

## 一、分库分表，能不分就不分！

- 1、为什么要分库分表？
- 2、分库分表的优势
- 3、分库分表的挑战

## 二、应用中如何将单数据库升级为集群

- 1、搭建MySQL集群，实现服务和数据的高可用

### 1>搭建基础MySQL服务。

#### 2>搭建MySQL主从集群

- 1》配置master服务
- 2》配置slave从服务
- 3》主从集群测试
- 4》全库同步与部分同步
- 5》GTID同步集群

#### 3>集群扩容与MySQL数据迁移

#### 4>搭建半同步复制

- 1》理解半同步复制
- 2》搭建半同步复制集群

#### 5>扩展：MySQL高可用方案

- 1》MMM
- 2》MHA
- 3》MGR

- 2、应用层提供管理多个数据源的能力
- 3、数据库与应用结合，实现数据库集群管理
- 4、将多个数据源抽象成一个统一的数据源

## 四、章节总结-重要

# MySQL集群架构搭建以及多数据源管理实战

-- 楼兰

这个课程我将带你进行数据库的分库分表操作，这是互联网大型应用所需要面对的最核心的问题。因为数据往往是一个应用最核心的价值所在。但是，在最开始的时候，我会给你强调，在实际应用中，对于数据库，**能不分就不分!!!**这也应该是你未来准备对你的应用下手进行分库分表之前需要考量的宗旨。为什么一上来就会要给你们泼这样一盆冷水呢？这就需要你提前思考清楚，为什么要分库分表。

## 一、分库分表，能不分就不分！

### 1、为什么要分库分表？

数据库，应该是一个应用当中最为核心的价值所在，也是开发过程中必须熟练掌握的工具。之前我们就学习过很多对MySQL的调优。但是随着现在互联网应用越来越大，数据库会频繁的成为整个应用的性能瓶颈。我们经常使用的MySQL数据库，也就不断面临数据量太大、数据访问太频繁、数据读写速度太快等一系列的问题。而传统的这些调优方式，在真正面对海量数据冲击时，往往就会显得很无力。因此，现在互联网对于数据库的使用也越来越小心谨慎。例如添加Redis缓存、增加MQ进行流量削峰等。但是，数据库本身如果性能得不到提升，这就相当于水桶理论中的最短板。

要提升数据库的性能，最直接的思路，当然是对数据库本身进行优化。例如对MySQL进行调优，优化SQL逻辑，优化索引结构，甚至像阿里等互联网大厂一样，直接优化MySQL的源码。但是这种思路在面对互联网环境时，会有很多非常明显的弊端。

- 数据量和业务量快速增长，会带来性能瓶颈、服务宕机等很多问题。
- 单点部署的数据库无法保证服务高可用。
- 单点部署的数据库无法进行水平扩展，难以应对业务爆发式的增长。

这些问题背后的核心还是数据。数据库不同于上层的一些服务，他所管理的数据甚至比服务本身更重要。即要保证数据能够持续稳定的写入，又不能因为服务故障造成数据丢失。现在互联网上的大型应用，动辄几千万上亿的数据量，就算做好数据的压缩，随随便便也可以超过任何服务器的存储能力。并且，服务器单点部署，也无法真正解决把鸡蛋放在一个篮子里的问题。将数据放在同一个服务器上，如果服务器出现崩溃，就很难保证数据的安全性。这些都不是光靠优化MySQL产品，优化服务器配置能够解决的问题。

## 2、分库分表的优势

那么自然就需要换另外一种思路了。我们可以像微服务架构一样，来维护数据库的服务。把数据库从单体服务升级到数据库集群，这样才能真正全方位解放数据库的性能瓶颈，并且能够通过水平扩展的方式，灵活提升数据库的存储能力。这也就是我们常说的分库分表。通过分库分表可以给数据库带来很大的好处：

- 提高系统性能：分库分表可以将大型数据库分成多个小型数据库，每个小型数据库只需要处理部分数据，因此可以提高数据库的并发处理能力和查询性能。
- 提高系统可用性：分库分表可以将数据复制到多个数据库中，以提高数据的可用性和可靠性。如果一个数据库崩溃了，其他数据库可以接管其工作，以保持系统的正常运行。
- 提高系统可扩展性：分库分表可以使系统更容易扩展。当数据量增加时，只需要增加更多的数据库和表，而不是替换整个数据库，因此系统的可扩展性更高。
- 提高系统灵活性：分库分表可以根据数据的使用情况，对不同的数据库和表进行不同的优化。例如，可以将经常使用的数据存储性能更好的数据库中，或者将特定类型的数据存储在特定的表中，以提高系统的灵活性。
- 降低系统成本：分库分表可以使系统更加高效，因此可以降低系统的运营成本。此外，分库分表可以使用更便宜的硬件和存储设备，因为每个小型数据库和表需要的资源更少。

## 3、分库分表的挑战

可能你会说，分库分表嘛，也不是很难。一个库存不下，那就把数据拆分到多个库。一张表数据太多了，就把同一张表的数据拆分到多张。至于怎么做，也不难啊。要操作多个数据库，那么建立多个JDBC连接就行了。要写到多张表，修改下SQL语句就行了。

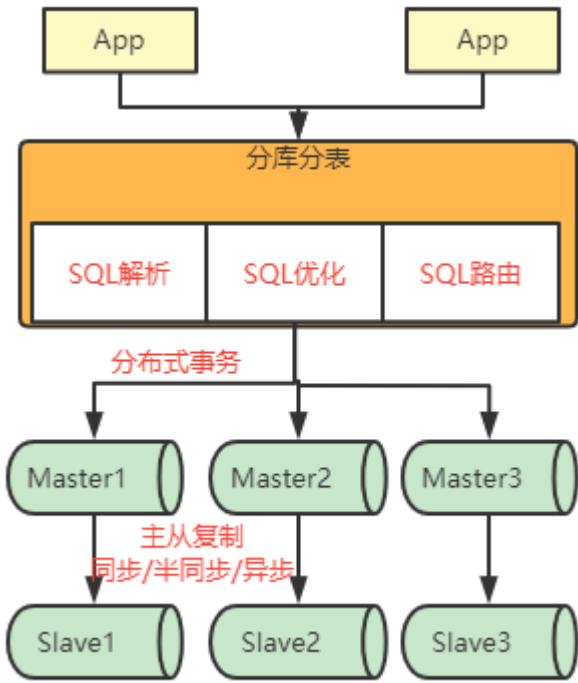
如果你也这么觉得，那么就大错特错了。**分库分表也并不是字面意义上的将数据分到多个库或者多个表这么简单，他需要的是一系列的分布式解决方案。**

因为数据的特殊性，造成数据库服务其实是几乎没有试错的成本的。在微服务阶段，从单机架构升级到微服务架构是很灵活的，中间很多细节步骤都可以随时做调整。比如对于微服务的接口限流功能，你并不需要一上来就用Sentinel这样复杂的流控工具。一开始不考虑性能，自己进行限流是很容易的事情。然后你可以慢慢尝试用Guava等框架提供的一些简单的流控工具进行零散的接口限流。直到整个应用的负载真正上来了之后，流控的要求更高更复杂了，再开始引入Sentinel，进行统一限流，这都没有问题。这种试错的过程其实是你能够真正用好一项技术的基础。

但是对于数据库就不一样了。当应用中用来存储数据的数据库，从一个单机的数据库服务升级到多个数据库组成的集群服务时，需要考虑的，除了分布式的各种让人摸不着边际的复杂问题外，还要考虑到一个更重要的因素，数据。**数据的安全性甚至比数据库服务本身更重要！**因此，如果你在一开始做分库分表时的方案不太成熟，对数据的规划不是很合理，那么这些问题大概率会随着数据永远沉淀下去，成为日后对分库分表方案进行调整时最大的拦路

虎。

所以在决定进行分库分表之前，一定需要提前对于所需要面对的各种问题进行考量。如果你没有考虑清楚数据要如何存储、计算、使用，或者你对于分库分表的各种问题都还没有进行过思考，那么千万不要在真实项目中贸然的进行分库分表。



分库分表，也称为Sharding。其实我觉得，Sharding应该比中文的分库分表更为贴切，他表示将数据拆分到不同的数据片中。由于数据往往是一个应用的基础，随着数据从单体服务拆分到多个数据分片，应用层面也需要面临很多新的问题。比如：

- 主键避重问题  
在分库分表环境中，由于表中数据同时存在不同数据库中，某个分区数据库生成的ID就无法保证全局唯一。因此需要单独设计全局主键，以避免跨库主键重复问题。
- 数据备份问题  
随着数据库由单机变为集群，整体服务的稳定性也会随之降低。如何保证集群在各个服务不稳定的情况下，依然保持整体服务稳定就是数据库集群需要面对的重要问题。而对于数据库，还需要对数据安全性做更多的考量。
- 数据迁移问题  
当数据库集群需要进行扩缩容时，集群中的数据也需要随着服务进行迁移。如何在不影响业务稳定性的情况下进行数据迁移也是数据库集群化后需要考虑的问题。
- 分布式事务问题  
原本单机数据库有很好的事务机制能够帮我们保证数据一致性。但是分库分表后，由于数据分布在不同库甚至不同服务器，不可避免会带来分布式事务问题。
- SQL路由问题  
数据被拆分到多个分散的数据库服务当中，每个数据库服务只能保存一部分的数据。这时，在执行SQL语句检索数据时，如何快速定位到目标数据所在的数据库服务，并将SQL语句转到对应的数据库服务中执行，也是提升检索效率必须要考虑的问题。
- 跨节点查询，归并问题

跨节点进行查询时，每个分散的数据库中只能查出一部分的数据，这时要对整体结果进行归并时，就会变得非常复杂。比如常见的limit、order by等操作。

在实际项目中，遇到的问题还会更多。从这里可以看出，Sharding其实是一个很复杂的问题，往往很难通过项目定制的方式整体解决。因此，大部分情况下，都是通过第三方的服务来解决Sharding的问题。比如像TiDB、ClickHouse、Hadoop这一类的NewSQL产品，大部分情况下是将数据问题整体封装到一起，从而提供Sharding方案。但是这些产品毕竟太重了。更灵活的方式还是使用传统数据库，通过软件层面来解决多个数据库之间的数据问题。这也诞生了很多的产品，比如早前的MyCat，还有后面我们要学习的ShardingSphere等。

另外，关于何时要开始考虑分库分表呢？当然是数据太大了，数据库服务器压力太大了就要进行分库分表。但是这其实是没有一个具体的标准的，需要根据项目情况进行灵活设计。业界目前唯一比较值得参考的详细标准，是阿里公开的开发手册中提到的，**建议预估三年内，单表数据超过500W，或者单表数据大小超过2G**，就需要考虑分库分表。

## 二、应用中如何将单数据库升级为集群

分库分表是一个很复杂的问题，因此在实际上手ShardingSphere这样的框架之前，我们其实有必要了解一下在实际开发中，我们如何将一个MySQL服务扩展成一个MySQL集群，这样，在后续面对ShardingSphere各种复杂的配置以及API时，才能弄明白到底在干些什么事情。

接下来会一步一步带你完成一个具有主从同步、读写分离这样典型数据库集群操作的应用，来理解一下分库分表要怎么做。

### 1、搭建MySQL集群，实现服务和数据的高可用

将数据库由单机升级为集群，这是分库分表必不可少的第一步。

#### 1>搭建基础MySQL服务。

以下准备两台服务器，用来搭建一个MySQL的服务集群。两台服务器均安装CentOS7操作系统。MySQL版本采用mysql-8.0.20版本。

两台服务器的IP分别为192.168.232.128和192.168.232.129。其中128服务器规划为MySQL主节点，129服务器规划为MySQL的从节点。

接下来需要在两台服务器上分别安装MySQL服务。

MySQL是很基础的服务，但是在Linux上搭建经常会出现各种各样的环境问题。如果大家在Linux上搭建MySQL有困难的话，可以改为用Windows安装，或者用Docker也行。甚至你可以使用宝塔这样的运维面板来协助搭建MySQL服务。宝塔官网地址：<https://www.bt.cn/new/index.html>

MySQL的安装有很多种方式，具体可以参考官网手册：<https://dev.mysql.com/doc/refman/8.0/en/binary-installation.html>。我们这里采用对系统环境依赖最低，出问题的可能性最小的tar包方式来安装。

上传mysql压缩包到worker2机器的root用户工作目录/root下，然后按照下面的指令，解压安装mysql。

```
groupadd mysql
useradd -r -g mysql -s /bin/false mysql #这里是创建一个mysql用户用于承载mysql服务，但是不需要登录权限。
tar -zxvf mysql-8.0.20-el7-x86_64.tar.gz #解压
ln -s mysql-8.0.20-el7-x86_64 mysql #建立软链接
cd mysql
mkdir mysql-files
chown mysql:mysql mysql-files
chmod 750 mysql-files
bin/mysqld --initialize --user=mysql #初始化mysql数据文件 注意点1
bin/mysql_ssl_rsa_setup
bin/mysqld_safe --user=mysql

cp support-files/mysql.server /etc/init.d/mysql.server
```

#### 注意点:

- 1、初始化过程中会初始化一些mysql的数据文件，经常会出现一些文件或者文件夹权限不足的问题。如果有文件权限不足的问题，需要根据他的报错信息，创建对应的文件或者文件夹，并配置对应的文件权限。
- 2、初始化过程如果正常完成，日志中会打印出一个root用户的默认密码。这个密码需要记录下来。

```
2020-12-10T06:05:28.948043Z 6 [Note] [MY-010454] [Server] A temporary password is
generated for root@localhost: P6kigsT6lg>=
```

## 2、启动MySQL服务

```
bin/mysqld --user=mysql &
```

#### 注意点:

- 1、这个启动过程会独占当前命令行窗口，如果要后台执行可以在后面添加一个 &。但是一般第一次启动mysql服务时，经常会出现一些错误，所以建议用独占窗口的模式跟踪下日志。

## 3、连接MySQL

MySQL服务启动完成后，默认是只能从本机登录，远程是无法访问的。所以需要用root用户登录下，配置远程访问的权限。

```
cd /root/mysql
bin/mysql -uroot -p #然后用之前记录的默认密码登录
```

#### 注意点:

- 1、如果有同学遇到 **ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)** 这个报错信息，可以参照下面的配置，修改下/etc/my.cnf配置文件，来配置下sock连接文件的地址。主要是下面client部分。

```
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

user=mysql
```

```
# Disabling symbolic-links is recommended to prevent assorted security risks
symbolic-links=0
# Settings user and group are ignored when systemd is used.
# If you need to run mysqld under a different user or group,
# customize your systemd unit file for mariadb according to the
# instructions in http://fedoraproject.org/wiki/Systemd

[mysqld_safe]
log-error=/var/log/mariadb/mariadb.log
pid-file=/var/run/mariadb/mariadb.pid

#
# include all files from the config directory
#
!includedir /etc/my.cnf.d

[client]
port=3306
socket=/var/lib/mysql/mysql.sock
```

登录进去后，需要配置远程登录权限：

```
alter user 'root'@'localhost' identified by '123456'; #修改root用户的密码
use mysql;
update user set host='%' where user='root';
flush privileges;
```

这样，Linux机器上的MySQL服务就搭建完成了。可以使用navicat等连接工具远程访问MySQL服务了。

接下来以相同方式在另外服务器上搭建MySQL即可。这里需要注意下的是，搭建主从集群的多个服务，有两个必要的条件。

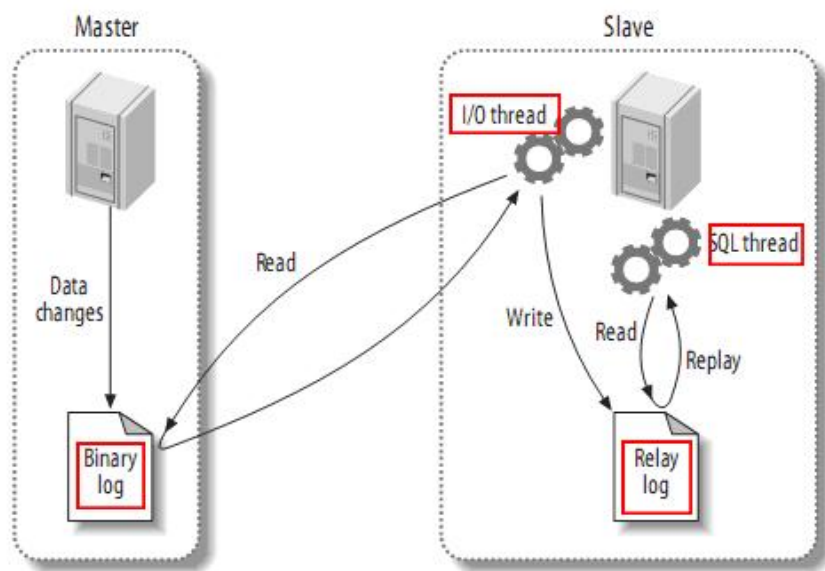
- 1、MySQL版本必须一致。
- 2、集群中各个服务器的时间需要同步。

## 2>搭建MySQL主从集群

既然要解决MySQL数据库的分布式集群化问题，那就不能不先了解MySQL自身提供的主从同步原理。这是构建MySQL集群的基础，也是后续进行分库分表的基础，更是MySQL进行生产环境部署的基础。

其实数据库的主从同步，就是为了要保证多个数据库之间的数据保持一致。最简单的方式就是使用数据库的导入导出工具，定时将主库的数据导出，再导入到从库当中。这是一种很常见，也很简单易行的数据库集群方式。也有很多的工具帮助我们来做这些事情。但是这种方式进行数据同步的实时性比较差。

而如果要保证数据能够实时同步，对于MySQL，通常就要用到他自身提供的一套通过Binlog日志在多个MySQL服务之间进行同步的集群方案。基于这种集群方案，一方面可以提高数据的安全性，另外也可以以此为基础，提供读写分离、故障转移等其他高级的功能。



即在主库上打开Binlog日志，记录对数据的每一步操作。然后在从库上打开RelayLog日志，用来记录跟主库一样的Binlog日志，并将RelayLog中的操作日志在自己数据库中进行重演。这样就能够更加实时的保证主库与从库的数据一致。

MySQL的Binlog默认是不打开的。

他的实现过程是在从库上启动一系列IO线程，负责与主库建立TCP连接，请求主库在写入Binlog日志时，也往从库传输一份。这时，主库上会有一个IO Dump线程，负责将Binlog日志通过这些TCP连接传输给从库的IO线程。而从库为了保证日志接收的稳定性，并不会立即重演Binlog数据操作，而是先将接收到的Binlog日志写入到自己的RelayLog日志当中。然后再异步的重演RelayLog中的数据操作。

MySQL的BinLog日志能够比较实时的记录主库上的所有操作，因此他也被很多其他工具用来实时监控MySQL的数据变化。例如Canal框架，可以模拟一个slave节点，同步MySQL的Binlog，然后将具体的数据操作按照定制的逻辑进行转发。例如转发到Redis实现缓存一致，转发到Kafka实现数据实时流转等。甚至像ClickHouse，还支持将自己模拟成一个MySQL的从节点，接收MySQL的Binlog日志，实时同步MySQL的数据。

接下来我们就在这两个MySQL服务的基础上，搭建一个主从集群。

## 1》配置master服务

首先，配置主节点的mysql配置文件：/etc/my.cnf(没有的话就手动创建一个)

这一步需要对master进行配置，主要是需要打开binlog日志，以及指定serverId。我们打开MySQL主服务的my.cnf文件，在文件中一行server-id以及一个关闭域名解析的配置。然后重启服务。

```
[mysqld]
server-id=47
#开启binlog
log_bin=master-bin
log_bin-index=master-bin.index
skip-name-resolve

# 设置连接端口
```



```
port=3306
# 设置mysql的安装目录
basedir=/usr/local/mysql
# 设置mysql数据库的数据的存放目录
datadir=/usr/local/mysql/mysql-files
# 允许最大连接数
max_connections=200
# 允许连接失败的次数。
max_connect_errors=10
# 服务端使用的字符集默认为UTF8
character-set-server=utf8
# 创建新表时将使用的默认存储引擎
default-storage-engine=INNODB
# 默认使用“mysql_native_password”插件认证
#mysql_native_password
default_authentication_plugin=mysql_native_password
```

配置说明：主要需要修改的是以下几个属性：

server-id：服务节点的唯一标识。需要给集群中的每个服务分配一个单独的ID。

log\_bin：打开Binlog日志记录，并指定文件名。

log\_bin-index：Binlog日志文件

重启MySQL服务， `service mysqld restart`

然后，我们需要给root用户分配一个replication slave的权限。

```
#登录主数据库
mysql -u root -p
GRANT REPLICATION SLAVE ON *.* TO 'root'@'%';
flush privileges;
#查看主节点同步状态：
show master status;
```

在实际生产环境中，通常不会直接使用root用户，而会创建一个拥有全部权限的用户来负责主从同步。

```
mysql> show master status;
```

File	Position	Binlog_Do_DB	Binlog_Ignore_DB	Executed_Gtid_Set
master-bin.000004	156			

```
1 row in set (0.00 sec)
```

这个指令结果中的File和Position记录的是当前日志的binlog文件以及文件中的索引。

而后面的Binlog\_Do\_DB和Binlog\_Ignore\_DB这两个字段是表示需要记录binlog文件的库以及不需要记录binlog文件的库。目前我们没有进行配置，就表示是针对全库记录日志。这两个字段如何进行配置，会在后面进行介绍。

开启binlog后，数据库中的所有操作都会被记录到datadir当中，以一组轮询文件的方式循环记录。而指令查到的File和Position就是当前日志的文件和位置。而在后面配置从服务时，就需要通过这个File和Position通知从服务从哪个地方开始记录binLog。



```

[root@worker1 mysql-files]# ll
总用量 179660
-rw-r----- 1 mysql mysql      56 12月 30 14:24 auto.cnf
-rw-r----- 1 mysql mysql    1676 12月 30 14:24 ca-key.pem
-rw-r--r-- 1 mysql mysql    1112 12月 30 14:24 ca.pem
-rw-r--r-- 1 mysql mysql    1112 12月 30 14:24 client-cert.pem
-rw-r----- 1 mysql mysql    1676 12月 30 14:24 client-key.pem
-rw-r----- 1 mysql mysql  196608 12月 30 14:33 #ib_16384_0.dblwr
-rw-r----- 1 mysql mysql  8585216 12月 30 14:24 #ib_16384_1.dblwr
-rw-r----- 1 mysql mysql     3768 12月 30 14:32 ib_buffer_pool
-rw-r----- 1 mysql mysql 12582912 12月 30 14:33 ibdata1
-rw-r----- 1 mysql mysql  50331648 12月 30 14:33 ib_logfile0
-rw-r----- 1 mysql mysql  50331648 12月 30 14:24 ib_logfile1
-rw-r----- 1 mysql mysql 12582912 12月 30 14:32 ibtmp1
drwxr-x--- 2 mysql mysql     187 12月 30 14:32 #innodb temp
-rw-r----- 1 mysql mysql     179 12月 30 14:32 master-bin.000001
-rw-r----- 1 mysql mysql    1829 12月 30 14:33 master-bin.000002
-rw-r----- 1 mysql mysql      40 12月 30 14:32 master-bin.index
drwxr-x--- 2 mysql mysql     143 12月 30 14:24 mysql
-rw-r----- 1 mysql mysql 28311552 12月 30 14:33 mysql.ibd
drwxr-x--- 2 mysql mysql     8192 12月 30 14:24 performance_schema
-rw-r----- 1 mysql mysql    1676 12月 30 14:24 private_key.pem
-rw-r--r-- 1 mysql mysql     452 12月 30 14:24 public_key.pem
-rw-r--r-- 1 mysql mysql    1112 12月 30 14:24 server-cert.pem
-rw-r----- 1 mysql mysql    1680 12月 30 14:24 server-key.pem
drwxr-x--- 2 mysql mysql      28 12月 30 14:24 sys
-rw-r----- 1 mysql mysql 10485760 12月 30 14:33 undo_001
-rw-r----- 1 mysql mysql 10485760 12月 30 14:33 undo_002
-rw-r----- 1 mysql mysql     2200 12月 30 14:32 worker1.err
-rw-r----- 1 mysql mysql        5 12月 30 14:32 worker1.pid
[root@worker1 mysql-files]# pwd
/usr/local/mysql/mysql-files

```

## 2》配置slave从服务

下一步，我们来配置从服务mysqls。我们打开mysqls的配置文件my.cnf，修改配置文件：

```

[mysqld]
#主库和从库需要不一致
server-id=48
#打开MySQL中继日志
relay-log-index=slave-relay-bin.index
relay-log=slave-relay-bin
#打开从服务二进制日志
log-bin=mysql-bin
#使得更新的数据写进二进制日志中
log-slave-updates=1
# 设置3306端口
port=3306
# 设置mysql的安装目录
basedir=/usr/local/mysql
# 设置mysql数据库的数据的存放目录
datadir=/usr/local/mysql/mysql-files
# 允许最大连接数
max_connections=200
# 允许连接失败的次数。
max_connect_errors=10
# 服务端使用的字符集默认为UTF8

```

```
character-set-server=utf8
# 创建新表时将使用的默认存储引擎
default-storage-engine=INNODB
# 默认使用“mysql_native_password”插件认证
#mysql_native_password
default_authentication_plugin=mysql_native_password
```

配置说明：主要需要关注的几个属性：

server-id：服务节点的唯一标识

relay-log：打开从服务的relay-log日志。

log-bin：打开从服务的bin-log日志记录。

然后我们启动mysqls的服务，并设置他的主节点同步状态。

```
#登录从服务
mysql -u root -p;
#设置同步主节点：
CHANGE MASTER TO
MASTER_HOST='192.168.232.128',
MASTER_PORT=3306,
MASTER_USER='root',
MASTER_PASSWORD='root',
MASTER_LOG_FILE='master-bin.000004',
MASTER_LOG_POS=156,
GET_MASTER_PUBLIC_KEY=1;
#开启slave
start slave;
#查看主从同步状态
show slave status;
或者用 show slave status \G; 这样查看比较简洁
```

注意，CHANGE MASTER指令中需要指定的MASTER\_LOG\_FILE和MASTER\_LOG\_POS必须与主服务中查到的保持一致。

并且后续如果要检查主从架构是否成功，也可以通过检查主服务与从服务之间的File和Position这两个属性是否一致来确定。

```
mysql> show slave status \G;
***** 1. row *****
      Slave_IO_State: Waiting for master to send event
      Master_Host: 127.0.0.1
      Master_User: root
      Master_Port: 3307
      Connect_Retry: 60
      Master_Log_File: master-bin.000004
      Read_Master_Log_Pos: 156
      Relay_Log_File: slave-relay-bin.000006
      Relay_Log_Pos: 373
      Relay_Master_Log_File: master-bin.000004
      Slave_IO_Running: Yes
      Slave_SQL_Running: Yes
      Replicate_Do_DB:
      Replicate_Ignore_DB:
      Replicate_Do_Table:
      Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
      Replicate_Wild_Ignore_Table:
      Last_Errno: 0
      Last_Error:
      Skip_Counter: 0
```

我们重点关注其中红色方框的两个属性，与主节点保持一致，就表示这个主从同步搭建是成功的。

从这个指令的结果能够看到，有很多Replicate开头的属性，这些属性指定了两个服务之间要同步哪些数据库、哪些表的配置。只是在我们这个示例中全都没有进行配置，就标识是全库进行同步。后面我们会补充如何配置需要同步的库和表。

### 3》主从集群测试

测试时，我们先用showdatabases，查看下两个MySQL服务中的数据库情况

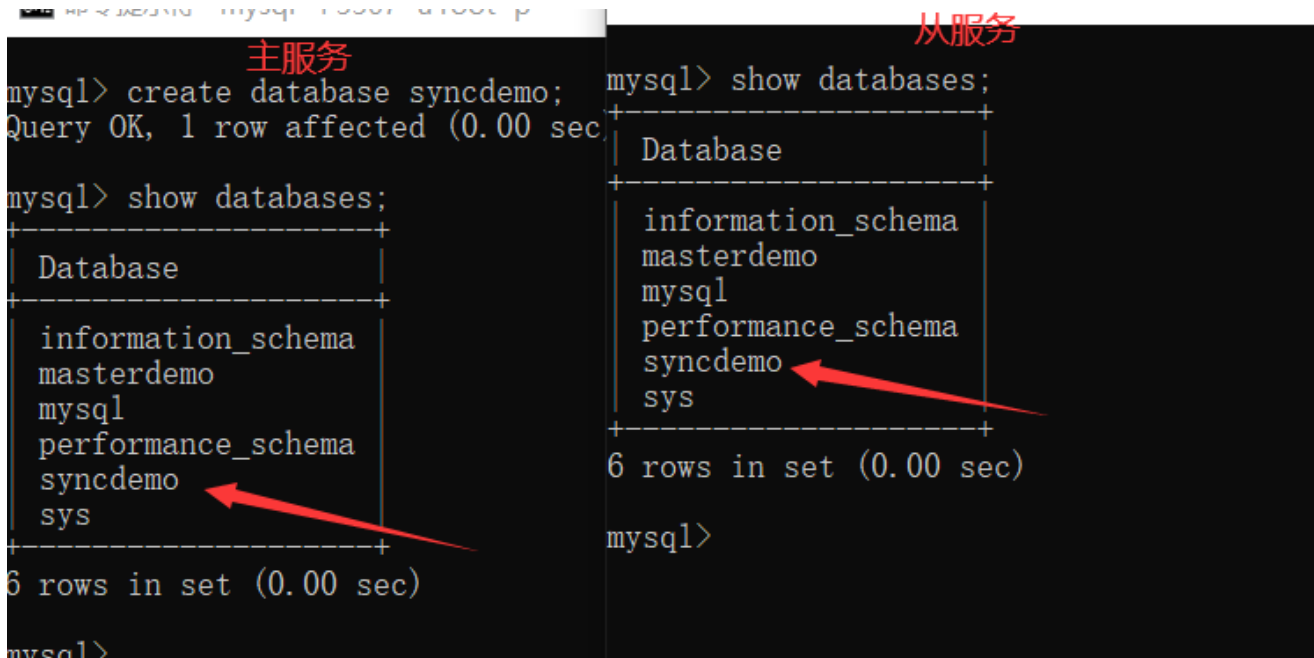
主服务	从服务
<pre>mysql&gt; show databases; +-----+   Database   +-----+   information_schema     masterdemo     mysql     performance_schema     sys   +-----+ 5 rows in set (0.00 sec)</pre>	<pre>mysql&gt; show databases; +-----+   Database   +-----+   information_schema     masterdemo     mysql     performance_schema     sys   +-----+ 5 rows in set (0.01 sec)</pre>

然后我们在主服务器上创建一个数据库

```
mysql> create database syncdemo;

Query OK, 1 row affected (0.00 sec)
```

然后我们再用show databases，来看下这个syncdemo的数据库是不是已经同步到了从服务。



The screenshot shows two MySQL terminal windows side-by-side. The left window is labeled '主服务' (Master Service) and the right window is labeled '从服务' (Slave Service). In the master window, the command 'create database syncdemo;' is executed, followed by 'show databases;', which lists 'information\_schema', 'masterdemo', 'mysql', 'performance\_schema', 'syncdemo', and 'sys'. A red arrow points to 'syncdemo'. In the slave window, the command 'show databases;' is executed, listing the same databases, with 'syncdemo' also highlighted by a red arrow. Both windows show '6 rows in set (0.00 sec)'.

```
mysql> create database syncdemo;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| masterdemo        |
| mysql              |
| performance_schema |
| syncdemo           |
| sys                 |
+-----+
6 rows in set (0.00 sec)

mysql>
```

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| masterdemo        |
| mysql              |
| performance_schema |
| syncdemo           |
| sys                 |
+-----+
6 rows in set (0.00 sec)

mysql>
```

接下来我们继续在syncdemo这个数据库中创建一个表，并插入一条数据。

```
mysql> use syncdemo;
Database changed
mysql> create table demoTable(id int not null);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into demoTable value(1);
Query OK, 1 row affected (0.01 sec)
```

然后我们也同样到主服务与从服务上都来查一下这个demoTable是否同步到了从服务。

**主服务**

```

mysql> use syncdemo;
Database changed
mysql> create table demoTable(id int);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into demoTable value (1);
Query OK, 1 row affected (0.01 sec)

mysql> show tables;
+-----+
| Tables_in_syncdemo |
+-----+
| demotable          |
+-----+
1 row in set (0.00 sec)

mysql> select * from demotable;
+----+
| id |
+----+
| 1  |
+----+
1 row in set (0.00 sec)

mysql>

```

**从服务**

```

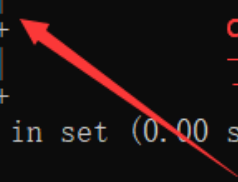
mysql> use syncdemo;
Database changed
mysql> show tables;
+-----+
| Tables_in_syncdemo |
+-----+
| demotable          |
+-----+
1 row in set (0.00 sec)

mysql> select * from demotable;
+----+
| id |
+----+
| 1  |
+----+
1 row in set (0.00 sec)

mysql>

```

主服务上创建的  
demoTable表已经同步到了从服务



从上面的实验过程看到，我们在主服务中进行的数据操作，就都已经同步到了从服务上。这样，我们一个主从集群就搭建完成了。

另外，这个主从架构是有可能失败的，如果在slave从服务上查看slave状态，发现Slave\_SQL\_Running=no，就表示主从同步失败了。这有可能是因为在从数据库上进行了写操作，与同步过来的SQL操作冲突了，也有可能是slave从服务重启后有事务回滚了。

如果是因为slave从服务事务回滚的原因，可以按照以下方式重启主从同步：

```

mysql> stop slave ;
mysql> set GLOBAL SQL_SLAVE_SKIP_COUNTER=1;
mysql> start slave ;

```

而另一种解决方式就是重新记录主节点的binlog文件消息

```

mysql> stop slave ;
mysql> change master to .....
mysql> start slave ;

```

但是这种方式要注意binlog的文件和位置，如果修改后和之前的同步接不上，那就会丢失部分数据。所以不太常用。

## 4》全库同步与部分同步

在完成这个基本的MySQL主从集群后，我们还可以进行后续的实验：

之前提到，我们目前配置的主从同步是针对全库配置的，而实际环境中，一般并不需要针对全库做备份，而只需要对一些特别重要的库或者表来进行同步。那如何针对库和表做同步配置呢？

首先在Master端：在my.cnf中，可以通过以下这些属性指定需要针对哪些库或者哪些表记录binlog

```
#需要同步的二进制数据库名
binlog-do-db=masterdemo
#只保留7天的二进制日志，以防磁盘被日志占满(可选)
expire-logs-days = 7
#不备份的数据库
binlog-ignore-db=information_schema
binlog-ignore-db=performance_schema
binlog-ignore-db=sys
```

然后在Slave端：在my.cnf中，需要配置备份库与主服务的库的对应关系。

```
#如果slave库名称与master库名相同，使用本配置
replicate-do-db = masterdemo
#如果master库名[mastdemo]与slave库名[mastdemo01]不同，使用以下配置[需要做映射]
replicate-rewrite-db = masterdemo -> masterdemo01
#如果不是要全部同步[默认全部同步]，则指定需要同步的表
replicate-wild-do-table=masterdemo01.t_dict
replicate-wild-do-table=masterdemo01.t_num
```

配置完成了之后，在show master status指令中，就可以看到Binlog\_Do\_DB和Binlog\_Ignore\_DB两个参数的作用了。

## 5》GTID同步集群

上面我们搭建的集群方式，是基于Binlog日志记录点的方式来搭建的，这也是最为传统的MySQL集群搭建方式。而在这个实验中，可以看到有一个Executed\_Grid\_Set列，暂时还没有用上。实际上，这就是另外一种搭建主从同步的方式，即GTID搭建方式。这种模式是从MySQL5.6版本引入的。

GTID的本质也是基于Binlog来实现主从同步，只是他会基于一个全局的事务ID来标识同步进度。GTID即全局事务ID，全局唯一并且趋势递增，他可以保证为每一个在主节点上提交的事务在复制集群中可以生成一个唯一的ID。

在基于GTID的复制中，首先从服务器会告诉主服务器已经在从服务器执行完了哪些事务的GTID值，然后主库会有把所有没有在从库上执行的事务，发送到从库上进行执行，并且使用GTID的复制可以保证同一个事务只在指定的从库上执行一次，这样可以避免由于偏移量的问题造成数据不一致。

他的搭建方式跟我们上面的主从架构整体搭建方式差不多。只是需要在my.cnf中修改一些配置。

在主节点上：

```
gtid_mode=on
enforce_gtid_consistency=on
log_bin=on
server_id=单独设置一个
binlog_format=row
```

在从节点上：

```
gtid_mode=on
enforce_gtid_consistency=on
log_slave_updates=1
server_id=单独设置一个
```

然后分别重启主服务和从服务，就可以开启GTID同步复制方式。

### 3>集群扩容与MySQL数据迁移

我们现在已经搭建成功了一主一从的MySQL集群架构，那要扩展到一主多从的集群架构，其实就比较简单了，只需要增加一个binlog复制就行了。

但是如果我们的集群是已经运行过一段时间，这时候如果要扩展新的从节点就有一个问题，之前的数据没办法从binlog来恢复了。这时候在扩展新的slave节点时，就需要增加一个数据复制的操作。

MySQL的数据备份恢复操作相对比较简单，可以通过SQL语句直接来完成。具体操作可以使用mysql的bin目录下的mysqldump工具。

```
mysqldump -u root -p --all-databases > backup.sql
#输入密码
```

通过这个指令，就可以将整个数据库的所有数据导出成backup.sql，然后把这个backup.sql分发到新的MySQL服务器上，并执行下面的指令将数据全部导入到新的MySQL服务中。

```
mysql -u root -p < backup.sql
#输入密码
```

这样新的MySQL服务就已经有了所有的历史数据，然后就可以再按照上面的步骤，配置Slave从服务的数据同步了。

### 4>搭建半同步复制

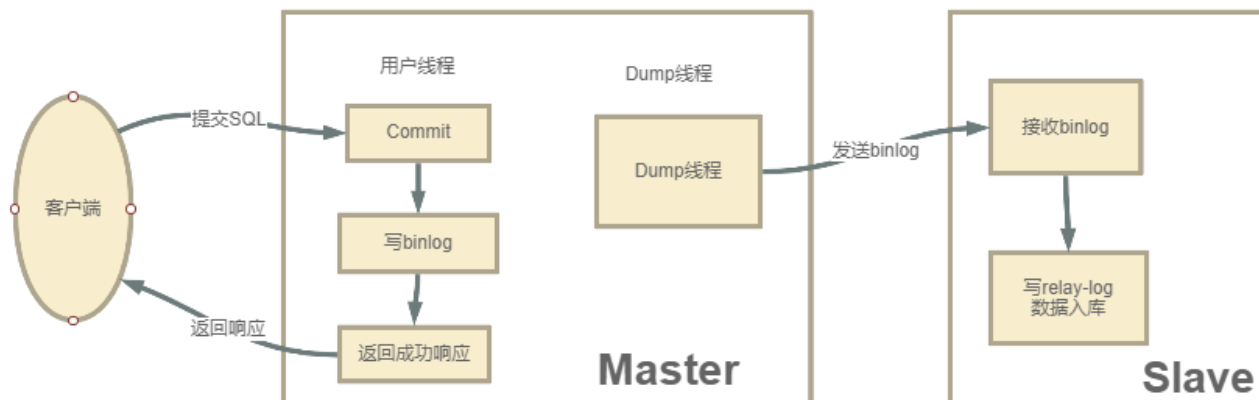
#### 1》理解半同步复制

到现在为止，我们已经可以搭建MySQL的主从集群，互主集群，但是我们这个集群有一个隐患，就是有可能会丢数据。这是为什么呢？这要从MySQL主从数据复制分析起。

MySQL主从集群默认采用的是一种异步复制的机制。主服务在执行用户提交的事务后，写入binlog日志，然后就给客户端返回一个成功的响应了。而binlog会由一个dump线程异步发送给Slave从服务。



## MySQL异步复制

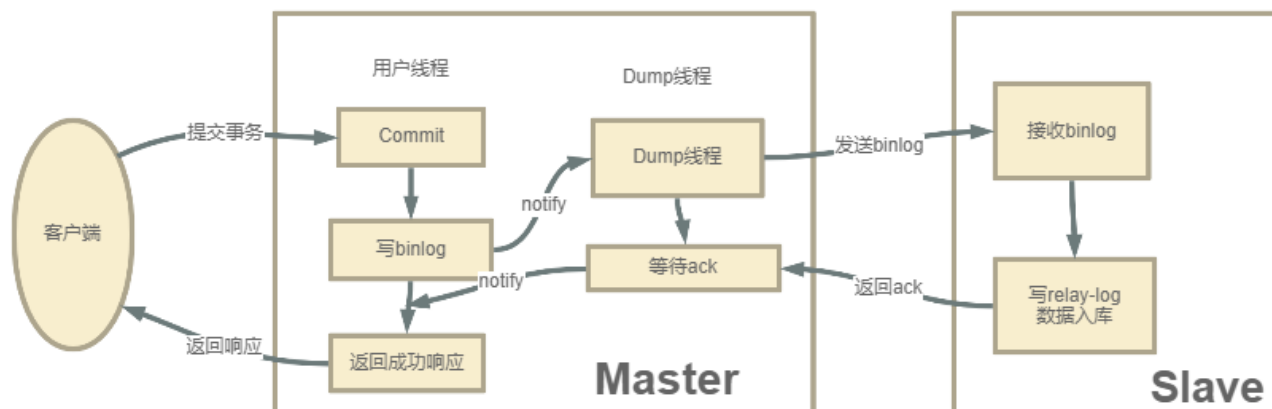


由于这个发送binlog的过程是异步的。主服务在向客户端反馈执行结果时，是不知道binlog是否同步成功了。这时候如果主服务宕机了，而从服务还没有备份到新执行的binlog，那就有可能会丢数据。

那怎么解决这个问题呢，这就要靠MySQL的半同步复制机制来保证数据安全。

半同步复制机制是一种介于异步复制和全同步复制之前的机制。主库在执行完客户端提交的事务后，并不是立即返回客户端响应，而是等待至少一个从库接收并写到relay log中，才会返回给客户端。MySQL在等待确认时，默认会等10秒，如果超过10秒没有收到ack，就会降级成为异步复制。

## MySQL半同步复制



这种半同步复制相比异步复制，能够有效的提高数据的安全性。但是这种安全性也不是绝对的，他只保证事务提交后的binlog至少传输到了一个从库，并且并不保证从库应用这个事务的binlog是成功的。另一方面，半同步复制机制也会造成一定程度的延迟，这个延迟时间最少是一个TCP/IP请求往返的时间。整个服务的性能是会有所下降的。而当从服务出现问题时，主服务需要等待的时间就会更长，要等到从服务的服务恢复或者请求超时才能给用户响应。

### 2》搭建半同步复制集群

半同步复制需要基于特定的扩展模块来实现。而mysql从5.5版本开始，往上的版本都默认自带了这个模块。这个模块包含在mysql安装目录下的lib/plugin目录下的semisync\_master.so和semisync\_slave.so两个文件中。需要在主服务上安装semisync\_master模块，在从服务上安装semisync\_slave模块。

```

[root@worker1 plugin]# pwd
/usr/local/mysql/lib/plugin
[root@worker1 plugin]# ll
总用量 90952
-rwxr-xr-x 1 7161 31415 166168 3月 26 2020 adt_null.so
-rwxr-xr-x 1 7161 31415 639696 3月 26 2020 authentication_ldap_sasl_client.so
-rwxr-xr-x 1 7161 31415 113784 3月 26 2020 auth_socket.so
-rwxr-xr-x 1 7161 31415 271816 3月 26 2020 component_audit_api_message_emit.so
-rwxr-xr-x 1 7161 31415 192920 3月 26 2020 component_log_filter_dragnet.so
-rwxr-xr-x 1 7161 31415 36968 3月 26 2020 component_log_sink_json.so
-rwxr-xr-x 1 7161 31415 146024 3月 26 2020 component_log_sink_syseventlog.so
-rwxr-xr-x 1 7161 31415 566016 3月 26 2020 component_mysqlbackup.so
-rwxr-xr-x 1 7161 31415 510288 3月 26 2020 component_validate_password.so
-rwxr-xr-x 1 7161 31415 1311200 3月 26 2020 connection_control.so
-rwxr-xr-x 1 7161 31415 3183232 3月 26 2020 ddl_rewriter.so
drwxr-xr-x 2 7161 31415 4096 3月 26 2020 debug
-rwxr-xr-x 1 7161 31415 60459336 3月 26 2020 group_replication.so
-rwxr-xr-x 1 7161 31415 633896 3月 26 2020 ha_example.so
-rwxr-xr-x 1 7161 31415 1303856 3月 26 2020 ha_mock.so
-rwxr-xr-x 1 7161 31415 1229744 3月 26 2020 innodb_engine.so
-rwxr-xr-x 1 7161 31415 2648040 3月 26 2020 keyring_file.so
-rwxr-xr-x 1 7161 31415 518872 3月 26 2020 keyring_udf.so
-rwxr-xr-x 1 7161 31415 1208848 3月 26 2020 libmemcached.so
-rwxr-xr-x 1 7161 31415 9064128 3月 26 2020 libpluginmecab.so
-rwxr-xr-x 1 7161 31415 91000 3月 26 2020 locking_service.so
-rwxr-xr-x 1 7161 31415 115064 3月 26 2020 mypluglib.so
-rwxr-xr-x 1 7161 31415 2675360 3月 26 2020 mysql_clone.so
-rwxr-xr-x 1 7161 31415 112048 3月 26 2020 mysql_no_login.so
-rwxr-xr-x 1 7161 31415 114176 3月 26 2020 rewrite_example.so
-rwxr-xr-x 1 7161 31415 1570552 3月 26 2020 rewriter.so
-rwxr-xr-x 1 7161 31415 1646120 3月 26 2020 semisync_master.so
-rwxr-xr-x 1 7161 31415 750408 3月 26 2020 semisync_slave.so
-rwxr-xr-x 1 7161 31415 438736 3月 26 2020 validate_password.so
-rwxr-xr-x 1 7161 31415 1340600 3月 26 2020 version_token.so

```

首先我们登陆主服务，安装semisync\_master模块：

```

mysql> install plugin rpl_semi_sync_master soname 'semisync_master.so';
Query OK, 0 rows affected (0.01 sec)

mysql> show global variables like 'rpl_semi%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| rpl_semi_sync_master_enabled | OFF |
| rpl_semi_sync_master_timeout | 10000 |
| rpl_semi_sync_master_trace_level | 32 |
| rpl_semi_sync_master_wait_for_slave_count | 1 |
| rpl_semi_sync_master_wait_no_slave | ON |
| rpl_semi_sync_master_wait_point | AFTER_SYNC |
+-----+-----+
6 rows in set, 1 warning (0.02 sec)

mysql> set global rpl_semi_sync_master_enabled=ON;
Query OK, 0 rows affected (0.00 sec)

```

这三行指令中，第一行是通过扩展库来安装半同步复制模块，需要指定扩展库的文件名。

第二行查看系统全局参数，rpl\_semi\_sync\_master\_timeout就是半同步复制时等待应答的最长等待时间，默认是10秒，可以根据情况自行调整。

第三行则是打开半同步复制的开关。

在第二行查看系统参数时，最后的一个参数`rpl_semi_sync_master_wait_point`其实表示一种半同步复制的方式。

半同步复制有两种方式，一种是我们现在看到的这种默认的`AFTER_SYNC`方式。这种方式下，主库把日志写入binlog，并且复制给从库，然后开始等待从库的响应。从库返回成功后，主库再提交事务，接着给客户端返回一个成功响应。

而另一种方式是叫做`AFTER_COMMIT`方式。他不是默认的。这种方式，在主库写入binlog后，等待binlog复制到从库，主库就提交自己的本地事务，再等待从库返回给自己一个成功响应，然后主库再给客户端返回响应。

然后我们登陆从服务，安装`semisync_slave`模块

```
mysql> install plugin rpl_semi_sync_slave soname 'semisync_slave.so';
Query OK, 0 rows affected (0.01 sec)

mysql> show global variables like 'rpl_semi%';
+-----+
| Variable_name          | Value |
+-----+
| rpl_semi_sync_slave_enabled | OFF   |
| rpl_semi_sync_slave_trace_level | 32    |
+-----+
2 rows in set, 1 warning (0.01 sec)

mysql> set global rpl_semi_sync_slave_enabled = on;
Query OK, 0 rows affected (0.00 sec)

mysql> show global variables like 'rpl_semi%';
+-----+
| Variable_name          | Value |
+-----+
| rpl_semi_sync_slave_enabled | ON     |
| rpl_semi_sync_slave_trace_level | 32    |
+-----+
2 rows in set, 1 warning (0.00 sec)

mysql> stop slave;
Query OK, 0 rows affected (0.01 sec)

mysql> start slave;
Query OK, 0 rows affected (0.01 sec)
```

slave端的安装过程基本差不多，不过要注意下安装完slave端的半同步插件后，需要重启下slave服务。

我们要注意，目前我们的这个MySQL主从集群是单向的，也就是只能从主服务同步到从服务，而从服务的数据表更是无法同步到主服务的。

```
mysql> 主服务
mysql> 插入记录为3的数据
mysql> insert into demotable value(3);
Query OK, 1 row affected (0.01 sec)

mysql> select * from demotable;
+----+
| id |
+----+
| 1  |
| 3  |
+----+
2 rows in set (0.00 sec)

mysql>

mysql> 从服务
mysql>
mysql> insert into demotable value(2);
Query OK, 1 row affected (0.00 sec)

mysql> select * from demotable;
+----+
| id |
+----+
| 1  |
| 2  |
| 3  |
+----+
3 rows in set (0.00 sec)

mysql>
```

数据只能从主服务同步到从服务，而不能反向同步

所以，在这种架构下，为了保证数据一致，通常会需要保证数据只在主服务上写，而从服务只进行数据读取。这个功能，就是大名鼎鼎的读写分离。但是这里要注意下，mysql主从本身是无法提供读写分离的服务的，需要由业务自己来实现。这也是我们后面要学的ShardingSphere的一个重要功能。

到这里可以看到，在MySQL主从架构中，是需要严格限制从服务的数据写入的，一旦从服务有数据写入，就会造成数据不一致。并且从服务在执行事务期间还很容易造成数据同步失败。

如果需要限制用户写数据，我们可以在从服务中将read\_only参数的值设为1(`set global read_only=1;`)。这样就可以限制用户写入数据。但是这个属性有两个需要注意的地方：

- 1、read\_only=1设置的只读模式，不会影响slave同步复制的功能。所以在MySQL slave库中设定了read\_only=1后，通过"show slave status\G"命令查看slave状态，可以看到slave仍然会读取master上的日志，并且在slave库中应用日志，保证主从数据库同步一致；
- 2、read\_only=1设置的只读模式，限定的是普通用户进行数据修改的操作，但不会限定具有super权限的用户的数据修改操作。在MySQL中设置read\_only=1后，普通的应用用户进行insert、update、delete等会产生数据变化的DML操作时，都会报出数据库处于只读模式不能发生数据变化的错误，但具有super权限的用户，例如在本地或远程通过root用户登录到数据库，还是可以进行数据变化的DML操作；如果需要限定super权限的用户写数据，可以设置super\_read\_only=0。另外 **如果要想连super权限用户的写操作也禁止，就使用"flush tables with read lock;"，这样设置也会阻止主从同步复制！**

## 5>扩展：MySQL高可用方案

我们之前的MySQL服务集群，都是使用MySQL自身的功能来搭建的集群。但是这样的集群，不具备高可用的功能。即如果是MySQL主服务挂了，从服务是没办法自动切换成主服务的。而如果要实现MySQL的高可用，需要借助一些第三方工具来实现。

这一部分方案只需要了解即可，因为一方面，这些高可用方案通常都是运维需要考虑的事情，作为开发人员没有必要花费太多的时间精力，偶尔需要用到时候能够用起来就够了。另一方面，随着业界技术的不断推进，也会冒出更多的新方案。例如ShardingSphere的5.x版本的目标实际上就是将ShardingSphere由一个数据库中间件升级成一个独立的数据库平台，而将MySQL、PostGreSql甚至是RocksDB这些组件作为数据库的后端支撑。等到新版本成熟时，又会冒出更多新的高可用方案。

常见的MySQL集群方案有三种：MMM、MHA、MGR。这三种高可用框架都有一些共同点：

- 对主从复制集群中的Master节点进行监控

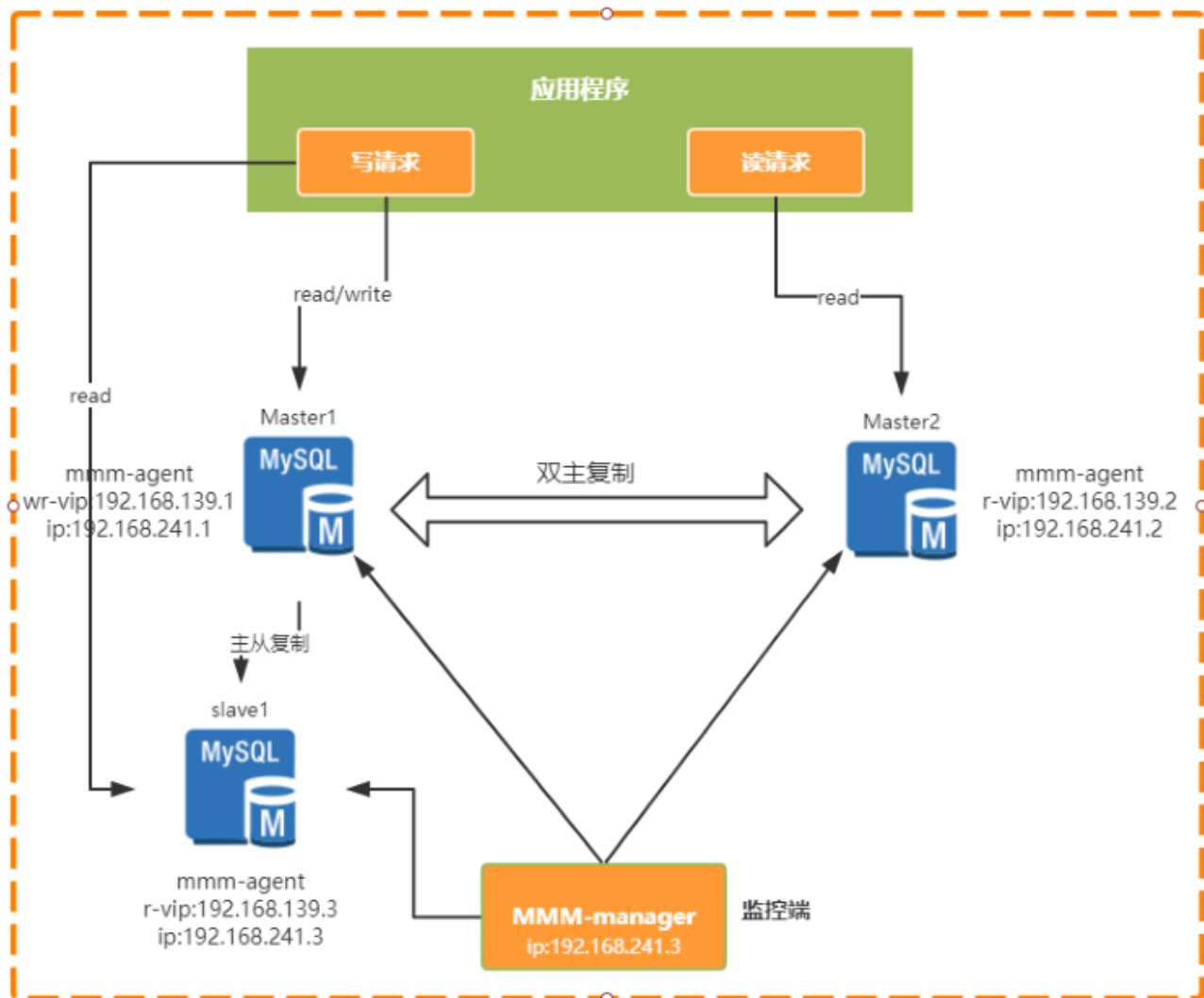
- 自动的对Master进行迁移，通过VIP。
- 重新配置集群中的其它slave对新的Master进行同步

## 1》MMM

MMM(Master-Master replication manager for Mysql, Mysql主主复制管理器)是一套由Perl语言实现的脚本程序，可以对mysql集群进行监控和故障迁移。他需要两个Master，同一时间只有一个Master对外提供服务，可以说是主备模式。

他是通过一个VIP(虚拟IP)的机制来保证集群的高可用。整个集群中，在主节点上会通过一个VIP地址来提供数据读写服务，而当出现故障时，VIP就会从原来的主节点漂移到其他节点，由其他节点提供服务。

### MMM高可用方案



优点：

- 提供了读写VIP的配置，使读写请求都可以达到高可用
- 工具包相对比较完善，不需要额外的开发脚本
- 完成故障转移之后可以对MySQL集群进行高可用监控

缺点：

- 故障简单粗暴，容易丢失事务，建议采用半同步复制方式，减少失败的概率
- 目前MMM社区已经缺少维护，不支持基于GTID的复制

适用场景：

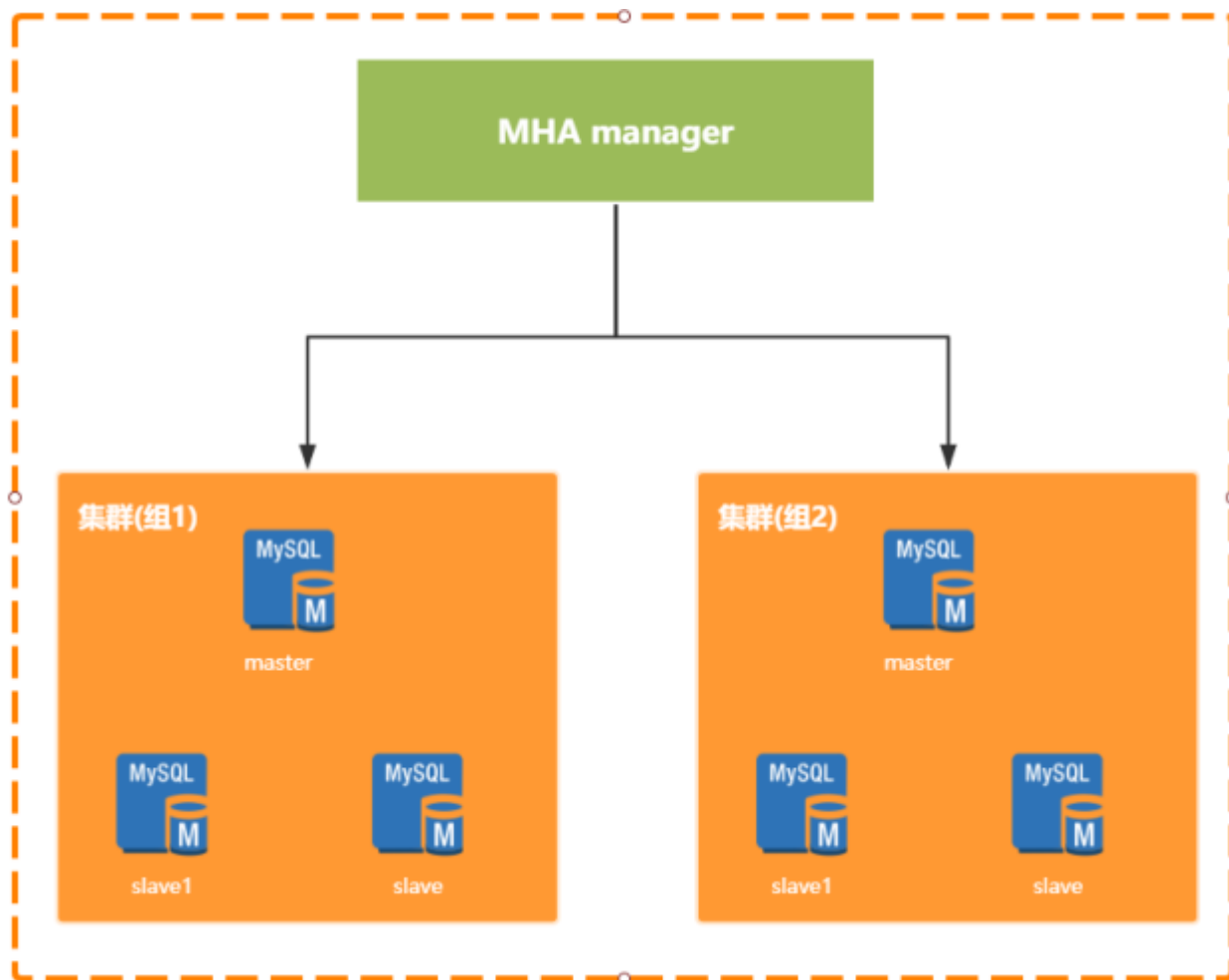
- 读写都需要高可用的
- 基于日志点的复制方式

## 2》MHA

Master High Availability Manager and Tools for MySQL。是由日本人开发的一个基于Perl脚本写的工具。这个工具专门用于监控主库的状态，当发现master节点故障时，会提升其中拥有新数据的slave节点成为新的master节点，在此期间，MHA会通过其他从节点获取额外的信息来避免数据一致性方面的问题。MHA还提供了mater节点的在线切换功能，即按需切换master-slave节点。MHA能够在30秒内实现故障切换，并能在故障切换过程中，最大程度的保证数据一致性。在淘宝内部，也有一个相似的TMHA产品。

MHA是需要单独部署的，分为Manager节点和Node节点，两种节点。其中Manager节点一般是单独部署的一台机器。而Node节点一般是部署在每台MySQL机器上的。Node节点得通过解析各个MySQL的日志来进行一些操作。

Manager节点会通过探测集群里的Node节点去判断各个Node所在机器上的MySQL运行是否正常，如果发现某个Master故障了，就直接把他的一个Slave提升为Master，然后让其他Slave都挂到新的Master上去，完全透明。



优点：

- MHA除了支持日志点的复制还支持GTID的方式
- 同MMM相比，MHA会尝试从旧的Master中恢复旧的二进制日志，只是未必每次都能成功。如果希望更少的数据丢失场景，建议使用MHA架构。

缺点：

MHA需要自行开发VIP转移脚本。

MHA只监控Master的状态，未监控Slave的状态

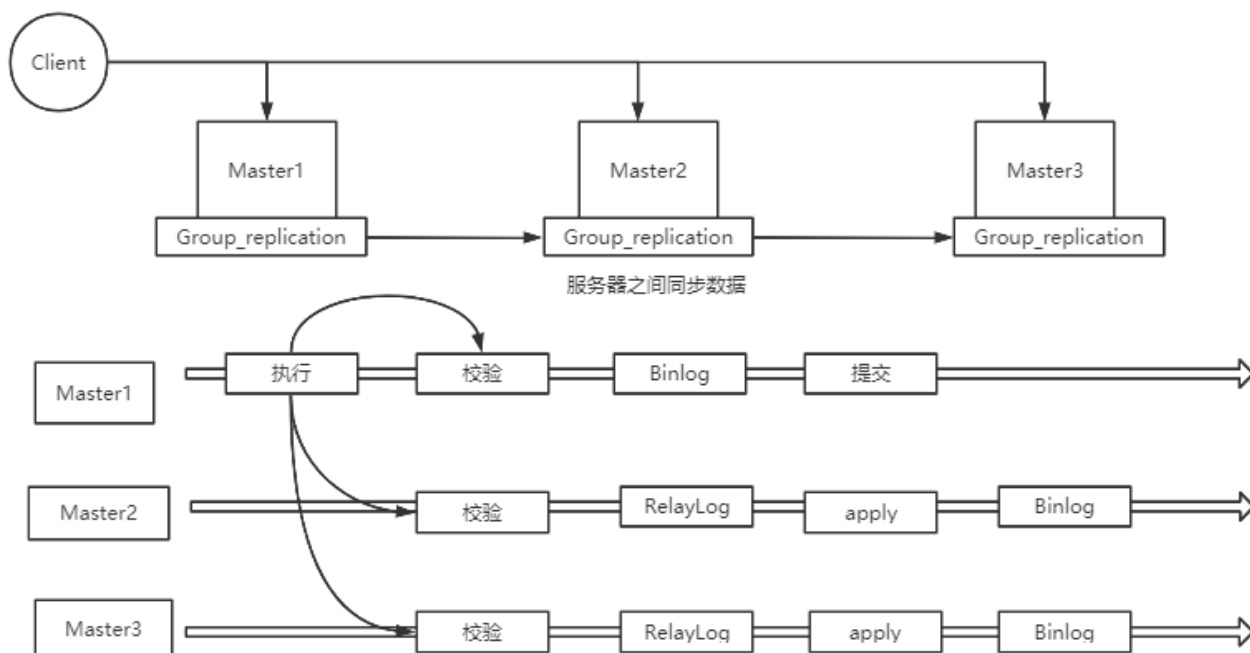
### 3》 MGR

MGR: MySQL Group Replication。是MySQL官方在5.7.17版本正式推出的一种组复制机制。主要是解决传统异步复制和半同步复制的数据一致性问题。

由若干个节点共同组成一个复制组，一个事务提交后，必须经过超过半数节点的决议并通过后，才可以提交。引入组复制，主要是为了解决传统异步复制和半同步复制可能产生数据不一致的问题。MGR依靠分布式一致性协议(Paxos协议的一个变体)，实现了分布式下数据的最终一致性，提供了真正的数据高可用方案(方案落地后是否可靠还有待商榷)。

支持多主模式，但官方推荐单主模式：

- 多主模式下，客户端可以随机向MySQL节点写入数据
- 单主模式下，MGR集群会选出primary节点负责写请求，primary节点与其它节点都可以进行读请求处理。



优点：

- 高一致性，基于原生复制及paxos协议的组复制技术，并以插件的方式提供，提供一致数据安全保证；
- 高容错性，只要不是大多数节点坏掉就可以继续工作，有自动检测机制，当不同节点产生资源争用冲突时，不会出现错误，按照先到者优先原则进行处理，并且内置了自动化脑裂防护机制；
- 高扩展性，节点的新增和移除都是自动的，新节点加入后，会自动从其他节点上同步状态，直到新节点和其他节点保持一致，如果某节点被移除了，其他节点自动更新组信息，自动维护新的组信息；
- 高灵活性，有单主模式和多主模式，单主模式下，会自动选主，所有更新操作都在主上进行；多主模式下，所有server都可以同时处理更新操作。

缺点：

- 仅支持InnoDB引擎，并且每张表一定要有一个主键，用于做write set的冲突检测；
- 必须打开GTID特性，二进制日志格式必须设置为ROW，用于选主与write set；主从状态信息存于表中 (--master-info-repository=TABLE、--relay-log-info-repository=TABLE)，--log-slave-updates打开；
- COMMIT可能会导致失败，类似于快照事务隔离级别的失败场景



- 目前一个MGR集群最多支持9个节点
- 不支持外键于save point特性，无法做全局间的约束检测与部分事务回滚

适用的业务场景：

- 对主从延迟比较敏感
- 希望对对写服务提供高可用，又不想安装第三方软件
- 数据强一致的场景

## 2、应用层提供管理多个数据源的能力

当我们有了集群化的后端数据库之后，接下来在应用层面，就需要能够随意访问多个数据库的能力。

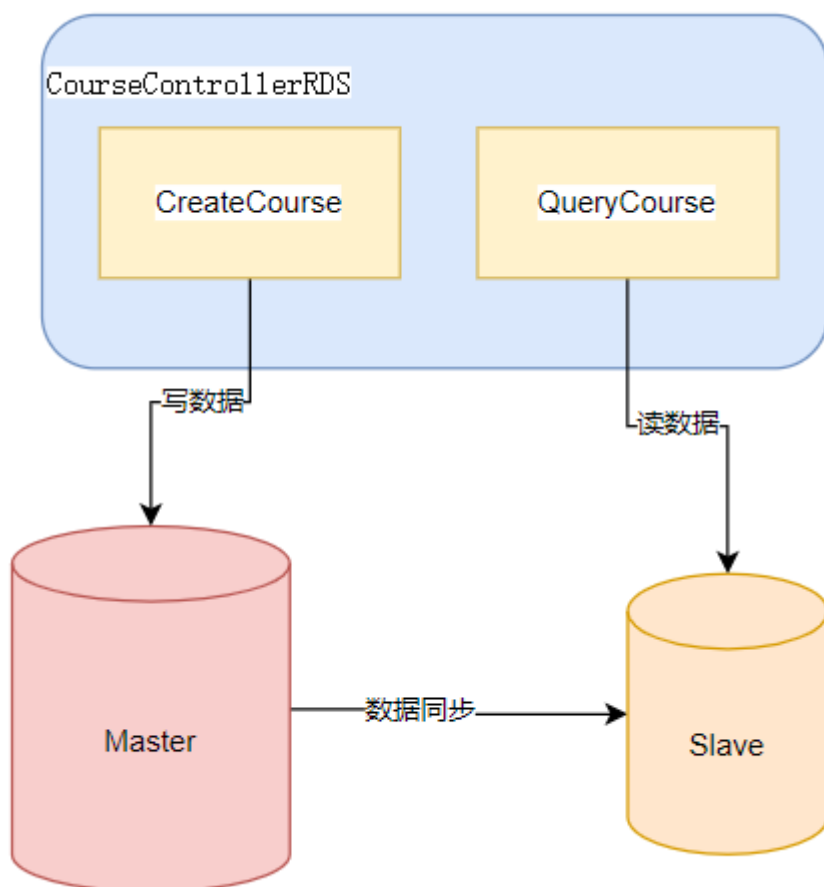
多数据源访问的实现方式有很多，例如基于Spring提供的AbstractRoutingDataSource组件，我们就可以快速切换后端访问的实际数据库。

具体代码详见课程配套示例的DynamicDS模块。

在示例中，我们可以配置两个不同的目标数据库，然后通过DynamicDataSource组件中的一个ThreadLocal变量实现快速切换目标数据库，从而让CreateCourse创建课程信息接口与QueryCourse查询课程接口分别操作两个不同的数据库。

## 3、数据库与应用结合，实现数据库集群管理

将DynamicDS模块后端的数据库指向我们之前搭建的数据库主从集群，就可以实现这样一种很有趣的实现方式：



由于主库与从库之间可以同步数据，虽然CreateCourse创建课程接口与QueryCourse查询课程接口是访问的不同的数据库，但是由于两个数据库之间可以通过主从集群进行数据同步，所以看起来，课程管理的两个接口就像是访问同一个数据库一样。这其实就是对于数据库非常常见的一种分布式的优化方案 **读写分离**。

数据库读写分离是一种常见的数据库优化方案，其基础思想是对数据的读请求和写请求分别分配到不同的数据库服务器上，以提高系统的性能和可扩展性。

一般情况下，数据库的读操作比写操作更为频繁，而且读操作并不会对数据进行修改，因此可以将读请求分配到多个从数据库服务器上进行处理。这样，即使一个从数据库服务器故障或者过载，仍然可以使用其他从数据库服务器来处理读请求，保证系统的稳定性和可用性。同时，将写操作分配到主数据库服务器上，可以保证数据的一致性和可靠性。主数据库服务器负责所有的写操作，而从数据库服务器只需要从主数据库服务器同步数据即可。由于主数据库服务器是唯一的写入点，可以保证数据的正确性和一致性。

读写分离只是数据库由单机服务升级为集群服务后带来的一个比较简单的业务场景。在这个过程中，只需要考虑切换数据库，而并不需要关注SQL以及数据是什么样子的。未来如果再涉及到对表数据的拆分，就会遇到更多，更复杂的业务场景。

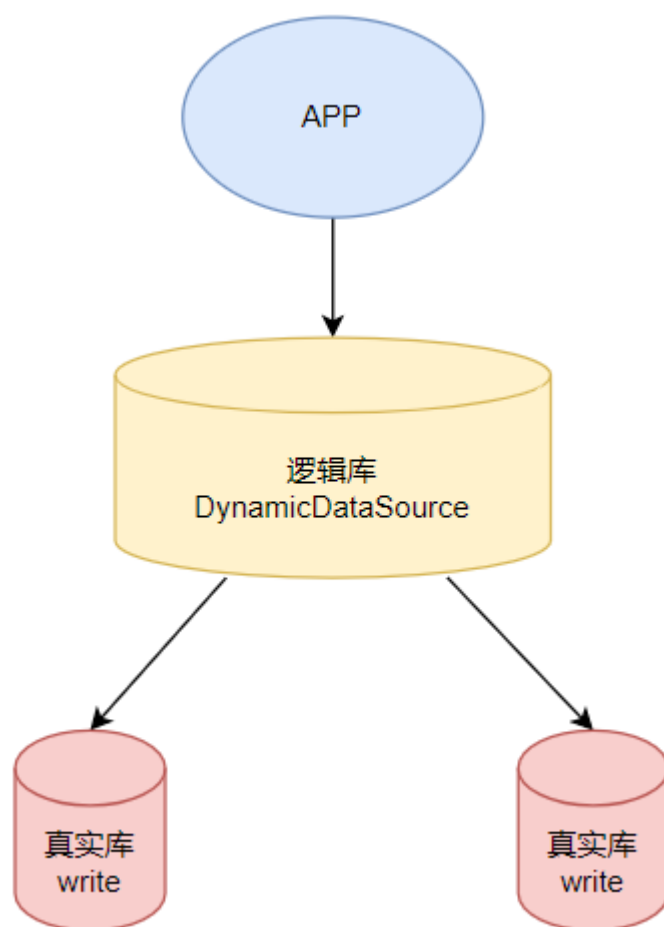
## 4、将多个数据源抽象成一个统一的数据源

在之前的示例中，我们已经实现了读写分离这样一个集群化场景下最为常见的数据库管理方案。但是，你会觉得，之前的实现方式其实对开发方式的侵入是挺大的，每次进行数据库操作之前，都需要先选择要操作那个数据库。有没有更为自然的多数据源管理方式呢？就是让业务真正像操作单数据源一样访问多个数据。这问题其实也已经有人帮我们想到了。MyBatis-Plus框架的开发者就开发了这样的一个框架DynamicDataSource，可以简化多数据源访问的过程。

具体参见课程配套示例的DynamicDataSource模块。

这个开源的DynamicDataSource小框架会自行在Spring容器当中注册一个具备多数据源切换能力的DataSource数据源，这样，在应用层面，只需要按照DynamicDataSource框架的要求修改配置接口，其他地方几乎感知不到与传统操作单数据源有什么区别。

这也就是说，应用只需要像访问单个数据源一样，访问DynamicDataSource框架提供的一个 **逻辑数据库**。而这个逻辑数据库会帮我们将实际的SQL语句转发到后面的 **真实数据库**当中去执行。



这种通过逻辑库来简化应用访问逻辑的方式，其实也是我们后面要学习的ShardingSphere框架需要做的事情。只不过，ShardingSphere提供的逻辑库功能会比我们这个简单的示例要强大很多，也复杂很多。

## 四、章节总结-重要

这一章节我们做了大量的实验，同时也构建了两个小应用来访问多个数据源。但是这些都不是这一章节的重点。

我们真正的重点，是希望你能通过这一系列的试验来理解我们即将要面对的分库分表到底是一个什么样复杂的场景。其实谈到分库分表，往往很难在一开始引起程序员们的注意。大家往往会更关注于微服务、分布式缓存等这一类应用层的设计，而数据库不过只是一个存储数据的工具而已。而这也造成了很多人即使用上了ShardingSphere这样的强大的分库分表工具，但是也不敢用得深。往往就把数据简单的拆分一下就结束了。但是，实际上，从我们这次的几个简单的示例中就能看到，就只说读写分离这样一个最为简单的数据库集群化的业务场景，也需要结合数据库产品以及应用层面一起，做大量的调整以及优化，才能形成一个很基础的解决方案。再结合一开始分析的分库分表需要面临的其他更复杂的问题，你可能更容易理解真实的分库分表是一个什么样的业务场景。

后续的章节我将会带你学习ShardingSphere这个非常强大的分库分表框架。但是希望你能够明白，框架虽然很强大，但是我们需要面对的问题也非常复杂。与MQ、微服务等场景不同，在分库分表这个业务场景下，ShardingSphere只是一个基础的工具，他能帮我们解决很多应用中最为常见的问题。但是这并不是全部。后续学习ShardingSphere的时候，我不希望你只是把他当成是一个框架来学习，而更希望他是你的一个工具，一个你解决分库分表各种稀奇古怪的问题时的一个可选项。并且做好一个思想准备，当你在复杂业务场景下需要使用ShardingSphere进行分库分表时，你遇到的很多问题大概率是无法直接用ShardingSphere解决的，很多时候还是需要你自己对ShardingSphere做出扩展。

有道云笔记连接: <https://note.youdao.com/s/hrlaj5X>