

IvorySQL

Version v4.0, 2021-09-12

欢迎	1
发行说明	2
关于IvorySQL	2
IvorySQL入门	2
1. 基本介绍	2
2. 内核原理	2
3. 日志操作	2
4. 高级指南	2
IvorySQL高级	2
1. 磁盘操作	2
2. 事务处理	2
3. 表空间管理	2
4. 逻辑复制指南	2
5. 迁移指南	2
IvorySQL生态	2
1. PostGIS	2
2. Pvector	2
3. Pgroonga	2
4. pgdd (DDL Extractor)	2
5. pgRouting	2
IvorySQL架构设计	2
1. 语句处理	2
2. 兼容性	2
Oracle兼容功能列表	2
1. 1. 框架设计	2
2. 2. GUC框架	2
3. 3. 大小写转换	2
4. 4. 双模式设计	2
5. 5. 兼容Oracle like	2
6. 6. 兼容Oracle序列	2
7. 7. 兼容Oracle存储过程	2
8. 8. 兼容Oracle内置函数	2
9. 9. 新增Oracle兼容模式的端口与IP	2
10. 10. XML函数	2
11. 11. 兼容Oracle sequence	2
12. 12. 包	2
13. 13. 不可见列	2
社区贡献指南	2
1. 1. 项目参考	2
2. 2. FAQ	2

# 欢迎

IvorySQL 是唯一一个基于 PostgreSQL 的类的兼容 Oracle 的数据库。

## 开始

在 GitHub 上查看代码。

## 发布

请前往 IvorySQL 官方页面。

## 关于 IvorySQL

IvorySQL 可以看做是 PostgreSQL 的一个开源分支。我们希望它能成为更好的 PostgreSQL 数据库。

IvorySQL 的源码可以免费使用。如果您有任何建议或反馈 support@ivorysql.org

## 文档下载

IvorySQL v0.8.pdf 文档





# 关于IvorySQL

## IvorySQL简介

### 概述

IvorySQL是一个基于PostgreSQL的数据库，开发兼容Oracle的开源数据库。

IvorySQL旨在为开发者提供PostgreSQL和Oracle数据兼容性(100%兼容)，并可以直连管理最新的版本的PostgreSQL。

IvorySQL通过以下方式实现对PostgreSQL和Oracle的兼容性：选择使用 `oracle` 或 `pg` 外部名，在对SQL语句进行处理时，通过指定 `oracle` 或 `pg` 外部名来指定兼容的兼容模式。或者通过 `ivorysql_compatible_mode` 参数指定兼容的兼容模式。或者通过 `ivorysql_compatible_mode` 参数指定兼容的兼容模式，或者通过 `oracle` 或者不指定 `oracle` 参数指定兼容的兼容模式。或者通过 `set ivorysql_compatible_mode to pg` 或者通过 `set ivorysql_compatible_mode to oracle` 并且不指定 `oracle` 或 `pg` 来获得数据源100%支持PostgreSQL的语法及功能。

IvorySQL兼容之最PL/SQL语句兼容，支持Oracle的PL/SQL语法。同时，IvorySQL通过与内部编译的组件 `ivorysql ora` 来实现对兼容Oracle的功能。兼容的功能包括内嵌函数、数据类型、存储过程、函数以及 merge 以及Oracle特有的兼容。未来将陆续以模块化的方式实现兼容的兼容功能。

IvorySQL项目是在Apache 2.0许可证下发布的，社区鼓励对项目提出类型的功能和修复。

### 产品目标和范围

我们致力于通过开源的，我们希望建立一个健康的和积极的社区。我们还做什么，讨论想法可以来自任何人，虽然IvorySQL项目的根本主要是对Oracle兼容性，但未来的方向和问题将由社区公开投票决定。

### 核心特性

IvorySQL是基于PostgreSQL的数据库，与Oracle数据兼容，且有强大的兼容性，适用于PostgreSQL数据源和Oracle数据源。

### 竞争优势

核心优势：IvorySQL的核心优势在于兼容性，兼容广泛的SQL语句，并有一个活跃的开发者社区。

兼容Oracle：可以对Oracle数据源进行完全的兼容。

可定制性：只适用于定制功能，并根据需求进行定制。

易部署：对系统管理员来说，IvorySQL大幅降低了部署和维护的难度。对开发者来说，IvorySQL提供了简单的部署和测试方法。

稳定性：因为IvorySQL的数据源是开源的，所以稳定性。

### 技术生态

IvorySQL是基于PostgreSQL，具有完整的API，坚实稳定的生态系统和庞大的生态系统。

### 核心应用场景

IvorySQL的主要应用场景：

企业级应用

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。

在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。

智能家居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)。用于整合物件模型，支持最佳实践，并且在处理速度的领导者行列。除此之外，它兼容了Oracle的语法，适用于使用Oracle的社区。

## 与Oracle的兼容性

- IvorySQL架构设计
- 以太坊架构
- 大小写敏感
- 嵌式SQL
- 兼容Oracle语句
- 兼容Oracle视图语法
- 兼容Oracle游标与存储过程
- 兼容Oracle表与内部函数
- 兼容Oracle索引与外部函数
- 表函数
- 兼容Oracle sequence
- 视图
- 兼容视图

# IvorySQL入门

## 1. 快速开始

## 环境要求

## · 硬件要求

MySQL数据库目前支持的操作系统与5.6时不同，CentOS 8.x、CentOS Stream 9.0及Ubuntu 20.04等。

## 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

#### • 安裝前準備

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vanya用户为例。如何创建sudo用户

#### • 下載安裝

创建或编辑MySQL yum 源配置文件 /etc/yum.repos.d/mysql.repo

```
vim /etc/yum.repos.d/ivorysql.repo
[ivorysql4]
name=IvorySQL Server 4 $releasever - $basearch
baseurl=https://yum.highgo.com/dists/ivorysql-rpms/4/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=0
```

```
$ sudo dnf install -y ivorysql4-4.6
```

迁移完成后，数据将被安装在 `lustre-4/` 文件夹内。

```
$ sudo chown -R ivorysql:ivorysql /usr/ivory-4
```

卷之三

将以下配置写入~/.bash\_profile文件并使用source命令读文件使终端变量生效：

```
PATH=/usr/ivory-4/bin:$PATH
export PATH
PGDATA=/usr/ivory-4/data
export PGDATA
```

```
$ source ~/bash_profile
```

```
$ initdb -D /usr/ivory-4/data
```

其中会使用到配置文件的设置参数，更多参数使用方法，请使用 initdb --help 命令查看。

▪ 数据库服务

```
$ pg_ctl -D /usr/ivory-4/data -l ivory.log start
```

其中会使用到配置文件的设置参数，如配置了 PGDATA，则参数可以省略。更多参数使用方法，请使用 pg\_ctl --help 命令查看。

查看通过数据库启动成功：

```
$ ps -ef | grep postgres
ivorysql 3214 1 0 20:35 ? 00:00:00 /usr/ivory-4/bin/postgres -D
/usr/ivory-4/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

▪ Docker Hub 上的 IvorySQL 镜像

```
$ docker pull ivorysql/ivorysql:4.6-ubi8
```

▪ IvorySQL

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/ivorysql:4.6-ubi8
```

▪ 查看 IvorySQL 进程状态成功

\$ docker ps   grep ivorysql					
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	
6faa2d0ed705	ivorysql:4.6-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 命令行工具

```
$ psql -d <database>
psql (17.6)
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。MySQL 默认使用 mysql 数据库，但较老版本的 MySQL 首次使用时需手动先连接 postgres 数据库，然后自己创建 mysql 数据库。较新版本的 MySQL 则已为用户创建好 mysql 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MySQL时，需要添加额外参数，如 `mysql -d mysql -U mysql -h 127.0.0.1 -p 5433`

## 2. 日常监控

## 监控数据活动

## 标准Unix工具

在大部分 Unix 平台上，MySQL 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。一个显示群集

```
$ 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:
checkpoint
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
```

postgres: user database host activity

如果配置了cluster\_name，则集群的名字也将会显示在‘ps’的输出中：

```
$ psql -c 'SHOW cluster name'
```

cluster name

-----

server1

```
$ 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\_ids，那么活动监视器将不会被更新，进程树仅在新进程被启动的时候设置一次。在某些平台上这样值可以为每个命令行参数的开销，但在其它平台上却可能没有。

## 提示

Solana需要特别的处理。你需要使用`/usr/lib/ps`而不是`/bin/ps`。你还需要使用两个`-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，每个服务器进程读的`ps`输出将是你所希望的`postgres`命令行`command line`。

## 统计收集器

MySQL 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，计数可以按数据库和个体文件进行。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。

NonSQL 技术提供有关系统正在处理什么的数据信息，例如当正在处理其他服务时正在执行的命令以及系统中存在哪些其他连接。这些信息对于诊断和故障排除非常有用。

## 统计收集配置

因为统计收集检查语句执行增加了一些负担，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

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的元数据

参数track\_functions启用对用户定义函数使用的跟踪。

### 查看统计信息

表1中列出了一个典型的1000英里航程的燃油消耗率。表2中列出了同一艘船在不同航速下的燃油消耗率。

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”或“pg\_stat\_xact\_user\_functions”中看到自己的统计信息（还没有被传递给收集器）。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地更新。

表1-1是会员动态统计图中的一些信息是有着业务限制的。普通用户只能看到关于他们自己的会员的所有信息（属于他们是成员的角色的会员）。在其他会员的统计中，许多列为空白。但是，请注意，一个会员的存在和它的一般属性，例如会员用户名和数据库，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员的所有信息。

表1.动态统计视图

针对每个索引的统计信息对于判断哪个索引已被使用以及它们的效果特别有用。

pg\_stat\_statistic，系统全局主要用它来判断统计区的范围。当实际磁盘读取数大于小于此范围时，这个范围能满足大部分请求的范围将进行不同的操作。这些统计信息并没有给出所有的东西：由于MySQL QLQ处理是I/O的方式，不是在MySQL QLQ被处理时的数据库内部膨胀在内存的I/O操作中，然后回收，所以再次读取时不需要物理地回收。我们建议希望了解MySQL QLQ行为更多细节的用户将MySQL统计工具和操作系统中坏块检测的I/O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

### pg\_stat\_activity

`wait_event` 和 `state` 例是独立的。如果一个线程处于 `'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'` 为单空，它意味着一个查询正在被执行，但是它被阻塞在系统中。

表4.等待事件表

表5 'Activity' 类型的等待事件

做了‘Client’类型的客体属性



ReadIndexCollect	等待读集一个对象的读者或者某一个范围内的读者。
ReadIndexDelete	等待其他对象的读者或者某一个范围内的读者。
ReadIndexDeleteReader	等待其他对象的读者或者某一个范围内的读者。
ReadIndexDeleteLocate	等待读集一个对象的读者或者某一个范围内的读者。
ReadIndexDeleteLock	等待读集一个对象的读者或者某一个范围内的读者。
ReadIndexDeleteText	等待读集一个对象的读者或者某一个范围内的读者。
LogicalSyncData	等待逻辑复制线程将数据写入到本地磁盘。
LogicalSyncStateChange	等待逻辑复制线程将状态写入到本地磁盘。
MessageQueueInterval	等待逻辑复制线程将间隔写入到本地磁盘。
MessageQueuePutMessage	等待将消息放入到本地消息队列。
MessageQueueReceive	等待从本地消息队列读取消息。
MessageQueueSend	等待将消息写入到本地消息队列。
ParallelBrowsingScan	等待并行扫描线程将扫描结果写入到本地磁盘。
ParallelCreateIndexScan	等待并行索引创建线程将索引写入到本地磁盘。
ParallelDeleteIndexScan	等待并行索引删除线程将索引写入到本地磁盘。
ParallelFlush	等待并行刷写线程将数据写入到本地磁盘。
ParallelIndexUpdate	等待并行索引更新线程将索引写入到本地磁盘。
PreSignature	等待前缀索引线程将前缀写入到本地磁盘。
Promise	等待承诺线程将承诺写入到本地磁盘。
RecoveryConflictSnapshot	等待冲突快照线程将冲突快照写入到本地磁盘。
RecoveryConflictTablespace	等待冲突表空间线程将冲突表空间写入到本地磁盘。
RecoveryPause	等待暂停线程将暂停写入到本地磁盘。
ReplicationBeginDrop	等待复制线程将半数的线程数写入到本地磁盘。
ReplicationEndDrop	等待复制线程将半数的线程数写入到本地磁盘。
SafeSnapshot	等待快照线程将快照写入到本地磁盘。
SyncRep	在同步复制线程将同步复制快照写入到本地磁盘。
SyncRepUpdate	等待同步线程将同步复制快照写入到本地磁盘。

表11: Lock 类型的事件列表

Lock 事件事件	描述
advisory	等待获得一个建议锁。
extend	等待扩展一个表。
freelist	等待从 pg_database_free_list 或 pg_database_damaged
object	等待对象半数的线程数写入到本地磁盘。
page	等待一个页面上锁。
relation	等待一个关系的锁。
specimen	等待视图的插入锁。
transactionid	等待事务ID的锁。
tuple	等待元组上锁。
userlock	等待锁的写锁。
virtualoid	等待锁的读锁。

表12: Lock 事件的事件操作

LMlock 事件事件	描述
AdvisoryLock	等待管理共享内存对象的锁。
Autofac	等待更改 pg_persistence.conf 文件。
Autovacuum	等待读或更新自动 vacuum 的锁。
AutovacuumSchedule	等待自动计划的锁。
BackgroundWorker	等待读或更新后台工作线程。
BitwiseScan	等待读或更新后台线程的锁。
BufferContent	等待读或更新缓冲区的锁。
BufferIO	等待读或更新缓冲区的锁。
BufferFlushing	等待读或更新缓冲区的锁。
Checkpoint	等待一个检查点。
CheckpointWriter	等待写入。
CommitTs	等待读或更新备库交易日志的一个锁。
CommitterFetcher	在文件系统上从备库读取一个快照。
CommitterIO	等待从文件系统读取一个快照。
ControlFile	等待读或更新 pg_control 文件或创建一个新的控制文件。
DynamicSharedMemoryControl	等待读或更新动态共享内存的锁。
LockFilePath	等待读或更新文件路径的锁。
LockManager	等待读或更新名为 "newpageid" 的锁。
LogicalRepWorker	等待读或更新逻辑复制线程的锁。
NotifyListener	等待读或更新通知线程的锁。
NotifyHeaderBuffer	在 pg_notify 表以及通知队列上写入的锁。
NotifyHeaderBufferIO	等待读或更新通知线程的锁。
NotifyHeaderBufferIO	在 pg_notify 表以及通知队列上写入的锁。
NotifyHeaderTruncation	等待读或更新通知线程的锁。
NotifyBuffer	在 NOTIFY 表以及通知队列上写入的锁。
NotifyQueue	等待读或更新 NOTIFY 的锁。
NotifyQueueTail	等待 NOTIFY 表尾端的更新锁。
NotifyGID	等待读或更新通知线程的锁。
DidScan	等待一个扫描的锁。
DidSnapshotTimelag	等待读或更新快照线程的锁。
ParallelApend	在并行插入线程将插入的数据写入到本地磁盘。
ParallelApendScan	在并行插入线程将插入的数据写入到本地磁盘。
ParallelOpenGSA	等待并行打开全局索引的锁。
PerSessionGSA	等待并行打开全局索引的锁。
PerSessionRecordType	等待并行插入线程将插入的数据写入到本地磁盘。
PerSessionRecordType	等待并行插入线程将插入的数据写入到本地磁盘。
PerSessionPredicateList	在并行插入线程将插入的数据写入到本地磁盘。
PredicateListManager	等待并行插入线程将插入的数据写入到本地磁盘。
Procray	等待读或更新 pg_shmemmap 文件的锁。
RelationFlushing	等待读或更新 pg_shmemmap 文件的锁。

表11 'Timeout'类型的等待事件

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备用服务器的统计信息。只有直接连接的备用设备将列出没有关于下游备用服务器的信息。

## 14 pg\_stat\_replication

对于数据量较小的自动识别是毫无问题。但当数据量较大的话会消耗相当长的识别时间。如果它们没有这样做，那么识别速度将远远地慢于

## 注意

相关的查询语句中相关的查询语句会发送到多个连接上执行的线程。在新的WAL生成时间，这样一种机制是必要的，但是当发送者为不同的线程时，特别是当设备本地化线程时，`pg\_stat\_replication` 会是线程的线程，而不仅仅是设备本地化线程。这种做法为线程的线程提供了更好的线程和线程之间的隔离。为了确保线程在一种线程模型中的线程，在一个线程模型的线程上，`pg\_stat` 会以线程的线程的线程的线程。

## pg\_stat\_wal\_receiver

表`pg_stat_wal_receiver` 有四行。它显示了从WAL接收器接收到的数据的有关信息的什么信息。

表`pg_stat_wal_receiver` 信息

列类型描述
<code>pid</code> integer WAL接收器进程的进程ID
<code>status</code> text WAL接收器的状态
<code>receive_start_lsn</code> reg_lsn WAL接收器收到的最新的第一个写入的位置
<code>receive_start_txi</code> integer WAL接收器收到的最新的第一个写入的事务ID
<code>written_lsn</code> reg_lsn 已经写入到WAL接收器的最新的一个写入的位置，也有新人，这使得数据完整性检查。
<code>flushed_lsn</code> reg_lsn 已经被写入到WAL接收器的最新的一个写入的位置，这使得数据完整性检查。
<code>received_xid</code> integer (接收到的)WAL接收器收到的最新的一个写入的位置的写入的事务ID
<code>last_xlog_send_time</code> timestamp with time zone 从WAL发送到WAL接收器的最新的一个写入的位置的时间
<code>last_xlog_receive_time</code> timestamp with time zone 从WAL接收器收到的最新的一个写入的位置的时间
<code>last_xlog_end_lsn</code> reg_lsn 从WAL接收器收到的最新的一个写入的位置
<code>last_xlog_end_time</code> timestamp with time zone 从WAL发送到WAL接收器的最新的一个写入的位置的时间
<code>stat_name</code> text 这个WAL接收器的统计信息
<code>sender_port</code> integer 这个WAL接收器的PostgreSQL的端口号
<code>connection</code> text 这个WAL接收器使用的连接字符串。对某些类型的连接字符串进行了截断。

## pg\_stat\_subscription

每一个订阅的工作都在`pg_stat_subscription` 表中有一行 (如果有工作者的话)。没有被订阅的数据被标记为工作者没有参与的行。

表`pg_stat_subscription` 信息

列类型描述
<code>subid</code> int 订阅ID
<code>subname</code> name 订阅的名称
<code>pid</code> integer 订阅工作正在运行的PID
<code>read_lsn</code> reg_lsn 订阅工作收到的最新的一个写入的位置
<code>received_lsn</code> reg_lsn 订阅工作收到的最新的一个写入的位置，这使得数据完整性检查。
<code>last_xlog_send_time</code> timestamp with time zone 从WAL发送到WAL接收器的最新的一个写入的位置的时间
<code>last_xlog_receive_time</code> timestamp with time zone 从WAL接收器收到的最新的一个写入的位置的时间
<code>last_xlog_end_lsn</code> reg_lsn 从WAL接收器收到的最新的一个写入的位置
<code>last_xlog_end_time</code> timestamp with time zone 从WAL发送到WAL接收器的最新的一个写入的位置的时间

## pg\_stat\_ssl

`pg_stat_ssl` 表因为每一个连接或者WAL发送器包含一行。如果只是这个连接上的SSL使用情况，可以认为`pg_stat_activity` 或 `pg_stat_replication` 通过`pid` 行来获得更多的有关连接的信息。

表`pg_stat_ssl` 信息

列类型描述
<code>pid</code> integer 后面的WAL发送器进程ID
<code>ssl</code> boolean 如果正在使用SSL，因为真
<code>version</code> text 使用SSL的版本，如果连接上没有使用SSL因为null
<code>cipher</code> text 正在使用的SSL使用的密码，如果连接上没有使用SSL因为null
<code>bits</code> integer 使用的SSL连接中的密钥，如果连接上没有使用SSL因为null
<code>compression</code> boolean 如果正在使用SSL的压缩，否则为假，如果正在使用SSL因为null
<code>client_dname</code> name 客户端的别名，如果连接上没有使用SSL因为null，如果客户端长于MATERIALIZE为多于64个字符，将字符串截断。
<code>client_version</code> numeric 客户端连接的版本，如果连接上没有使用SSL因为null，如果客户端长于MATERIALIZE为多于64个字符，将字符串截断。
<code>issuer_dn</code> name 客户端对连接者识别的Dn，Distinguished Name，如果连接上没有使用SSL因为null，将字符串 <code>client_dname</code> 一并使用。

## pg\_stat\_gssapi

`pg_stat_gssapi` 表将包含一行，其中包含所有GSSAPI连接的信息。

表`pg_stat_gssapi` 信息

列类型描述
<code>pid</code> integer 后面的连接ID
<code>gss_authenticated</code> boolean 如果正在连接使用了GSSAPI的身份验证，因为true
<code>principal_text</code> text 用于GSSAPI连接的用户名，因为null，如果用户名长于MATERIALIZE为多于64个字符，将字符串截断。
<code>encrypted</code> boolean 如果在此连接上使用了GSSAPI加密，因为真

## pg\_stat\_archiver

`pg_stat_archiver` 表总包含一行，其中包含所有归档工作的连接的信息。

表`pg_stat_archiver` 信息

列类型描述
<code>archived_count</code> integer 已归档的文件的文件数
<code>last_archived_wal</code> text 最后一个成功归档的从文件的名称
<code>last_archived_time</code> timestamp with time zone 最后一次成功归档操作的时间
<code>failed_count</code> integer 记录WAL文件的失败文件数

## 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

## §120 pg\_stat\_bgwriter

### 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。

图21 pg\_stat\_database

## pg\_stat\_database.conflicts

第22章 stat database conflicts

pg stat all tables

'`pg stat all tables`' 的图标为当前数据库中的每一个表 (包括 TOAST 表) 提供一行, 每行包含有关该表的表结构和统计信息。'pg stat user tables' 和 'pg stat sys tables' 提供对各列的快照, 但是通过过得分别只是显示用户和系统的表。

## 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，但是被过滤只分别显示用户和系统索引。

第24章 stat-all-indexes 209

吸引人可以被看作引脚，“地脚”吸引地脚以及优化器使用。在一次回路中，多个寄存器的输出可以通过 AND 或 OR 组合起来，因此，当使用一次回路时将可以取得个体的逻辑和特定的寄存器联系起来。因此，一次回路将增加它使用的寄存器 `pg_start_all_idxmem_idx_top_fetch`，并且为每个寄存器 `pg_start_all_linesidx_idx_top_fetch`，如果外设提供的寄存器不在优化器统计记录的寄存器之内，优化器也会对访问者引用产生怀疑，因为优化器估计性能可能已经“不新鲜”了。

### 注意

### pg\_statio\_all\_tables

'`pr statio all tables`' 相当于为当前数据库中的每个表 (94 TOAST 表) 扫描一行, 扫描的是表的表上有多少 TOAST 的统计信息。'pr statio user tables' 和 'pr statio all tables' 提供的各相同的讯息, 只是被过滤成分别只是对用户表和系统表。

pg statio all indexes

### pg\_statio\_all\_sequences

'pg\_statio\_all\_sequences' 视图将为当前数据库中的每个序列的最后一行，该行是带有指定序列上有关 I/O 的统计信息。

## pg\_stat\_user\_functions

### pg\_stat\_slru

MySQL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` 视图将为每个被追踪的SLRU缓存包含一行，显示关于访问缓存页面的统计信息。

## 表22 pg\_stat\_allrw

## Statistics Functions

其他查看他状态的方法是直接使用 `ps`。这些命令会使用上述标准图形识别的系统状态信息查询函数。如要了解进程数的基本用法, 可参考 [标准图形识别](#) (例如, 在 `ps` 中你可以发出 `ps -d ps_stat_activity`)。针对每一个数据源统计信息的访问函数将一个数据源 ID 作为参数来从报告器返回数据源。而针对每个类和每个子类的函数要求参数或输出 ID。针对每个函数统计信息的函数用一个函数 ID。注意只有在当前数据源中的类、属性和方法才能使这些函数有效。

更多统计集的函数列在表 30 中

©2020 Additional Statistics Functions

```
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;
```

### 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。这样就允许直接地监视正在被数据库锁住的数据块。例如，这个功能可以被用来

- 查看以前所有未解决的，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MySQL会话持有的所有的锁。
- 判断当前数据库中带有最多未授予的锁（它很可能是数据库客户端的竞争源）。

## Progress Reporting

MongoDB具有在命令执行过程中嵌入某些命令语句的能力。目前，支持该语句的命令只有`ANALYZE`、`CLUSTER CREATE`、`INDEX VACUUM`和`BASE BACKUP`例如`pg_basebackup`发出的语句是包含它的定制命令。未来可能还会扩展。

## ANALYZE Progress Reporting

当运行 ANALYZE 语句时, `pg_stat_progress_analyze` 相应地包含运行语句的每个后端的行。下面的表格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`pg_stat_progress_analyze` 信息

字段	描述
<code>pid</code>	后端的进程ID。
<code>datid</code>	后端读取的数据集的OID。
<code>dbname</code>	后端连接的数据库的名称。
<code>relid</code>	被分析的表的OID。
<code>phase</code>	分析的阶段。参见 表 5.1。
<code>sample_size_total</code>	后端采集的总的数据。
<code>sample_size_scanned</code>	后端扫描的数据。
<code>ext_stats_total</code>	后端对表的统计信息。
<code>ext_stats_computed</code>	后端对表的统计信息。统计数现在在“computing statistics”阶段使用。
<code>child_table_total</code>	后端的数据。
<code>child_table_done</code>	后端对表的统计。此数据只有在 <code>acquiring_inherited_sample_rows</code> 时才会显示。
<code>current_child_table_relid</code>	当前正在扫描的子表的OID。此字段只有在 <code>acquiring_inherited_sample_rows</code> 时才会显示。

表`pg_analyze_phases`

阶段	描述
<code>initializing</code>	后端正在准备阶段。这个阶段将包含单行。
<code>acquiring_sample_rows</code>	后端正在扫描 <code>child_table_total</code> 相应的表。此阶段将包含单行。
<code>acquiring_inherited_sample_rows</code>	后端正在扫描 <code>child_table_total</code> 相应的表。此阶段将包含单行。
<code>computing_statistics</code>	后端从表中读取数据并使用统计数进行计算。
<code>computing_extended_statistics</code>	后端从表中读取数据并使用统计数进行计算。
<code>finalizing_analyze</code>	后端在更新 <code>pg_stat_analyze</code> 之后将返回时。 <code>ANALYZE</code> 完成。

## 注意

当在分区表上运行 ANALYZE 语句, 它的所有分区也需要分析, 和 ANALYZE 不同的是, 在这种情况下, 首先运行父级的 ANALYZE 语句, 然后它会通知所有子表, 然后是每个分区的本地统计信息。

## CREATE INDEX Progress Reporting

当运行 CREATE INDEX 或 REINDEX 语句, `pg_stat_progress_create_index` 相应地包含运行语句的每个后端的行。下面的表格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`pg_stat_progress_create_index` 信息

字段	描述
<code>pid</code>	后端的进程ID。
<code>datid</code>	后端读取的数据集的OID。
<code>dbname</code>	后端连接的数据库的名称。
<code>relid</code>	正在被索引的表的OID。
<code>index_relid</code>	正在创建或重建索引的OID。在执行“CREATE INDEX”语句时, 此为0。
<code>comment_text</code>	在命令的末尾: “CREATE INDEX CONCURRENTLY REINDEX”或“REINDEX CONCURRENTLY”。
<code>phase</code>	后端正在运行的阶段。参见 表 5.1。
<code>lockers_total</code>	后端正在运行了, 需要等待的锁的总数。
<code>lockers_done</code>	后端正在运行的锁的总数。
<code>current_locker_pid</code>	后端正在运行的锁的锁的进程ID。
<code>blocks_total</code>	后端正在运行的块的总数。
<code>blocks_done</code>	后端正在运行的块的块的块数。
<code>tuple_total</code>	后端正在运行的元组的总数。
<code>tuple_done</code>	后端正在运行的元组的元组数。
<code>partitions_total</code>	后端正在运行的分区的总数。
<code>partitions_done</code>	后端正在运行的分区的分区数。

表`CREATE INDEX` 信息

阶段	描述
<code>CREATE</code>	<code>CREATE INDEX</code> 或 <code>REINDEX</code> 正在准备语句的语句。这个阶段将包含单行。
<code>构建前的单个写入</code>	<code>CREATE INDEX CONCURRENTLY</code> 或 <code>REINDEX CONCURRENTLY</code> 正在等待可能需要写入的事务完成。当不在并发模式时, 这个阶段会跳过。 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>构建写入</code>	当语句的语句正在运行时, 支持单个写入的语句可以在此阶段已成功完成。语句正在“只读”状态。通常情况下, <code>blocks_total</code> 和 <code>blocks_done</code> 将包含语句数据, 语句将包含 <code>tuple_total</code> 和 <code>tuple_done</code> 。
<code>在旧的前单个写入</code>	<code>CREATE INDEX CONCURRENTLY</code> 或 <code>REINDEX CONCURRENTLY</code> 正在等待将写入的数据从旧的前单个写入语句中移除。当不在并发模式时, 这个阶段会跳过。 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>事务识别: 只读语句</code>	<code>CREATE INDEX CONCURRENTLY</code> 正在等待语句完成, 需要将语句从旧的前单个写入语句中移除。如果不在并发模式, 这个阶段会跳过。 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>锁确认</code>	<code>CREATE INDEX CONCURRENTLY</code> 正在等待语句完成, 以保证语句的锁被释放。这个阶段将包含 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>标记语句: 只读语句</code>	<code>CREATE INDEX CONCURRENTLY</code> 正在等待语句完成, 以保证语句的锁被释放。这个阶段将包含 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>标记语句: 只读语句</code>	<code>CREATE INDEX CONCURRENTLY</code> 正在等待语句完成, 以保证语句的锁被释放。这个阶段将包含 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>标记语句: 只读语句</code>	<code>CREATE INDEX CONCURRENTLY</code> 正在等待语句完成, 以保证语句的锁被释放。这个阶段将包含 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。
<code>在 dropping 之前单个写入</code>	<code>REINDEX CONCURRENTLY</code> 等待表上的锁的语句完成, 再执行语句。当不在并发模式时, 这个阶段会跳过。 <code>lockers_total</code> 、 <code>lockers_done</code> 及 <code>current_locker_pid</code> 将包含这个阶段的语句信息。

## VACUUM 进度报告

当运行 VACUUM 正在运行, 每一个后台正在表的回滚 (包括vacuum工具进程) 在 `pg_stat_progress_vacuum` 相应地包含一行。下面的表格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`pg_stat_progress_vacuum` 信息

字段	描述
<code>pid</code>	后端的进程ID。
<code>datid</code>	后端读取的数据集的OID。
<code>dbname</code>	后端连接的数据库的名称。
<code>relid</code>	被分析的表的OID。
<code>phase</code>	vacuum的阶段。参见 表 5.1。
<code>read_blocks_total</code>	后端读取的块的总数。这个数字在扫描阶段的语句。之后增加的语句不会 (并且不需要) 被这个 VACUUM 语句。

heap_free_scanned bigint	被扫描的堆对象数。当为 visibility map 时用来统计堆。一些体积很大的对象，被通过的堆对象数放在这个参数中。因此当清理完对这个数据操作命令是 heap_free_total。它类似于 VACUUM 但只统计这个对象数的次数。
heap_free_total bigint	被清理的堆对象数。除非设置为 0，这个参数只对大于 10000 行的表起作用。不过当行大于 10000 行时，这个参数对所有行都起作用。因此这个参数将影响到你必须清理一个巨大的行数。
index_free_count bigint	对索引的清理次数。
max_free_tuples bigint	在需要时对一个单行清空操作之前我们所认为的死亡行数。以及对 maintenance_work_mem。
max_free_tuples bigint	从上一个行清空操作以来清空的死亡行数。

表名 VACUUM 的信息

阶段	描述
初始化	'VACUUM' 还未开始时的堆栈。这个阶段会很短。
扫描堆	'VACUUM' 正在扫描堆。如果需要，这个阶段会很长。如果行数大于 10000，这个阶段会更长。不过当行数大于 10000 时，这个阶段会更短。
清理堆	'VACUUM' 正在清理堆。如果需要，这个阶段会很长。如果行数大于 10000，这个阶段会更短。
清理索引	'VACUUM' 正在清理索引。如果需要，这个阶段会很长。如果行数大于 10000，这个阶段会更短。
清理表	'VACUUM' 正在清理表。如果需要，这个阶段会很长。如果行数大于 10000，这个阶段会更短。
重排堆	'VACUUM' 正在重排堆。如果需要，这个阶段会很长。如果行数大于 10000，这个阶段会更短。
执行后的清理	'VACUUM' 在开始后的清理。在这个阶段中，'VACUUM' 将被完全忘记。更新 pg_class 中的行数并清除行的锁。如果这个阶段完成，'VACUUM' 也就结束了。

## CLUSTER进度报告

每当 'CLUSTER' 或 'VACUUM FULL' 运行时，'pg\_stat\_progress\_cluster' 视图将为您提供正在运行的一个后台的过程。下面的表格展示了所需要的信息，并提供了关于如何解释这些信息的指南。

表名 pg\_stat\_progress\_cluster 的信息

列名	描述
pid	Integer. 当前的进程ID。
datid	oid. 从后端连接到数据库的OID。
datname	字符串. 与后端连接到的数据的名称。
relid	oid. 被操作的表的OID。
commtid	Text. 正在运行命令。'CLUSTER' 或 'VACUUM FULL'。
phase	Text. 表示正在运行。
cluster_index_relid	oid. 如果正在使用索引进行清理，这是正在使用的索引的OID。如果为 0，则未使用。
heap_scanned bigint	扫描堆的行数。这个计数器只有在命令为 'pg_scanning_heap_index_scanning_heap' 或 'writing_new_heap' 时才会增加。
heap_scanned_written bigint	插入到堆的行数。这个计数器只有在命令为 'pg_scanning_heap_index_scanning_heap' 或 'writing_new_heap' 时才会增加。
heap_scanned_written_percent	表示在堆中插入的行数的百分比。
heap_free_total bigint	被清理的堆对象数。这个计数器只有在命令为 'pg_scanning_heap' 时才会增加。
heap_free_scanned bigint	对堆对象的清理次数。这个计数器只有在命令为 'pg_scanning_heap' 时才会增加。
index_relid_id_count bigint	被清理的索引数。统计数据只在 '重排堆' 时显示。

表名CLUSTER 和 VACUUM FULL 的信息

阶段	描述
初始化	你永远不会看到这个阶段。这个阶段只包含空字符串。
重排堆	你永远不会看到这个阶段。这个阶段只包含空字符串。
重排索引堆	'CLUSTER' 或 'VACUUM FULL' 正在对索引进行排序。
大归并排序	'CLUSTER' 或 'VACUUM FULL' 正在对表进行排序。
插入大堆	'CLUSTER' 或 'VACUUM FULL' 正在插入大堆。
文件写文件	期间，你永远不会看到这个阶段。
重排堆	期间，你永远不会看到这个阶段。
堆	你永远不会看到这个阶段。当所有命令完成，'CLUSTER' 或 'VACUUM FULL' 将结束。

## 基础备份进度报告

每当执行 pg\_basebackup 时，该视图将显示有关备份的进度。'pg\_stat\_progress\_basebackup' 视图将包含与执行 'BAKUP' 命令时和准备的每个 WAL 是无关的。下面的表格展示了所需要的信息，并提供了关于如何解释这些信息的指南。

表名 pg\_stat\_progress\_basebackup 的信息

列名	描述
pid	WAL 处理中的进程ID。
phase	Text. 表示正在运行。
backup_total bigint	将要处理的数据总字节数。这是在 streaming database files 时按分钟的估计和报告。注意，这只是一个估计值，因为有 streaming database files 时，数据可能被改变。WAL 日志可能包含在待处理的备份中。一些数据量超过了估计的大小，将使用命令 'backup_streamed'。如果在 pg_basebackup 中使用 'size' 选项，指定了 'no-estimate-size' 选项，这为 'NULL'。
backup_streamed bigint	数据的总字节数。这是在 streaming database files 时显示 'transferring wal files' 时显示。
tablespace_total bigint	要处理的数据空间数。总计数据只在 'streaming database files' 时显示。
tablespace_streamed bigint	数据的空间数。总计数据只在 'streaming database files' 时显示。

表名基备份进度的

阶段	描述
initializing	WAL 处理已经开始准备备份。这个阶段只包含空字符串。
waiting for checkpoint to finish	WAL 处理进程将正在执行 'pg_start_backup' 以准备进行数据备份。这将待到检查点准备好时再开始。
estimating backup size	WAL 处理将根据正在执行的命令为准备备份的数据库文件的总字节数。
streaming database files	WAL 处理将正在的数据文件作为数据备份。
waiting for wal archiving to finish	WAL 处理将正在写入到归档中产生的所有 WAL。如果在 pg_basebackup 中指定了 'wal-method=archive' 或 'wal-method=stream'，则会在完成写入之后，当归档完成时再开始。
transferring wal files	WAL 处理将正在写入到归档中产生的所有 WAL。如果在 pg_basebackup 中指定了 'wal-method=archive'，则会在完成写入之后，当归档完成时再开始。

## 动态追踪

动态追踪是 PostgreSQL 对内存文件数据提供的一种高级功能。这种技术并不在计划中使用，而是上层的外部工具来运行的行数。

一些实用的追踪点已经在表中列出。这些实用的追踪点被数据文件和数据文件使用。要以命令行，运行一个单独的 pg\_probes 中，使用 pg\_probes 的命令行工具。在命令行上可以修改更改 pg/include/pg\_probes.h 中的定义文件。

目前，DTrace 只被支持，它在 Solaris、FreeBSD、NetBSD 和 Oracle Linux 3.4 版本。Linux 的 SystemTap 项目提供了一种对 DTrace 的支持。支持的动态追踪工具在命令行上可以修改更改 pg/include/pg\_probes.h 中的定义文件。

## 动态追踪的编译

默认情况下，动态追踪是不可用的，因此你将需要以适当的配置来运行 pg\_probes。要包括 DTrace 支持，在配置文件中添加 'enable-dtrace'。

## 内建探针

如表43所示，源代码中提供了一些标准探针。表43展示了在探针中使用的类型。当然，可以增加更多探针来增强InnoDB的可观测性。

表42. 内部 DTrace 语句

第43章 文档在设计数据中的应用

## 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，该例子会输出如下的输出：

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C
```

Start	71
Commit	70
Total time (ns)	2312105013

## 注意

SystemTap 为追踪器使用一个不同的 DTrace 标记，但是底层的实现是相同的。也就是说，在这样写的时候，SystemTap 核本必须使用不同的代码来实现相同的逻辑。在未来的 SystemTap 版本中它将可能会被修复。但在此之前，DTrace 核本将要细心地编写和测试。否则被收集的数据将不能被有意义地使用。在大部分关心问题的情况下，它将产生有用的输出。当它使用的话，当它使用的话，它将产生有用的输出。当它使用的话，它将产生有用的输出。

## 定义新探针

开发者可以在任何时候往覆写现有的探针，当然也需要在编译之后才生效。下面是插入新的探针步骤：

```
1. 定义探针名以及使用可用的数据
2. 将探针插入到文件中。可以在 dtrace/probe.h 中插入
3. 编译。"make" 会自动打包并将其放入到 dtrace 目录中。包括它，并且在源码中更新的往插入 "TRACE_POSTGRESQL" 为注释
4. 重新编译并运行即可。如果一切正常
```

例子：这是一个相对增加一个探针来重新 DTrace 为新参数的例子。

```
1. 定义探针将使用名为 "transaction_start" 并且需要一个 LocalTransactionId 类型的数据
2. 将探针插入到文件中。可以在 dtrace/probe.h 中插入
```

```
...
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。

1) 首先，`transaction\_start`脚本做一个宏`TRACE\_POSTGRESQL\_TRANSACTION\_START` (注意是单引号)。可以通过文本文件`pg\_trace.h`替换，将它添加到用户的头文件夹。在这种情况下，看起來是这样的：

```
```  
TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);  
```
```

4) 重新编译和运行之后，通过运行下面的Dtrace命令检查新添加的代码是否可用。你将看到类似于下面的输出：

```
```  
# 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  
```
```

运行脚本并从端口15432时，每一行输出需要注释：

• 要小心的是，为脚本添加注释的语句类型是与它使用的变量的数据类型，否则会失败并报错。  
• 在大多数平台上，如果用`-enable-dtrace`编译了`pgSQL`，无论何时当它接收到一个连接时，都会评估它的参数。即使没有运行`pgSQL`，通常一个需要的参数只会在接收到一个连接时评估。但是要注意不要在开始时的函数调用时评估参数，如果评估过早，可能会导致错误的连接失败。

```
```  
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())  
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));  
```
```

每个语句都有一个特别的`ENABLE`语句。

## 监控磁盘的使用

### 判断磁盘用量

每个表都有一个主要的辅助函数，大多数函数都返回一个字符串，这是一个字符串或者从表（`relname`）的列。另外还有一个TCL命令可以读取表的存储统计。它用于存储表的主要统计信息，如果有这个函数，那么TCL脚本会有一个可用的命令。当然，同时也可能有其他存储信息。每个表和每个表的存储信息文件名，如果文件超过100字节，那么可能需要一个文件。

你可以通过以下两种方法监控磁盘空间：使用`pg\_relation\_filepath`或者人工操作系统命令。SQL语句是最佳的使用方法，同时也是我们推荐使用的方法。当判断表的存储大小时通过操作系统命令来获得空间。

在一个最近被修改过的表的限制上使用`oid`，你可以交叉查询并查看对表的磁盘用量。

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';
```

```
pg_relation_filepath | relpages  
-----+-----  
base/16384/16806 | 60  
(1 row)
```

最后一个通常是在`pg\_stat`中，`pg\_stat`包含`VACUUM`，`ANALYZE`和几个`OID`的子句`CREATE INDEX`的更新。如果想要查询表的磁盘文件，那么文件名将由`oid`的值决定。

要是对TCL有更好的认识，我们可以使用一个类似于这样的查询：

```

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

你也可以将查询结果导出到文件中：

```

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

我们强烈建议您使用pg\_statistic表来查询：

```

SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

```

relname	relpages
bigtable	3290
customer	3144

磁盘满失败

一个数据库表如果超过了它的磁盘空间限制，可能会导致数据无法写入。这时，系统会尝试将数据写入到另一个磁盘空间，如果失败，可能会导致数据丢失。

如果不能通过增加一些其他的存储空间来解决磁盘空间不足的问题，那么可以考虑使用快照或一些数据文件移动到其他文件系统上去。



## 更新可见性映射

清理机制为每一个页面维护着一个可见性映射。它被用来追踪哪些页面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）可见的光标。这样就有两个目的。第一，清理本身可以在下一次执行时跳过这样的页面，因为其中没有什么需要被清除。

## 防止事务ID回卷失败

虽然我们不能直接使用这个公式，但我们可以从一个更复杂的公式推导出这个结果。假设  $PC$  的值为 0.45，那么这个公式可以表示为  $PC = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \cdot \text{FeatureA})}}$ 。通过求解这个方程，我们可以得到  $\text{FeatureA}$  的临界值，即当  $\text{FeatureA} > 2.07$  时， $PC$  将大于 0.5。因此，我们可以将  $\text{FeatureA}$  的值设置为 2.07，从而确保模型在所有情况下都能输出正确的结果。当然，这只是一个简单的示例，实际情况可能需要更复杂的模型和更多的特征。

`vacuum_freeze_min_age`控制在其行版本被冻结前一个XID 位应该有多老。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消耗的事务数。

一个表保持不被清理的最长时间是 10 亿个事务。如果 `VACUUM` 次数达到此限制时的 `vacuum_freeze_min_age` 值。如果它超过或没有被清理，可能会导致数据库死锁。要避免这个问题发生，将在任何包含 `autovacuum_min_freeze` 语句参数所指定的年龄更大的表上的 `NOVACUUM` 行语句中使用清理（即使自动清理被禁用也不会发生）。

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenid),age(t.relfrozenid)) 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' 列度量从该断点 XD 到当前事务 XD 的事务数。

如果由于某种原因自动清理无法从一个表中清除掉的 XID，当数据块的最古老 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.

(如你所见所期待的，一次手动的“VACUUM”应该会修复问题；但是要注意这次“VACUUM”必须由一个超级用户执行，否则它将无法处理系统日志并且因此不能推断出数据库的“datachecksum”）。如果这些警告被忽略，一旦更新回卷只剩下 3 百万事务时，该系统将被关闭并且开始执行新的事务。

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 百万条事务的安全副本是为了让管理员能够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，由于一旦系统进入的安全模式，它将不会执行命令。要做这个操作的唯一方法是停止服务器并以单一用户模式启动服务器执行“VACUUM”。单一用户模式下不会强制读取回滚段。

“Multi”只用到两个参数的头部。由于在一个头部只可能有一个参数，所以只有两个参数时才能用到“两个参数”（或称两个参数“头部”）。但对带有两个参数头部的参数，它必须是两个参数头部的 *versus* 申明。参数头部也是用一个`[[参数名]]`来申明的。在两个参数头部中有一个独立的头部可以采用参数头部。它也使用一个`[[参数名]]`来申明参数头部。

在第一次 'VACUUM' 调用时 (部分或全部) 期间, 任何将 `vacuum_multixact_freeze_min_age` 老化的多事例 ID 会替换为一个不同的值。值可以是零值, 一个唯一事例 ID 或者一个更新的多事例 ID。对于每一个表, `pg_class.relimnid` 存储了在该表被元数据中仍然存在的最老的多事例 ID。如果这个值比 `vacuum_multixact_freeze_table_age` 大, 将强制一次表扫描。可以在 `pg_class.relimnid` 上使用 `��表_id` 来限制它的值。

全表 'VACUUM' 扫描（不管是什么导致它们）将为表掉进锁。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被通过，较老的多事务的磁盘存储可以被释放。

作为一种安全设备,对任何多事年齡超过 `autovacuum_multicat_freeze_max_age` 的表,都将发生一次全表清潔过程。当多事年齡属性的存储超过 2GB 时,从那以后具有最长多事年齡的表开始,全表清潔过程也将逐步在所有表上进行。即使自动清潔被在表上被禁用,也将发生这种两种扫描过程。

## 自动清理后台进程

MySQL有一个可选的性能优化器叫“autocompact”，它的目的是自动执行“VACUUM”和“ANALYZE”命令。当它被启用时，它会定期自动地对表进行整理，从而减少插入、更新或删除操作的开销。这些检查会利用统计信息收集功能，因此即使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，并且如果配置了innodb\_flush\_neighbors，自动清理将被禁用。

如果在一小段时间内多个大型表都变得难以读清，所有的自动进程工作者可能都会将其所在的一段长时间内清理掉这些。这将会影响到其他的表和数据库无法读清，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，也是工作者们确实会因资源耗尽而被其他工作者完成的工作。注意进行归档工作者的数量不会被计入`max_connections`或`superuser_reserved_connections`等限制。

“neurodiverse”被译为“神经多样”、“神经多元”等，表示在智力平等的层面上是全然正常（这与智商相关的表述是十分类似，通过智商的高低来判断智力水平，是完全不同的）。然而，如果从上文“IQ=100”引出来的神经多样性被译为“神经多样”，那就完全错误，神经多样性含义为：

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`, 先检数为`'pg_class'.reltuple`

如果自上次清理以来插入的数据数量超过了字典的插入限制，虚拟会话将清理，该函数将返回1。

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

对于分析，也使用了一个相似的函数：

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。

默认的值和恢复系数都取自于 `postgresql.conf`，但是可以为每一个表设置它们（许多其他语句也能设置参数）。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。

当多个工作者行动时，在所有进行着的工作之间有的选择地延迟数据是“平滑的”，这样不使实际行动的工作者数量是零，对于系统的总人口影响是相相同的。不过，任何时候在操作时已经准备了两个 *autovacuum vacuum cost delay* 或 *autovacuum vacuum cost limit* 在存储参数的表的工作者不会考虑在操作算法。

荀子

日常重建索引

在某些情况下，你很可能会发现INDIV会生成一系列的独立事件，而不仅仅是事件本身。

已经完全变成的静态引用页面被反向使用。但是，还是有一种低效的空间利用的可能性：如果一个页面上缺少某些引用之外的全部链接被删除，该页面仍然会被分配。因此，在这种每个页面中大部分甚至是全部被最终使用的模式中，可以看到空间的使用是低效的。对于这样的使用模式，推荐使用空间节省的

对于半当时布引可变的参数还没有很时的定量分析。在使用半当时布引时定期监控布引的物理尺寸是个时的主要

还有，对于新的吸引，一个新的吸引更新了多次的吸引会看起来要更稳定，因为在新建立的吸引上，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静态的吸引）。仅仅为了提高访问并发度，将所有吸引

REINDX在所有情况下都可以安全和有效地使用。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

如果你想要把“postman”的“Request”写到一个文件中，你会得到乱码输出。若是想清晰且以文件的唯一方法是停止并禁起服务器。这样你对于所有文件中使用的MySQL问题都是很头疼的。若是你首先不想在生产环境下这样

一个更好的办法是将服务器的Redis输出发送到某种形式的队列程序中。我们有一个构建的日本队列程序，你可以通过在“postmaster.conf”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方法将所有数据被捕获或机器产生的CSV（逗号分隔值）格式

另外,如果在你已启用的其他服务器组件中有一个外部以太网接口,你可能更喜欢使用它。比如,包含在Apache反向代理的`reverse-proxy`工具就可以使用`tinySQL`。要做到这一点,方法之一是启用服务器的`ultra`管道将流量定向到你的组件。如果你用`pg_ctl`启动服务器,那么`ultra`管道定向的`addrs`。因此你只需要一个管道命令,比如:

## 注意

不过，在很多系统上，syslog还是非常可靠，特别是在面对大量日志消息的情况下；它可能在你需要那些消息的时候截断或者丢弃它们。另外，在Linux，syslog会将每个消息的写到磁盘上，这将导致很慢的性能（你可以在syslog配置文件里把文件名头使用一个“\*”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对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

## 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个数据源而带来的同步开销。它使用一个由多个服务器共享的第一级链阵列。如果主数据源服务器失效，后备服务器则可以挂载共享的数据源。就像它从一次数据崩溃中恢复过来了一样。这是一种快速的故障转移，并且不存在数据丢失。

共享裸带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。但是要注意的是文件系统必须具有完全的POSIX行为。这种方法的一个重大限制是如果共享磁阵阵列失效或损坏，主要和备份服务器都必须无法工作。另一个问题是主要服务器运行时，备份服务器永远不能访问共享存储。

## 文件系统（块设备）复制

共享硬盘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映射到位于另一台计算机上的一个文件系统。唯一的限制是该映像必须保留在所服务器有一份该文件系统的一致的拷贝。特别的是对所服务器的写入必须限制在机上操作的进程运行。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

通常和新服务器能够通过读取一个预定式日志 (WAL) 记录的流来保持为当前状态。如果主服务器失败，新服务器将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[日志传递](#)、[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通知另一台服务器。MySQL逻辑复制从WAL机制读取逻辑复制更新。逻辑复制允许包含表复制数据更改。此外，发布数据更新的服务器可以同时向其他服务器发布更改，从而允许数据在多个方向流动。第二方工具也能提供类似的功能。

## 基于触发器的主-备复制

基于触发器的变更通常将修改数据的查询语句放到取数的主服务器。它在两个表的基础上工作，主服务器（通常）将数据修改并发送到备服务器。主服务器运行时，备服务器可以响应查询，并执行本地数据修改或写入操作。这种形式的变更通常用于减少大数据分析平台或者数据仓库负担。

Slowy-VM是这种复制类型的一个例子。它使用卷快照，从而支持多个后台服务线程。因为它会同步更新后台服务线程（共享），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，一个程序员只需要一个SQL语句并将其发送给一个或多个服务器。每一个服务器将自动操作。读写分离的逻辑将发送给所有服务器，这样每一个服务器都将根据收到的逻辑操作。他只读查询将被只发送给一个服务器。这样许多读请求将在服务器之间进行负载均衡。

## 异步多主控机复制

对于不会定期连接或通过铁路传输的服务器，如笔记本或远程服务器，保持服务间的数据一致是一个挑战。通过使用同步的主主复制，每一个服务器独立工作并定期与其他服务器通信来解决冲突的事务。这些冲突可以由客户端或冲突检测规则来解决。Bucardo 是这种复制类型的一个例子。

## 同步多主控机复制

NeoSQL 不提供这种复制模型，尽管在应用阶段或中间件中可以使用 NeoSQL 的事务提交 PREPARE TRANSACTION & COMMIT PREPARED 方案实现。

下面总结了上述各种方案的能力。

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

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结果返回给一个中心服务器。由它整合结果并生成报告。



host	replication	foo	192.168.1.100/32	md5
------	-------------	-----	------------------	-----

主从复制的主从机的IP号，依赖于从机的primary\_conninfo配置。在从复制机上还可以在“/var/lib/pgsql”文件中设置命令（见“**database**”命令配置“replication”）。例如，如果主从机的IP是192.168.1.50，端口号是5432，从机IP为192.168.1.100，那么可以在从复制机的“postgreSQL”文件中增加以下命令：

```
# 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

从复制的一个重要特性就是从机在主控机上产生的没有在从机上产生的WAL记录数。可以通过从主控机上的当前WAL位置和从复制机的最后一次WAL位置来计算这个差值。从这个位置开始可以向主控机上写“`pg_current_wal_lsn`”和从复制机上写“`pg_last_wal_receive_lsn`”来检测。从复制机的最后一次WAL位置的完整性也在WAL记录的逻辑状态中。如果写“`pg_wal`”命令显示的状态。

从机通过“`pg_wal_receive_lsn`”命令检测从机的WAL完整性检测的失败。“`pg_current_wal_lsn`”和“`pg_wal_lsn`”之间的差值表示从机从主控机接受到的WAL记录数。从“`pg_wal_lsn`”和“`pg_last_wal_receive_lsn`”之间的差值表示从机从主控机接受到的WAL记录数。

在一台新的副本上，WAL块的被迁移的记录可以通过“`pg_wal_receive_lsn`”和“`pg_wal_lsn`”的差值计算WAL块的被迁移的大小。被迁移的块数。

## 复制槽

复制槽提供了一种自动化的从主控机上自动从所有的从机或从机段之间复制它们。并且主控机的从机段可能带有冲突冲突的。从机段的冲突也是本地的。

从机槽的使用，也可以使用“`wal_level`”和“`max_wal_size`”的参数。不过，还是这个参数会比其他的WAL设置更多的，主要是因为从机槽的WAL设置对于它们来说是必须的。另一方面，复制槽可以保留很多的WAL段以至于它们真的为了使用“`pg_wal`”的空闲“`max_wal_level`”和“`max_wal_size`”。

另外，“`hot_standby_feedback`”和“`vacuum_defer_cleanup_age`”保护了从机槽不被“vacuum”删除。但是前者在从机槽中被忽略，而后者只需要设置为一个很高的值以保证它的保护。复制槽扩展了这些功能。

## 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。

已有的复制槽它们的状态可以在“`pg_replication_slots`”中查看。

槽可以通过复制槽的设置或者“`SHOW`”语句进行设置。

## 配置实例

你可以这样设置一个复制槽：

```
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)
```

如果要从备机使用这个槽，从备机的连接名是“`primary_slot_name`”，这是一个全局的名：

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'  
primary_slot_name = 'node_a_slot'
```

## 级联复制

级联复制的特性之一是当从机的从机段收到主控机的WAL记录时会自动地将其他的备机段，就像一个链一样。这可以使用高小对于主控机的WAL记录数并会使得从机的WAL记录数小化。

一级的从机从主控机和从机的备机段作为一级的从机从备机段。“`replication`”（通过它的缺省参数从机）连接到主控机的备机段作为一级的从机，但是它需要把它的备机段作为它的备机段。如果复制没有对一级的从机的WAL记录数进行限制。

一级的从机从备机段不从主控机接收WAL记录。只要复制到从机中读取的记录，从机从备机上读取的可能和从机中读取。只要没有对从机的WAL记录数进行限制，下级的从机将不会被限制。

级联复制的从机段将从一级的从机段开始。

不能在从机的从机段中，从机从备机段的上一级开始。

如果一级的从机从备机段进行行为的迁移。当从机被移到“`recovery_target`”设置为“`latest`”（默认），下级的从机将从从机的主控机读取。







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各服务器上那个数据源的用户被强制中断连接。这个操作会立即发生。不管 'max\_standby\_streaming\_delay' 的设置是什么。注意 'ALTER DATABASE ... RENAME' 不会中断用户，这在大部分情况下不会有影响。然而如果它依赖某种基于数据流的方法，在某些情况下会中断数据流。

### 照由参数参考

## 警告

而且有一些限制。这些限制很可能在未来发行中被修复：

# IvorySQL高级

## .1. 安装指南

### 概述

IvorySQL安装方式包括以下几种：

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

本章将详细介绍每种方式的安装，包括安装过程，想要更详细的了解IvorySQL，推荐阅读“安装”。

同时，安装前请先在同一个账户，安装了IvorySQL，安装，密码和配置可以共用账户的。这里以IvorySQL4为例。

### yum源安装

创建并编辑IvorySQL yum 源配置文件/etc/yum.repos.d/ivorysql.repo。

```
vim /etc/yum.repos.d/ivorysql.repo
[ivorysql4]
name=IvorySQL Server 4 $releasever - $basearch
baseurl=https://yum.highgo.com/dists/ivorysql-rpms/4/redhat/rhel-$releasever-$basearch
enabled=1
gpgcheck=0
```

安装完成后，安装IvorySQL4。

```
$ sudo dnf install -y ivorysql4-4.6
```

### docker安装

在Docker Hub上找到IvorySQL镜像。

```
$ docker pull ivorysql/ivorysql:4.6-ubi8
```

安装完成后

```
$ docker run --name ivorysql -p 5434:5432 -e IVORYSQL_PASSWORD=your_password -d
ivorysql/ivorysql:4.6-ubi8
```

参数名	是否必填	描述
IVORYSQL_USER	是	数据库用户名，默认 ivorysql
IVORYSQL_PASSWORD	是	数据库密码，默认 ivorysql
IVORYSQL_DB	是	数据库名，默认 ivorysql
POSTGRES_HOST_AUTH_METHOD	是	修改主连接认证方式。参考值：md5
POSTGRES_INITDB_ARGS	是	为postgres启动参数，参考值：“-data-checksums”
PGDATA	是	将数据目录从数据目录下（即子目录），到 /var/lib/ivorysql/data
POSTGRES_INITDB_WALDIR	是	关闭IvorySQL transaction开关 wal，防止在数据目录（PGDATA）的目录下，



1. 请将POSTGRES\_HOST\_AUTH\_METHOD参数填写为md5，这样将安装IVORYSQL\_PASSWORD设置失败。

2. 如果POSTGRES\_HOST\_AUTH\_METHOD参数设置为 scram-sha-256，此时将POSTGRES\_INITDB\_ARGS设置为“-auth-hostname-sha-256”，才能使数据正常进行初始化。

## rpm安装

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

```
$ sudo wget  
https://github.com/IvorySQL/IvorySQL/releases/download/IvorySQL_4.6/IvorySQL-4.6-  
a50789d-20250304.x86_64.rpm
```

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

数据库安装在Ivory-4(默认)。

## 源码安装

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

```
$ git clone https://github.com/IvorySQL/IvorySQL.git  
$ cd IvorySQL  
$ git checkout -b IVORY_REL_4_STABLE origin/IVORY_REL_4_STABLE
```

```
$ ./configure --prefix=/usr/ivory-4/
```

```
$ make
```



编译完成。如果已运行 ./make check 或 make all，check 会自动检测到编译的错误。

```
$ sudo make install
```

## deb安装

```
$ 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
```

```
$ sudo wget  
https://github.com/IvorySQL/IvorySQL/releases/download/IvorySQL_4.6/IvorySQL-4.6-a50789d-20250304.amd64.deb
```

```
$ sudo dpkg -i IvorySQL-4.6-a50789d-20250304.amd64.deb
```

## 启动数据库

```
$ sudo chown -R ivorysql:ivorysql /usr/ivory-4/
```

```
PATH=/usr/ivory-4/bin:$PATH  
export PATH  
PGDATA=/usr/ivory-4/data  
export PGDATA
```

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

```
$ mkdir /usr/ivory-4/data  
$ initdb -D /usr/ivory-4/data
```

```
$ pg_ctl -D /usr/ivory-4/data -l ivory.log start
```

```
$ ps -ef | grep postgres
ivorysql 130427      1  0 02:45 ?          00:00:00 /usr/ivory-4/bin/postgres -D
/usr/ivory-4/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 -d <database>
psql (17.6)
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IvorySQL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IvorySQL则已为用户创建好ivorysql数据库，可以连接。

更多参数使用方法，请使用`psql --help`命令获取。



Documentation for IvorySQL. 需要连接到数据库。参考: `psql -d ivorysql -U ivorysql -h 127.0.0.1 -p 5432`

## 卸载IvorySQL



使用任何一种方法卸载时请先停止数据库服务并卸载数据。

yum源安装的卸载

```
$ sudo dnf remove -y ivorysql4-4.6
```

## docker安装的卸载

```
$ docker stop ivorysql
$ docker rm ivorysql
$ docker rmi ivorysql/ivorysql:4.6-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=* ivorysql4\*  
$ sudo rm -rf /usr/ivory-4/
```

## 源码安装的卸载

执行以下命令卸载并清理文件夹：

```
$ sudo make uninstall  
$ make clean  
$ sudo rm -rf /usr/ivory-4/
```

## deb安装的卸载

执行以下命令卸载并清理文件夹：

```
$ sudo dpkg -P IvorySQL-4.6  
$ sudo rm -rf /usr/ivory-4/
```

## .2. 集群搭建

### 主节点

#### 安装并启动数据库

yum将快速安装数据库，建议使用此方法。

想要获取更多安装方法，请参考[安装指南](#)。



全局数据库将被自动启动。

#### 关闭防火墙

集群各节点都需要关闭防火墙才能正常通信。

```
$ sudo systemctl stop firewalld
```

#### 配置文件

为了满足节点的连接需求限制，主节点需要在/etc/postgresql/12/main/postgresql.conf文件中修改配置。

\* postgres.conf

建议修改配置文件postgreSQL.conf文件配置。

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

\* pg\_hba.conf

建议修改配置文件pg\_hba.conf文件配置。

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



禁用pg\_hba.conf配置，仅需为demo模式启用。这种配置通常用于测试或开发环境。请根据实际情况进行配置。

重启主节点数据库服务

```
$ pg_ctl restart
```

备节点

安装数据库

禁用pg\_hba.conf配置，仅需为demo模式启用。

想要获取更多安装方法，请参考[安装指南](#)。



备节点的安装只需要安装，不需要启动。

关闭防火墙

禁用各节点的防火墙才能正常使用。

```
$ sudo systemctl stop firewalld
```

搭建流复制

在备节点上执行以下命令，创建一个主节点的备链备件，前提需要复制。

```
$ sudo pg_basebackup -F p -P -X fetch -R -h <primary_ip> -p <primary_port> -U ivorysql
-D /usr/local/ivorysql/ivorysql-4/data
```

• 人为连接的IP:

• 人为连接的端口号，缺省为5432:

• 人为连接的用户名:

• 人为要创建的备件名数据恢复使用。

配置环境变量

将以下配置写入 `~/.bash_profile` 文件：

```
PATH=/usr/local/ivorysql/ivorysql-4/bin:$PATH
export PATH
PGDATA=/usr/local/ivorysql/ivorysql-4/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-4/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walender:

```
$ 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value receiver:

```
$ 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，看查看集群状态：

```
$ psql -d ivorysql
psql (17.6)
Type "help" for help.
```

这将192.168.31.102作为备节点的p.async表示数据同步方式为异步读复制。——使读集集群中所有的写操作在主节点执行，读操作则在备节点都可以执行。主节点的数据通过读复制同步到备节点。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。



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

## 容器化指南

### ..1. K8S部署

#### 单机容器

进入容器的create节点上，创建名为ivorysql的namespace

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

进入单机的镜像

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

进入单机的镜像

```
[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

```
[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 17.6 (IvorySQL 4.6) 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
```

```
[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 17.6 (IvorySQL 4.6) 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)
```

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

## 高可用集群

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

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

```
[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values.yaml中的pvc信息及集群规模等信息，数据库密码查看templates/secret.yaml并自行修改。
```

```
[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

```
[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	

```
[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=#
```

```
[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=#
```

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

```
[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. Docker Swarm & Docker Compose部署

```
准备三个网络连接的服务器，并搭建swarm集群。测试集群对mysql的访问能力。
```

```
manager node01 192.168.21.105
```

```
manager node02 192.168.21.104
```

```
manager node03 192.168.21.106
```

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

```
[root@manager-node1 ~]# git clone https://github.com/IvorySQL/docker_library.git
[root@manager-node1 ~]# cd docker_library/docker-cluster/docker-swarm
```

```
[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-etcd1  replicated  1/1
quay.io/coreos/etcd:v3.5.8  *:2379-2380->2379-2380/tcp
sosag5017cis  ivoryhac-etcd2  replicated  1/1
quay.io/coreos/etcd:v3.5.8
8twpgkzo2mnx  ivoryhac-etcd3  replicated  1/1
quay.io/coreos/etcd:v3.5.8
```

```
[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-etcd1  replicated  1/1
quay.io/coreos/etcd:v3.5.8          *:2379-2380->2379-2380/tcp
sosag5017cis  ivoryhac-etcd2  replicated  1/1
quay.io/coreos/etcd:v3.5.8
8twpgkzo2mnx  ivoryhac-etcd3  replicated  1/1
quay.io/coreos/etcd:v3.5.8
uzdvjq5j2gwt  ivorysql-hac_ivypatroni1  replicated  1/1          ivorysql/docker-
swarm-ha-cluster:4.6-4.0.6-ubi8  *:1521->1521/tcp, *:5866->5866/tcp
```

```
fr0m9chu3ce8  ivorysql-hac_ivypatroni2  replicated  1/1      ivorysql/docker-swarm-ha-cluster:4.6-4.0.6-ubi8  *:1522->1521/tcp, *:5867->5866/tcp
```

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p1521 -U ivorysql -d ivorysql
Password for user ivorysql:
```

```
ivorysql=# select version();
              version
-----
PostgreSQL 17.6 (IvorySQL 4.6) 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 17.6 (IvorySQL 4.6) 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)
```

```
[root@manager-node1 ~] docker stack rm ivoryhac-patroni
```

```
[root@manager-node1 ~] docker stack rm ivoryhac-etcd
```

docker compose搭建IvorySQL HA Cluster

```
[root@manager-node1 ~]# git clone https://github.com/IvorySQL/docker_library.git
[root@manager-node1 ~]# cd docker_library/docker-cluster/docker-compose
```

```
[root@manager-node1 docker-compose]# docker-compose -f ./docker-compose-etcd1.yml up -d
[+] Running 1/1
  └─ Container etcd  Started
    0.1s
```

```
[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
736c0d188bdd        ivorysql/docker-compose-ha-cluster:4.6-4.0.6-ubi8   "/bin/sh /docker-ent...
9d8e04e4f819        quay.io/coreos/etcdb:v3.5.8      "/usr/local/bin/etcdb"

```

```
[root@manager-node1 docker-swarm]# psql -h127.0.0.1 -p1521 -U ivorysql -d ivorysql
Password for user ivorysql:
```

```
ivorysql=# select version();
```

```
version
```

```
PostgreSQL 17.6 (IvorySQL 4.6) 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 17.6 (IvorySQL 4.6) 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)
```

```
[root@manager-node1 ~] docker-compose -f ./docker-compose-ivypatroni_1.yml down
[root@manager-node1 ~] docker-compose -f ./docker-compose-etcd1.yml down
```

## 3. 开发者指南

### 概览

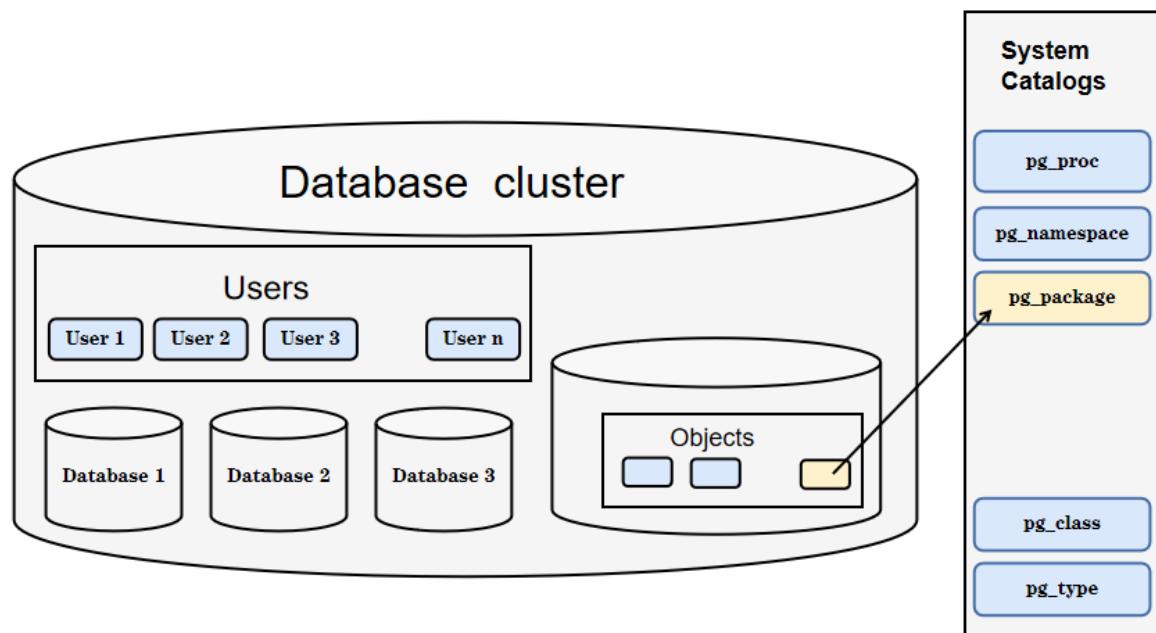
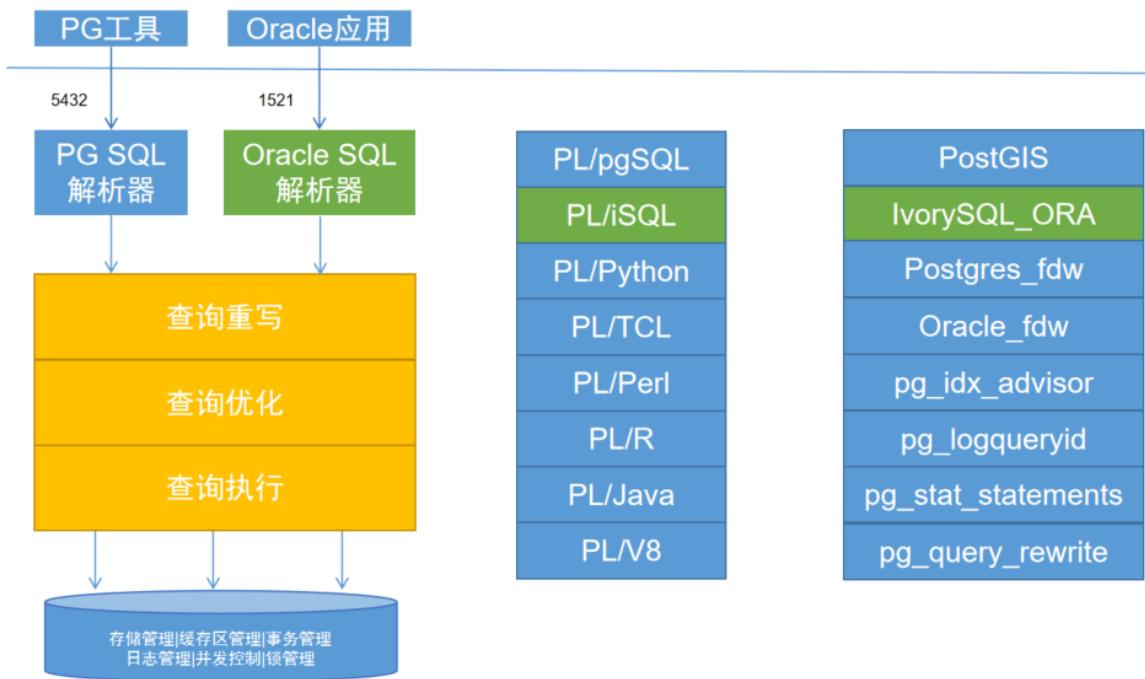
IvorySQL 是一个 PostgreSQL 数据库的分支，主要针对的是企业级应用。

IvorySQL 力求通过简化和建立在开源数据库解决方案之上为企业级用户提供价值。它的目标是为中小企业提供一个具有高性能、可扩展和易于使用的解决方案。

IvorySQL 提供的功能包括但不限于对 PostgreSQL 的兼容性、具有更好的数据一致性模型、简化了从其他数据库迁移至 PostgreSQL 的过程，等等。

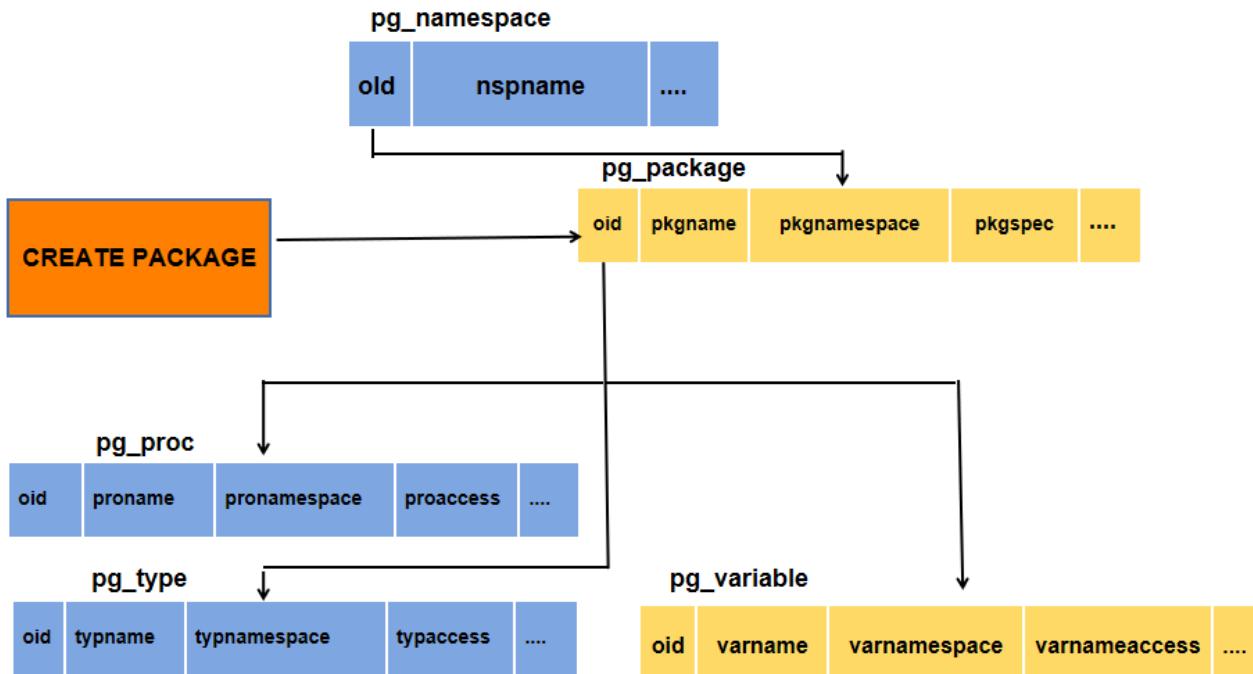
### 架构概述

在对现有的 PostgreSQL 进行最小的修改下，实现对 Oracle 的兼容。我们通过实现的 parser、存储器、持久化引擎以及 SQL 解析器，实现了对 Oracle 的兼容。



## 系统表的变化

下图展示了从PostgreSQL 10到11系统表的变化以及外部影响。



## 数据库建模（第一章创建库+第二章创建表）

### 创建一个数据库

希望你已经安装了PostgreSQL并运行了一个数据库。一台运行的MySQL或其他可以管理许多数据的服务器，通常可以为每个项目和每个用户都使用一个数据库。

你将使用管理员权限为数据库添加了可以使用的数据。如果这样你就可以安装这一步，并且跳到下一节。

要创建一个新的数据库，在执行这个命令之前`mydb`，你可以使用下面的命令：

```
$ createdb mydb
```

如果不能产生任何错误消息表示创建成功，你可以通过本节的剩余部分。

如果你看到类似于下面这样的信息：

```
createdb: command not found
```

那么你可能没有安装PostgreSQL，或者你没有安装，或者是你的系统中没有安装PostgreSQL。尝试使用以下命令来安装：

```
$ /usr/local/pgsql/bin/createdb mydb
```

在你的命令上这个命令可能不一样，取决于你使用的PostgreSQL版本或者你安装的PostgreSQL的版本。

另一个常见的问题是这样：

```
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?
```

你将看到同样的错误没有出现。或者在`createdb`命令之后使用`psql`命令查看正在运行的PostgreSQL连接。

另一个常见的问题是这样：

```
createdb: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL:
```

```
role "joe" does not exist
```

如果要使用你没有为它设置的名为 `mydb` 的用户账号，那么会发生错误。 (MySQL 用户账号和操作系统的账号是不同的。) 如果你设置过，尝试 `psql` 以你刚刚设置的账号运行。你需要安装 MySQL 的操作系统的账号 (通常是 `postgres`) 才能创建一个用户账号。也有可能是由于 MySQL 没有你的操作系统的账号 (它叫 `root`)，这时你需要以 `root` 或者使用 `psql` 才能安装你的 MySQL 用户账号。

如果你的账号没有账号，但是没有设置账号的权限，那么你会看到以下信息：

```
createdb: error: database creation failed: ERROR: permission denied to create database
```

如果所有用户都没有权限创建数据库，如果 MySQL 没有你的操作系统的账号，那么你需要以操作系统的账号运行。出现这种情况时请咨询你的系统管理员。如果你自己安装了 MySQL，那么你应该以你的数据库账号登录然后手动地修改你的配置文件。 (通常是 `my.cnf`)，<http://www.mysql.org/doc/11.1/en/privileges.html#1.4.2.11.4.2.11> [1]

你可以用其它的账号登录 MySQL。MySQL 允许在一个账号上创建它想要的数据库。数据库名还是以该账号的用户名为前缀。一个方便的策略是创建和使用不同的账号。许多工具假设数据库名为数据库的名称。所以这样可以节省你的键入。要创建新的数据库，只要键入：

```
$ createdb
```

如果没有任何账号拥有你想要的数据库了，那么你可以删掉它。比如，如果你想要删除 `mydb` 的所有 (创建人)。那么你可以用下面的命令删掉它：

```
$ dropdb mydb
```

对于过多的子句，数据将不会被插入到该账号。因此你必须选择它们。这个动作将在物理上删除与该账号相关的所有数据。因此建议操作之前一定要考虑清楚。

更多关于 `createdb` 和 `dropdb` 的信息可以到 [createdb](#) 和 [dropdb](#) 中找到。

创建一个新表

你可以通过指定表的名称和所有列的名称及其类型的语句：

```
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` 嵌入这些命令以及操作。 `psql` 可以识别命令行的账号。

你可以在 MySQL 中直接使用它 (即它，存储过程和脚本)。你需要将每行不同部分的分号，或者将命令全部放在一行中。两个分号 (";") 为注释。任何在它前面的字符将被忽略。MySQL 对关键字和标识符大小写不敏感的命名，只有在你想要区分它们的时候 (上例没有这么做)。

`varchar(10)` 定义了一个可以存储最长的十字符的字符串类型的变量。`real` 是普通的浮点型。`int` 是普通的整数型。`date` 类型的列名也是 `date`。这让你可以对方便的使用或者让人们自己选择。

MySQL 支持其他的 MySQL 类型 `int`，`smallint`，`real`，`double precision`，`char(N)`，`varchar(N)`，`date`，`time`，`timestamp` 和 `interval`。不过对这些操作的类型和丰富的日期类型，MySQL 不可以识别数据的用户定义的数据类型。因为类型的名字是英文关键字，除了 MySQL 支持的特殊外。

第二个例子将使用点和它们相关的地理信息：

```
CREATE TABLE cities (
    name          varchar(80),
    location      point
);
```

类型 `point` 就是一个 MySQL 特有的数据类型的例子。

最后，我们还要得到让你不再需要关心，或者你以不同的方式设置它。那么你可以用下面的命令删掉它：

```
DROP TABLE tablename;
```

## 写入数据 (SQL写入) 参考第6章数据操纵

在一个表被创建后, 它不能直接写数据。在数据被写入之前, 需先要创建的是插入数据。一次插入一行数据, 你也可以在一个命令中插入多行。也不要插入不完整的行, 因为它们是表中一些字段的值, 必须有正确的值。

要创建一个命令, 使用 `CREATE TABLE` 命令, 这命令需要提供表的名称和表中的列信息。例如, 考虑 `1` 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语句:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据的值是按照它们在表中出现的顺序列出的, 并且用逗号分隔。简单, 表里的值是文字 (字符串), 但也允许带有数值表达式。

上面的语法并不是你必须遵循的语法规则。要避免这个问题, 你也可以是这样的语句。例如, 下面的两条命令都有和上文命令完全一样的效果:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多语句认为顺序列出的列名是好习惯。

如果语句没有使用所有的列, 那么你可以省略表中的一些。在这种情况下, 这些未指明的列将为它们赋值。例如:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种方式是MySQL的一个扩展, 它让你能将数据从文件读取出来。有多少个输出的列将被读取多少个列。其他的将被忽略。

为了保持清晰, 你也可以是这样的语句。对于两个的语句请看下一节:

```
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（叫做多行、一个语句执行）：

```
INSERT INTO products (product_no, name, price)
SELECT product_no, name, price FROM new_products
WHERE release_date = '2025-05-29';
```

这运行了两个语句：插入语句和SELECT语句（从新表）的全集。

## 提示

在一次插入大量数据时，考虑使用 COPY命令。它比 INSERT命令效率更高。

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组合操作，交、并、差、并行组合、连接

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

`query1 query2`组合可以使用以上所有操作的查询，最后一个可以嵌套和嵌套，例如：

```
query1 UNION query2 UNION query3
```

实践自己的查询：

```
(query1 UNION query2) UNION query3
```

注意：有使用到“query2”的结果对“query1”的结果上（不过执行时保证它是进行实际必要的操作）。此外，它限制结果中所有重要的行，需要 DISTINCT 命令的限制，除非你使用了‘UNION ALL’。

INTERSECT 操作把两个存在“query1 query2”结果中的行，组合成一个“INTERSECT ALL”。否则不会有行被返回。

EXCEPT 操作存在“query1 query2”结果中但是不在“query2”的结果中（它的限制叫“差的”）。因此，除非使用“EXCEPT ALL”，否则不会有行被返回。

为了计算两个查询的交、并、差，这两个查询必须是“操作兼容的”。也就是说它们必须返回相同的列，并且对这些列有相同的数据类型。就像 11.3.1 节描述的那样。

并行查询

并行查询如何工作

当把处理器对准于某个特定的查询，并行查询是最快的执行策略时，优先执行一个“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
```



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

可选关键词。它们没有效果。

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。

## 例子

中止所有更改:

兼容性

BEGIN [ WORK | TRANSACTION ] [ transaction mode [ ... ] ]

其由 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`开始一个新的事务，也就是说所有 `BEGIN` 语句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`），MySQL 以“自动提交”模式执行事务，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的隐式地执行一次提交（如果执行成功，否则会完成一次回滚）。

在一个复杂的环境中必须进行选择，因为太多的开始，增加选择的范围会降低选择的效率。在进行多个选择时，选择的复杂性是选择的效率的决定性因素。在所有相关的决策还没有完成之前，其他决策将不断地看到新的选择。

如果指定了 `隔离级别`，语句便只对指定的级别。新事务也会有指定级别。新事务执行了 `SET TRANSACTION` 语句。

## 参数

`WORK TRANSACTION`

可选的关键词。它们没有效果。

这个语句和其他参数的意义请参考 `SET TRANSACTION`。

## 注解

`START TRANSACTION` 和 `BEGIN` 语句的效果相同。

使用 `COMMIT` 或者 `ROLLBACK` 立即做一个事务。

自己可以在事务块内使用 `BEGIN` 语句做一个警告消息。事务块不会被取消。要在在一个事务块中做事务，可以使用保存点（见 `SAVEPOINT`）。

关于语句的讨论。有关的 `transaction_mode` 语句的语义可以看我。

## 示例

开始一个事务：

```
BEGIN;
```

## 兼容性

`BEGIN` 是 MySQL 语句扩展。它类似于 SQL 标准的命令 `START TRANSACTION`，它的语义也有部分的兼容性信息。

`DEFERRABLE transaction_mode` 是 MySQL 语句扩展。

别忘了，`BEGIN` 只能出现在输入到 MySQL 中的一种大括号内。在你想要使用它时，你必须小心地将事务语句。

## COMMIT — 提交当前事务

## 大纲

`COMMIT` 提交当前事务。所有嵌套事务的更改会变得对其他人可见并被保留在新事务的连接上。

## 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`，则它会执行刚刚完成的事务并和新事务（也是 `SET TRANSACTION`）的新事务。新的，没有事务被启动。

## 注解

使用 `ROLLBACK` 中做一个事务。

当在一个事务内时使用 `COMMIT` 不会产生杰克。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

要是文档的事务并让所有更改持久化：

```
COMMIT;
```

## 兼容性

命令 `COMMIT` 有 SQL 标准。但是 `COMMIT TRANSACTION` 是 MySQL 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 提交一个为一个准备好的事务。

## 参数

transaction\_id  
要提交的事务的事务标识符。

## 注解

要提交一个准备好的事务，你必须首先执行该事务的同一账户或者超级用户，但是不需要为执行该事务的同一会话。

这个命令不能在一个事务块中执行。该准备事务将被立刻提交。

RE\_PREPARED\_TRANSACTION 为相同的会话中所有准备好的事务同时提交。

## 例子

提交准备好的 'foobar' 事务：

```
COMMIT PREPARED 'foobar';
```

## 兼容性

COMMIT PREPARED 是一个MySQL 5.7 版本，其兼容性为 5.7 之前所有兼容的。其中包含旧的冲突隔离（例如 Open AI），也是从 MySQL 5.7 版本开始引入的。

## END - 提交当前事务

## 大纲

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

## 描述

END 定义当前事务。所有在当前的语句将被对其他人可见并被提交至数据库的然后持久的。这个命令是 MySQL 5.7 版本，它类似于 COMMIT。

## 参数

WORK TRANSACTION  
可选关键词，它们没有效果。

AND CHAIN  
如果指定了 AND CHAIN，并且当前会话的事务具有相同事务标识符，那么 SET TRANSACTION 语句将为新的事务有效。否则，没有新事务的语句。

## 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 没有意义，但会发出一个警告消息。

## 示例

提交当前事务并自此点后更改行为：

```
END;
```

## 兼容性

END 是一个 MySQL 5.7 版本，它兼容 COMMIT 语句的语义。但是它没有事务隔离。

## PREPARE TRANSACTION – 为两阶段提交准备当前事务

## 大纲

```
PREPARE TRANSACTION transaction_id
```



## 兼容性

命令 ROLLBACK PREPARED 为 MySQL 特有。命令 ROLLBACK TRANSACTION 是一个 MySQL 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` 回滚一个为准备好的两阶段提交的事务。

### 参数

`transaction_id`  
要回滚的事务的事务号。

### 注解

要回滚一个准备好的事务，必须是尚未执行事务的同一个操作或者是一个操作串，但是你必须在执行事务的同一个会话中。

这个命令是在一个事务中执行的。准备好的事务不能被立刻回滚。

`pt-prepared-status` 命令输出中列出了所有可能的带有准备好的事务。

### 例子

准备好的命令 foobar 回滚对应的事务：

```
ROLLBACK PREPARED 'foobar';
```

## 兼容性

`ROLLBACK PREPARED` 是一个 MySQL 扩展。此原因是由于个别事物被限制的，其中自己已经执行某些（例如 `Open AL`），但是那些事物的SQL语句被限制。

## 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。

保存点是事务的一个特殊标记。它为所有在它被建立之后执行的命令的记录。因此事务的状态将受到它关于保存点的样子。

### 参数

`savepoint_name`  
新的保存点的名字。

### 注解

使用 `ROLLBACK TO` 命令到一个保存点。另外 `RELEASE SAVEPOINT` 则回滚一个保存点，但保存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中建立。可以在一个事务中定义多个保存点。

### 示例

要建立一个保存点并将其与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 1 和 3，但不会插入 2。

要建立并开始一个保存点：

```

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

当建立另一个新的保存点时，SQL 要求之前的那个保存点必须被释放。在 MySQL 中，除非保存点在之前被释放，不过在进行回滚或释放时也需要释放的前一个（即“RELEASE SAVEPOINT”和它的前一个保存点的语句之后再使用“ROLLBACK TO SAVEPOINT”或“RELEASE SAVEPOINT”语句）。在其他方面，“SAVEPOINT”完全支持 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”设置当前事务的约束检查时机。“IMMEDIATE”约束在每个语句执行时被检查，“DEFERRED”约束直到事务提交时被检查。每个约束都有自己的“IMMEDIATE”或“DEFERRED”模式。

在创建时，一个约束被指定为“DEFERRED”、“DEFERRED INITIALLY DEFERRED”、“DEFERRED INITIALLY IMMEDIATE”或“NOT DEFERRED”。第二类是“IMMEDIATE”，对于不能使用“SET CONSTRAINTS”命令的限制，限制在每个事务开始时都才指定的模式。但是它们的行为可以在一个事务中用“SET CONSTRAINTS”更改。

当启用“DEFERRED”或“DEFERRED INITIALLY DEFERRED”模式时，如果在语句中使用“ON UPDATE”或“ON DELETE”子句，那么在语句执行时将忽略它们。如果在语句中使用“ON UPDATE”或“ON DELETE”子句，那么在语句执行时将忽略它们。

当启用“IMMEDIATE”或“DEFERRED INITIALLY IMMEDIATE”模式时，如果在语句中使用“ON UPDATE”或“ON DELETE”子句，那么在语句执行时将忽略它们。如果在语句中使用“ON UPDATE”或“ON DELETE”子句，那么在语句执行时将忽略它们。

当启用“NOT DEFERRED”时，如果在语句中使用“ON UPDATE”或“ON DELETE”子句，那么在语句执行时将忽略它们。如果在语句中使用“ON UPDATE”或“ON DELETE”子句，那么在语句执行时将忽略它们。

如果启用了“FOREIGN KEY”或“ENCLOSE”语句，那么将忽略这个设置的控制。它们将在相关的语句执行时被忽略。

### 注解

在 MySQL 中，不要求的约束将被忽略（但是在内部要遵守）。可是有多个一个约束的限制是必须遵守的约束。在启动语句“SET CONSTRAINTS”命令在所有的语句上操作，对于一个单独语句的约束，一旦在要操作语句中的某个模式冲突于一个或者多个语句，那么所有的语句将不会被接受。

这个命令将忽略当前事务的约束行为。在事务操作前发出这个命令将产生一个警告并且也不会有任何效果。

## 兼容性

这个命令在 MySQL 和其他数据库的行为有一些区别。在 MySQL 中，它不能应用“NOT NULL”和“CHECK”约束上。但是，MySQL 支持的约束可能比其他的唯一，而不是限制在表建立时的语句。

## 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'设置一个会话后端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。

应用的基本特性是基本的继承机制、类型化语义模式（值/函数/对象）以及线程局部模式。此外，可以选择一个线程，通过只读操作读取基本的继承操作或全局操作。

一个复杂的顺序微弱地暗示其他复杂的平行执行时该事务的哪些阶段什么数据

READ COMMITTED

当前事务的所有语句只能看到这个事务中执行的第一个

## SERIALIZABLE

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'。

会话默认的事务模式也可以通过配置参数 `default_transaction_isolation`。

当前事务的模式可以类似的通过配置参数 `transaction_isolation`、`transa`

## 示例

要用一个已经存在的事务的回滚点开始一个新的事务，就必须从这个原有事务得出结论。这样会关闭这个事务，它的

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
pg_export_snapshot
-----
00000003-00000001B-1
(1 row)
```

然后在一个新开始的事务的开头把读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MySQLSQL扩展。

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，新的事务将具有这些特性，就像执行了 `SET TRANSACTION` 一样。这和 `RECORD` 令一样。

## 参数

这些参数对于这个语句的含义可参考 [SET TRANSACTION](#)

第五章

SQL参考 (第4章 SQL语法)

## 词法结构

```
SELECT * FROM MY_TABLE;  
UPDATE MY_TABLE SET A = 5;  
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

## 标识符和关键词

UPDATE MY TABLE SET A = 5;

μPDArTE my Table Set a = 5:

一个常见的问题是将双引号写成大写，从而导致小写，例如：

```
UPDATE my_table SET a = 5;
```

这里也有第二种形式的引号：“”（单引号）或“”（双引号）。它们是区分大小写的，指的是字符串（“”）括起来的任意字符序列。一个引用的字符串是一个有效的字符，因此“value”可以被字符串一个名为“value”的对象替换。而一个没有字符串的“value”将被当作一个关键词，从而在文本中被替换为它的含义（例如：在上例中被替换为5的值）：

```
UPDATE "my_table" SET "a" = 5;
```

引用的字符串可以包含任何字符，除了转义的字符（如果要包含一个转义符，那么两个转义符）。这使得可以构造出不为合法的表达式的名称，例如包含或不含引号的数字，也是十分常见的有效。

引号的种类也可以区分大小写，单引号的字符串名是单撇号（'），双引号的字符串名是双撇号（"）或反引号（`）。在MySQL中，将字符串的长度设置为大小写敏感并不需要，这样单引号不需要引号的长度设置为大写。因此，根据标准，"foo" 等于 "FOO" 而不是 "foo"，如果想要编写代码时使用，建议统一使用单撇号（'）。

一种常见的字符串类型是二进制类型的字符串的字节串（byte串）。这可以通过以 '0x'（大写的0x）后跟一个十六进制数开始。后跟的十六进制数的长度，再跟一个逗号，如 '0x1a'（注意这里没有空格）或 '0x1a,0x1a'（注意这里有两个逗号，也是在大写的0x后面跟一个逗号）。在引号内，byte串可以以二进制形式指定，这将阻止MySQL将字符串转换为长字符串或在字符串上执行转换操作的水平线。例如，转义的 'data' 可以写成：

```
U&"d\0061t\+000061"
```

下面的例子将插入字符串写出了转义的 "data"。（注意：`）

```
U&"\0441\043B\043E\043D"
```

如果希望使用其他字符串表示法，可以在字符串前使用 UESCAPE 字符，例如：

```
U&"d!0061t!+000061" UESCAPE '!'
```

转义字符串是除了注释符、注释、单引号、双引号、反引号、空字符串之外的任意字符串。请注意，转义字符串在 UESCAPE 之后将被自动转义为转义字符串。

万能转义字符包括字符串本身，转义为转义字符。

MySQL的转义字符都可以被用来定义UTF-8时对多字节的字符串进行转义。字符串形式的转义字符在使用这种转义字符串时，必须对字符串的每一个字节进行转义，而不是对字符串本身进行转义。如果不可见，转义成功。

## 常量

MySQL中有两种常量类型：字符串、字符串字节、字符串字节，字符串也可以被指定为类型。还可以使用字符串转义以及带有转义的字符串。这些字符串在后两个章节中讨论。

## 字符串常量

在MySQL，一个字符串常量是一个由单引号（'）包围的任意字符串。例如 "This is a string"。为了在一个字符串中包含一个单引号，可以写两个相连的单引号，例如 "Diane's home"。注意第一个单引号（'）不同。

两个单引号以及至少一个单引号的字符串的单引号会将单引号一起，并且将作为一个写在一起的字符串常量对待。例如：

```
SELECT 'foo'  
'bar';
```

结果：

```
SELECT 'foobar';
```

也就是说：

```
SELECT 'foo'      'bar';
```

## C风格转义的字符串常量

并不是所有的语句（这与古老的语句行为是SQL驱动的）。MySQL遵循了这些规则。

表4.1. 反斜线转义字符

跟隨在一個反斜線後面的任何其他字彙被當做其字面意思。因此，要包括一個反斜線字彙，請寫兩個反斜線 (\textbackslash\textbackslash)。在一個轉文字字符串中包括一個單引號除了普通方法' '之外，還可以寫成 ``'。

你要负责保证你创建的字符串序列或服务名等为基础的字符串的合法性。特别是在使用八进制或十六进制时。一个有效的替代方法是使用Unicode转义或替代的Unicode转义语法，如 4.1.2.3 节中所述；然后服务器将检查字符串是否可行。

111

如果想要使用 `student.student_name` 作为参数，那么在SQL语句中使用的是参数的别名，即在SQL语句中的参数为 `#{u}`。要想自己使用参数的别名，可以在 `NamedParameterJdbcTemplate` 上面加上 `mappedBy="参数的别名"`，这样在使用参数时就可以使用参数的别名了。如下所示：

带有 Unicode 转义的字符串常量

U&'d\0061t\+000061'

下面的例子用斯拉夫字母写出了俄语的单词“son”（大家）：

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'UNESCAPE'子句来指定。例如：

转文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。

要在每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。

<sup>10</sup> 三引用的字句中當是

\$\$Dianne's horse\$\$  
\$SomeTag\$Dianne's horse\$SomeTag\$

注意在美元引号字符串中，单引号可以在不被转义的情况下使用。事实上，在一个美元引号字符串中不需要对字符串进行转义：字符串内容是按其字面意思写出。反斜杠不是特殊的，并且美元符号也不是特殊的，除非它们是嵌配在单引号的一个序列的一部分。

可以通过在每一个参数级别上选择不同的标签来嵌套美元引用字符串常量。这通常被用在编写函数定义上。例如

```
$function$  
BEGIN  
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);  
END;  
$function$
```

## 位串常量

位串常量表示为字符串，并以位（二进制）或字节（十六进制）的形式表示。例如，字符串“00010110”加了一个“0”（十六进制表示法），则即为“00010110”。位串常量在字符串的字节间“0”和“1”。

作为一种选择，位串常量可以用十六进制符号表示，使用一个前导“0”（十六进制表示法）的“00010110”。这种记号法每行一个十六进制数和两个十六进制位的位串常量。

位串常量可以以位串常量或字节常量的方式进行转换。美元符号不能使用在位串常量中。

## 数字常量

在这一节中你可以阅读数字常量：

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

其中 **digits** 是一个或多个进制数字（即 0-9），如果使用了小数点，在小数点前面和后面必须至少有一个数字。如果存在一个后缀物化 **e**，在数值后面需要至少一个数字。在数值中不能插入任何其他字符。注意任何前导或尾随字符并不影响常量的值，它是一个任何前缀或后缀的算符。

这是显示数字常量的例子：

```
42
3.5
4.
.001
5e2
1.925e-3
```

如果一个包含小数点和进制的数字常量的值从类型 **integer**（32 位），它首先被假定为类型 **integer**，否则如果它的值低于类型 **integer**（34 位），它被假定为类型 **integer**，否则则它被假定为类型 **numeric**。包含小数点和进制的常量总是首先被假定为类型 **numeric**。

一个数字常量的假定的数据类型只是类型推导过程的一个开始。在大部分情况下，常量将根据上面的自动地推导的最合适的类型。必要时，可以通过类型注释或强制转换为一种指定的数据类型。例如，你可以这样强制一个数字使用类型 **real**： **float**：

```
REAL '1.23' -- string style
1.23::REAL -- IvorySQL (historical) style
```

这在某种程度上只遵循了本来应付的一种类型记号的特性。

## 其他类型的常量

一种任意类型的常量可以使用以下任何一种的任意一种插入：

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

字符串常量的文本被识别为 **type** 的类型的插入转换的驱动。其他类型没有类型的一个常量。如果对字符串的类型没有插入（例如，当它被假定为一个字符串）。显示类型也可以被忽略，在这种情况下它会自动地推导。

位串常量可以使用识别为十六进制的字符串。

也可以使用一个类型名的语法来指定一个类型转换：

```
typename ( 'string' )
```

但是并非所有类型名都可以所在这种方式中，详见 [4.2.5](#)。

如果 [4.2.5](#) 中的任何语句，**1. CAST** 以及语句语义也可以使用类型转换表达式进行类型转换。要读取语句语义，**type 'string'** 只要使用类型转换表达式类型的类型。**type 'string'** 通常分为一个限制符以及对转换类型的工作，如果一个数据类型的类型使用 **1** 或 **CAST**。

**CAST** 语法规则如下。**type 'string'** 语法规则的唯一例外：**SQL** 语句语义只有一种数据类型，但是 **IvorySQL** 允许所有类型。**type** 的语义是 **IvorySQL** 的语义，但是语义的语义是一致的。

## 操作符

一个操作符名是最多 `NAMEDATALEN`-1 (默认为4) 的一个字符串，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“`*/`”不能在一个操作符的任何地方出现，因为它们将被作为一段注释的开始。

· 一个字符串操作符名不能以“或“结尾，除非该名称也至少包含这些字符串中的一个：

例如，`||-`是一个被允许的操作符名，但`||`不是。这些限制允许MySQL解析SQL兼容的查询而不需要在运算符之间有空格。

当使用简单SQL标准的操作符名时，你通常需要使用空格分隔前缀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MySQL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。



## 注释

一段注释是以双撇号开始并且延伸到行结尾的一个字符串，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 */
 */
```

项目类江程类类 / **且纸像到匹配类的/** 项目类程是成功项目标准中重要的方式衡量。但和口头不同，这样我们项目的江程就一个对项目类江程的评价。

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 展示了 MySQL 平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在全局的。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指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. 操作物优劣顺序 (从高到低)

```
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

如果在之前的 SELECT 语句中使用了 WHERE 子句，那么在使用 BETWEEN 时，必须在 WHERE 子句之后，不能在 WHERE 子句之前。不过有一些数据库，比如 MySQL，可以在 WHERE 子句之前使用 BETWEEN 语句。

## 值表达式

他表达式被用于各种各样的操作中，例如在“SELECT”语句的目标列表中、作为“INSERT”或“UPDATE”语句中的新值或者作为“WHERE”中的搜索条件。为了避免一个表达式（是一个表）的结果，一个值表达式的结果有时候被称为一个“常量”。表达式因此也被称为“常量表达式”（或者甚至是简称“表达式”）。表达式语句允许使用算术、逻辑、算术和其他操作从原始部件计算

一个值表达式是下列之一

- 一个字符使用单引号
  - 一个空引号
  - 在“一个”函数定义或变量名中使用一个的单引号或双引号
  - 一个字符串或双引号
  - 一个连接字符串
  - 一个字符串使用单引号
  - 一个双引号使用双引号
  - 一个单引号使用单引号
  - 一个单引号使用双引号
  - 一个双引号使用单引号
  - 一个单引号使用单引号
  - 一个双引号使用双引号
  - 一个单引号使用双引号
  - 一个双引号使用单引号
  - 一个字符串或双引号
  - 在“一个”函数定义或变量名中使用一个的单引号或双引号

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物的语义并且在第 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 节中讨论过变量。下面的小节会讨论剩下的选项。

correlation.columnname

`correlation` 是一个表（有可能是以一个模式名取的）的名字，或者是在 'FROM' 子句中为一个表定义的别名。如果别名在当前查询所使用的表中都是唯一的，不带别称和什麼样的操作可以被使用（参见第 7 章）。

### 位置参数

一个位置参数引用被用来指示一个在SQL语句外都提供的值。参数被用于SQL函数定义和准备查询中。某些客户端端还支持独立于SQL命令字符串来指定数据值。在这种情况下参数被用来引用那些在线外数据值。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。

丁士一

expression[*subscript*]

## expression[lower\_subscript:upper\_subscript]

(注意, 方括号 [ ] 表示真字符串)。每一个 **下标** 表示着一个子表达式。它的值是插入表达式中的整数。

通常, 那些 **表达式** 必须加上括号, 但是当表达式中的表达式只是一个引用时它需要括号。括号可以是空的, 还有, 当表达式是函数时, 多个 () 都可以被忽略掉。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要的。详见 [第 10 章](#)。

## 域选择

如果一个表达式得到一个组合类型 (行类型) 的值, 那么可以使用选择语句的域:

## expression.fieldname

通常一个表达式得到一个组合类型 (行类型) 的值, 那么可以使用选择语句的域:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(注意, 一个需要的语句实际上只是表达式表达式的一种特例)。一种重要的特例是另一个组合类型的表达式 (例如最后一个域):

```
(compositecol).somefield
(mytable.compositecol).somefield
```

这里需要的语句是 `compositecol` 是一个表达式, 而第二维是 `mytable` 是一个表达式, 所以一个表达式。

也可以通过写 `*` 来请求一个组合类型的所有域:

```
(compositecol).*
```

这种方式的行为根据上下文会有不同, 详见 [第 10 章](#)。

## 操作符调用

对于一次操作符调用, 有两种可能的表达式:

expression operator expression (二元操作符表达式)
operator expression (一元操作符表达式)

其中 `operator` 必须遵循 [第 4.1.3 节](#) 的语法规则, 或者是逻辑门 AND, OR 和 NOT 之一, 或者是一个如下形式的宏表达式操作符:

```
OPERATOR(schema.operatorname)
```

那个特定操作符将在以及它们是一次的或是一次的语句或宏表达式中被使用。

## 函数调用

一个函数调用的表达式是一个函数的名称 (可能包含一个模式名) 后面跟上一个或多个圆括号中的参数列表:

```
function_name ([expression [, expression ... ] ] )
```

$\sqrt{2}$

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 20.3 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

一个实现第一组合型参数的函数可以被选择地称为域选择函数，并且反过来域选择可以被看成函数的风格。也就是说，记号“col

## 聚集表达式

一个聚集表达式表示在同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` 是任意本身不包含聚集表达式的表达式或一个窗口函数调用. 可选的 `order_by_clause` 和 `filter_clause` 请见下文.

第一种形式的参数表达式为每一个输入语句有一次聚集，第二种形式和第一种相同，因为 **ALL** 是缺省选择项。第二种形式为输入行表达式的每一个可区位值（或者对于多个表达式是他的可区分集合）调用一次聚集。第四种形式为每一个输入语句一次聚集，因为没有使用对输入值被指定，它通常只对 **count** 算法函数有效。最后一种形式被用于“**所有**”聚集函数，其结果如下：

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，对于所有内部聚集都是这样。

例如, `count(*)` 得到插入行的总数。 `count(f1)` 得到插入行中 `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。

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

在处理多参数聚合函数时，注意 **ORDER BY** 出现在所有聚合参数之后。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
```

后者在逻辑上是正确的，但它表示用两个 ORDER BY 做来调用一个单一步排序函数（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 `DISTINCT` 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

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
percentile_cont
-----
50489
```

这条语句 `households` 的 `income` 列得的第 50 个百分位数是 50489。这里 50 是一个直接参数，对于百分位数来说是一个在不同的语义下完全不同的值，所以它没有意义。

如果指定了 `FILTER`，那么只有对 `filter_clause` 计算为真的输入行会进行聚集函数，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)
```

相关的聚集函数在 [8.2.1](#) 中描述。其他聚集函数可以在 [用户指南](#)。

一个聚集表达式只能出现在 `SELECT` 语句的结果列表或 `WITHINGROUP` 子句中。在其他子句（`WHERE`）中禁止使用它，因为聚合子句的计算在逻辑上是在聚集的结果被形成之后。

当一个聚集表达式出现在一个子查询中（见 [8.2.1.2](#) 和 [8.2.3.7](#)），聚集通常在子查询的行上被计算。但是如果聚集的参数（以及 `filter_clause`，如果有）只包含外层查询的子查询（即 `WITHINGROUP` 子句）的行，那么聚集表达式将只在子查询的行上被计算。当子查询的参数（以及 `filter_clause`，如果有）只包含外层查询的行时，那么聚集表达式将只在子查询的行上被计算。只包含外层查询的参数（以及 `filter_clause`，如果有）的聚集表达式将只在子查询的行上被计算。

## 窗口函数调用

一个窗口函数调用在一个查询语句的某个子句上应用一个聚集类的函数，和普通的聚集函数不同，这个子句的表达式被选择的行作为一个单一的输出。在普通的表达式中，一个窗口函数调用的表达式将被选择的行作为一个单一的输出。不过，窗口函数调用的表达式将被选择的行作为一个单一的输出。一个窗口函数调用的表达式将被选择的行作为一个单一的输出。

```
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` 的语法规则

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...
] ]
[ frame_clause ]
```

相关的 `frame_clause` 语法规则

```
{ 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“绝对定位”的一种

UNBOUNDED PRECEDING



## 排序规则表达式

COLLATE 子句会重置一个表达式的排序规则。它将应用到它之后的表达式。如果表达式之后没有排序规则，那么它可以使用原排序。

```
expr COLLATE collation
```

这里 collation 可能是一个字符式常量或表达式。COLLATE 子句会重置排序规则。如果可以使用原排序，那么可以使用原排序。

如果没有显式指定排序规则，数据将从表或内联视图的列中得到一个排序规则。如果表达式之后没有排序，则数据以采用数据库的默认排序。

COLLATE 子句的两种常见使用场景是使用 ORDER BY 子句对结果排序。例如：

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

以为要对所有列应用排序规则，所以对所有列都应用排序规则。例如：

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

注意在另一种情况下，COLLATE 子句将只对表达式或函数的输出值起作用。COLLATE 子句对表达式的结果或函数的输出值上起作用，因为排序操作只对表达式或函数的输出值起作用，而不是对表达式或函数的输入值起作用。并且一个普通的 COLLATE 子句将重置所有其他表达式的排序规则（不过，别名中的 COLLATE 子句将多个一个表达式一种排序，详见 [别名](#)）。因此，这输出和前一个例子相同的结果。

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

也就是说同一个结果。

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

因为它尝试将一个排序规则应用到一个表达式的结果，而它的数据类型是单列排序的别名。

## 标量子查询

一个标量子查询是一种嵌套在另一个 SELECT 语句内的普通 SELECT 语句。它同时返回一行或一列（字符串常量或别名 [子查询](#)）。"SELECT" 语句将执行并返回与一个标量子查询使用相同的表达式文本。对一个语句返回一行或一列的表达式为一个标量子查询使用一种语法（但是如果在一次对字符串操作语句或字符串操作语句之后使用一个逗号，语句将从以逗号之后的字符第一次计算表达式文本，对其他语句子查询的表达式文本将从第 222 行）。

例如，下列语句将返回两个字符串最大的插入人口：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
  FROM states;
```

## 数组构造器

数组构造器是一种特殊并只使用它的或其参数的表达式。一个典型的数组构造器语句是不被 [ARRAY](#)、一个方括号 [ ]、一个带有斜线的表达式或逗号（[数组分隔](#)）以及最后一个右方括号 ] 组成。例如：

```
SELECT ARRAY[1,2,3+4];
array
```

```
-----
{1,2,7}
(1 row)
```

默认情况下，数组元素类型是与表达式类型的以类型。使用和 [DECODE](#) 或 [CASE](#) 语句（[见 215 页](#)）帮助控制类型。也可以通过显式转换构造器类型为必要的类型来重写。例如：

```
SELECT ARRAY[1,2,22.7]::integer[];
array
```

```
-----
{1,2,23}
```

(1 row)

你可以从一个表达式或表达式块中返回一个或多个类型的对象。这个表达式必须返回一个或多个行。

```
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嵌套的，所以同一级别的内部数组必须产生相同类型的子数组。也就是说两个是 `ARRAY` 语句的嵌套的或嵌套的内部的内部语句。

多维数组语句必须以适当的层级嵌套。内部的语句必须是 `ARRAY` 语句。例如，这将返回一个子-`ARRAY` 行。

```
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，但是因为无法再转一个无类型的数组。所以是无法构造的空数组或空类型的数组。例如：

```
SELECT ARRAY[]::integer[];  
array
```

```
-----  
{}  
(1 row)
```

你可以从一个子查询的结果构造一个数组。在这种情况下，影响构造语句为子查询 `ARRAY` 语句的一个或多个括号（大括号）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。如果子查询的结果是多列类型，结果的一列数据将为子查询结果中的每一行的一个元素。并且有一个与子查询的结果列数对应的数据类型。如果子查询的结果列数与一种的数据类型，结果将对应类型的一个数组。但是要有一个维度。在这种情况下，子查询的列数必须产生同样维度的数据。否则结果将产生错误类型。

有关 `array` 语句的一个常见错误是以为它是 `array`。更多关于数组的信息，请参见 [数组](#)。

## 行构造器

一个行构造器将被转换为一个行值（也称一个纪录类型）并被当作其后跟随的表达式。一个行构造器由逗号 `,` 一位圆括号，两个可选的逗号等分多个表达式（通过分号）以及最后一个右括号组成。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，不能使用 `ROW` 是可选的。

一个行构造器可以包括语句 `rowstar`。`*` 它提供了等于进行值的参数的一个列表。通常在一个语句 `SELECT` 表达式（见 [8.3.1.1行](#)）中使用。`*` 行表示的事情一样。例如，如果 `t` 有行 `t.f1`，那么这是正确的：

```
SELECT ROW(t.* , 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

## 注意

在 PostgreSQL 8.2 以前，`*` 表达式不会在行构造器中被扩展。这将与 `ROW(t.* , 42)` 语句是一个有两个值的行，其第一个值是另一个行值。好的行为通常更直观。如果这是你需要行构造器的默认行为，写成 `ROW(t, 42)`。

```
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(ROW(11,'this is a test',2.5) AS myrowtype));
```

```
getf1
```

```
-----  
11
```

```
(1 row)
```

行构造器可以使用其他操作符在一个表达式类型转换的语句。或者使用一个操作符类型的函数。还有，可以以任何行值，或使用 `IS NULL` 或 `IS NOT NULL` 的一行。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
```

语句 1.24 页，第 5.2.5 项的讨论。行构造器可以被用来与子查询语句。

## 表达式计算规则

子表达式计算操作符没有语义。换句话说，一个操作符或函数的输入不必依赖从它们和其他任何类型的表达式计算。

此外，如果一个表达式的结果可以通过计算其一部分来决定，那么就将子表达式可能完全不需要计算。例如，如果它们：

```
SELECT true OR somefunc();
```

那么 `somefunc()` 将（可能）完全不被调用。除非我们写出这样这样也是可能：

```
SELECT somefunc() OR true;
```

注意这和一些编程语言中的操作符从左至右的“短路”不同。

因此，在某些表达式不能使用操作符的语义是不可取的。在 `WHERE` 或 `SELECT` 子句中的操作符或计算将无其效。因为在建立一个行计划时这些子表达式必须被“先处理后读”。这些子表达式将永远（`AND` 或 `OR` 的结果）根据以它们为判定操作的行方而被重写。

当然必要操作的重写时，可以使用一个 `CASE` 语句（见 3.3.4 项）。例如，在一个 `WHERE` 子句中操作子表达式可能被重写是不可取的：

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

也就是说某些操作：

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

一个广泛使用操作的 `CASE` 语句的使用化简表达式。因此只关心操作才很重要（在适当的例子中，最好编写 `y > 1.5` 来回避这个问题）。

不过，`CASE` 不是完美的工具。上技术的一个限制。它无法阻止操作子表达式为短路计算。如果 SQL 语句中表达式，当表达式被操作时不是执行时，被表达式 `DISCARDED` 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;
```

是可能得失败的。因为对操作表达式化简表达式方式。这是是表达式中可能有一个操作 `x > 0`（这样操作时永远不会进入的 `DISCARD` 分支）也是正确的。

虽然这个特别的例子可能看起来，没有问题是将表达式放在函数内部的原因，因为因为函数操作的值和操作子表达式作为为共享，被插入到表达式中是可能的。例如，在 PostgreSQL 中，使用一个 `DISCARD` 表达式和另一种有趣的计算结合起来——`CASE` 表达式要完全符合。

另一个潜在的限制。一个 `CASE` 表达式上操作表达式的结果的计算。因为在考虑 `SELECT` 语句或 `MINUS` 子句中的其他表达式时，表达式被重写。例如，下例的表达式被一个操作错误。虽然看起来操作可能可以使用：

```

SELECT CASE WHEN min(employees) > 0
    THEN avg(expenses / employees)
END
FROM departments;

```

MySQL 和 PostgreSQL 都会自动将输入转换为计算。因此如果运行代码 `avg(expenses * 0)`，在有机关键字 `min()` 的时候之后，就会发生错误。为了避免这种情况，可以在一个 `MINUS` 或 `FLUSH` 语句之前将所有的输入行放进一个聚集函数。

## 调用函数

MySQL 和 PostgreSQL 都会自动将输入转换为计算。因此如果运行代码 `avg(expenses * 0)`，在有机关键字 `min()` 的时候之后，就会发生错误。为了避免这种情况，可以在一个 `MINUS` 或 `FLUSH` 语句之前将所有的输入行放进一个聚集函数。

在想要一种字符串连接，在 PostgreSQL 中给出了默认的字符串连接符不需要在语句中写出来。但是这在 MySQL 中是不行的，因为字符串连接符必须写出来。而在 PostgreSQL 中字符串连接符的使用是自动的。

MySQL 也支持字符串连接符。它组合了字符串和字符串连接符。在这种情况中，字符串连接符必须写出来并且字符串连接符。

以下的例子展示了所有三种字符串连接的方法：

```

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;

```

在语句 `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`，此时让它为 `false`，`a` 和 `b` 会输入字符串，并且将 `uppercase` 参数强制为大写或小写方式。这个函数的剩余部分对设置并不需要（详见第 30 章）。

## 使用位置记号

在 MySQL 中，位置记号法是给函数作参数的特别机制。一个例子：

```

SELECT concat_lower_or_upper('Hello', 'World', true);
concat_lower_or_upper
-----
HELLO WORLD
(1 row)

```

所有参数使用同样的格式，结果是大写形式，因为 `uppercase` 参数设为 `true`。另一个例子：

```

SELECT concat_lower_or_upper('Hello', 'World');
concat_lower_or_upper
-----
hello world
(1 row)

```

注意，`uppercase` 参数被忽略，因此它被设为它的默认值 `false`，并导致小写形式输出。在位置记号法中，参数可以使用从右向左的逻辑并从左向右的逻辑。

## 使用命名记号

在命名记号语句中，每一个参数名都必须与对应的参数值完全相同。例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');  
concat_lower_or_upper
```

```
-----  
hello world  
(1 row)
```

再如，参数 `uppercase` 是可选的，因此它被设为设置为 `false`。使用命名记号语句的一个好处是参数可以相对地动态化。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，参数 `>` 的旧语法仍然支持。

```
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

混合记号语句结合了位置和命名记号。不过，它们已经识别过。也就是说一个参数既可以用命名记号，也可以用位置记号。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);  
concat_lower_or_upper
```

```
-----  
HELLO WORLD  
(1 row)
```

在上述语句中，参数 `a` 和 `b` 既以位置指定，又 `uppercase` 通过名字指定。在这个例子中，只可设置了一个文件。在一个具有大量参数且参数的类型混杂的语句中，命名的混合记号语句可以节省大量的时间和减少出错的机率。

## 注意

命名的混合记号语句只适用于在 MySQL 5.7 及以上版本时使用。（但是与新版本的 MySQL 5.7 及以上版本时可以使用）。

# Oracle兼容功能

修改

\* Oracle兼容([https://docs.oracle.org/cd/E11876\\_01/server.112/e10592/02.html](https://docs.oracle.org/cd/E11876_01/server.112/e10592/02.html))

## 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 列的数据类型。  
expression 选择表达式。  
  
ADD keyword 增加新的列, 可以是单一个列或多个列。  
MODIFY keyword 修改现有的列, 可以修改一个列或多个列。  
DROP keyword 移除现有的列, 可以删除一个列或多个列。  
USING keyword 将表达式值

### 示例

```
ADD:  
create table tb_test1(id int, flg char(10));  
  
alter table tb_test1 add (name varchar);  
  
ALTER TABLE tb_test1  
  ADD address 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** 一个或多个列别名，它们可以在从句中使用，来从其他表的列。

**condition** 一个或多个列或表达式，将被 WHERE CURRENT OF 子句使用。

**cursor\_name** 将在 WHERE CURRENT OF 子句中使用的游标名。

**output\_expression** 一个行值表达式，将被 DELETE 语句从表中返回的表达式。

**output\_name** 将返回的表达式。

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 别名。  
column\_name 列名。  
expression 表达式。  
sub-SELECT 选择子句。  
from\_list 表达式。  
condition 一个或多个 Boolean 类型值的表达式。  
cursor\_name 是一个 WHERE CURRENT OF 子句中引用的游标名。  
output\_expression 为一个被引用的列名。必须与 SELECT 子句中引用的表达式相同。  
output\_name 对应引用的别名。

## 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 可以是SELECT语句、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE或FOR KEY SHARE子句SELECT子句。

ALL keyword 可以返回所有结果。

DISTINCT keyword 是从结果集中移除所有重复的行。

### 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

转义字符转义字符，即转义字符通常在SQL语句和函数的字符串之后使用，它们可以使用反斜杠(\)、撇号(')、逗号(,)、点(.)、A、B、#以及空格。

### 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	需要转换的日期

## 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数、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 ULLAALDPI 星期码

#### 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	指定的日期

#### 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	默认
date	oracle_date 类型, 需要格式为 DD/MM/YY
value	一个整数数据, 需要是32位的

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
src	需要转换的时区
dst	转换的时区
tz	时区ID

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	被转换的日期 (timestamp, timestampz, timestamp)
fmt	指定的格式, 如果被忽略, 则认为 "000"

## 例子

```
select trunc(timestamp '2020-07-28 19:16:12', 'Q');
       trunc
-----
2020-07-01 00:00:00
(1 row)

select trunc(timestampz '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 (timestamp, timestampz, timestamp)
fmt	指定的格式, 如果被忽略, 则认为 "000"

## 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 (value) 根据指定的格式将输入参数转换为 TO\_CHAR 数据类型的值。如果参数 value 为 NULL, 则返回的转换值为转换为 TO\_CHAR 值。如果 value 为 not null, 则返回值为 value。

## 参数

必填 输入参数 (必填类型)。

必填 输入输出参数, 并且类型为 `text`。

## 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 将输入的字符串转换为输入参数 or 时为 NUMBER 数据类型的值。如果参数 or 为数值或为字符串, 则将转换为 NUMBER。如果 or 是 NUMBER, 则返回该值。如果 or 是字符或空字符串, 则将它转换为 NUMBER。如果 or 为 NULL, 则返回空字符串。

## 参数

必填 输入参数 (必填类型)。

必填 输入输出参数, 并且类型为 `number`。

## 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函数将字符串转换为日期或时间类型的值。如果参数fmt为null, 日期将转换为系统的默认日期格式。如果fmt为null, 日期将被转换为DATE。如果fmt是L, 对于Adas, 将date转换为日期。

参数

str 输入参数: character, text, 可以是任何转换为上述类型的字符串, 除非由转换规则的字符串。

fmt 输入参数: 格式, 详见FORMAT。

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(*src*) 根据指定的格式将输入参数 *src* 转换为一个相对固定的时间。如果参数 *src*，的数据类型为字符串以格式中不同的区带的时间。如果 *src* 为 null，则该函数返回 null。如果无法转换为相对固定的时间，则该函数返回 null。

参数

必填 用于表示参数，详见参数说明。

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函数将输入参数作为时间间隔转换为半分钟或分钟的时间间隔。如果输入参数为半分钟，返回值为分钟。如果输入参数为分钟，返回值为分钟。如果输入参数为小时，返回值为分钟。如果输入参数为小时，返回值为分钟。

参数

输入参数 (int, 可以使用转换为文本类型, 必须是时间间隔类型, 等效 SQL 为表的 SQL 间隔格式, CO 为绝对时间格式为 02:00:12.04 秒数类型)。

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 将输入参数 as 的时间间隔转换为大数秒数的时时间隔。输入参数包括：日、时、分、秒和微妙。如果输入参数为 null，函数返回 null。如果输入参数包含分钟数或秒数，函数返回 null。

### 参数

str 输入参数 (text, 可以通过转换为文本类型)。必须是时间间隔格式。兼容 SQL 标准的 SQL 间隔格式。ISO 间隔时间格式为 ISO 8601 (2004 版本兼容)。

### 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) 根据指定的格式将输入参数 as 转换为带时区的时间戳。如果参数 tmt，将使用转换为具有时区以指定时区的时间戳。如果 as 为 null，转换函数返回 null。如果无法转换为带时区的时间戳，将返回 null。

### 参数

str 输入参数 (text, 可以通过转换为文本类型)。

mt 输入时区参数。详见 `getTz()`。

### 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,...) 会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, 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,...) 按照一个或多个表达式的结果值中的最小值。如果任何 expr 的计算结果为 NULL, 则返回值为 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)
HH04	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	纳秒 (000000-999999)
SSSS	午时 (分钟) (00-3000)
AM,pm,PM or pm	正午前/后 (午前或午后)
AM, a.m., PM or pm	正午前/后 (午前或午后)
U/YY	带区号的年 (四位或者更多位)
YY	年 (4位或者更多位)
MM	带区号的月
MM	带区号的月
Y	带区号的一位
YY	ISO 8601 范围带方括号的四位 (4位或者更多位)
YY	ISO 8601 范围带方括号的四位 (1位)
Y	ISO 8601 范围带方括号的最后一位
Y	ISO 8601 范围带方括号的最后一位
BC,bc,AD或卷ad	纪元前/后 (前或后)
BC,bc,AD或卷ad	纪元前/后 (前或后)
MONTH	全大写形式的月名 (空格前并带空格)
Month	全半形大写形式的月名 (空格前并带空格)
month	全半形大写形式的月名 (空格前并带空格)
MON	全大写形式的月名 (空格前并带空格)
Mon	全大写形式的月名 (空格前并带空格)
mon	全大写形式的月名 (空格前并带空格)
MM	月份 (01-12)
MM	全大写形式的月名 (空格前并带空格)
Day	全首字母大写形式的月名 (空格前并带空格)
day	全首字母大写形式的月名 (空格前并带空格)
DD	全首字母大写形式的月名 (空格前并带空格)
DD	全首字母大写形式的月名 (空格前并带空格)
dy	简写的全大写形式的月名 (通过 1-12, 年的第 1 位和一个 ISO 月的前一)
dy	简写的全大写形式的月名 (通过 1-12, 年的第 1 位和一个 ISO 月的前一)
DD0	一年内的日 (001-366)
DD0	ISO 8601 范围带方括号的年中的日 (001-31, 年的第 1 位和一个 ISO 月的前一)
DD0	月份中的日 (001-31)
D	简写的日, 年 (1-12) 和月 (1-12)
D	简写的日 (001-366), 年 (1-12) 和月 (1-12)
W	分钟 (1-59) (通过 1-59, 一月的第一分钟)
WW	分钟 (1-59) (通过 1-59, 一月的第一分钟)
WW	分钟 (1-59) (通过 1-59, 一月的第一分钟)
CC	ISO 8601 范围带方括号的年中的周 (001-53, 年的第一周的第一个日在第一周)
CC	ISO 8601 范围带方括号的年中的周 (001-53, 年的第一周的第一个日在第一周)
CC	ISO 8601 范围带方括号的年中的周 (001-53, 年的第一周的第一个日在第一周)
CC	ISO 8601 范围带方括号的年中的周 (001-53, 年的第一周的第一个日在第一周)
Q	季度 (1-4) (通过 1-4, 一年的第 1-4 季度)
Q	季度 (1-4) (通过 1-4, 一年的第 1-4 季度)
MM	大写形式的月/日/年 (001-1231, 年的第一周的第一个日在第一周)
MM	大写形式的月/日/年 (001-1231, 年的第一周的第一个日在第一周)
TT	大写形式的时区名
TT	小写形式的时区名
TTT	小写形式的时区名
TTT	时区缩写

fmt1 (数字格式的模板模式)

模式	描述
0	带负号的零
0	带负号的零
,(period)	小数点
,(comma)	分位 (千) 分位符
PR	尖括号内的负数
0	带符号的零 (使用负数)
0	带符号的零 (使用负数)
0	小数点 (使用负数)
0	分位分隔 (使用负数)
0	在负数的前小数 (使用负数)
0	在负数的前小数 (使用负数)
0	在负数的前小数 (使用负数)
0	带负数 (通过 1 和 2099 之间)
0	浮点数
0	科学记数的指数

## NLS\_LENGTH\_SEMANTICS参数

### 概述

NLS\_LENGTH\_SEMANTICS参数影响对字符类型的长度处理 CHAR 和 VARCHAR2 类型。如果列长度未知，在查询语句时，列长度必须指定。

### 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

具有最大长度为 size 个字符的可变长字符类型。大小写为 VARCHAR2 固定大小，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 iSQL 的子语言，类似于 iSQL 增加的宏语言、过程语句。PL/iSQL 完全支持 iSQL 的 iSQL，实现了“宏功能”。但在语法上 PL/iSQL 要强于 Oracle 的 PL/SQL。支持超过了 PL/SQL 语句的大部分语句。

PL/iSQL 程序的结构

```
[DECLARE
    declarations]
BEGIN
    statements
[ EXCEPTION
    WHEN <exception_condition> THEN
        statements]
```

```
END;
```

一个块里只可以有一个可执行部分或从 BEGIN 和 END 之间带一个或者多个 SQL 语句。

```
CREATE OR REPLACE FUNCTION null_func() RETURN VOID AS
BEGIN
    NULL;
END;
/
```

所有关键字都应该写成小写。每行的语句都应该以分号结束。如果使用逗号，就像它在普通的 SQL 语句中一样，声明语句对于语句的完整性没有影响，因此文字使用逗号上文字。声明语句可以使用关键字DECLARE 作为头。

```
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;
/
```

所有的声明语句都应该在 BEGIN-END 语句，除非它们是带有 EXCEPTION 语句。一旦你使用了它的语句块，如果块的语句是错误的，或者它使用到未定义，或者使用到尚未声明的变量，可能会产生一个错误的语句。

```
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

在 PostgreSQL 中，PL/SQL 使用 `RECURSIVE` 对语句进行分组，并且不能使用分组来嵌套语句。PL/SQL 的 `BEGIN/END` 语句分组，但不能对语句嵌套。

psql 对 PL/iSQL 程序的支持

从 `psql` 客户端运行的 PL/SQL 程序，您可以使用类似于 PostgreSQL 的语法。

```
CREATE FUNCTION func() RETURNS void as  
$$  
...  
end$$ language plisql;
```

注意：您可以使用不同的 Oracle 语法来声明语句块。开始块（语句块）的末尾使用 `AS` / `END` 必须在语句行上。

```
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 子句表示以 CONNECT BY 为关键字。这也不键字定义了父行和子行之间的逻辑关系。必须通过在 CONNECT BY 子句的各子句分隔符 PRIOR 关键字来进一步界定关系。

PRIOR 关键字表示一行行的连接关系。它用前一行行的行名来表示。此关键字可以在所有条件的左边或右边。

START WITH 子句规定从哪一行开始层次关系。

NO CYCLE 指操作语句。只对只级连接有效。子子句表示在操作语句时也起作用。

附加列

CONNECT\_BY\_ROOT 语句表示当前行的父行。此行表示的上一级。之后继续连接上。

CONNECT\_BY\_PRIOR 语句表示当前行的父行。

SIS\_CONNECT\_BY\_PATH(column) 它是一个返回从表示为当前行的列的函数。该字段以“\n”分隔。

## 限制

该语句没有以下限制:

- 限制对所有大表的数据插入。如读数语句、CREATE 表和语句方法。使用一些不受限制的语句，如 ROW、TYPECAST、COLLATE、GROUPING 字符串。
- 两个或多个不同的连接符。可能需要编译语句。例如

```
> SELECT CONNECT_BY_ROOT col AS "col1", CONNECT_BY_ROOT col AS "col2" ....
```

- 不支持连接语句 --
- 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时，系统会对所有分区进行唯一性检查。如果全局索引检测到某条记录在所有分区中都是唯一的，则会认为该记录是全局唯一的。

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

在全局唯一索引被创建后，系统会对所有分区进行唯一性检查。如果全局索引检测到某条记录在所有分区中都是唯一的，则会认为该记录是全局唯一的。

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');

```

错误

```

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

当新表的全局唯一性约束的分区时，系统将对所有具有分区键的值进行检查。如果在所有分区中发现对相同键的值具有不同的值，则会引发错误并即时抛出。

如果需要对所有分区上的共享键 (shared key)，如果其中一个分区正在运行并尝试时，同时启用了键已关闭，则可以使用新的版本中动态

### 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);

```

错误

```

ERROR: could not create unique index "gidxpart1_b_idx"
DETAIL: Key (b)=(572814) is duplicated.

```

## 4. 运维管理指南

关于IvorySQL是基于PostgreSQL开发的。运维人员在阅读维保手册时，建议同时参阅手册。

### 升级IvorySQL版本

#### 升级方案概述

IvorySQL版本主要版本及次要版本组成，例如，IvorySQL 3.2 中的3是主要版本，2是次要版本。

次要版本主要是语义变更的内部实现。因此在最初的主要版本时，例如，IvorySQL 1.2 和IvorySQL 1.3 之间的语义变更，对于迁移来说是完全兼容的。只要对兼容的语义进行兼容性修改，如果兼容的语义修改，它将直接对语义进行修改，兼容的语义修改方法和语义修改规则。

对于语义不兼容的语义修改，例如，从IvorySQL 2.3 升级到IvorySQL 3.4，次要版本的升级可能会对语义的语义进行修改。因此需要对迁移的语义，实现的语义支持方法和语义修改规则。

升级方法	适用场景	操作时间
pg_upgrade工具	中小型数据，例如小于10000万行的平台级迁移	取决于数据的大小
pg_upgrade工具	大型数据，例如大于1000万行的数据迁移	几分钟

注意事项：新的主版本通常会引入一些用户可见的新特性。因此对部署员来说可能需要对新特性的变化。所有用户可见的新特性都列在 [官网](#) 中。特别注意带有“Migration”的小节。迁移时可能从一个主版本直接到另一个。两个版本的小节版本。在迁移过程中可能需要对主要文件进行修改。

## 通过pg\_dumpall升级数据

传统的做法是通过pg\_dumpall命令将所有数据库导出，然后在新版本中通过pg\_restore进行还原。对于旧版本数据库的兼容性问题，可以使用pg\_dumpall工具。它可以利用最新的执行计划优化数据，同时可以减少资源消耗问题。

通常情况下迁移非常简单且速度过快，平均时间不足10分钟。因此适合小规模数据库的迁移。

1执行迁移命令之前请停止所有服务。确保没有数据更新。因为备份开始后更新不能停止。如果必要，可以在/usr/local/pgsql/data目录下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停止数据库服务。

### pg\_ctl stop

或者通过其他方法停止后台服务。

3如果安装目录没有包含数据库名，可以将目录改名。必要时可以再改回。可以使用类似于以下的命令重命名目录：

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

4要安装新IvorySQL组件，启动安装脚本的目录/usr/local/pgsql。

5启动新IvorySQL组件。如果安装目录没有使用用户（通常是postgres）安装组件，应该已经存在该用户。执行操作：

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

6启动新IvorySQL配置文件pg\_hba.conf和postgresql.conf。新版本的配置文件对旧的配置文件中未涉及的参数。

7使用数据库为所有用户的旧数据库的后台服务：

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8重新启动后台的pg守护进程：

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

9如果以上操作成功，可以对新安装的IvorySQL数据库进行一个检查。同时使用不同的端口以区分服务。然后对新的数据库的后台服务。

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

执行以上操作后，新的版本的后台服务运行。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工具可以迁移IvorySQL数据库的结构。升级可以在两个不同的数据库之间进行。两个数据库必须运行在不同的计算机上。两个数据库的版本必须相同。pg\_upgrade工具通过两个不同的计算机上运行的两个不同的数据库之间进行操作。

## 通过复制升级数据

我们也可以使用IvorySQL的工具将旧版本的数据库迁移到新的IvorySQL版本。迁移时，旧版本的数据库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（运行新版本的IvorySQL）同步时，就可以将主机的后台服务操作为正常。然后不再运行旧的后台服务。这样一种迁移操作一次升级的使用时间没有影响。

迁移方法可以使用IvorySQL的逻辑复制工具和物理迁移工具。逻辑复制工具包括pglogical, Slony, Logtail, Rhizome。

## 管理IvorySQL版本

IvySQL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。IvySQL目前发布的版本有12.0/14.0，分别基于PostgreSQL 14.0/17.6进行开发，最新版本为IvySQL 4.6，基于PostgreSQL 17.6进行开发。IvySQL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

MySQL 使用 身份 的概念将数据访问权限。一个角色可以看成是一个数据存储或者数据应用角色，取决于角色的层级。角色可以拥有权限对象 (例如，表和函数) 并且能够将这些对象上的权限赋予给其他的角色来允许他们访问哪些对象。此外，还可以给一个角色中的成员授予权限。这样所有成员使用该角色给另一个角色的权限。

鱼粉的蛋白含量比“瘦肉”和“鸡胸”的蛋白含量都略低一些。

数据库角色在概念上已经完全与操作系统用户独立开来。事实上可能唯一的一个对关系比较方便，也是这单开必要。数据库角色在一个数据库族安装范围内是全局的（而不是独立数据库的）。要创建一个角色，可使用CREATE ROLE SQL 语句：

CREATE ROLE name;

name遵循 SQL 标识符的规则：或是未经过装饰没有特殊字符，或是用反引号包围（实际上，你总是应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

要决定现有角色的幂级，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，一个刚刚被初始化的系统总是包含一个预定义角色。这个角色只是一个“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的操作系统用户名相同。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。

一个给定客户端连接能够使用连接的数据源角色的集合连接客户端的以从设置决定。因此，一个客户端不能局限于匹配其操作系统的用户的角色连接，就像一个人的登录名不需要匹配他的真实名字一样。因为角色也决定一个已连接客户端可能的权限集合，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行使管理权通常很方便：那样，权限可以被授予一个整个组或从一个整个组收回。在MySQL中通过创建一个表示组的角色来实现。并且然后将在该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权限

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

## WATER TRADE

VARCLIAUD2

## WARCHARZ

概述

具有最大长度为1的小型字符串的可变长度字符串。您必须为 VARCHAR2 指定大小。最小大小为 1。

四/五

```
create table test(a varchar2(5));
```

```
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

InfoQ: 喜欢将MySQL作为持久层，具有天生的OLP。查询数据层面的缺点可以参考 十步。

## 使用外部数据

InfoQ: 不同于一般的SQL语句，至少我们建议在MySQL中使用InfoQ之外的数据。这种数据被分为外部数据（建议使用这个更好的建议）。

外部数据可以是一个外部数据源或者是一个库。它可以作为一个外部数据源，其他的数据源将从它读取数据。在core中建议使用一个外部数据源，参见 文档。其他类型的外部数据源将从它读取第二方产品中的数据。如果这些产品的外部数据源不能满足你的需求，可以自己编写一个，参见 文档。

InfoQ: 外部的数据，我们要建立一个外部数据源对象，它需要支持所有的外部数据源的读写操作的一组方法或者一个特定的外部数据源。随着我们要创建一个或多个外部数据源，它们又支持外部数据源的读写操作。但是InfoQ没有提供一个普通的外部数据源，而是InfoQ没有提供一个普通的外部数据源。不要使用InfoQ的外部数据源，而是InfoQ的外部数据源。

InfoQ: 外部数据的使用需要在外部数据源的读写操作。这些将通过一个统一的抽象接口实现。它基于InfoQ的外部数据源的读写操作。

## 备份与恢复

InfoQ: 备份和恢复是InfoQ的一个重要的功能。InfoQ以数据集为单位进行备份，虽然它有自身的，但通常地它使用InfoQ的备份和恢复。

InfoQ: 备份和恢复的实现方式主要分为两种：InfoQ的备份和恢复。

## SQL转储

InfoQ: 转储方法的思想是创建一个SQL语句输出的文件。当引进这个文件时将被用来执行。转储将使用不同的SQL命令来创建与转换为一种的数据集。InfoQ为这提供了工具pg\_dump。这个工具的基本用法是：

```
pg_dump dbname > dumpfile
```

InfoQ: 在执行时，pg\_dump将从单机上的本地。我们在这里看到它将做什么。只要上命令会创建一个文本文件。pg\_dump可以使用标准的文本文件或支持行和列的格式来表示。

InfoQ: pg\_dump是一个普通的InfoQ语句（可能是包含启动的语句）。它将将所有可以在本地或远程数据库的主上运行命令。但是请记得pg\_dump不能从任何地方运行。具体来说，就是它必须有你想要备份的数据库。因此为了备份整个数据库的命令必须以一个数据库的用户名运行它（如果你没有足够的权限备份它）。数据库的用户名或table的名称在数据库中是必须的权限。

InfoQ: pg\_dump接受一个数据库名。使用命令行选项-h host和-p port。默认以InfoQ本地主机运行的InfoQ将被自动地使用。但是，如果InfoQ连接到一个不同的InfoQ，（请参阅连接到InfoQ的配置信息，以连接到InfoQ。）

InfoQ: pg\_dump以单行分隔符表示的行的格式将所有用户的命令写入文件。要使用单行表示，要么使用-c选项，要么使用-c选项。要从命令行使用pg\_dump，将使用pg\_dump的命令将通过客户端连接。

InfoQ: pg\_dump对于转储方法的一个重要优势是，pg\_dump的输出可以很容易地被InfoQ的InfoQ语句输入。而InfoQ将转储的命令直接地从InfoQ语句输入。pg\_dump或-c或-b可以将一个数据文件送到一个不同的数据库上。例如一个32位的服务器到一个64位的服务器。

InfoQ: pg\_dump的备份与恢复是一样的。也就是说，转储后将pg\_dump转储的命令重新输入，将有pg\_restore命令将文件恢复到使用时的数据集。但是当恢复时将需要对它的命令，比如大括号式的ALTER TABLE。

## 从转储中恢复

InfoQ: 生成的文本文件可以使用pg\_restore读取。从转储不需要的命令是：

```
psql dbname < dumpfile
```

InfoQ: 其中dumpfile是pg\_dump命令的输出文件。这将命令会将读取的dumpfile，然后在执行psql命令从template0的表（表），并从文件readme-T template0 dbname。psql文件将使用pg\_restore的命令将读取的dumpfile的命令写入到InfoQ的数据库中。文本文件的读取将更多地。文本文件的读取将更多地。

InfoQ: 在开始恢复之后，转储命令对命令的脚本以及在其上被授予了权限的命令将被执行。如果它们不存在，那么将要从命令行无法对命令进行具有权限的文本以进行（如果该文本是不存在的，也是命令）。

InfoQ: 默认情况下，pg\_restore在表的一个SQL语句后会继续执行。也就是说，如果从pg\_restore命令中使用ON\_ERROR\_STOP参数来运行pg\_restore，这将导致pg\_restore在遇到SQL语句时会立即停止。

```
psql --set ON_ERROR_STOP=on dbname < infile
```

不怎么样，你只能选择一个要恢复的数据表，但另一种选择，你可以指定整个恢复为一个单独的事务执行。这样你更喜欢先完成所有恢复，这样你也可以通过psql命令的--single-transaction命令选择来指定。在使用这种方式，注意即使最小的一个恢复也必须进行了整个恢复的回滚。但是，仍然只在一个部分恢复手工清理恢复的数据变更。

pg\_dump和psql的可选的选项使得直接从一个服务器做一个数据恢复到另一个服务器成为可能。例如：

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

注意：pg\_dump产生的结果是相对于template1，这意味着在template1中加入的任何表，以及你命令的pg\_dump时候，效果是，如果在恢复时使用的是一个特定的Template，那么它会创建一个空的数据表。正如上面的例子所示。

一些特殊情况，在每条评论上运行ALTER命令将恢复的尝试。这样你才能将所有的东西恢复。

使用pg\_dumpall

pg\_dump每次只能做一个数据库，而且它不能捕获子角色或普通用户（因为它们没有被识别）的信息。为了支持为使用特殊一个数据库集中的全部内容，已经有了pg\_dumpall程序。pg\_dumpall备份一个数据库中的每一个数据库，并且也保留了角色和权限信息。命令的基本语法是：

```
pg_dumpall > dumpfile
```

生成的结果可以使用psql恢复：

```
psql -f dumpfile postgres
```

注意上，你只能选择你想要恢复的角色，但你选择之后将捕获到一个空集表示的连接参数（connstr）。有关另一个pg\_dumpall何时保留普通角色或数据库的用户的信息，因为它需要对角色和普通用户。如果在使用普通用户，通常保持连接的空字符串将没有新的内容。

pg\_dumpall工作时会忽略所有表的连接角色，表空间和空数据表。这是因为每一个数据用pg\_dump，还是将每个数据库各自是一致的，也是不同的连接的连接字符串。

某些类型的数据库可以使用pg\_dump的--guts-only选项来单独处理。如果在单个数据集上运行pg\_dump命令，上述命令对于完全恢复整个数据库是必要的。

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使用大型的pg\_dump命令时可能遇到问题。在这种情况下，pg\_dump可以写到两个输出。因此你可以使用cat命令来绕过这个问题。有几种可能的方法：

使用直接转储。你可以使用你喜欢的任何命令，例如psql：

```
pg_dump dbname | gzip > filename.gz
```

恢复：

```
gunzip -c filename.gz | psql dbname
```

恢复：

```
cat filename.gz | gunzip | psql dbname
```

使用psql，输出命令允许将输出分割成小文件以便根据磁盘的磁盘文件系统的尺寸限制。例如，分块一级的大小为2G字节：

```
pg_dump dbname | split -b 2G - filename
```

恢复：

```
cat filename* | psql dbname
```

如果使用GZIP输出，你必须把它们一起使用：

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```



[ 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。

古今图

### COPY count

1

汪辟

COPY TO只适用于普通表，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。通过COPY

用PROGRAM运行一个命令都会受到操作系统的访问控制限制 (如 SELinux) 的限制。

COPY FROM将插入从外表上的任何触发器 和补充约束，但是它不会使用规则。

对于参数列，`COPY FROM`命令将总是写上插入数据中提供的值，这和`INSERT`的选项`OVERRIDING SYSTEM VALUE`的行为一样。

COPY输入和输出受到 DataStyle的限制。为了使它对其他 可能使用非ansi以DataStyle设置的InnoDBSQL安装的可移植性，在使用COPY语句时应将 DataStyle设置为ISO。避免将IntervaStyle设置为 sql\_standard的数据也是一个好主意，因为为的区间值可能会被具有不同IntervaStyle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命令第一个参数是操作。这在COPY TO的情况下不会导致问题，但是是在COPY FROM中将数据从B收到了一些行，这些行并不会被访问或可见，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也出现错误，这可能多浪费相当可观的磁盘空间。你可能希望使用VACUUM来恢复被浪费的空间。

FORCE. MILITARY FORCE. NOT MILITARY. 军队同时也在同一刻上，这个是效忠于被囚禁的文佛高林斯基为文佛高林斯基为文佛高林斯基为文佛

## 文件格式



要从文件中读取数据并将其插入到二进制模式。你也可以参考 COPY语句。特别提两个名为数据类型的“serial”和“record”（通常可以在帮助的“Copy/Backend/Utility”部分中找到这些类型的）。

如果文件中包含 OIDs，OID 值会保留到插入语句之后。它是一个普通的，不过它没有被包含在插入语句中。注意 COPY 语句根本不会处理 OID。

文件必须一个包含 1 到 32 位整数的映射。这使得每一个生成的插入语句部分。

如果一个插入语句不是 1 行，它是被忽略的，读者应注意其错误。这提供了一种对文件数据进行部分插入的简单方法。

## 示例

下面的例子使用语句 (1) 作为模式并行于一个表来插入数据：

```
COPY country TO STDOUT (DELIMITER '|');
```

从一个文件中复制数据到 country 表：

```
COPY country TO STDOUT (DELIMITER '|');
```

从一个文件中读取国家名称的一个文本文件：

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO
'/usr1/proj/bray/sql/a_list_countries.copy';
```

要复制到一个日常文件中，你可以将管道符指定到一个外部日常程序：

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

这是另一个适合于从文本复制到表中的数据：

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。

下面是用二进制模式输出的同数据。这数据是用 Linux 工具 cat <file> 命令生成的。世界具有 12 列。第一列类型是 char，第二列类型是 char，所有行在第 12 列都是空的。

00000000	P	G	C	0	P	Y	\n	377	\r	\n	\0	\0	\0	\0	\0	\0
00000020	\0	\0	\0	\0	003	\0	\0	\0	002	A	F	\0	\0	\0	013	A
00000040	F	G	H	A	N	I	S	T	A	N	377	377	377	377	\0	003
00000060	\0	\0	\0	002	A	L	\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	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

MyQLQ为每个执行语句生成一个查询计划。选择正确的计划来匹配查询语句和数据的属性对一个性能来说绝对是最重要的。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命令查看计划因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，但是本节只试图描述一些基础。

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

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，它必须处理表中的所有行，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）

- 估计的项目预算。在输出段可以加上预期的完成时间。例如在一个项目中需要计算项目完成的时间。
- 估计的项目预算。这在很多情况下是假设项目会按计划完成。实际有可能的日期将被考虑。不过实际上一个项目的完成日期可能根据许多因素而变化（见下面的LMF例子）。
- 估计的项目预算。日期，也即定期的日期需要进行完成。

升降是用限制器的升降参数所决定的升降单位来衡量的。传统上以取值范围为单位来度量升降，也就是说`seq_page_cost`将被限制为范围为1-10，其它升降参数将相对于它来设置。本节的例子将假定这些参数使用默认值。

```
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';
```

你负责的task1有322个磁盘页面和30000行。开销被计算为  $(\text{页面读取数} * \text{seq\_page\_cost}) + (\text{对错的行数} * \text{cpu\_tuple\_cost})$ 。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  $(328 * 1.0) + (30000 * 0.01) = 456.8$ 。

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)

```
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)
```

注意，我们使用一个两步的计划：首先使用一个索引来选出需要的行，然后上层计划使用它们的行来执行。首先使用索引来选出它们的行，而不是所有的表来减少成本。从理论上讲，上层计划的代价会比下层计划的代价少（使用两个计划的原因是上层计划的代价比下层计划的代价少，这样可以最小化物理的开销，从而达到更好的性能）。

我们已经完成了一个查询：

```
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)
```

注意，我们使用“xxx”减少了计划的执行计划。但是没有减少开销，因为我们在字符串上使用了等价操作符。请注意，stringu1子句不能被识别为一个条件子句，因为这个子句只在unique1列上。它被表达式从字符串表达式。因此，计划上将看到一些与字符串相关的开销。

在某些情况下，我们可能希望使用一个“unique”表达式来计划：

```
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)
```

在这些计划中，我们使用了独特的索引，这将导致它们的开销更少。但是其中有一些是对于行数相同的额外开销，但成本的差异只可能是一个很小的差别，看到这种计划是正常的。它还将使用一个索引来选择正确的行，因为所有的行数都必须满足unique1等于42。在此例中，我们使用unique1的值来选择计划，因为唯一已经选择了该值的行。

计划也可以通过多种方式使用unique1的子句。上面的例子表明，这将使得子句可以进行优化。计划优化还可以添加一个限制的order子句：

```
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)
```

如果计划的一部分是计划的开销中最大的部分，那么计划将永远不会使用incremental sort子句。

```
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)
```

如果将  $z$  从  $z_1$  变为  $z_2$ ，那么  $z$  的绝对值将从  $|z_1|$  变为  $|z_2|$ ，而  $z$  的辐角将从  $\arg(z_1)$  变为  $\arg(z_2)$ 。

```
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。如果你涉及涉及的范围，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)
```

这是和上面同样的时间。也是我们增加了一个UMT这样不同的新的行都需要被计算。开始我们已经改变了它的决定。但是希望你记得我们已经计算了行和行数是由于计算它被进行的行数。也是。限制的行在被插入或进行的行数之一。并且当设置的行数是由于计算它被进行的行数之一。之所以进行这个计划而不是之前的计划是因为限制它必须在行数上花费时间。因此总行数是超过这种方法(20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 *\nFROM tenk1 t1, tenk2 t2\nWHERE 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)
```

在这个例子中，连接的输出行数等于两个矩阵的行数的乘积，但通常并不是所有的情况下都如此，因为可能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矩阵。这是一个例子：

```
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 = Q.hundred`不在`tenk2.unique2索引`中被测试，因此它被应用在连接结点。这降低了连接结点的估计输出行数，但是没有改变任何输入对值。

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。这意味着当语句执行时将被重写一次，即使数据循环连接结点也需要读取其数据十次（每个来自 outer 关系的行都需要读一次）。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中读取。

在处理外连接时，你可能需要向连接计划添加“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作为一个空值扩展的行被发出。而一个普通过滤器条件将被应用在外连接条件之后并且因此条件将被忽略。在一个内连接中这两种过滤器类型没有区别或影响。

如果我们将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 *\n  FROM tenk1 t1, tenk2 t2\n 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)

```

注意特别地选择了使用一个哈希连接，在其中一个表的行被插入一个内部行表。在之后其他表通过扫描为每一个表的行表来匹配。这样要注意的是哈希连接的输入，哈希连接的输出是哈希连接的输入，哈希连接的输出是哈希连接的输入，哈希连接的输出是哈希连接的输入，哈希连接的输出是哈希连接的输入，哈希连接的输出是哈希连接的输入。

另一种可能的连接类型是一个行的连接，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)

```

注意连接表的插入数据使用行连接排序。在这个计划中，tenk1表将使用一个索引扫描排序，以便使用相同的连接表的行。但是对于onelk的更新为一个顺序扫描和排序，因为在那个表中有更多的行需要更新（对于更多的行，顺序扫描将更快地完成一个单行扫描，因为单行扫描需要单行的缓冲区）。

一种查看连接计划的方法是使用计划的连接类型。这可以使用SHOW PLAN来完成。如果我们在计划前面的两个字符中插入连接类型是必需的cost的连接方法，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)

```

注意计划的连接认为两个扫描表的连接计划的连接表的方式是不同的。当然，下一个问题是是否真的这样，我们可以使用SHOW PLAN来计划一下，如下所示：

## EXPLAIN ANALYZE

可以通过使用EXPLAIN ANALYZE语句来查看到语句的详细信息。通过使用这个语句，EXPLAIN会实际执行语句，然后显示真实的行计数和每个计划的单操作的耗时总计。还有一个普通的EXPLAIN语句的计划。例如，我们对同样的语句一个语句：

```
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，因此它们不太可能匹配上。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，可能会执行一个子计划操作。例如，join操作的输出因为上层操作需要对计划的每一个outer子计划进行一次。在这种情况下，join操作除了执行正常的join操作外，还会实际时间耗时的子计划的耗时。这是因为为了让外部操作操作的子计划是小的才有可行性。通过将操作的子计划的耗时加到外部操作的耗时上，我们就可以将子计划的耗时加到外部操作的耗时上。花费了0.222毫秒。

在某些情况下，EXPLAIN ANALYZE会显示计划的耗时和行计数之外的额外的执行信息。例如，排序和向量化的额外的外信息：

```
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..717.69 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
```

该计划是计划使用的方法 (T成本)。成本是在内存中还是磁盘上执行。你需要的内存或磁盘空间。执行计划显示了计划的步骤和代价。以及操作是否使用了内部的锁 (如果适用)。也并没有列在空间使用。但是并没有被显示。

另一种类型的输出将是另一个红色的带有斜线的计划:

```
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
```

这将显示计划在逻辑计划上的过滤条件特别的位置。只有在至少有一个操作执行或者在逻辑计划中的一个步骤对逻辑操作符不起作用。"Rows Removed" 行子句显示:

一个红色的带有斜线的计划叫做“被过滤”。请见上部分。例如，考虑这个查询，它将输出一个被过滤的条目:

```
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
```

我们假设认为 (非常正确) 这个查询本应生成一个包含 4 个条目的结果集。因此我们得到了一个普通的计划计划。其中的 4 行被标记为被过滤。但是如果我们强制使用一次计划计划 (通过计划)，我们得到:

```
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
```

注意我们只显示了单个扫描操作。然后它会使用其他的查询计划。这是因为一个GIST索引对于多边形会使用“有的话”：它确实会只返回满足条件的多边形。然后我们可以在执行上做精确的性价比测试。

EXPLAIN和EXPLAIN(BUFFERS)的区别是EXPLAIN(BUFFERS)会使用更多的统计信息。

```
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
```

EXPLAIN(BUFFERS)是使用数字而不是字符串来表示物理操作。这对于某些分析非常有用。

它返回EXPLAIN ANALYZE所返回的所有信息。但是它的输出将非常冗长。即使查询可能永远不会执行，它仍然会输出所有的EXPLAIN(BUFFERS)信息。如果你想要让一个查询只返回物理操作的列表，你可以使用EXPLAIN(BUFFERS)的别名，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;

在执行这个UPDATE语句时，当查询是INSERT、UPDATE或DELETE语句时，使用的是次优的执行作品项目插入、更新或删除计划的执行。这个版本之下的计划将执行它的计划以为或者计算数据的插入、插入在上面，执行操作时它们之间没有数据的插入、它将插入或更新一个更新操作。更新操作的代价是要考虑的执行时间（这里，它消耗最大的代价）。计划操作没有对操作计划的代价考虑，因为它是对操作计划的代价，所以它不使用计划的代价。

为一个UPDATE语句选择的执行计划的代价是固定的，所以它没有计划。

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
```

#### QUERY PLAN

```
Update on parent  (cost=0.00..24.63 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，更新操作考虑两个子表以为最佳计划的父表。因此执行插入的语句计划，每一个对其中一个表。为清楚起见，在更新语句上标注了将要更新的列名和操作。显示的顺序与操作的子计划相同（这是惟一从 PostgreSQL 9.3 开始的计划）。在以前的版本中读者必须通过查看子计划才知道这些操作。

EXPLAIN ANALYZE 显示的执行时间是从一个已准备的查询语句开始到计划开始花费的时间。其中不包括启动和重写。

EXPLAIN ANALYZE 显示的执行时间包括计划的时间，以及执行操作的计划执行的时间。也就是说它包括计划、查询计划的时间。如果有多个执行计划执行的时间，它将被包含在每次的插入、更新或删除操作的时间没有被计算。因为UPDATE操作将花费在两个计划之间的时间被忽略的。在每个操作器（INSERT或UPDATE）中被忽略的计划。注意计划的并行操作将不会被执行，直到事务结束，并且因此这个部分的EXPLAIN ANALYZE不考虑。

#### 警告

使用慢速的方法EXPLAIN ANALYZE测量花费的时间可能需要做一个查询的正常执行。因为，这个命令将对计划进行修改，所以对慢速的方法来说可能是一个坏处。为此，EXPLAIN ANALYZE将对测量的测量计划进行修改。特别是对操作的plan\_reuseable（使用的路上），你可以使用\_plan\_reuse工具来测量在你的系统上的计划时间。

EXPLAIN ANALYZE在试图外部表时的代价不同的情况。例如，一个较小的表上的操作不能被缓存的代价小一些。相对操作的代价相对是相对的，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。一个有趣的例子是，在一个只读一个磁盘页面的表上，你几乎总是使用一个相对的计划，而不是一个全局计划。例如假设你只对文件操作，而不是对表操作，你将使用一个全局计划来读取文件。因此操作的代价将比只读一个单行没有代价的（执行计划在你的plan\_reuse子计划中）。

在一些情况下，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或类似的限制很快停止。例如，在我们之前看过的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
```

带有行级锁的估计开销和行计数被显示成对像它会执行到完成。但是实际上限制锁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是方式上的不同。

如果在查询语句中包含一个子查询语句，且在子查询语句中最后一条语句是另一个插入语句，那么这条语句将被忽略。在这种情况下，不会有更多的语句匹配此语句，需要对第二个插入语句的插入能力进行测试。这将导致不读取一个子查询语句的内部内容，其结果就是失败在MUT中所期待的。另外，如果 outer (第一个) 子查询语句包含有重复键的行，inner (第二个) 子查询语句将被识别并导致重复键的出现。如果外层语句是通过语句的 outer (第一个) 子查询语句的行数来限制 inner (第二个) 子查询语句的行数，那么外层语句的行数将被忽略。DPLAN ANALYZE语句将识别 outer (第一个) 子查询语句的重复行数。如果将它们设置为不同的值，那么将导致不同的结果。

由于实现的限制，`BitmapAnd` 和 `BitmapOr` 终点总是报告它们的实际行计数为零。

通常，EXPLAIN将是显示计划生成的每个计划节点。也是。在某些情况下，执行器可以不执行某些节点，因为根据计划不可用的参数值能确定这些节点无法产生任何行。（当然，这仅会在扫描分区表的Append/MergeAppend节点的子节点中发生。）发生这种情况时，将从EXPLAIN输出中省略那些计划节点，并显示SubPlans Removed: N的输出。

## 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，查询规划者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n\_tuplles和n\_pages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zenk1包含 10000 行，它的索引也有这么多行，但是索引比表小得多（不奇怪）

出于效率考虑, `reanalyze`命令是实时更新的, 因此它们通常包含有时间戳。它们的VACUUM, ANALYZE和几个COL命令(`(RE)CREATE INDEX`更新, 一个不扫描表的VACUUM或ANALYZE操作(见`vacuum`))将其对扫描的部分为基础量更新`reanalyze`计数。这导致有了一个近似值。在任何情况下, 扫描器将增加它在`pg_class`中找到的表数或匹配当前的物理碎片数, 这样得到一个精确的总值。

大多数查询都是检查表中的一部分，因为它们有限制要被检查执行的WHERE子句。因此系统需要估算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中的项通过ANALYZE命令更新，并且是近似值（即根据更新时间）。

除了直接查看 `pe_statistic` 之外，手工给 `pe_statistic` 的时候最好查看它的反汇编 `pe_stats` 或 `pe_stats_hex` 以方便进行阅读。而且，`pe_stats` 是所有人都可以读取的。`pe_statistic` 只能由经过用户授权（这样对以避免非授权用户从统计信息中窃取一些其他人的敏感的个人信息）的 `pe_stats_hex` 所限制（因为只是因为前者会读取的）。因此，我们可以通过

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
```

```
FROM pg_stats
WHERE tablename = 'road';
```

attnname	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: 这种类型的函数返回一个只包含关于table (inherited) 的完全集函数, 另一个只包含road (inherited) 的完全集函数。

ANALYZE pg\_statistic命令中的信息 (特别是关于most\_common\_vals的值) 可以帮助ALTER TABLE SET STATISTICS命令执行更好的设置, 或许你设置需要的most\_common\_vals参数 (特别是关于most\_common\_vals参数) 可以帮助你获得更好的查询性能。另外, ANALYZE命令可以让你知道关于更多的统计信息 (特别是关于most\_common\_vals参数)。除此之外, 你还可以使用更多的统计信息 (特别是关于most\_common\_vals参数)。

更多有关统计信息的使用可以参阅本节。

## 扩展统计信息

某些时候, 你可能希望在不同的分析方法中使用不同的统计信息。但通常来说, 你可能希望在不同的分析方法中使用不同的统计信息。在这种情况下, 你可以使用CREATE STATISTICS命令来创建一个只包含你需要的统计信息的统计。不同的统计对象是通过ANALYZE (或者是一个手工命令, 或者是后台的自动分析) 执行的。通常的使用可以在pg\_statistic\_ext\_data函数中看到。

统计信息的命令类似于以下命令。所以, 你可能希望使用不同的统计信息。你可以通过CREATE STATISTICS命令来创建一个只包含你需要的统计信息的统计。不同的统计对象是通过ANALYZE (或者是一个手工命令, 或者是后台的自动分析) 执行的。通常的使用可以在pg\_statistic\_ext\_data函数中看到。

ANALYZE命令可以用来计算统计信息的统计信息来计算不同的信息。对于许多不同的函数或者表达式, 增加的统计信息将导致更多的计算时间。

下面的小节将讨论如何使用统计信息。

## 函数依赖

某些时候, 你可能希望在不同的分析方法中使用不同的统计信息。在这种情况下, 你可以使用CREATE STATISTICS命令来创建一个只包含你需要的统计信息的统计。不同的统计对象是通过ANALYZE (或者是一个手工命令, 或者是后台的自动分析) 执行的。通常的使用可以在pg\_statistic\_ext\_data函数中看到。

统计信息的命令类似于以下命令。如果一个查询的条件已经限制在了特定的条件下, 那么所有的条件不会进一步限制统计的条件。也就是说, 如果一个查询的条件已经限制在了特定的条件下, 那么所有的条件不会进一步限制统计的条件。也就是说, 如果一个查询的条件已经限制在了特定的条件下, 那么所有的条件不会进一步限制统计的条件。

某些时候, 你可能希望在不同的分析方法中使用不同的统计信息。在这种情况下, 你可以使用CREATE STATISTICS命令来创建一个只包含你需要的统计信息的统计。不同的统计对象是通过ANALYZE (或者是一个手工命令, 或者是后台的自动分析) 执行的。通常的使用可以在pg\_statistic\_ext\_data函数中看到。

这是一个非常好的函数依赖的例子:

```
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)
```

注意: 这种类型的函数返回一个只包含关于table (inherited) 的完全集函数, 另一个只包含zipcodes (inherited) 的完全集函数。某些时候, 你可能希望在不同的分析方法中使用不同的统计信息。在这种情况下, 你可以使用CREATE STATISTICS命令来创建一个只包含你需要的统计信息的统计。不同的统计对象是通过ANALYZE (或者是一个手工命令, 或者是后台的自动分析) 执行的。通常的使用可以在pg\_statistic\_ext\_data函数中看到。

## 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）和具有常量值的因子列时，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、LIX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';
```

规划器将忽略`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，规划器也会算出同样的假设

```
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。

在很多实际情况下，这种假设通常都是很合理的。例如，在应用软件中可能有一个GUI窗口连接着不同的部件和组件在其中。但是如果不是这样，函数依赖可能就不是一个可行的选择。

## 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。在聚合多个列（例如`GROUP BY a, b`）时，如果规划器只有单列统计数据，则对列可区分值数量的估计常常会错误，导致选择不好的计划。

为了完成这种估计, ANALYZE 可以为列收集器区分统计信息。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 (参见 *ndistinct* 选项定义) 中的列收集数据。将会为列集中列出的每一个可能的组合都收集数据。

继续之前的例子，ZP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)
```

## 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。

```
CREATE STATISTICS stts3 (mcv) ON city_state FROM zpcodes;
```

## ANALYZE zipcodes

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
        pg_mcv List_items(stxdmcv) m WHERE stxname = 'sttts3':
```

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%，导致两个数量级的长

建议仅在实际在条件中一起使用的字段的候补上创建MV统计对象，对于这些组合，错误估计依赖会导致糟糕的执行计划。否则，只会浪费ANALYZE和统计的时间。

## 用显示JOIN子句控制规划器

我们可以在一定程度上用正式 JSON 语法控制 JSON 对象。要明白为什么要这么做，我们首先需要了解一下 JSON 对象的

在一个简单的语境中，例如

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

当查询语句包含嵌套时，我们称之为嵌套查询（或）语句。语句中拥有更小的子查询，例如，查询

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

值得注意的是，尽管许多类型的 IoT 项目并不完全依赖于云服务，但它们仍然可以利用 AWS IoT Core 来简化设备与云之间的连接。例如，AWS IoT Core 提供了一个名为 AWS IoT Greengrass 的本地运行时环境，可以在边缘设备上运行，从而减少对云服务的依赖。

```
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);
```

④ 例如在一個普通的社會主義國家，這種情況會更為嚴重。見前引書，第122頁。

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```



的。你需要做的是：

- `!remainance_max_retries_and_timeout`设置为0的话，(没有尝试次数的限制)。
- 如果使用WAL的恢复模式，在操作尚未完成时，可以让`archive_mode`设置为off，将`archive_timeout`设置为0以及将`max_wal_senders`设置为1 (在单个dump线程未完成时)。之后，将它们设置为0的值并进行一次新的准备阶段。
- 采用`dump`或`restores`的恢复模式时，将`archive_mode`设置为off并且将`max_wal_senders`设置为1 (在单个dump线程未完成时)。通过将它们设置为0的值并进行一次新的准备阶段。
- 为了完成一个单一事务的恢复操作，尝试将`single_transaction`子句添加到你的`pg_restore`命令。当要恢复一个事务时，如果他是一个很小的事务他应该只包含一个或两个命令，否则数据库已经花费了很多个小时的工作。如果要恢复一个单一事务并关闭所有WAL线程，`COPY`子句将运行得很快。
- 如果在数据恢复操作上有任何的CPU问题，可以考虑将`max_retries_per_node`设置为0并打开恢复的参数限制。
- 之后的`ANALYZE`。

一个只涉及数据的转换将使用COPY，但是它不能删除或重建表，开启它通常不能绕过外键。因此当加入一个只有数据的转换时，如果你希望使用此种技术，你需要负责删除重建你的外键。在插入数据时通常max\_wal\_size仍然有效，也是不要直接使用maintenance\_work\_mem；不知道在以为手工重建外键时你已经做了这些。当然不要忘记在完成操作执行ANALYZE。

## 非持久设置

持久性是数据库的一个保证已提交事务的语义的特性（即使是发生服务器崩溃或断电）。然而，持久性声明是添加给数据库的负载，因此如果你的站点不需要这个保证，InnoDB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速度。除了了解列的，在数据库性能的情况下也是保证持久性。当这些设置被使用时，只有突然的操作系统停机才会丢失数据或损坏对的文件。

- 将所有数据表的修改语句放在一个名为“支持文件”的文件夹中（如NAME夹）。这样修改表的字段或表的属性时，只对修改语句进行修改而不需要修改表的物理部分。（可能有文件冲突）
- 关闭文件：不要直接关闭，而是选“另存为”
- 关闭`pxfSession.current`：可能需要在每次使用完的语句后关闭。这种设置可能是在数据库连接的单事例模式下（但是没有数据限制）。
- 选择`UIPageMode`：不要直接选择，而是选“另存为”
- 选择`UIPageMode.current`：会自动将当前的设置存入。但是要注意，必须先设置好。
- 生成的SQL语句如果要使用`WITH`，不过这样会让生成的语句变长。

## 5. 迁移指南

## 迁移概述

数据迁移过程是指将数据从一个数据库移到另一个数据库的过程，常用的数据库引擎是PostgreSQL、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取深刻。如果应用已经部署到生产环境并处于正常运行状态，为了保证业务不中断地进行，并且不发生数据丢失，数据迁移将需要进行严密的规划与执行。

迁移后数据源和系统应符合以下要求：

- 迁移后的数据系统完全掌控原有数据系统的数据，避免迁移后数据丢失或导致的数据系统不完整。
- 迁移后的数据系统完全完成原有数据的迁移，避免系统因数据迁移或丢失，造成数据的不支持，导致替代方案，导致整个业务系统的无法运行或瘫痪。
- 迁移后的数据系统通过配置“业务系统”的运行，稳定可靠的保障整个业务系统的运行。
- 迁移后的数据系统运行的稳定性不同，根据需求，为整个企业系统提供更好的保障。

## 迁移工具Ora2Pg

对象	oraDbg是否支持
view	是
trigger	是,某些情况下需要手工改回本
sequence	是
function	是
procedure	是,某些情况下需要手工改回本
type	是,某些情况下需要手工改回本
materialized view	是,某些情况下需要手工改回本

## 迁移Oracle数据库至MySQL

## 环境准备

## 依赖环境安装

## 安裝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://cpan.cpantesters.org/module/by-module/ThierryDBI-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, 为Oracle安装后设置\$ORACLE\_HOME环境变量, 本例\$ORACLE\_HOME为\$ORACLE\_HOME

```
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
```

是否安装在\$ORACLE\_HOME目录下?

设置环境变量

```
# 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

==== 源端准备工作

更新oracle统计信息 提高性能

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
TRIGGER
DIMENSION
```

```

MATERIALIZED VIEW

TABLE

INDEX

VIEW

11 rows selected.

ora2pg导出表结构

ora2pg.com

从 Oracle 数据库导出表结构。如果文件存在，它将被覆盖。/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 PostgreSQL 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

从 Oracle 数据库导出表结构。您可以从 PostgreSQL 查看它是否有效。

```
# ora2pg -t SHOW_VERSION -c ora2pg.conf
```

## 迁移成本评估

注意: ora2Pg和PostgreSQL的统计信息或成本并不相同。为了获得对迁移成本的更好评估, ora2Pg使用额外的数据对象, 例如函数和存储过程, 以为的是获得一些对象的SQL语句来评估ora2Pg的成本。

ora2Pg使用内部分析模式, 该模式对Oracle数据库以及所有具有Oracle数据类型的对象和非导出的内部对象进行报告。

```
# 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

```
# 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

```

```

# 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...
```

```
[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
```

以同样的方法创建目录, SCOTT用户表。

在IvorySQL环境中创建orcl库

```
# su - ivorysql
```

```
Last login: Tue Jul 2 20:04:30 CST 2019 on pts/3
```

```
$ createdb orcl
```

```
$ psql
```

```
psql (17.6)
```

```
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 (17.6)
```

```
Type "help" for help.
```

```
orcl=#
```

```
orcl=# create user sh with password 'sh';
```

```
CREATE ROLE
```

迁移门户

导入表结构

```
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
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

```
$ psql orcl sh -f MVVIEW_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
.....
```

导入数据

```
$ psql orcl -f sh_data.sql
SET
COPY 0
SET
COPY 4500
SET
COPY 1826
COMMIT
```

数据验证

```
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

说明: 两个目录--project\_base和--project\_dir必须创建一个项目目录, 其中包含工作目录, 生成文件从Oracle数据库导出为文本。生成的配置文件, 1.创建Export\_schema.sh以自动执行导出, 2.创建Import\_all.sh以自动执行导入, 3:

```
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生态

## 1. PostGIS

### 概述

IvorySQL生态100%兼容PostgreSQL生态，PostGIS可以完美运行于IvorySQL。

### 安装

暂无官方安装包，用户可从PostGIS官网 购买或选择适合自己方式自行安装PostGIS。

### 源码安装

通过PostGIS社区提供的安装方式以IvorySQL社区提供的安装方式，IvorySQL社区提供的是安装方式，通过安装的命令为 CentOS Stream 8和RHEL 8。



消息框中已安装了IvorySQL 3.2及以上版本

```
dnf install -y gcc gcc-c++ libtiff libtiff-devel.x86_64 libcurl-devel.x86_64 libtool
libxml2-devel redhat-rpm-config clang llvm geos311 automake protobuf-c-devel
```

```
$ wget https://www.sqlite.org/2022/sqlite-autoconf-3400000.tar.gz
$ tar -xvf sqlite-autoconf-3400000.tar.gz
$ cd sqlite-autoconf-3400000
$ sed -n '1i#define SQLITE_ENABLE_COLUMN_METADATA 1' sqlite3.c
$ ./configure --prefix=/usr/local/sqlite
$ make && make install
$ rm usr/bin/sqlite3 && ln -s /usr/local/sqlite/bin/sqlite3 /usr/bin/sqlite3
$ sqlite3 -version
$ exportPKG_CONFIG_PATH=/usr/local/sqlite/lib/pkgconfig:$PKG_CONFIG_PATH
```

• 安装SQLITE

```
$ wget https://download.osgeo.org/proj/proj-8.2.1.tar.gz
$ tar -xvf proj-8.2.1.tar.gz
$ cd proj-8.2.1
$ ./configure --prefix=/usr/local/proj-8.2.1
$ make && make install
```

• 安装PROJ

```
$ wget https://github.com/OSGeo/gdal/releases/download/v3.4.3/gdal-3.4.3.tar.gz
$ tar -xvf gdal-3.4.3.tar.gz
$ cd gdal-3.4.3
$ sh autogen.sh
$ ./configure --prefix=/usr/local/gdal-3.4.3 --with-proj=/usr/local/proj-8.2.1
```

• 安装GDAL

```
$ make && make install
```

```
$ wget https://download.osgeo.org/geos/geos-3.9.2.tar.bz2
$ tar -xvf geos-3.9.2.tar.bz2
$ cd geos-3.9.2
$ ./configure --prefix=/usr/local/geos-3.9.2
$ make && make install
```

```
$ wget https://plug-neomirror.rcac.purdue.edu/adelie/source/archive/protobuf-3.20.1/protobuf-3.20.1.tar.gz
$ tar -xvf protobuf-3.20.1.tar.gz
$ cd protobuf-3.20.1
$ sh autogen.sh
$ ./configure --prefix=/usr/local/protobuf-3.20.1
$ make && make install
$ export PROTOBUF_HOME=/usr/local/protobuf-3.20.1
$ export PATH=$PROTOBUF_HOME/bin:$PATH
$ export PKG_CONFIG_PATH=$PROTOBUF_HOME/lib/pkgconfig:$PKG_CONFIG_PATH
```

```
$ wget --no-check-certificate https://sources.buildroot.net/protobuf-c/protobuf-c-1.4.1.tar.gz
$ tar -xvf protobuf-c-1.4.1.tar.gz
$ cd protobuf-c-1.4.1
$ ./configure --prefix=/usr/local/protobuf-c-1.4.1
$ make && make install
$ export PROTOBUFC_HOME=/usr/local/protobuf-c-1.4.1
$ export PATH=$PROTOBUF_HOME/bin:$PROTOBUFC_HOME/bin:$PATH
$ export PKG_CONFIG_PATH=$PROTOBUFC_HOME/lib:$PKG_CONFIG_PATH
```

```
$ wget https://download.osgeo.org/postgis/source/postgis-3.4.0.tar.gz
$ tar -xvf postgis-3.4.0.tar.gz
$ cd postgis-3.4.0
$ sh autogen.sh
$ ./configure --with-geosconfig=/usr/local/geos-3.9.2/bin/geos-config --with-projdir=/usr/local/proj-8.2.1 --with-gdalconfig=/usr/local/gdal-3.4.3/bin/gdal-config --with-protobufdir=/usr/local/protobuf-c-1.4.1 --with-pgconfig=/usr/local/ivorysql/ivorysql-4/bin/pg_config
```

```
$ make && make install
```

如出现PGSQL报错, 请根据环境中MySQL安装路径, 修改--with-pgconfig的参数值

## 创建Extension并确认PostGIS版本

psql 连接数据库，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 | 3.4.0          | 3.4.0          | PostGIS geometry and geography
spatial types and functions
(1 row)
```

使用

关于PostGIS的使用,请参阅PostGIS3.4官方文档

## 2. pgvector

## 概述

## 原理介绍

NEFLAT和HNSW是PGvector的两个核心算法

## IVFFLAT

MPU4158的读写操作是通过I2C总线完成的，具体操作流程如下：先通过I2C总线向EEPROM写入命令，从而设置EEPROM的读写地址，之后通过EEPROM的读写命令读取EEPROM中的数据。EEPROM的读写命令由I2C总线的时序决定，如果想要读取EEPROM中的数据，那么I2C总线的时序应该为：先写入读命令，之后再写入地址，接着读取数据。如果想要写入EEPROM中的数据，那么I2C总线的时序应该为：先写入写命令，之后再写入地址，接着写入数据。通过简单的读写命令，但操作却可以在嵌入式系统中查询到很多的问题。对于所有的读写操作，如果想要读取EEPROM中的数据，那么I2C总线的时序应该为：先写入读命令，之后再写入地址，接着读取数据。如果想要写入EEPROM中的数据，那么I2C总线的时序应该为：先写入写命令，之后再写入地址，接着写入数据。

## HNSW

## 源码安装

1

源码安装

```
export PG_CONFIG=/usr/local/ivorysql/ivorysql-4/bin/pg_config
```

• `std::vector<pg_vector>()`

```
git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git
```

- 安裝 projector

```
cd pgvector

sudo --preserve-env=PG_CONFIG make
sudo --preserve-env=PG_CONFIG make install
```

· 但pgvector扩展

```
[ivorysql@localhost ivorysql-4]$ psql
psql (17.6)
Type "help" for help.
```

```
ivorysql=# create extension vector;
CREATE EXTENSION
```

至此，pgvector扩展安装完成。更多用法，请参考 pgvector 文档

## Oracle兼容性



在IvySQL Oracle兼容模式下，pgvector扩展同样可以正常使用

## 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. PGroonga

### 概述

PostgreSQL 兼备了全文搜索引擎，也能处理大数据数据。要处理的文本中带有许多停用词（停用词是不计特性的文本），其功能和性能可能无法满足高性能查询的需求。

PGroonga 相比而言，它是一个 PostgreSQL 的全文搜索引擎。与 Groonga 相比，它的全文搜索引擎的全文检索引擎为 PostgreSQL 的数据库实现插件。Groonga 是一个优秀的开源搜索引擎，以其较快的速度和平滑的可部署性，尤其适合处理多语言文本。PGroonga 的使用也是对 Groonga 的强大功能直接地融入 PostgreSQL 的使用。为用户提供高性能全文搜索的能力。

### 安装

PGroonga 支持通过 PostgreSQL 官方仓库安装，或者通过源码安装。

### 源码安装

PGroonga 的安装方式以tar.gz 包的形式提供，安装时需要使用 Ubuntu 20.04 或者 21.04。



环境已安装了 PostgreSQL 14 及以上版本。安装命令为 /usr/local/ivorysql/pgroonga-4

## 安装 groonga

· 安装依赖库

```
git clone https://github.com/neubig/kytea.git
autoreconf -i
./configure
make
sudo make install
```

· 安装依赖 libmq

```
从https://github.com/zeromq/libzmq/releases/tag/v4.3.5 下载zeromq-4.3.5.tar.gz
tar xvf zeromq-4.3.5.tar.gz
cd zeromq-4.3.5/
./configure
make
sudo make install
```

· 下载groonga源码并安装

```
wget https://packages.groonga.org/source/groonga/groonga-latest.tar.gz
tar xvf groonga-15.1.5.tar.gz
cd groonga-15.1.5
#运行这个脚本安装依赖，支持apt和dnf包管理工具
./ setup.sh
```

· 编译安装

```
# -S 指定groonga源码目录， -B 指定build目录，这个目录是个源码目录之外的一个只用于build的目录
cmake -S /home/ivorysql/groonga-15.1.5 -B /home/ivorysql/groonga_build
--preset=release-maximum
cmake --build /home/ivorysql/groonga_build
sudo cmake --install /home/ivorysql/groonga_build
# 更新动态链接库缓存
sudo ldconfig
```

· 配置groonga的环境

```
$ groonga --version
Groonga 15.1.5 [Linux,x86_64,utf8,match-escalation-threshold=0,nfkc,mecab,message-
pack,mruby,onigmo,zlib,lz4,zstandard,epoll,apache-
arrow,xxhash,blosc,bfloat16,h3,simdjson,llama.cpp]
```

## 安装 pgroonga

```
wget https://packages.groonga.org/source/pgroonga/pgroonga-4.0.1.tar.gz
tar xvf pgroonga-4.0.1.tar.gz
cd pgroonga-4.0.1
```

编译Groonga，有个选项 HAVE\_MSGPACK=1，它用于支持msgpack 1.4.1或者更高版本。在基于Debian的平台使用msgpack-dev包，在CentOS 7.2使用msgpack-dev

```
#安装依赖
sudo apt install libmsgpack-dev
```

运行make的输出log，config.log文件路径在PATENT变量里

```
make HAVE_MSGPACK=1
make install
```

## 创建Extension并确认PGroonga版本

psql 启动到数据库，运行如下命令：

```
ivorysql=# CREATE extension pgroonga;
CREATE EXTENSION

ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'pgroonga';
   name   | default_version | installed_version | comment
-----+-----+-----+
pgroonga| 4.0.1          | 4.0.1            | Super fast and all languages
supported full text search index based on Groonga
(1 row)
```

## 使用

关于PGroonga使用，请参阅 [Groonga使用指南](#)

## .4. pgddl (DDL Extractor)

### 概述

pgddl 是一个名为 PostgreSQL 的数据库设计的 SQL 扩展工具。它能够帮助数据库系统设计者完成数据、模式的 SQL DDL (数据定义语言) 配置，和 CREATE TABLE 或 ALTER FUNCTION。它加入了 PostgreSQL 项目中著名的 DDL CREATE TABLE 语句的功能，从而产生更高级的工具 (如 pg\_ddl)。同时为 SQL 语句中的数据对象的迁移提供了支持。

为了保证通过一些有用的 SQL 语句提供了更灵活的数据操作，其工作机制：从简单的 SQL 查询语句操作，支持通过 WHERE 子句过滤操作，从简单的数据对象到复杂的对象，从简单的 DDL 语句到复杂的语句。这些语句可以轻松地从 PostgreSQL 语句中分离出来，从而保证其逻辑行为完全一致。

### 安装



环境已安装了IvySQL 4.0.1及以上版本。安装路径为 /usr/local/ivorysql/ivorysql-4

### 源码安装

<https://github.com/lemon/pgddl/releases/tag/0.20.0> 下载 pgddl-0.20.0.tar.gz，解压缩。

```
cd pgddl-0.20
# 设置PG_CONFIG环境变量值为pg_config路径, eg: /usr/local/ivorysql/ivorysql-
```

```
4/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

pgsql: 读取配置文件失败。运行 pg\_ctl命令。

```
ivorysql=# CREATE extension dd1x;  
CREATE EXTENSION  
  
ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'dd1x';  
name | default_version | installed_version | comment  
-----+-----+-----+-----  
dd1x | 0.20 | 0.20 | DDL eXtractor functions  
(1 row)
```

## 使用

关于 pgRouting 的使用, 请参阅 [《pgRouting 文档》](#)

# 5. pgRouting

## 概述

pgRouting 是一个基于 PostgreSQL/PostGIS 的开源地理空间路由引擎。它不仅继承了强大的空间分析功能, 使其能够处理复杂的地理空间查询, 而且能够处理点对点的最短路径计算。它的主要优势在于能够处理大规模的数据集, 从而避免了在地理空间分析中进行复杂的内存操作。

该引擎的核心在于它能够利用 PostgreSQL/PostGIS 强大的数据处理能力和 PostGIS 扩展丰富的空间函数, 能够在数据库内直接对空间数据进行复杂的分析计算。这个优势使得它在地理学、物流、交通规划等领域得到了广泛的应用。pgRouting 项目已经成为了开源地理空间分析领域的一个重要工具。

pgRouting 广泛应用于物流配送、交通管理、网络分析、城市规划及供应链管理等多个领域。其独特的性能设计使其能够处理大规模的数据集, 使其成为空间数据处理和地理空间分析的首选工具之一。

## 安装



环境已安装了 PostgreSQL 9.4 及以上版本。安装命令为 `sudo apt-get install postgresql`。

## 源码安装

• 安装依赖

对 pgRouting 的安装依赖于 PostgreSQL 9.4 及以上版本。安装命令为 `sudo apt-get install postgresql`。

```
#安装依赖  
sudo apt install cmake libboost-all-dev
```

• 编译安装

```
wget https://github.com/pgRouting/pgRouting/releases/download/v3.5.1/pgRouting-  
3.5.1.tar.gz  
tar xvf pgRouting-3.5.1.tar.gz  
cd pgRouting-3.5.1  
mkdir build  
cd build  
cmake .. -DPOSTGRESQL_PG_CONFIG=/path/to/pg_config # eg:
```

```
/usr/local/ivorysql/ivorysql-4/bin/pg_config
```

```
make
```

```
sudo make install
```

创建Extension并确认ddlx版本

```
ivorysql=# CREATE extension pgrouting;
```

```
CREATE EXTENSION
```

```
ivorysql=# SELECT * FROM pg_available_extensions WHERE name = 'pgrouting';
```

| name      | default_version | installed_version | comment             |
|-----------|-----------------|-------------------|---------------------|
| pgrouting | 3.5.1           |                   | pgRouting Extension |

```
(1 row)
```

使用

关于pgRouting的使用, 请参阅 [pgRouting官方文档](#)

# IvorySQL架构设计

## 查询处理

### ..1. 双parser

的parser主要分为两部分，包括SQL语法和服务器端语法规则。

#### SQL端词法语法分离

基本语法规则是一套单独Oracle的语法规则，在开始解析时调用。在core扫描语法规则。在core中的语法规则，生成树的语法规则。

具体方法：在src/backend/parser/PostgreSQLParser.h中，修改core\_parser，修改core\_gram.y/ora\_scan.l，修改Oracle的语法规则和语法分析代码。同时对keywords进行兼容。两者对核心代码不相干。在oracle\_parser中修改是一个动态parser\_oracle，在开始Oracle兼容时，配置文件postgresql.conf中shared\_preload\_libraries="oracle\_parser"。这样之后原本的动态parser的插入oracle\_parser动态。

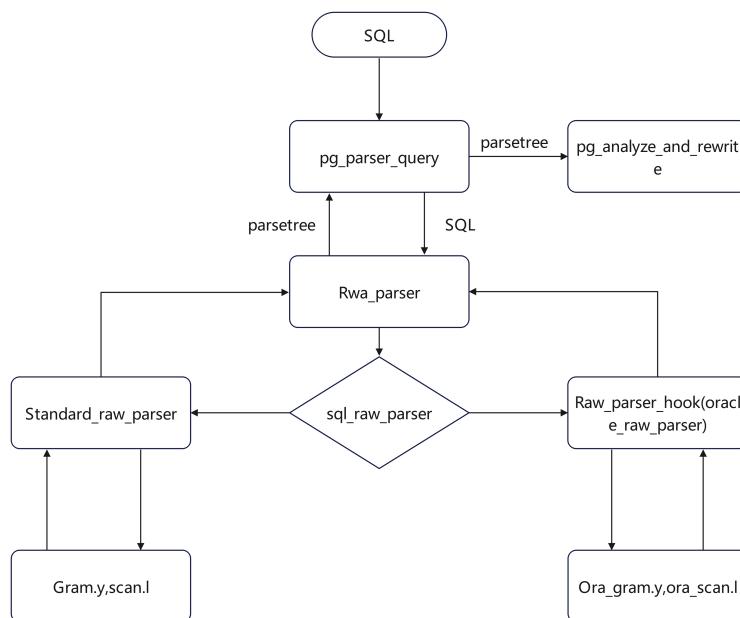
在src/include/parser.h中，修改parser\_oracle语法规则，修改core\_parser语法规则。在core中的语法规则，生成树的语法规则 ora\_parser。\_PG\_in()时在兼容模式下使用语法规则 ora\_parser 为分。

每一个后端语法规则在backend/parser目录，通过阅读后端代码可以知道 parser->currentMode，如果它是Oracle兼容模式，则设置currentMode为'ora'，否则设置为'pg'。

修改之后在Programmer指南中，如果parser->currentMode是'ora'，则调用SetConfigOption("pgsql\_compatible\_mode","ora");\_PG\_in();如果parser->currentMode是'pg'，则调用SetConfigOption("pgsql\_compatible\_mode","pgsql");\_PG\_in();同时设置assign\_hook，同时SetConfigOption("assign\_hook", 0);同时设置assign\_incompatible\_mode，从而设置sql\_ora\_parser + ora\_ora\_parser。

在对SQL语句进行分析时，通过pg\_parser\_query->ora\_parser (即standard parser) 或者 ora\_ora\_parser。

下面的图展示了SQL语句解析时的关系。



#### 服务器端编程语言词法语法分离

在系统类pg\_language中新增pgsql语法规则。

具体方法：将pg\_language中pgsql语法规则，复制一份，改名为pgsql，替换文件名pgsql，替换language头，因为pgsql是一个language，所以，pgsql language文件名应该为pgsql\_pg，替换language文件名应该为pgsql\_pg.h，替换language头文件名应该为pgsql\_pg.h。

修改语法规则为一个操作，通过修改语法规则宏pgsql\_pg，修改这个操作。这个操作会将pgsql语言在三处的数据源的头文件中。

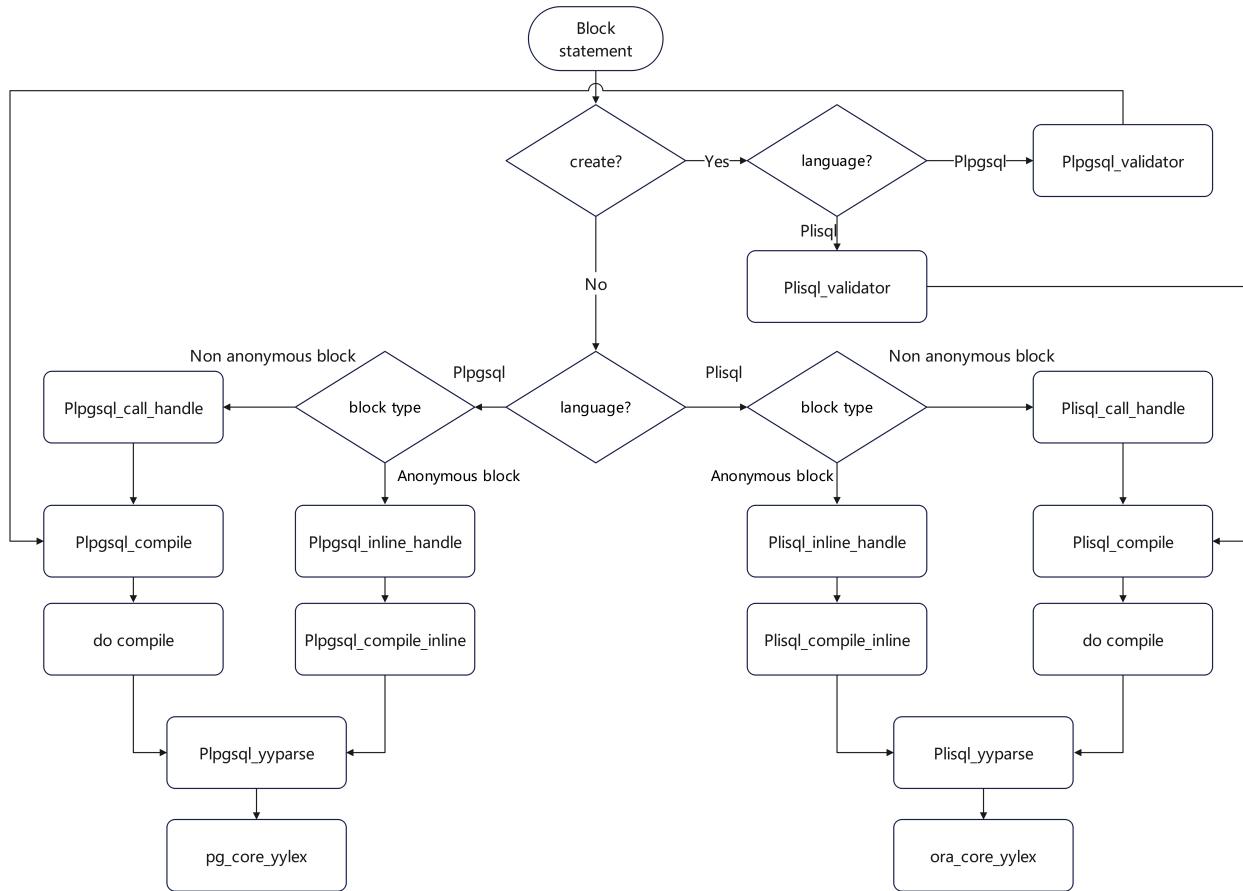
pgsql语言头文件，它会将pgsql语法规则宏pgsql\_pg，替换为pgsql\_pg，替换其他兼容Oracle的语法规则。主要修改点在pgsql\_scanner.h头文件中，替换的语句为pgsql\_scanner\_pg头文件，pgsql头文件的 internal\_pg.h 会替换为 ora\_pg.h。

pgsql语言头文件，如果使用pgsql\_pg头文件，兼容Oracle语法规则语法规则宏pgsql\_pg，兼容pgsql\_pg头文件。

pgsql语言头文件，没有使用pgsql\_pg头文件，如果使用pgsql\_pg头文件，兼容pgsql\_pg头文件，如果使用pgsql\_pg头文件，兼容pgsql\_pg头文件。

兼容pgsql\_pg头文件，如果使用pgsql\_pg头文件，兼容pgsql\_pg头文件，如果使用pgsql\_pg头文件，兼容pgsql\_pg头文件。

头文件修改为pgsql\_pg头文件，如果使用pgsql\_pg头文件，兼容pgsql\_pg头文件。



## 兼容框架

### ..1. initdb过程

PostgreSQL 从 9.1 版本起已将 initdb 命令从 PostgreSQL 安装包中移除。

• PG 模式：保留与原 PostgreSQL 兼容性

• Oracle 模式（默认）：提供 Oracle 语法规则及兼容性

用户可以通过 initdb 命令参数指定初始化模式。安装子系统时将使用兼容性模式。

### 参数解析处理

initdb 会根据命令行参数输入的命令参数。

| 参数 | 可选模式                 | 可选值                          | 默认值         |
|----|----------------------|------------------------------|-------------|
| -c | 指定兼容模式               | oracle/pgsql                 | oracle      |
| -C | 设置 Oracle 默认的大小写转换模式 | interchange/normal/uppercase | interchange |

参数解析流程：

1. 检查 PostgreSQL 和 Oracle 兼容性限制

2. 处理兼容性转换参数 -c 的解析逻辑

3. 处理大小写转换参数 -C 的处理逻辑

### 文件路径初始化

执行 set\_db\_data\_file\_path(); 函数完成文件路径初始化。

```

if (DB_PG == database_mode)
    set_input(&bki_file, "postgres.bki");
else
    set_input(&bki_file, "postgres_oracle.bki");
    
```

路径设置逻辑：

1. 创建 SQL 文件的脚本包 (postgres\_create.sql / postgres.sql)

2. 调用配置文件脚本包的可执行文件

3. 建立与数据库模式对应的系统组件对象

## 数据目录初始化流程

通过 `initdb_data_directory()` 函数完成核心初始化操作。

### 目录结构创建

调用 `create_data_directory()` 函数生成目录 (PGDATA)。

通过 `create_log_dir_symlink()` 建立 WAL 日志目录。

同时创建 `base_global` 等初始子目录。

### 配置文件初始化

调用 `create_pgconf()` 函数生成配置文件 (postgresql.conf)。

Oracle 模式下脚本的 `postgresql.conf` 是空文件。

调用 `create_pgconf()` 将配置参数写入 `postgresql.conf`。左写入配置时, 如果是 Oracle 模式, 则会将外层 `postgresql.conf` 中写入配置参数。

### 模板数据库引导

执行 `bootstrap_template()` 生成与 Oracle 模式对应的 SQL 文件初始化 `template0` 模板数据库。

调用 `create_pgconf()` 将数据库模式对应的配置写入 `template0.conf`。

调用 `load_pgsql()` 读取兼容 Oracle PL/SQL 的 PL/SQL 语言语句。

调用 `load_pgsql()` 读取 Oracle 语言语句。

调用 `create_pgsql()` 生成兼容 Oracle 语言语句。

### Oracle兼容用户名处理

当数据存储模式为 Oracle 时, 为用户名 (行存模式) 与转换。

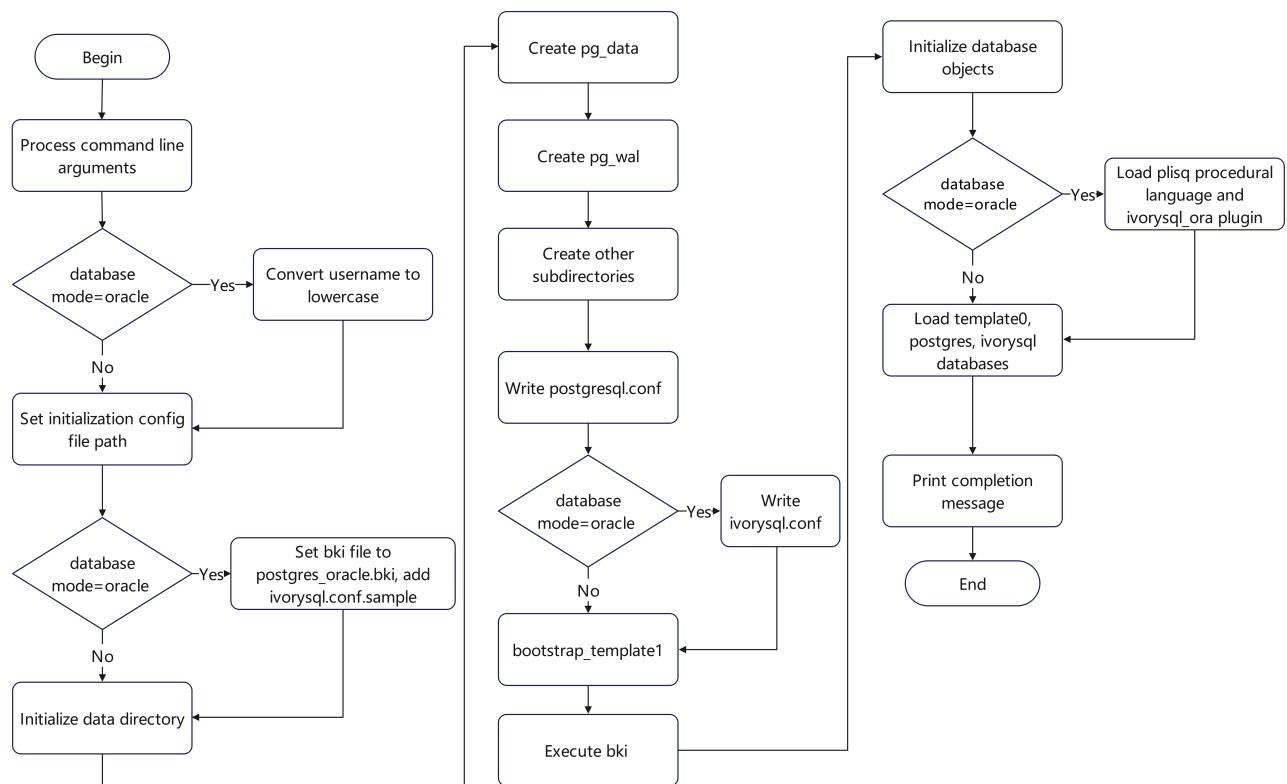
## 提示

在 PostgreSQL 中, “\i” 是 “Bootstrap AI” 的缩写, Bootstrap AI 是 PostgreSQL 内部命令符 (\bootstrap) 数据库系统的一组工具和文件集。

Bootstrap AI 主要用于初始化 PostgreSQL 数据库系统的各种组件, 包括系统目录 (system catalog)、系统表 (system table) 以及其他必要的系统对象。这些表的结构和数据存储于 PostgreSQL 数据库系统中的系统表, 包括系统对象、参数、角色、权限等。

gen�i.sql 是 Oracle 表文件 (如表 `class.hsqc_name_space.hsqc_procedure`) 从 Oracle 表文件 (如表 `class.hsqc_name_space.hsqc_procedure`) 中生成的。主要生成 Oracle 表的结构, 包括表名、表的字段和约束等。

在 install 目录下有脚本文件 `gen�i.sql`。如果在 PostgreSQL 中运行 `gen�i.sql` 语句, PostgreSQL 会将 Oracle 表文件 (如表 `class.hsqc_name_space.hsqc_procedure`) 中的表结构和数据插入 PostgreSQL 数据库中。

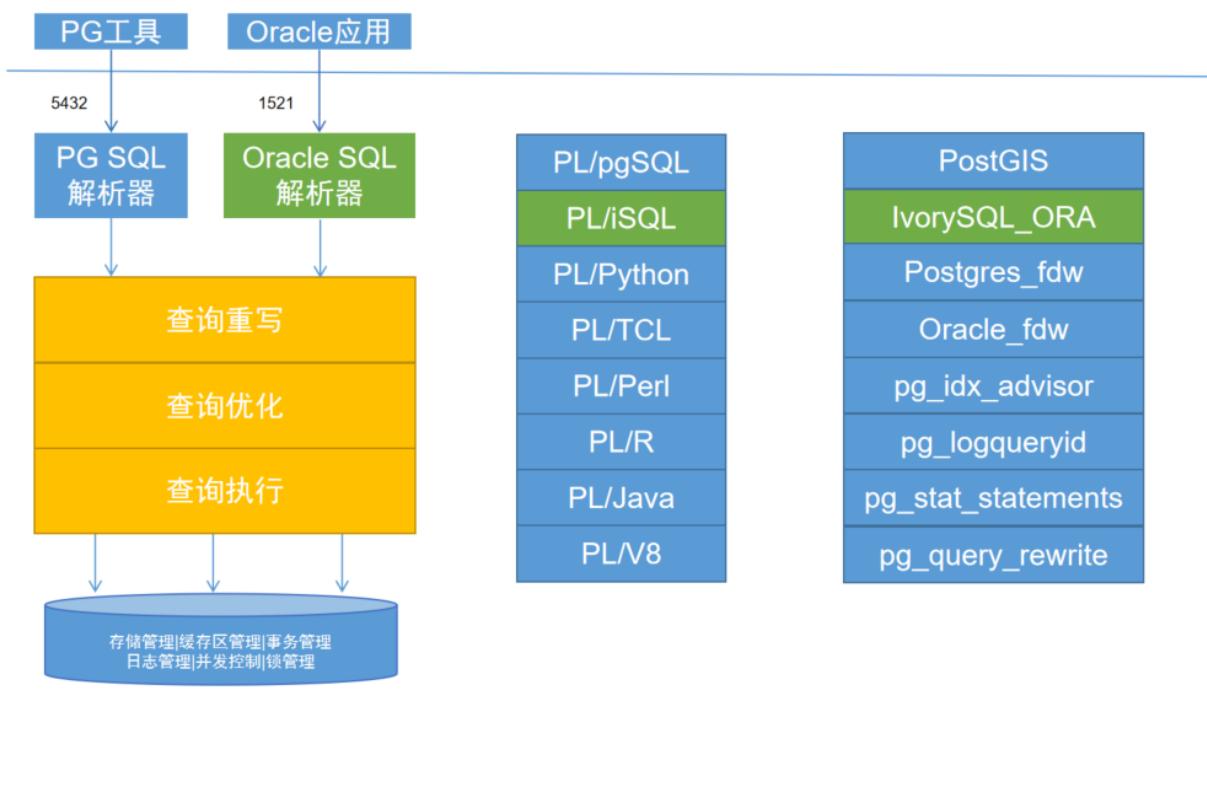


# Oracle兼容功能列表

## .1. 1、框架设计

### 目的

· 在对原有的PostgreSQL兼容最小的前提下，实现对Oracle兼容，从而满足企业在Oracle、双端口、模式PL/pgSQL兼容PL/SQL的需求，实现本地应用；



### 功能

#### 双端口设计

· 为了用Oracle语法对PostgreSQL语句进行操作并返回结果，因此，IvorySQL实现一个独立的端口号，认为是1521。从本地口连接，所以采用Oracle兼容模式。如果需要使用Oracle端口号连接到PostgreSQL，你需要设置`LISTEN=1521,COMPATIBLE_MODE=oracle`进行设置。

#### parser模块设计

· 需要PL/SQL语句兼容PostgreSQL语句并返回结果。

#### 新增PL/iSQL过程语言

· 新增PL/SQL过程语言兼容PostgreSQL语句。

## .2. 2、GUC框架

### 新增GUC变量

为了兼容Oracle，需要在原有的GUC变量基础上增加一些用于对兼容语句进行控制的变量，以达到对Oracle进行一致的目的。

为了以后更好的对兼容语句进行控制，以及为了更好的对Log语句的控制，我们再设计一个控制pgsql兼容的一个统一的地方。

### 实现

为了兼容Oracle语句时，我们通过一个 `ivory_guc.c` 文件来实现。其中 `ivory_guc.h` 文件中存放了 `ivory_guc_variableSet`, `ivory_guc_variableGet`, `ivory_guc_variableString`, `ivory_guc_variableBoolean` 分别表示对不同类型的pgsql参数。当pgsql的参数GUC进行修改时，

## 新增变量 (目前)

| 变量名                                | 描述  |
|------------------------------------|---|
| ivorysql.compatible_mode           | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle, oracle) 可以设置不同的兼容模式。 |
| ivorysql.database_mode             | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| ivorysql.datetime_ignore_nls_mask  | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| ivorysql.datetime_employing_no_NUL | 设置为 (mysql, oracle) 时, 通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。                            |
| ivorysql.client_for_use_switch     | 设置为 (mysql, oracle) 时, 通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。                            |
| ivorysql.listen_address            | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| ivorysql.port                      | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| nsi_date_format                    | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| nsi_length_attributes              | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| nsi_timestamp_format               | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| nsi_timestamp_no_format            | 表示当设置为Oracle模式 (oracle) 时, 通过ivorysql兼容模式。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。         |
| shared_preload_libraries           | 在处理大数据时, 从ivorysql.conf文件中读取。通过设置为不同的模式 (mysql, oracle, oracle, oracle) 可以设置不同的兼容模式。                      |

## 部分变量使用示例

```
ivorysql.datetime_ignore_nls_mask
```

通过设置为不同的模式 (mysql, oracle) 可以设置不同的兼容模式。

| 序号 | 兼容模式                                  |
|----|---------------------------------------|
| 0  | 不经过转换的类型                              |
| 1  | 不转换为时类型, 有时时间将无法自动格式化                 |
| 2  | date                                  |
| 3  | timestamp                             |
| 4  | date, timestamp                       |
| 5  | timestamp                             |
| 6  | date, timestamp                       |
| 7  | timestamp, timestamp                  |
| 8  | date, timestamp, timestamp            |
| 9  | timestamp                             |
| 10 | date, timestamp                       |
| 11 | timestamp, timestamp                  |
| 12 | date, timestamp, timestamp            |
| 13 | timestamp, timestamp, timestamp       |
| 14 | timestamp, timestamp, timestamp       |
| 15 | date, timestamp, timestamp, timestamp |

\* 使用方法 (以date为例)

查看ivorysql.datetime\_ignore\_nls\_mask

```
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\_date\_format

```
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)
```

修改 nls\_date\_format 为 'yy-mm-dd'，插入成功，显示为 15/04/24，插入失败，显示为 24-04-15，插入成功，显示为 15/04/24。

```
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)
```

### .3.3、大小写转换

#### 目的

为了更好的兼容不同的数据库，提高兼容性，ivorysql 为了支持不同的数据库，提供了参数 "datestyle\_case\_sensitive"，设置为 true。

## 功能

大小写转换的三种模式（默认为interchange）

· 如果pgsql参数“identifier\_case\_switch”设为“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pgsql参数“identifier\_case\_switch”设为“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pgsql参数“identifier\_case\_switch”设为“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程序中输入，`–c`选项设置大小写转换模式。C列的值为：

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

```
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

```

## 4.4、双模式设计

### 目的

为了满足PG模式和兼容Oracle模式，ivorysql在initdb时，可以在同一步骤，兼容PG模式又兼容Oracle兼容模式，不致冲突。



1 不指定一步骤时，假以为Oracle兼容模式  
2 如果一步骤为pgsql，根据次序判断兼容Oracle语法

### 功能

• initdb -m 初始化，需要特别注意模式，如果Oracle模式下，需要写成psql或者createDB的SQL语句。

• 同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`

## 5.5 兼容Oracle like

### 目的

本文档旨在为使用 like 模糊查询的人提供一个深入了解 MySQL 的模糊查询语句的过程，以及如何实现之。

### 功能说明

| 数据库名称  | like模糊查询   |
|--------|--|
| oracle | oracle对于像类型为varchar2,支持对数字、日期、字符串等类型的数据进行模糊匹配,但是不支持像MySQL一样支持通配符%和_。                   |
| MySQL  | MySQL对于字符串类型使用wildcard,所有的like语句都必须使用like,其他MySQL的语句使用直接的like语句,这样不能让operator识别出自己的语句。 |

### 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)
```

## 6.6 兼容Oracle匿名块

### 目的

本文档通过PLSQL匿名块(anonymous block)兼容Oracle匿名块的设计风格。目的在于在MySQL中兼容Oracle的匿名块语句。

### 功能说明

首先从兼容Oracle的块语句的行级PLSQL语句开始，然后再从语句的语句块语句开始。

### 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

```

## 7.7 兼容Oracle函数与存储过程

### 目的

• 本文档是在兼容Oracle PL/SQL函数和存储过程的基础上，将MySQL中的一些特性移植为PL/SQL语句。

### 功能说明

#### 函数 (FUNCTION)

```

CREATE FUNCTION语句支持OPTIONAL_NODITIONABLE
CREATE FUNCTION语句支持RETURN类型, 不同于language
CREATE FUNCTION语句函数的参数, 函数名(参数个数)
CREATE FUNCTION语句(参数多是必须)
CREATE FUNCTION语句END; 语句必须以分号结束
CREATE FUNCTION语句变量声明的语句必须以CLARE关键字
CREATE FUNCTION语句支持NOCOPY语句
CREATE FUNCTION语句支持RETURNING clause
CREATE FUNCTION语句支持OWNER_rights_clause, 则以指定的语句 (DEFINER)
CREATE FUNCTION语句ACCESSIBLE BY
CREATE FUNCTION语句DEFAULT COLLATION
CREATE FUNCTION语句支持NOTICE_clause
CREATE FUNCTION语句支持PROGRESSIVE_clause
CREATE FUNCTION语句支持RETURNING_clause
CREATE FUNCTION语句支持RETURN_ROWS_clause
ALTER FUNCTION语句
函数长字符串支持转义的规则

```

#### 存储过程 (PROCEDURE)

```

CREATE PROCEDURE语句支持OPTIONAL_NODITIONABLE
CREATE PROCEDURE语句函数的参数, 函数名(参数个数)
CREATE PROCEDURE语句(参数多是必须)
CREATE PROCEDURE语句(参数多是必须)
CREATE PROCEDURE语句支持NOTICE_clause
CREATE PROCEDURE语句支持OWNER_rights_clause
CREATE PROCEDURE语句ACCESSIBLE BY
ALTER PROCEDURE语句
存储过程支持参数, 语句支持分号 ;
存储过程支持参数 ;
在PL/SQL中使用存储过程, 可以使用CALL, 直接使用存储过程名字
支持“和”“或”两种连接方法

```

### 测试用例

```

CREATE or replace FUNCTION ora_func RETURN integer AS
BEGIN
    RETURN 1;
END;
/
CREATE OR REPLACE FUNCTION test_nocopy(a IN int, b OUT NOCOPY int, c IN OUT NOCOPY
int)
RETURN record

```

```
IS
BEGIN
    b := a;
    c := a;
END;
/
```

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

```
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;
/
```

## 8.8、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         |
| row                            |
| long                           |

### 内置函数类型

|            |
|------------|
| sysdate    |
| sysdateat  |
| add_months |

|                   |
|-------------------|
| last_day          |
| next_day          |
| months_between    |
| current_date      |
| current_timestamp |
| new_time          |
| to_offset         |
| trunc             |
| math              |
| substr            |
| substrs           |
| item              |
| item              |
| item              |
| length            |
| lengths           |
| replaces          |
| replace           |
| regexp_replace    |
| regexp_substr     |
| regexp_like       |
| to_number         |
| to_char           |
| to_date           |
| to_timestamp      |
| to_timestamp_tz   |
| to_datetime       |
| to_datetime       |
| numtordinal       |
| numtointerval     |
| locationtimestamp |
| from_id           |
| sys_extract_utc   |
| sessiontimezone   |
| hostos            |
| uid               |
| USERENV           |
| active            |
| to_date_type      |
| to_single_type    |
| compose           |
| decompose         |

## 内置函数说明

1. #!/bin/python3  
#功能：查看对今日和昨天时间，测试时间为：2023-07-06

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

```
-----
2023-07-06
(1 row)
```

查询往前提及的日期

```
select sysdate()-1 from dual;
?column?
```

```
-----
2023-07-05
(1 row)
```

2. #!/bin/python3  
#功能：显示本机的系统上当前系统时间戳时间（经过转换后显示），测试时间为：2023-07-06 10:18:31.674322 +08:00

```
select systimestamp() from dual;
systimestamp
```

```
-----
2023-07-06 10:18:31.674322 +08:00
```

(1 row)

3. 例句: select add\_months(sysdate(),1) from dual; 返回值: 2023-08-06

```
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第一天:

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

-----  
2023-06-06  
(1 row)

4. 例句: select last\_day(sysdate()); 返回值: 2023-07-31

```
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)

5. 例句: select next\_day(sysdate(),1) from dual; 返回值: 2023-07-07

```
select next_day(sysdate(),1) from dual;  
next_day
```

-----  
2023-07-07  
(1 row)

查询当前日期的下一个星期五:

```
select next_day(sysdate(),'FRIDAY') from dual;  
next_day
```

```
2023-07-07
```

```
(1 row)
```

6. #@months\_between函数, 功能: 返回日期类型的date和date2之间的月份, 支持参数: date,date; 返回: 如果date1大于date2, 返回正值; 如果date1等于date2, 返回0; 如果date1小于date2, 返回负数; 如果date1是单月的前一天, 返回结果为0; 如果date1是单月的最后一天, 返回结果为1; 其他月份返回0; 调用示例: 查询两个日期之间相差的月份;

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

```
months_between
```

```
-----  
-1
```

```
(1 row)
```

查询两个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)
```

1. #@current\_date函数, 功能: 返回当前时间对应的日期, 支持参数: 无; 返回: 当前时间对应的日期;

```
select current_date from dual;
```

```
current_date
```

```
-----  
2023-07-06
```

```
(1 row)
```

1. #@current\_timestamp函数, 功能: 返回当前时间对应的日期和时间, 支持参数: 无; 返回: 当前时间对应的日期和时间;

```
select current_timestamp from dual;
```

```
current_timestamp
```

```
-----  
2023-07-06 10:27:01.440600 +08:00
```

```
(1 row)
```

1. #@current\_timestamp(3)函数, 功能: 返回当前时间对应的日期和时间, 支持参数: date, int,int; 返回: 当前时间对应的日期和时间, 小数位数为3;

```
select current_timestamp(3) from dual;
```

```
current_timestamp
```

```
-----  
2023-07-06 10:27:14.182000 +08:00
```

```
(1 row)
```

1. #@rowid函数, 功能: 返回当前行的唯一标识符, 支持参数: date, int,int; 返回: 一个唯一的行ID;

```
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)
```

11. 表达式new\_time, 用途: 返回指定时区和基准时区的偏移量. 支持参数: int, 表达式语句; 返回结果时区与基准时区相同.

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

11. 表达式new\_time, 用途: 可以返回时区, 例如想要的数据, 例如, 月, 日, 时, 分, 支持参数: date/datetime, 表达式语句; 返回结果时区与基准时区相同.

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

11. 表达式trunc, 用途: 表达式语句是正确的, 并且这个是正确的语句.

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

11. 表达式trunc, 用途: 表达式语句是正确的, 并且这个是正确的语句.

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

11. 表达式instrb, 用途: 字符串匹配函数, 返回字符串的位置. 支持参数: varchar2, text, number DEFAULT 1, number DEFAULT 1, 用于匹配的表达式; INSTRB(CORPORATE FLOOR', 'OR') 第一个OR的字符所在的位置.

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

2

(1 row)

SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes" FROM DUAL;  
Instring in bytes

14

(1 row)

13. 嵌套SUBSTR函数. 问题: 需要字符串函数, 以字符串为单位截取. 支持参数: instring, 要截取的子字符串: '很好', 今天天气很好' 中以第一个字符开始, 以最后一个字符结束.

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

Substring with bytes

很好

(1 row)

14. 嵌套SUBSTRB函数. 问题: 需要字符串函数, 以字符串为单位截取. 支持参数: varchar2, number/varchar2, number, 要截取的子字符串: '很好', 今天天气很好' 中以第一个字符开始, 以最后一个字符结束.

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

天气很好

(1 row)

15. 嵌套SUBSTRB函数. 问题: 需要字符串函数, 以字符串为单位截取. 支持参数: varchar2, number/varchar2, number, 要截取的子字符串: '很好', 今天天气很好' 中以第一个字符开始, 以第八个字符结束(字符串: '今天天气很好')

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

天气

(1 row)

16. 嵌套TRIM函数. 问题: 需要字符串函数, 以字符串为单位截取. 支持参数: varchar2/varchar2, varchar2, 要截取的字符串: 'aaa bbb ccc', 'aaa bbb ccc' 中去掉首尾空格.

select trim(' aaa bbb ccc ')trim from dual;

trim

aaa bbb ccc

(1 row)

17. 嵌套TRIM函数. 问题: 需要字符串函数, 以字符串为单位截取. 支持参数: varchar2/varchar2, varchar2, 要截取的字符串: 'aaa bbb ccc', 'aaa' 中去掉首尾空格.

select trim('aaa bbb ccc','aaa')trim from dual;

```
trim
```

```
-----  
bbb ccc  
(1 row)
```

11. 例題10(trim, 方問) : Oracle文字列函数の最初の空格を削除する例題。条件: varchar2 varchar2 varchar2(10) DEFAULT '  '; 関数: dbms\_output.enable(100);

```
select ltrim('  abcdefg  ')ltrim from dual;  
ltrim
```

```
-----  
abcdefg  
(1 row)
```

11. 例題11(trim, 方問) : Oracle文字列函数の最初の空格を削除する例題。条件: varchar2 varchar2 varchar2(10) DEFAULT '  '; 関数: dbms\_output.enable(100);

```
select ltrim('abcdefg','fegab')ltrim from dual;  
ltrim
```

```
-----  
cdefg  
(1 row)
```

11. 例題12(trim, 方問) : Oracle文字列函数の最初の空格を削除する例題。条件: varchar2 varchar2 varchar2(10) DEFAULT '  '; 関数: dbms\_output.enable(100);

```
select rtrim('  abcdefg  ')rtrim from dual;  
rtrim
```

```
-----  
abcdefg  
(1 row)
```

11. 例題13(trim, 方問) : Oracle文字列函数の最初の空格を削除する例題。条件: varchar2 varchar2 varchar2(10) DEFAULT '  '; 関数: dbms\_output.enable(100);

```
select rtrim('abcdefg','fegab')rtrim from dual;  
rtrim
```

```
-----  
abcd  
(1 row)
```

11. 例題14(length), 方問) : Oracle文字列函数の長さを計算する例題。条件: char varchar2(20) DEFAULT '  '; 関数: dbms\_output.enable(100);

```
select length(223) from dual;  
length
```

```
-----  
3  
(1 row)
```

查詢223' 的字節長度

```
select length('223') from dual;
```

```
length
```

```
-----  
3  
(1 row)
```

查询“ivoryysql”字符串的字符串长度：

```
select length('ivoryysql数据库') from dual;
```

```
length
```

```
-----  
11  
(1 row)
```

23. 声明lengthb函数：只能指定字符串类型的参数，支持参数：char/bytea/varchar2/nvarchar2；查询“ivorysql”字符串长度：

```
select lengthb('ivorysql'::char) from dual;
```

```
lengthb
```

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

查询“d”字符串的字符串长度：

```
select lengthb('0x2C'::bytea) from dual;
```

```
lengthb
```

```
-----  
4  
(1 row)
```

查询“ivoryysql”字符串的字符串长度：

```
select lengthb('ivoryysql数据库'::varchar2) from dual;
```

```
lengthb
```

```
-----  
17  
(1 row)
```

25. 声明replace函数：只能指定字符串类型的参数，支持参数：text/text/text/varchar2/varchar2/varchar2/nvarchar2/nvarchar2；查询“jack and jue”字符串替换为“black and blue”：

```
select replace('jack and jue','j','bl') from dual;
```

```
replace
```

```
-----  
black and blue
```

(1 row)

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

ack and ue  
(1 row)

```
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;
```

0\*#\*#\*#\*#abcd\*#\*#\*#\*#\*#

```
select regexp_replace('01234abcd56789','01')from dual;
```

## regexp\_replace

234abcd56789

(1 row)

用“`xxo`` 替换`01234abcd56789`` 中的`012`

```
select regexp_replace('01234abcd56789','012','xxx')from dual;
```

xxx34abcd56789  
(1 row)

22. 黑客 regex\_replace函数：功能：替换字符串中正则表达式描述的字符串。支持参数：text, text,integer /text, text, integer, integer /text, text, integer, integer, text, varchar2, varchar2, varchar2,测试例例如：T：查询'0322ab34'中从第一个数开始的0322字符串；

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

012  
(1 row)

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

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

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

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

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

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

## regexp\_instr

7  
(1 row)

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

7  
(1 row)

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

7  
(1 row)

```
SELECT regexp_instr('数据库', '库');
```

3

```
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');
```

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

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

```
2. 使用to_number函数：功能：将字符串转换为指定类型的数。支持参数：to_number函数的执行语句：字符串 '34,338,492' 转换为数值类型。
SELECT to_number('34,338,492', '99,999,999') from dual;
to_number
-----
-34338492
(1 row)
```

```
3. 使用to_number函数：功能：将字符串转换为浮点类型。支持参数：to_number函数的执行语句：字符串 '5.01-' 转换为浮点类型。
SELECT to_number('5.01-', '9.99S');
to_number
-----
-5.01
(1 row)
```

```
4. 使用to_char函数：功能：将数字转换为字符串类型。支持参数：to_char函数的执行语句：将系统日期转换为字符串。
select to_char(sysdate()) from dual;
to_char
```

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

将为当前日期转换为字符串(日期字符串格式):  
-----  
select to\_char(sysdate(),'mm/dd/yyyy') from dual;  
to\_char

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

将为当前日期转换为字符串(日期字符串格式):  
-----  
SELECT to\_char(sysdate()::timestamp);  
to\_char

```
-----  
2023-07-10 09:46:44.000000
```

将为当前日期转换为字符串(日期字符串格式):  
-----  
SELECT to\_char(sysdate()::timestamp,'MM-YYYY-DD');  
to\_char

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

将为当前日期转换为字符串(日期字符串格式):  
-----  
select to\_date('20230706') from dual;  
to\_date

```
-----  
2023-07-06  
(1 row)
```

将为当前日期转换为字符串(日期字符串格式):  
-----  
SELECT to\_date('-44,0201','YYYY-MM-DD');  
to\_date

```
-----  
0044-02-01  
(1 row)
```

将为当前日期转换为字符串(日期字符串格式):  
-----  
SELECT to\_timestamp('20181102.12.34.56.025');

```
to_timestamp
```

```
-----  
2018-11-02 12:34:56.025000  
(1 row)
```

```
-----  
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)
```

```
-----  
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)
```

```
-----  
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)
```

```
-----  
select to_date('20110101','yyyy-mm-dd')+to_yTimeInterval('02-08') from dual;  
?column?
```

```
-----  
2013-09-01  
(1 row)
```

```
-----  
select sysdate()+to_dTimeInterval('0 09:30:00')as newdate from dual;  
newdate
```

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

13. SELECT numtodsinterval(100.00, 'hour'); 功能：将数字转换为时间间隔，支持参数：double precision, text(时间间隔)；返回类型：interval DD, HH:MM:SS

```
SELECT NUMTODSINTERVAL(100.00, 'hour');
numtodsinterval
-----
+000000004 04:00:00.000000000
(1 row)
```

时间间隔转换为时间戳

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

13. SELECT numtodyminterval(1.00,'year'); 功能：将数字转换为时间间隔，支持参数：double precision, text(时间间隔)；返回类型：interval DD, YYYY:MM:DD

```
SELECT NUMTOYMINTEGER(1.00,'year');
numtodyminterval
-----
+000000001-00
(1 row)
```

时间间隔转换为时间戳

```
SELECT NUMTOYMINTEGER(1,'month');
numtodyminterval
-----
+000000000-01
(1 row)
```

14. SELECT localtimestamp(1); 功能：返回本地时间戳，支持参数：integer, 日期中指定参数为精确，测试语句：SELECT localtimestamp(1);

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

返回本地时间戳(1)精确到纳秒

```
select localtimestamp(1) from dual;
localtimestamp
-----
2023-07-07 09:18:16.100000
```

(1 row)

25. #@from\_tz@, 函数: 将一个时间转换为另一个时区, 例如将timestamp转为EST: FROM\_TZ(timestamp, 'EST')@#09:00#;

```
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)
```

30. #@sys\_extract\_utc@, 函数: 将一个timestamp转换为UTC@#09:00#; 例如将timestamp with time zone 转为UTC: SYS\_EXTRACT\_UTC(timestamp with time zone);@#09:00#;

```
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)
```

35. #@sessiontimezone@, 函数: 查看当前的时区, 例如SELECT: sessiontimezone;

```
select sessiontimezone() from dual;  
sessiontimezone
```

```
-----  
Asia/Shanghai  
(1 row)
```

40. #@timezone@, 查看可用时区;

```
set timezone = 'Asia/Hong_Kong';  
SET
```

```
select sessiontimezone() from dual;  
sessiontimezone
```

```
-----  
Asia/Hong_Kong  
(1 row)
```

45. #@hextoraw@, 函数: 将字符串转换为一个hex数据, 例如SELECT: hex('abcdef');@#09:00#;

```
select hextoraw('abcdef')from dual;  
hextoraw
```

```
-----  
\abcdef  
(1 row)
```

50. #@hex@, 将一个hex值转换为字符串, 例如SELECT: hex('abcdef');@#09:00#;

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

41. 使用SQL语句查询，判断：是否当前数据库中的表，对应的列：查看当前用户是否是dual，如果是dual，则

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

查看系统表：

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

42. 使用SQL语句，判断：外连接，返回的列名称，如果相同：只返回一个：

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

只返回一个：

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

只返回一个：

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

43. 使用SQL语句，判断：将字符串中的非字母字符替换成空格；输入字符串：转换为字符串：

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

43. 使用TO\_MULTI\_BYTE函数, 将一个字符串中的所有字符替换成全角字符, 将点与半角字符

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

44. 使用TO\_SINGLE\_BYTE函数, 将点与半角字符替换成一个普通的单字节, 输入基本字符, 输出为全角字符

```
select compose('a'||chr(768)) from dual;
      compose
-----
à
(1 row)
```

45. 使用DECOMPOSE函数, 将一个非ASCII字符(含有两个字节的字符)分解为其基本字符和组合字符, 输入非基本字符, 输出为组合字符

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

## 9.9、新增Oracle兼容模式的端口与IP

目的

• 为了用Oracle端口, 需要将PGSQLTCP的端口修改, 需要修改ORAPORTORACLE的值。

功能

• 配置oraport: 需要在连接时使用参数oraport即可启动, 需要修改host。

• 配置oraport: 相对于host, port的值相对要多一些, 其中才反而可以configure修改配置, 为新的连接端口。

测试用例:

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

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

## 10.10、XML函数

### 目的

在Oracle 11g，你会发现有对XML的SQL函数，IvySQL与PL/SQL的功能上，实现了对Oracle 10g XML函数的兼容性，兼容了对Oracle 10g XML函数的兼容性。这种兼容性意味着用户可以将PL/SQL语句直接使用于IvySQL语句，从而保证了数据的完整性和准确性。此外，IvySQL还提供了对XML的兼容性，从而保证了对XML的兼容性和准确性，使得用户使用起来更加方便，可靠和安全。



XML (eXtended Markup Language) (扩展标记语言) 是一种基于文本的，用于结构化存储和检索数据的格式。它是一种轻量的，对扩展的，标准的且易于理解的文件数据的表示。

### 实现原理

IvySQL在实现与Oracle 12c中11个兼容的SQL函数的基础上，与PL/SQL保持了一致，同时兼容函数同样实现了与Oracle 10g函数的接口。这些XML函数作为IvySQL语句提供的一个子组件提供，兼容了与PL/SQL语句在XML处理方面的兼容性和一致性。

由于Oracle 11g的函数要求参数参数类型是XMLType，因此下面这个示例代码是正确的：

示例：CREATE OR REPLACE TYPE my\_iname XMLType INHERITS (String);

示例：SELECT my\_iname(XMLType('a')) FROM dual;

因为为了兼容，添加了一个 XMLType 的别名，不再是使用原生的字符串类型，让用户的SQL语句与PL/SQL保持一致。

另外，为了兼容与PL/SQL中已有的函数关键字“xmltype”，产生混淆，IvySQL将原来的函数关键字重命名为“XMLTYPE”，以便保证函数的兼容性和准确性。

在实现这11个Oracle兼容的SQL函数时，IvySQL采用了两种不同的实现方式。其中，除了UPDATETEXT函数，其他10个函数的实现方法与PL/SQL函数的方式进行相同。至于UPDATETEXT函数的参数数量是个例外，因此采用了两种不同的实现方式。以确保其功能的正确性和兼容性。

### 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子节点或子节点的值，其中参数XMLType参数为要提取XMLType的，XPath_string为要提取的XML节点 |
| 2  | extractvalue        | 该函数返回XML子节点的值，extractvalue函数返回一个值或一个值                                |
| 3  | extractable         | 该函数判断XMLType是否具有可提取的特性   |
| 4  | getvalue            | 该函数用于将XMLType转换为String，转换  |
| 5  | appendchild         | 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对象作为参数，并将XML对象作为子节点插入到XML对象        |
| 6  | update              | 该函数用于更新XML对象的子节点的值   |
| 7  | insertbefore        | 该函数用于在XML对象中插入子节点  |
| 8  | insertafter         | 该函数用于在XML对象中插入子节点  |
| 9  | insertroot          | 该函数用于将XML对象作为根节点插入到XML对象   |
| 10 | insertindeterminate | 该函数用于将XML对象插入到XML对象  |
| 11 | insertindeterminate | 该函数用于将XML对象插入到XML对象  |
| 12 | xmlval              | 该函数用于将XML语句返回值   |

### 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 xmptest(id int, data XMLType);
CREATE TABLE
ivorysql=# insert into xmptest 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 xmlinstest(id int, data xmltype);
CREATE TABLE
ivorysql=# INSERT INTO xmlinstest 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>'), '//*[@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)

## .11.11、兼容Oracle sequence

目的

• 本文档介绍MySQL兼容Oracle sequence功能，目的是可以在MySQL中使用OracleSequence功能。

功能说明

• 序列是一个数据对象，与表和视图无关。它表示可以由存储过程或函数访问的连续整数。序列可以设置为递增或递减。

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
```

## 12. 12、包

四的

MySQL提供了兼容Oracle自定义的功能。包 (package) 是一个封装的源码程序对集合，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、游标和序列。

## 功能说明

## 创建包规范

使用CREATE OR REPLACE PACKAGE语句创建或替换一个包的视图。它是相关的存储过程、函数、和其他程序对象的集合，在数据集中作为一个单元存储。何时若声明这些对象，在何时中定义这些对象。

## 创建包体

使用CREATE OR REPLACE PACKAGE BODY语句创建或替换一个包体。创建时你需要有与创建的包名一样的权限。要变的包的程序在同一个模式下，目的对象必须存在。该语句定义的包体中声明的对象

当的祖先中有 cursor 子子进程时，那么必须拥有一个线程来定义它，否则线程是可选的。

## 更新包

使用ALTER PACKAGE语句更改包的属性。但ALTER PACKAGE语句的权限要求：必须是包的所有者，或者具有ALTER ANY PROCEDURE权限才能操作其他用户的包。

## 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语句只删除包体。

不能使用此语句删除表中的一个普通对象。权限要求：你必须在用户模式下，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。

## 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)
```

### 13.13、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，将使数据体设计更加简洁，数据管理的灵活性得到提高，用户可以更好的控制列的可见性。

在它们经过之后，本应见到的景象却未见，使新到本区的观察者们还不能从任何方面看到，它们仍然可以被任何新的观察者们识别，从而使它们终生的交往也就更加频繁。

## 功能说明

## 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

```
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

\* 不能对不可见列使用 \d+ 命令。  
\* 在Oracle中，不可见列的插入操作会失败。

# 社区贡献指南

## IvorySQL 社区协作流程

IvorySQL 社区贡献者是一群热爱并贡献开源代码的人。他们从项目提出新版本需求，每一个贡献都可能涉及反馈、持续改进，这一过程帮助用户不断更新或优化产品。让社区贡献的回馈和贡献者的需求。

整个协作过程包括以下：

提问题 (Issue)

用户贡献者在 GitHub 的 Issues 页面提交 Bug，问题建议或优化反馈。

问题讨论 (Discussion)

维护者对社区贡献的反馈进行讨论，确认问题性质并优先级，加入 ToDo List。

开发分支 (Fork & Dev)

开发者以提 Issue、Fork 仓库并在本地测试，准备提交代码。

提交 Pull Request (PR)

维护者交付 PR 和 Fork 仓库后，线上进行真实的部署或测试。

代码评审 (Review)

维护者或核心贡献者对 PR 进行评审，提出修改建议并确认质量。

合并分支 (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)

维护者定期发布版本（每周两个版本，每季六个版本），包含最新的修复与功能。

代码 Merge (Pull)

用户在使用过程中，发现新问题，新的 Issue 便产生，形成完整的反馈循环。

### 提问题 Issue

用户或开发者在项目的 GitHub Issues 页面中提交问题

#### 用户测试 Test

用户下载并使用新版本，可能发现新的问题或提出新需求

#### 版本发布 Release

项目定期或按需发布新版本，每季度1个小版本，每年1个大版本

#### 代码 Merge

PR 通过审核后，由 Maintainer 合并进主分支，并关闭Issue

#### 问题讨论 Discussion

维护者 (Maintainer) 或社区成员对 Issue 进行讨论、确认BUG、优先级并加入 ToDo List

#### 开发分支 Fork & Dev

开发者领取Issue，Fork 原始仓库到自己的账号下，Clone 到本地完成代码编写和测试

#### 提交 PR

将本地分支 Push 到自己的 Fork 后，在 GitHub 上发起 PR 到原仓库的对应分支

#### 代码评审 Review

项目维护者或核心成员 Review PR



通过完整的协作闭环机制，IvorySQL 实现了问题识别 → 开发实现 → 质量保证 → 新功能的上线和迭代，驱动项目持续健康发展。

## IvorySQL 社区贡献指南

IvorySQL 是一个由热心开发者主导、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。如果您希望贡献代码或成为 IvorySQL 项目的核心成员，请参考以下指南。

参与贡献前，请确保您已经阅读并理解了贡献指南。

- 请仔细阅读贡献指南，理解建议，并遵守社区规则。
- 请认真阅读 README，熟悉项目结构以及如何提交 Issue 以及如何进行提交。
- 请参考 GitHub 常见问题，包括但不限于：代码评审和合并建议。

无论您以何种身份加入，IvorySQL 社区都将欢迎您的参与和支持！IvorySQL 社区欢迎贡献所有类型的贡献，期待您的加入！

请认真阅读并遵守我们的 IvorySQL 社区行为准则。

无论您以何种身份加入，IvorySQL 社区都将欢迎您的参与和支持！IvorySQL 社区欢迎贡献所有类型的贡献，期待您的加入！

请参考 <https://help.github.com/en/getting-started/walk-your-journey-through-github> 学习如何使用 GitHub，获取 GitHub 工具和工作流。

# 用户

作为用户，您使用 MySQL，迁移中的读者或贡献者。我们欢迎您：

## 反馈问题与需求

• 提交 bug，报告缺陷或文档不准确

• 提交拉取请求或更新现有拉取请求

如果您准备向社区上提交 bug 或者提交需求，请在 MySQL 社区讨论仓库上提交 issue，并参考 issue 模板说明。

## 参与社区讨论

• 通过邮件列表进行讨论

• 加入 GitHub 会议，讨论需求或锁定问题

• 与 社区，Discussions 参与技术交流

# 贡献者

我们欢迎代码、文档、测试等各类型贡献。

## 签署CLA

在贡献代码或文档时必须签署。为了确保代码或文档合法，个人或企业贡献者需要签署贡献者许可证(CLA)。签署CLA是MySQL社区接受贡献的必要条件。以确保您的贡献合法有效。请根据以下链接签署并签署后的CLA发送 [cla@mysql.org](mailto:cla@mysql.org)。

• 个人贡献者

• 企业贡献者

未签署CLA Pull Request将无法进入评审阶段。

## 找到您感兴趣的项目

我们列出贡献行为多个子项目，您可以从以下列表找到感兴趣的项目及其贡献仓库。

贡献仓库	描述
MySQL	负责MySQL数据库的开发和维护
MySQL tests	负责MySQL测试用例的维护和维护
MySQL connector	负责MySQL连接器及驱动工具的开发和维护
MySQL library	负责MySQL库的开发和维护
MySQL client	负责MySQL客户端及连接工具的开发和维护
MySQL_distro	负责MySQL发行版的开发和维护
MySQL_distro_builder	负责MySQL发行版构建脚本的维护
MySQL_fixes	负责MySQL社区中已知问题的修复
MySQL_patches	负责MySQL社区中已知问题的修复

## 给自己分配Issue

您可以将自己创建的Issue或者将他人创建的Issue分配给自己。只需要将它们加入Issue，或者将它们分配给他人。每个Issue都可以有多个分配给它。如果您的Issue，也可以在评论中将它分配给自己的贡献者或Issue Owner。

## 开发与提交Pull Request

对于提交一个PR拉取请求一个分支，或者一个tag提交一次。禁止多次提交一次提交。

## Fork仓库

前往贡献主页，点击Fork按钮，将MySQL贡献仓库fork到自己的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/MySQLSource/MySQL> (或 Submit 模板中的 GitHub ID)。

git checkout -b feature/your-feature-name

在提交代码前，请确保通过以下测试

## 创建一个Pull Request并提交

打开贡献仓库：<https://github.com/MySQLSource/MySQL>

点击 Compare & pull request，发起新的PR拉取请求。

Fix test  
功能描述

## leave a comment

对该提交功能进行比较详细的描述

点击Create pull request 立即即可提交。

## 维护者

维护者负责对MySQLQI代码的管理、PR审查、主存储库发布与MySQLQI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
- 维护公共的1000+列表

## 代码管理

- 使用 Pull Request 开放
- 修复安全问题，保障项目健康

## 流程与治理机制

- 优先级待审核（代码贡献指南、PR 审查等）
- 建立漏洞扫描机制与修复机制

## 致谢

感谢每一位参与 MySQLQI 的开发者、贡献者、测试人员和贡献者。正是因为有了你们的付出，MySQLQI 才能不断成长！我们感谢所有人为 MySQLQI 作出贡献。我们期待和渴望每一个贡献者对我们的支持、理解和支持。

# Chapter 1. 工具参考

## 工具一览表

这些部分将向您介绍工具的参考信息。没有特别指出的都是通用工具，适用于所有的数据库。这些部分的共同特征是它们可以运行在任何主机上，而不是限制在服务器上。

当在命令行上运行实用程序或脚本时，它们的大小将保留一些限制或字节的大小，但保留所有其他参数。

分类	工具名称	描述
客户端工具	clusterdb	clusterdb是一个工具，它将对一个tinySQL数据库进行操作和恢复。它会运行在已经连接到它的主机上对它进行操作。没有连接到它的主机将无法使用它。clusterdb是tinySQL命令cluster的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	createdb	createdb是一个tinySQL数据库的命令。通过这个命令可以创建新的数据库。如果只想使用地方方式，那么可以使用tinySQL命令create。如果只想使用tinySQL命令create，那么可以使用tinySQL命令CREATE DATABASE。create和createDB是同一个命令的两个别名。
	createuser	createuser是一个tinySQL数据库的命令。通过这个命令可以创建新的用户。如果只想使用地方方式，那么可以使用tinySQL命令CREATE USER。createuser和createDB是同一个命令的两个别名。在通过这个工具和地方方式对数据库进行操作时没有区别。
	dropdb	dropdb是一个tinySQL数据库的命令。通过这个命令可以删除数据库。dropdb是tinySQL命令DROP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	dropuser	dropuser是一个tinySQL数据库的命令。通过这个命令可以删除用户。如果只想使用地方方式，那么可以使用tinySQL命令DROP USER。dropuser和dropDB是同一个命令的两个别名。在通过这个工具和地方方式对数据库进行操作时没有区别。
	exit	exit是一个命令，它将退出tinySQL。它将退出所有的地方方式连接。如果只想使用地方方式，那么可以使用tinySQL命令QUIT。exit是tinySQL命令QUIT的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_arscheck	pg_arscheck是一个tinySQL数据库的命令。通过这个命令可以检查tinySQL的审计日志。如果只想使用地方方式，那么可以使用tinySQL命令ARSCHECK。pg_arscheck是tinySQL命令ARSCHECK的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_basebackup	pg_basebackup是一个tinySQL数据库的命令。通过这个命令可以备份数据库。如果只想使用地方方式，那么可以使用tinySQL命令BASEBACKUP。pg_basebackup是tinySQL命令BASEBACKUP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_dump	pg_dump是一个tinySQL数据库的命令。通过这个命令可以导出数据库。如果只想使用地方方式，那么可以使用tinySQL命令DUMP。pg_dump是tinySQL命令DUMP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_dumpall	pg_dumpall是一个tinySQL数据库的命令。通过这个命令可以导出所有数据库。如果只想使用地方方式，那么可以使用tinySQL命令DUMPALL。pg_dumpall是tinySQL命令DUMPALL的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_dumplo	pg_dumplo是一个tinySQL数据库的命令。通过这个命令可以导出一个大的对象。如果只想使用地方方式，那么可以使用tinySQL命令DUMPLO。pg_dumplo是tinySQL命令DUMPLO的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_load	pg_load是一个tinySQL数据库的命令。通过这个命令可以将文件加载到数据库。如果只想使用地方方式，那么可以使用tinySQL命令LOAD。pg_load是tinySQL命令LOAD的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_reindex	pg_reindex是一个tinySQL数据库的命令。通过这个命令可以重新索引数据库。如果只想使用地方方式，那么可以使用tinySQL命令REINDEX。pg_reindex是tinySQL命令REINDEX的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_rewind	pg_rewind是一个tinySQL数据库的命令。通过这个命令可以将数据库回滚到一个点。如果只想使用地方方式，那么可以使用tinySQL命令REWIND。pg_rewind是tinySQL命令REWIND的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_verifybackup	pg_verifybackup是一个tinySQL数据库的命令。通过这个命令可以验证备份。如果只想使用地方方式，那么可以使用tinySQL命令VERIFYBACKUP。pg_verifybackup是tinySQL命令VERIFYBACKUP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psql	psql是一个tinySQL数据库的命令。它将读取文件并运行命令。如果只想使用地方方式，那么可以使用tinySQL命令PSQL。psql是tinySQL命令PSQL的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psqlcheck	psqlcheck是一个tinySQL数据库的命令。通过这个命令可以检查tinySQL的审计日志。如果只想使用地方方式，那么可以使用tinySQL命令ARSCHECK。psqlcheck是tinySQL命令ARSCHECK的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psqlcontrol	psqlcontrol是一个tinySQL数据库的命令。通过这个命令可以启动或停止tinySQL。如果只想使用地方方式，那么可以使用tinySQL命令CONTROLCHECK。psqlcontrol是tinySQL命令CONTROLCHECK的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psqlstat	psqlstat是一个tinySQL数据库的命令。通过这个命令可以显示tinySQL的状态。如果只想使用地方方式，那么可以使用tinySQL命令STAT。psqlstat是tinySQL命令STAT的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psqlstat	psqlstat是一个tinySQL数据库的命令。通过这个命令可以显示tinySQL的状态。如果只想使用地方方式，那么可以使用tinySQL命令STAT。psqlstat是tinySQL命令STAT的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psqlupgrade	psqlupgrade是一个tinySQL数据库的命令。通过这个命令可以升级tinySQL。如果只想使用地方方式，那么可以使用tinySQL命令UPGRADE。psqlupgrade是tinySQL命令UPGRADE的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psql_wlmp	psql_wlmp是一个tinySQL数据库的命令。它主要处理连接数和线程数。它主要对数据库进行操作。如果只想使用地方方式，那么可以使用tinySQL命令WLMP。psql_wlmp是tinySQL命令WLMP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
服务器相关	initdb	initdb是一个tinySQL数据库的命令。通过这个命令可以初始化数据库。如果只想使用地方方式，那么可以使用tinySQL命令INITDB。initdb是tinySQL命令INITDB的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_archivecleanup	pg_archivecleanup是一个tinySQL数据库的命令。通过这个命令可以清理归档。如果只想使用地方方式，那么可以使用tinySQL命令ARCHCLEANUP。pg_archivecleanup是tinySQL命令ARCHCLEANUP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_cleancheck	pg_cleancheck是一个tinySQL数据库的命令。通过这个命令可以检查清理。如果只想使用地方方式，那么可以使用tinySQL命令CLEANCHECK。pg_cleancheck是tinySQL命令CLEANCHECK的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_ctl	pg_ctl是一个tinySQL数据库的命令。通过这个命令可以启动或停止数据库。如果只想使用地方方式，那么可以使用tinySQL命令CONTROLCHECK。pg_ctl是tinySQL命令CONTROLCHECK的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_ctlmon	pg_ctlmon是一个tinySQL数据库的命令。通过这个命令可以监控tinySQL。如果只想使用地方方式，那么可以使用tinySQL命令MONITOR。pg_ctlmon是tinySQL命令MONITOR的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_rewind	pg_rewind是一个tinySQL数据库的命令。通过这个命令可以将数据库回滚到一个点。如果只想使用地方方式，那么可以使用tinySQL命令REWIND。pg_rewind是tinySQL命令REWIND的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_tell	pg_tell是一个tinySQL数据库的命令。通过这个命令可以查询文件偏移量。如果只想使用地方方式，那么可以使用tinySQL命令TELL。pg_tell是tinySQL命令TELL的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_tell_sync	pg_tell_sync是一个tinySQL数据库的命令。通过这个命令可以将文件偏移量写入到文件。如果只想使用地方方式，那么可以使用tinySQL命令TELL_SYNC。pg_tell_sync是tinySQL命令TELL_SYNC的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_tell_trunc	pg_tell_trunc是一个tinySQL数据库的命令。通过这个命令可以将文件偏移量写入到文件。如果只想使用地方方式，那么可以使用tinySQL命令TELL_TRUNC。pg_tell_trunc是tinySQL命令TELL_TRUNC的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_upgrade	pg_upgrade是一个tinySQL数据库的命令。通过这个命令可以升级tinySQL。如果只想使用地方方式，那么可以使用tinySQL命令UPGRADE。pg_upgrade是tinySQL命令UPGRADE的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	pg_wlmp	pg_wlmp是一个tinySQL数据库的命令。它主要处理连接数和线程数。它主要对数据库进行操作。如果只想使用地方方式，那么可以使用tinySQL命令WLMP。pg_wlmp是tinySQL命令WLMP的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。
	psql	psql是一个tinySQL数据库的命令。通过这个命令可以读取文件并运行命令。如果只想使用地方方式，那么可以使用tinySQL命令PSQL。psql是tinySQL命令PSQL的一个子命令。在通过这个工具和地方方式对数据库进行操作时没有区别。

## 客户端工具

### clusterdb

#### 大纲

clusterdb[connection-option | --verbose | --table | -dbname]

clusterdb` [ *`connection-option` *...] [ `--verbose`   ` -v` ] `--all`   ` -a`
---

#### 选项

clusterdb接受下列命令行参数:

- - - - -

输出所有数据块。

- - d [dbname] - - owner [owner] - - db

当使用 - - all 时，如果要使用数据库名，那么必须使用 PGDATABASE 选项或连接名。如果还想使用更多的连接名，那么必须为连接名的名称设置。如果这样，连接的名称将被设置在中间的命令行。

- - - - -

输出 clusterdb 或其父级的连接参数的命令。

- - - - -

- - table - - tabletable

向集群 `table`，可以运行另一个 `-t` 并且会输出两个表。

`-v --version`

有类型限制的详细信息。

`-V --version`

从 `clusterdb` 命令行参数：

`-h host --host=host`

是关于 `clusterdb` 命令行参数的说明并指出，`clusterdb` 可接受下列命令行参数用于连接参数：

`-p port --port=port`

指定运行服务器的连接的端口号。如果未指明一个端口号，它使用与 `psql` 相兼容的端口号。

`-c connection-parameters`

指定从外部在连接时使用的 TCP 选项或使用 Unix 域套接字文件连接。

`-d database --user=username`

要为哪个用户连接。

`-w --no-password`

从不发出一个空口令。如果服务器要求从连接中省略其他方式的口令（例如“`\pssession`”），那么使用此选项无效。这个选项对于安全性来说非常有用，因为在其中没有一个用户来输入口令。

`-A --password`

强制 `clusterdb` 在连接到一个数据库之前要求输入一个口令。这个选项不是必不可少的，因为如果服务器要求口令以外，`clusterdb` 将自动地要求输入一个口令。但是，`clusterdb` 将会第一次连接尝试失败后另外再要求一个口令。在某些情况下，使用 `-A` 会失败的连接尝试。

`-r --no-connection-timeout`

当使用 `-s` 或 `-t` 时，指定要连接到的数据库和表的连接超时时间。如果未指定，将使用 `psql` 的设置，如果未设置，将使用 `temporal`。它只是 `connection_timeout`。如果是这样，连接的超时将根据设置的连接的超时。另外，连接到其他数据库时，除了数据库名字和其他连接时不同参数的设置。

## 环境

`-F PGDATABASE PGHOST PGPORT PGUSER`

### 默认连接参数

`-P 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 `psql` 工具相同。这个工具也使用 `PG_*` 变量的环境变量。

## 诊断

在处理命令时，可以在 `CLUSTERDB` 和 `test` 中指定问题或诊断消息的文本。如果未使用命令运行在脚本上。同样，使用 `psql` 的脚本使用以连接设置和诊断消息运行命令。

## 例子

要输出到 `psql` 中的一个表 `foo`：

`$ clusterdb test`

要输出在数据库 `xyzzy` 中的一个表 `foo`：

`$ clusterdb --table=foo xyzzy`

`createdb`

## 大纲

`createdb [connection-option] [option] [dbname description]`

## 选项

`createdb` 命令的命令行参数：

`-d dbname`

指定要创建的数据表名。该名称必须在这个集群中所有 `psql` 数据库中唯一。如果未设置一个名为 `createdb` 的用户名。

```

> description

指定当被创建的数据表被引用的一段注释。

> -d tablespaces -t tablespacespace

指定数据表空间让表空间（这个名称将当一个命令行的别名使用）。

> -e encoding

向createdb生成其父级的额外的命令。

> -E encoding -encodingencoding

指定要在这个数据库中使用的字符编码。

> -l locale -local=locale

指定要在这个数据库中使用的区域。这将对字符集指定“-lc-charset”和“-lc-ctype”设置。

> -lc-collate=collate

指定要在这个数据库中使用的LC_COLLATE 设置。

> -lc-ctype=collate

指定要在这个数据库中使用的LC_CTYPE 设置。

> -d owner -owner=owner

指定用于创建这个数据库的超级用户（这个名称将当一个命令行的别名使用）。

> -T template -tempalte=template

指定用于创建这个数据库的模板数据库（这个名称将当一个命令行的别名使用）。

> -V version

打印createdb版本信息。

> -h -help

显示关于createdb命令的帮助信息。参见 4.1.4. -h 和 -V 命令行参数 和 CREATE DATABASE 的信息。关于这些命令的更多信息参见 命令行参数。

createdb将接受一个命令行参数来连接到的数据库。
```

```
> -h host -host=host
```

指定服务器正在监听连接的TCP端口或使用 Unix 域套接字文件连接。

```
> -p port -port=port
```

指定服务器正在监听连接的TCP端口或使用 Unix 域套接字文件连接。

```
> -U username -username=username
```

要作为哪个用户连接。

```
> -w -no-password
```

如果用户需要登录认证但是没有其他认证方法（例如一个 [pgpass](#) 文件），那么将尝试密码。这个选择对数据库来说是不必要的，因为如果服务器要求密码，createdb将自动要求一个密码。如果选择一次连接请求失败服务器将要求一个密码。在某些情况下可能因为服务器的连接尝试。

```
> -c -connection=connection
```

指定要连接的其他数据库以使用其他的数据库。如果没指定，将使用 [postgres](#) 数据库。但如果它也不在（或者如果它是另一个连接数据库的名称），将使用 [template1](#)。它可以是 [connection string](#)。如果是这样，选择的字符串将被转换为与平常的命令行语法。

## 环境

```
> PGDATABASE
```

如果该设置，将使用该连接的数据名。除非在命令行中覆盖。

```
> PGHOST PGPORT PGUSER
```

默认连接设置。如果没在命令行用 [PGDATABASE](#) 指定要创建的数据名，[PGUSER](#) 也将使用该连接的数据名。

```
> PG_CLOB
```

规定在连接时不是必要的角色。可能的值为 [super](#)、[auto](#) 和 [nosuper](#)。

和大部分其他SQL工具相似，这个工具也使用SQL支持的环境变量。

## 诊断

在输出时，可以在 [CREATE DATABASE](#) 和 [CREATE](#) 中查看正在计划和请求命令的计划。数据命令将必须运行在脚本上。同时，任何SQL脚本将使用的命令在命令行和环境变量都将运行。

## 例子

要使用从数据文件创建的数据名 [demo](#)：

```
$ createdb demo
```

要在主机 [demo](#)，端口 5000 上使用 [tempfile](#) 增加新表的连接到 [demo](#)，这是命令分步命令和连接SQL命令：



```

+-->--help
显示有关 createuser 的命令行参数的简短说明。

createuser 也接受下列命令行参数作为连接参数：

+-->--host --host=host
指定使用哪个外部的主机连接到 PostgreSQL。如果未指定以一个斜杠分隔，它将使用 Unix 域套接字的连接。

+-->--port --port=port
指定服务器正在监听的 TCP 端口或使用 Unix 域套接字文件扩展。

+-->--username --username=username
要为哪个帐户选择（不是要创建的帐户）。

+-->--no-password
从不发出一个口令提示。如果服务器要从口令以某种方式识别用户（例如一个 pgpass 文件），那么将无法识别。这个选项对于长命性帐户根本有用，因为在其中设有一个帐户来输入口令。

+-->--password

```

如果 createuser 在连接到一个帐户时之连接不需要一个口令（例如连接到匿名帐户，而不是用户的口令），那么它不是必不可少的，因为如果服务器需要口令认证，createuser 将自己请求一个口令。但是，createuser 每次第一次连接尝试时仍然需要一个口令，在某些情况下它将使用 `--no-password` 选项的口令。

## 环境

```
+-->POSTGRES_PASSWORD
```

通过连接参数

```
+-->PG_CLOBBER
```

将它写入当前连接中最近连接的连接。可能的值为 `sleep` 或 `auto`。

## 诊断

在处理错误时，可以在 `CREATE ROLE` 和 `CREATE USER` 语句后附加 `NOVALIDATE` 以跳过该语句的检查。如果该语句通过所有检查，那么 `NOVALIDATE` 将被忽略。如果该语句通过所有检查，那么 `NOVALIDATE` 将被忽略。

## 例子

要在 PostgreSQL 服务器上创建一个用户 `joe`：

```
$ createuser joe
```

要从 PostgreSQL 服务器上创建一个用户 `joe` 并设置一些额外属性：

```
$ 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
```

要在端口 `eden`、端口 `5000` 上的服务器上创建一个用户 `joe`，并带有指定的属性，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` 为一个超级用户并为其分配一个口令：

```
$ 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，在输入口令时看不到并没有真正的回显，但是为了清晰，我们特别打印了出来。如此所示，接口将自动选择产生口令的散列值。

# dropdb

dropdb – 移除一个 PostgreSQL 数据库

## 大纲

dropdb [connection-option ...] option ... [dbname]

## 选项

dropdb接受下列命令行参数：

-h host

指定要移除的数据库的名字。

-i -echo

向终端输出dropdb在执行时读取参数的命令。

-f -force

在删除前向数据库写入。尝试从正在读取数据的所有的连接。有关此选项的详细信息，请参见DROP DATABASE。

-i --interactive

在操作时暂停操作工作之前输出一个命令提示符。

-v -version

打印dropdb版本号并退出。

-i --if-exists

如果数据库不存在也不报出一个错误。在远程情况下会发出一个警报。

-r -help

显示有关dropdb命令参数的帮助信息。

dropdb接受下列命令行参数作为连接参数：

-h host -host host

指定运行数据库的主机的主机名。如果省略以一个斜线开始，它将被当作连接参数的命令。

-p port -port port

指定服务器正在监听的TCP端口或主端口号或连接字符串的端口号。

-U username -username username

要作为哪个用户连接。

-w -no password

指定出口令牌。如果服务器要求出口令牌并且没有使用出口令牌（例如一个 `pgpass` 文件），那么连接尝试失败。这个选项对大多数特性都无效，因为在其中没有一个用户来输入口令。

-A -password

指定dropdb连接到一个数据库之前需要一个口令。这个选项是必不可少的，因为如果服务器要求口令认证，dropdb将自动要求一个口令。但是，dropdb将接受一次连接尝试之后自动地想要一个口令。在某些情况下，**–no password** 不是先要做的连接尝试。

-c -connection-dbname

指定一个数据库的名称。将连接到这个数据库以使用相同的数据库。如果省略指定，将使用 `postgres` 数据库。如果它也在（或者它就是正在被引用的连接参数），将使用 `tempdb`。这个值可以是一个连接字符串。如果没这样，连接字符串将被重新附加在连接的命令行之后。

## 环境

POSTGRES\_PASSWORD

默认连接参数

PG\_COLOR

指定连接颜色。可能的值为 `always`、`auto` 以及 `never`。

和大多数其他 PostgreSQL 工具相似，这个工具也使用与连接相关的环境变量。

## 诊断

在后台连接到数据库时，可以在 `psql` 中对连接进行故障排除的讨论。要连接到数据库必须运行在特权主机上。同样，也必须在连接中使用以连接设备和对连接参数进行适当的子句。

## 示例

要在本地数据库服务器上删除数据库 `demo`：

```
$ dropdb demo
```

要使用正在运行 `psql` 的本地会话连接到 `demo`，并带有连接子句，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pgSQL用户账户

### 大纲

`dropuser [connection-option ...] [option ...] [username]`

### 选项

dropuser接受下列的命令参数:

`-c [username]`

移除掉指定的pgSQL用户的账号, 如果没有在命令行指定会使用了`-i`或`-interactive`选项, 将会根据需要求一个用户名。

`-i [--interactive]`

向dropuser显示其帮助服务的命令。

`-h [--help]`

显示有关dropuser命令行参数的帮助信息。

`-k [host] [host] [host]`

指定运行服务器的机器的主机名, 如果没有指定一个的话将会, 它使用了Unix或套接字的连接。

`-p [port] [port] [port]`

指定服务器正在监听的TCP端口或是对Unix连接使用共享扩展。

`-r [username] [username] [username]`

要移除哪个用户的连接。

`-s [no] [password]`

不输出任何信息, 如果想要移除口令就必须使用可选的`-s` (例如一个`pgpass`文件), 那么将被尝试并失败, 这个失败时对数据库服务很危险, 因为在其中设有一个账户连接入口`$_`。

`-u [no] [password]`

强制dropuser在连接到一个数据库之前要求一个口令, 这个选项是必不可少的, 因为如果移除口令之后, dropuser将必须要求更多的口令, 但是, dropuser将花费一次连接尝试来移除服务的一个口令, 在某些情况下使用`-s`是更好的选择。

`-V [PGHOST PGPORT PGSOCK]`

默认连接参数

`-p [PGSQL]`

指定连接参数不是必需的, 可能的值为`always`, `auto`以及`never`。

和大部分其他pgSQL工具相似, 这个工具使用pgSQL支持的简单变量。

## 环境

`PGHOST PGPORT PGSOCK`

默认连接参数

`PGSQL`

指定连接参数不是必需的, 可能的值为`always`, `auto`以及`never`。

和大部分其他pgSQL工具相似, 这个工具使用pgSQL支持的简单变量。

## 诊断

在从新连接到数据库时, 可以在`psql`或`psql -c`中查看连接状态的输出, 数据库连接状态的行在输出之上, 同时, 在`psql`的输出中可以看到连接状态的输出。

## 示例

要移除数据库的用户名`jeff`:

```
$ dropuser joe
```

要删除正在运行的 joe，必须首先停止所有使用 joe 的连接。如果不确定是否，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 语句处理器

## 大纲

ecpg\_option :file

- -

处理 SQL 语句生成的 C 代码。当然，这从 **ECO SQL TYPE** 说起。

- -c mode

设置一个单独的变量。mode 可以是 **INFORMIX**, **INNODB**, **SE** 或 **ORACLE**。

- -d symbol

定义一个 C 项目符号。

- -i

处理头文件。输出文件名后，输出文件扩展名为 .apg 而不是 .app。默认输入文件扩展名为 .app 而不是 .apg。此外，将使用符号 -c 选项。

- -l

分析系统包头文件。

- -I directory

指定一个或多个包头目录。将本行添加到 **ECO SQL INCLUDE** 语句的文件，例如目录 -I /usr/local/include。在编译时定义的 PostgreSQL 语句 (假设 /usr/local/include 以及 /usr/include)。

- -o filename

指定 ecpg 语句将它的输出写入到文件 filename。写 -o 使得有输出类型和连接信息。

- -o option

选择 ecpg 的行为。option 可以是下列之一：no\_indicator 不使用游标指示符来表示游标状态。历史上曾有数据使用这种方法。prepare 在使用游标语句之前准备它们。游标语句一个接着一个的顺序并当它们被执行时重新准备它们。如果被关闭了，游标将被最少使用的话。quiescance 为兼容性原因才使用游标作为占位符。在插入语句时使用游标。

- -q

打开事务的自动提交。在这种模式下，每一个 SQL 命令会自动提交。除非它是一个显式事务块，在那时它将自动关闭。命令只有当 **ECO SQL COMMIT** 语句时才被提交。

- -r

打印额外信息，包括版本和“帮助”命令。

- -version

打印 ecpg 版本并退出。

- -h --help

显示关于 ecpg 令人惊奇的函数的信息。

在编译带有嵌入式 SQL 的 C 代码时，编译器需要将编译的 PostgreSQL 语句嵌入到 C/C++ 代码中。因此，在调用编译器时，你可能必须使用 -I 选项 (例如，-I /usr/local/include)。

使用带有嵌入式 SQL 的 C 代码的语句必须被链接到 libecpg。例如使用连接语句 -L /usr/local/include -l ecpg。

如果你安装了这个目录的库，你可以使用 -L 选项。

## 注解

```
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MySQL数据库中检查表

### 大纲

`pg_amcheck option ... [dbname]`

### 选项

以下命令行选项和参数是必需的：

`-i -a -o`

检查所有数据表，但通过 `--exclude-database` 排除的数据除外。

`-d patterns--databasepatterns`

检查与给定的 `pattern` 匹配的数据表，但 `--exclude-database` 排除的数据除外。可以多次指定此选项。

`-i patterns--exclude-databasepattern`

排除与给定的 `pattern` 匹配的数据表。可以多次指定此选项。

`-i patterns--indexpattern`

排除与给定的 `pattern` 匹配的索引。除非它已被排除在外，可以多次指定此选项。如果位于 `--relation` 之后，只要它只适用于索引，两个选项才有效。

`-i patterns--exclude-indexpattern`

排除与给定的 `pattern` 匹配的索引。如果多次指定此选项，它将多次指定此选项。如果位于 `--relation` 之后，只要它只适用于索引，两个选项才有效。

`-i patterns--relationpattern`

检查与给定的 `pattern` 匹配的表。除非它已被排除在外，可以多次指定此选项。要使用它是单模式的，例如 `myrel1`，或者它是模式模式的，如 `myschem1%myrel1%` 或者 `myrel1%myschem1%`。模式模式是通过将模式与表名或模式与模式组合的表名。

`-i patterns--exclude-relationpattern`

排除与给定的 `pattern` 匹配的表。可以多次指定此选项。与 `--relation` 一起，`patterns` 不以单模式的，模式模式的表名或模式与模式的。

`-i patterns--schemapatterns`

检查与给定的 `pattern` 匹配的表。除非它已被排除在外，可以多次指定此选项。如果多次指定与模式模式的模式中的表，该表将使用 `--table-SCHEMNAME.%--no-dependent-indexes`。要使用模式，必须使用 `--table-SCHEMNAME.%--no-dependent-indexes`。要使用模式，必须使用 `--table-SCHEMNAME.%--no-dependent-indexes`。例如，你可以编写 `--schema=pub myschem1` 在数据库中选择匹配 `myrel1` 的模式。

`-i patterns--schema`

排除与给定的 `pattern` 匹配的表。除非它已被排除在外，可以多次指定此选项。与 `--schema` 一起，模式可以是单模式的。

`-i patterns--tablepatterns`

检查与给定的 `pattern` 匹配的表。除非它已被排除在外，可以多次指定此选项。如果多次指定 `--relation` 之后，只要它只适用于表，两个选项才有效。

`-i patterns--exclude-tablepattern`

排除与给定的 `pattern` 匹配的表。可以多次指定此选项。如果多次指定 `--relation` 之后，只要它只适用于表，两个选项才有效。

`--no-dependent-indexes`

默认情况下，如果选出了一张表，它也将包含它的 `views` (如果有的话)。如果它没有通过 `-i` 和 `--table` 或 `--relation` 之后的选项来选择，此选项将自动执行。

`--no-dependent-index`

默认情况下，如果选出了一张表，它也将包含它的 `views` (如果有的话)。如果它没有通过 `-i` 和 `--table` 或 `--relation` 之后的选项来选择，此选项将自动执行。

`--no-strict-names`

默认情况下，如果在表中遇到 `view` 时，将自动将其转换为 `table`。如果它没有通过 `-i` 和 `--table` 或 `--relation` 之后的选项来选择，此选项将自动执行。

`--no-error-stop`

在发现错误的表的第一行后将停止运行。停止时将显示文本，只列出一个错误条目。请注意，部分行将只在第一个行之后停止，此选项将抑制此行为。

`--skipoption`

如果指定 `all-views`，将自动在视图的所有表中标记为全部通过的页面。如果指定了 `all-visible`，则视图将标记为所有表中标记为所有可能的页面。默认情况下，不能通过此方法。因此，必须使用 `--no-error-stop`，因为此选项将自动执行。

`--startBlock=block`

从指定的块号开始检查。如果正在检查的表不满足该少于 `start` 的行数，则将跳过该表。此选项不适用于 `views`。可能仅在检查单个表时有效。请参阅 `--endBlock` 了解更多信息。

`--endBlock=block`

在指定的块号结束检查。如果正在检查的表不满足该少于 `end` 的行数，则将跳过该表。此选项不适用于 `views`。可能仅在检查单个表时有效。如果通过 `start` 指定表，则此选项将自动执行。如果通过 `end` 指定表，则此选项将自动执行。如果通过 `start` 和 `end` 指定表，则此选项将自动执行。如果通过 `start` 和 `end` 指定表，则此选项将自动执行。

`--beginList=begin`

对于每个选中的表，使用 `begin` 和 `beginList` 选项将开始一个新行作为开始行的前导行。





```
... -2 --version
```

```
pg_basebackup版本号和帮助。
```

```
... -h --help
```

```
显示有关pg_basebackup命令行参数的详细信息。
```

## 环境

和大型的其他 PostgreSQL 工具相似，这个工具会使用与之完全相同的环境变量。

环境变量 **PG\_CLOB** 将会在诊断消息中显示颜色。可能的值为 **always**、**auto**、**never**。

## 注解

在备份的开始，将要在备份目录上的文件查看。这可能需要一些时间（尤其是在使用选项 **--checkpoints-first** 时），在此期间pg\_basebackup会关注于设置状态。

备份将根据目录和表空间中的所有文件。如果配置文件以兼容二进制方式在目录中包含多个文件，不过 PostgreSQL 是根据文件的文件名来识别文件的。也可以使用文件名来指定文件。也可以使用文件名来指定文件。如果 PostgreSQL 已经识别到目录的文件将被忽略。其他任何文件和文件设备文件将被忽略。

在普通文件中，将从目录和表空间上的所有文件，除非使用了 **--tablespace-mapping** 选项。如果有多个目录或表空间正在使用，有另一个表空间上进行普通的文件和表空间的文件，因为备份将写入新的表空间的目录。

在使用 tar 格式时，用户将负责在归档文件中的所有表。如果有额外的表空间，对于它们的 tar 文件需要被单独处理的分区，在这个例子中，服务器将根据包含在 **base.tar** 文件中的 **tablespace\_map** 文件的值为新的表空间创建适当的目录。

**pg\_basebackup** 可以具有附加的包含基本的映射“-F”工具。

如果在连接上启用了映射，**pg\_basebackup** 将忽略表空间的映射。

## 例子

要连接到名为 **mydbserver** 的一个普通名并将其存储在本地目录 **/usr/local/pgsql/data** 中：

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

要创建本地归档的一个备份，为其中每一个表空间产生一个归档的 tar 文件，并将它们储在目录 **backup** 中，在这个例子是另一个连接名：

```
$ pg_basebackup -D backup -Ft -z -P
```

要创建一个兼容本地数据的备份并将其存入目录 **ts**：

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

（如果在数据目录中有多个表空间，这个命令将失败）。

要创建一个支持映射的备份，其为 **/opt/ts** 中的表空间设置它到 **/backup/ts**：

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

## pgbench

pgbench - 在 PostgreSQL 上运行一个基准测试

## 大纲

```
pgbench -[option | -dname]
```

```
pgbench option | -dname
```

## 小结

**pgbench** - 一个连接到 **pgbench\_accounts**、**pgbench\_branches**、**pgbench\_history** 或 **pgbench\_tellers**，生成随机操作并从连接池读取，如果使用了 **-t** 选项，生成的随机操作将被忽略，如果使用了 **-c** 选项，一定主要使用一个连接。

在测试的连接中，“t”的值设为 1，这些操作的速率行为：

table	# of rows
<hr/>	
pgbench_branches	1
pgbench_tellers	10





打印pgbench的基本信息。

`... -h --help`

显示有关pgbench的运行参数的信息，并且帮助。

## 退出状态

打印开始和结束的信息。退出状态与执行状态有关。如失败的命令失败，运行命令的命令，失败数据读取或读取命令的命令将返回状态为1。在另一种情况下，pgbench将打印成功结果。

## 环境

`... PGDATA PGPORT PGPORT PGPORT`

输出参数。

此参数类似于大多数类型的MySQL实用程序一样，使用lsof获得的环境变量。

环境变量 `PG_CLOB` 指定是否在日志消息中使用转义。可能的是是 `always`、`auto` 或 `never`。

## pg\_config

`pg_config --option...`

## 大纲

`pg_config option...`

## 选项

要使用pg\_config，提供一个或多个以下选项：

`... --bindir`

打印可执行文件的目录。你需要将这个选项添加到pg\_config命令。这将使它将pg\_config命令指向你的目录。

`... --datadir`

打印文件目录的目录。

`... --includedir`

打印客户端编译的C头文件的位置。

`... --libdir`

打印客户端库文件的位置。

`... --includedir-server`

打印用于服务器编译的C头文件的位置。

`... --libdir`

打印对客户端库文件的位置。

`... --pgincludedir`

打印包含可插入模块的位置。或者要外部可能使用它们的位置（如果它们指向数据文件可能也需要放在这个目录）。

`... --includedir`

打印包含支持文件的位置（如果在MySQL数据库时没有配置文件支持，这将是一个空字符串）。

`... --readdir`

打印子目录的位置。

`... --sharedir`

打印包含支持文件的位置。

`... --spandir`

打印系统全局配置文件的位置。

`... --sys`

打印子目录module的位置。

`... --configure`

打印MySQL数据库的命令 `configure` 基本的选项。还可以使用 `--with-option` 和 `--without-option` 来指定选项，或者只指定一个选项（如果只指定了一个二进制，则（不过主要二进制通常包含所有相关的选项））。参见下面的例子。

`... --cc`

打印用来编译MySQL的CC变量名。这是你想要的环境变量。

`... --cflags`

打印用来编译MySQL的CFLAGS变量名。这是在另存时需要的C编译开关（典型的`-I`开关）。

`... --ctags`







## 诊断

pg\_dump 会执行 **SELECT** 语句。如果使用 pg\_dump 时出现问题，通常可能是从正在使用的数据库中选择错误。查看 `psql`。此外，`psql` 提供了帮助信息，可以帮助识别错误。要通过 `psql` 执行 `SELECT` 语句，可以使用 `\d` 或 `\d+` 命令。

pg\_dump 的帮助信息会根据以下参数进行调整。如果不想这样，可以通过 `\d` 或 `\d+` 命令来覆盖。要通过 `psql` 执行 `SELECT` 语句，可以使用 `\d` 或 `\d+` 命令。

## 注解

如果只想将表集插入 `template0`，则必须使用 `WITH TEMPLATE template0`。要通过 pg\_dump 执行一个真正的空数据表，必须使用 `CREATE TABLE` 语句和一个空的表结构。否则，将向表中插入的数据添加到表的现有数据中。

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

当一个只包含表的脚本被导入时，会使用 `DISABLE TRIGGERS`。pg\_dump 会禁用插入数据之后会自动执行的触发器，并且禁用在表中插入之后会自动执行的约束。如果想要在将表插入之后禁用触发器，系统将从表中禁用所有自动触发器。

pg\_dump 会将脚本文件包含在数据库目录中。因此，从一个脚本文件向表执行 `ALTER` 语句时，表的权限将不会被修改。

在禁用所有自动触发器时，pg\_dump 会使用 `CONNECT + FALSE` 向 `CREATE SUBSCRIPTION` 语句。这样便可以在不通过连接池的情况下禁用自动触发器。然后将使用 `ALTER` 语句禁用自动触发器。如果在执行 `ALTER` 语句之后，自动触发器仍会启用，那么将禁用自动触发器。

## 实例

要从一个数据库 `mydb` 导出到一个 SQL 脚本文件：

```
$ pg_dump mydb > db.sql
```

要将这样一个脚本导入到一个（新创建的）名为 `newdb` 的数据库中：

```
$ psql -d newdb -f db.sql
```

要创建一个数据文件并将其导出到一个 SQL 脚本文件：

```
$ pg_dump -Fc mydb > db.dump
```

要从这样一个数据文件重新导入到一个新创建的目录：

```
$ pg_dump -Fd mydb -f dumpdir
```

要从一个目录文件重新导入到一个（新创建的）名为 `newdb` 的数据库：

```
$ pg_restore -d newdb db.dump
```

要从一个名为 `mytab` 的表：

```
$ pg_restore -d postgres --clean --create db.dump
```

要从表 `detroit.emp` 导出所有表，排除名为 `employee_log` 的表：

```
$ pg_dump -t mytab mydb > db.sql
```

要从表 `detroit.emp` 导出所有表，排除名为 `employee_log` 的表：

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

要转换成以 `east` 或者 `west` 开始并以 `psql` 结尾的转存模式。同样地你也可以 `test` 的转存模式:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

同样地, 你也可以这样写:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

要转换成了名称以 `ts_` 开头的表之外的所有表的转存:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

要转换成一个大写形式或小写形式的转存, 你需要使用转存名称, 直到它会输出大写或小写形式。但是转存名称 `tsql` 是特别的, 所以反过来它们会输出小写。因此, 要转换一个转存为大小写形式的表, 你需要以这样的方式:

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

## pg\_dumpall

`pg_dumpall` – 将一个 PostgreSQL 数据库转存到一个文本文件中

## 大纲

`pg_dumpall [connection-option | option ...]`

## 描述

`pg_dumpall`工具可以将一个数据库中所有的表以文本文件的形式输出 (“转存”)。这个文本文件, 读取文本文件并将其插入SQL命令来恢复数据库。它从数据库中的每个数据表 (或 `pg_class` 表) 来完成此工作。`pg_dumpall`还转存所有数据库公用的全局对象 (`pg_global` 不转存这些对象), 也就是说数据库角色和授权会自动被转存。并且它会转存所有数据库的视图和存储函数。它也可以转存所有数据库的触发器等等。

因为`pg_dumpall`从外部数据源中读取数据, 所以你可能需要以一个数据库的用户名 (或无权限的连接) 登录。

SQL 基本语句写入转存的脚本。使用 `-F` 或 `-Fts` 选项或者 `file` 附件可以指定要输出的一个文件。

`pg_dumpall`需要多次连接到 PostgreSQL (每个数据源一次)。如果使用多个连接, 可能每次都需要登录。这种情况下使用一个 `-U` `pgpass` 会比较方便。

## 选项

下列命令行选项用于控制输出的命令和格式。

`-c --data-only`

只转储数据, 不转储模式 (或表定义)。

`-c --clean`

把转储在重映射到新之数据库 (转存)。它转存所有角色和表空间的 `OID`。角色和表空间的 `OID` 会自动插入进来。

`-c --encoding --encoding=encoding`

转存对字符集编码的转换规则。转存命令, 转储使用数据库的编码 (另一种将转储对字符的处理方式是设置 `PGCLIENTENCODING` 环境变量为想要的字符编码)。

`-c --filename=filename`

将输出文件写到指定的文件中。如果省略, 将使用转存输出。

`-c --global-only`

只转储全局 (角色和表空间)。两个转储的限制。

`-c --no-owner`

不能使用设置对所有对象的权限的命令。转存命令, `pg_dumpall`使用 `ALTER OWNER` 或 `SET SESSION AUTHORIZATION` 语句来设置被转存的模式之所有权限。转存命令是一个超级用户 (或者是转存库中所有对象的一个拥有者) 所执行。这意味着它只能设置所有对象的权限, 但不能设置所有对象的权限。可以通过 `-O`。

`-c --recheck`

只转储角色, 不转储数据表和表空间。

`-c --schema-only`

只转储对表定义 (模式)。不转储数据。

`-c --supersoname --superuser=superuser`

指定要在转储时使用的超级用户的用户名。只有使用 `-c` 选项才有效 (通常, 最好省略这个选项, 因为作为转储用户来说转储主会自动代替)。

`-c --tablespace-only`

只转储表空间, 不转储数据表和角色。

`-c --vacuum`

指定操作模式。选择将 `pg_dumpall` 的恢复操作限制为单线程以及恢复文件的开始/停止时间。还有些命令，要求你选择命令将在恢复操作上应用的限制或参数。这个选项只在操作模式的 `dump`。

`--version`

打印 `pg_dumpall` 版本并退出。

`--no-privilегes --no-aci`

禁止对命令行（授予/恢复命令）。

`--binary-upgrade`

这个选项用于对文件进行操作。我们不建议也不支持把它用于其他的。这个选项在未来的发行中可能被改变或不再通知。

`--column-inserts --attribute-inserts`

对数据插入为带有已识别的 `INSERT` 行（`INSERT INTO table(column,...) VALUES ...`）。这将使得恢复过程更简单。这主要对希望将数据插入到 MySQL 数据库。

`--disable-dollar-matching`

这个选项禁止在函数中使用美元符号。并在限制它们使用 SQL 和某些字符串函数时。

`--disable-triggers`

只在处理一个特别的数据时，我们才这样做。它将 `pg_dumpall` 限制在数据的恢复者插入的数据时禁用触发器。这将使在表上执行完整性检查或其他触发器。并会在数据被插入时禁用这些触发器。当我们在数据被插入时禁用它们，通常会是一个恢复点。因此，你应当使用 `--file` 为一个起始用户名，或者你可以为一个起始用户名的目录。

`--exclude-database* pattern`

不要将带有 `pattern` 且不为 `pattern`。可以通过编写多个 `--exclude-database` 为文件添加多个过滤。根据 `pgrep` 的命令使用相同的规则。因此，通过模式中使用通配符可以排除多个数据库。使用通配符时，请谨慎的使用它们。你需要的 `lstat` 来进行扩展。

`--extra-float-digits=digits`

对浮点数进行使用 `float`、`float8` 或 `float4` 可以。而不是最大的精度。备份时将进行的采样将这个值乘以 100。

`--if-exists`

对文件和命令（`CREATE` 或 `IF EXISTS`）来说将自动忽略。如果它失败了 ... `--if-exists`，这个选项会忽略。

`--inserts`

对数据插入 `INSERT` 命令（而不是 `COPY`）。这将使得恢复更简单。这主要对希望将数据插入到 MySQL 数据库。要注意命令行参数被限制了。当恢复时将一起执行。`--column-inserts` 选项对于命令行恢复是完全的。也是必须的。

`--load-via-partition=not`

在物理的恢复中不需要恢复的表。并是在指定的 `timeout` 内不需要做一个新的头的。那时你可以在 `SET statement_timeout` 选项的行。

`--lock-unit-timeout=timeout`

在物理的恢复中不需要恢复的表。并是在指定的 `timeout` 内不需要做一个新的头的。那时你可以在 `SET statement_timeout` 选项的行。

`--no-comments`

不输出注释。

`--no-publications`

不输出 publication。

`--no-role-passwords`

不输出角色的密码。在恢复后，角色的密码是空的。并会设置空的密码以防止密码不成功。除非你对这个选项并不需要以空值。角色的密码从恢复的 `pg_roles` 表中 `pg_authid` 中读出。因此，如果对 `pg_roles` 表没有写权限，那么这个选项会自动失败。

`--no-security-labels`

不输出安全标签。

`--no-subscriptions`

不输出 subscriptions。

`--no-type`

假设情况下，`pg_dumpall` 将输出所有不需要插入到数据库。这个选项会将 `pg_dumpall` 限制为将输出命令。这样会更快。但是要输出所有的命令的输出需要设置 `statement_timeout`。通常来说，这个选项是正确的。但你必须在主上安装存储数据的限制。

`--no-tablespaces`

不能输出其他空间的命令。通过这个选项，在恢复期间所有的数据必须在在文件中以文件的形式。

`--no-text-compression`

不要输出命令以设置 `TOAST` 为本地方法。采用这个选项，所有的命令以恢复时将被设置为本地。

`--no-unlogged-table-data`

不输出自己记录的内部。这个选项对于恢复来说是必要的。它只输出恢复的数据。

`--no-conflict-do-nothing`

在 `ON CONFLICT DO NOTHING` 的 `INSERT` 命令。除非 `--inserts` 或 `--column-inserts` 选项被设置，否则此选项才生效。

`--quote-all-identifiers`

强制所有字符的引号。在从一个叫 `pg_dumpall` 的不同的 MySQL 数据库的数据恢复时，你必须输出一个不同的基本的限制。除非使用这个选项。默认情况下，`pg_dumpall` 只关心为其他基本的字符设置的限制。在为其他版本的 MySQL 或者不同的数据库的限制时，这可能会导致兼容性问题。使用 `--quote-all-identifiers` 可以通过兼容性问题，但是代价是存储空间会更加昂贵。

`--reread-per-inserts`

从 MySQL 数据库的 `SET SESSION AUTORIZATION` 命令或 `ALTER USER` 命令恢复时的权限。这是对恢复时的权限的限制。这将使恢复无法正常完成。

`--use-set-session-authorization`

是所有 `pg_dumpall` 的命令参数的限制并退出。

下列表示各可用数据库连接参数。

-d *connname* --database=*connname*  
指定子连接的数据库。这个选项类似于 *dbname*，但是为了和其他参数统一，所以因为 *pg\_dump* 需要两个不同的参数，所以子连接中的数据库名将被忽略。使用 *-d* 选项指定一个数据库，该数据库将用于子连接，还将被忽略子连接自身的数据库名。

-h *host* --host=*host*

指定服务器正在运行的数据库的主机名。它将操作一个 Unix 域套接字的连接，所以是通过 **PGPORT** 端口来连接（如果被设置）。否则将尝试一次 Unix 域套接字连接。

-i *dbname* --database=*dbname*

指定要连接的数据库连接池以及是否要将连接池其他数据。如果设为指定，将使用 *postgres* 数据库。如果 *postgres* 不存在，将使用 *template1*。

-p *port* --port=*port*

指定服务器正在运行的连接的 TCP 端口号或 Unix 域套接字文件扩展名。默认是通过 **PGPORT** 端口连接中（如果被设置），否则使用连接字符串中的端口号。

-s *username* --username=*username*

用户名为哪个用户连接。

-w --no-password

从不提示输入密码。如果向数据库要连接口令且没有其他方式提供口令（例如一个 *.pgpass* 文件），那么将尝试从文件读取。这个选项对于处理文本和脚本有用，因为在其中设有一个用户名输入口令。

-A *password*

强制 *pg\_dump* 连接的一个数据库连接提供一个口令。这个选项从不关心，因为如果服务器要口令认证，*pg\_dump* 将会提示你输入一个口令。但是，*pg\_dump* 将会再一次连接过来为你提供一个口令，在其他情况下，你得键入 *A* 并按先输入的密码。注意对每个要连接的数据库，口令是相同的。通常，最好设置一个 *.pgpass* 文件来减少手工输入。

-c *role* --release=*role*

指定一个用来连接数据库的角色。这个选项将使 *pg\_dump* 在连接数据库后发出一个 **SET ROLE** *role* 命令。当只使用 *-c* 选项（或 *-A* 选项）时 *pg\_dump* 所做的操作就是将你指定的一个具有所有权限的角色，这个选项没有用。一些安装针对直接为超级用户的登录的限制，使用这个选项可以让连接在连接到集群时不受限制。

## 环境

-F PGPORT PGPORT PGPORT

默认连接参数

-P *color*

指定是否在消息中显示使用颜色。可能的值为 *always*, *auto*, *never*。

和大多数其他 PostgreSQL 工具相似，这个工具使用相同的环境变量。

## 注解

因为 *pg\_dump* 完全调用 *pg\_dump*，所以，一些注解也可以参考 *pg\_dump*。

假设用户的连接比本地连接还要慢，这时可以使用 *-c clean* 选项也有用。如果 *pg\_dump* 的连接比本地连接还要慢，那么对两个数据库文件采集集中相同的属性（例如 *sequence* 和 *rule*），如果不使用这个选项，这两个数据库将采集它们各自的数据库属性以区分各自的内部。

一旦停顿，建议在每个数据库上运行 **ANALYZE**，这样之后数据可以再用同样的语句。

不应该同时使用本地连接和远程连接。特别要注意当远程连接的每一个角色为同一个角色时（**CREATE ROLE** 是也），对于 *vacuum* 的结果会产生一个“role already exists”错误。除非你将连接作为一个不同的 **vacuum** 使用，这时错误是正常的并且没有影响。*-c clean* 选项将很可能要生成主要的有关于在对对象的完整性检查。不过可以通过 *-i* 或 *-F* 选项减少完整性检查。

*pg\_dump* 要求所有重要的类型必须在运行恢复之前被成功创建。否则，数据文件将永远在单独位置被忽略。

## 例子

要转换为数据库：

```
$ pg_dumpall > db.out
```

要从这个文件恢复数据库。你可以使用：

```
$ psql -f db.out postgres
```

注意你连接每一个数据库都需要一个连接字符串。因为 *pg\_dump* 只处理的连接的数据库将被忽略的数据库。一个作品，如果指定了 *-c clean*，所有的连接必须使用 *postgres* 数据库，否则将生成一个连接失败的错误。并且对连接字符串上的这个数据库将被忽略。

**pg\_isready**

*pg\_isready* – 修复一个 PostgreSQL 服务器的连接状态

## 大纲

**pg\_isready** connection-option | option

## 选项

-i *dbname* --database=*dbname*

指定要连接的数据库。*dbname* 可以是连接字符串。如果未设置，将使用所有连接字符串中的第一个。

-h *host* --host=*host*

指定运行服务器的连接参数。如果指定了一个连接参数，它将被用作连接数据库的URL。

`-c port -port port`

指定服务器连接时使用的TCP端口或本地Linux连接字符串。默认值为5432。如果连接字符串包含端号，那么以连接字符串中的端口（通常是5432）。

`-t -t quiet`

不显示连接消息。当连接本地时特别有用。

`-t -t seconds -timeout seconds`

尝试连接时，在返回服务器不同步之前等待的最大秒数。设置为0时无限。默认值是30s。

`-d database -user user -user`

作为用户名 `username` 连接数据库，而不是使用默认的。

`-v -version`

从 `pg_isready` 命名并退出。

`-h -help`

显示有关 `pg_isready` 命令的帮助信息并退出。

## 退出状态

如果服务器已连接成功，`pg_isready` 将返回 0；如果失败（连接失败或连接超时），将返回 1；如果连接尝试失败（连接超时或连接失败），将返回 2；如果连接尝试失败且连接失败，将返回 3；如果连接尝试失败且连接超时，将返回 4。

## 环境

和大多数其他的 PostgreSQL 工具一样，`pg_isready` 会使用环境变量的连接配置。

环境变量 `PG_CATALOG` 只能在分析消息中显示使用颜色。可能的值为 `always`、`auto`、`never`。

## 注解

参见附录 A，有关连接参数的详细信息。

## 例子

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

使用连接参数连接到本地 PostgreSQL 服务器：

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

使用连接参数连接到远程 PostgreSQL 服务器：

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

## pg\_recvreplay

`pg_recvreplay` 以流的方式从 PostgreSQL 服务器接收日志文件。

# 大纲

`pg_recvreplica option ...`

## 选项

`-d directory --directory=directory`

要输出收到哪个目录。这个参数是必需的。

`-i int --index=index`

当接收到正确的日志时，自动停止复制并且不再发送消息。如果有一个参数的话还要带 `int`，则该参数将被忽略。

`-r if-not-exists`

当指定 `--create-salt` 并且没有指定新的槽已经存在时不要输出。

`-s --no-loop`

不要在连接队列上循环。相反，直到一个错误时立刻退出。

`-v --verbose`

这个选项将 `pg_recvreplica` 不通知从数据源的回插。这样会更快，也是它要快于下面的命令的两倍会比从日志慢。通常，这个选项才特别有用，但不容易在对生产部署进行日志的使用。这个选项与 `--synchronous` 不兼容。

`-c interval --status-interval=interval`

指定发送回连接状态之间的间隔。这允许我们更容易地监控服务器的状态。一个带单位的值将自动转换为毫秒。这样我们可以在毫秒级别精确地设置。这比使用一个更长的字符串来表示秒和毫秒的间隔要好。默认值是 1 秒。

`-s saltname --salt=saltname`

要使用 `pg_recvreplica` 使用一个已有的盐值。在使用这个选项时，`pg_recvreplica` 将会自动地将新接收到的盐值替换。指定每一个盐值时必须使用一个等号。这样我们可以在毫秒级别精确地设置。这比使用一个更长的字符串来表示秒和毫秒的间隔要好。默认值是 `pg_recvreplica` 在接收到第一个连接时自动地生成的盐值。因此，该配置将正常地处理上的事务即使不同的连接无法从一个人那里工作。要让连接正常工作，必须同时选择 `--synchronous` (见下文)。

`--synchronous`

在从连接到从连接的来回跳转。只要在两个连接之间接收到一个盐值 (参见 `--status-interval`)，所有的文件名后缀将被忽略 (见 `-s`)。

`-c --check`

启动完整性检查。

`-l level --compression-level`

此选项写入到从连接的 pg\_log 回写，并且必须直接到 (DML, 从一个连接的最大日志)。所有的文件名后缀将被忽略 (见 `-s`)。

下列命令行选项和配置参数是兼容的。

`-d constr --database=constr`

指定用于连接服务器的参数。作为 `pg_hba.conf`，它将覆盖所有其他的命令行选项。为了和其他参数应用一起，该选项称为 `--db`。也是因为 `pg_recvreplica` 不连接到集群中的任何其他数据库。选择字符串的数据库名并使用它。

`-h host --host=host`

指定运行服务器机器的主机名。如果想要以一个特殊开头，它将使用 Unix 域套接字的参数。默认使用 `POSTGRES_HOST` 环境变量 (如果设置)。或者一个值是在命令中的任意值。

`-p port --port=port`

指定从连接到从连接的 TCP 端口或连接到 Unix 域套接字文件的端口。默认使用 `POSTGRES_PORT` 环境变量 (如果设置)。或者一个值是在命令中的任意值。

`-c connection --connection=connection`

要作为哪个用户的连接。

`-u --no-password`

从输出一个空密码。如果服务器想要从连接到从连接的连接有其他形式的密码 (例如一个 `pgpass` 文件)，那么连接尝试失败。这个选项不被必须使用。但是，`pg_recvreplica` 将会要求一个空口令。但是，`pg_recvreplica` 将会要求一次连接尝试失败后才能想要一个空口令。在某些情况下使用该 `--password` 来避免失败的连接尝试。

`-s --password`

强制 `pg_recvreplica` 在连接到一个数据库之前必须要求一个空口令。这个选项不被必须使用。因为如果服务器想要从连接到从连接的连接有其他形式的密码 (例如一个 `pgpass` 文件)，那么连接尝试失败。这个选项不被必须使用。但是，`pg_recvreplica` 将会要求一次连接尝试失败后才能想要一个空口令。在某些情况下使用该 `--password` 来避免失败的连接尝试。

为了帮助理解参数，`pg_recvreplica` 可以执行下列两种操作之一：

`-c create-salt`

所 `--salt` 指定的名称创建一个新的物理复制槽，然后退出。

`-c drop-salt`

删除 `--salt` 中指定的复制槽，然后退出。

其他选项也可用：

`-v --version`

显示 `pg_recvreplica` 版本并退出。

`-h --help`

显示有关 `pg_recvreplica` 的行参数的说明并退出。

## 退出状态

在接收到信号终止 (没有正确的文件句柄，因此不是一种错误) 时，`pg_recvreplica` 将会退出。对于命令错误或者其他信号，它将忽略并退出。

# 环境

和大多数其他命令行工具相似，这个工具也使用PG支持的环境变量。

环境变量 **PG\_CLOB** 指定在分析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 `trigger`。否则，服务器可能会在写入沃尔文件时将其重写，或者损坏它。因为没有文件锁（尽管来自 `archive_command` 或 `trigger` 的写入 WAL 会已标记为坏文件），不过要注意，如果读者没有检测到坏的 WAL 数据，一个要处理坏消息的服务器可能会。

如果在消息集上应用了快照点，`pg_receivewal` 将保留读取的 WAL 文件上的快照。

## 例子

要从位于 `mydbserver` 的服务器读取的数据写入并将其存储在本地目录 `/usr/local/pgsql/archive`：

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

## pg\_recvlogical

`pg_recvlogical` — 从逻辑复制接收消息

## 大纲

`pg_recvlogical` 选项:

## 选项

必须至少指定一个列选项之一来选择一个动作:

· --create-slots

为 `--dbname` 指定的数据表使用 `--slot` 指定的名称创建一个新的逻辑复制槽。使用 `--plug` 指定的插件插件。

· --drop-slots

删除所有使用 `--slot` 指定的逻辑槽。然后启动。

· --start

从 `--slot` 指定的数据表开始从 `--start` 指定的点开始更改。如果需要使用只读或者新连接可读更改的参数，必须在一个单独的命令行参数中指定。通过指定 `--no-loop` 可以防止这种情况进入循环模式。在从只读模式切换到可读的输出时将发生此操作。就像必须是连接的两个逻辑槽的同一个数据集上。

`--create-slot` 和 `--start` 可以一起使用。`--drop-slot` 不能和任何一个选项组合在一起。

下面的命令将逻辑复制的配置和参数以及数据集名为:

· -l 'list' --endpos 'list'

在 `--start` 模式下，当接收线程从指定的 LOG 表的逻辑槽上读取到以之为起始点的记录之后，如果开启了 `--start` 模式的话将这个点设为从头开始。如果开启了 `--start` 模式的话将这个点设为从头开始。如果有一个记录的 LOG 过早于 `list`，则该记录将被忽略。`--endpos` 不会将记录从世界范围可能会在一个事务中遗漏。对只读分段的参数将不会被忽略。并且在一次只读集中读取的将会是多条事务。单个的记录不会被忽略。

· -f filename --filenumber

将接收到的自启动时的数据写入到一个文件。参数 `-` 可以写到 `stdout`。

· -i interval\_seconds --sync-interval=interval\_seconds

通过 `pg_recvlogical` 使用的脚本文件不能完全地实现端到端的恢复。如果需要在接收到的数据写入到文件之前进行重写和重排，这个设置可以在之外的脚本中执行。这可能需要 `ls` 在完全停止之后 `sync` 脚本。但是由于接收到的数据是乱序的，在这种情况下，只能使用命令行参数。

· -l 'list' --startpos 'list'

在 `--start` 模式下，从指定的 LOG 开始复制。

· --if-not-exists

当使用 `--create-slot` 并且具有指定的槽已经存在时将不会报出错误。

· --no-loop

当服务器连接失败时，不要在循环中重试。直接退出。

· -o name [value] --optname=value

如果指定了输出选项，只使用值 `value` 为输出选项。`name`，并在输出选项以及它们的值之间以逗号分隔的输出选项。

· --plugin=plugin

在创建一个槽时使用指定的逻辑插件的插件。如果该槽已经存在，这个选项将被忽略。

· -a interval\_seconds --status-interval=interval\_seconds

这个选项和 `pg_receivewal` 中的 `status` 选项具有相同的效果。请参考 `status` 的描述。

· -S slot\_name --slot=slot\_name

在 `--start` 模式下，使用名为 `slot_name` 的已连接复制槽。在 `--create-slot` 模式下，使用这个名称创建槽。在 `--drop-slot` 模式下，删除这个名称指定的槽。

并行读取插件的使用。

下列命令行选项和数据源连接参数。

```
-d database --database=database

指定的数据源。这个选项的详细含义请见前面的描述。database 可以是 连接字符串、连接配置、连接字符串连接池 或 连接池连接。默认认为 连接字符串。
```

```
-h hostaddr-or-ip --host=hostaddr-or-ip

指定服务器正在运行的机器的主机名。如果未指定则使用一个空的。它使用一个 Unix 域套接字的连接。默认是以 PGHOST 环境变量中指定的连接。若以连接字符串连接，则认为 连接字符串。
```

```
-p port --port=port

指定服务器正在运行的端口。如果未指定则使用一个空的。默认是以 PGPORT 环境变量中（如果有设置），否则使用端口 5432。若以连接字符串连接，则认为 连接字符串。
```

```
-U user --username=user

指定为哪个用户连接。默认使用连接字符串中指定的用户名。
```

```
-w --no-password

从不提示输入密码。如果服务器要连接时没有其他方式识别口令（例如一个 pgpass 文件），那么将尝试读取口令。这个选项对于连接池和连接字符串。因为在其中没有一个用户来插入口令。
```

```
-A --password

从 pg_dump 读取的一个数据源之后提示输入一个口令。这个选项是必须的。因为如果服务器要求口令认证，pg_dump 时必须提供一个口令。但是，pg_dump 并未要求一次连接尝试失败后服务器想要一个口令。在某些情况下，你得键入 A 来提示输入的连接尝试。
```

```
-V --version

显示 pg_restore 的版本信息。
```

```
-h --help

显示关于 pg_restore 举行的参数的说明，并且退出。
```

## 环境

和大部分其他 `pg` 工具相比，这个工具也使用 `pg` 支持的环境变量。

环境变量 `PG_CATALOG` 可以在连接字符串中使用明码。可能的值为 `always`、`auto`、`never`。

## 注解

### pg\_restore

`pg_restore` – `h` – `一个` `log_start` 识别的文件名或一个 `pg` 数据库。

## 大纲

### 选项

`pg_restore` 支持下列命令行参数。

`-f` `filename`

指定要恢复到的文件（对于一个自包含的物理备份）的位置。如果没有指定，则使用标准输入。

`-d` `data-only`

只恢复数据，不恢复模式（恢复文件）。如果在文件中存在，表数据、大对象和序列将被忽略。这个选项必须与指定 `--sectiondata`，也是为了历史原因而不支持。

`-c` `--clean`

在重新加载数据源时之前清理（丢弃）它。除非使用了 `--if-exists`，否则使用了 `--if-exists`，否则将对表和序列数据源中不存在，这可能会生成一些不必要的错误消息。

`-C` `--create`

在恢复一个数据库之前创建它。如果未指定 `--create`，在选择的数据库之前丢失并自动重建它。如果使用 `--create`，`pg_restore` 将会恢复数据源的连接（如果有的话）以及与之相关的恢复参数，也就是说对所有表都执行 `ALTER DATABASE ... SET ...` 或 `ALTER ROLE ... IN DATABASE ... SET ...` 命令。不管是 `create` 或 `--no-ail`，数据文件的连接将被忽略。在使用这个选项，`--create` 的数据源只用于之后的 `DROP DATABASE` 或 `CREATE DATABASE` 命令。所有要恢复的数据源将不被忽略或应用到它们。

`-d` `database` `--database=database`

选择数据库。`database` 从连接字符串中选择数据库。`database` 可以是 `连接字符串`、`连接配置`、`连接字符串连接池` 或 `连接池连接`。默认认为 `连接字符串`。

`--exit-on-error`

在发生任何命令错误时退出。如果行为是中断并自动尝试恢复，则这是一个错误计划。

`-f` `filename` `--file=filename`

生成的脚本将被输出到文件，或是在 `-4` 选项一起使用的文件中。如果使用 `--format=tar`，为 `filename`。

`-4` `format` `--format=format`

指定 `pg_restore` 时的输出格式。其不一定指定为 `tar`，因为 `pg_restore` 将会根据连接类型自动选择。如果指定，可以是下列之一：`custom` 为 `pg_dump` 的输出文件。`directory` 为一个目录。`tar` 为一个 `tar` 文件。

`-1` `index` `--index=index`

只恢复指定的索引。可以通过更多一个 `-1` 为又指定更多索引。

`-j` `number-of-jobs` `--jobs=number-of-jobs`



```
* --user-set-session-authorization
```

```
禁止SQL语句的 SET SESSION AUTHORIZATION 语句对 ALTER OWNER 命令来说是无效的。这会限制对某些对象的权限，但是不能限制命令自身的权限，可能无法达到效果。
```

```
* -i --help
```

```
显示有关 pg_restore 的命令行参数的说明，并且退出。
```

```
pg_restore 也是显示下列有关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。如果连接失败并运行一段时间，它将使用一个从 pg_hba.conf 文件读取的连接，前提是使用 POSTGRES 身份验证（如果被设置）。否则将尝试一次 Unix 域套接字连接。
```

```
* -p port --port=port
```

```
指定服务器正在运行的机器的 TCP 端口号或 Unix 域套接字文件扩展名。默认是使用 POSTGRES 身份验证中（如果被设置），否则使用连接字符串中的端口号。
```

```
* -d database --database=database
```

```
要使用哪个数据库。
```

```
* -u --no-password
```

```
不发出一个口令提示。如果服务器要求口令并且没有其他方式提供口令（例如一个 pgpass），那么将尝试读取文件。这个选项对于本地连接来说是无效的，因为在此没有一个用户名插入口令。
```

```
* -w --password
```

```
假定 pg_restore 需要一个数据库之连接时将要求一个口令。这个选项不是必须的，因为如果服务器要求口令之后，pg_restore 会自动要求一个口令。但是，pg_restore 将会第一次连接时从连接参数中要求一个口令。在某些情况下，使用插入 -w 来避免密码的连接尝试。
```

```
* -r --role=role
```

```
指定一个用来连接数据库的角色。这个选项将对 pg_restore 的连接和数据库连接发出一个 SET ROLE 语句。当已连接时（即 -d 选项）禁止 pg_restore 读取的特权是能够识别的一个具有所需求的角色时。这个选项很有用，一些安装有针对直连作为连接用户登录的限制。使用这个选项可以连接在不能连接的数据库。
```

## 环境

```
* PGHOST PGPORT PGPORT PGUSER
```

```
从环境变量
```

```
* PG_CLOB
```

```
假定在连接时使用什么角色。可能的值为 always、auto、never。
```

```
和大多数其他 PostgreSQL 工具相似，这个工具也使用从环境变量中读取的连接参数。
```

## 诊断

```
当使用 -d 选项指定一个直接连接时，pg_restore 会尝试对 SELECT 命令执行。如果使用 pg_restore 的连接参数，要注意 pg_restore 的连接参数中的一个真正的空字符串。否则它很可能向不正确的数据库中选择连接，例如 foo。此外，pg_restore 在连接时尝试从连接参数中读取设置和环境变量的值。
```

## 注解

```
如果将连接参数指派给 temporal 的连接参数，要注意 pg_restore 的连接参数中的一个真正的空字符串。否则它很可能向不正确的数据库中选择连接，例如 foo。此外，pg_restore 在连接时尝试从连接参数中读取设置和环境变量的值。
```

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

```
下述将详细说明 pg_restore 的用法。
```

```
* 在恢复数据之前，如果在连接中启用了选项 -d disable_triggers，pg_restore 会从插入数据之前设置的命令中使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。如果恢复在中止停止，可能会丢失未完成于恢复的插入。
```

```
* pg_restore 不是处理表级的完整性约束。例如对恢复来说更大的约束，影响所有的大对象都会被忽略。如果通过 -A -I 或者对表级的行级锁，它们一个也不会被忽略。
```

```
一些限制需要，必须在每一个被恢复的表上运行 ANALYZE。这样它才能得到有用的统计信息。
```

## 示例

```
假设我们已经从另一个地方恢复了一个名为 mydb 的数据库：
```

```
$ pg_dump -Fc mydb > db.dump
```

```
要恢复此数据库并将其恢复到新的连接：
```

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

```
* 对于存储的数据可以使用任何对存在数据库中的数据表。pg_restore 会使用 mydb 为表 CREATE DATABASE 命令。通过 -C，将要连接到恢复到此表在文件中的数据库名称。
```

```
要运行的连接参数：
```

```
$ createdb -T template0 newdb
```

```
$ pg_restore -d newdb db.dump
```

正在执行一个命令，但是尚未执行脚本或读取任何参数。还要注意我们从 `tempfile` 的一个 `tempfile` 创建了临时脚本，以便让它能被运行。

要从数据库恢复所有，或在恢复时使用不同的参数。

```
$ pg_restore -l db.dump > db.list
```

列表文件将是一个文本和二进制格式。这将对每一个逻辑单元一视同仁。例如：

```
;;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMONS
;     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
```

正在执行一个逻辑单元，但是尚未执行了它要从每个逻辑单元中读取。

文件中的行可以被注释掉，删除以恢复执行。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
```

就这样一个文件作为pg\_restore的输入将会恢复到新数据库，并且恢复以恢复所有。

```
$ pg_restore -L db.list db.dump
```

# pg\_verifybackup

pg\_verifybackup – 检查PG兼容的备份文件的完整性

## 大纲

pg\_verifybackup [option ...]

## 选项

pg\_verifybackup 检查以下命令行参数:

- -exit-on-error

检测到备份后立即退出。如果没有指定这个选项, pg\_verifybackup 将在检测到问题后继续备份, 并将检测到的所有问题报告为错误。

- -i path --ignore-path

在检测中忽略与路径文件相对的 backup\_manifest 文件中的所有数据文件或目录。该文件将被忽略为新的路径。如果指定了目录, 将忽略该目录以及该目录下的所有子目录。如果指定了文件, 该文件及其目录将被忽略。如果文件不存在, 文件名将被忽略但该文件的目录将被忽略。可以多次指定选项。

- -o path --manifest-path

使用指定路径的兼容文件, 而不是从备份中读取的兼容文件。

- -o parsetest

不要从日志中从各处恢复外联的写入式只读数据。

- -q --quiet

成功验证后将不显示任何检测结果。

- -skip-checksums

不要验证数据文件的校验和。它们将被假设为文件以及这些文件的大小。这样将节省带宽, 因为文件本身不需要读取。

- -o path --out-directory

将检测结果写入指定目录的 SQL 文件, 而不是 pg\_out。如果备份中包含多个不同的目录, 则它们将被忽略。

其他选项(可选):

- -version

显示 pg\_verifybackup 版本信息。

- -help

显示 pg\_verifybackup 令行参数的帮助。

## 示例

要在 mydbserver 上检测兼容的基本备份并报告兼容的完整性:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ pg_verifybackup /usr/local/pgsql/data
```

要在 mydbserver 上检测兼容的基本备份, 请将兼容性检查目录之外的某个位置, 并且退出:

```
$ 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
```

要在忽略不兼容的数据文件的完整性检查, 并且忽略校验和:

```
$ 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
```



对文件进行操作时的表头参数。该命令 `-t` 表示 `[get_tablet, only]`。

指定要操作的 `table` 对象的选项。该头 `[put_tablet]`。

指定要操作的 `user` 对象的选项。该头 `[put_user]`。

作为用户 `username` 的用户名从本地读取密钥对。当然, 你必须是该用户的管理员。

该命令 `-u` 表示 `[get_user, password]`。

执行一个文字符命令。该命令 `-x` 也必须使用。通常你应该在命令行字符串中使用引号。

该命令 `-x --version`。

打印帮助菜单并退出。

该命令 `-x --no-passwd`。

从不发出一个登录提示。如果必须使用密码以认证自己从其他来源 (例如, 从一个脚本) 登录, 请使用该命令。

该命令 `-x --password`。

通过 `psql` 读取一个数据库以帮助你要求一个口号。如果口号会变得更长, 它将被截断。

该命令 `-x --repeated`。

打印扩展器模式的提示。该命令 `-x` 或者 `[put_expanded]`。

该命令 `-x --no-psqlrc`。

不读取配置文件 (要是从本地启动 `psql` 文件, 要从本地启动的 `\$psqlrc` 文件)。

该命令 `-x --Field-separator-zero`。

设置半角输出的记录分隔符为零字符。该命令 `-x` 或者 `[put_field_sep]`。

该命令 `-x --record-separator-zero`。

设置半角输出的记录分隔符为零字符。该命令 `-x` 或者 `[put_sep]`。

该命令 `-x --single-transaction`。

这个选项只能被用于一个或者多个 `-c` 选项 (或者 `-t` 选项)。它必须以单引号包围。

该命令 `-x --help["topic"]`。

## 退出状态

如果 `psql` 是要执行的，它会先 `shell` 但它将是一个命令级（限制文件操作，而不是文件）。它会运行。如果数据库的连接上没有设置为是生成式，它会运行2。如果在根本不会有连接，它会运行3 并且会是 `ON_ERROR_STOP` 的设置。

- `COLUMNS`

如果 `psql -c columns` 为零，这个时候变量会利用到了 `wrapped` 修饰符度以从头到尾是正确的。如果使用行或者块或者列的模式，它会使用行的模式。

- `PGDATABASE PGHOST PGPORT PGUSER`

假设在连接参数中没有设置的值，可能的值是 `always`, `auto`, `never`。

- `PG_CLOB`

假设在连接参数中没有设置的值，可能的值是 `always`, `auto`, `never`。

- `PGD_EDITION EDITOR VISUAL`

`\e`, `\p` 以及 `\ps` 会使用所有的编辑器。会根据列出的顺序首选这些变量。第一个设置的编辑器使用。如果都没有设置，那么是使用Linux系统上的 `vi` 或者Windows系统上的 `notepad.exe`。

- `PGD_EDITION_LINEMODE_ANS`

当 `\e`, `\ef` 或者 `\es` 有一个行参数时，这个变量会告诉 PostgreSQL 使用行模式编辑器的命令行参数。对于 `vi` 或者 `vim` 不同的编辑器，这个变量是一个加号。如果要在命令行和行之间有空格，那么在变量的值中添加一个适当的空格。例如：

- `PGD_HISTORY`

命令行历史会不同的管理位置。如果值是 `(-)` 会将命令执行。

- `PGD_PAGESIZE`

如果一个查询的结果是在屏幕上显示不了，它们会通过这个命令分页显示。典型的值是 `more` 或 `less`。通过将 `PGD_PAGESIZE` 设置为无穷大可以禁用分页器的使用。但是 `\pset` 会部分覆盖其他的设置以达到同样的效果。当使用行的连接参数时这个设置是禁用的。

- `PSQL`

用户的 `psql` 文本编辑器位置。如果值是 `(-)` 会禁用编辑。

- `SELL`

假设是命令行的命令。

- `TOPDIR`

## 文件

- `psqlrc` and `~/.psqlrc`

#### • `.pgsql_history`

命令行历史被存储在文件 `~/.pgsql_history` 中，或者是 Windows 的文件 `SUPPORTAN\postgresql\pgsql_history` 中。历史文件的位置可以通过 `HISTFILE` 环境变量或者 `PSQL_HISTORY` 环境变量所指的设置。

## 注解

- MySQL具有如同三版本或者更高的主版本最为匹配。如果服务器的版本比MySQL本体要高，则反向转换会令其容易失败。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三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。

如果你想将pgsql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pgsql。或者，你可以为每一个主版本保留一份pgsql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在包路径中插入一个命令。注意操作符的改变：

```
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    |           |           |
```

现在我们把提示修改一下：

```
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 `\set` 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 " \t ".

peter@localhost testdb=> SELECT second, first FROM my\_table;

one 1

two 2

three 3

```
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，将 `\df` 命令以及它的输出 `\t\g\pl\bigint` 分开第二部分用 `\bigint` 表示的函数：

```
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，可以使用\crosstabview命令以文本的形式查看结果：

```
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)
```

这两个例子展示了我的“私活”（功能）。行级别的可操作性对于理解复杂的分析方式非常有帮助。

```
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 - 是使用一个语句对所有表的索引

# 大纲

```
reindexdb connection-option [option] [-S] --schema schema [-A] --table table [-I] --index index [-D] --dbname
```

```
reindexdb` [*`connection-option`*...] [*`option`*...] ` -a` | `--all
```

```
reindexdb connection-option [option] [-S] --schema schema [-A] --table table [-I] --index index [-D] --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

当 -A / --all 不使用时，如果要重索引的数据表的名称，如果未指定，则必须包含 **PGDATABASE** 该环境变量的名称，如果未设置，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，连接字符串将被重置为连接的执行语句。

- -i --index

向reindexdb提供要重索引的数据库的名称。

- -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 --jobs --jobs jobs

通过同时运行 **jobs** 任务并行执行 **reindex** 任务，此选项可带来永远不会耗尽的并行性。它还会将连接池限制为 **jobs** 任务，因此连接数 **max\_connections** 会受到限制，可以同时执行连接。请注意，此选项与 **-i index** 和 **--system** 选项不兼容。

- -q --quiet

不显示进度消息。

- -s --system

向reindexdb提供要重索引的系统数据库。

- -S --schema --schemas schema

只对 **schema** 重索引。通过写多个`-S`开关可以指定多个要重索引的模式。

- -A --table --table table

只对 **table** 重索引。通过写多个`-A`开关可以指定多个表。

- --tablespace

指定要重索引的表空间。(这个命令必须与 **table** 一起使用。)

- -v --verbose

向reindexdb提供要重索引的表。

- -h --help

显示有关 **reindexdb** 的所有可用的选项和参数。

reindexdb已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。如果只指一个IP地址，它将使用 Unix 套接字字符串。

- -p port --port port

指定服务器在监听的 TCP 端口或使用 Unix 套接字文件名。

- -D database --dbname database

要使用哪个数据库。



将执行的“计划”为。

... force\_index\_cleanup

这是移除旧的冗余的索引条目。

... -j jobs --jobs=jobs

通过同时运行 `analyze` 命令来并行地处理多个表。这个选项可能永远不会使用，但是它会增加数据库对分析的处理。vacuum 使用 `analyze` 对数据库的连接，因此连接的 `max_connections` 设置将影响它的性能的连接。注意如果某些系统设置被打开，使用这种模式上 -F (FIFO) 选项可能会导致失败。

... -m mult\_idx mult\_idx

只在 mult\_idx 少于 `mult_idx` 的表上执行空闲或命令。这设置对于需要处理的表的连接数过大时，以防连接数溢出。对于此选项的用途，正常的 mult\_idx 值是正常的 TOAST 表的连接数最大的，如果你认为，由于 vacuum 执行的命令需要对表处理 TOAST 表的连接，它无须单独考虑。

... -m old\_idx old\_idx

仅处理多于 `old_idx` 的表。对于此选项的用途，正常的 old\_idx 值是正常的 TOAST 表的连接数最大的，如果你认为，由于 vacuum 执行的命令需要对表处理 TOAST 表的连接，它无须单独考虑。

... -m index\_cleanup

不扫描旧的冗余的索引条目。

... -n no\_process\_truncate

通过与要清理的表有相同的任何 TOAST 表。

... -n no\_truncate

不要在表的尾部插入空值。

... -N parallel\_workers --parallel=parallel\_workers

指定 parallel vacuum 的并发数量。这允许你同时使用多个 CPU 来处理表。请参见 [vacuum](#)。

... -q --quiet

不显示进度消息。

... -s skip\_locked

通过过滤掉所有锁定的表进行处理的表。

... -t table [column [..]] --table=table [column [..]]

只处理或分析 `table`，列或可能的 `-analyze` 或 `-analyze-only` 选项的表。通过写多个 `-t` 你可以处理多个表。使用此选项的唯一限制，你必须使用又来指明 `analyze` 的符号（以下面的例子）。

... -v --verbose

在处理期间打印详细的进度。

... -V --version

显示 vacuumdb 的版本信息。

... -x --analyze

只计算分区表的统计信息。

... --analyze-in-parallel

只对 `analyze-only` 表，只计算分区表的统计信息（不更新）。使用 `analyze` 选项设置设置执行平行的（`-V` 或 `-V`）以及更新分区表的统计信息。这个选项将并行一个副本时如果要使用 `analyze` 选项的数据库名称。这个选项将并行一个副本时如果要使用 `analyze` 选项的数据库名称，然后在后续的副本中“共享”的统计信息。

... -h --help

显示有关 vacuumdb 分行表的说明并退出。

vacuumdb 也接受下列分行参数用于连接参数：

... -A host --host=host

指定运行服务器的机器的主机名。如果省略以一个问号开头。它使用 Unix 可用连接字符串。

... -p port --port=port

指定要为端口在运行连接的 TCP 端口或使用 Unix 域套接字文件连接。

... -U username --username=username

提供为哪个用户连接。

... -W --no-password

从不提示为一个口令输入。如果首先你要口令但是没有其他口令（例如一个 `psql` 文件），那么连接尝试失败。这个选项对于处理挂起的连接非常有用，因为在其中没有一个用户连接入口。

... -w --password

强制 vacuumdb 在连接到一个数据库之后提示要输入一个口令。但是，vacuumdb 将首先一次连接到数据库并提示要输入一个口令。在某些情况下，`W` 会覆盖部分的连接尝试。

... -n database --dbname

当使用 `-A` 或 `-a` 时，必须提供连接的数据库名以及后台连接的数据库。如果未指定，将使用 `postgres` 的连接。如果不存在，将使用 `template`。这可以是 `connection_string`。如果是这样，连接字符串将被传递给连接的命令行选项。此外，在连接到其他数据库时，将使用带有数据库名和连接字符串的连接尝试。

## 环境

... PGDATABASE PGHOST PGPORT PGUSER

从环境变量读取。

... PG\_COLOR

指定在诊断命令是否使用颜色。可能的值为 `always`、`auto` 或 `never`。

和大多数其他 MySQL 工具相似，这个工具也使用 MySQL 支持的环境变量。

## 诊断

在命令行时，可以在 `vacuumdb` 和 `psql` 中设置问题和错误消息的显示。数据服务器必须运行在相同主机上。同样，使用 `psql` 的脚本使用以连接设置和环境变量相同的方式。

## 注解

`vacuumdb` 可能需要多次连接到 MySQL 服务器。每次都需要一个口号。在这种情况下，一个 `\c` 命令又会很方便。

## 例子

要连接到 `test`：

```
$ vacuumdb test
```

要连接到名为 `bigdb` 的数据库中的一个表 `foo`，并且为它分析表结构的 `bar` 列：

```
$ vacuumdb --analyze bigdb
```

要连接到名为 `xyzzy` 的数据库中的一个表 `foo(bar)`，并且为它分析表结构的 `xyzzy` 列：

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

## 服务器应用

### initdb

`initdb` — 创建一个新的 MySQL 数据库

## 大纲

`initdb` option [[`--options` | `-D` | `directory`]

## 选项

`--auth-method` `--auth=authmethod`

这个选项为本地用户指定在 `pg_hba.conf` 中使用的认证方法（`host` 和 `local`）。`initdb` 将使用指定的认证方法为半制连接以及复制连接读取 `pg_hba.conf`。除非你已经在本地账户，不要使用 `trust`。为了安全起见，`trust` 会默认设置。

`--auth-host=authmethod`

这个选项通过 TCP/IP 连接的本地用户指定在 `pg_hba.conf` 中使用的认证方法（`host`）。

`--auth-local=authmethod`

这个选项通过本地连接的本地用户指定在 `pg_hba.conf` 中使用的认证方法（`local`）。

`--data-directory` `--pgdata=directory`

这个选项指定的数据目录将被忽略的目录。这是 `initdb` 要求的唯一参数。但是你必须设置 `PGDATA` 环境变量来避免它。这可能是因为之后你将使用 `postgres` 可以使用的一个全局唯一的数据目录。

`--encoding` `--encoding=encoding`

选择数据的数据编码。这将被后来连接的任何数据库的默认编码。除非你设置它，将使用来自系统。或者如果你设置子集并使用为 `SQL_ASCII`。

`--file=alias-group-access`

大多数操作系统中的用户读取 `initdb` 生成的所有配置文件。Windows 会忽略此选项，因为它不支持 COMSPEC 的功能。

`--data-checksums`

在数据文件上使用校验码来帮助识别（C）系统连接的损坏。此功能将不会引起任何性能影响。如果设置，所有外存的数据将被校验，在整个数据集中。外存数据块将被写入到 `pg_stat_database`。

`--local-locale`

为数据集指定本地化区域。如果这个选项没有被设置，系统将从 `initdb` 所运行的计算机中读取。

`--lc-collate=locale` `--lc-type=locale` `--lc-messages=locale` `--lc-monetary=locale` `--lc-name=locale` `--lc-time=locale`

在 `initdb` 时，它将从本地运行的计算机设置区域。

`--no-locale`

将从 `--local-locale`。

## 注解

initdb 可以通过 pg\_ctl initdb 被调用。

## pg\_archivecleanup

pg\_archivecleanup – 清理MySQL WAL 归档文件

## 大纲

```
pg_archivecleanup [option ...] archive_location oldest
```

## 选项

`pg_archivecleanup`接受下列命令行参数:

`-h`

在 `stderr` 上打印很多或自定义输出。

`-v`

在 `stderr` 上打印要移除的文件的名称 (执行一次遍历)。

`-V --version`

`pg_archivecleanup`版本号。

`-x --extension`

提供一个扩展名，在指定的文件名前加上这个扩展名。将从文件名中移除这个扩展名。通常你倾向于希望对存储桶归档对象使用一个扩展名的文件。例如: `-x .gz`

`-h --help`

显示 `pg_archivecleanup` 所有参数的详细信息。

## 环境

环境变量 `PG_CLOB` 指定是否对输出消息中使用颜色。可能的值是 `always`, `auto` 或 `never`。

## 示例

在 Linux 或者 Unix 系统上, 可以使用:

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r
2>>cleanup.log'
```

其中的脚本将运行在备机脚本上, 这样 `archive_cleanup` 将直接与之通信。但是文件对于后备脚本来说是不可见的, 因此将

`-c cleanup.log` 作为参数提供

以从日志中移除所有需要的文件

## pg\_checksums

`pg_checksums` (–) verySQL 数据库脚本中包含。禁用或启用的脚本 (只读)

## 大纲

`pg_checksums` (option) (–) (–) `pgdata` (detail)

## 选项

下列表示所有可用的:

`-d directory` `--pgdata=directory`

指定存储数据的目录的目录。

`-c --check`

检查校验和。如果木块其它任何内容, 它将被忽略。

`-d --disable`

禁用校验和。

`-e --enable`

启用校验和。

`-f file` `--file=file`

仅读进文件或 `file` (文件中的所有文件)。

`-s --sync`

禁用 `pg_checksums` 会自动将文件完全移到磁盘上。你必须使用 `pg_checksums` “命令行”运行, 这样更安全。也就是说, 你必须在本地或网络存储系统上运行 `pg_checksums`。一般地, 你必须对文件使用, 但不能在生产机上。当使用 `--check` 时, 你必须关闭。

`-p --progress`

输出进度报告。在处理或读取文件时, 打开消息, 会提供进度报告。

`-v --verbose`

输出详细输出。列出所有检查的文件。

`-V --version`

`pg_checksums` 版本号。

`-h --help`

是关于 `pg_controldata` 行参数的常见问题。

## 环境

• PGDATA

指定要使用哪个目录。可以使用 `-d` 选项覆盖。

• PG\_COLOR

指定是否在控制台输出使用颜色。可能的值为 `always`, `auto`, `never`。

## 注意

在大型集群中使用较大的时间间隔会导致在连接中断时，写入数据的集合或其它核心资源未被写入，这将可能导致数据丢失。

当要设置与文件系统相关的连接参数时，可以使用 `pg_controldata` 一起使用。此操作将导致所有以 `pg_controldata` 为前缀的参数被忽略，如果未在参数上执行操作的话。要在集群中启用或禁用后台操作，使用 `pg_ctl` 时请使用 `background` 此参数带有各种数据集，这主要取决于操作，最后从头部到尾部逐行读取，也是安全的。

如果在启动或停止后台操作时使用 `pg_ctl`，那么 `pg_controldata`，将从集群的配置文件配置读取并忽略。

## pg\_controldata

`pg_controldata` 是一个 MySQL 数据库集群的控制程序。

## 大纲

• PGDATA

## 环境

• PGDATA

通过环境变量设置。

• PG\_COLOR

指定是否在输出中使用颜色。可能的值为 `always`, `auto`, `never`。

## pg\_ctl

`pg_ctl` 是一个 MySQL 数据库集群的控制程序。

## 大纲

• PGDATA

`pg_ctl init` (0 data(r) | -s) = init db-options;

`pg_ctl start` (0 data(r) | -s) (filename | -c) (-t seconds | -c) (-o options | -p path) | -c

`pg_ctl stop` (0 data(r) | -s) (start | -c) (mediated | -c) (-t seconds | -c)

`pg_ctl restart` (0 data(r) | -s) (start | -c) (mediated | -c) (-t seconds | -c) (-o options | -c)

`pg_ctl reload` (0 data(r) | -s)

`pg_ctl status` (0 data(r) | -s)

`pg_ctl promote` (0 data(r) | -c) (-t seconds | -c)

`pg_ctl logrotate` (0 data(r) | -s)

`pg_ctlkill` (signal\_name) (0 data(r) | -s)

在 Microsoft Windows 上，还有：

`pg_ctl register` (0 data(r) | -s) (servicename | -c) (username | -c) (password | -c) (sid | -c) (command | -c) (-o source | -c) (-t seconds | -c) (-o options | -c)

`pg_ctl unregister` (0 data(r) | -s)

## 选项

• -c --config-file

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 'data(r) | -s) --pgdata= data(r)

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`。

• -c '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。

```
• -o mode --mode=mode
指定文件模式。mode可以是 start, fast 或 immediate，或者这三个之一的第一个字符。如果这个选项被忽略，则 fast 是默认值。
```

```
• -o options --options=options
指定要设置在命令 postgres 中的选项。-o 可以被指定多次。所有被指定的选项都会被结合起来。任何选项的值都必须以等号或分号包围并用单引号括起来，它们作为一组参数。
```

```
• -o initdb-options --options-initdb-options
指定要设置在命令 initdb 中的选项。-o 可以被指定多次。所有被指定的选项都会被结合起来。任何选项的值都必须以等号或分号包围并用单引号括起来，它们作为一组参数。
```

```
• -o path
指定 postgres 可以访问的目录，类似于 PGDATA。postgres 可以从该目录以 pg_ctl 的同样方式获得。或者如果没有任何参数，则从该目录的 PGDATA 中读取。除非该目录中有一些不同的命令或脚本，这个选项不是必需的。在 init 模式下，这个选项被自动地从 initdb 为该目录指定的值。
```

```
• -o -s -client
只打印错误，并打印致命性的消息。
```

```
• -o seconds --timed=seconds
指定事件一个操作完成时要等待的最大时间（见选项 -o）。默认为 POSTGRES 并发主要的。如果该事件没有设置，则默认认为 0 秒。
```

```
• -o -version
打印 pg_ctl 版本并退出。
```

```
• -o -quit
等待操作完成。这是 start, stop, restart, promote 以及 register 之外的这个选项。并且对所有其他模式也是有效的。在等待时，pg_ctl 会一直检测各服务器的状态。在两次检查之间会等待一段时间。当从文件中读取各服务器已经完成准备就绪时，启动操作被认为完成。当服务器被标记为故障时，关闭操作被认为完成。当服务器在超过时间（见选项 -t）内未完成，或 pg_ctl 在以一个非零退出状态退出。但是要注意操作可能在后台操作时才被标记为完成。
```

```
• -o -no-salt
不带操作的。
```

```
• -o -help
显示有关 pg_ctl 命令参数的帮助并退出。
```

```
作为唯一的 Windows 服务启动。pg_ctl 在事件中记录它的事件级别的消息。默认是 INFO。注意这可能影响 pg_ctl 事件级别的消息。一旦启动，服务将使用 event_source 参数中指定的事件名。如果服务是在启动时（见参数 autostart）启动的，它可能也会使用以事件名名的 INFO 来记录。
```

```
• -o services
指定要启动的服务的名称。这个名称将用于服务名和显示名。默认是 INFO。
```

```
• -o -password
用于运行杀毒软件的用户名。
```

```
• -o start-type
指定服务启动的启动类型。启动类型可以是 auto, demand 或者两者之一的第一个字符。如果这个选项被忽略，则 auto 是默认值。
```

```
• -o username
用于运行杀毒软件的用户名。对于管理员，使用用户名 SA 或 sa。
```

## 环境

```
• PGDATABASE
指定数据库的连接参数。
```

```
大部分的 pg_ctl 命令都需要知道数据库的连接。因此 -d 选项是必需的。除非 PGDATA 被设置。
```

```
和大部分其他 pg 工具一样，pg_ctl 也需要 pg 支持的环境变量。
```

```
更多有关服务的文档请见 pg_ctl。
```

## 文件

```
• postmaster.pid
pg_ctl 在需要时会将这个文件写到服务器运行的目录。如果该文件不存在，则从 PGDATA 中读取。
```

```
• postmaster.opts
如果这个文件不存在于数据目录中，pg_ctl (除了 restart 之外) 将读取文件的路径为连接参数 options。除非通过 -o 选项进行了覆盖，这个文件的内容将被写入到 status 模式中。
```

## 例子

### 启动服务器

要启动服务器并从命令行界面接受连接：

```
$ pg_ctl start
```

要使用端口 5432 启动服务器并从命令行界面接受连接：

```
$ pg_ctl -o "-F -p 5433" start
```

### 停止服务器

要停止服务器，使用：

```
$ pg_ctl stop
```

要停止服务器并从命令行界面接受连接：

```
$ pg_ctl stop -m smart
```

### 重启服务器

要启动服务器并从命令行界面接受连接。不过 `pg_ctl` 将从保持开启的连接的运行实例的命令行启动。要以和之前相同的连接启动服务器，使用：

```
$ pg_ctl restart
```

要启动服务器并从命令行界面接受连接：

```
$ pg_ctl -o "-F -p 5433" restart
```

### 显示服务器状态

这是 `pg_ctl` 的完整输出的例子：

```
$ pg_ctl status
```

```
pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

第二行是在显示状态之后被调用的命令行。

## pg\_resetwal

`pg_resetwal` — 使用一个空的SQL脚本来重置归档写入点以及从其他控制连接

## 大纲

`pg_resetwal` [-f] [-force] [-s] [-dry-run] [options] [-D] [-pgdata] [dataDir]





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 WALS 文件）。默认是当前目录中的 `pg_test_fsync.out`。

\* -3--MCA-per-trat

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，但是它需要更多的同步运行。默认是5秒，这个庄程序在2分钟以内完成。

### **• -Y -version**

行到pg\_stat\_fsync状态并退出。

= -7--help

显示有关pg\_test\_fsync命令行参数的帮助并且退出。

丁亥立

## 环境

pg test tm

## pg\_test\_timing — 测量计时开销

大綱

牛頂

FFmpeg-test-timing@63-22-0-11:

= -7 --help

四

2004年北京马拉松（n=900）的第10名计时器读数为5:45

ANSWER: **100** (100% of the total area)

Testing timing overhead for 3 seconds.

Per loop time including overhe

```
istogram of timing durations:
<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。循环的解剖度可以在几个纳秒 (ns)，而单个计时调用只能解析到一个微秒 (μs)。

## 反重执行器设计开销

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
```

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

0.462 秒 64000 行的数表耗时 0.46 ms 的 EXPLAIN ANALYZE 基本的需要 12.02 ms。假设分析耗时 12.02 行上耗时 1.6 ms 的剩余耗时在分析上耗时为 1.6 ms。大概耗时 pg\_read\_page\_header 的 1/1000。假设这样耗时少的的话也就成了分析时间的数倍即耗时多出了 70ms。在更大的意义上，分析时间耗时的因子会随文件增大。

## 改变时间来源Changing time sources

在一些较老的 Linux 系统上，可以在任何时候更改系统时钟源的时钟来源。第二个例子显示了在上述快照使用的内核上切换时钟源为 acpi\_pm 的时间可能带来的影响。

```
# 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:
< us % of total count
 1 27.84870 1155682
 2 72.05956 2990371
 4 0.07810 3241
 8 0.01357 563
16 0.00007 3
```

在这种情况中，上面的例子 EXPLAIN ANALYZE 耗了 1.253 ms，比平均 1061 ns 的时钟耗时，说明这个工具在时钟切换时的一个小问题。这几乎的计算耗时被浪费到时钟本身只占了时间的一小部分。大部分的时间消耗在了时钟源的切换操作上。在这种情况下，选择时钟源时谨慎的 EXPLAIN ANALYZE 能会受到时钟耗时的影响。

freeBSD 也会出现时钟切换问题，并且它会记录在启动时间或每分钟时钟切换的次数。

```
# 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
```

某些系统可能对时钟切换时间敏感。在旧的 Linux 系统上，“clock”内核设置被认为更好的唯一方法，并且是在一些较老的系统上，对于一个时钟源的时钟切换唯一的选择是“clock”。但是，如果使用 Linux 的时钟模块，当它定期更新时钟时，它将根据时钟的精度，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:
< us % of total count
 1 90.23734 27694571
 2 9.75277 2993204
```



- -version

显示版本信息，然后退出

- -clean

使用相同的文件名（在一些系统上它被称为“softlink”），然后是文件或目录的完整路径。这可以导致对文件或目录的硬链接。从命令行 `ln -s link destination` 时的命令行参数不使用。文件将只在某些操作系统上被识别。如果连一个都不执行，`pg_upgrade` 将对 `pg_upgrade` 上的 `breakDOW`（文件系统的 `breakDOW`，以 `breakDOW` 为前缀）

- -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. 备份数据库 (可选)

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

不需要在启动。

2. 备份启动参数，编辑配置

所要修改的配置项 `config` 都是连接到 IvorySQL 的。在开始升级之后，`pg_upgrade` 将对 `pg_control_data` 来修改所有设置都是兼容的。

3. 备份新的 IvorySQL 二进制文件

```
make prefix=/usr/local/pgsql.new install
```

4. 备份现有的 IvorySQL 集群

运行 `initdb` 完成启动脚本。这是必要的因为 `pg_upgrade` 的 `initdb` 会自动地将所有的文件存放在新的位置。这将不会必要的地重新启动。

5. 备份扩展脚本对象文件

所有扩展脚本对对象文件必须是二进制的。如果 `pg_upgrade` 在运行时发现对象文件不是二进制的，它将自动地将所有的对象文件转换为二进制。如果对象文件不是二进制的，它将尝试在新集群中，通常通过脚本文件，而不是直接对对象文件，来重新定义对象。如果对象文件不是二进制的，`pg_upgrade` 将报告一个错误，因为它无法将对象文件转换为二进制。

6. 备份所有的文本对象文件

从对象脚本对对象文件（对对象文件的二进制表示，对义元，脚本，命令）

7. 适当认证

`pg_upgrade` 将会尝试连接到旧的服务器和新的服务器。因此，当可能想要有 `pg_hba.conf` 不同的设置时 `pg` 可能需要一个 `~/.pgpass` 文件。

8. 停止两个服务器

```
pg_ctl -D /opt/IvorySQL/1.5 stop
pg_ctl -D /opt/IvorySQL/2.1 stop
```

或者在 Windows 上使用正确的服务器：

255

## NET STOP postgresql-1.5 NET STOP postgresql-2.1

直到启动的计算机上，恢复和日志文件后备服务器可以保持运行。

4. 为后备服务器升级准备

开始准备服务器时，通常的主要操作后备服务器的 `log_minimal_writes` 和 `log_minimal_writes_start_time`。将此“`latest_checkpoint`”设置为需要的值（如果启用了 `min_writes` 或 `min_time` 后备服务器在运行主服务器之前将无法运行）。此外，确保保存到的主要上的 `postgresql.conf` 文件中，`wal_level` 为设置为 `minimal`。

22.0.0\pg\_upgrade

这是你执行的升级操作不是对服务器的 `pg_upgrade` 二进制文件。 `pg_upgrade` 要求使用 `pg_upgrade` 的文件（`1.5`）版本。你也可以使用 `pg_ctl` 和 `pg_ctl`，以作为后备想要继续使用二进制文件的数据库文件进行处理。

如果使用的是恢复模式，你将不会使用（不需要文件恢复）并使用更少的磁盘空间，但是在升级时一定会自动恢复。当使用恢复模式时，被操作的数据将来自恢复的备份位置（同一文件系统中）`(\emptyset)`。恢复模式提供了相同的恢复以及磁盘空间优势，但会将数据恢复到不同的位置。恢复模式只适用于数据恢复到同一文件系统中。此模式对某些操作系统和文件系统上可用。

**-jids** 选择使用多个CPU核心来恢复要恢复的数据文件以及进行物理和逻辑数据的转换。这个选项是一个计数的值是CPU核心数和空闲时间的最大值。这个选项可以是单机或在运行在同一多处理器上的多线程服务器上的数据恢复线程的值。

对于Windows用户，必须以一个超级用户登录，并以 `postgres` 用户启动一个 shell 并设置正确的路径：

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\IvorySQL\14\bin;
```

并启动命令的路径到 `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` 将验证两个集群是否兼容并运行。你可以使用 `pg_upgrade --check` 不运行检查，这种方式将使你在安装升级后运行时也能使用。 `pg_upgrade --check` 也将开始运行在更新后需要做的手工调整。如果你需要使用特定的恢复模式，你将使用 `--link` 或 `--clone` 选项和 `--check` 一起来选择恢复模式和你的集群。`pg_upgrade` 不将在当前命令中等待。

虽然，没有人可以在升级期间访问数据库。`pg_upgrade` 和从 `1.5` 版本到 `2.1` 版本的兼容性是完全的。在操作时，可以对两个集群使用相同的端口号，因为两个集群不会在同一时间运行。不过，为了避免一个端口的冲突，新的端口号必须不同。

如果在恢复数据时遇到错误，`pg_upgrade` 将会退出并告诉必须重新启动数据库。要再次尝试 `pg_upgrade`，你将需要将上次的恢复点，如果这是 `1.5` 版本的集群，你需要从 `1.5` 版本的集群中启动并运行 `pg_start` 在新的集群中。不过这样做的前提是集群没有使用其他用户的数据。

11. 准备恢复和日志文件后备服务器

如果使用恢复模式并且有其他集群正在运行后备服务器，你可以通过下面的步骤对它们进行兼容的升级。你将首先在现有后备服务器上运行 `pg_upgrade`，然后在主服务器上运行 `pg_upgrade`，但这并不是必须的。

如果还没有使用恢复模式，没有或不能使用 `pg_upgrade` 一种更好的解决方案，请阅读下一节中的过程并从 `pg_upgrade` 或从日志的主服务器开始运行兼容的集群。

1. 在后备服务器上安装的 `pg_upgrade`

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. 确保不存在新的后台数据库

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. 安装新集群的后台数据库

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。

如果后备服务器仍在运行，现在使用上述的指令停止它们。

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。

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。

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)。

22.修改 [pg\\_hba.conf](#)

如果你修改了 `pg_hba.conf`，则要将其恢复到原始的设置。你可能需要调整新安装中的其他配置文件来匹配替换，例如 `postgresql.conf`（以及它包含的任何文件）和 `postgresql.auto.conf`。

11. 痘痘的治疗与护理

我们在窗口命令行启动的23端口823，并且窗口设置自动执行的cmdc过的后台命令

334 *W. H. H. Ho*

如果需要做什么升级操作，`pe_upgrade` 将在完成时发出警告。它也将生成必须将管理器运行的脚本文件。这些脚本文件将连接到每一个需要做升级操作的数据源。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。

小明

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。

關於 ~~總~~ 聲

在运行 `pg_upgrade` 之后，如果你希望你复制到旧集群，有几个选项：

- 如果使用了 `-check` 选项，对所有的静态方法而言，它可能需要修改。
- 如果 `-link` 选项是启用的，对所有的静态方法而言，它可能需要修改。
- 如果使用了 `-link` 选项，对所有在静态方法中引用的类而言，它可能需要修改。
  - 如果启用了 `cg_reapply` 选项的静态方法，类本身没有被修改，它可能需要修改。
  - 如果类没有被修改，但静态方法引用了它，`cg_reapply` 选项的静态方法将调用 `SPAGNA/global/ag_ctrl.ref`，如果更多类引用，如 `SPAGNA/global/ag_ctrl.ref->A4` 时，将对类进行递归修改。

## 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在新旧数据库中启动缺省的postmaster。临时 Unix 套接字文件用于这些 postmaster 通信。默认情况下，在当前工作目录中运行。在某些情况下，当前目录的宿主名称可能太长，无法成为有效的套接字名称。这种情况下你可以使用 -A 选项将套接字文件放在某些具有较短宿主名称的目录中。为了安全原因，确保该目录不可被任何其他用户读取或写入。（这在 Windows 上不受支持）

如果失败，重建和重新启动将影响你的安装，`pg_upgrade` 将会报告这些情况。用来重建和重新启动的升级后版本将不会自动被建立。如果你正在尝试自动升级很多集群，你应该发现具有限制数据模式的策略。对所有集群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流模式而不是用户数据。

对于部署测试，创建一个只有模式的部署族副本，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体的升级。

regcollation  
regconfig  
regdictionary  
regnamespace  
regoper  
regoperator  
regproc  
regprocedure

如果想要使用连接模式并开启不想让的连接模式需要通过配置文件，考虑使用连接模式。如果使用连接模式不可用，可以尝试一些连接模式并在副本上以链接模式进行操作。要编辑连接的台账文件，请从可以在服务器端时使用 `rsync` 命令直接编辑。然后关闭连接并从副本再次运行 `rsync --checksum` 将更改更新到连接以让其（`--checksum` 是必要的，因为 `rsync` 在跳过文件时会尝试更多的精简才能识别到）。你可能想要编辑一些文件，例如 `postmaster.pid`，如果文件的文件系统支持文件锁定或者 `copy-on-write`

# pg\_waldump

pg\_waldump – 从人可读的格式读取一个MySQL 数据集的原生式文本

## 大纲

pg\_waldump option (starting endpoint)

## 选项

下列命令行选项可输出的任意形式。

- starting

从指定的日记文件开始读取。这将告知读取了要读取文件的路径以及要使用的时间点。

- ending

在读取指定的日记文件后停止。

-A -skip-details

输出没有备注的细节。

-A -end -end-end

在指定的日记文件停止读取，而不是一读取到的日记的末尾。

-A -follow

在到达可读 MySQL 的末尾之后，保持并检测一次是否有新的 MySQL 进程。

-A -last -last-last

显示指定数量的记录，然后停止。

-A path -path-path

指定要读取日记文件的路径或日记文件的名称 pg\_wal 子目录的目录。如果是在当前目录中使用，当前目录的 pg\_wal 子目录和 PGDATA 或 pg\_wal 无关。

-A -q -quiet

输出语句，不要打印到输出。当您想知道一条日记记录是否可以读取时也不要关心记录的时，此选项非常有用。

-A reg --regexp

只显示满足正则表达式匹配的记录。如果使用 list 作为读取管理者名称将通过这个过滤，同时只读取读管理者名称的记录并返回。

-A start -startstart

要从哪个 MySQL 端开始读取，假以是从文件的最简单的文件的第一个可用日记记录开始。

-A 'timeline' --timeline 'timeline'

要从哪个时间线读取日记。假以是使用 starting (如果启用) 中的值，否则默认为 1。

-A -version

从 pg\_waldump 读取并显示。

-A -xid -xidid

只是读取交易 ID 以标记的记录。

-A -stats[record]

显示统计 (记录的数目和行数) 或者记录 (记录的数目)。这不是显示每个记录，而是显示所有记录。如果要显示所有记录，必须使用 -A 选项。

-A -help

显示有关 pg\_waldump 的所有帮助的短语并退出。

## 环境

- PGDATA

数据目录，如果未设置，则为。

- PG\_CLOB

规定在读取记录时是否使用转义字符。可能的值为 always, auto, never。

## 注解

当阅读正在运行时可能会影响读取的记录。

只有指定的文件名会使用 (如果没有指定，则是上一个时间点)。其他时间点上的文件将被忽略。

pg\_waldump 不读取具有后缀 partial 的 MySQL 文件。如果需要读取这些文件，需要从文件名中移除 partial 后缀。

## postgres

postgres – MySQL 数据库名称



为单线程锁的锁。这个选项用于 `recheck`。

• -S

设置系统对连接锁的锁。这个选项对连接池的连接锁起作用。这是从您的连接池中锁定连接。

• - (psql) | planner | a)actor

对于每个主要线程锁的连接时间点，这个选项不能和 -A 选项一起使用。

• -T

此选项主要用作对服务器连接的连接锁的锁。对于连接锁的锁，如果锁被占用，它们将无法获得锁的共享锁和独占锁。这是因为一个锁的服务器锁将阻止其他已经共享锁的锁被释放。该选项向 `postgres` 并通过 `SSL` 语句禁止其他所有锁的锁。

• -protocol

声明这是从连接池的连接服务器对连接本机。该选项不能与 -a 选项一起使用。

• -seconds

在单线程锁的锁的锁，它实现延迟连接之连接延迟锁的锁。这将阻止新连接在单线程锁的锁。

## 用于单用户模式的选项

下面的选项仅适用于单用户模式（在 `single-user mode`）。

• -single

选择单用户模式。这是命令行中的最后一个选项。

• database

指定要访问的数据表的名称。这必须是命令行中的最后一个参数。如果省略它，则假定为用户名。

• -

在命令行命令之后显示的任何参数都将被忽略。

• -i

使用配置文件（如果有的话）而不是命令行参数作为单线程锁。

• -f file

将所有配置信息输出到 `file`。只有当作为命令行选项使用时，这个选项才会有效。

## 环境

• PGHOSTENCODING

客户端使用的字符集（字符串）（字符串可以是字符集或 `utf8`）。这个值也可以在配置文件中设置。

• PGDATA

从环境变量中读取。

• PGTABLESTYLE

`psql` 使用的参数的默认设置（这个特殊变量的使用已被废弃）。

• PGPORT

从环境变量中读取（无覆盖字符串设置）。

## 诊断

一个使用了 `smp` 或 `shard` 的连接将需要配置的本地连接共享锁和锁字典。你也可以通过将 `client_block` 或 `block` 为 MySQL 的共享内存连接，或者将 `max_connections` 增加到锁字典，这样可以更好的对锁的资源分配。

如果一个消息从另一个服务器已经运行，它将打印出来，例如本地连接，例如连接到的系统可以将命令

```
$ ps ax | grep postgres
```

如果连接没有从本地服务器运行，那么你可以删除连接共享锁的文件然后再次启动。

如果一个本地消息显示它无法锁定一个连接，可能你将连接口已经连接到 MySQL，但没有使用。如果使用 `postgres` 并且它无法锁定一个连接，它也可能由于连接池的锁。在这种情况下，你必须将连接池的连接数设置为 1，然后重新启动。

```
$ ps -ef | grep postgres
```

## 注解

关闭命令 `psql` 可以将单线程锁的锁的实现 `postgres` 是线程。

只要可能，对主要的 `SSL` 会话，`postgres` 将在连接到它所有的锁的连接（包括共享锁和锁字典）。这样可能是导致连接到 `postgres` 无法启动的原因。

如果同时对 `postgres` 有线程，可以使用 `SSL`，`SSL` 或者 `SSL` 会话。第一个在连接到所有的连接有共享锁上，第二个将连接到所有的连接有锁字典。第三个将连接到所有的连接有锁字典。

`SSL` 将会重新加载线程为线程文件。也可以对一个单独的连接设置为线程文件，但是这样就将连接设为共享。

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台进程，可向它发送 `SIGHUP`。在 SQL 中可调用的与这两种动作等效的命令参考 `pg_cancel_backend` 和 `pg_terminate_backend`。

`postgres` 服务器使用 `SIGQUIT` 来告诉该服务器进程停止而不响应来的清理。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—同一 `postgres` 进程将把这解释为一次崩溃，并且作为其启始崩溃恢复过程的一部分，它将强制所有的后台进程退出。

## 缺陷

选择在FreeRTOS或OpenRTOS上手试运行，具体步骤参见启动后的系统是个特例：如果这个特例没有被修复，该类的Linux/OS版本将难以一起解决方法。

## 单用户模式

要启动一个非用户模式的服务器，使用这样的命令：

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

用 -D 级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将执行的当命令插入的终止符。它不明白分号的作用，因为那是属于psql。要想把一个命令分成多行，必须在最后一个执行符之外的每个执行行前面敲一个反斜线。这个反斜线和旁边的执行行都会被从输入命令中去掉。注意即使在字符串或者注册中也会这样处理。

也就是说如果使用了`-E`命令行选项，那么整个脚本将不会停止命令输入。相反，另外一行命令的输入才会停止命令输入。也就是说，输入一个没有命令的分号。在这种模式下，反斜杠`\`行将不会被特殊对待。此外，在字符串或者注释内的反斜杠将不会被特殊对待。

## 例子

要用默认值在后台启动 postgres

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

要用指吧端口启动 `postgres`，例如 123-

```
$ 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。除了用于简短的实验外，更好的方法是编辑 `postgresql.conf` 中的设置，而不是临时命令行开关来设置参数。

# Chapter 2. FAQ

## IvorySQL贡献的许可

如果我提交的贡献是啥子性质的，那么我必须遵循IvorySQL版本的一套贡献协议吗？当然，遵循本项目遵循Apache2.0许可（2.0版本）。

如果我提交的代码不是源代码作品，同时我自己是其工具和库的贡献者，同时也有自己的作品，该如何处理呢？首先，建议你遵循Apache2.0许可。

1. 需要使用Apache2.0许可。

2. 如果你修改了代码，需要在修改过的文件中添加。

3. 在兄弟的代码中（修改和有著作权的代码）需要尊重贡献者的协议，同时，专利声明和其他贡献者同样需要包含的协议。

4. 如果再贡献的产品中包含一个Notice文件，则在Notice文件中需要带有Apache2.0许可，同时可以在Notice中增加自己的许可，也可以考虑使用Apache2.0许可进行更改。

最后，建议你，从半开放的代码中删除你从头开始不是一个分支。如果你的代码部分有对称的分支，建议你将所有分支都统一为一个分支。如果这个分支包含你的贡献的代码，请将这个分支从贡献列表中移除。

## 编码指南

如果再贡献和贡献的代码在贡献中的时候在很大程度上没有大的变化，如果想对想法发生了更大的变化，我们强烈建议你把大量的时间写代码之后，向国人和友人贡献你的代码，并为他们提供你的建议。即使你的建议得不到采纳，我们的建议将对你的工作有所帮助。建议你将你的工作写成分支，并提出了合理的文件名。

当使用IvorySQL和PostgreSQL时，我们建议使用PostgreSQL的编码格式，除此之外：

对于C/C++代码，如果需要，使用Tpproviders。我们建议在查看更改时使用git diff --color，这样提交文件时就不会出现任何奇怪的显示问题。

所有贡献IvorySQL的贡献者必须遵循其相关的代码规范。如果还不清楚如何进行贡献的工作，请在IvorySQL仓库中查看相关说明，社区的许多人将乐于帮助你。

另外，建议使用命令make installcheck test，以确保它没有潜在的内存泄漏。

## 适用于上游PostgreSQL的更改

如果你正在使用PostgreSQL或IvorySQL之间的通用功能，同时需要对两者都进行修改，那么你需要将两者都贡献到PostgreSQL或IvorySQL。这将是为了让你对PostgreSQL和IvorySQL的任何变化都从社区PostgreSQL“第一次”收到的审查中受益。一般来说，将这两个代码库放在一个分支上是十分重要的，这样它们可以满足你的更改所需要的经历。

## 补丁提交

一旦你准备好了IvorySQL库（或IvorySQL社区的其他成员共享的工作），建议你将所有贡献送到IvorySQL或IvorySQL社区的贡献者共享的工作中，并向他们发送通知。

## 补丁审查

我们建议你取最新的IvorySQL代码并使用IvorySQL社区的其他成员共享的工作来审查。同时发送邮件至IvorySQL仓库的讨论组，询问他们对你的贡献的看法。我们建议你发送IvorySQL社区的每个人对你的贡献的看法。虽然这不必成为核心贡献者或贡献者可以采纳这一条，因此我们建议你向IvorySQL仓库贡献者的核心贡献者发送一封电子邮件。

同时你也可以将你的贡献发送到IvorySQL仓库的讨论组，询问他们对你的贡献的看法。同时发送邮件至IvorySQL仓库的讨论组，询问他们对你的贡献的看法。

当向IvorySQL仓库发送贡献时，建议你使用git commit命令的“-s”选项，同时发送邮件至IvorySQL仓库的讨论组，询问他们对你的贡献的看法。

在发送贡献时，建议你使用git commit命令的“-s”选项，同时发送邮件至IvorySQL仓库的讨论组，询问他们对你的贡献的看法。

## 直接提交到存储库

有时，你会发现你的核心贡献从贡献者那里接收反馈。然而你进行的贡献工作，这可能是一个小的更改。我们建议你将贡献：如果你修改了任何对你的贡献者来说是重要的问题，那么它必须通过IvorySQL仓库的工作。另一方面，如果你修改了在代码库的单独部分（例如在主程序中修复了一个错误），则核心贡献者可以接受你的贡献。