送进程的应用的名称
client_addr	inet	连接到这个 WAL 发送进程的客户端的 IP 地址。如果这个域为空，它表示该客户端通过服务器机器上的一个Unix 套接字连接。
client_hostname	text	连接上的客户端的主机名，由一次对`client_addr`的逆向 DNS 查找报告。 这个域将只对 IP 连接非空，并且只有在log_hostname被启用时非空。
client_port	integer	客户端用来与这个 WAL 发送进程通讯的 TCP 端口号，如果使用 Unix 套接字则为`-1`

backend_start	timestamp with time zone	这个进程开始的时间，即客户端是何时连接到这个WAL 发送进程的。
backend_xmin	xid	由hot_standby_feedback报告的这个后备机的`xmin`水平线。
state	text	当前的 WAL 发送进程状态。 可能的值是：startup: 这个WAL发送器正在启动。catchup: 这个WAL发送者连接的后备服务器正在赶上主服务器。streaming: 在其连接的后备服务器赶上主服务器之后，这个WAL发送方正在流化变化。backup: 这个WAL发送器正在发送一个备份。stopping: 这个WAL发送器正在停止。
sent_lsn	pg_lsn	在这个连接上发送的最后一个预写式日志的位置
write_lsn	pg_lsn	被这个后备服务器写入到磁盘的最后一个预写式日志的位置
flush_lsn	pg_lsn	被这个后备服务器刷入到磁盘的最后一个预写式日志的位置
replay_lsn	pg_lsn	被重放到这个后备服务器上的数据库中的最后一个预写式日志的位置
write_lag	interval	从本地刷新近期的WAL与接收到此后备服务器已写入WAL的通知(但尚未刷新或应用它)之间的时间经过。 如果将此服务器配置为同步后备服务器，则可以使用此参数来衡量在提交时`synchronous_commit`级别`remote_write`所导致的延迟。
flush_lag	interval	在本地刷写近期的WAL与接收到后备服务器已经写入并且刷写它（但还没有应用）的通知之间流逝的时间。 如果这台服务器被配置为一个同步后备，这可以用来计量在提交时`synchronous_commit`的级别`on`所导致的延迟。
replay_lag	interval	在本地刷写近期的WAL与接收到后备服务器已经写入它、刷写它并且应用它的通知之间流逝的时间。 如果这台服务器被配置为一个同步后备，这可以用来计量在提交时`synchronous_commit`的级别`remote_apply`所导致的延迟。
sync_priority	integer	在基于优先的同步复制中，这台后备服务器被选为同步后备的优先级。在基于规定数量的同步复制中，这个值没有效果。

sync_state	text	这一台后备服务器的同步状态。可能的值是：async: 这台后备服务器是异步的。potential: 这台后备服务器现在是异步的，但可能在当前的同步后备失效时变成同步的。sync: 这台后备服务器是同步的。quorum: 这台后备服务器被当做规定数量后备服务器的候选。
reply_time	带时区的时间戳	从后备服务器收到的最后一条回复信息的发送时间

`pg_stat_replication`视图中报告的滞后时间近期的WAL被写入、刷写并且重放以及发送器知道这一切所花的时间的度量。如果远程服务器被配置为一台同步后备，这些时间表示由每一种同步提交级别所带来（或者是可能带来）的提交延迟。对于一台异步后备，`replay_lag`列是最近的事务变得对查询可见的延迟时间的近似值。如果后备服务器已经完全追上了发送服务器并且没有WAL活动，在短时间内将继续显示最近测到的滞后时间，再然后就会显示为NULL。

对于物理复制会自动测量滞后时间。逻辑解码插件可能会选择性地发出跟踪消息，如果它们没有这样做，跟踪机制将把滞后显示为NULL。

> 注意
>
> 报告的滞后时间并非按照当前的重放速率该后备还有多久才能追上发送服务器的预测。在新的WAL被生成期间，这样一种系统将显示类似的时间，但是当发送器变为闲置时会显示不同的值。特别是当后备服务器完全追上时，`pg_stat_replication`显示的是写入、刷写及重放最近报告的WAL位置所花的时间而不是一些用户可能预期的零。这种做法与为近期的写事务测量同步提交和事务可见性延迟的目的一致。为了降低用户预期一种不同的滞后模型带来的混淆，在一个完全重放完的闲置系统上，lag列会在一段比较短的时间后回复成NULL。监控系统应该选择将这种情况表示为缺失数据、零或者继续显示最近的已知值。

pg_stat_wal_receiver

`pg_stat_wal_receiver`事务只包含一行，它显示了从 WAL 接收器所连接的服务器得到的有关该接收器的统计信息。

表15.pg_stat_wal_receiver 视图

列	类型	描述
pid	integer	WAL接收器进程的进程ID
status	text	WAL接收进程的活动状态
receive_start_lsn	pg_lsn	WAL接收器启动时使用的第一个写前日志位置
receive_start_tli	integer	WAL接收器启动时使用的第一个时间线数字
written_lsn	pg_lsn	已经接收并写入磁盘的最后一个预写式日志位置，但没有刷入。这不能用于数据完整性检查。
flushed_lsn	pg_lsn	已经接收并刷入到磁盘的最后一个预写式日志位置，该字段的初始值是启动WAL接收器时使用的第一个日志位置

received_tli	integer	接收并刷入到磁盘的最后一个预写式日志位置的时间线数字，该字段的初始值为启动WAL接收器时使用的第一个日志位置的时间线数字
last_msg_send_time	timestamp with time zone	从源头WAL发送器收到的最后一条信息的发送时间
last_msg_receipt_time	timestamp with time zone	从源头WAL发送器收到的最后一条信息的接收时间
latest_end_lsn	pg_lsn	向源头WAL发送器报告的最后的预写式日志位置
latest_end_time	timestamp with time zone	向源头WAL发送方报告的最后一次写前日志位置的时间
slot_name	text	这个WAL接收器使用的复制槽的名称
sender_host	text	这个WAL接收器连接到的IvorySQL实例的主机。这可以是主机名、IP地址，或者目录路径，如果连接是通过Unix套接字进行的。(路径的情况可以区分，因为它总是以/开头的绝对路径。)
sender_port	integer	这个WAL接收器连接的IvorySQL实例的端口号。
conninfo	text	这个WAL接收器使用的连接字符串，对安全敏感的字段进行了模糊处理。

pg_stat_subscription

每一个订阅的主工作者都在`pg_stat_subscription`视图中有一行（如果工作者没有运行则PID为空），处理被订阅表的初始数据拷贝操作的工作者还会有额外的行。

表16.pg_stat_subscription 视图

列	类型	描述
subid	oid	订阅的OID
subname	name	订阅的名称
pid	integer	订阅工作者进程的进程ID
relid	oid	工作器正在同步的关系的OID;Null用于主应用工作器
received_lsn	pg_lsn	接收到的最后一个预写式日志位置，该字段的初始值为0
last_msg_send_time	timestamp with time zone	从WAL发送器收到的最后一条信息的发送时间
last_msg_receipt_time	timestamp with time zone	从WAL发送器收到的最后一条信息的接收时间
latest_end_lsn	pg_lsn	向WAL发送器报告的最后预写式日志位置
latest_end_time	timestamp with time zone	向WAL发送器报告的最后一次预写式日志位置的时间

pg_stat_ssl

`pg_stat_ssl`视图将为每一个后端或者 WAL 发送进程包含一行，用来显示这个连接上的 SSL 使用情况。

可以把它与`pg_stat_activity`或者`pg_stat_replication`通过`pid`列连接来得到更多有关该连接的细节。

表17.pg_stat_ssl 视图

列	类型	描述
pid	integer	后端或WAL发送器进程ID
ssl	boolean	如果在此连接上使用SSL，则为真
version	text	使用SSL的版本，如果此连接上没有使用SSL则为NULL
cipher	text	正在使用的SSL密码的名称，如果此连接上没有使用SSL则为NULL
bits	integer	使用的加密算法中的位数，如果此连接上没有使用SSL则为NULL
compression	boolean	如果使用SSL压缩则为真，否则为假，如果此连接未使用SSL则为NULL
client_dn	text	区别名称(DN，Distinguished Name)字段与使用的客户端证书，如果没有提供客户端证书或在此连接上没有使用SSL，则为NULL。如果DN字段长于NAMEDATALEN(标准构建中为64个字符)，则该字段将被截断。
client_serial	numeric	客户端证书的序列号，如果没有提供客户端证书或在此连接上没有使用SSL，则为NULL。证书序列号和证书颁发者的组合唯一标识一个证书(除非颁发者错误地重用序列号)。
issuer_dn	text	客户端证书颁发者的区别名称(DN，Distinguished Name)，如果没有提供客户端证书或在此连接上没有使用SSL，则为NULL。该字段像client_dn一样被截断。

pg_stat_gssapi

`pg_stat_gssapi`视图将包含每一个后端一个行，显示该连接上的GSSAPI使用情况。
它可以加入到`pg_stat_activity`或`pg_stat_replication`上的`pid`列，获取更多关于连接的详细信息。

表18.pg_stat_gssapi 视图

列	类型	描述
pid	integer	后端进程ID
gss_authenticated	boolean	如果此连接使用了GSSAPI身份验证，则为True
principal	text	用于验证此连接的主体，如果未使用GSSAPI对此连接进行身份验证，则为NULL。如果主体长度超过`NAMEDATALEN`(标准构建中为64个字符)，则该字段被截断。
encrypted	boolean	如果在此连接上使用了GSSAPI加密，则为真

pg_stat_archiver

`pg_stat_archiver`视图总是有一行，其中包含关于集群的存档进程的数据。

表19.pg_stat_archiver 视图

列	类型	描述
archived_count	bigint	已成功存档的WAL文件数
last_archived_wal	text	最后一个成功存档的WAL文件的名称
last_archived_time	timestamp with time zone	最后一次成功存档操作的时间
failed_count	bigint	记录WAL文件归档失败次数
last_failed_wal	text	最后一次失败的存档操作的WAL文件的名称
last_failed_time	timestamp with time zone	上次存档操作失败的时间
stats_reset	timestamp with time zone	这些统计数据最后一次重置的时间

pg_stat_bgwriter

`pg_stat_bgwriter`视图始终只有一行，其中包含集群的全局数据。

表20.pg_stat_bgwriter 视图

列	类型	描述
checkpoints_timed	bigint	已执行的预定检查点数
checkpoints_req	bigint	请求已执行的检查点数
checkpoint_write_time	double precision	检查点处理中将文件写入磁盘的部分所花费的总时间，以毫秒为单位
checkpoint_sync_time	double precision	检查点处理中将文件同步到磁盘的部分所花费的总时间，以毫秒为单位
buffers_checkpoint	bigint	检查点期间写入的缓冲区数
buffers_clean	bigint	后台写入器写入的缓冲区数
maxwritten_clean	bigint	后台写入器因为写入太多缓冲区而停止清理扫描的次数
buffers_backend	bigint	后端直接写入的缓冲区数
buffers_backend_fsync	bigint	后端必须执行自己的 fsync 调用的次数（通常后台写入器处理这些，即使后端执行自己的写入）
buffers_alloc	bigint	分配的缓冲区数
stats_reset	timestamp with time zone	这些统计数据最后一次重置的时间

pg_stat_database

`pg_stat_database`视图将包含一行用于集群中的每个数据库，加一行用于共享对象，显示数据库范围的统计信息。

表21. pg_stat_database 视图

列	类型	描述
datid	oid	该数据库的 OID，属于共享关系的对象为 0

datname	name	这个数据库的名称，或者共享对象为 NULL
numbackends	integer	当前连接到此数据库的后端数，对于共享对象则为 NULL。这是该视图中唯一返回反映当前状态的值的列；所有其他列返回自上次重置以来累积的值
xact_commit	bigint	此数据库中已提交的事务数
xact_rollback	bigint	该数据库中已回滚的事务数
blks_read	bigint	在该数据库中读取的磁盘块数
blks_hit	bigint	在缓存中发现磁盘块的次数，因此读取不是必需的（这只包括在 IvorySQL 缓存中，而不是在操作系统的文件系统缓存中）
tup_returned	bigint	这个数据库中查询返回的行数
tup_fetched	bigint	这个数据库中查询获取的行数
tup_inserted	bigint	查询在该数据库中插入的行数
tup_updated	bigint	这个数据库中查询更新的行数
tup_deleted	bigint	这个数据库中被查询删除的行数
conflicts	bigint	由于与此数据库中的恢复冲突而取消的查询数（冲突只发生在后备服务器上）
temp_files	bigint	这个数据库中查询创建的临时文件的数量。所有临时文件都将被计数，而不顾及临时文件为什么被创建（例如，排序或散列），也不考虑 log_temp_files 设置
temp_bytes	bigint	这个数据库中的查询写入临时文件的数据总量。所有临时文件都将被计数，而不考虑临时文件为什么被创建，也不考虑 log_temp_files 设置
deadlocks	bigint	在此数据库中检测到的死锁数
checksum_failures	bigint	在此数据库（或共享对象）中检测到的数据页校验码失败数，如果没有启用数据校验码则为 NULL
checksum_last_failure	timestamp with time zone	在此数据库（或共享对象）中检测到最后一个数据页校验码失败的时间，如果没有启用数据校验码则为 NULL
blk_read_time	double precision	在这个数据库中通过后端读取数据文件块所花费的时间，以毫秒为单位（如果启用了 track_io_timing，否则为零）
blk_write_time	double precision	在这个数据库中通过后端写数据文件块所花费的时间，以毫秒为单位（如果启用了 track_io_timing，否则为零）
stats_reset	timestamp with time zone	这些统计数据最后一次重置的时间

pg_stat_database_conflicts

`pg_stat_database_conflicts`视图为每一个数据库包含一行，
用来显示数据库范围内由于与后备服务器上的恢复过程冲突而被取消的查询的统计信息。
这个视图将只包含后备服务器上的信息，因为冲突会不发生在主服务器上。

表22.**pg_stat_database_conflicts** 视图

列	类型	描述
datid	**oid**	数据库的OID
datname	**name**	数据库的名称
confl_tablespace	**bigint**	这个数据库中由于删除表空间而取消的查询的数量
confl_lock	**bigint**	此数据库中由于锁定超时而被取消的查询数
confl_snapshot	**bigint**	此数据库中由于旧快照而取消的查询数
confl_bufferpin	**bigint**	此数据库中由于固定缓冲区而被取消的查询数
confl_deadlock	**bigint**	此数据库中由于死锁而被取消的查询数

pg_stat_all_tables

`pg_stat_all_tables`视图将为当前数据库中的每一个表（包括 TOAST
表）包含一行，该行显示与对该表的访问相关的统计信息。
`pg_stat_user_tables`和`pg_stat_sys_tables`视图包含相同的信息，但是被过滤得分别只显示用户和系统表。

表23.**pg_stat_all_tables** 视图

列	类型	描述
relid	oid	表的OID
schemaname	name	该表所在的模式的名称
relname	name	这个表的名称
seq_scan	bigint	在此表上启动的顺序扫描数
seq_tup_read	bigint	连续扫描获取的实时行数
idx_scan	bigint	对这个表发起的索引扫描数
idx_tup_fetch	bigint	索引扫描获取的实时行数
n_tup_ins	bigint	插入的行数
n_tup_upd	bigint	更新的行数（包括HOT更新的行）
n_tup_del	bigint	删除的行数
n_tup_hot_upd	bigint	HOT更新的行数（即不需要单独的索引更新）
n_live_tup	bigint	活的行的估计数量
n_dead_tup	bigint	僵死行的估计数量
n_mod_since_analyze	bigint	自上次分析此表以来修改的行的估计数量
n_ins_since_vacuum	bigint	自上次清空此表以来插入的行的估计数量

last_vacuum	timestamp with time zone	最后一次手动清理这个表（不包括VACUUM FULL）
last_autovacuum	timestamp with time zone	这个表最后一次被自动清理守护进程清理的时间
last_analyze	timestamp with time zone	上一次手动分析这个表
last_autoanalyze	timestamp with time zone	自动清理守护进程最后一次分析这个表
vacuum_count	bigint	这个表被手动清理的次数（VACUUM FULL不计数）
autovacuum_count	bigint	这个表被autovacuum守护进程清理的次数
analyze_count	bigint	手动分析这个表的次数
autoanalyze_count	bigint	这个表被autovacuum守护进程分析的次数

pg_stat_all_indexes

`pg_stat_all_indexes`视图将为当前数据库中的每个索引包含一行，该行显示关于对该索引访问的统计信息。`pg_stat_user_indexes`和`pg_stat_sys_indexes`视图包含相同的信息，但是被过滤得只分别显示用户和系统索引。

表24.**pg_stat_all_indexes** 视图

列	类型	描述
relid	oid	这个索引所在表的OID
indexrelid	oid	这个索引的OID
schemaname	name	这个索引所在的模式名称
relname	name	这个索引所在表的名称
indexrelname	name	这个索引的名称
idx_scan	bigint	在这个索引上开启的索引扫描的数量
idx_tup_read	bigint	扫描此索引返回的索引项数
idx_tup_fetch	bigint	使用此索引进行简单索引扫描获取的活动表行数

索引可以被简单索引扫描、“位图”索引扫描以及优化器使用。在一次位图扫描中，多个索引的输出可以被通过 AND 或 OR
规则组合，因此当使用一次位图扫描时难以将取得的个体堆行与特定的索引关联起来。因此，一次位图扫描会增加它使用的索引的`pg_stat_all_indexes`.**idx_tup_read`计数，并且为每个表增加`pg_stat_all_tables`.idx_tup_fetch`计数，但是它不影响`pg_stat_all_indexes`.idx_tup_fetch**。如果所提供的常量值不在优化器统计信息记录的范围之内，优化器也会访问索引来检查，因为优化器统计信息可能已经“不新鲜”了。

> **注意**
>
> 即使不用位图扫描，`idx_tup_read`和`idx_tup_fetch`计数也可能不同，因为`idx_tup_read`统计从该索引取得的索引项而`idx_tup_fetch`统计从表取得的活着的行。如果使用该索引取得了任何死亡行或还未提交的行，或者如果通过一次只用索引扫描的方式避免了任何堆获取，后者将较小。

pg_statio_all_tables

`pg_statio_all_tables`视图将为当前数据库中的每个表（包括 TOAST
表）包含一行，该行显示指定表上有关 I/O

的统计信息。`pg_statio_user_tables`和`pg_statio_sys_tables`视图包含相同的信息，但是被过滤得分别只显示用户表和系统表。

表25.pg_statio_all_tables 视图

列	类型	描述
relid	oid	表的OID
schemaname	name	该表所在的模式名
relname	name	这个表的名称
heap_blks_read	bigint	从该表中读取的磁盘块的数量
heap_blks_hit	bigint	该表中的缓冲区命中数
idx_blks_read	bigint	从这个表上所有索引读取的磁盘块数
idx_blks_hit	bigint	这个表上所有索引中的缓冲区命中数
toast_blks_read	bigint	从这个表的TOAST表中读取的磁盘块的数量（如果有的话）
toast_blks_hit	bigint	这个表的TOAST表中的缓冲区命中数（如果有的话）
tidx_blks_read	bigint	从这个表的TOAST表索引中读取的磁盘块的数量（如果有的话）
tidx_blks_hit	bigint	这个表的TOAST表索引中的缓冲区命中数（如果有的话）

pg_statio_all_indexes

`pg_statio_all_indexes`视图将为当前数据库中的每个索引包含一行，该行显示指定索引上有关 I/O 的统计信息。
`pg_statio_user_indexes`和`pg_statio_sys_indexes`视图包含相同的信息，但是被过滤得分别只显示用户索引和系统索引。

表26.pg_statio_all_indexes 视图

列	类型	描述
relid	oid	这个索引所在表的OID
indexrelid	oid	这个索引的OID
schemaname	name	索引所在的模式名称
relname	name	这个索引所在表的名称
indexrelname	name	这个索引的名称
idx_blks_read	bigint	从这个索引中读取的磁盘块数量
idx_blks_hit	bigint	这个索引中的缓冲区命中数

pg_statio_all_sequences

`pg_statio_all_sequences`视图将为当前数据库中的每个序列包含一行，该行显示在指定序列上有关 I/O 的统计信息。

表27.pg_statio_all_sequences 视图

列	类型	描述
relid	oid	序列的OID

schemaname	name	此序列所在的模式的名称
relname	name	此序列的名称
blks_read	bigint	从这个序列中读取的磁盘块的数量
blks_hit	bigint	在此序列中的缓冲区命中数

pg_stat_user_functions

`pg_stat_user_functions`视图将为每一个被追踪的函数包含一行，该行显示有关该函数执行的统计信息。track_functions参数控制到底哪些函数被跟踪。

表28.pg_stat_user_functions 视图

列	类型	描述
funcid	oid	函数的OID
schemaname	name	这个函数所在的模式的名称
funcname	name	这个函数的名称
calls	bigint	这个函数已经被调用的次数
total_time	double precision	在这个函数以及它所调用的其他函数中花费的总时间，以毫秒计
self_time	double precision	在这个函数本身花费的总时间，不包括被它调用的其他函数，以毫秒计

pg_stat_slru

IvorySQL通过*SLRU*(simple least-recently-used，简单的最近-最少-使用)缓存访问某些磁盘上的信息。`pg_stat_slru`视图将为每个被跟踪的SLRU缓存包含一行，显示关于访问缓存页面的统计信息。

表29.pg_stat_slru 视图

列	类型	描述
name	text	SLRU的名称
blks_zeroed	bigint	初始化期间被置零的块数
blks_hit	bigint	已经在SLRU中的磁盘块被发现的次数，因此不需要读取（这只包括SLRU中的命中，而不是操作系统的文件系统缓存）
blks_read	bigint	为这个SLRU读取的磁盘块数
blks_written	bigint	为这个SLRU写入的磁盘块数
blks_exists	bigint	为这个SLRU检查是否存在的块数
flushes	bigint	此SLRU的脏数据刷新数
truncates	bigint	这个SLRU的截断数
stats_reset	timestamp with time zone	这些统计数据最后一次重置的时间

Statistics Functions

其他查看统计信息的方法是直接使用查询，这些查询使用上述标准视图用到的底层统计信息访问函数。
如要了解如函数名等细节，可参考标准视图的定义（例如，在psql中你可以发出`\d+ pg_stat_activity`）。
针对每一个数据库统计信息的访问函数把一个数据库 OID
作为参数来标识要报告哪个数据库。而针对每个表和每个索引的函数要求表或索引 OID。
针对每个函数统计信息的函数用一个函数
OID。注意只有在当前数据库中的表、索引和函数才能被这些函数看到。

更多统计集合的函数列在 表 30 中。

表30.Additional Statistics Functions

函数	描述
pg_backend_pid () → integer	返回附加到当前会话的服务器进程的进程ID。
pg_stat_get_activity (integer) → setof record	使用指定的进程ID返回有关后端信息的记录，如果指定了NULL，则返回系统中每个活动后端的一条记录。返回的字段是pg_stat_activity视图中字段的子集。
pg_stat_get_snapshot_timestamp () → timestamp with time zone	返回当前统计快照的时间戳。
pg_stat_clear_snapshot () → void	丢弃当前的统计快照。
pg_stat_reset () → void	将当前数据库的所有统计计数器重置为零。默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
pg_stat_reset_shared (text) → void	根据参数的不同，将一些集群范围的统计计数器重置为零。参数可以是bgwriter来重置pg_stat_bgwriter视图中显示的所有计数器，或者archiver来重置pg_stat_archiver视图中显示的所有计数器。默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
pg_stat_reset_single_table_counters (oid) → void	将当前数据库中单个表或索引的统计信息重置为零。默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
pg_stat_reset_single_function_counters (oid) → void	将当前数据库中单个函数的统计信息重置为零。默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。
pg_stat_reset_slru (text) → void	将单个SLRU缓存或集群中所有SLRU的统计信息重置为零。如果该参数为NULL，则所有SLRU缓存的pg_stat_slru视图中显示的计数器将被重置。参数可以是CommitTs、MultiXactMember、MultiXactOffset、Notify、Serial、Subtrans或Xact中的一个，以便只重置该条目的计数器。如果参数是other（或实际上，任何无法识别的名称），那么所有其他SLRU缓存的计数器，如扩展定义的缓存，将被重置。默认情况下该函数仅限于超级用户，但是其他用户可以被授予EXECUTE来运行此函数。

`pg_stat_get_activity`是`pg_stat_activity`视图的底层函数，
它返回一个行集合，其中包含有关每个后端进程所有可用的信息。有时只获得该信息的一个子集可能会更方便。 在那些情况中，可以使用一组更老的针对每个后端的统计访问函数，这些显示在 表 31中。
这些访问函数使用一个后端 ID 号，范围从 1 到当前活动后端数目。
函数`pg_stat_get_backend_idset`提供了一种方便的方法为每个活动后端产生一行来调用这些函数。
例如，要显示PID以及所有后端当前的查询：

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

表31.Per-Backend Statistics Functions

函数	描述
pg_stat_get_backend_idset () → setof integer	返回当前活动后端ID号的集合(从1到活动后端数)。

pg_stat_get_backend_activity (integer) → text	返回此后端最近查询的文本。
pg_stat_get_backend_activity_start (integer) → timestamp with time zone	返回后端最近一次查询开始的时间。
pg_stat_get_backend_client_addr (integer) → inet	返回连接到此后端的客户端的IP地址。
pg_stat_get_backend_client_port (integer) → integer	返回客户端用于通信的TCP端口号。
pg_stat_get_backend_dbid (integer) → oid	返回此后端连接的数据库的OID。
pg_stat_get_backend_pid (integer) → integer	返回此后端进程ID。
pg_stat_get_backend_start (integer) → timestamp with time zone	返回该进程开始的时间。
pg_stat_get_backend_userid (integer) → oid	返回登录到此后端的用户的OID。
pg_stat_get_backend_wait_event_type (integer) → text	如果后端当前正在等待，返回等待事件类型名称，否则返回NULL。
pg_stat_get_backend_wait_event (integer) → text	如果后端当前正在等待，则返回等待事件名称，否则为NULL。
pg_stat_get_backend_xact_start (integer) → timestamp with time zone	返回后端当前事务开始的时间。

查看锁

监控数据库活动的另外一个有用的工具是`pg_locks`系统表。这样就允许数据库管理员查看在锁管理器里面未解决的锁的信息。例如，这个功能可以被用于：

- 查看当前所有未解决的锁、在一个特定数据库中的关系上所有的锁、在一个特定关系上所有的锁，或者由一个特定IvorySQL会话持有的所有的锁。
- 判断当前数据库中带有最多未授予锁的关系（它很可能是数据库客户端的竞争源）。
- 判断锁竞争给数据库总体性能带来的影响，以及锁竞争随着整个数据库流量的变化范围。

Progress Reporting

IvorySQL具有在命令执行过程中报告某些命令进度的能力。
目前，支持进度报告的命令只有`ANALYZE`,`CLUSTER`,`CREATE INDEX`, `VACUUM`, 和 BASE_BACKUP例如pg_basebackup发出的进行基础备份的复制命令。未来可能还会扩展。

ANALYZE Progress Reporting

每当`ANALYZE`运行时，`pg_stat_progress_analyze`视图将包含当前运行该命令的每个后端的一行。
下面的表描述了将要报告的信息，并提供了关于如何解释它们的信息。

表32.`pg_stat_progress_analyze` 视图

列	类型	描述
pid	integer	后端的进程ID。
datid	oid	后端连接到的数据库的OID。
datname	name	后端连接到的数据库的名称。
relid	oid	被分析的表的OID。
phase	text	当前处理阶段。参见 表 33。
sample_blks_total	bigint	将被采样的堆块的总数。
sample_blks_scanned	bigint	扫描的堆块数量。
ext_stats_total	bigint	扩展统计信息的数量。

ext_stats_computed	bigint	已经计算的扩展统计的数量。此计数器仅在 computing extended statistics 阶段增进。
child_tables_total	bigint	子表的数量。
child_tables_done	bigint	扫描的子表数。此计数器只有在 acquiring inherited sample rows 阶段才会增进。
current_child_table_relid	oid	当前正在扫描的子表的 OID。此字段仅在 acquiring inherited sample rows 时有效。

表33.ANALYZE phases

阶段	描述
`initializing`	命令正在准备开始扫描堆。这个阶段预计会非常短暂。
`acquiring sample rows`	该命令当前正在扫描`relid`给出的表以获得示例行。
`acquiring inherited sample rows`	该命令当前正在扫描子表以获得示例行。列`child_tables_total`,`child_tables_done`,和`current_child_table_relid`包含此阶段的进度信息。
`computing statistics`	该命令从表扫描期间获得的样例行计算统计信息。
`computing extended statistics`	该命令从表扫描期间获得的样例行计算扩展统计信息。
`finalizing analyze`	该命令在更新`pg_class`。当此阶段完成时，`ANALYZE`将结束。

> **注意**
>
> 当在分区表上运行`ANALYZE`时，它的所有分区也会被递归分析，如在ANALYZE中曾提到过。在这种情况下，首先报告父表的`ANALYZE`进度，收集它的继承统计信息，然后是每个分区的(继承统计信息)。

CREATE INDEX Progress Reporting

每当运行`CREATE INDEX`或`REINDEX`时，`pg_stat_progress_create_index`视图将包含当前正在创建索引的每个后端的一行。 下面的表描述了将要报告的信息，并提供了关于如何解释它的信息。

表34.`pg_stat_progress_create_index` 视图

列	类型	描述
`pid`	`integer`	后端的进程ID。
`datid`	`oid`	后端连接到的数据库的OID。
`datname`	`name`	后端连接到的数据库的名称。
`relid`	`oid`	正在创建索引的表的OID。
`index_relid`	`oid`	正在创建或重建索引的OID。在非并发`CREATE INDEX`的时候，此为0。

command	text	在运行的命令：CREATE INDEX 、CREATE INDEX CONCURRENTLY 、REINDEX 或 REINDEX CONCURRENTLY。
phase	text	索引创建的当前处理阶段。参见 [表 35](http://www.postgresql.org/docs/17/progress-reporting.html#CREATE-INDEX-PHASES)。
lockers_total	bigint	在适用的情况下，需要等待的储物柜总数。
lockers_done	bigint	已经等待的储物柜数量。
current_locker_pid	bigint	目前正在等待的储物柜的进程ID。
blocks_total	bigint	本阶段要处理的区块总数。
blocks_done	bigint	当前阶段已经处理的区块数量。
tuples_total	bigint	当前阶段要处理的元组总数。
tuples_done	bigint	在当前阶段已经处理的元组数量。
partitions_total	bigint	在分区表上创建索引时，该列被设置为要在其上创建索引的分区总数。
partitions_done	bigint	当在分区表上创建索引时，该列被设置为在其上完成索引的分区数。

表35.CREATE INDEX 的阶段

阶段	描述
初始化	`CREATE INDEX`或`REINDEX`正在准备创建索引。这个阶段预计会非常短暂。
构建前等待读写器	CREATE INDEX CONCURRENTLY`或`REINDEX CONCURRENTLY`正在等待有可能看到表的写锁的事务完成。 当不在并发模式时，这个阶段会被跳过。`lockers_total、 lockers_done 和 current_locker_pid 列包含了这个阶段的进度信息。
新建索引	索引是由访问方法专用代码建立的。 在这一阶段，支持进度报告的访问方法填写自己的进度数据，子阶段在这一栏中表示。 通常情况下，blocks_total`和`blocks_done`将包含进度数据，也可能包含`tuples_total`和`tuples_done。
在验证前等待读写器	CREATE INDEX CONCURRENTLY`或`REINDEX CONCURRENTLY`正在等待有可能写入表的事务完成写锁的事务。当不在并发模式时，这个阶段会被跳过。`lockers_total、 lockers_done 和 current_locker_pid 列包含了这个阶段的进度信息。
索引验证：扫描索引	CREATE INDEX CONCURRENTLY`正在扫描索引，搜索需要验证的图元组。如果不是在并发模式下，这个阶段会被跳过。列`blocks_total（设置为索引的总大小）和`blocks_done`包含了这个阶段的进度信息。
指数验证：排序元组	`CREATE INDEX CONCURRENTLY`正在对索引扫描阶段的输出进行排序。

索引验证：扫描表	`CREATE INDEX CONCURRENTLY`正在扫描表，以验证前两个阶段收集的索引图元。当不在并发模式时，这个阶段被跳过。`blocks_total`列（设置为表的总大小）和`blocks_done`列包含这个阶段的进度信息。
等待旧照	`CREATE INDEX CONCURRENTLY`或`REINDEX CONCURRENTLY`正在等待可能看到表的事务释放快照。 当不处于并发模式时，这个阶段会被跳过。`lockers_total`、`lockers_done`和`current_locker_pid`列包含了这个阶段的进度信息。
标记 dead之前等待readers	`REINDEX CONCURRENTLY`等待表上有读锁的事务完成后，再将旧索引标记为死索引。当不在并发模式时，这个阶段被跳过。`lockers_total`、`lockers_done`和`current_locker_pid`列包含了这个阶段的进度信息。
在 dropping之前等待readers	`REINDEX CONCURRENTLY`等待表上有读锁的事务完成后，再丢弃旧索引。当不在并发模式时，这个阶段被跳过。列`lockers_total`、`lockers_done`和`current_locker_pid`包含了这个阶段的进度信息。

VACUUM进度报告

只要`VACUUM`正在运行，每一个当前正在清理的后端（包括autovacuum工作者进程）在`pg_stat_progress_vacuum`视图中都会有一行。下面的表描述了将被报告的信息并且提供了如何解释它们的信息。`VACUUM FULL`命令的进度是通过`pg_stat_progress_cluster`报告的，因为`VACUUM FULL`和`CLUSTER`都是重写表，而普通的`VACUUM`只是原地修改表。

表36.`pg_stat_progress_vacuum`视图

列	类型	描述
pid	integer	后端的进程ID。
datid	oid	这个后端连接的数据库的OID。
datname	name	这个后端连接的数据库的名称。
relid	oid	被vacuum的表的OID。
phase	text	vacuum的当前处理阶段。
heap_blks_total	bigint	该表中堆块的总数。这个数字在扫描开始时报告，之后增加的块将不会（并且不需要）被这个VACUUM访问。
heap_blks_scanned	bigint	被扫描的堆块数量。由于visibility map被用来优化扫描，一些块将被跳过而不做检查，被跳过的块会被包括在这个总数中，因此当清理完成时这个数字最终将会等于heap_blks_total。仅当处于扫描堆阶段时这个计数器才会前进。
heap_blks_vacuumed	bigint	被清理的堆块数量。除非表没有索引，这个计数器仅在处于清理堆阶段时才会前进。不包含死亡元组的块会被跳过，因此这个计数器可能有时会向前跳跃一个比较大的增量。
index_vacuum_count	bigint	已完成的索引清理周期数。

max_dead_tuples	bigint	在需要执行一个索引清理周期之前我们可以存储的死亡元组数，取决于maintenance_work_mem。
num_dead_tuples	bigint	从上一个索引清理周期以来收集的死亡元组数。

表37.VACUUM的阶段

阶段	描述
初始化	`VACUUM`正在准备开始扫描堆。这个阶段应该很简短。
扫描堆	`VACUUM`正在扫描堆。如果需要，它将会对每个页面进行修建以及碎片整理，并且可能会执行冻结动作。`heap_blks_scanned`列可以用来监控扫描的进度。
清理索引	`VACUUM`当前正在清理索引。如果一个表拥有索引，那么每次清理时这个阶段会在堆扫描完成后至少发生一次。如果maintenance_work_mem不足以存放找到的死亡元组，则每次清理时会多次清理索引。
清理堆	**`VACUUM`当前正在清理堆。清理堆与扫描堆不是同一个概念，清理堆发生在每一次清理索引的实例之后。如果`heap_blks_scanned`小于`heap_blks_total`，**系统将在这个阶段完成之后回去扫描堆；否则，系统将在这个阶段完成后开始清理索引。
清除索引	`VACUUM`当前正在清除索引。这个阶段发生在堆被完全扫描并且对堆和索引的所有清理都已经完成以后。
截断堆	`VACUUM`正在截断堆，以便把关系尾部的空页面返还给操作系统。这个阶段发生在清除完索引之后。
执行最后的清除	`VACUUM`在执行最终的清除。在这个阶段中，`VACUUM`将清理空闲空间映射、更新`pg_class`中的统计信息并且将统计信息报告给统计收集器。当这个阶段完成时，`VACUUM`也就结束了。

CLUSTER进度报告

每当`CLUSTER`或`VACUUM FULL`运行时，`pg_stat_progress_cluster`视图将包含当前正在运行的每一个后台的记录。下面的表格描述了将被报告的信息，并提供了关于如何解释这些信息的信息。

表38.`pg_stat_progress_cluster` 视图

列	类型	描述
`pid`	`integer`	后台的进程ID。
`datid`	`oid`	该后端连接的数据库的OID。
`datname`	`name`	与此后端连接的数据库的名称。
`relid`	`oid`	被集群的表的OID。
`command`	`text`	正在运行的命令。`CLUSTER`或`VACUUM FULL`。
`phase`	`text`	当前处理阶段。
`cluster_index_relid`	`oid`	如果正在使用索引对表进行扫描，这就是正在使用的索引的OID；否则为0。

heap_tuples_scanned	bigint	扫描的堆元组数。这个计数器只有在阶段为`seq scanning heap`、`index scanning heap`或`writing new heap`时才会增进。
heap_tuples_written	bigint	写入的堆元组的数量。这个计数器只有在阶段为`seq scanning heap`、`index scanning heap`或`writing new heap`时才会前进。
heap_blks_total	bigint	表中的堆块总数。这个数字是在`seq scanning heap`的开始时报告的。
heap_blks_scanned	bigint	扫描的堆块数量。这个计数器只有在阶段为`seq scanning heap`时才会增进。
index_rebuild_count	bigint	重建的索引数。该计数器仅在`重建索引`阶段时才会增进。

表39.CLUSTER 和 VACUUM FULL 阶段

阶段	描述
初始化	该命令准备开始扫描堆栈。 这个阶段预计会非常短暂。
seq扫描堆	该命令目前采用顺序扫描的方式对表进行扫描。
索引扫描堆	`CLUSTER`目前正在使用索引扫描表。
元组排序	`CLUSTER`目前正在对元组进行排序。
新写入堆	`CLUSTER`目前正在编写新的堆。
交换关系文件	目前，该命令正在将新建立的文件调换到位。
重建索引	该命令目前正在重建一个索引。
清理	该命令正在执行最后的清理工作。 当此阶段完成后，`CLUSTER`或`VACUUM FULL`将结束。

基础备份进度报告

每当像pg_basebackup这样的应用程序进行基本备份时，`pg_stat_progress_basebackup`视图将包含当前运行`BASE_BACKUP`复制命令和流备份的每个WAL发送进程的一行。 下面的表描述了将要报告的信息，并提供了关于如何解释它的信息。

表40.pg_stat_progress_basebackup 视图

列	类型	描述
pid	integer	WAL发送方进程ID。
phase	text	目前的处理阶段。

backup_total	bigint	将被流输送的数据总量。这是在streaming database files阶段开始时的估计和报告。注意，这只是一个近似值，因为在streaming database files阶段，数据库可能会改变，而WAL日志可能会在稍后的备份中包含。一旦流数据量超过了估计的总大小，该值始终与backup_streamed相同。如果在pg_basebackup中禁用估算（也就是说，指定了—no-estimate-size选项），这为NULL。
backup_streamed	bigint	数据流的总量。这个计数器只在streaming database files阶段或transferring wal files时增进。
tablespaces_total	bigint	要流输送的表空间总数。
tablespaces_streamed	bigint	流输送的表空间数。此计数器仅在streaming database files阶段增进。

表41.基础备份阶段

阶段	描述
`initializing`	WAL发送器进程正在准备开始备份。这个阶段预计会非常短暂。
`waiting for checkpoint to finish`	WAL发送器进程目前正在执行`pg_start_backup`以准备进行基础备份，并等待启动备份检查点完成。
`estimating backup size`	WAL发送程序目前正在估计将作为基础备份流传输的数据库文件的总量。
`streaming database files`	WAL发送器当前正在流数据库文件作为基础备份。
`waiting for wal archiving to finish`	WAL发送方进程目前正在执行`pg_stop_backup`以完成备份，并等待基础备份所需的所有WAL文件成功存档。 如果在pg_basebackup中指定了`--wal-method=none`或`--wal-method=stream`，则备份将在此阶段完成后结束。
`transferring wal files`	WAL发送器进程正在传输备份过程中产生的所有WAL日志。 如果pg_basebackup中指定了`--wal-method=fetch`， 则该阶段发生在`waiting for wal archiving to finish`阶段之后。当此阶段完成时备份将结束。

动态追踪

IvorySQL提供了功能来支持数据库服务器的动态追踪。这样就允许在代码中的特定点上调用外部工具来追踪执行过程。

一些探针或追踪点已经被插入在源代码中。这些探针的目的是被数据库开发者和管理员使用。默认情况下，探针不被编译到IvorySQL中；用户需要显式地告诉配置脚本使得探针可用。

目前，DTrace已被支持，它在 Solaris、macOS、FreeBSD、NetBSD 和 Oracle Linux 上可用。 Linux 的SystemTap项目提供了一种可用的 DTrace 等价物。支持其他动态追踪工具在理论上可以通过改变`src/include/utils/probes.h`中的宏定义实现。

动态追踪的编译

默认情况下，探针是不可用的，因此你将需要显式地告诉配置脚本让探针在IvorySQL中可用。要包括 DTrace 支持，在配置时指定`--enable-dtrace`。

内建探针

如表 42所示，源代码中提供了一些标准探针。表
43显示了在探针中使用的类型。当然，可以增加更多探针来增强IvorySQL的可观测性。

表42.内建 DTrace 探针

名称	参数	描述
transaction-start	(LocalTransactionId)	在一个新事务开始时触发的探针。arg0 是事务 ID。
transaction-commit	(LocalTransactionId)	在一个事务成功完成时触发的探针。arg0 是事务 ID。
transaction-abort	(LocalTransactionId)	当一个事务失败完成时触发的探针。arg0 是事务 ID。
query-start	(const char *)	当一个查询的处理被开始时触发的探针。arg0 是查询字符串。
query-done	(const char *)	当一个查询的处理完成时触发的探针。arg0 是查询字符串。
query-parse-start	(const char *)	当一个查询的解析被开始时触发的探针。arg0 是查询字符串。
query-parse-done	(const char *)	当一个查询的解析完成时触发的探针。arg0 是查询字符串。
query-rewrite-start	(const char *)	当一个查询的重写被开始时触发的探针。arg0 是查询字符串。
query-rewrite-done	(const char *)	当一个查询的重写完成时触发的探针。arg0 是查询字符串。
query-plan-start	()	当一个查询的规划被开始时触发的探针。
query-plan-done	()	当一个查询的规划完成时触发的探针。
query-execute-start	()	当一个查询的执行被开始时触发的探针。
query-execute-done	()	当一个查询的执行完成时触发的探针。
statement-status	(const char *)	任何时候当服务器进程更新它的`pg_stat_activity`.`status`时触发的探针。arg0 是新的状态字符串。
checkpoint-start	(int)	当一个检查点被开始时触发的探针。arg0 传递位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。

checkpoint-done	(int, int, int, int, int)	当一个检查点完成时触发的探针（检查点处理过程中序列中列出的下一个触发的探针）。arg0 是要写的缓冲区数量。arg1 是缓冲区的总数。arg2、arg3 和 arg4 分别包含了增加、删除和循环回收的 WAL 文件的数量。
clog-checkpoint-start	(bool)	当一个检查点的 CLOG 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
clog-checkpoint-done	(bool)	当一个检查点的 CLOG 部分完成时触发的探针。arg0 的含义与`clog-checkpoint-start`中相同。
subtrans-checkpoint-start	(bool)	当一个检查点的 SUBTRANS 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
subtrans-checkpoint-done	(bool)	当一个检查点的 SUBTRANS 部分完成时触发的探针。arg0 的含义与`subtrans-checkpoint-start`中相同。
multixact-checkpoint-start	(bool)	当一个检查点的 MultiXact 部分被开始时触发的探针。arg0 为真表示正常检查点，为假表示关闭检查点。
multixact-checkpoint-done	(bool)	当一个检查点的 MultiXact 部分完成时触发的探针。arg0 的含义与`multixact-checkpoint-start`中相同。
buffer-checkpoint-start	(int)	当一个检查点的写缓冲区部分被开始时触发的探针。arg0 传递位标志来区分不同的检查点类型，例如关闭（shutdown）、立即（immediate）或强制（force）。
buffer-sync-start	(int, int)	当我们在检查点期间开始写脏缓冲区时（在标识哪些缓冲区必须被写之后）触发的探针。arg0 是缓冲区总数，arg1 是当前为脏并且需要被写的缓冲区数量。
buffer-sync-written	(int)	在检查点期间当每个缓冲区被写完之后触发的探针。arg0 是缓冲区的 ID。
buffer-sync-done	(int, int, int)	当所有脏缓冲区被写之后触发的探针。arg0 是缓冲区总数。arg1 是检查点进程实际写的缓冲区数量。arg2 是期望写的数目（`buffer-sync-start`的 arg1）；arg1 和 arg2 的任何的不同反映在该检查点期间有其他进程刷写了缓冲区。
buffer-checkpoint-sync-start	()	在脏缓冲区被写入到内核之后并且在开始发出 fsync 请求之前触发的探针。

buffer-checkpoint-done	()	当同步缓冲区到磁盘完成时触发的探针。
twophase-checkpoint-start	()	当一个检查点的两阶段部分被开始时触发的探针。
twophase-checkpoint-done	()	当一个检查点的两阶段部分完成时触发的探针。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	当一次缓冲区读被开始时触发的探针。arg0 和 arg1 包含该页的分叉号和块号（如果这是一次关系扩展请求，arg1 为 -1）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。arg6 为真表示一次关系扩展请求，为假表示正常读。
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	当一次缓冲区读完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号（如果这是一次关系扩展请求，arg1 现在包含新增加块的块号）。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是 InvalidBackendId（-1）。arg6 为真表示一次关系扩展请求，为假表示正常读。arg7 为真表示在池中找到该缓冲区，为假表示没有找到。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	在发出对一个共享缓冲区的任意写请求之前触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个写请求完成时触发的探针（注意这只反映传递数据给内核的时间，它通常并没有实际地被写入到磁盘）。参数和`buffer-flush-start`的相同。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一个服务器进程开始写一个脏缓冲区时触发的探针（如果这经常发生，表示shared_buffers太小，或需要调整后台写入器的控制参数）。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	当一次脏缓冲区写完成时触发的探针。参数与`buffer-write-dirty-start`相同。

wal-buffer-write-dirty-start	()	当一个服务器进程因为没有可用 WAL 缓冲区空间开始写一个脏 WAL 缓冲区时触发的探针（如果这经常发生，表示wal_buffers太小）。
wal-buffer-write-dirty-done	()	当一次脏 WAL 缓冲区完成时触发的探针。
wal-insert	(unsigned char, unsigned char)	当一个 WAL 记录被插入时触发的探针。arg0 是该记录的资源管理者（rmid）。arg1 包含 info 标志。
wal-switch	()	当请求一次 WAL 段切换时触发的探针。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始从一个关系读取一块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是`InvalidBackendId`（-1）。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一次块读取完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是`InvalidBackendId`（-1）。arg6 是实际读取的字节数，而 arg7 是请求读取的字节数（如果两者不同就意味着麻烦）。
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	当开始向一个关系中写入一个块时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3 和 arg4 包含表空间、数据库和关系 OID 用以识别该关系。对一个本地缓冲区，arg5 是创建临时关系的后端的 ID；对于一个共享缓冲区，arg5 是`InvalidBackendId`（-1）。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	当一个块写操作完成时触发的探针。arg0 和 arg1 包含该页的分叉号和块号。arg2、arg3和arg4 包含表空间、数据库和关系 OID来标识该关系。对于一个本地缓冲区，arg5 是创建临时关系的后端 ID；对于一个共享缓冲区，arg5 是`InvalidBackendId`（-1）。arg6 是实际写的字节数，而 arg7 是要求写的字节数（如果这两者不同，则意味着麻烦）。

sort-start	(int, bool, int, int, bool, int)	当一次排序操作开始时触发的探针。arg0 指示是堆排序、索引排序或数据排序。arg1 为真表示唯一值强制。arg2 是键列的数目。arg3 是允许使用的工作内存数（以千字节计）。如果要求随机访问排序结果，那么 arg4 为真。arg5为`0`时表示串行，为`1`时表示并行工作者，为`2`时表示并行领袖。
sort-done	(bool, long)	当一次排序完成时触发的探针。arg0 为真表示外排序，为假表示内排序。arg1 是用于一次外排序的磁盘块的数目，或用于一次内排序的以千字节计的内存。
lwlock-acquire	(char *, LWLockMode)	当成功获得一个 LWLock 时触发的探针。 arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-release	(char *)	当一个 LWLock 被释放时（但是注意还没有唤醒任何一个被释放的等待者）触发的探针。 arg0 是该 LWLock 所在的切片（Tranche）。
lwlock-wait-start	(char *, LWLockMode)	当一个 LWLock不是当即可用并且一个服务器进程因此开始等待该锁变为可用时触发的探针。 arg0 是该 LWLock 所在的切片（Tranche）。 arg1 请求的锁模式，是排他或共享。
lwlock-wait-done	(char *, LWLockMode)	当一个进程从对一个 LWLock 的等待中被释放时（它实际还没有得到该锁）时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-condacquire	(char *, LWLockMode)	当调用者指定无需等待而成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lwlock-condacquire-fail	(char *, LWLockMode)	当调用者指定无需等待而没有成功获得一个 LWLock 时触发的探针。arg0 是该 LWLock 所在的切片（Tranche）。 arg1 所请求的锁模式，是排他或共享。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr锁）的请求由于锁不可用开始等待时触发的探针。arg0 到 arg3 是标识被锁定对象的标签域。arg4 指示被锁对象的类型。arg5 表示被请求的锁类型。

lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	当一个重量级锁（lmgr 锁）的请求结束等待时（即已经得到锁）触发的探针。参数与`lock-wait-start`一样。
deadlock-found	()	当死锁检测器发现死锁时触发的探针。

表43.定义用在探针参数中的类型

类型	定义
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	unsigned char

使用探针

下面的例子展示了一个分析系统中事务计数的 DTrace 脚本，可以用来代替一次性能测试之前和之后的`pg_stat_database`快照：

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

当被执行时，该例子 D 脚本给出这样的输出：

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                        2312105013
```

> 注意
>
> SystemTap 为追踪脚本使用一个不同于 DTrace 的标记，但是底层的探针是兼容的。值得注意的是，在这样写的时候，SystemTap 脚本必须使用双下划线代替连字符来引用探针名。在未来的 SystemTap 发行中这很可能会被修复。你应该记住，DTrace 脚本需要细心地编写和调试，否则被收集的追踪信息可能会毫无意义。在大部分发现问题的情况中，它就是发生问题的部件，而不是底层系统。当讨论使用动态追踪发现的信息时，一定要附上使用的脚本以便其也被检查和讨论。

定义新探针

开发者可以在代码中任意位置定义新的探针，当然这要重新编译之后才能生效。下面是插入新探针的步骤：

1. 决定探针名称以及探针可用的数据
2. 把该探针定义加入到`src/backend/utils/probes.d`
3. 如果`pg_trace.h`还不存在于包含该探针点的模块中，包括它，并且在源代码中期望的位置插入`TRACE_POSTGRESQL`探针宏
4. 重新编译并验证新探针是可用的

例子：. 这里是一个如何增加一个探针来用事务 ID 追踪所有新事务的例子。

1. 决定探针将被命名为`transaction-start`并且需要一个`LocalTransactionId`类型的参数
2. 将该探针定义加入到`src/backend/utils/probes.d`：

   ```
   ```

   probe transaction__start(LocalTransactionId);
   ```
   ```

   > 注意探针名字中双下划线的使用。在一个使用探针的 DTrace 脚本中，双下划线需要被替换为一个连字符，因此 ，对用户而言`transaction-start`是文档名。

3. 在编译时，transaction__start`被转换成一个宏调用`TRACE_POSTGRESQL_TRANSACTION_START（注意这里是单下划线），可以通过包括头文件`pg_trace.h`获得。将宏调用加入到源代码中的合适位置。在这种情况下，看起来类似：

   ```
   ```

   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```
   ```

4. 在重新编译和运行新的二进制文件之后，通过运行下面的 DTrace 命令来检查新增的探针是否可用。你应该看到类似下面的输出：

```
# dtrace -ln transaction-start
   ID    PROVIDER          MODULE           FUNCTION NAME
18705 postgresql49878     postgres     StartTransactionCommand transaction-start
18755 postgresql49877     postgres     StartTransactionCommand transaction-start
18805 postgresql49876     postgres     StartTransactionCommand transaction-start
18855 postgresql49875     postgres     StartTransactionCommand transaction-start
18986 postgresql49873     postgres     StartTransactionCommand transaction-start
```

向C代码中添加追踪宏时，有一些事情需要注意：

- 需要小心的是，为探针参数指定的数据类型要匹配宏中使用的变量的数据类型，否则会发生编译错误。
- 在大多数平台上，如果用`--enable-dtrace`编译了IvorySQL，无论何时当控制经过一个追踪宏时，都会评估该宏的参数，即使没有进行追踪也会这样做。通常不需要担心你是否只在报告一些局部变量的值。但是要注意不要将开销大的函数调用放入参数中。如果你需要这样做，考虑通过检查追踪是否真的被启用来保护该宏：

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

每个追踪宏有一个对应的`ENABLED`宏。

监控磁盘的使用

判断磁盘用量

每个表都有一个主要的堆磁盘文件，大多数数据都存储在其中。如果一个表有着可能会很宽（尺寸大）的列，则另外还有一个TOAST文件与这个表相关联，它用于存储因为太宽而不能存储在主表里面的值。如果有这个附属文件，那么TOAST表上会有一个可用的索引。当然，同时还可能有索引和基表关联。每个表和索引都存放在单独的磁盘文件里 — 如果文件超过 1G 字节，甚至可能多于一个文件。

你可以以三种方式监视磁盘空间：使用oid2name模块或者人工观察系统目录。SQL函数是最容易使用的方法，同时也是我们通常推荐的方法。本节剩余的部分将展示如何通过观察系统目录来监视磁盘空间。

在一个最近清理过或者分析过的数据库上使用psql，你可以发出查询来查看任意表的磁盘用量：

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';


 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

每个页通常都是 8K 字节（记住，relpages`只会由`VACUUM、`ANALYZE`和少数几个 DDL 命令如`CREATE INDEX`所更新）。如果你想直接检查表的磁盘文件，那么文件路径名应该有用。

要显示TOAST表使用的空间，我们可以使用一个类似下面这样的查询：

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

你也可以很容易地显示索引的尺寸：

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

      relname      | relpages
-------------------+----------
 customer_id_index |       26
```

我们很容易用下面的信息找出最大的表和索引：

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
```

```
customer              |      3144
```

磁盘满失败

一个数据库管理员最重要的磁盘监控任务就是确保磁盘不会写满。一个写满了的数据磁盘可能不会导致数据的崩溃，但它肯定会让系统变得不可用。如果保存 WAL
文件的磁盘变满，会发生数据库服务器致命错误并且可能发生关闭。

如果你不能通过删除一些其他的东西来释放一些磁盘空间，那么你可以通过使用表空间把一些数据库文件移动到其他文件系统上去。

> 提示
>
> 有些文件系统在快满的时候性能会急剧恶化，因此不要等到磁盘完全满的时候才采取行动。

如果你的系统支持每用户的磁盘份额，那么数据库将自然地受制于用户所处的服务器给他的份额限制。超过份额的负面影响和完全用光磁盘是完全一样的。

.2. 日常维护

日常清理

IvorySQL数据库要求周期性的清理维护。对于很多安装，让自动清理守护进程来执行清理已经足够，你也可能需要调整其中描述的自动清理参数来获得最佳结果。某些数据库管理员会希望使用手动管理的`VACUUM`命令来对后台进程的活动进行补充或者替换，这通常使用cron或任务计划程序脚本来执行。要正确地设置手动管理的清理。

清理的基础知识

IvorySQL的 VACUUM命令出于几个原因必须定期处理每一个表：

1. 恢复或重用被已更新或已删除行所占用的磁盘空间。
2. 更新被IvorySQL查询规划器使用的数据统计信息。
3. 更新可见性映射，它可以加速只用索引的扫描。
4. 保护老旧数据不会由于事务ID回卷或多事务ID回卷而丢失。

正如后续小节中解释的，每一个原因都将指示以不同的频率和范围执行`VACUUM`操作。

有两种`VACUUM`的变体：标准`VACUUM`和`VACUUM FULL`。**`VACUUM FULL`可以收回更多磁盘空间但是运行起来更慢。另外，标准形式的`VACUUM`可以和生产数据库操作并行运行（`SELECT`、`INSERT`、`UPDATE`和`DELETE`等命令将继续正常工作，但在清理期间你无法使用`ALTER TABLE`等命令来更新表的定义）。`VACUUM FULL`要求在其工作的表上得到一个`ACCESS EXCLUSIVE`**锁，因此无法和对此表的其他使用并行。因此，通常管理员应该努力使用标准`VACUUM`并且避免`VACUUM FULL`。

`VACUUM`会产生大量I/O流量，这将导致其他活动会话性能变差。可以调整一些配置参数来后台清理活动造成的性能冲击.

恢复磁盘空间

在IvorySQL中，一次行的`UPDATE`或`DELETE`不会立即移除该行的旧版本。这种方法对于从多版本并发控制获益是必需的：当旧版本仍可能对其他事务可见时，它不能被删除。但是最后，任何事务都不会再对一个过时的或者被删除的行版本感兴趣。它所占用的空间必须被回收来用于新行，这样可避免磁盘空间需求的无限制增长。这通过运行`VACUUM`完成。

`VACUUM`的标准形式移除表和索引中的死亡行版本并将该空间标记为可在未来重用。不过，它将不会把该

空间交还给操作系统，除非在特殊的情况中表尾部的一个或多个页面变成完全空闲并且能够很容易地得到一个排他表锁。相反，`VACUUM FULL`通过把死亡空间之外的内容写成一个完整的新版本表文件来主动紧缩表。这将最小化表的尺寸，但是要花较长的时间。它也需要额外的磁盘空间用于表的新副本，直到操作完成。

例行清理的一般目标是多做标准的`VACUUM`来避免需要`VACUUM FULL`。自动清理守护进程尝试这样工作，并且实际上永远不会发出`VACUUM FULL`。在这种方法中，其思想不是让表保持它们的最小尺寸，而是保持磁盘空间使用的稳定状态：每个表占用的空间等于其最小尺寸外加清理之间将使用的空间量。尽管`VACUUM FULL`可被用来把一个表收缩回它的最小尺寸并将该磁盘空间交还给操作系统，但是如果该表将在未来再次增长这样就没什么意义。因此，对于维护频繁被更新的表，适度运行标准`VACUUM`运行比少量运行`VACUUM FULL`要更好。

一些管理员更喜欢自己计划清理，例如在晚上负载低时做所有的工作。根据一个固定日程来做清理的难点在于，如果一个表有一次预期之外的更新活动尖峰，它可能膨胀得真正需要`VACUUM FULL`来回收空间。使用自动清理守护进程可以减轻这个问题，因为守护进程会根据更新活动动态规划清理操作。除非你的负载是完全可以预估的，完全禁用守护进程是不理智的。一种可能的折中方案是设置守护进程的参数，这样它将只对异常的大量更新活动做出反应，因而保证事情不会失控，而在负载正常时采用有计划的`VACUUM`来做批量工作。

对于那些不使用自动清理的用户，一种典型的方法是计划一个数据库范围的`VACUUM`，该操作每天在低使用量时段执行一次，并根据需要辅以在重度更新表上的更频繁的清理（一些有着极高更新率的安装会每几分钟清理一次它们的最繁忙的表）。如果你在一个集簇中有多个数据库，别忘记`VACUUM`每一个，你会用得上vacuumdb程序。

> 提示
>
> 当一个表因为大量更新或删除活动而包含大量死亡行版本时，纯粹的`VACUUM`可能不能令人满意。如果你有这样一个表并且你需要回收它占用的过量磁盘空间，你将需要使用`VACUUM FULL`，或者CLUSTER，或者ALTER TABLE的表重写变体之一。这些命令重写该表的一整个新拷贝并且为它构建新索引。所有这些选项都要求`ACCESS EXCLUSIVE`锁。注意它们也临时使用大约等于该表尺寸的额外磁盘空间，因为直到新表和索引完成之前旧表和索引都不能被释放。

> 提示
>
> 如果你有一个表，它的整个内容会被周期性删除，考虑用TRUNCATE而不是先用`DELETE`再用`VACUUM`。`TRUNCATE`会立刻移除该表的整个内容，而不需要一次后续的`VACUUM`或`VACUUM FULL`来回收现在未被使用的磁盘空间。其缺点是会违背严格的 MVCC 语义。

更新规划器统计信息

IvorySQL查询规划器依赖于有关表内容的统计信息来为查询产生好的计划。这些统计信息由ANALYZE命令收集，它除了直接被调用之外还可以作为`VACUUM`的一个可选步骤被调用。拥有适度准确的统计信息很重要，否则差的计划可能降低数据库性能。

自动清理守护进程如果被启用，当一个表的内容被改变得足够多时，它将自动发出`ANALYZE`命令。不过，管理员可能更喜欢依靠手动的`ANALYZE`操作，特别是如果知道一个表上的更新活动将不会影响“感兴趣的”列的统计信息时。守护进程严格地按照一个被插入或更新行数的函数来计划`ANALYZE`，它不知道那是否将导致有意义的统计信息改变。

正如用于空间恢复的清理一样，频繁更新统计信息对重度更新的表更加有用。但即使对于一个重度更新的表，如果该数据的统计分布没有很大改变，也没有必要更新统计信息。一个简单的经验法则是考虑表中列的最大和最小值改变了多少。例如，一个包含行被更新时间的`timestamp`列将在行被增加和更新时有一直增加的最大值；这样一列将可能需要更频繁的统计更新，而一个包含一个网站上被访问页面 URL 的列则不需要。URL 列可以经常被更改，但是其值的统计分布的变化相对很慢。

可以在指定表上运行`ANALYZE`甚至在表的指定列上运行，因此如果你的应用需要，可以更加频繁地更新某

些统计。但实际上，通常只分析整个数据库是最好的，因为它是一种很快的操作。`ANALYZE`对一个表的行使用一种统计的随机采样，而不是读取每一个单一行。

> 提示
>
> 尽管对每列的`ANALYZE`频度调整可能不是非常富有成效，你可能会发现值得为每列调整被`ANALYZE`收集统计信息的详细程度。经常在`WHERE`中被用到的列以及数据分布非常不规则的列可能需要比其他列更细粒度的数据直方图。见`ALTER TABLE SET STATISTICS`，或者使用default_statistics_target配置参数改变数据库范围的默认值。

> 提示
>
> 自动清理守护进程不会为外部表发出`ANALYZE`命令，因为无法确定一个合适的频度。如果你的查询需要外部表的统计信息来正确地进行规划，比较好的方式是按照一个合适的时间表在那些表上手工运行`ANALYZE`命令。

> 提示
>
> autovacuum守护进程不会对分区表发出ANALYZE命令。继承性父表只有在父表本身发生变化时才会被分析—
> 对子表的变化不会触发对父表的自动分析。如果你的查询需要对父表进行统计以进行正确的规划，那么有必要定期对这些表运行手动ANALYZE以保持统计的最新性。

更新可见性映射

清理机制为每一个表维护着一个可见性映射，它被用来跟踪哪些页面只包含对所有活动事务（以及所有未来的事务，直到该页面被再次修改）可见的元组。这样做有两个目的。第一，清理本身可以在下一次运行时跳过这样的页面，因为其中没有什么需要被清除。

第二，这允许IvorySQL回答一些只用索引的查询，而不需要引用底层表。因为IvorySQL的索引不包含元组的可见性信息，一次普通的索引扫描会为每一个匹配的索引项获取堆元组，用来检查它是否能被当前事务所见。另一方面，一次*只用索引的扫描*会首先检查可见性映射。如果它了解到在该页面上的所有元组都是可见的，堆获取就可以被跳过。这对大数据集很有用，因为可见性映射可以防止磁盘访问。可见性映射比堆小很多，因此即使堆非常大，可见性映射也可以很容易地被缓存起来。

防止事务ID回卷失败

IvorySQL的 MVCC 事务语义依赖于能够比较事务 ID（XID）数字：如果一个行版本的插入 XID 大于当前事务的 XID，它就是“属于未来的”并且不应该对当前事务可见。但是因为事务 ID
的尺寸有限（32位），一个长时间（超过 40 亿个事务）运行的集簇会遭受到*事务 ID 回卷*问题：XID
计数器回卷到 0，并且本来属于过去的事务突然间就变成了属于未来 —
这意味着它们的输出变成不可见。简而言之，灾难性的数据丢失（实际上数据仍然在那里，但是如果你不能得到它也无济于事）。为了避免发生这种情况，有必要至少每 20 亿个事务就清理每个数据库中的每个表。

周期性的清理能够解决该问题的原因是，VACUUM会把行标记为
冻结，这表示它们是被一个在足够远的过去提交的事务所插入，这样从 MVCC
的角度来看，效果就是该插入事务对所有当前和未来事务来说当然都 是可见的。IvorySQL保留了一个特殊的 XID （FrozenTransactionId），这个 XID 并不遵循普通 XID 的比较规则 并且总是被认为比任何普通 XID
要老。普通 XID 使用模-232算 法来比较。这意味着对于每一个普通 XID都有 20 亿个 XID “更老”并且 有 20 亿个“更新”，另一种解释的方法是普通 XID
空间是没有端点的环。因此，一旦一个行版本创建时被分配了一个特定的普通 XID，该行版本将成为接下 来 20 亿个事务的“过去”（与我们谈论的具体哪个普通 XID 无关）。如果在 20
亿个事务之后该行版本仍然存在，它将突然变得好像在未来。要阻止这一切
发生，被冻结行版本会被看成其插入 XID 为`FrozenTransactionId`，
这样它们对所有普通事务来说都是“在过去”，而不管回卷问题。并且这样的行版本将一直有效直到被删除，不管它有多旧。

vacuum_freeze_min_age控制在其行版本被冻结前一个 XID
值应该有多老。如果被冻结的行将很快会被再次修改，增加这个设置可以避免不必要
的工作。但是减少这个设置会增加在表必须再次被清理之前能够流逝的事务数。

`VACUUM`通常会跳过不含有任何死亡行版本的页面，但是不会跳过那些含有带旧 XID
值的行版本的页面。要保证所有旧的行版本都已经被冻结，需要对整个表做一次扫描。vacuum_freeze_table_age控制`VACUUM`什么时候这样做：如果该表经过`vacuum_freeze_table_age`减去`vacuum_freeze_min_age`个事务还没有被完全扫描过，则会强制一次全表清扫。将这个参数设置为 0
将强制`VACUUM`总是扫描所有页面而实际上忽略可见性映射。

一个表能保持不被清理的最长时间是 20
亿个事务减去`VACUUM`上次扫描全表时的`vacuum_freeze_min_age`值。如果它超过该时间没有被清理，可能会导致数据丢失。要保证这不会发生，将在任何包含比autovacuum_freeze_max_age配置参数所指定的年龄更老的 XID 的未冻结行的表上调用自动清理（即使自动清理被禁用也会发生）。

这意味着如果一个表没有被清理，大约每`autovacuum_freeze_max_age`减去`vacuum_freeze_min_age`事务就会在该表上调用一次自动清理。对那些为了空间回收目的而被正常清理的表，这是无关紧要的。然而，对静态表（包括接收插入但没有更新或删除的表）就没有为空间回收而清理的需要，因此尝试在非常大的静态表上强制自动清理的间隔最大化会非常有用。显然我们可以通过增加`autovacuum_freeze_max_age`或减少`vacuum_freeze_min_age`来实现此目的。

vacuum_freeze_table_age的实际最大值是 0.95 *
autovacuum_freeze_max_age，高于它的设置将被上限到最大值。一个高于`autovacuum_freeze_max_age`的值没有意义，因为不管怎样在那个点上都会触发一次防回卷自动清理，并且 0.95
的乘数为在防回卷自动清理发生之前运行一次手动`VACUUM`留出了一些空间。作为一种经验法则，`vacuum_freeze_table_age`应当被设置成一个低于`autovacuum_freeze_max_age`的值，留出一个足够的空间让一次被正常调度的`VACUUM`或一次被正常删除和更新活动触发的自动清理可以在这个窗口中被运行。将它设置得太接近可能导致防回卷自动清理，即使该表最近因为回收空间的目的被清理过，而较低的值将导致更频繁的全表扫描。

增加`autovacuum_freeze_max_age`（以及和它一起的`vacuum_freeze_table_age`）的唯一不足是数据库集簇的`pg_xact`和`pg_commit_ts`子目录将占据更多空间，因为它必须存储所有向后`autovacuum_freeze_max_age`范围内的所有事务的提交状态和（如果启用了`track_commit_timestamp`）时间戳。提交状态为每个事务使用两个二进制位，因此如果`autovacuum_freeze_max_age`被设置为它的最大允许值 20
亿，`pg_xact`将会增长到大约 0.5
吉字节，`pg_commit_ts`大约20GB。如果这对于你的总数据库尺寸是微小的，我们推荐设置`autovacuum_freeze_max_age`为它的最大允许值。否则，基于你想要允许`pg_xact`和`pg_commit_ts`使用的存储空间大小来设置它（默认情况下 2 亿个事务大约等于`pg_xact`的 50
MB存储空间，`pg_commit_ts`的2GB的存储空间）。

减小`vacuum_freeze_min_age`的一个不足之处是它可能导致`VACUUM`做无用的工作：如果该行在被替换成`FrozenXID`之后很快就被修改（导致该行获得一个新的
XID），那么冻结一个行版本就是浪费时间。因此该设置应该足够大，这样直到行不再可能被修改之前，它们都不会被冻结。

为了跟踪一个数据库中最老的未冻结 XID 的年龄，`VACUUM`在系统表`pg_class`和`pg_database`中存储 XID
的统计信息。特别地，一个表的`pg_class`行的`relfrozenxid`列包含被该表的上一次全表`VACUUM`所用的冻结截止 XID。该表中所有被有比这个截断 XID 老的普通 XID 的事务插入的行
都确保被冻结。相似地，一个数据库的`pg_database`行的`datfrozenxid`列是出现在该数据库中的未冻结 XID 的下界 —
它只是数据库中每一个表的`relfrozenxid`值的最小值。一种检查这些信息的方便方法是执行这样的查询：

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');
```

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

`age`列度量从该截断 XID 到当前事务 XID 的事务数。

`VACUUM`通常只扫描从上次清理后备修改过的页面，但是只有当全表被扫描时`relfrozenxid`才能被推进。当`relfrozenxid`比`vacuum_freeze_table_age`个事务还老时、当`VACUUM`的`FREEZE`选项被使用时或当所有页面正好要求清理来移除死亡行版本时，全表将被扫描。当`VACUUM`扫描全表时，在它被完成后，`age(relfrozenxid)`应该比被使用的`vacuum_freeze_min_age`设置略大（比在`VACUUM`开始后开始的事务数多）。如果在`autovacuum_freeze_max_age`被达到之前没有全表扫描`VACUUM`在该表上被发出，将很快为该表强制一次自动清理。

如果出于某种原因自动清理无法从一个表中清除旧的 XID，当数据库的最旧 XID 和回卷点之间达到 4
千万个事务时，系统将开始发出这样的警告消息：

```
WARNING:  database "mydb" must be vacuumed within 39985967 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in that database.
```

（如该示例所建议的，一次手动的`VACUUM`应该会修复该问题；但是注意该次`VACUUM`必须由一个超级用户来执行，否则它将无法处理系统目录并且因而不能推进数据库的`datfrozenxid`）。如果这些警告被忽略，一旦距离回卷点只剩下 3 百万个事务时，该系统将会关闭并且拒绝开始任何新的事务：

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database
"mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

这 3
百万个事务的安全余量是为了让管理员能通过手动执行所要求的`VACUUM`命令进行恢复而不丢失数据。但是，由于一旦系统进入到安全关闭模式，它将不会执行命令。做这个操作的唯一方法是停止服务器并且以单一用户启动服务器来执行`VACUUM`。单一用户模式中不会强制该关闭模式。

*Multixact ID*被用来支持被多个事务锁定的行。由于在一个元组头部
只有有限的空间可以用来存储锁信息，所以只要有多于一个事务并发地锁住一个行，
锁信息将使用一个“多个事务 ID”（或简称多事务 ID）来编码。任何特定 多事务 ID 中包括的事务 ID
的信息被独立地存储在`pg_multixact`子目 录中，并且只有多事务 ID
出现在元组头部的`xmax`域中。和事务 ID 类似，多事务 ID 也是用一个 32
位计数器实现，并且也采用了相似的存储，这些都要
求仔细的年龄管理、存储清除和回卷处理。在每个多事务中都有一个独立的存储区域
保存成员列表，它也使用一个 32 位计数器并且也应被管理。

在一次`VACUUM`表扫描（部分或者全部）期间，任何比 vacuum_multixact_freeze_min_age 要老的多事务 ID 会被替换为一个不同的值，该值可以是零值、 一个单一事务 ID 或者一个更新的多事务 ID。
对于每一个表，`pg_class.relminmxid` 存储了在该表任意元组中仍然存在的最老可能多事务
ID。如果这个值比 vacuum_multixact_freeze_table_age老， 将强制一次全表扫描。可以在 `pg_class.relminmxid` 上使用`mxid_age()`来找到它的年龄。

全表`VACUUM`扫描（不管是什么导致它们）将为表推进该值。
最后，当所有数据库中的所有表被扫描并且它们的最老多事务值被推进，
较老的多事务的磁盘存储可以被移除。

作为一种安全设备，对任何多事务年龄超过 autovacuum_multixact_freeze_max_age的表，
都将发生一次全表清理扫描。当多事务成员占用的存储超过 2GB
时，从那些具有最老多事务年龄的表开始，全表清理扫描也将逐步在所有表上进行。即使自动清理被
在名义上被禁用，也会发生这两种主动扫描。

自动清理后台进程

IvorySQL有一个可选的但是被高度推荐的特性*autovacuum*，它的目的是自动执行`VACUUM`和`ANALYZE`命令。当它被启用时，自动清理会检查被大量插入、更新或删除元组的表。这些检查会利用统计信息收集功能，因此除非track_counts被设置为`true`，自动清理不能被使用。在默认配置下，自动清理是被启用的并且相关配置参数已被正确配置。

“自动清理后台进程”实际上由多个进程组成。有一个称为 自动清理启动器*的常驻后台进程，它负责为所有数据库启动*自动清理工作者*进程。 启动器将把工作散布在一段时间上，它每隔autovacuum_naptime秒尝试在每个数据库中启动一个工作者 （因此，如果安装中有N*个数据库，则每autovacuum_naptime/*N秒将启动一个新的工作者）。
在同一时间只允许最多autovacuum_max_workers个工作者进程运行。如果有超过`autovacuum_max_workers`个数据库需要被处理，下一个数据库将在第一个工作者结束后马上被处理。
每一个工作者进程将检查其数据库中的每一个表并且在需要时执行 VACUUM`和/或`ANALYZE。
可以设置log_autovacuum_min_duration来监控自动清理工作者的活动。

如果在一小段时间内多个大型表都变得可以被清理，所有的自动清理工作者可能都会被占用来在一段长的时间内清理这些表。这将会造成其他的表和数据库无法被清理，直到一个工作者变得可用。对于一个数据库中的工作者数量并没有限制，但是工作者确实会试图避免重复已经被其他工作者完成的工作。注意运行着的工作者的数量不会被计入max_connections或superuser_reserved_connections限制。

`relfrozenxid`值比autovacuum_freeze_max_age事务年龄更大的表总是会被清理（这页表示这些表的冻结最大年龄被通过表的存储参数修改过，参见后文）。否则，如果从上次`VACUUM`以来失效的元组数超过“清理阈值”，表也会被清理。清理阈值定义为：

```
清理阈值 = 清理基本阈值 + 清理缩放系数 * 元组数
```

其中清理基本阈值为autovacuum_vacuum_threshold，
清理缩放系数为autovacuum_vacuum_scale_factor， 元组数为`pg_class`.reltuples。

如果自上次清理以来插入的元组数量超过了定义的插入阈值，表也会被清理，该阈值定义为:

```
清理插入阈值 = 清理基础插入阈值 + 清理插入缩放系数 * 元组数
```

清理插入基础阈值为autovacuum_vacuum_insert_threshold，清理插入缩放系数为autovacuum_vacuum_insert_scale_factor。 这样的清理可以允许部分的表被标识为*all visible*，并且也可以允许元组被冻结，可以减小后续清理的工作需要。
对于可以接收`INSERT`操作但是不能或几乎不能`UPDATE`/DELETE`操作的表，
可能会从降低表的autovacuum_freeze_min_age中受益，因为这可能允许元组在早期清理中被冻结。
废弃元组的数量和插入元组的数量可从统计收集器中获得；它是一个半精确的计数，由每个`UPDATE、DELETE和 INSERT 操作进行更新。（它只是半精确的，因为一些信息可能会在重负载情况下丢失。）
如果表的`relfrozenxid`值大于`vacuum_freeze_table_age` 事务老的，
执行一个主动的清理来冻结旧的元组，并推进`relfrozenxid`;否则，只有上次清理以后修改过的页面被扫描。

对于分析，也使用了一个相似的阈值：

```
分析阈值 = 分析基本阈值 + 分析缩放系数 * 元组数
```

该阈值将与自从上次`ANALYZE`以来被插入、更新或删除的元组数进行比较。

临时表不能被自动清理访问。因此，临时表的清理和分析操作必须通过会话期间的SQL命令来执行。

默认的阈值和缩放系数都取自于`postgresql.conf`，但是可以为每一个表重写它们(和许多其他自动清理控制参数)， 详情参见Storage Parameters。
如果一个设置已经通过一个表的存储参数修改，那么在处理该表时使用该值，否则使用全局设置。

当多个工作者运行时，在所有运行着的工作者之间自动清理代价延迟参数是“平衡的”，这样不管实际运行的工作者数量是多少， 对于系统的总体 I/O 影响总是相同的。不过，任何正在处理已经设置了每表 `autovacuum_vacuum_cost_delay`或`autovacuum_vacuum_cost_limit` 存储参数的表的工作者不会被考虑在均衡算法中。

autovacuum工作进程通常不会阻止其他命令。如果某个进程尝试获取与autovacuum持有的`SHARE UPDATE EXCLUSIVE`锁冲突的锁，则锁获取将中断该autovacuum。有关冲突的锁定模式，但是，如果autovacuum正在运行以防止事务ID回卷（即在`pg_stat_activity`视图中的autovacuum查询名以`(to prevent wraparound)`结尾），则autovacuum不会被自动中断。

> 警告
>
> 定期运行需要获取与`SHARE UPDATE EXCLUSIVE`锁冲突的锁的命令（例如ANALYZE）可能会让autovacuum始终无法完成。

日常重建索引

在某些情况下值得周期性地使用REINDEX命令或一系列独立重构步骤来重建索引。

已经完全变成空的B树索引页面被收回重用。但是，还是有一种低效的空间利用的可能性：如果一个页面上除少量索引键之外的全部键被删除，该页面仍然被分配。因此，在这种每个范围中大部分但不是全部键最终被删除的使用模式中，可以看到空间的使用是很差的。对于这样的使用模式，推荐使用定期重索引。

对于非B树索引可能的膨胀还没有很好地定量分析。在使用非B树索引时定期监控索引的物理尺寸是个好主意。

还有，对于B树索引，一个新建立的索引比更新了多次的索引访问起来要略快，因为在新建立的索引上，逻辑上相邻的页面通常物理上也相邻（这样的考虑目前并不适用于非B树索引）。仅仅为了提高访问速度也值得定期重索引。

REINDEX在所有情况下都可以安全和容易地使用。 默认情况下，此命令需要一个`ACCESS EXCLUSIVE`锁，因此通常最好使用`CONCURRENTLY`选项执行它，该选项仅需要获取`SHARE UPDATE EXCLUSIVE`锁。

日志文件维护

把数据库服务器的日志输出保存在一个地方是个好主意， 而不是仅仅通过`/dev/null`丢弃它们。在进行问题诊断的时候，日志输出是非常宝贵的。不过，日志输出可能很庞大（特别是在比较高的调试级别上）， 因此你不会希望无休止地保存它们。你需要轮转日志文件，这样在一段合理的时间后会开始新的日志文件并且移除旧的。

如果你简单地把`postgres`的stderr定向到一个文件中，你会得到日志输出，但是截断该日志文件的唯一方法是停止并重起服务器。这样做对于开发环境中使用的IvorySQL可能是可接受的，但是你肯定不想在生产环境上这么干。

一个更好的办法是把服务器的stderr输出发送到某种日志轮转程序里。我们有一个内建的日志轮转程序，你可以通过在`postgresql.conf`里设置配置参数`logging_collector`为`true`的办法启用它。你也可以使用这种方法把日志数据捕捉成机器可读的CSV（逗号分隔值）格式。

另外，如果在你已经使用的其他服务器软件中有一个外部日志轮转程序，你可能更喜欢使用它。比如，包含在Apache发布里的rotatelogs工具就可以用于IvorySQL。要做到这一点，方法之一是把服务器的stderr用管道重定向到要用的程序。 如果你用`pg_ctl`启动服务器，那么stderr已经重定向到stdout，因此你只需要一个管道命令，比如：

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

您可以通过设置logrotate来收集由IvorySQL内置日志收集器生成的日志文件来组合这些方法。在这种情况下，日志收集器定义日志文件的名称和位置，而logrotate
则定期归档这些文件。启动日志轮转时，logrotate必须确保应用程序将进一步的输出发送到新文件。这通常是通过`postrotate`脚本完成的，该脚本向应用程序发送`SIGHUP`信号，使其重新打开日志文件。在IvorySQL中，您可以使用`logrotate`选项运行`pg_ctl`。服务器收到此命令后，服务器将切换到新的日志文件或重新打开现有文件，具体取决于日志记录配置。

> 注意
>
> 定期运行需要获取与`SHARE UPDATE
> EXCLUSIVE`锁冲突的锁的命令（例如ANALYZE）可能会让autovacuum始终无法完成使用静态日志文件名时，如果达到最大打开文件数限制或发生文件表溢出，则服务器可能无法重新打开日志文件。
> 在这种情况下，日志消息将发送到旧的日志文件，直到成功进行日志轮转为止。
> 如果将logrotate配置为压缩日志文件并将其删除，则服务器可能会丢失此时间范围内记录的消息。
> 为避免此问题，可以将日志收集器配置为动态分配日志文件名，并使用`prerotate`脚本忽略打开的日志文件。

另外一种生产级的管理日志输出的方法就是把它们发送给syslog，让syslog处理文件轮转。
要利用这个工具，我们需要设置`postgresql.conf`里的`log_destination`配置参数设置为`syslog`（记录`syslog`日志）。然后在你想强迫syslog守护进程开始写入一个新日志文件的时候， 你就可以发送一个`SIGHUP`信号给它。 如果你想自动进行日志轮转，可以配置logrotate程序处理 来自syslog的日志文件。

不过，在很多系统上，syslog不是非常可靠，特别是在面对大量日志消息的情况下；
它可能在你最需要那些消息的时候截断或者丢弃它们。另外，在Linux，syslog会把每个消息刷写到磁盘上，这将导致很差的性能（你可以在syslog配置文件里面的文件名开头使用一个“-”来禁用这种行为）。

请注意上面描述的所有解决方案关注的是在可配置的间隔上开始一个新的日志文件，
但它们并没有处理对旧的、不再需要的日志文件的删除。你可能还需要设置一个批处理任务来定期地删除旧日志文件。 另一种可能的方法是配置日志轮转程序，让它循环地覆盖旧的日志文件。

pgBadger
是一个外部项目，它可以进行日志文件的深度分析。check_postgres可在重要消息出现在日志文件中时向Nagios提供警告，也可以探测很多其他的特别情况。

高可用、负载均衡和复制

不同方案的比较

共享磁盘故障转移

共享磁盘故障转移避免了只使用一份数据库拷贝带来的同步开销。它使用一个由多个服务器共享的单一磁盘阵列。如果主数据库服务器失效，后备服务器则可以挂载并启动数据库，就好像它从一次数据库崩溃中恢复过来了。这是一种快速的故障转移，并且不存在数据丢失。

共享硬件功能在网络存储设备中很常见。也可以使用一个网络文件系统，但是要注意的是该文件系统应具有完全的POSIX行为。这种方法的一个重大限制是如果共享磁盘阵列失效或损坏，主要和后备服务器都会变得无法工作。另一个问题是在主要服务器运行时，后备服务器永远不能访问共享存储。

文件系统（块设备）复制

共享硬件功能的一种修改版本是文件系统复制，在其中对一个文件系统的所有改变会被镜像到位于另一台计算机上的一个文件系统。唯一的限制是该镜像过程必须能保证后备服务器有一份该文件系统的一致的拷贝 — 特别是对后备服务器的写入必须按照主控机上相同的顺序进行。DRBD是用于 Linux
的一种流行的文件系统复制方案。

预写式日志传送

温备和热备服务器能够通过读取一个预写式日志（WAL）记录的流来保持为当前状态。如果主服务器失效，后备服务器拥有主服务器的几乎所有数据，并且能够快速地被变成新的主数据库服务器。这可以是同步的或异步的，并且只能用于整个数据库服务器。

可以使用基于文件的日志传送、流复制或两者的组合来实现一个后备服务器。

逻辑复制

逻辑复制允许数据库服务器发送数据更新流给另一台服务器。IvorySQL逻辑复制从WAL构建出逻辑数据更新流。逻辑复制允许您逐个表复制数据更改。此外，发布数据更新的服务器可以同时订阅其他服务器的更改，从而允许数据在多个方向流动。第三方扩展也能提供类似的功能。

基于触发器的主-备复制

基于触发器的复制通常会将修改数据的查询发送到指定的主服务器。它在逐个表的基础上工作，主服务器（通常）将数据更改异步发送到后备服务器。
主服务器运行时，后备服务器可以响应查询，并执行本地数据修改或写入操作。这种形式的复制通常用于减轻大数据分析型平台或者数据仓库查询负荷。

Slony-
I是这种复制类型的一个例子。它使用表粒度，并且支持多个后备服务器。因为它会异步更新后备服务器（批量），在故障转移时可能会有数据丢失。

基于SQL的复制中间件

通过基于SQL的复制中间件，一个程序拦截每一个 SQL
查询并把它发送给一个或所有服务器。每一个服务器独立地操作。读写查询必须被发送给所有服务器，这样每一个服务器都能接收到任何修改。但只读查询可以只发送给一个服务器，这样允许读负载在服务器之间分布。

如果查询未经修改发送，则函数`random()`函数的随机值和`CURRENT_TIMESTAMP`函数的当前时间和序列值可能因不同服务器而异。 因为每个服务器独立运行，并且它发送 SQL
查询而没有真正的更改数据。如果这是不可接受的，那么中间件或应用程序必须从单一服务器源确定此类值，并将结果用于写入查询。 还必须注意确保所有服务器在提交或中止事务时都是相同的。这将涉及使用两阶段提交PREPARE TRANSACTION和COMMIT PREPARED。 Pgpool-II和Continuent
Tungsten就是这种复制的例子。

异步多主控机复制

对于不会被定期连接或通讯链路较慢的服务器，如笔记本或远程服务器，保持服务器间的数据一致是一个挑战。通过使用异步的多主控机复制，每一个服务器独立工作并且定期与其他服务器通信来确定冲突的事务。这些冲突可以由用户或冲突解决规则来解决。Bucardo 是这种复制类型的一个例子。

同步多主控机复制

在同步多主控机复制中，每一个服务器能够接受写请求，并且在每一个事务提交之前，被修改的数据会被从原始服务器传送给每一个其他服务器。繁重的写活动可能导致过多的锁定和提交延迟，进而导致很差的性能。读请求可以被发送给任意服务器。某些实现使用共享磁盘来减少通信负荷。同步多主控机复制主要对于读负载最好，尽管它的大优点是任意服务器都能接受写请求 —
没有必要在主服务器和后备服务器之间划分负载，并且因为数据修改被从一个服务器发送到另一个服务器，不会有非确定函数（如`random()`）的问题。

IvorySQL不提供这种复制类型，尽管在应用代码或中间件中可以使用 IvorySQL 的两阶段提交PREPARE TRANSACTION和COMMIT PREPARED来实现这种复制。

下表总结了上述多种方案的能力。

特性	共享磁盘	文件系统复制	预写式日志传送	逻辑复制	基于触发器的复制	SQL复制中间件	异步多主控机复制	同步多主控机复制
常用的示例	NAS	DRBD	内建流复制	内建逻辑复制，pglogical	Londiste，Slony	pgpool-II	Bucardo	
通信方法	共享磁盘	磁盘块	WAL	逻辑解码	表行	SQL	表行	表行和行锁

不要求特殊硬件		•	•	•	•	•	•	•
允许多个主控机服务器				•		•	•	•
无主服务器负载	•		•	•		•		
不等待多个服务器	•		with sync off	with sync off	•		•	
主控机失效将永不丢失数据	•	•	with sync on	with sync on		•		•
复制体接受只读查询			with hot	•	•	•	•	•
每个表粒度				•	•		•	•
不需要冲突解决	•	•	•		•	•		•

有一些方案不适合上述的类别：

- 数据分区

  数据分区将表分开成数据集。每个集合只能被一个服务器修改。例如，数据可以根据办公室划分，如伦敦和巴黎，每一个办公室有一个服务器。如果查询有必要组合伦敦和巴黎的数据，一个应用可以查询两个服务器，或者可以使用主/备复制来在每一台服务器上保持其他办公室数据的一个只读拷贝。

- 多服务器并行查询执行

  上述的很多方案允许多个服务器来处理多个查询，但是没有一个允许一个单一查询使用多个服务器来更快完成。 这种方案允许多个服务器在一个单一查询上并发工作。
  这通常通过把数据在服务器之间划分并且让每一个服务器执行该查询中属于它的部分，然后将结果返回给一个中心服务器，由它整合结果并发回给用户。
  这也可以使用PL/Proxy工具集来实现这种方案。

日志传送后备服务器

规划

创建主服务器和后备服务器通常是明智的，因此它们可以尽可能相似，至少从数据库服务器的角度来看是这样。特别地，与表空间相关的路径名将被未经修改地传递，因此如果该特性被使用，主、备服务器必须对表空间具有完全相同的挂载路径。记住如果CREATE TABLESPACE在主服务器上被执行，在命令被执行前，它所需要的任何新挂载点必须在主服务器和所有后备服务器上先创建好。硬件不需要完全相同，但是经验显示，在应用和系统的生命期内维护两个相同的系统比维护两个不相似的系统更容易。在任何情况下硬件架构必须相同 — 从一个 32 位系统传送到一个 64 位系统将不会工作。

通常，不能在两个运行着不同主版本IvorySQL的服务器之间传送日志。IvorySQL 全球开发组的策略是不在次版本升级中改变磁盘格式，因此在主服务器和后备服务器上运行不同次版本将会成功地工作。不过，在这方面并没有提供正式的支持，因此我们建议让主备服务器上运行的版本尽可能相同。当升级到一个新的次版本时，最安全的策略是先升级后备服务器 —

一个新的次版本发行更可能兼容从前一个次版本读取 WAL 文件。

后备服务器操作

服务器启动时，数据目录中存在 `standby.signal` 文件，服务器进入standby模式。

在后备模式中，服务器持续地应用从主控服务器接收到的 WAL。后备服务器可以从一个 WAL 归档restore_command或者通过一个 TCP 连接直接从主控机（流复制）读取 WAL。后备服务器将也尝试恢复在后备集簇的`pg_wal`目录中找到的 WAL。那通常在一次数据库重启后发生，那时后备机将在重启之前重播从主控机流过来的 WAL，但是你也可以在任何时候手动拷贝文件到`pg_wal`让它们被重播。

在启动时，后备机通过恢复归档位置所有可用的 WAL 来开始，这称为`restore_command`。一旦它到达那里可用的 WAL 的末尾并且`restore_command`失败，它会尝试恢复`pg_wal`目录中可用的任何 WAL。如果那也失败并且流复制已被配置，后备机会尝试连接到主服务器并且从在归档或`pg_wal`中找到的最后一个可用记录开始流式传送 WAL。如果那失败并且没有配置流复制，或者该连接后来断开，后备机会返回到步骤 1 并且尝试再次从归档里的文件恢复。这种尝试归档、`pg_wal`和流复制的循环会一直重复知道服务器停止或者一个触发器文件触发了故障转移。

当`pg_ctl promote`被运行，`pg_promote()`被调用，或一个触发器文件被找到（`promote_trigger_file`），后备模式会退出并且服务器会切换到普通操作。 在故障转移之前，在归档或`pg_wal`中立即可用的任何 WAL 将被恢复，但不会尝试连接到主控机。

为后备服务器准备主控机

在主服务器上设置连续归档到一个后备服务器可访问的归档目录。即使主服务器垮掉该归档位置也应当是后备服务器可访问的，即它应当位于后备服务器本身或者另一个可信赖的服务器，而不是位于主控服务器上。

如果你想要使用流复制，在主服务器上设置认证来允许来自后备服务器的复制连接。即创建一个角色并且在`pg_hba.conf`中提供一个或多个数据库域被设置为`replication`的项。还要保证在主服务器的配置文件中`max_wal_senders`被设置为足够大的值。如果要使用复制槽，请确保`max_replication_slots`也被设置得足够高。

建立一个后备服务器

要建立后备服务器，恢复从主服务器取得的基础备份。在后备服务器的集簇数据目录中创建一个文件standby.signal。将restore_command设置为一个从 WAL 归档中复制文件的简单命令。
如果你计划为了高可用性目的建立多个后备服务器，确认`recovery_target_timeline`被设置为`latest`(默认)来使得该后备服务器遵循发生在故障转移到另一个后备服务器之后发生的时间线改变。

> **注意**
>
> restore_command应该立即返回，如果必要该服务器将再次尝试该命令。

如果你想要使用流复制，在primary_conninfo中填入一个 libpq 连接字符串，其中包括主机名（或 IP 地址）和连接到主服务器所需的任何附加细节。如果主服务器需要一个口令用于认证，口令也应该被指定primary_conninfo中。

如果你正在为高性能目的建立后备服务器，像主服务器一样建立 WAL 归档、连接和认证，因为在故障转移后该后备服务器将作为一个主服务器工作。

如果你正在使用一个 WAL 归档，可以使用archive_cleanup_command参数来移除后备服务器不再需要的文件，这样可以最小化 WAL 归档的尺寸。pg_archivecleanup工具被特别设计为在典型单一后备配置下与`archive_cleanup_command`共同使用，见pg_archivecleanup。不过要注意，如果你正在为备份目的使用归档，有一些文件即使后备服务器不再需要你也需要保留它们，它们被用来从至少最后一个基础备份恢复。

配置的一个简单例子是：

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass options=''-c
wal_sender_timeout=5000'''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

你可以有任意数量的后备服务器，但是如果你使用流复制，确保你在主服务器上将`max_wal_senders`设置得足够高，这样可以允许它们能同时连接。

流复制

流复制允许一台后备服务器比使用基于文件的日志传送更能保持为最新的状态。后备服务器连接到主服务器，主服务器则在 WAL 记录产生时即将它们以流式传送给后备服务器而不必等到 WAL 文件被填充。

默认情况下流复制是异步的，在这种情况下主服务器上提交一个事务与该变化在后备服务器上变得可见之间存在短暂的延迟。不过这种延迟比基于文件的日志传送方式中要小得多，在后备服务器的能力足以跟得上负载的前提下延迟通常低于一秒。在流复制中，不需要`archive_timeout`来缩减数据丢失窗口。

如果你使用的流复制没有基于文件的连续归档，该服务器可能在后备机收到 WAL 段之 前回收这些旧的 WAL 段。如果发生这种情况，后备机将需要重新从一个新的基础备份初始化。通过设置`wal_keep_size`为一个足够高的值来确保旧 的 WAL 段不会被太早重用或者为后备机配置一个复制槽，可以避免发生这种情况。如果设置了一个后备机可以访问的 WAL 归档，就不需要这些解决方案，因为该归档可以为后备机保留足够的段，后备机总是可以使用该归档来追赶主控机。

要使用流复制，建立一个基于文件的日志传送后备服务器。将一个基于文件日志传送后备服务器转变成流复制后备服务器的步骤是把`recovery.conf`文件中的设置以指向主服务器。设置主服务器上的listen_addresses和认证选项（见`pg_hba.conf`），这样后备服务器可以连接到主服务器上的伪数据库`replication`。

在支持 keepalive 套接字选项的系统上，设置tcp_keepalives_idle、tcp_keepalives_interval和tcp_keepalives_count有助于主服务器迅速地注意到一个断开的连接。

设置来自后备服务器的并发连接的最大数目（详见max_wal_senders）。

当后备服务器被启动并且`primary_conninfo`被正确设置，后备服务器将在重放完归档中所有可用的 WAL 文件之后连接到主服务器。 如果连接被成功建立，你将在后备服务器中看到一个 `walreceiver`，并且在主服务器中有一个相应的 `walsender` 进程。

认证

设置好用于复制的访问权限非常重要，这样只有受信的用户可以读取 WAL 流，因为很容易从 WAL 流中抽取出需要特权才能访问的信息。后备服务器必须作为一个具有`REPLICATION`特权的账户或一个超级用户来向主服务器认证。推荐为复制创建一个专用的具有`REPLICATION`和`LOGIN`特权的用户账户。虽然`REPLICATION`特权给出了非常高的权限，但它不允许用户修改主系统上的任何数据，而`SUPERUSER`特权则可以。

复制的客户端认证由一个在*`database`*域中指定`replication`的`pg_hba.conf`记录控制。例如，如果后备服务器运行在主机 IP `192.168.1.100`并且用于复制的账户名为`foo`，管理员可以在主服务器上向`pg_hba.conf`文件增加下列行：

```
# 允许来自 192.168.1.100 的用户 "foo" 在提供了正确的口令时作为一个
# 复制后备机连接到主控机。
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
```

```
host    replication     foo             192.168.1.100/32        md5
```

主服务器的主机名和端口号、连接用户名和口令在primary_conninfo中指定。在后备服务器上还可以在`~/.pgpass`文件中设置口令（在*database*域中指定`replication`）。例如，如果主服务器运行在主机 IP 192.168.1.50、端口`5432`上，并且口令为`foopass`，管理员可以在后备服务器的`postgresql.conf`文件中增加下列行：

```
# 后备机要连接到的主控机运行在主机 192.168.1.50 上，
# 端口号是 5432，连接所用的用户名是 "foo"，口令是 "foopass"。
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

监控

流复制的一个重要健康指标是在主服务器上产生但还没有在后备服务器上应用的 WAL 记录数。你可以通过比较主服务器上的当前 WAL 写位置和后备服务器接收到的最后一个 WAL 位置来计算这个滞后量。这些位置分别可以用主服务器上的`pg_current_wal_lsn`和后备服务器上的`pg_last_wal_receive_lsn`来检索。后备服务器的最后 WAL 接收位置也被显示在 WAL 接收者进程的进程状态中，即使用`ps`命令显示的状态。

你可以通过pg_stat_replication视图检索 WAL 发送者进程的列表。`pg_current_wal_lsn`与`sent_lsn`域之间的巨大差异表示主服务器承受着巨大的负载，而`sent_lsn`和后备服务器上`pg_last_wal_receive_lsn`之间的差异可能表示网络延迟或者后备服务器正承受着巨大的负载。

在一台热后备上，WAL接收者进程的状态可以通过pg_stat_wal_receiver`视图检索到。`pg_last_wal_replay_lsn`和该视图的`flushed_lsn`的差别表示WAL的接收速度大于它被重放的速度。

复制槽

复制槽提供了一种自动化的方法来确保主控机在所有的后备机收到 WAL 段之前不会移除它们，并且主控机也不会移除可能导致恢复冲突的行，即使后备机断开也是如此。

作为复制槽的替代，也可以使用wal_keep_size阻止移除旧的 WAL 段，或者使用archive_command把段保存在一个归档中。 不过，这些方法常常会导致保留的 WAL 段比需要的更多，而复制槽只保留已知所需要的段。另一方面，复制槽可以保留很多的WAL段以至于它们填满了分配给`pg_wal`的空间；max_slot_wal_keep_size限制复制槽所保留的WAL文件的大小。

类似地，hot_standby_feedback和 vacuum_defer_cleanup_age保护了相关行不被 vacuum 移除，但是前者在后备机断开期间无法提供保护，而后者则需要被设置为一个很高的值已提供足够的保护。复制槽克服了这些缺点。

查询和操纵复制槽

每个复制槽都有一个名字，名字可以包含小写字母、数字和下划线字符。

已有的复制槽和它们的状态可以在 pg_replication_slots视图中看到。

槽可以通过流复制协议 或者 SQL 函数创建并且移除。

配置实例

你可以这样创建一个复制槽：

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
```

```
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

要配置后备机使用这个槽，在后备机中应该配置`primary_slot_name`。这里是一个简单的例子：

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

级联复制

级联复制特性允许一台后备服务器接收复制连接并且把 WAL 记录流式传送给其他后备服务器，就像一个转发器一样。这可以被用来减小对于主控机的直接连接数并且使得站点间的带宽开销最小化。

一台同时扮演着接收者和发送者角色的后备服务器被称为一台级联后备服务器。“更直接”（通过更少的级联后备服务器）连接到主控机的后备服务器被称为上游服务器，而那些离得更远的后备服务器被称为下游服务器。级联复制并没有对下游服务器的数量或布置设定限制。

一台级联后备服务器不仅仅发送从主控机接收到的 WAL 记录，还要发送那些从归档中恢复的记录。因此即使某些上游连接中的复制连接被中断，只要还有新的 WAL 记录可用，下游的流复制都会继续下去。

级联复制目前是异步的。同步复制设置当前对级联复制无影响。

不管在什么样的级联布置中，热备反馈都会向上游传播。

如果一台上游后备服务器被提升为新的主控机，且下游服务器的`recovery_target_timeline`被设置成’latest'`(默认)，下游服务器将继续从新的主控机得到流。

要使用级联复制，需配置级联后备服务器以允许接收复制连接（即设置max_wal_senders和hot_standby，并且配置基于主机的认证）。你还将需要设置下游后备服务器中的`primary_conninfo`指向级联后备服务器。

同步复制

PostgreSQL流复制默认是异步的。如果主服务器崩溃，则某些已被提交的事务可能还没有被复制到后备服务器，这会导致数据丢失。数据的丢失量与故障转移时的复制延迟成比例。

同步复制能够保证一个事务的所有修改都能被传送到一台或者多台同步后备服务器。这扩大了由一次事务提交所提供的标准持久化级别。在计算机科学理论中这种保护级别被称为 2-safe 复制。而当`synchronous_commit`被设置为`remote_write`时，则是 group-1-safe （group-safe 和 1-safe）。

在请求同步复制时，一个写事务的每次提交将一直等待，直到收到一个确认表明该提交在主服务器和后备服务器上都已经被写入到磁盘上的预写式日志中。数据会被丢失的唯一可能性是主服务器和后备服务器在同一时间都崩溃。这可以提供更高级别的持久性，尽管只有系统管理员要关系两台服务器的放置和管理。等待确认提高了用户对于修改不会丢失的信心，但是同时也不必要地增加了对请求事务的响应时间。最小等待时间是在主服务器和后备服务器之间的来回时间。

只读事务和事务回滚不需要等待后备服务器的回复。子事务提交也不需要等待后备服务器的响应，只有顶层提交才需要等待。长时间运行的动作（如数据载入或索引构建）不会等待最后的提交消息。所有两阶段提交动作要求提交等待，包括预备和提交。

同步后备可以是物理复制后备或者是逻辑复制订阅者。它还可以是任何其他物理或者逻辑WAL复制流的消费者，它懂得如何发送恰当的反馈消息。除内建的物理和逻辑复制系统之外，还包括`pg_receivewal`和`pg_recvlogical`之类的特殊程序，以及一些第三方复制系统和定制程序。同步复制支持的细节请查看相应的文档。

基本配置

一旦流复制已经被配置，配置同步复制就只需要一个额外的配置步骤：synchronous_standby_names必须被设置为一个非空值。**synchronous_commit`也必须被设置为`on**，但由于这是默认值，通常不需要改变。这样的配置将导致每一次提交都等待确认消息，以保证后备服务器已经将提交记录写入到持久化存储中。`synchronous_commit`可以由个体用户设置，因此它可以在配置文件中配置、可以为特定用户或数据库配置或者由应用动态配置，这样可以在一种每事务基础上控制持久性保证。

在一个提交记录已经在主服务器上被写入到磁盘后，WAL 记录接着被发送到后备服务器。每次一批新的 WAL 数据被写入到磁盘后，后备服务器会发送回复消息，除非在后备服务器上`wal_receiver_status_interval`被设置为零。如果`synchronous_commit`被设置为`remote_apply`，当提交记录被重放时后备服务器会发送回应消息，这会让该事务变得可见。如果根据主服务器的`synchronous_standby_names`设置选中该后备服务器作为一个同步后备，将会根据来自该后备服务器和其他同步后备的回应消息来决定何时释放正在等待确认提交记录被收到的事务。这些参数允许管理员指定哪些后备服务器应该是同步后备。注意同步复制的配置主要在主控机上。命名的后备服务器必须直接连接到主控机，主控机对使用级联复制的下游后备服务器一无所知。

将`synchronous_commit`设置为`remote_write`将导致每次提交都等待后备服务器已经接收提交记录并将它写出到其自身所在的操作系统的确认，但并非等待数据都被刷出到后备服务器上的磁盘。这种设置提供了比`on`要弱一点的持久性保障：在一次操作系统崩溃事件中后备服务器可能丢失数据，尽管它不是一次Ivory SQL崩溃。不过，在实际中它是一种有用的设置，因为它可以减少事务的响应时间。只有当主服务器和后备服务器都崩溃并且主服务器的数据库同时被损坏的情况下，数据丢失才会发生。

把`synchronous_commit`设置为`remote_apply`将导致每一次提交都会等待，直到当前的同步后备服务器报告说它们已经重放了该事务，这样就会使该事务对用户查询可见。在简单的情况下，这为带有因果一致性的负载均衡留出了余地。

如果请求一次快速关闭，用户将停止等待。不过，在使用异步复制时，在所有未解决的 WAL 记录被传输到当前连接的后备服务器之前，服务器将不会完全关闭。

多个同步后备

同步复制支持一个或者更多个同步后备服务器，事务将会等待，直到所有同步后备服务器都确认收到了它们的数据为止。事务必须等待其回复的同步后备的数量由`synchronous_standby_names`指定。这个参数还指定一个后备服务器名称及方法（**FIRST`和`ANY**）的列表来从列出的后备中选取同步后备。

方法`FIRST`指定一种基于优先的同步复制并且让事务提交等待，直到它们的WAL记录被复制到基于优先级选中的所要求数量的同步后备上为止。在列表中出现较早的后备被给予较高的优先级，并且将被考虑为同步后备。其他在这个列表中位置靠后的后备服务器表示可能的同步后备。如果任何当前的同步后备由于任何原因断开连接，它将立刻被下一个最高优先级的后备所替代。

基于优先的多同步后备的`synchronous_standby_names`示例为：

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

在这个例子中，如果有四个后备服务器`s1`、**s2**、`s3`和`s4`在运行，两个后备服务器`s1`和`s2`将被选中为同步后备，因为它们出现在后备服务器名称列表的前部。`s3`是一个潜在的同步后备，当`s1`或`s2`中的任何一个失效， 它就会取而代之。`s4`则是一个异步后备因为它的名字不在列表中。

方法`ANY`指定一种基于规定数量的同步复制并且让事务提交等待，直到它们的WAL记录至少被复制到列表中所要求数量的同步后备上为止。

`synchronous_standby_names`的基于规定数量的多同步后备的例子：

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

在这个例子中，如果有四台后备服务器`s1`、s2、`s3`以及`s4`正在运行，事务提交将会等待来自至少其中任意两台后备服务器的回复。`s4`是一台异步后备，因为它的名字不在该列表中。

后备服务器的同步状态可以使用`pg_stat_replication`视图查看。

性能规划

同步复制通常要求仔细地规划和放置后备服务器来保证应用能令人满意地工作。等待并不利用系统资源，但是事务锁会持续保持直到传输被确认。其结果是，不小心使用同步复制将由于响应时间增加以及较高的争用率而降低数据库应用的性能。

IvorySQL允许应用开发者通过复制来指定所要求的持久性级别。这可以为整个系统指定，不过它也能够为特定的用户或连接指定，甚至还可以为单个事务指定。

例如，一个应用的载荷的组成可能是这样：10% 的改变是重要的客户详情，而 90% 的改变是不太重要的数据，即使它们丢失业务也比较容易容忍（例如用户间的聊天消息）。

通过在应用级别（在主服务器上）指定的同步复制选项，我们可以为大部分重要的改变提供同步复制，并且不会拖慢整体的载荷。应用级别选项是使高性能应用享受同步复制的一种重要和实用的工具。

你应该认为网络带宽必须比 WAL 数据的产生率高。

高可用性规划

当`synchronous_commit`被设置为`on`、`remote_apply`或者`remote_write`时，`synchronous_standby_names`指定产生的事务提交要等待其回应的同步后备的数量和名称。如果任一同步后备崩溃，这类事务提交可能无法完成。

高可用的最佳方案是确保有所要求数量的同步后备。这可以通过使用`synchronous_standby_names`指定多个潜在后备服务器来实现。

在基于优先的同步复制中，出现在该列表前部的后备服务器将被用作同步后备。后面的后备服务器将在当前同步后备服务器失效时取而代之。

在基于规定数量的同步复制中，所有出现在该列表中的后备服务器都将被用作同步后备的候选。即使其中的一个失效，其他后备仍将继续担任候选同步后备的角色。

当一台后备服务器第一次附加到主服务器时，它将处于一种还没有正确同步的状态。这被描述为`追赶`模式。一旦后备服务器和主服务器之间的迟滞第一次变成零，我们就来到了实时的`流式`状态。在后备服务器被创建之后的很长一段时间内可能都是追赶模式。如果后备服务器被关闭，则追赶周期将被增加，增加量由后备服务器被关闭的时间长度决定。只有当后备服务器到达`流式`状态后，它才能成为一台同步后备。这种状态可以使用`pg_stat_replication`视图查看。

如果在提交正在等待确认时主服务器重启，那些正在等待的事务将在主数据库恢复时被标记为完全提交。没有办法确认所有后备服务器已经收到了在主服务器崩溃时所有还未处理的 WAL 数据。某些事务可能不会在后备服务器上显示为已提交，即使它们在主服务器上显示为已提交。我们提供的保证是：在 WAL 数据已经被所有后备服务器安全地收到之前，应用将不会收到一个事务成功提交的显式确认。

如果实在无法保持所要求数量的同步后备，那么应该减少`synchronous_standby_names`中指定的事务提交应该等待其回应的同步后备的数量（或者禁用），并且在主服务器上重载配置文件。

如果主服务器与剩下的后备服务器是隔离的，你应当故障转移到那些其他剩余后备服务器中的最佳候选者上。

如果在事务等待时你需要重建一台后备服务器，确保命令 pg_start_backup() 和 pg_stop_backup() 被运行在一个`synchronous_commit` = `off`的会话中，否则那些请求将永远等待后备服务器出现。

在后备机上连续归档

当在一个后备机上使用连续归档时，有两种不同的情景：WAL 归档在主服务器和后备机之间共享，或者后备机有自己的 WAL 归档。当后备机拥有其自身的 WAL 归档时，将`archive_mode`设置为 `always`，后备机将在收到每个 WAL 段时调用归档命令，不管它是从归档恢复还是使用流复制恢复。共享归档可以类似地处理，但是`archive_command`必须测试要被归档的文件是否 已经存在，以及现有的文件是否有相同的内容。这要求`archive_command`中有更多处理，因为它必须当心不要覆盖具有不同内容的已有文件，但是如果完全相同的文件被归档两次时应返回成功。并且如果两个服务器尝试同时归档同一个文件，所有这些都必须在没有竞争情况的前提下完成。

如果`archive_mode`被设置为`on`，归档器在恢复或者后备模式中无法启用。如果后备服务器被提升，它将在被提升后开始归档，但是它将不会归档任何不是它自身产生的WAL或时间线历史文件。 要在归档中得到完整的一系列 WAL 文件，你必须确保所有 WAL 在到达后备机之前都被归档。对于基于文件的日志传输来说天然就是这样，因为后备机只能恢复在归档中找到的文件，而启用了流复制时则不是这样。 当一台服务器不在恢复模式中时，在`on`和`always`模式之间没有差别。

故障转移

如果主服务器失效，则后备服务器应该开始故障转移过程。

如果后备服务器失效，则不会有故障转移发生。如果后备服务器可以被重启（即使晚一点），由于可重启恢复的优势，那么恢复处理也能被立即重启。如果后备服务器不能被重启，则一个全新的后备服务器实例应该被创建。

如果主服务器失效并且后备服务器成为了新的主服务器，那么接下来旧的主服务器重启后，你必须有一种机制来通知旧的主服务器不再成为主服务器。有些时候这被称为STONITH（Shoot The Other Node In The Head，关闭其他节点），这对于避免出现两个系统都认为它们是主服务器的情况非常必要，那种情况将导致混乱并且最终导致数据丢失。

很多故障转移系统仅使用两个系统，主系统和后备系统，它们由某种心跳机制连接来持续验证两者之间的连接性和主系统的可用性。也可能会使用第三个系统（称为目击者服务器）来防止某些不当故障转移的情况，但是除非非常小心地建立它并且经过了严格地测试，额外的复杂度可能会使该工作得不偿失。

IvorySQL并不提供在主服务器上标识失败并且通知后备数据库服务器所需的系统软件。现在已有很多这样的工具并且很好地与成功的故障转移所需的操作系统功能整合在一起，例如 IP 地址迁移。

一旦发生到后备服务器的故障转移，就只有单一的一台服务器在操作。这被称为一种退化状态。之前的后备服务器现在是主服务器，但之前的主服务器处于关闭并且可能一直保持关闭。要回到正常的操作，一个后备服务器必须被重建，要么在之前的主系统起来时使用它重建，要么使用第三台（可能是全新的）服务器来重建。在大型集簇上，pg_rewind功能可以被用来加速这个过程。一旦完成，主服务器和后备服务器可以被认为是互换了角色。某些人选择使用第三台服务器来为新的主服务器提供备份，直到新的后备服务器被重建，不过显然这会使得系统配置和操作处理更复杂。

因此，从主服务器切换到后备服务器可以很快，但是要求一些时间来重新准备故障转移集群。从主服务器到后备服务器的常规切换是有用的，因为它允许每个系统有常规的关闭时间来进行维护。这也可以作为一种对故障转移机制的测试，以保证在你需要它时它真地能够工作。我们推荐写一些管理过程来做这些事情。

要触发一台日志传送后备服务器的故障转移，运行`pg_ctl promote`，调用 `pg_promote()`，或者创建一个触发器文件，其文件名和路径由`promote_trigger_file`设置指定。如果你正在规划使用`pg_ctl promote`或调用`pg_promote()`以进行故障转移，`promote_trigger_file`就不是必要的。如果你正在建立只用于从主服务器分流只读查询而不是高可用性目的的报告服务器，你不需要提升它。

热备

术语热备用来描述处于归档恢复或后备模式中的服务器连接到服务器并运行只读查询的能力。这有助于复制目的以及以高精度恢复一个备份到一个期望的状态。术语热备也指服务器从恢复转移到正常操作而用户能继续运行查询并且保持其连接打开的能力。

在热备模式中运行查询与正常查询操作相似，尽管如下所述存在一些用法和管理上的区别。

当hot_standby参数在一台后备服务器上被设置为真时，一旦恢复将系统带到一个一致的状态它将开始接受连接。所有这些连接都被限制为只读，甚至临时表都不能被写入。

后备服务器上的数据需要一些时间从主服务器到达后备服务器，因此在主服务器和后备服务器之间将有一段可以度量的延迟。近乎同时在主服务器和后备服务器上运行相同的查询可能因此而返回不同的结果。我们说后备服务器上的数据与主服务器是*最终一致*的。一旦一个事务的提交记录在后备服务器上被重播，那个事务所作的修改将对后备服务器上所有新取得的快照可见。快照可以在每个查询或每个事务的开始时取得，这取决于当前的事务隔离级别。

在热备期间开始的事务可能发出下列命令：

- 查询访问: `SELECT`、`COPY TO`
- 游标命令: `DECLARE`、`FETCH`、`CLOSE`
- 设置: `SHOW`、`SET`、`RESET`
- 事务管理命令:
  - `BEGIN`、`END`、`ABORT`、`START TRANSACTION`
  - `SAVEPOINT`、`RELEASE`、`ROLLBACK TO SAVEPOINT`
  - `EXCEPTION`块或其他内部子事务
- `LOCK TABLE`，不过只在下列模式之一中明确发出： `ACCESS SHARE`、`ROW SHARE` 或 `ROW EXCLUSIVE`.
- 计划和资源: `PREPARE`、`EXECUTE`、 `DEALLOCATE`、`DISCARD`
- 插件和扩展: `LOAD`
- `UNLISTEN`

在热备期间开始的事务将不会被分配一个事务 ID
并且不能被写入到系统的预写式日志。因此，下列动作将产生错误消息：

- 数据操纵语言（DML）: `INSERT`、 `UPDATE`、`DELETE`、`COPY FROM`、`TRUNCATE`。注意不允许在恢复期间导致一个触发器被执行的动作。这个限制甚至被应用到临时表，因为不分配事务 ID 表行就不能被读或写，而当前不可能在一个热备环境中分配事务 ID。
- 数据定义语言（DDL）: `CREATE`、 `DROP`、`ALTER`、`COMMENT`。这个限制甚至被应用到临时表，因为执行这些操作会要求更新系统目录表。
- `SELECT … FOR SHARE | UPDATE`，因为不更新底层数据文件就无法取得行锁。
- `SELECT`语句上的能产生 DML 命令的规则。
- 显式请求一个高于`ROW EXCLUSIVE MODE`的模式的`LOCK`。
- 默认短形式的`LOCK`，因为它请求`ACCESS EXCLUSIVE MODE`。
- 显式设置非只读状态的事务管理命令：
  - `BEGIN READ WRITE`, `START TRANSACTION READ WRITE`
  - `SET TRANSACTION READ WRITE`, `SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE`
  - `SET transaction_read_only = off`
- 两阶段提交命令: `PREPARE TRANSACTION`、 `COMMIT PREPARED`、`ROLLBACK PREPARED`，因为即使只读事务也需要在准备阶段（两阶段提交中的第一个阶段）写 WAL。
- 序列更新 : `nextval()`、`setval()`
- `LISTEN`,`NOTIFY`

在正常操作中，“只读”事务被允许使用`LISTEN`和`NOTIFY`，因此热备会话在比普通只读会话更紧一点的限制下操作。这些限制中的某些可能会在一个未来的发行中被放松。

在热备期间，参数`transaction_read_only`总是为真并且不可以被改变。但是只要不尝试修改数据库，热备期间的连接工作起来更像其他数据库连接。如果发生故障转移或切换，该数据库将切换到正常处理模式。当服务器改变模式时会话将保持连接。一旦热备结束，它将可以发起读写事务（即使是一个在热备期间启动的

会话）。

用户可以通过`SHOW in_hot_standby`来检查hot standby会话是否是活跃的 (在服务器版本 14 之前该参数`in_hot_standby`不存在。对于更早版本的服务器，可行的替代方法是 `SHOW transaction_read_only`。) 此外， 还有一些函数允许用户访问有关后备服务器的信息。它们允许您编写程序来识别数据库当前的状态。用于监控恢复进度，或者您可以编写复杂的程序将数据库恢复到特定状态。

处理查询冲突

主服务器和后备服务器在多方面都松散地连接在一起。主服务器上的动作将在后备服务器上产生效果。结果是在它们之间有潜在的负作用或冲突。最容易理解的冲突是性能：如果在主服务器上发生一次大数据量的载入，那么着将在后备服务器上产生一个相似的 WAL 记录流，因而后备服务器查询可能要竞争系统资源（例如 I/O）。

随着热备发生的还可能有其他类型的冲突。对于可能需要被取消的查询和（某些情况中）解决它们的已断开会话来说，这些冲突是硬冲突。用户可以用几种方式来处理这种冲突。冲突情况包括：

- 在主服务器上取得了访问排他锁（包括显式`LOCK`命令和多种DDL动作）与后备查询中的表访问冲突。
- 在主服务器上删除一个表空间与使用该表空间存储临时工作文件的后备查询冲突。
- 在主服务器上删除一个数据库与在后备服务器上连接到该数据库的会话冲突。
- 从 WAL 清除记录的应用与快照仍能“看见”任意要被移除的行的后备事务冲突。
- 从 WAL 清除记录的应用与在后备服务器上访问该目标页的查询冲突，不管要被移除的数据是否为可见。

在主服务器上，这些情况仅仅会导致等待；并且用户可以选择取消这些冲突动作中间的一个。但是，在后备服务器上则没有选择：已被 WAL 记录的动作已经在主服务器上发生，那么后备服务器不能在应用它时失败。此外，允许 WAL 应用无限等待是非常不可取的，因为后备服务器的状态将变得逐渐远远落后于主服务器的状态。因此，提供了一种机制来强制性地取消与要被应用的 WAL 记录冲突的后备查询。

该问题情形的一个例子是主服务器上的一位管理员在一个表上运行`DROP TABLE`，而该表正在后备服务器上被查询。如果`DROP TABLE`被应用在后备服务器上，很明显该后备查询不能继续。如果这种情况在主服务器上发生，`DROP TABLE`将等待直到其他查询结束。但是当`DROP TABLE`被运行在主服务器上，主服务器没有关于运行在后备服务器上查询的信息，因此它将不会等待任何这样的后备查询。WAL 改变记录在后备查询还在运行时来到后备服务器上，导致一个冲突。后备服务器必须要么延迟 WAL 记录的应用（还有它们之后的任何事情），要么取消冲突查询这样`DROP TABLE`可以被应用。

当一个冲突查询很短时，我们通常期望能延迟 WAL 应用一小会儿让它完成；但是在 WAL 应用中的一段长的延迟通常是不受欢迎的。因此取消机制有参数，max_standby_archive_delay和max_standby_streaming_delay，它们定义了在 WAL 应用中的最大允许延迟。当应用任何新收到的 WAL 数据花费了超过相关延迟设置值时，冲突查询将被取消。设立两个参数是为了对从一个归档读取 WAL 数据（即来自一个基础备份的初始恢复或者“追赶”一个已经落后很远的后备服务器）和通过流复制读取 WAL数据的两种情况指定不同的延迟值。

在一台后备服务器上这主要是为了该可用性而存在，最好把延迟参数设置得比较短，这样服务器不会由于后备查询导致的延迟落后主服务器太远。但是，如果该后备服务器是位了执行长时间运行的查询，则一个较高甚至无限的延迟值更好。但是记住一个长时间运行的查询延迟了 WAL 记录的应用，它可能导致后备服务器上的其他会话无法看到主服务器上最近的改变。

一旦`max_standby_archive_delay`或`max_standby_streaming_delay`指定的延迟被超越，冲突查询将被取消。这通常仅导致一个取消错误，尽管在重放一个`DROP DATABASE`的情况下整个冲突会话都将被中断。另外，如果冲突发生在一个被空闲事务持有的锁上，该冲突会话会被中断（这种行为可能在未来被改变）。

被取消的查询可能会立即被重试（当然是在开始一个新的事务后）。因为查询取消依赖于 WAL 记录被重放的本质，如果一个被取消的查询被再次执行，它可能会很好地成功完成。

记住延迟参数是从 WAL 数据被后备服务器收到后流逝的时间。因此，留给后备服务器上任何一个查询的宽限期从不会超过延迟参数

，并且如果后备服务器已经由于等待之前的查询完成而落后或者因为过重的更新负载而无法跟上主服务器，宽限期可能会更少。

在后备查询和 WAL
重播之间发生冲突的最常见原因是“过早清除”。正常地，PostgreSQL允许在没有事务需要看到旧行版本时对它们进行清除，这样可以保证根据 MVCC
规则的正确的数据可见性。不过，这个规则只能被应用于执行在主控机上的事务。因此有可能主控机上的清除会移除对一个后备服务器事务还可见的行版本。

有经验的用户应当注意行版本清除和行版本冻结都可能与后备查询冲突。即便在一个没有被更新或被删除行的表上运行一次手工`VACUUM FREEZE`也可能导致冲突。

用户应当清楚，主服务器上被正常和重度更新的表将快速地导致后备服务器上长时间运行的查询被取消。在这样的情况下，`max_standby_archive_delay`或`max_standby_streaming_delay`的有限制设置可以被视作`statement_timeout`设置。

如果发现后备查询取消的数量不可接受，还是有补救的可能。第一种选项是设置参数
`hot_standby_feedback`，它阻止`VACUUM`
移除最近死亡的元组并且因此清除冲突不会产生。如果你这样做，你应当
注意这将使主服务器上的死亡元组清除被延迟，这可能会导致不希望发生
的表膨胀。不过，清除的情况不会比在主服务器上直接运行后备查询时更糟，
并且你仍然能够享受将执行分流到后备服务器的好处。如果后备服务器频繁地连接和
断开，你可能想要做些调整来处理无法提供`hot_standby_feedback`
反馈的时期。例如，考虑增加`max_standby_archive_delay`，这样 在断开连接的期间查询就不会快速地被 WAL 归档文件中的冲突取消。你也应该考虑
增加`max_standby_streaming_delay`来避免重新连接后新到达的流 WAL 项导致的快速取消。

另一个选项是增加主服务器上的vacuum_defer_cleanup_age，这样死亡行不会像平常那么快地被清理。这将允许在后备服务器上的查询能在被取消前有更多时间执行，并且不需要设置一个很高的`max_standby_streaming_delay`。但是，这种方法很难保证任何指定的执行时间窗口，因为`vacuum_defer_cleanup_age`是用主服务器上被执行的事务数来衡量的。

查询取消的数量和原因可以使用后备服务器上的`pg_stat_database_conflicts`系统视图查看。`pg_stat_database`系统视图也包含汇总信息。

当 WAL 重放由于冲突而需要比`deadlock_timeout`更长时间时，用户可以控制是否打印日志消息。由参数log_recovery_conflict_waits控制。

管理员概览

如果`hot_standby`在`postgresql.conf`中被设置为`on`并且存在一个standby.signal文件，服务器将运行在热备模式。但是，可能需要一些时间来允许热备连接，因为在服务器完成足够的恢复来为查询提供一个一致的状态之前，它将不会接受连接。在此期间，尝试连接的客户端将被一个错误消息拒绝。要确认服务器已经可连接，要么循环地从应用尝试连接，要么在服务器日志中查找这些消息：

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

在主服务器上，一旦创建一个检查点，一致性信息就被记录下来。当读取在特定时段（当在主服务器上`wal_level`没有被设置为`replica`或者`logical`的期间）产生的 WAL
时无法启用热备。在同时存在这些条件时，到达一个一致状态也会被延迟：

- 一个写事务有超过 64 个子事务
- 生存时间非常长的写事务

如果你正在运行基于文件的日志传送（“温备”），你可能需要等到下一个 WAL
文件到达，这可能和主服务器上的`archive_timeout`设置一样长。

设置几个参数可确定用于跟踪事务ID、锁和预备事务的共享内存大小。后备服务器上的设置必须大于或等于主服务器上的设置，以确保在恢复过程中不会耗尽共享内存。例如，如果主数据库正在执行预备事务，而备用数据库没有获取共享内存来跟踪预备事务，则备用数据库将无法继续恢复，直到配置更改。受影响的参数是：

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_wal_senders`
- `max_worker_processes`

确保这不是问题的可靠方法是使备用数据库上的这些参数的值等于或大于主数据库上的值。因此，如果您想增加这些值，您应该先更改后备服务器上的设置，然后再更改主服务器上的设置。相反，如果要减小这些值，则应先更改主服务器上的设置，然后再更改后备服务器上的设置。请记住，当一个备用数据库被提升时，它会成为后续备用数据库所需参数设置的新基准。因此，最好在所有后备服务器上保持这些设置相同，这样在切换/故障转移期间就不会出现问题。

WAL 跟踪主节点上这些参数的变化。如果热备处理一个
WAL，表明主节点当前值大于备用数据库上的值，它将记录一个警告并中止恢复。例如:

```
WARNING：由于参数设置不足，无法进行热备
详细信息：max_connections = 80 的设置低于主服务器上的设置，其值为 100。
LOG：恢复已暂停
详细信息：如果恢复未暂停，服务器将关闭。
提示：您可以在进行必要的配置更改后重新启动服务器。
```

此时，您应该更改备库设置并重新启动实例以继续恢复。如果备库不是热备，不兼容的参数更改将立即将其关闭而不会暂停。因为这样继续开机没有意义。

管理员为max_standby_archive_delay和max_standby_streaming_delay选择适当的设置很重要。最好的选择取决于业务的优先级。例如如果服务器主要的任务是作为高可用服务器，那么你将想要低延迟设置，甚至是零（尽管它是一个非常激进的设置）。如果后备服务器的任务是作为一个用于决策支持查询的额外服务器，那么将其最大延迟值设置为很多小时甚至 -1 （表示无限等待）可能都是可以接受的。

在主服务器上写出的事务状态 "hint bits" 是不被 WAL
记录的，因此后备服务器上的数据将可能重新写出该提示。这样，即使所有用户都是只读的，后备服务器仍将执行磁盘写操作；但数据值本身并没有发生改变。用户将仍写出大的排序临时文件并且重新生成 relcache 信息文件，这样在热备模式中数据库没有哪个部分是真正只读的。还要注意使用dblink模块写到远程数据库以及其他使用 PL 函数位于数据库之外的操作仍将可用，即使该事务是本地只读的。

在恢复模式期间，下列类型的管理命令是不被接受的：

- 数据定义语言（DDL）: e.g., `CREATE INDEX`
- 特权和所有权: `GRANT`, `REVOKE`,
- 维护命令: `ANALYZE`, `VACUUM`,`CLUSTER`, `REINDEX`

注意这些命令中的某些实际上在主服务器上的“只读”模式事务期间是被允许的。

结果是，你无法创建只存在于后备服务器上的额外索引以及统计信息。如果需要这些管理命令，它们应该在主服务器上被执行，并且最后那些改变将被传播到后备服务器。

`pg_cancel_backend()`和`pg_terminate_backend()`将在用户后端上工作，而不是执行恢复的 Startup 进程。`pg_stat_activity`不会为 Startup

进程显示一个项，也不会把恢复事务显示为活动。结果是在恢复期间`pg_prepared_xacts`总是为空。如果你希望解决不能确定的预备事务，查看主服务器上的`pg_prepared_xacts`并且发出命令来解决那里的事务或者在恢复结束后来解决它们。

和平常一样，**`pg_locks`将显示被后端持有的锁。`pg_locks`也会显示一个由 Startup 进程管理的虚拟事务，它拥有被恢复重播的事务所持有的所有`AccessExclusiveLocks`**。注意 Startup 进程不请求锁来做数据库更改，并且因此对于 Startup 进程除`AccessExclusiveLocks`之外的锁不显示在`pg_locks`中，它们仅被假定存在。

Nagios的插件check_pgsql将可以工作，因为它检查的简单信息是存在的。check_postgres监控脚本也将能工作，尽管某些被报告的值可能给出不同或者混乱的结果。例如，上一次清理时间将不会被维护，因为在后备服务器上不会发生清理。在主服务器上运行的清理仍会把它们的改变发送给后备服务器。

WAL 文件控制命令在恢复期间将不会工作，如`pg_start_backup`、`pg_switch_wal`等。

可动态载入的模块可以工作，包括`pg_stat_statements`。

咨询锁在恢复期间工作正常，包括死锁检测。注意咨询锁从来都不会被 WAL 记录，因此在主服务器或后备服务器上一个咨询锁不可能会与 WAL 重播发生冲突。也不可能会在主服务器上获得一个咨询锁并且在后备服务器上开始一个相似的咨询锁。咨询锁只与它们被取得的那个服务器相关。

基于触发器的复制系统（如Slony、Londiste和Bucardo）将根本不会运行在后备服务器上，然而只要改变不被发送到要被应用的后备服务器，它们将在主服务器上运行得很好。WAL 重播不是基于触发器的，因此你不能用后备服务器接替任何需要额外数据库写操作或依赖触发器使用的系统。

新的 OID 不能被分配，然而某些UUID生成器仍然能工作，只要它们不依赖于向数据库写新的状态。

当前，在只读事务期间不允许创建临时表，因此在某些情况中现有的脚本将不会正确运行。这个限制可能会在稍后的发行中被放松。这既是一个 SQL 标准符合问题也是一个技术问题。

只有在表空间为空时`DROP TABLESPACE`才能成功。某些后备服务器用户可能正在通过他们的`temp_tablespaces`参数使用该表空间。如果在该表空间中有临时文件，所有活动查询将被取消来保证临时文件被移除，这样该表空间可以被移除并且 WAL 重播可以继续。

在主服务器上运行`DROP DATABASE`或`ALTER DATABASE … SET TABLESPACE`将产生一个 WAL 项，它将导致所有连接到后备服务器上那个数据库的用户被强制地断开连接。这个动作会立即发生，不管`max_standby_streaming_delay`的设置是什么。注意`ALTER DATABASE … RENAME`不会断开用户，这在大部分情况中不会有提示，然而如果它依赖某种基于数据库名的方法，在某些情况中会导致程序混乱。

在普通（非恢复）模式中，如果你为具有登录能力的角色发出`DROP USER`或`DROP ROLE`，而该用户仍然连接着，则对已连接用户不会发生任何事情 - 他们保持连接。但是用户不能重新连接。这种行为也适用于恢复，因此在主服务器上的一次`DROP USER`不会使后备服务器上的用户断开。

在恢复期间统计收集器是活动的。所有扫描、读、阻塞、索引使用等将在后备服务器上被正常的记录。被重播的动作将不会重复它们在主服务器上的效果，因此重播一个插入将不会导致pg_stat_user_tables的 Inserts 列上的递增。在恢复的开始 stats 文件会被删除，因此来自主服务器和后备服务器的 stats 将不同；这被认为是一种特性而不是缺陷。

在恢复期间自动清理不是活动的。它将在恢复末尾正常启动。

检查点进程和后台写入进程在恢复期间是活动状态的。检查点进程将执行重启动点（与主服务器上的检查点相似），后台写入进程将执行正常的块清理活动。这可以包括存储在后备服务器上的提示位信息的更新。在恢复期间，`CHECKPOINT`命令会被接受，然而它会执行一个重启点而不是一个新的检查点。

热备参数参考

在主服务器上，可以使用参数wal_level和vacuum_defer_cleanup_age。在主服务器上设置max_standby_archive_delay和max_standby_streaming_delay不会产生效果。

在主服务器上，可以使用参数hot_standby、max_standby_archive_delay和max_standby_streaming_delay。只要服务器保持在后备模式vacuum_defer_cleanup_age就没有效果，然而当后备服务器变成主服务器时它将变得相关。

警告

热备有一些限制。这些限制很可能在未来的发行中被修复：

- 在能够取得快照之前，需要正在运行的事务的完整知识。使用大量子事务（目前指超过 64 个）的事务将延迟只读连接的启动，直到最长的运行着的写事务完成。如果发生这种情况，说明消息将被发送到服务器日志。
- 主服务器上的每一个检查点将产生用于后备查询的可用启动点。如果后备服务器在主控机处于关闭状态时被关闭，就没有办法在主服务器启动之前重新进入热后备，因此它在 WAL 日志中产生一个进一步启动点。这种情况在它可能发生的大部分常见情况中不是一个问题。通常，如果主服务器被关闭并且不再可用，这可能是由于某种严重错误要求后备服务器被转变成为一个新的主服务器来操作。并且在主服务器被故意关闭的情况下，协调保证后备服务器平滑地过渡为新的主服务器也是一种标准过程。
- 在恢复尾声，由预备事务持有的`AccessExclusiveLocks`将要求两倍的正常锁表项。如果你计划运行大量并发的通常要求`AccessExclusiveLocks`的预备事务，或者你计划运行一个需要很多`AccessExclusiveLocks`的大型事务，我们建议你为`max_locks_per_transaction`选择一个更大的值，也许是主服务器上该参数值的两倍。如果你的`max_prepared_transactions`设置为 0，你根本不需要考虑这个问题。
- 可序列化事务隔离级别目前在热备中不可用。尝试在热备模式中将一个事务设置为可序列化隔离级别将产生一个错误。

.3. 运维管理

由于IvorySQL基于PostgreSQL开发而成，运维人员在阅读理解本节内容时，建议同时参阅 手册。

升级IvorySQL版本

升级方案概述

IvorySQL版本号由主要版本和次要版本组成。例如，IvorySQL 3.2 中的3是主要版本，2是次要版本。

发布次要版本是不会改变内存的存储格式，因此总是和相同的主要版本兼容。例如，IvorySQL 3.2 和Ivory SQL 3.0 以及后续的 Ivory SQL 3.x 兼容。对于这些兼容版本的升级非常简单，只要关闭数据库服务，安装替换二进制的可执行文件，重新启动服务即可。

接下来，我们主要讨论IvorySQL的跨版本升级问题，例如，从 IvorySQL 2.3 升级到 IvorySQL 3.4。主要版本的升级可能会修改内部数据的存储格式，因此需要执行额外的操作。常用的跨版本升级方法和适用场景如下：

升级方法	适用场景	停机时间
通过pg_dumpall升级数据	中小型数据库，例如小于100GB支持跨平台数据迁移	取决于数据库的大小
pg_upgrade工具	大中型数据库，例如大于100GB本机就地升级	几分钟
逻辑复制	大中型数据库，例如大于100GB跨平台支持	几秒钟

注意事项：
新的主版本通常会引入一些用户可见的不兼容性，因此可能需要应用程序编程上的改变。所有用户可见的更

改都被列在
说明中，请特别注意标有“Migration”的小节。尽管你可能从一个主版本升级到另一个，而不用升级中间版本，你应该阅读全部中间版本的主要发行说明。

通过pg_dumpall升级数据

传统的跨版本升级方法利用pg_dump/pg_dumpall逻辑备份导出数据库，然后在新版本中通过pg_restore进行还原。导出旧版本数据库时推荐使用新版本的pg_dump/pg_dumpall工具，可以利用最新的并行导出和还原功能，同时可以减少数据库膨胀问题。

逻辑备份与还原非常简单但速度比较慢，停机时间取决于数据库的大小，因此适合中小型数据库的升级。

下面介绍这种升级方法的具体操作，假如当前IvorySQL软件的安装目录位于/usr/local/pgsql，同时数据目录位于/usr/local/pgsql/data，我们在同一台服务器上进行升级。

1. 执行逻辑备份之前停止应用程序，确保没有数据更新，因为备份开始后的更新不会被导出。如有必要，可以修改/usr/local/pgsql/data/pg_hba.conf文件禁止其他人访问数据库。然后备份数据库：

```
pg_dumpall > outputfile
```

> 如果已经安装了新版本的IvorySQL，可以使用新版本的pg_dumpall命令备份旧版本数据库。

2.停止旧版本的后台服务：

```
pg_ctl stop
```

或者通过其他方式停止后台服务。

3.如果安装目录没有包含特定版本标识，可以将目录改名，必要时可以再修改回来。可以使用类似以下的命令重命名目录：

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

4.安装新版本IvorySQL软件，假如安装目录仍然是/usr/local/pgsql。

5.初始化一个新的数据库集群，需要使用数据库专用用户（通常是postgres；如果升级版本，应该已经存在该用户）执行操作：

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

6.将旧版本配置文件pg_hba.conf 和 postgresql.conf 等中的改动在对应的新配置文件中再次进行修改。

7.使用数据库专用用户启动新版本的后台服务：

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8.最后利用新版本的psql命令还原数据：

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

为了减少停机时间，可以将新版本的IvorySQL安装到另一个目录，同时使用不同的端口启动服务。然后同时执行数据库的导出和导入：

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

执行以上操作时，新旧版本的后台服务同时运行，新版本使用5433端口，旧版本使用5432端口。

利用pg_upgrade 工具进行升级

pg_upgrade 工具是PostgreSQL
内置的跨版本升级工具，能够对数据库就地升级，不需要执行导出和导入操作。IvorySQL源自于PG，因此也能够使用pg_upgrade 工具进行大版本升级。 下面简要介绍一下CentOS8平台上如何使用 pg_upgrade
将IvorySQL升级到最新的5.6版本。

pg_upgrade 提供了升级前的兼容性检查（-c 或者 --check
选项）功能，可以发现插件、数据类型不兼容等问题。如果指定了—link
选项，新版本服务可以直接使用原有的数据库文件而不需要执行复制，通常可以在几分钟内完成升级操作。

常用的参数包括：

- -b bindir，--old-bindir=bindir：旧的 IvorySQL 可执行文件目录；
- -B bindir，--new-bindir=bindir：新的 IvorySQL 可执行文件目录；
- -d configdir，--old-datadir=configdir：旧版本的数据目录；
- -D configdir，--new-datadir=configdir：新版本的数据目录；
- -c，--check：只检查升级兼容性，不更改任何数据；
- -k，--link：硬链接方式升级；

升级准备：

首先停止旧版本的IvorySQL4.6数据库：

```
/usr/ivory-4/bin/pg_ctl -D ./data stop
```

然后安装新版本的IvorySQL5.6数据库：

```
dnf install -y ivorysql5-5.6
```

初始化新版IvorySQL5.6数据目录：

```
/usr/ivory-5/bin/initdb -D ./data
```

检查版本兼容性：

```
/usr/ivory-5/bin/pg_upgrade --old-datadir=/home/ivorysql/test/4.6/data --new
-datadir=/home/ivorysql/test/5.6/data --old-bindir=/usr/ivory-4/bin/ --new
```

```
-bindir=/usr/ivory-5/bin/ --check
```

最后出现 “Clusters are compatible” 表明两个版本之间的数据不存在兼容性问题，可以进行升级。

正式升级：

```
/usr/ivory-5/bin/pg_upgrade --old-datadir=/home/ivorysql/test/4.6/data --new
-datadir=/home/ivorysql/test/5.6/data --old-bindir=/usr/ivory-4/bin/ --new
-bindir=/usr/ivory-5/bin/
```

看到 Upgrade Complete 说明升级已经顺利完成。

更新统计信息：

pg_upgrade
会创建新的系统表，并重用旧的数据进行升级，统计信息并不会随升级过程迁移，所以在启用新版本之前，应该首先重新收集统计信息，避免没有统计信息导致错误的查询计划。 启动新版本数据库

```
/usr/ivory-5/bin/pg_ctl -D ./data -l logfile start
```

手动运行vacuum命令

```
vacuum --all --analyze-in-stage -h 127.0.0.1 -p 1521
```

升级后清理

```
rm -rf /home/ivorysql/test/4.6/data
```

pg_upgrade 文档概述了上述所需的步骤。

通过复制升级数据

我们也可以使用IvorySQL的已更新版本逻辑复制来创建一个后备服务器，逻辑复制支持在不同主版本的IvorySQL之间的复制。后备服务器可以在同一台计算机或者不同的计算机上。一旦它和主服务器（运行旧版本的IvorySQL）同步好，你可以切换主机并且将后备服务器作为主机，然后关闭旧的数据库实例。这样一种切换使得一次升级的停机时间只有数秒。

这种升级方法可以用内置的逻辑复制工具和外部的逻辑复制系统如pglogical，Slony，Londiste，和Bucardo。

使用 pg_upgrade 将 PostgreSQL 升级到 IvorySQL

`pg_upgrade` 支持将原生 PostgreSQL 集群升级到 IvorySQL。当源集群为原生 PostgreSQL ，而目标为 IvorySQL 集群时，必须使用 `-g`（`--using-ora-pg`）参数。

从 PostgreSQL 升级到 IvorySQL
属于“跨产品”迁移，建议在生产环境操作前完成完整的测试验证，并制定详细的回滚方案。

参数说明

参数	说明
-g / --using-ora-pg	表示源集群为 PostgreSQL，而目标为 IvorySQL 集群。启用此选项后，pg_upgrade 在连接两端集群时统一使用 PostgreSQL 标准端口，从而正确处理升级。

升级步骤

第一步：初始化新 IvorySQL 集群

```
# 使用 IvorySQL 的 initdb 初始化新集群
/opt/ivorysql/bin/initdb -D /data/ivorysql/data
```

第二步：停止源 PostgreSQL 集群

```
/usr/lib/postgresql/16/bin/pg_ctl -D /data/pg/data stop
```

第三步：运行升级检查（可选）

使用 -c 仅做兼容性检查，不修改任何数据：

```
/opt/ivorysql/bin/pg_upgrade \
  -b /usr/lib/postgresql/16/bin \   # 源 PG 可执行文件目录
  -B /opt/ivorysql/bin \            # 目标 IvorySQL 可执行文件目录
  -d /data/pg/data \                # 源 PG 数据目录
  -D /data/ivorysql/data \          # 目标 IvorySQL 数据目录
  -g \                              # 源为 PostgreSQL，目标为 IvorySQL
  -c                                # 仅检查，不升级
```

第四步：执行升级

```
/opt/ivorysql/bin/pg_upgrade \
  -b /usr/lib/postgresql/16/bin \
  -B /opt/ivorysql/bin \
  -d /data/pg/data \
  -D /data/ivorysql/data \
  -g
```

第五步：启动 IvorySQL 并验证

```
/opt/ivorysql/bin/pg_ctl -D /data/ivorysql/data start


# 验证数据库是否正常
/opt/ivorysql/bin/psql -p 5432 -c "SELECT version();"
```

第六步：清理旧集群

升级完成后，pg_upgrade 会生成一个清理脚本：

```
./delete_old_cluster.sh
```

常用参数速查

参数	长选项	说明
-b	--old-bindir	源集群可执行文件目录
-B	--new-bindir	目标集群可执行文件目录
-d	--old-datadir	源集群数据目录
-D	--new-datadir	目标集群数据目录
-g	--using-ora-pg	源为 PostgreSQL 升级到 IvorySQL
-p	--old-port	源集群端口
-P	--new-port	目标集群 PG 端口
-q	--old-oraport	源集群 Oracle 端口
-Q	--new-oraport	目标集群 Oracle 端口
-j	--jobs	并行进程数
-k	--link	使用硬链接替代文件复制
-c	--check	仅检查兼容性，不执行升级
-v	--verbose	输出详细日志

注意事项

- -g 参数仅在 源为纯 PostgreSQL 集群、目标为 IvorySQL 时使用。若源集群已是 IvorySQL，则无需此参数。
- 升级期间源集群和目标集群均须处于停止状态。
- 升级前务必对源集群做完整备份。

管理IvorySQL版本

IvorySQL基于PostgreSQL开发，版本更新频率与PostgreSQL版本更新频率保持一致，每年更新一个大版本，每季度更新一个小版本。IvorySQL目前发布的版本有1.0到5.6，分别基于PostgreSQL 14.0到18.6进行开发，最新版本为IvorySQL 5.6，基于PostgreSQL 18.6进行开发。IvorySQL的所有版本全部都做到了向下兼容。相关版本特性可以查看 官网。

管理IvorySQL数据库访问

IvorySQL使用 角色
的概念管理数据库访问权限。一个角色可以看成是一个数据库用户或者是数据库用户组，这取决于角色被怎样设置。角色可以拥有数据库对象（例如，表和函数）并且能够把那些对象上的权限赋予给其他角色来控制谁能访问哪些对象。此外，还可以把一个角色中的 成员资格
授予给另一个角色，这样允许成员角色使用被赋予给另一个角色的权限。

角色的概念把“用户”和“组”的概念都包括在内。

数据库角色在概念上已经完全与操作系统用户独立开来。
事实上可能维护一个对应关系会比较方便，但是这并非必需。
数据库角色在一个数据库集簇安装范围内是全局的（而不是独立数据库内）。
要创建一个角色，可使用CREATE ROLE SQL 命令：

```
CREATE ROLE name;
```

name遵循 SQL
标识符的规则：或是未经装饰没有特殊字符，或是用双引号包围（实际上，你将总是给该命令要加上额外选项，例如LOGIN。更多细节可见下文）。 要移除一个已有的角色，使用相似的DROP ROLE命令：

```
DROP ROLE name;
```

为了方便，createuser和dropuser程序被提供作为这些 SQL 命令的包装器，它们可以从 shell 命令行调用：

```
createuser name
dropuser name
```

要决定现有角色的集合，检查pg_roles系统目录，例如：

```
SELECT rolname FROM pg_roles;
```

psql程序的\du元命令也可以用来列出现有角色。

为了引导数据库系统，一个刚刚被初始化好的系统总是包含一个预定义角色。这个角色总是一个“superuser”，并且默认情况下（除非在运行initdb时修改）它的名字和初始化数据库集簇的操作系统用户相同。习惯上，这个角色将被命名为postgres。为了创建更多角色，你首先必须以初始角色的身份连接。

每一个到数据库服务器的连接都是使用某个特定角色名建立的，并且这个角色决定发起连接的命令的初始访问权限。要使用一个特定数据库连接的角色名由客户端指示，该客户端以一种应用相关的风格发起连接请求。例如，psql程序使用-
U命令行选项来指定要以哪个角色连接。很多应用假定该名字默认是当前操作系统用户（包括createuser和psql）。因此在角色和操作系统用户之间维护一个名字对应关系通常是很方便的。

一个给定客户端连接能够用来连接的数据库角色的集合由该客户端的认证设置决定，因此，一个客户端不止限于以匹配其操作系统用户的角色连接，就像一个人的登录名不需要匹配她的真实名字一样。因为角色身份决定一个已连接客户端可用的权限集合，在设置一个多用户环境时要小心地配置权限。

一个数据库角色可以有一些属性，它们定义角色的权限并且与客户端认证系统交互。

把用户分组在一起来便于管理权限常常很方便：那样，权限可以被授予一整个组或从一整个组回收。在IvorySQL中通过创建一个表示组的角色来实现，并且然后将在该组角色中的 成员关系 授予给单独的用户角色。

由于角色可以拥有数据库对象并且能持有访问其他对象的特权，删除一个角色常常并非一次DROP
ROLE就能解决。
任何被该用户所拥有的对象必须首先被删除或者转移给其他拥有者，并且任何已被授予给该角色的权限必须被收回。

更多有关数据库访问管理的细节，可以参阅 手册。

定义数据对象

IvorySQL基于PostgreSQL，具有完整的SQL，其定义数据对象可以参考
手册。在此基础之上，IvorySQL为兼容Oracle，还做了一些Oracle专有数据对象的兼容。

VARCHAR2

概述

具有最大长度大小字节或字符的可变长度字符串。 您必须为 VARCHAR2 指定大小。 最小大小为 1 个字节或 1 个字符。

语法

```
VARCHAR2(size)
```

用例

```
create table test(a varchar2(5));
CREATE TABLE

SET NLS_LENGTH_SEMANTICS TO CHAR;
SET

SHOW NLS_LENGTH_SEMANTICS;
 nls_length_semantics
----------------------
 char
(1 row)

insert into test values ('李老师您好');
INSERT 0 1
```

查询数据

IvorySQL基于PostgreSQL开发，具有完全的SQL，查询数据具体的操作可以参考 手册。

使用外部数据

IvorySQL实现了部分的SQL/MED规定，允许我们使用普通SQL查询来访问位于IvorySQL之外的数据。这种数据被称为外部数据（注意这种用法不要和外键混淆，后者是数据库中的一种约束）。

外部数据可以在一个外部数据包装器的帮助下被访问。一个外部数据包装器是一个库，它可以与一个外部数据源通讯，并隐藏连接到数据源和从它获取数据的细节。在contrib模块中有一些外部数据包装器，参见文档。其他类型的外部数据包装器可以在第三方产品中找到。如果这些现有的外部数据包装器都不能满足你的需要，可以自己编写一个，参见 手册。

要访问外部数据，我们需要建立一个外部服务器对象，它根据它所支持的外部数据包装器所使用的一组选项定义了如何连接到一个特定的外部数据源。接着我们需要创建一个或多个外部表，它们定义了外部数据的结构。一个外部表可以在查询中像一个普通表一样地使用，但是在IvorySQL服务器中外部表没有存储数据。不管使用什么外部数据包装器，IvorySQL会要求外部数据包装器从外部数据源获取数据，或者在更新命令的情况下传送数据到外部数据源。

访问远程数据可能需要在外部数据源的授权。这些信息通过一个用户映射提供，它基于当前的IvorySQL角色提供了附加的数据例如用户名和密码。

备份与恢复

由于包含着有价值的数据，IvorySQL数据库应当被定期地备份。虽然过程相当简单，但清晰地理解其底层技术和假设是非常重要的。

有三种不同的基本方法来备份 IvorySQL 数据：

- SQL转储
- 文件系统级备份
- 连续归档

SQL转储

SQL
转储方法的思想是创建一个由SQL命令组成的文件，当把这个文件回馈给服务器时，服务器将利用其中的SQL命令重建与转储时状态一样的数据库。 IvorySQL为此提供了工具pg_dump。这个工具的基本用法是：

```
pg_dump dbname > dumpfile
```

正如你所见，pg_dump把结果输出到标准输出。我们后面将看到这样做有什么用处。
尽管上述命令会创建一个文本文件，pg_dump可以用其他格式创建文件以支持并行
和细粒度的对象恢复控制。

pg_dump是一个普通的IvorySQL客户端应用（尽管是个
相当聪明的东西）。这就意味着你可以在任何可以访问该数据库的远端主机上进行备份工作。但是请记住pg_dump不会以任何特殊权限运行。具体说来，就是它必须要有你想备份的表的读权限，因此为了备份整个数据库你几乎总是必须以一个数据库超级用户来运行它（如果你没有足够的特权来备份整个数据库，你仍然可以使用诸如-n schema 或-t table选项来备份该数据库中你能够访问的部分）。

要声明pg_dump连接哪个数据库服务器，使用命令行选项-h host和 -p port。
默认主机是本地主机或你的HOST环境变量指定的主机。
类似地，默认端口是环境变量PORT或（如果PORT不存在）内建的默认值。
（服务器通常有相同的默认值，所以还算方便。）

pg_dump默认使用与当前操作系统用户名同名的数据库用户名进行连接。 要使用其他名字，要么声明-U选项，要么设置环境变量PGUSER。请注意pg_dump的连接也要通过客户认证机制。

pg_dump对于其他备份方法的一个重要优势是，pg_dump的输出可以很容易地在新版本的IvorySQL中载入，而文件级备份和连续归档都是极度的服务器版本限定的。pg_dump也是唯一可以将一个数据库传送到一个不同机器架构上的方法，例如从一个32位服务器到一个64位服务器。

由pg_dump创建的备份在内部是一致的，
也就是说，转储表现了pg_dump开始运行时刻的数据库快照，且在pg_dump运行过程中发生的更新将不会被转储。pg_dump工作的时候并不阻塞其他的对数据库的操作。（但是会阻塞那些需要排它锁的操作，比如大部分形式的ALTER TABLE）

从转储中恢复

pg_dump生成的文本文件可以由psql程序读取。 从转储中恢复的常用命令是：

```
psql dbname < dumpfile
```

其中dumpfile就是pg_dump命令的输出文件。这条命令不会创建数据库dbname，你必须在执行psql前自己从template0创建（例如，用命令createdb -T template0
dbname）。psql支持类似pg_dump的选项用以指定要连接的数据库服务器和要使用的用户名。参阅psql的手册获取更多信息。 非文本文件转储可以使用pg_restore工具来恢复。

在开始恢复之前，转储库中对象的拥有者以及在其上被授予了权限的用户必须已经存在。如果它们不存在，那么恢复过程将无法将对象创建成具有原来的所属关系以及权限（有时候这就是你所需要的，但通常不是）。

默认情况下，psql脚本在遇到一个SQL错误后会继续执行。你也许希望在遇到一个SQL错误后让psql退出，那么可以设置ON_ERROR_STOP变量来运行psql，这将使psql在遇到SQL错误后退出并返回状态3：

```
psql --set ON_ERROR_STOP=on dbname < infile
```

不管怎样，你将只能得到一个部分恢复的数据库。作为另一种选择，你可以指定让整个恢复作为一个单独的事务运行，这样恢复要么完全完成要么完全回滚。这种模式可以通过向psql传递-1或—single-transaction命令行选项来指定。在使用这种模式时，注意即使是很小的一个错误也会导致运行了数小时的恢复被回滚。但是，这仍然比在一个部分恢复后手工清理复杂的数据库要更好。

pg_dump和psql读写管道的能力使得直接从一个服务器转储一个数据库到另一个服务器成为可能，例如：

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

重要：pg_dump产生的转储是相对于template0。这意味着在template1中加入的任何语言、过程等都会被pg_dump转储。结果是，如果在恢复时使用的是一个自定义的template1，你必须从template0创建一个空的数据库，正如上面的例子所示。

一旦完成恢复，在每个数据库上运行ANALYZE是明智的举动，这样优化器就有有用的统计数据了。

使用pg_dumpall

pg_dump每次只转储一个数据库，而且它不会转储关于角色或表空间（因为它们是集簇范围的）的信息。为了支持方便地转储一个数据库集簇的全部内容，提供了pg_dumpall程序。pg_dumpall备份一个给定集簇中的每一个数据库，并且也保留了集簇范围的数据，如角色和表空间定义。该命令的基本用法是：

```
pg_dumpall > dumpfile
```

转储的结果可以使用psql恢复：

```
psql -f dumpfile postgres
```

（实际上，你可以指定恢复到任何已有数据库名，但是如果你正在将转储载入到一个空集簇中则通常要用（postgres）。在恢复一个pg_dumpall转储时常常需要具有数据库超级用户访问权限，因为它需要恢复角色和表空间信息。如果你在使用表空间，请确保转储中的表空间路径适合于新的安装。

pg_dumpall工作时会发出命令重新创建角色、表空间和空数据库，接着为每一个数据库pg_dump。这意味着每个数据库自身是一致的，但是不同数据库的快照并不同步。

集簇范围的数据可以使用pg_dumpall的—globals-only选项来单独转储。如果在单个数据库上运行pg_dump命令，上述做法对于完全备份整个集簇是必需的。

处理大型数据库

在一些具有最大文件尺寸限制的操作系统上创建大型的pg_dump输出文件可能会出现问题。幸运地是，pg_dump可以写出到标准输出，因此你可以使用标准Unix工具来处理这种潜在的问题。有几种可能的方法：

使用压缩转储。你可以使用你喜欢的压缩程序，例如gzip：

```
pg_dump dbname | gzip > filename.gz
```

恢复：

```
gunzip -c filename.gz | psql dbname
```

或者：

```
cat filename.gz | gunzip | psql dbname
```

使用split。split命令允许你将输出分割成较小的文件以便能够适应底层文件系统的尺寸要求。例如，让每一块的大小为2G字节：

```
pg_dump dbname | split -b 2G - filename
```

恢复：

```
cat filename* | psql dbname
```

如果使用GNU split，可能会把它和gzip一起使用：

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```

它可以使用zcat恢复。

使用pg_dump的自定义转储格式。. 如果 IvorySQL
所在的系统上安装了zlib压缩库，自定义转储格式将在写出数据到输出文件时对其压缩。这将产生和使用gzip时差不多大小的转储文件，但是这种方式的一个优势是其中的表可以被有选择地恢复。下面的命令使用自定义转储格式来转储一个数据库：

```
pg_dump -Fc dbname > filename
```

自定义格式的转储不是psql的脚本，只能通过pg_restore恢复，例如：

```
pg_restore -d dbname filename
```

更多细节可以参阅 手册。

对于非常大型的数据库，你可能需要将split配合其他两种方法之一进行使用。

使用pg_dump的并行转储特性。
为了加快转储一个大型数据库的速度，你可以使用pg_dump的并行模式。它将同时转储多个表。你可以使用-j参数控制并行度。并行转储只支持“目录”归档格式。

```
pg_dump -j num -F d -f out.dir dbname
```

你可以使用pg_restore
-j来以并行方式恢复一个转储。它只能适合于“自定义”归档或者“目录”归档，但不管归档是否由pg_dump -j创建。

文件系统级别备份

另外一种备份策略是直接复制
IvorySQL用于存储数据库中数据的文件，你可以采用任何你喜欢的方式进行文件系统备份，例如：

```
tar -cf backup.tar /usr/local/pgsql/data
```

但是这种方法有两个限制，使得这种方法不实用，或者说至少比pg_dump方法差：

1. 为了得到一个可用的备份，数据库服务器必须被关闭。例如阻止所有连接的半路措施是不起作用的（部分原因是tar和类似工具无法得到文件系统状态的一个原子的快照，还有服务器内部缓冲的原因）。不用说，在恢复数据之前你也需要关闭服务器。
2. 如果你已经深入地了解了数据库的文件系统布局的细节，你可能会有兴趣尝试通过相应的文件或目录来备份或恢复特定的表或数据库。这种方法也不会起作用，因为包含在这些文件中的信息只有配合提交日志文件（pg_xact/*）才有用，提交日志文件包含了所有事务的提交状态。一个表文件只有和这些信息一起才有用。当然也不可能只恢复一个表及相关的pg_xact数据，因为这会导致数据库集簇中所有其他表变得无用。因此文件系统备份值适合于完整地备份或恢复整个数据库集簇。

另一种文件系统备份方法是创建一个数据目录的“一致快照”，如果文件系统支持此功能（并且你相信它的实现正确）。典型的过程是创建一个包含数据库的卷的“冻结快照”，然后从该快照复制整个数据目录（如上，不能是部分复制）到备份设备，最后释放冻结快照。即使在数据库服务器运行时，这种方式也有效。但是，以这种方式创建的备份保存的文件看起来就像数据库没有被正确关闭时的状态。因此，当你从备份数据上启动数据库服务器时，它会认为上一次的服务器实例崩溃了并尝试重放WAL日志。这不是问题，只是需要注意（当然WAL文件必须要包括在备份中）。你可以在拍摄快照之前执行一次CHECKPOINT以便节省恢复时间。

如果你的数据库跨越多个文件系统，可能没有任何方式可以对所有卷获得完全同步的冻结快照。例如，如果你的数据文件和WAL日志放置在不同的磁盘上，或者表空间在不同的文件系统中，可能没有办法使用快照备份，因为快照必须是同步的。在这些情况下，一定要仔细阅读你的文件系统文档以了解其对一致快照技术的支持。

如果没有可能获得同步快照，一种选择是将数据库服务器关闭足够长的时间以建立所有的冻结快照。另一种选择是执行一次连续归档基础备份，因为这种备份对于备份期间发生的文件系统改变是免疫的。这要求在备份过程中允许连续归档，恢复时使用连续归档恢复。

还有一种选择是使用rsync来执行一次文件系统备份。其做法是先在数据库服务器运行时执行rsync，然后关闭数据库服务器足够长时间来做一次rsync --checksum （--checksum是必需的，因为rsync的文件修改时间粒度只能精确到秒）。第二次rsync会比第一次快，因为它只需要传送相对很少的数据，由于服务器是停止的，所以最终结果将是一致的。这种方法允许在最小停机时间内执行一次文件系统备份。

注意一个文件系统备份通常会比一个SQL转储体积更大（例如pg_dump不需要转储索引的内容，而是转储用于重建索引的命令）。但是，做一次文件系统备份可能更快。

连续归档和时间点恢复（PITR）

在任何时间，IvorySQL在数据集簇目录的pg_wal/子目录下都保持有一个预写式日志（WAL）。这个日志存在的目的是为了保证崩溃后的安全：如果系统崩溃，可以“重放”从最后一次检查点以来的日志项来恢复数据库的一致性。该日志的存在也使得第三种备份数据库的策略变得可能：我们可以把一个文件系统级别的备份和WAL文件的备份结合起来。当需要恢复时，我们先恢复文件系统备份，然后从备份的WAL文件中重放来把系统带到一个当前状态。这种方法比之前的方法管理起来要更复杂，但是有其显著的优点：

- 我们不需要一个完美的一致的文件系统备份作为开始点。备份中的任何内部不一致性将通过日志重放（这和崩溃恢复期间发生的并无显著不同）来修正。因此我们不需要文件系统快照功能，只需要tar或一个类似的归档工具。
- 由于我们可以结合一个无穷长的WAL文件序列用于重放，可以通过简单地归档WAL文件来达到连续备份。这对于大型数据库特别有用，因为在其中不方便频繁地进行完全备份。
- 并不需要一直重放WAL项一直到最后。我们可以在任何点停止重放，并得到一个数据库在当时的一致快照。这样，该技术支持时间点恢复：
  在得到你的基础备份以后，可以将数据库恢复到它在其后任何时间的状态。
- 如果我们连续地将一系列WAL文件输送给另一台已经载入了相同基础备份文件的机器，
  我们就得到了一个热后备系统： 在任何时间点我们都能提出第二台机器，
  它差不多是数据库的当前副本。

注意: pg_dump 和
pg_dumpall不会产生文件系统级别的备份，并且不能用于连续归档方案。这类转换是逻辑的并且不包含足够的信息用于WAL重放。

就简单的文件系统备份技术来说，这种方法只能支持整个数据库集簇的恢复，却无法支持其中一个子集的恢复。另外，它需要大量的归档存储：一个基础备份的体积可能很庞大，并且一个繁忙的系统将会产生大量需要被归档的WAL流量。尽管如此，在很多需要高可靠性的情况下，它是首选的备份技术。

要使用连续归档（也被很多数据库厂商称为“在线备份”）成功地恢复，你需要一个从基础备份时间开始的连续的归档WAL文件序列。为了开始，在你建立第一个基础备份之前，你应该建立并测试用于归档WAL文件的过程。对应地，我们首先讨论归档WAL文件的机制。关于如何建立归档和备份的方式以及操作过程中的要点，请参阅 手册。

装卸数据

copy 在 IvorySQL表和标准文件之间移动数据。COPY TO 把一个表的内容复制到一个文件，而COPY FROM
则从一个文件复制数据到一个表（把数据追加到表中原有数据）。COPY TO
也能复制一个SELECT查询的结果。

如果指定了一个列列表，COPY TO将只把指定列的数据复制到文件。对于COPY
FROM，文件中的每个字段将按顺序插入到指定列中。COPY
FROM命令的列列表中没有指定的表列则会采纳其默认值。

带一个文件名的COPY指示IvorySQL服务器直接从一个文件读取或者写入到一个文件。该文件必须是IvorySQL用户（运行服务器的用户ID）可访问的并且应该以服务器的视角来指定其名称。当指定了PROGRAM时，服务器执行给定的命令并且从该程序的标准输出读取或者写入到该程序的标准输入。该程序必须以服务器的视角指定，并且必须是IvorySQL用户可执行的。在指定STDIN或者STDOUT时，数据会通过客户端和服务器之间的连接传输。

运行COPY的每个后端将在pg_stat_progress_copy视图中报告其进度。

大纲

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

```
其中 option 可以是下列之一：

    FORMAT format_name
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

详细参数设置，请参阅 手册。

输出

在成功完成时，一个COPY命令会返回一个形为

```
COPY count
```

count 是被复制的行数。 注意： 如果命令不是COPY … TO STDOUT或者等效的psql命令\copy … to stdout, psql将只打印这个命令标签。这是为了防止弄混命令标签和刚刚打印的数据。

注解

COPY TO只能用于普通表，而不能用于视图，并且不能从子表或子分区复制行。 例如，COPY table TO 复制与SELECT * FROM ONLY table 相同的行。 语法COPY (SELECT * FROM table) TO … 可用于转储一个继承层次结构、分区表或视图中的所有行。

COPY FROM可以被用于普通表、外部表、分区表或者具有INSTEAD OF INSERT触发器的视图。

你必须拥有被COPY TO读取的表上的选择特权， 以及被COPY FROM插入的表上的插入特权。 拥有在命令中列出的列上的特权就足够了。

如果对表启用了行级安全性，相关的SELECT策略将应用于COPY table TO语句。当前，有行级安全性的表不支持COPY FROM。不过可以使用等效的INSERT语句。

COPY命令中提到的文件会被服务器（而不是客户端应用）直接读取或写入。因此它们必须位于数据库服务器（不是客户端）的机器上或者是数据库服务器可以访问的。它们必须是 IvorySQL 用户（运行服务器的用户ID）可访问的并且是可读或者可写的。类似地，用PROGRAM指定的命令也会由服务器（不是客户端应用）直接执行，它也必须是 IvorySQL 用户可以执行的。只允许数据库超级用户或者授予了角色pg_read_server_files、pg_write_server_files及pg_execute_server_program 之一的用户COPY一个文件或者命令，因为它允许读取或者写入服务器有特权访问的任何文件或者运行服务器有特权访问的程序。

不要把COPY和 psql指令 \copy 弄混。\copy会调用 COPY FROM STDIN或者COPY TO STDOUT，然后读取/存储一个 psql客户端可访问的文件中的数据。 因此，在使用\copy时，文件的可访问性和访问权利取决于客户端而不是服务器。

我们推荐在COPY中使用的文件名总是 指定为一个绝对路径。在COPY TO的
情况下服务器会强制这一点，但是对于 COPY FROM你可以选择从一个用相对
路径指定的文件中读取。该路径将根据服务器进程（而不是客户端）
的工作目录（通常是集簇的数据目录）解释。

用PROGRAM执行一个命令可能会受到操作系统 的访问控制机制（如 SELinux）的限制。

COPY FROM将调用目标表上的任何触发器 和检查约束。但是它不会调用规则。

对于标识列，COPY FROM命令将总是写上输入数据中提供的列值，这和INSERT的选项OVERRIDING SYSTEM VALUE的行为一样。

COPY输入和输出受到 DateStyle的影响。为了确保到其他 可能使用非默认DateStyle设置的 IvorySQL
安装的可移植性，在使用 COPY TO前应该把 DateStyle设置为ISO。避免转储把 IntervalStyle设置为
sql_standard的数据也是一个好主意，因为负的区间值可能会
被具有不同IntervalStyle设置的服务器解释错误。

即使数据会被服务器直接从一个文件读取或者写入一个文件而不通过
客户端，输入数据也会被根据ENCODING选项或者当前
客户端编码解释，并且输出数据会被根据ENCODING或 者当前客户端编码进行编码。

COPY会在第一个错误处停止操作。这在 COPY TO的情况下不会导致问题，但是 在COPY
FROM中目标表将已经收到了一 些行。这些行将不会变得可见或者可访问，但是它们仍然占据磁盘空间。
如果在一次大型的复制操作中出现错误，这可能浪费相当可观的磁盘空间。
你可能希望调用VACUUM来恢复被浪费的 空间。

FORCE_NULL和FORCE_NOT_NULL可以被同时
用在同一列上。这会导致把已被引用的空值串转换为空值并且把未引用的空值 串转换为空串。

文件格式

文本格式

在使用text格式时，读取或写入的是一个文本文件，其中每一行就是表中的一行。一行中的列被定界字符分隔。列值本身是由输出函数产生的或者是可被输入函数接受的属于每个属性数据类型的字符串。在为空值的列的位置使用指定的空值串。如果输入文件的任何行包含比预期更多或者更少的列，COPY
FROM将会抛出一个错误。

数据的结束可以表示为一个只包含反斜线-
点号（\.）的单一行。从一个文件读取时，数据结束标记并不是必要的，因为文件结束符就已经足够用了。
只有使用3.0客户端协议之前的客户端应用复制数据时才需要它。

反斜线字符（\）可以被用在COPY数据中来引用被用作行或者列界定符的字符。特别地，如果下列字符作为一个列值的一部分出现，它们 必须被前置一个反斜线：反斜线本身、新行、回车以及 当前的定界符字符。

COPY TO会不加任何反斜线返回指定的空值串。 相反，COPY FROM会在移除反斜线之前把输入
与空值串相匹配。因此，一个空值串（例如\N）不会与实 际的数据值\N（它会被表示为\\N）搞混。

COPY FROM识别下列特殊的反斜线序列：

序列	表示
\b	退格（ASCII 8）
\f	换页（ASCII 12）
\n	新行（ASCII 10）
\r	回车（ASCII 13）
\t	制表（ASCII 9）
\v	纵向制表（ASCII 11）

\digits	反斜线后跟一到三个八进制数字表示该数字代码对应的字节
\xdigits	反斜线加x后跟一到三个十六进制数字表示该数字代码对应的字节

当前，COPY TO不会发出一个八进制或十六进制位
反斜线序列，但是它确实把上面列出的其他序列用于那些控制字符。

任何上述表格中没有提到的其他反斜线字符将被当作表示其本身。不过，要注意
增加不必要的反斜线，因为那可能意外地产生一个匹配数据结束标记（
\.）或者空值串（默认是\N）的字符串。这些字符串 将在完成任何其他反斜线处理之前被识别。

强烈建议产生COPY数据的应用把数据新行和回车分别 转换为\n和\r序列。当前可以把一个数据回车表示为
一个反斜线和回车，把一个数据新行表示为一个反斜线和新行。不过，未来的发行
可能不会接受这些表示。如果在不同的机器之间（例如从 Unix 到 Windows）
传输COPY文件，它们也很容易受到破坏。

所有反斜杠序列都在编码转换后进行解释。
用八进制和十六进制数字反斜杠序列指定的字节必须在数据库编码中形成有效字符。

COPY TO将用一个 Unix 风格的新行（“\n”）终止每一行。运行在 Microsoft Windows
上的服务器则会输出回车/新行（“\r\n”），不过只对
COPY到一个服务器文件这样做。为了做到跨平台一致， COPY TO STDOUT总是发送“\n”而
不管服务器平台是什么。COPY FROM能够处理以
新行、回车或者回车/新行结尾的行。为了减少由作为数据的未加反斜线的新行
或者回车带来的风险，如果输出中的行结束并不完全相似， COPY FROM将会抱怨。

CSV格式

这种格式选项被用于导入和导出很多其他程序（例如电子表格）使用的逗号
分隔值（CSV）文件格式。不同于 IvorySQL标准文本格式使用的转义 规则，它产生并且识别一般的 CSV
转义机制。

每个记录中的值用DELIMITER字符分隔。如果值包含 定界符字符、QUOTE字符、NULL字符串、
一个回车或者换行字符，那么整个值会被加上QUOTE字符
作为前缀或者后缀，并且在该值内QUOTE字符或者 ESCAPE字符的任何一次出现之前放上转义字符。在输出
指定列中非NULL值时，还可以使用 FORCE_QUOTE来强制加上引用。

CSV格式没有标准方式来区分NULL值和空字符串。 IvorySQL的COPY用引用来处理
这种区分工作。NULL被按照NULL参数字符串输出 并且不会被引用，而匹配NULL参数字符串的非NULL
值会被加上引用。例如，使用默认设置时，NULL被写作一个未
被引用的空字符串，而一个空字符串数据值会被写成带双引号（""）。
值的读取遵循类似的规则。你可以用FORCE_NOT_NULL来防止 对指定列的NULL输入比较。你还可以使用
FORCE_NULL把带引用的空值字符串数据值转换成NULL。

因为反斜线在CSV格式中不是一种特殊字符，数据结束标记
\.也可以作为一个数据值出现。为了避免任何解释误会，在
一行上作为孤项出现的\.数据值输出时会自动被引用，并且
输入时如果被引用，则不会被解释为数据结束标记。如果正在载入一个由
另一个应用创建的文件并且其中具有一个未被引用的列且可能具有
\.值，你可能需要在输入文件中引用该值。

注意

CSV格式中，所有字符都是有意义的。一个被空白或者其他 非
DELIMITER字符围绕的引用值将包括那些字符。在导入
来自用空白填充CSV行到固定长度的系统的数据时，这可能
会导致错误。如果出现这种情况，在导入数据到 IvorySQL.之前，你可能需要预处理该
CSV文件以移除拖尾的空白。

注意

CSV 格式将识别并且产生带有包含嵌入的回车和换行的引用值的 CSV 文件。因此文件并不限于文本格式文件的每个表行一行的形式。

注意

很多程序会产生奇怪的甚至偶尔是不合常理的 CSV 文件，因此该文件格式更像是一种习惯而不是标准。因此你可能会碰到一些无法使用这种机制导入的文件，并且COPY也可能产生其他程序无 法处理的文件。

二进制格式

binary格式选项导致所有数据被以二进制格式 而不是文本格式存储/读取。它比文本和CSV格式要快一些，但是二进制格式文件在不同的机器架构和 IvorySQL 版本之间的可移植性要差些。还有，二进制格式与数据格式非常相关。例如不能从一个smallint列中输出二进制数据并且把它读入到一个 integer列中，虽然这样做在文本格式中是可行的。

binary文件格式由一个文件头、零个或者更多个包含行数据的元组以及一个文件尾构成。头部和数据都以网络字节序表示。

文件头

文件头由 15 字节的固定域构成，后面跟着一个变长的头部扩展区。

固定域有：

签名

11-字节的序列PGCOPY\n\377\r\n\0 — 注意零字节是签名的一个必要的部分（该签名是为了能容易地发现文件被 无法正确处理 8 位字符编码的传输所破坏。这个签名将被行尾翻译过滤器、删除零字节、删除高位或者奇偶修改等改变）。

标志域

32-位整数位掩码，用以表示该文件格式的重要方面。位被编号为 从 0 （LSB）到 31（MSB）。注意这个域以网络字节序存放（最高有效位在前），所有该文件格式 中使用的整数域都是这样。16-31 位被保留用来表示严重的文件格式 问题， 读取者如果在这个范围内发现预期之外的被设置位，它应该中止。0-15 位被保留用来表示向后兼容的格式问题，读取者应该简单地略过这个范围内任何预期之外的被设置位。当前只定义了一个标志 位，其他位必须为零：

位 16

如果为 1，表示数据中包含 OID；如果为 0，则不包含。IvorySQL不再支持Oid系统列，但是格式仍然包含该指示符。

头部扩展区长度

32-为整数，表示头部剩余部分的以字节计的长度，不包括其本身。当前，这个长度为零，并且其后就紧跟着第一个元组。未来对该格式的更改可能会允许在头部中表示额外的数据。如果读取者不知道要对头部扩展区数据做什么，可以安静地跳过它。

头部扩展区域被预期包含一个能自我解释的块的序列。 该标志域并不想告诉读取者扩展数据是什么。详细的头部扩展内容的设计留给后来的发行去做。

这种设计允许向后兼容的头部增加（增加头部扩展块或者设置低位标志位）以及非向后兼容的更改（设置高位标志位来表示这类更改并且在需要时向扩展区域 中增加支持数据）。

元组

每一个元组由一个表示元组中域数量的 16 位整数计数开始（当前，一个表中

的所有元组都应该具有相同的计数，但是这可能不会总是为真）。然后是元组 中的每一个域，它是一个32 位的长度字，后面则跟随着这么多个字节的域数
据（长度字不包括其本身，并且可以是零）。作为一种特殊情况，-1 表示一个 NULL 域值。在 NULL情况下，后面不会跟随值字节。

在域之间没有对齐填充或者任何其他额外的数据。

当前，一个二进制格式文件中的所有数据值都被假设为二进制格式（格式代码一）。
可以预见未来的扩展可能会增加一个允许独立指定各列的格式代码的头部域。

要为实际的元组数据决定合适的二进制格式，你应该参考 IvorySQL源码，特别是用于各列
数据类型的*send和*recv函数（通常可 以在源码的src/backend/utils/adt/目录中找到 这些函数）。

如果文件中包含 OID，OID 域会紧跟在域计数字之后。它是一个普通域，
不过它没有被包含在域计数中。注意IvorySQL当前版本不支持oid系统列。

文件尾

文件位由一个包含 -1 的 16 位整数字组成。这很容易与一个 元组的域计数字区分开。

如果一个域计数字不是 -1 也不是期望的列数，读取者应该报告错误。
这提供了一种针对某种数据不同步的额外检查。

示例

下面的例子使用竖线（|）作为域定界符把一个表复制到客户端：

```
COPY country TO STDOUT (DELIMITER '|');
```

从一个文件中复制数据到country表中：

```
COPY country TO STDOUT (DELIMITER '|');
```

只把名称以 'A' 开头的国家复制到一个文件中：

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/usr1/proj/bray/sql/a_list_countries.copy';
```

要复制到一个压缩文件中，你可以用管道把输出导到一个外部压缩程序：

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

这里是一个适合于从STDIN复制到表中的数据：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

> 注意每一行上的空白实际是一个制表符。

下面是用二进制格式输出的相同数据。该数据是用 Unix 工具 od -c过滤后显示的。该表具有三列，第一列类型是char(2)，第二列类型是text， 第三列类型是integer。所有行在第三列都是空值。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

剩余的详细信息可以参阅 手册。

性能管理

查询性能可能受多种因素影响。其中一些因素可以由用户控制，而其他的则属于系统下层设计的基本原理。

使用EXPLAIN

IvorySQL为每个收到查询产生一个查询计划。
选择正确的计划来匹配查询结构和数据的属性对于好的性能来说绝对是最关键的，因此系统包含了一个复杂的规划器来尝试选择好的计划。 你可以使用EXPLAIN命令察看规划器为任何查询生成的查询计划。
阅读查询计划是一门艺术，它要求一些经验来掌握，但是本节只试图覆盖一些基础。

这些例子使用EXPLAIN的默认“text”输出格式，这种格式紧凑并且便于阅读。如果你想把EXPLAIN的输出交给一个程序做进一步分析，你应该使用它的某种机器可读的输出格式（XML、JSON 或 YAML）。

EXPLAIN基础

查询计划的结构是一个计划结点的树。最底层的结点是扫描结点：它们从表中返回未经处理的行。
不同的表访问模式有不同的扫描结点类型：顺序扫描、索引扫描、位图索引扫描。
也还有不是表的行来源，例如VALUES子句和FROM中返回集合的函数，它们有自己的结点类型。如果查询需要连接、聚集、排序、或者在未经处理的行上的其它操作，那么就会在扫描结点之上有其它额外的结点来执行这些操作。 并且，做这些操作通常都有多种方法，因此在这些位置也有可能出现不同的结点类型。
EXPLAIN给计划树中每个结点都输出一行，显示基本的结点类型和计划器为该计划结点的执行所做的开销估计。 第一行（最上层的结点）是对该计划的总执行开销的估计；计划器试图最小化的就是这个数字。

这里是一个简单的例子，只是用来显示输出看起来是什么样的：

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

由于这个查询没有WHERE子句，它必须扫描表中的所有行，因此计划器只能选择使用一个简单的顺序扫描计划。被包含在圆括号中的数字是（从左至右）：

- 估计的启动开销。在输出阶段可以开始之前消耗的时间，例如在一个排序节点里执行排序的时间。
- 估计的总开销。这个估计值基于的假设是计划结点会被运行到完成，即所有可用的行都被检索。不过实际上一个结点的父节点可能很快停止读取所有可用的行（见下面的LIMIT例子）。
- 这个计划结点输出行数的估计值。同样，也假定该结点能运行到完成。
- 预计这个计划结点输出的行平均宽度（以字节计算）。

开销是用规划器的开销参数所决定的捏造单位来衡量的。传统上以取磁盘页面为单位来度量开销；也就是seq_page_cost将被按照习惯设为1.0，其它开销参数将相对于它来设置。本节的例子都假定这些参数使用默认值。

有一点很重要：一个上层结点的开销包括它的所有子结点的开销。还有一点也很重要：这个开销只反映规划器关心的东西。特别是这个开销没有考虑结果行传递给客户端所花费的时间，这个时间可能是实际花费时间中的一个重要因素；但是它被规划器忽略了，因为它无法通过修改计划来改变（我们相信，每个正确的计划都将输出同样的行集）。

行数值有一些小技巧，因为它不是计划结点处理或扫描过的行数，而是该结点发出的行数。这通常比被扫描的行数少一些， 因为有些被扫描的行会被应用于此结点上的任意WHERE子句条件过滤掉。理想中顶层的行估计会接近于查询实际返回、更新、删除的行数。

回到我们的例子：

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的产生非常直接。如果你执行：

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

你会发现tenk1有358个磁盘页面和10000行。 开销被计算为（页面读取数*seq_page_cost）+（扫描的行数*cpu_tuple_cost）。默认情况下，seq_page_cost是1.0，cpu_tuple_cost是0.01， 因此估计的开销是 (358 * 1.0) + (10000 * 0.01) = 458。

现在让我们修改查询并增加一个WHERE条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

请注意EXPLAIN输出显示WHERE子句被当做一个“过滤器”条件附加到顺序扫描计划结点。这意味着该计划结点为它扫描的每一行检查该条件，并且只输出通过该条件的行。因为WHERE子句的存在，估计的输出行数降低了。不过，扫描仍将必须访问所有 10000 行，因此开销没有被降低；实际上开销还有所上升（准确来说，上升了 10000 * cpu_operator_cost）以反映检查WHERE条件所花费的额外 CPU 时间。

这条查询实际选择的行数是
7000，但是估计的rows只是个近似值。如果你尝试重复这个试验，那么你很可能得到略有不同的估计。
此外，这个估计会在每次ANALYZE命令之后改变，
因为ANALYZE生成的统计数据是从该表中随机采样计算的。

现在，让我们把条件变得更严格：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

这里，规划器决定使用一个两步的计划：子计划结点访问一个索引来找出匹配索引条件的行的位置，然后上层计划结点实际地从表中取出那些行。独立地抓取行比顺序地读取它们的开销高很多，但是不是所有的表页面都被访问，这么做实际上仍然比一次顺序扫描开销要少（使用两层计划的原因是因为上层规划结点把索引标识出来的行位置在读取之前按照物理位置排序，这样可以最小化单独抓取的开销。结点名称里面提到的“位图”是执行该排序的机制）。

现在让我们给WHERE子句增加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

新增的条件stringu1 = 'xxx’减少了估计的输出行计数，
但是没有减少开销，因为我们仍然需要访问相同的行集合。
请注意，stringu1子句不能被应用为一个索引条件，因为这个索引只是在unique1列上。
它被用来过滤从索引中检索出的行。因此开销实际上略微增加了一些以反映这个额外的检查。

在某些情况下规划器将更倾向于一个“simple”索引扫描计划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

在这类计划中，表行被按照索引顺序取得，这使得读取它们开销更高，但是其中有一些是对行位置排序的额外开销。 你很多时候将在只取得一个单一行的查询中看到这种计划类型。
它也经常被用于拥有匹配索引顺序的ORDER BY子句的查询中，
因为那样就不需要额外的排序步骤来满足ORDER BY。在此示例中，添加 ORDER BY unique1将使用相同的计划，因为索引已经隐式提供了请求的排序。

规划器可以通过多种方式实现ORDER BY子句。上面的例子表明，这样的排序子句可以隐式实现。
规划器还可以添加一个明确的sort步骤：

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;
                           QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

如果计划的一部分保证对所需排序键的前缀进行排序，那么计划器可能会决定使用incremental sort步骤：

```
EXPLAIN SELECT * FROM tenk1 ORDER BY four, ten LIMIT 100;
                                            QUERY PLAN
-----------------------------------------------------------------
-----------------------------------
 Limit  (cost=521.06..538.05 rows=100 width=244)
   ->  Incremental Sort  (cost=521.06..2220.95 rows=10000 width=244)
         Sort Key: four, ten
         Presorted Key: four
         ->  Index Scan using index_tenk1_on_four on tenk1  (cost=0.29..1510.08
rows=10000 width=244)
```

与常规排序相比，增量排序允许在对整个结果集进行排序之前返回元组，这尤其可以使用LIMIT查询进行优化。
它还可以减少内存使用和将排序溢出到磁盘的可能性，但其代价是将结果集拆分为多个排序批次的开销增加。

如果在WHERE引用的多个行上有独立的索引，规划器可能会选择使用这些索引的一个 AND 或 OR 组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
```

```
                Index Cond: (unique2 > 9000)
```

但是这要求访问两个索引，所以与只使用一个索引并把其他条件作为过滤器相比，它不一定能胜出。如果你变动涉及到的范围，你将看到计划也会相应改变。

下面是一个例子，它展示了LIMIT的效果：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

这是和上面相同的查询，但是我们增加了一个LIMIT这样不是所有的行都需要被检索，并且规划器改变了它的决定。注意索引扫描结点的总开销和行计数显示出好像它会被运行到完成。但是，限制结点在检索到这些行的五分之一后就会停止，因此它的总开销只是索引扫描结点的五分之一，并且这是查询的实际估计开销。之所以用这个计划而不是在之前的计划上增加一个限制结点是因为限制无法避免在位图扫描上花费启动开销，因此总开销会是超过那种方法（25个单位）的某个值。

让我们尝试连接两个表，使用我们已经讨论过的列：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在这个计划中，我们有一个嵌套循环连接结点，它有两个表扫描作为输入或子结点。该结点的摘要行的缩进反映了计划树的结构。连接的第一个（或“outer”）子结点是一个与前面见到的相似的位图扫描。它的开销和行计数与我们从SELECT … WHERE unique1 < 10得到的相同，因为我们将WHERE子句unique1 < 10用在了那个结点上。t1.unique2 = t2.unique2子句现在还不相关，因此它不影响 outer 扫描的行计数。嵌套循环连接结点将为从 outer 子结点得到的每一行运行它的第二个（或“inner”）子结点。当前 outer 行的列值可以被插入 inner 扫描。这里，来自 outer 行的t1.unique2值是可用的，所以我们得到的计划和开销与前面见到的简单SELECT … WHERE t2.unique2 = constant情况相似（估计的开销实际上比前面看到的略低，是因为在t2上的重复索引扫描会利用到高速缓存）。循环结点的开销则被以 outer 扫描的开销为基础设置，外加对每一个 outer 行都要进行一次 inner 扫描

（10 * 7.87），再加上用于连接处理一点 CPU 时间。

在这个例子里，连接的输出行计数等于两个扫描的行计数的乘积，但通常并不是所有的情况中都如此，因为可能有同时提及两个表的
额外WHERE子句，并且因此它只能被应用于连接点，而不能影响任何一个输入扫描。这里是一个例子：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10
width=244)
               Index Cond: (unique2 < 10)
```

条件t1.hundred < t2.hundred不能在tenk2_unique2索引中被测试，因此它被应用在连接结点。这缩减了连接结点的估计输出行计数，但是没有改变任何输入扫描。

注意这里规划器选择了“物化”连接的 inner
关系，方法是在它的上方放了一个物化计划结点。这意味着t2索引扫描将只被做一次，即使嵌套循环连接结点需要读取其数据十次（每个来自 outer
关系的行都要读一次）。物化结点在读取数据时将它保存在内存中，然后在每一次后续执行时从内存返回数据。

在处理外连接时，你可能会看到连接计划结点同时附加有“连接过滤器”和普通“过滤器”条件。连接过滤器条件来自于外连接的ON子句，因此一个无法通过连接过滤器条件的行也能够作为一个空值扩展的行被发出。但是一个普通过滤器条件被应用在外连接条件之后并且因此无条件移除行。在一个内连接中这两种过滤器类型没有语义区别。

如果我们把查询的选择度改变一点，我们可能得到一个非常不同的连接计划：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
```

```
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0)
                     Index Cond: (unique1 < 100)
```

这里规划器选择了使用一个哈希连接，在其中一个表的行被放入一个内存哈希表，在这之后其他表被扫描并且为每一行查找哈希表来寻找匹配。同样要注意缩进是如何反映计划结构的：tenk1上的位图扫描是哈希结点的输入，哈希结点会构造哈希表。然后哈希表会返回给哈希连接结点，哈希连接结点将从它的 outer 子计划读取行，并为每一个行搜索哈希表。

另一种可能的连接类型是一个归并连接，如下所示：

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

归并连接要求它的输入数据被按照连接键排序。在这个计划中，tenk1数据被使用一个索引扫描排序，以便能够按照正确的顺序来访问行。但是对于onek则更倾向于一个顺序扫描和排序，因为在那个表中有更多行需要被访问（对于很多行的排序，顺序扫描加排序常常比一个索引扫描好，因为索引扫描需要非顺序的磁盘访问）。

一种查看变体计划的方法是强制规划器丢弃它认为开销最低的任何策略，这可以使用启用/禁用标志实现例如，如果我们并不认同在前面的例子中顺序扫描加排序是处理表onek的最佳方法，我们可以尝试：

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------
```

```
-----------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000
width=244)
```

这显示规划器认为用索引扫描来排序onek的开销要比用顺序扫描加排序的方式高大约12%。当然，下一个问题是是否真的是这样。我们可以通过使用EXPLAIN ANALYZE来仔细研究一下，如下文所述。

EXPLAIN ANALYZE

可以通过使用EXPLAIN的ANALYZE选项来检查规划器估计值的准确性。通过使用这个选项，EXPLAIN会实际执行该查询，然后显示真实的行计数和在每个计划结点中累计的真实运行时间，还会有一个普通EXPLAIN显示的估计值。例如，我们可能得到这样一个结果：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                        QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10
loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual
time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
(actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
(actual time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

> 注意“actual time”值是以毫秒计的真实时间，而cost估计值被以捏造的单位表示，因此它们不大可能匹配上。在这里面要查看的最重要的一点是估计的行计数是否合理地接近实际值。在这个例子中，估计值都是完全正确的，但是在实际中非常少见。

在某些查询计划中，可以多次执行一个子计划结点。例如，inner索引扫描可能会因为上层嵌套循环计划中的每一个 outer行而被执行一次。在这种情况下，loops值报告了执行该结点的总次数，并且 actual time和行数值是这些执行的平均值。这是为了让这些数字能够与开销估计被显示的方式有可比性。将这些值乘上l

oops值可以得到在该结点中实际消耗的总时间。在上面的例子中，我们在执行tenk2的索引扫描上花费了总共 0.220 毫秒。

在某些情况中，EXPLAIN ANALYZE会显示计划结点执行时间和行计数之外的额外执行统计信息。例如，排序和哈希结点提供额外的信息：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                              QUERY PLAN
-----------------------------------------------------------------
-------------------------------------------------------------------------
 Sort  (cost=717.34..718.09 rows=101 width=488) (actual time=7.761..7.774 rows=100
loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427
rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659
rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101
width=244) (actual time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
width=0) (actual time=0.049..0.049 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

排序结点显示使用的排序方法（尤其是，排序是在内存中还是磁盘上进行）和需要的内存或磁盘空间量。哈希结点显示了哈希桶的数量和批数，以及被哈希表所使用的内存量的峰值（如果批数超过一，也将会涉及到磁盘空间使用，但是并没有被显示）。

另一种类型的额外信息是被一个过滤器条件移除的行数：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                 QUERY PLAN
-------------------------------------------------------------------
--------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107
```

```
rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

这些值对于被应用在连接结点上的过滤器条件特别有价值。只有在至少有一个被扫描行或者在连接结点中一个可能的连接对被过滤器条件拒绝时，“Rows Removed”行才会出现。

一个与过滤器条件相似的情况出现在“有损”索引扫描中。例如，考虑这个查询，它搜索包含一个指定点的多边形：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                QUERY PLAN
-------------------------------------------------------------------
-----------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044
rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

规划器认为（非常正确）这个采样表太小不值得劳烦一次索引扫描，因此我们得到了一个普通的顺序扫描，其中的所有行都被过滤器条件拒绝。但是如果我们强制使得一次索引扫描可以被使用，我们看到：

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                        QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32)
(actual time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

这里我们可以看到索引返回一个候选行，然后它会被索引条件的重新检查拒绝。这是因为一个 GiST 索引对于多边形包含测试是
“有损的”：它确实返回覆盖目标的多边形的行，然后我们必须在那些行上做精确的包含性测试。

EXPLAIN有一个BUFFERS选项可以和ANALYZE一起使用来得到更多运行时统计信息：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                           QUERY PLAN
-----------------------------------------------------------------
-------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual
time=0.323..0.342 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0
loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
(actual time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
(actual time=0.227..0.227 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

BUFFERS提供的数字帮助我们标识查询的哪些部分是对 I/O 最敏感的。

记住因为EXPLAIN ANALYZE实际运行查询，任何副作用都将照常发生，即使查询可能输出的任何结果被丢弃来支持打印EXPLAIN数据。如果你想要分析一个数据修改查询而不想改变你的表，你可以在分析完后回滚命令，例如：

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                           QUERY PLAN
-----------------------------------------------------------------
-------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628
rows=0 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual
time=0.101..0.439 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
(actual time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
```

```
 Execution time: 14.727 ms

ROLLBACK;
```

正如在这个例子中所看到的，当查询是一个INSERT、UPDATE或DELETE命令时，应用表更改的实际工作由顶层插入、更新或删除计划结点完成。这个结点之下的计划结点执行定位旧行以及/或者计算新数据的工作。因此在上面，我们看到我们已经见过的位图表扫描，它的输出被交给一个更新结点，更新结点会存储被更新过的行。还有一点值得注意的是，尽管数据修改结点可能要可观的运行时间（这里，它消耗最大份额的时间），规划器当前并没有对开销估计增加任何东西来说明这些工作。这是因为这些工作对每一个正确的查询计划都得做，所以它不影响计划的选择。

当一个UPDATE或者DELETE命令影响继承层次时， 输出可能像这样：

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
   ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
         Filter: (f1 = 101)
   ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

在这个例子中，更新节点需要考虑三个子表以及最初提到的父表。因此有四个输入的扫描子计划，每一个对应于一个表。为清楚起见，在更新节点上标注了将被更新的相关目标表，显示的顺序与相应的子计划相同（这些标注是从 PostgreSQL 9.5 开始新增的，在以前的版本中读者必须通过 观察子计划才能知道这些目标表）。

EXPLAIN ANALYZE显示的 Planning time是从一个已解析的查询生成查询计划并进行优化所花费的时间，其中不包括解析和重写。

EXPLAIN ANALYZE显示的Execution time包括执行器的启动和关闭时间，以及运行被触发的任何触发器的时间，但是它不包括解析、重写或规划的时间。如果有花在执行BEFORE执行器的时间，它将被包括在相关的插入、更新或删除结点的时间内；但是用来执行AFTER触发器的时间没有被计算，因为AFTER触发器是在整个计划完成后被触发的。在每个触发器（BEFORE或AFTER）也被独立地显示。注意延迟约束触发器将不会被执行，直到事务结束，并且因此根本不会被EXPLAIN ANALYZE考虑。

警告

在两种有效的方法中EXPLAIN ANALYZE所度量的运行时间可能偏离同一个查询的正常执行。首先，由于不会有输出行被递交给客户端，网络传输开销和 I/O 转换开销没有被包括在内。其次，由EXPLAIN

ANALYZE所增加的度量开销可能会很可观，特别是在操作系统调用gettimeofday()很慢的机器上。你可以使用pg_test_timing工具来度量在你的系统上的计时开销。

EXPLAIN结果不应该被外推到与你实际测试的非常不同的情况。例如，一个很小的表上的结果不能被假定成适合大型表。规划器的开销估计不是线性的，并且因此它可能为一个更大或更小的表选择一个不同的计划。一个极端例子是，在一个只占据一个磁盘页面的表上，你将几乎总是得到一个顺序扫描计划，而不管索引是否可用。规划器认识到它在任何情况下都将采用一次磁盘页面读取来处理该表，因此用额外的页面读取去查看一个索引是没有价值的（我们已经在前面的polygon_tbl例子中见过）。

在一些情况中，实际的值和估计的值不会匹配得很好，但是这并非错误。一种这样的情况发生在计划结点的执行被LIMIT或类似的效果很快停止。例如，在我们之前用过的LIMIT查询中：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                          QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244)
(actual time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

索引扫描结点的估计开销和行计数被显示成好像它会运行到完成。但是实际上限制结点在得到两个行之后就停止请求行，因此实际的行计数只有 2
并且运行时间远低于开销估计所建议的时间。这并非估计错误，这仅仅一种估计值和实际值显示方式上的不同。

归并连接也有类似的现象。如果一个归并连接用尽了一个输入并且其中的最后一个键值小于另一个输入中的下一个键值，它将停止读取另一个输入。在这种情况下，不会有更多的匹配并且因此不需要扫描第二个输入的剩余部分。这会导致不读取一个子结点的所有内容，其结果就像在LIMIT中所提到的。另外，如果 outer（第一个）子结点包含带有重复键值的行，inner（第二个）子结点会被倒退并且被重新扫描来找能匹配那个键值的行。EXPLAIN ANALYZE会统计相同 inner 行的重复发出，就好像它们是真实的额外行。当有很多 outer 重复时，对 inner 子计划结点所报告的实际行计数会显著地大于实际在 inner 关系中的行数。

由于实现的限制，BitmapAnd 和 BitmapOr 结点总是报告它们的实际行计数为零。

通常，EXPLAIN将显示规划器生成的每个计划节点。
但是，在某些情况下，执行器可以不执行某些节点，因为根据规划时不可用的参数值能确定这些节点无法产生任何行。 （当前，这仅会在扫描分区表的Append或MergeAppend节点的子节点中发生。）
发生这种情况时，将从EXPLAIN输出中省略这些计划节点，并显示Subplans Removed：N的标识。

规划器使用的统计信息

单列统计信息

如我们在上一节所见，查询规划器需要估计一个查询要检索的行数，这样才能对查询计划做出好的选择。
本节对系统用于这些估计的统计信息进行一个快速的介绍。

统计信息的一个部分就是每个表和索引中的项的总数，以及每个表和索引占用的磁盘块数。这些信息保存在pg_class表的reltuples和relpages列中。 我们可以用类似下面的查询查看这些信息：

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

这里我们可以看到tenk1包含 10000 行， 它的索引也有这么多行，但是索引远比表小得多（不奇怪）。

出于效率考虑，reltuples和relpages不是实时更新的
，因此它们通常包含有些过时的值。它们被VACUUM、ANALYZE和几个 DDL 命令（例如CREATE INDEX）更新。一个不扫描全表的VACUUM或ANALYZE操作（常见情况）将以它扫描的部分为基础增量更新reltuples计数，这就导致了一个近似值。在任何情况中，规划器将缩放它在pg_class中找到的值来匹配当前的物理表尺寸，这样得到一个较紧的近似。

大多数查询只是检索表中行的一部分，因为它们有限制要被检查的行的WHERE子句。
因此规划器需要估算WHERE子句的选择度，即符合WHERE子句中每个条件的行的比例。
用于这个任务的信息存储在pg_statistic系统目录中。 在pg_statistic中的项由ANALYZE和VACUUM ANALYZE命令更新， 并且总是近似值（即使刚刚更新完）。

除了直接查看pg_statistic之外，
手工检查统计信息的时候最好查看它的视图pg_stats。pg_stats被设计为更容易阅读。
而且，pg_stats是所有人都可以读取的，而pg_statistic只能由超级用户读取（这样可以避免非授权用户从统计信息中获取一些其他人的表的内容的信息。pg_stats视图被限制为只显示当前用户可读的表）。例如，我们可以：

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+-----------------------------------
 name    | f         |  -0.363388 | I- 580                       Ramp+
         |           |            | I- 880                       Ramp+
         |           |            | Sp Railroad                      +
         |           |            | I- 580                           +
         |           |            | I- 680                       Ramp
 name    | t         |  -0.284859 | I- 880                       Ramp+
         |           |            | I- 580                       Ramp+
         |           |            | I- 680                       Ramp+
```

```
         |           |            | I- 580                          +
         |           |            | State Hwy 13                  Ramp
(2 rows)
```

注意，这两行显示的是相同的列，一个对应开始于road表（inherited=t）的完全继承层次，另一个只包括road表本身（inherited=f）。

ANALYZE在pg_statistic中存储的信息量（特别是每个列的most_common_vals中的最大项数和histogram_bounds数组）可以用ALTER TABLE SET STATISTICS命令为每一列设置，或者通过设置配置变量default_statistics_target进行全局设置。 目前的默认限制是 100 个项。提升该限制可能会让规划器做出更准确的估计（特别是对那些有不规则数据分布的列），其代价是在pg_statistic中消耗了更多空间，并且需要略微多一些的时间来计算估计数值。相比之下，比较低的限制可能更适合那些数据分布比较简单的列。

更多规划器对统计信息的使用可以参阅 手册。

扩展统计信息

常常可以看到由于查询子句中用到的多个列相互关联而运行着糟糕的执行计划的慢查询。规划器通常会假设多个条件是彼此独立的，这种假设在列值相互关联的情况下是不成立的。由于常规的统计信息天然的针对个体列的性质，它们无法捕捉到跨列关联的知识。不过，IvorySQL有能力计算多元统计信息，它能捕捉这类信息。

由于可能的列组合数非常巨大，所以不可能自动计算多元统计信息。可以创建扩展统计信息对象（更常被称为统计信息对象）来指示服务器获得跨感兴趣列集合的统计信息。

统计信息对象可以使用CREATE STATISTICS命令创建。这样一个对象的创建仅仅是创建了一个目录项来表示对统计信息有兴趣。实际的数据收集是由ANALYZE（或者是一个手工命令，或者是后台的自动分析）执行的。收集到的值可以在pg_statistic_ext_data目录中看到。

ANALYZE基于它用来计算常规单列统计信息的表行样本来计算扩展统计信息。由于样本的尺寸会随着表或者表列的统计信息目标（如前一节所述）增大而增加，更大的统计信息目标通常将会导致更准确的扩展统计信息，同时也会导致更多花在计算扩展统计信息之上的时间。

下面的小节介绍当前支持的扩展统计信息类型。

函数依赖

最简单的一类扩展统计信息跟踪函数依赖，这是在数据库范式定义中使用的概念。如果列a的值的知识足以决定列b的值，即不会有两个行具有相同的a值但是有不同的b值，我们就说列b函数依赖于列a。在一个完全规范化的数据库中，函数依赖应该仅存在于主键和超键上。不过，实际上很多数据集合会由于各种原因无法被完全规范化，常见的例子是为了性能而有意地反规范化。即使在一个完全规范化的数据库中，也会有某些列之间的部分关联，这些可以表达成部分函数依赖。

函数依赖的存在直接影响了特定查询中估计的准确性。如果一个查询包含独立列和依赖列上的条件，依赖列上的条件不会进一步降低结果的尺寸。但是如果没有函数依赖的知识，查询规划器将假定条件是独立的，导致对结果尺寸的低估。

要告知规划器有关函数依赖的信息，ANALYZE可以收集跨列依赖的测度。评估所有列组之间的依赖程度可能会昂贵到不可实现，因此数据收集被限制为针对那些在一个统计信息对象中一起出现的列组（用dependencies选项定义）。建议只对强相关的列组创建dependencies统计信息，以避免ANALYZE以及后期查询规划中不必要的开销。

这里是一个收集函数依赖统计信息的例子：

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;
```

```
ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
---------+---------+----------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

这里可以看到列1（邮编）完全决定列5（城市），因此系数为1.0，而城市仅决定42%的邮编，意味着有很多城市（58%）有多个邮编。

在为涉及函数依赖列的查询计算选择度时，规划器会使用依赖系数来调整针对条件的选择度估计，这样就不会产生低估。

函数依赖的限制

当前只有在考虑简单等值条件（将列与常量值比较）和具有常量值的IN
子句时，函数依赖才适用。不会使用它们来改进比较两个列或者比较列和表达式的等值条件的估计，
也不会用它们来改进范围子句、LIKE或者任何其他类型的条件。

在用函数依赖估计时，规划器假定在涉及的列上的条件是兼容的并且因此是冗余的。如果它们是不兼容的，正确的估计将是零行，但是那种可能性不会被考虑。例如，给定一个这样的查询

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

规划器将会忽视city子句，因为它不改变选择度，这是正确的。不过，即便真地只有零行满足下面的查询，规划器也会做出同样的假设

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

不过，函数依赖统计信息无法提供足够的信息来排除这种情况。

在很多实际情况中，这种假设通常是能满足的。例如，在应用程序中可能有一个GUI仅允许选择兼容的城市和邮编值用在查询中。但是如果不是这样，函数依赖可能就不是一个可行的选项。

多元可区分值计数

单列统计信息存储每一列中可区分值的数量。在组合多个列（例如GROUP BY a,
b）时，如果规划器只有单列统计数据，则对可区分值数量的估计常常会错误，导致选择不好的计划。

为了改进这种估计，ANALYZE可以为列组收集可区分值统计信息。和以前一样，为每一种可能的列组合做这件事情是不切实际的，因此只会为一起出现在一个统计信息对象（用ndistinct选项定义）中的列组收集数据。将会为列组中列出的列的每一种可能的组合都收集数据。

继续之前的例子，ZIP代码表中的可区分值计数可能像这样：

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM zipcodes;
```

```
ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts2';
-[ RECORD 1 ]-------------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

这表示有三种列组合有33178个可区分值：ZIP代码和州、ZIP代码和城市、ZIP代码+城市+周（事实上对于表中给定的一个唯一的ZIP代码，它们本来就应该是相等的）。另一方面，城市和州的组合只有27435个可区分值。

建议只对实际用于分组的列组合以及分组数错误估计导致了糟糕计划的列组合创建ndistinct统计信息对象。否则，ANALYZE循环只会被浪费。

多元MCV列表

为每列存储的另一种统计信息是频繁值列表。
这样可以对单个列进行非常准确的估计，但是对于在多个列上具有条件的查询，可能会导致严重的错误估计。

为了改善这种估计，ANALYZE可以收集列组合上的MCV列表。 与功能依赖和n-distinct系数类似，对每种可能的列分组进行此操作都是不切实际的。
在这种情况下，甚至更是如此，因为MCV列表（与功能依赖性和n-distinct系数不同）存储了公共列值。
因此，仅收集在使用mcv选项定义的统计对象中同时出现的那些列组的数据。

继续前面的示例，邮政编码表的MCV列表可能类似于以下内容（与更简单的统计信息不同，它需要一个函数来检查MCV内容）：

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts3';

 index |         values         | nulls | frequency | base_frequency
-------+------------------------+-------+-----------+----------------
     0 | {Washington, DC}       | {f,f} |  0.003467 |        2.7e-05
     1 | {Apo, AE}              | {f,f} |  0.003067 |        1.9e-05
     2 | {Houston, TX}          | {f,f} |  0.002167 |       0.000133
     3 | {El Paso, TX}          | {f,f} |     0.002 |       0.000113
     4 | {New York, NY}         | {f,f} |  0.001967 |       0.000114
     5 | {Atlanta, GA}          | {f,f} |  0.001633 |        3.3e-05
     6 | {Sacramento, CA}       | {f,f} |  0.001433 |        7.8e-05
     7 | {Miami, FL}            | {f,f} |    0.0014 |          6e-05
```

```
     8 | {Dallas, TX}          | {f,f} |  0.001367 |        8.8e-05
     9 | {Chicago, IL}         | {f,f} |  0.001333 |        5.1e-05
   ...
(99 rows)
```

这表明城市和州的最常见组合是华盛顿特区，实际频率（在样本中）约为0.35%。
组合的基本频率（根据简单的每列频率计算）仅为0.0027%，导致两个数量级的低估。

建议仅在实际在条件中一起使用的列的组合上创建MCV统计对象，对于这些组合，错误估计组数会导致糟糕的执行计划。 否则，只会浪费ANALYZE和规划时间。

用显示JOIN子句控制规划器

我们可以在一定程度上用显式JOIN语法控制查询规划器。要明白为什么需要它，我们首先需要一些背景知识。

在一个简单的连接查询中，例如：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

规划器可以自由地按照任何顺序连接给定的表。例如，它可以生成一个使用WHERE条件a.id = b.id连接 A 到 B 的查询计划，然后用另外一个WHERE条件把 C 连接到这个连接表。或者它可以先连接 B 和 C 然后再连接 A 得到同样的结果。 或者也可以连接 A 到 C 然后把结果与 B 连接 —
不过这么做效率不好，因为必须生成完整的 A 和 C
的迪卡尔积，而在WHERE子句中没有可用条件来优化该连接（IvorySQL执行器中的所有连接都发生在两个输入表之间， 所以它必须以这些形式之一建立结果）。
重要的一点是这些不同的连接可能性给出在语义等效的结果，但在执行开销上却可能有巨大的差别。
因此，规划器会对它们进行探索并尝试找出最高效的查询计划。

当一个查询只涉及两个或三个表时，那么不需要考虑很多连接顺序。但是可能的连接顺序数随着表数目的增加成指数增长。
当超过十个左右的表以后，实际上根本不可能对所有可能性做一次穷举搜索，甚至对六七个表都需要相当长的时间进行规划。
当有太多的输入表时，IvorySQL规划器将从穷举搜索切换为一种遗传概率搜索，它只需要考虑有限数量的可能性（切换的阈值用geqo_threshold运行时参数设置）。遗传搜索用时更少，但是并不一定会找到最好的计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度。例如，考虑：

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

尽管这个查询的约束表面上和前一个非常相似，但它们的语义却不同， 因为如果 A 里有任何一行不能匹配 B 和 C的连接表中的行，它也必须被输出。因此这里规划器对连接顺序没有什么选择：它必须先连接 B 到 C，然后把 A 连接到该结果上。
相应地，这个查询比前面一个花在规划上的时间更少。在其它情况下，规划器就有可能确定多种连接顺序都是安全的。例如，给定：

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

将 A 首先连接到 B 或 C 都是有效的。当前，只有FULL JOIN完全约束连接顺序。大多数涉及LEFT JOIN或RIGHT JOIN的实际情况都在某种程度上可以被重新排列。

显式连接语法（INNER JOIN、CROSS JOIN或无修饰的JOIN）在语义上和FROM中列出输入关系是一样的，因此它不约束连接顺序。

即使大多数类型的JOIN并不完全约束连接顺序，但仍然可以指示IvorySQL查询规划器将所有JOIN子句当作有连接顺序约束来对待。例如，这里的三个查询在逻辑上是等效的：

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

但如果我们告诉规划器遵循JOIN的顺序，那么第二个和第三个还是要比第一个花在规划上的时间少。这个效果对于只有三个表的连接而言是微不足道的，但对于数目众多的表，可能就是救命稻草了。

要强制规划器遵循显式JOIN的连接顺序， 我们可以把运行时参数join_collapse_limit设置为1(其它可能值在下文讨论)。

你不必为了缩短搜索时间来完全约束连接顺序，因为可以在一个普通FROM列表里使用JOIN操作符。例如，考虑：

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

如果设置join_collapse_limit = 1，那么这就强迫规划器先把 A 连接到 B，然后再连接到其它的表上，但并不约束它的选择。在这个例子中，可能的连接顺序的数目减少了 5 倍。

按照这种方法约束规划器的搜索是一个有用的技巧，不管是对减少规划时间还是对引导规划器生成好的查询计划。 如果规划器按照默认选择了一个糟糕的连接顺序，你可以通过JOIN语法强迫它选择一个更好的顺序 — 假设你知道一个更好的顺序。我们推荐进行实验。

一个非常相近的影响规划时间的问题是把子查询压缩到它们的父查询中。例如，考虑：

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能在使用包含连接的视图时出现；该视图的SELECT规则将被插入到引用视图的地方，得到与上文非常相似的查询。 通常，规划器会尝试把子查询压缩到父查询里，得到：

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这样通常会生成一个比独立的子查询更好些的计划（例如，outer 的WHERE条件可能先把 X 连接到 A 上，这样就消除了 A 中的许多行，因此避免了形成子查询的全部逻辑输出）。但是同时，我们增加了规划的时间；在这里，我们用五路连接问题替代了两个独立的三路连接问题。这样的差别是巨大的，因为可能的计划数的是按照指数增长的。如果有超过from_collapse_limit个FROM项将会导致父查询，规划器将尝试通过停止提升子查询来避免卡在巨大的连接搜索问题中。你可以通过调高或调低这个运行时参数在规划时间和计划的质量之间取得平衡。

from_collapse_limit和join_collapse_limit的命名相似，因为它们做的几乎是同一件事：一个控制规划器何时将把子查询“平面化”，另外一个控制何时把显式连接平面化。通常，你要么把join_collapse_limit设置成和from_collapse_limit一样（这样显式连接和子查询的行为类似）， 要么把join_collapse_limit设置为1（如果你想用显式连接控制连接顺序）。但是你可以把它们设置成不同的值，这样你就可以细粒度地调节规划时间和运行时间之间的平衡。

填充一个数据库

第一次填充数据库时可能需要插入大量的数据。本节包含一些如何让这个处理尽可能高效的建议。

禁用自动提交

在使用多个INSERT时，关闭自动提交并且只在最后做一次提交（在普通 SQL
中，这意味着在开始发出BEGIN并且在结束时发出COMMIT。某些客户端库可能背着你就做了这些，在这种情况下你需要确定在你需要做这些时该库确实帮你做了）。如果你允许每一个插入都被独立地提交，IvorySQL要为每一个被增加的行做很多工作。在一个事务中做所有插入的一个额外好处是：如果一个行的插入失败则所有之前插入的行都会被回滚，这样你不会被卡在部分载入的数据中。

使用COPY

使用COPY在一条命令中装载所有记录，而不是一系列INSERT命令。 COPY命令是为装载大量行而优化过的；它没INSERT那么灵活，但是在大量数据装载时导致的负荷也更少。
因为COPY是单条命令，因此使用这种方法填充表时无须关闭自动提交。

如果你不能使用COPY，那么使用PREPARE来创建一个预备INSERT语句也有所帮助，然后根据需要使用EXECUTE多次。这样就避免了重复分析和规划INSERT的负荷。不同接口以不同的方式提供该功能，
可参阅接口文档中的“预备语句”。

请注意，在载入大量行时，使用COPY几乎总是比使用INSERT快，
即使使用了PREPARE并且把多个插入被成批地放入一个单一事务。

同样的事务中，COPY比更早的CREATE TABLE或TRUNCATE命令更快。 在这种情况下，不需要写
WAL，因为在一个错误的情况下，包含新载入数据的文件不管怎样都将被移除。不过，只有当wal_level设置为minimal（此时所有的命令必须写 WAL）时才会应用这种考虑。

移除索引

如果你正在载入一个新创建的表，最快的方法是创建该表，用COPY批量载入该表的数据，然后创建表需要的任何索引。在已存在数据的表上创建索引要比在每一行被载入时增量地更新它更快。

如果你正在对现有表增加大量的数据，删除索引、载入表然后重新创建索引可能是最好的方案。
当然，在缺少索引的期间，其它数据库用户的数据库性能将会下降。
我们在删除唯一索引之前还需要仔细考虑清楚，因为唯一约束提供的错误检查在缺少索引的时候会丢失。

移除外键约束

和索引一样，“成批地”检查外键约束比一行行检查效率更高。
因此，先删除外键约束、载入数据然后重建约束会很有用。
同样，载入数据和约束缺失期间错误检查的丢失之间也存在平衡。

更重要的是，当你在已有外键约束的情况下向表中载入数据时，
每个新行需要一个在服务器的待处理触发器事件（因为是一个触发器的触发会检查行的外键约束）列表的条目。载入数百万行会导致触发器事件队列溢出可用内存，
造成不能接受的交换或者甚至是命令的彻底失败。因此在载入大量数据时，可能需要（而不仅仅是期望）删除并重新应用外键。如果临时移除约束不可接受，那唯一的其他办法可能是就是将载入操作分解成更小的事务。

增加 maintenance_work_mem

在载入大量数据时，临时增大maintenance_work_mem配置变量可以改进性能。这个参数也可以帮助加速CREATE INDEX命令和ALTER TABLE ADD FOREIGN KEY命令。
它不会对COPY本身起很大作用，所以这个建议只有在你使用上面的一个或两个技巧时才有用。

增加max_wal_size

临时增大max_wal_size配置变量也可以让大量数据载入更快。
这是因为向IvorySQL中载入大量的数据将导致检查点的发生比平常（由checkpoint_timeout配置变量指定）更频繁。无论何时发生一个检查点时，所有脏页都必须被刷写到磁盘上。

通过在批量数据载入时临时增加max_wal_size，所需的检查点数目可以被缩减。

禁用WAL归档和流复制

当使用 WAL
归档或流复制向一个安装中载入大量数据时，在录入结束后执行一次新的基础备份比处理大量的增量 WAL 数据更快。为了防止载入时记录增量
WAL，通过将wal_level设置为minimal、将archive_mode设置为off以及将max_wal_senders设置为零来禁用归档和流复制。 但需要注意的是，修改这些设置需要重启服务。

除了避免归档器或 WAL 发送者处理 WAL
数据的时间之外，这样做将实际上使某些命令更快，因为如果wal_level是minimal并且当前子事务（或顶级事务）创建或截断了它们更改的表或索引，则它们根本不编写 WAL。（通过在最后执行一个fsync而不是写 WAL，它们能以更小地代价保证崩溃安全）。

事后运行ANALYZE

不管什么时候你显著地改变了表中的数据分布后，我们都强烈推荐运行ANALYZE。这包括向表中批量载入大量数据。运行ANALYZE（或者VACUUM ANALYZE）保证规划器有表的最新统计信息。
如果没有统计数据或者统计数据过时，那么规划器在查询规划时可能做出很差劲决定，导致在任意表上的性能低下。需要注意的是，如果启用了 autovacuum 守护进程，它可能会自动运行ANALYZE。

关于pg_dump的一些注记

pg_dump生成的转储脚本自动应用上面的若干个（但不是全部）技巧。
要尽可能快地载入pg_dump转储，你需要手工做一些额外的事情（请注意，这些要点适用于恢复一个转储，而不是创建它的时候。同样的要点也适用于使用psql载入一个文本转储或用pg_restore从一个pg_dump归档文件载入）。

> **默认情况下，pg_dump使用COPY，并且当它在生成一个完整的模式和数据转储时，它会很小心地先装载数据，然后创建索引和外键。因此在这种情况下，一些指导方针是被自动处理的。你需要做的是：**

- 为maintenance_work_mem和max_wal_size设置适当的（即比正常值大的）值。
- 如果使用 WAL
  归档或流复制，在转储时考虑禁用它们。在载入转储之前，可通过将archive_mode设置为off、将wal_level设置为minimal以及将max_wal_senders设置为零（在录入dump前）来实现禁用。
  之后，将它们设回正确的值并执行一次新的基础备份。
- 采用pg_dump和pg_restore的并行转储和恢复模式进行实验并且找出要使用的最佳并发任务数量。通过使用-j选项的并行转储和恢复应该能为你带来比串行模式高得多的性能。
- 考虑是否应该在一个单一事务中恢复整个转储。要这样做，将-1或—single-transaction命令行选项传递给psql或pg_restore。
  当使用这种模式时，即使是一个很小的错误也会回滚整个恢复，可能会丢弃已经处理了很多个小时的工作。根据数据间的相关性， 可能手动清理更好。如果你使用一个单一事务并且关闭了 WAL
  归档，COPY命令将运行得最快。
- 如果在数据库服务器上有多个 CPU 可用，可以考虑使用pg_restore的—jobs选项。这允许并行数据载入和索引创建。
- 之后运行ANALYZE。

一个只涉及数据的转储仍将使用COPY，但是它不会删除或重建索引，并且它通常不会触碰外键。
因此当载入一个只有数据的转储时，如果你希望使用那些技术，你需要负责删除并重建索引和外键。在载入数据时增加max_wal_size仍然有用，但是不要去增加maintenance_work_mem；不如说在以后手工重建索引和外键时你已经做了这些。并且不要忘记在完成后执行ANALYZE。

非持久设置

持久性是数据库的一个保证已提交事务的记录的特性（即使是发生服务器崩溃或断电）。

然而，持久性会明显增加数据库的负荷，因此如果你的站点不需要这个保证，IvorySQL可以被配置成运行更快。在这种情况下，你可以调整下列配置来提高性能。除了下面列出的，在数据库软件崩溃的情况下也能保证持久性。当这些设置被使用时，只有突然的操作系统停止会产生数据丢失或损坏的风险。

- 将数据库集簇的数据目录放在一个内存支持的文件系统上（即RAM磁盘）。这消除了所有的数据库磁盘I/O,但将数据存储限制到可用的内存量（可能有交换区）。
- 关闭fsync； 不需要将数据刷入磁盘。
- 关闭synchronous_commit；可能不需要在每次提交时强制把WAL写入磁盘。这种设置可能会在数据库崩溃时带来事务丢失的风险（但是没有数据破坏）。
- 关闭full_page_writes； 不需要警惕部分页面写入。
- 增加max_wal_size和checkpoint_timeout； 这会降低检查点的频率，但会增加/pg_wal的存储要求。
- 创建不做日志的表来避免WAL写入，不过这会让表在崩溃时不安全。

数据迁移

.1. 迁移指南

迁移概述

数据库迁移是指将数据从一个数据库转移到另一个数据库的过程，两端的数据库可能是PostgreSQL, IvorySQL, MySQL, Oracle, SQL Server等。迁移过程是一个具有挑战性的复杂过程，需要对数据库的原理以及各自的特性了如指掌。如果应用已经部署到生产环境并处于正常运行状态，为了保持业务不中断运行，并且不发生数据丢失，数据库迁移后需进行平滑的应用迁移。

迁移后数据库和系统应符合以下要求：

- 迁移后的数据库系统应完全承载原数据库系统的数据。避免迁移过程中数据丢失导致新的数据库系统数据不完整。
- 迁移后的数据库系统应完全适配原有数据库的功能。避免迁移后因数据类型或语法、函数的不支持，且无替代方案，导致整个业务系统的无法运行或抛错。
- 迁移后的数据库应适配整个业务系统的上下游，稳定可靠的保障整个业务系统的运行。
- 迁移后的数据库各方面综合性能不能弱于原数据库，为整个业务系统提供性能保证。

迁移工具Ora2Pg

Ora2Pg 是一个免费的工具，用于将 Oracle 数据库迁移到 IvorySQL 兼容的模式。它连接您的 Oracle 数据库，自动扫描并提取它的结构或数据，然后生成可以装载到 IvorySQL 数据库的 SQL 脚本。Ora2Pg 可以从逆向工程 Oracle 数据库到大型企业数据库迁移，或者简单地将一些 Oracle 数据复制到 IvorySQL 数据库中。它非常容易使用，并且不需要任何 Oracle 数据库知识，而不需要提供连接到Oracle数据库所需的参数。

Ora2Pg 由一个 Perl 脚本（ora2pg）以及一个 Perl 模块（Ora2Pg.pm）组成，唯一需要做的事情就是修改它的配置文件 ora2pg.conf，设置连接 Oracle 数据库的 DSN 和一个可选的 SCHEMA 名称。完成之后，只需要设置导出的类型：TABLE（包括约束和索引）、VIEW、MVIEW、TABLESPACE、SEQUENCE、INDEXES、TRIGGER、GRANT、FUNCTION、PROCEDURE、PACKAGE、PARTITION、TYPE、INSERT 或 COPY、FDW、QUERY、KETTLE 以及 SYNONYM。

默认情况下，Ora2Pg 导出一个 SQL 文件，可以通过 IvorySQL 客户端工具 psql 执行导出的SQL文件。当进行数据迁移时，可以在配置文件中设置一个目标数据库的 DSN，直接将数据从Oracle导入到 IvorySQL 数据库中。

对象	ora2pg是否支持
view	是
trigger	是，某些情况下需要手工修改脚本
sequence	是
function	是
procedure	是，某些情况下需要手工修改脚本
type	是，某些情况下需要手工修改脚本
materialized view	是，某些情况下需要手工修改脚本

迁移Oracle数据库至IvorySQL

环境准备

linux环境	Oracle版本	IvorySQL版本

Centos Stream 9	19.0.0.0	4.2

依赖环境安装

安装Perl

```
[root@localhost /]# dnf install -y perl perl-ExtUtils-CBuilder perl-ExtUtils-MakeMaker
[root@localhost /]# perl -v


This is perl 5, version 16, subversion 3 (v5.16.3) built for x86_64-linux-thread-multi
(with 44 registered patches, see perl -V for more detail)


Copyright 1987-2012, Larry Wall


Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.


Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl".  If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.
```

安装DBI模块

DBI，Database Independent Interface，是 Perl 语言连接数据库的接口

下载地址 https://mirrors.sjtug.sjtu.edu.cn/cpan/modules/by-module/Plack/TIMB/DBI-1.643.tar.gz

```
# tar zxvf DBI-1.643.tar.gz
# cd DBI-1.643/
# perl Makefile.PL
# make && make install
```

安装DBD-Oracle

下载地址：https://sourceforge.net/projects/ora2pg/

设置环境变量; 加载环境变量;因为必须定义ORACLE_HOME环境变量;本例在ivorysql用户下配置环境变量

```
export ORACLE_HOME=/opt/oracle/product/19c/dbhome_1
# tar -zxvf DBD-Oracle-1.76.tar.gz
# source /home/postgres/.bashrc
# cd DBD-Oracle-1.76
```

```
# perl Makefile.PL
# make && make install
```

安装Ora2pg

下载地址：https://sourceforge.net/projects/ora2pg/

```
# tar -xjf ora2pg-24.0.tar.bz2
# cd ora2pg-24.0
# perl Makefile.PL
# make && make install
```

默认安装在/usr/local/bin/目录下

检查软件环境：

```
# vi check.pl
#!/usr/bin/perl
use strict;
use ExtUtils::Installed;
my $inst= ExtUtils::Installed->new();
my @modules = $inst->modules();
foreach(@modules)
{
        my $ver = $inst->version($_) || "???";
        printf("%-12s --  %s\n", $_, $ver);

}
exit;



# perl check.pl
DBD::Oracle  --  1.76
DBD::Pg      --  3.8.0
DBI          --  1.643
Ora2Pg       --  24.0
Perl         --  5.16.3
设置环境变量
```

export PERL5LIB=<your_install_dir> #export PERL5LIB=/usr/local/bin/

```
=== 源端准备工作
```

```
更新oracle统计信息 提高性能
```

BEGIN DBMS_STATS.GATHER_SCHEMA_STATS('SH'); DBMS_STATS.GATHER_SCHEMA_STATS('SCOTT'); DBMS_STATS.GATHER_SCHEMA_STATS('HR'); DBMS_STATS.GATHER_DATABASE_STATS; DBMS_STATS.GATHER_DICTIONARY_STATS; END;/

```
查询源端对象的类型

```bash
SYS@PROD1>set pagesize 200
SYS@PROD1>select distinct OBJECT_TYPE from  dba_objects where OWNER in
('SH','SCOTT','HR') ;
OBJECT_TYPE
-------------------
INDEX PARTITION
TABLE PARTITION
SEQUENCE
PROCEDURE
LOB                             X
TRIGGER
DIMENSION                       X
MATERIALIZED VIEW
TABLE
INDEX
VIEW
11 rows selected.
```

ora2pg导出表结构

配置ora2pg.conf:

默认情况下，Ora2Pg会查找/etc/ora2pg/ora2pg.conf配置文件，如果文件存在，您只需执行：/usr/local/bin/ora2pg

```
cat /etc/ora2pg/ora2pg.conf.dist  | grep -v ^# |grep -v ^$ >ora2pg.conf
vi ora2pg.conf
# cat ora2pg.conf
ORACLE_HOME     /opt/oracle/product/19c/dbhome_1
ORACLE_DSN      dbi:Oracle:host=localhost;sid=ORCLCDB;port=1521
ORACLE_USER     system
ORACLE_PWD      oracle
SCHEMA          SH
EXPORT_SCHEMA  1         # 将用户导入到PostgreSQL数据库中
DISABLE_UNLOGGED  1       #避免将NOLOGGING属性设为UNLOGGED
SKIP  fkeys ukeys checks      #跳过外键 唯一 和检查约束
```
```

```
TYPE
TABLE,VIEW,GRANT,SEQUENCE,TABLESPACE,PROCEDURE,TRIGGER,FUNCTION,PACKAGE,PARTITION,TYPE
,MVIEW,QUERY,DBLINK,SYNONYM,DIRECTORY,TEST,TEST_VIEW
NLS_LANG    AMERICAN_AMERICA.UTF8
OUTPUT     sh.sql
```

> 1. 只能同时执行一种类型的导出，因此TYPE指令必须是唯一的。如果您有多个，则只会在文件中找到最后一个。但我测试就可以同时导出多个类型的。
> 2. 请注意，您可以通过向TYPE指令提供以逗号分隔的导出类型列表来链接多个导出，但在这种情况下，您不能将COPY或INSERT与其他导出类型一起使用。
> 3. 某些导出类型不能或不应该直接加载到 IvorySQL 数据库中，仍然需要很少的手动编辑。GRANT，TABLESPACE，TRIGGER，FUNCTION，PROCEDURE，TYPE，QUERY和PACKAGE导出类型就是这种情况，特别是如果您有PLSQL代码或Oracle特定SQL。
> 4. 对于TABLESPACE，您必须确保系统上存在文件路径，对于SYNONYM，您可以确保对象的所有者和模式对应于新的PostgreSQL数据库设计。
> 5. 建议导出表结构时，一个类型一个类型的操作，避免其它错误相互影响。

测试连接

设置Oracle数据库DSN后，您可以执行ora2pg以查看它是否有效：

```
#  ora2pg -t SHOW_VERSION -c ora2pg.conf

Oracle Database 19c Enterprise Edition Release 19.0.0.0.0
```

迁移成本评估

估算从Oracle到PostgreSQL的迁移过程的成本并不容易。为了获得对此迁移成本的良好评估，Ora2Pg将检查所有数据库对象，所有函数和存储过程，以检测是否仍有一些对象和PL / SQL代码无法由Ora2Pg自动转换。

Ora2Pg具有内容分析模式，该模式检查Oracle数据库以生成有关Oracle数据库包含的内容和无法导出的内容的文本报告。

```
# ora2pg -t SHOW_REPORT --estimate_cost  -c ora2pg.conf
[========================>] 11/11 tables (100.0%) end of scanning.
[========================>] 11/11 objects types (100.0%) end of objects auditing.
--------------------------------------------------------------------------------
Ora2Pg v24.0 - Database Migration Report
--------------------------------------------------------------------------------
Version Oracle Database 19c Enterprise Edition Release 19.0.0.0.0
Schema  SH
Size    287.25 MB
--------------------------------------------------------------------------------
Object  Number  Invalid Estimated cost  Comments        Details
--------------------------------------------------------------------------------
```

```
DATABASE LINK   0       0       0       Database links will be exported as SQL/MED
IvorySQL's Foreign Data Wrapper (FDW) extensions using oracle_fdw.
DIMENSION       5       0       0
GLOBAL TEMPORARY TABLE  0       0       0       Global temporary table are not
supported by PostgreSQL and will not be exported. You will have to rewrite some
application code to match the PostgreSQL temporary table behavior.
INDEX   20      0       3.4     14 index(es) are concerned by the export, others are
automatically generated and will do so on PostgreSQL. Bitmap will be exported as
btree_gin index(es) and hash index(es) will be exported as b-tree index(es) if any.
Domain index are exported as b-tree but commented to be edited to mainly use FTS.
Cluster, bitmap join and IOT indexes will not be exported at all. Reverse indexes are
not exported too, you may use a trigram-based index (see pg_trgm) or a reverse()
function based index and search. Use 'varchar_pattern_ops', 'text_pattern_ops' or
'bpchar_pattern_ops' operators in your indexes to improve search with the LIKE
operator respectively into varchar, text or char columns.      11 bitmap index(es). 1
domain index(es). 2 b-tree index(es).

INDEX PARTITION 196     0       0       Only local indexes partition are exported,
they are build on the column used for the partitioning.

JOB     0       0       0       Job are not exported. You may set external cron job
with them.

MATERIALIZED VIEW       2       0       6       All materialized view will be exported
as snapshot materialized views, they are only updated when fully refreshed.

SYNONYM 0       0       0       SYNONYMs will be exported as views. SYNONYMs do not
exists with PostgreSQL but a common workaround is to use views or set the PostgreSQL
search_path in your session to access object outside the current schema.

TABLE   11      0       1.1     1 external table(s) will be exported as standard
table. See EXTERNAL_TO_FDW configuration directive to export as file_fdw foreign
tables or use COPY in your code if you just want to load data from external files.
Total number of rows: 1063384. Top 10 of tables sorted by number of rows:. sales has
918843 rows. costs has 82112 rows. customers has 55500 rows.
supplementary_demographics has 4500 rows. times has 1826 rows. promotions has 503
rows. products has 72 rows. countries has 23 rows. channels has 5 rows.
sales_transactions_ext has 0 rows. Top 10 of largest tables:.

TABLE PARTITION 56      0       5.6     Partitions are exported using table
inheritance and check constraint. Hash and Key partitions are not supported by
PostgreSQL and will not be exported.  56 RANGE partitions..

VIEW    1       0       1       Views are fully supported but can use specific
functions.


--------------------------------------------------------------------------------

Total   291     0       17.10   17.10 cost migration units means approximatively 1
```

```
man-day(s). The migration unit was set to 5 minute(s)
--------------------------------------------------------------------------------
Migration level : A-1
--------------------------------------------------------------------------------
Migration levels:

    A - Migration that might be run automatically

    B - Migration with code rewrite and a human-days cost up to 5 days

    C - Migration with code rewrite and a human-days cost above 5 days

Technical levels:

    1 = trivial: no stored functions and no triggers

    2 = easy: no stored functions but with triggers, no manual rewriting

    3 = simple: stored functions and/or triggers, no manual rewriting

    4 = manual: no stored functions but with triggers or views with code rewriting

    5 = difficult: stored functions and/or triggers with code rewriting

--------------------------------------------------------------------------------
```

导出SH表构

```
#  ora2pg  -c ora2pg.conf
[========================>] 11/11 tables (100.0%) end of scanning.

[========================>] 12/12 tables (100.0%) end of table export.

[========================>] 1/1 views (100.0%) end of output.

[========================>] 0/0 sequences (100.0%) end of output.

[========================>] 0/0 procedures (100.0%) end of procedures export.

[========================>] 0/0 triggers (100.0%) end of output.

[========================>] 0/0 functions (100.0%) end of functions export.
```

```
[========================>] 0/0 packages (100.0%) end of output.

[========================>] 56/56 partitions (100.0%) end of output.

[========================>] 0/0 types (100.0%) end of output.

[========================>] 2/2 materialized views (100.0%) end of output.
[========================>] 0/0 dblink (100.0%) end of output.

[========================>] 0/0 synonyms (100.0%) end of output.

[========================>] 2/2 directory (100.0%) end of output.

Fixing function calls in output files....
```

导出SH用户数据

配置ora2pg.conf 的TYPE 为COPY 或 INSERT

```
# cp ora2pg.conf sh_data.conf

# vi sh_data.conf

ORACLE_HOME     /opt/oracle/product/19c/dbhome_1

ORACLE_DSN      dbi:Oracle:host=localhost;sid=ORCLCDB;port=1521

ORACLE_USER     system

ORACLE_PWD      oracle

SCHEMA          SH

EXPORT_SCHEMA  1

DISABLE_UNLOGGED  1

SKIP  fkeys ukeys checks

TYPE           COPY

NLS_LANG    AMERICAN_AMERICA.UTF8
```

```
OUTPUT     sh_data.sql
```

导出数据

```
# ora2pg  -c sh_data.conf

[========================>] 11/11 tables (100.0%) end of scanning.

[========================>] 5/5 rows (100.0%) Table CHANNELS (5 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

[>                        ]     0/82112 rows (0.0%) Table COSTS_1995 (0 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

[>                        ]     0/82112 rows (0.0%) Table COSTS_H1_1997 (0 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

[>                        ]     0/82112 rows (0.0%) Table COSTS_1996 (0 recs/sec)

[>                        ]       5/1063384 total rows (0.0%) - (0 sec., avg: 5
recs/sec).

...............................................................

[========================>] 4500/4500 rows (100.0%) Table SUPPLEMENTARY_DEMOGRAPHICS
(4500 recs/sec)

[======================== ] 1061558/1063384 total rows (99.8%) - (45 sec., avg: 23590
recs/sec).

[========================>] 1826/1826 rows (100.0%) Table TIMES (1826 recs/sec)

[========================>] 1063384/1063384 total rows (100.0%) - (45 sec., avg: 23630
recs/sec).

[========================>] 1063384/1063384 rows (100.0%) on total estimated data (45
sec., avg: 23630 recs/sec)
```

```
Fixing function calls in output files...
```

查看导出的文件：

```
[root@test01 ora2pg]# ls -lrt *.sql

-rw-r--r-- 1 root root 15716 Jul  2 21:21 TABLE_sh.sql

-rw-r--r-- 1 root root   858 Jul  2 21:21 VIEW_sh.sql

-rw-r--r-- 1 root root  2026 Jul  2 21:21 TABLESPACE_sh.sql

-rw-r--r-- 1 root root   345 Jul  2 21:21 SEQUENCE_sh.sql

-rw-r--r-- 1 root root  2382 Jul  2 21:21 GRANT_sh.sql

-rw-r--r-- 1 root root   344 Jul  2 21:21 TRIGGER_sh.sql

-rw-r--r-- 1 root root   346 Jul  2 21:21 PROCEDURE_sh.sql

-rw-r--r-- 1 root root   344 Jul  2 21:21 PACKAGE_sh.sql

-rw-r--r-- 1 root root   345 Jul  2 21:21 FUNCTION_sh.sql

-rw-r--r-- 1 root root  6771 Jul  2 21:21 PARTITION_sh.sql

-rw-r--r-- 1 root root   341 Jul  2 21:21 TYPE_sh.sql

-rw-r--r-- 1 root root   342 Jul  2 21:21 QUERY_sh.sql

-rw-r--r-- 1 root root   950 Jul  2 21:21 MVIEW_sh.sql

-rw-r--r-- 1 root root   344 Jul  2 21:21 SYNONYM_sh.sql

-rw-r--r-- 1 root root   926 Jul  2 21:21 DIRECTORY_sh.sql

-rw-r--r-- 1 root root   343 Jul  2 21:21 DBLINK_sh.sql

-rw-r--r-- 1 root root 55281235 Jul  2 17:11 sh_data.sql
```

以同样的方法分别导出HR，SCOTT用户数据。

在IvorySQL环境中创建orcl库

创建ORCL 数据库

```
# su - ivorysql

Last login: Tue Jul  2 20:04:30 CST 2019 on pts/3

$ createdb orcl

$ psql

psql (18.6)

Type "help" for help.




ivorysql=# \l

                                                List of databases
   Name    |  Owner   | Encoding  | Collate | Ctype | ICU Locale | Locale Provider |
Access privileges
-----------+----------+-----------+---------+-------+------------+-----------------+
-----------------------
 ivorysql  | ivorysql | SQL_ASCII | C       | C     |            | libc            |
 orcl      | ivorysql | SQL_ASCII | C       | C     |            | libc            |
 postgres  | ivorysql | SQL_ASCII | C       | C     |            | libc            |
 template0 | ivorysql | SQL_ASCII | C       | C     |            | libc            |
=c/ivorysql          +
           |          |           |         |       |            |                 |
ivorysql=CTc/ivorysql
 template1 | ivorysql | SQL_ASCII | C       | C     |            | libc            |
=c/ivorysql          +
           |          |           |         |       |            |                 |
ivorysql=CTc/ivorysql


(5 rows)

ivorysql=#
```

```
创建SH,HR,SCOTT 用户:
```

```
$ psql orcl

psql (18.6)

Type "help" for help.

orcl=#

orcl=# create user sh with password 'sh';

CREATE ROLE
```

迁移门户

导入表结构

由于有物化视图，在TABLE_sh.sql
里包含了物化视图的索引，会创建失败。需先创建表，在创建物化视图，最后创建索引。

取消物化视图索引，后面单独创建：

```
CREATE INDEX fw_psc_s_mv_chan_bix ON fweek_pscat_sales_mv (channel_id);

CREATE INDEX fw_psc_s_mv_promo_bix ON fweek_pscat_sales_mv (promo_id);

CREATE INDEX fw_psc_s_mv_subcat_bix ON fweek_pscat_sales_mv (prod_subcategory);

CREATE INDEX fw_psc_s_mv_wd_bix ON fweek_pscat_sales_mv (week_ending_day);

CREATE TEXT SEARCH CONFIGURATION en (COPY = pg_catalog.english);
ALTER TEXT SEARCH CONFIGURATION en ALTER MAPPING FOR hword, hword_part, word WITH
unaccent, english_stem;
```

```
psql orcl  -f  tab.sql

ALTER TABLE PARTITION sh.sales OWNER TO sh;
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
```

```
ALTER TABLE
ALTER TABLE
ALTER TABLE
...................................
```

给对象授权

```
cat psql orcl  -f  GRANT_sh.sql
CREATE USER SH WITH PASSWORD 'change_my_secret' LOGIN;
ALTER TABLE sh.fweek_pscat_sales_mv OWNER TO sh;
GRANT ALL ON  sh.fweek_pscat_sales_mv TO sh;
```

导入物化视图结构

物化视图需要相关查询权限,所以导入权限,注意这里要跟上用户

```
$  psql orcl sh -f  MVIEW_sh.sql
SELECT 0
SELECT 0
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

导入视图

```
$  psql orcl  -f  VIEW_sh.sql
SET
SET
SET
CREATE VIEW
```

导入分区表

```
$  psql orcl  -f  PARTITION_sh.sql
SET
SET
SET
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
```

```
CREATE TABLE
..........................
```

导入数据

```
$  psql orcl   -f   sh_data.sql
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 0
SET
COPY 4500
SET
COPY 1826
COMMIT
```

数据验证

源库,目标端抽取部份对象对比:

```
SYS@PROD1>select count(*) from sh.products;
  COUNT(*)
----------
        72

orcl=#  select count(*) from sh.products;
 count
-------
    72
(1 row)
---------------------------------------------------------------------------

SYS@PROD1>select count(*) from sh.channels;
```

```
  COUNT(*)

----------

         5

orcl=#   select count(*) from sh.channels;
 count
-------

     5

(1 row)

---------------------------------------------------------------------------

SYS@PROD1>select count(*) from sh.customers ;

  COUNT(*)
----------

     55500
orcl=# select count(*) from sh.customers ;
 count
-------
 55500
(1 row)
```

生成迁移模板

使用时，两个选项—project_base和—init_project向Ora2Pg表明他必须创建一个项目模板，其中包含工作树，配置文件和从Oracle数据库导出所有对象的脚本。 生成通用配置文件。
1.创建脚本export_schema.sh以自动执行所有导出。2.创建脚本import_all.sh以自动执行所有导入。例：

```
mkdir -p  /ora2pg/migration

# ora2pg --project_base /ora2pg/migration/ --init_project test_project
Creating project test_project.
/ora2pg/migration//test_project/
        schema/
                dblinks/
                directories/
```

```
                functions/
                grants/
                mviews/
                packages/
                partitions/
                procedures/
                sequences/
                synonyms/
                tables/
                tablespaces/
                triggers/
                types/
                views/
        sources/
                functions/
                mviews/
                packages/
                partitions/
                procedures/
                triggers/
                types/
                views/
        data/
        config/
        reports/
Generating generic configuration file
Creating script export_schema.sh to automate all exports.
Creating script import_all.sh to automate all imports.
```

IvorySQL开发者

社区贡献指南

IvorySQL社区协作流程

IvorySQL 社区采用一套
闭环式开源协作流程，确保从问题提出到版本发布，每一个环节都能闭合反馈、持续改进。这一流程鼓励用户与开发者形成良性互动，让社区开发始终围绕实际需求持续演进。

整个协作闭环流程如下：

⊠ 提问题（Issue）

用户或开发者在 GitHub 的 Issues 页面提交 Bug、功能建议或使用反馈。

⊠ 问题讨论（Discussion）

维护者与社区成员就问题展开讨论，确认问题性质与优先级，加入 ToDo List。

⊠⊠ 开发分支（Fork & Dev）

开发者认领 Issue，Fork 仓库并在本地开发测试，准备提交代码。

⊠ 提交 Pull Request（PR）

将开发分支 Push 到 Fork 仓库后，向上游仓库发起合并请求。

⊠ 代码评审（Review）

维护者或核心开发者对 PR 进行评审，提出修改建议并确保质量。

⊠ 合并主分支（Merge）

审核通过后，PR 被合并至主分支，对应的 Issue 被关闭。

⊠ 版本发布（Release）

项目定期发布新版本（每季度小版本，每年大版本），包含最新的修复与功能。

⊠ 用户测试（Test）

用户升级使用新版，反馈新问题，新的 Issue 随之产生，形成完整的反馈循环。

通过这套完整的闭环协作机制，IvorySQL 实现了问题响应 → 开发贡献 → 质量保障 → 发布反馈的全流程闭合，推动项目持续健康演进。

IvorySQL社区贡献指南

IvorySQL是一个由核心开发团队主导、社区共同维护的开源项目。我们欢迎用户、贡献者和维护者的加入，共同推动IvorySQL的发展。如果您希望看到您的代码或文档更改被添加到IvorySQL并出现在将来的版本中，本节的内容介绍是您需要知道的。

在参与贡献前，请确认您当前的参与身份，以便更高效地了解适合您的贡献方式：

- 用户 ☒ 欢迎提交问题反馈、功能建议，并参与社区讨论。
- 贡献者 ☒ 请先签署 CLA，然后选择您感兴趣的项目模块进行 Issue 认领和代码提交。
- 维护者 ☒ 请参考维护职责，包括社区规划、代码评审和协作机制建设。

无论您以何种身份加入，IvorySQL 社区都非常欢迎您的参与和支持！
IvorySQL社区欢迎并赞赏所有类型的贡献，期待您的加入！

☒ 请务必阅读并遵守我们的 IvorySQL社区行为准则。

注册Github账号

无论您是要提交 Issue、参与讨论，还是贡献代码与文档，您都需要使用 GitHub 账号登录并与 IvorySQL 项目进行交互。

请参考https://docs.github.com/en/get-started/start-your-journey注册您的github账号，并熟悉Git工具和工作流。

IvorySQL源码托管在github: https://github.com/IvorySQL。

用户

作为用户，您在使用 IvorySQL 过程中扮演着重要角色。我们鼓励您：

反馈问题与需求

- 发现 Bug、性能缺陷或文档不准确？
- 有新的功能建议或使用体验改进？

如果您准备向社区上报 Bug 或者提交需求，请在 IvorySQL 社区对应的仓库上提交 Issue，并参考Issue

提交指南。

参与社区讨论

- 通过 邮件列表进行讨论
- 加入 GitHub讨论，补充信息或验证问题
- 在 微信、Discord等聊天群参与技术交流

贡献者

我们欢迎代码、文档、测试等各类贡献。

文档贡献

如果您新增或更新某个插件适配文档，请同步更新 IvorySQL生态插件适配列表
概述页面中的插件表格，至少补充或更新插件名称、版本、功能描述、适用场景以及详情页链接，确保概述页与插件详情页信息一致。

签署CLA

在提交代码或文档贡献之前，为了确保代码合法合规，个人或企业贡献者需要签署贡献者许可协议(CLA)。签署CLA是IvorySQL社区接受贡献的必要条件，以确保您的贡献被合法分发。请根据下列链接下载CLA进行签署并将签署后的CLA发送至 cla@ivorysql.org。

- 个人贡献者
- 企业贡献者

未签署CLA的Pull Request将无法进入评审阶段。

找到您感兴趣的项目

我们将仓库划分为多个子项目，您可以从如下列表中找到感兴趣的项目及其代码仓库

代码仓库	描述
IvorySQL	负责社区IvorySQL数据库的开发和维护
Ivory-www	负责社区的官网开发和维护
ivory-operator	负责IvorySQL云原生数据库及周边工具开发和维护
docker_library	负责IvorySQL多架构的镜像构建开发和维护
ivory-cloud	负责IvorySQL云服务平台及周边生态开发和维护
Ivorysql_docs	负责社区的文档中心开发和维护
ivory-doc-builder	负责Ivorysql_docs的编译
Ivorysql_web	负责社区的文档中心网站维护
Ivorysql_wasm	负责IvorySQL在线易用体验网站的开发和维护

给自己分配Issue

您可以将自己创建的Issue或者愿意处理的Issue分配给自己。
只需要在评论框内输入/assign，机器人就会将问题分配给您。 每个 Issue
下面可能已经有参与者的交流和讨论，如果您感兴趣，也可以在评论框中发表自己的意见参与 Issue 讨论。

开发与提交Pull Request

对于提交一个PR应该保持一个功能，或者一个bug提交一次。禁止多个功能一次提交。

提交 GitHub PR 之前，请先创建或认领对应的 Issue。所有 PR 都必须有至少一个对应的
Issue，用于关联需求、缺陷、讨论、评审和合并结果；没有对应 Issue 的 PR 不符合社区贡献流程。

Fork仓库

前往项目主页，点击Fork按钮，将IvorySQL项目Fork到您自己的GitHub账户中。

编码

使用如下命令将项目克隆到本地进行开发：

git clone https://github.com/$user/IvorySQL.git （将 $user 替换为你的 GitHub ID）。

git checkout -b feature/your-feature-name

在提交代码前，请确保通过回归测试

创建一个Pull Request并提交

打开你 Fork 的仓库： https://github.com/$user/IvorySQL.git

点击 Compare & pull request 按钮填写PR信息

```
Fix test
功能描述
```

```
leave a comment
对该提交功能进行比较详细的描述
```

请在 PR 标题或描述中显式关联对应 Issue 编号，例如 `Fixes #123` 或 `Refs #123`。

点击Create pull request 按钮即可提交。

维护者

维护者负责进行IvorySQL代码的管理，PR审查，主导版本发布与IvorySQL发展方向。

社区规划

- 制定版本规划和 Roadmap
- 跟踪与评估社区需求
- 维护公开的 TODO 列表

代码管理

- 参与 Pull Request 评审
- 审查安全问题，保障项目健康

流程与治理机制

- 优化协作机制（代码贡献指南、PR 模板等）
- 建立漏洞响应机制和行为守则

致谢

感谢每一位参与 IvorySQL 的开发者、文档编辑者、测试人员和使用者。正是有了你们的付出，IvorySQL 才能不断成长！我们欢迎所有人参与 IvorySQL
社区贡献，我们的目标是发展一个由贡献者组成的活跃、健康的社区。

.1. 开发者指南

概览

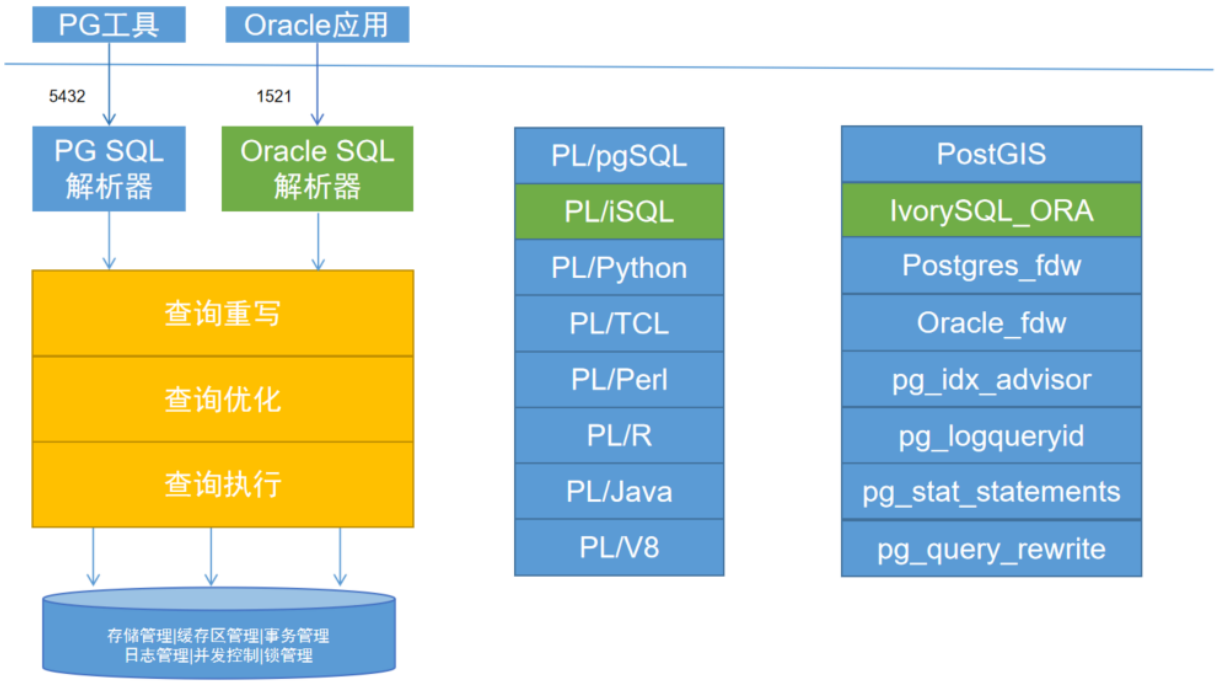
IvorySQL在开源PostgreSQL数据库的基础上提供独特的附加功能。

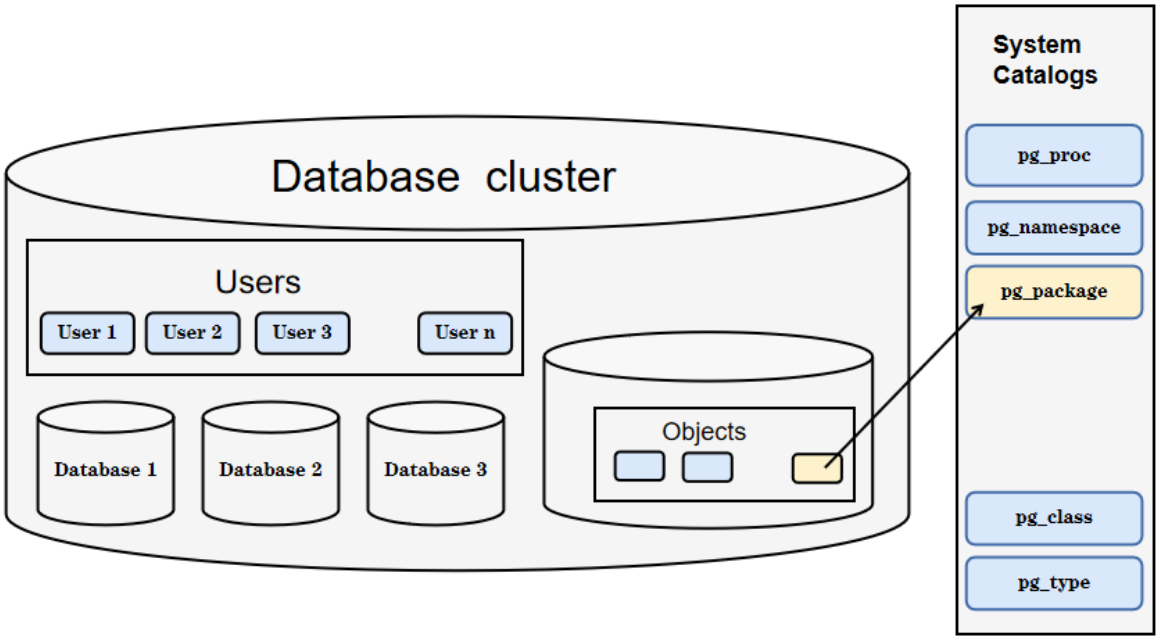
IvorySQL致力于通过创新和建立在开源数据库解决方案之上为其终端用户提供价值。我们的目标是为中小型企业提供一个具有高性能、可扩展性、可靠性和易于使用的解决方案。

IvorySQL提供的扩展功能将使用户能够建立高性能和可扩展的PostgreSQL数据库集群，具有更好的数据库兼容性和管理。这简化了从其他DBMS迁移到PostgreSQL的过程，增强了数据库管理经验。

架构概述

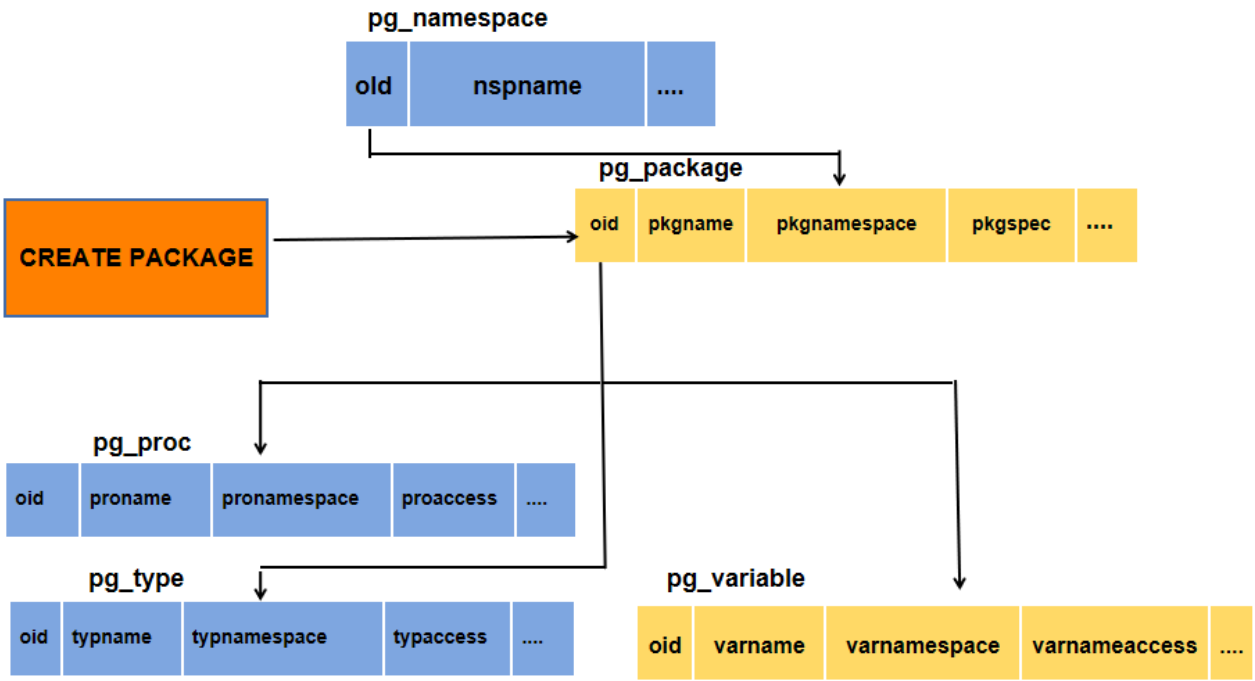
在对原有的PostgreSQL改动最小的前提下，实现对Oracle的兼容。我们需要实现双parser、双端口、模式PL/pgSQL实现PL/iSQL的框架。实现流程图如下：





系统表的变化

下图描述了对PostgreSQL现有系统表的变化以及所做的补充。



数据库建模（第一章创建库+第二章创建表）

创建一个数据库

看看你能否访问数据库服务器的第一个例子就是试着创建一个数据库。一台运行着的IvorySQL服务器可以管理许多数据库。通常我们会为每个项目和每个用户单独使用一个数据库。

你的站点管理员可能已经为你创建了可以使用的数据库。如果这样你就可以省略这一步，并且跳到下一节。

要创建一个新的数据库，在我们这个例子里叫`mydb`，你可以使用下面的命令：

```
$ createdb mydb
```

如果不产生任何响应则表示该步骤成功，你可以跳过本节的剩余部分。

如果你看到类似下面这样的信息：

```
createdb: command not found
```

那么就是IvorySQL没有安装好。或者是根本没安装，或者是你的shell搜索路径没有设置正确。尝试用绝对路径调用该命令试试：

```
$ /usr/local/pgsql/bin/createdb mydb
```

在你的站点上这个路径可能不一样。和你的站点管理员联系或者看看安装指导获取正确的位置。

另外一种响应可能是这样：

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: No such
file or directory
        Is the server running locally and accepting connections on that socket?
```

这意味着该服务器没有启动，或者在`createdb`期望去连接它的时候没有在监听。同样，你也要查看安装指导或者咨询管理员。

另外一个响应可能是这样：

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
role "joe" does not exist
```

在这里提到了你自己的登录名。如果管理员没有为你创建IvorySQL用户帐号，就会发生这些现象。（IvorySQL用户帐号和操作系统用户帐号是不同的。） 如果你是管理员，参阅 第 22 章
获取创建用户帐号的帮助。你需要变成安装IvorySQL的操作系统用户的身份（通常是
postgres）才能创建第一个用户帐号。也有可能是赋予你的IvorySQL用户名和你的操作系统用户名不同；这种情况下，你需要使用`-U`选项或者使用`PGUSER`环境变量指定你的IvorySQL用户名。

如果你有个数据库用户帐号，但是没有创建数据库所需要的权限，那么你会看到下面的信息：

```
createdb: error: database creation failed: ERROR:  permission denied to create
database
```

并非所有用户都被许可创建新数据库。如果IvorySQL拒绝为你创建数据库，那么你需要让站点管理员赋予你创建数据库的权限。出现这种情况时请咨询你的站点管理员。如果你自己安装了IvorySQL，那么你应该以你启动数据库服务器的用户身份登录然后参考手册完成权限的赋予工作。http://www.postgresql.org/docs/17/tutorial-createdb.html#ftn.id-1.4.3.4.10.4[[1\]]

你还可以用其它名字创建数据库。IvorySQL允许你在一个站点上创建任意数量的数据库。数据库名必须是以字母开头并且小于63个字符长。一个方便的做法是创建和你当前用户名同名的数据库。许多工具假设该数据库名为缺省数据库名，所以这样可以节省你的敲键。要创建这样的数据库，只需要键入：

```
$ createdb
```

如果你再也不想使用你的数据库了，那么你可以删除它。比如，如果你是数据库`mydb`的所有人（创建人），那么你就可以用下面的命令删除它：

```
$ dropdb mydb
```

（对于这条命令而言，数据库名不是缺省的用户名，因此你就必须声明它）。这个动作将在物理上把所有与该数据库相关的文件都删除并且不可取消，因此做这中操作之前一定要考虑清楚。

更多关于 `createdb` 和 `dropdb` 的信息可以分别在 createdb和 dropdb中找到。

创建一个新表

你可以通过指定表的名字和所有列的名字及其类型来创建表：

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- 最低温度
    temp_hi         int,           -- 最高温度
    prcp            real,          -- 湿度
    date            date
);
```

你可以在 `psql` 输入这些命令以及换行符。`psql` 可以识别该命令直到分号才结束。

你可以在SQL命令中自由使用空白（即空格、制表符和换行符）。这就意味着你可以用和上面不同的对齐方式键入命令，或者将命令全部放在一行中。两个划线（“`--`”）引入注释。任何跟在它后面直到行尾的东西都会被忽略。SQL是对关键字和标识符大小写不敏感的语言，只有在标识符用双引号包围时才能保留它们的大小写（上例没有这么做）。

`varchar(80)` 指定了一个可以存储最长80个字符的任意字符串的数据类型。`int` 是普通的整数类型。`real` 是一种用于存储单精度浮点数的类型。`date` 类型应该可以自解释（没错，类型为`date`的列名字也是`date`。这么做可能比较方便或者容易让人混淆 — 你自己选择）。

IvorySQL支持标准的SQL类型`int`、`smallint`、`real`、`double precision`、`char(N)`、`varchar(N)`、`date`、`time`、`timestamp`和`interval`，还支持其他的通用功能的类型和丰富的几何类型。IvorySQL中可以定制任意数量的用户定义数据类型。因而类型名并不是语法关键字，除了SQL标准要求支持的特例外。

第二个例子将保存城市和它们相关的地理位置：

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

类型`point`就是一种IvorySQL特有数据类型的例子。

最后，我们还要提到如果你不再需要某个表，或者你想以不同的形式重建它，那么你可以用下面的命令删除它：

```
DROP TABLE tablename;
```

写入数据（SQL写入）参考第 6 章 数据操纵

当一个表被创建后，它不包含任何数据。在数据库发挥作用之前，首先要做的是插入数据。一次插入一行数据。你也可以在一个命令中插入多行，但不能插入不完整的行。即使只知道其中一些列的值，也必须创建完整的行。

要创建一个新行，使用 INSERT命令。这条命令要求提供表的名字和其中列的值。例如，考虑 第 5 章中的产品表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);

CREATE TABLE new_products (
    product_no int ,
    name varchar(255),
    price DECIMAL(10, 2),
    release_date DATE
);
```

一个插入一行的命令将是：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据的值是按照这些列在表中出现的顺序列出的，并且用逗号分隔。通常，数据的值是文字（常量），但也允许使用标量表达式。

上面的语法的缺点是你必须知道表中列的顺序。要避免这个问题，你也可以显式地列出列。例如，下面的两条命令都有和上文那条 命令一样的效果：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多用户认为明确列出列的名字是个好习惯。

如果你没有获得所有列的值，那么你可以省略其中的一些。在这种情况下，这些列将被填充为它们的缺省值。例如：

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是IvorySQL的一个扩展。它从使用给出的值从左开始填充列，有多少个给出的列值就填充多少个列，其他列的将使用缺省值。

为了保持清晰，你也可以显式地要求缺省值，用于单个的列或者用于整个行：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

你可以在一个命令中插入多行：

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);

INSERT INTO new_products (product_no, name, price, release_date) VALUES
    (1, 'A', 100.00, '2025-05-29'),
    (2, 'B', 150.50, '2024-11-20'),
    (3, 'C', 200.75, '2025-05-29');
```

也可以插入查询的结果（可能没有行、一行或多行）：

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = '2025-05-29';
```

这提供了用于计算要插入的行的SQL查询机制（第 7 章）的全部功能。

> 提示
>
> 在一次性插入大量数据时，考虑使用 COPY命令。它不如 INSERT命令那么灵活，但是更高效。

查询数据 参考 第七章查询的组合查询 第十五章 并行查询

组合查询

两个查询的结果可以用集合操作并、交、差进行组合。语法是:

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

`query1`*和`query2`*都是可以使用以上所有特性的查询。集合操作也可以嵌套和级连，例如:

```
query1 UNION query2 UNION query3
```

实际执行的是：

```
(query1 UNION query2) UNION query3
```

UNION`有效地把*`query2的结果附加到**query1***的结果上（不过我们不能保证这就是这些行实际被返回的顺序）。此外，它将删除结果中所有重复的行，就像`DISTINCT`做的那样，除非你使用了`UNION ALL`。

INTERSECT`返回那些同时存在于*`query1和**query2***的结果中的行，除非声明了`INTERSECT ALL`，否则所有重复行都被消除。

EXCEPT`返回所有在*`query1的结果中但是不在**query2**的结果中的行（有时侯这叫做两个查询的*差）。同样的，除非声明了`EXCEPT ALL`，否则所有重复行都被消除。

为了计算两个查询的并、交、差，这两个查询必须是“并操作兼容的”，也就意味着它们都返回同样数量的列，并且对应的列有兼容的数据类型，如 第 10.5 节中描述的那样。

并行查询

并行查询如何工作

当优化器判断对于某一个特定的查询，并行查询是最快的执行策略时，优化器将创建一个查询计划。该计划包括一个 *Gather*或者*Gather Merge*节点。下面是一个简单的例子：

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
-----------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

在所有的情形下，`Gather`或*Gather Merge*节点都只有一个子计划，它是将被并行执行的计划的一部分。如果`Gather`或*Gather Merge*节点位于计划树的最顶层，那么整个查询将并行执行。如果它位于计划树的其他位置，那么只有查询中在它之下的那一部分会并行执行。在上面的例子中，查询只访问了一个表，因此除`Gather`节点本身之外只有一个计划节点。因为该计划节点是`Gather`节点的孩子节点，所以它会并行执行。

使用 EXPLAIN命令，你能看到规划器选择的工作者数量。当查询执行期间到达`Gather`节点时，实现用户会话的进程将会请求和规划器选中的工作者数量一样多的 后台工作者进程。规划器将考虑使用的后台工作者的数量被限制为最多 max_parallel_workers_per_gather个。任何时候能够存在的后台工作者进程的总数由 max_worker_processes和 max_parallel_workers限制。因此，一个并行查询可能会使用比规划中少的工作者来运行，甚至有可能根本不使用工作者。最优的计划可能取决于可用的工作者的数量，因此这可能会导致不好的查询性能。如果这种情况经常发生，那么就应当考虑一下提高`max_worker_processes`和`max_parallel_workers`的值，这样更多的工作者可以同时运行；或者降低`max_parallel_workers_per_gather`，这样规划器会要求少一些的工作者。

为一个给定并行查询成功启动的后台工作者进程都将会执行计划的并行部分。这些工作者的领导者也将执行该计划，不过它还有一个额外的任务：它还必须读取所有由工作者产生的元组。当整个计划的并行部分只产生了少量元组时，领导者通常将表现为一个额外的加速查询执行的工作者。反过来，当计划的并行部分产生大量的元组时，领导者将几乎全用来读取由工作者产生的元组并且执行`Gather`或`Gather Merge`节点上层计划节点所要求的任何进一步处理。在这些情况下，领导者所作的执行并行部分的工作将会很少。

当计划的并行部分的顶层节点是`Gather

Merge`而不是`Gather`时，它表示每个执行计划并行部分的进程会产生有序的元组，并且领导者执行一种保持顺序的合并。相反，`Gather`会以任何方便的顺序从工作者读取元组，这会破坏可能已经存在的排序顺序。

何时会用到并行查询?

有几种设置会导致查询规划器在任何情况下都不生成并行查询计划。为了让并行查询计划能够被生成，必须配置好下列设置。

- max_parallel_workers_per_gather必须被设置为大于零的值。这是一种特殊情况，更加普遍的原则是所用的工作者数量不能超过`max_parallel_workers_per_gather`所配置的数量。

此外，系统一定不能运行在单用户模式下。因为在单用户模式下，整个数据库系统运行在单个进程中，没有后台工作者进程可用。

如果下面的任一条件为真，即便对一个给定查询通常可以产生并行查询计划，规划器都不会为它产生并行查询计划：

- 查询要写任何数据或者锁定任何数据库行。如果一个查询在顶层或者CTE中包含了数据修改操作，那么不会为该查询产生并行计划。一种例外是，**CREATE TABLE … AS**、`SELECT INTO`以及`CREATE MATERIALIZED VIEW`这些创建新表并填充它的命令可以使用并行计划。
- 查询可能在执行过程中被暂停。只要在系统认为可能发生部分或者增量式执行，就不会产生并行计划。例如：用 DECLARE CURSOR创建的游标将永远不会使用并行计划。类似地，一个`FOR x IN query LOOP .. END LOOP`形式的 PL/pgSQL
循环也永远不会使用并行计划，因为当并行查询进行时，并行查询系统无法验证循环中的代码执行起来是安全的。
- 使用了任何被标记为`PARALLEL UNSAFE`的函数的查询。大多数系统定义的函数都被标记为`PARALLEL SAFE`，但是用户定义的函数默认被标记为`PARALLEL UNSAFE`。参见 第 15.4 节中的讨论。
- 该查询运行在另一个已经存在的并行查询内部。例如，如果一个被并行查询调用的函数自己发出一个SQL
查询，那么该查询将不会使用并行计划。这是当前实现的一个限制，但是或许不值得移除这个限制，因为它会导致单个查询使用大量的进程。

即使对于一个特定的查询已经产生了并行查询计划，在一些情况下执行时也不会并行执行该计划。如果发生这种情况，那么领导者将会自己执行该计划在`Gather`节点之下的部分，就好像`Gather`节点不存在一样。上述情况将在满足下面的任一条件时发生：

- 因为后台工作者进程的总数不能超过 max_worker_processes，导致不能得到后台工作者进程。
- 由于为并行查询目的启动的后台工作者数量不能超过
max_parallel_workers这一限制而不能得到后台工作者。
- 客户端发送了一个执行消息，并且消息中要求取元组的数量不为零。执行消息可见
扩展查询协议中的讨论。因为
libpq当前没有提供方法来发送这种消息，所以这种情况只可能发生在不依赖 libpq
的客户端中。如果这种情况经常发生，那在它可能发生的会话中设置
max_parallel_workers_per_gather为零是一个很好的主意，这样可以避免产生连续运行时次优的查询计划。

并行计划

因为每个工作者只执行完成计划的并行部分，所以不可能简单地产生一个普通查询计划并使用多个工作者运行它。每个工作者都会产生输出结果集的一个完全，因而查询并不会比普通查询运行得更快甚至还会产生不正确的结果。相反，计划的并行部分一定被查询优化器在内部当作一个*部分计划*，即它必须被构建出来，这样每一个执行该计划的进程将以无重复地方式产生输出行的一个子集，即保证每一个所需要的输出行正好只被一个合作进程生成。通常，这意味着该查询的驱动表上的扫描必须是一种可并行的扫描。

并行扫描

当前支持下列可并行的表扫描。

- 在一个*并行顺序扫描*中，表块将在合作进程之间被划分。一次会分发一个块，这样对表的访问还是保持顺序方式。
- 在一个*并行位图堆扫描*中，一个进程被选为领导者。这个进程执行对一个或者多个索引的扫描并且构建出一个位图指示需要访问哪些表块。这些表块接着会在合作进程之间划分（和并行顺序扫描中一样）。换句话说，堆扫描以并行方式进行但底层的索引扫描不是并行。
- 在一个*并行索引扫描*或者*并行只用索引的扫描*中，合作进程轮流从索引读取数据。当前，并行索引扫描仅有B-
  树索引支持。每一个进程将认领一个索引块并且扫描和返回该索引块引用的所有元组，其他进程可以同时地从一个不同的索引块返回元组。并行B-树扫描的结果会以每个工作者进程内的顺序返回。

其他扫描类型（例如非B-树索引的扫描）可能会在未来支持并行扫描。

并行连接

正如在非并行计划中那样，驱动表可能被使用嵌套循环、哈希连接或者归并连接连接到一个或者多个其他表。连接的内侧可以是任何类型的被规划器支持的非并行计划，假设它能够安全地在并行工作者中运行。根据连接类型，内侧还可以是一种并行计划。

- 在一个*嵌套循环连接*中，内侧总是非并行的。尽管它会被完全执行，如果内侧是一个索引扫描也会很高效，因为外侧元组以及在索引中查找值的循环会被划分到多个合作进程。
- 在一个*归并连接*中，内侧总是一个非并行计划并且因此会被完全执行。这可能是不太高效的，特别是在排序必须被执行时，因为在每一个合作进程中工作数据和结果数据是重复的。
- 在一个哈希连接\（没有“并行”前缀）中，每个合作进程都会完全执行内侧以构建哈希表的相同。如果哈希表很大或者该计划开销很大，这种方式就很低效。在一个并行哈希连接*中，内侧是一个*并行哈希\，它把构建共享哈希表的工作划分到多个合作进程。

并行聚集

IvorySQL通过按两个阶段进行聚集来支持并行聚集。首先，每个参与到查询并行部分的进程执行一个聚集步骤，为该进程注意到的每个分组产生一个部分结果。这在计划中反映为一个`Partial Aggregate`节点。然后，部分结果通过`Gather`或者`Gather Merge`被传输到领导者。最后，领导者对来自所有工作者的结果进行重新聚集得到最终的结果。这在计划中反映为一个`Finalize Aggregate`节点。

因为`Finalize Aggregate`节点运行在领导者进程上，如果查询产生的分组数相对于其输入行数来说比较大，则查询规划器不会喜欢它。例如，在最坏的情况下，`Finalize Aggregate`节点看到的分组数可能与所有工作者进程在`Partial Aggregate`阶段看到的输入行数一样多。对于这类情况，使用并行聚集显然得不到性能收益。查询规划器会在规划过程中考虑这一点并且不太会在这种情况下选择并行聚集。

并行聚集并非在所有情况下都被支持。每一个聚集都必须是对并行安全的并且必须有一个组合函数。如果该聚集有一个类型为`internal`的转移状态，它必须有序列化和反序列化函数。更多细节请参考 CREATE AGGREGATE。如果任何聚集函数调用包含`DISTINCT`或`ORDER BY`子句，则不支持并行聚集。对于有序集聚集或者当查询涉及`GROUPING SETS`时，也不支持并行聚集。只有在查询中涉及的所有连接也是该计划并行部分的组成部分时，才能使用并行聚集。

并行Append

只要当IvorySQL需要从多个源中整合行到一个单一结果集时，它会使用`Append`或`MergeAppend`计划节点。在实现`UNION ALL`或扫描分区表时常常会发生这种情况。就像这些节点可以被用在任何其他计划中一样，它们可以被用在并行计划中。不过，在并行计划中，规划器使用的是`Parallel Append`节点。

当一个`Append`节点被用在并行计划中时，每个进程将按照子计划出现的顺序执行子计划，这样所有的参与进程会合作执行第一个子计划直到它被完成，然后同时移动到第二个计划。而在使用`Parallel Append`时，执行器将把它的子计划尽可能均匀地散布在参与进程中，这样多个子计划会被同时执行。这避免了竞争，也避免了子计划在那些不执行它的进程中产生启动代价。

此外，和常规的`Append`节点不同（在并行计划中使用时仅有部分子计划），`Parallel`

Append`节点既可以有部分子计划也可以有非部分子计划。非部分子计划将仅被单个进程扫描，因为扫描它们不止一次会产生重复的结果。因此涉及到追加多个结果集的计划即使在没有有效的部分计划可用时，也能实现粗粒度的并行。例如，考虑一个针对分区表的查询，它只能通过使用一个不支持并行扫描的索引来实现。规划器可能会选择常规`Index Scan`计划的`Parallel Append。每个索引扫描必须被单一的进程执行完，但不同的扫描可以由不同的进程同时执行。

enable_parallel_append可以被用来禁用这种特性。

并行计划小贴士

如果我们想要一个查询能产生并行计划但事实上又没有产生，可以尝试减小parallel_setup_cost或者parallel_tuple_cost。当然，这个计划可能比规划器优先产生的顺序计划还要慢，但也不总是如此。如果将这些设置为很小的值（例如把它们设置为零）也不能得到并行计划，那就可能是有某种原因导致查询规划器无法为你的查询产生并行计划。可能的原因可见 第 15.2 节和 第 15.4 节。

在执行一个并行计划时，可以用`EXPLAIN (ANALYZE,VERBOSE)`来显示每个计划节点在每个工作者上的统计信息。这些信息有助于确定是否所有的工作被均匀地分发到所有计划节点以及从总体上理解计划的性能特点。

事务（参考Sql命令）

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

描述

ABORT`回滚当前事务并且导致由该事务所作的所有更新被丢弃。这个命令的行为与标准SQL命令`ROLLBACK的行为一样，并且只是为了历史原因存在。

参数

- WORK TRANSACTION

  可选关键词。它们没有效果。

- AND CHAIN

  如果规定了`AND CHAIN` ，新事务立即启动，具有与刚刚完成的事务相同的事务特征(参见http://www.postgresql.org/docs/17/sql-set-transaction.html[`SET TRANSACTION`])。否则，不会启动新事务。

注解

使用 COMMIT成功地终止一个事务。

在一个事务块之外发出`ABORT`会发出一个警告消息并且不会产生效果。

例子

中止所有更改：

```
ABORT;
```

兼容性

这个命令是一个因为历史原因而存在的IvorySQL扩展。`ROLLBACK`是等效的标准SQL命令。

BEGIN — 开始一个事务块

大纲

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]

其中 transaction_mode 是以下之一:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

`BEGIN`开始一个事务块，也就是说所有`BEGIN`命令之后的所有语句将被在一个事务中执行，直到给出一个显式的`COMMIT`或者`ROLLBACK`。默认情况下（没有`BEGIN`），IvorySQL在“自动提交”模式中执行事务，也就是说每个语句都在自己的事务中执行并且在语句结束时隐式地执行一次提交（如果执行成功，否则会完成一次回滚）。

在一个事务块内的语句会执行得更快，因为事务的开始/提交也要求可观的CPU和磁盘活动。在进行多个相关更改时，在一个事务内执行多个语句也有助于保证一致性：在所有相关更新还没有完成之前，其他会话将不能看到中间状态。

如果指定了隔离级别、读/写模式或者延迟模式，新事务也会有那些特性，就像执行了`SET TRANSACTION`一样。

参数

- `WORK``TRANSACTION`

  可选的关键词。它们没有效果。

这个语句其他参数的含义请参考 SET TRANSACTION。

注解

`START TRANSACTION`具有和`BEGIN` 相同的功能。

使用`COMMIT`或者`ROLLBACK`来终止一个事务块。

在已经在一个事务块中时发出`BEGIN`将惹出一个警告消息。事务状态不会被影响。要在一个事务块中嵌套事务，可以使用保存点（见 SAVEPOINT）。

由于向后兼容的原因，连续的`transaction_modes`之间的逗号可以被省略。

示例

开始一个事务块：

```
BEGIN;
```

兼容性

`BEGIN` 是一种IvorySQL语言扩展。它等效于SQL标准的命令 `START TRANSACTION`，它的参考页包含额外的兼容性信息。

`DEFERRABLE transaction_mode` 是一种IvorySQL语言扩展。

附带地，`BEGIN`关键词被用于嵌入式SQL中的一种不同目的。在移植数据库应用时，我们建议小心对待事务语义。

COMMIT — 提交当前事务

大纲

`COMMIT`提交当前事务。所有由该事务所作的更改会变得对他人可见并且被保证在崩溃发生时仍能持久。

参数

`WORK TRANSACTION`
: 可选的关键词。它们没有效果。

`AND CHAIN`
: 如果指定了 `AND CHAIN`，则立即启动与刚刚完成的事务具有相同事务特征（参见 SET TRANSACTION）的新事务。否则，没有新事务被启动。

注解

使用 ROLLBACK中止一个事务。

当不在一个事务内时发出 `COMMIT` 不会产生危害，但是它会产生一个警告消息。当`COMMIT AND CHAIN`不在事务内时是一个错误。

示例

要提交当前事务并且让所有更改持久化：

```
COMMIT;
```

兼容性

命令 `COMMIT`符合SQL标准。表单`COMMIT TRANSACTION`为IvorySQL扩展。

COMMIT PREPARED — 提交一个早前为两阶段提交预备的事务

大纲

```
COMMIT PREPARED transaction_id
```

描述

`COMMIT PREPARED`提交一个处于预备状态的事务。

参数

transaction_id
要被提交的事务的事务标识符。

注解

要提交一个预备的事务，你必须是原先执行该事务的同一用户或者超级用户。但是不需要处于执行该事务的同一会话中。

这个命令不能在一个事务块中执行。该预备事务将被立刻提交。

pg_prepared_xacts系统视图中列出了所有当前可用的预备事务。

例子

提交由事务标识符`foobar`标识的事务：

```
COMMIT PREPARED 'foobar';
```

兼容性

COMMIT PREPARED 是一种IvorySQL扩展。其意图是用于外部事务管理系统，其中有些已经被标准涵盖（例如X/Open XA），但是那些系统的SQL方面未被标准化。

END - 提交当前事务

大纲

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

描述

END
提交当前事务。所有该事务做的更改便得对他人可见并且被保证发生崩溃时仍然是持久的。这个命令是一种IvorySQL扩展，它等效于 **COMMIT**。

参数

WORK TRANSACTION
可选关键词，它们没有效果。

AND CHAIN
如果规定了`AND CHAIN`，则立即启动与刚完成事务具有相同事务特征(参见 SET TRANSACTION)的新事务。否则，没有新事务被启动。

注解

使用 **ROLLBACK**可以中止一个事务。

当不在一个事务中时发出`END`没有危害，但是会产生一个警告消息。

示例

要提交当前事务并且让所有更改持久化：

```
END;
```

兼容性

END 是一种IvorySQL扩展，它提供和 **COMMIT**等效的功能，后者在SQL标准中指定。

PREPARE TRANSACTION — 为两阶段提交准备当前事务

大纲

```
PREPARE TRANSACTION transaction_id
```

描述

`PREPARE TRANSACTION`为两阶段提交准备当前事务。在这个命令之后，该事务不再与当前会话关联。相反，它的状态被完全存储在磁盘上，并且有很高的可能性它会被提交成功（即便在请求提交前发生数据库崩溃）。

一旦被准备好，事务稍后就可以分别用 **COMMIT PREPARED**或者 **ROLLBACK PREPARED**提交或者回滚。可以从任何会话而不仅仅是执行原始事务的会话中发出这些命令。

从发出命令的会话的角度来看，`PREPARE TRANSACTION`不像`ROLLBACK`命令：在执行它之后，就没有活跃的当前事务，并且该预备事务的效果也不再可见（如果该事务被提交，效果将重新变得可见）。

如果由于任何原因`PREPARE TRANSACTION`命令失败，它会变成一个`ROLLBACK`：当前事务会被取消。

参数

transaction_id

一个任意的事务标识符，`COMMIT PREPARED`或者`ROLLBACK PREPARED`以后将用这个标识符来标识这个事务。该标识符必须写成一个字符串,并且长度必须小于200字节。它也不能与任何当前已经准备好的事务的标识符相同。

注解

PREPARE TRANSACTION`并不是设计为在应用或者交互式会话中使用。它的目的是允许一个外部事务管理器在多个数据库或者其他事务性来源之间执行原子的全局事务。除非你在编写一个事务管理器，否则你可能不会用到`PREPARE TRANSACTION。

这个命令必须在一个事务块中使用。事务块用 **BEGIN**开始。

当前在已经执行过任何涉及到临时表或者会话的临时命名空间、创建带`WITH HOLD`的游标或者执行`LISTEN`、`UNLISTEN`或`NOTIFY`的事务中，不允许`PREPARE`该事务。这些特性与当前会话绑定得太过紧密，所以对一个要被准备的事务来说没有什么用处。

如果用`SET`（不带`LOCAL`选项）修改过事务的任何运行时参数，这些效果会持续到`PREPARE TRANSACTION`之后，并且将不会被后续的任何`COMMIT PREPARED`或`ROLLBACK PREPARED`所影响。因此，在这一 方面`PREPARE TRANSACTION`的行为更像`COMMIT`而不是`ROLLBACK`。

所有当前可用的准备好事务被列在 **pg_prepared_xacts**系统视图中。

小心

让一个事务处于准备好状态太久是不明智的。这将会干扰`VACUUM`回收存储的能力，并且在极限情况下可能导致数据库关闭以阻止事务ID回卷（见 第 25.1.5 节）。还要记住，该事务会继续持有它已经持有的锁。该特性的设计用法是，只要一个外部事务管理器已经验证其他数据库也准备好了要提交，一个准备好的事务将被正常地提交或者回滚。

如果没有建立一个外部事务管理器来跟踪准备好的事务并且确保它们被迅速地结束，最好禁用准备好事务特性（设置 max_prepared_transactions为零）。这将防止意外地创建准备好事务，不然该事务有可能被忘记并且最终导致问题。

例子

为两阶段提交准备当前事务，使用`foobar`作为事务标识符：

```
PREPARE TRANSACTION 'foobar';
```

兼容性

`PREPARE TRANSACTION`是一种IvorySQL扩展。其意图是用于外部事务管理系统，其中有些已经被标准涵盖（例如X/Open XA），但是那些系统的SQL方面未被标准化。

ROLLBACK — 中止当前事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

描述

`ROLLBACK`回滚当前事务并且导致该事务所作的所有更新都被抛弃。

参数

WORK TRANSACTION
可选关键词，没有效果。

AND CHAIN
如果指定了`AND CHAIN`，则立即启动与刚刚完成事务具有相同事务特征（参见 SET TRANSACTION）的新事务。否则，不会启动任何新事务。

注解

使用 **COMMIT**可成功地终止一个事务。

在一个事务块之外发出`ROLLBACK`会发出一个警告并且不会有效果。事务块之外的`ROLLBACK AND CHAIN`是一个错误。

示例

要中止所有更改：

```
ROLLBACK;
```

兼容性

命令`ROLLBACK`符合SQL标准。窗体`ROLLBACK TRANSACTION`是一个IvorySQL扩展。

ROLLBACK PREPARED — 取消一个之前为两阶段提交准备好的事务

大纲

```
ROLLBACK PREPARED transaction_id
```

描述

`ROLLBACK PREPARED`回滚一个处于准备好状态的事务。

参数

`transaction_id`
要被回滚的事务的事务标识符。

注解

要回滚一个准备好的事务，你必须是原先执行该事务的同一个用户或者是一个超级用户。但是你必须处在执行该事务的同一个会话中。

这个命令不能在一个事务块内被执行。准备好的事务会被立刻回滚。

`pg_prepared_xacts`系统视图中列出了当前可用的所有准备好的事务。

例子

用事务标识符`foobar`回滚对应的事务：

```
ROLLBACK PREPARED 'foobar';
```

兼容性

`ROLLBACK PREPARED`是一种IvorySQL扩展。其意图是用于外部事务管理系统，其中有些已经被标准涵盖（例如X/Open XA），但是那些系统的SQL方面未被标准化。

SAVEPOINT — 在当前事务中定义一个新的保存点

大纲

```
SAVEPOINT savepoint_name
```

描述

`SAVEPOINT`在当前事务中建立一个新保存点。

保存点是事务内的一种特殊标记，它允许所有在它被建立之后执行的命令被回滚，把该事务的状态恢复到它处于保存点时的样子。

参数

savepoint_name
给新保存点的名字。

注解

使用 ROLLBACK TO回滚到一个保存点。使用 RELEASE SAVEPOINT销毁一个保存点，但保持在它被建立之后执行的命令的效果。

保存点只能在一个事务块内建立。可以在一个事务内定义多个保存点。

示例

要建立一个保存点并且后来撤销在它建立之后执行的所有命令的效果：

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入值 1 和 3，但不会插入 2。

要建立并且稍后销毁一个保存点：

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入 3 和 4。

兼容性

当建立另一个同名保存点时，SQL要求之前的那个保存点自动被销毁。在IvorySQL中，旧的保存点会被保留，不过在进行回滚或释放时只能使用最近的那一个（用`RELEASE SAVEPOINT`释放较新的保存点将会导致较旧的保存点再次变得可以被`ROLLBACK TO SAVEPOINT`和`RELEASE SAVEPOINT`访问）。在其他方面，`SAVEPOINT`完全符合SQL。

SET CONSTRAINTS — 为当前事务设置约束检查时机

大纲

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

描述

`SET

CONSTRAINTS`设置当前事务内约束检查的行为。`IMMEDIATE`约束在每个语句结束时被检查。`DEFERRED`约束直到事务提交时才被检查。每个约束都有自己的`IMMEDIATE`或`DEFERRED`模式。

在创建时，一个约束会被给定三种特性之一：**DEFERRABLE INITIALLY DEFERRED**、**DEFERRABLE INITIALLY IMMEDIATE`或者`NOT DEFERRABLE**。第三类总是`IMMEDIATE`并且不会受到`SET CONSTRAINTS`命令的影响。前两类在每个事务开始时都处于指定的模式，但是它们的行为可以在一个事务内用`SET CONSTRAINTS`更改。

带有一个约束名称列表的`SET CONSTRAINTS`只更改那些约束（都必须是可延迟的）的模式。每一个约束名称都可以是模式限定的。如果没有指定模式名称，则当前的模式搜索路径将被用来寻找第一个匹配的名称。`SET CONSTRAINTS ALL`更改所有可延迟约束的模式。

当`SET CONSTRAINTS`把一个约束的模式从`DEFERRED`改成`IMMEDIATE`时，新模式会有追溯效果：任何还没有解决的数据修改（本来会在事务结束时被检查）会转而在`SET CONSTRAINTS`命令的执行期间被检查。如果任何这种约束被违背，**SET CONSTRAINTS`将会失败（并且不会改变该约束模式）。这样，`SET CONSTRAINTS**可以被用来在一个事务中的特定点强制进行约束检查。

当前，只有`UNIQUE`、**PRIMARY KEY**、**REFERENCES**（外键）以及`EXCLUDE`约束受到这个设置的影响。`NOT NULL`以及`CHECK`约束总是在一行被插入或修改时立即检查（不是在语句结束时）。没有被声明为`DEFERRABLE`的唯一和排除约束也会被立刻检查。

被声明为“约束触发器”的触发器的引发也受到这个设置的控制 — 它们会在相关约束被检查的同时被引发。

注解

因为IvorySQL并不要求约束名称在模式内唯一（但是在表内要求唯一），可能有多于一个约束匹配指定的约束名称。在这种情况下`SET CONSTRAINTS`将会在所有的匹配上操作。对于一个非模式限定的名称，一旦在搜索路径中的某个模式中发现一个或者多个匹配，路径中后面的模式将不会被搜索。

这个命令只修改当前事务内约束的行为。在事务块外部发出这个命令会产生一个警告并且也不会有任何效果。

兼容性

这个命令符合SQL标准中定义的行为，但有一点限制：在 IvorySQL中，它不会应用在`NOT NULL`和`CHECK`约束上。还有，IvorySQL会立刻检查非可延迟的唯一约束，而不是按照标准建议的在语句结束时检查。

SET TRANSACTION — 设置当前事务的特性

大纲

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

其中transaction_mode是下列之一:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

`SET TRANSACTION`命令设置当前会话的特性。`SET SESSION CHARACTERISTICS`设置一个会话后续事务的默认事务特性。在个体事务中可以用`SET TRANSACTION`覆盖这些默认值。

可用的事务特性是事务隔离级别、事务访问模式（读/写或只读）以及可延迟模式。此外，可以选择一个快照，不过只能用于当前事务而不能作为会话默认值。

一个事务的隔离级别决定当其他事务并行运行时该事务能看见什么数据：

READ COMMITTED
一个语句只能看到在它开始前提交的行。这是默认值。

REPEATABLE READ
当前事务的所有语句只能看到这个事务中执行的第一个查询或者数据修改语句之前提交的行。

SERIALIZABLE
当前事务的所有语句只能看到这个事务中执行的第一个查询或者数据修改语句之前提交的行。如果并发的可序列化事务间的读写模式可能导致一种那些事务串行（一次一个）执行时不可能出现的情况，其中之一将会被回滚并且得到一个`serialization_failure`错误。

SQL标准定义了一种额外的级别：**READ UNCOMMITTED**。在IvorySQL中`READ UNCOMMITTED`被视作`READ COMMITTED`。

一个事务执行了第一个查询或者数据修改语句（**SELECT**、**INSERT**、**DELETE**、**UPDATE**、**FETCH`或`COPY**）之后就无法更改事务隔离级别。更多有关事务隔离级别和并发控制的信息可见 第 13 章。

事务的访问模式决定该事务是否为读/写或者只读。读/写是默认值。当一个事务为只读时，如果SQL命令`INSERT`、**UPDATE**、**DELETE`和`COPY FROM`要写的表不是一个临时表，则它们不被允许。不允许`CREATE**、**ALTER`以及`DROP`命令。不允许`COMMENT**、**GRANT**、**REVOKE**、**TRUNCATE**。如果`EXPLAIN ANALYZE`和`EXECUTE`要执行的命令是上述命令之一，则它们也不被允许。这是一种高层的只读概念，它不能阻止所有对磁盘的写入。

只有事务也是`SERIALIZABLE`以及`READ ONLY`时，`DEFERRABLE`事务属性才会有效。当一个事务的所有这三个属性都被选择时，该事务在第一次获取其快照时可能会阻塞，在那之后它运行时就不会有`SERIALIZABLE`事务的开销并且不会有任何牺牲或者被一次序列化失败取消的风险。这种模式很适合于长时间运行的报表或者备份。

SET TRANSACTION SNAPSHOT`命令允许新的事务使用与一个现有事务相同的*快照*运行。已经存在的事务必须已经把它的快照用`pg_export_snapshot 函数（见 第 9.27.5 节）导出。该函数会返回一个快照标识符，**SET TRANSACTION SNAPSHOT`需要被给定一个快照标识符来指定要导入的快照。在这个命令中该标识符必须被写成一个字符串，例如’ 000003A1-1'。SET TRANSACTION SNAPSHOT`只能在一个事务的开始执行，并且要在该事务的第一个查询或者数据修改语句（`SELECT**、**INSERT**、**DELETE**、**UPDATE**、**FETCH`或`COPY**）之前执行。此外，该事务必须已经被设置为`SERIALIZABLE`或者`REPEATABLE READ`隔离级别（否则，该快照将被立刻抛弃，因为`READ COMMITTED`模式会为每一个命令取一个新快照）。如果导入事务使用了`SERIALIZABLE`隔离级别，那么导入快照的事务必须也使用该隔离级别。还有，一个非只读可序列化事务不能导入来自只读事务的快照。

注解

如果执行`SET TRANSACTION`之前没有`START TRANSACTION`或者`BEGIN`，它会发出一个警告并且不会有任何效果。

可以通过在`BEGIN`或者`START TRANSACTION`中指定想要的`transaction_modes**来省掉`SET TRANSACTION**。但是在`SET TRANSACTION SNAPSHOT`中该选项不可用。

会话默认的事务模式也可以通过配置参数 default_transaction_isolation、http://www.postgresql.org/docs/17/runtime-config-client.html#GUC-DEFAULT-TRANSACTION-READ-ONLY[default_transaction_read_only]和

default_transaction_deferrable来设置或检查（实际上`SET SESSION CHARACTERISTICS`只是用`SET`设置这些变量的等效体）。这意味着可以通过配置文件、ALTER DATABASE等方式设置默认值。详见 第 20 章。

当前事务的模式可以类似的通过配置参数 transaction_isolation、 transaction_read_only、和 transaction_deferrable来设置或检查。设置这其中一个参数的作用与相应的`SET TRANSACTION`选项相同，在它何时可以完成方面，也有相同的限制。但是，这些参数不能在配置文件中设置，或者从活动SQL以外的任何来源来设置。

示例

要用一个已经存在的事务的同一快照开始一个新事务，首先要从该现有事务导出快照。这将会返回快照标识符，例如：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

然后在一个新开始的事务的开头把该快照标识符用在一个`SET TRANSACTION SNAPSHOT`命令中：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

兼容性

SQL标准中定义了这些命令，不过`DEFERRABLE`事务模式和`SET TRANSACTION SNAPSHOT`形式除外，这两者是IvorySQL扩展。

SERIALIZABLE`是标准中默认的事务隔离级别。在IvorySQL中默认值是普通的`READ COMMITTED，但是你可以按上述的方式更改。

在SQL标准中，可以用这些命令设置一个其他的事务特性：诊断区域的尺寸。这个概念与嵌入式SQL有关，并且因此没有在IvorySQL服务器中实现。

SQL标准要求连续的 transaction_modes 之间有逗号，但是出于历史原因IvorySQL允许省略逗号。

START TRANSACTION — 开始一个事务块

大纲

```
START TRANSACTION [ transaction_mode [, ...] ]

其中 transaction_mode 是下列之一：

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

描述

这个命令开始一个新的事务块。如果指定了隔离级别、读写模式或者可延迟模式，新的事务将会具有这些特性，就像执行了 `SET TRANSACTION`一样。这和 `BEGIN`命令一样。

参数

这些参数对于这个语句的含义可参考 SET TRANSACTION。

兼容性

在标准中，没有必要发出`START TRANSACTION`来开始一个事务块：任何SQL命令会隐式地开始一个块。IvorySQL的行为可以被视作在每个命令之后隐式地发出一个没有跟随在`START TRANSACTION`（或者`BEGIN`）之后的`COMMIT`并且因此通常被称作“自动提交”。为了方便，其他关系型数据库系统也可能会提供自动提交特性。

`DEFERRABLE transaction_mode` 是一种IvorySQL语言扩展。

SQL标准要求在连续的 `transaction_modes` 之间有逗号，但是由于历史原因IvorySQL允许省略逗号。

Sql参考（第 4 章 SQL语法）

词法结构

SQL输入由一个 命令 序列组成。一个命令由一个 记号 的序列构成，并由一个分号（“;”）终结。输入流的末端也会标志一个命令的结束。具体哪些记号是合法的与具体命令的语法有关。

一个记号可以是一个 关键词、一个 标识符、一个 带引号的标识符、一个 literal（或常量）或者一个特殊字符符号。记号通常以空白（空格、制表符、新行）来分隔，但在无歧义时并不强制要求如此（唯一的例子是一个特殊字符紧挨着其他记号）。

例如，下面是一个（语法上）合法的SQL输入：

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

这是一个由三个命令组成的序列，每一行一个命令（尽管这不是必须地，在同一行中可以有超过一个命令，而且命令还可以被跨行分割）。

另外，注释 也可以出现在SQL输入中。它们不是记号，它们和空白完全一样。

根据标识命令、操作符、参数的记号不同，SQL的语法不很一致。最前面的一些记号通常是命令名，因此在上面的例子中我们通常会说一个“SELECT”、一个“UPDATE”和一个“INSERT”命令。但是例如`UPDATE`命令总是要求一个`SET`记号出现在一个特定位置，而`INSERT`则要求一个`VALUES`来完成命令。每个命令的精确语法规则在 第 VI 部分中介绍。

标识符和关键词

上例中的`SELECT`、`UPDATE`或`VALUES`记号是 关键词 的例子，即SQL语言中具有特定意义的词。记号`MY_TABLE`和`A`则是 标识符 的例子。它们标识表、列或者其他数据库对象的名字，取决于使用它们的命令。因此它们有时也被简称为“名字”。关键词和标识符具有相同的词法结构，这意味着我们无法在没有语言知识的前提下区分一个标识符和关键词。一个关键词的完整列表可以在 附录 C中找到。

SQL标识符和关键词必须以一个字母（`a-z`，也可以是带变音符的字母和非拉丁字母）或一个下划线（`_`）开始。后续字符可以是字母、下划线（`_`）、数字（`0-9`）或美元符号（`$`）。注意根据SQL标准的字母规定，美元符号是不允许出现在标识符中的，因此它们的使用可能会降低应用

的可移植性。SQL标准不会定义包含数字或者以下划线开头或结尾的关键词，因此这种形式的标识符不会与未来可能的标准扩展冲突。

系统中一个标识符的长度不能超过`NAMEDATALEN`-1字节，在命令中可以写超过此长度的标识符，但是它们会被截断。默认情况下，`NAMEDATALEN`的值为64，因此标识符的长度上限为63字节。如果这个限制有问题，可以在`src/include/pg_config_manual.h`中修改`NAMEDATALEN`常量。

关键词和不被引号修饰的标识符是大小写不敏感的。因此：

```
UPDATE MY_TABLE SET A = 5;
```

可以等价地写成：

```
uPDaTE my_TabLE SeT a = 5;
```

一个常见的习惯是将关键词写成大写，而名称写成小写，例如：

```
UPDATE my_table SET a = 5;
```

这里还有第二种形式的标识符：受限标识符*或*被引号修饰的标识符。它是由双引号（"）包围的一个任意字符序列。一个受限标识符总是一个标识符而不会是一个关键字。因此`"select"`可以用于引用一个名为“select”的列或者表，而一个没有引号修饰的`select`则会被当作一个关键词，从而在本应使用表或列名的地方引起解析错误。在上例中使用受限标识符的例子如下：

```
UPDATE "my_table" SET "a" = 5;
```

受限标识符可以包含任何字符，除了代码为0的字符（如果要包含一个双引号，则写两个双引号）。这使得可以构建原本不被允许的表或列的名称，例如包含空格或花号的名字。但是长度限制依然有效。

引用标识符也使其区分大小写，而未引用的名称总是折叠成小写。例如，标识符`FOO`、**foo`和`"foo"`在IvorySQL中被认为是相同的，但是`"Foo"`和`"FOO"`与这三个不同，并且彼此不同。（在IvorySQL中，将不带引号的名称折叠为小写与SQL标准不兼容，SQL标准规定不带引号的名称应折叠为大写。因此，根据标准，`foo`应等同于`"FOO"`而不是`"foo"**。如果您想编写可移植应用程序，建议您始终引用某个特定的名称，或者永远不要引用它。）

一种受限标识符的变体允许包括转义的用代码点标识的Unicode字符。这种变体以`U&`（大写或小写U跟上一个花号）开始，后面紧跟双引号修饰的名称，两者之间没有任何空白，如`U&"foo"`（注意这里与操作符`&`似乎有一些混淆，但是在`&`操作符周围使用空白避免了这个问题）。在引号内，Unicode字符可以以转义的形式指定：反斜线接上4位16进制代码点号码或者反斜线和加号接上6位16进制代码点号码。例如，标识符`"data"`可以写成：

```
U&"d\0061t\+000061"
```

下面的例子用斯拉夫语字母写出了俄语单词“slon”（大象）：

```
U&"\0441\043B\043E\043D"
```

如果希望使用其他转义字符来代替反斜线，可以在字符串后使用`UESCAPE`子句，例如：

```
U&"d!0061t!+000061" UESCAPE '!'
```

转义字符可以是除了16进制位、加号、单引号、双引号、空白字符之外的任意单个字符。请注意，转义字符在 UESCAPE 之后用单引号而不是双引号书写。

为了在标识符中包括转义字符本身，将其写两次即可。

4位或6位转义形式都可以被用来定义UTF-16代理对来组成代码点大于U+FFFF的字符，尽管6位形式的存在使得这种做法变得不必要（代理对并不被直接存储，而是绑定成一个单独的代码点）。

如果服务器编码不是UTF-8，则由其中一个转义序列标识的Unicode代码点转换为实际的服务器编码；如果不可能，则报告错误。

常量

在IvorySQL中有三种隐式类型常量：字符串、位串和数字。常量也可以被指定显式类型，这可以使得它被更精确地展示以及更有效地处理。这些选择将会在后续小节中讨论。

字符串常量

在SQL中，一个字符串常量是一个由单引号（ ' ）包围的任意字符序列，例如’ This is a string'`。为了在一个字符串中包括一个单引号，可以写两个相连的单引号，例如’ Dianne''s horse'`。注意这和一个双引号（"）不同。

两个只由空白及至少一个新行分隔的字符串常量会被连接在一起，并且将作为一个写在一起的字符串常量来对待。例如：

```
SELECT 'foo'
'bar';
```

等同于：

```
SELECT 'foobar';
```

但是：

```
SELECT 'foo'      'bar';
```

则不是合法的语法（这种有些奇怪的行为是SQL指定的，IvorySQL遵循了该标准）。

C风格转义的字符串常量

IvorySQL也接受“转义”字符串常量，这也是SQL标准的一个扩展。一个转义字符串常量可以通过在开单引号前面写一个字母`E`（大写或小写形式）来指定，例如`E’ foo'`（当一个转义字符串常量跨行时，只在第一个开引号之前写`E`）。在一个转义字符串内部，一个反斜线字符（\）会开始一个C风格的 反斜线转义序列，在其中反斜线和后续字符的组合表示一个特殊的字节值（如 表 4.1中所示）。

表 4.1. 反斜线转义序列

反斜线转义序列	解释
\b	退格

`\f`	换页
`\n`	换行
`\r`	回车
`\t`	制表符
`*`o*, `*`oo*, `*`ooo*` (o = 0–7)	八进制字节值
`\x*`h, `\x`hh` (h* = 0–9, A–F)	十六进制字节值
`\u*`xxxx, `\U`xxxxxxxx` (x* = 0–9, A–F)	16 或 32-位十六进制 Unicode 字符值

跟随在一个反斜线后面的任何其他字符被当做其字面意思。因此，要包括一个反斜线字符，请写两个反斜线（`\\`）。在一个转义字符串中包括一个单引号除了普通方法’ ``之外，还可以写成`\'`。

你要负责保证你创建的字节序列由服务器字符集编码中合法的字符组成，特别是在使用八进制或十六进制转义时。一个有用的替代方法是使用Unicode转义或替代的Unicode转义语法，如 第 4.1.2.3 节中所述；然后服务器将检查字符转换是否可行。

> 小心
>
> 如果配置参数 standard_conforming_strings为`off`，那么IvorySQL对常规字符串常量和转义字符串常量中的反斜线转义都识别。不过，在IvorySQL中该参数的默认值为`on`，意味着只在转义字符串常量中识别反斜线转义。这种行为更兼容标准，但是可能打断依赖于历史行为（反斜线转义总是会被识别）的应用。作为一种变通，你可以设置该参数为`off`，但是最好迁移到符合新的行为。如果你需要使用一个反斜线转义来表示一个特殊字符，为该字符串常量写上一个`E`。在`standard_conforming_strings`之外，配置参数 escape_string_warning和 backslash_quote也决定了如何对待字符串常量中的反斜线。代码零的字符不能出现在一个字符串常量中。

带有 Unicode 转义的字符串常量

IvorySQL也支持另一种类型的字符串转义语法，它允许用代码点指定任意Unicode字符。一个Unicode转义字符串常量开始于`U&`（大写或小写形式的字母U，后跟花号），后面紧跟着开引号，之间没有任何空白，例如`U&'foo'`（注意这产生了与操作符`&`的混淆。在操作符周围使用空白来避免这个问题）。在引号内，Unicode字符可以通过写一个后跟4位十六进制代码点编号或者一个前面有加号的6位十六进制代码点编号的反斜线来指定。例如，字符串’ data'`可以被写为

```
U&'d\0061t\+000061'
```

下面的例子用斯拉夫字母写出了俄语的单词“slon”（大象）：

```
U&'\0441\043B\043E\043D'
```

如果想要一个不是反斜线的转义字符，可以在字符串之后使用`UESCAPE`子句来指定，例如：

```
U&'d!0061t!+000061' UESCAPE '!'
```

转义字符可以是出一个十六进制位、加号、单引号、双引号或空白字符之外的任何单一字符。

要在一个字符串中包括一个表示其字面意思的转义字符，把它写两次。

4位或6位转义形式可用于指定UTF-16代理项对，以组成代码点大于U+FFFF的字符，尽管从技术上讲，6位形式的可用性使得这是不必要的（代

理项对不是直接存储的，而是合并到单个代码点中。）

如果服务器编码不是UTF-8，则由这些转义序列之一标识的Unicode代码点将转换为实际的服务器编码；如果不可能，则会报告错误。

此外，字符串常量的Unicode转义语法仅在配置参数standard_conforming_strings开启时才有效。这是因为否则这种语法可能会混淆解析SQL语句的客户端，可能导致SQL注入和类似的安全问题。如果该参数设置为off，则此语法将被拒绝并显示错误消息。

美元引用的字符串常量

虽然用于指定字符串常量的标准语法通常都很方便，但是当字符串中包含了很多单引号或反斜线时很难理解它，因为每一个都需要被双写。要在这种情形下允许可读性更好的查询，IvorySQL提供了另一种被称为“美元引用”的方式来书写字符串常量。一个美元引用的字符串常量由一个美元符号（$）、一个可选的另个或更多字符的“标签”、另一个美元符号、一个构成字符串内容的任意字符序列、一个美元符号、开始这个美元引用的相同标签和一个美元符号组成。例如，这里有两种不同的方法使用美元引用指定字符串“Dianne’s horse”：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

注意在美元引用字符串中，单引号可以在不被转义的情况下使用。事实上，在一个美元引用字符串中不需要对字符进行转义：字符串内容总是按其字面意思写出。反斜线不是特殊的，并且美元符号也不是特殊的，除非它们是匹配开标签的一个序列的一部分。

可以通过在每一个嵌套级别上选择不同的标签来嵌套美元引用字符串常量。这最常被用在编写函数定义上。例如：

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

这里，序列`$q$[\t\r\n\v\\]$q$表示一个美元引用的文字串[\t\r\n\v\\]，当该函数体被IvorySQL执行时它将被识别。但是因为该序列不匹配外层的美元引用的定界符$function$`，它只是一些在外层字符串所关注的常量中的字符而已。

一个美元引用字符串的标签（如果有）遵循一个未被引用标识符的相同规则，除了它不能包含一个美元符号之外。标签是大小写敏感的，因此`$tag$String content$tag$是正确的，但是$TAG$String content$tag$`不正确。

一个跟着一个关键词或标识符的美元引用字符串必须用空白与之分隔开，否则美元引用定界符可能会被作为前面标识符的一部分。

美元引用不是SQL标准的一部分，但是在书写复杂字符串文字方面，它常常是一种比兼容标准的单引号语法更方便的方法。当要表示的字符串常量位于其他常量中时它特别有用，这种情况常常在过程函数定义中出现。如果用单引号语法，上一个例子中的每个反斜线将必须被写成四个反斜线，这在解析原始字符串常量时会被缩减到两个反斜线，并且接着在函数执行期间重新解析内层字符串常量时变成一个。

位串常量

位串常量看起来像常规字符串常量在开引号之前（中间无空白）加了一个`B`（大写或小写形式），例如`B'1001'`。位串常量中允许的字符只有`0`和`1`。

作为一种选择，位串常量可以用十六进制记号法指定，使用一个前导`X`（大写或小写形式）,例如`X'1FF'`。这种记号法等价于一个用四个二进制位取代每个十六进制位的位串常量。

两种形式的位串常量可以以常规字符串常量相同的方式跨行继续。美元引用不能被用在位串常量中。

数字常量

在这些一般形式中可以接受数字常量：

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

其中 `digits` 是一个或多个十进制数字（0到9）。如果使用了小数点，在小数点前面或后面必须至少有一个数字。如果存在一个指数标记（`e`），在其后必须跟着至少一个数字。在该常量中不能嵌入任何空白或其他字符。注意任何前导的加号或减号并不实际被考虑为常量的一部分，它是一个应用到该常量的操作符。

这些是合法数字常量的例子：

```
42
3.5
4.
.001
5e2
1.925e-3
```

如果一个不包含小数点和指数的数字常量的值适合类型`integer`（32位），它首先被假定为类型`integer`。否则如果它的值适合类型`bigint`（64位），它被假定为类型`bigint`。再否则它会被取做类型`numeric`。包含小数点和/或指数的常量总是首先被假定为类型`numeric`。

一个数字常量初始指派的数据类型只是类型转换算法的一个开始点。在大部分情况中，常量将被根据上下文自动被强制到最合适的类型。必要时，你可以通过造型它来强制一个数字值被解释为一种指定数据类型。例如，你可以这样强制一个数字值被当做类型`real`（`float4`）：

```
REAL '1.23'  -- string style
1.23::REAL   -- IvorySQL (historical) style
```

这些实际上只是接下来要讨论的一般造型记号的特例。

其他类型的常量

一种任意类型的一个常量可以使用下列记号中的任意一种输入：

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

字符串常量的文本被传递到名为 `type` 的类型的输入转换例程中。其结果是指定类型的一个常量。如果对该常量的类型没有歧义（例如，当它被直接指派给一个表列时），显式类型造型可以被忽略，在那种情况下它会被自动强制。

字符串常量可以使用常规SQL记号或美元引用书写。

也可以使用一个类似函数的语法来指定一个类型强制：

```
typename ( 'string' )
```

但是并非所有类型名都可以用在这种方法中，详见 第 4.2.9 节。

如 第 4.2.9 节中讨论的，`::`、`CAST()`以及函数调用语法也可以被用来指定任意表达式的运行时类型转换。要避免语法歧义，`type 'string'`语法只能被用来指定简单文字常量的类型。`type 'string'`语法上的另一个限制是它无法对数组类型工作，指定一个数组常量的类型可使用`::`或`CAST()`。

`CAST()`语法符合SQL。`type 'string'`语法是该标准的一般化：SQL指定这种语法只用于一些数据类型，但是IvorySQL允许它用于所有类型。带有`::`的语法是IvorySQL的历史用法，就像函数调用语法一样。

操作符

一个操作符名是最多`NAMEDATALEN`-1（默认为63）的一个字符序列，其中的字符来自下面的列表：

```
\+ - * / < > = ~ ! @ # % ^ & | ` ?
```

不过，在操作符名上有一些限制：

- `--`和`/*`不能在一个操作符名的任何地方出现，因为它们将被作为一段注释的开始。
- 一个多字符操作符名不能以`+`或`-`结尾，除非该名称也至少包含这些字符中的一个：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

例如，`@-`是一个被允许的操作符名，但`*-`不是。这些限制允许IvorySQL解析SQL兼容的查询而不需要在记号之间有空格。

当使用非SQL标准的操作符名时，你通常需要用空格分隔相邻的操作符来避免歧义。例如，如果你定义了一个名为`@`的前缀操作符，你不能写`X*@Y`，你必须写`X* @Y`来确保IvorySQL把它读作两个操作符名而不是一个。

特殊字符

一些不是数字字母的字符有一种不同于作为操作符的特殊含义。这些字符的详细用法可以在描述相应语法元素的地方找到。这一节只是为了告知它们的存在以及总结这些字符的目的。

- 跟随在一个美元符号（`$`）后面的数字被用来表示在一个函数定义或一个预备语句中的位置参数。在其他上下文中该美元符号可以作为一个标识符或者一个美元引用字符串常量的一部分。
- 圆括号（`()`）具有它们通常的含义，用来分组表达式并且强制优先。在某些情况中，圆括号被要求作为一个特定 SQL 命令的固定语法的一部分。
- 方括号（`[]`）被用来选择一个数组中的元素。更多关于数组的信息见 第 8.15 节。
- 逗号（`,`）被用在某些语法结构中来分割一个列表的元素。
- 分号（`;`）结束一个 SQL 命令。它不能出现在一个命令中间的任何位置，除了在一个字符串常量中或者一个被引用的标识符中。
- 冒号（`:`）被用来从数组中选择“切片”（见 第 8.15 节）。在某些 SQL 的“方言”（例如嵌入式 SQL）中，冒号被用来作为变量名的前缀。

- 星号（`*`）被用在某些上下文中标记一个表的所有域或者组合值。当它被用作一个聚集函数的参数时，它还有一种特殊的含义，即该聚集不要求任何显式参数。
- 句点（`.`）被用在数字常量中，并且被用来分割模式、表和列名。

注释

一段注释是以双横杠开始并且延伸到行结尾的一个字符序列，例如：

```
-- This is a standard SQL comment
```

另外，也可以使用 C 风格注释块：

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

这里该注释开始于 `/*` 并且延伸到匹配出现的`*/`。这些注释块可按照SQL标准中指定的方式嵌套，但和C中不同。这样我们可以注释掉一大段可能包含注释块的代码。

在进一步的语法分析前，注释会被从输入流中被移除并且实际被替换为空白。

操作符优先级

表4.2显示了IvorySQL中操作符的优先级和结合性。大部分操作符具有相同的优先并且是左结合的。操作符的优先级和结合性被硬写在解析器中。如果您希望以不同于优先级规则所暗示的方式解析具有多个运算符的表达式，请添加括号。

表 4.2. 操作符优先级（从高到低）

操作符/元素	结合性	描述
`.`	左	表/列名分隔符
`::`	左	IvorySQL-风格的类型转换
`[ ]`	左	数组元素选择
`+ -`	右	一元加、一元减
`^`	左	指数
`* / %`	左	乘、除、模
`+ -`	左	加、减
（任意其他操作符）	左	所有其他本地以及用户定义的操作符
`BETWEEN IN LIKE ILIKE SIMILAR`		范围包含、集合成员关系、字符串匹配
`< > = <= >= <>`		比较操作符
`IS ISNULL NOTNULL`		`IS TRUE`、`IS FALSE`、`IS NULL`、`IS DISTINCT FROM`等
`NOT`	右	逻辑否定
`AND`	左	逻辑合取
`OR`	左	逻辑析取

注意该操作符有限规则也适用于与上述内建操作符具有相同名称的用户定义的操作符。例如，如果你为某种自定义数据类型定义了一个“”操作符，它将具有和内建的“”操作符相同的优先级，不管你的操作符要做什么。

当一个模式限定的操作符名被用在`OPERATOR`语法中时，如下面的例子：

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

`OPERATOR`结构被用来为“任意其他操作符”获得 表 4.2中默认的优先级。不管出现在`OPERATOR()`中的是哪个指定操作符，这都是真的。

> **注意**
>
> 版本9.5之前的PostgreSQL使用的操作符优先级规则略有不同。特别是，<=、>= 和 `<>`习惯于被当作普通操作符，`IS`测试习惯于具有较高的优先级。并且在一些认为`NOT`比`BETWEEN`优先级高的情况下，`NOT BETWEEN`和相关的结构的行为不一致。为了更好地兼容SQL标准并且减少对逻辑上等价的结构不一致的处理，这些规则也得到了修改。在大部分情况下，这些变化不会导致行为上的变化，或者可能会产生“no such operator”错误，但可以通过增加圆括号解决。不过在一些极端情况中，查询可能在没有被报告解析错误的情况下发生行为的改变。

值表达式

值表达式被用于各种各样的环境中，例如在`SELECT`命令的目标列表中、作为`INSERT`或`UPDATE`中的新列值或者若干命令中的搜索条件。为了区别于一个表表达式（是一个表）的结果，一个值表达式的结果有时候被称为一个 标量。值表达式因此也被称为 标量表达式（或者甚至简称为 表达式）。表达式语法允许使用算数、逻辑、集合和其他操作从原始部分计算值。

一个值表达式是下列之一：

- 一个常量或文字值
- 一个列引用
- 在一个函数定义体或预备语句中的一个位置参数引用
- 一个下标表达式
- 一个域选择表达式
- 一个操作符调用
- 一个函数调用
- 一个聚集表达式
- 一个窗口函数调用
- 一个类型转换
- 一个排序规则表达式
- 一个标量子查询
- 一个数组构造器
- 一个行构造器
- 另一个在圆括号（用来分组子表达式以及重载优先级）中的值表达式

在这个列表之外，还有一些结构可以被分类为一个表达式，但是它们不遵循任何一般语法规则。这些通常具有一个函数或操作符的语义并且在 第 9 章中的合适位置解释。一个例子是`IS NULL`子句。

我们已经在 第 4.1.2 节中讨论过常量。下面的小节会讨论剩下的选项。

列引用

一个列可以以下面的形式被引用：

```
correlation.columnname
```

correlation
是一个表（有可能以一个模式名限定）的名字，或者是在`FROM`子句中为一个表定义的别名。如果列名在当前索引所使用的表中都是唯一的，关联名称和分隔用的句点可以被忽略（另见 第 7 章）。

位置参数

一个位置参数引用被用来指示一个由SQL语句外部提供的值。参数被用于SQL函数定义和预备查询中。某些客户端库还支持独立于SQL命令字符串来指定数据值，在这种情况中参数被用来引用那些线外数据值。一个参数引用的形式是：

```
$number
```

例如，考虑一个函数`dept`的定义：

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

这里 $1 引用函数被调用时第一个函数参数的值。

下标

如果一个表达式得到了一个数组类型的值，那么可以抽取出该数组值的一个特定元素：

```
expression[subscript]
```

或者抽取出多个相邻元素（一个“数组切片”）：

```
expression[lower_subscript:upper_subscript]
```

（这里，方括号 [ ] 表示其字面意思）。每一个 下标 自身是一个表达式，它将四舍五入到最接近的整数值。

通常，数组 表达式
必须被加上括号，但是当要被加下标的表达式只是一个列引用或位置参数时，括号可以被忽略。还有，当原始数组是多维时，多个下标可以被连接起来。例如：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最后一个例子中的圆括号是必需的。详见 第 8.15 节。

域选择

如果一个表达式得到一个组合类型（行类型）的值，那么可以抽取该行的指定域:

```
expression.fieldname
```

通常行 **表达式**
必须被加上括号，但是当该表达式是仅从一个表引用或位置参数选择时，圆括号可以被忽略。例如:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

（因此，一个被限定的列引用实际上只是域选择语法的一种特例）。一种重要的特例是从一个组合类型的表列中抽取一个域:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

这里需要圆括号来显示 `compositecol` 是一个列名而不是一个表名，在第二种情况中则是显示 `mytable` 是一个表名而不是一个模式名。

你可以通过书写 `.*` 来请求一个组合值的所有域:

```
(compositecol).*
```

这种记法的行为根据上下文会有不同，详见 第 8.16.5 节。

操作符调用

对于一次操作符调用，有两种可能的语法:

`expression operator expression`	（二元中缀操作符）
`operator expression`	（一元前缀操作符）

其中 `operator` 记号遵循 第 4.1.3 节
的语法规则，或者是关键词`AND`、`OR`和`NOT`之一，或者是一个如下形式的受限定操作符名:

```
OPERATOR(schema.operatorname)
```

哪个特定操作符存在以及它们是一元的还是二元的取决于由系统或用户定义的那些操作符。第 9 章
描述了内建操作符。

函数调用

一个函数调用的语法是一个函数的名称（可能受限于一个模式名）后面跟上封闭于圆括号中的参数列表:

```
function_name ([expression [, expression ... ]] )
```

例如，下面会计算 2 的平方根：

```
sqrt(2)
```

当在一个某些用户不信任其他用户的数据库中发出查询时，在编写函数调用时应遵守 第 10.3 节中的安全防范措施。

内建函数的列表在 第 9 章 中。其他函数可以由用户增加。

参数可以有选择地被附加名称。详见 第 4.3 节。

> **注意**
>
> 一个采用单一组合类型参数的函数可以被有选择地称为域选择语法，并且反过来域选择可以被写成函数的风格。也就是说，记号`col(table)`和`table.col`是可以互换的。这种行为是非 SQL 标准的但是在IvorySQL中被提供，因为它允许函数的使用来模拟“计算域”。

聚集表达式

一个 聚集表达式
表示在由一个查询选择的行上应用一个聚集函数。一个聚集函数将多个输入减少到一个单一输出值，例如对输入的求和或平均。一个聚集表达式的语法是下列之一：

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE
filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE
filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE
filter_clause ) ]
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause ) [ FILTER
( WHERE filter_clause ) ]
```

这里 `aggregate_name` 是一个之前定义的聚集（可能带有一个模式名限定），并且 `expression` 是任意自身不包含聚集表达式的值表达式或一个窗口函数调用。可选的 `order_by_clause` 和 `filter_clause` 描述如下。

第一种形式的聚集表达式为每一个输入行调用一次聚集。第二种形式和第一种相同，因为 `ALL` 是默认选项。第三种形式为输入行中表达式的每一个可区分值（或者对于多个表达式是值的可区分集合）调用一次聚集。第四种形式为每一个输入行调用一次聚集，因为没有特定的输入值被指定，它通常只对于 `count()` 聚集函数有用。最后一种形式被用于 `*`有序集 聚集函数，其描述如下。

大部分聚集函数忽略空输入，这样其中一个或多个表达式得到空值的行将被丢弃。除非另有说明，对于所有内建聚集都是这样。

例如，`count(*)` 得到输入行的总数。 `count(f1)` 得到输入行中 `f1` 为非空的数量，因为 `count` 忽略空值。而 `count(distinct f1)` 得到 `f1` 的非空可区分值的数量。

一般地，交给聚集函数的输入行是未排序的。在很多情况中这没有关系，例如不管接收到什么样的输入，`min` 总是产生相同的结果。但是，某些聚集函数（例如 `array_agg` 和 `string_agg` ）依据输入行的排序产生结果。当使用这类聚集时，可选的 `order_by_clause` 可以被用来指定想要的顺序。`order_by_clause` 与查询级别的 `ORDER BY` 子句（如 第 7.5 节 所述）具有相同的语法，除非它的表达式总是仅有表达式并且不能是输出列名称或编号。例如：

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

在处理多参数聚集函数时，注意 ORDER BY 出现在所有聚集参数之后。例如，要这样写：

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

而不能这样写：

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- 不正确
```

后者在语法上是合法的，但是它表示用两个`ORDER BY`键来调用一个单一参数聚集函数（第二个是无用的，因为它是一个常量）。

如果在 order_by_clause 之外指定了 DISTINCT ，那么所有的 ORDER BY 表达式必须匹配聚集的常规参数。也就是说，你不能在 DISTINCT 列表没有包括的表达式上排序。

> **注意**
>
> 在一个聚集函数中指定 DISTINCT 以及 ORDER BY 的能力是一种IvorySQL扩展。按照到目前为止的描述，如果一般目的和统计性聚集中排序是可选的，在要为它排序输入行时可以在该聚集的常规参数 列表中放置 ORDER BY 。有一个聚集函数的子集叫做 有序集聚集 ，它要求一个 order_by_clause，通常是因为该聚集的计算只对其输入行的特定顺序有意义。有序集聚集的典型例子包括排名和百分位计算。按照上文的最后一种语法，对于 一个有序集聚集， order_by_clause 被写在 WITHIN GROUP (…) 中。 order_by_clause 中的表达式会像普通聚集参数一样对每一个输入行计算一次，按照每个 order_by_clause 的要求排序并且交给该聚集函数作为输入参数（这和非 WITHIN GROUP order_by_clause 的情况不同，在其中表达式的结果不会被作为聚集函数的参数）。如果有在 WITHIN GROUP 之前的参数表达式，会把它们称 为直接参数 以便与列在 order_by_clause 中的 聚集参数 相区分。与普通聚集参数不同，针对每次聚集调用只会计算一次直接参数，而不是为每一个输入行 计算一次。这意味着只有那些变量被 GROUP BY 分组时，它们才能包含这些变量。这个限制同样适用于根本不在一个聚集表达式内部的直接参数。直接参数通常被用于百分数之类的东西，它们只有作为每次聚集计算用一次的单一值才有意义。直接参数列表可以为空，在这种情况下，写成 () 而不是 (*) （实际上IvorySQL接受两种拼写，但是只有第一种符合 SQL 标准）。

有序集聚集的调用例子：

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

这会从表 households 的 income 列得到第 50 个百分位或者中位的值。这里`0.5`是一个直接参数，对于百分位部分是一个 在不同行之间变化的值的情况它没有意义。

如果指定了 FILTER ，那么只有对 filter_clause 计算为真的输入行会被交给该聚集函数，其他行会被丢弃。例如：

```
SELECT
```

```
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

预定义的聚集函数在 第 9.21 节 中描述。其他聚集函数可以由用户增加。

一个聚集表达式只能出现在 SELECT 命令的结果列表或是 HAVING 子句中。在其他子句（如 WHERE ）中禁止使用它，因为那些子句的计算在逻辑上是在聚集的结果被形成之前。

当一个聚集表达式出现在一个子查询中（见 第 4.2.11 节 和 第 9.23 节），聚集通常在该子查询的行上被计算。但是如果该聚集的参数（以及 filter_clause，如果有）只包含外层变量则会产生一个异常：该聚集则属于最近的那个外层，并且会在那个查询的行上被计算。该聚集表达式从整体上则是对其所出现于的子查询的一种外层引用，并且在那个子查询的任意一次计算中都作为一个常量。只出现在结果列表或 HAVING 子句的限制适用于该聚集所属的查询层次。

窗口函数调用

一个*窗口函数调用*表示在一个查询选择的行的某个部分上应用一个聚集类的函数。和非窗口聚集函数调用不同，这不会被约束为将被选择的行分组为一个单一的输出行 — 在查询输出中每一个行仍保持独立。不过，窗口函数能够根据窗口函数调用的分组声明（PARTITION BY 列表）访问属于当前行所在分组中的所有行。一个窗口函数调用的语法是下列之一：

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ]
OVER ( window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
```

其中 window_definition 的语法是

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [,
...] ]
[ frame_clause ]
```

可选的 frame_clause 是下列之一

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

其中 frame_start 和 frame_end 可以是下面形式中的一种

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

而 frame_exclusion 可以是下列之一

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

这里，expression 表示任何自身不含有窗口函数调用的值表达式。

window_name 是对定义在查询的 WINDOW 子句中的一个命名窗口声明的引用。还可以使用在 WINDOW 子句中定义命名窗口的相同语法在圆括号内给定一个完整的 window_definition，详见 SELECT 参考页。值得指出的是，OVER wname 并不严格地等价于 OVER (wname …)，后者表示复制并修改窗口定义，并且在被引用窗口声明包括一个帧子句时会被拒绝。

PARTITION BY 选项将查询的行分组成为 分区，窗口函数会独立地处理它们。PARTITION BY 工作起来类似于一个查询级别的 GROUP BY 子句，不过它的表达式总是只是表达式并且不能是输出列的名称或编号。如果没有 PARTITION BY，该查询产生的所有行被当作一个单一分区来处理。ORDER BY 选项决定被窗口函数处理的一个分区中的行的顺序。它工作起来类似于一个查询级别的 ORDER BY 子句，但是同样不能使用输出列的名称或编号。如果没有 ORDER BY，行将被以未指定的顺序被处理。

frame_clause 指定构成 窗口帧 的行集合，它是当前分区的一个子集，窗口函数将作用在该帧而不是整个分区。帧中的行集合会随着哪一行是当前行而变化。在 RANGE、ROWS 或者 GROUPS 模式中可以指定帧，在每一种情况下，帧的范围都是从 frame_start 到 frame_end。如果 frame_end 被省略，则末尾默认为 CURRENT ROW。

UNBOUNDED PRECEDING 的一个 frame_start 表示该帧开始于分区的第一行，类似地 UNBOUNDED FOLLOWING 的一个 frame_end 表示该帧结束于分区的最后一行。

在 RANGE 或 GROUPS 模式中，CURRENT ROW 的一个 frame_start 表示帧开始于当前行的第一个 平级行（被窗口的 ORDER BY 子句排序为与当前行等效的行），而 CURRENT ROW 的一个 frame_end 表示帧结束于当前行的最后一个平级行。在 ROWS 模式中，CURRENT ROW 就表示当前行。

在 offset PRECEDING 以及 offset FOLLOWING 帧选项中，offset 必须是一个不包含任何变量、聚集函数或者窗口函数的表达式。offset 的含义取决于帧模式：

- 在 ROWS 模式中， offset 必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行之前或者之后指定数量的行。
- 在 GROUPS 模式中，offset 也必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行的平级组之前或者之后指定数量的*平级组*，这里平级组是在 ORDER BY 顺序中等效的行集合（要使用 GROUPS 模式，在窗口定义中就必须有一个 ORDER BY 子句）。
- 在 RANGE 模式中，这些选项要求 ORDER BY 子句正好指定一列。offset 指定当前行中那一列的值与它在该帧中前面或后面的行中的列值的最大差值。offset 表达式的数据类型会随着排序列的数据类型而变化。对于数字的排序列，它通常是与排序列相同的类型，但对于日期时间排序列它是一个 interval。例如，如果排序列是类型 date 或者 timestamp，我们可以写 RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING。offset 仍然要求是非空且非负，不过“非负”的含义取决于它的数据类型。

在任何一种情况下，到帧末尾的距离都受限于到分区末尾的距离，因此对于离分区末尾比较近的行来说，帧可能会包含比较少的行。

注意在 ROWS 以及 GROUPS 模式中， 0 PRECEDING 和 0 FOLLOWING 与 CURRENT ROW 等效。通常在 RANGE 模式中，这个结论也成立（只要有一种合适的、与数据类型相关的“零”的含义）。

frame_exclusion
选项允许当前行周围的行被排除在帧之外，即便根据帧的开始和结束选项应该把它们包括在帧中。EXCLUDE CURRENT ROW 会把当前行排除在帧之外。EXCLUDE GROUP
会把当前行以及它在顺序上的平级行都排除在帧之外。EXCLUDE TIES
把当前行的任何平级行都从帧中排除，但不排除当前行本身。EXCLUDE NO OTHERS
只是明确地指定不排除当前行或其平级行的这种默认行为。

默认的帧选项是 RANGE UNBOUNDED PRECEDING，它和 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 相同。如果使用 ORDER BY，这会把该帧设置为从分区开始一直到当前行的最后一个 ORDER BY
平级行的所有行。如果不使用 ORDER
BY，就意味着分区中所有的行都被包括在窗口帧中，因为所有行都成为了当前行的平级行。

限制是 frame_start 不能是 UNBOUNDED FOLLOWING 、frame_end 不能是 UNBOUNDED PRECEDING，并且在上述 frame_start 和 frame_end 选项的列表中 frame_end 选择不能早于 frame_start 选择出现 — 例如不允许 RANGE BETWEEN CURRENT ROW AND offset PRECEDING，但允许 ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING ，虽然它不会选择任何行。

如果指定了 FILTER ，那么只有对 filter_clause
计算为真的输入行会被交给该窗口函数，其他行会被丢弃。只有是聚集的窗口函数才接受 FILTER 。

内建的窗口函数在 表 9.60
中介绍。用户可以加入其他窗口函数。此外，任何内建的或者用户定义的通用聚集或者统计性聚集都可以被用作窗口函数（有序集和假想集聚集当前不能被用作窗口函数）。

使用 * 的语法被用来把参数较少的聚集函数当作窗口函数调用，例如 count(*) OVER (PARTITION BY x ORDER BY y)。星号（*）通常不被用于窗口相关的函数。窗口相关的函数不允许在函数参数列表中用 DISTINCT 或 ORDER BY。

只有在 SELECT 列表和查询的 ORDER BY 子句中才允许窗口函数调用。

更多关于窗口函数的信息可以在 第 3.5 节、 第 9.22 节 以及 第 7.2.5 节 中找到。

类型转换

一个类型造型指定从一种数据类型到另一种数据类型的转换。IvorySQL接受两种等价的类型造型语法：

```
CAST ( expression AS type )
expression::type
```

CAST 语法遵从 SQL，而用 :: 的语法是IvorySQL的历史用法。

当一个造型被应用到一种未知类型的值表达式上时，它表示一种运行时类型转换。只有已经定义了一种合适的类型转换操作时，该造型才会成功。注意这和常量的造型（如 第 4.1.2.7 节
中所示）使用不同。应用于一个未修饰串文字的造型表示一种类型到一个文字常量值的初始赋值，并且因此它将对任意类型都成功（如果该串文字的内容对于该数据类型的输入语法是可接受的）。

如果一个值表达式必须产生的类型没有歧义（例如当它被指派给一个表列），通常可以省略显式类型造型，在这种情况下系统会自动应用一个类型造型。但是，只有对在系统目录中被标记为“OK to apply implicitly”的造型才会执行自动造型。其他造型必须使用显式造型语法调用。这种限制是为了防止出人意料的转换被无声无息地应用。

还可以用像函数的语法来指定一次类型造型：

```
typename ( expression )
```

不过，这只对那些名字也作为函数名可用的类型有效。例如，double precision 不能以这种方式使用，但是等效的 float8 可以。还有，如果名称 interval、time 和 timestamp 被用双引号引用，那么由于语法冲突的原因，它们只能以这种风格使用。因此，函数风格的造型语法的使用会导致不一致性并且应该尽可能被避免。

> **注意**
>
> 函数风格的语法事实上只是一次函数调用。当两种标准造型语法之一被用来做一次运行时转换时，它将在内部调用一个已注册的函数来执行该转换。简而言之，这些转换函数具有和它们的输出类型相同的名字，并且因此“函数风格的语法”无非是对底层转换函数的一次直接调用。显然，一个可移植的应用不应当依赖于它。详见 CREATE CAST。

排序规则表达式

COLLATE 子句会重载一个表达式的排序规则。它被追加到它适用的表达式：

```
expr COLLATE collation
```

这里 collation 可能是一个受模式限定的标识符。COLLATE 子句比操作符绑得更紧，需要时可以使用圆括号。

如果没有显式指定排序规则，数据库系统会从表达式所涉及的列中得到一个排序规则，如果该表达式没有涉及列，则会默认采用数据库的默认排序规则。

COLLATE 子句的两种常见使用是重载 ORDER BY 子句中的排序顺序，例如：

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

以及重载具有区域敏感结果的函数或操作符调用的排序规则，例如：

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

注意在后一种情况中，COLLATE 子句被附加到我们希望影响的操作符的一个输入参数上。COLLATE 子句被附加到该操作符或函数调用的哪个参数上无关紧要，因为被操作符或函数应用的排序规则是考虑所有参数得来的，并且一个显式的 COLLATE 子句将重载所有其他参数的排序规则（不过，附加非匹配 COLLATE 子句到多于一个参数是一种错误。详见 第 24.2 节）。因此，这会给出和前一个例子相同的结果：

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

但是这是一个错误：

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

因为它尝试把一个排序规则应用到 > 操作符的结果，而它的数据类型是非可排序数据类型 boolean。

标量子查询

一个标量子查询是一种圆括号内的普通 SELECT 查询，它刚好返回一行一列（关于书写查询可见 第 7 章）。`SELECT`查询被执行并且该单一返回值被使用在周围的值表达式中。将一个返回超过一行或一列的查询作为一个标量子查询使用是一种错误（但是如果在一次特定执行期间该子查询没有返回行则不是错误，该标量结果被当作为空）。该子查询可以从周围的查询中引用变量，这些变量在该子查询的任何一次计算中都将作为常量。对于其他涉及子查询的表达式还可见 第 9.23 节。

例如，下列语句会寻找每个州中最大的城市人口：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

数组构造器

一个数组构造器是一个能构建一个数组值并且将值用于它的成员元素的表达式。一个简单的数组构造器由关键词 ARRAY、一个左方括号 [、一个用于数组元素值的表达式列表（用逗号分隔）以及最后的一个右方括号] 组成。例如：

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

默认情况下，数组元素类型是成员表达式的公共类型，使用和 UNION 或 CASE 结构（见 第 10.5 节）相同的规则决定。你可以通过显式将数组构造器造型为想要的类型来重载，例如：

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

这和把每一个表达式单独地造型为数组元素类型的效果相同。关于造型的更多信息请见 第 4.2.9 节。

多维数组值可以通过嵌套数组构造器来构建。在内层的构造器中，关键词 ARRAY 可以被忽略。例如，这些语句产生相同的结果：

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
```

```
 {{1,2},{3,4}}
(1 row)
```

因为多维数组必须是矩形的，处于同一层次的内层构造器必须产生相同维度的子数组。任何被应用于外层 ARRAY 构造器的造型会自动传播到所有的内层构造器。

多维数组构造器元素可以是任何得到一个正确种类数组的任何东西，而不仅仅是一个子- ARRAY 结构。例如：

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

你可以构造一个空数组，但是因为无法得到一个无类型的数组，你必须显式地把你的空数组造型成想要的类型。例如：

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

也可以从一个子查询的结果构建一个数组。在这种形式中，数组构造器被写为关键词 ARRAY 后跟着一个加了圆括号（不是方括号）的子查询。例如：

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                              array
-------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

子查询必须返回一个单一列。如果子查询的输出列是非数组类型，
结果的一维数组将为该子查询结果中的每一行有一个元素，
并且有一个与子查询的输出列匹配的元素类型。如果子查询的输出列

是一种数组类型，结果将是同类型的一个数组，但是要高一个维度。
在这种情况下，该子查询的所有行必须产生同样维度的数组，否则结果 就不会是矩形形式。

用 ARRAY 构建的一个数组值的下标总是从一开始。更多关于数组的信息，请见 第 8.15 节。

行构造器

一个行构造器是能够构建一个行值（也称作一个组合类型）并用值作为其成员域的表达式。一个行构造器由关键词
ROW、一个左圆括号、用于行的域值的零个或多个表达式（用逗号分隔）以及最后的一个右圆括号组成。例如：

```
SELECT ROW(1,2.5,'this is a test');
```

当在列表中有超过一个表达式时，关键词 ROW 是可选的。

一个行构造器可以包括语法 rowvalue .*，它将被扩展为该行值的元素的一个列表，就像在一个顶层 SELECT 列表（见 第 8.16.5 节）中使用 .* 时发生的事情一样。例如，如果表 t 有列 f1 和 f2，那么这些是相同的：

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

注意

在PostgreSQL 8.2 以前，.* 语法不会在行构造器中被扩展，这样写 ROW(t.*, 42)
会创建一个有两个域的行，其第一个域是另一个行值。新的行为通常更有用。如果你需要嵌套行值的旧行为，写内层行值时不要用 .*，例如 ROW(t, 42)。

默认情况下，由一个 ROW
表达式创建的值是一种匿名记录类型。如果必要，它可以被造型为一种命名的组合类型 —
或者是一个表的行类型，或者是一种用 CREATE TYPE AS
创建的组合类型。为了避免歧义，可能需要一个显式造型。例如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 不需要造型因为只有一个 getf1() 存在
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;
```

```
-- 现在我们需要一个造型来指示要调用哪个函数:
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

行构造器可以被用来构建存储在一个组合类型表列中的组合值，或者被传递给一个接受组合参数的函数。还有，可以比较两个行值，或者用 IS NULL 或 IS NOT NULL 测试一个行，例如：

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

详见 第 9.24 节。如 第 9.23 节 中所讨论的，行构造器也可以被用来与子查询相连接。

表达式计算规则

子表达式的计算顺序没有被定义。特别地，一个操作符或函数的输入不必按照从左至右或其他任何固定顺序进行计算。

此外，如果一个表达式的结果可以通过只计算其一部分来决定，那么其他子表达式可能完全不需要被计算。例如，如果我们写：

```
SELECT true OR somefunc();
```

那么 somefunc() 将（可能）完全不被调用。如果我们写成下面这样也是一样：

```
SELECT somefunc() OR true;
```

注意这和一些编程语言中布尔操作符从左至右的“短路”不同。

因此，在复杂表达式中使用带有副作用的函数是不明智的。在 WHERE 和 HAVING 子句中依赖副作用或计算顺序尤其危险，因为在建立一个执行计划时这些子句会被广泛地重新处理。这些子句中布尔表达式（AND / OR / NOT 的组合）可能会以布尔代数定律所允许的任何方式被重组。

当有必要强制计算顺序时，可以使用一个 CASE 结构（见 第 9.18 节）。例如，在一个 WHERE 子句中使用下面的方法尝试避免除零是不可靠的：

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

但是这是安全的：

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

一个以这种风格使用的 CASE
结构将使得优化尝试失败，因此只有必要时才这样做（在这个特别的例子中，最好通过写 y > 1.5*x
来回避这个问题）。

不过，CASE 不是这类问题的万灵药。上述技术的一个限制是， 它无法阻止常量子表达式的提早计算。如 第 38.7 节 中所述，当查询被规划而不是被执行时，被标记成 IMMUTABLE 的函数和操作符可以被计算。因此

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

很可能会导致一次除零失败，因为规划器尝试简化常量子表达式。即便是 表中的每一行都有 x >
0（这样运行时永远不会进入到 ELSE 分支）也是这样。

虽然这个特别的例子可能看起来愚蠢，没有明显涉及常量的情况可能会发生
在函数内执行的查询中，因为因为函数参数的值和本地变量可以作为常量
被插入到查询中用于规划目的。例如，在PL/pgSQL函数 中，使用一个 IF -THEN -ELSE 语句来
保护一种有风险的计算比把它嵌在一个 CASE 表达式中要安全得多。

另一个同类型的限制是，一个 CASE 无法阻止其所包含的聚集表达式 的计算，因为在考虑 SELECT 列表或
HAVING 子句中的 其他表达式之前，会先计算聚集表达式。例如， 下面的查询会导致一个除零错误，
虽然看起来好像已经这种情况加以了保护：

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

min() 和 avg() 聚集会在所有输入行上并行地计算， 因此如果任何行有 employees 等于零，在有机会测试
min() 的结果之前，就会发生除零错误。取而代之的是，可以使用 一个 WHERE 或 FILTER
子句来首先阻止有问题的输入行到达一个聚集函数。

调用函数

IvorySQL允许带有命名参数的函数被使用 位置 或 命名
记号法调用。命名记号法对于有大量参数的函数特别有用，因为它让参数和实际参数之间的关联更明显和可靠。在位置记号法中，书写一个函数调用时，其参数值要按照它们在函数声明中被定义的顺序书写。在命名记号法中，参数根据名称匹配函数参数，并且可以以任何顺序书写。对于每一种记法，还要考虑函数参数类型的效果，这些在 第 10.3 节 有介绍。

在任意一种记号法中，在函数声明中给出了默认值的参数根本不需要在调用中写出。但是这在命名记号法中特别有用，因为任何参数的组合都可以被忽略。而在位置记号法中参数只能从右往左忽略。

IvorySQL也支持 混合
记号法，它组合了位置和命名记号法。在这种情况中，位置参数被首先写出并且命名参数出现在其后。

下列例子将展示所有三种记号法的用法：

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

函数 concat_lower_or_upper 有两个强制参数，a 和 b。此外，有一个可选的参数 uppercase，其默认值为 false。a 和 b 输入将被串接，并且根据 uppercase
参数被强制为大写或小写形式。这个函数的剩余细节对这里并不重要（详见 第 38 章）。

使用位置记号

在IvorySQL中，位置记号法是给函数传递参数的传统机制。一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

所有参数被按照顺序指定。结果是大写形式，因为 uppercase 被指定为 true。另一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

这里，uppercase 参数被忽略，因此它接收它的默认值
false，并导致小写形式的输出。在位置记号法中，参数可以按照从右往左被忽略并且因此而得到默认值。

使用命名记号

在命名记号法中，每一个参数名都用 ⇒ 指定来把它与参数表达式分隔开。例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

再次，参数 uppercase 被忽略，因此它被隐式地设置为 false。使用命名记号法的一个优点是参数可以用任何顺序指定，例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

为了向后兼容性，基于 ":=" 的旧语法仍被支持：

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

使用混合记号

混合记号法组合了位置和命名记号法。不过，正如已经提到过的，命名参数不能超越位置参数。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

在上述查询中，参数 a 和 b 被以位置指定，而 uppercase 通过名字指定。在这个例子中，这只增加了一点文档。在一个具有大量带默认值参数的复杂函数中，命名的或混合的记号法可以节省大量的书写并且减少出错的机会。

> 注意
>
> 命名的和混合的调用记号法当前不能在调用聚集函数时使用（但是当聚集函数被用作窗口函数时它们可以被使用）。

Oracle兼容功能

更改表

语法

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
action;

action:
    ADD ( add_coldef [, ... ] )
    | MODIFY ( modify_coldef [, ... ] )
    | DROP [ COLUMN ] ( column_name [, ... ] )

add_coldef:
    column_name data_type

modify_coldef:
    column_name data_type alter_using

alter_using:
    USING expression
```

参数

`name` 表名.

`column_name` 列名.

`data_type` 列类型.

`expression` 值表达式.

`ADD keyword` 增加表的列, 可以增加一个列或多个列.

`MODIFY keyword` 修改表的列, 可以修改一个列或多个列.

`DROP keyword` 删除表的列, 可以删除一个列或多个列.

`USING keyword` 修改列的值.

示例

```
ADD:
create table tb_test1(id int, flg char(10));

alter table tb_test1 add (name varchar);

ALTER TABLE tb_test1
    ADD adress varchar,
    ADD num int,
```

```
    ADD flg1 char;

\d tb_test1
                  Table "public.tb_test1"
 Column |      Type       | Collation | Nullable | Default
--------+-----------------+-----------+----------+---------
 id     | pg_catalog.int4 |           |          |
 flg    | char(10)        |           |          |
 name   | varchar2(4000)  |           |          |
 adress | varchar2(4000)  |           |          |
 num    | pg_catalog.int4 |           |          |
 flg1   | char(1)         |           |          |

MODIFY:
create table tb_test2(id int, flg char(10), num varchar);

insert into tb_test2 values('1', 2, '3');

ALTER TABLE tb_test2 ALTER COLUMN id TYPE char;

\d tb_test2
                 Table "public.tb_test2"
 Column |      Type      | Collation | Nullable | Default
--------+----------------+-----------+----------+---------
 id     | char(1)        |           |          |
 flg    | char(10)       |           |          |
 num    | varchar2(4000) |           |          |

DROP:
create table tb_test3(id int, flg1 char(10), flg2 char(11), flg3 char(12), flg4
char(13),
                       flg5 char(14), flg6 char(15));

ALTER TABLE tb_test3 DROP id;

\d tb_test3
              Table "public.tb_test3"
 Column |   Type   | Collation | Nullable | Default
--------+----------+-----------+----------+---------
 flg1   | char(10) |           |          |
 flg2   | char(11) |           |          |
 flg3   | char(12) |           |          |
 flg4   | char(13) |           |          |
```

```
 flg5   | char(14) |          |          |
 flg6   | char(15) |          |          |
```

删除表

语法

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE [ FROM ] [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

参数

`table_name` 表名.

`alias` 表别名.

`using_list` 一个表表达式的列表，它允许在WHERE条件中出现 来自其他表的列.

`condition` 一个返回boolean类型值的表达式.

`cursor_name` 要在WHERE CURRENT OF情况中使用的游标的名称.

`output_expression` 在每一行被删除后，会被DELETE计算并且返回的表达式.

`output_name` 被返回列的名称.

使用

```
create table tb_test4(id int, flg char(10));

insert into tb_test4 values(1, '2'), (5, '6');

delete from tb_test4 where id = 1;

table tb_test4;
 id |    flg
----+------------
 5  | 6
(1 row)
```

更新表

语法

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
```

```
    SET { [ table_name | alias ] column_name = { expression | DEFAULT }
| ( [ table_name | alias ] column_name [, ...] ) = [ ROW ]    ( { expression | DEFAULT
} [, ...] )
| ( [ table_name | alias ] column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

参数

`table_name` 表名.

`alias` 表别名.

`column_name` 列名.

`expression` 值表达式.

`sub-SELECT` select子句.

`from_list` 表表达式.

`condition` 一个返回boolean类型值的表达式.

`cursor_name` 要在WHERE CURRENT OF情况中使用的游标 的名称.

`output_expression` 在每一行被删除后，会被DELETE计算并且返回的表达式.

`output_name` 被返回列的名称.

示例

```
create table tb_test5(id int, flg char(10));

insert into tb_test5 values(1, '2'), (3, '4'), (5, '6');

update tb_test5 a set a.id = 33 where a.id = 3;

table tb_test5;
Id  |    flg
----+------------
  1 | 2
  5 | 6
 33 | 4
(3 rows)
```

GROUP BY

示例

```
set compatible_mode to oracle;

create table students(student_id varchar(20) primary key ,
student_name varchar(40),
student_pid int);

select student_id,student_name from students group by student_id;
ERROR:  column "students.student_name" must appear in the GROUP BY clause or be used
in an aggregate function
```

UNION

示例

```
SELECT 100 AS value FROM DUAL UNION SELECT 200 AS value FROM DUAL UNION SELECT 100 AS
value FROM DUAL;
 value
-------
   100
   200
(2 rows)
```

Minus Operator

语法

```
select_statement MINUS [ ALL | DISTINCT ] select_statement;
```

参数

`select_statement` 任何没有ORDER BY、LIMIT、 FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE子句的 SELECT语句.

`ALL keyword` 包含重复行结果.

`DISTINCT keyword` 显示的消除重复行.

示例

```
select * from generate_series(1, 3) g(i) MINUS select * from generate_series(1, 3)
g(i) where i = 1;
 i
---
 2
 3
```

```
(2 rows)
```

转义字符

概述

使用 q\' 转义特殊字符。q\' 转义字符通常在! [] {} () \<> 和其他转义字符之后使用, 您也可以使用 \,
也可以使用字母, 数字, \=, +, -, *, \&, \$, \%, #, 等, 不允许使用空格。

示例

```
select q''' is goog '';
  ?column?
------------
 ' is goog
(1 row)
```

序列

语法

```
SELECT [ database {schema} | schema ] sequence {nextval | currval};
```

参数

sequence 序列名.

示例

```
create sequence sq;

select sq.nextval;
 nextval
-----------
      1
(1 row)

select sq.currval;
 nextval
-----------
      1
(1 row)
```

兼容时间和日期函数

from_tz

目的

将给定的不带时区的时间戳转换为指定的带时区的时间戳，如果指定时区或者时间戳为NULL，则返回NULL。

参数描述

参数	描述
day	不带时区的时间戳
tz	指定的时区

例子

```
select from_tz('2021-11-08 09:12:39','Asia/Shanghai') from dual;
              from_tz
-----------------------------------
 2021-11-08 09:12:39 Asia/Shanghai
(1 row)

select from_tz('2021-11-08 09:12:39','SAST') from dual;
         from_tz
--------------------------
 2021-11-08 09:12:39 SAST

select from_tz(NULL,'SAST') from dual;
 from_tz
---------

(1 row)

select from_tz('2021-11-08 09:12:31',NULL) from dual;
 from_tz
---------

(1 row)
```

systimestamp

目的

获取当前数据库系统的时间戳。

例子

```
select systimestamp();
         systimestamp
-------------------------------
 2021-12-02 14:38:59.879642+08
(1 row)

select systimestamp;
      statement_timestamp
-------------------------------
 2021-12-02 14:39:33.262828+08
```

sys_extract_utc

目的

将给定的带时区的时间戳转换为不带时区的UTC时间。

参数描述

参数	描述
day	需要转化带时区的时间戳

例子

```
select sys_extract_utc('2018-03-28 11:30:00.00 +09:00'::timestamptz) from dual;
   sys_extract_utc
---------------------
 2018-03-28 02:30:00
(1 row)

select sys_extract_utc(NULL) from dual;
 sys_extract_utc
-----------------

(1 row)
```

sessiontimezone

目的

获取当前会话的时区。

例子

```
select sessiontimezone() from dual;
 sessiontimezone
-----------------
 PRC
(1 row)

set timezone to UTC;

 select sessiontimezone();
 sessiontimezone
-----------------
 UTC
(1 row)
```

next_day

目的

```
next_day 返回由格式名相同的第一个工作日的日期，该日期晚于当前日期。
无论日期的数据类型如何，返回类型始终为 DATE。
返回值具有与参数日期相同的小时、分钟和秒部分。
```

参数描述

参数	描述
value	开始时间戳
weekday	星期几，可以是 "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" 或者 0,1,2,3,4,5,6,0代表星期天

例子

```
select next_day(to_timestamp('2020-02-29 14:40:50', 'YYYY-MM-DD HH24:MI:SS'),
'Tuesday') from dual;
      next_day
---------------------
 2020-03-03 14:40:50
(1 row)

select next_day('2020-07-01 19:43:51 +8'::timestamptz, 1) from dual;
      next_day
---------------------
 2020-07-05 19:43:51
```

```
(1 row)
```

last_day

目的

```
last_day返回档期日期所在月份的最后一天。
```

参数描述

参数	描述
value	指定的时间戳

例子

```
select last_day(timestamp '2020-05-17 13:27:19') from dual;
      last_day
---------------------
 2020-05-31 13:27:19
(1 row)

select last_day('2020-11-29 19:20:40 +08'::timestamptz) from dual;
      last_day
---------------------
 2020-11-30 19:20:40
(1 row)
```

add_months

目的

```
add_months 返回日期加上整数月份。 date 参数可以是日期时间值或任何可以隐式转换为 DATE 的值。 整数参数可以是整数或任何可以隐式转换为整数的值。
```

参数描述

参数	描述
day	oracle.date类型，需要被改变的时间戳
value	一个整形数据，需要增加的月数

例子

```
select add_months(date '2020-02-15',7) from dual;
     add_months
---------------------
```

```
 2020-09-15 00:00:00
(1 row)

select add_months(timestamp '2018-12-15 19:12:09',12) from dual;
     add_months
---------------------
 2019-12-15 19:12:09
(1 row)
```

sysdate

目的

```
sysdate 返回数据库服务器的操作系统时间。
```

例子

```
select sysdate;
  statement_sysdate
---------------------
 2021-12-09 16:20:34
(1 row)

select sysdate();
       sysdate
---------------------
 2021-12-09 16:21:39
(1 row)
```

new_time

目的

```
转换第一个时区的时间到第二个时区的时间. 时区包括了 "ast", "adt", "bst", "bdt", "cst",
"cdt", "est", "edt", "gmt", "hst", "hdt", "mst", "mdt", "nst", "pst", "pdt", "yst",
"ydt".
```

参数描述

参数	描述
day	需要被转换的时间戳
tz1	时间戳的时区
tz2	目标时区

例子

```
select new_time(timestamp '2020-12-12 17:45:18', 'AST', 'ADT') from dual;
      new_time
---------------------
 2020-12-12 18:45:18
(1 row)

select new_time(timestamp '2020-12-12 17:45:18', 'BST', 'BDT') from dual;
      new_time
---------------------
 2020-12-12 18:45:18
(1 row)

select new_time(timestamp '2020-12-12 17:45:18', 'CST', 'CDT') from dual;
      new_time
---------------------
 2020-12-12 18:45:18
(1 row)
```

trunc

目的

```
trunc函数返回一个日期，按照指定格式被截断，fmt包括 "Y", "YY", "YYY", "YYYY","YEAR",
"SYYYY", "SYEAR", "I", "IY", "IYY", "IYYY", "Q", "WW", "Iw", "W", "DAY", "DY", "D",
"MONTH", "MONn", "MM", "RM", "CC", "SCC", "DDD", "DD", "J", "HH", "HH12", "HH24",
"MI".
```

参数描述

参数	描述
value	指定的日期（oracle.date, timestamp, timestamptz）
fmt	指定的格式, 如果被省略, 默认为 "DDD"

例子

```
select trunc(timestamp '2020-07-28 19:16:12', 'Q');
        trunc
---------------------
 2020-07-01 00:00:00
(1 row)

select trunc(timestamptz '2020-09-27 18:30:21 + 08', 'MONTH');
         trunc
```

```
------------------------
 2020-09-01 00:00:00+08
(1 row)
```

round

目的

```
round函数返回一个日期,按照指定的格式四舍五入, fmt 包括了 "Y", "YY", "YYY",
"YYYY","YEAR", "SYYYY", "SYEAR", "I", "IY", "IYY", "IYYY", "Q", "WW", "Iw", "W",
"DAY", "DY", "D", "MONTH", "MONn", "MM", "RM", "CC", "SCC", "DDD", "DD", "J", "HH",
"HH12", "HH24", "MI".
```

参数描述

参数	描述
value	被转换的日期（oracle.date, timestamp, timestamptz）
fmt	指定的格式, 如果被省略, 默认为 "DDD"

例子

```
select round(timestamp '2050-06-12 16:40:55', 'IYYY');
        round
---------------------
 2050-01-03 00:00:00
(1 row)
```

兼容转换和比较以及与NULL相关的函数

TO_CHAR

目的

TO_CHAR（str,[fmt]） 根据给定的格式将输入参数转换为 TEXT 数据类型的值。 如果省略 fmt，则数据将转换为系统默认格式的 TEXT 值。 如果 str 为 null，则该函数返回 null。

参数

`str` 输入参数 （任意类型）。

`fmt` 输入格式参数,详见格式fmt。

示例

```
select to_char('3 2:20:05');
     to_char
 -----------------
  3 days 02:20:05
```

```
 (1 row)

select to_char('4.00'::numeric);
  to_char
 ---------
  4
 (1 row)

select to_char(NULL);
  to_char
 ---------

 (1 row)

select to_char(123,'xx');
  to_char
 ---------
  7b
 (1 row)
```

TO_NUMBER

目的

TO_NUMBER(str,[fmt1]) 根据给定的格式将输入参数 str 转换为 NUMREIC 数据类型的值。 如果省略 fmt1，则数据将转换为系统默认格式的 NUMERIC 值。 如果 str 是 NUMERIC，则该函数返回 str。如果 str 计算结果为 null，则该函数返回 null。 如果它不能转换为 NUMERIC 数据类型，则该函数返回错误。

参数

`str` 输入参数包括以下数据类型（double precision，numeric，text，integer等，但必须隐式转换为numeric）。

`fmt1` 输入格式参数，详见格式fmt1。

示例

```
select to_number(1210.73::numeric, 9999.99::numeric);
 to_number
-----------
   1210.73
(1 row)

select to_number(NULL);
 to_number
-----------
```

```
(1 row)

select to_number('123'::text);
 to_number
-----------
   123
(1 row)
```

TO_DATE

目的

TO_DATE(str,[fmt]) 根据给定的格式将输入参数 str 转换为日期数据类型的值。 如果省略 fmt，则数据将转换为系统默认格式的日期值。 如果 str 为 null，则该函数返回 null。 如果 fmt 是 J，对于 Julian，则 char 必须是整数。 如果无法转换为 DATE，则该函数返回错误。

参数

str 输入参数（integer，text，可以隐式转换为上述类型，符合日期格式的字符串）。

fmt 输入格式参数，详见格式fmt。

示例

```
select to_date('50-11-28 ','RR-MM-dd ');
       to_date
---------------------
 1950-11-28 00:00:00
(1 row)

select to_date(2454336, 'J');
       to_date
---------------------
 2007-08-23 00:00:00
(1 row)

select to_date('2019/11/22', 'yyyy-mm-dd');
       to_date
---------------------
 2019-11-22 00:00:00
(1 row)

select to_date('20-11-28 10:14:22','YY-MM-dd hh24:mi:ss');
       to_date
---------------------
 2020-11-28 10:14:22
(1 row)
```

```
select to_date('2019/11/22');
       to_date
---------------------
 2019-11-22 00:00:00
(1 row)

select to_date('2019/11/27 10:14:22');
       to_date
---------------------
 2019-11-27 10:14:22
(1 row)

select to_date('2020','RR');
       to_date
---------------------
 2020-01-01 00:00:00
(1 row)

select to_date(NULL);
 to_date
---------

(1 row)

select to_date('-4712-07-23 14:31:23', 'syyyy-mm-dd hh24:mi:ss');
       to_date
----------------------
 -4712-07-23 14:31:23
(1 row)
```

TO_TIMESTAMP

目的

TO_TIMESTAMP(str,[fmt]) 根据给定的格式将输入参数 str 转换为不带时区的时间戳。 如果省略 fmt，则数据将转换为系统默认格式中不带时区值的时间戳。 如果 str 为 null，则该函数返回 null。 如果无法转换为不带时区的时间戳，则该函数返回错误。

参数

`str` 输入参数（double precision,text，可以隐式转换为上述类型）。

`fmt` 输入格式参数，详见格式fmt。

示例

```
select to_timestamp(1212121212.55::numeric);
       to_timestamp
----------------------------
 2008-05-30 12:20:12.55
(1 row)

select to_timestamp('2020/03/03 10:13:18 +5:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM');
      to_timestamp
------------------------
 2020-03-03 13:13:18
(1 row)

select to_timestamp(NULL,NULL);
 to_timestamp
--------------

(1 row)
```

TO_YMINTERVAL

目的

TO_YMINTERVAL(str) 将输入参数 str 时间间隔转换为年到月范围内的时间间隔。
只处理年月，其他部分省略。
如果输入参数为NULL，函数返回NULL，如果输入参数格式错误，函数返回错误。

参数

str 输入参数（text，可以隐式转换为文本类型，必须是时间间隔格式。 兼容 SQL 标准的 SQL 间隔格式，ISO 持续时间格式与 ISO 8601:2004 标准兼容）。

示例

```
select to_yminterval('P1Y-2M2D');
 to_yminterval
---------------
 +000000000-10
(1 row)

select to_yminterval('P1Y2M2D');
 to_yminterval
---------------
 +000000001-02
(1 row)
```

```
select to_yminterval('-01-02');
 to_yminterval
---------------
 -000000001-02
(1 row)
```

TO_DSINTERVAL

目的

TO_DSINTERVAL(str) 将输入参数 str 的时间间隔转换为天到秒范围内的时间间隔。
输入参数包括：日、时、分、秒和微秒。
如果输入参数为NULL，函数返回NULL，如果输入参数包含年月或格式错误，函数返回错误。

参数

str 输入参数（text，可以隐式转换为文本类型，必须是时间间隔格式。 兼容 SQL 标准的 SQL 间隔格式，ISO 持续时间格式与 ISO 8601:2004 标准兼容）。

示例

```
select to_dsinterval('100 00 :02 :00');
         to_dsinterval
-------------------------------
 +000000100 00:02:00.000000000
(1 row)

select to_dsinterval('-100 00:02:00');
         to_dsinterval
-------------------------------
 -000000100 00:02:00.000000000
(1 row)

select to_dsinterval(NULL);
 to_dsinterval
---------------

(1 row)
```

TO_TIMESTAMP_TZ

目的

TO_TIMESTAMP_TZ(str,[fmt]) 根据给定的格式将输入参数 str 转换为带时区的时间戳。 如果省略 fmt，则数据将转换为具有系统默认格式带时区值的时间戳。 如果 str 为 null，则该函数返回 null。
如果无法转换为带时区的时间戳，则该函数返回错误。

参数

str 输入参数（text，可以隐式转换为文本类型）。

`fmt` 输入格式参数，详见格式fmt。

示例

```
select to_timestamp_tz('2019','yyyy');
    to_timestamp_tz
------------------------
 2019-01-01 00:00:00+08
(1 row)

select to_timestamp_tz('2019-11','yyyy-mm');
    to_timestamp_tz
------------------------
 2019-11-01 00:00:00+08
(1 row)

select to_timestamp_tz('2003/12/13 10:13:18 +7:00');
    to_timestamp_tz
------------------------
 2003-12-13 11:13:18+08
(1 row)

select to_timestamp_tz('2019/12/13 10:13:18 +5:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM');
    to_timestamp_tz
------------------------
 2019-12-13 13:13:18+08
(1 row)

select to_timestamp_tz(NULL);
 to_timestamp_tz
-----------------

(1 row)
```

GREATEST

目的

GREATEST(expr1,expr2,…) 获取一个或多个表达式的输入列表中的最大值。 如果任何 expr 的计算结果为 NULL，则该函数返回 NULL。

参数

```
expr1` 输入参数（任意类型）。
`expr2` 输入参数（任意类型）。
```

```
`...
```

示例

```
select greatest('a','b','A','B');
 greatest
----------
 b
(1 row)

select greatest(',','.','/',';','!','@','?');
 greatest
----------
 @
(1 row)

select greatest('瀚','高','数','据','库');
 greatest
----------
 高
(1 row)

SELECT greatest('HARRY', 'HARRIOT', 'HARRA');
 greatest
----------
 HARRY
(1 row)

SELECT greatest('HARRY', 'HARRIOT', NULL);
 greatest
----------

(1 row)

SELECT greatest(1.1, 2.22, 3.33);
 greatest
----------
     3.33
(1 row)

SELECT greatest('A', 6, 7, 5000, 'E', 'F','G') A;
 a
```

```
---
 G
(1 row)
```

LEAST

目的

LEAST(expr1,expr2,…) 获取一个或多个表达式的输入列表中的最小值。 如果任何 expr 的计算结果为 NULL，则该函数返回 NULL。

参数

```
expr1` 输入参数（任意类型）。
`expr2` 输入参数（任意类型）。
`...
```

示例

```
SELECT least(1,' 2', '3' );
 least
-------
     1
(1 row)

SELECT least(NULL, NULL, NULL);
 least
-------

(1 row)

SELECT least('A', 6, 7, 5000, 'E', 'F','G') A;
  a
------
 5000
(1 row)

select least(1,3,5,10);
 least
-------
     1
(1 row)

select least('a','A','b','B');
```

```
 least
-------
 A
(1 row)

select least(',','.','/',';','!','@');
 least
-------
 !
(1 row)

select least('瀚','高','据','库');
 least
-------
 库
(1 row)

SELECT least('HARRY', 'HARRIOT', NULL);
 least
-------

(1 row)
```

fmt（日期/时间格式的模板模式）

模式	描述
HH	一天中的小时 （01-12）
HH12	一天中的小时 （01-12）
HH24	一天中的小时 （00-23）
MI	分钟 （00-59）minute (00-59)
SS	秒（00-59）
MS	毫秒（000-999）
US	微秒（000000-999999）
SSSS	午夜后的秒（0-86399）
AM, am, PM or pm	正午指示器（不带句号）
A.M., a.m., P.M. or p.m.	正午指示器（带句号）
Y,YYY	带逗号的年（4 位或者更多位）
YYYY	年（4 位或者更多位）
YYY	年的后三位
YY	年的后两位
Y	年的最后一位
IYYY	ISO 8601 周编号方式的年（4 位或更多位）
IYY	ISO 8601 周编号方式的年的最后 3 位

IY	ISO 8601 周编号方式的年的最后 2 位
I	ISO 8601 周编号方式的年的最后一位
BC, bc, AD或者ad	纪元指示器（不带句号）
B.C., b.c., A.D.或者a.d.	纪元指示器（带句号）
MONTH	全大写形式的月名（空格补齐到 9 字符）
Month	全首字母大写形式的月名（空格补齐到 9 字符）
month	全小写形式的月名（空格补齐到 9 字符）
MON	简写的大写形式的月名（英文 3 字符，本地化长度可变）
Mon	简写的首字母大写形式的月名（英文 3 字符，本地化长度可变）
mon	简写的小写形式的月名（英文 3 字符，本地化长度可变）
MM	月编号（01-12）
DAY	全大写形式的日名（空格补齐到 9 字符）
Day	全首字母大写形式的日名（空格补齐到 9 字符）
day	全小写形式的日名（空格补齐到 9 字符）
DY	简写的大写形式的日名（英语 3 字符，本地化长度可变）
Dy	简写的首字母大写形式的日名（英语 3 字符，本地化长度可变）
dy	简写的小写形式的日名（英语 3 字符，本地化长度可变）
DDD	一年中的日（001-366）
IDDD	ISO 8601 周编号方式的年中的日（001-371，年的第 1 日时第一个 ISO 周的周一）
DD	月中的日（01-31）
D	周中的日，周日（1）到周六（7）
ID	周中的 ISO 8601 日，周一（1）到周日（7）
W	月中的周（1-5）（第一周从该月的第一天开始）
WW	年中的周数（1-53）（第一周从该年的第一天开始）
IW	ISO 8601 周编号方式的年中的周数（01 - 53；新的一年的第一个周四在第一周）
CC	世纪（2 位数）（21 世纪开始于 2001-01-01）
J	儒略日（从午夜 UTC 的公元前 4714 年 11 月 24 日开始的整数日数）
Q	季度（to_date和to_timestamp会忽略）
RM	大写形式的罗马计数法的月（I-XII；I 是 一月）
rm	小写形式的罗马计数法的月（i-xii；i 是 一月）
TZ	大写形式的时区名称
tz	小写形式的时区名称
OF	时区偏移量

fmt1（数字格式的模板模式）

模式	描述

9	带有指定位数的值
0	带前导零的值
. (period)	小数点
, (comma)	分组（千）分隔符
PR	尖括号内的负值
S	带符号的数字（使用区域）
L	货币符号（使用区域）
D	小数点（使用区域）
G	分组分隔符（使用区域）
MI	在指定位置的负号（如果数字 < 0）
PL	在指定位置的正号（如果数字 > 0）
SG	在指定位置的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH or th	序数后缀
V	移动指定位数（参阅注解）
EEEE	科学记数的指数

NLS_LENGTH_SEMANTICS参数

概述

NLS_LENGTH_SEMANTICS 使您能够使用字节或字符长度语义创建 CHAR 和 VARCHAR2 列。
现有列不受影响。 在这种情况下，默认语义是 BYTE。

语法

```
SET NLS_LENGTH_SEMANTICS TO [NONE | BYTE | CHAR];
```

取值范围说明

```
BYTE:数据以字节长度来存储。
CHAR:数据以字符长度来存储。
NONE:数据使用原生PostgreSQL存储方式。
```

用例

--测试“CHAR”

```
create table test(a varchar2(5));
CREATE TABLE

SET NLS_LENGTH_SEMANTICS TO CHAR;
SET

SHOW NLS_LENGTH_SEMANTICS;
```

```
 nls_length_semantics
----------------------
 char
(1 row)

insert into test values ('李老师您好');
INSERT 0 1
```

--测试“BYTE”

```
SET NLS_LENGTH_SEMANTICS TO BYTE;
SET

SHOW NLS_LENGTH_SEMANTICS;
 nls_length_semantics
----------------------
 byte
(1 row)

insert into test values ('李老师您好');
2021-12-14 15:28:11.906 HKT [6774] ERROR:  value too long for type varchar2(5 byte)
2021-12-14 15:28:11.906 HKT [6774] STATEMENT:  insert into test values ('李老师您好');
ERROR:  value too long for type varchar2(5 byte)
```

VARCHAR2(size)类型

概述

具有最大长度大小字节或字符的可变长度字符串。 您必须为 VARCHAR2 指定大小。 最小大小为 1 个字节或 1 个字符。

语法

```
VARCHAR2(size)
```

用例

```
create table test(a varchar2(5));
CREATE TABLE

SET NLS_LENGTH_SEMANTICS TO CHAR;
SET

SHOW NLS_LENGTH_SEMANTICS;
```

```
 nls_length_semantics
----------------------
 char
(1 row)


insert into test values ('李老师您好');
INSERT 0 1
```

PL/iSQL

PL/iSQL 是 IvorySQL 的过程语言，用于为 IvorySQL 编写自定义函数、过程和包。 PL/iSQL 派生自 PostgreSQL 的 PL/pgsql，并增加了一些功能，但在语法上 PL/iSQL 更接近 Oracle 的 PL/SQL。
本文档描述了 PL/iSQL 程序的基本结构和构造。

PL/iSQL 程序的结构

iSQL 是一种程序化的块结构语言，支持四种不同的 程序类型，即 PACKAGES、PROCEDURES、FUNCTIONS 和 TRIGGERS。 iSQL 对每种类型的受支持程序使用相同的块结构。
一个块最多由三个部分组成：声明部分，可执行文件，和异常部分。 而声明和异常部分是可选的。

```
[DECLARE
      declarations]
    BEGIN
      statements
  [ EXCEPTION
      WHEN <exception_condition> THEN
        statements]
    END;
```

一个块至少可以由一个可执行部分组成 在 BEGIN 和 END 关键字中包含一个或多个 iSQL 语句。

```
CREATE OR REPLACE FUNCTION null_func() RETURN VOID AS
BEGIN
    NULL;
END;
/
```

所有关键字都不区分大小写。 标识符被隐式转换为小写，除非双引号， 就像它们在普通 SQL 命令中一样。
声明部分可用于声明变量和游标，并取决于使用块的上下文， 声明部分可以以关键字 DECLARE 开头。

```
CREATE OR REPLACE FUNCTION null_func() RETURN VOID AS
DECLARE
    quantity integer := 30;
    c_row pg_class%ROWTYPE;
    r_cursor refcursor;
    CURSOR c1 RETURN pg_proc%ROWTYPE;
```

```
BEGIN
    NULL;
end;
/
```

可选的异常部分也可以包含在 BEGIN - END 块中。 异常部分以关键字 EXCEPTION
开始，一直持续到它出现的块的末尾。
如果块内的语句抛出异常，程序控制转到异常部分，根据异常和异常部分的内容，可能会或不会处理抛出的异常。

```
CREATE OR REPLACE FUNCTION reraise_test() RETURN void AS
BEGIN

    BEGIN
        RAISE syntax_error;
        EXCEPTION
            WHEN syntax_error THEN

            BEGIN
                raise notice 'exception % thrown in inner block, reraising', sqlerrm;
                RAISE;
                EXCEPTION
                WHEN OTHERS THEN
                    raise notice 'RIGHT - exception % caught in inner block', sqlerrm;
                END;
            END;
    EXCEPTION
        WHEN OTHERS THEN
        raise notice 'WRONG - exception % caught in outer block', sqlerrm;
END;
/
```

注意

与 PL/pgSQL 类似，PL/iSQL 使用 BEGIN/END 对语句进行分组，
并且不要将它们与用于事务控制的同名 SQL 命令混淆。 PL/iSQL 的 BEGIN/END 仅用于分组；
他们不开始或结束事务

psql 对 PL/iSQL 程序的支持

要从 psql 客户端创建 PL/iSQL 程序，您可以使用类似于 PL/pgSQL 的$$语法

```
CREATE FUNCTION func() RETURNS void as
$$
..
```

```
end$$ language plisql;
```

或者，您可以使用不带 $$ 的 Oracle 兼容语法的引用和语言规范， 并使用 /（正斜杠） 结束程序定义。/（正斜杠）必须在换行符上

```
CREATE FUNCTION func() RETURN void AS
...
END;
/
```

PL/iSQL 程序语法

PROCEDURES

```
CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter_list)]
is
[DECLARE]
    -- variable declaration
BEGIN
    -- stored procedure body
END;
/
```

FUNCTIONS

```
CREATE [OR REPLACE] FUNCTION function_name ([parameter_list])
RETURN return_type AS
[DECLARE]
    -- variable declaration
BEGIN
    -- function body
   return statement
END;
/
```

PACKAGES

PACKAGE HEADER

```
CREATE [ OR REPLACE ] PACKAGE [schema.] *package_name* [invoker_rights_clause] [IS |
AS]
   item_list[, item_list ...]
END [*package_name*];
```

```
invoker_rights_clause:
     AUTHID [CURRENT_USER | DEFINER]

item_list:
[
   function_declaration    |
   procedure_declaration   |
   type_definition         |
   cursor_declaration      |
   item_declaration
]


function_declaration:
     FUNCTION function_name [(parameter_declaration[, ...])] RETURN datatype;

procedure_declaration:
     PROCEDURE procedure_name [(parameter_declaration[, ...])]

type_definition:
     record_type_definition      |
     ref_cursor_type_definition

cursor_declaration:
   CURSOR name [(cur_param_decl[, ...])] RETURN rowtype;

item_declaration:
     cursor_declaration            |
     cursor_variable_declaration   |
     record_variable_declaration   |
     variable_declaration          |

record_type_definition:
   TYPE record_type IS RECORD  ( variable_declaration [, variable_declaration]... ) ;

ref_cursor_type_definition:
   TYPE type IS REF CURSOR [ RETURN type%ROWTYPE ];

cursor_variable_declaration:
   curvar curtype;
```

```
record_variable_declaration:
   recvar { record_type | rowtype_attribute | record_type%TYPE };

variable_declaration:
   varname datatype [ [ NOT NULL ] := expr ]

parameter_declaration:
   parameter_name [IN] datatype [[:= | DEFAULT] expr]
```

PACKAGE BODY

```
CREATE [ OR REPLACE ] PACKAGE BODY [schema.] package_name [IS | AS]
   [item_list[, item_list ...]] |
   item_list_2 [, item_list_2 ...]
   [initialize_section]
END [package_name];


initialize_section:
   BEGIN statement[, ...]

item_list:
[
   function_declaration    |
   procedure_declaration   |
   type_definition         |
   cursor_declaration      |
   item_declaration
]

item_list_2:
[
   function_declaration
   function_definition
   procedure_declaration
   procedure_definition
   cursor_definition
]

function_definition:
   FUNCTION function_name [(parameter_declaration[, ...])] RETURN datatype  [IS | AS]
   [declare_section] body;
```

```
procedure_definition:
   PROCEDURE procedure_name [(parameter_declaration[, ...])] [IS | AS]
   [declare_section] body;

cursor_definition:
   CURSOR name [(cur_param_decl[, ...])] RETURN rowtype IS select_statement;

body:
   BEGIN statement[, ...] END [name];

statement:
   [<<LABEL>>] pl_statments[, ...];
```

层级查询

语法

```
 {
 CONNECT BY [ NOCYCLE ] [PRIOR] condition [AND [PRIOR] condition]... [ START WITH
condition ]
 | START WITH condition CONNECT BY [ NOCYCLE ] [PRIOR] condition [AND [PRIOR]
condition]...
 }
```

CONNECT BY 查询语法以 CONNECT BY
关键字开头，这些关键字定义了父行和子行之间的分层相互依赖关系。 必须通过在 CONNECT BY
子句的条件部分指定 PRIOR 关键字来进一步限定结果.

PRIOR PRIOR 关键字是一元运算符，它将前一行与当前行联系起来。
此关键字可以用在相等条件的左边或右边。

START WITH 该子句指定从哪一行开始层次结构。

NOCYCLE 无操作语句。 目前只有语法支持。 该子句表示即使存在循环也返回数据。

附加列

LEVEL 返回层次结构中当前行的级别，从根节点的 1 开始，之后每级别递增 1。

CONNECT_BY_ROOT expr 返回层次结构中当前行的父列。

SYS_CONNECT_BY_PATH(col, chr) 它是一个返回从根到当前节点的列值的函数，由字符“chr”分隔。

限制

目前此功能有以下限制:

- 附加列可用于大多数表达式，如函数调用、CASE 语句和通用表达式，但有一些不受支持的列，如 ROW、TYPECAST、COLLATE、GROUPING 子句等
- 两个或多个列相同的情况下，可能需要输出列名，例如

```
> SELECT CONNECT_BY_ROOT col AS "col1", CONNECT_BY_ROOT col AS "col2" ....
```

- 不支持间接运算符或“*”
- 不支持循环检测（Loop detection）

全局唯一索引

创建全局唯一索引

语法

```
CREATE UNIQUE INDEX [IF NOT EXISTS] name ON table_name [USING method] (columns) GLOBAL
```

示例

```
CREATE UNIQUE INDEX myglobalindex on mytable(bid) GLOBAL;
```

全局唯一性保证

在创建全局唯一索引期间，系统会对所有现有分区执行索引扫描，如果发现来自其他分区的重复项而不是当前分区，则会引发错误。例如：

命令

```
create table gidxpart (a int, b int, c text) partition by range (a);
create table gidxpart1 partition of gidxpart for values from (0) to (100000);
create table gidxpart2 partition of gidxpart for values from (100000) to (199999);
insert into gidxpart (a, b, c) values (42, 572814, 'inserted first on gidxpart1');
insert into gidxpart (a, b, c) values (150000, 572814, 'inserted second on
gidxpart2');
create unique index on gidxpart (b) global;
```

输出

```
ERROR:  could not create unique index "gidxpart1_b_idx"
DETAIL:  Key (b)=(572814) is duplicated.
```

插入和更新

插入和更新的全局唯一性保证

在全局唯一索引创建过程中，系统会对所有现有分区执行索引扫描，如果在其他分区而不是当前分区中发现重复项，则会引发错误。

示例

命令

```
create table gidx_part (a int, b int, c text) partition by range (a);
create table gidxpart (a int, b int, c text) partition by range (a);
create table gidxpart1 partition of gidxpart for values from (0) to (10);
create table gidxpart2 partition of gidxpart for values from (10) to (100);
create unique index gidx_u on gidxpart using btree(b) global;

insert into gidxpart values (1, 1, 'first');
insert into gidxpart values (11, 11, 'eleventh');
insert into gidxpart values (2, 11, 'duplicated (b)=(11) on other partition');
```

输出

```
ERROR:  duplicate key value violates unique constraint "gidxpart2_b_idx"
DETAIL:  Key (b)=(11) already exists.
```

附加和分离

附加语句的全球唯一性保证

将新表附加到具有全局唯一索引的分区表时，系统将对所有现有分区进行重复检查。
如果在现有分区中发现与附加表中的元组匹配的重复项，则会引发错误并且附加失败。

附加需要所有现有分区上的共享锁（SHARE LOCK）。 如果其中一个分区正在进行并发
INSERT，则附加将等待它先完成。 这可以在未来的版本中改进

示例

运行命令

```
create table gidxpart (a int, b int, c text) partition by range (a);
create table gidxpart1 partition of gidxpart for values from (0) to (100000);
insert into gidxpart (a, b, c) values (42, 572814, 'inserted first on gidxpart1');
create unique index on gidxpart (b) global;
create table gidxpart2 (a int, b int, c text);
insert into gidxpart2 (a, b, c) values (150000, 572814, 'dup inserted on gidxpart2');

alter table gidxpart attach partition gidxpart2 for values from (100000) to (199999);
```

输出

```
ERROR:  could not create unique index "gidxpart1_b_idx"
DETAIL:  Key (b)=(572814) is duplicated.
```

IvorySQL架构设计

查询处理

...1. 双parser

双parser框架主要分为两部分，包括SQL端和服务器端编程语言。

SQL端词法语法分离

基本做法是新增一套兼容Oracle风格的语法和词法，在开启Oracle兼容的情况下，走Oracle风格的语法分析，生成相应的语法树。

具体方法：
在src/backend/下面，新建一个oracle_parser目录，将src/backend/parser/下的scan.l和gram.y复制到该目录下，改名成ora_gram.y和ora_scan.l，添加Oracle风格的语法和词法分析代码，同时复制keywords.c到该目录下，用来存放自己的关键字。该oracle_parser目录编译成一个动态库libparser_oracle.so。当开启Oracle兼容的时候，配置文件ivorysql.conf被嵌入到postgresql.conf文件的末尾。配置文件ivorysql.conf中的shared_preload_libraries参数中添加“liboracle_parser”，这样当数据库启动时能够自动载入liboracle_parser动态库。

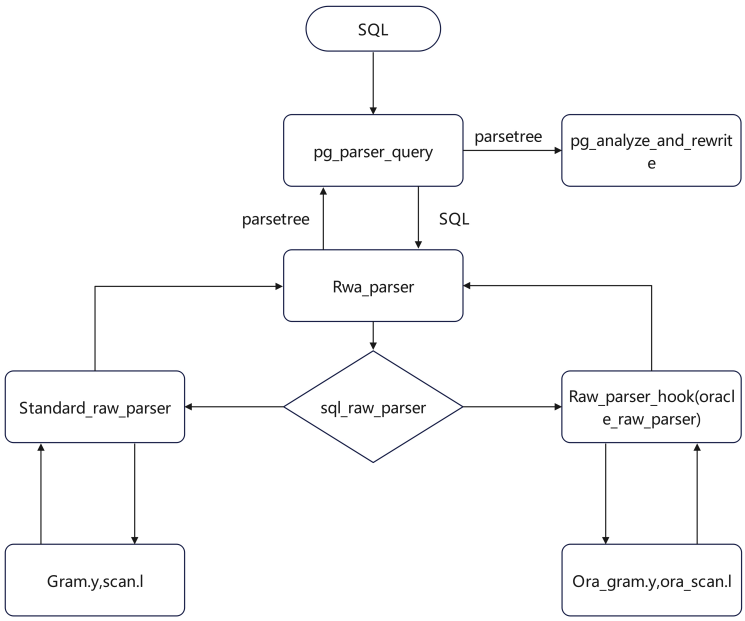
新增ora_raw_parser 函数指针，当libparser_oracle.so动态库被加载时，该动态库中的 _PG_init() 函数将oracle_raw_parser() 函数的地址赋值给 ora_raw_parser，_PG_fini()则在兼容模式切换时负责重置ora_raw_parser 为空。

每一个后端进程调用BackendInitialize()函数，通过后端进程所连接的端口号来设置port→connmode。如果端口是Oracle兼容端口，则设置connmode为’o’，否则设置为’p’。

这之后PostgresMain()调用InitIvorysql()，如果port→connmode是’o’，则调用函数SetConfigOption("ivorysql.compatible_mode", "oracle", PGC_USERSET, PGC_S_OVERRIDE);
由于这个参数设置了 assign_hook，因此当 SetConfigOption() 函数里执行assign_hook()时，相当于调用assign_compatible_mode()，从而设置 sql_raw_parser = ora_raw_parser;

在对SQL语句进行分析时，函数pg_parse_query()→raw_parser() 通过函数指针 sql_raw_parser 调用standard parser() 或者 ora_raw_parser()。

下面的图演示了SQL语句分析时发生的事情。



服务器端编程语言词法语法分离

在系统表pg_language中新增pliSQL语言。

具体实现方法：
将PG源码中的plpgsql目录，复制一份，改名为plisql，里面的文件名称，改成plisql开头，因为plpgsql是一个language，改造的plisql也是一个language，因此，plpgsql language的注册函数如plpgsql_validator, plpgsql_call_handler,
plpgsql_inline_handler都需要改成plisql开头，里面其他函数的改名都改成以plisql开头。

plisql目录构建为一个插件，initdb时如果数据库模式是Oracle，则创建这个插件。这个插件会将pliSQL语言注册到数据库的系统表中。

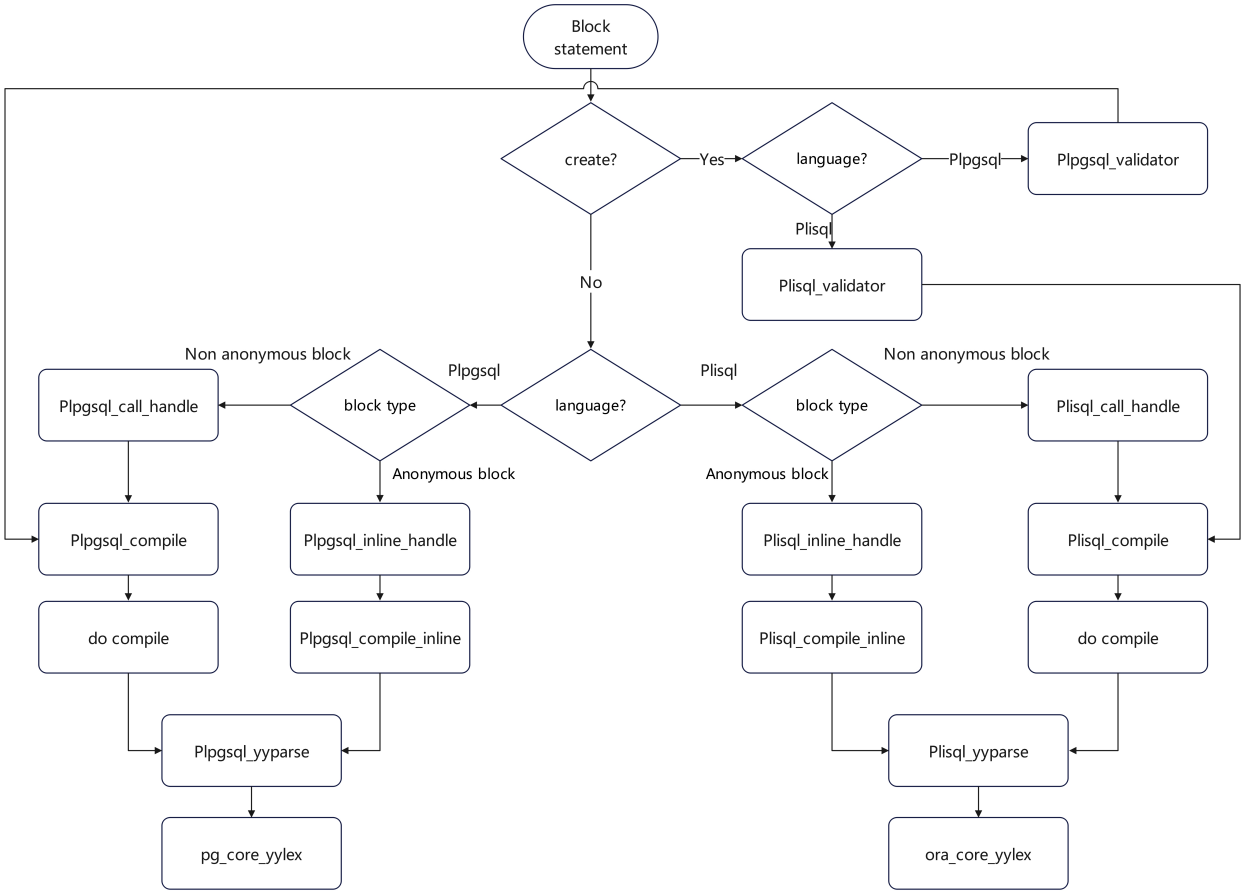
pliSQL端没有自己的词法规约，它依赖SQL端的词法规约，因此，针对pliSQL，强制其走兼容Oracle的词法规约，主要改造点是在 plisql_scanner_init 函数中，里面应该调用ora_scanner_init()函数，plisql目录中的internal_yylex() 函数调用 ora_core_yylex。

pliSQL语法规则在plisql/src/pl_gram.y中，兼容Oracle plSQL块的语法规则都在这个模块中实现。

SQL端创建函数，没有指定language的时候，如果当前是兼容Oracle模式，则默认language是plisql，如果是兼容pg模式，则默认language是plpgsql。oracle_parser中的ora_gram.y默认是plisql，pg parser中的gram.y默认是plpgsql。

匿名块没有指定language的时候，如果是兼容Oracle模式，默认plisql，如果是兼容pg模式，则默认plpgsql。

处理过程的ExecuteDoStmt函数同样根据兼容模式来决定默认language。

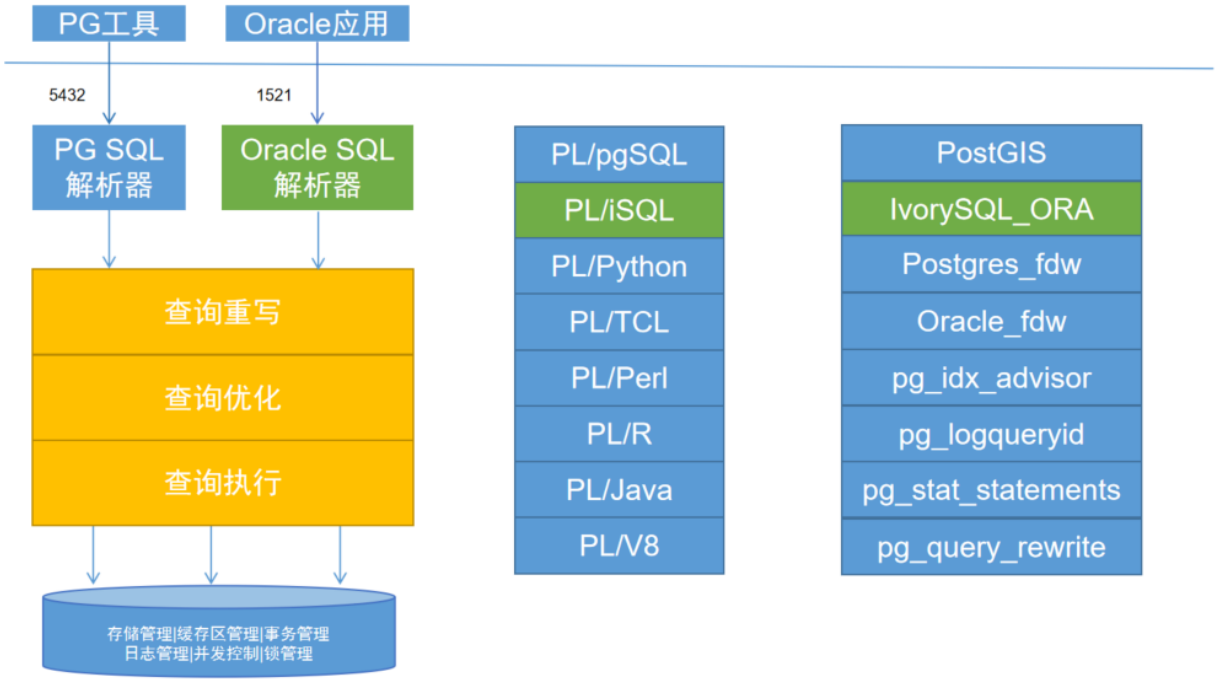


兼容框架

...1. 框架设计

目的

- 在对原有的PostgreSQL改动最小的前提下，实现对Oracle兼容。我们需要实现双parser、双端口、模式PL/PGSQL实现PL/iSQL的框架。实现流程图如下：



功能

双端口设计

- 保持了Ivorysql 5432端口兼容原有postgres情况，因此Ivorysql采用另一个独立的端口登录，默认为1521。从该端口登录，默认采用Oracle兼容模式。如果需要从5432端口登录且还要进入兼容模式，则需要通过`SET ivorysql.compatible_mode=oracle`进行设置。

parser模块设计

- 为了将Oracle语法与PostgreSQL语法间的相互干扰降到最低，因此新增parser模块，用于处理Oracle相关的语法。

新增PL/iSQL过程语言

- 新增PL/iSQL过程语言兼容Oracle的PL/SQL过程语言。

...2. GUC框架

新增GUC变量

为了兼容Oracle，需要在原有的GUC变量基础之上增加一些用于控制数据库执行结果的变量，以达到和Oracle行为一致的目的。

为了更好地添加兼容的guc参数，以及为了更少地改动pg内核源码，我们需要设计一个框架将guc添加到一个统一的地方。

实现

添加兼容版本的guc参数时，我们需统一在 ivy_guc.c 文件中添加。其中 `Ivy_ConfigureNamesBool`、`Ivy_ConfigureNamesInt`、`Ivy_ConfigureNamesString`、`Ivy_ConfigureNamesReal` 和 `Ivy_ConfigureNamesEnum`
分别表示5种不同类型的guc参数，添加guc参数时，将guc的值添加到对应数组中即可。

新增变量（目前）

变量名	描述
ivorysql.compatible_mode	表示当前兼容哪种数据库（pg/oracle），可以通过show命令查看，set命令更改该变量，reset命令重置为连接时的数据库模式，reset all会影响该变量
ivorysql.database_mode	表示当前数据库的模式（pg/oracle），可以通过show命令查看，set/reset/reset all命令不影响该变量
ivorysql.datetime_ignore_nls_mask	表示日期格式是否会受NLS参数影响，默认为0，可以通过set命令设置，reset 命令重置，reset all命令会重置该变量
ivorysql.enable_emptystring_to_NULL	取值为（on/off）,该变量为on时，会将插入的空字符串转成NULL值存储
ivorysql.identifier_case_switch	设置字符大小写转换模式
ivorysql.listen_address	表示兼容模式监听的地址，在初始化数据库时，从ivorysql.conf文件中读取该配置，在配置文件中修改该值，需要重启数据库生效，可以通过show命令查看
ivorysql.port	表示兼容模式下连接的端口号，在初始化数据库时，从ivorysql.conf文件中读取该配置，在配置文件中修改该值，需要重启数据库生效，可以通过show命令查看
nls_date_format	表示默认的日期格式，可以通过show命令查看，默认为‘YYYY-MM-DD’，可以通过set命令设置，可以通过reset命令重置回默认值，reset all 命令会重置该变量
nls_length_semantics	兼容Oracle的同名参数，控制一个字符所占内存的大小
nls_timestamp_format	兼容Oracle的同名参数，控制带时间的日期格式
nls_timestamp_tz_format	兼容Oracle的同名参数，控制带时区的日期格式
shared_preload_libraries	在初始化数据库时，从ivorysql.conf文件中读取，可以通过show命令查看，在配置文件中修改该值，重启数据库生效。

部分变量使用示例

ivorysql.datetime_ignore_nls_mask

该GUC变量的可选值为0-15

可选值	不经过nls格式化的类型
0	不屏蔽任何类型，所有时间格式均由nls格式化
1	date
2	timestamp
3	date、timestamp
4	timestamptz
5	date、timestamptz
6	timestamp、timestamptz
7	date、timestamp、timestamptz
8	timestampltz
9	date、timestampltz
10	timestamp、timestampltz
11	date、timestamp、timestampltz
12	timestamptz、timestampltz
13	date、timestamptz、timestampltz

14	timestamp、timestamptz、timestampltz
15	date、timestamp、timestamptz、timestampltz

- 使用方法(以date为例)

查看nls date 的格式及datetime_ignore_nls_mask值

```
ivorysql=# set ivorysql.compatible_mode to oracle;
SET
ivorysql=# show nls_date_format;
 nls_date_format
-----------------
 YYYY-MM-DD
(1 row)
ivorysql=# show ivorysql.datetime_ignore_nls_mask;
 ivorysql.datetime_ignore_nls_mask
-----------------------------------
 0
(1 row)
```

创建测试表

```
ivorysql=# create table test_nls_date(a int, created_at date);
CREATE TABLE
```

插入数据

```
ivorysql=# insert into test_nls_date values(1, '2024/04/05');
INSERT 0 1
ivorysql=# select * from test_nls_date;
 a | created_at
---+------------
 1 | 2024-04-05
(1 row)
```

修改nls_date_format

```
ivorysql=# set nls_date_format to 'yy-mm-dd';
SET
```

插入nls格式数据并查看，插入成功

```
ivorysql=# insert into test_nls_date values(2, '24/04/15');
INSERT 0 1
```

```
ivorysql=# select * from test_nls_date;
 a | created_at
---+------------
 1 | 24-04-05
 2 | 24-04-15
(2 rows)
```

将date类型改为不经过nls处理，插入相同格式数据，如改成 1 (3,5,7等都可以)，插入数据将报错，nls格式化将不对date的查询结果起效。

```
ivorysql=# set ivorysql.datetime_ignore_nls_mask to 1;
SET
ivorysql=# insert into test_nls_date values(3, '24/05/15');
ERROR:  date/time field value out of range: "24/05/15"
LINE 1: insert into test_nls_date values(3, '24/05/15');
                                            ^
HINT:  Perhaps you need a different "datestyle" setting.
ivorysql=#  select * from test_nls_date;
 a | created_at
---+------------
 1 | 24-04-05
 2 | 24-04-15
(2 rows)
```

...3. 双模式设计

目的

- 为了满足PG模式和兼容Oracle模式， IvorySQL在initdb时，可以指定-m参数，来获取PG模式数据库或Oracle兼容模式数据库。

> 1. 不指定-m参数时，默认为Oracle兼容模式
> 2. 指定-m参数为pg时，数据库将不再兼容Oracle语法

功能

- initdb -m 初始化，需要判断不同的模式，其中Oracle模式下，需要执行postgres_oracle.bki的SQL语句;
- 启动时会根据初始化模式，判断是否为oracle兼容模式。

```
database_mode：用于表示初始化模式；
database_mode=DB_PG，PG模式，且不可切换；
database_mode=DB_ORACLE，Oracle兼容模式；
```

测试用例

```
初始化PG模式：
```

```
./initdb -D ../data -m pg

初始化oracle兼容模式:
./initdb -D ../data -m oracle
或
./initdb -D ../data
```

...4. initdb过程

IvorySQL 在初始化过程中支持两种数据库模式：

- PG 模式：保持与原生 PostgreSQL 的兼容性
- Oracle 模式（默认）：提供 Oracle 语法兼容及增强功能

用户可通过 initdb 命令参数指定初始化模式，实现不同场景下的兼容性需求。

参数解析处理

initdb在开始阶段会解析输入的命令行参数。

参数	功能描述	可选值	默认值
-m	指定数据库模式	oracle/pg	oracle
-C	设置 Oracle 兼容的大小写转换模式	interchange/normal/lowercase	interchange

参数解析流程：

1.继承 PostgreSQL 原有参数处理机制

2.新增模式选择参数 -m 的解析逻辑

3.增加大小写转换参数 -C 的处理模块

文件路径初始化

执行 setup_data_file_paths() 函数完成关键文件路径配置：

```
    if (DB_PG == database_mode)
        set_input(&bki_file, "postgres.bki");
    else
        set_input(&bki_file, "postgres_oracle.bki");
```

路径检查机制：

1.验证 BKI 文件的存在性（postgres_oracle.bki / postgres.bki）

2.确认配置文件模板的可用性

3.建立与数据库模式对应的系统目录结构

数据目录初始化流程

通过 initialize_data_directory() 函数完成核心初始化操作：

目录结构创建

调用 create_data_directory() 创建主数据目录（PGDATA）。

通过 create_xlog_or_symlink() 建立 WAL 日志目录。

循环创建 base, global 等标准子目录。

配置文件初始化

调用set_null_conf()创建空的postgresql.conf。

Oracle 模式下额外创建 ivorysql.conf 配置文件。

调用setup_config()将配置信息写入postgresql.conf。在写入配置时，如果是oracle模式，则会额外向ivorysql.conf中写入配置信息。

模板数据库引导

执行 bootstrap_template1() 加载对应模式的 BKI 文件初始化template1模板数据库，

IvorySQL会额外设置template1模板数据库的数据库模式(oracle/pg)和大小写转换模式。

load_plisql()：安装兼容 Oracle PL/SQL 的 PL/iSQL 过程语言

load_ivorysql_ora()：加载核心Oracle兼容层扩展

make_ivorysql()：创建默认的 ivorysql 数据库

Oracle兼容用户名处理

当数据库模式是oracle时，会对用户名进行强制小写转换

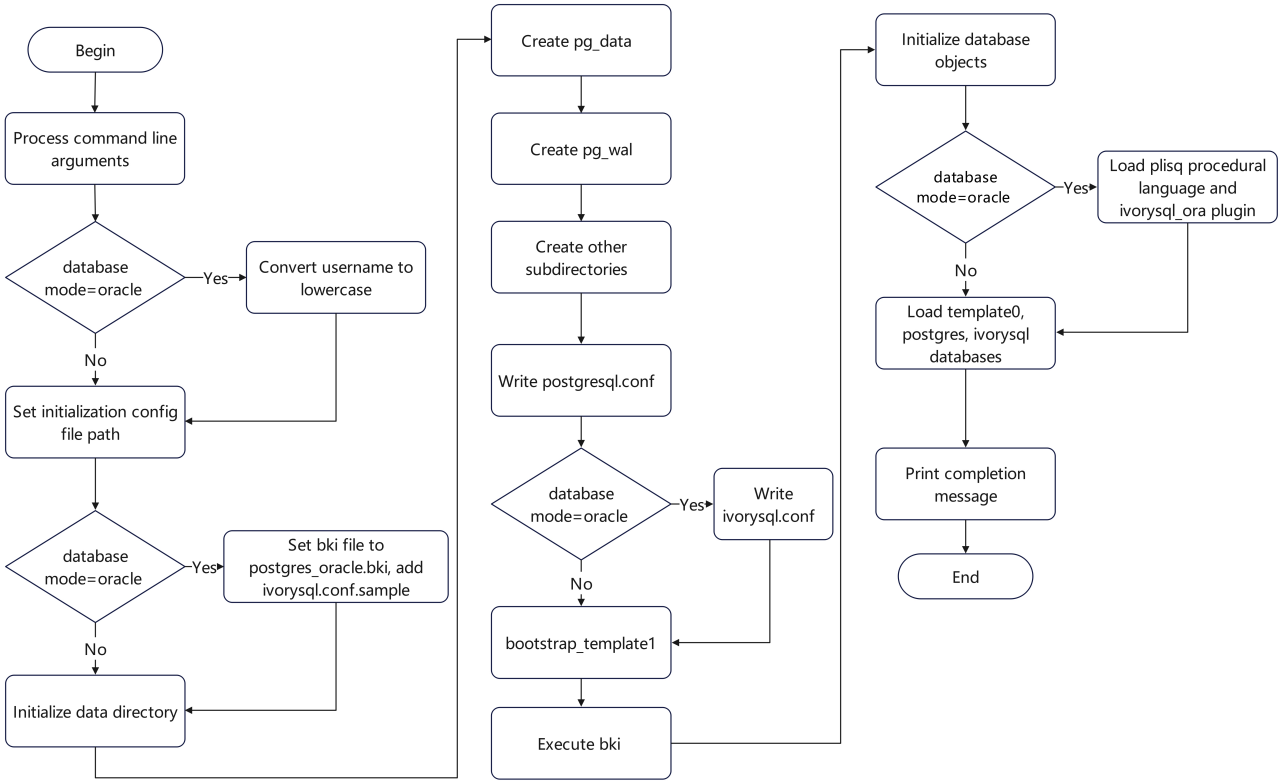
提示

在 PostgreSQL 中，"bki" 是 "Bootstrap Kit" 的简写。Bootstrap Kit 是 PostgreSQL 内部用于引导（bootstrap）数据库系统的一组工具和文件集合。

Bootstrap Kit 主要用于初始化 PostgreSQL 数据库系统的基础结构，包括系统目录（system catalog）、系统表（system table）以及其他必要的系统对象。这些系统目录和表存储了 PostgreSQL 数据库系统的元数据信息，如数据库对象、表结构、索引信息等。

genbki.pl
处理系统表文件如pg_class.h,pg_namespace.h,pg_proc.h等，并生成postgres.bki供initdb加载，主要是加载初始化数据，如创建系统表，系统函数和预定义类型等。

在initdb前的编译阶段，会通过genbki.pl生成postgres.bki 和 postgres_ora.bki，postgres.bki保持PG原样，postgres_ora.bki在pg的基础上增加oracle 兼容的系统对象。



兼容特性

...1. like

Oracle和IvorySQL中的 LIKE
语法是相同的，他们的区别在于表达式的类型，Oracle支持对数字、日期、字符串类型的列用 LIKE
关键字配合通配符来实现模糊查询。原生PostgreSQL只支持字符串类型，不支持日期、数字类型，IvorySQL
通过扩展数据类型支持和操作符重载，实现了与 Oracle 兼容的 LIKE 操作符功能。

实现原理

PostgreSQL的字符串基本类型是text，所以 LIKE
是以text为基础，其他PostgreSQL类型隐式转换成text，不用创建operator就能自动转换；IvorySQL中oracle
兼容的字符串类型是varchar2，因此创建一个varchar2的 LIKE
操作符，其他oracle的类型也通过隐式转换成varchar2实现不用创建操作符，也能使用 LIKE 操作符。

在之前实现oracle兼容数据类型时，IvorySQL做了integer，float8，float4
等一些数据类型到varchar2的隐式转换，没有直接到text的。因此实现这些兼容类型的 LIKE
操作符兼容，有两种方式。一种需要针对每个类型添加一个 LIKE 操作符，另一种是做一个基本的varchar2的
LIKE
操作符。在第二种实现方式中，IvorySQL针对float8，integer，number等已经做了向varchar2类型的隐式转
换，这些数据类型可以和varchar2用同一个操作符，这样在创建操作符的时候只需要创建varchar2类型的
LIKE 操作符就可以。

类型转换

可以通过以下SQL语句查看已经存在的类型隐式转换

```
-- 查询已存在的隐式转换路径(9503为varchar2的OID)
SELECT t1.typname AS source_type, t2.typname AS target_type
```

```
FROM pg_cast C
JOIN pg_type t1 ON C.castsource = t1.OID
JOIN pg_type t2 ON C.casttarget = t2.OID
WHERE C.casttarget = 9503;
```

经过查看可得，需要进行兼容的类型全部都在该表中做了隐式转换，所以可以直接将这些类型转换为text类型然后进行模糊查询。

核心函数实现

```
CREATE OR REPLACE FUNCTION sys.varchar2like(varchar2, varchar2)
RETURNS bool AS $$
SELECT $1::text LIKE $2::text;
$$ LANGUAGE SQL IMMUTABLE STRICT;

CREATE OPERATOR ~~ (
  PROCEDURE = sys.varchar2like,
  LEFTARG   = varchar2,
  RIGHTARG  = varchar2
);
```

第一段代码定义了一个名为sys.varchar2like的函数，它接收两个varchar2类型的参数，通过将它们转换为PostgreSQL原生的text类型后执行标准的LIKE模式匹配，最终返回布尔值表示是否匹配。
第二段代码创建了一个名为~~的操作符，它将使用前面定义的varchar2like函数作为实现，并指定该操作符左右两边的参数都必须是varchar2类型。
这样就在IvorySQL中建立了一个与Oracle兼容的LIKE操作符，当用户使用~~操作符时，实际上是通过类型转换后调用PostgreSQL原生的LIKE功能来完成模式匹配，从而实现了Oracle的LIKE语义兼容。

...2. RowID

目的

IvorySQL提供了兼容Oracle
RowID的功能。RowID是一种伪列，在创建表时由数据库自动生成，对于数据库中的每一行，RowID
伪列返回该行的地址。

RowID 应当具备以下特性：

1. 逻辑地标识每一行，且值唯一
2. 可以通过ROWID快速查询和修改表的其他列，自身不能被插入和修改
3. 用户可以控制是否开启此功能

实现说明

在IvorySQL中系统列 ctid 字段代表了数据行在表中的物理位置，也就是行标识（tuple
identifier），由一对数值组成（块编号和行索引），可以通过ctid快速的查找表中的数据行，这样和Oracle的RowID行为很相似，但是ctid值有可能会改变（例如当update/ vacuum
full时），因此ctid不适合作为一个长期的行标识。

我们选择了表的oid加一个序列值组成的复合类型来作为RowID值，其中的序列是系统列。如果RowID功能被开启，则在建表的同时创建一个名为table-id_rowid_seq 的序列。同时在heap_form_tuple构造函数中，为HeapTupleHeaderData 的长度增加8个字节，并标识td→t_infomask = HEAP_HASROWID

位来表示rowid的存在。

在开启了ROWID的GUC参数或建表时带上 WITH ROWID 选项，或对普通表执行 ALTER TABLE … SET WITH ROWID 时会通过增加序列创建命令来创建一个序列。

```
/*
         * Build a CREATE SEQUENCE command to create the sequence object,
         * and add it to the list of things to be done before this CREATE/ALTER TABLE
         */
        seqstmt = makeNode(CreateSeqStmt);
        seqstmt->with_rowid = true;
        seqstmt->sequence = makeRangeVar(snamespace, sname, -1);
        seqstmt->options = lcons(makeDefElem("as",
                                 (Node *) makeTypeNameFromOid(INT8OID, -1),
                                 -1),
                                 seqstmt->options);
        seqstmt->options = lcons(makeDefElem("nocache",
                                 NULL,
                                 -1),
                                 seqstmt->options);
```

同时为了快速通过RowID伪列查询到一行数据，默认会在表的RowID列上创建一个UNIQUE索引，以提供快速查询功能。

RowID列作为系统属性列其实现是通过在 heap.c 中新增一个系统列来实现的。

```
/*
 * Compatible Oracle ROWID pseudo column.
 */
static const FormData_pg_attribute a7 = {
    .attname = {"rowid"},
    .atttypid = ROWIDOID,
    .attlen = -1,
    .attnum = RowIdAttributeNumber,
    .attcacheoff = -1,
    .atttypmod = -1,
    .attbyval = false,
    .attalign = TYPALIGN_SHORT,
    .attstorage = TYPSTORAGE_PLAIN,
    .attnotnull = true,
    .attislocal = true,
};
```

在pg_class系统表中增加一个 bool 类型的字段 relhasrowid，用于标识建表时的 WITH ROWID选项，如果建表时带了WITH ROWID选项，则 relhasrowid为 t，否则为f。 用户在执行 ALTER table …

SET WITH ROWID/ WITHOUT ROWID 命令时，也会修改这个值。

```
    /* T if we generate ROWIDs for rows of rel */
bool        relhasrowid BKI_DEFAULT(f);
```

在RowID的存储方面，如果启用了RowID 伪列功能，则在插入表之前heap_form_tuple函数会根据参数TupleDesc 中tdhasrowid 是否为true 在 HeapTupleHeaderData 中增加8个字节来存储序列值。 在heap_prepare_insert 函数中获取序列的nextval值，存在HeapTupleHeader相应的位置。

```
    if (relation->rd_rel->relhasrowid)
{
    //  Get the sequence next value
    seqnum = nextval_internal(relation->rd_rowdSeqid, true);
    //  Set the HeapTupleHeader
    HeapTupleSetRowId(tup, seqnum);
}
```

...3. OUT 参数

IvorySQL提供了兼容Oracle的out参数功能，包括含有out参数的函数与过程、匿名块支持out参数、libpq支持out参数。

实现原理

含有out参数的函数

对于pliSQL函数，在创建函数时，系统表pg_proc中存储函数参数为全部参数个数（包括OUT参数个数）及对应的参数数据类型。

在处理函数参数的interpret_function_parameter_list()函数中，根据参数模式，判断如果是IN OUT，则参数不能有默认值。

在make_return_stmt函数中，去掉如果发现有out参数报错的处理。

通过修改FuncnameGetCandidates函数，在函数查找时匹配包括out参数在内的所有参数。

函数编译时，构造一个row变量来容纳OUT参数变量和返回值变量。修改编译好的函数返回类型（function→fn_rettype）为RECORDOID。

函数执行时，在ExecInitFunc函数中调用新函数ExecInitFuncOutParams，构造OUT参数计算节点，通过计算函数plisql_out_param将函数返回值和OUT参数值从tuple中分离，并且为OUT参数向外赋值。

匿名块支持out参数

为了支持冒号占位符形式的绑定变量，修改ora_scan.l文件，添加语法，遇到冒号占位符返回ORAPARAM。在ora_gram.y文件中，在c_expr和赋值plassign_target的语法中，添加对ORAPARAM添加处理，构造一个OraParamRef节点。

新增 DO + USING 语法：

```
DO [ LANGUAGE lang_name ] code [USING IN | OUT | IN OUT, ...]
```

修改ora_gram.y文件，增加DO+USING语法支持。DoStmt结构体中增加 paramsmode
用于存储匿名块中的绑定变量模式等信息的列表。

对PBE过程的修改包括：exec_parse_message函数中，根据应用接口传过来的参数类型oid，识别出实参的模式；在exec_bind_message函数中，对于匿名块DO+USING，识别出USING后面参数的模式，向执行器传递参数信息。

含OUT参数匿名块的执行

1. 在PortalStart函数中对匿名块语句，调用CreateTupleDescFromParams函数，增加参数构造描述信息。
2. PLiSQL_function成员fn_prokind增加新值PROKIND_ANONYMOUS_BLOCK用来表示匿名块。
3. 在plisql_exec_function函数中，对有out参数的匿名块做判断，调用plisql_anonymous_return_out_parameter函数完成对OUT参数构造PLiSQL_row类型的变量，最后将row类型的变量求值当作匿名块函数的返回值。

libpq中调用含out参数的函数

修改Libpq以支持按位置和按参数名字绑定，涉及SQL端，PLiSQL端，libpq接口端。

1. SQL端：在服务器端实现系统函数
   get_parameter_description，该函数根据SQL语句，返回变量名字与位置的关系。这个函数被用在libpq接口函数中。 返回的第一行name显示SQL类型，后面行依次显示占位符名字、位置信息。

   ```
   ```
   ivorysql=# select * from get_parameter_description('insert into t values(:x,
   :y);');
    name | position
   -------+----------
    false | 0
    :x | 1
    :y | 2
   (3 rows)
   ```
   对于匿名块语句，尚不支持。
   ```

2. PLiSQL端：主要是PL/iSQL块根据参数位置或参数名称调整参数内部标识。

   ```
   执行函数需要从绑定句柄中获取参数的值与类型的信息；
   ```

   ```
   对out参数的返回列名称做一个特殊处理。如果是out参数，那么其列名称为_column_xxx，其中xxx是out参数的位置，从而根据绑定位置与返回的位置从结果集中给out参数赋值；
   ```

   ```
   在PLiSQL执行端，根据参数名字出现的位置，调整名字转换成内部标识的变量，如$number。在返回到客户端的时候，发送描述信息给libpq，
   从而达到返回的列名是由参数名字构造，LIBPQ端根据列名来给out参数赋值。
   ```

3. libpq接口端：提供准备、绑定、执行函数，这些函数与OCI接口相应函数类似。

```
一般调用流程如下:
使用IvyHandleAlloc分配语句句柄和错误句柄。
调用IvyStmtPrepare准备语句。
调用IvyBindByPos或IvyBindByName 绑定参数。
调用IvyStmtExecute 执行，可重复执行。
调用IvyFreeHandle 释放语句句柄和错误句柄。
```

```
另外还实现了Ivyconnectdb，Ivystatus，Ivyexec，IvyresultStatus，IvyCreatePreparedStatement，IvybindOutParameterByPos，IvyexecPreparedStatement，IvyexecPreparedStatement2，Ivynfields，Ivyntuples，Ivyclear等一系列接口函数。
```

...4. %TYPE、%ROWTYPE

目的

IvorySQL提供了兼容Oracle的plsql数据类型功能，包括%TYPE、%ROWTYPE。

实现说明

引用类型发生变化时，如何确保%TYPE声明的变量也随之相应改变

这是一个被动的过程。 当前的实现方案是记录函数（存储过程）与 tablename.columname 的依赖关系。当引用类型发生变化时，根据依赖关系，让函数（存储过程）缓存失效。这样一来，函数被调用时就会执行强制编译。从而确保函数获取最新的变量类型。

在系统表 pg_proc 中添加一个字段 prostatus 用来表示函数（存储过程）状态，有3种状态: validate(v)，invalidate(i)，N/A(n)。函数编译成功后，这个状态会被设置为valid。

在解析函数内容时，plisql_parse_cwordtype，plisql_parse_wordrowtype，plisql_parse_cwordrowtype 等函数识别出%TYPE， %ROWTYPE引用的对象，并记录在 plisql_referenced_object 链表中，最后添加到 pg_depend 系统表中。

增加一种新的依赖类型 DEPENDENCY_TYPE = 't'，表示%TYPE或%ROWTYPE依赖。在添加对象引用关系到pg_depend系统表时，设置依赖类型为 't'。

对表进行操作时（修改表类型、删除表等），查看系统表pg_depend，如果存在依赖类型deptype=’t’，并且依赖对象是函数，则调用函数 plisql_free_function 删除函数缓存，并修改 pg_proc 系统表中的函数状态prostatus 为 N/A(n)。

%TYPE声明的变量继承引用变量的约束

在 PLiSQL_type 结构体中添加成员 bool notnull;

```
/*
 * Postgres data type
 */
typedef struct PLiSQL_type
{
    bool        notnull;        /* the type is built by variable%type,
                                 * isnull or notnull of the variable */
```

在负责解析%TYPE类型函数的plisql_parse_wordtype或plisql_parse_cwordtype函数中，判断如果引用变量类型指定了NOT NULL约束，为返回的datatype的成员 bool notnull 属性赋值为true。 在pl_gram.y 文件的decl_statement 语法中，根据PLiSQL_type 的成员 bool notnull 为变量PLiSQL_variable *var的notnull属性赋值。这样就继承了引用变量的约束。

表名%ROWTYPE或视图名%ROWTYPE作为函数或存储过程的参数类型和函数返回值类型

在 ora_gram.y 中为 func_type 添加%ROWTYPE支持。

```
| type_function_name attrs '%' ROWTYPE
                {
                    $$ = makeTypeNameFromNameList(lcons(makeString($1), $2));
                    $$->row_type = true;
                    $$->location = @1;
                }
```

在 TypeName 结构体中添加成员 bool row_type 用来标记是否指定了%ROWTYPE。

```
typedef struct TypeName
{
    bool        pct_type;       /* %TYPE specified? */
    bool        row_type;   /* %ROWTYPE specified? */
```

在 LookupTypeName 函数中，如果 TypeName 的成员 row_type 为TRUE，则根据TypeName的成员names中获得schema名与表名，然后获取表的typeoid。

INSERT语句增强

在 ora_gram.y 中添加新的语法，支持 VALUES 后面不带括号’（‘。

```
values_clause_no_parens:
            VALUES columnref
                {
                    SelectStmt *n = makeNode(SelectStmt);
                    n->valuesLists = list_make1(list_make1($2));
                    n->valuesIsrow = true;
                    $$ = (Node *) n;
                }
```

为结构体 SelectStmt 添加一个字段 bool valuesIsrow 表示 values 是一个 row。

在为insert语句做 transform 时，也就是函数 transformInsertStmt 中，在处理 INSERT … VALUES 语句时，如果 valuesIsrow 为true，调用新函数 transformRowExpression，将 row_variable 转换成等价的 row_variable.field1,…,row_variable.fieldN。

UPDATE语句增强

在UPDATE语句做transform时候，也就是transformUpdateStmt的时候，如果是Oracle兼容模式，调用新添加的 transformIvyUpdateTargetList 函数。

在这个新函数中，对于参数origTlist(即targetList)中没有名字为row的情况，按原来UPDATE的transform流程执行 transformUpdateTargetList 函数。

对参数origTlist中有名字为row的情况：因为PostgreSQL中row可以作为列名，而Oracle中row是保留关键字，不可以作为列名，所以需要区分row是否是表中的列，如果row不是要更新的表中的列，则调用新函数 transformUpdateRowTargetList 把语句

```
UPDATE table_name SET ROW = row_variable [WHERE …];
```

转换成等价的

```
UPDATE table_name SET table_name.column1 = row_variable.column1, table_name.column2 =
row_variable.filed2,… table_name.columnN = row_variable.columnN [WHERE …];
```

如果语句“UPDATE table_name SET ROW = row_variable”中的变量row_variable不是组合类型，就按原来UPDATE的transform流程执行 transformUpdateTargetList 函数；
如果语句中的变量 row_variable
是组合类型，表中的名为row的那一列也是组合类型，并且两者的类型Oid也一样，则按原来UPDATE的transform流程执行 transformUpdateTargetList 函数； 其他所有情况，调用新函数 transformUpdateRowTargetList 处理；

...5. NLS 参数

IvorySQL提供了兼容Oracle的NLS参数功能，包含如下参数。

参数名称	功能描述
ivorysql.datetime_ignore_nls_mask	表示日期格式是否忽略NLS参数影响，默认为0。
nls_length_semantics	兼容Oracle的同名参数，表示char/varchar/varchar2的类型修饰符的大小单位是字节还是字符。
nls_date_format	表示默认的日期格式，可以通过show命令查看，默认为‘YYYY-MM-DD’。
nls_timestamp_format	兼容Oracle的同名参数，控制带时间的日期格式。
nls_timestamp_tz_format	兼容Oracle的同名参数，控制带时区的日期时间格式。
nls_territory	兼容Oracle的同名参数，指定数据库的默认区域。
nls_iso_currency	兼容Oracle的同名参数，指定国家和区域对应的唯一货币符。
nls_currency	兼容Oracle的同名参数，指定显示本地货币的符号，对应数字字符串格式中占位符L。

实现原理

`nls_length_semantics` 参数

IvorySQL中的数据类型存在一个属性修饰符 `typmod` ，是对类型的补充说明，比如在 `VARCHAR(n)` 类型中，`n` 就是类型修饰符。

在创建或修改表的列时可以指定长度类型，例如:

```
ivorysql=# create table t1(name varchar2(2 byte));
```

对于列类型为 `CHAR` 、 `VARCHAR` 和 `VARCHAR2` 字符型的列，当没有显式指定列的长度类型时，IvorySQL使用 `nls_length_semantics` 参数的值来决定长度类型，有 `byte` 和 `char` 两种值，默认为 `byte` 。

需要特别注意的是， nls_length_semantics
参数的值仅影响新创建的列，对已经存在的列不会产生任何影响。

在语法解析文件 ora_gram.y中，存在如下代码来根据 nls_length_semantics 把原本的 char/varchar/varchar2 类型改成 oracharchar 或者 oracharbyte：

```
CharacterWithLength:  character '(' Iconst ')'
                {
                    if (ORA_PARSER == compatible_db)
                    {
                        if (strcmp($1, "bpchar"))
                        {
                            if (nls_length_semantics == NLS_LENGTH_CHAR)
                                $1 = "oravarcharchar";
                            else
                                $1 = "oravarcharbyte";
                        }
                        else
                        {
                            if (nls_length_semantics == NLS_LENGTH_CHAR)
                                $1 = "oracharchar";
                            else
                                $1 = "oracharbyte";
                        }

                        $$ = OracleSystemTypeName($1);
                        $$->typmods = list_make1(makeIntConst($3, @3));
                        $$->location = @1;
                    }
                    else
                    {
                        ...
                    }
                }
        ;
```

IvorySQL 中数据类型 oracharchar 和 oracharbyte 的修饰符输入输出函数包括：

```
oravarcharchartypmodout()
oravarcharbytetypmodout()
oracharbytetypmodout()
oracharchartypmodout()
```

上面这些函数调用C语言实现的函数 anychar_typmodout()，后者根据 nls_length_semantics

的值来调整输出的内容是否包含 byte/char 的说明。

nls_length_semantics 另一个作用是限制表中的列长度：根据上述代码在语法解析文件ora_gram.y中，如果原本的 varchar 类型被转换成了 oracharchar 类型，则函数 oravarcharchar() 会被调用，而 pg_mbcharcliplen() 函数计算字符长度，而不是字节长度。

```
Datum
oravarcharchar(PG_FUNCTION_ARGS)
{
    VarChar    *source = PG_GETARG_VARCHAR_PP(0);
    int32       typmod = PG_GETARG_INT32(1);
    bool        isExplicit = PG_GETARG_BOOL(2);
    int32       len,
                maxlen;
    size_t      maxmblen;
    char       *s_data;

    len = VARSIZE_ANY_EXHDR(source);
    s_data = VARDATA_ANY(source);
    maxlen = typmod - VARHDRSZ;

    /* No work if typmod is invalid or supplied data fits it already */
    if (maxlen < 0 || len <= maxlen)
        PG_RETURN_VARCHAR_P(source);

    maxmblen = pg_mbcharcliplen(s_data, len, maxlen);

    ...
}
```

GUC参数 datetime_ignore_nls_mask

这个参数被定义为一个int值，低四位分别表示是否在相应的日期时间格式上忽略NLS参数的影响，掩码定义如下：

```
#define ORADATE_MASK            0x01
#define ORATIMESTAMP_MASK       0x02
#define ORATIMESTAMPTZ_MASK     0x04
#define ORATIMESTAMPLTZ_MASK    0x08
```

在源代码中，这个GUC参数被用于下面这些函数：

```
oradate_in()
oratimestamp_in()
oratimestampltz_in()
```

```
oratimestamptz_in()
```

如果相应的掩码被设置，则调用原生PG的处理函数，否则调用兼容代码并忽略NLS格式。

GUC参数 `nls_date_format`/`nls_timestamp_format`/`nls_timestamp_tz_format`

这三个GUC参数，在函数 `ora_do_to_timestamp()`
中作为格式字符串，对输入的字符串进行格式检查与模式识别。

下面是其默认值，可以通过设置其值为"pg"使其失效。"pg"表示禁用NLS特定行为，恢复为PostgreSQL的默认行为。

```
char        *nls_date_format = "YYYY-MM-DD";
char        *nls_timestamp_format = "YYYY-MM-DD HH24:MI:SS.FF6";
char        *nls_timestamp_tz_format = "YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM";
```

GUC参数 `nls_currency`/`nls_iso_currency`/`nls_territory`

目前，`nls_territory` 和 `nls_iso_currency` 支持CHINA与AMERICA两个值。

默认值如下:

```
char        *nls_territory = "AMERICA";
char        *nls_currency = "$";
char        *nls_iso_currency = "AMERICA";
```

这三个参数将在oracle兼容函数 `to_number()` 中被使用。

> `to_number()` 函数尚未实现。

...6. 函数与存储过程

目的

PostgreSQL支持函数(FUNCTION)和存储过程(PROCEDURE)，但是在语法上和Oracle有差异，为了让Oracle的PLSQL语句可以在IvorySQL上执行，也就是“语法兼容”，IvorySQL采取这样的解决方案：如果Oracle的子句在IvorySQL中存在相同功能的子句，则直接映射为对应的IvorySQL子句，否则只实现其语法，不实现其功能。

实现说明

PL/iSQL是IvorySQL中的过程语言名称，专门用来兼容Oracle的PLSQL语句。为了兼容Oracle风格的函数与存储过程语法，需要对psql客户端工具、SQL端以及PL/iSQL端做相应处理。

客户端工具psql

Oracle的sqlplus工具使用斜线(/)来结束函数和存储过程，IvorySQL的客户端工具psql需要兼容同样的语法，也就是说我们常规的遇到分号发送语句给服务端的机制，当遇到Oracle风格的函数和存储过程命令时失效，改为使用斜线(/)发送命令。

为此在 PsqlScanStateData 结构体中增加如下字段：

```
    bool        cancel_semicolon_terminator; /* not send command when semicolon found
```

```
*/

    /*
     * State to track boundaries of Oracle ANONYMOUS BLOCK.
     * Case 1: Statements starting with << ident >> is Oracle anonymous block.
     */
    int     token_count;            /* # of tokens, not blank or newline since start
of statement */
    bool        anonymous_label_start;      /* T if the first token is "<<" */
    bool        anonymous_label_ident;  /* T if the second token is an identifier */
    bool        anonymous_label_end;        /* T if the third token is ">>" */

    /*
     * Case 2: DECLARE BEGIN ... END is Oracle anonymous block syntax.
     * DECLARE can also be a PostgreSQL cursor declaration statement, we need to tell
this.
     */
    bool        maybe_anonymous_declare_start;  /* T if the first token is DECLARE */
    int     token_cursor_idx;       /* the position of keyword CURSOR in SQL statement
*/

    /*
     * Case 3: DECLARE BEGIN ... END is Oracle anonymous block syntax.
     * BEGIN can also be a PostgreSQL transaction statement.
     */
    bool        maybe_anonymous_begin_start;    /* T if the first token is BEGIN */
```

同时修改 ora_psqlscan.l，添加和修改相应的词法规则, 以下是代码片段示例：

```
{
    if (is_oracle_slash(cur_state, cur_state->scanline))
    {
        /* Terminate lexing temporarily */
        cur_state->cancel_semicolon_terminator = false;
        cur_state->maybe_anonymous_declare_start = false;
        cur_state->maybe_anonymous_begin_start = false;
        cur_state->anonymous_label_start = false;
        cur_state->anonymous_label_ident = false;
        cur_state->anonymous_label_end = false;
        cur_state->start_state = YY_START;
        cur_state->token_count = 0;
        cur_state->token_cursor_idx = 0;
        cur_state->identifier_count = 0;
```

```
        cur_state->begin_depth = 0;
        cur_state->ora_plsql_expect_end_symbol = END_SYMBOL_INVALID;
        return LEXRES_SEMI;
    }
    ECHO;
```

避免将注释等场景中的斜线误判为结束符，为此在 oracle_fe_utils/ora_psqlscan.l 文件中新增 is_oracle_slash 函数接口用于检测：

```
bool
is_oracle_slash(PsqlScanState state, const char *line)
{
    bool result = false;

    switch (state->start_state)
    {
        case INITIAL:
        case xqs:   /* treat these like INITIAL */
            {
                int len, i;
                bool has_slash = false;

                len = strlen(line);
                for (i = 0; i < len; i++)
                {
                    /* allow special char */
                    if (line[i] == '\t' ||
                        line[i] == '\n' ||
                        line[i] == '\r' ||
                        line[i] == ' ')
                        continue;

                    if (line[i] == '/')
                    {
                        if (has_slash)
                            break;
                        has_slash = true;
                        continue;
                    }
                    /* others */
                    break;
                }
```

```
                if (i == len && has_slash)
                    result = true;
            }
            break;
        default:
            break;
    }

    return result;
}
```

SQL端

SQL端要能够识别函数和存储过程的创建语法，这是通过修改ora_base_yylex来实现的。这个函数预取并缓存token，如果是Oracle语法格式则组织一个SCONST发送给PLSQL端，否则从堆栈中获取之前预读的token，按照原生PG的逻辑进行处理。

在ora_base_yy_extra_type数据结构中增加如下字段：

```
    /*
     * The native PG only cache one-token info include yylloc, yylval and token
     * number in yyextra, IvorySQL cache multiple tokens info using two arrays.
     */
    int max_pushbacks;      /* the max size of cache array */
    int loc_pushback;       /* # of used tokens */
    int num_pushbacks;      /* # of cached tokens */
    int *pushback_token;            /* token number array */
    TokenAuxData *pushback_auxdata; /* auxdata array */

    OraBodyStyle body_style;
    int          body_start;
    int          body_level;
```

增加token堆栈的操作接口：

push_back_token
forward_token
ora_internal_yylex
internal_yylex

⊠⊠
ora_base_yylex函数中在创建函数、过程、匿名块时会预读部分token，使用上述结构缓存到堆栈中，是为了构造一个符合Oracle PL/SQL语法的SCONST发送给PL/iSQL端去处理。具体请参考源代码。

PL/iSQL端

该部分主要修改了pl_gram.y文件，以兼容PLSQL的函数和存储过程语法，在不影响PG原生的PL/pgSQL的前提下去兼容Oracle PL/SQL语法形式，如下是DECLARE部分兼容的代码示例，更多请参考IvorySQL源代码。

```
/*
 * The declaration section of the outermost block in Oracle does not have the DECLARE
keyword.
 */
ora_outermost_pl_block: ora_decl_sect K_BEGIN proc_sect exception_sect K_END opt_label
                    {
                        PLiSQL_stmt_block *new;

                        new = palloc0(sizeof(PLiSQL_stmt_block));

                        new->cmd_type   = PLISQL_STMT_BLOCK;
                        new->lineno     = plisql_location_to_lineno(@2);
                        new->stmtid     = ++plisql_curr_compile->nstatements;
                        new->label      = $1.label;
                        new->n_initvars = $1.n_initvars;
                        new->initvarnos = $1.initvarnos;
                        new->body       = $3;
                        new->exceptions = $4;

                        check_labels($1.label, $6, @6);
                        plisql_ns_pop();

                        $$ = (PLiSQL_stmt *)new;
                    }
                ;

ora_decl_sect: opt_block_label opt_ora_decl_start opt_ora_decl_stmts
                {
                        if ($2)
                        {
                                if ($1 == NULL)
                                {
                                        plisql_ns_push(NULL, PLISQL_LABEL_BLOCK);
                                }
                        }
                }
                opt_ora_decl_stmts
                {
                        if ($4)
                        {
                                plisql_IdentifierLookup = IDENTIFIER_LOOKUP_NORMAL;
                                $$.label          = ($1 == NULL ?
```

```
plisql_curr_compile->namelabel : $1);
                                if ($2 && $1 == NULL)
                                        $$.popname = true;
                                else
                                        $$.popname = false;
                                /* Remember variables declared in decl_stmts */
                                $$.n_initvars =
plisql_add_initdatums(&($$.initvarnos));
                        }
                        else
                        {
                                plisql_IdentifierLookup = IDENTIFIER_LOOKUP_NORMAL;
                                $$.label          = ($1 == NULL ?
plisql_curr_compile->namelabel : $1);
                                $$.n_initvars = 0;
                                if ($2 && $1 == NULL)
                                        $$.popname = true;
                                else
                                        $$.popname = false;
                                $$.initvarnos = NULL;
                        }
                }
                ;

opt_ora_decl_start: K_DECLARE
                    {
                        /* Forget any variables created before block */
                        plisql_add_initdatums(NULL);
                        /*
                         * Disable scanner lookup of identifiers while
                         * we process the decl_stmts
                         */
                        plisql_IdentifierLookup = IDENTIFIER_LOOKUP_DECLARE;
                        $$ = true;
                    }
                | /*EMPTY*/
                    {
                        /* Forget any variables created before block */
                        plisql_add_initdatums(NULL);
                        /*
                         * Disable scanner lookup of identifiers while
                         * we process the decl_stmts
                         */
```

```
                        plisql_IdentifierLookup = IDENTIFIER_LOOKUP_DECLARE;
                        $$ = false;
                    }
                ;

opt_ora_decl_stmts:
                ora_decl_stmts
                    {
                        $$ = true;
                    }
                | /*EMPTY*/
                    {
                        $$ = false;
                    }

ora_decl_stmts: ora_decl_stmts ora_decl_stmt
                | ora_decl_stmt
                ;

ora_decl_stmt: decl_statement
                ;
```

...7. 嵌套子函数

目的

- 嵌套子函数：定义在函数、存储过程或匿名块内部的函数或存储过程，也称为 subproc 或 inner 函数。
- 父函数：承载嵌套函数的外层函数、存储过程或匿名块，执行过程中负责实际触发子函数调用。

实现说明

嵌套子函数语法识别

识别嵌套写法

当 `DECLARE` 块里出现 `function … is/as begin … end` 结构时，`pl_gram.y` 会调用 `plisql_build_subproc_function()`（对应创建普通函数，这一阶段相当于在 `pg_proc` 中更新 catalog 中的注册信息）：

1. 在 `PLiSQL_function` 的 `subprocfuncs[]` 数组中创建 `PLiSQL_subproc_function` 结构，记录名称、参数、返回类型、属性等，获得一个下标 `fno` 作为该子函数的标识。
2. 调用 `plisql_check_subprocfunc_properties()` 校验声明与定义的合法组合。

数据项保存

保存到父函数的 `datum` 表：编译期的 `PLiSQL_function→datums` 描述子函数里的变量、记录字段，`PLiSQL_execstate→datums` 保存着执行过程中的变量。

保存多态函数模板

如果子函数里使用了多态参数，在语法阶段保存到 subprocfunc→src，同时将 has_poly_argument 设成 true，执行时按不同实参类型重新编译。

父函数重新编译

1. 父函数的 PLiSQL_function 结构多了一个 subprocfuncs 数组，里面每个元素就是刚才创建的 PLiSQL_subproc_function。
2. 子函数结构体 PLiSQL_subproc_function 有一个哈希表指针 HTAB *poly_tab，默认为空。当子函数里使用了多态函数时，has_poly_argument 为 true，则会在初次编译时初始化 poly_tab。poly_tab 的 key 是 PLiSQL_func_hashkey，记录着子函数的 fno、输入参数类型等； value 是编译好的 PLiSQL_function *（plisql 函数的执行上下文）。

调用时解析

1. 编译过程中，pg 解析器会生成一个 ParseState 结构，plisql_subprocfunc_ref() 会通过 ParseState→p_subprocfunc_hook() 找到父函数的 PLiSQL_function，调用 plisql_ns_lookup() 找到所有同名子函数的 fno，根据参数个数、类型找到最合适的多态函数。
2. FuncExpr 结构构造时会对子函数进行标记，方便后期执行阶段识别：function_from = FUNC_FROM_SUBPROCFUNC，parent_func 指向父级 PLiSQL_function，funcid = fno。
3. plisql_call_handler() 当 function_from == FUNC_FROM_SUBPROCFUNC，会用 parent_func + fno 找到对应的 PLiSQL_subproc_function：
   a. 如果不是多态：直接复用 subprocfunc→function 里的动作树。
   b. 如果是多态：先在 poly_tab 查有没有编译结果；没有就调用 plisql_dynamic_compile_subproc() 编译，放进 poly_tab 缓存。
4. 子函数开始执行之前，plisql_init_subprocfunc_globalvar() 会把父函数的 datum 表中有关的变量 fork 一份，这样子函数可以获取到父函数的变量值，也不会污染父函数的变量空间；执行后由 plisql_assign_out_subprocfunc_globalvar() 把需要回写的变量更新到父函数的 datum 表。

模块设计

PL/iSQL 语法扩展

- pl_gram.y 新增子过程声明、嵌套定义的产生式，并在创建过程中记录 lastoutvardno、子过程信息等元数据。
- 支持在子函数内引用父过程变量、子过程以及自定义类型。

当 DECLARE 块内出现 function … is/as begin … end 结构时，pl_gram.y 会调用 plisql_build_subproc_function() 进行编译：

1. 在 PLiSQL_function 的 subprocfuncs[] 中创建 PLiSQL_subproc_function 结构，记录名称、参数、返回类型和属性，分配下标 fno 作为子函数标识。
2. 调用 plisql_check_subprocfunc_properties() 校验声明与定义属性组合是否合法，防止重复或缺失声明造成的语义错误。

数据项保存

父函数的 Datum 表在编译期和执行期分别缓存子函数能访问的变量：

1. PLiSQL_function→datums 保存子函数编译阶段可见的变量与记录字段信息。
2. PLiSQL_execstate→datums 在执行阶段持有实时的变量数值，实现运行期访问。

多态函数模板

若子函数包含多态参数，语法阶段会：

1. 将子函数源文本拷贝到 subprocfunc→src。

2. 设置 `has_poly_argument = true`，为后续按实参类型动态编译做好准备。

父函数重新编译

- 父函数的 `PLiSQL_function` 结构新增 `subprocfuncs` 数组，每个元素对应一个 `PLiSQL_subproc_function`。
- `PLiSQL_subproc_function` 持有 `HTAB *poly_tab` 指针；当 `has_poly_argument` 为 `true` 时，在首次编译时初始化该缓存，键为 `PLiSQL_func_hashkey`（子函数 `fno` + 实参类型），值为编译后的 `PLiSQL_function`。

解析器钩子

编译期间 PostgreSQL 解析器会构造 `ParseState`，`plisql_subprocfunc_ref()` 通过 `ParseState→p_subprocfunc_hook()` 连接父函数，调用 `plisql_ns_lookup()` 找到同名子函数的全部 `fno`，并依据参数个数与类型挑选最佳候选，实现重载分发。

FuncExpr 标记

构造 `FuncExpr` 时会标记嵌套调用信息，便于执行阶段识别：

- `function_from = FUNC_FROM_SUBPROCFUNC`。
- `parent_func` 指向父级 `PLiSQL_function`。
- `funcid = fno`，用于快速定位子函数定义。

嵌套函数查找机制

- `plisql_subprocfunc_ref()` 作为 `ParseState→p_subprocfunc_hook` 实现入口，复用名称空间查询逻辑。
- `plisql_get_subprocfunc_detail()` 依据参数数量、类型与命名匹配规则挑选最优候选，是嵌套函数重载的关键。

执行路径

1. `plisql_call_handler()` 判断 `function_from` 后，通过 `parent_func + fno` 找到目标 `PLiSQL_subproc_function`。
2. 对普通子函数，直接复用 `subprocfunc→function` 缓存；
3. 对多态子函数，先查询 `poly_tab`，未命中时调用 `plisql_dynamic_compile_subproc()` 动态编译并写入缓存。

变量同步

- `plisql_init_subprocfunc_globalvar()` 在子函数执行前拷贝父函数 Datum 表中的相关变量，保证子函数读取到外层最新状态。
- `plisql_assign_out_subprocfunc_globalvar()` 在返回前回写 OUT/INOUT 变量，确保父子函数数据一致性且互不污染。

PSQL 端语句发送

- `psqlscan.l` 调整 `proc_func_define_level` 和 `begin_depth` 的入栈/出栈逻辑，确保嵌套函数体整体发送至 SQL 端。
- 只有当嵌套层级回到 0 且遇到分号时，才触发发送，避免子函数块被拆分。

SQL 层返回值获取

- 普通函数通过 `funcid` 访问 `pg_proc`；嵌套函数依赖 `FuncExpr.parent_func` 承载的 `PLiSQL_function`。
- 为此实现一组函数指针（`plisql_register_internal_func()` 注册）供 SQL 层回调，按需获取嵌套函数名称、返回类型与 OUT 参数信息。

...8. Force View

目的

- Force View 提供 Oracle 兼容的 `CREATE [OR REPLACE] FORCE VIEW` 与 `ALTER VIEW … COMPILE` 行为，允许开发者先落地视图占位，再在依赖补齐后恢复正常执行。
- 系统将原始 SQL 及标识符大小写模式写入专属目录项，依赖恢复时可自动或手动编译为普通视图，且保持 Oracle 风格的告警与错误提示。

实现说明

PG在删除基表时，如果有视图依赖于此基表，则不允许删除，需要使用cascade关键字全部删除；Oracle删除基表时，允许直接删除基表，而保留force view并将force view转为无效状态，且创建普通视图后，也可以删除此视图的基表，但把视图转为无效状态。

为了支持新的语法，需要增加对应的语法规则。且FORCE VIEW的状态在多个session之间是共享的，新增一张系统表用来存储FORCE VIEW的相关信息。

语法与解析入口

Force View 语法支持

- `src/backend/oracle_parser/ora_gram.y` 注册 Force View 关键字。
- 语法阶段为 `ViewStmt→force` 赋值，并在 `AlterTableStmt` 阶段生成 `AT_ForceViewCompile` 选项。
- 解析阶段保留 `ViewStmt→stmt_literal`，后续占位时需要复用原始文本。

```
/* Insert or update the pg_force_view catalog as needed */
if (need_store)
{
    ......

    StoreForceViewQuery(address.objectId, stmt->replace, ident_case, stmt-
>stmt_literal ? stmt->stmt_literal : queryString);
}
```

AST 字段扩展

- `src/include/nodes/parsenodes.h` 为 `ViewStmt` 增加 `bool force`、`char *stmt_literal` 字段。
- `AT_ForceViewCompile` 在 `parsenodes.h` 与 `tablecmds.c` 成对定义，确保 `ALTER VIEW … COMPILE` 进入 AlterTable 流程。
- `parse_analyze` 仍按普通路径执行，Force 模式只在解析报错后介入。

解析入口

- `DefineView()`（`src/backend/commands/view.c`）统一处理普通与 Force 视图。
- 代码在 `PG_TRY/PG_CATCH` 包裹下先尝试普通解析，捕获语义错误后根据 `stmt→force` 决定是否进入Force View 流程。
- 成功解析继续调用 `StoreViewQuery()` 写入 `_RETURN` 规则，失败则进入 Force View 分支，保证视图对象仍然存在。

pg_force_view 目录

- src/include/catalog/pg_force_view.h 定义目录表，字段包含视图 OID (fvoid)、标识符大小写模式 (ident_case)、原始 SQL 文本 (source)。
- 唯一索引 pg_force_view_fvoid_index 搭配 syscache FORCEVIEWOID（src/include/catalog/syscache_info.h）实现按视图 OID 查找。
- 目录表开启 TOAST，长 SQL 文本不会被截断，可覆盖复杂迁移脚本。

```
CATALOG(pg_force_view,9120,ForceViewRelationId)
{
    /* oid of force view */
    Oid         fvoid;

    /* see IDENT_CASE__xxx constants below */
    char        ident_case;

#ifdef CATALOG_VARLEN           /* variable-length fields start here */

    /* sql definition */
    text        source;
#endif
} FormData_pg_force_view;
```

视图状态

- bool rel_is_force_view(Oid relid) 位于 src/backend/commands/view.c，通过判断 _RETURN 规则是否存在确认视图是否处于 Force 状态。
- Force View 仍登记为 relkind = RELKIND_VIEW，避免额外兼容分支。
- pg_class.relhasrules 在占位状态设置为 false，配合作为判断依据。

创建与替换流程

普通 view

- DefineView() 在解析成功后调用 DefineVirtualRelation() 写入 pg_class、pg_attribute 等结构。
- StoreViewQuery() 生成 _RETURN 规则并维护依赖关系。
- 该路径不会触及 pg_force_view，视图完成后立即可用。

Force view

- CreateForceVirtualPlaceholder()（src/backend/commands/view.c）负责生成或复用占位视图：
- 若视图不存在，调用 DefineVirtualRelation() 建立基础对象，但不写入 _RETURN 规则。
- 若已有 Force View，复用当前记录，更新列定义或清理旧元数据。
- 若已有普通视图且声明了 OR REPLACE，调用 make_view_invalid() 失效原视图，再写入新的占位定义。
- StoreForceViewQuery() 将 stmt_literal 与当前 ivorysql.identifier_case_switch 持久化到 pg_force_view，以便后续恢复大小写语义。
- 视图占位完成后向客户端返回 WARNING: View created with compilation

errors，提示仍处于不可用状态。

依赖失效与回退

主动失效逻辑

- make_view_invalid() 在依赖对象被 DROP/ALTER 等操作影响时调用。
- 函数删除 _RETURN 规则、清理 pg_depend 依赖、重置 pg_class.relhasrules 并清空 pg_attribute 列信息。
- 同步生成 CREATE FORCE VIEW … AS <pg_get_viewdef> 文本写入 pg_force_view，ident_case 设置为 IDENT_CASE_UNDEFINE 表示该文本由系统构造。

失效后的可见行为

- 视图仍然存在并可在元数据中枚举，但实际访问会触发 Force 判定。
- _RETURN 缺失导致执行器在打开视图时调用 compile_force_view()，若编译仍失败则抛出 view "<schema>.<name>" has errors。
- 用户可通过 ALTER VIEW … COMPILE 或再次 CREATE OR REPLACE FORCE VIEW 来恢复。

自动与主动编译

自动编译触发点

- parse_relation.c 的 addRangeTableEntry() 和 parse_clause.c 的目标关系打开逻辑在判定 Force View 后调用 compile_force_view()。
- 该函数重新执行 raw_parser 与 parse_analyze，成功则再安装 _RETURN 规则，失败直接终止当前语句。

主动编译

- AT_ForceViewCompile 在 tablecmds.c Phase 2 阶段执行，先申请 AccessExclusiveLock 再调用 compile_force_view()。
- 成功编译返回常规 ALTER VIEW 完成信息；失败打印 WARNING: View altered with compilation errors，视图仍保持占位。

列校验与元数据更新

- compile_force_view() 读取 pg_force_view.source 并构造 ViewStmt，随后调用 compile_force_view_internal()。
- 函数通过 checkViewColumns() 比对旧列，允许新增列但禁止类型不兼容的替换；新增列由 AT_AddColumnToView() 完成。
- _RETURN 规则由 StoreViewQuery() 重新生成，最后 DeleteForceView() 删除目录项，视图恢复为普通状态。

...9. 大小写转换

目的

为了满足Oracle的引用标识符大小写兼容，ivorysql设计了三种引用标识符的大小写转换模式。

实现说明

如果在数据库初始化时附加了参数 -C，值可以为 normal/interchange/lowercase，则代码中 Initdb.c→main() 函数会处理该参数，根据参数值设置全局变量 caseswitchmode。然后 initdb 命令会以 -boot 模式启动一个 postgres 进程用于设置 template1 模板数据库，同时赋予参数 -C ivorysql.identifier_case_switch=caseswitchmode 给新进程。

这个新启动的后端进程会通过下面的代码将 identifier_case_switch 信息写入 pg_control 文件：

```
BootstrapModeMain() -> BootStrapXLOG();
    /* save database compatible level value */
    ControlFile->dbmode = bootstrap_database_mode;
    ControlFile->casemode = identifier_case_switch;

    /* some additional ControlFile fields are set in WriteControlFile() */
    WriteControlFile();
```

当用户使用 `pg_ctl` 命令启动数据库时，`postmaster` 进程会读取 `pg_control` 文件的内容，代码调用路径为：

```
PostmasterMain()-->SetCaseGucOption()-->GetCaseSwitchModeFromControl()
```

读取到参数值后调用 `SetConfigOption()` 函数进行赋值。

在每个新的后端进程开始的时候，由于是从 `postmaster` 进程 `fork` 而来，会自动拥有相同的 `ivorysql.identifier_case_switch` 参数值。首先处理 `startup` 包，其中如果包含 `database` 或者 `user` 这两个参数，则根据 `identifier_case_switch` 对参数值做相应的处理。

源代码为：

```
BackendMain()->BackendInitialize()-->ProcessStartupPacket()
```

另外，在处理用户的 SQL 语句时，如果包含标识符，也会进行同样的处理，代码分布在：SplitIdentifierString()，quoteOneName() 和 SplitGUCList() 这几个函数中。

...10. sys_guid 函数

目的

IvorySQL的sys_guid()
是一个强大的随机数产生函数，它产生并返回一个由16个字节组成的数据库级别唯一的标识符(原始值)。

实现说明

IvorySQL的sys_guid()函数通过修改插件uuid-ossp的代码实现。为了充分利用uuid的多种基础库，采用如下逻辑：

1. 如果系统有uuid-ossp，就使用uuid_make();
2. 如果系统有uuid-e2fs，就使用 uuid_generate_random();
3. 否则就调用 arc4random();

同时修改代码使得IvorySQL能够自动载入uuid-ossp插件。

...11. 空字符串转null

目的

在Oracle数据库中，空字符串('')被视为NULL值，这是Oracle的一个重要特性。为了兼容Oracle的这一行为，IvorySQL提供了空字符串转NULL功能。当启用该功能后，SQL语句中的空字符串将自动转换为NULL值，从而保证Oracle应用程序在IvorySQL上的行为一致性。

实现说明

参数 ivorysql.enable_emptystring_to_NULL 对应的GUC变量为 enable_emptystring_to_NULL。

在文件 ora_scan.l 中可以看到对这个变量的使用：

```
case xe:
    yylval->str = litbufdup(yyscanner);
    if (strcmp(yylval->str, "") == 0 &&
        ORA_PARSER == compatible_db &&
        enable_emptystring_to_NULL)
    {
        return NULL_P;
    }
    return SCONST;
```

其中 xe 代表被引号包围的字符串：

```
<xe> extended quoted strings (support backslash escape sequences)
```

上面代码的逻辑是，在词法分析时如果遇到空字符串，在空转NULL功能启用的情况下，返回 NULL_P，否则返回 SCONST。

在语法分析文件 ora_gram.y 里，对语句 insert into abc values(''); 是这样解析的：

```
values_clause:
    VALUES '(' expr_list ')'
    {
        SelectStmt *n = makeNode(SelectStmt);

        n->valuesLists = list_make1($3);
        $$ = (Node *) n;
    }

expr_list:      a_expr
        {
            $$ = list_make1($1);
        }

a_expr:       c_expr                    { $$ = $1; }

c_expr:       AexprConst                { $$ = $1; }

AexprConst: Iconst
       | Sconst
```

```
            {
                $$ = makeStringConst($1, @1);
            }
        | NULL_P
            {
                $$ = makeNullAConst(@1);
            }
```

以上代码对词法分析时返回的 NULL_P 或者 SCONST 分别构建对应的节点进行处理。

...12. CALL INTO

目的

当前，PostgreSQL数据库的CALL语句存在以下限制：

- 不支持INTO子句；
- 无法调用有返回值的函数；
- 无法将结果赋值给客户端变量（即 Oracle 中的绑定变量 / host variables）。

为提升对 Oracle 的兼容性，IvorySQL 实现了对 CALL func(…) INTO :var;
语法的支持，允许用户通过绑定变量（如 :x）接收函数返回值，并在行为（如精度检查、错误处理）上与 Oracle 保持一致。

整体设计思路

由于 PostgreSQL/IvorySQL 本身不支持 SQL 层直接向客户端变量赋值，因此本方案采用 “客户端重写 + 服务端协同” 的方式实现：

- 当 CALL 语句包含绑定变量（如 :x）时：

  ```
  客户端将其重写为一个特殊的匿名 PL 块（DO $$ ... $$）；
  使用扩展查询协议发送，以便传递参数类型和精度信息；
  服务端执行该匿名块，并将结果返回给客户端；
  客户端再将结果写入对应的绑定变量。
  ```

- 当 CALL 语句不含绑定变量时：

  ```
  行为与原生 PostgreSQL 完全一致，使用简单查询协议，不做任何重写。
  ```

实现原理

交互式终端

为了在接口中兼容CALL
[INTO]语句，需将其转换为匿名PL/iSQL块，并借助匿名块对OUT参数的支持来实现功能等价。这就要求get_parameter_description函数能够正确识别CALL语句，并在遇到CALL INTO时，返回重写后的PL语句。
相应地，get_hostvariables例程需要将这些信息（如是否为 CALL
语句、是否包含INTO、重写后的语句等）保存到HostVariable结构中。HostVariable的定义如下：

```
typedef struct HostVariable
```

```
{
    HostVariableEntry *hostvars;
    int     length;
    bool    isdostmt;
    bool    iscallstmt; // 是否来自 CALL 语句
    char    *convertcall;   // 重写后的语句
} HostVariable;
```

服务端

在服务端需要修改语法解析器部分，在ora_gram.y中添加CALL INTO语法规则，并在action部分生成重写后的PL语句，如“x := add(1,2);”

```
CallStmt:   CALL func_application
                {
                    CallStmt *n = makeNode(CallStmt);
                    n->funccall = castNode(FuncCall, $2);
                    $$ = (Node *)n;
                }
            | CALL func_application INTO ORAPARAM
                {
                    CallStmt *n = makeNode(CallStmt);
                    OraParamRef *hostvar = makeNode(OraParamRef);
                    char    *callstr = NULL;
                    n->funccall = castNode(FuncCall, $2);
                    hostvar->number = 0;
                    hostvar->location = @4;
                    hostvar->name = $4;
                    n->hostvariable = hostvar;
                    callstr = pnstrdup(pg_yyget_extra(yyscanner)-
>core_yy_extra.scanbuf + @2, @3 - @2);
                    n->callinto = psprintf("%s := %s;", $4, callstr);
                    pfree(callstr);
                    $$ = (Node *)n;
                }
        ;
```

CallStmt结构需要保存INTO子句和转换后的PL语句

```
typedef struct CallStmt
{
  NodeTag   type;
  FuncCall  *funccall;    /* from the parser */
  FuncExpr  *funcexpr;    /* transformed call, with only input args */
```

```
  List    *outargs;    /* transformed output-argument expressions */
  OraParamRef *hostvariable; /* only used for get_parameter_description() */
  char    *callinto;    /* rewrite CALL INTO to a PL assign stmt */
} CallStmt;
```

为区分普通 DO 语句和由 CALL 转换而来的匿名块，语法中新增GENERATED FROM CALL关键字：

```
opt_do_from_where:
            GENERATED FROM CALL         { $$ = true; }
            | /*EMPTY*/                 { $$ = false; }
        ;
```

生成的 DoStmt 节点将设置 do_from_call = true，供执行器识别。

```
typedef struct DoStmt
{
  NodeTag   type;
  List    *args;      /* List of DefElem nodes */
  List    *paramsmode;  /* List of parameters mode */
  List    *paramslen;   /* List of length for parameter datatypes */
  bool    do_from_call;  /* True if DoStmt is come from CallStmt */
} DoStmt;
```

在IVY接口中，占位符信息是通过一个名为get_parameter_description的集合返回函数（SRF）获取的。该函数需要能够识别输入语句的类型，并在遇到CALL INTO语句时，返回重写后的PL/iSQL赋值语句。
为此，IvorySQL对该函数的返回结构（TupleDesc）进行了扩展：新增了一个hint 字段，专门用于返回CALL INTO语句重写后的PL代码；对于其他类型的语句，该字段保持为NULL。
此外，原函数结果集的第一条元组的第一个字段原本仅用true/false来区分语句是否为匿名块。为了更准确地识别语句类型（尤其是CALL语句），现已将其修改为返回对应解析树的 CommandTag。
所有这些元数据信息最终会被封装到一个用户上下文结构中，以便在 SRF
函数的多次调用之间高效传递和复用。

```
{
    OraParamExtralData *extral;
    const char     *cmdtag;
    char        *callintoexpr;
} outparam_fctx;
```

接口层 CALL涉及的ivy前缀的接口包括：

IvyStmtExecute

IvyStmtExecute2

IvyexecPreparedStatement

IvyexecPreparedStatement2

在上述接口中，用户传入的CALL
[INTO]语句会被重写为一种“特殊”的匿名块语句。为了明确标识这类由CALL转换而来的匿名块，在接口的语句类型定义中新增了一种专用类型。该类型的作用是在IvyHandleDostmt中正确识别此类语句，并生成形如 DO ... USING … — GENERATED FROM CALL 的执行语句。

```
typedef enum IvyStmtType
{
    IVY_STMT_UNKNOW,
    IVY_STMT_DO,
    IVY_STMT_DOFROMCALL, /* new statement type */
    IVY_STMT_DOHANDLED,
    IVY_STMT_OTHERS
} IvyStmtType;
```

在重写 CALL 语句时，如果遇到调用函数的 CALL INTO
语句，接口需要对绑定变量的顺序进行内部调整。这一调整对用户是完全透明的：用户在绑定参数时，只需按照 CALL 语句中出现的顺序操作即可——即 INTO 子句中的变量在原语句中位于最后。

然而，在重写生成的特殊匿名块中，该 INTO
变量会作为赋值表达式的左值（即第一个参数）出现。因此，接口必须在内部将绑定顺序正确调整，确保执行逻辑与用户预期一致。

所有涉及此逻辑的接口例程都需要实现这一处理，相关例程如下：

Ivyreplacenamebindtoposition

Ivyreplacenamebindtoposition2

Ivyreplacenamebindtoposition3

在IvyexecPreparedStatement 和 IvyexecPreparedStatement2 这类接口中，用户需要显式提供每个参数的 paramvalues、paramlengths、paramformats 和 parammode。对于 CALL
语句，这些参数数组中的元素顺序必须根据重写后的匿名块结构进行位置调整，以确保绑定与执行逻辑一致。

其中，IvyexecPreparedStatement2 更为特殊：它要求用户额外提供一个 IvyBindOutInfo*
类型的输出绑定列表。该列表不仅用于绑定 OUT 参数，还被 IvyAssignPLISQLOutParameter 在获取 PL/iSQL 过程返回结果时用来识别每个 OUT 参数的数据类型。因此，在处理 CALL语句时，接口会先对用户传入的 IvyBindOutInfo*列表进行位置重排（将INTO对应的输出变量移至首位），再将其写入IvyPreparedStatement 语句句柄中，供后续赋值使用。

关于输出参数的精度处理：当CALL语句中的输出绑定变量与实际返回值的精度不匹配时，系统可能报错，也可能自动截断——具体行为取决于绑定变量的数据类型是否与过程/函数声明的参数类型完全一致。
在PL/iSQL inline
handler中，每个OUT参数的精确数据类型均可通过ParamListInfo在绑定阶段获取。如果当前执行的匿名块是由CALL语句转换而来的特殊DoStmt，那么在执行赋值时，系统会进行如下判断：

若ParamListInfo中记录的类型与函数/存储过程形参的类型完全相同，则采用强制类型转换赋值；
否则，采用隐式类型转换赋值。

这一机制旨在兼容Oracle的行为，确保在类型不完全匹配时仍能安全、合理地完成赋值。

```
-- 原始 CALL语句:
CALL my_func(:in1, :in2) INTO :out;
-- 重写为:
```

```
do $$BEGIN
  :out := my_func(:in1, :in2);
END$$ using
  out INOUT, in1 INOUT, in2 INOUT
  paramslength -1,-1,-1
GENERATED FROM CALL;
```

内置函数

...1. sys_context

IvorySQL提供兼容Oracle内置函数 `SYS_CONTEXT('namespace', 'parameter' [, length ])`，返回当前时刻与给定上下文关联参数的值，可以在SQL和PLSQL语言中使用。

提供以下命名空间：

- USERENV - 用于描述当前会话。
- SYS_SESSION_ROLES - 指示某个特定角色在当前会话中是否已启用。

实现原理

`SYS_CONTEXT` 的实现原理是通过动态查询系统表与postgres的内置函数，确保结果的实时性， 使用 `SECURITY INVOKER` 确保函数以调用者的权限执行，避免权限提升问题。 该方法的实现逻辑如下：

```
CREATE OR REPLACE FUNCTION sys.sys_context(a varchar2, b varchar2)
RETURNS varchar2 AS $$
DECLARE
    res varchar2;
BEGIN
    IF upper(a) = 'USERENV' THEN
      CASE upper(b)
          WHEN 'CURRENT_SCHEMA' THEN
            SELECT current_schema() INTO res;
          WHEN 'LANG' THEN
            SELECT sys.get_lang() INTO res;
          ...
          ELSE
            RAISE EXCEPTION 'invalid USERENV parameter: %', b;
      END CASE;
    ELSIF upper(a) = 'SYS_SESSION_ROLES' THEN
        CASE upper(b)
            WHEN 'LOGIN' THEN
                SELECT CASE WHEN rolcanlogin = 't' THEN 'TRUE' ELSE 'FALSE' END INTO
res FROM pg_roles WHERE oid = current_user::regrole::oid;
...
            ELSE
```

```
                RAISE EXCEPTION 'invalid SYS_SESSION_ROLES parameter: %', b;
        END CASE;
    ELSE
      SELECT current_setting(a||'.'||b, true) INTO res;
    END IF;
    RETURN res;
END;
$$ LANGUAGE plisql SECURITY INVOKER;
```

命名空间USERENV支持的参数

参数名称	返回值
CURRENT_SCHEMA	当前活动默认模式的名称。在会话期间，可以通过 ALTER SESSION SET CURRENT_SCHEMA 语句更改该值。该值也可能在会话期间发生变化，以反映任何活动定义者权限对象的所有者。如果直接在视图定义的正文中使用，那么将返回在执行使用视图的游标时使用的默认模式。
CURRENT_SCHEMAID	当前活动的默认模式的标识符。
SESSION_USER	当前session的用户 会话用户（登录用户）的名称。在数据库会话期间，随着 Real Application Security 会话的附加或分离，该名称可能会发生变化。对于企业用户，返回模式。对于其他用户，返回数据库用户名。如果数据库会话当前附加了 Real Application Security 会话，则返回用户 XS$NULL。
SESSION_USERID	当前session的用户ID。
PROXY_USER	代表 SESSION_USER 打开当前会话的数据库用户名称。
PROXY_USERID	代表 SESSION_USER 打开当前会话的数据库用户的标识符。
CURRENT_USER	当前会话中执行操作的用户。
CURRENT_USERID	当前会话中执行操作的用户标识符。
CURRENT_EDITION_NAME	当前版本的名字。
CLIENT_PROGRAM_NAME	客户端程序的名称。

参数名称	返回值
IP_ADDRESS	客户端的IP地址。
HOST	客户端主机的名称。
ISDBA	如果当前用户是数据库管理员，则返回TRUE。
LANG	语言的缩写名称，比现有的 “LANGUAGE ” 参数更简短。
LANGUAGE	当前会话的使用的语言和地区，以及数据库字符集，格式为：language_territory.characterset。
NLS_DATE_FORMAT	当前session的日期格式。
PLATFORM_SLASH	在用户的平台上用作文件路径分隔符的斜线字符（Unix或Linux是’ / ’，Windows是’ \ ’）。
DB_NAME	当前连接的CDB的名称。
SID	当前会话的系统标识符。
SESSIONID	审计会话标识符。不能在分布式 SQL 语句中使用此属性。

CLIENT_INFO	返回最多 64 字节的用户会话信息，应用程序可使用 DBMS_APPLICATION_INFO 包存储这些信息。
ENTRYID	当前审计条目编号。审计条目编号序列在细粒度审计记录和常规审计记录之间共享。不能在分布式 SQL 语句中使用此属性。只有通过标准或细粒度审计的审计处理程序才能看到正确的审计条目标识符。
TERMINAL	当前会话客户端的操作系统标识符。在分布式 SQL 语句中，该属性返回本地会话的标识符。在分布式环境中，仅支持远程 SELECT 语句，不支持远程 INSERT、UPDATE 或 DELETE 操作。(此参数的返回长度可能因操作系统而异)。

命名空间SYS_SESSION_ROLES支持的参数

参数名称	返回值
DBA	如果当前用户是数据库管理员，则返回TRUE。
LOGIN	如果当前用户是登录角色，则返回TRUE。
CREATEROLE	如果当前session的用户具有创建角色的权限，则返回TRUE。
CREATEDB	如果当前session的用户具有创建数据库的权限，则返回TRUE。

...2. userenv

IvorySQL提供兼容Oracle内置函数 `USERENV('parameter')` ,用于返回当前会话的相关信息。
这是一个遗留函数，建议使用 `SYS_CONTEXT` 函数及其内置的 `USERENV` 命名空间来获取当前功能。

实现原理

`USERENV` 通过Bison规则解析函数调用，检查参数合法性并映射到对应的具体SQL函数。 解析规则在 `ora_gram.y` 中实现，逻辑如下：

```
USERENV '(' Sconst ')'
    {
        char *normalized_param = downcase_identifier($3, strlen($3), true, true);

        #define CHECK_AND_CALL(param, func_name) \
            if (strcmp(normalized_param, param) == 0) \
                $$ = (Node *) makeFuncCall(OracleSystemFuncName(func_name), NIL,
COERCE_EXPLICIT_CALL, @1);

        CHECK_AND_CALL("client_info", "get_client_info")
        else CHECK_AND_CALL("entryid", "get_entryid")
        else CHECK_AND_CALL("terminal", "get_terminal")
        else CHECK_AND_CALL("isdba", "get_isdba")
        else CHECK_AND_CALL("lang", "get_lang")
        else CHECK_AND_CALL("language", "get_language")
        else CHECK_AND_CALL("sessionid", "get_sessionid")
        else CHECK_AND_CALL("sid", "get_sid")
        else
            ereport(ERROR,
```

```
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                        errmsg("invalid USERENV parameter: \"%s\".", $3)));
        #undef CHECK_AND_CALL
    }
```

具体功能则是在 `builtin_functions—1.0.sql` 中实现。例如：

```
CREATE OR REPLACE FUNCTION sys.get_language()
RETURNS varchar2
AS $$
    SELECT (regexp_split_to_array(current_setting('lc_monetary'), '\.'))[1]||'.'
||pg_client_encoding();
$$ LANGUAGE sql STRICT;
```

USERENV支持的参数

参数名称	返回值
sid	当前session的用户ID。
sessionid	返回审计会话标识符。
language	返回数据库当前会话的语言、地域和字符集。
lang	返回ISO缩写语言名称。
isdba	如果用户已经被认证为管理员;或者是通过操作系统或口令文件具有管理员特权的，返回“TRUE"，否则返回"FALSE"。

..1. 国标GB18030

目的

PostgreSQL 服务端提供了对 GB18030
字符集的全面支持。GB18030是中国国家标准，旨在包含所有汉字和多种少数民族文字，实现与Unicode的统一。在PostgreSQL中正确配置和使用GB18030字符集，对于处理和存储需要符合此标准的中文数据至关重要。

服务端 GB18030 支持应当具备以下特性：

1. 支持 GB18030 作为服务端编码：initdb -E GB18030 可用，SHOW server_encoding 显示为 GB18030。
2. 提供 GB18030 <→ UTF8 的双向转换。
3. 支持多字节边界判定。

实现说明

initdb时通过-E指定GB18030或GB18030_2022

PostgreSQL已支持GB18030-
2000版本作为客户端编码，通过扩展的方式支持GB18030_2022字符集与UTF的转换。

修改pg_enc来实现可指定GB18030作为服务端编码，PostgreSQL编码框架中增加底层函数以供pg内核调用。

设置一个全局变量is_load_gb18030_2022,默认为true,当用户指定-
E选项时,在get_encoding_id中判断其设置的是否为gb18030_2022,如果是,将其字符串转为gb18030,然后将is_load_gb18030_2022 设为true,如果-E 选项为Gb18030,将其设为false。

在适当位置判断是否要加载插件,如果是,执行load_gb18030_2022,并将ivorysql.conf中的shared_preload_library添加gb18030_2022。

```
if (encoding_name && *encoding_name)
{
    encoding_name_modify = pg_strdup(encoding_name);
    if(pg_strcasecmp(encoding_name,"gb18030_2022") == 0)
    {
        encoding_name_modify = pg_strdup("gb18030");
        is_load_gb18030_2022 =  true;
    }
    else if(pg_strcasecmp(encoding_name,"gb18030") == 0)
    is_load_gb18030_2022 = false;

    if ((enc = pg_valid_server_encoding((const char *)encoding_name_modify)) >= 0)
        return enc;
}
```

多字节处理

wchar.c，增加 GB18030 的函数指针：

pg_gb180302wchar_with_len(const unsigned char *from, pg_wchar *to, int len) gb18030 → wchar

pg_wchar2gb18030_with_len(const pg_wchar *from, unsigned char *to, int len) wchar → gb18030

pg_gb18030_mblen(const unsigned char *s)：返回 1/2/4。

pg_gb18030_dsplen(const unsigned char *s)：ASCII 显示宽度 1；其它按 1 处理（与UTF8一致）。

pg_gb18030_verifier(const unsigned char *s, int len)：校验字节范围，拒绝非法序列。

与客户端的交互

接收数据: 如果一个使用 UTF-8 编码的客户端连接上来，服务端在接收到数据后，会调用其内部的 utf8_to_gb18030 函数，将数据转换为 GB18030 格式，然后才进行验证和存储。

发送数据: 当该客户端执行 SELECT 查询时，服务端会从磁盘/内存中读取原生的 GB18030 数据，然后调用 gb18030_to_utf8 函数将其转换为 UTF-8 格式，最后再通过网络协议发送给客户端。

新增GB18030-2022.xml数据文件，通过perl脚本解析为map文件，提供 gb18030_to_utf8() 与 utf8_to_gb18030()，优先表驱动，覆盖不到的区间通过算法映射。

```
static inline uint32
unicode_to_utf8word(uint32 c)
{
    uint32      word;

    if (c <= 0x7F)
    {
```

```
        word = c;
    }
    else if (c <= 0x7FF)
    {
        word = (0xC0 | ((c >> 6) & 0x1F)) << 8;
        word |= 0x80 | (c & 0x3F);
    }
    else if (c <= 0xFFFF)
    {
        word = (0xE0 | ((c >> 12) & 0x0F)) << 16;
        word |= (0x80 | ((c >> 6) & 0x3F)) << 8;
        word |= 0x80 | (c & 0x3F);
    }
    else
    {
        word = (0xF0 | ((c >> 18) & 0x07)) << 24;
        word |= (0x80 | ((c >> 12) & 0x3F)) << 16;
        word |= (0x80 | ((c >> 6) & 0x3F)) << 8;
        word |= 0x80 | (c & 0x3F);
    }

    return word;
}

static uint32
conv_18030_2022_to_utf8(uint32 code)
{
#define conv18030(minunicode, mincode, maxcode) \
    if (code >= mincode && code <= maxcode) \
        return unicode_to_utf8word(gb_linear(code) - gb_linear(mincode) + minunicode)

    conv18030(0x0452, 0x8130D330, 0x8136A531);
    conv18030(0x2643, 0x8137A839, 0x8138FD38);
    conv18030(0x361B, 0x8230A633, 0x8230F237);
    conv18030(0x3CE1, 0x8231D438, 0x8232AF32);
    conv18030(0x4160, 0x8232C937, 0x8232F837);
    conv18030(0x44D7, 0x8233A339, 0x8233C931);
    conv18030(0x478E, 0x8233E838, 0x82349638);
    conv18030(0x49B8, 0x8234A131, 0x8234E733);
    conv18030(0x9FA6, 0x82358F33, 0x8336C738);
    conv18030(0xE865, 0x8336D030, 0x84308534);
    conv18030(0xFA2A, 0x84309C38, 0x84318537);
```

```
    conv18030(0xFFE6, 0x8431A234, 0x8431A439);
    conv18030(0x10000, 0x90308130, 0xE3329A35);
    /* No mapping exists */
    return 0;
}
```

参考指南

.1. 工具参考

工具一览表

这部份包含IvorySQL客户端应用和工具的参考信息。不是所有这些命令都是通用工具，某些需要特殊权限。这些应用的共同特征是它们可以被运行在任何主机上，而不管数据库服务器在哪里。

当在命令行上指定用户和数据库名时，它们的大小写会被保留 —
空格或特殊字符的出现可能需要使用引号。表名和其他标识符的大小写不会被保留并且可能需要使用引号。

分类	工具名称	描述
客户端工具	clusterdb	clusterdb是一个工具，它用来对一个IvorySQL数据库中的表进行重新聚簇。它会寻找之前已经被聚簇过的表，并且再次在最后使用过的同一个索引上对它们重新聚簇。没有被聚簇过的表将不会被影响。clusterdb是SQL命令 CLUSTER 的一个包装器。在通过这个工具和其他方法访问服务器来聚簇数据库之间没有实质性的区别。
	createdb	createdb创建一个新的IvorySQL数据库。通常，执行这个命令的数据库用户将成为新数据库的所有者。但是，如果执行用户具有合适的权限，可以通过 `-O` 选项指定一个不同的所有者。createdb是SQL命令 `CREATE DATABASE`的一个包装器。在通过这个工具和其他方法访问服务器来创建数据库之间没有实质性的区别。
	createuser	createuser创建一个新的IvorySQL用户（或者更准确些，是一个角色）。只有超级用户和具有 `CREATEROLE` 特权的用户才能创建新用户，因此createuser必须被以上两种用户调用。如果你希望创建一个新的超级用户，你必须作为一个超级用户连接，而不仅仅是具有 `CREATEROLE` 特权。作为一个超级用户意味着绕过数据库中所有访问权限检查的能力，因此超级用户访问权限不能轻易被授予。createuser是SQL命令 `CREATE ROLE`的一个包装器。在通过这个工具和其他方法访问服务器来创建用户之间没有实质性的区别。
	dropdb	ropdb毁掉一个现有的IvorySQL数据库。执行这个命令的用户必须是一个数据库超级用户或该数据库的拥有者。dropdb是SQL命令 DROP DATABASE 的一个包装器。在通过这个工具和其他方法访问服务器来删除数据库之间没有实质性的区别。
	dropuser	dropuser移除一个已有的IvorySQL用户。只有超级用户以及具有 `CREATEROLE` 特权的用户能够移除IvorySQL用户（要移除一个超级用户，你必须自己是一个超级用户）。dropuser是SQL命令 DROP ROLE 的一个包装器。在通过这个工具和其他方法访问服务器来删除用户之间没有实质性的区别。
	ecpg	`ecpg` 是用于 C 程序的嵌入式 SQL 预处理器。它通过将 SQL 调用替换为特殊函数调用把带有嵌入式 SQL 语句的 C 程序转换为普通 C 代码。输出文件可以被任何 C 编译器工具链处理。`ecpg` 将把命令行中给出的每一个输入文件转换为相应的 C 输出文件。如果输入文件名没有任何扩展名，则假定为 `.pgc`。文件扩展名将由 `.c` 替换以构造输出文件名。 但是输出文件名可以使用 `-o` 选项覆盖。如果输入文件名只是 `-`，`ecpg` 从标准输入读取程序（并写入标准输出，除非用 `-o` 重写）。

	pg_amcheck	pg_amcheck支持对一个或多个数据库运行 amcheck 的损坏检查函数，并提供选项来选择要检查的模式、表和索引、要执行的检查类型以及是否并行执行检查，如果是，按并行数建立连接并使用。当前仅支持表关系和btree索引。其他关系类型将自动跳过。如果指定了 `dbname`，则它应该是要检查的单个数据库的名称，并且不应该存在其他数据库选择选项。否则，如果存在任何数据库选择选项，将检查所有匹配的数据库。如果不存在此类选项，将选中默认数据库。数据库选择选项包括 `--all`，`--database` 和 `--exclude-database`。它们还包括 `--relation`，`--exclude-relation`，`--table`，`--exclude-table`，`--index`，和 `--exclude-index`，但仅当这些选项与三段式模式一起使用时（例如，`mydb*.myschema*.myrel*`）。最后，它们包括 `--schema` 和 `--exclude-schema` 当这些选项与两段式模式一起使用时（例如 `mydb*.myschema*`）。
	pg_basebackup	pg_basebackup被用于获得一个正在运行的IvorySQL数据库集簇的基础备份。获得这些备份不会影响数据库的其他客户端，并且可以被用于时间点恢复，以及用作一个日志传送或流复制后备服务器的开始点。pg_basebackup对数据库群集的文件进行精确复制，同时确保服务器自动进入和退出备份模式。备份总是从整个数据库集簇获得，不可能备份单个数据库或数据库对象。关于选择性备份，必须使用一个像 pg_dump 的工具。备份通过一个使用复制协议常规IvorySQL连接制作。该连接必须由一个具有 `REPLICATION` 权限或者具有超级用户权限的用户ID建立，并且 `pg_hba.conf`必须允许该复制连接。该服务器还必须被配置，使 max_wal_senders 设置得足够高以提供至少一个walsender用于备份以及一个用于WAL流（如果使用流）。在同一时间可以有多个 `pg_basebackup` 运行，但是从性能的角度来说，只进行一次备份并且复制结果通常更好。pg_basebackup不仅能从主控机也能从后备机创建一个基础备份。要从后备机获得一个备份，设置后备机让它能接受复制连接（也就是，设置 `max_wal_senders` 和 hot_standby，并且适当配置其 `pg_hba.conf`）。你将也需要在主控机上启用 full_page_writes。注意在来自后备机的备份中有一些限制：不会在被备份的数据库集簇中创建备份历史文件。 pg_basebackup不能强制后备服务器在备份结束时切换到新的WAL文件。当正在使用 `-X none` 时，如果服务器上的写活动比较低，pg_basebackup可能需要等待很长时间，以便切换和归档备份所需要的最后的WAL文件。 在这种情况下，在主服务器上运行 `pg_switch_wal` 以触发立即的WAL文件切换可能是有用的。 如果在备份期间后备机被提升为主控机，备份会失败。 备份所需的所有 WAL 记录必须包含足够的全页写，这要求你在主控机上启用 `full_page_writes` 并且不使用一个类似pg_compresslog的工具以 `archive_command` 从 WAL 文件中移除全页写。每当pg_basebackup进行基本备份时，服务器的 `pg_stat_progress_basebackup` 视图将报告备份的进度。
	pgbench	pgbench是一种在IvorySQL上运行基准测试的简单程序。它可能在并发的数据库会话中一遍一遍地运行相同序列的 SQL 命令，并且计算平均事务率（每秒的事务数）。默认情况下，pgbench会测试一种基于 TPC-B 但是要更宽松的场景，其中在每个事务中涉及五个 `SELECT`、`UPDATE` 以及 `INSERT` 命令。但是，通过编写自己的事务脚本文件很容易用来测试其他情况。
	pg_config	pg_config工具打印当前安装版本的IvorySQL的配置参数。它的设计目的之一是便于想与IvorySQL交互的软件包能够找到所需的头文件和库。

	pg_dump	pg_dumppg_dump是用于备份一种IvorySQL数据库的工具。即使数据库正在被并发使用，它也能创建一致的备份。pg_dump不阻塞其他用户访问数据库（读取或写入）。pg_dump只转储单个数据库。要备份一个集簇或者集簇中对于所有数据库公共的全局对象（例如角色和表空间），应使用pg_dumpall。转储可以被输出到脚本或归档文件格式。脚本转储是包含 SQL 命令的纯文本文件，它们可以用来重构数据库到它被转储时的状态。要从这样一个脚本恢复，将它输入到psql。脚本文件甚至可以被用来在其他机器和其他架构上重构数据库。在经过一些修改后，甚至可以在其他 SQL 数据库产品上重构数据库。另一种可选的归档文件格式必须与 pg_restore 配合使用来重建数据库。它们允许pg_restore能选择恢复什么，或者甚至在恢复之前对条目重排序。归档文件格式被设计为在架构之间可移植。当使用归档文件格式之一并与pg_restore组合时，pg_dump提供了一种灵活的归档和传输机制。pg_dump可以被用来备份整个数据库，然后pg_restore可以被用来检查归档并/或选择数据库的哪些部分要被恢复。最灵活的输出文件格式是“自定义”格式（`-Fc`）和“目录”格式（`-Fd`）。它们允许选择和重排序所有已归档项、支持并行恢复并且默认是压缩的。“目录”格式是唯一一种支持并行转储的格式。当运行pg_dump时，我们应该检查输出中有没有任何警告（打印在标准错误上）
	pg_dumpall	pg_dumpall工具可以一个集簇中所有的IvorySQL数据库写出到（“转储”）一个脚本文件。该脚本文件包含可以用作 psql 的输入SQL命令来恢复数据库。它会对集簇中的每个数据库调用 pg_dump 来完成该工作。pg_dumpall还转储对所有数据库公用的全局对象（pg_dump 不保存这些对象），也就是说数据库角色和表空间都会被转储。目前这包括适数据库用户和组、表空间以及适合所有数据库的访问权限等属性。因为pg_dumpall从所有数据库中读取表，所以你很可能需要以一个数据库超级用户的身份连接以便生成完整的转储。同样，你也需要超级用户特权执行保存下来的脚本，这样才能增加角色和组以及创建数据库。SQL 脚本将被写出到标准输出。使用 `-f` / `--file` 选项或者 shell 操作符可以把它重定向到一个文件。pg_dumpall需要多次连接到IvorySQL服务器（每个数据库一次）。如果你使用口令认证，可能每次都会要求口令。这种情况下使用一个 `~/.pgpass` 会比较方便
	pg_isready	pg_isready是一个用来检查一个IvorySQL数据库服务器的连接状态的工具。其退出状态指定了连接检查的结果。
	pg_receivewal	pg_receivewal被用来从一个运行着的IvorySQL集簇以流的方式得到预写式日志。预写式日志会被使用流复制协议以流的方式传送，并且被写入到文件的一个本地目录。这个目录可以被用作归档位置来做一次使用时间点恢复的恢复。当预写式日志在服务器上被产生时，pg_receivewal实时以流的方式传输预写式日志，并且不像 archive_command 那样等待段完成。由于这个原因，在使用pg_receivewal时不必设置 archive_timeout。与 IvorySQL 后备服务器上的 WAL 接收进程不同，pg_receivewal默认只在一个 WAL 文件被关闭时才刷入 WAL 数据。要实时刷入 WAL 数据，必须指定选项 `--synchronous`。由于pg_receivewal不应用于WAL，当 synchronous_commit 等于 `remote_apply` 时，你将不允许它成为同步备用。如果发生这样的情况，它将成为一个永远不能拉起的备用数据库，并且会导致事务提交阻塞。 为了避免这种情况，你应该为 synchronous_standby_names 配置一个适当的值，或规定为pg_receivewal 的 `application_name` 与它不匹配，或将 `synchronous_commit` 的值更改为 `remote_apply` 以外的内容。预写式日志在一个常规IvorySQL连接上被以流式传送，并且使用复制协议。连接必须由一个具有 `REPLICATION` 权限的用户或者一个超级用户建立，并且 `pg_hba.conf` 必须允许复制连接。服务器也必须被配置一个足够高的 max_wal_senders 来至少留出一个可用会话给流。首先，扫描WAL段文件所写入的目录，并发现最新完成的段文件，作为下一个段文件的开始的起始点。这是独立计算的，根据用于压缩每个段的压缩方法。如果用前面的方法无法计算出起点，最新的WAL刷写位置用作由服务器通过 `IDENTIFY_SYSTEM` 命令的报告。

	pg_recvlogical	pg_recvlogical 控制逻辑解码复制槽以及来自这种复制槽的流数据。它会创建一个复制模式的连接，因此它受到和 pg_receivewal 相同的约束，还有逻辑复制的约束。pg_recvlogical 与逻辑解码SQL接口的peek和get模式没有等效性。它咋接收到数据以及干净地退出时，它会惰性地发送数据的确认。为了检查一个槽上还未消费的待处理数据，可以使用 pg_logical_slot_peek_changes。
	pg_restore	pg_restore是一个用来从 pg_dump 创建的非文本格式归档恢复IvorySQL数据库的工具。它将发出必要的命令把该数据库重建成它被保存时的状态。这些归档文件还允许pg_restore选择恢复哪些内容或者在恢复前对恢复项重排序。这些归档文件被设计为可以在不同的架构之间迁移。pg_restore可以在两种模式下操作。如果指定了一个数据库名称，pg_restore会连接那个数据库并且把归档内容直接恢复到该数据库中。否则，会创建一个脚本，其中包含着重建该数据库所必要的 SQL 命令，它会被写入到一个文件或者标准输出。这个脚本输出等效于pg_dump的纯文本输出格式。因此，一些控制输出的选项与pg_dump的选项类似。显然，pg_restore无法恢复不在归档文件中的信息。例如，如果归档使用“以 INSERT 命令转储数据”选项创建， pg_restore将无法使用 COPY 语句装载数据。
	pg_verifybackup	pg_verifybackup用于根据备份时服务器生成的 backup_manifest 检查使用 pg_basebackup 进行的数据库群集备份的完整性。备份必须以“普通”格式存储；“tar”格式的备份可以在解压缩后进行检查。需要注意的是，由pg_verifybackup执行的验证不包括也不可能包括运行中的服务器在尝试使用备份时执行的所有检查。 即使使用此工具，也应执行测试还原，并验证生成的数据库是否按预期工作，以及它们是否包含正确的数据。但是，pg_verifybackup可以检测到由于存储问题或用户错误而经常出现的许多问题。备份验证分四个阶段进行。首先，pg_verifybackup 读取 backup_manifest 文件。如果该文件不存在、无法读取、格式不正确或无法根据其内部校验和进行验证，pg_verifybackup 将以致命错误终止。其次，pg_verifybackup 将尝试验证当前存储在磁盘上的数据文件是否与服务器打算发送的数据文件完全相同，下面将介绍一些例外情况。 除了少数例外，额外和丢失的文件将被检测到。此步骤将忽略 postgresql.auto.conf、standby.signal 和 recovery.signal 的存在与否或对其的任何修改，因为预计这些文件可能是在备份过程中创建或修改的。它也不会抱怨目标目录中的 backup_manifest 文件或 pg_wal 中的任何内容，即使这些文件不会列在备份清单中。只检查文件；不验证目录的存在与否，除非间接验证：如果目录丢失，则它应该包含的任何文件也必然会丢失。
	psql	psql是一个IvorySQL的基于终端的前端。它让你能交互式地键入查询，把它们发送给IvorySQL，并且查看查询结果。或者，输入可以来自于一个文件或者命令行参数。此外，psql还提供一些元命令和多种类似 shell 的特性来为编写脚本和自动化多种任务提供便利。
	reindexdb	reindexdb是用于重建一个IvorySQL数据库中索引的工具。reindexdb是 SQL 命令 REINDEX的一个包装器。 在通过这个工具和其他方法访问服务器来重索引数据库之间没有实质性的区别。
	vacuumdb	vacuumdb是用于清理一个IvorySQL数据库的工具。vacuumdb也将产生由IvorySQL查询优化器所使用的内部统计信息。

服务器应用	initdb	initdb 创建一个新的IvorySQL数据库集簇。一个数据库集簇是由一个单一服务器实例管理的数据库的集合。一个数据库集簇的创建包括创建存放数据库数据的目录、生成共享目录表（属于整个集簇而不是任何特定数据库的表）并且创建 template1 和 postgres 数据库。当你后来创建一个新的数据库时，任何在 template1 数据库中的东西都会被复制（因此，任何已安装在 template1 中的东西都会被自动地复制到后来创建的每一个数据库中）。 postgres 数据库是便于用户、工具和第三方应用使用的默认数据库。尽管 initdb 将尝试创建指定的数据目录，它可能没有权限（如果想要的数据目录的父目录被根用户拥有）。要在这样一种设置中初始化，作为 root 创建一个空数据目录，然后使用 chown 将该目录赋予给数据库用户账户，再然后 su 成为该数据库用户并运行 initdb。initdb 必须以将拥有该服务器进程的用户运行，因为该服务器需要访问 initdb 创建的文件和目录。因为该服务器不能作为 root 运行，你不能以 root 运行 initdb （事实上它会拒绝这样做）。由于安全原因，由 initdb 创建的新集簇默认将只能由集簇拥有者访问。--allow-group-access 选项允许与集簇拥有者同组的任何用户读取集簇中的文件。这对非特权用户执行备份很有用。initdb 初始化该数据库集簇的默认区域和字符集编码。当一个数据库被创建时，其字符集编码、排序顺序（LC_COLLATE）和字符集类（LC_CTYPE，例如大写、小写、数字）可以被独立设置。initdb 为 template1 数据库确定那些设置，它们将作为所有其他数据库的默认值。要修改默认排序顺序或字符集类，使用 --lc-collate 和 --lc-ctype 选项。除 C 或 POSIX 之外的排序顺序也有性能罚值。由于这些原因，在运行 initdb 时选择正确的区域很重要。余下的区域分类可以在服务器启动之后改变。你也可以使用 --locale 来为所有区域分类设置默认值，包括排序顺序和字符集类。所有服务器区域值（lc_*）可以通过 SHOW ALL 显示。要修改默认编码，使用 --encoding 。
	pg_archivecleanup	pg_archivecleanup被设计用作 archive_cleanup_command 在作为后备服务器运行时来清理 WAL 文件归档。pg_archivecleanup也可以被用作一个单独的程序来清理 WAL 文件归档。要配置一个后备服务器以使用pg_archivecleanup，把下面的内容放在 postgresql.conf 配置文件中：archive_cleanup_command = 'pg_archivecleanup archivelocation %r' 其中 archivelocation 是要从中移除 WAL 段文件的目录。当被用在 archive_cleanup_command中时，所有逻辑上在 %r 参数的值之前的 WAL 文件都将被从 archivelocation 移除。这能最小化需要被保留的文件数量，同时能保留崩溃后重启的能力。如果对于这台特定的后备服务器，archivelocation 是一个短暂需要的区域，使用这个参数就是合适的，但是当 archivelocation 要用作一个长期的 WAL 归档区域或者当多个后备服务器正在从这个归档位置恢复时，使用这个参数就不合适。当被用作一个单独的程序时，所有逻辑上在 oldestkeptwalfile 之前的 WAL 文件将被从 archivelocation 中移除。在这种模式中，如果指定了 .partial 或者 .backup 文件名，则 只有该文件前缀将被用作 oldestkeptwalfile。这种对 .backup 文件名的处理允许你移除所有在一个特定基础备份之前归档的 WAL 文件而不出错。
	pg_checksums	pg_checksums在IvorySQL集簇中检查、启用或禁用数据校验和。运行pg_checksums之前，必须彻底关闭服务器。验证校验和时，如果没有校验和错误，则退出状态为零，如果检测到至少一个校验和失败，则退出状态为非零。启用或禁用校验和时，如果操作失败，则退出状态为非零。验证校验和时，集簇中的每个文件都要被扫描。启用校验和时，集簇中的每个文件都会被原地重写。禁用校验和时，仅更新 pg_control 文件。
	pg_controldata	pg_controldata 打印在 initdb 期间初始化的信息，例如目录版本。它也显示关于预写式日志和检查点处理的信息。这种信息是集簇范围的，并且不针对任何一个数据库。这个工具只能由初始化集簇的用户运行，因为它要求对数据目录的读访问。你可以在命令行中指定数据目录，或者使用环境变量 PGDATA。这个工具支持选项 -V 和 --version，它们打印pg_controldata版本并退出。它也支持选项 -? 和 --help，它们输出支持的参数。

	pg_ctl	pg_ctl是一个用于初始化IvorySQL数据库集簇，启动、停止或重启IvorySQL数据库服务器（postgres），或者显示一个正在运行服务器的状态的工具。尽管服务器可以被手工启动，pg_ctl包装了重定向日志输出以及正确地从终端和进程组脱离等任务。它也提供了方便的选项用来控制关闭。
	pg_resetwal	`pg_resetwal` 会清除预写式日志（WAL）并且有选择地重置存储在 `pg_control` 文件中的一些其他控制信息。如果这些文件已经被损坏，某些时候就需要这个功能。当服务器由于这样的损坏而无法启动时，这只应该被用作最后的手段。在运行这个命令之后，就可能可以启动服务器，但是记住数据库可能包含由于部分提交事务产生的不一致数据。你应当立刻转储你的数据、运行 `initdb` 并且重新载入。重新载入后，检查不一致并且根据需要修复之。这个工具只能被安装服务器的用户运行，因为它要求对数据目录的读写访问。出于安全原因，你必须在命令行中指定数据目录。`pg_resetwal` 不使用环境变量 `PGDATA`。如果 `pg_resetwal` 抱怨它无法为 `pg_control` 决定合法数据，你可以通过指定 `-f`（强制）选项强制它继续。在这种情况下，丢失的数据将被替换为看似合理的值。可以期望大部分域是匹配的，但是下一个 OID、下一个事务 ID 和纪元、下一个多事务 ID 和偏移以及 WAL 开始位置域可能还是需要人工协助。这些域可以使用下面讨论的选项设置。如果你不能为所有这些域决定正确的值，`-f` 还是可以被使用，但是恢复的数据库还是值得怀疑：一次立即的转储和重新载入是势在必行的。在你转储之前不要在该数据库中执行任何数据修改操作，因为任何这样的动作都可能使破坏更严重。
	pg_rewind	pg_rewind是用于在集簇的时间线分叉以后，同步一个 IvorySQL 集簇和同一集簇的另一份拷贝的工具。一种典型的场景是在失效后让一个旧的主服务器重新上线，同时有一个后备机跟随着新的主机。成功回放后，目标数据目录的状态类似于源数据目录的基本备份。与进行新的基本备份或使用rsync等工具不同，pg_rewind不需要比较或复制集群中未更改的关系块。仅复制现有关系文件中更改的块；所有其他文件（包括新的关系文件、配置文件和WAL段）都将被完整复制。因此，当数据库很大并且集群之间只有一小部分块不同时，倒带操作比其他方法要快得多。pg_rewind检查源集簇和目标集簇的时间线历史来判断它们在哪一点分叉，并且期望在目标集簇的 `pg_wal` 目录中找到 WAL 来返回到分叉点。分叉点可能会在目标时间线、源时间线或者它们的共同祖先上找到。在典型的失效场景中，目标集簇在分叉后很快就被关闭，这不是问题，但是如果目标集簇在分叉后已经运行了很长时间，旧的 WAL 文件可能已经不存在了。在这样的情况下，您可以手动将它们从WAL存档复制到 `pg_wal` 目录，或使用 `-c` 选项运行pg_rewind以自动从WAL存档检索它们。pg_rewind的使用并不限于失效的场景，例如一个后备服务器可能被提升、运行一些写事务，然后被倒回再次成为一个后备。在运行pg_rewind之后，需要完成WAL重放以使数据目录处于一致状态。当目标服务器再次启动时，它将进入归档恢复，并从分歧点之前的最后一个检查点重放源服务器中生成的所有 WAL。当pg_rewind被运行时有某些 WAL 在源服务器上不可用，并且因此无法被pg_rewind会话所复制，则在目标服务器被启动时必须让这些 WAL 可用。 这可以通过在目标数据目录中创建一个 `recovery.signal` 文件并且在 `postgresql.conf` 中配置适合的 restore_command 来实现。pg_rewind要求目标服务器在 `postgresql.conf` 中启用了 wal_log_hints 选项，或者在用initdb初始化集簇时启用了数据校验。目前默认情况下这两者都没有被打开。http://www.postgresql.org/docs/17/runtime-config-wal.html#GUC-FULL-PAGE-WRITES[full_page_writes] 也必须被设置为 `on`，这是默认的。
	pg_test_fsync	pg_test_fsync是想告诉你在特定的系统上，哪一种 wal_sync_method 最快，还可以在发生认定的 I/O 问题时提供诊断信息。不过，pg_test_fsync 显示的区别可能不会在真实的数据库吞吐量上产生显著的区别，特别是由于很多数据库服务器被它们的预写日志限制了速度。 pg_test_fsync为 `wal_sync_method` 报告以微秒计的平均文件同步操作时间，也能被用来提示用于优化 commit_delay 值的方法。
	pg_test_timing	pg_test_timing是一种度量在你的系统上计时开销以及确认系统时间绝不会回退的工具。收集计时数据很慢的系统会给出不太准确的 `EXPLAIN ANALYZE` 结果。

	pg_upgrade	pg_upgrade允许存储在IvorySQL数据文件中的数据被升级到一个较晚的IvorySQL主版本而无需进行主版本升级通常所需的数据转储/重载。对于次版本升级则不需要这个程序。IvorySQL主发行版本通常会加入新的特性，这些新特性常常会更改系统表的布局，但是内部数据存储格式很少会改变。pg_upgrade使用这一事实来通过创建新系统表并且重用旧的用户数据文件来执行快速升级。如果未来的主发行版本更改了数据存储格式，导致旧数据格式不可读，那么pg_upgrade将无法用于此类升级。（社区将努力避免这种情况）。pg_upgrade会尽力（例如通过检查兼容的编译时设置）确保新旧集簇在二进制上也是兼容的，包括 32/64位二进制。保持外部模块也是二进制兼容的也很重要，不过pg_upgrade无法检查这一点。
	pg_waldump	**pg_waldump** 显示预写式日志（WAL），它主要用于调试或者教育目的。这个工具只能由安装该服务器的用户运行，因为它要求对数据目录的只读访问。
	postgres	**postgres** 是IvorySQL数据库服务器。一个客户端应用为了能访问一个数据库，它会（通过一个网络或者本地）连接到一个运行着的 **postgres** 实例。该 **postgres** 实例接着会开始一个独立的服务器进程来处理该连接。一个 **postgres** 实例总是管理正好一个数据库集簇的数据。一个数据库集簇是一个数据库的集合，它们被存储在一个共同的文件系统位置（“数据区”）上。一个系统上可以同时运行多个 **postgres** 实例，只要它们使用不同的数据区和不同的通信端口（见下文）。**postgres** 启动时需要知道数据区的位置，该位置必须通过 **-D** 选项或 **PGDATA** 环境变量指定，对此是没有默认值的。通常，**-D** 或 **PGDATA** 会直接指向由 initdb 创建的数据区目录。默认情况下，**postgres** 会在前台启动并将日志消息打印到标准错误流。但在实际应用中，**postgres** 应当作为一个后台进程启动，而且多数是在系统启动时自动启动。**postgres** 还能在单用户模式中被调用。这种模式的主要用途是在启动过程中由 initdb 使用。有时候它也被用于调试或者灾难性恢复。注意，运行一个单用户模式服务器并不真地适合调试服务器，因为不会发生实际的进程间通信和锁定。当从 shell 中调用单用户模式时，用户可以输入查询并且结果会被以一种更适合开发者阅读（不适合普通用户）的形式打印在屏幕上。在单用户模式中，会话用户将被设置为 ID 为 1的用户，并且这个用户会被隐式地赋予超级用户权限。该用户不必实际存在，因此单用户模式运行可以被用来对某些意外损坏的系统目录进行手工恢复。

客户端工具

clusterdb

大纲

clusterdb [**connection-option**⋯] [**--verbose** | **-v**] [**--table** | **-t table**] ⋯ [**dbname**]

```
clusterdb` [*`connection-option`*...] [ `--verbose` | `-v` ] `--all` | `-a
```

选项

clusterdb接受下列命令行参数：

- **-a --all**

聚簇所有数据库。

- **[-d] dbname [--dbname=] dbname**

当不使用 **-a**/ **--all** 时，指定要被聚簇的数据库名称。 如果数据库名称未指定，则从环境变量 **PGDATABASE**

中读取数据库名称。 如果该环境变量也没有被设置，则使用为连接指定的用户名作数据库名。 `dbname` 可以是 connection string。 如果是这样，连接时的字符串参数将覆盖所有冲突的命令行选项。

- `-e --echo`

回显clusterdb生成并发送给服务器的命令。

- `-q --quiet`

不显示进度消息。

- `-t table --table=table`

只聚簇 `table`。可以通过写多个 `-t` 开关来聚簇多个表。

- `-v --verbose`

在处理期间打印详细信息。

- `-V --version`

打印clusterdb版本并退出。

- `-? --help`

显示关于clusterdb命令行参数的帮助并退出。clusterdb也接受下列命令行参数用于连接参数：

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制clusterdb在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，clusterdb将自动提示要求一个口令。但是，clusterdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

- `--maintenance-db= dbname`

当使用 `-a` / `--all` 时，指定要连接到的数据库名称来发现哪些其他数据库应该被聚簇。
如果没有指定，将使用 `postgres` 数据库。而如果它也不存在，将使用 `template1`。 这可以是 connection string。如果是这样，连接时的字符串参数将覆盖所有冲突的命令行选项。
另外，连接到其他数据库时，除了数据库名字本身其他的连接时字符串参数将被重新使用。

环境

- `PGDATABASE PGHOST PGPORT PGUSER`

> 默认连接参数

- PG_COLOR

> 规定在诊断消息中是否使用颜色。可选的值为 `always` ， `auto` ， `never`

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 CLUSTER 和 psql
中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要聚簇数据库 test：

```
$ clusterdb test
```

要聚簇在数据库 xyzzy 中的一个表 foo：

```
$ clusterdb --table=foo xyzzy
```

createdb

大纲

createdb [connection-option…] [option…] [dbname [description]]

选项

createdb接受下列命令行参数：

- dbname

指定要被创建的数据库名。该名称必须在这个集簇中所有IvorySQL数据库中唯一。默认是创建一个与当前系统用户同名的数据库。

- description

指定与新创建的数据库相关联的一段注释。

- -D tablespace --tablespace=tablespace

指定该数据库的默认表空间（这个名称被当做一个双引号引用的标识符处理）。

- -e --echo

回显createdb生成并发送到服务器的命令。

- -E encoding --encoding=encoding

指定要在这个数据库中使用的字符编码模式。

- `-l locale --locale=locale`

指定要在这个数据库中使用的区域。这等效于同时指定`--lc-collate`和`--lc-ctype`。

- `--lc-collate=locale`

指定要在这个数据库中使用的 LC_COLLATE 设置。

- `--lc-ctype=locale`

指定要在这个数据库中使用的 LC_CTYPE 设置。

- `-O owner --owner=owner`

指定拥有这个新数据库的数据库用户（这个名称被当做一个双引号引用的标识符处理）。

- `-T template --template=template`

指定用于创建这个数据库的模板数据库（这个名称被当做一个双引号引用的标识符处理）。

- `-V --version`

打印createdb版本并退出。

- `-? --help`

显示关于createdb命令行参数的帮助并退出。选项 `-D`、`-l`、`-E`、`-O`和`-T` 对应于底层 SQL 命令 `CREATE DATABASE`的选项，关于这些选项的信息可见该命令的内容。

createdb也接受下列命令行参数用于连接参数：

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制createdb在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，createdb将自动提示要求一个口令。但是，createdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

- `--maintenance-db=dbname`

指定要连接到来发现哪些其他数据库应该被聚簇的数据库名。如果没有指定，将使用 `postgres` 数据库。而如果它也不存在（或者如果它就是要创建新数据库的名称），将使用 `template1`。这可以是 connection string。如果是这样，连接时字符串参数将覆盖所有冲突的命令行选项。

环境

- PGDATABASE

如果被设置，就是要创建的数据库名，除非在命令行中覆盖。

- PGHOST PGPORT PGUSER

默认连接参数。如果没有在命令行或 PGDATABASE 指定要创建的数据库名， PGUSER 也决定要创建的数据库名。

- PG_COLOR

规定在诊断消息中是否使用颜色。 可能的值为 always，auto 和 never。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 CREATE DATABASE和 psql 中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要使用默认数据库服务器创建数据库 demo：

```
$ createdb demo
```

要在主机 eden、端口 5000 上使用 template0 模板数据库创建数据库 demo，这里是命令行命令和底层SQL命令：

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

createuser

createuser — 定义一个新的IvorySQL用户账户

大纲

createuser [connection-option…] [option…] [username]

选项

createuser接受下列命令行参数：

- username

指定要被创建的IvorySQL用户的名称。这个名称必须与这个IvorySQL安装中所有现存角色不同。

- -c number --connection-limit=number

为该新用户设置一个最大连接数。默认值为不设任何限制。

- -d --createdb

新用户将被允许创建数据库。

- `-D --no-createdb`

新用户将不被允许创建数据库。这是默认值。

- `-e --echo`

回显createuser生成并发送给服务器的命令。

- `-E --encrypted`

此选项已过时，但为了实现向后兼容仍然接受。

- `-g role --role=role`

指定一个角色，这个角色将立即加入其中成为其成员。 如果要把这个角色加入到多个角色中作为成员，可以写多个 `-g` 开关。

- `-i --inherit`

新角色将自动继承把它作为成员的角色的特权。这是默认值。

- `-I --no-inherit`

新角色将不会自动继承把它作为成员的角色的特权。

- `--interactive`

如果在命令行没有指定用户名，提示要求用户名，并且在命令行没有指定选项 `-d` / `-D`、 `-r` / `-R`、 `-s` / `-S` 时也提示。

- `-l --login`

新用户将被允许登入（即，该用户名能被用作初始会话用户标识符）。这是默认值。

- `-L --no-login`

新用户将不被允许登入（一个没有登录特权的角色仍然可以作为管理数据库权限的方式而存在）。

- `-P --pwprompt`

如果给定，createuser将发出一个提示要求新用户的口令。如果你没有计划使用口令认证，这就不是必须的。

- `-r --createrole`

新用户将被允许创建新的角色（即，这个用户将具有 `CREATEROLE` 特权）。

- `-R --no-createrole`

新用户将不被允许创建新角色。这是默认值。

- `-s --superuser`

新用户将成为一个超级用户。

- `-S --no-superuser`

新用户将不会成为一个超级用户。这是默认值。

- `-V --version`

打印createuser版本并退出。

- `--replication`

新用户将具有 `REPLICATION` 特权，这在 CREATE ROLE 的文档中有更完整的描述。

- `--no-replication`

新用户将不具有 `REPLICATION` 特权，这在 CREATE ROLE 的文档中有更完整的描述。

- `-? --help`

显示有关createuser命令行参数的帮助并退出。

createuser也接受下列命令行参数作为连接参数：

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接（不是要创建的用户名）。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制createuser在连接到一个数据库之前提示要求一个口令（用来连接到服务器，而不是新用户的口令）。这个选项不是必不可少的，因为如果服务器要求口令认证，createuser将自动提示要求一个口令。但是，createuser将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

环境

- `PGHOST PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`， `auto` 和 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 CREATE ROLE 和 psql 中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

例子

要在默认数据库服务器上创建一个用户 `joe`：

```
$ createuser joe
```

要在默认数据库服务器上创建一个用户 joe 并提示要求一些额外属性：

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

要使用在主机 eden、端口 5000 上的服务器创建同一个用户 joe，并带有显式指定的属性，看看下面的命令：

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

要创建用户 joe 为一个超级用户并且立刻分配一个口令：

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB
CREATEROLE INHERIT LOGIN;
```

在上面的例子中，在录入新口令时新口令并没有真正地被回显，但是为了清晰，我们特意把它列了出来。如你所见，该口令在被发送给客户端之前会被加密。

dropdb

dropdb — 移除一个IvorySQL数据库

大纲

dropdb [**connection-option**…] [**option**…] **dbname**

选项

dropdb接受下列命令行参数：

- **dbname**

指定要被移除的数据库的名字。

- **-e --echo**

回显dropdb生成并发送给服务器的命令。

- **-f --force**

在删除目标数据库之前，尝试终止与该数据库的所有现有连接。有关此选项的详细信息，请参见 DROP DATABASE。

- **-i --interactive**

在做任何破坏性的工作之前发出一个验证提示。

- `-V --version`

打印dropdb版本并退出。

- `--if-exists`

如果数据库不存在也不抛出一个错误。在这种情况下会发出一个提示。

- `-? --help`

显示有关dropdb命令行参数的帮助并退出。

dropdb也接受下列命令行参数作为连接参数：

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作Unix域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的TCP端口或本地Unix域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

不发出口令提示。如果服务器要求口令认证并且没有可用的口令（例如一个 `.pgpass` 文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制dropdb在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，dropdb将自动提示要求一个口令。但是，dropdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

- `--maintenance-db=dbname`

指定一个数据库的名称，将连接到这个数据库以便删除目标数据库。如果没有指定，将使用 `postgres` 数据库。而如果它也不存在（或者它就是正在被删除的目标数据库），将使用 `template1`。这个值可以是一个 连接字符串。
如果是这样，连接字符串参数将覆盖任何有冲突的命令行选项。

环境

- `PGHOST PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto` 以及 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 DROP DATABASE 和 psql
中找找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

示例

要在默认数据库服务器上毁掉数据库 demo：

```
$ dropdb demo
```

要使用在主机 eden、端口5000上的服务器中毁掉数据库 demo，并带有验证和回显，看看下面的命令：

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

dropuser

dropuser — 移除一个IvorySQL用户账户

大纲

dropuser [connection-option…] [option…] [username]

选项

dropuser接受下列命令行参数：

- username

指定要移除的IvorySQL用户的名字。如果没有在命令行指定并且使用了 -i / --interactive
选项，你将被提醒要求一个用户名。

- -e --echo

回显dropuser生成并发送给服务器的命令。

- -i --interactive

在实际移除该用户之前提示要求确认，并且在没有在命令行指定用户名提示要求一个用户名。

- -V --version

打印dropuser版本并退出。

- --if-exists

如果用户不存在时不要抛出一个错误。在这种情况下将发出一个提示。

- -? --help

显示有关dropuser命令行参数的帮助并退出。

dropuser也接受下列命令行参数作为连接参数：

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作Unix域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的TCP端口或本地Unix域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

不发出口令提示。如果服务器要求口令认证并且没有可用的口令（例如一个 `.pgpass` 文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制dropuser在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，dropuser将自动提示要求一个口令。但是，dropuser将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

环境

- `PGHOST PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、`auto` 以及 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 DROP ROLE 和 psql 中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

示例

要从默认数据库服务器移除用户 `joe`：

```
$ dropuser joe
```

要使用在主机 `eden`、端口5000上的服务器移除用户 `joe`，并带有验证和回显，可使用下面的命令：

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

ecpg

ecpg — 嵌入式 SQL C 预处理器

大纲

`ecpg` [`option`…] `file`…

- `-c`

自动从 SQL 代码生成确定的 C 代码。当前，这对 `EXEC SQL TYPE` 起效。

- `-C mode`

设置一个兼容性模式。`mode` 可以是 `INFORMIX`，`INFORMIX_SE` 或 `ORACLE`。

- `-D symbol`

定义一个 C 预处理器符号。

- `-h`

处理头文件。指定此选项后，输出文件扩展名变为 `.h` 而不是 `.c`，默认输入文件扩展名为 `.pgh` 而不是 `.pgc`。此外，将强制启用 `-c` 选项。

- `-i`

分析系统也包括文件。

- `-I directory`

指定一个额外的包括路径，用来寻找通过 `EXEC SQL INCLUDE` 包括的文件。默认值是 `.`（当前目录）、`/usr/local/include`、在编译时定义的IvorySQL包括目录（默认：`/usr/local/pgsql/include`）以及 `/usr/include`。

- `-o` `filename`

指定 `ecpg` 应该将它的所有输出写到给定的 `filename`。写 `-o-` 将所有输出发送到标准输出。

- `-r` `option`

选择运行时行为。`option` 可以是下列之一：`no_indicator`
不使用指示器而使用特殊值来表示空值。历史上曾有数据库使用这种方法。`prepare`
在使用所有语句之前准备它们。libecpg
将保持一个预备语句的缓冲并当语句再被执行时重用该语句。如果缓冲满了，libecpg
将释放最少使用的语句。`questionmarks`为兼容性原因允许使用问号作为占位符。在很久以前这被用作默认值。

- `-t`

打开事务的自动提交。在这种模式下，每一个 SQL
命令会被自动提交，除非它位于一个显式事务块中。在默认模式中，命令只有当 `EXEC SQL COMMIT`
被发出时才被提交。

- `-v`

打印额外信息，包括版本和“包括”路径。

- `--version`

打印ecpg版本并退出。

- `-? --help`

显示关于ecpg命令行参数的帮助并退出。

注解

在编译预处理好的 C
代码文件时，编译器需要能够找到IvorySQL包括目录中的ECPG头文件。因此，在调用编译器时，你可能必须使用 `-I` 选项（例如， `-I/usr/local/pgsql/include` ）。

使用带有嵌入式 SQL 的 C 代码的程序必须被链接到 `libecpg` 库，例如使用链接器选项
`-L/usr/local/pgsql/lib -lecpg`。

适合于安装的这些目录的值可以使用 pg_config 找到。

例子

如果你有一个名为 `prog1.pgc` 的嵌入式 SQL C 源文件，你可以使用下列命令序列创建一个可执行程序：

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_amcheck

pg_amcheck — 在一个或多个IvorySQL数据库中检查损坏

大纲

`pg_amcheck` [`option`…] [`dbname`]

选项

以下命令行选项控制要检查的内容：

- `-a --all`

检查所有数据库，但通过`--exclude-database`排除的数据库除外。

- `-d pattern --database=pattern`

检查与指定的 `pattern`匹配的数据库，但`--exclude-database`排除的数据库除外。可以多次指定此选项。

- `-D pattern --exclude-database=pattern`

排除与给定的 `pattern`匹配的数据库。可以多次指定此选项。

- `-i pattern --index=pattern`

检查与指定的 `pattern`匹配的索引，除非它们被排除在外。可以多次指定此选项。这类似于 `--relation`
选项，只是它只适用于索引，而不适用于表。

- `-I pattern --exclude-index=pattern`

排除与指定的 `pattern`匹配的索引。可以多次指定此选项。这类似于 `--exclude-relation`
选项，只是它只适用于索引，而不适用于表。

- `-r pattern --relation=pattern`

检查与指定的
pattern匹配的关系，除非它们被排除在外。可以多次指定此选项。模式可能是非限定的，例如
myrel*，或者它们可能是模式限定的，如 myschema*.myrel* 或数据库限定和模式限定，例如
mydb*.myscheam*.myrel*。数据库限定模式将向要检查的数据库列表中添加匹配的数据库。

- -R pattern --exclude-relation=pattern

排除与指定的 pattern匹配的关系。可以多次指定此选项。与`--relation`一样, pattern
可以是非限定的、模式限定的或数据库和模式限定的。

- -s pattern --schema=pattern

检查模式中与指定的
pattern匹配的表和索引，除非另有排除。可以多次指定此选项。要仅选择与特定模式匹配的模式中的表，请考虑使用类似 --table=SCHEMAPAT.* --no-dependent-indexes。要仅选择索引，请考虑使用类似
--index=SCHEMAPAT. 的方法。模式模式可以是数据库限定的。例如，你可以编写 --schema=mydb.myschema*
在数据库中选择匹配 mydb* 的模式。

- -S pattern --exclude-schema=pattern

排除模式中与指定的 pattern匹配的表和索引。可以多次指定此选项。与`--
schema`一样，模式可以是数据库限定的。

- -t pattern --table=pattern

检查与指定的 pattern匹配的表，除非它们被排除在外。可以多次指定此选项。这类似于`--
relation`选项，只是它只适用于表，而不适用于索引。

- -T pattern --exclude-table=pattern

排除与指定的 pattern 匹配的表。可以多次指定此选项。这类似于 --exclude-relation
选项，只是它只适用于表，而不适用于索引。

- --no-dependent-indexes

默认情况下，如果选中了一个表，则该表的任何btree索引也将被选中，即使它们没有被诸如 --index 或
--relation 之类的选项显式选择。此选项将抑制该行为。

- --no-dependent-toast

默认情况下，如果选中了一个表，则也将检查它的toast表（如果有），即使它没有通过诸如 --table 或
--relation 之类的选项显式选择。此选项将抑制该行为。

- --no-strict-names

默认情况下，如果 --database， --table， --index，或 --relation
的参数不匹配任何对象，则这是一个致命错误。此选项将该错误降级为警告。

以下命令行选项用于控制表的检查：

- --exclude-toast-pointers

默认情况下，每当在表中遇到toast指针时，都会执行查找以确保它引用toast表中明显有效的条目。这些检查可能非常慢，可以使用此选项跳过这些检查。

- --on-error-stop

在发现损坏的表的第一页上报告所有损坏后，停止处理该表关系，并转到下一个表或索引。请注意，索引检查总是在第一个损坏页之后停止。此选项仅与表关系相关。

- --skip=option

如果给定 all-frozen，表损坏检查将跳过所有表中标记为全部冻结的页面。如果给定了 all-

visible，则表损坏检查将跳过所有表中标记为所有可见的页面。默认情况下，不跳过任何页面。这可以指定为 none，但由于这是默认值，因此无需提及。

- --startblock=block

从指定的块号开始检查。如果正在检查的表关系的块数少于此数，则会发生错误。此选项不适用于索引，可能仅在检查单个表关系时有用。请参阅 --endblock 了解更多注意事项。

- --endblock=`block`

在指定的块号结束检查。如果正在检查的表关系的块数少于此数，则会发生错误。此选项不适用于索引，可能仅在检查单个表关系时有用。如果同时选中常规表和toast表，则此选项将同时适用于这两个表，但在验证toast指针时，仍然可以访问编号更高的toast块，除非使用 --exclude-toast-pointers 抑制。

以下命令行选项用来控制B树索引的检查：

- --heapallindexed

对于每个选中的索引，使用 amcheck's heapallindexed 选项验证索引中是否存在作为索引元组的所有堆元组。

- --parent-check

对于检查的每个btree索引，使用 amcheck 的 bt_index_parent_check 函数，该函数在索引检查期间对父/子关系执行其他检查。默认情况是使用amcheck的 bt_index_check 函数，但请注意，使用 --rootdescend 选项会隐式选择 bt_index_parent_check。

- --rootdescend

对于每个选中的索引，通过使用 amcheck's rootdescend 选项从根页面对每个元组执行新搜索，在叶级重新查找元组。使用此选项也会隐式选择 --parent-check 选项。这种形式的验证最初是为了帮助开发btree索引功能而编写的。它在帮助检测实践中发生的损坏类型方面可能用处有限，甚至毫无用处。它还可能导致损坏检查花费相当长的时间，并消耗服务器上相当多的资源。

警告

当指定 --parent-check 选项或 --rootdescend 选项时，对B树索引执行的额外检查需要相对强的关系级锁。这些检查是唯一会阻止 INSERT， UPDATE，和 DELETE 命令并发数据修改的检查。

以下命令行选项用来控制与服务器的连接：

- -h hostname --host=hostname

指定运行服务器的计算机的主机名。如果该值以斜杠开头，它将用作Unix域套接字的目录。

- -p port --port=port

指定服务器正在侦听连接的TCP端口或本地Unix域套接字文件扩展名。

- -U --username=username

连接的用户名。

- -w --no-password

永远不要发出密码提示。如果服务器需要密码身份验证，并且密码无法通过其他方式（如 .pgpass 文件，连接尝试将失败。在没有用户输入密码的批处理作业和脚本中，此选项非常有用。

- -W --password

在连接到数据库之前，强制pg_amcheck提示输入密码。此选项从来都不是必需的，因为如果服务器要求密

码身份验证，pg_amcheck将自动提示输入密码。但是，pg_amcheck将浪费一次连接尝试，以发现服务器需要密码。在某些情况下，键入 -W 以避免额外的连接尝试是值得的。

- --maintenance-db=dbname

指定用于发现要检查的数据库列表的数据库或 连接字符串。如果既不使用 --all
也不使用任何包含数据库模式的选项，则不需要这种连接，并且该选项不起任何作用。否则，连接到正在检查的数据库时，也将使用此选项值中包含的数据库名称以外的任何连接字符串参数。如果省略此选项，则默认值为 postgres，或如果失败，则为 template1。

其它允许的选项：

- -e --echo

回显发送到服务器的所有SQL。

- -j num --jobs=num

使用 num
个到服务器的并发连接，或每个要检查的对象使用一个连接，以较小者为准。默认情况是使用单个连接。

- -P --progress

显示进度信息。进度信息包括已完成检查的关系数以及这些关系的总大小。它还包括最终要检查的关系的总数，以及这些关系的估计大小。

- -v --verbose

打印更多消息。特别是，这将为每个正在检查的关系打印一条消息，并将增加服务器错误的详细程度。

- -V --version

打印pg_amcheck版本并退出。

- --install-missing --install-missing=`schema`

安装检查数据库所需的任何缺少的扩展。如果尚未安装，每个扩展的对象将安装到给定的 schema
中，或者如果未指定则安装到 pg_catalog 模式中。目前，唯一需要的扩展是 amcheck。

- -? --help

显示有关pg_amcheck命令行参数的帮助，然后退出。

pg_basebackup

pg_basebackup — 获得一个IvorySQL集簇的一个基础备份

大纲

pg_basebackup [option…]

选项

下列命令行选项控制输出的位置和格式：

- -D directory --pgdata=directory

设置目标目录以将输出写入。如果该目录不存在，pg_basebackup将创建该目录（以及所有丢失的父目录）。如果已经存在，则必须为空。当备份处于 tar 模式中，目标目录可以被指定为 - （破折号），从而将 tar 文件写入 stdout。这个选项是必需的。

- -F format --format=format

为输出选择格式。format 可以是下列之一： p plain
把输出写成平面文件，使用和源服务器数据目录和表空间相同的布局。
当集簇没有额外表空间时，整个数据库将被放在目标目录中。
如果集簇包含额外的表空间，主数据目录将被放置在目标目录中，但是所有其他表空间将被放在它们位于源服务器上的相同的绝对路径中。 (参见 --tablespace-mapping 来更改。)这是默认格式。t tar
将输出作为tar文件写入目标目录中。 主数据目录的内容将被写入名为 base.tar
的文件，而其他表空间将被写入以该表空间的OID命名的单独的tar文件。如果将目标目录指定为 -
（破折号），则tar内容将被写入标准输出，该标准输出适用于管道传输至gzip（例如）。
只有当集簇没有额外表空间并且没有使用WAL流时才允许这样做。

- -R --write-recovery-conf

创建一个 standby.signal 文件，并将连接设置附加到目标目录（或使用tar格式的基本存档文件中）的 postgresql.auto.conf 文件中。 这样可以简化使用备份结果设置后备服务器的过程。
postgresql.auto.conf
文件将记录连接设置（如果有）以及pg_basebackup所使用的复制槽，这样流复制后面就会使用相同的设置。

- -T olddir=newdir --tablespace-mapping=olddir=newdir

在备份期间将目录 olddir 中的表空间重定位到 newdir 中。为使之有效，olddir
必须与源服务器上定义的表空间的路径规范完全匹配。（但如果备份中没有包含 olddir
中的表空间也不是错误）。同时，newdir 是接收主机文件系统中的目录。与主目标目录一样，newdir
不必已经存在，但是如果确实存在，则必须为空。olddir 和 newdir
必须是绝对路径。如果一个路径凑巧包含了一个 =
符号，可用反斜线对它转义。对于多个表空间可以多次使用这个选项。如果以这种方法重定位一个表空间，主数据目录中的符号链接会被更新成指向新位置。因此新数据目录已经可以被一个所有表空间位于更新后位置的新服务器实例使用。目前，这个选项仅以普通输出格式工作，如果选择了tar格式，请忽略它。

- --waldir=waldir

指定用于预写式日志目录的位置。waldir
必须是绝对路径。只有当备份是平面文件模式时才能指定事务日志目录。
设置要写入WAL（预写日志）文件的目录。 默认情况下，WAL文件将放置在目标目录的 pg_wal
子目录中，但是此选项可用于将它们放置在其他位置。waldir 必须是绝对路径。与主目标目录一样，waldir
不必已经存在，但是如果确实存在，则必须为空。只有当备份是平面文件模式时才可以指定此选项。

- -X method --wal-method=method

在备份中包括所需的WAL（预写式日志）文件。这包括所有在备份期间产生的预写式日志。除非指定了方法 none，可以在目标目录中启动postmaster而无需参考日志归档，从而使输出成为完全独立的备份。支持下列收集预写式日志的 方法： n none 不要在备份中包括预写式日志。f fetch
在备份末尾收集预写式日志文件。因此，有必要把源服务器的 wal_keep_size
参数设置得足够高，这样在备份末尾之前需要的日志数据不会被移除。如果所需的日志数据在传输之前已被回收，则备份将失败并且无法使用。如果使用tar格式，预写式日志文件将被包含在 base.tar 文件。s stream
在进行备份时，流式传输预写式日志数据。将开启一个到服务器的第二连接并且在运行备份时并行开始流传输预写式日志。因此，它将需要两个复制连接，而不仅仅是一个。
只要客户端可以跟上预写式日志数据，使用此方法就不需要在源服务器上保存额外的预写式日志。如果使用tar格式，预写式日志文件被写入到一个单独的名为 pg_wal.tar
的文件（如果服务器的版本超过10，该文件将被命名为 pg_xlog.tar ）。这个值是默认值。

- -z --gzip

启用对 tar 文件输出的 gzip 压缩，使用默认的压缩级别。只有使用 tar
格式时压缩才可用，并且会在所有tar文件名后面自动加上后缀 .gz。

- -Z level --compress=level

启用对 tar 文件输出的 gzip 压缩，并且制定压缩机别（0 到 9，0 是不压缩，9 是最佳压缩）。只有使用 tar
格式时压缩才可用，并且会在所有tar文件名后面自动加上后缀 .gz。

下列命令行选项控制备份的生成和程序的调用：

- `-c fast|spread --checkpoint=fast|spread`

将检查点模式设置为 fast（立刻）或 spread（默认）。

- `-C --create-slot`

指定在启动备份之前应创建由 `--slot` 选项命名的复制插槽。如果插槽已存在，则会引发错误。

- `-l label --label=label`

为备份设置标签。如果没有指定，将使用一个默认值“`pg_basebackup base backup`”。

- `-n --no-clean`

默认情况下，当 `pg_basebackup` 因为一个错误而中止时，它会把它意识到无法完成该工作之前已经创建的目录（例如目标目录和预写式日志目录）都移除。这个选项可以禁止这种清洗，因此可以用于调试。注意不管哪一种方式都不会清除表空间目录。

- `-N --no-sync`

默认情况下，`pg_basebackup` 将等待所有文件被安全地写到磁盘上。这个选项导致 `pg_basebackup` 不做这种等待就返回，这样会更快一些，但是也意味着后续发生的操作系统崩溃可能会使得这个基础备份损坏。通常这个选项对测试比较有用，在创建生产安装时不应该使用。

- `-P --progress`

启用进度报告。启用这个选项将在备份期间发表一个大致的进度报告。由于数据库可能在备份期间改变，这仅仅是一种近似并且可能不会刚好在 `100%` 结束。特别地，当 WAL 日志被包括在备份中时，总数据量无法预先估计，并且在这种情况中估计的目标尺寸会在它经过不带 WAL 的总估计后增加。

- `-r rate --max-rate=rate`

设置从源服务器收集数据的最大传输速率。这有助于限制pg_basebackup对服务器的影响。值以每秒千字节为单位。使用后缀 `M` 表示每秒兆字节数。后缀 `k` 也可以接受，没有任何影响。有效值介于每秒32千字节和每秒1024兆字节之间。此选项始终影响数据目录的传输。只有收集方法为 `fetch` 时，才会影响WAL文件的传输。

- `-S slotname --slot=slotname`

这个选项仅能与 `-X stream` 一起使用。它导致WAL流使用指定的复制槽。如果该基础备份的目的是被用作一台使用复制槽的流复制后备，则它应该使用与 primary_slot_name 中相同的复制槽名称。这可以确保主服务器不会移除位于该基础备份结束与新备用数据库上流复制开始之间产生的任何所需的WAL数据。指定的复制槽必须已经存在，除非同时使用了选项 `-C`。如果这个选项没有被指定并且服务器支持临时复制槽（版本10以后），则会自动使用一个临时复制槽来进行WAL流。

- `-v --verbose`

启用冗长模式。将在启动和关闭期间输出一些额外步骤，并且如果进度报告也被启用，还会显示当前正在被处理的确切文件名。

- `--manifest-checksums=algorithm`

指定应应用于备份清单中包含的每个文件的校验和算法。目前，可用的算法有`NONE`、`CRC32C`、`SHA224`、`SHA256`、`SHA384`和`SHA512`。默认值为`CRC32C`。如果选择了`NONE`，备份清单将不包含任何校验和。否则，它将包含备份中使用指定算法的每个文件的校验和。此外，清单将始终包含自身内容的`SHA256`校验和。`SHA`算法比`CRC32C`算法占用大量CPU，因此选择其中一种算法可能会增加完成备份所需的时间。使用SHA散列函数可为希望验证备份是否遭到篡改的用户提供每个文件的加密安全摘要，而CRC32C算法提供的校验和计算速度更快。它擅长捕获由于意外更改引起的错误，但不能抵抗恶意修改。请注意，为了对有权访问备份的对手有用，备份清单必须安全地存储在其他位置，否则必须经过验证以确保自从进行备份以来未

进行任何修改。 pg_verifybackup 可用于根据备份清单检查备份的完整性。

- `--manifest-force-encode`

强制备份清单中的所有文件名采用十六进制编码。如果未指定此选项，则仅对非UTF8文件名进行十六进制编码。此选项主要用于测试读取备份清单文件的工具是否正确处理此情况。

- `--no-estimate-size`

阻止服务器估计将要流式传输的备份数据总量，从而导致 `pg_stat_progress_basebackup` 视图中的 `backup_total` 列始终为 `NULL`
。如果没有此选项，则备份将从枚举整个数据库的大小开始，然后返回并发送实际内容。这可能会使备份花费的时间稍长一些，特别是在发送第一个数据之前，备份将花费更长的时间。
如果估计时间过长，此选项将有助于避免此类估计时间。使用 `--progress` 时不允许使用此选项。

- `--no-manifest`

禁用备份清单的生成。如果未指定此选项，则服务器将生成并发送备份清单，可以使用 pg_verifybackup 进行验证。清单是备份中存在的每个文件的列表，可能包含的所有WAL文件除外。它还存储大小、上次修改时间和每个文件的可选校验和。

- `--no-slot`

防止为备份创建临时复制插槽。默认情况下，如果选择了日志流，但没有用选项 `-S`
指定槽名称，则会创建一个临时复制插槽（如果源服务器支持）。这个选项的主要目的是允许在服务器没有空闲复制槽可用时制作基础备份。使用复制槽几乎总是最好的方式，因为它能防止备份期间所需的WAL被删除。

- `--no-verify-checksums`

如果在取基础备份的服务器上启用了校验码验证，则禁用校验码验证。默认情况下，校验码会被验证并且校验码失败将会导致一种非零的退出状态。不过，基础备份在这种情况下将不会被移除，就好像使用了 `--no-clean` 选项一样。校验和验证失败也将报告在 `pg_stat_database` 视图中。

以下命令行选项控制到源服务器的连接：

- `-d connstr --dbname=connstr`

指定用于连接到服务器的参数，比如
连接字符串；这些将覆盖所有冲突的命令行选项。为了和其他客户端应用一致，该选项被称为 `--dbname`。但是因为pg_basebackup并不连接到集簇中的任何特定数据库，连接字符串中的任何数据库名将被忽略。

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。默认值取自 `PGHOST` 环境变量（如果设置），否则会尝试一个 Unix 域套接字连接。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认用 `PGPORT`
环境变量中的值（如果设置），或者一个编译在程序中的默认值。

- `-s interval --status-interval=interval`

指定发送回源服务器的状态包之间的秒数。较小的值可以更准确地监视服务器的备份进度。一个零值完全禁用这种周期性的状态更新，不过当服务器需要时还是会有一个更新会被发送来避免超时导致的断开连接。默认值是 10 秒。

- `-U username --username=username`

指定连接的用户名。

- `-w --no-password`

防止发出口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制pg_basebackup在连接到源服务器之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，pg_basebackup将自动提示要求一个口令。但是，pg_basebackup将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

其他选项也可用：

- `-V --version`

打印pg_basebackup版本并退出。

- `-? --help`

显示有关pg_basebackup命令行参数的帮助并退出。

环境

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

环境变量 `PG_COLOR` 规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

注解

在备份的开始时，需要在源服务器上执行检查点。这可能需要一点时间（尤其在没有使用选项 `--checkpoint=fast` 时），在此期间pg_basebackup看起来处于闲置状态。

备份将包括数据目录和表空间中的所有文件，包括配置文件以及由第三方放在该目录中的任何额外文件，不过由IvorySQL管理的特定临时文件除外。但只有常规文件和目录会被拷贝，但用于表空间的符号链接会被保留。指向IvorySQL已知的特定目录的符号链接被拷贝为空目录。其他符号链接和特殊设备文件会被跳过。

在普通格式中，表空间将备份到源服务器上的相同路径，除非使用了 `--tablespace-mapping` 选项。如果没有这个选项并且表空间正在使用，在同一台服务器上进行普通格式的基础备份将无法工作，因为备份必须要写入到与原始表空间相同的目录位置。

在使用 tar 格式时，用户应负责在启动使用数据的 IvorySQL 服务器前解压每一个 tar 文件。如果有额外的表空间，用于它们的 tar 文件需要被解压到正确的位置。在这种情况下，服务器将根据包含在 `base.tar` 文件中的 `tablespace_map` 文件的内容为那些表空间创建符号链接。

pg_basebackup可以和具有相同或较低主版本的服务器一起工作.

如果在源集簇上启用了组权限，pg_basebackup将保留数据文件的组权限。

例子

要创建服务器 `mydbserver` 的一个基础备份并将它存储在本地目录 `/usr/local/pgsql/data` 中：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

要创建本地服务器的一个备份，为其中每一个表空间产生一个压缩过的 tar 文件，并且将它存储在目录 `backup` 中，在运行期间显示一个进度报告：

```
$ pg_basebackup -D backup -Ft -z -P
```

要创建一个单一表空间本地数据库的备份并且使用bzip2压缩它：

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

（如果在该数据库中有多个表空间，这个命令将失败）。

要创建一个本地数据库的备份，其中 `/opt/ts` 中的表空间被重定位到 `./backup/ts`：

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

pgbench

pgbench — 在IvorySQL上运行一个基准测试

大纲

`pgbench -i` [`option`…] [`dbname`]

`pgbench` [`option`…] [`dbname`]

小心

`pgbench -i` 会创建四个表 `pgbench_accounts`、 `pgbench_branches`、 `pgbench_history` 以及 `pgbench_tellers`，如果同名表已经存在会被先删除。如果你已经有同名表，一定注意要使用另一个数据库！

在默认的情况下“比例因子”为 1，这些表初始包含的行数为：

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

你可以使用 `-s` （比例因子）选项增加行的数量。`-F` （填充因子）选项也可以在这里使用。

一旦你完成了必要的设置，你就可以用不包括 `-i` 的命令运行基准，也就是：

```
pgbench [ options ] dbname
```

在近乎所有的情况中，你将需要一些选项来做一次有用的测试。最重要的选项是 `-c` （客户端数量）、 `-t` （事务数量）、`-T` （时间限制）以及 `-f` （指定一个自定义脚本文件）。完整的列表见下文。

选项

下面分成三个部分。数据库初始化期间使用的选项和运行基准时会使用不同的选项，但也有一些选项在两种情况下都使用。

pgbench接受下列命令行初始化参数：

- dbname

指定要测试的数据库的名称。 如果这个没有被指定，将使用环境变量 PGDATABASE 。
如果那个也没有设定，将使用指定要连接的用户名称。

- -i --initialize

要求调用初始化模式。

- -I init_steps --init-steps=init_steps

只执行选出的一组普通初始化步骤。init_steps
指定要被执行的初始化步骤，每一个步骤使用一个字符代表。每一个步骤都以指定的顺序被调用。默认是 dtgvp。可用的步骤是：d（删除）删除任何已有的pgbench表。t（创建表）创建标准pgbench场景使用的表，即 pgbench_accounts、pgbench_branches、pgbench_history`以及 `pgbench_tellers。g 或 G（生成数据、客户端或服务器端）生成数据并且装入到标准的表中，替换掉已经存在的任何数据。使用 g（客户端数据生成），数据在 pgbench 客户端生成，然后发送到服务器。这通过 COPY 广泛使用客户端/服务器带宽。使用 g 会导致日志记录每 100,000 行打印一条消息，同时为 pgbench_accounts 表生成数据。使用 G（服务器端数据生成），仅从 pgbench 客户端发送少量查询，然后在服务器中实际生成数据。 此变体不需要大量带宽，但服务器将完成更多工作。使用 G`会导致日志记录在生成数据时不打印任何进度消息。默认的初始化行为使用客户端数据生成（相当于`g）。v（清理）在标准的表上调用 VACUUM。p（创建主键）在标准的表上创建主键索引。f（创建外键）在标准的表之间创建外键约束（注意这一步默认不会被执行）。

- -F fillfactor --fillfactor= fillfactor

用给定的填充因子创建表 pgbench_accounts、pgbench_tellers 以及 pgbench_branches。默认是100。

- -n --no-vacuum

在初始化期间不执行清理（这个选项会抑制 v 初始化步骤，即便在`-I` 中指定了该步骤）。

- -q --quiet

把记录切换到安静模式，只是每 5 秒产生一个进度消息。默认的记录会每 100,000 行打印一个消息，这经常会在每秒钟输出很多行（特别是在好的硬件上）。如果在 -I 中指定了 G，则此设置无效。

- -s scale_factor --scale= scale_factor

将生成的行数乘以比例因子。例如，-s 100 将在 pgbench_accounts 表中创建 10,000,000 行。默认为 1。当比例为 20,000 或更高时，用来保存账号标识符的列（aid 列）将切换到使用更大的整数（bigint），这样才能足以保存账号标识符。

- --foreign-keys

在标准的表之间创建外键约束（如果 f 在初始化步骤序列中不存在，这个选项会把它加入）。

- --index-tablespace=index_tablespace

在指定的表空间而不是默认表空间中创建索引。

- --partition-method=NAME

使用 NAME 方法创建一个分区的 pgbench_accounts 表。 预期值为 range 或 hash。 此选项要求 --partitions 设置为非零。 如果未指定，默认值为 range。

- --partitions=NUM

创建一个分区 pgbench_accounts 表，其中 NUM 分区的大小与按比例缩放的帐户数几乎相等。 默认为 0，表示没有分区。

- --tablespace=tablespace

在指定的表空间而不是默认表空间中创建表。

- --unlogged-tables

把所有的表创建为非日志记录表而不是永久表。

基准选项

pgbench接受下列命令行基准参数：

- -b scriptname[@weight] --builtin= scriptname[@weight]

把指定的内建脚本加入到要被执行的脚本列表中。 可用的内建脚本有： tpcb-like、 simple-update 和 select-only 。 这里也接受内建名称无歧义的前缀缩写。 如果用上特殊的名字 list，将会显示内建脚本的列表并且立刻退出。可选的，在 @ 后面写一个整数权重以调整选择这个脚本而不是其它的概率。 默认的权重为1。 详情如下。

- -c clients --client= clients

模拟的客户端数量，也就是并发数据库会话数量。默认为 1。

- -C --connect

为每一个事务建立一个新连接，而不是只为每个客户端会话建立一个连接。这对于度量连接开销有用。

- -d --debug

打印调试输出。

- -D varname = value --define= varname = value

定义一个由自定义脚本（见下文）使用的变量。允许多个 -D 选项。

- -f filename[@weight] --file= filename[@weight]

把一个从 filename 读到的事务脚本加入到被执行的脚本列表中。可选的，在 @ 后写一个整数权重以调整选择这个脚本而非其他的概率。 默认权重为1. (要使用包含 @ 字符的脚本文件名，附加权重在后面以避免歧义，例如 filen@me@1 。) 详情如下。

- -j threads --jobs= threads

pgbench中的工作者线程数量。在多 CPU 机器上使用多于一个线程会有用。客户端会尽可能均匀地分布到可用的线程上。默认为 1。

- -l --log

把与每一个事务相关的信息写到一个日志文件中。详见下文。

- -L limit --latency-limit= limit

对持续超过 limit 毫秒的事务进行独立的计数和报告， 这些事务被认为是 迟到（late）了的事务。在使用限流措施时（ --rate=… ），滞后于计划超过 limit 毫秒并且因此没有希望满足延迟限制的事务根本 不会被发送给服务器。这些事务被认为是被跳过（skipped） 的事务，它们会被单独计数并且报告。

- -M querymode --protocol=querymode

要用来提交查询到服务器的协议： simple ：使用简单查询协议。 extended 使用扩展查询协议。 prepared ：使用带预备语句的扩展查询语句。在 prepared
模式中，pgbench重用从第二次查询迭代开始的语法分析结果，因此pgbench运行速度比其他模式快。默认是简单查询协议。

- -n --no-vacuum

在运行测试前不进行清理。如果你在运行一个不包括标准的表 pgbench_accounts 、 pgbench_branches 、 pgbench_history 和 pgbench_tellers 的自定义测试场景时，这个选项是必需的。

- -N --skip-some-updates

运行内建的简单更新脚本。这是 -b simple-update 的简写。

- -P sec --progress=sec

每 sec 秒显示进度报告。该报告包括运行了多长时间、从上次报告以来的 tps
以及从上次报告以来事务延迟的平均值和标准偏差。如果低于限流值（ -R
），延迟会相对于事务预定的开始时间（而不是实际的事务开始时间）计算，因此其中也包括了平均调度延迟时间。

- -r --report-latencies

在基准结束后，报告平均的每个命令的每语句等待时间（从客户端的角度来说是执行时间）。详见下文。

- -R rate --rate= rate

按照指定的速率执行事务而不是尽可能快地执行（默认行为）。该速率 以
tps（每秒事务数）形式给定。如果目标速率高于最大可能速率，则
该速率限制不会影响结果。该速率的目标是按照一条泊松分布的调度时间线开始事务。期望的开始
时间表会基于客户端第一次开始的时间（而不是上一个事务结束的时
间）前移。这种方法意味着当事务超过它们的原定结束时间时，更迟的
那些有机会再次追赶上来。当限流措施被激活时，运行结束时报告的事务延迟是从预订的开始时间计
算而来的，因此它包括每一个事务不得不等待前一个事务结束所花的时
间。该等待时间被称作调度延迟时间，并且它的平均值和最大值也会被
单独报告。关于实际事务开始时间的事务延迟（即在数据库中执行事务
所花的时间）可以用报告的延迟减去调度延迟时间计算得到。如果把 --latency-limit 和 --rate 一起使用，
当一个事务在前一个事务结束时已经超过了延迟限制时，它可能会滞后
非常多，因为延迟是从计划的开始时间计算得来。这类事务不会被发送
给服务器，而是一起被跳过并且被单独计数。一个高的调度延迟时间表示系统无法用选定的客户端和线程数
按照指定
的速率处理事务。当平均的事务执行时间超过每个事务之间的调度间隔时，每一个后续事务将会落后更多，
并且随着测试运行时间越长，调度延迟时间将持续增加。发生这种情况时，你将不得不降低指定的事务速率
。

- -s scale_factor --scale= scale_factor

在pgbench的输出中报告指定的比例因子。对于内建测试，这并非必需；正确的比例因子将通过对 pgbench_branches 表中的行计数来检测。不过，当只测试自定义基准（ -f 选项）时，比例因子将被报告为 1（除非使用了这个选项）。

- -S --select-only

执行内建的只有选择的脚本。是 -b select-only 简写形式。

- -t transactions --transactions= transactions

每个客户端运行的事务数量。默认为 10。

- -T seconds --time= seconds

运行测试这么多秒，而不是为每个客户端运行固定数量的事务。-t 和 -T 是互斥的。

- `-v --vacuum-all`

在运行测试前清理所有四个标准的表。在没有用 `-n` 以及 `-v` 时， pgbench将清理 `pgbench_tellers` 和 `pgbench_branches` 表，并且截断 `pgbench_history`。

- `--aggregate-interval=`seconds`

聚集区间的长度（单位是秒）。仅可以与 `-l` 选项一起使用。通过这个选项，日志会包含针对每个区间的概要数据，如下文所述。

- `--log-prefix=prefix`

设置 `--log` 创建的日志文件的文件名前缀。默认是 `pgbench_log`。

- `--progress-timestamp`

当显示进度（选项 `-P` ）时，使用一个时间戳（Unix 时间）取代从运行开始的秒数。单位是秒，在小数点后是毫秒精度。这可以有助于比较多种工具生成的日志。

- `--random-seed= seed`

设置随机数生成器种子。为系统的随机数生成器提供种子，然后随机数生成器会产生一个初始生成器状态序列，每一个线程一个状态。`seed` 的值可以是： `time` （默认值，种子基于当前时间）、 `rand` （使用一种强随机源，如果没有可用的源则失败）或者一个无符号十进制整数值。一个pgbench脚本中会显式（ `random…` 函数）地或者隐式地（如 `--rate` 使用随机数生成器调度事务）调用随机数生成器。在被明确设置时，用作种子的值会显示在终端上。还可以通过环境变量 `PGBENCH_RANDOM_SEED` 提供用于 `seed` 的值。为了确保所提供的种子影响所有可能的使用，把这个选项放在第一位或者使用环境变量。明确地设置种子允许准确地再生一个 `pgbench` 运行，对随机数而言。因为随机状态是针对每个线程管理，这意味着如果每一个线程有一个客户端并且没有外部或者数据依赖，则对于一个相同的调用就会有完全相同的 `pgbench` 运行。从一种统计的角度来看，再生运行不是什么好主意，因为它能隐藏性能可变性或者不正当地改进性能，即通过命中前一次运行的相同页面来改进性能。不过，它也可以对调试起到很大帮助作用，例如重新运行一种导致错误的棘手用例。请善用。

- `--sampling-rate=`rate`

采样率，在写入数据到日志时被用来减少日志产生的数量。如果给出这个选项，只有指定比例的事务被记录。1.0 表示所有事务都将被记录，0.05 表示只有 5% 的事务会被记录。在处理日志文件时，记得要考虑这个采样率。例如，当计算TPS值时，你需要相应地乘以这个数字（例如，采样率是 0.01，你将只能得到实际TPS的 1/100）。

- `--show-script= scriptname`

在 stderr 上显示内置脚本 `scriptname` 的实际代码，并立即退出。

普通选项

pgbench 还接受以下用于连接参数的常见命令行参数：

- `-h hostname --host= hostname`

数据库服务器的主机名

- `-p port --port=port`

数据库服务器的端口号

- `-U login --username= login`

要作为哪个用户连接

- `-V --version`

打印pgbench版本并退出。

- `-? --help`

显示有关pgbench命令行参数的信息，并且退出。

退出状态

成功运行将以状态0退出。 退出状态为1表示静态问题，如无效的命令行选项。
运行过程中的错误，例如数据库错误或脚本中的问题将导致退出状态为2。
在后一种情况下，pgbench将打印部分结果。

环境

- `PGDATABASE PGHOST PGPORT PGUSER`

默认连接参数。

此实用程序与大多数其他 IvorySQL 实用程序一样，使用 libpq 支持的环境变量。

环境变量 `PG_COLOR` 指定是否在诊断消息中使用颜色。 可能的值是 `always`、 `auto` 和 `never`.

pg_config

pg_config — 获取已安装的IvorySQL的信息

大纲

`pg_config` [`option`…]

选项

要使用pg_config，提供一个或多个下列选项：

- `--bindir`

打印用户可执行文件的位置。例如使用这个选项来寻找 `psql` 程序。这通常也是 `pg_config` 程序所在的位置。

- `--docdir`

打印文档文件的位置。

- `--htmldir`

打印 HTML 文档文件的位置。

- `--includedir`

打印客户端接口的 C 头文件的位置。

- `--pkgincludedir`

打印其它 C 头文件的位置。

- `--includedir-server`

打印用于服务器编程的 C 头文件的位置。

- `--libdir`

打印对象代码库的位置。

- `--pkglibdir`

打印动态可载入模块的位置，或者服务器可能搜索它们的位置（其它架构独立数据文件可能也被安装在这个目录）。

- `--localedir`

打印区域支持文件的位置（如果在IvorySQL被编译时没有配置区域支持，这将是一个空字符串）。

- `--mandir`

打印手册页的位置。

- `--sharedir`

打印架构独立支持文件的位置。

- `--sysconfdir`

打印系统范围配置文件的位置。

- `--pgxs`

打印扩展 makefile 的位置。

- `--configure`

打印当IvorySQL被配置编译时给予 `configure`
脚本的选项。这可以被用来重新得到相同的配置，或者找出是哪个选项编译了一个二进制包（不过注意二进制包通常包含厂商相关的自定补丁）。参见下面的例子。

- `--cc`

打印用来编译IvorySQL的 `CC` 变量值。这显示被使用的 C 编译器。

- `--cppflags`

打印用来编译IvorySQL的 `CPPFLAGS` 变量值。这显示在预处理时需要的 C 编译器开关（典型的是 `-I` 开关）。

- `--cflags`

打印用来编译IvorySQL的 `CFLAGS` 变量值。这显示被使用的 C 编译器开关。

- `--cflags_sl`

打印用来编译IvorySQL的 `CFLAGS_SL` 变量值。这显示被用来编译共享库的额外 C 编译器开关。

- `--ldflags`

打印用来编译IvorySQL的 `LDFLAGS` 变量值。这显示链接器开关。

- `--ldflags_ex`

打印用来编译IvorySQL的 `LDFLAGS_EX` 变量值。这只显示被用来编译可执行程序的链接器开关。

- `--ldflags_sl`

打印用来编译IvorySQL的 `LDFLAGS_SL` 变量值。这只显示被用来编译共享库的链接器开关。

- `--libs`

打印用来编译IvorySQL的 LIBS 变量值。这通常包含用于链接到IvorySQL中的外部库的 -l 开关。

- --version

打印IvorySQL的版本。

- -? --help

显示有关pg_config命令行参数的帮助并退出。

如果给定多于一个选项，将按照相同的顺序打印信息，每行一项。如果没有给定选项，将打印所有可用信息，并带有标签。

例子

要重建当前 IvorySQL 安装的编译配置，可运行下列命令：

```
eval ./configure `pg_config --configure`
```

pg_config --configure 的输出包含 shell 引号，这样带空格的参数可以被正确地表示。因此，为了得到正确的结果需要使用 eval。

pg_dump

pg_dump — 把IvorySQL数据库抽取为一个脚本文件或其他归档文件

大纲

pg_dump [connection-option…] [option…] [dbname]

选项

下列命令选项控制输出的内容和格式。

- dbname

指定要被转储的数据库名。如果没有指定，将使用环境变量 PGDATABASE。如果环境变量也没有设置，则使用指定给该连接的用户名。

- -a --data-only

只转储数据，而不转储模式（数据定义）。表数据、大对象和序列值都会被转储。这个选项类似于指定 --section=data，但是由于历史原因又不完全相同。

- -b --blobs

在转储中包括大对象。这是当 --schema、 --table 或 --schema-only 被指定时的默认行为。因此，只有在请求转储一个特定方案或者表的情况中， -b 开关才对向转储中加入大对象有用。注意blobs是被考虑的数据，因此在使用 --data-only 时将被包括在内，但在使用 --schema-only 时则不会包括。

- -B --no-blobs

在转储中排除大对象。当同时给定 -b 和 -B 时，行为是在数据被转储时输出大对象，请参考 -b 文档。

- -c --clean

在输出创建数据库对象的命令之前输出清除（删除）它们的命令 （除非也指定了 --if-exists，如果任何对象不存在于目的数据库中，恢复可能会产生一些伤害性的错误消息）。当发出归档(非文本)输出文件时，这个选项被忽

略。对于归档格式，你可以在调用 pg_restore 时指定该选项。

- -C --create

使得在输出的开始是一个创建数据库本身并且重新连接到被创建的数据库的命令（通过这种形式的一个脚本，在运行脚本之前你连接的是目标安装中的哪个数据库都没有关系）。如果也指定了 --clean，脚本会在重新连接到目标数据库之前先删除它然后再重建。通过 --create，输出还会包括数据库的注释（如果有）以及与这个数据库相关的任何配置变量设置，也就是任何提到了这个数据库的 ALTER DATABASE … SET … 命令和 ALTER ROLE … IN DATABASE … SET … 命令。该数据库本身的访问特权也会被转储，除非指定有 --no-acl。当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用 pg_restore 时指定该选项。

- -e pattern --extension=pattern

只转储匹配 pattern 的扩展。 如果未指定此选项，目标数据库中的所有非系统扩展都将被转储。通过写入多个 -e 开关，可以选择多个扩展。 pattern 参数可以理解为与psql’s \d 命令使用的规则相同的模式， 所以还可以通过在模式中写入通配符来选择多个扩展。在使用通配符时，如果需要防止来自扩展通配符的shell，请小心的引用模式。任何由 pg_extension_config_dump 注册的配置关系被包括在转储中，如果他的扩展已经被 --extension 指定了。注意当指定了 -e ，pg_dump不尝试转储任何所选扩展可能依赖的其他数据库对象。也就是说，这里不保证指定扩展转储的结果能够由他们自己成功恢复到一个干净的数据库中。

- -E encoding --encoding=encoding

以指定的字符集编码创建转储。在默认情况下，该转储会以该数据库的编码创建（另一种得到相同结果的方式是将 PGCLIENTENCODING 环境变量设置成想要的转储编码）。

- -f file --file=file

将输出发送到指定文件。对于基于输出格式的文件这个参数可以被忽略，在那种情况下将使用标准输出。不过对于目录输出格式必须给定这个参数，在目录输出格式中指定的是一个目录而不是一个文件。在这种情况中，该目录会由 pg_dump 创建并且不需要以前就存在。

- -F format --format=format

选择输出的格式。format 可以是下列之一： p plain 输出一个纯文本形式的SQL脚本文件（默认值）。c custom 输出一个适合于作为pg_restore输入的自定义格式归档。和目录输出格式一起，这是最灵活的输出格式，它允许在恢复时手动选择和排序已归档的项。这种格式在默认情况还会被压缩。 d directory 输出一个适合作为pg_restore输入的目录格式归档。这将创建一个目录，其中每个被转储的表和大对象都有一个文件，外加一个所谓的目录文件，该文件以一种pg_restore能读取的机器可读格式描述被转储的对象。一个目录格式归档能用标准 Unix 工具操纵，例如一个未压缩归档中的文件可以使用gzip工具压缩。这种格式默认情况下是被压缩的并且也支持并行转储。t tar 输出一个适合于输入到pg_restore中的 tar -格式归档。tar 格式可以兼容目录格式，抽取一个 tar 格式的归档会产生一个合法的目录格式归档。不过，tar 格式不支持压缩。还有，在使用 tar 格式时，表数据项的相对顺序不能在恢复过程中被更改。

- -j njobs --jobs=njobs

通过同时归档 njobs 个表来运行并行转储。这个选项可能会减少执行转储所需的时间，但也会增加数据库服务器上的负载。你只能和目录输出格式一起使用这个选项，因为这是唯一一种让多个进程能在同一时间写其数据的输出格式。pg_dump将打开 njobs + 1 个到该数据库的连接，因此确保你的 max_connections 设置足够高以容纳所有的连接。在运行一次并行转储时请求数据库对象上的排他锁可能导致转储失败。其原因是，pg_dump领导者进程会在工作者进程将要稍后转储的对象上请求共享锁，以便确保在转储运行时不会有人删除它们并让它们出错。如果另一个客户端接着请求一个表上的排他锁，那个锁将不会被授予但是会被排入队列等待领导者进程的共享锁被释放。 因此，任何其他对该表的访问将不会被授予或者将排在排他锁请求之后。这包括尝试转储该表的工作者进程。如果没有任何防范措施，这可能会是一种经典的死锁情况。要检测这种冲突，pg_dump工作者进程使用 NOWAIT 选项请求另一个共享锁。

如果该工作者进程没有被授予这个共享锁，其他某人必定已经在同时请求了一个排他锁并且没有办法继续转储，因此pg_dump除了中止转储之外别无选择。对于一个一致的备份，数据库服务器需要支持同步的快照，在IvorySQL的主服务器和10的后备服务器中引入了一种特性。
有了这种特性，即便数据库客户端使用不同的连接，也可以保证他们看到相同的数据集。 `pg_dump -j` 使用多个数据库连接，它用领导者进程连接到数据一次，并且为每一个工作者任务再一次连接数据库。
如果没有同步快照特征，在每一个连接中不同的工作者任务将不能被保证看到相同的数据，这可能导致一个不一致的备份。

- `-n pattern --schema=pattern`

只转储匹配 `pattern` 的模式，这会选择模式本身以及它所包含的所有对象。当没有指定这个选项时，目标数据库中所有非系统模式都将被转储。多个模式可以通过书写多个 `-n` 开关来选择。另外， `pattern` 参数可以被解释为一种根据psql的`\d`命令所用的相同规则（请参见下面的Patterns）编写的模式，这样多个模式也可以通过在该模式中书写通配字符来选择。在使用通配符时，如果需要阻止 shell 展开通配符需要小心引用该模式，注意当 `-n` 被指定时，pg_dump不会尝试转储所选模式可能依赖的任何其他数据库对象。因此，无法保证一次指定模式转储的结果能够仅凭其本身被成功地恢复到一个干净的数据库中。注意当 `-n` 被指定时，非模式对象（如二进制大对象）不会被转储。你可以使用 `--blobs` 开关将二进制大对象加回到该转储中。

- `-N pattern --exclude-schema=pattern`

不转储匹配 `pattern` 模式的任何模式。该模式被根据 `-n` 所用的相同规则被解释。`-N` 可以被给定多次来排除匹配几个模式中任意一个的模式。当 `-n` 和 `-N` 都被给定时，该行为是只转储匹配至少一个 `-n` 开关但是不匹配 `-N` 开关的模式。如果只有 `-N` 而没有 `-n`，那么匹配 `-N` 的模式会被从一个正常转储中排除。

- `-O --no-owner`

不输出设置对象拥有关系来匹配原始数据库的命令。默认情况下，pg_dump会发出`ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句来设置被创建的数据库对象的拥有关系。除非该脚本被一个超级用户（或是拥有脚本中所有对象的同一个用户）启动，这些语句都将会失败。要使一个脚本能够被任意用户恢复，但把所有对象的拥有关系都给这个用户，可指定 `-O`。当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用`pg_restore`时指定该选项。

- `-R --no-reconnect`

这个选项已经废弃，但是为了向后兼容仍然能被接受。

- `-s --schema-only`

只转储对象定义（模式），而非数据。这个选项是 `--data-only` 的逆选项。它和指定 `--section=pre-data` `--section=post-data` 相似，但是由于历史原因又不完全相同。（不要把这个选项和 `--schema` 选项混淆，后者在“schema”的使用上有不同的含义）。要为数据库中表的一个子集排除表数据，见 `--exclude-table-data`。

- `-S username --superuser=username`

指定要在禁用触发器时使用的超级用户的用户名。只有使用 `--disable-triggers` 时，这个选项才相关（通常，最好省去这个选项，而作为超级用户来启动结果脚本来取而代之）。

- `-t pattern --table=pattern`

只转储名字匹配 `pattern` 的表。通过写多个 `-t` 开关可以选择多个表。另外，`pattern` 参数可以被解释为一种根据psql’s `\d` 命令所用的相同规则编写的模式，这样多个表也可以通过在该模式中书写通配字符来选择。在使用通配符时，如果需要阻止 shell 展开通配符需要小心引用该模式，当 `-t` 被使用时，`-n` 和 `-N` 开关不会有效果，因为被 `-t` 选择的表将被转储而无视那些开关，并且非表对象将不会被转储。注意当 `-t` 被指定时，pg_dump不会尝试转储所选表可能依赖的任何其他数据库对象。因此，无法保证一次指定表转储的结果能够仅凭其本身被成功地恢复到一个干净的数据库中。

- `-T pattern --exclude-table=pattern`

不转储匹配 `pattern` 模式的任何表。该模式被根据 `-t` 所用的相同规则被解释。`-T` 可以被给定多次来排除匹配几个模式中任意一个的模式。当 `-t` 和 `-T` 都被给定时，该行为是只转储匹配至少一个 `-t` 开关但是不匹配 `-T` 开关的表。如果只有 `-T` 而没有 `-t`，那么匹配 `-T` 的表会被从一个正常转储中排除。

- `-v --verbose`

指定冗长模式。这将导致pg_dump向标准错误输出详细的对象注释以及转储文件的开始/停止时间，还有进度消息。 重复该选项会导致在标准错误上出现附加的调试级消息。

- `-V --version`

pg_dump版本并退出。

- `-x --no-privileges --no-acl`

防止转储访问特权（授予/收回命令）。

- `-Z 0..9 --compress=0..9`

指定要使用的压缩级别。零意味着不压缩。对于自定义和目录归档格式，这会指定个体表数据段的压缩，并且默认是进行中等级别的压缩。对于纯文本输出，设置一个非零压缩级别会导致整个输出文件被压缩，就好像它被gzip处理过一样，但是默认是不压缩。tar 归档格式当前完全不支持压缩。

- `--binary-upgrade`

这个选项用于就地升级功能。我们不推荐也不支持把它用于其他目的。这个选项在未来的发行中可能被改变而不做通知。

- `--column-inserts --attribute-inserts`

将数据转储为带有显式列名的 `INSERT` 命令（`INSERT INTO table (column, …) VALUES …`）。这将使得恢复过程非常慢，这主要用于使转储能够被载入到非IvorySQL数据库中。重新加载期间的任何错误都将导致有问题的 `INSERT` 相关的行将丢失，而不是整个表内容。

- `--disable-dollar-quoting`

这个选项禁止在函数体中使用美元符号引用，并且强制它们使用 SQL 标准字符串语法被引用。

- `--disable-triggers`

只有在创建一个只转储数据的转储时，这个选项才相关。它指示pg_dump包括在数据被重新载入时能够临时禁用目标表上的触发器的命令。如果你在表上有引用完整性检查或其他触发器，并且你在数据重新载入期间不想调用它们，请使用这个选项。当前，为 `--disable-triggers` 发出的命令必须作为超级用户来执行。因此，你还应当使用 `-S` 指定一个超级用户名，或者宁可作为一个超级用户启动结果脚本。当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用 `pg_restore` 时指定该选项。

- `--enable-row-security`

只有在转储具有行安全性的表的内容时，这个选项才相关。默认情况下， pg_dump将把 row_security 设置为 off 来确保从该表中转储 出所有的数据。如果用户不具有足够能绕过行安全性的特权，那么会抛出一个错误这个参数指示pg_dump将 row_security 设置为 on，允许用户只转储该表中它们能够访问到的部分内容。注意如果当前你使用了这个选项，你可能还想得到 `INSERT` 格式的转储，因为恢复期间的 `COPY FROM` 不支持行安全性。

- `--exclude-table-data=pattern`

不转储匹配 `pattern` 模式的任何表中的数据。该模式根据 `-t` 的相同规则被解释。`--exclude-table-data` 可以被给定多次来排除匹配多个模式的表。当你需要一个特定表的定义但不想要其中的数据时，这个选项就有用了。要排除数据库中所有表的数据，见 `--schema-only`。

- `--extra-float-digits=ndigits`

在转储浮点数据时使用规定的 `extra_float_digits` 值，而不是最大可用精度。以备份目的生成的常规转储不使用此选项。

- `--if-exists`

时间条件性命令（即增加一个 `IF EXISTS` 子句）来清除数据库和其他对象。 只有同时指定了 `--clean` 时，这个选项才可用。

- `--include-foreign-data=`foreignserver`

使用与 `foreignserver` 模式匹配的外部服务器转储任何外部表的数据。可以通过编写 多个 `--include-foreign-data` 开关来选择多个外部服务器。 同样，根据psql’ s `\d` 命令使用的相同规则， 将 `foreignserver` 参数解释为模式。
因此也可以通过在模式中写入通配符来选择多个外部服务器。使用通配符时，
如果需要，请小心引用该模式，以防止Shell扩展通配符。 唯一的例外是不允许使用空模式。注意指定 `--include-foreign-data` 时，pg_dump不会检查外部表是否可写。因此，不能保证可以成功还原外部表转储的结果。

- `--inserts`

将数据转储为 `INSERT` 命令（而不是 `COPY` ）。这将使得恢复非常慢，这主要用于使转储能够被载入到非IvorySQL数据库中。重新加载期间的任何错误都将导致有问题的 `INSERT` 相关的行将丢失，而不是整个表内容。注意如果你已经重新安排了列序，该恢复可能会一起失败。`--column-inserts` 选项对于列序改变是安全的，但是会更慢。

- `--load-via-partition-root`

在为一个分区表转储数据时，让 `COPY` 语句或者 `INSERT` 语句把包含它的分区层次的根而不是分区自身作为目标。这导致在数据被装载时，会为每一个行重新确定合适的分区。如果在一台服务器上重新装载数据时会出现行并不是总是落入到和原始服务器上相同的分区中的情况，这个选项就很有用。例如，如果分区列是文本类型并且两个系统中用于排序分区列的排序规则有着不同的定义，就会发生这种情况。在从用这个选项制作的归档恢复时，最好不要使用并行，因为pg_restore将不能准确地知道一个给定的归档数据项将把数据装载到哪个分区中。这会导致效率不高，因为在并行任务见会有锁冲突，或者甚至可能由于在所有的相关数据被装载前建立了外键约束而导致重新装载失败。

- `--lock-wait-timeout=`timeout`

在转储的开始从不等待共享表锁的获得。而是在指定的 `timeout` 内不能锁定一个表时失败。超时时长可以用 `SET statement_timeout` 接受的任何格式指定。

- `--no-comments`

不转储注释。

- `--no-publications`

不转储publication。

- `--no-security-labels`

不转储安全标签。

- `--no-subscriptions`

不转储订阅。

- `--no-sync`

默认情况下，`pg_dump` 将等待所有文件被安全地写入磁盘。这个选项会让 `pg_dump` 不等待直接返回，这样会更快，但是也意味着后续的一次操作系统崩溃会让该转储损坏。通常这个选项对测

试有用，但是不应该在从生产安装中转储数据时使用。

- `--no-tablespaces`

不要输出选择表空间的命令。通过这个选项，在恢复期间所有的对象都会被创建在任何作为默认的表空间中。当发出归档(非文本)输出文件时，这个选项被忽略。对于归档格式，你可以在调用 `pg_restore` 时指定该选项。

- `--no-toast-compression`

不要输出命令以设置 TOAST 压缩方法。 采用这个选项，所有列将按默认压缩设置进行恢复。

- `--no-unlogged-table-data`

不转储非日志记录表的内容。这个选项对于表定义（模式）是否被转储没有影响，它只会限制转储表数据。当从一个后备服务器转储时，在非日志记录表中的数据总是会被排除。

- `--on-conflict-do-nothing`

增加 `ON CONFLICT DO NOTHING` 到 `INSERT` commands。 除非规定了 `--inserts`, `--column-inserts` 或 `--rows-per-insert`，否则此选项是无效的.

- `--quote-all-identifiers`

强制引用所有标识符。当从IvorySQL主版本与pg_dump不同的服务器上转储一个数据库时或者当输出准备载入到一个具有不同主版本的服务器时，推荐使用这个选项。默认情况下，pg_dump只对在其主版本中是被保留词的标识符加上引号。在转储其他版本服务器时，这种默认行为有时会导致兼容性问题，因为那些版本可能具有些许不同的被保留词集合。使用 `--quote-all-identifiers` 能阻止这种问题，但代价是转储脚本更难阅读。

- `--rows-per-insert=nrows`

数据转储为 `INSERT` 命令（而不是 `COPY`）。 控制每个 `INSERT` 命令的最大行数。指定的值必须大于零。重新加载期间的任何错误都将导致有问题的`INSERT`相关的行将丢失，而不是整个表内容。

- `--section=sectionname`

只转储命名节。节的名称可以是 `pre-data`、 `data` 或 `post-data`。这个选项可以被指定多次来选择多个节。默认是转储所有节。数据节包含真正的表数据、大对象内容和序列值。数据后项包括索引、触发器、规则和除了已验证检查约束之外的约束的定义。数据前项包括所有其他数据定义项。

- `--serializable-deferrable`

为转储使用一个 **可序列化** 事务，以保证所使用的快照与后来的数据库状态是一致的。但是这样做是在事务流中等待一个点，在该点上不能存在异常，这样就不会有转储失败或者导致其他事务带着 `serialization_failure` 回滚的风险。对于一个只为灾难恢复存在的转储，这个选项没什么益处。如果一个转储被用来在原始数据库持续被更新期间载入一份用于报表或其他只读负载的数据库拷贝时，这个选项就有所帮助。如果没有这个选项，转储可能会反映一个与最终提交事务的任何执行序列都不一致的状态。例如，如果使用了批处理技术，一个批处理在转储中可以显示为关闭，而其中的所有项都不出现。如果 pg_dump 被启动时没有读写事务在活动，则这个选项没有什么不同。如果有读写事务在活动，该转储的启动可能会被延迟一段不确定的时间。一旦开始运行，有没有这个开关的表现是相同的。

- `--snapshot=snapshotname`

在做一个数据库的转储时指定一个同步的快照。在需要把转储和一个逻辑复制槽或者一个并发会话同步时可以用上这个选项。在并行转储的情况下，将使用这个选项指定的快照名而不是取一个新快照。

- `--strict-names`

要求每一个扩展(-e / --extension)，模式（-n / --schema）和表（-t / --table）限定符匹配要转储的数据库中至少一个扩展/模式/表。
注意，如果没有找到有这样的扩展/模式/表限定符匹配，即便没有 --strict-names，pg_dump也将生成一个错误。这个选项对 -N / --exclude-schema、-T / --exclude-table 或者 --exclude-table-data 没有效果。无法匹配任何对象的排除模式不会被当作错误。

- --use-set-session-authorization

输出 SQL-标准的 SET SESSION AUTHORIZATION 命令取代 ALTER OWNER 命令来确定对象的所有关系。这让该转储更加兼容标准，但是取决于该转储中对象的历史，该转储可能无法正常恢复。而且，一个使用 SET SESSION AUTHORIZATION 的转储将一定会要求超级用户特权来正确地恢复，而 ALTER OWNER 要求更少的特权。

- -? --help

显示有关pg_dump命令行参数的帮助并退出。

下列命令行选项控制数据库连接参数。

- -d dbname --dbname=dbname

指定要连接的数据库的名称。 这等效于在命令行上将 dbname 指定为第一个非选项参数。dbname 可以是连接字符串。 如果是这样，连接字符串参数将覆盖所有冲突的命令行选项。

- -h host --host=host

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从 PGHOST 环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

- -p port --port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在`PGPORT`环境变量中（如果被设置），否则使用编译在程序中的默认值。

- -U username --username=username

要作为哪个用户连接。

- -w --no-password

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 .pgpass 文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- -W --password

强制pg_dump在连接到一个数据库之前提示要求一个口令。这个选项从来不是必须的，因为如果服务器要求口令认证，pg_dump将自动提示要求一个口令。但是，pg_dump将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入`-W`来避免额外的连接尝试。

- --role=rolename

指定一个用来创建该转储的角色名。这个选项导致pg_dump在连接到数据库后发出一个 SET ROLE rolename 命令。当已认证用户（由 -U 指定）缺少pg_dump所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

环境

- PGDATABASE PGHOST PGOPTIONS PGPORT PGUSER

默认连接参数

- PG_COLOR

规定在诊断消息中是否使用颜色。可能的值为 always、 auto、 never。.

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

pg_dump在内部执行 SELECT
语句。如果你运行pg_dump时出现问题，确定你能够从正在使用的数据库中选择信息，例如
psql。此外，libpq前端-后端库所使用的任何默认连接设置和环境变量都将适用。

pg_dump的数据库活动会被统计收集器正常地收集。如果不想这样，你可以通过 PGOPTIONS 或 ALTER USER
命令设置参数 track_counts 为假。

注解

如果你的数据库集簇对于 template1
数据库有任何本地添加，要注意将pg_dump的输出恢复到一个真正的空数据库。否则你很可能由于以增加对象的重复定义而得到错误。要创建一个不带任何本地添加的空数据库，从 template0 而不是 template1
复制它，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

当一个只含数据的转储被选中并且使用了选项 --disable-triggers
时，pg_dump在开始插入数据之前会发出命令禁用用户表上的触发器，并且接着在数据被插入之后发出命令重新启用它们。如果恢复中途被停止，系统目录可能会停留在一种错误状态。

pg_dump产生的转储文件不包含优化器用来做出查询计划决定的统计信息。因此，在从一个转储文件恢复后运行 ANALYZE 来确保最优性能是明智的。

在转储逻辑复制订阅时，pg_dump将生成使用 connect = false 选项的 CREATE SUBSCRIPTION
命令，这样恢复订阅时不会建立远程连接来创建复制槽或者进行初始的表拷贝。通过这种方式，可以无需到远程服务器的网络访问就能恢复该转储。然后就需要用户以一种合适的方式重新激活订阅。如果涉及到的主机已经改变，连接信息可能也必须被改变。在开启一次新的全表拷贝之前，截断目标表也可能是合适的。

实例

要把一个数据库 mydb 转储到一个 SQL 脚本文件：

```
$ pg_dump mydb > db.sql
```

要把这样一个脚本重新载入到一个（新创建的）名为 newdb 的数据库中：

```
$ psql -d newdb -f db.sql
```

要转储一个数据库到一个自定义格式归档文件：

```
$ pg_dump -Fc mydb > db.dump
```

要转储一个数据库到一个目录格式的归档：

```
$ pg_dump -Fd mydb -f dumpdir
```

要用 5 个并行的工作者任务转储一个数据库到一个目录格式的归档：

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

要把一个归档文件重新载入到一个（新创建的）名为 newdb 的数据库：

```
$ pg_restore -d newdb db.dump
```

把一个归档文件重新装载到同一个数据库（该归档正是从这个数据库中转储得来）中，丢掉那个数据库中的当前内容：

```
$ pg_restore -d postgres --clean --create db.dump
```

要转储一个名为 mytab 的表：

```
$ pg_dump -t mytab mydb > db.sql
```

要转储 detroit 模式中名称以 emp 开始的所有表，排除名为 employee_log 的表：

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

要转储名称以 east 或者 west 开始并且以 gsm 结束的所有模式，排除名称包含词`test`的任何模式：

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

同样，用正则表达式记号法来合并开关：

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

要转储除了名称以 ts_ 开头的表之外的所有数据库对象：

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

要在 -t
和相关开关中指定一个大写形式或混合大小写形式的名称，你需要双引用该名称，否则它会被折叠到小写形式。但是双引号对于 shell 是特殊的，所以反过来它们必须被引用。因此，
要转储一个有混合大小写名称的表，你需要类似这样的东西：

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

pg_dumpall

pg_dumpall — 将一个IvorySQL数据库集簇抽取到一个脚本文件中

大纲

pg_dumpall [**connection-option**…] [**option**…]

描述

pg_dumpall工具可以一个集簇中所有的IvorySQL数据库写出到（“转储”）一个脚本文件。该脚本文件包含可以用作 psql 的输入SQL命令来恢复数据库。它会对集簇中的每个数据库调用 pg_dump 来完成该工作。pg_dumpall还转储对所有数据库公用的全局对象（pg_dump 不保存这些对象），也就是说数据库角色和表空间都会被转储。
目前这包括适数据库用户和组、表空间以及适合所有数据库的访问权限等属性。

因为pg_dumpall从所有数据库中读取表，所以你很可能需要以一个数据库超级用户的身份连接以便生成完整的转储。同样，你也需要超级用户特权执行保存下来的脚本，这样才能增加角色和组以及创建数据库。

SQL 脚本将被写出到标准输出。使用 **-f** / **--file** 选项或者 shell 操作符可以把它重定向到一个文件。

pg_dumpall需要多次连接到IvorySQL服务器（每个数据库一次）。如果你使用口令认证，可能每次都会要求口令。这种情况下使用一个 **~/.pgpass** 会比较方便。

选项

下列命令行选项用于控制输出的内容和格式。

- **-a** **--data-only**

只转储数据，不转储模式（数据定义）。

- **-c** **--clean**

包括在重建数据库之前清除（移除）它们的 SQL 命令。角色和表空间的 **DROP** 命令也会被加入进来。

- **-E encoding** **--encoding=encoding**

用指定的字符集编码创建转储。默认情况下，转储使用数据库的编码创建（另一种得到相同结果的方法是设置 **PGCLIENTENCODING** 环境变量为想要的转储编码）。

- **-f filename** **--file=filename**

将输出发送到指定的文件中。如果省略，将使用标准输出。

- **-g** **--globals-only**

只转储全局对象（角色和表空间），而不转储数据库。

- **-O** **--no-owner**

不输出用于设置对象所有权以符合原始数据库的命令。默认情况下，pg_dumpall发出 **ALTER OWNER** 或 **SET SESSION AUTHORIZATION**
语句来设置被创建的模式元素的所有权。除非脚本是由一个超级用户（或者是拥有脚本中所有对象的同一个用户）所运行，这些语句在脚本运行时会失败。要使得一个脚本能被任意用户恢复，但又不想给予该用户所有对象的所有权，可以指定 **-O**。

- **-r** **--roles-only**

只转储角色，不转储数据库和表空间。

- `-s --schema-only`

只转储对象定义（模式），不转储数据。

- `-S username --superuser=username`

指定要在禁用触发器时使用的超级用户的用户名。只有使用 `--disable-triggers`
时，这个选项才相关（通常，最好省去这个选项，而作为超级用户来启动结果脚本来取而代之）。

- `-t --tablespaces-only`

只转储表空间，不转储数据库和角色。

- `-v --verbose`

指定细节模式。这将导致pg_dumpall向标准错误输出详细的对象注释以及转储文件的开始/停止时间，还有进度消息。 重复该选项会导致在标准错误上出现附加的调试级信息。 这个选项还会被传递到pg_dump。

- `-V --version`

打印pg_dumpall版本并退出。

- `-x --no-privileges --no-acl`

防止转储访问特权（授予/收回命令）。

- `--binary-upgrade`

这个选项用于就地升级功能。我们不推荐也不支持把它用于其他目的。这个选项在未来的发行中可能被改变而不做通知。

- `--column-inserts --attribute-inserts`

将数据转储为带有显式列名的 `INSERT` 命令（`INSERT INTO table(column, …) VALUES …`
）。这将使得恢复过程非常慢，这主要用于使转储能够被载入到非IvorySQL数据库中。

- `--disable-dollar-quoting`

这个选项禁止在函数体中使用美元符号引用，并且强制它们使用 SQL 标准字符串语法被引用。

- `--disable-triggers`

只有在创建一个只转储数据的转储时，这个选项才相关。它指示pg_dumpall包括在数据被重新载入时能够临时禁用目标表上的触发器的命令。如果你在表上有引用完整性检查或其他触发器，并且你在数据重新载入期间不想调用它们，请使用这个选项。当前，为 `--disable-triggers`
发出的命令必须作为超级用户来执行。因此，你还应当使用 `-S`
指定一个超级用户名，或者宁可作为一个超级用户启动结果脚本。

- `--exclude-database=`pattern`

不要转储名字与 `pattern` 匹配的数据库。可以通过编写多个 `--exclude-database` 开关来排除多个模式。`pattern` 参数被解释为模式，根据psql的 `\d` 命令使用的相同规则
，因此，通过在模式中编写通配符也可以排除多个数据库。
使用通配符时，请谨慎的引用模式，如果需要防止shell通配符扩展。

- `--extra-float-digits=ndigits`

在转储浮点数据时使用extra_float_digits规定的值，而不是最大可用精度。备份目的进行的常规转储不使用此选项。

- `--if-exists`

时间条件性命令（即增加一个 `IF EXISTS` 子句）来清除数据库和其他对象。 只有同时指定了 `--clean`

时，这个选项才可用。

- `--inserts`

将数据转储为 `INSERT` 命令（而不是 `COPY`
）。这将使得恢复非常慢，这主要用于使转储能够被载入到非IvorySQL数据库中。注意如果你已经重新安排了列序，该恢复可能会一起失败。`--column-inserts` 选项对于列序改变是安全的，但是会更慢。

- `--load-via-partition-root`

在为一个分区表转储数据时，让 `COPY` 语句或者 `INSERT`
语句把包含它的分区层次的根而不是分区自身作为目标。这导致在数据被装载时，会为每一个行重新确定合适的分区。如果在一台服务器上重新装载数据时会出现行并不是总是落入到和原始服务器上相同的分区中的情况，这个选项就很有用。例如，如果分区列是文本类型并且两个系统中用于排序分区列的排序规则有着不同的定义，就会发生这种情况。

- `--lock-wait-timeout=timeout`

在转储的开始从不等待共享表锁的获得。 而是在指定的 `timeout` 内不能锁定一个表时失败。 超时时长可以用 `SET statement_timeout` 接受的任何格式指定。

- `--no-comments`

不转储注释。

- `--no-publications`

不转储publication。

- `--no-role-passwords`

不为角色转储口令。在恢复完后，角色的口令将是空口令，并且在设置口令之前口令认证都不会成功。由于指定这个选项时并不需要口令值，角色信息将从目录视图 `pg_roles` 而不是 `pg_authid` 中读出。因此，如果对 `pg_authid` 的访问被某条安全性策略所限制，那么这个选项也会有所帮助。

- `--no-security-labels`

不转储安全标签。

- `--no-subscriptions`

不转储subscription。

- `--no-sync`

默认情况下，`pg_dumpall` 将等待所有文件被安全地写入到磁盘。这个选项会让 `pg_dumpall`
不做这种等待而返回，这样会更快，但是意味着后续的操作系统崩溃可能留下被损坏的转储。通常来说，这个选项对测试有用，但不应该在从生产安装中转储数据时使用。

- `--no-tablespaces`

不要输出选择表空间的命令。通过这个选项，在恢复期间所有的对象都会被创建在任何作为默认的表空间中。

- `--no-toast-compression`

不要输出命令以设置 TOAST 压缩方法。 采用这个选项，所有列将会以默认压缩设置来恢复。

- `--no-unlogged-table-data`

不转储非日志记录表的内容。这个选项对于表定义（模式）是否被转储没有影响，它只会限制转储表数据。

- `--on-conflict-do-nothing`

添加 `ON CONFLICT DO NOTHING` 到 `INSERT` 命令。 除非 `--inserts` 或 `--column-inserts` 也被规定，否则此选项不生效。

- `--quote-all-identifiers`

强制引用所有标识符。在从一个与pg_dumpall主版本不同的IvorySQL服务器转储数据库时或者要将输出载入到一个不同主版本的服务器时，推荐使用这个选项。默认情况下，pg_dumpall只会对为其主版本中保留词的标识符加上引号。在与其他版本的具有不同保留词集合的服务器交互时，这有时会导致兼容性问题。使用 `--quote-all-identifiers` 可以阻止这类问题，但是代价是转储脚本会更加难读。

- `--rows-per-insert=nrows`

将数据转储为 `INSERT` 命令（而不是 `COPY`）。控制每个 `INSERT` 命令的最大行数。指定的值必须是大于零的数。重新加载期间的任何错误都将导致仅丢失有问题的 `INSERT` 的行，而不是整个表内容。

- `--use-set-session-authorization`

输出 SQL-标准的 `SET SESSION AUTHORIZATION` 命令取代 `ALTER OWNER` 命令来确定对象的所有关系。这让该转储更加兼容标准，但是取决于该转储中对象的历史，该转储可能无法正常恢复。

- `-? --help`

显示有关pg_dumpall命令行参数的帮助并退出。

下列命令行选项控制数据库连接参数。

- `-d connstr --dbname=connstr`

指定用于连接到服务器的参数，比如 连接字符串；这些将覆盖所有冲突的命令行选项。这个选项被称为 `--dbname` 是为了和其他客户端应用一致，但是因为pg_dumpall需要连接多个数据库，连接字符串中的数据库名将被忽略。使用 `-l` 选项指定一个数据库，该数据库被用于初始连接，这将转储全局对象并且发现需要转储哪些其他数据库。

- `-h host --host=host`

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从 `PGHOST` 环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

- `-l dbname --database=dbname`

指定要连接到哪个数据库转储全局对象以及发现要转储哪些其他数据库。如果没有指定，将会使用 `postgres` 数据库，如果 `postgres` 不存在，就使用 `template1`。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在 `PGPORT` 环境变量中（如果被设置），否则使用编译在程序中的默认值。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制pg_dumpall在连接到一个数据库之前提示要求一个口令。这个选项从来不是必须的，因为如果服务器要求口令认证，pg_dumpall将自动提示要求一个口令。但是，pg_dumpall将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入 `-W` 来避免额外的连接尝试。注意对每个要被转储的数据库，口令提示都会再次出现。通常，最好设置一个 `~/.pgpass` 文件来减少手工口令输入。

- `--role=`rolename`

指定一个用来创建该转储的角色名。这个选项导致pg_dump在连接到数据库后发出一个 `SET ROLE rolename` 命令。当已认证用户（由 `-U` 指定）缺少pg_dump所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

环境

- `PGHOST PGOPTIONS PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

注解

因为pg_dumpall在内部调用pg_dump，所以， 一些诊断消息可以参考pg_dump。

即使当用户的目的是把转储脚本恢复到一个空的集簇中，`--clean` 选项也有用武之地。`--clean` 的使用让该脚本删除并且重建内建的 `postgres` 和 `template1` 数据库，确保这两个数据库保持与源集簇中相同的属性（例如locale和编码）。如果不用这个选项，这两个数据库将保持它们现有的数据库级属性以及任何已有的内容。

一旦恢复，建议在每个数据库上运行 `ANALYZE`，这样优化器就可以得到有用的统计信息。你也可以运行 `vacuumdb -a -z` 来分析所有数据库。

不应该预期转储脚本运行到结束都不出错。特别是由于脚本将为源集簇中已有的每一个角色发出 `CREATE ROLE` 语句，对于bootstrap超级用户当然会得到一个“role already exists”错误，除非目标集簇用一个不同的bootstrap超级用户名完成的初始化。这种错误是无害的并且应该被忽略。`--clean` 选项的使用很可能会产生额外的有关于不存在对象的无害错误消息，不过可以通过加上 `--if-exists` 减少这类错误消息。

pg_dumpall要求所有需要的表空间目录在进行恢复之前就必须存在；否则，数据库创建就会由于在非默认位置创建数据库而失败。

例子

要转储所有数据库：

```
$ pg_dumpall > db.out
```

要从这个文件重新载入数据库，你可以使用：

```
$ psql -f db.out postgres
```

这里你连接哪一个数据库并不重要，因为由pg_dumpall创建的脚本将包含合适的命令来创建和连接到被保存的数据库。一个例外是，如果指定了 `--clean`，则开始时必须连接到 `postgres` 数据库，该脚本将立即尝试删除其他数据库，并且这种动作对于已连接上的这个数据库将会失败。

pg_isready

pg_isready — 检查一个IvorySQL服务器的连接状态

大纲

`pg_isready` [`connection-option`…] [`option`…]

选项

- `-d dbname --dbname=dbname`

指定要连接的数据库名。`dbname` 可以是连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

- `-h hostname --host=hostname`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认值取自 `PGPORT` 环境变量。如果环境变量没有设置，则默认值使用编译时指定的端口（通常是 5432）。

- `-q --quiet`

不显示状态消息。当脚本编程时有用。

- `-t seconds --timeout=seconds`

尝试连接时，在返回服务器不响应之前等待的最大秒数。设置为 0 则禁用。默认值是 3 秒。

- `-U username --username=username`

作为用户 `username` 连接数据库，而不是用默认用户。

- `-V --version`

打印pg_isready版本并退出。

- `-? --help`

显示有关pg_isready命令行参数的帮助并退出。

退出状态

如果服务器正常接受连接，pg_isready返回 `0` 给 shell；如果服务器拒绝连接（例如处于启动阶段）则返回 `1`；如果连接尝试没有被相应则返回 `2`；如果没有尝试（例如由于非法参数）则返回 `3`。

环境

和大部分其他IvorySQL工具相似，`pg_isready` 也使用libpq支持的环境变量。

环境变量 `PG_COLOR` 规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

注解

要获得服务器状态，不一定需要提供正确的用户名、口令或数据库名。
不过，如果提供了不正确的值，服务器将会记录一次失败的连接尝试。

例子

标准用法：

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

使用连接参数运行连接到处于启动中的IvorySQL集簇：

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

使用连接参数运行连接到无响应的IvorySQL集簇：

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — 以流的方式从一个IvorySQL服务器得到预写式日志

大纲

`pg_receivewal` [`option`…]

选项

- `-D directory` `--directory=directory`

要把输出写到哪个目录。这个参数是必需的。

- `-E lsn` `--endpos=lsn`

当接收到达指定的LSN时，自动停止复制并且以正常退出状态0退出。如果有一个记录的LSN正好等于 `lsn`，则该记录将会被处理。

- `--if-not-exists`

当指定 `--create-slot` 并且具有指定名称的槽已经存在时不要抛出错误。

- `-n --no-loop`

不要在连接错误上循环。相反，碰到一个错误时立刻退出。

- `--no-sync`

这个选项导致 `pg_receivewal`
不强制WAL数据被刷回磁盘。这样会更快，但是也意味着接下来的操作系统崩溃会让WAL段损坏。通常，这个选项对于测试有用，但不应该在对生产部署进行WAL归档时使用。这个选项与 `--synchronous` 不兼容。

- `-s interval --status-interval=interval`

指定发送回服务器的状态包之间的秒数。这允许我们更容易地监控服务器的进度。
一个零值完全禁用这种周期性的状态更新，不过当服务器需要时还是会有一个更新
会被发送来避免超时导致的断开连接。默认值是 10 秒。

- `-S slotname --slot=slotname`

要求pg_receivewal使用一个已有的复制槽。在使用这个选项时，
pg_receivewal将会报告给服务器一个刷写位置，指示每一个
段是何时被同步到磁盘的，这样服务器可以在不需要该段时移除它。当pg_receivewal的复制客户端在服务器上被配置为一个同步后备时，那么使用一个复制槽将会向服务器报告刷写 位置，但只在一个 WAL
文件被关闭时报告。因此，该配置将导致主服务
器上的事务等待很长的时间并且无法令人满意地工作。要让这种配置工作 正确，还必须制定选项
`--synchronous` （见下文）。

- `--synchronous`

在 WAL 数据被收到后立即刷入到磁盘。还要在刷写后立即向服务器回送 一个状态包（不考虑 `--status -interval`）。如果pg_receivewal的复制客户端在服务器
上被配置为一个同步后备，应该指定这个选项来确保向服务器发送及时的反馈。

- `-v --verbose`

启用冗长模式。

- `-Z level --compress=level`

启用预写式日志上的gzip压缩，并且指定压缩级别（0到9，0是不压缩而9是最大压缩）。所有的文件名后都将被追加后缀 `.gz`。

下列命令行选项控制数据库连接参数。

- `-d connstr --dbname=connstr`

指定用于连接到服务器的参数，作为
连接字符串；这些将覆盖所有冲突的命令行选项。为了和其他客户端应用一致，该选项被称为 `--dbname`。但是因为pg_receivewal并不连接到集簇中的任何特定数据库，连接字符串中的数据库名将被忽略。

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。默认值取自 `PGHOST` 环境变量（如果设置），否则会尝试一个 Unix 域套接字连接。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。默认用 `PGPORT`
环境变量中的值（如果设置），或者一个编译在程序中的默认值。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制pg_receivewal在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，pg_receivewal将自动提示要求一个口令。但是，pg_receivewal将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

为了控制物理复制槽，pg_receivewal 可以执行下列两种动作之一：

- `--create-slot`

用 `--slot` 中指定的名称创建一个新的物理复制槽， 然后退出。

- `--drop-slot`

删除 `--slot` 中指定的复制槽，然后退出。

其他选项也可用：

- `-V --version`

打印pg_receivewal版本并退出。

- `-? --help`

显示有关pg_receivewal命令行参数的帮助并退出。

退出状态

在被SIGINT信号终止（没有正常的方式结束它。因此这不是一种错误）时，pg_receivewal将以状态0退出。对于致命错误或者其他信号，退出状态将不是零。

环境

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

环境变量 `PG_COLOR` 规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

注解

在使用pg_receivewal替代 archive_command 作为主要的 WAL 备份方法时，强烈建议使用复制槽。否则，服务器可能会在预写式日志文件被备份好之前重用或者移除它们，因为没有任何信息（不管是来自 archive_command 或是复制槽）能够指示 WAL 流已经被归档到什么程度。不过要注意，如果接收者没有持续地取走 WAL 数据，一个复制槽将会填满服务器的磁盘空间。

如果在源集簇上启用了组权限，pg_receivewal将保留接收到的WAL文件上的组权限。

例子

要从位于 `mydbserver` 的服务器流式传送预写式日志并且将它存储在本地目录`/usr/local/pgsql/archive`：

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

pg_recvlogical

pg_recvlogical — 控制 IvorySQL 逻辑解码流

大纲

`pg_recvlogical` [`option`…]

选项

必须至少要指定下列选项之一来选择一个动作：

- `--create-slot`

为 `--dbname` 指定的数据库用 `--slot` 指定的名称创建一个新的逻辑复制槽，使用 `--plugin` 指定的输出插件。

- `--drop-slot`

删除名称由 `--slot` 指定的复制槽，然后退出。

- `--start`

从 `--slot` 指定的逻辑复制槽开始进行流式传送更改，一直继续
到被一个信号终止。如果服务器端关机或者断开连接导致更改流结束，会进入一个 循环一直重试，通过指定 `--no-loop`
可以防止这种情况下进入循环重试。流格式由槽创建时指定的输出插件决定。连接必须是连接到用于创建该槽的同一个数据库上。

`--create-slot` 和 `--start` 可以被一起指定。 `--drop-slot` 不能和另一个动作组合在一起。

下面的命令行选项控制输出的位置和格式以及其他复制行为：

- `-E` `lsn` `--endpos=`lsn`

在 `--start`
模式中，当接收过程到达指定的LSN时会自动地停止复制并且以正常的退出状态0退出。如果不处于 `--start` 模式时指定这个选项，则会发生错误。如果有一个记录的LSN正好等于 `lsn`，则该记录将被输出。`--endpos` 不会察觉到事务边界并且可能会在一个事务中间截断输出。任何部分输出的事务都将不会被消费，并且在下一次从该槽中读取时将会重放该事务。单个的消息不会被截断。

- `-f filename` `--file=filename`

把接收到并且解码好的事务数据写入到一个文件。使用 `-` 可以写到stdout。

- `-F interval_seconds` `--fsync-interval=interval_seconds`

指定pg_recvlogical发出 `fsync()`
调用确保输出文件被安全地刷到磁盘的频度。服务器将会偶尔要求客户端执行一次刷写并且把刷写位置报告给服务器。 这个设置可以在此之外更加频繁地执行刷写。指定间隔为 `0` 会完全禁止发出 `fsync()`
调用，但是仍会报告进度给服务器。在这种情况下，发生崩溃会导致数据丢失。

- `-I lsn` `--startpos=lsn`

在 `--start` 模式中，从给定的 LSN 开始复制。

- `--if-not-exists`

当指定 `--create-slot` 并且具有指定名称的槽已经存在时不要抛出错误。

- `-n` `--no-loop`

当服务器连接丢失时，不要在循环中重试，直接退出。

- `-o name [=value] --option=name[=value]`

如果指定了输出插件，把选项值 `value` 传递给选项 `name`。存在哪些选项以及它们的效果取决于使用的输出插件。

- `-P plugin --plugin=plugi`

在创建一个槽时使用指定的逻辑解码输出插件。如果该槽已经存在，这个选项没有效果。

- `-s interval_seconds --status-interval=interval_seconds`

这个选项和 pg_receivewal 中的同名选项具有相同的效果。请参考那里的描述。

- `-S slot_name --slot=slot_name`

在 `--start` 模式中，使用名为 `slot_name` 的已有逻辑复制槽。在 `--create-slot` 模式中，使用这个名称创建该槽。在 `--drop-slot` 模式中，删除这个名称指定的槽。

- `-v --verbose`

开启详细输出模式。

下列命令行选项控制数据库连接参数。

- `-d database --dbname=database`

要连接的数据库。这个选项的详细含义请见动作的描述。`dbname` 可以是 连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。默认为用户名。

- `-h hostname-or-ip --host=hostname-or-ip`

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线， 它被用作一个 Unix 域套接字的目录。默认是从 `PGHOST` 环境变量中取得（如果被设置）， 否则将尝试一次 Unix 域套接字连接。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。 默认是放在 `PGPORT` 环境变量中（如果被设置）， 否则使用编译在程序中的默认值。

- `-U user --username=user`

要作为哪个用户连接。默认是用当前操作系统用户名。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有 其他方式提供口令（例如一个 `.pgpass` 文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用， 因为在其中没有一个用户来输入口令。

- `-W --password`

强制pg_dump在连接到一个数据库之前提示要求一个口令。这个选项不是必须的，因为如果服务器要求口令认证， pg_dump将自动提示要求一个口令。 但是，pg_dump将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入 `-W` 来避免额外的连接尝试。

还有下列附加选项可用：

- `-V --version`

打印pg_recvlogical的版本并且退出。

- `-? --help`

显示关于pg_recvlogical命令行参数的帮助，并且退出。

环境

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

环境变量`PG_COLOR`规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

注解

如果在源服务器上启用了组权限，pg_recvlogical将会在接收到的WAL文件上保留组权限。

pg_restore

pg_restore — 从一个由pg_dump创建的归档文件恢复一个IvorySQL数据库

大纲

`pg_restore` [`connection-option`…] [`option`…] [`filename`]

选项

pg_restore接受下列命令行参数。

- `filename`

指定要被恢复的归档文件（对于一个目录格式的归档则是目录）的位置。如果没有指定，则使用标准输入。

- `-a` `--data-only`

只恢复数据，不恢复模式（数据定义）。如果在归档中存在，表数据、大对象和序列值会被恢复。这个选项类似于指定 `--section=data`，但是由于历史原因两者不完全相同。

- `-c` `--clean`

在重新创建数据库对象之前清除（丢弃）它们（除非使用了 `--if-exists`，如果有对象在目标数据库中不存在，这可能会生成一些无害的错误消息）。

- `-C` `--create`

在恢复一个数据库之前县创建它。如果还指定了 `--clean`，在连接到目标数据库之前丢弃并且重建它。如果使用 `--create`，pg_restore还会恢复数据库的注释（如果有）以及与其相关的配置变量设置，也就是任何提到过这个数据库的 `ALTER DATABASE … SET …` 和 `ALTER ROLE … IN DATABASE … SET …` 命令。不管是否指定 `--no-acl`，数据库本身的访问特权都会被恢复。在使用这个选项时，`-d` 提到的数据库只被用于发出初始的 `DROP DATABASE` 和 `CREATE DATABASE` 命令。所有要恢复到该数据库名中的数据都出现在归档中。

- `-d dbname` `--dbname=dbname`

连接到数据库 `dbname` 并且直接恢复到该数据库中。`dbname` 可以是 连接字符串。如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

- `-e` `--exit-on-error`

在发送 SQL 命令到该数据库期间如果碰到一个错误就退出。默认行为是继续并且在恢复结束时显示一个错误计数。

- `-f filename` `--file=filename`

为生成的脚本指定输出文件，或在与 `-l` 选项一起使用时为列表指定输出文件。为 stdout用 `-`。

- `-F format` `--format=format`

指定归档的格式。并不一定要指定该格式，因为pg_restore将会自动决定格式。如果指定，可以是下列之一：`c custom` 归档是pg_dump的自定义格式。`d directory` 归档是一个目录归档。`t tar` 归档是一个 `tar` 归档。

- `-I index` `--index=index`

只恢复提及的索引的定义。可以通过写多个 `-I` 开关指定多个索引。

- `-j number-of-jobs` `--jobs=number-of-jobs`

使用并发任务运行pg_restore中最耗时的步骤 — 载入数据、创建索引或者创建约束； 同时，最多使用 `number-of-jobs` 个并发会话。对于一个运行在多处理器机器上的服务器，这个选项能够大幅度减少恢复一个大型数据库的时间。当发出脚本而不是直接连接到数据库服务器时，将忽略此选项。每一个任务是一个进程或者一个线程，这取决于操作系统，它们都使用一个独立的服务器连接。这个选项的最佳值取决于服务器、客户端以及网络的硬件设置。因素包括 CPU 的核心数和磁盘设置。一个好的建议是服务器上 CPU 的核心数，但是更大的值在很多情况下也能导致更快的恢复时间。当然，过高的值会由于超负荷反而导致性能降低。这个选项只支持自定义和目录归档格式。输入必须是一个常规文件或目录（例如，不能是一个管道或者标准输出）。还有，多任务不能和选项 `--single-transaction` 一起用。

- `-l` `--list`

列出归档的内容的表格。这个操作的输出能被用作 `-L` 选项的输入。注意如果把 `-n` 或 `-t` 这样的过滤开关与 `-l` 一起使用，它们将会限制列出的项。

- `-L list-file` `--use-list=list-file`

只恢复在 `list-file` 中列出的归档元素，并且按照它们出现在该文件中的顺序进行恢复。注意如果把 `-n` 或 `-t` 这样的过滤开关与 `-L` 一起使用，它们将会进一步限制要恢复的项。`list-file` 通常是编辑一个 `-l` 操作的输出来创建。行可以被移动或者移除，并且也可以通过在行首放一个（`;`）将其注释掉。例子见下文。

- `-n schema` `--schema=schema`

只恢复在被提及的模式中的对象。可以用多个 `-n` 开关来指定多个模式。这可以与`-t`选项组合在一起只恢复一个指定的表。

- `-N schema` `--exclude-schema=schema`

不恢复所提及方案中的对象。可以用多个 `-N` 开关指定多个要被排除的方案。如果对同一个方案名称同时给出了 `-n` 和 `-N`，则 `-N` 会胜出并且该方案会被排除。

- `-O` `--no-owner`

不要输出将对象的所有权设置为与原始数据库匹配的命令。默认情况下，pg_restore会发出 `ALTER OWNER` 或者 `SET SESSION AUTHORIZATION` 语句来设置已创建的模式对象的所有权。除非到该数据库的初始连接是一个超级用户（或者拥有脚本中所有对象的同一个用户）建立的，这些语句将会失败。通过 `-O`，任何用户名都可以被用于初始连接，并且这个用户将会拥有所有被创建的对象。

- `-P function-name(argtype [, …])` `--function=function-name(argtype [, …])`

只恢复被提及的函数。要小心地拼写函数的名称和参数使它们正好就是出现在转储文件的内容表中的名称和参数。可以使用多个 `-P` 开关指定多个函数。

- `-R` `--no-reconnect`

这个选项已被废弃，但是出于向后兼容性的目的，系统仍然还接受它。

- `-s` `--schema-only`

只恢复归档中的模式（数据定义）不恢复数据。这个选项是 `--data-only` 的逆选项。它与指定 `--section=pre-data` `--section=post-data` 相似，但是由于历史原因并不完全相同。（不要把这个选项和 `--schema` 选项弄混，后者把词“schema”用于一种不同的含义）。

- `-S username` `--superuser=username`

指定在禁用触发器时要用的超级用户名。只有使用 `--disable-triggers` 时这个选项才相关。

- `-t table` `--table=table`

只恢复所提及的表的定义和数据。出于这个目的，“table”包括视图、物化视图、序列和外部表。可以写上多个 `-t` 开关可以选择多个表。这个选项可以和 `-n` 选项结合在一起指定一个特定模式中的表。注意在指定 `-t` 时，pg_restore不会尝试恢复所选表可能依赖的任何其他数据库对象。因此，无法确保能成功地把一个特定表恢复到一个干净的数据库中。注意这个标志的行为和pg_dump的 `-t` 标志不一样。在pg_restore中当前没有任何通配符匹配的规定，也不能在其 `-t` 选项中包括模式的名称。而且，虽然pg_dump的 `-t` 标志也会转储选中表的附属对象（例如索引），但是pg_restore的 `-t` 标志不包括这些附属对象。

- `-T trigger` `--trigger=trigger`

只恢复所提及的触发器。可以用多个 `-T` 开关指定多个触发器。

- `-v` `--verbose`

指定冗长模式。这将导致pg_restore输出详细的对象注释和启动/停止时间到输出文件，并将推送消息输出到标准错误。重复该选项会造成附加的调试级消息出现在标准报错上。

- `-V` `--version`

打印该pg_restore的版本并退出。

- `-x` `--no-privileges` `--no-acl`

阻止恢复访问特权（授予/收回命令）。

- `-1` `--single-transaction`

将恢复作为单一事务执行（即把发出的命令包裹在 `BEGIN` / `COMMIT` 中）。这可以确保要么所有命令完全成功，要么任何改变都不被应用。这个选项隐含了 `--exit-on-error`。

- `--disable-triggers`

只有在执行一个只恢复数据的恢复时，这个选项才相关。它指示pg_restore在装载数据时执行命令临时禁用目标表上的触发器。如果你在表上有参照完整性检查或者其他触发器并且你不希望在数据载入期间调用它们时，请使用这个选项。目前，为 `--disable-triggers` 发出的命令必须以超级用户身份完成。因此你还应该用`-S`指定一个超级用户名，或者更好的方法是以一个IvorySQL超级用户运行pg_restore。

- `--enable-row-security`

只有在恢复具有行安全性的表的内容时，这个选项才相关。默认情况下，pg_restore将把 row_security 设置为 off 来确保所有数据都被恢复到表中。如果用户不拥有足够绕过行安全性的特权，那么会抛出一个错误。这个参数指示pg_restore把 row_security 设置为on允许用户尝试恢复启用了行安全性的表的内容。如果用户没有从转储向表中插入行的权限，这仍将失败。注意当前这个选项还要求转储处于 `INSERT` 格式，因为 `COPY FROM` 不支持行安全性。

- `--if-exists`

使用条件命令（即增加一个 `IF EXISTS` 子句）删除数据库对象。只有指定了 `--clean` 时，这个选项才有效。

- `--no-comments`

即便归档中包含注释也不输出恢复注释的命令。

- `--no-data-for-failed-tables`

默认情况下，即便表的创建命令失败（例如因为表已经存在），表数据也会被恢复。通过这个选项，对这类表的数据会被跳过。如果目标数据库已经包含了想要的表内容，这种行为又很有有用。例如，IvorySQL扩展（如PostGIS）的辅助表可能已经被载入到目标数据库中，指定这个选项就能阻止把重复的或者废弃的数据载入到这些表中。只有当直接恢复到一个数据库中时这个选项才有效，在产生SQL脚本输出时这个选项不会产生效果。

- `--no-publications`

即便归档中包含publication也不输出恢复publication的命令。

- `--no-security-labels`

不要输出恢复安全标签的命令，即使归档中包含安全标签。

- `--no-subscriptions`

即便归档中包含subscription也不输出恢复subscription的命令。

- `--no-tablespaces`

不输出命令选择表空间。通过这个选项，所有的对象都会被创建在恢复时的默认表空间中。

- `--section=`sectionname`

只恢复提及的小节。小节的名称可以是 `pre-data`、`data` 或者 `post-data`。可以把这个选项指定多次来选择多个小节。默认值是恢复所有小节。数据小节包含实际的表数据以及大对象定义。Post-data 项由索引定义、触发器、规则和除已验证的检查约束之外的约束构成。Pre-data 项由所有其他数据定义项构成。

- `--strict-names`

要求每一个模式（`-n` / `--schema`）以及表（`-t` / `--table`）限定词匹配备份文件中至少一个模式/表。

- `--use-set-session-authorization`

输出 SQL 标准的 `SET SESSION AUTHORIZATION` 命令取代 `ALTER OWNER` 命令来决定对象拥有权。这会让转储更加兼容标准，但是依赖于转储中对象的历史，可能无法正确恢复。

- `-? --help`

显示有关pg_restore命令行参数的帮助，并且退出。

pg_restore也接受下列用于连接参数的命令行参数：

- `-h host --host=host`

指定服务器正在运行的机器的主机名。如果该值开始于一个斜线，它被用作一个 Unix 域套接字的目录。默认是从 `PGHOST` 环境变量中取得（如果被设置），否则将尝试一次 Unix 域套接字连接。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展名。默认是放在 `PGPORT` 环境变量中（如果被设置），否则使用编译在程序中的默认值。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那么连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制pg_restore在连接到一个数据库之前提示要求一个口令。这个选项不是必须的，因为如果服务器要求口令认证，pg_restore将自动提示要求一个口令。但是，pg_restore将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下，值得键入 `-W` 来避免额外的连接尝试。

- `--role=`rolename`

指定一个用来创建该转储的角色名。这个选项导致pg_restore在连接到数据库后发出一个 `SET ROLE rolename` 命令。当已认证用户（由 `-U` 指定）缺少pg_restore所需的特权但是能够切换到一个具有所需权利的角色时，这个选项很有用。一些安装有针对直接作为超级用户登录的策略，使用这个选项可以让转储在不违反该策略的前提下完成。

环境

- `PGHOST PGOPTIONS PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

当使用 `-d` 选项指定一个直接数据库连接时，pg_restore在内部执行 `SELECT` 语句。如果你运行pg_restore时出现问题，确定你能够从正在使用的数据库中选择信息，例如psql。此外，libpq前端-后端库所使用的任何默认连接设置和环境变量都将适用。

注解

如果你的数据库集簇对于 `template1` 数据库有任何本地添加，要注意将pg_restore的输出载入到一个真正的空数据库。否则你很可能由于以增加对象的重复定义而得到错误。要创建一个不带任何本地添加的空数据库，从 `template0` 而不是 `template1` 复制它，例如：

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

下面将详细介绍pg_restore的局限性。

- 在恢复数据到一个已经存在的表中并且使用了选项 `--disable-triggers` 时，pg_restore会在插入数据之前发出命令禁用用户表上的触发器，然后在完成数据插入后重新启用它们。如果恢复在中途停止，可能会让系统目录处于错误的状态。
- pg_restore不能有选择地恢复大对象，例如只恢复特定表的大对象。如果一个归档包含大对象，那么所有的大对象都会被恢复，如果通过 `-L`、 `-t` 或者其他选项进行了排除，它们一个也不会被恢复。

一旦完成恢复，应该在每一个被恢复的表上运行 `ANALYZE`，这样优化器能得到有用的统计信息。

示例

假设我们已经以自定义格式转储了一个叫做 mydb 的数据库：

```
$ pg_dump -Fc mydb > db.dump
```

要删除该数据库并且从转储中重新创建它：

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

-d 开关中提到的数据库可以是任何已经存在于集簇中的数据库，pg_restore只会用它来为 mydb 发出 CREATE DATABASE 命令。通过 -C，数据总是会被恢复到出现在归档文件的数据库名中。

要把转储重新载入到一个名为 newdb 的新数据库中：

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

注意我们不使用 -C，而是直接连接到要恢复到其中的数据库。还要注意我们是从 template0 而不是 template1 创建了该数据库，以保证它最初是空的。

要对数据库项重排序，首先需要转储归档的表内容：

```
$ pg_restore -l db.dump > db.list
```

列表文件由一个头部和一些行组成，这些行每一个都用于一个项，例如：

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
```

```
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

分号表示开始一段注释，行首的数字表明了分配给每个项的内部归档 ID。

文件中的行可以被注释掉、删除以及重排序。例如：

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

把这样一个文件作为pg_restore的输入将会只恢复项 10 和 6，并且先恢复 10 再恢复 6。

```
$ pg_restore -L db.list db.dump
```

pg_verifybackup

pg_verifybackup — 验证IvorySQL集群的基础备份的完整性

大纲

`pg_verifybackup` [`option`…]

选项

pg_verifybackup 接受以下命令行参数：

- `-e --exit-on-error`

检测到备份问题后立即退出。 如果没有指定这个选项，`pg_verifybackup` 将在检测到问题后继续检查备份，并将检测到的所有问题报告为错误。

- `-i path --ignore=path`

在将备份中实际存在的数据文件列表与 `backup_manifest` 文件中列出的数据文件列表进行比较时，忽略指定的文件或目录，该文件或目录应表示为相对路径名。如果指定了目录，则此选项会影响以该位置为根的整个子树。
如果相对路径名与指定的路径名匹配，有关额外文件、丢失文件、文件大小差异或校验和不匹配的投诉将被抑制。 可以多次指定此选项。

- `-m path --manifest-path=path`

使用指定路径的清单文件，而不是位于备份目录根目录中的清单文件。

- `-n --no-parse-wal`

不要试图解析从该备份恢复所需的预写式日志数据。

- `-q --quiet`

成功验证备份后不要打印任何内容。

- `-s --skip-checksums`

不要验证数据文件校验和。但仍检查是否存在文件以及这些文件的大小。这样将会快得多，因为文件本身不需要读取。

- `-w path --wal-directory=path`

尝试解析存储在指定目录中的 WAL 文件，而不是 `pg_wal`。
如果备份存储在与WAL存档不同的位置，则这可能很有用。

其他选项也可用：

- `-V --version`

打印 pg_verifybackup 版本并退出。

- `-? --help`

显示有关pg_verifybackup命令行参数的帮助，然后退出。

示例

要在 `mydbserver` 上创建服务器的基本备份并验证备份的完整性：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ pg_verifybackup /usr/local/pgsql/data
```

要在 `mydbserver` 上创建服务器的基本备份，请将清单移动到备份目录之外的某个位置，并验证备份：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/backup1234
$ mv /usr/local/pgsql/backup1234/backup_manifest
/my/secure/location/backup_manifest.1234
$ pg_verifybackup -m /my/secure/location/backup_manifest.1234
/usr/local/pgsql/backup1234
```

要在忽略手动添加到备份目录的文件的同时验证备份，并跳过校验和验证：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ edit /usr/local/pgsql/data/note.to.self
$ pg_verifybackup --ignore=note.to.self --skip-checksums /usr/local/pgsql/data
```

psql

psql — IvorySQL的交互式终端

大纲

`psql` [`option`…] [`dbname` [`username`]]

选项

- `-a --echo-all`

把所有非空输入行按照它们被读入的形式打印到标准输出（不适用于交互式行读取）。这等效于把变量 ECHO 设置为 all。

- -A --no-align

切换到非对齐输出模式（默认输出模式是 对齐 的）。这等效于 \pset format unaligned。

- -b --echo-errors

把失败的 SQL 命令打印到标准错误输出。这等效于把变量 ECHO 设置为 errors。

- -c command --command=command

指定psql执行一个给定的命令字符串 command。这个选项可以重复多次并且以任何顺序与 -f 选项组合在一起。当 -c 或者 -f 被指定时，psql不会从标准输入读取命令，直到它处理完序列中所有的 -c 和 -f 选项之后终止。command 必须是一个服务器完全可解析的命令字符串（即不包含psql相关的特性）或者单个反斜线命令。因此不能在一个 -c 选项中混合SQL和psql元命令。要那样做，可以使用多个 -c 选项或者把字符串用管道输送到psql中，例如： psql -c '\x' -c 'SELECT * FROM foo;' ` 或者 `echo '\x \\ SELECT * FROM foo;' | psql ` （ `\\ 是分隔符元命令）。每一个被传递给 -c 的SQL命令字符串会被当做一个单独的请求发送给服务器。因此，即便该字符串包括多个SQL命令，服务器也会把它当做一个事务来执行，除非在该字符串中有显式的 BEGIN / COMMIT 命令把它划分成多个事务。此外，psql只会打印出该字符串中最后一个SQL命令的结果。这和从文件中读取同一字符串或者把同一字符串传给psql的标准输出时的行为不同，因为那两种情况下psql会独立地发送每一个SQL命令。由于这种行为，把多于一个SQL命令放在 -c 字符串中通常会得到意料之外的结果。最好使用多个 -c 命令或者把多个命令输送给psql的标准输入，按照上文所说的使用echo或者通过一个shell，例如： `psql <<EOF \x SELECT * FROM foo; EOF `

- --csv

切换到CSV（逗号分隔值）输出模式。 这相当于 \pset format csv。

- -d dbname --dbname=dbname

指定要连接的数据库的名称。这等效于指定 dbname 为命令行上的第一个非选项参数。dbname 可以是连接字符串。 如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

- -e --echo-queries

也把发送到服务器的所有 SQL 命令复制到标准输出。这等效于把变量 ECHO 设置为 queries。

- -E --echo-hidden

回显 \d 以及其他反斜线命令生成的实际查询。可以用它来学习psql的内部操作。这等效于把变量 ECHO_HIDDEN 设置为 on。

- -f filename --file=filename

从文件 filename 而不是标准输入中读取命令。这个选项可以被重复多次，也可以以任意顺序与 -c 选项组合。当 -c 或者 -f 被指定时，psql不会从标准输入读取命令，直到它处理完序列中所有的 -c 和 -f 选项之后终止。除此以外，这个选项很大程度上等价于元命令 \i。如果 filename 是 -（连字符），那么会读取标准输入直到遇见一个 EOF 指示或者 \q 元命令。这种方式可以用把自多个文件的输入组合成一种交互式输入。不过注意在这种情况下不会使用 Readline（很像指定了 -n 的情况）。使用这个选项与 psql < filename 有细微的不同。通常，两种形式都可以做到我们所期望的，但是使用 -f 启用了一些好的特性，例如带有行号的错误消息。使用这个选项还有一丝机会可以降低启动开销。在另一方面，使用shell输入重定向的变体（理论上）保证会得到与手工输入时相同的输出。

- -F separator --field-separator=separator

使用 separator 作为非对齐输出的域分隔符。这等效于 \pset fieldsep 或者 \f。

- `-h hostname --host=hostname`

指定运行服务器的机器的主机名。如果这个值由一个斜线开始，它会被用作 Unix 域套接字的目录。

- `-H --html`

切换到HTML输出模式。这等效于 `\pset format html` 或者 `\H` 命令。

- `-l --list`

列出所有可用的数据库，然后退出。其他非连接选项会被忽略。这与元命令 `\list` 类似。在使用这个选项时，psql将连接到数据库 `postgres`，除非在命令行上提及一个不同的数据（选项 `-d` 或非选项参数，可能是通过一个服务项，但不能通过一个环境变量）。

- `-L filename --log-file=filename`

除了把所有查询输出写到普通输出目标之外，还写到文件 `filename` 中。

- `-n --no-readline`

不使用Readline做行编辑并且不使用命令历史。在剪切和粘贴时，关掉 Tab 展开会有所帮助。

- `-o `filename` --output=`filename``

把所有查询输出放到文件 `filename` 中。这等效于命令 `\o`。

- `-p port --port=port`

指定服务器用于监听连接的 TCP 端口或者本地 Unix 域套接字文件扩展。默认是 `PGPORT` 环境变量的值，如果没有设置，则默认为编译时指定的端口号（通常是5432）。

- `-P assignment --pset=assignment`

以 `\pset` 的形式指定打印选项。注意，这里你必须用一个等号而不是空格来分隔名称和值。例如，要设置输出格式为 LaTeX，应该写成 `-P format=latex`。

- `-q --quiet`

指定psql应该安静地工作。默认情况下，它会打印出欢迎消息以及多种输出。如果使用了这个选项，以上那些就都不会输出。在使用 `-c` 选项时，配合这个选项很有用。这等效于设置变量 `QUIET` 为 `on`。

- `-R separator --record-separator=separator`

把 `separator` 用作非对齐输出的记录分隔符。这等效于 `\pset recordsep` 命令。

- `-s --single-step`

运行在单步模式中。这意味着在每个命令被发送给服务器之前都会提示用户一个可以取消执行的选项。使用这个选项可以调试脚本。

- `-S --single-line`

运行在单行模式中，其中新行会终止一个 SQL 命令，就像分号的作用一样。注意这种模式被提供给那些坚持使用它的用户，但是并不一定要使用它。特别地，如果在一行中混合了SQL和元命令，那对于没有经的用户来说，它们的执行顺序可能不总是那么清晰。

- `-t --tuples-only`

关闭打印列名和结果行计数页脚等。这等效于 `\t` 或者 `\pset tuples_only` 命令。

- `-T table_options --table-attr=table_options`

指定要替换HTML table 标签的选项。详见 \pset tableattr。

- -U username --username=username

作为用户 username 而不是默认用户连接到数据库（当然，你必须具有这样做的权限）。

- -v assignment --set=assignment --variable=assignment

执行一次变量赋值，和 \set
元命令相似。注意你必须在命令行上用等号分隔名字和值（如果有）。要重置一个变量，去掉等号就行。要把一个变量置为空值，使用等号但是去掉值。这些赋值在命令行处理期间被完成，因此反映连接状态的变量将在稍后被覆盖。

- -V --version

打印psql版本并且退出。

- -w --no-password

从不发出一个口令提示。如果服务器要求口令认证并且口令不能从其他来源（例如一个 .pgpass
文件）获得，那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。注意这个选项将对整个会话保持设置，并且因此它会影响元命令 \connect
的使用，就像初始的连接尝试那样。

- -W --password

强制psql在连接到一个数据库之前提示要求一个口令，即使口令不会被使用。如果服务器需要口令认证并且口令不能从其他来源获得，例如 .pgpass 文件，psql 在任何情况下都会提示输入口令。 然而，psql
将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 -W
来避免额外的连接尝试。注意这个选项将对整个会话保持设置，并且因此它会影响元命令 \connect
的使用，就像初始的连接尝试那样。

- -x --expanded

打开扩展表格式模式。这等效于 \x 或者 \pset expanded 命令。

- -X, --no-psqlrc

不读取启动文件（要么是系统范围的 psqlrc 文件，要么是用户的 ~/.psqlrc 文件）。

- -z --field-separator-zero

设置非对齐输出的域分隔符为零字节。这等效于 \pset fieldsep_zero。

- -0 --record-separator-zero

设置非对齐输出的记录分隔符为零字节。例如，这对与 xargs -0 配合有关。这等效于 \pset
recordsep_zero。

- -1 --single-transaction

这个选项只能被用于与一个或者多个 -c 以及/或者 -f 选项组合。它会让psql在第一个上述选项之前发出一条 BEGIN 命令并且在最后一个上述选项之后发出一条 COMMIT
命令，这样就把所有的命令都包裹在一个事务中。这个选项可以保证要么所有的命令都成功地完成，要么不应用任何更改。如果命令本身包含 BEGIN 、 COMMIT 或者
ROLLBACK，这个选项将不会得到想要的效果。还有，如果当个命令不能在一个事务块中执行，指定这个选项将导致整个事务失败。

- -? --help[=`topic]`

显示有关psql的帮助并且退出。可选的 topic 参数（默认为 options ）选择要解释哪一部分的psql：
commands 描述psql的反斜线命令； options 描述可以被传递给psql的命令行选项；而 variables
则显示有关psql配置变量的帮助。

退出状态

如果psql正常完成，它会向 shell 返回 0。如果它自身发生一个致命错误（例如内存用完、找不到文件），它会返回 1。如果到服务器的连接出问题并且事务不是交互式的，它会返回 2。如果在脚本中发生错误，它会返回 3 并且变量 `ON_ERROR_STOP` 会被设置。

环境

- `COLUMNS`

如果 `\pset columns` 为零，这个环境变量控制用于 `wrapped` 格式的宽度以及用来确定是否输出需要用到分页器或者切换到扩展自动模式中的垂直格式的宽度。

- `PGDATABASE PGHOST PGPORT PGUSER`

默认连接参数。

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

- `PSQL_EDITOR EDITOR VISUAL`

`\e`、 `\ef` 以及 `\ev` 命令所使用的编辑器。会按照列出的顺序检查这些变量，第一个被设置的将被使用。如果都没有被设置，默认是使用Unix系统上的 `vi` 或者Windows系统上的 `notepad.exe`。

- `PSQL_EDITOR_LINENUMBER_ARG`

当 `\e`、 `\ef` 或者 `\ev` 带有一个行号参数时，这个变量指定用于传递起始行号给用户编辑器的命令行参数。对于Emacs或者vi之类的编辑器，这个变量是一个加号。如果需要在选项名称和行号之间有空格，可以在该变量的值中包括一个结尾的空格。例如： `PSQL_EDITOR_LINENUMBER_ARG=''` `PSQL_EDITOR_LINENUMBER_ARG='--line '` `` 在 Unix 系统上默认是 `` （对应于默认编辑器 `vi` ，且对很多其他常见编辑器可用）。在 Windows 系统上没有默认值。

- `PSQL_HISTORY`

命令历史文件的替代位置。波浪线（`~`）扩展会被执行。

- `PSQL_PAGER PAGER`

如果一个查询的结果在屏幕上放不下，它们会通过这个命令分页显示。典型的值是 `more` 或 `less`。通过把 `PSQL_PAGER` 或 `PAGER` 设置为空字符串可以禁用分页器的使用，调整 `\pset` 命令与分页器相关的选项也能达到同样的效果。会按照列出的顺序检查这些变量，第一个被设置的将被使用。如果都没有被设置，则大部分平台上默认使用 `more`，但在Cygwin上使用 `less`。

- `PSQLRC`

用户的 `.psqlrc` 文件的替代位置。波浪线（`~`）扩展会被执行。

- `SHELL`

被 `\!` 命令执行的命令。

- `TMPDIR`

存储临时文件的目录。默认是 `/tmp` 。

和大部分其他IvorySQL工具一样，这个工具也使用libpq所支持的环境变量。

文件

- psqlrc and ~/.psqlrc

如果没有 -X 选项，在连接到数据库后但在接收正常的命令之前，psql会尝试依次从系统级的启动文件（psqlrc）和用户的个人启动文件（~/.psqlrc）中读取并且执行命令。这些文件可以被用来设置客户端或者服务器，通常是一些 \set 和 SET 命令。系统级的启动文件是 psqlrc，它应该在安装好的IvorySQL的“系统配置”目录中，最可靠的定位方法是运行 pg_config --sysconfdir。默认情况下，这个目录将是 ../etc/（相对于包含IvorySQL可执行文件的目录）。可以通过 PGSYSCONFDIR 环境变量显式地设置这个目录的名称。用户个人的启动文件是 .psqlrc，它应该在调用用户的主目录中。在 Windows 上，由于没有用户主目录的概念，个人的启动文件是 %APPDATA%\postgresql\psqlrc.conf。用户启动文件的位置可以通过 PSQLRC 环境变量设置。系统级和用户个人的启动文件都可以弄成是针对特定psql版本的，方法是在文件名后面加上一个横线以及IvorySQL的主、次版本号，版本最为匹配的文件会优先于不那么匹配的文件读入。

- .psql_history

命令行历史被存储在文件 ~/.psql_history 中，或者是 Windows 的文件 %APPDATA%\postgresql\psql_history 中。历史文件的位置可以通过 HISTFILE psql变量或者 PSQL_HISTORY 环境变量明确的设置。

注解

- psql和具有相同主版本或者更老的主版本服务器最为匹配。如果服务器的版本比psql本身要高，则反斜线命令尤其容易失败。运行 SQL 命令并且显示查询结果的一般功能应该也能和具有更新主版本的服务器一起使用，但是并非在所有的情况下都能保证如此。

如果你想用psql连接到多个具有不同主版本的服务器，推荐使用最新版本的psql。或者，你可以为每一个主版本保留一份psql拷贝，并且针对相应的服务器使用匹配的版本。但实际上，这种额外的麻烦是不必要的。

示例

第一个例子展示了如何如何跨越多行输入一个命令。注意提示符的改变：

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

现在再看看表定义：

```
testdb=> \d my_table
              Table "public.my_table"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 first  | integer |           | not null | 0
 second | text    |           |          |
```

现在我们把提示符改一改：

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

假定已经用数据填充了这个表并且想看看其中的数据：

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

你可以用 `\pset` 命令以不同的方式显示表：

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
```

```
peter@localhost testdb=> \pset format csv
Output format is csv.
peter@localhost testdb=> \pset tuples_only
Tuples only is on.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep '\t'
Field separator is "    ".
peter@localhost testdb=> SELECT second, first FROM my_table;
one     1
two     2
three   3
four    4
```

或者使用短命令：

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

此外，可以使用 \g 为一个查询设置这些输出格式选项：

```
peter@localhost testdb=> SELECT * FROM my_table
```

```
peter@localhost testdb-> \g (format=aligned tuples_only=off expanded=on)
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

这是一个示例，用 `\df` 命令以发现名称匹配 `int*pl` 并且第二参数是 `bigint` 类型的函数:

```
testdb=> \df int*pl * bigint
                          List of functions
   Schema   |  Name   | Result data type | Argument data types | Type
------------+---------+------------------+---------------------+------
 pg_catalog | int28pl | bigint           | smallint, bigint    | func
 pg_catalog | int48pl | bigint           | integer, bigint     | func
 pg_catalog | int8pl  | bigint           | bigint, bigint      | func
(3 rows)
```

如果需要，可以用 `\crosstabview` 命令以交叉表的形式显示查询结果:

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
-------+--------+-----
     1 | one    | f
     2 | two    | f
     3 | three  | t
     4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
     1 | f   |     |       |
     2 |     | f   |       |
     3 |     |     | t     |
```

```
     4 |     |     |       | t
(4 rows)
```

这第二个例子展示了表的“乘法”（连接），行按照序号降序排序且列按照独立的、升序的方式排序。

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as
"AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — 重索引一个IvorySQL数据库

大纲

reindexdb [connection-option…] [option…] [-S | --schema schema] … [-t | --table table] … [-i | --index index] … [dbname]

```
reindexdb` [*`connection-option`*...] [*`option`*...] `-a` | `--all
```

reindexdb [connection-option…] [option…] -s | --system [dbname]

选项

reindexdb接受下列命令行参数：

- -a --all

  重索引所有数据库。

- --concurrently

  使用 `CONCURRENTLY` 选项。 请参阅 http://www.postgresql.org/docs/17/sql-reindex.html[REINDEX]，其中详细解释了此选项的所有注意事项。

- [-d] dbname dbname

当 -a / --all 未使用时，指定要重新索引的数据库的名称。 如果未指定，则从环境变量 PGDATABASE 中读取数据库名称。 如果未设置，则使用为连接指定的用户名。 dbname 可以是 连接字符串。 如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。

- -e --echo

回显reindexdb生成并发送到服务器的命令。

- -i index --index=index

> 只是重建 *`index`*。可以通过写多个 `-i` 开关来重建多个索引。

- -j njobs --jobs=njobs

通过同时运行 njobs 命令并行执行 reindex 命令。
此选项可能会减少处理时间，但也会增加数据库服务器上的负载。reindexdb将打开到数据库的 njobs 连接，因此请确保 max_connections 设置足够高，可以容纳所有连接。请注意，此选项与 --index 和 --system 选项不兼容。

- -q --quiet

不显示进度消息。

- -s --system

仅索引数据库的系统目录。

- -S schema --schema=schema

只对 schema 重建索引。 通过写多个 -S 开关可以指定多个要重建索引的模式。

- -t table --table=table

只索引 table。可以通过写多个 -t 开关来重索引多个表。

- --tablespace=tablespace

指定要重建索引的表空间。 （这个名字以双引号标识符来处理。）

- -v --verbose

在处理时打印详细信息。

- -V --version

打印reindexdb版本并退出。

- -? --help

显示有关reindexdb命令行参数的帮助并退出。

reindexdb也接受下列命令行参数用于连接参数：

- -h host --host=host

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

- -p port --port=port

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制reindexdb在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，reindexdb将自动提示要求一个口令。但是，reindexdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

- `--maintenance-db=dbname`

当使用 `-a` / `--all` 时，指定要连接到的数据库名称以发现应重新索引哪些数据库。 如果未指定，将使用 `postgres` 数据库， 如果不存在，将使用 `template1` 。 这可以是 连接字符串。
如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。
此外，在连接到其他数据库时，将重新使用除数据库名称本身之外的连接字符串参数。

环境

- `PGDATABASE PGHOST PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 REINDEX和 psql
中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

注解

reindexdb可能需要多次连接到IvorySQL服务器，每一次都会询问一个口令。在这种情况下使用一个 `~/.pgpass` 文件会更方便。

例子

要重索引数据库 `test`:

```
$ reindexdb test
```

要重索引名为 `abcd` 的数据库中的表 `foo` 和索引 `bar`:

```
$ reindexdb --table foo --index bar abcd
```

vacuumdb

vacuumdb — 对一个IvorySQL数据库进行垃圾收集和分析

大纲

vacuumdb [**connection-option**⋯] [**option**⋯] [**-t** | **--table table** [(**column** [,⋯])]] ⋯ [**dbname**]

```
vacuumdb` [*`connection-option`*...] [*`option`*...] `-a` | `--all
```

选项

vacuumdb接受下列命令行参数：

- **-a --all**

清理所有数据库。

- **[-d] dbname dbname**

指定要清理或分析的数据库的名称，当不使用 **-a** / **--all`时。 如果未指定，则从环境变量 `PGDATABASE** 中读取数据库名称。 如果环境变量也没有设置，指定给该连接的用户名将用作数据库名。 **dbname** 可以是 connection string。 如果是这样，连接时的字符串参数将覆盖任何冲突的命令行选项。

- **--disable-page-skipping**

根据可见性地图的内容禁用跳过页面。

- **-e --echo**

回显vacuumdb生成并发送给服务器的命令。

- **-f --full**

执行“完全”清理。

- **-F --freeze**

强有力地“冻结”元组。

- **--force-index-cleanup**

总是移除指向死元组的索引条目。

- **-j njobs --jobs=njobs**

通过同时运行 **njobs** 个命令来并行执行清理或者分析命令。这个选项可能会减少处理的时间，但是它也会增加数据库服务器的负载。vacuumdb将开启 **njobs** 个到数据库的连接，因此请确认你的 max_connections 设置足够高以容纳所有的连接。注意如果某些系统目录被并行处理，使用这种模式加上 **-f**（**FULL**）选项可能会导致死锁失败。

- **--min-mxid-age mxid_age**

仅在multixact ID 年龄至少为 **mxid_age** 的表上执行清空或分析命令。此设置对于确定要处理的表的优先级比较有用，以防止multixact ID 回绕。对于此选项的用途，关系的multixact ID年龄是主关系及其关联的TOAST表的年龄中最大的，如果存在的话。由于vacuumdb发出的命令在需要时还将处理TOAST表的关系，它无需单独考虑。

- **--min-xid-age xid_age**

仅在事务ID 年龄至少为 `xid_age` 的表上执行清空或分析命令。
此设置对于确定要处理的表的优先级比较有用，以防止事务ID
回绕。对于此选项的用途，关系的事务ID年龄是主关系及其关联的TOAST表的年龄中最大的，如果存在的话
。由于vacuumdb发出的命令在需要时还将处理TOAST表的关系，它无需单独考虑。

- `--no-index-cleanup`

不移除指向死元组的索引条目。

- `--no-process-toast`

略过与要清理的表有关联的任何 TOAST 表。

- `--no-truncate`

不要在表的结尾截断空页面。

- `-P parallel_workers --parallel=parallel_workers`

指定 parallel vacuum 的并发数量。这允许清理利用多个 CPU 来处理索引。请参见 VACUUM。

- `-q --quiet`

不显示进度消息。

- `--skip-locked`

跳过无法立即锁定以进行处理的关系。

- `-t` `table [ (column [,…]) ]` `--table=` `table [ (column [,…]) ]`

只清理或分析 `table`。列名只能和 `--analyze` 或 `--analyze-only` 选项一起被指定。通过写多个 `-t`
开关可以清理多个表。提示如果你指定列，你可能必须转义来自 shell 的括号（见下面的例子）。

- `-v --verbose`

在处理期间打印详细信息。

- `-V --version`

打印vacuumdb版本并退出。

- `-z --analyze`

也计算优化器使用的统计信息。

- `-Z --analyze-only`

只计算优化器使用的统计信息（不清理）。

- `--analyze-in-stages`

与 `--analyze-only` 相似，只计算优化器使用的统计信息（不做清理）。
使用不同的配置设置运行分析的几个（目前是
3个）阶段以更快地产生可用的统计信息。这个选项对分析一个刚从转储恢复或者通过 `pg_upgrade`
得到的数据库有用。这个选项将尝试尽可能快地创建一些统计信息来让该数据库可用，然后在后续的阶段中
产生完整的统计信息。

- `-? --help`

显示有关vacuumdb命令行参数的帮助并退出。

vacuumdb也接受下列命令行参数用于连接参数：

- `-h host --host=host`

指定运行服务器的机器的主机名。如果该值以一个斜线开始，它被用作 Unix 域套接字的目录。

- `-p port --port=port`

指定服务器正在监听连接的 TCP 端口或本地 Unix 域套接字文件扩展。

- `-U username --username=username`

要作为哪个用户连接。

- `-w --no-password`

从不发出一个口令提示。如果服务器要求口令认证并且没有其他方式提供口令（例如一个 `.pgpass` 文件），那儿连接尝试将失败。这个选项对于批处理任务和脚本有用，因为在其中没有一个用户来输入口令。

- `-W --password`

强制vacuumdb在连接到一个数据库之前提示要求一个口令。这个选项不是必不可少的，因为如果服务器要求口令认证，vacuumdb将自动提示要求一个口令。但是，vacuumdb将浪费一次连接尝试来发现服务器想要一个口令。在某些情况下值得用 `-W` 来避免额外的连接尝试。

- `--maintenance-db=dbname`

当使用 `-a` / `--all` 时，指定要连接到的数据库名称以发现应该清理的数据库。 如果未指定，将使用 `postgres` 数据库，如果不存在，将使用 `template1`。 这可以是 connection string。
如果是这样，连接字符串参数将覆盖任何冲突的命令行选项。
此外，在连接到其他数据库时，将重新使用除数据库名称本身之外的连接字符串参数。

环境

- `PGDATABASE PGHOST PGPORT PGUSER`

默认连接参数

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto` 和 `never`。

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

诊断

在有困难时，可以在 VACUUM 和 psql
中找潜在问题和错误消息的讨论。数据库服务器必须运行在目标主机上。同样，任何libpq前端库使用的默认连接设置和环境变量都将适用于此。

注解

vacuumdb可能需要多次连接到IvorySQL服务器，每次都询问一个口令。在这种情况下有一个 `~/.pgpass` 文件会很方便。

例子

要清理数据库 `test`：

```
$ vacuumdb test
```

要清理和为优化器分析一个名为 bigdb 的数据库：

```
$ vacuumdb --analyze bigdb
```

要清理在名为 xyzzy 的数据库中的一个表 foo，并且为优化器分析该表的 bar 列：

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

服务器应用

initdb

initdb — 创建一个新的IvorySQL数据库集簇

大纲

initdb [option…] [--pgdata | -D] ` directory`

选项

- -A authmethod --auth=authmethod

这个选项为本地用户指定在 pg_hba.conf 中使用的默认认证方法（ host 和`local` 行）。 initdb 将使用指定的认证方法为非复制连接以及复制连接填充 pg_hba.conf 项。除非你信任你系统上的所有本地用户，不要使用 trust。为了安装方便，trust 是默认值。

- --auth-host=authmethod

这个选项为通过 TCP/IP 连接的本地用户指定在 pg_hba.conf 中使用的认证方法（ host 行）。

- --auth-local=authmethod

这个选项为通过 Unix 域套接字连接的本地用户指定在 pg_hba.conf 中使用的认证方法（ local 行）。

- -D directory --pgdata=directory

这个选项指定数据库集簇应该存放的目录。这是 initdb 要求的唯一信息，但是你可以通过设定 PGDATA 环境变量来避免书写它，这很方便因为之后数据库服务器（ postgres ）可以使用同一个变量来找到数据库目录。

- -E encoding --encoding=encoding

选择模板数据库的编码。这也将是后来创建的任何数据库的默认编码，除非你覆盖它。默认值来自于区域，或者如果该值不起作用则为 SQL_ASCII。

- -g --allow-group-access

允许与集簇拥有者同组的用户读取 initdb 创建的所有集簇文件。
Windows会忽略此选项，因为它不支持POSIX样式的组权限。

- -k --data-checksums

在数据页面上使用校验码来帮助检测 I/O 系统造成的损坏。 启用校验码将会引起显著的性能惩罚。

如果设置，则为所有对象计算校验和，在整个数据库中。 所有校验和失败都将报告在 `pg_stat_database` 视图。

- `--locale=locale`

为数据库集簇设置默认区域。如果这个选项没有被指定，该区域将从 `initdb` 所运行的环境中继承。

- `--lc-collate=locale` `--lc-ctype=locale` `--lc-messages=locale` `--lc-monetary=locale` `--lc-numeric=locale` `--lc-time=locale`

和 `--locale` 相似，但是只在指定的分类中设置区域。

- `--no-locale`

等效于 `--locale=C`。

- `-N` `--no-sync`

默认情况下，`initdb` 将等待所有文件被安全地写到磁盘。这个选项会导致 `initdb` 不等待就返回，这当然更快，但是也意味着一次后续的操作系统崩溃可能让数据目录损坏。通常，这个选项对测试有用，但是不应该在创建生产安装时使用。

- `--no-instructions`

默认情况下，`initdb` 将在输出的尾部写入如何启动集群的指令。 这个选项会导致那些指令被抛弃。这主要由平台特定行为中封装 `initdb` 的工具来使用，那些指令有可能是错误的。

- `--pwfile=filename`

让 `initdb` 从一个文件读取数据库超级用户的口令。该文件的第一行被当作口令。

- `-S` `--sync-only`

安全地把所有数据库文件写入到磁盘并退出。这不会执行任何正常的initdb操作。

- `-T config` `--text-search-config=config`

设置默认的文本搜索配置。详见 default_text_search_config。

- `-U username` `--username=username`

选择数据库超级用户的用户名。这个的默认值是实际运行 `initdb` 的用户的名称。超级用户的名字是什么真的不重要，但是你可以选择保留常用的名字postgres，即使操作系统的用户名不同。

- `-W` `--pwprompt`

让 `initdb` 提示要求为数据库超级用户给予一个口令。如果你没有计划使用口令认证，这就不重要。否则在你设置一个口令之前你就无法使用口令认证。

- `-X directory` `--waldir=directory`

这个选项指定预写式日志会被存储在哪个目录中。

- `--wal-segsize=`size`

设置WAL段尺寸，以兆字节为单位。这是WAL日志中每个文件的尺寸。默认的尺寸为16兆字节。该值必须位于2的1次幂和1024次幂（兆字节）之间。这个选项只能在初始化期间设置，并且之后不能更改。调整这个值来控制WAL日志传送或者归档可能会有用。此外，在有大量WAL的数据库中，每个目录中数量巨大的WAL文件可能会成为性能和管理问题。增加WAL文件尺寸将会降低WAL文件的数量。

其他较少使用的选项：

- `-d --debug`

打印来自引导后端的调试输出以及普通大众不那么感兴趣的一些消息。引导后端被程序 `initdb` 用来创建目录表。这个选项会生成大量极端无聊的输出。

- `--discard-caches`

使用 `debug_discard_caches=1` 选项运行引导程序后端。 这需要很长时间并且只适用于深度调试。

- `-L directory`

指定 `initdb` 应从哪里寻找它的输入文件来初始化数据库集簇。这通常没有必要。如果你需要显式指定它们的位置，你应该被告知。

- `-n --no-clean`

默认情况下，当 `initdb` 确定有一个错误阻止它完整地创建数据库集簇，它会移除在它发现无法完成任务之前创建的任何文件。这个选项会抑制这种整理并且对调试有用。

其他选项：

- `-V --version`

打印initdb版本并退出。

- `-? --help`

显示有关initdb命令行参数的帮助并退出。

环境

- `PGDATA`

指定数据库集簇应该被存放的目录，可以使用 `-D` 选项覆盖。

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

- `TZ`

指定创建的数据集簇的默认时区。值应该是一个完整的时区名称

和大部分其他IvorySQL工具相似，这个工具也使用libpq支持的环境变量。

注解

`initdb` 可以通过 `pg_ctl initdb` 被调用。

pg_archivecleanup

pg_archivecleanup — 清理IvorySQL WAL 归档文件

大纲

`pg_archivecleanup` [`option`…] `archivelocation oldestkeptwalfile`

选项

pg_archivecleanup接受下列命令行参数：

- `-d`

在 `stderr` 上打印很多调试日志输出。

- `-n`

在 `stdout` 上打印将被移除的文件的名字（执行一次演习）。

- `-V --version`

打印pg_archivecleanup版本并退出。

- `-x extension`

提供一个扩展名，在决定所有的文件
是否应该被删除之前，将从文件名中剥离这个扩展名。这通常有助于清理已经
存储期间被压缩过并且被压缩程序增加了一个扩展名的归档。例如： `-x .gz`。

- `-? --help`

显示pg_archivecleanup命令行参数的帮助并退出。

环境

环境变量 `PG_COLOR` 指定是否在诊断消息中使用颜色。 可能的值是 `always`, `auto` 和 `never`。

示例

在 Linux 或者 Unix 系统上，你可能会用：

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r
2>>cleanup.log'
```

其中归档目录位于后备服务器上，这样 `archive_command` 通过 NFS
来访问它，但是文件对于后备服务器来说是本地的。这将会

- 在 `cleanup.log` 中产生调试输出
- 从归档目录中移除不再需要的文件

pg_checksums

pg_checksums — 在IvorySQL数据库集簇中启用、禁用或检查数据校验和

大纲

`pg_checksums` [`option`…] [[`-D` | `--pgdata`] `datadir`]

选项

下列命令选项可用：

- `-D directory --pgdata=directory`

指定存储数据库集簇的目录。

- `-c --check`

检查校验和。如果未指定其它任何内容，这是缺省模式。

- `-d --disable`

禁用校验和。

- `-e --enable`

启用校验和。

- `-f filenode --filenode=filenode`

仅验证文件节点为 `filenode` 的关系中的校验和。

- `-N --no-sync`

缺省地，`pg_checksums` 会等待所有文件安全地写到磁盘上。该选项使得 `pg_checksums` 不等待就返回，这样更快，但意味着后续如果操作系统崩溃会让更新的数据目录损坏。一般地，该选项对测试有用，但不应用在生产安装上。当使用 `--check` 时，该选项无效。

- `-P --progress`

启用进度报告。在检查或启用校验和时，打开该选项，会提供进度报告。

- `-v --verbose`

启用详细输出。列出所有检查的文件 。

- `-V --version`

打印pg_checksums版本并退出。

- `-? --help`

显示关于pg_checksums命令行参数的帮助并退出。

环境

- `PGDATA`

指定数据库集簇存储的目录；可以用 `-D` 选项覆盖。

- `PG_COLOR`

指定是否在诊断消息中使用颜色。可能的值为 `always`, `auto`, `never`。

注意

在大型集簇中启用校验和的时间可能很长。在此操作期间，写到数据目录的集簇或其它程序必须是未启动的，否则可能出现数据丢失。

当复制设置与执行关系文件块的直接拷贝的工具（例如 pg_rewind）一起使用时，启用和禁用校验和会导致以不正确校验和形式出现的页面损坏，如果未在所有节点上执行一致的操作的话。故在复制设置中启用或禁用校验和时，推荐一致地切换所有集簇之前停止所有集簇。此外销毁所有备用数据库，在主数据库上执行操作，最后从头开始重建后备服务器，也是安全的。

如果在启用或禁用校验和时异常终止或杀掉pg_checksums，那么集簇的数据校验和配置保持不变，pg_checksums可以重新运行以执行相同操作。

pg_controldata

pg_controldata — 显示一个IvorySQL数据库集簇的控制信息

大纲

`pg_controldata` [`option`] [[`-D` | `--pgdata`] `datadir`]

环境

- `PGDATA`

默认的数据目录位置。

- `PG_COLOR`

规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

pg_ctl

pg_ctl — 初始化、启动、停止或控制一个IvorySQL服务器

大纲

`pg_ctl init[db]` [`-D datadir`] [`-s`] [`-o initdb-options`]

`pg_ctl start` [`-D datadir`] [`-l filename`] [`-W`] [`-t seconds`] [`-s`] [`-o options`] [`-p path`] [`-c`]

`pg_ctl stop` [`-D datadir`] [`-m s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t seconds`] [`-s`]

`pg_ctl restart` [`-D datadir`] [`-m s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t seconds`] [`-s`] [`-o options`] [`-c`]

`pg_ctl reload` [`-D datadir`] [`-s`]

`pg_ctl status` [`-D datadir`]

`pg_ctl promote` [`-D datadir`] [`-W`] [`-t seconds`] [`-s`]

`pg_ctl logrotate` [`-D datadir`] [`-s`]

`pg_ctl kill signal_name process_id`

在Microsoft Windows上，还有：

`pg_ctl register` [`-D datadir`] [`-N servicename`] [`-U username`] [`-P password`] [`-S a[uto]` | `d[emand]`] [`-e source`] [`-W`] [`-t seconds`] [`-s`] [`-o options`]

`pg_ctl unregister` [`-N servicename`]

选项

- `-c --core-files`

  > 在可行的平台上尝试允许服务器崩溃产生核心文件，方法是提升在核心文件上的任何软性资源限制。这通过允许从一个失败的服务器进程中获得一个栈跟踪而有助于调试或诊断问题。

- `-D` `datadir` `--pgdata=` `datadir`

指定数据库配置文件的文件系统位置。如果这个选项被忽略，将使用环境变量`PGDATA`。

- `-l `filename` --log=`filename``

追加服务器日志输出到*`filename`*。如果该文件不存在，它会被创建。umask被设置成077，这样默认情况下不允许其他用户访问该日志文件。

- `-m mode --mode=mode`

指定关闭模式。`mode` 可以是 `smart`、`fast` 或 `immediate`，或者这三者之一的第一个字母。如果这个选项被忽略，则 `fast` 是默认值。

- `-o options --options=options`

指定被直接传递给 `postgres` 命令的选项。`-o` 可以被指定多次，所有给定的选项都会被传过去。这些选项应该通常被单引号或双引号包围来确保它们被作为一个组传递。

- `-o initdb-options --options=initdb-options`

指定要直接传递给 `initdb` 命令的选项。`-o` 可以被指定多次，所有给定的选项都会被传过去。这些选项应该通常被单引号或双引号包围来确保它们被作为一个组传递。

- `-p path`

指定 `postgres` 可执行程序的位置。默认情况下，`postgres` 可执行程序可以从 `pg_ctl` 相同的目录得到，或者如果没有在那里找到，则在硬写的安装目录中获得。除非你正在做一些不同寻常的事并且得到错误说没有找到 `postgres` 可执行程序，这个选项不是必需的。在 `init` 模式中，这个选项类似于指定了 `initdb` 可执行程序的位置。

- `-s --silent`

只打印错误，不打印信息性的消息。

- `-t seconds --timeout=seconds`

指定等待一个操作完成时要等待的最大秒数（见选项 `-w`）。默认为 `PGCTLTIMEOUT` 环境变量的值，如果该环境变量没有设置则默认为60秒。

- `-V --version`

打印pg_ctl版本并退出。

- `-w --wait`

等待操作完成。模式 `start`、 `stop`、 `restart`、 `promote` 以及 `register` 支持这个选项，并且对那些模式是默认的。在等待时，`pg_ctl` 会一遍又一遍地检查服务器的PID文件，在两次检查之间会休眠一小段时间。当PID文件指示该服务器已经做好准备接受连接时，启动操作被认为完成。当服务器移除PID文件时，关闭操作被认为完成。`pg_ctl` 会基于启动或关闭的成功与否返回一个退出代码。如果操作在超时时间（见选项 `-t`）内未能完成，则 `pg_ctl` 会以一个非零退出状态退出。但是注意该操作可能会在后台继续进行并且最终取得成功。

- `-W --no-wait`

不等待操作完成。这是选项 `-w` 的对立面。如果禁用等待，所请求的动作会被触发，但是不会有关于其成功与否的反馈。在这种情况下，可能必须用服务器日志文件或外部监控系统来检查该操作的进度以及成功与否。在以前版本的IvorySQL中，这是除 `stop` 模式之外的模式的默认选项。

- `-? --help`

显示有关pg_ctl命令行参数的帮助并退出。

如果一个指定的选项有效，但与选中的操作模式无关，则pg_ctl会忽略它。

用于 Windows 的选项

- `-e source`

作为一个 Windows 服务运行时，pg_ctl用来 在事件日志中记录日志的事件源的名称。默认是 `IvorySQL`。注意这只控制由pg_ctl本身发送的消息，一旦开始， 服务器将使用 event_source 参数中指定的事件源。如果服务器在启动时很早（在该参数被设置前）就失败，它可能也会使用默认的事件源名称 `IvorySQL` 来记录。

- `-N servicename`

要注册的系统服务的名称。这个名称将被用于服务名和显示名。默认是 `IvorySQL`。

- `-P password`

用于运行该服务的用户的口令。

- `-S start-type`

要注册的系统服务的启动类型。启动类型可以是 `auto`、 `demand` 或者两者之一的第一个字母。如果这个选项被忽略，则 `auto` 是默认值。

- `-U username`

用于运行该服务的用户的用户名。对于域用户，使用格式 `DOMAIN\username`。

环境

- `PGCTLTIMEOUT`

等待启动或者关闭完成时要等待的默认秒数限制。如果没有设置， 默认值是 60 秒。

- `PGDATA`

默认的数据目录位置。

大部分的 `pg_ctl` 模式都要求知道数据目录的位置，因此 `-D` 选项是必需的，除非 `PGDATA` 被设置。

和大部分其他IvorySQL工具相似，`pg_ctl` 也使用libpq支持的环境变量。

更多影响服务器的变量请见 postgres。

文件

- `postmaster.pid`

pg_ctl在数据目录中检查这个文件来判断服务器当前是否正在运行。

- `postmaster.opts`

如果这个文件存在于数据目录中，pg_ctl（处于 `restart` 模式中）将把该文件的内容作为选项传递给postgres，除非通过 `-o` 选项进行了覆盖。这个文件的内容也会被显示在 `status` 模式中。

例子

启动服务器

要启动服务器并且等到服务器接受连接：

```
$ pg_ctl start
```

要使用端口 5433 启动服务器并且运行时不使用 `fsync`：

```
$ pg_ctl -o "-F -p 5433" start
```

停止服务器

要停止服务器，使用：

```
$ pg_ctl stop
```

`-m` 选项允许控制服务器如何关闭：

```
$ pg_ctl stop -m smart
```

重启服务器

重启服务器几乎等价于停止服务器并且再次启动它，不过 `pg_ctl`
默认会保存并重用被传递给之前的运行实例的命令行选项。要以和之前相同的选项重启服务器，使用：

```
$ pg_ctl restart
```

但是如果指定了 `-o`，则会替换任何之前的选项。要使用端口 5433 重启并在重启时禁用 `fsync`：

```
$ pg_ctl -o "-F -p 5433" restart
```

显示服务器状态

这里是pg_ctl状态输出的例子：

```
$ pg_ctl status

pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

第二行是在重启模式可能被调用的命令行。

pg_resetwal

pg_resetwal — 重置一个IvorySQL数据库集簇的预写式日志以及其他控制信息

大纲

`pg_resetwal` [`-f` | `--force`] [`-n` | `--dry-run`] [`option`…] [`-D` | `--pgdata`] `datadir`

选项

- `-f` `--force`

即使 `pg_resetwal` 无法从 `pg_control` 中确定有效的数据（如前面所解释的），也强迫 `pg_resetwal` 继续运行。

- `-n` `--dry-run`

`-n` / `--dry-run` 选项指示 `pg_resetwal` 打印从 `pg_control` 重构出来的值以及要被改变的值，然后不修改任何东西退出。这主要是一个调试工具，但是可以用来在允许 `pg_resetwal` 真正执行下去之前进行完整性检查。

- `-V` `--version`

显示版本信息然后退出。

- `-?` `--help`

显示帮助然后退出。

只有当 `pg_resetwal` 无法通过读取 `pg_control` 确定合适的值时，才需要下列选项。安全值可以按下文所述来确定。对于接收数字参数的值，可以使用前缀 `0x` 指定 16 进制值。

- `-c xid, xid` `--commit-timestamp-ids=xid, xid`

手工设置提交时间可以检索到的最老的和最新的事务 ID。能检索到提交时间的最老事务 ID 的安全值（第一部分）可以通过在数据目录下 `pg_commit_ts` 目录中数字上最小的文件名来决定。反过来，能检索到提交时间的最新事务 ID 的安全值（第二部分）可以通过同一个目录中数字上最大的文件名来决定。文件名都是十六进制的。

- `-e xid_epoch` `--epoch=xid_epoch`

手工设置下一个事务 ID 的 epoch。事务 ID 的 epoch 实际上并没有存储在数据库中的任何地方，除了被 `pg_resetwal` 设置在这个域中，所以只要关心的是数据库本身，任何值都可以用。你可能需要调整这个值来确保诸如Slony-I和Skytools之类的复制系统正确地工作 — 如果确实需要调整，应该可以从下游的复制数据库的状态中获得一个合适的值。

- `-l` `` `walfile` `` `--next-wal-file=` `` `walfile` ``

通过指定下一个WAL段文件名称来手工设置WAL开始位置。下一个WAL段文件的名称应该比当前存在于数据目录下 `pg_wal` 目录中的任意 WAL 段文件名更大。这些名称也是十六进制的并且有三个部分。第一部分是“时间线 ID”并且通常应该被保持相同。例如，如果 `000000010000000320000004A` 是 `pg_wal` 中最大的项，则使用 `-l` `000000010000000320000004B` 或更高的值。注意在使用非默认WAL段尺寸时，WAL文件名中的数字与系统函数和系统视图报告的LSN不同。这个选项要的是WAL文件名而不是LSN。注意 `pg_resetwal` 本身查看 `pg_wal` 中的文件并选择一个超出最新现存文件名的默认 `-l` 设置。因此，只有当你知道 WAL 段文件当前不在 `pg_wal` 中时，或者当 `pg_wal` 的内容完全丢失时，才需要对 `-l` 的手工调整，例如一个离线归档中的项。

- `-m mxid,mxid` `--multixact-ids=mxid,mxid`

手工设置下一个和最老的多事务 ID。确定下一个多事务 ID（第一部分）的安全值的方法：在数据目录下的 pg_multixact/offsets 目录中查找最大的数字文件名，然后在它的基础上加一并且乘以 65536 (0x10000)。反过来，确定最老的多事务 ID（ -m 的第二部分）的方法：在同一个目录中查找最小的数字文件名并且乘以 65536。文件名是十六进制的数字，因此实现上述方法最简单的方式是以十六进制指定选项值并且追加四个零。

- -o oid --next-oid=oid

手工设置下一个 OID。没有相对容易的方法来决定超过数据库中最大 OID 的下一个 OID。但幸运的是正确地得到下一个 OID 设置并不是决定性的。

- -O mxoff --multixact-offset=mxoff

手工设置下一个多事务偏移量。确定安全值的方法：查找数据目录下 pg_multixact/members 目录中最大的数字文件名，然后在它的基础上加一并且乘以 52352 (0xCC80)。文件名是十六进制数字。没有像其他选项那样追加零的简单方法。

- --wal-segsize=wal_segment_size

设置新的WAL段尺寸，以兆字节为单位。这个值必须被设为2的1次幂和1024次幂（兆字节）之间。更多信息请参考 initdb 的相同选项。注意虽然 pg_resetwal 将把WAL起始地址设置成超过最新的现有WAL段文件，但一些段尺寸的改变可能导致之前的WAL文件名被重用。如果WAL文件名重叠会导致归档策略出现问题，推荐把 -l 和这个选项一起使用来手动设置WAL起始地址。

- -u xid --oldest-transaction-id=xid

手工设置最老的未冻结事务ID。一个安全值，可以通过在数据目录下的 pg_xact 目录中查找数字最小的文件名然后乘以1048576 (0x100000)的方式来确定。 注意该文件名是十六进制的。 以十六进制来指定选项值通常也是最简单的。 例如,如果 0007 是 pg_xact 中最小的记录， -u 0x700000 将有效(五个后补零提供适当的乘数)。

- -x xid --next-transaction-id=xid

手工设置下一个事务 ID。确定安全值的方法：在数据目录下的 pg_xact 目录中查找最大的数字文件名，然后在它的基础上加一并且乘以 1048576 (0x100000)。注意文件名是十六进制的数字。通常以十六进制的形式指定该选项值也是最容易的。例如，如果 0011 是 pg_xact 中的最大项， -x 0x1200000 就可以（五个尾部的零就表示了前面说的乘数）。

环境

- PG_COLOR

规定在诊断消息中是否使用颜色。可能的值为 always、 auto、 never。

注解

这个命令不能在服务器正在运行时被使用。如果在数据目录中发现一个服务器锁文件，pg_resetwal 将拒绝启动。如果服务器崩溃那么一个锁文件可能会被留下，在那种情况下你能移除该锁文件来让 pg_resetwal 运行。但是在你那样做之前，再次确认没有服务器进程仍然存活。

pg_resetwal 仅能在具有相同主版本的服务器上使用。

pg_rewind

pg_rewind — 把一个IvorySQL数据目录与另一个从该目录中复制出来的数据目录同步

大纲

pg_rewind [option…] { -D | --target-pgdata }*` directory`* { --source-pgdata=`directory` | --source

`-server=`connstr`` }

警告

如果在处理时pg_rewind失败，则目标的数据目录很可能不在可恢复的状态。在这种情况下，推荐创建一个新的备份。

由于 pg_rewind
完全从源复制配置文件，因此可能需要在重新启动目标服务器之前更正用于恢复的配置，特别是当目标服务器作为源的后备服务器重新引入时。
如果在倒带操作完成后重新启动服务器但未配置恢复，则目标可能会再次与主服务器分离。

如果pg_rewind发现它无法直接写入的文件，它将立刻失败。例如当源服务器和目标服务器为只读的SSL密钥及证书使用相同的文件映射，就会发生这种情况。如果在目标服务器上存在这样的文件，推荐在运行pg_rewind之前移除它们。在做了rewind之后，一些那样的文件可能已经被从源服务器拷贝，这样就有必要移除已经拷贝的数据并且恢复到rewind之前使用的链接集合。

选项

pg_rewind接受下列命令行参数：

- `-D directory` `--target-pgdata=directory`

这个选项指定要与源数据目录同步的目标数据目录。在运行pg_rewind之前目标服务器必须被干净地关闭。

- `--source-pgdata=directory`

指定要和目标服务器同步的源服务器的数据目录的文件系统路径。这个选项要求源服务器必须被干净地关闭。

- `--source-server=`connstr``

指定一个 libpq 连接串用于连接要与目标服务器同步的源IvorySQL服务器。
连接必须是常规（非复制）连接，角色具有足够权限执行源服务器上pg_rewind使用的函数（详请参阅备注部分）或超级用户角色。 这个选项要求源服务器正在运行并且接受连接。

- `-R` `--write-recovery-conf`

创建 `standby.signal` 并将连接设置附加到输出目录中的 `postgresql.auto.conf` 中。`--source-server` 对于此选项是必需的。

- `-n` `--dry-run`

做除了实际修改目标目录之外的其他所有事情。

- `-N` `--no-sync`

默认情况下，`pg_rewind` 将等待所有文件安全地写入磁盘。 此选项会导致 `pg_rewind` 不等待即可返回，这更快，但意味着后续操作系统崩溃会使同步数据目录损坏。通常情况，此选项可用于测试，但不应使用于生产安装。

- `-P` `--progress`

启用进度报告。在从源集簇拷贝数据时，打开这个选项将会发送一个近似的进度报告。

- `-c` `--restore-target-wal`

如果在 `pg_wal` 目录中不再可用这些文件，请使用在目标群集配置中定义的 `restore_command` 从WAL存档中检索WAL文件。

- `--debug`

打印冗长的调试输出，这主要对于调试pg_rewind的开发者有用。

- `--no-ensure-shutdown`

pg_rewind 要求目标服务器在倒带之前彻底关闭。 默认情况下，如果目标服务器没有完全关闭，pg_rewind 会以单用户模式启动目标服务器，先完成崩溃恢复，然后将其停止。
通过传递这个选项，如果服务器没有完全关闭，pg_rewind 会跳过这个并立即出错。
在这种情况下，用户应该自己处理这种情况。

- `-V --version`

显示版本信息然后退出。

- `-? --help`

显示帮助然后退出。

环境

在使用 `--source-server` 选项时，pg_rewind也使用libpq支持的环境变量。

环境变量`PG_COLOR`规定在诊断消息中是否使用颜色。可能的值为 `always`、 `auto`、 `never`。

注解

当使用在线群集作为源执行pg_rewind时，具有充足权限来执行pg_rewind在源群集上使用的函数的角色可以用来代替超级用户。 这里介绍如何创建这样的角色，在这里命名 `rewind_user`：

```
CREATE USER rewind_user LOGIN;
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint,
boolean) TO rewind_user;
```

当使用近期升级的在线群集作为源执行pg_rewind时，必须在升级后执行 `CHECKPOINT`
以便其控制文件反映最新的时间线信息，
pg_rewind使用这些信息检查目标群集是否可以使用指定的源群集倒回。

如何工作

其基本思想是从源集簇拷贝所有文件系统级别的改变到目标集簇：

1.以源集簇的时间线历史从目标集簇分叉出来的点之前的最后一个检查点为起点，扫描目标集簇的 WAL
日志。对于每一个 WAL
记录，读取每一个被动过的数据块。这会得到在目标集簇中从源集簇被分支出去以后所有被更改过的数据块列表。如果某些 WAL 文件 不再可用，请尝试使用 `-c` 选项重新运行 pg_rewind 以在 WAL
存档中搜索丢失的文件。 2.使用直接的文件系统访问（`--source-pgdata`）或者 SQL （`--source-server`），把所有那些更改过的块从源集簇拷贝到目标集簇。
关系文件现在的状态相当于源和目标的WAL时间线偏离点之前最后一个完成的检查点的时刻加上偏离点之后目标上更改的任何块的源上的当前状态。 3.将所有其他文件，包括新的关系文件、WAL段、`pg_xact`
和配置文件，从源集群复制到目标集群。与基本备份类似，从源集群复制的数据中省略了目录
`pg_dynshmem/`、 `pg_notify/`、 `pg_replslot/`、 `pg_serial/`、 `pg_snapshots/`、 `pg_stat_tmp/` 以及
`pg_subtrans/` 的内容。文件 `backup_label`、 `tablespace_map`、 `pg_internal.init`、 `postmaster.opts` 以及
`postmaster.pid`，以及任何以 `pgsql_tmp` 开始的文件或目录都会被忽略。 4.创建一个 `backup_label`
文件，在故障转移时创建的检查点处开始WAL重放，并将 `pg_control`
文件配置为最小一致性LSN，该LSN定义为从活动源回放时的 `pg_current_wal_insert_lsn()`
结果，或从停止的源回放时的最后一个检查点LSN。

5.启动目标时，IvorySQL将重放所有必需的WAL，从而使数据目录处于一致状态。

pg_test_fsync

pg_test_fsync — 为IvorySQL判断最快的 `wal_sync_method`

大纲

`pg_test_fsync` [`option`…]

选项

pg_test_fsync接受下列命令行选项：

- `-f --filename`

指定要写入测试数据到其中的文件名。这个文件必须位于和 `pg_wal` 目录所在或者将被放置的同一个文件系统中（`pg_wal` 包含WAL文件）。默认是当前 目录中的 `pg_test_fsync.out`。

- `-s --secs-per-test`

指定每次测试的秒数。每个测试的时间越长，测试的精度就越高，但是 它需要更多时间来运行。默认是 5 秒，这允许程序在 2 分钟以内完成。

- `-V --version`

打印pg_test_fsync版本并且退出。

- `-? --help`

显示有关pg_test_fsync命令行参数的帮助并且退出。

环境

环境变量 `PG_COLOR` 指定是否在诊断消息中使用颜色。可能的值为 `always`、 `auto` 和 `never`.

pg_test_timing

pg_test_timing — 度量计时开销

大纲

`pg_test_timing` [`option`…]

选项

pg_test_timing接受下列命令行选项：

- `-d duration --duration=duration`

指定测试的持续时间，以秒计。更长的持续时间会给出更好一些的精确度，并且更可能发现系统时钟回退的问题。默认的测试持续时间是 3 秒。

- `-V --version`

打印pg_test_timing版本并退出。

- `-? --help`

显示有关pg_test_timing的命令行参数，然后退出。

用法

结果解读

好的结果将显示大部分（>90%）的单个计时调用用时都小于 1 微秒。每次循环的平均开销将会更低，低于 100 纳秒。下面的例子来自于一台使用了一份 TSC 时钟源码的 Intel i7-860 系统，它展示了非常好的性能：

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  &lt; us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

注意每次循环时间和柱状图用的单位是不同的。循环的解析度可以在几个纳秒（ns），而单个计时调用只能解析到一个微秒（us）。

度量执行器计时开销

当查询执行器使用 `EXPLAIN ANALYZE`
运行一个语句时，单个操作会被计时，总结也会被显示。你的系统的负荷可以通过使用psql程序计数行来检查：

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

i7-860 系统测到运行该计数查询用了 9.8 ms 而 `EXPLAIN ANALYZE` 版本则需要 16.6 ms，每次处理都在 100,000 行上进行。6.8 ms 的差别意味着在每行上的计时负荷是 68 ns，大概是 pg_test_timing
估计的两倍。即使这样相对少量的负荷也造成了带有计时的计数语句耗时多出了
70%。在更大量的查询上，计时开销带来的问题不会有这么明显。

改变时间来源Changing time sources

在一些较新的 Linux
系统上，可以在任何时候更改用来收集计时数据的时钟来源。第二个例子显示了在上述快速结果的相同系统上切换到较慢的 acpi_pm 时间源可能带来的降速：

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
```

```
Histogram of timing durations:
  &lt; us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

在这种配置中，上面的例子 `EXPLAIN ANALYZE` 用了 115.9 ms。其中有 1061 ns 的计时开销，还是用这个工具直接度量结果的一个小倍数。这么多的计时开销意味着实际的查询本身只占了时间的一个很小的分数，大部分的时间都耗在了计时所需的管理开销上。在这种配置中，任何涉及到很多计时操作的`EXPLAIN ANALYZE`都会受到计时开销的显著影响。

FreeBSD 也允许即时更改时间源，并且它会记录在启动期间有关计时器选择的信息：

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

其他系统可能只允许在启动时设定时间源。在旧的 Linux 系统上，“clock”内核设置是做这类更改的唯一方法。并且即使在一些更近的系统上，对于一个时钟源你将只能看到唯一的选项 "jiffies"。Jiffies 是老的 Linux 软件时钟实现，当有足够快的计时硬件支持时，它能够具有很好的解析度，就像在这个例子中：

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  &lt; us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

时钟硬件和计时准确性

收集准确的计时信息在计算机上通常是使用具有不同精度的时钟硬件完成的。使用一些硬件，操作系统能几乎直接把系统时钟时间传递给程序。一个系统时钟也可以得自于一块简单地提供计时中断、在某个已知时间区间内的周期性滴答的芯片。在两种情况中，操作系统内核提供一个隐藏这些细节的时钟源。但是时钟源的精确度以及能多快返回结果会根据底层硬件而变化。

不精确的计时能够导致系统不稳定性。对任何时钟源的更改都要仔细地测试。操作系统默认是有时会更倾向于可靠性而不是最好的精确性。并且如果你在使用一个虚拟机器，应查看与之兼容的推荐时间源。在模拟计时器时虚拟硬件面临着额外的困难，并且提供商常常会建议每个操作系统的设置。

时间戳计数器（TSC）时钟源是当前一代 CPU 上最精确的一种。当操作系统支持 TSC 并且 TSC 可靠时，它是跟踪系统时间更好的方式。有多种方式会使 TSC 无法提供准确的计时源，这会让它不可靠。旧的系统能有一种基于 CPU 温度变化的 TSC 时钟，这让它不能用于计时。尝试在一些就的多核 CPU 上使用 TSC 可能在多个核心之间给出不一致的时间报告。这可能导致时间倒退，这个程序会检查这种问题。并且即使最新的系统，在非常激进的节能配置下也可能无法提供准确的 TSC 计时。

更新的操作系统可能检查已知的 TSC 问题并且当它们被发现时切换到一种更慢、更稳定的时钟源。如果你的系统支持 TSC 时间但是并不默认使用它，很可能是由于某种充分的理由才禁用它。某些操作系统可能无法正确地检测所有可能的问题，或者即便在知道 TSC 不精确的情况下也允许使用 TSC。

如果系统上有高精度事件计时器（HPET）并且 TSC 不准确，该系统将会更喜欢 HPET 计时器。计时器芯片本身是可编程的，最高允许 100 纳米的解析度，但是在你的系统时钟中可能见不到那么高的准确度。

高级配置和电源接口（ACPI）提供了一种电源管理（PM）计时器，Linux 把它称之为 acpi_pm。得自于 acpi_pm 的时钟最好时将能提供 300 纳秒的解析度。

在旧的 PC 硬件上使用的计时器包括 8254 可编程区间计时器（PIT）、实时时钟（RTC）、高级可编程中断控制器（APIC）计时器以及 Cyclone 计时器。这些计时器是以毫秒解析度为目标的。

pg_upgrade

pg_upgrade — 升级IvorySQL服务器实例

大纲

pg_upgrade -b oldbindir -B newbindir -d oldconfigdir -D newconfigdir [**option**…]

选项

pg_upgrade接受下列命令行参数：

- **-b bindir --old-bindir=bindir**

旧的 IvorySQL 可执行文件目录； 环境变量 **PGBINOLD**

- **-B** bindir **--new-bindir=bindir**

新的 IvorySQL 可执行文件目录； 默认为pg_upgrade所在的目录； 环境变量 **PGBINNEW**

- **-c --check**

只检查集簇，不更改任何数据

- **-d configdir --old-datadir=configdir**

旧的集簇数据目录；环境变量 **PGDATAOLD**

- `-D configdir --new-datadir=configdir`

新的集簇数据目录；环境变量 `PGDATANEW`

- `-j --jobs=`njobs`

要同时使用的进程或线程数

- `-k --link`

使用硬链接来代替将文件拷贝到新集簇

- `-o options --old-options options`

直接传送给旧 `postgres` 命令的选项，多个选项可以追加在后面

- `-O options --new-options options`

直接传送给新 `postgres` 命令的选项，多个选项可以追加在后面

- `-p port --old-port=port`

旧的集簇端口号；环境变量 `PGPORTOLD`

- `-P port --new-port=port`

新的集簇端口号；环境变量 `PGPORTNEW`

- `-r --retain`

即使在成功完成后也保留 SQL 和日志文件

- `-s dir --socketdir=dir`

用于升级期间postmaster套接字的目录;默认是当前目录; 环境变量 `PGSOCKETDIR`

- `-U username --username=username`

集簇的安装用户名；环境变量 `PGUSER`

- `-v --verbose`

启用详细的内部日志

- `-V --version`

显示版本信息，然后退出

- `--clone`

使用有效的文件克隆（在一些系统上也被称为“reflinks”），而不是将文件拷贝到新群集。
这可以导致数据文件接近瞬时的复制，从而获得 `-k` / `--link`
的速度优势，同时保留旧群集不受影响。文件克隆仅在某些操作系统和文件系统上得到支持。如果选中但不被支持，则 pg_upgrade运行将会出错。
目前，它支持在Linux（内核4.5或更高版本）上的Btrfs和XFS（在文件系统创建reflink支持），以及macOS上的APFS。

- `-? --help`

显示帮助，然后退出

使用

下面是用pg_upgrade执行一次升级的步骤：

1.移动旧集簇（可选）

> 如果你的安装目录不是版本相关的（例如 `/usr/local/pgsql`）， 就有必要移动当前的 IvorySQL 安装目录，以免它干扰新的 IvorySQL安装。一旦当前的 IvorySQL服务器被关闭，就可以安全地重命名 IvorySQL 安装目录。假设旧目录是 `/usr/local/pgsql`，你可以这样：
>
> ```
> mv /usr/local/pgsql /usr/local/pgsql.old
> ```
>
> 来重命名该目录。

2.对于源码安装，编译新版本

> 用兼容旧集簇的 `configure` 标记编译新的 IvorySQL 源码。在开始升级之前，pg_upgrade 将检查 `pg_controldata` 来确保所有设置都是兼容的。

3.安装新的 IvorySQL 二进制文件

> 安装新服务器的二进制文件和支持文件。pg_upgrade 会被包含在默认的安装中。
>
> 对于源码安装，如果你希望把新服务器安装在一个自定义的位置， 可以使用 `prefix` 变量：
>
> ```
> make prefix=/usr/local/pgsql.new install
> ```

4.初始化新的 IvorySQL 集簇

> 使用 `initdb` 初始化新集簇。这里也要使用与 旧集簇相兼容的 `initdb` 标志。许多预编译的安装程序会自动做这个步骤。这里没有必要启动新集簇。

5.安装扩展共享对象文件

> 许多扩展和自定义模块，无论是来自 `contrib` 或其他源，使用共享对象文件(或DLLs)，例如，`pgcrypto.so`。
> 如果旧集群使用过这些，匹配新服务器二进制的共享对象文件，必须安装在新集群中。
> 通常是通过操作系统命令。 不要加载模式定义，例如 `CREATE EXTENSION pgcrypto`，因为这些将从旧集群复制。
> 如果扩展更新是可用的，pg_upgrade将报告这一点，并创建一个脚本，可以稍后运行来更新它们。

6.拷贝定制的全文本检索文件

> 从旧集群向新集群拷贝任何定制化全文本检索文件(词典、同义词、辞典、停用词)

7.调整认证

pg_upgrade 将会多次连接到旧服务器和新服务器，因此 你可能想要在 pg_hba.conf 中把认证设置成 peer 或者使用一个 ~/.pgpass 文件。

8.停止两个服务器

确认两个数据库服务器都被停止使用，例如在 Unix 上可以：

```
pg_ctl -D /opt/IvorySQL/1.5 stop
pg_ctl -D /opt/IvorySQL/2.1 stop
```

或者在 Windows 上使用正确的服务名：

```
NET STOP postgresql-1.5
NET STOP postgresql-2.1
```

直到后面的步骤之前，流复制和日志传送后备服务器可以保持运行。

9.为后备服务器升级做准备

升级后备服务器时，请对旧的主集簇和后备集簇运行pg_controldata以验证旧的后备服务器已经完全追上。验证“Latest checkpoint location”值在所有集簇中都匹配（如果旧后备服务器在旧的主服务器之前被关闭或者如果旧的后备服务器仍在运行，则将会出现失配）。此外，请确保在新的主集簇上的 postgresql.conf 文件中，wal_level 未设置为 minimal。

10.运行 pg_upgrade

总是应该运行新服务器而不是旧服务器的pg_upgrade二进制文件。
pg_upgrade要求制定新旧集簇的数据和可执行文件（bin）目录。
你也可以指定用户和端口值，以及你是否想要用链接或克隆来取代默认的复制行为对数据文件进行处理。

如果你使用链接模式，升级将会快很多（不需要文件拷贝）并且将使用更少的磁盘空间，但是在升级后一旦启动新集簇，旧集簇就无法被访问。
链接模式也要求新旧集簇数据目录位于同一个文件系统中（表空间和 pg_wal 可以在不同的文件系统中）。
克隆模式提供了相同的速度以及磁盘空间优势，但不会导致新群集启动后旧群集不可用。
克隆模式还需要新旧数据目录位于同一文件系统中。 此模式仅在某些操作系统和文件系统上可用。

--jobs 选项允许多个 CPU 核心被用来复制/链接文件以及并行地转储和重载数据库模式。这个选项一个比较好的值是 CPU 核心数 和表空间数的最大值。这个选项可以显著地减少升级运行在一台多处理器机器上的多数据库服务器的时间。

对于 Windows 用户，你必须以一个超级账号登录，并且以 `postgres` 用户启动一个 shell 并且设置正确的路径：

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\IvorySQL\14\bin;
```

并且用带引号的目录运行pg_upgrade，例如：

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/IvorySQL/1.5/data"
        --new-datadir "C:/Program Files/IvorySQL/2.1/data"
        --old-bindir "C:/Program Files/IvorySQL/1.5/bin"
        --new-bindir "C:/Program Files/IvorySQL/2.1/bin"
```

一旦启动，`pg_upgrade` 将验证两个集簇是否兼容并且 执行升级。你可以使用 `pg_upgrade --check` 来只执行检查， 这种模式即使在旧服务器还在运行时也能使用。 `pg_upgrade --check` 也将列出任何在更新后需要做的手工调整。 如果你将要使用链接或克隆模式，你应该使用 `--link` 或 `--clone` 选项和 `--check` 一起来启用链接模式相关的检查。 `pg_upgrade` 要求在当前目录中的写权限。

显然，没有人可以在升级期间访问这些集簇。pg_upgrade 默认会在端口 50432 上运行服务器来避免意外的客户端连接。在做升级时，可以对两个集簇使用相同的端口号，因为新旧集簇不会在同时被运行。不过，在检查一个旧的运行中服务器时，新旧端口号必须不同。

如果在恢复数据库模式时发生错误，`pg_upgrade` 将会退出 并且你必须按照恢复旧集簇。要再次尝试 `pg_upgrade`，你将需要修改旧集簇，这样 pg_upgrade 模式会成功恢复。如果问题是一个 `contrib` 模块， 你可能需要从旧集簇中卸载该模块并且在升级后重新把它安装在新集簇中，不过这样做的前提是该模块没有被用来存储用户数据。

11.升级流复制和日志传送后备服务器

如果使用链接模式并且有流复制或者日志 传送后备服务器，你可以遵照下面的步骤对它们进行快速的升级。你将不用在这些后备服务器上运行 pg_upgrade，而是在主服务器上运行rsync。 到这里还不要启动任何服务器。

如果你没有使用链接模式、没有或不想使用rsync或者想用一种更容易的解决方案，请跳过这一节中的过程并且在pg_upgrade完成并且新的主集簇开始运行后重建后备服务器。

1. 在后备服务器上安装新的 IvorySQL 二进制文件

   确保新的二进制和支持文件被安装在所有后备服务器上。

2. 确保不存在新的后备机数据目录

   确保新的后备机数据目录不存在或者为空。如果运行过initdb，请删除后备服务器的新数据目录。

3. 安装扩展共享对象文件

   在新的后备机上安装和新的主集簇中相同的扩展共享对象文件。

4. 停止后备服务器

   如果后备服务器仍在运行，现在使用上述的指令停止它们。

5. 保存配置文件

从旧后备机的配置目录保存任何需要保留的配置文件，例如`postgresql.conf`（以及它包含的任何文件）、`postgresql.auto.conf`、`pg_hba.conf`，因为这些文件在下一步中会被重写或者移除。

6. 运行rsync

在使用链接模式时，后备服务器可以使用rsync快速升级。为了实现这一点，在主服务器上一个高于新旧数据库集簇目录的目录中为每个后备服务器运行这个命令：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive old_cluster
new_cluster remote_dir
```

其中`old_cluster`和`new_cluster`是相对于主服务器上的当前目录的，而`remote_dir`是后备服务器上高于新旧集簇目录的一个目录。在主服务器和后备服务器上指定目录之下的目录结构必须匹配。指定远程目录的详细情况请参考rsync的手册，例如：

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive
/opt/IvorySQL/1.5 \
      /opt/IvorySQL/2.1 standby.example.com:/opt/IvorySQL
```

可以使用rsync的`--dry-run`选项验证该命令将做的事情。虽然在主服务器上必须为至少一台后备运行rsync，可以在一台已经升级过的后备服务器上运行rsync来升级其他的后备服务器，只要已升级的后备服务器还没有被启动。

这个命令所做的事情是记录由pg_upgrade的链接模式创建的链接，它们连接主服务器上新旧集簇中的文件。该命令接下来在后备服务器的旧集簇中寻找匹配的文件并且为它们在该后备的新集簇中创建链接。主服务器上没有被链接的文件会被从主服务器拷贝到后备服务器（通常都很小）。这提供了快速的后备服务器升级。不幸地是，rsync会不必要地拷贝与临时表和不做日志表相关的文件，因为通常在后备服务器上不存在这些文件。

如果你已经把`pg_wal`放在数据目录外面，也必须在那些目录上运行rsync。

7. 配置流复制和日志传送后备服务器

为日志传送配置服务器（不需要运行`pg_start_backup()`以及`pg_stop_backup()`或者做文件系统备份，因为从属机 仍在与主机同步）。

12.恢复 pg_hba.conf

> 如果你修改了 pg_hba.conf，则要将其恢复到原始的设置。
> 也可能需要调整新集簇中的其他配置文件来匹配旧集簇，例如 postgresql.conf（以及它包含的任何文件）和 postgresql.auto.conf。

13.启动新服务器

现在可以安全地启动新的服务器，并且可以接着启动任何rsync过的后备服务器。

14.升级后处理

如果需要做任何升级后处理，pg_upgrade 将在完成后发出警告。它也将生成必须由管理员运行的脚本文件。这些脚本文件将连接到每一个需要做升级后处理的数据库。每一个脚本应该这样运行：

```
psql --username=postgres --file=script.sql postgres
```

> 这些脚本可以以任何顺序运行并且在运行之后立即删除。

> 小心
>
> 通常在重建脚本运行完成之前访问重建脚本中引用的表是不安全的，这样做可能会得到不正确的结果或者很差的性能。没有在重建脚本中引用的表可以 随时被访问。

15.统计信息

由于 pg_upgrade 并未传输优化器统计信息，在升级的尾声你将被指示运行一个命令来生成这些信息。你可能需要设置连接参数来匹配你的新集簇。

16.删除旧集簇

一旦你对升级表示满意，你就可以通过运行 pg_upgrade 完成时提到的脚本来删除旧集簇的数据目录（如果在旧数据目录中有用户定义的表空间就不可能实现自动删除）。你也可以删除旧安装目录（例如 bin、 share ）。

17.恢复到旧集簇

在运行 pg_upgrade 之后，如果你希望恢复到 旧集簇，有几个选项：

- 如果使用了 --check 选项, 则旧集群没有被修改；它可以被重新启动。
- 如果 --link 选项没有被使用, 旧集群没有被修改；它可以被重新启动。
- 如果使用了 --link 选项, 数据文件可能在新旧群集之间共享:
  - 如果 pg_upgrade 在链接启动之前中止，旧集群没有被修改，它可以重新启动。
  - 如果你没有启动新集群，旧集群没有被修改，当链接启动时，一个 .old 后缀会附加到 $PGDATA/global/pg_control。 如果要重用旧集群，从 $PGDATA/global/pg_control 移除 .old 后缀；你就可以重启旧集群。
  - 如果你已经启动新群集，它已经写入了共享文件，并且使用旧群集会不安全。这种情况下，需要从备份中还原旧群集。

注解

pg_upgrade创建不同的工作文件，如模式转储，在当前工作目录中。为了安全，请确保该目录不可被任何其他用户读取或者写入。

pg_upgrade在新旧数据目录中启动短期的postmasters。临时 Unix
套接字文件用于与这些postmasters通信，默认情况下，在当前工作目录中进行。
在某些情况下，当前目录的路径名称可能太长，无法成为有效的套接字名称。这种情况下你可以使用 `-s`
选项将套接字文件放在某些具有较短路径名称的目录中。
为了安全原因，请确保该目录不可被任何其他用户读取或者写入。（这在 Windows 上不受支持。）

如果失败、重建和重索引会影响你的安装，pg_upgrade
将会报告这些情况。用来重建表和索引的升级后脚本将会自动被建立。
如果你正在尝试自动升级很多集簇，你应该发现具有相同数据库模式的集簇
对所有集簇升级都要求同样的升级后步骤，这是因为升级后步骤是基于数据 库模式而不是用户数据。

对于部署测试，创建一个只有模式的旧集簇副本，在其中插入假数据并且升级。

pg_upgrade不支持包含使用这些 `reg*` OID-引用 系统数据类型的表列的数据库的升级：

`regcollation`
`regconfig`
`regdictionary`
`regnamespace`
`regoper`
`regoperator`
`regproc`
`regprocedure`

(`regclass`, `regrole`, and `regtype` can be upgraded.)

如果你想要使用链接模式并且你不想让你的旧集簇在新集簇启动时被修改，考虑使用克隆模式。
如果(克隆模式)不可用，可以复制一份旧集簇并且在副本上以链接模式进行升级。要创建旧集簇的一
份合法拷贝，可以在服务器运行时使用 `rsync` 创建旧集簇的 一份脏拷贝，然后关闭旧服务器并且再次运行
`rsync --checksum` 把更改更新到该拷贝以让其一致（`--checksum` 是必要的，因为 `rsync`
在判断文件修改时间的更改时的精度只能到秒级）。你可能想要排除 一些文件，例如
`postmaster.pid`。如果你的文件系统支持文 件系统快照或者 copy-on-write
文件副本，你可以使用它们来创建旧集簇和
表空间的一个备份，不过快照和副本必须被同时创建或者在数据库服务器关闭 期间被创建。

pg_waldump

pg_waldump — 以人类可读的形式显示一个IvorySQL 数据库集簇的预写式日志

大纲

`pg_waldump` [`option`…] [`startseg` [`endseg`]]

选项

下列命令行选项控制输出的位置和格式：

- `startseg`

从指定的日志段文件开始读取。这也隐含地决定了要搜索文件的路径以及 要使用的时间线。

- `endseg`

在读取指定的日志段文件后停止。

- `-b --bkp-details`

输出有关备份块的细节。

- `-e end --end=end`

在指定的WAL位置停止读取，而不是一直读取到日志流的末尾。

- `-f --follow`

在到达可用 WAL 的末尾之后，保持每秒轮询一次是否有新的 WAL 出现。

- `-n limit --limit=limit`

显示指定数量的记录，然后停止。

- `-p path --path=path`

指定搜索日志段文件的目录或包含这些文件的包含 `pg_wal` 子目录的目录。
缺省值是在当前目录中搜索，当前目录的 `pg_wal` 子目录和 `PGDATA` 的 `pg_wal` 子目录。

- `-q --quiet`

除错误外，不要打印任何输出。当您想知道一系列WAL记录是否可以成功解析但不关心记录内容时，此选项非常有用。

- `-r rmgr --rmgr=rmgr`

只显示由指定资源管理器生成的记录。如果把 `list` 作为资源管理器名称
传递给这个选项，则打印出可用资源管理器名称的列表然后退出。

- `-s start --start=start`

要从哪个WAL位置开始读取。默认是从找到的最早的文件的第一个可用日志记录开始。

- `-t `timeline` --timeline=`timeline``

要从哪个时间线读取日志记录。默认是使用 `startseg` （如果指定） 中的值，否则默认为 1。

- `-V --version`

打印pg_waldump版本并且退出。

- `-x xid --xid=xid`

只显示用给定事务 ID 标记的记录。

- `-z --stats[=record]`

显示概括统计信息（记录的数量和尺寸以及全页镜像）而不是显示
每个记录。可以选择针对每个记录生成统计信息，而不是针对每个资源管理器生成。

- `-? --help`

显示有关pg_waldump命令行参数的帮助并且退出。

环境

- `PGDATA`

数据目录；另请参阅 -p 选项。

- PG_COLOR

规定在诊断消息中是否使用颜色。可能的值为 always、 auto、 never。

注解

当服务器正在运行时可能会给出错误的结果。

只有指定的时间线 会被显示（如果没有指定，则显示默认时间线）。 其他时间线上的记录会被忽略。

pg_waldump不能读取具有后缀 .partial 的 WAL 文件。如果需要读取那些文件，需要从文件名中移除 .partial 后缀。

postgres

postgres — IvorySQL数据库服务器

大纲

postgres [选项…]

选项

postgres
接受下列命令行参数。你也可以通过设置一个配置文件来减少键入大部分这些选项。有些（安全）选项还可以从连接的客户端以一种与应用相关只应用于会话的方法设置。例如，如果设置了 PGOPTIONS 环境变量，那么基于libpq的客户端将都把那个字符串传递给服务器，它将被服务器解释成 postgres 命令行选项。

通用选项

- -B nbuffers

设置被服务器进程使用的共享内存缓冲区数量。这个参数的默认值是initdb自动选择的。指定这个选项等效于设置 shared_buffers 配置参数。

- -c name=value

设置一个命名的运行时参数。大多数其它命令行选项实际上都是这种参数赋值的短形式。-c 可以出现多次用于设置多个参数。

- -C name

打印命名运行时参数的值，并且退出（详见上面的 -c 选项）。这可以被用在一个运行服务器上，并且从 postgresql.conf 中返回值，这些值可能被在这次调用中的任何参数修改过。它并不反映集簇启动时提供的参数。这个选项用于与一个服务器实例交互的其他程序来查询配置参数值，例如 pg_ctl。面向用户的应用应该使用 SHOW or the pg_settings 视图。

- -d debug-level

设置调试级别。数值设置得越高，写到服务器日志的调试输出就越多。取值范围是从 1 到 5。还可以针对某个特定会话使用 -d 0 来阻止父 postgres 进程的服务器日志级别被传播到这个会话。

- -D datadir

指定数据库配置文件的文件系统位置。

- `-e`

把默认日期风格设置为“European”，也就是输入日期域的顺序是
`DMY`。这也导致在一些日期输出格式中把日打印在月之前。

- `-F`

禁用 `fsync` 调用以提高性能，但是要冒系统崩溃时数据损坏的风险。指定这个选项等效于禁用 fsync
配置参数。

- `-h hostname`

指定 `postgres` 监听来自客户端应用 TCP/IP 连接的 IP
主机名或地址。该值也可以是一个用逗号分隔的地址列表，或者 `*`
表示监听所有可用的地址。一个空值表示不监听任何 IP 地址，在这种情况下可以使用 Unix
域套接字连接到服务器。缺省只监听localhost。声明这个选项等效于设置listen_addresses配置参数。默认只监听localhost。指定这个选项等效于设置 listen_addresses 配置参数。

- `-i`

允许远程客户端使用 TCP/IP （互联网域）连接。没有这个选项，将只接受本地连接。这个选项等效于在
`postgresql.conf` 中或者通过 `-h` 选项将 `listen_addresses` 设为 `*`。这个选项已经被废弃，因为它不允许访问
listen_addresses 的完整功能。所以最好直接设置 `listen_addresses`。

- `-k `directory``

指定 `postgres` 用来监听来自客户端应用连接的 Unix
域套接字的目录。这个值也可以是一个逗号分隔的目录列表。一个空值指定不监听任何 Unix
域套接字，在这种情况下只能用 TCP/IP 套接字来连接到服务器。默认值通常是
`/tmp`，但是可以在编译的时候修改。指定这个选项等效于设置 unix_socket_directories 配置参数。

- `-l`

启用使用SSL的安全连接。要使这个选项可用，编译IvorySQL时必须打开SSL支持。有关使用SSL的更多信息
。

- `-N max-connections`

设置该服务器将接受的最大客户端连接数。该参数的默认值由initdb自动选择。指定这个选项等效于设置
max_connections 配置参数。

- `-p `port``

指定 `postgres` 用于监听客户端应用连接的 TCP/IP 端口或本地 Unix 域套接字文件扩展。默认为 `PGPORT`
环境变量的值。如果 `PGPORT` 没有设置，那么默认值是编译期间设立的值（通常是
5432）。如果你指定了一个非默认端口，那么所有客户端应用都必须用命令行选项或者 `PGPORT`
指定同一个端口。

- `-s`

在每条命令结束时打印时间信息和其它统计信息。这个选项对测试基准和调节缓冲区数量有用处。

- `-S work-mem`

指定在使用临时磁盘文件之前排序和散列表使用的基本内存量。

- `-V --version`

打印postgres版本并退出。

- `--name=value`

设置一个命名的运行时参数；其缩写形式是 `-c`。

- `--describe-config`

这个选项会用制表符分隔的 `COPY` 格式导出服务器的内部配置变量、描述以及默认值。设计它的目的是用于管理工具。

- `-? --help`

显示有关postgres的命令行参数，并且退出。

半内部选项

这里描述的选项主要被用于调试目的，并且在某些情况下可以协助恢复严重受损的数据库。在生产数据库环境中应该不会去使用它们。在这里列举它们只是为了让IvorySQL系统开发者使用。此外，这些选项可能在将来的版本中更改或删除而不另行通知。

- `-f { s | i | o | b | t | n | m | h }`

禁止某种扫描和连接方法的使用：`s` 和 `i` 分别禁用顺序和索引扫描，`o`、`b` 和 `t` 分别禁用只用索引扫描、位图索引扫描以及 TID 扫描，而 `n`、`m` 和 `h` 则分别禁用嵌套循环、归并和哈希连接。顺序扫描和嵌套循环连接都不可能完全被禁用。`-fs` 和 `-fn` 选项仅仅是在有其他选择时不鼓励优化器使用这些计划类型。

- `-n`

该选项主要用于调试导致服务器进程异常崩溃的问题。对付这种情况的一般策略是通知所有其它服务器进程，让它们终止并且接着重新初始化共享内存和信号量。这是因为一个错误的服务器进程可能在终止之前就已经对共享状态造成了破坏。该选项指定 `postgres` 将不会重新初始化共享数据结构。一个有经验的系统程序员这时就可以使用调试器检查共享内存和信号量状态。

- `-O`

允许修改系统表的结构。这个选项用于 `initdb`。

- `-P`

读取系统表时忽略系统索引（但在更改系统表时仍然更新索引）。这在从损坏的系统索引中恢复时有用。

- `-t pa[rser] | pl[anner] | e[xecutor]`

打印与每个主要系统模块相关的查询的时间统计。这个选项不能和 `-s` 选项一起使用。

- `-T`

该选项主要用于调试导致服务器进程异常崩溃的问题。对付这种情况的一般策略是通知所有其它服务器进程，让它们终止并且接着重新初始化共享内存和信号量。这是因为一个错误的服务器进程可能在终止之前就已经对共享状态造成了破坏。该选项指定 `postgres` 将通过发送 `SIGSTOP` 信号停止其他所有服务器进程，但是并不让它们终止。这样就允许系统程序员手动从所有服务器进程收集内核转储。

- `-v protocol`

声明这次会话使用的前/后服务器协议的版本数。该选项仅在内部使用。

- `-W seconds`

在一个新服务器进程被启动时，它实施认证过程之后会延迟这个选项所设置的秒数。这就留出了机会来用一个调试器附着在服务器进程上。

用于单用户模式的选项

下面的选项仅适用于单用户模式（见 Single-User Mode）。

- `--single`

选择单用户模式。这必须是命令行中的第一个选项。

- `database`

指定要访问的数据库的名称。这必须是命令行中的最后一个参数。如果省略它，则默认为用户名。

- `-E`

在执行命令之前回显所有命令到标准输出。

- `-j`

使用跟着两个新行的分号而不是仅用新行作为命令终止符。

- `-r filename`

将所有服务器日志输出发送到 `filename` 中。 只有在作为一个命令行选项提供时，这个选项才会兑现。

环境

- `PGCLIENTENCODING`

客户端使用的默认字符编码（客户端可以独立地覆盖它）。这个值也可以在配置文件中设置。

- `PGDATA`

默认的数据目录位置。

- `PGDATESTYLE`

DateStyle 运行时参数的默认值（这个环境变量的使用已被废弃）。

- `PGPORT`

默认端口号（在配置文件中设置更好）

诊断

一个提到了 `semget` 或 `shmget`
的错误消息可能意味着你需要配置内核来提供足够的共享内存和信号量。你也可以通过降低 shared_buffers
值减少IvorySQL的共享内存消耗， 或者降低 max_connections
值减少信号量消耗，这样可以推迟对内核的重新配置。

如果一个消息说另外一个服务器已经在运行，应该仔细地检查，例如根据你的系统可以用命令

```
$ ps ax | grep postgres
```

或

```
$ ps -ef | grep postgres
```

如果你确信没有冲突的服务器正在运行，那么你可以删除消息中提到的锁文件然后再次尝试。

如果一个失败消息指示它无法绑定到一个端口，可能意味着该端口已经被某些非IvorySQL进程所使用。如果你终止 postgres
并且立即使用相同的端口重启它，你也可能会得到这种错误。在这种情况系，你必须等待几秒直到操作系统关闭该端口，然后再重试。最后，如果你指定了一个操作系统认为需要保留的端口号，你可能也会得到这个错误。例如，很多版本的 Unix 认为低于 1024 的端口号是“可信的”并且只允许 Unix 超级用户访问它们。

注解

实用命令 pg_ctl 可以用来安全方便地启动和关闭 postgres 服务器。

只要有可能，就不要使用 SIGKILL 杀死主 postgres 服务器。这样会阻止 postgres
在终止前释放它持有的系统资源（例如共享内存和信号量）。这样可能会导致启动新的 postgres
进程时出现问题。

要正常地终止 postgres 服务器，可以使用 SIGTERM、 SIGINT 或者 SIGQUIT
信号。第一个在退出前将等待所有客户端终止，第二个将强行断开所有客户端的连接，第三个会不做正确的关闭立即退出并且会导致重启时的恢复。

SIGHUP 信号会重新加载服务器配置文件。也可以向一个单独的服务器进程发送 SIGHUP
信号，但是这样做通常没什么意义。

要取消一个正在运行的查询，可以向运行该查询的进程发送 SIGINT
信号。要干净地终止一个后端进程，可向它发送`SIGTERM`。在 SQL
中可调用的与这两种动作等效的命令可参考 pg_cancel_backend 和 pg_terminate_backend。

postgres 服务器使用 SIGQUIT
来告诉子服务器进程终止但不做正常的清理。该信号不应该被用户使用。向一个服务器进程发送 SIGKILL
也是不明智的 — 主 postgres
进程将把这解释为一次崩溃，并且作为其标准崩溃恢复过程的一部分，它将强制所有的后代进程退出。

缺陷

-- 选项在FreeBSD或OpenBSD上无法运行，应该使用 -c。这在受影响的系统里是个缺陷；
如果这个缺陷没有被修复，将来的IvorySQL版本将提供一种解决方案。

单用户模式

要启动一个单用户模式的服务器，使用这样的命令

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

用 -D 给服务器提供正确的数据库目录的路径，或者确保环境变量 PGDATA
被设置。同时还要指定你想在其中工作的特定数据库的名字。

通常，单用户模式的服务器会把换行符当做命令输入的终止符。它不明白分号的作用，因为那属于psql。要想把一个命令分成多行，必须在最后一个换行符以外的每个换行符前面敲一个反斜线。这个反斜线和旁边的新行都会被从输入命令中去掉。注意即使在字符串或者注释中也会这样做。

但是如果使用了 -j 命令行选项，那么单个新行将不会终止命令输入。相反，分号-新行-
新行的序列才会终止命令输入。也就是说，输入一个紧跟着空行的分号。在这种模式下，反斜线-
新行不会被特殊对待。此外，在字符串或者注释内的这类序列也不会被特殊对待。

不管在哪一种输入模式中，如果输入的一个分号不是正好在命令终止符之前或者不是命令终止符的一部分，它会被认为是一个命令分隔符。当真正输入一个命令终止符时，已经输入的多个语句将被作为一个单个事务执行。

要退出会话，输入EOF（通常是 Control + D
）。如果从上一个命令终止符以来已经输入了任何文本，那么EOF将被当作命令终止符，并且如果要退出则需要另一个EOF。

请注意单用户模式的服务器不会提供复杂的行编辑特性（例如没有命令历史）。但用户模式也不会做任何后台处理，例如自动检查点或者复制。

例子

要用默认值在后台启动 `postgres`：

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

要用指定端口启动 `postgres`，例如 1234：

```
$ postgres -p 1234
```

要使用psql连接到这个服务器，用 -p 选项指定这个端口：

```
$ psql -p 1234
```

或者设置环境变量 `PGPORT`：

```
$ export PGPORT=1234
$ psql
```

命名运行时参数可以用这些形式之一设置：

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

两种形式都覆盖 `postgresql.conf` 中可能存在的 `work_mem` 设置。请注意在参数名中的下划线在命令行可以写成下划线或连字符。除了用于短期的实验外，更好的习惯是编辑 `postgresql.conf` 中的设置， 而不是倚赖命令行开关来设置参数。

.2. asciidoc语法快速参考

IvorySQL文档是用“asciidoc”编写的。为确保格式的质量和一致性，在修改更新文档时应遵循某些 asciidoc 规则。

规范说明

asciidoc规范

1、标题从一级开始递增使用，禁止跳级使用。例如：一级标题下面不能直接使用三级标题；二级标题下面不能直接使用四级标题。

2、标题必须统一使用 ATX 风格，即在标题前加 = 号来表示标题级别。

3、标题的引导符号 = 后必须空一格再接标题内容。

4、标题的引导符号“=”后只能空一格后再接标题内容，不能有多个空格。

5、标题必须出现在一行行首，即标题的 = 号前不能有任何空格。

6、标题末尾仅能出现中英文问号、反引号、中英文单双引号等符号。其余如冒号、逗号、句号、感叹号等符号均不能在标题末尾使用。

7、标题上面须空一行。

8、文档中不能连续出现内容重复的标题，如一级标题为 = IvorySQL 架构，紧接着的二级标题就不能是 == IvorySQL 架构。如果不是连续的标题，则标题内容可重复。

9、文档中只能出现一个一级标题。

10、一般来说，除 TOC.adoc 文件可缩进 2 个空格外，其余所有 .adoc 文件每缩进一级，默认须缩进 4 个空格。

11、文档中（包括代码块内）禁止出现 Tab 制表符，如需缩进，必须统一用空格代替

12、禁止出现连续的空行。

13、块引用符号 > 后禁止出现多个空格，只能使用一个空格，后接引用内容。

14、使用有序列表时，必须从 1 开始，按顺序递增。

15、使用列表时，每一列表项的标识符（+、-、* 或数字）后只能空一格，后接列表内容。

16、列表（包括有序和无序列表）前后必须各空一行。

17、代码块前后必须各空一行。

18、文档中禁止出现裸露的 URL。如果希望用户能直接点击并打开该 URL，则使用一对尖括号 (<URL>) 包裹该 URL。如果由于特殊情况必须使用裸露的 URL，不需要用户通过点击打开，则使用一对反引号 `URL` 包裹该 URL。

19、使用加粗、斜体等强调效果时，在强调标识符内禁止出现多余的空格。如不能出现 ** 加粗文本 **。

20、单个反引号包裹的代码块内禁止出现多余的空格。如不能出现 ` 示例文本`。

21、链接文本两边禁止出现多余的空格。如不能出现 某链接。

22、链接必须有链接路径。如不能出现空链接等情况。

示例

1、标题从一级开始递增使用，禁止跳级使用。

```
= Heading 1
=== Heading 3

We skipped out a 2nd level heading in this document
```

2、标题必须统一使用 ATX 风格，即在标题前加 = 号来表示标题级别。

```
= Heading 1
```

```
== Heading 2
=== Heading 3
==== Heading 4
== Another Heading 2
=== Another Heading 3
```

3、标题的引导符号 = 后必须空一格再接标题内容。禁止=后多个空格，禁止=前出现空格

错误示范：

```
= Heading 1
== Heading 2
```

正确示范：

```
= Heading 1
== Heading 2
```

4、标题末尾仅能出现中英文问号、反引号、中英文单双引号等符号。

错误示范

```
= This is a heading.
```

正确示范

```
= This is a heading
```

5、标题上面空一行

错误示范

```
= Heading 1
Some text
Some more text## Heading 2
```

正确示范

```
= Heading 1
Some text
Some more text

== Heading 2
```

6、文档中不能连续出现内容重复的标题，如一级标题为 = IvorySQL 描述，紧接着的二级标题就不能是 ==

IvorySQL 描述。如果不是连续的标题，则标题内容可重复。

错误示范

```
= Some text

== Some text
```

正确示范

```
= Some text

== Some more text
```

7、文档中只能出现一个一级标题。

错误示范

```
= Top level heading

= Another top-level heading
```

正确示范

```
= Title

== Heading

== Another heading
```

8、一般来说，除 TOC.adoc 文件可缩进 2 个空格外，其余所有 .adoc 文件每缩进一级，默认须缩进 4 个空格。

错误示范

```
* List item
  * Nested list item indented by 3 spaces
```

正确示范:

```
* List item
    * Nested list item indented by 4 spaces
```

9、文档中（包括代码块内）禁止出现 Tab 制表符，如需缩进，必须统一用空格代替

错误示范:

```
Some text
    * hard tab character used to indent the list item
```

正确示范:

```
Some text
  * Spaces used to indent the list item instead
```

10、禁止出现连续的空行

错误示范

```
Some text here



Some more text here
```

正确示范:

```
Some text here

Some more text here
```

11、块引用符号 > 后禁止出现多个空格，只能使用一个空格，后接引用内容。

错误示范

```
>  This is a blockquote with bad indentation>  there should only be one.
```

正确示范

```
> This is a blockquote with correct> indentation.
```

12、使用有序列表时，必须从 1 开始，按顺序递增。

错误示范:

```
1. Do this.
1. Do that.
1. Done.
```

```
0. Do this.
1. Do that.
```

```
2. Done.
```

```
正确示范:
```

```
1. Do this.
2. Do that.
3. Done.
```

13、使用列表时，每一列表项的标识符（+、-、* 或数字）后只能空一格，后接列表内容。

正确示范

```
* Foo
* Bar
* Baz

1. Foo
  * Bar
1. Baz
```

14、列表（包括有序和无序列表）前后必须各空一行。

错误示范

```
Some text* Some* List

1. Some2. List

Some text
```

正确示范

```
Some text

* Some
* List

1. Some
2. List

Some text
```

15、代码块前后必须各空一行。

错误示范

```
Some text
```
Code block
```
```
Another code block
```
Some more text
```

正确示范

```
Some text

```
Code block
```

```
Another code block
```

Some more text
```

16、文档中禁止出现裸露的 URL。如果希望用户能直接点击并打开该 URL，则使用一对尖括号 (<URL>) 包裹该 URL。如果由于特殊情况必须使用裸露的 URL，不需要用户通过点击打开，则使用一对反引号 `URL` 包裹该 URL。

错误示范

```
For more information, see https://www.example.com/.
```

正确示范

```
For more information, see <https://www.example.com/>.
```

17、使用加粗、斜体等强调效果时，在强调标识符内禁止出现多余的空格。如不能出现 ** 加粗文本 **。

错误示范

```
Here is some ** bold ** text.

Here is some * italic * text.
```

```
Here is some more __ bold __ text.

Here is some more _ italic _ text.
```

正确示范:

```
Here is some **bold** text.

Here is some *italic* text.

Here is some more __bold__ text.

Here is some more _italic_ text.
```

18、单个反引号包裹的代码块内禁止出现多余的空格。如不能出现 ` 示例文本` 。

错误示范:

```
some text
 some text
```

正确示范:

```
some text
```

19、链接文本两边禁止出现多余的空格。如不能出现 某链接。

错误示范

```
[ a link ](https://www.example.com/)
```

正确示范:

```
[a link](https://www.example.com/)
```

20、链接必须有链接路径。如不能出现空链接等情况。

错误示范

```
[an empty link]()

[an empty fragment](#)
```

正确示范:

```
[a valid link](https://example.com/)

[a valid fragment](#fragment)
```

21、文档中的代码块统一使用三个反引号进行包裹，禁止使用缩进四格风格的代码块。

错误示范：

```
Some text.

  # Indented code

More text.
```

正确示范

```
```ruby
# Fenced code
```
More text.
```

PG参数参考手册

PostgreSQL 参数参考手册

本文参考来源： https://www.postgresql.org/docs/13/runtime-config.html

本文档贡献单位及贡献者:

瀚高基础软件股份有限公司：徐云鹤、张波、咸士杰、类延良

中国开源软件推进联盟PostgreSQL分会：魏波

本文档错误反馈方式：

请在 https://github.com/IvorySQL/ivorysql_docs/issues提交issue。

文档下载: PostgreSQL参考手册

Resource Usage / Disk

temp_file_limit

参数名称	temp_file_limit
参数值数据类型	integer
默认值	-1,表示不限制
取值范围	-1到2147483647
参数单位	如果指定的值不带单位，则以KB为单位

参数含义	指定一个进程可以占用的最大的磁盘空间。该磁盘空间可以用于temporary files(sort及hash使用)或者持有cursor的storage file。试图超过该限制的transaction会被取消。
是否可session级修改	可以，此时仅限于superuser可以在session级修改
修改后何时生效	非session级修改Reload即可生效；Session级修改时会立即在session级生效

Replication

track_commit_timestamp

参数名称	track_commit_timestamp
参数值数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	记录transaction的提交时间。注意：只有当这个参数是on时，函数pg_xact_commit_timestamp(xid)、pg_last_committed_xact()才能获得已提交事务的时间值。
是否可session级修改	否
修改后何时生效	重启PG instance生效

Version and Platform Compatibility / Other Platforms and Clients

transform_null_equals

参数名称	transform_null_equals
参数值数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	当本参数值为on时，PG会将expr=NULL(或者NULL=expr)视为expr IS NULL.本参数仅仅影响=NULL这个形式，不会影响其他形式（如in）
是否可session级修改	可以，superuser和普通用户均可以在session级修改
修改后何时生效	Reload即可生效Session级修改时会立即在session级生效

Resource Usage / Kernel Resources

max_files_per_process

参数名称	max_files_per_process
参数值数据类型	integer
默认值	1000
取值范围	64到2147483647
参数单位	
参数含义	本参数用于设置每个server subprocess允许同时打开的files的最大数量。如果遇到 "Too many open files"的提示，请尝试减少本参数值。
是否可session级修改	否

修改后何时生效	重启PG instance生效。

Process Title

cluster_name

参数名称	cluster_name
参数值数据类型	string
默认值	空值
取值范围	
参数单位	
参数含义	本参数用于设置PGCluster的名字，可以用操作系统命令ps -ef| grep postgres查看该名字。该名字会出现在所有server process的开头位置，另外，对于standby连接来说，该名字还是默认的application name.
是否可session级修改	否
修改后何时生效	重启PG instance生效。

update_process_title

参数名称	update_process_title
参数值数据类型	Bool
默认值	on（除windows之外的平台）；off（Windows平台）
取值范围	on和off
参数单位	
参数含义	每当服务器接收到一个新的 SQL 命令时都更新process title
是否可session级修改	是，此时仅限于superuser可以在session级修改
修改后何时生效	Reload即可生效Session级修改时会立即在session级生效

Replication / Master Server

synchronous_standby_names

参数名称	synchronous_standby_names
参数值数据类型	string
默认值	空字符串
取值范围	
参数单位	

参数含义	本参数为primary角色库参数。本参数值区分大小写。本参数指定了支持同步复制的standby server清单。standby server的name是standby连接信息中的application_name,在物理复制standby中，standby server的name是在primary_conninfo中设置的，如果cluster_name配置参数的配置过了，那么standby server的name默认是cluster_name参数值，否则是walreceiver对于逻辑复制，可以在subscription中设置连接信息，默认值是subscription name本参数使用如下语法指定了一系列的standby servers的列表：[FIRST] num_sync (standby_name [, …])ANY num_sync (standby_name [, …])standby_name [, …] num_sync代表同步standby的个数举例，FIRST 3 (s1, s2, s3, s4)的含义:s1, s2, s3这三个standby是同步standbys4这个standby是潜在的同步standby(potential synchronous standbys)如果当前同步standby不论什么原因断开，它的位置会立即被next-highest-priority standby取代。关键字FIRST是可选的。FIRST 3 (s1, s2, s3, s4)中，优先级由大到小为s1>s2>s3>s4 举例， ANY 3 (s1, s2, s3, s4) 的含义：s1, s2, s3, s4这四个standby中的任意3个standby server应用wal变化之后，primary端的事务才能提交请务必注意：FIRST和ANY关键字是大小写敏感的。如果这两个关键字被用于standby server的name，那么standby_name需要用双引号引起来。在PostgreSQL 9.6之前版本中使用的第三种语法已经受到支持。该语法与"FIRST 1"是相同作用。比如， FIRST 1 (s1, s2) 和 s1, s2 有相同的含义：要么s1要么s2被选为同步standby 表示匹配任何standby name 没有机制来确保standby name的唯一性。在重复发生时，匹配到的standby会被视为更高的优先级。尽管确切地说哪个是不确定的注意：每个standby_name应该是有效地SQL标识符，除非是。如果需要,你可以使用双引号。但是请注意：standby_names与standby application_name 比较时，是不区分大小写的，不论有无双引号。如果synchronous_standby_names参数不指定值，同步复制不会启用。即使同步复制被启用，通过设置synchronous_commit配置参数为local或者off，单个事务可以被配置为不等待standby端的wal被应用。 本参数可以在postgresql.conf文件或者在the server command line设置
是否可session级修改	否
修改后何时生效	Reload即可生效。

vacuum_defer_cleanup_age

参数名称	vacuum_defer_cleanup_age
参数值数据类型	integer
默认值	0
取值范围	0到1000000
参数单位	

参数含义	该参数指定了在primary库上vacuum和HOT updates 会延迟清理dead row versions的事务的数量。默认值是零个事务，零个事务的含义是dead row versions会尽可能快的被removed掉。尽可能快是指dead row versions不会被任何open transaction可见。你可以在hot standby环境中的primary端为此参数设置为非零值，这允许standby端的query运行更长的时间以便完成，而不会遇到过早的dead row versions cleanup而导致冲突。由于本参数值是根据在primary上发生的写入事务的数量来衡量的，因此，很难预测会为standby query增加多长时间。 本参数可以在postgresql.conf文件或者在the server command line设置 你可以考虑在standby端使用hot_standby_feedback配置参数作为vacuum_defer_cleanup_age的替代。本参数不会防止那些已经达到age（由old_snapshot_threshold配置参数指定）的dead rows的清理。
是否可session级修改	否
修改后何时生效	Reload即可生效。

Replication / Subscribers

max_logical_replication_workers

参数名称	max_logical_replication_workers
参数值数据类型	integer
默认值	4
取值范围	0到262143
参数单位	
参数含义	本参数设置Logical replication workers进程的最大数量，此处的worker进程包括两种：apply workers 和table synchronization workers。Logical replication workers进程的数量从max_worker_processes配置参数定义的池中获取。
是否可session级修改	否
修改后何时生效	重启PG instance生效。

max_sync_workers_per_subscription

参数名称	max_sync_workers_per_subscription
参数值数据类型	integer
默认值	2
取值范围	0到262143
参数单位	
参数含义	指定了每个subscription中synchronization workers的最大数量。本参数控制在subscription初始化 或者新表加入时的initial data copy的并行度。目前，每个table只能指定一个synchronization workersynchronization workers从max_logical_replication_workers配置参数定义的池中取得。
是否可session级修改	否
修改后何时生效	Reload即可生效。

Write-Ahead Log / Archive Recovery

restore_command

参数名称	restore_command
参数值数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	恢复时指定archive wal的位置 restore_command = 'cp /mnt/server/archivedir/%f "%p"' restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"' # Windows restore_command对应的命令执行成功之后，restore_command返回零，%f代表archived wal文件名 %p代表copy destination path name %r代表含有最后一个有效的resart point的file的文件名 %%代表一个%字符(当需要写入%字符的情况下使用) 本参数需要在pg instance启动之前进行设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效。

archive_cleanup_command

参数名称	archive_cleanup_command
参数值数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	本参数定义了在每个restart point时所执行的shell命令。 archive_cleanup_command参数的目的是提供一个清理不再被standby server所需要的老的archived wal file的机制。 %r代表最后一个有效的restart point的wal file。该wal file是最早一个必须保留的文件，以便允许restore操作可以被restart。注意：restart point 是一个 point ，该point用于standby server重启recovery操作。因此，所有早于%r的文件可以被安全的清理掉。本信息可以用来truncate掉archive wal file，以便满足从当前restore可以restart的最小需求。 pg_archivecleanup实用命令常被用在单个standby配置的archive_cleanup_command参数中 archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r' %%代表一个%字符(当需要写入%字符的情况下使用)如果命令返回非零值，一个警告日志消息会被写入。一个例外是：当命令被一个signal终止或者shell中有错误时，一个fatal error会被抛出。 本参数可以在postgresql.conf设置，也可以在server command line设置。
是否可session级修改	否
修改后何时生效	Reload即可生效。

recovery_end_command

参数名称	recovery_end_command
参数值数据类型	String
默认值	空字符串
取值范围	

参数单位	
参数含义	本参数用于定义在recovery结束时候的需要执行的shell命令。如果命令返回非零值，一个警告日志消息会被写入，然后数据库会继续启动。一个例外是：当命令被一个signal终止或者shell中错误时，数据库不会继续启动。%r代表最后一个有效的restart point的wal file
是否可session级修改	否
修改后何时生效	Reload即可生效。

Error Handling

exit_on_error

参数名称	exit_on_error
参数值数据类型	Bool
默认值	off
取值范围	on和off
参数单位	
参数含义	当设置为on时，任何error均会导致session被终止。默认情况下，本参数值为off，只有FATAL error才会终止session。 postgres=# set exit_on_error=on; SET postgres=# show exit_on_error ; -[RECORD 1]-+--- exit_on_error | on postgres=# begin; BEGIN postgres=# select \ form t1; FATAL: syntax error at or near "form" LINE 1: select * form t1; ^ server closed the connection unexpectedly This probably means the server terminated abnormally before or while processing the request. The connection to the server was lost. Attempting reset: Succeeded.postgres=#
是否可session级修改	是
修改后何时生效	Reload即可生效Session级别修改可以立即生效。

restart_after_crash

参数名称	restart_after_crash
参数值数据类型	bool
默认值	on
取值范围	
参数单位	
参数含义	当设置为on时，PostgreSQL会在backend crash之后重新初始化(即:restart)。在某些情况下，比如PostgreSQL被Clusterware调用时，需要禁用PostgreSQL的restart(将该参数值设置为off)以便clusterware获得PostreSQL的控制权并采取适当的操作。
是否可session级修改	否
修改后何时生效	Reload即可生效

data_sync_retry

参数名称	data_sync_retry
参数值数据类型	bool
默认值	off
取值范围	off 和on
参数单位	

参数含义	如果设置为off(默认值),当PostgreSQL flush数据到文件系统上遇到failure时，PostgreSQL会抛出一个panic级别的error。这会导致数据库服务crash掉。本参数只能在数据库服务启动之前设置。在某些操作系统中，当回写(write-back)失败之后，kernel的page cache中的数据状态是未知的(unknown).在一些情况下，它可能会被完全忘记，设置为尝试是不安全的，在实际上数据已经丢失的情况下，第二次尝试可能会报告成功，在这些场景中，避免数据损失的唯一方法是解决掉硬件问题之后从wal中恢复。当设置为on时，PostgreSQL会报告一个error并会继续运行，data flushing操作会在稍后的checkpoint中尝试。注意：仅在调查操作系统对buffered data的处理以防回写失败的情况下，才将其设置为on。
是否可session级修改	否
修改后何时生效	重启PG instance生效。

Write-Ahead Log / Archiving

archive_mode

参数名称	archive_mode
参数值数据类型	enum
默认值	off
取值范围	off,on,always
参数单位	
参数含义	指定归档模式。on表示启用归档模式，当wal_level配置参数值为minimal时, archive_mode 不能设置为on。当本参数值设置为always时，standby 实例会为收到的每个wal segments调用归档命令,也就是standby实例上也会进行wal的归档操作。本参数仅仅能在pg启动时设置
是否可session级修改	否
修改后何时生效	重启PG instance生效。

archive_command

参数名称	archive_command
参数值数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	定义对wal进行归档的命令。当archive_mode配置参数启用并且archive_command配置参数是空字符串时，wal archiving暂时被禁用，但是pg会继续积累wal segment文件。archive_command参数值设置为/bin/true会禁用归档，但这样会导致wal文件归档中断，归档中断是无法进行归档恢复的，请注意这一点。使用举例如下： archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f' # Unix archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"' # Windows 退出码是零代表本命令正常运行。本参数可以在postgresql.conf或者server command line设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

archive_timeout

参数名称	archive_timeout
参数值数据类型	integer
默认值	0
取值范围	0到1073741823
参数单位	若是不指定单位，那默认的单位是秒
参数含义	强制进行wal segments归档的时长。使用本参数归档出来的wal segment与正常的wal segment是相同的大小。因此，本参数值设置太小会对存储空间有浪费。本参数可以在postgresql.conf或者server command line设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

Resource Usage / Background Writer

bgwriter_delay

参数名称	bgwriter_delay
参数值数据类型	integer
默认值	200
取值范围	10到10000
参数单位	若是不指定单位，那默认的单位是毫秒(ms)
参数含义	在每一轮循环中，background writer都会为一定数量的脏缓冲区发出写操作，然后background writer进行睡眠，睡眠的时间为bgwriter_delay参数值，然后再唤醒，然后重复。设置bgwriter_delay为不是10的倍数的值可能与将其设置为10的下一个更高的倍数具有相同的结果。本参数可以在postgresql.conf或者server command line设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

bgwriter_lru_maxpages

参数名称	bgwriter_lru_maxpages
参数值数据类型	integer
默认值	100
取值范围	0到1073741823
参数单位	
参数含义	每轮bgwriter进程写入LRU Page的最大数量。本参数设置为零会禁用background writing行为，但是background writer进程依然会存在。本参数可以在postgresql.conf或者server command line设置
是否可session级修改	否
修改后何时生效	Reload即可

bgwriter_lru_multiplier

参数名称	bgwriter_lru_multiplier

参数值数据类型	real
默认值	2.0
取值范围	0到10
参数单位	
参数含义	每轮写dirty buffer的数量基于服务器进程(即:backend process)在最近几轮中所需要的新buffer数量。最近的平均需要量乘以本参数值就是下一轮需要的新buffers数量的估计值。Dirty buffers会一直写入，直到有那么多干净的、可重用的buffers为止。因此，本参数值为1.0表示准确预计了需要的buffers数量。较大的本参数值提供了一些缓冲，以防需求激增，而较小的本参数值会有意将写dirty buffers操作留给服务器进程(即:backend process)完成。本参数可以在postgresql.conf或者server command line设置
是否可session级修改	否
修改后何时生效	Reload即可生效

bgwriter_flush_after

参数名称	bgwriter_flush_after
参数值数据类型	integer
默认值	Linux平台下是64，也就是512KB；其他平台下是0
取值范围	源码中BLCKSZ 符号常量为8KB时，取值范围是0到256；源码中BLCKSZ 符号常量不为8KB时，等比例计算即可。
参数单位	8KB
参数含义	当background writer写入的数据量超过本参数值之后，尝试强制操作系统发送这些写入的数据到底层存储中。这么做会限制kernel中page cache的dirty data的数量，减少在checkpoint末尾发生fsync时发生卡顿的可能性。本参数可以在postgresql.conf或者server command line设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

Statistics / Monitoring

log_statement_stats

参数名称	log_statement_stats
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	在本参数设置为on的情况下，pg会向运行日志中写入系统性能统计信息。本参数on时，其他参数（log_parser_stats 、log_planner_stats、log_executor_stats）必须设置为off在本参数设置为on的情况下，当前台发出SQL语句时（如下以psql中执行select * from pg_database为例子），在pg的运行日志中，会有如下的输出：2021-03-31 09:55:21.621 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,6,"",2021-03-31 09:51:32 CST,3/0,0,LOG,00000,"QUERY STATISTICS","! system usage stats: ! 0.000657 s user, 0.000000 s system, 0.000657 s elapsed ! [0.003464 s user, 0.001403 s system total] ! 6288 kB max resident size ! 0/8 [0/24] filesystem blocks in/out ! 0/128 [0/1295] page faults/reclaims, 0 [0] swaps ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent! 0/0 [11/2] voluntary/involuntary context switches",,,,,"select *from pg_database;",,,"psql","client backend"

是否可session级修改	是
修改后何时生效	Session级修改立即生效；非session级修改reload生效，仅仅superuser能修改本参数

log_parser_stats

参数名称	log_parser_stats
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	在本参数设置为on的情况下，pg会向运行日志中写入解析统计信息。在本参数设置为on的情况下，log_statement_stats配置参数无法设置为on，设置为on时会有如下的错误： postgres=# alter system set log_statement_stats='on'; ERROR: invalid value for parameter "log_statement_stats": 1 DETAIL: Cannot enable "log_statement_stats" when "log_parser_stats", "log_planner_stats", or "log_executor_stats" is true. postgres=#在本参数设置为on的情况下，当前台发出SQL语句时（如下以psql中执行select * from pg_database为例子），在pg的运行日志中，会有如下的输出： 2021-03-31 10:16:25.962 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,41,"",2021-03-31 09:51:32 CST,3/3486,0,LOG,00000,"PARSER STATISTICS","! system usage stats: ! 0.000019 s user, 0.000000 s system, 0.000019 s elapsed ! [0.015481 s user, 0.001403 s system total] ! 7720 kB max resident size ! 0/0 [0/56] filesystem blocks in/out ! 0/0 [0/1673] page faults/reclaims, 0 [0] swaps ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent ! 0/0 [39/6] voluntary/involuntary context switches",,,,,"select * from pg_database;",,,"psql","client backend" 2021-03-31 10:16:25.962 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,42,"",2021-03-31 09:51:32 CST,3/3486,0,LOG,00000,"statement: select * from pg_database;",,,,,,,,,"psql","client backend" 2021-03-31 10:16:25.962 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,43,"",2021-03-31 09:51:32 CST,3/3486,0,LOG,00000,"PARSE ANALYSIS STATISTICS","! system usage stats: ! 0.000031 s user, 0.000000 s system, 0.000030 s elapsed ! [0.015565 s user, 0.001403 s system total] ! 7720 kB max resident size ! 0/0 [0/56] filesystem blocks in/out ! 0/0 [0/1673] page faults/reclaims, 0 [0] swaps ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent ! 0/0 [39/6] voluntary/involuntary context switches",,,,,"select * from pg_database;",,,"psql","client backend" 2021-03-31 10:16:25.962 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,44,"",2021-03-31 09:51:32 CST,3/3486,0,LOG,00000,"REWRITER STATISTICS","! system usage stats: ! 0.000003 s user, 0.000000 s system, 0.000003 s elapsed ! [0.015582 s user, 0.001403 s system total] ! 7720 kB max resident size ! 0/0 [0/56] filesystem blocks in/out ! 0/0 [0/1673] page faults/reclaims, 0 [0] swaps ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent! 0/0 [39/6] voluntary/involuntary context switches",,,,,"select * from pg_database;",,,"psql","client backend"
是否可session级修改	是，但是不能在log_statement_stats=on时设置本参数为on
修改后何时生效	Session级修改立即生效非session级修改reload生效，仅仅superuser能修改本参数

log_planner_stats

参数名称	log_planner_stats
参数值数据类型	boolean
默认值	off
取值范围	off和on
参数单位	

参数含义	在本参数设置为on的情况下，pg会向运行日志中写入规划器统计信息。在本参数设置为on的情况下，log_statement_stats配置参数无法设置为on，设置为on时会有如下的错误： postgres=# alter system set log_statement_stats='on'; ERROR: invalid value for parameter "log_statement_stats": 1 DETAIL: Cannot enable "log_statement_stats" when "log_parser_stats", "log_planner_stats", or "log_executor_stats" is true. postgres=#在本参数设置为on的情况下，当前台发出SQL语句时（如下以psql中执行select * from pg_database为例子），在pg的运行日志中，会有如下的输出： 2021-03-31 10:24:39.323 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,63,"",2021-03-31 09:51:32 CST,3/3491,0,LOG,00000,"PLANNER STATISTICS","! system usage stats: ! 0.000041 s user, 0.000006 s system, 0.000045 s elapsed ! [0.017384 s user, 0.002173 s system total] ! 7740 kB max resident size ! 0/0 [0/80] filesystem blocks in/out ! 0/0 [0/1674] page faults/reclaims, 0 [0] swaps ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent! 0/0 [55/10] voluntary/involuntary context switches",,,,,"select * from pg_database;",,,"psql","client backend"
是否可session级修改	是，但是不能在log_statement_stats=on时设置本参数为on
修改后何时生效	Session级修改立即生效;非session级修改reload生效，仅仅superuser能修改本参数

log_executor_stats

参数名称	log_executor_stats
参数值数据类型	Bool
默认值	off
取值范围	off和on
参数单位	
参数含义	在本参数设置为on的情况下，pg会向运行日志中写入执行器统计信息。在本参数设置为on的情况下，log_statement_stats配置参数无法设置为on，设置为on时会有如下的错误： postgres=# alter system set log_statement_stats='on'; ERROR: invalid value for parameter "log_statement_stats": 1 DETAIL: Cannot enable "log_statement_stats" when "log_parser_stats", "log_planner_stats", or "log_executor_stats" is true. postgres=#在本参数设置为on的情况下，当前台发出SQL语句时（如下以psql中执行select * from pg_database为例子），在pg的运行日志中，会有如下的输出： 2021-03-31 10:27:59.858 CST,"pg131","postgres",8896,"[local]",6063d5a4.22c0,71,"",2021-03-31 09:51:32 CST,3/3496,0,LOG,00000,"EXECUTOR STATISTICS","! system usage stats: ! 0.000031 s user, 0.000000 s system, 0.000030 s elapsed ! [0.018581 s user, 0.003320 s system total] ! 7744 kB max resident size ! 0/0 [0/104] filesystem blocks in/out ! 0/0 [0/1676] page faults/reclaims, 0 [0] swaps ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent! 0/0 [71/14] voluntary/involuntary context switches",,,,,"select * from pg_database;",,,"psql","client backend"
是否可session级修改	是，但是不能在log_statement_stats=on时设置本参数为on
修改后何时生效	Session级修改立即生效非session级修改reload生效，仅仅superuser能修改本参数

Client Connection Defaults / Shared Library Preloading

jit_provider

参数名称	jit_provider
数据类型	string
默认值	llvmjit
取值范围	
参数单位	

参数含义	本参数值指定了jit provider library的名称，本参数仅仅可以在pg启动时设置。若是指定了不存在的library，则jit不可用，但是不会有报错抛出。
是否可session级修改	否
修改后何时生效	重启PG instance生效

local_preload_libraries

参数名称	local_preload_libraries
数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	本参数指定在连接启动时被预加载的一个或多个共享库。多个库名称以逗号分隔，其中每个库名的解释与load命令相同。库名之间的空格会被忽略，若是需要在库名中包括空格或者逗号，请用双引号将库名引起来。参数值仅仅在连接开始时生效，后续更改无效。如果找不到指定的库，则连接会将失败(即:连接将无法登陆到数据库中)。本参数任何user都可以设置。因此，被加载的共享库仅仅限于在$libdir/plugins目录下。数据库管理员有责任确保该目录下的共享库的安全性。本参数的参数值可以明确指定目录，比如$libdir/plugins/mylib，或者仅仅指定library名称，比如mylib，mylib的作用等同于$libdir/plugins/mylib本参数的目的是允许非授权用户加载debugging或者performance-measurement库到特定session中，而无需load命令。为此，可以在客户端上使用设置了本参数的PGOPTIONS环境变量或者使用alter role set
是否可session级修改	是
修改后何时生效	Session级修改立即生效非session级修改reload生效

session_preload_libraries

参数名称	session_preload_libraries
数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	本参数值指定了在连接启动时被预加载的一个或者多个共享库。多个库名称以逗号分隔，其中每个库名的解释与load命令相同。库名之间的空格会被忽略，若是需要在库名中包括空格或者逗号，请用双引号将库名引起来。参数值仅仅在连接开始时生效，后续更改无效。如果找不到指定的库，则连接会将失败(即:连接将无法登陆到数据库中)。本参数的目的是允许加载debugging或者performance-measurement 库到特定session中，而无需load命令。比如，使用alter role set 语句设置本参数可以将auto_explain针对所有session生效。与shared_preload_libraries不同，在会话开始时（而不是在首次使用时）加载库没有太大的性能优势。但是，使用连接池时有一些优点。
是否可session级修改	可以，只有superuser可以改变本参数值
修改后何时生效	Session级修改立即生效非session级修改reload生效，reload之后，仅仅对新启动的session生效。

shared_preload_libraries

参数名称	shared_preload_libraries
数据类型	string

默认值	空字符串
取值范围	
参数单位	
参数含义	本参数值指定了在PostgreSQL数据库启动时被预加载的一个或者多个共享库。多个库名称以逗号分隔，其中每个库名的解释与load命令相同。库名之间的空格会被忽略，若是需要在库名中包括空格或者逗号，请用双引号将库名引起来。该参数只能在PostgreSQL数据库启动时设置，若是找不到指定的库，PostgreSQL数据库启动会失败。设置了本参数之后，新的backend process的启动时间会略有延长，即使这个backedn process不用这个共享库。因此，本参数推荐用于哪些在绝大多数session中使用的共享库。另外，由于本参数需要重启PostgreSQL数据库生效，因此，若是用于短期的debugging目的，请使用session_preload_libraries参数。请注意，在Windows主机上，在服务器启动时预加载库不会减少启动每个新服务器进程所需的时间。每个服务器进程将重新加载所有预加载库。但是，对于需要在postmaster启动时执行操作的库，本参数在windows主机上是很有用的。
是否可session级修改	否
修改后何时生效	重启PG instance生效

Lock Management

deadlock_timeout

参数名称	deadlock_timeout
数据类型	string
默认值	1000
取值范围	1到2147483647
参数单位	毫秒
参数含义	在检查是否存在deadlock条件之前，等待一个lock的时间长度。死锁检查是相对昂贵的，因此，pg不会在每次等待lock时都运行死锁检测。PostgreSQL会乐观的认为死锁在生产应用程序中并不常见，只需要等待一段时间后再检查死锁。这是您在实践中想要设置的最小值，在负载很重的pg中，可能需要提高本参数值。在理想情况下，本参数值应该超过典型的transaction 时间，以提高在PostgreSQL决定检查死锁之前将释放锁的几率。仅仅superuser可以修改本参数值。在log_lock_waits参数启用的情况下，deadlock_timeout参数值也决定了一个有关lock wait的 log message被写入运行日志之前的等待时间。如果你正在尝试调查锁定延迟(locking delays)，你可能希望设置比正常deadlock_timeout参数值更短的时间。
是否可session级修改	是(仅仅superuser可修改)
修改后何时生效	Session级修改立即生效(仅仅superuser)；非session级别修改Reload即可生效

max_locks_per_transaction

参数名称	max_locks_per_transaction
数据类型	integer
默认值	64
取值范围	10到2147483647
参数单位	

参数含义	共享锁表(shared lock table)追踪max_locks_per_transaction * (max_connections + max_prepared_transactions) 个lock objects，因此，一次只能锁定这么多不同的对象。本参数控制分配给每个事务的object lock的平均数量。单个事务可以锁定更多的对象，只要所有事务的锁都在锁表(lock table)中. 本参数值不是可以锁定的行数；可以锁定的行数是无限的。max_locks_per_transaction的默认值是64在历史上是被证明足够的，但是如果在多个事务中有涉及多个不同的表的SQL语句(比如一个有多个子表的父表的SQL语句),则可能提高该数值。本参数只能在PostgreSQL数据库启动时设置。当存在standby server时，standby server上的本参数值要大于或者等于master server上的本参数值，否则，查询语句在standby server上是不被允许的。
是否可session级修改	否
修改后何时生效	重启PG instance生效

max_pred_locks_per_transaction

参数名称	max_pred_locks_per_transaction
数据类型	Integer
默认值	64
取值范围	10到2147483647
参数单位	
参数含义	共享谓词锁表(shared predicate lock table)追踪max_pred_locks_per_transaction * (max_connections + max_prepared_transactions) 个lock objects，因此，一次只能锁定这么多不同的对象。本参数控制分配给每个事务的object lock的平均数量。单个事务可以锁定更多的对象，只要所有事务的锁都在锁表(lock table)中. 本参数值不是可以锁定的行数；可以锁定的行数是无限的。本参数默认值64在测试环境中是足够的，但是如果你的client在单个串行化事务中用到了多个不同的table，你可能需要加大本参数值。本参数只能在PostgreSQL数据库启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

max_pred_locks_per_relation

参数名称	max_pred_locks_per_relation
数据类型	Integer
默认值	-2
取值范围	-2147483648到2147483647
参数单位	
参数含义	本参数控制在将锁提升为覆盖整个relation之前，单个relation中能有多少pages或者tuples能被predicate-locked 。本参数值大于等于零时表示绝对限制，本参数值为负值时表示max_pred_locks_per_transaction/本参数值的绝对值。本参数可以在postgresql.conf设置也可以在server command line设置。
是否可session级修改	否
修改后何时生效	Reload生效

max_pred_locks_per_page

参数名称	max_pred_locks_per_page
数据类型	integer
默认值	2
取值范围	0到2147483647
参数单位	
参数含义	本参数控制在将锁提升为覆盖整个page之前，单个page中能有多少tuples能被predicate-locked。本参数可以在postgresql.conf设置也可以在server command line设置。
是否可session级修改	否
修改后何时生效	Reload生效

Resource Usage / Cost-Based Vacuum Delay

vacuum_cost_delay

参数名称	vacuum_cost_delay
数据类型	Real
默认值	0，零代表禁用了基于成本的vacuum delay特性。
取值范围	0到100
参数单位	毫秒
参数含义	vacuum_cost_limit参数值是一个累积vacuum io操作的成本值，当vacuum操作达到vacuum_cost_limit参数值之后，vacuum操作会停下来休眠一段时间，该段时间是vacuum_cost_delay参数值指定的，然后，PostgreSQL会重置计数器并继续vacuum操作。本参数的参数值建议尽量小，大值是没有帮助的。
是否可session级修改	是
修改后何时生效	session修改会立即生效；非session级修改reload即可生效。

vacuum_cost_page_hit

参数名称	vacuum_cost_page_hit
数据类型	Integer
默认值	1
取值范围	0到10000
参数单位	
参数含义	对shared buffer中发现的一个buffer进行vacuum的估计成本，本成本代表：锁定buffer pool的成本+查找shared hash table的成本+扫描page中内容的成本。
是否可session级修改	是
修改后何时生效	session修改会立即生效;非session级修改reload即可生效。

vacuum_cost_page_miss

参数名称	vacuum_cost_page_miss
数据类型	Integer
默认值	10
取值范围	0到10000

参数单位	
参数含义	对不在shared buffer中的一个buffer(只能从disk读取)进行vacuum的估计成本,本成本代表：锁定buffer pool的成本+查找shared hash table的成本+从disk中读取该block到buffer的成本+扫描page中内容的成本。
是否可session级修改	是
修改后何时生效	session修改会立即生效;非session级修改reload即可生效。

vacuum_cost_page_dirty

参数名称	vacuum_cost_page_dirty
数据类型	Integer
默认值	20
取值范围	0到10000
参数单位	
参数含义	Vacuum改变一个之前是干净的block的估计成本。本成本代表：额外的io操作以便将dirty block刷新到disk中。
是否可session级修改	是
修改后何时生效	session修改会立即生效；非session级修改reload即可生效。

vacuum_cost_limit

参数名称	vacuum_cost_limit
数据类型	Integer
默认值	200
取值范围	1到10000
参数单位	
参数含义	本参数值是一个累积vacuum io操作的成本值，当vacuum操作达到本参数值之后，vacuum操作会停下来休眠一段时间。
是否可session级修改	是
修改后何时生效	session修改会立即生效；非session级修改reload即可生效。

Replication / Sending Servers

max_wal_senders

参数名称	max_wal_senders
数据类型	Integer
默认值	10
取值范围	0到262143，0表示复制被禁用
参数单位	
参数含义	同时运行wal sender process的最大数量。运行wal sender process的场景包括流复制和以streaming模式进行base backup(译者注：其实还包括逻辑复制使用逻辑复制槽的场景，但是在PostgreSQL文档中并没有写明这一点)。本参数仅能在PostgreSQL启动时进行设置。配置参数waL_level必须设置为replica或者更高的值以便允许来自standby server的连接。在流复制环境中，在流复制备库上的该参数值至少要等于或者大于流复制主库上的该参数值，否则在备库上的query是不允许的。

是否可session级修改	否
修改后何时生效	重启PG instanc生效

max_replication_slots

参数名称	max_replication_slots
数据类型	Integer
默认值	10
取值范围	0到262143
参数单位	
参数含义	设置PostgreSQL能支持的复制槽的最大数量，本参数仅能在PostgreSQL启动时进行设置。当本参数值小于当前已经存在的replicat slot时，会导致PG instance无法启动。配置参数waL_level必须设置为replica或者更高的值以便允许replication slot被用到。
是否可session级修改	否
修改后何时生效	重启PG instanc生效

wal_keep_size

参数名称	wal_keep_size
数据类型	Integer
默认值	0，0表示PostgreSQL不会为流复制环境保留额外的wal segments，
取值范围	0到2147483647
参数单位	MB，若是设置时不带单位，则默认为MB
参数含义	指定在pg_wal目录中保留的wal segment file的最小大小，以便流复制环境中的standby server 从主库中fetch wal file。如果连接到发送服务器的standby server超过本参数值的大小，发送服务器可能会删除掉standby server仍然需要的wal segments，此时，复制连接会终止，下游的连接最终也会因此失败，若是wal进行了归档，standby server可以从归档中进行fetch segments，以便恢复流复制。本参数值仅仅设置保留在pg_wal中的wal segments的最小大小。系统可能需要保留更多的wal segments以便进行wal归档或者从检查点中恢复。本参数值仅仅在postgresql.conf中进行设置或者在server command line中进行设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

max_slot_wal_keep_size

参数名称	max_slot_wal_keep_size
数据类型	Integer
默认值	-1，-1表示复制槽保留不限数量的wal files。
取值范围	-1到2147483647
参数单位	MB
参数含义	用来指定复制槽保留pg_wal目录中wal file的最大大小。当一个复制槽的restart_lsn落后于current_lsn超过本参数值时，使用了复制槽的standby server不能再继续流复制，因为已经删除掉了需要的wal files
是否可session级修改	否

修改后何时生效	Reload即可生效

wal_sender_timeout

参数名称	wal_sender_timeout
数据类型	Integer
默认值	60000
取值范围	0到2147483647，0代表禁用timeout机制。
参数单位	毫秒，若是指定时不带单位，默认为为毫秒数
参数含义	当处于inactive状态多长时间(多长时间由本参数指定)之后，终止掉复制连接。这对于发送服务器检测standby crash或者网络中断很有用。当主备节点分布于不同的地理位置时，不同地理位置的节点使用不同的本参数值会带来管理上的灵活性。对于低延迟网络连接，较小的本参数值有助于快速的连接故障检测；对于地理位置较远的高延迟网络连接，较高的本参数值有助于判断standby server的健康状况。
是否可session级修改	是
修改后何时生效	Session修改立即生效，允许非superuser修改本参数值非session级修改reload生效。

Connections and Authentication / Authentication

authentication_timeout

参数名称	authentication_timeout
数据类型	Integer
默认值	60
取值范围	1到600
参数单位	秒，如果指定本参数值时不带单位，默认为秒
参数含义	允许完成连接认证的最长时间。如果客户端没有在本参数值指定的时间内完成连接认证，PostgreSQL数据库会关闭此次连接。本参数用于防止大量的客户端无限制的占用连接。本参数仅仅可以在postgresql.conf设置或者在server command line设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

password_encryption

参数名称	password_encryption
数据类型	enum
默认值	md5
取值范围	md5和scram-sha-256，写on也可以，on与md5有相同的作用。
参数单位	

参数含义	本参数决定了create role或者alter role语句中指定的密码的加密算法。请注意，老的客户端版本不支持SCRAM-SHA-256密码加密算法，具体请见下面的描述。有两个关键的标准去确定是否支持SCRAM密码加密算法。 ☒正在运行PostgreSQL 10及其更高版本 ☒你用来连接到PostgreSQL数据库的驱动有SCRAM兼容性。PostgreSQL社区已经提供了一个驱动清单(https://wiki.postgresql.org/wiki/List_of_drivers#Drivers)如果您的系统满足上述两个标准，您可以使用SCRAM密码加密算法。请注意：要把现有的环境从md5升级到scram-sha-256，可以在确保所有在用的客户端已经足以支持SCRAM之后，在postgresql.conf中设置password_encryption = 'scram-sha-256'，然后让所有用户设置新口令并且在pg_hba.conf中将认证方法说明改为scram-sha-256。
是否可session级修改	是
修改后何时生效	Session级修改立即生效；非session级修改reload生效

krb_server_keyfile

参数名称	krb_server_keyfile
数据类型	string
默认值	FILE:<sysconfdir>/krb5.keytab，注意,sysconfdir代表sysconfdir目录，该目录可以用pg_config --sysconfdir操作系统命令获得。
取值范围	
参数单位	
参数含义	设置服务器Kerberos key file的位置。如果本参数设置为空字符串(empty string),PostgreSQL会忽略这个空字符串，此时系统默认值会被使用。本参数值在postgresql.conf或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

krb_caseins_users

参数名称	krb_caseins_users
数据类型	bool
默认值	off，off代表大小写敏感
取值范围	
参数单位	
参数含义	本参数用于控制GSSAPI user names是否被视为大小写不敏感，本参数值仅能在postgresql.conf或者server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

db_user_namespace

参数名称	db_user_namespace
数据类型	bool
默认值	off
取值范围	off和on
参数单位	

参数含义	若本参数值设置为on，那么创建的用户名的格式是username@dbname，当client传递username时，@符以及database name会被附加到username的末尾，并且PostgreSQL数据库查询用户是查找username@dbname这个格式的用户。当你在SQL环境中使用username@dbname这个格式的用户名时，请使用引号引起来username@dbname。当本参数值设置为on时，你依然可以建立普通的全局users，在客户端指定username时加上@即可，当PostgreSQL查找username之前，@符号会被去掉。 本参数会导致客户端和服务器端的用户名称表示方法不同。认证始终检查服务器端的username。因此，认证方法必须被配置为服务器端的username。因为md5密码加密算法在客户端和服务器端均使用username作为salt，因此，md5密码加密算法不能与本参数值一起使用。 本参数值仅能在postgresql.conf或者server command line中设置。请注意，本特性的目的是在一个完整的解决方案找到之前作为临时措施，到那时，本参数会被remove掉。
是否可session级修改	否
修改后何时生效	Reload即可生效

Statistics / Query and Index Statistics Collector

track_activities

参数名称	track_activities
数据类型	boolean
默认值	on
取值范围	on 和off
参数单位	
参数含义	当设置为on时，当每个session中执行SQL语句或者命令时，PostgreSQL会收集信息（包括命令执行的时间信息）。收集的信息仅仅对superuser和执行命令所在session的owner可见。
是否可session级修改	是，仅限于superuser可进行session级修改
修改后何时生效	Session级修改立即生效;非session级修改reload生效

track_activity_query_size

参数名称	track_activity_query_size
数据类型	Integer
默认值	1024
取值范围	100到1048576
参数单位	Byte，当指定本参数不带单位时，默认是Bytes
参数含义	为每个active的session保留当前执行命令的文本的大小，该文本对应pg_stat_activity.query列。本参数仅仅能在PostgreSQL start时设置
是否可session级修改	否
修改后何时生效	重启PG instance生效

track_counts

参数名称	track_counts
数据类型	Bool
默认值	on
取值范围	on和off

参数单位	
参数含义	当设置为on时，针对database activity收集统计信息。Autovacuum进程需要这些收集的信息。
是否可session级修改	是，仅限于superuser可进行session级修改
修改后何时生效	Session级修改立即生效；非session级修改reload生效

track_io_timing

参数名称	track_io_timing
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	是否为database中的 io call/activity启用计时（即:io操作消耗了多少时间）。io计时信息体现在如下方面： pg_stat_database中 explain 带buffer时的输出中 pg_stat_statements中当设置为on时,PostgreSQL会重复查询操作系统的当前时间，这个可能会导致在某些平台下有大量开销。您可以使用pg_test_timing工具(该工具在bin目录下)来衡量系统的计时开销。仅superuser可以更改本参数值
是否可session级修改	是，仅限于superuser可进行session级修改
修改后何时生效	Session级修改立即生效非session级修改reload生效

track_functions

参数名称	track_functions
数据类型	enum
默认值	none，none的含义是禁用本功能
取值范围	none,pl,all
参数单位	
参数含义	启用追踪函数调用计数和消耗的时间。参数值pl的含义是仅仅对procedural-language函数进行追踪，参数值all的含义是追踪SQL和C语言函数。仅superuser可以更改本参数值
是否可session级修改	是，仅限于superuser可进行session级修改
修改后何时生效	Session级修改立即生效；非session级修改reload生效

stats_temp_directory

参数名称	stats_temp_directory
数据类型	string
默认值	pg_stat_tmp
取值范围	
参数单位	
参数含义	本参数值用于指定存储临时统计信息的目录。本参数值可以是绝对路径，也可以是相对于data directory的相对路径本参数值指定为基于RAM的文件系统会减少物理IO并提升性能。本参数仅能在postgresql.conf中设置或者在server command line设置。
是否可session级修改	否
修改后何时生效	Reload生效

File Locations

data_directory

参数名称	data_directory
数据类型	string
默认值	
取值范围	
参数单位	
参数含义	本参数值表示PostgreSQL的数据目录的位置
是否可session级修改	否
修改后何时生效	重启PG instance生效

config_file

参数名称	config_file
数据类型	string
默认值	在数据目录中
取值范围	
参数单位	
参数含义	本参数值指定了postgresql.conf的位置本参数值仅仅可以在postgres command line设置
是否可session级修改	否
修改后何时生效	重启PG instance生效

hba_file

参数名称	hba_file
数据类型	string
默认值	在数据目录中
取值范围	
参数单位	
参数含义	本参数值指定了pg_hba.conf的位置；本参数值仅仅可以在PostgreSQL启动时设置
是否可session级修改	否
修改后何时生效	重启PG instance生效

ident_file

参数名称	ident_file
数据类型	string
默认值	在数据目录中
取值范围	
参数单位	
参数含义	本参数值指定了pg_ident.conf的位置本参数值仅仅可以在PostgreSQL启动时设置
是否可session级修改	否

修改后何时生效	重启PG instance生效

external_pid_file

参数名称	external_pid_file
数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	指定了将postmaster的pid写入哪个文件本参数值仅仅可以在PostgreSQL启动时设置。 postgres=# alter system set external_pid_file='/home/pg131/postmasterpid.txt'; ALTER SYSTEM postgres=# exit [pg131@VM-0-8-centos ~]$ pg_ctl restart waiting for server to shut down…. done server stopped waiting for server to start….2021-04-09 10:09:21.776 CST [5977] LOG: redirecting log output to logging collector process 2021-04-09 10:09:21.776 CST [5977] HINT: Future log output will appear in directory "log". done server started [pg131@VM-0-8-centos ~]$ cd [pg131@VM-0-8-centos ~]$ ll total 136260 -rw-rw-r-- 1 pg131 pg131 0 Mar 10 20:47 12345aa drwxrwxr-x 2 pg131 pg131 4096 Apr 9 10:09 archive drwx------ 20 pg131 pg131 4096 Apr 9 10:09 data -rw-rw-r-- 1 pg131 pg131 72 Mar 23 10:34 para.sql drwxrwxr-x 6 pg131 pg131 4096 Dec 20 14:32 postgresql-13.1 -rw-r—r-- 1 pg131 pg131 139499520 Dec 20 14:07 postgresql-13.1.tar -rw-r—r-- 1 pg131 pg131 5 Apr 9 10:09 postmasterpid.txt drwxrwxr-x 7 pg131 pg131 4096 Dec 20 14:45 soft drwx------ 3 pg131 pg131 4096 Dec 20 16:04 ts1 [pg131@VM-0-8-centos ~]$ cat postmasterpid.txt 5977 [pg131@VM-0-8-centos ~]$ pg_ctl status pg_ctl: server is running (PID: 5977) /home/pg131/soft/bin/postgres[pg131@VM-0-8-centos ~]$
是否可session级修改	否
修改后何时生效	重启PG instance生效

Reporting and Logging / When to Log

log_min_messages

参数名称	log_min_messages
数据类型	enum
默认值	warning
取值范围	debug5,debug4,debug3,debug2,debug1,info,notice,warning,error,log,fatal,panic
参数单位	
参数含义	控制哪种message level写入系统日志。取值范围见上。在上面的取值范围中，前一个取值包括其后的取值。在上面的取值范围中，级别越在后边，写入系统日志中的messages就越少。注意: 本参数的参数值log的级别与client_min_messages配置参数的参数值log的级别不同。仅允许superuser可以修改本参数值。
是否可session级修改	是，此时仅限于superuser在session级别修改
修改后何时生效	Session级别修改立即生效；其他级别修改reload生效

log_min_error_statement

参数名称	log_min_error_statement

数据类型	enum
默认值	error，表示那些导致了errors、log messages、fatal errors、panics的SQL语句会被记录到日志中。
取值范围	debug5,debug4,debug3,debug2,debug1,info,notice,warning,error,log,fatal,panic
参数单位	
参数含义	控制导致何种级别错误的SQL语句内容被记录到日志中。为了关闭对错误SQL语句的记录，请设置被参数值为panic仅允许superuser可以修改本参数值。
是否可session级修改	是，此时仅限于superuser在session级别修改
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

log_min_duration_statement

参数名称	log_min_duration_statement
数据类型	integer
默认值	-1，-1表示禁用本功能。
取值范围	-1到2147483647
参数单位	毫秒，若是不指定单位的话。
参数含义	所有运行超过本参数值的SQL语句会打印到系统日志中。本参数对追踪未优化的SQL语句有很大帮助。本参数值为零表示打印所有的SQL语句。仅允许superuser可以修改本参数值。本参数会覆盖log_min_duration_sample参数，这意味持续时间超过本参数值的SQL语句将不进行采样，而是始终记录下来。对于使用扩展查询协议的客户端，Parse、Bind、Execute阶段会被各自独立记录下来。本参数与log_statement参数合用时，由于log_statement参数起作用而记录到log中的SQL文本不会被重复记录。在不使用syslog时，推荐使用log_line_prefix参数，以便记录PID或者Session ID。
是否可session级修改	是，此时仅限于superuser在session级别修改
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

log_min_duration_sample

参数名称	log_min_duration_sample
数据类型	Integer
默认值	-1，-1表示禁用本功能。
取值范围	-1到2147483647，参数值为零时，记录所有语句的sample
参数单位	毫秒
参数含义	举例说明，如果本参数值设置为100ms，那么运行时间超过100ms的SQL语句会被考虑进行采样。仅允许superuser可以修改本参数值。当流量太大而无法记录所有查询时，启用此参数会很有帮助。本参数的优先级比log_min_duration_statement参数优先级要低。这意味着持续时间超过log_min_duration_statement参数值的SQL语句不受采样限制，并且总是被记录下来。
是否可session级修改	是，此时仅限于superuser在session级别修改
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

log_statement_sample_rate

参数名称	log_statement_sample_rate
数据类型	Real
默认值	1

取值范围	0到1
参数单位	
参数含义	确定将记录的持续时间超过log_min_duration_sample参数值的的语句的比率。例如0.5意味着统计上每两个给定语句中就有一个被记录的机会。默认值为1.0，意味着记录所有采样语句。本参数值设置为0表示禁用sampled statement-duration logging。仅允许superuser可以修改本参数值。
是否可session级修改	是，此时仅限于superuser在session级别修改
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

log_transaction_sample_rate

参数名称	log_transaction_sample_rate
数据类型	Real
默认值	0，参数值为0表示不记录任何其他事务的语句
取值范围	0到1，参数值为1表示记录所有事物的所有语句
参数单位	
参数含义	设置除其他原因记录的语句之外，所有记录语句的事务的百分比，本参数适用于每个新事务，而不考虑其语句的持续时间。采样是随机的，例如0.1意味着在统计上，每十个交易中有一个机会记录任何给定的事务。本参数值有助于构建一个事务的采样。仅允许superuser可以修改本参数值。另外请注意：与所有statement-logging选项一样，本参数会显著增加系统负载。
是否可session级修改	是，此时仅限于superuser在session级别修改
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

Checkpoints

checkpoint_timeout

参数名称	checkpoint_timeout
数据类型	Integer
默认值	300
取值范围	30到86400
参数单位	秒
参数含义	设置两次automatic WAL checkpoints的时间间隔。增加本参数值会导致进行crash recovery的时间长度增大。本参数值仅能在postgresql.conf设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

checkpoint_completion_target

参数名称	checkpoint_completion_target
数据类型	Real
默认值	0.5
取值范围	0到1
参数单位	

参数含义	指定了checkpoint 完成的目标，即:checkpoint需要在checkpoint_timeout* checkpoint_completion_target的时间长度内完成。本参数值仅能在postgresql.conf设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

checkpoint_flush_after

参数名称	checkpoint_flush_after
数据类型	Integer
默认值	32，注意：linux平台下本参数默认值为32，Windows平台下本参数默认值为0
取值范围	0到256，如果BLCKSZ符号常量值不是8KB，请根据比例进行修改。
参数单位	8KB，若是指定参数值时不带单位，默认是BLCKSZ bytes，在不修改BLCKSZ符号常量值的情况下，该符号常量值为8KB。
参数含义	每当在执行检查点时写入的数据量超过本参数值时，会尝试强制操作系统向底层存储发出这些写入操作。这样会限制kernel page cache中dirty data的总量，减少了在checkpoint的末尾发出fsync命令时，或者当操作系统在后台以更大的批写入数据时，PostgreSQL暂停的可能性。这通常会大大减少事务延迟，但也有一些情况，特别是在工作负载大于shared_buffers，但小于操作系统page cache的情况下，性能可能会降低。本参数值仅可以在postgresql.conf中设置或者在server command line进行设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

checkpoint_warning

参数名称	checkpoint_warning
数据类型	integer
默认值	30
取值范围	0到2147483647，0会禁用warning。
参数单位	秒。若是指定本参数值不带单位，默认是秒
参数含义	如果由于填充WAL段文件而导致的检查点发生的时间间隔比本参数值更小（这表明应该加大max_wal_size配置参数值），则向服务器日志写入一条消息。当checkpoint_timeout参数值小于checkpoint_warning参数值时，warning不会产生。本参数值仅可以在postgresql.conf中设置或者在server command line进行设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

max_wal_size

参数名称	max_wal_size
数据类型	Integer
默认值	1024
取值范围	2到2147483647
参数单位	MB

参数含义	触发automatic checkpoints的最大wal 大小。这是软限制，wal size可以在如下情况下超过本参数值的限制：重负载，失败的archive_command命令，以及高的wal_keep_size参数值设置。增加本参数值会导致进行crash recovery的时间长度增大。本参数值仅能在postgresql.conf设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

min_wal_size

参数名称	min_wal_size
数据类型	Integer
默认值	80
取值范围	2到2147483647
参数单位	MB
参数含义	只要WAL Disk的使用低于本参数值，旧的wal文件总是在checkpoint发生时被循环使用，而不是删除wal文件。这可以用来确保保留足够的WAL空间来处理WAL使用中的峰值，例如在运行大型批处理作业时。此参数只能在postgresql.conf文件文件或在server command line上。
是否可session级修改	否
修改后何时生效	Reload即可生效

dynamic_library_path

参数名称	dynamic_library_path
数据类型	string
默认值	$libdir, $libdir代表的绝对路径请见pg_config --pkglibdir命令的输出
取值范围	
参数单位	
参数含义	如果需要打开可动态加载的模块，并且create function或LOAD命令中指定的文件名没有目录组件（即名称不包含斜杠），系统将在该路径中搜索所需的文件。本参数值必须是由冒号分隔(针对Linux平台)或者分号分隔(针对Windows平台)的绝对路径列表。如果本参数值以特殊字符串$libdir开头，那么编译后的PostgreSQL package library directory会取代$libdir, PostgreSQL package library directory请见pg_config --pkglibdir命令的结果。若是本参数值被设置为空字符串，表示自动path search功能会被关闭。superuser可以在运行时更改此参数，但这样做的设置只会持续到客户端连接结束，因此应将此方法保留用于开发目的。
是否可session级修改	是，此时仅限于superuser在session级别修改。
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

gin_fuzzy_search_limit

参数名称	gin_fuzzy_search_limit
数据类型	integer
默认值	0，0表示无限制
取值范围	0到2147483647
参数单位	

参数含义	GIN索引扫描返回的结果集大小的软上限。开发GIN索引的主要目的是在PostgreSQL中创建对高度可伸缩的全文搜索的支持，而且通常情况下，全文搜索会返回非常大的结果集。此外，当查询包含非常频繁的单词时，这种情况经常发生，因此大的结果集甚至没有用处。因为从磁盘读取许多元组并对它们进行排序可能需要花费大量时间，这对于生产来说是不可接受的。（请注意，索引搜索本身非常快。）为了便于此类查询的受控执行，GIN对返回的行数有一个可配置的软上限，这就是gin_fuzzy_search_limit配置参数。默认设置为0（表示无限制）。如果设置了非零限制，则返回的集是随机选择的整个结果集的子集。"软"意味着返回结果的实际数量可能与指定的限制有所不同，这取决于查询和系统随机数生成器的质量。从经验来看，以千为单位的本参数值（例如，5000-20000）工作正常。
是否可session级修改	否
修改后何时生效	Reload生效

tcp_keepalives_count

参数名称	tcp_keepalives_count
数据类型	
默认值	0，0表示取操作系统的默认值，
取值范围	0到2147483647
参数单位	
参数含义	指定在服务器与客户端的连接被视为已断开之前可以被丢弃的TCP keepalive messages.在一个基于Unix-domain socket的session中，本参数会被忽略，此时本参数值被视为零。本参数在Windows操作系统上不被支持，因此，在Windows操作系统上，本参数值必须为0。本参数仅仅支持符合如下条件的操作系统：那些支持TCP_KEEPCNT或者等价socket option的操作系统。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

tcp_keepalives_idle

参数名称	tcp_keepalives_idle
数据类型	integer
默认值	0，0表示取操作系统的默认值
取值范围	0到2147483647
参数单位	秒
参数含义	指定在没有网络活动的情况下，经过多长时间之后，操作系统应该向Client端发送TCP keepalive message.本参数仅仅支持符合如下条件的操作系统：那些支持TCP_KEEPIDLE或者等价socket option的操作系统。因此，在Windows以及其他操作系统中，本参数值必须为0，在一个基于Unix-domain socket的session中，本参数会被忽略，此时本参数值被视为0。在Windows操作系统上，本参数值为0时，本参数值将会被视为2小时，因为，Windows不提供读取系统默认值的方法。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

tcp_keepalives_interval

参数名称	tcp_keepalives_interval

数据类型	Integer
默认值	0，0表示取操作系统的默认值
取值范围	0到2147483647
参数单位	秒
参数含义	指定TCP keepalive message重传的时间间隔。本参数仅仅支持符合如下条件的操作系统：那些支持TCP_KEEPINTVL或者等价socket option的操作系统。因此，在Windows以及其他操作系统中，本参数值必须为零，在一个基于Unix-domain socket的session中，本参数会被忽略，此时本参数值被视为零。在Windows操作系统上，本参数值为零时，本参数值将会被视为1秒，因为，Windows不提供读取系统默认值的方法。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

tcp_user_timeout

参数名称	tcp_user_timeout
数据类型	Integer
默认值	0，0表示取操作系统的默认值
取值范围	0到2147483647
参数单位	毫秒
参数含义	指定在强制关闭TCP连接之前，传输的数据可能保持未确认状态的时间长度本参数仅仅支持符合如下条件的操作系统：那些支持TCP_USER_TIMEOUT或者等价socket option的操作系统。因此，在其他操作系统中，本参数值必须为零，在一个基于Unix-domain socket的session中，本参数会被忽略，此时本参数值被视为零。在Windows操作系统上，本参数不受支持，此时本参数值必须设置为零。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

Query Tuning / Genetic Query Optimizer

geqo

参数名称	geqo
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或者禁用genetic query optimization，生产环境中不应该将本参数设置为off。配置参数geqo_threshold提供了对GEQO更细粒度的控制。遗传查询优化器（GEQO）是一种使用启发式搜索进行查询计划的算法。这减少了用于复杂查询（那些包含许多关系的查询）的计划时间，但所产生的计划的成本有时不如普通穷举搜索算法所找到的计划。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

geqo_threshold

参数名称	geqo_threshold
数据类型	integer
默认值	12
取值范围	2到2147483647
参数单位	
参数含义	本参数是启用geqo的阈值，即:当from之后的关系数量至少有本参数值指定的个数时，才启用geqo。注意一个FULL OUTER JOIN只被计为一个FROM项。对于简单的SQL语句，通常最好使用常规的穷举搜索planner，但是对于具有许多表的查询，穷举搜索花费的时间太长，通常比执行次优计划的代价要长，因此，本阈值是管理geqo的一种方法。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

geqo_effort

参数名称	geqo_effort
数据类型	Integer
默认值	5
取值范围	1到10
参数单位	
参数含义	控制GEQO中计划时间和查询计划质量之间的权衡。此变量必须是1到10之间的整数。默认值为5。较大的值会增加生成查询计划所花费的时间，但也会增加选择高效查询计划的可能性。本参数实际上并不直接做任何事情；它只用于计算影响geqo行为的其他变量的默认值。如果愿意，可以手动设置其他参数。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

geqo_pool_size

参数名称	geqo_pool_size
数据类型	Integer
默认值	0，0表示PostgreSQL会根据SQL语句中表的数量以及geqo_effort参数值选择一个合适的值。
取值范围	0到2147483647
参数单位	
参数含义	控制GEQO使用的pool size，即:遗传群体中的个体数，至少是2，有用的值通常是100到1000. 参数值为0表示PostgreSQL会根据SQL语句中表的数量以及geqo_effort参数值选择一个合适的值。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

geqo_generations

参数名称	geqo_generations
数据类型	Integer
默认值	0

取值范围	0到2147483647
参数单位	
参数含义	控制GEQO使用的generations，即算法的迭代次数。它必须至少是一个，并且有用的值通常是100到1000。如果设置为零（默认设置），则PostgreSQL会根据geqo_pool_size选择合适的值。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

geqo_selection_bias

参数名称	geqo_selection_bias
数据类型	real
默认值	2
取值范围	1.5到2
参数单位	
参数含义	控制GEQO使用的选择偏差。选择偏差是population内部的选择压力
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

geqo_seed

参数名称	geqo_seed
数据类型	real
默认值	0
取值范围	0到1
参数单位	
参数含义	控制GEQO用于通过连接顺序搜索空间选择随机路径的随机数生成器的初始值，更改该值将更改所探索的连接路径集。并可能导致找到更好或更差的最佳路径
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

Write-Ahead Log / Recovery Target

recovery_target

参数名称	recovery_target
数据类型	string
默认值	空串
取值范围	immediate
参数单位	
参数含义	此参数指定恢复应在达到一致状态后尽快结束，即尽早结束。从联机备份还原时，这意味着在备份结束的时间点结束恢复。
是否可session级修改	否
修改后何时生效	本参数不支持修改。

recovery_target_name

参数名称	recovery_target_name
数据类型	string
默认值	空串
取值范围	
参数单位	
参数含义	此参数指定将继续恢复(recovery)到的命名还原点,该还原点由函数pg_create_restore_point()创建。
是否可session级修改	否
修改后何时生效	重启PG Instance生效

recovery_target_time

参数名称	recovery_target_time
数据类型	string
默认值	空串
取值范围	
参数单位	
参数含义	此参数指定将继续恢复(recovery)到的time stamp，精确的停止点受到recovery_target_inclusive配置参数影响。此参数的值是一个时间戳，其格式与timestamp with time zone数据类型所接受的格式相同，只是您不能使用时区缩写（除非在配置文件的前面设置了timezone_abbreviations变量）。首选样式是使用UTC的数字偏移量，或者您可以编写完整的时区名称，例如，Europe/Helsinki，而不是EEST
是否可session级修改	否
修改后何时生效	重启PG Instance生效

recovery_target_xid

参数名称	recovery_target_xid
数据类型	string
默认值	
取值范围	
参数单位	
参数含义	此参数指定将继续恢复(recovery)到的trasaction ID。请记住，虽然事务ID在事务开始时按顺序分配，但事务可以按不同的数字顺序完成。要恢复的事务是在指定事务之前提交的事务（也可以选择包括指定事务），精确的停止点受到recovery_target_inclusive配置参数影响。
是否可session级修改	否
修改后何时生效	重启PG Instance生效

recovery_target_lsn

参数名称	recovery_target_lsn
数据类型	string
默认值	空串
取值范围	

参数单位	
参数含义	此参数指定将继续恢复(recovery)到的LSN，精确的停止点受到recovery_target_inclusive配置参数影响。本参数使用系统数据类型pg_lsn解析。
是否可session级修改	否
修改后何时生效	重启PG Instance生效

recovery_target_inclusive

参数名称	recovery_target_inclusive
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当本参数设置为on时指定到达指定的recovery target之后停止当本参数设置为off时指定到达指定的recovery target之前停止本参数与如下参数配合使用：recovery_target_lsn、recovery_target_time、recovery_target_xid。本参数控制含有精确recovery target(恢复目标)的LSN、commit time、或者trasaction id是否被包括恢复(recovery)之内。
是否可session级修改	否
修改后何时生效	重启PG Instance生效

recovery_target_timeline

参数名称	recovery_target_timeline
数据类型	string
默认值	latest
取值范围	
参数单位	
参数含义	指定recovery进入一个特定的timeline，本参数值可以是一个数字的timeline id或者一个特定的值，值current的含义：recover到做base backup时的那个timeline；值latest表示recover到在archive中找到的最新的timeline，值latest适用于standby server。您通常只需要在复杂的重新恢复情况下设置本参数，在这种情况下，您需要返回到在时间点恢复之后达到的状态。
是否可session级修改	否
修改后何时生效	重启PG Instance生效

recovery_target_action

参数名称	recovery_target_action
数据类型	enum
默认值	pause
取值范围	pause,promote,shutdown
参数单位	

参数含义	指定了一旦达到recovery target，PostgreSQL该采取的操作。默认值pause表示恢复暂停(pause)，值promote表示恢复过程完成并且PostgreSQL将会启动以接受外部连接，值shutdown表示在达到recovery target之后会stop PostgreSQL。pause值的预期用途是允许对数据库执行查询，以检查此恢复目标是否是最理想的恢复点,pg_wal_replay_resume()函数表示paused state resumed，这意味着恢复走到了终点。若是recovery target不是期望的那个stopping point，那么请关闭PostgreSQL，改变recovery target到更为之后的一个target，然后重启recovery即可。
是否可session级修改	否
修改后何时生效	重启PG Instance生效

Query Tuning / Other Planner Options

default_statistics_target

参数名称	default_statistics_target
数据类型	integer
默认值	100
取值范围	1到10000
参数单位	
参数含义	为没有通过ALTER TABLE SET STATISTICS设置column-specific target的表列设置默认统计目标。较大的本参数值会增加analyze的时间，但也会提升planner估计的质量。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

constraint_exclusion

参数名称	constraint_exclusion
数据类型	enum
默认值	partition
取值范围	partition,on,off on表示检查所有表的约束 off表示不检查约束partition表示仅仅对继承的子表以及union all子查询检查约束。
参数单位	
参数含义	控制planner是否使用constraints优化查询,如果表的约束保证没有与查询匹配的行，table scan会被跳过。本参数经常与传统的继承树一起使用以提高性能，比如如下的例子： CREATE TABLE parent(key integer, …); CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent); CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent); … SELECT * FROM parent WHERE key = 2400;当本参数启用时，select语句就不会扫描child1000这个表。目前，默认情况下仅对经常用于通过继承树实现表分区的情况启用约束排除。为所有表启用它会带来额外的计划开销，这在简单查询中非常明显，而且对于简单查询通常不会产生任何好处。如果没有使用传统继承进行分区的表，则可能希望完全关闭它。（请注意，分区表的等效功能由单独的配置参数enable_partition_pruning控制。）
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

cursor_tuple_fraction

参数名称	cursor_tuple_fraction
数据类型	real
默认值	0.1
取值范围	0到1
参数单位	
参数含义	设置planner’s估计，本参数设置较少的值时，会使planner倾向于对cursors使用"fast start" plan，这会快速的检索前几行，而获取所有行可能需要很长时间。较大的值更强调总的估计时间。在最大设置为1.0时，游标的规划与常规查询完全相同，只考虑总的估计时间，而不考虑第一行的交付时间。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

from_collapse_limit

参数名称	from_collapse_limit
数据类型	integer
默认值	8
取值范围	1到2147483647
参数单位	
参数含义	如果生成的FROM列表包含的项目不超过本参数值，那么planner将会把子查询合并到上面的查询中。较小的值会减少计划时间，但可能会产生较差的查询计划。设置本参数值为geqo_threshold参数值或者更大可能会触发GEQO Planner的使用，进而导致非优化的执行计划。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

jit

参数名称	jit
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	确定jit编译是否被PostgreSQL使用。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

join_collapse_limit

参数名称	join_collapse_limit
数据类型	integer
默认值	8
取值范围	1到2147483647
参数单位	

参数含义	每当生成的items列表不超过本参数值时,planner就会将explicit join结构（FULL JOINs除外）重写为FROM items列表。较小的值会减少计划时间，但可能会产生较差的查询计划.默认情况下，此参数值设置为与from_collapse_limit参数值相同，这适用于大多数用途。将其设置为1可防止explicit join的任何重新排序。因此，查询中指定的显式连接顺序将是连接关系的实际顺序。因为查询规划器并不总是选择最佳的连接顺序，所以高级用户可以选择将此参数值临时设置为1，然后显式指定所需的连接顺序。设置本参数值为geqo_threshold参数值或者更大可能会触发GEQO Planner的使用，进而导致非优化的执行计划。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

parallel_leader_participation

参数名称	parallel_leader_participation
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	允许leader进程在Gather和Gather Merge node下执行查询计划，而不是等待worker进程。将此值设置为off可降低worker进程被阻塞的可能性，因为leader进程读取元组的速度不够快，但需要leader进程等待工作进程启动，然后才能生成第一个元组。Leader进程能够帮助或阻碍性能的程度取决于计划类型、worker数量和查询持续时间。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

force_parallel_mode

参数名称	force_parallel_mode
数据类型	enum
默认值	off
取值范围	off,on,regress off表示仅仅当预期能提升性能时，才使用parallel mode on表示对被视为safe的查询语句强制使用并行查询。regress表示除了有本参数值设置为"on"的相同的所有效果之外，还有一些旨在促进自动回归测试的附加效果
参数单位	
参数含义	是否使用parallel query功能。更具体地说，将此值设置为on将在任何查询计划的顶部添加一个Gather节点，这看起来是安全的，因此查询在并行worker进程中运行，即使并行worker进程不可用或无法使用，也将禁止在并行查询上下文中禁止的操作（如启动子事务），除非planner认为这将导致查询失败，如果设置此选项时出现故障或意外结果，则查询使用的某些函数可能需要标记为PARALLEL UNSAFE（或者，可能是PARALLEL RESTRICTED）通常，来自并行worker进程的消息包含一个context line，但是设置为regress会抑制该context line，以便输出与非并行执行中的输出相同。此外，通过此设置，添加到计划中的Gather nodes将隐藏在EXPLAIN的输出中，以便当本参数值设置为off时，输出与获得的结果相匹配。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

plan_cache_mode

参数名称	plan_cache_mode
数据类型	enum
默认值	auto
取值范围	auto,force_generic_plan,force_custom_plan
参数单位	
参数含义	可以使用自定义或generic的plan执行prepared statements（explicitly prepared或implicitly generated，例如通过PL/pgSQL），针对每一次的执行，自定义plan使用特定的一组参数值被重新生成，generic plans不依赖本plan_cache_mode参数值，可以被多次的执行重复使用。此外，使用generic plans会节省 planning time，但是，如果理想的plan强烈依靠本参数，那么，generic plan可能效率低下。这些选项之间的选择通常是自动进行的，但它可以被plan_cache_mode配置参数覆盖。在执行缓存的计划时，而不是在准备计划时，会考虑本参数值
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

Version and Platform Compatibility / Previous PostgreSQL Versions

array_nulls

参数名称	array_nulls
数据类型	bool
默认值	on，本参数值为on时表示允许输入包括NULL的数组值
取值范围	on和off
参数单位	
参数含义	本参数控制数组输入解析器是否将未加引号的NULL识别为指定NULL数组元素。8.2版本之前的PostgreSQL不支持数组中的null值，因此将null视为使用字符串值"null"指定普通数组元素。为了向后兼容需要旧行为的应用程序，可以关闭此参数。请注意，即使本参数值是off，也可以创建包含NULL的数组值
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

backslash_quote

参数名称	backslash_quote
数据类型	enum
默认值	safe_encoding
取值范围	safe_encoding,on,off on表示始终允许\' off表示始终拒绝'safe_encoding表示仅当客户端编码不允许在多字节字符中使用ASCII\时才允许.
参数单位	
参数含义	本参数用于设置字符串的引用符是否是'符号。SQL标准中是'',而PostgreSQL历史上也接受了'符号，但是，使用'会带来安全风险，这是因为，在一些客户端字符集编码中，存在多字节字符，其中最后一个字节在数字上等同于ASCII \.如果客户端没有正确的转义，那么SQL注入攻击是可能的。这种风险可以通过使PostgreSQL数据库拒绝引用符被backslash反斜杠转义的查询来防止。请注意，在标准一致性字符串文本中，无论如何，\仅仅表示\。此参数仅影响对非标准一致性文本的处理，包括转义字符串语法（E'…'）

是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

escape_string_warning

参数名称	escape_string_warning
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当设置为on时，表示当\出现在一个普通的字符串('…'这样的)中并且standard_conforming_strings参数值为off时，一个warning会发出。希望使用反斜杠作为转义的应用程序应该修改为使用转义字符串语法（E'…'）. 因为根据SQL标准，普通字符串的默认行为现在是将反斜杠视为普通字符. 可以启用本参数来帮助定位需要更改的代码
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

lo_compat_privileges

参数名称	lo_compat_privileges
数据类型	bool
默认值	off
取值范围	
参数单位	
参数含义	对large object，为权限检查启用面向backward的兼容性模式。在PostgreSQL9.0之前，large object并不具有访问权限，因此，large object总是能被所有的user进行读和写。设置本参数值为on将会禁用新权限的检查，这会与PostgreSQL9.0之前的版本保持兼容。
是否可session级修改	是，仅限于superuser进行修改
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

operator_precedence_warning

参数名称	operator_precedence_warning
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	启用此选项后，解析器将对自PostgreSQL 9.4以来由于运算符优先级的更改而可能已更改含义的任何构造发出警告。对于审核应用程序以查看优先级更改是否破坏了任何内容非常有用；但这并不意味着在生产环境中保持打开状态，因为它将警告某些完全有效、符合标准的SQL代码
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

quote_all_identifiers

参数名称	quote_all_identifiers
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	当数据库生成SQL语句时，即使标识符不是关键字，也会将标识符引用起来。这会影响explain的输出以及诸如pg_get_viewdef函数的结果，比如下面的例子： postgres=# select pg_get_viewdef('company_view'); -[RECORD 1]--+---------- pg_get_viewdef
SELECT company.id,+	company.name,
company.age	FROM company; postgres=# set quote_all_identifiers=on; SET postgres=# select pg_get_viewdef('company_view'); -[RECORD 1]--+---------- pg_get_viewdef
SELECT "company"."id",+	"company"."name",
"company"."age"	FROM "company"; postgres=#
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

standard_conforming_strings

参数名称	standard_conforming_strings
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	本参数控制普通字符串文本（'…'）是否按SQL标准中的指定按字面含义处理反斜杠.从PostgreSQL9.1开始，默认值为on。此参数的存在也可以被视为表示支持转义字符串语法（E'…'）如果应用程序希望反斜杠被视为转义字符，则应使用转义字符串语法
是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

synchronize_seqscans

参数名称	synchronize_seqscans
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	这允许对大表的顺序扫描彼此同步，以便并发扫描在大约相同的时间读取相同的块，从而共享I/O工作负载。启用本参数后，扫描可能从表的中间开始，然后"环绕"结尾以覆盖所有行，以便与正在进行的扫描活动同步。这可能导致没有ORDERBY子句的查询返回的行顺序发生不可预知的更改。将此参数设置为off可确保与PostgreSQL 8.3之前的行为，即顺序扫描始终从表的开头开始

是否可session级修改	是
修改后何时生效	Session级别修改立即生效，其他级别修改reload生效

Connections and Authentication / Connection Settings

listen_addresses

参数名称	listen_addresses
数据类型	string
默认值	localhost
取值范围	
参数单位	
参数含义	设置数据库服务器监听的hostname或者IP地址，监听多个hostname或者IP地址时，请用逗号隔开。*号表示监听所有可用的IP地址，0:0:0:0监听代表所有IPV4的地址，::代表监听所有IPV6的地址。当本参数值为空时，PostgreSQL不会监听任何的IP，此时，只有Unix-domain sockets才可以用于连接到数据库中。本参数控制哪些IP地址上接受连接尝试，这可以帮助防止在不安全的IP地址上重复出现恶意连接请求。该参数只能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

port

参数名称	port
数据类型	integer
默认值	5432
取值范围	1到65535
参数单位	
参数含义	PostgreSQL监听的TCP端口。请注意，PostgreSQL监听的所有IP地址都使用相同的端口号，本参数只能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

max_connections

参数名称	max_connections
数据类型	integer
默认值	100
取值范围	1到262143
参数单位	
参数含义	本参数值指定了PostgreSQL数据库的最大连接数。本参数只能在PostgreSQL启动时设置。在流复制备库上，必须将本参数值设置为与主库相同或者比主库参数值大,否则，后备服务器将不允许查询操作。
是否可session级修改	否
修改后何时生效	重启PG instance生效

superuser_reserved_connections

参数名称	superuser_reserved_connections
数据类型	integer
默认值	3
取值范围	0到262143
参数单位	
参数含义	为superusers保留的connection "slots"的数量，当PostgreSQL的活跃的并发连接的数量等于max_connections参数值减去superuser_reserved_connections参数值之后，仅仅可以面向super user建立新的连接，并且，replication connection也是建立不了的。本参数只能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

unix_socket_directories

参数名称	unix_socket_directories
数据类型	string
默认值	在Linux中默认值为/tmp；在Windows中默认值为空
取值范围	
参数单位	
参数含义	本参数指定了PostgreSQL监听客户端连接的Unix-domain socket(s)目录，多个目录可以用逗号隔开，两个目录之间的空格会被忽略，如果你想在目录名中包括逗号或者空格，请使用双引号引起来。本参数值为空时表示不监听任何的Unix-domain socket(s)目录，这意味着只有TCP/IP sockets可以被用来连接到PostgreSQL中。本参数只能在PostgreSQL启动时设置。在本参数值指定的目录下，会有名为s.PGSQL.nnnn的socket file，nnnn是PostgreSQL使用的端口号，在该目录下，还有一个名为.s.PGSQL.nnnn.lock的文件。这两个文件请不要手工删除。
是否可session级修改	否
修改后何时生效	重启PG instance生效

unix_socket_group

参数名称	unix_socket_group
数据类型	string
默认值	空字符串，空字符串表示使用启动PostgreSQL 的user的默认组。
取值范围	
参数单位	
参数含义	设置Unix-domain socket(s).的owning group(请注意：socket的owning user总是启动PostgreSQL的user)。本参数与unix_socket_permissions结合使用可以将其作为Unix-domain connections的额外访问控制机制。本参数只能在PostgreSQL启动时设置。本参数不支持在Windows上使用，在Windows上，本参数取任何值都会被忽略。
是否可session级修改	否
修改后何时生效	重启PG instance生效

unix_socket_permissions

参数名称	unix_socket_permissions
数据类型	integer
默认值	0777，0777表示任何人都可以连接
取值范围	0到511
参数单位	
参数含义	本参数用于设置Unix-domain socket(s)的访问权限。Unix-domain socket使用通常的Unix文件系统权限集。参数值预计是被chmod和umask系统调用接受的数字模式。（要使用惯用的八进制格式，数字必须以0（零）开头。）合理的参数值是0770（仅用户和组，另请参见unix_socket_group）和0700（仅用户）。请注意，对于Unix-domain socket，仅写许可权是重要的，因此设置或撤消读或执行许可权没有意义。本参数只能在PostgreSQL启动时设置。在完全忽略socket permissions的操作系统上（特别是自solaris10起的Solaris），此参数是不相关的。在此类操作系统上，可以通过将unix目录指向一个搜索权限仅限于所需访问用户的目录来实现类似的效果。
是否可session级修改	否
修改后何时生效	重启PG instance生效

bonjour

参数名称	bonjour
数据类型	boolean
默认值	off
取值范围	off和on
参数单位	
参数含义	本参数控制是否通过Bonjour机制通告PostgreSQL的存在。本参数只能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效。

bonjour_name

参数名称	bonjour_name
数据类型	string
默认值	空字符串。空字符串表示使用计算机名
取值范围	
参数单位	
参数含义	本参数用于设置Bonjour service name。当PostgreSQL软件没有编译Bonjour support时，本参数会被忽略。本参数只能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效。

Resource Usage / Asynchronous Behavior

effective_io_concurrency

参数名称	effective_io_concurrency
数据类型	integer

默认值	1
取值范围	0到1000，0表示禁止发布异步I/O请求
参数单位	
参数含义	有效利用disk子系统的并发disk io的数量。目前，本参数仅仅影响bitmap heap scans。对于HDD(机械硬盘)，本参数的起始值应该是PostgreSQL database用到的、组成raid 0 条带或者raid 1镜像的单独 disk drive的个数。但是，如果PostgreSQL数据库经常忙于并发会话中发出的多个查询，则较低的值可能足以使磁盘阵列保持繁忙。高于使磁盘保持忙碌的本参数值只会导致额外的CPU开销，固态盘(SSD)以及其他基于内存的存储可以经常处理很多并发请求，因此，此时最佳的值可能是几百。异步IO请求取决于一个有效的posix_fadvise函数，该函数在有些操作系统上是缺少的。在该函数缺少时，设置本参数为非零值会导致错误发生。在有些操作系统上(比如Solaris),本函数存在但是并没有实际的执行任何操作。在受到支持的操作系统上，本参数值的默认值为1，否则为零。本参数值可以被tablespace级别的同名参数值所覆盖。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效非session级修改reload生效

maintenance_io_concurrency

参数名称	maintenance_io_concurrency
数据类型	integer
默认值	10
取值范围	0到1000
参数单位	
参数含义	本参数类似于effective_io_concurrency参数，但本参数用于代表许多客户端会话执行的维护工作。在受到支持的操作系统上，本参数值的默认值为10，否则为零。本参数值可以被tablespace级别的同名参数值所覆盖。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效非session级修改reload生效

max_worker_processes

参数名称	max_worker_processes
数据类型	integer
默认值	8
取值范围	0到262143
参数单位	
参数含义	本参数值指定了PostgreSQL所能支持的并发worker进程的最大数量。本参数值仅能在PostgreSQL启动时设置。当运行在standby server环境中时，在standby server上的本参数值必须大于或者等于在master server上的本参数值。否则，query是不能运行在standby server上。当改变本参数值时，请考虑调整max_parallel_workers,参数，max_parallel_maintenance_workers参数，max_parallel_workers_per_gather参数
是否可session级修改	否
修改后何时生效	重启PG instance生效

max_parallel_workers_per_gather

参数名称	max_parallel_workers_per_gather

数据类型	integer
默认值	2
取值范围	0到1024，0表示禁用parallel query execution
参数单位	
参数含义	本参数指定了可以被单个Gather或者Gather Merge Node启动的worker进程的最大数量。Parallel Workers进程从由max_worker_processes配置参数建立的进程池中获取，受限于max_parallel_workers配置参数。请注意，请求的workers的数量在运行时并不一定实际可用，如果这个情况发生，执行计划会以少于期望个数的workers去运行，这可能是低效的。请注意，parallel queries可能比非parallel queries消耗更多的资源，因为每个工作进程是一个完全独立的进程，其对系统的影响与附加用户会话大致相同。在为此设置选择值时，以及在配置控制资源利用率的其他设置（如work_mem）时，都应考虑到这一点。诸如work_mem之类的资源限制单独应用于每个worker，这意味着所有进程的总利用率可能比任何单个进程的正常利用率高得多。例如，一个使用4个worker的并行查询可能会使用多达5倍的CPU时间、内存、I/O带宽。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效非session级修改reload生效

max_parallel_maintenance_workers

参数名称	max_parallel_maintenance_workers
数据类型	integer
默认值	2
取值范围	0到1024，0表示禁用由实用命令使用的parallel workers
参数单位	
参数含义	可以被单个实用命令启动的parallel worker的最大数量。当前，支持parallel workers的并行实用命令有如下几种： create index 建立B-Tree时； vacuum 不带full选项时； Parallel Workers进程从由max_worker_processes配置参数建立的进程池中获取，受限于max_parallel_workers配置参数。请注意，请求的workers的数量在运行时并不一定实际可用，如果这个情况发生，执行计划会以少于期望个数的workers去运行，这可能是低效的。注意，并行实用程序命令不应该比等效的非并行操作消耗更多的内存，这种策略不同于并行查询，在并行查询中资源限制通常适用于每个worker进程。并行实用命令将资源限制maintenance_work_mem视为应用于整个实用命令的限制，而不考虑并行工作进程的数量。然而，并行实用命令仍然可能消耗更多的CPU资源和I/O带宽。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效非session级修改reload生效

max_parallel_workers

参数名称	max_parallel_workers
数据类型	integer
默认值	8
取值范围	0到1024
参数单位	

参数含义	本参数值指定了PostgreSQL支持的parallel操作的worker的最大数量。当增加或者减少本参数值时，请考虑同时调整max_parallel_maintenance_workers配置参数和max_parallel_workers_per_gather配置参数。请注意：当本参数值高于max_worker_processes参数值时不会有任何作用。因为，因为parallel workers是从max_worker_processes配置参数所建立的工作进程池中获取的。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效非session级修改reload生效

backend_flush_after

参数名称	backend_flush_after
数据类型	integer
默认值	0，0表示禁用强制writeback
取值范围	0到256
参数单位	8KB
参数含义	每当单个backend进程写入的数据超过本参数值时，尝试强制操作系统向底层存储发出这些写入操作。这样做将限制内核page cache中脏数据的数量，减少在检查点结束时发出fsync时，或者当操作系统在后台以更大的批写入数据时暂停的可能性。这通常会大大减少事务延迟，但也有一些情况，特别是在工作负载大于共享缓冲区，但小于操作系统page cache的情况下，性能可能会降低。此参数可能在某些平台下无效。如果指定该值时没有单位，则将其作为块，即源码中的符号常量BLCKSZ个字节，通常为8KB。如果BLCKSZ不是8KB，则最大值与之成比例缩放。
是否可session级修改	是
修改后何时生效	Session级别修改立即生效非session级修改reload生效

old_snapshot_threshold

参数名称	old_snapshot_threshold
数据类型	intger
默认值	-1，-1代表禁用本特性，表示snapshot age 是无限制的。
取值范围	-1到86400
参数单位	分钟
参数含义	设置在使用snapshot时可以使用查询快照而不会出现"snapshot too old"错误的最短时间。处于dead状态的且超过本参数值的数据可以被vacuum掉。这有助于防止snapshot长期使用时出现表膨胀。为了防止由于清除快照可见的数据而产生不正确的结果，当快照早于本参数值并且快照用于读取自创建快照以来已修改的页面时，将生成一个错误。对生产环境有用的值可能从几个小时到几天不等。只允许使用较小的值（如0或1min），因为它们有时可能对测试有用。虽然允许高达60d的设置，但请注意，在许多工作负载中，在更短的时间内可能会出现极端膨胀或事务ID环绕(transaction ID wraparound)。启用本特性后，relation末尾释放的空间将无法释放给操作系统，因为这可能会删除检测"snapshot too old"条件所需的信息。分配给relation的所有空间都与该relation保持关联，以便仅在该relation中重用，除非显式释放（例如，使用VACUUM FULL）。此参数并不试图保证在任何特定情况下都会生成错误。事实上，如果正确的结果可以从一个已经物化的结果集中生成，则即使引用表中的底层行已被vacuum，也不会生成错误。某些表不能安全地提前vacuum，因此不受此设置的影响，例如系统目录(system catalogs)。对于这样的表，此设置既不会减少膨胀，也不会在扫描时产生"snapshot too old"错误的可能性。
是否可session级修改	否

修改后何时生效	重启PG instance生效

Replication / Standby Servers

primary_conninfo

参数名称	primary_conninfo
数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数值指定了连接到sending server的连接字符串(连接信息）.在本连接字符串中缺少的option，PostgreSQL会到环境变量中取寻找。若是环境变量中也没有，那么就用默认值。连接字符串应该指定如下信息： Sending Server的主机名或者IP地址、 Sending Server的端口号（若是跟standby server端口号不同） Username(该用户在sending-server端有合适的权限) Password：请注意，密码可以在primary_conninfo中提供，也可以在standby server上的~/.pgpass文件中提供（此时，是用replication作为database name）。请不要在primary_conninfo中指定database的名字本参数仅能在postgresql.conf文件或者server command line中设置。当在wal receiver process正在运行的情况下修改本参数值，wal receiver process会被发信号通知关闭然后再以新参数值去启动，除非primary_conninfo配置参数为空字符串。当PostgreSQL没有处于standby mode的情况下，本参数无效。
是否可session级修改	否
修改后何时生效	Reload即可生效

primary_slot_name

参数名称	primary_slot_name
数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数指定了当要连接到sending-server端时，要使用的sending-server端的已经存在的replication slot，用来控制sending-server端的wal removal。本参数仅能在postgresql.conf文件或者server command line中设置。当在wal receiver process正在运行的情况下修改本参数值，wal receiver process会被发信号通知关闭然后再以新参数值去启动，本参数在如下情况下没有作用：PostgreSQL没有处于standby mode或者primary_conninfo参数值没有设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

promote_trigger_file

参数名称	promote_trigger_file
数据类型	string

默认值	空字符串
取值范围	
参数单位	
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数值指定了一个trigger file，该文件用于standby server结束recovery状态进行角色提升。本参数值没有设置时，你可以使用pg_ctl promote或者使用函数pg_promote()进行角色提升。本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

hot_standby

参数名称	hot_standby
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数控制standby server是否允许运行只读查询语句。本参数仅能在server start时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

max_standby_archive_delay

参数名称	max_standby_archive_delay
数据类型	integer
默认值	30000
取值范围	-1到2147483647，-1表示允许standby server一直在等待直到冲突的query执行完毕。
参数单位	毫秒
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。当hot standby在被启用的状态下，本参数决定了standby server在取消掉standby中运行的且与WAL日志应用有冲突的查询语句之前的等待时间。本参数适用于从wal 归档读取wal data的情况。当不指定单位时，本参数的单位是毫秒。本参数仅能在postgresql.conf文件或者server command line中设置。 本参数值与查询语句在取消之前可以运行的最长时间不同。相反，本参数值是：一旦standby server从sending-server收到wal data，所允许的应用wal data的最长时间。因此，如果一个查询在一个wal segment之前导致了显著的延迟，那么在standby server再次赶上之前，后续冲突查询的容忍时间(宽限时间)将少得多。
是否可session级修改	否
修改后何时生效	Reload即可生效

max_standby_streaming_delay

参数名称	max_standby_streaming_delay

数据类型	integer
默认值	30000
取值范围	-1到2147483647，-1表示允许standby server一直在等待直到冲突的query执行完毕。
参数单位	毫秒
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。当hot standby在被启用的状态下，本参数决定了standby server在取消掉standby中运行的且与WAL日志应用有冲突的查询语句之前的等待时间。本参数适用于wal data通过流复制被接收的情况。当不指定单位时，本参数的单位是毫秒。本参数仅能在postgresql.conf文件或者server command line中设置。本参数值与查询语句在取消之前可以运行的最长时间不同。相反，本参数值是从主服务器接收到WAL数据后允许应用该数据的最长总时间，因此，如果一个查询导致了显著的延迟，那么在后备服务器再次赶上之前，后续冲突查询的容忍时间(宽限时间)将少得多
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_receiver_create_temp_slot

参数名称	wal_receiver_create_temp_slot
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数用于指定当永久的复制槽没有配置(复制槽配置请见配置参数primary_slot_name)的情况下，wal receiver是否在远程PG Cluster上创建一个临时的复制槽。本参数仅能在postgresql.conf文件或者server command line中设置。当wal receiver进程正在运行的同时本参数被改变时，wal receiver进程会关闭并重启以读取到新值。
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_receiver_status_interval

参数名称	wal_receiver_status_interval
数据类型	integer
默认值	10
取值范围	0到2147483，0表示完全禁止状态更新。
参数单位	秒
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数指定了备库上的wal receiver 进程向primary库或者sending-server库发送复制进度(replication progress)的最短时间频率。备库将报告它写到的wal的位置，它flush到disk的位置以及它应用wal的位置。本参数值是两次报告之间的最大时间间隔。每次写入(write)或刷新(flush)位置更改时都会发送更新，或者至少与本参数指定的频率相同。因此，应用的位置可能稍微落后于真实位置。本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否

修改后何时生效	Reload即可生效

hot_standby_feedback

参数名称	hot_standby_feedback
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。是否允许一个hot standby的备库发送关于正在备库执行的查询的feedback给primary库或者upstream库。本参数值可以消除由于cleanup records导致的query cancels，但是会导致primary库上发生膨胀。feedback信息的发送频率不会超过配置参数wal_receiver_status_interval。当是级联复制环境时，feedback会通过upstream(即:中间节点)会最终到达primary库。Standby节点除了向upstream反馈信息外，没有其他用途。本参数不会覆盖primary端的配置参数old_snapshot_threshold的行为。Standby端上超过primary端age阈值的snapshot会变为invalid，这导致standby端上的查询被取消。这是因为配置参数old_snapshot_threshold的目的是在时间上对dead row导致膨胀提供绝对的限制，否则会因为standby的配置违反该限制。
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_receiver_timeout

参数名称	wal_receiver_timeout
数据类型	integer
默认值	60000
取值范围	0到2147483647，0代表禁用本timeout检测机制。
参数单位	毫秒
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。当本备端与sending-server端或者primary端的inactive状态超过本参数值时，终止掉复制连接。本参数对receiving standby server检测primary node crash或者网络故障很有用。本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_retrieve_retry_interval

参数名称	wal_retrieve_retry_interval
数据类型	integer
默认值	5000
取值范围	1到2147483647
参数单位	毫秒

参数含义	本参数为备库参数，本参数在sending-server端会被忽略。本参数控制standby server遇到从所有源头(streaming replication、本地pg_wal目录、WAL Archive位置)获取wal而获取不到时的等待时间。本参数对处于in recovery的节点需要控制等待新的wal data可用的时间很有用。比如，在archive recovery中，通过降低本参数值，在检测一个新wal log file时，可以使recovery更快响应。在一个wal file产生很少的PostgreSQL中，增加本参数值会减少访问wal archive的请求次数。这在那些基础设施的访问次数会被考虑在内的云环境中非常有用。本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

recovery_min_apply_delay

参数名称	recovery_min_apply_delay
数据类型	integer
默认值	0，0代表无延迟。
取值范围	0到2147483647
参数单位	毫秒
参数含义	本参数为备库参数，本参数在sending-server端会被忽略。默认情况下，standby server会及时的restore 来自于sending server端的WAL 记录。数据的延迟copy可能很有用，它提供了纠正数据丢失错误的机会。本参数值指定了允许你指定延迟recovery的时间长度。比如，你设置本参数值为5分钟，那么只有当standby端的系统时间比master端报告的提交时间至少晚5分钟时，standby端才会重放每个事物的提交。
是否可session级修改	否
修改后何时生效	Reload即可生效

Query Tuning / Planner Cost Constants

seq_page_cost

参数名称	seq_page_cost
数据类型	real
默认值	1
取值范围	0到1.79769e+308
参数单位	
参数含义	本参数设置了顺序读取磁盘上的page的成本。在table以及index上的表空间级别的同名本参数可以覆盖配置参数中的该参数值。增大本参数值会使索引扫描看起来相对更昂贵
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

random_page_cost

参数名称	random_page_cost
数据类型	real
默认值	4
取值范围	0到1.79769e+308

参数单位	
参数含义	本参数设置了非顺序读取磁盘上的page的成本。在table以及index上的表空间级别的同名本参数可以覆盖配置参数中的该参数值。相对于seq_page_cost减少本参数值会使PostgreSQL更倾向于使用索引扫描。随机访问机械硬盘要比四次顺序访问还要昂贵。但是，使用较低的默认值(4.0)，因为大多数对磁盘的随机访问(如索引读取)都假定在cache中。默认值可以认为是将随机访问模型比顺序访问慢40倍，同时，期望90%的随机读取都是在cache中。如果您认为90%的缓存比率对于您的工作负载来说是一个不正确的假设，那么你可以增加本参数值以反映随机存储读取的真实成本。相应的，如果你的数据可能完全在cache中，比如在database大小小于服务器内存，减少本参数值是合适的。那些相对于顺序读具有低随机读成本的存储（比如固态驱动器），也可以使用较低本参数值（比如1.1）
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

cpu_tuple_cost

参数名称	cpu_tuple_cost
数据类型	real
默认值	0.01
取值范围	0到1.79769e+308
参数单位	
参数含义	本参数值设置在执行一个SQL语句时planner预计的处理每行数据的成本
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

cpu_index_tuple_cost

参数名称	cpu_index_tuple_cost
数据类型	real
默认值	0.005
取值范围	0到1.79769e+308
参数单位	
参数含义	本参数用于设置在索引扫描期间planner预计的处理每个index entry的成本。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

cpu_operator_cost

参数名称	cpu_operator_cost
数据类型	real
默认值	0.0025
取值范围	0到1.79769e+308
参数单位	
参数含义	本参数值设置在执行一个SQL语句时planner预计的处理执行的每个operator或者function的成本。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

parallel_setup_cost

参数名称	parallel_setup_cost
数据类型	real
默认值	1000
取值范围	0到1.79769e+308
参数单位	
参数含义	本参数值设置planner预估的启动parallel worker processes的成本。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

parallel_tuple_cost

参数名称	parallel_tuple_cost
数据类型	real
默认值	0.1
取值范围	0到1.79769e+308
参数单位	
参数含义	本参数值设置了从一个parallel worker进程将一个tuple传输到另外一个parallel worker进程的成本。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

min_parallel_table_scan_size

参数名称	min_parallel_table_scan_size
数据类型	integer
默认值	1024，默认情况下是8MB
取值范围	0到715827882
参数单位	8KB
参数含义	本参数设置了考虑使用parallel scan的table的数据(data)的最小大小，对于并行顺序扫描，table的数据(data)的最小大小总是等于table的大小。但是当index被用到时，table的数据(data)的最小大小通常会笑。如果本参数不带单位，取的是源码中符号常量BLCKSZ的值，通常为8KB。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

min_parallel_index_scan_size

参数名称	min_parallel_index_scan_size
数据类型	integer
默认值	64，默认情况下是512KB
取值范围	0到715827882
参数单位	8KB

参数含义	本参数设置了考虑使用parallel scan的index的数据(data)的最小大小，请注意：一个并行索引扫通常不会扫过整个index。本参数是planner认为扫描实际会涉及的页数，本参数也用来决定一个特定的索引是否可以参与并行vacuum。如果本参数不带单位，取的是源码中符号常量BLCKSZ的值。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

effective_cache_size

参数名称	effective_cache_size
数据类型	integer
默认值	524288
取值范围	1到2147483647
参数单位	8KB
参数含义	本参数设置了planner对单个SQL语句可用的disk cache的有效大小。这被纳入使用索引的成本估算中。值越大，使用索引扫描的可能性越大；值越小，使用顺序扫描的可能性越大。设置本参数时，应该同时考虑PostgreSQL的shared buffers和kernel的page cache用于PostgreSQL数据文件的部分，虽然这两个位置都可能存在一些数据。另外，还要考虑在不同表上预计的并发SQL语句数量，因为他们必须共享可用空间。本参数不影响PostgreSQL分配的shared buffers大小，也不会保留kernel的page cache。它仅用于估算目的。系统也不坚定在两次查询之间数据扔保留在磁盘cache中。如果本参数不带单位，取的是源码中符号常量BLCKSZ的值。通常为8KB。如果BLCKSZ不是8KB，则默认值按照定比例缩放。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

jit_above_cost

参数名称	jit_above_cost
数据类型	real
默认值	100000
取值范围	-1到1.79769e+308。-1表示禁用JIT编译
参数单位	
参数含义	当SQL语句的查询成本高于本参数值时，JIT编译会被激活，若是JIT编译被enable的情况下。执行JIT会消耗planning time但是可以加速查询执行。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

jit_inline_above_cost

参数名称	jit_inline_above_cost
数据类型	real
默认值	500000
取值范围	-1到1.79769e+308，-1表示禁用inline
参数单位	

参数含义	当SQL语句的查询成本高于本参数值时，JIT编译会尝试inline functions and operators。Inlining会增加planning time，但是会提升执行速度。将本参数值设置为低于jit_above_cost是没有意义的。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

jit_optimize_above_cost

参数名称	jit_optimize_above_cost
数据类型	real
默认值	500000
取值范围	-1到1.79769e+308，-1表示禁用expensive optimizations
参数单位	
参数含义	当SQL语句的查询成本高于本参数值时，JIT编译会尝试进行expensive optimizations。这种优化会增加planning time，但是会提升执行速度。将本参数值设置为低于jit_above_cost是没有意义的，将本参数值设置为高于jit_inline_above_cost也是没有益处的。
是否可session级修改	是
修改后何时生效	Session修改立即生效；其他修改Reload即可生效

autovacuum

autovacuum

参数名称	autovacuum
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	控制PostgreSQL是否允许autovacuum launcher daemon.autovacuum进程能正常工作的前提条件是配置参数track_counts设置为启用。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。即使本参数被禁用，若是需要防止事务id wraparound，PostgreSQL会启动autovacuum进程。
是否可session级修改	否
修改后何时生效	Reload即可生效

log_autovacuum_min_duration

参数名称	log_autovacuum_min_duration
数据类型	integer
默认值	0，0表示记录所有autovacuum操作到PostgreSQL运行日志中
取值范围	-1到2147483647， -1表示禁止autovacuum进程执行信息打印到PostgreSQL运行日志中。
参数单位	毫秒

参数含义	当autovacuum进程运行的时间长度大于等于本参数值时，autovacuum进程执行时的每个动作会被记录到PostgreSQL运行日志中。举例，若是本参数值设置为250ms，那么所有运行时间长度大于等于250ms的automatic vacuum和analyze操作会记录到PostgreSQL运行日志中。当本参数值被设置为-1之外的值时，若是autovacumm由于冲突所或者并发dropped relation的原因被skipped掉，一条信息会打印到PostgreSQL运行日志中。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_max_workers

参数名称	autovacuum_max_workers
数据类型	integer
默认值	3
取值范围	1到262143
参数单位	
参数含义	本参数设置了同时可以运行的autovacuum worker进程的最大个数。本参数仅仅可以在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

autovacuum_naptime

参数名称	autovacuum_naptime
数据类型	integer
默认值	60
取值范围	1・到2147483
参数单位	秒
参数含义	本参数指定了两次autovacuum进程运行的最小时间间隔。在每次autovacuum进程运行时，autovacuum守护进程检查数据库并针对需要的表发出vacuum和analyze命令。本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_vacuum_threshold

参数名称	autovacuum_vacuum_threshold
数据类型	integer
默认值	50
取值范围	0到2147483647
参数单位	
参数含义	本参数用于设置作为vacuum操作候选表的条件，即:更新或者删除掉多少条记录可以被作为vacuum操作的候选表。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。

是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_vacuum_insert_threshold

参数名称	autovacuum_vacuum_insert_threshold
数据类型	integer
默认值	1000
取值范围	-1到2147483647， -1表示即使有本参数值数量的insert记录，autovacuum守护进程也不会针对该表触发vacuum操作。
参数单位	
参数含义	本参数用于设置作为vacuum操作候选表的条件，即:插入多少条记录可以触发针对该表的vacuum。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_analyze_threshold

参数名称	autovacuum_analyze_threshold
数据类型	integer
默认值	50
取值范围	0到2147483647
参数单位	
参数含义	本参数用于设置作为analyze操作候选表的条件，即:插入、更新、删除多少条记录可以触发针对该表的analyze。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_vacuum_scale_factor

参数名称	autovacuum_vacuum_scale_factor
数据类型	real
默认值	0.2，0.2表示表大小的20%
取值范围	0到100
参数单位	
参数含义	本参数设置了表大小的一个比率，当决定一个表是否触发vacuum操作时，该比率会被配置参数autovacuum_vacuum_threshold考虑在内。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_vacuum_insert_scale_factor

参数名称	autovacuum_vacuum_insert_scale_factor
数据类型	real
默认值	0.2，0.2表示表大小的20%
取值范围	0到100
参数单位	
参数含义	本参数设置了表大小的一个比率，当决定一个表是否触发vacuum操作时，该比率会被配置参数autovacuum_vacuum_insert_threshold考虑在内。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_analyze_scale_factor

参数名称	autovacuum_analyze_scale_factor
数据类型	real
默认值	0.1
取值范围	0到100
参数单位	
参数含义	本参数设置了表大小的一个比率，当决定一个表是否触发vacuum操作时，该比率会被配置参数autovacuum_analyze_threshold考虑在内。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_freeze_max_age

参数名称	autovacuum_freeze_max_age
数据类型	integer
默认值	200000000
取值范围	100000到2000000000
参数单位	
参数含义	指定表的pg_class.relfrozenxid字段在强制执行vacuum操作以防止事务ID wraparound之前可以达到的最大年龄，请注意，即使autovacuum进程被禁用，PostgreSQL也会为了防止wraparound而启动autovacuum进程。Vacuum还允许从pg_xact子目录中删除旧文件，这就是为什么默认值是相对较低的2亿个事务的原因。此参数只能在服务器启动时设置，但是可以通过更改表存储参数来减少单个表的设置值。
是否可session级修改	否
修改后何时生效	重启PG instance生效

autovacuum_multixact_freeze_max_age

参数名称	autovacuum_multixact_freeze_max_age
数据类型	integer

默认值	400000000
取值范围	10000到2000000000
参数单位	
参数含义	指定表的pg_class. relminmxid字段在强制执行vacuum操作以防止multixact ID wraparound之前可以达到的最大年龄，请注意，即使autovacuum进程被禁用，PostgreSQL也会为了防止wraparound而启动autovacuum进程。Vacuuming multixacts还允许从pg_multixact/members和pg_multixact/offsets子目录中删除旧文件，这就是为什么默认值是相对较低的4亿个事务的原因。此参数只能在服务器启动时设置，但是可以通过更改表存储参数来减少单个表的设置值。
是否可session级修改	否
修改后何时生效	重启PG instance生效

autovacuum_vacuum_cost_delay

参数名称	autovacuum_vacuum_cost_delay
数据类型	real
默认值	2
取值范围	-1到100，-1表示使用配置参数vacuum_cost_delay的参数值。
参数单位	毫秒
参数含义	本参数是在autovacuum中vacuum操作用到的cost delay值。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

autovacuum_vacuum_cost_limit

参数名称	autovacuum_vacuum_cost_limit
数据类型	integer
默认值	-1，-1表示使用配置参数vacuum_cost_limit的参数值。
取值范围	-1到10000
参数单位	
参数含义	本参数是在autovacuum中vacuum操作用到的cost limit值。请注意，如果有多个autovacuum workers，则本参数值按比例分布在运行中的autovacuum workers之间，以便每个autovacuum worker该值的总和不超过本参数值。本参数仅能在postgresql.conf文件或者server command line中设置。本参数可以在表级设置，在表级设置的优先级高于配置参数中的优先级。
是否可session级修改	否
修改后何时生效	Reload即可生效

Connections and Authentication / SSL

ssl

参数名称	ssl

数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	是否启用ssl连接
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_ca_file

参数名称	ssl_ca_file
数据类型	string
默认值	空字符串，空字符串表示没有CA File被加载，并且Client身份验证不会被执行。
取值范围	
参数单位	
参数含义	本参数指定了包括SSL server certificate authority (CA)的文件名。本参数值是相对于data directory的相对路径，本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_cert_file

参数名称	ssl_cert_file
数据类型	string
默认值	server.crt
取值范围	
参数单位	
参数含义	本参数指定了包括SSL server certificate的文件名。本参数值是相对于data directory的相对路径，本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_crl_file

参数名称	ssl_crl_file
数据类型	string
默认值	空字符串，空字符串表示没有CRL文件被加载。
取值范围	
参数单位	
参数含义	本参数指定了包括SSL server certificate revocation list (CRL)的文件名。本参数值是相对于data directory的相对路径，本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否

修改后何时生效	Reload即可生效

ssl_key_file

参数名称	ssl_key_file
数据类型	string
默认值	server.key
取值范围	
参数单位	
参数含义	本参数指定了包括SSL server private key的文件名。本参数值是相对于data directory的相对路径，本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_ciphers

参数名称	ssl_ciphers
数据类型	string
默认值	HIGH:MEDIUM:+3DES:!aNULL
取值范围	
参数单位	
参数含义	设置允许被SSL连接使用的SSL ciphers suites的清单。有关此设置的语法和支持值的列表，请参阅OpenSSL package中的ciphers manual page。有使用TLS版本1.2及更低版本的连接才会受到影响。当前没有控制TLS版本1.3连接使用的密码选择的设置。默认值通常是一个合理的选择，除非您有特定的安全要求。本参数仅能在postgresql.conf文件或者server command line中设置。下面解释一下默认值的含义： HIGH：使用HIGH Group的Cipher suites（例如，AES、Camellia、3DES）MEDIUM：使用MEDIUM Group的Cipher suites（例如，RC4, SEED） +3DES：针对HIGH的OpenSSL默认排序是存在问题的，因为3DES排序高于AES 128.这是错误的，因为3DES比ASE128提供了少的安全性，并且3DES更慢。在所有除了HIGH和MEDIUM的ciphers外，+3DES进行了重新排序 !aNULL:禁用掉没有经过身份验证的匿名cipher suites。此类cipher suites易于遭受中间人攻击，因此不应该使用。可用的cipher suites在不同OpenSSL版本中的细节差异很大。使用操作系统命令openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL’ 可以看到当前安装的OpenSSL库的实际细节。请注意：这个list会在运行时基于server key type被过滤
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_prefer_server_ciphers

参数名称	ssl_prefer_server_ciphers
数据类型	bool
默认值	on
取值范围	on和off
参数单位	

参数含义	指定是否优先使用server端的SSL cipher，还是使用client端的。本参数仅能在postgresql.conf文件或者server command line中设置。更老的PostgreSQL版本没有本参数，更老的PostgreSQL版本总是优先使用client端的SSL cipher。本参数仅能在postgresql.conf文件或者server command line中设置。本参数的目的是用于向后兼容性。优先使用server端通常更好，因为更可能的是server端的配置是适当的
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_ecdh_curve

参数名称	ssl_ecdh_curve
数据类型	string
默认值	prime256v1
取值范围	
参数单位	
参数含义	指定要在ECDH密钥交换中使用的曲线的名称。它需要所有连接的客户端支持。它不需要与服务器的椭圆曲线密钥使用的曲线相同。本参数仅能在postgresql.conf文件或者server command line中设置。最常见曲线的OpenSSL名称是：prime256v1（NIST P-256）、secp384r1（NIST P-384）、secp521r1（NIST P-521）。可用曲线的完整列表可以通过操作系统命令openssl ecparam -list_curves显示。但并不是所有这些都可以在TLS中使用。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_min_protocol_version

参数名称	ssl_min_protocol_version
数据类型	enum
默认值	TLSv1.2
取值范围	{TLSv1,TLSv1.1,TLSv1.2,TLSv1.3}
参数单位	
参数含义	设置使用的SSL/TLS协议的最小版本。OpenSSLLibrary的更老的版本不支持所有的值。如果选择了不支持的设置，将引发错误。TLS 1.0之前的协议版本（即SSL version 2和3）始终处于禁用状态。默认值是TLSv1.2，该默认值是满足本文档编写时的工业最佳实践。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_max_protocol_version

参数名称	ssl_max_protocol_version
数据类型	enum
默认值	空字符串，空字符串意味着运行任何的协议版本。
取值范围	{"",TLSv1,TLSv1.1,TLSv1.2,TLSv1.3}
参数单位	

参数含义	设置使用的SSL/TLS协议的最大版本，这对测试或者有些组件使用新版本协议出现问题是有用的。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_dh_params_file

参数名称	ssl_dh_params_file
数据类型	string
默认值	空字符串，空字符串意味着使用默认DH参数编译。
取值范围	
参数单位	
参数含义	指定包含Diffie-Hellman参数的文件名，该参数用于所谓的临时DH SSL密码族。如果攻击者设法破解众所周知的编译在DH中的参数，那么使用自定义DH参数可以减少暴露。您可以使用操作系统命令openssl dhparam-out dhparams.pem 2048创建自己的DH参数文件本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_passphrase_command

参数名称	ssl_passphrase_command
数据类型	string
默认值	空字符串，空字符串表示使用内置的提示机制。
取值范围	
参数单位	
参数含义	设置需要获取用于解密SSL文件（如私钥）的密码短语时要调用的外部命令。命令必须将密码短语打印到标准输出，并以代码0退出。在参数值中，%p替换为提示字符串(写%%表示文本值%)。请注意提示字符串可能包含空格，因此请确保引用足够的引号。命令实际上不必提示用户输入密码短语，它可以从一个文件中读取它，从一个keychain工具中获取它，或者类似的。由用户来确定所选机制是否足够安全本参数仅能在postgresql.conf文件或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

ssl_passphrase_command_supports_reload

参数名称	ssl_passphrase_command_supports_reload
数据类型	bool
默认值	off，off表示在reload时，ssl_passphrase_command配置参数会被忽略，并且在passphrase被需要时，SSL configuration不会被reload
取值范围	on和off
参数单位	

参数含义	当一个key file需要passphrase时，本参数决定由ssl_passphrase_command配置参数设置的passphrase command在configuration reload期间是否被调用本参数仅能在postgresql.conf文件或者server command line中设置。该设置适用于需要TTY进行提示的命令，该命令在PostgreSQL运行时可能不可用。例如，如果密码短语是从文件获取的，则将本参数设置为on可能是合适的
是否可session级修改	否
修改后何时生效	Reload即可生效

Resource Usage / Memory

shared_buffers

参数名称	shared_buffers
数据类型	integer
默认值	16384，即：128MB
取值范围	16到1073741823
参数单位	8KB，若是指定本参数值时不带单位，则取自源码中的符号常量BLCKSZ大小，默认是8KB。非默认的BLCKSZ大小会改变本参数的最小值。本参数值仅仅能在PostgreSQL启动时设置。建议将OS物理内存的25%分配给本参数值。因为PostgreSQL依赖于操作系统的cache（也就是文件系统的page cache），因此将超过40%的RAM分配给本参数值的情况不太可能比分配较小的缓冲区效果更好。较大的本参数值通常需要相应的增加max_wal_size, 以便在较长时间内分散写入大量新数据或更改数据的过程。
参数含义	
是否可session级修改	否
修改后何时生效	重启PG instance生效

huge_pages

参数名称	huge_pages
数据类型	enum

默认值	try，try的含义是PostgreSQL会尝试使用huge pages，如果尝试失败将返回默认值。on的含义是请求huge pages失败将阻止PostgreSQL启动，off的含义是不会启用huge pages。目前，本参数值仅支持Linux和Windows平台。当在其他平台下本参数值设置为try时，本参数值会被忽略。 huge pages的使用导致了更小的page tables以及耗费在内存管理方面的更小的CPU时间，这提高了性能。在Windows平台上，huge pages被称之为large pages。开启large pages的步骤： 0.安装PostgreSQL 数据库 1.关闭windows uac控制面板—系统和安全— 更改用户账户控制设置—改为’从不通知'，点击确定。 2.在windows组策略编辑器，赋予运行PostgreSQL的操作系统用户’锁定内存页’的权力（Lock Pages in Memory）计算机配置\Windows设置\安全设置\本地策略\用户权利分配\下的"锁定内存页"这个策略。 3.将windows服务中的postgresql-x64-11这个服务改为手动启动。 4.重新启动OS 5.设置postgresql.conf的配置参数huge_pages=on 6.设置postgresql.conf的配置参数shared_buffers=2048MB 7.以管理员身份打开cmd窗口，执行pg_ctl start -D e:\postgresql\11\data启动PostgreSQL 8.验证是否启用请注意：当启用large pages时，不能用Windows服务中的PostgreSQL服务启动postgresql。请注意，本参数设置仅影响main shared memory area. Linux、FreeBSD和Illumos等操作系统也可以自动使用huge pages（也称为"super" pages 或者 "large" pages）进行正常内存分配，而无需PostgreSQL的显式请求。在Linux上，这被称为"transparent huge pages"（THP）。众所周知，对于某些Linux版本的某些用户来说，该特性会导致PostgreSQL的性能下降，因此目前不鼓励使用它（与显式使用huge_pages）。
取值范围	{off,on,try}
参数单位	
参数含义	控制shared_buffers是否使用huge pages。
是否可session级修改	否
修改后何时生效	重启PG instance生效

temp_buffers

参数名称	temp_buffers
数据类型	integer
默认值	1024
取值范围	100到1073741823
参数单位	8KB
参数含义	设置每个session使用的temporary buffers的最大值，供临时表使用。指定本参数值时若是不带单位，以blocks为单位，取自源码中符号常量BLCKSZ,通常为8KB。若是BLCKSZ不为8KB，则本参数按照等比例变化。本参数值可以在单个session中更改，但只能在session中首次使用临时表之前更改；随后更改该值的尝试不会对该session产生任何影响。session将根据需要分配临时缓冲区，直到到达本参数值指定的限制。
是否可session级修改	是
修改后何时生效	Reload即可生效

max_prepared_transactions

参数名称	max_prepared_transactions
数据类型	integer
默认值	100

取值范围	0到262143，0代表禁用prepared-transaction特性。
参数单位	设置同时可以处于prepared状态的最大的事务个数。若是你不打算使用prepared transaction，请将本参数值设置为零以防止偶然建立prepared transaction。如果你使用prepared transaction，你可能希望本参数值与max_connections一样大。这样每个session都可以有一个prepared transaction pending。当有流复制环境是，你必须确保在备库上该参数值要大于等于在主库上的该参数值，否则，不能在备库上运行查询语句。
参数含义	
是否可session级修改	否
修改后何时生效	重启PG instance生效

work_mem

参数名称	work_mem
数据类型	integer
默认值	4096
取值范围	64到2147483647
参数单位	KB
参数含义	在写入磁盘临时文件之前，被查询操作（比如sort或者hash table）使用的内存最大值。若是指定本参数值时不带单位，那么是以KB为单位。请注意，针对复杂的查询，sort或者hash操作可能以parallel去运行，在开始向磁盘的临时文件写入数据之前，每个操作被允许最多使用本参数值对应的内存大小。同事，正在运行的多个session会并发执行这些操作。因此，使用的所有内存可以是本参数值的若干倍，当设置本参数值时，需要明确记住这个事实。Sort操作被如下使用：order by、distinct、merge joins；hash table被如下使用：hash joins、hash-base aggregation、在in子句中的基于hash的处理。 基于哈希的操作通常比基于排序的等效操作对内存可用性更加敏感。哈希表可用的内存是通过work_mem*hash_mem_multiplier来计算的。这使得基于哈希的操作使用的内存量可能超过通常的work_mem数量。
是否可session级修改	是
修改后何时生效	Session级修改立即生效其他修改Reload即可生效

hash_mem_multiplier

参数名称	hash_mem_multiplier
数据类型	real
默认值	1
取值范围	1到1000
参数单位	
参数含义	本参数值用于计算hash-base操作可以使用的最大内存。该最大内存是由work_mem*hash_mem_multiplier决定的。在经常有大量查询操作的环境中，考虑增加本参数值，特别是当简单的增加work_mem会导致内存压力时（内存压力通常以间歇性内存不足的形式出现）。设置我1.5或者2.0可能对混合负载有效。在work_mem参数值已经增加到40MB或者更多时，2.0-8.0或者更大范文内的更高设置可能会有效。
是否可session级修改	是
修改后何时生效	Session级修改立即生效其他修改Reload即可生效

maintenance_work_mem

参数名称	maintenance_work_mem
数据类型	integer
默认值	65536
取值范围	1024到2147483647
参数单位	KB
参数含义	指定用于维护操作(maintenance operations)的最大可用的内存。维护操作包括：vacuum,create index, ALTER TABLE ADD FOREIGN KEY.若指定本参数值时不带单位，那默认的单位是KB。较大的本参数值有助于提升vacuuming以及restore database时的性能。请注意：当autovacuum运行时，最多有本参数值* autovacuum_max_workers的内存被使用，因此，请小心设置本参数值，不要设置太高。可以使用配置参数autovacuum_work_mem来单独控制autovacuum时的内存使用。
是否可session级修改	是
修改后何时生效	Session级修改立即生效其他修改Reload即可生效

autovacuum_work_mem

参数名称	autovacuum_work_mem
数据类型	integer
默认值	-1，-1表示使用maintenance_work_mem配置参数值
取值范围	-1到2147483647
参数单位	KB
参数含义	本参数用于设置每个autovacuum worker process使用的最大内存大小。若指定本参数值时不带单位，那默认的单位是KB。当vacuum的行为运行在其他上下文环境中时，本参数不生效。
是否可session级修改	否
修改后何时生效	Reload即可生效

logical_decoding_work_mem

参数名称	logical_decoding_work_mem
数据类型	integer
默认值	65536，即64MB
取值范围	64到2147483647
参数单位	KB
参数含义	本参数用于设置在解码后的变化写入到本地disk之前，逻辑解码使用的最大内存大小。本参数限制了逻辑流复制连接使用的内存大小。由于每个复制连接只使用此大小的单个buffer，并且通常不会有许多这样的连接(受到max_wal_senders配置参数限制)。因此可以将本参数值设置为比work_mem高，从而减少写入磁盘的解码变化量。
是否可session级修改	是
修改后何时生效	session级修改立即生效其他修改Reload即可生效

max_stack_depth

参数名称	max_stack_depth

数据类型	integer
默认值	2048
取值范围	100到2147483647
参数单位	KB
参数含义	本参数值指定PostgreSQL execution stack的最大安全深度。本参数值的理想设置是kernel执行的实际stack size limit（可以用操作系统命令ulimit -s或者ulimit -a进行查看）减去大约1MB的安全裕度。所以需要安全裕度，是因为并不是在服务器中的每个例程中都检查stack depth，而是只在可能递归的关键例程中检查stack depth。若指定本参数值时不带单位，那默认的单位是KB。本参数默认值为2MB，该值相对保守，不太可能有crash的风险。但是，它可能太小，无法执行复杂的函数。只有superuser才能更改此设置。本参数值设置为高于kernel的限制将意味着一个失控的递归函数可以crash掉一个单独的backend process。在PostgreSQL可以确定内核限制的平台上，服务器将不允许把本参数值设置为不安全的值。但是并非所有平台都提供该信息。因此建议谨慎选择本参数值。
是否可session级修改	是，仅限于superuser可进行session级修改。
修改后何时生效	session级修改立即生效其他修改Reload即可生效

shared_memory_type

参数名称	shared_memory_type
数据类型	enum
默认值	mmap
取值范围	{sysv,mmap,windows}, sysv,mmap适用于Linux平台；windows适用于windows平台。
参数单位	
参数含义	指定shared memory的实现方式。 mmap是指使用mmap分配的anonymous shared memory； sysv是指通过shmget分配的 System V shared memory ；windows是指Windows shared memory。通常不鼓励使用sysv值，因为它通常需要非默认的内核设置来允许较大的分配，而sysv值在任何平台上都不是默认的。
是否可session级修改	否
修改后何时生效	重启PG instance生效

dynamic_shared_memory_type

参数名称	dynamic_shared_memory_type
数据类型	enum
默认值	posix
取值范围	{posix,sysv,mmap,windows} 其中posix,sysv,mmap适用于Linux平台；windows适用于Windows平台
参数单位	

参数含义	指定PostgreSQL使用的dynamic shared memory的实现方式。posix是指使用shm_open分配的POSIX shared memory; sysv是指使用shmget分配的System V shared memory； windows是指Windows shared memory； mmap是指使用存在的data directory中的memory-mapped files模拟的shared memory。通常不鼓励使用mmap值，这在任何平台上都不是默认值，因为操作系统可能会反复将修改过的页面写回磁盘，从而增加系统I/O负载；但是，当pg_dynshmem子目录存储在RAM磁盘上，或者其他共享内存工具不可用时，它可能对调试有用。
是否可session级修改	否
修改后何时生效	重启PG instance生效

Reporting and Logging / Where to Log

log_destination

参数名称	log_destination
数据类型	string
默认值	stderr
取值范围	Linux平台下的取值有：stderr、csvlog、syslogWindows平台下的取值有：stderr、csvlog、eventlog
参数单位	
参数含义	PostgreSQL支持多种记录PostgreSQL消息的方法。包括stderr、csvlog、syslog，在windows平台下还有eventlog。当本参数值是一个列表时，请使用逗号分隔每个值。本参数仅能在postgresql.conf中设置或者在server command line中设置。当本参数值中含有csvlog时，日志条目被输出为csv格式（comma separated value），这便于程序加载log内容。需要将配置参数logging_collector设置为on才可以生成csv格式的log。当本参数值含有stderr或者csvlog时，current_logfiles文件被建立用来记录当前正在被logging collector进程使用的logfile的位置。这提供了一个查找当前正在使用的log的简便方法。 current_logfiles文件的内容如下（举例）: stderr log/postgresql.log csvlog log/postgresql.csv当新的logfile被建立或者log_destination被加载时，current_logfiles文件会被重建。当本参数值不含有stderr或者csvlog，以及logging collector被禁用时，current_logfiles文件会被删除掉。
是否可session级修改	否
修改后何时生效	Reload即可生效

logging_collector

参数名称	logging_collector
数据类型	bool
默认值	off
取值范围	on和off
参数单位	

参数含义	本参数会启用logging collector进程，该进程是一个background process，用来捕获发送到stderr的log messages并重定向这些信息到log file中。这个方法比记录到syslog中要有用。因为有些类型的messages不会显示在syslog的输出中（一个例子是dynamic-linker failure messages，另外一个是脚本产生的error messages，比如archive_command）.本参数仅仅可以在PostgreSQL启动时设置。logging collector设计为永不丢失消息。这意味着在负载极高的情况下，当logging collector落后时，服务器进程可能会在尝试发送额外日志消息时被阻止。相反，syslog更喜欢在无法写入消息时丢弃消息，这意味着在这种情况下它可能无法记录某些消息，但不会阻塞系统的其余部分。
是否可session级修改	否
修改后何时生效	重启PG instance生效

log_directory

参数名称	log_directory
数据类型	string
默认值	log
取值范围	
参数单位	
参数含义	在logging_collector配置参数设置为on的情况下，本参数用于确定log file被建立在哪个目录下。本参数值可以是绝对路径，也可以是相对于data directory的相对路径。本参数仅能在postgresql.conf中设置或者在server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

log_filename

参数名称	log_filename
数据类型	string
默认值	postgresql-%Y-%m-%d_%H%M%S.log
取值范围	
参数单位	
参数含义	在logging_collector配置参数设置为on的情况下，本参数用于确定所创建的log file的文件名。本参数值被视为strftime pattern，因此可以使用%-escapes指定随时间变化的文件名(请注意，如果存在任何依赖于时区的%-转义，则计算将在配置参数log_timezone指定的时区中完成。）支持的%-转义与开放组的strftime规范中列出的那些类似。请注意，系统的strftime不是直接使用的，因此特定于平台（非标准）的扩展不起作用。如果本参数值中不带转义符，你应使用log rotation工具以避免日志填充满整个磁盘。若是在log_destination配置参数中启用了csv格式的输出，那么会将.CSV附加到带时间戳的日志文件名.以创建CSV格式输出的文件名(如果日志文件名以.log结尾，则替换后缀。）本参数仅能在postgresql.conf中设置或者在server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

log_file_mode

参数名称	log_file_mode

数据类型	string
默认值	0600， 0600表示只有PostgreSQL的owner可以读取或者写入log files
取值范围	0到511
参数单位	
参数含义	在logging_collector配置参数设置为on的情况下，在Unix/Linux系统中本参数用于设置logfile的权限（在Microsoft Windows平台上，本参数被忽略）。 参数值应该是以chmod和umask系统调用接受的格式指定的数字模式。 0600表示只有PostgreSQL的owner可以读取或者写入log files.其他通常有用的设置是0640，0640允许owner的group的成员去读取log files。请注意，要使用这样的设置，您需要更改log_directory目录以将文件存储在data directory目录之外的某个位置。无论如何，让日志文件具有广泛的可读性是不明智的，因为它们可能包含敏感数据本参数仅能在postgresql.conf中设置或者在server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

log_rotation_age

参数名称	log_rotation_age
数据类型	integer
默认值	1440
取值范围	0到35791394，0表示禁用掉基于time的new log file创建。
参数单位	分钟
参数含义	在logging_collector配置参数设置为on的情况下，本参数确定了一个单独的log file能使用的最长时间。在将此数量的数据发送到日志文件后，将创建一个新的日志文件。若指定本参数值时不带单位，那默认的单位是分钟。默认值是24小时。本参数仅能在postgresql.conf中设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

log_rotation_size

参数名称	log_rotation_size
数据类型	integer
默认值	10240
取值范围	0到2097151，0表示禁用掉基于size的new log file创建
参数单位	
参数含义	在logging_collector配置参数设置为on的情况下，本参数确定了一个单独的log file能使用的最大大小。将此数量的数据发送到日志文件后，将创建一个新的日志文件。若指定本参数值时不带单位，那默认的单位是KB。默认值是10MB。本参数仅能在postgresql.conf中设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

log_truncate_on_rotation

参数名称	log_truncate_on_rotation
数据类型	bool

默认值	off
取值范围	on和off
参数单位	
参数含义	在logging_collector配置参数设置为on的情况下，本参数将导致PostgreSQL截断（覆盖）而不是附加到任何同名的现有日志文件。请注意，只有在由于基于time(时间)的rotate而打开新文件时才会发生截断，而不是在服务器启动或基于size的rotate期间。当本参数值设置off时，在所有情况下都会附加到已有文件中。将本参数值与日志文件名（如postgresql-%H.log）结合使用会生成24个每小时一次的日志文件，然后周期性地覆盖它们。本参数仅能在postgresql.conf中设置或者在server command line中设置。举例：为了达到如下目的：保留7天的log，每天的log文件的文件名是server_log.Mon,server_log.Tue等等，然后用本周的log自动覆盖上周的log。该需求需要设置log_filename为server_log.%a, log_truncate_on_rotation为on，log_rotation_age为1440举例：为了达到如下目的：保留24小时的log，每小时一个log file，如果log file大小超过1GB时也需要很快进行rotate。该需求需要设置log_filename为server_log.%H%M, log_truncate_on_rotation为on，log_rotation_age为60，log_rotation_size为1000000.在log_filename中包括%M允许任何基于size的且可能在同一个小时内有多个log file的rotate
是否可session级修改	否
修改后何时生效	Reload即可生效

syslog_facility

参数名称	syslog_facility
数据类型	enum
默认值	local0
取值范围	{local0,local1,local2,local3,local4,local5,local6,local7}
参数单位	
参数含义	当配置参数log_destination被设置为syslog时，本参数用于确定使用的syslog facility。本参数仅能在postgresql.conf中设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

syslog_ident

参数名称	syslog_ident
数据类型	string
默认值	postgres
取值范围	
参数单位	
参数含义	本参数设置了在操作系统syslog日志中用于标识PostgreSQL消息的关键字。本参数仅仅适用于Linux平台，因为只有Linux平台才有syslog.虽然在Windows平台下也能修改本参数值，但是该修改没有任何意义。本参数仅能在postgresql.conf中设置或者在server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

syslog_sequence_numbers

参数名称	syslog_sequence_numbers
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当配置参数log_destination被设置为syslog并且本参数值设置为on时，PostgreSQL会在每条消息的前面加一个依次增大的顺序号，同一个命令的多条输出会以[2-1],[2-2]之类的次序标出：（如下截取自linux的/var/log/messages）image::PostgreSQL13%E6%96%87%E6%A1%A3%E4%B9%8BPG%E5%8F%82%E6%95%B0%E5%8F%82%E8%80%83%E6%89%8B%E5%86%8C_html_b376f05a85e6462b.png["",357,140] 本参数设置为off的效果如下：（如下截取自linux的/var/log/messages）image::PostgreSQL13%E6%96%87%E6%A1%A3%E4%B9%8BPG%E5%8F%82%E6%95%B0%E5%8F%82%E8%80%83%E6%89%8B%E5%86%8C_html_340e68e94edcda77.png["",361,131]本参数仅能在postgresql.conf中设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

syslog_split_messages

参数名称	syslog_split_messages
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当配置参数log_destination被设置为syslog并且本参数值设置为on时，消息被按行分割，长的行会被分割，以便填满1024Bytes，这是传统syslog实现上的典型大小限制。当设置为off时，PostgreSQL日志消息将按照原样传递给syslog服务，并由syslog服务来处理可能非常庞大的消息。如果syslog最终记录到一个文本文件中，那么无论哪种方式，效果都是一样的，最好保持该设置处于on状态，因为大多数syslog实现要么不能处理大型消息，要么需要专门配置来处理它们。但是，如果syslog最终要写入其他介质，那么将消息逻辑地保持在一起可能是必要的或更有用的。本参数仅能在postgresql.conf中设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

event_source

参数名称	event_source
数据类型	bool
默认值	PostgreSQL
取值范围	
参数单位	

参数含义	本参数仅仅适用于Windows平台，因为只有Windows平台才有event log，本参数值会体现在Windows操作系统的事件查看器的"Windows日志"-"应用程序"-"来源"一列。虽然在Linux平台下也能修改本参数值，但是该修改没有任何意义。本参数仅能在postgresql.conf中设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	重启PG instance生效

Preset Options

block_size

参数名称	block_size
数据类型	Integer
默认值	8192
取值范围	8192
参数单位	
参数含义	数据库使用的disk block的大小，该参数也影响其他的配置参数，比如shared_buffers，该参数通过编译时./configure命令的—with-blocksize选项确定
是否可session级修改	否，不可修改。
修改后何时生效	不可修改。

data_checksums

参数名称	data_checksums
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	显示data checksums是否启用。本参数值通过initdb时的-k选项或者—data-checksums选项确定。本参数值可以使用pg_checksums进行修改，pg_checksums命令具体用法请见pg_checksums --help
是否可session级修改	否
修改后何时生效	立即生效

data_directory_mode

参数名称	data_directory_mode
数据类型	integer
默认值	0700
取值范围	0到511
参数单位	
参数含义	在linux/unix中显示data_directory参数值对应的目录的权限；在windows系统中，本参数值总是显示为0700
是否可session级修改	否
修改后何时生效	不可修改

debug_assertions

参数名称	debug_assertions
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	显示PG在编译时是否启用assertion。本参数值通过编译时./configure命令的—enable-cassert选项确定。
是否可session级修改	否
修改后何时生效	不可修改

integer_datetimes

参数名称	integer_datetimes
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	支持64bit整型的date和time。本参数通过编译时./configure命令确定。注意：configure的—disable-integer-datetimes选项，是"obsolete option, no longer supported"
是否可session级修改	否
修改后何时生效	不可修改

lc_collate

参数名称	lc_collate
数据类型	string
默认值	en_US.UTF-8(根据操作系统环境变量不同而不同，见os命令locale的返回结果)
取值范围	
参数单位	
参数含义	字符串的排序规则(String sort order)。本参数值通过initdb时的—lc-collate确定，若是不指定该选项，initdb会取环境变量
是否可session级修改	否
修改后何时生效	在PG Database级别确定，确定后不可修改，只读参数

lc_ctype

参数名称	lc_ctype
数据类型	string
默认值	en_US.UTF-8(根据操作系统环境变量不同而不同，见os命令locale的返回结果)
取值范围	
参数单位	
参数含义	字符分类(是否区分大小写，什么是字符等)。本参数通过initdb时的—lc-ctype确定，若是不指定该选项，initdb会取环境变量
是否可session级修改	否

修改后何时生效	在PG Database级别确定，确定后不可修改，只读参数。

max_function_args

参数名称	max_function_args
数据类型	integer
默认值	100
取值范围	
参数单位	
参数含义	返回函数参数的最大个数，
是否可session级修改	否
修改后何时生效	不可修改，只读参数

max_identifier_length

参数名称	max_identifier_length
数据类型	integer
默认值	63
取值范围	
参数单位	
参数含义	返回标识符的最大长度
是否可session级修改	否
修改后何时生效	不可修改，只读参数

max_index_keys

参数名称	max_index_keys
数据类型	integer
默认值	32
取值范围	
参数单位	
参数含义	一个index最多能含有多少列
是否可session级修改	否
修改后何时生效	不可修改，只读参数

segment_size

参数名称	segment_size
数据类型	integer
默认值	131072
取值范围	
参数单位	8KB
参数含义	本参数值确定了每个disk file中的page的数量。本参数值通过编译时./configure命令执行的—with-segsize=SEGSIZE选项确定。
是否可session级修改	否

修改后何时生效	不可修改，只读参数

server_encoding

参数名称	server_encoding
数据类型	string
默认值	UTF8
取值范围	
参数单位	
参数含义	数据库的编码 (即:字符集),本参数通过initdb时的-E或者—encoding参数确定。
是否可session级修改	否
修改后何时生效	不可修改，只读参数

server_version

参数名称	server_version
数据类型	string
默认值	默认值是当前PostgreSQL版本，格式为主版本号.次版本号
取值范围	
参数单位	
参数含义	返回PostgreSQL的版本号
是否可session级修改	否
修改后何时生效	不可修改，只读参数

server_version_num

参数名称	server_version_num
数据类型	integer
默认值	130000，返回主版本号
取值范围	
参数单位	
参数含义	返回PostgreSQL的版本号
是否可session级修改	否
修改后何时生效	不可修改，只读参数

ssl_library

参数名称	ssl_library
数据类型	string
默认值	OpenSSL
取值范围	
参数单位	
参数含义	返回PostgreSQL软件编译时的SSL library。本参数值通过./configure的—with-openssl选项确定。
是否可session级修改	否

修改后何时生效	不可修改，只读参数

wal_block_size

参数名称	wal_block_size
数据类型	integer
默认值	8192
取值范围	
参数单位	
参数含义	返回WAL中disk block的大小。本参数值通过./configure的—with-wal-blocksize选项确定。
是否可session级修改	否
修改后何时生效	不可修改，只读参数

wal_segment_size

参数名称	wal_segment_size
数据类型	string
默认值	16777216
取值范围	
参数单位	Byte
参数含义	返回WAL Segments的大小。本参数值通过initdb的—wal-segsize选项确定。
是否可session级修改	否
修改后何时生效	不可修改，只读参数

Client Connection Defaults / Locale and Formatting

DateStyle

参数名称	DateStyle
数据类型	string
默认值	ISO, MDY
取值范围	
参数单位	
参数含义	本参数用于设置date和time的显示格式，以及解释不明确日期输入值的规则。由于历史原因，本参数值包括两个部分：第一部分：output format specification (ISO, Postgres, SQL, or German)；第二部分：the input/output specification for year/month/day ordering (DMY, MDY, or YMD)。上述两者可以单独设置，也可以一起设置。请注意，关键词Euro和European是DMY的同义词；关键词US和NonEnro是MDY的同义词；Initdb生成配置参数文件时，会根据os的locale命令的lc_time结果来设置本参数值。
是否可session级修改	是
修改后何时生效	Reload即可生效

IntervalStyle

参数名称	IntervalStyle
数据类型	enum

默认值	postgres
取值范围	postgres,postgres_verbose,sql_standard,iso_8601
参数单位	
参数含义	本参数用于设置interval values的显示格式。参数值sql_standard表示产生匹配SQL标准interval常量的输出；参数值postgres表示当DateStyle参数值设置为ISO时，产生匹配PostgreSQL8.4版本之前的输出；参数值postgres_verbose表示当DateStyle参数值设置为非ISO时，表示当DateStyle参数值设置为ISO时，参数值iso_8601表示产生匹配在ISO 8601标准的4.4.3.2章节中定义的时间间隔带"format with designators"匹配的输出。本参数值也会影响模糊的interval输入的解释。
是否可session级修改	是
修改后何时生效	Reload即可生效

TimeZone

参数名称	TimeZone
数据类型	String
默认值	内置的默认值是GMT，一般情况下，本参数值依据initdb时的操作系统时区而定，即：initdb时的操作系统时区是什么，initdb后postgresql.conf中的本参数值就是什么。注意：CentoOS7/8的操作系统当前时区请使用操作系统命令timedatectl status查询。
取值范围	
参数单位	
参数含义	为显示以及翻译timestamp设置时区。可用的时区名称可以通过pg_timezone_names视图进行查询。
是否可session级修改	是
修改后何时生效	Reload即可生效

timezone_abbreviations

参数名称	timezone_abbreviations
数据类型	string
默认值	Default
取值范围	
参数单位	
参数含义	被PostgreSQL接受的datetime输入的时区简写。默认值Default在世界上大部分地区都有效。另外的有效值是Australia和India。
是否可session级修改	是
修改后何时生效	Reload即可生效

extra_float_digits

参数名称	extra_float_digits
数据类型	string
默认值	1
取值范围	-15到3
参数单位	

参数含义	设置浮点值(float4,float8以及地理数据类型)的显示的位数。extra_float_digits大于0时，原有精度最多增加3位， 小于0时， 精度最多减掉相应数值， 等于0时， float4精度为6位， float8精度为15
是否可session级修改	否
修改后何时生效	Reload即可生效

client_encoding

参数名称	client_encoding
数据类型	string
默认值	取自Database encoding
取值范围	
参数单位	
参数含义	本参数用于设置客户端字符集。
是否可session级修改	是
修改后何时生效	Reload即可生效。

lc_messages

参数名称	lc_messages
数据类型	string
默认值	与系统locale设置有关
取值范围	
参数单位	
参数含义	本参数用于设置消息显示的语言。可接受的参数值是跟系统相关的。本参数值会影响发送到服务器运行日志以及客户端的消息，不正确的本参数值会影响服务器日志的可读性。
是否可session级修改	是
修改后何时生效	Reload即可生效，且只有超级用户才能更改此设置。

lc_monetary

参数名称	lc_monetary
数据类型	string
默认值	与系统locale设置有关
取值范围	
参数单位	
参数含义	设置用于格式化货币金额的语言环境，可接受的参数值是跟系统相关的。
是否可session级修改	是
修改后何时生效	Reload即可生效

lc_numeric

参数名称	lc_numeric
数据类型	string
默认值	与系统locale设置有关
取值范围	

参数单位	
参数含义	本参数值用于设置格式化数字。可接受的参数值是跟系统相关的。
是否可session级修改	是
修改后何时生效	Reload即可生效

lc_time

参数名称	lc_time
数据类型	string
默认值	
取值范围	
参数单位	
参数含义	本参数值用于设置格式化的日期和时间。可接受的参数值是跟系统相关的。
是否可session级修改	是
修改后何时生效	Reload即可生效

default_text_search_config

参数名称	default_text_search_config
数据类型	string
默认值	依据initdb时的lc_ctype对应的设置而定
取值范围	
参数单位	
参数含义	本参数值用于设置当全文检索函数中不明确指定配置项时所用的全文检索配置项。
是否可session级修改	是
修改后何时生效	Reload即可生效

Write-Ahead Log / Settings

wal_level

参数名称	wal_level
数据类型	enum
默认值	replica
取值范围	{minimal,replica,logical}
参数单位	

参数含义	本参数定义了有多少信息被写入到WAL日志中。参数值replica表示WAL日志中写入了足够的信息以支持WAL Archiving和replicatin（包括在standby server上的read-only查询），参数值minimal表示WAL日志中移除了除了需要从crash或者immeidate shutdown进行recovery之外的信息。参数值logical添加了必要信息以支持逻辑解码(logical decoding)，在本参数使用logical参数值的情况下，会使得WAL日志的数量变大，特别是很多表被配置为replica identity full并且很多delete和update被执行的情况下。本参数仅仅可以在PostgreSQL Start时修改。每个更高级别的参数值会包括更低级别参数值中记录的所有内容。但是最少的WAL不能包含足够的信息来从基本备份和WAL日志中重建数据，因此必须使用replica参数值来提供更多的信息以启用WAL归档（archive_mode）和流复制.在9.6之前的版本中，此参数值还可以是archive和hot_standby。在PG13版本中，这些仍被接受，但已映射到replica参数值上。当本参数设置为minimal时，如下操作会更快，因为如下操作在创建或重写它们的事务的其余部分中，没有记录永久关系的信息： ALTER … SET TABLESPACE CLUSTER CREATE TABLE REFRESH MATERIALIZED VIEW（无CONCURRENTLY） REINDEX TRUNCATE
是否可session级修改	否
修改后何时生效	重启PG instance生效

fsync

参数名称	fsync
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当本参数值为on时，通过发出fsync系统调用或者各种等价的方法(参见wal_sync_method配置参数)，PostgreSQL会尝试确保更新被物理的写入disk。这就保证了PostgreSQL Cluster可以在操作系统或者硬件crash之后能recover(恢复)到一个一致的状态。当本参数值为off时，会带来性能优势，在电源故障或者系统crash时，这会导致不可恢复的数据损坏。仅仅当数据库的全部数据来源自外部数据的情况下，才建议将本参数值设置为off关闭fsync的场景包括：从备份文件中加载数据到PG中；使用PG进行数据的批处理，然后数据库会被废弃并重建；一个经常被重建的只读数据库克隆，并且不会用于failover.高质量的硬件不足以让本参数值设置为off基于可靠恢复的原因将本参数值从off改为on时，需要强制将kernel中修改过的buffer刷新到持久存储中，可以通过如下方法达到本目的： PostgreSQL Cluster关闭；当fsync参数值为on时运行initdb –sync-only,运行sync操作系统命令，unmount掉文件系统或者重启服务器操作系统。在很多场景中，针对非关键事务将synchronous_commit参数值设置为off可以提供比将fsync参数值设置为off更大的潜在优势，不会造成数据损坏等风险。本参数值只能在postgresql.conf文件或server command line中设置。如果关闭此参数，请考虑关闭full_page_writes。
是否可session级修改	否
修改后何时生效	Reload即可生效

synchronous_commit

参数名称	synchronous_commit
数据类型	enum
默认值	on

取值范围	{local,remote_write,remote_apply,on,off}
参数单位	
参数含义	本参数值指定了在PostgreSQL数据库返回一个成功的提示给客户端之前，必须处理多少WAL数据。如果synchronous_standby_names是空，那么本参数有意义的参数值为on和off；remote_apply、remote_write、和local提供了与on参数值一样的本地同步级别。所有非off模式的本地行为是等待本地WAL刷新到Disk中。在off模式下，没有等待，因此存在一个延迟，这个延迟是给client报告成功与以后保证事务安全以防server crash之间的延迟（最大的延迟是三倍的wal_writer_delay参数值）。与fsync参数值不同，将此参数设置为off不会造成数据库不一致的任何风险。操作系统或者数据库的crash可能会导致一些最近据称已提交的事务丢失，但数据库状态将与这些事务完全中止时的状态相同，因此，当性能比事务持久性的精确确定性更重要时，将本参数设置为off是一个有用的选择。如果synchronous_standby_names是非空，本参数值用来控制事务提交是否等待在standby server上的WAL Record被处理。 当本参数值为remote_apply时,commit会等待直到从当前的同步standby指示它们已经收到事务的commit record并应用(applied),因此，它可以在standby 端被query可见，并且在standby端被写入持久性的存储中。这会导致比之前设置的更大的提交延迟，因为primary端会等standby端的WAL replay。 当本参数值为on时，commit会等待直到当前同步standby指示已经收到事务的commit record并且已经flush到持久性存储中。这确保了事务不会丢失,除非primary端和所有同步standby端的存储都发生损坏。 当本参数值为remote_write时,commit会等待直到当前同步standby指示已经收到事务的commit record并且已经写入到它们的文件系统中。本参数确保了在一个PostgreSQL 实例crash之后的数据持久，但是不针对standby端的操作系统级别的crash，因为，数据在standby端可能没有flush到持久性存储中。当参数值为local时，会导致commits一直在等待直到local flush到disk，而不是复制。当同步复制使用时，通常不需要这么做，但是为了完整而提供了这种方法。 此参数可随时更改；任何一个事务的行为都由提交时生效的设置决定。因此，让一些事务同步提交，而其他事务异步提交是可能的，也是有用的。例如，要在默认值相反时使单个多语句事务异步提交，请在事务中发出SET LOCAL synchronous_commit TO OFF
是否可session级修改	是
修改后何时生效	Reload即可生效

wal_sync_method

参数名称	wal_sync_method
数据类型	enum
默认值	Linux和FreeBSD下默认值是fdatasyncWindows下默认值是open_datasync
取值范围	{fsync,fdatasync,open_sync,open_datasync}
参数单位	
参数含义	本参数值指定了强制将WAL写入Disk时使用的方法。当fsync参数值为off时，本参数值无关紧要。 open_datasync：使用open()的O_DSYNC选项写WAL Files fdatasync:每次commit时调用fdatasync() fsync:每次commit时调用fsync() fsync_writethrough:每次commit时调用fsync(),对disk write cache强制write-through open_sync：使用open()的O_SYNC选项写WAL Files open_*选项使用O_DIRECT(若是可用)，不是上述所有选项都在所有平台下可用。默认值是在该平台下支持的上述列表中的第一个方法，除了在Linux平台和FreeBSD平台下fdatasync是默认值。默认值不一定理想，可能有必要更次此设置或系统配置的其他方面，以创建crash的安全配置或实现最佳性能。本参数值只能在postgresql.conf文件或server command line中设置。

是否可session级修改	否
修改后何时生效	Reload即可生效

full_page_writes

参数名称	full_page_writes
数据类型	bool
默认值	on
取值范围	
参数单位	
参数含义	当本参数值为on时，在一个checkpoint发生之后，当第一个修改发生时，将full page（即：该修改所在的disk page中的全部内容）写入WAL日志中。这是需要的，因为在操作系统crash期间正在进行的page write可能仅部分完成，从而导致磁盘上的页包含新旧数据的混合。通常存储在WAL中的行级更改数据不足以在crash之后的恢复中完全还原一个page。存储full page image可以保证页面可以正确还原，但代价是增加必须写入WAL的数据量(因为WAL replay总是从检查点开始，所以在检查点之后的每个页面的第一次更改期间这样做就足够了。因此，降低整页写入成本的一种方法是增加checkpoint interval参数。）关闭此参数可加快正常操作，但在系统发生故障后，可能会导致不可恢复的数据损坏或静默数据损坏。风险类似于关闭fsync，虽然较小，但应该仅基于为fsync参数建议的相同情况关闭本参数。关闭此参数不会影响将WAL存档用于时间点恢复（PITR）。此参数只能在postgresql.conf文件或server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_log_hints

参数名称	wal_log_hints
数据类型	bool
默认值	off
取值范围	
参数单位	
参数含义	当本参数值设置为on时，在一个checkpoint发生之后，当第一个修改发生时，将full page（即：该修改所在的disk page中的全部内容）写入WAL日志中。即使对于所谓的提示位的非关键修改也是如此。如果data ckecksums被启用，hint bit updates总是被记录在wal中，此时本参数值被忽略。你可以使用本参数值来测试当数据库cluster启用checksum之后的会生成多少额外的wal-logging.本参数仅能在server start时设置。
是否可session级修改	否
修改后何时生效	PG instance重启生效

wal_compression

参数名称	wal_compression
数据类型	bool
默认值	off

取值范围	on和off
参数单位	
参数含义	当本参数值为on时，在如下情况下PostgreSQL压缩一个全部的page image并写入到WAL中：full_page_writes参数值为on或者执行一次base backup。压缩的page image会在wal replay期间进行解压。启用本参数可以降低WAL日志的产生量而不会增加不可恢复的数据损坏的风险，但是在wal压缩以及解压过程中会有额外的cpu消耗。
是否可session级修改	是，仅限于superuser
修改后何时生效	Session级修改立即生效，非session修改relaod生效

wal_init_zero

参数名称	wal_init_zero
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当本参数值为on时，会导致新的wal file以零填充，在某些文件系统中，这确保了在需要写入wal records之前的空间已被分配。但是，Copy-On-Write (COW)文件系统不会从本参数中获益，因此可以选择跳过不必要的工作。如果设置为off，则在创建文件时仅写入最后一个字节，以使其具有预期的大小。
是否可session级修改	是，仅限于superuser
修改后何时生效	Session级修改立即生效，非session修改relaod生效

wal_recycle

参数名称	wal_recycle
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	本参数值为on时，会导致wal file被循环使用(通过renaming wal file的方式)，避免了创建新wal file的需要。在COW文件系统中，本参数可能让create new wal file更快。
是否可session级修改	是，仅限于superuser
修改后何时生效	Session级修改立即生效，非session修改relaod生效

wal_buffers

参数名称	wal_buffers
数据类型	integer
默认值	-1，即:shared_buffers配置参数值的1/32大小，不小于64KB也不大于一个wal segment大小。请注意这是默认值的取值范围，不是手工设置本参数值的取值范围。
取值范围	-1到262143
参数单位	8KB

参数含义	分配给尚未写入到disk中的wal data的共享内存大小；如果自动选择太大或太小，则可以手动设置本参数值，但是任何小于32 KB的正数参数值.将被视为32 KB.如果指定的该值不带单位，则单位是WAL块的个数，即源码中符号常量XLOG_BLCKSZ，通常为8KB. 本参数值只能在服务器启动时设置。当每次事务提交时，wal buffer中的内容都会被写入disk中，因此，极大的本参数值不可能带来显著的益处。
是否可session级修改	否
修改后何时生效	PG instance重启生效

wal_writer_delay

参数名称	wal_writer_delay
数据类型	integer
默认值	200
取值范围	1到10000
参数单位	毫秒
参数含义	指定了wal writer进程写wal buffer到disk中的频率，以时间去衡量，当wal writer进程flush wal之后，wal writer进程会休眠本参数值一段时间，除非被一个异步提交事务很快唤醒。如果最后一次flush在小于wal_writer_delay参数值之前发生 以及小于wal_writer_flush_after参数值对应的WAL生成量，那么WAL被写入操作系统而不是flush到disk中。请注意，在很多操作系统中，sleep delays的有效精度是10毫秒，
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_writer_flush_after

参数名称	wal_writer_flush_after
数据类型	Integer
默认值	128，即128*8KB=1MB
取值范围	0到2147483647
参数单位	8KB
参数含义	本参数值指定了wal writer进程flush wal的频率，以wal的大小来衡量。如果最后一次flush在小于wal_writer_delay参数值之前发生 以及小于wal_writer_flush_after参数值对应的WAL生成量，那么WAL被写入操作系统而不是flush到disk中。本参数值为0代表wal data总是被立即被flush到disk中。如果指定的该值不带单位，则单位是WAL块的个数，即源码中符号常量XLOG_BLCKSZ，通常为8KB. 本参数值只能在postgresql.conf或者server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

wal_skip_threshold

参数名称	wal_skip_threshold
数据类型	Integer
默认值	2048

取值范围	0到2147483647
参数单位	KB
参数含义	当wal_level参数值为minimal并且在create或者rewriting一个永久关系之后进行事务提交，本参数控制怎么持久化新的数据。如果数据小于本参数值，PostgreSQL会将这些数据写入wal log，否则，PostgreSQL会针对受影响的文件使用一个fsync。取决于你的存储属性，如果这些commit正在拖慢并发事务，增大或者降低本参数值可能会有帮助. 本参数值不带单位时，默认以KB为单位。
是否可session级修改	是
修改后何时生效	Reload即可生效。

commit_delay

参数名称	commit_delay
数据类型	Integer
默认值	0，零表示没有延迟。
取值范围	0到100000
参数单位	毫秒
参数含义	设置本参数值会增加在wal flush启动之前的延迟。如果系统负载足够高以至于在给定的时间间隔内准备好提交其他事务，则这可以通过允许经由单个 WAL flush以提交大量事务来提高组提交吞吐量。但是，每次wal flush会增加延迟，最大延迟是本参数指指定的时间。如果没有其他事务会变成准备提交状态，延迟的时间是被浪费的，因此，如果当wal flush被初始化时，至少有commit_siblings个其他事务是active的情况下，延迟才被执行。同样，如果fsync被关闭，delay是不会被执行的。
是否可session级修改	是，仅限于superuser
修改后何时生效	Session级修改立即生效，其他修改reload生效

commit_siblings

参数名称	commit_siblings
数据类型	Integer
默认值	5
取值范围	0到1000
参数单位	
参数含义	在执行配置参数commit_delay指定的延迟之前，需要的最小的并发打开的事务数。值越大，在延迟间隔期间至少有一个其他事务准备好提交的可能性就越大
是否可session级修改	是
修改后何时生效	Reload即可生效

Developer Options

allow_system_table_mods

参数名称	allow_system_table_mods
数据类型	bool
默认值	off
取值范围	on和off

参数单位	
参数含义	允许修改系统表的结构以及对系统表进行某些其他有风险的操作。否则即使对于超级用户也是不允许的。不明智地使用此设置可能会导致无法挽回的数据丢失或严重损坏数据库系统。只有超级用户才能更改此设置。
是否可session级修改	是，仅限于superuser
修改后何时生效	Session级修改立即生效，其他修改reload生效

backtrace_functions

参数名称	backtrace_functions
数据类型	string
默认值	空字符串
取值范围	
参数单位	
参数含义	本参数值是以逗号分隔的C函数名称。如果一个错误被抛出并且错误发生地方的内部C函数名称与本参数值list中的值相匹配，然后backtrace与error message会被写入server log。不是所有的平台都支持backtrace功能，backtrace的质量取决于编译选项。
是否可session级修改	是，仅限于superuser
修改后何时生效	Session级修改立即生效，其他修改reload生效

ignore_system_indexes

参数名称	ignore_system_indexes
数据类型	boolean
默认值	off
取值范围	off和on
参数单位	
参数含义	当读取system table时忽略system index（但是当修改表时，依然会更新索引）。本参数用于从损坏的索引中恢复的场景中。本参数当session启动后无法改变
是否可session级修改	否
修改后何时生效	Reload即可生效，但是仅仅针对reload之后的新session生效。

post_auth_delay

参数名称	post_auth_delay
数据类型	Integer
默认值	0，零表示禁用延迟。
取值范围	0到2147
参数单位	秒
参数含义	在PostgreSQL构建认证阶段之后，当一个新server process被fork出来时的延迟的时间。本参数值用来给开发人员一个用debugger attach到server process的机会。
是否可session级修改	否
修改后何时生效	Reload即可生效，但是仅仅针对reload之后的新session生效。

pre_auth_delay

参数名称	pre_auth_delay
数据类型	Integer
默认值	0，零表示禁用延迟。
取值范围	0到60
参数单位	秒
参数含义	在PostgreSQL构建认证阶段之前，在一个新server process被fork出来之后的延迟的时间。本参数值用来给开发人员一个使用debugger attach到server process以跟踪在认证阶段异常行为的机会。本参数只能在postgresql.conf中或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

trace_notify

参数名称	trace_notify
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	为LISTEN和NOTIFY命令生成大量的debugging输出。配置参数client_min_messages或者log_min_messages必须设置为DEBUG1或者更低才能发送该输出到client或者server log
是否可session级修改	是
修改后何时生效	Reload即可生效

trace_recovery_messages

参数名称	trace_recovery_messages
数据类型	enum
默认值	log，log不会影响log决策
取值范围	{debug5,debug4,debug3,debug2,debug1,log,notice,warning,error}
参数单位	
参数含义	启用记录恢复相关的调试输出，否则将不会被记录。此参数允许用户覆盖log_min_messages的正常设置，但仅限于特定消息。本参数的目的是调试hot standby。其他值导致该优先级或更高优先级的与恢复相关的调试消息被记录，就好像它们具有LOG优先级一样。针对配置参数log_min_messages的通用设置会导致无条件发送messages到server log中。本参数只能在postgresql.conf中或者server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

trace_sort

参数名称	trace_sort
数据类型	bool
默认值	off

取值范围	off和on
参数单位	
参数含义	当本参数值为on时，在sort操作期间，会发出有关资源使用的信息。本参数仅仅针对PostgreSQL编译时TRACE_SORT macro被定义时才有效（默认情况下，TRACE_SORT是被定义的）
是否可session级修改	是
修改后何时生效	Reload即可生效

trace_locks

参数名称	trace_locks
数据类型	bool
默认值	
取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

trace_lwlocks

参数名称	trace_lwlocks
数据类型	
默认值	
取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

trace_userlocks

参数名称	trace_userlocks
数据类型	
默认值	
取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

trace_lock_oidmin

参数名称	trace_lock_oidmin
数据类型	
默认值	

取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

trace_lock_table

参数名称	trace_lock_table
数据类型	
默认值	
取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

debug_deadlocks

参数名称	debug_deadlocks
数据类型	
默认值	
取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

log_btree_build_stats

参数名称	log_btree_build_stats
数据类型	
默认值	
取值范围	
参数单位	
参数含义	
是否可session级修改	
修改后何时生效	

wal_consistency_checking

参数名称	wal_consistency_checking
数据类型	String
默认值	空串，空串意味着禁用本特性。
取值范围	

参数单位	
参数含义	本参数的目的是检查WAL redo routine中的错误。当启用本参数时，与WAL记录一起修改的任何buffer的full page image都会添加到记录中。如果该记录随后被重放，系统将首先应用每个记录，然后测试由该记录修改的buffer是否与存储的image匹配。在某些情况下（如提示位），可以接受较小的变化，并将被忽略。任何意外的差异都将导致致命错误，从而终止恢复。本参数值可以设置为all，all表示检查所有记录，本参数值也可以设置为逗号分隔的resource manager列表，受到支持的resource manager是heap、heap2、btree、hash、gin、gist、sequence、spgist、brin、generic
是否可session级修改	是，仅限于superuser进行修改
修改后何时生效	Reload即可生效

wal_debug

参数名称	wal_debug
数据类型	bool
默认值	
取值范围	
参数单位	
参数含义	当设置为on时，发出与wal相关的debugging out。本参数仅仅当PostgreSQL编译时定义了WAL_DEBUG macro的情况下才适用。
是否可session级修改	
修改后何时生效	

ignore_checksum_failure

参数名称	ignore_checksum_failure
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	本参数仅仅在data checksums启用的情况下才生效。在读取过程中检查到checksum失败通常会导致PostgreSQL报告一个error，从而中止当前事务。本参数值设置为on会导致PostgreSQL忽略故障（但是依然会报告警告）并继续处理。此行为可能导致crash、传播或隐藏corruption或其他严重问题。但是，如果block header是正常的，可能允许您越过错误并检索可能仍然存在于表中的未损坏的tuple。如果block header已损坏，那么即使启用本选项，也会报告错误。
是否可session级修改	是，仅限于superuser在session级修改
修改后何时生效	Reload即可生效

zero_damaged_pages

参数名称	zero_damaged_pages
数据类型	bool
默认值	off
取值范围	on和off

参数单位	
参数含义	检测到一个损坏的page header通常会导致PostgreSQL报错一个error，并终止当前事务。设置本参数值为on会导致系统报告warning，将内存中损坏的页面清零，然后继续处理。这种行为会破坏数据，即：损坏页面上的所有行。但是，本参数确实允许您克服错误并从表中可能存在的任何未损坏的页面中检索行。如果由于硬件或软件错误而发生损坏，它对于恢复数据很有用。在您放弃从表的损坏页恢复数据的希望之前，您通常不应设置本参数。。清零页不会被强制写入磁盘，因此建议在再次关闭此参数之前重新创建表或索引。本参数值只能由超级用户更改。
是否可session级修改	是，仅限于superuser
修改后何时生效	Reload即可生效

ignore_invalid_pages

参数名称	ignore_invalid_pages
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	当本参数值是off时，在recovery期间检测wal record中存在invalid pages之后，PostgreSQL会发出一个PANIC-level的error，进而会终止recovery。当本参数值设置为on时会导致PostgreSQL忽略wal record中的invalid pages（但是依然会报warning）继续进行recovery。本行为可能会导致crash，data loss，传播或者隐藏corruption，或者其他严重问题。但是，本参数会允许你绕过的PANIC-level的error，进而完成recovery，进而启动PostgreSQL。本参数仅能在PostgreSQL启动时设置。本参数仅仅在recovery或者standby mode下生效。
是否可session级修改	否
修改后何时生效	重启PG instance生效

jit_debugging_support

参数名称	jit_debugging_support
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	注册JIT编译函数给debugger使用，本参数仅能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	Reload即可对新的connection生效

jit_dump_bitcode

参数名称	jit_dump_bitcode
数据类型	bool
默认值	off
取值范围	on和off
参数单位	

参数含义	写出LLVM的bitcode以便利于JIT debugging，
是否可session级修改	是，仅限于superuser
修改后何时生效	Reload即可生效

jit_expressions

参数名称	jit_expressions
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当 JIT 编译被启用时，本参数用于确定表达式是否被 JIT 编译
是否可session级修改	是
修改后何时生效	Reload即可生效

jit_profiling_support

参数名称	jit_profiling_support
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	如果 LLVM 具有所需的功能，则发出允许perf分析 JIT 生成的函数所需的数据。这会将文件写入$HOME/.debug/jit/下; 用户负责在需要时执行清理。默认设置是off。该参数只能在服务器启动时设置 。本参数仅能在PostgreSQL启动时设置。
是否可session级修改	否
修改后何时生效	Reload即可对新的connection生效

jit_tuple_deforming

参数名称	jit_tuple_deforming
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当JIT编译被启用的状态下，本参数用于确定允许不允许tuple deforming进行jit编译
是否可session级修改	是
修改后何时生效	Reload接口生效

Query Tuning / Planner Method Configuration

enable_bitmapscan

参数名称	enable_bitmapscan
数据类型	bool

默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对位图扫描计划类型的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_gathermerge

参数名称	enable_gathermerge
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对gather merge计划类型的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_hashagg

参数名称	enable_hashagg
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对hashed aggregation计划类型的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_hashjoin

参数名称	enable_hashjoin
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对hash-join计划类型的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_incremental_sort

参数名称	enable_incremental_sort
数据类型	bool
默认值	on

取值范围	on和off
数单位	
参数含义	启用或禁用query planner对incremental sort steps的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_indexscan

参数名称	enable_indexscan
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对index-scan的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_indexonlyscan

参数名称	enable_indexonlyscan
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对index-only-scan的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_material

参数名称	enable_material
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对materialization的使用，完全抑制物化是不可能的，但是关闭本参数可以防止query planner插入物化节点，除非在需要正确性的情况下。
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_mergejoin

参数名称	enable_mergejoin
数据类型	bool
默认值	on

取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对merge-join的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_nestloop

参数名称	enable_nestloop
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对nested-loop的使用。完全抑制nested-loop连接是不可能的，但是如果有其他方法可用，关闭本参数会阻止query planner使用nested-loop。
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_parallel_append

参数名称	enable_parallel_append
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对parallel-aware append的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_parallel_hash

参数名称	enable_parallel_hash
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对parall hash join的使用，在hash-join被禁用的情况下，本参数值无效。
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_partition_pruning

参数名称	enable_partition_pruning
数据类型	bool

默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner的partitioned table中消除partition。这也控制计划器生成查询计划的能力。允许查询执行器在查询执行期间删除（忽略）分区
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_partitionwise_join

参数名称	enable_partitionwise_join
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对partitionwise join的使用，这允许在partitioned table上的join在匹配的partition上执行join。Partitionwise 目前仅适用于连接条件包括所有分区键的情况，这些分区键必须是相同的数据类型并且具有一对一匹配的子分区集。由于分区连接规划在规划期间可以使用明显更多的 CPU 时间和内存，本参数默认值为off
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_partitionwise_aggregate

参数名称	enable_partitionwise_aggregate
数据类型	bool
默认值	off
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对partitionwise grouping or aggregation的使用，这允许在partitioned table上的partitionwise grouping or aggregation会在每个partition上单独执行。如果GROUP BY子句不包含分区键，则只能在每个分区的基础上执行部分聚合，并且必须稍后执行最终确定。由于partitionwise grouping or aggregation在规划期间可能会使用更多的 CPU 时间和内存，本参数值为off
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_seqscan

参数名称	enable_seqscan
数据类型	bool
默认值	on
取值范围	on和off
参数单位	

参数含义	启用或禁用query planner对sequential scan的使用。完全抑制顺序扫描是不可能的，但是如果有其他方法可用，关闭本参数会阻止query planner使用sequential scan。
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_sort

参数名称	enable_sort
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对explicit sort steps的使用。完全抑制显式排序是不可能的，但是如果有其他方法可用，关闭本参数会阻止query planner使用显式排序。
是否可session级修改	是
修改后何时生效	Reload即可生效

enable_tidscan

参数名称	enable_tidscan
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	启用或禁用query planner对TID scan的使用
是否可session级修改	是
修改后何时生效	Reload即可生效

Reporting and Logging / What to Log

application_name

参数名称	application_name
数据类型	string
默认值	psql
取值范围	
参数单位	
参数含义	本参数值的长度小于NAMEDATALEN值（NAMEDATALEN是源码中的符号常量，默认是64个字符）。本参数被连接到服务器上的应用程序设置，本参数值会显示在pg_stat_activity视图中，也会显示在csv log中。本参数值也可以被包括在配置参数log_line_prefix的参数值中。只有可打印的ASCII字符才可以用于本参数值。其他字符会被问号(?)取代。
是否可session级修改	是
修改后何时生效	Reload即可生效

debug_print_parse

参数名称	debug_print_parse
数据类型	bool
默认值	
取值范围	
参数单位	
参数含义	在运行日志中记录query的parse tree。本消息在LOG消息级别的情况下被发出，因此，默认情况下，本消息会出现在运行日志中，但不会发给client。您可以通过调整client_min_messages配置参数、log_min_messages配置参数让消息发送给client。
是否可session级修改	是
修改后何时生效	Reload即可生效

debug_print_rewritten

参数名称	debug_print_rewritten
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	在运行日志中记录query的query rewriter output,。本消息在LOG消息级别的情况下被发出，因此，默认情况下，本消息会出现在运行日志中，但不会发给client。您可以通过调整client_min_messages配置参数、log_min_messages配置参数让消息发送给client。
是否可session级修改	是
修改后何时生效	Reload即可生效

debug_print_plan

参数名称	debug_print_plan
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	在运行日志中记录query的execution plan。本消息在LOG消息级别的情况下被发出，因此，默认情况下，本消息会出现在运行日志中，但不会发给client。您可以通过调整client_min_messages配置参数、log_min_messages配置参数让消息发送给client。
是否可session级修改	是
修改后何时生效	Reload即可生效

debug_pretty_print

参数名称	debug_pretty_print
数据类型	bool
默认值	on
取值范围	on和off

参数单位	
参数含义	本参数设置为on时，会将debug_print_parse, debug_print_rewritten, or debug_print_plan配置参数产生的消息进行缩进，这会导致比设置为off时的compact格式更具有可读性但输出会更长。
是否可session级修改	是
修改后何时生效	Reload即可生效

log_checkpoints

参数名称	log_checkpoints
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	将checkpoints 和restartpoints记录到运行日志中，一些统计信息被包括到消息中，如写入buffer的数量以及写buffer的耗时等。本参数仅能在postgresql.conf设置或者在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

log_connections

参数名称	log_connections
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	当本参数值为on时，每一次尝试连接到PostgreSQL数据库以及客户端成功完成身份认证都会被记录到运行日志中。请注意：有些客户端程序(如psql)会连接两次以确定password是否需要，在该情况下，会有重复的"connection received"消息出现，该消息不一定表示有问题。
是否可session级修改	否
修改后何时生效	Reload即可对新的connection生效

log_disconnections

参数名称	log_disconnections
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	当本参数值为on时， session终止的信息会被记录到运行日志中。日志输出类似log_connections配置参数的输出信息，以及session的持续时间。
是否可session级修改	否
修改后何时生效	Reload即可对新的connection生效

log_error_verbosity

参数名称	log_error_verbosity
数据类型	enum
默认值	default
取值范围	{terse,default,verbose}
参数单位	
参数含义	设置日志消息的详细程度。控制在PG运行日志中为记录的每条消息写入的详细程度。Terse排除了DETAIL、HINT、QUERY、和CONTEXT 级别的信息。Verbose输出包括SQLSTATE错误代码以及源码文件名称、函数名称、生成错误的行号。仅有superuser能修改本参数值
是否可session级修改	是，仅限于superuser
修改后何时生效	Reload即可生效

log_hostname

参数名称	log_hostname
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	默认情况下，在运行日志中显示连接主机的 IP 地址。打开此参数也会导致记录主机名。请注意，根据您的主机名解析设置，这可能会造成不可忽视的性能损失。此参数只能在postgresql.conf文件中或在server command line中设置。
是否可session级修改	否
修改后何时生效	Reload即可生效

log_line_prefix

参数名称	log_line_prefix	
数据类型	string	
默认值	%m [%p]	
取值范围		
参数单位		
参数含义	控制运行日志中每行开头的输出内容。	
是否可session级修改	否	
修改后何时生效	Reload即可生效。	

log_lock_waits

参数名称	log_lock_waits
数据类型	bool
默认值	off
取值范围	off和on
参数单位	

参数含义	当一个session的等待时间长于配置参数deadlock_timeout参数值以获得锁时是否生成日志消息。这在锁定等待是否导致性能不佳时非常有用。只有superuser才能更改本设置。
是否可session级修改	是，仅限于superuser进行修改
修改后何时生效	Reload即可生效。

log_parameter_max_length

参数名称	log_parameter_max_length
数据类型	integer
默认值	-1
取值范围	-1到1073741823
参数单位	Byte
参数含义	如果本参数值大于零，则使用非错误语句日志消息记录的每个绑定参数值将被修剪为本参数值个字节。零禁用非错误语句日志的绑定参数的日志记录。-1（默认）允许完整记录绑定参数。如果此值未指定单位，则将其视为字节。只有超级用户才能更改此设置。 此设置仅影响由于log_statement、log_duration和相关设置而打印的日志消息。此设置的非零值会增加一些开销，特别是如果参数以二进制形式发送，则需要转换为文本。
是否可session级修改	是，仅限于superuser进行修改
修改后何时生效	Reload即可生效

log_parameter_max_length_on_error

参数名称	log_parameter_max_length_on_error
数据类型	integer
默认值	0
取值范围	-1到1073741823
参数单位	Byte
参数含义	如果大于零，错误消息中报告的每个绑定参数值都会被修剪到本参数值个字节。零（默认值）禁用在错误消息中包含绑定参数。-1允许完整打印绑定参数。如果此值未指定单位，则将其视为字节此设置的非零值会增加开销，因为PostgreSQL需要在每个语句开始时将参数值的文本表示存储在内存中，无论最终是否发生错误。以二进制形式发送绑定参数时的开销比以文本形式发送时要大，因为前者需要数据转换，而后者只需要复制字符串
是否可session级修改	是
修改后何时生效	Reload即可生效

log_statement

参数名称	log_statement
数据类型	enum
默认值	all
取值范围	{none,ddl,mod,all} none是off的意思， ddl的含义是记录所有数据定义语句，如create、alter、drop; mod的含义是记录DDL语句之外，还会记录数据修改语句，如insert、update、delete、truncate以及copy from语句。PREPARE、EXECUTE和EXPLAIN ANALYZE语句所包含的命令的类型适当时，也会记录这些语句。all的含义是记录所有语句。

参数单位	
参数含义	控制在PG的运行日志中写入哪些类型的SQL语句。对于使用扩展查询协议的客户端来说，记录到运行日志是从一个执行消息被收到开始的。绑定参数值也会被记录。请注意，即使是本参数值设置为all，运行日志中也不会显示那些包含简单语法错误的SQL语句。因为仅仅在基本的解析完成并确定SQL语句的type之后，log message才会被发出。在扩展查询协议的情况下，本参数值同样不会记录在执行阶段之前(即:在parser分析或planner期间)失败的语句。设置log_min_error_statement为error(或者更低)以记录此类语句
是否可session级修改	是，仅限于superuser进行修改
修改后何时生效	Reload即可生效

log_replication_commands

参数名称	log_replication_commands
数据类型	bool
默认值	off
取值范围	off和on
参数单位	
参数含义	是否将每个复制命令记录到PG运行日志中
是否可session级修改	是，仅限于superuser修改
修改后何时生效	Reload即可生效

log_temp_files

参数名称	log_temp_files
数据类型	integer
默认值	-1，代表禁用本log功能
取值范围	-1到2147483647
参数单位	KB
参数含义	控制是否记录本临时文件的文件名和大小。临时文件被创建用以排序、hash以及临时查询结果。当本参数值被启用时，当临时文件被删除时，一个日志条目会被发出。本参数值为0时会记录所有的临时文件信息。本参数值为正数时表示仅仅会记录那些临时文件大小大于等于本参数值的临时文件。本参数值不指定单位时，默认值以KB为单位。本参数仅仅superuser可以修改。
是否可session级修改	是，仅限于superuser进行修改。
修改后何时生效	Reload即可生效

log_timezone

参数名称	log_timezone
数据类型	String
默认值	内置默认值是GMT，但该内置默认值通常在postgresql.conf中被覆盖;initdb将在会设置本参数值以便与系统环境相对应。
取值范围	
参数单位	

参数含义	设置在log message中使用的time zone。与TimeZone配置参数不同，此值是PostgreSQL Cluster范围的，因此所有会话将一致地报告时间戳。此参数只能在postgresql.conf文件中或在server command line中设置
是否可session级修改	否
修改后何时生效	Reload即可生效

Client Connection Defaults / Statement Behavior

client_min_messages

参数名称	client_min_messages
数据类型	enum
默认值	notice
取值范围	{debug5,debug4,debug3,debug2,debug1,log,notice,warning,error}
参数单位	
参数含义	控制将哪些消息级别发送到客户端。有效值为DEBUG5，DEBUG4，DEBUG3，DEBUG2，DEBUG1，LOG，NOTICE，WARNING和ERROR。每个级别包括其后的所有级别。级别越靠后，发送的消息就越少。请注意，本参数值log的级别与log_min_messages配置参数值log的级别是不同的。
是否可session级修改	是
修改后何时生效	Reload即可生效

search_path

参数名称	search_path
数据类型	string
默认值	"$user", public
取值范围	
参数单位	

参数含义	当一个object(table, data type, function等等)不带前面的schema name时，本参数用来指定哪个schema name会被搜索到。当在不同的schema name中有相同的object name时，在search_path中第一个匹配的object的那个schema会被使用。若是一个object没有被search_path中的任何schema匹配到，那么，这个object使用时需要带schema name(即:前面带上 "schema.").本参数值需要以逗号分隔，当本参数值包括不存在schema name或者user没有对该schema name的usage权限时，此类参数值会被忽略。本参数值中出现的$user，其含义是current_user，若是存在与该user同名的schema name并且该user有对该schema name的usage权限（若不是这样，$user会被忽略）不论在本参数值中包括不包括pg_catalog，pg_catalog总会被搜索到，若是在本参数值中包括pg_catalog，那么会按照在本参数值中出现的顺序去搜，若是在本参数值中不包括pg_catalog,那么，PostgreSQL会在本参数值之前去搜索pg_catalog.类似的，当前的session临时表schema pg_temp_nnn 若是存在的话，会总被搜索到。可以在本参数值中使用pg_temp来明确指定要对pg_temp_nnn进行搜索。若是在本参数值中不明确列出，那么，PostgreSQL会首先搜索pg_temp_nnn（在pg_catalog之前）。临时schema name仅仅被用来搜索relation(表、视图、序列等等)和data type name，临时schema name不能用来搜索function name或者operator name当一个object被建立时其前面不指定任何schema name时，该object的schema name是在本参数值中搜索到的第一个有效的schema。当本参数值为空时，一个错误会被抛出。 本参数的默认值是"$user", public，该默认值支持对一个数据库的共享使用（即:没有user有私有的schemas，并且都对public共享使用）、每个user schema的私有使用以及混合使用。其他的影响可以通过alter语句修改本参数值来获得，这个修改可以是全局的也可以每个user的。可以通过函数current_schemas来检查当前使用的有效的schema name，这与search_path的参数值并不完全一致，因为函数current_schemas显示了在本参数值中出现的item是如何解析的。
是否可session级修改	是
修改后何时生效	Reload即可生效

row_security

参数名称	row_security
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	本参数控制是否抛出错误以代替应用行安全策略。on代表正常应用行安全策略，off代表查询失败除非应用了至少行安全策略。在限制了行可见性导致错误的结果的情况下，可以将本参数值改为off。在默认情况下，pg_dump会进行该修改(即:修改本参数值为off)。本参数对可以绕过行安全策略的roles是无效的，即:superuser和那些带有bypassrls属性的role。
是否可session级修改	是
修改后何时生效	Reload即可生效

default_table_access_method

参数名称	default_table_access_method
数据类型	string

默认值	heap
取值范围	
参数单位	
参数含义	本参数指定了create table或者create materialized views不带访问方法时table或者materialized views的访问方法，以及select …into语句使用时用到的访问方法。
是否可session级修改	是
修改后何时生效	Reload即可生效

default_tablespace

参数名称	default_tablespace
数据类型	string
默认值	空字符串，空字符串代表当前database的默认tablespace
取值范围	
参数单位	
参数含义	在create object(table和index)语句不明确指定tablespace情况下，本参数值指定了该object对应的tablespace。如果指定的本参数值与现有的任何tablespace不匹配，那么PostgreSQL会自动使用当前database的默认tablespace。如果一个非默认表空间被指定，那么user必须带有对该非默认表空间的create的权限，否则创建会失败本参数不适用于临时表。临时表对应的配置参数是temp_tablespaces。本参数也不适用于create database语句。默认情况下，一个新的database从template数据库中继承了tablespace。若是本参数值是非空字符串，然后partitioned table被创建，那么，该partitioned table的表空间被设置为该非空字符串，该非空字符串也是将来partition被建立的默认表空间，即使本配置参数值之后被更改。
是否可session级修改	是
修改后何时生效	Reload即可生效。

temp_tablespaces

参数名称	temp_tablespaces
数据类型	string
默认值	空字符串，空字符串代表所有的temporary objects都会被创建在当前数据库的默认表空间中。
取值范围	
参数单位	
参数含义	在create temporary object(temp tables 和temp tables上的indexes)语句不明确指定tablespace情况下，本参数值指定了该object对应的temporary tablespace。诸如排序生成的临时文件也被建立到这个temp_tablespaces指定的临时表空间中。本参数值是一系列的tablespace name的清单，当有多个tablespace name的情况下，每次创建临时对象时，PostgreSQL都会从列表中随机选择一个tablespace name。除了在事务中，连续创建的临时对象被放置在列表中的连续表空间中。如果列表中选定的是空字符串，PostgreSQL会自动使用当前数据库的默认表空间。当temp_tablespaces以交互方式设置时，指定不存在的表空间会报错，指定用户没有CREATE权限的表空间也会报错。但是，当使用先前设置的值时，将忽略不存在的表空间，以及用户缺乏CREATE权限的表空间。此规则适用于在postgresql.conf设置本参数值的情况下

是否可session级修改	是
修改后何时生效	Reload即可生效

check_function_bodies

参数名称	check_function_bodies
数据类型	bool
默认值	on
取值范围	on和off
参数单位	
参数含义	当设置为off时，它会在CREATE FUNCTION期间禁用函数体字符串的验证。禁用验证可避免验证过程的副作用，并避免由于前向引用等问题导致的误报。在代表其他用户加载函数之前，将此参数设置为off；pg_dump会自动执行此操作
是否可session级修改	是
修改后何时生效	Reload即可生效

default_transaction_isolation

参数名称	default_transaction_isolation
数据类型	enum
默认值	read committed
取值范围	{serializable,"repeatable read","read committed","read uncommitted"}
参数单位	
参数含义	本参数用来控制新事务的隔离级别
是否可session级修改	是
修改后何时生效	Reload即可生效

default_transaction_read_only

参数名称	default_transaction_read_only
数据类型	bool
默认值	off，off的意思是read/write
取值范围	off和on
参数单位	
参数含义	本参数用来控制每个新事务的默认只读状态。一个只读事务不能改变(alter)一个非临时表.
是否可session级修改	是
修改后何时生效	Reload即可生效

default_transaction_deferrable

参数名称	default_transaction_deferrable
数据类型	bool
默认值	off
取值范围	off和on
参数单位	

参数含义	当运行在serializable隔离级别中时，一个deferrable的只读事务在它被允许继续处理之前可能会被延迟。然而，一旦它开始执行，它就不会产生任何确保可串行化所需的开销；所以序列化代码没有理由因为并发更新而强制中止该只读事务，这确保了对长时间运行的只读事务是很合适的。本参数控制每个新事务的默认的deferrable状态。当前，本参数对read-write事务是无效的，对隔离级别低于serializable的事务也是无效的
是否可session级修改	是
修改后何时生效	Reload即可生效

session_replication_role

参数名称	session_replication_role
数据类型	enum
默认值	origin
取值范围	{origin,replica,local}
参数单位	
参数含义	控制针对当前session的与复制相关的triggers和rules的触发(firing),设置本参数需要superuser权限，并且会导致之前cache起来的查询计划被丢弃掉。本参数的用意是逻辑复制系统在应用复制的更改时将本参数值设置为replica，这样做的结果是，触发器和规则（没有从默认配置更改）不会在复制副本上触发。详细信息，请参阅ALTER TABLE子句ENABLE TRIGGER和ENABLE RULE PostgreSQL在内部将参数值origin和local视为相同的值，第三方复制软件可能把这两个参数用于他们的目的，比如，使用local来指示一个session的变化不会被复制的session。因为外键是通过trigger实现的，因此，设置本参数为replica也会disable掉所有的外键约束检查。
是否可session级修改	是，仅限于superuser可进行修改。
修改后何时生效	Reload即可生效

statement_timeout

参数名称	statement_timeout
数据类型	integer
默认值	0，零代表禁用超时机制
取值范围	0到2147483647
参数单位	毫秒
参数含义	终止掉运行时间超过本参数值时间长度的SQL语句。如果log_min_error_statement配置参数被设置为error或者更低级别，超时的SQL语句也会被记录下来。超时时间是如下计算的：从SQL语句到PostgreSQL数据库服务器端开始算，直到该语句被服务器处理完成。如果单个简单查询消息(single simple-Query message)中出现多个SQL语句，则将分别对每条语句应用超时.(PostgreSQL13之前的版本通常将超时处理为应用于整个查询字符串)，在扩展查询协议中，当任何与查询相关的消息（Parse、Bind、Execute、describe）到达PostgreSQL数据库时，超时就开始计时，并通过完成Execute或Sync消息而取消。在postgresql.conf中设置本参数值是不推荐的，因为这会影响所有session
是否可session级修改	是
修改后何时生效	Reload即可生效

lock_timeout

参数名称	lock_timeout
数据类型	integer

默认值	0，零代表禁用超时机制
取值范围	0到2147483647
参数单位	毫秒
参数含义	终止掉在尝试获取表、索引、行或其他数据库对象上的锁的等待时间超过本参数值的任何语句。时间限制分别适用于每次锁获取尝试，该限制既适用于显式锁定请求（如LOCK TABLE，或SELECT FOR UPDATE without NOWAIT），也适用于隐式获取的锁。如果指定的值没有单位，则以毫秒为单位。值为零（默认值）将禁用超时。与statement_timeout配置参数不同，本超时只能在锁等待时发生。请注意，如果statement_timeout非零，设置lock_timeout为相同或更大的值是毫无意义的，因为语句超时总是首先触发。如果log_min_error_statement设置为ERROR或更低，则将记录超时的语句。在postgresql.conf中设置本参数值是不推荐的，因为这会影响所有session
是否可session级修改	是
修改后何时生效	Reload即可生效

idle_in_transaction_session_timeout

参数名称	idle_in_transaction_session_timeout
数据类型	integer
默认值	0，零代表禁用超时机制
取值范围	0到2147483647
参数单位	毫秒
参数含义	终止掉那些处于open transaction状态的并且idle时间长度超过本参数值的任何会话。这允许释放该会话持有的任何锁，并重用连接槽；它还允许对仅对此事务可见的元组进行vacuum。
是否可session级修改	是
修改后何时生效	Reload即可生效

vacuum_freeze_table_age

参数名称	vacuum_freeze_table_age
数据类型	integer
默认值	150000000
取值范围	0到2000000000
参数单位	
参数含义	如果表的pg_class.relfrozenxid字段值已经达到本参数值，那么vacuum会执行一次侵略性的扫描（aggressive scan）。侵略性的扫描（aggressive scan）不同于常规vacuum(regular VACUUM), 侵略性的扫描（aggressive scan）会查看每个page以确定page中是否包括未冻结的xids或者mxids，而不是仅仅查看page中是否包括dead tuples。尽管用户可以将此值设置为 0 到 20 亿之间的任何值，但VACUUM会默默地将有效值限制为autovacuum_freeze_max_age 的95% ，以便有机会定期在为表启动反环绕自动清理(anti-wraparound autovacuum)之前手工运行VACUUM。
是否可session级修改	是
修改后何时生效	Reload即可生效

vacuum_freeze_min_age

参数名称	vacuum_freeze_min_age

数据类型	integer
默认值	50000000
取值范围	0到1000000000
参数单位	
参数含义	本参数用于指定VACUUM在扫描表时用于决定是否冻结行版本的截止年龄(cutoff age，以事物计算), 尽管用户可以将此值设置为 0 到 10 亿之间的任何值，但VACUUM会默默地将有效值限制为autovacuum_freeze_max_age参数值的一半，这样强制 autovacuum之间的时间不会过短。
是否可session级修改	
修改后何时生效	

vacuum_multixact_freeze_table_age

参数名称	vacuum_multixact_freeze_table_age
数据类型	integer
默认值	150000000
取值范围	0到2000000000
参数单位	
参数含义	如果表的pg_class.relminmxid字段值已经达到本参数值，那么vacuum会执行一次侵略性的扫描（aggressive scan）。侵略性的扫描（aggressive scan）不同于常规vacuum(regular VACUUM), 侵略性的扫描（aggressive scan）会查看每个page以确定page中是否包括未冻结的xids或者mxids，而不是仅仅查看page中是否包括dead tuples。尽管用户可以将此值设置为 0 到 20 亿之间的任何值，但VACUUM会默默地将有效值限制为autovacuum_multixact_freeze_max_age的95% ，以便有机会定期在为表启动反环绕自动清理(anti-wraparound autovacuum)之前手工运行VACUUM。
是否可session级修改	是
修改后何时生效	Reload即可生效

vacuum_multixact_freeze_min_age

参数名称	vacuum_multixact_freeze_min_age
数据类型	integer
默认值	5000000
取值范围	0到1000000000
参数单位	
参数含义	本参数用于指定VACUUM在扫描表时用于决定是否冻结行版本的multixacts截止年龄(cutoff age), 尽管用户可以将此值设置为 0 到 10 亿之间的任何值，但VACUUM会默默地将有效值限制为autovacuum_multixact_freeze_max_age参数值的一半，这样强制 autovacuum之间的时间不会过短。
是否可session级修改	是
修改后何时生效	Reload即可生效

bytea_output

参数名称	bytea_output
数据类型	enum
默认值	hex

取值范围	{escape,hex}
参数单位	
参数含义	本配置参数定义了对数据类型bytea的输出格式。 bytea数据类型在输入时对两种格式(escape和hex)均接受。escape代表传统的PostgreSQL格式。
是否可session级修改	是
修改后何时生效	Reload即可生效

xmlbinary

参数名称	xmlbinary
数据类型	enum
默认值	base64
取值范围	{base64,hex}
参数单位	
参数含义	设置二进制值在 XML 中的编码方式。例如，这适用于通过函数xmlelement或xmlforest将bytea值转换为XML的情况。可能的值为base64和hex，它们都在 XML 模式标准中定义。
是否可session级修改	是
修改后何时生效	Reload即可生效

xmloption

参数名称	xmloption
数据类型	enum
默认值	content
取值范围	{content,document}
参数单位	
参数含义	本参数用于设置在XML和字符串值之间转换时DOCUMENT或CONTENT是隐式的。在PostgreSQL中，也可以使用下面的SQL标准来设置:SET XML OPTION { DOCUMENT | CONTENT };
是否可session级修改	是
修改后何时生效	Reload即可生效

gin_pending_list_limit

参数名称	gin_pending_list_limit
数据类型	integer
默认值	4096
取值范围	64到2147483647
参数单位	KB
参数含义	设置GIN索引的挂起列表(pending list)的最大大小，该列表在启用fastupdate时使用。如果列表的大小超过此最大值，则通过将列表中的条目批量移动到索引的主数据结构来清除列表。如果指定此值时没有单位，则将其作为千字节。默认值为4MB。通过更改索引存储参数，可以为单个GIN索引重写此设置.
是否可session级修改	是
修改后何时生效	Reload即可生效

PG函数参考手册

PostgreSQL 函数参考手册

current_catalog

函数名称	current_catalog
参数数据类型	无输入参数
函数返回值数据类型	Name
函数含义	获得当前的database 名称。 注意：在SQL标准中，databases被称之为catalogs。

使用举例	`postgres=# select current_catalog;`
	` current_catalog`
	`----------`
	` postgres`
	`(1 row)`
	
	`postgres=#`

current_database()

函数名称	current_database()
参数数据类型	无输入参数
函数返回值数据类型	name
函数含义	获得当前的database 名称。

使用举例	`postgres=# select current_database();`
	` current_database`
	`----------`
	` postgres`
	`(1 row)`
	
	`postgres=#`

current_query()

函数名称	current_query()
参数数据类型	无输入参数
函数返回值数据类型	text
函数含义	获得client端提交的当前执行的SQL语句的文本

使用举例	

```
postgres=#   BEGIN;
BEGIN
postgres=*# select 1,current_query();
 ?column? |       current_query
----------+---------------------------
        1 | select 1,current_query();
(1 row)

postgres=*# select 2,current_query();
 ?column? |       current_query
----------+---------------------------
        2 | select 2,current_query();
(1 row)

postgres=*# ROLLBACK;
ROLLBACK
postgres=#
```

current_role

函数名称	current_role
参数数据类型	无输入参数
函数返回值数据类型	name
函数含义	与current_user作用相同，获得当前执行上下文的username
使用举例	

```
postgres=# select current_role;
 current_role
--------------
 pg131
(1 row)

postgres=#
```

current_schema

函数名称	current_schema
参数数据类型	无输入参数
函数返回值数据类型	name
函数含义	获得search path中的第一个schema名称（当search path为空时，返回空值）。当表或者其他对象建立不带schema名称时，本函数返回的schema就是该表或者该对象的schema

使用举例	

```
postgres=# select current_schema;
 current_schema
----------------
 public
(1 row)

postgres=#
```

current_schema()

函数名称	current_schema()
参数数据类型	无输入参数
函数返回值数据类型	name
函数含义	作用与current_schema函数相同：获得search path中的第一个schema名称（当search path为空时，返回空值）。当表或者其他对象建立不带schema名称时，本函数返回的schema就是该表或者该对象的schema
使用举例	

```
postgres=# select current_schema();
 current_schema
----------------
 public
(1 row)

postgres=#
```

current_schemas(include_implicit boolean)

函数名称	current_schemas(include_implicit boolean)
参数数据类型	bool
函数返回值数据类型	name
函数含义	以优先级顺序返回有效search_path中当前所有模式名称，如果入参为true，那么在函数返回结果中会包括隐式搜索的系统模式pg_catalog

使用举例	<pre>postgres=# select current_schemas(false);
	 current_schemas
	---------------- 	
	 {public}
	(1 row)
	
	postgres=# select current_schemas(true);
	  current_schemas
	--------------------- 	
	 {pg_catalog,public}
	(1 row)
	
	postgres=#</pre>

current_user

函数名称	current_user
参数数据类型	无输入参数
函数返回值数据类型	name
函数含义	获得当前执行上下文的username

使用举例	<pre>postgres=# select current_user;
	 current_user
	-------------- 	
	 pg131
	(1 row)
	
	postgres=#</pre>

inet_client_addr()

函数名称	inet_client_addr()
参数数据类型	无输入参数
函数返回值数据类型	inet
函数含义	获得当前client的ip地址。如果返回值为空，表示当前连接是经过Unix-domain socket的

使用举例	`postgres=# select inet_client_addr();`
	` inet_client_addr`
	`------------------`
	
	`(1 row)`
	
	`postgres=#`

inet_client_port()

函数名称	inet_client_port()
参数数据类型	无输入参数
函数返回值数据类型	integer
函数含义	获得当前client的端口号。如果返回值为空，表示当前连接是经过Unix-domain socket的

使用举例	`postgres=# select inet_client_port();`
	` inet_client_port`
	`------------------`
	
	`(1 row)`
	
	`postgres=#`

inet_server_addr()

函数名称	inet_server_addr()
参数数据类型	无输入参数
函数返回值数据类型	inet
函数含义	获得当前连接的pg数据库服务器的ip地址。如果返回值为空，表示当前连接是经过Unix-domain socket的。

使用举例	`postgres=# select inet_server_port();`
	` inet_server_port`
	`------------------`
	
	`(1 row)`
	
	`postgres=#`

inet_server_port()

函数名称	inet_server_port()
参数数据类型	无输入参数

函数返回值数据类型	integer
函数含义	获得当前连接的pg数据库服务器的端口号。如果返回值为空，表示当前连接是经过Unix-domain socket的。

使用举例	`postgres=# select inet_server_port();`
	` inet_server_port`
	`------------------`
	
	`(1 row)`
	
	`postgres=#`

pg_backend_pid()

函数名称	pg_backend_pid()
参数数据类型	无输入参数
函数返回值数据类型	integer
函数含义	获得附加到当前session的backend进程的进程ID

使用举例	`postgres=# select pg_backend_pid();`
	` pg_backend_pid`
	`----------------`
	`4068`
	`(1 row)`
	
	`postgres=#`

pg_blocking_pids(integer)

函数名称	pg_blocking_pids(integer)
参数数据类型	Integer
函数返回值数据类型	Integer[]

函数含义	返回入参pid的blocking pid(即:阻塞者pid)若是本函数返回空值，表示不存在阻塞者若是本函数返回值是零，那么表示阻塞者是prepared transacion，见下面的输出：
	
	`postgres=# select pg_blocking_pids(12947); pg_blocking_pids`
	`------------------`
	`{0}`
	`(1 row)`
	` postgres=#`
	
	频繁调用本函数会对数据库性能有影响，因为本函数在短时间内对lock manager’s shared state进行排他访问(exclusive accesss)

使用举例	

```
Session1:
postgres=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
          32262
(1 row)

postgres=# CREATE TABLE tbl_students(rno int, name character
varying(10));
CREATE TABLE
postgres=# BEGIN TRANSACTION;
BEGIN
postgres=*# LOCK tbl_students IN ACCESS EXCLUSIVE MODE;
LOCK TABLE
postgres=*#


session2:
postgres=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
          32439
(1 row)

postgres=# INSERT INTO tbl_students VALUES (1,'Anvesh');


session3:
postgres=# SELECT pg_blocking_pids(32439);
 pg_blocking_pids
------------------
 {32262}
(1 row)

postgres=#
```

pg_conf_load_time()

函数名称	pg_conf_load_time()
参数数据类型	无输入参数
函数返回值数据类型	timestamp with time zone
函数含义	返回PG配置参数文件最后一次被reload的时间

使用举例	

```
postgres=# select pg_conf_load_time();
       pg_conf_load_time
-------------------------------
 2021-03-22 12:46:18.962643+08
(1 row)

postgres=#
```

pg_current_logfile([text]);

函数名称	pg_current_logfile([text]);
参数数据类型	无输入参数或者text型输入参数 (csvlog、stderr)
函数返回值数据类型	Text
函数含义	返回正在被logging collector进程使用的log file的path(是log_directory的值)和文件名。当logging collector是被禁用时，本函数返回结果是null。当有不同格式的多种log files存在时，不带任何参数的pg_current_logfile()函数返回有序列表(stderr、csvlog)中找到的第一种格式的文件path和文件名。当没有这两个格式的文件时，本函数返回NULL

使用举例	

```
postgres=# select pg_current_logfile();
          pg_current_logfile
--------------------------------------
 log/postgresql-2021-03-22_124618.log
(1 row)

postgres=# select pg_current_logfile('stderr');
          pg_current_logfile
--------------------------------------
 log/postgresql-2021-03-22_124618.log
(1 row)

postgres=# select pg_current_logfile('csvlog');
 pg_current_logfile
--------------------

(1 row)

postgres=# show log_destination ;
 log_destination
-----------------
 stderr
(1 row)

postgres=#
```

pg_my_temp_schema()

函数名称	pg_my_temp_schema()
参数数据类型	无输入参数
函数返回值数据类型	oid
函数含义	返回当前会话的临时schema的OID，如果没有临时schema，本函数返回0（即：数字零）
使用举例	

```
postgres=# select pg_my_temp_schema();
 pg_my_temp_schema
-------------------
 0
(1 row)

postgres=#
```

pg_is_other_temp_schema(oid)

函数名称	pg_is_other_temp_schema(oid)
参数数据类型	oid
函数返回值数据类型	boolean
函数含义	当入参的oid是其他session中临时schema的oid时，本函数返回true。

使用举例	`postgres=# select pg_is_other_temp_schema(16426);`
	` pg_is_other_temp_schema`
	`------------------------`
	` f`
	`(1 row)`
	
	`postgres=#`

pg_jit_available()

函数名称	pg_jit_available()
参数数据类型	无输入参数
函数返回值数据类型	boolean
函数含义	如果jit编译器扩展是可用的并且jit配置参数设置为on的话，本函数返回true

使用举例	`postgres=# select pg_jit_available();`
	` pg_jit_available`
	`------------------`
	` f`
	`(1 row)`
	
	`postgres=# select * from pg_config where name='CONFIGURE'`
	`AND SETTING LIKE '%--with-llvm%';`
	` name | setting`
	`------+---------`
	`(0 rows)`
	
	`postgres=# show jit;`
	` jit`
	`-----`
	` on`
	`(1 row)`
	
	`postgres=#`

pg_listening_channels()

函数名称	pg_listening_channels()
参数数据类型	无输入参数
函数返回值数据类型	setof text
函数含义	当前session正在监听的异步通知通道的名称

使用举例	`postgres=# select pg_listening_channels();`
	` pg_listening_channels`
	`----------`
	`(0 rows)`
	
	`postgres=#`

pg_notification_queue_usage()

函数名称	pg_notification_queue_usage()
参数数据类型	无输入参数
函数返回值数据类型	double precision
函数含义	返回被正在等待处理的通知占据的异步通知队列最大大小的比例。

使用举例	`postgres=# select pg_notification_queue_usage();`
	` pg_notification_queue_usage`
	`----------`
	`0`
	`(1 row)`
	
	`postgres=#`

pg_postmaster_start_time()

函数名称	pg_postmaster_start_time()
参数数据类型	无输入参数
函数返回值数据类型	timestamp with time zone
函数含义	返回PG intance 启动的时间点

使用举例	`postgres=# select pg_postmaster_start_time();`
	`  pg_postmaster_start_time`
	`----------`
	` 2021-03-23 14:01:33.352011+08`
	`(1 row)`
	
	`postgres=#`

pg_safe_snapshot_blocking_pids(integer)

函数名称	pg_safe_snapshot_blocking_pids(integer)
参数数据类型	Integer
函数返回值数据类型	integer[]
函数含义	返回阻止指定服务器进程ID获取安全快照的进程ID若是没有阻塞，则返回空array。一个运行SERIALIZABLE事务的session阻塞了一个SERIALIZABLE READ ONLY DEFERRABLE事务，防止后者获得snapshot，直到后者确认可以安全的避免获取任何谓词锁。频繁调用本函数会对数据库性能有影响，因为它在短时间内需要访问谓词锁管理器的共享状态(predicate lock manager’s shared state)

使用举例	`postgres=# select pg_safe_snapshot_blocking_pids(1234);`
	` pg_safe_snapshot_blocking_pids`
	`---------- `
	` {}`
	`(1 row)`
	
	`postgres=#`

pg_trigger_depth()

函数名称	pg_trigger_depth()
参数数据类型	无输入参数
函数返回值数据类型	integer
函数含义	返回PostgreSQL触发器的当前嵌套级别(如果没有从触发器内部直接或间接调用，则为返回值为0)

使用举例	`postgres=# select pg_trigger_depth();`
	` pg_trigger_depth`
	`---------- `
	`  0`
	`(1 row)`
	
	`postgres=#`

user

函数名称	user
参数数据类型	无输入参数
函数返回值数据类型	name
函数含义	获得当前执行上下文的username，等同于current_user函数

使用举例	`postgres=# select user;`
	` user`
	`-------`
	` pg131`
	`(1 row)`
	
	`postgres=#`

Version()

函数名称	Version()
参数数据类型	无输入参数
函数返回值数据类型	text
函数含义	返回带有数据库版本号的字符串，您也可以查询server_version配置参数获得数据库版本号。对于机器可读的版本，请使用server_version_num配置参数。在软件开发过程中应该使用server_version_num配置参数或者PqserverVersion，而不是去解析文本的版本号(见下面的使用举例)

使用举例	`postgres=# select version();`
	`version`
	`----------`
	`----------`
	` PostgreSQL 13.1 on x86_64-pc-linux-gnu, compiled by gcc`
	`(GCC) 4.8.5 20150623 (Red Hat 4.8.5-44), 64-bit`
	`(1 row)`
	
	`postgres=#`

has_any_column_privilege(user, table, privilege)

函数名称	has_any_column_privilege(user, table, privilege)
参数数据类型	user, table, privilege或者table, privilege
函数返回值数据类型	boolean
函数含义	确定一个user是否对当前PG database中的某个table任一列有列权限确定当前user是否对当前PG database中的某个table任一列有列权限权限类型是SELECT, INSERT, UPDATE,REFERENCES的组合。

使用举例	

```
[pg131@VM-0-8-centos ~]$ psql -d cwbase3 -U lc0039999
psql (13.1)
Type "help" for help.

cwbase3=> \d
         List of relations
 Schema | Name  | Type  |   Owner
--------+-------+-------+-----------
 public | test1 | table | lc0039999
(1 row)

cwbase3=> select has_any_column_privilege('lc0039999'
,'test1','select');
 has_any_column_privilege
--------------------------
 t
(1 row)

cwbase3=>
cwbase3=> select has_any_column_privilege('test1'
,'select,insert');
 has_any_column_privilege
--------------------------
 t
(1 row)

cwbase3=>

cwbase3=> select has_any_column_privilege('test1','select');
 has_any_column_privilege
--------------------------
 t
(1 row)

cwbase3=>
```

has_column_privilege(user, table, column, privilege)

函数名称	has_column_privilege(user, table, column, privilege);
参数数据类型	user, table, column, privilege或者table, column, privilege
函数返回值数据类型	boolean
函数含义	特定user对特定表的某一列是否具有某种权限当前user对特定表的某一列是否具有某种权限权限类型是SELECT, INSERT, UPDATE,REFERENCES的组合。

使用举例	

```
cwbase3=> \d test1
                Table "public.test1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           |          |

cwbase3=>
cwbase3=> select has_column_privilege('lc0039999','test1'
,'id','select');
 has_column_privilege
----------------------
 t
(1 row)

cwbase3=>
cwbase3=> select has_column_privilege('test1','id',
'select');
 has_column_privilege
----------------------
 t
(1 row)

cwbase3=>
```

has_database_privilege

函数名称	has_database_privilege
参数数据类型	user, database, privilege或者database, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定database有某种权限当前user是否对特定database有某种权限权限类型是CREATE, CONNECT, TEMPORARY, or TEMP的组合。

使用举例	

```
postgres=# select has_database_privilege('lc0039999'
,'cwbase3','create,connect,temp');
 has_database_privilege
------------------------
 t
(1 row)

postgres=# select has_database_privilege('lc0019999'
,'cwbase3','create,connect,temp');
 has_database_privilege
------------------------
 t
(1 row)

postgres=#
```

has_foreign_data_wrapper_privilege

函数名称	has_foreign_data_wrapper_privilege
参数数据类型	user, fdw, privilege或者fdw, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定fdw有usage权限当前user是否对特定fdw有usage权限此处的权限只能指定usage
使用举例	

```
postgres=# select  has_foreign_data_wrapper_privilege
('lc0019999','file_fdw','usage');
 has_foreign_data_wrapper_privilege
------------------------------------
 f
(1 row)

postgres=#
postgres=# select  has_foreign_data_wrapper_privilege
('file_fdw','usage');
 has_foreign_data_wrapper_privilege
------------------------------------
 t
(1 row)

postgres=#
```

has_function_privilege

函数名称	has_function_privilege
参数数据类型	user, function, privilege或者function, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定function有EXECUTE权限当前user是否对特定function有EXECUTE权限此处的权限只能是EXECUTE

使用举例	`postgres=# select has_function_privilege('lc0039999' ,'version()','execute');`
	`has_function_privilege`
	`------------------------`
	`t`
	`(1 row)`
	` postgres=# postgres=# select has_function_privilege ('version()','execute');`
	` has_function_privilege`
	` ------------------------`
	` t`
	` (1 row)`
	`postgres=#`

has_language_privilege

函数名称	has_language_privilege
参数数据类型	user, language, privilege或者language, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定procedural language有usage权限当前user是否对特定procedural language有usage权限此处的权限必须是usage

使用举例	`postgres=# select has_language_privilege('lc0019999','sql' ,'usage');`
	`has_language_privilege`
	`------------------------`
	`t`
	`(1 row)`
	`postgres=# select has_language_privilege('plpgsql', 'usage');`
	`has_language_privilege`
	`------------------------`
	`t`
	`(1 row)`
	`postgres=#`
	
	注意：此处的language可以从pg_language系统表查询到。

has_schema_privilege

函数名称	has_schema_privilege
参数数据类型	user, schema, privilege或者schema, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定schema有usage或者create权限当前user是否对特定schema有usage或者create权限此处的权限必须是create或者usage的组合

使用举例	

```
postgres=# \dnS
      List of schemas
        Name        | Owner
--------------------+-------
 information_schema | pg131
 pg_catalog         | pg131
 pg_toast           | pg131
 public             | pg131
 s_abc              | pg131
(5 rows)

postgres=# select has_schema_privilege('lc0019999','s_abc'
,'usage');
 has_schema_privilege
----------------------
 f
(1 row)

postgres=# select has_schema_privilege('lc0019999','s_abc'
,'usage,create');
 has_schema_privilege
----------------------
 f
(1 row)

postgres=#
```

has_sequence_privilege

函数名称	has_sequence_privilege
参数数据类型	user, sequence, privilege或者sequence, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定sequence有特定权限当前user是否对特定sequence有特定权限此处的权限是USAGE, SELECT, UPDATE的组合。

使用举例	

```
postgres=# \ds+
                                   List of relations
 Schema |     Name     |   Type   | Owner | Persistence |
Size    | Description
--------+--------------+----------+-------+-------------+
------------+-------------
 public | gen_y_c1_seq | sequence | pg131 | permanent   |
8192 bytes |
(1 row)

postgres=# select has_sequence_privilege('pg131'
,'gen_y_c1_seq','usage');
 has_sequence_privilege
------------------------
 t
(1 row)

postgres=# select has_sequence_privilege('lc0039999'
,'gen_y_c1_seq','usage');
 has_sequence_privilege
------------------------
 f
(1 row)

postgres=#
```

has_server_privilege

函数名称	has_server_privilege
参数数据类型	user, server, privilege或者server, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定foreign server有usage权限特定user是否对特定foreign server有usage权限此处的权限必须是usage

使用举例	

```
postgres=# select has_server_privilege('s1','usage');
has_server_privilege
----------------------
t
(1 row)
postgres=# select has_server_privilege('s1','USAGE');
has_server_privilege
----------------------
t
(1 row)
postgres=# select has_server_privilege('pg123','s1',
'USAGE');
has_server_privilege
----------------------
t
(1 row)
postgres=#
```

has_table_privilege

函数名称	has_table_privilege
参数数据类型	user, table, privilege或者table, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定table有特定权限当前user是否对特定table有特定权限此处的权限是指如下的组合： SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER，还可以带有WITH GRANT OPTION用来测试privilege是否带有grant option。当权限字符串有多个值(比如’ insert, select')时,只要有任何一个权限是满足的，本函数会返回t有关对权限字符串的说明：权限字符串不区分大小写权限字符串的前后可以有一个或者多个空格（权限名字内部不能有空格）

使用举例

```
postgres=# select has_table_privilege('t1','  select,
insert ');
 has_table_privilege
---------------------
 t
(1 row)

postgres=#
[pg131@VM-0-8-centos ~]$ psql -d cwbase3 -U lc0039999
psql (13.1)
Type "help" for help.

cwbase3=>
cwbase3=> grant select on test1 to lc0019999;
GRANT
cwbase3=> exit
[pg131@VM-0-8-centos ~]$ psql -d cwbase3 -U lc0019999
psql (13.1)
Type "help" for help.

cwbase3=> select * from test1;
 id
 ----
  1
  2
  3

  4
  5
(6 rows)

cwbase3=> select has_table_privilege('test1','insert, select
');
 has_table_privilege
---------------------
 t
(1 row)

cwbase3=> select has_table_privilege('test1','insert');
 has_table_privilege
---------------------
 f
```

has_tablespace_privilege

函数名称	has_tablespace_privilege
参数数据类型	user, tablespace, privilege或者tablespace, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定tablesapce有特定权限当前user是否对特定tablespace有特定权限此处的权限必须是create

使用举例	`postgres=# select has_tablespace_privilege('ts1','create ');`
	`has_tablespace_privilege`
	`----------`
	`t`
	`(1 row)`
	`postgres=# select has_tablespace_privilege('lc0019999', 'ts1','create ');`
	`has_tablespace_privilege`
	`----------`
	`f`
	`(1 row)`
	`postgres=#`

has_type_privilege

函数名称	has_type_privilege
参数数据类型	user, type, privilege或者type, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定type有特定权限当前user是否对特定type有特定权限特定type可以使用type的name，也可以使用oid::regtype的形式此处的权限必须是USAGE

使用举例	

```
postgres=# select has_type_privilege('bytea','usage');
has_type_privilege
--------------------
t
(1 row)
postgres=# select has_type_privilege('lc0019999','bytea'
,'usage');
has_type_privilege
--------------------
t
(1 row)
postgres=# select has_type_privilege('lc0019999'
,'17:regtype','usage');
ERROR: syntax error at or near "17"
CONTEXT: invalid type name "17:regtype"
postgres=# select has_type_privilege('lc0019999',17::
regtype,'usage');
has_type_privilege
--------------------
t
(1 row)
postgres=# select has_type_privilege('lc0019999'
,'17::regtype','usage');
ERROR: syntax error at or near "17"
CONTEXT: invalid type name "17::regtype"
postgres=#
```

pg_has_role

函数名称	pg_has_role
参数数据类型	user, role, privilege或者role, privilege
函数返回值数据类型	boolean
函数含义	特定user是否对特定role group有特定权限当前user是否对特定role group有特定权限此处的权限必须是member或者usage当权限是member时，表示特定user是否是特定role group的member当权限是usage时，表示在不执行set role时，特定user是否有特定role的权限

使用举例	

```
postgres=# create role editors;
CREATE ROLE
postgres=# create user maxwell;
CREATE ROLE
postgres=# create user ernest;
CREATE ROLE
postgres=# grant authors to editors; --editors can do what
authors can do
GRANT ROLE
postgres=# grant editors to maxwell; --maxwell is an editor
GRANT ROLE
postgres=# grant authors to ernest; --ernest is an author
GRANT ROLE
postgres=# select pg_has_role('maxwll','editors','member');
ERROR:  role "maxwll" does not exist
postgres=# select pg_has_role('maxwell','editors','member');
 pg_has_role
-------------
 t
(1 row)

postgres=# select pg_has_role('maxwell','editors','usage');
 pg_has_role
-------------
 t
(1 row)

postgres=# select pg_has_role('maxwell','ernest','usage');
 pg_has_role
-------------
 f
(1 row)

postgres=# select pg_has_role('maxwell','ernest','member');
 pg_has_role
-------------
 f
(1 row)

postgres=#
```

row_security_active

函数名称	row_security_active
参数数据类型	Table，可以是table name，也可以是table的oid
函数返回值数据类型	boolean
函数含义	在current_user的上下文中，特定table是否已开启row security
使用举例	

```
postgres=# select oid from pg_class where relname='test';
  oid
-------
 16860
(1 row)

postgres=# select row_security_active('test');
 row_security_active
---------------------
 f
(1 row)

postgres=# select row_security_active(16860);
 row_security_active
---------------------
 f
(1 row)

postgres=#
```

acldefault

函数名称	acldefault
参数数据类型	type, ownerId
函数返回值数据类型	aclitem[]
函数含义	返回属于某个user oid对象的默认访问权限。当一个对象的ACL条目为null时这些访问权限是该对象的默认权限Type参数的取值如下：'c' for COLUMN’ r' for TABLE 以及类似对象’ s' for SEQUENCE’ d' for DATABASE’ f' for FUNCTION 或者PROCEDURE’ l' for LANGUAGE’ L' for LARGE OBJECT’ n' for SCHEMA’ t' for TABLESPACE’ F' for FOREIGN DATA WRAPPER’ S' for FOREIGN SERVER’ T' for TYPE or DOMAIN

使用举例	

```
postgres=# select acldefault('f',16877);
acldefault
-------------------------------------
{=X/lc0039999,lc0039999=X/lc0039999}
(1 row)
postgres=#
```

aclexplode

函数名称	aclexplode
参数数据类型	aclitem[]
函数返回值数据类型	setof record
函数含义	解析 acl 权限
使用举例	

```
postgres=# select aclexplode
('{=X/lc0039999,lc0039999=X/lc0039999}');
 aclexplode
-------------------------
 (16877,0,EXECUTE,f)
 (16877,16877,EXECUTE,f)
 (2 rows)
postgres=# postgres=# select \* from aclexplode
('{=X/lc0039999,lc0039999=X/lc0039999}');
 grantor | grantee | privilege_type | is_grantable
---------+---------+----------------+--------------
 16877 | 0 | EXECUTE | f
 16877 | 16877 | EXECUTE | f
(2 rows)
postgres=#
```

makeaclitem

函数名称	makeaclitem
参数数据类型	grantee, grantor, privilege, grantable
函数返回值数据类型	aclitem
函数含义	构建aclitem

使用举例	

```
postgres=# select \* from makeaclitem(16877,16877,'EXECUTE'
,false);
makeaclitem
------------------------
lc0039999=X/lc0039999
(1 row)
postgres=#
```

pg_collation_is_visible

函数名称	pg_collation_is_visible
参数数据类型	collation_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定collation是否可见。

使用举例	

```
postgres=# select * from pg_collation where collname =
'zh_CN';
  oid  | collname | collnamespace | collowner | collprovider
| collisdeterministic | collencoding | collcollate |
collctype  | collversion
-------+----------+---------------+-----------+
--------------+--------------------+--------------+
-------------+------------+-------------
 13086 | zh_CN    |            11 |        10 | c
| t                   |            2 | zh_CN       | zh_CN
| 2.17
 13243 | zh_CN    |            11 |        10 | c
| t                   |            6 | zh_CN.utf8  | zh_CN
.utf8 | 2.17
(2 rows)

postgres=# select  pg_collation_is_visible(13086);
 pg_collation_is_visible
-------------------------
 f
(1 row)

postgres=# select  pg_collation_is_visible(13243);
 pg_collation_is_visible
-------------------------
 t
(1 row)

postgres=#
postgres=# select  pg_collation_is_visible('"zh_CN"'
::regcollation);
 pg_collation_is_visible
-------------------------
 t
(1 row)

postgres=#
```

pg_conversion_is_visible

函数名称	pg_conversion_is_visible
参数数据类型	conversion_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定conversion是否可见

使用举例	

```
postgres=# select *  from pg_conversion  where conname like
'%cn%';
 oid  |    conname     | connamespace | conowner |
conforencoding | contoencoding |    conproc     | condefault
------+----------------+--------------+----------+
----------------+---------------+----------------+
------------
 4422 | euc_cn_to_mic  |           11 |       10 |
2 |             7 | euc_cn_to_mic  | t
 4423 | mic_to_euc_cn  |           11 |       10 |
7 |             2 | mic_to_euc_cn  | t
 4480 | euc_cn_to_utf8 |           11 |       10 |
2 |             6 | euc_cn_to_utf8 | t
 4481 | utf8_to_euc_cn |           11 |       10 |
6 |             2 | utf8_to_euc_cn | t
(4 rows)

postgres=# select pg_conversion_is_visible(4480);
 pg_conversion_is_visible
--------------------------
 t
(1 row)

postgres=#
```

pg_function_is_visible

函数名称	pg_function_is_visible
参数数据类型	function_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定function、procedures、aggregates是否可见。对于函数，如果路径前面没有相同名称和参数数据类型的对象，则搜索路径中的对象可见

使用举例	

```
postgres=# select oid from pg_proc where proname='version';
 oid
-----
  89
(1 row)

postgres=# select pg_function_is_visible(89);
 pg_function_is_visible
------------------------
 t
(1 row)

postgres=#
postgres=# select pg_function_is_visible('version'::
regproc);
 pg_function_is_visible
------------------------
 t
(1 row)

postgres=# select pg_function_is_visible('version()'
::regprocedure);
 pg_function_is_visible
------------------------
 t
(1 row)

postgres=#
```

pg_opclass_is_visible

函数名称	pg_opclass_is_visible
参数数据类型	opclass_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定operator class是否可见考虑因素是operator class name和相关的索引访问方法。

使用举例	

```
postgres=# select oid from pg_opclass where opcname
='array_ops';
  oid
-------
 10000
 10001
 10063
(3 rows)

postgres=# select pg_opclass_is_visible(10000);
 pg_opclass_is_visible
-----------------------
 t
(1 row)

postgres=#
```

pg_operator_is_visible

函数名称	pg_operator_is_visible
参数数据类型	operator_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定operator 是否可见对于运算符，如果路径前面没有相同名称和参数数据类型的对象， 则搜索路径中的对象可见

使用举例	

```
postgres=# select oprname from pg_operator where oid=15;
 oprname
---------
 =
(1 row)

postgres=# select pg_operator_is_visible(15);
 pg_operator_is_visible
------------------------
 t
(1 row)

postgres=#
postgres=#  select pg_operator_is_visible
('=(integer,integer)'::regoperator);
 pg_operator_is_visible
------------------------
 t
(1 row)

postgres=#
```

pg_opfamily_is_visible

函数名称	pg_opfamily_is_visible
参数数据类型	opclass_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定operator family是否可见
使用举例	

```
postgres=# select *  from pg_opfamily where oid='397';
 oid | opfmethod |  opfname  | opfnamespace | opfowner
-----+-----------+-----------+--------------+----------
 397 |       403 | array_ops |           11 |       10
(1 row)

postgres=# select pg_opfamily_is_visible(397);
 pg_opfamily_is_visible
------------------------
 t
(1 row)

postgres=#
```

pg_statistics_obj_is_visible

函数名称	pg_statistics_obj_is_visible
参数数据类型	stat_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定operator class是否可见
使用举例	

pg_table_is_visible

函数名称	pg_table_is_visible
参数数据类型	table_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定的table、views、materialized views、indexes、 sequences、foreign tables是否可见
使用举例	

```
postgres=# select oid from pg_class where relname='data1';
  oid
-------
 16384
(1 row)

postgres=# select pg_table_is_visible(16384);
 pg_table_is_visible
---------------------
 t
(1 row)

postgres=#
postgres=# SELECT pg_table_is_visible('public.data1'
::regclass);
 pg_table_is_visible
---------------------
 t
(1 row)

postgres=#
```

pg_ts_config_is_visible

函数名称	pg_ts_config_is_visible
参数数据类型	config_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定text search configuration是否可见

使用举例	

```
postgres=# select oid from pg_ts_config where cfgname
='greek';
  oid
-------
 13861
(1 row)

postgres=# select pg_ts_config_is_visible(13861);
 pg_ts_config_is_visible
-------------------------
 t
(1 row)

postgres=#
postgres=# select pg_ts_config_is_visible('greek'::
regconfig);
 pg_ts_config_is_visible
-------------------------
 t
(1 row)

postgres=#
```

pg_ts_dict_is_visible

函数名称	pg_ts_dict_is_visible
参数数据类型	dict_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定text search dictionary是否可见

使用举例	

```
postgres=# select oid,dictname,dictinitoption from
pg_ts_dict where dictname='greek_stem';
  oid  |  dictname  |   dictinitoption
-------+------------+--------------------
 13860 | greek_stem | language = 'greek'
(1 row)

postgres=# select pg_ts_dict_is_visible(13860);
 pg_ts_dict_is_visible
-----------------------
 t
(1 row)

postgres=#
postgres=#  select pg_ts_dict_is_visible('greek_stem'
::regdictionary);
 pg_ts_dict_is_visible
-----------------------
 t
(1 row)

postgres=#
```

pg_ts_parser_is_visible

函数名称	pg_ts_parser_is_visible
参数数据类型	parser_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定text search parser是否可见

使用举例	

```
postgres=# select * from pg_ts_parser ;
 oid  | prsname | prsnamespace |  prsstart  |    prstoken
|  prsend  |  prsheadline  |  prslextype
------+---------+--------------+------------+
----------------+----------+---------------+--------------
 3722 | default |           11 | prsd_start | prsd_nexttoken
| prsd_end | prsd_headline | prsd_lextype
(1 row)

postgres=# select pg_ts_parser_is_visible(3722);
 pg_ts_parser_is_visible
-------------------------
 t
(1 row)

postgres=#
```

pg_ts_template_is_visible

函数名称	pg_ts_template_is_visible
参数数据类型	template_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定text search template是否可见

使用举例	

```
postgres=# select * from pg_ts_template ;
  oid  | tmplname  | tmplnamespace |    tmplinit    |
tmpllexize
-------+-----------+---------------+----------------+
------------------
  3727 | simple    |            11 | dsimple_init   |
dsimple_lexize
  3730 | synonym   |            11 | dsynonym_init  |
dsynonym_lexize
  3733 | ispell    |            11 | dispell_init   |
dispell_lexize
  3742 | thesaurus |            11 | thesaurus_init |
thesaurus_lexize
 13845 | snowball  |            11 | dsnowball_init |
dsnowball_lexize
(5 rows)

postgres=# select pg_ts_temp

postgres=# select pg_ts_template_is_visible(3730);
 pg_ts_template_is_visible
---------------------------
 t
(1 row)

postgres=#
```

pg_type_is_visible

函数名称	pg_type_is_visible
参数数据类型	type_oid
函数返回值数据类型	boolean
函数含义	在当前schema search path中，特定type、domain是否可见

使用举例	

```
postgres=# select oid,typname from pg_type where typname
='bool';
 oid | typname
-----+---------
  16 | bool
(1 row)

postgres=# select pg_type_is_visible(16);
 pg_type_is_visible
--------------------
 t
(1 row)

postgres=#
postgres=# SELECT pg_type_is_visible('pg_catalog.bool'
::regtype);
 pg_type_is_visible
--------------------
 t
(1 row)

postgres=#
```

format_type

函数名称	format_type
参数数据类型	type_oid, typemod
函数返回值数据类型	Text
函数含义	得到一个data type的SQL Name

使用举例

```
postgres=# select * from pg_type where typname='bool';
-[ RECORD 1 ]--+---------
oid            | 16
typname        | bool
typnamespace   | 11
typowner       | 10
typlen         | 1
typbyval       | t
typtype        | b
typcategory    | B
typispreferred | t
typisdefined   | t
typdelim       | ,
typrelid       | 0
typelem        | 0
typarray       | 1000
typinput       | boolin
typoutput      | boolout
typreceive     | boolrecv
typsend        | boolsend
typmodin       | -
typmodout      | -
typanalyze     | -
typalign       | c
typstorage     | p
typnotnull     | f
typbasetype    | 0
typtypmod      | -1
typndims       | 0
typcollation   | 0
typdefaultbin  |
typdefault     |
typacl         |

postgres=#
postgres=# select format_type(16,0);
 format_type
-------------
 boolean
(1 row)
```

pg_get_constraintdef

函数名称	pg_get_constraintdef
参数数据类型	constraint_oid或者constraint_oid, pretty_bool
函数返回值数据类型	text
函数含义	得到一个constraint的定义

使用举例	`postgres=# select pg_get_constraintdef(16918);`
	`pg_get_constraintdef`
	`----------`
	`PRIMARY KEY (c1)`
	`(1 row)`
	`postgres=#`

pg_get_expr

函数名称	pg_get_expr
参数数据类型	pg_node_tree, relation_oid,或者 pg_node_tree, relation_oid, pretty_bool
函数返回值数据类型	Text
函数含义	反编译指定expression的内部格式，假设：其中的任何变量引用第二个参数指示的relation

使用举例	`postgres=# CREATE TABLE tab1 (a int, b int) PARTITION BY RANGE(a);`
	`CREATE TABLE`
	`postgres=# CREATE TABLE tab1_p1 PARTITION OF tab1 FOR VALUES FROM (0) TO (100);`
	`CREATE TABLE`
	`postgres=# CREATE TABLE tab1_p2 PARTITION OF tab1 FOR VALUES FROM (100) TO (200);`
	`CREATE TABLE`
	`postgres=# select relname, pg_get_expr(relpartbound, oid) from pg_class where relispartition and relname ~ 'tab1' order by relname;`
	`relname | pg_get_expr`
	`---------+----------`
	`tab1_p1 | FOR VALUES FROM (0) TO (100)`
	`tab1_p2 | FOR VALUES FROM (100) TO (200)`
	`(2 rows)`
	
	`postgres=#`

pg_get_functiondef

函数名称	pg_get_functiondef

参数数据类型	func_oid
函数返回值数据类型	text
函数含义	获得function或者procedure的定义

使用举例	<pre>postgres=# select pg_get_functiondef(89);
	pg_get_functiondef
	---------- 	
	CREATE OR REPLACE FUNCTION pg_catalog.version()+ RETURNS
	text + LANGUAGE internal + STABLE PARALLEL SAFE STRICT + AS
	$function$pgsql_version$function$
	
	(1 row)
	postgres=#</pre>

pg_get_function_arguments

函数名称	pg_get_function_arguments
参数数据类型	func_oid
函数返回值数据类型	text
函数含义	获得function或者procedure的传入参数清单

使用举例	

```
postgres=# select oid from pg_proc where proname
='has_database_privilege';
-[ RECORD 1 ]
oid | 2250
-[ RECORD 2 ]
oid | 2251
-[ RECORD 3 ]
oid | 2252
-[ RECORD 4 ]
oid | 2253
-[ RECORD 5 ]
oid | 2254
-[ RECORD 6 ]
oid | 2255

postgres=# select pg_get_function_arguments(2250);
-[ RECORD 1 ]-------------+-----------------
pg_get_function_arguments | name, text, text

postgres=# \x
Expanded display is off.
postgres=# select pg_get_function_arguments(2250);
 pg_get_function_arguments
---------------------------
 name, text, text
(1 row)

postgres=#
```

pg_get_function_identity_arguments

函数名称	pg_get_function_identity_arguments
参数数据类型	func_oid
函数返回值数据类型	text
函数含义	获得参数清单以标识出function或者procedure

使用举例	`postgres=# select pg_get_function_identity_arguments(2250);`
	`pg_get_function_identity_arguments`
	`------------------------------------`
	`name, text, text`
	`(1 row)`
	`postgres=#`

pg_get_function_result

函数名称	pg_get_function_result
参数数据类型	func_oid
函数返回值数据类型	text
函数含义	获得function，若是procedure，返回null

使用举例	`postgres=# select pg_get_function_result(89);`
	`pg_get_function_result`
	`------------------------`
	`text`
	`(1 row)`
	`postgres=#`

pg_get_indexdef

函数名称	pg_get_indexdef
参数数据类型	index_oid或者index_oid, column_no, pretty_bool
函数返回值数据类型	text
函数含义	获得索引的定义语句获得某个索引上第n个索引列的名称

使用举例	

```
postgres=# select oid from pg_class where relname='idx_2';
  oid
-------
 16427
(1 row)

postgres=# select pg_get_indexdef(16427);
                     pg_get_indexdef
-------------------------------------------------------
 CREATE INDEX idx_2 ON public.data1 USING btree (c1, c3)
(1 row)

postgres=#

postgres=# select pg_get_indexdef(16427,2,true);
 pg_get_indexdef
-----------------
 c3
(1 row)

postgres=# select pg_get_indexdef(16427,1,true);
 pg_get_indexdef
-----------------
 c1
(1 row)

postgres=#
```

pg_get_keywords

函数名称	pg_get_keywords
参数数据类型	无输入参数
函数返回值数据类型	setof record
函数含义	返回PostgreSQL数据库中的SQL关键字清单返回结果中有3列(以逗号分隔)，第一列是关键字名称，第二列catcode是category code U for unreserved C for column name T for type or function name R for reserved第三列catdesc是描述

使用举例	

```
postgres=# select pg_get_keywords();
pg_get_keywords
-----------------------------------------------------------
------
 (abort,U,unreserved)
 (absolute,U,unreserved)
 (access,U,unreserved)
 (action,U,unreserved)
 (add,U,unreserved)
 (admin,U,unreserved)
 (after,U,unreserved)
 (aggregate,U,unreserved)
 (all,R,reserved)
 (also,U,unreserved)
 (alter,U,unreserved)
 (always,U,unreserved)
 (analyse,R,reserved)
 (analyze,R,reserved)
 (and,R,reserved)
限于篇幅，本函数的输出仅摘录到此处
```

pg_get_ruledef

函数名称	pg_get_ruledef
参数数据类型	rule_oid或者rule_oid, pretty_bool
函数返回值数据类型	Text
函数含义	获得rule的create rule语句

使用举例	

```
postgres=# select oid ,rulename from pg_rewrite where
rulename='pg_settings_u';
  oid  |   rulename
-------+---------------
 12168 | pg_settings_u
(1 row)

postgres=# select pg_get_ruledef(12168);
                                          pg_get_ruledef
-------------------------------------------------------------
--------------------------------------
 CREATE RULE pg_settings_u AS
+
     ON UPDATE TO pg_catalog.pg_settings
+
    WHERE (new.name = old.name) DO  SELECT set_config(old
.name, new.setting, false) AS set_config;
(1 row)

postgres=#
```

pg_get_serial_sequence

函数名称	pg_get_serial_sequence
参数数据类型	table_name, column_name
函数返回值数据类型	text
函数含义	获得自增列或者标识列使用的sequence的name
使用举例	

```
postgres=# select pg_get_serial_sequence('gen_y','c1');
pg_get_serial_sequence
------------------------
public.gen_y_c1_seq
(1 row)
postgres=#
```

pg_get_statisticsobjdef

函数名称	pg_get_statisticsobjdef
参数数据类型	statobj_oid
函数返回值数据类型	text
函数含义	获得extended statistics object的CREATE STATISTICS语句。
使用举例	

pg_get_triggerdef

函数名称	pg_get_triggerdef
参数数据类型	trigger_oid 或者trigger_oid, pretty_bool
函数返回值数据类型	text
函数含义	获得trigger的create trigger语句

使用举例	`postgres=# select pg_get_triggerdef(16992);`
	`pg_get_triggerdef`
	`----------`
	`----------`
	`CREATE TRIGGER example_trigger AFTER INSERT ON public`
	`.company FOR EACH ROW EXECUTE FUNCTION auditlogfunc()`
	`(1 row)`
	`postgres=#`

pg_get_userbyid

函数名称	pg_get_userbyid
参数数据类型	role_oid
函数返回值数据类型	name
函数含义	获得特定role oid的role name

使用举例	`postgres=# select pg_get_userbyid(16426);`
	`pg_get_userbyid`
	`----------`
	`lc0029999`
	`(1 row)`
	`postgres=#`

pg_get_viewdef

函数名称	pg_get_viewdef
参数数据类型	view_name 或者 view_name, pretty_bool或者 view_oid或者 view_oid, pretty_bool或者view_oid, wrap_column_int
函数返回值数据类型	text
函数含义	获得view的定义

使用举例	`postgres=# select pg_get_viewdef('company_view');`
	`pg_get_viewdef`
	`----------`
	`SELECT company.id,+ company.name, + company.age + FROM`
	`company;`
	`(1 row)`
	`postgres=#`

pg_index_column_has_property

函数名称	pg_index_column_has_property
参数数据类型	index regclass, column integer, property text
函数返回值数据类型	boolean
函数含义	测试索引列是否有特定的索引列属性。索引列属性见下： asc 索引列升序 desc 索引列降序 nulls_first null值在前 nulls_last null值在后 orderable 列是否具有任何定义的排序顺序？ distance_orderable 列是否可以被"distance" operator 顺序扫描到, 比如 ORDER BY col <-> constant ? returnable index-only扫描是否可以返回列值 search_array 列是否原生支持 col = ANY(array)搜索 search_nulls 列是否支持IS NULL 和IS NOT NULL 搜索 注意：extension的访问方法可以为它们的indexes定义额外的属性名称。本函数在如下情况下会返回NULL：属性名称未知(not known)或者不适用于特定的对象Oid或者column number不能标识出一个有效的对象。

使用举例	`postgres=# select pg_index_column_has_property('idx_2'::regclass,1,'desc');`
	`pg_index_column_has_property`
	`----------`
	`f`
	`(1 row)`
	`postgres=# select pg_index_column_has_property('idx_2'::regclass,1,'asc');`
	`pg_index_column_has_property`
	`----------`
	`t`
	`(1 row)`
	`postgres=#`

pg_index_has_property

函数名称	pg_index_has_property
参数数据类型	index regclass, property text
函数返回值数据类型	boolean
函数含义	测试一个index是否具有特定的索引属性。如下是索引属性的清单 clusterable 在cluster命令中是否可以用到index index_scan index 是否支持 plain (non-bitmap) scans? bitmap_scan index 是否支持 bitmap scans? backward_scan Can the scan direction be changed in mid-scan (to support FETCH BACKWARD on a cursor without needing materialization)? 注意：extension的访问方法可以为它们的indexes定义额外的属性名称。本函数在如下情况下会返回NULL：属性名称未知(not known)或者不适用于特定的对象oid不能标识出一个有效的对象。

使用举例	`postgres=# select pg_index_has_property('idx_2'::regclass,'index_scan');`
	`pg_index_has_property`
	`----------`
	`t`
	`(1 row)`
	`postgres=#`

pg_indexam_has_property

函数名称	pg_indexam_has_property
参数数据类型	am oid, property text
函数返回值数据类型	boolean
函数含义	测试一个index访问方法是否具有特定的属性。如下是索引访问方法属性清单： can_order 访问方法是否支持 ASC, DESC 以及CREATE INDEX的related keywords can_unique 访问方法是否支持unique indexes? can_multi_col 访问方法是否支持多列indexes? can_exclude 访问方法是否支持exclusion constraints? can_include 访问方法是否支持CREATE INDEX的 INCLUDE 子句 本函数在如下情况下会返回NULL：属性名称未知(not known)或者不适用于特定的对象 oid不能标识出一个有效的对象。

使用举例	

```
postgres=# select * from pg_am;
 oid  | amname |      amhandler       | amtype
------+--------+----------------------+--------
    2 | heap   | heap_tableam_handler | t
  403 | btree  | bthandler            | i
  405 | hash   | hashhandler          | i
  783 | gist   | gisthandler          | i
 2742 | gin    | ginhandler           | i
 4000 | spgist | spghandler           | i
 3580 | brin   | brinhandler          | i
(7 rows)

postgres=#
postgres=# select pg_indexam_has_property(4000
,'can_multi_col');
 pg_indexam_has_property
-------------------------
 f
(1 row)

postgres=#
postgres=# select pg_indexam_has_property(403,'can_unique');
 pg_indexam_has_property
-------------------------
 t
(1 row)

postgres=#
```

pg_options_to_table

函数名称	pg_options_to_table
参数数据类型	options_array text[]
函数返回值数据类型	setof record (option_name text, option_value text)
函数含义	返回pg_class.reloptions或者pg_attribute.attoptions表示的存储选项

使用举例	

```
postgres=# select pg_options_to_table(reloptions) from
pg_class where oid='16384';
pg_options_to_table
---------------------------------------------
(autovacuum_vacuum_insert_threshold,10000)
(autovacuum_vacuum_insert_scale_factor,0.1)
(2 rows)
postgres=#
```

pg_tablespace_databases

函数名称	pg_tablespace_databases
参数数据类型	tablespace oid
函数返回值数据类型	setof oid
函数含义	返回特定tablespace中保存的object的database的oid，如果本函数返回任何值，那么就表示tablespace不为空并且不能被drop掉。若想查出在该表空间内有哪些对象，你需要连接到本函数返回的database中，并查询pg_class获得这些对象。
使用举例	

```
postgres=# select pg_tablespace_databases(1663);
pg_tablespace_databases
-------------------------
16424
1
16878
14174
14173
(5 rows)
postgres=#
```

pg_tablespace_location

函数名称	pg_tablespace_location
参数数据类型	tablespace oid
函数返回值数据类型	Text
函数含义	返回指定tablespace的文件系统路径
使用举例	

```
postgres=# select pg_tablespace_location(16391);
pg_tablespace_location
------------------------
/home/pg131/ts1
(1 row)
postgres=#
```

pg_typeof

函数名称	pg_typeof
参数数据类型	any
函数返回值数据类型	regtype
函数含义	指定值的数据类型的oid

使用举例	`postgres=# SELECT pg_typeof(33.339999);`
	`-[ RECORD 1 ]------`
	`pg_typeof | numeric`
	
	`postgres=# SELECT typlen FROM pg_type WHERE oid = pg_typeof (33.339999);`
	`-[ RECORD 1 ]`
	`typlen | -1`
	
	`postgres=#`

COLLATION FOR

函数名称	COLLATION FOR
参数数据类型	"any"
函数返回值数据类型	text
函数含义	返回指定值的collation的name，本函数的返回值是被双引号引起来的，如果入参表达式不能派生出collation，则返回null如果传入的参数不是可以collatable的data type，则会抛出错误

使用举例	`postgres=# select collation for(datname) from pg_database;`
	`pg_collation_for`
	`------------------`
	`"C"`
	`"C"`
	`"C"`
	`"C"`
	`"C"`
	`(5 rows)`
	`postgres=#`

to_regclass

函数名称	to_regclass
参数数据类型	text
函数返回值数据类型	Regclass
函数含义	将文本形式的relation name转换为oid，本函数的输入参数不可以是数字形式的oid。如果本函数未找到输入参数指定的name，本函数会返回NULL

使用举例	

```
postgres=# select to_regclass('public.data1');
-[ RECORD 1 ]------
to_regclass | data1

postgres=# \x
Expanded display is off.
postgres=# select oid, relname from pg_class where oid
=to_regclass('public.data1');
  oid  | relname
-------+---------
 16384 | data1
(1 row)

postgres=#
```

to_regcollation

函数名称	to_regcollation
参数数据类型	Text
函数返回值数据类型	regcollation
函数含义	将文本形式的collation name转换为oid，本函数的输入参数不可以是数字形式的oid。如果本函数未找到输入参数指定的name，本函数会返回NULL

使用举例	

```
postgres=# select to_regcollation('"POSIX"');
 to_regcollation
-----------------
 "POSIX"
(1 行记录)

postgres=# select * from pg_collation where oid
=to_regcollation('"POSIX"');
 oid | collname | collnamespace | collowner | collprovider |
collisdeterministic | collencoding | collcollate | collctype
| collversion
-----+----------+---------------+-----------+--------------+
---------------------+--------------+-------------+
-----------+-------------
 951 | POSIX    |            11 |        10 | c            |
t                   |           -1 | POSIX       | POSIX
|
(1 行记录)

postgres=# select oid,collname from pg_collation where oid
=to_regcollation('"POSIX"');
 oid | collname
-----+----------
 951 | POSIX
(1 行记录)

postgres=#
```

to_regnamespace

函数名称	to_regnamespace
参数数据类型	text
函数返回值数据类型	regnamespace
函数含义	将文本形式的schema name转换为oid，本函数的输入参数不可以是数字形式的oid。如果本函数未找到输入参数指定的name，本函数会返回NULL
使用举例	

```
postgres=# select to_regnamespace('public');
to_regnamespace
-----------------
public
(1 行记录)
postgres=#
```

to_regoper

函数名称	to_regoper
参数数据类型	text
函数返回值数据类型	regoper
函数含义	将文本形式的operator name转换为oid，本函数的输入参数不可以是数字形式的oid。如果本函数未找到输入参数指定的name或者含义不明确，本函数会返回NULL

使用举例	`postgres=# select oid, oprname  from pg_operator where oid =to_regoper('!');`
	`  oid | oprname`
	`-----+---------`
	`  388 | !`
	`(1 row)`
	
	`postgres=# select  to_regoper('!');`
	`  to_regoper`
	`------------`
	`  !`
	`(1 row)`
	
	`postgres=#`

to_regoperator

函数名称	to_regoperator
参数数据类型	text
函数返回值数据类型	regoperator
函数含义	将文本形式的operator name(带参数类型)转换为oid，本函数的输入参数不可以是数字形式的oid。如果本函数未找到输入参数指定的name，本函数会返回NULL.

使用举例	

```
postgres=# select oid, oprname from pg_operator where oid
=to_regoperator('+(integer,integer)');
 oid | oprname
-----+---------
 551 | +
(1 row)

postgres=#  select  to_regoperator('+(int4,int4)');
   to_regoperator
--------------------
 +(integer,integer)
(1 row)

postgres=#
```

to_regproc

函数名称	to_regproc
参数数据类型	text
函数返回值数据类型	regproc
函数含义	将文本的function或者procedure转换为其oid。如果本函数未找到输入参数指定的name或者含义不明确，本函数会返回NULL。本函数的输入参数不可以是数字形式的oid。
使用举例	

```
postgres=# select to_regproc('pg_stat_statements');
     to_regproc
--------------------
 pg_stat_statements
(1 row)

postgres=# select oid,proname from pg_proc where oid
=to_regproc('pg_stat_statements');
  oid  |      proname
-------+--------------------
 16401 | pg_stat_statements
(1 row)

postgres=#
```

to_regprocedure

函数名称	to_regprocedure
参数数据类型	text

函数返回值数据类型	regprocedure
函数含义	将文本的function或者procedure(带参数)转换为其oid。如果本函数未找到输入参数指定的name，本函数会返回NULL。本函数的输入参数不可以是数字形式的oid。
使用举例	`postgres=# select to_regprocedure('pg_get_viewdef(text)');` `  to_regprocedure` `----------` ` pg_get_viewdef(text)` `(1 row)` `postgres=# select oid, proname from pg_proc where oid=` `to_regprocedure('pg_get_viewdef(text)');` ` oid	proname` `------+----------` ` 1640	pg_get_viewdef` `(1 row)`  `postgres=#`

to_regrole

函数名称	to_regrole
参数数据类型	text
函数返回值数据类型	regrole
函数含义	将文本的role name转换为其oid。如果本函数未找到输入参数指定的name，本函数会返回NULL。本函数的输入参数不可以是数字形式的oid。
使用举例	`postgres=# select oid,rolname from pg_roles where oid` `=to_regrole('lc0039999');` ` oid	rolname` `-------+-----------` ` 16877	lc0039999` `(1 row)`  `postgres=# select to_regrole('lc0039999');` ` to_regrole` `------------` ` lc0039999` `(1 row)`  `postgres=#`

to_regtype

函数名称	to_regtype
参数数据类型	text
函数返回值数据类型	regtype
函数含义	将文本的type name转换为其oid。如果本函数未找到输入参数指定的name，本函数会返回NULL。本函数的输入参数不可以是数字形式的oid。

使用举例	`postgres=# select to_regtype('int4');`
	`to_regtype`
	`------------`
	`integer`
	`(1 row)`
	`postgres=# select oid, typname from pg_type where oid =to_regtype('int4');`
	`oid | typname`
	`-----+---------`
	`23 | int4`
	`(1 row)`
	`postgres=#`

pg_describe_object

函数名称	pg_describe_object
参数数据类型	classid oid, objid oid, objsubid integer
函数返回值数据类型	text
函数含义	用于描述pg_depend系统表中保存的依赖对象。

使用举例	

```
postgres=# select oid,relname from pg_class where oid in
(select distinct classid from pg_depend) order by oid;
 oid  |         relname
------+-------------------------
 1247 | pg_type
 1255 | pg_proc
 1259 | pg_class
 2328 | pg_foreign_data_wrapper
 2602 | pg_amop
 2603 | pg_amproc
 2606 | pg_constraint
 2612 | pg_language
 2616 | pg_opclass
 2617 | pg_operator
 2618 | pg_rewrite
 2620 | pg_trigger
 2753 | pg_opfamily
 3079 | pg_extension
 3600 | pg_ts_dict
 3602 | pg_ts_config
 3764 | pg_ts_template
(17 rows)

postgres=# select * from pg_foreign_data_wrapper ;
  oid  | fdwname  | fdwowner | fdwhandler | fdwvalidator |
fdwacl | fdwoptions
-------+----------+----------+------------+--------------+
--------+------------
 16954 | file_fdw |       10 |      16952 |        16953 |
|
(1 row)

postgres=# select pg_describe_object(2328,16954,0);
      pg_describe_object
-------------------------------
 foreign-data wrapper file_fdw
(1 row)

postgres=#
```

pg_identify_object

函数名称	pg_identify_object

参数数据类型	classid oid, objid oid, objsubid integer
函数返回值数据类型	record (type text, schema text, name text, identity text)
函数含义	返回一个数据库对象的详细信息

使用举例	`postgres=# select pg_identify_object(2328,16954,0);`
	`pg_identify_object`
	`----------`
	`("foreign-data wrapper",,file_fdw,file_fdw)`
	`(1 row)`
	`postgres=#`

pg_identify_object_as_address

函数名称	pg_identify_object_as_address
参数数据类型	classid oid, objid oid, objsubid integer
函数返回值数据类型	record (type text, object_names text[], object_args text[])
函数含义	返回一个数据库对象的足够信息以标识出数据库对象

使用举例	`postgres=# select pg_identify_object_as_address(2328,16954,`
	`0);`
	`pg_identify_object_as_address`
	`----------`
	`("foreign-data wrapper",{file_fdw},{})`
	`(1 row)`
	`postgres=#`

pg_get_object_address

函数名称	pg_get_object_address
参数数据类型	type text, object_names text[], object_args text[]
函数返回值数据类型	record (classid oid, objid oid, objsubid integer)
函数含义	本函数是pg_identify_object_as_address的反函数

使用举例	`postgres=# select pg_get_object_address('foreign-data`
	`wrapper','{file_fdw}','{}');`
	`pg_get_object_address`
	`----------`
	`(2328,16954,0)`
	`(1 row)`
	`postgres=#`

col_description

函数名称	col_description
参数数据类型	table oid, column integer
函数返回值数据类型	text
函数含义	获取某个表的某个列的comment
使用举例	

```
postgres=# create table t_comment(c1 int );
CREATE TABLE
postgres=# comment on column t_comment.c1 is 'this is c1
comment!';
COMMENT
postgres=# select oid from pg_class where oid=to_regclass
('t_comment')
postgres-# ;
  oid
-------
 17001
(1 row)
postgres=# select col_description(17001,1);
   col_description
---------------------
 this is c1 comment!
(1 row)
postgres=#
postgres=# select col_description(to_regclass('t_comment'),
1);
   col_description
---------------------
 this is c1 comment!
(1 row)

postgres=#
```

obj_description

函数名称	obj_description
参数数据类型	object oid, catalog name 或者object oid
函数返回值数据类型	text
函数含义	返回特定数据库对象的注释信息

使用举例	

```
postgres=# comment on table t_comment is  ' table-level test
comment';
COMMENT
postgres=# \dt+ data1
                        List of relations
 Schema | Name  | Type  | Owner | Persistence | Size  |
Description
--------+-------+-------+-------+-------------+-------+
-------------
 public | data1 | table | pg131 | permanent   | 42 MB |
(1 row)

postgres=# \dt+ t_comment;
                                  List of relations
 Schema |   Name    | Type  | Owner | Persistence |  Size
|        Description
--------+-----------+-------+-------+-------------+
---------+---------------------------
 public | t_comment | table | pg131 | permanent   | 0 bytes
|  table-level test comment
(1 row)

postgres=# select obj_description(17001);
      obj_description
---------------------------
  table-level test comment
(1 row)

postgres=#
```

shobj_description

函数名称	shobj_description
参数数据类型	object oid, catalog name
函数返回值数据类型	text
函数含义	返回特定共享数据库对象(如databases, roles, and tablespaces)的注释信息.
使用举例	

```
postgres=# select shobj_description(14174,'pg_database');
shobj_description
-------------------------------------------
default administrative connection database
(1 row)
postgres=#
```

pg_current_xact_id

函数名称	pg_current_xact_id
参数数据类型	无输入参数
函数返回值数据类型	xid8
函数含义	返回当前的事务id，若是没有当前事务id，系统会自动指派一个新的事务id

使用举例	`postgres=# select pg_current_xact_id();`
	`pg_current_xact_id`
	`--------------------`
	`720`
	`(1 row)`
	`postgres=#`

pg_current_xact_id_if_assigned()

函数名称	pg_current_xact_id_if_assigned()
参数数据类型	无输入参数
函数返回值数据类型	xid8
函数含义	返回当前的current transaction’s ID，若是没有transaction’s ID被指派，本函数返回NULL。如果事务可能是只读的，最好使用本函数，以避免不必要的XID消耗。

使用举例	`postgres=*# select pg_current_xact_id_if_assigned();`
	`pg_current_xact_id_if_assigned`
	`--------------------------------`
	`(1 row)`
	`postgres=*# update data1 set c2='data1_1' where c1=1;`
	`UPDATE 1`
	`postgres=*# select pg_current_xact_id_if_assigned();`
	`pg_current_xact_id_if_assigned`
	`--------------------------------`
	`722`
	`(1 row)`
	`postgres=*#`

pg_xact_status

函数名称	pg_xact_status
参数数据类型	xid8
函数返回值数据类型	text

函数含义	返回近期事务的提交状态。本函数的执行结果是下面三个中的一个:in progress、commited、aborted。当事务足够老而没有在PostgreSQL中保存下来时，commit status会被丢弃，此时本函数返回结果为NULL。请注意，针对prepared transactions，本函数的返回结果为in progress,应用程序必须检查pg_prepared_xacts以确定一个transaction ID是否属于prepared transaction

使用举例	`postgres=# select pg_xact_status(pg_current_xact_id());`
	`pg_xact_status`
	`----------`
	`in progress`
	`(1 row)`
	`postgres=# postgres=# select pg_xact_status(722::text::xid8);`
	`pg_xact_status`
	`----------`
	`committed`
	`(1 row)`
	`postgres=#`

pg_current_snapshot

函数名称	pg_current_snapshot
参数数据类型	无输入参数
函数返回值数据类型	pg_snapshot
函数含义	返回当前的snapshot,这是一种显示哪个transaction id是正处于in-progress状态的数据结构。注意：有关pg_snapshot数据类型：pg_snapshot数据类型用于存储在一个特定时间点上的transaction ID visibility信息。 pg_snapshot数据类型的表示方法是xmin:xmax:xip_list这个xmin:xmax:xip_list就是pg_snapshot数据类型组成。见下： xmin:处于active状态的最小的transaction id，所有小于xmin的transaction id要么是committed (visible),要么是rolled back(dead) xmax:大于或者等于本xmax值的transaction id是在snapshot结束时未完成的，因此，xmax是不可见的。 xip_list:在生成snapshot时，处于in progress状态的transactions。在xmin ⇐ X < xmax范围内并且不在本xip_list中的transaction ID表示在snapshot生成时是已经完成的transaction ID,因此，根据commit status，要么是visible的要么是dead，本列表中不包括subtransactions的transaction IDs

使用举例	`postgres=# select pg_current_snapshot();`
	`pg_current_snapshot`
	`----------`
	`727:727:`
	`(1 row)`
	`postgres=#`

pg_snapshot_xip

函数名称	pg_snapshot_xip
参数数据类型	pg_snapshot
函数返回值数据类型	setof xid8
函数含义	返回snapshot中处于in-progress状态的transaction id

使用举例	`postgres=# select pg_snapshot_xip('10:20:10,14,15');`
	`pg_snapshot_xip`
	`-----------------`
	`10`
	`14`
	`15`
	`(3 rows)`
	`postgres=#`

pg_snapshot_xmax

函数名称	pg_snapshot_xmax
参数数据类型	pg_snapshot
函数返回值数据类型	xid8
函数含义	返回snapshot的xmax

使用举例	`postgres=# select pg_snapshot_xmax('10:20:10,14,15');`
	`pg_snapshot_xmax`
	`------------------`
	`20`
	`(1 row)`
	`postgres=#`

pg_snapshot_xmin

函数名称	pg_snapshot_xmin
参数数据类型	pg_snapshot
函数返回值数据类型	xid8
函数含义	返回snapshot的xmin

使用举例	`postgres=# select pg_snapshot_xmin('10:20:10,14,15');`
	`pg_snapshot_xmin`
	`------------------`
	`10`
	`(1 row)`
	`postgres=#`

pg_visible_in_snapshot

函数名称	pg_visible_in_snapshot
参数数据类型	xid8, pg_snapshot
函数返回值数据类型	bool
函数含义	给定的事务ID是否对snapshot可见，即：给定的事务ID是否在生成snapshot之前完成。本函数对subtransaction ID不能给出正确的结果。

使用举例	`postgres=# select pg_visible_in_snapshot(722::text::xid8 ,'10:20:10,14,15');`
	`pg_visible_in_snapshot`
	`------------------------`
	`f`
	`(1 row)`
	`postgres=# select pg_visible_in_snapshot(12::text::xid8 ,'10:20:10,14,15');`
	`pg_visible_in_snapshot`
	`------------------------`
	`t`
	`(1 row)`
	`postgres=#`

txid_current

函数名称	txid_current
参数数据类型	无输入参数
函数返回值数据类型	bigint
函数含义	返回当前的事务id，若是没有当前事务id，系统会自动指派一个新的事务id。本函数（返回值是bigint数据类型）的作用与pg_current_xact_id(返回值是xid8数据类型)的作用是类似的，只是两个函数的返回值数据类型不同。

使用举例	`postgres=# select txid_current();`
	`txid_current`
	`--------------`
	`735`
	`(1 row)`
	`postgres=#`

txid_current_if_assigned()

函数名称	txid_current_if_assigned()
参数数据类型	无输入参数
函数返回值数据类型	bigint
函数含义	本函数（注意：返回值是bigint数据类型）的作用与pg_current_xact_id_if_assigned(注意：返回值是xid8数据类型)的作用是类似的，只是两个函数的返回值数据类型不同。

使用举例	

```
[pg131@VM-0-8-centos ~]$ psql -d postgres psql (13.1)
Type "help" for help.
postgres=# select txid_current_if_assigned();
txid_current_if_assigned
--------------------------
(1 row)
postgres=# begin;
BEGIN
postgres=*# select txid_current_if_assigned();
txid_current_if_assigned
--------------------------
(1 row)
postgres=*# select txid_current();
txid_current
--------------
738
(1 row)
postgres=*# select txid_current_if_assigned();
txid_current_if_assigned
--------------------------
738
(1 row)
```

txid_current_snapshot()

函数名称	txid_current_snapshot()
参数数据类型	无输入参数
函数返回值数据类型	txid_snapshot
函数含义	返回当前的snapshot,这是一种显示哪个transaction id是正处于in-progress状态的数据结构。本函数（注意：返回值是txid_snapshot数据类型）的作用与pg_current_snapshot (注意：返回值是pg_snapshot数据类型)的作用是类似的，只是两个函数的返回值数据类型不同。

使用举例	

```
postgres=# select txid_current_snapshot();
txid_current_snapshot
-----------------------
741:741:
(1 row)
 postgres=# \gdesc
 Column | Type
 -----------------------+---------------
 txid_current_snapshot | txid_snapshot
 (1 row)
postgres=#
```

txid_snapshot_xip(txid_snapshot)

函数名称	txid_snapshot_xip(txid_snapshot)
参数数据类型	txid_snapshot
函数返回值数据类型	setof bigint
函数含义	返回snapshot中处于in-progress状态的transaction id。本函数（注意：返回值是txid_snapshot数据类型）的作用与pg_snapshot_xip (注意：返回值是pg_snapshot数据类型)的作用是类似的，只是两个函数的返回值数据类型不同。
使用举例	

```
postgres=# select txid_snapshot_xip('10:20:10,14,15');
txid_snapshot_xip
-------------------
10
14
15
(3 rows)
 postgres=# \gdesc
 Column | Type
 -------------------+--------
 txid_snapshot_xip | bigint
 (1 row)
```

txid_snapshot_xmax

函数名称	txid_snapshot_xmax
参数数据类型	txid_snapshot
函数返回值数据类型	bigint
函数含义	返回snapshot的xmax。本函数（注意：返回值是bigint数据类型）的作用与pg_snapshot_xmax (注意：返回值是pg_snapshot数据类型)的作用是类似的，只是两个函数的返回值数据类型不同。

使用举例	

```
postgres=# select txid_snapshot_xmax('10:20:10,14,15');
txid_snapshot_xmax
--------------------
20
(1 row)
 postgres=# \gdesc
 Column | Type
 --------------------+--------
 txid_snapshot_xmax | bigint
 (1 row)
postgres=#
```

txid_snapshot_xmin

函数名称	txid_snapshot_xmin
参数数据类型	txid_snapshot
函数返回值数据类型	bigint
函数含义	返回snapshot的xmin。本函数（注意：返回值是bigint数据类型）的作用与pg_snapshot_xmin(注意：返回值是pg_snapshot数据类型)的作用是类似的，只是两个函数的返回值数据类型不同。
使用举例	

```
postgres=# select txid_snapshot_xmin('10:20:10,14,15');
txid_snapshot_xmin
--------------------
10
(1 row)
 postgres=# \gdesc
 Column | Type
 --------------------+--------
 txid_snapshot_xmin | bigint
 (1 row)
postgres=#
```

txid_visible_in_snapshot

函数名称	txid_visible_in_snapshot
参数数据类型	bigint, txid_snapshot
函数返回值数据类型	boolean
函数含义	给定的事务ID是否对snapshot可见，即：给定的事务ID是否在生成snapshot之前完成。本函数对subtransaction ID不能给出正确的结果。

使用举例	``` postgres=# select txid_visible_in_snapshot(722 ,'10:20:10,14,15'); txid_visible_in_snapshot --------------------------- f (1 row) postgres=# select txid_visible_in_snapshot(12 ,'10:20:10,14,15'); txid_visible_in_snapshot --------------------------- t (1 row) postgres=# ```

txid_status

函数名称	txid_status
参数数据类型	bigint
函数返回值数据类型	text
函数含义	返回近期事务的提交状态。本函数的执行结果是下面三个中的一个:in progress、commited、aborted。当事务足够老而没有在PostgreSQL中保存下来时，commit status会被丢弃，此时本函数返回结果为NULL。请注意，针对prepared transactions，本函数的返回结果为in progress,应用程序必须检查pg_prepared_xacts以确定一个transaction ID是否属于prepared transaction.

使用举例	```
	postgres=# select txid_status(txid_current());
	txid_status
	------------- 	
	in progress
	(1 row)
	postgres=# select txid_current();
	txid_current
	-------------- 	
	742
	(1 row)
	postgres=# select txid_status(722);
	txid_status
	------------- 	
	committed
	(1 row)
	postgres=#
	``

pg_xact_commit_timestamp

函数名称	pg_xact_commit_timestamp
参数数据类型	xid
函数返回值数据类型	timestamp with time zone

函数含义	返回一个事务的提交时间。本函数能正常使用的前提就是配置参数track_commit_timestamp设置为on，否会有如下提示：
	`postgres=# select pg_xact_commit_timestamp(722::text::xid);`
	`ERROR: could not get commit timestamp data`
	`HINT: Make sure the configuration parameter`
	`"track_commit_timestamp" is set.`
	`postgres=#`

使用举例	`postgres=# select txid_current();`
	`txid_current`
	`--------------`
	`743`
	`(1 row)`
	`postgres=# select pg_xact_commit_timestamp(743::text::xid);`
	`pg_xact_commit_timestamp`
	`-------------------------------`
	`2021-04-29 16:52:37.19133+08`
	`(1 row)`
	`postgres=#`

pg_last_committed_xact

函数名称	pg_last_committed_xact
参数数据类型	无输入数据类型
函数返回值数据类型	record (xid xid, timestamp timestamp with time zone)
函数含义	返回最后一个提交事务的transaction id和commit timestamp

使用举例	`postgres=# select pg_last_committed_xact();`
	`pg_last_committed_xact`
	`--------------------------------------`
	`(743,"2021-04-29 16:52:37.19133+08")`
	`(1 row)`
	`postgres=#`

pg_control_checkpoint

函数名称	pg_control_checkpoint
参数数据类型	无输入数据类型

函数返回值数据类型	record
函数含义	本函数是从PostgreSQL控制文件中获取数据。本函数返回当前checkpoint状态的相关信息
使用举例	

pg_control_system

函数名称	pg_control_system
参数数据类型	无输入数据类型
函数返回值数据类型	record
函数含义	本函数是从PostgreSQL控制文件中获取数据。本函数返回当前控制文件状态的相关信息。
使用举例	

```
postgres=# select pg_control_system();
pg_control_system
-----------------------------------------------------------
---
(1300,202007201,6908228805274066467,"2021-04-29
16:57:56+08")
(1 row)
postgres=#
```

pg_control_init

函数名称	pg_control_init
参数数据类型	无输入数据类型
函数返回值数据类型	record
函数含义	本函数是从PostgreSQL控制文件中获取数据。本函数返回cluster初始化时的相关信息
使用举例	

```
postgres=# select pg_control_init();
pg_control_init
--------------------------------------------------
(8,8192,131072,8192,16777216,64,32,1996,2048,t,0)
(1 row)
postgres=#
```

pg_control_recovery

函数名称	pg_control_recovery
参数数据类型	无输入数据类型
函数返回值数据类型	record
函数含义	本函数是从PostgreSQL控制文件中获取数据。本函数返回恢复状态的信息。

使用举例	

```
postgres=# select pg_control_recovery();
pg_control_recovery
---------------------
(0/0,0,0/0,0/0,f)
(1 row)
postgres=#
```

gen_random_uuid

函数名称	gen_random_uuid
参数数据类型	无输入参数
函数返回值数据类型	uuid
函数含义	本函数返回version 4版本的uuid
使用举例	

```
postgres=# select gen_random_uuid();
gen_random_uuid
--------------------------------------
39158a63-f6d0-4883-8765-8b8e9c9cae62
(1 row)
postgres=#
```

array_to_tsvector

函数名称	array_to_tsvector
参数数据类型	text[]
函数返回值数据类型	tsvector
函数含义	将array of lexemes转换为tsvector
使用举例	

```
postgres=# select array_to_tsvector('{fat,cat,rat}'::
text[]);
array_to_tsvector
-------------------
'cat' 'fat' 'rat'
(1 row)
postgres=#
```

get_current_ts_config

函数名称	get_current_ts_config
参数数据类型	无输入参数
函数返回值数据类型	regconfig
函数含义	返回当前默认的text search配置的oid（参见配置参数default_text_search_config）

使用举例	`postgres=# select get_current_ts_config();`
	`get_current_ts_config`
	`------------------------`
	`english`
	`(1 row)`
	`postgres=#`

length

函数名称	length
参数数据类型	tsvector
函数返回值数据类型	integer
函数含义	返回lexemes的数量

使用举例	`postgres=# select length('fat:2,4 cat:3 rat:5A'::tsvector);`
	`length`
	`--------`
	`3`
	`(1 row)`
	`postgres=#`

numnode

函数名称	numnode
参数数据类型	tsquery
函数返回值数据类型	integer
函数含义	返回lexemes加operator的数量

使用举例	`postgres=# select numnode('(fat & rat) | cat'::tsquery) ;`
	`numnode`
	`---------`
	`5`
	`(1 row)`
	`postgres=#`

plainto_tsquery

函数名称	plainto_tsquery
参数数据类型	[config regconfig,] query text
函数返回值数据类型	tsquery
函数含义	将text转换为tsquery。其中，字符串中的任何标点都将被忽略

使用举例	

```
postgres=# select plainto_tsquery('english', 'The Fat
Rats');
plainto_tsquery
-----------------
'fat' & 'rat'
(1 row)
postgres=#
```

phraseto_tsquery

函数名称	phraseto_tsquery
参数数据类型	[config regconfig,] query text
函数返回值数据类型	tsquery
函数含义	将text转换为tsquery。其中，字符串中的任何标点都将被忽略
使用举例	

```
postgres=# select phraseto_tsquery('english', 'The Fat
Rats') ;
phraseto_tsquery
------------------
'fat' <-> 'rat'
(1 row)
postgres=#
```

websearch_to_tsquery

函数名称	websearch_to_tsquery
参数数据类型	[config regconfig,] query text
函数返回值数据类型	tsquery
函数含义	将text转换为tsquery。引用的单词序列被转换成短语测试。"or"一词被理解为产生or运算符，破折号产生NOT运算符，忽略其他标点符号
使用举例	

```
postgres=# select websearch_to_tsquery('english', '"fat rat"
or cat dog');
websearch_to_tsquery
----------------------------------
'fat' <-> 'rat' | 'cat' & 'dog'
(1 row)
postgres=#
```

querytree

函数名称	querytree
参数数据类型	tsquery
函数返回值数据类型	text

函数含义	返回一个tsquery中的可索引部分。当返回结果为空或者T时，表示这是一个不可索引的query。

使用举例	`postgres=# select querytree('foo & ! bar'::tsquery);`
	`querytree`
	`-----------`
	`'foo'`
	`(1 row)`
	`postgres=#`

setweight

函数名称	setweight
参数数据类型	vector tsvector, weight "char"或者vector tsvector, weight "char", lexemes text[]
函数返回值数据类型	tsvector
函数含义	为vector中的每个元素指定权重或者为在lexemes之内的vector中的每个元素指定权重

使用举例	`postgres=# select setweight('fat:2,4 cat:3 rat:5B'::`
	`tsvector, 'A');`
	`setweight`
	`-------------------------------`
	`'cat':3A 'fat':2A,4A 'rat':5A`
	`(1 row)`
	` postgres=#`
	` postgres=# select setweight('fat:2,4 cat:3 rat:5,6B'`
	`::tsvector, 'A', '{cat,rat}');`
	` setweight`
	` -------------------------------`
	` 'cat':3A 'fat':2,4 'rat':5A,6A`
	` (1 row)`
	`postgres=#`

strip

函数名称	strip
参数数据类型	tsvector
函数返回值数据类型	tsvector
函数含义	从tsvector中抹掉position和weight

使用举例	`postgres=# select strip('fat:2,4 cat:3 rat:5A'::tsvector);`
	`strip`
	`-------------------`
	`'cat' 'fat' 'rat'`
	`(1 row)`
	`postgres=#`

to_tsquery

函数名称	to_tsquery
参数数据类型	regconfig, text 或者text
函数返回值数据类型	tsquery
函数含义	将text转换为tsquery，本函数的返回结果必须由有效的tsquery operator进行连接。

使用举例	`postgres=# select to_tsquery('english', 'The & Fat & Rats');`
	`to_tsquery`
	`---------------`
	`'fat' & 'rat'`
	`(1 row)`
	` postgres=# select to_tsquery( 'The & Fat & Rats');`
	` to_tsquery`
	` ---------------`
	` 'fat' & 'rat'`
	` (1 row)`
	`postgres=#`

to_tsvector

函数名称	to_tsvector
参数数据类型	regconfig,text
函数返回值数据类型	tsvector
函数含义	将text转换为tsvector，结果中包括位置信息

使用举例	`postgres=# select to_tsvector('english', 'The Fat Rats');`
	`to_tsvector`
	`-----------------`
	`'fat':2 'rat':3`
	`(1 row)`
	`postgres=#`

to_tsvector

函数名称	to_tsvector
参数数据类型	regconfig,json或者regconfig,jsonb
函数返回值数据类型	tsvector
函数含义	将json文档中的每个字符串值转换为tsvector，结果值是按照文档顺序进行连接后输出的，当输入参数数据类型为jsonb时，json对象的字段的文档顺序是与实现相关的

使用举例	`postgres=# select to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json);`
	`to_tsvector ------------------------`
	`'dog':5 'fat':2 'rat':3`
	`(1 row)`
	` postgres=# postgres=# select to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb);`
	` to_tsvector`
	` ------------------------`
	` 'dog':1 'fat':4 'rat':5`
	` (1 row)`
	`postgres=#`

json_to_tsvector

函数名称	json_to_tsvector
参数数据类型	[config regconfig,] json, jsonb
函数返回值数据类型	tsvector
函数含义	将json文档中满足filter条件的每个item转换为一个tsvector。结果值是按照文档顺序进行连接后输出的，当输入参数数据类型为jsonb时，json对象的字段的文档顺序是与实现相关的。Filter条件必须是一个包括另个或者多个关键字的json 数组：string代表包括所有字符串值numeric代表包括所有数字值boolean代表包括所有boolean值key代表包括所有keyall代表包括上述所有。作为一个简单的例子，filter条件不可以是一个简单的并且是上述关键字之一的json值。

使用举例	`postgres=# select json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]');`
	`json_to_tsvector`
	`------------------------`
	`'123':5 'fat':2 'rat':3`
	`(1 row)`
	`postgres=#`

jsonb_to_tsvector

函数名称	jsonb_to_tsvector
参数数据类型	[config regconfig,] jsonb,jsonb

函数返回值数据类型	tsvector
函数含义	将json文档中满足filter条件的每个item转换为一个tsvector。结果值是按照文档顺序进行连接后输出的，当输入参数数据类型为jsonb时，json对象的字段的文档顺序是与实现相关的。Filter条件必须是一个包括另个或者多个关键字的json 数组：string代表包括所有字符串值numeric代表包括所有数字值boolean代表包括所有boolean值key代表包括所有keyall代表包括上述所有。作为一个简单的例子，filter条件不可以是一个简单的并且是上述关键字之一的json值。

使用举例	`postgres=# select jsonb_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::jsonb, '["string", "numeric"]');`
	`jsonb_to_tsvector`
	`----------`
	`'123':5 'fat':2 'rat':3`
	`(1 row)`
	`postgres=#`

ts_delete

函数名称	ts_delete
参数数据类型	tsvector,text
函数返回值数据类型	tsvector
函数含义	从vector中删除掉指定的lexeme

使用举例	`postgres=# select ts_delete('fat:2,4 cat:3 rat:5A':: tsvector, 'fat');`
	`ts_delete`
	`----------`
	`'cat':3 'rat':5A`
	`(1 row)`
	`postgres=#`

ts_delete

函数名称	ts_delete
参数数据类型	tsvector,text[]
函数返回值数据类型	tsvector
函数含义	从vector中删除掉指定的lexeme

使用举例	`postgres=# select ts_delete('fat:2,4 cat:3 rat:5A':: tsvector, ARRAY['fat','rat']);`
	`ts_delete ----------`
	`'cat':3`
	`(1 row)`
	`postgres=#`

ts_filter

函数名称	ts_filter
参数数据类型	tsvector, "char"[]
函数返回值数据类型	tsvector
函数含义	从vector中筛选出带有指定weights的元素

使用举例	`postgres=# select ts_filter('fat:2,4 cat:3b,7c rat:5A' ::tsvector, '{a,b}');`
	`ts_filter`
	`-------------------`
	`'cat':3B 'rat':5A`
	`(1 row)`
	`postgres=#`

ts_headline

函数名称	ts_headline
参数数据类型	[config regconfig,] document text, query tsquery [, options text]
函数返回值数据类型	text
函数含义	以简写的形式显示在document匹配到的query

使用举例	`postgres=# select ts_headline('The fat cat ate the rat.', 'cat');`
	`ts_headline`
	`----------------------------------`
	`The fat <b>cat</b> ate the rat.`
	`(1 row)`
	`postgres=#`

ts_headline

函数名称	ts_headline
参数数据类型	[config regconfig,] document json, query tsquery [, options text]
函数返回值数据类型	json
函数含义	以缩写的形式显示在json document匹配到的query

使用举例	`postgres=# select ts_headline('{"cat":"raining cats and dogs"}'::json, 'cat');`
	`ts_headline`
	`----------------------------------------`
	`{"cat":"raining <b>cats</b> and dogs"}`
	`(1 row)`

ts_headline

函数名称	ts_headline
参数数据类型	[config regconfig,] document jsonb, query tsquery [, options text]
函数返回值数据类型	jsonb
函数含义	以缩写的形式显示在json document匹配到的query

使用举例	`postgres=# select ts_headline('{"cat":"raining cats and dogs"}'::jsonb, 'cat');`
	`ts_headline`
	`----------`
	`{"cat": "raining <b>cats</b> and dogs"}`
	`(1 row)`
	`postgres=#`

ts_rank

函数名称	ts_rank
参数数据类型	[weights real[],] vector tsvector, query tsquery [, normalization integer]
函数返回值数据类型	real
函数含义	计算query与vector的匹配程度

使用举例	`postgres=# SELECT ts_rank(to_tsvector('The quick brown fox jumps over the lazy dog.'),to_tsquery('fox & dog')); ts_rank`
	`----------`
	`0.09148999`
	`(1 row)`
	`postgres=#`

ts_rank_cd

函数名称	ts_rank_cd
参数数据类型	[weights real[],] vector tsvector, query tsquery [, normalization integer]
函数返回值数据类型	real
函数含义	使用覆盖密度算法来计算query与vector的匹配程度

使用举例	`postgres=# SELECT ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', to_tsvector('PostgreSQL full text search is a wonderful method'),to_tsquery('wonderful'));`
	`ts_rank_cd`
	`----------`
	`0.1`
	`(1 row)`
	`postgres=#`

ts_rewrite

函数名称	ts_rewrite
参数数据类型	query tsquery, target tsquery, substitute tsquery
函数返回值数据类型	tsquery
函数含义	用substitute替换掉query中的target

使用举例	`postgres=# SELECT ts_rewrite('c & b'::tsquery, 'b'::tsquery, 'bad|bat'::tsquery);`
	`ts_rewrite`
	`------------------------`
	`( 'bad' | 'bat' ) & 'c'`
	`(1 row)`
	`postgres=#`

ts_rewrite

函数名称	ts_rewrite
参数数据类型	query tsquery, select text
函数返回值数据类型	tsquery
函数含义	

使用举例	`postgres=# CREATE TABLE test (col1 tsquery PRIMARY KEY, COL2 tsquery);`
	`CREATE TABLE`
	`postgres=# INSERT INTO test values ('p','q');`
	`INSERT 0 1`
	`postgres=# SELECT ts_rewrite('r & q'::tsquery, 'SELECT col1,col2 FROM test');`
	`ts_rewrite`
	`------------`
	`'r' & 'q'`
	`(1 row)`
	`postgres=#`

tsquery_phrase

函数名称	tsquery_phrase
参数数据类型	query1 tsquery, query2 tsquery
函数返回值数据类型	tsquery
函数含义	构造一个phrase query用于在连续的词位匹配query1和query2(同<→操作符)

使用举例	`postgres=# select tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'));`
	`tsquery_phrase`
	`------------------`
	`'fat' <-> 'cat'`
	`(1 row)`
	`postgres=#`

tsquery_phrase

函数名称	tsquery_phrase
参数数据类型	query1 tsquery, query2 tsquery, distance integer
函数返回值数据类型	tsquery
函数含义	构造一个词组查询，该词组搜索匹配query1且query2完全distance分开词素的匹配项

使用举例	`postgres=# select tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);`
	`tsquery_phrase`
	`-------------------`
	`'fat' <10> 'cat'`
	`(1 row)`
	`postgres=#`

tsvector_to_array

函数名称	tsvector_to_array
参数数据类型	tsvector
函数返回值数据类型	text[]
函数含义	将tsvector转换为词素数组

使用举例	`postgres=# select tsvector_to_array('fat:2,4 cat:3 rat:5A' ::tsvector) ;`
	`tsvector_to_array`
	`--------------------`
	`{cat,fat,rat}`
	`(1 row)`
	`postgres=#`

unnest

函数名称	unnest
参数数据类型	tsvector
函数返回值数据类型	setof record (lexeme text, positions smallint[], weights text)

函数含义	将tsvector扩展为一行一个词素
使用举例	

```
postgres=# select * from unnest('cat:3 fat:2,4 rat:5A'
::tsvector);
 lexeme | positions | weights
--------+-----------+---------
 cat    | {3}       | {D}
 fat    | {2,4}     | {D,D}
 rat    | {5}       | {A}
(3 rows)
postgres=#
```

abbrev

函数名称	abbrev
参数数据类型	inet
函数返回值数据类型	text
函数含义	将inet转换为text的缩写显示格式
使用举例	

```
postgres=# select abbrev(inet '10.1.1.3');
abbrev
----------
10.1.1.3
(1 row)
postgres=#
```

abbrev

函数名称	abbrev
参数数据类型	cidr
函数返回值数据类型	text
函数含义	将cidr转换为text的缩写显示格式，返回结果中去掉了netmask中右边的零
使用举例	

```
postgres=# select abbrev(cidr '10.1.0.0/16');
abbrev
---------
10.1/16
(1 row)
postgres=#
```

broadcast

函数名称	broadcast
参数数据类型	inet

函数返回值数据类型	inet
函数含义	计算一个ip地址的广播地址

使用举例	`postgres=# select broadcast(inet '192.168.1.5/24');`
	`broadcast`
	`------------------`
	`192.168.1.255/24`
	`(1 row)`
	`postgres=#`

family

函数名称	family
参数数据类型	inet
函数返回值数据类型	integer
函数含义	返回一个地址是IPV4（返回4）还是IPV6（返回6）

使用举例	`postgres=# select family('fe80::5054:ff:fe99:540c') ;`
	`family`
	`--------`
	`6`
	`(1 row)`
	` postgres=# select family(inet '172.1.0.23') ;`
	` family`
	` --------`
	` 4`
	` (1 row)`
	`postgres=#`

host

函数名称	host
参数数据类型	inet
函数返回值数据类型	text
函数含义	返回IP地址的text形式，且忽略掉netmask

使用举例	`postgres=# select host(inet '192.168.1.0/24');`
	`host`
	`-------------`
	`192.168.1.0`
	`(1 row)`
	`postgres=#`

hostmask

函数名称	hostmask
参数数据类型	Inet
函数返回值数据类型	Inet
函数含义	返回ip地址的host mask

使用举例	`postgres=# select hostmask(inet '192.168.23.20/30') ;`
	`hostmask`
	`----------`
	`0.0.0.3`
	`(1 row)`
	`postgres=#`

inet_merge

函数名称	inet_merge
参数数据类型	inet, inet
函数返回值数据类型	cidr
函数含义	返回包含两个给定网络的最小网络

使用举例	`postgres=# select inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24');`
	`inet_merge`
	`----------`
	`192.168.0.0/22`
	`(1 row)`
	`postgres=#`

inet_same_family

函数名称	inet_same_family
参数数据类型	inet，inet
函数返回值数据类型	bool
函数含义	返回两个ip地址是否属于同一个family

使用举例	`postgres=# select inet_same_family(inet '192.168.1.5/24', inet '::1');`
	`inet_same_family`
	`----------`
	`f`
	`(1 row)`
	`postgres=#`

masklen

函数名称	masklen
参数数据类型	inet
函数返回值数据类型	integer
函数含义	返回以位为单位的网络掩码长度

使用举例	`postgres=# select masklen(inet '192.168.1.5/24');`
	`masklen`
	`---------`
	`24`
	`(1 row)`
	`postgres=#`

netmask

函数名称	netmask
参数数据类型	inet
函数返回值数据类型	inet
函数含义	返回网络地址的网络掩码

使用举例	`postgres=# select netmask(inet '192.168.1.5/24');`
	`netmask`
	`---------------`
	`255.255.255.0`
	`(1 row)`
	`postgres=#`

network

函数名称	network
参数数据类型	inet
函数返回值数据类型	inet
函数含义	返回地址的网络部分，将网络掩码右边的内容清零

使用举例	`postgres=# select network(inet '192.168.1.5/24') ;`
	`network`
	`----------------`
	`192.168.1.0/24`
	`(1 row)`
	`postgres=#`

set_masklen

函数名称	set_masklen

参数数据类型	inet, integer
函数返回值数据类型	inet
函数含义	为inet设置netmask长度，网络地址部分保持不变

使用举例	`postgres=# select set_masklen(inet '192.168.1.5/24', 16);`
	`set_masklen`
	`----------`
	`192.168.1.5/16`
	`(1 row)`
	`postgres=#`

set_masklen

函数名称	set_masklen
参数数据类型	cidr, integer
函数返回值数据类型	cidr
函数含义	为cidr设置netmask长度，网络地址右边的设置为零

使用举例	`postgres=# select set_masklen(cidr '192.168.1.0/24', 16);`
	`set_masklen`
	`----------`
	`192.168.0.0/16`
	`(1 row)`
	`postgres=#`

text

函数名称	text
参数数据类型	inet
函数返回值数据类型	text
函数含义	返回ip地址及其掩码的text形式

使用举例	`postgres=# select text(inet '192.168.1.5');`
	`text`
	`----------`
	`192.168.1.5/32`
	`(1 row)`
	`postgres=#`

trunc

函数名称	trunc
参数数据类型	macaddr
函数返回值数据类型	macaddr

函数含义	将mac地址的最后三个字节清零，前缀可以与特定的网卡制造商关联

使用举例	`postgres=# select trunc(macaddr '12:34:56:78:90:ab') ;`
	`trunc`
	`-------------------`
	`12:34:56:00:00:00`
	`(1 row)`
	`postgres=#`

trunc

函数名称	trunc
参数数据类型	macaddr8
函数返回值数据类型	macaddr8
函数含义	将mac地址的最后五个字节清零，前缀可以与特定的网卡制造商关联

使用举例	`postgres=# select trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')`
	`;`
	`trunc`
	`-------------------------`
	`12:34:56:00:00:00:00:00`
	`(1 row)`
	`postgres=#`

macaddr8_set7bit

函数名称	macaddr8_set7bit
参数数据类型	macaddr8
函数返回值数据类型	macaddr8
函数含义	将地址的第7位设置为1，从而创建被称为修改后的EUI-64，以包含在IPv6地址中。

使用举例	`postgres=# select macaddr8_set7bit(macaddr8`
	`'00:34:56:ab:cd:ef');`
	`macaddr8_set7bit`
	`-------------------------`
	`02:34:56:ff:fe:ab:cd:ef`
	`(1 row)`
	`postgres=#`

area

函数名称	area
参数数据类型	geometric_type

函数返回值数据类型	double precision
函数含义	计算面积，针对box, path, circle适用，path必须是能closed，否则会返回NULL

使用举例	`postgres=# select area(box '(2,2),(0,0)');`
	`area`
	`------`
	`4`
	`(1 row)`
	`postgres=#`

center

函数名称	center
参数数据类型	geometric_type
函数返回值数据类型	point
函数含义	计算中点，针对box, circle适用

使用举例	`postgres=# select center(box '(1,2),(0,0)') ;`
	` center`
	` ---------`
	` (0.5,1)`
	` (1 row)`
	`postgres=#`

diagonal

函数名称	diagonal
参数数据类型	box
函数返回值数据类型	lseg
函数含义	获取box的对角线作为一个line segment，作用与lseg(box)一样

使用举例	`postgres=# select diagonal(box '(1,2),(0,0)');`
	`diagonal`
	`---------------`
	`[(1,2),(0,0)]`
	`(1 row)`
	`postgres=#`

diameter

函数名称	diameter
参数数据类型	circle
函数返回值数据类型	double precision

函数含义	计算circle的直径

使用举例	`postgres=# select diameter(circle '<(0,0),2>') ;`
	`diameter`
	`----------`
	`4`
	`(1 row)`
	`postgres=#`

height

函数名称	height
参数数据类型	box
函数返回值数据类型	double precision
函数含义	计算box的垂直大小

使用举例	`postgres=# select height(box '(1,2),(0,0)');`
	`height`
	`--------`
	`2`
	`(1 row)`
	`postgres=#`

isclosed

函数名称	isclosed
参数数据类型	path
函数返回值数据类型	bool
函数含义	path是否是closed的

使用举例	`postgres=# select isclosed(path '((0,0),(1,1),(2,0))');`
	`isclosed`
	`----------`
	`t`
	`(1 row)`
	`postgres=#`

isopen

函数名称	isopen
参数数据类型	path
函数返回值数据类型	bool
函数含义	Path是否是open的

使用举例	

```
postgres=# select isopen(path '[(0,0),(1,1),(2,0)]');
isopen
 --------
  t
(1 row)
postgres=#
```

length

函数名称	length
参数数据类型	geometric_type
函数返回值数据类型	double precision
函数含义	计算总长度，对lseg以及path适用
使用举例	

```
postgres=# select length(path '((-1,0),(1,0))') ;
 length
 --------
  4
(1 row)
postgres=#
```

npoints

函数名称	npoints
参数数据类型	geometric_type
函数返回值数据类型	integer
函数含义	返回point的数量，对path以及polygon适用
使用举例	

```
postgres=# select npoints(path '[(0,0),(1,1),(2,0)]');
npoints
---------
 3
 (1 row)
postgres=#
```

pclose

函数名称	pclose
参数数据类型	path
函数返回值数据类型	path
函数含义	将patch转换为closed的形式

使用举例	`postgres=# select pclose(path '[(0,0),(1,1),(2,0)]');`
	`pclose`
	`----------`
	`((0,0),(1,1),(2,0))`
	`(1 row)`
	`postgres=#`

popen

函数名称	popen
参数数据类型	path
函数返回值数据类型	path
函数含义	将patch转换为open的形式

使用举例	`postgres=# select popen(path '((0,0),(1,1),(2,0))');`
	`popen`
	`----------`
	`[(0,0),(1,1),(2,0)]`
	`(1 row)`
	`postgres=#`

radius

函数名称	radius
参数数据类型	circle
函数返回值数据类型	double precision
函数含义	求一个圆形的半径

使用举例	`postgres=# select radius(circle '<(0,0),2>');`
	` radius`
	` --------`
	` 2`
	` (1 row)`
	`postgres=#`

slope

函数名称	slope
参数数据类型	point,point
函数返回值数据类型	double precision
函数含义	计算通过两个点的直线的斜率

使用举例	

```
postgres=# select slope(point '(0,0)', point '(2,1)');
 slope
 -------
  0.5
(1 row)
postgres=#
```

width

函数名称	width
参数数据类型	double precision
函数返回值数据类型	box
函数含义	计算box的horizontal size
使用举例	

```
postgres=# select width(box '(1,2),(0,0)') ;
width
-------
1
(1 row)
postgres=#
```

box

函数名称	box
参数数据类型	circle
函数返回值数据类型	box
函数含义	计算圆内切的box
使用举例	

```
postgres=# select box(circle '<(0,0),2>');
box
-------------------------------------------------------------
--------------------
(1.414213562373095,1.414213562373095),(-1.414213562373095,-
1.414213562373095)
(1 row)
postgres=#
```

常见问题解答

.1. FAQ

IvorySQL贡献的许可

如果您提交的贡献是原创作品，那么您可以假设IvorySQL将作为整个IvorySQL版本的一部分发布给下游用户，该版本将遵循Apache许可证2.0版本。

如果您提交的内容不是原创作品，同样鼓励代码共享和尊重原作者的著作权，同样允许代码修改，再发布。请注意需要满足如下条件：

1、需要给代码的用户一份Apache许可证。

2、如果您修改了代码，需要在被修改的文件中说明。

3、在延伸的代码中（修改和有源代码衍生的代码中）需要带有原来代码中的协议、商标、专利声明和其他原来作者规定需要包含的说明。

4、如果再发布的产品中包含一个Notice文件，则在Notice文件中需要带有Apache许可证。您可以在Notice中增加自己的许可，但不可以表现为对Apache许可证构成更改。

最后，请记住，从非原始的工作中删除许可标头从来都不是一个好主意。即使您使用的文件部分最初在顶部有许可标头，您也应该保留它。与往常一样，如果您不太确定您的贡献所涉及的许可问题，请随时在开发人员邮件列表中联系我们。

编码指南

您获得反馈和看到代码合并到项目中的机会在很大程度上取决于更改的粒度。如果您的想法发生了更大的变化，我们强烈建议您在花大量时间编写代码之前，先加入开发人员的邮件列表，并与我们分享您的建议。即使您的建议得到社区的验证，我们仍然建议您将实际工作作为一系列小型的、独立的提交来完成。这使得评审员的工作更加容易，并提高了反馈的及时性。

当谈到IvorySQL的C和C++部分时，我们尝试遵循PostgreSQL编码约定。除此之外：

对于C和Perl代码，如果需要，请运行pgindent。我们建议在查看更改时使用git diff—color，这样您提交的代码中就不会出现任何虚假的空白问题。

所有贡献给IvorySQL的新功能都应该由与其一起贡献的回归测试覆盖。如果您不确定如何测试或记录您的工作，请在ivorysql-hackers邮件列表中提出问题，社区的开发人员将尽力帮助您。

至少，您应该始终运行make installcheck world，以确保您没有破坏任何东西。

适用于上游PostgreSQL的更改

如果您正在进行的更改涉及PostgreSQL和IvorySQL之间的通用功能，则可能会要求您将其转发到PostgreSQL。这不仅是为了我们不断减少两个项目之间的差异，而且是为了让与PostgreSQL相关的任何变化都能从对上游PostgreSQL社区更广泛的审查中受益。一般来说，将这两个代码库都放在手边是个好主意，这样您就可以确定您的更改是否需要前移。

补丁提交

一旦您准备好与IvorySQL核心团队和IvorySQL社区的其他成员共享您的工作，您应该将所有提交推送到从官方IvorySQL派生的分支的您自己的存储库中，并向我们发送请求。

补丁审查

假定提交的拉取请求通过验证检查，可供同行审查。同行审查是确保对IvorySQL的贡献具有高质量并与路线图和社区期望保持一致的过程。我们鼓励IvorySQL社区的每个成员审查请求并提供反馈。由于您不必成为核

心团队成员就可以做到这一点，因此我们建议您向有兴趣成为IvorySQL长期贡献者的任何人提供一系列拉动式评论。

同行评审的一个结果可能是达成共识，即您需要以某些方式修改pull请求。GitHub允许您将其他提交推送到从中发送请求的分支中。这些额外的提交将对所有审阅者可见。

当同行评议收到参与者至少+1张+1和no-
1张的选票时，同行评议会趋于一致。在这一点上，您应该期望核心团队成员之一将您的更改引入到项目中
。

在补丁审查期间的任何时候，您都可能会因审查人员和核心团队成员的工作效率而遇到延迟。请耐心点，也不要气馁。如果您在几天内没有收到预期的反馈，请添加一条评论，要求更新pull请求本身，或者向邮件列表发送一封电子邮件。

直接提交到存储库

有时，您会看到核心团队成员直接提交到存储库，而无需执行pull请求工作流。这仅适用于小的更改，我们使用的经验法则是：如果更改涉及任何可能导致测试失败的功能，那么它必须通过pull请求工作流。另一方面，如果更改发生在代码库的非功能部分（例如在注释块中修复打字错误），则核心团队成员可以决定直接提交到存储库。