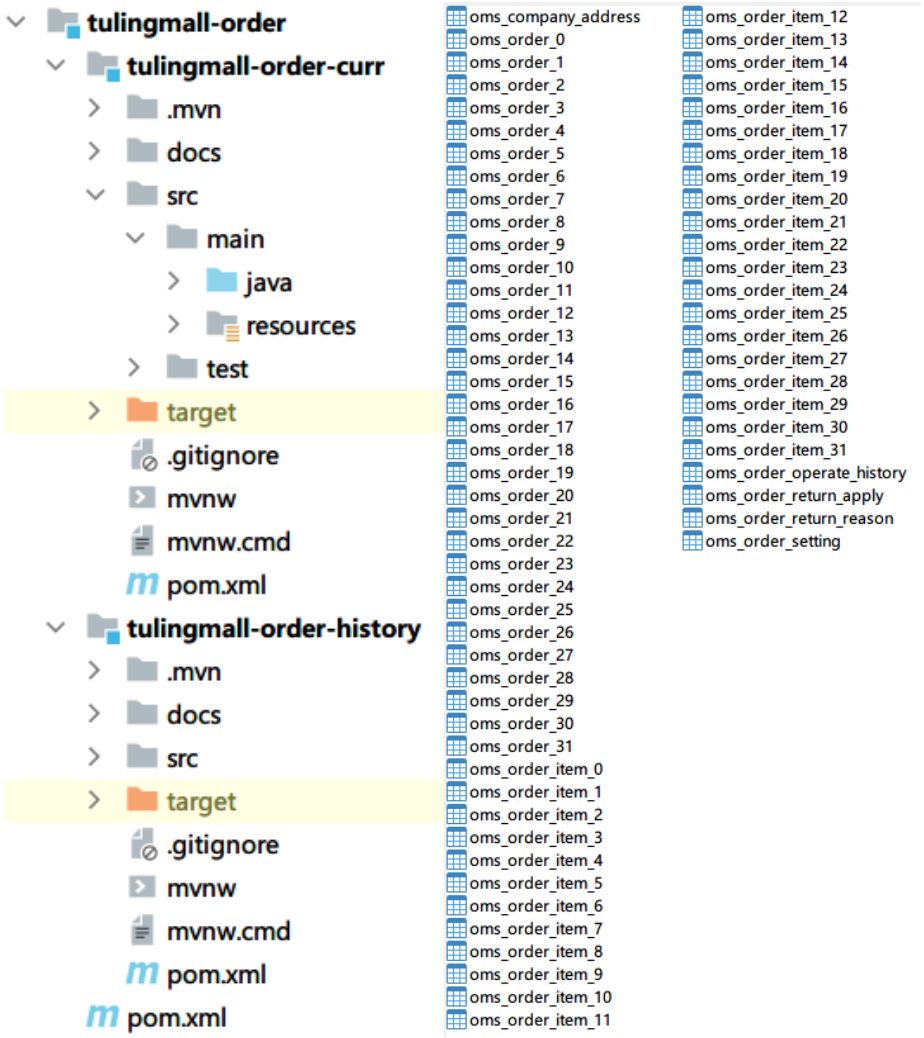
订单系统的设计与海量数据处理实战

订单系统可以说是整个电商系统中最重要的一个子系统，因此订单数据可以算作电商企业最重要的数据资产。这节课我们来看看在我们的商城系统中订单服务是如何实现的，特别是在设计和实现一个订单系统的过程中有哪些问题是需要特别考虑的。首先概略了解商城系统中的订单系统：



可以看到，订单系统从代码上来说，分为两个部分：tulingmall-order-curr 订单程序和 tulingmall-order-history 历史订单处理程序，数据存储也进行了分库分表。更具体实现我们接下来分析。

订单系统业务分析

对于一个合格的订单系统,最基本的要求是什么？数据不能出错。用户的每一次购物，从下单开始到支付、发货,再到收货，流程中的每个环节，都需要同步更新订单数据，每次更新操作可能都需要同时更新好几张表。这些操作可能会随机分发到不同的服务器节点上执行，服务器或网络都有可能会出问题，在这么复杂的情况下，如何保证订单数据不出错呢？

第一，代码必须是正确的没有 Bug，当然这个要求很简单也很复杂，全是 bug 系统无法正常运行，但是也没有什么系统能保证没有一个 bug。当然要确保不能因为代码 Bug 而导致数据错误。

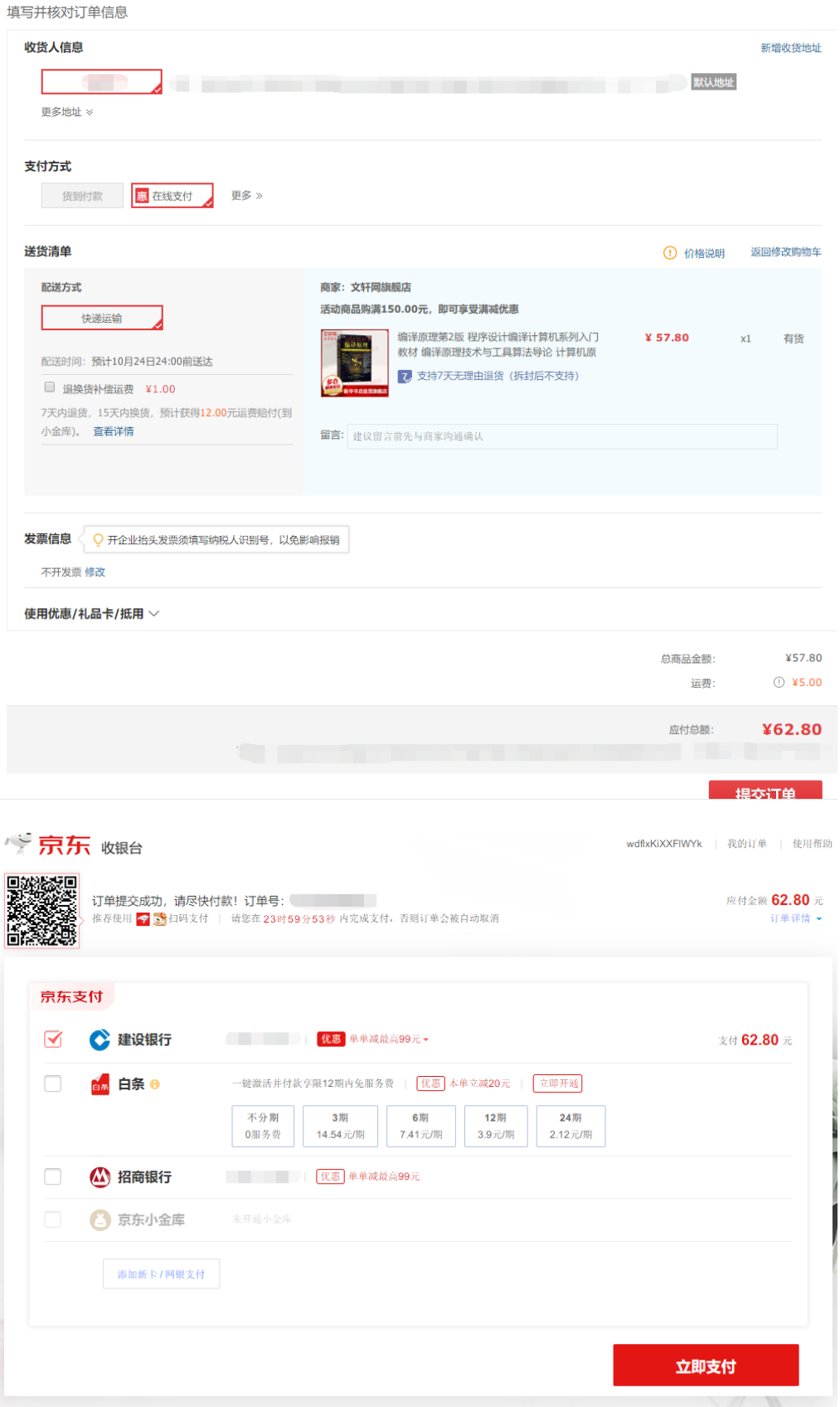
第二，要能够正确地使用事务。比如，在创建订单的时候,如果需要同时在订单表和订单商品表中插入数据，那么我们必须在一个数据库事务中执行这些插入数据的 INSERT 语句，数据库事务可以确保:执行需要同时进行的操作语句时,要么一起成功,要么一起失败。而实际上，在微服务下，仅仅使用数据库事务是不够的，很多时候还需要分布式事务。后面课程会专门的讲解分布式事务在项目中的实现。

即使满足了上面列举的这两个基本要求，某些特殊情况也仍然可能会引发数据错误，是什么样的数据错误问题？如何解决呢？都会在本章中讲到。

在此之前,我们需要首先了解对于一个订单系统而言，它的核心功能和数据结构是怎样的。其实任何一个公司的电商系统，其订单系统的功能都是独一无二的，因为订单系统会基于其业务配置了很多的功能，并且都很复杂。因此我们的电商系统只能化繁为简，聚焦那些最核心的、共通的业务功能和数据模型，并且以此为基础讨论其中的实现技术。

订单系统的核心功能和数据表

本节首先简单梳理一下订单系统所必备的功能，其包含但远远不限于如下功能。



创建订单，从上面京东的流程可以看到，用户从购物车选择了商品后去结算而创建订单其实包括两个业务步骤，一是订单确认，二是订单提交。订单提交后则进入支付流程。

随着购物流程推进更新订单状态。

查询订单。

为了支撑这些必备功能，一般订单数据库中至少需要具备如下 4 张表。

订单主表：也称订单表，用于保存订单的基本信息，也就是我们的 oms_order 表。

订单商品表：用于保存订单中的商品信息，也就是我们的 oms_order_item 表。

订单支付表：用于保存订单的支付和退款信息。

订单优惠表：用于保存订单使用的所有优惠信息。

这 4 张表之间的关系是订单主表与后面的几个子表都是一对多的关系，关联的外键就是订单主表的主键，即订单 ID。

在我们的商城系统中，做了适度的改造和简化，表现在：

因为取消了一般的促销优惠，自然没有专门的订单优惠表；

订单的是否支付和是否退款直接保存在订单主表中，没有设计单独订单支付表用以保存支付和退款相关信息；

我们系统中没有单独的库存系统，所以库存扣减由订单系统发起，由产品服务执行实际的扣减库存操作。

对应到存储上，则是 oms_order 作为订单主表，其中的 status 字段表示了订单状态，包括“0->待付款；1->待发货；2->已发货；3->已完成；4->已关闭；5->无效订单”。

产品的库存则是保存在产品服务 tulingmall-product 对应数据库 tl_mall_goods 的 pms_sku_stock 表中。

从上面我们对订单的流程描述来看，订单系统的实现其实并不复杂，就是标准的 CRUD。

但是前面我们也说过，某些情况也仍然可能会引发数据错误，是哪些情况呢？我们来一一分析下。

订单重复下单问题

仔细分析一下订单创建的场景：订单系统为用户提供创建订单的 HTTP 接口，用户在浏览器页面上点击“提交订单”按钮，浏览器向订单系统发送一条创建订单的请求，订单系统的后端服务收到请求，向数据库的订单表中插入一条订单数据，至此，订单创建成功。

那么我们设想一下，用户在点击“提交订单”的按钮时，不小心点了两下，那么浏览器就会向服务端连续发送两条创建订单的请求,最终的结果将会是什么？很自然会创建两条一模一样的订单。这样肯定是不行的，因此我们还需要做好防重工作，怎么做呢？

可能有人会想到，前端页面上应该防止用户重复提交表单，当用户“提交订单”的按钮后，将该按钮置灰不可用。但是仔细想想，即使前端控制了用户不重复提交，网络错误也有可能会导致重传，很多 RPC 框架和网关都拥有自动重试机制，所以对于订单服务来说，重复请求的问题是客观存在的。

解决办法是,让订单服务具备幂等性。什么是幂等性？幂等操作的特点是，操作任意多次执行所产生的影响，均与一次执行所产生的影响相同。也就是说，对于幂等方法，使用同样的参数，对它进行多次调用和一次调用，其对系统产生的影响是一样的。

例如：

```
update tableA set count = 10 where id = 1
```

这个操作多次执行，id 等于 1 的记录中的 count 字段的值都为 10，这个操作就是幂等的，我们不用担心这个操作被重复。

```
update tableA set count = count + 1 where id = 1;
```

这样的 SQL 操作就不是幂等的，一旦重复，结果就会产生变化。

所以，不用担心幂等方法的重复执行会对系统造成任何改变。如果创建订单的服务具备幂等性，那么无论创建订单的请求发送了多少次，正确的结果都是数据库只有一条新创建的订单记录。

这里又会涉及一个不太好解决的问题：对于订单服务来说，如何判断收到的创建订单的请求是不是重复请求呢？

在插入订单数据之前，先查询一下订单表里面有没有重复的订单，是不是就可以做出判断了呢？这个方法看起来容易。实际上却很难实现。原因是我们很难通过 SQL 的 WHERE 语句来定义“重复的订单”，如果订单的用户、商品、数量和价格都一样,是否就能认为它们是重复订单呢？这个其实是无法确定的，因为有可能用户就是连续下了两个一模一样的订单。

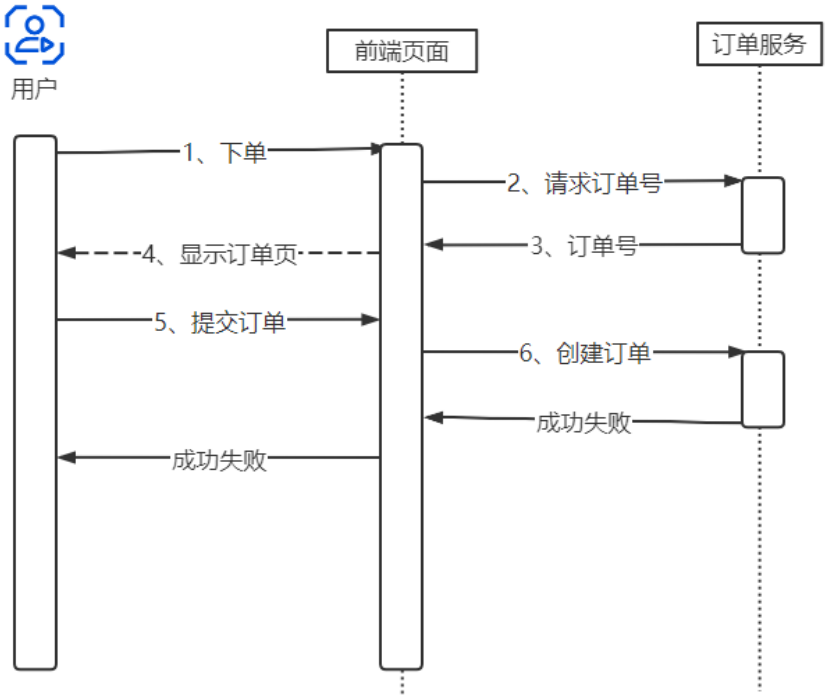
这个问题的思路是利用数据库的唯一约束来判断数据是否重复。在数据库的最佳实践中，其中一条是要求数据库的每个表都有主键。在非分库分表的情况下，我们在向数据库的表中插入一条记录的时候，无需提供主键，插入的同时由数据库自动生成一个主键。这样，重复的请求就会导致插入重复的数据。

表的主键是自带唯一约束的,如果我们在一条 INSERT 语句中提供了主键,并且这个主键的值已经存在于表中,那么这条 INSERT 语句就会执行失败,数据也不会成功插入表中。我们可以利用数据库的这种“主键唯一约束”特性，在插入数据的时候带上主键,来解决创建订单服务的幂等性问题。

具体做法如下：首先，为订单系统增加一个“生成订单号”的服务,这个服务没有参数,返回值就是一个新的、全局唯一的订单号。在用户进入创建订单的页面时，前端页面会先调用这个生成订单号的服务得到一个订单号，在用户提交订单的时候，在创建订单的请求中带着这个订单号。

这个订单号就是订单表的主键，这样，无论是用户原因,还是网络原因等各种情况导致的重试,这些重复请求中的订单号都是相同的。订单服务在订单表中插入数据的时候，这些重复的 INSERT 语句中的主键，都是同一个订单号。数据库的主键唯一约束特性就可以保证，只有一次 INSERT 语句的执行是成功的,这样就实现了创建订单服务的幂等性。

时序图如下：



所以，可以看到，在 OmsPortalOrderController 中专门提供了 generateOrderId 方法供外部系统获得订单 ID。

```
@ApiOperation("获取orderId，可避免重复下单")
@GetMapping(value = "/generateOrderId")
@ResponseBody
public CommonResult generateOrderId(@RequestHeader("memberId") Long memberId)
    Long orderId = portalOrderService.generateOrderId(memberId);
    return CommonResult.success(orderId);
}
```

而秒杀系统相关的微服务中虽然没有提供类似的 generateOrderId 方法，但依然注意了避免重复下单问题，在生成订单确认信息时，将预先生成的订单 ID 传递给了前端订单确认页。

```
/*秒杀订单确认信息*/
1 usage   Mark
@Override
public CommonResult generateConfirmMiaoShaOrder(Long productId
        , Long memberId, String token,Long flashPromotionId) throws

    String orderIdStr = orderIdList.poll();
    String orderItemIdStr = orderItemIdList.poll();
```

```
ConfirmOrderResult result = new ConfirmOrderResult();
result.setOrderId(Long.valueOf(orderIdStr));
```

还有一点需要注意的是，在具体实现时，如果是因为重复订单导致插入订单表的语句失败，那么订单服务就不要再把这个错误返回给前端页面了。否则，就有可能会出现用户点击创建订单按钮后，页面提示创建订单失败，而实际上订单已经创建成功了。正确的做法是,遇到这种情况,订单服务直接返回“订单创建成功”的响应即可。

要做到这一点，可以捕获 java.sql.SQLIntegrityConstraintViolationException 或者 org.springframework.dao.DuplicateKeyException 来实现。

订单 ABA 问题和解决

订单系统中，各种更新订单的服务同样也需要具备幂等性。

更新订单的服务，比如支付、发货等这些步骤中的更新订单操作，最终都会落到订单库上，都是对订单主表进行更新操作。

比如对支付操作的数据库的更新操作、无论是执行一次还是重复执行多次,订单状态都是已支付，不用我们额外设置任何逻辑，这就是天然幂等性。

在实现这些更新订单的服务时，还有哪些问题需要特别注意呢？在并发环境下，我们需要特别注意 ABA 问题。

什么是更新下的 ABA 问题呢？我们知道并发编程下的 CAS 有 ABA 问题，这个 ABA 问题和并发的 ABA 问题有相似之处。我们来看这么一个例子：

订单支付完成，填入物流单号 666 提交后，发现填错了，修改成正确的单号 888，对于订单服务来说，这里就产生了两个更新订单的请求。

按照我们的设想，正常情况下，订单中的快递单号会先更新成 666，再更新成 888，这是没有问题的。但是现实生活有很多不正常的情况，比如，更新成 666 的请求到了，快递单号更新成 666，然后更新成 888 的请求到了，快递单号又更新成 888。但是订单服务在向调用方返回 666 更新成功的响应时，这个响应在网络传输过程中丢失了。如果调用方没有收到成功响应，触发自动重试逻辑，再次发起更新成 666 的请求，快递单号将会再次更新成 666,这种情况下数据显然就会出错了。这就是 ABA 问题。

那么 ABA 问题应该怎么解决呢？仔细想想并发编程里怎么解决 ABA 问题的？版本戳。所以这里同样可以使用版本戳。

为订单主表增加一列，列名可以叫 version、也就是“版本号”的意思。每次查询订单的时候，版本号需要随着订单数据返回给页面。页面在更新数据的请求时,需要把该版本号作为更新请求的参数再带回给订单更新服务。

订单服务在更新数据的时候需要比较订单当前数据的版本号与消息中的版本号是否一致，如果不一致就拒绝更新数据。如果版本号一致，则还需要在更新数据的同时，把版本号加 1。当然需要特别注意的是，“比较版本号、更新数据和把版本号加 1”这个过程必须在同一个事务里面执行，只有这一系列操作具备原子性，才能真正保证并发操作的安全性。

具体的 SQL 语句参考如下:

UPDATE orders set tracking_number = 666,version = version + 1 WHERE version = ?;

版本号的机制可用于保证，从打开某条订单记录开始，一直到这条订单记录更新成功，这期间不会存在有其他人修改过这条订单数据的情况。因为如果被其他人修改过，数据库中的版本号就会发生改变，那么更新订单的操作就不会执行成功，而只能重新查询新版本的订单数据，然后再尝试更新。

所以可以看到，在 oms_order 中专门设计了 version 字段：

但是因为牵涉到更新订单的操作未执行成功（表现为 update 语句返回行数为 0）时的重试机制，代码修改较大，所以在 OmsOrderMapper.xml 和相关订单业务方法中没有实现上述的 ABA 解决方案，感兴趣的同学可以自行调整。

总的来说，因为网络、服务器等导致的不确定因素，重试请求是普遍存在且不可避免的问题。具有幂等性的服务可以克服由于重试问题而导致的数据错误。

所以，总的来说，对于创建订单的服务，可以通过预先生成订单号作为主键,然后利用数据库中“主键唯一约束”的特性，避免重复写入订单，实现创建订单服务的幂等性。对于更新订单的服务，可以通过一个版本号机制，即在每次更新数据之前校验版本号，以及在更新数据的同时自增版本号这样的方式来解决 ABA 问题,以确保更新订单服务的幂等性。

通过这样两种具备幂等性的实现方法,我们可以保证,无论是不是重复请求,订单表中的数据都是正确的。

当然这里讲到的实现订单幂等性的方法，在其他需要实现幂等性的服务中也完全可以套用，只需要这个服务操作的数据保存在数据库中，并且数据表带有主键即可。

实现服务幂等性的方法，远不止本章介绍的这两种，其实，实现幂等性的方法可分为两大类，一类是通过一些精巧的设计让更新本身就是幂等的，这种方法并不能适用于所有的业务。另一类是利用外部的具备一致性的存储（比如 MySQL）来做冲突检测，在设计幂等方法的时候,通常可以顺着这两个思路来展开。

读写分离与分库分表

使用 Redis 作为 MySQL 的前置缓存，可以帮助 MySQL 挡住绝大部分的查询请求。这种方法对于像电商中的商品系统、搜索系统这类与用户关联不大的系统、

效果特别好。因为在这些系统中、任何人看到的内容都是一样的，也就是说，对后端服来说，任何人的查询请求和返回的数据都是一样的。在这种情况下，Redis 缓存的命中率非常高，几乎所有的请求都可以命中缓存。

但是与用户相关的系统（不是用户系统本身，用户信息等相关数据在用户登录时进行缓存，就价值很高），使用缓存的效果就没有那么好了，比如，订单系统、账户系统、购物车系统、订单系统等等。对于这些系统而言，各个用户查询的信息与用户自身相关，即使同一个功能界面，用户看到的数据也是不一样的。

比如，“我的订单”这个功能，用户看到的都是自己的订单数据。在这种情况下，缓存的命中率就比较低了，会有相当一部分查询请求因为命中不了缓存，穿透到 MySQL 数据库中。

随着系统的用户数量越来越多,穿透到 MySQL 数据库中的读写请求也会越来越多，当单个 MySQL 支撑不了这么多的并发请求时,该怎么办?

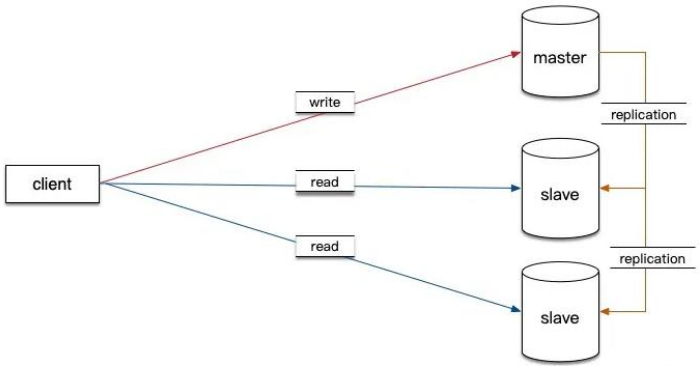
读写分离

读写分离是提升 MySQL 并发能力的首选方案，当单个 MySQL 无法满足要求的时候，只能用多个 MySQL 实例来承担大量的读写请求。MySQL 与大部分常用的关系型数据库一样,都是典型的单机数据库,不支持分布式部署。用一个单机数据库的多个实例组成一个集群,提供分布式数据库服务,是一件非常困难的事情。

一个简单且非常有效的是用多个具有相同数据的 MySOL 实例来分担大量查询请求，也就是“读写分离”。很多系统，特别是互联网系统，数据的读写比例严重不均衡，读写比例一般在 9:1 到几十比 1，即平均每发生几十次查询请求，才会有一次更新请求，那就是说数据库需要应对的绝大部分请求都是只读查询请求。

分布式存储系统支持分布式写是非常困难的，因为很难解决好数据一致性的问题。但分布式读相对来说就简单得多，能够把数据尽可能实时同步到只读实例上，它们就可以分担大量的查询请求了。

读写分离的另一个好处是，实施起来相对比较简单。把使用单机 MySQL 的系统升级为读写分离的多实例架构非常容易，一般不需要修改系统的业务逻辑，只需要简单修改 DAO (Data Access Object,一般指应用程序中负责访问数据库的抽象层)层的代码,把对数据库的读写请求分开，请求不同的 MySQL 实例就可以了。通过读写分离这样一个简单的存储架构升级，数据库支持的并发数量就可以增加几倍到十几倍。所以，当系统的用户数越来越多时，读写分离应该是首要考虑的扩容方案。



主库负责执行应用程序发来的数据更新请求，然后将数据变更同步到所有的从库中。这样，主库和所有从库中的数据一致，多个从库可以共同分担应用的查询请求。

读写分离的数据不一致问题

读写分离的一个副作用是，可能会存在数据不一致的问题。原因是数据库中的数据在主库完成更新后，是异步同步到每个从库上的，这个过程会有一个微小的时间差。正常情况下，主从延迟非常小，以几毫秒计。但即使是这样小的延迟，也会导致在某个时刻主库和从库上数据不一致的问题。

应用程序需要能够接受并克服这种主从不一致的情况，否则就会引发一些由于主从延迟而导致的数据错误。

回顾我们的订单系统业务，用户对购物车发起商品结算创建订单，进入订单页，打开支付页面进行支付，支付完成后，按道理应该再返回到支付之前的订单页。但如果这时马上自动返回到订单页，就很有可能会出现订单状态还是显示“未支付”的问题。因为支付完成后，订单库的主库中订单状态已经更新了，但订单页查询的从库中这条订单记录的状态可能还未更新，如何解决这种问题呢？

其实这个问题并没有特别好的技术手段来解决，所以可以看到，稍微上点规模的电商网站并不会支付完成后自动跳到到订单页，而是增加了一个支付完成页面，这个页面其实没有任何新的有效信息，就是告诉你支付成功的信息。如果想再查看一下刚刚支付完成的订单，需要手动选择，这样就能很好地规避主从同步延迟的问题。

如果是那些数据更新后需要立刻查询的业务，这两个步骤可以放到一个数据库事务中，同一个事务中的查询操作也会被路由到主库，这样就可以规避主从不一致的问题了，还有一种解决方式则是对查询部分单独指定进行主库查询。

总的来说，对于这种因为主从延迟而带来的数据不一致问题，并没有一种简单方便且通用的技术方案可以解决，对此，我们需要重新设计业务逻辑，尽量规避更新数据后立即去从库查询刚刚更新的数据。

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库还是分表之前,我们需要首先明确一个原则，那就是能小拆就小非，能少抖就小多拆。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。选择分库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，系统对订单的查询万式，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。

```java
 /*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。

MySQL 应对海量数据

归档历史数据

订单数据会随着时间一直累积的数据，前面我们说过预估订单的数量每个月订单 2000W，一年的订单数可达 2.4 亿，三年可达 7.2 亿。

数据量越大，数据库就会越慢，这是为什么？我们需要理解造成这个问题的根本原因。无论是“增、删、改、查”中的哪个操作，其本质都是查找数据，因为我们需要先找到数据，然后才能操作数据。

无论采用的是哪种存储系统，一次查询所耗费的时间,都取决于如下两个因素。

1）查找的时间复杂度。2）数据总量。

查找的时间复杂度又取决于如下两个因素。

1）查找算法。

2）存储数据的数据结构。

这两个因素也是面试问题中经常考察的知识。 所以面试官并不是非要问一些“用不上”的问题来为难求职者，这些知识点不是用不上，而是求职者很多时候不知道怎么用。

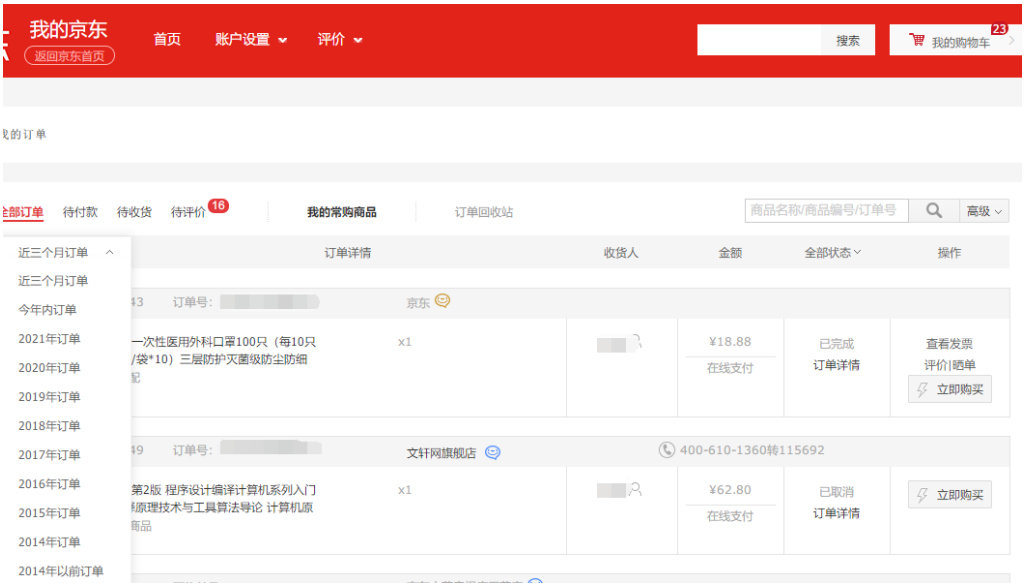
大多数做业务的系统，采用的都是现成的数据库，数据的存储结构和查找算法都是由数据库来实现的，对此，业务系统基本上无法做出任何改变。我们知道 MySQL 的 InnoDB 存储引擎，其存储结构是 B+树，查找算法大多数时候是对树进行查找，查找的时间复杂度就是 O(log n)，这些都是固定的。我们唯一能改变的就是数据总量了。

所以，解决海量数据导致存储系统慢的问题，方法非常简单，就是一个“拆”字，把大数据拆分成若干份小数据，学名称为“分片”(Shard)。拆开之后,每个分片里的数据就没那么多了，然后让查找尽量落在某一个分片上,以此来提升查找性能。

存档历史订单数据

订单数据一般保存在 MySQL 的订单表里，说到拆分 MySQL 的表，前面我们不是已经将到了“分库分表”吗？其实分库分表很多的时候并不是首选的方案，应该先考虑归档历史数据。

以京东为例



可以看到在“我的订单”中查询时，分为了近三个月订单、今年内订单、2021 年订单、2020 年订单等等，这就是典型的将订单数据归档处理。

所谓归档，也是一种拆分数据的策略。简单地说，就是把大量的历史订单移到另外一张历史订单表或数据存储中。为什这么做呢？订单数据有个特点：具备时间属性的，并且随着系统的运行，数据累计增长越来越多。但其实订单数据在使用上有个特点，最近的数据使用最频繁，超过一定时间的数据很少使用，这被称之为热尾效应。

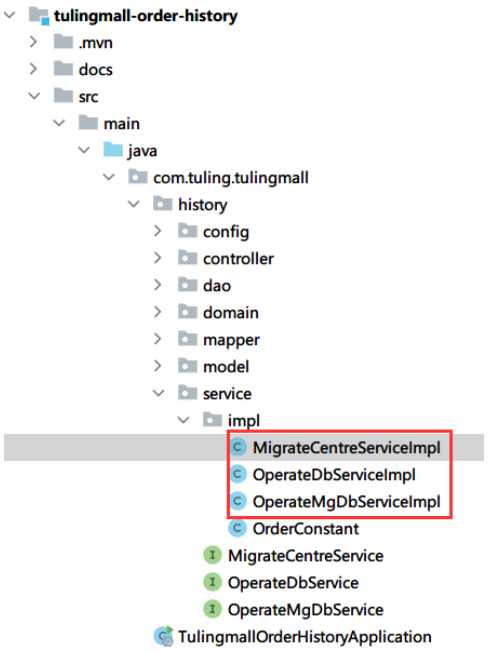
因为新数据只占数据息量中很少的一部分，所以把新老数据分开之后，新数据的数据量就少很多，查询速度也会因此快很多。虽然与之前的总量相比，老数据没有减少太多，但是因为老数据很少会被访问到，所以即使慢一点儿也不会有太大的问题，而且还可以使用其他的存储系统提升查询速度。

这样拆分数据的另外一个好处是，拆分订单时，系统需要改动的代码非常少。对订单表的大部分操作都是在订单完成之前执行的，这些业务逻辑都是完全不用修改的。即使是像退货退款这类订单完成之后的操作，也是有时限的，这些业务逻辑也不需要修改,还是按照之前那样操作订单即可。

基本上只有查询统计类的功能会查到历史订单，这些都需要稍微做些调整。按照查询条件中的时间范围，选择去订单表还是历史订单中查询就可以了。很多大型互联网电商在逐步发展壮大的过程中，长达数年的时间采用的都是这种订单拆分的方案，正如我们前面看到的京东就是如此。

商城历史订单服务的实现

商城历史订单的归档由 tulingmall-order-history 服务负责，其中比较关键的是三个 Service



既然是历史订单的归档，归档到哪里去呢？我们可以归档到另外的 MySQL 数据库，也可以归档到另外的存储系统，这个看自己的业务需求即可，在我们的系统中，我们选择归档到 MongoDB 数据库。

对于数据的迁移归档，我们总是在 MySQL 中保留 3 个月的订单数据，超过三个月的数据则迁出。前面我们说过，预估每月订单 2000W，一张订单下的商品平均为 10 个，如果只保留 3 个月的数据，则订单详情数为 6 亿，分布到 32 个表中，每个表容纳的记录数刚好在 2000W 左右，这也是为什么前面的分库分表将订单表设定为 32 个的原因。

在我们的实现中，OperateDbServiceImpl 负责读取 MySQL 的订单数据和删除已迁出的订单，OperateMgDbServiceImpl 负责将订单数据批量插入 MongoDB，MigrateCentreServiceImpl 负责进行调度服务。

在进行数据迁移的过程需要注意以下两点：

分布式事务?

考察迁移的过程，我们是逐表批次删除，对于每张订单表，先从 MySQL 从获得指定批量的数据，写入 MongoDB，再从 MySQL 中删除已写入 MongoDB 的部分，这里存在着一个多源的数据操作，为了保证数据的一致性，看起来似乎需要分布式事务。但是其实这里并不需要分布式事务，解决的关键在于写入订单数据到 MongoDB 时，我们要记住同时写入当前迁入数据的最大订单 ID，让这两个操作执行在同一个事务之中。

```
@Bean
MongoTransactionManager transactionManager(MongoDatabaseFactory factory){
    //事务操作配置
    TransactionOptions txnOptions = TransactionOptions.builder()
            .readPreference(ReadPreference.primary())
            .readConcern(ReadConcern.MAJORITY)
            .writeConcern(WriteConcern.MAJORITY)
            .build();
    return new MongoTransactionManager(factory,txnOptions);
}
```

```
@Transactional
public void saveToMgDb(List<OmsOrderDetail> orders, long curMaxOrderId,String tableName) {
    log.info("准备将表{}数据迁移入MongoDB，参数curMaxOrderId = {}",tableName,curMaxOrderId);
    mongoTemplate.insert(orders,OmsOrderDetail.class);
    /*记录本次迁移的最大订单ID，下次迁移时需要使用*/
    Query query = new Query(Criteria.where(ORDER_MAX_ID_KEY).is(tableName));
    Update update = new Update();
    update.set(ORDER_MAX_ID_KEY,curMaxOrderId);
    UpdateResult updateResult = mongoTemplate.upsert(query,update,MongoOrderId.class);
    log.info("已记录表{}本次迁移最大订单ID = {}",tableName,curMaxOrderId);
}
```

这样，在 MySQL 执行数据迁移时，总是去 MongoDB 中获得上次处理的最大 OrderId，作为本次迁移的查询起始 ID

```
/*获得上次处理的最大OrderId，作为本次迁移的起始ID*/
long currMaxOrderId = operateMgDbService.getMaxOrderId(factTableName);
log.info("本次表[{}]数据迁移查询记录起始ID = {}",factTableName,currMaxOrderId);
List<OmsOrderDetail> fetchRecords = operateDbService.getOrders(currMaxOrderId,
        tableNo,maxDate,FETCH_RECORD_NUMBERS);
```

当然数据写入 MongoDB 后，还要记得删除 MySQL 中对应的数据。

在这个过程中，我们需要注意的问题是，尽量不要影响线上的业务。迁移如此大量的数据，或多或少都会影响数据库的性能，因此应该尽量选择在闲时迁移而且每次数据库操作的记录数不宜太多。按照一般的经验，对 MySQL 的操作的记录条数每次控制在 10000 一下是比较合适，在我们的系统中缺省是 2000 条。更重要的是，迁移之前一定要做好备份，这样的话，即使不小心误操作了，也能用备份来恢复。

如何批量删除大量数据

在迁移历史订单数据的过程中，还有一个很重要的细节问题:如何从订单表中删除已经迁走的历史订单数据？

虽然我们是按时间迁出订单表中的数据，但是删除最好还是按 ID 来删除，并且同样要控制住每次删除的记录条数，太大的数量容易遇到错误。

```
delete from ${orderTableName} o
WHERE o.id >= #{minOrderId} and o.id <= #{maxOrderId}
order by id
```

这样每次删除的时候，由于条件变成了主键比较，而在 MySQL 的 InnoDB 存储引擎中，表数据结构就是按照主键组织的一棵 B+树，同时 B+树本身就是有序的，因此优化后不仅查找变得非常快,而且也不需要再进行额外的排序操作了。

为什么要加一个排序的操作呢？因为按 ID 排序后，每批删除的记录基本上都是 ID 连续的一批记录，由于 B+树的有序性，这些 ID 相近的记录，在磁盘的物理文件上，大致也是存放在一起的，这样删除效率会比较高，也便于 MySQL 回收页。

关于大批量删除数据，还有一个点需要注意一下，执行删除语句后，最好能停顿一小会，因为删除后肯定会牵涉到大量的 B+树页面分裂和合并，这个时候 MySQL 的本身的负载就不小了，停顿一小会，可以让 MySQL 的负载更加均衡。

本文档分享地址

http://note.youdao.com/noteshare?id=fa12fc29c0651010dba2d69da01c7610&sub=06C0AB3017964A3DB96F4549850B32F2