排除PostgreSQL的故障

在第11章 "决定有用的扩展 "中，我们了解了一些被广泛采用的有用的扩展，它们可以给你的部署带来真正的提升。作为后续，现在将向你介绍PostgreSQL的故障排除。其目的是给你一个系统的方法来检查和修复你的系统，以提高性能和避免常见的陷阱。 有一个系统的方法肯定会得到回报。

在这一章中，将包括以下主题。

- 接近一个未知的数据库
- 检查pg_stat_activity
- 检查慢的查询
- 检查日志
- 检查丢失的索引
- 检查内存和I/O
- 了解值得注意的错误情况

请记住，许多事情都可能出错，因此，对数据库进行专业监测是很重要的。要弄清楚什么地方出了问题，你必须以专业的方式来研究这个系统。

1.接近一个未知的数据库

如果你碰巧要管理一个大规模的系统，你可能不知道系统到底在做什么。管理数以百计的系统意味着你不会知道每个系统发生了什么。 当涉及到故障排除时，最重要的事情可以归结为一个词：数据。如果没有足够的数据，就没有办法解决事情。 因此，排除故障的第一步就是一定要设置一个监控工具，比如pgwatch2（可在https://www.cybertec-postgresql.com/en/products/pgwatch2/），它可以让你对数据库服务器有一些了解。 一旦报告工具告诉你一个值得检查的情况，就意味着它已经被证明是有用的，可以有组织地接近系统。

2.检查pg_stat_activity

首先，让我们检查pg_stat_statements的内容，并回答以下问题。

- 目前你的系统上有多少个并发的查询正在执行?
- 你是否看到类似类型的查询一直显示在查询栏中?
- 你是否看到有的查询已经运行了很长时间?
- 是否有任何锁没有被授予?
- 你是否看到有来自可疑主机的连接?

应该首先检查pg_stat_activity视图，因为它可以让我们了解系统上正在发生什么。当然，图形化监控应该给你一个系统的第一印象。 然而，在一天结束的时候，它真正归结为在服务器上实际运行的查询。因此，由pg_stat_activity提供的对系统的良好概述对于追踪问题是非常重要的。

为了让你更轻松，我汇编了一些我认为对尽快发现问题有用的查询。

2.1 查询 pg_stat_activity

下面的查询显示你的数据库目前正在执行多少个查询。

```
test=# SELECT datname,
 count(*) AS open,
 count(*) FILTER (WHERE state = 'active') AS active,
 count(*) FILTER (WHERE state = 'idle') AS idle,
 count(*) FILTER (WHERE state = 'idle in transaction')
 AS idle_in_trans
FROM pg_stat_activity
WHERE backend_type = 'client backend'
GROUP BY ROLLUP(1);
 datname | open | active | idle | idle_in_trans
---------+------+--------+------+---------------
 test | 2 | 1 | 0 | 1
 | 2 | 1 | 0 | 1
(2 rows)
```

为了在同一个屏幕上显示尽可能多的信息，使用了部分聚合。我们可以看到活跃、闲置和闲置在交易中的查询。 如果我们可以看到大量的闲置在事务中的查询，那么肯定要深入挖掘，以便弄清楚这些事务被保持了多长时间。下面的列表显示了可以找到多长时间的交易。

```
test=# SELECT pid, xact_start, now() - xact_start AS duration
 FROM pg_stat_activity
 WHERE state LIKE '%transaction%'
 ORDER BY 3 DESC;
 pid | xact_start | duration
--------+-------------------------------+-----------------
 19758 | 2020-09-26 20:27:08.168554+01 | 22:12:10.194363
(1 row)
```

前面列表中的事务已经持续了22个多小时。现在的主要问题是：一个事务怎么可能开放这么长时间？在大多数应用中，一个需要这么长时间的事务是非常可疑的，而且可能是非常危险的。危险从何而来？正如我们在本书前面所学到的，只有在没有事务可以看到死行的情况下，VACUUM命令才能清理这些死行。现在，如果一个事务持续开放几个小时甚至几天，VACUUM命令就不能产生有用的结果，这将导致

表的膨胀。

因此，我们强烈建议确保长的事务被监控或杀死，以防它们变得太长。从9.6版本开始，PostgreSQL有一个叫做快照过期的功能，它允许我们在快照过期的情况下终止长事务。

检查是否有任何长期运行的查询，也是一个好主意。

```
test=# SELECT now() - query_start AS duration, datname, query
 FROM pg_stat_activity
 WHERE state = 'active'
 ORDER BY 1 DESC;
 duration | datname | query
-----------------+---------+---------------------------
 00:00:38.814526 | dev | SELECT pg_sleep(10000);
 00:00:00 | test | SELECT now() - query_start AS duration,
 datname, query
 FROM pg_stat_activity
 WHERE state = 'active'
 ORDER BY 1 DESC;
(2 rows)
```

在这种情况下，所有活跃的查询都被抽取出来，报表计算出每个查询已经活跃了多长时间。通常情况下，我们会看到类似的查询排在前面，这可以给我们提供一些有价值的线索，说明我们的系统正在发生什么。

2.2 处理 Hibernate 语句

许多对象关系映射 (ORM)，例如 Hibernate，会生成非常长的 SQL 语句。问题在于：pg_stat_activity 只会存储系统视图中查询的前 1024 个字节。其余部分被截断。对于 Hibernate 等 ORM 生成的长查询，查询在感兴趣的部分（FROM 子句等）真正开始之前被切断。

这个问题的解决方案是在 postgresql.conf 文件中设置一个配置参数：

```
test=# SHOW track_activity_query_size;
 track_activity_query_size
---------------------------
 1kB
(1 row)
```

如果我们把这个参数增加到一个合理的高值（也许是32,768），并重新启动PostgreSQL，那么我们将能够看到更长的查询，并能够更容易地发现问题。

2.3 找出查询的来源

在检查pg_stat_activity时，有一些字段会告诉我们一个查询来自哪里。

```
client_addr | inet |
client_hostname | text |
client_port | integer |
```

这些字段将包含IP地址和主机名（如果已配置）。但是，如果每个应用程序都从同一个IP发送请求，例如，所有的应用程序都驻扎在同一个应用服务器上，会发生什么？我们将很难看到哪个应用程序产生了某个查询。

解决这个问题的办法是要求开发人员设置一个application_name变量。

```
test=# SHOW application_name ;
 application_name
------------------
 psql
(1 row)
test=# SET application_name TO 'some_name';
SET
test=# SHOW application_name ;
 application_name
------------------
 some_name
(1 row)
```

如果人们合作，application_name变量将显示在系统视图中，使人们更容易看到查询的来源。application_name变量也可以作为连接字符串的一部分来设置。在下一步，我们将尝试弄清与缓慢查询有关的一切。

3.检查慢的查询

在检查了pg_stat_activity之后，看一下缓慢的、耗时的查询是有意义的。基本上，有两种方法可以解决这个问题：

- 在日志中寻找单个的慢速查询
- 找出耗时过长的查询类型

寻找单一的、缓慢的查询是性能调整的经典方法。 通过设置log_min_duration_statement变量到一个期望的阈值，PostgreSQL将开始为每个超过这个阈值的查询写一个日志行。默认情况下，慢查询日志是关闭的，如下所示。

```
test=# SHOW log_min_duration_statement;
 log_min_duration_statement
----------------------------
 -1
(1 row)
```

然而，将这个变量设置为一个合理的值是非常合理的。根据你的工作量，所需的时间当然可能不同。 在许多情况下，期望值可能因数据库而异。 因此，也可以以更精细的方式使用该变量。

在许多情况下，所需的值可能因数据库而异。 因此，也可以以更精细的方式使用该变量。

```
test=# ALTER DATABASE test SET log_min_duration_statement TO 10000;
ALTER DATABASE
```

如果你的数据库面临不同的工作负荷，只为某个数据库设置该参数是非常有意义的。

当使用慢速查询日志时，必须考虑一个重要的因素--许多较小的查询可能会导致更大的负载，而不仅仅是少数几个运行缓慢的查询。当然，意识到个别慢速查询总是有意义的，但有时，这些查询并不是问题所在。

考虑下面的例子：在你的系统上，执行了100万个查询，每个需要500毫秒，同时还有一些分析性查询，每个运行了几毫秒。显然，真正的问题永远不会出现在慢速查询日志中，而每一次数据导出、每一次索引创建和每一次批量加载（无论如何在大多数情况下是无法避免的）都会在日志中出现垃圾信息，给我们指出错误的方向。

因此，我个人的建议是，使用慢查询日志，但要小心使用，谨慎使用。但最重要的是，要注意我们真正在测量什么。

在我看来，更好的方法是更深入地使用pg_stat_statements视图。它将提供汇总的信息，而不仅仅是关于单个查询的信息。本书前面已经讨论了pg_stat_statements视图。然而，这个模块的重要性怎么强调都不为过。

3.1 检查单个查询

有时，慢的查询被识别出来，但我们仍然不知道到底发生了什么。下一步当然是检查查询的执行计划，看看会发生什么。识别计划中那些导致不良运行时间的关键操作是相当简单的。试着使用下面的检查表。

- 试着看看在计划的什么地方，时间开始急剧上升。
- 检查是否有缺失的索引（性能不佳的主要原因之一）。
- 使用EXPLAIN子句（缓冲区为真，分析为真，以此类推），看看你的查询是否使用了太多的缓冲区。
- 打开track_io_timing参数，弄清是I/O问题还是CPU问题（明确检查是否有随机I/O发生）。
- 寻找不正确的估计，并尝试修复它们。
- 寻找那些执行频率过高的存储过程。
- 试着找出其中一些是否可以被标记为STABLE或IMMUTABLE，如果可以的话。

注意，pg_stat_statements不考虑解析时间，所以如果你的查询非常长（比如查询字符串），那么pg_stat_statements可能会有一点误导。

3.2 使用 perf 进行更深入的挖掘

在大多数情况下，通过这个微小的检查表工作将帮助你以相当快和有效的方式追踪大多数的问题。然而，即使是从数据库引擎中提取的信息有时也是不够的。

perf工具是一个用于Linux的分析工具，它允许你直接看到哪些C函数在你的系统上引起问题。通常情况下，perf不是默认安装的，所以建议你安装它。要在你的服务器上使用perf，只需以root身份登录并运行以下命令。

```
perf top
```

屏幕每隔几秒钟就会刷新一次，你将有机会看到现场发生的情况。下面的列表向你展示了一个标准的、只读的基准可能是什么样子。

```
Samples: 164K of event 'cycles:ppp', Event count (approx.): 109789128766
Overhead Shared Object Symbol
 3.10% postgres [.] AllocSetAlloc
 1.99% postgres [.] SearchCatCache
 1.51% postgres [.] base_yyparse
 1.42% postgres [.] hash_search_with_hash_value
 1.27% libc-2.22.so [.] vfprintf
 1.13% libc-2.22.so [.] _int_malloc
 0.87% postgres [.] palloc
 0.74% postgres [.] MemoryContextAllocZeroAligned
 0.66% libc-2.22.so [.] __strcmp_sse2_unaligned
```

```
  0.66% [kernel] [k] _raw_spin_lock_irqsave
  0.66% postgres [.] _bt_compare
  0.63% [kernel] [k] __fget_light
  0.62% libc-2.22.so [.] strlen
```

你可以看到，在我们的样本中，没有任何一个函数占用过多的CPU时间，这告诉我们，系统是很好的。

然而，情况可能并不总是这样的。有一个叫做自旋锁争夺的问题是很常见的。自旋锁被PostgreSQL核心用来同步诸如缓冲区访问等事情。自旋锁是现代CPU提供的一个功能，以避免操作系统对小操作（如增加一个数字）的互动。如果你认为你可能面临自旋锁的争夺，其症状如下。

- 一个真正的高CPU负载。
- 难以置信的低吞吐量（通常需要几毫秒的查询突然需要几秒钟）。
- I/O异常低，因为CPU正忙于交易锁。

在许多情况下，自旋锁争用是突然发生的。你的系统刚刚还好好的，突然间，负载上升了，吞吐量像石头一样下降了。perf top命令会显示，大部分时间都花在一个叫s_lock的C函数上。如果是这种情况，你应该尝试做以下工作。

```
huge_pages = try # on, off, or try
```

将 huge_pages 从 try 改为 off。在操作系统层面上完全关闭巨大页面可能是一个好主意。一般来说，似乎有些内核比其他内核更容易产生这类问题。红帽2.6.32系列似乎特别糟糕（注意，我在这里使用了似乎这个词）。

如果你正在使用PostGIS，perf工具也很有趣。如果列表中的顶级函数都是与GIS有关的（比如，来自一些底层库），你就知道问题很可能不是来自PostgreSQL的不良调整，而只是与需要时间完成的昂贵操作有关。

4.检查日志

如果你的系统闻到了麻烦的味道，检查日志以了解发生了什么是有意义的。重要的一点是：并不是所有的日志条目都是同样创建的。PostgreSQL有一个从DEBUG到PANIC的日志条目的层次结构。

对于管理员来说，以下三个错误级别是非常重要的。

- ERROR
- FATAL
- PANIC

ERROR用于处理诸如语法错误、权限相关问题等问题。你的日志将总是包含错误信息。关键的因素是--某种类型的错误出现的频率如何？ 产生数以百万计的语法错误当然不是运行数据库服务器的理想策略。

FATAL比ERROR更可怕；你会看到诸如无法为共享内存名分配内存或意外的walreceiver状态等信息。换句话说，这些错误信息已经非常可怕了，会告诉你事情正在出错。

最后，是PANIC。如果你碰到这种信息，你就知道事情真的非常非常不对劲。PANIC的典型例子是锁表被破坏，或者创建了太多的信号灯。这些都会导致关机。在下一节中，你将了解丢失索引的情况。

5.检查丢失的索引

一旦我们完成了前三个步骤，重要的是看一下总体的性能。正如我在本书中一直强调的那样，缺失的索引是造成超差性能的全部原因。所以，每当我们面对一个缓慢的系统时，建议我们检查是否有缺失的索引，并部署任何需要的东西。

通常情况下，客户要求我们优化RAID级别，调整内核，或者其他一些花哨的东西。在现实中，这些复杂的要求往往可以归结为少量的索引丢失。根据我的判断，花一些额外的时间来检查所有需要的索引是否在那里总是有意义的。检查缺失的索引既不难也不费时，所以应该一直这样做，不管你面临什么样的性能问题。

这里是我最喜欢的查询，以获得对可能缺少索引的地方的印象。

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
 idx_scan, seq_tup_read / seq_scan AS avg
FROM pg_stat_user_tables
WHERE seq_scan > 0
ORDER BY seq_tup_read DESC
LIMIT 20;
```

试着找到经常被扫描的大表（平均价值高）。 这些表通常会排在前面。

6.检查内存和I/O

一旦我们完成了寻找丢失的索引，我们就可以检查内存和I/O。为了弄清楚发生了什么，激活track_io_timing是有意义的。 如果它是开启的，PostgreSQL将收集关于磁盘等待时间的信息并将其呈现给你。

通常，客户问的主要问题是：如果我们增加更多的磁盘，它是否会更快？有可能猜到会发生什么，但一般来说，测量是更好、更有用的策略。 启用track_io_timing将帮助你收集数据，以真正搞清楚这个问题。

PostgreSQL以各种方式暴露了磁盘等待时间。检查事情的一个方法是看一下pg_stat_database。

```
test=# \d pg_stat_database
 View "pg_catalog.pg_stat_database"
Column | Type | Modifiers
---------------+--------------------------+-----------
datid | oid |
datname | name |
...
conflicts | bigint |
temp_files | bigint |
temp_bytes | bigint |
...
blk_read_time | double precision |
blk_write_time | double precision |
```

请注意，在最后有两个字段 - blk_read_time和blk_write_time。它们将告诉我们PostgreSQL花了多少时间来等待操作系统的响应。请注意，我们在这里不是真正的测量磁盘等待时间，而是测量操作系统返回数据所需的时间。

如果操作系统产生缓存命中，这个时间会相当低。如果操作系统要做非常讨厌的随机I/O，我们会看到一个区块甚至需要几毫秒。

在很多情况下，当temp_files和temp_bytes显示高数字时，就会出现高的blk_read_time和blk_write_time。另外，在许多情况下，这指向一个糟糕的work_mem设置或一个糟糕的maintain_work_mem设置。记住这一点：如果PostgreSQL不能在内存中做事情，它必须溢出到磁盘上。你可以使用temp_files操作来检测这一点。只要有temp_files，就有可能出现讨厌的磁盘等待时间。

虽然在每个数据库层面上的全局视图是有意义的，但它并不能产生关于麻烦的真正来源的深入信息。通常情况下，只有少数查询是导致性能不佳的原因。发现这些问题的方法是使用pg_stat_statements。

```
test=# \d pg_stat_statements
 View "public.pg_stat_statements"
 Column | Type | Modifiers
---------------------+------------------+-----------
 ...
 query | text |
 calls | bigint |
 total_time | double precision |
 ...
 temp_blks_read | bigint |
 temp_blks_written | bigint |
 blk_read_time | double precision |
 blk_write_time | double precision |
```

你将能够看到，在每个查询的基础上，是否有磁盘等待。重要的部分是blk_time值和total_time的组合。这个比例才是最重要的。一般来说，一个显示超过30%的磁盘等待的查询可以被看作是严重的I/O绑定。

一旦我们完成了对PostgreSQL系统表的检查，检查Linux上的vmstat命令告诉我们的情况是有意义的。或者，我们可以使用iostat命令。

```
[hs@zenbook ~]$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r b swpd free buff cache si so bi bo in cs us sy id wa st
 0 0 367088 199488 96 2320388 0 2 83 96 106 156 16 6 78 0 0
 0 0 367088 198140 96 2320504 0 0 0 10 595 2624 3 1 96 0 0
 0 0 367088 191448 96 2320964 0 0 0 8 920 2957 8 2 90 0 0
```

在做数据库工作时，我们应该把注意力集中在三个字段上：bi、bo和wa。bi字段告诉我们读取的块数；1,000相当于1 Mbps。bo字段是关于块的输出。它告诉我们写到磁盘上的数据量。在某种程度上，bi和bo是原始吞吐量。我不会认为一个数字是有害的。如果一个问题的wa值很高呢？bi和bo字段的低值，加上一个高的wa值，告诉我们一个潜在的磁盘瓶颈，这很可能与你系统上发生的大量随机I/O有关。wa值越高，你的查询速度就越慢，因为你在等待磁盘的响应。

> 良好的原始吞吐量是一件好事，但有时，它也可以指出一个问题。如果在线交易处理（OLTP）系统需要高吞吐量，它可以告诉你没有足够的RAM来缓存东西，或者索引丢失，PostgreSQL不得不读取太多的数据。请记住，事物是相互联系的，不应该把数据看作是孤立的。

7.了解值得注意的错误情况

在通过基本指南找出您将在数据库中遇到的最常见问题之后，接下来的部分将讨论 PostgreSQL 世界中发生的一些最常见的错误场景。

7.1 面对堵塞腐败

PostgreSQL有一个叫做提交日志的东西（现在叫做pg_xact；它的正式名称是pg_clog）。它跟踪系统中每个事务的状态，帮助PostgreSQL确定是否可以看到某一行。一般来说，一个事务可以处于四种状态。

```
#define TRANSACTION_STATUS_IN_PROGRESS 0x00
#define TRANSACTION_STATUS_COMMITTED 0x01
#define TRANSACTION_STATUS_ABORTED 0x02
#define TRANSACTION_STATUS_SUB_COMMITTED 0x03
```

在PostgreSQL数据库实例（pg_xact）中，clog有一个单独的目录。

在过去，人们曾报告过一种叫做clog损坏的东西，它可能是由有问题的磁盘或PostgreSQL中的bug引起的，这些年已经被修复了。一个损坏的提交日志是一个相当讨厌的东西，因为我们所有的数据都在那里，但PostgreSQL不知道事情是否仍然有效。这方面的损坏无异于一场彻底的灾难。

管理员是如何发现提交日志被破坏的呢？下面是我们通常看到的情况

```
ERROR: could not access status of transaction 118831
```

如果PostgreSQL不能访问一个事务的状态，就会出现问题。 主要的问题是--如何才能解决这个问题？直截了当地告诉你，没有办法真正解决这个问题--我们只能尝试尽可能多地拯救数据。

正如我们已经说过的，提交日志为每个事务保留 2 位。这意味着我们每个字节有四个事务，每个块剩下 32,768 个事务。一旦我们弄清楚它是哪个块，我们就可以伪造事务日志：

```
dd if=/dev/zero of=<data directory location>/pg_clog/0001 bs=256K count=1
```

我们可以用dd来伪造事务日志，并将提交状态设置为所需的值。核心问题其实是--应该使用哪种事务状态？答案是，任何状态其实都是错的，因为我们真的不知道这些事务是如何结束的。

然而，通常情况下，为了减少数据损失，将它们设置为已提交是一个好主意。这真的取决于我们的工作量和我们的数据，以决定什么是较少的破坏性

当我们不得不使用这种技术时，我们应该在必要时尽可能少地伪造堵塞。记住，我们基本上是在伪造提交状态，这对数据库引擎来说不是一件好事。

一旦我们完成了伪造的堵塞，我们应该以最快的速度创建一个备份，并从头开始重新创建数据库实例。我们正在使用的系统不再是非常值得信赖的，所以我们应该尝试尽可能快地提取数据。请记住这一点：我们即将提取的数据可能是矛盾的和错误的，所以我们将确保对我们能够从数据库服务器中抢救出来的任何东西进行一些质量检查。

7.2 了解检查点消息

检查点对于数据完整性和性能至关重要。检查点相距越远，性能通常越好。在 PostgreSQL 中，默认配置通常相当保守，因此检查点相对较快。如果同时在数据库核心中更改大量数据，则 PostgreSQL 可能会告诉我们它认为检查点过于频繁。日志文件将显示以下条目：

```
LOG: checkpoints are occurring too frequently (2 seconds apart)
LOG: checkpoints are occurring too frequently (3 seconds apart)
```

在由于转储/恢复或其他一些大型操作导致的大量写入期间，PostgreSQL 可能会注意到配置参数太低。一条消息被发送到 LOG 文件以准确地告诉我们

如果我们看到这种信息，出于性能的考虑，强烈建议我们通过大幅增加max_wal_size参数来增加检查点的距离（在旧版本中，该设置称为checkpoint_segments）。在最近的PostgreSQL版本中，默认配置已经比以前好很多了。然而，写入数据过于频繁的情况仍然容易发生

当我们看到关于检查点的信息时，有一件事我们必须牢记。过于频繁的检查点一点都不危险--它只是碰巧导致了糟糕的性能。写只是比原来慢了很多，但我们的数据没有危险。适当增加两个检查点之间的距离会使错误消失，同时也会加快我们的数据库实例的速度

7.3 管理损坏的数据页

PostgreSQL是一个非常稳定的数据库系统。它尽可能地保护数据，而且多年来已经证明了它的价值。然而，PostgreSQL依赖于坚实的硬件和正常工作的文件系统。如果存储系统坏了，PostgreSQL也会坏--除了增加复制以使事情更安全之外，我们对此没有什么办法。

偶尔会发生文件系统或磁盘故障的情况。然而，在许多情况下，整个系统不会出问题；只是有几个区块因某种原因而损坏。最近，我们已经看到这种情况发生在虚拟环境中。一些虚拟机默认不刷新磁盘，这意味着PostgreSQL不能依赖写入磁盘的东西。这种行为会导致难以预测的随机问题。

当一个区块不能再被读取时，你可能会遇到一个错误信息，如下面这样。

```
"could not read block %u in file "%s": %m"
```

你将要运行的查询会出错并停止工作。 幸运的是，PostgreSQL有一个处理这些事情的方法。

```
test=# SET zero_damaged_pages TO on;
SET
test=# SHOW zero_damaged_pages;
 zero_damaged_pages
--------------------
 on
(1 row)
```

zero_damaged_pages变量是一个配置变量，它允许我们处理破碎的页面。PostgreSQL不会抛出一个错误，而是将该区块简单地填充为零。

请注意，这肯定会导致数据丢失。但请记住，无论如何，这些数据之前就已经损坏或丢失了，所以这只是处理我们的存储系统中发生的坏事所造成的损坏的一种方法。

> 我建议大家小心处理zero_damaged_pages这个变量--当你调用它时要注意你在做什么。

7.4 粗心的连接管理

在PostgreSQL中，每个数据库连接都是一个独立的进程。所有这些进程都使用共享内存进行同步（技术上讲，在大多数情况下，它是映射的内存，但对于这个例子，这没有什么区别）。 这个共享内存包含了I/O缓存、活动的数据库连接列表、锁和其他使系统正常运行的重要东西。

当一个连接被关闭时，它将从共享内存中删除所有相关的条目，并使系统处于正常的状态。然而，当一个数据库连接由于某种原因简单地崩溃时，会发生什么呢?

Postmaster（主进程）将检测到其中一个子进程丢失。然后，所有其他的连接将被终止，一个前滚进程将被初始化。为什么必须这样做呢？当一个进程崩溃时，很可能发生共享内存区被该进程编辑的情况。换句话说，一个崩溃的进程可能会使共享内存处于损坏的状态。因此，postmaster会做出反应，在损坏在系统中蔓延之前将所有人踢出去。 所有的内存都被清理了，每个人都必须重新连接

从终端用户的角度来看，这感觉就像PostgreSQL崩溃并重新启动了，但事实并非如此。由于一个进程不能对自己的崩溃（分段故障）或其他一些信号做出反应，为了保护你的数据，清理一切是绝对必要的。

如果你在一个数据库连接上使用kill -9命令，也会发生同样的情况。该连接不能捕捉信号（根据定义，-9不能被捕捉），因此，postmaster必须再次做出反应。

7.5 对抗表的膨胀

在处理PostgreSQL时，表的膨胀是最重要的问题之一。当我们面临糟糕的性能时，弄清楚是否有对象需要的空间比它们应该有的多得多，总是一个好主意。

我们怎样才能弄清表的膨胀发生在哪里？请看pg_stat_user_tables视图。

```
test=# \d pg_stat_user_tables
 View "pg_catalog.pg_stat_user_tables"
 Column | Type | Modifiers
------------------+------------------+-----------
 relid | oid |
 schemaname | name |
 relname | name |
 ...
 n_live_tup | bigint |
 n_dead_tup | bigint |
```

n_live_tup和n_dead_tup字段让我们对正在发生的事情有一个印象，我们也可以使用pgstattuple

如果出现了严重的表膨胀，我们可以做什么呢？第一个选择是运行VACUUM FULL命令。问题是，VACUUM FULL子句需要一个表锁。在一个大表上，这可能是一个真正的问题，因为在表被重写的时候，用户不能写到该表上

> 如果你至少使用PostgreSQL 9.6，你可以使用一个叫做pg_squeeze的工具。它在幕后组织一个表，而不会阻塞（https://www.cybertec-postgresql.com/en/products/pg_squeeze/）。如果你要重新组织一个非常大的表，这特别有用

8.总结

在这一章中，我们已经学会了如何系统地接近数据库系统，并检测人们在使用PostgreSQL时所面临的最常见的问题。我们了解了一些重要的系统表，以及其他一些可以决定我们是成功还是失败的重要因素。

在本书的最后一章，我们将把注意力放在迁移到PostgreSQL上。如果你正在使用Oracle或其他一些数据库系统，你可能想看看PostgreSQL。在第13章，迁移到PostgreSQL，我们将讨论与此有关的一切。

9.问题

- 为什么数据库不能自我管理?
- PostgreSQL是否经常遇到损坏?
- PostgreSQL 是否需要经常照顾?