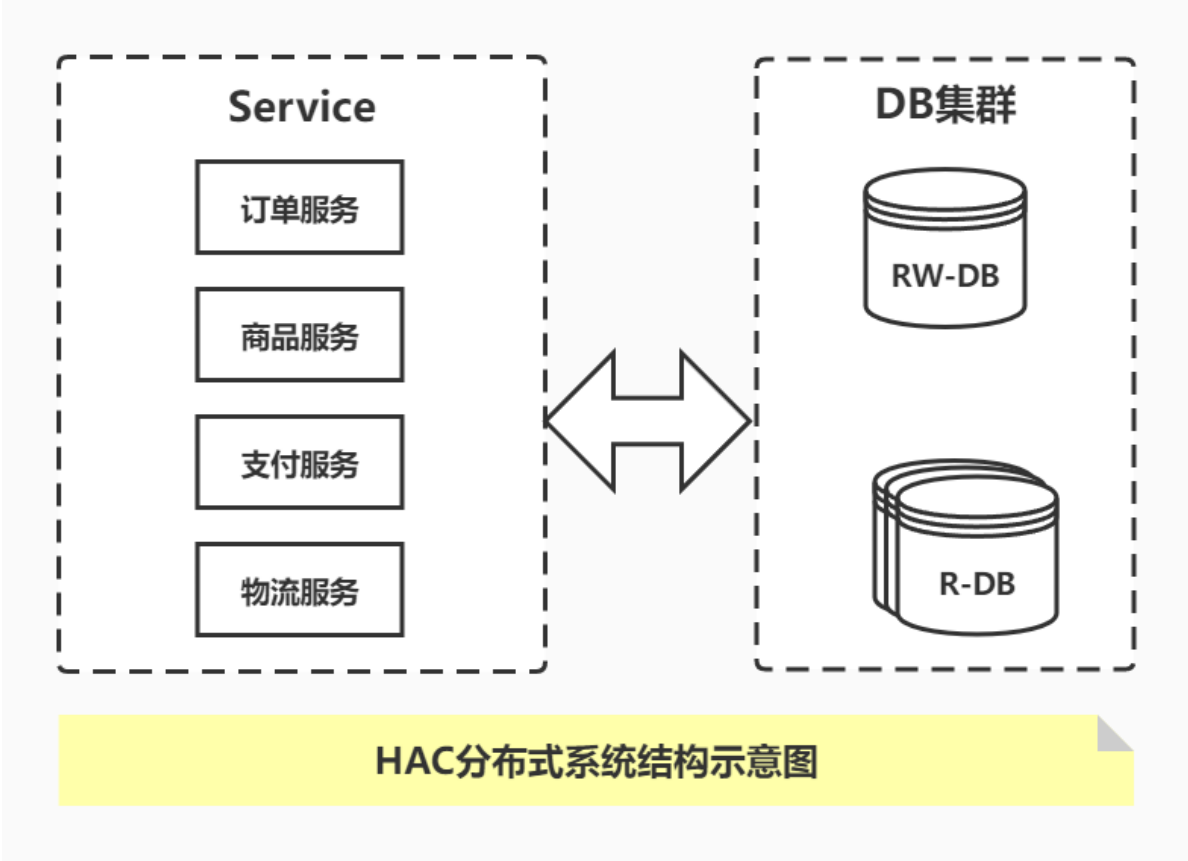
MySQL高可用集群篇

1. 集群搭建之主从复制

。。。

2. 集群搭建之读写分离

2.1 读写分离的理解

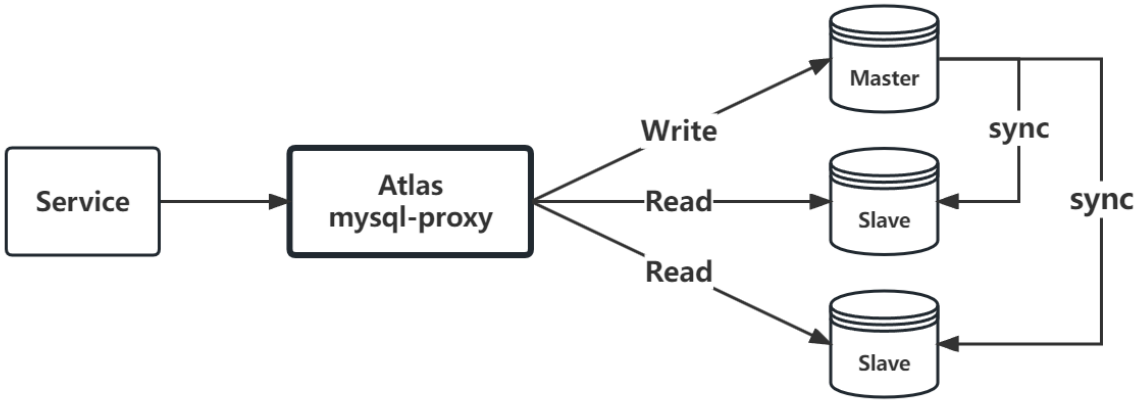


HAC：High Availability Cluster高可用集群

2.2 案例：Atlas配置读写分离

2.2.1 简介

Atlas是由`Qihoo 360`公司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。它在MySQL官方推出的`MySQL-Proxy 0.8.2版本`的基础上，修改了大量bug，添加了很多功能特性。该项目在360公司内部得到了广泛应用，很多MySQL业务已经接入了Atlas平台，每天承载的读写请求数达几十亿条。同时，有超过50家公司在生产环境中部署了Atlas。



2.2.2 安装

```
wget https://github.com/Qihoo360/Atlas/releases/download/2.2.1/Atlas-2.2.1.el6.x86_64.rpm
```

下载好了之后，进行安装

```
# 卸载rpm包
rpm -e Atlas

# 安装rpm包
rpm -ivh Atlas-2.2.1.el6.x86_64.rpm
```

安装好了，它会默认在"`/usr/local/mysql-proxy`"下给你生成4个文件夹，以及需要配置的文件，如下：

```
cd /usr/local/mysql-proxy
```

- bin目录下放的都是可执行文件
    1. "encrypt"是用来生成MySQL密码加密的，在配置的时候会用到
    2. "mysql-proxy"是MySQL自己的读写分离代理
    3. "mysql-proxyd"是360弄出来的，后面有个"d"，服务的启动、重启、停止。都是用他来执行的
- conf目录下放的是配置文件
    1. "test.cnf"只有一个文件，用来配置代理的，可以使用vim来编辑
- lib目录下放的是一些包，以及Atlas的依赖
- log目录下放的是日志，如报错等错误信息的记录

2.2.3 配置

进入bin目录，使用`encrypt`来对数据库的密码进行加密，我的MySQL数据的用户名是root，密码是root，我需要对密码进行加密

```
/usr/local/mysql-proxy/bin/encrypt root
DAJnl8cVzy8=
```

配置Atlas，使用vim进行编辑

```
vim /usr/local/mysql-proxy/conf/test.cnf
```

进入后，可以在Atlas进行配置，360写的中文注释都很详细，根据注释来配置信息，其中比较重要，需要说明的配置如下：

> 这是用来登录到Atlas的管理员的账号与密码，与之对应的是"#Atlas监听的管理接口IP和端口"，也就是说需要设置管理员登录的端口，才能进入管理员界面，默认端口是2345，也可以指定IP登录，指定IP后，其他的IP无法访问管理员的命令界面。方便测试，我这里没有指定IP和端口登录。

```
# 管理接口的用户名
admin-username = hero
# 管理接口的密码
admin-password = hero
```

这是用来配置主数据的地址与从数据库的地址，这里配置的主数据库是132，从数据库是133

```
#Atlas后端连接的MySQL主库的IP和端口，可设置多项，用逗号分隔
proxy-backend-addresses = 172.17.187.78:3306

#Atlas后端连接的MySQL从库的IP和端口，@后面的数字代表权重，用来作负载均衡，若省略则默认为1，可设置多项，用逗号分隔
proxy-read-only-backend-addresses = 172.17.187.78:3306@1,172.17.187.79:3306@2
```

这个是用来配置MySQL的账户与密码的，我的MySQL的用户是root,密码是hello,刚刚使用Atlas提供的工具生成了对应的加密后密码。

```
#用户名与其对应的加密过的MySQL密码，密码使用PREFIX/bin目录下的加密程序encrypt加密，下行的user1和user2为示例，将其替换为你的MySQL的用户名和加密密码!
pwds = root:qyMGHucYPhGZnKb0g+dxdA==
```

这是设置工作接口与管理接口的，如果ip设置的"0.0.0.0"就是说任意IP都可以访问这个接口，当然也可以指定IP和端口，方便测试我这边没有指定，工作接口的用户名密码与MySQL的账户对应的，管理员的用户密码与上面配置的管理员的用户密码对应。

```
#Atlas监听的工作接口IP和端口
proxy-address = 0.0.0.0:1234

#Atlas监听的管理接口IP和端口
admin-address = 0.0.0.0:2345
```

2.2.4 启动

```
/usr/local/mysql-proxy/bin/mysql-proxyd test start
OK: MySQL-Proxy of test is started
```

使用如下命令，进入Atlas的管理模式`mysql -h127.0.0.1 -P2345 -uuser -ppwd`，能进去说明Atlas正常运行，因为它会把自己当成一个MySQL数据库，所以在不需要数据库环境的情况下，也可以进入到MySQL数据库模式。

```
[root@localhost bin]# mysql -h127.0.0.1 -P2345 -uhero -phero
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.99-agent-admin

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>
```

可以访问“help”表，来看MySQL管理员模式都能做些什么。可以使用SQL语句来访问

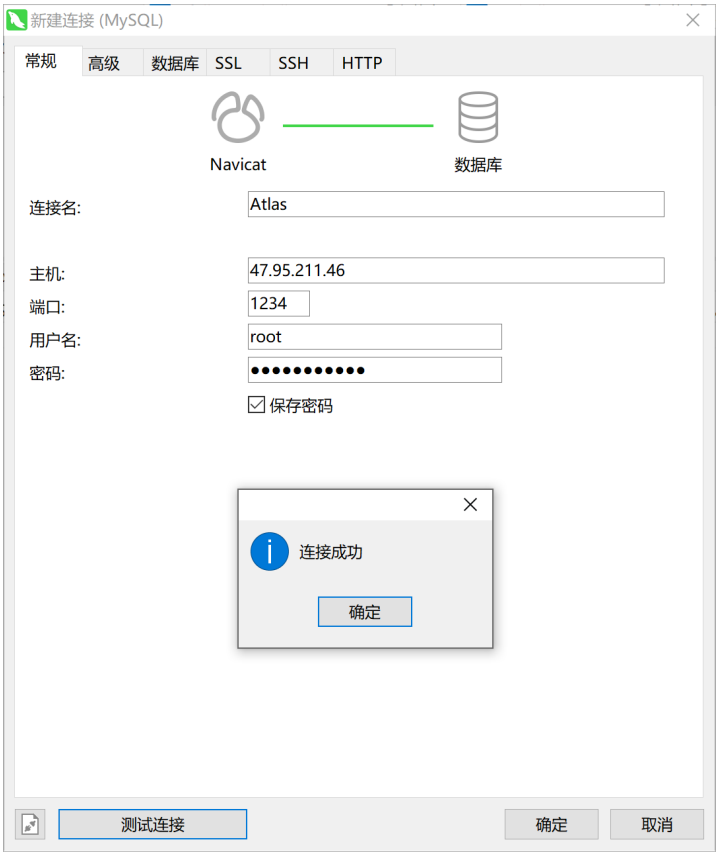
```
mysql> select * from help;
+----------------------------+-----------------------------------------------
-----------+
| command                    | description
          |
+----------------------------+-----------------------------------------------
-----------+
| SELECT * FROM help         | shows this help
          |
| SELECT * FROM backends     | lists the backends and their state
          |
| SET OFFLINE $backend_id    | offline backend server, $backend_id is
backend_ndx's id |
| SET ONLINE $backend_id     | online backend server, ...
          |
| ADD MASTER $backend        | example: "add master 127.0.0.1:3306", ...
          |
| ADD SLAVE $backend         | example: "add slave 127.0.0.1:3306", ...
          |
| REMOVE BACKEND $backend_id | example: "remove backend 1", ...
          |
| SELECT * FROM clients      | lists the clients
          |
| ADD CLIENT $client         | example: "add client 192.168.1.2", ...
          |
| REMOVE CLIENT $client      | example: "remove client 192.168.1.2", ...
          |
```

```
| SELECT * FROM pwds          | lists the pwds
            |
| ADD PWD $pwd                | example: "add pwd user:raw_password", ...
            |
| ADD ENPWD $pwd              | example: "add enpwd user:encrypted_password",
...         |
| REMOVE PWD $pwd             | example: "remove pwd user", ...
            |
| SAVE CONFIG                 | save the backends to config file
            |
| SELECT VERSION              | display the version of Atlas
            |
+-----------------------------+-----------------------------------------------
-----------+
16 rows in set (0.00 sec)

mysql>
```

也可以使用工作接口来访问，使用命令`mysql -h127.0.0.1 -P1234 -uroot -proot`

```
[root@hero03 conf]# mysql -h127.0.0.1 -P1234 -uroot -proot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.81-log MySQL Community Server (GPL)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>
```

如果工作接口可以进入了，就可以在Windows平台下，使用Navicat来连接数据库，填写对应的host，Port，用户名，密码就可以。



2.2.5 测试

使用语句`mysql -uroot -proot -P1234 --protocol=tcp -e"select @@hostname"`来测试读写分离。

```
mysql -uroot -proot -P1234 --protocol=tcp -e"select @@hostname"
```

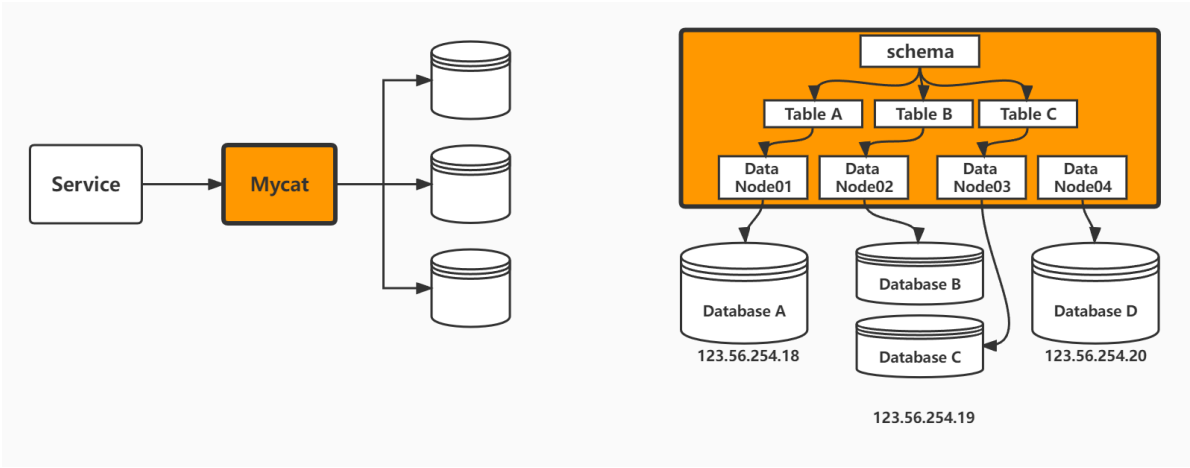
2.3 案例：Mycat配置读写分离

2.3.1 Mycat简介

什么是Mycat?

Mycat是一个数据库中间件，支持**读写分离**、分库分表、还支持水平分片与垂直分片。

- 水平分片：一个表格的数据分割到多个节点上，按照行分割
- 垂直分片：一个数据库中多个表格A，B，C，A存储到节点1上，B存储到节点2上，C存储到节点3上。



Mycat核心概念：

Mycat通过定义表的**分片规则**来实现分片，每个表格可以捆绑一个分片规则，每个分片规则指定一个分片字段并绑定一个函数，来实现动态分片算法。

- **Schema**：**逻辑库**与MySQL中的Database数据库对应，一个逻辑库中定义了所包括的Table。
- **Table**：表，即**物理数据库中存储的某一张表**，与传统数据库不同，这里的表格需要声明其所存储的逻辑数据节点DataNode。**在此指定表的分片规则。**
- **DataNode**：Mycat的**逻辑数据节点**，是存放table的具体物理节点，也称之为分片节点，通过DataHost来关联到后端某个具体数据库上
- **DataHost**：定义某个**物理库的访问地址**，用于捆绑到Datanode上

2.3.2 下载安装

> 注意：需要jdk

下载Mycat

```
wget http://dl.mycat.org.cn/1.6.7.1/Mycat-server-1.6.7.1-release-20190627191042-linux.tar.gz
```

解压缩

```
tar -zxf Mycat-server-1.6.7.1-release-20190627191042-linux.tar.gz
```

进行mycat/bin目录，启动Mycat

```
/root/mycat/bin/mycat start
/root/mycat/bin/mycat stop
/root/mycat/bin/mycat restart
/root/mycat/bin/mycat status
```

访问Mycat

使用MySQL的客户端直接连接mycat服务。默认服务端口为【8066】。

```
mysql -uroot -p123456 -h127.0.0.1 -P8066
```

2.3.3 配置Mycat

1）配置端口和密码

修改server.xml，配置端口和密码：

```
<!--修改mycat服务端口-->
<property name="serverPort">8067</property>
<user name="root" defaultAccount="true">
    <property name="password">123456</property>
    <property name="schemas">mycat</property>
</user>

<user name="user">
    <property name="password">user</property>
    <property name="schemas">mycat</property>
    <property name="readOnly">true</property>
</user>
```

2）配置读写分离

修改schema.xml，配置读写分离：

```
<?xml version="1.0"?>
<!DOCTYPE mycat:schema SYSTEM "schema.dtd">
<mycat:schema xmlns:mycat="http://io.mycat/">

    <schema name="mycat" checkSQLschema="true" sqlMaxLimit="100">
        <table name="tb_user" dataNode="dn1" />
        <table name="t1" dataNode="dn1" />
        <table name="t2" dataNode="dn1" />
        <table name="t3" dataNode="dn1" />
    </schema>
    <dataNode name="dn1" dataHost="host1" database="mycat" />
    <dataHost name="host1" maxCon="1000" minCon="10" balance="1"
              writeType="0" dbType="mysql" dbDriver="native" switchType="1"
 slaveThreshold="100">
        <heartbeat>select user()</heartbeat>
        <!-- can have multi write hosts -->
        <writeHost host="hostM1" url="172.17.187.78:3306" user="root"
password="root">
            <!-- can have multi read hosts -->
            <readHost host="hostS2" url="172.17.187.79:3306" user="root"
password="root" />
        </writeHost>
        <!-- <writeHost host="hostM2" url="localhost:3316" user="root"
password="123456"/> -->
```

```
    </dataHost>
</mycat:schema>
```

或者

```
<?xml version="1.0"?>
<!DOCTYPE mycat:schema SYSTEM "schema.dtd">
<mycat:schema xmlns:mycat="http://io.mycat/">

    <schema name="mycat" checkSQLschema="false" sqlMaxLimit="100">
        <table name="tb_user" dataNode="dn1" />
        <table name="t1" dataNode="dn1" />
        <table name="t2" dataNode="dn1" />
        <table name="t3" dataNode="dn1" />
    </schema>
    <dataNode name="dn1" dataHost="host1" database="mycat" />
    <dataHost name="host1" maxCon="1000" minCon="10" balance="1"
              writeType="0" dbType="mysql" dbDriver="native" switchType="1"
 slaveThreshold="100">
        <heartbeat>select user()</heartbeat>
        <!-- can have multi write hosts -->
        <writeHost host="hostM1" url="123.57.135.5:3306" user="root"
password="root">
          <!-- can have multi read hosts -->
          <!--<readHost host="hostS2" url="47.95.211.46:3306" user="root"
password="root" />-->
        </writeHost>
        <!-- <writeHost host="hostM2" url="localhost:3316" user="root"
password="123456"/> -->
        <writeHost host="hostM2" url="47.95.211.46:3306" user="root"
password="root"/>
    </dataHost>
</mycat:schema>
```

2.4.4 读写分离测试

1）创建表

```
use mycat;
CREATE TABLE tb_user (
  login_name VARCHAR ( 32 ) COMMENT '登录名',
  user_id BIGINT COMMENT '用户标识',
  TYPE INT COMMENT '用户类型 1 商家，2买家',
  passwd VARCHAR ( 128 ) COMMENT '密码',
PRIMARY KEY ( user_id )
);
```

2）插入数据与查询数据

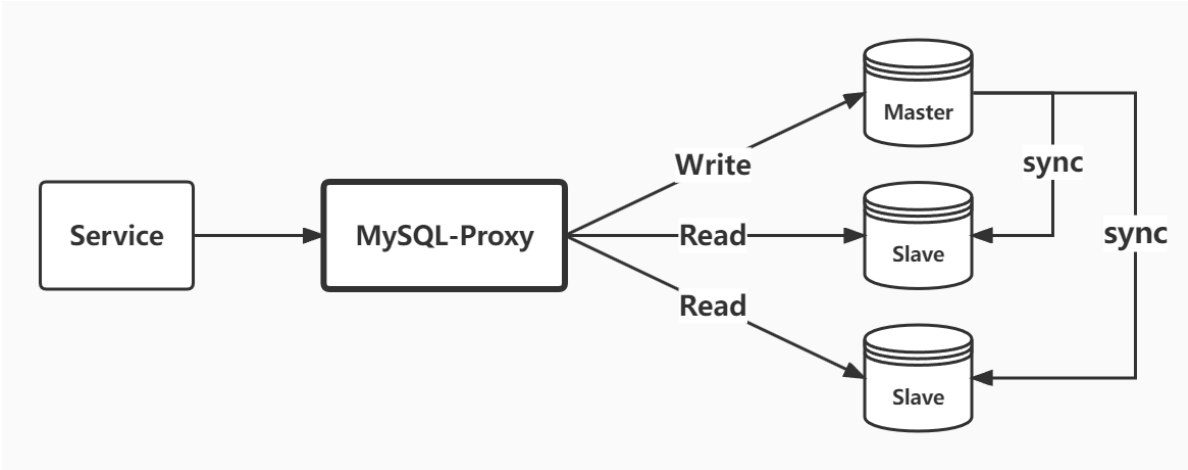
```
# 新增
INSERT INTO `tb_user`(`login_name`,`user_id`,`TYPE`,`passwd`) VALUES ('name-1',22,1,'passwd-A');
INSERT INTO `tb_user`(`login_name`,`user_id`,`TYPE`,`passwd`) VALUES ('name-2',22,1,'passwd-A');
# 查
select * from tb_user;
```

3）测试

```
SELECT @@server_id;
```

2.4 案例：MySQL Proxy配置读写分离

mysql-proxy是mysql官方提供的mysql中间件服务，课提供读写分离的功能。



MySQLProxy虽然可以实现读写分离的操作，但是MySQLProxy官方并没有推出稳定版，其中的坑还是挺多的，并不推荐在生产环境使用。官方推荐使用MySQLRouter，所以关于MySQLProxy的使用大家做为了解内容即可。

2.4.1 MySQL-Proxy安装

- 下载

```
wget https://cdn.mysql.com/archives/mysql-proxy/mysql-proxy-0.8.5-linux-el6-x86-64bit.tar.gz
```

- 解压缩

```
tar -zxf mysql-proxy-0.8.5-linux-el6-x86-64bit.tar.gz -C mysql-proxy-0.8.5
```

2.4.2 MySQL-Proxy配置

- 创建mysql-proxy.conf文件

```
[mysql-proxy]
user=root # 运行mysql-proxy用户
admin-username=root      # 主从mysql共有的用户
admin-password=hero      # admin密码
proxy-address=124.221.171.208:4040  # mysql-proxy运行ip和端口，不加端口，默认4040
proxy-backend-addresses=124.221.171.208:3306  # 指定后端主master写入数据
proxy-read-only-backend-addresses=43.142.77.65:3306  # 指定后端从slave读取数据
proxy-lua-script=./share/doc/mysql-proxy/rw-splitting.lua   # lua位置，参见上面的版本信息
log-file=./logs/mysql-proxy.log   # 日志文件存储路径
log-level=debug
keepalive=true   # 保持连接启动进程会有2个， 一号进程用来监视二号进程
daemon=true   # mysql-proxy以守护进程方式运行
```

- 修改mysql-proxy.conf文件的权限

```
chmod 660 mysql-proxy.conf
```

- 修改rw-splitting.lua脚本

```
vim share/doc/mysql-proxy/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 测试需要
    max_idle_connections = 1, # 默认8，改为1测试需要
    is_debug = false
  }
end
```

2.4.3 MySQL-Proxy启动域测试

- 启动命令

```
nohup bin/mysql-proxy --defaults-file=mysql-proxy.conf > mysql-proxy.out 2>&1 &
# 注意：如果没有配置profile文件的环境变量，则需要去拥有mysql-proxy命令的目录通过./mysql-proxy进行启动。
```

- 在其他客户端，通过mysql命令去连接MySQL Proxy机器

```
mysql -uroot -proot -h124.221.171.208 -P4040
```

2.4.5 小结

MySQL Proxy虽然可以实现读写分离的操作，但是MySQL Proxy的坑还是挺多的，并不推荐在生产环境使用。官方推荐使用MySQL Router，所以关于MySQL Proxy的使用大家做为了解内容即可。

2.5 案例：MySQL Router配置读写分离

2.5.1 简介

MySQL Router最早是作为MySQL-Proxy的替代方案出现的。作为一个轻量级中间件，MySQL Router可在应用程序和后端MySQL服务器之间提供透明路由和负载均衡，从而有效提高MySQL数据库服务的高可用性与可伸缩行。

MySQL Router 2.0是其初始版本，适用于MySQL Fabric用户，但已被弃用，不再支持。

MySQL Router 2.1为支持MySQL InnoDB Cluster而引入，**MySQL Router 8.0则是MySQL Router 2.1上的扩展**，版本号与MySQL服务器版本号保持一致。即Router 2.1.5作为Router 8.0.3（以及MySQL Server 8.0.3）发布，2.1.x分支被8.0.x取代。这两个分支完全兼容。当前最新版本为8.0.17，MySQL强烈建议使用Router 8与MySQL Server 8和5.7一起使用。

2.5.2 下载安装

```
wget http://ftp.iij.ad.jp/pub/db/mysql/Downloads/MySQL-Router/mysql-router-8.0.20-el7-x86_64.tar.gz
```

MySQL Router的安装过程依赖于所使用的操作系统和安装介质，二进制包的安装通常非常简单，而源码包则需要先编译再安装。例如在Linux上的安装最新的MySQL Router二进制包，只需要用mysql用户执行一条解压命令就完成了：

```
tar zxf mysql-router-8.0.20-el7-x86_64.tar.gz
```

2.5.3 配置

创建mysqlrouter.conf并写入如下内容：

```
[logger]
level = INFO

[routing:secondary]
bind_address = localhost
bind_port = 7001
destinations = 172.17.187.78:3306,172.17.187.79:3306
routing_strategy = round-robin

[routing:primary]
bind_address = localhost
bind_port = 7002
destinations = 172.17.187.78:3306
routing_strategy = first-available
```

1. 这里设置了两个路由策略：
    - 通过本地7001端口，配置读取服务，循环连接到192.168.68.132:3306、192.168.68.133:3306、192.168.68.134:3306三个实例，由round-robin路由策略所定义；
    - 通过本地7002端口，配置写入服务，并设置首个可用策略。
        - 首个可用策略：使用目标列表中的第一个可用服务器，即当192.168.68.132:3306可用时，所有7002端口的连接都转发到它，否则转发到后面的服务器，以此类推。Router不会检查数据包，也不会根据分配的策略或模式限制连接
2. 因此应用程序可以据此确定将读写请求发送到不同的服务器。
3. 本例中可将读请求发送到本地7001端口，将读负载均衡到三台服务器。同时将写请求发送到7002，这样只写一个服务器，从而实现的读写分离。

2.5.4 启动并测试

进入mysql-router/bin目录下，启动mysqlrouter：

```
/root/mysql-router-8.0.20-el7-x86_64/bin/mysqlrouter -c /root/mysql-router-8.0.20-el7-x86_64/conf/mysqlrouter.conf &
```

测试：

```
mysql -uroot -proot -P7001 --protocol=tcp -e"select @@hostname"
```

由上可见，发送到本地7001端口的请求，被循环转发到三个服务器，而发送到本地7002端口的请求，全部被转发到192.168.68.132:3306。

routing_strategy是MySQL Router的核心选项，从8.0.4版本开始引入，当前有效值为first-available、next-available、round-robin、round-robin-with-fallback。

顾名思义，该选项实际控制路由策略，即客户端请求最终连接到哪个MySQL服务器实例。相对于以前版本mode的选项，routing_strategy选项更为灵活，并且不能同时设置routing_strategy和mode，静态路由的设置只能选择其中之一。对于InnoDB Cluster而言，该设置时可选的，缺省使用round-robin策略。

- **round-robin**：每个新连接都以循环方式连接到下一个可用的服务器，以实现负载平衡。
- **round-robin-with-fallback**：用于InnoDB Cluster。每个新的连接都以循环方式连接到下一个可用的secondary服务器。如果secondary服务器不可用，则以循环方式使用primary服务器。
- **first-available**：新连接从目标列表路由到第一个可用服务器。如果失败，则使用下一个可用的服务器，如此循环，直到所有服务器都不可用为止。
- **next-available**：与first-available类似，新连接从目标列表路由到第一个可用服务器。与first-available不同的是，如果一个服务器被标记为不可访问，那么它将被丢弃，并且永远不会再次用作目标。重启Router后，所有被丢弃服务器将再次可选。

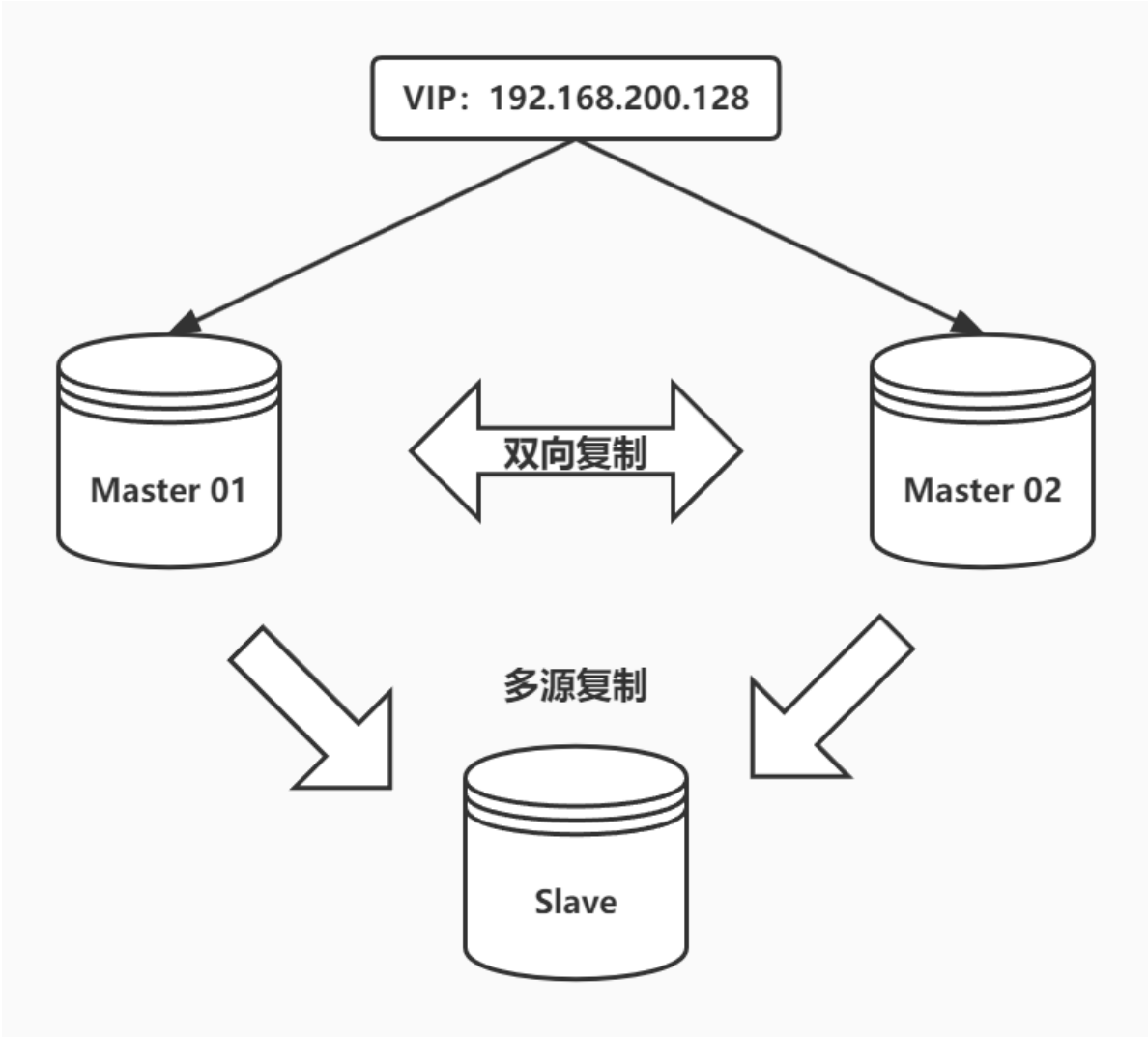
3. 高可用集群解决方案：基于主从复制

双节点主从 + keepalived/heartbeat方案，一般来说，中小型规模的时候，采用这种架构是最省事的。两个节点可以采用简单的**一主一从**模式，或者**双主**模式，并且放置于**同一个VLAN**中，在Master节点发生故障后，利用keepalived/heartbeat的高可用机制实现快速切换到slave节点。

在这个方案里，有几个需要注意的地方：

- **主键冲突：**把两个节点的**auto_increment_increment**（自增起始值）和**auto_increment_offset**（自增步长）设成不同值。
    - 其目的是为了避免Master节点意外宕机时，可能会有部分binlog未能及时复制到Slave上被应用，从而会导致Slave新写入数据的自增值和原先Master上冲突了，因此一开始就使其错开；也可以采用其他主键生成机制代替主键自增，例如：雪花算法。
- **热备节点（Slave）硬件配置不能低于Master节点：**Slave节点服务器配置不要太差，否则更容易导致复制延迟。
    - 如果对延迟问题很敏感的话，可考虑使用MariaDB分支版本，或者直接上线MySQL 5.7最新版本，利用多线程复制的方式可以很大程度降低复制延迟；
- **检测机制完善：**keepalived的检测机制需要适当考虑，不能仅仅只是检查mysqld进程是否存活，或者MySQL服务端口是否可通，还应该进一步做数据写入或者运算的探测，判断响应时间，如果超过设定的阈值，就可以启动切换机制；
- **数据一致性保障：**keepalived或heartbeat自身都无法解决**脑裂**的问题，因此在进行服务异常判断时，可以调整判断脚本，通过对第三方节点补充检测来决定是否进行切换，可降低**脑裂**问题产生的风险。
    - 直接切换可能因为复制延迟有些数据无法查询到而重复写入。
    - keepalived最终确定进行切换时，还需要判断slave的延迟程度。需要事先定好规则，以便决定在延迟情况下，采取直接切换或等待何种策略。

双节点主从+keepalived/heartbeat方案架构示意图：



3.1 配置双主集群

MySQL的可以配置两台服务器互为主从。用来搭建MySQL的高可用架构，用来保证MySQL服务器宕机的时候，能够自动的切换的另一台MySQL服务器。主从的配置可以是基于日志点的也可以是基于GTID的，我们上面讲到了GTID的配置，所以我们现在配置一个基于GTID的双主集群。

3.1.1 master1配置

修改`/etc/my.cnf`文件

```
# 服务器id，一般是ip的最后一段
server-id=132
# 开启binlog
log-bin=mysql-bin
# 表示自增长字段每次递增的量，其默认值是1，取值范围是1 .. 65535
auto_increment_increment=2
# 表示自增长字段从那个数开始，他的取值范围是1 .. 65535，另外一台服务器的offset为2，防止生成的主键冲突
auto_increment_offset=1
# 开启基于GTID的复制
gtid_mode = on
# 只记录对基于gtid的复制安全的语句
enforce-gtid-consistency=true
```

3.1.2 master2配置

```
server-id=133
log-bin=mysql-bin
auto_increment_increment=2
# 生成主键从2开始
auto_increment_offset=2
gtid_mode = on
enforce-gtid-consistency=true
# 需要备份的数据库
binlog-do-db=hello
# 不需要备份的数据库
binlog-ignore-db=mysql
```

3.1.3 建立主从关系

如果之前已经开启的主从复制，建议使用`stop slave`关闭。在每个节点上切换binlog，执行如下语句：

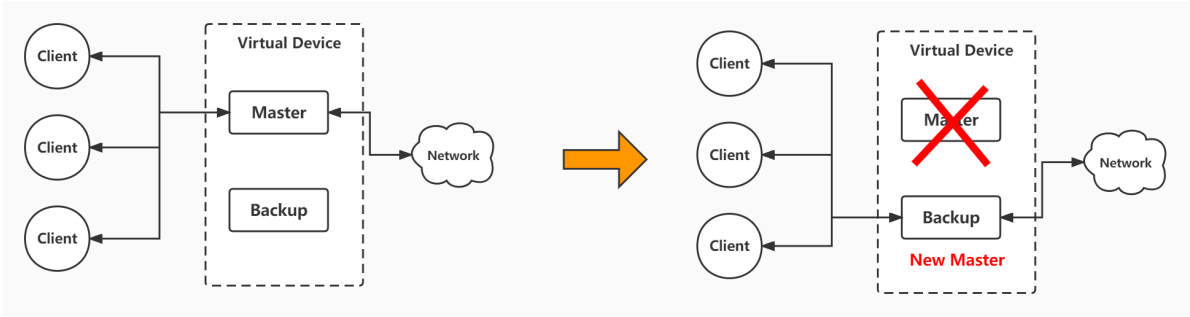
```
# 使用新的binlog
mysql> flush logs;
# 清空binlog
mysql> reset master;
```

然后使用`change master`语句建立主从关系：

```
mysql> change master to
master_host='192.168.68.132',
master_port=3306,
master_user='root',
master_password='root',
master_auto_position = 1;
```

3.2 安装keepalived

Keepalived高可用服务对之间的**故障切换转移**，是通过 VRRP（Virtual Router Redundancy Protocol，虚拟路由器冗余协议）来实现的。在 Keepalived服务正常工作时，主 Master节点会不断地向备节点发送（多播的方式）心跳消息，用以告诉备Backup节点自己还活着，当主 Master节点发生故障时，就无法发送心跳消息，备节点也就因此无法继续检测到来自主 Master节点的心跳了，于是调用自身的接管程序，接管主Master节点的 IP资源及服务。而当主 Master节点恢复时，备Backup节点又会释放Master节点故障时自身接管的IP资源及服务，恢复到原来的备用角色。



什么是VRRP呢?

VRRP ,全称 Virtual Router Redundancy Protocol ,中文名为虚拟路由冗余协议， VRRP的出现就是为了解决静态踣由的单点故障问题，VRRP是通过一种竞选机制来将路由的任务交给某台VRRP路由器的。

3.2.1 安装keepalived

安装keepalived非常简单可以直接使用yum方式在线安装：

```
yum install keepalived -y
```

> 如果出现如下错误信息：
>
> ```
> mysql-community-libs-compat-5.*.*.x86_64.rpm 的公钥尚未安装
> ```
>
> 执行如下命令解决：
>
> ```
> # 安装失败
> rpm --import https://repo.mysql.com/RPM-GPG-KEY-mysql-2022
> ```

获取配置文件路径

```
rpm -qc keepalived
/etc/keepalived/keepalived.conf
/etc/sysconfig/keepalived
```

3.2.2 配置keepalived

```
# 配置通知的email
global_defs {
   notification_email {
     acassen@firewall.loc
     failover@firewall.loc
     sysadmin@firewall.loc
   }
   notification_email_from Alexandre.Cassen@firewall.loc
   smtp_server 192.168.200.1
   smtp_connect_timeout 30
   router_id LVS_DEVEL
```

```
    vrrp_skip_check_adv_addr
    # vrrp_strict
    vrrp_garp_interval 0
    vrrp_gna_interval 0
}
# 检查mysql脚本，定时执行
vrrp_script check_run {
    script "/usr/local/keepalived/check_run.sh"
    interval 3
}
# 设置虚拟ip
vrrp_instance VI_1 {
     # 当前节点的状态MASTER、BACKUP
     state MASTER
     # 当前服务器使用的网卡名称，使用ifconfig查看
     interface ens33
     #VRRP组名，两个节点的设置必须一样
     virtual_router_id 51
     #Master节点的优先级（1-254之间）
     priority 100
     #组播信息发送间隔，两个节点设置必须一样
     advert_int 1
     #设置验证信息，两个节点必须一致
     authentication {
         auth_type PASS
         auth_pass 1111
     }
     #虚拟IP,对外提供MySQL服务的IP地址
     virtual_ipaddress {
         192.168.200.131
     }
     track_script {
         check_run
     }
}
```

3.2.3 检查脚本check_run.sh

```
mkdir /usr/local/keepalived
vim /usr/local/keepalived/check_run.sh
chmod 755 /usr/local/keepalived/check_run.sh
# 不要放在root目录，会报错
```

注意：如果check_run.sh就是不执行，而且日志输出如下错误：`/usr/local/keepalived/check_run.sh exited with status 127`，说明：当前SELinux不支持执行自定义脚本需要执行如下一行语句。

```
chcon -t keepalived_unconfined_script_exec_t
/usr/local/keepalived/check_run.sh
```

```
#!/bin/bash
```

```
. /root/.bashrc
# 检测次数
count=1

# 循环执行检测
while true
do
# 检测MySQL是否可以执行SQL命令，i=0则正常，i=1则不正常
mysql -uroot -proot -S /var/lib/mysql/mysql.sock -e "select now();" > /dev/null 2>&1
i=$?
# 检测MySQL守护进程是否正常，j=0则正常，j=1则不正常
ps aux | grep mysqld | grep -v grep > /dev/null 2>&1
j=$?
# 判断
if [ $i = 0 ] && [ $j = 0 ]
then
# i和j都为0，说明mysql正常则跳出循环
   exit 0
else
   if [ $i = 1 ] && [ $j = 0 ]
   then
   # i=1和j=0，说明mysql守护进程正常，但SQL无法执行
       exit 0
   else
        if [ $count -gt 5 ]
        then
        # 已执行5次检测，break跳出循环，停止keepalived
              break
        fi
   # 次数加一，continue继续循环执行检测
   let count++
   continue
   fi
fi

done
# 停止keepalived
systemctl stop keepalived.service
```

3.2.4 启动keepalived

```
systemctl start keepalived
```

3.2.5 查看vip

```
[root@hero02 ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:16:3e:35:c9:4b brd ff:ff:ff:ff:ff:ff
    inet 172.17.187.78/20 brd 172.17.191.255 scope global dynamic eth0
       valid_lft 315115392sec preferred_lft 315115392sec
    inet 172.17.187.83/32 scope global eth0
       valid_lft forever preferred_lft forever
```

查看keepalived日志信息

```
tail -22f /var/log/messages | grep Keepalived
```

3.3 配置多源复制Slave节点

MySQL 5.7已经开始支持了多源复制，MySQL 5.7之前只能实现一主一从、一主多从或者多主多从的复制，如果想实现多主一从的复制，只好使用MariaDB，但是MariaDB又与官方的MySQL版本不兼容的，在MySQL 5.7版本已经可以实现多主一从的复制了。MySQL 5.7版本相比之前的版本，无论在功能还是性能、安全等方面都已经提升了不少。

多主一从架构带来的好处：

- **方便统计**：在从服务器进行数据汇总，如果我们的主服务器进行了分库分表的操作，为了实现后期的一些数据统计功能，往往需要把数据汇总在一起再统计。
- **合并备份**：如果我们想在从服务器时时对主服务器的数据进行备份，在MySQL 5.7之前每一个主服务器都需要一个从服务器，这样很容易造成资源浪费，同时也加大了DBA的维护成本，但MySQL 5.7引入多源复制，可以把多个主服务器的数据同步到一个从服务器进行备份。

3.3.1 将Master节点的数据同步到Slave节点

> 略

3.3.2 配置my.cnf

```
server-id=134
gtid_mode=ON
enforce-gtid-consistency=ON
master_info_repository=table
relay_log_info_repository=table
```

3.3.3 配置多源复制

```
mysql> change master to
master_host='123.57.135.5',
master_port=3306,
master_user='root',
master_password='root',
master_auto_position = 1
FOR CHANNEL 'm-132';

mysql> change master to
master_host='47.95.211.46',
master_port=3306,
master_user='root',
master_password='root',
master_auto_position = 1
FOR CHANNEL 'm-133';
```

和普通复制不同的是需要增加`FOR CHANNEL 'xxx'`语句指定不同的频道复制，多源复制必须指定参数`master_info_repository=table`

3.3.4 配置跳过的GTID集合

```
#master节点 :
mysql> flush logs;
mysql> show global variables like 'gtid_executed' \G

#slave节点:
mysql> reset master;
Query OK, 0 rows affected (0.00 sec)

mysql> set global gtid_purged='52010ef6-550b-11ed-8295-00163e35c94b:1-
15,535a0306-550b-11ed-aab3-00163e142bef:1-7';
Query OK, 0 rows affected (0.00 sec)

#启动2个Slave复制频道
mysql> start slave for channel 'm-132';
mysql> start slave for channel 'm-133';

#查看2个Slave复制频道状态
mysql> show slave status for channel 'm-132' \G;
mysql> show slave status for channel 'm-133' \G;
```

MySQL分库分表篇

1. 为什么要分库分表?

关系型数据库以MySQL为例，单机的存储能力、连接数是有限的，它自身就很容易会成为系统的瓶颈。

- 当【单数据库中表的数量】达到了几百上千张表时，众多的业务模块都访问这个数据库，压力会比较大，性能和数据库可用性也在下降。
- 当【单表数据量】在百万以里时，我们还可以通过添加从库、优化索引提升性能。一旦数据量朝着千万以上趋势增长，再怎么优化数据库，很多操作性能仍下降严重。

为了减少数据库的负担，提升数据库响应速度，缩短查询时间，这时候就需要进行`分库分表`。分库分表就是要将大量数据分散到多个数据库中，使每个数据库中数据量小响应速度快，以此来提升数据库整体性能。

2. 如何分库分表?

分库分表就是要将大量数据分散到多个数据库中，使每个数据库中数据量小响应速度快，以此来提升数据库整体性能。

核心理念：对数据进行切分（`Sharding`），以及切分后如何对数据的快速定位与整合。

针对数据切分类型，大致可以分为：垂直（纵向）切分和水平（横向）切分两种。

2.1 垂直切分

垂直分片： 按照业务来对数据进行分片，又称为纵向分片，核心理念就是**专库专用**。

垂直切分又细分为`垂直分库`和`垂直分表`

垂直分库

垂直分库是基于业务分类的，和我们常听到的微服务治理观念很相似，每一个独立的服务都拥有自己的数据库，需要不同业务的数据需接口调用。而垂直分库也是按照业务分类进行划分，每个业务有独立数据库，这个比较好理解。

垂直分表

垂直分表是基于数据表的列为依据切分的，是一种大表拆小表的模式。

例如：一个`order`表有很多字段，把长度较大且访问不频繁的字段，拆分出来创建一个单独的扩展表`work_extend`进行存储。

`order`表：

id	workNo	price	describe	.....
int（12）	int（2）	int（15）	varchar（2000）	

拆分后

`order`核心表：

id	workNo	price	.....
int（12）	int（2）	int（15）	

`order_ext`表：

id	workNo	describe	.....
int（12）	int（2）	varchar（2000）	

数据库是以行为单位将数据加载到内存中，这样拆分以后核心表大多是访问频率较高的字段，而且字段长度也都较短，可以加载更多数据到内存中，增加查询的命中率，减少磁盘IO，以此来提升数据库性能。

垂直切分优缺点：

- **优点：**
  - 业务间解耦，不同业务的数据进行独立的维护、监控、扩展
  - 在高并发场景下，一定程度上缓解了数据库的压力
- **缺点：**
  - 提升了开发的复杂度，由于业务的隔离性，很多表无法直接访问，必须通过接口方式聚合数据，
  - 分布式事务管理难度增加
  - 数据库还是存在单表数据量过大的问题，并未根本上解决，需要配合水平切分

2.2 水平切分

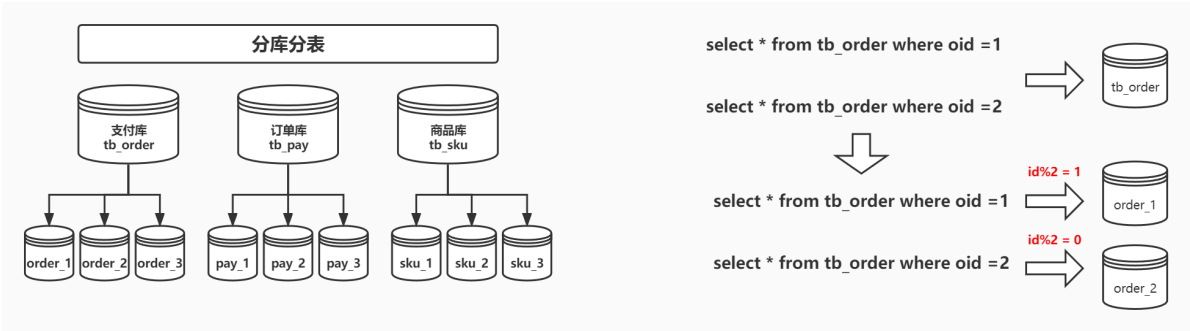
水平分片：又称横向分片。相对于垂直分片，它不再将数据根据业务逻辑分类，而是通过某个字段(或某几个字段)，根据某种规则将数据分散至多个库或表中，每个分片仅包含数据的一部分。

水平切分又分有`库内分表`和`分库分表`

库内分表

库内分表虽然将表拆分，但子表都还是在同一个数据库实例中，只是解决了单一表数据量过大的问题，并没有将拆分后的表分布到不同机器的库上，还在竞争同一个物理机的CPU、内存、网络IO。

分库分表



水平分片突破了单机数据量处理的瓶颈，并且扩展相对自由是分库分表的标准解决方案。

优点：

- 解决高并发时单库数据量过大的问题，提升系统稳定性和负载能力
- 业务系统改造的工作量不是很大

缺点：

- 跨分片的事务一致性难以保证
- 跨库的join关联查询性能较差
- 扩容的难度和维护量较大

3. 数据该往哪个库的表存?

分库分表以后会出现一个问题，一张表会出现在多个数据库里，到底该往哪个库的表里存呢?

常用的分片策略：

- 取余\取模 ： 优点 均匀存放数据，缺点 扩容非常麻烦
- 按照范围分片： 比较好扩容， 数据分布不够均匀
- 按照时间分片： 比较容易将热点数据区分出来。
- 按照枚举值分片： 例如按地区分片
- 按照目标字段前缀指定进行分区：自定义业务规则分片

3.1 根据取值范围

按照`时间区间`或`ID区间`来切分

举个栗子：假如我们切分的是用户表，可以定义每个库的`User表`里只存10000条数据，第一个库`userId`从1 ~ 9999，第二个库10000 ~ 19999，第三个库20000~ 29999.....以此类推。

优点：

- 单表数据量是可控的
- 水平扩展简单只需增加节点即可，无需对其他分片的数据进行迁移
- 能快速定位要查询的数据在哪个库

缺点：

- 由于连续分片可能存在数据热点，如果按时间字段分片，有些分片存储最近时间段内的数据，可能会被频繁的读写，而有些分片存储的历史数据，则很少被查询

3.2 hash取模

hash取模mod（**对hash结果取余数 (hash() mod N)**）的切分方式比较常见，还拿`User表`举例，对数据库从0到N-1进行编号，对`User表`中`userId`字段进行取模，得到余数`i`，`i=0`存第一个库，`i=1`存第二个库，`i=2`存第三个库....以此类推。

这样同一个用户的数据都会存在同一个库里，用`userId`作为条件查询就很好定位了。

优点：

- 数据分片相对比较均匀，不易出现某个库并发访问的问题

缺点：

- 但这种算法存在一些问题，当某一台机器宕机，本应该落在该数据库的请求就无法得到正确的处理，这时宕掉的实例会被踢出集群，此时算法变成hash(userId) mod N-1，用户信息可能就不再在同一个库中。

4. 有哪些分库分表的工具?

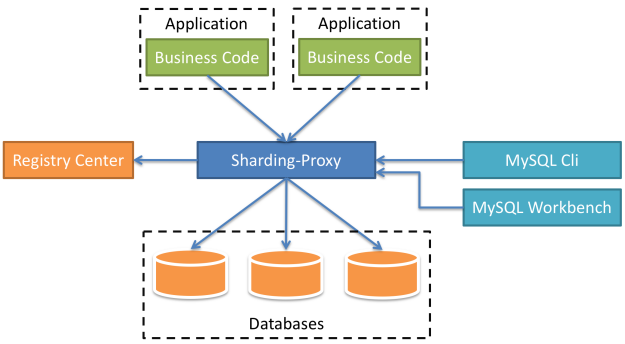
自己开发分库分表工具的工作量是巨大的，好在业界已经有了很多比较成熟的分库分表中间件，我们可以将更多的时间放在业务实现上

常见的分库分表工具有：

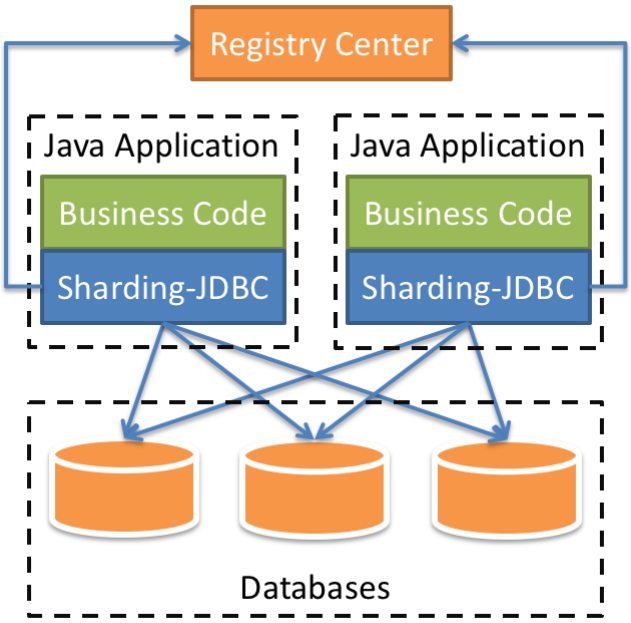
- `sharding-sphere（当当）`
- `MyCAT（基于Cobar）`
- Atlas（奇虎360）
- Cobar（阿里巴巴）
- TSharding（蘑菇街）
- Oceanus（58同城）
- Vitess（谷歌）

分库分表实现核心原理：

- 基于Proxy实现



- 基于JDBC框架实现



5. Sharding JDBC介绍

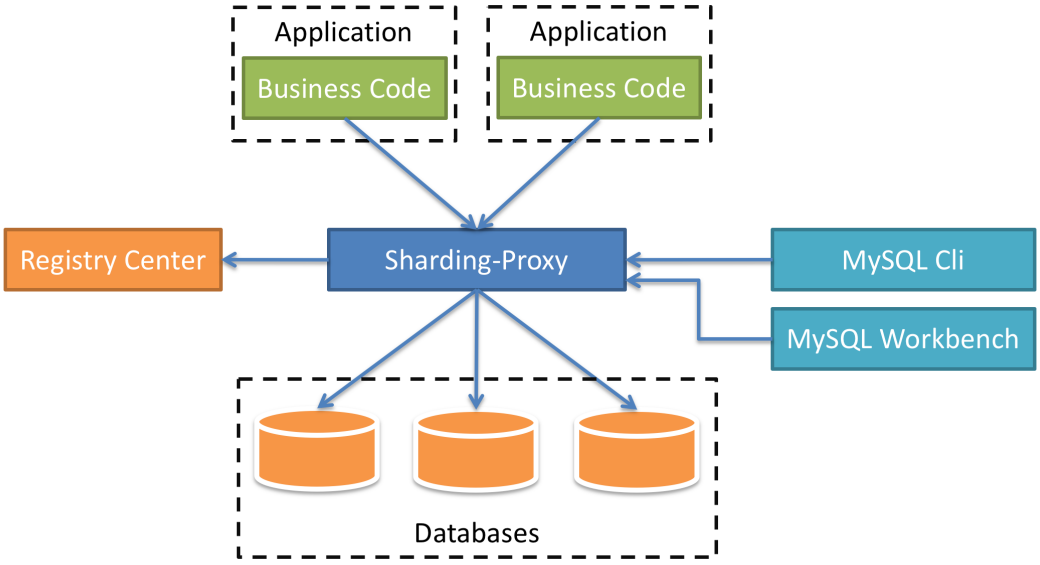
5.1 什么是ShardingSphere?

Apache ShardingSphere 是一款分布式的数据库生态系统，可以将任意数据库转换为分布式数据库，并通过数据分片、弹性伸缩、加密等能力对原有数据库进行增强。

由`Sharding-JDBC`、`Sharding-Proxy`和`Sharding-Sidecar`这3款相互独立，却又能够混合部署配合使用的产品组成。

5.1.1 什么是 Sharding-Proxy?

ShardingSphere-Proxy 定位为透明化的数据库代理端，通过实现数据库二进制协议，对异构语言提供支持。 向应用程序完全透明，可直接当做 MySQL使用；**适用于任何兼容 MySQL 协议的的客户端**，如：MySQL Command Client, MySQL Workbench, Navicat 等。

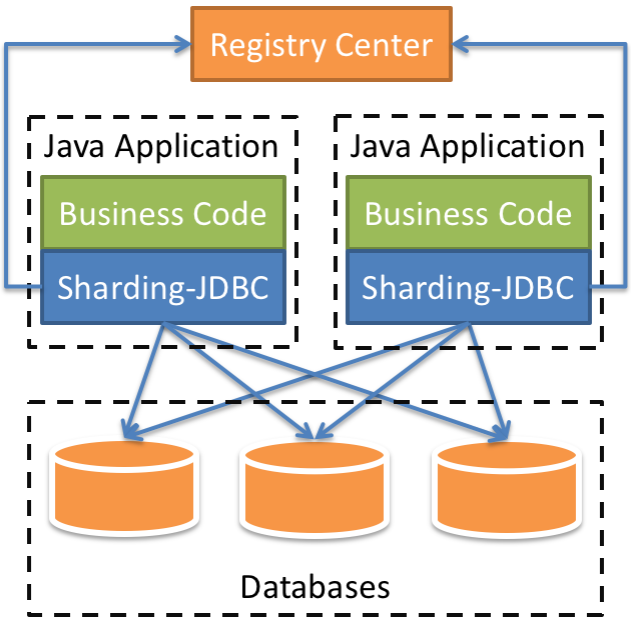


Sharding-Proxy类似前面讲解过的MySQL Router和MySQL Proxy，此处不再赘述，下面咱们重点说一下Sharding-JDBC。

5.1.2 什么是Sharding-JDBC?

ShardingSphere-JDBC 是**增强版的 JDBC 驱动**，定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。 它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖

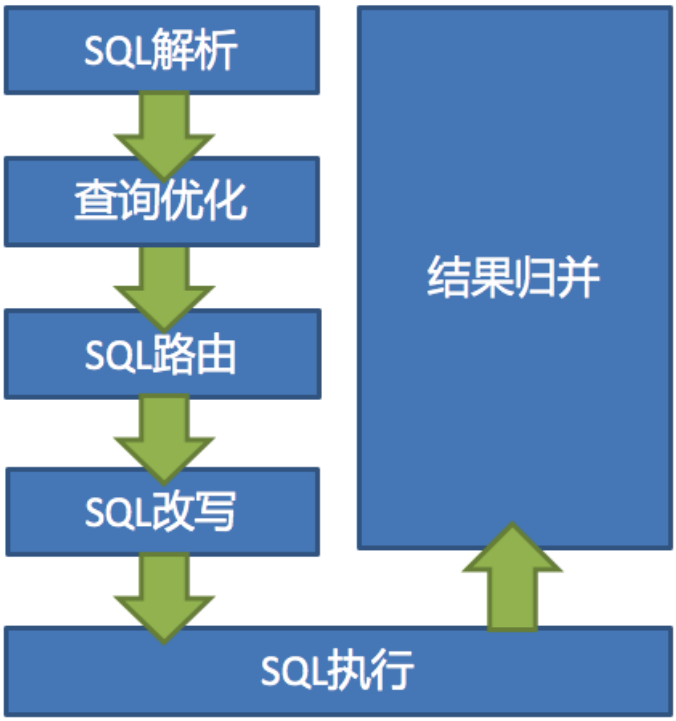
- 适用于任何基于 JDBC 的 ORM 框架，如：JPA、Hibernate、Mybatis、Spring JDBC Template 或直接使用 JDBC；
- 支持任何第三方的数据库连接池，如：DBCP, C3P0, BoneCP, HikariCP 等；
- 支持任意实现 JDBC 规范的数据库，目前支持 MySQL，Oracle，PostgreSQL，SQLServer 以及任何可使用 JDBC 访问的数据库



5.1.3 Sharding JDBC中的核心概念

- **逻辑表**：水平拆分的数据库（表）的相同逻辑和数据结构表的总称。
- **真实表**：在分片的数据库中真实存在的物理表。
- **数据节点**：数据分片的最小单元。由数据源名称和数据表组成。
- 绑定表：指分片规则一致的主表和子表。
    - 例如：`t_order`表和`t_order_item`表，均按照`order_id`分片，则此两张表互为绑定表关系。绑定表之间的多表关联查询不会出现笛卡尔积关联，关联查询效率将大大提升。
- **广播表**：指所有的分片数据源中都存在的表，表结构和表中的数据在每个数据库中均完全一致。适用于数据量不大且需要与海量数据的表进行关联查询的场景，例如：字典表。

5.2 Sharding JDBC架构



数据分片主要流程是完全一致的。 核心由`SQL解析 => 执行器优化 => SQL路由 => SQL改写 => SQL执行 => 结果归并`的流程组成

详细执行流程：

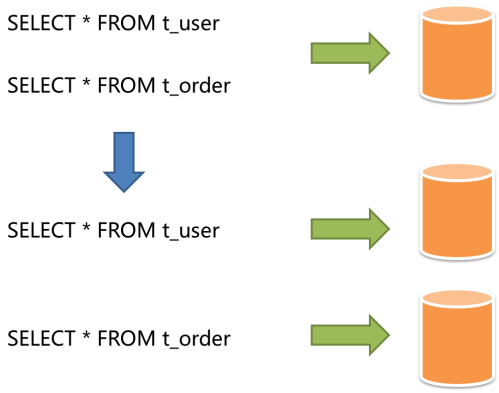
- **SQL解析**
    - 分为**词法解析**和**语法解析。**
    - 先通过**词法解析器**将SQL拆分为一个个不可再分的单词。再使用**语法解析器**对SQL进行理解，并最终提炼出解析上下文。
    - 解析上下文包括：表、选择项、排序项、分组项、聚合函数、分页信息、查询条件以及可能需要修改的占位符的标记。
- **执行器优化**：合并和优化分片条件，如OR等。
- **SQL路由**：根据解析上下文匹配用户配置的分片策略，并生成路由路径。目前支持分片路由和广播路由。

- **SQL改写**：将SQL改写为在真实数据库中可以正确执行的语句。SQL改写分为正确性改写和优化改写。
- **SQL执行**：通过多线程执行器异步执行。
- **结果归并**：将多个执行结果集归并以便于通过统一的JDBC接口输出。结果归并包括流式归并、内存归并和使用装饰者模式的追加归并这几种方式。

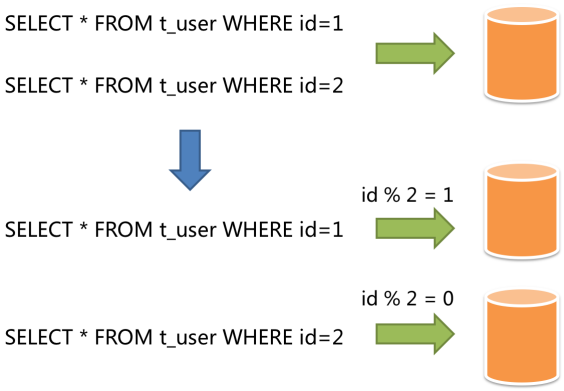
5.3 Sharding JDBC分片策略

5.3.1 数据分片

1）垂直切分：



2）水平切分：



5.3.2 分片策略

分片策略包含**分片键 和 分片算法**。分片键是用于分片的数据库字段，是将数据库(表)水平拆分的关键字段。

分片算法：通过分片算法将数据分片，支持`=`、`BETWEEN AND`和`IN`分片。分片算法需要应用方开发者自行实现，实现灵活度很高。

- 精确分片算法（`PreciseShardingAlgorithm`）用于处理使用单一键作为分片键的`=`与`IN`进行分片的场景
  - 需要配合**标准分片策略**使用
- 范围分片算法（`RangeShardingAlgorithm`）用于处理使用单一键作为分片键的`BETWEEN AND`进行分片的场景

    - 需要配合**标准分片策略**使用
- 复合分片算法（`ComplexKeysShardingAlgorithm`）用于处理使用多键作为分片键进行分片的场景
    - 需要配合**复合分片策略**使用
- Hint分片算法（`HintShardingAlgorithm`）用于处理使用Hint行分片的场景
    - 需要配合**Hint分片策略**使用

分片策略：

- **标准分片策略（`StandardShardingStrategy`）**提供对SQL语句中的`=, IN`和`BETWEEN AND`的分片操作支持。
    - StandardShardingStrategy只支持`单分片键`，提供`PreciseShardingAlgorithm`和`RangeShardingAlgorithm`两个分片算法。
    - **PreciseShardingAlgorithm**是必选的，用于处理`=`和`IN`的分片
    - **RangeShardingAlgorithm**是可选的，用于处理`BETWEEN AND`分片
- **复合分片策略（`ComplexShardingStrategy`）**提供对SQL语句中的`=, IN`和`BETWEEN AND`的分片操作支持。
    - `ComplexShardingStrategy`支持`多分片键`，由于多分片键之间的关系复杂，因此并未进行过多的封装，而是直接将分片键值组合以及分片操作符透传至分片算法，完全由应用开发者实现，提供最大的灵活度。
- **行表达式分片策略（`InlineShardingStrategy`）**使用`Groovy`的Inline表达式，提供对SQL语句中的`=`和`IN`的分片操作支持，
    - `InlineShardingStrategy 只支持单分片键`，对于简单的分片算法，可以通过简单的配置使用，从而避免繁琐的Java代码开发。
        - 如：`t_user_${user_id % 8}` 表示t_user表按照user_id按8取模分成8个表，表名称为t_user_0到t_user_7。
    - 行表达式的使用非常直观，只需要在配置中使用`${ expression }`或`$->{ expression }`标识行表达式即可。 目前支持数据节点和分片算法这两个部分的配置。
    - 行表达式的内容使用的是Groovy的语法，Groovy能够支持的所有操作，行表达式均能够支持。
- **Hint分片策略（`HintShardingStrategy`）**通过Hint而非SQL解析的方式分片的策略。
    - 对于分片字段非SQL决定，而由其他外置条件决定的场景，可使用SQL Hint灵活的注入分片字段。
    - 例：内部系统，按照员工登录主键分库，而数据库中并无此字段。
    - SQL Hint支持通过Java API和SQL注释(待实现)两种方式使用。
- **不分片的策略（`NoneShardingStrategy`）**

5.4 案例：读写分离

```sql
CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(10) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `address` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

5.4.1 基于Spring Boot

```
spring.shardingsphere.datasource.names=master,slave0,slave1
# 配置主库
spring.shardingsphere.datasource.master.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.master.driverClassName=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.master.jdbc-url=jdbc:mysql://123.57.135.5:3306/hello?serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8
spring.shardingsphere.datasource.master.username=root
spring.shardingsphere.datasource.master.password=hero@202207
# 配置第一个从库
spring.shardingsphere.datasource.slave0.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.slave0.driverClassName=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.slave0.jdbc-url=jdbc:mysql://47.95.211.46:3306/hello?serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8
spring.shardingsphere.datasource.slave0.username=root
spring.shardingsphere.datasource.slave0.password=hero@202207
# 配置第二个从库
spring.shardingsphere.datasource.slave1.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.slave1.driverClassName=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.slave1.jdbc-url=jdbc:mysql://123.57.135.5:3306/hello?serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8
spring.shardingsphere.datasource.slave1.username=root
spring.shardingsphere.datasource.slave1.password=hero@202207

spring.shardingsphere.masterslave.name=ms
spring.shardingsphere.masterslave.master-data-source-name=master
spring.shardingsphere.masterslave.slave-data-source-names=slave0,slave1

# spring.shardingsphere.props.sql.show=true
```

5.4.2 不使用Spring

```
public class MasterSlaveDataSource {
    private static DataSource dataSource;

    public static DataSource getInstance() {
        if (dataSource != null) {
            return dataSource;
        }
        try {
            return create();
```

```
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        return null;
    }

    private static DataSource create() throws SQLException {
        // 配置真实数据源
        Map<String, DataSource> dataSourceMap = new HashMap<>();

        // 配置第 1 个数据源
        DruidDataSource masterDataSource = new DruidDataSource();
        masterDataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        masterDataSource.setUrl("jdbc:mysql://123.57.135.5:3306/hello?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8");
        masterDataSource.setUsername("root");
        masterDataSource.setPassword("root");
        dataSourceMap.put("master", masterDataSource);

        // 配置第一个从库
        DruidDataSource slaveDataSource1 = new DruidDataSource();
        slaveDataSource1.setDriverClassName("com.mysql.cj.jdbc.Driver");
        slaveDataSource1.setUrl("jdbc:mysql://47.95.211.46:3306/hello?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8");
        slaveDataSource1.setUsername("root");
        slaveDataSource1.setPassword("root");
        dataSourceMap.put("slave01", slaveDataSource1);
        // 配置第二个从库
        DruidDataSource slaveDataSource2 = new DruidDataSource();
        slaveDataSource2.setDriverClassName("com.mysql.cj.jdbc.Driver");
        slaveDataSource2.setUrl("jdbc:mysql://123.57.135.5:3306/hello?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8");
        slaveDataSource2.setUsername("root");
        slaveDataSource2.setPassword("root");
        dataSourceMap.put("slave02", slaveDataSource2);

        // 配置读写分离规则
        MasterSlaveRuleConfiguration masterSlaveRuleConfig = new
MasterSlaveRuleConfiguration("masterSlaveDataSource", "master",
Arrays.asList("slave01", "slave02"));

        // 获取数据源对象
        dataSource =
MasterSlaveDataSourceFactory.createDataSource(dataSourceMap,
masterSlaveRuleConfig , new Properties());
        //返回数据源
        return dataSource;
    }
}
```

5.5 案例：实现分库分表

```sql
create database ds0;
use ds0;
CREATE TABLE `t_order0` (
  `order_id` int(11) NOT NULL,
  `user_id` int(11) NOT NULL,
  `info` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `t_order1` (
  `order_id` int(11) NOT NULL,
  `user_id` int(11) NOT NULL,
  `info` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

5.5.1 基于Spring Boot

```properties
spring.shardingsphere.datasource.names: ds0,ds1

# 配置数据源ds0
spring.shardingsphere.datasource.ds0.type:
com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds0.driverClassName: com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds0.jdbc-url:
jdbc:mysql://123.57.135.5:3306/ds0?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8
spring.shardingsphere.datasource.ds0.username: root
spring.shardingsphere.datasource.ds0.password: hero@202207

# 配置数据源ds1
spring.shardingsphere.datasource.ds1.type:
com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds1.driverClassName: com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds1.jdbc-url:
jdbc:mysql://47.95.211.46:3306/ds1?
serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8
spring.shardingsphere.datasource.ds1.username: root
spring.shardingsphere.datasource.ds1.password: hero@202207

# 配置分库策略
spring.shardingsphere.sharding.default-database-strategy.inline.sharding-
column: user_id
spring.shardingsphere.sharding.default-database-strategy.inline.algorithm-
expression: ds$->{user_id % 2}

# 配置分表策略
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes: ds$->
{0..1}.t_order$->{0..1}
```

```
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.sharding-column: order_id
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.algorithm-expression: t_order$->{order_id % 2}

```

5.5.2 不使用Spring

```
public class ShardingDataSource {
    private static DataSource dataSource;

    public static DataSource getInstance() {
        if (dataSource != null) {
            return dataSource;
        }
        try {
            return create();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        return null;
    }

    private static DataSource create() throws SQLException {
        // 配置真实数据源
        Map<String, DataSource> dataSourceMap = new HashMap<>();

        // 配置第一个数据源
        DruidDataSource dataSource1 = new DruidDataSource();
        dataSource1.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource1.setUrl("jdbc:mysql://123.57.135.5:3306/ds0?serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8");
        dataSource1.setUsername("root");
        dataSource1.setPassword("hero@202207");
        dataSourceMap.put("ds0", dataSource1);

        // 配置第二个数据源
        DruidDataSource dataSource2 = new DruidDataSource();
        dataSource2.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource2.setUrl("jdbc:mysql://47.95.211.46:3306/ds1?serverTimezone=Asia/Shanghai&useUnicode=true&characterEncoding=utf-8");
        dataSource2.setUsername("root");
        dataSource2.setPassword("hero@202207");
        dataSourceMap.put("ds1", dataSource2);

        // 配置Order表规则
        TableRuleConfiguration orderTableRuleConfig = new TableRuleConfiguration("t_order","ds${0..1}.t_order${0..1}");

        // 配置分库 + 分表策略
        // user_id % 2等于0，则进入ds0库，如果等于1则进入ds1库
        orderTableRuleConfig.setDatabaseShardingStrategyConfig(new InlineShardingStrategyConfiguration("user_id", "ds${user_id % 2}"));
```

```
        // order_id % 2等于0，则进入t_order0表，如果等于1则进入t_order1表
        orderTableRuleConfig.setTableShardingStrategyConfig(new InlineShardingStrategyConfiguration("order_id", "t_order${order_id % 2}"));

        // 配置分片规则
        ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration();
        shardingRuleConfig.getTableRuleConfigs().add(orderTableRuleConfig);

        // 创建数据源
        DataSource dataSource = ShardingDataSourceFactory.createDataSource(dataSourceMap, shardingRuleConfig, new Properties());
        return dataSource;
    }
}
```