理解备份和复制

1.了解事务日志
- 1.1 查看事务日志
- 1.2 了解检查点
- 1.3 优化事务日志

2.事务日志的存档和恢复
- 2.1 配置归档
- 2.2 配置 pg_hba.conf 文件
- 2.3 创建基础备份
- 2.4 减少备份带宽
- 2.5 映射表空间
- 2.6 使用不同的格式
- 2.7 测试事务日志归档
- 2.8 重放事务日志
- 2.9 找到正确的时间戳
- 2.10 清理事务日志存档

3.设置异步复制
- 3.1 执行基本设置
- 3.2 提高安全性
- 3.3 停止和恢复复制
- 3.4 检查复制以确保可用性
- 3.5 执行故障转移和了解时间表
- 3.6 管理冲突
- 3.7 使复制更可靠

4.升级到同步复制
- 4.1 调整耐久度

5.利用复制槽
- 5.1 处理物理复制槽
- 5.2 处理逻辑复制槽
- 5.3 逻辑复制槽的用例

6.利用CREATE PUBLICATION和CREATE SUBSCRIPTION命令

7.总结

8.问题

在第9章 "处理备份和恢复 "中，我们学到了很多关于备份和恢复的知识，这对管理来说至关重要。到目前为止，只涉及到了逻辑备份；我将在本章中改变这种情况。

这一章是关于PostgreSQL的事务日志，以及我们可以用它来改善我们的设置，使事情更安全。

在本章中，我们将讨论以下主题：

- 了解事务日志
- 事务日志的存档和恢复
- 设置异步复制
- 升级到同步复制
- 利用复制槽
- 利用CREATE PUBLICATION和CREATE SUBSCRIPTION命令

在本章结束时，你将能够设置事务日志存档和复制。请记住，本章不可能成为复制的全面指南；它只是一个简短的介绍。对复制的全面介绍需要500页左右。只是作为比较，仅PostgreSQL Replication一书，也是来自Packt，就接近400页。本章将以更紧凑的形式涵盖最基本的东西。

1.了解事务日志

每一个现代的数据库系统都提供了一些功能，以确保系统能够在出错或有人拔掉插头的情况下经受住崩溃。这对文件系统和数据库系统都是如此。 PostgreSQL也提供了一种方法来确保崩溃不能损害数据的完整性或数据本身。它可以保证，如果断电，系统总是能够再次启动并完成其工作。

提供这种安全的手段是通过提前写入日志（WAL）或xlog实现的。这个想法是不直接写进数据文件，而是先写进日志。为什么这很重要？想象一下，我们正在写一些数据，如下所示。

```
INSERT INTO data ... VALUES ('12345678');
```

我们假设这些数据是直接写到数据文件中的。如果操作中途失败，数据文件就会被破坏。它可能包含写了一半的行，没有索引指针的列，丢失的提交信息，等等。由于硬件并不能真正保证大块数据的原子写入，所以必须找到一种方法来使其更加健壮。通过写到日志而不是直接写到文件，这个问题可以得到解决。

> 在 PostgreSQL 中，事务日志由记录组成。

一个单一的写可以由各种记录组成，这些记录都有一个校验和，并被链在一起。一个单一的事务可能包含B树、索引、存储管理器、提交记录，以及更多。每种类型的对象都有自己的WAL条目，以确保该对象能够在崩溃后存活。如果发生崩溃，PostgreSQL将启动并根据事务日志修复数据文件，以确保不允许发生永久性损坏。

介绍完了，现在让我们对事务日志有一个基本的了解。

1.1 查看事务日志

在PostgreSQL中，WAL通常可以在数据目录下的pg_wal目录中找到，除非在initdb中另外指定。在PostgreSQL的旧版本中，WAL目录被称为pg_xlog，但随着PostgreSQL 10.0的引入，该目录被重新命名。

其原因是，更多的时候，人们会删除pg_xlog目录的内容，这当然会导致严重的问题和潜在的数据库损坏。因此，社区采取了史无前例的措施，对PostgreSQL实例内的目录进行重命名。 希望能使这个名字足够可怕，以至于没有人敢于再次删除内容。

以下清单显示了 pg_wal 目录的样子：

```
[postgres@zenbook pg_wal]$ pwd
/var/lib/pgsql/13/data/pg_wal
[postgres@zenbook pg_wal]$ ls -l
total 688132
-rw-------. 1 postgres postgres 16777216 Jan 19 07:58 0000000100000000000000CD
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04 0000000100000000000000CE
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04 0000000100000000000000CF
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04 0000000100000000000000D0
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04 0000000100000000000000D1
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04 0000000100000000000000D2
```

我们可以看到的是，事务日志是一个16MB的文件，由24位数字组成。编号是十六进制的。我们可以看到，CF后面是D0。这些文件总是一个固定的大小。

> 有一点需要注意的是，在PostgreSQL中，事务日志文件的数量与事务的大小无关。你可以有一组非常小的事务日志文件，但仍然可以轻松地运行一个多TB级的事务。

传统上，WAL目录通常由16MB的文件组成。然而，自从引入PostgreSQL后，现在可以用initdb设置WAL段的大小。在某些情况下，这可以加快事情的进展。下面是它的工作原理。 下面的例子告诉我们如何将WAL文件的大小改为32MB。

```
initdb -D /pgdata --wal-segsize=32
```

1.2 了解检查点

正如我前面提到的，每一个变化都是以二进制格式写入WAL的（它不包含SQL）。问题是这样的--数据库服务器不能永远向WAL写下去，因为随着时间的推移，它将消耗越来越多的空间。所以，在某些时候，事务日志必须被回收。这是由检查点完成的，它在后台自动发生。

这个想法是，当数据被写入时，它首先进入事务日志，然后一个脏缓冲区被放入共享缓冲区。这些脏缓冲区必须进入磁盘，由后台写入器或在检查点期间写出到数据文件中。一旦所有的脏缓冲区都被写入，事务日志就可以被删除。

> 请永远不要手动删除事务日志文件。在崩溃的情况下，数据库服务器将无法再次启动，而且随着新事务的到来，所需的磁盘空间量无论如何都会被回收。永远不要手动触摸事务日志。PostgreSQL会自己处理事情，在那里做事情真的很有害。

1.3 优化事务日志

检查点是自动发生的，由服务器触发。 然而，有一些配置设置决定何时启动检查点。postgresql.conf文件中的下列参数负责处理检查点。

```
#checkpoint_timeout = 5min # range 30s-1d
#max_wal_size = 1GB
#min_wal_size = 80MB
```

启动检查点有两个原因：

- 如果我们可能耗尽了时间或空间。
- 两个检查点之间的最大时间由checkpoint_timeout变量定义。

为存储事务日志提供的空间量将在min_wal_size和max_wal_size变量之间变化。PostgreSQL会自动触发检查点，真正需要的空间量将介于这两个数字之间。

> max_wal_size变量是一个软限制，PostgreSQL可能（在重负载下）暂时需要多一点空间。换句话说，如果我们的事务日志是在一个单独的磁盘上，确保实际上有多一点的空间可以用来存储WAL是有意义的。

有人如何调整PostgreSQL 9.6和13.0中的事务日志？在9.6中，对后台写入器和检查点机制做了一些改变。在旧版本中，有一些用例，从性能的角度来看，较小的检查点距离实际上是有意义的。在9.6及以后的版本中，这种情况已经基本改变，更宽的检查点距离基本上总是非常有利的，因为许多优化可以在数据库和操作系统层面上应用，以加快事情。最值得注意的优化是，块在被写出之前被排序，这大大减少了机械磁盘上的随机I/O。

但还有一点。大的检查点距离实际上会减少创建的WAL的数量。是的，这是正确的 - 较大的检查点距离将导致更少的WAL。

这样做的原因很简单。每当一个区块在检查点之后第一次被触及，它就必须被完全发送到WAL。如果区块被更频繁的改变，只有改变的部分才会被送到日志中。较大的距离基本上会导致较少的全页写入，这反过来又减少了首先创建的WAL的数量。这种差异可能是相当大的，正如在我的一篇博文中所看到的：https://www.postgresql-support.com/checkpoint-distance-and-amount-of-wal/。

PostgreSQL还允许我们配置检查点是否应该短而密集，或者是否应该分散在较长的时间内。默认值是0.5，这意味着检查点的方式应该是在当前检查点和下一个检查点之间，进程已经完成一半。下面的列表显示了checkpoint_completion_target。

```
#checkpoint_completion_target = 0.5
```

增加这个值基本上意味着检查点被拉长，强度降低。在许多情况下，一个较高的值已被证明有利于平缓由密集检查点引起的I/O峰值。

然而，性能并不是唯一重要的话题。让我们也来看看日志归档和恢复的问题。

2.事务日志的存档和恢复

在我们简单介绍了事务日志的总体情况后，现在是时候关注事务日志的归档过程了。正如我们已经看到的，事务日志包含了对存储系统所做的二进制变化的序列。那么，为什么不使用它来复制数据库实例，并做很多其他很酷的事情，比如归档?

2.1 配置归档

在本章中，我们要做的第一件事是创建一个配置来执行标准的时间点恢复（PITR）。与普通转储相比，使用PITR有几个优点。

- 我们将损失更少的数据，因为我们可以将数据恢复到某个时间点，而不是仅仅恢复到固定的备份点。
- 恢复的速度会更快，因为索引不需要从头开始创建。它们只是被复制过来，并且可以随时使用。

PITR的配置很简单。只需在postgresql.conf文件中做一些修改，如下表中所示。

```
wal_level = replica # used to be "hot_standby" in older versions
max_wal_senders = 10 # at least 2, better at least 2
```

wal_level变量表示服务器应该产生足够的事务日志，以允许进行PITR。如果wal_level变量被设置为最小值（这是到PostgreSQL 9.6为止的默认值），事务日志将只包含足够的信息来恢复单节点设置--它不够丰富，无法处理复制。在PostgreSQL 10.0中，默认值已经正确，不再需要改变大多数设置。

max_wal_senders变量将允许我们从服务器上流传WAL。它将允许我们使用pg_basebackup来创建一个初始备份，而不是传统的基于文件的复制。这里的好处是pg_basebackup更容易使用。同样，10.0中的默认值已经被改变，对于90%的设置，不需要改变。

WAL流背后的想法是，创建的事务日志被复制到一个安全的地方进行存储。基本上，有两种传输WAL的手段。

- 使用pg_receivewal（到9.6为止，这被称为pg_receivexlog）
- 使用文件系统作为存档手段

在本节中，我们将看看如何设置第二个选项。在正常的操作中，PostgreSQL会不断向这些WAL文件写东西。当我们在postgresql.conf文件中设置archive_mode = on时，PostgreSQL将为每一个文件调用archive_command变量。

一个配置可能看起来如下。首先，可以创建一个存储这些交易日志文件的目录。

```
mkdir /archive
chown postgres.postgres archive
```

可以在postgresql.conf文件中修改以下条目。

```
archive_mode = on
archive_command = 'cp %p /archive/%f'
```

重启就可以实现归档，但让我们先配置pg_hba.conf文件，把停机时间降到绝对最低。

> 注意，我们可以把任何命令放入archive_command变量中。

许多人使用rsync、scp和其他方式将他们的WAL文件传送到一个安全的地方。如果我们的脚本返回0，PostgreSQL会认为该文件已经被归档。如果返回的是其他信息，PostgreSQL将尝试再次归档该文件。这是必要的，因为数据库引擎必须确保没有文件丢失。为了执行恢复过程，我们必须要有每一个文件可用；不允许有一个文件丢失。在下一步，我们将调整pg_hba.conf文件中的配置

2.2 配置 pg_hba.conf 文件

现在postgresql.conf文件已经配置成功，有必要对pg_hba.conf文件进行流式配置。注意，只有当我们计划使用pg_basebackup时才有必要这样做，它是创建基础备份的最先进的工具。

基本上，我们在pg_hba.conf文件中的选项与我们在第8章管理PostgreSQL安全中已经看到的选项相同。只有一个主要问题需要记住，可以借助下面的代码来理解。

```
# Allow replication connections from localhost, by a user with the
# replication privilege.
local replication postgres trust
host replication postgres 127.0.0.1/32 trust
host replication postgres ::1/128 trust
```

我们可以定义标准的pg_hba.conf文件规则。重要的是，第二列说的是复制。普通的规则是不够的--添加明确的复制权限真的很重要。另外，请记住，我们不一定要以超级用户的身份做这件事。我们可以创建一个特定的用户，只允许他进行登录和复制

同样，PostgreSQL 10及以后的版本已经按照我们在本节中所概述的方式进行了配置。当开箱即用的远程IP必须被添加到pg_hba.conf中时，本地复制就可以工作了

现在pg_hba.conf文件已经被正确配置，可以重新启动PostgreSQL。

2.3 创建基础备份

在教会PostgreSQL如何归档这些WAL文件之后，是时候创建第一个备份了。我们的想法是要有一个备份，并根据该备份重放WAL文件，以达到任何时间点。

为了创建一个初始备份，我们可以求助于pg_basebackup，它是一个用于执行备份的命令行工具。让我们调用pg_basebackup，看看它是如何工作的。

```
pg_basebackup -D /some_target_dir
 -h localhost
 --checkpoint=fast
 --wal-method=stream
```

如我们所见，我们将在这里使用四个参数：

- -D：我们要把基本备份放在哪里？PostgreSQL需要一个空目录。在备份结束时，我们将看到服务器的数据目录（目标）的副本。
- -h: 这表示主服务器（源）的IP地址或名称。 这是你要备份的服务器。
- --checkpoint=fast。通常情况下，pg_basebackup会等待主服务器创建一个检查点。这样做的原因是，重放过程必须从某个地方开始。一个检查点可以确保数据已经写到某一点，所以PostgreSQL可以安全地跳到那里，开始重放过程。基本上，不使用-- checkpoint=fast 参数也可以做到这一点。然而，在这种情况下，pg_basebackup可能需要一段时间才能开始复制数据。检查点的间隔可以达到1小时，这可能会不必要地拖延我们的备份。
- --wal-method=stream。默认情况下，pg_basebackup会连接到主服务器并开始复制文件过来。现在，请记住，这些文件在复制的过程中会被修改。因此，到达备份的数据是不一致的。这种不一致可以在恢复过程中使用WAL进行修复。然而，备份本身是不一致的。通过添加-wal-method=stream参数，可以创建一个独立的备份；它可以直接启动，而不需要重放事务日志。如果我们只想克隆一个实例而不使用PITR，这是一个不错的方法。幸运的是，-wal-method=stream实际上在PostgreSQL 10.0或更高版本中已经是默认的了。然而，在9.6或更早的版本中，建议使用其前身，名为-xlogmethod=stream。简而言之：在PostgreSQL 13.0中不需要再明确地设置这个了。

现在让我们看一下带宽管理。

2.4 减少备份带宽

当pg_basebackup启动时，它试图尽快完成其工作。如果我们有一个良好的网络连接，pg_basebackup肯定能在一秒钟内从远程服务器上获取数百兆字节的数据。如果我们的服务器有一个薄弱的I/O系统，这可能意味着pg_basebackup可以轻易地吸走所有的资源，而终端用户可能会因为他们的I/O请求太慢而遭遇糟糕的性能

为了控制最大传输速率，pg_basebackup提供了以下内容。

```
-r, --max-rate=RATE
 maximum transfer rate to transfer data directory
 (in kB/s, or use suffix "k" or "M")
```

当我们创建一个基本的备份时，我们需要确保主站的磁盘系统能够真正承受住负荷。因此，调整我们的传输速率可以有很大的意义。

2.5 映射表空间

通常，如果我们在目标系统上使用相同的文件系统布局，就可以直接调用pg_basebackup。如果不是这种情况，pg_basebackup允许你将主系统的文件系统布局映射到所需的布局上。-T选项允许我们进行映射。

```
-T, --tablespace-mapping=OLDDIR=NEWDIR
 relocate tablespace in OLDDIR to NEWDIR
```

如果你的系统很小，把所有东西都放在一个表空间里可能是个好主意。

如果I/O不是问题（也许是因为你只管理几千兆字节的数据），这一点是成立的。

2.6 使用不同的格式

pg_basebackup命令行工具可以创建各种格式。默认情况下，它将把数据放在一个空目录中。基本上，它将连接到源服务器并通过网络连接创建一个.tar文件，然后把数据放到所需的目录中。

这种方法的麻烦在于pg_basebackup会创建很多文件，如果我们想把备份转移到外部备份解决方案（如Tivoli Storage Manager），这并不适合。下面的列表显示了pg_basebackup支持的有效输出格式。

```
-F, --format=p|t output format (plain (default), tar)
```

要创建一个单一的文件，我们可以使用-F=t选项。默认情况下，它将创建一个名为base.tar的文件，然后可以更容易地管理它。当然，缺点是在执行PITR之前，我们必须再次对文件进行解包。

2.7 测试事务日志归档

在我们深入研究实际的重放过程之前，通过使用简单的ls命令，实际检查归档情况，以确保其工作完美，符合预期，如以下代码所示。

```
[hs@zenbook archive]$ ls -l
total 229384
-rw------- 1 hs staff 16777216 Oct 2 12:38 000000010000000000000007
-rw------- 1 hs staff 339 Oct 2 12:38
000000010000000000000007.00000188.backup
-rw------- 1 hs staff 16777216 Oct 2 12:38 000000010000000000000008
-rw------- 1 hs staff 16777216 Oct 2 12:31 000000010000000000000009
-rw------- 1 hs staff 16777216 Oct 2 12:31 00000001000000000000000A
-rw------- 1 hs staff 16777216 Oct 2 12:31 00000001000000000000000B
-rw------- 1 hs staff 16777216 Oct 2 12:38 00000001000000000000000C
-rw------- 1 hs staff 16777216 Oct 2 12:38 00000001000000000000000D
drwx------ 4 hs staff 136 Oct 2 12:38 archive_status
```

一旦数据库中出现任何严重的活动，WAL文件就应该被送到归档中。

除了仅检查文件之外，以下视图也很有用：

```
test=# \d pg_stat_archiver
 View "pg_catalog.pg_stat_archiver"
 Column | Type | Modifiers
--------------------+---------------------------+-----------
 archived_count | bigint |
 last_archived_wal | text |
 last_archived_time | timestamp with time zone |
 failed_count | bigint |
 last_failed_wal | text |
 last_failed_time | timestamp with time zone |
 stats_reset | timestamp with time zone |
```

pg_stat_archiver 系统视图对于确定归档是否因任何原因停止以及何时停止非常有用。它将告诉我们已经归档的文件的数量（archived_count）。我们还可以看到哪个文件是最后一个，以及事件发生的时间。最后，pg_stat_archiver系统视图可以告诉我们什么时候归档出了问题，这是至关重要的信息。 不幸的是，表中没有显示错误代码或信息，但由于archive_command可以是一个任意的命令，所以很容易记录下

在存档中还有一件事要看。正如我们前面所描述的，检查那些文件是否真的被归档是很重要的。但还有一点。 当pg_basebackup命令行工具被调用时，我们会在WAL文件流中看到一个.backup文件。它很小，只包含关于基础备份本身的一些信息--它纯粹是信息性的，重放过程不需要。然而，它给了我们一些重要的线索。当我们以后开始重放事务日志时，我们可以删除所有比.backup文件更早的WAL文件。在这种情况下，我们的备份文件被称为000000001000000000000007.00000188.backup。这意味着重放过程在文件...0007内的某个地方开始（在...188位置）。这也意味着我们可以删除所有比...0007更早

的文件。旧的WAL文件将不再需要用于恢复。请记住，我们可以保留不止一个备份，所以我只指当前的备份。

现在，归档工作已经完成，我们可以把注意力转向重放过程。

2.8 重放事务日志

让我们总结一下到目前为止的过程。我们调整了postgresql.conf文件（wal_level、max_wal_senders、archive_mode和archive_command），我们在pg_hba.conf文件中允许使用pg_basebackup命令。然后，数据库被重新启动，并成功产生了一个基础备份。

请记住，基础备份只能在数据库完全运行的情况下发生--只需要短暂的重启就可以改变max_wal_sender和wal_level变量。

现在，系统已经正常工作，我们可能会面临崩溃，我们要从中恢复。因此，我们可以执行PITR，尽可能多地恢复数据。我们要做的第一件事是采取基础备份，并把它放在理想的位置。

> 保存旧的数据库集群可能是一个好主意。即使它已经坏了，我们的PostgreSQL支持公司可能需要它来追踪崩溃的原因。你仍然可以在以后删除它，一旦你让一切重新运行起来。

鉴于前面的文件系统布局，我们可能想做如下事情。

```
cd /some_target_dir
cp -Rv * /data
```

我们假设新的数据库服务器将位于/data目录下。在你复制基础备份之前，请确保该目录是空的。

在PostgreSQL 13中，有些事情发生了变化：在旧版本中，我们必须配置recovery.conf来控制副本或PITR的一般行为。所有控制这些东西的配置设置都被移到了主配置文件postgresql.conf中。如果你正在运行旧的设置，那么现在是时候改用新的界面了，以确保你的自动化不会被破坏。

那么，让我们看看如何配置重放过程。尝试将restore_command和recovery_target_time放入postgresql.conf中。

```
restore_command = 'cp /archive/%f %p'
recovery_target_time = '2020-10-02 12:42:00'
```

在修复了postgresql.conf文件后，我们可以简单地启动我们的服务器。输出可能看起来如下。

```
waiting for server to start....
2020-10-02 12:42:10.085 CEST [52779] LOG: starting PostgreSQL 13.0 on x86_64-
apple-darwin17.7.0,
 compiled by Apple LLVM version 10.0.0 (clang-1000.10.44.4), 64-bit
2020-10-02 12:42:10.092 CEST [52779] LOG: listening on IPv6 address "::1", port
5432
2020-10-02 12:42:10.092 CEST [52779] LOG: listening on IPv6 address
"fe80::1%lo0",
port 5432
2020-10-02 12:42:10.092 CEST [52779] LOG: listening on IPv4 address "127.0.0.1",
port 5432
2020-10-02 12:42:10.093 CEST [52779] LOG: listening on Unix socket
"/tmp/.s.PGSQL.5432"
2020-10-02 12:42:10.099 CEST [52780] LOG: database system was interrupted; last
known up at 2020-10-02 12:38:25 CEST
cp: /tmp/archive/00000002.history: No such file or directory
2020-10-02 12:42:10.149 CEST [52780] LOG: entering standby mode
2020-10-02 12:42:10.608 CEST [52780] LOG: restored log file
```

```
"000000010000000000000007" from archive
2020-10-02 12:42:10.736 CEST [52780] LOG: redo starts at 0/7000188
2020-10-02 12:42:10.737 CEST [52780] LOG: consistent recovery state reached at
0/7000260
2020-10-02 12:42:10.737 CEST [52779] LOG: database system is ready to accept
read
only connections
 done
server started
2020-10-02 12:42:11.164 CEST [52780] LOG: restored log file
"000000010000000000000008" from archive
cp: /tmp/archive/000000010000000000000009: No such file or directory
2020-10-02 12:42:11.292 CEST [52788] LOG: started streaming WAL from primary at
0/9000000 on timeline 1
```

当服务器启动时，有几条信息需要寻找，以确保我们的恢复工作完美。达到一致的恢复状态是最重要的一条信息。一旦你达到了这一点，你就可以确定你的数据库是一致的，没有被破坏。根据你所选择的时间戳，你可能在最后丢失了一些事务（如果需要的话），但总的来说，你的数据库将是一致的（没有违反键等）。

如果你使用了一个表示未来某个时间点的时间戳，PostgreSQL会抱怨说它找不到下一个WAL文件，并终止重放过程。如果你使用的时间戳是在基础备份结束后到崩溃前的某个地方，你当然不会看到这样的消息。

一旦这个过程完成，服务器将成功启动。

2.9 找到正确的时间戳

到目前为止，我们的进展是假设我们知道我们想要恢复的时间戳，或者我们只是想重放整个事务日志以减少数据损失。然而，如果我们不想重放所有的东西呢？如果我们不知道要恢复到哪个时间点呢？在日常生活中，这其实是一个非常常见的场景。我们的一个开发人员在早上丢失了一些数据，而我们应该让事情恢复正常。问题是这样的：在早上的哪个时间点？一旦恢复结束，就不能轻易重启。一旦恢复完成，系统就会被推广，而一旦被推广，我们就不能继续重播WAL。

然而，我们可以做的是，在没有升级的情况下暂停恢复，检查数据库里面的内容，然后继续。

做到这一点很容易。我们首先要确定的是，在postgresql.conf文件中，hot_standby变量被设置为on。这将确保在数据库仍处于恢复模式时，它是可读的。然后，在postgresql.conf中设置以下变量。

```
recovery_target_action = 'pause'
```

有各种recovery_target_action设置。如果我们使用暂停，PostgreSQL将在所需的时间暂停，让我们检查已经重放的内容。我们可以调整我们想要的时间，重新启动，并再次尝试。 另外，我们也可以将该值设置为提升或关闭。

还有一种暂停事务日志回放的方法。基本上，它也可以在执行PITR时使用。但是，在大多数情况下，它是与流式复制一起使用的。下面是在WAL重放期间可以做的事情。

```
postgres=# \x
Expanded display is on.
postgres=# \df *pause*
List of functions
-[ RECORD 1 ]-------+-------------------------
Schema | pg_catalog
Name | pg_is_wal_replay_paused
```

```
Result data type | boolean
Argument data types |
Type | normal
-[ RECORD 2 ]-------+-------------------------
Schema | pg_catalog
Name | pg_wal_replay_pause
Result data type | void
Argument data types |
Type | normal
postgres=# \df *resume*
List of functions
-[ RECORD 1 ]-------+----------------------
Schema | pg_catalog
Name | pg_wal_replay_resume
Result data type | void
Argument data types |
Type | normal
```

我们可以调用SELECT pg_wal_replay_pause();命令来停止WAL重放，直到我们调用SELECT pg_wal_replay_resume();命令。

我们的想法是要弄清楚已经重放了多少WAL，并在必要时继续重放。然而，请记住这一点：一旦一个服务器被提升，我们就不能在没有进一步预防措施的情况下继续重放WAL。

正如我们已经看到的，要弄清楚我们需要恢复多远的时间，可能是相当棘手的。因此，PostgreSQL为我们提供了一些帮助。 考虑下面这个现实世界的例子：在午夜，我们运行一个夜间进程，在某个通常不为人知的点结束。我们的目标是精确地恢复到夜间进程结束的时间点。问题是这样的。我们如何知道这个过程何时结束？在大多数情况下，这很难搞清楚。那么，为什么不在事务日志中添加一个标记呢？这方面的代码如下。

```
postgres=# SELECT pg_create_restore_point('my_daily_process_ended');
 pg_create_restore_point
-------------------------
 1F/E574A7B8
(1 row)
```

如果我们的进程一结束就调用这个SQL语句，就可以通过在postgresql.conf文件中添加以下指令，使用事务日志中的这个标签，准确地恢复到这个时间点。

```
recovery_target_name = 'my_daily_process_ended'
```

通过使用这个设置而不是recovery_target_time，重放过程将把我们准确地传送到夜间进程的终点。

当然，我们也可以重放到某个事务ID。然而，在现实生活中，这被证明是困难的，因为管理员很少知道确切的事务ID，因此，这并没有什么实际价值。请记住，设置标记必须在恢复之前完成。这一点很重要。

2.10 清理事务日志存档

到目前为止，数据一直在被写入归档文件，没有注意再次清理归档文件以释放文件系统的空间。PostgreSQL不能为我们做这项工作，因为它不知道我们是否要再次使用存档。因此，我们要负责清理事务日志。当然，我们也可以使用备份工具--然而，重要的是要知道，PostgreSQL没有机会为我们做清理工作。

假设我们想清理一个不再需要的旧交易日志。也许我们想在周围保留几个基础备份，并清理所有不再需要的事务日志，以恢复其中一个备份。

在这种情况下，pg_archivecleanup命令行工具正是我们需要的。 我们可以简单地将归档目录和备份文件的名称传递给pg_archivecleanup命令，它将确保这些文件从磁盘上被删除。使用这个工具使我们的生活变得更容易，因为我们不必自己去计算要保留哪些交易日志文件。 下面是它的工作原理。

```
pg_archivecleanup removes older WAL files from PostgreSQL archives.
Usage:
 pg_archivecleanup [OPTION]... ARCHIVELOCATION OLDESTKEPTWALFILE
Options:
 -d generate debug output (verbose mode)
 -n dry run, show the names of the files that would be removed
 -V, --version output version information, then exit
 -x EXT clean up files if they have this extension
 -?, --help show this help, then exit
For use as archive_cleanup_command in postgresql.conf:
 archive_cleanup_command = 'pg_archivecleanup [OPTION]... ARCHIVELOCATION %r'
 e.g.
 archive_cleanup_command = 'pg_archivecleanup /mnt/server/archiverdir %r'
Or for use as a standalone archive cleaner:
 e.g.
 pg_archivecleanup /mnt/server/archiverdir
000000010000000000000010.00000020.backup
Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
```

这个工具可以轻松使用，并可在所有平台上使用。

现在我们已经看了一下事务日志存档和PITR，我们可以把注意力集中在当今PostgreSQL世界中最广泛使用的功能之一：流式复制。

3.设置异步复制

流式复制的想法很简单。在最初的基础备份之后，辅助系统可以连接到主系统，实时获取事务日志并加以应用。事务日志重放不再是一个单一的操作，而是一个连续的过程，只要集群存在，就应该一直运行下去。

3.1 执行基本设置

在这一节中，我们将学习如何快速、轻松地设置异步复制。我们的目标是建立一个由两个节点组成的系统。

基本上，大部分的工作已经为WAL归档完成了。 然而，为了便于理解，我们将看一下设置流媒体的整个过程，因为我们不能假设WAL运输真的已经按需要设置好了。

首先要做的是进入postgresql.conf文件，调整以下参数。

```
wal_level = replica
max_wal_senders = 10 # or whatever value >= 2
 # this is the default value already in more recent versions
hot_standby = on # in already a default setting
```

> 从PostgreSQL 10.0开始，其中一些已经是默认选项。

正如我们之前所做的，wal_level变量必须被调整，以确保PostgreSQL产生足够的事务日志来维持一个从属系统。 然后，我们必须配置max_wal_senders变量。当一个从属系统启动和运行时，或者当一个基础备份被创建时，一个WAL发送器进程将与客户端的WAL接收器进程对话。max_wal_senders的设置允许PostgreSQL创建足够的进程来服务这些客户端。

> 理论上，只需一个WAL发送程序就足够了。然而，这是很不方便的。一个使用-wal-method=stream参数的基础备份已经需要两个WAL发送器进程。如果你想同时运行一个slave和执行一个基础备份，已经有三个进程在使用了。因此，请确保你允许PostgreSQL创建足够的进程，以防止无意义的重新启动。

然后，还有hot_standby变量。基本上，主会忽略hot_standby变量，不把它考虑在内。它所做的只是在WAL重放期间使从节点可读。那么，我们为什么要关心呢？请记住，pg_basebackup命令将克隆整个服务器，包括其配置。这意味着，如果我们已经在主服务器上设置了这个值，那么当数据目录被克隆时，从服务器将自动得到它。

设置完postgresql.conf文件后，我们可以把注意力转向pg_hba.conf文件：只要通过添加规则，允许从机执行复制。基本上，这些规则与我们已经看到的PITR的规则是一样的。 然后，重新启动数据库服务器，就像对PITR所做的那样

现在，可以在slave上调用pg_basebackup命令了。在我们这样做之前，确保/target目录是空的。如果我们使用的是RPM包，确保你关闭了一个可能正在运行的实例，并清空目录（例如，/var/lib/pgsql/data）。下面的代码显示了如何使用pg_basebackup。

```
pg_basebackup -D /target
 -h master.example.com
 --checkpoint=fast
 --wal-method=stream -R
```

只要把/target目录替换成你想要的目标目录，把master.example.com替换成你的主的IP或DNS名称。--checkpoint=fast参数将触发一个即时检查点。然后，还有--wal-method=stream参数；它将打开两个流。一个将复制数据，而另一个将获取备份运行时创建的WAL数据。

最后，还有 -R 标志：

```
  -R, --write-recovery-conf # write configuration for replication
```

-R标志是一个非常好的功能。pg_basebackup命令可以自动创建从属配置。在旧版本中，它将在recovery.conf文件中添加各种条目。在PostgreSQL 12及以上版本中，它将自动对postgresql.conf进行修改。

```
standby_mode = on primary_conninfo = ' ...'
```

第一个设置说PostgreSQL应该一直重放WAL--如果整个事务日志已经被重放了，它应该等待新的WAL目录到来。第二个设置将告诉PostgreSQL主在哪里。这是一个正常的数据库连接。

> 从系统也可以连接到其他从系统来流转事务日志。通过简单地从一个从服务器创建基本备份，可以进行级联复制。所以，主服务器在这里真的意味着源服务器。

运行pg_basebackup命令后，可以启动服务。我们应该检查的第一件事是主是否显示了wal sender进程。

```
[hs@zenbook ~]$ ps ax | grep sender
17873 ? Ss 0:00 postgres: wal sender process
 ah ::1(57596) streaming 1F/E9000060
```

如果是，slave 也会显示 wal 接收进程：

```
17872 ? Ss 0:00 postgres: wal receiver process
 streaming 1F/E9000060
```

如果这些进程在那里，我们就已经在正确的轨道上了，而且复制正在按预期工作。现在双方都在相互交谈，WAL从主流向从。

3.2 提高安全性

到目前为止，我们已经看到，数据是以超级用户的身份流转的。然而，允许超级用户从远程站点访问并不是一个好主意。幸运的是，PostgreSQL允许我们创建一个只允许消费事务日志流的用户，但不能做其他事情。

创建一个仅用于流式传输的用户很容易。下面是它的工作原理：

```
test=# CREATE USER repl LOGIN REPLICATION;
CREATE ROLE
```

通过将复制分配给用户，有可能只用于流式传输--其他一切都被禁止了。

强烈建议不要使用你的超级用户账户来设置流式传输。只需将配置文件改为新创建的用户。 不暴露超级用户账户将极大地提高安全性，就像给复制用户一个密码一样。

3.3 停止和恢复复制

一旦设置了流式复制，它就能完美地工作，不需要管理员过多的干预。然而，在某些情况下，停止复制并在以后恢复复制可能是有意义的。为什么有人想这样做呢？

考虑以下用例：你负责一个主/从设置，它正在运行一个不合格的内容管理系统（CMS）或一些可疑的论坛软件。假设你想把你的应用程序从糟糕的CMS 1.0更新到糟糕的CMS 2.0。一些变化将在你的数据库中执行，这些变化将立即被复制到从属数据库中。如果升级过程中出现了错误怎么办？由由于流式传输，错误将立即复制到两个节点

为了避免即时复制，我们可以停止复制，然后根据需要恢复。 在我们的CMS更新案例中，我们可以简单地做以下事情。

1. 停止复制。
2. 在主服务器上执行应用程序更新。
3. 检查我们的应用程序是否仍然工作。如果是，恢复复制。如果不是，则故障转移到副本，该副本仍有旧的数据。

通过这种机制，我们可以保护我们的数据，因为我们可以回退到问题发生前的数据。在本章的后面，我们将学习如何将一个从属服务器提升为新的主服务器。

现在的主要问题是：我们如何才能停止复制？下面是它的工作方式。在备用机上执行以下一行。

```
test=# SELECT pg_wal_replay_pause();
```

这一行将停止复制。请注意，事务日志仍然会从主服务器流向从节点--只是重放过程被停止了。你的数据仍然受到保护，因为它被持久化在从服务器上。在服务器崩溃的情况下，没有数据会丢失。

请记住，重放过程必须在从机上停止。 否则，PostgreSQL将抛出一个错误。

```
ERROR: recovery is not in progress
HINT: Recovery control functions can only be executed during recovery.
```

一旦要恢复复制，在从属机构上需要有以下一行。

```
SELECT pg_wal_replay_resume();
```

PostgreSQL 将再次开始重放 WAL。

3.4 检查复制以确保可用性

每个管理员的核心工作之一是确保复制在任何时候都能保持正常运行。如果复制出现故障，如果主服务器崩溃，数据就有可能丢失。因此，保持对复制的关注是绝对必要的。

幸运的是，PostgreSQL提供了系统视图，使我们能够深入了解正在发生的事情。其中一个视图是pg_stat_replication。

```
test=# \d pg_stat_replication
 View "pg_catalog.pg_stat_replication"
 Column | Type | Collation | Nullable | Default
------------------+--------------------------+-----------+----------+---------
 pid | integer | | |
 usesysid | oid | | |
 usename | name | | |
 application_name | text | | |
 client_addr | inet | | |
 client_hostname | text | | |
 client_port | integer | | |
 backend_start | timestamp with time zone | | |
 backend_xmin | xid | | |
 state | text | | |
 sent_lsn | pg_lsn | | |
 write_lsn | pg_lsn | | |
 flush_lsn | pg_lsn | | |
 replay_lsn | pg_lsn | | |
 write_lag | interval | | |
 flush_lag | interval | | |
 replay_lag | interval | | |
 sync_priority | integer | | |
 sync_state | text | | |
 reply_time | timestamp with time zone | | |
```

pg_stat_replication视图将包含关于发送者的信息。我不想在这里使用主站这个词，因为从站可以连接到其他一些从站。有可能建立一个服务器树。在服务器树的情况下，主站将只拥有它直接连接的从站的信息。

在这个视图中，我们将看到的第一件事是WAL发送进程的进程ID。它可以帮助我们在出错的情况下识别该进程。 通常不会出现这种情况。然后，我们将看到从属进程用来连接到其发送服务器的用户名。client_*字段将表明从属进程的位置。我们将能够从这些字段中提取网络信息。backend_start字段显示了从属设备何时开始从我们的服务器进行流式传输。

然后，有一个神奇的backend_xmin字段。假设你正在运行一个主/从设置。可以告诉从属系统向主系统报告其事务ID，这背后的想法是延迟 master 上的清理，这样数据就不会从 slave 上运行的事务中获取

state字段通知我们关于服务器的状态。如果我们的系统没有问题，该字段将包含流。否则，需要仔细检查。

接下来的四个字段是真正重要的。sent_lsn字段，以前是send_location字段，表示有多少WAL已经到达对方，这意味着这些字段已经被WAL接收方接受。 我们可以用它来计算出有多少数据已经到达了从机。然后，是write_lsn字段，以前是write_location字段。 一旦WAL被接受，它就被传递给操作系统。Write_lsn字段将告诉我们，WAL的位置已经安全地传到了操作系统。 flush_lsn字段，也就是以前的flush_location字段，将知道数据库有多少WAL已经被刷到磁盘上了。

最后是replay_lsn，以前是replay_location字段。WAL已经到了备用机的磁盘上，但这并不意味着PostgreSQL已经重放或已经被最终用户看到。 假设复制被暂停了。数据仍然会流向备用机。 然而，它将在以后被应用。replay_lsn字段将告诉我们有多少数据是已经可见的。

在PostgreSQL 10.0中，更多的字段被添加到pg_stat_replication中；*_lag字段表示从属系统的延迟，并提供了一个方便的方法来查看从属系统的落后程度。

> 这些字段的间隔不同，这样我们可以更清楚地看到时间上的差异。

最后，PostgreSQL告诉我们复制是同步的还是异步的。

如果我们还在PostgreSQL 9.6上，我们可能会发现计算发送和接收服务器之间的字节差是很有用的。

9.6版的*_lag字段还不能做到这一点，所以有字节的差异会非常有利。下面是它的工作原理。

```
SELECT client_addr, pg_current_wal_location() - sent_location AS diff
FROM pg_stat_replication;
```

当在主服务器上运行时，pg_current_wal_location()函数返回当前的交易日志位置。PostgreSQL 9.6有一个特殊的数据类型用于事务日志位置，叫做pg_lsn。它有几个运算符，这里用将主服务器的WAL位置中减去从属服务器的WAL位置。因此，这里概述的视图返回两个服务器之间的差异，单位是字节（复制延迟）。

> 注意，这个语句只在PostgreSQL 10中起作用。在较早的版本中，这个函数曾经被称为pg_current_xlog_location()。

pg_stat_replication系统视图包含发送方的信息，而pg_stat_wal_receiver系统视图将为我们提供接收方的类似信息。

```
test=# \d pg_stat_wal_receiver
 View "pg_catalog.pg_stat_wal_receiver"
 Column | Type | Collation | Nullable | Default
----------------------+--------------------------+-----------+----------+-------
-
-
 pid | integer | | |
 status | text | | |
 receive_start_lsn | pg_lsn | | |
 receive_start_tli | integer | | |
 written_lsn | pg_lsn | | |
 flushed_lsn | pg_lsn | | |
```

```
received_tli | integer | | |
last_msg_send_time | timestamp with time zone | | |
last_msg_receipt_time | timestamp with time zone | | |
latest_end_lsn | pg_lsn | | |
latest_end_time | timestamp with time zone | | |
slot_name | text | | |
sender_host | text | | |
sender_port | integer | | |
conninfo | text | | |
```

在WAL接收器进程的进程ID之后，PostgreSQL会向你提供该进程的状态。然后，receive_start_lsn字段将告诉你WAL接收器开始的事务日志位置，而receive_start_tli字段将告知你WAL接收器启动时使用的时间轴。

written_lsn和flusged_lsn字段包含了关于WAL位置的信息，分别是已经写入和刷入磁盘的。 然后我们得到了一些关于时间的信息，以及插槽和连接的信息。

一般来说，许多人发现阅读pg_stat_replication系统视图比阅读pg_stat_wal_receiver视图更容易，而且大多数工具都是围绕pg_stat_replication视图建立的。

3.5 执行故障转移和了解时间表

一旦创建了主/从设置，它通常会在很长一段时间内完美无缺地工作。然而，一切都可能失败，因此，了解如何用一个备份系统取代一个失败的服务器是很重要的。

PostgreSQL使故障转移和推广变得很容易。基本上，我们所要做的就是使用pg_ctl参数来告诉一个副本来提升自己。

```
pg_ctl -D data_dir promote
```

服务器将断开自己与主站的连接，并立即执行升级。请记住，从服务器在升级时可能已经支持数千个只读连接。PostgreSQL的一个很好的特点是，在晋升过程中，所有打开的连接都会变成读/写连接--甚至不需要重新连接。

> 请注意，PostgreSQL 12也能够使用普通的SQL将数据库从从站提升到主站。只要使用SELECT pg_promote();。

当提升一个服务器时，PostgreSQL将递增时间线：如果你建立了一个全新的服务器，它将在时间线1。如果从该服务器克隆出一个从属服务器，它将和它的主服务器在同一时间线。所以，两个盒子都将在时间线1中。如果从机被提升为独立的主机，它将进入时间线2。

时间线对于PITR来说特别重要。假设我们在午夜时分创建一个基础备份。在上午12点，从属系统被提升。在下午3点，有东西崩溃了，我们想恢复到下午2点。我们将重放基础备份后创建的事务日志，并跟踪我们所需服务器的WAL流，因为这两个节点在上午12点开始出现了分歧。

时间线的变化也将在事务日志文件的名称中可见。下面是一个时间线1的WAL文件的例子。

```
000000010000000000000F5
```

如果时间线切换到 2，新文件名将如下所示：

```
000000020000000000000F5
```

正如你所看到的，来自不同时间线的WAL文件理论上可以存在于同一个存档目录中。

3.6 管理冲突

到目前为止，我们已经学到了很多关于复制的知识。然而，看一看复制冲突是很重要的。出现的主要问题是：冲突首先是如何发生的?

考虑以下示例：

Master	Slave
DROP TABLE tab;	BEGIN;
	SELECT ... FROM tab WHERE ...
	... running ...
	... conflict happens ...
	... transaction is allowed to continue for 30 seconds ...
	... conflict is resolved or ends before timeout ...

这里的问题是，主站不知道从站上有一个事务发生。因此，DROP TABLE命令不会阻塞，直到读取事务消失。如果这两个事务发生在同一个节点上，当然会是这样的情况。然而，我们在这里看到的是两个服务器。DROP表命令将正常执行，而杀死磁盘上那些数据文件的请求将通过事务日志到达从机。从机没有麻烦：如果表从磁盘上被删除，SELECT子句就必须死亡--如果从机在应用WAL之前等待SELECT子句的完成，它可能会无可奈何地落后。

理想的解决方案是可以使用配置变量来控制的折中方案。

```
max_standby_streaming_delay = 30s
 # max delay before canceling queries
 # when reading streaming WAL;
```

我们的想法是，在解决冲突之前，先等待30秒，在从机上杀死查询。根据我们的应用，我们可能想把这个变量改成一个更积极的设置。注意，30秒是针对整个复制流，而不是针对单个查询。可能因为其他查询已经等待了一段时间，所以单个查询被提前杀死。

虽然DROP TABLE命令显然是一种冲突，但有一些操作却不那么明显。下面是一个例子。

```
BEGIN;
...
DELETE FROM tab WHERE id < 10000;
COMMIT;
...
VACUUM tab;
```

再一次，让我们假设在slave上有一个长期运行的SELECT子句。在这里，DELETE子句显然不是问题，因为它只是将该行标记为已删除--它实际上并没有删除它。提交也不是一个问题，因为它只是将事务标记为完成。从物理上讲，该行仍然在那里。

当VACUUM等操作启动时，问题就开始了。它将破坏磁盘上的行。当然，这些变化会进入WAL，并最终到达从机，这时从机就会出现问题。

为了防止由标准OLTP工作负载引起的典型问题，PostgreSQL开发团队引入了一个配置变量。

```
hot_standby_feedback = off
 # send info from standby to prevent
 # query conflicts
```

如果这个设置是打开的，从属系统将定期向主系统发送最古老的事务ID。然后，VACUUM将知道在系统的某个地方有一个较长的事务在进行，并将清理年龄推迟到以后安全的时候再清理这些行。事实上，hot_standby_feedback参数引起的效果与主站的长事务相同。

我们可以看到，默认情况下，hot_standby_feedback参数是关闭的。为什么会这样呢？嗯，有一个很好的理由：如果它是关闭的，一个从站不会对主站产生实际影响。事务日志流不会消耗大量的CPU功率，使流式复制变得廉价和高效。然而，如果一个从站（甚至可能不在我们的控制之下）保持事务开放时间过长，我们的主站可能会因为清理过晚而受到表膨胀的影响。在默认设置中，这比减少冲突更不可取。

如果hot_standby_feedback = on，通常可以避免99%的OLTP相关冲突，如果你的交易时间超过几毫秒，这一点就特别重要。

3.7 使复制更可靠

在本章中，我们已经看到，设置复制是很容易的，不需要花费很多精力。然而，总有一些角落的情况会导致操作上的挑战。其中一个角落案例就是关于事务日志的保留。

考虑以下场景：

- 获取了一个基本的备份。
- 备份后，1小时内没有任何反应
- 从机启动。

请记住，主站并不太关心从站的存在。因此，从机启动所需的事务日志可能已经不存在于主机上了，因为它可能已经被检查点移除。问题是，需要重新同步才能启动从属系统。在一个多TB的数据库中，这显然是一个问题。

这个问题的一个潜在解决方案是使用wal_keep_segments设置。

```
wal_keep_segments = 0 # in logfile segments, 16MB each; 0 disables
```

默认情况下，PostgreSQL保留足够的事务日志，以便在意外的崩溃中存活，但不会多到哪里去。通过wal_keep_segments设置，我们可以告诉服务器保留更多的数据，这样即使从属系统落后，也能赶上。

重要的是要记住，服务器落后不仅是因为它们太慢或太忙--在许多情况下，延迟发生是因为网络太慢。假设你正在为一个1TB的表创建一个索引。 PostgreSQL会对数据进行排序，当索引真正建立时，也会被发送到交易日志中。试想一下，当几百兆的WAL通过一条也许只能处理1GB左右的线路发送时会发生什么。 许多千兆字节的数据的丢失可能就是这个后果，而且会在几秒钟内发生。因此，调整wal_keep_segments设置不应该关注典型的延迟，而应该关注管理员可以容忍的最高延迟（也许有一定的安全系数）。

为wal_keep_segments设置投资一个合理的高设置是非常有意义的，我建议确保周围总是有足够的数据。

解决事务日志耗尽问题的另一个办法是复制槽，这将在本章后面介绍。

4.升级到同步复制

到目前为止，我们已经对异步复制进行了合理的阐述。 然而，异步复制意味着允许从机上的提交在主机上的提交之后发生。如果主站崩溃了，即使复制正在发生，还没有到从站的数据也可能会丢失。

同步复制就是为了解决这个问题--如果PostgreSQL同步复制，一个提交必须由至少一个副本写入到磁盘上才能在主服务器上通过。因此，同步复制基本上可以大大降低数据丢失的几率。

在PostgreSQL中，配置同步复制很容易。只有两件事要做（按任何顺序）。

- 在主站的postgresql.conf文件中调整synchronous_standby_names设置。
- 在副本的配置文件中的primary_conninfo参数中增加一个application_name设置。

让我们从 master 上的 postgresql.conf 文件开始：

```
synchronous_standby_names = ''
 # standby servers that provide sync rep
 # number of sync standbys and comma-separated
 # list of application_name
 # from standby(s); '*' = all
```

如果我们输入'*'，所有的节点都将被视为同步候选。 然而，在现实生活中，更有可能的是，只有几个节点会被列出。下面是一个例子。

```
synchronous_standby_names = 'slave1, slave2, slave3'
```

在前面的案例中，我们得到了三个同步的候选人。现在，我们必须改变配置文件，加入application_name。

```
primary_conninfo = '... application_name=slave2'
```

复制体现在将作为slave2连接到master。master将检查它的配置，发现slave2是列表中第一个构成可行的slave。因此，PostgreSQL将确保只有在从属系统确认事务存在的情况下，master的提交才会成功。

现在，让我们假设slave2由于某种原因发生故障。PostgreSQL将尝试把另外两个节点中的一个变成同步的备用节点。问题是这样的：如果没有其他服务器怎么办？

在这种情况下，如果一个事务应该是同步的，PostgreSQL将永远等待提交。除非至少有两个可行的节点可用，否则PostgreSQL不会继续提交。记住，我们已经要求PostgreSQL在至少两个节点上存储数据--如果我们不能在任何时间点提供足够的主机，那就是我们的错。在现实中，这意味着同步复制最好在至少三个节点上实现--一个主节点和两个从节点--因为总是有可能有一个主机被丢失。

谈到主机故障，在这一点上有一件重要的事情需要注意--如果一个同步伙伴在提交过程中死亡，PostgreSQL将等待它回来。另外，同步提交也可能发生在其他潜在的同步伙伴身上。最终用户可能根本不会注意到同步伙伴已经改变。

在某些情况下，仅仅将数据存储在两个节点上可能是不够的：也许我们想进一步提高安全性，将数据存储在更多的节点上。为了达到这个目的，我们可以在PostgreSQL 9.6或更高版本中利用以下语法。

```
synchronous_standby_names =
 '4(slave1, slave2, slave3, slave4, slave5, slave6)'
```

当然，这也是有代价的--请记住，如果我们增加越来越多的同步复制，速度就会下降。世界上没有免费的午餐。PostgreSQL提供了几种方法来控制性能开销，我们将在下一节讨论。

在PostgreSQL 10.0中，甚至增加了更多的功能。让我们来看看协议的相关部分。

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

ANY和FIRST关键字已经被引入。FIRST允许你设置服务器的优先级，而ANY让PostgreSQL在提交同步事务时有更多的灵活性。

4.1 调整耐久度

在本章中，我们已经看到，数据要么是同步复制，要么是异步复制。然而，这并不是一个全局性的东西。为了确保良好的性能，PostgreSQL允许我们以一种非常灵活的方式来配置事物。我们有可能同步或异步地复制所有的东西，但是在很多情况下，我们可能想以更精细的方式来做事情。 这正是需要synchronous_commit设置的时候。

假设在postgresql.conf文件中已经配置了同步复制、application_name设置和synchronous_standby_names设置，synchronous_commit设置将提供以下选项

- off：这基本上是一个异步复制。WAL不会立即被刷新到主站的磁盘上，主站不会等待从站将所有内容写入磁盘。如果主站发生故障，一些数据可能会丢失（最多三次 - wal_writer_delay）。
- local：事务日志在主站提交时被刷新到磁盘。 然而，主站并不等待从站（异步复制）。
- remote_write：remote_write 的设置已经使 PostgreSQL 同步复制。然而，只有主站会将数据保存到磁盘。 对于从站来说，把数据发送到操作系统就足够了。我们的想法是不要等待第二次磁盘刷新来加速事情。两个存储系统完全在同一时间崩溃的可能性非常小。因此，数据丢失的风险接近于零。
- on：在这种情况下，如果主站和从站都成功地将事务刷到了磁盘上，那么事务就是OK的。除非数据被安全地存储在两个服务器上（或更多，取决于配置），否则应用程序不会收到提交。
- remote_apply：虽然在上确保了数据被安全地存储在两个节点上，但它并不能保证我们可以马上简单地进行负载平衡。数据被刷新在磁盘上的事实并不能保证用户已经可以看到数据。例如，如果有冲突，从机将停止事务重放--然而，在冲突期间，事务日志仍然被发送到从机，并被刷新到磁盘。简而言之，数据可能会被刷新在从机上，即使它对终端用户还不可见。

remote_apply选项解决了这个问题。它确保数据在副本上必须是可见的，这样下一个读取请求就可以在从属设备上安全地执行，从属设备已经可以看到对主设备所做的修改并将其暴露给最终用户。当然，remote_apply选项是复制数据的最慢方式，因为它要求从属设备已经将数据暴露给终端用户。

在PostgreSQL中，synchronous_commit参数不是一个全局值。它可以在不同层次上进行调整，就像许多其他设置一样。我们可能想做如下的事情。

```
test=# ALTER DATABASE test SET synchronous_commit TO off;
ALTER DATABASE
```

有时，只有一个数据库应该以某种方式进行复制。如果我们作为一个特定的用户连接，也可以只进行同步复制。最后，但同样重要的是，也可以告诉单个事务如何提交。通过实时调整synchronous_commit参数，我们甚至可以在每个事务层面上控制事情。

例如，考虑以下两种情况：

- 写入一个日志表，我们可能想使用异步提交，因为我们想快速完成。
- 存储信用卡付款，我们想保证安全，所以可能需要同步交易。

我们可以看到，非常相同的数据库可能有不同的要求，这取决于哪些数据被修改。因此，在事务层面改变数据是非常有用的，有助于提高速度。

5.利用复制槽

在介绍了同步复制和动态可调的持久性之后，我想重点介绍一个叫做复制槽的功能

复制槽的目的是什么？让我们考虑下面的例子。有一个主站和一个从站。在主服务器上，一个大型交易被执行，网络连接的速度不足以及时运送所有的数据。在某个时候，主站删除了它的事务日志（检查点）。如果从属系统落后太多，就需要重新同步。正如我们已经看到的，wal_keep_segments设置可以用来减少复制失败的风险。问题是：wal_keep_segments设置的最佳值是什么？当然，越多越好，但多少才是最好的？

复制槽将为我们解决这个问题：如果我们使用复制槽，主站只能在事务日志被所有的复制体消耗完后再回收它。这里的好处是，从机永远不会落后到需要重新同步的程度。

问题是，如果我们在没有告诉主服务器的情况下关闭了一个副本怎么办？主服务器会永远保留交易日志，主服务器上的磁盘最终会被填满，造成不必要的停机时间。

为了减少主站的这种风险，复制槽应该只与适当的监控和警报结合使用。只是有必要留意那些有可能导致问题或可能不再使用的开放复制槽。

在 PostgreSQL 中，有两种类型的复制槽：

- 物理复制槽
- 逻辑复制槽

物理复制槽可用于标准流复制。 它们将确保数据不被过早回收。逻辑复制槽做同样的事情。然而，它们被用于逻辑解码。逻辑解码背后的想法是让用户有机会附加到事务日志上，并用插件对其进行解码。因此，逻辑事务槽是数据库实例的某种-f tail。它允许用户提取对数据库所做的改变--因此也是对事务日志的改变--以任何格式和任何目的。在许多情况下，逻辑复制槽被用于逻辑复制

5.1 处理物理复制槽

为了利用复制槽，必须对postgresql.conf文件进行修改，具体如下。

```
wal_level = logical
max_replication_slots = 5 # or whatever number is needed
```

对于物理槽，逻辑槽是没有必要的 - 一个副本就足够了。然而，对于逻辑槽，我们需要一个更高的wal_level设置。那么，如果我们使用的是PostgreSQL 9.6或以下版本，就必须改变max_replication_slots的设置。PostgreSQL 10.0已经有一个改进的默认设置。基本上，我们只是输入一个符合我们目的的数字。 我的建议是增加一些备用槽，这样我们就可以很容易地附加更多的消费者，而不必沿途重启服务器

重新启动后，可以创建插槽：

```
test=# \x
Expanded display is on.
test=# \df *create*physicalslot
 List of functions
-[ RECORD 1 ]-------+------------------------------------------------------------
--
----------
Schema | pg_catalog
Name | pg_create_physical_replication_slot
Result data type | record
Argument data types | slot_name name, immediately_reserve boolean DEFAULT false,
 temporary boolean DEFAULT false, OUT slot_name name, OUT lsn
```

```
pg_lsn
Type | func
```

pg_create_physical_replication_slot函数是用来帮助我们创建槽的。它可以用两个参数之一来调用：如果只传递一个槽的名字，那么这个槽将在第一次使用时被激活。如果第二个参数是true，槽将立即开始保存交易日志。

```
test=# SELECT * FROM pg_create_physical_replication_slot('some_slot_name',
true);
 slot_name | lsn
-----------------+---------------
 some_slot_name | 0/EF8AD1D8
(1 row)
```

要想知道哪些插槽在主站上处于活动状态，可以考虑运行以下SQL语句。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-------+---------------
slot_name | some_slot_name
plugin |
slot_type | physical
datoid |
database |
temporary | f
...
```

视图告诉我们很多关于槽的信息。它包含关于正在使用的槽的类型、事务日志位置等信息。

为了利用这个槽，我们所要做的就是把它添加到配置文件中，如下所示。

```
primary_slot_name = 'some_slot_name'
```

一旦流媒体被重新启动，该插槽将被直接使用，并将保护复制。如果我们不想要我们的槽位了，我们可以轻松地放弃它。

```
test=# \df *drop*slot*
List of functions
-[ RECORD 1 ]-------+-------------------------
Schema | pg_catalog
Name | pg_drop_replication_slot
Result data type | void
Argument data types | name
Type | normal
```

当一个槽被放弃时，不再有逻辑槽和物理槽的区别。只需将槽的名称传递给函数并执行它。

> 当槽被放弃时，不允许任何人使用该槽。否则，PostgreSQL会有充分的理由出错。

5.2 处理逻辑复制槽

逻辑复制槽对逻辑复制至关重要。由于本章篇幅有限，很遗憾，我们不可能涵盖逻辑复制的所有方面。然而，我想概述一些基本概念，这些概念对逻辑解码至关重要，因此也对逻辑复制至关重要

如果我们想创建一个复制槽，这里是如何进行的。这里需要的函数需要两个参数：第一个参数将定义复制槽的名称，而第二个参数将携带用于解码交易日志的插件。它将决定PostgreSQL将使用何种格式来返回数据。让我们来看看如何制作一个复制槽。

```
test=# SELECT *
 FROM pg_create_logical_replication_slot('logical_slot', 'test_decoding');
 slot_name | lsn
---------------+---------------
 logical_slot | 0/EF8AD4B0
(1 row)
```

我们可以使用先前使用的相同命令来检查槽的存在。为了检查槽的真正作用，可以创建一个小测试。

```
test=# CREATE TABLE t_demo (id int, name text, payload text);
CREATE TABLE
test=#BEGIN;
BEGIN
test=# INSERT INTO t_demo
VALUES (1, 'hans', 'some data');
INSERT 0 1
test=# INSERT INTO t_demo VALUES (2, 'paul', 'some more data');
INSERT 0 1
test=# COMMIT;
COMMIT
test=# INSERT INTO t_demo VALUES (3, 'joe', 'less data');
INSERT 0 1
```

请注意，有两个事务被执行。现在可以从槽中提取对这些事务的修改。

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
 pg_logical_slot_get_changes
-------------------------------------------------------------------------
 (0/EF8AF5B0,606546,"BEGIN 606546")
 (0/EF8CCCA0,606546,"COMMIT 606546")
 (0/EF8CCCD8,606547,"BEGIN 606547")
 (0/EF8CCCD8,606547,"table public.t_demo: INSERT: id[integer]:1
 name[text]:'hans' payload[text]:'some data'")
 (0/EF8CCD60,606547,"table public.t_demo: INSERT: id[integer]:2
 name[text]:'paul' payload[text]:'some more data'")
 (0/EF8CCDE0,606547,"COMMIT 606547")
 (0/EF8CCE18,606548,"BEGIN 606548")
 (0/EF8CCE18,606548,"table public.t_demo: INSERT: id[integer]:3
 name[text]:'joe' payload[text]:'less data'")
 (0/EF8CCE98,606548,"COMMIT 606548")
(9 rows)
```

这里使用的格式取决于我们之前选择的输出插件。有各种用于PostgreSQL的输出插件，例如wal2json

> 如果使用默认值，逻辑流将包含真实值，而不仅仅是函数。逻辑流有最终进入底层表的数据。

另外，请记住，一旦槽被消耗，就不再返回数据。

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
 pg_logical_slot_get_changes
-----------------------------
(0 rows)
```

因此，第二次调用的结果集是空的。如果我们想重复地获取数据，PostgreSQL提供了pg_logical_slot_peek_changes函数。它的工作原理与pg_logical_slot_get_changes函数一样，但确保数据在槽中仍然可用。

当然，使用普通SQL并不是消费事务日志的唯一方法。还有一个叫做pg_recvlogical的命令行工具。它可以与在整个数据库实例上使用-f tail相比，并实时接收数据流

让我们使用以下命令启动 pg_recvlogical 工具：

```
[hs@zenbook ~]$ pg_recvlogical -S logical_slot -P test_decoding -d test -U
postgres --start -f -
```

在这种情况下，该工具连接到测试数据库并从logical_slot消耗数据。-f意味着数据流将被发送到stdout。让我们杀死一些数据。

```
test=# DELETE FROM t_demo WHERE id < random()*10;
DELETE 3
```

这些变化将进入事务日志。然而，在默认情况下，数据库只关心删除后的表是什么样子的。它知道哪些块必须被触及等等，但它不知道它以前是什么样子。

```
BEGIN 606549
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
COMMIT 606549
```

因此，输出结果是相当无意义的。为了解决这个问题，下面这一行就来了。

```
test=# ALTER TABLE t_demo REPLICA IDENTITY FULL;
ALTER TABLE
```

如果表被重新填充数据并再次被删除，事务日志流将看起来如下。

```
BEGIN 606558
table public.t_demo: DELETE: id[integer]:1 name[text]:'hans'
 payload[text]:'some data'
table public.t_demo: DELETE: id[integer]:2 name[text]:'paul'
 payload[text]:'some more data'
table public.t_demo: DELETE: id[integer]:3 name[text]:'joe'
 payload[text]:'less data'
COMMIT 606558
```

现在，所有的变化都在。接下来让我们看看逻辑复制槽的情况。

5.3 逻辑复制槽的用例

逻辑复制槽有多种用例。最简单的用例是如下所示。可以从服务器上获取所需格式的数据，并用于审计、调试，或简单地监控数据库实例。

下一个合乎逻辑的步骤是把这个变化流用于复制。双向复制（BDR）等解决方案完全基于逻辑解码，因为二进制级别的变化在多主复制中不起作用。

最后，有时需要在不停机的情况下进行升级。 记住，二进制事务日志流不能用于在不同版本的PostgreSQL之间进行复制。因此，未来的PostgreSQL版本将支持一个叫做pglogical的工具，它有助于在不停机的情况下进行升级。

在本节中，你学习了逻辑解码和其他一些基本技术。现在让我们看一下CREATE PUBLICATION和CREATE SUBSCRIPTION命令。

6.利用CREATE PUBLICATION和CREATE SUBSCRIPTION命令

对于10.0版本，PostgreSQL社区创建了两个新命令。CREATE PUBLICATION和CREATE SUBSCRIPTION。这些可以用于逻辑复制，这意味着你现在可以有选择地复制数据，实现接近零停机时间的升级。到目前为止，二进制复制和事务日志复制已经完全涵盖。然而，有时候，我们可能不想复制整个数据库实例--复制一两个表可能就够了。这正是逻辑复制的正确使用时机。

在开始之前，首先要做的是在postgresql.conf中把wal_level改为logical，如下所示，然后重新启动。

```
wal_level = logical
```

然后，我们可以创建一个简单的表：

```
test=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

同样的表布局也必须存在于第二个数据库中，以使其发挥作用。PostgreSQL不会自动为我们创建这些表。

```
test=# CREATE DATABASE repl;
CREATE DATABASE
```

创建数据库后，可以添加相同的表：

```
repl=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

这里的目标是将测试数据库中的t_test表的内容发布到其他地方。在这种情况下，它将被简单地复制到同一实例的数据库中。为了发布这些变化，PostgreSQL提供了CREATE PUBLICATION命令。

```
test=# \h CREATE PUBLICATION
Command: CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
 [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
 | FOR ALL TABLES ]
 [ WITH ( publication_parameter [= value] [, ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createpublication.html
```

语法其实很简单。我们所需要的只是一个名称和一个系统应该复制的所有表的列表。

```
test=# CREATE PUBLICATION pub1 FOR TABLE t_test;
CREATE PUBLICATION
```

现在，可以创建订阅。同样，语法非常简单明了：

```
test=# \h CREATE SUBSCRIPTION
Command: CREATE SUBSCRIPTION
Description: define a new subscription
Syntax:
CREATE SUBSCRIPTION subscription_name
 CONNECTION 'conninfo'
 PUBLICATION publication_name [, ...]
 [ WITH ( subscription_parameter [= value] [, ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createsubscription.html
```

基本上，直接创建一个订阅是绝对没有问题的。 但是，如果我们在同一个实例里面从测试数据库到复制数据库玩这个游戏，就必须手动创建使用中的复制槽。否则，CREATE SUBSCRIPTION将永远不会完成。

```
test=# SELECT pg_create_logical_replication_slot('sub1', 'pgoutput');
 pg_create_logical_replication_slot
------------------------------------
(sub1,0/27E2B2D0)
(1 row)
```

在这种情况下，在主数据库上创建的槽的名称叫做sub1。然后，我们需要连接到目标数据库并运行以下命令。

```
repl=# CREATE SUBSCRIPTION sub1
 CONNECTION 'host=localhost dbname=test user=postgres'
 PUBLICATION pub1
 WITH (create_slot = false);
CREATE SUBSCRIPTION
```

当然，我们必须调整我们数据库的CONNECTION参数。然后，PostgreSQL将同步数据，我们就完成了。

> 注意，create_slot = false的使用只是因为测试是在同一个数据库服务器实例内运行。如果我们碰巧使用不同的数据库，就不需要手动创建槽，也不需要create_slot = false。

7.总结

在这一章中，我们了解了PostgreSQL复制的最重要特性，比如流式复制和复制冲突。然后我们了解了PITR，以及复制槽。请注意，一本关于复制的书除非跨度在400页左右，否则永远不会完整，但我们已经学到了每个管理员应该知道的最重要的东西。不过，你已经学会了如何设置复制，了解了最重要的方面。

下一章，第11章，决定有用的扩展，是关于PostgreSQL的有用扩展。我们将了解那些提供更多功能并被业界广泛采用的扩展。

8.问题

- 逻辑复制的目的是什么?
- 同步复制的性能影响是什么?
- 为什么不总是使用同步复制?

这些问题的答案可以在 GitHub 仓库中找到 (https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition)