老肉

关于分库分表最全的一篇文章

这里介绍设计分库分表框架时应该考虑的设计要点，并给出相应的解决方案。

一、整体的切分方式

简单来说，数据的切分就是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）中，以达到分散单台设备负载的效果，即分库分表。

数据的切分根据其切分规则的类型，可以分为如下两种切分模式。

- 垂直（纵向）切分：把单一的表拆分成多个表，并分散到不同的数据库（主机）上。
- 水平（横向）切分：根据表中数据的逻辑关系，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。

1. 垂直切分

一个数据库由多个表构成，每个表对应不同的业务，垂直切分是指按照业务将表进行分类，将其分布到不同的数据库上，这样就将数据分担到了不同的库上（专库专用）。

案例如下：

#有如下几张表

--------------+--------------+------------------

用户信息(User)+ 交易记录(Pay)+ 商品(Commodity)|

--------------+--------------+------------------

针对以上案例，垂直切分就是根据每个表的不同业务进行切分，比如User表、Pay表和Commodity表，将每个表切分到不同的数据库上。

垂直切分的优点如下：

- 拆分后业务清晰，拆分规则明确。
- 系统之间进行整合或扩展很容易。

- 按照成本、应用的等级、应用的类型等将表放到不同的机器上，便于管理。
- 便于实现动静分离、冷热分离的数据库表的设计模式。
- 数据维护简单。

垂直切分的缺点如下：

- 部分业务表无法关联（Join），只能通过接口方式解决，提高了系统的复杂度。
- 受每种业务的不同限制，存在单库性能瓶颈，不易进行数据扩展和提升性能。
- 事务处理复杂。

垂直切分除了用于分解单库单表的压力，也用于实现冷热分离，也就是根据数据的活跃度进行拆分，因为对拥有不同活跃度的数据的处理方式不同。

我们可将本来可以在同一个表中的内容人为地划分为多个表。所谓“本来”，是指按照关系型数据库第三范式的要求，应该在同一个表中，将其拆分开就叫作反范化（Denormalize）。

例如，对配置表的某些字段很少进行修改时，将其放到一个查询性能较高的数据库硬件上；对配置表的其他字段更新频繁时，则将其放到另一个更新性能较高的数据库硬件上。

这里我们再举一个例子：在微博系统的设计中，一个微博对象包括文章标题、作者、分类、创建时间等属性字段，这些字段的变化频率低，查询次数多，叫作冷数据。而博客的浏览量、回复数、点赞数等类似的统计信息，或者别的变化频率比较高的数据，叫作活跃数据或者热数据。

我们把冷热数据分开存放，就叫作冷热分离，在MySQL的数据库中，冷数据查询较多，更新较少，适合用MyISAM引擎，而热数据更新比较频繁，适合使用InnoDB存储引擎，这也是垂直拆分的一种。

我们推荐在设计数据库表结构时，就考虑垂直拆分，根据冷热分离、动静分离的原则，再根据使用的存储引擎的特点，对冷数据可以使用MyISAM，能更好地进行数据查询；对热数据可以使用InnoDB，有更快的更新速度，这样能够有效提升性能。

其次，对读多写少的冷数据可配置更多的从库来化解大量查询请求的压力；对于热数据，可以使用多个主库构建分库分表的结构，请参考下面关于水平切分的内容，后续的三四五章提供了不同的分库分表的具体实施方案。

注意，对于一些特殊的活跃数据或者热点数据，也可以考虑使用Memcache、Redis之类的缓存，等累计到一定的量后再更新数据库，例如，在记录微博点赞数量的业务中，点赞数量被存储在缓存中，每增加1000个点赞，才写一次数据。

2. 水平切分

与垂直切分对比，水平切分不是将表进行分类，而是将其按照某个字段的某种规则分散到多个库中，在每个表中包含一部分数据，所有表加起来就是全量的数据。

简单来说，我们可以将对数据的水平切分理解为按照数据行进行切分，就是将表中的某些行切分到一个数据库表中，而将

embed(1)
Java(1)
工作流(1)
跨服务器连接查询(1)

随笔分类

Android(5)
asp.net(67)
C#(31)
div+css+html(24)
hadoop(1)
Java(9)
jquery/js(15)
JS/jquery特效(3)
LINQ(1)
Linux(3)
MVC(6)
NoSql(4)
sprint.net(1)
sql(25)
WebSocket(2)
安卓(4)
报错汇总(4)
大数据(2)
多线程(3)

其他行切分到其他数据库表中。

这种切分方式根据单表的数据量的规模来切分，保证单表的容量不会太大，从而保证了单表的查询等处理能力，例如将用户的信息表拆分成User1、User2等，表结构是完全一样的。我们通常根据某些特定的规则来划分表，比如根据用户的ID来取模划分。

例如，在博客系统中，当读取博客的量很大时，就应该采取水平切分来减少每个单表的压力，并提升性能。

以微博表为例，当同时有100万个用户在浏览时，如果是单表，则单表会进行100万次请求，假如是单库，数据库就会承受100万次的请求压力；假如将其分为100个表，并且分布在10个数据库中，每个表进行1万次请求，则每个数据库会承受10万次的请求压力，虽然这不可能绝对平均，但是可以说明问题，这样压力就减少了很多，并且是成倍减少的。

水平切分的优点如下：

- 单库单表的数据保持在一定的量级，有助于性能的提高。
- 切分的表的结构相同，应用层改造较少，只需要增加路由规则即可。
- 提高了系统的稳定性和负载能力。

水平切分的缺点如下：

- 切分后，数据是分散的，很难利用数据库的Join操作，跨库Join性能较差。
- 拆分规则难以抽象。
- 分片事务的一致性难以解决。
- 数据扩容的难度和维护量极大。

综上所述，垂直切分和水平切分的共同点如下：

- 存在分布式事务的问题。
- 存在跨节点Join的问题。
- 存在跨节点合并排序、分页的问题。
- 存在多数据源管理的问题。

在了解这两种切分方式的特点后，我们就可以根据自己的业务需求来选择，通常会同时使用这两种切分方式，垂直切分更偏向于业务拆分的过程，在技术上我们更关注水平切分的方案。

二、水平切分方式的路由过程和分片维度

这里讲解水平切分的路由过程和分片维度。

1. 水平切分的路由过程

我们在设计表时需要确定对表按照什么样的规则进行分库分表。例如，当有新用户时，程序得确定将此用户的信息添加到哪个表中；同理，在登录时我们需要通过用户的账号找到数据库中对应的记录，所有这些都需要按照某一规则进行路由请

开源(1)
领域驱动设计DDD(1)
设计模式(4)
数据结构与算法(1)
四色原型(1)
网址(3)
系统架构(13)
系统监控(3)

随笔档案

2020年2月(1)
2019年12月(4)
2019年11月(1)
2019年10月(5)
2019年9月(2)
2019年8月(5)
2019年7月(3)
2018年11月(1)
2018年10月(4)
2018年8月(2)
2018年7月(5)
2018年5月(2)
2018年4月(1)
2018年3月(1)
2017年12月(1)

求，因为请求所需要的数据分布在不同的分片表中。

针对输入的请求，通过分库分表规则查找到对应的表和库的过程叫作路由。例如，分库分表的规则是user_id % 4，当用户新注册了一个账号时，假设用户的ID是123，我们就可以通过123 % 4 = 3确定此账号应该被保存在User3表中。当ID为123的用户登录时，我们可通过123 % 4 = 3计算后，确定其被记录在User3中。

2. 水平切分的分片维度

对数据切片有不同的切片维度，可以参考Mycat提供的切片方式（见本书3.4节），这里只介绍两种最常用的切片维度。

1）按照哈希切片

对数据的某个字段求哈希，再除以分片总数后取模，取模后相同的数据为一个分片，这样的将数据分成多个分片的方法叫作哈希分片。

按照哈希分片常常应用于数据没有时效性的情况，比如所有数据无论是在什么时间产生的，都需要进行处理或者查询，例如支付行业的客户要求可以对至少1年以内的交易进行查询和退款，那么1年以内的所有交易数据都必须停留在交易数据库中，否则就无法查询和退款。

如果这家公司在一年内能做10亿条交易，假设每个数据库分片能够容纳5000万条数据，则至少需要20个表才能容纳10亿条交易。在路由时，我们根据交易ID进行哈希取模来找到数据属于哪个分片，因此，在设计系统时要充分考虑如何设计数据库的分库分表的路由规则。

这种切片方式的好处是数据切片比较均匀，对数据压力分散的效果较好，缺点是数据分散后，对于查询需求需要进行聚合处理。

2）按照时间切片

与按照哈希切片不同，这种方式是按照时间的范围将数据分布到不同的分片上的，例如，我们可以将交易数据按照月进行切片，或者按照季度进行切片，由交易数据的多少来决定按照什么样的时间周期对数据进行切片。

这种切片方式适用于有明显时间特点的数据，例如，距离现在1个季度的数据访问频繁，距离现在两个季度的数据可能没有更新，距离现在3个季度的数据没有查询需求。

针对这种情况，可以通过按照时间进行切片，针对不同的访问频率使用不同档次的硬件资源来节省成本：假设距离现在1个季度的数据访问频率最高，我们就用更好的硬件来运行这个分片；假设距离现在3个季度的数据没有任何访问需求，我们就可以将其整体归档，以方便DBA操作。

在实际的生产实践中，按照哈希切片和按照时间切片都是常用的分库分表方式，并被广泛使用，有时可以结合使用这两种方式，例如：对交易数据先按照季度进行切片，然后对于某一季度的数据按照主键哈希进行切片。

三、分片后的事务处理机制

本节讲解分片后的事务处理机制。

1. 分布式事务

由于我们将单表的数据切片后存储在多个数据库甚至多个数据库实例中，所以依靠数据库本身的事务机制不能满足所有场景的需要。

但是，我们推荐在一个数据库实例中的操作尽可能使用本地事务来保证一致性，跨数据库实例的一系列更新操作需要根据事务路由在不同的数据源中完成，各个数据源之间的更新操作需要通过分布式事务处理。

这里只介绍实现分布式操作一致性的几个主流思路，保证分布式事务一致性的具体方法请参考《分布式服务架构：原理、设计与实战》中第2章的内容。

主流的分布式事务解决方案有三种：两阶段提交协议、最大努力保证模式和事务补偿机制。

1）两阶段提交协议

两阶段提交协议将分布式事务分为两个阶段，一个是准备阶段，一个是提交阶段，两个阶段都由事务管理器发起。

基于两阶段提交协议，事务管理器能够最大限度地保证跨数据库操作的事务的原子性，是分布式系统环境下最严格的事务实现方法。符合J2EE规范的AppServer（例如：Websphere、Weblogic、Jboss等）对关系型数据库数据源和消息队列都实现了两阶段提交协议，只需在使用时配置即可。

但是，两阶段提交协议也带来了性能方面的问题，难于进行水平伸缩，因为在提交事务的过程中，事务管理器需要和每个参与者进行准备和提交的操作的协调，在准备阶段锁定资源，在提交阶段消费资源。

但是由于参与者较多，锁定资源和消费资源之间的时间差被拉长，导致响应速度较慢，在此期间产生死锁或者不确定结果的可能性较大。因此，在互联网行业里，为了追求性能的提升，很少使用两阶段提交协议。

另外，由于两阶段提交协议是阻塞协议，在极端情况下不能快速响应请求方，因此有人提出了三阶段提交协议，解决了两阶段提交协议的阻塞问题，但仍然需要事务管理器在参与者之间协调，才能完成一个分布式事务。

2）最大努力保证模式

这是一种非常通用的保证分布式一致性的模式，很多开发人员一直在使用，但是并未意识到这是一种模式。最大努力保证模式适用于对一致性要求并不十分严格但是对性能要求较高的场景。

具体的实现方法是，在更新多个资源时，将多个资源的提交尽量延后到最后一刻处理，这样的话，如果业务流程出现问题，则所有的资源更新都可以回滚，事务仍然保持一致。

唯一可能出现问题的情况是在提交多个资源时发生了系统问题，比如网络问题等，但是这种情况是非常罕见的，一旦出现这种情况，就需要进行实时补偿，将已提交的事务进行回滚，这和我们常说的TCC模式有些类似。

2014年2月(4)
2014年1月(2)
2013年12月(7)
2013年11月(1)
2013年10月(2)
2013年9月(2)
2013年8月(2)
2013年4月(2)
2013年2月(1)
2013年1月(4)
2012年12月(1)
2012年5月(1)
2012年3月(1)
2012年2月(2)
2012年1月(1)
2011年12月(1)
2011年11月(11)
2011年5月(2)
2010年12月(4)
2010年11月(16)
2010年10月(51)
2010年9月(1)

文章分类

领域驱动设计DDD

下面是使用最大努力保证模式的一个样例，在该样例中涉及两个操作，一个是从消息队列消费消息，一个是更新数据库，需要保证分布式的一致性。

（1）开始消息事务。

（2）开始数据库事务。

（3）接收消息。

（4）更新数据库。

（5）提交数据库事务。

（6）提交消息事务。

这时，从第1步到第4步并不是很关键，关键的是第5步和第6步，需要将其放在最后一起提交，尽最大努力保证前面的业务处理的一致性。

到了第5步和第6步，业务逻辑处理完成，这时只可能发生系统错误，如果第5步失败，则可以将消息队列和数据库事务全部回滚，保持一致。如果第5步成功，第6步遇到了网络超时等问题，则这是唯一可能产生问题的情况。

在这种情况下，消息的消费过程并没有被提交到消息队列，消息队列可能会重新发送消息给其他消息处理服务，这会导致消息被重复消费，但是可以通过幂等处理来保证消除重复消息带来的影响。

当然，在使用这种模式时，我们要充分考虑每个资源的提交顺序。我们在生产实践中遇到的一种反模式，就是在数据库事务中嵌套远程调用，而且远程调用是耗时任务，导致数据库事务被拉长，最后拖垮数据库。

因此，上面的案例涉及的是消息事务嵌套数据库事务，在这里必须进行充分评估和设计，才可以规避事务风险。

3）事务补偿机制

显然，在对性能要求很高的场景中，两阶段提交协议并不是一种好方案，最大努力保证模式也会使多个分布式操作互相嵌套，有可能互相影响。这里，我们给出事务补偿机制，其性能很高，并且能够尽最大可能地保证事务的最终一致性。

在数据库分库分表后，如果涉及的多个更新操作在某一个数据库范围内完成，则可以使用数据库内的本地事务保证一致性；对于跨库的多个操作，可通过补偿和重试，使其在一定的时间窗口内完成操作，这样就可以实现事务的最终一致性，突破事务遇到问题就滚回的传统思路。

如果采用事务补偿机制，则在遇到问题时，我们需要记录遇到问题的环境、信息、步骤、状态等，后续通过重试机制使其达到最终一致性，详细内容可以参考《分布式服务架构：原理、设计与实战》第2章，彻底理解ACID原理、CAP理论、BASE原理、最终一致性模式等内容。

2. 事务路由

无论使用上面哪种方法实现分布式事务，都需要对分库分表的多个数据源路由事务，一般通过对Spring环境的配置，为不同的数据源配置不同的事务管理器（TransactionManager）。

这样，如果更新操作在一个数据库实例内发生，便可以使用数据源的事务来处理。对于跨数据源的事务，可通过在应用层使用最大努力保证模式和事务补偿机制来达成事务的一致性。

当然，有时我们需要通过编写程序来选择数据库的事务管理器，根据实现方式的不同，可将事务路由具体分为以下三种。

1）自动提交事务路由

自动提交事务通过依赖JDBC数据源的自动提交事务特性，对任何数据库进行更新操作后会自动提交事务，不需要开发人员手工操作事务，也不需要配置事务，实现起来很简单，但是只能满足简单的业务逻辑需求。

在通常情况下，JDBC在连接创建后默认设置自动提交为true，当然，也可以在获取连接后手工修改这个属性，代码如下：

```
connnection conn = null;
try{
conn = getConnnection();
conn.setAutoCommit(true);
// 数据库操作
……
conn.commit();
}catch(Throwable e){
if(conn!=null){
try {
conn.rollback();
} catch (SQLException e1) {
e1.printStackTrace();
}
}

throw new RuntimeException(e);
}finally{
if(conn!=null){
try {
conn.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
```

}

我们基本不需要使用原始的JDBC API来改变这些属性，这些操作一般都会被封装在我们使用的框架中。本书3.6节介绍的开源数据库分库分表框架dbsplit默认使用的就是这种模式。

2）可编程事务路由

我们在应用中通常采用Spring的声明式的事务来管理数据库事务，在分库分表时，事务处理是个问题，在一个需要开启事务的方法中，需要动态地确定开启哪个数据库实例的事务，也就是说在每个开启事务的方法调用前就必须确定开启哪个数据源的事务。下面使用伪代码来说明如何实现一个可编程事务路由的小框架。

首先，通过Spring配置文件展示可编程事务小框架是怎么使用的：

```
<?xml version="1.0?>
<beans>
<bean id="sharding-db-trx0"class="org.springframework.jdbc.datasource.Data
SourceTransactionManager">
<property name="dataSource">
<ref bean="sharding-db0" />
</property>
</bean>
<bean id="sharding-db-trx1"
class="org.springframework.jdbc.datasource.DataSourceTransactionMana ger">
<property name="dataSource">
<ref bean="sharding-db1" />
</property>
</bean>
<bean id="sharding-db-trx2"
class="org.springframework.jdbc.datasource.DataSourceTransactionMana ger">
<property name="dataSource">
<ref bean="sharding-db2" />
</property>
</bean>
<bean id="sharding-db-trx3"
class="org.springframework.jdbc.datasource.DataSourceTransactionMana ger">
<property name="dataSource">
<ref bean="sharding-db3" />
</property>
</bean>
```

```
<bean id="shardingTransactionManager" class="com.robert.dbsplit.core. ShardingTransactionManager">
<property name="proxyTransactionManagers">
<map value-type="org.springframework.transaction.PlatformTran sactionManager">
<entry key="sharding0" value-ref="sharding-db-trx0" />
<entry key="sharding1" value-ref="sharding-db-trx1" />
<entry key="sharding2" value-ref="sharding-db-trx2" />
<entry key="sharding3" value-ref="sharding-db-trx3" />
</map>
</property>
</bean>

<aop:config>
<aop:advisor advice-ref="txAdvice" pointcut="execution(* com.robert.biz.*insert(..))"/>
<aop:advisor advice-ref="txAdvice" pointcut="execution(* com.robert.biz.*update(..))"/>
<aop:advisor advice-ref="txAdvice" pointcut="execution(* com.robert.biz.*delete(..))"/>
</aop:config>
<tx:advice id="txAdvice" transaction-manager="shardingTransactionManager">
<tx:attributes>
<tx:method name="*" rollback-for="java.lang.Exception"/>
</tx:attributes>
</tx:advice>

</beans>
```

这里使用Spring环境的aop和tx标签来拦截com.robert.biz包下的所有插入、更新和删除的方法，当指定的包的方法被调用时，就会使用Spring提供的事务Advice，Spring的事务Advice(tx:advice)会使用事务管理器来控制事务，如果某个方法发生了异常，那么Spring的事务Advice就会使shardingTransactionManager回滚相应的事务。

我们看到shardingTransactionManager的类型是ShardingTransactionManager，这个类型是我们开发的一个组合的事务管理器，这个事务管理器聚合了所有分片数据库的事务管理器对象，然后根据某个标记来路由到不同的事务管理器中，这些事务管理器用来控制各个分片的数据源的事务。

这里的标记是什么呢？我们在调用方法时，会提前把分片的标记放进ThreadLocal中，然后在ShardingTransactionManager的getTransaction方法被调用时，取得ThreadLocal中存的标记，最后根据标记来判断使用哪个分片数据库的事务管理器对象。

为了通过标记路由到不同的事务管理器，我们设计了一个专门的ShardingContextHolder类，在该类的内部使用了一个ThreadLocal类来指定分片数据库的关键字，在ShardingTransaction Manager中通过取得这个标记来选择具体的分片数据库的事务管理器对象。

因此，这个类提供了setShard和getShard的方法，setShard用于使用者编程指定使用哪个分片数据库的事务管理器，而getShard用于ShardingTransactionManager获取标记并取得分片数据库的事务管理器对象。相关代码如下：

```
public class ShardingContextHolder<T> {
private static final ThreadLocal shardHolder = new ThreadLocal();

public static <T> void setShard(T shard) {
Validate.notNull(shard, "请指定某个分片数据库！");
shardHolder.set(shard);
}

public static <T> T getShard() {
return (T) shardHolder.get();
}
}
```

有了ShardingContextHolder类后，我们就可以在ShardingTransactionManager中根据给定的分片配置将事务操控权路由到不同分片的数据库的事务管理器上，实现很简单，如果在ThreadLocal中存储了某个分片数据库的事务管理器的关键字，就使用那个分片的数据库的事务管理器：

```
public class ShardingTransactionManager implements PlatformTransactionManager {
private Map<Object, PlatformTransactionManager> proxyTransactionManagers =
new HashMap<Object, PlatformTransactionManager>();

protected PlatformTransactionManager getTargetTransactionManager() {
Object shard = ShardingContextHolder.getShard();
Validate.notNull(shard, "必须指定一个路由的shard！");
return targetTransactionManagers.get(shard);
}
public void setProxyTransactionManagers(Map<Object, PlatformTransaction Manager>
targetTransactionManagers) {
this.targetTransactionManagers = targetTransactionManagers;
}

public void commit(TransactionStatus status) throws TransactionException {
getProxyTransactionManager().commit(status);
}

public TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {
return getProxyTransactionManager().getTransaction(definition);
}
```

```
public void rollback(TransactionStatus status) throws TransactionException
{
getProxyTransactionManager().rollback(status);
}
}
```

有了这些使用类，我们的可编程事务路由小框架就实现了，这样在某个具体的服务开始之前，我们就可以使用如下代码来控制使用某个分片的数据库的事务管理器了：

```
RoutingContextHolder.setShard("sharding0");
return userService.create(user);
```

3）声明式事务路由

在上一小节实现了可编程事务路由的小框架，这个小框架通过让开发人员在ThreadLocal中指定数据库分片并编程实现。

大多数分库分表框架会实现声明式事务路由，也就是在实现的服务方法上直接声明事务的处理注解，注解包含使用哪个数据库分片的事务管理器的信息，这样，开发人员就可以专注于业务逻辑的实现，把事务处理交给框架来实现。

下面是笔者在实际的线上项目中实现的声明式事务路由的一个使用实例：

```
@TransactionHint(table = "INVOICE", keyPath = "0.accountId")
public void persistInvoice(Invoice invoice) {
// Save invoice to DB
this.createInvoice(invoice);

for (InvoiceItem invoiceItem : invoice.getItems()) {
invoiceItem.setInvId(invoice.getId());
invoiceItemService.createInvoiceItem(invoice.getAccountId(), invoiceItem);
}

// Save invoice to cache
invoiceCacheService.set(invoice.getAccountId(), invoice.getInvPeriodStart().getTime(),
invoice.getInvPeriodEnd().getTime(),
invoice);

// Update last invoice date to Account
Account account = new Account();
account.setId(invoice.getAccountId());
account.setLstInvDate(invoice.getInvPeriodEnd());
```

accountService.updateAccount(account);

}

在这个实例中，我们开发了一个持久发票的服务方法。持久发票的服务方法用来保存发票信息和发票项的详情信息，这里，发票与发票项这两个领域对象具有父子结构关系。

由于在设计过程中通过账户ID对这个父子表进行分库分表，因此在进行事务路由时，也需要通过账户ID控制使用哪个数据库分片的事务管理器。在这个实例中，我们配置了 TransactionHint，TransactionHint的声明如下：

@Target({ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface TransactionHint {

String table() default "";

String keyPath() default "";

}

可以看到，TransactionHint包含了两个属性，第1个属性table指定这次操作涉及分片的数据库表，第2个属性指定这次操作根据哪个参数的哪个字段进行分片路由。该实例通过table指定了INVOICE表，并通过keyPath指定了使用第1个参数的字段accountId作为路由的关键字。

这里的实现与可编程事务路由的小框架实现类似，在方法persistInvoice被调用时，根据TransactionHint提供的操作的数据库表名称，在Spring环境的配置中找到这个表的分库分表的配置信息，例如：一共分了多少个数据库实例、数据库和表。

下面是在Spring环境中配置的INVOICE表和INVOICE_ITEM表的具体信息，我们看到它们一共使用了两个数据库实例，每个实例有两个库，每个库有8个表，使用水平下标策略。配置如下：

<bean name="billingInvSplitTable" class="com.robert.dbsplit.core.Split Table"init-method="init">

<property name="dbNamePrefix" value="billing_inv"/>

<property name="tableNamePrefix" value="INVOICE"/>

<property name="dbNum" value="2"/>

<property name="tableNum" value="8"/>

<property name="splitStrategyType" value="HORIZONTAL"/>

<property name="splitNodes">

<list>

<ref bean="splitNode0"/>

```
<ref bean="splitNode1"/>
</list>
</property>

<property name="readWriteSeparate" value="true"/>
</bean>

<bean name="billingInvItemSplitTable" class="com.robert.dbsplit.core.SplitTable"
init-method="init">

<property name="dbNamePrefix" value="billing_inv"/>
<property name="tableNamePrefix" value="INVOICE_ITEM"/>

<property name="dbNum" value="2"/>
<property name="tableNum" value="8"/>

<property name="splitStrategyType" value="HORIZONTAL"/>
<property name="splitNodes">
<list>
<ref bean="splitNode0"/>
<ref bean="splitNode1"/>
</list>
</property>

<property name="readWriteSeparate" value="true"/>
</bean>
```

然后，在方法被调用时通过AOP进行拦截，根据TransactionHint配置的路由的主键信息keyPath = "0.accountId"，得知这次根据第0个参数Invoice的accountID字段来路由，根据Invoice的accountID的值来计算这次持久发票表具体涉及哪个数据库分片，然后把这个数据库分片的信息保存到ThreadLocal中。具体的实现代码如下：

```
SimpleSplitJdbcTemplate simpleSplitJdbcTemplate =
(SimpleSplitJdbcTemplate) ReflectionUtil.getFieldValue(field SimpleSplitJdbcTemplate,
invocation.getThis());

Method method = invocation.getMethod();
// Convert to th method of implementation class
method = targetClass.getMethod(method.getName(), method.getParameter Types());

TransactionHint[] transactionHints = method.getAnnotationsByType (TransactionHint.class);
```

```
if (transactionHints == null || transactionHints.length < 1)
throw new IllegalArgumentException("The method " + method + " includes illegal transaction hint.");
TransactionHint transactionHint = transactionHints[0];

String tableName = transactionHint.table();
String keyPath = transactionHint.keyPath();

String[] parts = keyPath.split("\.");
int paramIndex = Integer.valueOf(parts[0]);

Object[] params = invocation.getArguments();
Object splitKey = params[paramIndex];

if (parts.length > 1) {
String[] paths = Arrays.copyOfRange(parts, 1, parts.length);
splitKey = ReflectionUtil.getFieldValueByPath(splitKey, paths);
}

SplitNode splitNode = simpleSplitJdbcTemplate.decideSplitNode(tableName, splitKey);

ThreadContextHolder.INST.setContext(splitNode);
```

ThreadContextHolder是一个单例的对象，在该对象里封装了一个ThreadLocal，用来存储某个方法在某个线程下关联的分片信息：

```
public class ThreadContextHolder<T> {
public static final ThreadContextHolder<SplitNode> INST = new ThreadContextHolder<SplitNode>();

private ThreadLocal<T> contextHolder = new ThreadLocal<T>();

public T getContext() {
return contextHolder.get();
}

public void setContext(T context) {
contextHolder.set(context);
}
}
```

接下来与可编程式事务路由类似，实现一个定制化的事务管理器，在获取目标事务管理器时，通过我们在ThreadLocal中保存的数据库分片信息，获得这个分片数据库的事务管理器，然后返回：

```
public class RoutingTransactionManager implements PlatformTransactionManager {
protected PlatformTransactionManager getTargetTransactionManager() {
SplitNode splitNode = ThreadContextHolder.INST.getContext();
return splitNode.getPlatformTransactionManager();
}

public void commit(TransactionStatus status) throws TransactionException {
getTargetTransactionManager().commit(status);
}

public TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {
return getTargetTransactionManager().getTransaction(definition);
}

public void rollback(TransactionStatus status) throws TransactionException
{
getTargetTransactionManager().rollback(status);
}
}
```

本书3.6节介绍的开源数据库分库分表框架dbsplit是一个分库分表的简单示例实现，在笔者所工作的公司内部有内部版本，在内部版本中实现了声明式事务路由，但是这部分功能并没有开源到dbsplit项目，原因是有些与业务结合的逻辑无法分离。如果感兴趣，则可以加入我们的开源项目开发中。

四、读写分离

在实际应用中的绝大多数情况下读操作远大于写操作。MySQL提供了读写分离的机制，所有写操作必须对应到主库（Master），读操作可以在主库（Master）和从库（Slave）机器上进行。

主库与从库的结构完全一样，一个主库可以有多个从库，甚至在从库下还可以挂从库，这种一主多从的方式可以有效地提高数据库集群的吞吐量。

在DBA领域一般配置主-主-从或者主-从-从两种部署模型。

所有写操作都先在主库上进行，然后异步更新到从库上，所以从主库同步到从库机器有一定的延迟，当系统很繁忙时，延迟问题会更加严重，从库机器数量的增加也会使这个问题更严重。

此外，主库是集群的瓶颈，当写操作过多时会严重影响主库的稳定性，如果主库挂掉，则整个集群都将不能正常工作。

根据以上特点，我们总结一些最佳实践如下。

- 当读操作压力很大时，可以考虑添加从库机器来分解大量读操作带来的压力，但是当从库机器达到一定的数量时，就需要考虑分库来缓解压力了。
- 当写压力很大时，就必须进行分库操作了。

可能会因为种种原因，集群中的数据库硬件配置等会不一样，某些性能高，某些性能低，这时可以通过程序控制每台机器读写的比重来达到负载均衡，这需要更加复杂的读写分离的路由规则。

五、分库分表引起的问题

分库分表按照某种规则将数据的集合拆分成多个子集合，数据的完整性被打破，因此在某种场景下会产生多种问题。

1. 扩容与迁移

在分库分表后，如果涉及的分片已经达到了承载数据的最大值，就需要对集群进行扩容。扩容是很麻烦的，一般会成倍地扩容。

通用的扩容方法包括如下5个步骤：

Step1：按照新旧分片规则，对新旧数据库进行双写。

Step2：将双写前按照旧分片规则写入的历史数据，根据新分片规则迁移写入新的数据库。

Step3：将按照旧的分片规则查询改为按照新的分片规则查询。

Step4：将双写数据库逻辑从代码中下线，只按照新的分片规则写入数据。

Step5：删除按照旧分片规则写入的历史数据。

这里，在第2步迁移历史数据时，由于数据量很大，通常会导致不一致，因此，先清洗旧的数据，洗完后再迁移到新规则的新数据库下，再做全量对比，对比后评估在迁移的过程中是否有数据的更新，如果有的话就再清洗、迁移，最后以对比没有差距为准。

如果是金融交易数据，则最好将动静数据分离，随着时间的流逝，某个时间点之前的数据是不会被更新的，我们就可以拉长双写的时间窗口，这样在足够长的时间流逝后，只需迁移那些不再被更新的历史数据即可，就不会在迁移的过程中由于历史数据被更新而导致代理不一致。

在数据量巨大时，如果数据迁移后没法进行全量对比，就需要进行抽样对比，在进行抽样对比时要根据业务的特点选取一些具有某类特征性的数据进行对比。

在迁移的过程中，数据的更新会导致不一致，可以在线上记录迁移过程中的更新操作的日志，迁移后根据更新日志与历史数据共同决定数据的最新状态，来达到迁移数据的最终一致性。

2. 分库分表维度导致的查询问题

在分库分表以后，如果查询的标准是分片的主键，则可以通过分片规则再次路由并查询；但是对于其他主键的查询、范围查询、关联查询、查询结果排序等，并不是按照分库分表维度来查询的。

例如，用户购买了商品，需要将交易记录保存下来，那么如果按照买家的纬度分表，则每个买家的交易记录都被保存在同一表中，我们可以很快、很方便地查到某个买家的购买情况，但是某个商品被购买的交易数据很有可能分布在多张表中，查找起来比较麻烦。

反之，按照商品维度分表，则可以很方便地查找到该商品的购买情况，但若要查找到买家的交易记录，则会比较麻烦。

所以常见的解决方式如下：

- 在多个分片表查询后合并数据集，这种方式的效率很低。
- 记录两份数据，一份按照买家纬度分表，一份按照商品维度分表。
- 通过搜索引擎解决，但如果实时性要求很高，就需要实现实时搜索。

实际上，在高并发的服务平台下，交易系统是专门做交易的，因为交易是核心服务，SLA的级别比较高，所以需要和查询系统分离，查询一般通过其他系统进行，数据也可能是冗余存储的。

这里再举个例子，在某电商交易平台下，可能有买家查询自己在某一时间段的订单，也可能有卖家查询自己在某一时间段的订单，如果使用了分库分表方案，则这两个需求是难以满足的。

因此，通用的解决方案是，在交易生成时生成一份按照买家分片的数据副本和一份按照卖家分片的数据副本，查询时分别满足之前的两个需求，因此，查询的数据和交易的数据可能是分别存储的，并从不同的系统提供接口。

另外，在电商系统中，在一个交易订单生成后，一般需要引用到订单中交易的商品实体，如果简单地引用，若商品的金额等信息发生变化，则会导致原订单上的商品信息也会发生变化，这样买家会很疑惑。

因此，通用的解决方案是在交易系统中存储商品的快照，在查询交易时使用交易的快照，因为快照是个静态数据，永远都不会更新，所以解决了这个问题。

可见查询的问题最好在单独的系统中使用其他技术来解决，而不是在交易系统中实现各类查询功能；当然，也可以通过对商品的变更实施版本化，在交易订单中引用商品的版本信息，在版本更新时保留商品的旧版本，这也是一种不错的解决方案。

最后，关联的表有可能不在同一数据库中，所以基本不可能进行联合查询，需要借助大数据技术来实现，也就是上面所说的第3种方法，即通过大数据技术统一聚合和处理关系型数据库的数据，然后对外提供查询操作

3. 跨库事务难以实现

要避免在一个事务中同时修改数据库db0和数据库db1中的表，因为操作起来很复杂，对效率也会有一定的影响。请参考第三章的内容。

4. 同组数据跨库问题

要尽量把同一组数据放到同一台数据库服务器上，不但在某些场景下可以利用本地事务的强一致性，还可以使这组数据自

治。

以电商为例，我们的应用有两个数据库db0和db1，分库分表后，按照id维度，将卖家A的交易信息存放到db0中。当数据库db1挂掉时，卖家A的交易信息不受影响，依然可以正常使用。也就是说，要避免数据库中的数据依赖另一数据库中的数据。

分类： 大数据

« 上一篇： 根据自增ID生成不重复序列号
» 下一篇： 电商促销优惠规则业务分析建模

posted @ 2018-04-19 09:38 老皮肉 阅读(6726) 评论(0) 编辑 收藏