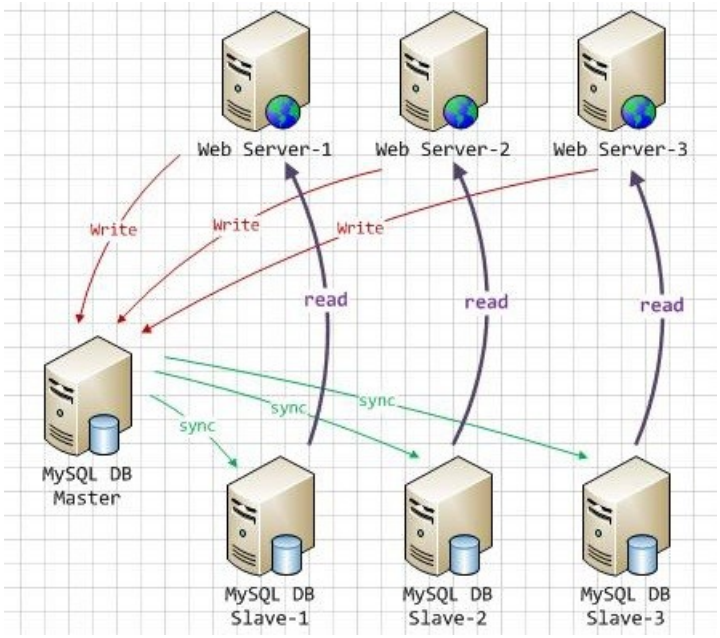
mysql-proxy 实现读写分离

文章来自整理:http://blog.jobbole.com/94606/

其中 Amoeba for MySQL 也是实现读写分离



环境描述:

操作系统：CentOS6.5 32 位

主服务器 Master：192.168.179.146

从服务器 Slave：192.168.179.147

调度服务器 MySQL-Proxy：192.168.179.142

由于电脑配置不行，安装了三台虚拟机，就卡死了，只能将就一下,由于是一主

一从，所以，导致读写都在 master 上，有机会，再弄两台 slave 来测试

一.mysql 主从复制，参考：http://www.cnblogs.com/lin3615/p/5679828.html

二、mysql-proxy 实现读写分离

1、安装 mysql-proxy

实现读写分离是有 lua 脚本实现的，现在 mysql-proxy 里面已经集成，无需再安装

下载：http://dev.mysql.com/downloads/mysql-proxy/ 一定要下载对应的版本

```
tar zxvf mysql-proxy-0.8.5-linux-glibc2.3-x86-32bit.tar.gz
mv mysql-proxy-0.8.5-linux-glibc2.3-x86-32bit /usr/local/mysql-proxy
```

2、配置 mysql-proxy，创建主配置文件

```
cd /usr/local/mysql-proxy
mkdir lua #创建脚本存放目录
mkdir logs #创建日志目录
cp share/doc/mysql-proxy/rw-splitting.lua ./lua #复制读写分离配置文件
cp share/doc/mysql-proxy/admin-sql.lua ./lua #复制管理脚本
vi /etc/mysql-proxy.cnf    #创建配置文件
[mysql-proxy]
```

```
user=root #运行 mysql-proxy 用户
admin-username=lin3615 #主从 mysql 共有的用户
admin-password=123456 #用户的密码
proxy-address=192.168.179.142:4040 #mysql-proxy 运行 ip 和端口，不加端口，默认 4040
proxy-read-only-backend-addresses=192.168.179.147 #指定后端从 slave 读取数据
proxy-backend-addresses=192.168.179.146 #指定后端主 master 写入数据
proxy-lua-script=/usr/local/mysql-proxy/lua/rw-splitting.lua #指定读写分离配置文件
位置
admin-lua-script=/usr/local/mysql-proxy/lua/admin-sql.lua #指定管理脚本
log-file=/usr/local/mysql-proxy/logs/mysql-proxy.log #日志位置
log-level=info #定义 log 日志级别，由高到低分别有(error|warning|info|message|debug)
daemon=true    #以守护进程方式运行
keepalive=true #mysql-proxy 崩溃时，尝试重启
#保存退出！
chmod 660 /etc/mysql-porxy.cnf
```

3、修改读写分离配置文件

```
vim /usr/local/mysql-proxy/lua/rw-splitting.lua
if not proxy.global.config.rwsplit then
 proxy.global.config.rwsplit = {
  min_idle_connections = 1, #默认超过 4 个连接数时，才开始读写分离，改为 1
  max_idle_connections = 1, #默认 8，改为 1
  is_debug = false
 }
end
```

4、启动 mysql-proxy

```
/usr/local/mysql-proxy/bin/mysql-proxy --defaults-file=/etc/mysql-proxy.cnf
netstat -tupln | grep 4000 #已经启动

killall -9 mysql-proxy #关闭 mysql-proxy 使用
```

5、测试读写分离

(1).在主服务器创建 proxy 用户用于 mysql-proxy 使用，从服务器也会同步这个操作

```
mysql> grant all on *.* to 'lin3615'@'192.168.179.142' identified by '123456';
```

(2).使用客户端连接 mysql-proxy

```
mysql -u lin3615 -h 192.168.179.142 -P 4040 -p123456
```

接下来就按基本的 curd 执行即可,由于只有一台 slave，测试时，每次读写都是从 master，电脑性能无法开四台虚拟机，所以有机会，再测试多台 slave，看下是否 OK

读写分离，延迟是个大问题

在 slave 服务器上执行 show slave status,
可以看到很多同步的参数,要注意的参数有：
Master_Log_File:slave 中的 I/O 线程当前正在读取的 master 服务器二进制式日志文件名.
Read_Master_Log_Pos:在当前的 master 服务器二进制日志中，slave 中的 I/O 线程已经读取的位置
Relay_Log_File:SQL 线程当前正在读取与执行中继日志文件的名称
Relay_Log_Pos:在当前的中继日志中，SQL 线程已读取和执行的位置
Relay_Master_Log_File:由 SQL 线程执行的包含多数近期事件的 master 二进制日志文件的名称

Slave_IO_Running:I/O 线程是否被启动并成功连接到 master
Slave_SQL_Running:SQL 线程是否被启动
Seconds_Behind_Master:slave 服务器 SQL 线程和从服务器 I/O 线程之间的差距，单位为秒计

slave 同步延迟情况出现：
1.Seconds_Behind_Master 不为了，这个值可能会很大
2.Relay_Master_Log_File 和 Master_Log_File 显示 bin-log 的编号相差很大，说明 bin-log 在 slave 上没有及时同步，所以近期执行的 bin-log 和当前 I/O 线程所读的 bin-log 相差很大
3.mysql 的 slave 数据库目录下存在大量的 mysql-relay-log 日志，该日志同步完成之后就会被系统自动删除，存在大量日志，说明主从同步延迟很厉害

mysql 主从同步延迟原理
mysql 主从同步原理
主库针对读写操作，顺序写 binlog，从库单线程去主库读"写操作的 binlog",从库取到 binlog 在本地原样执行(随机写),来保证主从数据逻辑上一致.
mysql 的主从复制都是单线程的操作，主库对所有 DDL 和 DML 产生 binlog，binlog 是顺序写，所以效率很高，slave 的 Slave_IO_Running 线程到主库取日志，效率比较高，下一步问题来 了，slave 的 slave_sql_running 线程将主库的 DDL 和 DML 操作在 slave 实施。DML，DDL 的 IO 操作是随即的，不能顺序的，成本高很多，还有可能 slave 上的其他查询产生 lock，由于 slave_sql_running 也是单线程的，所以 一个 DDL 卡住了，需求需求执行一段时间，那么所有之后的 DDL 会等待这个 DDL 执行完才会继续执行，这就导致了延迟.由于 master 可以并发，Slave_sql_running 线程却不可以，所以主库执行 DDL 需求一段时间，在 slave 执行相同的 DDL 时，就产生了延迟.

主从同步延迟产生原因
当主库的 TPS 并发较高时，产生的 DDL 数量超过 Slave 一个 sql 线程所能承受的范围，那么延迟就产生了，当然还有就是可能与 slave 的大型 query 语句产生了锁等待
首要原因：数据库在业务上读写压力太大，CPU 计算负荷大，网卡负荷大，硬盘随机 IO 太高
次要原因：读写 binlog 带来的性能影响，网络传输延迟
主从同步延迟解决方案
架构方面
1.业务的持久化层的实现采用分库架构，mysql 服务可平行扩展分散压力
2.单个库读写分离，一主多从，主写从读，分散压力。
3.服务的基础架构在业务和 mysql 之间加放 cache 层
4.不同业务的 mysql 放在不同的机器
5.使用比主加更了的硬件设备作 slave
反正就是 mysql 压力变小，延迟自然会变小

硬件方面：
采用好的服务器

mysql 主从同步加速
1、sync_binlog 在 slave 端设置为 0
2、–logs-slave-updates 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。
3、直接禁用 slave 端的 binlog
4、slave 端，如果使用的存储引擎是 innodb，innodb_flush_log_at_trx_commit =2

从文件系统本身属性角度优化
master 端
修改 linux、Unix 文件系统中文件的 etime 属性， 由于每当读文件时 OS 都会将读取操作发生的时间回写到磁盘上，对于读操作频繁的数据库文件来说这是没必要的，只会增加磁盘系统的负担影响 I/O 性能。可以 通过设置文件系统的 mount 属性，组织操作系统写 atime 信息，在 linux 上的操作为：
打开/etc/fstab，加上 noatime 参数
/dev/sdb1 /data reiserfs noatime 1 2
然后重新 mount 文件系统
#mount -oremount /data

主库是写，对数据安全性较高，比如 sync_binlog=1，innodb_flush_log_at_trx_commit = 1 之类的设置是需要的
而 slave 则不需要这么高的数据安全，完全可以讲 sync_binlog 设置为 0 或者关闭
binlog，innodb_flushlog 也可以设置为 0 来提高 sql 的执行效率
1、sync_binlog=1 o
MySQL 提供一个 sync_binlog 参数来控制数据库的 binlog 刷到磁盘上去。
默认，sync_binlog=0，表示 MySQL 不控制 binlog 的刷新，由文件系统自己控制它的缓存的刷新。这时候的性能是最好的，但是风险也是最大的。一旦系统 Crash，在 binlog_cache 中的所有 binlog 信息都会被丢失。
如 果 sync_binlog>0，表示每 sync_binlog 次事务提交，MySQL 调用文件系统的刷新操作将缓存刷下去。最安全的就是 sync_binlog=1 了，表示每次事务提交，MySQL 都会把 binlog 刷下去，是最安全但是性能损耗最大的设置。这样的话，在数据库所在的主机 操作系统损坏或者突然掉电的情况下，系统才有可能丢失 1 个事务的数据。
但是 binlog 虽然是顺序 IO，但是设置 sync_binlog=1，多个事务同时提交，同样很大的影响 MySQL 和 IO 性能。
虽然可以通过 group commit 的补丁缓解，但是刷新的频率过高对 IO 的影响也非常大。对于高并发事务的系统来说，
“sync_binlog”设置为 0 和设置为 1 的系统写入性能差距可能高达 5 倍甚至更多。
所以很多 MySQL DBA 设置的 sync_binlog 并不是最安全的 1，而是 2 或者是 0。这样牺牲一定的一致性，可以获得更高的并发和性能。
默 认情况下，并不是每次写入时都将 binlog 与硬盘同步。因此如果操作系统或机器(不仅仅是 MySQL 服务器)崩溃，有可能 binlog 中最后的语句丢失 了。要想防止这种情况，你可以使用 sync_binlog 全局变量(1 是最安全的值，但也是最慢的)，使 binlog 在每 N 次 binlog 写入后与硬盘同 步。即使 sync_binlog 设置为 1,出现崩溃时，也有可能表内容和 binlog 内容之间存在不一致性。

2、innodb_flush_log_at_trx_commit （这个很管用）
抱怨 Innodb 比 MyISAM 慢 100 倍？那么你大概是忘了调整这个值。默认值 1 的意思是每一次事务提交或事务外的指令都需要把日志写入（flush）硬盘，这是很费时的。特别是使用电 池供电缓存（Battery backed up cache）时。设成 2 对于很多运用，特别是从 MyISAM 表转过来的是可以的，它的意思是不写入硬盘而是写入系统缓存。
日志仍然会每秒 flush 到硬 盘，所以你一般不会丢失超过 1-2 秒的更新。设成 0 会更快一点，但安全方面比较差，即使 MySQL 挂了也可能会丢失事务的数据。而值 2 只会在整个操作系统 挂了时才可能丢数据。

3、ls(1) 命令可用来列出文件的 atime、ctime 和 mtime。
atime 文件的 access time 在读取文件或者执行文件时更改的
ctime 文件的 create time 在写入文件，更改所有者，权限或链接设置时随 inode 的内容更改而更改
mtime 文件的 modified time 在写入文件时随文件内容的更改而更改
ls -lc filename 列出文件的 ctime
ls -lu filename 列出文件的 atime
ls -l filename 列出文件的 mtime
stat filename 列出 atime，mtime，ctime
atime 不一定在访问文件之后被修改
因为：使用 ext3 文件系统的时候，如果在 mount 的时候使用了 noatime 参数那么就不会更新 atime 信息。

这三个 time stamp 都放在 inode 中.如果 mtime,atime 修改,inode 就一定会改, 既然 inode 改了,那 ctime 也就跟着改了.
之所以在 mount option 中使用 noatime, 就是不想 file system 做太多的修改, 而改善读取效能

4.进行分库分表处理，这样减少数据量的复制同步操作