# 为什么要关注Mysql架构？

## 好处

### 提高可用性

如果某个数据库实例出现问题,对业务来说，这时可由其他的数据库实例提供服务，或者可以快速切换到其他的数据库实例，对业务来说基本无感知，也不会导致业务的中断。同时通过数据在多个实例之间的复制，提高数据的安全性和可用性。

### 提高性能

业务对数据的访问可以分散到不同的数据库实例上，可以根据数据访问类型不同，将不同性质的访问操作，进行分离，都可以降低单个数据库实例的访问压力。

## Mysql集群的可行方案

### MySQL Cluster

由Mysql本身提供，优势：可用性非常高，性能非常好。每份数据至少可在不同主机存一份拷贝，且冗余数据拷贝实时同步。但它的维护非常复杂，存在部分Bug，目前还不适合比较核心的线上系统，所以不推荐。

### DRBD磁盘网络镜像方案

Distributed Replicated Block Device，其实现方式是通过网络来镜像整个设备(磁盘).它允许用户在远程机器上建立一个本地块设备的实时镜像,与心跳链接结合使用,也可看做一种网络RAID。

优势：软件功能强大，数据可在底层快设备级别跨物理主机镜像，且可根据性能和可靠性要求配置不同级别的同步。IO操作保持顺序，可满足数据库对数据一致性的苛刻要求。但非分布式文件系统环境无法支持镜像数据同时可见，性能和可靠性两者相互矛盾，无法适用于性能和可靠性要求都比较苛刻的环境，维护成本高于MySQL Replication。另外，DRBD也是官方推荐的可用于MySQL高可用方案之一，所以这个大家可根据实际环境来考虑是否部署。

### MySQL Replication

在实际应用场景中，MySQL Replication是使用最为广泛的一种提高系统扩展性的设计手段。众多的MySQL使用者通过Replication功能提升系统的扩展性后，通过简单的增加价格低廉的硬件设备成倍 甚至成数量级地提高了原有系统的性能。

# Mysql复制

## 什么是Mysql复制？

复制是指将主数据库的 DDL和 DML 操作通过二进制日志传到复制服务器(也叫从库)上,然后在从库上对这些日志重新执行(也叫重做),从而使得从库和主库的数据保持同步。MysQL支持一台主库同时向多台从库进行复制,从库同时也可以作为其他服务器的主库,实现链状的复制 。

**注意:**

由于MySQL实现的是并不是完全同步的复制,所以主从库之间存在一定的差距,在从库上进行的査询操作需要考虑到这些数据的差异, 一般只有更新不频繁的数据或者对实时性要求不高的数据可以通过从库查询, 实时性要求高的数据仍然需要从主数据库获得。

**名称解释：**

DML（data manipulation language）数据操纵语言：

就是我们最经常用到的 SELECT、UPDATE、INSERT、DELETE。 主要用来对数据库的数据进行一些操作。

DDL（data definition language）数据库定义语言：

其实就是我们在创建表的时候用到的一些sql，比如说：CREATE、ALTER、DROP等。DDL主要是用在定义或改变表的结构，数据类型，表之间的链接和约束等初始化工作上

### 好处

1. 如果主库出现问题,可以快速切换到从库提供服务。
2. 可以在从库上执行查询操作, 降低主库的访问压力。
3. 某些数据库维护工作，比如备份，可以在从库上执行，以避免备份期间影响主库的服务。

### 原理概述

( 1 )首先, MySQL主库在事务提交时会把数据变更作为事件 Events 记录在二进制日志文件Binlog中; MySQL主库上的 sync\_binlog参数控制 Binlog日志刷新到磁盘。

( 2 )主库推送二进制日志文件 Binlog中的事件到从库的中继日志 Relay Log, 之后从库根据中继日志 Relay Log重做数据变更操作,通过逻辑复制以此来达到主库和从库的数据一致。

MySQL通过3个线程来完成主从库间的数据复制:其中 Binlog Dump线程跑在主库上, I/0线程和 SQL线程跑在从库上。当在从库上启动复制时,首先创建I/0程连接主库,主库随后创建 Binlog Dump线程读取数据库事件并发送给 I/0线程, I0线程获取到事件数据后更新到从库的中继日志 Relay Log中去,之后从库上的 SQL线程读取中继日志RelayLog中更新的数据库事件并应用。

可以通过 SHOW PROCESSLIST命令在主库上査看 BinlogDump线程,从 BinlogDump 线程的状态可以看到, Mysql的复制是主库主动推送日志到从库去的,是属于“**推**”日志的方式来做同步。同样地,在从库上通过 SHOW PROCESSLIST可以看到l/O线程和 SQL线程, l/O线程等待主库上的 Binlog Dump线程.发送事件并更新到中继日志 RelayLog, SQL线程读取中继日志并应用变更到数据库。

## 复制中的各类文件解析

### 日志文件

复制过程中涉及了两类非常重要的日志文件: 二进制日志文件( Binlog)和中继日志文件( Relay Log)。二进制日志文件( Binlog)会把 MysQL中的所有数据修改操作以二进制的形式记录到日志文件中,包括 Create、 Drop、 Insert、 Update、 Delete 操作等,但二进制日志文件(Binlog) 不会记录 Select操作, 因为 Select操作并不修改数据。

可以通过 show variables査看 Binlog的格式, Binlog支持 Statement、 Row、 Mixed三种格式,也对应了 MysQL的3种复制技术。

中继日志文件 Relay Log的文件格式、 内容和二进制日志文件 Binlog一样, 唯一的区别在于从库上的 SQL线程在执行完当前中继日志文件 Relay Log中的事件之后, SQL线程会自动删除当前中继日志文件 Relay Log，避免从库上的中继日志文件 Relay Log占用过多的磁盘空间。为了保证从库 Crash重启之后,从库的 I/0线程和 SQL线程仍然能够知道从哪里开始复制, 从库上默认还会创建两个日志文件 master.info和 relay\_log.info用来保存复制的进度。这两个文件在磁盘上以文件形式分别记录了从库的 l/0线程当前读取主库二进制日志 Binlog的进度和SQL线程应用中继日志 RelayLog的进度。

可以通过 show slave status命令能够看到当前从库复制的状态。

**主要参数：**

Master Host: 主库的 IP.

Master User 主库上, 主从复制使用的用户账号,

Master Port:主库 MySQL的端口号,

Master\_Log\_File:从库的I/0线程当前正在读取的主库 Binlog的文件 。

Read\_Master Log\_Pos:从库I/0线程当前读取到的位置。

Relay\_Log\_File: 从库 SQL线程正在读取和应用的中继日志 Relay Log的文件名 。

Relay\_Log\_Pos: 从库 SQL线程当前读取并应用的中继日志 Relay Log的位置。

Relay\_Master\_Log\_File:从库 SQL线程正在读取和应用的 Relay Log对应于主库Binlog的文件名 。

Exec\_Master\_Log\_Pos:中继日志 RelayLog中 Relay\_Log\_Pos位置对应于主库 Binlog 的位置。

### 三种复制技术

二进制日志文件 Binlog有三种格式：

Statement:基于 SQL语句级别的 Binlog,每条修改数据的 SQL都会保存到 Binlog里。

Row:基于行级别,记录每一行数据的变化,也就是将每行数据的变化都记录到 Binlog 里面, 记录得非常详细, 但是并不记录原始 SQL; 在复制的时候, 并不会因为存储过程或触发器造成主从库数据不一致的问通, 但是记录的日志量较 Statement格式要大得多 。

Mixed:混合Statement和Row模式,默认情况下采用 Statement模式记录,某些情况下会切换到 Row模式

同时也对应了 MysQL复制的3种技术。

在 binlog\_format设置为 Row格式时, MySQL实际上在 Binlog中逐行记录数据的变更, Row格式比 Statement格式更能保证从库数据的一致性(复制的是记录,而不是单纯操作 SQL)。当然, Row格式下的 Binlog的日志量很可能会增大非常多,在设置时需要考虑到磁盘空间间题。

参数 binlog\_format可以在全局设置或者在当前 session动态设置: 在全局设置会影响所有session,而在当前 session设置则仅仅影响当前 Session。可以通过 SET命令来实时修改二进日志文件(Binlog)的格式。

### 相关命令

查看当前复制方式

show variables like '%binlog%format%';

更改复制方式

set global binlog\_format = 'ROW';

set global binlog\_format = 'STATEMENT';

# 常用的复制架构

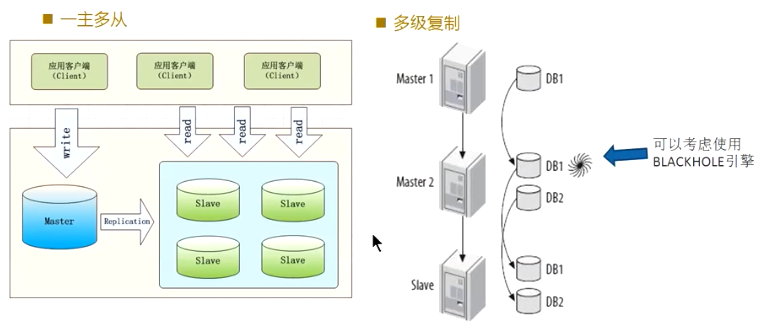
复制的3种常见架构有一主多从复制架构、多级复制架构和双主复制/DrualMaster架构

## 一主多从

在主库读取请求压力非常大的场景下, 可以通过配置一主多从复制架构实现读写分离, 把大量对实时性要求不是特别高的读请求通过负载均衡分布到多个从库上, 降低主库的读取压力，在主库出现异常宕机的情况下, 可以把一个从库切换为主库继续提供服务 。

## 多级复制

一主多从的架构能够解决大部分读请求压力特别大的场景的需求, 考虑到 MysQL的复制是主库“推送” Binlog日志到从库,主库的 I/0压力和网络压力会随着从库的增加而增长(每个从库都会在主库上有一个独立的 Binlog Dump线程来发送事件), 而多级复制架构解决了一主多从场景下,主库额外的 I/0和网络压力。

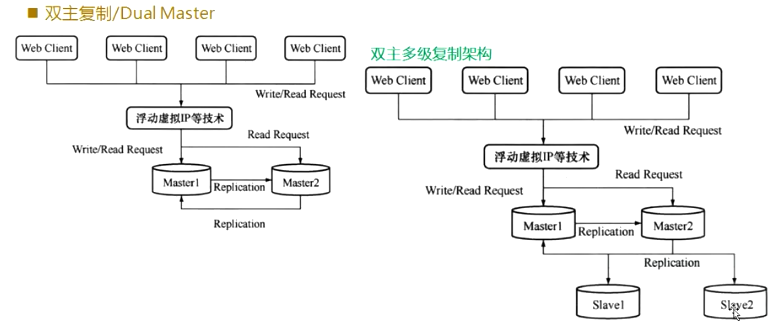


## 双主复制/Dual Master

其实就是主库 Master和 Master2互为主从， client客户端的写请求都访问主库 Master,而读请求可以选择访问主库 Master或 Master2。

### 双主多级复制架构

当然双主复制还能和主从复制联合起来使用:在 Master2库下配置从库 Slave、 Slave2等,这样即可通过从库 Slave等来分担读取压力，MyQL的双主多级复制架构如图所示



# 复制过程搭建

## 异步复制

主库执行完Commit后，在主库写入Binlog日志后即可成功返回客户端，无需等Binlog日志传送给从库。

步骤：

1、确保主从库安装了相同版本的数据库。

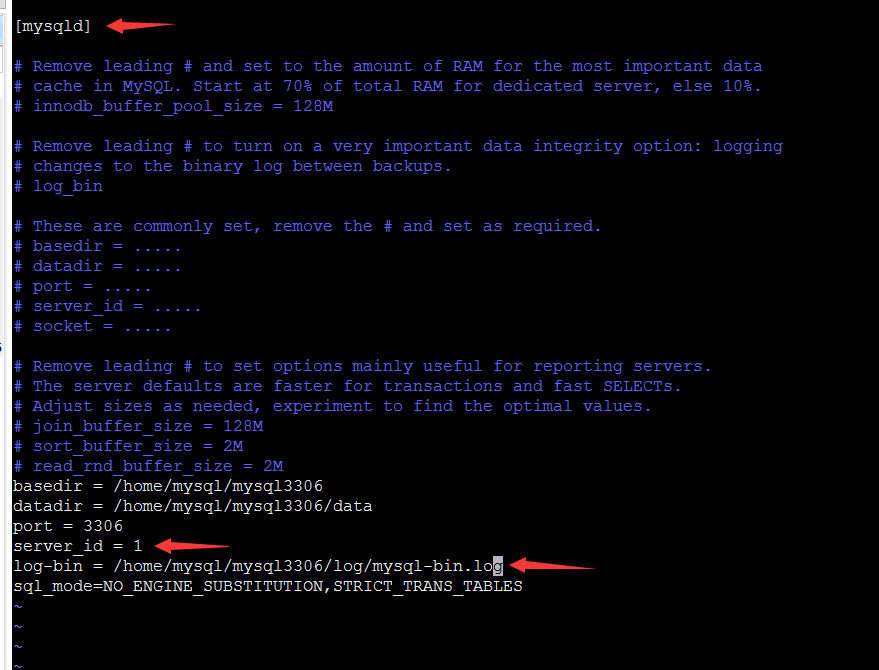
2、在主库上,设置一个复制使用的账户,并授予 REPLICATION SLAVE权限。这里创建一个复制用户repl,可以从IP为192.169.56.103的主机进行连接:



命令文本：GRANT REPLICATION SLAVE ON \*.\* To 'rep1'@'192.168.56.103' IDENTIFIED BY '1234test';

3、修改主数据库服务器的配置文件 my.cnf,开启 BINLOG,并设置 server-id的值。这两个参数的修改需要重新启动数据库服务才可以生效

在 my cnf中修改如下:



[mysqld]

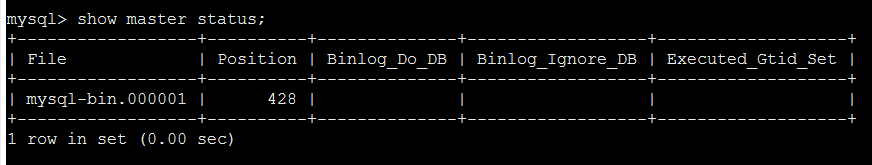
log-bin=/home/ mysql/log/mysql-bin. log

server-id= 1

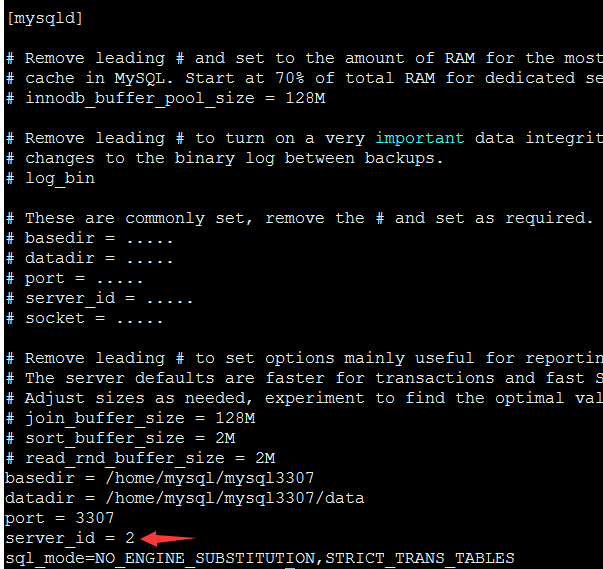
注意：如果mysql目录下无log目录，请先创建log目录

4、然后得到主库上当前的二进制日志名和偏移量值。这个操作的目的是为了在从数据库启动以后,从这个点开始进行数据的恢复。

执行show master status：



5、修改从数据库的配置文件 my.cnf,增加 server-id参数。注意 server-id的值必须是唯一的,不能和主数据库的配置相同,如果有多个从数据库服务器,每个从数据库服务器必须有自己唯一的 server-id值。



在 mycnf中修改如下:

[mysqld]

server-id=2

6、在从库上,使用 - -skip-slave- start选项启动从数据库,这样不会立即启动从数据库服务上的复制进程,方便我们对从数据库的服务进行进一步的配置:



操作命令：./bin/mysqld\_safe --defaults-file=/home/mysql/mysql3307/my.cnf --skip-slave-start

7、对从数据库服务器做相应设置,指定复制使用的用户,主数据库服务器的IP、端口

以及开始执行复制的日志文件和位置等,参考代码如下:

mysql> CHANGE MASTER TO

->MASTER\_HOST=master host name

->MASTER\_USER=replication\_user\_name

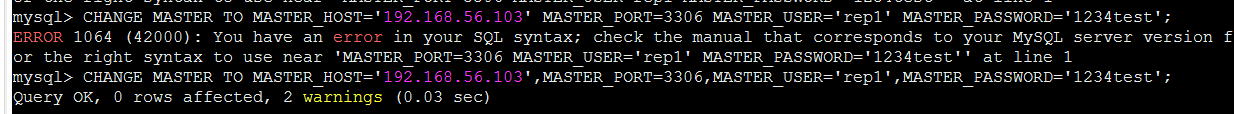
-> MASTER PASSWORD=replication\_password

->MASTER\_LOG\_FILE='recorded\_log\_file\_name

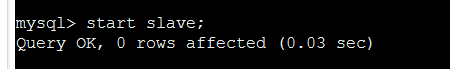
->MASTER\_LOG\_POS=recorded \_log\_position

举例说明如下:

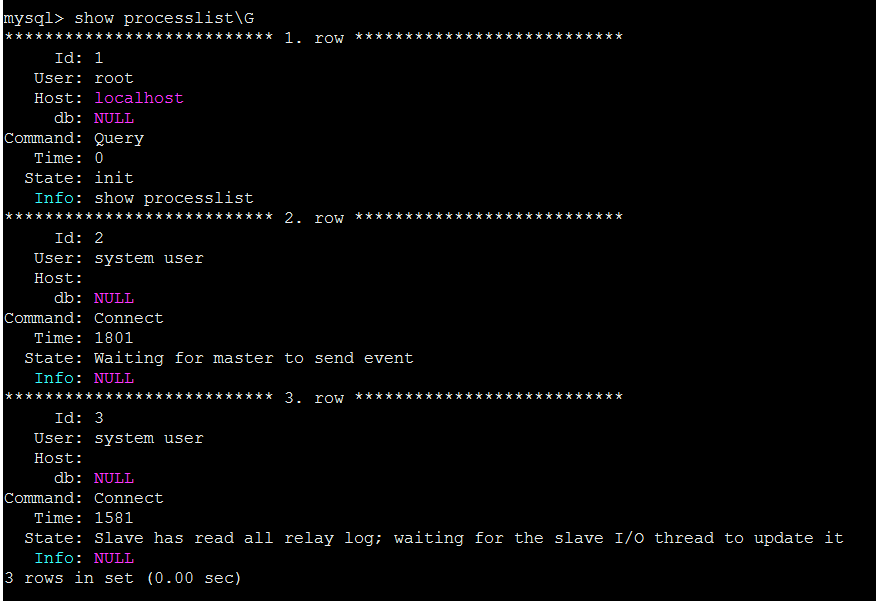
CHANGE MASTER TO MASTER\_HOST='192.168.56.103',MASTER\_PORT=3306,MASTER\_USER='rep1',MASTER\_PASSWORD='1234test' ,MASTER\_LOG\_FILE='mysql-bin.000001',MASTER\_LOG\_POS=428;



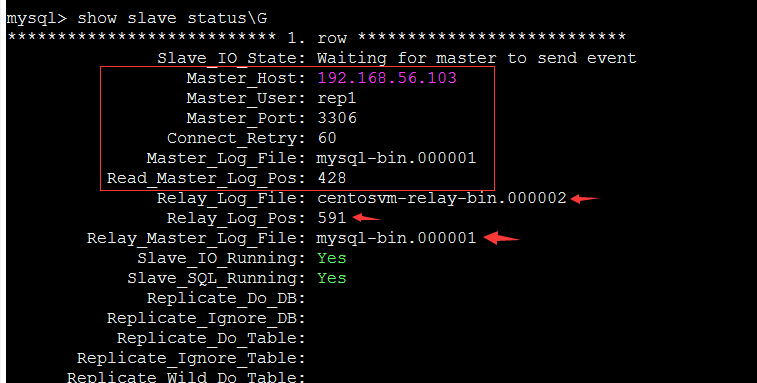
8、在从库上,启动 slave线程:



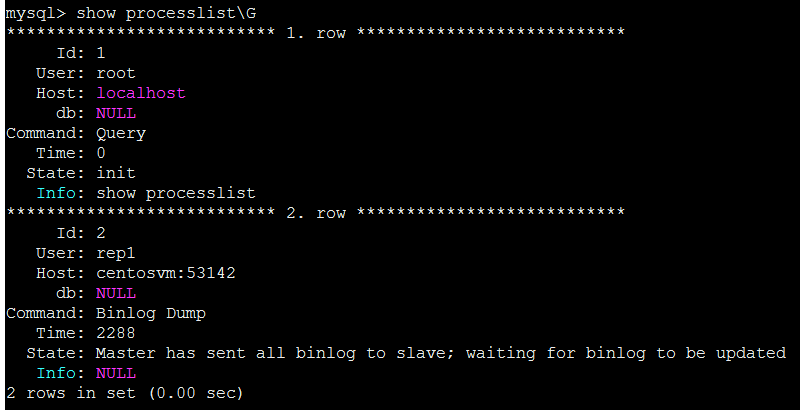
9、这时slave上执行 show processlist命令将显示类似如下的进程:



执行show slave status;将显示：



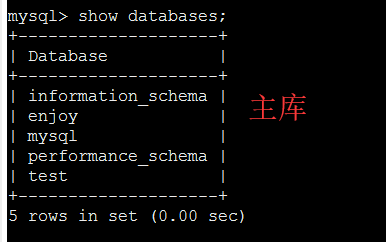
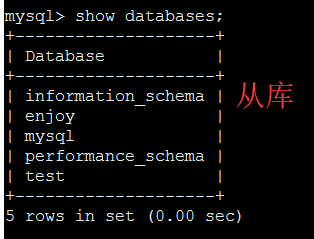
11、在master上执行 show processlist命令将显示类似如下的进程:



12、测试一下：

在主库上新建数据库和表，并插入数据，看看从库中是否会自动创建相关的数据库和表、插入数据。

新建数据库和表之前：



主库中创建数据库orders和表order\_exp，并插入数据



检查从库



## 半同步复制

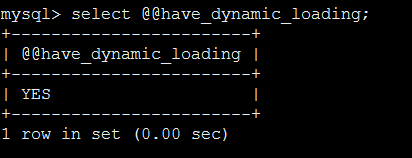
在 MySQL5.5之前, MySQL的复制是异步操作,主库和从库的数据之间存在一定的延迟,这样存在一个隐患:当在主库上写人一个事务并提交成功,而从库尚未得到主库推送的 Binlog日志时,主库宕机了,例如主库可能因磁盘损坏、内存故障等造成主库上该事务 Binlog丢失,此时从库就可能损失这个事务,从而造成主从不一致。

而半同步复制，是等待其中一个从库也接收到Binlog事务并成功写入Relay Log之后，才返回Commit操作成功给客户端；如此半同步就保证了事务成功提交后至少有两份日志记录，一份在主库Binlog上，另一份在从库的Relay Log上，从而进一步保证数据完整性；半同步复制很大程度取决于主从网络RTT（往返时延），以插件 semisync\_master/semisync\_slave 形式存在。

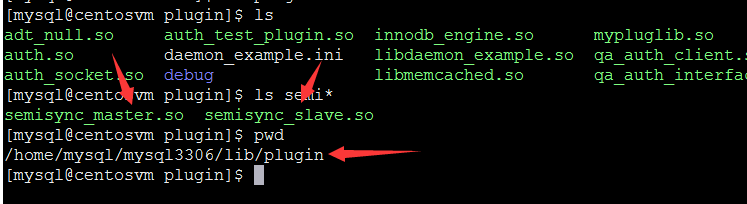
安装比较简单,在上一小节异步复制的环境上,安装半同步复制插件即可。

(1)首先,判断 MySQL服务器是否支持动态增加插件:

mysql> select @@have\_dynamic\_loading;



2)确认支持动态增加插件后,检查 MySQL的安装目录下是否存在插件：



安装插件：

在主库上安装插件semisync\_master.so:

mysql> install plugin rpl\_semi\_sync\_master SONAME 'semisync\_master.so'

从库上则安装 semisync\_slave.so插件:

mysql> install plugin rpl\_semi\_sync\_slave SONAME 'semisync\_slave.so';

安装完成后,从 plugin表中能够看到刚才安装的插件

mysql> select \* from mysql.plugin;

3)需要分别在主库和从库上配置参数打开半同步semi-sync,默认半同步设置是不打开的,主库上配置全局参数：

mysql> set global rpl\_semi\_sync\_master\_enabled=1;

mysql> set global rpl\_semi\_sync\_master timeout 30000;

从库上一样配置全局参数：

mysql> set global rpl\_semi\_sync\_slave\_enabled=1;

4)其他步骤同异步复制

# 读写分离实战

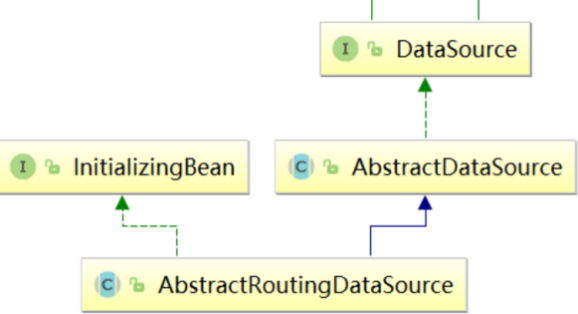
## SpringBoot+MyBatis结合MySQL读写分离

读写分离要做的事情就是对于一条SQL该选择哪个数据库去执行，至于谁来做选择数据库这件事儿，无非两个，要么中间件帮我们做，要么程序自己做。因此，一般来讲，读写分离有两种实现方式。第一种是依靠中间件帮我们做SQL分离；第二种是应用程序自己去做分离。这里我们选择程序自己来做，主要是利用Spring提供的路由数据源，以及AOP

然而，应用程序层面去做读写分离最大的弱点（不足之处）在于无法动态增加数据库节点，因为数据源配置都是写在配置中的，新增数据库意味着新加一个数据源，必然改配置，并重启应用。当然，好处就是相对简单。

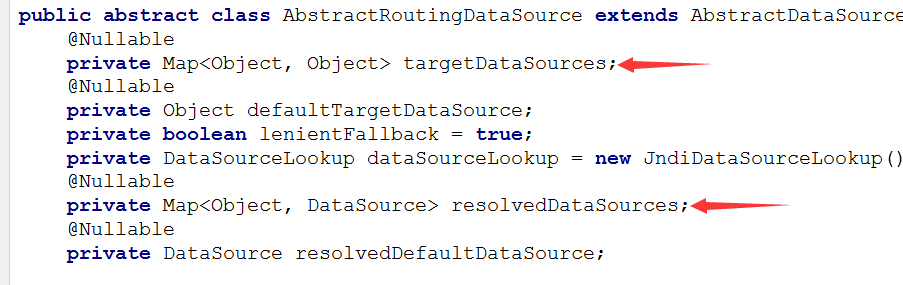
主要思路：

1、选择数据库执行这个关键性问题，考察Spring的封装体系中

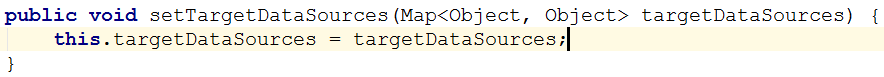


 的AbstractRoutingDataSource内部维护了一组目标数据源，并且做了路由key与目标数据源之间的映射，提供基于key查找数据源的方法。因此我们需要声明一个自己的的数据源MyRoutingDataSource来负责，并应该继承自AbstractRoutingDataSource，覆盖determineCurrentLookupKey方法，这个方法返回结果就是我们对数据源的选择。

2、考察AbstractRoutingDataSource的内部，有两个容器：



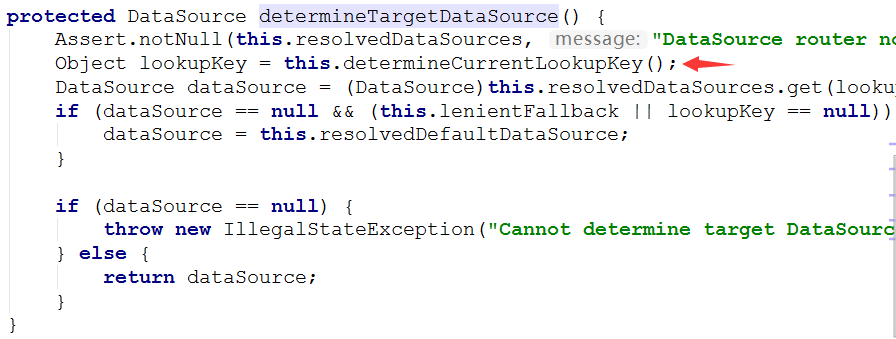
其中targetDataSources是指我们通过方法



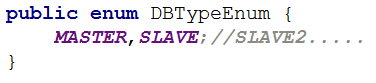
设置的目标数据源，而resolvedDataSources是实际的数据源，Spring通过方法

进行了转换。

而在实际运行时，Spring通过getConnection方法获取数据库连接，在getConnection方法中最终调用 determineTargetDataSource方法来定位实际的数据源，在determineTargetDataSource方法就用到了我们前面覆盖的determineCurrentLookupKey方法来从容器resolvedDataSources获得实际的数据源

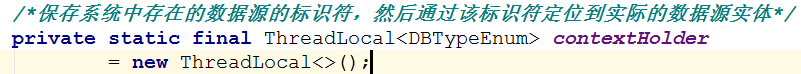


3、所以，我们先定义一个枚举类



来表示当前有几个数据源，并用这个类作为两个数据源容器的key。

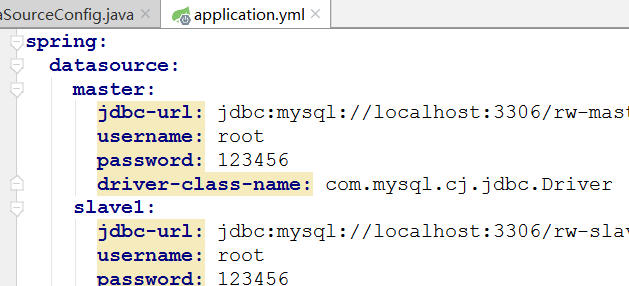
4、同时为了保证线程安全，每个对数据库操作的线程当前所持有的数据源对象，我们用一个ThreadLocal来保存



然后用类DBContextHolder进行包装，并对外提供主从库切换方法、每个线程的数据源get和set方法

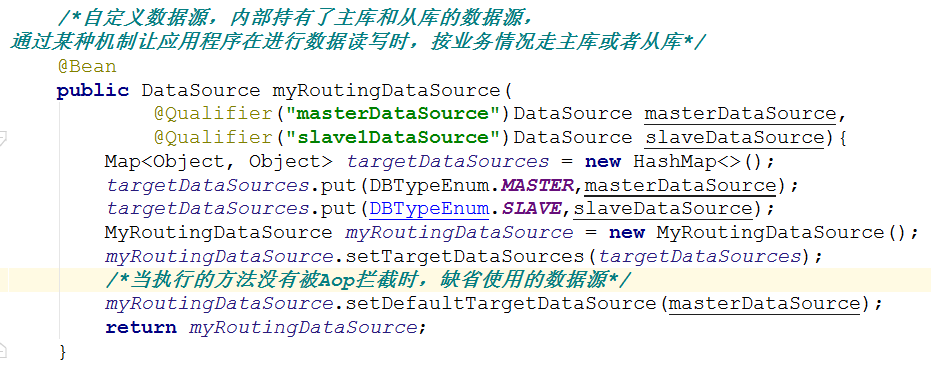
5、在我们自己的的数据源MyRoutingDataSource的determineCurrentLookupKey方法中，只需要即可。

6、在application.yml中配置原生数据源后通过配置类DataSourceConfig加载，





7、然后在 DataSourceConfig中生成一个虚拟数据源myRoutingDataSource的实例，专门负责实际数据源的持有和路由，



8、应用中的事务管理器和MyBatis均使用这个虚拟数据源myRoutingDataSource



9、为了减少对业务代码的侵入，我们定义了一个SpringAop类DataSourceAop，其中定义了两个切点，slavePointcut和masterPointcut，按照方法名进行数据源的切换，比如查询方法要求为**query**开头等等。

这样默认情况下，所有的查询都走从库，插入/修改/删除走主库

10、特殊情况是某些情况下我们需要强制读主库，针对这种情况，我们定义一个注解用该注解标注的就读主库，对应的，修改下DataSourceAop中切点的定义。

11、测试下我们的代码即可，对应的代码在模块rw-separation下。

# Mysql+Keepalived实现双主高可用

## 什么是Keepalived？

Keepalived是一个工作在IP/TCP协议栈的IP层，TCP层及应用层实现交换机制的软件。

Keepalived的作用是检测服务器的状态，如果有一台服务器宕机，或工作出现故障，Keepalived将检测到，并将有故障的服务器从系统中剔除，同时使用其他服务器代替该服务器的工作，当服务器工作正常后Keepalived自动将服务器加入到服务器群中，这些工作全部自动完成，不需要人工干涉，需要人工做的只是修复故障的服务器。

原理如下：

IP层： Keepalived会定期向服务器群中的服务器发送一个ICMP的数据包（既我们平时用的Ping程序）,如果发现某台服务的IP地址没有激活，Keepalived便报告这台服务器失效，并将它从服务器群中剔除。

TCP层:主要以TCP端口的状态来决定服务器工作正常与否。如web server的服务端口一般是80，如果Keepalived检测到80端口没有启动，则Keepalived将把这台服务器从服务器群中剔除。

应用层：对指定的URL执行HTTP GET。然后使用MD5算法对HTTP GET结果进行求和。如果这个总数与预期值不符，那么测试是错误的，服务器将从服务器池中移除。该模块对同一服务实施多URL获取检查。

Keepalived是基于VRRP协议的一款高可用软件。Keepailived有一台主服务器和多台备份服务器，在主服务器和备份服务器上面部署相同的服务配置，使用一个虚拟IP地址对外提供服务，当主服务器出现故障时，虚拟IP地址会自动漂移到备份服务器。

名词解释：

VRRP（Virtual Router Redundancy Protocol，虚拟路由器冗余协议），由IETF提出的解决局域网中配置静态网关出现单点失效现象的路由协议，1998年已推出正式的RFC2338协议标准。相类似的协议有：GLBP（思科Cisco独有）, HSRP等等。

## 和HAProxy的区别是什么？

**HAProxy请求分发**  
可以理解为类似于通过nginx来作后端的负载均衡，haproxy可以通过监听一个统一的端口对外提供能力，然后内部进行分发。除支持http7层处理外，还顺便为mysql支持4层转发。(更高级的可以考虑采用lvs) 。

举例说明，将两个mysql分别配置在haproxy当中，程序进行访问时，就不再访问具体的mysql机器，而是访问这个haproxy所在的机器。这里就提到需要一个额外的机器来提供服务，但是由于只为haproxy使用，其根据很低，一个最低配机器即可，费用不大。同时，相应的端口也填写haproxy所暴露的端口即可。对外即认为也只仍然只有一个mysql，即对外是透明的。

haproxy在进行处理时，将自己根据相应的策略进行转发，最简单的策略就是轮询(ribbon),当然其它加权或者是随机等，需要具体进行配置。同时，根据设定的具体时间间隔，对后端服务进行有效性检测，当mysqlA或B不能工作时，将自动从可用列表中移除。

在加上haproxy之后，负载的问题被解决，但另一个问题又来了，即服务单点的问题。如果一旦这个haproxy机器挂掉(或网络原因)。虽然,mysql服务器没挂，但整个服务也是不可用了。前端程序不会自动退回到去访问原始的mysql(甚至由于防火墙的问题，它也不能访问)。这时候就要用到另一个东西，保证haproxy不会成为单点，即KeepAlived。

保证服务不会单点的作法就是加机器，但加机器就会出现多个ip，如何保证前端程序使用单个ip又能保证后端的实际处理机器为多台，这就是KeepAlived的作用。

我们为了保证haproxy的高可用，已经又加了一个机器，即为haproxyA和haproxyB（相当于原来的2个mysql机器，又加了2台haproxy机器)。

通过KeepAlived，我们可以创造出第3个IP（也称虚拟IP），由虚拟IP来对外提供服务，但是与HaProxy的转发性质不同，这里的虚拟IP实际上就是HaProxyA和HaProxyB的一个同名映射。可以理解为HaProxyA和HaProxyB都在争抢这个ip，哪个争抢到了，就由哪个来提供服务。

因为KeepAlived不提供任何处理能力，实际上最终的处理落在能够处理信息的程序上。因此，我们可以将KeepAlived和HaProxy部署在一起，即KeepAlived负责抢ip，接收前端的请求，在接收到了请求之后，由系统自动将请求分发到同一个机器上的HaProxy上进行处理。即一个机器有2个IP，ip1负责接收请求,ip2负责实际的信息处理。

最终部署模型

a、MysqlA(ip5)和MysqlB(ip6)水平扩展，通过双主进行数据通信，并且同时提供服务能力

b、通过HaProxyA(ip3)和HaProxyB(ip4)提供mysql的负载能力，将请求路由到指定的mysql服务器，同时监控后端的mysql数据库可用性，如果取消这一层，则Mysql的两个服务器不能同时提供写服务。

c、将KeepAlivedA和KeepAlivedB分别和HaProxyA和HaProxyB部署在一起，同时绑定VIP，对外提供访问ip，同时监控本机的HaProxy的可用性

## Mysql+Keepalived双主搭建说明

最关键的部分其实是Mysql的两个服务器要互为主从，所以在my.cnf的配置文件中，除了log-bin参数以外，还需要增加并设置参数logs-slave-updates=1

logs-slave-updates参数解释：

通常情况，从服务器从主服务器接收到的更新不记入它的二进制日志。该选项告诉从服务器将其SQL线程执行的更新记入到从服务器自己的二进制日志。为了使该选项生效，还必须用--logs-bin选项启动从服务器以启用二进制日志。如果想要应用链式复制服务器，应使用--logs-slave-updates。例如，可能你想要这样设置：

A -> B -> C

也就是说，A为从服务器B的主服务器，B为从服务器C的主服务器。为了能工作，B必须既为主服务器又为从服务器。你必须用--logs-bin启动A和B以启用二进制日志，并且用--logs-slave-updates选项启动B。

Mysql其余的配置同异步复制，比如：A🡨--🡪B

那么A、B上都要配置复制用户名，都要执行CHANGE MASTER TO语句，都要执行start slave语句。

至于Keepalived的安装和配置比较复杂，在技能上至少要求可以编写sh脚本。有兴趣的同学可以参考：

https://www.cnblogs.com/ivictor/p/5522383.html

# 分库分表综述

## 分库分表需求的产生和定义

分库分表, 顾名思义, 就是使用多个库和多个表甚至多个数据库实例来存储海量的数据。为了分散数据库的压力, 我们会采用分库分表的方案, 将一个表结构分为多个表, 或者将一个表的数据分片后放入多个表, 这些表可以放在同一个库里, 也可以放到不同的库里, 甚至可以放在不同的数据库实例上。

## 不同的业务规模阶段数据库的变化

#### 单实例单库

单库单表是最常见的数据库设计,例如现在我们有用户表(以下简称 user表)，订单表(以下简称 order表)， 所有用户和订单的信息都被存储在同一个数据库中。随着业务规模的增加，我们的数据库在架构上的变化有几种可能的路径。

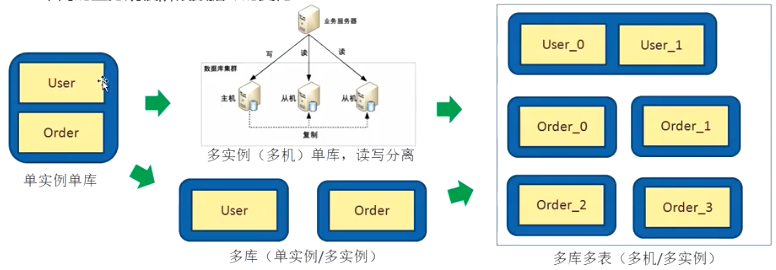
#### 多实例（多机）单库，读写分离

用户表和订单表仍然归属同一个数据库，但是数据库有多份，并放在多个实例上，同时实例之间启用了复制功能和读写分离。

#### 多库（单实例/多实例）

用户表和订单表分开来放到不同的数据库上，比如用户库和订单库。同时这些库分布在单个数据库实例上或者多个数据库实例上都有可能。至于是否启用读写分离则看情况而定。

#### 多库多表

当业务规模进一步增加，单表的数据已经超过了一定量，比如1000万，这个时候，就会进入另一个变化阶段。会将不同的业务表拆分到不同业务数据库，同时业务表也会进行拆分，也就是分库分表。

## 什么情况下需要分库分表？

总结一下，首先, 如果在一个库中的表数据超过了一定的数量, 例如在 MySQL的表中达到千万级别, 就需要考虑进行分表, 这样, 数据就被分散在不同的表上, 单表的索引大小得到了挠制, 会提升査询性能, 对索引及表结构的变更会很方便和高效 。 当数据库实例的吞吐量达到性能的瓶颈时,我们需要扩展数据库实例,让每个数据库实例承担其中一部分数据库的请求,分解总体的大请求量带来的压力。

所以，基本上就是两点：

如果在数据库中表的数量达到了一定的量级, 则需要进行分表, 分解单表的大数据量对索引査询带来的压力, 并方便对索引和表结构的变更 。

如果数据库的吞吐量达到了瓶颈, 就需要增加数据库实例, 利用多个数据库实例来分解大量的数据库请求带来的系统压力。

## 分库分表后，如何对数据进行访问？

### 客户端分片

客户端分片就是使用分库分表的数据库的应用层直接操作分片逻辑, 分片规则需要在同一个应用的多个节点间进行同步, 每个应用层都嵌入一个操作切片的逻辑实现 (分片规则),这一般通过依赖 Jar包来实现具体的实现方式分为三种:在应用层直接实现、通过定制JDBC协议实现、通过定制 ORM框架实现。

#### 应用层直接实现

这是一种非常通用、 简单的解决方案, 直接在应用层读取分片规则, 然后解析分片规则, 根据分片规则实现切分的路由逻辑,从应用层直接决定每次操作应该使用哪个数据库实例、数据库及哪个数据库的表等。

这种解决方案虽然侵入了业务, 但是实现起来比较简单, 适合快速上线, 而且性能更高，切分逻辑是自己开发的,如果在生产上产生了问题,则都比较容易解决。但是,这种实现方式会让数据库保持的连接比较多。

#### 定制JDBC协议

在应用层直接实现或多或少使切片逻辑侵入了应用的业务实现, 而且对开发人员的能力要求较高, 开发人员既需要实现业务逻辑, 也需要实现框架需求。

为了让开发人员把精力集中在业务逻辑的实现上,我们可以通过定制 JDBc协议来实现,也就是针对业务逻辑层提供与 JDBC 一致的接口,让业务开发人员不必关心分库分表的实现,让分库分表在 JDBC的内部实现,对业务层保持透明。

这种解决方案对业务透明,不侵入业务,让开发和分库分表的配置人员在一定程度上可以分离,可以让开发人员更大限度地聚焦在业务逻辑的实现上,但开发人员需要理解 JDBC协议才能实现分库分表逻辑。Sharding JDBC便采用了这种方案。

#### 定制ORM框架

因为现在ORM框架有了大量的应用，因此有了通过定制ORM框架实现分库分表的方案，也就是把分片规则实现放到ORM框架中或者通过ORM框架支持的扩展机制来完成。

## 代理分片

代理分片就是在应用层和数据库层之间增加一个代理层。代理层对外提供与JDBC兼容的接口给应用层，应用层的开发人员不用关心分片规则，只需要关心业务逻辑的实现。

这种方案的优点是让应用层的开发人员专注于业务逻辑的实现, 把分库分表的配置留给代理层做;缺点是增加了代理层。尽管代理层是轻量级的转发协议,但是毕竟要实现 JDBC协议的解析, 并通过分片的路由规则来路由请求, 对每个数据库操作都增加了一层网络传输, 这对性能是有影响的,需要维护增加的代理层,也有硬件成本,还要有能够解决Bug的技术专家, 成本很高。

通过代理分片实现的主要框架有 Cobar 、Mycat和Sharding-Proxy等。

## 分库分表第三方产品

### 京东金融的ShardingSphere

网址：<http://shardingsphere.io/index_zh.html>

脱胎于当当网的Sharding-JDBC，在京东金融得到重生，并且目前已经进入Apache孵化器，目前已经组成了分布式数据库中间件解决方案组成的生态圈，它由Sharding-JDBC、Sharding-Proxy和Sharding-Sidecar（计划中）这3款相互独立的产品组成。 他们均提供标准化的数据分片、分布式事务和数据库治理功能。作为京东第一个进入Apache基金会的开源产品，值得持续关注。

### 开源的Cobar和MyCat

Cobar 是由 Alibaba 开源的 MySQL 分布式处理中间件，目前已停止维护，它可以在分布式的环境下看上去像传统数据库一样提供海量数据服务。社区使用过程中发现存在一些比较严重的问题及使用限制，于是Mycat诞生。开源以后，最终Mycat发展成为一个由众多软件公司的实力派架构师和资深开发人员维护的社区型开源软件。

Mycat对Cobar的代码进行了彻底的重构，使用NIO重构了网络模块，并且优化了Buffer内核，增强了聚合，Join等基本特性，同时兼容绝大多数数据库成为通用的数据库中间件。1.4 版本以后完全的脱离基本Cobar内核。

目前看来，Mycat虽然市场占有率很高，但是Mycat现在呈现一个发展停滞的状态，未来前景如何还有待观察。

网址： http://www.mycat.io/

### 其他第三方的中间件

例如Oneproxy ，由原支付宝的首席DBA为创始人的公司开发，提供了免费版和商用版。使用C&C++开发，单个OneProxy实例可支持高达40W的QPS/TPS 。商用版则有更强的性能，更多的功能，还可以得到正式的技术支持。

有专门的运维或者DBA的公司，则可以考虑使用。最大的好处是对开发人员完全透明。

网址： <http://www.onexsoft.com/>

更多的分库分表中间件：

TDDL：已经不再使用，下一代是 DDRS。淘宝根据自己的业务特点开发了 TDDL （Taobao Distributed Data Layer 外号:头都大了 ）框架。 TDDL 通过部署在客户端的 jar 包，将用户的 SQL 路由到指定的数据库中。

Heisenberg：源自Coba，目前并不活跃，由百度员工个人编写。

Atlas ： 360 Web平台部基础架构团队开发维护的一个基于 MySQL 协议的数据中间层项目。它是在mysql-proxy 0.8.2版本的基础上，对其进行了优化，增加了一些新的功能特性

Vitess：谷歌开发的数据库中间件，集群基于 ZooKeeper 管理，通过 RPC 方式进行数据处理

DRDS：阿里分布式关系型数据库服务（Distribute Relational Database Service，简称DRDS）是一种水平拆分、可平滑扩缩容、读写分离的在线分布式数据库服务。前身为淘宝 TDDL。

还有：Mango，Amoeba等等

## 常用切分方式

### 垂直切分

把单一库中的表按业务拆分到多个库，或者把单一的表拆分成多个表，并分散到不同的数据库(主机)上。

垂直切分除了用于分解单库单表的压力, 也用于实现冷热分离, 也就是根据数据的活跃度进行拆分, 因为对拥有不同活跃度的数据的处理方式不同 。

我们可将本来可以在同一个表中的内容人为地划分为多个表。比如在微博系统的设计中, 一个微博对象包括文章标题、作者、分类、创建时间等属性字段,这些字段的变化频率低,査询次数多,叫作冷数据。而博客的浏览量、回复数、 点赞数等类似的统计信息, 或者别的变化频率比较高的数据, 叫作活跃数据或者热数据。我们把冷热数据分开存放,就叫作冷热分离,在 MysQL的数据库中,冷数据査询较多, 更新较少可以用MyISAM引擎,而热数据更新比较频繁,适合使用InnoDB存储引擎,这也是垂直拆分的一种。

其次, 对读多写少的冷数据可配置更多的从库来化解大量査询请求的压力; 对于热数据, 可以使用多个主库构建分库分表的结构,对于一些特殊的活跃数据或者热点数据,也可以考虑使用 Memcache、 Redis之类的缓存,等累计到一定的量后再更新数据库。

优点：

拆分后业务清晰,拆分规则明确。

系统之间进行整合或扩展很容易。

按照成本、应用的等级、应用的类型等将表放到不同的机器上,便于管理。

便于实现动静分离、冷热分离的数据库表的设计模式。

数据维护简单。

缺点：

部分业务表无法关联(Join),只能通过接口方式解决,提高了系统的复杂度。

事务处理复杂。

### 水平切分

根据表中数据的逻辑关系,将同一个表中的数据按照某种条件拆分到多台数据库(主机)上。在每个表中包含一部分数据, 所有表加起来就是全量的数据。这种切分方式根据单表的数据量的规模来切分, 保证单表的容量不会太大, 从而保证了单表的査询等处理能力,例如将用户的信息表拆分成user1、 User2等,表结构是完全一样的。我们通常根据某些特定的规则来划分表, 比如根据用户的 ID来取模划分。

优点：

单库单表的数据保持在一定的量级,有助于性能的提高。

切分的表的结构相同,应用层改造较少,只需要增加路由规则即可。

提高了系统的稳定性和负载能力 。

缺点：

切分后,数据是分散的,很难利用数据库的 Join操作,跨库 Join性能较差。

分片事务的一致性难以解决。

数据扩容的难度和维护量极大。

我们就可以根据自己的业务需求来选择, 通常会同时使用这两种切分方式, 垂直切分更偏向于业务拆分的过程, 在技术上我们更关注水平切分的方案。

### 水平切分的路由和常见分片纬度

#### 水平切分的路由过程

我们在设计表时需要确定对表按照什么样的规则进行分库分表。 例如, 当有新用户登录时, 程序需要确定将此用户的信息添加到哪个表中; 同理, 在登录时我们需要通过用户的账号找到数据库中对应的记录, 所有这些都需要按照某一规则进行路由请求, 因为请求所需要的数据分布在不同的分片表中。

通过分库分表规则査找到对应的表和库的过程叫作路由。假设，分库分表的规则是 user\_id%4, 当用户新注册了一个帐号时,如果用户的 ID是123,我们就可以通过123%4=3确定此账号应该被保存在 User3表中。当 ID为123的用户登录时,我们可通过123 % 4 = 3计算后, 确定其被记录在 User3中 。

#### 水平切分的分片维度

##### 1)按照哈希切片

对数据的東个字段求哈希, 再除以分片总数后取模, 取模后相同的数据为一个分片的将数据分成多个分片的方法叫作哈希分片,合希分片常常应用于数据没有时效性的情况，这种切片方式的好处是数据切片比较均匀, 对数据压力分散的效果较好, 缺点是数据分片后, 对于査询需求需要进行聚合处理,

##### 2）按照时间切片

按照哈希切片不同, 这种方式是按照时间的范围将数据分布到不同的分片上的, 例如, 我们可以将交易数据按照月进行切片, 或者按照季度进行切片, 由交易数据的多少来决定按照什么样的时间周期对数据进行切片,,这种切片方式适用于有明显时间特点的数据,例如,距离现在1个季度的数据访问频繁，距离现在2个季度的数据不需要更新，距离现在3个季度的数据没有查询要求。就可以按时间进行分片，用不同的介质进行存放。

在实践中，按照哈希切片和按时间切片都很常见，而且有时会两种结合起来使用。

## 分库分表引发的问题

### 扩容与迁移

在分库分表后, 如果涉及的分片已经达到了承载数据的最大值, 就需要对集群进行扩容。 扩容是很麻烦的, 一般会成信地扩容。

### 分库分表维度导致的查询间题

在分库分表以后, 如果査询的标准是分片的主键, 则可以通过分片规则再次路由并查询。

但是对于其他主键的查询、范围査询、关联査询、查询结果排序等,并不是按照分库分表维度来査询的。例如,用户购买了商品,需要将交易记录保存下来,那么如果按照买家的纬度分表,则每个买家的交易记录都被保存在同一表中, 我们可以很快、 很方便地査到某个买家的购买情况, 但是某个商品被购买的交易数据很有可能分布在多张表中, 査找起来比较麻烦 。 反之, 按照商品维度分表, 则可以很方便地査找到该商品的购买情况, 但若要査找到买家的交易记录, 则会比较麻烦 。

所以常见的解决方式如下。

( 1 ) 在多个分片表查询后合并数据集, 这种方式的效率很低。

( 2 ) 记录两份数据, 一份按照买家纬度分表, 一份按照商品维度分表,,

( 3 ) 通过搜索引擎解决, 但如果实时性要求很高, 就需要实现实时搜索

实际上,在高并发的服务平台下,交易系统是专门做交易的,因为交易是核心服务,所以需要和査询系统分离, 査询一般通过其他系统进行, 数据也可能是冗余存储的。

在某电商交易平台下, 可能有买家査询自己在某一时间段的订单, 也可能有卖家査询自已在某一时间段的订单, 如果使用了分库分表方案, 则这两个需求是难以满足的, 因此, 通用的解决方案是, 在交易生成时生成一份按照买家分片的数据副本和一份按照卖家分片的数据副本,查询时分别满足之前的两个需求,因此,查询的数据和交易的数据可能是分别存储的,并从不同的系统提供接口。

另外,在电商系统中,在一个交易订单生成后,一般需要引用到订单中交易的商品实体如果简单地引用,若商品的金额等信息发生变化,则会导致原订单上的商品信息也会发生变化这样买家会很疑惑。因此,通用的解决方案是在交易系统中存储商品的快照,在査询交易时使用交易的快照,因为快照是个静态数据,永远都不会更新,所以解决了这个问题。可见查询的问题最好在单独的系统中使用其他技术来解决,而不是在交易系统中实现各类查询功能。当然当然,也可以通过对商品的变更实施版本化,在交易订单中引用商品的版本信息,在版本更新时保留商品的旧版本,这也是一种不错的解决方案。

#### 3.跨库事务难以实现

要避免在一个事务中同时修改数据库db0和数据库dbl中的表,因为操作起来很复杂,对效率也会有一定的影响。

#### 4.同组数据跨库问题

要尽量把同一组数据放到同一台数据库服务器上,不但在某些场景下可以利用本地事务的强一致性,还可以使这组数据自治。以电商为例,我们的应用有两个数据库db0和dbl,分库分表后,按照id维度,将卖家A的交易信息存放到db0中。当数据库dbl挂掉时,卖家A的交易信息不受影响,依然可以正常使用。也就是说,要避免数据库中的数据依赖另一数据库中的数据。