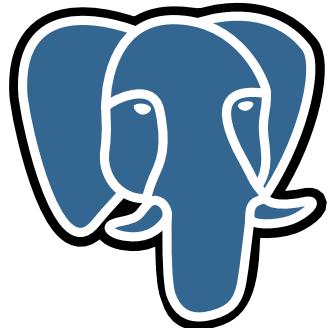




PostgreSQL 进阶之路

国人剖析 PostgreSQL 内核技术的原创之作

作者：小布



吾生也有涯，而知也无涯

自序

缘起

在 IT 行业浪迹了二十多年之后，我决定写人生的第一本书，算是对自己职业生涯的一个总结，此即阁下面前的这本《PostgreSQL 进阶之路》。或许是因为命运的安排，十多年前我来到美国后，没能成为这条街最靓的 Java 仔，却阴差阳错地进入到数据库管理员 (Database Administrator, DBA) 的行业，从此我职业生涯的大部分时间都是和 Oracle/PostgreSQL 等数据库软件打交道。本书的内容是对著名的开源数据库软件 PostgreSQL 相关技术的讲解，大约类似“某某某从入门到精通”的套路，但绝不会教你如何从删库到跑路。

二十世纪中后期至二十一世纪上半期，是 IT 技术革命大爆发的时代，此乃人类之幸事。但科技的快速发展也带来了某些副作用，其中之一就是劳动者需要更长的时间和精力来学习，才能具备从事某种专业领域工作的能力。君不见，学士、硕士、博士不断内卷，直逼壮士、猛士、乃至烈士。等到博士毕业后，才蓦然发现自己早已宿命悲白发，可怜依然单身狗。当代年轻人晚婚，不婚，少子化等社会问题越来越普遍，科技进步亦难辞其咎。在 IT 技术领域，随着技术复杂度的增加，进入其中的壁垒亦越来越高，后学者想入门的难度越来越大。譬如 Linux 内核的开发，对于初学者来说，其学习曲线异常陡峭，除非投入巨大的精力和时间成本，否则连老鸟的谈话都很难听懂。若无足够的利益吸引，想进入 IT 技术领域的年轻人会越来越少。老鸟终究会慢慢凋零，新鲜血液补充不足，是不少基础性 IT 技术普遍面临的问题。我写本书的目的就是希望能够在学习者进入 PostgreSQL 领域的过程中，为他们降低学习门槛做一些贡献。

或问：坊间相关 PostgreSQL 的书籍早已汗牛充栋，此书又有什么独特之处？答曰：我对目前所有的相关书籍都不甚满意，而不满是技术进步的根本动力。我把本书的目标定为中文世界里帮助学习者进入 PostgreSQL 领域最好的拐杖。我的 PostgreSQL 学习之旅充满了痛苦：很多初级读物流于肤浅，而深入的著作因为缺乏前导知识，又让初读之人不知所云。我不得不像一只勤劳的小蜜蜂，不断在互联网的世界里飞来飞去，四处艰难地采集我苦寻的“干货”。我要花费大量时间，辨别各种资料的谬误，不断做实验，分析源码，才能够有一点点的收获，我的学习笔记上寥寥数语心得的背后都是一把辛酸泪。我不想后学者再重复这个痛苦的过程，就产生了一种冲动：写一本书，遵循普通人的认知规律，循序渐进地讲解 PostgreSQL 的核心知识，让读者只要按顺序学习，就能以较少的痛苦达到一定程度的“登堂入室”。

写书有三种做法：编书，著书和编著。此三者的区别显而易见，所谓编书，就是把别人的内容汇编成册，自己只进行整理。所谓著书，就要把自己消化理解后的知识和经验写出来，把茶壶里的饺子倒出来。编著则是既“编”又“著”。显然，“著书”层次最高，要花费的精力和心血最多，是个良心活。曹氏曰：“满纸荒唐言，一把辛酸泪。都云作者痴，谁解其中味？”此语真乃著书的天花板！我亦想竭尽自己有限的才智“著”一本书，整齐百家诸语，成一家之言，然后藏之名山，以俟后世君子。至于能否达到此等境界，我只能说：尽吾志而不能至者，无悔矣！

本书遵循的写作原则

循序渐进：人类学习知识的过程肯定是循序渐进的：如果概念 B 是建立在概念 A 的基础之上，则必须要先搞懂概念 A，然后才能学习概念 B，这当然是一句废话。然而，很多书籍和技术高手往往忽视这个根本性的原则，导致读者或听众如鲠在喉，想吐槽却无处说理，吾亦深受其苦。所以“循序渐进”是本书遵循的最高原则，我对书中的章节安排和内容取舍颇费心力，让每个新概念的引入绝不突兀，都是建立在已有知识的基础之上。知我者谓我心忧，不知我者问我何求。若读者学习本书时的痛苦感大为减少，则不负作者的一片苦心。

用图说话：一图胜千言，此语在 IT 行业尤甚。面对庞大繁杂的源代码，用简明扼要的图来描述其关键思想，是理解技术内幕极其重要的途径，故本书使用了大量的图来展示 PostgreSQL 的关键设计思想。把源代码转化成图实乃最耗费心力之事，此项工作花费了我大量的心血和时间。书中的每张图均是我手工绘制，且尽可能简单，

易于读者理解和记忆，此乃本书最具特色之处。

内外兼修：我们学习商业闭源软件技术时，往往只能学习“外在”的功能性知识，却不知其内部的实现机理，此即老话说的“知其然而不知其所以然”。我从事 Oracle DBA 十几年了，若你问我对 Oracle 数据库技术掌握到什么程度，我只能默然良久。这是因为 Oracle 的源代码是商业机密，无论小子多么努力，总有一种隔靴挠痒之感。我们都有体会：如果感觉自己彻底理解了一个技术，就会带来很大的成就感和自信心，这是人性。此种成就感和自信心在开源软件身上才能够获得，所以本书在讲解 PostgreSQL 软件的外在功能时，会引导读者阅读其背后的关键源代码，内外兼修，通过源代码和外在功能的互相验证，达到“知其然且知其所以然”的境界，增加学习时的满足感，甚至某种幸福感。

注重细节：坊间的 IT 技术书籍，大体上分为两类，一类是针对初学者的读物，只介绍如何操作，却不谈背后的原理，读者很容易对琐碎的细节产生厌烦。第二类是高级读物，介绍技术内幕和原理，但常常只谈理论知识，却鲜有用具体的例子指导读者去理解抽象的理论如何一步步变成具体的技术实现。魔鬼隐藏在细节之中，没有细节的理论讲解往往让读者产生困惑，这是我对我很多“高级”书籍不满的主要原因。本书讲解理论时，尽可能用源代码中的关键细节引导读者从小处入手，让读者萌生感慨：哇，原来我也可以阅读和理解高深的源代码。

勤奋是普通人最可靠的资本，如果我们穷尽毕生钻研一门技术到精深的境界，衣食无忧，又无需操心过多烦人的人际关系，快然自足，不知老之将至，亦是一种幸福的人生。我旅美近二十年，深深地感到，美国科技强大的背后，有无数勤勤恳恳的“匠人”在默默地支撑。PostgreSQL 背后的灵魂人物之一 Tom Lane 博士，一生淡泊名利，年近古稀却至今依然奋斗在编程第一线。我曾有幸和大师同住一个城市，却无缘拜会，甚为遗憾。青青子衿，悠悠我心，但为君故，沉吟至今。

经过 30 多年不懈的研发，目前 PostgreSQL 的技术发展已进入了快车道，全球的用户越来越多，就业市场上也正在涌现越来越多的相关高薪工作岗位，投身该领域必定能给你带来新的发展机遇，只要你足够勤奋。若卿不负韶华，君尔妾亦然，自本书始。

致谢

在本书的写作过程中，我不断在互联网上汲取诸位前辈对 PostgreSQL 技术内幕的探索成果，心中对这些事了拂衣去，深藏身与名的侠士们充满敬佩之情。在此尤其要特别感谢日本的铃木启修君，他像一位孤独的忍者，蹲踞在探索 PostgreSQL 技术内幕的道路上，让默默追随他背影的吾辈受益良多。技术高手常有，乐于搭梯助后学者登堂入室的导师不常有。铃木君对 PostgreSQL 的源代码抽丝剥茧，把隐藏其中的技术内幕一一展示出来，造福了全世界。吾尝闻长者之遗风，亦思后世之来者，遂成此作，致敬先生。

虽我已尽吾所能，但因才疏学浅，书中错误在所难免，敬请读者斧正。吾国之复兴，吾民之富足，皆仰赖基础性原创性的科技创造。希望本书能和铃木君的作品一样，激励出更多踏踏实实的原创佳作。虽世殊事异，所以兴怀，其致一也。嗟乎，希后之览者，亦有感于斯文！

癸卯三月于洛基山脚之西山居

目录

第一章 体系架构概述	1
1.1 创建实验环境	1
1.1.1 从源代码开始安装 PostgreSQL	2
1.1.1.1 环境检查及源码包下载	2
1.1.1.2 源代码的编译和软件安装	2
1.1.2 初次体验 PostgreSQL 的基本使用	4
1.1.2.1 设置环境变量	4
1.1.2.2 数据库的创建和启停	5
1.1.2.3 数据库的基本使用	6
1.2 体系架构概览	8
1.2.1 PostgreSQL 的进程	8
1.2.1.1 程序, 进程和线程	8
1.2.1.2 Postgres 的服务器进程	10
1.2.2 PostgreSQL 的内存结构	12
1.2.3 数据库集群	13
1.2.3.1 数据库集群目录的布局	14
1.2.3.2 同时运行多个数据库集群	15
1.2.3.3 表空间的基本概念	16
1.2.4 辅助文件	18
1.2.4.1 参数文件	19
1.2.4.2 控制文件	20
1.2.4.3 锁文件	21
1.2.4.4 主进程参数文件	21
第二章 数据文件	23
2.1 PostgreSQL 源代码的基础知识	23
2.1.1 如何搜索源代码	23
2.1.2 基本的数据类型	24
2.1.3 内存对齐	24
2.2 数据文件	26
2.2.1 数据文件的基本结构	26
2.2.2 一个简单的实验	27
2.2.3 数据块的结构	27
2.2.3.1 页头的结构分析	28
2.2.3.2 数据指针的结构分析	30
2.2.3.3 记录的结构分析	31
2.2.3.4 fsm 和 vm 文件	32
2.2.4 分析数据块和内存页的工具	33
2.2.4.1 pg_buffercache 的使用	34
2.2.4.2 pageinspect 的使用	36
2.3 性能测试工具 pgbench 的使用	38

2.3.1 配置远程连接	38
2.3.2 pgbench 的基本使用	39
第三章 理解 WAL	42
3.1 WAL 的基础知识	42
3.1.1 WAL 背后的思想	42
3.1.2 WAL 概念的含义	43
3.1.3 日志顺序号	44
3.1.4 WAL 文件	45
3.1.4.1 WAL 文件的命名规则	46
3.1.4.2 WAL 文件的内部结构	47
3.1.5 分析 WAL 记录的工具	50
3.1.5.1 pg_waldump	51
3.1.5.2 pg_walinspect	56
3.2 检查点	57
3.2.1 检查点的执行过程	58
3.2.2 检查点的 WAL 记录	62
3.2.3 检查点的执行时机	64
3.2.4 全页写	65
3.2.5 PostgreSQL 的写磁盘操作	69
3.2.6 监控 WAL 记录和检查点	71
3.3 崩溃恢复	74
3.3.1 WAL 记录的回放	74
3.3.2 崩溃恢复的流程	78
3.3.2.1 数据库实例的关闭模式	79
3.3.2.2 恢复进程的工作内容	80
3.4 WAL 文件的管理	84
第四章 备份和恢复	86
4.1 物理备份	86
4.1.1 归档模式	87
4.1.1.1 设置归档模式的实验	88
4.1.1.2 监控归档的系统视图	90
4.1.1.3 归档命令	91
4.1.2 使用底层函数完成物理备份	93
4.1.3 使用 pg_basebackup 进行备份	97
4.1.3.1 两种不同的网络协议	97
4.1.3.2 pg_basebackup 的基本使用	99
4.1.3.3 备份有效性的验证	102
4.2 数据库的恢复	105
4.2.1 数据库恢复的实验	105
4.2.2 恢复目标	108
4.2.3 时间线	112
4.3 逻辑备份和恢复	114
4.3.1 pg_dump 的基本使用	114

4.3.2 pg_restore 的基本使用	115
第五章 物理复制	116
5.1 快速搭建流复制	118
5.1.1 在主库机器上的配置	118
5.1.2 在备库机器上的配置	118
5.1.3 验证流复制是否工作	120
5.2 主库和备库的通讯过程	121
5.2.1 复制槽	123
5.2.2 复制协议	126
5.3 流复制的监控	128
5.4 主备库之间的切换	130
5.4.1 正常切换 (switch-over)	131
5.4.2 灾难切换 (fail-over)	134
5.5 从备库上执行备份	138
第六章 堆表和 B 树索引	139
6.1 堆表	139
6.1.1 TOAST	139
6.1.2 堆表的空闲空间管理	143
6.1.3 分区表	148
6.2 B 树索引	150
6.2.1 索引的基本概念	150
6.2.2 B 树索引	152
6.2.3 集簇 (Cluster)	163
第七章 多版本并发控制	164
7.1 事务	164
7.1.1 事务并发带来的结果	164
7.1.1.1 提交读 (Read Committed)	165
7.1.1.2 不可重复读 (Non-repeatable Read)	166
7.1.1.3 幻读 (Phantom read)	166
7.1.1.4 修改丢失 (Lost Update)	167
7.1.1.5 死锁 (Dead lock)	168
7.1.2 事务的隔离级别	169
7.2 多版本并发控制	170
7.2.1 事务号	170
7.2.1.1 记录的结构	173
7.2.2 事务的冻结	177
7.2.2.1 事务的年龄	177
7.2.3 提交日志 (Commit Log)	178
7.2.4 事务快照 (Transaction Snapshot)	179
7.2.5 可见性 (Visibility) 规则	182
7.2.6 pg_resetwal 的使用	183
7.2.7 HOT	184
7.2.8 数据页的修剪	188

7.2.9 事务的冻结	189
第八章 表和索引的清理 (VACUUM)	192
8.1 一个 VACUUM 的实验	192
8.2 VM 文件	196
8.3 CLog 的清理	199
8.4 统计信息的更新	199
8.5 HOT	200
8.6 VACUUM 的过程	204
8.6.1 第一阶段 - 扫描堆表	206
8.6.2 第二阶段 - 对索引进行 VACUUM	207
8.6.3 第三阶段 - 堆表的 VACUUM	208
8.6.4 第四阶段 - 索引的清理	208
8.6.5 第五阶段 - 堆表尾部数据块的处理	208
8.7 自动 VACUUM	208
8.8 VACUUM 的监控	209
8.9 对表的全清理 (VACUUM FULL)	209
第九章 数据库的逻辑复制	211
9.1 快速搭建逻辑复制	212
9.2 逻辑复制的体系结构	214
9.3 逻辑复制的具体内容	216
9.3.1 相关的系统视图	216
9.3.2 相关的后台进程	218
9.3.3 初始数据的同步	219
9.3.4 逻辑复制槽	221
9.3.5 冲突的处理	221
9.4 从备库进行逻辑复制	224
第十章 内存池管理和常用数据结构	225
10.1 内存池子系统	225
10.1.1 内存池的整体结构	226
10.1.2 内存池的相关数据结构	228
10.1.2.1 内存上下文 MemoryContext	228
10.1.2.2 分配集上下文 AllocSetContext	231
10.1.2.3 内存块 AllocBlock	231
10.1.2.4 内存片 MemoryChunk	232
10.1.2.5 空闲内存片数组 freelist	233
10.1.2.6 内存池操作的函数指针	234
10.1.3 内存池的操作函数	236
10.1.3.1 内存池创建函数之分析	236
10.1.3.2 内存池重置函数之分析	239
10.1.3.3 内存池销毁函数之分析	240
10.1.3.4 内存片分配函数之分析	241
10.1.3.5 内存片释放函数之分析	244
10.1.3.6 分配内存片接口函数之分析	245

10.2 动态哈希算法	246
10.2.1 基本原理	246
10.2.1.1 通用哈希算法的基本原理	246
10.2.1.2 动态哈希的基本原理	247
10.2.2 PostgreSQL 的内存哈希表	251
10.2.2.1 相关的数据结构	251
10.2.2.2 相关的函数代码分析	258
第十一章 锁和共享内存	267
11.1 进程间通讯的基本知识	267
11.1.1 系统调用 fork()	267
11.1.2 共享内存	268
11.1.2.1 私有内存和共享内存	270
11.1.3 Latch	272
11.1.4 信号	272
11.1.5 信号量	273
11.2 锁	275
11.2.1 自旋锁	275
11.2.1.1 X86/X64 中的 TAS 指令	276
11.2.1.2 PostgreSQL 中的自旋锁	277
11.2.2 轻量级锁	279
11.3 PostgreSQL 共享内存的结构	283
11.3.1 共享内存的分配机制	283
11.3.2 主索引	284
11.3.2.1 子进程和主进程的通讯	287
11.3.3 共享池	288
11.3.3.1 分配策略	294
11.3.4 磁盘管理	303
11.3.5 WAL Buffer 池	311
11.4 PostgreSQL 进程间通讯	312
附录 A 使用 GDB 调试 PostgreSQL 源代码快速入门	313
附录 B 作者小传	314

第一章 体系架构概述

PostgreSQL，简称 PG，是著名的开源数据库软件，具有良好的设计和高质量的源码。全球范围内不少的公司都以 PostgreSQL 为内核，开发出各具特色的数据库软件产品，譬如中国的华为公司的 openGauss，阿里巴巴的 PolarDB PostgreSQL 版，美国的 EDB 公司的 Advancede Server，俄罗斯的 PostgreSQL Professional 公司的 PostgreSQL Pro 数据库等等。也有不少大学的计算机相关专业以 PostgreSQL 为技术原型，研究和传授数据库相关的理论和具体实现。

学习任何一个新东西，第一件事情就是要对它有一个整体印象，所以第一章自然要介绍 PostgreSQL 的体系结构。但在此之前，当务之急是手头上有个可以做各种实验的学习环境，所以我们在第一节要先了解如何搭建 PostgreSQL 的学习实验环境。学习开源软件的技术，如果不去阅读源代码，以求理解其背后的技术秘密，就辜负了“开源”两个字，所以本章只讲解如何从源代码开始安装 PostgreSQL 软件，且在后续的学习中坚持既学习软件的外在功能，又探究其背后源代码的具体实现，努力让读者做到“知其然且知其所以然”。

本书的全部实验都是在 Linux 环境下进行的，因此要求读者具备一定的 Linux 使用经验，且为了能够理解 PostgreSQL 的源代码，读者还需要具备初步的 C 语言编程的能力。如果你不具备这两个领域的基本能力，也不要担心，因为本书需要的 Linux 和 C 语言的知识非常少：对于 Linux 的操作，只要求掌握 ps、ls、pwd 和 vi 等常用的命令。对于 C 语言部分，只要求知道十六进制和二进制、十进制之间的转换，理解结构体 (struct) 和指针 (pointer) 的基本使用等。互联网上相关内容的文档和视频等学习资料非常丰富，你可以快速学习。只需恶补一两个星期左右，你所掌握的 Linux 基本操作和 C 语言编程的知识就足够应付本书的学习。

在本书开始写作时，PostgreSQL 的最新版本是 16，所以本书的内容以 PostgreSQL 16 作为基本的学习版本。可能你在阅读本书时 PostgreSQL 又发布了更新的版本，但本书绝大部分的内容和 PostgreSQL 的版本关系不大，适用于 PostgreSQL 10 以后的所有版本。

1.1 创建实验环境

在讲解如何从源码开始安装 PostgreSQL 的内容之前，我假定你有一台可以使用的 Linux 服务器，且使用的用户是 postgres。我使用的 Linux 机器是一台运行在 VMWare Player 虚拟机中的 Debian Linux 12，我的实验环境整体如图 1.1 所示。

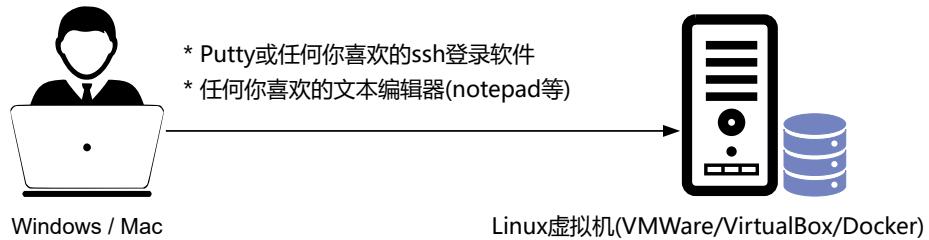


图 1.1：实验环境整体架构图

我建议你使用 VMWare/Virtual Box/Docker 等虚拟机软件来运行 Linux。因为绝大部分的读者都使用 Windows 操作系统，你也可以使用 Windows 自带的 WSL(Windows Subsystem for Linux) 作为运行 PostgreSQL 的基本环境。关于如何安装 Linux 环境，互联网上这方面的资料非常多，在此我就不重复了。本书所有的实验基本上都是在一台或者若干台 Debian Linux 12 上完成的，偶尔我也会使用 WSL 的环境做实验。不过 PostgreSQL 的实验和 Debian/WSL 基本上没有关系，你可以选择任何自己喜欢的 Linux 发行版本，都能够顺利地完成本书的各种实验。

1.1.1 从源代码开始安装 PostgreSQL

1.1.1.1 环境检查及源码包下载

搭建实验环境，首先要做两件事情：一个是下载源码包，另外一个是检查操作系统的环境。我们访问 PostgreSQL 的官方网站 (www.postgresql.org)，很容易找到下载 (download) 的入口，注意要选择下载源码包，而不是已经编译好的二进制软件包。例如我下载了文件 `postgresql-16.0.tar.bz2` 或者 `postgresql-16.0.tar.gz`，它们都是 PostgreSQL 16 的源码包，只不过是压缩的格式不同而已。我们需要使用类似 WinSCP 的工具把源码包上传到 Linux 实验服务器上，然后在 `postgres` 用户拥有写权限的某个目录下，例如`/home/postgres`，把源码包解开。操作的细节请参考下面的实验：

```
$ id /* 任何用户都可以。我推荐使用postgres用户，它所在的组也是postgres */
uid=1001(postgres) gid=1001(postgres) groups=1001(postgres)
$ pwd
/home/postgres
$ ls -l
total 23956
-rw-r--r-- 1 root root 24528207 Oct  1 08:45 postgresql-16.0.tar.bz2
/* 解压缩源码包。如果是*.tar.gz格式，就使用tar zxvf的选项 */
$ tar jxvf postgresql-16.0.tar.bz2
$ ls -l /* 我们会发现解压缩后产生的目录postgresql-16.0 */
total 23960
drwxrwxr-x 6 postgres postgres 4096 Sep 11 14:29 postgresql-16.0 /* <-- 解压缩后产生的文件夹 */
-rw-r--r-- 1 root      root   24528207 Oct  1 08:45 postgresql-16.0.tar.bz2
```

注意：本书使用 C 语言风格的多行注释/* ... */来解释实验或者源代码中每一步的含义和要点。这些注释仅仅是为了方便读者理解实验过程的要点或者源代码的含义，并不是输入的命令或者输出结果的一部分。

有了源码包之后，我们要检查一下操作系统的配置。因为需要对源代码进行编译，常用的开发编译工具 `gcc` 和 `gmake` 必不可少，所以你必须确保 `gcc` 和 `gmake` 在你的 Linux 服务器上已经安装好了，检测的方法是执行下面的命令：

```
$ gcc --version
gcc (Debian 12.2.0-14) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
.....
$ gmake --version
GNU Make 4.3
.....
$ gdb --version
GNU gdb (Debian 13.1-3) 13.1
.....
```

如果出现类似“command not found”的错误，就说明 `gcc` 或者 `gmake` 没有安装好，你需要先解决这个问题，再进行下面的安装步骤。在我的研究过程中，经常使用调试工具 `gdb`。这个是可选项，如果你不调试程序，可以不安装它。本书的附录 A 介绍了使用 `gdb` 调试 PostgreSQL 源代码的快速入门指南。

1.1.1.2 源代码的编译和软件安装

在源代码目录里有很多文件和子目录，其中两个子目录需要注意：一个是 `src` 子目录，这里包含了 PostgreSQL 的所有核心源代码。另一个是 `contrib` 子目录，它里面包含了一些外围工具包的源代码。假设我们准备把

PostgreSQL 软件安装到/opt/software 目录下，把数据库的目录放在/opt/data 中，你需要以 root 用户，使用 chown 命令让 postgres 用户拥有这两个目录，请参考下面的实验：

```
/* 以root用户在/opt目录下创建software和data两个子目录 */
# cd /opt
# mkdir software data
/* 把这两个子目录划归给postgres用户所有 */
# chown -R postgres:postgres software data
# ls -l /opt /* 可以看到这两个目录已经归postgres用户所有了 */
total 8
drwxr-xr-x 3 postgres postgres 4096 Oct  1 09:08 data
drwxr-xr-x 3 postgres postgres 4096 Oct  1 08:56 software
```

以上准备工作妥当之后，在 PostgreSQL 的源代码目录下按顺序运行如下三条命令就可以进行源代码编译和安装了：

```
./configure --prefix=/opt/software/pg16 --enable-debug --enable-cassert CFLAGS="-O0" --without-icu
make world
make install-world
```

命令 configure 是配置命令，它会检测你的 Linux 环境，然后产生一些必要的补充文件，并最终产生 Makefile，供下一步的 make 命令使用。我使用了--enable-debug --enable-cassert 这两个选项是为了在后面方便使用 gdb 进行调试。CFLAGS="-O0" 则表示禁止在编译过程中对代码进行优化，进一步提高了调试过程中源代码的可读性。注意：这三个选项仅仅是为了方便我们调试使用，如果你打算编译准备部署在生产环境中的 PostgreSQL 软件，务必要把这三个选项去掉。从 16 版本开始，PostgreSQL 开始支持 icu 库，这是一个支持国际化编码的库，往往我们的系统上没有安装。你可以使用"--without-icu" 把它跳过去，不会影响基本的使用。如果你的 Linux 系统上已经安装好了 icu 库，你可以去掉这个选项。

命令 make 会把 src 下面的源码进行编译，而选项 world 会 contrib 目录里面的工具包也一并编译，这些工具包对后续的分析工作非常有帮助，所以我建议大家加上 world 的选项。命令 make install-world 则是把编译好的软件拷贝到在第一步 configure 命令中指定的/opt/software/pg16 目录。

这里一个常见的问题是 configure 命令可能会抱怨 readline 和 zlib 的开发库找不到。开发库 readline 是供 PostgreSQL 的客户端 psql 使用的，它可以在 psql 中使用上下箭头把以前输入的命令重新调出来，避免重复输入，而压缩库 zlib 则是 PostgreSQL 在压缩数据库备份时使用的。请仔细阅读 configure 命令的输出，如果你发现 configure 抱怨缺乏 readline 和 zlib 开发库的信息，请在互联网上搜索解决方案，自行解决。注意：需要的是 readline 和 zlib 的开发库，不是运行库。如果你实在解决不了这些问题，configure 也提供了--without-readline 和--without-zlib 两个选项跳过这些开发包，但我建议你最好还是使用这些库。

第二步的 make 命令会编译所有的源代码，需要的时间可能有点长，请稍微耐心等待一会儿。第三步 make install-word 命令结束后，你可以执行如下命令，检查一下软件是否安装好了。

```
$ ls -l /opt/software
total 4
drwxrwxr-x 6 postgres postgres 4096 Oct  1 08:56 pg16
$ ls -l /opt/software/pg16/
total 16
drwxrwxr-x 2 postgres postgres 4096 Oct  1 08:56 bin
drwxrwxr-x 4 postgres postgres 4096 Oct  1 08:56 include
drwxrwxr-x 4 postgres postgres 4096 Oct  1 08:56 lib
drwxrwxr-x 5 postgres postgres 4096 Oct  1 08:56 share
/* 所有的PG软件都放在bin目录下 */
$ ls -l /opt/software/pg16/bin/postgres
```

```
-rwxr-xr-x 1 postgres postgres 32315440 Oct  1 08:56 /opt/software/pg16/bin/postgres
```

如果出现类似上面的结果，那么我要恭喜你：你已经把 PostgreSQL 软件安装成功了！下面就可以进行一次“猪八戒吃人参果”式的快速体验，了解一下传说中的 PostgreSQL。

注意：从 16 版本开始，PostgreSQL 的核心开发团队开始引入一个新的编译系统 meson。不过本书写作时，官方文档中关于源码编译的方法依然支持 config/make/make install 这三件套。可能你开始学习本书时，已经彻底转向了 meson 的编译方法。具体请参考官方文档。官方文档写的很清楚，只要仔细阅读，不难学会使用 meson 进行源码编译的方法。

1.1.2 初次体验 PostgreSQL 的基本使用

面对一个新的软件，我们要学会三件事情：如何启动它，如何关闭它，在软件启动之后如何做一些简单的操作来理解它的基本功能。我们下面的体验就是学习这三个任务是如何实现的。

1.1.2.1 设置环境变量

在 Linux 环境下的软件往往需要设置一些环境变量 (environment variable) 来配合软件的运行，如 Oracle 数据库软件需要设置著名的 ORACLE_HOME 和 ORACLE_SID 等环境变量。PostgreSQL 常用的环境变量有两个：PGHOME 和 PGDATA，它们的含义如下：

- PGHOME 指向 PostgreSQL 的软件安装目录，在我们这里是 /opt/software/pg16。由于所有的 PostgreSQL 的程序都在 \$PGHOME/bin 目录下，所以你可以把 PGHOME/bin 加入到 PATH 环境变量，以后就可以在任何地方使用 PostgreSQL 的各种工具，无须每次运行时都要指定该程序所在的目录，避免了繁琐。其次，最好把 PGHOME/lib 这个目录加到环境变量 LD_LIBRARY_PATH 这个系统环境变量中，方便 Linux 查找 PostgreSQL 所需要的动态库文件。
- PGDATA 指向 PostgreSQL 的数据库目录。PostgreSQL 的很多工具都存在一个基本逻辑：如果命令行中输入了 -D 的选项（注意是大 D，不是小 d），则使用 -D 后面的路径信息作为数据库的目录。如果没有 -D 选项，则读取 PGDATA 里面的路径信息作为数据库的目录。所以我强烈建议大家设置 PGDATA 指向你的数据库目录，避免每次都要使用 -D 参数。

本书后续的实验均假设这两个环境变量都已经设置好了，你可以参考下面的例子设置这两个环境变量。

```
/* 使用 vi 或者其它文本编辑器来创建 set.env 文本文件，其内容如下 */
$ cat set.env
PGHOME=/opt/software/pg16
PGDATA=/opt/data/pgdata1
PATH=$PGHOME/bin:$PATH
LD_LIBRARY_PATH=$PGHOME/lib:$LD_LIBRARY_PATH
export PGHOME PGDATA PATH LD_LIBRARY_PATH

$ chmod +x set.env      /* 给这个文件赋予可以执行的权限 */
$ . ./set.env          /* 这个命令是设置环境变量。注意两个点中间有个空格 */
$ env | grep PG        /* 检查一下 PostgreSQL 的这两个环境变量是否生效了 */
PGHOME=/opt/software/pg16
PGDATA=/opt/data/pgdata1
```

当然你也可以把 set.env 里面的内容放在 postgres 用户的 .bash_profile 或者 .profile 中，以后你每次登录 Linux 服务器，这些设置会自动生效，更加方便。

1.1.2.2 数据库的创建和启停

PostgreSQL 的软件安装好之后，下面的工作就是要创建一个 PostgreSQL 数据库，创建数据库的命令是 initdb。PostgreSQL 软件中的程序通常都会带一个--help 的选项（注意是两个连起来的减号），来显示本软件的功能和各种输入参数的含义，你可以试试 initdb --help，观察它的输出。下面的实验演示了如何使用 initdb 创建数据库。

```
$ env | grep PG      /* 检查一下环境变量是否设置好了 */
PGHOME=/opt/software/pg16
PGDATA=/opt/data/pgdata1
$ ls -l /opt/data  /* /opt/data/pgdata1目录可以不存在，initdb会自动创建它 */
total 0
$ initdb -D /opt/data/pgdata1 /* 因为环境变量PGDATA已经设置好了，所以-D选项是多余的 */
.....
/* 此处删去了XXXX个字 */
.....
$ ls -l /opt/data  /* 发现pgdata1目录已经被创建了 */
total 4
drwx----- 19 postgres postgres 4096 Oct  1 09:08 pgdata1
$ ls -l /opt/data/pgdata1 /* 查看一下新鲜出炉的数据库的基本目录布局 */
total 120
-rw----- 1 postgres postgres     3 Oct  1 09:08 PG_VERSION
drwx----- 5 postgres postgres  4096 Oct  1 09:08 base
drwx----- 2 postgres postgres  4096 Oct  1 09:08 global
drwx----- 2 postgres postgres  4096 Oct  1 09:08 pg_commit_ts
.....
/* 此处删去了XXXX个字 */
.....
drwx----- 3 postgres postgres  4096 Oct  1 09:08 pg_wal
drwx----- 2 postgres postgres  4096 Oct  1 09:08 pg_xact
-rw----- 1 postgres postgres    88 Oct  1 09:08 postgresql.auto.conf
-rw----- 1 postgres postgres 29697 Oct  1 09:08 postgresql.conf
```

数据库创建工具 initdb 的基本逻辑是：指定的数据库目录无需存在，initdb 会自动创建它。如果该目录已经存在，则 initdb 会检查该目录是否为空，如果为空，就继续在该目录下创建数据库。如果该目录不为空，为了避免覆盖这个目录下的文件，initdb 就会退出。当数据库创建完毕后，我们可以使用 pg_ctl start 命令来启动数据库，使用 pg_ctl stop 来关闭数据库，其具体过程请参考下面的实验操作。

```
$ pg_ctl status  /* 这条命令是检查数据库的状态，结果是没有数据库在运行 */
pg_ctl: no server running
/* 下面这条命令看看内存里有没有相关的进程，结果没有 */
$ ps -ef | grep postgres | grep -v grep
/* 下面这条命令启动数据库，日志文件为当前目录的logfile */
$ pg_ctl start -l logfile
waiting for server to start.... done
server started
/* 再次检查数据库的状态，数据库已经运行了，主进程的进程号是11194 */
$ pg_ctl status
pg_ctl: server is running (PID: 11194)
/opt/software/pg16/bin/postgres
$ ps -ef | grep postgres          /* 再次使用ps大杀器，发现内存中有了几个PG的进程 */
```

```

postgres  11194      378  0 09:13 ?          00:00:00 /opt/software/pg16/bin/postgres
postgres  11195      11194  0 09:13 ?          00:00:00 postgres: checkpointer
postgres  11196      11194  0 09:13 ?          00:00:00 postgres: background writer
postgres  11198      11194  0 09:13 ?          00:00:00 postgres: walwriter
postgres  11199      11194  0 09:13 ?          00:00:00 postgres: autovacuum launcher
postgres  11200      11194  0 09:13 ?          00:00:00 postgres: logical replication launcher
$ pg_ctl stop -D /opt/data/pgdata1 /* 使用这条命令来关闭数据库，注意：-D选项是多余的 */
waiting for server to shut down.... done
server stopped
$ pg_ctl status /* 再次检查数据库是否在运行 */
pg_ctl: no server running
/* 不放心的话，用ps再次检查一下，结果上面的进程消失了 */
$ ps -ef | grep postgres | grep -v grep
$
```

1.1.2.3 数据库的基本使用

在学习完如何启动和关闭数据库之后，我们要学习 PostgreSQL 的基本使用。首先再次数据库启动，然后使用客户端 psql 来登录数据库。下面的实验创建一个数据库和一张表，并且往表里插入两条记录，然后查询这个表，这些都是数据库最基本的功能。

```

$ psql
psql (16.0)
Type "help" for help.
/* 创建一个数据库，名字叫oracle */
postgres=# CREATE DATABASE oracle;
CREATE DATABASE
postgres=# \c oracle      /* 连接到oracle数据库 */
You are now connected to database "oracle" as user "postgres".
/* 创建一个简单的state表，并插入两条记录 */
oracle=# CREATE TABLE state(id INT, name CHAR(2));
CREATE TABLE
oracle=# INSERT INTO state VALUES(0, 'TX');
INSERT 0 1
oracle=# INSERT INTO state VALUES(1, 'MA');
INSERT 0 1
oracle=# SELECT * FROM state; /* 查询一下state表里面的内容 */
 id | name
----+---
  0 | TX
  1 | MA
(2 rows)
oracle=# \q                  /* \q是退出psql的命令 */
$
```

注意：如果你的 Linux 用户不是 postgres，当你使用 psql 时，可能会出错。这是因为 psql 缺省会连接到和你使用的 Linux 用户同名的数据库中。Linux 的 postgres 用户会在 psql 登录后自动连接到 postgres 数据库，而这个数据库肯定存在的。假设你使用的 Linux 用户叫 oracle，你可以使用 psql -d postgres 连接到缺省的 postgres 数据库后，再执行 CREATE DATABASE oracle 的命令创建数据库。下次你就可以直接使用 psql，不带任何参数，它

就连接到了 oracle 数据库。客户端软件 psql 类似 Oracle 数据库中著名的 sqlplus，是我们把玩 PostgreSQL 时需要天天打交道的好帮手，建议你使用 `psql --help` 来稍微了解一下它的各种参数的含义。

至此，PostgreSQL 实验环境的搭建工作已经完成，我们也初步体验了启停数据库和数据库的基本使用。下一节会对 PostgreSQL 的体系结构进行一个基本的介绍。

1.2 体系架构概览

本节将对 PostgreSQL 的体系结构做一个鸟瞰式的整体概述，使得读者心中有一副完整的地图，方便后面的学习。在深入学习数据库的知识之前，我们必须搞清楚一个基本问题：数据库到底能给我们带来什么价值？如果把数据库当做一个黑盒子，我们会发现数据库提供了两个基本功能：其一是要把用户存入数据库中的数据可靠地保存起来，就类似我们去银行存款，银行会可靠地保存我们的资金；其二是能够快速返回用户想查询的数据。第一个功能的核心是可靠，第二个功能的核心是快速。可靠性的保障说到底就是写入到磁盘的数据掉电后不丢失。所以只要正确地把数据写入到掉电不丢失的磁盘上，数据库就完成了可靠保存用户数据的承诺。为了可靠且快速地把数据写入到磁盘，数据库引入了一个核心概念：提前写日志 WAL，这是第三章讨论的主题。数据库第二个功能的核心是快速地返回用户的查询结果。为此，数据库对外提供了 SQL 语言的接口，并且提供 SQL 的解释执行器执行用户的 SQL 查询命令。如何让 SQL 的解释执行器更快地工作，也是数据库的核心研究课题。数据库所有的功能设计，都是围绕这两个基本功能而展开的，这是我们在学习数据库技术细节时不要忘记的基本问题。因为数据库涉及的知识非常广泛，所以本书的内容只围绕第一个基本问题展开，不涉及如何让 SQL 查询运行的更快。

理解了数据库的基本功能以后，下面我们就来学习 PostgreSQL 的体系架构。简而言之，任何数据库，或者任何软件，都离不开三大件：进程，内存和文件，PostgreSQL 也不例外。图 1.2 展示了 PostgreSQL 的整个体系架构，下面我们结合这张体系架构图依次学习 PostgreSQL 的进程，内存和文件这三大组件。

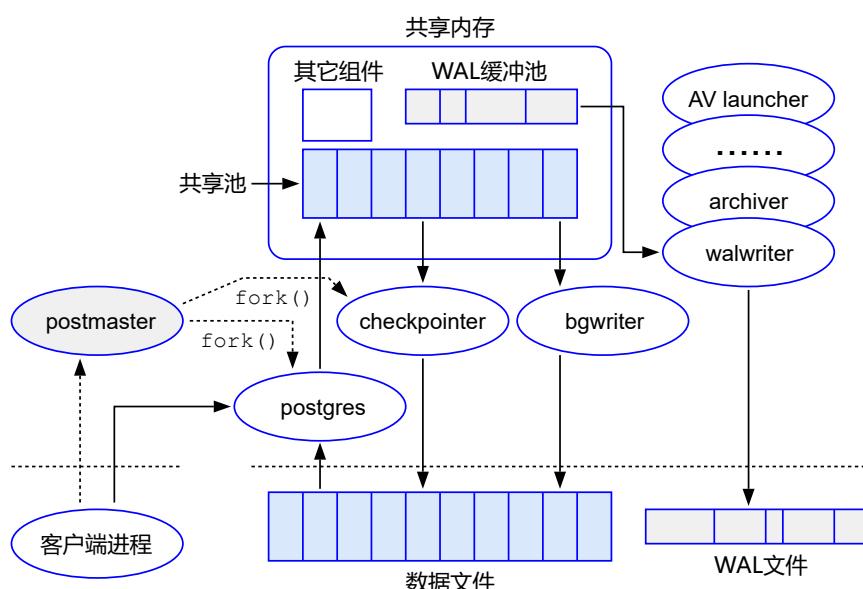


图 1.2: PostgreSQL 体系架构图

注意：本书的示例图中一般用椭圆形表示进程，矩形表示内存。

1.2.1 PostgreSQL 的进程

1.2.1.1 程序，进程和线程

为了方便基础不牢的读者，本节简单扼要地介绍一下程序，进程和线程这三个重要概念和相互的关系。众所周知，我们可以使用各种编程语言开发软件，如 C, Java, Go, Rust 和 Python 等等。软件开发的最终成果是一些可以被执行的文件，它们运行起来就会提供开发时设定的功能。这种可以被执行的文件，叫做“程序”(program)。用 C/Java/Go/Rust 等编译型语言编写好的源程序，需要被编译器编译成二进制形态的可执行文件。下面的实验演示了如何编译一个非常简单的 C 语言源代码文件并运行编译后的程序。

```
/* 使用vi或者其它编辑器编写源代码文件hello.c, 其内容如下: */
$ cat hello.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char* argv[])
{
    printf("Hello, World!\n");
    sleep(60); /* sleep for 60 seconds */
    return 0;
}
/* 我们用gcc进行编译, 产生了一个可执行文件hello */
$ gcc -Wall hello.c -o hello
$ ./hello /* 运行可执行文件hello, 它会创建一个进程在内存中停留60秒 */
Hello, World!
$
```

上述程序的逻辑非常简单，就是打印一个字符串，然后休眠 60 秒后退出。我们在它休眠期间快速打开另外一个 putty 登录到服务器上，执行 ps 命令进行查看：

```
$ ps -ef | grep hello | grep -v grep
postgres      9630      8868  0 01:33 pts/0    00:00:00 ./hello
```

结果我们看到了一个名字叫 hello 的进程 (process)。回顾一下上面的实验，我们发现整个过程分为三步：第一步人类准备了一个由纯字符组成的源代码文件 hello.c。第二步是编译器 gcc 对这个源代码文件进行编译，产生了二进制文件 hello，这就是一个程序。在第一节安装 PostgreSQL 软件的实验中所使用 make world 命令的本质就是编译，它调用 gcc 编译器对 PostgreSQL 的 C 语言源代码文件进行编译，最终产生了 \$PGHOME/bin/postgres 和其它的可执行文件，文件 postgres 就是 PostgreSQL 软件中最核心的程序。上述实验的第三步是运行我们编译好的可执行文件 hello，然后我们使用 ps 命令去查看进程。

程序只是躺在磁盘上的某个文件，只有用户运行它后，它才能够提供应有的服务，已经运行起来的程序被称为“进程”。在上述实验中我们用./hello 的命令运行当前目录下的程序 hello，就会产生一个进程。命令 ps 可以查看当前进程的一些基本信息，由上面 ps 命令的输出可知，进程 hello 的进程号是 9630，它的父进程号是 8868。所谓进程号，就是操作系统给这个进程分配的唯一性编号。在计算机加电运行后会创建第一个进程，其后的任何一个进程都是别的进程创建出来的，所以每一个进程还会有一个父进程的编号。再例如，我们每打开一个 putty 窗口，Windows 操作系统就创建了一个进程，打开 n 个 Putty 窗口，就是启动了 n 个进程。这些 Putty 的进程可以在 Windows 中的任务管理器 (Task Manager) 中被查看到。

操作系统会为进程分配很多资源，包括该进程独享的内存空间，这导致不同进程之间，鸡犬之声相闻，老死不相往来，彼此都过着独立的日子。为了进一步提高进程的执行效率，人类又发明了“线程” (thread) 的概念。简而言之，线程就是在一一个进程内部的多个可以并发执行的单元。线程的执行效率也比进程更高，但线程技术有利有弊。在一个进程内部的多个线程共享本进程的内存，所以线程之间可能会导致冲突，从而导致多线程软件开发比较困难。如果控制不好，采用多线程技术会导致软件的稳定性存在一些问题，也不容易排查。关于进程线程的概念和关系的进一步探讨，涉及到操作系统等深入的知识，我们就止步在此。这两个概念是非常重要的核心概念，对它们的理解越深入越好。建议读者在网上搜索相关资料进一步研究。

数据库软件是非常重要的基础性软件，它需要经年累月地不间断运行，所以人类对它的稳定性有极高的要求。为了提高软件的稳定性，PostgreSQL 只采用多进程的体系架构，拒绝使用线程。这是有历史原因的：在 PostgreSQL 诞生时，线程还是一个比较新的技术，所以 PostgreSQL 采用了传统的多进程体系架构。这种做法可能有点保守，也一直是开发社区中争论的话题。不使用线程导致 PostgreSQL 的性能相对于其它使用线程的数据库软件如 MySQL 等，并不是最领先的，但差距非常小。采用传统的多进程架构，使得 PostgreSQL 的稳定性非常高，是所有用户

都交口称赞的。正如本山大叔对我们的敦敦教诲：“不要看广告，要看疗效。谁用谁知道！”作为一枚苦逼的数据库运维人员，我深深地知道稳定性意味着什么：没有人想半夜三更被老板的电话叫醒。华为公司的开源数据库软件 openGauss 是基于 PostgreSQL 内核进行开发的，但进行了大量的改造，其中一个重大的变更就是使用了多线程技术。如果读者对这个话题感兴趣，可以去研究一下 openGauss。在本书写作的时候，采用多线程的架构在 PostgreSQL 内核社区又引发的广泛的关注，但是这涉及到对整个代码库的重大调整，改起来不容易，需要持续数年的努力。如果你对这个话题感兴趣，可以直接在官网上关注 PostgreSQL 内核开发的邮件列表中的讨论。

1.2.1.2 Postgres 的服务器进程

当 PostgreSQL 数据库运行起来后，我们可以使用 ps 命令可以查看它有哪些进程在运行，下面是一个 ps 输出结果的例子：

```
/* 所有的pg进程都以postgres开头，所以可以用grep postgres进行过滤 */
$ ps -ef | grep postgres | grep -v grep
postgres 11216 378 0 09:15 ? 00:00:00 /opt/software/pg16/bin/postgres
postgres 11217 11216 0 09:15 ? 00:00:00 postgres: checkpointer
postgres 11218 11216 0 09:15 ? 00:00:00 postgres: background writer
postgres 11220 11216 0 09:15 ? 00:00:00 postgres: walwriter
postgres 11221 11216 0 09:15 ? 00:00:00 postgres: autovacuum launcher
postgres 11222 11216 0 09:15 ? 00:00:00 postgres: logical replication launcher
```

上面的输出显示后台有好几个 postgres 的进程。按照源代码里面的分类标准，PostgreSQL 的进程可以分为三类：

- 主进程 (postmaster)
- 后台进程 (background process)，即图 1.2 中围绕大矩形框的一系列进程。
- 后端进程 (backend process)，即图 1.2 中的 postgres 进程。

在不需要刻意区分时，这三种进程统一被称为“服务器进程”，这个术语是针对客户端软件的进程而言的。服务器进程的生命周期不尽相同，有的服务器进程会一直存在，直至整个数据库被关闭。有的服务器进程会根据需要而启动，一旦完成自己的工作，就会自动退出。当 PostgreSQL 数据库启动时，运行的程序是 \$PGHOME/bin/postgres 这个可执行文件。该程序产生的第一个进程在源代码内部被叫做 postmaster，它被称为“主进程”。主进程会首先完成一些初始化的工作，最重要的步骤之一就是创建一块大的共享内存。紧接着主进程会通过 fork() 这个非常重要的系统调用创建一系列的子进程。这些子进程被称为“后台”进程。它们类似生产流水线旁的工人，在后台默默地工作，分别处理不同的任务。在完成各种初始化工作后，主进程会监听某一个 TCP 端口（缺省是 5432）。当某个客户端程序打算连接到数据库时，它首先用 PostgreSQL 服务器的 IP 地址和监听端口号与主进程建立一个 TCP 连接，主进程会通过系统调用 fork() 函数创建一个后端进程，这个后端进程将一对一和客户端的进程建立 TCP 连接，并对客户端进行身份验证，验证通过后，就为其服务。主进程类似看门大爷或者甩手掌柜，基本上什么也不过问，有客人来访就登记一下，然后直接甩给别人。

函数 fork() 是著名的 Linux 系统调用，它的任务是创建子进程，本书后面的章节会介绍它的知识。我们经常可以说：进程 A fork 出了进程 B，就是指进程 A 调用了 fork() 函数，产生出一个子进程 B，这是计算机领域的一句行话。为了简化设计，提高稳定性，PostgreSQL 采用了一个非常容易理解的模型：所有的后台进程和后端进程统一由主进程创建，主进程的工作尽量简化，能交给子进程做的事情主进程绝对不染指。譬如一个后台进程 A 想启动另外一个进程 B，它并不是自己调用 fork()，而是向主进程发某种信号。主进程接收到该信号后，创建子进程 B，进程 A 和进程 B 是兄弟关系。这种简约的设计模型经过三十多年的实践，被证明是非常稳定的，可谓简约而不简单。在本书后面的论述中，为了简化，可能会说：自动清理 (Autovacuum) 子系统的启动 (launcher) 进程创建了工作 (worker) 进程，这句话就是说启动进程通知主进程帮它创建了工作进程。请读者稍微留意一下这种简化表述背后的真实过程，以利于更好地理解技术细节。

主进程通过 fork() 系统调用，创建了多个后台进程，如检查点进程 (checkpointer)，后台写进程 (background writer)，WAL 写进程 (walwriter)，自动清理的启动进程 (autovacuum launcher) 和逻辑复制的启动进程 (logical replication launcher) 等等。下面列出这些后台进程的主要功能，其中涉及的一些概念和知识，我们目前还不具备，所以你只要大致有个印象就行了。

- 检查点进程负责数据库周期性的重要操作 - 检查点，这个概念会在第三章重点介绍。
- 后台写进程，有时候简称 bgwriter，它会周期性地把内存中的脏数据页写回到磁盘上的数据文件里。
- WAL 写进程，和后台写进程类似，周期性地把内存中的 WAL 记录写到磁盘上的 WAL 文件里。
- 自动清理的启动进程在图 1.2 中被缩写为 AV launcher，它会在满足一定条件下，创建自动清理的工作进程清理数据文件中已经“死亡”的记录。
- 逻辑复制的启动进程在数据库配置逻辑复制功能后，会创建逻辑复制的工作进程从远端的数据源中抓取指定的数据。

在上面的 ps 命令的输出中，请注意一下 11216 号进程，此进程就是主进程。如果你的 Linux 环境支持 pstree 命令，可以执行如下命令：

```
$ pstree -Ap 11216
postgres(11216)-+-postgres(11217)
    |-postgres(11218)
    |-postgres(11220)
    |-postgres(11221)
    `--postgres(11222)
```

命令 pstree 可以用树形结构显示多个进程之间的血缘关系。上述的进程树很清楚地表明：进程 11217, 11218, 11220, 11221 和 11222 的父进程都是 11216。下面我们运行客户端 psql 连接到数据库后，再查看进程信息：

```
$ psql
psql (16.0)
Type "help" for help.

postgres=# /* \! 是在psql中调用shell的命令，注意!后有一个空格，否则会出错。 */
postgres=# \! ps -ef | grep postgres | grep -v grep
postgres  11216      378  0 09:15 ?          00:00:00 /opt/software/pg16/bin/postgres
postgres  11217  11216  0 09:15 ?          00:00:00 postgres: checkpointer
postgres  11218  11216  0 09:15 ?          00:00:00 postgres: background writer
postgres  11220  11216  0 09:15 ?          00:00:00 postgres: walwriter
postgres  11221  11216  0 09:15 ?          00:00:00 postgres: autovacuum launcher
postgres  11222  11216  0 09:15 ?          00:00:00 postgres: logical replication launcher
postgres  11303      674  0 09:35 pts/2      00:00:00 psql
postgres  11304  11216  0 09:35 ?          00:00:00 postgres: postgres [local] idle
postgres=# \! pstree -Ap 11216      /* 查看进程树 */
postgres(11216)-+-postgres(11217)
    |-postgres(11218)
    |-postgres(11220)
    |-postgres(11221)
    |-postgres(11222)
    `--postgres(11304)
postgres=# \! ps -ef | grep 11304 | grep -v grep
postgres  11304  11216  0 09:35 ?          00:00:00 postgres: postgres [local] idle
```

```
postgres=# \! ps -ef | grep 11303 | grep -v grep
postgres  11303     674  0 09:35 pts/2    00:00:00 psql
postgres=# \! ps -ef | grep 674 | grep -v grep
postgres   674     666  0 08:46 pts/2    00:00:00 -bash
postgres  11303     674  0 09:35 pts/2    00:00:00 psql
```

仔细对比前一个实验的输出，你会发现这次实验多出了两个进程：11303 和 11304，其中 11303 进程是客户端进程 psql，11304 进程是后端进程。11304 进程是由主进程创建的，用于服务 11303 客户，两者是一对一的关系。注意：11303 号进程并不是主进程创建的，因为它的父进程的进程号是 674。674 号进程是 bash 进程，因为我们是在 bash 中运行 psql 的嘛。本实验中，客户端 psql 和数据库运行在同一台机器上，进程 11303 和 11304 之间的连接是“本地连接”。如果客户端程序在另外一台机器上连接 PostgreSQL 数据库，这种连接是“远程连接”。本地连接和远程连接并无太多本质上的不同，关于远程连接的配置和使用，本书后面会介绍。结合上述实验和图 1.2，相信你已经大致理解了 PostgreSQL 服务器进程的类型和相互关系。

1.2.2 PostgreSQL 的内存结构

任何进程都需要使用内存，进程的内存分为两种：私有内存 (private memory) 和共享内存 (shared memory)。私有内存又被称为“本地内存”(local memory)，它只供本进程独享，所以是“私有”的，通常情况下，别的进程无法访问本进程的私有内存。进程和线程最大的区别之一是进程都有自己独立的内存空间，而线程却没有，所以多个线程可以同时访问本进程的私有内存。私有内存是通过类似 malloc() 的系统调用 (system call) 向操作系统申请的，通过 free() 的系统调用释放给操作系统。

共享内存，顾名思义，就是允许多个进程对这块内存进行读写操作，共享之。共享内存是通过 shmget()/mmap() 等系统调用创建，使用 shmctl()/munmap() 等系统调用释放。在图 1.2 中，最大的矩形框表示共享内存。在 PostgreSQL 的共享内存里有很多组件，包括共享池 (shared buffer pool), WAL 缓冲池 (WAL buffer) 等等。图 1.3 展示了 PostgreSQL 的共享内存和私有内存的关系。

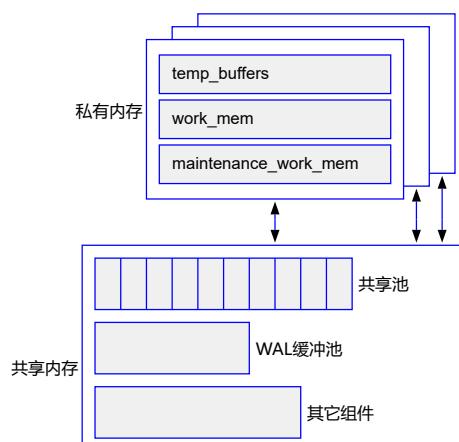


图 1.3: 共享内存和私有内存的关系

图 1.3 中下半部的大矩形框代表共享内存，它是由主进程在刚启动时创建。主进程的孩子们，就是所有的后端进程和后台进程，都可以读写这块内存。图 1.3 中上半部的三个大矩形分别表示三个后端进程的私有内存，一个进程一份。共享内存中包括了共享池，WAL 缓冲池和其它组件，私有内存中包括了 work_mem, maintenance_work_mem 等组件。私有内存，共享内存和磁盘文件之间都存在着双向的传输。命令 ipcs 可以查询 PostgreSQL 数据库使用共享内存的情况：

```
$ ipcs
```

```
----- Message Queues -----
key          msqid      owner      perms      used-bytes     messages

----- Shared Memory Segments -----
key        shmid      owner      perms      bytes      nattch      status
0x00060c4f 0          postgres    600          56          6

----- Semaphore Arrays -----
key        semid      owner      perms      nsems
```

命令 ipcs 显示了 System V 进程间通讯资源的使用情况，包括共享内存，信号量和消息队列。上面的输出显示数据库只使用了 56 个字节的共享内存。实际上 PostgreSQL 使用 mmap() 系统调用创建了大块内存，在 ipcs 中是看不到的，本书后面的章节会探究其中的原因。

在共享内存中最引人注目的组件是共享池，它是整个共享内存中体积最大的组件，往往会占整个共享内存 90% 左右的份额。PostgreSQL 数据库中的数据存储在数据文件中，数据文件通常会按照 8KB(由源代码中的常量 BLCKSZ 规定)字节的尺寸划分成一个个的数据块 (block)。这些数据块被读入到内存后就放在共享池中，变成了一个个 8KB 的数据页 (page)。你可以把共享池理解为一个巨大的数组，其每个成员都是 BLCKSZ 个字节大小的共享内存块。共享池的尺寸在主进程启动后就固定下来，在数据库运行期间不能改变。图 1.4 展示了共享池，数据文件，数据页和数据块的基本关系。

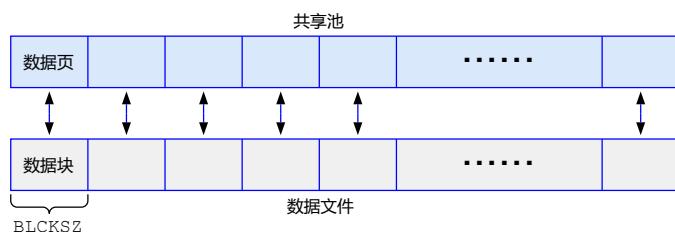


图 1.4: 共享池和磁盘数据文件的关系

在图 1.4 中，上半部长长的大矩形代表共享池，下半部长长的大矩形框代表磁盘上的数据文件。它们分别被划分成固定尺寸的数据页和数据块。在磁盘上的数据块和内存中的数据页的体积都是一样的，缺省情况下都是 8192 字节，两者之间是一对一的关系，数据库中的数据就在数据页和数据块之间来回传输。关于共享池更具体的内容，我们会本书的后续章节进行分析。

在图 1.2 中，另外一个引人注意的共享内存组件是 WAL 缓冲池，它和磁盘上的 WAL 文件对应，WAL 的知识是第三章的核心内容。共享内存中的其它组件，也会随着研究的深入而慢慢被我们熟悉，这里不展开论述了。

1.2.3 数据库集群

数据库存储的大量业务数据当然是放在磁盘上的数据文件 (data file) 中的。在 PostgreSQL 中有一个重要的术语：数据库集群 (database cluster)，它是一堆文件和操作这些文件的后台进程的总称。当数据库集群被关闭后，进程消失了，只剩下磁盘上的一堆文件。“集群”这个概念很容易让初学者迷惑，因为在大部分人的头脑里，集群指的是协同工作的多台机器的集合，这些机器可以被集群软件有效地管理成为一个整体，例如 Oracle 数据库中的 RAC，SQL Server 中的 Windows 集群。但在 PostgreSQL 的世界里，数据库集群的所有文件都放在一个目录中，这个目录被称为“数据库集群目录”，以后我们用 \$PGDATA 来指代这个目录。在这个目录中包含了多个数据库，每个数据库就是一个子目录，多个数据库在一起，因此成了“集群”。所以 PostgreSQL 的数据库集群是运行在单台机器上的，这一点可能会让初学者有点失望。多台机器组成的真正的数据库集群，涉及的技术非常复杂，成熟的产品不是很多，譬如折腾很久的 PostgreSQL-XL 项目进展缓慢。如果你有兴趣，可以去研究一下 Greenplum 数据库，它是基于 PostgreSQL 内核的真正的集群数据库，而且已经开源了。正在运行的数据库集群

也可以被称为数据库实例 (instance)，后文中我们会交叉使用这两个术语。图 1.5 展示了 PostgreSQL 数据库集群的文件的逻辑结构：

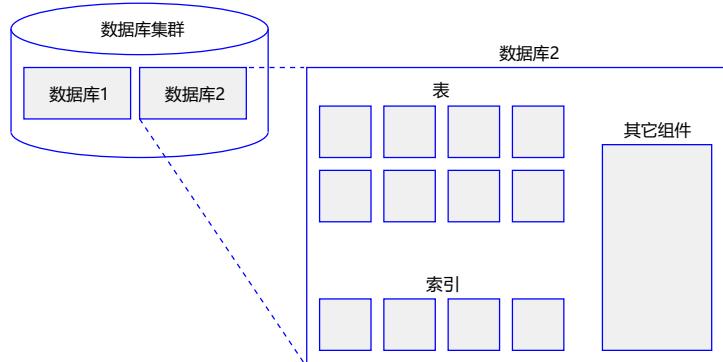


图 1.5：数据库集群的基本结构

图 1.5 中的数据库集群包括了数据库 1 和数据库 2 两个数据库。把数据库 2 放大后进行观察，可以看见数据库中包含了表，索引等常见的数据库对象。数据库 1 和数据库 2 就是数据库集群目录下的两个子目录而已，后面我们会看到这一点。

1.2.3.1 数据库集群目录的布局

数据库集群所有的文件都放在一个目录中，由环境变量 \$PGDATA 指向它。当我们查看这个目录时，会发现它里面有很多文件和子目录，如下所示。因为读者现在是初学，所以我并不打算逐一介绍每个子目录和文件的作用，随着学习的不断进行，我们会慢慢熟悉这些目录和文件的。

```
$ ls -l $PGDATA
total 128
drwx----- 6 postgres postgres 4096 Apr  5 08:36 base /* <-- 大量的数据库文件存放在这个目录下 */
drwx----- 2 postgres postgres 4096 Apr  5 08:36 global
.....
drwx----- 2 postgres postgres 4096 Apr  5 08:33 pg_tblspc
.....
drwx----- 3 postgres postgres 4096 Apr  5 08:30 pg_wal /* <-- 这个目录也非常重要，第三章讨论它 */
...
```

在数据库集群目录中，大量的数据往往是放在 base 目录。在这个目录中，每一个子目录代表一个数据库。数据库创建伊始，有三个缺省的数据库：template0，template1 和 postgres。其中，template0 和 template1 是模板库，postgres 是保存着重要信息的系统库。一般情况下，请不要修改这三个数据库里面的内容。上一节中创建的数据 oracle 是真正用于业务开发和运行的数据库，它是由模板库复制而来的。下面的实验展示了一个数据库集群中的数据库信息。

```
/* 你也可以使用 \l 的快捷命令来显示数据库的列表 */
postgres=# select oid, datname from pg_database order by oid;
   oid   | datname
-----+-----
      1 | template1
      4 | template0
      5 | postgres
 16388 | oracle
```

```
(4 rows)
/* 我们可以看到，四个数据库的oid就是在base目录下的四个子目录 */
postgres=# ! ls -l $PGDATA/base
total 16
drwx----- 2 postgres postgres 4096 Oct  1 09:18 1
drwx----- 2 postgres postgres 4096 Oct  1 09:16 16388
drwx----- 2 postgres postgres 4096 Oct  1 09:08 4
drwx----- 2 postgres postgres 4096 Oct  1 09:17 5
```

PostgreSQL 号称是面向对象的关系型数据库，数据库中的对象都有唯一的标识 Oid(object identifier)，它是 4 个字节的无符号整数。从上面的输出可知，oracle 数据库的 Oid 是 16388。在数据库集群的目录下还有一个重要的目录是 pg_wal，数据库中最重要的 WAL 文件就保存在此目录中，第三章会介绍 WAL 的知识。

1.2.3.2 同时运行多个数据库集群

在实际工作中，为了节省资源，我们往往需要在一台机器上运行多个数据库集群。因为一个正在运行的数据集群就是一个目录加上若干后台进程，不同的数据集群放在不同的目录下就可以了，所以只要用 initdb -D 的方式，指定不同的目录，就可以创建不同的数据集群目录。两个数据集群进程有冲突的唯一可能性是两个数据集群的主进程都试图侦听在同一个端口，所以你还要保证同一台机器上的每个集群拥有不同的 TCP 侦听端口。举例来说，数据集群 1 的目录是 /opt/data/pgdata1，它的 TCP 侦听端口是 5432。数据集群 2 的目录是 /opt/data/pgdata2，它的 TCP 侦听端口是 5433，则这两个数据集群就不会打架。TCP 侦听端口是由参数 port 来规定的，所以我们需要在参数文件 postgresql.conf 中修改这个参数确保两个数据集群的侦听端口不同。下面的实验展示了如何在一台机器上同时运行两个数据集群：

```
$ pwd
/opt/data
$ ls -l /* 现在 /opt/data 目录下只有上一节创建的第一个数据集群 pgdata1 */
total 4
drwx----- 19 postgres postgres 4096 Oct  1 09:15 pgdata1
/* 下面使用 initdb 命令再创建一个数据集群 pgdata2 */
$ initdb -D /opt/data/pgdata2
.....
/* 此处省去XXXX个字 */
.....
$ ls -l /* 再次查看，发现数据集群 pgdata2 已经被成功创建了 */
total 8
drwx----- 19 postgres postgres 4096 Oct  1 09:15 pgdata1
drwx----- 19 postgres postgres 4096 Oct  1 09:46 pgdata2
/* 为了防止两个数据集群打架，我们要修改 $PGDATA/postgresql.conf 文件中的参数 port */
$ cat pgdata1/postgresql.conf | grep port
port = 5432          /* 第一个数据集群的侦听端口是 5432 */
$ cat pgdata2/postgresql.conf | grep port
port = 5433          /* 第二个数据集群的侦听端口是 5433 */
$ pg_ctl start -l logfile1 -D /opt/data/pgdata1    /* 启动数据集群 pgdata1 */
waiting for server to start.... done
server started
$ pg_ctl start -l logfile2 -D /opt/data/pgdata2    /* 启动数据集群 pgdata2 */
waiting for server to start.... done
server started
```

```
$ ps -ef | grep postgres
postgres 11504 378 0 09:50 ? 00:00:00 /opt/software/pg16/bin/postgres -D /opt/data/pgdata1
postgres 11505 11504 0 09:50 ? 00:00:00 postgres: checkpointer
postgres 11506 11504 0 09:50 ? 00:00:00 postgres: background writer
postgres 11508 11504 0 09:50 ? 00:00:00 postgres: walwriter
postgres 11509 11504 0 09:50 ? 00:00:00 postgres: autovacuum launcher
postgres 11510 11504 0 09:50 ? 00:00:00 postgres: logical replication launcher
postgres 11527 378 0 09:51 ? 00:00:00 /opt/software/pg16/bin/postgres -D /opt/data/pgdata2
postgres 11528 11527 0 09:51 ? 00:00:00 postgres: checkpointer
postgres 11529 11527 0 09:51 ? 00:00:00 postgres: background writer
postgres 11531 11527 0 09:51 ? 00:00:00 postgres: walwriter
postgres 11532 11527 0 09:51 ? 00:00:00 postgres: autovacuum launcher
postgres 11533 11527 0 09:51 ? 00:00:00 postgres: logical replication launcher
$ pstree -Ap 11504 /* 数据库集群pgdata1的内核成员列表如下: */
postgres(11504)-+-postgres(11505)
    |-postgres(11506)
    |-postgres(11508)
    |-postgres(11509)
    `--postgres(11510)
$ pstree -Ap 11527 /* 数据库集群pgdata2的内核成员列表如下: */
postgres(11527)-+-postgres(11528)
    |-postgres(11529)
    |-postgres(11531)
    |-postgres(11532)
    `--postgres(11533)
/* 登录不同的数据库集群，要使用不同的端口，下面的命令登录数据库集群pgdata2 */
$ psql -p 5433
psql (16.0)
Type "help" for help.

postgres=#
```

1.2.3.3 表空间的基本概念

数据库集群目录中包含一个子目录 pg_tblspc，这个子目录记录了表空间 (tablespace) 的相关信息。表空间是一些数据库中的常见概念，譬如 Oracle 数据库中就有表空间的概念，但 PostgreSQL 的表空间和 Oracle 的表空间含义有所不同。PostgreSQL 缺省情况下会把所有数据库的表，索引等对象放在 \$PGDATA/base 目录下，该目录被称为缺省表空间 (pg_default)。但这个目录的磁盘空间终归有限，所以需要一种手段把数据库的对象放置在另外一块磁盘上。Linux 等类 Unix 的操作系统没有 Windows 操作系统中的 C: 盘和 D: 盘等概念。在 Linux 中，所有的磁盘都会被挂载 (mount) 到一个统一的文件系统当中，从使用者的角度来看，一块磁盘的挂载点 (mount point) 就是一个目录。PostgreSQL 中的表空间本质上就是一个目录，下面我们就通过一个简单的实验来创建一个表空间，看看它长什么样子。

```
/* 查看一下本数据库集群目前的表空间情况，只有两个已经存在的表空间 */
postgres=# SELECT * FROM pg_tablespace;
oid  | spcname   | spcowner | spcacl | spcoptions
-----+-----+-----+-----+
1663 | pg_default |      10 |        |
```

```

1664 | pg_global |      10 |      |
(2 rows)

postgres=# \! ls -l $PGDATA/pg_tblspc /* 查看一下pg_tblspc的系统目录，里面是空的 */
total 0

/* 创建一个新目录/opt/data/tblspace, 这是一个空目录 */
postgres=# \! mkdir /opt/data/tblspace
postgres=# \! ls -l /opt/data/tblspace
total 0

/* 创建一个表空间mytbs, 指向了/opt/data/tblspace */
postgres=# CREATE TABLESPACE mytbs LOCATION '/opt/data/tblspace';
CREATE TABLESPACE

/* 再次查看，发现表空间mytbs创建成功了，其oid是16392*/
postgres=# SELECT * FROM pg_tablespace;
   oid  | spcname   | spcowner | spcacl | spcoptions
-----+-----+-----+-----+
  1663 | pg_default |      10 |      |
  1664 | pg_global |      10 |      |
  16392 | mytbs    |      10 |      |
(3 rows)

/* 查看pg_tblspc系统子目录，里面有一个快捷方式指向了目标目录*/
postgres=# \! ls -l $PGDATA/pg_tblspc
total 0
lrwxrwxrwx 1 postgres postgres 18 Oct  1 09:56 16392 -> /opt/data/tblspace
/* 查看目标目录，发现里面有了一子目录 */
postgres=# \! ls -l /opt/data/tblspace
total 4
drwx----- 2 postgres postgres 4096 Oct  1 09:56 PG_16_202307071

```

一个数据库集群包含两个缺省的表空间，其中 `pg_default` 指向了 `$PGDATA/base` 这个目录，如果你不明确指定，所有的数据库对象都会存在这个目录中。`pg_global` 指向了 `$PGDATA/global` 目录，在这个目录中包含了一些全局的系统表。创建表空间的命令 `CREATE TABLESPACE` 本质上就是给磁盘上某个目录起了个名字而已，一个表空间就对应一个目录。如果让表空间指向一块磁盘的挂载点或其下的一个子目录，就可以让数据库的表或者索引保存在这个磁盘中，这就解决了 `base` 目录空间不够的问题。非缺省表空间都会在 `pg_tblspc` 这个目录下创建一个链接 (symbolic link) 指向对应的表空间。表空间和数据库集群，数据库的关系可以归纳为：一个数据集集群中可以有多个表空间，也可以有多个数据库。数据库和表空间是多对多的关系，即一个数据库中的对象可以保存在不同的表空间内，一个表空间可以存放多个数据库的对象，如图 1.6 所示。

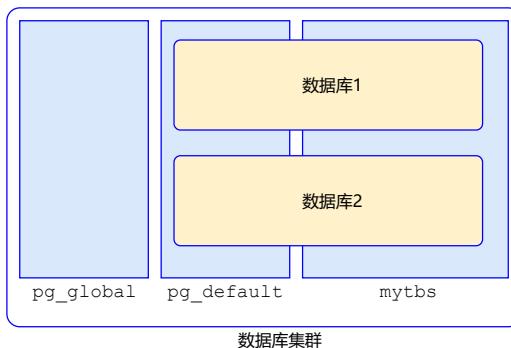


图 1.6: 表空间和数据库的关系

现在我们在 oracle 数据库中创建一个表，存放在刚刚创建的 mytbs 表空间中：

```
/* 在oracle数据库中创建一个表，放置在mytbs表空间中 */
oracle=# CREATE TABLE department(id INT PRIMARY KEY, name VARCHAR(32)) TABLESPACE mytbs;
CREATE TABLE
oracle=# SELECT pg_relation_filepath('department'); /* 查看一下该表的具体位置 */
      pg_relation_filepath
-----
pg_tblspc/16392/PG_16_202307071/16388/16393
(1 row)

oracle=# \! ls -l /opt/data/tblspace/PG_16_202307071
total 4
drwx----- 2 postgres postgres 4096 Oct  1 09:58 16388
oracle=# \! ls -l /opt/data/tblspace/PG_16_202307071/16388
total 0
-rw----- 1 postgres postgres 0 Oct  1 09:58 16393
```

由上可知，我们可以在 CREATE TABLE 命令的后面加上 TABLESPACE 选项来指定这张表要存放在哪个表空间下。表 department 被保存在了 /opt/data/tblspace/PG_16_202307071/16388/16393 这个文件中了，而不是缺省的 \$PGDATA/base 目录下。引入了表空间的概念后，如果想定位某张表，就需要一个三元组 (a,b,c) 来表示：

- a = 该表所在的表空间的 Oid
- b = 该表所在的数据库的 Oid
- c = 该表的 Oid

这个基本概念会帮助我们理解后面的某些关键性的数据结构，请参考数据结构 RelFileLocator，其定义如下：

```
typedef unsigned int Oid;
typedef Oid RelFileName;
/* in src/include/storage/relfilenode.h */
typedef struct RelFileLocator {
    Oid          spcOid;        /* tablespace */
    Oid          dbOid;         /* database */
    RelFileName relNumber;     /* relation */
} RelFileLocator;
```

很清楚，RelFileLocator 是一个三元组，它包含了定位一张表的全部信息：这个表所在的表空间，数据库和表本身的 Oid。

1.2.4 辅助文件

在数据库集群中，除了占体积最大的数据文件和同样重要的 WAL 文件以外，还有一些小不点的辅助文件，在数据库集群的运行过程中也起到了不可忽视的作用。这些文件有参数文件，控制文件，锁文件等等。我们使用如下命令列出一个数据库集群所有的辅助文件：

```
$ ls -1F $PGDATA | grep -v /
total 128
-rw----- 1 postgres postgres      3 Oct  1 09:08 PG_VERSION
-rw----- 1 postgres postgres  5711 Oct  1 09:08 pg_hba.conf
-rw----- 1 postgres postgres  2640 Oct  1 09:08 pg_ident.conf
-rw----- 1 postgres postgres     88 Oct  1 09:08 postgresql.auto.conf
```

```
-rw----- 1 postgres postgres 29696 Oct  1 09:48 postgresql.conf
-rw----- 1 postgres postgres     57 Oct  1 09:50 postmaster.opts
-rw----- 1 postgres postgres    84 Oct  1 09:50 postmaster.pid
$ cat $PGDATA/PG_VERSION
16
```

上面的输出展示了一个数据库集群里面的所有辅助文件，其中 PG_VERSION 是一个非常简单的文本文件，里面包含了 PG 的主版本信息。PostgreSQL 的软件内部都包含一个写死的版本号，主进程在启动阶段会读取 PG_VERSION 里面的版本号信息，和程序内部写死的版本号进行对比，如果不一致，就拒绝启动。所以这个文件虽然简单，但是也不要随手删除掉。下面我们依次介绍其它辅助文件的内容和作用。

1.2.4.1 参数文件

为了控制数据库的各种行为，必然要有一个参数文件记录各种参数。几乎每一个数据库都有成百上千的参数。PostgreSQL 数据库集群的参数文件有四个，如下所示：

```
$ ls -l $PGDATA/*.conf
-rw----- 1 postgres postgres  5711 Oct  1 09:08 /opt/data/pgdata1/pg_hba.conf
-rw----- 1 postgres postgres   2640 Oct  1 09:08 /opt/data/pgdata1/pg_ident.conf
-rw----- 1 postgres postgres    88 Oct  1 09:08 /opt/data/pgdata1/postgresql.auto.conf
-rw----- 1 postgres postgres 29696 Oct  1 09:48 /opt/data/pgdata1/postgresql.conf
```

其中 pg_hba.conf 是控制客户端连接的参数文件，pg_ident.conf 是管理用户映射的参数文件，因为暂时用不到这两个文件，所以本章就跳过它们。下面重点介绍主参数文件 postgresql.conf 和辅助的参数文件 postgresql.auto.conf。

在 postgresql.conf 文件的开头部分有一段清晰的注释来说明该文件的用法，参数文件中的参数都是以“名 = 值”(name = value)的形式来设置的，每个参数的值可以有 kB/MB/GB/TB 等体积单位，也可以有 us/ms/s/min/h/d 等时间单位。参数文件的注释是用 # 来表示的，和 shell 脚本的注释方式相同。主进程在启动初期会根据某些参数计算出共享内存的尺寸，按照这个尺寸创建的共享内存的体积在数据库集群运行周期内是不能改变的。所以，当影响共享内存体积的某个参数修改后，为了让新值生效，就必须重新启动数据库实例。对于一些不影响共享内存尺寸的参数，则无需重启数据库实例，只要执行 pg_reload_conf() 函数就可以让数据库实例重新加载这个参数的新值。至于哪些参数需要使用 pg_reload_conf() 函数进行重新加载，哪些参数必须重新启动数据库集群才能生效，我的方法是不用去死记硬背，而是使用一个小诀窍：当修改完参数后首先执行 pg_reload_conf()，然后使用 show 命令查看该参数的新值是否生效。如果没有，则该参数的修改需要重新启动数据库集群才能生效，具体操作请参考下面的实验。由于手工修改 postgres.conf 比较繁琐，PostgreSQL 效法了 Oracle 的 spfile 的概念，又设立了一个辅助参数文件 postgresql.auto.conf。你可以使用“ALTER SYSTEM SET name = value”的命令，把修改后的值存放在 postgresql.auto.conf 中。下面的实验演示了如何修改参数并使其生效的过程。

```
/* 首先查看postgresql.auto.conf里面的内容，除了注释，空空如也 */
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
postgres=# SHOW work_mem; /* 显示一下当前的参数work_mem */
work_mem
-----
4MB
(1 row)

postgres=# ALTER SYSTEM SET work_mem = 10240; /* 把参数work_mem修改为10MB */
ALTER SYSTEM
postgres=# SHOW work_mem; /* 再次检查该参数的值，发现没有生效，因为还是老值 */
```

```

work_mem
-----
4MB
(1 row)
/* 执行pg_reload_conf()函数，刷新一下。这个参数的修改无需重新启动数据库实例 */
postgres=# SELECT pg_reload_conf();
pg_reload_conf
-----
t
(1 row)
postgres=# SHOW work_mem; /* 参数work_mem的新值已经生效了 */
work_mem
-----
10MB
(1 row)
/* 查看postgresql.auto.conf的内容，发现多了一行，即ALTER SYSTEM命令修改的内容 */
postgres=# \! cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
work_mem = '10240'

```

在上述实验中，如果执行 `pg_reload_conf()` 函数后，该参数的值依然是老值，则说明需要重新启动数据库才行。但是重启数据库不是一件随便就可以做的事情，你需要提前规划好，往往是在晚上执行，以免影响白天的正常业务运行。参数文件的原则是“后者为大”，即：如果同一个参数被设置了两次，PostgreSQL 会以后面的值为准。譬如在下面的例子中，`wal_level` 先后设置了两次，最终的有效值是 `replica`，而不是 `logical`：

```

$ cat postgresql.conf | grep wal_level
wal_level = logical
wal_level = replica

```

主进程启动后，会先读取 `postgresql.conf` 文件，再读取 `postgresql.auto.conf` 文件，根据“后者为大”的原则，`postgresql.auto.conf` 里面的参数设置为最终的有效值。在参数文件中还可以使用 `include` 指令，把其它参数文件的内容包含进来，其机制类似 C 语言的 `include` 头文件。一般而言，直接在 `postgresql.conf` 中修改就足够了，只有在特别复杂的情况下才使用 `include`。

1.2.4.2 控制文件

参数文件中的参数描述的都是数据库集群的静态状况，一旦修改生效完毕后就一直保持不变。而在数据库集群启动和运行时，也需要一些重要的参数描述其运行的状态，这些参数会不断发生变化，而记录这些变化的参数的文件就是控制文件。控制文件是 `$PGDATA/global/pg_control`，它是一个很小的二进制文件，体积有 8192 个字节，但真正的内容只有几百个字节。PostgreSQL 提供了一个实用小工具 `pg_controldata` 来展示控制文件里的内容，请参考下面的实验。

```

$ pg_controldata -D /opt/data/pgdata1
pg_control version number:          1300
Catalog version number:            202307071
Database system identifier:        7285007205183695787
Database cluster state:           in production
.....

```

在上面的输出中，大家可以观察到一个有趣的参数：数据库系统标识符 (Database system identifier)。它实际上是 8 字节，在 initdb 创建数据库集群的过程中，通过某种算法随机产生的唯一的标识符，用来标识该数据库集群。还有一个参数：数据库集群状态 (database cluster state)，它表明数据库集群是在运行还是停止状态中。控制文件中的其余内容在后文中会逐步涉及到，在此我们就不展开论述了。

1.2.4.3 锁文件

当数据库实例启动时，PostgreSQL 会创建一个锁文件 postmaster.pid，里面记录了主进程的进程号，侦听端口，启动时间等参数，该文件在数据库实例关闭后被自动删除。下面是它里面的具体内容：

```

0 $ cat $PGDATA/postmaster.pid
1 11504
2 /opt/data/pgdata1
3 1696175428
4 5432
5 /tmp
6 localhost
7     64982      11
8 ready

```

锁文件的第一行是主进程的进程号，第二行是数据库集群的目录，第三行是数据库实例启动时的系统时间，在 PostgreSQL 内部，时间往往是 8 个字节，如果要变成人可读的时间信息，还要进行一下转换。第四行是 TCP 侦听端口，第五行是 Unix 域套接字 (Unix domain socket) 所在的目录，客户端进行本地连接时就使用域套接字。第六行是数据库实例侦听的 IP 地址，第七行是 System V 共享内存的键 (key) 和 ID(shmid)，第八行表示数据库实例的状态。其中有些信息我们目前还不理解，可以忽略之。

1.2.4.4 主进程参数文件

稍微熟悉使用 C 语言进行命令行软件编程的读者都知道，在入口函数 main() 中有两个参数 argc 和 argv，通常用于各种输入参数来控制程序运行时的行为，下面简单的 C 程序演示了这些输入参数的含义。

```

$ cat arg.c
#include <stdio.h>
int main(int argc, char* argv[])
{   for(int i=0; i<argc; i++) printf("argv[%d] = [%s]\n", i, argv[i]);  return 0; }
$ gcc -Wall arg.c -o arg
$ ./arg -D /opt/pgdata/data1
argv[0] = [./arg]
argv[1] = [-D]
argv[2] = [/opt/pgdata/data1]

```

由上可知：C 程序的入口函数 main 有两个输入参数，一个是整型参数 argc 记录输入参数的个数，另一个是字符串数组 argv 记录每一个参数的内容。argv[0] 表示本程序的文件名，其后的输入参数通过空格来区分。\$PGHOME/bin/postgres 是 PostgreSQL 软件的核心程序，所有的服务器进程都来自这个程序。我们常使用 pg_ctl 来启动和关闭数据库实例，其实它只是为了方便用户而设计的一个“包裹”(wrapper) 程序。它在幕后实际上是启动 postgres 这个核心程序。我们也可以跳过 pg_ctl，直接运行 postgres 程序，请参考下面的操作。

```

/* 不使用pg_ctl，直接启动postgresql数据库实例 */
$ $PGHOME/bin/postgres -D /opt/data/pgdata1 > logfileX 2>&1 &
[1] 11770

```

```
/* 查看postmaster.opts里面的内容 */
$ cat /opt/data/pgdata1/postmaster.opts
/opt/software/pg16/bin/postgres "-D" "/opt/data/pgdata1"
```

当数据库实例崩溃后，在诊断原因时，可能需要知道主进程在启动时使用了哪些输入参数，主进程参数文件 postmaster.opts 就记录着这些参数。每次数据库实例启动时，该文件里的内容都会被重写，其具体的操作在 CreateOptsFile() 函数完成。这个函数的代码非常简单，读者稍微读一下就能明白它的内容。

```
/* in src/backend/postmaster/postmaster.c */
#define OPTS_FILE      "postmaster.opts"
static bool CreateOptsFile(int argc, char *argv[], char *fullprogname) { .... }
```

至此，我们已经对 PostgreSQL 的体系结构有了一个初步了解。从第二章开始，我们就要逐步深入学习 PostgreSQL 的各个组件的相关内容。

第二章 数据文件

地球人都知道：在数据库中，真正的业务数据是存放在数据文件中的。对于数据文件结构的理解，是我们学习更加深入知识的前提，所以本章将讲解 PostgreSQL 数据文件的基本结构。为了更深刻地理解所学的知识，我们在学习 PostgreSQL 外在功能的同时，要阅读相关的源代码，内外验证，这是迅速提高技术水平的不二法门。所以本章的第一节先对 PostgreSQL 源代码的基本知识做一些介绍。

2.1 PostgreSQL 源代码的基础知识

2.1.1 如何搜索源代码

PostgreSQL 作为经典的软件作品，其源代码质量非常高，且可读性超过 MySQL 的源代码。虽然如此，PostgreSQL 16 已经有一百多万行的源代码，分布在共计 2000 多个头文件 (*.h) 和源代码文件 (*.c) 文件中，所以阅读它的源代码是一个巨大的工作，需要长期的努力。我们可以使用如下方法统计 PostgreSQL 16 有多少行源程序：

```
$ pwd  
/home/postgres/postgresql-16.0/src  
$ find . -name *.h | wc -l  
1056  
$ find . -name *.c | wc -l  
1296  
$ find . -name *.[hc] | xargs wc -l | grep total  
1724040 total
```

由上可知，PostgreSQL 一共有 1056 个头文件和 1296 个源代码文件。所有这些文件一共包含了 172 万行 (1724040) 的源代码。我们研读分析源代码时，往往希望能快速地搜索到某个数据结构、函数或变量的定义。PostgreSQL 的在线源代码阅读网站 (doxygen.postgresql.org) 使用非常方便，推荐大家使用。你也可以在互联网上寻找一些源代码索引软件，但如何使用它们需要一些学习成本，我在 Linux 平台下经常使用一种几乎不需要学习的暴力搜索方式，如下所示：

```
/* 在当前目录和其子目录的所有*.h文件中搜索字符串 PageHeaderData */  
$ find . -name \*.h | xargs grep PageHeaderData  
/* 在当前目录和其子目录的所有*.c文件中搜索字符串 ShmemInitStruct */  
$ find . -name \*.c | xargs grep ShmemInitStruct  
/* 如果当前目录下没有*.c文件，你可以把烦人的\去掉 */  
$ find . -name *.c | xargs grep ShmemInitStruct  
/* 对结果可使用typedef或define进一步过滤，快速找到数据结构的定义 */  
$ find . -name \*.h | xargs grep PageHeaderData | grep typedef  
$ find . -name \*.h | xargs grep BLCKSZ | grep define
```

本书中，为了免去读者的搜索之苦，当我引用某些数据结构和源代码时，会用类似/* in src/include/storage/bufpage.h */的注释表示所引用的数据结构和代码是在 bufpage.h 文件中定义的。这是一个相对路径，假设源代码目录是/code，则 bufpage.h 文件的绝对路径就是/code/src/include/storage/bufpage.h，这样方便你直接打开该文件进行查找和阅读。类似/* xlog.c:CreateCheckPoint() */的注释则表示 xlog.c 文件中的 CreateCheckPoint() 函数。此外，有些复杂的数据结构的成员变量依然比较复杂，我会把其成员变量的数据结构也罗列出来，让读者在一页纸的范围内就可以了解该数据结构，避免了来回翻书查找。

源代码有一些头文件里定义了大量的宏，用来控制源代码的编译行为，如 src/include/pg_config_manual.h，它们可以被称为配置头文件。其中有三个配置头文件 pg_config.h, pg_config_os.h 和 pg_config_ext.h 不在源码包中。

当你运行 `configure` 命令后，`configure` 根据对操作系统环境的检测，自动生成这些头文件，或链接到和平台相关的头文件上。如果你在阅读源代码时发现无法找到某些宏的定义，可以运行 `configure` 命令产生这三个配置头文件，可能你要找的宏就在这三个头文件中。

2.1.2 基本的数据类型

在 PostgreSQL 源代码中使用了大量的基础数据类型，我们需要提前熟悉它们。常用的基础数据类型的定义如下：

```
/* in src/include/c.h */
typedef signed char int8;          /* == 8 bits */
typedef signed short int16;         /* == 16 bits */
typedef signed int int32;           /* == 32 bits */
typedef unsigned char uint8;        /* == 8 bits */
typedef unsigned short uint16;      /* == 16 bits */
typedef unsigned int uint32;         /* == 32 bits */
typedef uint8 bits8;               /* >= 8 bits */
typedef uint16 bits16;              /* >= 16 bits */
typedef uint32 bits32;              /* >= 32 bits */
typedef long int int64;
typedef unsigned long int uint64;

typedef size_t Size; /* 8 bytes in 64-bit Linux */
typedef uint32 TransactionId;
/* in src/include/postgres_ext.h */
typedef unsigned int Oid;
```

大家很容易熟悉上面自定义的基础数据类型的命名规律：`intXX` 是有符号整数，可以表示负数和正数。`uintXX` 是无符号整数，只能表示正数，最小值为 0。其中 `XX` 为 8/16/32/64，表示这个数据类型有多少个比特 (bit)。`Size` 也是源代码中被大量使用的一个基础数据类型，在 64 位平台上它有 8 个字节的长度。另外两个常见的数据类型是 `Oid` 和 `TransactionId`，`Oid`(Object Id) 表示某一个对象 (表空间，数据库，表，索引等) 的唯一性标识。`TransactionId` 表示事务 (transaction) 的唯一性标识，这两个数据类型都是 4 个字节的无符号整数。

本书在引用数字时，会采用 C 语言的语法规范，十六进制用 `0x` 作为前缀，二进制则用 `0b` 作为前缀，没有前缀的数字为十进制，例如 `0x1A` 表示十进制的 26，`0b0111` 表示十进制的 7。如果读者觉得这些进制之间的转换十分费脑，可以借助 Windows 操作系统自带的计算器进行不同进制之间的互相转换。

2.1.3 内存对齐

目前常见的计算机分为 32 位 (32-bit) 和 64 位 (64-bit)，早期的计算机还有 8 位 (8-bit) 和 16 位 (16-bit) 的类型。由于 IT 技术的迅猛发展，现在市场上的智能手机都是 64 位了，更不要说计算机能力更加强大的服务器，所以本书假设 PostgreSQL 运行在 64 位的计算机上。所谓 32 位计算机，指的是 CPU 的数据总线 (data bus) 和地址总线 (address bus) 都是 32 比特 (4 字节)，64 位计算机的 CPU 数据总线和地址总线则是 64 比特 (8 字节)。以 64 位计算机来说，CPU 一次性从内存中会读取 8 个字节。譬如你想访问 6 号地址内存单元中一个字节，则 CPU 一条读内存指令就把 0 到 7 号地址的共计 8 个字节都读入 CPU 内部的寄存器中，然后只挑选 6 号的一个字节使用。如果你想读取 6 号到 9 号地址之间的 4 个字节，则 CPU 需要读取内存两次：第一次读取 0 到 7 号地址的 8 个字节，第二次读取 8 到 15 号地址的 8 个字节，共计 16 个字节，然后在 CPU 内部拼接后获得 6 到 9 号的 4 个字节，这种操作无疑是低效率的。

为了提高 CPU 读写内存的速度，就产生了“对齐”的概念，其思想就是确保被访问的数据在内存中的起始

地址和数据的尺寸都是 8 的整数倍，被称为“按 8 字节对齐”，对齐技术可以让 CPU 减少一次访问内存的操作。PostgreSQL 源代码中大量充斥着对齐的操作，我们需要提前熟悉它的基本规律，方便后面对源代码的学习。对齐操作中使用最多的是 MAXALIGN 宏，下面是它的定义：

```
/* in src/include/pg_config.h */
#define MAXIMUM_ALIGNOF 8      /* 8个字节表示PG运行在64位的操作系统上 */

/* in src/include/c.h */
#define TYPEALIGN(ALIGNVAL,LEN) \
    (((uintptr_t) (LEN) + ((ALIGNVAL) - 1)) & ~((uintptr_t) ((ALIGNVAL) - 1)))

#define MAXALIGN(LEN)  TYPEALIGN(MAXIMUM_ALIGNOF, (LEN))
```

根据以上的定义，你稍微心算一下就很容易推导出如下结果：

```
MAXALIGN(x) = ((uintptr_t) (x) + 7) & ~((uintptr_t) (7))
/* ~((uintptr_t) (7) = 0xFFFFFFFFFFFFFFF8 */
```

那么 uintptr_t 又是个什么东西呢？在 C99 的标准中，uintptr_t 是系统库头文件 <stdint.h> 定义的一个数据类型。在 64 位的机器上，它就是一个 8 字节的无符号整数。在 PostgreSQL 官方文档中，有这么一句话：

```
Code in PostgreSQL should only rely on language features available in the C99 standard.
```

这句话告诉想为 PostgreSQL 添砖加瓦的 C 语言程序员：你写的源代码必须遵循 C99 的标准。C99 标准是 20 多年前的有关 C 语言的国际标准，已经比较古老了。之所以有这个规定，是为了确保 PostgreSQL 可以运行在各种操作系统上，包括比较古老的操作系统。由此可知：7(0b0111) 按照 8 个字节进行取反操作，其结果为 0xFFFFFFFFFFFFFFF8。如果一个值 x 是 8 的整数倍，则 MAXALIGN(x) = x。如果 x 不是 8 的整数倍，MAXALIGN(x) 就往比它大的且是 8 的整数倍的那个数上凑。例如 x = 21，它介于 16 (= 2 X 8) 和 24 (= 3 X 8) 之间，它就往 24 上凑：MAXALIGN(21) = 24。当然 MAXALIGN(17) 到 MAXALIGN(24) 的值都是 24。我们可以记住几个规律：

- MAXALIGN(x) 是 8 的整数倍。
- MAXALIGN(x) \geq x，且最多比 x 大 7。
- MAXALIGN(0) = 0

在 PostgreSQL 的源代码中常有这样的代码：alignedSize = MAXALIGN(size)；其中 size 表示要申请的内存大小（单位是字节）。在分配内存之前，要通过类似的语句把内存的尺寸按 8 个字节做齐，得到一个新尺寸 alignedSize，这样申请下来的内存块的大小就是按照 8 字节对齐的。这种做齐的方式虽然浪费了几个字节，但是提高了软件的性能，这种编程手法值得我们借鉴和运用。PostgreSQL 源代码中还有其它类似的对齐定义的宏，我们遇到后再临时分析一下也不迟，这里就不过多介绍了。

2.2 数据文件

毫无疑问，数据文件是数据库中体积最大，也是最重要的文件之一。数据在数据文件中可以按行存储，也可以按列存储，由此产生了行式数据库（row oriented databases）和列式数据库（columnar database）两种类型。传统的关系型数据库，包括 Oracle, Microsoft SQL Server, MySQL 和 PostgreSQL 都是行式数据库。本节对 PostgreSQL 数据文件的基本结构进行初步研究，为后续更深入的知识学习做好铺垫。

2.2.1 数据文件的基本结构

在 PostgreSQL 数据库中，每一张表对应一个数据文件，其文件名都是数字，且一张表的最大尺寸是 32TB。很显然，这么巨大的文件不便于管理，常见的操作系统对于文件的尺寸亦有限制。譬如在 32 位的操作系统中，文件的最大尺寸不能超过 2GB 或者 4GB。我们可以把一张表的文件理解为一个“虚拟文件”，或称之为“逻辑文件”。为了方便管理，逻辑文件可以进一步被切分成很多尺寸相同的物理文件，这些物理文件被称为该逻辑文件的“段”（segment），如图 2.1 所示。

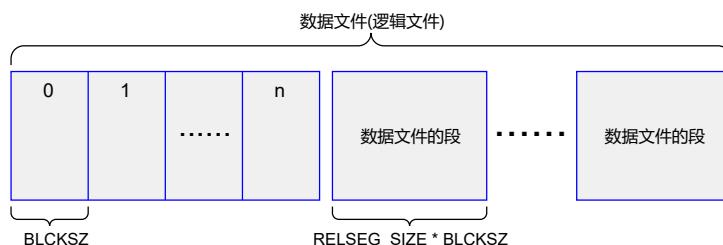


图 2.1：数据文件的段

缺省情况下，PostgreSQL 把数据文件的段的尺寸设置为 1G，这个尺寸在各种操作系统下基本上都没有问题。数据文件的每个段，又被细分成“数据块”（block），简称为“块”。PostgreSQL 的源代码中有两个常量，BLCKSZ 规定了块的大小，单位是字节，而 RELSEG_SIZE 则规定了段的大小，单位是块，其定义如下：

```
/* in src/include/pg_config.h */
#define BLCKSZ 8192
#define RELSEG_SIZE 131072
```

由上可知，缺省情况下每个块是 8192 字节。每个段的体积则是 RELSEG_SIZE * BLCKSZ，即 1GB 大小（= 131072×8192 ）。假设一张表在磁盘上对应的文件的名字是 16389，该文件实际上是该表的第一个段，当表的体积超过 1GB 后，数据库就为它创建第二个段，文件名叫 16389.1，第三个段的文件名则叫 16389.2，依次类推。

数据文件中的块的编号是一个 4 字节的无符号整数，从 0 开始。这个编号是跨段的，就是一个表的所有段文件的数据块统一编号。因此一张表中块的最大编号是 $0xFFFFFFFF(2^{32}-1)$ ，表的最大尺寸就是 32TB（= $4GB \times 8KB$ ），这个尺寸足够使用了。如果某张表真有这么大，该反思的就是你了：为什么把表搞这么大？数据块的最大尺寸是 32KB，所以一张表的最大体积理论上可以达到 128TB。下面的源代码定义了数据块编号的数据类型。

```
/* in src/include/storage/block.h */
typedef uint32 BlockNumber;
#define InvalidBlockNumber ((BlockNumber) 0xFFFFFFFF) /* 无效的块编号 */
#define MaxBlockNumber ((BlockNumber) 0xFFFFFFF) /* 最大的块编号 */
typedef struct BlockIdData { /* 把块编号进一步分为高低两部分 */
    uint16 bi_hi;
    uint16 bi_lo;
} BlockIdData;
```

参数 BLCKSZ 和 RELSEG_SIZE 是在编译时确定的，一旦软件编译完成后就不能更改了。配置命令 configure 提供了两个选项，可供在编译之前指定新的尺寸。

```
$ ./configure --help | grep SIZE /* 在PostgreSQL的源代码目录中执行该命令 */
--with-blocksize=BLOCKSIZE
--with-segsize=SEGSIZE  set table segment size in GB [1]
--with-segsize-blocks=SEGSIZE_BLOCKS
--with-wal-blocksize=BLOCKSIZE
```

在绝大多数情况下，缺省的 8KB 和 1GB 都是非常理想的尺寸，所以一般情况下没有必要去改变这两个参数的设置。除非特别指明，这两个参数的缺省值是本书后面内容的基本假设和前提。

2.2.2 一个简单的实验

为了研究数据文件，我们做一个简单的小实验，其过程如下所示：

```
postgres=# \c oracle /* 建议每次实验都不要在postgres数据库中操作 */
You are now connected to database "oracle" as user "postgres".
oracle=# CREATE TABLE state(id INT, name CHAR(2));
CREATE TABLE
/* 通过pg_relation_filepath()函数拿到表文件的路径 */
oracle=# SELECT pg_relation_filepath('state');
 pg_relation_filepath
-----
base/16384/16385
(1 row)
/* 通过ls -l命令查看这个文件在磁盘的信息 */
oracle=# \! ls -l $PGDATA/base/16384/16385
-rw----- 1 postgres postgres 0 Oct  1 12:36 /opt/data/pgdata1/base/16384/16385
/* 注意该文件的大小为0，因为这是一个空表，PG还没有为它分配磁盘空间 */
oracle=# INSERT INTO state VALUES(0, 'TX'); /* 现在往表中插入一条记录 */
INSERT 0 1
/* 为保证数据真正落盘，执行一个CHECKPOINT命令，其概念会在下一章介绍 */
oracle=# checkpoint;
CHECKPOINT
/* 再次查看该数据文件，发现它的大小为8192字节 */
oracle=# \! ls -l $PGDATA/base/16384/16385
-rw----- 1 postgres postgres 8192 Oct  1 12:37 /opt/data/pgdata1/base/16384/16385
```

以上实验结果表明，当一张表有了第一条记录后，其大小变成了 8192 字节，正好是一个块的大小。由此可知，当第一条记录被写入到数据文件中时，数据库为该文件在磁盘上分配第一个块，其编号是 0。下面我们就来研究一下这个块里面到底有什么东西。

2.2.3 数据块的结构

数据块 (block) 和数据页 (page) 是 PostgreSQL 世界里经常互换使用的两个术语。当一个数据块被读入到内存后，数据库实例会在共享池中为其分配 8K 的空间，这个在内存中的块被称为“数据页”，简称为“页”。简而言之：在磁盘上叫数据块，在内存中则为数据页，这两者的内容是一模一样的，一个字节都不差。在磁盘上的数据块有编号，在内存中的数据页也有自己的编号，两者是不同的概念。当然，PostgreSQL 在内存中为数据页进行编号时，也会记录这个数据页所对应的数据块的编号，从而建立起两个编号的映射关系。在本书的后面，块

和页这两个术语会根据上下文交替使用，请读者留意。整个数据页划分为四个区域：头部区域，数据指针区域，数据区域和特殊区域。图 2.2 来展示了一个数据页的基本结构。

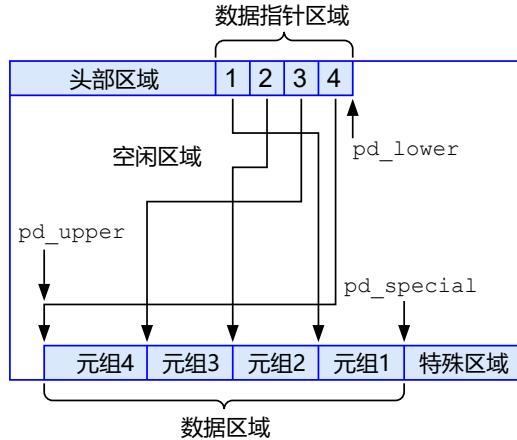


图 2.2: 数据页的基本结构

头部区域，顾名思义，就是在数据页的开始部分，也叫数据页的页头。其后是数据指针区域，数据指针区域是一个数组，我们称之为记录指针数组，有时候也叫 lp 数组。该数组的每个成员固定是 4 个字节，上图中用 1, 2, 3, 4 来表示。从数据页的尾部往前数，首先是特殊区域 (special area)，特殊区域的前面是数据区域，存放着真正的数据，即表中的记录。记录指针数组中的每个成员都指向数据区域的一条记录。这里需要注意两点：一是记录的编号从 1 开始，不是从 0 开始。其二，数据指针区域和数据区域是相向增加的，它们中间的空白部分就是本数据页的空闲空间。在图 2.2 中你可以看到 lp[1] 和 lp[2] 指针是交叉的，就不难理解了相向增加的含义了。在头部区域有三个指针：pd_lower，它指向了本页中空闲区域的开始地址；pd_upper，它指向了本页中空闲区域的结束地址；pd_special，它指向了特殊区域的开始地址。根据图 2.2，相信大家不难理解这三个指针的含义。

PostgreSQL 出身于计算机领域鼎鼎有名的加州伯克利大学，学院派气质比较浓厚，所以它的一些术语都来自数据库理论，譬如我们通常说的记录 (row/record) 在 PostgreSQL 领域中被称为“元组” (tuple)，而表 (table) 被叫做“关系” (relation)，列 (column) 则被称为“属性” (attribute) 等等。但是记录、表、列等术语更加常见，所以我们依然采用流行的术语，它们和英文文档中 PostgreSQL 的传统术语的对应关系，希望读者稍加留意。

2.2.3.1 页头的结构分析

数据页的头部区域是一个 C 语言的结构体 (struct)，叫做 PageHeaderData，其相关定义如下：

```
/* in src/include/c.h */
#define FLEXIBLE_ARRAY_MEMBER /* empty */

typedef uint32 TransactionId; /* 32-bit */

/* in src/include/storage/bufpage.h */
typedef uint16 LocationIndex;
typedef struct {
    uint32 xlogid; /* high bits */
    uint32 xrecoff; /* low bits */
} PageXLogRecPtr;

typedef struct PageHeaderData {
    PageXLogRecPtr pd_lsn; /* LSN */
    uint16 pd_checksum; /* checksum */
    uint16 pd_flags; /* flag bits */
    LocationIndex pd_lower; /* offset to start of free space */
}
```

```

    LocationIndex pd_upper;      /* offset to end of free space */
    LocationIndex pd_special;    /* offset to start of special space */
    uint16        pd_pagesize_version;
    TransactionId pd_prune_xid;  /* oldest prunable XID, or zero if none */
    ItemIdData    pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* line pointer array */
} PageHeaderData;

```

结构体 PageHeaderData 的重要成员变量介绍如下：

- pd_lsn 是 8 字节，表示 LSN。LSN 是 PostgreSQL 的核心概念，下一章会给出该概念的定义，现在忽略之。
- pd_checksum 是校验码，用于校验本数据页是否损坏。
- pd_lower/pd_upper/pd_special 是三个内部指针，指向本数据页内部的关键位置，请参考图 2.2。
- pd_pagesize_version 记录了本页的大小和 PostgreSQL 的版本信息。
- pd_prune_xid 是 pruning 的事务 id，以后会介绍，现在可以忽略。
- pd_linp 即上图中的 lp 数组，它是一个可变长的数组，每个成员的长度是 4 字节。

指针 pd_lower/pd_upper/pd_special 的长度都是 16 比特，它们是在本页内部的相对偏移量。假设一个指针 char* p 指向了某个数据页的开始位置，则 p + pd_lower 指向该页的空闲空间的起始位置，p + pd_upper 指向该页的空闲空间的结束位置，新数据的插入位置就很容易被计算出来。

万能分析工具 hexdump 可以把任何文件的每个字节都以 16 进制的格式显示出来，展示了文件最原始的样子。我们可以使用这个工具直接把上述实验中产生的数据文件的原始面貌扒出来，验证一下 PageHeaderData 的各成员变量。具体操作请参考下面的操作。

```

oracle=# ! ls -l $PGDATA/base/16384/16385
-rw----- 1 postgres postgres 8192 Oct  1 12:37 /opt/data/pgdata1/base/16384/16385
oracle=# ! hexdump -C $PGDATA/base/16384/16385
00000000  00 00 00 00 F0 09 87 01  00 00 00 00 1C 00 E0 1F  |....|
00000010  00 20 04 20 00 00 00 00  E0 9F 3E 00 00 00 00 00  |. ....>....|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |....|
*          /* 这行的星号表示重复上一行的内容很多次，减少屏幕上的输出 */
00001FE0  DC 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |....|
00001FF0  01 00 02 00 02 08 18 00  00 00 00 00 07 54 58 00  |.....TX..|
00002000

```

注意：hexdump 缺省情况下以小写字母表示十六进制中的六个字母。由于后面经常使用的 LSN 使用大写字母，所以我把 hexdump 输出结果中的 abcdef 都变成了大写的形式。

拿到 hexdump 的输出结果后，请稍微耐心一点，仔细对比输出结果中每个字节和源代码中的数据结构的相关定义，就很容易分析出各个成员变量的值。这里要注意是：X86-64 的 CPU 是小端结构 (little endian)，高位在高地址，低位在低地址，所以在分析多个字节组成的数据时，要把这些字节反转过来，才能得到正确的值。譬如，在上面的输出中的右上角有两个字节 E0 1F，因为 1F 在 E0 右边，是高地址，所以它是高位的值，真正的值是 0x1FE0。大小端的概念不难理解，请读者自行研究，基本上花费几分钟就可以搞明白。从上面的输出中不难计算出 PageHeaderData 的各成员变量的值：pd_lsn 的值是 0x18709F0；pd_checksum 的值是 0x00，表示页的校验功能没有打开；pd_flags 的值是 0；pd_lower 的值是 0x1C；pd_upper 的值是 0x1FE0；pd_special 的值是 0x2000；pd_pagesize_version 的值是 0x2004；pd_prune_xid 的值是 0。

从以上分析可知，pd_lower 的值是 0x1C，即 28，pd_upper 的值是 0x1FE0，即 8160，这说明从偏移量 28 到 8160 这个区域的空间是空闲的，即本数据页的空闲空间是 8132 个字节 (=8160 - 28)。注意：在数据页中，字节的编号从 0 到 8191。另外你可以看到 pd_special 的值 0x2000，即 8192，已经指向了该页的末尾，所以特殊区域在这个数据页中是不存在的。pd_pagesize_version 的值是 0x2004，它分为 0x2000 和 4 两部分，0x2000 表示本数据页的体积是 8192 字节，4 表示数据格式的版本是 4，这些数据基本上不会轻易改变。本节后面会介绍更加便捷

的工具来研究数据页，但 hexdump 是最朴素的分析工具，可以展示数据的最原始的面目，让你有更加实在的体验，所以在以后的分析中，我会交叉使用 hexdump 和 PostgreSQL 的分析工具。

这里还有一个小知识点，就顺手介绍一下。PageHeaderData 结构的第二个成员变量 pd_checksum 是两个字节的校验码，用来校验该数据块是否损坏。创建数据库的工具 initdb 有一个参数 -k 可以打开数据库的校验码功能。如果你在数据库创建完毕后想更改这个设置，可以先把数据库实例关闭，再使用 pg_checksums 工具来打开或关闭校验码功能。控制文件中有一个域记录着数据库实例是否打开了校验码功能，示例如下：

```
$ initdb --help | grep checksums
-k, --data-checksums      use data page checksums
/* 检查控制文件中的校验码状态：1表示校验码功能已经打开，0则表示关闭 */
$ pg_controldata | grep checksum
Data page checksum version:          0    /* 0 - 表示校验码没有打开，1 - 表示校验码已经打开 */
```

打开校验码功能肯定会带来一些性能上的损失。但是根据网上许多用户的反馈报告，这个性能似乎影响不大，所以建议在生产库中打开校验码功能。具体实施之前，你也可以做一些测试工作，再做决定。

2.2.3.2 数据指针的结构分析

数组 pd_linp 是可变长度的，可以为 0。一个数据块中有多少条记录，则 pd_linp 数组就有多少个成员。其成员变量的类型是 ItemIdData 结构，4 字节长，记录了对应的记录的位置和长度信息，相关数据结构如下：

```
/* in src/include/storage/itemid.h */
#define LP_UNUSED      0      /* unused (should always have lp_len=0) */
#define LP_NORMAL      1      /* used (should always have lp_len>0) */
#define LP_REDIRECT    2      /* HOT redirect (should have lp_len=0) */
#define LP_DEAD        3      /* dead, may or may not have storage */
typedef struct ItemIdData {
    unsigned lp_off:15,     /* offset to tuple (from start of page) */
                lp_flags:2,   /* state of line pointer, see below */
                lp_len:15;    /* byte length of tuple */
} ItemIdData;
```

这是一个非常小巧的数据结构，包含元组在一个数据块内部的偏移量和长度，以及 2 个比特的标志位，可表示四种不同的情况，标志位的四种含义也在上面的代码中列出了。图 2.3 可以帮助大家理解和记忆这个数据结构。

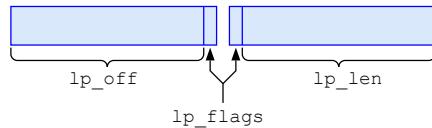


图 2.3: 元组指针 ItemIdData 的结构

因为 pd_lower/pd_upper/pd_special 三个本地指针的长度都是 16 个比特，其可寻址的空间是 $2^{16} = 64\text{KB}$ ，但在 ItemIdData 结构里的指针 lp_off 的长度是 15 个比特，其可寻址的空间只有 $2^{15} = 32\text{KB}$ ，所以 PostgreSQL 的数据块的最大尺寸为 32KB，而不是 64KB。通常情况下选择缺省的 8KB 足够使用了，这个值是对 OLTP 事务性数据库和 OLAP 分析性数据库比较折中平衡的设置，Oracle/SQL Server 数据库中的数据块的缺省尺寸也是 8KB。

如何知道一个数据块中有多少条记录呢？因为在 PageHeaderData 结构和 pd_lower 指向的位置之间是 pd_linp 数组，其每个数据成员的长度都是固定的 4 字节，而 PageHeaderData 结构的长度是 24 字节，所以 (pd_lower -

$24)/4$ 就是一个数据块中包含的记录的数量。举例来说，如果 $pd_lower=48$ ，则本 Page 中包含了 6 条记录 ($= (48 - 24)/4$)，这是一个很实用的小技巧，请读者稍加留意。下面的代码展示了这个逻辑，相信不难理解：

```
/* in src/include/c.h */
#define offsetof(type, field) ((long) &((type *)0)->field)
/* in src/include/storage/bufpage.h */
#define SizeOfPageHeaderData (offsetof(PageHeaderData, pd_linp))
static inline OffsetNumber PageGetMaxOffsetNumber(Page page)
{
    PageHeader      pageheader = (PageHeader) page;
    if (pageheader->pd_lower <= SizeOfPageHeaderData) return 0;
    else return (pageheader->pd_lower - SizeOfPageHeaderData) / sizeof(ItemIdData);
}
```

2.2.3.3 记录的结构分析

存储在数据块的每条记录，其结构都分为两个部分：记录头 HeapTupleHeaderData 和真正的数据区，如图 2.4 所示。

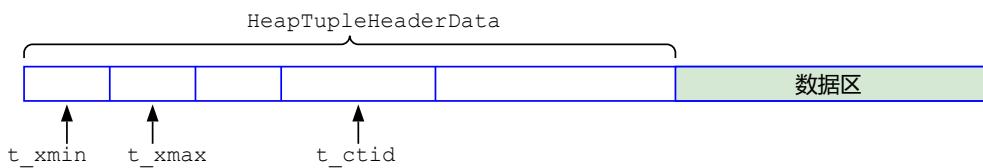


图 2.4: 记录的基本结构

数据结构 `HeapTupleHeaderData` 稍微复杂一点，目前我们只需观其大概即可，以后会详细分析。其定义如下：

```
/* in src/include/access/htup_details.h */
struct HeapTupleHeaderData {
    union {
        HeapTupleFields t_heap;
        DatumTupleFields t_datum;
    } t_choice;
    ItemPointerData t_ctid; /* current TID of this or newer tuple (or a
                           * speculative insertion token) */
    uint16 t_infomask2; /* number of attributes + various flags */
    uint16 t_infomask; /* various flag bits, see below */
    uint8 t_hoff; /* sizeof header incl. bitmap, padding */
    /* ^ - 23 bytes - ^ */
    bits8 t_bits[FLEXIBLE_ARRAY_MEMBER]; /* bitmap of NULLs */
    /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```

记录头最小长度是 23 个字节，也就是说，如果一张表只有一列，数据类型是 `CHAR(1)`，每条记录里面真正的数据只有一个字符，它也要 23 个字节的额外开销，所以这个开销是很大的。这是 PostgreSQL 设计上决定的，已经没有办法轻易优化了。那么一个数据块最多能存储多少条记录呢？因为每条记录最少 24 个字节，每条记录在头部还有 4 个字节的记录指针，所以一条记录就需要最少 28 个字节。扣除数据块的块头 24 个字节，所以一个 8KB 的数

据块可以存储的最大记录数是 $291 = (8192 - 24)/28$ 。有兴趣的读者可以参考如下的定义，`MaxHeapTuplesPerPage` 的值就是 291。

```
#define MaxHeapTuplesPerPage ((int) ((BLCKSZ - SizeOfPageHeaderData) / \
(MAXALIGN(PageSizeofHeapTupleHeader) + sizeof(ItemIdData))))
```

在 23 字节的记录头里面有很多信息，下面的实验展示了如何获取一些最常用的信息：

```
/* xmin, xmax, ctid 是隐藏列，每张表都有这些信息 */
oracle=# SELECT xmin, xmax, ctid, id, name FROM state;
xmin | xmax | ctid | id | name
-----+-----+-----+
726 | 0 | (0,1) | 0 | TX
727 | 0 | (0,2) | 1 | PA
(2 rows)
```

其中 `xmin` 表示插入这条记录的事务的事务号 (`xid`)，`xmax` 则表示删除这条记录的事务的事务号，如果 `xmax` 不是 0 的话，则表明该记录事实上已经被删除了。隐藏列 `ctid` 的长度是 6 字节，是 4 加 2 的结构，表示一个数据块的编号和 `pd_linp` 数据的下标，譬如 (5,2) 则表示 5 号数据块中的 `pd_linp` 数组的第二个成员，它是一个指针，可以指向同一条记录不同版本的数据。关于这部分的知识，我们在研究事务和多版本并发控制 (MVCC) 的章节会对其进行详细分析，在此就不展开论述了。

2.2.3.4 fsm 和 vm 文件

当你观察数据文件时，可能会发现类似下面的情况：

```
oracle# \! ls -l $PGDATA/base/16384/12597*
-rw----- 1 postgres postgres 8192 Mar 18 14:47 /opt/data/pgdata1/base/16384/12597
-rw----- 1 postgres postgres 24576 Mar 18 14:47 /opt/data/pgdata1/base/16384/12597_fsm
-rw----- 1 postgres postgres 8192 Mar 18 14:47 /opt/data/pgdata1/base/16384/12597_vm
```

由上可知，表文件 12597 还有两个兄弟文件，其文件名就是在表文件名之后加上了 `_fsm` 和 `_vm` 的后缀。`fsm` 和 `vm` 文件，是数据文件的辅助文件。假设一张表有 8GB 大小，那么它就有 $8\text{GB}/8\text{KB} = 1\text{M}$ 个数据块。当往该表中插入一条长度为 x 字节的新记录时，PostgreSQL 需要在这一百多万个数据块中快速寻找一个空闲空间大于 x 的数据块来存放该条记录。如何在这海量的数据块中快速寻找合适的候选者，就是个需要解决的问题。`fsm` 文件记录了数据文件中每个数据块的空闲空间的信息，所以查询 `fsm` 文件就可以拿到答案。为了快速搜索，`fsm` 文件内部被组成了二叉树的结构。`vm` 文件则是为了清除数据块中死亡记录的 `Vacuum` 操作中加速执行速度所使用。你可以把 `vm` 文件理解为一个长长的数组，每 2 个比特描述数据文件中的一个数据块。本书后面的章节会分析 `fsm` 文件和 `vm` 文件的技术内幕，在此大略了解其作用即可。这三种文件的关系可以用图 2.5 来解释。

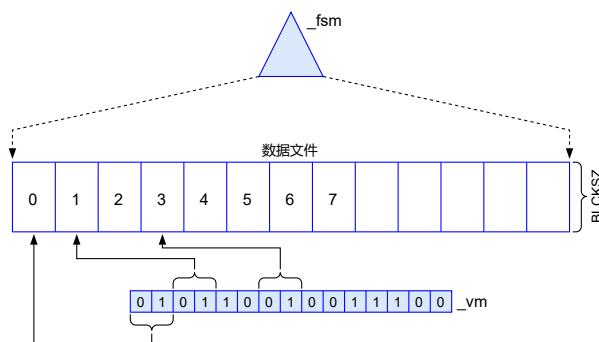


图 2.5: fsm/vm 文件和数据文件的关系

在源代码内部，使用了一个术语 Fork 来表示这三种数据文件的“衍生”类型，其定义如下：

```
/* in src/include/common/relop.h */
typedef enum ForkNumber {
    InvalidForkNumber = -1,
    MAIN_FORKNUM = 0,
    FSM_FORKNUM,
    VISIBILITYMAP_FORKNUM,
    INIT_FORKNUM
} ForkNumber;
```

PostgreSQL 中一共有 4 种衍生类型：存放真正数据的数据文件，其衍生类型是 MAIN_FORKNUM，值为 0，fsm 文件的衍生类型是 FSM_FORKNUM，其值为 1，vm 文件的衍生类型是 VISIBILITYMAP_FORKNUM，其值为 2。还有一种衍生类型叫 INIT_FORKNUM，其值为 3。和真正的数据文件的分块思想一模一样，fsm 文件和 vm 文件也是按照 8KB 固定大小的数据块来划分它们的数据文件的，而且它们的数据块也会读入到共享池中，和数据文件的数据页放在一个池子里。所以为了区分共享池中的数据页，需要一个五元组 (a,b,c,d,e) 的信息，其中 a 是表空间的 Oid，b 是数据库的 Oid，c 是表的 Oid，d 是数据块的编号，e 是这个数据块的衍生类型。这个不难理解。

2.2.4 分析数据块和内存页的工具

为了分析数据块或数据页的内容，PostgreSQL 提供了几款实用的小工具，这些工具都是深入研究技术内幕的有力助手。PostgreSQL 的很多工具都是以“扩展”(extension)的形式提供，也可以通俗地称之为“插件”。本节介绍 pageinspect 和 pg_buffercache 这两款工具的安装和基本使用。在第一章我们使用 make world 命令来编译源代码，这个 world 选项就是把 contrib 目录下的插件一并编译。所以只要你编译时加上了 world 选项，安装和使用这些插件就非常简单，下面的实验演示了这些工具的安装。

```
/* 安装插件必须用超级用户postgres。plpgsql是缺省安装的唯一插件 */
postgres=# SELECT oid, extname, extversion FROM pg_extension;
      oid      | extname | extversion
-----+-----+-----
 12756 | plpgsql | 1.0
(1 row)

postgres=# CREATE EXTENSION pageinspect; /* 安装pageinspect插件 */
CREATE EXTENSION
postgres=# CREATE EXTENSION pg_buffercache; /* 安装pg_buffercache插件 */
CREATE EXTENSION

/* 检查一下，两个插件都安装成功了 */
postgres=# SELECT oid, extname, extversion FROM pg_extension;
      oid      |   extname   | extversion
-----+-----+-----
 12756 | plpgsql     | 1.0
 16392 | pageinspect  | 1.12
 16437 | pg_buffercache | 1.4
(3 rows)
```

插件是跟着数据库走的，你在 postgres 数据库中安装好的插件，并不能在另外的数据库中使用。当切换到同一个数据库集群中的另外一个数据库后，为了使用这些插件，依然要执行 CREATE EXTENSION 命令来安装，好在安装非常简单，所以麻烦不大。

2.2.4.1 pg_buffercache 的使用

插件 pg_buffercache 的作用是分析共享池。共享池是共享内存中最大的组件，它本质上是一个巨大的数组，其成员就是一个个的数据页。共享池的大小在主进程启动后就固定分配下来，在整个数据库实例运行期间不能改变。重要参数 shared_buffers 规定了共享池的大小，单位是字节，共享池中一共有 shared_buffers / BLCKSZ 个数据页。假设 shared_buffers=128M，BLCKSZ=8KB，那么共享池里就有 16384 个数据页 (= 128MB/8KB)。在源代码中，共享池的数据页的总数量由一个全局变量 NBuffers 来记录，所以会有下面的公式：

```
shared_buffers = NBuffers * BLCKSZ
```

如果共享池中的某个数据页被修改后，还没有被写回到对应的数据块，这个数据页就变脏了，被称为脏页 (dirty page)。因为内存中的数据页和磁盘上的数据块都是 8KB 大小，没有多余的空间来保存如某个页是否为脏页等额外的信息，共享内存中有另外一个小的数据页描述数组 (buffer descriptor) 来存储这些额外的信息，请参考图 2.6。

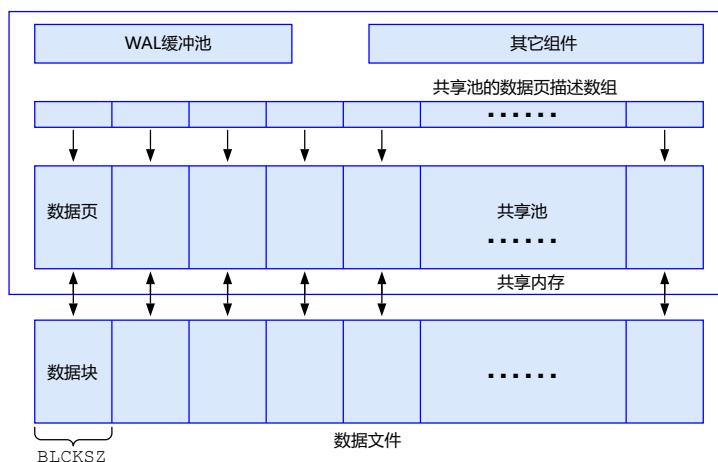


图 2.6: 共享池和数据文件的对应关系

数据页描述数组中的成员和共享池中的数据页是一一对应的，BufferDescriptor[x] 描述了 BuffePool[x] 的信息，其中 x 表示两个数组的下标，在源代码内部被称为 buffer id。插件 pg_buffercache 只提供了一个系统视图，它的每行记录就是数据页描述数组里一个成员的内容，描述了某个页是否是脏页等状态信息，所以 pg_buffercache 系统视图中共有 NBuffers 条记录，下面的小实验可以验证这一点。

```
postgres=# \d pg_buffercache
          View "public.pg_buffercache"
 Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+
bufferid | integer |           |           |
relfilenode | oid    |           |           |
reltablespace | oid   |           |           |
reldatabase | oid   |           |           |
relforknumber | smallint |           |           |
relblocknumber | bigint |           |           |
isdirty | boolean |           |           |
usagecount | smallint |           |           |
pinning_backends | integer |           |           |
postgres=# SHOW shared_buffers;
shared_buffers
```

```

-----
128MB
(1 row)

postgres=# SELECT count(*) FROM pg_buffercache;
 count
-----
16384
(1 row)

```

由上可知，pg_buffercache 视图里共有 16384 条记录， $16384 * 8192 = 128\text{MB}$ ，和 shared_buffers 的值完全吻合，即 NBuffers=16384。在 pg_buffercache 的表结构中，bufferid 是数组的下标，其取值范围从 0 到 NBuffers-1。pg_buffercache 对应的内部数据结构如下：

```

typedef unsigned int Oid;
typedef uint32 BlockNumber;
/* in src/include/common/relpath.h */
typedef Oid RelFileName;
/* in src/include/storage/buf_internals.h */
typedef struct buftag
{
    Oid          spcOid;           /* tablespace oid */
    Oid          dbOid;            /* database oid */
    RelFileName relNumber;        /* relation file number */
    ForkNumber   forkNum;         /* fork number */
    BlockNumber  blockNum;        /* blknum relative to begin of reln */
} BufferTag;

```

BufferTag 实际上就是前文说的五元组 (a,b,c,d,e)，描述了一个数据页的隶属关系，表示该数据页属于哪个表空间？哪个数据库？哪个表？是哪个数据块？哪种衍生类型 (main/fsm/vm)？对应着 pg_buffercache 系统视图中的 reltablespace, reldatabase, relfilenode, relforknumber 和 relblocknumber 五列信息。数据页描述数组的成员的真正的数据类型是 BufferDesc，其定义如下：

```

/* in src/include/port/atomics/arch-x86.h */
typedef struct pg_atomic_uint32 { volatile uint32 value; } pg_atomic_uint32;
/* in src/include/storage/buf_internals.h */
typedef struct BufferDesc
{
    BufferTag      tag;           /* ID of page contained in buffer */
    int             buf_id;        /* buffer's index number (from 0) */
    /* state of the tag, containing flags, refcount and usagecount */
    pg_atomic_uint32 state;
    int             wait_backend_pgprocno; /* backend of pin-count waiter */
    int             freeNext;       /* link in freelist chain */
    LWLock          content_lock;  /* to lock access to buffer contents */
} BufferDesc;

```

由上可知，对共享池中某个数据页的状态描述是长度为 4 字节的 state 变量，其具体含义请参考如下注释：

```

/* in src/include/storage/buf_internals.h */
/*

```

```
* Buffer state is a single 32-bit variable where following data is combined.
* - 18 bits refcount
* - 4 bits usage count
* - 10 bits of flags
*/
#define BM_DIRTY (1U << 23) /* data needs writing */
```

这段注释明确地说明了 state 变量中的 32 bit 的具体分配用途，我们就此打住，不再往下深究技术细节了，但其中的一个 bit 是 BM_DIRTY，表示该数据页是否是脏页，这个很容易理解。pg_buffercache 系统视图中的另外两列，usagecount 和 pinning_backends，我们现在还不能理解，暂时忽略。本书后面会专门有章节剖析共享池的内幕，到那时再来深入学习。至此，我们已经对该插件提供的信息大致有了一个理解，通过这个工具可以很容易查询共享池中每个数据页的具体情况，或者做一些统计查询，请参考下面的操作。

```
/* 查询编号为123的Page中的有关信息 */
oracle=# SELECT * FROM pg_buffercache WHERE bufferid=123;
-[ RECORD 1 ]-----+
bufferid      | 123
relfilename   | 1249
reltablespace | 1663
reldatabase   | 1
relforknumber | 0
relblocknumber| 21
isdirty       | f
usagecount    | 5
pinning_backends | 0
/* 查询Shared Buffer池中有多少个脏页 */
oracle=# SELECT count(*) FROM pg_buffercache WHERE isdirty = true;
      count
-----
88      /* ----- 一共88个脏页 */
(1 row)
/* 查询Shared Buffer池中有多少个fsm页 */
oracle=# SELECT count(*) FROM pg_buffercache WHERE relforknumber=1;
      count
-----
237     /* ----- 一共个fsm文件的页 */
(1 row)
```

2.2.4.2 pageinspect 的使用

插件 pg_buffercache 是从宏观上观察共享池的情况，并不深入每个数据页的内部，插件 pageinspect 的作用则是查看一个数据页或数据块中的信息，这两个工具配合起来使用，就能让我们了解所有的数据页的信息。和 pg_buffercache 不同，pageinspect 并不提供一个单一的系统视图，而是提供了一系列函数，分别是通用函数，针对堆表 (Heap) 的函数，针对不同类型的索引的函数。我把目前能用到的函数罗列在此：

- 函数 get_raw_page(relname text, fork text, blkno bigint)：返回指定数据页原始的 8192 个字节的数据。第一个参数 relname 是表的名字，blkno 则是数据块的编号，fork 则表示该数据块的衍生类型 (main = 0, fsm = 1, vm = 2)。

- 函数 `get_raw_page(relname text, blkno bigint)`：等于 `get_raw_page(relname, 'main', blkno)`，就是只查看数据文件本身的数据页，不考虑 fsm/vm 等衍生类型。
- 函数 `page_header(page bytea)`：返回某一个数据页的页头信息，可以参考数据结构 `PageHeaderData` 的定义来理解它的返回结果。
- 函数 `heap_page_items(page bytea)`：返回每一条记录的信息，可以参考记录头 `HeapTupleHeaderData` 的数据结构。

下面的几个例子展示了 `pageinspect` 工具的基本使用。

```
/* state里面只有两条记录，且每条记录非常短，故它只有一个数据块，编号为0 */
oracle=# SELECT * FROM state;
 id | name
----+---
 0 | TX
 1 | PA
(2 rows)

/* 查看0号块的页头信息，请参考PageHeaderData的定义理解之 */
oracle=# SELECT * FROM page_header(get_raw_page('state',0));
 lsn      | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 0/187BB10 |         0 |       0 |     32 |   8128 |    8192 |     8192 |       4 |        0
(1 row)

oracle=# \x
Expanded display is on.

/* 这里展示了每条记录的记录头的信息，请参考HeapTupleHeaderData结构理解之 */
oracle=#  SELECT * FROM heap_page_items(get_raw_page('state',0)) LIMIT 1;
-[ RECORD 1 ]-----
lp          | 1
lp_off      | 8160
lp_flags    | 1
lp_len      | 31
t xmin     | 726
t xmax     | 0
t_field3   | 0
t_ctid     | (0,1)
t_infomask2 | 2
t_infomask  | 2306
t_hoff     | 24
t_bits     |
t_oid      |
t_data     | \x000000000075458
```

此外 `pageinspect` 还提供了许多针对不同索引类型的函数，本书后面研究索引时会举例说明 `pageinspect` 各种索引函数的用法，在此就按下不表了。

2.3 性能测试工具 pgbench 的使用

在研究 PostgreSQL 技术的过程中，往往需要有一个足够体积和足够工作负荷 (work load) 的数据库，所以在本节我们介绍 PostgreSQL 自带的性能测试工具 pgbench，方便后续的学习之旅。很显然，如果测试工具和数据库集群运行在同一台机器上，测试工具的运行也会占用数据库服务器的资源，从而导致最终的测试数据不准确。我们应该有一台单独的客户机器来运行测试工具，如图 2.7 所示。

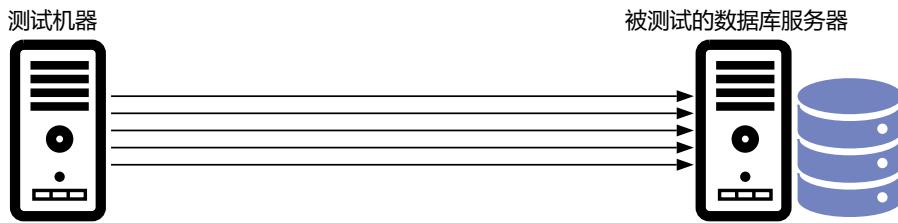


图 2.7：数据库性能测试的基本架构

在上图中，左边的机器是测试工具运行的机器，它产生指定的工作负荷，发往右边的被测试的数据库服务器。这里面就涉及到如何配置 PostgreSQL 远程连接的问题。在以往的实验中，我们使用客户端工具 psql 连接到数据库，psql 和 PostgreSQL 数据库集群都在一台机器上，这种连接方式叫做本地连接。更普遍的情况是数据库集群在远端的一台服务器上，譬如云端。我们需要使用本地的 psql 访问远端的数据库，这种连接就是远程连接。下面我们就先讲解如何配置 PostgreSQL 的远程连接。

2.3.1 配置远程连接

PostgreSQL 有一个重要的参数 listen_addresses，它表示主进程侦听的 IP 地址。它的缺省值是 localhost，表示侦听本地环路 127.0.0.1，此时外部的客户端是无法连接到这个 IP 地址的，所以我们必须把它的值改为星号，如下所示：

```
$ cat $PGDATA/postgresql.conf | grep listen
listen_addresses = '*'
```

星号表示主进程会侦听数据库服务器上的所有 IP 地址。这样的话，外部的客户端才有可能通过数据库服务器的正常的 IP 地址连接进来，这是允许远程连接的第一步。修改这个参数后要重新启动数据库集群才能生效。

PostgreSQL 通过一个配置文件 pg_hba.conf 来控制允许谁远程连接进来，拒绝谁远程连接。hba 是“基于主机的认证”(host-based authentication) 的缩写。缺省情况下，pg_hba.conf 存放在数据库集群的根目录下，你也可以使用 postgresql.conf 中的 hba_file 参数指定它的位置。关于如何配置 pg_hba.conf 的问题，我们不需要深入，只是通过例子做最常用的配置介绍，满足我们的学习需要就行。打开这个文件，里面的注释写的比较清楚，每一行为一条记录，分为 5 个域。我们仅仅在最后加上如下两行：

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	host	all	all	192.168.137.0/24	md5

因为第一行以 # 开头，表示注释，这一行仅仅是为了展示第二行的每列的含义。真正起作用的是第二行，它一共分为 5 列。第一列 host 表示这是一个远程连接，其余的远程连接的类型还包括 hostssl, hostnoss, hostgssenc, hostnogssenc 等。host 类型是最简单和最普遍的远程连接类型。第二列表示允许连接的数据库，all 表示允许连接所有的数据库。第三列表示允许连接的用户名，all 表示所有的用户均可连接。第四列表示允许连接的客户端的 IP 地址范围。在我使用的实验环境里，客户端和服务器都在 192.168.137.0 这个网段，子网掩码是 255.255.255.0，就是 3 个 8，可以缩写为 24，所以 ADDRESS 这个域我们写成了“192.168.137.0/24”，表示只要是 192.168.137 开头的 IP 地址都被允许进行远程连接。类似的，“192.168.137.12/32”则表示只有 192.168.137.12 的 IP 地址才可以

连接。第五列表示连接时使用的认证方式，md5 表示采用口令认证。所以这一行的意思是：只要你客户端来自 192.168.137.0 这个网段，你可以用任何用户身份访问所有的数据库，但是需要输入口令。修改 pg_hba.conf 后并不需要重启数据库集群，只要执行以下 pg_reload_conf() 刷新一下配置信息即可。

目前我们只有一个超级用户 postgres，我们就使用这个用户测试远程连接。既然远程连接需要口令，而我们在本地连接的时候，并不需要输入口令，所以我们也不知道 postgres 初始口令是什么。我们可以用本地连接登录数据库后，按照如下方式修改 postgres 的口令：

```
$ psql -U postgres /* 以超级用户登录数据库 */
psql (16.0)
Type "help" for help.
/* 使用\password来修改自己的口令 */
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=#

```

完成以上配置以后，我们可以使用另外一台机器上的 psql 进行测试。

```
$ psql -h 192.168.137.12 -U postgres -d oracle
Password for user postgres:
psql (16.0)
Type "help" for help.
/* 下面这条命令显示自己的连接信息 */
oracle=# \conninfo
You are connected to database "oracle" as user "postgres" on host "192.168.137.12" at port "5432".
```

其中的参数-h 表示连接的机器的名字或者 IP 地址，-U 表示连接的用户名，-d 表示连接哪个数据库。如果这个数据库集群侦听在 5433 端口，你可以使用参数-p 指定连接的端口号。如果不指定，端口的缺省值是 5432。这几个参数在很多 PostgreSQL 的客户端软件中都是通用的，pgbench 也是使用这些参数进行远程连接的。

2.3.2 pgbench 的基本使用

pgbench 是 PostgreSQL 自带的性能测试工具，能够满足基本的性能测试需要。它的使用分为两步，第一步是初始化测试环境，第二步是进行性能测试。我们先看看第一步，如何初始化测试环境。首先在数据库集群中创建一个测试数据库：

```
$ psql
psql (16.0)
Type "help" for help.
/* 创建测试数据库，名字随便起。所有的测试数据都在这个数据库中。*/
postgres=# CREATE DATABASE mydb;
CREATE DATABASE
postgres=#

```

然后我们进行测试数据库的初始化工作。这一步可以在客户端远端执行，也可以在数据库服务器端本地执行。

```
$ pgbench -h 192.168.137.12 -U postgres -d mydb -i -s 1
Password:
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
```

```

NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.15 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.65 s (drop tables 0.00 s, create tables 0.01 s,
client-side generate 0.32 s, vacuum 0.19 s, primary keys 0.13 s).

```

初始化的时候必须指定参数-i 表示进行初始化。初始化的工作主要是创建四张表，并往其中注入一些初始数据，如下所示：

```

postgres=# \c mydb
You are now connected to database "mydb" as user "postgres".
mydb=# \dt+
          List of relations
 Schema |        Name         | Type  | Owner   | Persistence | Size   | Description
-----+---------------------+-----+-----+-----+-----+-----+
 public | pgbench_accounts | table | postgres | permanent  | 13 MB | 
 public | pgbench_branches | table | postgres | permanent  | 40 kB | 
 public | pgbench_history  | table | postgres | permanent  | 0 bytes |
 public | pgbench_tellers  | table | postgres | permanent  | 40 kB | 
(4 rows)

```

其中 pgbench_accounts 表的体积最大，它里面有 100000 条记录。初始化命令面的参数-s 表示倍增因子，如果它的值为 2，则四张测试表里的记录数加倍。所以我们控制这个参数的值，很容易创造出一个体积足够大的数据库。初始化完毕后，就可以各种执行性能测试任务了。譬如我们执行如下命令：

```
$ pgbench -h 192.168.137.12 -U postgres -d mydb1 -j 2 -c 20 -T 300
```

上述的命令中，参数-j 表示使用多少个线程，-c 表示模拟多少同时连接的并发用户数，-T 表示测试的执行时间。这条命令的含义是使用 2 个线程模拟 20 个并发用户，向数据库服务器发起缺省负载的压力，持续时间是 300 秒。在这里面比较容易让初学者迷惑的是线程和并发用户数的关系。首先我们要了解客户端的同步模式和异步模式，如图 2.8 所示。

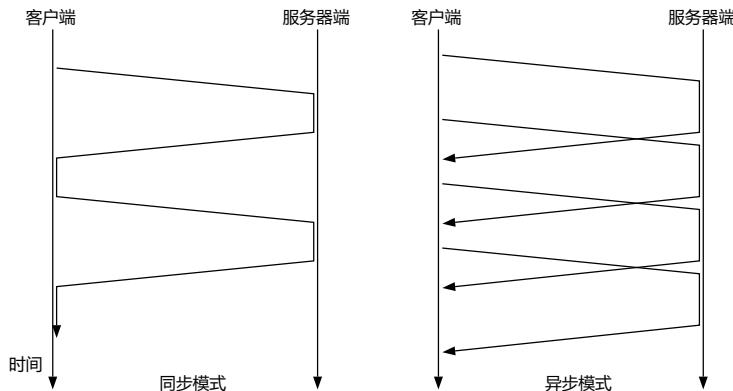


图 2.8：网络协议的同步和异步模式

上图中的左边表示同步模式，就是客户端向服务器发出一个请求后，就阻塞了，等待服务器的回复。等客

客户端接收到服务器的回复后才能发起第二个请求。我们知道线程是串行执行的，所以如果采用同步模式，一个线程只能模拟一个客户端的连接，如果想模拟 100 个用户连接就需要启动 100 个线程。上图右边表示异步模式，它和同步模式的区别很容易理解：就是客户端向服务器发出请求后，并不需要等待服务器的返回结果，而是可以继续发第二个，第三个请求。在异步模式下，一个线程可以模拟很多数据库的并发用户。理解了这一点，我们就容易理解 pgbench 中的线程和客户端数量这两个参数的关系，如图 2.9 所示：

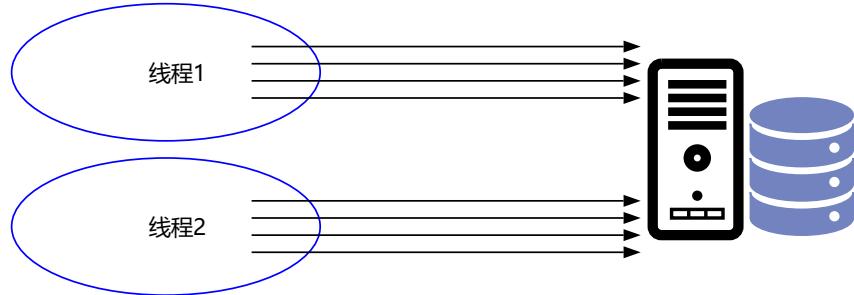


图 2.9：线程和数据库连接数的关系

假设我们在 pgbench 中指定线程数是 2，客户端数量是 8，则每个线程会分配 4 个客户端。如果你的测试机的 CPU 是 n 个核，你测试的时候就指定 n 个线程，这样每个 CPU 核可以运行一个线程，充分发挥测试机器的潜力。客户端的数量会在每个线程内要尽可能得均匀分配，所以你最好让客户端的数量是线程数的整数倍，这样每个线程分配的客户端数量是相同的。关于 pgbench 的各种用法，我们在使用的时候再介绍。

第三章 理解 WAL

提前写日志 WAL(Write Ahead Log) 是 PostgreSQL 数据库的核心概念之一，它也是 Oracle/SQL Server/MySQL 等其它关系型数据库的核心概念。对它的正确理解，是我们掌握 PostgreSQL 数据库备份和恢复，物理复制和逻辑复制等重要技术的基本前提。本章对 WAL 相关的知识进行讲解。

3.1 WAL 的基础知识

3.1.1 WAL 背后的思想

为了理解 WAL，我们先考察一下图 3.1 中所展示的一个极其简单的理论模型。

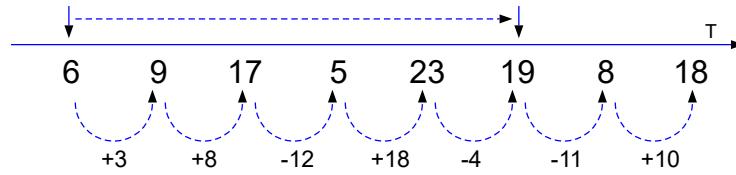


图 3.1: 一个简单的理论模型

假设有一个数字在不停的变化，且每次变化后都会立刻通知观察者去观察。如图所示，这个数字最初的值是 6，依次变化为 9, 17, 5, 23, 19 等等。为了记录这个数字的变化，很自然的方案就是：观察者接到通知后就把这个数字的当前值记录在笔记上，所以笔记里记录的是 6, 9, 17, 5 等原始值。除了这种自然的记录方案以外，观察者还可能有第二种记录方案：那就是首先记录一个起始值 6，在随后的观察活动中，观察者只记录增量。譬如当 6 变成 9 时，不再记录原始值 9，而是只记录增量 3。一旦有了一个初始值，再加上一个增量，观察者就可以通过简单的加法计算得到下一个值，我们称这个初始值为“基值”。只要一个基值 6 和连续的增量，即图中底部的 +3, +8, -12, +18 等数据，观察者就可以恢复出从基值开始变化的任何一种状态。从这个意义上说，增量“保护”了原始值。在此方案中还有一个很容易理解的规律：若要从基值 6 恢复到当前值，即第八个数字 18，我们需要进行 7 次加法运算。但当基值右移到新的基值 19 时，只需要两次加法运算，就可以恢复当前值，恢复的速度提高了。所以为了加速恢复当前值，我们时不时的要把基值的位置往右移动。

显然，这种基值加增量的方案在这个简单的理论模型中并没有什么优势。如果我们加上一个限制条件：在笔记上记录原始值的时间远大于记录增量的时间，很显然在这个前提下，第二种方案的整体记录速度要比只记录原始值的第一种方案更快。现在我们把注意力放在数据库设计领域。目前的存储技术有了很大的进步，SSD 硬盘越来越流行，断电不丢失的内存也已经出现了，但是量大价优的机械硬盘和量更大价格更优的磁带依然不会被淘汰。当前流行的关系型数据库都是在机械磁盘的时代被设计出来的。在机械磁盘时代有一个客观规律：对机械磁盘的顺序写 (sequential write) 的速度要远大于随机写 (random write) 的速度。如果记录原始值是随机写，记录增量是顺序写，写少数原始值加连续的增量，其速度明显比只纯写原始值要快得多，这就是 WAL 的思想起源。

我们知道，一个实际的数据库，其数据文件非常庞大，几百个 GB，几十个 TB 都是很普遍的。对于这么庞大的数据文件的写操作往往是随机的，东一榔头，西一棒槌。很显然，这种一旦数据块被修改了，就要直接写数据块本身的操作是非常低效的。如果把对海量数据块的随机写变换为某个固定文件的顺序写，即：不考虑被修改数据块的位置，只是机械地在某个固定文件的文件尾进行追加，毫无疑问，这种把直接写改成顺序写的模式会让数据库的运行速度大大地提高。

数据文件的数据块对应图 3.1 中的原始值（基值），WAL 是 WAL 记录的简称，它就是增量。把共享池中的内存页写入到磁盘中的数据块是随机写，WAL 记录被按时间的先后顺序追加到某个磁盘文件的尾部，是顺序写。为了提高恢复速度，我们把基值从 6 往右移动到 19，这个过程在数据库领域被称为检查点 (checkpoint)。WAL 和

检查点都是数据库领域的重要概念。理解了上述简单的理论模型后，下面我们来考察 PostgreSQL 如何实现基值加增量的存储方案。

3.1.2 WAL 概念的含义

当我们对表的数据进行修改时，包括增加、删除和修改等任何可以改变数据的操作，本质上就是修改内存中共享池里面的数据页。该数据页在被写回到对应的数据块之前，因为它和数据块的内容不一致了，因此被称为脏页 (dirty page)。内存中“脏”的数据被写回到磁盘文件的动作叫做“刷盘”或者“落盘”，落盘这个中文术语很优雅，让人想起了《琵琶行》中“大珠小珠落玉盘”的境界。由于数据页落盘是随机写，为了提高性能，共享池中某数据页变脏后，PostgreSQL 并不会立刻把它落盘，而是产生以某种“特殊”格式存储的记录来描述数据修改的细节，并将这些记录按照数据修改发生的先后顺序追加到某个磁盘文件的尾部。这些记录被设计成能够把曾经的数据修改精确地重现出来，我们把这些特殊格式的记录称为 WAL 记录，简称为 WAL，而用于保存 WAL 记录的磁盘文件则被称为 WAL 文件。图 3.2 展示了 WAL 记录的设计思想。

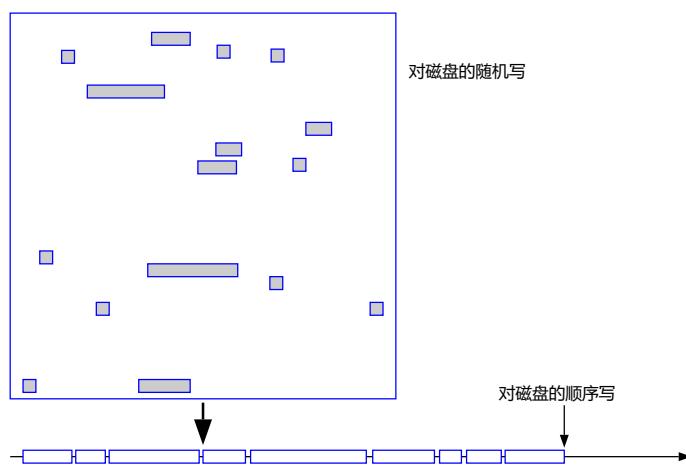


图 3.2: 把随机写变换为顺序写

在上图中，最大的矩形表示所有的数据文件的集合，里面的小矩形表示对某些数据块的修改。每个数据块都有一个位置信息，即这个数据块是哪个数据文件的第几块，我们可以形象地用二维坐标 (x,y) 来表示。数据库对于数据文件的写行为是不可预测的，这取决于来自用户的请求，所以我们不知道它什么时候发生，也不知道在哪个位置发生。当你往 (x,y) 位置写一个数据块时，我们能够达到相同记录效果的信息顺序地追加到一个固定的文件尾部，这个文件就是 WAL 文件。这些信息，即 WAL 记录，当然也包括 (x,y) 这样的位置信息，让我们知道修改了哪些数据块。这种把随机写等价变换为顺序写的设计，毫无疑问会大大提高写的速度。只要数据被可靠地记录在 WAL 文件中，就意味着数据不会丢失，因为我们可以拿磁盘上的数据块，再加上这些 WAL 记录的“修正”，就可以把一个数据块恢复到它目前状态后面的任何一个时间点的状态。WAL 文件可以认为是无限大的，但是在工程实现上，我们把它分割成统一大小的比较小的文件，给这些文件按顺序编号。WAL 文件还有一个尾部指针，始终指向文件的最后的位置，新来的记录就在这个位置追加即可。为了进一步提高写入的速度，我们还可以把 WAL 文件存放在一个速度更快的磁盘上，如 SSD 硬盘，而海量的数据文件可以放在速度较慢的机械硬盘上。

数据库里任何数据的修改操作都包含在一个事务中。事务的最终结局有两个，要么被提交 (commit)，即该事务所包含的修改动作都成功了，要么被回滚 (rollback)，即所有的修改动作都被撤销了。这种不成功则成仁 (all-or-nothing) 的特点被称为事务的原子性。本书有专门的章节来讨论数据库中的事务。当某个事务被提交后，由它产生的 WAL 记录会被以顺序写的方式保存在磁盘中。在 WAL 记录被可靠地写入磁盘后的某一个时间点，与之对应的脏页才会落盘。也就是说，WAL 记录被“提前”落盘了，这就是 write ahead 的名字来由。由此看来，修改的数据会被写两次：第一次是把保护数据页的 WAL 记录写入磁盘，第二次才是把真正的数据页落盘。对数据

的修改操作才会产生对应的 WAL 记录，查询是只读操作，一般不会产生 WAL 记录，但是也有例外情况，以后我们会看到。当拿到一个数据块和对应的 WAL 记录后，我们就可以把该数据块恢复到下一个变化后的新状态，这类似理论模型中的基值和增量的加法运算。这种用 WAL 记录来“修正”数据块的行为被称为回放 (replay)。

正是存储设备的随机写和顺序写的速度差异，才导致了几乎所有的数据库软件都有 WAL 的概念和设计：在 Oracle 中 WAL 被叫做 redo，在 SQL Server 中 WAL 被叫做事务日志 (transaction log)。如果未来存储设备的顺序写和随机写没有任何速度上的差异，则数据库的体系架构就会发生重大的变革。现在已经有人在研究稍后写 (write-behind log) 的概念和技术可行性，有兴趣的读者可以自行研究。但在看得见的未来，WAL 的思想和技术实现是不会被推翻的，依然是各种数据库技术的基石。

3.1.3 日志顺序号

日志顺序号 (LSN: log sequence number) 是 PostgreSQL 领域中非常重要的底层概念，正确理解 LSN 是掌握 WAL 和检查点，以及随之而来的备份、恢复和数据库复制等技术的前提。在 SQL Server 和 Oracle 中有同样的 LSN 概念，SQL Server 也使用 LSN 这个术语，在 Oracle 数据库中，LSN 被叫做系统改变号 (SCN: system change number)。从本质上来说，PostgreSQL 的 LSN 是一个 8 字节的无符号整数，表示一个有序的空间 (space)。学习过线性代数的读者对空间这个概念并不陌生，在数学上它指的是一个集合。8 字节的 LSN 可以表示 2^{64} 个数字。现在我们设想有一个长长的数组，从左到右，每个字节都有一个编号，第一个字节的编号是 0，最后一个字节的编号是 0xFFFFFFFFFFFFFF。字节的编号也叫该字节相对于第一个字节的偏移量，数组中第一个字节的偏移量是 0。随着数据库中数据修改活动的持续进行，描述这些修改的 WAL 记录会不断产生。如果把这些 WAL 记录按照时间的先后顺序，从左到右依次存储在这个巨大的数组中，则这个数组被称为 WAL 空间，如图 3.3 所示，其中每个小的正方形表示一个字节。

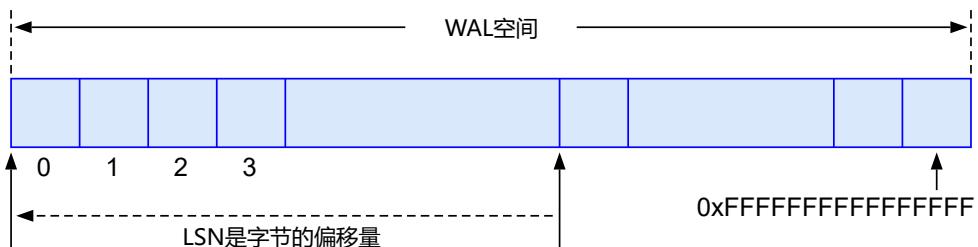


图 3.3: LSN 和 WAL 空间的关系

有了这个模型，LSN 的定义就非常简单了：WAL 空间中每个字节的编号被称为该字节的 LSN，这就是 LSN 的定义！假设某个字节的编号是 716751593320，就是十六进制的 0xA6E1B95F68，为了方便阅读，我们把这个 8 个字节分成高低各 4 个字节，中间用斜线分割，A6E1B95F68 可以表示为 A6/E1/B95/F68，这个就是该字节的 LSN。WAL 空间能够表示 16EB，就是 16777216 个 TB。虽然这个空间是有限的，但是因为其非常非常巨大，在我们的有生之年可以认为它是无限的。假设一个数据库一天产生 10TB 的 WAL 记录，这已经是数据修改活动非常非常频繁的数据库了。为了耗完 WAL 空间，它需要 $16EB / 10 TB = 1677721$ 天，就是 4596 年，所以在我们有限的生命时间范围内完全可以认为 WAL 空间是无限大的。

WAL 记录是由多个字节组成的，我们把 WAL 记录的第一个字节的 LSN 被称为该 WAL 记录的 LSN，表示该 WAL 记录在 WAL 空间的位置。虽然一条 WAL 记录的每个字节都有唯一的 LSN，但除了第一个字节以外的其余字节的 LSN 并没有什么意义，被称为无效的 LSN，只有指向 WAL 记录第一个字节的 LSN 才被称为有效的 LSN，请参考图 3.4。

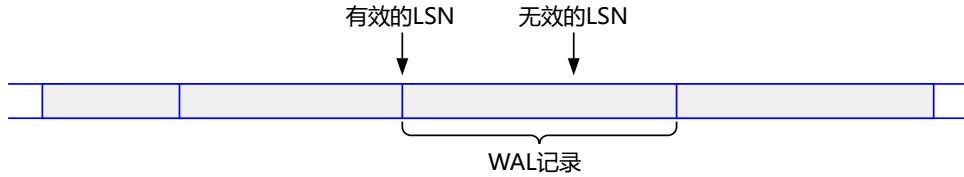


图 3.4: 有效的 LSN 和无效的 LSN

随着时间的流逝, WAL 记录在不断积累, 且按从左到右的顺序存储在 WAL 空间中, 所以 LSN 是一个代表时间的概念: 如果 LSN_2 大于 LSN_1 , 则表明 LSN_2 所代表的修改活动在 LSN_1 代表的修改活动发生之后才发生。你可以想象有一个指针, 指向准备写入下一条 WAL 记录的位置, 这个指针被称为“当前 WAL 指针”, 如图 3.5 所示。随着不断有 WAL 记录的写入, 这个当前 WAL 指针不断向后移动。

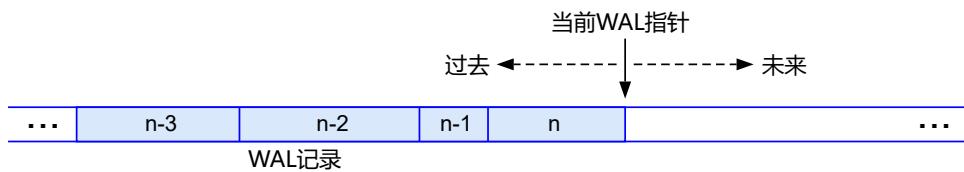


图 3.5: 当前 WAL 指针

相较英文, 中文是一种更加表意的语言, 同一个字, 在不同的语境下有不同的含义。譬如“史前时代”中的“前”表示更古老的历史, “前程似锦”中的“前”则有未来的意思。为了避免混淆, 本书规定: 过去/历史是“前”, 未来是“后”。当前 WAL 指针向后移动, 就是向未来方向移动的意思, 即图 3.5 中的向右移动。PostgreSQL 提供了几个函数, 返回当前 WAL 指针的位置:

```
postgres=# SELECT pg_current_wal_lsn(), pg_current_wal_insert_lsn();
 pg_current_wal_lsn | pg_current_wal_insert_lsn
-----+-----
 0/193ECF0      | 0/1940A10
(1 row)
```

在数据库中数据修改不频繁的情况下, `pg_current_wal_lsn()` 函数和 `pg_current_wal_insert_lsn()` 函数返回的 LSN 是相同的。至于它们之间的区别, 目前不需要深究。一般我们可以把 `pg_current_wal_lsn()` 理解为当前 WAL 指针。LSN 是数据库最底层的概念, 它是我们即将建立的 PostgreSQL 知识体系大厦的基石, 所以请读者务必仔细阅读和思考, 正确理解 LSN 所代表的含义, 才会在后面的学习过程中不犯迷糊。

3.1.4 WAL 文件

理论上, 整个 WAL 空间就是一个文件, 它的体积是 16EB。在目前的技术条件下, 任何存储设备都无法单独存储这么大的文件。为了真正存储 WAL 记录, 可以采用类似数据文件的分割思想, 把这个巨大的虚拟文件分割成很多个小文件。由于 LSN 是 8 个字节组成, 很自然地被分为高 4 字节和低 4 字节两部分, 所以我们可以把这个虚拟文件分割成体积为 4GB 字节的“小”文件, 这些小文件的数量一共有 4G 个, 它们被称为逻辑 WAL 文件 (logical file)。逻辑 WAL 文件的编号从 0 到 0xFFFFFFFF, 即 LSN 的高 4 个字节。4GB 字节大小的逻辑 WAL 文件通常情况下依然显得比较大, 可以进一步把它划分成 16MB, 32MB, 或 1GB 大小的更小的文件。这种更小的文件才是在磁盘上真正存储 WAL 记录的物理文件, 被称为“段”(segment)文件, 以后我们就用“WAL 文件”这个术语来指代 WAL 的段文件。逻辑 WAL 文件和段 WAL 文件之间的关系可用图 3.6 来表示。



图 3.6: 逻辑 WAL 文件和段 WAL 文件

WAL 文件的最小尺寸是 1MB，最大尺寸是 1GB。缺省情况下，每个 WAL 文件的体积是 16MB，则一个逻辑 WAL 文件可以分为 $256 (= 4\text{GB} / 16\text{MB})$ 个段，其编号 (segment id) 从 0 到 255。如果 WAL 文件的体积是 32M，则一个逻辑 WAL 文件可以分为 128 个段，其编号从 0 到 127，以此类推。一个数据库集群的 WAL 文件的大小必须在该数据库集群被创建之前指定。数据库集群被创建后，其 WAL 文件的大小是不能再改变的。创建数据库集群的工具 initdb 中有一个选项可以指定被创建的数据库集群的 WAL 文件的大小。

```
$ initdb --help | grep segsize
--wal-segsize=SIZE      size of WAL segments, in megabytes
```

由上可知，通过--wal-segsize 的选项就可以指定该数据库集群中 WAL 文件的大小。WAL 文件的尺寸设置为多大更合适呢？这个话题是性能调优的内容，我们暂时不考虑。如果不加以特别说明，本书后面的内容均按 WAL 文件的缺省值 16MB 作为讨论的前提。在源码中，有三个常量定义了 WAL 文件的缺省大小和最大最小值，请参考下面的定义：

```
/* in src/include/pg_config_manual.h */
#define DEFAULT_XLOG_SEG_SIZE  (16*1024*1024)
/* in src/include/access/xlog_internal.h */
#define WalSegMinSize 1024 * 1024      /* WAL文件体积的最小值是1MB */
#define WalSegMaxSize 1024 * 1024 * 1024 /* WAL文件体积的最大值是1GB */
```

3.1.4.1 WAL 文件的命名规则

在数据库集群运行期间产生的 WAL 文件被统一保存在数据库集群目录下的 pg_wal 子目录中，你可以到这个目录下一探究竟：

```
$ ls -l $PGDATA/pg_wal
total 32772
-rw----- 1 postgres postgres 16777216 Oct  1 13:19 000000010000000000000001 /* <- WAL文件 */
-rw----- 1 postgres postgres 16777216 Oct  1 13:19 000000010000000000000002 /* <- WAL文件 */
drwx----- 2 postgres postgres     4096 Oct  1 12:35 archive_status
```

从上面的输出可以看到，WAL 文件的文件名非常有规律，它实际上是由 12 个字节组成，由于每个字节可以用两位十六进制数字来表示，所以 WAL 文件的文件名恒定为 24 个字符，且分为三部分，如图 3.7 所示。



图 3.7: WAL 文件的文件名的组成部分

WAL 文件的文件名的三部分的具体含义如下：

- 高 8 个字符表示时间线。目前我们还没有时间线的概念，暂时可以理解高 8 个字符是 00000001。

- 中间 8 个字符表示 LSN 的高 4 个字节，即逻辑 WAL 文件的编号。
- 低 8 个字符表示 WAL 段文件的编号。

逻辑 WAL 文件一共有 4G 个，所以它的编号从 0 到 0xFFFFFFFF，这是中间 8 个字符的变化范围。WAL 段文件的大小为 16MB 时，一个逻辑 WAL 文件可以分为 256 个 WAL 段文件，所以段编号是 0 到 0xFF，这就意味着低 8 个字符只可能从 00000000 到 000000FF 之间变化，即它的高位的 6 个字符恒定是 0。同理，如果 WAL 段文件的大小为 256MB 时，一个逻辑 WAL 文件可以分为 16 个 WAL 段文件，所以段编号是 0 到 15，即低 8 个字符只可能从 00000000 到 0000000F 之间变化，即高位的 7 个字符恒定为 0。

在 WAL 段文件的体积为 16MB 的情况下，最大的 WAL 文件的名字为：FFFFFFFFFFFFFFF000000FF。给定了一个 WAL 文件的文件名，我们就可以推知它里面包含的 LSN 范围。反之，给定了一个 LSN 和时间线，也可以推知它所在的 WAL 文件的名字。以图 3.7 中的 WAL 文件为例，在这个文件中，最小的 LSN 是 2D/BE000000，最大的 LSN 是 2D/BFFFFF。假设一个 LSN 是 ABC/8EDCB00，且时间线是 7，则它对应的 WAL 文件是 000000070000ABC0000008E。注意：对于一个指定的 LSN，我们还需要知道它所在的时间线，才能够知道它所在的 WAL 文件的文件名。譬如一个 LSN 是 4CDEA/2D1EB2B0，它可能存在的 WAL 文件的文件名的规律是 XXXXXXXX0004CDEA0000002D，其中 X 是任何合法的十六进制字符。PostgreSQL 提供了函数 pg_walfile_name() 可以帮助你计算一个 LSN 所在的 WAL 文件的名字，具体演示如下：

```
postgres=# \! pg_controldata | grep TimeLineID /* 在控制文件中查看当前的时间线，其值为1 */
Latest checkpoint's TimeLineID:      1
Latest checkpoint's PrevTimeLineID:   1
/* 确定其值为76/7D000028的LSN所在的WAL文件的文件名 */
postgres=# SELECT pg_walfile_name('76/7D000028');
 pg_walfile_name
-----
 00000001000000760000007D
(1 row)
```

虽然有函数可帮助你把一个 LSN 对应的 WAL 文件的名字显示出来，我依然建议你用心算的方法来真正搞懂里面的细节问题。这些细节对于你理解后面更复杂的概念是有很大帮助的。

3.1.4.2 WAL 文件的内部结构

为了简化设计，PostgreSQL 把几乎所有的文件都按照 8KB 字节的大小划分成固定尺寸的数据块，WAL 文件也不例外，其基本结构可以用图 3.8 来表示。

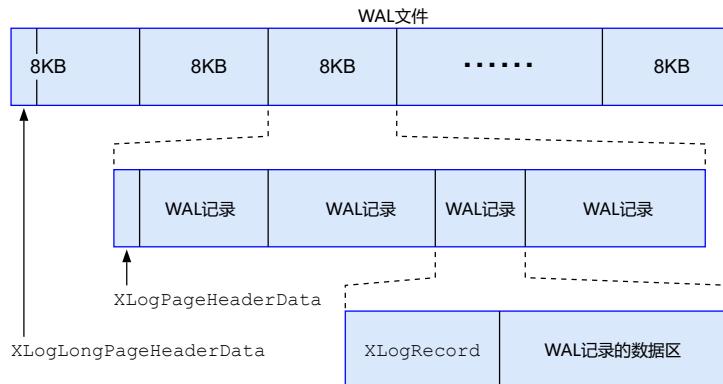


图 3.8: WAL 文件的内部结构

每个 WAL 文件和数据文件一样，也是按照固定尺寸划分成多个数据块，其尺寸由 XLOG_BLCKSZ 来决定，缺省也是 8KB，其具体定义如下：

```
/* in src/include/pg_config.h */
#define XLOG_BLCKSZ 8192
```

如果每个 WAL 文件是 16MB，则每个 WAL 文件中包含 2048 个数据块 (=16MB/8KB)。每个数据块的头部都有一个页头，是一个叫做 XLogPageHeaderData 的结构体。每个 WAL 文件的第一个数据块的页头稍微不同，叫做 XLogLongPageHeaderData。每个数据块中有若干个 WAL 记录，每个 WAL 记录又分为固定大小的通用记录头 XLogRecord 和真正的数据部分。注意：WAL 记录的尺寸可长可短，并不是固定的。结构体 XLogPageHeaderData 和 XLogLongPageHeaderData 的定义如下：

```
typedef uint32 TimeLineID; /* 4 bytes */
typedef uint64 XLogRecPtr; /* 8 bytes */
/* in src/include/access/xlog_internal.h */

typedef struct XLogPageHeaderData {
    uint16          xlp_magic;      /* magic value for correctness checks */
    uint16          xlp_info;       /* flag bits */
    TimeLineID      xlp_tli;        /* TimeLineID of first record on page */
    XLogRecPtr      xlp_pageaddr;   /* XLOG address of this page */
    uint32          xlp_rem_len;    /* total len of remaining data for record */
} XLogPageHeaderData;
typedef struct XLogLongPageHeaderData {
    XLogPageHeaderData std;         /* standard header fields */
    uint64          xlp_sysid;     /* system identifier from pg_control */
    uint32          xlp_seg_size;   /* just as a cross-check */
    uint32          xlp_xlog_blkksz; /* just as a cross-check */
} XLogLongPageHeaderData;
```

由上可知，XLogLongPageHeaderData 仅仅比 XLogPageHeaderData 多了三个成员变量，共计 16 个字节，其目的是为了校验 WAL 文件。从定义可以推知，XLogPageHeaderData 有 20 个字节，为了按 8 字节对齐，实际上后面还有 4 个补齐字节，共计 24 个字节，XLogLongPageHeaderData 则是 40 个字节。下面是一个 WAL 文件的头部的原始面貌：

```
$ hexdump -C -n 48 /opt/data/pgdata1/pg_wal/000000010000000000000000
00000000  13 D1 06 00 01 00 00 00  00 00 00 04 00 00 00 00 |.....
00000010  00 00 00 00 00 00 00 00  7B B1 56 13 0A BC 19 65 |.....{.V....E|
00000020  00 00 00 01 00 20 00 00  32 00 00 00 00 00 00 00 |.....2....|
00000030
```

头两个字节是 13 和 D1，翻转过来就是 D113，这就是源代码中定义的 XLOG_PAGE_MAGIC。

```
/* in src/include/access/xlog_internal.h */
#define XLOG_PAGE_MAGIC 0xD113
```

从第 5 到第 8 个字节代表时间线，其值为 1。xlp_sysid 是 0x6519BC0A1356B17B，你查看控制文件中的系统标识，会发现两者是一致的：

```
$ pg_controldata | grep system
Database system identifier: 7285060623708631419 /* = 0x6519BC0A1356B17B */
```

所以当你用一个来路不明的 WAL 文件去修复数据库集群时，PostgreSQL 就会根据系统标志来判断该 WAL 文件是否属于这个数据库集群。从上面的实验结果还可以看出 `xlp_seg_size=0x1000000`, 即 16MB, 表示 WAL 文件的大小是 16MB, 而 `xlp_xlog_blksize=0x2000`, 即 8KB, 表示 WAL 文件的每一个块都是 8KB。

WAL 记录要保存各种修改行为的信息，其格式是非常复杂的，分很多种类型，但记录头 XLogRecord 是通用的，和 WAL 记录的类型无关，其尺寸也固定不变，为 24 个字节。它的数据结构定义如下：

```
typedef uint32 TransactionId; /* 4 bytes */
typedef uint64 XLogRecPtr;    /* 8 bytes */
typedef uint8 RmgrId;         /* 1 byte */
typedef uint32 pg_crc32c;     /* 4 bytes */
/* in src/include/access/xlogrecord.h */
typedef struct XLogRecord {
    uint32          xl_tot_len; /* 4 bytes */
    TransactionId   xl_xid;    /* xact id, 4 bytes */
    XLogRecPtr      xl_prev;   /* 8 bytes */
    uint8           xl_info;   /* flag bits, 1 byte */
    RmgrId          xl_rmid;   /* resource manager for this record, 1 byte */
    /* 2 bytes of padding here, initialize to zero */
    pg_crc32c       xl_crc;    /* 4 bytes */
} XLogRecord;
```

从数据结构的定义来看，XLogRecord 有 22 个字节，由于对齐的原因，它实际上占了 24 个字节，在 `xl_rmid` 和 `xl_crc` 之间有两个补齐字节。XLogRecord 的成员变量的含义介绍如下：

- `xl_tot_len`：表示本 WAL 记录的大小，包括记录头 XLogRecord 和后面的数据。
- `xl_xid`：表示产生该 WAL 记录的事务的事务号 (XID)。
- `xl_prev`：指向前一条 WAL 记录的 LSN。
- `xl_info`：一些标志位，目前略过它的含义。
- `xl_rmid`：资源管理器的编号。
- `xl_crc`：CRC 校验码，用于校验本 WAL 记录是否损坏了。

图 3.9 展示了 XLogRecord 的基本结构。

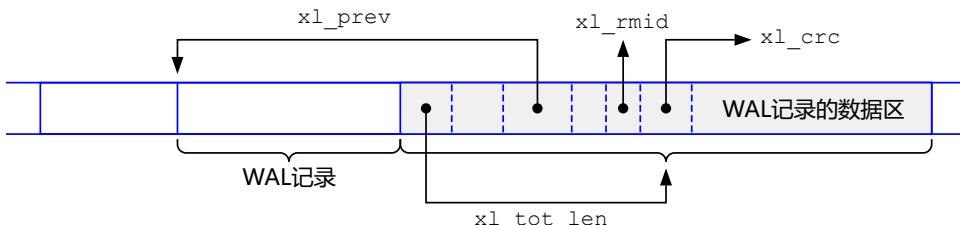


图 3.9: XLogRecord 的基本结构

WAL 记录按照“资源”管理的不同被划分成不同的类型，譬如有的 WAL 记录保存对表本身的修改信息，有的 WAL 记录保存对索引的修改信息，这里面的表，索引就是不同的资源。不同的索引类型，譬如 B-树索引，哈希索引，BRIN 索引，就是不同的资源。PostgreSQL 针对每一种资源都有对应的资源标识，被称为 RmgrId。成员变量 `xl_rmid` 记录资源的种类。源代码文件 `rmgrlist.h` 列出了已经支持的资源管理的类型，目前我们不必深究这么细节的东西，只要知道 PostgreSQL 按修改资源或者对象的不同，把 WAL 记录划分成不同的种类即可。

```
/* in src/include/access/rmgrlist.h */
/* symbol name, textual name, redo, desc, identify, startup, cleanup */
PG_Rmgr(RM_XLOG_ID, "XLOG", xlog_redo, xlog_desc, xlog_identify, NULL, NULL, NULL, xlog_decode)
```

```

PG_RMGR(RM_XACT_ID, "Transaction",xact_redo,xact_desc,xact_identify,NULL,NULL,NULL,xact_decode)
.....
/* in src/include/access/rmgr.h */
typedef enum RmgrIds {
#include "access/rmgrlist.h"
    RM_NEXT_ID
} RmgrIds;

```

当我们拿到了一个 WAL 记录的 LSN，即该 WAL 记录的第一个字节的 LSN，从前往后读 WAL 记录时，根据 xl_tot_len 的信息，我们很容易计算下一个 WAL 记录的 LSN。当从后往前读 WAL 记录时，需要 xl_prev 指针的指引，形成了一个由后往前（从未来到过去）的单向链表结构，如图 3.10 所示。

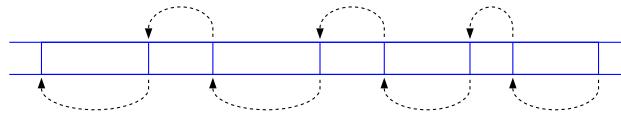


图 3.10: WAL 记录的单向链表

因为 WAL 记录本身包含了可以定位前后 WAL 记录位置的信息，所以我们可以非常方便地在 WAL 文件中前后移动来搜索我们感兴趣的 WAL 记录。下面我们研究两款帮助我们阅读 WAL 记录的有用的工具。这些工具在我们日常进行深入的故障诊断时往往能够发挥不可替代的作用。

3.1.5 分析 WAL 记录的工具

因为 WAL 记录的复杂性。本章并不对其进行深入剖析，只是考察几个较为简单的 WAL 记录的细节，让我们对 WAL 记录有一定的感性认识。在介绍 WAL 记录的分析工具之前，我们需要学习一下事务的基本知识。事务最基本的特性之一就是原子性 (all-or-nothing)，最经典的说明事务原子性的例子是银行转账，请看下面的例子。

```

oracle=# CREATE TABLE account(act_id INT, name VARCHAR(16), balance NUMERIC);
oracle=# INSERT INTO account VALUES(1, 'Alice', 84),(2, 'Bob', 264);
INSERT 0 2
oracle=# select * from account order by act_id;
 act_id | name   | balance
-----+-----+
 1 | Alice |      84
 2 | Bob   |     264
(2 rows)
oracle=# \! cat /tmp/t.sql /* 显示一下要执行的脚本的内容，注意BEGIN后面有一个分号 */
BEGIN;
    update account set balance = balance - 8 where act_id = 1;
    update account set balance = balance + 8 where act_id = 2;
COMMIT;
oracle=# \i /tmp/t.sql /* 使用\i命令执行该SQL脚本文件 */
BEGIN
UPDATE 1
UPDATE 1
COMMIT
oracle=# SELECT * FROM account ORDER BY act_id;
 act_id | name   | balance
-----+-----+

```

```

1 | Alice |      76
2 | Bob   |     272
(2 rows)

```

当 Alice 向 Bob 汇款时，实际上在账目表中执行了两个修改 (UPDATE) 操作，这两个修改操作被 BEGIN 和 COMMIT 包围，形成一个整体，被称为事务。一个事务要么全部成功，要么全部失败，才符合银行的业务要求，这就是所谓的“原子性”。在数据库集群范围内，每个事务都有唯一的编号，叫做 XID。一个事务可能涉及到表中多条记录的修改，所以可能会产生多条 WAL 记录，如果两条 WAL 记录中的 XID 相同，则表示它们属于同一个事务。数据库中可能存在大量并发的事务，它们都在不停地产生 WAL 记录，而这些 WAL 记录又都添加到 WAL 文件的尾部，在一维的空间中按时间先后顺序排列，所以一个事务的多条 WAL 记录在 WAL 文件中可能并不是连续的，它们中间可能会夹杂着其它事务的 WAL 记录。但根据 WAL 记录中的 XID 就可以把这些非连续 WAL 记录串连成一个完整的事务，请参考图 3.11。

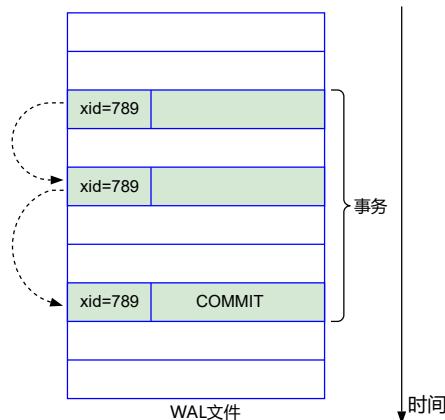


图 3.11: 事务和 WAL 记录的关系

在以前的演示中，插入 (INSERT) 操作并没有使用 BEGIN 和 COMMIT，这是因为在缺省情况下，psql 会设置为“自动提交模式”，即插入、修改和删除等单行命令被隐含地包围在一对 BEGIN 和 COMMIT 当中了。命令 echo 可以显示当前的事务提交模式，也可以按照下面的操作打开或者关闭自动提交模式。在自动提交模式被关闭后，事务就需要被包含在 BEGIN 和 COMMIT 当中了。

```

postgres=# \echo :AUTOCOMMIT /* 显示当前的事务模式的状态 */
on
postgres=# \set AUTOCOMMIT off /* 关闭自动提交模式 */
postgres=# \echo :AUTOCOMMIT
off
postgres=# \set AUTOCOMMIT on /* 打开自动提交模式 */
postgres=# \echo :AUTOCOMMIT
on

```

有了以上知识的铺垫，下面我们用例子来介绍 pg_waldump 和 pg_walinspect 这两款 WAL 记录分析工具是如何使用的。

3.1.5.1 pg_waldump

工具 pg_waldump 是在 \$PGHOME/bin 目录下的一个可执行程序，它可以解析指定 WAL 文件里的 WAL 记录，输出在屏幕上。通常情况下，使用者需要指定一个开始的 LSN 和一个结束的 LSN，该工具会在指定的目录下自动判别需要读取哪个 WAL 文件。下面的实验演示了该工具的具体操作过程。

```
/* 做一下准备工作：创建一张表，并插入一条记录 */
oracle=# create table walt(id char(2) not null, name char(6) not null);
CREATE TABLE
oracle=# insert into walt(id, name) values('TX', 'Dallas');
INSERT 0 1
oracle=# select pg_current_wal_lsn(); /* 记录下一个事务开始之前的LSN */
pg_current_wal_lsn
-----
0/803DEB0
(1 row)

/* 发起一个事务，插入一条记录 */
oracle=# insert into walt(id, name) values('MA', 'Boston');
INSERT 0 1
oracle=# select pg_current_wal_lsn(); /* 记录一下事务结束后的LSN */
pg_current_wal_lsn
-----
0/803DF20
(1 row)
```

上面的实验往一个表里插入了一条记录，在自动提交模式下，该插入操作就是一个完整的事务。有了该事务开始前和结束后的 LSN，你就可以用 pg_waldump 把该事务产生的所有的 WAL 记录都打印出来，执行的命令和输出结果如下。

```
$ pg_waldump -p $PGDATA/pg_wal -s 0/803DEB0 -e 0/803DF20
rmgr: Heap      len (rec/tot):    65/    65, tx:        803, lsn: 0/0803DEB0, prev 0/0803DE88,
desc: INSERT off 2 flags 0x00, blkref #0: rel 1663/16384/16530 blk 0
rmgr: Transaction len (rec/tot):    34/    34, tx:        803, lsn: 0/0803DEF8, prev 0/0803DEB0,
desc: COMMIT 2023-03-30 10:11:26.301425 MDT
```

在 pg_waldump 的选项中，-p 指定从哪个目录中寻找对应的 WAL 文件，-s 表示开始的 LSN，-e 表示终止的 LSN。上面的实验一共输出了 2 条 WAL 记录，第一条是插入操作，第二条是提交 (COMMIT) 操作。这两条记录的 tx 都是 803，这个就是事务的 XID。因为测试表上没有任何索引，所以一条插入命令只会产生一条插入类型的 WAL 记录和一条提交类型的 WAL 记录。上述输出中的 len(rec/tot) 中的 tot 是 total 的意思，即 WAL 记录的总长度，也就是 XLogRecord 结构体中的 xl_tot_len。rec 这一列我们这里不深究它的含义。这两者在很多 WAL 记录中都是相等的，但对于下一节要介绍的全页写 (Full Page Write, FPW) 的 WAL 记录，就有所不同了。一般情况下我们只关心 WAL 记录的总长度 (tot)。另外一个需要注意的信息是“rel 1663/16384/16530”，这个就是表 walt 对应的 Oid，下面的实验可以验证这个事实。

```
/* walt表在缺省表空间base的目录下 */
oracle=# select pg_relation_filepath('walt');
pg_relation_filepath
-----
base/16384/16530
(1 row)

/* 缺省表空间base的Oid是1663 */
oracle=# select oid, spcname from pg_tablespace where spcname='pg_default';
oid | spcname
-----+-----
1663 | pg_default
```

(1 row)

我喜欢把这两条 WAL 记录原始的面目扒出来看看，这样更加有真实感。为了使用 hexdump 解析某条 WAL 记录，首先需要确定该 WAL 记录存放在哪个 WAL 文件中，利用该 WAL 记录的 LSN 就可以很容易知道。LSN 为 0/0803DEB0 和 0/0803DEF8 的 WAL 记录都在 000000010000000000000008 文件中，因为插入操作的 LSN 是 0/0803DE88，所以它的 WAL 记录在这个文件中的偏移量是 0x3DE88，即 253616，下面是该插入操作的 WAL 记录的原始输出结果。

```
$ hexdump -C 000000010000000000000008 -n 65 -s 253616
0003deb0 41 00 00 00 23 03 00 00 88 DE 03 08 00 00 00 00 |A...#....|
0003dec0 00 0A 00 00 24 DB F6 C4 00 20 10 00 7F 06 00 00 |....$.... |
0003ded0 00 40 00 00 92 40 00 00 00 00 00 FF 03 02 00 |..@...@....|
0003dee0 02 08 18 00 07 4D 41 0F 42 6F 73 74 6F 6E 02 00 |.....MA.Boston..|
0003def0 00
|.|
```

任何 WAL 记录开始的头 24 个字节是 XLogRecord 结构体。对比其定义和上面的输出，可以发现头 4 个字节是 xl_tot_len，其值为 0x41，即 65，表示本 WAL 记录总长是 65 个字节。紧接着的 4 个字节是引发该操作所属的事务的 XID，其值为 0x323，即 803。再紧接着的 8 个字节 (“88 DE 03 08 00 00 00 00”) 是前一个 WAL 记录的 LSN，其值是 0/0803DE88。再后面的两个字节是 xl_info(=0x00) 和 xl_rmid(=0x0A)。跳过两个空白字节后就是 4 个字节的校验码 (“24 DB F6 C4”)，校验码有效地保证了 WAL 记录的完整性，所以如果一个 WAL 记录可以被回放，就说明它是完好无损的。图 3.12 展示了该条 WAL 记录的具体组成部分，其中的数字代表了相应的结构体的长度，单位是字节。

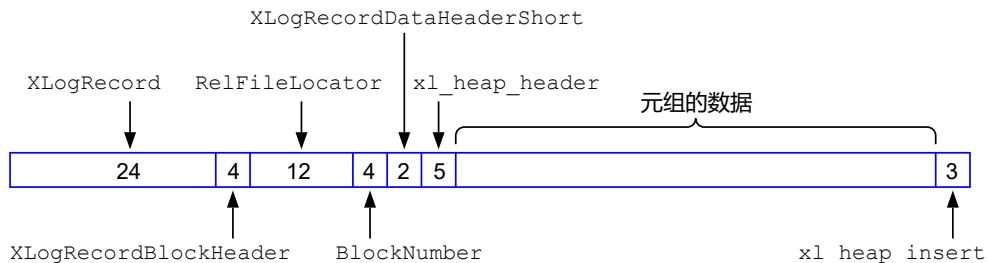


图 3.12: 简单的插入操作的 WAL 记录的组成

图 3.12 中的相关结构体的定义如下：

```
/* in src/include/access/xlogrecord.h */
typedef struct XLogRecordBlockHeader {
    uint8 id;           /* block reference ID */
    uint8 fork_flags;   /* fork within the relation, and flags */
    uint16 data_length; /* number of payload bytes (not including page image) */
} XLogRecordBlockHeader;
typedef unsigned int Oid;
typedef Oid RelFileName;
/* in src/include/storage/relfilelocator.h */
typedef struct RelFileLocator {
    Oid spcOid;        /* tablespace */
    Oid dbOid;         /* database */
    RelFileName relNumber; /* relation */
} RelFileLocator;
```

```

/* in src/include/storage/block.h */
typedef uint32 BlockNumber
/* in src/include/access/xlogrecord.h */
#define XLR_BLOCK_ID_DATA_SHORT      255
typedef struct XLogRecordDataHeaderShort {
    uint8  id;          /* XLR_BLOCK_ID_DATA_SHORT */
    uint8  data_length; /* number of payload bytes */
} XLogRecordDataHeaderShort;
/* in src/include/access/heapam_xlog.h */
typedef struct xl_heap_header {
    uint16      t_infomask2;
    uint16      t_infomask;
    uint8       t_hoff;
} xl_heap_header;
/* This is what we need to know about insert */
typedef struct xl_heap_insert {
    OffsetNumber offnum; /* inserted tuple's offset */
    uint8       flags;
} xl_heap_insert;

```

在结构体 XLogRecord 之后是 XLogRecordBlockHeader 结构体，共 4 个字节 ("00 20 10 00")，具体含义目前还不清楚，忽略之。在其之后，是连续的 16 个字节，代表 RelFileNode 结构体和 BlockNumber，表示该条插入操作是在 1663 号表空间 ("7F 06 00 00") 中的 16384 号数据库 ("00 40 00 00") 中的 16530 号表 ("92 40 00 00") 中的 0 号块 ("00 00 00 00") 中发生的，其含义是非常清楚的。

紧接着的 2 个字节 ("FF 03") 是 XLogRecordDataHeaderShort 结构体。相邻的 5 个字节 ("02 00 02 08 18") 是 xl_heap_header 结构体，该结构体之后就是被插入的记录的真正数据了 ("00 07 4D 41 0F 42 6F 73 74 6F 6E")，在 hexdump 输出结果中，右边的输出栏中显示了'MA', 'Boston' 的字符串，证明该内容就是被插入的具体数据。最后 3 个字节 ("02 00 00") 是 xl_heap_insert 结构体。下面再稍微研究一下 xl_heap_header 和 xl_heap_insert 这两个结构体。我们执行如下操作：

```

oracle=# select lp,t_infomask2,t_infomask,t_hoff,t_data from heap_page_items(get_raw_page('walt',0));
lp | t_infomask2 | t_infomask | t_hoff |      t_data
----+-----+-----+-----+
 1 |        2 |     2050 |      24 | \x0754580f44616c6c6173
 2 |        2 |     2050 |      24 | \x074d410f426f73746f6e
(2 rows)

```

因为上面分析 WAL 记录是针对 walt 表中的第二条记录，所以我们只关注 lp=2 的记录。如果把上面输出的十进制转换成十六进制，你会发现 t_infomask2, t_infomask 和 t_hoff 和结构体 xl_heap_header 中的值完全吻合。结构体 xl_heap_insert 的头两个字节的值 0x02，表示该插入操作的记录被插入到对应的数据块中的第二条记录的位置。至此，我们把一条简单的插入操作对应的 WAL 记录一个字节不落地观摩了一遍。可以想象：有了这些信息，在数据库恢复过程中，就可以利用该 WAL 记录对数据块进行修复，完全可以精准重复插入操作的全部过程。这种修复过程被叫做 WAL 记录的“回放”(replay)，就是重做(redo)的意思，本章第三节中会讨论 WAL 记录的回放流程。下面我们再研究一下提交类型的 WAL 记录，它的格式比较简单，可以用图 3.13 表示。

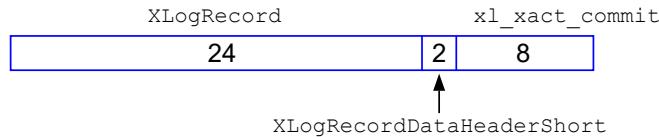


图 3.13: 提交类型的 WAL 记录的组成

由上图可知，提交类型的 WAL 记录一共 34 个字节 ($=24+2+8$)，共分为三部分，记录头，中间两个字节，和最后的数据部分，其中头两部分的数据结构内容在前文已经详细分析过了，最后的 xl_xact_commit 结构体的定义如下，它异常简单，仅仅是 8 个字节而已，代表着该 WAL 记录发生时的时间戳。

```
/* in src/include/access/xact.h */
typedef int64 TimestampTz;
typedef struct xl_xact_commit {
    TimestampTz xact_time;      /* time of commit */
} xl_xact_commit;
```

提交类型的 WAL 记录对应的 LSN 是 0/0803DEF8，其偏移量是 0x3DEF8，即 253688，原始结果输出如下：

```
$ hexdump -C 00000001000000000000000008 -n 34 -s 253688
0003def8 22 00 00 00 23 03 00 00 B0 DE 03 08 00 00 00 00 |"....#.....|
0003df08 00 01 00 00 1F 1D 71 1D FF 08 F1 C0 00 1D 1F 9B |.....q.....|
0003df18 02 00                                         |..|
```

由上可知，头 4 个字节的值是 0x22，即 34，表示该条 WAL 记录总长 34 个字节。再后面的 4 个字节对应 xl_xid=0x0323，即事务号是 803。再后面的 8 个字节 ("B0 DE 03 08 00 00 00 00") 是 xl_prev(LSN=0/0803DEB0)，xl_rmid=0x01，中间的"FF 08"两个字节是 XLogRecordDataHeaderShort 的内容，其成员变量 id 的值为 0xFF，成员变量 data_length 的值是 0x08，表示其后真正的数据区有 8 个字节 ("F1 C0 00 1D 1F 9B 02 00")，即 0x29B1F1D00C0F1。这 8 个字节的数据表示该事务被提交时的时间戳，就是 pg_waldump 中输出的"2023-03-30 10:11:26.301425 MDT"。如果你有兴趣，可以进一步研究一下 PostgreSQL 中时间记录的格式和转换。LSN 的本意是 WAL 空间的某一个字节的编号，通过提交类型的 WAL 记录中的时间戳信息，LSN 就切切实实地和时间的概念挂上钩了。下一章会讲解基于时间点的恢复 (PITR)，在 PITR 的过程中需要指定恢复目标 (recovery target)，恢复目标的类型可是时间，也可以是 LSN，这两者本质上是一回事，其联系的纽带就是提交类型的 WAL 记录中的时间戳。

我们也可以使用事务的 XID 来显示某个事务包含的所有 WAL 记录。PostgreSQL 数据库中的每条记录都有 xmin 隐藏列，保存着插入这条记录的事务的 XID，下面的操作展示了如何获得一条记录的 xmin 隐藏列的信息。

```
oracle=# select xmin, xmax, id, name from walt;
xmin | xmax | id | name
-----+-----+-----+
 802 |     0 | TX | Dallas
 803 |     0 | MA | Boston
(2 rows)

/* 拿到了xmin，就可以用-x参数把该事务所有的WAL记录都显示出来 */
$ pg_waldump -p $PGDATA/pg_wal -s 0/803DEB0 -e 0/803DF20 -x 803
rmgr: Heap      len (rec/tot): 65/65, tx:803, lsn: 0/0803DEB0, prev 0/0803DE88, desc: INSERT ...
rmgr: Transaction len (rec/tot): 34/34, tx:803, lsn: 0/0803DEF8, prev 0/0803DEB0, desc: COMMIT ...
```

分析到这里，相信你已经看出已经提交的事务的一个规律，那就是前面有若干个插入/修改/删除等操作对应的 WAL 记录，最后一条记录肯定是提交类型的 WAL 记录，类似：INSERT, UPDATE, DELETE,, COMMIT 的模式。你可以使用 pg_waldump --help 显示这个工具更多的参数信息，在这里面稍微提一下-f 参数。这个参数

类似 tail -f 的功能。由于 WAL 记录在不断增加，pg_waldump -f 可以不断显示正在更新的 WAL 记录的内容而不退出，也是一个非常好用的参数。

3.1.5.2 pg_walinspect

另外一个分析工具 pg_walinspect 是一位印度小哥开发的插件(国货当自强啊!)，功能和 pg_waldump 差不多。目前，在云端使用 PostgreSQL 数据库的情况越来越多，基于数据安全的考虑，很多云平台不提供 ssh 直接登录云端的数据库服务器的功能。譬如，AWS 的 RDS 只能使用类似 psql 的数据库客户端软件访问数据库，不提供 shell 接口。在此种情景下，用户就无法使用 pg_waldump 了。开发一个具备 SQL 查询接口的工具在数据库客户端软件中运行就显得尤为必要，pg_walinspect 就应运而生了。使用 SQL 接口的另外一个好处是：pg_walinspect 提供的结果是表的形式，可以很容易通过 SQL 语言和其它系统视图进行联合查询，从而构造出更加复杂的查询功能。原产印度的 pg_walinspect 的安装和使用都很简单，下面是它的安装和使用的示例：

```
/* 安装pg_walinspect插件 */
oracle=# CREATE EXTENSION pg_walinspect;
CREATE EXTENSION
oracle=# SELECT oid, extname, extversion FROM pg_extension;
   oid  |    extname     | extversion
-----+-----+-----
 12755 | plpgsql      | 1.0
 16461 | pg_walinspect | 1.0
(2 rows)

/* 查看LSN=0/0803DEB0的WAL记录的详细内容 */
oracle=# select * from pg_get_wal_record_info('0/0803DEB0');
-[ RECORD 1 ]-----+
start_lsn      | 0/803DEB0
end_lsn        | 0/803DEF8
prev_lsn       | 0/803DE88
xid            | 803
resource_manager | Heap
record_type     | INSERT
record_length   | 65
main_data_length | 3
fpi_length     | 0
description     | off 2 flags 0x00
block_ref       | blkref #0: rel 1663/16384/16530 fork main blk 0
```

插件 pg_walinspect 有几个函数，都非常容易理解，请自行在官方文档中查阅。我建议读者可以用 pg_waldump 和 pg_walinspect 两种工具分析同一批 WAL 记录，然后分析输出结果的异同，就能更容易地掌握这两种工具的使用。这两个工具是我们探究 PostgreSQL 背后的技术秘密的有力武器，希望大家能够掌握它们的基本使用。

3.2 检查点

检查点是数据库的重要概念，它是我们掌握备份恢复和流复制技术的前提概念。在本章开始讨论的理论模型中我们可以看到：为了恢复当前值 18，如果只有基值 6，就需要做七次加法运算。如果把基值后移到 19，则只需要做两次加法运算。为了加快对当前值的恢复，需要把基值后移，这就是检查点概念的基本内容，即：检查点的作用是为了减少未来数据库恢复所需要的时间。我们可以参考图 3.14 来理解检查点：

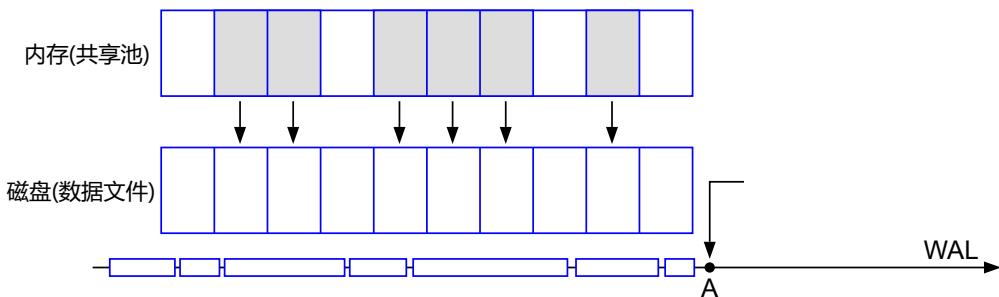


图 3.14: 检查点的基本含义

上图中灰色的矩形代表在内存中的脏页，需要写入到磁盘上。如果我们在 A 点下达一个命令，让共享池中全部的脏页都写入到磁盘中，这就是检查点操作最主要的工作内容。假设检查点操作可以瞬间完成，那么当本次检查点成功执行后，A 点左边的 WAL 记录就不需要了。这是因为 A 点之前的 WAL 记录保存着内存中脏页和磁盘上对应数据块的不同，现在在 A 点发生的检查点操作瞬间完成后，所有的脏页都没有了，内存中的数据页和磁盘上的数据块里面的内容完全一致，A 点左边的这些 WAL 记录当然也就不再需要了，这是非常容易理解的一件事情。所以检查点最大的特点是：在一次检查点成功执行后，该检查点前面的所有 WAL 记录对于正在运行的数据库没有用了。不过瞬间把所有的脏页都写入磁盘太理想化了，现实中不存在。实际上，需要把内存中所有的脏页都写回到磁盘中的工作是非常耗时的。举个例子，假设共享池配置为 32GB 大小，其中 10% 的数据页是脏页，检查点就需要把 3.2G 的数据共 40 多万个数据页 ($= 32\text{GB}/8\text{KB}/10$) 写回到磁盘。这当然需要一定的时间，想象一下把一个 3GB 的文件从 C 盘拷贝到 D 盘需要多长时间你就明白了。所以检查点操作不可能瞬间完成，它必然存在一个起点和终点。图 3.15 展示了检查点操作的起点和终点的概念。



图 3.15: 检查点的起点和终点的概念

假设在 A 点执行一个检查点，我们立刻记录此时的 LSN，作为检查点操作的起点。当检查点操作完成后，再插入一条检查点类型的 WAL 记录，它的位置可能已经在 A 点右边很远的地方了，这是因为在脏页落盘的过程中可能又会产生大量的 WAL 记录。A 点是检查点的起点，也是检查点 WAL 记录本来应该待的“逻辑”位置。检查点 WAL 记录物理所在的 B 点则是检查点的终点。检查点的 WAL 记录里有一个指针，记录了 A 点的 LSN，A 点被称为“重做点”(redo point)，这是一个非常重要的概念。当检查点成功完成后，假设数据库突然崩溃了，重新启动后要进行崩溃恢复，那么 A 点之前的 WAL 记录是崩溃恢复不需要的，但是 A 点和 B 点之间的 WAL 记录依然是需要的。重做点是数据库崩溃恢复的起点。检查点的 LSN 不重要，检查点 WAL 记录里面记录的重做

点的 LSN 才重要。这个请读者务必理解和记住。

3.2.1 检查点的执行过程

检查点的具体过程是由检查点进程 (checkpointer) 来执行的，我们很容易看到这个进程的身影：

```
$ ps -ef | grep postgres | grep checkpointer | grep -v grep
postgres 830 829 0 05:45 ? 00:00:00 postgres: checkpointer
```

图 3.16 来展示了一次检查点的执行过程，水平实线表示各种对象，包括检查点进程，共享池，WAL 缓冲池和各种磁盘文件等等，其含义在左边对应列出。水平虚线分割上下两部分，上部分表示内存，下部分表示磁盘。

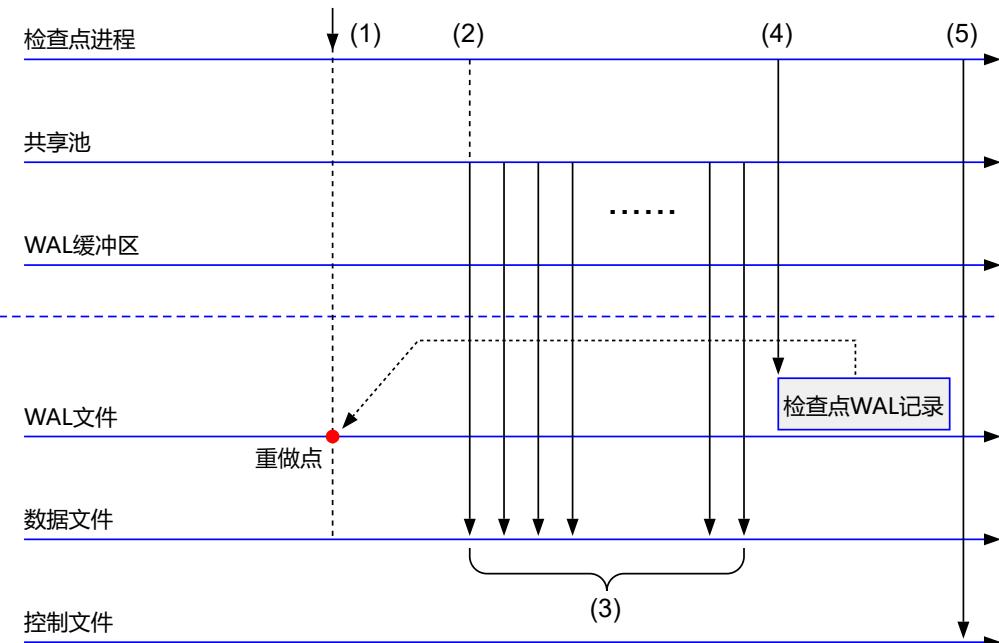


图 3.16: 检查点的执行过程

由上可知，一个检查点的执行总体上分为五个主要的动作：

- 步骤 1 是检查点发生的开始，此时检查点进程会首先记录当前的 WAL 指针的位置，这个位置就是检查点 WAL 记录的逻辑位置，即重做点。
- 步骤 2 开始在共享池中寻找脏页，依次把这些脏页写入到数据文件中，即脏页的落盘。
- 步骤 3 表示脏页落盘的过程。这个过程耗时最长，取决于有多少个脏页要落盘，持续几分钟甚至更长时间都不罕见。
- 步骤 4 的动作是往 WAL 文件中插入一条检查点的 WAL 记录。
- 步骤 5 是把步骤 4 中的检查点信息写入到控制文件中，以备将来恢复时读取重做点，作为数据库恢复的起点。

函数 CreateCheckPoint() 是执行检查点的唯一函数，我们可以参考它的关键源代码来理解上述过程。

```
/* in src/backend/access/transam/xlog.c */
void CreateCheckPoint(int flags)
{
    ...
    /* Begin filling in the checkpoint WAL record */
```

```

MemSet(&checkPoint, 0, sizeof(checkPoint));
checkPoint.time = (pg_time_t) time(NULL); /* 记录检查点的开始时间 */

.....
curInsert = XLogBytePosToRecPtr(Insert->CurrBytePos);
...
checkPoint.redo = curInsert; /* 步骤(1) - 把当前WAL的位置LSN记录在redo中 */
...
/* 步骤(2)/(3) - CheckPointGuts函数是写脏页到数据文件中。它的执行时间最长 */
CheckPointGuts(checkPoint.redo, flags);
...
XLogBeginInsert(); /* 步骤(4) - 把CheckPoint WAL记录写入WAL文件中 */
XLogRegisterData((char *) (&checkPoint), sizeof(checkPoint));
recptr = XLogInsert(RM_XLOG_ID, shutdown ? XLOG_CHECKPOINT_SHUTDOWN : XLOG_CHECKPOINT_ONLINE);

XLogFlush(recptr);
...
UpdateControlFile(); /* 步骤(5) - 更新控制文件 */
...
}

```

上述 5 个步骤都成功执行后，该检查点才算成功完成的。为了恢复当前数据库，只需要重做点之后的 WAL 记录。爱思考的读者可能会问一个问题：既然检查点是耗时的操作，就存在执行失败的可能性。如果检查点执行失败了，可能脏页并没有百分百可靠地写入到磁盘上，在这种情况下，我们依然需要重做点之前的 WAL 记录吧？对于这个问题的解答如下：假设数据库突然因为掉电而崩溃了，在它重新启动的阶段，会从控制文件中获取重做点的，即图中的步骤 5 往控制文件中写入的重做点。既然你已经拿到了重做点，就证明步骤 5 已经执行成功了，这意味着步骤 2, 3 和 4 也执行成功了。即：所有的脏页都被可靠地写回数据文件中了，检查点的 WAL 记录也可靠地被写入到了 WAL 文件中了。简而言之，你能拿到一个重做点，就意味着这个重做点对应的检查点的执行过程是百分百成功的。数据库恢复是从某个重做点开始的，也可以笼统地说，数据库恢复是从某个检查点开始的，这种说法的真正意思就是指从该检查点对应的重做点开始恢复。图 3.17 展示了两种情况，第一种情况是正在执行的检查点还没有执行成功时，系统突然崩溃了，则控制文件中依然记录着前一次成功的检查点操作的信息，所以数据库会从前一次成功的检查点开始恢复，即 A 点。第二种情况是，本次检查点执行成功后，系统突然崩溃了，则可以从本次的检查点开始恢复，即 B 点。此种情景下，你也可以从 A 点开始恢复，但没有必要，因为 A 点到 B 点之间的 WAL 记录对于恢复当前数据库没有用处。

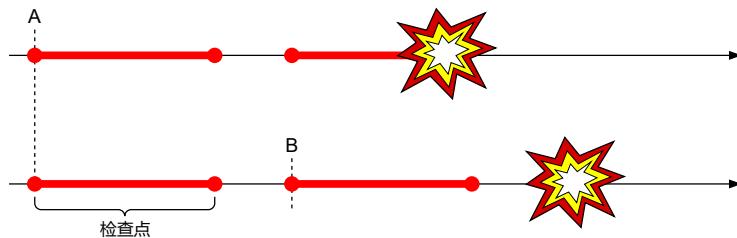


图 3.17: 不同时间点的崩溃

每次数据库实例启动时，都需要检查上一次关闭是否“干净”，如果上一次关闭不干净，则需要恢复数据文件，这个过程叫做崩溃恢复 (crash recovery)。如果需要做崩溃恢复，则 PostgreSQL 会查看控制文件里的检查点和重做点的信息，以此重做点作为崩溃恢复的起点，如图 3.18 所示。后面我们会看到，从控制文件中获得上一次成功的检查点的位置信息并不是唯一的选项，我们还可以把检查点的信息保存在别的地方，譬如一个叫做 backup_label 的文本文件中，后面我们会看到这个小小的但是非常重要的文本文件的作用。

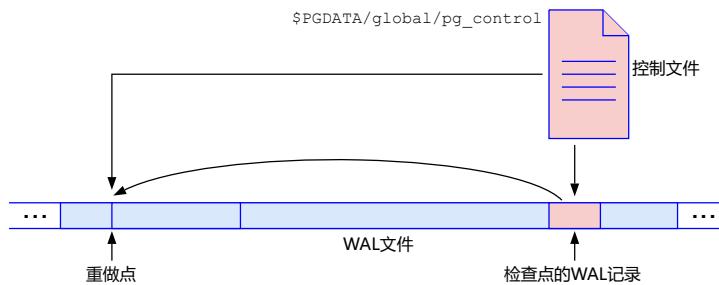


图 3.18: 从控制文件中获得重做点作为数据库恢复的起点

控制文件里面的检查点和重做点的信息可以用 pg_controldata 进行查看。

```
/* 控制文件里记录了最近一次成功的检查点和它相应的重做点 */
$ pg_controldata | grep checkpoint | grep location
Latest checkpoint location:          0/A0000AO
Latest checkpoint's REDO location:   0/A0000AO
```

在检查点执行过程中，步骤 3 是最耗时的，下面大致了解一下这个步骤到底做了什么，其源码如下：

```
/* in src/backend/access/transam/xlog.c */
static void CheckPointGuts(XLogRecPtr checkPointredo, int flags)
{
    CheckPointRelationMap();
    CheckPointReplicationSlots();
    CheckPointSnapBuild();
    CheckPointLogicalRewriteHeap();
    CheckPointReplicationOrigin();
    TRACE_POSTGRESQL_BUFFER_CHECKPOINT_START(flags);
    CheckpointStats.ckpt_write_t = GetCurrentTimestamp();
    CheckPointCLOG();
    CheckPointCommitTs();
    CheckPointSUBTRANS();
    CheckPointMultiXact();
    CheckPointPredicate();
    CheckPointBuffers(flags); /* <--把共享池中的脏页写到磁盘上 */
    TRACE_POSTGRESQL_BUFFER_CHECKPOINT_SYNC_START();
    CheckpointStats.ckpt_sync_t = GetCurrentTimestamp();
    ProcessSyncRequests();
    CheckpointStats.ckpt_sync_end_t = GetCurrentTimestamp();
    TRACE_POSTGRESQL_BUFFER_CHECKPOINT_DONE();
    CheckPointTwoPhase(checkPointredo);
}
```

可以看出，这个函数比较粗暴，就是依次把共享内存中的相关数据写入对应的文件中，其中最主要的步骤当然是 CheckPointBuffers()，即把共享池中的所有脏页刷盘。它只是简单地调用 BufferSync() 函数，下面是相关的源代码：

```
/* in src/backend/storage/buffer/bufmgr.c */
static void BufferSync(int flags)
{
```

```

int num_to_scan;
int mask = BM_DIRTY; /* 标志位, 表明该页是否为脏页 */

.....
num_to_scan = 0; /* 记录被发现的脏页个数 */
for (buf_id = 0; buf_id < NBuffers; buf_id++) { /* 扫描整个共享池 */
    BufferDesc *bufHdr = GetBufferDescriptor(buf_id);
    buf_state = LockBufHdr(bufHdr);
    if ((buf_state & mask) == mask) { /* 若某页是脏页, 则标记该页被列为刷盘的候选对象 */
        CkptSortItem *item;
        buf_state |= BM_CHECKPOINT_NEEDED;
        item = &CkptBufferIds[num_to_scan++]; /* 脏页的个数加一 */
        .....
    }
    .....
}
if (num_to_scan == 0) return; /* 如果没有发现脏页, 就立刻返回 */
.....
num_processed = 0;
num_written = 0;
while (!binaryheap_empty(ts_heap)) { /* 开始把上述扫描中发现的脏页写盘 */
    BufferDesc *bufHdr = NULL;
    CkptTsStatus *ts_stat = (CkptTsStatus *)
        DatumGetPointer(binaryheap_first(ts_heap));
    buf_id = CkptBufferIds[ts_stat->index].buf_id;
    bufHdr = GetBufferDescriptor(buf_id);
    num_processed++;
    if (pg_atomic_read_u32(&bufHdr->state) & BM_CHECKPOINT_NEEDED) {
        /* SyncOneBuffer()函数是把指定的页面写盘的具体执行人 */
        if (SyncOneBuffer(buf_id, false, &wb_context) & BUF_WRITTEN) {
            .....
            num_written++;
        }
    }
    .....
}
.....
}

```

当检查点执行完毕后,会在日志里记录相关信息,如下所示,里面清楚地记录了写了 7 个脏页 (wrote 7 buffers),占比多少等等信息。

```

2023-03-24 05:19:36.572 MDT [4818] LOG: checkpoint starting: immediate force wait
2023-03-24 05:19:36.577 MDT [4818] LOG: checkpoint complete: wrote 7 buffers (0.0%);
0 WAL file(s) added, 0 removed, 0 recycled; write=0.002 s, sync=0.001 s, total=0.005 s;
sync files=5, longest=0.001 s, average=0.001 s; distance=0 kB, estimate=0 kB

```

重做点是检查点的逻辑位置,只不过因为检查点操作是一个非常耗时的操作,所以它有起点和终点两个不同的位置。重做点就是检查点操作的起点,它的 LSN 当然小于或者等于该检查点的 WAL 记录的 LSN,我们可以简称为: 重做点小于等于检查点。如果重做点小于检查点,则表明在步骤 2 和 3 进行时,数据库中依然有活动的事务在修改数据。什么情况下重做点和检查点相等呢?有两种情况。第一种情况是检查点发生时,共享池中

没有脏页，函数 CheckPointGuts() 就立刻返回，导致检查点的 WAL 记录立即被写入到它的逻辑位置，就是重做点的位置。第二种情况发生在正常关闭数据库时。检查点分为两种类型，在线检查点 (onLine checkpoint) 和关闭型检查点 (shutdown checkpoint)，其相关定义如下：

```
/* in src/include/catalog/pg_control.h */
#define XLOG_CHECKPOINT_SHUTDOWN          0x00
#define XLOG_CHECKPOINT_ONLINE           0x10
```

顾名思义，在数据库运行过程中发生的普通检查点基本上都是在线类型的。数据库关闭时，关闭流程在最后会写入一个检查点，它的类型被叫做关闭型。因为此时已经不可能有数据被修改的情况发生了，故关闭型的检查点必定等于其重做点。下面我们做一个实验：

```
$ pg_ctl stop /* 干净地关闭数据库 */
waiting for server to shut down.... done
server stopped
$ pg_controldata | grep location | grep checkpoint /* 查看检查点信息 */
Latest checkpoint location:      0/18CB0D8    /* 两者是相等的 */
Latest checkpoint's REDO location: 0/18CB0D8    /* 两者是相等的 */
$ pg_waldump -p $PGDATA/pg_wal -s 0/18CB0D8 -n 1 /* 把这条WAL记录打印出来 */
rmgr: XLOG      len (rec/tot):   114/   114, tx:          0, lsn: 0/018CB0D8, prev 0/018CB0A0,
desc: CHECKPOINT_SHUTDOWN redo 0/18CB0D8; tli 1; prev tli 1; fpw true; xid 0:745; oid 16395;
multi 1; offset 0; oldest xid 722 in DB 1; oldest multi 1 in DB 1; oldest/newest commit
timestamp xid: 0/0; oldest running xid 0; shutdown
$
```

我们可以看到，当数据库被干净地关闭以后，其检查点的类型为 CHECKPOINT_SHUTDOWN，即关闭型检查点，它的检查点的 LSN 和重做点的 LSN 是完全一样的，都是 0/18CB0D8。根据这个规律，我们可以判断一个数据库是否是被干净地关闭掉的。因为重做点和检查点本质上是一回事，所以我们在后文中可以把这两个术语交叉使用。

3.2.2 检查点的 WAL 记录

理解了检查点的执行过程后，本节考察一下它的 WAL 记录的细节。其 WAL 记录的格式非常简单，和提交型的 WAL 记录格式差不多，也分为三个部分：记录头，数据，外加中间一个小头，如图 3.19 所示。

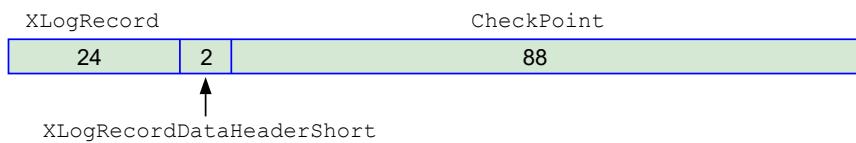


图 3.19: 检查点 WAL 记录的结构

上一节已经介绍了 XLogRecord 和 XLogRecordDataHeaderShort 的具体结构，下面我们只看一下数据部分的内容，它实际上是一个 CheckPoint 结构，其具体定义如下：

```
typedef uint64 XLogRecPtr;
typedef int64 pg_time_t;
/* in src/include/catalog/pg_control.h */
typedef struct CheckPoint {
    XLogRecPtr      redo;           /* RedoPoint */
    TimeLineID     ThisTimeLineID; /* current TLI */
```

```

TimeLineID      PrevTimeLineID;    /* previous TL */
bool            fullPageWrites;   /* current full_page_writes */
FullTransactionId nextXid;        /* next free transaction ID */
Oid              nextOid;          /* next free OID */
MultiXactId     nextMulti;        /* next free MultiXactId */
MultiXactOffset nextMultiOffset;  /* next free MultiXact offset */
TransactionId   oldestXid;       /* cluster-wide minimum datfrozenxid */
Oid              oldestXidDB;     /* database with minimum datfrozenxid */
MultiXactId     oldestMulti;     /* cluster-wide minimum datminmxid */
Oid              oldestMultiDB;   /* database with minimum datminmxid */
pg_time_t       time;            /* time stamp of checkpoint */
TransactionId   oldestCommitTsXid;
TransactionId   newestCommitTsXid;
TransactionId   oldestActiveXid;
} CheckPoint;

```

该结构体比较复杂，根据目前所学的知识，我们只关心三个成员变量：redo，即本检查点的重做点。fullPageWrites 表示是否是全页写，后面会介绍其具体含义。time，记录检查点发生的开始时间，可以参考上述 CreateCheckPoint() 函数。下面的实验演示了如何查看检查点 WAL 记录的原始内容。

```

postgres=# \! pg_controldata | grep checkpoint | grep location
Latest checkpoint location:          0/3000098
Latest checkpoint's REDO location:  0/3000060
postgres=# checkpoint;      /* 手工执行一个检查点 */
CHECKPOINT
/* 发现检查点和重做点的LSN均向后移动了 */
postgres=# \! pg_controldata | grep checkpoint | grep location
Latest checkpoint location:          0/3000180
Latest checkpoint's REDO location:  0/3000148
postgres=# \! pg_controldata | grep TimeLine /* 当前的时间线是1 */
Latest checkpoint's TimeLineID:      1
Latest checkpoint's PrevTimeLineID:  1

```

手工执行 CHECKPOINT 命令后，控制文件里的检查点和重做点的 LSN 都向后移动了：检查点从 0/3000098 移动到了 0/3000180，重做点从 0/3000060 移动到了 0/3000148。下面使用 pg_waldump 和 pg_walinspect 来查看一下该条 WAL 记录。

```

postgres=# \! pg_waldump -p $PGDATA/pg_wal -n 1 -s 0/3000180
rmgr: XLOG      len (rec/tot):    114/    114, tx:          0, lsn: 0/03000180, prev 0/03000148,
desc: CHECKPOINT_ONLINE redo 0/3000148; tli 1; prev tli 1; fpw true; xid 0:765; oid 24700;
multi 1; offset 0; oldest xid 716 in DB 1; oldest multi 1 in DB 1; oldest/newest commit
timestamp xid: 0/0; oldest running xid 765; online
postgres=# SELECT * FROM pg_get_wal_record_info('0/3000180');
-[ RECORD 1 ]-----+
start_lsn      | 0/3000180
end_lsn        | 0/30001F8
prev_lsn       | 0/3000148
xid            | 0
resource_manager | XLOG
record_type     | CHECKPOINT_ONLINE

```

```

record_length | 114
main_data_length | 88
fpi_length | 0
description | redo 0/3000148; tli 1; prev tli 1; fpw true; xid 0:765; oid 24700;
multi 1; offset 0; oldest xid 716 in DB 1; oldest multi 1 in DB 1; oldest/newest
commit timestamp xid: 0/0; oldest running xid 765; online
block_ref |

```

对比两个工具的输出结果，可以得到如下分析结果：

- 该 WAL 记录的总长度是 114 个字节，真正的记录长度是 88 个字节，因为 tot = 114, main_data_length=88。
- 该 WAL 记录的前一条 WAL 记录的 LSN 是 0/3000148，后一条是 0/30001F8。
- 该检查点的重做点是 0/3000148，因为有”redo 0/3000148”的输出。
- 该检查点是 CHECKPOINT_ONLINE 类型的。
- 数据库集群处于全页写模式 (fpw true)。

如果不甘心，我们还可以使用大杀器 hexdump 观察其原始的形态，结果如下：

```

postgres=# \! hexdump -C $PGDATA/pg_wal/000000010000000000000003 -s 384 -n 114
00000180  72 00 00 00 00 00 00 00 48 01 00 03 00 00 00 00 |r.....H.....|
00000190  10 00 00 00 DD EC D2 20 FF 58 48 01 00 03 00 00 |..... .XH....|
000001a0  00 00 01 00 00 00 01 00 00 00 01 00 00 00 00 00 |.....|
000001b0  00 00 FD 02 00 00 00 00 00 00 7C 60 00 00 01 00 |.....|`....|
000001c0  00 00 00 00 00 00 CC 02 00 00 01 00 00 00 01 00 |.....|
000001d0  00 00 01 00 00 00 00 00 00 00 FB 7C 1E 64 00 00 |.....|.d..|
000001e0  00 00 00 00 00 00 00 00 00 00 FD 02 00 00 00 00 |.....|
000001f0  00 00
000001f2

```

仔细对比上述的原始结果和 CheckPoint 的数据结构，很容易分析出 XLogRecord 的各成员变量的取值。xl_tot_len 的值是 114 (0x72)，表明整个 WAL 记录是 114 个字节 (24 + 2 + 88)。xl_xid 的值为 0，检查点不属于任何一个事务。xl_prev 的值是 0/3000148 ("48 01 00 03 00 00 00 00")，代表本 WAL 记录前面的 WAL 记录的 LSN。xl_info 的值是 0x10。RmgrId 的值是 0，表示类型是 RM_XLOG_ID。xl_crc 的值是 0x20D2ECDD ("DD EC D2 20")，表示校验码。XLogRecordDataHeaderShort 各成员变量的取值为：id 的值是 0xFF，表示这是一条短类型的记录，参考 XLR_BLOCK_ID_DATA_SHORT 的定义。data_length 的值是 0x58，表示其后的 CheckPoint 结构的总长度，共 88 个字节。

CheckPoint 各成员变量的取值是：redo 的值是 0/3000148，这就是本检查点 WAL 记录对应的重做点。ThisTimeLineID 的值为 1，表示当前时间线是 1。PrevTimeLineID 是 1，表示前一个时间线也是 1。时间线的最小值是 1，以后我们再讨论时间线的问题。fullPageWrites 的值是 1，表示数据库集群的全页写模式处于激活状态。time 是 8 个字节，表示本检查点发生开始的时间戳，里面的内容我们就不需要关心了。

3.2.3 检查点的执行时机

前面的实验都是以超级用户的身份手工执行 CHECKPOINT 命令。当然这种情况非常少见，你总不能搬个小板凳坐在数据库服务器面前，时不时手工执行一条 CHECKPOINT 命令吧。PostgreSQL 设置了一个时间间隔，由 checkpoint_timeout 参数规定，缺省值为 5 分钟，即每隔 5 分钟就会自动触发一次检查点。除此之外，触发检查点的条件还有如下几种情况：

- 每当 \$PGDATA/pg_wal 目录下的 WAL 文件的总体积超过一定大小时，也会触发一个检查点。这个体积由参数 max_wal_size 规定，缺省值是 1GB，即 pg_wal 目录下的 WAL 文件累积到 1GB 时自动触发检查点，导

致全部的 WAL 文件都可以被安全删除，缓解 pg_wal 目录中文件体积过大的压力。

- 某一些特定操作，如开始备份，数据库恢复结束后，关闭数据库集群的最后阶段，也会触发检查点。

由于检查点的操作非常耗时，所以有两种选择，一种是开足马力，尽快地把内存中的全部脏页刷到磁盘上，这种模式被称为“立即”模式或者“快速”模式。这种模式会导致系统的 I/O 激增，数据库的性能下降，影响用户的使用体验。另外一种方式是悠着点来，让检查点在 `checkpoint_timeout` 参数规定的时间内慢悠悠地写完。此种模式下，I/O 的负荷被均匀地分散在一个比较宽松的时间段内，所以对数据库的性能冲击变小了。参数 `checkpoint_completion_target` 规定了两次检查点之间的时间段的百分比，其缺省值是 0.9。例如，`checkpoint_timeout` 的值是 5 分钟，`checkpoint_completion_target` 是 0.9，则一次检查点必须在 4.5 分钟 ($= 5 * 0.9$) 内完成。这个参数的含义请参考图 3.20。

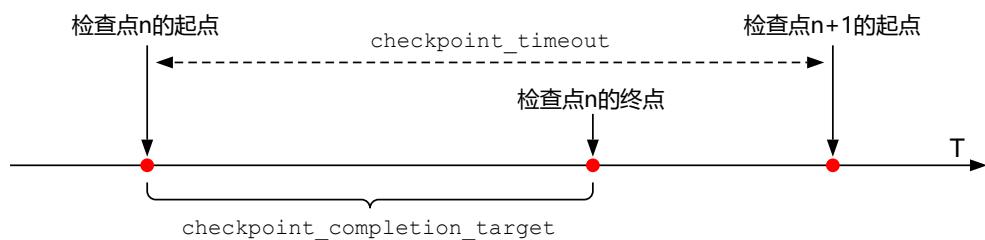


图 3.20: 检查点的执行时间

针对把所有的脏页刷盘的全检查点比较耗时的问题，Oracle 数据库和华为的 OpenGauss 数据库中提出了“增量检查点”的概念，这种技术是对全检查点的一种优化，有兴趣的读者可以自行研究这个概念和具体的技术细节。

3.2.4 全页写

很显然，为了恢复原始值，必须有基值和增量，两者缺一不可。没有基值，光有增量，也是无根之木，无源之水。但是有些情况下基值可能会丢失，全页写 (FPW: full page write) 技术就是为了解决数据块损坏的问题而设计的。它的基本思想是：在某次检查点发生之后，若某个数据页是第一次被修改，则对应的 WAL 记录保存的不是本次的修改信息，而是把整个数据页的数据都写入到 WAL 记录中，即：在 WAL 文件中不仅仅记录增量，也记录基值。这样一来，即使真正的数据块损坏了，它的副本在 WAL 文件中还能找到，这就有效地解决了数据块损坏的问题。参数 `full_page_writes` 控制 FPW 功能的开启和关闭，缺省值是打开状态 (on)。普通的 WAL 记录只有几百个字节，而 FPW 的 WAL 记录可能有几 K 字节。所以，如果 FPW 开启后检查点发生的很频繁，WAL 文件的体积会快速膨胀。对于 WAL 文件快速膨胀的问题，解决方法就是调整 `checkpoint_timeout` 和 `max_wal_size` 等参数，让检查点发生的不要那么频繁。譬如我公司生产库的检查点发生时间被控制在 30 分钟。此外，参数 `wal_compression` 可以把 FPW 类型的 WAL 记录压缩，在一定程度上缓解了 WAL 文件体积膨胀的问题。下面的实验展示了 FPW 的基本内容。

```
/* 检查一下两个参数 */
oracle=# \! cat $PGDATA/postgresql.conf | grep -E 'full_page_writes|wal_compression'
full_page_writes = on                      # recover from partial page writes
wal_compression = off                      # enables compression of full-page writes;
/* 做一些准备工作，准备一张表和两条记录 */
oracle=# CREATE TABLE stateus(id CHAR(2), name VARCHAR(64));
CREATE TABLE
oracle=# INSERT INTO stateus VALUES('TN', 'Tennessee'), ('MA', 'Massachusetts');
INSERT 0 2
```

```

oracle=# SELECT id, name FROM stateus ORDER BY id;
id |      name
---+-----
MA | Massachusetts
TN | Tennessee
(2 rows)

oracle=# CHECKPOINT; /* 手工执行一个检查点，确保后面的插入操作能够触发FPW */
CHECKPOINT

oracle=# SELECT pg_current_wal_lsn();
pg_current_wal_lsn
-----
0/1A9A5E8
(1 row)

oracle=# INSERT INTO stateus VALUES('WY', 'Wyoming'); /* 再插入一条记录 */
INSERT 0 1

oracle=# SELECT pg_current_wal_lsn();
pg_current_wal_lsn
-----
0/1A9A728
(1 row)

```

表 stateus 里面已经有了 2 条记录，当手工执行一个检查点后，紧接着再往此表中插入一条记录，肯定会产生一条 FPW 类型的 WAL 记录。下面是 pg_waldump 解析出的该条 WAL 记录的内容。

```

$ pg_waldump -s 0/1A9A5E8 -n 2
rmgr: Heap      len (rec/tot):      54/    218, tx:        762, lsn: 0/01A9A5E8, prev 0/01A9A5B0,
desc: INSERT off 3 flags 0x00, blkref #0: rel 1663/16384/16505 blk 0 FPW /* <-- FPW表示全页写 */
rmgr: Transaction len (rec/tot):      34/     34, tx:        762, lsn: 0/01A9A6C8, prev 0/01A9A5E8,
desc: COMMIT 2023-03-24 09:31:27.173624 MDT

```

LSN 为 0/01A9A5E8 的 WAL 记录最后显示了“FPW”的字样，表明这条 WAL 记录是 FPW 类型，FPW 类型的记录在源代码中被称为“备份块”(backup block)或者“全页镜像”(FPI: full-page image)。你还可以观察到长度信息(rec/tot)分别显示为 54 和 218，其中 218 表示该记录的总长度，54 表示记录头有 54 个字节，两者的差值就是数据页的全部数据的长度。FPI 分为压缩和未压缩两种形式，未压缩的 WAL 记录的格式可以参考图 3.21，其中的数字代表对应的结构的长度，单位是字节。

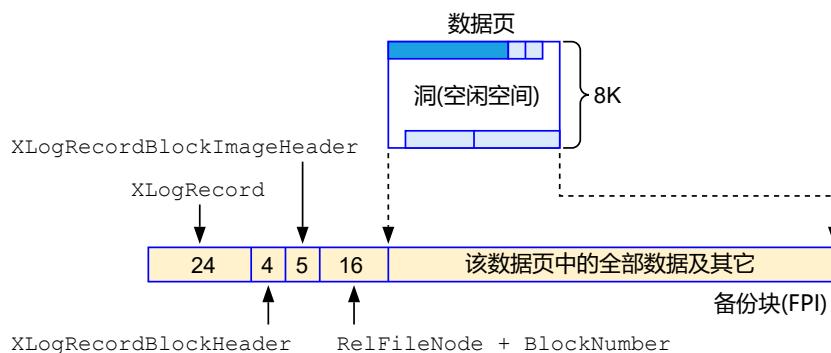


图 3.21：备份块/FPI 的基本结构

相关的数据结构的具体定义如下。在此我们不去深究每一个数据结构的具体含义，只要一个整体的概念即

可。FPI 中包含了一个数据块的全部内容，所以在未来做数据库恢复时，直接拿这些数据对数据块进行全覆盖即可，数据块本身的内容无关紧要，损坏了也没有关系。数据块中的空闲空间是一个“洞”(hole)，在 WAL 记录中存储时肯定不会存放这部分无效的信息，所以 `XLogRecordBlockImageHeader` 有一个成员变量 `hole_offset` 记录数据块中空闲空间的偏移量。

```
/* in src/include/access/xlogrecord.h */
typedef struct XLogRecordBlockHeader {
    uint8    id;           /* block reference ID */
    uint8    fork_flags;   /* fork within the relation, and flags */
    uint16   data_length;  /* number of payload bytes (not including page image) */
} XLogRecordBlockHeader;

typedef struct XLogRecordBlockImageHeader {
    uint16       length;    /* number of page image bytes */
    uint16       hole_offset; /* number of bytes before "hole" */
    uint8        bimg_info; /* flag bits, see below */
} XLogRecordBlockImageHeader;

typedef unsigned int Oid;

/* in src/include/storage/relfilenode.h */
typedef struct RelFileNode {
    Oid  spcNode; /* tablespace */
    Oid  dbNode;  /* database */
    Oid  relNode; /* relation */
} RelFileNode;

/* in src/include/storage/block.h */
typedef uint32 BlockNumber

/* 具体代码可参考 src/backend/access/transam/xloginsert.c : XLogRecordAssemble() */
```

下面的实验显示了该 FPI 记录的原始数据。因为 LSN 是 0/01A9A5E8，所以其偏移量为 0xA9A6C8，即 11118056，原始输出的结果如下：

```
$ hexdump -C 000000010000000000000001 -s 11118056 -n 218
00a9a5e8 DA 00 00 00 FA 02 00 00 B0 A5 A9 01 00 00 00 00 | .....|.
00a9a5f8 00 0A 00 00 6E A1 96 95 00 10 00 00 A4 00 24 00 |....n.....$.|.
00a9a608 03 7F 06 00 00 00 40 00 00 79 40 00 00 00 00 00 |.....@..y@.....|.
00a9a618 00 FF 03 00 00 00 00 A0 A4 A9 01 00 00 00 00 24 |.....|.....$|.
00a9a628 00 80 1F 00 20 04 20 00 00 00 00 D8 9F 4A 00 A8 |.....|.....J..|.
00a9a638 9F 52 00 80 9F 46 00 FA 02 00 00 00 00 00 00 00 |.R...F.....|.
00a9a648 00 00 00 00 00 00 00 03 00 02 00 02 08 18 00 07 |.....|.
00a9a658 57 59 11 57 79 6F 6D 69 6E 67 00 00 00 00 00 F9 |WY.Wyoming.....|.
00a9a668 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 |.....|.
00a9a678 00 02 00 02 09 18 00 07 4D 41 1D 4D 61 73 73 61 |.....MA.Massal|.
00a9a688 63 68 75 73 65 74 74 73 00 00 00 00 00 00 00 F9 |chusetts.....|.
00a9a698 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 |.....|.
00a9a6a8 00 02 00 02 09 18 00 07 54 4E 15 54 65 6E 6E 65 |.....TN.Tenne|.
00a9a6b8 73 73 65 65 00 00 00 03 00 00 |ssee.....|.
```

只要你足够耐心，对照上面的数据结构，不难从原始输出中找出对应的成员变量的值。虽然我们对数据页保存在 FPI 中的部分的具体格式不清楚，但从上述输出的右边信息栏中可以看到除了新插入的“Wyoming”，还有“Massachusetts” 和 “Tennessee”的字符串，单一的插入操作居然保存了整个数据块的三条记录，明显和前一节分析的插入操作的 WAL 记录不同，由此证明了整个数据页都被保存在 FPI 中了。所以，即使未来这个数据块损

坏了，依然可以通过这条 WAL 记录完整恢复它。工具 pg_waldump 有一个选项--save-fullpage，可以把一条 FPI 类型的 WAL 记录转化成一个单独的数据页，存放在指定的目录下，具体操作演示如下：

```
$ pg_waldump -s 0/01889348 -n 1 --save-fullpage=/opt/data/fpi
rmgr: Heap      len (rec/tot):    54/   166, tx:        736, lsn: 0/01889348, prev 0/01889310,
desc: INSERT off: 2, flags: 0x00, blkref #0: rel 1663/24576/24577 blk 0 FPW
$ cd fpi/
$ ls -l /* 可以看到被转储出来的数据页是8192个字节，它的文件名包含了数据块的编号等信息 */
total 8
-rw-rw-r-- 1 postgres postgres 8192 Dec 28 18:09 00000001-00000000-01889348.1663.24576.24577.0_main
$ hexdump -C 00000001-00000000-01889348.1663.24576.24577.0_main
00000000  00 00 00 00 98 91 88 01  00 00 00 00 20 00 b0 1f  |.....|.
00000010  00 20 04 20 00 00 00 00  d8 9f 48 00 b0 9f 42 00  |. . . . . H . . B .|.
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|.
*
00001fb0  e0 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|.
00001fc0  02 00 02 00 02 08 18 00  07 54 58 0d 54 65 78 61  |.....TX.Texa|.
00001fd0  73 00 00 00 00 00 00 00  df 02 00 00 00 00 00 00  |s.....|.
00001fe0  00 00 00 00 00 00 00 00  01 00 02 00 02 08 18 00  |.....|.
00001ff0  07 43 4f 13 43 6f 6c 6f  72 61 64 6f 00 00 00 00  |.CO.Colorado....|.
00002000
```

为了减少因为全页写而导致的 WAL 文件膨胀的问题，我们可以把 WAL 记录进行压缩。下面的实验展示了打开 WAL 压缩功能后的情形。

```
postgres=# ALTER SYSTEM SET wal_compression=on;
ALTER SYSTEM
postgres=# SELECT pg_reload_conf();
pg_reload_conf
-----
t
(1 row)
postgres=# SHOW wal_compression; /* WAL压缩功能已经打开，使用pglz压缩算法 */
wal_compression
-----
pglz
(1 row)
oracle=# CHECKPOINT; /* 再执行一个检查点 */
CHECKPOINT
oracle=# SELECT pg_current_wal_lsn();
pg_current_wal_lsn
-----
0/1A9B048
(1 row)
oracle=# INSERT INTO stateus VALUES('MD', 'MaryLand'); /* 插入新记录 */
INSERT 0 1
oracle=# \! pg_waldump -s 0/1A9B048 -n 3 /* 看看里面的结果 */
rmgr: Standby      len (rec/tot):    50/    50, tx:        0, lsn: 0/01A9B048, .....
rmgr: Heap      len (rec/tot):    56/   197, tx:        763, lsn: 0/01A9B080, prev 0/01A9B048,
desc: INSERT off 4 flags 0x00, blkref #0: rel 1663/16384/16505 blk 0 FPW
```

```

rmgr: Transaction len (rec/tot):      34/     34, tx:          763, lsn: 0/01A9B148, prev 0/01A9B080,
desc: COMMIT 2023-03-24 09:49:10.427555 MDT
/* 使用hexdump大杀器 */
oracle=# \! hexdump -C $PGDATA/pg_wal/00000001000000000000000000000001 -s 11120768 -n 197
00a9b080  C5 00 00 00 FB 02 00 00  48 B0 A9 01 00 00 00 00 |.....H.....|
00a9b090  00 0A 00 00 A1 36 25 7A  00 10 00 00 8D 00 28 00 |.....6%z.....(.)|
00a9b0a0  07 30 1F 7F 06 00 00 00  40 00 00 79 40 00 00 00 |.0.....@..y@...|
00a9b0b0  00 00 00 FF 03 00 00 00  00 00 C8 A6 A9 01 01 01 |.....|
00a9b0c0  08 28 00 58 1F 00 20 04  02 20 01 0C D8 9F 4A 00 |.(.X.. . . .J.|_
00a9b0d0  A8 9F 00 52 00 80 9F 46  00 58 9F 30 48 00 FB 02 |...R...F.X.OH...|
00a9b0e0  01 16 07 01 04 00 00 02  00 02 08 18 00 07 4D 00 |.....M.|_
00a9b0f0  44 13 4D 61 72 79 4C 61  54 6E 64 01 18 FA 0C 28 |D.MaryLaTnd....(|_
00a9b100  03 05 28 57 00 59 11 57  79 6F 6D 69 6E AA 67 02 |..(W.Y.Wyomin.g.|_
00a9b110  18 F9 0C 28 02 01 02 09  01 50 00 41 1D 4D 61 73 |...(. ....P.A.Mas|_
00a9b120  73 61 63 80 68 75 73 65  74 74 73 04 20 05 0D 30 |sac.husetts. ..0|_
00a9b130  01 05 30 54 4E 15 54 65  80 6E 6E 65 73 73 65 65 |..OTN.Te.nnessee|_
00a9b140  00 19 04 00 00                                     |.....|

```

上面的输出很清晰地显示：只有 3 条记录的 FPI 总长度是 218 个字节，而有 4 条记录的 FPI 的总长度只有 197 个字节，压缩的效果显现出来了，而且里面的内容已经相对不可读了。

Oracle 数据库拥有自己独立的文件系统 ASM，PostgreSQL 却要依赖操作系统的文件系统提供的基本磁盘读写功能，但文件系统并不能总让 PostgreSQL 满意，这导致 PostgreSQL 不得不做一些可靠性设计，FPW 就是一种可靠性设计。建议用户打开该功能，所以它的缺省值被设置为 on。

3.2.5 PostgreSQL 的写磁盘操作

设计 WAL 记录的目的是为了保护数据文件中的数据块，所以 WAL 记录被可靠地写入掉电不丢失数据的磁盘上，是数据不丢失的根本保证。PostgreSQL 本身并没有控制磁盘读写的功能，它要依赖操作系统的文件系统来完成最终的磁盘读写工作，在 Linux 平台上就是使用 open/read/write/close 等系统调用，在 Windows 平台上使用 WriteFile() 系统调用。下面是一个简单的磁盘文件写操作的例子：

```

$ cat writefile.c /* 显示源代码的内容 */
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
char data_buf[16] = {0};
int main(int argc, char* argv[])
{
    int fd = open("data.bin", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        printf("Cannot open file\n");
        return 1;
    }
    for(int i=0; i<16; i++) data_buf[i]=i+1;
    ssize_t bytes_written = write(fd, data_buf, 16);
    if (bytes_written == -1) {
        printf("Cannot write to file\n");
        close(fd);
    }
}

```

```

    return 2;
}
close(fd);
return 0;
}

$ gcc -Wall writefile.c -o w /* 把上述源码编译成一个可执行文件w*/
$ ./w /* 执行这个程序 */
$ ls -l
total 24
-rw----- 1 postgres postgres 16 Jan 6 16:28 data.bin /* 这是程序创建的文件 */
-rwxr-xr-x 1 postgres postgres 16136 Jan 6 16:28 w
-rw-r--r-- 1 postgres postgres 550 Jan 6 16:28 writefile.c
$ hexdump -C data.bin
00000000  01 02 03 04 05 06 07 08  09 0A 0B 0C 0D 0E 0F 10  |.....
00000010
$
```

上述源程序的逻辑很简单，就是打开一个文件 data.bin，往里面写入 16 个字节。这是任何在 Linux 平台下写文件的基本套路，它使用了 open() 系统调用打开文件，使用 write() 系统调用往文件中写入数据，最后使用 close() 系统调用关闭这个文件。PostgreSQL 也是使用这个模式往磁盘上写文件的。应用软件，操作系统和磁盘三者的关系可以可以用图 3.22 来表示。

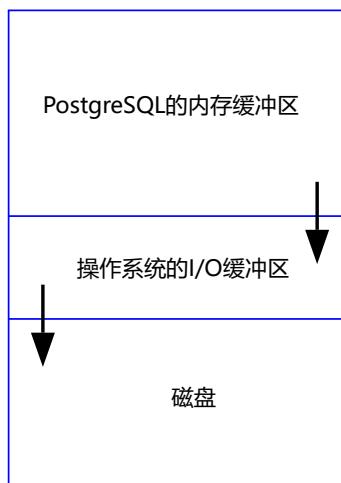


图 3.22: PostgreSQL 的写磁盘操作

当 PostgreSQL 通过 write() 系统调用把内存中的数据写入磁盘时，数据可能并没有真正的写入磁盘，而是保留在操作系统的 I/O 缓冲区内。操作系统认为合适的时机，再把数据真正地写入到磁盘中。操作系统的 I/O 缓冲区也是内存，掉电后也会消失。如果数据依然在内存中，并没有真正写到磁盘上，此时服务器突然掉电，数据就丢失了。为了真正地把数据写入磁盘，Linux 提供了 fsync() 和其它类似的函数来确保把内存中的数据刷到磁盘上。也就是说：write() 成功了并不等于真正可靠，必须 fsync() 才能可靠地把数据写入到磁盘。为此，PostgreSQL 提供了两个参数。第一个参数是 fsync，它是一个布尔变量，取值范围为 on 或者 off，表示是否打开或者关闭 fsync 功能。当然，如果 fsync=off，会带来性能上的提升，但是带来的风险就是可能会发生数据丢失，所以在真正的生产系统中，fsync 必须设置为 on。在 fsync 为 on 的情况下，第二个参数 wal_sync_method 可以控制 fsync 的类型，它的取值范围包括 open_datasync, fdatasync, fsync, fsync_writethrough, opne_sync 几种可能性。这些取值和数据文件所在的文件系统相关，有的文件系统支持某一种或者某几种类型。为了测试不同类型的性能，PostgreSQL 提

供了一个简单使用的测试工具 pg_test_fsync。它的使用方法非常简单，如下所示：

```
$ pg_test_fsync
5 seconds per test
_ DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
  open_datasync           4508.202 ops/sec    222 usecs/op
  fdatasync                4275.360 ops/sec    234 usecs/op
  fsync                     3640.199 ops/sec    275 usecs/op
  fsync_writethrough        n/a
  open_sync                 3772.740 ops/sec    265 usecs/op
  .....
Non-sync'ed 8kB writes:
  write                    448530.775 ops/sec    2 usecs/op
```

从上面的结果来看，如果 fsync 关闭，每次操作只要 2 微秒，在打开 fsync 的情况下，最快的 open_datasync 也需要 222 微秒。可见打开关闭 fsync 的性能差异很大，但是对于生产数据库，必须打开 fsync，牺牲性能换取数据的可靠性。正式因为有了 write 和 fsync 在文件系统中的区别，所以 PostgreSQL 的很多系统视图也提供了磁盘读写的 write 和 fsync 不同的指标。

3.2.6 监控 WAL 记录和检查点

对于 WAL 记录的产生，PostgreSQL 提供了一个系统视图 pg_stat_wal。这个系统视图可以帮助我们了解 WAL 记录的产生情况。我们可以查看一下这个视图中的信息：

```
oracle=# select * from pg_stat_wal;
-[ RECORD 1 ]-----+
wal_records      | 1284
wal_fpi          | 902
wal_bytes         | 4162675
wal_buffers_full | 1536
wal_write         | 1555
wal_sync          | 97
wal_write_time   | 0
wal_sync_time    | 0
stats_reset      | 2023-10-16 04:44:17.35114-06
```

关于这个系统视图的各列的含义，官方文档中有详细的说明。我们在这里把我们目前能够理解的指标的含义介绍一下。首先你可以注意最后一列，stats_reset。很多 PostgreSQL 的系统视图中的数据是累积性 (cumulative) 的，即这些数据是从某个时间点开始计算，随着时间的推移，数据的值是累积的，只增不减。stats_reset 这一列就记录了其它列的数据是从什么时候开始累积的。第一列 wal_records 的值是 1284，则表明从 2023 年 10 月 16 日开始，迄今为止，共计产生了 1284 条 WAL 记录。我们往往需要选取一个感兴趣的时间段，把开始和结束时刻采集的两个值进行相减，才能得到比较有意义的数据。譬如我们知道一天之内产生了多少条 WAL 记录，就可以在某个时刻查询一下该系统视图，过了 24 个小时以后再查询一下，所获得的两个值相减，就是一天之内产生的数据。列 wal_fpi 是共计产生了多少条全页写类型的 WAL 记录。列 wal_bytes 是共计产生了多少字节的 WAL 记录。以上几列都是非常容易理解的。wal_write_time 和 wal_sync_time 记录 WAL 数据写入磁盘所需要的时间。很显然，这两列的值如果很大，则表明磁盘速度有问题，或者某种因素阻碍了 WAL 记录的落盘。这两个值在数

据库整体性能调优方面是需要关注的指标。为了获取这两个值，需要打开 track_wal_io_timing 参数，具体细节请查阅官方文档。

我们可以通过系统函数 pg_stat_reset_shared() 来重置这张系统表的统计信息，具体操作如下：

```
oracle=# select pg_stat_reset_shared('wal');
pg_stat_reset_shared
-----
(1 row)

oracle=# select * from pg_stat_wal;
-[ RECORD 1 ]-----+
wal_records | 0
wal_fpi | 0
wal_bytes | 0
wal_buffers_full | 0
wal_write | 0
wal_sync | 0
wal_write_time | 0
wal_sync_time | 0
stats_reset | 2023-12-09 14:22:23.416003-07
```

我们可以看到，这张系统表的所有数据都被清零了，然后新的数据从 2023 年 12 月 9 日开始继续计数。另外一张有用的系统视图是 pg_stat_bgwriter，它可以显示检查点的某些信息。我们来看一下它的具体定义：

```
postgres=# \d pg_stat_bgwriter
          View "pg_catalog.pg_stat_bgwriter"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 checkpoints_timed | bigint | | | |
 checkpoints_req | bigint | | | |
 checkpoint_write_time | double precision | | | |
 checkpoint_sync_time | double precision | | | |
 buffers_checkpoint | bigint | | | |
 buffers_clean | bigint | | | |
 maxwritten_clean | bigint | | | |
 buffers_backend | bigint | | | |
 buffers_backend_fsync | bigint | | | |
 buffers_alloc | bigint | | | |
 stats_reset | timestamp with time zone | | | |
```

第一列 checkpoints_timed 的含义是已经执行的规划 (scheduled) 的检查点的次数。我们知道参数 checkpoint_timeout 规定了必须执行的检查点的时间间隔，超过了这个间隔必须执行一次检查点，这种检查点就是规划的检查点。第二列 checkpoints_req 表示被请求 (requested) 的检查点的执行次数。如果我们通过 CHECKPOINT 命令手工执行一次检查点，你就会发现这一列的值会增加一。因为手工执行的检查点并不是按照固定的节奏自动执行的，而是被请求的。在第四章讨论的数据库备份过程中，在备份开始时会执行一次检查点，这是另外一种类型的请求型检查点。第三列 checkpoint_write_time 表示所有已经发生的检查点 (包括第一列和第二列) 操作总共花费的写磁盘的时间，单位是毫秒。我们用这个值除于第一列和第二列之和，就可以得到平均每次检查点的操作时间。第四列 checkpoint_sync_time 是所有已经发生的检查点操作花费在磁盘同步上的时间。这个时间过大，则表明磁盘的写速度比较慢。第五列 buffers_checkpoint 表示所有检查点操作写入的数据页的数量。

类似的我们可以使用 `pg_stat_reset_shared('bgwriter')` 函数来把这个系统视图中的数据重置清零。从名字上看, `pg_stat_bgwriter` 是为后台进程 `bgwriter` 而设计的, 为什么里面有检查点的信息呢? 这是因为历史上 `bgwriter` 执行了检查点的工作, 后来检查点的功能分化出来, 设立了一个新的后台进程 `checkpointer`, 但是它的统计信息依然在 `bgwriter` 相关的系统视图中。在即将发布的 PG 17 中核心开发团队正在考虑为检查点设立一个单独的系统视图。

作为重要的基础性软件, 数据库的可观测性也是一种重要的特性, PostgreSQL 提供了越来越丰富的系统监控视图帮助用户深入理解数据库的运行状态。这些系统视图非常多, 本书不打算给大家列一张大而全的清单, 而是学习到哪块知识点, 就见缝插针地介绍一下相关的系统视图。这种安排可以让读者不知不觉中就熟悉了常用的系统视图。

3.3 崩溃恢复

数据库集群在运行过程中如果遇到突然断电的情况，内存中来不及写到磁盘上的脏页就丢失了。等数据库集群再次启动后，就需要使用从最近一次重做点开始的 WAL 记录来修复那些已经被修改，但是还来不及保存的数据块，这种修复叫做崩溃恢复。当主进程启动后，它会无条件启动一个后台进程，叫做恢复进程 (startup)。根据 startup 字面的意思应该翻译为“启动进程”，但是由于该进程的主要作用是回放 WAL 记录来恢复数据库，我觉得把它翻译成“恢复进程”更贴切。恢复进程会首先判断数据库是否需要恢复 (recovery)。如果数据库是正常关闭的，数据文件处于一致状态，则不需要恢复数据库，恢复进程就会默默地退出。如果数据库被粗暴地关闭，或者因为掉电而导致的突然崩溃，则数据库处于不一致的状态。所谓不一致的状态，就是磁盘上的某些数据块的修改被保存在了 WAL 记录中，但是还没有更新到数据块上。恢复进程就要修复这种不一致。本节我们来研究恢复进程所做的工作，进一步加深对 WAL 的认识。图 3.23 展示了崩溃恢复的基本思想。



图 3.23: 崩溃恢复的工作内容

如上图所示，当恢复进程启动后，会读取控制文件里面的最后一次检查点的信息，拿到重做点，然后就从这个重做点开始，依次扫描 pg_wal 目录下的 WAL 文件，把所有能得到的 WAL 记录按照先后次序回放到相应数据块上，直至所有的 WAL 记录都回放完毕。这就是崩溃恢复的主要工作内容。

3.3.1 WAL 记录的回放

WAL 记录是为了保护数据块而设计的，那么如果需要用 WAL 记录来修正数据块，真正的回放过程是如何的呢？从第二章我们已经知道，在数据块的页头 PageHeaderData 结构中有一个 8 字节的 pd_lsn，它记录着最近一次对本数据块修改的操作对应的 WAL 记录的 LSN，如图 3.24 所示。我们也可以利用 pageinspect 插件查看任何一个表的任何一个数据块的 pd_lsn。

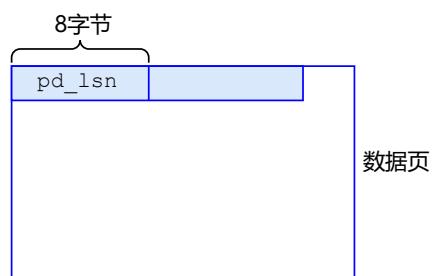


图 3.24: 数据块中的 LSN

既然 pd_lsn 记录着最近一次对本数据块修改的操作所对应的 WAL 记录的 LSN，当我们拿到一个修改本数据块的 WAL 记录时，只要判断该 WAL 记录的 LSN 是否大于 pg_lsn，就可以决定这条 WAL 记录所保存的修改到底有没有被施加到本数据块上。如果 WAL 记录的 LSN 小于等于数据块的 pd_lsn，就说明这条 WAL 记录的修改已经反应在此时的数据块上了，无需再次执行，我们就可以跳过这条 WAL 记录。如果 WAL 记录的 LSN 大于

数据块的 `pd_lsn`, 则说明这条 WAL 记录所记载的修改信息还没有施加到数据块上, 就需要把该 WAL 记录保存的修改信息在这个数据块上回放一下。下面我们从 WAL 记录的诞生和消费两个角度来分析这个对数据库恢复至关重要的过程, 图 3.25 展示了 WAL 记录产生的基本过程。

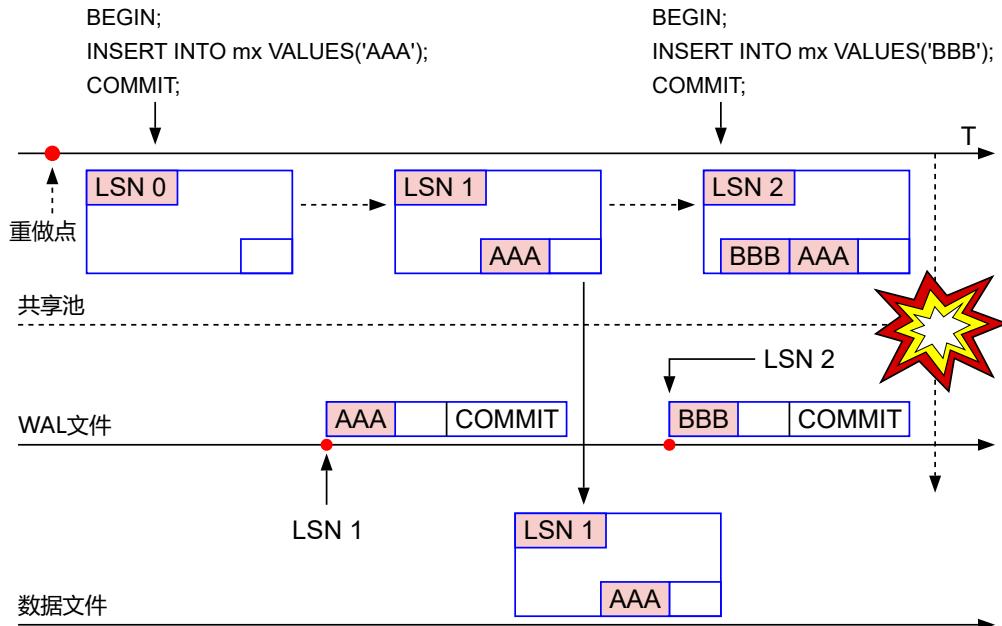


图 3.25: WAL 记录的产生

如图所示, 某个事务往 `mx` 表中插入一条记录 ('AAA'), 这个插入操作有两个动作, 其一是产生相应的 WAL 记录, 假设其位置是 LSN_1 。其二是在数据页中插入数据, 并且把该数据页的 `pd_lsn` 修改为 LSN_1 , 表示 LSN_1 的 WAL 记录是修改此数据页的最后操作。当该事务被提交后, 由它产生的 WAL 记录, 通常是两条, 一条插入, 一条提交, 被首先保存在 WAL 文件中。这个脏的数据页会一直滞留在内存中, 不一定会落盘。过了一段时间, 后台写进程可能会把这个数据页落盘, 但这个不重要, 因为只要 WAL 记录落盘了, 数据就不会丢失, 数据页落盘与否无关紧要, 很快我们就会看到为什么会这样。假设这个数据页被写回到磁盘上了, 则对应的数据块上的 `pd_lsn` 也是 LSN_1 。后来, 另一个事务往 `mx` 表中插入了新的记录 ('BBB'), 且成功提交了, 它产生的两条 WAL 记录也被可靠地添加到 WAL 文件的尾部, 位置是 LSN_2 。在此之后, 突然停电, 导致数据库崩溃了。这种情况下我们无需惊慌, 因为 WAL 记录已经被可靠地保存在磁盘上了, 数据块可以被 WAL 记录修正到最新的状态。等数据库集群再次启动后, 将进入到崩溃恢复阶段, 图 3.26 展示了回放过程。

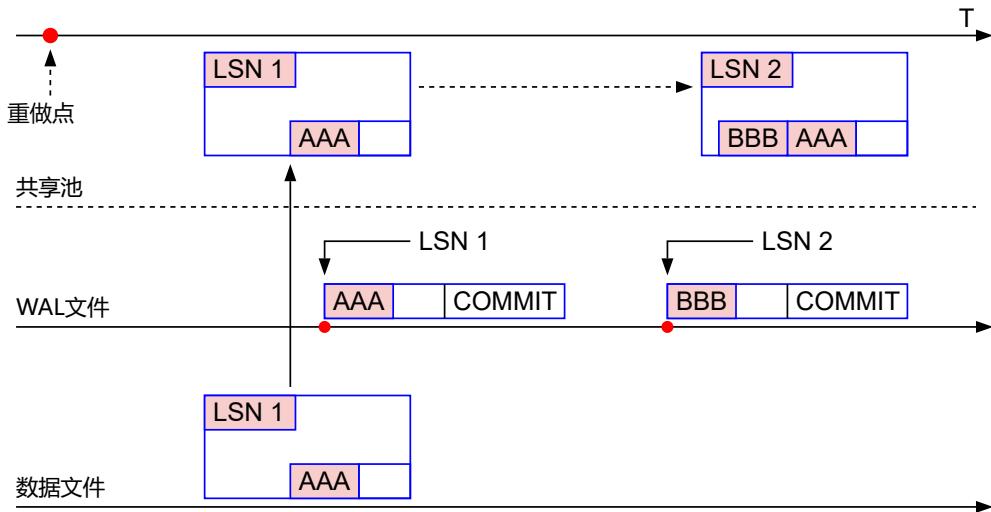


图 3.26: WAL 记录的回放

数据库集群再次启动后，恢复进程发现数据库不是干净的关闭，它就会从控制文件中拿到重做点，就从这一点开始在 WAL 文件中向后(向未来的方向)依次读取 WAL 记录进行回放。注意：WAL 记录是按先后顺序被写入到 WAL 文件当中的，读取的时候也要依次读取，次序非常重要，绝对不能搞乱。当恢复进程读取到 LSN_1 的 WAL 记录后，因为 WAL 记录中包含数据块的位置信息，恢复进程就把该数据块读入到内存中，变成了数据页 X。恢复进程然后对比该 WAL 记录和数据页 X 中的 LSN，发现它们都是 LSN_1 ，则直接跳过该条 WAL 记录，因为该 WAL 记录插入的记录早已被保存在数据页 X 中了，无需重复插入。等恢复进程读取到 LSN_2 的 WAL 记录后，发现 LSN_2 比 LSN_1 大，则恢复进程会根据此条 WAL 记录的信息，再次把‘BBB’插入到数据页 X 中，同时把该页的 pd_lsn 修改为 LSN_2 ，这就是一次完整的 WAL 记录的回放过程。

恢复进程就这样机械地依次读取 WAL 记录，对 WAL 记录的 LSN 和数据页中的 pd_lsn 比大小：如果 WAL 记录的 LSN 小于或者等于数据页的 pd_lsn，就跳过该条 WAL 记录。如果 WAL 记录的 LSN 大于数据页的 pd_lsn，则重新执行该条 WAL 记录的动作，该插入的插入，该修改的修改，该删除的删除。恢复进程会一直重复这样的工作，直至 pg_wal 目录下所有的 WAL 记录都被回放完为止，就完成了恢复工作，恢复进程就退出了。这就是数据库崩溃恢复过程的基本轮廓。下面我们看一段关键代码。

```
/* in src/include/storage/bufpage.h */
#define PageXLogRecPtrGet(val) ((uint64) (val).xlogid << 32 | (val).xrecoff)
#define PageGetLSN(page) PageXLogRecPtrGet(((PageHeader) (page))->pd_lsn)
/* in src/include/access/xlogutils.h */

typedef enum {
    BLK_NEEDS_REDO, /* changes from WAL record need to be applied */
    BLK_DONE,        /* block is already up-to-date */
    BLK_RESTORED,   /* block was restored from a full-page image */
    BLK_NOTFOUND    /* block was not found (and hence does not need to be replayed) */
} XLogredoAction;

/* in src/backend/access/transam/xlogutils.c */
XLogredoAction XLogReadBufferForRedoExtended(XLogReaderState *record, uint8 block_id,
                                              ReadBufferMode mode, bool get_cleanup_lock, Buffer *buf)
{
    XLogRecPtr lsn = record->EndRecPtr;
    .....
    if (lsn <= PageGetLSN(BufferGetPage(*buf))) /* LSN的比较逻辑 */

```

```

    return BLK_DONE;
else
    return BLK_NEEDS_REDO;
.....
}

```

在上面的代码中，`lsn` 表示 WAL 记录的 LSN，它会和数据页的 LSN(通过 `PageGetLSN()` 这个宏)相比较，如果 `lsn` 小于等于后者，则返回 `BLK_DONE`，表示该数据页已经更新到了最新的状态，无需再拿这个 WAL 记录进行回放了。否则就返回 `BLK_NEEDS_REDO`，表示需要用本 WAL 记录来修正数据页中的内容。下面我们研究一下全页写(FPI)类型的 WAL 记录是如何修复数据块的恢复流程，图 3.27 展示了全页写 WAL 记录产生的过程。

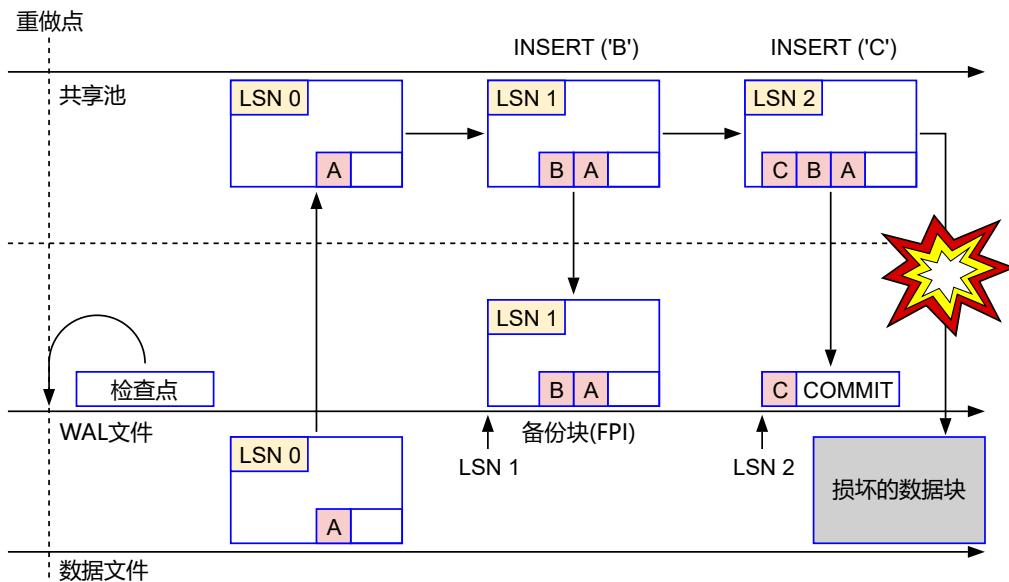


图 3.27: 全页写 WAL 记录的产生过程

在上图的最左边，一个检查点执行完毕后，在 LSN_1 的位置，某个事务往数据页 X 中插入了一条记录 ('B')，因为这是该数据页自检查点发生以来的第一次修改，所以 PostgreSQL 会把该页的全部内容作为一个备份块写入到 WAL 文件中，即图中的 FPI，里面包含了老记录 ('A') 和本次事务插入的新记录 ('B')。当然，全页写的 WAL 记录格式肯定和数据页 X 的格式不同，但两者的内容是一样的。为了更形象地演示，图中把全页写 WAL 记录的形象画成了和数据页的结构一样。在 LSN_2 的位置，另外一个事务往该数据页中插入了一条记录 ('C')。这次修改就会写入一条普通的 WAL 记录，里面只包含了 'C' 的内容。第二个事务提交成功后，系统发生了崩溃，导致数据块 X 发生了损坏。那么如何修复该数据块呢？图 3.28 展示了全页写 WAL 记录的修复过程。

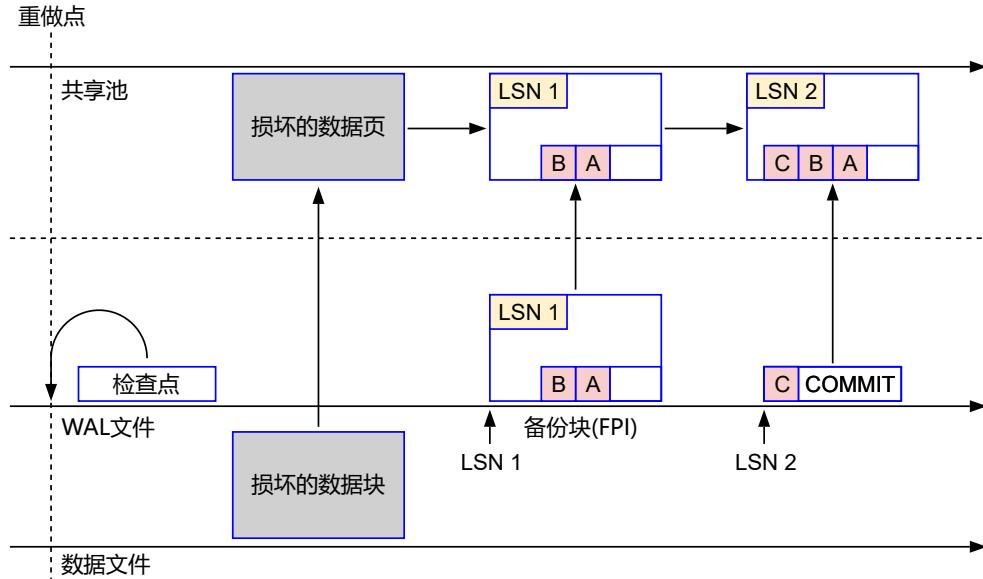


图 3.28: 全页写 WAL 记录对数据块的修复过程

数据库集群启动后，恢复流程自重做点开始恢复。当它读到 LSN_1 位置的备份块后，也会把损坏的数据块 X 读入到共享池中，变成了数据页 X。此时恢复进程发现该条 WAL 记录是全页写类型的，就根本不理会数据页 X 中的内容，也不会对比 LSN，而是直接拿备份块以全覆盖的形式，修复了数据页 X，里面包含了'A' 和'B' 两条记录。此时，数据页 X 中的 pd_lsn 为 LSN_1 。当恢复流程读取到了 LSN_2 位置的 WAL 记录，发现 LSN_2 比 LSN_1 大，就拿 LSN_2 的 WAL 记录更新数据页 X 里的内容，再次插入了记录'C'。至此，数据已经可靠地恢复到了崩溃之前的状态。由此可知，全页写一旦打开，数据块损坏与否并不重要，只要备份块的 WAL 记录被可靠地写入到磁盘，该数据块照样可以被恢复出来。这一个结论非常重要，它是我们理解备份和恢复的一个关键要点。

3.3.2 崩溃恢复的流程

理解了 WAL 记录回放的基本过程后，我们来研究一下数据库是如何恢复的。在数据库运行期间可能会发生各种故障和灾难，数据库软件必须有能力在各种情形下做恢复。数据库的恢复有三种模式：

- 崩溃恢复 (crash recovery)：就是数据库在运行过程中遭遇如突然断电等事故后，要把数据库重新启动并恢复到崩溃之前的状态。
- 归档恢复 (archive recovery)：就是常说的利用数据库的备份进行恢复，也可以称之为备份恢复。我们将在下一章进行专题研究。
- 备库恢复 (standby recovery)：即数据库的流复制技术，把 WAL 记录通过网络传播到远端的备库上进行恢复，第五章会讲解。

本节只研究崩溃恢复的流程。数据库实例启动时，恢复进程负责数据库的恢复工作。平日我们看不见这个进程，是因为在大部分情况下恢复进程做的工作是崩溃恢复，而在正常情况下，数据库是被干净地关闭的。即使因为断电而崩溃，`pg_wal` 目录下的需要回放的 WAL 记录也不多，恢复进程很快就完成了回放任务就退出了，因此我们用 `ps` 命令往往看不到它的身影。在第四章数据库恢复和第五章的数据库复制实验中，你会看到恢复进程常驻内存的倩影。

3.3.2.1 数据库实例的关闭模式

在了解崩溃恢复过程之前，需要了解数据库实例的关闭模式。pg_ctl 是用来关闭数据库实例的，它由如下的选项：

```
$ pg_ctl --help
pg_ctl is a utility to initialize, start, stop, or control a PostgreSQL server.

Usage:
.....
pg_ctl stop      [-D DATADIR] [-m SHUTDOWN-MODE] [-W] [-t SECS] [-s]
.....
Options for stop or restart:
-m, --mode=MODE      MODE can be "smart", "fast", or "immediate"

Shutdown modes are:
smart      quit after all clients have disconnected
fast       quit directly, with proper shutdown (default)
immediate  quit without complete shutdown; will lead to recovery on restart
```

由上可知，关闭数据库实例有三个选项：聪明模式 (smart)，快速模式 (fast) 和立即模式 (immediate)。聪明模式一点都不聪明，它必须要等待所有的客户端断开后才关闭数据库实例，这个模式太软弱了，DBA 一般不用它。快速模式是缺省模式，它无视客户端的存在，做一些扫尾工作后直接关闭数据库实例。聪明模式和快速模式都可以干净地关闭数据库实例，但有时数据库实例无法被干净地关闭，你就需要使用比较粗暴的立即模式，它类似 Oracle 数据库中的 shutdown abort 命令，会强行关闭数据库实例。下面的实验展示了快速模式和立即模式的不同之处。

```
$ pg_ctl status          /* 查看一下数据库的运行状态，此时数据库处于运行状态 */
pg_ctl: server is running (PID: 18058)
/opt/software/pg16/bin/postgres
/* 现在使用快速模式关闭数据库实例，由于快速模式是缺省模式，所以也可以不用指定。*/
$ pg_ctl stop -m fast
waiting for server to shut down.... done
server stopped
/* 数据库实例被关闭后，控制文件中的内容是"shut down"，表明数据库是被干净地关闭的 */
$ pg_controldata | grep state
Database cluster state:           shut down
$ pg_ctl start -l logfile /* 再次重新启动数据库实例 */
waiting for server to start.... done
server started
/* 再次检查数据库实例的运行状态，是"in production"，一切安好。 */
$ pg_controldata | grep state
Database cluster state:           in production
$ pg_ctl stop -m immediate /* 本次采用粗暴的立即模式关闭数据库实例 */
waiting for server to shut down.... done
server stopped
/* 数据库实例被关闭后，控制文件中的状态依然是"in production" */
$ pg_controldata | grep state
Database cluster state:           in production
```

由上可知，当数据库实例运行时，控制文件中的”Database cluster state”域记录的是”in production”。使用快速模式正常关闭数据库实例后，该值变成了”shut down”。但用立即模式粗暴关闭数据库实例后，控制文件中的内容依然显示为”in production”，这说明数据库集群被没有干净地关闭，可能是因为某种原因而突然崩溃的。数据库实例再次启动时就需要进行崩溃恢复，此时恢复进程就要出场了。

3.3.2.2 恢复进程的工作内容

本节要研究一下恢复进程的工作内容，这部分知识是理解后面备份恢复和流复制技术的前提。恢复进程的主要工作逻辑在 StartupXLOG() 函数中，相关代码如下：

```
/* in src/postmaster/startup.c */
void StartupProcessMain(void)
{
    .....
    StartupXLOG();
    .....
}

/* in src/backend/access/transam/xlog.c */
void StartupXLOG(void)
{
    .....
    InitWalRecovery(ControlFile, &wasShutdown, &haveBackupLabel, &haveTblspcMap);
    .....
    if (InRecovery) {
        .....
        PerformWalRecovery();
    }
    .....
}
```

由上面的代码骨架我们可以看出，恢复进程的入口函数是 StartupProcessMain()，这个函数最主要的工作就是调用 StartupXLOG() 函数。因为恢复进程的代码涵盖了三种恢复模式，所以理解 StartupXLOG() 函数的工作内容非常重要。该函数的主要工作有两个，第一是判断数据库是否要进行恢复，这部分工作由 InitWalRecovery() 函数执行；第二是如果需要做恢复，就执行恢复，该部分工作由函数 PerformWalRecovery() 执行。下面我们就重点考察这两个函数。

InitWalRecovery() 函数会调用 readRecoverySignalFile() 函数检查两个文件是否存在，standby.signal 和 recovery.signal，相关代码如下：

```
/* in src/include/access/xlog.h */
#define RECOVERY_SIGNAL_FILE      "recovery.signal"
#define STANDBY_SIGNAL_FILE       "standby.signal"
#define BACKUP_LABEL_FILE         "backup_label"
/* in src/backend/access/transam/xlogrecovery.c */
static void readRecoverySignalFile(void)
{
    .....
    if (stat(STANDBY_SIGNAL_FILE, &stat_buf) == 0) {
        .... /* 如果找到了standby.signal文件，就进入备库模式 */
        standby_signal_file_found = true;
    }
}
```

```

else if (stat(RECOVERY_SIGNAL_FILE, &stat_buf) == 0) {
    ..... /* 如果找到了recovery.signal, 就进入归档恢复模式 */
    recovery_signal_file_found = true;
}

.....
StandbyModeRequested = false;
ArchiveRecoveryRequested = false;
if (standby_signal_file_found) {
    StandbyModeRequested = true;
    ArchiveRecoveryRequested = true;
}
else if (recovery_signal_file_found) {
    StandbyModeRequested = false;
    ArchiveRecoveryRequested = true;
}
else return;
}

```

由上述代码我们可以知道，恢复进程首先会检查有没有 standby.signal 这个文件，如果有，则意味着该数据库要进入到备库模式，StandbyModeRequested 为 true，这是第五章要讨论的内容。如果没有，再检查是否有 recovery.signal 这个文件。如果有，则数据库进入到归档恢复的模式，ArchiveRecoveryRequested 为 true。否则就再进一步检查是否需要进入到崩溃恢复模式。这两个文件是信号文件，即它们的存在表示某种信号，文件本身的内容没有什么作用。由于这两个文件往往需要用户手动创建，所以明白这部分逻辑对于我们做数据库恢复和创建备库有非常重要的意义。

紧接着，恢复进程会调用 read_backup_label() 函数，判断 backup_label 这个文件是否存在。这个文件是一个小小的文本文件，里面记录一个重做点，指示数据库要从该重做点进行恢复。如果该文件不存在，则恢复进程从控制文件中读取重做点。关于这个文件的详细讨论，我们在下一章再研究，现在我们只要知道必须要从控制文件或者一个文本文件那里拿到一个重做点即可。拿到了这个重做点之后，要对这个重做点对应的检查点 WAL 记录的有效性进行判断。我们知道：如果数据库没有被干净地关闭，控制文件中的状态信息是“in production”，此时需要做恢复。如果数据库集群被干净地关闭，控制文件中的状态是“shut down”，而且控制文件中记录的检查点 WAL 记录是 XLOG_CHECKPOINT_SHUTDOWN 类型的，它的规律是检查点的 LSN 和重做点的 LSN 是相等的，恢复进程根据这个规律来判断数据库是否被干净地关闭掉了，下面的一段代码展示了这个逻辑：

```

/* in src/backend/access/transam/xlogutil.c */
bool InRecovery = false; /* 该全局变量表明数据库是否处于恢复状态中 */
/* in src/backend/access/transam/xlogrecovery.c */
static XLogRecPtr CheckPointLoc = InvalidXLogRecPtr;
void InitWalRecovery(...)
{
    bool wasShutdown;
    CheckPoint checkPoint;
    .....
    /* 如果控制文件中的检查点WAL记录是XLOG_CHECKPOINT_SHUTDOWN类型，则wasShutdown = TRUE */
    wasShutdown = ((record->xl_info & ~XLR_INFO_MASK) == XLOG_CHECKPOINT_SHUTDOWN);
    InRecovery = true; /* force recovery even if SHUTDOWNED */
    .....
    /* 如果RedoPoint < CheckPoint，表明两个LSN之间存在一些WAL记录，需要恢复，InRecovery = true*/
    if (checkPoint.redo < CheckPointLoc) {
        /* 一个XLOG_CHECKPOINT_SHUTDOWN类型的检查点，其RedoPoint < CheckPoint是不正常的 */
    }
}

```

```

if (wasShutdown) /* 此种情况下，数据库实例拒绝启动，直接退出 */
    ereport(PANIC, (errmsg("invalid redo record in shutdown checkpoint")));
InRecovery = true;
} else if (ControlFile->state != DB_SHUTDOWNED) InRecovery = true;
}

```

上述恢复进程的逻辑可以用图 3.29 来总结其基本的处理流程：

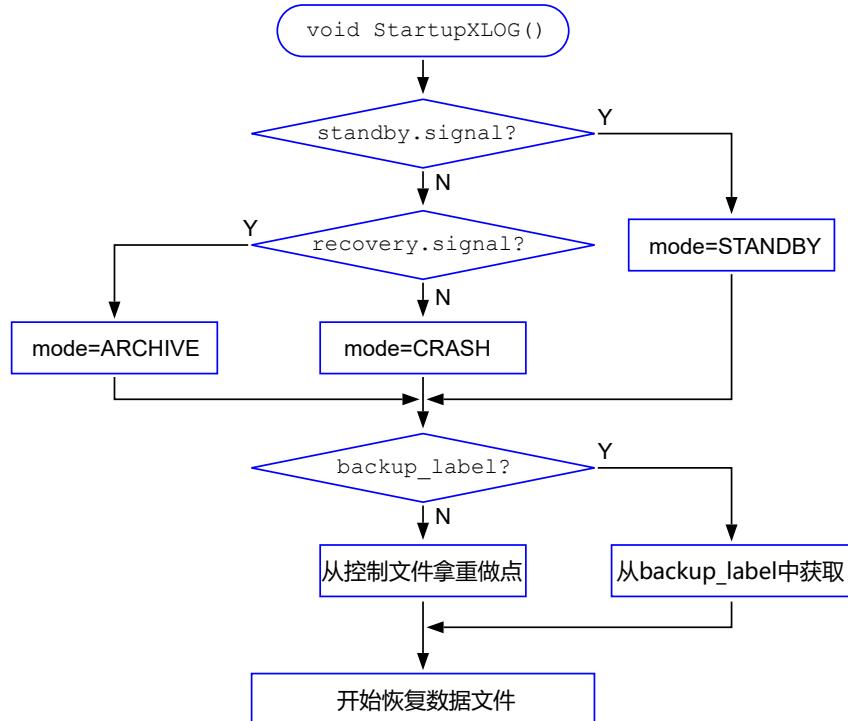


图 3.29：恢复进程的基本逻辑

在恢复进程判断恢复模式的过程中，涉及到了三个文件：standby.signal, recovery.signal 和 backup_label。这里要注意两点：第一是两个信号文件的作用是指示数据库进入何种模式，standby.signal 的优先级比 recovery.signal 的优先级高。第二是 backup_label 文件的作用是规定重做点从哪里开始，它和 standby.signal, recovery.signal 这两个文件没有关系，是相互独立的。

如果确定要进入到恢复阶段，恢复进程会调用 PerformWalRecovery() 函数来做真正的数据库恢复工作。无论哪种的恢复模式，其基本原理是一样的，就是从某一个重做点开始，依次读取 WAL 文件中的记录，把这些 WAL 记录中的 LSN 和相应的数据块的 LSN 比大小，以此决定是否修正数据块的内容。这里存在一个 WAL 记录来源的问题，即从哪里获得 WAL 文件和 WAL 记录。在源码中的相关定义如下：

```

/* in src/backend/access/transam/xlogrecovery.c */
typedef enum {
    XLOG_FROM_ANY = 0, /* request to read WAL from any source */
    XLOG_FROM_ARCHIVE, /* restored using restore_command */
    XLOG_FROM_PG_WAL, /* existing file in pg_wal */
    XLOG_FROM_STREAM   /* streamed from primary */
} XLogSource;
/* human-readable names for XLogSources, for debugging output */
static const char *const xlogSourceNames[] = {"any", "archive", "pg_wal", "stream"};

```

由这个定义可知，恢复进程获取 WAL 记录的来源有三个：XLOG_FROM_PG_WAL 表示在 pg_wal 目录下寻找，XLOG_FROM_ARCHIVE 表示通过参数 restore_command 里规定的命令把 WAL 文件从某一个地方拷贝到 pg_wal 目录中，XLOG_FROM_STREAM 表示通过网络从别的数据库获得 WAL 记录。恢复进程就反复在这三种来源中尝试获得更多的 WAL 记录。恢复进程在什么时候完成恢复任务呢？三种恢复模式各不相同。在崩溃恢复模式下，恢复进程只需要把 pg_wal 目录下能发现的 WAL 记录恢复完毕，就可以退出了。归档恢复模式下，在恢复之前会指定一个终点，恢复进程回放 WAL 记录的位置达到或者超过这个终点，归档恢复就完成了，恢复进程也就退出了。在备库模式下，恢复永远在路上，是永无止境的。归档恢复和备份恢复的内容在第四章和第五章中会详细讨论，这里就不再展开论述了。下面是 PerformWalRecovery() 函数的基本结构：

```
/* in src/backend/access/transam/xlogrecovery.c */
void PerformWalRecovery(void)
{
    .....
    do {
        .....
        /* Have we reached our recovery target? */
        if (recoveryStopsBefore(xlogreader)) {
            reachedRecoveryTarget = true;
            break;
        }
        .....
        /* Apply the record */
        ApplyWalRecord(xlogreader, record, &replayTLI);
        /* Exit loop if we reached inclusive recovery target */
        if (recoveryStopsAfter(xlogreader)) {
            reachedRecoveryTarget = true;
            break;
        }
        /* Else, try to fetch the next WAL record */
        record = ReadRecord(xlogprefetcher, LOG, false, replayTLI);
    } while (record != NULL);
    .....
}
```

我们可以看到，该函数的基本框架就是一个循环：它会不断地通过 ReadRecord() 函数读取 WAL 记录，然后调用 ApplyWalRecord() 函数对 WAL 记录进行回放，直至达到设定的恢复点或者没有 WAL 记录可读为止。ReadRecord() 函数就会在三种 WAL 记录的来源中来回巡视，试图获得更多的 WAL 记录。

3.4 WAL 文件的管理

我们知道 WAL 文件的尺寸是固定的，缺省是 16MB。随着往它里面写入的 WAL 记录不断增加，必然存在一个写满的时候。当一个 WAL 文件被写满后，数据库实例会自动创建一个新的 WAL 文件，这个过程叫作 WAL 文件的切换 (WAL segment switch)。你也可以手工执行 pg_switch_wal() 函数来强制数据库实例进行 WAL 文件的切换。下面的实验展示了如何手工切换 WAL 文件：

```
postgres=# SELECT pg_current_wal_lsn(); /* 查看当前WAL指针 */
pg_current_wal_lsn
-----
0/18CB150
(1 row)

postgres=# ! ls -l $PGDATA/pg_wal /* 查看pg_wal目录下的WAL文件 */
total 16388
-rw----- 1 postgres postgres 16777216 Oct 28 13:19 000000010000000000000000
drwx----- 2 postgres postgres      4096 Oct 28 05:45 archive_status
```

我们可以看到，因为当前的 LSN 指针是 0/18CB150，它属于 1 号 WAL 文件，在 pg_wal 目录下只有这一个 WAL 文件。现在我们执行手工切换：

```
postgres=# SELECT pg_switch_wal(); /* 手工执行WAL文件的切换 */
pg_switch_wal
-----
0/18CB1A0
(1 row)

postgres=# SELECT pg_current_wal_lsn(); /* 查看当前WAL指针 */
pg_current_wal_lsn
-----
0/2000060
(1 row)

postgres=# ! ls -l $PGDATA/pg_wal /* 查看pg_wal目录下的WAL文件，结果产生一个新的WAL文件，编号为2 */
total 32772
-rw----- 1 postgres postgres 16777216 Oct 28 13:20 000000010000000000000001
-rw----- 1 postgres postgres 16777216 Oct 28 13:20 000000010000000000000002
drwx----- 2 postgres postgres      4096 Oct 28 05:45 archive_status
postgres=*
```

可以看出，当我们手工切换后，当前 LSN 指针变成了 0/2000060，它所在的 WAL 文件是 2 号，所以 1 号 WAL 文件就变成了老的 WAL 文件。此时在 pg_wal 目录下我们可以看到 1 号和 2 号两个 WAL 文件。这里需要注意的一点是：在手工切换过程中，如果从上一次切换到现在，并没有产生新的 WAL 记录，则 PostgreSQL 会拒绝进行切换。所以你如果连续快速地做几次手动切换，你会发现有时候并没有产生新的 WAL 文件，原因就是你切换的太快，在两次切换之间并没有产生新的 WAL 文件，则 PostgreSQL 认为没有切换的必要。

WAL 文件的切换触发条件有 3 个：第一个就是手工切换，当然这种情况比较少，毕竟很少有人闲着没事手工切换着玩。第二种就是当 WAL 文件写满后，就自动切换，这种情况最频繁。第三种情况是当数据库处于归档模式时，参数 archive_timeout 规定了一个时间间隔，当超过这个时间间隔后，检查点进程会自动执行一次 WAL 文件的切换，具体可以参阅源代码 checkpointer.c 中的 CheckArchiveTimeout() 函数。

数据库实例把所有的 WAL 文件都存放在 \$PGDATA/pg_wal 目录下，随着时间的推移，里面积累的 WAL 文件势必越来越多，这就存在一个清理的问题。那么，哪些 WAL 文件是当前数据库所不需要的呢？很显然，最近一次成功执行的检查点对应的重做点之前的 WAL 记录是当前运行的数据库不需要的，可以被删除掉，如图 3.30

所示。

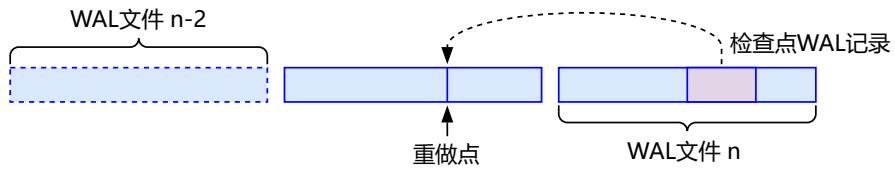


图 3.30: WAL 文件可以被删除的条件

如上图所示，最后一次检查点的 WAL 记录在 n 号 WAL 文件上，它的重做点在 n-1 号 WAL 文件上，则编号为 n-2 的 WAL 文件和以前的 WAL 文件都可以被安全的删除了。当然，这些能被删除的老的 WAL 文件加上物理备份，可以把数据库恢复到重做点之前的某一个状态，这就是下一章要介绍的备份和归档模式，本节只考虑崩溃恢复的需要，不考虑归档。每次检查点完成后，数据库集群会根据一定的算法，来决定删除哪些旧的 WAL 文件，以免撑爆 pg_wal 目录，这个 WAL 文件的清理工作是由检查点进程在执行检查点操作时执行的。PostgreSQL 提供了两个参数，规定了在 pg_wal 目录下的 WAL 文件的体积的上限和下限。参数 min_wal_size 定义了需要保存的最少的 WAL 文件的体积，它的缺省单位是 MB，缺省值是 80MB。如果 WAL 文件的体积是 16MB，则 80MB 意味着在 pg_wal 目录下必须保存 5 个 WAL 文件。当 pg_wal 目录下 WAL 文件的体积小于 min_wal_size 规定的值时，即将被删除的老的 WAL 文件并不会被删除，而是重新改名，变成新的 WAL 文件加以利用。参数 max_wal_size 规定了 WAL 文件的最大体积，它的缺省单位是 MB，缺省值是 1024MB。一旦 pg_wal 目录下所有的 WAL 文件的总体积超过了 max_wal_size，数据库集群就会强制发生一次检查点，确保目前的 WAL 文件可以删除掉。max_wal_size 的值是软性指标，在一些情况下，譬如归档失败，复制槽对应的备库宕机等情况下，pg_wal 目录下的 WAL 文件可能会积累出比 max_wal_size 多得多的 WAL 文件。

数据库的负荷不同，数据修改的频度也不同。活跃的数据库几秒钟就要切换一下 WAL 文件，而相对静止的数据库很长时间也不会把一个 WAL 文件填满。每次发生检查点时，PostgreSQL 会预先分配一定数量的 WAL 文件，保持一定的富裕度。到底预留多少个 WAL 文件合适呢？很显然这没有一个绝对的数值，需要根据数据库的活跃程度进行动态调整。一个很自然的算法是：根据本次检查点和上一次检查点的距离来确定所需要预留的 WAL 文件的数量。WAL 文件被自动删除，在备份和流复制领域是一个需要关注的问题，这个问题会在后续章节进行讨论。

第四章 备份和恢复

数据库的备份 (backup) 和各种条件下的恢复 (recovery) 技术，是数据库运维领域永恒的主题。DBA 是数据的守护神，如果没有可靠的备份导致数据丢失，DBA 只能跑路了。当然，一切无法恢复的备份都是耍流氓，备份是为恢复服务的，两者是一个硬币的两个方面，缺一不可。本章我们来研究 PostgreSQL 数据库备份和恢复的课题。

PostgreSQL 的备份方式分为物理备份和逻辑备份两种。所谓物理备份就是利用上一章研讨的崩溃恢复的原理，就是先把数据文件和 WAL 文件备份下来，在恢复时从某个重做点开始，利用 WAL 记录的回放功能，依次修正数据块，使得备份数据库和源库的状态达成在备份结束那个时刻的一致。逻辑备份就是把数据库中的数据转化成 SQL 语句，写入一个脚本文件中。在这个脚本文件里面包含大量的如 CREATE TABLE 和 INSERT 等语句。恢复时运行这个脚本中的 SQL 命令，再次创建空表，重新插入记录，完成数据的恢复。物理备份和恢复的速度远超逻辑备份，在体积越大的数据库上速度差异越大，所以物理备份是最主要的备份方式，但是物理备份灵活性不足，它只能备份整个数据库集群，不能备份指定的数据库或者表。逻辑备份却可以指定备份哪个数据库，哪张表，因其灵活性，可以作为备份的辅助手段。所以在实际工作中，企业的备份策略几乎全部都是以物理备份为主，逻辑备份为辅。我们本节讨论的是物理备份，逻辑备份放在第三节中进行学习。

4.1 物理备份

数据库的物理备份的基本原理和上一章的崩溃恢复并无本质区别，请参考图 4.1：

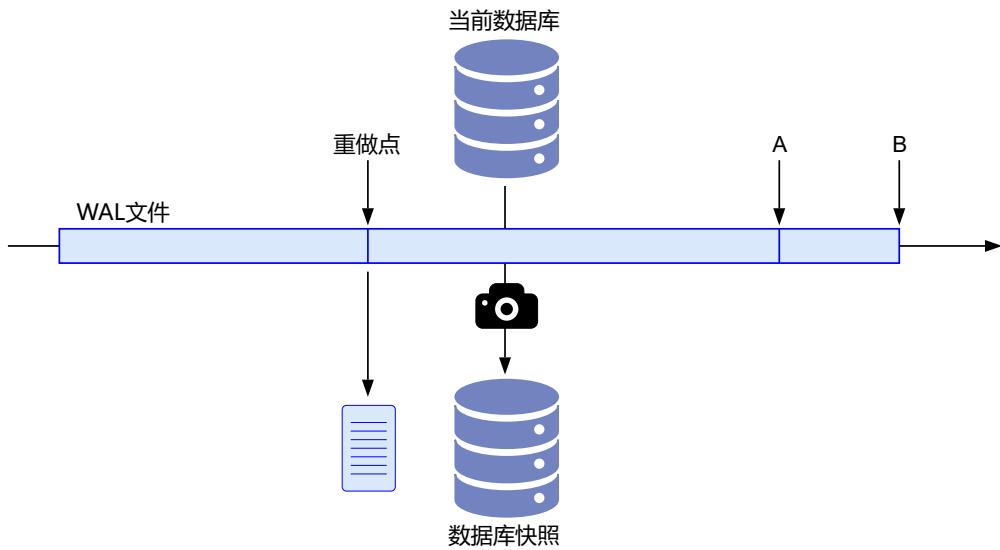


图 4.1：数据库备份的基本原理

假设我们有一个正在运行的数据库 (running database)，这个数据库刚刚完成了一次检查点，其对应的重做点已经标记在图中。如果该数据库崩溃了，则恢复进程从重做点开始，不断回放 WAL 记录，就可以把数据库恢复到重做点右边的任何一点，如 A 点，或者已有的 WAL 记录的终点，即图中的 B 点。我们把 A 点和 B 点称为恢复的“目标点” (recovery target)。

假设我们有一种快照 (snapshot) 技术，能够在瞬间，即所用时间为零，为当前数据库做一次快照，产生一个快照数据库 (snapshot database)，这个数据库和当前数据库并没有任何不同，所以如果当前数据库可以做崩溃恢复，快照数据库没有理由不行。我们手里有了三个东西，一个数据库的快照，重做点和自重做点开始的连续不断

的 WAL 记录，这三个东西就组成了一套完整的物理备份。当然，重做点的 LSN 不能保存在控制文件中，必须保存在别处，因为控制文件中的重做点会随着后续持续发生的检查点操作而被不断覆盖。我们用一个简单的文本文件保存它就可以了，这个文本文件就是我们后面要使用的 `backup_label`。

为什么我们希望有瞬间完成的快照功能呢？因为我们拷贝数据库时，该数据库处于活动状态，依然有 `bgwriter` 和 `checkpointer` 等后台进程可能把内存中的脏页写入数据文件中，如果我们读的过程不能瞬间完成，就存在前脚读的数据没有被修改，后脚读的数据已经被修改的情况，造成读取的数据不一致。很显然，这种快照技术只存在于理论中，现实中你拷贝一个数据库的文件总需要时间，不可能瞬间拷贝完成。我们知道数据文件的体积超过 1GB，就会分成两个文件。拷贝 1GB 的文件总是需要一定时间才能完成的。既然瞬间快照不可能，在拷贝数据库文件过程中，会出现什么可能性呢？让我们来分析一下，请看图 4.2：

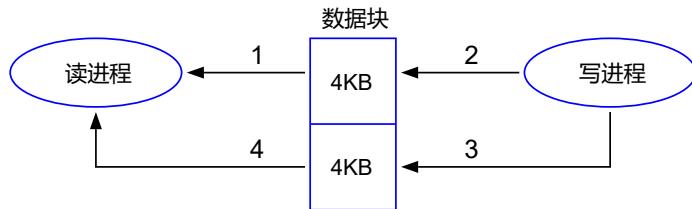


图 4.2：部分写的问题

在上图中，一个读进程 (reader process) 正在读取一个数据文件中的一个数据块，同时另外一个写进程 (writer process) 正在把数据写入该数据块。因为传统的机械硬盘的扇区的大小通常是 512 字节，所以操作系统对数据的原子性读写一般为 512 字节。PostgreSQL 的数据页是通常是 8KB 字节大小，所以需要 16 次原子性读写才能完成，很显然，拷贝 PostgreSQL 的数据块的操作不可能是原子性的读。那么会发生什么问题呢？

读进程首先读取了数据块中前半段的数据，即上图中所示的步骤 1。紧接着，写进程把新的数据写入到该数据块的前半段中，即步骤 2。然后该写进程又往后半段中写入了新数据，即步骤 3。最后读进程把后半段的数据拷贝走，即步骤 4，所以整个读写过程发生的先后顺序是从步骤 1 到步骤 2 到步骤 3 到步骤 4。很显然，读进程读取的这个数据块是损坏的，这种情况被称为部分写 (partially written)，叫部分读 (partially read) 也可以，这个损坏的数据块的英文术语是 `torn page`，我们可以称之为“坏块”。另外一种读写顺序也可能产生部分写的现象，就是从步骤 2 到步骤 1 到步骤 4 到步骤 3。如果读写顺序变成了从步骤 1 到步骤 2 到步骤 4 到步骤 3 或者从步骤 2 到步骤 1 到步骤 3 到步骤 4，则这是正常的读写操作：读进程读到了完整的数据，写进程也把完整的数据写入到了磁盘。

拷贝数据文件是需要花费时间的，所以在拷贝的过程中发生部分写的现象不可避免。由部分写导致的坏块会影响未来的恢复吗？在前一章我们讨论过全页写 (FPW) 这个功能，我们知道：在全页写模式打开的情况下，任何一个数据页在最近一次检查点发生后的第一次修改，它的全部信息作为一个 FPW 类型的 WAL 记录被写入到 WAL 文件中。假设我们在拷贝之前做两个动作，一个是打开全页写模式，一个是执行一个检查点操作，如果在拷贝期间某个数据块发生了修改，在它第一次修改时，它的全部内容必定会被以 FPW 类型的 WAL 记录保存在 WAL 文件中。在未来恢复时，恢复进程会直接使用该 FPW 类型的 WAL 记录中的数据对整个数据页进行全覆盖，而我们拷贝到的这个数据块是否是损坏的，就没有任何关系了。由此看来，只要在拷贝之前强制打开全页写模式，再做一次检查点操作，在随后的数据文件拷贝过程中，我们根本不需要瞬间快照的技术，完全可以用操作系统提供的 `cp` 或者 `tar` 等任何有效的拷贝命令慢悠悠地拷贝数据，多长时间都没有关系，拷贝的坏块不会影响未来的数据库恢复。后面我们会看到 PostgreSQL 在备份之前会强制变成 FPW 模式，并且强制执行一次检查点操作，目的就是解决部分写的问题。

4.1.1 归档模式

在学习数据库备份之前，我们需要先了解一下归档模式这个概念。PostgreSQL 的运行模式可以分为归档模式和非归档模式，它们的概念可以用图 4.3 来表示：

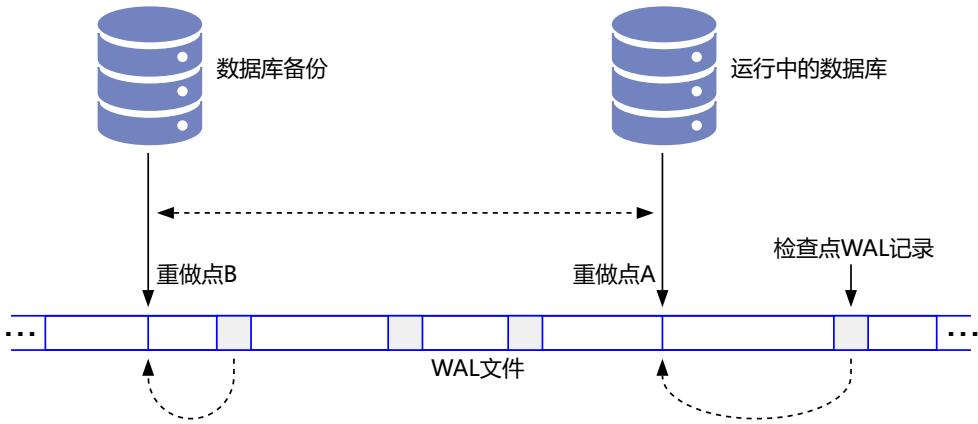


图 4.3: 数据库的归档模式

如上图所示，在右上角的数据库是当前数据库 (current database)，与之对应的有一个重做点 A 被保存在控制文件中。假设我们有一个物理备份 (base backup)，如图中左上角所示，它对应的重做点 B 被保存在某一个文件中。虽然 A 和 B 之间的 WAL 记录对于当前数据库的崩溃恢复是不需要的，但对于左上角的数据库备份而言，为了成功恢复它，必须保存从重做点 B 开始的连续的 WAL 文件，所以 A 和 B 之间的 WAL 记录对于它的恢复是必需的。关于是不是需要保留 A 和 B 之间的这段 WAL 记录，有两种处理模式：

- 不保留，则只能做从重做点 A 开始的崩溃恢复。数据库备份无效。我们把这种模式称为“非归档”模式。
- 保留，则我们可以从重做点 B 开始恢复数据库备份。我们把这种模式称为“归档”模式。

所谓归档 (archiving)，是把档案归类保存之意，在这里实际上就是把重做点 A 之前的，当前数据库不需要的，老的 WAL 文件拷贝到一个指定的地方 (本地目录，或者云端的存储，譬如 AWS 的 S3)。我们把这个目录称为“归档目录”。归档模式由参数 `archive_mode` 控制，一旦设定为归档模式，则我们还需要指定一个归档命令，告诉 PostgreSQL 如何把 WAL 文件从 `pg_wal` 目录拷贝到归档目录，这个命令由参数 `archive_command` 设定。在归档模式下，每当 WAL 文件进行切换时，PostgreSQL 会调用 `archive_command` 参数指定的拷贝命令把老 WAL 文件拷贝到归档目录。此外，还需要确保 `wal_level` 的值至少是 `replica`，这个是缺省值，一般不需要修改。

4.1.1.1 设置归档模式的实验

下面我们通过实验来演示如何修改 PostgreSQL 的归档模式，具体操作如下：

```
$ psql /* 以超级用户用psql登录数据库 */
psql (16.0)
Type "help" for help.
/* wal_level参数必须是replica或者logical */
postgres=# SHOW wal_level;
wal_level
-----
replica
(1 row)

postgres=# SHOW archive_mode; /* 目前处于非归档模式 */
archive_mode
-----
off
(1 row)

postgres=# \! vi $PGDATA/postgresql.conf    /* 修改postgresql.conf配置文件 */
```

```
/* 检查archive_mode和archive_command参数 */
postgres=# \! cat $PGDATA/postgresql.conf | grep -E 'archive_mode|archive_command'
archive_mode = on
archive_command = 'test ! -f /home/postgres/a/%f && cp %p /home/postgres/a/%f'
postgres=# \! mkdir /home/postgres/a    /* 创建archive_command中指定的归档目录 */
$ pg_ctl restart          /* 重新启动数据库集群才能使archive_mode参数的修改生效 */
waiting for server to shut down.... done
server stopped
.....
server started
```

把数据库设置为归档模式还是比较简单的，只需要修改 `archive_mode` 和 `archive_command` 这两个参数。其中参数 `archive_mode` 控制数据库集群的归档模式，它的取值范围的定义如下：

```
/* in src/include/access/xlog.h */
typedef enum ArchiveMode {
    ARCHIVE_MODE_OFF = 0,      /* disabled */
    ARCHIVE_MODE_ON,           /* enabled while server is running normally */
    ARCHIVE_MODE_ALWAYS        /* enabled always (even during recovery) */
} ArchiveMode;
```

可以看出，这个参数的取值有三种：当它的值为 `off` 时，数据库集群处于非归档模式，`on` 则是归档模式。此外它还有一个值 `always`，这个值的含义表示备库也可以归档，我们在第五章搭建物理备库时再研究它的具体用法。当数据库集群处于归档模式时，还必须设置个参数 `archive_command` 来指定归档动作的具体命令。归档命令本质上就是一个拷贝动作，把需要归档的 WAL 文件从 `pg_wal` 目录拷贝到归档目录。修改参数 `archive_mode` 后需要重启数据库集群才能让它生效，修改参数 `archive_command` 则不需要重启数据库，只要执行 `pg_reload_conf()` 重新加载参数即可。所以我们在里做了数据库重启的操作。在打开了数据库的归档模式以后，我们做一下手工的 WAL 文件切换，看看归档目录下是否有 WAL 文件。具体操作如下：

```
$ psql /* 重新启动数据库以后，检查归档参数是否生效 */
psql (16.0)
Type "help" for help.
/* 检查archive_mode参数，结果为on */
postgres=# SHOW archive_mode;
archive_mode
-----
on
(1 row)

postgres=# SHOW archive_command; /* 检查archive_command参数 */
archive_command
-----
test ! -f /home/postgres/a/%f && cp %p /home/postgres/a/%f
(1 row)

postgres=# \! ls -l /home/postgres/a      /* 检查归档目录，结果为空 */
total 0
postgres=# SELECT pg_switch_wal();       /* 做一次WAL文件的切换 */
pg_switch_wal
-----
0/3000210
(1 row)
```

```
/* 再次检查归档目录，发现老的WAL文件在切换后被拷贝到该目录了 */
postgres=# \! ls -l /home/postgres/a
total 16384
-rw----- 1 postgres postgres 16777216 Oct  1 15:07 0000000100000000000000000003
postgres=# \! ps -ef | grep postgres | grep archive | grep -v grep /* 发现多了个后台进程archiver */
postgres 16863 16857 0 15:06 ? 00:00:00 postgres: archiver last was 0000000100000000000000000003
```

我们看到了，手工执行 WAL 文件切换后，在归档目录/home/postgres/a 中果然发现了被拷贝过来的 WAL 文件。我们还发现：一旦数据库进入了归档模式，主进程就会生成一个新的后台进程专门负责归档工作，这个后台进程被称为“归档进程”(archiver)。在上面的实验中我们通过 ps 命令看到了归档进程。读者可以参考如下的函数：

```
/* in src/backend/postmaster/pgarch.c */
static bool pgarch_archiveXlog(char *xlog)
{
    char activitymsg[MAXFNAMELEN + 16];
    .....
    sprintf(activitymsg, sizeof(activitymsg), "archiving %s", xlog);
    set_ps_display(activitymsg);

    ret = ArchiveCallbacks->archive_file_cb(archive_module_state, xlog, pathname);
    if (ret)
        sprintf(activitymsg, sizeof(activitymsg), "last was %s", xlog);
    else
        sprintf(activitymsg, sizeof(activitymsg), "failed on %s", xlog);
    set_ps_display(activitymsg);
    .....
}
```

从上述源码中我们可以看出，归档进程会调用 ArchiveCallbacks->archive_file_cb 函数执行具体的归档动作。这是一个函数指针，返回值为 true 表示归档成功，否则是归档失败。缺省情况下它调用 shell_archive_file() 函数，后面我们会研究这个函数的功能。归档进程的名字中显示了它正在处于的状态。从源代码中不难看出：如果归档进程显示“archiving”的字样，则表明它正在执行参数 archive_command 中规定的命令拷贝这个文件，但是还没有结束。如果显示“last was”的字样，则表明该 WAL 文件已经被成功地拷贝到了归档目录中。如果显示“failed on”的字样，则表明该 WAL 文件没有被拷贝到归档目录，归档进程过一段时间会反复重试。从实验的输出结果可以看出：归档进程目前的状态是已经是把 WAL 文件 000000010000000000000003 成功拷贝到了归档目录中。

4.1.1.2 监控归档的系统视图

PostgreSQL 提供了一个系统视图 pg_stat_archiver 来让用户掌握归档的情况。我们看一下该系统视图的内容和含义，具体操作如下：

```
oracle=# select pg_stat_reset_shared('archiver'); /* 把累积的统计数据清空 */
-[ RECORD 1 ]-----+
pg_stat_reset_shared |
oracle=# select * from pg_stat_archiver;
-[ RECORD 1 ]-----+
archived_count      | 0
last_archived_wal  |
last_archived_time |
```

```

failed_count | 0
last_failed_wal |
last_failed_time |
stats_reset   | 2023-12-09 14:47:28.328572-07
oracle=# select pg_switch_wal(); /* 手工执行一次WAL文件的切换*/
-[ RECORD 1 ]+-----+
pg_switch_wal | 0/63FD3A8
oracle=# select * from pg_stat_archiver; /* 再次查看该系统视图，发现只成功归档了一次WAL文件*/
-[ RECORD 1 ]+-----+
archived_count | 1
last_archived_wal | 0000000200000000000000000006
last_archived_time | 2023-12-09 14:47:43.926315-07
failed_count | 0
last_failed_wal |
last_failed_time |
stats_reset   | 2023-12-09 14:47:28.328572-07

```

这个系统视图的各列的含义不难理解。头三列统计归档成功的信息，后三列统计归档失败的信息。`archived_count` 表示共计成功归档了多少个 WAL 文件。`last_archived_wal` 表示最后一次成功归档的 WAL 文件的文件名。`last_archived_time` 表示最后一次成功归档的时间。`failed_count` 表示归档失败的次数。`last_failed_wal` 表示最后一次归档失败的 WAL 文件的文件名。`last_failed_time` 则表示最后一次归档失败的时间。对于这个系统视图的使用，要注意两点。第一点是：如果有归档失败的数据，用户要立刻排查原因。我们知道 WAL 文件必须要连续，才能够顺利恢复。只要中断一次，你顶多能恢复到中断点之前的某一个位置，中断点之后的 WAL 文件就没有用处了。譬如我们成功归档了 1 号、2 号、3 号、5 号和 6 号 WAL 文件，4 号文件丢失了，则 5 号和 6 号的 WAL 文件就没有用处了，因为我们只能最多恢复到 3 号 WAL 文件的某一个位置。很显然这不是我们想要的。第二点需要注意的是：通常情况下，归档是按照顺序进行的。但是这不是百分百保证的。在一些特殊情况下，譬如把备库变主库，或者数据库重新启动了，某些 WAL 文件并没有被成功归档。所以当你看到最后一次成功归档的 WAL 文件是 6 号文件，并不意味着 1 号到 5 号都已经百分百成功归档了，当然正常情况下，6 号之前的 WAL 文件很大概率被成功归档。为了百分百确保 WAL 文件被正常归档，用户需要开发一个脚本对已经归档的 WAL 文件的连续性进行检测。一旦发现了中断，就应该立刻排查原因，确保未来的数据库恢复不会遇到麻烦。

4.1.1.3 归档命令

PostgreSQL 并没有提供专有的归档命令，而是把这个灵活性交给了用户。稍微对 Linux 熟悉的用户都知道一个规律：每个进程在退出时，会有一个返回码 (exit code)，通常情况下返回码为 0 表示正常退出，非 0 则表示异常退出。在 Linux 的 shell 中有一个特殊的变量 `$?` 保存着上一次命令的返回码。我们可以看一个小例子：

```

$ cat exitcode.c /* 这是仅仅一行的极简C程序，它的返回值和输入参数的个数相关 */
int main(int argc, char* argv[]) { return (argc - 1); }
$ gcc -Wall exitcode.c -o exitcode
$ ./exitcode           /* 输入的参数个数为0，则返回值为0 */
$ echo $?
0
$ ./exitcode a         /* 输入的参数个数为1，则返回值为1 */
$ echo $?
1
$ ./exitcode a b c    /* 输入的参数个数为3，则返回值为3 */
$ echo $?
3

```

我们看到了，main() 函数的返回值就是返回码，被 \$? 这个特殊变量所记录。PostgreSQL 并不知道你的归档命令要做什么事情，它只有一个简单的判断标准：如果归档命令返回码是 0，则 PostgreSQL 认为归档成功了，返回码是非 0，则归档失败。我们看一段源代码：

```
/* in src/backend/archive/shell_archive.c */
static bool shell_archive_file(ArchiveModuleState *state, const char *file, const char *path)
{
    char      *xlogarchcmd;
    int       rc;
    .....
    ereport(DEBUG3, (errmsg_internal("executing archive command \"%s\"", xlogarchcmd)));
    rc = system(xlogarchcmd);
    if (rc != 0) {
        .....
        return false;
    }
    .....
    return true;
}
```

由上面的代码可知，PostgreSQL 实际上调用的是 system() 这个系统调用 (system call) 来执行你在 archive_command 参数中指定的命令。这个函数的输入参数是一个字符串，这个字符串就是要执行的 shell 命令。譬如，system("ls -l /tmp") 就等效于我们手工执行 ls -l /tmp 的命令。这个系统调用不难理解，建议读者自行查阅一下这个系统调用的具体细节，加深对它和 PostgreSQL 归档机制的认知。系统调用 system() 的返回值就是被执行的命令的返回码。所以你可以根据你的需求准备归档命令，这个命令的具体内容 PostgreSQL 并不知道，也不操心，用户必须要保证归档命令执行成功后一定要返回 0，执行不成功就返回非 0 值。这就给了用户极大的灵活性，用户往往使用 shell/python/perl 等编写一个比较复杂的脚本来执行 WAL 文件的归档动作。

Linux 操作系统提供了两个哑命令：true 和 false。它们什么也不做，命令 true 的返回码为 0，命令 false 的返回码为 1。下面的实验展示了这两个哑命令的用法：

```
$ /usr/bin/true
$ echo $? /* 检查返回码，结果为0 */
0
$ /usr/bin/false
$ echo $? /* 检查返回码，结果为1 */
1
```

如果我们并不关心归档是否成功，仅仅是希望骗过 PostgreSQL，可以使用这两个哑命令来达到我们的目的。后面我们在恢复过程中会看到这两个小工具的用法。

上述实验中使用的归档命令来自 PostgreSQL 的官方文档给出的示例。它的作用一目了然：首先判断归档目录下是否有即将被拷贝的文件，如果没有就把 pg_wal 目录下的 WAL 文件拷贝过去，避免了文件覆盖的问题。在归档命令中有两个特殊的变量 %p 和 %f 是经常使用的，其中 %p 表示存放在 pg_wal 里面的 WAL 文件，而 %f 只是表示 WAL 文件本身，不带目录信息。假设要归档的 WAL 文件是 000000010000000000000003，则 %f = 000000010000000000000003，而 %p = pg_wal/000000010000000000000003。PostgreSQL 主进程在启动阶段会把它的当前工作目录 (current working directory) 切换到数据库集群目录 \$PGDATA，请参考源码文件 miscinit.c 中的 ChangeToDataDir() 函数。所以由主进程派生出的子进程的当前工作目录均指向了 \$PGDATA，%p 就指向了 \$PGDATA 下的 pg_wal 目录中的当前要操作的 WAL 文件。在上述源码中，变量 xlogarchcmd 包含了具体的归档

命令。当执行归档时，PostgreSQL 会在日志中显示这个变量里面的值，但需要在 DEBUG3 级别才能显示。你可以设置参数 `log_min_messages = debug3`，然后手工执行一次 `pg_switch_wal()` 切换一下 WAL 文件，就会在日志中看到具体的归档命令了。对比你设置的 `archive_command` 的参数，你就对 %p 和 %f 的含义有了更深入的理解。这两个变量经常使用，希望读者能够区分两者的含义。

归档模式是数据库物理备份的前提，非归档模式下做的物理备份是无效的，因为它恢复所需要的 WAL 文件可能丢失了。相对于归档模式，非归档模式在性能上并没有大的提高，却带来了数据安全的隐患，所以我们应该把所有包含重要数据的数据库都变成归档模式，并且定期对数据库做备份。如果数据库里的数据即使丢失也很容易从别的地方拿到，则产生的归档却没有任何用处，白白占用大量磁盘空间，这时就可以使用非归档模式，或者归档模式下使用 `/usr/bin>true` 来欺骗 PostgreSQL。

4.1.2 使用底层函数完成物理备份

我们做物理备份时，往往使用专门的备份工具来进行。这些备份工具有本质上是调用 PostgreSQL 提供的备份函数来进行的，我们当然也可以直接使用这些底层的备份函数手工完成备份。用底层的备份函数的方式现在已经不被鼓励使用了，因为它相对繁琐，自动化程度不高，不适应现在运维自动化的需要了。但为了深刻理解物理备份的具体过程和内幕，使用底层备份函数的方式依然有非常重要的学习价值。下面我们就使用底层的备份函数来完成物理备份。PostgreSQL 提供的底层备份函数有两个：`pg_backup_start()` 和 `pg_backup_stop()`，利用这两个函数做备份的过程如参考图 4.4 所示：

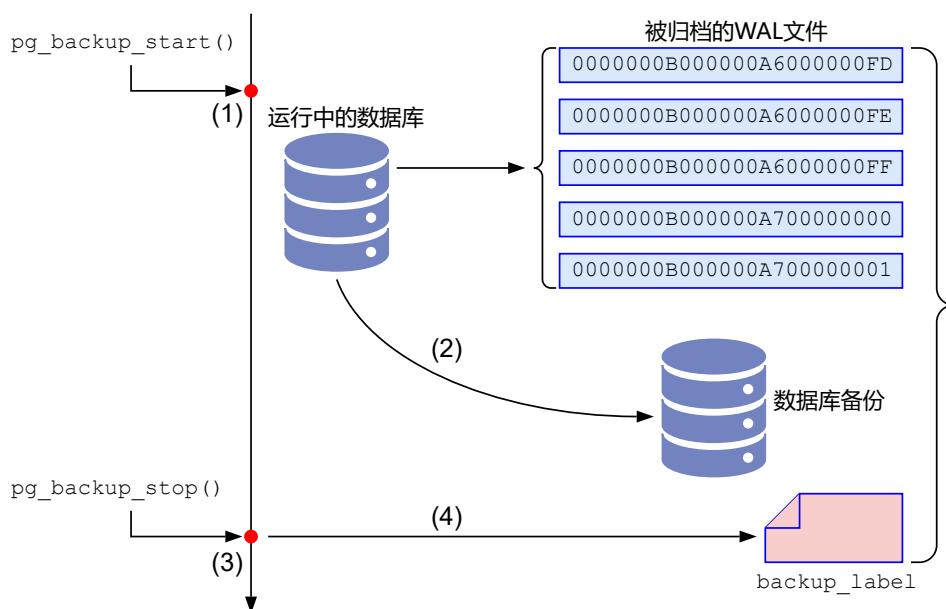


图 4.4：使用底层函数完成物理备份的步骤

使用底层函数完成物理备份的整个过程实际上非常简单，一共分为四步。注意：这四步的执行顺序不能够搞混，下一步必须在上一步执行成功的前提下才能执行，否则产生的备份可能无效。整个备份的过程如下：

- 步骤 1：通过 `psql` 以超级用户的身份执行 `pg_backup_start()` 函数，表示备份的开始。
- 步骤 2：使用诸如 `tar`, `cp`, `rsync` 等任何你喜欢的拷贝命令把整个数据库集群目录备份到某个安全的地方。
- 步骤 3：在步骤 1 的 `psql` 会话中，执行 `pg_backup_stop()` 函数结束备份。注意：必须在同一个 `psql` 会话中执行。如果此时步骤 1 中的会话连接断开了，则本次备份失败。
- 步骤 4：把步骤 3 中 `pg_backup_stop()` 函数的返回内容写进一个 `backup_label` 的文本文件中。

通过以上四步，我们得到了三样东西：一个备份到某个地方的数据库，一个 `backup_label` 的文本文件，其中保存了某个重做点作为恢复的起点，还有在备份期间产生的归档 WAL 文件，它需要从 `backup_label` 中指定的重做点开始连续保存，不能够中断。这三样东西都是一套物理备份不可或缺的组成部分，要妥善保管好，才能完成未来可能的数据库恢复工作。下面的实验展示了具体的执行过程：

```
/* 执行pg_backup_start()命令 */
postgres=# SELECT pg_backup_start('bk1', fast=>true);
pg_backup_start
-----
0/2000028      /* <-- 这个返回的LSN就是重做点 */
(1 row)

postgres=# \! mkdir /home/postgres/bk      /* 创建一个备份目录 */
/* 把数据集集群中所有的文件和目录都拷贝到备份目录中。这一步耗时最长，取决于数据库集群的大小 */
postgres=# \! cp -R $PGDATA/* /home/postgres/bk
/* 在和pg_backup_start()同一个session中执行pg_backup_stop()来结束备份 */
postgres=# SELECT pg_backup_stop(false);
pg_backup_stop
-----
(0/2000138,"START WAL LOCATION: 0/2000028 (file 00000001000000000000000002)+"
CHECKPOINT LOCATION: 0/2000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-12-28 17:24:37 MST
LABEL: bk1
START TIMELINE: 1
",")
(1 row)
```

备份函数 `pg_backup_start()` 有两个输入参数，第一个是字符串，就是本次备份的一个标识，可以写上你能理解的内容。该函数要执行一个检查点，第二个参数 `fast` 就是指定要立刻开足马力执行一个检查点 (`=true`)，还是等着 `checkpoint_timeout` 等参数触发的检查点完成后再返回 (`=false`)。很显然，`fast=true` 会加速 `pg_backup_start()` 函数返回的速度，但会造成磁盘 I/O 负荷的突然加大。只有在检查点操作完成之后，我们才能够进入到第二步。

步骤 2 的过程平淡无奇，就是使用 `cp` 命令把数据库集群中的所有文件都拷贝走，你也可以使用 `tar` 命令。如果你还有别的表空间，也需要把这些表空间的内容都拷贝到备份目录。这一步耗时最长，且在这一步的过程中，执行 `pg_backup_start()` 函数的那个数据库连接不能中断，否则备份就算是失败了。

步骤 3 执行 `pg_backup_stop()` 必须在步骤 1 的 `psql` 会话中执行。如果此时步骤 1 的 `psql` 连接中断，则备份是失败的，需要从步骤 1 重新开始。该函数只有一个输入参数 `wait_for_archive`，该参数为 `false` 则表示立刻返回，不需要等待当前的 WAL 文件被归档。函数 `pg_backup_stop()` 返回了一系列信息，我们必须把它拷贝下来，稍微编辑一下，保存在一个叫 `backup_label` 的文件中，如下所示：

```
postgres=# \! cat /home/postgres/bk/backup_label
(0/2000138,"START WAL LOCATION: 0/2000028 (file 00000001000000000000000002)"
CHECKPOINT LOCATION: 0/2000060
BACKUP METHOD: streamed    /* <-- stream表示是联机备份，即热备份 */
BACKUP FROM: primary      /* <-- 表示该备份是在主库上完成的 */
START TIME: 2023-12-28 17:24:37 MST
LABEL: bk1
START TIMELINE: 1
```

这个小小的文本文件的内容非常容易理解：第一行记录了这个备份的恢复起点，就是一个重做点。第二行记录了包含这个重做点的检查点的 WAL 记录的位置。第三行表示备份方法。第四行表示本次备份是从主库 (primary) 上进行的。第五行是备份的起始时间。第六行是备份的名称。第七行是备份开始的时间线，时间线这个概念我们后续讨论。这里要注意，需要把 pg_backup_stop() 函数的输出稍微剪辑一下，剪辑的要点是：每行的最后一个非空格字符后面必须是一个回车符，不能有空白字符。这里的原因请参考如下代码：

```
/* in src/backend/access/transam/xlogrecovery.c */
static bool read_backup_label(...)

{
    .....
    char ch;
    .....
    /* ch读取一行中的最后一个字符，它必须是'\n'，即回车符 */
    if (fscanf(lfp, "START WAL LOCATION: %X/%X (file %08X%16s)%c",
               &hi, &lo, &tli_from_walseg, startxlogfilename, &ch) != 5 || ch != '\n')
        ereport(FATAL, (errcode(ERRCODE_OBJECT_NOT_IN_PREREQUISITE_STATE),
                        errmsg("invalid data in file \"%s\"", BACKUP_LABEL_FILE)));
    .....
}
```

因为源代码中有一个判断右括号的下一个字符必须是回车符，所以你编辑 pg_backup_stop() 函数的输出时要小心翼翼。为什么要搞这么麻烦呢？其实在 PostgreSQL 15 版本之前，pg_backup_start() 函数会自动帮你产生 backup_label 文件。但是从 15 版本开始，不鼓励大家使用底层函数进行备份了，所以就搞得繁琐一些，逼迫你不要使用这种原始的备份方法。但为了深入理解备份的内幕，我们还得按这种方式做物理备份。你在做这个实验的时候，可以用后文中的备份工具 pg_basebackup 产生的 backup_label 文件作为模版。下面我们就来了解这两个函数幕后到底做了什么工作。

执行 pg_backup_start() 函数是物理备份的第一步，它幕后执行了的动作在 xlog.c 中的 do_pg_backup_start() 函数中。总结一下，这个函数的动作是：首先是强制进入全页写模式，然后执行一次 WAL 文件的切换。等 WAL 文件切换完成后，开始执行检查点操作，检查点执行的方式根据第二个输入参数 fast 是 true 还是 false 来决定尽快执行还是慢悠悠地执行。检查点执行完毕后，记录检查点和重做点的 LSN，这些是 backup_label 文件的头两行的内容，供后面的 pg_backup_stop() 函数使用。这就是为什么 pg_backup_stop() 函数必须在 pg_backup_start() 函数同一个会话中执行的原因。pg_backup_start() 的返回结果是一个 LSN，这个 LSN 就是第三步执行的检查点对应的重做点，你可以看到它和 pg_backup_stop 返回结果中“START WAL LOCATION”是相等的。函数 pg_backup_start() 在其触发的检查点操作执行完毕后才返回。这个过程可能有一定的时间，我们必须等待该函数返回后才能进入执行拷贝命令的第二步，目的就是确保拷贝动作一定在检查点执行成功后才能进行。

结束物理备份的函数 pg_backup_stop() 做的工作在 xlog.c 中的 do_pg_backup_stop() 函数中。它的主要工作内容是：第一步把强制的 FPW 模式恢复到 pg_backup_start() 函数之前的状态，然后在 WAL 文件中写入一个备份结束(BACKUP_END) 的 WAL 记录，然后切换 WAL 文件，然后根据 pg_backup_start() 函数中记录的检查点信息和自己获取的备份结束信息，构造 backup_label 文件的内容，并显示在屏幕上。

pg_backup_stop() 还会创建一个备份的历史文件。备份历史文件和 backup_label 文件一样，记录了恢复所需要的重做等信息，还包含了备份的结束信息。这个文件的目的主要是用于记录备份的历史，并不用于数据库的恢复工作。它的文件名分为三部分：WAL 文件的文件名加上重做点的位置，加上 backup，这三部分用点符号分割，我们来看一下它的内容。

```
$ ls -l $PGDATA/pg_wal
total 49160
-rw----- 1 postgres postgres 16777216 Dec 28 17:24 00000001000000000000000000000000
-rw----- 1 postgres postgres 16777216 Dec 28 17:24 00000001000000000000000000000002
```

```

-rw----- 1 postgres postgres      316 Dec 28 17:24 0000000100000000000000002.0000028.backup
-rw----- 1 postgres postgres 16777216 Dec 28 17:27 0000000100000000000000003
drwx----- 2 postgres postgres     4096 Dec 28 17:24 archive_status
/* 查看这个备份历史文件的内容 */
$ cat $PGDATA/pg_wal/000000010000000000000002.0000028.backup
START WAL LOCATION: 0/2000028 (file 000000010000000000000002)      /* <-- 备份起点的LSN */
STOP WAL LOCATION: 0/2000138 (file 000000010000000000000002)      /* <-- 备份终点的LSN */
CHECKPOINT LOCATION: 0/2000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-12-28 17:24:37 MST
LABEL: bk1
START TIMELINE: 1
STOP TIME: 2023-12-28 17:24:52 MST
STOP TIMELINE: 1

```

从备份历史文件中我们可以看到，备份的起点的 LSN 是 0/2000028。因为 `pg_backup_start()` 会触发一个检查点操作。这个 LSN 就是该检查点的起点，也是未来恢复这个备份的起点。第二行中的 STOP WAL LOCATION，顾名思义，就是备份结束的中终点，它的 LSN 是 0/2000138。下面我们使用 `pg_waldump` 来观察一下备份终点的 WAL 记录：

```

$ pg_waldump -p /home/postgres/a -n 1 -s 0/2000138
rmgr: XLOG      len (rec/tot):    24/    24, tx:          0,
lsn: 0/02000138, prev 0/02000110, desc: SWITCH

```

这是一条表示切换 WAL 文件的 WAL 记录，我们看看这条切换 WAL 记录之前的那条 WAL 记录，它的 LSN 是 0/02000110，我们执行如下命令：

```

$ pg_waldump -p /home/postgres/a -n 1 -s 0/02000110
rmgr: XLOG      len (rec/tot):    34/    34, tx:          0,
lsn: 0/02000110, prev 0/020000D8, desc: BACKUP_END 0/2000028

```

我们看到，在备份结束的时候，会插入一条 `BACKUP_END` 的 WAL 记录表示备份已经结束了。它的长度是 34 个字节，和提交类型的 WAL 记录基本相同，扣除 24 个字节的 `XLogRecord` 和 2 个字节的 `XLogRecordDataHeaderShort` 结构，剩下了 8 个字节，这 8 个字节记录了备份起点的 LSN，从上面可以看出，这个值是 0/2000028。我们可以用图 4.5 来表示和备份相关的 WAL 记录。

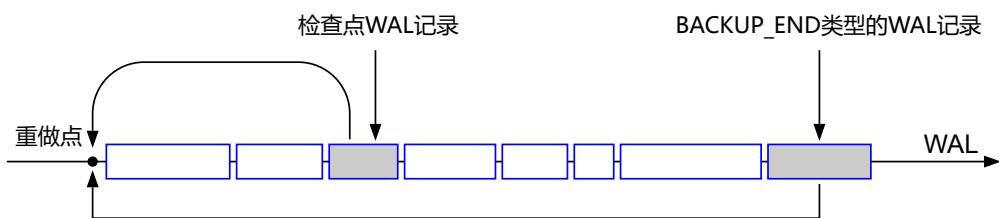


图 4.5：备份的起点和终点

从上图中可以看出，函数 `pg_backup_start()` 会触发一个检查点操作，这个检查点的重做点就是恢复的起点。等拷贝结束以后，函数 `pg_backup_stop()` 会插入一条 `BACKUP_END` 的 WAL 记录，这条 WAL 记录是备份的终点，且在该条记录里面包含了重做点的信息。我们使用这个备份做数据库恢复时，恢复进程必须把回放进度推

进到备份终点的右边，数据库才能处于一致状态，此时数据库才能被打开，供用户访问。我们可以使用图 4.6 来理解数据库达到一致性状态的条件：

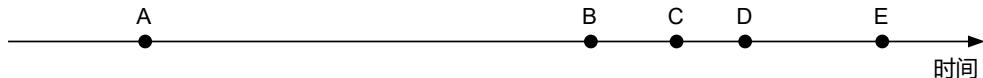


图 4.6：数据库恢复到一致性状态的条件

如上图所示，假设我们在 A 点开始执行备份的，在 E 点结束了备份，所以重做点的 LSN 是 A 点，`BACKUP_END` 的 WAL 记录的 LSN 是 E 点。在进行数据库恢复的过程中，我们必须从 A 点开始回放 WAL 记录，一直到 E 点或者右边更远的位置，数据库才能够达到一致性状态，才能够被访问。这是为什么呢？我们拷贝的动作发生在 A 点和 E 点之间。拷贝的过程中可能会拷贝到坏块。但是这些坏块必然有 FPW 类型的 WAL 记录，且它们的位置必然在于 A 点和 E 点之间。当我们回放到 E 点后，所有坏块的 FPW 记录已经被回放了，即所有的坏块已经被修正了。此时数据库没有任何坏块了，可以打开了。如果我们只恢复到 A 点和 E 点之间的某一个点，譬如 B 点，会发生什么情况呢？假设我们在 D 点拷贝某个数据块时，发生了部分写，我们拷贝到的该数据块是坏块。它对应的 FPW 的 WAL 记录必然在 D 点之前被写入到了 WAL 文件中，譬如 C 点。因为我们只恢复到了 B 点，所以无法用 C 点的 FPW 记录来修正该数据块，此时打开数据库，必然存在坏的数据块。通过以上分析可知，数据库在恢复的过程中，必须把 WAL 记录回放的位置推进到 `BACKUP_END` 记录的位置或者右边更远的位置，数据库才能不包含损坏的数据块，这是数据库一致性的标准。所谓数据库达到一致性，指的是备份数据库恢复出来的数据库和原来的数据库在 `BACKUP_END` 这个位置是一致的。

`backup_label` 是一个很小的文本文件，但是非常重要，因为它记录着这个备份进行恢复的起点。如果 `backup_label` 文件丢失了，我们依然可以通过备份历史文件中的信息创建出一个新的 `backup_label` 文件，因为 `backup_label` 文件无非就是一个简单的文本文件嘛。所以历史备份文件应该保留，不要轻易删除掉，紧急时刻它可以救命。所有的备份工具在底层都是调用这两个函数来完成备份的。使用底层函数备份的用途往往是为了深入研究备份的内部细节，在实际中使用的并不常见。但是对于它的深入理解有助于我们排查各种备份和恢复相关的错误。

4.1.3 使用 pg_basebackup 进行备份

在实际的备份工作中，我们往往使用各种各样的备份工具进行备份。其中 PostgreSQL 自带了一个备份工具 `pg_basebackup`，因为它是官方内置的，无需额外安装，所以得到了广泛的应用，本节我们就来介绍一下如何使用该备份工具。

4.1.3.1 两种不同的网络协议

当类似 `psql` 这样的客户端连接到数据库实例时，PostgreSQL 采用的是普通的 `libpq` 协议。但是为了执行备份和 WAL 记录的传输，PostgreSQL 支持第二种协议，叫做复制协议 (replication protocol)，请参考图 4.7：

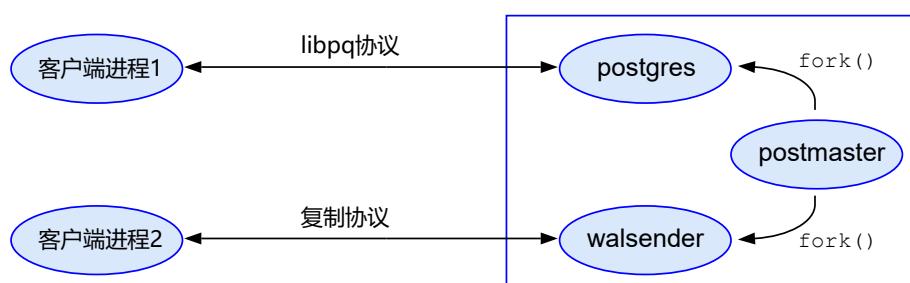


图 4.7：两种不同的协议

我们可以看到，客户端 1 采用普通的协议和数据库实例连接，在服务器这一侧由后端进程 `postgres` 与之对接。客户端 2 则采用复制协议和数据库实例对话，在服务器端由一个新的后台进程 `walsender` 与之对接，后文中统一把 `walsender` 进程称为“WAL 发送进程”。我们做一个实验就能明白具体的含义。假设数据库服务器的 IP 地址是 192.168.137.12，我们创建两个用户 `alice` 和 `bob`，分别用着两个用户从另外一台机器上进行远程登录，比较其中的差异。

```
$ psql
psql (16.0)
Type "help" for help.

/* 创建用户的第一种方法，使用CREATE USER的命令 */
postgres=# CREATE USER alice WITH PASSWORD 'Alice123';
CREATE ROLE
postgres=# ALTER USER alice REPLICATION; /* *赋予alice用户REPLICATION的权限 */
ALTER ROLE
$ createuser -P bob /* 创建用户的第二种方法，使用createuser工具 */
Enter password for new role:
Enter it again:
```

创建完这两个用户后，我们还需要在数据库集群的 `pg_hba.conf` 中增加一行，允许客户端以复制协议登录进来。具体内容如下所示：

host	all	all	192.168.137.0/24	md5
host	replication	all	192.168.137.0/24	md5

上面两行的意思是：允许来自 192.168.137 网段的客户端以 libpq 和复制协议登录本数据库集群，但是要输入密码。修改完毕后，执行 `pg_reload_conf()` 函数使得上面的配置生效，然后我们就可以测试两种不同方式的登录了。

```
$ psql -h 192.168.137.12 -d oracle -U bob /* 以bob用户登录数据库服务器 */
Password for user bob:
psql (16.0)
Type "help" for help.

/* 查看当前的连接信息 */
oracle=> \conninfo
You are connected to database "oracle" as user "bob" on host "192.168.137.12" at port "5432".
oracle=> IDENTIFY_SYSTEM;
ERROR:  syntax error at or near "IDENTIFY_SYSTEM"
LINE 1: IDENTIFY_SYSTEM; /* 这个命令服务器不认 */

oracle=> SELECT pg_backend_pid(); /* 查看和我这个客户端对接的后端进程的进程号 */
pg_backend_pid
-----
1732
(1 row)

/* 保持第一个连接的同时，我们使用alice用户以复制协议协议登录后台 */
$ psql "replication=yes host=192.168.137.12 user=alice dbname=oracle"
Password for user alice:
psql (16.0)
Type "help" for help.

/* IDENTIFY_SYSTEM这条命令可以被成功执行，它返回系统的标识符 */
oracle=> IDENTIFY_SYSTEM;
```

systemid	timeline	xlogpos	dbname
7287369332113755196	1	0/90001C0	

(1 row)

使用复制协议连接数据库集群的要点是在连接字符串中指定 `replication=yes`，数据库集群看到这个参数后就明白客户端想使用复制协议，而不是 libpq 协议进行通讯。通过对比这两个不同的连接的表现差异，我们可以看到：Bob 以普通的 libpq 协议登录远端的数据库服务器，无法执行 `IDENTIFY_SYSTEM` 命令，而 Alice 使用特殊的复制协议就可以顺利地执行 `IDENTIFY_SYSTEM` 这条命令。原因是该命令是复制协议特有的，其目的是获得控制文件中的系统标识。在保持 bob 和 alice 的远程连接的同时，我们在后台查看相关进程，结果如下：

```
$ ps -ef | grep -E 'bob|alice' | grep -v grep
postgres 1732 1706 0 08:00 ? 00:00:00 postgres: bob oracle 192.168.137.1(49436) idle
postgres 1781 1706 0 08:12 ? 00:00:00 postgres: walsender alice 192.168.137.1(49364) idle
```

上面的结果很清楚地表明：当客户端使用普通协议连接时，数据库集群有一个后端进程 `postgres` 和其对接。当客户端使用复制协议连接时，数据库集群使用了一个新的后端进程，即 `WAL` 发送进程和其对接。

4.1.3.2 pg_basebackup 的基本使用

备份工具 `pg_basebackup` 采用复制协议和数据库集群连接。它既可以在数据库服务器端使用，也可以远端连接到数据库服务器，非常灵活。下面我们学习一下如何使用该备份工具。请在任何一台包含 `pg_basebackup` 软件的机器上执行如下命令：

```
$ pwd
/home/kevin/bk
$ ls -l
total 0
$ pg_basebackup -h 192.168.137.12 -U alice -D bk1 -P
Password:
29758/29758 kB (100%), 1/1 tablespace
$ ls -l
total 4
drwx----- 19 kevin kevin 4096 Oct  8 09:57 bk1
```

在 `pg_basebackup` 的输入参数中，`-h` 表示连接到哪台服务器上，`-U` 表示使用哪个用户，`-P` 表示显示备份的进度信息，在数据库比较大的时候，这个参数可以让我们知道正在进行的备份进度，非常实用。参数`-D` 表示要把备份保存在哪里，注意，这个`-D` 表示本地的一个目录，它可以不存在，`pg_basebackup` 会自动创建它。如果它已经存在，要保证里面没有任何文件。请不要把这个`-D` 和 `pg_ctl/initdb` 等服务器端工具使用的`-D` 混淆，两者不一样：一个是服务器端的数据库集群的目录，一个是客户端的用于保存备份的目录。我们看到，使用具备 `replication` 权限的用户 `alice`，我们成功地把数据库备份到了本地的 `bk1` 目录中。

我们希望备份工具产生的备份是“自给自足”的，就是这个备份是完整且独立的，不依赖外部任何信息就可以成功完成数据库的恢复任务，恢复完成后，数据库可以打开供用户使用。通过上一节使用底层函数来执行备份的学习，我们知道：一个完整的备份有三个重要的组成部分：数据库本身的备份，重做点和 `WAL` 文件。数据库本身的备份的个头最大，往往几百 GB 或者几个 TB。重做点被保存在 `backup_label` 文件中。从该重做点开始的连续不断的 `WAL` 文件，必须让该备份能够恢复到一致的状态。如果要做到自给自足，这三样东西一个都不能少。保存重做点的 `backup_label` 文件非常小，就几百个字节，很容易搞定，你可以到 `bk1` 目录下看一下，里面有一个 `backup_label` 文件。从重做点开始连续不断地 `WAL` 文件到哪里才是个头呢？我们知道备份结束后会插入一个备份结束(`BACKUP_END`)的 `WAL` 记录。只要你手里的 `WAL` 记录从 `backup_label` 里规定的重做点开始，到

BACKUP_END 的 WAL 记录之间的 WAL 记录连续不断，就可以保证数据库在恢复后能够打开。pg_basebackup 使用参数-X 来拷贝从检查点开始的 WAL 记录，直至备份结束 WAL 记录为止。下面我们做第二个实验来了解这个参数的用法。依然在同一个目录下执行如下命令：

```
$ pg_basebackup -h 192.168.137.12 -U alice -D bk2 -P -X stream
Password:
29758/29758 kB (100%), 1/1 tablespace
$ ls -l
total 8
drwx----- 19 kevin kevin 4096 Oct  8 09:57 bk1
drwx----- 19 kevin kevin 4096 Oct  8 10:05 bk2

$ cat bk2/backup_label
START WAL LOCATION: 0/14000028 (file 0000000100000000000000014)
CHECKPOINT LOCATION: 0/14000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-10-08 11:05:26 MDT
LABEL: pg_basebackup base backup
START TIMELINE: 1
$ ls -l bk2/pg_wal/
total 16388
-rw----- 1 kevin kevin 16777216 Oct  8 10:05 0000000100000000000000014
drwx----- 2 kevin kevin     4096 Oct  8 10:05 archive_status
```

我们看到了，在新的备份 bk2 的目录下，backup_label 指出：如果要恢复该备份，需要从 0/14000028 这个重做点开始，它在 WAL 文件 0000000100000000000000014 中。然后我们在 pg_wal 目录中发现了该文件。这表明一个完整备份的三件套已经齐备了，不再依赖任何外部的信息就可以通过 bk2 目录里面的文件恢复出一个可以正常运行的数据库。当然了，因为 bk2 只有 0000000100000000000000014 这一个 WAL 文件，备份终点的 BACKUP_END 记录必然也在这个文件中，你可以使用 pg_waldump 工具检查这个文件中是否包含 BACKUP_END 记录。

pg_basebackup 通过远程连接执行备份时，我们面临一个问题：产生的备份数据存放在哪里？上面的两个例子都是把备份保存到了客户端。实际的数据库体积都比较大，譬如我目前支持的数据库有 11TB。这样巨量的数据如果要保存到客户端，必然存在服务器和客户端之间大量的网络传输流量。如果保存在服务器端，pg_basebackup 只是在客户端进行远程操控，就不存在这个问题，下面我们做第三个备份的实验：

```
$ pwd /* 首先在服务器端，确保/home/postgres/backup存在 */
/home/postgres/backup
/* 以超级用户postgres登录psql，把pg_write_server_files权限赋给alice */
postgres=# GRANT pg_write_server_files TO alice;
GRANT ROLE
/* 在客户端执行如下命令 */
$ pg_basebackup -h 192.168.137.12 -U alice -P -X fetch -t server:/home/postgres/backup/bk3
Password:
46144/46144 kB (100%), 1/1 tablespace
$ ls -l
total 8
drwx----- 19 kevin kevin 4096 Oct  8 09:57 bk1
drwx----- 19 kevin kevin 4096 Oct  8 10:05 bk2
$ pwd          /* 然后跑到服务器端，查看备份bk3被创建了 */
/home/postgres/backup/bk3
```

```
$ ls -l
total 46324
-rw----- 1 postgres postgres 181409 Oct  8 11:16 backup_manifest
-rw----- 1 postgres postgres 47251456 Oct  8 11:16 base.tar
```

在这个例子中，用户 alice 要在服务器端创建目录 bk3，所以必须要有 pg_write_server_files 的权限。使用-t server:xxx 就是告诉备份工具：请把该备份保留在服务器上，无需通过网络传输到客户端，这样就避免了大量的网络流量。我们还看到 bk3 和 bk1/bk2 不同，它把所有的备份都压缩成了一个.tar 文件。你也可以通过-F 参数指定，也可以使用-z 参数指定压缩，具体请使用 pg_basebackup --help 来查看各种选项的含义和用法。

通过以上三个简单但是实用的备份例子，我们初步掌握了 pg_basebackup 的基本使用。在第二个实验和第三个实验，我们都使用了-X 参数来打包相关的 WAL 文件，一个是流模式 (stream)，一个是抓取模式 (fetch)，两者有什么不同呢？实际上对 WAL 文件的抓取可以分为三种模式：none 表示不抓取 WAL 文件，fetch 和 stream 都会抓取 WAL 文件，这两者的区别可以用下面两张图表示：

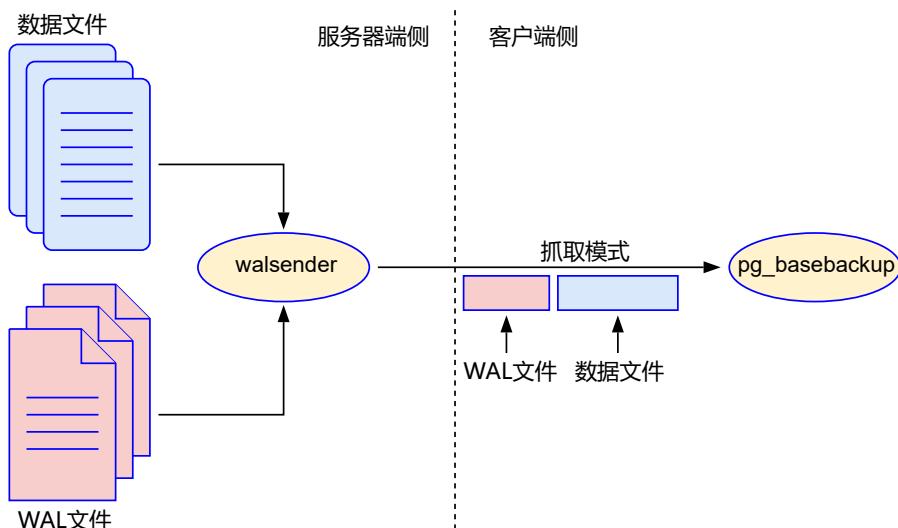


图 4.8: pg_basebackup 的抓取模式

图 4.8 表示的是抓取模式，在抓取模式下，pg_basebackup 和数据库实例只有一个网络连接，数据文件和 WAL 文件都走这个通道，而且是先抓取数据文件，后抓取 WAL 文件。这种模式可能存在一个问题：因为拷贝数据文件需要很长时间，等数据文件拷贝完以后，相关的 WAL 文件可能已经被 PostgreSQL 从 pg_wal 目录中删除掉了，从而导致抓取 WAL 文件失败。这个时候可以使用-C 参数创建一个复制槽 (replication slot) 来确保在备份完成之前，相关的 WAL 文件不会被删除。关于复制槽的知识，我们放在下一章进行详细讨论。

图 4.9 表示的是流模式，在流模式下，pg_basebackup 会启动一个子进程，专门负责抓取 WAL 文件，而主进程负责抓取数据文件。子进程通过管道把 WAL 文件传送给父进程。这种方法确保抓取 WAL 文件和巨大的数据文件同步进行，避免了抓取模式存在的问题，所以它成为最常见的模式。

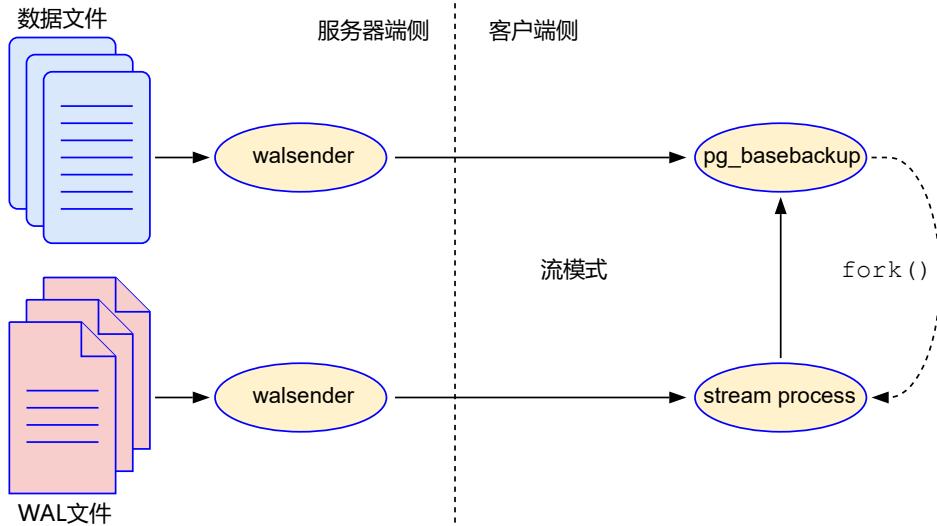


图 4.9: pg_basebackup 的流模式

在我们使用-t server:/the/location/to/backup 的参数时, pg_basebackup 就直接把备份保存到服务器端, 避免了大量的网络传输, 如图 4.10 所示:

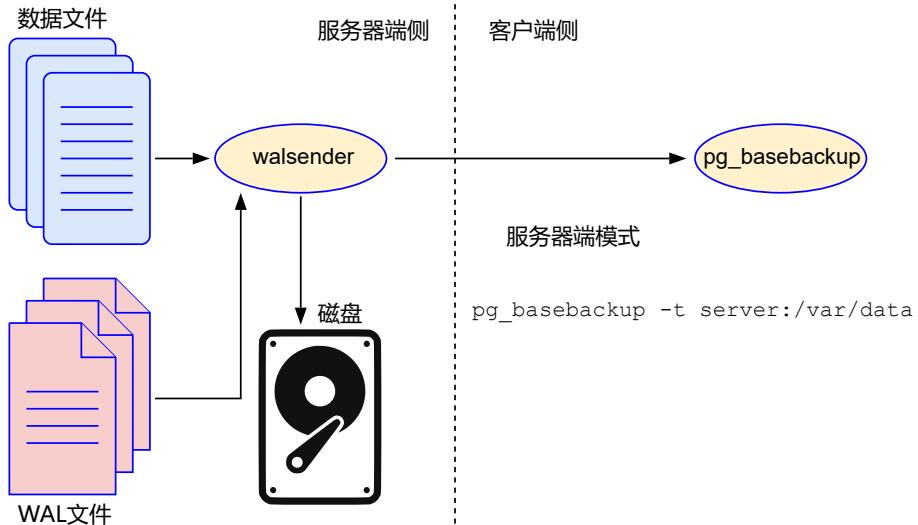


图 4.10: pg_basebackup 的服务器备份模式

很显然, 在服务器备份模式下, 没有必要使用 stream 模式抓取 WAL 文件到客户端, 所以此模式下只能使用 fetch 模式来备份 WAL 文件。在实际备份的工作中, 基本上是在服务器端搞一个 NFS 类型的大磁盘进行备份, 而且实际的数据库往往很大, 完成一次备份可能需要好几个小时, 甚至更长时间, 这个时候你可以使用 nohup 命令把 pg_basebackup 运行在后台, 这样就不用担心网络断开的问题了。关于 nohup 的用法请自行在互联网上搜索。

4.1.3.3 备份有效性的验证

备份完成后, 我们自然关心一个问题: 这个备份是否有效。检验一个备份是否有效的终极解决方法就是用这个备份进行数据库恢复。如果能够成功恢复, 则表明这个备份是有效的。在实际工作中, 我们要经常操练备份的恢复工作, 确保在灾难发生后, 企业宝贵的数据不会丢失。但恢复工作很耗时, 譬如我负责的一个 5TB 的数据库, 它的备份是保存在 AWS 云的 S3 存储中。从下载备份到完成恢复, 需要 2 个小时或者更长时间。所以

用户不可能每次备份后都要进行恢复检验。如果有一些工具能够验证验证备份的有效性，毫无疑问是非常好的事情，`pg_verifybackup` 就是一个检验备份有效性的实用工具。注意，`pg_verifybackup` 只是“在一定程度上”检验备份的有效性，它并不能百分百确保备份可以成功恢复，所以你不能完全依赖这样的工具，最终还是要以备份能否恢复作为备份有效性的终极判断标准。不过 `pg_verifybackup` 可以检测大部分磁盘和用户相关的常见备份错误，所以它还是有一定实用价值的，可以做备份和恢复过程中的一个辅助工具。

我们在使用 `pg_basebackup` 产生的备份中会看到一个特殊的文件。这个文件被称为备份清单文件 (`manifest`)。下面我们看一下它的具体内容：

```
$ pwd
/opt/data/bk1
$ ls -l *manifest
-rw----- 1 postgres postgres 181234 Dec  9 15:19 backup_manifest
$ cat backup_manifest | more
{
  "PostgreSQL-Backup-Manifest-Version": 1,
  "Files": [
    {
      "Path": "backup_label",
      "Size": 225,
      "Last-Modified": "2023-12-09 22:19:13 GMT",
      "Checksum-Algorithm": "CRC32C",
      "Checksum": "b9a43fba"
    },
    ...
    {
      "Path": "base/5/3081",
      "Size": 16384,
      "Last-Modified": "2023-10-16 10:43:59 GMT",
      "Checksum-Algorithm": "CRC32C",
      "Checksum": "b65ae9b3"
    },
    {
      "Path": "base/5/2605_vm",
      "Size": 8192,
      "Last-Modified": "2023-10-16 10:43:59 GMT",
      "Checksum-Algorithm": "CRC32C",
      "Checksum": "294a3ca3"
    },
    {
      "Path": "base/5/1255",
      "Size": 802816,
      "Last-Modified": "2023-10-16 10:43:59 GMT",
      "Checksum-Algorithm": "CRC32C",
      "Checksum": "5e2a52ad"
    },
    ...
  ]
}
```

很显然，这个备份清单文件是一个 json 格式的文本文件，里面不仅仅列出了备份的全部文件清单，还列出了每个文件的 CRC32C 校验码，这个就非常有价值了。我们很容易想到，通过核对这个校验码，我们就很容易知道某个文件是否损坏了。`pg_verifybackup` 就是依靠这个备份清单文件来检验备份的有效性。因为这个备份清单文件是由 `pg_basebackup` 产生的，所以 `pg_verifybackup` 只能检验由 `pg_basebackup` 生成的备份。其它备份工具如果不能产生兼容的备份清单文件，`pg_verifybackup` 就无能为力。下面我们就看看这个小工具的使用方法：

```
$ pg_verifybackup bk1
backup successfully verified
```

可以看出，这个小工具的使用非常简单，只要指定备份的目录就行了，最终的输出结果就是有效或者无效两种结果。如果无效，`pg_verifybackup` 会指出哪些文件存在问题。如果 `pg_basebackup` 产生的是 tar 格式的备份，你必须把这个 tar 文件解开成一个目录，才能够使用 `pg_verifybackup`。这个验证工具首先要读取清单备份文件 `backup_manifest`。如果这个文件在别的地方，你可以使用 `-m` 参数指定它的位置。如果找不到这个文件，或者读取后解析失败，`pg_verifybackup` 就报错退出。备份的验证工作分为如下几个步骤。第一个步骤就是拿着清单文件，依次检查清单上列出的每个文件是否存在备份的目录中。如果存在，再检查一下文件大小是否匹配。第二步是读取每个文件的内容，计算该文件的 CRC32 校验码，和清单文件上的校验码对比。这一步是最耗时的步骤。如果用户确信不需要，可以使用 `-s` 参数跳过这一步。第三个步骤是检验 WAL 文件。我们可以在备份清单文件的底部看到如下的信息：

```
"WAL-Ranges": [
  {
    "Timeline": 2,
    "Start-LSN": "0/8000028",
    "End-LSN": "0/8000138"
  }
],
"Manifest-C checksum": "f735f783b6c496e6ca02b71f1ed2c20e83c44aba8912a70702d6d3e338521455"}
```

这部分信息记录了为了让该备份成功恢复，所需要的最少的 WAL 记录的范围，包括 WAL 记录的起点和终点。很显然，开始的 Start-LSN 就是 backup_label 里面的重做点。结束的 LSN 就是能够确保数据库达到一致状态的最小 LSN。pg_verifybackup 会调用 pg_waldump 工具来解析此范围内的 WAL 记录，确保它们是有效且连续的。用户可以使用-n 参数跳过此步骤。参数-w 则告诉 pg_verifybackup 到哪里去寻找 WAL 文件。

因为校验需要一定时间，所以参数-P 可以显示校验的进度，让用户心中有数。如果一个备份能够通过以上几步的校验，十之八九这个备份是有效的。这就是 pg_verifybackup 的价值。当然，你依然需要时不时的做真正的恢复，才能最终确保备份的有效性和可靠性。如何做数据库的恢复是下一节要探讨的内容。

至此，我们基本上掌握了 pg_basebackup 的使用。在下一章我们还会看到，该工具除了做日常的数据库备份以外，还可以很方便地帮助我们在流复制中创建备库。一些公司开发了自己的备份工具，如 WAL-G，Barman 等等。我公司使用的是 WAL-G，这也是一款非常优秀的备份软件，它同时支持备份压缩和加密，可以很方便地把备份上传到亚马逊，微软，谷歌等云平台，确保备份和数据库之间有足够的安全距离，减少被一锅端的可能性。

4.2 数据库的恢复

数据库备份的目的是为了恢复，所以检验一个备份是否有效，终极方法是拿这个备份恢复数据库。数据库恢复的原理和第三章研究的灾难恢复没有任何区别。我们知道任何完整的备份都包含一个重做点，被记录在 `backup_label` 这个小小的文本文件中，这个重做点是恢复的起点，基值加上从起点开始的增量，我们想恢复到哪个时间点都可以。图 4.11 展示的恢复数据库的基本原理：

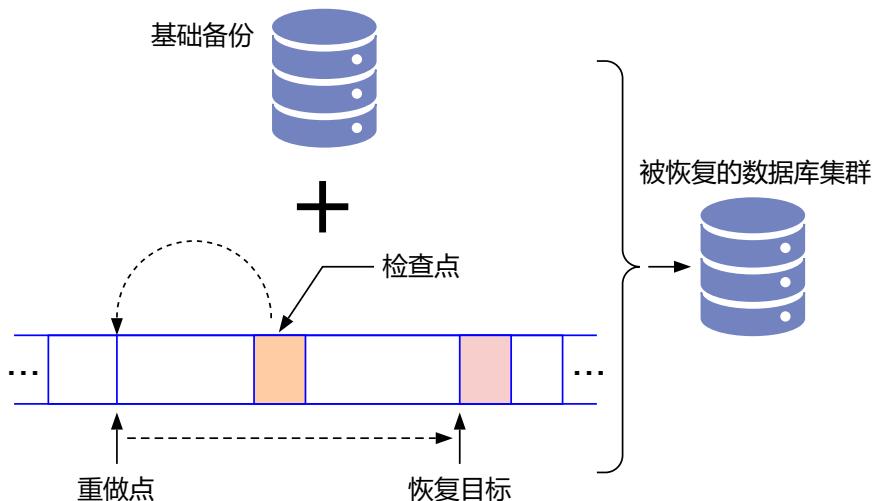


图 4.11：数据库的恢复原理

数据库恢复过程分为两步：先 `restore`，后 `recovery`。这两个术语翻译成中文似乎都是“恢复”的意思，但是它们两者有很大的不同：术语 `restore` 指的是把基础备份（数据文件）拷贝到指定目录，它实际上是一个拷贝的动作。因为实际的数据库非常大，所以这一步往往耗时最长，我们可以称之为“还原”。术语 `recovery` 是在还原完成之后进行的动作，它是从重做点开始，依次读取 `WAL` 文件，对还原操作拷贝的状态不一致的数据文件进行修正。当所有的数据文件都达到一个一致的状态后，数据库就可以被顺利打开，对外营业了。所以我们可以把 `recovery` 称之为“恢复”。`PITR` 是数据库恢复领域另外一个常用术语，它是 `Point-in-Time Recovery` 的缩写，翻译成中文可以叫做“基于时间点的恢复”，它的含义是把数据库恢复到指定的时间点附近的一致状态。恢复数据库往往有这样几个需求：

- 我只求数据库能够尽快进入到一致状态，不求恢复到最新状态。这个恢复是最快的。
- 我指定一个时间点，本质上就是一个 `LSN`，请恢复到该时间点的附近。
- 把数据库恢复到已有的 `WAL` 文件能够达到的尽头，这个时候数据库处于最新的状态，当然恢复也是最慢的。

4.2.1 数据库恢复的实验

基于前一节的备份，无论是使用底层的系统函数的备份，还是使用 `pg_basebackup` 制作的备份都可以，下面我们来进行恢复实验。如果你想把备份恢复到和正在运行的数据库在同一台服务器上，你无需关闭数据库，可以找一块足够大的磁盘，把备份恢复到这里。我们知道：只要端口号不同，两个数据库集群是可以在同一台机器上和平共处的。譬如，一般正在运行的数据库都使用缺省的 5432 端口，我们恢复的数据库可以使用 5433 端口。

下面我们就进行数据库恢复的实验，我们使用上一节存储在服务器上的备份 `bk3` 来进行。首先执行的数据恢复的步骤：

```
postgres@debian:/opt/data$ pwd
```

```

/opt/data
postgres@debian:/opt/data$ ls -l
total 4
drwx----- 19 postgres postgres 4096 Oct  8 12:29 pgdata1
postgres@debian:/opt/data$ mkdir restore
postgres@debian:/opt/data$ cd restore
/* 解压缩备份的tar文件，这一步被叫做还原(restore)*/
postgres@debian:/opt/data/restore$ tar xvf /home/postgres/backup/bk3/base.tar
postgres@debian:/opt/data/restore$ pwd
/opt/data/restore
postgres@debian:/opt/data/restore$ ls -l /* 我们看到一个数据库集群的完整目录 */
total 124
-rw----- 1 postgres postgres 227 Oct  8 11:16 backup_label
drwx----- 6 postgres postgres 4096 Oct  8 10:52 base
.....
drwx----- 3 postgres postgres 4096 Oct  8 12:31 pg_wal
postgres@debian:/opt/data/restore$ cat backup_label /* 查看重做点 */
START WAL LOCATION: 0/17000028 (file 00000001000000000000000017)
CHECKPOINT LOCATION: 0/17000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-10-08 11:16:32 MDT
LABEL: pg_basebackup base backup
START TIMELINE: 1
postgres@debian:/opt/data/restore$ ls -l pg_wal /* 查看对应的WAL文件是否到位 */
total 16388
-rw----- 1 postgres postgres 16777216 Oct  8 11:16 00000001000000000000000017
drwx----- 2 postgres postgres 4096 Oct  8 12:31 archive_status

```

通过 `backup_label` 文件，我们知道该备份的恢复起点是 0/17000028，对应的 WAL 文件是 17 号，被保存到了 `pg_wal` 中了。数据库还原工作做完后，我们就可以进行数据库的恢复 (recovery) 工作了，恢复就是启动数据库，开始拿 WAL 文件对数据文件进行修正。在启动数据库之前，我们需要修改一下主配置文件 `postgresql.conf`，在最后加上如下几行：

```

recovery_target='immediate'
restore_command='/usr/bin/true'
port=5433

```

参数 `recovery_target` 的值是 `immediate`，它告诉 PostgreSQL：一旦数据库达到一致的状态，就终止进一步的恢复。参数 `restore_command` 和备份时候指定的 `archive_command` 是相反的动作，它告诉 PostgreSQL 从哪里把需要的 WAL 文件拷贝到 `pg_wal` 目录中。由于我们现在只有一个 WAL 文件，而且已经保存到了 `pg_wal` 目录中了，无需进一步的拷贝动作，我们就指定哑命令 `/usr/bin/true` 来骗过 PostgreSQL。然后，我们在 `/opt/data/restore` 这个即将被恢复的数据库集群目录下创建按一个空文件 `recovery.signal`，这是一个信号文件，里面的内容无关紧要。根据第三章学习的 PostgreSQL 恢复进程的工作流程，我们可以知道，恢复进程一旦看到这个信号文件，就进入到归档恢复模式。此外，我们还需要将 `restore` 目录的权限改成 700，这是 PostgreSQL 的规定。请参考如下命令：

```

$ pwd
/opt/data/restore
$ touch recovery.signal /* 创建恢复所需要的信号文件 */
$ cd ..

```

```
$ pwd
/opt/data
$ chmod -R 700 restore /* 把数据库集群目录设置为700属性 */
$ ls -l
total 8
drwx----- 19 postgres postgres 4096 Oct  8 12:29 pgdata1
drwx----- 19 postgres postgres 4096 Oct  8 12:50 restore
```

这些准备工作就绪后，我们就可以启动这个新的数据库，执行如下命令：

```
$ pg_ctl start -D /opt/data/restore -l logrestore.log
waiting for server to start.... done
server started
$ tail logrestore.log
.....
2023-10-08 12:55:45.443 MDT [2401] LOG: consistent recovery state reached at 0/17000100
2023-10-08 12:55:45.443 MDT [2401] LOG: recovery stopping after reaching consistency
2023-10-08 12:55:45.443 MDT [2401] LOG: pausing at the end of recovery
2023-10-08 12:55:45.443 MDT [2401] HINT: Execute pg_wal_replay_resume() to promote.
2023-10-08 12:55:45.443 MDT [2398] LOG: database system is ready to accept read-only connections
```

启动日志的最后几行表明：数据库已经进入了一致的状态，你可以使用 `pg_wal_replay_resume()` 来把数据库从恢复状态提升 (promote) 到正常的可读可写的工作状态。下面我们就登录到数据库中，执行如下动作：

```
postgres@debian:/opt/data$ psql -p 5433 /* 需要指定端口，和正常运行的数据库区分开 */
psql (16.0)
Type "help" for help.
/* 连接到oracle数据库中，查看测试表state */
postgres=# \c oracle
You are now connected to database "oracle" as user "postgres".
oracle=# SELECT * FROM state ORDER BY 1;
 id | name
----+---
 0 | Ohio
(1 row)
/* 我们往state表中插入一条新记录，结果失败，因为此时数据库处于恢复中，是只读状态*/
oracle=# INSERT INTO state VALUES(1, 'California');
ERROR: cannot execute INSERT in a read-only transaction
/* 你可以看到startup进程已经完成了17号WAL文件的恢复工作，正在翘首以盼18号WAL文件 */
oracle=# \! ps -ef | grep postgres | grep startup | grep -v grep
postgres  2616  2613  0 13:05 ? 00:00:00 postgres: startup waiting for 00000001000000000000000018
/* 查看一下数据库是否处于恢复状态，结果为t (true) */
oracle=# SELECT pg_is_in_recovery();
pg_is_in_recovery
-----
 t /* t 表示该数据库还处于恢复状态 */
(1 row)
/* 现在终止恢复，把数据库提升到可读可写的正常工作状态 */
oracle=# SELECT pg_wal_replay_resume();
pg_wal_replay_resume
-----
```

```
(1 row)
/* 结果发现startup进程消失了 */
oracle=# \! ps -ef | grep postgres | grep startup | grep -v grep
/* 再次检查数据库的状态，结果为f(false)，表明数据库已经处于可读可惜的工作状态 */
oracle=# SELECT pg_is_in_recovery();
pg_is_in_recovery
-----
f
(1 row)
/* 往测试表里面插入记录，结果成功，说明数据库可以对外营业了 */
oracle=# INSERT INTO state VALUES(1, 'California');
INSERT 0 1
oracle=# SELECT * FROM state ORDER BY 1;
 id |      name
----+-----
 0 | Ohio
 1 | California
(2 rows)
```

通过上面的实验，我们完整地把数据库恢复并运行起来了。这个时候，你会发现 pg_wal 目录下有一些异常的东西，请执行如下命令：

```
oracle=# \! ls -l /opt/data/restore/pg_wal
total 32776
-rwx----- 1 postgres postgres 16777216 Oct  8 11:16 00000001000000000000000017
-rw----- 1 postgres postgres 16777216 Oct  8 13:12 00000002000000000000000017
-rw----- 1 postgres postgres      33 Oct  8 13:07 00000002.history
drwx----- 2 postgres postgres   4096 Oct  8 13:07 archive_status
```

你会看到有两个 17 号的 WAL 文件，但是它们的时间线不一样，就是开始的 8 个字符，从老的 00000001 变成了 00000002。从本节开始，我们就要理解“时间线”这个概念了，后文会进行详细讨论。

4.2.2 恢复目标

在上面的实验中，我们指定了参数 recovery_target，它的作用是让数据库第一次达到一致状态后就停止恢复。但是现实中我们往往需要指定某一个时间点。譬如某一个用户不小心把一张重要的表的数据删除了，他非常懊恼地寻求数据库管理员的帮助。管理员问他什么时候删除的，这位用户回忆说可能是下午 3 点左右，那么我们把数据库恢复到下午 2 点，肯定此时该表中的数据还没有被删除掉，这个时候我们就要指定 PITR 恢复目标。PITR 恢复目标有如下几种类型：

- 基于时间的恢复 (recovery_target_time)
- 基于 lsn 的恢复 (recovery_target_lsn)
- 基于事务 id 的恢复 (recovery_target_xid)
- 基于名字的恢复 (recovery_target_name)
- 尽快恢复 (recovery_target_immediate)

如果我们不指定恢复目标，PostgreSQL 就会把数据恢复到可用的 WAL 记录的尽头，此时数据处于最新的一致性状态。在第三章，我们详细考察了一条 COMMIT 的 WAL 记录，其里面的数据仅仅是 8 个字节，表明该 Transaction 提交时的时间戳，这就把时间和 LSN 联系起来了，所以基于时间的恢复也是基于 LSN 的恢复。事实

上以上所有的恢复目标类型，其本质都是基于 LSN 的恢复。所以我们只研究基于 LSN 的恢复的问题。下面我们演示一个基于 LSN 的恢复的实验来揭示其关键的技术要点。

现在我们有一个正常运行的数据库，其归档模式都已经配置好了，我们使用 pg_basebackup 执行一个备份 bk4：

```
postgres=# show archive_mode;
archive_mode
-----
on
(1 row)

postgres=# show archive_command;
archive_command
-----
test ! -f /home/postgres/a/%f && cp %p /home/postgres/a/%f
(1 row)

$ pg_basebackup -h localhost -U postgres -X fetch -t server:/home/postgres/backup/bk4
$ ls -l /home/postgres/backup
total 8
drwx----- 2 postgres postgres 4096 Oct  8 11:16 bk3
drwx----- 2 postgres postgres 4096 Oct  8 16:22 bk4
$ ls -l /home/postgres/backup/bk4
total 46324
-rw----- 1 postgres postgres 181409 Oct  8 16:22 backup_manifest
-rw----- 1 postgres postgres 47251456 Oct  8 16:22 base.tar
```

然后我们检查一个测试表，并做一次 WAL 文件的切换动作后，记录一下当前的 LSN：

```
$ psql -d oracle
psql (16.0)
Type "help" for help.
/* 查询一下测试表中的数据 */
oracle=# SELECT * FROM state ORDER BY 1;
 id | name
----+---
 0 | Ohio
(1 row)
/* 查看一下归档目录里面的WAL文件的信息 */
oracle=# \! ls -l /home/postgres/a
total 49156
-rw----- 1 postgres postgres 16777216 Oct  8 16:19 00000001000000000000000019
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 0000000100000000000000001A
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 0000000100000000000000001B
-rw----- 1 postgres postgres      341 Oct  8 16:22 0000000100000000000000001B.00000028.backup
oracle=# SELECT pg_switch_wal(); /* 切换一下WAL文件，翻过一个新的篇章 */
pg_switch_wal
-----
0/1C000078
(1 row)
/* 再次查看一下归档目录里面的WAL文件的信息，发现多了1C号WAL文件 */
```

```

oracle=# \! ls -l /home/postgres/a
total 65540
-rw----- 1 postgres postgres 16777216 Oct  8 16:19 00000001000000000000000019
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 0000000100000000000000001A
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 0000000100000000000000001B
-rw----- 1 postgres postgres      341 Oct  8 16:22 0000000100000000000000001B.00000028.backup
-rw----- 1 postgres postgres 16777216 Oct  8 16:26 0000000100000000000000001C
oracle=# SELECT pg_current_wal_lsn(); /* 记录一下当前的LSN的位置信息 */
pg_current_wal_lsn
-----
0/1D000060
(1 row)

```

现在我们进行人为的破坏，把 state 表 TRUNCATE 掉，模拟灾难发生了，并且假装不知道，继续切换 WAL 文件，表示时间继续流逝：

```

oracle=# TRUNCATE TABLE state; /* 我们犯下了错误！ */
TRUNCATE TABLE
oracle=# SELECT pg_switch_wal(); /* 切换一下WAL文件，翻过一个新的篇章 */
pg_switch_wal
-----
0/1D004028
(1 row)
/* 再次查看一下归档目录里面的WAL文件的信息，发现多了1D号WAL文件 */
oracle=# \! ls -l /home/postgres/a
total 81924
-rw----- 1 postgres postgres 16777216 Oct  8 16:19 00000001000000000000000019
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 0000000100000000000000001A
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 0000000100000000000000001B
-rw----- 1 postgres postgres      341 Oct  8 16:22 0000000100000000000000001B.00000028.backup
-rw----- 1 postgres postgres 16777216 Oct  8 16:26 0000000100000000000000001C
-rw----- 1 postgres postgres 16777216 Oct  8 16:31 0000000100000000000000001D

```

过了一段时间，我们发现了我们犯的错误，现在需要把 state 表中的数据找回来。现在我们手里有了一个备份 bk4，很显然我们要把该数据库恢复到 LSN=0/1D000060 这个状态。在实际中我们不可能记住具体的 LSN，但是我们依稀记得错误发生之前的时间点，恢复到大约昨天晚上七点半即可等等，其实质依然还是基于 LSN 的恢复。现在我们开始做 PITR 的恢复工作。首先创建一个目录，并把 bk4 的备份还原到该目录，这和第一个恢复实验并没有什么区别：

```

$ cd /opt/data
$ ls -l
total 4
drwx----- 19 postgres postgres 4096 Oct  8 12:29 pgdata1
$ mkdir pitr
$ cd pitr
$ tar xvf /home/postgres/backup/bk4/base.tar
$ pwd
/opt/data/pitr
$ cat backup_label
START WAL LOCATION: 0/1B000028 (file 000000010000000000000001B)

```

```

CHECKPOINT LOCATION: 0/1B000060
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2023-10-08 16:22:48 MDT
LABEL: pg_basebackup base backup
START TIMELINE: 1
$ ls -l pg_wal
total 16388
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 000000010000000000000001B
$ ls -l /home/postgres/a
total 81924
-rw----- 1 postgres postgres 16777216 Oct  8 16:19 0000000100000000000000019
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 000000010000000000000001A
-rw----- 1 postgres postgres 16777216 Oct  8 16:22 000000010000000000000001B
-rw----- 1 postgres postgres      341 Oct  8 16:22 000000010000000000000001B.00000028.backup
-rw----- 1 postgres postgres 16777216 Oct  8 16:26 000000010000000000000001C
-rw----- 1 postgres postgres 16777216 Oct  8 16:31 000000010000000000000001D

```

我们知道 bk4 需要从 0/1B000028 这个重做点开始恢复，直至到 0/1D000060 这一点，所以它需要的 WAL 文件包括 1B, 1C, 1D 三个 WAL 文件。这三个文件我们都有，但是只有 1B 文件被保存在了 pg_wal 目录中，1C 和 1D 两个文件在归档目录/home/postgres/a 这个目录下。你可以选择手工把这两个文件拷贝到/opt/data/pitr/pg_wal 中，但如果涉及到的 WAL 文件有成千上万个，这种手工拷贝的方法就比较笨。更聪明的办法是设置 restore_command 参数，让它在需要的时候自动拷贝。这次我们不再使用哑命令，而是写一个真实的拷贝命令，所以我们编辑/opt/data/pitr 下的主配置文件，在最后增加这几行参数：

```

recovery_target_lsn='0/1D000060'
restore_command='cp /home/postgres/a/%f %p'
port=5433

```

很显然，参数 recovery_target_lsn 指定我们要恢复到哪个 LSN 点，而 restore_command 规定了从哪里拷贝必要的 WAL 文件到 pg_wal 目录中。这是和第一个恢复实验不同的地方。剩下的步骤和第一个恢复实验完全一样，下面是具体的操作流程：

```

$ pwd
/opt/data/pitr
$ touch recovery.signal /* 创建恢复所需要的信号文件recovery.signal */
$ cd ..
$ ls -l
total 8
drwx----- 19 postgres postgres 4096 Oct  8 12:29 pgdata1
drwxr-xr-x 19 postgres postgres 4096 Oct  8 16:47 pitr
$ chmod -R 700 pitr
$ ls -l
total 8
drwx----- 19 postgres postgres 4096 Oct  8 12:29 pgdata1
drwx----- 19 postgres postgres 4096 Oct  8 16:47 pitr
$ pg_ctl start -D pitr -l logPITR
waiting for server to start.... done
server started
$ tail logPITR

```

```

.....
2023-10-08 16:47:58.026 MDT [4591] LOG: consistent recovery state reached at 0/1B000100
2023-10-08 16:47:58.026 MDT [4591] LOG: recovery stopping after WAL location (LSN) "0/1D000060"
2023-10-08 16:47:58.026 MDT [4591] LOG: pausing at the end of recovery
2023-10-08 16:47:58.026 MDT [4591] HINT: Execute pg_wal_replay_resume() to promote.
2023-10-08 16:47:58.026 MDT [4588] LOG: database system is ready to accept read-only connections
/* 在数据库启动之后，登录进去检查被删除的数据是否找回来了 */
$ psql -p 5433 -d oracle
psql (16.0)
Type "help" for help.
/* 查看测试表，发现里面的数据已经找回来了 */
oracle=# SELECT * FROM state;
 id | name
----+-----
 0 | Ohio
(1 row)
oracle=# SELECT pg_wal_replay_resume(); /* 把数据库提升为可读可写的正常状态 */
pg_wal_replay_resume
-----

(1 row)
oracle=# SELECT pg_is_in_recovery();
pg_is_in_recovery
-----
f
(1 row)
oracle=# \! ls -l /opt/data/pitr/pg_wal /* 查看pg_wal目录下的WAL文件*/
total 65544
-rw----- 1 postgres postgres 16777216 Oct  8 16:49 000000010000000000000001D
-rw----- 1 postgres postgres 16777216 Oct  8 16:49 000000020000000000000001D
-rw----- 1 postgres postgres 16777216 Oct  8 16:47 000000020000000000000001E
-rw----- 1 postgres postgres 16777216 Oct  8 16:47 000000020000000000000001F
-rw----- 1 postgres postgres      35 Oct  8 16:49 00000002.history
drwx----- 2 postgres postgres     4096 Oct  8 16:49 archive_status
/* 查看时间线切换文件的内容 */
oracle=# \! cat /opt/data/pitr/pg_wal/00000002.history
1      0/1D000098      after LSN 0/1D000060

```

当我们查看 pg_wal 里面的 WAL 文件时，发现时间线已经从 1 升级到了 2，其中 1D 号 WAL 文件在两个时间线都有。里面还有一个 00000002.history 文件，这个是时间线的切换文件，我们看看里面的内容。如果我们把不同的时间线理解为不同的跑道的话，该文件表明从 0/1D000098 这个位置开始从时间线 1 切换到了时间线 2 的。下面我们就来研究一下时间线这个概念，为啥要引入这个概念。

4.2.3 时间线

我们看到了，每次做完 PITR 以后，时间线会自动加 1。什么是时间线呢？我们首先看看如果不引入这个概念，会产生什么问题，图 4.12 展示了时间线的示意图：

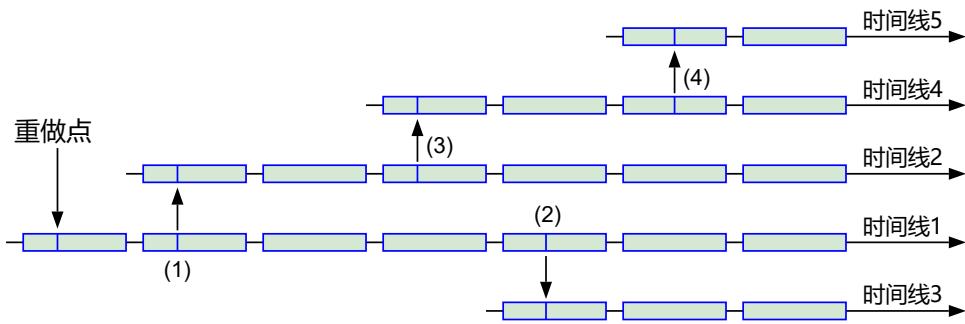


图 4.12: 时间线的基本概念

假设没有时间线的概念，我们做一个 PITR，从最左边的重做点开始，恢复到了 1 号点，通过上面的第二个恢复实验，我们已经熟悉了它的过程。等恢复成功以后，数据库继续运行一段时间，后来我们发现恢复错了，我们需要恢复到 2 号点，却发现办不到，为什么呢？因为恢复到了 1 号点之后，数据库会继续产生新的 WAL 文件，这些新产生的 WAL 文件和 1 号点与 2 号点之间的老的 WAL 文件重名，所以这段老的 WAL 文件就被覆盖掉了，导致我们无法恢复到 2 号点。如果恢复到 1 号点后，后续产生的新的 WAL 文件使用不同的名字，就不存在这个问题了。我们知道：WAL 文件的文件名是由三部分共 24 个字符组成，如果第一部分的时间线不同，即头 8 个字符不同，就算后面两部分相同，文件名还是不一样，从而避免了新文件覆盖老文件的问题，这就是时间线的来由，它通过改变 WAL 文件的文件名的高八位的字符来避免覆盖老的 WAL 文件。搞了半天，这个非常具有科幻色彩的名字本质上就是为了避免老的 WAL 文件被新 WAL 文件覆盖掉而创造出来的。我们可以把时间线理解为赛道的编号。在数据库集群创建伊始，赛道的编号是 1，每做一次数据库恢复，就切换到新赛道，新赛道编号是老赛道编号加一，PostgreSQL 就用这种简单的办法来避免覆盖掉老的 WAL 文件。

在图 4.12 中，恢复到 1 号点后，时间线从 1 切换到了 2，换了一个赛道继续产生新的 WAL 文件。如果再次从最左边的重做点出发，恢复到了 2 号点，时间线就由 2 变成了 3。如果在 1 号点已有的基础上在 3 号点进行恢复，时间线就从 2 切换到了 4。总之，时间线是永远增加的，每做一次 PITR，当前时间线就增加一。如果一个数据库做过 n 次 PITR 恢复，则它有 $n+1$ 个时间线。很显然，除了要保存所有的 WAL 文件以外，我们还需要保存时间线切换点，记录从哪个 LSN 开始从时间线 m 切换到时间线 n 的。这些时间线切换历史的文件名的规律是：时间线.history。譬如切换到时间线 5 的切换文件叫做 0000005.history，它里面记录了自时间线 1 不停地切换赛道最终达到 5 号时间线的所有切换点，以图 4.11 为例，为了从时间线 1 成功切换到时间线 5，我们要记录三个切换点：1 号点，3 号点和 4 号点。这个时间线历史文件和 WAL 文件一样，对于成功恢复数据库至关重要，也需要被妥善保存，否则我们根本不知道从哪个位置切换赛道，PostgreSQL 在归档时，除了保存 WAL 文件以外，也会自动保存这些时间线切换的历史信息到归档目录。因为这些切换文件往往就是几十个或者几百个字节的小的文本文件，所以保存它们的代价很小。你在日常工作中务必不要删除它们。

4.3 逻辑备份和恢复

相对于物理备份和恢复的复杂性，逻辑备份就简单很多了，它使用的工具主要是 pg_dump 和 pg_restore。其中 pg_dump 的作用是做备份，pg_restore 的作用是做恢复。逻辑备份的思想也非常简单，就是以普通的用户登录数据库，把需要备份的数据变成 SQL 命令，保存在脚本文件中。恢复的时候读取该脚本文件，执行里面的 SQL 即可。不过逻辑备份有着物理备份所不具备的灵活性：它可以指定备份某一个数据库，某一个 schema，或者某一张表。它可以只备份 DDL，不备份数据，也可以只抽取数据，变成 INSERT 语句等等。正因为这种灵活性，所以企业中的数据库备份以物理备份为主，逻辑备份为辅，充分利用这两种不同备份技术的优点。下面我们分别介绍 pg_dump 和 pg_restore 的基本使用。

4.3.1 pg_dump 的基本使用

逻辑备份工具 pg_dump 和 psql 一样，是一个普通的客户端软件，它并不需要非常特殊的权限，只要可以读取相关的数据库对象即可。因为一个数据库里有很多对象，普通用户可能对有些对象有读的权限，但是对一些对象没有读取的权限，pg_dump 照样可以成功备份，除非备份的数据不全而已。因为这个原因，在备份整个数据库时，往往使用超级用户，因为超级用户可以读取任何对象。它可以备份的参数很多，可以使用 pg_dump --help 来查看，一些常用的参数，譬如 -d 指定备份整个数据库，-n 备份指定的 schema，-t 备份指定的表，-s 表示只备份 DDL，不备份数据，这个对建立一个空壳数据库非常有用。

我们下面通过具体的例子演示一些常用的用法。首先需要在实验服务器上搭建一套测试用的数据库：

```
$ cat $PGDATA/postgresql.conf | grep listen /* 打开侦听端口 */
listen_addresses = '*'
$ cat pgdata1/pg_hba.conf /* 允许用户远程连接 */
host      all      all      192.168.137.0/24      md5
$ pg_ctl -D /opt/data/pgdata1 -l logfile start /* 启动数据库 */
waiting for server to start.... done
server started
$ psql
psql (16.0)
Type "help" for help.
/* 创建测试数据库oracle */
postgres=# create database oracle;
CREATE DATABASE
postgres=# \c oracle /* 切换到oracle数据库，创建测试表并插入测试记录 */
You are now connected to database "oracle" as user "postgres".
oracle=# create table state(id int primary key, name varchar(16));
CREATE TABLE
oracle=# insert into state values(0, 'Teks');
INSERT 0 1
oracle=# CREATE USER datadump WITH PASSWORD '123456'; /* 创建一个测试用户 */
CREATE ROLE
oracle=# GRANT ALL ON state TO datadump; /* 赋予必要的权限 */
GRANT
```

然后我们在另外一台机器上执行 pg_dump 命令

```
/* 把一个数据库的全部内容备份到 oracle.sql这个文本文件中 */
$ pg_dump -U postgres -h 192.168.137.12 -d oracle > oracle.sql
/* 把一个数据库的全部DDL备份到 oracle_ddl.sql这个文本文件中 */
```

```
$ pg_dump -U postgres -h 192.168.137.12 -d oracle -s > oracle_ddl.sql
/* 把表state的内容备份下来 */
$ pg_dump -U datadump -h 192.168.137.12 -d oracle -t state
/* 把表state的数据变成INSERT语句 */
$ pg_dump -d oracle -h 192.168.137.12 -U datadump --column-inserts --data-only --table=state
```

在缺省的情况下，`pg_dump` 把输入写到标准输出 `stdout`，即屏幕上。在实际使用中往往通过重定向把结果写入到一个脚本中，上述实验中使用的大于号就是用于把原本写入到标准输出的结果改写到一个文本文件中了。关于 Linux 重定向的问题，在互联网上稍微查询一下就知道怎么回事了，这里不再赘述。`pg_dump` 备份的数据是一致的，即 `pg_dump` 在开始备份时会创建一个数据的快照，在备份的过程中，即使这些数据发生了变化，也不会影响 `pg_dump` 的数据的一致性，而且 `pg_dump` 在运行时，基本上不会阻碍其它客户的正常操作。

`pg_dump` 产生的是 SQL 脚本，完全可以在 `psql` 中运行。这里值得一提的时候，运行的脚本可能会发生错误，缺省情况下脚本会跳过错误，继续运行下一条 SQL 命令。如果你想让脚本在发生错误时终止运行，可以使用 `ON_ERROR_STOP` 参数，把它设置为 `true` 即可。你很容易在互联网上找到如何设置的方法。

`pg_dump` 在备份数据的时候并不会备份角色 (ROLE) 和表空间等信息，因为这些对象都是全局的。为了把整个数据库集群的内容备份下来，PostgreSQL 提供了另外一个工具 `pg_dumpall`，它的作用就是备份整个数据库集群，它的大部分参数和 `pg_dump` 保持一致，所以你熟悉 `pg_dump` 后再学习 `pg_dumpall` 是非常迅速的。

4.3.2 pg_restore 的基本使用

`pg_restore` 的基本使用。

第五章 物理复制

通过前两章的学习，我们已经对 WAL 记录的基本作用，数据库的备份和恢复有了一个整体了解。WAL 记录除了可以用在本地数据库的恢复以外，也可以通过网络传输到远端，恢复远端的数据库，这就诞生了数据库复制 (replication) 技术。数据库复制技术是实现数据库高可用性 (HA : high availability) 的主要技术手段。所谓高可用性，通俗来说，就是万一某个非常重要的数据库无法正常工作，必须有一个相同的数据库在关键时刻能够顶替它，保证整个应用系统和业务运营 (business operation) 不会中断。毫无疑问，这是企业迫切需要的核心功能。从实质上来说，数据库复制和本地恢复并无本质的区别，其核心思想都是使用 WAL 记录来恢复数据库，无非一个是本地手动执行，一个是把 WAL 记录通过网络传输到远端，自动执行罢了。这里的“远端”的含义并不局限在网络带宽非常大的局域网，两个数据库可以相距上万公里之遥，通过广域网进行连接。图 5.1 展示了数据库复制技术的基本概念：数据库复制涉及两个数据库，左边的数据库被称为主库 (primary database)，右边的数据库被称为备库 (standby database)。主库是源头，备库目标。备库是主库的克隆，它的内容和主库一模一样，在主库中修改的任何数据都会源源不断地流向并修改备库。数据库的复制技术是单向的数据传输。

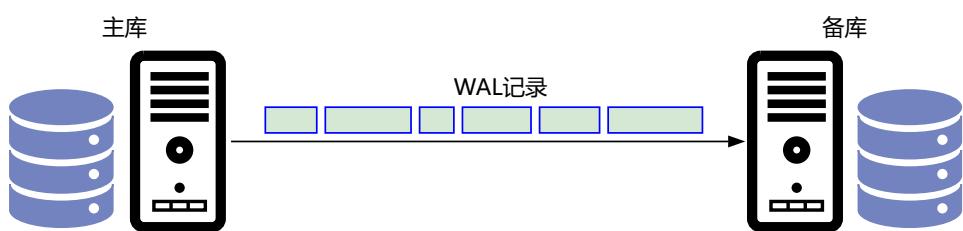


图 5.1: 数据库复制技术的概念

数据库复制技术分为物理复制 (physical replication) 和逻辑复制 (logical replication) 两种类型。简单来说，物理复制就是本地数据库恢复的远端版本，即：把 WAL 记录通过网络传送到远端，然后再使用 WAL 记录回放的方式恢复远端的数据库。逻辑复制是把 WAL 记录进行解码，变成了类似 SQL 语句的指令，在远端执行。物理复制的优点是执行速度快，非常可靠，缺点是只能对整个数据库集群进行复制，无法单独复制某一个数据库或者某些表，灵活性不足。逻辑复制的优缺点和物理复制正好颠倒过来，逻辑复制可以被配置成只复制哪些表，不复制哪些表，也可以指定数据过滤条件，只复制符合过滤条件的记录，所以逻辑复制有更多的灵活性。在企业实践中，往往把这两种复制技术结合起来，用来满足不同需求场景的数据复制需求。各大主流数据库都有自己的复制技术，例如 Oracle 数据库有 Data Guard 的物理复制和 GoldenGate 的逻辑复制解决方案；微软的 SQL Server 有 Always On 的解决方案等等。PostgreSQL 之所以能够被越来越广泛的企业用户所采用，它当然也有非常成熟的物理复制和逻辑复制技术。本章专门研究 PostgreSQL 的物理复制技术，逻辑复制在后续章节讲解。

物理复制，可以基于 WAL 文件 (wal file based)，也可以基于 WAL 记录 (wal record based)。基于 WAL 文件的物理复制，就是等主库的一个 WAL 文件写满后再传输到备库。众所周知，通常 WAL 文件的大小为 16M 或者更大，写满一个 WAL 文件总是需要一定时间的。这种方式延迟比较大，不能让备库的数据实时更新，所以基本上被淘汰了，只应用在某些特殊场合。基于 WAL 记录的物理复制，就是一旦有新的 WAL 记录产生，就会立刻传输到备库，无需等到 WAL 文件被写满。一条 WAL 记录的长度，短的就几十个字节，长的也就几百上千个字节，所以它的传输速度快。在网络条件良好的情况下，备库的数据几乎和主库是实时同步的。所以基于 WAL 记录的物理复制成为主流，它又被称为流复制 (stream replication)，表示数据库更新的数据像水流一样源源不断，以示和基于 WAL 文件的传输方式的区别。后文中我们会根据上下文交叉使用流复制和物理复制这两个术语。图 5.2 展示了 PostgreSQL 的流复制的体系架构。

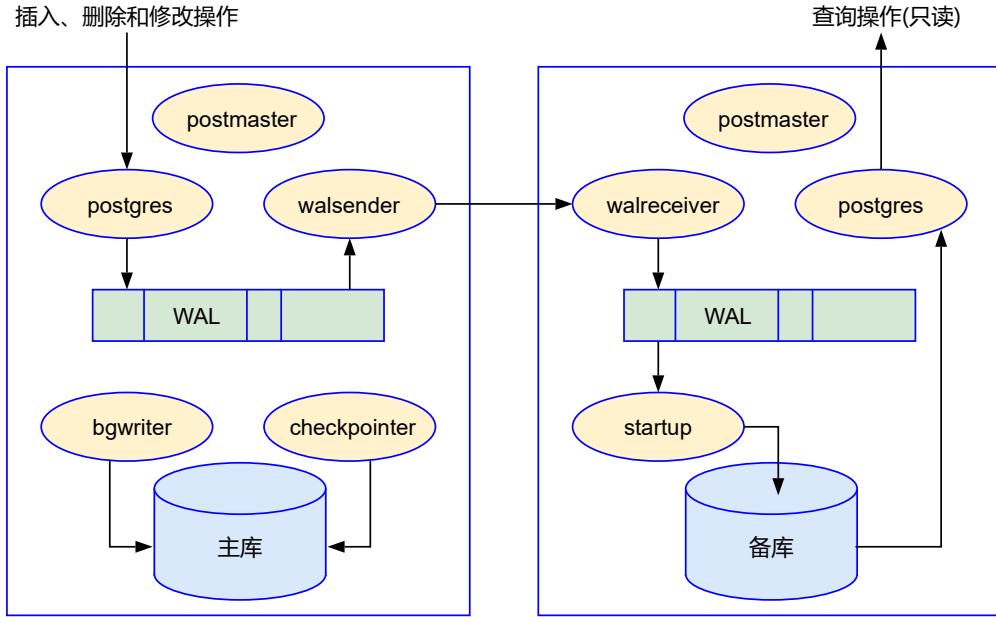


图 5.2: 流复制的整体架构

在上图中，左边的大矩形框表示主库，右边的大矩形框表示备库。用户在主库端发起修改数据的事务，即插入、修改和删除等操作，就会产生相关的 WAL 记录。这些 WAL 记录会被写入到主库的 pg_wal 目录下的 WAL 文件中，这个过程我们已经非常熟悉了。主库有一个后端进程，被称为 WAL 发送进程 (walsender)。它会不断地读取 WAL 文件中的记录，通过 TCP 连接，传输给备库上的 WAL 接收进程 (walreceiver)。WAL 接收进程是备库的一个后台进程，它收到来自主库的 WAL 记录后，会写入备库的 pg_wal 目录下的 WAL 文件中。备库上还有一个恢复进程 (startup)，不断读取 WAL 文件中的 WAL 记录，把它们不断回放到数据文件上。这就是流复制的数据传输的基本过程。关于恢复进程的内容，我们已经对它相当熟悉了：恢复进程的任务就是从某个重做点开始，依次读取它能够获得的 WAL 记录，用来修正数据文件中的数据块，使之达到更新的状态。恢复进程获得 WAL 记录的来源有三个：一是通过 restore_command 参数指定的命令把 WAL 文件拷贝到 pg_wal 目录中，二是 pg_wal 目录中已经存在的 WAL 文件，三是通过网络获得。通过网络获得的 WAL 记录就需要备库端的 WAL 接收进程和主库端的 WAL 发送进程来配合完成。主库的 WAL 发送进程，备库的 WAL 接收进程和恢复进程这三个进程，是流复制的主要进程。

物理复制中的备库可以称为“物理备库”，逻辑复制中的备库可以称为“逻辑备库”。主库和物理备库的最大区别是：主库是可读可写的，而物理备库是只读的。这个原因非常容易理解：物理备库的目的是成为主库的克隆，必须保持和主库一模一样。只有这样，当主库无法工作时，物理备库才能够立刻顶上去，变成新主库。所以必须确保物理备库的修改百分百地都来自主库，这样才能保证物理备库和主库的数据完全一致。逻辑复制则不同，因为逻辑备库只是有条件地接受来自主库的部分修改，它和主库并不是百分百一样，甚至非常不一样，只是某些表的数据是相同的，它没有必要保持只读状态。所以逻辑备库是可读可写的：除了接收来自主库的修改以外，它还可以接受其他用户的数据修改请求。物理备库切换成新主库以后，应用 (application) 最多知道数据库服务器的 IP 地址发生了变化，别的和老主库没有任何区别。如果采用 HAProxy 和 PgBouncer 等代理服务器，应用连接的是代理服务器，代理服务器的 IP 地址保持不变，在它身后的数据库服务器的 IP 地址发生了变化，应用是毫无察觉的，这就是所谓高可用的透明性。

5.1 快速搭建流复制

我们先通过一个实验，快速搭建 PostgreSQL 的流复制环境，以期获得初步的感性认识。流复制的实验环境需要两台计算机，一台计算机作为主库服务器，IP 地址是 192.168.137.16，其上已经运行了一个 PostgreSQL 的数据库集群。另一台计算机作为物理备库服务器，它的 IP 地址是 192.168.137.17，上面只安装好了 PostgreSQL 的软件，并没有数据库集群在运行。整个实验分为在主库上做的配置和在备库上做的配置两方面的内容，下面是实验的具体细节。

5.1.1 在主库机器上的配置

首先，我们需要修改主库的若干参数，确保主库处于归档模式，可以接收远程客户的连接等等。因为有些参数需要重新启动主库后才能生效，所以我们首先把主库关闭。然后我们修改它的 `postgresql.conf`，设置如下参数：

```
listen_addresses = '*'          # 表示可以接受远端的网络连接
wal_level = replica            # 可以设置为 replica 或者 logical。
hot_standby = on                # 这个参数确保让备库可以接受只读的查询请求
max_wal_senders = 10            # 主库允许有多少个 walsender 进程被启动
max_replication_slots = 10      # 设置复制槽的个数
archive_mode = on                # 归档模式，这个我们已经熟悉了
archive_command = 'test ! -f /opt/data/a/%f && cp %p /opt/data/a/%f'
```

在上面的配置中，参数 `hot_standby` 的值为 `on`，表示备库可以接受只读请求。参数 `max_wal_senders` 表示主库可以启动多少个 WAL 发送进程。你有一个备库，主库就需要有一个 WAL 发送进程与之联系，所以你有多少个备库，这个参数的最小值就是几个。我们的实验中只有一个备库，这个参数的值设置为 1 也可以，大一点也可以，只是稍微占用一点共享内存而已。参数 `max_replication_slots` 是设置复制槽的个数，关于复制槽我们后面会介绍，这里照抄一下即可。为了允许备库的 WAL 接收进程可以连接到主库上，我们需要修改主库的 `pg_hba.conf`，在文件的最后加上如下一行：

host	replication	all	192.168.137.0/24	md5
------	-------------	-----	------------------	-----

上述一行配置的目的是确保任何 192.168.137 网段内的机器都可以使用复制协议连接本数据库。以上工作完毕后，重新启动主库。当主库启动后，我们需要在主库中创建一个账号，因为备库实际上是主库的一个客户端，备库会使用该账号登录主库。当然，你可以使用 `postgres` 这个超级用户作为复制的账号，但从安全的角度，一般不建议这样做。我们使用 `psql` 以超级用户登录主库，执行如下命令创建一个专门的账号：

```
$ psql -U postgres
psql (16.0)
Type "help" for help.
/* 创建用户kevin，而且赋予这个用户REPLICATION的权限 */
postgres=# CREATE USER kevin WITH REPLICATION LOGIN PASSWORD '123456';
CREATE ROLE
```

我们在创建 `kevin` 账号时指定了 `WITH REPLICATION` 的选项，就表明这个账号具有以复制协议连接服务器的权限。以上就是在主库上的全部工作。在完成主库上的配置后，我们在备库上通过 `pg_basebackup` 创建一个备份，这个备份就是物理备库的起点。

5.1.2 在备库机器上的配置

首先我们以 `kevin` 用户，在备库机器上，使用 `psql` 客户端，通过复制协议远程登录主库，测试一下远程连接是否能够成功。具体操作细节如下：

```
$ psql "replication=yes host=192.168.137.16 user=kevin dbname=postgres"
Password for user kevin: /* <-- 在这里输入kevin的口令，是123456 */
psql (16.1)
Type "help" for help.
/* 测试使用IDENTIFY_SYSTEM命令获取主库的系统标识符 */
postgres=> IDENTIFY_SYSTEM;
systemid      | timeline | xlogpos   | dbname
-----+-----+-----+
7321027155043554108 |       1 | 0/14C1858 |
(1 row)
```

如果这一步失败了，说明主库的远程连接的配置没有设置正确，请仔细检查。你可以参考本书前面的关于远程配置的内容。常见的错误是Linux服务器上的防火墙阻断了网络通讯，所以你要在防火墙上把5432的端口打开。如果这一步成功了，我们就可以备库机器上执行pg_basebackup创建一个备份，执行如下命令：

```
$ pg_basebackup -h 192.168.137.16 -U kevin -D /opt/data/standby -Fp -R -c fast -X stream
Password:
```

上述命令使用了一些新的参数，其中-Fp表示创建的备份是一个目录，和源数据库的结构一样。参数-R很重要，它创建了物理复制所需要的文件，等备份结束后我们会看到这些文件。参数-c表示立刻执行一个检查点，相当于pg_backup_start()函数中的fast参数的值为true。当上述备份命令完成后，我们发现pg_basebackup在备库机器上创建了一个目录/opt/data/standby，下面的操作检查一下备份目录，检查的内容有三个要点，展示如下：

```
$ ls -l /opt/data/standby
total 260
-rw----- 1 postgres postgres    225 Jan  6 09:54 backup_label    /* <--注意这个文件！ */
-rw----- 1 postgres postgres 137318 Jan  6 09:54 backup_manifest
drwx----- 5 postgres postgres   4096 Jan  6 09:54 base
drwx----- 2 postgres postgres   4096 Jan  6 09:54 global
.....
-rw----- 1 postgres postgres    392 Jan  6 09:54 postgresql.auto.conf
-rw----- 1 postgres postgres  29898 Jan  6 09:54 postgresql.conf
-rw----- 1 postgres postgres      0 Jan  6 09:54 standby.signal /* <--注意这个文件！ */
$ cat /opt/data/pgdata1/postgresql.auto.conf | grep primary    /* 注意primary_conninfo这个参数 */
primary_conninfo = 'user=kevin password=123456 host=192.168.137.16 port=5432 ....'
```

我们可以看到，备份的目录中包括了backup_label和standby.signal两个文件。结合第三章中关于恢复进程启动的逻辑，我们知道：如果在数据库集群目录下存在standby.signal这个文件，则该数据库集群启动以后就会进入备库模式。因为这是一个备份，所以第一次启动时必须由backup_label来指定它的恢复起点。同时，我们看到了在postgresql.auto.conf中有一个参数primary_conninfo。很显然，这个参数包含了主库的IP地址，端口号，用户名和密码登信息，就是告诉备库如何和主库建立连接。这些工作都是-R参数来完成的。做完这些检查后，我们就可以启动备库了，在备库机器上执行如下启动命令：

```
$ pg_ctl start -l logfile -D /opt/data/standby
waiting for server to start.... done
server started
```

由此我们可以看出，一个数据库是备库有三个要点：一是必须有standby.signal文件的存在作为信号，指示该数据库进入备库模式；二是要设定参数primary_conninfo来告知数据库从哪里获得WAL记录；三是要有一个恢复的起点，这个起点可以保存在backup_label文件中，也可以保存在控制文件中。backup_label文件仅仅在备库第

一次启动时才需要，等备库成功启动后，下一次启动的起点已经被记录在备库的控制文件中了，所以 `backup_label` 这个文件就不需要了。在备库第一次启动时，恢复进程从 `backup_label` 文件中获得恢复的起点后，就顺手把它的名字改为 `backup_label.old` 了。在备库第二次启动时，它就会读取控制文件中的检查点作为恢复的起点。你在备库成功启动后可以看到 `backup_label` 变成 `backup_label.old` 文件。而 `standby.signal` 文件并不会被删除，因为它是数据库集群进入备库模式的信号文件，只要备库不改变角色，这个文件应该始终存在于备库的数据库集群目录中。下面是备库启动后的两个文件的状态：

```
$ ls -l /opt/data/standby/backup_label*
-rw----- 1 postgres postgres 225 Jan  6 10:26 /opt/data/standby/backup_label.old
$ ls -l /opt/data/standby/standby*
-rw----- 1 postgres postgres 0 Jan  6 10:26 /opt/data/standby/standby.signal
```

主库和备库的区别就在于有没有 `standby.signal` 文件。如果有了这个文件，数据库集群启动后就进入备库模式。在备库模式中，恢复进程在恢复完 `pg_wal` 目录中的 WAL 文件和执行 `restore_command` 参数中规定的命令所获得的 WAL 文件后，并不会退出，而是根据 `primary_conninfo` 参数的设置来尝试连接主库获得更多的 WAL 文件。恢复进程会持续不断地在这三种来源中来回寻找 WAL 记录，无限循环。如果目前没有新的 WAL 记录，恢复进程就会休眠一段时间后再次尝试通过这三种渠道获取 WAL 记录。当我们给备库发出升级 (promote) 到主库的指令后，备库的恢复进程先通知 WAL 接收进程退出，然后把手头上剩下的 WAL 记录回放完毕，也会退出。后面我们会讨论这种角色转换的过程。

5.1.3 验证流复制是否工作

验证流复制环境是否成功的方法很简单，就是在主库中创建一张测试表，并且往测试表中插入一条记录，然后看看这条记录是否在备库中。测试过程的细节如下：

```
/* ===== 在主库中执行如下操作 ===== */
$ psql -d oracle
psql (16.0)
Type "help" for help.
/* 创建一张测试表 */
oracle=# CREATE TABLE testab(id INT PRIMARY KEY, name VARCHAR(16));
CREATE TABLE
oracle=# INSERT INTO testab VALUES(0, 'Dallas'); /* 在测试表中插入一条测试记录 */
INSERT 0 1
/* ===== 在备库中执行如下操作 ===== */
$ psql -d oracle
psql (16.1)
Type "help" for help.
/* 可以看到测试的数据被立刻传递到了备库，说明物理复制成功了 */
oracle=# select * from testab;
 id | name
----+---
  0 | Dallas
(1 row)
```

由上可知，在主库中插入的记录被立刻复制到了备库中，说明了这个物理复制的实验顺利完成。紧接着我们在备库机器上查看有哪些后台进程：

```
$ ps -ef | grep postgres | grep -v grep
postgres 1031      1  0 10:26 ?    00:00:00 /opt/software/pg16/bin/postgres -D /opt/data/standby
```

```
postgres 1032 1031 0 10:26 ? 00:00:00 postgres: checkpointer
postgres 1033 1031 0 10:26 ? 00:00:00 postgres: background writer
postgres 1034 1031 0 10:26 ? 00:00:00 postgres: startup recovering 0000000100000000000000000006
postgres 1035 1031 0 10:26 ? 00:00:00 postgres: walreceiver streaming 0/6000060
```

果不其然，我们看到了恢复进程的身影，进程号是 1034。我们也看到了 WAL 接收进程，进程号是 1035。你注意这两个进程的父进程的进程号都是 1031，这是主进程。这个现象说明它们都是主进程的儿子，两者是兄弟关系。从上面的输出中我们也可以看到：恢复进程正在恢复 6 号 WAL 文件，WAL 接收进程接收到 LSN 为 0/6000060 这条 WAL 记录。然后我们在主库的机器上查看后台进程：

```
$ ps -ef | grep walsender | grep -v grep
postgres 1115 1089 0 12:26 ? 00:00:00 postgres: walsender kevin 192.168.137.17(40766) streaming
0/6000060
```

我们看到了 WAL 发送进程，它的进程号是 1115，它正在和 192.168.137.17 这台机器的 40766 端口建立连接，当前发送的 WAL 记录的 LSN 是 0/6000060。我们可以进一步使用一些网络命令来查看这个网络连接。譬如在 Linux 下可以使用 ss 查看哪个进程在备库机器上使用 40766 端口。我们在备库的机器上执行如下命令：

```
$ ss -np | grep 40766 /* 可以看到进程1035正在使用40766端口和IP地址为192.168.137.16的5432端口连接 */
tcp ESTAB 0 0 192.168.137.17:40766 192.168.137.16:5432 users:((("postgres",pid=1035,fd=5)))
$ ps -ef | grep 1035 | grep -v grep /* 我们看看进程1035到底是谁，结果为walreceiver */
postgres 1035 1031 0 10:26 ? 00:00:00 postgres: walreceiver streaming 0/6000148
```

实锤找到了：上面的结果清晰无误地表明备库机器上的 WAL 接收进程以 40766 端口和主库机器上的 WAL 发送进程建立了一个 TCP 连接，且目标端口是 5432。由此可知，主库所产生的 WAL 记录就是通过这一个 TCP 连接传输到了备库端的。

5.2 主库和备库的通讯过程

在快速吃完人参果以后，我们来考察主库和备库之间的通讯过程，以便更深入地理解流复制。图 5.3 展示了主库和备库的通讯过程的主要步骤。你可以把备库理解为主库的一个客户端，它通过复制协议和主库保持长期的网络连接，其实这个和 pg_basebackup 的工作原理并没有太多区别，无非是 pg_basebackup 完成数据库备份后就退出了，而备库中的 WAL 接收进程和主库上的 WAL 发送进程之间的网络连接可以长年累月地保持着。主库和备库的通讯过程分为几个步骤。我们依次解释每一步的具体工作内容。

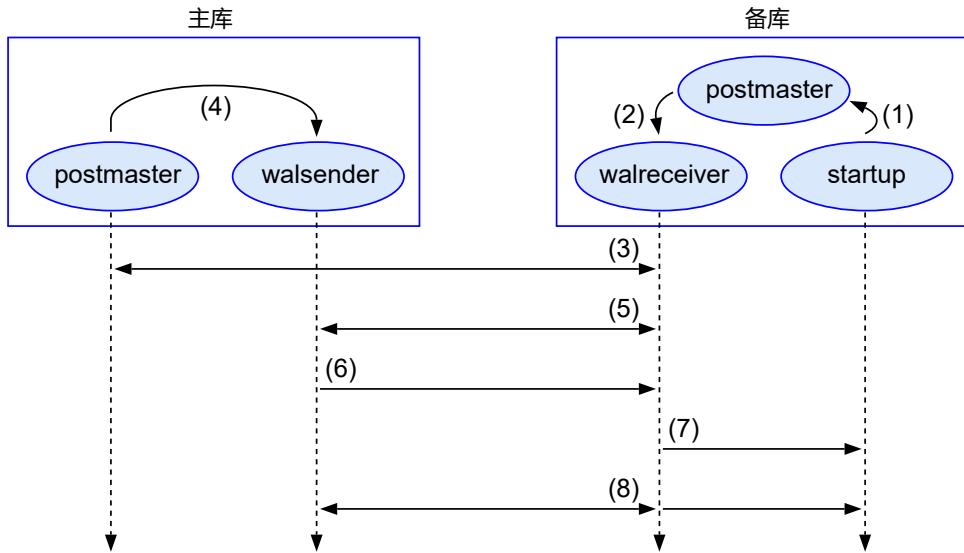


图 5.3: 流复制中主库和备库的通讯过程

在第一个步骤中，备库启动后，主进程无条件地创建恢复进程，恢复进程启动后会进行例行的检查。它发现了 `standby.signal` 文件，意识到现在处于备库模式，就开始从 `backup_label` 规定的重做点开始进行 WAL 记录的回放。如果 `backup_label` 文件不存在，恢复进程就从控制文件中读取重做点，以此点作为数据库恢复的起点。恢复进程首先会检查 `pg_wal` 目录下已经存在的 WAL 文件，还有利用参数 `restore_command` 指定的命令来获得 WAL 文件。当这两种来源的 WAL 文件已经回放完毕后，恢复进程便请求主进程创建 WAL 接收进程。第二个步骤比较简单，就是备库的主进程接到恢复进程的请求后，创建了 WAL 接收进程。

在第三个步骤中，备库的 WAL 接收进程根据参数 `primary_conninfo` 里面的信息，向主库的主进程发起 TCP 连接请求。如果此时主库无法连接，则 WAL 接收进程会过一段时间再次尝试和主库进行连接，周而复始。由此可以看出，主库和备库的网络连接是松耦合的，所以你可以对主库和备库随便启动和关闭，并不会破坏流复制的配置。流复制的框架在设计时已经充分考虑了网络无法连接的可能性。第四步的任务比较简单，就是主库这边的主进程接收到备库的 WAL 接收进程发起的 TCP 连接请求后创建 WAL 发送进程与之对接。

在第五个步骤中，主库方的 WAL 发送进程和备库方的 WAL 接收进程建立了一对一的 TCP 连接，后续的通讯均发生在这两个进程之间。WAL 接收进程会首先向 WAL 发送进程汇报备库的最新的 LSN，告诉主库：我已经成功接受到了 LSN_1 的 WAL 记录，请把这个位置以后的 WAL 记录传给我。这个步骤被称为“握手”阶段。

第六个步骤中，主库的 WAL 发送进程如果发现备库最新的 LSN_1 小于主库最新的 LSN_2 （通常情况下都是这种情况），说明两个库之间存在落差（gap），就把 LSN_2 和 LSN_1 之间的 WAL 记录发送给备库。这些 WAL 记录被保存在备库数据库集群目录下的 `pg_wal` 目录下的 WAL 文件中。这个过程被称为“追赶”阶段。由于 LSN_2 一旦获得，就是固定不变的，虽然此时主库可能继续产生大量的新的 WAL 记录，但是备库从 LSN_1 追赶到 LSN_2 在有限的时间内总是可以完成的，所以追赶阶段不会永远持续下去。第七个步骤也比较简单，就是备库的 WAL 接收进程接收到来自主库的 WAL 记录后，把它们写入到本地的 WAL 文件中，并通知恢复进程利用这些 WAL 记录恢复备库。在追赶阶段结束后，就进入了第八个步骤的“流复制”阶段。此时主库可能产生了新的 WAL 记录，WAL 发送进程会尽快地把这些 LSN_2 后面的 WAL 记录发送给备库进行新的同步，其过程和第六步和第七步一样，就是消费者（WAL 接收进程）不断告知生产者（WAL 发送进程）它消费的 WAL 记录的最后的位置，生产者把这个位置后面的 WAL 记录尽快传给消费者。如果主库产生 WAL 记录的速度太快，备库的恢复进程回放 WAL 记录的速度比较慢，就会出现消费者的消费速度赶不上生产者的生产速度的现象，备库会一直和主库存在比较大的落差。因为备库的恢复进程是单一进程，目前没有并发回放 WAL 记录的机制，这一点是需要改进的。

我们看到了，无论是 `pg_basebackup` 还是 WAL 接收进程，它们从本质上都是使用复制协议和数据库服务器

连接的客户端，由 WAL 发送进程统一为它们服务。WAL 发送进程是一种特殊的后端进程，它在内部维持一个状态机，共计有这么几个状态，其定义如下：

```
/* in src/include/replication/walsender_private.h */
typedef enum WalSndState {
    WALSNDSTATE_STARTUP = 0,
    WALSNDSTATE_BACKUP,
    WALSNDSTATE_CATCHUP,
    WALSNDSTATE_STREAMING,
    WALSNDSTATE_STOPPING
} WalSndState;
```

这几种状态的基本含义是：启动 (STARTUP)，即和 WAL 接收进程处于握手阶段；追赶 (CATCHUP)：即把主库和备库之间的落差传送给备库；流复制阶段 (STREAMING)，即追赶阶段结束后，主库产生新的 WAL 记录，会持续不断地传送到备库；备份 (BACKUP)，即把整个数据库集群的文件传送给类似 `pg_basebackup` 这样的备份工具。我们可以查询后文介绍的 `pg_stat_replication` 这个系统视图中的 `state` 这一列来查看 WAL 发送进程目前处于什么状态。

图 5.4 展示了一条 WAL 记录的传输轨迹。当用户修改主库的数据时，产生的 WAL 记录会保存在内存的 WAL 缓冲区中，然后在用户提交 COMMIT 命令时，这条 WAL 记录会被写入到主库 `pg_wal` 目录下的 WAL 文件中。主库的 WAL 发送进程从磁盘上读取 WAL 记录，通过网络发送给备库的 WAL 接收进程。备库的 WAL 接收进程接受到这条 WAL 记录后，把它写入到备库的 `pg_wal` 目录中的 WAL 文件里。然后备库的恢复进程会从 WAL 文件中读取该 WAL 记录，把它回放，来更新备库的数据文件，从而达到备库和主库同步的目的。

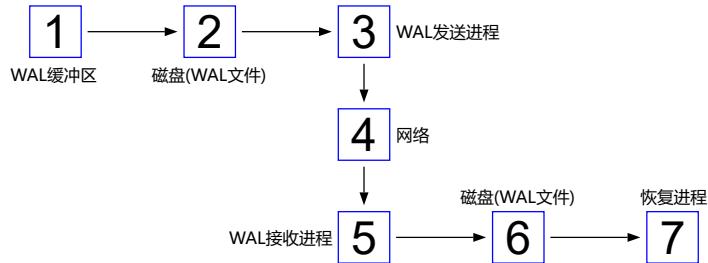


图 5.4: WAL 记录传输的各个环节

5.2.1 复制槽

在流复制的架构中，主库和备库是通过 WAL 发送进程和 WAL 接收进程建立的 TCP 连接松散地连接在一起的。在实践中，备库可能被关闭很长时间，尤其是在一个主库配置多个备库的情况下，某个备库可能因为某种原因，被关闭了好几天。等该备库重新运行后，它会向主库索要 WAL 记录。我们知道，为了避免 `pg_wal` 目录被撑满，PostgreSQL 有一些参数会控制该目录下的 WAL 文件的数量，如 `min_wal_size`, `max_wal_size` 和 `wal_keep_size` 等等。主库的检查点进程在执行检查点操作时，会根据这些参数的规定，删除老的 WAL 文件。这就存在一个可能性：这个长期怠工的备库需要的 WAL 文件已经被从 `pg_wal` 目录下删除掉了。备库无法获取所需要的 WAL 记录，所以它就无法追赶上主库，只能停滞不前。

如果主库配置了归档模式，我们可以在主库的归档目录中找到备库所需要的 WAL 文件，手工拷贝到备库的 `pg_wal` 目录下。因为备库的恢复进程会周期性的检查 `pg_wal` 目录，试图发现新的 WAL 文件，一旦它发现有了新的 WAL 文件，而且不是中断的，恢复进程就会继续执行恢复。这种手工的方法就是跳过 WAL 接收进程，自己搞定 WAL 记录的传输问题。这种办法虽然有效，但是比较“土”。为了解决 WAL 文件被删除导致备库无法工作的问题，PostgreSQL 引入了一个概念：复制槽 (replication slot)。假设备库收到的最后一条 WAL 记录的 LSN 是 LSN_1 ，它使用的复制槽会把这个位置记录下来，通知主库不要删除 LSN_1 和后面的 WAL 记录。复制槽的数据

在检查点操作中会刷新到磁盘上，确保即使主库重新启动，复制槽的数据也不会丢失。图 5.5 展示了复制槽的基本概念：

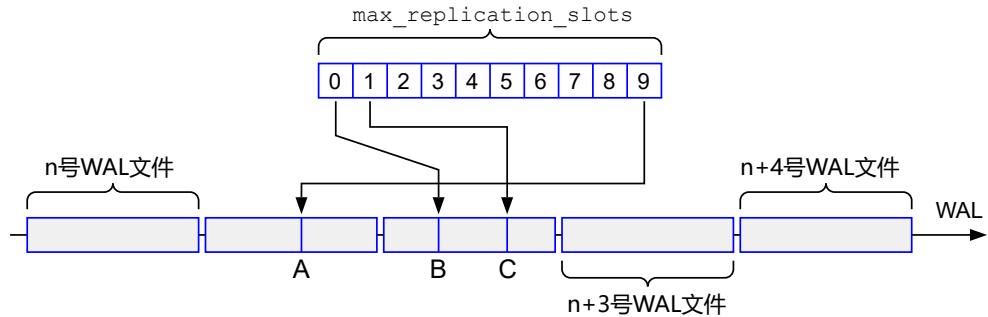


图 5.5：复制槽的概念

参数 `max_replication_slots` 控制整个数据库集群范围内最多有多少个复制槽，PostgreSQL 在共享内存中创建了一个数组，包含 `max_replication_slots` 个成员，每一个成员是一个复制槽。复制槽最重要的参数之一是 `restart_lsn`，它是防止 WAL 文件被删除的关键指标。在图 5.5 中，有三个活跃的复制槽，分别是 0 号，1 号和 9 号。它们都有自己的 `restart_lsn`，分别指向了 A、B 和 C 三个 LSN。其中 9 号复制槽的 `restart_lsn` 指向了 A 点，它是三个复制槽的 `restart_lsn` 中的最小值，指向了 $n+1$ 号 WAL 文件的某一个位置，则 n 号 WAL 文件或者更老的 WAL 文件可以被删除，但是 $n+1$ 号 WAL 文件和它后面的 WAL 文件必须保留，不能删除。

复制槽在对于防止备库所需要的 WAL 文件被删除方面很有用，但也会有副作用。复制槽有两种状态，活跃 (active) 或者不活跃 (inactive)。如果有客户端进程，譬如备库的 WAL 接收进程，使用了某个复制槽，则该复制槽处于活跃状态。如果没有任何客户端进程使用某个复制槽，则该复制槽处于非活跃状态。假设某个备库使用了 9 号复制槽，但该备库被关闭了很长时间，则 9 号复制槽长期处于不活跃状态，它的 `restart_lsn` 就固定在 A 点，无法往后移动。在这种情况下， $n+1$ 号 WAL 文件和后面的 WAL 文件始终无法删除。因为非活跃复制槽的存在，阻止了从某一点开始的 WAL 文件都不会被删除掉，所积累的 WAL 文件最后可能撑爆 `pg_wal` 目录所在的文件系统。为了解决这个问题，PostgreSQL 又引入了一个新的参数 `max_slot_wal_keep_size`，这个参数规定了复制槽保留在 `pg_wal` 目录下的 WAL 文件的体积的上限。它的缺省值是 -1，表示没有上限，此时就存在因为备库长期怠工导致主库磁盘被撑爆的可能性。你可以通过调整这个值的大小来控制复制槽所能保留的 WAL 文件体积的大小。

复制槽分为物理复制槽和逻辑复制槽，逻辑复制槽在我们学习逻辑复制的章节中再会讨论，本节只讨论物理复制槽。从生命周期角度分，复制槽有三种类型：永久性 (persistent) 的，短暂 (ephemeral) 的，和临时性 (temporary) 的。所谓永久性的，就是该复制槽的状态会被保存在磁盘上，具体来说就是 `pg_replslot` 子目录中。即使数据库集群被重新启动，永久性复制槽也不会被删除。短暂类型的复制槽不保存在磁盘上，它在数据库集群重启后就消失了。临时性的复制槽是会话级别的，它是比短暂性更短命的复制槽，就是一个会话结束后，或者发生错误后，它就被释放。DBA 主要关注的是永久性复制槽的管理，因为它可能会触发文件系统塞爆的事故。创建复制槽的函数是 `pg_create_physical_replication_slot()`，这个函数有三个参数：复制槽的名字，是否立即保存 WAL 文件，该复制槽是否是临时性的复制槽。下面的实验演示如何手工创建永久性复制槽。我们在主库的机器上执行如下命令：

```
oracle=# \! ls -l /opt/data/pgdata1/pg_replslot /* 首先查看对应磁盘目录，里面为空 */
total 0
/* 手动创建物理复制槽 */
oracle=# SELECT * FROM pg_create_physical_replication_slot('node1_slot', true);
 slot_name |      lsn
-----+-----
 node1_slot | 0/343D5B8
(1 row)
/* 再次查看对应磁盘目录，里面有了一个目录，其名字和复制槽的名字相同，里面有一个state文件 */
oracle=# \! ls -l /opt/data/pgdata1/pg_replslot
```

```
total 0
drwx----- 2 postgres postgres 19 Jan 27 16:00 node1_slot
oracle=# \! ls -l /opt/data/pgdata1/pg_replslot/node1_slot
total 4
-rw----- 1 postgres postgres 200 Jan 27 16:00 state
```

我们很清楚地看到：一开始在数据库集群目录下的 pg_replslot 子目录下没有任何文件。当我们创建了一个名字叫做 node1_slot 的复制槽，则在该目录下出现了同名的一个子目录，里面有一个文件 state，记录了复制槽在共享内存中的信息。检查点操作会把所有非空的复制槽信息写入到磁盘上，具体代码请参考 slot.c 中的 CheckPointReplicationSlots() 这个函数。复制槽的名字的长度最多 63 个字符，而且只能包括小写字母，数字和下划线，请参考 slot.c 中的 ReplicationSlotValidateName() 这个函数。我们在创建复制槽时指定了第二个参数为 true，要求立刻保留当前的 WAL 文件。PostgreSQL 会帮我们选择一个 LSN 作为该复制槽的 restart_lsn，并在创建复制槽函数结束后返回给我们，就是上面实验的 0/343D5B8。该 LSN 表明不能删除的 WAL 文件的下限。当然，如果有多个下限，就取最小值作为最终的下限。PostgreSQL 实际上是选择最近一次检查点的重做点作为复制槽的 restart_lsn，我们创建完复制槽后，趁着下一次检查点操作还没有发生，抓紧时间查询一下当前的检查点位置：

```
oracle=# select checkpoint_lsn, redo_lsn from pg_control_checkpoint();
checkpoint_lsn | redo_lsn
-----+-----
0/343D5F0      | 0/343D5B8
(1 row)
```

果然，我们看到了当前检查点的重做点就是 0/343D5B8，和我们预判的完全一致。系统视图 pg_replication_slots 是监控复制槽的主要接口之一。我们可以执行如下查询：

```
postgres=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-----+
slot_name          | node1_slot
plugin             |
slot_type          | physical
datoid             |
database           |
temporary          | f
active              | f
active_pid          |
xmin               |
catalog_xmin        |
restart_lsn         |
confirmed_flush_lsn |
wal_status          |
safe_wal_size       |
two_phase           | f
conflicting         |
```

下面把到目前为止我们能够理解的该系统视图中各列含义给大家介绍一下。第一列 slot_name 是复制槽的名字。第二列 plugin 对于逻辑复制槽才有意义，物理复制槽的这一列是空值。第三列是复制槽的类型，分为物理复制槽和逻辑复制槽两种。第四列 datoid 是复制槽连接的数据库的 Oid，第五列是数据库的名称，这两在逻辑复制槽中才有意义。第六列 temporary 表示该复制槽是否是临时性的。因为我们创建的复制槽是永久性的，所以这一列的值是 f。第七列 active 表示该复制槽是否处于活跃状态。如果复制槽处于活跃状态，第八列则是使用该复制槽的 WAL 发送进程的进程号。列 restart_lsn 我们已经讨论过了。如果数据库集群有多个复制槽，就取这些复

制槽的 restart_lsn 的最小值作为可以被删除的 WAL 文件的上限值，或者说是不可以被删除的 WAL 文件的下限值。下面我们在备库端演示如何使用这个复制槽。下面的操作在备库中执行：

```
$ cat /opt/data/standby/postgresql.conf | grep primary_slot_name
primary_slot_name='node1_slot'
/* 这个参数无需重启数据库集群，只要刷新一下配置文件即可 */
oracle=# select pg_reload_conf();
pg_reload_conf
-----
t
(1 row)

oracle=# SELECT pid, status, slot_name FROM pg_stat_wal_receiver;
 pid | status | slot_name
-----+-----+
 1173 | streaming | node1_slot /* 可以看到备库正在使用node1_slot这个复制槽 */
(1 row)
```

然后我们在主库上查询 pg_replication_slots，看看里面的状态信息是否发生了变化，结果如下所示：

```
oracle=# SELECT slot_name, active, active_pid, restart_lsn, wal_status FROM pg_replication_slots;
slot_name | active | active_pid | restart_lsn | wal_status
-----+-----+-----+-----+
node1_slot | t | 1377 | 0/343E6C0 | reserved /* 该槽处于active状态，被进程1377使用 */
oracle=# \! ps -ef | grep 1377 /* 使用这个复制槽的是WAL发送进程 */
postgres 1377 1088 0 16:17 ? 00:00:00 postgres: walsender kevin 192.168.137.17(56954) streaming
0/343E6C0
```

我们看到了，进程 1377 使用了该复制槽，这个进程就是 WAL 发送进程，同时我们观察到了该复制槽的 restart_lsn 从 0/343D5B8 变成了 0/343E6C0，说明该 LSN 在往后移动。主库是从备库那里获得 0/343E6C0 这个 LSN 的。该 LSN 是备库已经可靠地接收到的 LSN，自然在它之前的 LSN 备库就不需要了，可以被删除掉。因为删除只能是文件级别，不能删除一个 WAL 文件中的一条 WAL 记录，你从这个 restart_lsn 应该可以推算出能够被删除的 WAL 文件的上限是多少，这里就不再赘述了。对于处于非活跃的复制槽，如果你确信没有哪个备库在使用它，你可以使用 pg_drop_replication_slot() 函数进行手动删除，以确保 pg_wal 目录下的文件体积不会无限增大。

5.2.2 复制协议

物理复制，逻辑复制和 pg_basebackup 都使用流复制协议和主库进行通讯。流复制协议是一种比较简单的协议，总共也就几条命令而已，这些命令都是文本格式。下面我们可以使用 psql 来模拟流复制协议登录数据库集群，学习一下几种常用的复制协议命令，以期对底层的通讯过程有初步的了解。

```
$ psql "replication=yes host=192.168.137.16 user=kevin dbname=oracle"
Password for user kevin:
psql (16.1)
Type "help" for help.
/* 执行IDENTIFY_SYSTEM命令 */
oracle=> IDENTIFY_SYSTEM;
 systemid | timeline | xlogpos | dbname
-----+-----+-----+
 7321027155043554108 | 1 | 0/6000148 |
(1 row)
```

当客户端和远端的数据库集群通过复制协议建立连接后，首先执行的命令往往就是 IDENTIFY_SYSTEM。这条命令用来获取数据库集群的系统标识符，当前所处的时间线。xlogpos 是数据库已经可靠地写入到磁盘上的 WAL 记录的 LSN。dbname 是客户端连接的数据库的名字。因为我们在连接字符串中指定了 replication=yes，并没有指定数据库的名字，所以这一列为空。第二条常用的命令是 SHOW，它可以提取数据库集群的参数信息，类似 libq 协议连接数据库后执行的 SHOW 命令。譬如下面的操作显示了参数 wal_segment_size 和 full_page_writes 的值：

```
oracle=> SHOW wal_segment_size;
wal_segment_size
-----
16MB
(1 row)

oracle=> SHOW full_page_writes;
full_page_writes
-----
on
(1 row)
```

命令 CREATE_REPLICATION_SLOT/READ_REPLICATION_SLOT/DROP_REPLICATION_SLOT 是对复制槽进行创建，读取和删除的命令，具体的演示如下：

```
oracle=> CREATE_REPLICATION_SLOT myslot PHYSICAL; /* 创建一个物理复制槽 */
slot_name | consistent_point | snapshot_name | output_plugin
-----+-----+-----+
myslot   | 0/0           |          |
(1 row)

oracle=> READ_REPLICATION_SLOT node1_slot; /* 读取前面实验中创建的复制槽node1_slot的信息 */
slot_type | restart_lsn | restart_tli
-----+-----+-----+
physical | 0/6000148   |          1
(1 row)

oracle=> DROP_REPLICATION_SLOT myslot; /* 删除物理复制槽 */
DROP_REPLICATION_SLOT
```

真正执行流复制任务的指令是 START_REPLICATION，它分为物理复制和逻辑复制两种，具体指令格式如下：

```
START_REPLICATION [ SLOT slot_name ] [ PHYSICAL ] XXX/XXX [ TIMELINE tli ]
START_REPLICATION SLOT slot_name LOGICAL XXX/XXX [ ( option_name [ option_value ] [ , ... ] ) ]
```

其中第一条是物理复制的指令，第二条是逻辑复制的指令。对比两者个区别，我们可以看到，物理复制中复制槽是可选项，但是在逻辑复制中，复制槽是必选项。它们都要指定开始复制的一个起点 LSN，以及时间线。一旦发出这条指令，主库会源源不断地把 WAL 记录传送给客户端，但是我们无法在 psql 这种客户端中演示。图 5.6 展示了 START_REPLICATION 指令发出以后，来自主库的消息包的格式。

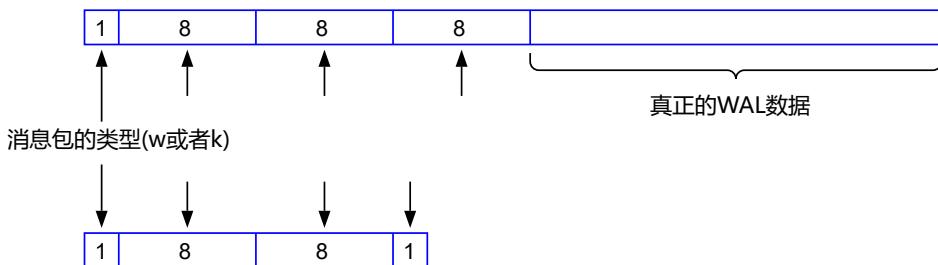


图 5.6: 流复制协议传输 WAL 记录的消息包的格式

当 WAL 接收进程接收到来自 WAL 发送进程传来的消息包后，它首先会检查第一个字节。这个字节是消息包的类型，共计两种类型。如果该字节是 w，表示该消息包是包含 WAL 记录的消息包；如果该字节是 k，则表示该消息包是一个心跳消息包 (keep-alive)。对于 w 类型的消息包，第一个字节后面连续的 24 个字节分为 3 个域，分别表示此次发送的 WAL 记录的起始 LSN，结束 LSN 和发送该消息包时主库的时间戳，再往后就是真正的 WAL 记录数据了。WAL 接收进程只需要“无脑”地把这些 WAL 记录写入本地的 WAL 文件中正确的位置（因为 LSN 本身就指定了 WAL 记录的存放位置），再通知恢复进程有新的数据进来了。对于 k 类型的心跳消息包，第一个字节后面的第一个 8 字节表示主库已经发给备库的最后一条 WAL 记录的 LSN，这个信息会保存在系统视图 pg_stat_wal_receiver 中的 latest_end_lsn 列。第二个 8 字节表示主库发送此消息包的时间戳。再紧挨着的一个字节表示主库是否需要备库回复，非零值表示需要备库的回复。图 5.7 展示了备库发往主库用于汇报 WAL 记录回放进度的消息包。

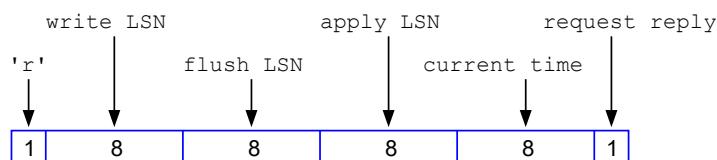


图 5.7: 流复制协议心跳消息包的格式

其中第一个字节 r 表示消息包的类型，它是“回复”(response)的意思。第二个域是 8 字节，表示 WAL 接收进程已经写入到备库磁盘上的最后一条 WAL 记录的 LSN。第三个域也是 8 字节，表示已经可靠地刷新到磁盘上的最后一条 WAL 记录的 LSN。第四个域也是 8 字节，表示恢复进程已经回放的最后一条 WAL 记录的 LSN。第五个域也是 8 字节，表示时间戳。第六个域是一个字节，表示是否需要主库回应。

5.3 流复制的监控

我们已经看到了，流复制的搭建是非常容易的。流复制在实践中运行的也非常可靠，几乎不需要进行维护。但因为网络中断或者主库的负荷突然增大等原因，备库和主库之间可能存在落差 (gap 或者 lag)。所谓落差，就是主库和备库的当前 WAL 指针位置的差异。我们知道，可以使用 `pg_current_wal_lsn()` 函数来检查数据库的最新的 LSN，所以你可以分别登录主库和备库，执行这条命令，获取两者最新的 LSN，这两个 LSN 相减得到的差值就是落差。注意：因为 LSN 表示在 WAL 空间的位置，虽然它们和时间概念紧密联系，但是两个 LSN 之间的差值的单位是字节，不是分秒等时间单位。对于主库和备库之间的落差的监控是维护流复制的主要工作。譬如我所在的公司，主备库的落差超过 10 分钟就开始发出警告，要求 DBA 进行排查。当主库和备库突然存在较大落差的情况下，一般不需要 DBA 做什么工作，只需要等网络恢复正常或者传输的数据量下降后，这个落差会自动消失。如果落差长期存在，DBA 就需要排查原因。DBA 排查的方向就是搞清楚什么原因导致了网络阻塞或者传输的 WAL 记录突然增加。所以理解主备库落差的含义是十分必要的。如果它们有相当大差异，就说明存在主备库不同步的问题。

在 WAL 接收进程和 WAL 发送进程的长连接中，WAL 接收进程会不断把备库的 WAL 记录的同步信息向主库汇报。主库提供了一个系统视图 pg_stat_replication，可以允许我们方便地查看主库和备库之间的各种信息，其执行结果如下：

```
oracle=# select * from pg_stat_replication; /* 该系统表只存在于主库上 */
-[ RECORD 1 ]-----+
pid          | 1377
usesysid     | 16384
username      | kevin
application_name | walreceiver
client_addr   | 192.168.137.17
client_hostname | 
client_port    | 56954
backend_start  | 2024-01-27 16:17:46.485845-05
backend_xmin   | 
state         | streaming
sent_lsn      | 0/343E6C0
write_lsn     | 0/343E6C0
flush_lsn     | 0/343E6C0
replay_lsn    | 0/343E6C0
write_lag      | 
flush_lag      | 
replay_lag     | 
sync_priority  | 0
sync_state     | async
reply_time    | 2024-01-27 17:06:17.845559-05
```

你有几个备库正在和主库连接，该系统视图就有几条记录，每条记录对应一个备库。在这系统视图中，第一列 pid 就是 WAL 发送进程的进程号。第二列和第三列分别是连接用户的 id 和名字。第四列 application_name 仅仅是一个字符串，用于区分备库。你可以在备库的 primary_conninfo 参数中设置这个字符串。以 client 开头的三列的含义不言而喻，分别代表备库的 IP 地址，主机名和端口号。backend_start 记录 WAL 发送进程启动的时间。state 列表示 WAL 发送进程和接收进程之间的状态变化，这个在前文中已经解释了，请参考 walsender_private.h 中的枚举类型 WalSendState 的定义。在备库正常工作的情况下这一列的值应该是 streaming，表示主库正在持续不断地把 WAL 记录发送给备库。后面的列里面包含了各种 LSN 的指标数据。请不要发晕，图 5.8 可以帮助我们形象地理解各项指标的含义：

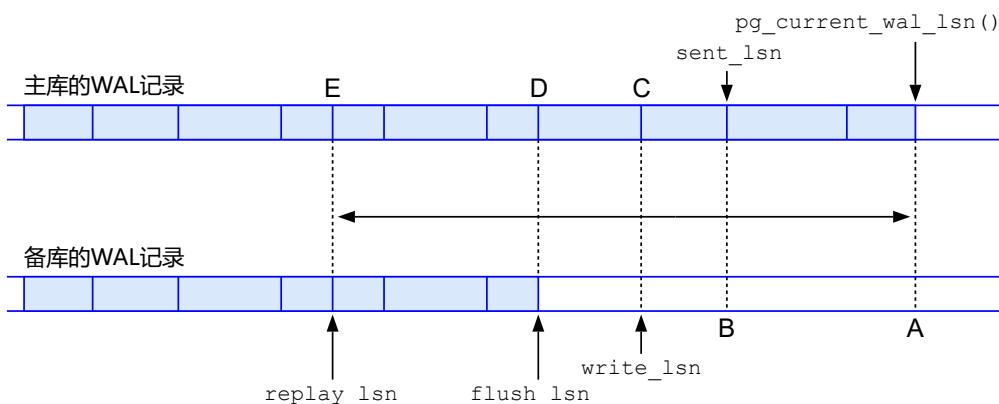


图 5.8：流复制的落差指标

结合上图，我们可以非常容易理解这些指标的含义。各种指标的含义具体如下：

- A 点是主库最新的 WAL 记录的位置，B 点是被网络发送到备库的 WAL 记录的最后的位置，两者之间的差异显示了被网络发送到备库的 WAL 记录和主库已经产生的 WAL 记录的差距。
- C 点是备库已经写到磁盘上的位置，D 点是数据真正落盘的位置。这里面涉及到文件系统的一些知识：当 PostgreSQL 把数据写入 (write) 磁盘后，数据可能并没有真正地写入到磁盘，而是保存在文件系统的缓存中，就是依然在内存里。只有缓存被刷新 (flush) 到磁盘上，数据才算真正地写入到磁盘。所以存在 write 和 flush 的区别。通常 C 和 D 两者没有什么区别。如果你看到了 C 和 D 的区别，就要研究如何调整文件系统的缓存问题了。
- 很显然由 B 到 C，是经过网络传输的，中间可能横跨几千公里，所以 B 点和 C 点之间的差异表明了网络延迟。如果两者的差距过大，表明网络延迟比较严重。
- E 点是恢复进程已经更新到数据文件的位置。因为一旦 WAL 接收进程把来自主库的 WAL 记录写入到本地的 WAL 文件中，恢复进程会立刻用它来修正数据文件。由于恢复进程是一个进程，目前还无法做到并发恢复，所以可能回放 WAL 记录成为瓶颈，造成 E 点和 D 点有较大的差异。

很显然，A 点和 E 点之间的差异是总的滞后值，也是我们监控系统最关心的指标。A 点的信息并不能从 pg_stat_replication 获取到，不过 A 点和 B 点通常一样，你计算 B 点和 E 点的差距就足够了。在备库上也有一个系统表 pg_stat_wal_receiver 可供我们观察备库接收 WAL 记录的情况，以及备库和主库通讯的时间信息。这个系统视图只有一条记录，因为每个备库只有一个 WAL 接收进程。

```
oracle=# select * from pg_stat_wal_receiver;
-[ RECORD 1 ]-----+
pid          | 1173
status        | streaming
receive_start_lsn | 0/3000000
receive_start_tli | 1
written_lsn    | 0/343E6C0
flushed_lsn    | 0/343E6C0
received_tli   | 1
last_msg_send_time | 2024-01-27 17:12:18.026232-05
last_msg_receipt_time | 2024-01-27 17:12:18.026636-05
latest_end_lsn | 0/343E6C0
latest_end_time | 2024-01-27 16:17:46.483234-05
slot_name      | node1_slot
sender_host     | 192.168.137.18
sender_port     | 5432
conninfo       | user=kevin password=***** .....
```

其中 receive_start_lsn 和 receive_start_tli 表示 WAL 接收进程启动时发往主库的起点 LSN。last_msg_send_time 是主库发送的最后一条消息包的时间戳，里面的时间信息是主库机器的。last_msg_receipt_time 是备库接收到最后一条来自主库的消息包的时间，这个时间是备库机器的时间。latest_end_lsn 是主库告诉备库它发送的最后一条 WAL 记录的 LSN，就是图 5.8 中的 B 点。其余各列的含义比较容易懂，不再赘述。你可以查阅官方文档，进一步了解具体的含义解释。

5.4 主备库之间的切换

流复制的主要作用有两个：第一个是做容灾。备库保持和主库的同步，万一主库挂掉了，备库可以被提升 (promote) 为主库，保证业务的中断时间最短。第二个作用是备库可以作为只读的数据库，把一部分只读的请求

分流过来，减轻主库的压力。在做容灾时，就存在一个主备库切换的问题，即把备库变成主库，主库变成备库。这种切换又分为两种，一个叫做正常切换 (switch-over)，一个叫灾难切换 (fail-over)。正常切换是按照计划执行的切换，譬如，我需要升级主库服务器里面的东西，为了不影响数据库的正常使用，临时性地把两者进行切换，工作完毕后可以再切换回去。因为这种切换是有计划地进行，不是突然事件，所以可以做到在切换时零数据丢失。灾难切换是在主库突然无法正常工作的情况下，紧急把备库变成主库，这个时候，老主库因为发生了故障，无法变成备库，就被从流复制的框架中踢出了。等备库接管了老主库的工作后，我们往往需要重新构建老主库，使之成为新的备库，用于下一次的切换。在灾难恢复的情况下，主库的部分数据有可能无法传递到备库，造成少量的数据丢失。下面我们分别介绍正常切换和灾难切换两种情况。

5.4.1 正常切换 (switch-over)

PostgreSQL 的正常切换过程是这样的：首先把老主库干净地关闭掉，即在 `pg_ctl stop` 命令中使用 `smart` 或者 `fast` 模式；然后把备库提升 (`promote`) 为新主库。此时流复制的链条就断掉了，我们需要把老主库变成新备库，重新恢复流复制的链条，这个就是正常切换。在执行切换之前，先确保主备库是同步的，就是通过查询 `pg_stat_replication` 的系统视图，确保没有非常大的落差，后面我们会分析这个问题。当确定主备库基本同步以后，我们分两步走，第一步是把主库关闭掉；第二步是把备库变成可读可写的新主库。注意两步的次序不能颠倒，后面会讲解其中的原因。首先我们在老主库的机器上以超级用户执行如下命令：

```
$ pg_ctl stop -D /opt/data/pgdata1 /* 干净地关闭主库 */
waiting for server to shut down.... done
server stopped

$ pg_controldata -D /opt/data/pgdata1 | grep state /* 检查主库是否被干净的关闭*/
Database cluster state:          shut down /* shut down表示主库被干净地关闭了 */
```

在主库关闭后，我们需要检查一个重要的指标，就是控制文件中的检查点。

```
$ pg_controldata | grep location | grep checkpoint
Latest checkpoint location:          0/4000028
Latest checkpoint's REDO location:   0/4000028
$ pg_waldump -n 1 -s 0/4000028
rmgr: XLOG      len (rec/tot):    114/    114, tx:           0, lsn: 0/04000028, prev 0/0343E6C0,
desc: CHECKPOINT_SHUTDOWN redo 0/4000028; tli 1; prev tli 1; fpw true; xid 0:744; oid 16395; multi 1;
offset 0; oldest xid 722 in DB 1; oldest multi 1 in DB 1; oldest/newest commit timestamp xid: 0/0;
oldest running xid 0; shutdown
```

从主库的控制文件中获得的检查点和重做点是一致的，说明这是一个 `SHUTDOWN` 类型的检查点。我们用 `pg_waldump` 检查一下，也验证了这个检查点是 `SHUTDOWN` 类型。这个重做点 `0/4000028` 很重要，它是老主库的“停止点”，即老主库止步于此，它的数据不再有任何变化了。下一步我们要把备库变成新主库，这个过程被称为备库的“提升” (`promote`)。在执行之前，我们在备库上执行如下命令：

```
oracle=# select pg_last_wal_receive_lsn();
pg_last_wal_receive_lsn
-----
0/40000AO
(1 row)
```

我们可以看到，上述 LSN 是备库已经收到的来自主库的最后一条 WAL 记录的 LSN，它是大于主库的控制文件的重做点 `0/4000028` 的。我们先记下这个事实，然后开始把备库提升为主库。提升备库的方法有两个，一个是利用 `pg_ctl` 的 `promote` 选项，另一个是执行 `pg_promote()` 的系统函数，两者并无本质上的不同。我们先使用 `pg_promote()` 函数执行，后续实验再使用 `pgctl`。我们在备库机器上以超级用户执行如下命令：

```

postgres=# SELECT pg_is_in_recovery(); /* 查看是否处于备库模式, 结果为t, 表示此时依然处于备库模式 */
pg_is_in_recovery
-----
t
(1 row)

postgres=# SELECT pg_promote(); /* 把备库升级为主库, 也可以使用pg_ctl promote -D xxxxx的命令 */
pg_promote
-----
t
(1 row)

postgres=# SELECT pg_is_in_recovery(); /* 查看是否处于备库模式, 结果为f, 表示该数据库已经是主库了 */
pg_is_in_recovery
-----
f
(1 row)

```

当备库提升成功以后, 你会发现恢复进程和 WAL 接收进程都消失了。这个可以理解, 因为它们是备库特有的进程, 现在备库变主库, 它们已经完成了历史使命, 就自动退出了。那么提升备库的指令到底做了什么事情呢? 图 5.9 展示了提升备库的具体过程:

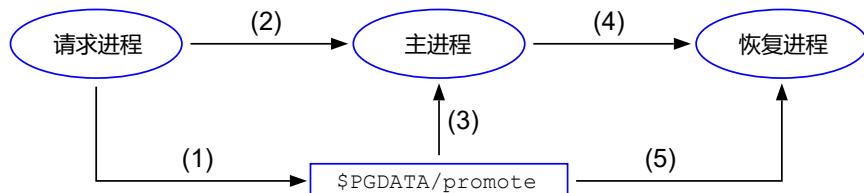


图 5.9: 备库切换为主库的过程

提升备库的第一步是请求进程先在数据库集群目录下创建一个信号文件 `promote`。这个文件类似 `standby.signal` 和 `recovery.signal`, 仅仅作为一个信号存在, 里面的内容不重要。请求进程可以是 `pg_ctl`, 也可以是我们执行 `pg_promote()` 函数的那个后端进程。第二步是请求进程向主进程发送 `SIGUSR1` 信号, 通知主进程进行备库的提升。第三步是主进程在接收到信号后, 检查 `promote` 信号文件是否存在。如果存在, 主进程就向恢复进程发送 `SIGUSR2` 信号。第四步是恢复进程接收到主进程的信号后, 检查 `promote` 信号文件是否存在。如果存在, 恢复进程就会把手头上已有的 WAL 记录回放完毕, 然后设置数据库集群状态为可读可写的模式, 最后事了拂衣去, 深藏身与名。从这个过程我们可以看到, 所谓备库的提升, 就是一次 PITR, 基于时间点的恢复。这个 PITR 的恢复终点是备库所拥有的 WAL 记录的尽头, 也就是 `0/40000A0` 这个位置。当恢复进程回放到这个位置后, 就会产生一个新的时间线, 后续的修改操作在新时间线上继续运行。我们可以在备库的 `pg_wal` 目录下看到这个时间线的历史文件:

```

$ pwd
/opt/data/standby/pg_wal
$ ls -l *.history
-rw----- 1 postgres postgres 41 Jan 27 18:45 00000002.history
$ cat 00000002.history
1      0/40000A0      no recovery target specified

```

我们可以看到, 备库是在 `0/40000A0` 这个 LSN 上从时间线 1 升级到时间线 2 的。这个点称为备库的主库的“分叉点”(divergence), 也可以称之为“切换点”。备库提升后, 老备库, 此时已经是新主库了, 和老主库的关系

可以用图 5.10 来表示，其中 B 点是分叉点。

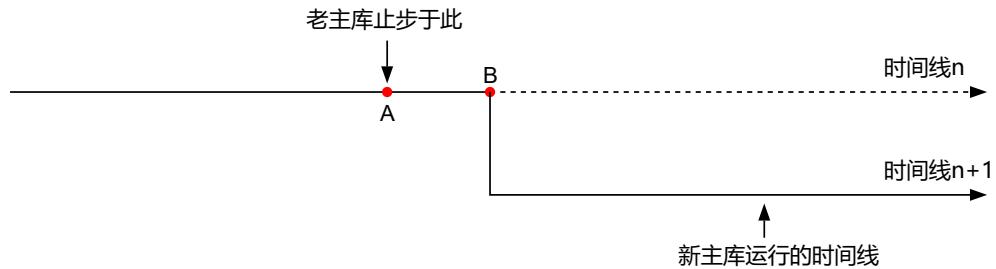


图 5.10: 切换点在主库的停止点之后

我们可以看到，老主库停止在了 A 点，备库在 B 点，也就是分叉点切换成了新主库。A 点在 B 点之前。如果在切换之前，备库和主库存在很大的落差，在老主库关闭以后，备库在老主库的停止点之前做了提升，有可能出现图 5.11 所示的情景，就是切换点在主库的停止点之前。

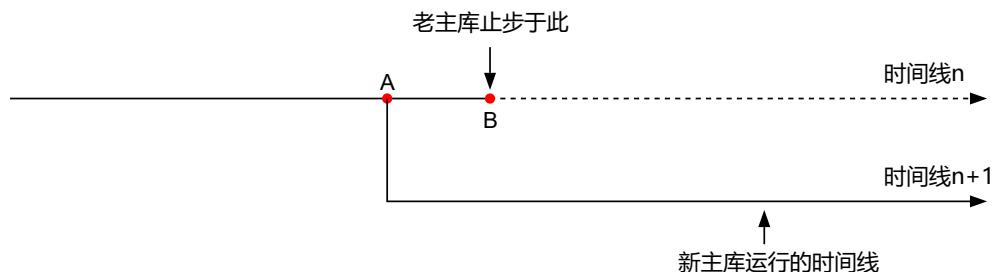


图 5.11: 切换点在主库的停止点之前

主库和备库的分叉点在老主库停止点之前还是之后，决定了老主库是否能够顺利地变成新备库。如果分叉点在老主库的停止点之后，或者和老主库的停止点相同，这种切换被称为正常切换。反之，我们把这种切换称为灾难切换，后面我们会讨论这种情况。

备库被成功提升为主库后，我们有两个主库：已经关闭的老主库和由老备库转化而来的新主库，而且不再存在流复制的网络连接了。我们下一步的任务是要把老主库变成新备库，所需要做的工作有两项。第一项工作是在老主库的数据库集群目录中创建一个 `standby.signal` 的信号文件。当老主库启动后，看到了这个文件，就知道自己已经从领导岗位上退下来了，要变成备库的角色。第二项工作是我们需要告知老主库如何连接新主库，这就是通过设置 `primary_conninfo` 参数来完成。它的内容和前文创建第一个备库的实验没有任何本质区别，无非就是修改一下 IP 地址指向新备库而已。注意：此时老主库上并没有 `backup_label` 文件，所以当它启动后，它会从控制文件中读取检查点和重做点进行恢复。我们在老主库的机器上执行如下命令：

```
$ pwd
/opt/data/pgdata1
$ touch standby.signal /* 创建信号文件，确保启动后，老主库进入备库模式 */
$ cat postgresql.conf | grep primary_conn /* 添加primary_conninfo参数，注意IP地址是老备库机器的 */
primary_conninfo = 'host=192.168.137.17 port=5432 user=kevin password=123456'
$ pg_ctl start -l /home/postgres/newstandby.log -D /opt/data/pgdata1 /* 启动老主库 */
waiting for server to start.... done
server started
```

然后你就可以检查老主库是否变成了新备库，检查方法和我们第一次创建备库的实验中使用的命令是一样的。如果流复制是一对多的情况，就是一个主库带两个或者更多个备库，当把一个备库提升为主库后，其余的备库存在一个“跟随”(follow)的问题，即其余的备库不再指向老主库，而是要改换门庭，指向新主库。这个工作

很简单，因为老主库的原先的小兄弟，就是老的备库，无论如何，不可能恢复到超越分叉点的位置。我们只需要修改其它老备库的 `primary_conninfo` 参数，指向新主库即可。因为这个参数的生效需要重启数据库，所以修改完毕后，需要把这些老备库重新启动一下。老备库的 `standby.signal` 信号文件始终存在，所以无需重新创建。由此看来，相对于其它老备库的追随，老主库变成新备库只是额外多了一步，即创建 `standby.signal` 信号文件。

我们可以想象一下老主库启动后发生了什么事情。老主库意识到自己是备库模式后，会读取控制文件中的重做点，告诉新主库自己目前所处的位置。由于新主库和老主库已经处于不同的时间线上，所以新主库除了给新备库，就是老主库，发来新的 WAL 记录以外，也会发送时间线历史文件。新备库根据时间线历史记录就可以知道从哪一点切换到新的时间线，从而顺利追赶上新备库的脚步。在复制协议中有一条指令 `TIMELINE_HISTORY` 就是专门做这件事情的。我们以复制协议登录新主库，执行这条命令，看看发生了什么：

```
$ psql "replication=yes host=192.168.137.17 user=kevin dbname=oracle"
Password for user kevin:
psql (16.1)
Type "help" for help.
/* 从主库上索取时间线2的切换记录 */
oracle=> TIMELINE_HISTORY 2;
filename | content
-----+
00000002.history | 1      0/40000AO      no recovery target specified+
(1 row)
```

我们可以清楚地看到，`0/40000AO` 是从时间线 1 切换到时间线 2 的切换点。只要老主库的恢复起点比这个 LSN 小，就可以顺利地从时间线 1 的赛道切换到时间线 2 的赛道。在 WAL 接收进程和 WAL 发送进程进行网络通讯的过程中，已经充分考虑了主库和备库处于不同时间线的可能性，必要的时间线切换文件会被传输到备库，指导备库进行时间线的自动切换。我们所做的工作就是必须保证老主库的停止点在切换点之前，才能够顺利地把老主库变成新备库。所以在正常切换过程中，必须先关闭应用，禁止老主库接收任何新的数据请求，以免产生新的 WAL 记录。然后我们需要等待备库逐步追赶上主库。因为不再有新的 WAL 记录产生，你总能等到一个时刻，在这个时刻，你会看到 `pg_stat_replication` 这个系统视图中的 `sent_lsn / write_lsn / flush_lsn / replay_lsn` 这四列的 LSN 完全一致。此时提升主库，就可以确保主库的停止点一定在切换点之前。如果主库的停止点在切换点之后，也就是图 5.11 所表示的情况，就是我们下面要讨论的灾难切换的内容。

5.4.2 灾难切换 (fail-over)

在正常切换的实验中，第一步是关闭主库，第二步是提升备库。这两步的次序不能颠倒，否则老主库无法正常变成新备库。如果我们在不关闭主库的情况下直接提升备库，会出现什么情况呢？图 5.12 展示了这种情景：

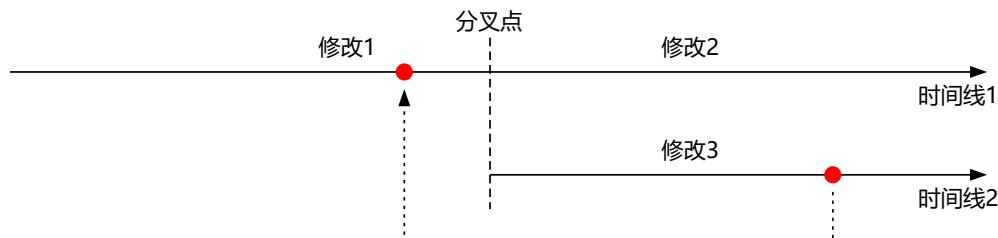


图 5.12：灾难切换的基本概念

我们可以看到，一开始主库和备库都是在时间线 1 上运行，修改 1 发生在切换之前，所以它会在主库和备库中都进行更新。如果主库在切换之前没有关闭，直接把备库提升为新主库，那么，老主库和新主库在切换点这

个时刻就开始分家了，各走各的道儿。修改 2 只发生在老主库上，修改 3 则只发生在新主库上，两者并无任何关系，各自独立进行。老主库自然无法变成新主库的备库。我们必须有一种方法，把修改 2 去掉，让老主库回到分叉点之前的状态。一个直观且简单的方法就是删除老主库，重新利用 pg_basebackup 创建一个新备库。相信读者已经掌握了这种方法了。但这种方法对于体积巨大的生产库来说，是非常耗时的。举个实际的例子：我公司的生产库体积是 4TB，构建一个新备库需要好几个小时。实际上老主库和新主库的绝大部分数据都是一样的，重新创建一个新备库的大部分时间都在做无用功。那么能不能只修正老主库上那些从分叉点开始的数据块呢？这样就无需重新创建体积巨大的数据库，有效地缩短了把老主库变成新备库的时间。PostgreSQL 提供了一个数据库回卷工具 pg_rewind，它的作用就是读取新老主库的数据进行对比，找到分叉点，然后对老主库进行回卷，让老主库回退到分叉点之前的某一个重做点。

pg_rewind 的工作原理解释起来也不难，共计分为五大步。我们首先关闭老主库，让它不再有新的 WAL 记录产生。第二步是找到分叉点。这个不难，直接在新主库的 pg_wal 目录下就能找到对应的时间线历史文件，分叉点就在历史文件中。第三步是从分叉点开始往后扫描老主库上的 WAL 记录。因为老主库已经关闭了，它的 WAL 记录总有一个尽头。从分叉点到 WAL 记录的尽头扫描一遍后，我们分析这些 WAL 记录，就可以很清楚地知道从分叉点以后，老主库哪些数据块发生了改变，我们记录这些数据块，形成一个集合。第四步是根据第三步得到数据块集合，把新主库上的数据块拷贝过来，覆盖掉老主库的数据块。就可以把老主库从分叉点开始后的修改都消除掉。如果这些数据块只占整个数据库集群全部数据块极小的一部分，第三步和第四步所需要花费的时间相比较重新建立一个备库，毫无疑问会大大缩短。这就是 pg_rewind 的最大价值。第五步是把老主库的控制文件中的重做点变成分叉点之前的某一个重做点，这个工作很简单，就是从分叉点往前扫描老主库的 WAL 记录，得到一个检查点 WAL 记录后，找到它的重做点，产生一个 backup_label 文件即可。

以上五个步骤都做完后，重新启动老主库，根据我们目前掌握的知识，老主库就会从新的重做点开始执行恢复。关于时间线的切换，我们也无需担心，因为新备库会把时间线历史文件发给老主库，指导它在分叉点处切换赛道。下面我们就通过一个实验来了解 pg_rewind 的基本使用。首先按照前文中的搭建流复制的实验环境，把主库和备库建立起来。主库的 IP 地址是 192.168.137.16，备库的 IP 地址是 192.168.137.17。我们先在主库上创建一个新的数据库集群，执行如下命令：

```
$ initdb -D /opt/data/pgdata1 /* 创建一个新的数据库 */
/* 修改 /opt/data/pgdata1/postgresql.conf，在文件最后加入如下参数 */
listen_addresses = '*'
wal_level = replica
hot_standby = on
max_wal_senders = 10
max_replication_slots = 10
archive_mode = on
archive_command = 'test ! -f /opt/data/a/%f && cp %p /opt/data/a/%f'
wal_log_hints = on # 这是一个新参数
/* 修改 /opt/data/pgdata1/pg_hba.conf，在文件最后加入如下参数 */
host      all            all            192.168.137.0/24      md5
host      replication    all            192.168.137.0/24      md5
$ mkdir /opt/data/a      /* 创建归档目录 */
$ pg_ctl -D /opt/data/pgdata1 -l logfile start      /* 启动数据库*/
$ psql -U postgres
postgres=# \password /* 以超级用户 postgres 登录，修改一下口令 */
Enter new password for user "postgres": /* 口令为 123456 */
Enter it again:
```

以上这些操作在搭建流复制实验中已经做过了，相信大家很熟悉了。这里有两点不同，第一点是 pg_hba.conf 中除了允许客户端以复制协议连接服务器，还要允许以普通的 libpq 协议访问服务器，这是因为 pg_rewind 需要使用普通协议访问远程数据库。第二点是必须设置参数 wal_log_hints=on。这个参数和 full_page_writes 一样，打

开全页写模式。它和 full_page_writes 不一样的地方是，对一些不重要的修改，也会执行全页写操作。下面的英文解释可以方便英文程度较高的读者更权威地理解这个参数的含义。

```
When wal_log_hints parameter is on, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint, even for non-critical modifications of so-called hint bits. If data checksums are enabled, hint bit updates are always WAL-logged and this setting is ignored.
```

回顾我们第二章对记录格式的讲解，我们知道每条记录都有 infomask 等域，里面不同的比特表示不同的含义。有些修改操作仅仅修改了这些比特，并没有修改真正的数据。参数 wal_log_hints 打开以后，对于这些微小的修改，如果是在检查点执行完毕后第一次修改，则也要做全页写。这个特性保证我们不会遗漏所有被修改的数据块。而且这个参数必须是在切换之前就要设置好，而且设置好之后，最好在切换之前做一下手动检查点，确保检查点后面的 WAL 记录都可靠地记录了所有可能的被修改的数据块。如果你的数据库在创建之初，使用了 initdb 的 -k 参数打开了数据页校验功能，这个参数的功能自然就有了。在这种情况下，你可以不设置这个参数。因为我们的数据库已经存在了，可能没有打开校验码功能，所以我们可以使用 wal_log_hints 参数来补救。我们使用 postgres 这个超级用户来执行流复制任务。我们在备库机器上执行如下命令创建一个备库：

```
/* 创建备库 */
$ pg_basebackup -h 192.168.137.16 -U postgres -D /opt/data/newprim -Fp -R -c fast -X stream
$ mkdir /opt/data/a /* 创建归档目录 */
$ pg_ctl start -l logfile -D /opt/data/newprim /* 启动备库 */
```

相信上述实验步骤读者已经可以很轻松地完成了。在流复制环境准备完毕后，我们现在假定主库不能正常工作了，譬如它的网络连接不了，我们无法对主库做任何事情。现在只能硬着头皮，首先把备库提升为新主库，请在备库的机器上执行如下命令：

```
$ pg_ctl promote -D /opt/data/newprim /* 把老备库变成了新主库 */
waiting for server to promote.... done
server promoted
$ ls -l newprim/pg_wal/*.history
-rw----- 1 postgres postgres 41 Jan 27 20:01 newprim/pg_wal/00000002.history
$ cat newprim/pg_wal/00000002.history
1      0/3046C88      no recovery target specified
```

对备库的提升，实质上就是做一次 PITR 恢复，它会产生新的时间线，所以有一个分叉点。我们从时间线变更文件 00000002.history 中可以看到，分叉点为 0/3046C88，备库就是从这个点开始和老主库分道扬镳的。假设当备库变成新主库以后，老主库又恢复了正常工作。这时你无法按照前文中的办法顺利地把老主库变成新备库。为了让老主库能够变成新的备库，要么删除掉主库重新使用 pg_basebackup 创建新备库，但是所需要的时间很长。要么使用 pg_rewind 这个法宝把老主库的状态回滚到分叉点之前的某一个点，这个点就是最靠近分叉点的一个重做点。毫无疑问，只要切换后不久就使用 pg_rewind 对老主库进行修正，因为此时从分叉点开始被修改的数据块的数量相对于整个数据库集群的数据块的总数来说只是极小的一部分，所以它需要的修复时间要比重新创建备库要少非常多。

下面我们正式使用 pg_rewind 工具把老主库回退到分叉点之前。第一步我们关闭老主库，下面的操作都在老主库的机器上完成的。

```
$ pg_ctl stop -D /opt/data/pgdata1      /* 关闭老主库 */
waiting for server to shut down.... done
server stopped
```

pg_rewind 命令提供了一个-n 参数，它表示是预演 (dry-run)，仅仅是做一些检查工作，并不会有任何修改操作。所以我们先用这个参数预演一下：

```
$ pg_rewind -D /opt/data/pgdata1 -P -n --source-server='host=192.168.137.17
> port=5432 user=postgres password=123456 dbname=postgres'
pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/3046C88 on timeline 1
pg_rewind: error: could not open file "/opt/data/pgdata1/pg_wal/00000001000000000000000003":
           No such file or directory
pg_rewind: error: could not find previous WAL record at 0/3046C88
```

上面的错误显示 000000010000000000000003 这个 WAL 文件在 pg_wal 目录下不存在。它存在于我们的归档目录/opt/data/a 里面。你可以手工把这条 WAL 记录拷贝到 pg_wal 目录下，继续重复上面的预演命令。如果还有别的 WAL 文件缺失，你如法炮制，从归档目录中找到这些 WAL 文件，统统手工拷贝到 pg_wal 目录下，直到预演成功。如果出现类似下面的输出结果，就说明预演成功了。

```
$ pg_rewind -D /opt/data/pgdata1 -P -n --source-server='host=192.168.137.17
> port=5432 user=postgres password=123456 dbname=postgres'
pg_rewind: connected to server
pg_rewind: servers diverged at WAL location 0/3046C88 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/2000060 on timeline 1
pg_rewind: reading source file list
pg_rewind: reading target file list
pg_rewind: reading WAL in target
pg_rewind: need to copy 51 MB (total source directory size is 69 MB)
53192/53192 kB (100%) copied
pg_rewind: creating backup label and updating control file
pg_rewind: syncing target data directory
pg_rewind: Done!
```

在上面的输出中，我们可以看到，分叉点是 0/3046C88，pg_rewind 扫描了 51MB 的数据块需要从新主库拷贝到老主库。注意 pg_rewind 中的 source-server 参数，指的是新主库的服务器，不是原来的老主库所在的服务器，因为现在新主库是“源”，而老主库是“目标”，请不要搞混了。如果预演成功，就去掉-n 参数，使用 pg_rewind 真刀真枪地把老主库恢复到分叉点之前的某一个状态：

```
$ pg_rewind -D /opt/data/pgdata1 -P --source-server='host=192.168.137.17
> port=5432 user=postgres password=123456 dbname=postgres'
```

等回退成功之后，我们看到老主库中有一个 backup_label 的文件，它是 pg_rewind 创建的，里面记录了从分叉点往前的第一个检查点。老主库重新启动时会从这一点开始做恢复工作。

```
$ cat backup_label
START WAL LOCATION: 0/2000028 (file 000000010000000000000002)
CHECKPOINT LOCATION: 0/2000060
BACKUP METHOD: pg_rewind
BACKUP FROM: standby
START TIME: 2024-01-27 22:06:37 EST
```

为了要让老主库启动后进入备库模式，我们还需要生成一个 standby.signal 信号文件，同时修改 primary_conninfo 指向新主库。pg_rewind 已经把 primary_conninfo 参数写入到了 postgresql.auto.conf 中了，但是 IP 地址还是老的，所以我们要修改一下：

```
$ touch /opt/data/pgdata1/standby.signal /* 创建备库信号文件 */
$ cat postgresql.auto.conf | grep primary
```

```
primary_conninfo = 'user=postgres password=123456 host=192.168.137.17 port=5432 ....'
$ pg_ctl start -l standby.log -D /opt/data/pgdata1 /* 启动老主库，现在它变成了新备库 */
waiting for server to start.... done
server started
```

经过检查，老主库果然变成了新备库，顺利服役了。pg_rewind 利用分叉之后老主库上修改的数据块的个数只占全部数据块的极少的一部分这个规律，大大地缩短了把老主库变成新备库的时间。它在对付生产库切换失败的问题上是一把利刃。我们需要思考和理解其背后的工作原理，才能够在实践中得心应手。

5.5 从备库上执行备份

我们知道，备份是比较耗时的工作。譬如我负责的一个数据库的体积是 5TB。每天晚上有一个备份脚本运行对它进行全备份，需要 6 个多小时才能完成。为了不影响主库的性能，我们希望把备份这种影响数据库性能的操作移到备库上来进行操作。在备库上备份的一个核心问题是备库如何归档。我们知道备库是被动的接收来自主库的 WAL 记录。等主库归档后，备库并不会自动归档。如果想让备库也归档，需要设置备库的 archive_mode=always。图 5.9 展示了在备库上进行归档的要点：

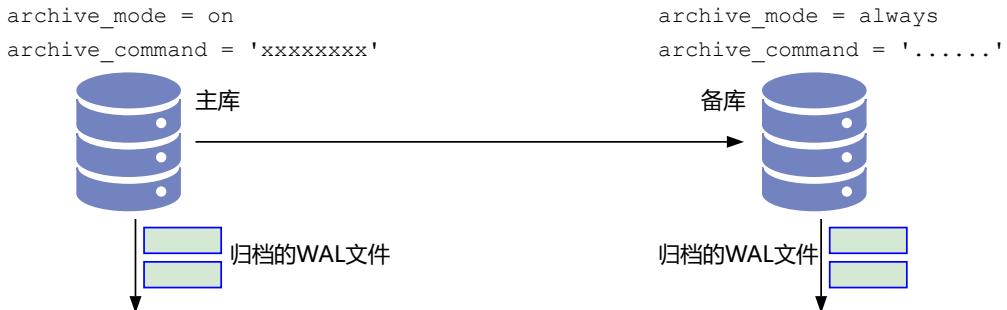


图 5.13：在备库上进行归档

当你设置备库为 always 以后，你可以在主库上执行 pg_switch_wal() 手动切换和归档 WAL 文件，然后分别在主库和备库上查询系统视图 pg_stat_archiver，观察两边的归档情况。对于主库而言，如果它的 archive_mode 参数设置为 on 或者 always，效果是一样的。

在第四章备份和恢复的学习中，我们学习的是如何在主库上做备份。我们已经知道，备份开始的时候必须要执行两个动作，一个是设置全页写模式，第二个是触发一个检查点。只有在这两个动作完成之后，你才能拷贝数据文件。拷贝结束后，还会在 WAL 文件中插入一条备份结束的 WAL 记录。现在问题来了，因为备库的 WAL 文件百分百来自主库，备库是无法修改 WAL 文件的，只能“消费”它，所以备库无法控制全页写和插入备份结束 WAL 记录这种操作。那么如果要在备库上做一个有效的备份，该怎么做呢？

对于全页写的问题，做法很简单，那就是：如果你想在备库上做备份，你必须在主库上设置全页写模式，即主库的 full_page_writes 的值设置为 on，确保在任何时刻全页写模式都是打开的，备库无需做任何事情。对于第二个问题，即需要在写 WAL 记录，PostgreSQL 引入了一个新的概念：restart point，我们可以翻译为“重启点”，以示和重做点这个重要的概念有所区别。重起点做的内容和检查点几乎一样，但是它不会往 WAL 文件中写入记录。

第六章 堆表和 B 树索引

在第二章我们学习了表的基本结构。表就是若干固定长度的数据块组成的一个线性结构。在后文的学习中，涉及到表和索引之间关系的理解，所以本章讲解表和最常用的 B 树索引，作为后面学习内容的预备知识，目的是帮助读者建立起表和索引的关系的概念。PostgreSQL 支持的索引类型非常多，这是 PostgreSQL 的一大特色。但是 90% 的应用场景都是使用 B 树索引，所以本章只讲解 B 树索引。其余的索引类型会单独成为一章，作为对索引的研讨专题。

6.1 堆表

数据库中表有很多种类型，但在 PostgreSQL 中只有堆表 (heap table) 一种类型的表。所谓堆 (heap)，指的是像草堆一样乱七八糟堆集在一起，没有固定的顺序。顾名思义，堆表指的是表中的记录并没有固定的顺序。我们可以用图 6.1 理解堆表：

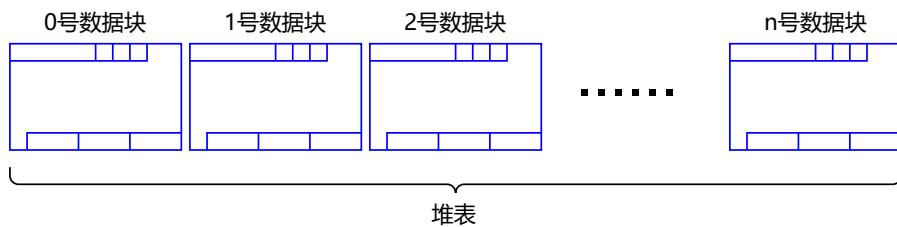


图 6.1：堆表的整体结构

堆表是若干数据块组成，这些数据块的编号从 0 开始，依次增加。无序是堆表的最本质的特征。这里的无序是指表中的记录是无序的，并不是指数据块，堆表的数据块是有序排列的。

6.1.1 TOAST

Postgresql 的表是由固定大小的数据块组成，数据块的缺省尺寸是 8KB 字节，这就意味着在一个数据块中无法存储超过 8KB 的数据。PostgreSQL 不允许一条记录跨数据块存储，即任何一个数据块中都包含完整的记录。如果一条记录的某列的长度超过 8KB 该怎么办呢？我们可以采用压缩 (compress)，分割 (split) 和行外存储 (out-of-line storage) 这三种技术来解决这个问题。基本的解决思路就是先把这一列的数据压缩，看看能否放在原数据块中。如果放不下，就进行分割后保存在别的地方。然后在原表中保存一个指针指向被压缩、分割的数据，这就是行外存储的含义。这种技术被称为超长列存储技术 (TOAST: The Oversized-Attribute Storage Technique)。本节我们就来研究堆表中的 TOAST 技术。注意：并不是所有的数据类型都支持 TOAST。譬如整型 (int) 只有 4 个字节，很显然不需要 TOAST 技术。只有那些数据长度可变的数据类型，如 varchar, text, json, jsonb 和 bytea 等才支持 TOAST 存储。被压缩和分割的数据保存在什么地方呢？如果原表的某一列可以用 TOAST 技术来保存，那么 PostgreSQL 就会创建一个相应的 TOAST 表来保存这些超长数据。

下面我们通过一个实验来体验一下 TOAST 技术。首先我们创建一张测试表，具体操作如下：

```
oracle=# create table blog(id int, title varchar, content text);
CREATE TABLE
oracle=# \d+ blog
              Table "public.blog"
 Column |      Type       | Storage | Compression | Stats target | Description
-----+----------------+-----+-----+-----+-----+
```

```

id      | integer          | plain   |           |
title   | character varying | extended |           |
content | text            | extended |           |
Access method: heap

```

在测试表 blog 中，列 title 的数据类型是 varchar，列 content 的数据类型是 text。这两种数据类型都是支持 TOAST 技术的可变长度的数据类型。在 PostgreSQL 的表中，每一列都有一个存储方式的属性。由上面的实验结果我们可以看到：id 这一列的存储方式是 plain，而其余两列的存储方式是 extended。PostgreSQL 支持四种存储方式，具体如下：

- **PLAIN**：避免压缩和行外存储。只有那些不需要使用 TOAST 技术就能存放的数据类型使用此种存储方式，如 int, boolean 类型。
- **EXTENDED**：允许压缩和行外存储。一般会先压缩，优先保存在原数据块中。如果还是太大，在原数据块中放不下，就会使用行外存储，保存在相应的 TOAST 表中。
- **EXTERNAL**：允许行外存储，但不许压缩。类似字符串这种会对数据的一部分进行操作的字段，因为不需要读取出整行数据再解压，采用此方法可能获得更高的性能，但是它的代价就是需要的存储空间比较大。
- **MAIN**：允许压缩，但不许行外存储。不过实际上，如果超长数据被压缩后都无法保存在原数据块中，行外存储作为最后手段还是会被启动。因此可以把这种方法理解为：尽量不使用行外存储。

以上四种存储方式在源代码中，相关的定义如下：

```

/* in src/include/catalog/pg_type_d.h */
#define TYPSTORAGE_PLAIN      'p' /* type not prepared for toasting */
#define TYPSTORAGE_EXTERNAL    'e' /* toastable, don't try to compress */
#define TYPSTORAGE_EXTENDED   'x' /* fully toastable */
#define TYPSTORAGE_MAIN        'm' /* like 'x' but try to store inline */

```

既然测试表 blog 中的列的数据类型支持 TOAST 存储，那么如何它对应的 TOAST 表呢？这里面存在一个简单的规律：如果一个表的 Oid 是 12345，那么它对应的 TOAST 表就是 pg_toast 下的 pg_toast_12345 这张表。我们执行如下查询来确定测试表 blog 对应的 TOAST 表的名称：

```

oracle=# select relfilenode,reltoastrelid,reltoastrelid::regclass from pg_class
oracle# where relname='blog';
  relfilenode |  reltoastrelid |      reltoastrelid
-----+-----+-----
       16462 |         16465 | pg_toast.pg_toast_16462
(1 row)

oracle# \d+ pg_toast.pg_toast_16462
TOAST table "pg_toast.pg_toast_16462"
 Column | Type   | Storage
-----+-----+-----
 chunk_id | oid    | plain
 chunk_seq | integer | plain
 chunk_data | bytea  | plain
Owning table: "public.blog"

Indexes:
  "pg_toast_16462_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method: heap

```

我们可以看到：因为 blog 表的 Oid 是 16462，所以它对应的 TOAST 表的名称是 pg_toast.pg_toast_16462。所有的 TOAST 表的结构都是一样的，共分为 3 列：第一列 chunk_id 是原表中相关记录的唯一性标识，即原表中

不同的记录会被分配一个唯一的 id，就是这个 chunk_id。一条记录压缩后可能会被分割成不同的部分，被称为 chunk，我们可以翻译成“数据条”。同一条记录的不同数据条的序列号由第二列 chunk_seq 来区分。很显然 seq 是 sequence 的缩写，序列号从 0 开始计数，依次加 1。只要按照序列号排序，把属于同一 chunk_id 的不同的数据条连接起来，就能够得到完成的数据。第三列 chunk_data 很显然就是保存了真正的数据，这个数据可能压缩了，也可能没有压缩，取决于存储方式。原表和 TOAST 表的关系，可以使用图 6.2 来进行理解：

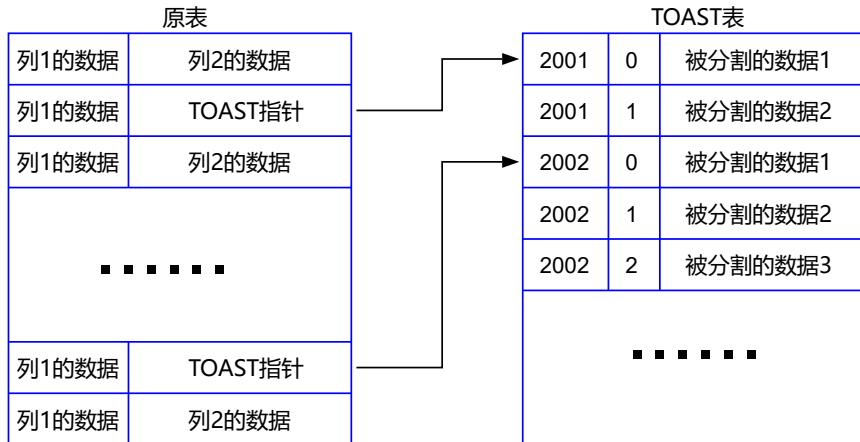


图 6.2：原表和 TOAST 表的关系

由上图可知：如果原表中的数据经过压缩后可以保存在原表中，就不会保存在对应的 TOAST 表中。只有放不下的数据才保存在 TOAST 表中，同时原表中保存一个 TOAST 指针 (toast pointer) 指向了 TOAST 表中对应的记录。你可能注意到了，TOAST 表有一个主键 (primary key)，由 chunk_id 和 chunk_seq 组成，参见上面实验中的 pg_toast_16462_index 索引。如果我们能够拿到 chunk_id，就可以迅速定位全部的数据。所以我们猜想：在原表中的 TOAST 指针中一定包含这个 chunk_id 的信息，后面的实验会验证这一点。

关于数据的压缩方式，目前 PostgreSQL 支持两种压缩方式，pglz 和 lz4，未来可能会扩展新的压缩方式。压缩方式的相关定义如下：

```
/* in src/include/access/toast_compression.h */
typedef enum ToastCompressionId {
    TOAST_PGLZ_COMPRESSION_ID = 0,
    TOAST_LZ4_COMPRESSION_ID = 1,
    TOAST_INVALID_COMPRESSION_ID = 2
} ToastCompressionId;
```

参数 default_toast_compression 规定了缺省的压缩方式，缺省值是 pglz。如果你想支持 lz4 压缩方式，需要在编译 PostgreSQL 源码的时候指定--with-lz4 选项：

```
$ pwd
/home/postgres/postgresql-16.0
$ ./configure --help | grep lz4
--with-lz4           build with LZ4 support
```

下面我们往原表中插入一条超长的记录，观察 TOAST 表中的数据。

```
oracle=# insert into blog values(1, 'This is TITLE column', repeat('X', 320000));
INSERT 0 1
oracle=# select id, title, length(content) from blog;
 id |      title       | length
    |                 |
    |                 | 320000
```

```

-----+-----+
 1 | This is TITLE column | 320000
(1 row)

oracle=# select chunk_id, chunk_seq, length(chunk_data) from pg_toast.pg_toast_16462 order by 1,2;
chunk_id | chunk_seq | length
-----+-----+
 16467 |          0 |    1996
 16467 |          1 |   1675
(2 rows)

```

我们插入的第一条记录的 content 列的原始长度为 320000 个字节，远远超过 8192 字节，所以这一列会被压缩，分割，保存到对应的 TOAST 表中，分为 2 个数据条，总长度仅为 3671(=1996 + 1675) 字节，压缩比为 1.15%(=3671/32000)，非常小了。那么我们看看原表中的记录是什么样子。

```

oracle=# SELECT * FROM page_header(get_raw_page('blog',0));
 lsn      | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 0/1992B88 |        0 |     0 |    28 |  8120 |    8192 |    8192 |       4 |        0
(1 row)

```

我们看到在原表中，这条记录只有 72 个字节 (=8192 - 8120)，我们看一下这条记录的原始内容：

```

oracle=# SELECT pg_relation_filepath('blog');
pg_relation_filepath
-----
base/16384/16462
(1 row)

oracle=# \! ls -l data16/base/16384/16462
-rw----- 1 postgres postgres 8192 Dec  2 21:50 data16/base/16384/16462
oracle=# checkpoint; /* 手工执行一个检查点，确保数据被写入到磁盘中 */
CHECKPOINT
oracle=# \! hexdump -C data16/base/16384/16462
00000000  00 00 00 00 88 2B 99 01  00 00 00 00 1C 00 B8 1F  |.....+.....|
00000010  00 20 04 20 00 00 00 00  B8 9F 86 00 00 00 00 00  |. . . . . . . . |
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |. . . . . . . . |
*
00001fb0  00 00 00 00 00 00 00 00  07 03 00 00 00 00 00 00  |. . . . . . . . |
00001fc0  00 00 00 00 00 00 00 00  01 00 03 00 06 09 18 00  |. . . . . . . . |
00001fd0  01 00 00 00 2B 54 68 69  73 20 69 73 20 54 49 54  |. . . +This is TIT|
00001fe0  4C 45 20 63 6F 6C 75 6D  6E 01 12 04 E2 04 00 57  |LE column.....W|
00001ff0  0E 00 00 53 40 00 00 51  40 00 00 00 00 00 00 00  |...S@..Q@.....|
00002000      ^~~~      ^~ ^~
```

我们看到，title 这一列的值是被保存到了原数据块中，而 content 这一列的值并没有保存到原数据块中，而是由一个 TOAST 指针指向相关的 TOAST 表中相关的记录。因为该测试表对应的 TOAST 表的 Oid 是 16465，十六进制是 0x4051，TOAST 表中的 chunk_id 是 16467，十六进制是 0x4053，我们从上面的原始数据中可以找到这两个数值，表明指向 TOAST 表的指针记录了 TOAST 表的 Oid 和对应的 chunk_id。通过这两个关键信息，就可以迅速定位被压缩和分割的 content 列的数据。

我们知道，四种存储方式中，EXTERNAL 和 EXTENDED 的区别是：EXTERNAL 只分割，不压缩，现在我们测试一下 EXTERNAL 的存储方式。我们执行如下命令：

```
oracle=# alter table blog alter title set storage external;
ALTER TABLE
oracle=# \d+ blog
      Table "public.blog"
 Column | Type          | Storage | Compression | Stats target | Description
-----+-----+-----+-----+-----+
 id   | integer       | plain   |           |           |
 tag  | character(3) | extended |           |           |
 title | character varying | external |           |           |
 content | text         | extended |           |           |
Access method: heap
```

我们看到，title 这一列的存储方式已经由 EXTENDED 变成了 EXTERNAL。下面我们插入第二条记录，在 title 这一列中存入 3000 个字符。

```
oracle=# insert into blog values(2, repeat('X', 3000), 'This is content');
INSERT 0 1
oracle=# select chunk_id, chunk_seq, length(chunk_data) from pg_toast.pg_toast_16462 order by 1,2;
chunk_id | chunk_seq | length
-----+-----+-----+
 16467 | 0 | 1996
 16467 | 1 | 1675
 16468 | 0 | 1996
 16468 | 1 | 1004
(4 rows)
```

我们看到，新增加的 chunk_id=16468 就是第二条记录的 title 这一列的值。它分为两个数据条，总长度为 $1996 + 1004 = 3000$ 字节。因为原始记录中 title 这一列的值的原始长度是 3000 字节，说明这一列只进行了分割，并没有进行压缩。这是 EXTERNAL 和 EXTENDED 的区别。按照类似的实验思路，你可以自己动手，实验一下 MAIN 存储方式的特性。这里就不再赘述了。

<https://cloud.tencent.com/developer/article/1004455>

实际上，对于可变长度的数据类型，PostgreSQL 采用了一种比较简单的数据模型来表示它们，其具体定义如下：

```
/* src/include/c.h */
#define FLEXIBLE_ARRAY_MEMBER /* empty */
struct varlena {
    char vl_len_[4];          /* Do not touch this field directly! */
    char vl_dat[FLEXIBLE_ARRAY_MEMBER]; /* Data content is here */
};
```

由源代码中的数据结构定义可知：开始的 4 个字节表示数据长度，随后的是真正的数据。在这 4 个字节的长度信息中，PostgreSQL 征用了 2 个比特用作特殊用途，所以真正能够表示数据长度的只有 30 比特，即 TOAST 数据的最大长度是 1GB 字节。

6.1.2 堆表的空闲空间管理

当我们想往一张堆表里插入一条记录时，因为堆表的数据是无序的，所以只要找到第一个空闲空间足够大的数据块即可。如何快速地在堆表中快速寻找一个具有足够空闲空间的数据块呢？假设一张表有 8GB 大小，这

个体积在实际中很常见，则该表就有 $8\text{GB}/8\text{KB} = 1\text{M}$ ，即 100 多万个数据块。如何在这数量庞大的数据块中快速寻找一个由足够空闲空间的数据块显然是一个难题。空间空间映射文件 fsm 就是为了加速寻找具有足够空闲空间的数据块而设计的辅助文件。fsm 是 free space map 的缩写。在 fsm 文件中用一个字节表示堆表中一个数据块的空闲空间的大小。因为一个字节的取值范围是 0 到 255，所以该字节的值和数据块空闲空间的关系可以用图 6.2 表示：

值	代表的空闲空间
0	0 ~ 31字节
1	32 ~ 63字节
2	63 ~ 95字节
3	96 ~ 127字节
.....
254	8128 ~ 8159字节
255	8160 ~ 8191字节

图 6.3: fsm 文件中一个字节表示的空闲空间大小

由于数据块的编号实际上是四字节的无符号整数，一张表的数据块的最大数量是 4G，所以 fsm 文件最大的体积也是 4G。我们可以把 fsm 文件中表示数据块空闲空间的这些字节理解为一个一维数组，每一个字节的值描述对应堆表中一个数据块的空间空间。很显然，在一个巨大的一维数据进行查找的效率是低效的，所以在 fsm 文件内部，实际上是一个二叉树的结构，二叉树的叶子节点是描述空闲空间信息的字节。所以 fsm 文件的最大体积比 4G 要大一些，因为它的叶子节点就有 4GB 字节，还有额外的非叶子节点。图 6.3 展示了 fsm 文件和堆表数据文件的关系：

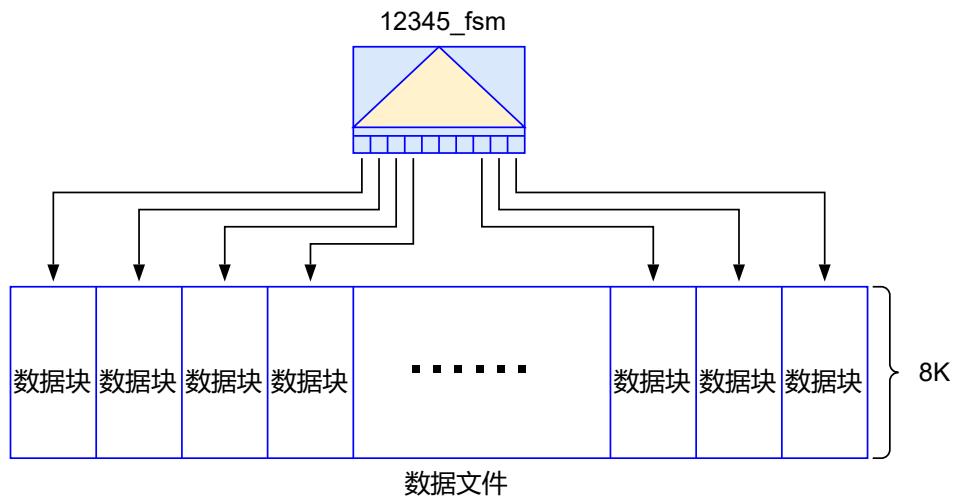


图 6.4: fsm 文件和堆表数据文件的关系

对数据结构有初步了解的读者都明白，二叉树是父节点只有左右两个子节点的数据结构。如果底层的叶子节点表示每一个数据块的空闲空间的大小，父节点中存储的是它两个子节点中的最大值，那么在 fsm 文件内部就形成了如图 6.4 所示的结构：

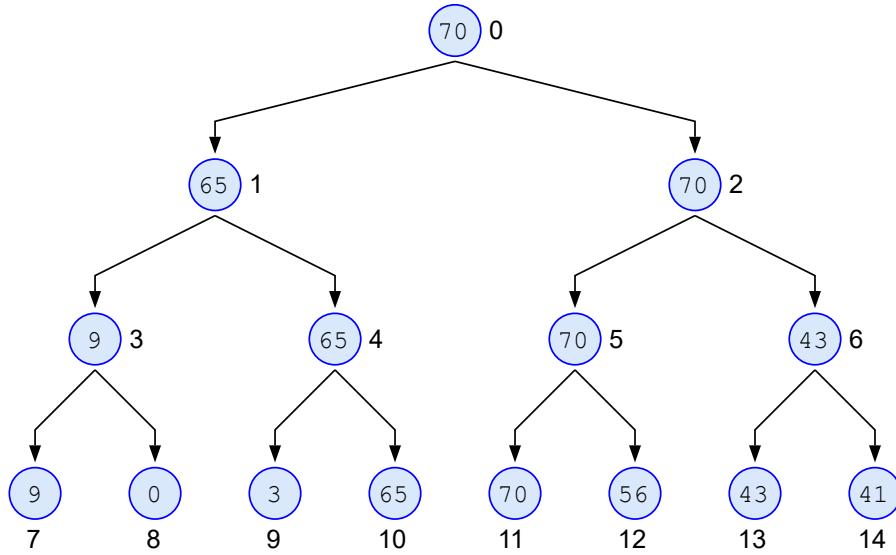


图 6.5: fsm 文件的结构

在图 6.4 中，每一个圆代表一个字节，圆内的数字代表其存储的值，圆外的数字表示其编号，根节点的编号是 0，然后依次编号。我们很容易发现父子节点的编号规律：左节点的编号是父节点的编号的两倍加一，右节点的编号比左节点的编号多个一。我们用如下代码来表示：

```

/* src/backend/storage/freespace/fsmpage.c */
#define leftchild(x)      (2 * (x) + 1)
#define rightchild(x)     (2 * (x) + 2)
#define parentof(x)       (((x) - 1) / 2)
static int rightneighbor(int x)
{
    x++;
    if (((x + 1) & x) == 0)  x = parentof(x);
    return x;
}
  
```

其中寻找右边邻居的代码 `rightneighbor()` 在处理最右边节点的时候，回绕到了同一层的最左边。譬如 3 号节点的右边邻居是 4 号节点，而 6 号节点的右边邻居是 3 号节点。因为最左边的节点编号是 3,7,15 等，即 $2^n - 1$ 的形式，其二进制的表示的各位全部都是 1，所以很容易通过 $((x + 1) \& x) == 0$ 判断出来。知道了这个规律，上面的代码非常容易理解。fsm 文件也是划分成 8KB 固定大小的数据块，在一个数据块中保存的二叉树的形式如图 6.5 所示：

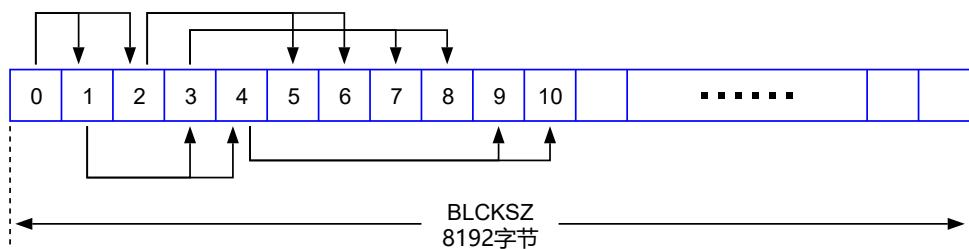


图 6.6: fsm 数据块中各字节的关系

在 fsm 文件的一个数据块中，一共可以存放 $1 + 2 + 4 + 8 + 16 + \dots + 4096 = 8191$ 个字节，还空闲了一

个字节。其中叶子节点的数目是 4096，非叶子节点的数目是 4095，等于 $(BLCKSZ / 2 - 1)$ ，所以有一个常量 NonLeafNodesPerPage 来表示一个数据块中二叉树的非叶子节点的数量：

```
/* in src/include/pg_config.h */
#define BLCKSZ 8192
/* in src/include/storage/fsm_internals.h */
#define NonLeafNodesPerPage (BLCKSZ / 2 - 1)
```

为了管理 Block，PostgreSQL 在 Block 的头部增加了页头 PageHeaderData，共 24 个字节，还有一个偏移量变量 fp_next_slot，如下图所示：

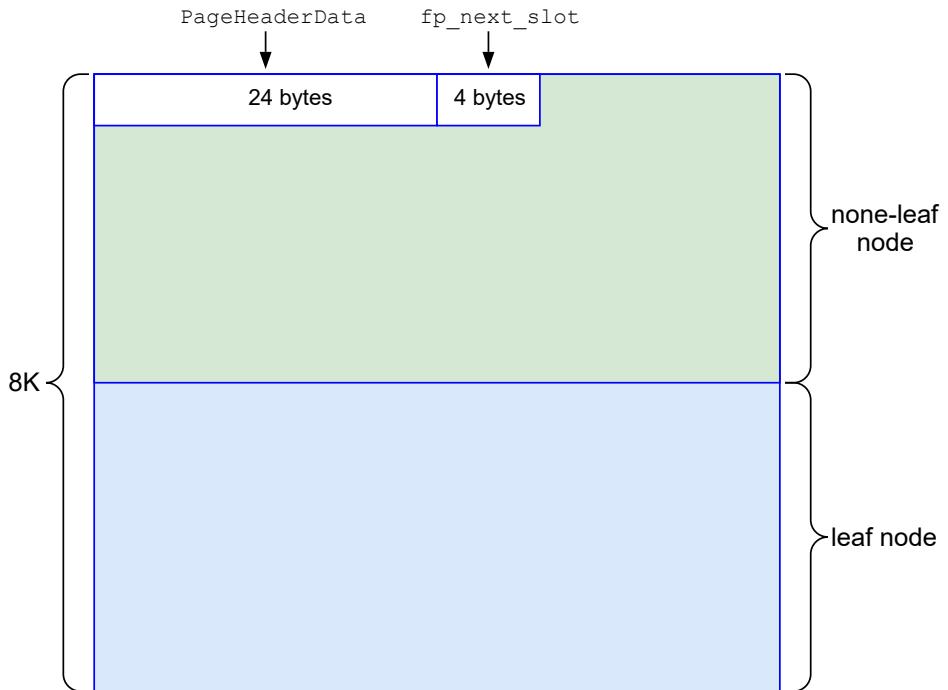


图 6.7: XXXXX

相关的数据结构的定义如下：

```
/* in src/include/storage/fsm_internals.h */
typedef struct {
    int          fp_next_slot;
    uint8       fp_nodes[FLEXIBLE_ARRAY_MEMBER];
} FSMPagedata;

typedef FSMPagedata *FSMPage;

#define offsetof(type, field)      ((long) &((type *)0)->field)
#define SizeOfPageHeaderData (offsetof(PageHeaderData, pd_linp))

#define NonLeafNodesPerPage (BLCKSZ / 2 - 1)

#define NodesPerPage (BLCKSZ - MAXALIGN(SizeOfPageHeaderData) - \
                  offsetof(FSMPagedata, fp_nodes))
```

```
#define LeafNodesPerPage (NodesPerPage - NonLeafNodesPerPage)
```

由于多了 28 个字节的控制信息，所以真正的 fsm 文件的 Block 的布局如下：

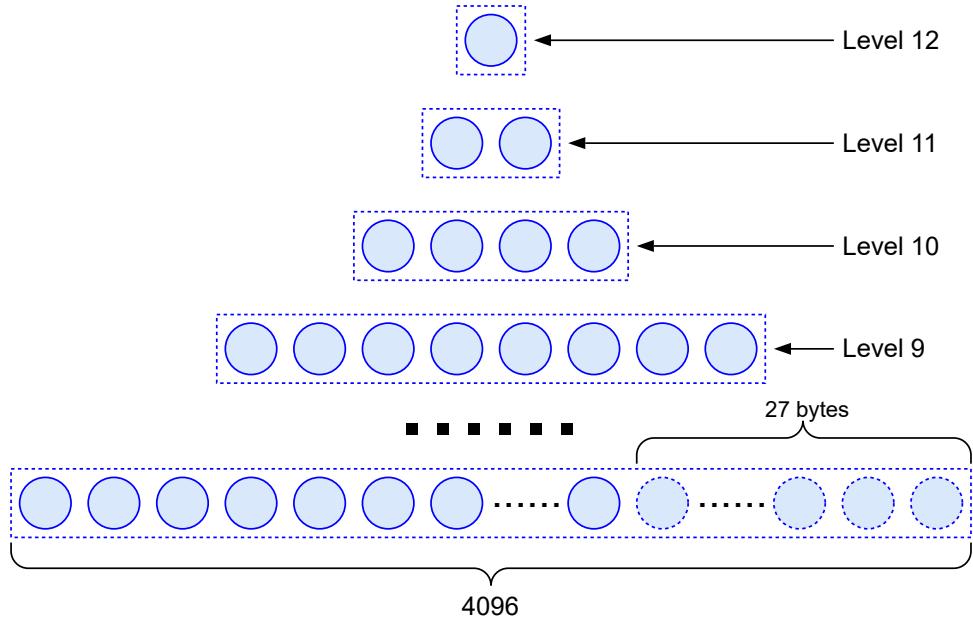


图 6.8: XXXXX

其中非叶子节点依然是 4095 个，但是叶子节点共计 $4096 - 28 + 1 = 4069$ 个。所以一个 fsm 文件的 Block 可以描述数据文件中 $4069 * 8K$ 约为 32M 数据的空闲空间的信息。NodesPerPage 的值是 $8192 - 28 = 8164$ ，LeafNodesPerPage 的值是 $8164 - 4095 = 4069$ 。fp_next_slot 指向本 Block 中的叶子节点，fp_next_slot = 0 则表示指向了第一个叶子节点。我觉得 fp_next_slot 还可以进一步优化，完全用不到 4 个字节，用 2 个字节足够了，这样又可以省出两个字节给叶子节点，一个 Block 中的叶子节点数量是 4071。

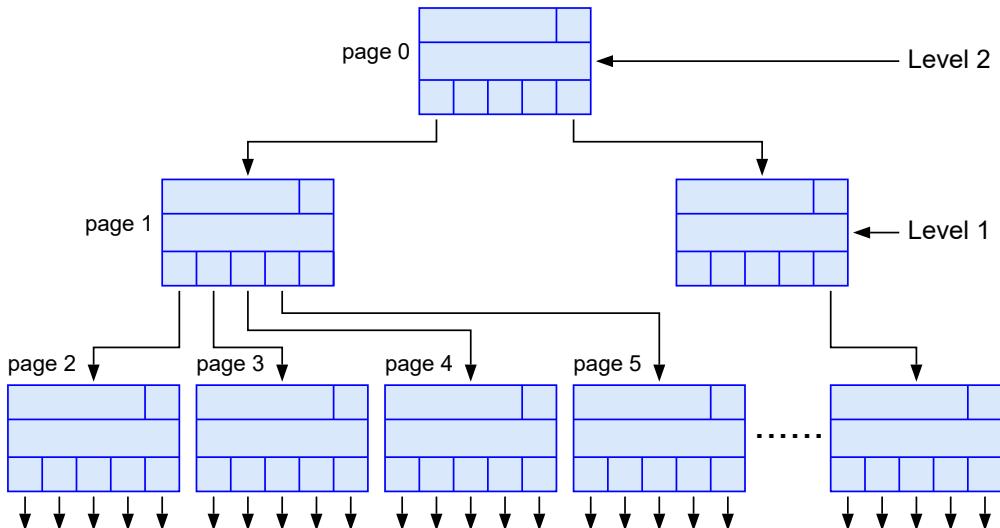


图 6.9: XXXXX

6.1.3 分区表

随着每天业务的不断运营，数据库中的数据不断增长，是生产环境下所有数据库的一个基本规律。随着表的体积不断变大，对其的各种操作，包括查询数据，都会变得越来越慢。对表进行分区（partitioning）是解决表不断增大的有力手段。它的基本思想很容易理解，就是典型的分而治之的策略。我们可以把一张表想象成一块大的排骨，然后把它分成很多小的排骨，这些小排骨连接在一起，由可以看做一块大的排骨。图 6.3 展示了分区表的基本概念：

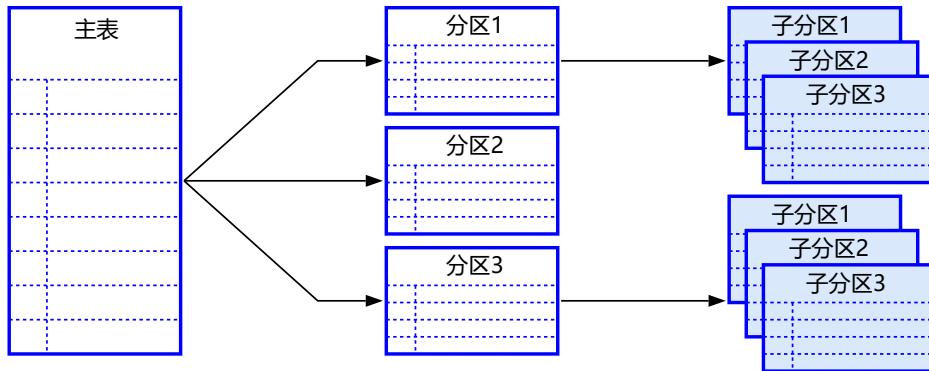


图 6.10: 分区表基本结构

由上图可知，一个大表被分成了结构相同的一系列小表，这些小表被称为“分区”（partition），拥有分区的表，即图中的大表，被称为分区表（partitioned table）。PostgreSQL 和 Oracle 一样，支持最多两级分区，即一个分区表的分区可以进一步被划分成二级分区，这些二级分区被称为 sub-partition。下面我们通过实验来创建一个分区表，让我们有一个感性的认识：

```
oracle=# create table sales_region(id int, amount int, region char(2)) partition by list(region);
CREATE TABLE
oracle=# create table tx partition of sales_region for values in ('TX');
CREATE TABLE
oracle=# create table ma partition of sales_region for values in ('MA');
CREATE TABLE
oracle=# create table co partition of sales_region for values in ('CO');
CREATE TABLE

oracle=# insert into sales_region values(1, 1, 'TX');
INSERT 0 1
oracle=# insert into sales_region values(2, 1, 'MA');
INSERT 0 1
oracle=# insert into sales_region values(3, 1, 'CO');
INSERT 0 1
oracle=# select * from sales_region;
 id | amount | region
----+-----+-----
 3 |      1 | CO
 2 |      1 | MA
 1 |      1 | TX
(3 rows)
oracle=# select * from tx;
```

```
id | amount | region
---+-----+-----
 1 |      1 | TX
(1 row)
oracle=# select * from ma;
id | amount | region
---+-----+-----
 2 |      1 | MA
(1 row)
oracle=# select * from co;
id | amount | region
---+-----+-----
 3 |      1 | CO
(1 row)
```

分区表毫无疑问会带来很多好处，譬如一个分区表按照日期来分区，每天有一个分区表。如果某一个查询的 WHERE 条件中有 partitioned

6.2 B 树索引

6.2.1 索引的基本概念

索引 (index) 是用来帮助提高数据库的查询速度的一种辅助手段。我们知道堆表中的记录是无序存放的，而索引的基本思想是：在一个有序的空间中进行查找的速度远远大于在一个无序的空间中查找的速度。这个规律是不言自明的。既然存在这个规律，那么我们就可以提前把表的一部分数据（一列或者几列）提前排好序，单独保存在一个地方。如果你的查询条件中有这些有序的数据，查询的速度会大大加快。这就是索引的最朴素的思想。所以虽然堆表是无序的，但是索引一定是有序的。

索引只是解决了查询快慢的问题，并没有解决数据有无的问题，所以它可有可无，并不是数据库不可或缺的东西。但数据库一些重要的功能，如主键 (primary key)，唯一性约束 (unique constraint) 都需要索引在底层的支撑，另外数据库的性能问题可能会导致数据库的可用与否，所以在实践中，索引是数据库中极其重要的组件，索引类型的丰富程度是衡量一个数据库软件功能的重要指标。PostgreSQL 提供了非常丰富的索引类型：B 树索引、BRIN 索引、哈希索引、GIN 索引、布隆索引、GiST 索引和 SP-GiST 索引，同时也提供了开放接口可以供未来的新型索引的引入。在这些索引中，应用最广泛的所以是 B 树索引，所以本节只介绍 B 树索引，通过本节的学习，我们会建立表和索引的关系的基本概念，方便后续内容的学习。其它索引类型会单独一章在本书的后面做专题讲述。

下面我们创建一个简单的索引：

```
oracle=# create table idxdemo(id int, name varchar(16));
CREATE TABLE
oracle=# create index idx1 on idxdemo(id);
CREATE INDEX
oracle=# \d idxdemo
      Table "public.idxdemo"
 Column |          Type          | Collation | Nullable | Default
-----+----------------+-----+-----+-----+
 id    | integer        |          |          |
 name  | character varying(16) |          |          |
Indexes:
"idx1" btree (id) /* <== 这里列出了这张表中的所有索引，目前只有一个，类型是btree */
```

上述实验创建了一张表 idxdemo，然后在它的 id 这一列上创建了一个索引 idx1。如果不指定索引的类型，则缺省的索引类型是 B 树类型。索引中的有序数据往往组成树形结构，以利于加速查找的过程，所以我们往往用一个三角形表示索引。图 XX 展示了堆表和索引的关系：

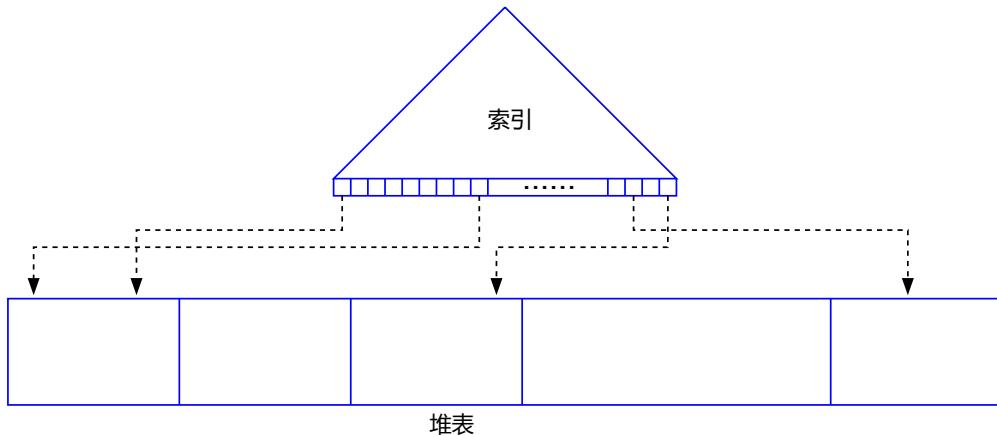


图 6.11: 堆表和索引的关系

索引是独立于堆表而独立存储的数据文件。譬如一个堆表的数据文件叫做 12345，它的一个索引的数据文件可能就是 54321。因为索引也需要占用磁盘空间，所以如果我们想知道一张表有多大，有两个不同的概念，一个是表本身的体积，一个是表加上其上的索引的总体积。在上图中的三角形就表示一个索引，它的叶子节点(关于叶子节点的概念下面会讲述)包含一个个指针，每一个指针指向了堆表的一条记录，两者是一对一的关系，即如果表中有 n 条记录，则索引中就有 n 个指针分别指向它们。

因为我们并没有学习 SQL 的查询优化的知识，所以本节只考察 B 树索引的存储结构。索引的数据文件也是按照缺省 8KB 的大小来划分的一个个数据块的，索引的数据块的格式就是第二章我们介绍的数据块的格式。不同索引的数据块的格式根据索引的类型不同而稍微有所不同，但是都有一个通用的模式，如图 XX 所示。

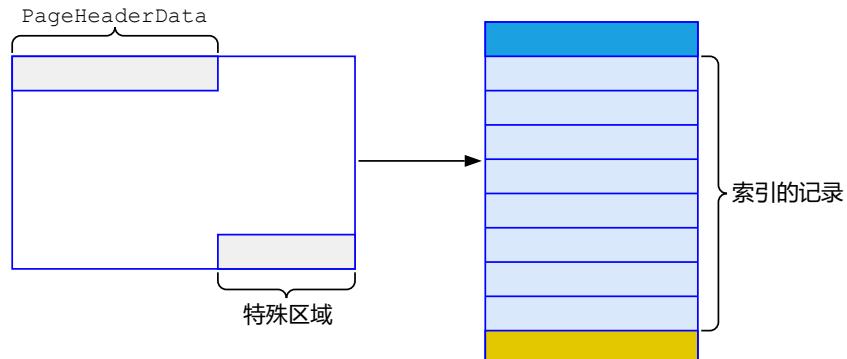


图 6.12: 索引页的通用结构

索引的数据块的结构分为三部分，页头是 PageHeaderData 结构，共计 24 个字节，页尾是特殊区域，页头和页尾之间的部分就是索引的一条条记录，当然也可能包括一些空闲区域。我们从第二章的分析中知道，堆表的数据块是没有特殊区域的，特殊区域是各种索引经常使用的空间，由 PageHeaderData 结构中的 pd_special 指针来指定其起始位置，其大小往往是 8 字节，或者 16 字节。在后面的讨论中，有时候只强调索引的逻辑结构，不考虑其各部分的尺寸，我们也可以把一个索引页面画成图 XX 右边的样子，请读者留心一下。

在索引中一个重要的数据结构是 TID(tuple identifier)，它是一个指向堆表中某一条记录的指针，具体结构如下：

```
/* in src/include/storage/block.h */
typedef struct BlockIdData {
    uint16          bi_hi;
    uint16          bi_lo;
```

```

} BlockIdData;
/* in src/include/storage/itemptr.h */
typedef uint16 OffsetNumber;
typedef struct ItemPointerData {
    BlockIdData ip_blkid;
    OffsetNumber ip_posid;
}

```

由上可知，TID 的数据结构 ItemPointerData 由 6 个字节组成，其中成员变量 ip_blkid 的长度是 4 字节，它表示堆表的数据块的编号。成员变量 ip_posid 的长度是 2 字节，它表示索引的 TID 指向的是该数据块中的第几条记录，也就是结构体 PageHeaderData 后面的 pd_linp 数组的下标，而不是这条记录在该数据块中的偏移量。这种间接寻址的设计带来了更大的灵活性，请读者稍加留意其中的含义。注意：ip_posid 的编号是从 1 开始，而不是通常的 0。例如 (2,1) 表示指向堆表中块号为 2 的数据块中的第一条记录。

6.2.2 B 树索引

B 树 (B-Tree) 索引是使用领域最广的索引。树是计算机科学领域重要的数据结构，初学者很容易理解，因为在大自然中有各种各样的树，每一个人都身处在一个家族树当中。图 XX 展示了计算机科学领域中的一棵树：

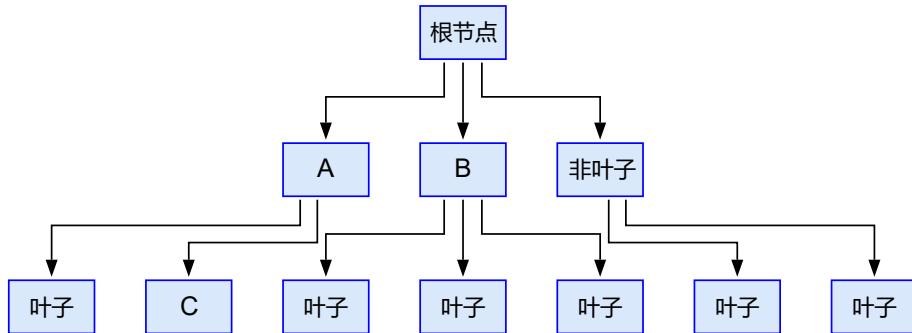


图 6.13: 计算机科学中树的基本概念

我们可以看到，树由一些节点 (node) 和它们之间的连线组成。树有一些重要的概念，其定义如下：

- **根节点 (root node)**：就是没有任何父母的节点。每一棵树有且只有一个根节点。我们经常把根节点画在最上层。
- **叶子节点 (leaf node)**：就是没有任何子孙后代的节点。我们经常把叶子节点画在最下层。
- **非叶子节点 (non-leaf node)**：在树中至少有一个子孙后代的节点被称为非叶子节点。树中的节点分为两种类型：叶子节点和非叶子节点。
- **路径长度 (path length)**：我们把从树中某一节点 A 到节点 B 的路径长度定义为此路径中所经过的节点总个数 (包含它们自身)。例如在上面的树中，根节点到 A 节点的路径长度是 2，根节点到 C 节点的路径长度是 3。A 节点到 C 节点的路径长度是 2。A 节点到 B 节点则不可到达，因为箭头是单向的，A 节点无法上溯到它的父节点。

树有两种典型的类型：平衡树 (balance tree) 和二叉树 (binary tree)。初学者很容易把这两个概念搞混淆，因为两者都是以 B 开头。Binary 的含义是“两个”，所以二叉树的核心特征是：在二叉树中，每个父节点最多只能有两个子节点。平衡树的定义是：从根节点到任何一个叶子节点的路径长度都是一样的，则此棵树被称为平衡树。Balance 是“平衡”的意思，它强调的是从根节点到叶子节点的路径是一样的，即平衡的。平衡树并不限制一个父节点有多少个子节点，往往一个父节点有几百个子节点，这和二叉树根本不同。图 XX 中的树中从根节

点到任何一个叶子节点的路径长度都是 3，所以它是一棵平衡树。直观上，一棵树可以在水平方向上被分为一层层 (level) 的。如上面的平衡树共分为 3 层，叶子节点所在的层的层号是 0，往上走，层号不断增加，根节点的层号是 2。

B 树索引的内部存储结构就是平衡树，它的每一个节点是 8KB 大小的数据块。为了方便同一层中的节点互相访问，会在同一层增加了双向链表，如下图所示：

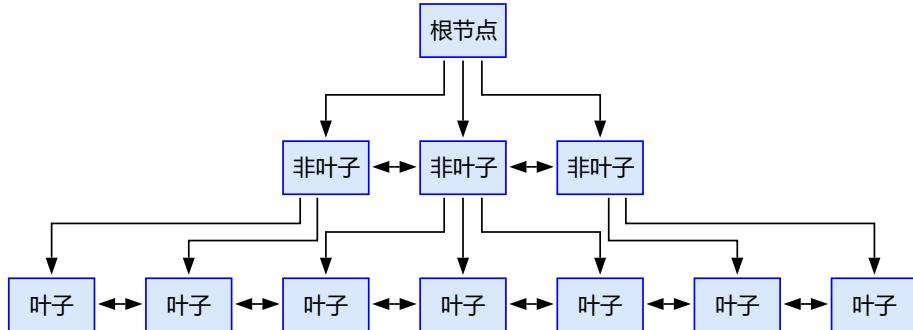


图 6.14：带有双向链表的 B 树索引

有了同一层的双向链表，且同一层的数据是排序的情况下，当我们到达某一层后，可以很方便地进行水平遍历。譬如一个查询要求某一列 $m \geq 3$ AND $m \leq 17$ ，当我们找到了 3，只要水平往右，达到 17 就可以停止搜索了，因为再往右的值比 17 大，无需搜索了。

B 树索引的数据文件也是由一个个 8KB 大小的数据块组成，其中 0 号数据块，就是第一个数据块被称为元数据块。我们可以用图 XX 表示索引的数据文件的基本结构：

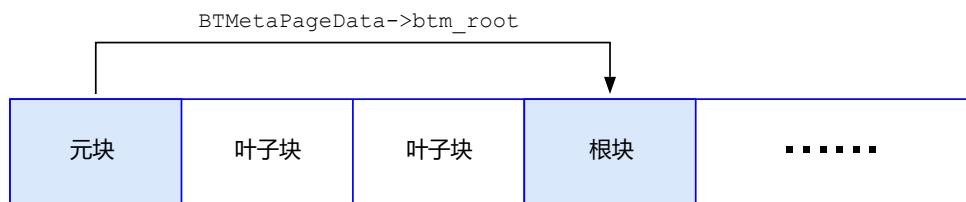


图 6.15：B-树索引的数据块类型

其中第一个数据块是元数据块，它记录了该索引的一些基本信息。当我们访问一个索引的时候，首当其中的任务就是要访问根节点，然后顺藤摸瓜，依次访问更低层的非叶子节点和最终的叶子节点。所以元数据块中有一个指针 btm_root 指向了根节点的数据块。元数据块的数据结构定义如下：

```

/* in src/include/access/nbtree.h */
#define BTREE_MAGIC      0x053162      /* magic number in metapage */
typedef uint32 BlockNumber;
typedef struct BTMetaPageData {
    uint32          btm_magic;        /* should contain BTREE_MAGIC */
    uint32          btm_version;      /* nbtree version (always <= BTREE_VERSION) */
    BlockNumber     btm_root;         /* current root location */
    uint32          btm_level;        /* tree level of the root page */
    BlockNumber     btm_fastroot;     /* current "fast" root location */
    uint32          btm_fastlevel;    /* tree level of the "fast" root page */
    uint32          btm_last_cleanup_num_delpages;
    float8         btm_last_cleanup_num_heap_tuples;
}
  
```

```

    bool          btm_allequalimage; /* are all columns "equalimage"? */
} BTMetaPageData;

```

下面我们通过一个实验查看一下一个索引的元数据块。首先创建一个测试表和它上面的索引，具体过程如下：

```

oracle=# CREATE TABLE state(id INT PRIMARY KEY, name VARCHAR(64));
CREATE TABLE
oracle=# \d state
      Table "public.state"
 Column | Type           | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 id    | integer        |           | not null |
 name  | character varying(64) |           |           |
Indexes:
"state_pkey" PRIMARY KEY, btree (id)

```

在上面的测试表中，我们规定第一列 id 为主键。其实在底层，PostgreSQL 就悄悄地帮我们创建了一个 B 树索引，它的名字是 state_pkey。我们查看测试表和对应索引的文件名是什么，执行如下操作：

```

oracle=# SELECT pg_relation_filepath('state');
pg_relation_filepath
-----
base/16384/16485
(1 row)

oracle=# SELECT pg_relation_filepath('state_pkey');
pg_relation_filepath
-----
base/16384/16488
(1 row)

oracle=# INSERT INTO state VALUES(0, 'Georgia'),(1, 'Delaware'); /* 插入两条测试数据*/
INSERT 0 2
oracle=# CHECKPOINT; /* 手工执行CheckPoint，确保所有的数据落盘 */
CHECKPOINT
postgres=# \! ls -l data16/base/16384/16485          /* 查看表的体积，只有8192字节 */
-rw----- 1 postgres postgres 8192 Dec  3 00:48 data16/base/16384/16485
postgres=# \! ls -l data16/base/16384/16488          /* 查看索引的体积，有16384个字节 */
-rw----- 1 postgres postgres 16384 Dec  3 00:48 data16/base/16384/16488

```

我们只插入了两条记录，这两条记录很短，显然只需要一个数据块就可以保存，所以堆表本身只有 8192 个字节，这是很容易理解的。但索引的体积却有 16384 个字节，即 2 个数据块 ($16384 = 2 \times 8192$)。我们可以由图 XX 可知，第一个数据块是元数据块，第二个数据块是根节点，也是叶子节点。下面我们用 hexdump 来检查第一个数据块的内容：

```

postgres=# \! hexdump -C data16/base/16384/16488 -n 8192
00000000  00 00 00 00 C8 AA 9E 01  00 00 00 00 48 00 F0 1F  |.....H...|
00000010  F0 1F 04 20 00 00 00 00  62 31 05 00 04 00 00 00  |....b1....|
00000020  01 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |.....|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 F0 BF  |.....|
00000040  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|

```

```

*
00001ff0 00 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00 |.....|
00002000

```

我们可以看到，跳过头 24 个字节的页头，第 25 个字节开始的 4 个字节是 0x053162，即 BTREE_MAGIC。紧随其后的 4 个字节是 B 树索引的版本号，为 4，紧随其后的 4 个字节是 1，表示根节点的数据块编号是 1，再紧随其后的 4 个字节是 0，表示根节点的层数是 0，即根节点也是叶子节点。如果我们安装了 pageinspect 插件，我们可以执行如下命令：

```

oracle=# select magic, version, root, level from bt_metap('state_pkey');
magic | version | root | level
-----+-----+-----+
340322 |      4 |     1 |     0
(1 row)

```

上述查询得到的结果和用 hexdump 得到的结果是一样的。

除了第一个数据块以外，其余的数据块都是索引的保存索引数据的普通块。这些普通块分为非叶子节点和叶子节点，它们都包含了一个特殊区域。现在我们查看一下索引 state_pkey 的第二个数据块的信息：

```

oracle=# SELECT * FROM page_header(get_raw_page('state_pkey', 1));
lsn    | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+
0/19EAB90 |        0 |      0 |     32 |   8144 |    8176 |     8192 |       4 |        0
(1 row)

```

有实验结果可知，这个数据块包含了一个特殊区域，共 16 个字节 (=8192 - 8176)。这个特殊区域的数据定义如下：

```

/* in src/include/access/nbtree.h */
typedef uint32 BlockNumber;
typedef uint16 BTCycleId;
typedef struct BTPageOpaqueData {
    BlockNumber btpo_prev; /* left sibling, or P_NONE if leftmost */
    BlockNumber btpo_next; /* right sibling, or P_NONE if rightmost */
    uint32      btpo_level; /* tree level --- zero for leaf pages */
    uint16      btpo_flags; /* flag bits, see below */
    BTCycleId   btpo_cycleid; /* vacuum cycle ID of latest split */
} BTPageOpaqueData;

```

在上面的数据结构中，包含了两个域：btpo_prev 和 btpo_next。这是两个指针，分别指向和本数据块同一层的相邻的前导数据块和后面的数据块。所以依靠着两个指针，同一层的数据块形成了双向链表结构，如图 XX 所示。

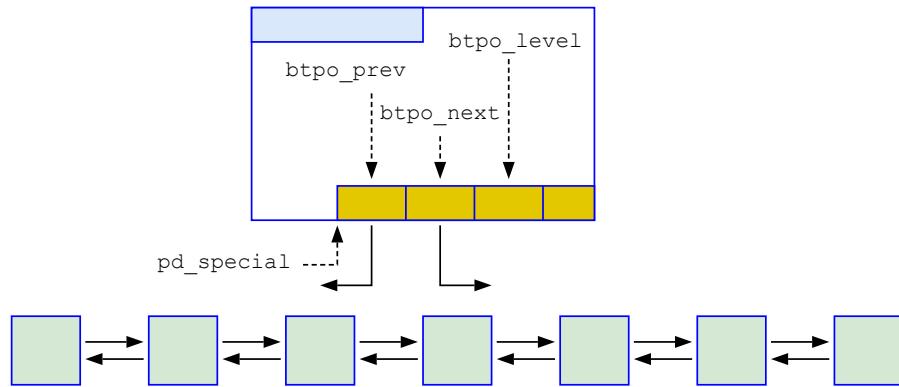


图 6.16: 索引数据块中特殊区域保存的指针形成双向链表结构

我们可以查询一下编号为 1 的数据块的信息:

```
oracle=# SELECT * FROM bt_page_stats('state_pkey',1);
-[ RECORD 1 ]-----
blkno      | 1      /* 本数据库的编号 */
type       | 1
live_items | 2      /* 本数据块里有2条记录 */
dead_items | 0
avg_item_size | 16
page_size   | 8192
free_size   | 8108
btvo_prev   | 0      /* 本数据块的前导数据块, 0表示空, 因为0号数据块是元数据块, 不是普通块 */
btvo_next   | 0      /* 本数据块的后面的数据块, 0表示空 */
btvo_level  | 0      /* 本数据块的层数为0, 即本块为叶子节点 */
btvo_flags  | 3
```

我们可以使用 `bt_page_items` 函数查看某一个索引数据块的记录的内容, 执行结果如下:

```
oracle=# select * from state;
 id | name
----+-----
 0 | Georgia
 1 | Delaware
(2 rows)

oracle=# SELECT * FROM bt_page_items('state_pkey',1);
 itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
    1 | (0,1) |     16 | f     | f     | 00 00 00 00 00 00 00 00 | f     | (0,1) |
    2 | (0,2) |     16 | f     | f     | 01 00 00 00 00 00 00 00 | f     | (0,2) |
(2 rows)
```

由此我们可以看出, 堆表中有两条记录, 对应的索引里面也有两条记录, `id=0` 的记录的 `ctid=(0,1)`, 指向了堆表的 0 号数据块的第一条记录。`id=1` 的记录的 `ctid=(0,2)`, 指向了堆表的 0 号数据块的第 2 条记录。现在我们运行一个脚本, 往表中插入一批记录:

```
oracle=# \! cat /tmp/ins.sql
```

```

DO $$

BEGIN
    FOR i IN 2..3000 LOOP
        INSERT INTO state VALUES(i, 'Connecticut');
    END LOOP;
END; $$

oracle=# \i /tmp/ins.sql
DO
oracle=# checkpoint;
CHECKPOINT
oracle=# SELECT count(1) FROM state;
count
-----
3001
(1 row)

oracle=# SELECT magic, version, root, level FROM bt_metap('state_pkey');
magic | version | root | level
-----+-----+-----+-----+
340322 |      4 |     3 |      1
(1 row)

```

我们可以看到，根节点已经从 1 号数据块变成了 3 号，它的层数是 1，由此我们可以推知 1 号和 2 号数据块都是叶子节点。我们查看一下根节点的一些统计信息：

```

oracle=# \! ls -l data16/base/16384/16488
-rw----- 1 postgres postgres 90112 Dec  3 02:24 data16/base/16384/16488
oracle=# SELECT * FROM bt_page_stats('state_pkey',3);
-[ RECORD 1 ]-----
blkno      | 3
type       | r
live_items | 9
dead_items | 0
avg_item_size | 15
page_size  | 8192
free_size   | 7976
btpp_prev   | 0
btpp_next   | 0
btpp_level  | 1
btpp_flags  | 2

```

我们看到了，根节点中有 9 条记录，分别指向了 1,2 号数据块和 4,5,6,7,8,9,10 号数据块。索引的体积是 90112 字节，即 11 个数据块，即 9 个叶子节点的数据块加上一个根节点的数据块，再加上一个元数据块。图 XX 展示了随着不断往表里插入记录时，索引的数据块的变化情况。

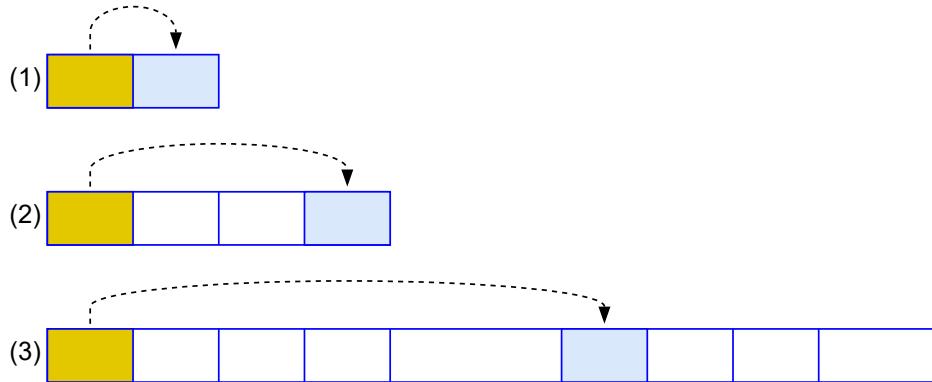


图 6.17: 索引的数据块随着数据插入的变化情况

一开始表中记录数很少的时候，只需要一个数据块，即数据块 1，既做根节点，也做叶子节点。随着数据不断插入表中，索引数据块中的记录也不断增加，因为索引的记录数和表中的记录数是一样多的。当数据块 1 插满后，就诞生数据块 2 作为新的叶子节点，同时产生数据块 3 作为新的根节点指向数据块 1 和 2。注意，此时根节点中的记录条数为 2，因为根节点只要指向数据块 1 和 2 即可。随着新数据的插入，又陆续产生 4,5,6,7,8 等数据块作为叶子节点，但是根节点的记录数依然很少，所以此时索引只有 2 层。当根节点的数据块被插满后，就会再产生一个非叶子节点，根节点的层数变为了 3。依次类推。

```
oracle=# SELECT * FROM bt_page_items('state_pkey',3);
itemoffset | ctid | itemlen | nulls | vars | data | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 1 | (1,0) |     8 | f | f |       |       |       |       |
 2 | (2,1) |    16 | f | f | 6e 01 00 00 00 00 00 00 |       |       |
 3 | (4,1) |    16 | f | f | dc 02 00 00 00 00 00 00 |       |       |
 4 | (5,1) |    16 | f | f | 4a 04 00 00 00 00 00 00 |       |       |
 5 | (6,1) |    16 | f | f | b8 05 00 00 00 00 00 00 |       |       |
 6 | (7,1) |    16 | f | f | 26 07 00 00 00 00 00 00 |       |       |
 7 | (8,1) |    16 | f | f | 94 08 00 00 00 00 00 00 |       |       |
 8 | (9,1) |    16 | f | f | 02 0a 00 00 00 00 00 00 |       |       |
 9 | (10,1) |   16 | f | f | 70 0b 00 00 00 00 00 00 |       |       |
(9 rows)
```

```
oracle=# SELECT * FROM bt_page_stats('state_pkey',1);
-[ RECORD 1 ]-----
blkno | 1
type | 1
live_items | 367
dead_items | 0
avg_item_size | 16
page_size | 8192
free_size | 808
btpp_prev | 0
btpp_next | 2
btpp_level | 0
btpp_flags | 1
```

```
oracle=# SELECT * FROM bt_page_stats('state_pkey',2);
```

```
-[ RECORD 1 ]-----  
blkno      | 2  
type       | 1  
live_items | 367  
dead_items | 0  
avg_item_size | 16  
page_size   | 8192  
free_size   | 808  
btpp_o_prev | 1  
btpp_o_next | 4  
btpp_o_level| 0  
btpp_o_flags| 1
```

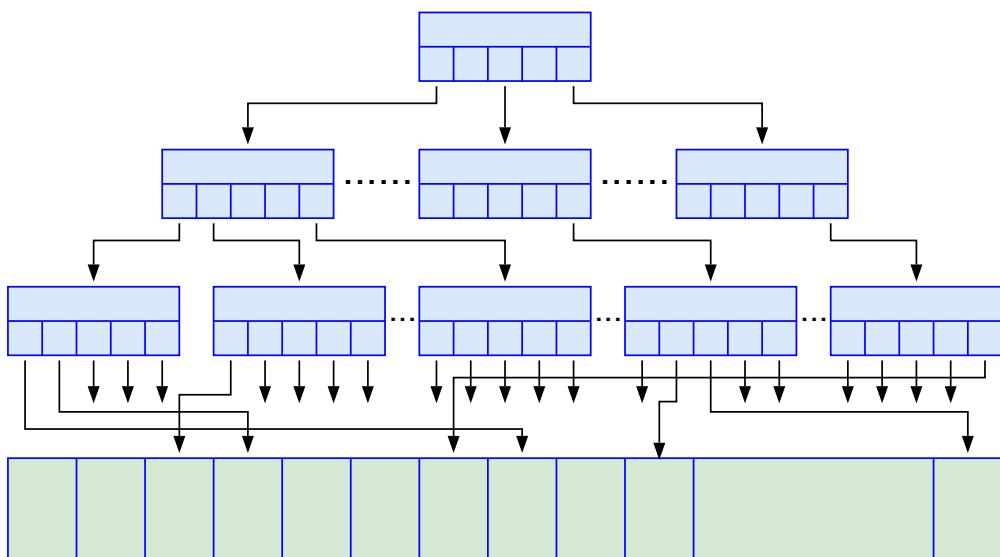


图 6.18: XXXXX

```
/* 步骤0 - 做一些准备工作：创建表，插入记录，查看磁盘上对应的文件 */  
oracle=# CREATE TABLE state(id INT PRIMARY KEY, name VARCHAR(64));  
CREATE TABLE  
oracle=# \d state  
      Table "public.state"  
 Column |          Type          | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
  id    | integer           |           | not null |  
  name  | character varying(64) |           |           |  
  
Indexes:  
  "state_pkey" PRIMARY KEY, btree (id)  
  
oracle=# SELECT pg_relation_filepath('state');  
-[ RECORD 1 ]-----+-----  
pg_relation_filepath | base/16384/16489  
  
oracle=# SELECT pg_relation_filepath('state_pkey');
```

```

-[ RECORD 1 ]-----+
pg_relation_filepath | base/16384/16492

oracle=# INSERT INTO state VALUES(0, 'Georgia'),(1, 'Delaware');
INSERT 0 2
/* 手工执行CheckPoint，确保所有的数据落盘 */
oracle=# CHECKPOINT;
CHECKPOINT
/* 注意索引文件的大小是16384= 2 * 8192，正好2个Block */
oracle=# \! ls -l $PGDATA/base/16384/16492
-rw----- 1 postgres postgres 16384 Mar 19 07:40 /opt/data/pgdata1/base/16384/16492
/* 步骤1 - 检查Meta Block里的内容，可见Root Block编号是1，层数是0 */
oracle=# SELECT magic, version, root, level FROM bt_metap('state_pkey');
magic | version | root | level
-----+-----+-----+
340322 |      4 |     1 |      0
(1 row)

/* 步骤2 - 检查Root Block里的内容，它里面有两条记录，且它既是Root，也是Leaf */
oracle=# SELECT * FROM bt_page_stats('state_pkey',1);
-[ RECORD 1 ]-----
blkno      | 1    /* 本Block的编号是1 */
type       | 1
live_items | 2    /* 本Block里面有2条记录 */
dead_items | 0
avg_item_size | 16
page_size   | 8192
free_size    | 8108
btppro_prev | 0    /* 本Block没有前导Block */
btppro_next  | 0    /* 本Block没有后导Block */
btppro_level | 0    /* 本Block的level是 0 */
btppro_flags | 3    /* BTP_LEAF | BTP_ROOT，既是Root，也是Leaf */
/* 步骤3 - 检查Root Block里两条记录的具体内容 */
oracle=# SELECT * FROM bt_page_items('state_pkey',1);
 itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+
 1 | (0,1) |      16 | f     | f     | 00 00 00 00 00 00 00 00 | f     | (0,1) |
 2 | (0,2) |      16 | f     | f     | 01 00 00 00 00 00 00 00 | f     | (0,2) |
(2 rows)

/* 步骤4 - 运行一个简单的脚本，往state表里再插入2999条记录 */
oracle=# \! cat /tmp/ins.sql
DO $$

BEGIN
  FOR i IN 2..3000 LOOP
    INSERT INTO state VALUES(i, 'Connecticut');
  END LOOP;
END; $$

oracle=# \i /tmp/ins.sql
DO
oracle=# CHECKPOINT;

```

```

CHECKPOINT
oracle=# SELECT count(*) FROM state;
count
-----
3001
(1 row)

/* 步骤5 - 检查Meta Block里的内容，可见Root Block编号已经变成了3。Block 1,2都是Leaf */
oracle=# SELECT magic, version, root, level FROM bt_metap('state_pkey');
magic | version | root | level
-----+-----+-----+-----
340322 |      4 |     3 |      1
(1 row)

/* 步骤6 - 检查Block 3, Root Block里面的内容，其包含9条记录，指向下层的9个Block */
oracle=# SELECT * FROM bt_page_stats('state_pkey',3);
-[ RECORD 1 ]-----
blkno      | 3
type       | r
live_items | 9 /* 这些Block的编号是1,2,4,5,6,7,8,9,10 */
dead_items | 0
avg_item_size | 15
page_size   | 8192
free_size   | 7976
btpp_prev   | 0
btpp_next   | 0
btpp_level  | 1
btpp_flags  | 2 /* BTP_ROOT */

/* 步骤7 - 检查Leaf Block 2, 它的前导Block是1, 后导Block是4 */
oracle=# SELECT * FROM bt_page_stats('state_pkey',2);
-[ RECORD 1 ]-----
blkno      | 2
type       | l
live_items | 367 /* 本Block里面有367条记录 */
dead_items | 0
avg_item_size | 16
page_size   | 8192
free_size   | 808
btpp_prev   | 1
btpp_next   | 4
btpp_level  | 0
btpp_flags  | 1

/* 步骤8 - 检查Leaf Block 10, 它的前导Block是9, 没有后导Block */
oracle=# SELECT * FROM bt_page_stats('state_pkey',10);
-[ RECORD 1 ]-----
blkno      | 10
type       | l
live_items | 73
dead_items | 0
avg_item_size | 16
page_size   | 8192

```

```

free_size      | 6688
btppro_prev    | 9
btppro_next    | 0
btppro_level   | 0
btppro_flags   | 1 /* BTP_LEAF */
/* 步骤9 - 检查Leaf Block 1, 注意第1条记录和第2,3条记录有所不同 */
oracle=# SELECT * FROM bt_page_items('state_pkey',1) LIMIT 3;
itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 1 | (1,1) |     16 | f     | f     | 6e 01 00 00 00 00 00 00 | | | |
 2 | (0,1) |     16 | f     | f     | 00 00 00 00 00 00 00 00 | f | (0,1) | |
 3 | (0,2) |     16 | f     | f     | 01 00 00 00 00 00 00 00 | f | (0,2) | |
(3 rows)
/* 步骤10 - 检查Leaf Block 2, 注意第1条记录和第2,3条记录有所不同 */
oracle=# SELECT * FROM bt_page_items('state_pkey',2) LIMIT 3;
itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 1 | (3,1) |     16 | f     | f     | dc 02 00 00 00 00 00 00 | | | |
 2 | (1,182) |    16 | f     | f     | 6e 01 00 00 00 00 00 00 | f | (1,182) | |
 3 | (1,183) |    16 | f     | f     | 6f 01 00 00 00 00 00 00 | f | (1,183) | |
(3 rows)
/* 步骤11 - 检查Leaf Block 10, 注意Block1,2中特殊的第1条记录没有了 */
oracle=# SELECT * FROM bt_page_items('state_pkey',10) LIMIT 3;
itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 1 | (15,154) |    16 | f     | f     | 70 0b 00 00 00 00 00 00 | f | (15,154) | |
 2 | (15,155) |    16 | f     | f     | 71 0b 00 00 00 00 00 00 | f | (15,155) | |
 3 | (15,156) |    16 | f     | f     | 72 0b 00 00 00 00 00 00 | f | (15,156) | |
(3 rows)
/* 步骤12 - 检查Root Block里面的记录信息 */
oracle=# SELECT * FROM bt_page_items('state_pkey',3);
itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 1 | (1,0) |      8 | f     | f     | | | | |
 2 | (2,1) |     16 | f     | f     | 6e 01 00 00 00 00 00 00 | | | |
 3 | (4,1) |     16 | f     | f     | dc 02 00 00 00 00 00 00 | | | |
 4 | (5,1) |     16 | f     | f     | 4a 04 00 00 00 00 00 00 | | | |
 5 | (6,1) |     16 | f     | f     | b8 05 00 00 00 00 00 00 | | | |
 6 | (7,1) |     16 | f     | f     | 26 07 00 00 00 00 00 00 | | | |
 7 | (8,1) |     16 | f     | f     | 94 08 00 00 00 00 00 00 | | | |
 8 | (9,1) |     16 | f     | f     | 02 0a 00 00 00 00 00 00 | | | |
 9 | (10,1) |    16 | f     | f     | 70 0b 00 00 00 00 00 00 | | | |
(9 rows)

```

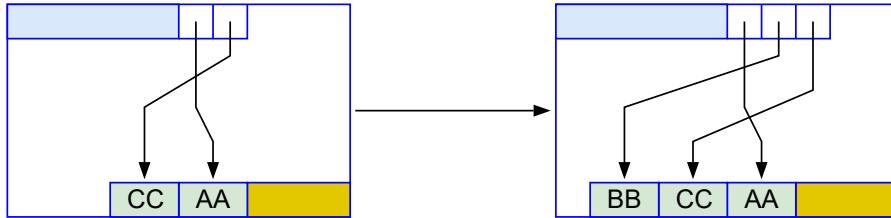


图 6.19: XXXXX

```

oracle=# CREATE TABLE btorder(id CHAR(2) NOT NULL, value VARCHAR(64));
CREATE TABLE
oracle=# CREATE INDEX btorder_idx ON btorder USING BTREE(id);
CREATE INDEX
oracle=# INSERT INTO btorder(id, value) VALUES('AA', 'Alabama');
INSERT 0 1
oracle=# INSERT INTO btorder(id, value) VALUES('CC', 'California');
INSERT 0 1
oracle=# CHECKPOINT;
CHECKPOINT
oracle=# SELECT * FROM bt_page_items('btorder_idx', 1);
itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
      1 | (0,1) |      16 | f     | t     | 07 41 41 00 00 00 00 00 | f     | (0,1) |
      2 | (0,2) |      16 | f     | t     | 07 43 43 00 00 00 00 00 | f     | (0,2) |
(2 rows)

oracle=# INSERT INTO btorder(id, value) VALUES('BB', 'Pennsylvania');
INSERT 0 1
oracle=# SELECT * FROM bt_page_items('btorder_idx', 1);
itemoffset | ctid | itemlen | nulls | vars |          data          | dead | htid | tids
-----+-----+-----+-----+-----+-----+-----+-----+-----+
      1 | (0,1) |      16 | f     | t     | 07 41 41 00 00 00 00 00 | f     | (0,1) |
      2 | (0,3) |      16 | f     | t     | 07 42 42 00 00 00 00 00 | f     | (0,3) |
      3 | (0,2) |      16 | f     | t     | 07 43 43 00 00 00 00 00 | f     | (0,2) |
(3 rows)

oracle=# SELECT lp, lp_off, lp_len, lp_flags FROM heap_page_items(get_raw_page('btorder_idx', 1));
lp | lp_off | lp_len | lp_flags
---+-----+-----+-----
  1 |   8160 |     16 |       1
  2 |   8128 |     16 |       1
  3 |   8144 |     16 |       1
(3 rows)

```

6.2.3 集簇(Cluster)

堆表是无序的。

第七章 多版本并发控制

数据库可以保存数据，文件系统也可以保存数据。我们天天使用的 Windows 操作系统，其中的 C 盘和 D 盘保存着我们非常重要的文件。那么数据库有别于文件系统的独特价值在哪里呢？一个是数据库是建立在关系模型上，提供了 SQL 语言的接口，可以帮助我们快速地查找所需要的数据，这个是文件系统欠缺的。数据库另外一个非常重要的价值在于保证数据的完整性，一个经典的例子是银行转账，我们要求一笔钱在一个账户上减少，在另外一个账户上增加，这两个动作必须同时成功或者同时失败。数据库提供了事务 (transaction) 的支持来保证这种重要的特性。PostgreSQL 采用了多版本并发控制 (MVCC: Multi-Version Concurrency Control) 模型来实现对事务的支持。本章讨论事务，MVCC 和与之而来的 PostgreSQL 所特有的 vacuum 的问题。

<https://en.wikibooks.org/wiki/PostgreSQL/Visibility>

7.1 事务

数据库事务最著名的就是 ACID 四大特性。

- 原子性 (Atomicity): 事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
- 一致性 (Consistency): 事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束。
- 隔离性 (Isolation): 多个事务并发执行时，一个事务的执行不应影响其他事务的执行。
- 持久性 (Durability): 已被提交的事务对数据库的修改应该永久保存在数据库中。

在 PostgreSQL 中，事务的状态有四种：正在进行中，已经提交，已经回滚和子事务已经提交。前三种的含义是非常明显的，最后一种状态我们不予讨论。在源代码中，有如下的四个常量分别表示这四种状态。

```
/* in src/include/access/clog.h */  
#define TRANSACTION_STATUS_IN_PROGRESS      0x00  
#define TRANSACTION_STATUS_COMMITTED        0x01  
#define TRANSACTION_STATUS_ABORTED         0x02  
#define TRANSACTION_STATUS_SUB_COMMITTED    0x03
```

事务的四个特性中，持久性我们已经有一定程度的理解：当一个事务被成功提交之后，数据库必须保证它对应的 WAL 记录已经被可靠地写入到次磁盘上了。

7.1.1 事务并发带来的结果

假设有一张表只有两列 (id INT, value INT)，下面是两个事务 (TA 和 TB) 并发执行时可能出现的一些结果：

- TA 读取了 id=1 的记录，其 value 列的值为 X。与此同时，TB 也读取了该条记录，也拿到了 X。TA 把 X 的值增加了 1，TB 也把 X 的值增加了 1，两者都会认为在本事务提交后，最终的结果应该是 X+1。但我们会从后面的实验中看到，最终的结果是 X+2，这并不是两个事务想要的结果。此种情况被称之为“更新丢失”(lost update)。
- TA 把 id=1 的 value 从 X 修改为 X+1，那么 TB 在 TA 提交之前，它会看到 X，还是 X+1？如果它在 TA 未被提交之前看到了 X+1，则此种情况被称为“脏读”(dirty read)，脏读在 PostgreSQL 中是不被允许的。
- TA 读取了 id=1 的记录，其 value 为 X。TB 也读取了该条记录，把 X 修改为了 X+1，并提交了。TA 再次读取该条记录，发现 value 变成了 X+1。也就是说，TA 两次读取同一条记录，发现了不同的值。此种情况被称为“不可重复读”(non-repeatable read)。

- TA 按某种过滤条件读取表，发现了 n 条记录。TB 往该表中插入了一条新记录，并提交了。TA 再次按同样的过滤条件查询该表，发现了 n+1 条记录。TA 两次查询的结果集并不一样，此种情况被称之为“幻读”(phantom read)。
- TA 读取并更新了 id=1 的记录的 value 列。TB 读取并更新了 id=2 的记录的 value 列，TB 又读取试图更新 id=1 的记录的 value 列，但此时 TA 并没有提交，所以 TB 必须等待 TA 的提交或者回滚。紧接着 TA 试图更新 id=2 的记录的 value 列，同样的道理，因为 TB 没有提交，所以 TA 也必须等待 TB 的提交或者回滚。TA 和 TB 都陷入了无限期等待对方做下一步动作的泥潭中无法动弹。这种情况被称之为“死锁”(dead lock)。

下面的实验演示了以上各种情况。首先准备一张测试表，并往其中插入一条记录：

```
oracle=# CREATE TABLE population (id CHAR(2), value INT);
CREATE TABLE
oracle=# INSERT INTO population VALUES('TX', 100);
INSERT 0 1
oracle=# select id, value from population order by id;
 id | value
----+-----
 TX |    100
(1 row)
```

7.1.1.1 提交读 (Read Committed)

现在打开两个 putty 窗口，分别连接到同一个数据库集群上，具体实验操作步骤如下：

```
----- 会话A/事务A
oracle=# BEGIN TRANSACTION;
BEGIN
oracle==# update population set value=101 where id='TX';
UPDATE 1
oracle==# select id, value from population order by id;
 id | value
----+-----
 TX |    101
(1 row)
+
| ----- 会话B/事务B
| oracle=# select id, value from population order by id;
| id | value
| ----+-----
| TX |    100
| (1 row)
+
| oracle=# select id, value from population order by id;
| id | value
| ----+-----
| TX |    101
```

```
| (1 row)
```

从上面的实验结果，我们可以很清楚地看到：在事务 A 未被提交之前，事务 B 并不能看到它的修改。只有事务 A 提交以后，事务 B 才能看到它的修改，也就是说没有脏读的发生。这是 PostgreSQL 的缺省隔离模式 - 提交读。但是在事务 A 中，前一条修改操作即使没有提交，后面的查询命令依然可以看到修改结果，这个是很自然的预期，同一个事务中的 SQL 命令可以看到前面 SQL 命令的修改结果，不需要等到该事务被提交。

7.1.1.2 不可重复读 (Non-repeatable Read)

```
----- 会话A/事务A
oracle=# BEGIN;
BEGIN
oracle=# select id, value from population order by id;
 id | value
-----+
 TX |    101
(1 row)

----- 会话B/事务B
| oracle=# BEGIN;
| BEGIN
| oracle=# update population set value = value + 1 where id='TX';
| UPDATE 1
| oracle=# select id, value from population order by id;
| id | value
| -----+
| TX |    102
| (1 row)
| oracle=# COMMIT;
| COMMIT

-----+
oracle=# select id, value from population order by id;
 id | value
-----+
 TX |    102
(1 row)
```

从上面的实验结果，我们可以很清楚地看到：虽然事务 B 比事务 A 后发起，但是 B 先提交了。A 在 B 提交前和提交后读取的结果不一样。这种情况就是“不可重复读”。

7.1.1.3 幻读 (Phantom read)

```
----- 会话A/事务A
oracle=# BEGIN;
BEGIN
oracle=# select count(1) from population;
 count
-----
 1
```

```
(1 row)
-----
| ----- 会话B/事务B
| oracle=# BEGIN;
| BEGIN
| oracle==# insert into population values('MA', 200);
| INSERT 0 1
| oracle==# COMMIT;
| COMMIT
-----
oracle==# select count(1) from population;
count
-----
2
(1 row)
```

从上面的实验结果，我们可以很清楚地看到：虽然事务 B 比事务 A 后发起，但是 B 先提交了。A 在 B 提交前看到表中有一条记录，但在 B 提交后却看到了 2 条记录，这就是“幻读”。

7.1.1.4 修改丢失 (Lost Update)

修改丢失指的是当两个事务读取同一条记录后，其中一个事务更新了这条记录，后来第二个事务也更新了这条记录，但是第二个事务并没有考虑第一个事务的更新。举一个例子：一条记录包含了 100\$，两个事务 A 和 B 都读取了这个值，并且试图把该账户余额增加 10\$。事务 A 把 100 变成 110，事务 B 也把 100 变成了 110。最终数据库中保存的结果是 110。但是有两个事务分别往这个账户增加了 10\$，最终的余额应该是 120\$ 猜对，怎么会是 110\$ 呢？很显然该账户的用户损失了 10\$，这就是修改丢失。修改丢失是数据库应该禁止发生的行为。

```
-- session A / transaction A
oracle=# BEGIN;
BEGIN
oracle==# select id, value from population order by id;
id | value
-----+
TX |    101
(1 row)

oracle==# update population set value = value + 1 where id='TX';
UPDATE 1
oracle==# select id, value from population order by id;
id | value
-----+
TX |    102
(1 row)

-----
| -- session B / transaction B
| oracle=# BEGIN;
| BEGIN
| oracle==# select id, value from population order by id;
| id | value
| -----+
```

```

| TX | 101
| (1 row)
| oracle=># update population set value = value + 1 where id='TX';
| /* 因为A和B都修改同一条记录，本UPDATE会被阻滞，等待A的提交或回滚 */
-----
oracle=># COMMIT;
COMMIT /* Transaction A 提交完毕 */

-----
| UPDATE 1 /* A提交后，本UPDATE才可以正常完成 */
| oracle=># select id, value from population order by id;
| id | value
| ----+---
| TX | 103 /* 结果发现修改的结果不是本Transaction期望的结果 */
| (1 row)
| oracle=># COMMIT;
| COMMIT
| oracle=># select id, value from population order by id;
| id | value
| ----+---
| TX | 103
| (1 row)

-----
oracle=> select id, value from population order by id;
id | value
----+---
TX | 103 /* Transaction A 提交前和提交后看到的结果是不一样的 */
(1 row)

```

从上面的实验结果，我们可以很清楚地看到：事务 A 看到原始值是 101，它决定把这个值加一，变成了 102，结果提交本事务以后，却惊讶地发现最终的值是 103。事务 B 也面临同样的问题，明明想把看到的 101 变成 102，结果最终的结果却是 103，两个事务的修改都“丢失”了。

7.1.1.5 死锁 (Dead lock)

```

oracle=> select id, value from population order by id;
id | value
----+---
MA | 200
TX | 100
(2 rows)

-----
----- 会话A/事务A
oracle=> BEGIN;
BEGIN
oracle=># update population set value = value + 1 where id='TX';
UPDATE 1

-----
| ----- 会话B/事务B
| oracle=> BEGIN;

```

```

| BEGIN
| oracle=# update population set value = value + 1 where id='MA';
| UPDATE 1
| oracle=# update population set value = value + 1 where id='TX';
| /* 因为A和B都修改同一条记录, 本UPDATE会被阻滞, 等待A的提交或回滚 */
+-----+
oracle=# update population set value = value + 1 where id='MA';
/* 过了一会, PG检测到死锁的发生, 抛出如下信息, 然后A被回滚了 */
ERROR: deadlock detected
DETAIL: Process 8055 waits for ShareLock on transaction 744; blocked by process 8072.
Process 8072 waits for ShareLock on transaction 743; blocked by process 8055.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,2) in relation "population"
+-----+
| UPDATE 1 /* A被回滚后, B可以继续运行 */
| oracle=# COMMIT;
| COMMIT
| oracle=# select id, value from population order by id;
|   id | value
| -----+
|   MA |    201
|   TX |    101
| (2 rows)
+-----+
oracle!=# COMMIT;
ROLLBACK

```

在本次实验中, 事务 A 修改了一条记录, 事务 B 修改了另外一条记录, 它们彼此都想继续修改对方正在修改的记录, 每个人手里都拿到了一个东西, 又想抢夺对方手里的东西, 互相僵持, 这种状态就是死锁。

7.1.2 事务的隔离级别

在理解了不可重复读, 幻读等基本概念后, 我们研究一下 PostgreSQL 的事务隔离机制。在 SQL 标准定义了四种事务的隔离模式, 分别是未提交读 (Read Uncommitted)、提交读 (Read Committed)、可重复读 (Repeatable Read) 和序列化 (Serializable) 四种事务的隔离机制。PostgreSQL 实现了 SQL 标准中的后 3 种, 不支持未提交读。

7.2 多版本并发控制

多版本并发控制 (MVCC: multi-version concurrency control) 是一种保证事务的原子性和隔离性的技术，也是主流的关系型数据库所采用的主要技术框架。它的核心思想是：当对记录进行修改时，保存该记录修改之前的老版本，即老值。当另一个事务读取这个记录中的数据时，数据库会根据事务隔离级别的规则，选择合适的版本。MVCC 最大的优点是：读不阻塞写，写不阻塞读。当然，如果两个客户同时修改同一条记录，先发起的写操作当然会阻塞后发起的写操作，这是无法避免的。

7.2.1 事务号

每当一个事务发起时，PostgreSQL 会为其分配一个在数据库集群范围内唯一的标识，即事务号 (Transaction Id)，也被称为 XID。XID 是一个 4 字节的无符号整数，其范围是 0 到 0xFFFFFFFF，但其中的 0, 1, 2 三个 XID 被预留了，作为特殊的事务号：0 表示是无效的 XID，1 表示 Bootstrap 的 XID，2 表示被冻结的 XID，关于事务冻结，我们后面会详细讨论。所以可以被使用的普通事务号的范围是 3 到 0xFFFFFFFF，其具体定义如下：

```
typedef uint32 TransactionId;
/* in src/include/access/transam.h */
#define InvalidTransactionId      ((TransactionId) 0)
#define BootstrapTransactionId    ((TransactionId) 1)
#define FrozenTransactionId       ((TransactionId) 2)
#define FirstNormalTransactionId ((TransactionId) 3)
#define MaxTransactionId         ((TransactionId) 0xFFFFFFFF)
```

给定一个事务号 XID，只要判断它是否大于等于 3 就可以知道它是普通的事务号还是特殊的事务号。在任何时刻，整个数据库集群中有一个唯一的当前 XID(current XID)，当前 XID 会随着新的事务的发起，不断递增，每次加 1，请参考图 XX。

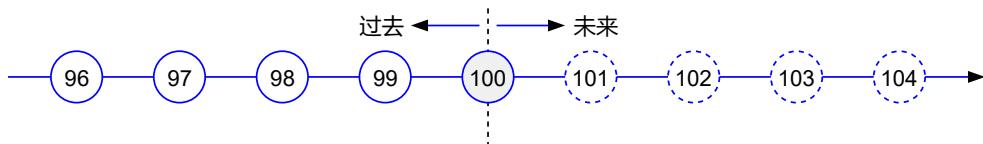


图 7.1: 事务号 XID

上图中，当前事务的 XID 是 100，则 99 号事务先于 100 号发生，处于其过去的历史当中，而 101 号事务还没有发生，处于 100 号事务的未来之中。所以从当前事务的角度看，99 号事务的修改是可以看见的，而 101 号事务的修改是不可能看见的，因为未来可以预测，但是不可以实实在在地看到它的发生。在这里我们再定义一下“前”和“后”，“老”(old) 和“新”(young) 等术语的含义，避免后文中产生混淆。现假设有两个 XID: XID_1 , XID_2 。

- 如果 XID_1 先于 XID_2 发生，则称 XID_1 在 XID_2 的前面，而 XID_2 在 XID_1 的后面。
- 如果 XID_1 先于 XID_2 发生，则称 XID_1 比 XID_2 老， XID_2 比 XID_1 新。

我们通过一个小实验展示了 XID 的变化规律。使用 psql 登录到数据库中，执行如下命令：

```
oracle=# BEGIN; /* 显式地发起一个事务 */
BEGIN
oracle==# SELECT pg_current_xact_id();
 pg_current_xact_id
```

```

-----
745
(1 row)
oracle=# SELECT pg_current_xact_id();
pg_current_xact_id
-----
745 /* <-- 注意，事务号并没有发生变化 */
(1 row)
oracle=# COMMIT;
COMMIT
oracle=# SELECT pg_current_xact_id(); /* 本SELECT是一个独立的事务 */
pg_current_xact_id
-----
746
(1 row)
oracle=# SELECT pg_current_xact_id(); /* 本SELECT是一个新的事务，和前一个SELECT无关 */
pg_current_xact_id
-----
747 /* <-- 注意，事务号发生了变化 */
(1 row)

```

我们要注意一点：在 PostgreSQL 中，INSERT/UPDATE/DELETE 是事务，SELECT 也是事务，也拥有事务号 XID。在上述实验中，第一次用 BEGIN 命令显式地发起了一个事务，PostgreSQL 为其分配了一个唯一的 XID。在该事务没有被提交或者回滚之前，其 XID 是固定不变的。第二次和第三次的查询语句处于自动提交模式下，这两条查询命令就是两个独立的事务，所以它们都拥有属于自己的 XID，所以你看到每次 SELECT 语句的事务号是不同的。

由于 XID 只有 4 个字节的长度，其空间中一共有 2^{32} 个数字，可用的 XID 为 $2^{32} - 3$ ，所以存在 XID 被用完的情况。当 XID = 0xFFFFFFFF 时，再加 1 就溢出了，变成 0 了，下面简单的 C 程序展示了从 0xFFFFFFFF 变到 0 的现象：

```

$ cat w.c
#include <stdio.h>
int main(int argc, char* argv[])
{
    int x = 0xFFFFFFFF;
    int y = x + 1;
    printf("x=%u y=%u\n", x, y);
    return 0;
}
$ gcc w.c
$ ./a.out
x=4294967295 y=0

```

我们可以看到，0xFFFFFFFF 加 1 就变成了 0。这种现象被称为 XID 的“回卷”(wraparound)，XID 每发生一次回卷现象，被称为一个“世代”(epoch)。把事务号设置为 4 个字节，可以包含 40 多亿个事务，看似够用了，实则太短了。譬如假设一个数据库每秒钟产生 1000 个事务，大概过了 49 天就会发生一次回卷。32 位事务号带来了一系列的问题，所以它是 PostgreSQL 的一个痛点，选择 64 位的事务号是一个用户迫切的要求，但是后文中我们会看到每一条记录都包含两个事务号，如果采用 64 位事务号，就要额外增加 8 个字节的开销，这个成本是巨大的。目前还没有 64 位事务号的实现计划。

宏 TransactionIdAdvance(xid) 用于把 XID 不断递增，它的代码如下：

```
/* in src/include/access/transam.h */
/* advance a transaction ID variable, handling wraparound correctly */
#define TransactionIdAdvance(dest) \
do { (dest)++; \
    if ((dest) < FirstNormalTransactionId) (dest) = FirstNormalTransactionId; \
} while(0)
```

它的代码非常容易理解，当 XID 发生回卷时，也就是从 0xFFFFFFFF 变成 0 时，直接从 0 跳到了第一个普通的 XID，即 3。我们可以把 XID 的空间想象成一个环形，站在当前事务的角度来看，整个 XID 的空间被均匀地分为两部分，一部分是在它前面已经发生的老事务的 XID，另外一部分是在它后面的，未来的事务可用的 XID，这两部分各有 2^{31} 个 XID(扣除 0、1、2 这三个特殊的 XID)。图 XX 展示了这种概念。

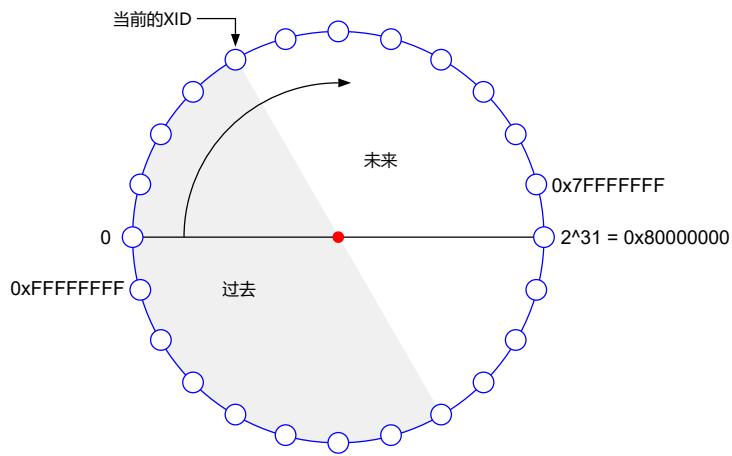


图 7.2: 事务号 XID 的空间

由于当前事务是不断递增的，即图 XX 中的顺时针旋转，所以过去和未来的范围也处于不断变化中。假设当前事务的事务号为 C_{xid} ，在它之前的，老的事务的事务号为 P_{xid} ，在它之后的尚未发生的新的事务的事务号为 F_{xid} 。则 P_{xid} 和 F_{xid} 有如下规律：

```
IF(Cxid < 2^31) THEN
    Pxid < Cxid OR Pxid > Cxid + 2^31
    Fxid > Cxid AND Fxid <= Cxid + 2^31
ELSE
    Pxid < Cxid AND Pxid > Cxid - 2^31
    Fxid > Cxid OR Fxid <= Cxid - 2^31
ENDIF
```

结合图 XX，就很容易理解上述算法的含义。你可以把当前事务的事务号 XID 分成两种情况，图中的左边表示了 XID 小于 2^{31} 的情况；右边表示 XID 大于等于 2^{31} 的情况。这两种情况对应上上述算法的两种情况。

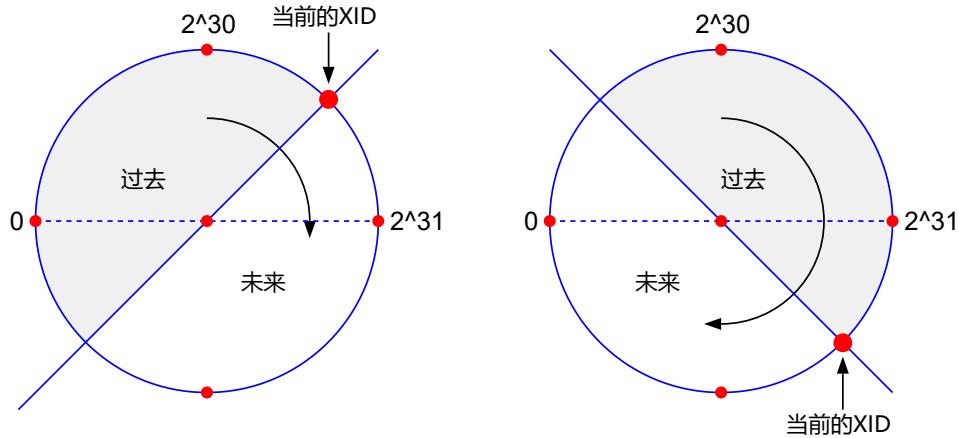


图 7.3: 当前事务的过去和未来

给定两个 XID: id1 和 id2, 如何判断 id1 是否在 id2 之前呢? 下面的代码给出了判断的算法:

```
/* in src/backend/access/transam/transam.c */
#define FirstNormalTransactionId ((TransactionId) 3)
#define TransactionIdIsNormal(xid) ((xid) >= FirstNormalTransactionId)
bool TransactionIdPrecedes(TransactionId id1, TransactionId id2)
{
    int32 diff;
    if (!TransactionIdIsNormal(id1) || !TransactionIdIsNormal(id2)) return (id1 < id2);
    diff = (int32) (id1 - id2);
    return (diff < 0);
}
```

函数 `TransactionIdPrecedes(id1, id2)` 判断 id1 是否在 id2 之前发生, 如果 id1 在 id2 之前, 则该函数返回 true, 否则返回 false。你可以选择几个测试值试试, 譬如 id1=5, id2=6, 则该函数返回 true, 表明事务 5 在事务 6 之前。如果 id1 = 0xFFFFFFFF, id2 = 3, 虽然 id1 是远远大于 id2 的一个数值, 但是 id1 依然是比 id2 老的事务号。因为根据上面的逻辑, $diff = id1 - id2 = 0xFFFFFFFF - 3 = 0xFFFFFFFFFC$ 。注意: diff 是一个有符号的整型。对于有符号的整型来说, 0xFFFFFFFFFC 实际上表示了 -4, 则 $diff < 0$ 为 true, 表明 0xFFFFFFFF 在 3 之前。这个算法利用了有符号整数的范围和无符号整数范围的差异来完成判断的, 其逻辑不难理解。

https://en.wikibooks.org/wiki/PostgreSQL/Wraparound_and_Freeze

7.2.1.1 记录的结构

在第二章中, 我们学习了数据页和记录的基本结构。我们知道, 每一个数据页包含如下几部分内容:

- 数据页的页头, 共计 24 个字节。
- 一个记录指针数组, 其中每一个成员均为 4 个字节。
- 空闲空间
- 一条条记录 (记录又可以被称为“元组”)
- 特殊空间

关于数据页的整体结构, 可以参考第二章中图 2.2 进行理解。我们看一下每条记录的结构细节。以及它和事务之间的关系。图 XX 展示了一条记录的基本结构。

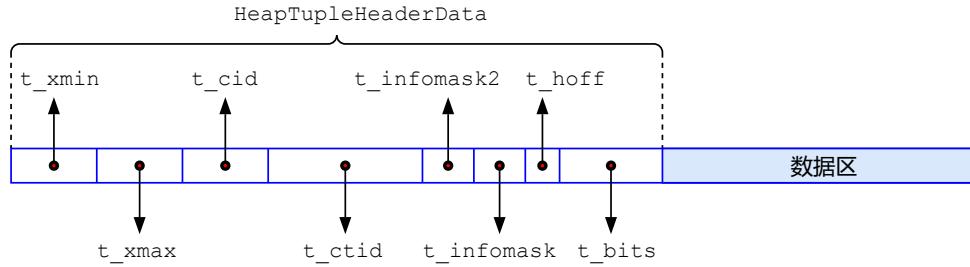


图 7.4: 记录的基本结构

从上图中我们可以看到，一条记录分为两部分，真正的数据和记录头。真正的数据千变万化，但是记录头是固定的，所以我们研究记录头的数据结构，PostgreSQL 就是利用记录头的这些信息实现了 MVCC。记录头的具体数据结构如下：

```

/* in src/include/storage/block.h */
typedef struct BlockIdData {
    uint16          bi_hi;
    uint16          bi_lo;
} BlockIdData;

/* in src/include/storage/itemptr.h */
typedef uint16 OffsetNumber; /* 2 bytes */
typedef struct ItemPointerData {
    BlockIdData ip_blkid;
    OffsetNumber ip_posid;
}

/* in src/include/access/htup_details.h */
typedef uint32 TransactionId; /* 4 bytes */
typedef uint32 CommandId; /* 4 bytes */
typedef struct HeapTupleFields {
    TransactionId t_xmin;           /* inserting xact ID */
    TransactionId t_xmax;           /* deleting or locking xact ID */
    union {
        CommandId   t_cid;         /* inserting or deleting command ID, or both */
        TransactionId t_xvac;      /* old-style VACUUM FULL xact ID */
    } t_field3;
} HeapTupleFields;
struct HeapTupleHeaderData {
    union {
        HeapTupleFields t_heap;
        DatumTupleFields t_datum;
    } t_choice;
    ItemPointerData t_ctid;
    uint16  t_infomask2;
    uint16  t_infomask;
    uint8   t_hoff;
    bits8   t_bits[FLEXIBLE_ARRAY_MEMBER];
};

```

上面的数据结构虽然复杂，我们现在只关心 `t_xmin`, `t_xmax` 这两个事务号和 `t_ctid`。大家知道，对一条记录的修改无非三种操作：增删改，即 `INSERT/DELETE/UPDATE`。头两个操作都非常容易理解。当 PostgreSQL 插入一条记录时，插入这条记录的事务的 XID 被存储在 `t_xmin` 中，`t_xmax` 的值设为 0。当删除一条记录时，并不会真正把这条记录从磁盘上删除，而是把删除这条记录的事务的 XID 记录在 `t_xmax` 中。所以我们可以根据 `t_xmax` 是否为 0 来判断这条记录是否被“逻辑”地删除了。`UPDATE` 操作的逻辑并不是直接修改记录，而是分成两个操作：先删后增，就是先把老记录做一个删除标志，再插入一条新记录。这里的原因也不难理解，当事务 A 修改一条记录时，譬如把老值 123 变成新值 321，它可能还没有提交，根据我们前一节的实验得知，为了满足事务的隔离需要，另外一个事务 B 必须看到老值 123，所以 `UPDATE` 操作必须同时保存老值和新值这两个版本。

理解了 PostgreSQL 的 `UPDATE` 操作的逻辑，`t_xmin` 和 `t_xmax` 的含义就不难理解了。当 PostgreSQL 插入一条记录时，会把插入该记录的事务的 XID 记录在 `t_xmin` 中，同时设置该条记录的 `t_xmax` 为 0。下面的实验就揭示了这个规律。因为这个实验用到了 `pageinspect` 这个扩展，所以你需要提前把这个扩展安装一下，请参考第二章中关于 `pageinspect` 的安装和使用的内容。

```
oracle=# create table state(id char(2), name varchar(64));
CREATE TABLE
oracle=# insert into state(id, name) values('TX', 'Texas');
INSERT 0 1
oracle=# \! cat /tmp/s.sql          /* 我们把查询语句写入一个脚本s.sql里 */
select lp, lp_flags, t_xmin, t_xmax, t_ctid, to_hex(t_infomask) as infomask
from heap_page_items(get_raw_page('state',0)) order by lp;
oracle=# \i /tmp/s.sql
lp | lp_flags | t_xmin | t_xmax | t_ctid | infomask
---+---+---+---+---+---
 1 |       1 |    749 |      0 | (0,1) | 802
(1 row)
```

从上面的实验中可以看出，当我们插入第一条记录时，`t_xmin` 的值是 749，表明这条记录是事务 749 插入的。它的 `t_xmax` 为 0，表明没有任何事务删除这条记录。同时我们也留意一下 `t_ctid` 这一列的信息。`t_ctid` 的值有两个，表示形式为 (x,y)。根据前面 `t_ctid` 数据结构的定义，我们不难理解，x 表示记录所在的数据页的编号 (BlockIdData)，y 则表示这条记录是该数据页的第几条记录 (OffsetNumber)。譬如：`t_ctid=(0,1)` 表示 0 号数据页的第一条记录；`t_ctid=(123,456)` 则表明 123 号数据页的第 456 条记录，以此类推。下面我们修改这条记录，执行结果如下：

```
oracle=# update state set name='Dallas' where id='TX';
UPDATE 1
oracle=# \i /tmp/s.sql
lp | lp_flags | t_xmin | t_xmax | t_ctid | infomask
---+---+---+---+---+---
 1 |       1 |    749 |    751 | (0,2) | 102
 2 |       1 |    751 |      0 | (0,2) | 2802
(2 rows)
```

我们可以看到，根据第一条记录中的 `t_xmax=751` 可以得知，`UPDATE` 操作实际上就是把第一条记录删除掉了。第二条记录是 `UPDATE` 操作插入的新记录。这个实验清楚地表明了 `UPDATE` 操作的实质是“先删后增”。第一条记录只是标识了被删除，但是它的值依然还在。其它事务就可以读取第一条记录的老值来保证它们只看到了事务 751 提交之前的值。这就是 MVCC 中 MV 的含义，即多版本 (multiple version)，通过保存老值和新值来保证事务的隔离级别。我们同时也注意到了这两条记录的 `t_ctid` 都是 (0,2)，表示它们指向了这条记录的最新版本，`t_ctid` 实际上是一个指针，把一条记录的不同版本串联起来。如果 `UPDATE` 操作中发现本数据页的空间不够，则新插入的记录会被放在另外一个空间足够的数据页上，`t_ctid` 的第一个值指向了新的数据页的编号。

下面我们再研究一下删除操作的背后发生了什么。执行如下操作：

```
oracle=# insert into state(id, name) values('ME', 'Maine');
INSERT 0 1
oracle=# \i /tmp/s.sql
lp | lp_flags | t_xmin | t_xmax | t_ctid | infomask
-----+-----+-----+-----+-----+
 1 |      1 |    749 |    751 | (0,2) | 102
 2 |      1 |    751 |      0 | (0,2) | 2802
 3 |      1 |    752 |      0 | (0,3) | 802
(3 rows)

oracle=# delete from state where id='ME';
DELETE 1
oracle=# \i /tmp/s.sql
lp | lp_flags | t_xmin | t_xmax | t_ctid | infomask
-----+-----+-----+-----+-----+
 1 |      1 |    749 |    751 | (0,2) | 502
 2 |      1 |    751 |      0 | (0,2) | 2902
 3 |      1 |    752 |    753 | (0,3) | 102
(3 rows)

oracle=# select id, name from state order by id;
 id | name
----+-----
 TX | Dallas
(1 row)
```

我们可以看到，删除操作就是简单地把该条记录的 `t_xmax` 变成了删除该条记录的事务的 XID，这里是 753。而正常的查询只能看到没有被删除的记录，所以查询该表时只能看到第二条记录，第一条和第三条记录被删除了，无法看到，只能通过 `pageinspect` 插件或者 `hexdump` 这种工具才能够看到。图 XX 展示了上述实验完成后这个数据页的布局。

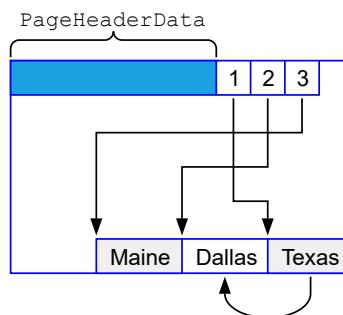


图 7.5：修改和删除后数据块的布局

当删除操作被提交以后，后续发起的事务无论如何是看不到第三条记录的，所以第三条记录是“死”记录，即没有任何再利用的价值。UPDATE 操作提交后，或者回滚后，第一条记录也是死记录。这些记录需要被定期清理掉，才能够保证表的体积不会膨胀太多。清理死记录是后文中 Vacuum 操作的主要内容之一。

7.2.2 事务的冻结

7.2.2.1 事务的年龄

我们都理解出生日期和年龄这两个概念，譬如：小明的出生日期是 1995 年 5 月 18 日，假设今天是 2024 年 5 月 18 日，他的年龄就是 29 岁，即当前时间减去他的出生时间。事务号可以理解为一个事务的出生日期，我们比较两个事务的事务号，就可以知道谁比谁老。另外一个重要的概念是事务的年龄。类似人类的年龄，事务的年龄就是当前事务的 XID 减去该事务的 XID。假设当前事务的 XID 是 100，则事务号 99 的年龄为 1(等于 100 - 99)，事务号 81 的年龄是 19(等于 100 - 81)。由于当前事务的 XID 是不断增加的，所以对于给定的事务号 xid，它的年龄也是随着时间的流逝，更准确的说法是随着新事务的产生而不断增加。我们可以通过 age() 这个函数计算一个事务号的年龄：

```
oracle=# select 100::text as xid, age(100::text::xid), pg_current_xact_id() as current_xid;
  xid | age | current_xid
-----+----+
  100 | 658 |      758
(1 row)

oracle=# select 100::text as xid, age(100::text::xid), pg_current_xact_id() as current_xid;
  xid | age | current_xid
-----+----+
  100 | 659 |      759
(1 row)

oracle=# select age(0::text::xid) as age0, age(1::text::xid) as age1, age(2::text::xid) as age2;
  age0 |   age1 |   age2
-----+----+
2147483647 | 2147483647 | 2147483647
(1 row)
```

我们可以看到，对于一个正常事务号，即不是 0、1、2 这三个值，其年龄是当前事务号和它的差值。对于 3 个特殊的事务号，它们的年龄恒定为 2147483647，即 $2^{31}-1$ 。关于计算某一个事务号的年龄的函数代码如下，从代码中我们可以很清楚地看到它的逻辑，不难理解。

```
/* in src/backend/utils/adt/xid.c */
Datum xid_age(PG_FUNCTION_ARGS)
{
    TransactionId xid = PG_GETARG_TRANSACTIONID(0);
    TransactionId now = GetStableLatestTransactionId();
    /* Permanent XIDs are always infinitely old */
    if (!TransactionIdIsNormal(xid)) PG_RETURN_INT32(INT_MAX);
    PG_RETURN_INT32((int32) (now - xid));
}
```

我们看一下图 XX 所展示的事务过程。

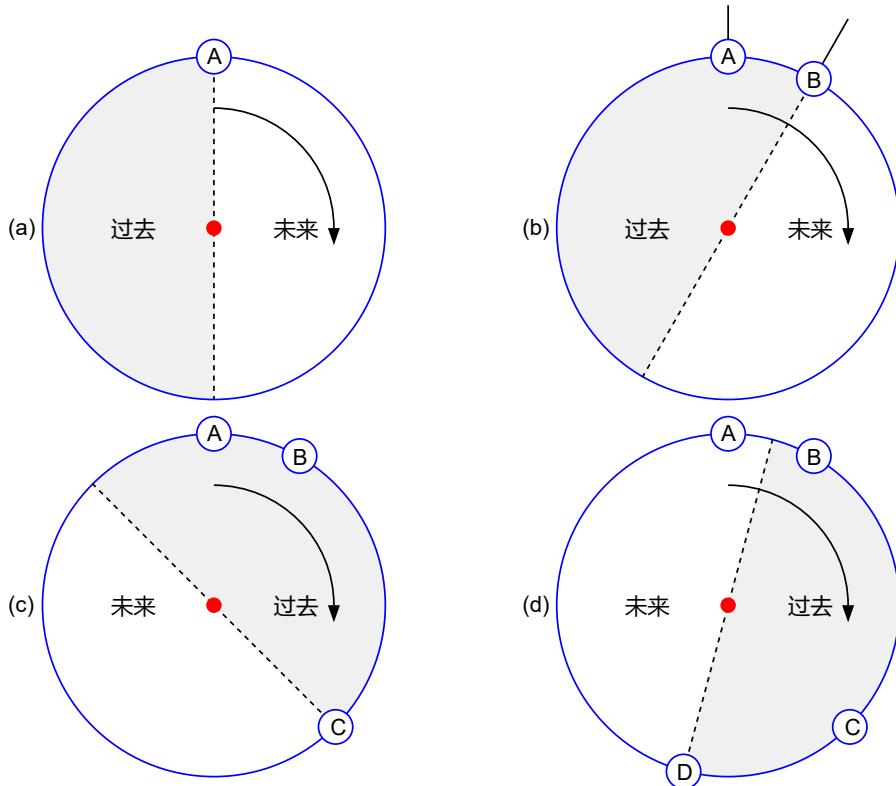


图 7.6: 事务的冻结

假设某个事务 A 在 A 点插入了一条记录 R，则这条记录的 t_{xmin} 的事务号为 A。数据库继续运行，当前事务号为 B，事务 A 的年龄变成了 $B - A$ ，它处于事务 B 的过去中，所以事务 B 是可以看到事务 A 的修改内容的。数据库继续运行，当前事务变成了 C，A 和 B 都处于 C 的过去，所以 C 可以看到 A 和 B 的修改内容，一切正常。当前事务变成了 D 以后，因为 $D - A$ 已经大于 2^{31} ，所以 A 处于 D 的未来范围，其结果就是 D 突然看不见记录 R 了。很显然这不符合逻辑。记录 R 明明是在很早以前的修改，它静静地呆在那里，没有任何进一步的修改了，怎么突然就看不到了呢？为了解决这个问题，PostgreSQL 引入了一个新的概念：事务冻结。它的含义是这样的：对于一个事务，如果它的事务号 XID 比任何事务号都老，也就是说任何事务都可以看到这个事务所修改的数据，则该事务称为被“冻结”了。在 PostgreSQL 早期版本中，解决方法是把 A 的 t_{xmin} 变成 2，因为 2 是一个特殊的事务号，表示被冻结的事务。现在的版本是在记录 R 的 $t_{infomask}$ 域中设置一些比特位来表示该记录的事务号被冻结了。这些比特位的定义如下：

```
/* in src/include/access/htup_details.h */
#define HEAP_XMIN_COMMITTED          0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID            0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMIN_FROZEN             (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
```

现在就存在了一个问题：由谁来对一个老的事务进行冻结呢？答案是下一节要讲述的 Vacuum 操作。冻结的操作很简单，就是 vacuum 进程扫描一张表中的所有记录，如果某一条记录的 t_{xmin} 的年龄大于参数 $vacuum_freeze_min_age$ 的值（缺省是 5 千万），则设置该条记录状态为 $HEAP_XMIN_FROZEN$ ，表示这条记录的事务被冻结了。细节我们在 Vacuum 这一节中再详细讨论。

7.2.3 提交日志 (Commit Log)

当一个事务被成功提交以后，它的状态应该被保存在某一个地方。PostgreSQL 使用了一个 CLOG 结构，它把记录事务提交状态的信息保存在 PGDATA/pg_xact 目录下。

```
postgres@schedule-db00:~/9.6/main$ ls -l pg_clog/
total 49596
-rw----- 1 postgres postgres 262144 Nov  7 15:33 0000
-rw----- 1 postgres postgres 262144 Nov  8 00:43 0001
-rw----- 1 postgres postgres 262144 Nov  7 18:54 0002
-rw----- 1 postgres postgres 262144 Nov  7 21:08 0003
-rw----- 1 postgres postgres 262144 Nov  8 00:45 0004
-rw----- 1 postgres postgres 262144 Nov  8 05:22 0005
-rw----- 1 postgres postgres 262144 Nov  8 17:11 0006
-rw----- 1 postgres postgres 262144 Nov  8 13:54 0007
-rw----- 1 postgres postgres 262144 Nov  8 16:01 0008
-rw----- 1 postgres postgres 262144 Nov  9 01:05 0009
-rw----- 1 postgres postgres 262144 Nov  8 19:22 000A
-rw----- 1 postgres postgres 262144 Nov  8 21:19 000B
-rw----- 1 postgres postgres 262144 Nov  9 00:49 000C
-rw----- 1 postgres postgres 262144 Nov  9 06:52 000D
-rw----- 1 postgres postgres 262144 Nov  9 16:49 000E
-rw----- 1 postgres postgres 262144 Nov  9 14:26 000F
-rw----- 1 postgres postgres 262144 Nov 10 00:27 0010
-rw----- 1 postgres postgres 262144 Nov  9 18:29 0011
```

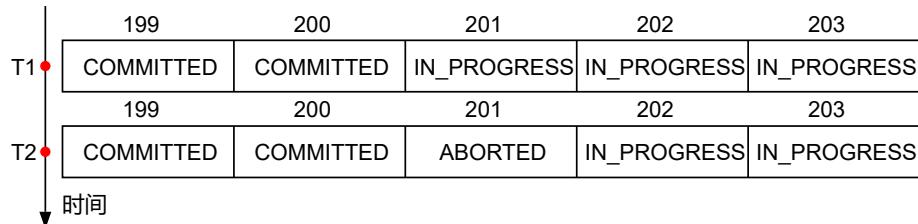


图 7.7: CLOG 中的事务状态变化

7.2.4 事务快照 (Transaction Snapshot)

<https://habr.com/en/company/postgrespro/blog/479512/>

```
/* in src/include/utils/snapshot.h */
typedef struct SnapshotData
{
    SnapshotType    snapshot_type; /* type of snapshot */
    TransactionId  xmin;          /* all XID < xmin are visible to me */
    TransactionId  xmax;          /* all XID >= xmax are invisible to me */
    TransactionId *xip;
    uint32          xcnt;          /* # of xact ids in xip[] */
    TransactionId *subxip;
    int32           subxcnt;        /* # of xact ids in subxip[] */
    bool            suboverflowed; /* has the subxip array overflowed? */
    bool            takenDuringRecovery; /* recovery-shaped snapshot? */
    bool            copied; /* false if it's a static snapshot */
    CommandId      curcid; /* in my xact, CID < curcid are visible */
    uint32          speculativeToken;
```

```

struct GlobalVisState *vistest;
uint32 active_count; /* refcount on ActiveSnapshot stack */
uint32 regd_count; /* refcount on RegisteredSnapshots */
pairingheap_node ph_node; /* link in the RegisteredSnapshots heap */
TimestampTz whenTaken; /* timestamp when snapshot was taken */
XLogRecPtr lsn; /* position in the WAL stream when taken */
uint64 snapXactCompletionCount;
} SnapshotData;

```

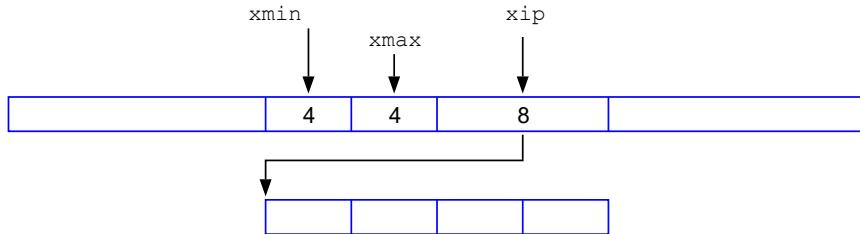


图 7.8: 事务快照

```

oracle=# SELECT pg_current_snapshot();
pg_current_snapshot
-----
744:744:
(1 row)

```

事务快照的形式是: xmin:xmax:xip_list, 譬如 100:104:100,102。xmin 的含义: 任何在 xmin 之前的事务都已经提交或者回滚了。xmax 是活跃事务中最早的事务的事务号。xmax 的含义: 任何比这个事务更晚的事务, 包括 xmax, 都没有发生。xip_list: 这是一个活跃事务列表, 包含的范围在 xmin 和 xmax 之间。

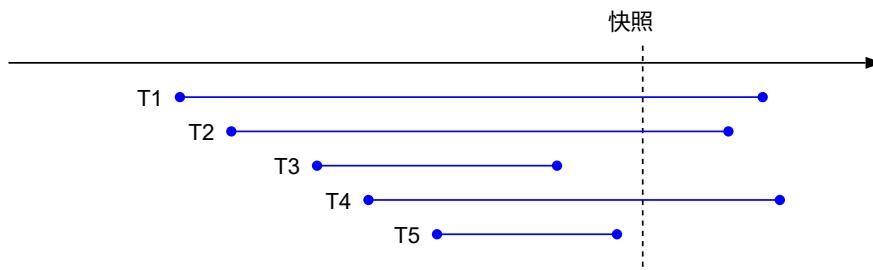


图 7.9: 事务快照

上图的快照是: T1:T5:T1,T2,T4。因为在获取快照的时刻, T3 和 T5 已经结束了。

```

/* in src/include/storage/proc.h */
struct PGPROC {
    .....
    TransactionId xid; /* id of top-level transaction currently being
                           * executed by this proc, if running and XID

```

```

        * is assigned; else InvalidTransactionId.
        * mirrored in ProcGlobal->xids[pgxactoff] */

TransactionId xmin;           /* minimal running XID as it was when we were
                               * starting our xact, excluding LAZY VACUUM:
                               * vacuum must not remove tuples deleted by
                               * xid >= xmin ! */

LocalTransactionId lxit;      /* local id of top-level transaction currently
                               * being executed by this proc, if running;
                               * else InvalidLocalTransactionId */

int                  pid;       /* Backend's process ID; 0 if prepared xact */

/*
 * latest transaction id among the transaction's main XID and
 * subtransactions
 */
TransactionId procArrayGroupMemberXid;

TransactionId clogGroupMemberXid;    /* transaction id of clog group member */
KidStatus     clogGroupMemberKidStatus;   /* transaction status of clog
                                           * group member */
int          clogGroupMemberPage;    /* clog page corresponding to
                                           * transaction id of clog group member */
};

}

```

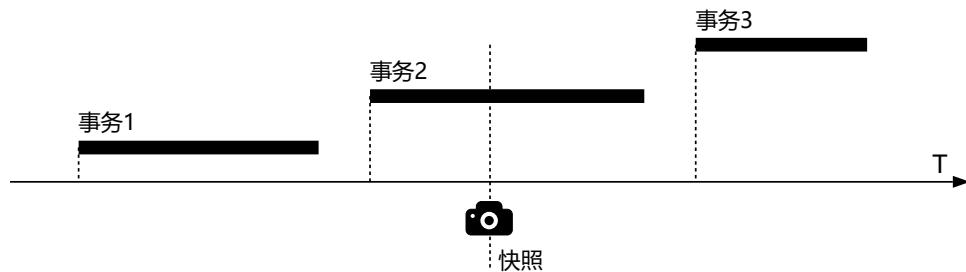


图 7.10: 事务和快照

事务 1 可见，事务 3 不可见。

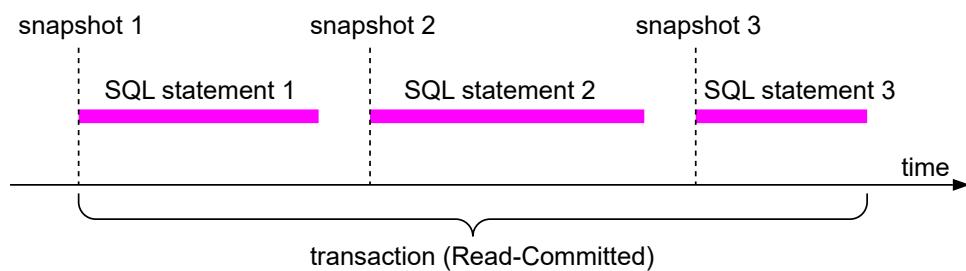


图 7.11: XXXXX

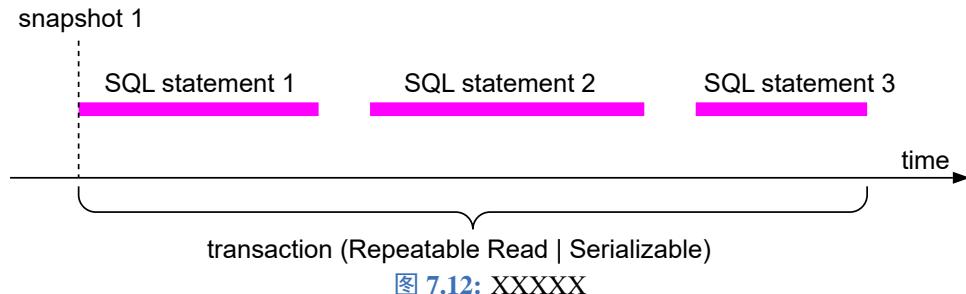


图 7.12: XXXXX

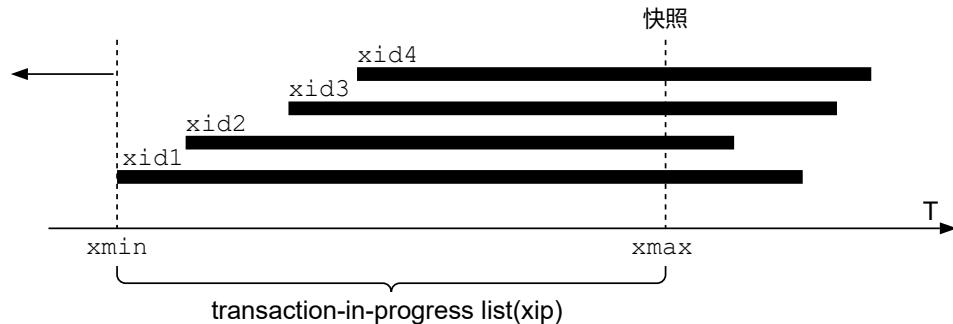


图 7.13: 快照

7.2.5 可见性 (Visibility) 规则

可见性规则是一套有每条 Tuple 的 xmin/xmax 和 Clog/事务快照一起决定的规则。

规则 0 - 任何事务都可以看到它本身的修改。

规则 1 - ABORTED 的 xmin 是不可见的。

```
IF STATUS(xmin) == 'ABORTED' THEN
    RETURN 'Invisible'
ENDIF
```

规则 2 - 一个 Tuple 的 xmin 处于 IN_PROGRESS 状态，需要如下判断：

```
IF STATUS(xmin) == 'IN_PROGRESS' THEN
    IF xmin == current_xid THEN
        IF xmax == INVALID THEN
            RETURN 'Visible'
        ELSE
            RETURN 'Invisible'
        ENDIF
    ELSE
        RETURN 'Invisible'
    ENDIF
ENDIF
```

```
/* in src/include/storage/itemid.h */
#define LP_UNUSED      0      /* unused (should always have lp_len=0) */
#define LP_NORMAL      1      /* used (should always have lp_len>0) */
#define LP_REDIRECT    2      /* HOT redirect (should have lp_len=0) */
```

```

#define LP_DEAD      3 /* dead, may or may not have storage */
typedef struct ItemIdData {
    unsigned lp_off:15, /* offset to tuple (from start of page) */
            lp_flags:2, /* state of line pointer, see below */
            lp_len:15; /* byte length of tuple */
} ItemIdData;

/* in src/include/access/htup_details.h */

#define HEAP_HASNULL          0x0001 /* has null attribute(s) */
#define HEAP_HASVARWIDTH       0x0002 /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL       0x0004 /* has external stored attribute(s) */
#define HEAP_HASOID_OLD        0x0008 /* has an object-id field */
#define HEAP_XMAX_KEYSHR_LOCK  0x0010 /* xmax is a key-shared locker */
#define HEAP_COMBOCID           0x0020 /* t_cid is a combo CID */
#define HEAP_XMAX_EXCL_LOCK    0x0040 /* xmax is exclusive locker */
#define HEAP_XMAX_LOCK_ONLY     0x0080 /* xmax, if valid, is only a locker */
#define HEAP_XMIN_COMMITED      0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID       0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITED      0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID       0x0800 /* t_xmax invalid/aborted */
#define HEAP_XMAX_IS_MULTI      0x1000 /* t_xmax is a MultiXactId */
#define HEAP_UPDATED             0x2000 /* this is UPDATED version of row */
#define HEAP_MOVED_OFF           0x4000
#define HEAP_MOVED_IN            0x8000 /* moved from another place by pre-9.0
#define HEAP_XACT_MASK           0xFFFF /* visibility-related bits */
#define HEAP_XMAX_SHR_LOCK      (HEAP_XMAX_EXCL_LOCK | HEAP_XMAX_KEYSHR_LOCK)
#define HEAP_LOCK_MASK           (HEAP_XMAX_SHR_LOCK | HEAP_XMAX_EXCL_LOCK | HEAP_XMAX_KEYSHR_LOCK)
#define HEAP_XMIN_FROZEN         (HEAP_XMIN_COMMITED|HEAP_XMIN_INVALID)
#define HEAP_MOVED               (HEAP_MOVED_OFF | HEAP_MOVED_IN)

/* in src/backend/storage/ipc/procarray.c */
bool TransactionIdIsInProgress(TransactionId xid)
/* in src/backend/access/transam/transam.c */
bool TransactionIdDidCommit(TransactionId transactionId)

bool TransactionIdDidAbort(TransactionId transactionId)

/* in src/include/access/htup_details.h */
#define HEAP_XMIN_COMMITED      0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID        0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITED      0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID        0x0800 /* t_xmax invalid/aborted */

```

7.2.6 pg_resetwal 的使用

```

$ pg_resetwal --help
pg_resetwal resets the PostgreSQL write-ahead log.

```

Usage:

```
pg_resetwal [OPTION]... DATADIR
```

Options:

-c, --commit-timestamp-ids=XID,XID	set oldest and newest transactions bearing commit timestamp (zero means no change)
[-D, --pgdata=]DATADIR	data directory
-e, --epoch=XIDEPOCH	set next transaction ID epoch
-f, --force	force update to be done
-l, --next-wal-file=WALFILE	set minimum starting location for new WAL
-m, --multixact-ids=MXID,MXID	set next and oldest multitransaction ID
-n, --dry-run	no update, just show what would be done
-o, --next-oid=OID	set next OID
-O, --multixact-offset=OFFSET	set next multitransaction offset
-u, --oldest-transaction-id=XID	set oldest transaction ID
-V, --version	output version information, then exit
-x, --next-transaction-id=XID	set next transaction ID
--wal-segsize=SIZE	size of WAL segments, in megabytes
-?, --help	show this help, then exit

Report bugs to <pgsql-bugs@lists.postgresql.org>.

PostgreSQL home page: <<https://www.postgresql.org/>>

7.2.7 HOT

HOT 是 Heap Only Tuples 的缩写，它是一种优化技术。我们通过一个例子来展示它的基本概念。具体的实验过程如下：

```
oracle=# create table hot(id int primary key, name char(1));
CREATE TABLE
oracle=# insert into hot values(1000, 'A');
INSERT 0 1
oracle=# update hot set name='B' where id=1000;
UPDATE 1
```

上面的操作很简单，就是创建了一张表，往里面插入了一条记录，然后对这条记录进行了更新。因为这张表有主键，就是第一列 id，所以它有一个 B 树索引，我们可以看到这个索引的名称叫 hot_pkey。

```
oracle=# \d+ hot
                                         Table "public.hot"
   Column |      Type       | Collation | Nullable | Default | Storage  | Compression | Stats target |
Description
-----+-----+-----+-----+-----+-----+-----+
    id   | integer        |           | not null |          | plain     |            |
   name  | character(1)  |           |           |          | extended  |            |
Indexes:
  "hot_pkey" PRIMARY KEY, btree (id)
Access method: heap
```

因为这张表极小，所以数据文件只有一个数据块，编号为 0，而且索引有两个数据块，第一个是元数据块，编号为 0；第二个数据块是根数据块，也是叶子数据块，编号为 1。我们做如下查询：

```
oracle=# select lp, lp_flags, lp_off, lp_len,t_xmin, t_xmax, t_ctid, t_infomask2
oracle# from heap_page_items(get_raw_page('hot',0));
lp | lp_flags | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_infomask2
-----+-----+-----+-----+-----+-----+-----+
1 | 1 | 8160 | 30 | 759 | 760 | (0,2) | 16386
2 | 1 | 8128 | 30 | 760 | 0 | (0,2) | 32770
(2 rows)

oracle=# SELECT itemoffset,ctid,itemlen,data,dead FROM bt_page_items('hot_pkey',1);
itemoffset | ctid | itemlen | data | dead
-----+-----+-----+-----+-----+
1 | (0,1) | 16 | e8 03 00 00 00 00 00 00 | f
(1 row)
```

从上面的结果可以很清楚地看到，表的数据块中有两条记录。因为 UPDATE 操作是先删后插，第一条是死亡记录，第二条是正常记录。同时我们也注意到了，索引上只有一条记录，指向了第一条死亡记录，并不是指向了第二条记录。第一条记录的 t_ctid 这一列指向了第二条记录。第一条记录的 t_infomask2 的值是 16386，就是十六进制的 0x4002。第二条记录的 t_infomask2 的值是 32770，就是十六进制的 0x8002。下面我们看如下的定义：

```
/* in src/include/access/htup_details.h */
#define HEAP_HOT_UPDATED          0x4000 /* tuple was HOT-updated */
#define HEAP_ONLY_TUPLE            0x8000 /* this is heap-only tuple */
```

所以第一条记录的 HEAP_HOT_UPDATED 位为 1，而第二条记录的 HEAP_ONLY_TUPLE 为 1。我们可以用图 XX 来表示上述的实验结果：

	t_xmin	t_xmax	t_ctid	t_infomask2	用户的数据
记录1	195		(5,1)		1000, 'A'

	t_xmin	t_xmax	t_ctid	t_infomask2	用户的数据
记录1	195	196	(5,2)	HEAP_HOT_UPDATED	1000, 'A'
记录2	196		(5,2)	HEAP_ONLY_TUPLE	1000, 'B'

图 7.14: HOT 的基本概念

整个实验只进行了一次 UPDATE 操作，但是这个 UPDATE 操作有一些特点：第一个特点是它只修改了没有索引的列 name，并没有修改索引的列 id。第二个特点是 UPDATE 先删后插的时候，在被删除的记录所在的数据页上还有足够的空间来保存插入的新记录，所以两条记录在同一个页面上。因为避免修改索引上的记录，HOT 技术就是保持索引的内容不变，而是通过一个链条 (chain) 把同一条记录的不同版本串联了起来。如图 XX 所示：

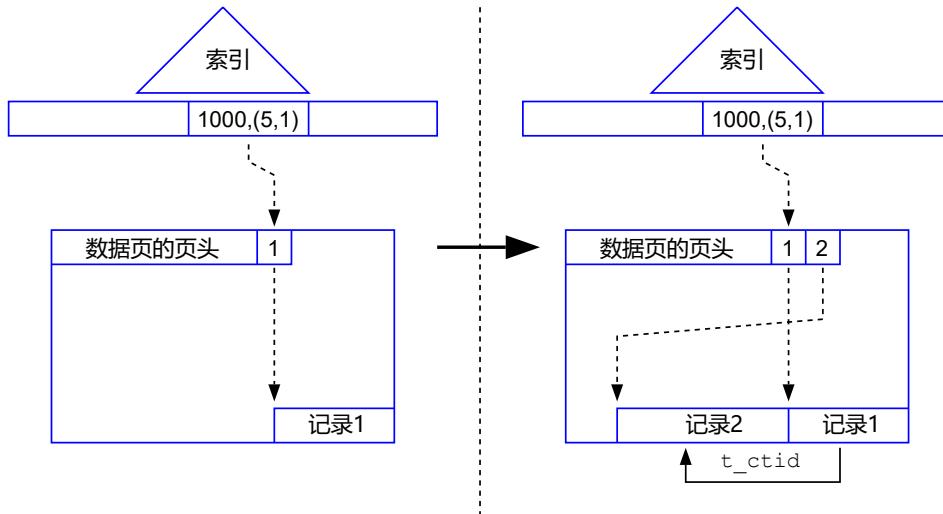


图 7.15: HOT 的链条

当某一个查询访问索引时，它拿到的是指向第一条记录的指针，所以它首先找到了第一条记录，它进一步判断这条记录的 `t_infomask2` 的 `HEAP_HOT_UPDATED` 是否为 1，如果是 1，它就知道这并不是最终版本的记录，它就可以通过这条记录的 `t_ctid` 这一列顺藤摸瓜，找到了第二条记录。因为第二条记录的 `HEAP_ONLY_TUPLE` 位是 1，所以这是该查询所需要的最终版本的记录。如果一次 `UPDATE` 操作修改的列上没有任何索引，而且要插入的新记录可以存放在和老记录同一个数据页上，就会出现上述的现象，即索引并没有被修改，只是通过一个链条把不同版本的记录串联起来，这种技术就叫做 HOT 技术。因为 `t_ctid` 是六个字节，包含 4 个字节的数据块的编号和 2 个字节的偏移量，理论上第二个要求不是必须的，因为可以通过数据块的编号信息找到另外的数据块。但是从性能的角度考虑，HOT 被限制在同一个数据页上，这一点要特别注意。

过了一段时间，死亡的第一条记录可能被重复使用，为了避免 `t_ctid` 这个的中断，PostgreSQL 就会执行数据页的修剪 (page pruning) 的动作。经过数据页修剪后，数据页变成了如图 XX 所示的布局：

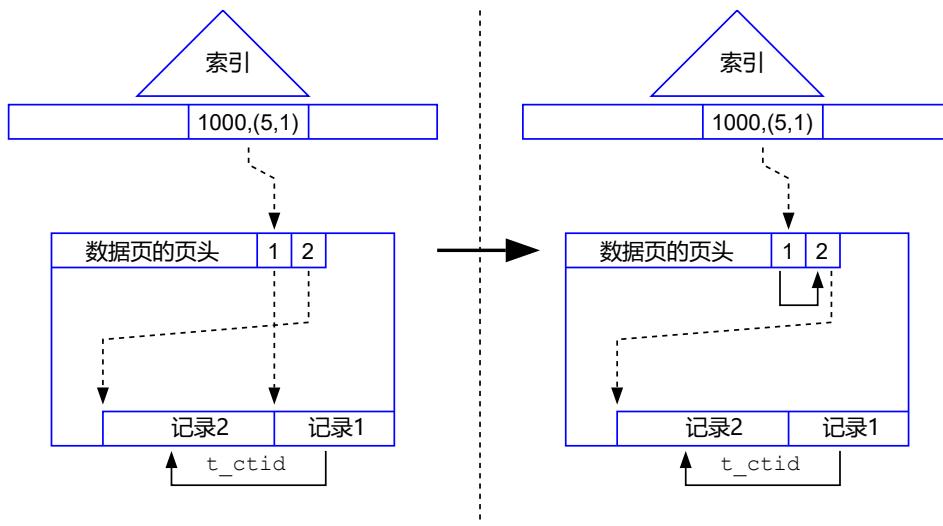


图 7.16: 记录的重定向

我们很清楚地观察到，第一条记录的指针指向了第二条记录。这样即使第一条记录被覆盖，这个 HOT 链条依然有效。在满足一定条件的情况下，`SELECT` 操作也可能触发数据页的修剪操作，从而让该数据页变成脏页，也会产生相应的 `WAL` 记录来保护这种修剪的修改动作。我们下面使用 `Vacuum` 操作来触发修剪动作。关于 `Vacuum` 的具体内容，后文中会进行详细介绍。我们执行如下动作：

```

oracle=# vacuum hot;
VACUUM
oracle# select lp, lp_flags, lp_off, lp_len,t_xmin, t_xmax, t_ctid, t_infomask2 from heap_page_items(
  get_raw_page('hot',0));
lp | lp_flags | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_infomask2
---+-----+-----+-----+-----+-----+-----+
 1 |      2 |      2 |      0 |       |       |       |
 2 |      1 |    8160 |      30 |     755 |       0 | (0,2) |     32770
(2 rows)

```

对比前面的实验结果，我们可以观察到一些变化：第一条记录的 lp_flags 从 1 变成了 2，lp_off 从 8160 变成了 2，从它的长度 lp_len 从 30 变成了 0。我们可以参考如下定义：

```

/* in src/include/storage/itemid.h */
#define LP_UNUSED    0 /* unused (should always have lp_len=0) */
#define LP_NORMAL    1 /* used (should always have lp_len>0) */
#define LP_REDIRECT   2 /* HOT redirect (should have lp_len=0) */
#define LP_DEAD      3 /* dead, may or may not have storage */

```

由上面的定义可知，第一条记录变成了 LP_REDIRECT 类型的指针，它的 lp_off 为 2，表示它直接指向了第二条记录。如果我们继续修改这条记录：

```

oracle# update hot set name='C' where id=1000;
UPDATE 1
oracle# vacuum;
VACUUM
oracle# select lp, lp_flags, lp_off, lp_len,t_xmin, t_xmax, t_ctid, t_infomask2 from heap_page_items(
  get_raw_page('hot',0));
lp | lp_flags | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_infomask2
---+-----+-----+-----+-----+-----+-----+
 1 |      2 |      3 |      0 |       |       |       |
 2 |      0 |      0 |      0 |       |       |       |
 3 |      1 |    8160 |      30 |     761 |       0 | (0,3) |     32770
(3 rows)

```

我们可以看到，第一条记录的指针依然是 LP_REDIRECT 类型，它的 lp_off 的值为 3，表明它指向了第三条记录，即最后版本的那条记录。整个修改过程如图 XX 所示：

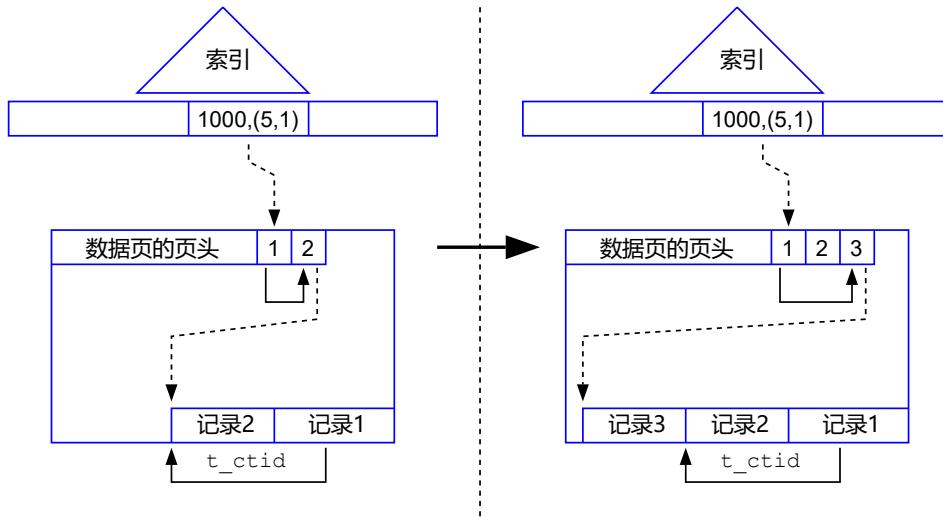


图 7.17: 多个版本的记录的 HOT 链条

由此可见，为了缩短 HOT 链条的长度，第一条 LP_REDIRECT 类型的指针会一竿子到底，指向了最终版本的那条记录，这样就保证了链条上最多有两条记录，缩短了遍历链条的时间。

第一条记录死亡以后，它所占用的空间就变成了空闲空间，可以被重复利用。但是该数据页的空闲空间也出现了碎片化的现象，因为在第二条记录的前面和后面都有空闲空间。为了把空闲空间合并成一个大块，方便未来的空间分配，在做页面修剪的同时，也会对数据页进行碎片化的整理，被称为 defragmentation。它的基本概念如图 XX 所示。

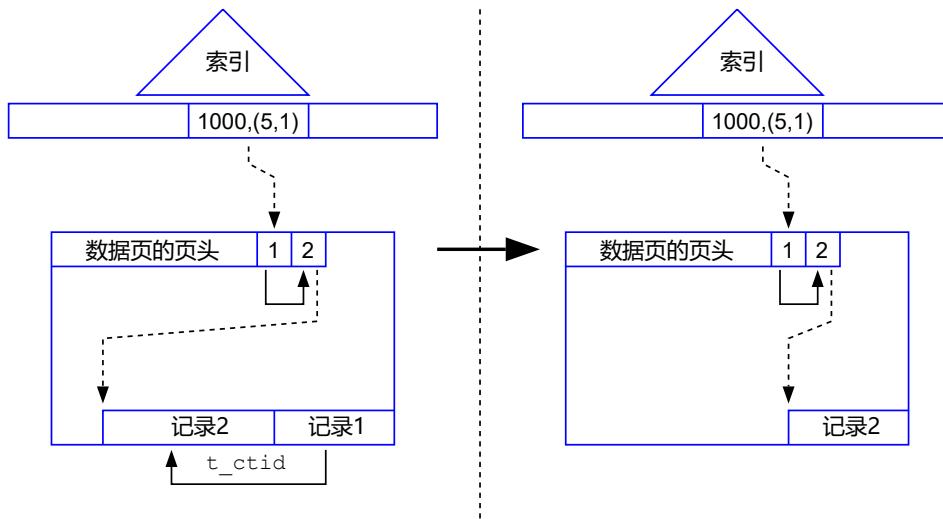


图 7.18: 碎片化整理的概念

Vacuum 操作会执行数据页的修剪和碎片化的整理工作。大家可以利用 pageinspect 这个插件的函数查阅一下空闲空间的变化情况，应该非常容易，这里我们就不详细展示了。

7.2.8 数据页的修剪

Page running

7.2.9 事务的冻结

我们知道事务号是一个 32 比特的无符号整型值，它的最大值大概 42 亿个，等事务号变成最大值以后，再增加 1 就归零了，所以我们可以把事务号的空间想象成一个无限循环的环状。我们以当前事务的角度，把这个环均匀的分为两半，每部分大约 21 亿个数值，前半部分是过去已经发生的事务，后半部分是未来尚未发生的事。我们看一下图 XX 所展示的事务不断增加的过程。

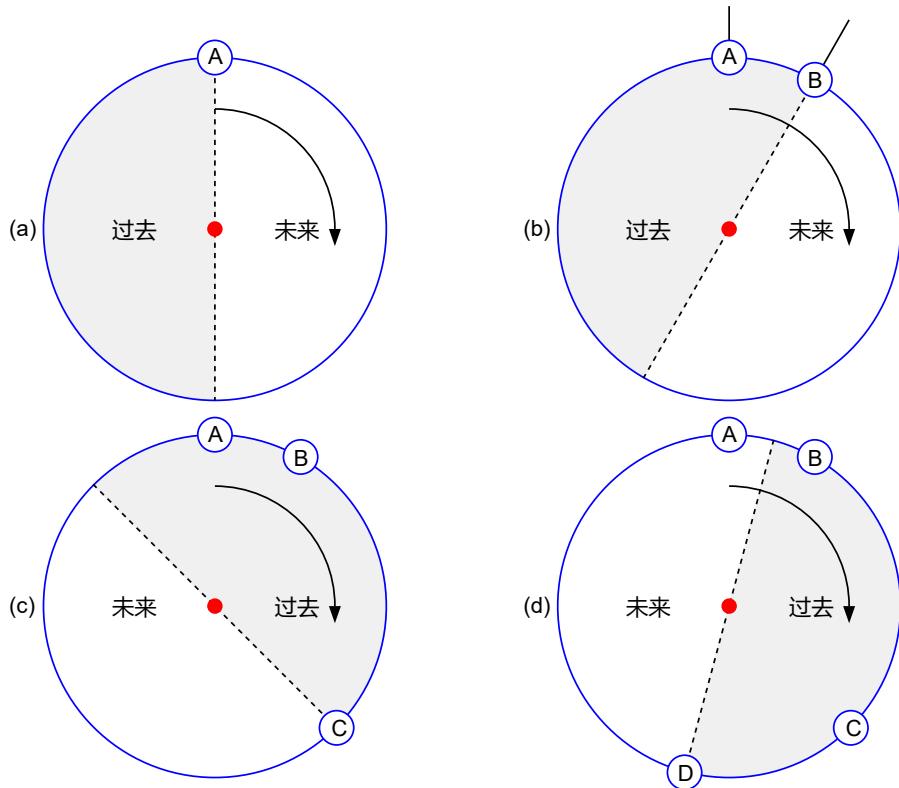


图 7.19: 事务的环绕问题

假设某个事务 A 在 A 点插入了一条记录 R，则这条记录的 t_{xmin} 的事务号为 A。数据库继续运行，当前事务号为 B。此时事务 A 处于事务 B 的过去中，所以事务 B 是可以看到事务 A 的修改内容的。数据库继续运行，当前事务变成了 C，A 和 B 都处于 C 的过去，所以 C 可以看到 A 和 B 的修改内容，一切正常。当前事务变成了 D 以后，因为 $D - A$ 已经大于 2^{31} ，所以 A 处于 D 的未来范围，其结果就是 D 突然看不见记录 R 了。很显然这不符合逻辑。记录 R 明明是在很早以前的修改，它静静地呆在那里，没有任何进一步的修改了，怎么突然就看不到了呢？这种现象被称为事务的环绕 (transaction wraparound) 问题。为了解决这个不合理的问题，PostgreSQL 引入了一个新的概念：事务冻结 (freeze)。它的含义是这样的：对于一个事务 A，我们设置它的事务号从 X 修改为特殊的事务号 2，表示它的事务号比任何事务号都老，任何事务都可以看到这个事务所修改的数据，则该事务称为被“冻结”了。在 PostgreSQL 早期版本中，解决方法就是把 A 的 t_{xmin} 变成 2，表示被冻结的事务。现在的版本是在记录 R 的 $t_{infomask}$ 域中设置一些比特位来表示该记录的事务号被冻结了。这些比特位的定义如下：

```
/* in src/include/access/htup_details.h */
#define HEAP_XMIN_COMMITTED          0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID             0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMIN_FROZEN              (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
```

现在就存在了两个问题：什么时候冻结某一个事务，以及由谁来对一个老的事务进行冻结。对于第一个问题，我们首先引入一个概念：事务的年龄。我们都很容易理解出生日期和年龄这两个不同但是紧密联系的概念，

譬如：小明的出生日期是 1995 年 5 月 18 日，假设今天是 2024 年 5 月 18 日，他的年龄就是 29 岁，即年龄等于当前时间减去他的出生时间。事务号可以理解为一个事务的出生日期。类似人类的年龄，事务的年龄就是当前事务的 XID 减去该事务的 XID。假设当前事务的 XID 是 100，则事务号 99 的年龄为 1(等于 100 - 99)，事务号 81 的年龄是 19(等于 100 - 81)。由于当前事务的 XID 是不断增加的，所以对于给定的事务号 xid，它的年龄也是随着时间的流逝，更准确的说法是随着新事务的产生而不断增加。我们可以通过 age() 这个函数计算一个事务号的年龄：

```
oracle=# select 100::text as xid, age(100::text::xid), pg_current_xact_id() as current_xid;
  xid | age | current_xid
-----+----+-----
  100 | 658 |      758
(1 row)

oracle=# select 100::text as xid, age(100::text::xid), pg_current_xact_id() as current_xid;
  xid | age | current_xid
-----+----+-----
  100 | 659 |      759
(1 row)

oracle=# select age(0::text::xid) as age0, age(1::text::xid) as age1, age(2::text::xid) as age2;
   age0  |   age1  |   age2
-----+-----+-----
 2147483647 | 2147483647 | 2147483647
(1 row)
```

我们可以看到，对于一个正常事务号，即不是 0、1、2 这三个值，其年龄是当前事务号和它的差值。对于 3 个特殊的事务号，它们的年龄恒定为 2147483647，即 $2^{31}-1$ 。关于计算某一个事务号的年龄的函数代码如下，从代码中我们可以很清楚地看到它的逻辑，不难理解。

```
/* in src/backend/utils/adt/xid.c */
Datum xid_age(PG_FUNCTION_ARGS)
{
    TransactionId xid = PG_GETARG_TRANSACTIONID(0);
    TransactionId now = GetStableLatestTransactionId();
    /* Permanent XIDs are always infinitely old */
    if (!TransactionIdIsNormal(xid)) PG_RETURN_INT32(INT_MAX);
    PG_RETURN_INT32((int32) (now - xid));
}
```

理解了事务的年龄以后，我们就可以学习在什么条件下冻结事务。PostgreSQL 冻结事务的逻辑非常简单：如果某一条记录的 t_xmin 的年龄大于某一个规定的值 X，则就冻结该记录的事务。这个 X 就是由参数 vacuum_freeze_min_age 来规定，缺省值是 5 千万。也就是说，如果某一条记录的 t_xmin 比当前事务号的距离大于 5 千万，就冻结这条记录的事务号，即设置它的 t_infomask 域的两个比特为 1，具体细节由 HEAP_XMIN_FROZEN 规定。别的事务在扫描到这条记录时，如果发现该记录的事务号已经被冻结了，就知道这条记录对自己是可见的。我们可以参考图 XX 来理解事务的冻结问题。假设当前事务是 50000101，一张表中有四条记录，如图 XX 左边所示：

The diagram illustrates the state transition of a table row during a transaction freeze. It consists of two tables side-by-side, connected by a horizontal arrow pointing from left to right.

事务冻结前 (Before Freeze):

t_xmin	t_infomask	data
99		AA
100		BB
1500000		CC
2000000		DD

事务冻结后 (After Freeze):

t_xmin	t_infomask	data
99	HEAP_XMIN_FROZEN	AA
100	HEAP_XMIN_FROZEN	BB
1500000		CC
2000000		DD

图 7.20: 事务的冻结

假设 vacuum_freeze_min_age 使用缺省值 5 千万，很显然，第一条记录的事务年龄是 50000002，第二条记录的事务年龄是 50000001，它们的年龄均超过了参数 vacuum_freeze_min_age 规定的值，所以头两条记录会被冻结。第三条和第四条记录的年龄还没有超过 5 千万，所以它们不会被冻结。事务冻结后的示意图如上图右边所示。第一条和第二条记录的 t_infomask 中的 HEAP_XMIN_FROZEN 被置位 (两个比特都是 1)，表示这两条记录对任何活跃或者未来的事务来说，都是可见的。

数据库里面的记录多如牛毛，总得有一个进程要依次扫描所有表的所有记录，一条条检查，确保该冻结事务的记录得到及时的冻结。这个进程是什么呢？答案是下一节要讲述的 Vacuum 操作。冻结的操作很简单，就是 vacuum 进程扫描一张表中的所有记录，如果发现某条记录符合冻结条件，就冻结它，其中的细节我们在 Vacuum 这一节中再详细讨论。

第八章 表和索引的清理 (VACUUM)

根据前一章讨论的 MVCC 模型，我们知道被删除的记录并不会真正从数据块上真正删除。那么时间长了，必然在数据文件上积累了大量死亡的记录。这些死亡的记录如果不被清理，会大大影响数据库的性能。举一个实际的例子，我负责维护的数据库里有一张表，真正的体积，加上相关的索引，只有 1.3TB。因为这张表上每天都有几百万次的修改和删除操作，如果不加以控制，不到一个月，它的体积就会膨胀到 10TB，和这张表相关的查询的性能都普遍下降。所以我们需要对数据库定期做一些维护工作，其中对表和索引上的死亡记录进行清理是最重要的维护内容之一，这种维护操作在 PostgreSQL 领域有一个术语，叫做 VACUUM，这个单词的本意是“清理”意思，譬如吸尘器的英文单词就叫做 vacuum cleaner。因为清理这个中文词语很普通，很容易淹没在字里行间，所以我们使用 VACUUM 这个单词，不进行翻译。本章探讨 VACUUM 的知识。

8.1 一个 VACUUM 的实验

我们先通过一个小实验对 VACUUM 有一个初步认识。这个实验需要安装 pageinspect 插件，具体做法请参考第二章的相关内容。首先创建一个测试表和一个索引，并插入若干记录：

```
oracle=# create table v(id char(2) primary key, name varchar(8));
CREATE TABLE
oracle=# insert into v values ('AA', 'aaaa'), ('BB', 'bbbb'), ('CC', 'cccc'), ('DD', 'dddd');
INSERT 0 4
oracle=# \d v
      Table "public.v"
Column |          Type          | Collation | Nullable | Default
-----+----------------+-----+-----+-----+
id    | character(2)        |           | not null |
name  | character varying(8) |           |           |
Indexes:
"v_pkey" PRIMARY KEY, btree (id)
```

因为列 id 是主键，所以 PostgreSQL 自动创建了一个 B 树索引 v_pkey。很显然，插入的四条记录都很短，一个数据块保存它们绰绰有余，所以这张表目前只有一个数据块，即 0 号数据块。我们使用 pageinspect 插件提供的函数查看一下 0 号数据块的页头信息，衡量一下这个数据块的空闲空间。

```
oracle=# SELECT * FROM page_header(get_raw_page('v',0));
 lsn   | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----+
 0/18AC788 |       0 |     0 |     40 |  8064 |     8192 |     8192 |       4 |         0
(1 row)
```

根据第二章学习的知识，我们知道一个数据块的 upper 指针减去 lower 指针就是该数据块的空闲空间，所以 0 号数据块的空闲空间为 8024 字节 (=8064-40)。现在我们删除第二条记录和第四条记录，并且执行检查点，确保所有的数据都落盘了。为什么选择删除第二条和第四条记录呢？因为插入时是按照顺序插入的，第二条记录在中间，而第四条记录在尾部。我们想对比这两种情景的变化有什么不同。

```
oracle=# delete from v where id in ('BB', 'DD'); /* 删除第二条和第四条记录 */
DELETE 2
oracle=# checkpoint; /* 手工执行检查点，确保所有的数据都写入磁盘中 */
CHECKPOINT
```

```
oracle=# select lp,lp_len,t_xmin,t_xmax,t_ctid from heap_page_items(get_raw_page('v',0));
lp | lp_len | t_xmin | t_xmax | t_ctid
---+-----+-----+-----+
1 | 32 | 742 | 0 | (0,1)
2 | 32 | 742 | 744 | (0,2)
3 | 32 | 742 | 0 | (0,3)
4 | 32 | 742 | 744 | (0,4)
(4 rows)
```

从上面的输出结果我们可以看到，被删除的记录并不会真正从磁盘上抹去，除非是它们的 t_xmax 变成了非 0 值，就是删除它们的事务的事务号。我们现在观察一下这个数据块的空闲空间有没有变化。我们预期它没有任何变化。

```
oracle=# SELECT * FROM page_header(get_raw_page('v',0));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+
0/19055B0 | 0 | 0 | 40 | 8064 | 8192 | 8192 | 4 | 744
(1 row)
```

对比前面的结果，我们看到了，果然这个数据块的空闲空间没有任何变化。这种死亡记录白白占用磁盘空间的问题，是 VACUUM 要解决的主要问题。然后我们继续使用 pageinspect 插件考察一下索引里面的内容：

```
oracle=# select magic, version, root, level from bt_metap('v_pkey');
magic | version | root | level
-----+-----+-----+
340322 | 4 | 1 | 0 /* <-- 根节点是1号数据块，它的层数是0，是叶子节点 */
(1 row)

oracle=# select itemoffset,ctid,itemlen,data,dead from bt_page_items('v_pkey',1);
itemoffset | ctid | itemlen | data | dead
-----+-----+-----+-----+
1 | (0,1) | 16 | 07 41 41 00 00 00 00 00 | f
2 | (0,2) | 16 | 07 42 42 00 00 00 00 00 | f /* <-- BB */
3 | (0,3) | 16 | 07 43 43 00 00 00 00 00 | f
4 | (0,4) | 16 | 07 44 44 00 00 00 00 00 | f /* <-- DD */
(4 rows)
```

从上面的结果可知，索引的根节点是 1 号数据块，根节点也是叶子节点，因为它的层数 (level) 是 0，所以这个索引的全部记录都保存在 1 号数据块。我们也清楚地看到：虽然表里的第二条和第四记录被删除了，但是索引中的相应记录并没有被删除掉。索引数据中的 42 是'B' 的 ASCII 码值，44 是'D' 的 ASCII 码值。根据 data 这一列的值，我们看到第二条记录包含了'BB'，它的 TID 是 (0,2)，很显然指向了表中的已经被删除的第二条记录。它的 dead 这一列的值为 f，表示它是正常记录，但是它指向的表中的记录已经死亡了。所以如果某一个 SQL 查询根据这个貌似正常的指针去堆表中查找，可能会白忙活一趟，这就降低了查询的性能。由此我们可以得出一个结论：当表中的记录被删除掉以后，与之关联的索引的记录并没有被删除掉，这种想象会降低数据库的查询性能。所以在 VACUUM 时，不仅仅要清理表中的死亡记录，也要清理索引中的对应记录。图 8.1 表示 vacuum 之前的索引和表的数据块的基本布局：

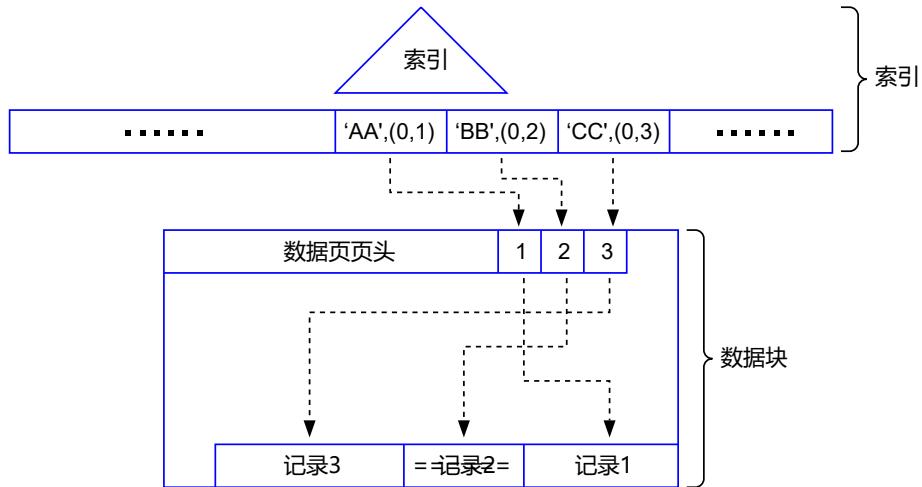


图 8.1: VACUUM 之前的数据页

在上图中，第二条和第四条记录已经被删除掉了，但是索引中依然存在指向第二条和第四条记录的索引记录。现在我们对这张表执行 vacuum，操作非常简单，如下所示：

```
oracle=# vacuum verbose v; /* 手动对表进行vacuum操作，verbose表示显示详细的输出结果 */
INFO: vacuuming "oracle.public.v"
INFO: finished vacuuming "oracle.public.v": index scans: 1
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 2 removed, 2 remain, 0 are dead but not yet removable
removable cutoff: 745, which was 0 XIDs old when operation ended
new relfrozenxid: 745, which is 14 XIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 2 tuples frozen
index scan needed: 1 pages from table (100.00% of total) had 2 dead item identifiers removed
index "v_pkey": pages: 2 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 0.000 MB/s, avg write rate: 120.192 MB/s
buffer usage: 11 hits, 0 misses, 7 dirtied
WAL usage: 7 records, 4 full page images, 9929 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

手动清理表的命令非常简单，就是 VACUUM 后面加上表的名字即可。verbose 选项是打印出丰富的输出信息，当然目前我们还看不懂。经过 VACUUM 操作以后，我们再次使用 pageinspect 插件来检查表的数据块内部的变化情况。首先看一下 0 号数据块的页头信息：

```
oracle=# select * from page_header(get_raw_page('v',0));
lsn      | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----+
0/1910B68 |        0 |      5 |     36 |   8128 |    8192 |     8192 |       4 |        0
(1 row)
```

我们看到了，这个数据块的空闲空间为 8092 字节 (= 8128 - 36)，相比较 VACUUM 之前的 8024 字节，增大了 68 个字节。这说明 VACUUM 的确释放了一部分空闲空间。根据 lower 的值为 36，我们推知这个数据块上还有 3 条记录。因为第四条记录在尾部，就被砍掉了。但是第二条记录在中间，没法砍掉，就保留下来了。从下面的输出中也可以验证这一点：

```
oracle=# select lp,lp_len,t_xmin,t_xmax,t_ctid from heap_page_items(get_raw_page('v',0));
lp | lp_len | t_xmin | t_xmax | t_ctid
---+-----+-----+-----+
1 |     32 |     742 |       0 | (0,1)
2 |       0 |         |       |
3 |     32 |     742 |       0 | (0,3)
(3 rows)
```

从上面的输出中我们可以看到，第四条记录因为在尾部，所以清理起来非常容易。但是第二条在中间，就保留下来了。为什么是这样呢？后面我们就知道原因了。下面我们考察一下索引的变化情况。执行如下操作：

```
oracle=# select itemoffset,ctid,itemlen,data,dead from bt_page_items('v_pkey',1);
itemoffset | ctid | itemlen | data | dead
---+-----+-----+-----+
1 | (0,1) |      16 | 07 41 41 00 00 00 00 00 | f
2 | (0,3) |      16 | 07 43 43 00 00 00 00 00 | f
(2 rows)
```

由上可知，索引中的死亡记录已经被彻底清除了，虽然第二条记录夹在中间，但是依然被清除了，这一点和对表的清除不一样，索引变得比表本身更干净。图 8.2 可以表示 VACUUM 之后数据块的布局。对比它和图 8.1 的差异，我们很容易了解 VACUUM 大致做了什么工作。

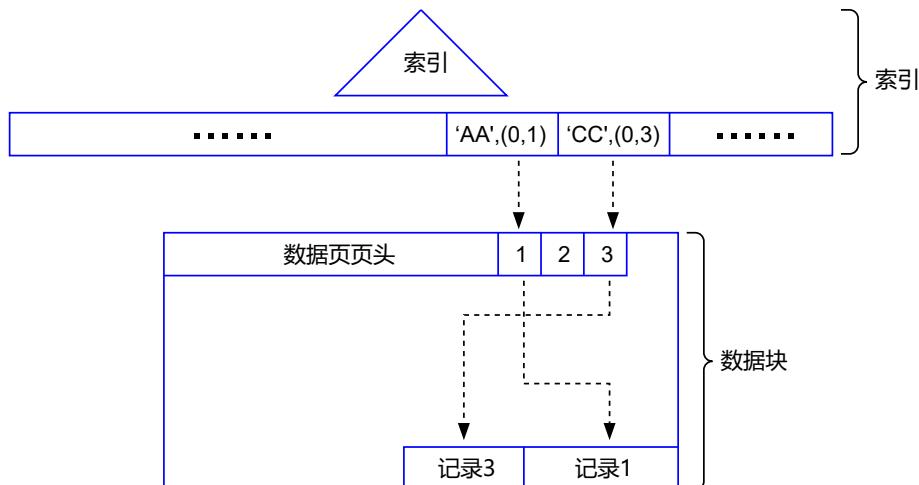


图 8.2: VACUUM 之后的数据页

由此我们可以看出，VACUUM 操作同时清空了堆表和相关索引中的死亡记录，并且对表的零碎的空闲空间进行了整理，把所有的非死亡记录都堆积在一个连续的区域，空闲区域也是连续的，提高了后面空闲空间查找和分配的性能。为什么表不能像索引一样把记录指针数组中夹在中间的第二条记录也清空呢？这个原因也很容易理解：如果把第三条记录移动到第二条记录的位置来填补空缺。它就变成了第二条记录，索引中相关的记录也要修改，这种操作的代价很高。所以第二条记录的空缺继续保留，这样的话索引就不需要被修改了。

通过上述实验，我们对 VACUUM 有了一个基本认识。大体说来，VACUUM 包含的工作有如下一些内容：

- 清理死亡记录，包括清理表和索引中的死亡记录，并且把碎片化的数据块进行整理，变得更加规整。
- 根据一定的规则冻结老的事务号，更新相关的系统表，如 pg_database 和 pg_class，并且清理不需要的 clog 文件。
- 更新 FSM 和 VM 文件，使之保持最新的状态，必要时更新数据的统计信息，如每张表有多少条记录等等。

下面我们讲解这些清理工作的相关知识。

8.2 VM 文件

VACUUM 是一种非常耗时的操作，为了提高 VACUUM 的执行效率，PostgreSQL 引入了 VM 文件。VM 是 visibility map 的缩写，表示可见性。我们查看一下上一节中创建的表 v 对应的 VM 文件。我们首先通过 pg_relation_filepath() 这个函数拿到堆表文件的名字，然后查找以它开头的所有文件：

```
oracle=# SELECT pg_relation_filepath('v');
pg_relation_filepath
-----
base/16384/16385
(1 row)

oracle=# ! ls -l $PGDATA/base/16384/16385*
-rw----- 1 postgres postgres 8192 Jan 28 11:31 /opt/data/pgdata1/base/16384/16385
-rw----- 1 postgres postgres 24576 Jan 28 11:31 /opt/data/pgdata1/base/16384/16385_fsm
-rw----- 1 postgres postgres 8192 Jan 28 11:31 /opt/data/pgdata1/base/16384/16385_vm /* <-- 这个就是 VM 文件 */
```

从上我们看到了，表 v 在磁盘上的文件名叫做 16385，它对应的 VM 文件就是 16385_vm。VM 文件的基本思路是用 2 个比特表示堆表中的一个数据块。其中第一个比特如果为 1，则表示对应的数据块中没有任何死亡记录，所以 VACUUM 操作就可以跳过这个数据块。如果第二个比特为 1，则表示这个数据块没有记录需要冻结事务号，VACUUM 同样也可以跳过这个数据块，所以 VM 文件就是 VACUUM 操作时参考的一个字典，依据 VM 文件中的信息，VACUUM 操作决定是否检查某个数据块。这两个比特的定义如下：

```
/* in src/include/access/visibilitymapdefs.h */
#define VISIBILITYMAP_ALL_VISIBLE 0x01
#define VISIBILITYMAP_ALL_FROZEN 0x02
```

这里需要注意的两个比特的含义：如果该比特的值为 1，则表示某事是百分百存在；如果该比特的值为 0，则表示某事是可能存在的。第一个比特的值是 1，则它对应的表的数据块里面肯定没有死亡记录，但是如果它的值为 0，则表示对应的数据块里可能有死亡记录，也可能没有。第二个比特的值是 1，则它对应的表的数据块里所有记录的事务号都被冻结了，但是如果它的值为 0，则表示对应的数据块里可能有记录的事务号没有被冻结，也可能所有的记录的事务号都被冻结了。那么谁来把比特的值从 1 变成 0，又是谁把值从 0 变成 1 呢？图 8.3 展示比特值变化的情况：

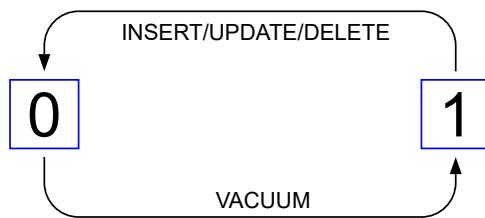


图 8.3: VM 文件中比特值的变化

由上图可知，当在一个数据块中执行插入/修改/删除操作时，就可能打破这个数据块里百分百没有死亡记录，或者所有记录的事务都已经冻结的结论。在这个时候，VM 文件中对应的比特位就变成了 0。在 VACUUM 清理完一个数据块中的所有死亡记录，或者冻结这个数据块中所有的记录的事务号以后，就可以放心地把 VM 文件中对应的比特位的值变为 1。2 位比特可以表示 4 中可能性，但是如果一个数据块中所有的记录的事务都被冻结

的话，则表示这个数据块上所有的记录都是可见的，不存在死亡记录，所以 10 这种状态是不可能的。最终 2 位比特表示三种有效的状态，它们之间的状态转换可以用图 8.4 来表示：

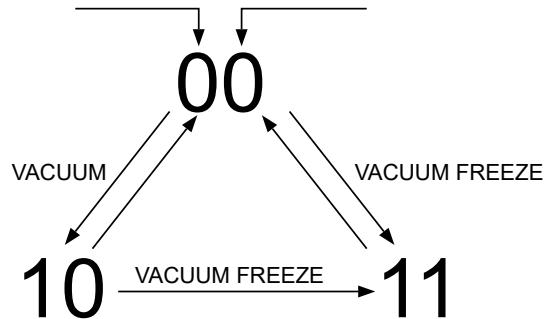


图 8.4: VM 比特的状态转换

由上图可知，我们可以通过 VACUUM 把状态变成 10。VACUUM 命令有一个选项叫做 FREEZE，表示在进行 VACUUM 操作的时候同时执行事务冻结操作。它可以让两个比特的状态变成 11。VACUUM 命令还有其它选项，譬如 ANALYZE 选项指定在 VACUUM 操作的时候也要更新表的统计信息，让统计信息更加准确，提高了查询优化器的决策正确性和最终查询的性能。

VM 文件和堆表文件的对应关系可以用图 8.5 表示。我们可以把 VM 文件理解为一个数组，其中每 2 个比特对应表中的一个数据块。因为 VM 文件仅仅使用 2 个比特表示一个数据块，所以它和堆表文件的体积比是： $8192 * 8 : 2$ ，就是堆表的体积是 VM 文件的三万多倍。因为表的最大数据块数量差不多是 4GB 个，所以 VM 文件的最大体积是 $4GB * 2 / 8 = 1GB$ 。通常情况下，VM 文件都是很小的。譬如，一个 1TB 大小的堆表文件，它对应的 VM 文件只有 32MB 而已。

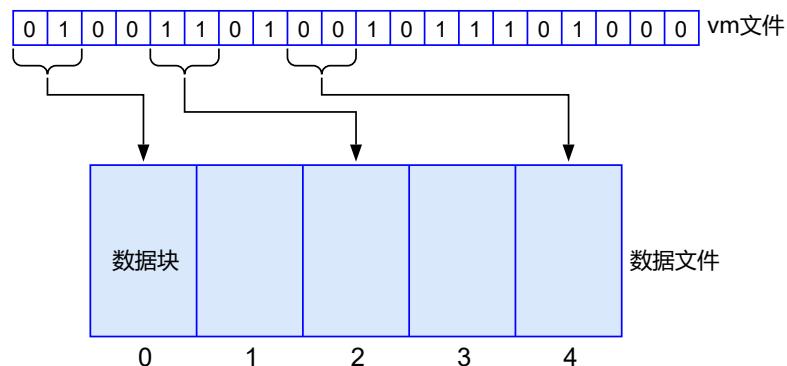


图 8.5: VM 文件和数据文件的关系

VM 文件也是由 8KB 的数据块组成，它的数据块格式非常简单：开始的 24 字节的页头和堆表的页头格式完全一样，其余的字节全部用来保存对应表的数据块的信息。所以一个 VM 文件的数据块可以记录 $(8192 - 24) * 4$ ，即对应表的 32672 个数据块。

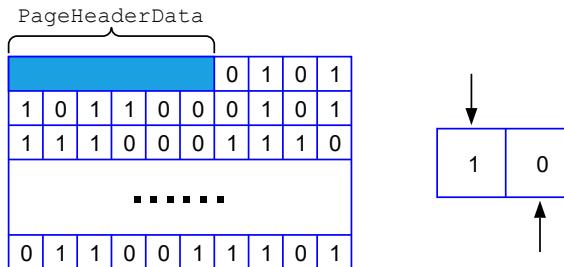


图 8.6: VM 文件的数据块的结构

在理解了上面的 VM 文件和数据块的基本结构以后，再去看和它相关的源码，就比较容易一些了。譬如，下面的一些宏定义就不难理解。

```
/* in src/backend/access/heap/visibilitymap.c */
#define MAPSIZE (BLCKSZ - MAXALIGN(sizeof(PageHeaderData)))
#define BITS_PER_BYTE     8
#define BITS_PER_HEAPBLOCK 2
/* Number of heap blocks we can represent in one byte */
#define HEAPBLOCKS_PER_BYTE (BITS_PER_BYTE / BITS_PER_HEAPBLOCK)
/* Number of heap blocks we can represent in one visibility map page. */
#define HEAPBLOCKS_PER_PAGE (MAPSIZE * HEAPBLOCKS_PER_BYTE)
/* Mapping from heap block number to the right bit in the visibility map */
#define HEAPBLK_TO_MAPBLOCK(x) ((x) / HEAPBLOCKS_PER_PAGE)
#define HEAPBLK_TO_MAPBYTE(x) (((x) % HEAPBLOCKS_PER_PAGE) / HEAPBLOCKS_PER_BYTE)
#define HEAPBLK_TO_OFFSET(x) (((x) % HEAPBLOCKS_PER_BYTE) * BITS_PER_HEAPBLOCK)
```

在上述宏定义中，BITS_PER_BYTE 表示每个字节有 8 个比特，BITS_PER_HEAPBLOCK 表示每个数据块由 2 个比特表示。MAPSIZE 表示 vm 文件中的数据块里面有多少个字节可以表示表的数据块，它是 8192 字节扣除页头的 24 个字节。HEAPBLOCKS_PER_PAGE 表示一个 vm 文件的数据块可以存储多少个表的数据块的描述信息，很显然它的值是 32672。知道这些规律，给你一个堆表的数据块的编号，你很容易推算出它对应的 2 个比特在 VM 文件的哪个数据块的哪个字节的哪个偏移量。这就是 HEAPBLK_TO_MAPBLOCK、HEAPBLK_TO_MAPBYTE 和 HEAPBLK_TO_OFFSET 三个宏的功能。

我们如果不了解细节，只要在头脑中想象 VM 文件是一个长长的数组，每个成员占用 2 个比特即可。VACUUM 操作会从堆表的 0 号块一次扫描，直到堆表的最后一个数据块。在这个漫长的扫描和处理过程中，VACUUM 操作会依靠这个 VM 数组里面的信息来决定手里的数据块是否可以被跳过，从而加速了 VACUUM 操作的处理性能。这就是 VM 文件被设计的根本目的。

8.3 CLog 的清理

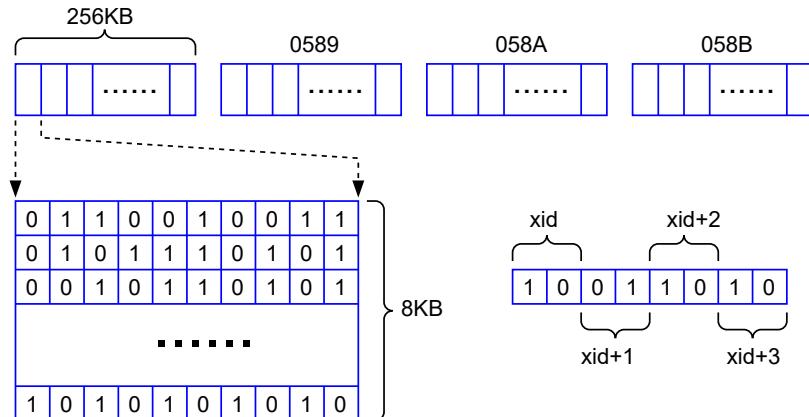


图 8.7: CLog 的结构

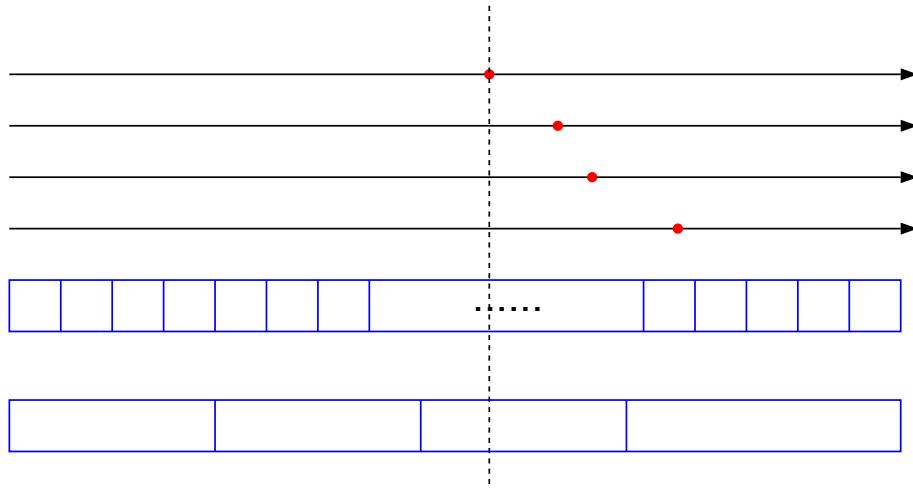


图 8.8: CLog 的清理

8.4 统计信息的更新

所谓统计信息，指的是描述表中数据的信息，譬如这张表里面有多少条记录？这张表占据了多少个数据块？这些信息对于 SQL 执行器产生高效的执行计划至关重要。实际中有一个经常做的操作，先把一张表 truncate 掉，然后使用一个批处理程序批量加载一千万条记录到这个表中。如果统计信息更新不及时，数据库可能认为这是一个空表。很显然，基于这样陈旧的信息，数据库很难产生正确的执行计划，从而导致后面的查询性能大大下降。下面我们通过一个小实验，来对统计信息建立一个初步的概念。首先我们创建一个测试表，然后往里面插入一些随机数据，具体操作如下：

```
oracle=# CREATE TABLE test1(id INT PRIMARY KEY, value CHAR(40));
CREATE TABLE
oracle=# INSERT INTO test1 SELECT generate_series(1,300), md5(random()::text);
INSERT 0 300
oracle=# SELECT * FROM test1 ORDER BY id LIMIT 5;
 id |          value
----+-----
```

```

-----+
1 | b2c0fbbf855e96ae64f50b4a8f4ee17e
2 | 0ba52309fb325606c24ddbde60763072
3 | 565f62a2eecd61cd5335efe6d2c78c90
4 | 6a7e3288ec2eab34c6dbf05fa56f2a35
5 | ae5d8e4c251f647e571af25f9b483985
(5 rows)

```

上述产生批量数据用到了 3 个函数，具体细节你可以自行查阅。这些技巧可以帮我们产生大批量的测试数据，是非常有用的。PostgreSQL 有若干重要的系统表 pg_statistic、pg_class 等来描述每一张表的统计信息，其中 pg_class 的 reltuples 列表示表里面有多少条记录，relpages 表示表占用了多少个数据块。现在我们查询一下这个测试表的统计信息：

```

oracle=# SELECT oid, relname, reltuples,relpages FROM pg_class WHERE relname='test1';
      oid | relname | reltuples | relpages
-----+-----+-----+-----+
 16428 | test1 |         -1 |          0
(1 row)

```

```

oracle=# ANALYZE test1;
ANALYZE
oracle=# SELECT oid, relname, reltuples,relpages FROM pg_class WHERE relname='test1';
      oid | relname | reltuples | relpages
-----+-----+-----+-----+
 16428 | test1 |        300 |          3
(1 row)

```

我们可以看到，一开始这张测试表的统计信息是过时的。我们执行 ANALYZE 这条命令后，该表的统计信息就被更新到了最新值。我们可以看到该表中有 300 条记录，一共占据了 3 个数据块。因为统计信息对于查询的性能至关重要，所以 PostgreSQL 会在合适的时机自动执行统计信息的更新任务。ANALYZE 命令无非是给我们提供了一种人工干预统计信息更新的手段而已。

8.5 HOT

HOT 是 Heap Only Tuples 的缩写，它是一种优化技术。我们通过一个例子来展示它的基本概念。具体的实验过程如下：

```

oracle=# create table hot(id int primary key, name char(1));
CREATE TABLE
oracle=# insert into hot values(1000, 'A');
INSERT 0 1
oracle=# update hot set name='B' where id=1000;
UPDATE 1

```

上面的操作很简单，就是创建了一张表，往里面插入了一条记录，然后对这条记录进行了更新。因为这张表有主键，就是第一列 id，所以它有一个 B 树索引，我们可以看到这个索引的名称叫 hot_pkey。

```

oracle=# \d+ hot
                                         Table "public.hot"
   Column |      Type       | Collation | Nullable | Default | Storage  | Compression | Stats target |
             Description

```

```

-----+-----+-----+-----+-----+-----+
| id   | integer      |          | not null |          | plain    |          |
| name | character(1) |          |           |          | extended |
-----+
Indexes:
  "hot_pkey" PRIMARY KEY, btree (id)
Access method: heap

```

因为这张表极小，所以数据文件只有一个数据块，编号为 0，而且索引有两个数据块，第一个是元数据块，编号为 0；第二个数据块是根数据块，也是叶子数据块，编号为 1。我们做如下查询：

```

oracle=# select lp, lp_flags, lp_off, lp_len,t_xmin, t_xmax, t_ctid, t_infomask2
oracle=# from heap_page_items(get_raw_page('hot',0));
lp | lp_flags | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_infomask2
-----+-----+-----+-----+-----+-----+
 1 |       1 | 8160 |     30 |    759 |    760 | (0,2) | 16386
 2 |       1 | 8128 |     30 |    760 |      0 | (0,2) | 32770
-----+
(2 rows)

oracle=# SELECT itemoffset,ctid,itemlen,data,dead FROM bt_page_items('hot_pkey',1);
itemoffset | ctid | itemlen | data | dead
-----+-----+-----+-----+
 1 | (0,1) | 16 | e8 03 00 00 00 00 00 00 | f
-----+
(1 row)

```

从上面的结果可以很清楚地看到，表的数据块中有两条记录。因为 UPDATE 操作是先删后插，第一条是死亡记录，第二条是正常记录。同时我们也注意到了，索引上只有一条记录，指向了第一条死亡记录，并不是指向了第二条记录。第一条记录的 t_ctid 这一列指向了第二条记录。第一条记录的 t_infomask2 的值是 16386，就是十六进制的 0x4002。第二条记录的 t_infomask2 的值是 32770，就是十六进制的 0x8002。下面我们看如下的定义：

```

/* in src/include/access/htup_details.h */
#define HEAP_HOT_UPDATED          0x4000 /* tuple was HOT-updated */
#define HEAP_ONLY_TUPLE            0x8000 /* this is heap-only tuple */

```

所以第一条记录的 HEAP_HOT_UPDATED 位为 1，而第二条记录的 HEAP_ONLY_TUPLE 为 1。我们可以用图 XX 来表示上述的实验结果：

	t_xmin	t_xmax	t_ctid	t_infomask2	用户的数据
记录1	195		(5,1)		1000, 'A'

	t_xmin	t_xmax	t_ctid	t_infomask2	用户的数据
记录1	195	196	(5,2)	HEAP_HOT_UPDATED	1000, 'A'
记录2	196		(5,2)	HEAP_ONLY_TUPLE	1000, 'B'

图 8.9: HOT 的基本概念

整个实验只进行了一次 UPDATE 操作，但是这个 UPDATE 操作有一些特点：第一个特点是它只修改了没有索引的列 name，并没有修改索引的列 id。第二个特点是 UPDATE 先删后插的时候，在被删除的记录所在的数据

页上还有足够的空间来保存插入的新记录，所以两条记录在同一个页面上。因为避免修改索引上的记录，HOT 技术就是保持索引的内容不变，而是通过一个链条（chain）把同一条记录的不同版本串联了起来。如图 XX 所示：

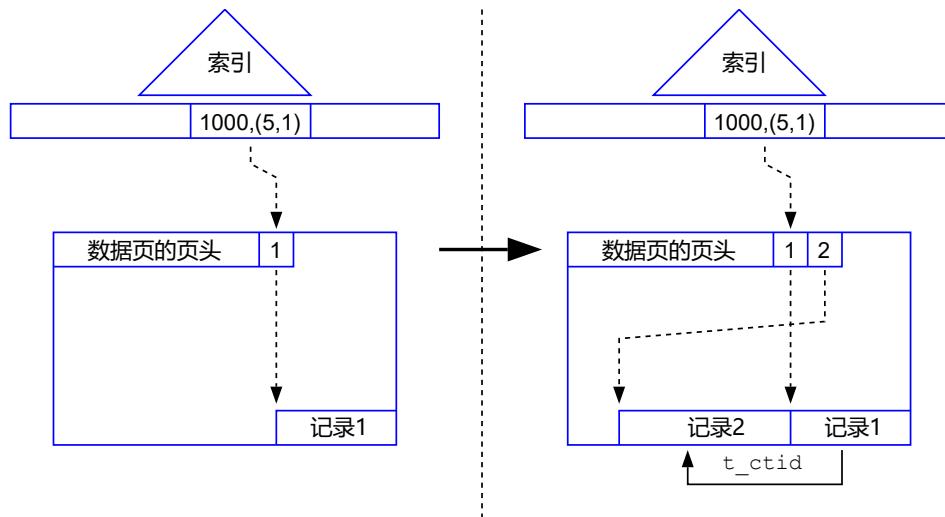


图 8.10: HOT 的链条

当某一个查询访问索引时，它拿到的是指向第一条记录的指针，所以它首先找到了第一条记录，它进一步判断这条记录的 `t_infomask2` 的 `HEAP_HOT_UPDATED` 是否为 1，如果是 1，它就知道这并不是最终版本的记录，它就可以通过这条记录的 `t_ctid` 这一列顺藤摸瓜，找到了第二条记录。因为第二条记录的 `HEAP_ONLY_TUPLE` 位是 1，所以这是该查询所需要的最终版本的记录。如果一次 `UPDATE` 操作修改的列上没有任何索引，而且要插入的新记录可以存放在和老记录同一个数据页上，就会出现上述的现象，即索引并没有被修改，只是通过一个链条把不同版本的记录串联起来，这种技术就叫做 HOT 技术。因为 `t_ctid` 是六个字节，包含 4 个字节的数据块的编号和 2 个字节的偏移量，理论上第二个要求不是必须的，因为可以通过数据块的编号信息找到另外的数据块。但是从性能的角度考虑，HOT 被限制在同一个数据页上，这一点要特别注意。

过了一段时间，死亡的第一条记录可能被重复使用，为了避免 `t_ctid` 这个的中断，PostgreSQL 就会执行数据页的修剪 (page pruning) 的动作。经过数据页修剪后，数据页变成了如图 XX 所示的布局：

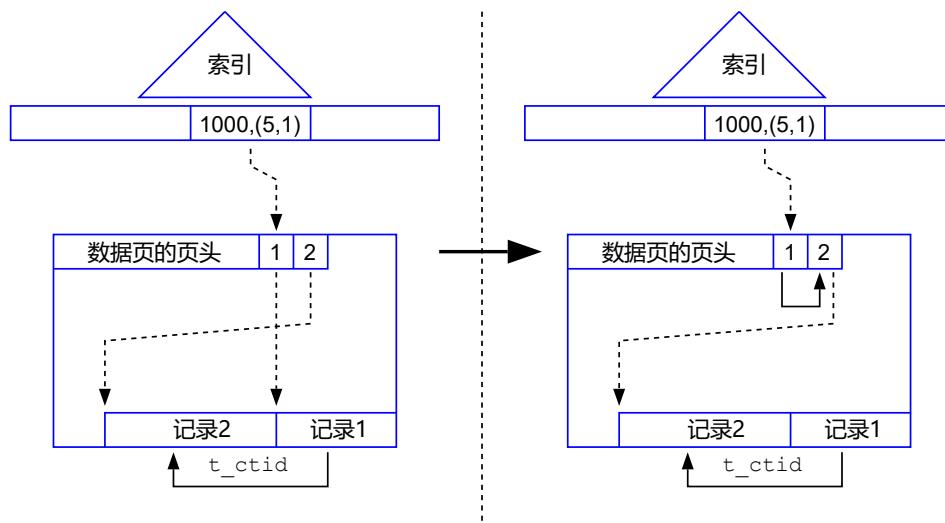


图 8.11: 记录的重定向

我们很清楚地观察到，第一条记录的指针指向了第二条记录。这样即使第一条记录被覆盖，这个 HOT 链条依然有效。在满足一定条件下，`SELECT` 操作也可能触发数据页的修剪操作，从而让该数据页变成脏

页，也会产生相应的 WAL 记录来保护这种修剪的修改动作。我们下面使用 Vacuum 操作来触发修剪动作。关于 Vacuum 的具体内容，后文中会进行详细介绍。我们执行如下动作：

```
oracle=# vacuum hot;
VACUUM

oracle# select lp, lp_flags, lp_off, lp_len,t_xmin, t_xmax, t_ctid, t_infomask2 from heap_page_items(
    get_raw_page('hot',0));
lp | lp_flags | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_infomask2
---+-----+-----+-----+-----+-----+-----+
 1 |      2 |      2 |      0 |          |          |          |
 2 |      1 |  8160 |     30 |    755 |          0 | (0,2) |      32770
(2 rows)
```

对比前面的实验结果，我们可以观察到一些变化：第一条记录的 lp_flags 从 1 变成了 2，lp_off 从 8160 变成了 2，从它的长度 lp_len 从 30 变成了 0。我们可以参考如下定义：

```
/* in src/include/storage/itemid.h */
#define LP_UNUSED    0 /* unused (should always have lp_len=0) */
#define LP_NORMAL    1 /* used (should always have lp_len>0) */
#define LP_REDIRECT   2 /* HOT redirect (should have lp_len=0) */
#define LP_DEAD       3 /* dead, may or may not have storage */
```

由上面的定义可知，第一条记录变成了 LP_REDIRECT 类型的指针，它的 lp_off 为 2，表示它直接指向了第二条记录。如果我们继续修改这条记录：

```
oracle# update hot set name='C' where id=1000;
UPDATE 1
oracle# vacuum;
VACUUM

oracle# select lp, lp_flags, lp_off, lp_len,t_xmin, t_xmax, t_ctid, t_infomask2 from heap_page_items(
    get_raw_page('hot',0));
lp | lp_flags | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_infomask2
---+-----+-----+-----+-----+-----+-----+
 1 |      2 |      3 |      0 |          |          |          |
 2 |      0 |      0 |      0 |          |          |          |
 3 |      1 |  8160 |     30 |    761 |          0 | (0,3) |      32770
(3 rows)
```

我们可以看到，第一条记录的指针依然是 LP_REDIRECT 类型，它的 lp_off 的值为 3，表明它指向了第三条记录，即最后版本的那条记录。整个修改过程如图 XX 所示：

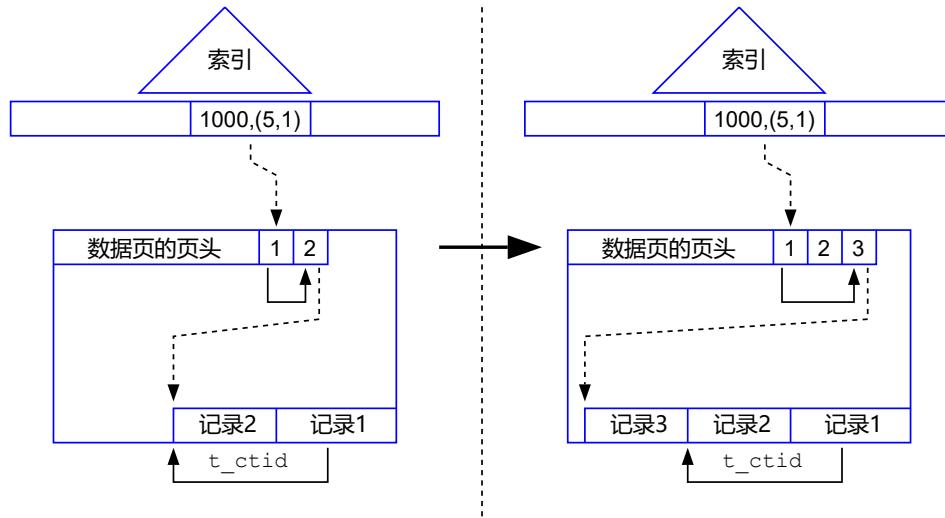


图 8.12: 多个版本的记录的 HOT 链条

由此可见，为了缩短 HOT 链条的长度，第一条 LP_REDIRECT 类型的指针会一竿子到底，指向了最终版本的那条记录，这样就保证了链条上最多有两条记录，缩短了遍历链条的时间。

第一条记录死亡以后，它所占用的空间就变成了空闲空间，可以被重复利用。但是该数据页的空闲空间也出现了碎片化的现象，因为在第二条记录的前面和后面都有空闲空间。为了把空闲空间合并成一个大块，方便未来的空间分配，在做页面修剪的同时，也会对数据页进行碎片化的整理，被称为 defragmentation。它的基本概念如图 XX 所示。

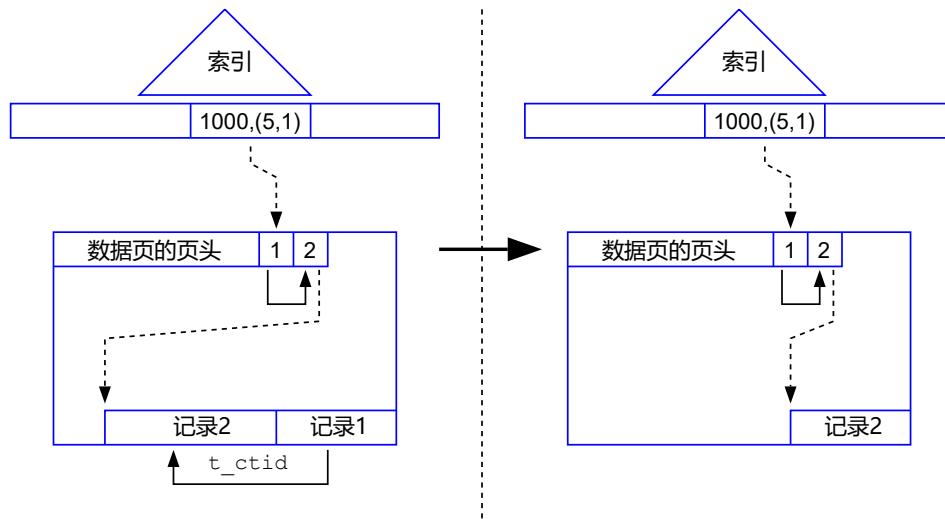


图 8.13: 碎片化整理的概念

Vacuum 操作会执行数据页的修剪和碎片化的整理工作。大家可以利用 pageinspect 这个插件的函数查阅一下空闲空间的变化情况，应该非常容易，这里我们就不详细展示了。

8.6 VACUUM 的过程

VACUUM 最重要的工作是清理堆表和索引中的死亡记录，所以它大体上分为两个阶段：搜集垃圾，处理垃圾。VACUUM 操作会从堆表的 0 号数据块开始扫描，一直到最后一个数据块。这个过程的主要函数是 `lazy_scan_heap()`，它的基本骨架如下所示：

```
/* in src/backend/access/heap/vacuumlazy.c */
static void lazy_scan_heap(LVRelState *vacrel)
{
    BlockNumber blkno, rel_pages;
    rel_pages = vacrel->rel_pages; /* rel_pages里面记录了这张堆表有多少个数据块 */
    for (blkno = 0; blkno < rel_pages; blkno++)
    {
        ..... /* 做VACUUM的工作 */
    }
}
```

可以看出，主要工作内容就是一个 for 循环，非常简单易懂。VACUUM 第一步当然要搜集死亡记录的信息。它会把搜集到的死亡记录放在一个数组中，我们可以称这个数组为“死亡记录数组”。它的基本结构如下所示：

```
#define FLEXIBLE_ARRAY_MEMBER /* empty */

typedef uint16 OffsetNumber;

/* in src/include/storage/block.h */
typedef struct BlockIdData {
    uint16          bi_hi;
    uint16          bi_lo;
} BlockIdData;

/* in src/include/storage/itemptr.h */
typedef struct ItemPointerData {
    BlockIdData ip_blkid;
    OffsetNumber ip_posid;
}

/* in src/include/commands/vacuum.h */
typedef struct VacDeadItems {
    int             max_items;      /* # slots allocated in array */
    int             num_items;      /* current # of entries */
    ItemPointerData items[FLEXIBLE_ARRAY_MEMBER];
} VacDeadItems;
```

我们可以使用图 XX 来表示这个死亡记录数组的基本结构。我们可以看到，它实际上就是一个长长的数组，每一个成员都是一个 6 字节的 TID 指针，通过这个指针，我们就知道死亡记录的具体位置。max_items 表示这个数组的最大体积，num_items 表示数组中当前有多少个元素。因为扫描是从堆表的 0 号数据块依次进行的，所以这些 TID 是排序的。

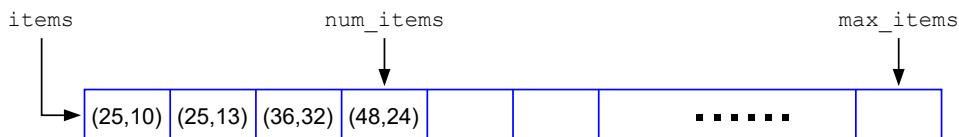


图 8.14：死亡记录数组的结构

那么死亡记录数组到底有多大呢？参数 maintenance_work_mem 控制着它的体积上限。目前我们介绍的是手工 VACUUM 的过程，后面我们会介绍自动 VACUUM(auto vacuum)。自动 VACUUM 会使用参数 autovacuum_work_mem 控制死亡记录数组的最大体积，如果这个参数不指定，就使用 maintenance_work_mem 参数。如果这个参数设置很大，同时有很多 VACUUM 在操作，毫无疑问会吃光服务器的内存。所以 PostgreSQL 有一个固定限制，就是

死亡记录数组的最大体积不能超过 1GB。因为 $1\text{GB} / 6 = 178956970$, 这个就是死亡记录数组中成员数量的最大值。如果一个表非常庞大, 死亡数组装不下所有的死亡记录, 该怎么办呢? VACUUM 操作会采用多趟扫描的方法, 死亡记录数组满了, 就处理一批, 然后清空这个数组, 继续扫描, 如此反复, 直到这张堆表中所有的死亡记录被处理完毕。

死亡记录数组是 VACUUM 操作中最核心也是占用内存最大的数据结构。VACUUM 所有的操作基本上都是围绕这个数组来展开的。一次完整的 Vacuum 的过程分为若干个阶段。这些阶段的定义如下所示:

```
/* in src/backend/access/heap/vacuumlazy.c */
typedef enum {
    VACUUM_ERRCB_PHASE_UNKNOWN,
    VACUUM_ERRCB_PHASE_SCAN_HEAP,
    VACUUM_ERRCB_PHASE_VACUUM_INDEX,
    VACUUM_ERRCB_PHASE_VACUUM_HEAP,
    VACUUM_ERRCB_PHASE_INDEX_CLEANUP,
    VACUUM_ERRCB_PHASE_TRUNCATE
} VacErrPhase;
```

VACUUM 操作的第一个阶段是对堆表进行扫描 (heap scan), 这个阶段是搜集死亡记录的阶段。第二个阶段是对索引进行清理 (index vacuum), 是根据第一阶段的死亡记录删除索引中的对应记录。第三个阶段是堆表清理 (heap vacuum), 是把堆表中数据块的记录指针数组中标记为死亡的指针变成空闲指针, 方便未来的插入修改删除等操作。第四个阶段为称为索引的清除 (index clean), 第五个阶段被称为堆表数据块的砍尾工作。我们后面会依次介绍每个阶段的具体内容。图 XX 表示 VACUUM 的不同阶段。如果死亡记录非常多, 死亡记录数组一次装不下, 就会出现多趟扫描的现象, 所以头三个阶段可能是反复循环的状态。最后的索引清除和砍掉尾页的操作是一次性的。

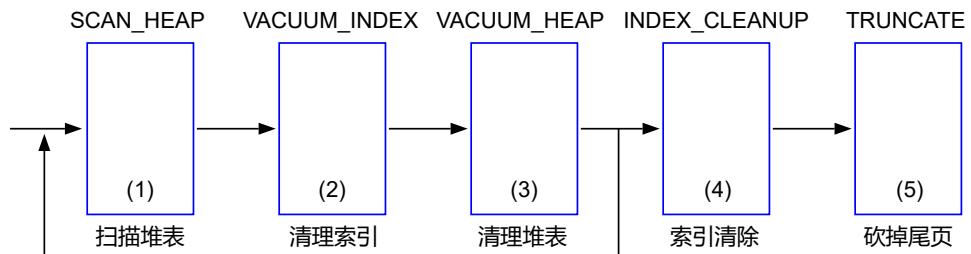


图 8.15: VACUUM 的不同阶段

下面我们就依次介绍每个阶段具体做了什么事情。

8.6.1 第一阶段 - 扫描堆表

每个成员占据 6 个字节, 如果 maintenance_work_mem 是 6GB, 则一共可以存储 10 亿条记录 ($=1024 * 1024 * 1024$), 基本上够用了。

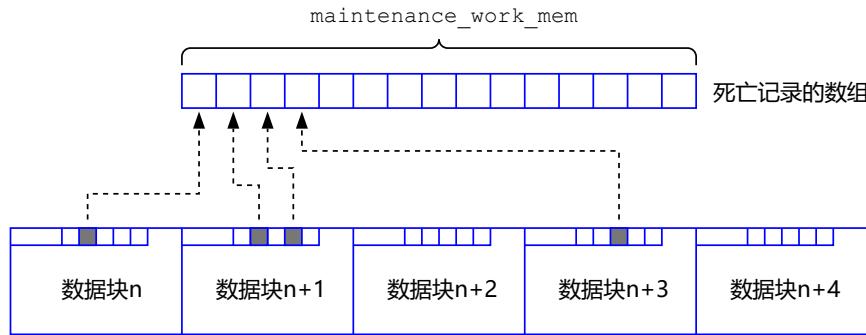


图 8.16: Vacuum 的第一阶段

扫描整个堆表, 确认哪些记录是死亡记录, 然后把它们的 TID 放在内存中, 内存的大小由 maintenance_work_mem 参数来决定。重新组织数据页, (reorganize prune), 更新 FSM 信息。

```
/* in src/include/commands/vacuum.h */
typedef struct VacDeadItems {
    int             max_items;      /* # slots allocated in array */
    int             num_items;      /* current # of entries */
    /* Sorted array of TIDs to delete from indexes */
    ItemPointerData items[FLEXIBLE_ARRAY_MEMBER];
} VacDeadItems;
```

8.6.2 第二阶段 - 对索引进行 VACUUM

扫描 maintenance_work_mem 缓冲区, 去掉死亡记录。

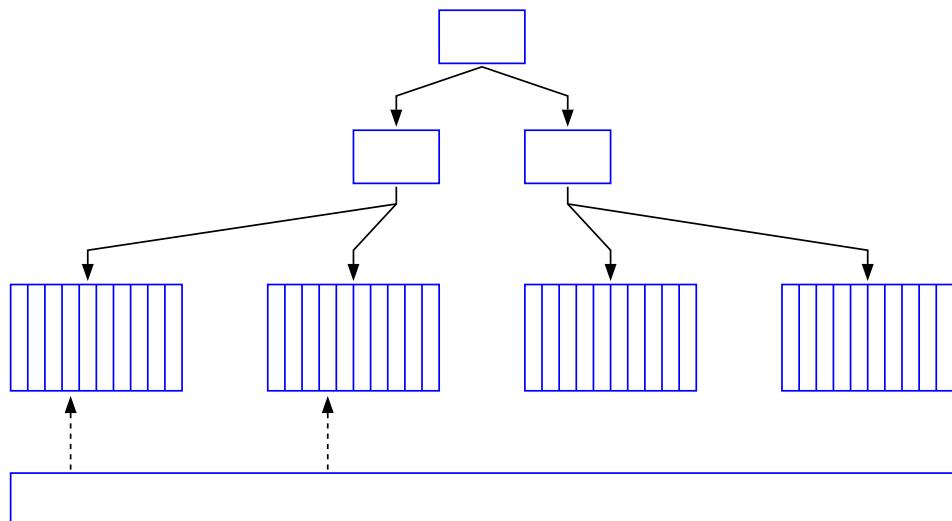


图 8.17: Vacuum 的第一阶段

8.6.3 第三阶段 - 堆表的 VACUUM

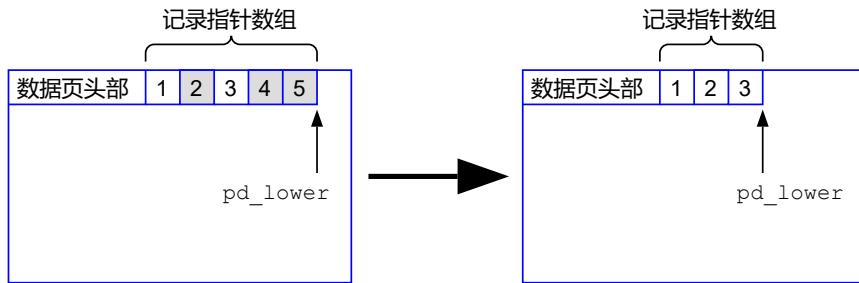


图 8.18: Vacuum 的第一阶段

8.6.4 第四阶段 - 索引的清理

8.6.5 第五阶段 - 堆表尾部数据块的处理

8.7 自动 VACUUM

自动 Vacuum 的条件:

```
oracle=# select oid, relname, relfrozenxid from pg_class where relname='state';
oid | relname | relfrozenxid
-----+-----+
16389 | state | 742
(1 row)
```

`relfrozenxid + autovacuum_freeze_max_age`
`autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor × reltuples`
`autovacuum_vacuum_insert_threshold + autovacuum_vacuum_insert_scale_factor × reltuples`

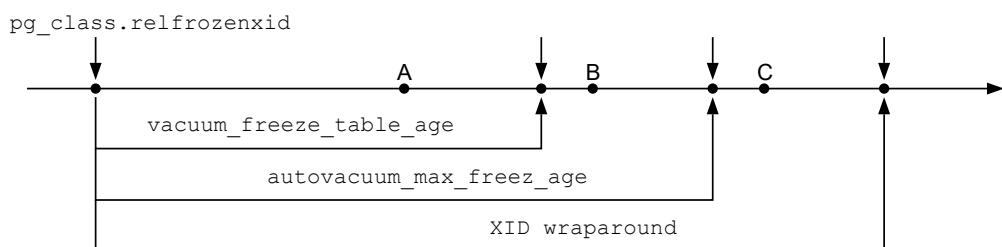


图 8.19: Vacuum 的第一阶段

在源代码中我们会看到这么一段注释:

```
/* in src/backend/postmaster/autovacuum.c */
/* A table needs to be vacuumed if the number of dead tuples exceeds a
 * threshold. This threshold is calculated as
 *
 * threshold = vac_base_thresh + vac_scale_factor * reltuples
 */
```

它给出了一个判断一个表是否需要 Vacuum 的条件。

```
autovacuum_vacuum_threshold = 50      # min number of row updates before
autovacuum_vacuum_scale_factor = 0.2   # fraction of table size before vacuum
```

根据以上的配置，假设一张表共计 10000 条记录， $50 + 0.2 * 10000 = 2050$ ，则这个表中的死亡记录超过 2050，这张表就需要 Vacuum。

8.8 VACUUM 的监控

```
onshift=# select * from pg_stat_progress_vacuum where relid=24319;
-[ RECORD 1 ]-----+
pid          | 3674038
datid        | 23272
datname      | onshift
relid        | 24319
phase        | vacuuming indexes
heap_blks_total | 138909868
heap_blks_scanned | 136710119
heap_blks_vacuumed | 132412038
index_vacuum_count | 2
max_dead_tuples | 178956970
num_dead_tuples | 178956698
```

扫描的块数的理解，假设一张表有 10000 个数据块，可以被跳过的块是 150 块，则被扫描的块共计 9850 块。所以扫描的块除于总块数只能大致估计扫描的进度，并不准确，因为你不知道有多少块可以被跳过。

8.9 对表的全清理 (VACUUM FULL)

上述 VACUUM 操作并不会阻止别的用户对该表的读写，所以它是在线 (online) 操作，可以随时运行。我们称这种 VACUUM 为标准 VACUUM。当然，在标准 VACUUM 执行过程中，无法对该表进行 CREATE INDEX, ALTER TABLE 等涉及到表结构修改的操作。因为标准 VACUUM 只是清除了数据块中的死亡记录，并不会把剩下的记录重新组织成一个更加紧凑的形式，所以标准 VACUUM 并不会缩小表的体积。图 XX 展示了标准 VACUUM 的效果：

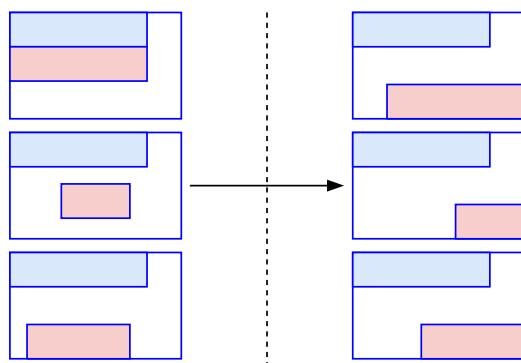


图 8.20: 标准的 VACUUM 的效果

我们可以看到，虽然剩余的记录可以存放在一个数据块中，但是 VACUUM 依然使用了三个数据块来保存记

录。无非是每一个数据块的内部被重新组织了。另外一种形式的 VACUUM 被称为全 VACUUM(vacuum full)。它执行后的效果如图 XX 所示：

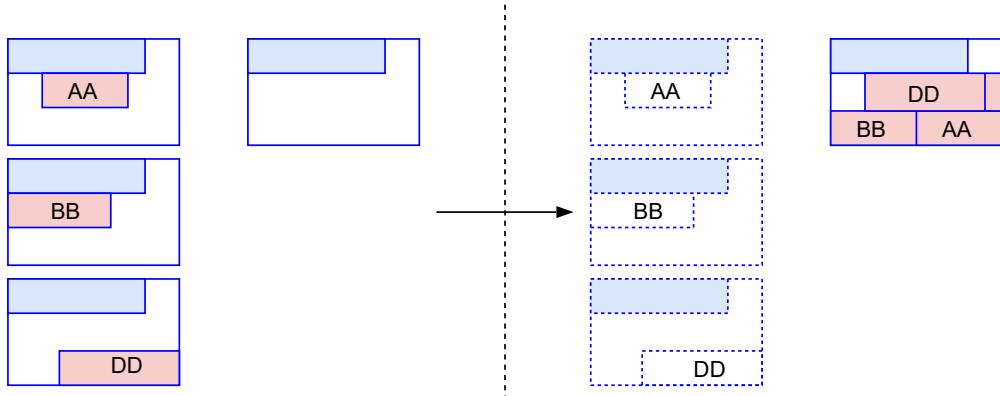


图 8.21: XXXXX

从上图中我们可以看出，全 vacuum 会把所有的页面进行合并。譬如表中有 3 个页面，经过全 vacuum 后，如果所有的数据都可以放在一个页面中，全 vacuum 会把老的数据页都删除掉，用一个全新的数据页来保存最终的数据。很显然全 vacuum 把数据打包的更结实紧凑。这种效果有时候是我们想要的，有时候却不是我们想要的。譬如这张表未来会进行密集的 UPDATE 操作，很自然如果页面中有预留的空间会更好。全 vacuum 的最大缺点是它在执行的过程中会把整张表进行独占式锁定，所以在全 vacuum 没有完成之前，任何对这张表的操作都是不可能的。因为这个限制，全 vacuum 使用的频率并不是很高。

第九章 数据库的逻辑复制

数据库的逻辑复制是和物理复制有很大区别的数据库复制技术。通过第五章的学习，我们知道：物理复制是机械地把 WAL 记录通过网络传输到备库，然后利用这些 WAL 记录，让备库的数据和主库百分之百一致。物理复制的备库是只读的，它只能接受来自主库的修改，不接受来自其他用户的修改。逻辑复制则不然，它的本质是对主库的 WAL 记录进行反解析，变成类似 SQL 语句的指令，然后把这些指令在备库上执行。我们站在备库的角度，对比它接收到的物理复制和逻辑复制的消息包格式，就能理解这其中的差异。图 9.1 表示的是备库接收到的一个典型的物理复制的消息包：

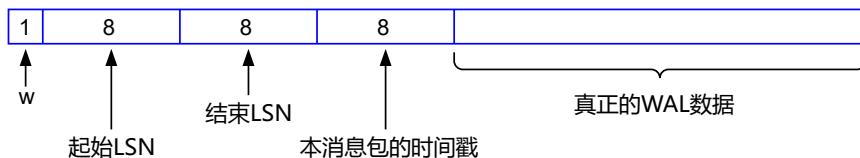


图 9.1：物理复制的消息包

我们可以看出，该消息包分为五个域。第一个域的长度为一个字节，表示消息包的类型，小写的 w 表示这是一个包含数据的消息包，而真正的 WAL 数据在第五个域中。第二、三、四个域的长度都是 8 字节，第二个域表示 WAL 数据的起始 LSN，第三个域表示 WAL 数据的终止 LSN，第四个域表示该消息包的时间戳。我们可以很清楚地看到，备库接收到来自主库的 WAL 记录是“原汁原味”的，没有任何改变。备库的 WAL 接收进程只不过是“不动脑子”地把这些 WAL 记录写入到 WAL 文件中正确的位置，然后通知恢复进程进行处理。图 9.2 表示逻辑复制的一个消息包，它的格式和上述的物理复制的消息包就有很大的不同。

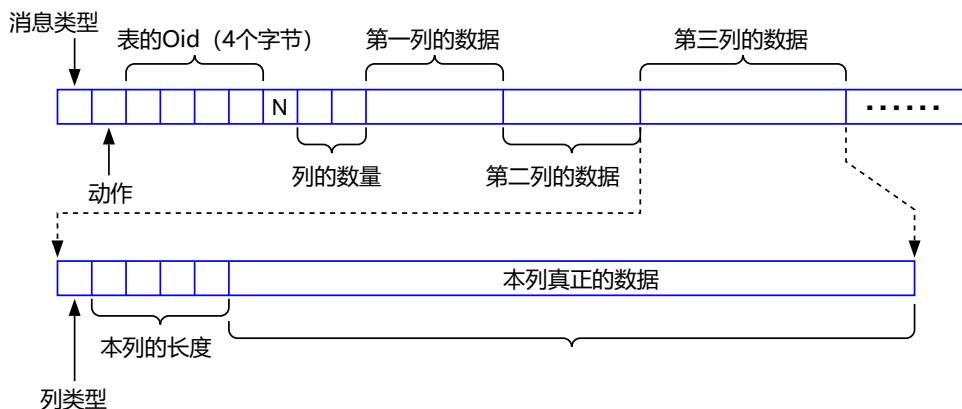


图 9.2：逻辑复制的消息包示例

这个消息包表示要插入一条记录，它分成了更多个域。第一个域的长度为一个字节，表示消息类型，其含义和物理复制的消息包完全一样。第二个域的长度依然是一个字节，表示本次修改的动作类型。如果是插入操作的话，这个字节为大写的 I，表示 INSERT 的意思。第三个域的长度为 4 字节，表示要插入的表的 Oid。第四个域的长度是一个字节，目前固定写死为大写的 N。第五个域的长度为 2 字节，表示后面的数据有多少个列。每一个列的格式列在了图 9.2 的下半部分，它的格式也很容易懂，分别表示列的类型，列的数据的长度和真正的数据本身。

通过对物理复制和逻辑复制的消息包的格式差异，我们可以看出，在逻辑复制的环境中，主库是对 WAL 记录进行了解析的工作，把原始的 WAL 记录的信息解析成类似 SQL 指令的消息包。逻辑备库拿到这些指令后就可以轻松地执行各种修改操作了。这一点看似不起眼的差异，造成了逻辑复制和物理复制有一些重大的区别：一是逻辑复制更加灵活，它可以选择只执行特定的 SQL，譬如只复制指定的表，而且对同一张表，可以指定过

滤条件，类似 SQL 的 WHERE 子句，只复制感兴趣的数据集。二是逻辑复制的备库是可读可写的，因为逻辑复制的备库的目的是选择性地复制来自主库的数据，并不要求和主库百分百同步，所以逻辑复制的备库也可以接受任何合法用户的修改。所以在本节中，我们把逻辑复制的主库称为“源数据库”，备库称为“目标数据库”，这样就避免了备库一定是只读的印象。逻辑复制和物理复制的第三个区别是：逻辑复制可以跨大版本，即源数据库和目标数据库的大版本可以不同。假如一个 PostgreSQL 的版本为 15.3，则 15 为大版本，PostgreSQL 12.16 的大版本为 12。例如：源数据库为 PostgreSQL 10，而目标数据库的版本为 PostgreSQL 16。物理复制则要求主库和备库的大版本必须相同，譬如主库是 PostgreSQL 15.2，备库的版本为 15.4，但是不能为 PostgreSQL 14 或者 PostgreSQL 16。

对数据库稍微了解的读者都知道，数据库的应用大体上分为两种类型：联机事务处理 (OLTP: Online Transaction Processing) 和联机分析处理 (OLAP: Online Analytical Processing) 系统两大类。它们两者的区别也非常容易理解：OLTP 的用户量大，事务短，主要起到业务数据搜集的作用；OLAP 用户量小，运行的都是复杂的查询，主要是企业内部用来分析业务数据的，OLAP 又可以被称为数据仓库 (data warehouse)。图 9.3 展示了一个典型的企业应用场景，它同时使用了数据库的物理复制和逻辑复制两种复制技术。

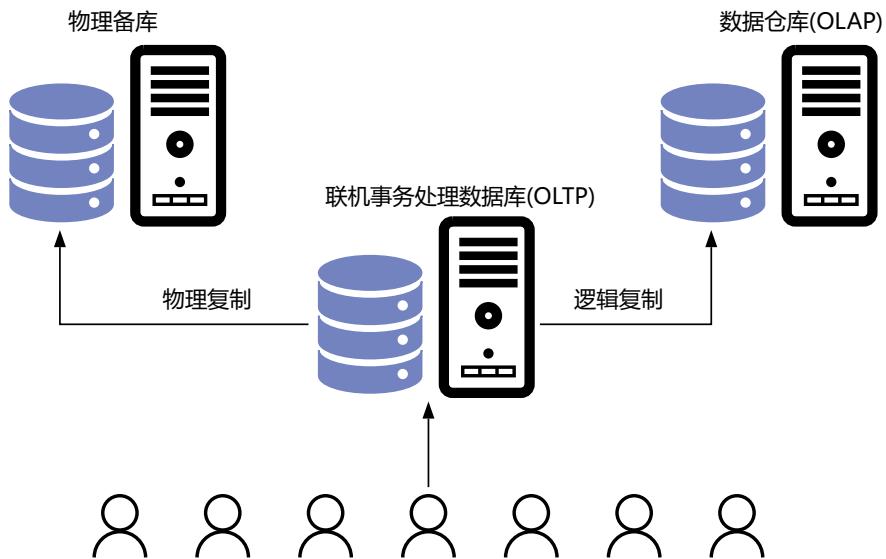


图 9.3: 典型的 OLTP/OLAP 系统架构

处于核心地位的是 OLTP 系统，它面对成千上万的用户，是企业的命脉所在。为了保证这个数据库和相关系统的不间断运行，企业往往采用流复制的技术，搭建一套或者多套备库，如图 9.3 左上角所示。为了让企业高层人员能够实时掌握企业的经营动态，第一手的原始数据会采用逻辑复制的技术，近乎实时地传输到数据仓库系统中，如图 9.3 中右上角所示，然后由各种数据抽取和处理工具 (ETL) 进行处理，最终以各种报表的形式展现在高层管理人员面前。所以企业往往会同时采用流复制和逻辑复制，把它们的长处运用在不同的企业需求里。

9.1 快速搭建逻辑复制

逻辑复制采用“发布 (publish)-订阅 (subscribe)”的模型，我们先通过一个快速的实验，搭建逻辑复制，让我们先对其有一个感性的认识，然后再讨论这个模型和它背后的体系架构。这个实验需要两台 Linux 机器，一台是源数据库服务器，其 IP 地址是 192.168.137.12；另外一台是目标数据库服务器，其 IP 地址是 192.168.137.13。具体的实验过程如下：

```
/* ===== 在源数据库机器192.168.137.12上执行以下命令===== */
$ initdb -D /opt/data/logical      /* 创建一个实验数据库集群 */
```

```

/* 修改/opt/data/logical/postgresql.conf, 在文件尾加入如下几行参数 */
listen_addresses = '*'
wal_level = logical
max_wal_senders = 10
max_replication_slots = 10

/* 修改/opt/data/logical/pg_hba.conf, 在文件尾加入如下几行参数 */
host      all            all            192.168.137.0/24      md5
host      replication    all            192.168.137.0/24      md5
$ pg_ctl -D /opt/data/logical -l logfile start /* 启动数据库 */
$ psql   /* 以超级用户postgres登录数据库集群 */
psql (16.0)
Type "help" for help.

/* 创建一个专用于逻辑复制的用户kevin */
postgres=# CREATE USER kevin WITH REPLICATION LOGIN PASSWORD 'Welcome123';
CREATE ROLE

postgres=# CREATE DATABASE oracle; /* 创建测试数据库oracle */
CREATE DATABASE

postgres=# \c oracle /* 切换到oracle数据库, 创建两张测试表, 并插入记录 */
You are now connected to database "oracle" as user "postgres".
/* 注意每张测试表一定要有主键(primary key) */
oracle=# CREATE TABLE tab1(id INT PRIMARY KEY, value TEXT);
CREATE TABLE
oracle=# CREATE TABLE tab2(id INT PRIMARY KEY, value TEXT);
CREATE TABLE

oracle=# INSERT INTO tab1 VALUES(0, 'Boston');
INSERT 0 1
oracle=# INSERT INTO tab2 VALUES(1, 'Chicago');
INSERT 0 1
oracle=# GRANT ALL ON tab1 TO kevin; /* 把测试表的必要权限赋予kevin用户 */
GRANT
oracle=# GRANT ALL ON tab2 TO kevin; /* 把测试表的必要权限赋予kevin用户 */
GRANT

/* 创建一个发布服务 pub, 其中包含了两张测试表 */
oracle=# CREATE PUBLICATION pub FOR TABLE tab1, tab2;
CREATE PUBLICATION

/* ===== 在目标数据库机器192.168.137.13上执行以下命令===== */
$ initdb -D /opt/data/target /* 创建一个实验数据库集群 */
$ pg_ctl -D /opt/data/target -l logfile start /* 启动数据库 */
$ psql   /* 以超级用户postgres登录数据库集群 */
psql (16.0)
Type "help" for help.

/* 创建测试数据库oracle */
postgres=# CREATE DATABASE oracle;
CREATE DATABASE

postgres=# \c oracle; /* 切换到oracle数据库, 创建两张测试表, 结构和源数据库中的表一样 */
You are now connected to database "oracle" as user "postgres".
oracle=# CREATE TABLE tab1(id INT PRIMARY KEY, value TEXT);

```

```

CREATE TABLE
oracle=# CREATE TABLE tab2(id INT PRIMARY KEY, value TEXT);
CREATE TABLE
/* 创建一个订阅者，指明要订阅源数据库上的pub */
oracle=# CREATE SUBSCRIPTION sub CONNECTION
oracle=# 'dbname=oracle host=192.168.137.12 user=kevin password=Welcome123' PUBLICATION pub;
NOTICE: created replication slot "bsub" on publisher
CREATE SUBSCRIPTION
oracle=# SELECT * FROM tab1 ORDER BY id; /* 可以看到数据顺利地复制到了目标数据库 */
id | value
----+-----
0 | Boston
(1 row)
oracle=# SELECT * FROM tab2 ORDER BY id; /* 可以看到数据顺利地复制到了目标数据库 */
id | value
----+-----
1 | Chicago
(1 row)

```

通过这个快速实验,我们可以清楚地看到,整个实验的关键点是在源数据库上通过”CREATE PUBLICATION”命令创建一个发布者,指明要发布哪些表,在目标数据库上通过”CREATE SUBSCRIPTION”命令创建一个订阅者,指明从源数据库上订阅哪个发布者的信息。我们可以查看一下相关的后台进程:

```

/* ===== 在目标数据库机器192.168.137.13上执行以下命令===== */
$ ps -ef | grep postgres | grep -v grep
.....
postgres    4648    4636  0 06:24 ?          00:00:00 postgres: logical replication apply worker for
subscription 16399
/* ===== 在源数据库机器192.168.137.12上执行以下命令===== */
$ ps -ef | grep postgres | grep -v grep
.....
postgres    4267    4261  0 06:14 ?          00:00:00 postgres: logical replication launcher
postgres    4307    4261  0 06:24 ?          00:00:00 postgres: walsender kevin oracle
      192.168.137.13(39790) START_REPLICATION
/* ===== 在目标数据库机器192.168.137.13上执行以下命令===== */
$ ss -np | grep 39790
tcp  ESTAB 0 0 192.168.137.13:39790 192.168.137.12:5432 users:(("postgres",pid=4648,fd=6))
$ ps -ef | grep 4648 | grep -v grep
postgres  4648  4636  0 06:24 ? 00:00:00 postgres: logical replication apply worker for subscription
16399

```

我们可以看到,类似物理复制的walreceiver和walsender进程对,在逻辑复制中,也有logical replication apply worker和walsender进程对维系网络连接,进行数据传输。吃完人参果以后,我们结合实验步骤来理解逻辑复制的体系。

9.2 逻辑复制的体系结构

首先,我们定义一些术语。在英文语境里面,有 publication 和 publisher, subscription 和 subscriber 四个术语,都是名词。我们把 publication 翻译为“发布”, publisher 翻译为“发布者”, subscription 翻译为“订阅”, subscriber

翻译为“订阅者”，但是“发布”和“订阅”在中文里面还有动词的意义，我们在后文论述的过程中如果必须使用其动词的含义，且不会引起重大混淆时，就统一使用“发布者”指代 publication 或 publisher，统一使用“订阅者”指代 subscription 或 subscriber。

“发布-订阅”这个模型非常容易理解，就类似报社发行报纸，读者订阅报纸的模式。报社是发布者，它发行的报纸就是“发布”，读者就是订阅者，读者家的邮箱就是“订阅”。在逻辑复制领域，发布可以定义为：一组表的集合和其上的修改 (modification)，我们可以使用“CREATE PUBLICATION”命令来创建发布。一个发布包含两个要素：一组表，以及针对这组表的修改，这些修改分为插入 (INSERT)，更新 (UPDATE)，删除 (DELETE) 和清空 (TRUNCATE) 四种类型。我们在创建发布时可以指定只捕获这些修改类型中的一种或者多种，如果不指定，则捕获所有的修改类型。这些修改的数据被解析成类似 SQL 的指令，传送给订阅。发布者是运行一个或多个发布的数据库集群，在不特地强调发布和发布者的区别的语境中，可以笼统地使用发布者指代两者。发布者在捕获 UPDATE 和 DELETE 操作时，它必需要有一个复制标识 (replica identity) 来确定是哪条记录发生了修改，通常情况下这个复制标识就是一个表的主键 (primary key)，所以逻辑复制的一个最佳实践就是：力求确保每张被复制的表都有一个主键。如果做不到这一点，可以考虑使用某一个唯一性索引。如果还不行，逻辑复制会使用整条记录的内容来确定 UPDATE/DELETE 到底发生在哪条记录上。这是最后的办法，性能也很底下，所以通常不建议使用。

订阅是发布产生的数据的消费者，我们通过“CREATE SUBSCRIPTION”命令在目标数据库中创建订阅。订阅接收来自发布传送的数据修改后，在目标数据库上执行这些类 SQL 指令，让目标数据库和源数据库保持同步。订阅者则是订阅所在的目标数据库。发布和订阅是多对多的关系，即一个发布可以被多个订阅所“订阅”；而一个订阅可以“订阅”多个发布，注意双引号里面的订阅是动词。逻辑复制的整体架构可以由图 9.4 来表示：

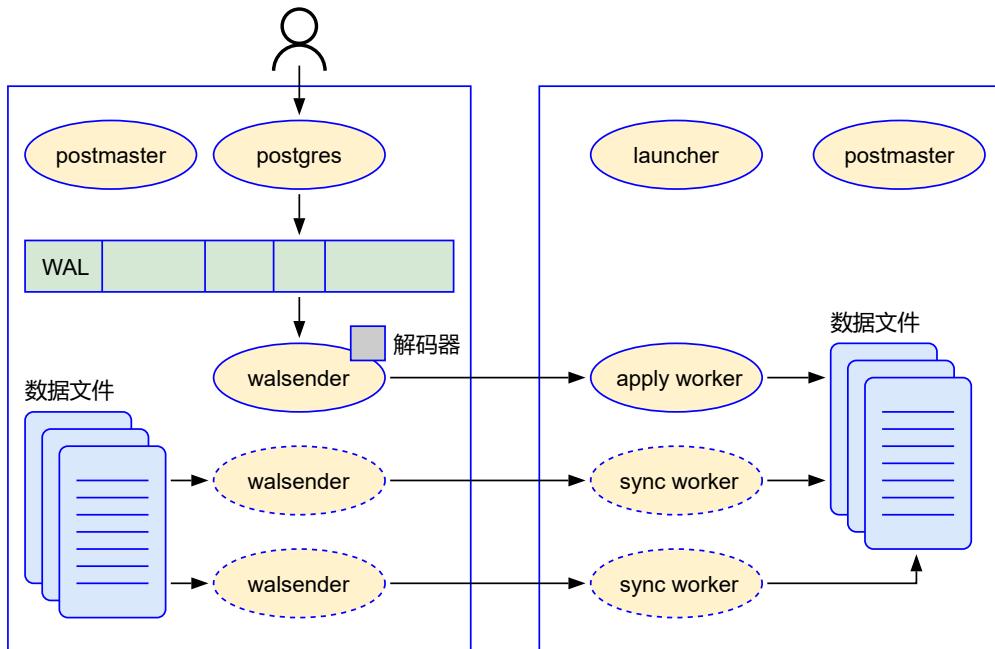


图 9.4: 逻辑复制的体系架构

上图的结构大体上类似物理复制的结构。目标数据库有一个常驻的后台进程，逻辑复制启动 (logical replication launcher) 进程。它的任务是为每一个订阅启动一个逻辑复制的工作进程 (apply worker)。工作进程会和源数据库的主进程建立连接，源数据库的主进程会派出 WAL 发送进程与之对接。由此可知，不管是物理复制还是逻辑复制，主库，或者源数据库，统一使用 WAL 发送进程来负责处理具体的发送任务。物理备库使用 WAL 接收进程来接收消息包，而逻辑复制使用工作进程来接收消息包。假设你想把源库中的表 T 复制到目标数据库中，在复制之前，表 T 中往往有大量的数据，所以逻辑复制的第一步是数据的同步，就是把表 T 中的初始数据先拷

贝到目标数据库的表 T 中。在数据同步阶段，工作进程会为订阅中的每一张表产生一个同步进程 (sync worker)。假设订阅包含了 10 张表，就会启动 10 个同步进程，每个同步进程负责一张表的数据同步任务。如果你在执行 CREATE SUBSCRIPTION 的命令时指定了 copy_data=off 这个参数，逻辑复制就会跳过表的初始数据同步这个阶段。当表的初始数据传输完毕后，同步进程完成了其历史使命，就会退出。未来的逻辑复制任务由工作进程和主库的 WAL 发送进程一对一的处理，这一点类似物理复制环境中的 WAL 接收进程和 WAL 发送进程的关系。

注意在图 9.4 上，WAL 发送进程包含了一个小方块，叫做解码器，它的任务是把 WAL 记录解析成类似 SQL 的指令发送给目标数据库，这就是为什么逻辑复制的消息包和物理复制的消息包有很大不同的根本原因。解码器作为一个“插件”(plugin)的形式存在，由 WAL 发送进程执行。PostgreSQL 内核提供了缺省的解码器叫做 pgoutput，同时允许第三方的插件存在。在 PostgreSQL 源码目录的 contrib 子目录中，有一个 test_decoding 的解码器示例，向第三方开发者展示了如何开发一个解码器。专门负责逻辑复制的另外一个著名的扩展叫做 pglogical，它的解码器叫做 pglogical_output。解码器的细节比较多，我们不打算进行深入的学习，只需要理解其大致作用即可。你可以把解码器当做一个黑盒子，它的输入是原始的 WAL 记录，它的输出就是很多类似图 9.2 所示的数据包。图 9.5 展示了解码器的基本架构：

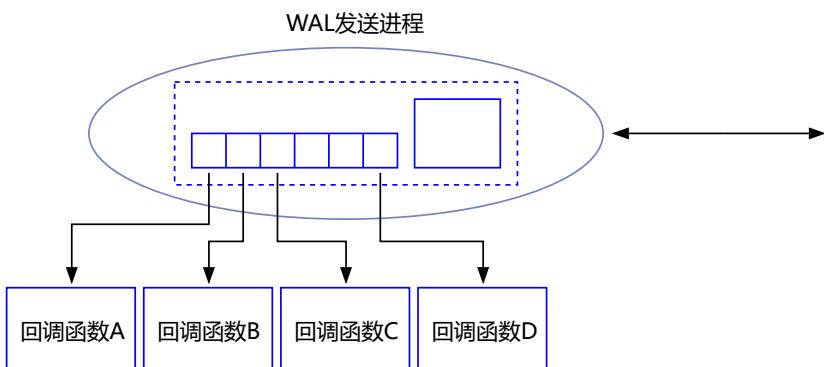


图 9.5：逻辑复制解码器的基本架构

解码器提供了一个框架，如上图中的虚线矩形框所示。在这个框架中，定义了一些内存的数据结构和 21 种回调函数，这些回调函数涵盖了解码器的启动，退出阶段，每一个事务的发起和提交阶段的处理等等，具体细节请参考 output_plugin.h 中的 OutpuPluginCallbacks 这个数据结构。解码器的实现者不一定要实现所有的回调函数功能，只需要实现规定必须有的回调函数即可。WAL 发送进程在解析 WAL 记录时，在事务的不同处理阶段会调用相应的回调函数，完成对 WAL 记录的解码工作。

在理解的逻辑复制的大体架构以后，我们分别从相关的系统视图、后台进程等各个角度来理解逻辑复制的进一步的细节。

9.3 逻辑复制的具体内容

9.3.1 相关的系统视图

当用户执行 CREATE PUBLICATION 后，PostgreSQL 会在系统表 pg_publication 中插入一条记录，同时在 pg_publication_rel 系统表中插入该发布所包含的所有的表，我们在源数据库服务器上执行如下命令：

```
oracle=# select * from pg_publication;
   oid | pubname | pubowner | puballtables | pubinsert | pubupdate | pubdelete | pubtruncate | pubviaroot
-----+-----+-----+-----+-----+-----+-----+-----+-----+
16404 | pub     |      10 | f          | t        | t        | t        | t        | f
(1 row)
/* 查看发布所包含的所有的表 */
```

```
oracle=# SELECT *, prrelid::regclass FROM pg_publication_rel ORDER BY oid;
      oid | prpubid | prrelid | prqual | prattr | prrelid
-----+-----+-----+-----+-----+
    16405 |   16404 | 16386 |        |        | tab1
    16406 |   16404 | 16393 |        |        | tab2
(2 rows)

oracle=# SELECT * FROM pg_publication_namespace; /* 如果表在不同的schema中，可以查看此系统表 */
      oid | pnpubid | pnnspid
-----+-----+
(0 rows)
```

我们可以看到，发布规定了捕获的修改的类型，这就是 pubinsert/pubupdate/pubdelete/pubtruncate 几列所表示的内容。譬如，pubinsert 为 t 表示需要捕获插入操作，f 则表示不需要捕获插入操作。我们可以通过查询 pg_publication_rel 这个系统视图来得到一个发布所包含的表的清单。同样，当创建订阅时，PostgreSQL 也会提供几张系统视图可供查看，我们在目标数据库服务器上执行如下查询：

```
oracle=# SELECT * FROM pg_subscription;
-[ RECORD 1 ]-----+
      oid | 16404
subdbid | 16388
subskiplsn | 0/0
subname | sub
subowner | 10
subenabled | t
subbinary | f
substream | f
subtwophasestate | d
subdisableonerr | f
subpasswordrequired | t
subrunasowner | f
subconninfo | dbname=oracle host=192.168.137.12 user=kevin password=Welcome123
subslotname | sub
subsynccommit | off
subpublications | {pub}
suborigin | any
oracle=# SELECT *, srrelid::regclass FROM pg_subscription_rel ORDER BY 1;
      srsubid | srrelid | srsubstate | srsublsn | srrelid
-----+-----+-----+-----+
     16404 | 16389 | r | 0/1B10268 | tab1
     16404 | 16396 | r | 0/1B10268 | tab2
(2 rows)
```

关于系统视图中各列的含义，你可以参阅官方文档。我们在使用某一列的时候，再进行解释。你稍微注意一下 pg_subscription_rel 这个系统视图的 srsubstate 这一列。它的值为 r，表示 ready 的意思，就是说表的存量数据同步已经成功完成了。逻辑复制必须使用复制槽来进行复制工作，复制槽的创建在你执行 CREATE SUBSCRIPTION 当订阅被成功创建后，我们在源数据库服务器上查看复制槽的信息：

```
oracle=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-----+
slot_name | sub
```

```

plugin          | pgoutput
slot_type       | logical
datoid          | 16385
database        | oracle
temporary       | f
active          | t
active_pid      | 2428
xmin            |
catalog_xmin    | 749
restart_lsn     | 0/1B10440
confirmed_flush_lsn | 0/1B10478
wal_status      | reserved
safe_wal_size   |
two_phase       | f
conflicting     | f

```

请注意上述查询结果中，`slot_type` 表示该复制槽为逻辑复制槽，这和上一节的物理复制槽是不同类型的复制槽，`plugin` 表示该复制槽采用的是 `pgoutput` 这个解码器，这是 PostgreSQL 内核自带的缺省解码器。`active` 这一列为 `t`，表明该复制槽处于活跃状态，通过 `active_pid` 我们知道该复制槽正在被进程号为 2428 的进程所使用，你可以查看一下该进程，实际上就是 WAL 发送进程，由此可知，主库统一使用 WAL 发送进程处理物理复制和逻辑复制这两种情况。

9.3.2 相关的后台进程

在逻辑复制框架中，有四种进程配合完成。首先看第一种进程，在源数据库或者目标数据库服务器上运行如下命令：

```
$ ps -ef | grep postgres | grep logical | grep -v grep
postgres 96272 96265 0 12:44 ? 00:00:00 postgres: logical replication launcher
```

上面的输出中有一个进程，叫做 `logical replication launcher`，进程号是 96272，我们称之为逻辑复制的“启动进程”。逻辑复制的启动进程是逻辑复制中的一个总管，也是一个缺省进程，一旦数据库集群启动，它就自动运行。启动进程会周期性地检查系统视图 `pg_subscription`，判断是否有新的订阅产生，或者老的订阅发生了变化。一旦该系统视图中的订阅发生了变化，启动进程就会启动第二种进程：`logical replication apply worker`，我们可以称之为逻辑复制的“工作进程”。工作进程和订阅是一对一的关系，即，你创建多少个订阅，就有多少个工作进程与之对应。当用户发出了“`CREATE SUBSCRIPTION`”命令之后，启动进程就会创建工作进程。所以你执行完该命令后，在目标数据库上执行 `ps` 命令，就会看到工作进程的身影：

```
$ ps -ef | grep postgres | grep logical | grep -v grep
postgres 34567 34561 0 12:46 ? 00:00:00 postgres: logical replication launcher
postgres 34639 34561 0 12:49 ? 00:00:00 postgres: logical replication apply worker for
subscription 16399
```

当然，启动进程并不会直接 `fork` 工作进程。启动进程会通知主进程来创建工作进程，你从这两个进程的父进程号都是 34561 这一点可以看出它们是兄弟关系，不是父子关系。工作进程类似物理复制中的 `WAL` 接收进程，它的职责是把源数据库的更新数据施加在目标数据库上。该进程启动后，面临一个问题：源表里面可能已经有数据了，这些数据被称为“初始数据”。这里面就存在一个初始数据同步的问题，就是首先要把这些初始数据拷贝到目标表中。工作进程会遍历订阅，为每一张表启动一个同步进程 (`sync worker`)，这些同步进程的使命很明确：就是尽快地把初始数据拷贝到目标表中。一旦它的使命完成，就自动退出，所以对于很小的表，你不容易看到它的身影。同步进程会使用 PostgreSQL 的 `COPY` 命令把原表中的存量数据拷贝到目标数据库中。你查阅一

下 COPY 命令的用法，就会发现 COPY 命令的主要作用是把表中的数据拷贝到磁盘上的某一个文件中，或者把磁盘上某个文件的数据拷贝到表中。COPY 命令支持回调函数 (CALLBACK) 的方式，可以把表的数据通过网络传输到另外一台机器上。同步进程就是利用 CALLBACK 方式下的 COPY 命令进行数据的远程复制的。更新进程和同步进程分别独立地和源数据库连接，在源数据库端，由 WAL 发送进程与它们对接。在图 9.4 中，因为我们创建的订阅里只有两张表，所以 PostgreSQL 启动了两个同步进程，在源数据库端，有 3 个 WAL 发送进程和它们对接。当初始数据同步完成后，就由工作进程独自一人负责后续的增量数据的更新操作了。

总结一下，目标数据库的逻辑复制启动进程，工作进程，同步进程和源数据库的 WAL 发送进程，共同组成了逻辑复制的核心后台进程。对比一下物理复制，则有主库的 WAL 发送进程，备库的 WAL 接收进程和恢复进程三种进程。在逻辑复制中，目标数据库是可读可写的，没有恢复进程，因为恢复进程在目标数据库启动时完成了任务就退出了，它不需要再目标数据库正常运行的时候存在，就如果你在主库上也看不见恢复进程一样。

9.3.3 初始数据的同步

我们知道，逻辑复制的第一步是把源数据库的表的初始数据复制到目标端的表中，让两者处于同一起跑线。我们称这个阶段为初始数据的同步，它是由工作进程和同步进程协同完成的。工作进程诞生以后，会检查它所负责的发布里面有多少张表，为每一张表派生出一个同步进程。一个同步进程只负责一张表的数据同步，同步进程也使用逻辑复制槽和源数据库进行网络连接，源数据库由 WAL 进程负责接待来自目标数据库的工作进程和同步进程的请求。在系统视图 pg_subscription_rel 中有一列 srsubstate，该列记录着一张表的数据同步状态，一共有 5 种状态，用 5 个字母来表示，其意义如下：

- i 表示处于初始化 (initialize) 状态。
- d 表示处于数据拷贝 (data is being copied) 状态。
- f 表示数据已经拷贝完成 (finished table copy)。
- s 表示数据已经同步了 (synchronized)。
- r 表示就绪 (ready) 状态。

它们在源代码中的定义如下：

```
/* in src/backend/catalog/pg_subscription_rel_d.h */
#define SUBREL_STATE_INIT          'i'   /* initializing (sublsn NULL) */
#define SUBREL_STATE_DATASYNC      'd'   /* data is being synchronized */
#define SUBREL_STATE_FINISHEDCOPY  'f'   /* tablesync copy phase is completed */
#define SUBREL_STATE_SYNCDONE      's'   /* synchronization finished in front of apply */
#define SUBREL_STATE_READY         'r'   /* ready (sublsn set) */
```

其中就绪状态是最终的成功状态，当一张表进入这种状态后，同步进程就完成了它的历史使命，自动退出了。图 9.6 展示了更新进程和同步进程配合的过程：

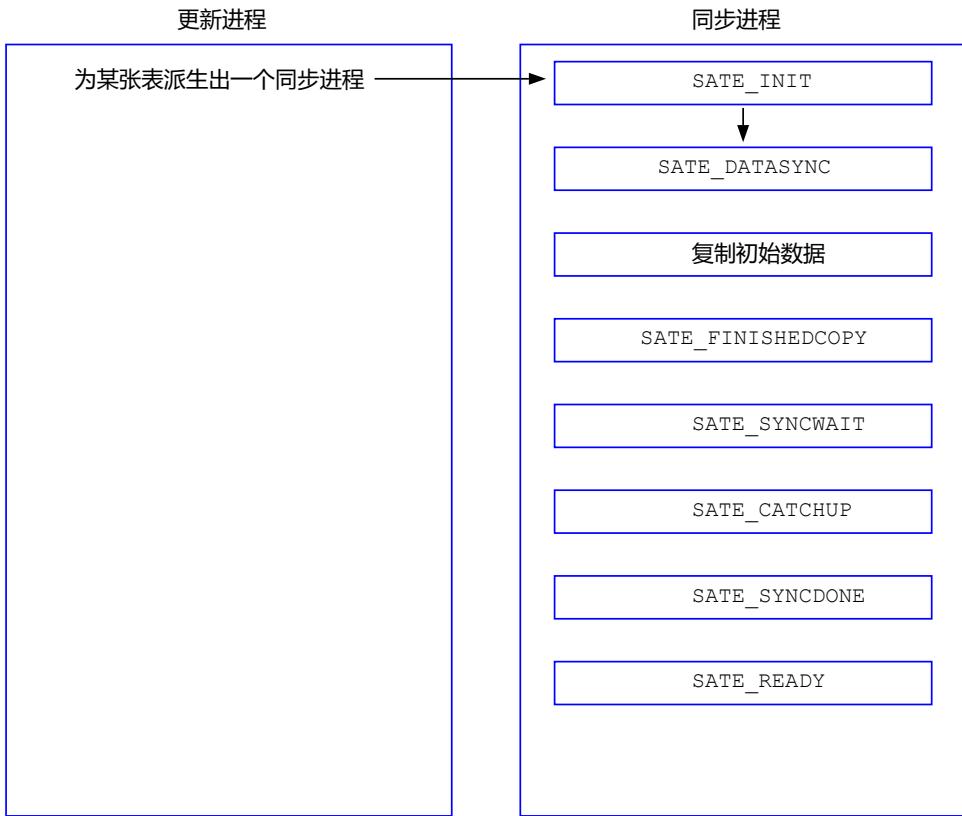


图 9.6: 逻辑复制的初始数据同步的过程

当工作进程发现一张表需要同步时，就启动了一个同步进程。此时，同步进程的状态为 INIT，即初始状态。工作进程和同步进程的交互方式分为两种途径，一种是基于磁盘的，就是系统视图 pg_subscription_rel 中的 srsubstate 列。在这种方式中，进程会把状态写入到磁盘上，供对方检查，我们在 pg_subscription_rel 中可以看到这种状态的变化。第二种交流方式是基于共享内存的，我们在 pg_subscription_rel 中就看不到了。

同步进程启动后开始使用 COPY 命令拷贝数据，这个过程的状态被称为 DATASYNC。当数据拷贝完成后，同步进程把状态设置为 FINISHEDCOPY。当同步进程走到这一步以后，就把共享内存中自己的状态改为 SYNCWAIT，表示请求工作进程进行检阅后发出下一步的指令。工作进程会周期性地检查每个同步进程是否处于 SYNCWAIT 状态，如果找到了，就把这个同步进程在共享内存中的状态改为 CATCHUP，即“追赶”的意思，并指定一个 LSN 作为追赶的终点。同步进程发现自己的状态从 SYNCWAIT 变成了 CATCHUP 以后，就继续从源数据库获取数据，直至达到或者超过了工作进程指定的 LSN，然后把自己的状态改为 SYNCDONE，并退出。工作进程然后不停地循环检查每一个同步进程的状态，直到某一个同步进程的状态变成 SYNCDONE 为止。工作进程发现了 SYNCDONE 状态以后，把这个同步进程对应的表的状态在系统视图 pg_subscription_rel 中改为 READY，并接管后续的数据复制任务。所以我们可以根据系统视图 pg_subscription_rel 中的 srsubstate 列的状态观察到数据同步走到了哪一步。

同步进程使用 COPY 命令完成表的初始数据复制任务，如果初始化数据量很大，数据同步这个环节可能需要花费很长时间。我们在等待的过程中迫切希望知道它的进度。除了查询系统视图 pg_subscription_rel 以外，PostgreSQL 还给出了显示 COPY 进度的系统视图，如下所示：

```
=# SELECT relid::regclass, command, type, bytes_processed, bytes_total,
       tuples_processed, tuples_excluded FROM pg_stat_progress_copy;
 relid   |  command  |  type  | bytes_processed | bytes_total | tuples_processed | tuples_excluded
-----+-----+-----+-----+-----+-----+-----+
 copy_tab | COPY FROM | FILE |      52 |      52 |          5 |          5
```

(1 row)

在 pg_stat_progress_copy 系统视图中，tuples_processed 这一列表示已经复制了多少条记录。我们再使用 SELECT count(*) FROM xxxx 就可以知道表一共有多少条记录。有了这两个指标，我们很容易知道 COPY 命令目前的进度，以及是不是在继续往前移动。

<https://www.percona.com/blog/logical-replication-decoding-improvements-in-postgresql-13-and-14/>

9.3.4 逻辑复制槽

本小节讨论逻辑复制槽。

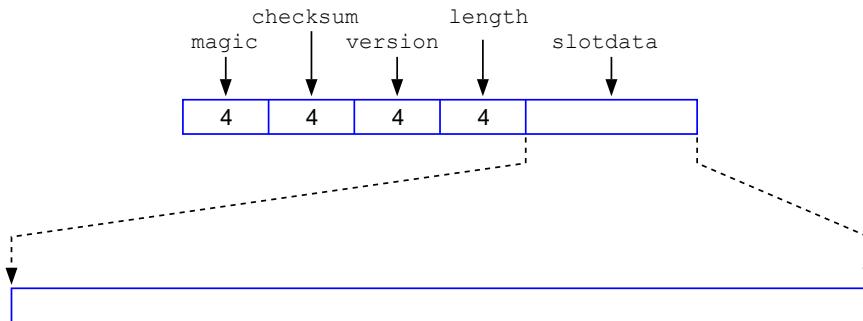


图 9.7：从备库进行逻辑复制

9.3.5 冲突的处理

我们知道不同的事务修改的记录条数可能不同，即事务的大小不同。我们通过一个实验来体验一下事务的大小的含义。首先我们创建一张测试表，往里面插入 5 条记录：

```
oracle=# create table state(id int, name char(2));
CREATE TABLE
oracle=# insert into state values(0,'MA'),(1,'TX'),(2,'CO'),(3,'PA'),(4,'WM');
INSERT 0 5
oracle=# select * from state;
 id | name
----+---
 0 | MA
 1 | TX
 2 | CO
 3 | PA
 4 | WM
(5 rows)
```

紧接着我们执行一条 UPDATE 操作，更新里面的 3 条记录。在更新之前和更新之后，我们都记录一下当前的 LSN。

```
oracle=# select pg_current_wal_lsn();
pg_current_wal_lsn
-----
0/2A032B90
```

```
(1 row)

oracle=# update state set name='XX' where id>1;
UPDATE 3
oracle=# select pg_current_wal_lsn();
pg_current_wal_lsn
-----
0/2A032C90
(1 row)
```

知道了这个事务发生之前和之后的 LSN，该事务产生的全部 WAL 记录必然都在这个范围之内。我们使用 pg_waldump 工具打印出这段范围内的 WAL 记录，结果如下：

```
postgres@ubuntui:~$ pg_waldump -s 0/2A032B90 -e 0/2A032C90
rmgr: Heap      len (rec/tot):    70/    70, tx:        760, lsn: 0/2A032B90, prev 0/2A032B58,
desc: HOT_UPDATE old_xmax: 760, old_off: 3, old_infobits: [], flags: 0x20, new_xmax: 0,
new_off: 6, blkref #0: rel 1663/16384/16414 blk 0
rmgr: Heap      len (rec/tot):    70/    70, tx:        760, lsn: 0/2A032BD8, prev 0/2A032B90,
desc: HOT_UPDATE old_xmax: 760, old_off: 4, old_infobits: [], flags: 0x20, new_xmax: 0,
new_off: 7, blkref #0: rel 1663/16384/16414 blk 0
rmgr: Heap      len (rec/tot):    70/    70, tx:        760, lsn: 0/2A032C20, prev 0/2A032BD8,
desc: HOT_UPDATE old_xmax: 760, old_off: 5, old_infobits: [], flags: 0x20, new_xmax: 0,
new_off: 8, blkref #0: rel 1663/16384/16414 blk 0
rmgr: Transaction len (rec/tot):    34/    34, tx:        760, lsn: 0/2A032C68, prev 0/2A032C20,
desc: COMMIT 2024-01-21 19:40:01.049284 UTC
postgres@ubuntui:~$
```

由上可知，这个事务产生了 4 条 WAL 记录。因为它修改了三条记录，每条被修改的记录都会产生一条对应的 WAL 记录。因为这个事务被成功提交了，所以最后一条 WAL 记录是 COMMIT 类型的。我们可以用图 XX 表示上述实验的结果：

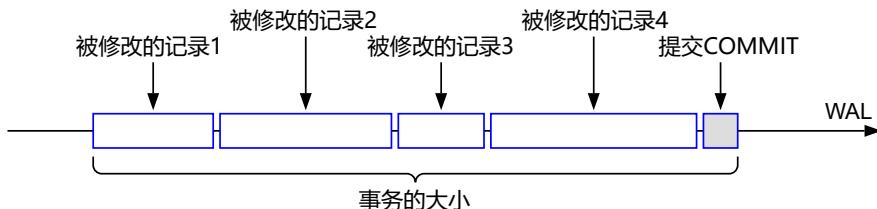


图 9.8: 事务的大小

由此可见，如果一个事务只修改了一条记录，它产生的 WAL 记录就很少。如果一个事务修改了成千上万条记录，它产生的 WAL 记录就有成千上万个。事务的大小就可以由它产生的 WAL 记录的字节数来衡量。小事务和大事务的尺寸相差巨大。WAL 发送进程在进行逻辑解码的时候，它会一直解析 WAL 记录，直到遇到 COMMIT 类型的 WAL 记录，它才会一股脑地把这个事务全部的内容发送给目标数据库。对于巨大的事务，WAL 发送进程在遇到 COMMIT 之前，必须申请很多内存来记录该事务的所有修改。这就造成了一个问题：巨大的事务可能消耗掉 WAL 发送进程上百 G 的内存，严重影响了主库的性能。为此，PostgreSQL 设置了参数 logical_decoding_work_mem 来控制解码大事务所需要的内存消耗。这个参数表示逻辑复制解码所需要内存的上限，缺省值是 64MB。如果 WAL 发送进程在逻辑解码过程中，某一个事务所占用的内存大小超过了这个值，WAL 发送进程就会把这个事务的数据写入到磁盘上，等遇到 COMMIT 时候，再把数据从磁盘上读入内存，发送给目标数据库。我们可以通过

系统视图 pg_stat_replication_slots 来观察写入到磁盘数据的情况：

```
postgres=# select * from pg_stat_replication_slots;
 slot_name | spill_txns | spill_count | spill_bytes | stream_txns | stream_count | stream_bytes |
 total_txns | total_bytes | stats_reset
-----+-----+-----+-----+-----+-----+-----+
(0 rows)
```

其中我们需要注意的是 spill_txns/spill_count/spill_bytes 这三列。

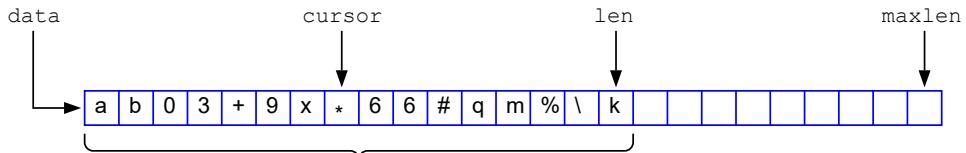


图 9.9: 逻辑复制的消息包的格式

第一个字节是消息类型，只有两种合法的值，w 和 k。k 表示 keep-alive，w 表示真正的数据。第二个字节是工作，它的类型由 LogicalRepMsgType 定义。

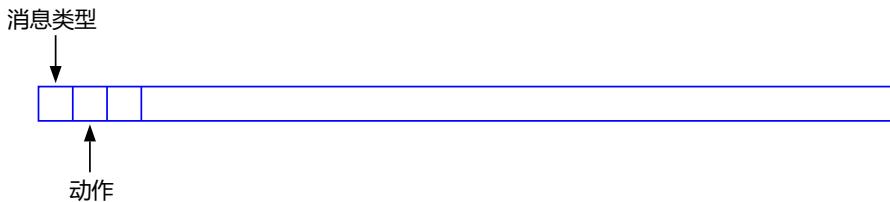


图 9.10: 逻辑复制的消息包的格式

```
typedef enum LogicalRepMsgType
{
    LOGICAL_REP_MSG_BEGIN = 'B',
    LOGICAL_REP_MSG_COMMIT = 'C',
    LOGICAL_REP_MSG_ORIGIN = 'O',
    LOGICAL_REP_MSG_INSERT = 'I',
    LOGICAL_REP_MSG_UPDATE = 'U',
    LOGICAL_REP_MSG_DELETE = 'D',
    LOGICAL_REP_MSG_TRUNCATE = 'T',
    LOGICAL_REP_MSG_RELATION = 'R',
    LOGICAL_REP_MSG_TYPE = 'Y',
    LOGICAL_REP_MSG_MESSAGE = 'M',
    LOGICAL_REP_MSG_BEGIN_PREPARE = 'b',
    LOGICAL_REP_MSG_PREPARE = 'P',
    LOGICAL_REP_MSG_COMMIT_PREPARED = 'K',
    LOGICAL_REP_MSG_ROLLBACK_PREPARED = 'r',
    LOGICAL_REP_MSG_STREAM_START = 'S',
    LOGICAL_REP_MSG_STREAM_STOP = 'E',
    LOGICAL_REP_MSG_STREAM_COMMIT = 'c',
    LOGICAL_REP_MSG_STREAM_ABORT = 'A',
}
```

```

LOGICAL_REP_MSG_STREAM_PREPARE = 'p'
} LogicalRepMsgType;

```

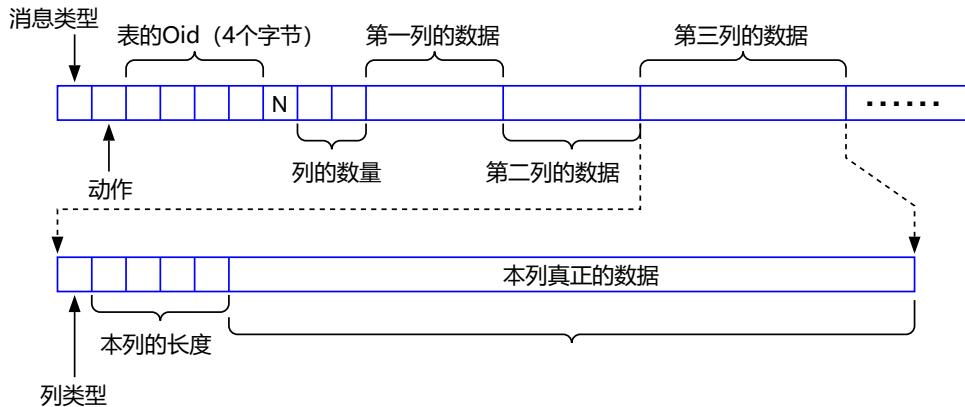


图 9.11: INSERT 消息包的格式

<https://www.postgresql.fastware.com/blog/addressing-replication-conflicts-using-alter-subscription-skip#text=%3E%20Wrapping>
 ,The%20ALTER%20SUBSCRIPTION%20SKIP%20command,to%20indicate%20the%20failed%20transaction.

```

postgres=# select * from pg_stat_subscription_stats;
 subid | subname | apply_error_count | sync_error_count | stats_reset
-----+-----+-----+-----+-----
(0 rows)

pg_replication_origin_advance()

```

9.4 从备库进行逻辑复制

逻辑复制的体系架构如下图所示：

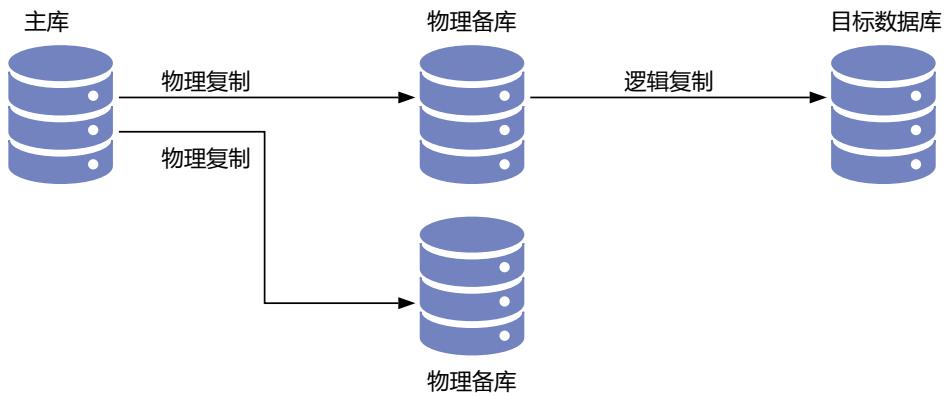


图 9.12: 从备库进行逻辑复制

第十章 内存池管理和常用数据结构

数据库软件是吃内存的大户,为了最高的性能,它可以把一台服务器的所有内存几乎吃尽。本章对 PostgreSQL 频繁使用的内存管理和底层数据结构,如动态哈希表等,进行剖析。对于这些技术的理解,是我们深入 PostgreSQL 内核其它技术的基础。

10.1 内存池子系统

PostgreSQL 内核是采用 C 语言编写的,C 语言作为排名第一的系统软件开发语言,它给了程序员最大的内存管理自由度,也带来了很多潜在的问题,其中最头疼的问题是内存泄漏。使用 C 语言开发的高质量软件往往采用内存池(memory pool)技术来减少内存泄漏,PostgreSQL 也不例外。本节介绍 PostgreSQL 内存池子系统的设计思想和具体实现的关键细节。

C 语言初学者都知道,在 C 程序中申请和释放内存使用的是 malloc() 和 free() 等系统调用(system call),下面简单的 C 语言代码演示了这两个内存管理函数的基本使用。

```
$ cat m.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    char* mem = (char*)malloc(8192);
    if(NULL != mem) {
        mem[0] = mem[1] = 'A'; mem[2] = 0x00;
        printf("mem is [%s]\n", mem);
        free(mem);
    }
    return 0;
}
$ gcc -Wall m.c -o m
$ ./m
mem is [AA]
```

上述代码的逻辑非常简单:通过 malloc() 一次性从操作系统那里申请 8192 个字节,紧接着头两个字节被写入了'A',第三个字节存放一个 0,截断字符串,然后把这个字符串打印出来,就是 [AA],最后通过 free() 释放内存,程序退出。malloc() 和 free() 是成对出现的,有 malloc(),就必然有 free(),否则就容易造成内存泄漏。这一对函数的用法非常简单,初学者一学就会。不过我们需要注意一点:如果你用 malloc() 申请一块内存,保存在指针 mem 中,释放该块内存时,传入到 free() 函数中的参数必须是 mem,也就是指向这块内存的开始地址的指针,任何别的内存地址都不行,即使这个地址是 mem 表示的内存的一部分也不行,这是因为底层的操作系统维护一套完整的数据结构来管理和跟踪通过 malloc() 申请的内存,如果 free() 函数拿到的地址不是内存的开始地址,操作系统就无法获得完整的管理数据。

这两个函数均是系统调用,其开销相对较大,为了提高软件的性能,就要尽量减少对其调用的次数。因为 malloc(10) 和 malloc(10000) 的开销并无多大差别,所以为什么不一次性申请一块大的内存,然后再慢慢切蛋糕享用呢?对,这个想法其实就是内存池技术的核心思想,通俗地说就是“一次批发,多次零售”。内存池管理器一次性使用 malloc() 分配大块内存,类似批发,其后的小额内存申请会从这个大块内存中不断切出尺寸合适的内存片,类似零售。而这种切蛋糕的操作只是内存指针的位置进行简单地移动,不再涉及开销很大的系统调用,所以非常高效。当这块大内存上所有的内存都不再使用时,通过 free() 一次性把这块内存释放掉即可。图 6.1 展

示了内存池的基本思想：

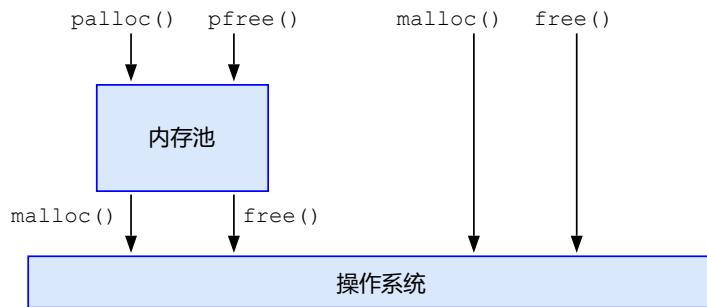


图 10.1：内存池的思想

从上图中我们可以看到，当通过内存池来申请和释放内存时，不再使用 `malloc()` 和 `free()`，而是通过内存池管理系统提供的 `palloc()` 和 `pfree()`。很显然，这两个函数是模拟 `malloc()` 和 `free()` 的，用法也差不多。因为 `palloc()` 和 `pfree()` 的操作只是单纯的指针位置的计算和移动，基本上不会使用系统调用，所以效率要比 `malloc()` 和 `free()` 高很多。内存池是内存使用者和操作系统之间的一个内存中介，最终的内存还是来自操作系统，所以内存池在底层依然使用 `malloc()` 和 `free()` 等系统调用向操作系统申请或者释放内存。

10.1.1 内存池的整体结构

我们可以把 PostgreSQL 的内存池理解为一堆内存块 (block) 的集合，它们都是通过 `malloc()` 分配的，最终通过 `free()` 释放给操作系统。这些内存块在内存池中组成了一个双向链表，便于管理。图 6.2 展示了一个内存池的整体结构。

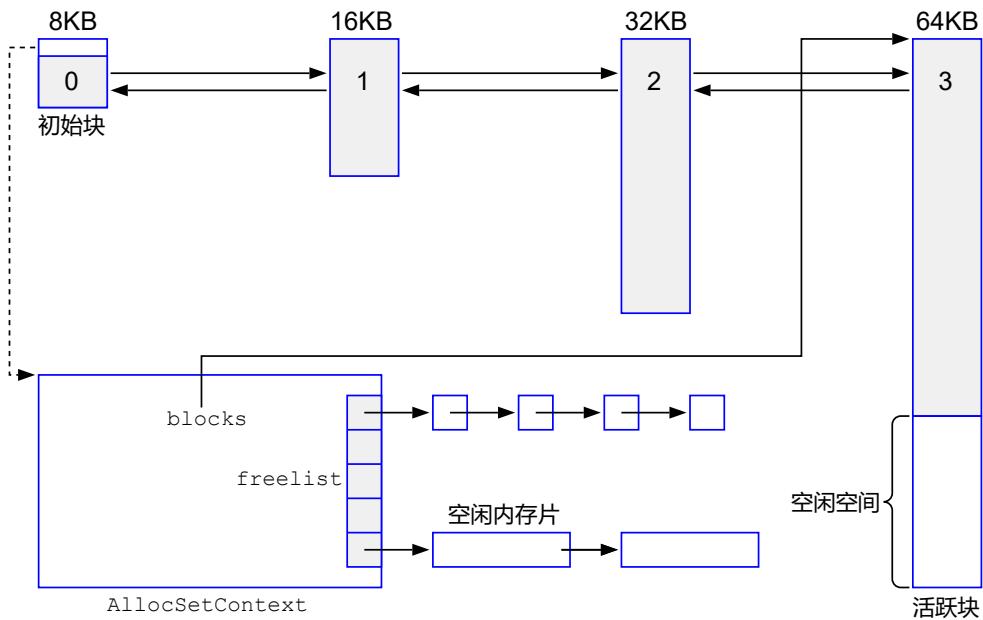


图 10.2：内存池的整体结构

图中的内存池里一共有 4 个内存块，最左边的 0 号内存块是内存池刚创建时用 `malloc()` 分配的第一个内存块，被称为初始块 (initial block)。初始块的大小由整型变量 `initBlockSize` 控制，一般为 1024 字节或 8192 字节。当内存池中的空闲内存不够外面的内存申请时，内存池就会分配新的内存块。内存池使用三个变量来控制每次

分配的内存块的大小：initBlockSize 控制初始块的大小，它是内存块的下限；nextBlockSize 控制下一次分配的内存块的大小；而 maxBlockSize 是内存块的上限，防止内存块的体积无限增长。

内存池的缺省实现策略是：每当通过 malloc() 分配一个新的内存块时，其体积都比前一次分配的内存块大一倍。例如：初始块的大小是 8KB，则第二个内存块的大小是 16KB，第三块是 32KB，第四块是 64KB，依次类推，直至新内存块的体积等于 maxBlockSize，就不再增加了，以后就按照 maxBlockSize 大小分配新的内存块。

图 6.2 中的所有内存块形成一条双向链表，3 号块为在头部，初始块(0 号块)永远在尾部。双向链表头部的内存块被称为活跃块 (active block)，活跃的含义是只有该内存块上才有空闲内存可供分配，最右边的 3 号内存块便是活跃块。初始块和其后诞生的内存块（包括活跃块）有一点不同：在初始块的头部有个“控制头”(MemoryContextData)，这是一个 C 语言的结构体，用于控制整个内存池的分配、销毁和运行时的行为。MemoryContextData 是一个通用的控制头，为了适用不同的内存需求，PostgreSQL 在这个通用的控制头的基础上，产生了不同的内存管理的具体实现策略，其中缺省的策略叫做 AllocSet，其对应的数据结构叫 AllocSetContext。AllocSetContext 和 MemoryContextData 的关系类似 C++ 中的子类和父类的关系：MemoryContextData 是父类，而 AllocSetContext 是子类，它继承了父类，我们在后文中两者的数据结构定义中就会很清楚地看到这一点：MemoryContextData 实际上是 AllocSetContext 的第一个成员变量。除了缺省的 AllocSet 实现策略以外，PostgreSQL 还支持 Generation, SLAB 两种内存管理策略。本节重点研究缺省的 AllocSet 实现方式，对于其它两种内存分配方式在本节的最后进行讨论。

我们可以这样通俗地理解内存池：内存池是一堆内存块组成的，这些内存块通过双向链表被组织成一个整体。在第一个内存块（初始块）上有一个控制头来管理这些内存块的有序使用，譬如，AllocSetContext 中有一个指针变量 blocks，始终指向内存块双向链表的头部，即当前的活跃块。这个控制头虽然体积很小，但是类似牵着牛鼻子的绳子，它是理解内存池的关键所在。图 6.2 中左下角的大矩形就是 AllocSetContext，它实际上只是初始块头部很小的一块区域，不过为了显示其细节，图中进行了放大。正因为初始块和其后诞生的其它内存块有这个区别，所以产生了内存池的删除 (delete) 和重置 (reset) 两种操作的概念，删除内存池很好理解，就是彻底把一个内存池所有的内存块都通过 free() 释放掉，这个内存池也就不复存在了；而重置内存池就是释放掉除初始块以外的所有内存块，整个内存池回到了刚刚诞生的初始状态，即只有一个初始块。

内存池管理器对外提供 palloc() 和 pfree() 等内存分配和释放的接口函数来代替 malloc() 和 free()，所以在 PostgreSQL 中需要通过内存池申请和释放内存时，统一使用 palloc() 和 pfree()。下面列出了这两组函数的定义，你会发现这两组函数长的几乎一模一样。

```
void* malloc(size_t size);
void free(void *pointer);
void* palloc(size_t size);
void pfree(void *pointer);
```

当申请者通过 palloc() 函数向内存池申请内存时，内存池管理器返回的内存叫内存片 (chunk)，内存片是从某个内存块中切下来的一小片内存。内存池管理器在寻找合适的空闲内存时，只在活跃块中寻找，因为双向链表上其它的内存块没有空闲内存了。为什么有这个规律呢？其原因很好理解：在内存池创建伊始，只有一个初始块，所以初始块是内存池初期的活跃块。当使用者向内存池申请内存时，只要活跃块上有足够的空闲内存，当然会从这里进行内存分配。当目前的活跃块中的空闲内存不够时，内存池才会再次创建一个更大的新块，并且把它放在内存块双向链表的头部，作为新的活跃块，所有非活跃块可以被称为老块 (old block)。因为活跃块只在老块没有足够的空闲内存时才会诞生，所以内存池中只有活跃块才可能存在空闲内存供未来的分配，这就是“活跃”的含义。产生新的活跃块时，双向链表中前一个老块上的空闲内存不够申请者使用，但可能还有一些零碎。这些零碎的空闲内存也不会浪费，它的具体使用等我们分析源代码的时候你就会看到是如何处理的。活跃块永远是双向链表中的第一个块，内存池只在活跃块里面寻找空闲内存，不用费力地遍历双向链表，在每一个老块上寻找，提高了搜索空闲内存的效率。

内存池，内存块和内存片是内存池系统中的三个基本概念。它们之间的关系可以总结为：

- 一个内存池包含一个或者多个内存块。在内存池被创建后，只有一个初始块。随着内存申请次数的增加，

内存池会创建出更多的内存块。只有活跃块上才存在可供分配的空闲内存。

- 任何一个内存块只属于一个内存池，只有内存块才能通过 `malloc()` 和 `free()` 进行内存的实际创建和释放。
- 一个内存块包含一个或者多个内存片。内存片才是返回给申请者的内存，它是某个内存块的一部分。

内存片通过 `pfree()` 释放给内存池后，因为它只是内存块的一部分，不能够通过 `free()` 释放掉，内存池会组织一个空闲内存数组，把释放回本内存池的内存片放在其中，供未来的内存申请使用。图 6.2 中的 `AllocSetContext` 中重要的成员变量 `freelist`，就是一个空闲内存片数组，其每个成员都是一个指针，指向由尺寸不同的空闲内存片组成的单向链表。这些空闲内存片是申请者通过 `pfree()` 退还给内存池的，可供未来的内存片申请使用。

对内存池的操作有以下几种。

- 内存池的创建 (Create)：通过 `malloc()` 创建初始块，把头部设置为 `AllocSetContext` 结构，并初始化其成员变量。
- 内存片的分配 (Alloc)：从内存池中申请一个内存片，返回指向该内存片的指针或者 `NULL`（如果失败），类似 `malloc()`。
- 内存片的释放 (Free)：把从某个内存池中申请的内存片返回给该内存池，供后来的申请者继续使用，类似 `free()`。
- 内存片的重新分配 (Realloc)：调用者嫌手里的内存片尺寸不合适，还给内存池，再从池中申请一个大小合适的新内存片。
- 内存池的删除 (Delete)：彻底销毁一个内存池，其中的内存块一个不留。内存池删除后，指向该内存池的指针就无效了。
- 内存池的重置 (Reset)：彻底销毁该内存池中除初始块以外的所有内存块，再把初始内存块重置为该内存池创建时的初始状态。

10.1.2 内存池的相关数据结构

内存池子系统中有几个数据结构：`MemoryContext`（内存上下文）、`AllocSetContext`（分配集合上下文）、`AllocBlock`（管理内存块的结构体）和 `MemoryChunk`（管理内存片的结构体），下面依次介绍它们的含义和相互关系。

10.1.2.1 内存上下文 `MemoryContext`

`MemoryContext` 是指向一个结构体 `MemoryContextData` 的指针。PostgreSQL 对结构体命名时，喜欢用 `xxxxData` 表示真正的结构体，用 `xxxx` 表示指向该结构体的指针。在后面的讨论中，为了减少单词的长度，除非特别说明，本书把结构体和指向该结构体的指针视为同一个概念，可能会交叉使用相关术语，即 `MemoryContext` 和 `MemoryContextData` 表示同一个东西，其余类似的概念也采用这个术语规则。

`MemoryContext` 是内存池的一种抽象的控制头，其具体的实现是 `AllocSetContext` 结构体。内存池的创建和销毁，内存片的分配和释放等工作由 `AllocSetContext` 来控制。这种设计类似 C++/Java 等面向对象程序设计语言中的父类和子类的关系：`MemoryContext` 是父类，`AllocSetContext` 是子类。`MemoryContext` 和 `MemoryContextData` 的定义如下：

```
/* in src/include/nodes/nodes.h */
#define pg_node_attr(...)

/* in src/include/nodes/memnodes.h */
typedef struct MemoryContextData *MemoryContext;
typedef struct MemoryContextData
{
    pg_node_attr(abstract)      /* there are no nodes of this type */
    NodeTag        type;        /* identifies exact kind of context */
```

```

/* these two fields are placed here to minimize alignment wastage: */
bool      isReset;          /* T = no space allocoed since last reset */
bool      allowInCriticalSection; /* allow palloc in critical section */
Size      mem_allocated;    /* track memory allocated for this context */
const MemoryContextMethods *methods; /* virtual function table */

MemoryContext parent;        /* NULL if no parent (toplevel context) */
MemoryContext firstchild;   /* head of linked list of children */
MemoryContext prevchild;    /* previous child of same parent */
MemoryContext nextchild;    /* next child of same parent */
const char *name;           /* context name (just for debugging) */
const char *ident;          /* context ID if any (just for debugging) */
MemoryContextCallback *reset_cbs; /* list of reset/delete callbacks */

} MemoryContextData;

```

MemoryContext 的重要成员变量的含义如下：

- pg_node_attr - 空定义，可以忽略。
- type - 内存池的类型，恒定为 T_AllocSetContext，其目的是快速判断一个指针的类型，可以参考 nodes.h 中的 IsA()。
- isReset - 该内存池是否被重置过。内存池被重置后只有一个初始块，没有任何内存片被分配出去，isReset 为 true。一旦有内存片被分配出去，isReset 则为 false。该变量用于某些操作中快速判断内存池的状态。
- mem_allocated - 本内存池中所有内存块的大小之和，即本内存池的总体积，单位是字节。
- methods - 内存池创建、销毁、内存片分配和释放等操作的函数指针。其定义后文会讨论。
- name - 用于调试内存池使用。因为内存池不在内部保存它，仅仅一个指针指向该字符串，所以它必须是一个常量字符串，如”MyPool”，不能是一个字符串变量。
- ident - 也是用于调试时使用，这里可以忽略。

因为 PostgreSQL 在不同的场景中都会频繁使用内存池，譬如一个复杂的查询被分解成若干简单的子查询后，每一个子查询都可能有独立的内存池。为了管理这些有联系的内存池，需要把它们有效组织起来，避免遗漏。MemoryContext 有四个 MemoryContext 指针：parent、firstchild、prevchild 和 nextchild。我们不难想象，多个内存池通过这四个指针就可以形成一个树形结构，如图 6.3 所示，其中每一个矩形都表示一个内存池。我觉得 prevchild 和 nextchild 变量的名字起的不好。它们实际上指向了自己的兄弟，所以应该叫做 pre_sibling 和 next_sibling 似乎更明晰。

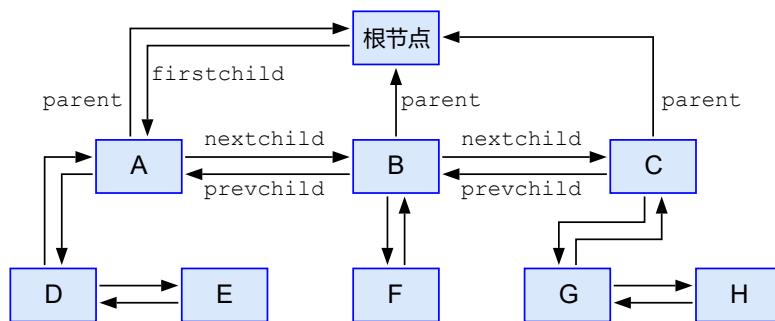


图 10.3: 内存池系统的树形结构

为了访问某个内存池，首先必须拿到一个指向初始块头部的 MemoryContext 指针。在 PostgreSQL 源代码中设置了若干 MemoryContext 指针的全局变量用于记录一些常用内存池，列表如下：

```

/* in src/backend/utils/mmgr/mcxt.c */

```

```

MemoryContext TopMemoryContext = NULL;
MemoryContext CurrentMemoryContext = NULL;
MemoryContext ErrorContext = NULL;
MemoryContext PostmasterContext = NULL;
MemoryContext CacheMemoryContext = NULL;
MemoryContext MessageContext = NULL;
MemoryContext TopTransactionContext = NULL;
MemoryContext CurTransactionContext = NULL;

```

在数据库实例运行期间，每一个服务器进程都有被组织成树形结构的若干内存池。其中根节点的内存池被称为 `TopMemoryContext`，主进程启动时就创建了它，并被子进程自动继承，参见 `MemoryContextInit()` 函数。该内存池贯穿该服务器进程的生命周期，在进程死亡之前才被销毁。在 `TopMemoryContext` 内存池下面还有若干子内存池，具有不同的生命周期。`CurrentMemoryContext` 代表当前的内存池，对内存池进行各种操作时，如果不指定内存池指针，就在 `CurrentMemoryContext` 指向的内存池中进行操作。譬如，`palloc()` 函数只有一个表示内存大小的 `size` 变量，并没有指明要从哪个内存池中申请，实际上它就是在 `CurrentMemoryContext` 指向的内存池中申请内存的。这些常见内存池在数据库实例中形成图 6.4 所示的树形结构。

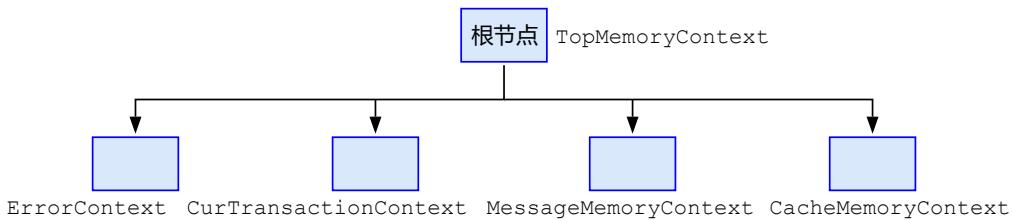


图 10.4: 数据库实例的内存池结构

PostgreSQL 14 版本提供了一个系统视图 `pg_backend_memory_contexts`，可以查看当前服务器进程的内存池使用情况，下面的实验演示了该系统视图的具体使用。

```

postgres=# select * from pg_backend_memory_contexts where name='TopMemoryContext';
-[ RECORD 1 ]+-
name      | TopMemoryContext
ident     |
parent    |
level     | 0
total_bytes | 97696
total_nblocks | 5
free_bytes   | 13152
free_chunks   | 19
used_bytes   | 84544
postgres=# select count(1) from pg_backend_memory_contexts where parent='TopMemoryContext';
 count
-----
 23
(1 row)

```

结合 `MemoryContextData` 的定义，该系统视图非常容易理解。由上可知，`TopMemoryContext` 内存池一共有 5 个内存块 (`total_nblocks`)，总体积为 97696 字节 (`total_bytes`)，使用了 84544 字节 (`used_bytes`)，层数为 0 (`level`)。内存池树的根节点的层数为 0，往下依次增加。根内存池 `TopMemoryContext` 旗下目前有 23 个子内存池，想了解它们的具体内容，你可以继续查询 `pg_backend_memory_contexts` 系统视图。

10.1.2.2 分配集上下文 AllocSetContext

结构体 AllocSetContext 是 MemoryContext 具体实现方案的其中之一，也是 MemoryContext 的缺省实现方案，本节中我们只讨论该实现方案。其定义如下：

```
/* in src/backend/utils/mmgr/aset.c */
#define ALLOCSET_NUM_FREELISTS 11
typedef struct AllocSetContext
{
    MemoryContextData header; /* Standard memory-context fields */
    AllocBlock blocks; /* head of list of blocks in this set */
    MemoryChunk *freelist[ALLOCSET_NUM_FREELISTS]; /* free chunk lists */
    Size initBlockSize; /* initial block size */
    Size maxBlockSize; /* maximum block size */
    Size nextBlockSize; /* next block size to allocate */
    Size allocChunkLimit; /* effective chunk size limit */
    AllocBlock keeper; /* keep this block over resets */
    int freeListIndex; /* index in context.freelists[], or -1 */
} AllocSetContext;

typedef AllocSetContext *AllocSet;
```

结构体 AllocSetContext 成员变量的含义如下：

- header - MemoryContextData 结构。必须把它设置为第一个成员，因为它是父类。所以一个指向 AllocSetContext 的指针也是指向 MemoryContextData 的指针。这也是 C 语言设计的一个常用技巧。
- blocks : 指向本内存池的活跃块。
- freelist 是一个数组，共计 11 个元素，它们指向空闲内存片的单向链表，具体含义后文会讨论。
- initBlockSize/maxBlockSize/nextBlockSize : 三个控制内存块体积的变量，前文已经论述过了。
- allocChunkLimit - 内存片的尺寸阈值。申请的内存的大小若等于或低于此阈值，则从 freelist 中寻找空闲内存片。超过此阈值则用 malloc() 单独分配一个内存块做为一个内存片，后面会讨论具体内容。
- keeper - 指向初始块。在内存池重置操作时通过它来判断哪一个内存块是初始块。
- freeListIndex - 用于内存池销毁时的优化，减少对 malloc() 和 free() 的调用次数。后文会讨论它的具体内容。

10.1.2.3 内存块 AllocBlock

结构体 AllocBlockData 表示一个内存块，AllocBlock 是相应的指针。AllocBlockData 的数据结构定义如下：

```
/* in src/backend/utils/mmgr/aset.c */
typedef struct AllocBlockData
{
    AllocSet aset; /* aset that owns this block */
    AllocBlock prev; /* prev block in aset's blocks list, if any */
    AllocBlock next; /* next block in aset's blocks list, if any */
    char *freeptr; /* start of free space in this block */
    char *endptr; /* end of space in this block */
} AllocBlockData;

typedef struct AllocBlockData *AllocBlock;
```

一个内存块的结构分为两部分：头部的 AllocBlockData 结构和其后的数据区。初始块和其它内存块的结构稍微不同，初始块的结构分为三部分：AllocSetContext，AllocBlockData 和数据区。初始块和非初始块的区别可以用图 6.5 来表示：

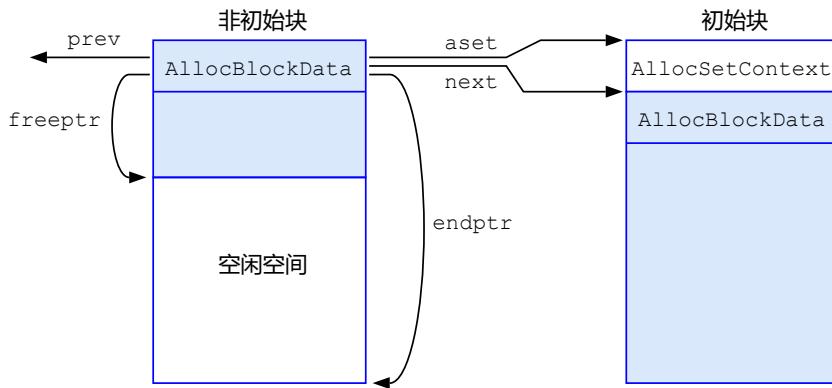


图 10.5: 初始块和非初始块的区别

结构体 AllocBlockData 各成员变量的含义比较简单，参考图 6.5 和源代码中的注释就不难理解。指针 aset 始终指向初始块头部的 AllocSetContext，指针 prev 和 next 用于形成内存块的双向链表，指针 freeptr 和 endptr 表示本块包含的空闲内存的首尾指针，本内存块空闲内存的大小为 endptr - freeptr，单位是字节。

10.1.2.4 内存片 MemoryChunk

结构体 MemoryChunk 表示一个内存片，每次调用 palloc() 从内存池中分配的内存都是一个内存片。MemoryChunk 的数据结构非常简单，定义如下：

```
/* in src/include/utils/memutils_memorychunk.h */
typedef struct MemoryChunk
{
    uint64_t hdrmask; /* must be last */
} MemoryChunk;
```

一个内存片分为两个部分：头部和数据区。头部是固定的 8 字节，保存在变量 hdrmask 中，图 6.6 显示了内存片的基本结构。

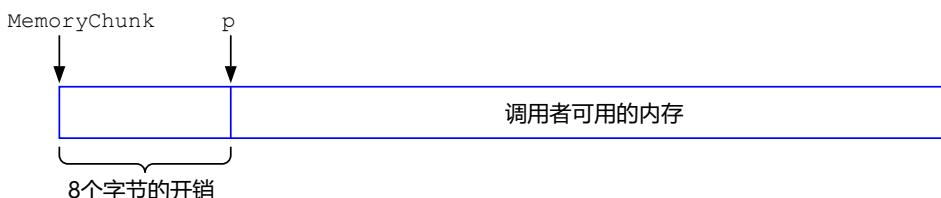


图 10.6: 内存片的基本结构

从图上我们不难理解内存片的基本结构，就是真正属于申请者可用的内存加上在开始的 8 个字节的开销，叫做 hdrmask。这 8 个字节用于管理该内存片的申请和释放的操作。函数 palloc() 申请的内存片最终返回给申请者的指针是图中的 p 指针，而不是内存片的开头。在 PG 16 之前，内存片的结构被称为 AllocChunkData，它包含了 16 个字节的开销 (overhead)，从 PG 16 开始，这个开销进一步优化成了 8 字节，这是 PostgreSQL 内存管理的一

一个重大改进。指针 p 指向的内存区非常容易理解，和我们使用 malloc() 申请的内存区并没有什么不同，那么如何理解这 8 个字节的开销呢？这个需要学习空闲内存片数组 freelist 和内存分配策略后才能理解，我们暂时放下。

10.1.2.5 空闲内存片数组 freelist

内存池是一次批发，多次零售。前文讲过：free() 释放的内存必须是 malloc() 申请的内存，而 pfree() 申请的内存只是从 malloc() 申请的大内存块中切下来的一部分，所以，通过 palloc() 从某个内存池中申请的内存片是无法通过 free() 立刻还给操作系统的。请记住：只有内存块才能够 malloc() 和 free()。难道这些不用的内存片就白白浪费掉吗？当然不会。在 AllocSetContext 里有个重要的成员变量 freelist 数组，其成员是指向内存片的指针，它管理着通过 pfree() 释放回本内存池的内存片。这些空闲内存能再次利用，可被未来的内存申请者所使用。变量 allocChunkLimit 记录着 freelist 数组中最大的内存片的尺寸，如果申请的内存尺寸小于等于 allocChunkLimit，则内存池管理器首先会在 freelist 中寻找。如果找不到，才会在活跃块中寻找空闲空间进行分配。数组 freelist 共计 11 个成员，图 6.7 展示了 freelist 的布局情况。

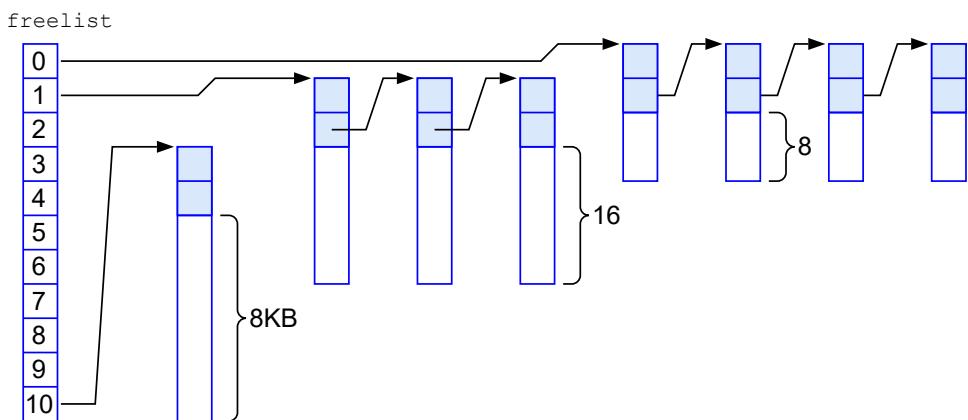


图 10.7：空闲内存片的管理数组

我们看到，数组 freelist 上的内存片的尺寸并不是随意的，而是统一为 2 的幂次方，即 $2^n(n = 3, 4, 5, 6, \dots)$ 。freelist[0] 指向的内存片的大小统一为 8 字节，此尺寸是申请者可以实际使用的内存的尺寸，不包括内存片头部的 8 个字节，所以 freelist[0] 指向的内存片实际上是 16 字节。freelist[1] 指向的内存片的大小统一为 16 字节，实际尺寸是 24 字节 (= 8 + 16)，依次类推，最后一个成员 freelist[10] 指向的内存片的大小统一为 8192 字节，实际尺寸是 8200 字节 (= 8 + 8192)，所以 allocChunkLimit = 8192，其值由如下代码设置：

```
/* in src/backend/utils/mmgr/aset.c */
#define ALLOC_MINBITS 3 /* smallest chunk size is 8 bytes */
#define ALLOCSET_NUM_FREELISTS 11
#define ALLOC_CHUNK_LIMIT (1 << (ALLOCSET_NUM_FREELISTS-1+ALLOC_MINBITS))
/* in src/backend/utils/mmgr/aset.c:AllocSetContextCreateInternal() */
    set->allocChunkLimit = ALLOC_CHUNK_LIMIT;
```

假设用户要申请 11 个字节的内存，很显然它要在 freelist[1] 中找，因为这里的空闲内存片的大小为 16 个字节。如果要申请 33 个字节，就在 freelist[3] 中寻找，因为它里面的内存片都是 64 个字节。表面上这种做齐内存片大小的方法可能存在一些浪费，因为申请的内存大小都按照 8 字节对齐，所以可以把浪费的空间控制在一个稳定的范围。如果外部申请者申请的内存尺寸 size 都是 2 的幂次方，就不存在任何浪费了。通过做齐内存的尺寸，提高了内存操作的性能，这是一个典型的空间换时间的设计。

基于 freelist 的这种设计，就需要一个操作：如何根据内存的尺寸 size 寻找 freelist 数组的下标 idx。例如给出了 size=33，得到的 idx=3。函数 AllocSetFreeIndex() 就是干这个事的，有兴趣的读者可以阅读一下这个函数的

代码，它采用了 CPU 提供的硬件指令加速计算，把性能压榨到了极致，同时还提供了不依赖 CPU 的通用的查表法来快速计算。

10.1.2.6 内存池操作的函数指针

无论内存池采用何种分配策略，无非就是内存池的创建、销毁、重置，内存片的申请和释放等等操作。PostgreSQL 把这些行为抽象出一个统一的模型，采用函数指针的形式进行统一管理。它的数据结构是 `MemoryContextMethods`，其定义如下：

```
/* in src/include/nodes/memnodes.h */
typedef struct MemoryContextMethods
{
    void        (*alloc) (MemoryContext context, Size size);
    void        (*free_p) (void *pointer);
    void        (*realloc) (void *pointer, Size size);
    void        (*reset) (MemoryContext context);
    void        (*delete_context) (MemoryContext context);
    MemoryContext (*get_chunk_context) (void *pointer);
    Size        (*get_chunk_space) (void *pointer);
    bool        (*is_empty) (MemoryContext context);
    void        (*stats) (MemoryContext context,
                         MemoryStatsPrintFunc printfunc, void *passthru,
                         MemoryContextCounters *totals,
                         bool print_to_stderr);
} MemoryContextMethods;
```

上述每一个函数指针均对应一个内存池的操作，譬如 `alloc` 就是从内存池获得内存片，`free_p` 是把内存片释放给内存池，`delete_context` 是删除内存池，`reset` 是重置内存池。有了这个数据结构，PostgreSQL 针对不同的内存分配策略，预留了 8 种可能的内存分配方法，其定义如下：

```
/* in src/include/utils/memutils_internals.h */
typedef enum MemoryContextMethodID
{
    MCTX_UNUSED1_ID,           /* 000 occurs in never-used memory */
    MCTX_UNUSED2_ID,           /* glibc malloc'd chunks usually match 001 */
    MCTX_UNUSED3_ID,           /* glibc malloc'd chunks > 128kB match 010 */
    MCTX_ASET_ID,
    MCTX_GENERATION_ID,
    MCTX_SLAB_ID,
    MCTX_ALIGNED_REDIRECT_ID,
    MCTX_UNUSED4_ID           /* 111 occurs in wipe_mem'd memory */
} MemoryContextMethodID;
/* in src/backend/utils/mmgr/mcxt.c */
static const MemoryContextMethods mcxt_methods[] = {
    /* aset.c */
    [MCTX_ASET_ID].alloc = AllocSetAlloc,
    [MCTX_ASET_ID].free_p = AllocSetFree,
    [MCTX_ASET_ID].realloc = AllocSetRealloc,
    [MCTX_ASET_ID].reset = AllocSetReset,
    [MCTX_ASET_ID].delete_context = AllocSetDelete,
```

```

[MCTX_ASET_ID].get_chunk_context = AllocSetGetChunkContext,
[MCTX_ASET_ID].get_chunk_space = AllocSetGetChunkSpace,
[MCTX_ASET_ID].is_empty = AllocSetIsEmpty,
[MCTX_ASET_ID].stats = AllocSetStats,
/* generation.c */
[MCTX_GENERATION_ID].alloc = GenerationAlloc,
.....
/* slab.c */
[MCTX_SLAB_ID].alloc = SlabAlloc,
...
};


```

函数指针数组 mcxt_methods 有 8 个元素，对应八种可能的内存分配策略，其中真正使用的，目前只有三种，分别是 AllocSet, Generation 和 Slab。我们重点研究缺省的实现方式 AllocSet。

掌握了这些知识，我们再回头看内存片的头 8 个字节的含义。很显然，我们要在这 8 个字节，共计 64-bit 的空间里，要存储三个信息：这个内存片是哪种分配策略分配的？这个内存片有多大？这个内存片属于哪个内存池的？在 PG 16 之前，这些信息的存储是比较粗放的，譬如该内存片的尺寸，就要占 8 个字节。PG 16 现在对这些信息的存储进行了进一步优化，统统放在这 8 个字节中了。

我们想一下：一共预留了八种可能的内存分配策略，只需要 3 个比特就可以了。有了 freelist 的空闲内存片数组，可以用该数组的下标来表示内存片的大小，譬如 1 表示 16 字节，10 表示 8192 个字节，这些信息也占不了多少比特。如何由内存片拿到所属内存池的信息呢？如果能够通过该内存片拿到该内存片所属的内存块指针，就可以进一步获得该内存片所属内存池的指针了，因为内存块结构中有一个 aset 指针指向了该内存块所属的内存池的控制头。要解决这个问题，我们可以先了解图 6.8：

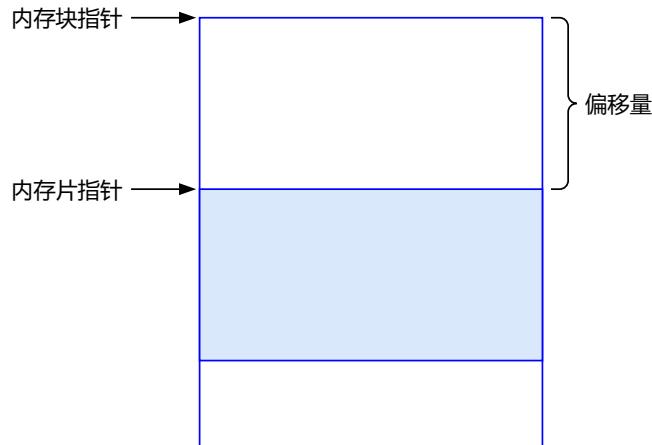


图 10.8: 内存片和内存块的关系

我们知道，内存块是通过 malloc() 申请的一块连续的内存区，而内存片又是在该内存块中切出来的一段连续的内存区。如果我们有了内存片指针和内存块指针之间的偏移量 offset，只要用内存片的指针减去这个偏移量，不就可以获得内存块的指针了吗？综上所述，内存片的头 8 个字节可以被设计成三部分，如图 6.9 所示：

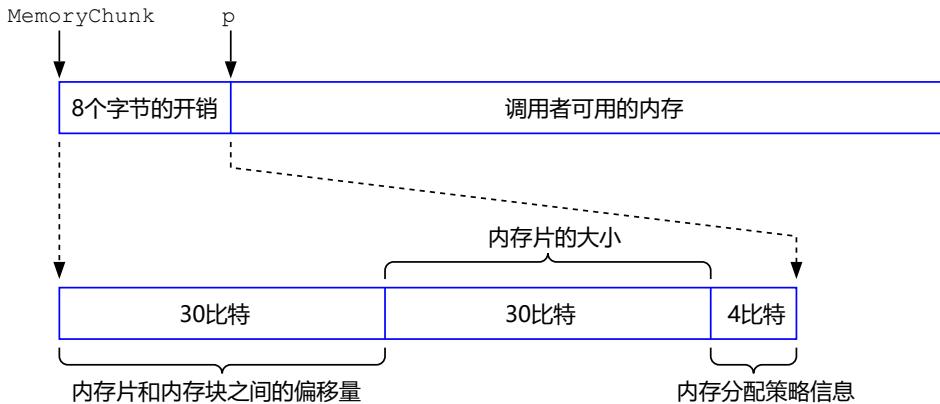


图 10.9: 内存片的头 8 个字节

在上图中，内存分配策略被安排成为最低的 3 个比特，第 4 个比特空闲，可能未来会使用。中间的 30 个比特被设计成该内存片的尺寸。如果该内存片的尺寸小于阈值 allocChunkLimit，则保存该内存片对应的空闲内存片数组 freelist 的下标。其实也可以直接保存尺寸，30 个比特足够保存 1GB 的大小了。最高位的 30 个比特用于记录该内存片和内存块开始地址的偏移量，所以一个内存块最大尺寸不能超过 1GB，否则这 30 个比特就不够用了。一次性分配 1GB 的内存的情况几乎没有，所以这个限制问题不大。有了这个设计，下面我们看看一段关键代码：

```
/* in src/include/utils/memutils_memorychunk.h */
static inline void
MemoryChunkSetHdrMask(MemoryChunk *chunk, void *block, Size value, MemoryContextMethodID methodid)
{
    Size blockoffset = (char *) chunk - (char *) block;
    Assert((char *) chunk >= (char *) block);
    Assert(blockoffset <= MEMORYCHUNK_MAX_BLOCKOFFSET);
    Assert(value <= MEMORYCHUNK_MAX_VALUE);
    Assert((int) methodid <= MEMORY_CONTEXT_METHODID_MASK);
    chunk->hdrmask = (((uint64) blockoffset) << MEMORYCHUNK_BLOCKOFFSET_BASEBIT) |
        (((uint64) value) << MEMORYCHUNK_VALUE_BASEBIT) | methodid;
}
```

上述代码就是把这三部分信息保存在这 8 字节当中，其逻辑一目了然，不用过多论述。其中很多的常量定义，请自行在源代码中寻找。

10.1.3 内存池的操作函数

在熟悉内存池的相关数据结构之后，阅读内存池相关操作的具体源代码就不难了。下面我就带领读者走读一下其中的关键代码。如果你在分析源代码的过程中陷入迷茫，请再次阅读前一节的数据结构的关键点。

10.1.3.1 内存池创建函数之分析

内存池的创建是由 AllocSetContextCreateInternal() 完成的。由于内存池的创建和销毁可能会非常频繁。例如，每次处理 SQL 查询时，都要为该查询创建内存池，查询结束后再释放内存池。这种频繁的内存池创建和销毁的操作势必会频繁地调用 malloc() 和 free()。为了进一步减少系统调用，提高性能，PostgreSQL 建立了一个小小的 context_freetlist 数组来跟踪记录即将被销毁的内存池。注意，这个 freelist 数组是关于内存池的，不是管理内存片的，请读者不要混淆。某个内存池在被销毁之前，会首先释放除了初始块以外的所有内存块，然后把初始块挂在

context_freelist 指向的单向链表中。下次创建内存池时，就可以直接在这个单向链表中寻找某个只有初始块的空闲内存池，加快了内存池的创建速度。数组 context_freelist 只有两个成员，维系了两个单向链表，一个是初始块为 8KB 的普通内存池，另外一个是初始块为 1KB 的小内存池，图 6.10 展示了空闲内存池链表的基本形态。

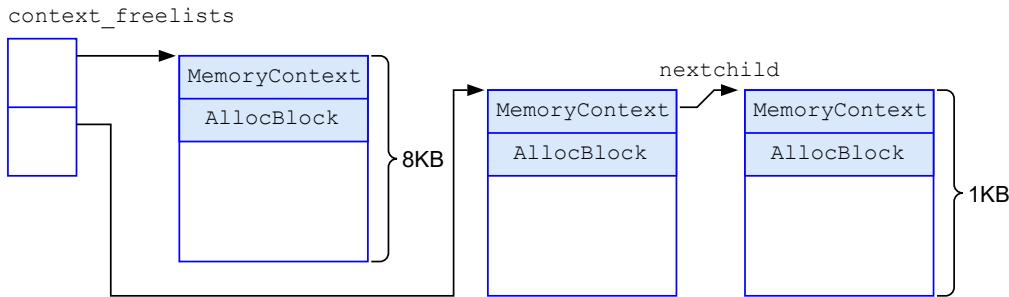


图 10.10: 空闲内存池的单向链表

数组 context_freelist 的数据结构定义如下：

```

/* in src/backend/utils/mmgr/aset.c */
#define MAX_FREE_CONTEXTS 100 /* arbitrary limit on freelist length */
typedef struct AllocSetFreeList
{
    int      num_free;    /* current list length */
    AllocSetContext *first_free; /* list header */
} AllocSetFreeList;

/* context_freelists[0] is for default params, [1] for small params */
static AllocSetFreeList context_freelists[2] = { { 0, NULL }, { 0, NULL } };
/* 初始块的大小分为1KB的小块和8KB的普通块两种情况 */
#define ALLOCSET_DEFAULT_MINSIZE 0
#define ALLOCSET_DEFAULT_INITSIZE (8 * 1024)
#define ALLOCSET_SMALL_MINSIZE 0
#define ALLOCSET_SMALL_INITSIZE (1 * 1024)

```

变量 num_free 记录了本单向链表中的空闲内存池的个数，first_free 则指向了空闲内存池单向链表，单向链表之间用结构体 MemoryContext 的 nextchild 指针来进行串连。

在函数 AllocSetContextCreateInternal() 的入口参数中，除了初始块的大小 initBlockSize 以外，还可以指定一个 minContextSize 变量。此两者的关系是：如果 minContextSize 的值不是 0，则初始块的大小就以它为准，否则就以 initBlockSize 为准。这增加了对内存池中 Block 尺寸控制的精细化。下面我们走读一下 AllocSetContextCreateInternal() 的源代码。

```

/* in src/include/utils/memutils.h */
#define AllocSetContextCreate AllocSetContextCreateInternal
/* in src/backend/utils/mmgr/aset.c */
MemoryContext AllocSetContextCreateInternal(MemoryContext parent, const char *name,
                                             Size minContextSize, Size initBlockSize, Size maxBlockSize)
{
    int      freeListIndex;
    Size     firstBlockSize;
    AllocSet  set;
    AllocBlock block;

```

```

.....
if (minContextSize == ALLOCSET_DEFAULT_MINSIZE && initBlockSize == ALLOCSET_DEFAULT_INITSIZE)
    freeListIndex = 0; /* 8KB的普通池，被销毁时就挂在context_freeLists[0]的链表中 */
else if (minContextSize == ALLOCSET_SMALL_MINSIZE && initBlockSize == ALLOCSET_SMALL_INITSIZE)
    freeListIndex = 1; /* 1KB的小池，被销毁时就挂在context_freeLists[1]的链表中 */
else freeListIndex = -1; /* 其它尺寸的内存池销毁时就彻底销毁，不再挽留 */

if (freeListIndex >= 0) { /* 若要创建的内存池初始大小是1KB或8KB，就在context_freeLists中寻找空闲池 */
    AllocSetFreeList *freelist = &context_freeLists[freeListIndex];
    if (freelist->first_free != NULL) { /* 若该freelist上中有空闲内存池，则用之 */
        set = freelist->first_free;
        freelist->first_free = (AllocSet) set->header.nextchild;
        freelist->num_free--; /* 摘下一个内存池，数目减一 */
        /* Update its maxBlockSize; everything else should be OK */
        set->maxBlockSize = maxBlockSize;
        /* Reinitialize its header, installing correct name and parent */
        MemoryContextCreate((MemoryContext) set, T_AllocSetContext,
            MCTX_ASET_ID, parent, name);
        ((MemoryContext) set)->mem_allocated = set->keeper->endptr - ((char *) set);
        return (MemoryContext) set; /* 返回给调用者 */
    }
}
/* 下面的代码逻辑处理两种可能性：一是没有找到空闲池，二是该池的初始块的大小不是8KB或1KB */
/* 初始块中必须包含AllocSetContext, AllocBlockData和AllocChunkData三个结构 */
firstBlockSize = MAXALIGN(sizeof(AllocSetContext)) +
    ALLOC_BLOCKHDRSZ + ALLOC_CHUNKHDRSZ;
if (minContextSize != 0)
    firstBlockSize = Max(firstBlockSize, minContextSize);
else
    firstBlockSize = Max(firstBlockSize, initBlockSize);
/* 至此，该内存池的初始块的大小已经确定，记录在firstBlockSize中 */
set = (AllocSet) malloc(firstBlockSize); /* 创建该内存池的初始块 */
if (set == NULL) { /* 若创建失败，整个程序就退出了 */
    /* 初始块的开始部分是AllocSetContext, AllocSetContext后面是AllocBlockData */
    block = (AllocBlock) (((char *) set) + MAXALIGN(sizeof(AllocSetContext)));
    block->aset = set; /* 指向内存池的控制头，即初始块的头部 */
    block->freeptr = ((char *) block) + ALLOC_BLOCKHDRSZ; /* 本块目前没有内存片 */
    block->endptr = ((char *) set) + firstBlockSize; /* 本块的空闲内存结束的地址 */
    block->prev = NULL; /* 现在只有初始块，故双向链表的前导和后导指针都为NULL */
    block->next = NULL;
    set->blocks = block; /* blocks指向内存块双向链表的头部，即活跃块，此时也是初始块 */
    set->keeper = block; /* keeper永远指向初始块，在内存池重置时才能保证其不被释放 */
    MemSetAligned(set->freelist, 0, sizeof(set->freelist)); /* 清空空闲内存片数组freelist */
    /* 做一些登记工作 */
    set->initBlockSize = initBlockSize; set->maxBlockSize = maxBlockSize;
    set->nextBlockSize = initBlockSize; set->freeListIndex = freeListIndex;
    set->allocChunkLimit = ALLOC_CHUNK_LIMIT; /* 设置空闲内存片大小的阈值，并做一些必要的调整 */
    while ((Size) (set->allocChunkLimit + ALLOC_CHUNKHDRSZ) >

```

```

        (Size) ((maxBlockSize - ALLOC_BLOCKHDRSZ) / ALLOC_CHUNK_FRACTION))
    set->allocChunkLimit >= 1;
/* 初始化MemoryContext中的内容 */
MemoryContextCreate((MemoryContext) set, T_AllocSetContext, MCTX_ASET_ID, parent, name);
((MemoryContext) set)->mem_allocated = firstBlockSize;
return (MemoryContext) set;
}

```

当上述函数执行完毕后，整个内存池只有一个初始块，其内存布局如图 6.11 所示：

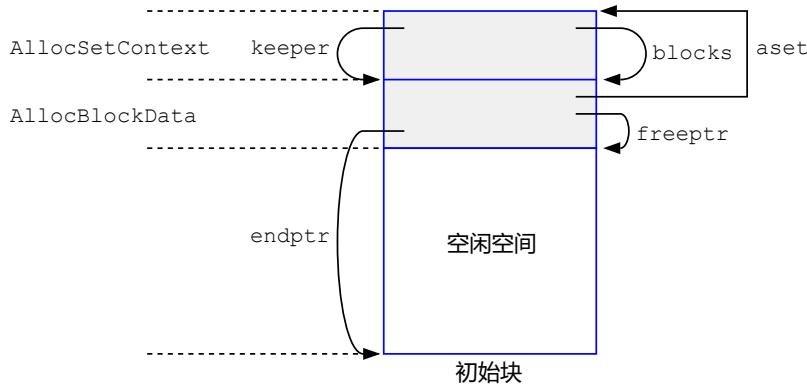


图 10.11：内存池刚建立时的状态

10.1.3.2 内存池重置函数之分析

函数 AllocSetReset() 的作用是重置本内存池，即只保留初始块，其余的内存块全部通过 free() 释放给操作系统。结构体 AllocSet 的成员变量 keeper 指针指向初始块，在该内存池的整个生命周期中都不会改变，所以只要判断一个 block 指针是否和 keeper 指针相等就能知道这个内存块是否是初始块。这个函数的逻辑比较简单，下面我们走读一下 AllocSetReset() 的源代码。

```

/* in src/backend/utils/mmgr/aset.c */
void AllocSetReset(MemoryContext context)
{
    AllocSet  set = (AllocSet) context;
    AllocBlock  block;
    Size      keepersize PG_USED_FOR_ASSERTS_ONLY;
    /* 获得该内存池初始块的大小 */
    keepersize = set->keeper->endptr - ((char *) set);
    /* 快速地把freelist数组清空，MemSetAligned的定义在src/include/c.h中 */
    MemSetAligned(set->freelist, 0, sizeof(set->freelist));
    block = set->blocks; /* block指向了内存块双向链表的头部 */
    set->blocks = set->keeper; /* blocks指针指向初始块，这是内存池刚创建后的状态 */
    while (block != NULL) { /* 开始遍历block的双向链表 */
        AllocBlock  next = block->next;
        if (block == set->keeper) { /* 该内存块是初始块，需重新设置其内容，但并不释放之 */
            /* block指向AllocSetContext的后面，故加一个块头，就是该块的数据区 */
            char      *datastart = ((char *) block) + ALLOC_BLOCKHDRSZ;
            wipe_mem(datastart, block->freeprt - datastart); /* 把本块残存的内容消除掉 */
            block->freeprt = datastart; /* 把空闲内存的起始指针指向数据区的开始部分 */
        }
    }
}

```

```

block->prev = NULL; block->next = NULL; /* 把初始块从双链中切下来，该链表的其它块都会被释放掉 */
}
else { /* 这个block是非初始块 */
    context->mem_allocated -= block->endptr - ((char *) block); /* 调整总体大小 */
    wipe_mem(block, block->freeprt - ((char *) block)); /* 把残存的内容消除掉 */
    free(block); /* 把这个内存块释放给操作系统 */
}
block = next; /* 继续处理双向链表中的下一个内存块 */
}

/* 当除了初始块以外的所有内存块都被释放后，本内存池的体积就是初始块的大小 */
Assert(context->mem_allocated == keepersize);
set->nextBlockSize = set->initBlockSize; /* 现在本内存池中只有一个初始块，调整一下nextBlockSize */
}

```

10.1.3.3 内存池销毁函数之分析

函数 AllocSetDelete() 的工作是销毁内存池，它的入口参数是一个指向该内存池 MemoryContext 指针 context。在本函数执行完毕后，context 指针指向的内存池片甲不留，context 指针也失效了，不能在继续引用里面的内容了。下面我们走读一下其源代码。

```

/* in src/backend/utils/mmgr/aset.c */
void AllocSetDelete(MemoryContext context)
{
    AllocSet set = (AllocSet) context;
    AllocBlock block = set->blocks; /* block指向内存块双向链表的头部，即活跃块 */
    Size keepersize PG_USED_FOR_ASSERTS_ONLY;
    /* 计算初始块的尺寸 */
    keepersize = set->keeper->endptr - ((char *) set);
    if (set->freeListIndex >= 0) { /* 可以把该池重置后放在context_freelist数组中，并不真正的销毁 */
        AllocSetFreeList *freelist = &context_freelists[set->freeListIndex];
        if (!context->isReset) /* 先调用AllocSetReset()重置该内存池*/
            MemoryContextResetOnly(context);
        if (freelist->num_free >= MAX_FREE_CONTEXTS) { /* 若空闲内存池积累的太多了，则把所有的空闲内存池释放 */
            while (freelist->first_free != NULL) { /* 依次遍历该单向链表 */
                AllocSetContext *oldset = freelist->first_free; /* 摘下来第一个成员 */
                freelist->first_free = (AllocSetContext *) oldset->header.nextchild;
                freelist->num_free--;
                free(oldset);
            }
        }
        /* 把本次即将释放的内存池挂在这个context_freelist上 */
        set->header.nextchild = (MemoryContext) freelist->first_free;
        freelist->first_free = set;
        freelist->num_free++;
        return;
    }
    /* 该池的尺寸不是8KB或1KB，就彻底销毁之。遍历双链，依次释放其上的内存块 */
    while (block != NULL) {

```

```

AllocBlock next = block->next;
if (block != set->keeper) /* 若不是初始块，就释放该内存块 */
    context->mem_allocated -= block->endptr - ((char *) block); /* 调整总体积 */
wipe_mem(block, block->freeptr - ((char *) block));
if (block != set->keeper) free(block); /* 释放非初始块 */
block = next; /* 继续遍历内存块的双向数组 */
}
/* 最后剩下的内存块只有初始块，所以内存池的总大小等于初始Block的大小 */
Assert(context->mem_allocated == keepersize);
/* Finally, free the context header, including the keeper block */
free(set); /* 最后释放初始块，自然set指针也就失效了 */
}

```

10.1.3.4 内存片分配函数之分析

函数 AllocSetAlloc() 承担了从内存池中分配内存片的工作，它当然是使用最频繁的内存池函数之一了。如果内存片申请成功，则把分配的内存片指针返回给调用者，否则返回 NULL。每当接受新的内存申请后，该函数采用图 6.12 所示的分配逻辑来寻找合适的内存片：

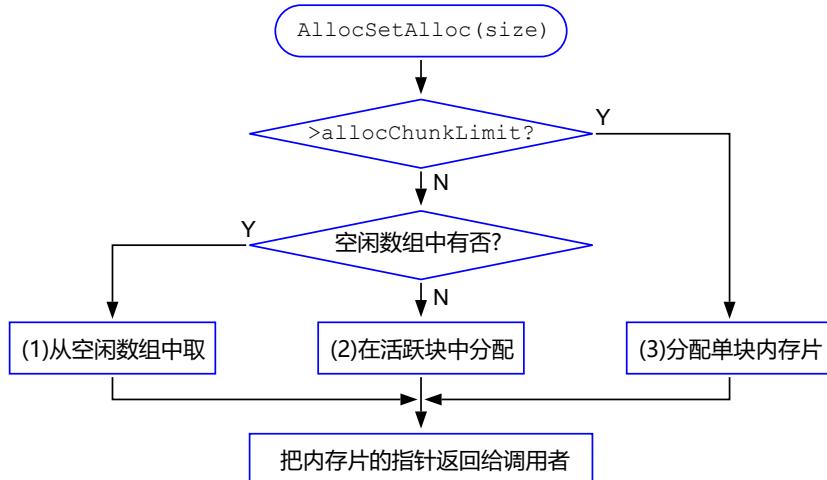


图 10.12：内存片的分配策略

根据上面的流程图，分配策略分为三种情况：1. 若申请的内存尺寸小于等于阈值 allocChunkLimit，就在空闲内存片数组 freelist 中寻找空间内存片。如果找到，就直接返回给调用者。2. 如果在空闲内存片数组中没有找到合适的空闲内存片，就在活跃块中寻找空闲空间，找到后就从空闲空间中切下来一块内存片返回给调用者。如果在活跃块中找不到足够的空闲空间，则申请一个新的内存块，其体积是当前活跃块的两倍，直至达到内存块的上限 maxBlockSize 后就不再增加了。这个新块就是新的活跃块，老活跃块上可能会残存一些空闲空间，顺手整理一下挂在空闲内存片数组上，避免浪费。3. 若申请的内存尺寸大于阈值 allocChunkLimit，就简单通过 malloc() 分配一个内存块，整个块作为一个内存片返回给调用者。这种只有一个内存片的内存块被称为“单片内存块”(single-chunk block)，其上的内存片被称为“单块内存片”(single-block chunk)。这个内存块创建完毕后被挂在活跃块的后面。

由此可知，当一个内存片的尺寸大于 allocChunkLimit 时，就可以判断它为单块内存片。单块内存片是直接使用 malloc() 分配的，对它的释放自然也是用 free()。单片内存块可以用图 6.13 来表示。



图 10.13: 单片内存块的结构

我们知道内存片的开始部分是 8 字节，分为三部分，这种设计是针对非单块内存片的，即其尺寸小于等于阈值 allocChunkLimit。对于单块内存片，这 8 个字节又如何设计呢？我们可以理解这段代码：

```
/* in src/include/utils/memutils_memorychunk.h */
#define MEMORYCHUNK_MAGIC (UINT64CONST(0xB1A8DB858EB6EFBA) \
    >> MEMORYCHUNK_VALUE_BASEBIT << MEMORYCHUNK_VALUE_BASEBIT)

static inline void MemoryChunkSetHdrMaskExternal(MemoryChunk *chunk, MemoryContextMethodID methodid)
{
    chunk->hdrmask = MEMORYCHUNK_MAGIC | (((uint64) 1) << MEMORYCHUNK_EXTERNAL_BASEBIT) | methodid;
}
```

由上面的代码可知，对于单块内存片的设计比较简单粗暴，往这 8 个字节里面塞一个写死的魔幻数即可，这个魔幻数由 MEMORYCHUNK_MAGIC 定义。所以我们拿到一个内存片后，往前移动 8 个字节就能够拿到这个 8 字节的开销，根据里面的内容判断是否包含魔幻数，就能知道它是普通的小内存片，还是大的单块内存片。下面两个宏定义就展示了由 palloc() 返回的供使用者实际使用的内存指针和内存片指针的关系，无非就是前后移动 8 个字节而已。

```
/* in src/include/utils/memutils_memorychunk.h */
/* Get the MemoryChunk from the pointer */
#define PointerGetMemoryChunk(p) ((MemoryChunk *) ((char *) (p) - sizeof(MemoryChunk)))
/* Get the pointer from the MemoryChunk */
#define MemoryChunkGetPointer(c) ((void *) ((char *) (c) + sizeof(MemoryChunk)))
```

设置阈值 allocChunkLimit 的主要原因是，若所有内存片都放在空闲内存片数组中进行管理，就存在一种可能性：某个申请者一下子申请了 100M 的内存，使用后通过 pfree() 释放它，于是这个 100M 的大块内存被挂在空闲内存片数组上，后面的内存申请者再也没有申请过这么大的内存了，就导致这块大内存迟迟不能被释放给操作系统，白白占用了宝贵的内存资源。所以空闲内存片数组的上限控制在 8KB。对于超过 8KB 的内存申请，内存池会直接创建一个整块给申请者，避免了可能的内存浪费。函数 AllocSetAlloc() 的源代码比较长，所以我们按流程图列出的三种情况，分段分析该函数的源代码。

情景 1 - 申请的内存片尺寸小于等于阈值

这种情景的逻辑代码比较简单，就是在空闲内存片数组中找有没有合适的内存片，有的话就直接返回给申请者。下面我们走读一下此种情景下的代码逻辑。

```
/* in src/backend/utils/mmgr/aset.c */
void * AllocSetAlloc(MemoryContext context, Size size)
{
    AllocSet set = (AllocSet) context;
    AllocBlock block;
    MemoryChunk *chunk;
```

```

int      fidx;
Size     chunk_size;
Size     blksize;
if (size > set->allocChunkLimit) { /* 参考情景3的分析 */ }
/* 现在申请的内存片的尺寸不超过阈值，则在freelist上查找是否有空闲内存片 */
fidx = AllocSetFreeIndex(size); /* 根据申请内存的尺寸计算freelist的下标 */
if (chunk != NULL) { /* freelist[fidx]上有空闲的内存片 */
    AllocFreeListLink *link = GetFreeListLink(chunk);
    set->freelist[fidx] = link->next; /* 把首个空闲内存片摘下来 */
    return MemoryChunkGetPointer(chunk); /* 返回该内存片的真正数据指针给调用者 */
}

```

情景 2 - 在活跃内存块上寻找空闲空间

如果在空闲内存片数组中找不到空闲内存片，就要在活跃块上寻找空闲空间。这里又分为两种情况，其一是活跃块有足够的空闲空间，这比较简单，直接切下一块走人。其二是活跃块没有足够的空闲空间，就需要申请一个更大的块作为新的活跃块，老活跃块上残存的空闲空间可以被挂在空闲内存片数组中。下面我们走读一下此种情景下的代码逻辑。

```

/* in src/backend/utils/mmgr/aset.c:AllocSetAlloc() */
/* 计算要分配的内存片的大小，肯定是8,16,32,64 ... 8192这样的值 */
chunk_size = GetChunkSizeFromFreeListIdx(fidx);
if ((block = set->blocks) != NULL) { /* block肯定不为NULL，因为至少它指向了初始块 */
    /* 现在block指向活跃块，检查活跃块的空闲空间是否够分配 */
    Size availspace = block->endptr - block->freeprt; /* 计算活跃块的空闲空间的大小*/
    if (availspace < (chunk_size + ALLOC_CHUNKHRSZ)) { /* 活跃块的空闲内存不够 */
        /* 把当前活跃块上的空闲空间挂在空闲内存片数组中，这部分代码作为练习题，请自行分析之
         其基本思想就是把老块上的零碎的空闲内存片整理一下，挂在空闲内存片数组freelist上
        */
        while (availspace >= ((1 << ALLOC_MINBITS) + ALLOC_CHUNKHRSZ)) { ..... }
        block = NULL; /* 把block设为NULL，表明需要通过malloc()申请一个新块 */
    }
}
if (block == NULL) { /* 申请一个更大的新块，此代码的分析作为练习题，请自行分析 */
    /* 走到这里，当前活跃块中的空闲空间足够分配了，下面进行真正的分配 */
    chunk = (MemoryChunk *) (block->freeprt); /* 拿到活跃块的空闲空间的起始地址 */
    /* chunk_size是真正的数据的大小，前面还要加8个字节 */
    block->freeprt += (chunk_size + ALLOC_CHUNKHRSZ); /* 调整活跃块的空闲内存起始地址 */
    /* 填充该内存片开始的8个字节。因为该内存片是小内存片，所以需要填充3部分信息，前文已经讨论过了 */
    MemoryChunkSetHdrMask(chunk, block, fidx, MCTX_ASET_ID);
    return MemoryChunkGetPointer(chunk); /* 返回该chunk真正的数据区指针 */
}

```

情景 3 - 申请的内存片尺寸大于阈值

此种情况的逻辑比较简单粗暴，直接使用 malloc() 分配一个新的内存块，把整个块都作为一个内存片返回给申请者。下面我们走读一下它的代码。

```

/* in src/backend/utils/mmgr/aset.c:AllocSetAlloc() */
if (size > set->allocChunkLimit) {
    chunk_size = MAXALIGN(size); /* 先把尺寸按照8字节做齐 */

```

```

blksize = chunk_size + ALLOC_BLOCKHRSZ + ALLOC_CHUNKHRSZ;
block = (AllocBlock) malloc(blksize); /* 直接调用malloc()进行内存分配该块 */
if (block == NULL) return NULL; /* 分配失败就返回NULL */
context->mem_allocated += blksize; /* 调整本内存池中已分配内存的总体尺寸 */
block->aset = set; /* 把aset指针指向其所属内存池的控制头 */
/* 因为本块整个算一个内存片，所以没有空闲空间了，调整一下首位指针指向尾部 */
block->freeptr = block->endptr = ((char *) block) + blksize;
chunk = (MemoryChunk *) (((char *) block) + ALLOC_BLOCKHRSZ);
/* 往该内存片的头8个字节里面塞入魔幻数，由此可以判断该内存片是单块内存片 */
MemoryChunkSetHdrMaskExternal(chunk, MCTX_ASET_ID);
if (set->blocks != NULL) { /* 把这个新块插入到活跃块的后面 */
    block->prev = set->blocks;
    block->next = set->blocks->next;
    if (block->next)
        block->next->prev = block;
    set->blocks->next = block;
}
else { block->prev = NULL; block->next = NULL; set->blocks = block; } /* 这种情况不可能出现 */
return MemoryChunkGetPointer(chunk);
}

```

上述代码走完后，其内存布局如图 6.14 所示：

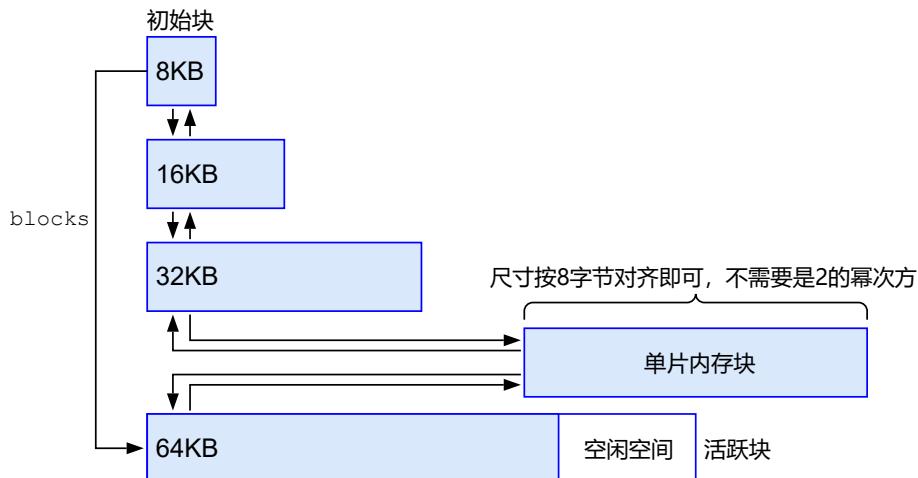


图 10.14: 申请单片内存块后的情景

10.1.3.5 内存片释放函数之分析

函数 AllocSetFree() 用于把内存片返还给内存池。理解了内存片如何分配的过程，内存片返还的过程就非常容易理解。下面我们走读一下代码。

```

/* in src/backend/utils/mmgr/aset.c */
void AllocSetFree(void *pointer)
{
    AllocSet set;
    MemoryChunk *chunk = PointerGetMemoryChunk(pointer);
    if (MemoryChunkIsExternal(chunk)) { /* 该内存片必是单块内存片，直接free()释放之 */

```

```

AllocBlock block = ExternalChunkGetBlock(chunk);
set = block->aset;
/* 检查通过后，就把该块从内存块双向链表中摘下来，准备释放之 */
if (block->prev) block->prev->next = block->next;
else set->blocks = block->next;
if (block->next) block->next->prev = block->prev;
/* 调整一下内存池的总体积 */
set->header.mem_allocated -= block->endptr - ((char *) block);
free(block); /* 把该块释放给操作系统 */
}

else { /* 如果不是单块内存片，就是普通内存片，把它挂在空闲内存片数组中 */
    AllocBlock block = MemoryChunkGetBlock(chunk);
    int      fidx;
    AllocFreeListLink *link;
    set = block->aset;
    fidx = MemoryChunkGetValue(chunk);
    link = GetFreeListLink(chunk);
    link->next = set->freelist[fidx];
    set->freelist[fidx] = chunk;
}
}
}

```

10.1.3.6 分配内存片接口函数之分析

内存池管理器提供了外部使用的接口函数，最主要的内存分配函数是 `palloc()` 和 `pfree()`。在理解了底层函数的逻辑后，它们的代码逻辑就一目了然了，下面我们走读一下。

```

/* in src/backend/utils/mmgr/mcxt.c */
void * palloc(Size size)
{
    void      *ret;
    MemoryContext context = CurrentMemoryContext;
    context->isReset = false; /* 因为要从该内存池中分配内存了，isReset就设为false */
    ret = context->methods->alloc(context, size); /* 实际调用AllocSetAlloc()分配内存 */
    if (unlikely(ret == NULL)) { /* 申请失败后的错误处理 */
        return ret;
    }

#define MCXT_METHOD(pointer, method)      mcxt_methods[GetMemoryChunkMethodID(pointer)].method
void pfree(void *pointer)
{
    MCXT_METHOD(pointer, free_p) (pointer);
}

```

至此，我们已经把 PostgreSQL 整个内存池的思想和相关细节都考察了一边。内存池还有其它的函数，如重新分配内存片的 `realloc` 等，请读者作为练习题目自行分析之。

10.2 动态哈希算法

哈希 (Hash) 算法是计算机科学领域的一项伟大发明，因其极快的查找速度，被广泛地运用到计算机工业的各个领域，在 PostgreSQL 各个组件中也有大量的应用。例如，PostgreSQL 想把某个数据文件中的一个数据块读入到内存，譬如四元组 (a,b,c,d) 表示要读取 a 号表空间的 b 号数据库的 c 号表的第 d 号数据块。在读取之前，PostgreSQL 需要知道该数据块是否已经被读入到共享池中的某个数据页中了。假设共享池的大小为 32GB，就有 $32\text{GB} / 8\text{KB} = 4194304$ 个数据页。如何在这 400 多万个数据页中快速查找，就是一个必须要解决的问题，类似这样的需求就是哈希算法大显身手的地方。所以，理解哈希算法的基本概念和具体实现细节，对于深入理解 PostgreSQL 的各种技术，具有基础性的作用。PostgreSQL 使用的哈希技术分为基于内存的哈希表和基于磁盘的哈希索引两种，本节只介绍内存哈希表及其算法的具体实现，哈希索引则会在后面的章节中进行研讨。

10.2.1 基本原理

10.2.1.1 通用哈希算法的基本原理

哈希是最快的查找算法之一，如果设计得当，它的查找速度是常值，即时间复杂度为 $O(1)$ 。假设我们有一个“键” (Key, 简称 K) 和一个“值” (Value, 简称 V)，该键和值组成一对，被称为“键值对”，简称 KV 对。你可以把键和值理解为长度已知的一个字节串，例如 ('PostgreSQL', 'RDBMS')，其中 'PostgreSQL' 是键，'RDBMS' 是值。当该 KV 对存放在一个被叫做哈希表的数据结构中后，哈希算法像个魔法盒子，会根据输入的 K，以极快的速度在该哈希表中查找到对应的 V。图 6.15 展示了哈希算法的基本原理。

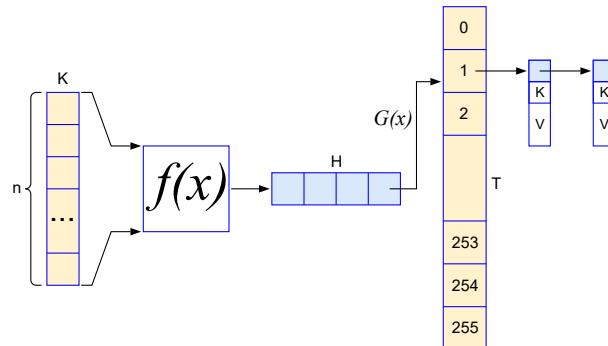


图 10.15: 哈希算法的基本原理

最左边是一个长度为 n 字节的字节串 K ，这就是键，该键经过一个哈希函数 $f(x)$ 的运算，得到了一个固定长度的哈希值 (Hash Value)，即图中所示的 H ， $H = f(K)$ 。最右边有一个长度为 N 的数组 T ，其下标的取值范围是 0 到 $N-1$ ，这个数组就是哈希表，亦可称为哈希数组。哈希数组中每个成员都是一个指针，可以指向一个存放 KV 对的结构体。如果想把某个 KV 对存放在该哈希数组中，就面临如何确定数组下标 x 的问题。通常的操作是用 H 除以 N ，取其余数，即取模运算， $x = G(H) = H \bmod N$ 。确定了 x 以后，该 KV 对就存放在数组成员 $T[x]$ 中，这个操作就是对哈希表的插入操作。在本节中，我们借鉴了 C 语言的语法，用符号 $\%$ 表示取模，用符号 $\&$ 表示比特的“与”运算 (bitwise and)。 $7 \% 3 = 1$, $11 \& 7 = 3$ ，这些都不难理解。

哈希数组中的每个成员被称为“桶” (bucket)，桶号就是数组的下标，从 0 开始，最大桶号是 $N-1$ 。插入哈希表的 KV 对被称为哈希表的“元素” (element)，你可以把哈希表中的元素想象成放在桶里的砖头，这样比较形象。对哈希表的操作有五种：创建哈希表 (creation)、销毁哈希表 (destroy)、插入元素 (insertion)、查找元素 (retrieval) 和删除元素 (deletion)。

不同的哈希值对 N 进行取模运算后可能结果相同，即 $G(H_1) = G(H_2)$ ，从而导致两个 KV 对都要被放在同一个桶中。这种一个桶里存放两块元素的情况被称为“冲突” (collision)。解决冲突最常见的方法就是做一个单向链表，把映射到同一个桶里的多个 KV 对串起来，这个单向链表被称为“冲突链” (collision chain)。如果一个冲

冲突链上有两个或者更多个元素，则表明该哈希表中有冲突的发生。如果哈希表所有的冲突链上最多只有一个元素，则该哈希表中没有冲突，这是一种理想状态。图 6.16 展示了哈希表的冲突，左边是极端冲突的情况，所有的元素都放在一个桶里，右边是一个桶里只放一个元素，是没有任何冲突的完美状态。

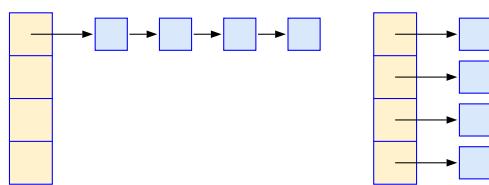


图 10.16: 哈希表的冲突

哈希表有一个概念，叫“装填因子”(fill factor)。假设某哈希表中一共有 b 个桶，有 e 个元素，则装填因子 $a = e / b$ 。装填因子是衡量哈希表冲突概率的重要参数，如果 $a > 1$ ，根据鸽笼原理，此时该哈希表中肯定有冲突。如果 $a \leq 1$ ，则表明冲突的概率比较低，并不能保证没有冲突。

知道了如何把元素插入到哈希表的过程后，拿着某个键 K 在哈希表中查找其值 V 的过程就容易理解了。查找过程的输入值是 K，哈希查找算法首先计算该元素在哈希数组中对应的下标 x，即 $x = G(f(K))$ 。查找算法直接在 $T[x]$ 指向的冲突链中进行查找就行了，根本不需要对整个数组进行遍历，所以哈希算法的查找速度和哈希数组的长度 N 没有关系。哈希查找算法的速度之所以快，其秘密就在这里：直接一步到位，无需遍历。当然，如果桶 $T[x]$ 的冲突链中有两个或者两个以上的元素，依然需要在冲突链中进行遍历，这是哈希查找过程中最容易降低查找性能的部分。所以为了提高哈希查找的速度，就要避免或者减少冲突，这需要设计良好的哈希函数 $f(x)$ ，尽可能把哈希值均匀地分布在一定范围，另一个技术就是本节介绍的动态哈希 (dynamic hashing)。

图 6.16 中涉及到两个函数， $f(x)$ 和 $G(x)$ ，函数 $G(x)$ 是用于确定哈希数组下标的，被称为“寻址函数”。你可以把 $G(x)$ 理解为一个更大的哈希函数 h 的一部分，即 $x = h(K) = G(f(K))$ 。哈希函数 $f(x)$ 的设计是一个比较烧脑的纯数学问题，无需智商普通的你我之辈去操心。寻址函数 G 对于理解动态哈希的核心思想至关重要，所以在后面的讨论中，我们更关注它。如果哈希数组的长度 N 可以变化，根据不同的 N 值， G 函数形成了一个函数家族。例如当哈希数组的长度为 N_1 时，为了定位某个元素，我们使用的寻址函数为 G_1 ， $G_1(x) = x \% N_1$ ，当哈希数组的长度变为 N_2 时，则用 G_2 ， $G_2(x) = x \% N_2$ ，依次类推。取模运算相对来说是比较耗时的操作，如果哈希表数组的长度 N 被设置为 2 的幂次方，即 $N = 2^n$ ($n=0,1,2,\dots$)，取模运算可以进一步被优化成比特的“与”运算。众所周知，位运算的速度是极快的，一条机器指令即可完成，这是哈希表工程实现中常使用的一个小技巧。例如，哈希表的长度被控制为 256，其下标的范围从 0 到 255，因为 $H \% 256 = H \& 255$ ，所以我们可以用与运算代替取模运算，下面的代码就不难理解了。

```
/* in src/backend/utils/hash/dynahash.c */
/* Fast MOD arithmetic, assuming that y is a power of 2 ! */
#define MOD(x,y) ((x) & ((y)-1))
```

10.2.1.2 动态哈希的基本原理

如果哈希表的长度 N 是固定的，这种哈希表被称为静态哈希表，如果 N 可以根据插入的元素的数量进行动态调整，则这种哈希表被称为动态哈希表。动态哈希表的长度可以变大也可以变小，但是更多的需求是长度变大。哈希表的长度也可以被叫做哈希表的体积，长度的增加即体积的扩容，本节后面的内容只专注在哈希表扩容方面。很显然，动态哈希表更能适应不同的应用场景，PostgreSQL 中的哈希表就是动态哈希表，本节探讨其技术背后的基本原理。PostgreSQL 动态哈希表的主要技术实现细节都在源代码文件 `dynahash.c` 中，它里面有这么一段注释：

```
/*
 * Original comments:
```

```

* Dynamic hashing, after CACM April 1988 pp 446-457, by Per-Ake Larson.
* Coded into C, with minor code improvements, and with hsearch(3) interface,
* by ejp@ausmelb.oz, Jul 26, 1988: 13:16;
* ...
*/

```

上面的注释清楚地表明，PostgreSQL 实现的动态哈希技术是以 Larson 教授的某篇论文为基础的，这篇论文很容易在互联网上免费获得，本节的内容就是对这篇论文的通俗讲解。Larson 教授提出的动态哈希技术的一个特点是：哈希表的每次扩容，只需增加一个桶，让整个哈希表缓慢地增加。

现在来考察一下图 6.17 所示的一个简单的哈希表。

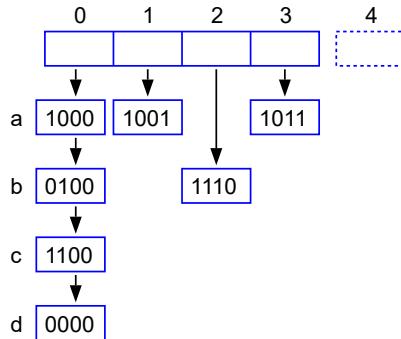


图 10.17: 一个简单的哈希表

若 N 表示哈希表的体积，该哈希表目前有 4 个桶，即 $N = 4$ ，桶号从 0 到 3，在后文的讨论中，我们总是假设 N 是 2 的幂次方。哈希表中所有的元素都是经过寻址函数 G 运算后分配的， $G(x) = x \% 4$ 。为了方便大家理解，图中哈希表的各元素的哈希值都以二进制的形式表示。你很容易观察到一个规律：0 号桶里所有元素的最低两位比特是 00，1 号桶里所有元素的最低两位比特是 01，2 号桶里所有元素的最低两位比特是 10，3 号桶里所有元素的最低两位比特是 11。请稍微留意一下 0 号桶里面的 4 块元素：a 和 d 的最低三位比特是 000，而 b 和 c 的最低三位比特是 100，这是一个关键点。

该哈希表有 7 个元素，所以装填因子 $a = 1.75$ ，说明有冲突，应该增加新桶来降低装填因子。新桶增加后，一部分元素需要从老桶转移到新桶，降低冲突链的长度。图 6.18 展示了增加新桶后如何转移元素的过程。

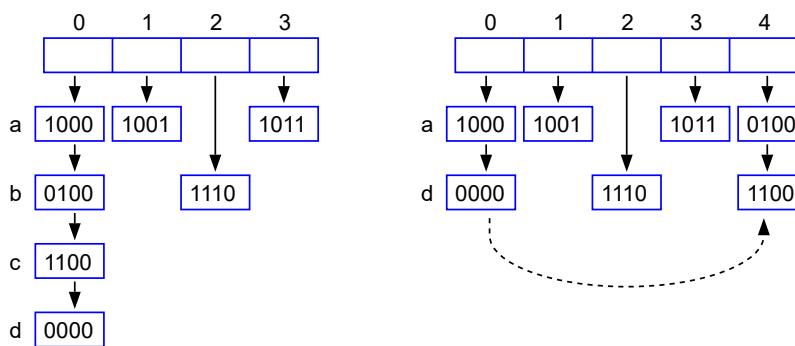


图 10.18: 元素的转移过程

当增加 4 号桶后，哈希表一共有 5 个桶。 G 函数的寻址范围是 0 - 3，已经不能满足要求了。此时动态哈希算法会启用新的寻址函数 $g(x) = x \% 8$ ，其寻址范围是 0 - 7，满足此时的需要。在新的寻址函数 g 的统治下，哈希表中所有元素的地址都要被重新计算。放入 4 号桶里的元素有一个特征，就是它们最低三位比特应该是 100。或许你早已发现了一个规律：1 号到 3 号桶的元素经过 g 函数的计算后，不可能放在 4 号桶，因为它们最低三位

比特是 $x01, x10, x11$ 的形式 (x 为 0 或者 1)。被转移的元素只可能在 0 号桶里，因为 0 号桶里的元素最低三位比特有 000 和 100 两种值，最低位三个比特是 100 的元素要放入在 4 号桶里，最低位三个比特是 000 的元素依然在 0 号桶中，不需要转移。经过寻址函数 g 的计算，0 号桶里的 b 和 c 被转移到了 4 号桶，而 a 和 d 依然留在了 0 号桶。这种把老桶中的一部分元素转移到新桶的过程被称为桶的“分裂”(bucket split)。寻址函数 G 和 g 有共同的规律，可以抽象为一个更加通用的形式，如图 6.19 所示。

$$G_i(x) = x \bmod 2^i$$

图 10.19: 寻址函数 G 的通用形式

虽然这个公式看起来很高大上，其实很简单，下面的 G_i 系列函数都是非常容易理解的。

- $G_2(x) = x \% 4$
- $G_3(x) = x \% 8$
- $G_4(x) = x \% 16$
- $G_5(x) = x \% 32$
- $G_6(x) = x \% 64$

在 G_i 函数家族的统治下，有一个非常好的规律，每次增加一个新桶，只可能有一个老桶被分裂，其余桶里的元素纹丝不动。在后文中，老桶这个术语专门指被分裂的桶。理解了桶分裂的过程后，我们考察一下动态哈希表的扩容过程，如图 6.20 所示。

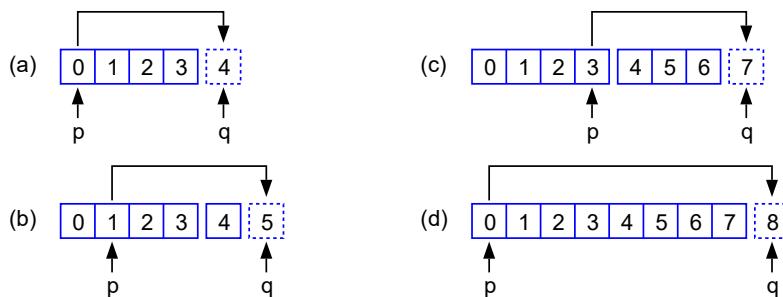


图 10.20: 动态哈希的扩容过程

情景 (a) 表示哈希表的第一次扩容，所使用的寻址函数是 G_2 ，其寻址范围为 0 - 3。增加 4 号新桶后，因为哈希表有了 5 个桶， G_2 函数的寻址空间不够用了，此时就要使用 G_3 来把哈希表中所有的元素进行重新分配。函数 G_3 的寻址范围是 0 - 7，在新增加的桶的桶号变为 8 之前都是够用的。图 6.20 中有两个指针，指针 p 指向老桶，指针 q 指向新桶，你很容易看出新桶和老桶的编号之间存在的规律： $q = p + N$ 。在此我们引入一个术语：“当前体积”，它表示哈希表中目前有多少个桶，此时哈希表有 5 个桶，所以该哈希表的当前体积是 5，即 $q+1$ 。

情景 (b) 展示了哈希表被扩容到 5 号桶的情况。此时整个哈希表中有 6 个桶，没有超过 G_3 函数的寻址范围，所以依然可以用 G_3 来计算每一块元素的位置。当增加 5 号桶后，要分裂的老桶只有 1 号桶，新桶为 6 时对应的老桶就是 2 号桶。

情景 (c) 展示了扩容到 7 号桶时的情况，此时老桶是 3 号桶，但情景 (c) 和情景 (b) 稍有不同，此时哈希表里有 8 个桶了，函数 G_3 的寻址范围是 0 - 7，已经处于临界状态了。如果再继续增加一个桶的话， G_3 函数不够用了，就需要使用 G_4 函数来重新计算所有元素的位置了，因为 G_4 的寻址范围是 0 - 15。

情景 (d) 展示了继续增加一个新桶的情况，此时哈希表的当前体积是 9，超过了寻址函数 G_3 的范围，需要启用 G_4 。根据 G_4 ，新桶 8 对应的老桶应该是 0 号桶，即 p 指针被归零了，这是情景 (d) 和 (a)(b)(c) 最大的不同。为

了更容易理解上述内容描述的扩容过程，在此引入一个概念：哈希表的最大体积 M 。假设哈希表的当前体积为 m ，如果 m 是一个 2 的幂次方，则 $M = m$ ，否则 M 是比 m 大且最接近 m 的一个 2 的幂次方。

情景 (a)(b)(c) 的当前体积分别是 5、6 和 8，在这三种情景下该哈希表的最大体积均是 8。情景 (c) 下哈希表的当前体积已经等于其最大体积了，即该哈希表已经满了，如果再增加一个桶，势必要把最大体积变大。哈希表的当前体积超过最大体积时，动态哈希算法会先把最大体积 M 翻倍，然后再增加新桶。情景 (d) 中，因为要增加第 9 个桶， M 先由 8 变成了 16，再增加第 9 号桶(其桶号为 8)。我们还可以引入一个术语“世代”(generation)来表示哈希表的不同状态。假设 $M = 2^n$ ，则称该哈希表处于 n 世代。情景 (a)(b)(c) 的哈希表都处于 3 世代，而情景 (d) 的哈希表处于 4 世代，任何一个世代的最大体积都是前一个世代的两倍，前一个世代的哈希表变成了下一个世代的哈希表的前半部分。

在情景 (d) 中，9 号桶对应的老桶是 1 号桶，15 号桶对应的老桶是 7 号桶。当新桶编号为 16 时，整个哈希表中有 17 个桶， G_4 函数又不够用了，就需要启用 G_5 ，此时该哈希表进入了 5 世代，以此类推，后面的场景读者不难想象和理解。

这种哈希表的扩容和桶分裂的方式是按照预定的顺序依次展开的，这种技术被称为线性哈希 (linear hashing, LH)。线性哈希技术因其简单高效，成为动态哈希技术的主要方式。动态哈希表有两个体积在增长，一个是当前体积 m ，它每次增加一个桶，另一个是最大体积 M ，当 $m > M$ 时， M 就会扩大一倍，这其中的区别请读者稍加留意。

世代为 n 的哈希表使用的寻址函数是 G_n ，此时该哈希表的前半部分的元素使用的寻址函数是 $G_{(n-1)}$ ，所以在任何时刻，只有 G_n 和 $G_{(n-1)}$ 两个寻址函数起作用。PostgreSQL 的源代码中设置了高低两个 mask 值， $high_mask$ (简称为 H_m) 和 low_mask (简称为 L_m)，分别代表这两个寻址函数。我们用 B_{old} 表示老桶的编号， B_{new} 表示新桶的编号，在哈希表处于 n 世代时，则下面的公式均成立：

- $H_m = 2^n - 1$
- $L_m = H_m - 2^{n-1}$
- $G_n(x) = x \& H_m$
- $G_{(n-1)}(x) = x \& L_m$
- $B_{old} = B_{new} \& L_m$

图 6.21 展示了哈希表从前一个世代变成下一个世代的过程。

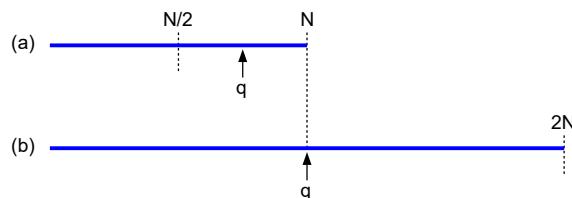


图 10.21：哈希表的世代变迁

在情景 (a) 中，哈希表的最大体积为 N ，指针 q 表示目前的最大桶号，其初始值被设置为 $N/2-1$ ，所以 q 大于等于 $N/2-1$ ，且小于 N 。编号从 0 到 q 的桶里可能有元素，也可能没有，但编号从 $q+1$ 到 $N-1$ 的桶里肯定没有元素。可能装有元素的桶分布在两个地方：从 0 到 $(N/2 - 1)$ ，从 $(N/2)$ 到 q ，理解了这一点，由哈希值 H 计算它在哈希表中的桶号 x 的算法就很容易理解了，下面的伪代码展示了计算哈希值 H 对应的桶号 x 的算法。

```

H = f(K);
Hm = N - 1;
Lm = N/2 - 1;
x = H & Hm;
if(x > q) x = H & Lm;

```

其中 K 表示键, H_m 就是 H_m , L_m 就是 L_m 。情景 (b) 描述了 $q = N$ 的临界状态, 此时哈希表的当前体积是 $N + 1$, 已经突破的原来的最大体积 N , 最大体积会增加一倍, 变成 $2N$, 同时要调整 H_m 和 L_m 的值: $H_m = 2N - 1$, $L_m = N - 1$ 。随着哈希表的继续扩容, q 从 N 一直增加到 $2N - 1$, 在 $q = 2N$ 之前, 整个哈希表的最大体积保持不变, L_m 和 H_m 的值亦保持不变。当 $q = 2N$ 时, 该哈希表的最大体积变为了 $4N$, L_m 调整为 $2N - 1$, H_m 的值调整为 $4N - 1$, 依次类推。

后文中把哈希表的“扩容”定义为增加一个新桶, 把“世代交替”定义为哈希表的最大体积翻倍。随着扩容和世代交替, 整个哈希表的当前体积和最大体积都在不断增加, 其中最大体积是指数级增加, 而当前体积是缓慢增加, 每次增加一个桶。虽然哈希表的最大体积是指数级增加, 但它是一个虚拟的概念, 描述最大体积的信息并不会占用很多内存, 真正占内存的是当前体积, 所以动态哈希表实际的体积增长是缓慢的, 平滑的。以上就是 Larson 教授的动态哈希表的基本思想, 他的论文中对这种动态哈希技术的性能进行了枯燥的数学分析, 我们就不操心了。

10.2.2 PostgreSQL 的内存哈希表

理解了 Larson 教授的动态哈希技术的基本原理后, 本节的内容就来解剖 PostgreSQL 如何基于以上理论来实现动态哈希表及其相关插入和查找等算法的。PostgreSQL 提供了如下的哈希函数, 其中 `hash_bytes` 是最底层的函数, 它可以把指定长度的 k 运算成一个 4 字节的哈希值 h, 其余的函数都是在其基础之上根据不同的应用进行的封装。这些函数都是纯数学问题, 我们不需要操心其细节。这些不同的哈希函数, 不论输入参数如何变化, 它们的返回结果都是固定的 4 个字节, 这 4 个字节的哈希值在源代码中又被称为“哈希码”(Hash Code)。

```
/* in src/common/hashfn.c */
uint32 hash_bytes(const unsigned char *k, int keylen) { ... }
uint32 hash_bytes_uint32(uint32 k) { ... }
uint32 string_hash(const void *key, Size keysiz) { ... }
uint32 tag_hash(const void *key, Size keysiz) { ... }
uint32 uint32_hash(const void *key, Size keysiz) { ... }
```

10.2.2.1 相关的数据结构

程序等于数据结构加算法, 熟悉算法的数据结构是理解算法的基础。在源代码中, 有三个基础性的数据结构: `HASHELEMENT`, `HASHBUCKET` 和 `HASHSEGMENT`, 它们的具体定义如下:

```
/* in src/include/utils/hsearch.h */
typedef struct HASHELEMENT {
    struct HASHELEMENT *link; /* link to next entry in same bucket */
    uint32      hashvalue; /* hash function result for this entry */
} HASHELEMENT;
/* in src/backend/utils/hash/dynahash.c */
typedef HASHELEMENT *HASHBUCKET; /* A hash bucket is a linked list of HASHELEMENTs */
typedef HASHBUCKET *HASHSEGMENT; /* A hash segment is an array of bucket headers */
```

`HASHELEMENT` 代表哈希表中的一个元素, 即 KV 对, 图 6.22 展示了一个冲突链上的三块元素。

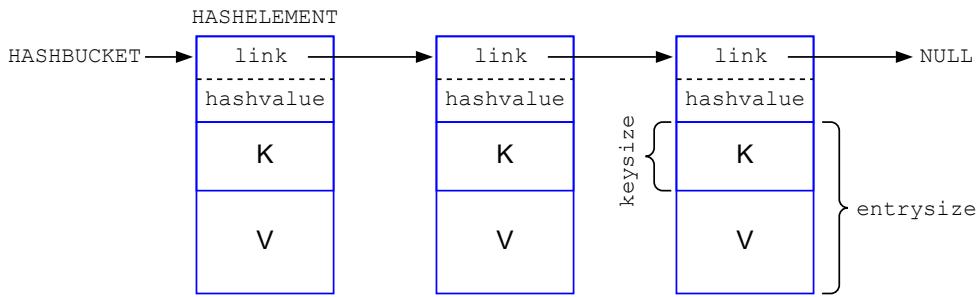


图 10.22: HASHELEMENT 的结构

结构体 HASHELEMENT 实际上定义了一个元素的头部结构，该头部结构包含了一个指针 link，用于形成冲突链的单向链表，其后是固定四个字节的哈希值。在 HASHELEMENT 结构的后面，才是 KV 对，这个由使用者自己定义其内容。PostgreSQL 动态哈希表中的 K 和 V 的长度都是统一的，所以只要在哈希表总控的数据结构中设置两个变量 keysize 和 entrysize 即可，不需要在每个 HASHELEMENT 中都保存长度信息。变量 keysize 是 K 的长度，entrysize 是 K 和 V 的总长度，而不是 V 的长度，所以哈希表中一个元素的总长度 elementSize 可以用如下公式计算。

```
/* in src/backend/utils/hash/dynahash.c */
Size elementSize = MAXALIGN(sizeof(HASHELEMENT)) + MAXALIGN(entrysize);
```

因为 HASHELEMENT 是一个 8 字节的指针和一个 4 字节的哈希值，共 12 个字节，按 8 字节取整，就要占据 16 个字节，其中 4 个字节被浪费掉了。对于一个上百万元素的哈希表来说，这部分浪费的字节就有好几 MB，我认为这是需要改进的地方。源代码中还定义了由指向 HASHELEMENT 的指针如何拿到指向 K 的指针的方法，以及由指向 K 的指针如何得到指向 HASHELEMENT 的指针的方法，无非就是前后移动 MAXALIGN(sizeof(HASHELEMENT)) 个字节而已，其具体代码如下。

```
/* in src/backend/utils/hash/dynahash.c */
/* Key (also entry) part of a HASHELEMENT */
#define ELEMENTKEY(helem) (((char *) (helem)) + MAXALIGN(sizeof(HASHELEMENT)))
/* Obtain element pointer given pointer to key */
#define ELEMENT_FROM_KEY(key) \
    ((HASHELEMENT *) (((char *) (key)) - MAXALIGN(sizeof(HASHELEMENT))))
```

HASHELEMENT 代表着哈希表中的一个桶，它是指向某个冲突链的指针。在理论探讨部分，我们说哈希数组就是哈希表，从现在起，我们严格区分哈希数组和哈希表这两个术语。哈希数组和上节理论探讨中的哈希表或者哈希数组含义相同，而在复杂的工程实现中，哈希表是在哈希数组之上一个更大的概念，这就是为什么有 HASHBUCKET 和 HASHSEGMENT 两个数据结构的原因。这两个数据结构的关系请参考图 6.23。

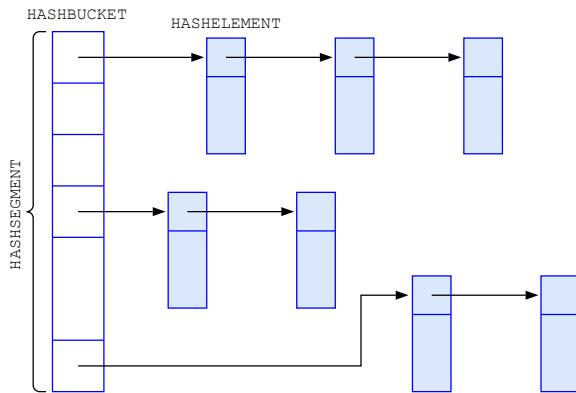


图 10.23: HASHSEGMENT、HASHBUCKET 和 HASHELEMENT 的关系

图 6.23 中的哈希数组，被称为一个“段”(segment)，由 HASHSEGMENT 来表示，在哈希数组之上，我们可以进一步构造出一个二维数组，称为“目录”(dir)，请参考图 6.24。

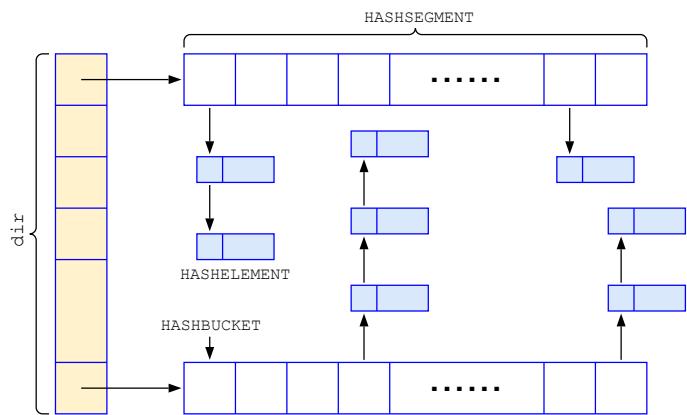


图 10.24: 目录和段的关系

目录也是一个数组，它的每个成员都是一个指针，指向某个段。目录数组加上段数组，才构成一个哈希表的完整概念，这种设计是为了实现哈希表的动态扩展功能。你可以这样理解和记忆：目录是一个指针数组，每一个指针指向一个段，段也是一个指针数组，里面的成员是桶，一个桶就是指向一个冲突链的指针。目录数组的长度是由变量 `dsize` 来表示，其变量名的含义是“dir size”。段的长度由 `ssize` 变量来记录，其变量名的含义是“segment size”。在哈希表被创建后，`ssize` 就保持固定不变，且它的值设置为 2 的幂次方。学习完以上概念和它们之间的关系后，我们可以用图 6.25 来理解动态哈希表的整体结构。

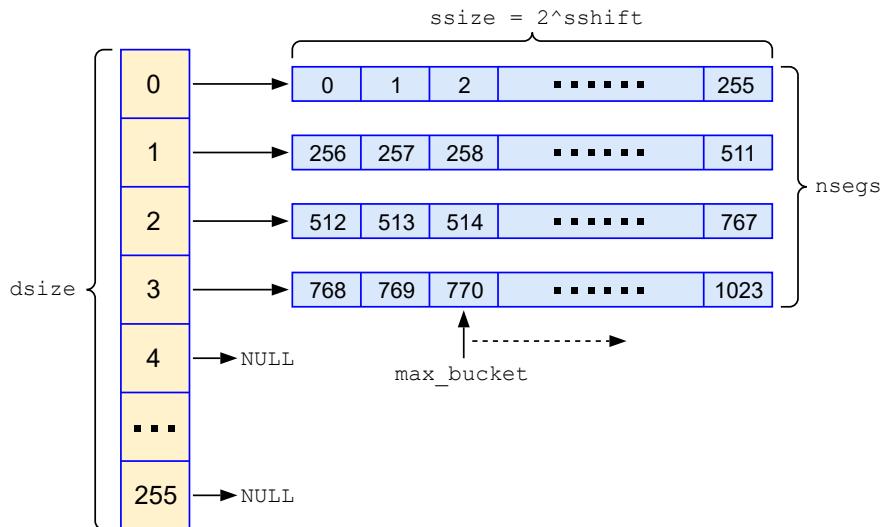


图 10.25: 哈希表的整体结构

由上可知，哈希表中最多有 $dsiz * ssize$ 个桶。变量 max_buket ，相当于图 6.20 中的 q 指针，它记录着哈希表的当前体积：从 0 到 max_buket 的桶里可能装有元素，也可能是空桶，但桶号大于 max_buket 的桶肯定是空桶。譬如 $max_buket = 770$ ，表示从 0 到 770 号的这 771 个桶里可能有元素，任何桶号大于 770 的桶里都没有元素。当哈希表中元素的总个数大于 $max_buket + 1$ 时，即装填因子 $a > 1$ ，某些桶里肯定存放了两块或更多块元素，此时就要增加一个新桶， max_buket 加 1，往右移动，此即动态哈希表的扩容。

哈希表中的段是按需分配的，由变量 $nsegs$ 记录。图 6.25 中目录数组的长度 $dsiz$ 是 256，但目前只分配了 4 个段，所以 $nsegs = 4$ ， $dir[0]$ 到 $dir[3]$ 的四个指针指向了 4 个段，而 $dir[4]$ 到 $dir[255]$ 的指针都是 $NULL$ 值。哈希表的最大体积为 $dsiz * ssize$ ，实际上有 $nsegs * ssize$ 个桶，当前体积则为 $max_buket + 1$ 。哈希表的当前体积等于其最大体积时，如果还需要继续扩容的话，就要进行世代交替，哈希表进入下一个世代，最大体积加倍，把 $dsiz$ 乘以 2，具体代码可以参考 `dynahash.c` 中的 `dir_realloc()` 函数，这些和上一节讨论的理论完全一致。源代码为 $dsiz$ 和 $ssize$ 设置了缺省值，具体如下：

```
/* in src/backend/utils/hash/dynahash.c */
#define DEF_SEGSIZE      256
#define DEF_SEGSIZE_SHIFT    8 /* must be log2(DEF_SEGSIZE) */
#define DEF_DIRSIZE      256
```

其中 $DEF_SEGSIZE$ 是 $ssize$ 的缺省值， $DEF_DIRSIZE$ 是 $dsiz$ 的缺省值。另外一个变量 $sshift$ 是 $ssize$ 以 2 为底的对数，如果 $ssize = DEF_SEGSIZE$ ，即 2^8 ，则 $sshift = 8$ ，即 $DEF_SEGSIZE_SHIFT$ 。设置变量 $sshift$ 是为了加快运算的速度。在哈希查的过程中，经常需要用桶号 $bucket$ 除以 $ssize$ 来确定该桶所在的段号，此时只要把 $bucket$ 右移 $sshift$ 个比特，就可以实现除以 $ssize$ 的目的。很显然位右移的速度快于除法的速度，这些都是优化性能的一些小技巧，值得我们玩味和学习。

如果要获得桶号为 770 的桶的指针，该怎么做呢？从图 6.25 中可知，770 号桶保存在 $dir[3]$ 指向的段里面，且是这个段的第 3 个元素，编号是 2。对照这个例子，下面的代码的逻辑就不难理解了。

```
/* in src/backend/utils/hash/dynahash.c : hash_search_with_hash_value() */
uint32      bucket;
long        segment_num, segment_ndx;
HASHSEGMENT segp;
HASHEBCKET *prevBucketPtr;
/* bucket 代表要寻找的桶的编号 */
segment_num = bucket >> hashp->sshift; /* 计算该桶所在的段号 */
```

```

segment_ndx = MOD(bucket, hashp->ssize); /* 计算该桶在该段中的编号 */

segp = hashp->dir[segment_num]; /* segp指向了bucket的桶所在的段 */
prevBucketPtr = &segp[segment_ndx]; /* 拿到了指向该桶的指针 */

```

理解了内存哈希表的大体结构，我们就要学习哈希表的“总控数据结构” HTAB。这里有三个结构体：HTAB、HASHHDR 和 HASHCTL，让初学者发晕。其实此三者的含义和关系不难理解，其中的 HASHCTL 是最不重要的，它实际上是一堆参数的打包。在一个哈希表被创建时，调用者需要指定一系列的参数，如上面的 dsize, ssize, keysize, entrysize 等等，HASHCTL 实际上就是把这些参数打包在一起，作为输入参数传入到哈希表的创建函数 hash_create() 中。哈希表创建完成后，就没有它什么事了。结构体 HASHCTL 的具体定义如下：

```

/* in src/include/utils/hsearch.h */
typedef struct HASHCTL {
    long          num_partitions; /* # partitions (must be power of 2) */
    long          ssize;         /* segment size */
    long          dsize;         /* (initial) directory size */
    long          max_dsize;     /* limit to dsize if dir size is limited */
    Size          keysize;       /* hash key length in bytes */
    Size          entrysize;     /* total user element size in bytes */
    HashValueFunc hash;         /* hash function */
    HashCompareFunc match;      /* key comparison function */
    HashCopyFunc  keycopy;      /* key copying function */
    HashAllocFunc alloc;        /* memory allocator */
    MemoryContext hcxt;        /* memory context to use for allocations */
    HASHHDR      *hctl;         /* location of header in shared mem */
} HASHCTL;

```

结构体 HASHCTL 的大部分成员变量和 HTAB, HASHHDR 中的成员变量相对应，这里先略过。真正控制哈希表行为的数据结构是 HTAB 结构体，那么为什么又要搞出来 HASHHDR 这个东东呢？这是因为 PostgreSQL 的哈希表可能存放在私有内存中，也可能存放在共享内存中。哈希表的各种参数分为两类，一类是哈希表创建后就始终保持不变的参数，譬如 ssize, sshift, keysize, entrysize 等，第二类是随着哈希表中元素的增加和减少而不断变化的参数，如 dsize, max_bucket 等。对于存放在共享内存中的哈希表，设计者把 HTAB 中变化的参数单独摘出来，做成了一个独立的数据结构 HASHHDR，也存放在共享内存中，方便各进程访问。而自哈希表创建后就不变的信息，则保存在驻留于私有内存中的 HTAB 里。

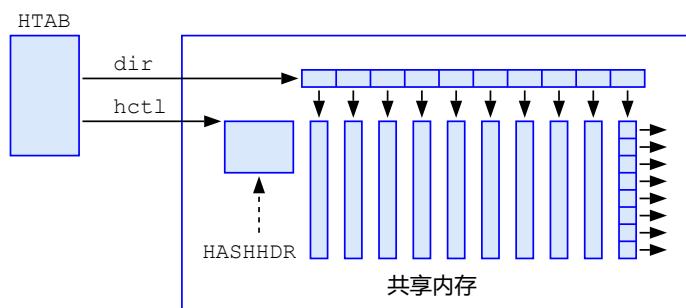


图 10.26: 共享内存中的哈希表

图 6.26 显示了哈希表存放在共享内存的情景。保存哈希表动态变化信息的 HASHHDR 被存放在共享内存中，在私有内存中的 HTAB 有一个指针 hctl 指向了它。当某个后台进程需要获取哈希表中不变的信息时，只需要访问私有内存，无需访问共享内存。访问共享内存的代价肯定要比访问私有内存昂贵，因为访问共享内存时

需要进行加锁和解锁等操作来协调各个进程，因此有必要把 HTAB 和 HASHHDR 分开，这是一个很聪明的设计。你会在后面的 HASHHDR 和 HTAB 的定义中看到，成员变量 keyszie, ssize 和 sshift 在两个结构体中都有，这种重复就是要把共享内存中不变的信息复制到私有内存中，降低并发竞争的几率。

在共享内存中的哈希表一旦创建，其尺寸，即 dsize 和 ssize，都会保持不变，因此该哈希表中全部桶的数量是固定的，不能随着插入其中的元素的增加而增加，其具体原因我们在学习共享内存的分配机制后就会明白。假设哈希表有 m 个桶，插入其中的元素的总数目是 n ，如果 $n > m$ ，就会出现冲突，降低了哈希表查找的速度。哈希表创建函数 hash_create() 有一个参数 nelem 让调用者指定打算往该哈希表中放入元素的总个数，调用者在共享内存中创建哈希表之前，需要预估 nelem 的最大值，以及 dsize 和 ssize 的尺寸，尽量确保一个桶里只有一块砖的原则。为了提供更细粒度的并发控制，在共享内存中的哈希表被分为不同的区 (partition)。两个进程同时访问同一个哈希表中不同的区是没有竞争的，因为不同的分区由不同的锁进行保护，这种分区设计提高了并发度和系统性能。总结一下：存储在共享内存中的哈希表是不能动态扩展的，但是可以分区。放在私有内存中的哈希表则和它截然相反，即私有内存的哈希表可以动态扩展，但是不分区。

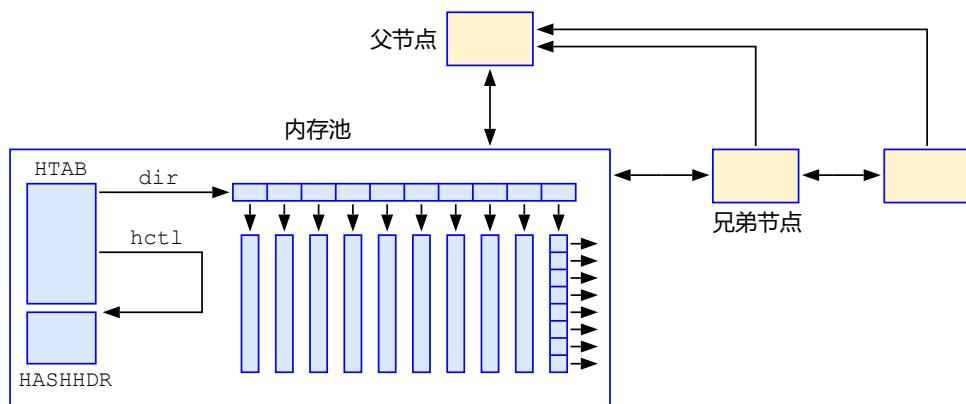


图 10.27: 私有内存中的哈希表

图 6.27 表示了哈希表在私有内存中的情景。在此情况下，你可以把 HASHHDR 视为 HTAB 的一部分，两者是一个整体。在私有内存中的哈希表因为只供本进程使用，所以无需分区，但可以随着插入其中的元素的增加而自动扩容，因此，在私有内存中创建哈希表时，调用者不需要预估未来往该哈希表中存放的元素的最大数目，只需要指定一个大于 0 的初始值即可。

在私有内存中的哈希表被保存在一个单独的内存池中，该内存池由 HTAB 的成员变量 hcxt 来记录。所以销毁哈希表的操作非常简单，只需把该内存池销毁即可。理解了这个思想，即使我们不了解 HTAB 和 HASHHDR 的具体内容，也很容易看懂哈希表的销毁函数 hash_destroy()，其完整代码如下：

```
/* in src/backend/utils/hash/dynahash.c */
void hash_destroy(HTAB *hashp) {
    if (hashp != NULL) {
        /* Free everything by destroying the hash table's memory context. */
        MemoryContextDelete(hashp->hcxt);
    }
}
```

啰嗦了这么多的理论，下面我们就来撸代码，看一下 HTAB 和 HASHHDR 这两个数据结构的庐山真面目，其定义如下：

```
/* in src/backend/utils/hash/dynahash.c */
#define NUM_FREELISTS 32
```

```

struct HASHHDR {
    FreeListData freeList[NUM_FREELISTS];
    /* These fields can change, but not in a partitioned table */
    /* Also, dsize can't change in a shared table, even if unpartitioned */
    long    dsize;           /* directory size */
    long    nsegs;          /* number of allocated segments (<= dsize) */
    uint32 max_bucket;     /* ID of maximum bucket in use */
    uint32 high_mask;      /* mask to modulo into entire table */
    uint32 low_mask;       /* mask to modulo into lower half of table */
    /* These fields are fixed at hashtable creation */
    Size    keysize;        /* hash key length in bytes */
    Size    entrysize;      /* total user element size in bytes */
    long    num_partitions; /* # partitions (must be power of 2), or 0 */
    long    max_dsize;      /* 'dsize' limit if directory is fixed size */
    long    ssize;          /* segment size --- must be power of 2 */
    int    sshift;          /* segment shift = log2(ssize) */
    int    nelem_alloc;     /* number of entries to allocate at once */
};

struct HTAB {
    HASHHDR      *hctl;      /* => shared control information */
    HASHSEGMENT   *dir;       /* directory of segment starts */
    HashValueFunc hash;      /* hash function */
    HashCompareFunc match;   /* key comparison function */
    HashCopyFunc   keycopy;   /* key copying function */
    HashAllocFunc  alloc;     /* memory allocator */
    MemoryContext  hcxt;     /* memory context if default allocator used */
    char         *tabname;   /* table name (for error messages) */
    bool         isshared;   /* true if table is in shared memory */
    bool         isfixed;    /* if true, don't enlarge */
    /* freezing a shared table isn't allowed, so we can keep state here */
    bool         frozen;     /* true = no more inserts allowed */
    /* We keep local copies of these fixed values to reduce contention */
    Size    keysize;        /* hash key length in bytes */
    long    ssize;          /* segment size --- must be power of 2 */
    int    sshift;          /* segment shift = log2(ssize) */
};

```

这两个结构体成员众多，结构复杂，我们可以这样理解：其一是把两者视为一个整体，HASHHDR 是 HTAB 的一部分，其次是区分两个数据结构中自哈希表创建之后变和不变的成员变量。下面列出到目前为止我们能够理解的成员变量的含义，对于尚不能够理解的成员变量，结合后面的代码走读，也很容易搞明白。

- dsize、ssize 和 sshift 的含义，前文已经介绍过了，这里不再赘述。
- nsegs 表示 dir 数组中已经分配段的数量。
- max_bucket 的含义前文已经讨论过了。
- high_mask 和 low_mask 对应上一节理论讨论中的 H_m 和 L_m ，这兄弟俩是哈希表动态增长时非常关键的变量。
- keysize 和 entrysize 的含义请参考图 6.22。
- num_partitions 在私有内存模式下的哈希表中恒定为 0，表示不分区。
- nelem_alloc 表示一次性从内存池中批发多少个空白的元素过来，后面的代码分析中会讨论。

- hash/match/keycopy/alloc 是一捆函数，分别涉及到哈希计算，键匹配，键拷贝和内存分配等内容，其含义不难理解。
 - ishared, isfixed, frozen 三个布尔型变量表示哈希表是否在共享内存中，是否被冻结，从而不能插入新的 KV 对，是否固定，不能够动态增加等。

为了代码复用，设计者同时考虑了哈希表在共享内存和私有内存两种情况的处理，所以代码逻辑比较绕，不容易理解。由于目前我们还不具备共享内存方面的知识，所以后面的内容只讨论在私有内存中的动态哈希表。我在展示代码时会把涉及到共享内存部分的代码删除，这样整个逻辑就清晰多了。读者只需比较本文引用的代码和源程序的代码的不同，就很容易区分其中的差异。

在 HASHHDR 中有一个数组 freeList 用于保存空闲元素的单向链表，在私有内存模式下的哈希表只有一个分区，所以只使用了 freeList[0]。这些空闲元素的来源有两个，一个是一次性从内存池中批发的若干元素，批发的数量由 nelem_alloc 指定，第二个来源就是从哈希表中删除的元素也不会返还给内存池，而是存放在这个空闲列表中。每次往哈希表中插入新元素时，就会在这个空闲链表中寻找候选对象，如果找不到了，就再从内存中批发一批新的元素放在这个单向链表中。图 6.28 演示了 freeList 和哈希表之间的关系。

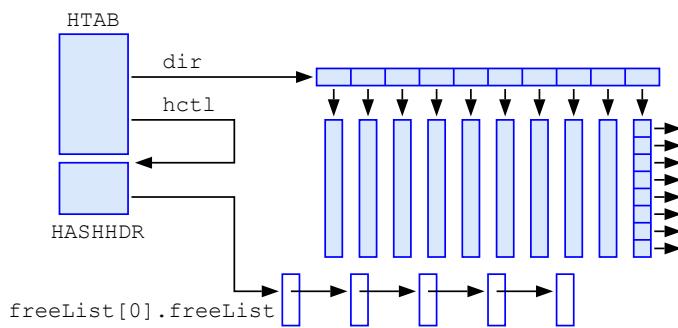


图 10.28: 空闲链表和哈希表的关系

数组 freeList 的数据结构 FreeListData 的定义如下：

```
/* in src/backend/utils/hash/dynahash.c */
typedef struct {
    slock_t      mutex;      /* spinlock for this freelist */
    long         nentries;   /* number of entries in associated buckets */
    HASHELEMENT *freeList;   /* chain of free elements */
} FreeListData;
```

这里要特别强调的是：`freeList[0].nentries` 记录了哈希表中所有元素的数量，而不是 `freeList[0].freeList` 这个单向链表中的空闲元素的数量。当往哈希表中插入一个元素时，`freeList[0].nentries` 的值就会加 1。当从哈希表中删除一个元素时，该值就会减 1，这是初学者容易理解错误的地方。变量 `mutex` 在私有内存模式下则没有任何用处。

10.2.2.2 相关的函数代码分析

熟悉了相关数据结构，下面开始分析相关函数。我们遵循着“先易后难”的原则，先分析两个比较简单的函数，然后再涉及复杂的函数。函数 `element_alloc()` 会一次性从内存池中申请 `nelem_alloc` 个元素的内存，把它们组成一个链表结构，挂在 `freeList[0].freeList` 上。当需要往哈希表中插入元素时，新元素所使用的内存直接从 `freeList[0].freeList` 上获得，不需要再向内存池申请了，这是一个提高性能的措施。下面我们走读一下该函数的源代码。

```
static bool element_alloc(HTAB *hashp, int nelem)
```

```

{
    HASHHDR      *hctl = hashp->hctl;
    Size         elementSize;
    HASHELEMENT *firstElement, *tmpElement, *prevElement;

    if (hashp->isfixed) return false; /* 如果本哈希表时固定的，就啥也不做，因为不能插入 */
    /* Each element has a HASHELEMENT header plus user data. */
    elementSize = MAXALIGN(sizeof(HASHELEMENT)) + MAXALIGN(hctl->entrysize);

    CurrentDynaHashCxt = hashp->hcxt;
    /* 从内存池中申请nelem * elementSize字节大小的连续内存 */
    firstElement = (HASHELEMENT *) hashp->alloc(nelem * elementSize);
    if (!firstElement) return false;
    /* prepare to link all the new entries into the freelist */
    prevElement = NULL;
    tmpElement = firstElement;
    for (int i = 0; i < nelem; i++) { /* 把这些连续的内存组成单向链表 */
        tmpElement->link = prevElement; prevElement = tmpElement;
        tmpElement = (HASHELEMENT *) (((char *) tmpElement) + elementSize);
    }
    /* 把空闲元素的链表挂在freeList[0].freeList上 */
    firstElement->link = hctl->freeList[0].freeList;
    hctl->freeList[0].freeList = prevElement;
    return true;
}

```

函数 `element_alloc()` 执行完后，它的内存情况可以用图 6.29 来帮助理解。

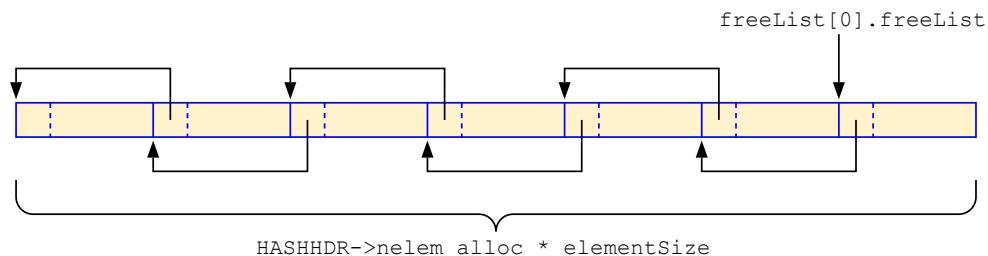


图 10.29: 一次性从内存池批发若干块元素

函数 `get_hash_entry()` 是从 `freeList[0].freeList` 指向的单向链表中拿到一个空闲元素，我把它的代码做了最大程度的简化，理解起来非常容易。

```

static HASHBUCKET get_hash_entry(HTAB *hashp)
{
    HASHHDR      *hctl = hashp->hctl;
    HASHBUCKET  newElement;

    for (;;) { /* try to get an entry from the freelist */
        newElement = hctl->freeList[0].freeList;
        if (newElement != NULL) break;
        if (!element_alloc(hashp, hctl->nelem_alloc)) return NULL;
    }
}

```

```

    }

    /* remove entry from freelist, bump nentries */
    hctl->freeList[0].freeList = newElement->link;
    hctl->freeList[0].nentries++;
    return newElement;
}

```

下面走读一下哈希表创建函数 `hash_create()` 的代码。在该函数的入口参数中，`tabname` 指向一个字符串，用于 `debug` 使用。和内存池的名字不同，该字符串不需要是一个常量，因为 `hash_create()` 会把该字符串的内容复制下来。参数 `nelem` 记录着调用者打算往该哈希表中插入的元素的总数，对于私有内存的哈希表，随便指定一个大于 0 的数字即可，因为动态哈希表可以不断扩容。`info` 和 `flags` 都是各种参数的信息，创建哈希表时会参考这些参数。

```

/* in src/backend/utils/hash/dynahash.c */
HTAB * hash_create(const char *tabname, long nelem, const HASHCTL *info, int flags)
{
    HTAB      *hashp;
    HASHHDR   *hctl;

    if (flags & HASH_SHARED_MEM) { /* 共享内存中的哈希表，跳过 */ }
    else {
        /* Create the hash table's private memory context */
        if (flags & HASH_CONTEXT) CurrentDynaHashCxt = info->hcxt;
        else CurrentDynaHashCxt = TopMemoryContext;
        /* 创建一个独立的内存池用于存放本哈希表 */
        CurrentDynaHashCxt = AllocSetContextCreate(CurrentDynaHashCxt,
                                                    "dynahash", ALLOCSET_DEFAULT_SIZES);
    }

    /* 从内存池中申请一块小内存，用于存放HTAB的信息和tabname字符串的信息 */
    /* Initialize the hash header, plus a copy of the table name */
    hashp = (HTAB *) DynaHashAlloc(sizeof(HTAB) + strlen(tabname) + 1);
    MemSet(hashp, 0, sizeof(HTAB));
    hashp->tabname = (char *) (hashp + 1);
    strcpy(hashp->tabname, tabname);

    /* 选择合适的哈希函数，我们可以跳过这段冗长的代码
     * Select the appropriate hash function (see comments at head of file). */
    .....

    if (flags & HASH_SHARED_MEM) { /* 共享内存中的哈希表，跳过 */ }
    else {
        /* setup hash table defaults */
        hashp->hctl = NULL; hashp->dir = NULL; hashp->isshared = false;
        hashp->hcxt = CurrentDynaHashCxt;
    }

    if (!hashp->hctl) { /* 在内存池中分配HASHHDR结构 */
        hashp->hctl = (HASHHDR *) hashp->alloc(sizeof(HASHHDR));
        if (!hashp->hctl) { /* 如果分配不成功过， 报错处理，略过 */ }
    }
}

```

```

hashp->frozen = false;
/* 为哈希表设置一些缺省值 */
hdefault(hashp);

hctl = hashp->hctl;
if (flags & HASH_SEGMENT) {
    hctl->ssize = info->ssize; hctl->sshift = my_log2(info->ssize);
}
if (flags & HASH_DIRSIZE) {
    hctl->max_dsize = info->max_dsize; hctl->dsize = info->dsize;
}
/* remember the entry sizes, too */
hctl->keysize = info->keysize; hctl->entrysize = info->entrysize;
/* make local copies of heavily-used constant fields */
hashp->keysize = hctl->keysize;
hashp->ssize = hctl->ssize; hashp->sshift = hctl->sshift;

/* Build the hash directory structure 初始化哈希表 */
if (!init_htab(hashp, nelem))
    elog(ERROR, "failed to initialize hash table \"%s\"", hashp->tabname);

if ((flags & HASH_SHARED_MEM) || nelem < hctl->nelem_alloc) {
    /* 这段代码跳过, 不影响整体理解 */
}
if (flags & HASH_FIXED_SIZE) hashp->isfixed = true;
return hashp;
}

```

上述代码中引用的初始化哈希表的函数 `hdefault()` 比较简单, 就是为 HTAB 设置一些缺省值, 读者可以自行分析。下面走读哈希表初始化函数 `init_htab()` 的代码。

```

/* in src/backend/utils/hash/dynahash.c */
/* 入口参数nelem是调用者打算往本哈希表中存放的元素的个数 */
static bool init_htab(HTAB *hashp, long nelem)
{
    HASHHDR     *hctl = hashp->hctl;
    HASHSEGMENT *segp;
    int          nbuckets, nsegs, i;

    /* 哈希表初始化时桶的总数目是nelem往上靠的2^n, 譬如nelem=17, 则nbuckets=32 */
    nbuckets = next_pow2_int(nelem);
    /* max_bucket先设置为总的桶数, 表示目前所有的桶都可以插入元素 */
    hctl->max_bucket = hctl->low_mask = nbuckets - 1;
    hctl->high_mask = (nbuckets << 1) - 1;
    /* 上述两行代码表示该哈希表的最大体积是nbuckets * 2 */
    nsegs = (nbuckets - 1) / hctl->ssize + 1; /* 确定实际需要多少个段 */
    nsegs = next_pow2_int(nsegs);
    if (nsegs > hctl->dsize) {
        if (!(hashp->dir)) hctl->dsize = nsegs;
        else return false;
    }
}

```

```

}

if (!(hashp->dir)) { /* 分配dir数组的内存 */
    CurrentDynaHashCxt = hashp->hcxt;
    hashp->dir = (HASHSEGMENT *)hashp->alloc(hctl->dsize * sizeof(HASHSEGMENT));
    if (!hashp->dir) return false;
}

for (segp = hashp->dir; hctl->nsegs < nsegs; hctl->nsegs++, segp++) {
    *segp = seg_alloc(hashp); /* 每次分配一个段，一共分配nsegs个段 */
if (*segp == NULL) return false;
}

/* 通过一个简单的算法确定一次性需要从内存池批发多少个空白元素 */
hctl->nelem_alloc = choose_nelem_alloc(hctl->entrysize);
return true;
}

```

根据上述的逻辑，该哈希表的最大体积 M 是 $nbuckets * 2$, $high_mask = M - 1$, $low_mask = M/2 - 1$ 。如果你现在依然糊涂，请再仔细学习上一节的理论内容，因为上述代码是对理论的严格实现。

除了创建和销毁哈希表，对哈希表最重要的操作还有三个：插入元素，查找元素和删除元素，源代码把这三个基本操作都集中在 `hash_search_with_hash_value()` 一个函数中，其入口参数 `action` 表示对哈希表实行的动作，有如下类型：

```

/* in src/include/utils/hsearch.h */

typedef enum {
    HASH_FIND,          /* look up key in table */
    HASH_ENTER,         /* look up key in table, creating entry if not present */
    HASH_REMOVE,        /* look up key in table, remove entry if present */
    HASH_ENTER_NULL    /* same, but return NULL if out of memory */
} HASHACTION;

```

其中动作类型 `HASH_FIND` 表示在哈希表中根据 K 进行查找，找到后就返回指向对应元素的指针，否则返回 `NULL`。动作类型 `HASH_ENTER` 会根据 K 先查找，如果找到了，就返回指向对应元素的指针，若找不到，则把要查找的 K 和 V 插入到哈希表中，此种情况下调用者要提前准备好 V 。往哈希表中插入一个 KV 对时可能因为内存不足导致无法插入，在这种情况下 `HASH_ENTER` 会报错。`HASH_ENTER_NULL` 做的动作和 `HASH_ENTER` 都一样，唯一的区别是当因为内存不足而导致插入不进去时，`HASH_ENTER_NULL` 简单地返回 `NULL`，并不报错。`HASH_REMOVE` 则是先查后删，若找到该元素就删除之。

随着不断往哈希表中插入元素，元素的个数超过了桶的个数，必定存在一个桶里有两块砖的情况，降低了哈希表的查找速度。这种情况下就要增加一个桶，确保装填因子 $a = 1$ 。下面的代码显示了选择扩容时机的逻辑。

```

/* in src/backend/utils/hash/dynahash.c:hash_search_with_hash_value() */

if (action == HASH_ENTER || action == HASH_ENTER_NULL) {
    if (hctl->freeList[0].nentries > (long) hctl->max_bucket &&
        !IS_PARTITIONED(hctl) && !hashp->frozen && !has_seq_scans(hashp))
        (void) expand_table(hashp);
}

```

由上面的代码可知，当打算往哈希表中插入一个元素时，就要检查一下是不是该扩容了。哈希表扩容的条件很简单，就是 $nentries > max_bucket$ 。检查条件还有其它逻辑判断：如果该哈希表有分区，表明它是在共享内存中，是不能扩容的。如果哈希表被冻结，或者在其上有别的进程在扫描，也是不能扩容的。如果有别的进程在扫描，也说明这个哈希表在共享内存中，在私有内存中的哈希表只有一个进程在使用。哈希表扩容的函数是 `expand_table()`，下面我们看看它的具体内容。

```

/* in src/backend/utils/hash/dynahash.c */
static bool expand_table(HTAB *hashp)
{
    HASHHDR    *hctl = hashp->hctl;
    HASHSEGMENT old_seg, new_seg;
    long        old_bucket, new_bucket, new_segnum, new_segndx, old_segnum, old_segndx;
    HASHBUCKET *oldlink, *newlink, currElement, nextElement;
    /* 要增加的新桶的桶号是max_bucket + 1 */
    new_bucket = hctl->max_bucket + 1;
    /* 根据新桶的桶号计算它所在的段和在该段中的编号 */
    new_segnum = new_bucket >> hashp->sshift;
    new_segndx = MOD(new_bucket, hashp->ssize);
    if (new_segnum >= hctl->nsegs) { /* 如果新桶所在的段超过了nsegs, 则要增加一个段 */
        if (new_segnum >= hctl->dsize)
            /* 如果新桶所在的段找过了当前dir数组的最大长度, 就要把哈希表的最大体积加倍 */
            if (!dir_realloc(hashp)) return false;
        /* 增加一个新段, 确保新桶有地方可存放 */
        if (!(hashp->dir[new_segnum] = seg_alloc(hashp))) return false;
        hctl->nsegs++; /* 调整本哈希表的段的总数 */
    }
    /* 把max_bucket往右移动一个位置, 增加一个新桶 */
    hctl->max_bucket++;
    /* 根据新桶的编号计算老桶的编号 */
    old_bucket = (new_bucket & hctl->low_mask);
    /* 因为high_mask的值是n世代的哈希表的最大体积M - 1, 所以new_bucket > high_mask表明总的桶数已经超过
     * 哈希表的最大体积了, 哈希表需要进入到下一个世代, high_mask和low_mask都要进行调整。
     */
    if ((uint32) new_bucket > hctl->high_mask) {
        hctl->low_mask = hctl->high_mask; hctl->high_mask = (uint32) new_bucket | hctl->low_mask;
    }
    old_segnum = old_bucket >> hashp->sshift; old_segndx = MOD(old_bucket, hashp->ssize);
    old_seg = hashp->dir[old_segnum]; new_seg = hashp->dir[new_segnum];
    oldlink = &old_seg[old_segndx]; newlink = &new_seg[new_segndx];
    /* 拆分老桶, 遍历其冲突链, 把里面的元素重新审查一遍, 有些转头可能要放在新桶里 */
    for (currElement = *oldlink; currElement != NULL; currElement = nextElement) {
        nextElement = currElement->link;
        if ((long) calc_bucket(hctl, currElement->hashvalue) == old_bucket) {
            *oldlink = currElement; oldlink = &currElement->link;
        } else { *newlink = currElement; newlink = &currElement->link; }
    }
    *oldlink = NULL; *newlink = NULL;
    return true;
}

```

我们结合一个例子, 对上述代码的逻辑理解的会更加清晰一些。假设创建哈希表时段的长度 $ssize = 256$, 指定要插入的元素的总数 $nelem=137$, 则哈希表创建完后的模样如图 6.30 所示, 此时 $max_bucket = 255$, $low_mask = 0xFF$, $high_mask = 0x1FF$, 哈希表的最大体积 $M = 512$ 。

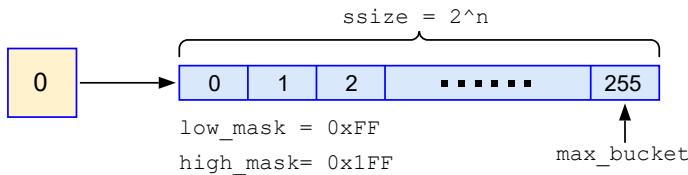


图 10.30: 哈希表刚创建时的情景

当要插入的元素个数超过 256，第一次扩容就发生了，扩容后的哈希表如图 6.31 所示。

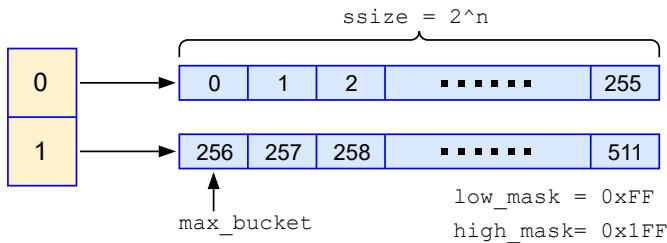


图 10.31: 哈希表的第一次扩容

第一次扩容后，新桶的桶号是 256，和其对应的老桶是 0 号， $B_{old} = B_{new} \& 0xFF$ 。增加新桶后，就要检查老桶里面的元素有没有必要被转移到新桶中来。检查的关键点是：

```
if ((long)calc_bucket(hctl, currElement->hashvalue)==old_bucket) { /*元素继续留在老桶里*/ }
else { /*元素被转移到了新桶里*/ }
```

这里的关键点是函数 `calc_bucket()`，它就是理论篇中的寻址函数 G_i ，其作用就是根据哈希值来确定该元素应该归属于哪个桶，完整的代码如下。

```
/* in src/backend/utils/hash/dynahash.c */
static inline uint32 calc_bucket(HASHHDR *hctl, uint32 hash_val)
{
    /* Convert a hash value to a bucket number */
    uint32 bucket = hash_val & hctl->high_mask;
    if (bucket > hctl->max_bucket) bucket = bucket & hctl->low_mask;
    return bucket;
}
```

老桶(0号桶)中元素的哈希值 $H \& 0x1FF$ 后有两种可能的值，0x000 或者 0x100。如果是 0x000 就依然归属于 0 号桶，如果是 0x100 就归属于 256 号桶，从而达到了把老桶中的元素搬一部分到新桶中的目的，请参考图 6.18 理解其含义。注意，这个算法并不确保老桶的元素一份为二，依然存在老桶的元素纹丝不动，而新桶里面空空如也的可能性。

随着不断的扩容，`max_bucket` 的编号不断增加，会从 256 逐步变成 511。当新桶是 257 时，老桶就是 1，当新桶是 511 时，老桶就是 255。在 `max_bucket` 从 256 变成 511 的过程中，`low_mask` 和 `high_mask` 并不会发生变化，因为这个时候哈希表还没有世代交替。但当 `max_bucket` 从 511 变成 512 时，就发生了图 6.32 所示的变化。

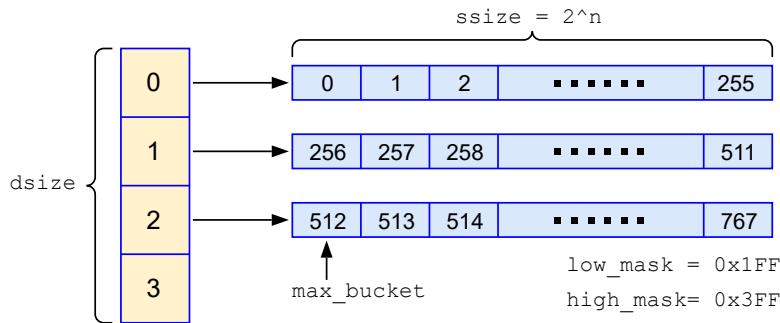


图 10.32: 哈希表进行了世代交替

因为 `high_mask=0x1FF`, 即十进制的 511, 现在 `max_bucket` 为 512, 就要调整 `low_mask` 和 `high_mask` 的值, 结果 `low_mask=0x1FF`, `high_mask=0x3FF`, 哈希表的最大体积 $M = 1024$ 。新桶 512 对应的老桶依然是 0 号桶, 513 号桶对应的老桶是 1 号桶, $B_{old} = B_{new} \& 0x1FF$ 。这种情况就是图 6 中的情景 (d) 所表示的内容。

学习到这里, 最核心的哈希查找函数 `hash_search_with_hash_value()` 的源代码应该非常容易理解, 下面我们走读一遍。

```
/* 入口参数中, keyPtr是要查找的键, hashvalue是其的哈希值, 输出参数foundPtr表示找到了没有 */
void * hash_search_with_hash_value(HTAB *hashp, const void *keyPtr, uint32 hashvalue,
                                    HASHACTION action, bool *foundPtr)
{
    HASHHDR      *hctl = hashp->hctl;
    Size          keysz;
    uint32        bucket;
    long          segment_num, segment_ndx;
    HASHSEGMENT   segp;
    HASHBUCKET   currBucket, *prevBucketPtr;
    HashCompareFunc match;

    if (action == HASH_ENTER || action == HASH_ENTER_NULL) { /* 扩容的问题已经分析过了 */ }

    /* 根据hashvalue, 计算要在哪一个桶里去寻找, 哈希查找速度快的秘密就在于此!!!! */
    bucket = calc_bucket(hctl, hashvalue);
    segment_num = bucket >> hashp->sshift; segment_ndx = MOD(bucket, hashp->ssize);
    segp = hashp->dir[segment_num];
    if (segp == NULL) hash_corrupted(hashp);

    /* 拿到这个桶的桶号后, 在它的冲突链中寻找键 */
    prevBucketPtr = &segp[segment_ndx];
    currBucket = *prevBucketPtr;
    match = hashp->match;           /* save one fetch in inner loop */
    keysz = hashp->keysz;          /* ditto */

    while (currBucket != NULL) {
        /* 哈希值H和键K都匹配上了, 找到了该元素! */
        if (currBucket->hashvalue == hashvalue &&
            match(ELEMENTKEY(currBucket), keyPtr, keysz) == 0)
            break;
        prevBucketPtr = &(currBucket->link);
        currBucket = *prevBucketPtr;
    }
}
```

```

}

/* 设置找到的标志，返回给调用者。currBucket != NULL表示找到，否则就是没有找到 */
if (foundPtr) *foundPtr = (bool) (currBucket != NULL);

switch (action) { /* 根据不同的动作，做不同的操作 */
    case HASH_FIND: /* 只找，不做额外动作，就直接把结果返回给调用者 */
        if (currBucket != NULL) return (void *) ELEMENTKEY(currBucket);
        return NULL;
    case HASH_REMOVE: /* 找到后要删除该元素 */
        if (currBucket != NULL) {
            /* 如果找到了，就把该元素从哈希表中摘下来，挂在freeList[0].freeList上 */
            hctl->freeList[0].nentries--; /* 哈希表中元素的总个数减一 */
            .....
            return (void *) ELEMENTKEY(currBucket);
        }
        return NULL;
    case HASH_ENTER_NULL:
    case HASH_ENTER:
        /* 如果找到了就直接把该元素的指针返回给调用者，否则就要把该元素插入到哈希表中 */
        if (currBucket != NULL) return (void *) ELEMENTKEY(currBucket);
        /* 如果该哈希表被冻结了，就插不进去了 */
        if (hashp->frozen) elog(ERROR, ".....");
        /* 从freeList[0].freeList的空闲元素的链表中找一个空闲的元素 */
        currBucket = get_hash_entry(hashp, freelist_idx);
        if (currBucket == NULL) {
            /* 内存不够了，报错 */
            if (action == HASH_ENTER_NULL) return NULL;
            /* report a generic message */
            .....
        }
        /* 把新元素插入到该桶的冲突链中 */
        .....
        return (void *) ELEMENTKEY(currBucket);
    }
    elog(ERROR, "unrecognized hash action code: %d", (int) action);
    return NULL; /* keep compiler quiet */
}

```

至此，我们对私有内存中的动态哈希表有了从理论到实践，自上而下的认识。从源代码中可以看到，整个动态哈希表的逻辑都是按照 Larson 教授的论文来实现的。关于在共享内存中的哈希表技术，后面有关的章节会进行讨论。动态哈希表在 PostgreSQL 中是一个基础性的数据和算法，被引用在很多组件里面，你只要在 PostgreSQL 的源代码目录下做下面的搜索，就容易看到它到底应用在哪些地方了。

```
$ find . -name \*.c | xargs grep hash_create | grep -v "backend/utils/hash/dynahash.c" | wc -l
107
```

理解了动态哈希表背后的思想和具体的代码实现，就很容易理解 PostgreSQL 其它源代码中如何使用该哈希算法的，这些不再赘述。

第十一章 锁和共享内存

PostgreSQL 数据库实例的核心部件之一就是共享内存，共享内存可以被多个后台进程所访问，为了保证共享内存中的数据不被多进程的并发执行破坏，必须使用锁来控制进程的访问顺序。本章研究 PostgreSQL 的锁、共享内存和进程间通讯方面的知识。

11.1 进程间通讯的基本知识

两个或更多个进程同时运行是内存被共享的前提，所以在探讨共享内存之前，不能不研究多进程和进程间通讯 (Inter-Process Communication, IPC) 的知识。本节并不打算对多进程和 IPC 的知识进行全面讲解，而是为服务我们理解 PostgreSQL 的源代码进行扼要的介绍。如果读者需要理解这方面知识的细节，可以参考两本非常好的书。一本书是《The Linux Programming Interface》，作者是 Michael Kerrisk。另外一门书是《Computer Systems : A Programmer's Perspective》，作者是 Randal E. Bryant 和 David R. O'Hallaron 教授。

11.1.1 系统调用 fork()

前文已经交代过，为了追求稳定性，PostgreSQL 的核心开发团队坚持使用进程，拒绝使用线程，仅在相对不重要的客户端工具如性能测试工具 pgbench 等处才使用线程。下面的研究只涉及进程，不讨论线程。在 Linux 等类 Unix 的操作系统中，一个进程创建新的进程，主要的方法是通过著名的系统调用 fork()，其定义如下：

```
#include <unistd.h>
pid_t fork(void);
```

从外表看，fork() 的定义非常简单，就是一个没有任何输入参数的普通函数。实际上它是一个重要的 Linux 系统调用，是创建进程的主要手段。图 9.1 展示了 fork() 的基本概念。

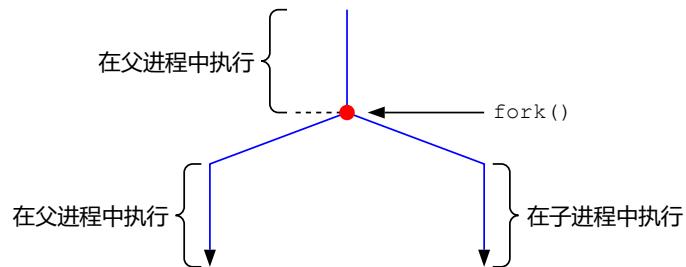


图 11.1: fork() 创建子进程的基本概念

图中的直线表示串行执行的指令流。一开始只有一个进程在执行单一的指令流，当它调用 fork() 后，就派生出一个子进程，老进程自动升格为父进程。父子进程在 fork() 执行完毕后开始分叉，各走各的路。该函数返回一个 pid_t 类型的值，其实就是一个整数。当返回值为 -1 时，表示创建子进程失败。当返回值为 0 时，表示后面的代码将在子进程中运行。当返回值大于 0，则表示后面的代码依然在父进程中运行。所以 fork() 的使用往往和 if 语句结合在一起，基本形式是这样的：

```
pid_t result;
/* 此时此刻依然在父进程中 */
result = fork(); /* <-- 该函数执行完毕后，就兵分两路了 */
if(-1 == result) { /* 若创建子进程失败则做一些异常处理工作 */}
```

```

/* do something; */

} else if(0 == result) { /* 现在处于子进程中 */

    .....

} else { /* 现在依然在父进程中, result是子进程的进程号 */

    .....

}

```

下面是 PostgreSQL 中 postmaster 主进程创建子进程的关键代码片段：

```

/* in src/backend/postmaster/fork_process.c */
pid_t fork_process(void)
{
    pid_t          result;
    .....
    result = fork();
    if (result == 0) {
        /* fork succeeded, in child */
        .....
    }
    else {
        .....
        /* in parent, restore signal mask */
    }
    return result;
}

```

函数 fork_process() 就是对 fork() 的封装，其套路和上面的示例程序几乎一模一样。理解了 fork() 的基本概念之后，下面开始研究共享内存。

11.1.2 共享内存

我们都知道，小至手机，大到超级服务器，计算机的基本组成都是一样的，都是冯诺依曼结构，整个计算机由 CPU、内存和输入输出等部件组成。图 9.2 展示了 CPU 和内存交互的基本概念。CPU 在地址总线上发出一个地址信息，要求读取从 2 号内存单元开始的连续 4 个字节，则内存会把 2、3、4、5 共计 4 个内存单元里面的内容放在数据总线上，CPU 就可以把这 4 个字节的内容读入到内部的寄存器中。CPU 在地址总线上发出的地址信息被称为物理地址 (PA : physical address)。

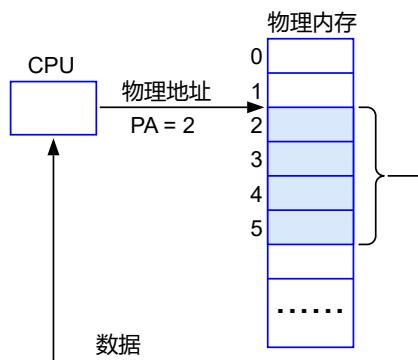


图 11.2: 物理地址

如果我们进一步研究 CPU 芯片内部的结构，就会发现其中有个叫做“内存管理单元”(MMU: memory management unit)的部件。MMU 的任务是把“虚拟地址”(VA: virtual address)转化成物理地址 PA。这里面就牵扯到了虚拟地址 VA 的概念。在 64 位操作系统中，每一个程序理论上可以访问的地址是从 0 到 0xFFFFFFFFFFFFFFFFF，和 WAL 空间是一样大。这个地址空间就被称为 VA。把 VA 和 PA 分开，方便了软件开发，因为每台计算机上的物理内存可能大小不同，应用程序的开发人员在开发程序时并不操心这些差异，在他的头脑中有一个统一的一维的线性空间可供自己使用，这个模型和真实的物理机器没有关系。开发人员就是围绕着这个抽象而统一的模型来开发各种应用程序的。开发的程序被编译成可执行文件，就是我们在 \$PGHOME/bin 目录下看到的各种程序。在这些程序里面使用的地址都是虚拟地址。等程序在某个真实的计算机上运行时，就存在把程序里的虚拟地址 VA 转换成这台计算机的物理地址 PA 的过程，这个转换工作不需要程序员操心，它是由 CPU 硬件和操作系统来完成的，开发操作系统内核的程序员才会关心这个问题。虚拟地址到物理地址的转换可以用图 9.3 来表示：

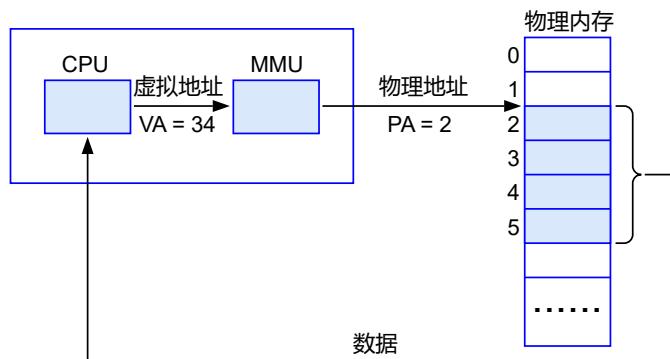


图 11.3: 虚拟地址到物理地址的转换

在 64 位操作系统上，虚拟地址是 8 字节，而且在进程的这个虚拟地址空间里，进程各部分的布局是固定的，如图 9.4 所示。

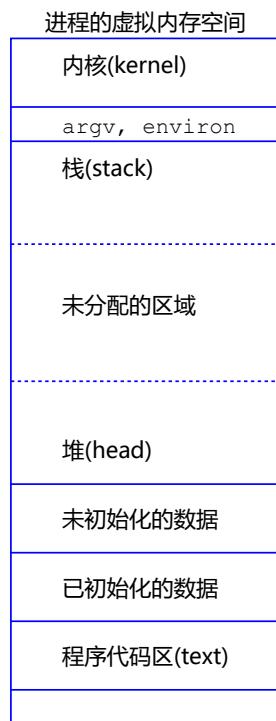


图 11.4: 进程的虚拟内存空间的布局

下面我们讨论的私有内存和共享内存的分配，都是在堆这一部分进行分配的。

11.1.2.1 私有内存和共享内存

我们先演示两个简单的 C 程序，对比多进程情况下私有内存和共享内存的差异。在第一个例子中，父进程通过 malloc() 创建一块私有内存，继而通过 fork() 创建子进程，我们研究子进程和父进程创建的私有内存的关系，源代码如下：

```
$ cat m1.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define MEMORY_SIZE 8192
char* gmem = NULL;
int main(int argc, char* argv[])
{
pid_t pid;
int result = 0;
gmem = (char*)malloc(MEMORY_SIZE);
if(NULL == gmem) { result = 1; goto myquit; }
gmem[0] = gmem[1] = 'A'; gmem[2] = 0x00;

pid = fork(); /* 关键点尽在此处!!! */

if(pid < 0) { result = 1; goto myquit; }
if(0 == pid) { /* fork()函数执行成功了，下面的代码在子进程中执行 */
    printf("(Child): I will sleep for a while.\n");
    sleep(10); /* sleep for 10 seconds */
    printf("(Child): gmem is [%s]\n", gmem);
    goto myquit;
} else { /* fork()函数执行成功了，下面的代码在父进程中执行 */
    gmem[0] = gmem[1] = 'B'; gmem[2] = 0x00;
    sleep(20); /* sleep for 20 seconds */
    printf("(Parent): gmem is [%s]\n", gmem);
}
myquit:
if(NULL != gmem) free(gmem);
return result;
}
```

上述源代码不难理解。父进程首先通过 malloc() 函数申请一块 8192 字节的内存块，保存在全局变量 gmem 中，然后往该内存块写入两个大写的 A。紧接着父进程通过 fork() 创建子进程。子进程诞生后，立刻休眠 10 秒钟，醒来后开始检查 gmem 指向的内存中的值。父进程在 fork() 成功以后，再次修改了 gmem 指向的内存的内容，写入两个大写的 B。然后休眠 20 秒钟后再次检查内存块里内容。下面是该程序运行的结果：

```
$ gcc -Wall m1.c -o m1
$ ./m1
(Child): I will sleep for a while.
(Child): gmem is [AA]
(Parent): gmem is [BB]
```

上述结果显示，父进程看到的是 BB，而子进程看到的依然是原始值 AA。通过这个例子，我们可以得出如下结论：

- 系统调用 malloc() 申请的内存是本进程私有的内存，只能够被本进程所读写。
- 父进程 fork() 出子进程后，子进程会自动继承父进程 malloc() 所申请的内存，父子进程各一份，互不冲突。

现在研究另一个例子：父进程创建一块共享内存，然后 fork() 出子进程。由此来观察父子进程读写这块共享内存的基本规律，源代码如下：

```
$ cat m2.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#ifndef MAP_ANONYMOUS
#define MAP_ANONYMOUS      MAP_ANON
#endif
#ifndef MAP_HASSEMAPHORE
#define MAP_HASSEMAPHORE    0
#endif
#define PG_MMAP_FLAGS  (MAP_SHARED|MAP_ANONYMOUS|MAP_HASSEMAPHORE)
#define MEMORY_SIZE     8192
void* gmem = MAP_FAILED;
int main(int argc, char* argv[])
{
pid_t pid;
int result = 0;
char *p;
gmem=mmap(NULL,MEMORY_SIZE,PROT_READ|PROT_WRITE,PG_MMAP_FLAGS,-1,0);
if(MAP_FAILED == gmem) { result = 1; goto myquit; }
p = (char*)gmem; p[0] = p[1] = 'A'; p[2] = 0x00;

pid = fork(); /* <--The key point!!! */

if(pid < 0) { result = 1; goto myquit; }
if(0 == pid) { /* fork succeeded, in child process */
sleep(10);
p = (char*)gmem; p[0] = p[1] = 'B'; p[2] = 0x00;
exit(0);
} else { /* fork succeeded, in parent process */
printf("(Parent) - before sleep: [%s]\n", p);
sleep(20);
printf("(Parent) - after sleep: [%s]\n", p);
}
myquit:
if(MAP_FAILED != gmem) munmap(gmem, MEMORY_SIZE);
return result;
}
```

编译运行上述的 m2.c 程序，有如下结果：

```
$ gcc -Wall m2.c -o m2
$ ./m2
(Parent) - before sleep: [AA]
(Parent) - after sleep: [BB]
```

由上述实验的结果可知，父进程在休眠前从共享内存中读到的值是 AA，睡了一觉，醒来后发现同一块共享内存里的值变成了 BB。很显然，在父进程休眠期间，子进程修改了共享内存里面的内容。通过这个例子，我们可以得出如下结论：

- 父进程通过 mmap() 创建的匿名共享内存，可以自动地子进程继承并共享。
- 如果没有保护机制，共享内存里的内容很容易被破坏。

通过 m1.c 和 m2.c 这两个例子的对比，我们很容易理解 malloc() 和 mmap() 创建的内存的不同，父进程创建的私有内存，子进程自动继承，父子一人一份。父进程创建的共享内存，子进程可以自动“贴”(attach)上去，但共享内存只有一份，父进程可允许子进程对其读写。这些知识是理解 PostgreSQL 中锁和共享内存的基础。

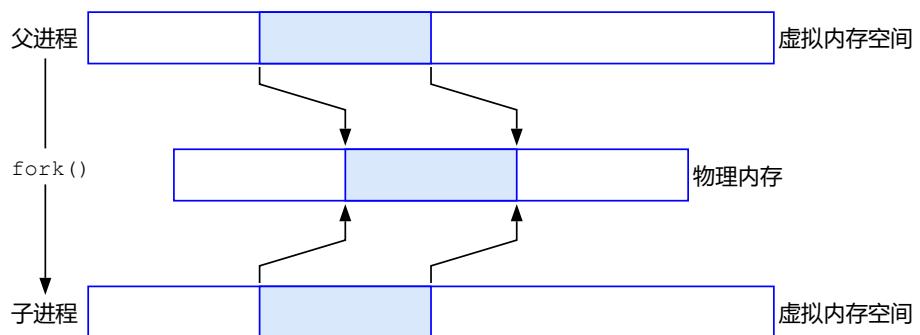


图 11.5: 进程虚拟内存和物理内存的映射

11.1.3 Latch

Latch

11.1.4 信号

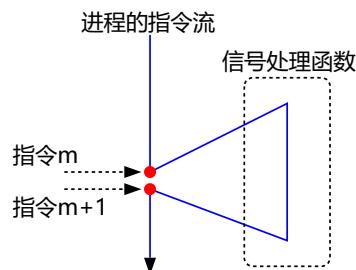


图 11.6: 处理信号的流程

```
/* in src/include/port.h */
typedef void (*pqsigfunc) (int signo);
/* in src/port/pqsignal.c */
pqsigfunc pqsignal(int signo, pqsigfunc func)
```

```
{
    struct sigaction act, oact;
    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_RESTART;
#ifndef SA_NOCLDSTOP
    if (signo == SIGCHLD)
        act.sa_flags |= SA_NOCLDSTOP;
#endif
    if (sigaction(signo, &act, &oact) < 0) return SIG_ERR;
    return oact.sa_handler;
}
```

当两个进程进行通讯时，第一步往共享内存存放一些信息，然后使用 kill() 函数向对方发信号。对方的信号处理函数检查共享内存中的相关数据，就可以获知更多的信息了。

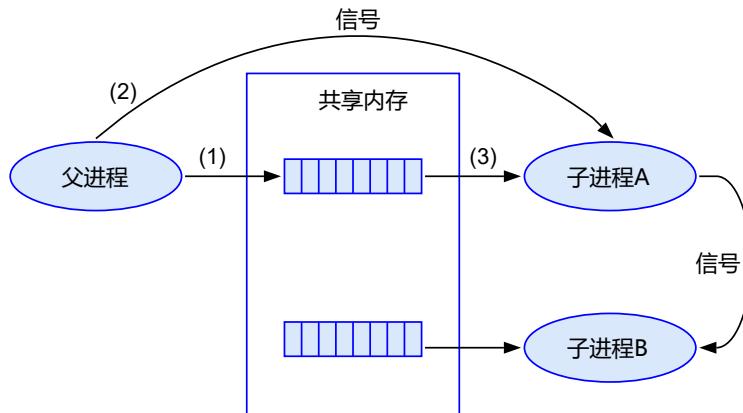


图 11.7: 进程间通讯

11.1.5 信号量

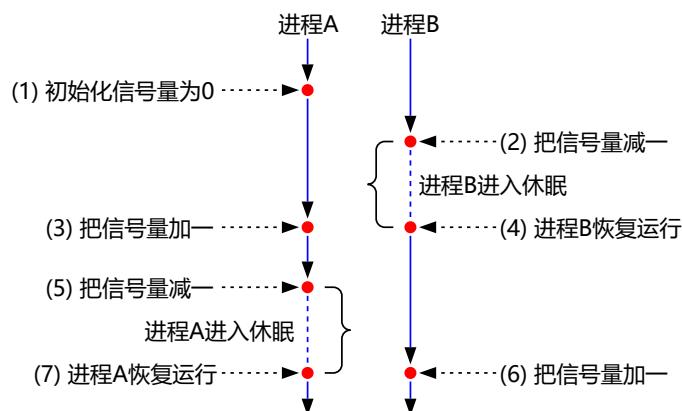


图 11.8: 信号量同步进程的执行

```
/* in /usr/include/bits/semaphore.h */
typedef union {
```

```
char __size[32];
long int __align;
} sem_t;
/* in src/backend/port posix_sema.c */
#define PG_CACHE_LINE_SIZE    128
typedef union SemTPadded {
    sem_t pgsem;
    char pad[PG_CACHE_LINE_SIZE];
} SemTPadded;
typedef struct PGSemaphoreData {
    SemTPadded sem_padded;
} PGSemaphoreData;
typedef struct PGSemaphoreData *PGSemaphore;

#define PG_SEM_REF(x)  (&(x)->sem_padded.pgsem)
void PGSemaphoreLock(PGSemaphore sema)
{
    int errStatus;
    do {
        errStatus = sem_wait(PG_SEM_REF(sema));
    } while (errStatus < 0 && errno == EINTR);
    if (errStatus < 0) elog(FATAL, "sem_wait failed: %m");
}
void PGSemaphoreUnlock(PGSemaphore sema)
{
    int errStatus;
    do {
        errStatus = sem_post(PG_SEM_REF(sema));
    } while (errStatus < 0 && errno == EINTR);
    if (errStatus < 0) elog(FATAL, "sem_post failed: %m");
}
```

11.2 锁

11.2.1 自旋锁

由于很多进程都可以访问同一块共享内存，且它们的访问时间和顺序是不可预知的，若不加保护，就可能出现这种情况：某进程刚刚往内存单元 x 中存入 123，再次读取 x 中的内容，结果发现它变成了 321。在此种情况下，我们认为内存单元 x 中的内容被破坏了。为了避免这种不可预知性，采用多进程架构的软件必须使用“锁”机制使多个进程对同一个内存单元的访问过程“串行化”：在任何时刻，只有一个进程修改某块共享内存。一个进程结束对某块共享内存的写操作之前，不允许别的进程修改这块内存的内容。锁的概念对初学者非常神秘，其实它就是共享内存，一把锁往往只有一个字节，它可以保护一块共享内存。图 7.2 展示了锁和被保护的共享内存之间的关系。

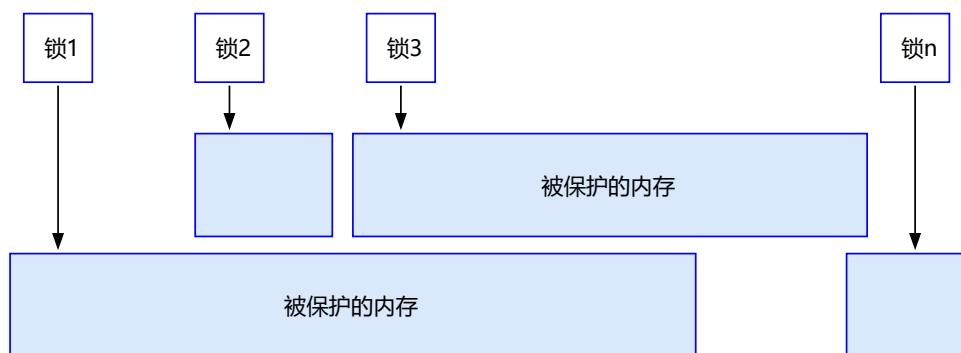


图 11.9: 锁和共享内存

假设有 n 把锁，你可以理解为它们是共享内存中的 n 个字节，每一把锁都可以保护一块共享内存。在 PostgreSQL 中有自旋锁，轻量级锁等不同类型的锁，本节介绍自旋锁 (spin lock)。自旋，就是自我旋转的意思。我们小时候可能都玩过陀螺，用鞭子抽打的陀螺会不停地旋转，这种状态就叫“自旋”。自旋锁的名字非常形象，所蕴含的思想也和旋转的陀螺类似，图 7.3 展示了自旋锁的基本概念。

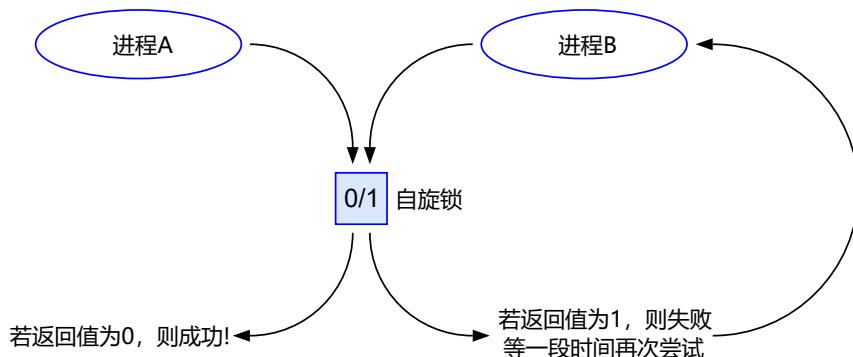


图 11.10: 自旋锁的基本原理

图中的正方形代表一把自旋锁，它实际上就是共享内存中的一个字节。如果该字节的值为 0，表示没有进程用这把锁，如果其值为 1 或者非零值，表示有进程正在使用这把锁。进程 A 想修改该自旋锁保护的共享内存，会首先检查该锁的状态，如果其值为 0，就把它设置为 1，然后就可以放心地修改共享内存了。进程 B 亦想修改该自旋锁保护的共享内存，也会检查该锁的状态，结果发现其值为 1，进程 B 就不停地循环，反复检查该锁的状态，直至进程 A 释放该锁，即它的值变成了 0，进程 B 才获得了该锁。进程 B 反复循环检测锁的状态，就是“自旋”，这是自旋锁名字的来由。下面的 C 代码来描述自旋锁的工作逻辑：

```

char smem[N] = some_value; /* 假设smem数组是某块共享内存 */
char* lockX = some_value; /* 自旋锁lockX亦在共享内存中 */
/* get_spin_lock()是获得自旋锁的函数 */
void get_spin_lock(char* lock)
{
    spin_again:
        if(0 == *lock) { /* 若lock值为0, 则无人使用该锁 */
            *lock = 1; /* 加锁! */
            return; /* 返回! */
        }
        goto spin_again; /* 否则就不停地循环检测 */
}
/* release_spin_lock()是释放自旋锁的函数 */
void release_spin_lock(char* lock) { if(NULL != lock) (*lock) = 0; }
/* 下面是使用自旋锁的套路 */
get_spin_lock(lockX);
..... /* 现在可以放心地修改smem[0]到smem[N-1]的内容了 */
free_spin_lock(lockX);

```

上述代码非常浅显易懂，却存在一个致命的问题：在 `get_spin_lock()` 函数中，第 7 行的语句 (`if(0 == *lock)`) 检查自旋锁 `lock` 的状态，是一条测试指令 (TEST)，第 8 行的语句 (`*lock = 1`) 设置自旋锁的值为 1，是一条赋值指令 (SET)。这是两条独立的指令，不具备原子性，即 TEST 指令和 SET 指令之间，可能被别的进程中断。也就是说刚刚检测到自旋锁的状态为 0，打算自己占这个坑时，可能别的进程捷足先登，在 TEST 和 SET 指令之间执行了一条 SET 指令，把 `lock` 占住了，而本进程却毫无察觉。为了保证可靠地获取自旋锁，必须保证 TEST 指令和 SET 指令之间不能被别人横插一脚。

众所周知，一条机器指令的执行过程是不能够被中断的。如果是多核 CPU 或多个 CPU 的情况，也可以使用 LOCK 前缀独占指令总线和数据总线，从而保证一条指令的原子性。因此我们需要硬件提供一条同时执行测试和赋值的指令，这是自旋锁可靠性的根本保证。这样的指令被称为 TAS(Test and Set) 指令。有的硬件不支持 TAS 指令，PostgreSQL 就用信号量来进行模拟，但性能非常慢。X86/X64 和 ARM 等主流 CPU 在硬件上均支持 TAS 指令，下面我们研究 X64 平台上的 TAS 指令。

11.2.1.1 X86/X64 中的 TAS 指令

X64 CPU 使用 XCHG 等指令来实现 TAS，该指令很简单，图 7.4 展示了其基本功能：

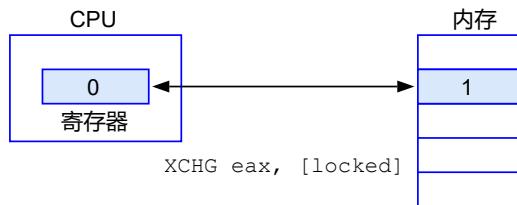


图 11.11：指令 XCHG 的逻辑

EAX 是 X64 CPU 中的一个寄存器，locked 是一个内存地址，`XCHG eax, [locked]` 这条指令的功能是把 EAX 寄存器中的内容和 locked 指向的内存单元中的内容互换。如果 EAX 的内容为 1，locked 内存单元的内容为 0，该指令执行完毕后，EAX 的内容变成 0，而 locked 内存单元的内容变成 1。由此看来，XCHG 一条指令就获得了 locked 内存单元的当前值 (TEST)，且把 locked 内存单元的内容变成了 1(SET)，这就实现了原子性的 TAS 功能。

下面的代码展示了用 X64 汇编语言编写的获取和释放自旋锁的基本实现。可能读者不懂汇编，无需害怕，你不用过多关注指令本身的细节，把注意力放在每条指令后面的注释上，就能搞明白代码的逻辑。

```

locked:           /* locked是一个内存单元, 初始值为0 */
    dd      0          /* 0表示没有上锁, 1表示已经上锁 */
;

;-----[spin_lock]-----[spin_unlock]-----;
spin_lock:        /* 获取自旋锁的代码的开始部分 */
    mov    eax, 1      /* 往寄存器EAX中放入1 */
    xchg  eax, [locked] /* XCHG指令为最核心的部分, 把EAX和locked的值互换! */
    test   eax, eax    /* EAX的值0否? 其值即locked先前的值, 此时locked已被置为1 */
    jnz    spin_lock   /* 若EAX的值为1, 表明locked正被别的进程占用 */
                    /* 若EAX的值为1, 则跳转至spin_lock处, 继续尝试获取 */
    ret     /* 若EAX的值为0, 则该锁无人占用, 吾已置其为1, 返回! */
                    /* 此时locked为1, 别的进程试图获取该锁时, 必陷入循环中 */
;

spin_unlock:     /* 释放自旋锁的代码的开始部分 */
    xor    eax, eax    /* 把EAX寄存器的值置为0 */
    xchg  eax, [locked] /* 把EAX中的内容存入locked中, locked的值为0 */
    ret     /* 返回。别的进程试图获取该锁时就可轻松拿到 */
;
```

11.2.1.2 PostgreSQL 中的自旋锁

下面的源代码是 PostgreSQL 中对 SpinLock 的具体实现。

```

/* in src/include/storage/spin.h */
#define SpinLockInit(lock)      S_INIT_LOCK(lock)
#define SpinLockAcquire(lock)    S_LOCK(lock)
#define SpinLockRelease(lock)    S_UNLOCK(lock)
#define SpinLockFree(lock)       S_LOCK_FREE(lock)
;
```

上述操作自旋锁的宏的入口参数 lock 是指向 slock_t 类型的指针。在 X64 平台上，slock_t 实际上是字符类型 (unsigned char)。这些宏的作用如下：

- SpinLockInit(lock) - 初始化自旋锁。其实就是把 lock 的值变成 0，表示现在没有加锁。该函数没有返回值。
- SpinLockAcquire(lock) - 获取自旋锁。此为核心操作。当别的进程已获得自旋锁 lock 时，调用该操作，就会陷入循环等待中。
- SpinLockRelease(lock) - 释放自旋锁，就是把 lock 的值置为 0。
- SpinLockFree(lock) - 测试自旋锁 lock 的状态。如果 lock 为 0，则表示该锁处于未加锁状态，该宏返回 true。如果 lock 不为 0 则表示该自旋锁已经被别的进程加锁了，该宏返回 false。我们常常在试图获取某个自旋锁之前用这个宏测试该自旋锁是否可能获得，降低了冲突的概率。

底层 X64 硬件的实现代码如下，其中 volatile 关键字告诉编译器，产生最终的机器指令时，volatile 修饰的变量要放在内存里，不能被缓存在 CPU 的寄存器里。

```

/* in src/include/storage/s_lock.h */
#define S_INIT_LOCK(lock)      S_UNLOCK(lock)
#define S_LOCK_FREE(lock)      (*(lock) == 0)
#define S_LOCK(lock)  (TAS(lock) ? s_lock((lock), __FILE__, __LINE__, PG_FUNCNAME_MACRO) : 0)
#define S_UNLOCK(lock) \
do { __asm__ __volatile__("") : : : "memory"; *(lock) = 0; } while (0)
;
```

由上可知，S_UNLOCK(lock) 比较简单，直接往 lock 内存单元中存入 0 就完事了。最核心的函数是加锁操作函数 S_LOCK(lock)。根据 S_LOCK(lock) 的定义可知：它首先调用 TAS(lock)，如果 TAS(lock) 返回 0，则表明此时没有别的进程使用该锁，TAS 会把 lock 里的内容变成 1，表明既然大家目前都不使用该锁，那本人就不客气了，直接上锁走人，此时 S_LOCK(lock) 立刻返回 0。如果 TAS(lock) 返回非 0，则表明无法获得这把锁，就开始调用 slock() 函数。下面是 TAS(lock) 的源代码：

```
/* in src/include/storage/s_lock.h */
#ifndef __x86_64__           /* AMD Opteron, Intel EM64T */
/* __x86_64__ 是编译器预定义的宏，表明产生的代码是X64硬件平台的 */
typedef unsigned char slock_t; /* 此即自旋锁的真实面目！*/
#define TAS(lock)      tas(lock)
#define TAS_SPIN(lock) (*lock) ? 1 : TAS(lock) /* 先测试锁状态，再试图获取锁 */
static __inline__ int tas(volatile slock_t *lock)
{
    register slock_t _res = 1; /* 让编译器把_res分配到某个寄存器中 */
    __asm__ __volatile__(

        " lock          \n" /* lock是前缀，表示独占数据总线 */
        " xchgb %0,%1 \n" /* 此为最关键处，原子性完成_res和内存lock的内容交换 */
        : "+q"(_res), "+m"(*lock)
        : /* no inputs */
        : "memory", "cc");

    return (int) _res; /* 返回寄存器中的值，此值即lock内存之前的值 */
}
#endif /* __x86_64__ */
```

下面是 s_lock() 的源代码，其中最主要的就是 while 循环，反复测试测试自旋锁 lock 的状态。在等待的时候使用 NOP 空指令进行空转。虽然该函数有返回值，但是实际使用自旋锁时，我们并不关心其返回值。

```
/* in src/backend/storage/lmgr/s_lock.c */
int s_lock(volatile slock_t *lock, const char *file, int line, const char *func)
{
    .....
    while (TAS_SPIN(lock)) { /* 此处while循环即是自旋 */
        perform_spin_delay(&delayStatus);
    }
    .....
}

void perform_spin_delay(SpinDelayStatus *status) { SPIN_DELAY(); ..... }

/* in src/include/storage/s_lock.h */
#define SPIN_DELAY() spin_delay()
static __inline__ void spin_delay(void)
{
    __asm__ __volatile__(" rep; nop             \n");
}
```

自旋锁的使用非常简单，只有一个模式，可以参考下面的源代码：

```
/* in src/backend/storage/lmgr/proc.c */
/* 在共享内存中分配一把自旋锁ProcStructLock */
slock_t *ProcStructLock = (slock_t *)ShmemAlloc(sizeof(slock_t));
SpinLockInit(ProcStructLock); /* 初始化该自旋锁 */
```

```
.....
SpinLockAcquire(ProcStructLock);
..... /* 在这里修改被ProcStructLock保护的共享内存 */
SpinLockRelease(ProcStructLock);
```

理解了自旋锁的基本思想和具体实现细节后，我们很容易掌握它的一些基本特性：其一是它是最轻量级的锁，核心只有一条 XCHG 的指令。其二是获取锁需要循环等待，为了避免这种情况频繁发生，自旋锁只适用于对共享内存进行极短时间的访问，通常就是十几条或几十条机器指令能够完成的对共享内存的修改操作。其三是自旋锁只有独占模式 (exclusive)，非我即他，且没有死锁控制机制，需要使用者注意。

11.2.2 轻量级锁

```
pg_atomic_init_u32
pg_atomic_read_u32
pg_atomic_compare_exchange_u32
pg_atomic_fetch_or_u32
pg_atomic_fetch_and_u32
pg_atomic_fetch_add_u32
pg_atomic_fetch_sub_u32
pg_atomic_sub_fetch_u32

XADD m, r          ; t = r; r = m; m = r + t;
CMPXCHG m, r      ; IF (eax == m) m = r ELSE eax = m
```

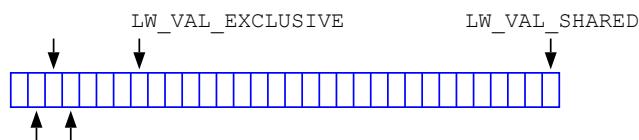


图 11.12: XXXXX

```
/* in src/include/storage/proc.h */
struct PGPROC
{
    /* proc->links MUST BE FIRST IN STRUCT (see ProcSleep, ProcWakeup, etc) */
    SHM_QUEUE    links;           /* list link if process is in a list */
    PGPROC       **procgloballist; /* procglobal list that owns this PGPROC */

    PGSemaphore sem;             /* ONE semaphore to sleep on */
    ProcWaitStatus waitStatus;

    Latch        procLatch;      /* generic latch for process */

    TransactionId xid;          /* id of top-level transaction currently being
                                   * executed by this proc, if running and XID
                                   * is assigned; else InvalidTransactionId.
```

```

        * mirrored in ProcGlobal->xids[pgxactoff] */

TransactionId xmin;           /* minimal running XID as it was when we were
                               * starting our xact, excluding LAZY VACUUM:
                               * vacuum must not remove tuples deleted by
                               * xid >= xmin ! */

LocalTransactionId lxid;      /* local id of top-level transaction currently
                               * being executed by this proc, if running;
                               * else InvalidLocalTransactionId */

int          pid;             /* Backend's process ID; 0 if prepared xact */

int          pgxactoff;       /* offset into various ProcGlobal->arrays with
                               * data mirrored from this PGPROC */

int          pgprocno;

/* These fields are zero while a backend is still starting up: */
BackendId   backendId;       /* This backend's backend ID (if assigned) */

Oid          databaseId;     /* OID of database this backend is using */

Oid          roleId;          /* OID of role using this backend */

Oid          tempNamespaceId; /* OID of temp schema this backend is
                               * using */

bool         isBackgroundWorker; /* true if background worker. */

/*
 * While in hot standby mode, shows that a conflict signal has been sent
 * for the current transaction. Set/cleared while holding ProcArrayLock,
 * though not required. Accessed without lock, if needed.
 */
bool         recoveryConflictPending;

/* Info about LWLock the process is currently waiting for, if any. */
bool         lwWaiting;       /* true if waiting for an LW lock */
uint8        lwWaitMode;      /* lwlock mode being waited for */
proclist_node lwWaitLink;    /* position in LW lock wait list */

/* Support for condition variables. */
proclist_node cvWaitLink;    /* position in CV wait list */

/* Info about lock the process is currently waiting for, if any. */
/* waitLock and waitProcLock are NULL if not currently waiting. */
LOCK         *waitLock;        /* Lock object we're sleeping on ... */
PROCLOCK    *waitProcLock;    /* Per-holder info for awaited lock */
LOCKMODE    waitLockMode;     /* type of lock we're waiting for */
LOCKMASK    heldLocks;        /* bitmask for lock types already held on this
                               * lock object by this backend */

pg_atomic_uint64 waitStart;  /* time at which wait for lock acquisition

```

```

        * started */

int          delayChkptFlags;      /* for DELAY_CHKPT_* flags */

uint8        statusFlags;         /* this backend's status flags, see PROC_*
                                 * above. mirrored in
                                 * ProcGlobal->statusFlags[pgxactoff] */

/*
 * Info to allow us to wait for synchronous replication, if needed.
 * waitLSN is InvalidXLogRecPtr if not waiting; set only by user backend.
 * syncRepState must not be touched except by owning process or WALSender.
 * syncRepLinks used only while holding SyncRepLock.
 */
XLogRecPtr   waitLSN;           /* waiting for this LSN or higher */
int          syncRepState;       /* wait state for sync rep */
SHM_QUEUE    syncRepLinks;      /* list link if process is in syncrep queue */

/*
 * All PROCLOCK objects for locks held or awaited by this backend are
 * linked into one of these lists, according to the partition number of
 * their lock.
 */
SHM_QUEUE    myProcLocks[NUM_LOCK_PARTITIONS];

XidCacheStatus subxidStatus;    /* mirrored with
                                 * ProcGlobal->subxidStates[i] */
struct XidCache subxids;        /* cache for subtransaction XIDs */

/* Support for group XID clearing. */
/* true, if member of ProcArray group waiting for XID clear */
bool         procArrayGroupMember;
/* next ProcArray group member waiting for XID clear */
pg_atomic_uint32 procArrayGroupNext;

/*
 * latest transaction id among the transaction's main XID and
 * subtransactions
 */
TransactionId procArrayGroupMemberXid;

uint32       wait_event_info;    /* proc's wait information */

/* Support for group transaction status update. */
bool         clogGroupMember;    /* true, if member of clog group */
pg_atomic_uint32 clogGroupNext; /* next clog group member */
TransactionId clogGroupMemberXid; /* transaction id of clog group member */
KidStatus    clogGroupMemberXidStatus; /* transaction status of clog
                                         * group member */

```

```

int          clogGroupMemberPage;    /* clog page corresponding to
                                         * transaction id of clog group member */
XLogRecPtr   clogGroupMemberLsn; /* WAL location of commit record for clog
                                         * group member */

/* Lock manager data, recording fast-path locks taken by this backend. */
LWLock       fpInfoLock;        /* protects per-backend fast-path state */
uint64       fpLockBits;        /* lock modes held for each fast-path slot */
Oid          fpRelId[FP_LOCK_SLOTS_PER_BACKEND]; /* slots for rel oids */
bool         fpVXIDLock;        /* are we holding a fast-path VXID lock? */
LocalTransactionId fpLocalTransactionId; /* lxid for fast-path VXID
                                         * lock */

/*
 * Support for lock groups.  Use LockHashPartitionLockByProc on the group
 * leader to get the LWLock protecting these fields.
 */
PGPROC      *lockGroupLeader;    /* lock group leader, if I'm a member */
dlist_head   lockGroupMembers;   /* list of members, if I'm a leader */
dlist_node   lockGroupLink;     /* my member link, if I'm a member */
};

/* in src/include/port/atomics/arch-x86.h */
typedef struct pg_atomic_uint32 { volatile uint32 value; } pg_atomic_uint32;
/* in src/include/storage/proclist_types.h */
typedef struct proclist_head {
    int head; /* pgprocno of the head PGPROC */
    int tail; /* pgprocno of the tail PGPROC */
} proclist_head;

/* in src/include/storage/lwlock.h */
typedef struct LWLock {
    uint16      tranche; /* tranche ID */
    pg_atomic_uint32 state; /* state of exclusive/nonexclusive lockers */
    proclist_head waiters; /* list of waiting PGPROCs */
} LWLock;

typedef enum LWLockMode {
    LW_EXCLUSIVE,
    LW_SHARED,
    LW_WAIT_UNTIL_FREE
} LWLockMode;

/* in src/backend/storage/lmgr/lwlock.c */

```

11.3 PostgreSQL 共享内存的结构

主进程在启动伊始，会创建一个固定尺寸的共享内存。你可以把它理解为一个巨大的数组，主进程和其创建的子进程均可访问该数组。在这块共享内存中有三种数据结构：固定大小的结构、链式队列和哈希表。固定大小的结构包含了各种模块的全局变量，在共享内存初始化阶段时就完成了其的分配和初始化。在共享内存中的哈希表的最大尺寸是固定的，不能够动态增长，但其实际尺寸会随着插入的元素的增加会分配更多的空间。链式队列可能会在固定大小的结构内分配，或者其成员为哈希表的元素。每个共享数据结构都有一个字符串名称以标识它。

在类 Unix 的操作系统中，父进程创建的子进程会自动继承父进程的虚拟内存布局和全局变量，所以主进程派生的后台和后端进程自动拥有对共享内存的访问能力，不需要做额外的工作。

共享内存一旦分配就无法释放。每个哈希表都有自己的空闲元素列表，所以当删除某个元素时，该元素的内存可以被重复使用。但是某一个哈希表变大之后再收缩，其空间无法分配给其它模块使用。在目前的架构下，这种情况几乎不存在，所以 PostgreSQL 并没有设计哈希表的垃圾搜集器 (garbage collection, GC)。

11.3.1 共享内存的分配机制

主进程在启动伊始，会调用函数 `CreateSharedMemoryAndSemaphores()` 创建共享内存和信号量，该函数的逻辑非常容易理解，首先是通过函数 `CalculateShmemSize()` 计算出共享内存的大小，然后调用函数 `PGSharedMemoryCreate()` 创建共享内存。共享内存创建成功后，`CreateSharedMemoryAndSemaphores()` 会进行一系列的共享内存初始化的工作。图 XX 展示了共享内存的基本布局和分配机制。

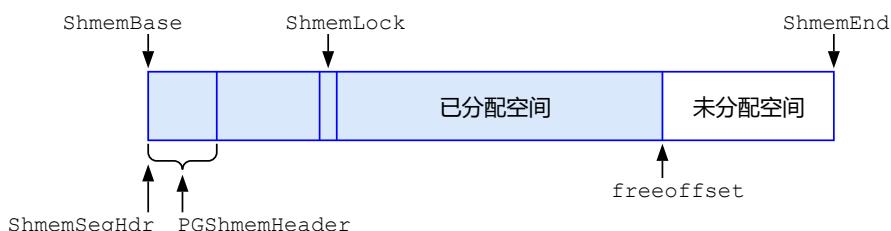


图 11.13: 共享内存的分配机制

我们可以把共享内存理解为一个巨大的数组。其头部是 `PGShmemHeader` 结构体，另外有三个指针分别指向该数组的头和尾 (`ShmemBase`/`ShmemEnd`/`ShmemSegHdr`)。结构体 `PGShmemHeader` 的定义如下：

```
/* in src/include/storage/pg_shmem.h */
typedef uint32 dsm_handle;
#define PGShmemMagic 679834894
typedef struct PGShmemHeader {
    int32      magic;          /* magic # to identify Postgres segments */
    pid_t      creatorPID;     /* PID of creating process (set but unread) */
    Size       totalsize;       /* total size of segment */
    Size       freeoffset;      /* offset to first free space */
    dsm_handle dsm_control;   /* ID of dynamic shared memory control seg */
    void      *index;          /* pointer to ShmemIndex table */
    dev_t      device;         /* device data directory is on */
    ino_t      inode;          /* inode number of data directory */
} PGShmemHeader;
```

该结构体的各成员变量的含义不难理解：magic 写死为一个固定的魔幻数用于判断该共享内存的类型，creatorPID 是创建该共享内存的进程的进程号，即主进程的进程号，totalsize 是共享内存的总大小，单位为字节，freeoffset 为空闲内存的偏移量，dsm_control 为指向动态共享内存的指针，index 指向了主哈希表，device 和 inode 可以忽略。

在共享内存创建完毕后，PostgreSQL 会初始化一个自旋锁 ShmemLock，用于串行化各进程在共享内存中分配内存的操作，具体分配内存的函数是 ShmemAllocRaw()，其逻辑也不难理解，就是把 freeoffset 偏移量往后移动而已，其代码如下：

```
/* in src/backend/storage/ipc/shmem.c */
static void * ShmemAllocRaw(Size size, Size *allocated_size)
{
    Size newStart, newFree;
    void *newSpace;
    size = CACHELINEALIGN(size); *allocated_size = size;
    SpinLockAcquire(ShmemLock);
    newStart = ShmemSegHdr->freeoffset;    newFree = newStart + size;
    if (newFree <= ShmemSegHdr->totalsize) {
        newSpace = (void *) ((char *) ShmemBase + newStart);
        ShmemSegHdr->freeoffset = newFree;
    } else newSpace = NULL;
    SpinLockRelease(ShmemLock);
    return newSpace;
}
```

11.3.2 主索引

在主进程初始化阶段会创建一个名为 ShmemIndex 的哈希表，我们称之为“主哈希表”，每个模块都在主哈希表中查找其共享数据结构。如果不存在数据结构，则调用者可以分配一个新的并初始化它。如果数据结构存在，则调用者通过在本地地址空间中初始化指针“附加”到结构。Shmem 索引有两个目的：首先，它给我们提供了一个简单的模型，说明后端进程初始化时世界的样子。如果 Shmem 索引中存在某些东西，则它被初始化。如果不存在，则未初始化。其次，Shmem 索引允许我们按需分配共享内存，而不是尝试预分配结构并在头文件中硬编码大小和位置。如果您在许多不同的地方使用大量共享内存（并在开发过程中更改内容），则这一点很重要。

在主哈希表中的元素结构定义如下，我们可以看到 K 就是一个最大长度为 47 个字符的字符串，而 V 则是指向其对应的共享内存区域。

```
/* in src/include/storage/shmem.h */
#define SHMEM_INDEX_KEYSIZE (48)
typedef struct {
    char key[SHMEM_INDEX_KEYSIZE]; /* string name */
    void *location;           /* location in shared mem */
    Size size;                /* # bytes requested for the structure */
    Size allocated_size;      /* # bytes actually allocated */
} ShmemIndexEnt;
```

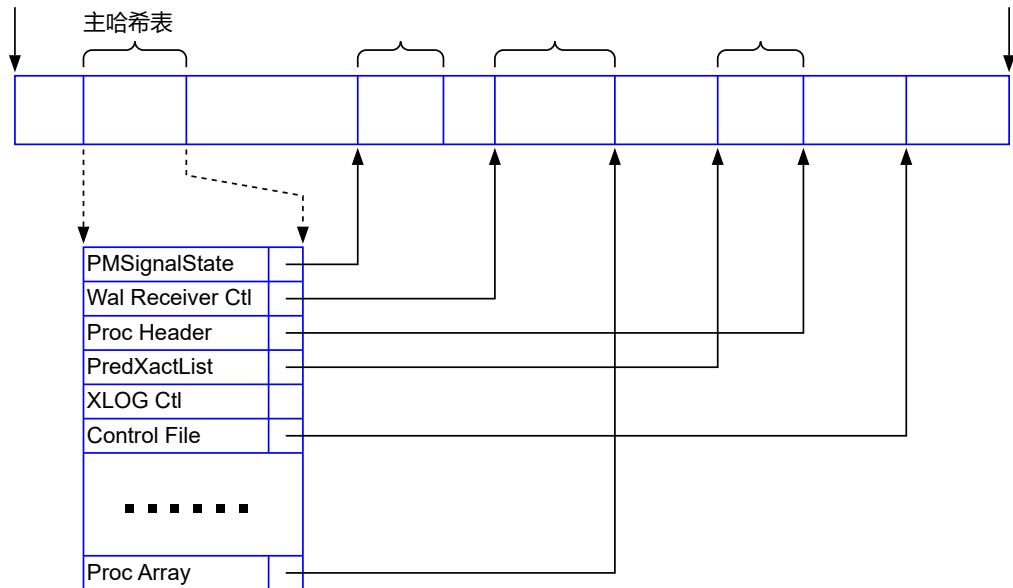


图 11.14: 共享内存中的主哈希表

下面是主哈希表的初始化函数。

```
/* in src/backend/storage/ipc/shmem.c */
#define SHMEM_INDEX_SIZE (64)
void InitShmemIndex(void)
{
    HASHCTL info;
    info.keysize = SHMEM_INDEX_KEYSIZE;
    info.entrysize = sizeof(ShmemIndexEnt);
    ShmemIndex = ShmemInitHash("ShmemIndex", SHMEM_INDEX_SIZE, SHMEM_INDEX_SIZE,
                               &info, HASH_ELEM | HASH_STRINGS);
}
```

该函数比较简单，就是指定了 K 和 V 的大小，调用函数 ShmemInitHash()，下面是相关代码：

```
/* in src/backend/storage/ipc/shmem.c */
HTAB * ShmemInitHash(const char *name, long init_size, long max_size,
                      HASHCTL *infoP, int hash_flags)
{
    bool found;
    void *location;

    /*
     * Hash tables allocated in shared memory have a fixed directory; it can't
     * grow or other backends wouldn't be able to find it. So, make sure we
     * make it big enough to start with.
     *
     * The shared memory allocator must be specified too.
     */
    infoP->dsize = infoP->max_dsize = hash_select_dirsize(max_size);
    infoP->alloc = ShmemAllocNoError;
    hash_flags |= HASH_SHARED_MEM | HASH_ALLOC | HASH_DIRSIZE;
```

```

/* look it up in the shmem index */
location = ShmemInitStruct(name,
                            hash_get_shared_size(infoP, hash_flags),
                            &found);

/*
 * if it already exists, attach to it rather than allocate and initialize
 * new space
 */
if (found) hash_flags |= HASH_ATTACH;
/* Pass location of hashtable header to hash_create */
infoP->hctl = (HASHHDR *) location;
return hash_create(name, init_size, infoP, hash_flags);
}

```

函数 ShmemInitStruct() 初始化某一个共享内存区。

```

/* in src/backend/storage/ipc/shmem.c */
void * ShmemInitStruct(const char *name, Size size, bool *foundPtr)
{
    ShmemIndexEnt *result;
    void *structPtr;
    LWLockAcquire(ShmemIndexLock, LW_EXCLUSIVE);
    if (!ShmemIndex) { /* 此时主哈希表还没有创建，下面的代码创建主哈希表 */
        PGShmemHeader *shmemseghdr = ShmemSegHdr;
        if (IsUnderPostmaster) {
            structPtr = shmemseghdr->index;
            *foundPtr = true;
        } else {
            structPtr = ShmemAlloc(size);
            shmemseghdr->index = structPtr;
            *foundPtr = false;
        }
        LWLockRelease(ShmemIndexLock);
        return structPtr;
    }
    /* 在主哈希表中查找name，如果没有找到，就插入该K */
    result = (ShmemIndexEnt *) hash_search(ShmemIndex, name, HASH_ENTER_NULL, foundPtr);
    if (!result) { /* 如果插入失败，报错 */
        LWLockRelease(ShmemIndexLock);
        ereport(ERROR, ....);
    }
    /* 至此两种可能，一个是在主哈希表中找到该K了，另外一个是未找到，但是我们把K插入主哈希表了 */
    if (*foundPtr) {
        if (result->size != size) { /* 做必要的检查，判断其大小 */
            LWLockRelease(ShmemIndexLock);
            ereport(ERROR, ....);
        }
        structPtr = result->location;
    } else { /* 在主哈希表中未找到该K，我们已经将其插入主哈希表了，下面更新V */

```

```

    Size      allocated_size;
    structPtr = ShmemAllocRaw(size, &allocated_size);
    if (structPtr == NULL) { /* 申请内存失败，则删除该K，报错 */
        /* out of memory; remove the failed ShmemIndex entry */
        hash_search(ShmemIndex, name, HASH_REMOVE, NULL);
        LWLockRelease(ShmemIndexLock);
        ereport(ERROR, ....);
    }
    result->size = size;
    result->allocated_size = allocated_size;
    result->location = structPtr;
}
LWLockRelease(ShmemIndexLock);
return structPtr;
}

```

11.3.2.1 子进程和主进程的通讯

Postmaster 通过发送 SIGUSR1 信号被其子进程通知。具体原因通过共享内存中的标志进行通信。我们为每个可能的“原因”保留一个布尔标志，以便不同的后端可以同时发出不同的原因信号。(但是，如果同时发出相同的原因信号，postmaster 只会观察到一次。)

为了实现最大的可移植性，这些标志实际上被声明为“volatile sig_atomic_t”。这应该确保标志值的加载和存储是原子的，从而允许我们省略任何显式锁定。

除了每个原因的标志之外，我们还存储了一组针对每个子进程的标志，目前仅用于检测是否存在未执行适当关闭而退出的后端。每个子进程的标志有三种可能的状态：UNUSED、ASSIGNED 和 ACTIVE。一个 UNUSED 插槽可用于分配。一个 ASSIGNED 插槽与一个 postmaster 子进程相关联，但是要么该进程尚未触及共享内存，要么它已成功清理自己。ACTIVE 插槽表示进程正在主动使用共享内存。插槽随机分配给子进程，postmaster.c 负责跟踪哪个插槽与哪个 PID 相对应。

实际上还有第四种状态，即 WALSENDER。这与 ACTIVE 相同，但携带了子进程是 WAL 发送器的额外信息。WAL 发送器也从 ACTIVE 状态开始，但一旦开始流式传输 WAL，就会切换到 WALSENDER (之后永远不会再回到 ACTIVE)。

我们还有一个用于在相反方向上进行通信的共享内存字段，从 postmaster 向子进程传递：如果确实广播了 SIGQUIT 信号，它告诉 postmaster 广播 SIGQUIT 信号的原因。

```

/* in src/include/storage/pmsignal.h */
typedef enum {
    PMSIGNAL_RECOVERY_STARTED,      /* recovery has started */
    PMSIGNAL_BEGIN_HOT_STANDBY,    /* begin Hot Standby */
    PMSIGNAL_ROTATE_LOGFILE,       /* send SIGUSR1 to syslogger to rotate logfile */
    PMSIGNAL_START_AUTOVAC_LAUNCHER, /* start an autovacuum launcher */
    PMSIGNAL_START_AUTOVAC_WORKER,  /* start an autovacuum worker */
    PMSIGNAL_BACKGROUND_WORKER_CHANGE, /* background worker state change */
    PMSIGNAL_START_WALRECEIVER,    /* start a walreceiver */
    PMSIGNAL_ADVANCE_STATE_MACHINE, /* advance postmaster's state machine */
    NUM_PMSIGNALS                  /* Must be last value of enum! */
} PMSignalReason;

/* in src/backend/storage/ipc/pmsignal.c */

```

```

#define PM_CHILD_UNUSED      0 /* these values must fit in sig_atomic_t */
#define PM_CHILD_ASSIGNED    1
#define PM_CHILD_ACTIVE      2
#define PM_CHILD_WALSENDER   3

/* "typedef struct PMSignalData PMSignalData" appears in pmsignal.h */
struct PMSignalData {
    /* per-reason flags for signaling the postmaster */
    sig_atomic_t PMSignalFlags[NUM_PMSIGNALS];
    /* global flags for signals from postmaster to children */
    QuitSignalReason sigquit_reason; /* why SIGQUIT was sent */
    /* per-child-process flags */
    int           num_child_flags; /* # of entries in PMChildFlags[] */
    sig_atomic_t PMChildFlags[FLEXIBLE_ARRAY_MEMBER];
};

```

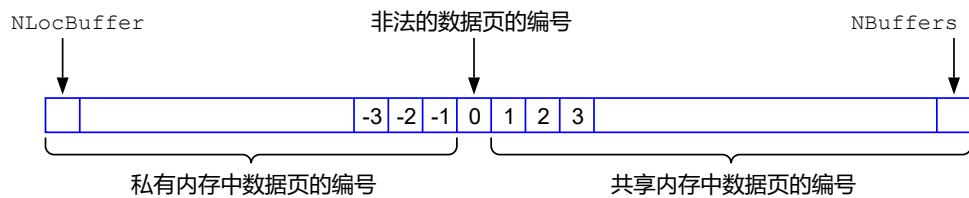


图 11.15: 数据页的编号

参考 BufferIsValid() 函数

11.3.3 共享池

共享池分配三部分，最顶部是一个哈希表 SharedBufHash，中间是数据页描述数组 BufferDescriptors，最下面是真正的共享池 BufferBlocks，这也是一个数组，是整个共享内存中最大的数组。

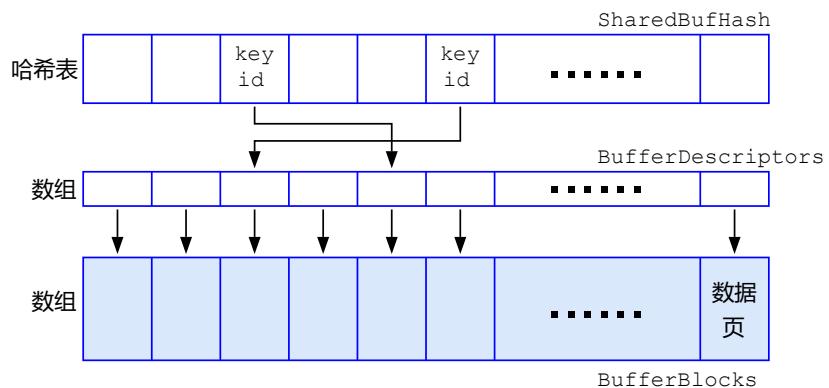


图 11.16: 共享池的整体结构

理解了整个共享内存的分配机制，共享内存不过是这个巨大数组上的一部分而已，但是其体积占据了整个共享内存 80% 到 90% 的空间。

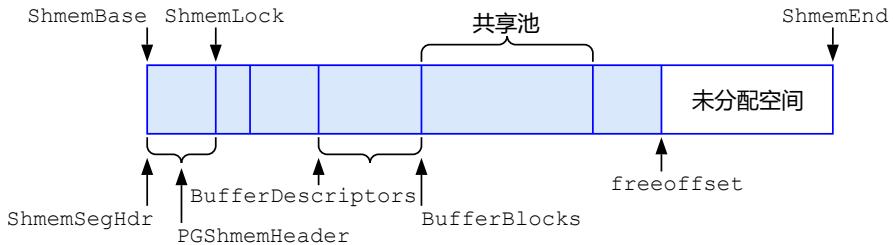


图 11.17: 共享池的结构

相关的数据结构定义如下：

```
/* in src/backend/storage/buffer/bufmgr.c */
#define BufHdrGetBlock(bufHdr) ((Block) (BufferBlocks + ((Size) (bufHdr)->buf_id) * BLCKSZ))
#define BufferGetLSN(bufHdr) (PageGetLSN(BufHdrGetBlock(bufHdr)))

typedef uint32 BlockNumber;
typedef unsigned int Oid;
/* in src/include/storage/relfilenode.h */
typedef struct RelFileNode {
    Oid spcNode; /* tablespace */
    Oid dbNode; /* database */
    Oid relNode; /* relation */
} RelFileNode;
/* in src/include/common/relpath.h */
typedef enum ForkNumber {
    InvalidForkNumber = -1,
    MAIN_FORKNUM = 0,
    FSM_FORKNUM,
    VISIBILITYMAP_FORKNUM,
    INIT_FORKNUM
} ForkNumber;
/* in src/include/storage/buf_internals.h */
typedef struct buftag {
    RelFileNode rnode; /* physical relation identifier */
    ForkNumber forkNum;
    BlockNumber blockNum; /* blknum relative to begin of reln */
} BufferTag;

/* in src/include/port/atomics/arch-x86.h */
typedef struct pg_atomic_uint32 { volatile uint32 value; } pg_atomic_uint32;
/* in src/include/storage/buf_internals.h */
typedef struct BufferDesc {
    BufferTag tag; /* ID of page contained in buffer */
    int buf_id; /* buffer's index number (from 0) */
    /* state of the tag, containing flags, refcount and usagecount */
    pg_atomic_uint32 state;
    int wait_backend_pgprocno; /* backend of pin-count waiter */
    int freeNext; /* link in freelist chain */
}
```

```

LWLock          content_lock;           /* to lock access to buffer contents */
} BufferDesc;

#define SIZEOF_VOID_P     8
#define BUFFERDESC_PAD_TO_SIZE  (SIZEOF_VOID_P == 8 ? 64 : 1)
typedef union BufferDescPadded {
    BufferDesc      bufferdesc;
    char            pad[BUFFERDESC_PAD_TO_SIZE];
} BufferDescPadded;
#define GetBufferDescriptor(id) (&BufferDescriptors[(id)].bufferdesc)

/* in src/backend/storage/buffer/buf_table.c */
typedef struct {
    BufferTag key; /* Tag of a disk page */
    int        id;  /* Associated buffer ID */
} BufferLookupEnt;

```

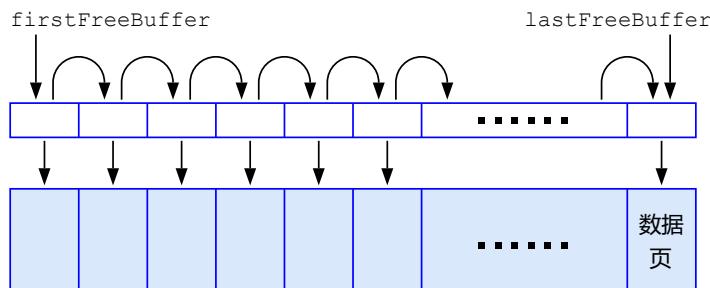


图 11.18: 共享池的空闲链表

判断共享池中是否有空闲数据页的逻辑非常简单，只要看 `firstFreeBuffer` 是否大于等于 0。

```

/* in src/backend/storage/buffer/freelist.c */
bool have_free_buffer(void)
{
    if (StrategyControl->firstFreeBuffer >= 0) return true;
    else return false;
}

```

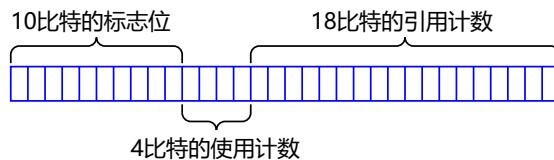


图 11.19: 共享池的状态信息

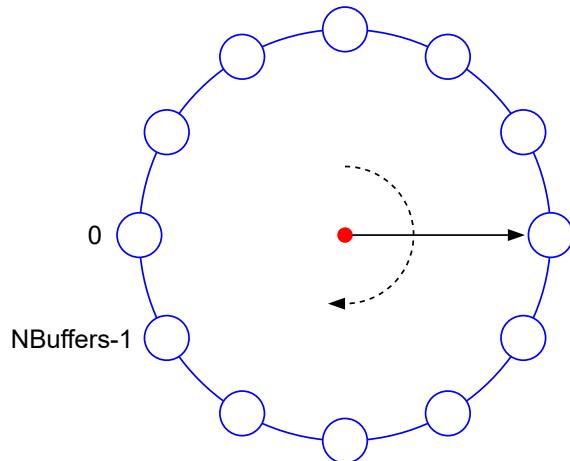


图 11.20: 数据页的替换算法

函数 WaitBufHdrUnlocked()

函数 `WaitBufHdrUnlocked()` 比较粗暴，就是反复检查该页的 `BM_LOCKED` 位是否为 0，如果为 0 就返回，否则就无限循环检查。

```
static uint32 WaitBufHdrUnlocked(BufferDesc *buf)
{
    SpinDelayStatus delayStatus;
    uint32          buf_state;

    init_local_spin_delay(&delayStatus);
    buf_state = pg_atomic_read_u32(&buf->state);
    while (buf_state & BM_LOCKED) {
        perform_spin_delay(&delayStatus);
        buf_state = pg_atomic_read_u32(&buf->state);
    }
    finish_spin_delay(&delayStatus);
    return buf_state;
}
```

对数据页进行加锁和解锁操作

```
#define BM_LOCKED  (1U << 22)
/* in src/backend/storage/buffer/bufmgr.c */
uint32 LockBufHdr(BufferDesc *desc)
{
    SpinDelayStatus delayStatus;
    uint32          old_buf_state;
    init_local_spin_delay(&delayStatus);
    while (true) { /* 通过自旋的方式尝试对该页进行加锁 */
        old_buf_state = pg_atomic_fetch_or_u32(&desc->state, BM_LOCKED);
        /* if it wasn't set before we're OK */
        if (!(old_buf_state & BM_LOCKED)) break; /* 若此时无人锁住该页则返回之 */
    }
}
```

```

        perform_spin_delay(&delayStatus);
    }
    finish_spin_delay(&delayStatus);
    return old_buf_state | BM_LOCKED;
}

/* in src/include/storage/buf_internals.h */
#define UnlockBufHdr(desc, s) \
    do { \
        pg_write_barrier(); \
        pg_atomic_write_u32(&(desc)->state, (s) & (~BM_LOCKED)); \
    } while (0)

/* in src/include/port/atomics.h */
#define pg_write_barrier() pg_write_barrier_impl()
./include/port/atomics/arch-x86.h
#define pg_write_barrier_impl() pg_compiler_barrier_impl()

./include/port/atomics/generic-gcc.h
#define pg_compiler_barrier_impl() __asm__ __volatile__("") ::: "memory")

"./include/port/atomics.h"
static inline uint32
pg_atomic_fetch_or_u32(volatile pg_atomic_uint32 *ptr, uint32 or_)
{
    AssertPointerAlignment(ptr, 4);
    return pg_atomic_fetch_or_u32_impl(ptr, or_);
}

/* in src/include/storage/proclist_types.h */
typedef struct proclist_head {
    int head; /* pgprocno of the head PGPROC */
    int tail; /* pgprocno of the tail PGPROC */
} proclist_head;

/* in src/include/storage/condition_variable.h */
typedef struct {
    spinlock_t      mutex; /* spinlock protecting the wakeup list */
    proclist_head  wakeup; /* list of wake-able processes */
} ConditionVariable;

#define CV_MINIMAL_SIZE (sizeof(ConditionVariable) <= 16 ? 16 : 32)
typedef union ConditionVariableMinimallyPadded {
    ConditionVariable cv;
    char             pad[CV_MINIMAL_SIZE];
} ConditionVariableMinimallyPadded;

/* in src/include/storage/buf_internals.h */
typedef struct CkptSortItem {
    Oid          tsId;

```

```

Oid          relNode;
ForkNumber   forkNum;
BlockNumber  blockNum;
int          buf_id;

} CkptSortItem;

/* in src/backend/storage/buffer/buf_init.c */
void InitBufferPool(void)
{
    bool foundBufs, foundDescs, foundIOCV, foundBufCkpt;

    BufferDescriptors = (BufferDescPadded *)
        ShmemInitStruct("Buffer Descriptors",
                        NBuffers * sizeof(BufferDescPadded),
                        &foundDescs);

    BufferBlocks = (char *) ShmemInitStruct("Buffer Blocks",
                                             NBuffers * (Size) BLCKSZ, &foundBufs);

    BufferIOCVArray = (ConditionVariableMinimallyPadded *)
        ShmemInitStruct("Buffer IO Condition Variables",
                        NBuffers * sizeof(ConditionVariableMinimallyPadded),
                        &foundIOCV);

/*
 * The array used to sort to-be-checkpointed buffer ids is located in
 * shared memory, to avoid having to allocate significant amounts of
 * memory at runtime. As that'd be in the middle of a checkpoint, or when
 * the checkpointer is restarted, memory allocation failures would be
 * painful.
*/
    CkptBufferIds = (CkptSortItem *) ShmemInitStruct("Checkpoint BufferIds",
                                                       NBuffers * sizeof(CkptSortItem), &foundBufCkpt);

    if (foundDescs || foundBufs || foundIOCV || foundBufCkpt) {
        /* should find all of these, or none of them */
        Assert(foundDescs && foundBufs && foundIOCV && foundBufCkpt);
        /* note: this path is only taken in EXEC_BACKEND case */
    } else {
        int i;
        /*
         * Initialize all the buffer headers.
         */
        for (i = 0; i < NBuffers; i++) {
            BufferDesc *buf = GetBufferDescriptor(i);
            CLEAR_BUFFERTAG(buf->tag);
            pg_atomic_init_u32(&buf->state, 0);
            buf->wait_backend_pgprocno = INVALID_PGPROCNO;
            buf->buf_id = i;
            /*
             * Initially link all the buffers together as unused. Subsequent

```

```

    * management of this list is done by freelist.c.
    */
    buf->freeNext = i + 1;
    LWLockInitialize(BufferDescriptorGetContentLock(buf),
                     LWTRANCHE_BUFFER_CONTENT);
    ConditionVariableInit(BufferDescriptorGetIOCV(buf));
}
/* Correct last entry of linked list */
GetBufferDescriptor(NBuffers - 1)->freeNext = FREENEXT_END_OF_LIST;
}
/* Init other shared buffer-management stuff */
StrategyInitialize(!foundDescs);
/* Initialize per-backend file flush context */
WritebackContextInit(&BackendWritebackContext, &backend_flush_after);
}

```

11.3.3.1 分配策略

```

/* in src/include/storage/bufmgr.h */
typedef enum BufferAccessStrategyType {
    BAS_NORMAL,      /* Normal random access */
    BAS_BULKREAD,   /* Large read-only scan (hint bit updates are ok) */
    BAS_BULKWRITE,  /* Large multi-block write (e.g. COPY IN) */
    BAS_VACUUM      /* VACUUM */
} BufferAccessStrategyType;

/* in src/include/storage/buf.h */
typedef int Buffer;
#define InvalidBuffer 0
/*
 * Private (non-shared) state for managing a ring of shared buffers to re-use.
 * This is currently the only kind of BufferAccessStrategy object, but someday
 * we might have more kinds.
*/
typedef struct BufferAccessStrategyData
{
    /* Overall strategy type */
    BufferAccessStrategyType btype;
    /* Number of elements in buffers[] array */
    int                  ring_size;

    /*
     * Index of the "current" slot in the ring, ie, the one most recently
     * returned by GetBufferFromRing.
    */
    int                  current;

    /*

```

```

 * True if the buffer just returned by StrategyGetBuffer had been in the
 * ring already.
 */
bool current_was_in_ring;

/*
 * Array of buffer numbers. InvalidBuffer (that is, zero) indicates we
 * have not yet selected a buffer for this ring slot. For allocation
 * simplicity this is palloc'd together with the fixed fields of the
 * struct.
 */
Buffer buffers[FLEXIBLE_ARRAY_MEMBER];
} BufferAccessStrategyData;
typedef struct BufferAccessStrategyData *BufferAccessStrategy;

/* in src/backend/storage/buffer/freelist.c */

/*
 * The shared freelist control information.
 */
typedef struct {
    /* Spinlock: protects the values below */
    slock_t buffer_strategy_lock;
    /*
     * Clock sweep hand: index of next buffer to consider grabbing. Note that
     * this isn't a concrete buffer - we only ever increase the value. So, to
     * get an actual buffer, it needs to be used modulo NBuffers.
     */
    pg_atomic_uint32 nextVictimBuffer;
    int firstFreeBuffer;    /* Head of list of unused buffers */
    int lastFreeBuffer; /* Tail of list of unused buffers */
    /*
     * NOTE: lastFreeBuffer is undefined when firstFreeBuffer is -1 (that is,
     * when the list is empty)
     */
    uint32 completePasses; /* Complete cycles of the clock sweep */
    pg_atomic_uint32 numBufferAllocs; /* Buffers allocated since last reset */
    /*
     * Bgworker process to be notified upon activity or -1 if none. See
     * StrategyNotifyBgWriter.
     */
    int bgwprocno;
} BufferStrategyControl;
}

```

```

/* Pointers to shared state */

static BufferStrategyControl *StrategyControl = NULL;

/*
 * StrategyInitialize -- initialize the buffer cache replacement
 *          strategy.
 *
 * Assumes: All of the buffers are already built into a linked list.
 *          Only called by postmaster and only during initialization.
 */
void StrategyInitialize(bool init)
{
    bool found;

    /*
     * Initialize the shared buffer lookup hashtable.
     *
     * Since we can't tolerate running out of lookup table entries, we must be
     * sure to specify an adequate table size here. The maximum steady-state
     * usage is of course NBuffers entries, but BufferAlloc() tries to insert
     * a new entry before deleting the old. In principle this could be
     * happening in each partition concurrently, so we could need as many as
     * NBuffers + NUM_BUFFER_PARTITIONS entries.
     */
    InitBufTable(NBuffers + NUM_BUFFER_PARTITIONS);

    /*
     * Get or create the shared strategy control block
     */
    StrategyControl = (BufferStrategyControl *)
        ShmemInitStruct("Buffer Strategy Status", sizeof(BufferStrategyControl), &found);

    if (!found) {
        /*
         * Only done once, usually in postmaster
         */
        Assert(init);
        SpinLockInit(&StrategyControl->buffer_strategy_lock);
        /*
         * Grab the whole linked list of free buffers for our strategy. We
         * assume it was previously set up by InitBufferPool().
         */
        StrategyControl->firstFreeBuffer = 0;
        StrategyControl->lastFreeBuffer = NBuffers - 1;
        /* Initialize the clock sweep pointer */
        pg_atomic_init_u32(&StrategyControl->nextVictimBuffer, 0);
        /* Clear statistics */
        StrategyControl->completePasses = 0;
        pg_atomic_init_u32(&StrategyControl->numBufferAllocs, 0);
        /* No pending notification */
    }
}

```

```

        StrategyControl->bgwprocno = -1;
    }
    else
        Assert(!init);
}

/* in src/backend/storage/buffer/freelist.c:StrategyGetBuffer() */
#define BUF_REFCOUNT_ONE 1
trycounter = NBuffers;
for (;;) {
    buf = GetBufferDescriptor(ClockSweepTick());
    /*
     * If the buffer is pinned or has a nonzero usage_count, we cannot use
     * it; decrement the usage_count (unless pinned) and keep scanning.
     */
    local_buf_state = LockBufHdr(buf);
    if (BUF_STATE_GET_REFCOUNT(local_buf_state) == 0) {
        if (BUF_STATE_GET_USAGECOUNT(local_buf_state) != 0) {
            local_buf_state -= BUF_USAGECOUNT_ONE;
            trycounter = NBuffers;
        } else {
            /* Found a usable buffer */
            if (strategy != NULL) AddBufferToRing(strategy, buf);
            *buf_state = local_buf_state;
            return buf;
        }
    } else if (--trycounter == 0) {
        /*
         * We've scanned all the buffers without making any state changes,
         * so all the buffers are pinned (or were when we looked at them).
         * We could hope that someone will free one eventually, but it's
         * probably better to fail than to risk getting stuck in an
         * infinite loop.
         */
        UnlockBufHdr(buf, local_buf_state);
        elog(ERROR, "no unpinned buffers available");
    }
    UnlockBufHdr(buf, local_buf_state);
}

```

函数 StrategyGetBuffer() 分析

```

/* in src/backend/storage/buffer/freelist.c */
/*
 * StrategyGetBuffer
 *
 * Called by the bufmgr to get the next candidate buffer to use in
 * BufferAlloc(). The only hard requirement BufferAlloc() has is that
 * the selected buffer must not currently be pinned by anyone.

```

```

/*
 * strategy is a BufferAccessStrategy object, or NULL for default strategy.
 *
 * To ensure that no one else can pin the buffer before we do, we must
 * return the buffer with the buffer header spinlock still held.
 */

#define FREENEXT_NOT_IN_LIST (-2)

BufferDesc* StrategyGetBuffer(BufferAccessStrategy strategy, uint32 *buf_state)
{
    BufferDesc  *buf;
    int          bgwprocno, trycounter;
    uint32       local_buf_state; /* to avoid repeated (de-)referencing */

    /*
     * If given a strategy object, see whether it can select a buffer. We
     * assume strategy objects don't need buffer_strategy_lock.
     */
    if (strategy != NULL) {
        buf = GetBufferFromRing(strategy, buf_state);
        if (buf != NULL) return buf;
    }

    /*
     * If asked, we need to waken the bgwriter. Since we don't want to rely on
     * a spinlock for this we force a read from shared memory once, and then
     * set the latch based on that value. We need to go through that length
     * because otherwise bgwprocno might be reset while/after we check because
     * the compiler might just reread from memory.
     *
     * This can possibly set the latch of the wrong process if the bgwriter
     * dies in the wrong moment. But since PGPROC->procLatch is never
     * deallocated the worst consequence of that is that we set the latch of
     * some arbitrary process.
     */
    bgwprocno = INT_ACCESS_ONCE(StrategyControl->bgwprocno);
    if (bgwprocno != -1) {
        /* reset bgwprocno first, before setting the latch */
        StrategyControl->bgwprocno = -1;
        /*
         * Not acquiring ProcArrayLock here which is slightly icky. It's
         * actually fine because procLatch isn't ever freed, so we just can
         * potentially set the wrong process' (or no process') latch.
         */
        SetLatch(&ProcGlobal->allProcs[bgwprocno].procLatch);
    }

    /*
     * We count buffer allocation requests so that the bgwriter can estimate
     * the rate of buffer consumption. Note that buffers recycled by a

```

```

 * strategy object are intentionally not counted here.
 */
pg_atomic_fetch_add_u32(&StrategyControl->numBufferAllocs, 1);

/*
 * First check, without acquiring the lock, whether there's buffers in the
 * freelist. Since we otherwise don't require the spinlock in every
 * StrategyGetBuffer() invocation, it'd be sad to acquire it here -
 * uselessly in most cases. That obviously leaves a race where a buffer is
 * put on the freelist but we don't see the store yet - but that's pretty
 * harmless, it'll just get used during the next buffer acquisition.
 *
 * If there's buffers on the freelist, acquire the spinlock to pop one
 * buffer of the freelist. Then check whether that buffer is usable and
 * repeat if not.
 *
 * Note that the freeNext fields are considered to be protected by the
 * buffer_strategy_lock not the individual buffer spinlocks, so it's OK to
 * manipulate them without holding the spinlock.
*/
if (StrategyControl->firstFreeBuffer >= 0) { /* 若有空闲页 */
    while (true) {
        /* Acquire the spinlock to remove element from the freelist */
        SpinLockAcquire(&StrategyControl->buffer_strategy_lock);
        if (StrategyControl->firstFreeBuffer < 0) { /* 最后一个空闲页被别人捷足先登 */
            SpinLockRelease(&StrategyControl->buffer_strategy_lock);
            break;
        }
        /* 空闲页还在，则取之 */
        buf = GetBufferDescriptor(StrategyControl->firstFreeBuffer);
        Assert(buf->freeNext != FREENEXT_NOT_IN_LIST);

        /* Unconditionally remove buffer from freelist */
        StrategyControl->firstFreeBuffer = buf->freeNext;
        buf->freeNext = FREENEXT_NOT_IN_LIST;

        /*
         * Release the lock so someone else can access the freelist while
         * we check out this buffer.
         */
        SpinLockRelease(&StrategyControl->buffer_strategy_lock);

        /*
         * If the buffer is pinned or has a nonzero usage_count, we cannot
         * use it; discard it and retry. (This can only happen if VACUUM
         * put a valid buffer in the freelist and then someone else used
         * it before we got to it. It's probably impossible altogether as
         * of 8.3, but we'd better check anyway.)
         */
    }
}

```

```

local_buf_state = LockBufHdr(buf);
if (BUF_STATE_GET_REFCOUNT(local_buf_state) == 0
    && BUF_STATE_GET_USAGECOUNT(local_buf_state) == 0) {
    if (strategy != NULL)
        AddBufferToRing(strategy, buf);
    *buf_state = local_buf_state;
    return buf;
}
UnlockBufHdr(buf, local_buf_state);
}

/* Nothing on the freelist, so run the "clock sweep" algorithm */
trycounter = NBuffers;
for (;;) {
    buf = GetBufferDescriptor(ClockSweepTick());

    /*
     * If the buffer is pinned or has a nonzero usage_count, we cannot use
     * it; decrement the usage_count (unless pinned) and keep scanning.
     */
    local_buf_state = LockBufHdr(buf);

    if (BUF_STATE_GET_REFCOUNT(local_buf_state) == 0) {
        if (BUF_STATE_GET_USAGECOUNT(local_buf_state) != 0) {
            local_buf_state -= BUF_USAGECOUNT_ONE;
            trycounter = NBuffers;
        } else {
            /* Found a usable buffer */
            if (strategy != NULL) AddBufferToRing(strategy, buf);
            *buf_state = local_buf_state;
            return buf;
        }
    } else if (--trycounter == 0) {
        /*
         * We've scanned all the buffers without making any state changes,
         * so all the buffers are pinned (or were when we looked at them).
         * We could hope that someone will free one eventually, but it's
         * probably better to fail than to risk getting stuck in an
         * infinite loop.
         */
        UnlockBufHdr(buf, local_buf_state);
        elog(ERROR, "no unpinned buffers available");
    }
    UnlockBufHdr(buf, local_buf_state);
}
}

```

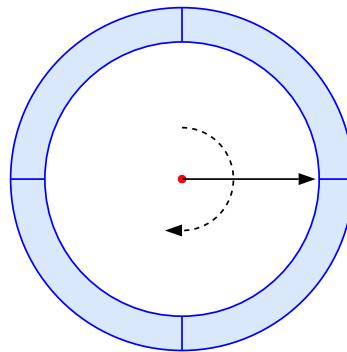


图 11.21: 数据页形成的环

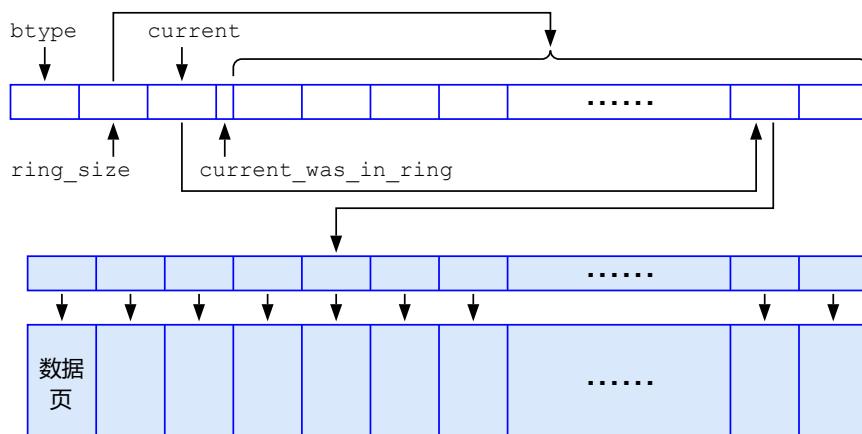


图 11.22: 结构体 BufferAccessStrategy 的结构

```

/* in src/backend/storage/buffer/freelist.c */
/*
 * GetAccessStrategy -- create a BufferAccessStrategy object
 *
 * The object is allocated in the current memory context.
 */
BufferAccessStrategy GetAccessStrategy(BufferAccessStrategyType btype)
{
    BufferAccessStrategy strategy;
    int             ring_size;

    /*
     * Select ring size to use.  See buffer/README for rationales.
     *
     * Note: if you change the ring size for BAS_BULKREAD, see also
     * SYNC_SCAN_REPORT_INTERVAL in access/heap/syncscan.c.
     */
    switch (btype) {
        case BAS_NORMAL:
            /* if someone asks for NORMAL, just give 'em a "default" object */
            return NULL;
    }
}

```

```

    case BAS_BULKREAD:
        ring_size = 256 * 1024 / BLCKSZ;
        break;
    case BAS_BULKWRITE:
        ring_size = 16 * 1024 * 1024 / BLCKSZ;
        break;
    case BAS_VACUUM:
        ring_size = 256 * 1024 / BLCKSZ;
        break;
    default:
        elog(ERROR, "unrecognized buffer access strategy: %d", (int) btype);
        return NULL;           /* keep compiler quiet */
}

/* Make sure ring isn't an undue fraction of shared buffers */
ring_size = Min(NBuffers / 8, ring_size);

/* Allocate the object and initialize all elements to zeroes */
strategy = (BufferAccessStrategy)
    palloc0(offsetof(BufferAccessStrategyData, buffers) + ring_size * sizeof(Buffer));

/* Set fields that don't start out zero */
strategy->btype = btype;
strategy->ring_size = ring_size;

return strategy;
}

/*
 * GetBufferFromRing -- returns a buffer from the ring, or NULL if the
 *         ring is empty.
 *
 * The bufhdr spin lock is held on the returned buffer.
 */
static BufferDesc* GetBufferFromRing(BufferAccessStrategy strategy, uint32 *buf_state)
{
    BufferDesc  *buf;
    Buffer      bufnum;
    uint32      local_buf_state; /* to avoid repeated (de-)referencing */

    /* Advance to next ring slot */
    if (++strategy->current >= strategy->ring_size) strategy->current = 0;

    /*
     * If the slot hasn't been filled yet, tell the caller to allocate a new
     * buffer with the normal allocation strategy. He will then fill this
     * slot by calling AddBufferToRing with the new buffer.
     */
    bufnum = strategy->buffers[strategy->current];
}

```

```

if (bufnum == InvalidBuffer) {
    strategy->current_was_in_ring = false;
    return NULL;
}

/*
 * If the buffer is pinned we cannot use it under any circumstances.
 *
 * If usage_count is 0 or 1 then the buffer is fair game (we expect 1,
 * since our own previous usage of the ring element would have left it
 * there, but it might've been decremented by clock sweep since then). A
 * higher usage_count indicates someone else has touched the buffer, so we
 * shouldn't re-use it.
*/
buf = GetBufferDescriptor(bufnum - 1);
local_buf_state = LockBufHdr(buf);
if (BUF_STATE_GET_REFCOUNT(local_buf_state) == 0
    && BUF_STATE_GET_USAGECOUNT(local_buf_state) <= 1) {
    strategy->current_was_in_ring = true;
    *buf_state = local_buf_state;
    return buf;
}
UnlockBufHdr(buf, local_buf_state);

/*
 * Tell caller to allocate a new buffer with the normal allocation
 * strategy. He'll then replace this ring element via AddBufferToRing.
 */
strategy->current_was_in_ring = false;
return NULL;
}

```

11.3.4 磁盘管理

这里研究读写磁盘的具体细节。

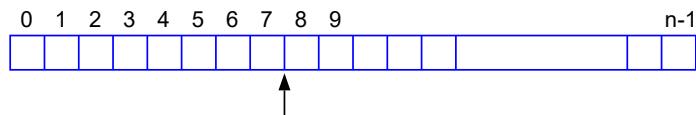


图 11.23: 文件的基本概念

```

typedef struct RelFileNode {
    Oid spcNode; /* tablespace */
    Oid dbNode; /* database */
    Oid relNode; /* relation */
} RelFileNode;
typedef int BackendId;

```

```

/*
 * Augmenting a relfilenode with the backend ID provides all the information
 * we need to locate the physical storage. The backend ID is InvalidBackendId
 * for regular relations (those accessible to more than one backend), or the
 * owning backend's ID for backend-local relations. Backend-local relations
 * are always transient and removed in case of a database crash; they are
 * never WAL-logged or fsync'd.
 */
typedef struct RelFileNodeBackend {
    RelFileNode    node;
    BackendId     backend;
} RelFileNodeBackend;
/* in src/include/storage/smgr.h */
typedef struct SMgrRelationData {
    /* rnode is the hashtable lookup key, so it must be first! */
    RelFileNodeBackend smgr_rnode; /* relation physical identifier */

    /* pointer to owning pointer, or NULL if none */
    struct SMgrRelationData **smgr_owner;

    /*
     * The following fields are reset to InvalidBlockNumber upon a cache flush
     * event, and hold the last known size for each fork. This information is
     * currently only reliable during recovery, since there is no cache
     * invalidation for fork extension.
     */
    BlockNumber smgr_targblock; /* current insertion target block */
    BlockNumber smgr_cached_nblocks[MAX_FORKNUM + 1]; /* last known size */

    /* additional public fields may someday exist here */

    /*
     * Fields below here are intended to be private to smgr.c and its
     * submodules. Do not touch them from elsewhere.
     */
    int    smgr_which;          /* storage manager selector */

    /*
     * for md.c; per-fork arrays of the number of open segments
     * (md_num_open_segs) and the segments themselves (md_seg_fds).
     */
    int    md_num_open_segs[MAX_FORKNUM + 1];
    struct _MdfdVec *md_seg_fds[MAX_FORKNUM + 1];

    /* if unowned, list link in list of all unowned SMgrRelations */
    dlist_node      node;
} SMgrRelationData;

typedef SMgrRelationData *SMgrRelation;

```

文件的基本操作

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
```

函数 `read()` 和 `pread()` 是最基础的对文件读取的系统调用。函数 `read()` 会从描述符为 `fd` 的文件中读取 `count` 个字节存入 `buf` 中，而 `pread()` 则是从描述符为 `fd` 的文件中，从 `offset` 位置开始，读取 `count` 个字节存入 `buf` 中。如果读取成功，这两个系统调用都将返回读取的字节数。因此，这两个系统调用主要的区别就在于读取的位置，其它功能均类似。

```
/* in src/port/pread.c */
#ifndef HAVE_PREAD
#define pg_pread pread
#endif
#ifndef HAVE_PWRITE
#define pg_pwrite pwrite
#endif
ssize_t pg_pread(int fd, void *buf, size_t size, off_t offset)
{
    if (lseek(fd, offset, SEEK_SET) < 0) return -1;
    return read(fd, buf, size);
}
/* in src/port/pwrite.c */
ssize_t pg_pwrite(int fd, const void *buf, size_t size, off_t offset)
{
    if (lseek(fd, offset, SEEK_SET) < 0) return -1;
    return write(fd, buf, size);
}
```

```
/* in src/backend/storage/smgr/smgr.c */
typedef struct f_smgr {
    void (*smgr_init) (void); /* may be NULL */
    void (*smgr_shutdown) (void); /* may be NULL */
    void (*smgr_open) (SMgrRelation reln);
    void (*smgr_close) (SMgrRelation reln, ForkNumber forknum);
    void (*smgr_create) (SMgrRelation reln, ForkNumber forknum, bool isRedo);
    bool (*smgr_exists) (SMgrRelation reln, ForkNumber forknum);
    void (*smgr_unlink) (RelFileNodeBackend rnode, ForkNumber forknum, bool isRedo);
    void (*smgr_extend) (SMgrRelation reln, ForkNumber forknum,
                         BlockNumber blocknum, char *buffer, bool skipFsync);
    bool (*smgr_prefetch) (SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum);
    void (*smgr_read) (SMgrRelation reln, ForkNumber forknum,
                      BlockNumber blocknum, char *buffer);
    void (*smgr_write) (SMgrRelation reln, ForkNumber forknum,
                       BlockNumber blocknum, char *buffer, bool skipFsync);
    void (*smgr_writeback) (SMgrRelation reln, ForkNumber forknum,
                           BlockNumber blocknum, BlockNumber nblocks);
    BlockNumber (*smgr_nblocks) (SMgrRelation reln, ForkNumber forknum);
}
```

```

void (*smgr_truncate) (SMgrRelation reln, ForkNumber forknum, BlockNumber nblocks);
void (*smgr_immedsync) (SMgrRelation reln, ForkNumber forknum);
} f_smgr;

static const f_smgr smgrsw[] = {
    /* magnetic disk */
{
    .smgr_init = mdinit,
    .smgr_shutdown = NULL,
    .smgr_open = mdopen,
    .smgr_close = mdclose,
    .smgr_create = mdcreate,
    .smgr_exists = mdexists,
    .smgr_unlink = mdunlink,
    .smgr_extend = mdextend,
    .smgr_prefetch = mdprefetch,
    .smgr_read = mdread,
    .smgr_write = mdwrite,
    .smgr_writeback = mdwriteback,
    .smgr_nblocks = mdnblocks,
    .smgr_truncate = mdtruncate,
    .smgr_immedsync = mdimmedsync,
}
};

void mdread(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum, char *buffer)
{
    off_t      seekpos;
    int        nbytes;
    MdfdVec   *v;

    TRACE_POSTGRESQL_SMGR_MD_READ_START(forknum, blocknum,
                                         reln->smgr_rnode.node.spcNode,
                                         reln->smgr_rnode.node.dbNode,
                                         reln->smgr_rnode.node.relNode,
                                         reln->smgr_rnode.backend);

    v = _mdfd_getseg(reln, forknum, blocknum, false,
                      EXTENSION_FAIL | EXTENSION_CREATE_RECOVERY);

    seekpos = (off_t) BLCKSZ * (blocknum % ((BlockNumber) RELSEG_SIZE));

    Assert(seekpos < (off_t) BLCKSZ * RELSEG_SIZE);

    nbytes = FileRead(v->mdfd_vfd, buffer, BLCKSZ, seekpos, WAIT_EVENT_DATA_FILE_READ);

    TRACE_POSTGRESQL_SMGR_MD_READ_DONE(forknum, blocknum,
                                         reln->smgr_rnode.node.spcNode,
                                         reln->smgr_rnode.node.dbNode,
                                         reln->smgr_rnode.node.relNode,
                                         reln->smgr_rnode.backend);
}

```

```

        reln->smgr_rnode.node.relNode,
        reln->smgr_rnode.backend,
        nbytes,
        BLCKSZ);

if (nbytes != BLCKSZ) {
    if (nbytes < 0)
        ereport(ERROR,
            (errcode_for_file_access(),
             errmsg("could not read block %u in file \"%s\": %m",
                   blocknum, FilePathName(v->mdfd_vfd))));

/*
 * Short read: we are at or past EOF, or we read a partial block at
 * EOF. Normally this is an error; upper levels should never try to
 * read a nonexistent block. However, if zero_damaged_pages is ON or
 * we are InRecovery, we should instead return zeroes without
 * complaining. This allows, for example, the case of trying to
 * update a block that was later truncated away.
 */
if (zero_damaged_pages || InRecovery)
    MemSet(buffer, 0, BLCKSZ);
else
    ereport(ERROR,
        (errcode(ERRCODE_DATA_CORRUPTED),
         errmsg("could not read block %u in file \"%s\": read only %d of %d bytes",
               blocknum, FilePathName(v->mdfd_vfd),
               nbytes, BLCKSZ)));
}

}

/*
 * mdwrite() -- Write the supplied block at the appropriate location.
 *
 * This is to be used only for updating already-existing blocks of a
 * relation (ie, those before the current EOF). To extend a relation,
 * use mdextend().
 */
void mdwrite(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum,
             char *buffer, bool skipFsync)
{
    off_t      seekpos;
    int       nbytes;
    MdfdVec  *v;

    /* This assert is too expensive to have on normally ... */
#endifif CHECK_WRITE_VS_EXTEND
    Assert(blocknum < mdnblocks(reln, forknum));
#endifif
}

```

```

TRACE_POSTGRESQL_SMGR_MD_WRITE_START(forknum, blocknum,
                                       reln->smgr_rnode.node.spcNode,
                                       reln->smgr_rnode.node.dbNode,
                                       reln->smgr_rnode.node.relNode,
                                       reln->smgr_rnode.backend);

v = _mdfd_getseg(reln, forknum, blocknum, skipFsync,
                  EXTENSION_FAIL | EXTENSION_CREATE_RECOVERY);

seekpos = (off_t) BLCKSZ * (blocknum % ((BlockNumber) RELSEG_SIZE));

Assert(seekpos < (off_t) BLCKSZ * RELSEG_SIZE);

nbytes = FileWrite(v->mdfd_vfd, buffer, BLCKSZ, seekpos, WAIT_EVENT_DATA_FILE_WRITE);

TRACE_POSTGRESQL_SMGR_MD_WRITE_DONE(forknum, blocknum,
                                      reln->smgr_rnode.node.spcNode,
                                      reln->smgr_rnode.node.dbNode,
                                      reln->smgr_rnode.node.relNode,
                                      reln->smgr_rnode.backend,
                                      nbytes,
                                      BLCKSZ);

if (nbytes != BLCKSZ) {
    if (nbytes < 0)
        ereport(ERROR,
                (errcode_for_file_access(),
                 errmsg("could not write block %u in file \"%s\": %m",
                        blocknum, FilePathName(v->mdfd_vfd))));
    /* short write: complain appropriately */
    ereport(ERROR,
            (errcode(ERRCODE_DISK_FULL),
             errmsg("could not write block %u in file \"%s\": wrote only %d of %d bytes",
                    blocknum,
                    FilePathName(v->mdfd_vfd),
                    nbytes, BLCKSZ),
             errhint("Check free disk space.")));
}

if (!skipFsync && !SmgrIsTemp(reln))
    register_dirty_segment(reln, forknum, v);
}

/* in src/backend/storage/file/fd.c */

int FileRead(File file, char *buffer, int amount, off_t offset, uint32 wait_event_info)
{
    int    returnCode;

```

```

Vfd *vfdP;

Assert(FileIsValid(file));

DO_DB(elog(LOG, "FileRead: %d (%s) " INT64_FORMAT " %d %p",
           file, VfdCache[file].fileName,
           (int64) offset,
           amount, buffer));

returnCode = FileAccess(file);
if (returnCode < 0)
    return returnCode;

vfdP = &VfdCache[file];

retry:
pgstat_report_wait_start(wait_event_info);
returnCode = pg_pread(vfdP->fd, buffer, amount, offset);
pgstat_report_wait_end();

if (returnCode < 0) {
/*
 * Windows may run out of kernel buffers and return "Insufficient
 * system resources" error. Wait a bit and retry to solve it.
 *
 * It is rumored that EINTR is also possible on some Unix filesystems,
 * in which case immediate retry is indicated.
 */
#endif WIN32
DWORD      error = GetLastError();

switch (error)
{
    case ERROR_NO_SYSTEM_RESOURCES:
        pg_usleep(1000L);
        errno = EINTR;
        break;
    default:
        _dosmaperr(error);
        break;
}
#endif /* OK to retry if interrupted */
if (errno == EINTR)
    goto retry;
}

return returnCode;
}

```

VfdCache 是一个可以动态增加的数组，VfdCache[0] 比较特殊，仅仅表明环的开始和结束。VfdCache 池有两个链表，一个是 LRU 池的双向链表，一个是空闲链表，这是一个单向链表，首部是 VfdCache[0].nextFree，如果该值为 0，则表明空闲链表中没有空闲的 VfdCache 了，需要扩容。

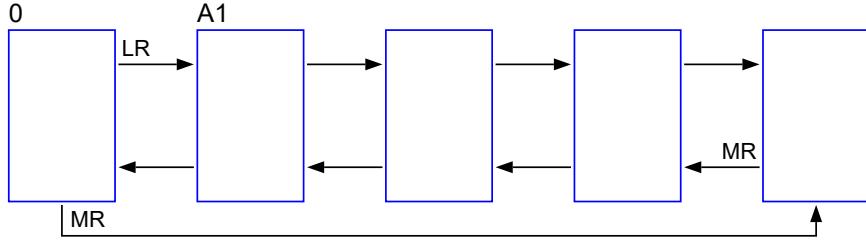


图 11.24: VFD 的 LRU 池

```

/* in src/backend/storage/file/fd.c */
/*
 * A number of platforms allow individual processes to open many more files
 * than they can really support when *many* processes do the same thing.
 * This GUC parameter lets the DBA limit max_safe_fds to something less than
 * what the postmaster's initial probe suggests will work.
 */
int max_files_per_process = 1000;
#define FD_MINFREE 48
int max_safe_fds = FD_MINFREE; /* default if not changed */

#define VFD_CLOSED (-1)
typedef int File;
typedef struct vfd {
    int fd; /* current FD, or VFD_CLOSED if none */
    unsigned short fdstate; /* bitflags for VFD's state */
    ResourceOwner resowner; /* owner, for automatic cleanup */
    File nextFree; /* link to next free VFD, if in freelist */
    File lruMoreRecently; /* doubly linked recency-of-use list */
    File lruLessRecently;
    off_t fileSize; /* current size of file (0 if not temporary) */
    char *fileName; /* name of file, or NULL for unused VFD */
    /* NB: fileName is malloc'd, and must be free'd when closing the VFD */
    int fileFlags; /* open(2) flags for (re)opening the file */
    mode_t fileMode; /* mode to pass to open(2) */
} Vfd;

/*
 * Virtual File Descriptor array pointer and size. This grows as
 * needed. 'File' values are indexes into this array.
 * Note that VfdCache[0] is not a usable VFD, just a list header.
 */

```

```
static Vfd *VfdCache;  
static Size SizeVfdCache = 0;
```

11.3.5 WAL Buffer 池

这里研究 WAL Buffer 池的具体细节。

11.4 PostgreSQL 进程间通讯

这里介绍 PostgreSQL 各进程之间是如何通讯的。

附录 A 使用 GDB 调试 PostgreSQL 源代码快速入门

虽然我们可以通过阅读各种介绍 PostgreSQL 内部技术内幕的资料来深入学习 PostgreSQL，但纸上得来终觉浅，绝知此事要躬行。千看万看，不如撸起袖子自己干，只有自己动手运行和单步调试 PostgreSQL 源代码，才能获得第一手的知识。而且调试过程中获得的很多细节知识是很微妙的，无法在互联网上找到，但对于清晰地理解技术内幕又非常关键，所以我建议读者学习到适当的程度后一定要自己动手调试 PostgreSQL 的源代码。在 Linux 环境中所使用的调试器的头牌自然是大名鼎鼎的 GDB 了。本附录介绍一下 GDB 调试器的基本使用和在调试 PostgreSQL 源代码过程中的一些使用技巧。

附录 B 作者小传

先生不知何许人也，亦不详其姓字，仅以“小布”之名自称，小小布衣之意也。先生漂泊海外多年，闲静少言，不慕荣利，好读书，不求甚解，每有会意，便欣然忘食。性嗜古文，中华书局之《古文观止》竖排繁体版环侍左右，终生不厌。先生亦爱汇编，C，Perl，SVG 和 L^AT_EX，平生未尝写过 Java 程式。常著 IT 技术文章自娱，颇示己志，忘怀得失，以此自终。

钱氏曰：鸡蛋好吃，你又何必认识下蛋的母鸡。故欲联系作者，仅一邮件地址 (itgotousa@gmail.com)，供读者反馈鸡蛋味道之用耳。