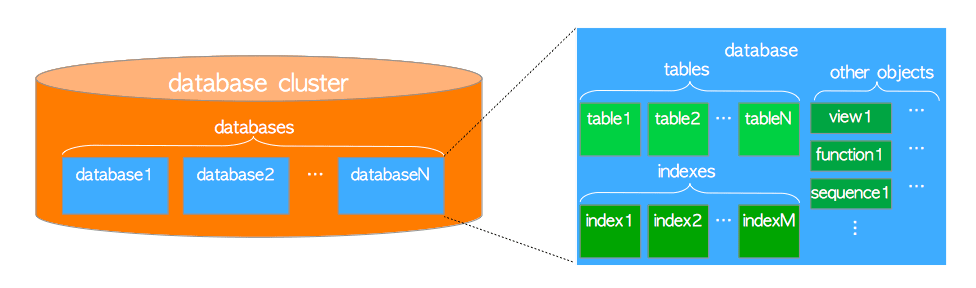
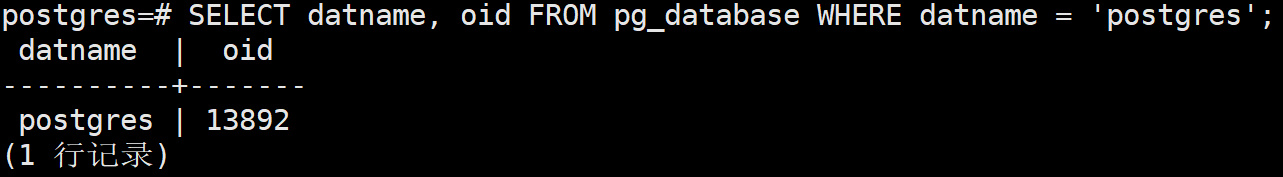
**逻辑存储结构**

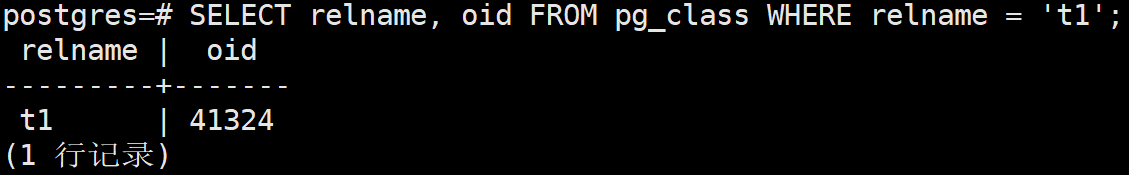
数据库集簇（database cluster）是一组数据库（database） 的集合，由一个PostgreSQL服务器管理。

数据库（database）是数据库对象（database objects）的集合，例如表、索引，序列，视图，函数等。 在PostgreSQL中数据库本身也是数据库对象，并在逻辑上彼此分离。 所有其他的数据库对象（例如表，索引等）归属于各自相应的数据库。



在PostgreSQL内部，所有的数据库对象都通过相应的对象标识符（Object Identifiers, OID）进行管理，这些标识符是无符号的4字节整型。数据库对象与相应OID之间的关系存储在相应的系统目录中，依具体的对象类型而异。 例如数据库和堆表对象的OID分别存储在pg\_database和pg\_class中：





**软件目录结构**

bin:二进制可执行文件

Include:头文件

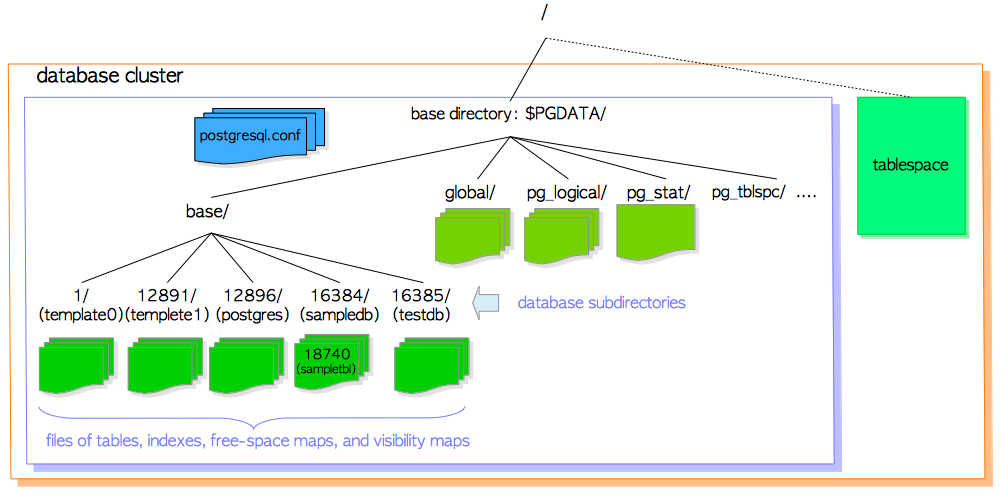
lib:动态链接库

share:文档、模板配置文件、extension

**物理存储结构**

数据库集簇在本质上就是一个文件目录（基础目录 base directory），包含着一系列子目录与文件。 执行 initdb 命令会在指定目录下创建基础目录从而初始化一个新的数据库集簇。 通常会将基础目录的路径配置到环境变量PGDATA中。

base子目录中的每一个子目录都对应一个数据库，数据库中每个表和索引都会在相应子目录下存储为（至少）一个文件；还有几个包含特定数据的子目录，以及配置文件。 PostgreSQL也支持表空间（Tablespace）。 PostgreSQL中的表空间对应一个包含基础目录之外数据的目录。



**数据库集簇的布局**

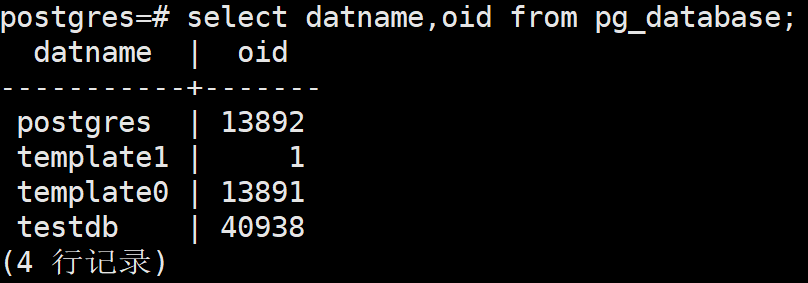
数据库集簇主要的文件与子目录：

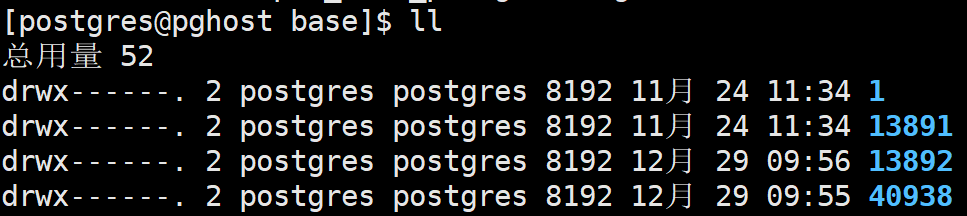
|  |  |
| --- | --- |
| 文件 | 描述 |
| current\_logfiles | 记录当前使用日志文件 |
| PG\_VERSION | 包含PostgreSQL主版本号 |
| pg\_hba.conf | 控制PosgreSQL客户端认证 |
| pg\_ident.conf | 控制PostgreSQL用户名映射 |
| postgresql.conf | 配置参数 |
| postgresql.auto.conf | 存储使用 ALTER SYSTEM 修改的配置参数（9.4或更新版本） |
| postmaster.opts | 记录服务器上次启动的命令行选项 |
| postmaster.pid | 锁文件，记录当前的数据库状态 |

|  |  |
| --- | --- |
| 子目录 | 描述 |
| base/ | 每个数据库对应的子目录存储于此 |
| global/ | 数据库集簇范畴的表（例如pg\_database），以及pg\_control文件。 |
| log/ | 存放日志文件 |
| pg\_commit\_ts/ | 事务提交的时间戳数据（9.5及更新版本）。 |
| pg\_dynshmem/ | 动态共享内存子系统中使用的文件（9.4或更新版本）。 |
| pg\_logical/ | 逻辑解码的状态数据（9.4或更新版本）。 |
| pg\_multixact/ | 多事务状态数据 |
| pg\_notify/ | LISTEN/NOTIFY 状态数据 |
| pg\_repslot/ | 复制槽数据（9.4或更新版本）。 |
| pg\_serial/ | 已提交的可串行化事务相关信息（9.1或更新版本） |
| pg\_snapshots/ | 导出快照（9.2或更新版本）。 PostgreSQL函数pg\_export\_snapshot在此子目录中创建快照信息文件。 |
| pg\_stat/ | 统计子系统的永久文件 |
| pg\_stat\_tmp/ | 统计子系统的临时文件 |
| pg\_subtrans/ | 子事务状态数据 |
| pg\_tblspc/ | 指向表空间的符号链接 |
| pg\_twophase/ | 两阶段事务（prepared transactions）的状态文件 |
| pg\_wal/ (10+) | WAL（ Write Ahead Logging）段文件（10或更新版本），从pg\_xlog重命名而来。 |
| pg\_xact/ (10+) | 事务提交状态数据，（10或更新版本），从pg\_clog重命名而来。 |
| pg\_xlog/ (9.6-) | WAL（Write Ahead Logging）段文件（9.6及更老版本），它在版本10中被重命名为pg\_wal。 |

**数据库布局**

一个数据库与base子目录下的一个子目录对应；且该子目录的名称与相应数据库的OID相同。 例如当数据库testdb的OID为40938时，它对应的子目录名称即为40938。

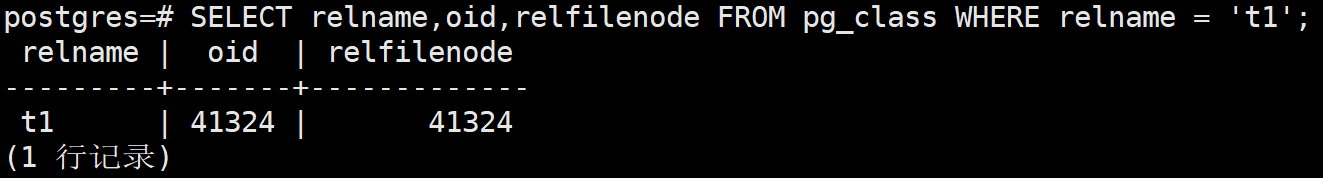




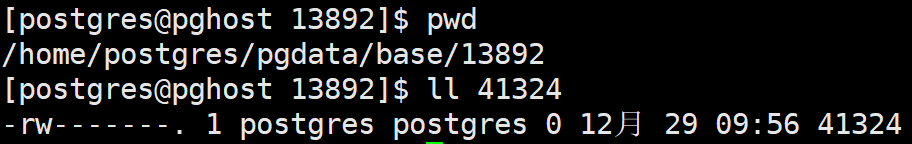
**表与索引相关文件的布局**

每个小于1GB的表或索引都在相应的数据库目录中存储为单个文件。在数据库内部，表和索引作为数据库对象是通过OID来管理的，而这些数据文件则由变量relfilenode管理。 表和索引的relfilenode值通常与其OID一致，但也有例外。

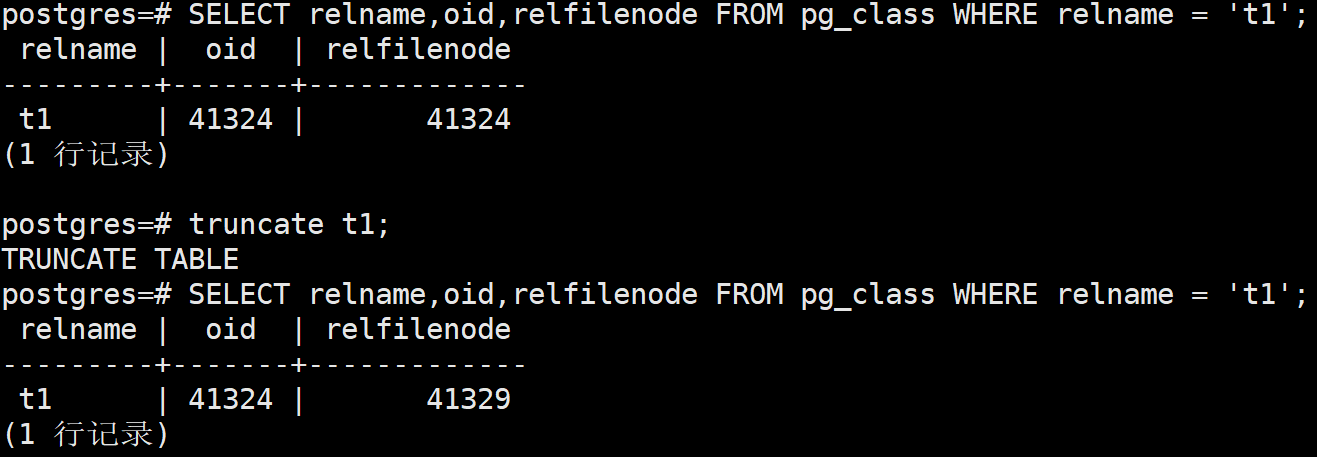
查看表t1的oid和relfilenode：



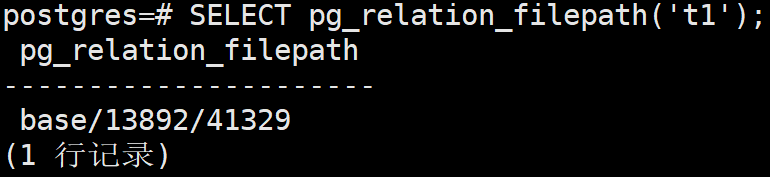
从上面的结果可以看出oid和relfilenode值相等。还可以看到表t1的数据文件路径是base/13892/41324。



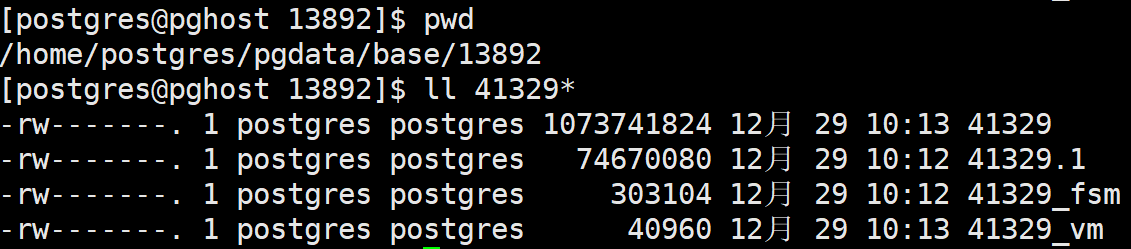
表和索引的relfilenode值会被一些命令（例如TRUNCATE，REINDEX，CLUSTER）所改变。 例如对表 t1 执行TRUNCATE，PostgreSQL会为表分配一个新的relfilenode（41329），删除旧的数据文件（41324），并创建一个新的数据文件（41329）。



可以使用内建函数 pg\_relation\_filepath 查看关系对应的文件路径。



当表和索引的文件大小超过1GB时，PostgreSQL会创建并使用一个名为relfilenode.1的新文件。如果新文件也填满了，则会创建下一个名为relfilenode.2的新文件，依此类推。



在构建PostgreSQL时，可以使用配置选项--with-segsize更改表和索引的最大文件大小。

每个表都有两个与之相关联的文件，后缀分别为\_fsm和\_vm。它们是空闲空间映射（free space map）和可见性映射（visibility map） 文件，分别存储了表文件每个页面上的空闲空间信息与可见性信息。索引没有可见性映射文件，只有空闲空间映射文件。

在数据库系统内部，这些文件（主体数据文件，空闲空间映射文件，可见性映射文件等）也被称为相应关系的分支（fork）；空闲空间映射是表/索引数据文件的第一个分支（分支编号为1），可见性映射表是数据文件的第二个分支（分支编号为2），数据文件的分支编号为0。

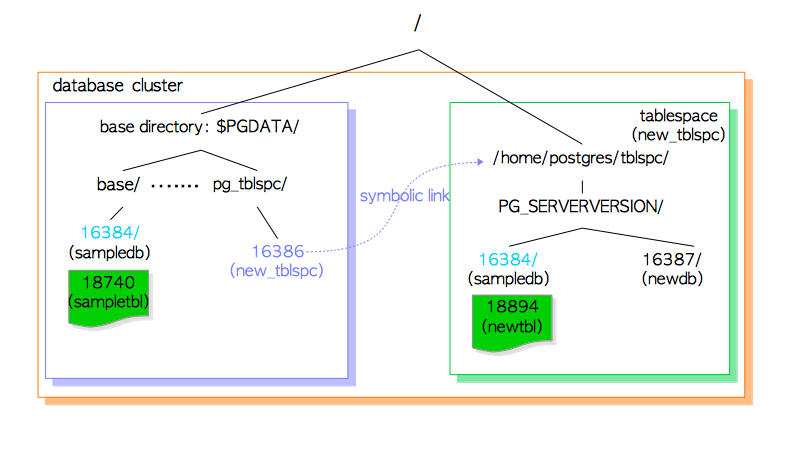
注意：每个 关系（relation） 可能会有四种分支，分支编号分别为0，1，2，3，0号分支main为关系数据文件本体，1号分支fsm保存了main分支中空闲空间的信息，2号分支vm保存了main分支中可见性的信息，3号分支init是很少见的特殊分支，通常用于不被日志记录（unlogged）的表与索引。

每个分支都会被存储为磁盘上的一到多个文件：PostgreSQL会将过大的分支文件切分为若干个段，以免文件的尺寸超过某些特定文件系统允许的大小，也便于一些归档工具进行并发复制，默认的段大小为1GB。

进程结构

**表空间**

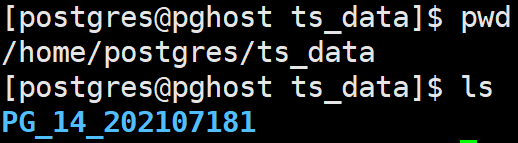
PostgreSQL中的表空间（Tablespace）是基础目录之外的附加数据区域。



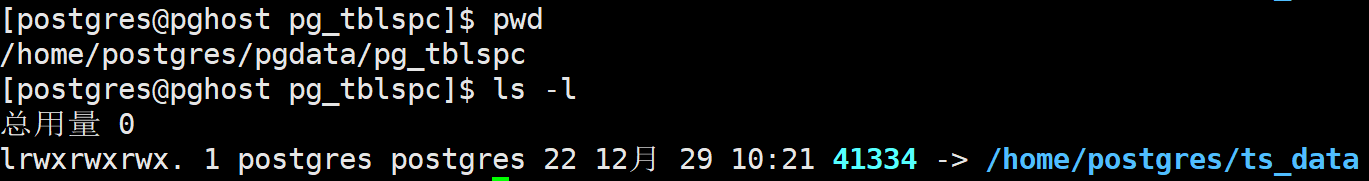
执行CREATE TABLESPACE语句会在指定的目录下创建表空间。而在该目录下还会创建版本特定的子目录（例如PG\_14\_202107181）。版本特定的命名方式为：

PG\_主版本号\_目录版本号

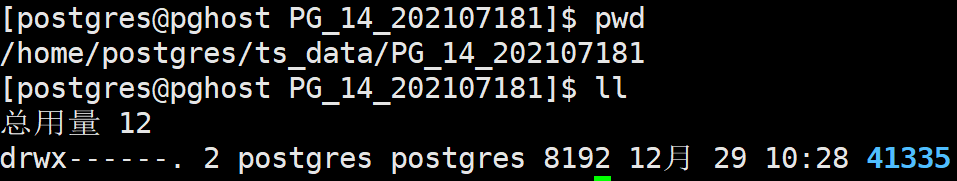
举个例子，如果在 /home/postgres/ts\_data 中创建一个表空间 ts\_data，其oid为41334，则会在表空间下创建一个名如PG\_14\_202107181的子目录。



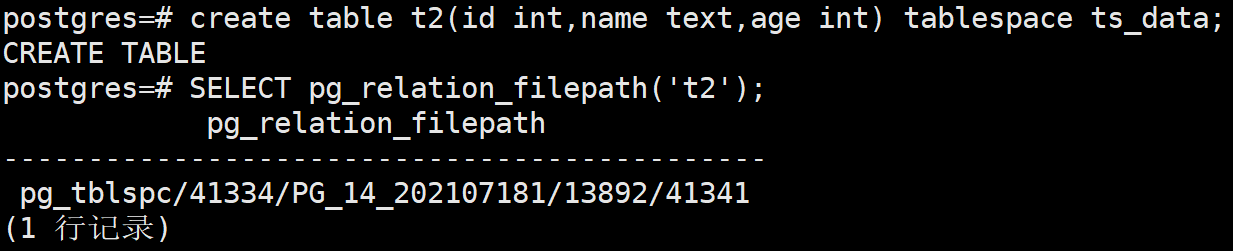
表空间目录通过pg\_tblspc子目录中的符号链接寻址，链接名称与表空间的OID值相同。



如果在该表空间下创建新的数据库（OID为41335），则会在版本特定的子目录下创建相应目录。



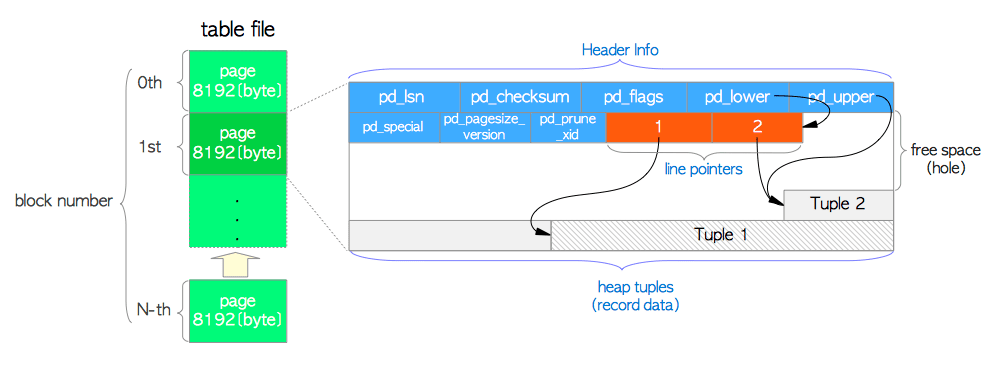
如果在该表空间内创建一个新表，那么PG会创建一个名称与现有数据库OID相同的新目录，然后将新表文件放置在刚创建的目录下。



**堆表的内部结构**

在数据文件（堆表，索引，也包括空闲空间映射和可见性映射）内部，它被划分为固定长度的页（pages），或 块（blocks），大小默认为8192字节（8KB）。 每个文件中的页从0开始按顺序编号，这些数字称为块号（block numbers）。 如果文件已填满，PostgreSQL通过在文件末尾追加一个新的空页来增长文件。

堆表的页面结构



表的页面包含了三种类型的数据：

1.堆元组（heap tuples） —— 堆元组就是数据记录本身。它们从页面底部开始依序堆叠。

2.行指针（line pointer） —— 每个行指针占4个字节，保存着指向堆元组的指针。它们也被称为项目指针（item pointer）。行指针简单地组织为一个数组，扮演了元组索引的角色。每个索引项从1开始依次编号，称为偏移号（offset number）。当向页面中添加新元组时，一个相应的新行指针也会被放入数组中，并指向新添加的元组。

3.首部数据（header data） —— 页面的起始位置分配了由结构PageHeaderData定义的首部数据。它的大小为24个字节，包含关于页面的元数据。该结构的主要成员变量为：

pd\_lsn —— 本页面最近一次变更所写入XLOG记录对应的LSN。它是一个8字节无符号整数，与WAL机制相关。

pd\_checksum —— 本页面的校验和值。

pd\_flags —— 标记位

pd\_lower —— pd\_lower指向行指针的末尾（空闲空间开始位置）。

pd\_upper —— pd\_upper指向最新堆元组的起始位置（空闲空间结束位置）。

pd\_special —— 在索引页中会用到该字段。在堆表页中它指向页尾。（在索引页中它指向特殊空间的起始位置，特殊空间是仅由索引使用的特殊数据区域，包含特定的数据，具体内容依索引的类型而定，如B树，GiST，GiN等。

pd\_pagesize\_version —— 页面的大小，以及页面布局的版本号

pd\_prune\_xid —— 本页面中可以修剪的最老的元组中的XID.

行指针的末尾与最新元组起始位置之间的空余空间称为空闲空间（free space）。

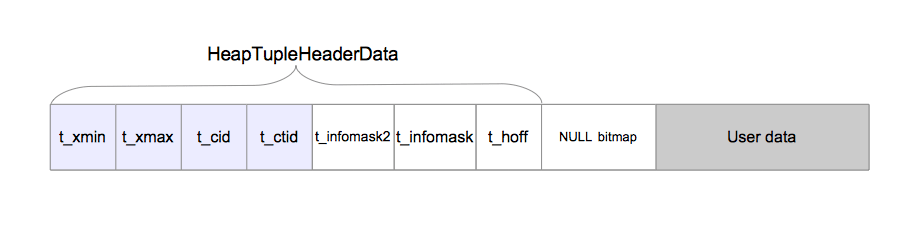
为了识别表中的元组，数据库内部会使用元组标识符（tuple identifier, TID）。TID由一对值组成：元组所属页面的区块号，及指向元组的行指针的偏移号。

此外，大小超过约2KB（8KB的四分之一）的堆元组会使用一种称为 TOAST（The Oversized-Attribute Storage Technique，超大属性存储技术） 的方法来存储与管理。

**元组结构**

可以将表页中的堆元组分为两类：普通数据元组与TOAST元组。

堆元组由三个部分组成，即HeapTupleHeaderData结构，空值位图，以及用户数据。



虽然HeapTupleHeaderData结构包含七个字段。

t\_xmin保存插入此元组的事务的txid。

t\_xmax保存删除或更新此元组的事务的txid。如果尚未删除或更新此元组，则t\_xmax设置为0，即无效。

t\_cid保存命令标识（command id, cid），cid意思是在当前事务中，执行当前命令之前执行了多少SQL命令，从零开始计数。例如，假设我们在单个事务中执行了三条INSERT命令BEGIN;INSERT;INSERT;INSERT;COMMIT;。如果第一条命令插入此元组，则该元组的t\_cid会被设置为0。如果第二条命令插入此元组，则其t\_cid会被设置为1，依此类推。

t\_ctid保存着指向自身或新元组的元组标识符（tid）。在更新该元组时，其t\_ctid会指向新版本的元组；否则t\_ctid会指向自己。

t\_infomask2 属性掩码，记录元组中字段个数、HOT（heap only tuple，堆内元组）更新标记等。

t\_infomask 属性掩码，记录是否有空字段标记、是否有变长字段标记、是否有外部TOAST（the oversized-attribute storage technique，过长字段存储技术）标记、是否有OID字段标记、是否有压缩标记、插入事务是否提交/回滚标记、删除事务是否提交/回滚标记、是否被更新标记等。

t\_hoff 到用户数据的偏移量

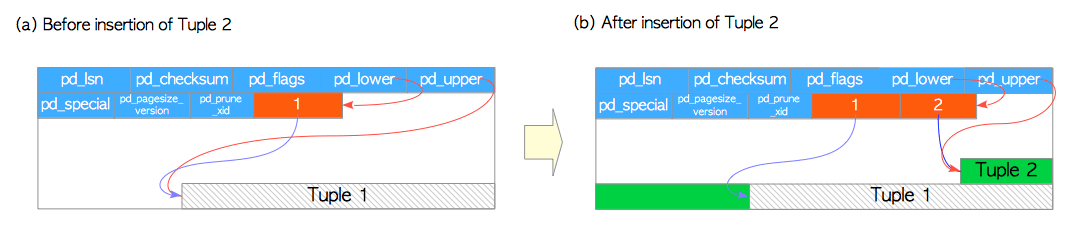
NULL bitmap 空值位图，有null值，这里会记录

**读写元组的方式**

**写入堆元组**

假设有一个表，仅由一个页面组成，且该页面只包含一个堆元组。 此页面的pd\_lower指向第一个行指针，而该行指针和pd\_upper都指向第一个堆元组。

当第二个元组被插入时，它会被放在第一个元组之后。第二个行指针被插入到第一个行指针的后面，并指向第二个元组。 pd\_lower更改为指向第二个行指针，pd\_upper更改为指向第二个堆元组。 页面内的首部数据（例如pd\_lsn，pg\_checksum，pg\_flag）也会被改写为适当的值。

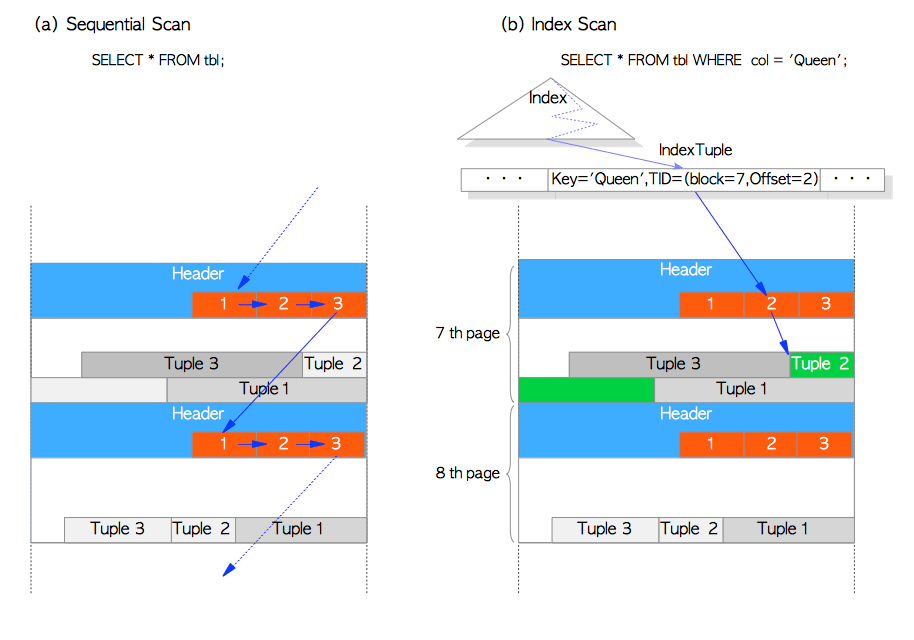


**读取堆元组**

这里简述两种典型的访问方式：顺序扫描与B树索引扫描：

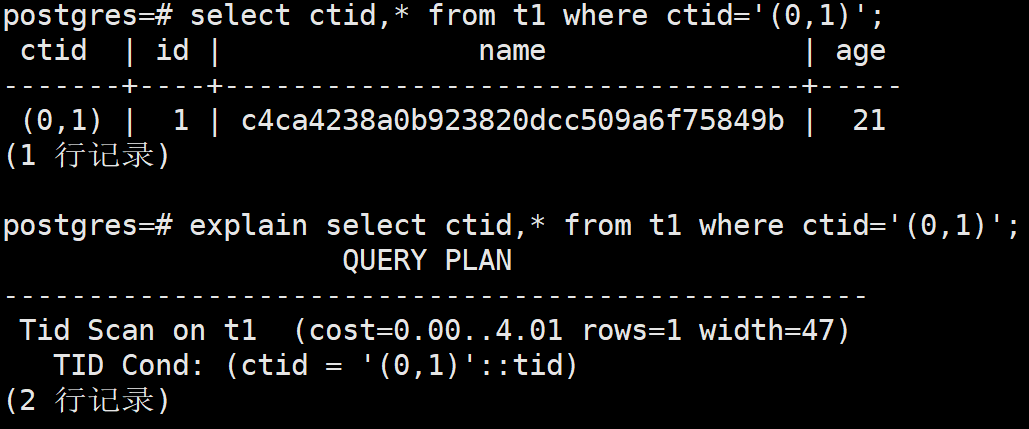
顺序扫描 —— 通过扫描每一页中的行指针，依序读取所有页面中的所有元组。

B树索引扫描 —— 索引文件包含着索引元组，索引元组由一个键值对组成，键为被索引的列值，值为目标堆元组的TID。进行索引查询时，首先使用键进行查找，如果找到了对应的索引元组，PostgreSQL就会根据相应值中的TID来读取对应的堆元组 。例如在图(b)中，所获索引元组中TID的值为（区块号 = 7，偏移号 = 2）， 这意味着目标堆元组是表中第7页的第2个元组，因而PostgreSQL可以直接读取所需的堆元组，而避免对页面做不必要的扫描。



PostgreSQL还支持TID扫描，位图扫描（Bitmap-Scan）和仅索引扫描（Index-Only-Scan）。

TID扫描是一种通过使用所需元组的TID直接访问元组的方法。 例如要在表中找到第0个页面中的第1个元组，可以执行以下查询：



**进程结构**

PostgreSQL是一个客户端/服务器风格的关系型数据库管理系统，采用多进程架构，运行在单台主机上。

Postgresql 是一个C/S架构的关系型数据库，由多个后台进程管理数据库，下面分别介绍一些这些进程

postgres server process  是所有进程的父进程

backend process 每一个客户端的连接都有一个后端进程存在

backgroud processes 为管理数据库而产生的一些进程

​我们通常所谓的“PostgreSQL服务器（PostgreSQL Server）” 实际上是一系列协同工作的进程集合，包含着下列进程：

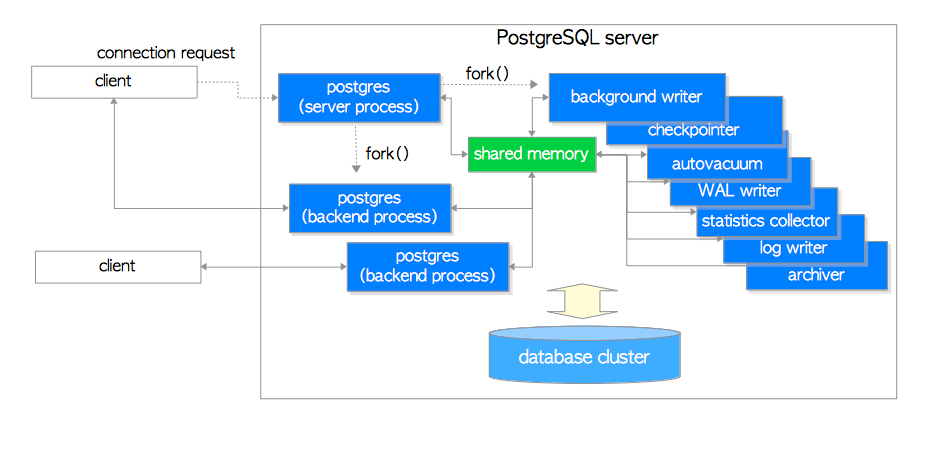
postgres服务器进程（Postgres Server Process）是所有进程的父进程。

后端进程（Backend Process）每一个客户端的连接都有一个后端进程存在，负责处理客户端发出的查询和语句。

各种后台进程（Background Process） 负责执行各种数据库管理任务（例如清理过程与检查点过程）。

各种复制相关（Replication Associated Process）的进程负责流复制。

后台工作进程（Background Worker Process） 在9.3版被引入，它能执行任意由用户实现的处理逻辑。



**Postgres服务器进程**

postgres服务器进程（postgres server process）是PostgreSQL服务器中所有进程的父进程，在早期版本中它被称为“postmaster“。

执行 pg\_ctl start 会启动一个postgres服务器进程。它会在内存中分配共享内存区域，启动各种后台进程，如有必要还会启动复制相关进程与后台工作进程，并等待来自客户端的连接请求。 每当接收到来自客户端的连接请求时，它都会启动一个后端进程 （然后由启动的后端进程处理该客户端发出的所有查询）。

一个postgres服务器进程只会监听一个网络端口，默认端口为5432。如果要在同一台主机上运行多个PostgreSQL服务器，则应为每个服务器配置不同的监听端口，如5432，5433等。

**后端进程**

每个后端进程（也称为”postgres“）由postgres服务器进程启动，并处理连接另一侧的客户端发出的所有查询。它通过单条TCP连接与客户端通信，并在客户端断开连接时终止。

因为一条连接只允许操作一个数据库，因此必须在连接到PostgreSQL服务器时显式指定要连接的数据库。

PostgreSQL允许多个客户端同时连接；配置参数max\_connections用于控制最大客户端连接数（默认为100）。

因为PostgreSQL没有原生的连接池功能，因此如果许多客户端频繁地重复与PostgreSQL服务器建立断开连接（譬如WEB应用），则会导致建立连接与创建后端进程的开销变大。这种情况对数据库服务器的性能有负面影响，通常可以使用池化中间件（pgbouncer或pgpool-II）来避免该问题。

**后台进程**

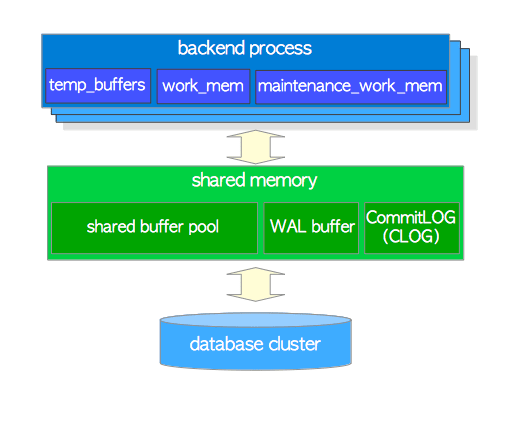
|  |  |
| --- | --- |
| 进程 | 概述 |
| background writer | 本进程负责将共享缓冲池中的脏页逐渐刷入持久化存储中（例如，HDD，SSD）（在9.1及更旧版本中，它还负责处理**检查点（checkpoint）**） |
| checkpointer | 在9.2及更新版本中，该进程负责处理检查点。 |
| autovacuum launcher | 周期性地启动自动清理工作进程（更准确地说，它向Postgres服务器请求创建自动清理工作进程） |
| WAL writer | 本进程周期性地将WAL缓冲区中的WAL数据刷入持久存储中。 |
| statistics collector | 本进程负责收集统计信息，用于诸如pg\_stat\_activity，pg\_stat\_database等系统视图。 |
| logging collector (logger) | 本进程负责将错误消息写入日志文件。 |
| archiver | 本进程负责将日志归档。 |

**内存结构**

PostgreSQL的内存架构可以分为两部分：

本地内存区域 —— 由每个后端进程分配，供自己使用。

共享内存区域 —— 供PostgreSQL服务器的所有进程使用。



**本地内存区域**

​每个后端进程都会分配一块本地内存区域用于查询处理。该区域会分为几个子区域 —— 子区域的大小有的固定，有的可变。 表2.2列出了主要的子区域。

|  |  |
| --- | --- |
| 子区域 | 描述 |
| work\_mem | 执行器在执行ORDER BY和DISTINCT时使用该区域对元组做排序，以及存储归并连接和散列连接中的连接表。 |
| maintenance\_work\_mem | 某些类型的维护操作使用该区域（例如VACUUM，REINDEX）。 |
| temp\_buffers | 执行器使用此区域存储临时表。 |

**共享内存区域**

​PostgreSQL服务器启动时会分配共享内存区域。该区域分为几个固定大小的子区域。

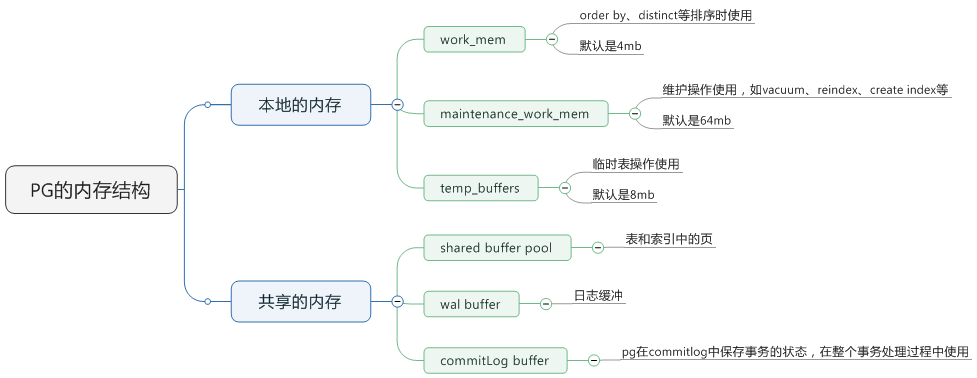
|  |  |
| --- | --- |
| 子区域 | 描述 |
| shared buffer pool | PostgreSQL将表和索引中的页面从持久存储加载至此，并直接操作它们。 |
| WAL buffer | 为确保服务故障不会导致任何数据丢失，PostgreSQL实现了WAL机制。 WAL数据（也称为XLOG记录）是PostgreSQL中的事务日志；WAL缓冲区是WAL数据在写入持久存储之前的缓冲区。 |
| commit log | **提交日志（Commit Log, CLOG）**为并发控制（CC）机制保存了所需的所有事务状态（例如进行中，已提交，已中止等）。 |

除了上面这些，PostgreSQL还分配了这几个区域：

用于访问控制机制的子区域（例如信号量，轻量级锁，共享和排他锁等）。

各种后台进程使用的子区域，例如checkpointer和autovacuum。

用于事务处理的子区域，例如保存点（save-point） 与 两阶段提交（2PC）。



**PostgreSQL常见概念**

cluster：集簇，同instance，

tablespace：表空间，对象存放的物理路径

schema：模式，同namespace

relation：关系，所有对象统称为relation，一般指表、索引

tuple：元组，同row

attribute：列，同column

oid：object\_id，对象标识符

ctid：tuple的物理位置

page：数据块，同block

buffer：内存中的数据块

wal：write ahead log，同redo log

dead tuple：删除或标记为不可见的行

vacuum：postgresql回收dead tuple，释放空间

extension：扩展模块，同plugin

case insensitive：postgresql大小写不敏感，除非加双引号

backend：postgresql为连接分配的后端进程

background：后台系统进程

TOAST：The OversizedAttribute Storage Technique，超尺寸属性存储技术，主要用于存储大字段

HOT：Heap-Only Tuples，堆内元组技术

FDW：foreign-data wrapper，外部数据包装器

role：同user