订单系统分库分表方案设计与实现

分库分表

除了访问MySQL的并发问题，还要解决海量数据的问题，很多的时候，我们会使用分布式的存储集群，因为MySQI本质上是一个单机数据库，所以很多场景下，其并不适合存储TB级别以上的数据。

但是绝大部分电商企业的在线交易类业务，比如订单、支付相关的系统，还是无法离开MySQL的。原因是只有MySOL 之类的关系型数据库，才能提供金融级的事务保证。目前的分布式事务的各种解法方案多少都有些不够完善。

虽然 MySQL 无法支持这么大的数据量，以及这么高的并发需求，但是交易类系统必须用它来保证数据一致性，那么，如何才能解决这个问题呢？这个时候我们就要考虑分片，也就是拆分数据。

如果一个数据库无法支撑1TB的数据，那就把它拆分成100个库，每个库就只有10GB的数据了。这种拆分操作就是MySOL的分库分表操作。

如何规划分库分表

以订单表为例，首先,我们需要思考的问题是，选择分库还是分表，或者两者都有，分库就是把数据拆分到不同的MySQL 数据库实例中，分表就是把数据拆分到一个数据库的多张表里面。

在考虑到底是选择分厍还是分表之前,我们需要首先明确一个原则，能不拆就不拆。原因很简单，数据拆得越分散,并发和维护就越麻烦，系统出问题的概率也就越大。

遵循上面这个原则，还需要进一步了解，哪种情况适合分表，哪种情况适合分库。选择分厍或是分表的目的是解决如下两个问题。

第一，是为了解决因数据量太大而导致查询慢的问题。这里所说的“查询”，其实主要是事务中的查询和更新操作，因为只读的查询可以通过缓存和主从分离来解决。分表主要用于解决因数据量大而导致的查询慢的问题。

第二，是为了应对高并发的问题。如果一个数据库实例撑不住，就把并发请求分散到多个实例中，所以分库可用于解决高并发的问题。

简单地说，如果数据量太大，就分表；如果并发请求量高，就分库。一般情况下,我们的解决方案大都需要同时做分库分表,我们可以根据预估的并发量和数据量，分别计算应该拆分成多少个库以及多少张表。

商城订单服务的实现

数据量

在设计系统，我们预估订单的数量每个月订单2000W，一年的订单数可达2.4亿。而每条订单的大小大致为1KB，按照我们在MySQL中学习到的知识，为了让B+树的高度控制在一定范围，保证查询的性能，每个表中的数据不宜超过2000W。在这种情况下，为了存下2.4亿的订单，我们似乎应该将订单表分为16（12往上取最近的2的幂）张表。

但是这样设计，有个问题，我们只考虑了订单表，没有考虑订单详情表。我们预估一张订单下的商品平均为10个，那既是一年的订单详情数可以达到24亿，同样以每表2000W记录计算，应该订单详情表为128（120往上取最近的2的幂）张，而订单表和订单详情表虽然记录数上是一对一的关系，但是表之间还是一对一，也就是说订单表也要为128张。经过再三分析，我们最终将订单表和订单详情表的张数定为32张。

这会导致订单详情表单表的数据量达到8000W，为何要这么设计呢？原因我们后面再说。

选择分片键

既然决定订单系统分库分表，则还有一个重要的问题，那就是如何选择一个合适的列作为分表的依据，该列我们一般称为分片键（Sharding Key)。选择合适的分片键和分片算法非常重要，因为其将直接影响分库分表的效果。

选择分片链有一个最重要的参考因素是我们的业务是如何访问数据的?

比如我们把订单ID作为分片键来诉分订单表。那么拆分之后,如果按照订单ID来查询订单,就需要先根据订单ID和分片算法,计算所要查的这个订单具体在哪个分片上，也就是哪个库的哪张表中，然后再去那个分片执行查询操作即可。

但是当用户打开“我的订单”这个页面的时候，它的查询条件是用户ID，由于这里没有订单ID，因此我们无法知道所要查询的订单具体在哪个分片上，也就没法查了。如果要强行查询的话，那就只能把所有的分片都查询一遍，再合并查询结果，这个过程比较麻烦，而且性能很差，对分页也很不友好。

那么如果是把用户ID作为分片键呢？答案是也会面临同样的问题，使用订单ID作为查询条件时无法定位到具体的分片上。

这个问题的解决办法是，在生成订单ID的时候，把用户ID的后几位作为订单ID的一部分。这样按订单ID查询的时候，就可以根据订单ID中的用户ID找到分片。 所以在我们的系统中订单ID从唯一ID服务获取ID后，还会将用户ID的后两位拼接，形成最终的订单ID。

```
/**
 * 生成订单的orderId
 * @param memberId 用戶ID
 */
2 usages   Mark
public Long generateOrderId(Long memberId){
    String leafOrderId = unqidFeignApi.getSegmentId(OrderConstant.LEAF_ORDER_ID_KEY);
    String strMemberId = memberId.toString();
    String OrderIdTail = memberId < 10 ? "0" + strMemberId
            : strMemberId.substring( beginIndex: strMemberId.length() - 2);
    log.debug("生成订单的orderId，组成元素为：{},{}",leafOrderId,OrderIdTail);
    return Long.valueOf( s: leafOrderId + OrderIdTail);
}
```

然而，系统对订单的查询方式，肯定不只是按订单ID或按用户ID查询两种方式。比如如果有商家希望查询自家家店的订单，有与订单相关的各种报表。对订单做了分库分表，就没法解决了。这个问题又该怎么解决呢?

一般的做法是，把订单里数据同步到其他存储系统中，然后在其他存储系统里解决该问题。比如可以再构建一个以店铺ID作为分片键的只读订单库，专供商家使用。或者数据同步到Hadoop分布式文件系统（HDFS）中，然后通过一些大数据技术生成与订单相关的报表。

在分片算法上，我们知道常用的有按范围，比如时间范围分片，哈希分片，查表法分片。我们这里直接使用哈希分片，对表的个数32直接取模

```
/* 对可用的表名求余数，获取到真实的表的后缀*/
.map(idSuffix -> idSuffix % availableTargetNames.size())
```

一旦做了分库分表，就会极大地限制数据库的查询能力，原本很简单的查询，分库分表之后，可能就没法实现了。分库分表一定是在数据量和并发请求量大到所有招数都无效的情况下，我们才会采用的最后一招。

具体实现

如何在代码中实现读写分离和分库分表呢？一般来说有三种方法。

1）纯手工方式：修改应用程序的DAO层代码，定义多个数据源，在代码中需要访问数据库的每个地方指定每个数据库请求的数据源。

2）组件方式：使用像Sharding-JDBC 这些组件集成在应用程序内，用于代理应用程序的所有数据库请求，并把请求自动路由到对应的数据库实例上。

3）代理方式:在应用程序和数据库实例之间部署一组数据库代理实例,比如Atlas或Sharding-Proxy。对于应用程序来说,数据库代理把自己伪装成一个单节点的MySQL实例,应用程序的所有数据库请求都将发送给代理，代理分离请求，然后将分离后的请求转发给对应的数据库实例。

在这三种方式中一般推荐第二种，使用分离组件的方式。采用这种方式,代码侵入非常少,同时还能兼顾性能和稳定性。如果应用程序是一个逻辑非常简单的微服务,简单到只有几个SQL,或者应用程序使用的编程语言没有合适的读写分离组件,那么也可以考虑通过纯手工的方式。

不推荐使用代理方式(第三种方式），原因是代理方式加长了系统运行时数据库请求的调用链路,会浩成一定的性能损失，而且代理服务本身也可能会出现故障和性能瓶颈等问题。代理方式有一个好处，对应用程序完全透明。

所以在我们的订单服务中，使用了第二种方式，引入了Sharding-JDBC，考虑要同时支持读写分离和分库分表，配置如下：

```yaml
#分库分表配置
shardingsphere:
  #数据源配置
  datasource:
    names: ds-master,ds-slave
    ds-master:
      type: com.alibaba.druid.pool.DruidDataSource
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://192.168.65.223:3306/tl_mall_order?serverTimezone=UTC&useSSL=false&useUnicode=tr
      initialSize: 5
      minIdle: 10
      maxActive: 30
      validationQuery: SELECT 1 FROM DUAL
      username: tlmall
      password: tlmall123
    ds-slave:
      type: com.alibaba.druid.pool.DruidDataSource
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://192.168.65.137:3306/tl_mall_order?serverTimezone=UTC&useSSL=false&useUnicode=tr
      initialSize: 5
      minIdle: 10
      maxActive: 30
      validationQuery: SELECT 1 FROM DUAL
      username: tlmall
      password: tlmall123
```

```yaml
sharding:
  default-data-source-name: ds-master
  default-database-strategy:
    none:
  tables:
    oms_order:
      actual-data-nodes: ds_ms.oms_order_$->{0..31}
      table-strategy:
        complex:
          sharding-columns: id,member_id
          algorithm-class-name: com.tuling.tulingmall.ordercurr.sharding.OmsOrderShardingAlgorithm
    oms_order_item:
      actual-data-nodes: ds_ms.oms_order_item_$->{0..31}
      table-strategy:
        complex:
          sharding-columns: order_id
          algorithm-class-name: com.tuling.tulingmall.ordercurr.sharding.OmsOrderItemShardingAlgorithm
  binding-tables:
    - oms_order,oms_order_item
  broadcastTables:
    - oms_company_address
    - oms_order_operate_history
    - oms_order_return_apply
    - oms_order_return_reason
    - oms_order_setting
  #读写分离配置
  master-slave-rules:
    ds_ms:
      master-data-sourceName: ds-master
      slave-data-sourceNames:
        - ds-slave
      load-balance-algorithmType: ROUND_ROBIN
props:
  sql:
    show: true
```

在分片键的选择上，订单信息的查询往往会指定订单的ID或者用户ID，所以oms_order的分片键为表中的id、member_id两个字段。而oms_order_item表通过order_id字段和oms_order的id进行关联，所以它的分片键选择为order_id。对应在代码中有专门的分片算法实现类：OmsOrderShardingAlgorithm和OmsOrderItemShardingAlgorithm，分别用于对订单和订单详情进行分片。

```
 /*获取订单编号*/
Collection<String> orderSns = complexKeysShardingValue.getColumnNameAndShardingValuesMap()
        .getOrDefault(COLUMN_ORDER_SHARDING_KEY, new ArrayList<>( initialCapacity: 1));
/* 获取客戶id*/
Collection<String> customerIds = complexKeysShardingValue.getColumnNameAndShardingValuesMap()
        .getOrDefault(COLUMN_CUSTOMER_SHARDING_KEY, new ArrayList<>( initialCapacity: 1));

/*合并订单id和客戶id到一个容器中*/
List<String> ids = new ArrayList<>( initialCapacity: 16);
if (Objects.nonNull(orderSns)) ids.addAll(ids2String(orderSns));
if (Objects.nonNull(customerIds)) ids.addAll(ids2String(customerIds));

return ids.stream() Stream<String>
        /*截取 订单号或客戶id的后2位*/
        .map(id -> id.substring( beginIndex: id.length() - 2))
        /* 去重*/
        .distinct()
        /* 转换成int*/
        .map(Integer::new) Stream<Integer>
        /* 对可用的表名求余数，获取到真实的表的后缀*/
        .map(idSuffix -> idSuffix % availableTargetNames.size())
        /*转换成string*/
        .map(String::valueOf) Stream<String>
        /* 获取到真实的表*/
        .map(tableSuffix -> availableTargetNames.stream().
                filter(targetName -> targetName.endsWith(tableSuffix)).findFirst().orElse( other: null))
        .filter(Objects::nonNull)
        .collect(Collectors.toList());
```

其中的OmsOrderShardingAlgorithm负责对订单进行分片，在实现上获得订单的Id或者member_id的后两位，然后对表的个数进行取模以定位到实际的物理oms_order表。OmsOrderItemShardingAlgorithm负责对订单详情进行分片，实现上与OmsOrderShardingAlgorithm类似。

> 有道云笔记链接：https://note.youdao.com/s/IF3Le454