**day14-淘淘商城**

# 今日大纲

* Mysql的读写分离

# 课堂笔记

# 背景

我们一般应用对数据库而言都是“读多写少”，也就说对数据库读取数据的压力比较大，有一个思路就是说采用数据库集群的方案，

其中一个是主库，负责写入数据，我们称之为：写库；

其它都是从库，负责读取数据，我们称之为：读库；

那么，对我们的要求是：

1. 读库和写库的数据一致(最终一致)；
2. 写数据必须写到写库；
3. 读数据必须到读库(不一定)；

这里存在两个问题：

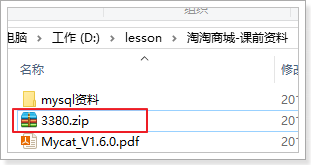
* 1. 如何保证读库和写库一致？
  2. 如何让程序实现动态数据库切换？

我们一个一个来解决

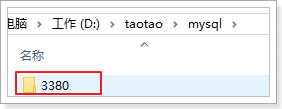
# 部署两台MySQL

## 安装第一台

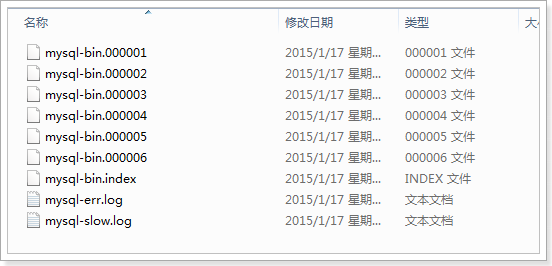
解压缩课前资料这的3380.zip：



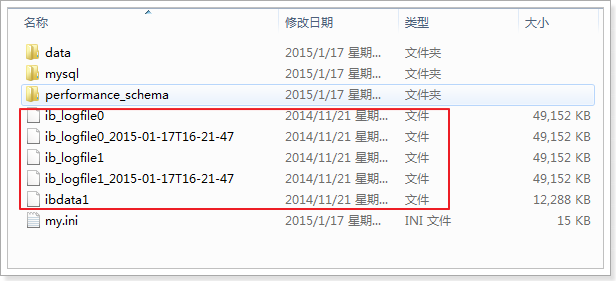
解压到项目目录：



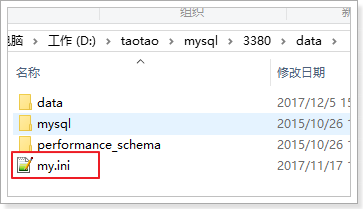
进入3380目录，删除logs目录下的所有文件：



进入3380/data目录，删除以下文件：

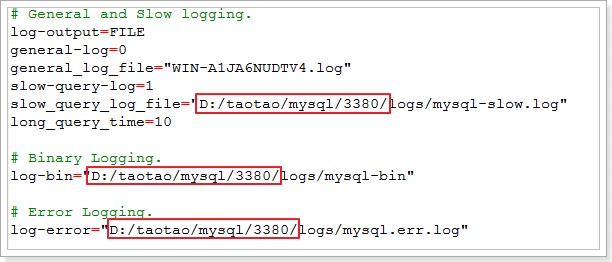


编辑3380/data目录下的my.ini文件：



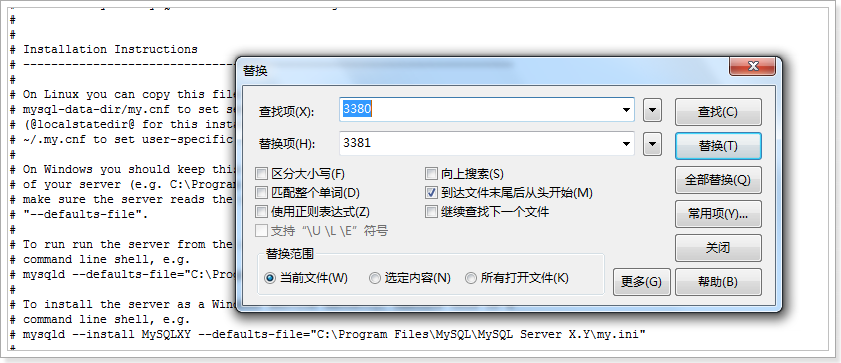
需要修改以下内容：

1）将所有路径替换为你自己的安装路径：



**不止这3处，建议搜索并修改**

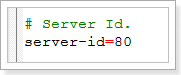
2）如果当前是准备安装第一台，3380不动。如果是第2台，将所有3380替换成3381



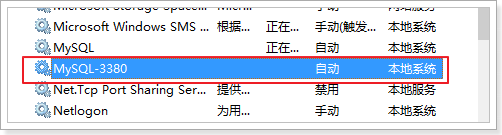
3）注释下面两行内容：



4）保证server-id唯一：



安装MySQL为本地服务：

1. 打开cmd，进入xxx\3380\bin目录，执行如下命令：  
   .\mysqld.exe install MySQL-3380 --defaults-file="xxx\mysql\3380\data\my.ini" 注意改为自己的配置文件路径  
   
2. 进入系统服务查看  
   
3. 修改启动方式为手动，安装完成。
4. 如需删除服务执行如下命名即可：  
   .\mysqld.exe remove MySQL-3380

或者：sc delete Mysql-3380

## 安装第2台

重复前面的步骤，把3380替换为3381

# MySQL主从复制

## 主从同步原理



mysql主(称master)从(称slave)复制的原理：

1. master将数据改变记录到二进制日志(binary log)中,也即是配置文件log-bin指定的文件(这些记录叫做二进制日志事件，binary log events)
2. slave将master的binary log events拷贝到它的中继日志(relay log)
3. slave重做中继日志中的事件,将改变反映它自己的数据(数据重演)

## 主从配置需要注意的地方

1. 主DB server和从DB server数据库的版本一致
2. 主DB server和从DB server数据库数据一致[ 这里就会可以把主的备份在从上还原，也可以直接将主的数据目录拷贝到从的相应数据目录]
3. 主DB server开启二进制日志,主DB server和从DB server的server\_id都必须唯一

## 主库配置（windows，Linux下也类似）

在my.ini修改：

*#开启主从复制，主库的配置*

*log-bin = mysql3306-bin*

*#指定同步的数据库，如果不指定则同步全部数据库*

*binlog-do-db=taotao*

## 在主库创建同步用户

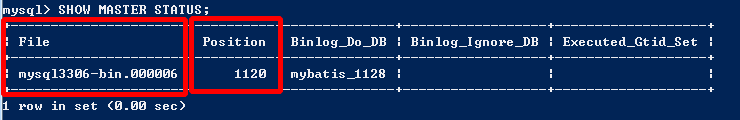
#授权用户slave01使用123456密码登录mysql

grant replication slave on \*.\* to 'slave01'@'127.0.0.1' identified by '123456';

flush privileges;

执行SQL语句查询状态：

*SHOW MASTER STATUS*



需要记录下Position值，需要在从库中设置同步起始值。

## 从库配置

在my.ini修改：

#指定serverid，只要不重复即可，从库也只有这一个配置，

执行SQL：

*CHANGE MASTER TO*

*master\_host='127.0.0.1',*

*master\_user='slave01',*

*master\_password='123456',*

*master\_port=3306,*

*master\_log\_file='mysql3306-bin.000006',*

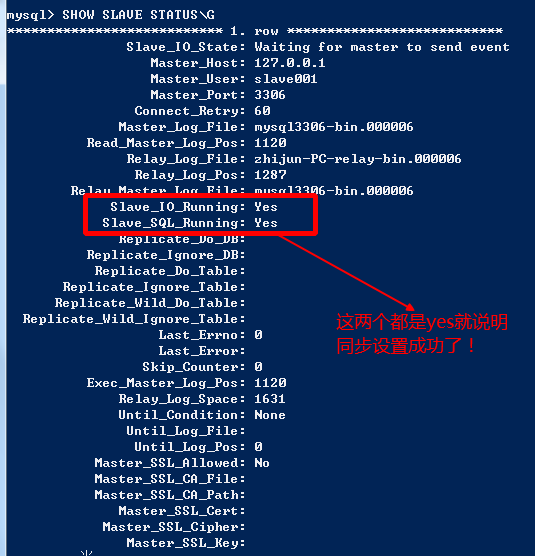
*master\_log\_pos=1120;*

*#启动slave同步*

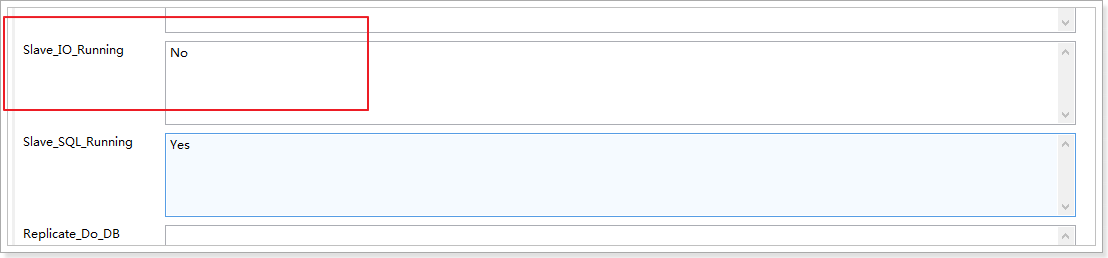
*START SLAVE;*

*#查看同步状态*

*SHOW SLAVE STATUS;*

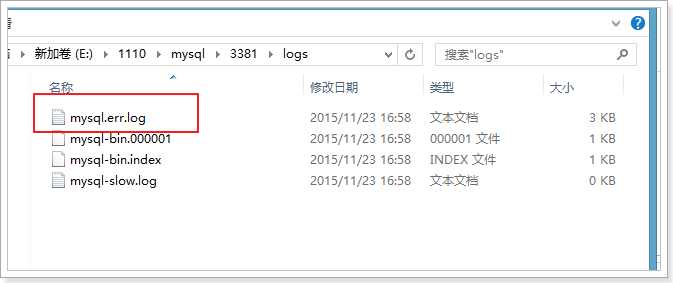


## 主从设置失败解决

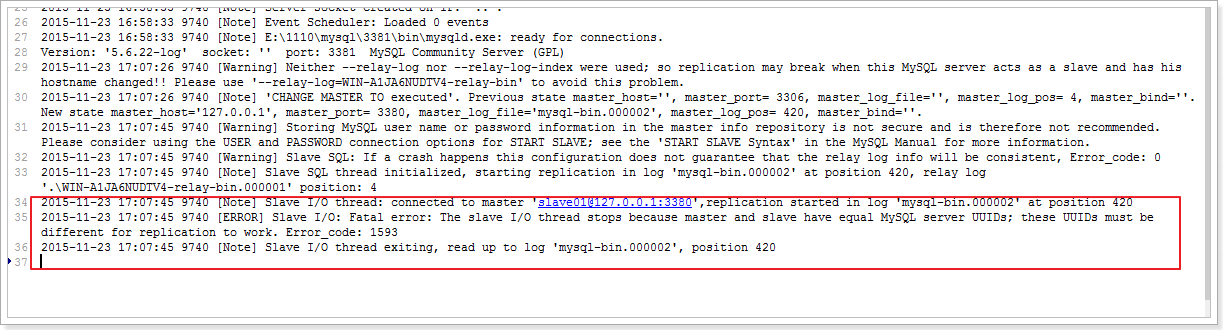


如何定位问题？

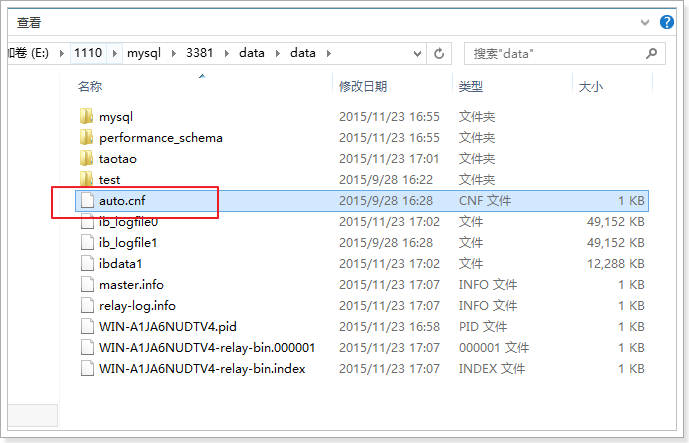
查看3381的日志：

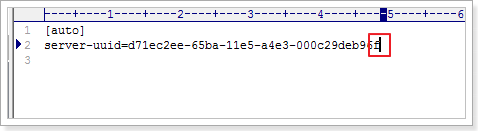


问题：

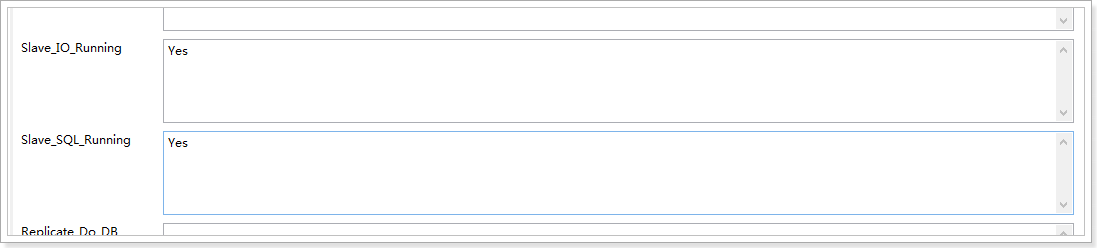


如何设置Mysql的UUID？





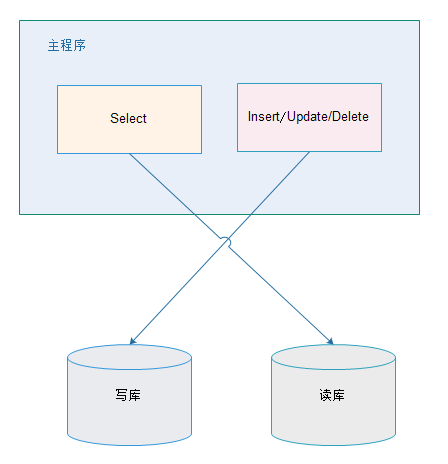
问题解决：



# 动态数据库切换方案

解决读写分离的方案有两种：应用层解决和中间件解决。

## 应用层解决：



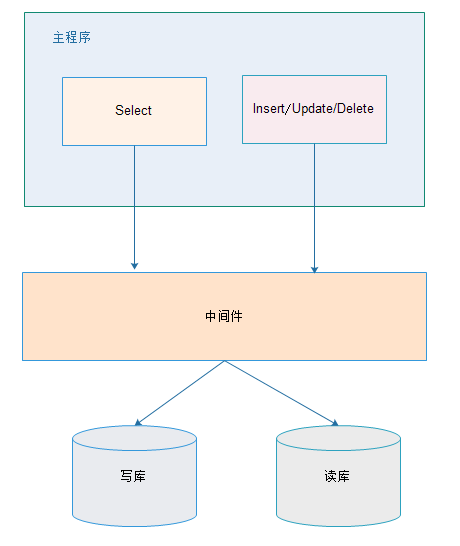
优点：

1. 多数据源切换方便，由程序自动完成；
2. 不需要引入中间件；
3. 理论上支持任何数据库；

缺点：

1. 由程序员完成，运维参与不到；
2. 不能做到动态增加数据源；

## 中间件解决



目前比较好的中间件：Mycat

优缺点：

优点：

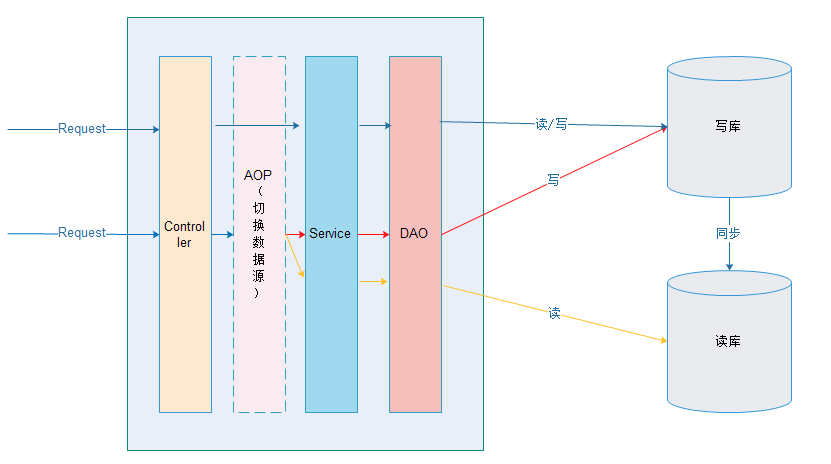
1. 源程序不需要做任何改动就可以实现读写分离；
2. 动态添加数据源不需要重启程序；

缺点：

1. 程序依赖于中间件，会导致切换数据库变得困难；
2. 由中间件做了中转代理，性能有所下降；

# 基于SpringAOP应用层实现

## 原理



在进入Service之前，使用AOP来做出判断，是使用写库还是读库，判断依据可以根据方法名判断，比如说以query、find、get等开头的就走读库，其他的走写库。

## DynamicDataSource

**import** org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

/\*\*

\* 定义动态数据源，实现通过集成Spring提供的AbstractRoutingDataSource，只需要实现determineCurrentLookupKey方法即可

\*

\* 由于DynamicDataSource是单例的，线程不安全的，所以采用ThreadLocal保证线程安全，由DynamicDataSourceHolder完成。

\*

\* **@author** zhijun

\*

\*/

**public** **class** DynamicDataSource **extends** AbstractRoutingDataSource{

// 实现这个方法，并且返回一个DataSource的key，这样Spring就会根据这个key去配置文件中找对应的DataSource

// 从而实现了动态DataSource

@Override

**protected** Object determineCurrentLookupKey() {

// 使用DynamicDataSourceHolder保证线程安全，并且得到当前线程中的数据源key

**return** DynamicDataSourceHolder.*getDataSourceKey*();

}

}

## DynamicDataSourceHolder

/\*\*

\*

\* 使用ThreadLocal技术来记录当前线程中的数据源的key

\*

\* **@author** zhijun

\*

\*/

**public** **class** DynamicDataSourceHolder {

//写库对应的数据源key

**private** **static** **final** String ***MASTER*** = "master";

//读库对应的数据源key

**private** **static** **final** String ***SLAVE*** = "slave";

//使用ThreadLocal记录当前线程的数据源key

**private** **static** **final** ThreadLocal<String> ***holder*** = **new** ThreadLocal<String>();

/\*\*

\* 设置数据源key

\* **@param** key

\*/

**public** **static** **void** putDataSourceKey(String key) {

***holder***.set(key);

}

/\*\*

\* 获取数据源key

\* **@return**

\*/

**public** **static** String getDataSourceKey() {

**return** ***holder***.get();

}

/\*\*

\* 标记写库

\*/

**public** **static** **void** markMaster(){

*putDataSourceKey*(***MASTER***);

}

/\*\*

\* 标记读库

\*/

**public** **static** **void** markSlave(){

*putDataSourceKey*(***SLAVE***);

}

}

## DataSourceAspect

**import** org.apache.commons.lang3.StringUtils;

**import** org.aspectj.lang.JoinPoint;

/\*\*

\* 定义数据源的AOP切面，通过该Service的方法名判断是应该走读库还是写库

\*

\* **@author** zhijun

\*

\*/

**public** **class** DataSourceAspect {

/\*\*

\* 在进入Service方法之前执行

\*

\* **@param** point 切面对象

\*/

**public** **void** before(JoinPoint point) {

// 获取到当前执行的方法名

String methodName = point.getSignature().getName();

**if** (isSlave(methodName)) {

// 标记为读库

DynamicDataSourceHolder.*markSlave*();

} **else** {

// 标记为写库

DynamicDataSourceHolder.*markMaster*();

}

}

/\*\*

\* 判断是否为读库

\*

\* **@param** methodName

\* **@return**

\*/

**private** Boolean isSlave(String methodName) {

// 方法名以query、find、get开头的方法名走从库

**return** StringUtils.*startsWithAny*(methodName, "query", "find", "get");

}

}

## 配置2个数据源

### jdbc.properties

jdbc.master.driver=com.mysql.jdbc.Driver

jdbc.master.url=jdbc:mysql://127.0.0.1:3306/mybatis\_1128?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

jdbc.master.username=root

jdbc.master.password=123456

jdbc.slave01.driver=com.mysql.jdbc.Driver

jdbc.slave01.url=jdbc:mysql://127.0.0.1:3307/mybatis\_1128?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

jdbc.slave01.username=root

jdbc.slave01.password=123456

### 定义连接池

<!-- 配置主库连接池 -->

<bean id=*"masterDataSource"* class=*"com.jolbox.bonecp.BoneCPDataSource"*

destroy-method=*"close"*>

<!-- 数据库驱动 -->

<property name=*"driverClass"* value=*"${jdbc.master.driver}"* />

<!-- 相应驱动的jdbcUrl -->

<property name=*"jdbcUrl"* value=*"${jdbc.master.url}"* />

<!-- 数据库的用户名 -->

<property name=*"username"* value=*"${jdbc.master.username}"* />

<!-- 数据库的密码 -->

<property name=*"password"* value=*"${jdbc.master.password}"* />

<!-- 检查数据库连接池中空闲连接的间隔时间，单位是分，默认值：240，如果要取消则设置为0 -->

<property name=*"idleConnectionTestPeriod"* value=*"60"* />

<!-- 连接池中未使用的链接最大存活时间，单位是分，默认值：60，如果要永远存活设置为0 -->

<property name=*"idleMaxAge"* value=*"30"* />

<!-- 每个分区最大的连接数 -->

<property name=*"maxConnectionsPerPartition"* value=*"150"* />

<!-- 每个分区最小的连接数 -->

<property name=*"minConnectionsPerPartition"* value=*"5"* />

</bean>

<!-- 配置从库连接池 -->

<bean id=*"slave01DataSource"* class=*"com.jolbox.bonecp.BoneCPDataSource"*

destroy-method=*"close"*>

<!-- 数据库驱动 -->

<property name=*"driverClass"* value=*"${jdbc.slave01.driver}"* />

<!-- 相应驱动的jdbcUrl -->

<property name=*"jdbcUrl"* value=*"${jdbc.slave01.url}"* />

<!-- 数据库的用户名 -->

<property name=*"username"* value=*"${jdbc.slave01.username}"* />

<!-- 数据库的密码 -->

<property name=*"password"* value=*"${jdbc.slave01.password}"* />

<!-- 检查数据库连接池中空闲连接的间隔时间，单位是分，默认值：240，如果要取消则设置为0 -->

<property name=*"idleConnectionTestPeriod"* value=*"60"* />

<!-- 连接池中未使用的链接最大存活时间，单位是分，默认值：60，如果要永远存活设置为0 -->

<property name=*"idleMaxAge"* value=*"30"* />

<!-- 每个分区最大的连接数 -->

<property name=*"maxConnectionsPerPartition"* value=*"150"* />

<!-- 每个分区最小的连接数 -->

<property name=*"minConnectionsPerPartition"* value=*"5"* />

</bean>

### 定义DataSource

<!-- 定义数据源，使用自己实现的数据源 -->

<bean id=*"dataSource"* class=*"cn.itcast.usermanage.spring.DynamicDataSource"*>

<!-- 设置多个数据源 -->

<property name=*"targetDataSources"*>

<map key-type=*"java.lang.String"*>

<!-- 这个key需要和程序中的key一致 -->

<entry key=*"master"* value-ref=*"masterDataSource"*/>

<entry key=*"slave"* value-ref=*"slave01DataSource"*/>

</map>

</property>

<!-- 设置默认的数据源，这里默认走写库 -->

<property name=*"defaultTargetDataSource"* ref=*"masterDataSource"*/>

</bean>

## 配置事务管理以及动态切换数据源切面

### 定义事务管理器

<!-- 定义事务管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

### 定义事务策略

<!-- 定义事务策略 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!--定义查询方法都是只读的 -->

<tx:method name=*"query\*"* read-only=*"true"* />

<tx:method name=*"find\*"* read-only=*"true"* />

<tx:method name=*"get\*"* read-only=*"true"* />

<!-- 主库执行操作，事务传播行为定义为默认行为 -->

<tx:method name=*"save\*"* propagation=*"REQUIRED"* />

<tx:method name=*"update\*"* propagation=*"REQUIRED"* />

<tx:method name=*"delete\*"* propagation=*"REQUIRED"* />

<!--其他方法使用默认事务策略 -->

<tx:method name=*"\*"* />

</tx:attributes>

</tx:advice>

### 定义切面

<!-- 定义AOP切面处理器 -->

<bean class=*"cn.itcast.usermanage.spring.DataSourceAspect"* id=*"dataSourceAspect"* />

<aop:config>

<!-- 定义切面，所有的service的所有方法 -->

<aop:pointcut id=*"txPointcut"* expression=*"execution(\* xx.xxx.xxxxxxx.service.\*.\*(..))"* />

<!-- 应用事务策略到Service切面 -->

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"txPointcut"*/>

<!--将通知应用到自定义的切点上，-9999保证该切面优先级最高执行 -->

<aop:aspect ref=*"dataSourceAspect"* order=*"-9999"*>

<aop:before method=*"before"* pointcut-ref=*"txPointcut"* />

</aop:aspect>

</aop:config>

## 改进切面实现，使用事务策略规则匹配

之前的实现我们是将通过方法名匹配，而不是使用事务策略中的定义，我们使用事务管理策略中的规则匹配。

### 改进后的配置

<!-- 定义AOP切面处理器 -->

<bean class=*"cn.itcast.usermanage.spring.DataSourceAspect"* id=*"dataSourceAspect"*>

<!-- 指定事务策略 -->

<property name=*"txAdvice"* ref=*"txAdvice"*/>

<!-- 指定slave方法的前缀（非必须） -->

<property name=*"slaveMethodStart"* value=*"query,find,get"*/>

</bean>

### 改进后的实现

**import** java.lang.reflect.Field;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Map;

**import** org.apache.commons.lang3.StringUtils;

**import** org.aspectj.lang.JoinPoint;

**import** org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource;

**import** org.springframework.transaction.interceptor.TransactionAttribute;

**import** org.springframework.transaction.interceptor.TransactionAttributeSource;

**import** org.springframework.transaction.interceptor.TransactionInterceptor;

**import** org.springframework.util.PatternMatchUtils;

**import** org.springframework.util.ReflectionUtils;

/\*\*

\* 定义数据源的AOP切面，该类控制了使用Master还是Slave。

\*

\* 如果事务管理中配置了事务策略，则采用配置的事务策略中的标记了ReadOnly的方法是用Slave，其它使用Master。

\*

\* 如果没有配置事务管理的策略，则采用方法名匹配的原则，以query、find、get开头方法用Slave，其它用Master。

\*

\* **@author** zhijun

\*

\*/

**public** **class** DataSourceAspect {

**private** List<String> slaveMethodPattern = **new** ArrayList<String>();

**private** **static** **final** String[] ***defaultSlaveMethodStart*** = **new** String[]{ "query", "find", "get" };

**private** String[] slaveMethodStart;

/\*\*

\* 读取事务管理中的策略

\*

\* **@param** txAdvice

\* **@throws** Exception

\*/

@SuppressWarnings("unchecked")

**public** **void** setTxAdvice(TransactionInterceptor txAdvice) **throws** Exception {

**if** (txAdvice == **null**) {

// 没有配置事务管理策略

**return**;

}

//从txAdvice获取到策略配置信息

TransactionAttributeSource transactionAttributeSource = txAdvice.getTransactionAttributeSource();

**if** (!(transactionAttributeSource **instanceof** NameMatchTransactionAttributeSource)) {

**return**;

}

//使用反射技术获取到NameMatchTransactionAttributeSource对象中的nameMap属性值

NameMatchTransactionAttributeSource matchTransactionAttributeSource = (NameMatchTransactionAttributeSource) transactionAttributeSource;

Field nameMapField = ReflectionUtils.*findField*(NameMatchTransactionAttributeSource.**class**, "nameMap");

nameMapField.setAccessible(**true**); //设置该字段可访问

//获取nameMap的值

Map<String, TransactionAttribute> map = (Map<String, TransactionAttribute>) nameMapField.get(matchTransactionAttributeSource);

//遍历nameMap

**for** (Map.Entry<String, TransactionAttribute> entry : map.entrySet()) {

**if** (!entry.getValue().isReadOnly()) {//判断之后定义了ReadOnly的策略才加入到slaveMethodPattern

**continue**;

}

slaveMethodPattern.add(entry.getKey());

}

}

/\*\*

\* 在进入Service方法之前执行

\*

\* **@param** point 切面对象

\*/

**public** **void** before(JoinPoint point) {

// 获取到当前执行的方法名

String methodName = point.getSignature().getName();

**boolean** isSlave = **false**;

**if** (slaveMethodPattern.isEmpty()) {

// 当前Spring容器中没有配置事务策略，采用方法名匹配方式

isSlave = isSlave(methodName);

} **else** {

// 使用策略规则匹配

**for** (String mappedName : slaveMethodPattern) {

**if** (isMatch(methodName, mappedName)) {

isSlave = **true**;

**break**;

}

}

}

**if** (isSlave) {

// 标记为读库

DynamicDataSourceHolder.*markSlave*();

} **else** {

// 标记为写库

DynamicDataSourceHolder.*markMaster*();

}

}

/\*\*

\* 判断是否为读库

\*

\* **@param** methodName

\* **@return**

\*/

**private** Boolean isSlave(String methodName) {

// 方法名以query、find、get开头的方法名走从库

**return** StringUtils.*startsWithAny*(methodName, getSlaveMethodStart());

}

/\*\*

\* 通配符匹配

\*

\* Return if the given method name matches the mapped name.

\* <p>

\* The default implementation checks for "xxx\*", "\*xxx" and "\*xxx\*" matches, as well as direct

\* equality. Can be overridden in subclasses.

\*

\* **@param** methodName the method name of the class

\* **@param** mappedName the name in the descriptor

\* **@return** if the names match

\* **@see** org.springframework.util.PatternMatchUtils#simpleMatch(String, String)

\*/

**protected** **boolean** isMatch(String methodName, String mappedName) {

**return** PatternMatchUtils.*simpleMatch*(mappedName, methodName);

}

/\*\*

\* 用户指定slave的方法名前缀

\* **@param** slaveMethodStart

\*/

**public** **void** setSlaveMethodStart(String[] slaveMethodStart) {

**this**.slaveMethodStart = slaveMethodStart;

}

**public** String[] getSlaveMethodStart() {

**if**(**this**.slaveMethodStart == **null**){

// 没有指定，使用默认

**return** ***defaultSlaveMethodStart***;

}

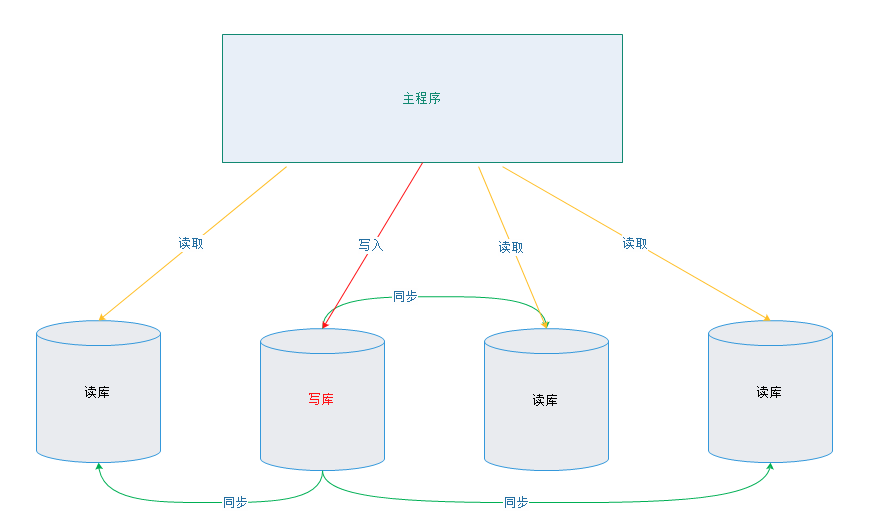
**return** slaveMethodStart;

}

}

## 一主多从的实现

很多实际使用场景下都是采用“一主多从”的架构的，所有我们现在对这种架构做支持，目前只需要修改DynamicDataSource即可。



一旦是一主多从，就需要自己实现负载均衡：

轮询：

随机：

### 实现

**import** java.lang.reflect.Field;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.concurrent.atomic.AtomicInteger;

**import** javax.sql.DataSource;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

**import** org.springframework.util.ReflectionUtils;

/\*\*

\* 定义动态数据源，实现通过集成Spring提供的AbstractRoutingDataSource，只需要实现determineCurrentLookupKey方法即可

\*

\* 由于DynamicDataSource是单例的，线程不安全的，所以采用ThreadLocal保证线程安全，由DynamicDataSourceHolder完成。

\*

\* **@author** zhijun

\*

\*/

**public** **class** DynamicDataSource **extends** AbstractRoutingDataSource {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(DynamicDataSource.**class**);

**private** Integer slaveCount;

// 轮询计数,初始为-1,AtomicInteger是线程安全的

**private** AtomicInteger counter = **new** AtomicInteger(-1);

// 记录读库的key

**private** List<Object> slaveDataSources = **new** ArrayList<Object>(0);

@Override

**protected** Object determineCurrentLookupKey() {

// 使用DynamicDataSourceHolder保证线程安全，并且得到当前线程中的数据源key

**if** (DynamicDataSourceHolder.*isMaster*()) {

Object key = DynamicDataSourceHolder.*getDataSourceKey*();

**if** (***LOGGER***.isDebugEnabled()) {

***LOGGER***.debug("当前DataSource的key为: " + key);

}

**return** key;

}

Object key = getSlaveKey();

**if** (***LOGGER***.isDebugEnabled()) {

***LOGGER***.debug("当前DataSource的key为: " + key);

}

**return** key;

}

@SuppressWarnings("unchecked")

@Override

**public** **void** afterPropertiesSet() {

**super**.afterPropertiesSet();

// 由于父类的resolvedDataSources属性是私有的子类获取不到，需要使用反射获取

Field field = ReflectionUtils.*findField*(AbstractRoutingDataSource.**class**, "resolvedDataSources");

field.setAccessible(**true**); // 设置可访问

**try** {

Map<Object, DataSource> resolvedDataSources = (Map<Object, DataSource>) field.get(**this**);

// 读库的数据量等于数据源总数减去写库的数量

**this**.slaveCount = resolvedDataSources.size() - 1;

**for** (Map.Entry<Object, DataSource> entry : resolvedDataSources.entrySet()) {

**if** (DynamicDataSourceHolder.***MASTER***.equals(entry.getKey())) {

**continue**;

}

slaveDataSources.add(entry.getKey());

}

} **catch** (Exception e) {

***LOGGER***.error("afterPropertiesSet error! ", e);

}

}

/\*\*

\* 轮询算法实现

\*

\* **@return**

\*/

**public** Object getSlaveKey() {

// 得到的下标为：0、1、2、3……

Integer index = counter.incrementAndGet() % slaveCount;

**if** (counter.get() > 9999) { // 以免超出Integer范围

counter.set(-1); // 还原

}

**return** slaveDataSources.get(index);

}

}

# 基于中间件MyCat的实现

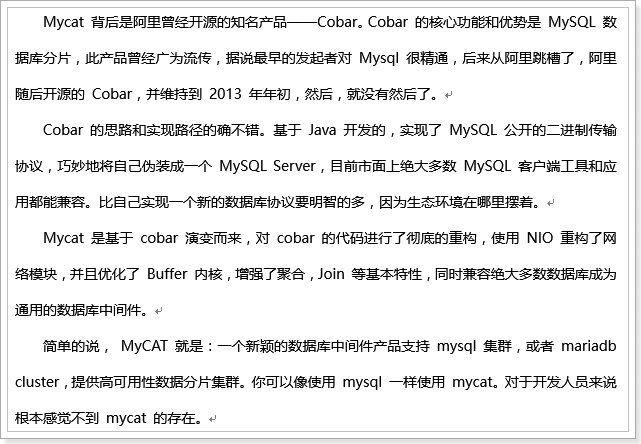


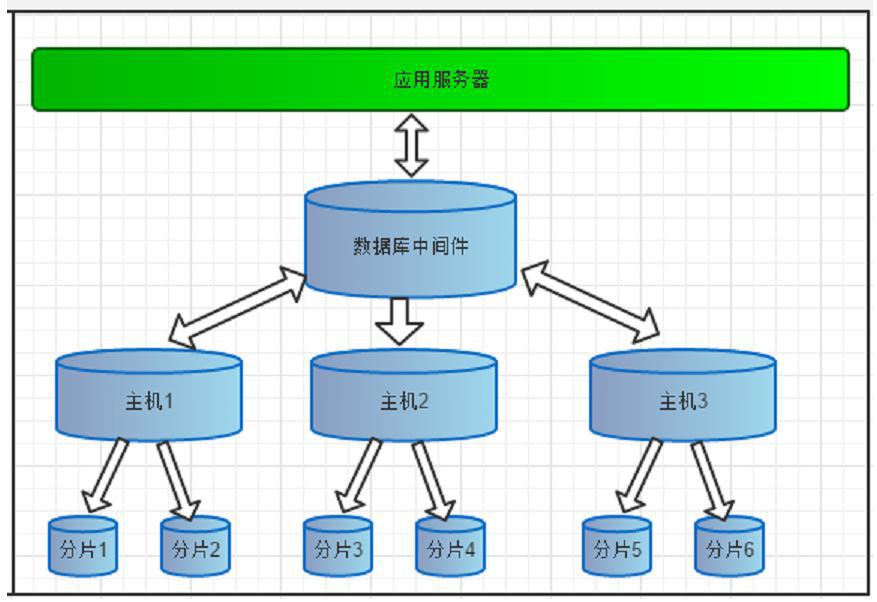
如今随着互联网的发展，数据的量级也是撑指数的增长，从 GB 到 TB 到 PB。对数据的各种操作也是愈加的困难，传统的关系性数据库已经无法满足快速查询与插入数据的需求。这个时候 NoSQL 的出现暂时解决了这一危机。它通过降低数据的安全性，减少对事务的支持，减少对复杂查询的支持，来获取性能上的提升。

但是，在有些场合 NoSQL 一些折衷是无法满足使用场景的，就比如有些使用场景是绝对要有事务与安全指标的。这个时候 NoSQL 肯定是无法满足的，所以还是需要使用关系性数据库。如果使用关系型数据库解决海量存储的问题呢？此时就需要做数据库集群，为了提高查询性能将一个数据库的数据分散到不同的数据库中存储。

## MyCat简介

element-UI





**MyCat 支持的数据库：**



## 发展历史

2013 年阿里的 Cobar 在社区使用过程中发现存在一些比较严重的问题，及其使用限制，经过 Mycat 发起人第一次改良，第一代改良版——Mycat 诞生。 Mycat 开源以后，一些 Cobar 的用户参与了 Mycat 的开发，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。

2014 年 Mycat 首次在上海的《中华架构师》大会上对外宣讲，更多的人参与进来，随后越来越多的项目采用了Mycat。

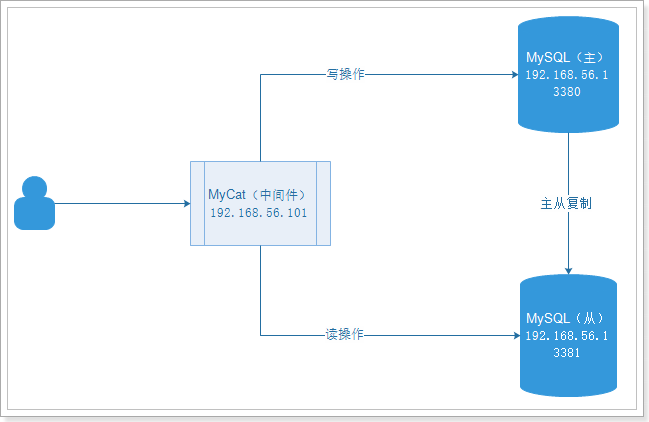
2015 年 5 月，由核心参与者们一起编写的第一本官方权威指南《Mycat 权威指南》电子版发布，累计超过 500本，成为开源项目中的首创。

2015 年 10 月为止，Mycat 项目总共有 16 个 Committer。截至 2015 年 11 月，超过 300 个项目采用 Mycat，涵盖银行、电信、电子商务、物流、移动应用、O2O 的众多领域和公司。

截至 2015 年 12 月，超过 4000 名用户加群或研究讨论或测试或使用 Mycat。

## MyCat读写分离原理

首先搞清楚我们目前的搭建结构：



MyCat是我们的程序与MySQL数据库之间的桥梁，我们的请求不再直接到达数据库，而是到达MyCat，由MyCat决定读和写的去向。

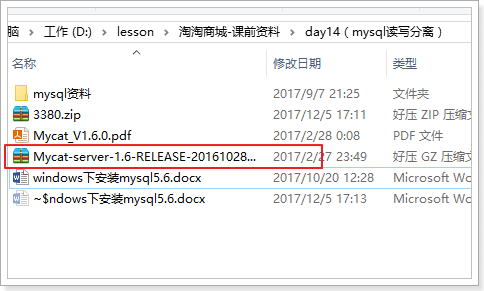
也就是说：

1. 我们的Jdbc连接是连接MyCat，程序代码没有任何变化。（在用户眼里，MyCat就是一个数据库）
2. MyCat中需要配置好主库从库关系，MyCat会根据请求的sql判断读写，实现读写分离。

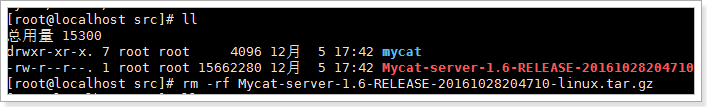
## 不废话，动手试试

### 安装MyCat

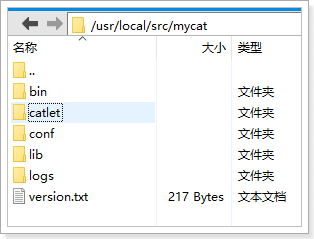
1）将课前资料的安装包上传到/usr/loca/src目录：



2）解压缩，得到mycat文件夹：



### mycat目录结构：



bin：二进制执行文件

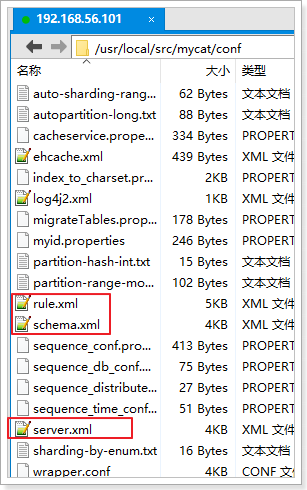
conf：配置文件

lib：依赖

logs：日志

### conf目录的核心配置：

进入conf目录，可以看到大量的配置：



其中核心的配置有3个：

server.xml：配置MyCat作为虚拟数据库的信息（地址、数据库名、用户名、密码等信息）

schemal.xml：mycat代理的真实数据库的信息

rule.xml：分库分表规则配置

### 配置server.xml

server.xml中是MyCat作为虚拟数据库的基本信息。



mycat的默认端口是：8066

### 配置schema.xml实现读写分离

<?xml version=*"1.0"*?>

<!DOCTYPE mycat:schema SYSTEM "schema.dtd">

<mycat:schema xmlns:mycat=*"http://io.mycat/"*>

<!-- 虚拟库与真实库的映射

name="TESTDB" 虚拟库的名字，对应刚刚在server.xml中设置的TESTDB

sqlMaxLimit="100"，允许最大查询条数

checkSQLschema="false" 是否检查自动删除 “虚拟库名”

-->

<schema name=*"TESTDB"* checkSQLschema=*"false"* sqlMaxLimit=*"100"*>

<!-- 可以访问的表，只有设置在这里的表才会被MyCat管理访问

dataNode:虚拟库对应的真实database，对应<dataNode>标签。如果做分片，则配置多个

rule：分片规则，如果没有则删除

-->

<table name=*"tb\_item"* dataNode=*"dn1"*/>

</schema>

<!-- 真实的database信息，每一个dataNode就是一个数据库分片

name：虚拟名称

dataHost：真实库的主机信息，对应<dataHost>标签

database：真实database名称

-->

<dataNode name=*"dn1"* dataHost=*"localhost1"* database=*"taotao"* />

<!-- 真实库的主机信息

name：主机名

maxCon：最大连接， minCon：最小连接

balance：负载均衡方式：0~3四种选项。0，不开启读写分离。1~3都开启，区别是主是否参与读

writeType：写负载均衡。永远设置0

dbDriver：驱动类型，推荐native，可选jdbc

switchType：主从的自动切换

-->

<dataHost name=*"localhost1"* maxCon=*"1000"* minCon=*"10"* balance=*"3"*

writeType=*"0"* dbType=*"mysql"* dbDriver=*"native"* switchType=*"1"*>

<heartbeat>select user()</heartbeat>

<!-- can have multi write hosts -->

<writeHost host=*"hostM1"* url=*"192.168.56.1:3380"* user=*"root"*

password=*"root"*>

<!-- can have multi read hosts -->

<readHost host=*"hostS2"* url=*"192.168.56.1:3381"* user=*"root"* password=*"root"* />

</writeHost>

</dataHost>

</mycat:schema>

### 启动MyCat

进入 mycat/bin目录：

启动： ./mycat start

停止： ./mycat stop

重启： ./mycat restart

状态： ./mycat status

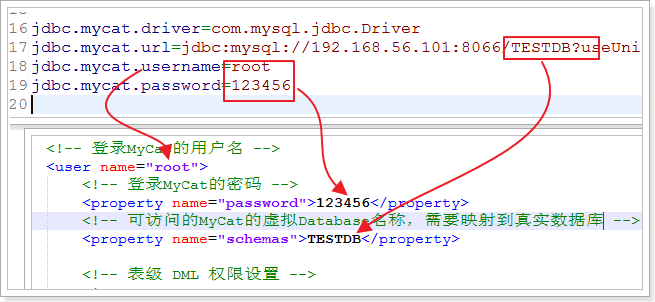
### 配置远程连接权限

在两台需要代理的机器上配置 允许远程连接：

grant all privileges on \*.\* to root@'%' identified by 'root';

### 在项目中使用

其实项目中，只要改一个地方即可，就是jdbc的连接参数。



**注意：**mycat中的，一旦你给sql 加了事务，一定走主库，读写分离失效。

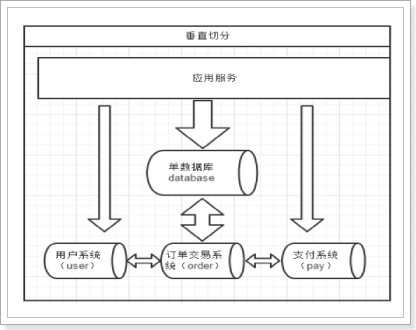
## MyCat数据分片原理

### 什么是数据分片

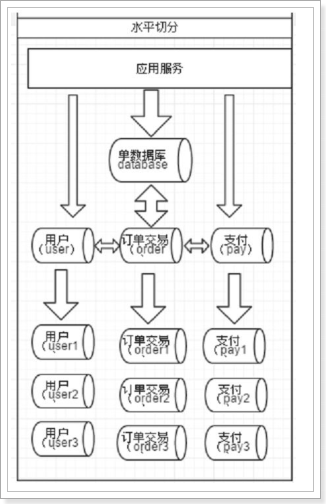
简单来说，就是指通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。

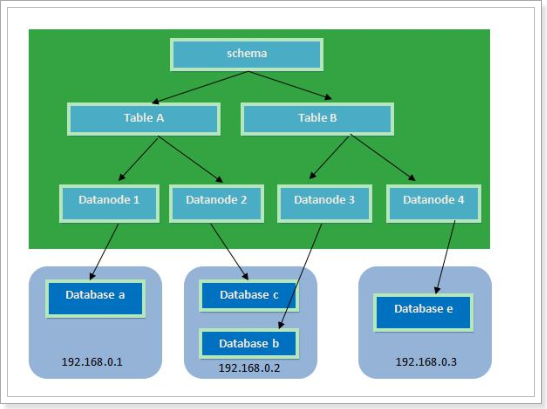
（1）一种是按照不同的表（或者 Schema）来切分到不同的数据库（主机）之上，这种切分可以称之为数据的垂直（纵向）切分



（2）另外一种则是根据表中的数据的逻辑关系，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）上面，这种切分称之为数据的水平（横向）切分。



MyCat的分片实现：



**逻辑库(schema) ：**

MyCat作为一个数据库中间件，起到一个程序与数据库的桥梁作用。开发人员无需知道MyCat的存在，只需要知道数据库的概念即可。为了让MyCat更透明，它会把自己“伪装”成一个MySQL数据库，因此需要有一个虚拟的 database，在MyCat中也叫逻辑库，英文就是schema。

**逻辑表（table）：**

既然有逻辑库，那么就会有逻辑表，分布式数据库中，对应用来说，读写数据的表就是逻辑表。逻辑表，可以是数据切分后，分布在一个或多个分片库中，也可以不做数据切分，不分片，只有一个表构成。

分片表：是指那些原有的很大数据的表，需要切分到多个数据库的表，这样，每个分片都有一部分数据，所有分片构成了完整的数据。 总而言之就是需要进行分片的表。

非分片表：一个数据库中并不是所有的表都很大，某些表是可以不用进行切分的，非分片是相对分片表来说的，就是那些不需要进行数据切分的表。

**分片节点(dataNode)**

数据切分后，一个大表被分到不同的分片数据库上面，每个表分片所在的数据库就是分片节点（dataNode）。

**节点主机(dataHost)**

数据切分后，每个分片节点（dataNode）不一定都会独占一台机器，同一机器上面可以有多个分片数据库，这样一个或多个分片节点（dataNode）所在的机器就是节点主机（dataHost）,为了规避单节点主机并发数限制，尽量将读写压力高的分片节点（dataNode）均衡的放在不同的节点主机（dataHost）。

**分片规则(rule)**

前面讲了数据切分，一个大表被分成若干个分片表，就需要一定的规则，这样按照某种业务规则把数据分到某个分片的规则就是分片规则，数据切分选择合适的分片规则非常重要，将极大的避免后续数据处理的难度。

分片的schema配置：

<?xml version="1.0"?>

<!DOCTYPE mycat:schema SYSTEM "schema.dtd">

<mycat:schema xmlns:mycat="http://io.mycat/">

<schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">

<table name="student" primaryKey="id" dataNode="dn1,dn2" rule="rule1"/>

</schema>

<dataNode name="dn1" dataHost="host1" database="usermanage" />

<dataNode name="dn2" dataHost="host2" database="usermanage" />

<dataHost name="host1" maxCon="1000" minCon="10" balance="0"

writeType="0" dbType="mysql" dbDriver="native" switchType="1" slaveThreshold="100">

<heartbeat>select user()</heartbeat>

<writeHost host="masterM1" url="192.168.56.101:3306" user="root"

password="123456"/>

</dataHost>

<dataHost name="host2" maxCon="1000" minCon="10" balance="0"

writeType="0" dbType="mysql" dbDriver="native" switchType="1" slaveThreshold="100">

<heartbeat>select user()</heartbeat>

<writeHost host="masterM1" url="192.168.56.102:3306" user="root"

password="123456"/>

</dataHost>

</mycat:schema>