软件架构与中间件

涂志莹 tzy_hit@hit.edu.cn

苏统华 thsu@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第4章

数据层的软件架构技术

第4章 数据层的软件架构技术

4.1 数据驱动的软件架构演化

4.2 数据读写与主从分离

4.3 数据分库分表

4.4 数据缓存

4.5 非关系型数据库

4.6 数据层架构案例

4.3
数据分库分表

1、分库分表的基本概念

2、分库分表的解决方案

3、分库分表的架构设计

4、分库分表的中间件简介

4.3.3 分库分表的架构设计

- 切分方法
- 水平切分方式的路由过程和分片维度
- 分片后的事务处理机制
- 分库分表引起的问题

切分方法

垂直切分

- 垂直切分是指按照业务将表进行分类或分拆，将其分布到不同数据库上
  - 按业务进行分库
  - 按业务进行分表
- 不同业务模块的数据可以分散到不同数据库服务器
  - 例如，User数据、Pay数据、Commodity数据
- 也可以冷热分离，根据数据的活跃度将数据进行拆分。
  - ➢ 冷数据：变化更新频率低，查询次数多的数据。
  - ➢ 热数据：变化更新频率高，活跃的数据。
- 也可以人为将一个表中的内容划分为多个表，例如将查询较多，变化不多的字段拆分成一张表放在查询性能高的服务器，而将频繁更新的字段拆分并部署到更新性能高的服务器。

垂直切分例子

- 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，这些字段属于变化频率低的冷数据，而每篇微博的浏览数、回复数、点赞数等类似的统计信息属于变化频率高的热数据。
- 因此，一篇博客的数据可以按照冷热差异，拆分成两张表。冷数据存放的数据库可以使用MyISAM引擎，能更好地进行数据查询；热数据存放的数据库可以使用InnoDB存储引擎，更新性能好。
- 读多写少的冷数据库可以部署到缓存数据库上。

垂直切分的优缺点

- 优点：
  - 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、应用的等级或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
- 缺点：
  - 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

水平切分

- 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，所有表加起来是全量数据。
- 简言之，将数据按一定规律，按行切分，并分配到不同的库表里，表结构完全一样。
- 例如：在博客系统中，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10万次请求。当然还可以分配到不同服务器的服务实例中，分的表越多，每个单表的压力越小。

水平切分的优缺点

- 优点：
  - 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
- 缺点：
  - 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

垂直、水平切分的共同点

- 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

- 垂直切分更偏向于业务拆分的过程
- 水平切分更偏向于技术性能指标

水平切分方式的路由过程和分片维度

水平切分的路由过程

- 分库分表后，数据将分布到不同的分片表中，通过分库分表规则查找到对应的表和库的过程叫做路由。
- 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，当生成新用户时，程序得确定将此用户的信息添加到哪个表中。
- 同样，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。

水平切分的分片维度

- 按哈希切片
  - 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，这样将数据分成多个分片
  - 好处：数据切片比较均匀，对数据压力分散的效果较好
  - 缺点：数据分散后，对于查询需求需要进行聚合处理
- 按照时间切片
  - 按照时间的范围将数据分布到不同的分片

分片后的事务处理机制

1、分布式事务

2、事务路由

CAP理论

- CAP理论：一个分布式系统不能同时满足“一致性(C)、可用性(A)和分区容错性(P)”需求，最多只能同时满足两个。
  - C: Consistency(一致性)：任何一个读操作总是能读取到之前完成的写操作结果，也就是在分布式环境中，多点的数据是一致的；
  - A: Availability(可用性)：每一个操作总是能够在确定的时间内返回，也就是系统随时都是可用的；
  - P: Tolerance of Network Partition(分区容忍性)：在出现网络分区的情况下，分离的系统也能正常运行；
- 对于分布式存储系统而言，分区容错性(P)是基本需求，只有CP和AP两种模式的选择。
  - CP模式：保证分布在网络上不同节点数据一致性，但对可用性支持不足。
  - AP模式：以实现“最终一致性(Eventual Consistency)”来确保可用性和分区容忍性，但弱化了数据一致性要求。

CAP理论

- CAP关注的力度是数据，而不是整个系统
  - 每个系统都会处理不同类型的数据，有的数据可以遵守CP，有的可以遵守AP。
- CAP是忽略网络延迟的
  - 理论中的C并不可能完美的实现，因为网络延迟。
- 正常运行情况下，不存在CP和AP的选择，可以同时满足CA
  - 如果不分区的情况下
- 放弃并不等于什么都不做，需要为分区恢复后做准备

两阶段提交协议

二阶段提交的算法思路为: 参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

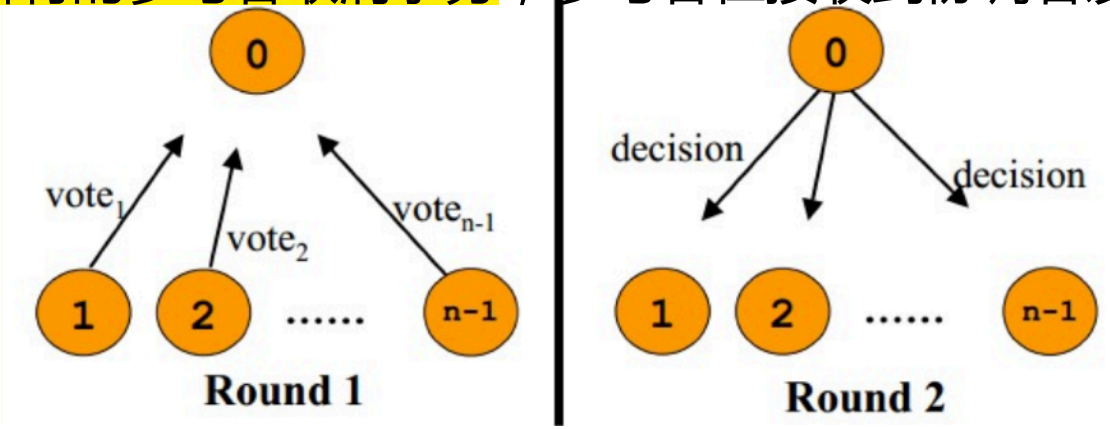
(1) 请求阶段(表决)：

事务协调者通知每个参与者准备提交或取消事务，然后进入表决过程，参与者要么在本地执行事务，写本地的redo和undo日志，但不提交。请求阶段，参与者将告知协调者自己的决策: 同意(事务参与者本地作业执行成功)或取消（本地作业执行故障）

(2) 提交阶段(执行):

在该阶段，写调整将基于第一个阶段的投票结果进行决策: 提交或取消。

当且仅当所有的参与者同意提交事务，协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务，参与者在接收到协调者发来的消息后将执行响应的操作。



两阶段提交协议缺点

- 1.同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
- 2.单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
- 3.数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这会导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。

三阶段提交协议

针对“单点故障”问题，在第一、二阶段间加入“准备阶段”，当协调者故障后，参与者可以通过超时提交来避免一致阻塞。

(1) canCommit阶段

3PC的canCommit阶段其实和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回yes响应，否则返回no响应

(2) preCommit阶段

协调者根据参与者canCommit阶段的响应来决定是否可以继续事务的preCommit操作

a) 协调者从所有参与者得到的反馈都是yes:
那么进行事务的预执行，协调者向所有参与者发送preCommit请求，并进入prepared阶段。参与者接收到preCommit请求后会执行事务操作，并将undo和redo信息记录到事务日志中。如果一个参与者成功地执行了事务操作，则返回ACK响应，同时开始等待最终指令

b) 协调者从所有参与者得到的反馈有一个是No或是等待超时之后协调者都没收到响应:
那么就要中断事务，协调者向所有的参与者发送abort请求。参与者在收到来自协调者的abort请求，或超时后仍未收到协调者请求，执行事务中断。

三阶段提交协议

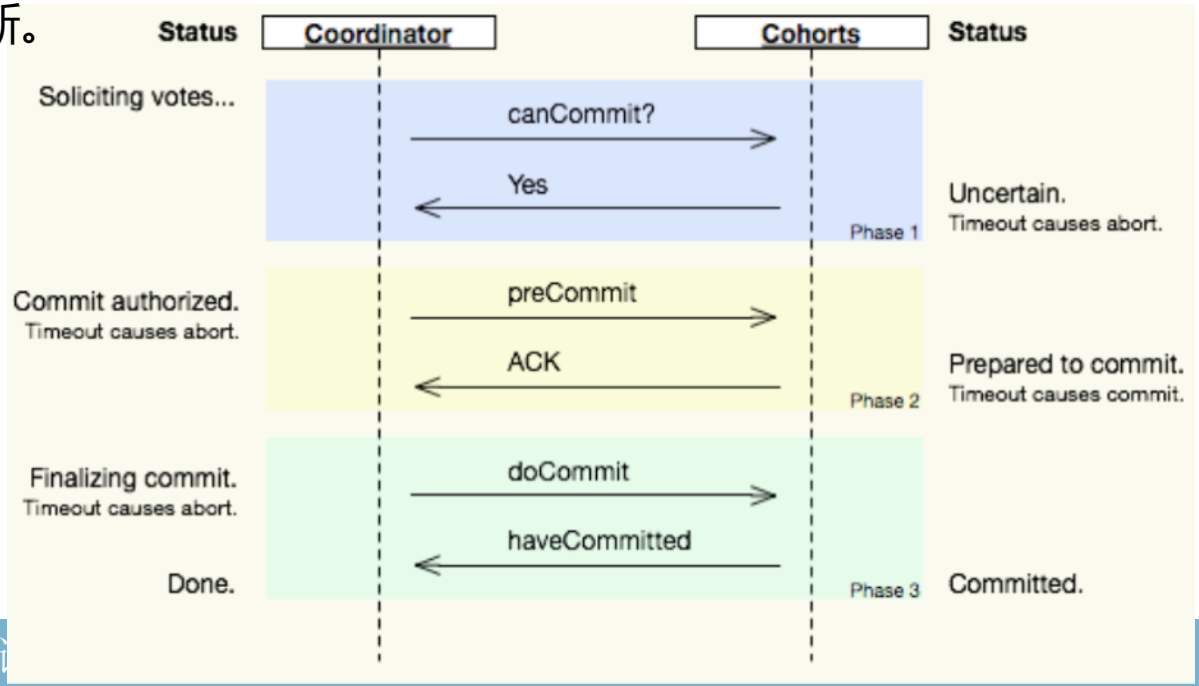
(3) doCommit阶段

协调者根据参与者preCommit阶段的响应来决定是否可以继续事务的doCommit操作

a) 协调者从参与者得到了ACK的反馈:

协调者接收到参与者发送的ACK响应，那么它将从预提交状态进入到提交状态，并向所有参与者发送doCommit请求。参与者接收到doCommit请求后，执行正式的事务提交，并在完成事务提交之后释放所有事务资源，并向协调者发送haveCommitted的ACK响应。那么协调者收到这个ACK响应之后，完成任务。

b) 协调者从参与者没有得到ACK的反馈, 也可能是接收者发送的不是ACK响应，也可能是响应超时：执行事务中断。



最大努力保证模式

- 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。
- 具体实现方法：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，如果业务流程出现问题，则所有资源更新都回滚，保持事务一致。
- 以消息队列消息消费和更新数据库为例：
  - 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

但需注意事务的嵌套

事务补偿机制

- 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，使其在一定时间窗口内完成操作
- 这样既保证了事务的最终一致性，又突破了事务遇到问题就回滚的传统思想。
- 如果采用事务补偿机制，则在遇到问题时，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。

分片后的事务处理机制

1、分布式事务

2、事务路由

事务路由

- 无论使用哪种分布式事务处理方法，都需要对分库分表的多个数据源路由事务。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。
- 这就需要通过编写程序来选择数据库的事务管理器，即事务路由。
  - 自动提交事务路由
  - 可编程事务路由
  - 声明式事务路由

自动提交事务路由

- 自动提交事务路由通过依赖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，不需要开发人员手动操作事务，也不需要配置事务，但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。

```
Connection conn = null;
try {
    conn=getConnection();
    conn.setAutoCommit(true);
    ……
    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();
        }
}
```

可编程事务路由

- 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。

可编程事务路由

```
// 调用dao的方法
// 业务逻辑，写转账业务
public void accountMoney() {
    transactionTemplate.execute(new TransactionCallback<Object>() {

        @Override
        public Object doInTransaction(TransactionStatus status) {
            Object result = null;
            try {
                // 小马多1000
                ordersDao.addMoney();
                // 加入出现异常模拟银行突然停电等，结果：小马账户多了1000而小王账户没有少钱
                // 解决办法是出现异常后进行事务回滚
                int i = 10 / 0;// 事务管理配置后异常已经解决
                // 小王 少1000
                ordersDao.reduceMoney();
            } catch (Exception e) {
                status.setRollbackOnly();
                result = false;
                System.out.println("Transfer Error!");
            }

            return result;
        }
    }
```

事务处理机制 → 事务路由

声明式事务路由

```
<aop:config>
        <!-- 切入点 -->
        <aop:pointcut expression="execution(* cn.itcast.service.OrdersService.*(..))"
                id="pointcut1" />
        <!-- 切面 -->
        <aop:advisor advice-ref="txadvice" pointcut-ref="pointcut1" />
</aop:config>



<!-- 对象生成及属性注入 -->
<bean id="ordersService" class="cn.itcast.service.OrdersService">
        <property name="ordersDao" ref="ordersDao"></property>
</bean>
<bean id="ordersDao" class="cn.itcast.dao.OrdersDao">
        <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
</bean>
```

注解方式的声明事务路由

- 即在实现的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。

```
<!-- 第二步： 开启事务注解 -->
<tx:annotation-driven transaction-manager="dataSourceTransactionManager" />
<!-- 第三步 在方法所在类上加注解 -->


<!-- 对象生成及属性注入 -->
<bean id="ordersService" class="cn.itcast.service.OrdersService">
        <property name="ordersDao" ref="ordersDao"></property>
</bean>
<bean id="ordersDao" class="cn.itcast.dao.OrdersDao">
        <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
</bean>
```

分库分表引起的问题

扩容与迁移

- 通用的处理方法：
  - 1、按照新旧分片规则，对新旧数据库进行双写
  - 2、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
  - 3、将按照旧的分片规则查询改为按照新的分片规则查询
  - 4、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
  - 5、删除按照旧分片规则写入的历史数据

扩容与迁移的问题

- 数据一致性问题：
  - 由于数据量大，通常会造成不一致问题，因此，通常先清理旧数据，洗完后再迁移到新规则的新数据库下，再做全量对比。还需要对比评估迁移过程中是否有数据更新，如果有需要迭代清洗，直至一致。
  - 如果数据量巨大，无法全量对比，需要抽样对比，抽样特征需要根据业务特点进行选取。
  - 注意：线上记录迁移过程中的更新操作日志，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，以达到迁移数据的最终一致性。
- 动静数据分离问题：
  - 对于一些动静敏感的数据，如交易数据，最好将动静数据分离。选取时间点对静历史数据进行迁移。

查询问题

- 查询问题解释：
  - 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。
- 查询问题的解决方案：
  - 1、在多个分片表查询后合并数据集（效率很低）
  - 2、按查询需求定义多分片维度，形成多张分片表（空间换时间）
  - 3、通过搜索引擎解决，如果有实时要求，还需要实时搜索。（难度大）

分布式事务问题

- 多库多表分布式所引发的一致性问题。

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，还可以使这组数据实现自治。

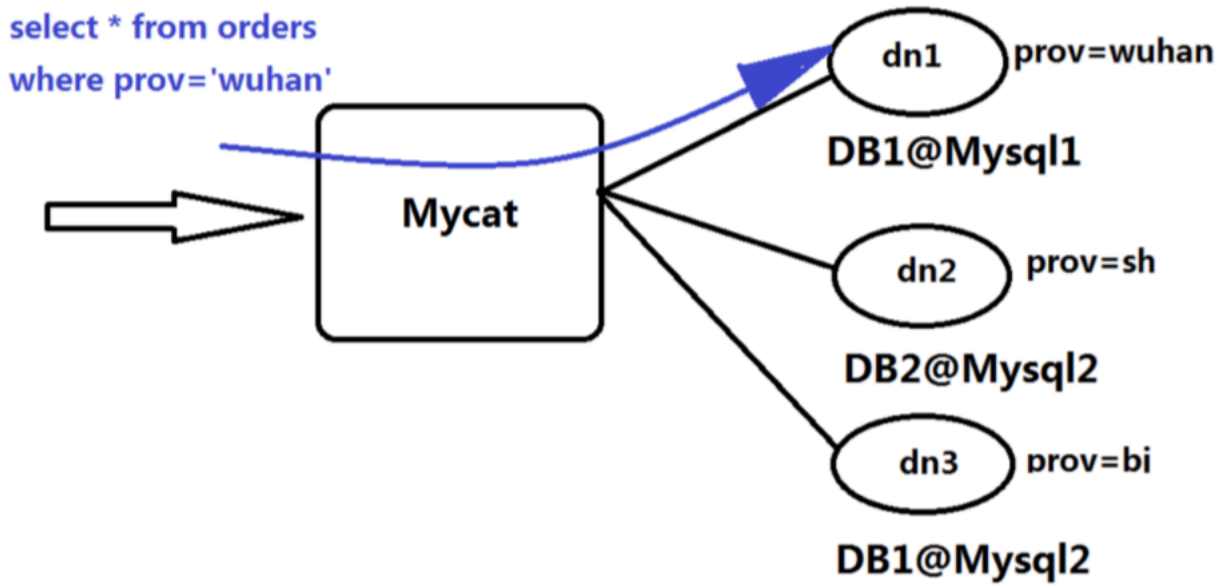
4.3.4 分库分表的中间件简介

- Mycat
- Sharding JDBC

Mycat

- Mycat是一个强大的数据库中间件，不仅仅可以用作读写分离、以及分表分库、容灾备份，而且可以用于多租户应用开发、云平台基础设施。
- Mycat后面连接的Mycat Server，就好象是MySQL的存储引擎，如InnoDB，MyISAM 等，因此，Mycat 本身并不存储数据，数据是在后端的MySQL上存储的，因此数据可靠性以及事务等都是MySQL保证的。

Mycat基本原理



- Mycat拦截了用户发送过来的SQL语句，首先对SQL语句做一些特定的分析：如分片分析、路由分析、读写分离分析、缓存分析等，然后将此 SQL 发往后端的真实数据库， 并将返回的结果做适当的处理，最终再返回给用户。
- 当 Mycat收到一个SQL时，会先解析这个SQL，查找涉及到的表，然后看此表的定义，如果有分片规则， 则获取到SQL里分片字段的值，并匹配分片函数，得到该SQL对应的分片列表，然后将SQL发往这些分片去执行，最后收集和处理所有分片返回的结果数据，并输出到客户端。

Mycat应用场景

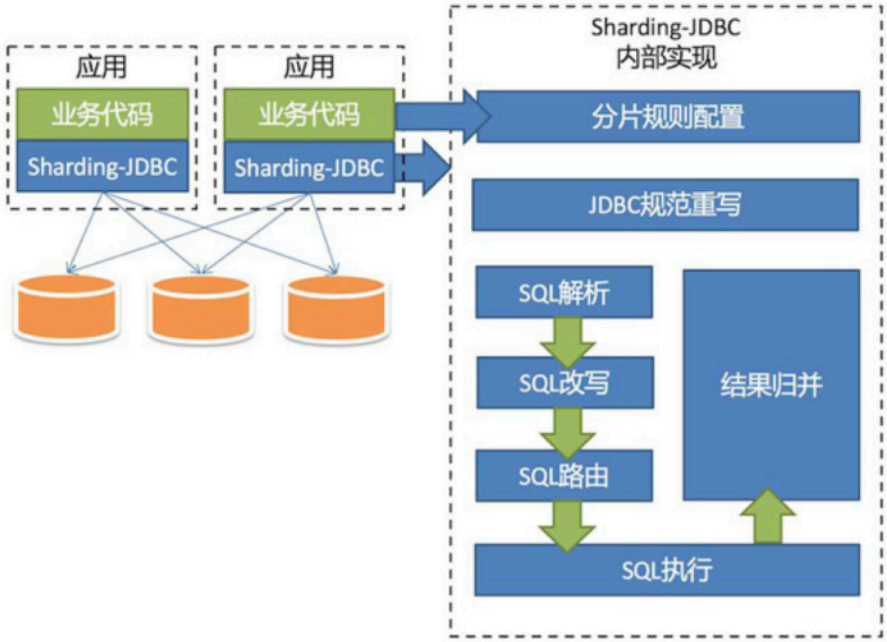
- 单纯的读写分离，此时配置最为简单，支持读写分离，主从切换
- 分表分库，对于超过1000万的表进行分片，最大支持1000亿的单表分片
- 多租户应用，每个应用一个库，但应用程序只连接Mycat，从而不改造程序本身，实现多租户化
- 报表系统，借助于Mycat 的分表能力，处理大规模报表的统计
- 替代Hbase，分析大数据
- 作为海量数据实时查询的一种简单有效方案，比如100亿条频繁查询的记录需要在 3 秒内查询出来结果，除了基于主键的查询，还可能存在范围查询或其他属性查询，此时 Mycat 可能是最简单有效的选择

Sharding JDBC

- Sharding-JDBC是当当应用框架ddframe中，从关系型数据库模块dd-rdb中分离出来的数据库水平分片框架，实现透明化数据库分库分表访问。Sharding-JDBC直接封装JDBC API，可以理解为增强版的JDBC驱动，旧代码迁移成本几乎为零：
  - 可适用于任何基于Java的ORM框架，如JPA、Hibernate、Mybatis、Spring JDBC Template或直接使用JDBC。
  - 可基于任何第三方的数据库连接池，如DBCP、C3P0、BoneCP、Druid等。
  - 理论上可支持任意实现JDBC规范的数据库。虽然目前仅支持MySQL，但已有支持Oracle、SQLServer等数据库的计划。
- Sharding-JDBC定位为轻量Java框架，使用客户端直连数据库，以jar包形式提供服务，无proxy代理层，无需额外部署，无其他依赖，DBA也无需改变原有的运维方式。
- Sharding-JDBC分片策略灵活，可支持等号、between、in等多维度分片，也可支持多分片键。
- SQL解析功能完善，支持聚合、分组、排序、limit、or等查询，并支持Binding Table以及笛卡尔积表查询。

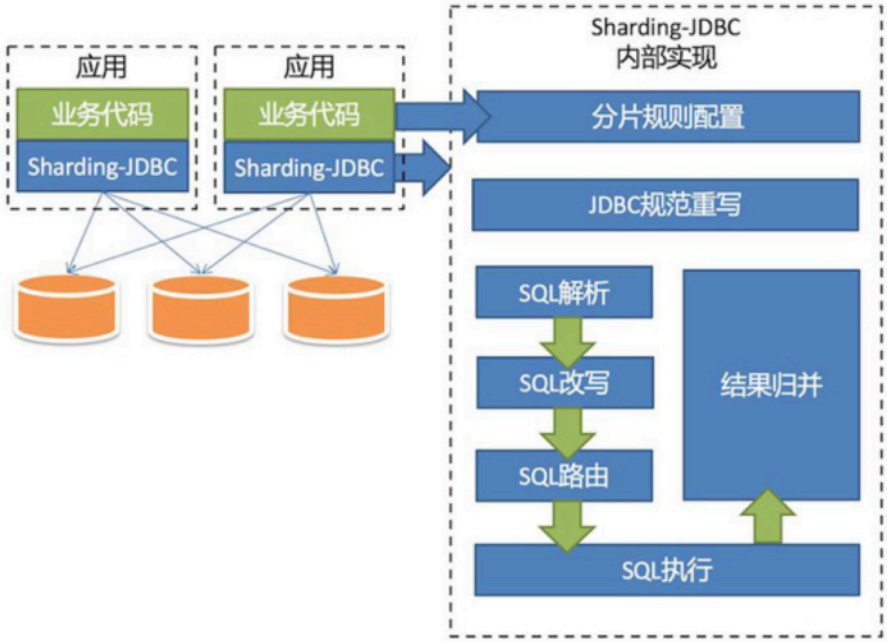
Sharding JDBC原理

- 分片规则配置
  - Sharding-JDBC的分片逻辑非常灵活，支持分片策略自定义、复数分片键、多运算符分片等功能。
    - 如：根据用户ID分库，根据订单ID分表这种分库分表结合的分片策略；或根据年分库，月份+用户区域ID分表这样的多片键分片。
  - Sharding-JDBC除了支持等号运算符进行分片，还支持in/between运算符分片，提供了更加强大的分片功能。
  - Sharding-JDBC提供了spring命名空间用于简化配置，以及规则引擎用于简化策略编写。由于目前刚开源分片核心逻辑，这两个模块暂未开源，待核心稳定后将会开源其他模块。



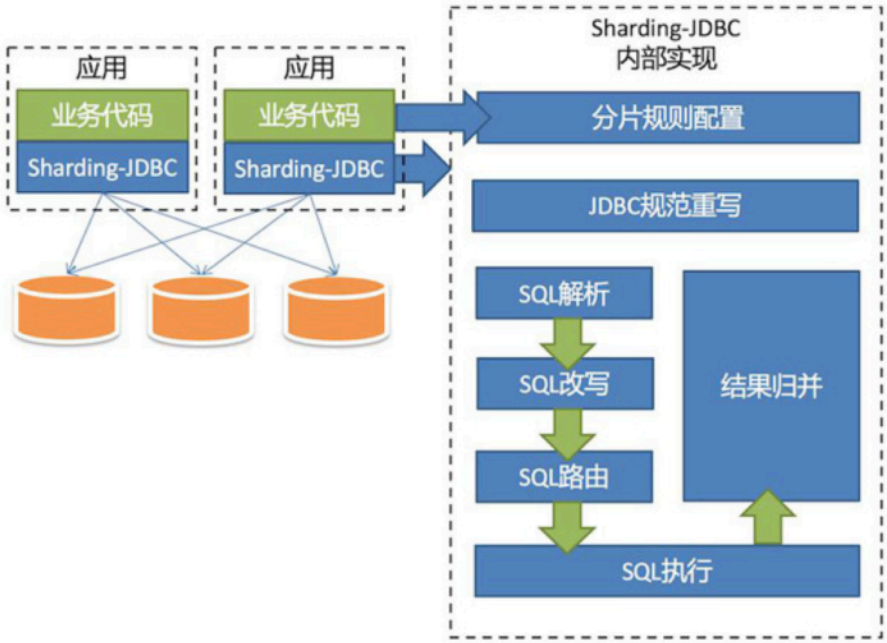
Sharding JDBC原理

- JDBC规范重写
  - Sharding-JDBC对JDBC规范的重写思路是针对DataSource、Connection、Statement、PreparedStatement和ResultSet五个核心接口封装，将多个真实JDBC实现类集合（如：MySQL JDBC实现/DBCP JDBC实现等）纳入Sharding-JDBC实现类管理。
  - Sharding-JDBC尽量最大化实现JDBC协议，包括addBatch这种在JPA中会使用的批量更新功能。
  - 但分片JDBC毕竟与原生JDBC不同，所以目前仍有未实现的接口，包括Connection游标，存储过程和savePoint相关、ResultSet向前遍历和修改等不太常用的功能。此外，为了保证兼容性，并未实现JDBC 4.1及其后发布的接口（如：DBCP 1.x版本不支持JDBC 4.1）。



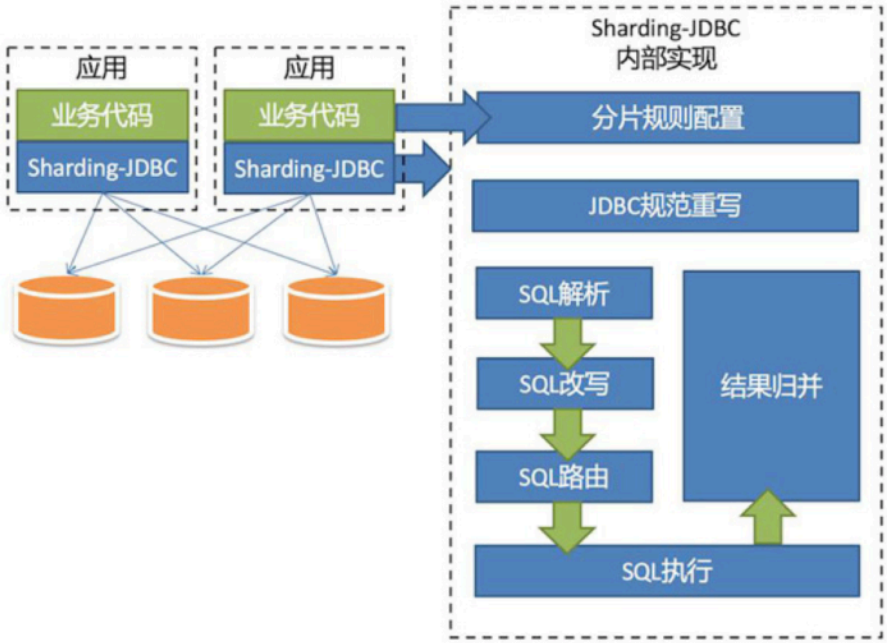
Sharding JDBC原理

- SQL解析
  - SQL解析作为分库分表类产品的核心，性能和兼容性是最重要的衡量指标。目前常见的SQL解析器主要有fdb/jsqlparser和Druid。Sharding-JDBC使用Druid作为SQL解析器，经实际测试，Druid解析速度是另外两个解析器的几十倍。
  - 目前Sharding-JDBC支持join、aggregation（包括avg）、order by、 group by、limit、甚至or查询等复杂SQL的解析。目前不支持union、部分子查询、函数内分片等不太应在分片场景中出现的SQL解析。



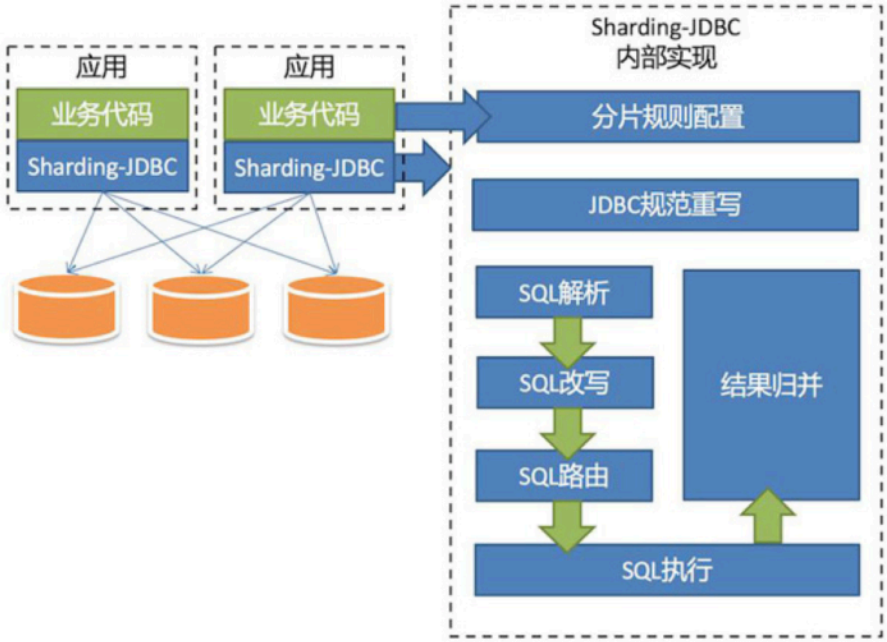
Sharding JDBC原理

- SQL改写
  - SQL改写分为两部分，一部分是将分表的逻辑表名称替换为真实表名称。另一部分是根据SQL解析结果替换一些在分片环境中不正确的功能。这里具两个例子：
  - 如avg计算。在分片的环境中，以avg1 +avg2+avg3/3计算平均值并不正确，需要改写为（sum1+sum2+sum3）/（count1+count2+ count3）。这就需要将包含avg的SQL改写为sum和count，然后再结果归并时重新计算平均值。



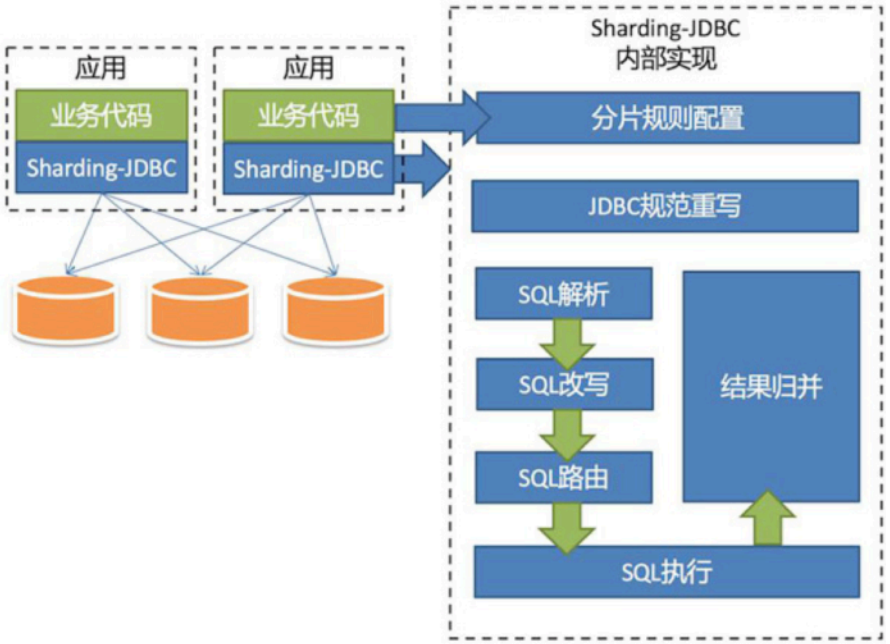
Sharding JDBC原理

- SQL路由
  - SQL路由是根据分片规则配置，将SQL定位至真正的数据源。主要分为单表路由、Binding表路由和笛卡尔积路由。
  - 单表路由最为简单，但路由结果不一定落入唯一库（表），因为支持根据between和in这样的操作符进行分片，所以最终结果仍然可能落入多个库（表）。
  - Binding表可理解为分库分表规则完全一致的主从表。
  - 笛卡尔积查询最为复杂，因为无法根据Binding关系定位分片规则的一致性，所以非Binding表的关联查询需要拆解为笛卡尔积组合执行。查询性能较低，而且数据库连接数较高，需谨慎使用。



Sharding JDBC原理

- SQL执行
  - 路由至真实数据源后，Sharding-JDBC将采用多线程并发执行SQL，并完成对addBatch等批量方法的处理。



Sharding JDBC原理

- 结果归并
  - 结果归并包括4类：普通遍历类、排序类、聚合类和分组类。每种类型都会先根据分页结果跳过不需要的数据。
  - 普通遍历类最为简单，只需按顺序遍历ResultSet的集合即可。
  - 排序类结果将结果先排序再输出，因为各分片结果均按照各自条件完成排序，所以采用归并排序算法整合最终结果。
  - 聚合类分为3种类型，比较型、累加型和平均值型。比较型包括max和min，只返回最大（小）结果。累加型包括sum和count，需要将结果累加后返回。平均值则是通过SQL改写的sum和count计算，相关内容已在SQL改写涵盖，不再赘述。
  - 分组类最为复杂，需要将所有的ResultSet结果放入内存，使用map-reduce算法分组，最后根据排序和聚合条件做相关处理。最消耗内存，最损失性能的部分即是此，可以考虑使用limit合理的限制分组数据大小。
- 结果归并部分目前并未采用管道解析的方式，之后会针对这里做更多改进。

第4章

数据层的软件架构技术

Thanks for listening

涂志莹、苏统华
哈尔滨工业大学计算机学院
企业与服务计算研究中心