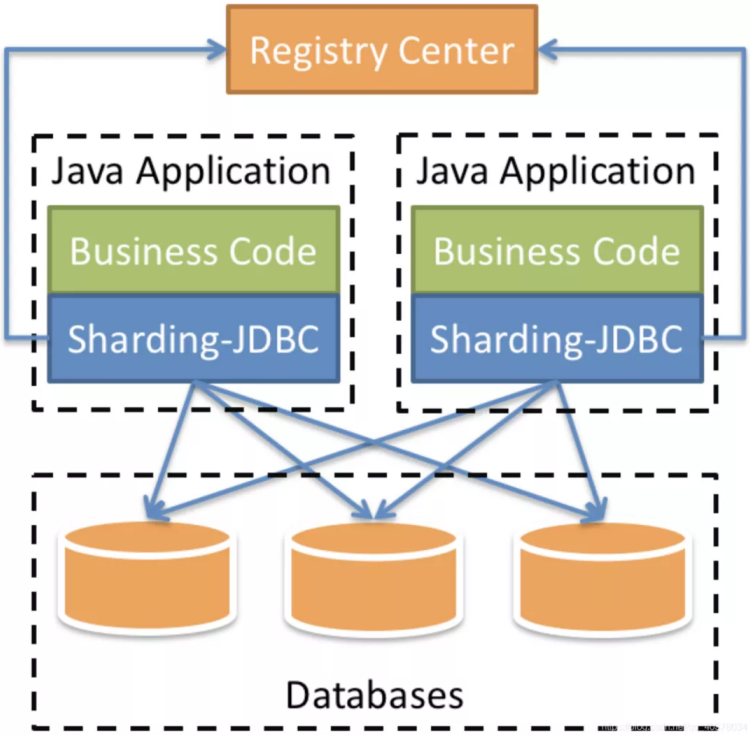
Sharding-Jdbc 实现读写分离 + 分库分表，写得太好了！

邋遢的流浪剑客 石杉的架构笔记 今天

来源：https://blog.csdn.net/qq_40378034/article/details/115264837

概览

ShardingSphere-Jdbc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，可理解为增强版的Jdbc驱动，完全兼容Jdbc和各种ORM框架。



| MySQL主从复制

1）docker配置mysql主从复制

1）创建主服务器所需目录

```
mkdir -p /usr/local/mysqlData/master/cnf
mkdir -p /usr/local/mysqlData/master/data
```

2）定义主服务器配置文件

```
vim /usr/local/mysqlData/master/cnf/mysql.cnf
[mysqld]
## 设置server_id,注意要唯一
server-id=1
## 开启binlog
log-bin=mysql-bin
## binlog缓存
binlog_cache_size=1M
## binlog格式(mixed、statement、row,默认格式是statement)
binlog_format=mixed
```

3）创建并启动mysql主服务

```
docker run -itd -p 3306:3306 --name master -v /usr/local/mysqlData/master/cnf:/etc/mysql/conf.
```

4）添加复制master数据的用户reader，供从服务器使用

```
[root@aliyun /]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
6af1df686fff        mysql:5.7           "docker-entrypoint..."   5 seconds ago       Up 4 secc
[root@aliyun /]# docker exec -it master /bin/bash
root@41d795785db1:/# mysql -u root -p123456

mysql> GRANT REPLICATION SLAVE ON *.* to 'reader'@'%' identified by 'reader';
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

5）创建从服务器所需目录，编辑配置文件

```
mkdir /usr/local/mysqlData/slave/cnf -p
mkdir /usr/local/mysqlData/slave/cnf -p
vim /usr/local/mysqlData/slave/cnf/mysql.cnf
[mysqld]
## 设置server_id,注意要唯一
server-id=2
## 开启binlog,以备Slave作为其它Slave的Master时使用
log-bin=mysql-slave-bin
## relay_log配置中继日志
relay_log=edu-mysql-relay-bin
## 如果需要同步函数或者存储过程
log_bin_trust_function_creators=true
## binlog缓存
binlog_cache_size=1M
## binlog格式(mixed、statement、row,默认格式是statement)
binlog_format=mixed
## 跳过主从复制中遇到的所有错误或指定类型的错误,避免slave端复制中断
## 如:1062错误是指一些主键重复,1032错误是因为主从数据库数据不一致
slave_skip_errors=1062
```

6）创建并运行mysql从服务器

```
docker run -itd -p 3307:3306 --name slaver -v /usr/local/mysqlData/slave/cnf:/etc/mysql/conf.d
```

7）在从服务器上配置连接主服务器的信息

首先主服务器上查看 `master_log_file` 、 `master_log_pos` 两个参数，然后切换到从服务器上进行主服务器的连接信息的设置。

主服务上执行：

```
root@6af1df686fff:/# mysql -u root -p123456

mysql> show master status;
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-bin.000003 |      591 |              |                  |                   |
+------------------+----------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

docker查看主服务器容器的ip地址：

```
[root@aliyun /]# docker inspect --format='{{.NetworkSettings.IPAddress}}' master
172.17.0.2
```

从服务器上执行：

```
[root@aliyun /]# docker exec -it slaver /bin/bash
root@fe8b6fc2f1ca:/# mysql -u root -p123456

mysql> change master to master_host='172.17.0.2',master_user='reader',master_password='reader'
```

8）从服务器启动I/O 线程和SQL线程

```
mysql> start slave;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> show slave status\G
*************************** 1. row ***************************
               Slave_IO_State: Waiting for master to send event
                  Master_Host: 172.17.0.2
                  Master_User: reader
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000003
```

```
          Read_Master_Log_Pos: 591
               Relay_Log_File: edu-mysql-relay-bin.000002
                Relay_Log_Pos: 320
        Relay_Master_Log_File: mysql-bin.000003
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
```

`Slave_IO_Running: Yes，Slave_SQL_Running: Yes` 即表示启动成功。

2）binlog和redo log回顾

1）redo log（重做日志）

InnoDB首先将redo log放入到redo log buffer，然后按一定频率将其刷新到redo log file。

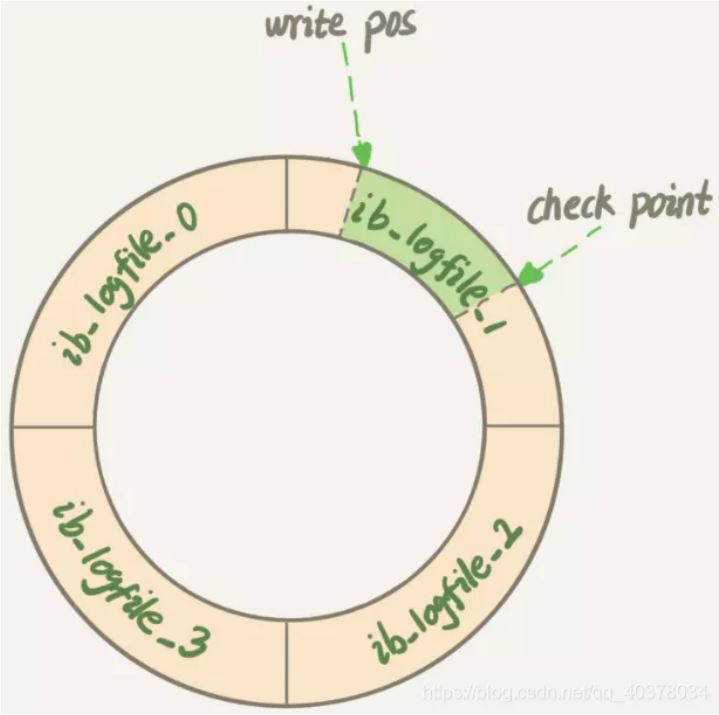
下列三种情况下会将redo log buffer刷新到redo log file：

- Master Thread每一秒将redo log buffer刷新到redo log file。
- 每个事务提交时会将redo log buffer刷新到redo log file。
- 当redo log缓冲池剩余空间小于1/2时，会将redo log buffer刷新到redo log file。

MySQL里常说的WAL技术，全称是Write Ahead Log，即当事务提交时，先写redo log，再修改页。也就是说，当有一条记录需要更新的时候，InnoDB会先把记录写到redo log里面，并更新Buffer Pool的page，这个时候更新操作就算完成了。

Buffer Pool是物理页的缓存，对InnoDB的任何修改操作都会首先在Buffer Pool的page上进行，然后这样的页将被标记为脏页并被放到专门的Flush List上，后续将由专门的刷脏线程阶段性的将这些页面写入磁盘。

InnoDB的redo log是固定大小的，比如可以配置为一组4个文件，每个文件的大小是1GB，循环使用，从头开始写，写到末尾就又回到开头循环写（顺序写，节省了随机写磁盘的IO消耗）。



Write Pos是当前记录的位置，一边写一边后移，写到第3号文件末尾后就回到0号文件开头。Check Point是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件。

Write Pos和Check Point之间空着的部分，可以用来记录新的操作。如果Write Pos追上Check Point，这时候不能再执行新的更新，需要停下来擦掉一些记录，把Check Point推进一下。

当数据库发生宕机时，数据库不需要重做所有的日志，因为Check Point之前的页都已经刷新回磁盘，只需对Check Point后的redo log进行恢复，从而缩短了恢复的时间。

当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Check Point，将脏页刷新回磁盘。

2）binlog（归档日志）

MySQL整体来看就有两块：一块是Server层，主要做的是MySQL功能层面的事情；还有一块是引擎层，负责存储相关的具体事宜。redo log是InnoDB引擎特有的日志，而Server层也有自己的日志，称为binlog。

binlog记录了对MySQL数据库执行更改的所有操作，不包括SELECT和SHOW这类操作，主要作用是用于数据库的主从复制及数据的增量恢复。

使用mysqldump备份时，只是对一段时间的数据进行全备，但是如果备份后突然发现数据库服务器故障，这个时候就要用到binlog的日志了。

binlog格式有三种：STATEMENT，ROW，MIXED。

- STATEMENT模式：binlog里面记录的就是SQL语句的原文。优点是并不需要记录每一行的数据变化，减少了binlog日志量，节约IO，提高性能。缺点是在某些情况下会导致master-slave中的数据不一致。
- ROW模式：不记录每条SQL语句的上下文信息，仅需记录哪条数据被修改了，修改成什么样了，解决了STATEMENT模式下出现master-slave中的数据不一致。缺点是会产生大量的日志，尤其是alter table的时候会让日志暴涨。
- MIXED模式：以上两种模式的混合使用，一般的复制使用STATEMENT模式保存binlog，对于STATEMENT模式无法复制的操作使用ROW模式保存binlog，MySQL会根据执行的SQL语句选择日志保存方式。

3）redo log和binlog日志的不同

- redo log是InnoDB引擎特有的；binlog是MySQL的Server层实现的，所有引擎都可以使用。
- redo log是物理日志，记录的是在某个数据也上做了什么修改；binlog是逻辑日志，记录的是这个语句的原始逻辑，比如给ID=2这一行的c字段加1。
- redo log是循环写的，空间固定会用完；binlog是可以追加写入的，binlog文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

4）两阶段提交

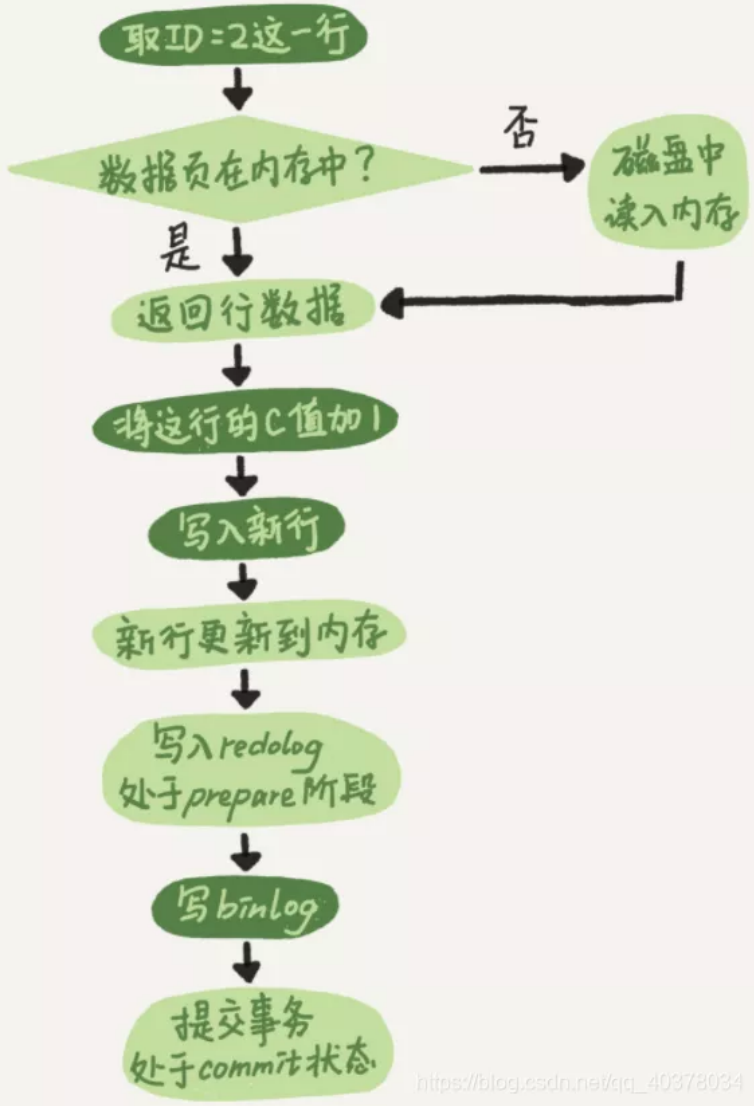
```
create table T(ID int primary key, c int);
update T set c=c+1 where ID=2;
```

执行器和InnoDB引擎在执行这个update语句时的内部流程：

- 执行器先找到引擎取ID=2这一行。ID是主键，引擎直接用树搜索找到这一行。如果ID=2这一行所在的数据也本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。

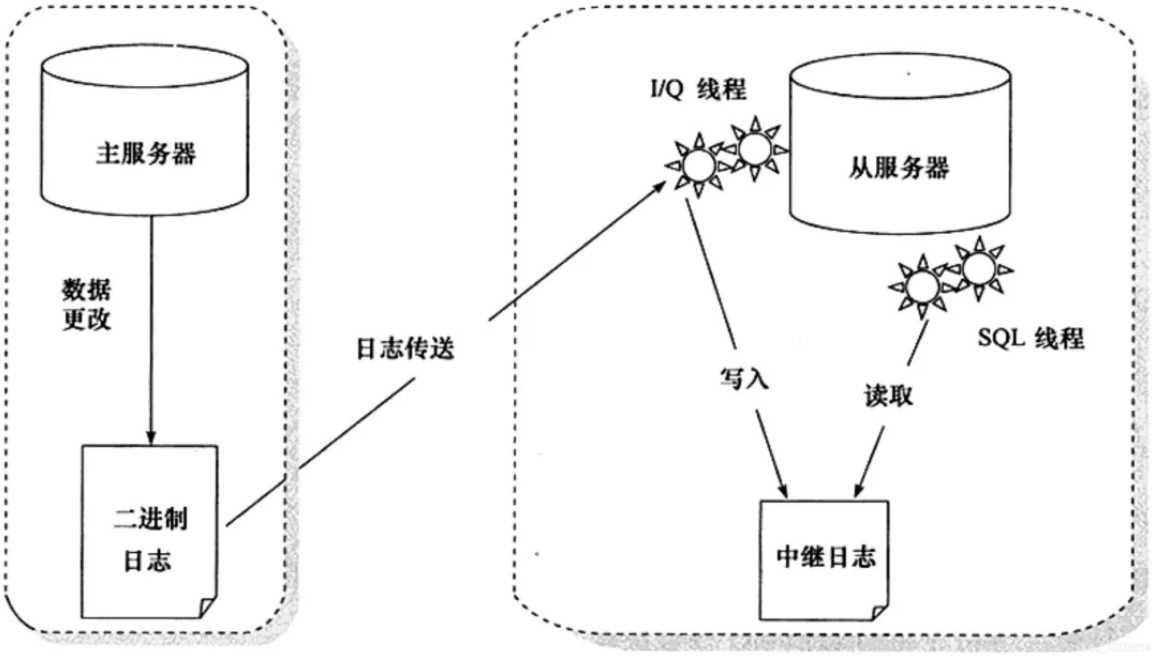
- 执行器拿到引擎给的行数据，把这个值加上1，得到新的一行数据，再调用引擎接口写入这行新数据。
- 引擎将这行新数据更新到内存中，同时将这个更新操作记录到redo log里面，此时redo log处于prepare状态。然后告知执行器执行完成了，随时可以提交事务。
- 执行器生成这个操作的binlog，并把binlog写入磁盘
- 执行器调用引擎的提交事务接口，引擎把刚刚写入的redo log改成提交状态，更新完成。

update语句的执行流程图如下，图中浅色框表示在InnoDB内部执行的，深色框表示是在执行器中执行的。



将redo log的写入拆成了两个步骤：prepare和commit，这就是两阶段提交。

3）MySQL主从复制原理



从库B和主库A之间维持了一个长连接。主库A内部有一个线程，专门用于服务从库B的这个长连接。一个事务日志同步的完整过程如下：

- 在从库B上通过change master命令，设置主库A的IP、端口、用户名、密码，以及要从哪个位置开始请求binlog，这个位置包含文件名和日志偏移量。
- 在从库B上执行start slave命令，这时从库会启动两个线程，就是图中的I/O线程和SQL线程。其中I/O线程负责与主库建立连接。
- 主库A校验完用户名、密码后，开始按照从库B传过来的位置，从本地读取binlog，发给B。
- 从库B拿到binlog后，写到本地文件，称为中继日志。
- SQL线程读取中继日志，解析出日志里的命令，并执行。

由于多线程复制方案的引入，SQL线程演化成了多个线程。

主从复制不是完全实时地进行同步，而是异步实时。这中间存在主从服务之间的执行延时，如果主服务器的压力很大，则可能导致主从服务器延时较大。

| Sharding-Jdbc实现读写分离

1）新建**Springboot**工程，引入相关依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

```
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.1.4</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.21</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shardingsphere</groupId>
        <artifactId>sharding-jdbc-spring-boot-starter</artifactId>
        <version>4.0.0-RC1</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2）application.properties配置文件

```
spring.main.allow-bean-definition-overriding=true
#显示sql
spring.shardingsphere.props.sql.show=true

#配置数据源
spring.shardingsphere.datasource.names=ds1,ds2,ds3

#master-ds1数据库连接信息
spring.shardingsphere.datasource.ds1.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds1.url=jdbc:mysql://47.101.58.187:3306/sharding-jdbc-db?useU
spring.shardingsphere.datasource.ds1.username=root
spring.shardingsphere.datasource.ds1.password=123456
```

```
spring.shardingsphere.datasource.ds1.maxPoolSize=100
spring.shardingsphere.datasource.ds1.minPoolSize=5

#slave-ds2数据库连接信息
spring.shardingsphere.datasource.ds2.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds2.url=jdbc:mysql://47.101.58.187:3307/sharding-jdbc-db?useU
spring.shardingsphere.datasource.ds2.username=root
spring.shardingsphere.datasource.ds2.password=123456
spring.shardingsphere.datasource.ds2.maxPoolSize=100
spring.shardingsphere.datasource.ds2.minPoolSize=5

#slave-ds3数据库连接信息
spring.shardingsphere.datasource.ds3.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds3.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds3.url=jdbc:mysql://47.101.58.187:3307/sharding-jdbc-db?useU
spring.shardingsphere.datasource.ds3.username=root
spring.shardingsphere.datasource.ds3.password=123456
spring.shardingsphere.datasource.ds.maxPoolSize=100
spring.shardingsphere.datasource.ds3.minPoolSize=5

#配置默认数据源ds1 默认数据源,主要用于写
spring.shardingsphere.sharding.default-data-source-name=ds1
#配置主从名称
spring.shardingsphere.masterslave.name=ms
#置主库master,负责数据的写入
spring.shardingsphere.masterslave.master-data-source-name=ds1
#配置从库slave节点
spring.shardingsphere.masterslave.slave-data-source-names=ds2,ds3
#配置slave节点的负载均衡均衡策略,采用轮询机制
spring.shardingsphere.masterslave.load-balance-algorithm-type=round_robin

#整合mybatis的配置
mybatis.type-aliases-package=com.ppdai.shardingjdbc.entity
```

3）创建t_user表

```
CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `nickname` varchar(100) DEFAULT NULL,
  `password` varchar(100) DEFAULT NULL,
  `sex` int(11) DEFAULT NULL,
  `birthday` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb4;
```

4）定义Controller、Mapper、Entity

```
@Data
public class User {
    private Integer id;

    private String nickname;

    private String password;

    private Integer sex;

    private String birthday;
}
@RestController
@RequestMapping("/api/user")
public class UserController {
    @Autowired
    private UserMapper userMapper;

    @PostMapping("/save")
    public String addUser() {
        User user = new User();
        user.setNickname("zhangsan" + new Random().nextInt());
        user.setPassword("123456");
        user.setSex(1);
        user.setBirthday("1997-12-03");
        userMapper.addUser(user);
        return "success";
    }

    @GetMapping("/findUsers")
    public List<User> findUsers() {
        return userMapper.findUsers();
    }
}
public interface UserMapper {

    @Insert("insert into t_user(nickname,password,sex,birthday) values(#{nickname},#{password}
    void addUser(User user);

    @Select("select * from t_user")
    List<User> findUsers();
}
```

5）验证

启动日志中三个数据源初始化成功：

```
2021-03-22 07:05:23.471  INFO 57725 --- [           main] com.alibaba.druid.pool.DruidDataSource   : {dataSource-1} inited
2021-03-22 07:05:23.920  INFO 57725 --- [           main] com.alibaba.druid.pool.DruidDataSource   : {dataSource-2} inited
2021-03-22 07:05:25.162  INFO 57725 --- [           main] com.alibaba.druid.pool.DruidDataSource   : {dataSource-3} inited
```

调用 `http://localhost:8080/api/user/save` 一直进入到ds1主节点。

```
2021-03-22 07:08:15.717  INFO 57725 --- [nio-8080-exec-4] ShardingSphere-SQL                       : Rule Type: master-slave
2021-03-22 07:08:15.717  INFO 57725 --- [nio-8080-exec-4] ShardingSphere-SQL                       : SQL: insert into t_user(nickname,
 password,sex,birthday) values(?,?,?,?) ::: DataSources: ds1
```

调用 `http://localhost:8080/api/user/findUsers` 一直进入到ds2、ds3节点，并且轮询进入。

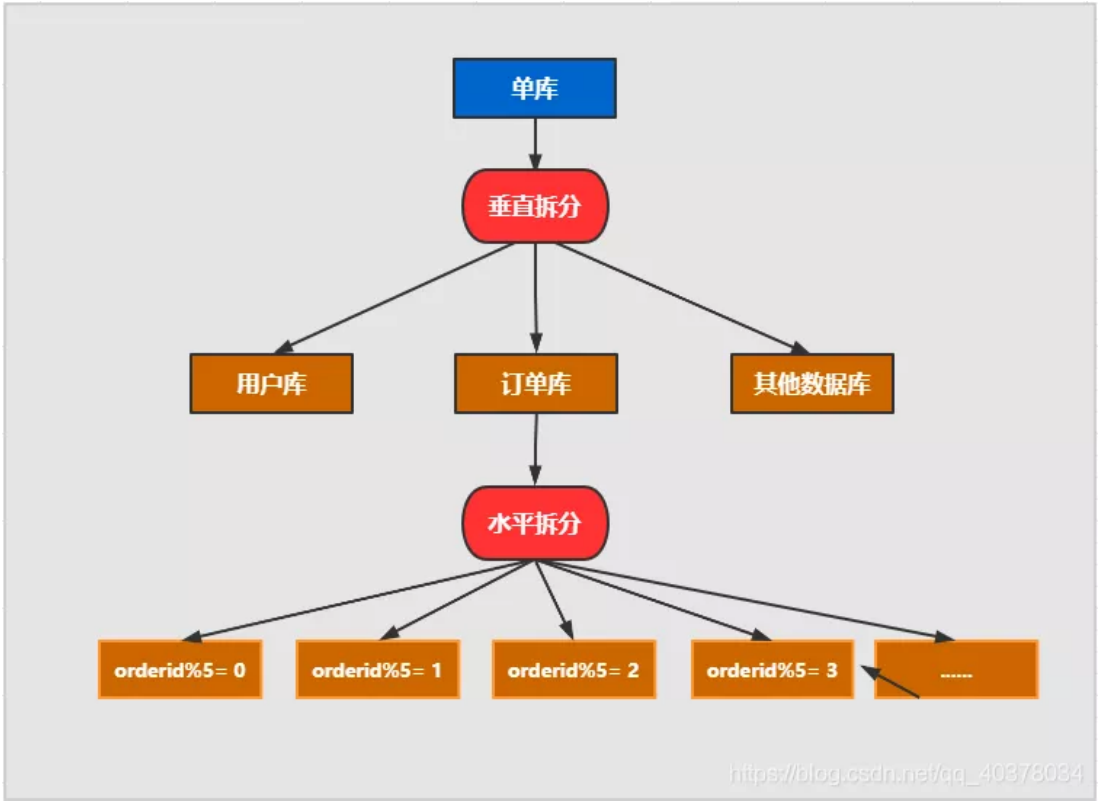
```
2021-03-22 07:09:24.245  INFO 57725 --- [nio-8080-exec-7] ShardingSphere-SQL                       : Rule Type: master-slave
2021-03-22 07:09:24.245  INFO 57725 --- [nio-8080-exec-7] ShardingSphere-SQL                       : SQL: select * from t_user :::
 DataSources: ds2
2021-03-22 07:09:25.747  INFO 57725 --- [nio-8080-exec-8] ShardingSphere-SQL                       : Rule Type: master-slave
2021-03-22 07:09:25.748  INFO 57725 --- [nio-8080-exec-8] ShardingSphere-SQL                       : SQL: select * from t_user :::
 DataSources: ds3
```

| MySQL分库分表原理

1）分库分表

水平拆分：同一个表的数据拆到不同的库不同的表中。可以根据时间、地区或某个业务键维度，也可以通过hash进行拆分，最后通过路由访问到具体的数据。拆分后的每个表结构保持一致

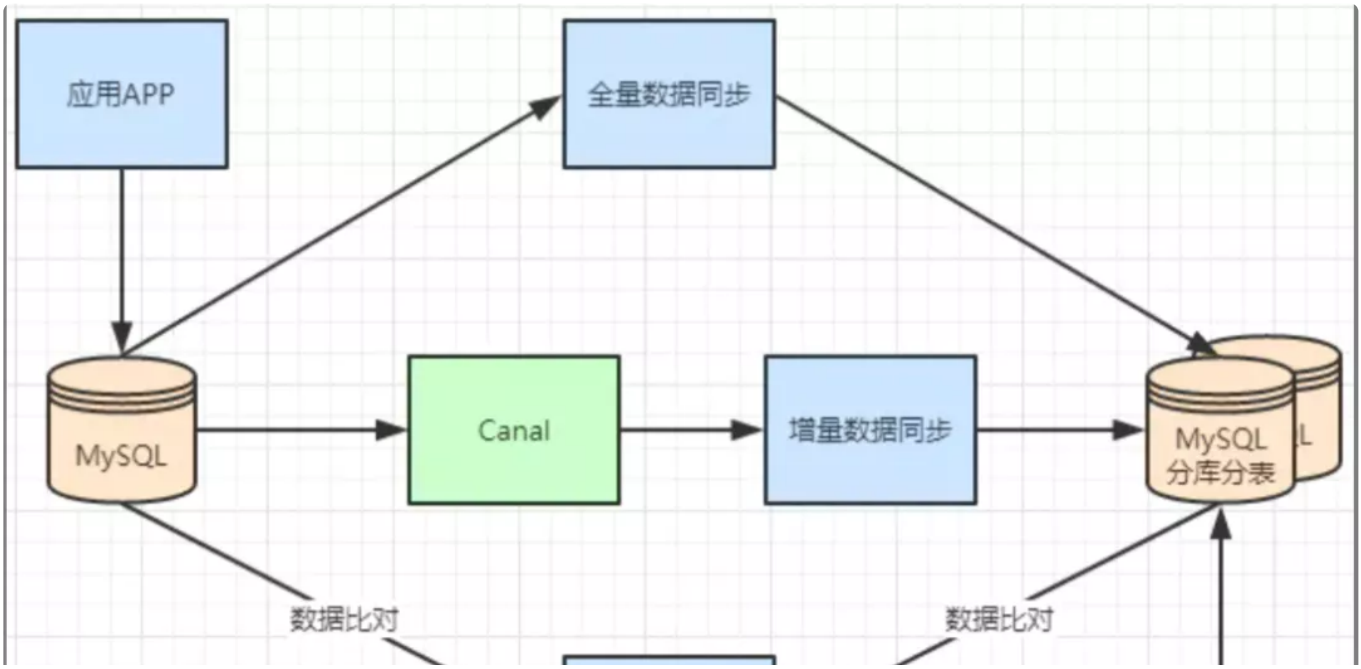
垂直拆分：就是把一个有很多字段的表给拆分成多个表，或者是多个库上去。每个库表的结构都不一样，每个库表都包含部分字段。一般来说，可以根据业务维度进行拆分，如订单表可以拆分为订单、订单支持、订单地址、订单商品、订单扩展等表；也可以，根据数据冷热程度拆分，20%的热点字段拆到一个表，80%的冷字段拆到另外一个表。



2）不停机分库分表数据迁移

一般数据库的拆分也是有一个过程的，一开始是单表，后面慢慢拆成多表。那么我们就看下如何平滑的从MySQL单表过度到MySQL的分库分表架构。

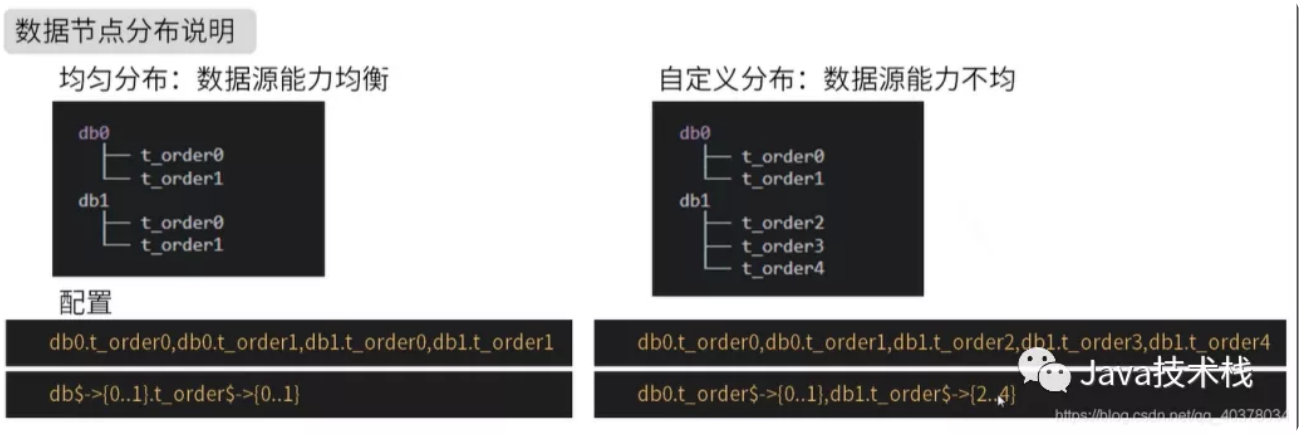
- 利用MySQL+Canal做增量数据同步，利用分库分表中间件，将数据路由到对应的新表中。
- 利用分库分表中间件，全量数据导入到对应的新表中。
- 通过单表数据和分库分表数据两两比较，更新不匹配的数据到新表中。
- 数据稳定后，将单表的配置切换到分库分表配置上。



| Sharding-Jdbc实现分库分表

1）逻辑表

用户数据根据订单id%2拆分为2个表，分别是：t_order0和t_order1。他们的逻辑表名是： `t_order` 。



多数据源相同表：

```
#多数据源$->{0..N}.逻辑表名$->{0..N} 相同表
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds$->{0..1}.t_order$->{0..1}
```

多数据源不同表：

```
#多数据源$->{0..N}.逻辑表名$->{0..N} 不同表
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds0.t_order$->{0..1},ds1.t_ord
```

单库分表：

```
#单数据源的配置方式
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds0.t_order$->{0..4}
```

全部手动指定：

```
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds0.t_order0,ds1.t_order0,ds0.
```

2）inline分片策略

```
spring.shardingsphere.sharding.tables.t_order.actual-data-nodes=ds$->{0..1}.t_order$->{0..1}
#数据源分片策略
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.sharding-column=user_id
#数据源分片算法
spring.shardingsphere.sharding.tables.t_order.database-strategy.inline.algorithm-expression=ds
#表分片策略
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.sharding-column=order_id
#表分片算法
spring.shardingsphere.sharding.tables.t_order.table-strategy.inline.algorithm-expression=t_ord
```

上面的配置通过user_id%2来决定具体数据源，通过order_id%2来决定具体表。

`insert into t_order(user_id,order_id) values(2,3)`，`user_id%2 = 0` 使用数据源ds0，`order_id%2 = 1` 使用t_order1，insert语句最终操作的是数据源ds0的t_order1表。

3）分布式主键配置

Sharding-Jdbc可以配置分布式主键生成策略。默认使用雪花算法，生成64bit的长整型数据，也支持UUID的方式。

```
#主键的列名
spring.shardingsphere.sharding.tables.t_order.key-generator.column=id
#主键生成策略
spring.shardingsphere.sharding.tables.t_order.key-generator.type=SNOWFLAKE
```

4）inline分片策略实现分库分表

需求：

对1000w的用户数据进行分库分表，对用户表的数据进行分表和分库的操作。根据年龄奇数存储在t_user1，偶数t_user0，同时性别奇数存储在ds1，偶数ds0。

表结构：

```
CREATE TABLE `t_user0` (
  `id` bigint(20) DEFAULT NULL,
  `nickname` varchar(200) DEFAULT NULL,
  `password` varchar(200) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `sex` int(11) DEFAULT NULL,
  `birthday` varchar(100) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `t_user1` (
  `id` bigint(20) DEFAULT NULL,
  `nickname` varchar(200) DEFAULT NULL,
  `password` varchar(200) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `sex` int(11) DEFAULT NULL,
  `birthday` varchar(100) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

两个数据库中都包含 `t_user0` 和 `t_user1` 两张表。

application.properties：

```
spring.main.allow-bean-definition-overriding=true
#显示sql
spring.shardingsphere.props.sql.show=true

#配置数据源
spring.shardingsphere.datasource.names=ds0,ds1

#ds0数据库连接信息
spring.shardingsphere.datasource.ds0.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds0.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds0.url=jdbc:mysql://47.101.58.187:3306/t_user_db0?useUnicode
spring.shardingsphere.datasource.ds0.username=root
spring.shardingsphere.datasource.ds0.password=123456
spring.shardingsphere.datasource.ds0.maxPoolSize=100
spring.shardingsphere.datasource.ds0.minPoolSize=5

#ds1数据库连接信息
spring.shardingsphere.datasource.ds1.type=com.alibaba.druid.pool.DruidDataSource
spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.cj.jdbc.Driver
spring.shardingsphere.datasource.ds1.url=jdbc:mysql://47.101.58.187:3306/t_user_db1?useUnicode
spring.shardingsphere.datasource.ds1.username=root
spring.shardingsphere.datasource.ds1.password=123456
```

```
spring.shardingsphere.datasource.ds1.maxPoolSize=100
spring.shardingsphere.datasource.ds1.minPoolSize=5

#整合mybatis的配置
mybatis.type-aliases-package=com.ppdai.shardingjdbc.entity

spring.shardingsphere.sharding.tables.t_user.actual-data-nodes=ds$->{0..1}.t_user$->{0..1}
#数据源分片策略
spring.shardingsphere.sharding.tables.t_user.database-strategy.inline.sharding-column=sex
#数据源分片算法
spring.shardingsphere.sharding.tables.t_user.database-strategy.inline.algorithm-expression=ds$
#表分片策略
spring.shardingsphere.sharding.tables.t_user.table-strategy.inline.sharding-column=age
#表分片算法
spring.shardingsphere.sharding.tables.t_user.table-strategy.inline.algorithm-expression=t_user
#主键的列名
spring.shardingsphere.sharding.tables.t_user.key-generator.column=id
spring.shardingsphere.sharding.tables.t_user.key-generator.type=SNOWFLAKE
```

测试类：

```
@SpringBootTest
class ShardingJdbcApplicationTests {

    @Autowired
    private UserMapper userMapper;

    /**
     * sex:奇数
     * age:奇数
     * ds1.t_user1
     */
    @Test
    public void test01() {
        User user = new User();
        user.setNickname("zhangsan" + new Random().nextInt());
        user.setPassword("123456");
        user.setAge(17);
        user.setSex(1);
        user.setBirthday("1997-12-03");
        userMapper.addUser(user);
    }

    /**
     * sex:奇数
     * age:偶数
     * ds1.t_user0
     */
    @Test
```