

01 | 创建和更新订单时，如何保证数据准确无误？

李玥 · 后端存储实战课



你好，我是李玥。

订单系统是整个电商系统中最重要的一個子系统，订单数据也就是电商企业最重要的数据资产。今天这节课，我来和你说一下，在设计和实现一个订单系统的存储过程中，有哪些问题是要特别考虑的。

一个合格的订单系统，最基本的要求是什么？**数据不能错。**

一个购物流程，从下单开始、支付、发货，直到收货，这么长的一个流程中，每一个环节，都少不了更新订单数据，每一次更新操作又需要同时更新好几张表。这些操作可能被随机分布到很多台服务器上执行，服务器有可能故障，网络有可能出问题。

在这么复杂的情况下，保证订单数据一笔都不能错，是不是很难？实际上，只要掌握了方法，其实并不难。

首先，你的代码必须是正确没 Bug 的，如果说是因为代码 Bug 导致的数据错误，那谁也救不了你。

然后，你要会正确地使用数据库的事务。比如，你在创建订单的时候，同时要在订单表和订单商品表中插入数据，那这些插入数据的 INSERT 必须在一个数据库事务中执行，数据库的事务可以确保：执行这些 INSERT 语句，要么一起都成功，要么一起都失败。

我相信这些“基本操作”对于你来说，应该不是问题。

但是，还有一些情况下会引起数据错误，我们一起来看一下。不过在此之前，我们要明白，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。

因为，任何一个电商，它的订单系统的功能都是独一无二的，基于它的业务，有非常多的功能，并且都很复杂。我们在讨论订单系统的存储问题时，必须得化繁为简，只聚焦那些最核心的、共通的业务和功能上，并且以这个为基础来讨论存储技术问题。

订单系统的核心功能和数据

我先和你简单梳理一下一个订单系统必备的功能，它包含但远远不限于：

1. 创建订单；
2. 随着购物流程更新订单状态；
3. 查询订单，包括用订单数据生成各种报表。

为了支撑这些必备功能，在数据库中，我们至少需要有这样几张表：

1. 订单主表：也叫订单表，保存订单的基本信息。
2. 订单商品表：保存订单中的商品信息。
3. 订单支付表：保存订单的支付和退款信息。
4. 订单优惠表：保存订单使用的所有优惠信息。

这几个表之间的关系是这样的：订单主表和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，也就是订单号。

绝大部分订单系统它的核心功能和数据结构都是这样的。

如何避免重复下单？

接下来我们来看一个场景。一个订单系统，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”按钮的时候，浏览器就会给订单系统发一个创建订单的请求，订单系统的后端服务，在收到请求之后，往数据库的订单表插入一条订单数据，创建订单成功。

假如说，用户点击“创建订单”的按钮时手一抖，点了两下，浏览器发了两个 HTTP 请求，结果是什么？创建了两条一模一样的订单。这样肯定不行，需要做防重。

有的同学会说，前端页面上应该防止用户重复提交表单，你说的没错。但是，网络错误会导致重传，很多 RPC 框架、网关都会有自动重试机制，所以对于订单服务来说，重复请求这个事儿，你是没办法完全避免的。

解决办法是，**让你的订单服务具备幂等性**。什么是幂等呢？一个幂等操作的特点是，其任意多次执行所产生的影响均与一次执行的影响相同。也就是说，一个幂等的方法，使用同样的参数，对它进行调用多次和调用一次，对系统产生的影响是一样的。所以，对于幂等的方法，不用担心重复执行会对系统造成任何改变。一个幂等的创建订单服务，无论创建订单的请求发送多少次，正确的结果是，数据库只有一条新创建的订单记录。

这里面有一个不太好解决的问题：对于订单服务来说，它怎么知道发过来的创建订单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，行不行？不太行，因为你很难用 SQL 的条件来定义“重复的订单”，订单用户一样、商品一样、价格一样，就认为是重复订单么？不一定，万一用户就是连续下了两个一模一样的订单呢？所以这个方法说起来容易，实际上很难实现。

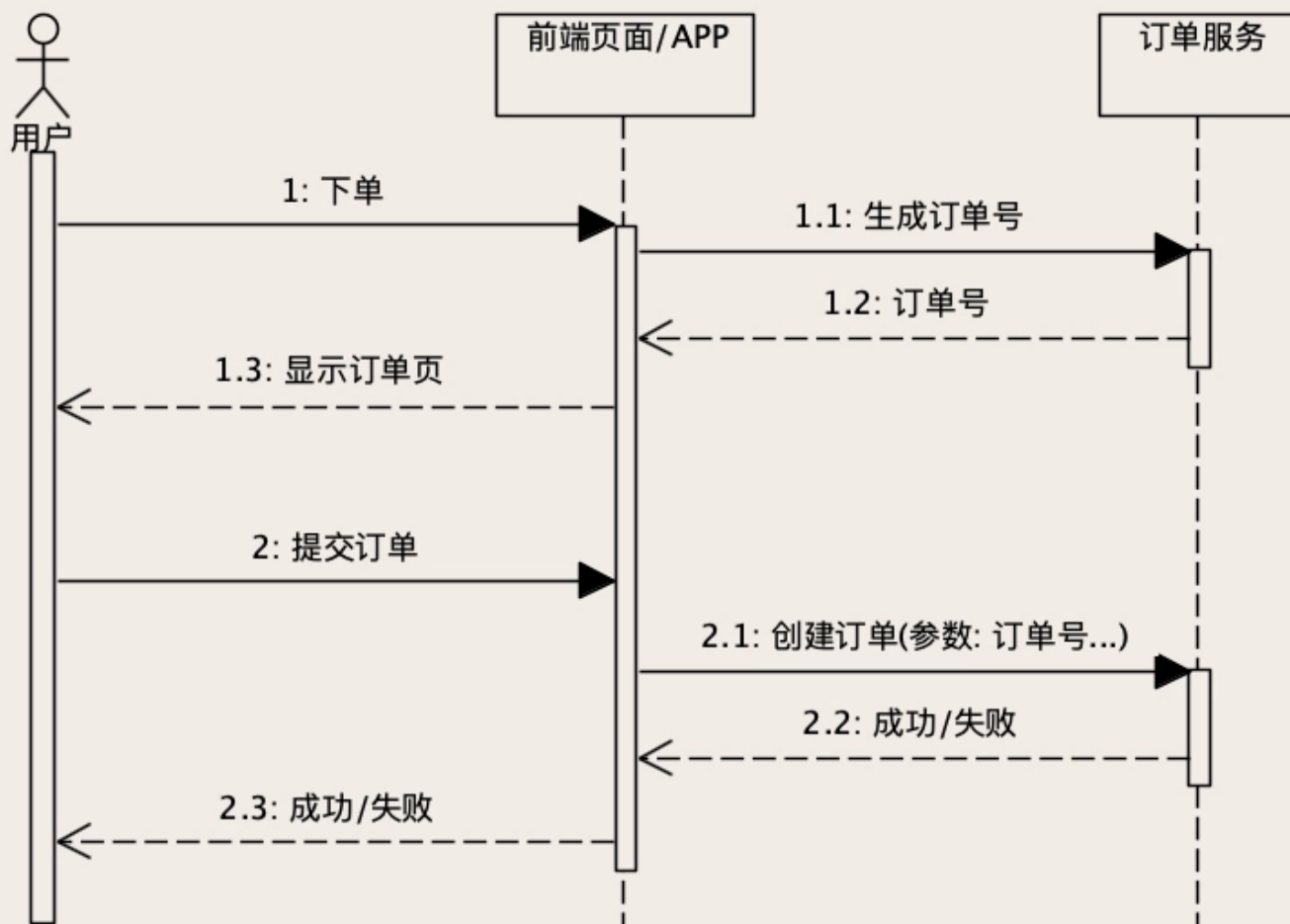
很多电商解决这个问题的思路是这样的。在数据库的最佳实践中有一条就是，数据库的每个表都要有主键，绝大部分数据表都遵循这个最佳实践。一般来说，我们在往数据库插入一条记录的时候，都不提供主键，由数据库在插入的同时自动生成一个主键。这样重复的请求就会导致插入重复数据。

我们知道，表的主键自带唯一约束，如果我们在一条 INSERT 语句中提供了主键，并且这个主键的值在表中已经存在，那这条 INSERT 会执行失败，数据也不会被写入表中。**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。**

具体的做法是这样的，我们给订单系统增加一个“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，前端页面先调用这个生成订单号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。

这个订单号也是我们订单表的主键，这样，无论是用户手抖，还是各种情况导致的重试，这些重复请求中带的都是同一个订单号。订单服务在订单表中插入数据的时候，执行的这些重复 INSERT 语句中的主键，也都是同一个订单号。数据库的唯一约束就可以保证，只有一次 INSERT 语句是执行成功的，这样就实现了创建订单服务幂等性。

为了便于你理解，我把上面这个幂等创建订单的流程，绘制成了时序图供你参考：



还有一点需要注意的是，如果是因为重复订单导致插入订单表失败，订单服务不要把这个错误返回给前端页面。否则，就有可能出现这样的情况：用户点击创建订单按钮后，页面提示创建订单失败，而实际上订单却创建成功了。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就可以了。

如何解决 ABA 问题？

同样，订单系统各种更新订单的服务一样也要具备幂等性。

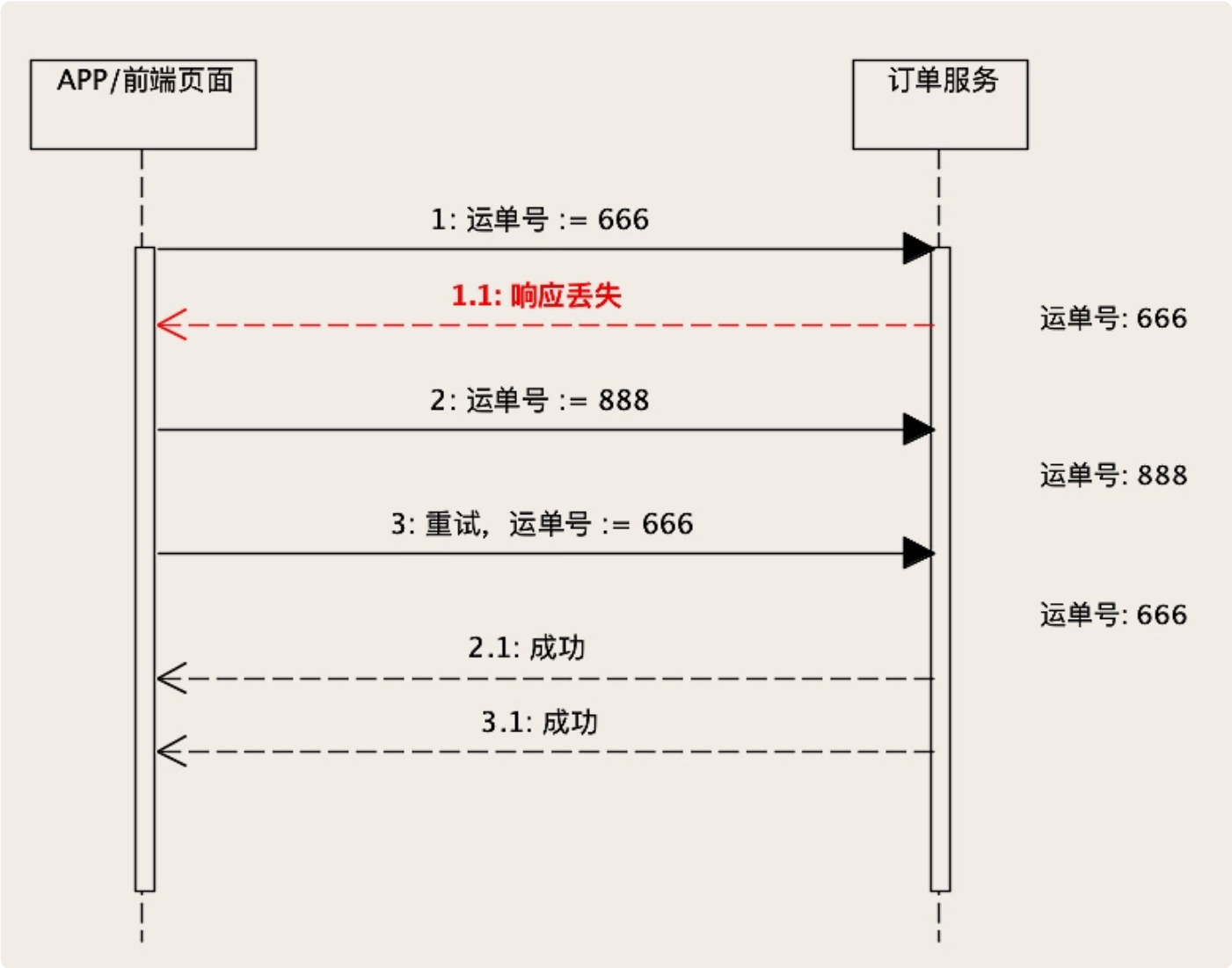
这些更新订单服务，比如说支付、发货等等这些步骤中的更新订单操作，最终落到订单库上，都是对订单主表的 UPDATE 操作。数据库的更新操作，本身就具备天然的幂等性，比如说，你把订单状态，从未支付更新成已支付，执行一次和重复执行多次，订单状态都是已支付，不用我们做任何额外的逻辑，这就是天然幂等。

那在实现这些更新订单服务时，还有什么问题需要特别注意的吗？还真有，在并发环境下，你需要注意 ABA 问题。

什么是 ABA 问题呢？我举个例子你就明白了。比如说，订单支付之后，小二要发货，发货完成后要填个快递单号。假设说，小二填了一个单号 666，刚填完，发现填错了，赶紧再修改成 888。对订单服务来说，这就是 2 个更新订单的请求。

正常情况下，订单中的快递单号会先更新成 666，再更新成 888，这是没问题的。那不正常情况呢？666 请求到了，单号更新成 666，然后 888 请求到了，单号又更新成 888，但是 666 更新成功的响应丢了，调用方没收到成功响应，自动重试，再次发起 666 请求，单号又被更新成 666 了，这数据显然就错了。这就是非常有名的 ABA 问题。


具体的时序你可以参考下面这张时序图：



ABA 问题怎么解决？这里给你提供一个比较通用的解决方法。给你的订单主表增加一列，列名可以叫 version，也即是“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中，需要把这个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，是否和消息中的版本号一致，如果不一致就拒绝更新数据。如果版本号一致，还需要再更新数据的同时，把版本号 +1。“比较版本号、更新数据和版本号 +1”，这个过程必须在同一个事务里面执行。

具体的 SQL 可以这样来写：

 复制代码

```
1 UPDATE orders set tracking_number = 666, version = version + 1
2 WHERE version = 8;
```

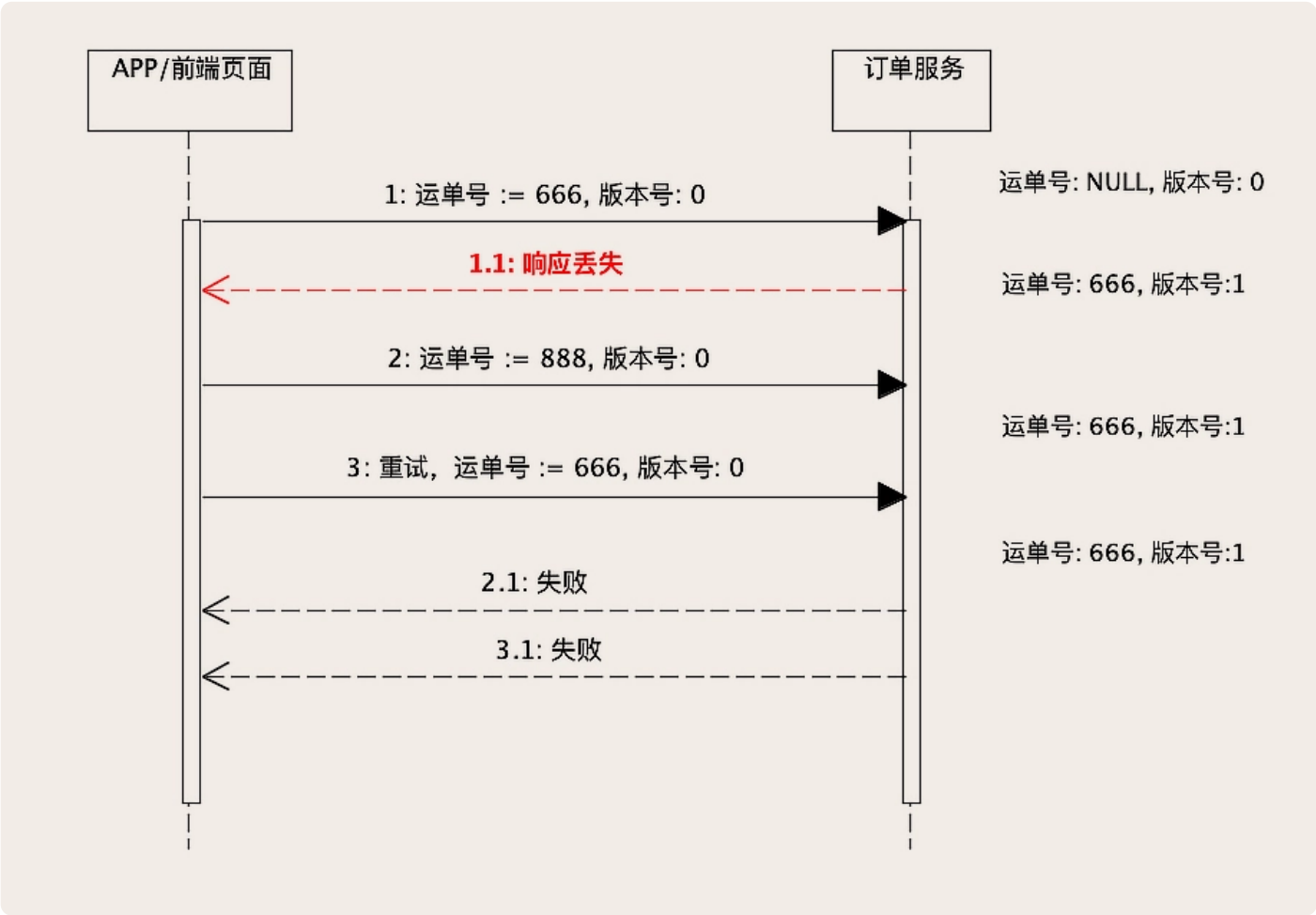
在这条 SQL 的 WHERE 条件中，version 的值需要页面在更新的时候通过请求传进来。

通过这个版本号，就可以保证，从我打开这条订单记录开始，一直到我更新这条订单记录成功，这个期间没有其他人修改过这条订单数据。因为，如果有其他人修改过，数据库中的版本号就会改变，那我的更新操作就不会执行成功。我只能重新查询新版本的订单数据，然后再尝试更新。

有了这个版本号，再回头看一下我们上面那个 ABA 问题的例子，会出现什么结果？可能出现两种情况：

1. 第一种情况，把运单号更新为 666 的操作成功了，更新为 888 的请求带着旧版本号，那就会更新失败，页面提示用户更新 888 失败。
2. 第二种情况，666 更新成功后，888 带着新的版本号，888 更新成功。这时候即使重试的 666 请求再来，因为它和上一条 666 请求带着相同的版本号，上一条请求更新成功后，这个版本号已经变了，所以重试请求的更新必然失败。

无论哪种情况，数据库中的数据与页面上给用户的反馈都是一致的。这样就可以实现幂等更新并且避免了 ABA 问题。下图展示的是第一种情况，第二种情况也是差不多的：



小结

我们把今天这节课的内容做一个总结。今天这节课，实际上就讲了一个事儿，也就是，实现订单操作的幂等的方法。

因为网络、服务器等等这些不确定的因素，重试请求是普遍存在并且不可避免的。具有幂等性的服务可以完美地克服重试导致的数据错误。

对于创建订单服务来说，可以通过预先生成订单号，然后利用数据库中订单号的唯一约束这个特性，避免重复写入订单，实现创建订单服务的幂等性。对于更新订单服务，可以通过一个版本号机制，每次更新数据前校验版本号，更新数据同时自增版本号，这样的方式，来解决 ABA 问题，确保更新订单服务的幂等性。

通过这样两种幂等的实现方法，就可以保证，无论请求是不是重复，订单表中的数据都是正确的。当然，上面讲到的实现订单幂等的方法，你完全可以套用在其他需要实现幂等的服务中，只需要这个服务操作的数据保存在数据库中，并且有一张带有主键的数据表就可以了。

思考题

实现服务幂等的方法，远不止我们这节课上介绍的这两种，课后请你想一下，在你负责开发的业务系统中，能不能用这节课中讲到的方法来实现幂等？除了这两种方法以外，还有哪些实现服务幂等的方法？欢迎你在留言区与我交流互动。

感谢你的阅读，如果你觉得今天的内容对你有所帮助，也欢迎把它分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (87)



李玥 置顶

2020-02-26

hello，我是李玥。之后我都会在留言板上同步上节课思考题的答案，欢迎你跟我一起学习讨论。

在课前加餐这节课里，我给你留了道思考题，让你作为公司的CTO，想一想上节课我们提到的电商系统，它的技术选型应该是什么样的？

关于这个问题，我是这么理解的。

技术选型本身没有好与坏，更多的是选择“合适”的技术。对于编程语言和技术栈的选择，我认为需要从两方面考虑，一方面就是团队的人员配置，尽量选择大家熟悉的技术，第二个方面就是要考察选择的技术它的生态是不是够完善。这两个原则在选择编程语言、技术栈、云服务和存储的时候都是适用的。

如何根据业务来选择合适的存储系统，这是个很大的话题，我会在后面的课程中陆续穿插的来和你讲，什么场景下应该选择什么样的存储，敬请期待。

共 6 条评论 >

👍 64



Panmax

2020-03-05

课程中给的那个 SQL 语句很危险啊😂，where 条件丢掉了最重要的订单ID。应该改为：

```
UPDATE orders set tracking_number = 666, version = version + 1  
WHERE id = 12345 and version = 8;
```

共 16 条评论 >

👍 110



川杰

2020-02-26

请问，生成订单号 服务的一般逻辑会是怎样的？想来想去，如果要想这个ID全局唯一，只能带上时间，可是如果带上时间，像那种，不小心点了两次按钮的情况，必然是两个不同的订单号；请问这个问题怎么解决？

作者回复: 如果单纯是生成GUID（全局唯一ID）方法有很多，比如小规模系统完全可以用MySQL的Sequence或者Redis来生成。大规模系统也可以采用类似雪花算法之类的方式分布式生成GUID。

但是订单号这个东西又有点儿特殊要求，比如在订单号中最好包含一些品类、时间等信息，便于业务处理，再比如，订单号它不能是一个单纯自增的ID，否则别人很容易根据订单号计算出你大致的销量，所以订单号的生产算法在保证不重复的前提下，一般都会加入很多业务规则在里面，这个每家都不一样，算是商业秘密吧。

共 20 条评论 >

👍 62



业余爱好者

2020-02-26

每次请求之前必须先生成一个唯一的请求id,服务端将该id暂时放入redis。客户端请求时必须携带上这个id，接口会首先到redis中查询，如何有的话就继续后续的处理逻辑，同时删除该id,没有的话就退出，返回不能重复请求的错误到客户端。

一句话总结：每次处理必须对应一个一次性的token。

作者回复: 这个思路非常好，并且可以适用很多的场景。

如果是超大规模的系统，可能用一个Redis实例来生成和验证这个GUID就忙不过来了，大家也可以想一下有没有什么性能上更好的方式？

共 26 条评论 >

👍 53



家庆

2020-03-06

李老师好，请问生成一个唯一订单号放在前端，如何保证客户绕过客户端直接发送请求恶意乱填订单号的问题，符合规则的订单号，而这个订单号有可能是后续系统生成的。

作者回复: 有的同学已经在留言区给出了答案，那就是，在Redis里面缓存一下给出去的订单号，收到客户端请求的时候，先验证一下这个订单号在缓存中是否存在，就可以解决这个问题了。

共 5 条评论 >

👍 31



鸠摩智

2020-02-28

老师，生成全局唯一订单号如果不是自增的，插入mysql innodb表的时候，底层的B+树索引是不是会发生页分裂等问题，影响插入性能？如果遇到大促，短时间生成大量订单，写入会成为瓶颈不？

作者回复: 这确实是一个需要考虑的性能问题。

所以很多业务在设计订单号规则的时候都不是完全随机的，一般都是递增的。这种情况下，页分裂就不会特别严重。

共 13 条评论 >

👍 29



Sunday

2020-03-07

老师，用户手机号注册会出现多次注册现象，但因为有注销功能，我又不能设置成唯一主键，所以想问下有没有什么好的解决办法？目前是用redis锁控制并发的。还有ABA问题，一般业务中这种更新都是牵扯到更新多条，使用事务，但事务的话因为隔离级别问题，比如可重复读，感觉并发情况还是有问题

作者回复: 手机号注册重复这种情况，可以简单的用insert if not exist来解决，比如：

```
insert into user (name, phone)
select * from
(select '张三', 13988888888)
as tmp
where not exists (
```

```
select phone from user where phone = 13988888888  
) limit 1;
```

共 13 条评论 >

 21



约书亚

2020-02-27

请教您一个实际问题：

无论采用哪种生成订单id的方式，短时间内都可能出现靠前的id后提交（生成订单的问题），这样一定程度上数据库接收到的id不是严格按时间递增的。而页面浏览的场景，尤甚。请问在您做过的系统里，这会带来一定程度上的性能下降嘛？

作者回复： 你是指在数据库中插入数据的时候，写入索引的性能下降吗？

共 23 条评论 >

 17



王超

2020-03-24

老师，如果是一个预约中台的项目，只对外提供API能力，前端的预约入口应用不由自己负责，无法要求别人的前端应用在进入下单界面的时候就调用统一生成单号的服务，这种情况下有什么推荐方案么，感谢

作者回复： 这种下单接口，订单内容中一般都会有用户ID，可以通过“1秒钟内每个用户只允许最多下1单”，这种方式来判重。因为一般来说，由于重试导致的重复请求之间的时间间隔都是比较短的。

```
INSERT INTO my_orders (...)
```

```
SELECT * FROM
```

```
(SELECT 'aaa', 'bbb', 'ccc') AS tmp -- 实际要insert的数据
```

```
WHERE NOT EXISTS (
```

```
    SELECT userid FROM my_orders WHERE userid = 'bbb' and last_update_time > now - 1秒
```

```
) LIMIT 1;
```



 17



fgdgtz

2020-03-21

老师你好，我觉的订单号设为唯一索引约束即可，仍然有个自增主键，订单号对外用，主键不暴露，这样表空间还是连续紧密的，其他关联订单表仍然以订单号作外键

作者回复: 这样也是没问题的，很多情况下架构不是只有一种解决方案的。这种设计的好处就是主键连续，使用订单号做主键的好处是，最常用的按订单号查询会少走一次索引，各有优劣，都可以选择。

共 8 条评论 >

👍 16



leo

2020-03-22

关于数据报表类数据，现在主流解决方案能否推荐一下。本来想推es，但运营人员觉得操作复杂。大厂的实践是什么呢

作者回复: 后面我会专门讲，海量数据应该怎么查才会快一些。

但是，有一个原则上，底层用什么存储，应该尽量对最终用户透明，不要因为换了存储系统，就改变用户的操作习惯。

共 2 条评论 >

👍 10



撒旦的堕落

2020-03-03

老师 我曾经看到文章说用disruptor大规模生成预订单 可以动态改变生成速率 并且可以防止瞬间流量 不过没有说细节 老师在这方面有思路么

作者回复: disruptor是一个高性能队列，我没有听过类似的用法，我猜测可能是利用disruptor这个队列，预先生成大量的订单号，高并发的情况下，可以瞬间从队列中取出大量预先生成好的订单号，避免订单号生成的速度跟不上瞬时的请求量。

共 3 条评论 >

👍 10



Javatar

2020-11-24

老师一言不和就讲了两个基本的面试题：

1. 如何防止重复提交？
2. 谈谈乐观锁？

db唯一约束搭配乐观锁，日常防止并发场景应该是足够用了。

还是题目中的场景，实现幂等应该也可以使用redis：

订单号还是事先生成，然后用每次创建之前，先在redis中setnx一把，key要包含订单号。如果setnx成功，证明没有创建；如果setnx失败，说明已经创建了，此时应该返回幂等成功。

当然这会带来一个问题，就是redis中的数据会不断增多，可以考虑增加一个定时清理的任务，把已经“结束”的订单，对应的key清理掉

共 1 条评论 >

👍 9



L

2020-03-01

最喜欢看评论了，总能找到解决我疑问的答案。



👍 7



Ling

2021-05-25

1. 作者讲了什么？

针对**必须保证数据可靠性的写需求**（例如订单操作）的操作，如何保证数据库的可靠性（准确）

引入了**幂等**的概念，表明重复请求在实际生产环境中不可避免；

要避免重复请求，就需要使请求幂等，并通过列子说明，如何实现请求的幂等

2. 作者是怎么把事情说明白的？

通过举了电商系统订单的实际的几个例子：

1. 订单创建，如何保证保证不重复下单
2. 订单更新，如何保证准确和幂等

3. 为了讲明白，作者讲了哪些要点？ 哪些是亮点？

要点

1. 实际生产环境，重复请求不可避免（客户端重试、服务间重试）

2. 增：

1. 生成唯一订单ID，订单ID全局唯一，并递增（为了MySQL建立索引时，如果不递增引起页分裂等问题，导致写入性能下降）

2. 将生成的ID放入缓存

3. order表的订单ID字段设置唯一索引约束

4. 客户端请求带上订单ID，如果重复插入会引起唯一索引重复，导致事务失败

5. 由于订单已经创建成功，对于由于唯一索引冲突导致的创建失败，需要给前端返回“创建成功”，防止对客户产生困扰

3. 改：更改请求接口，带上当前version；SQL条件加上`WHERE version=x`实现天然幂等

```
```sql
UPDATE order SET tarcining_num=666 AND version=version+1 WHERE id=100 AND version=6
```
```

4. 对于作者所讲，我有哪些发散性思考？

1. 对于任何“写”操作的接口，在设计的时候，都需要考虑**幂等**的概念。对于需要强保证数据可靠性的服务，需要设计**幂等**

2. 对于下订单、发表文章、发表评论这些写入服务的接口，除了幂等，还有防止重复提交、快速提交的问题

– 一方面：客户端需要做锁，用户点击“发布”按钮后，在网络请求没有返回，或者超时之前，用户都不可以继续点击“发布按钮”，界面可以将按钮置灰或者转菊花

– 另一方面：服务端需要进行加锁

1. 请求接口时，获取一个锁

锁的粒度：同一用户的同一操作逻辑

锁名称规则：业务名称+用户ID（例如：post_msg:100101）

2. 给锁设置过期时间2-3秒，防止业务逻辑执行错误，用户一直被锁住

3. 如果被锁了，返回“正在处理，请勿重复提交”

4. 没有被锁，执行正常逻辑，在逻辑结束后，删掉锁

5. 在未来哪些场景，我可以使用它？

1. 在做任何更新接口时，时刻考虑是否需要实现**幂等**



7



京京beaver

2020-03-03

提供一种思路，电商网站一般都是从购物车进入结算的，意味着购物车已经有了待结算商品列表。在结算成单接口，做下单和清除购物车商品两个动作。对于幂等来讲，购物车系统对本次购物车数据有个购物车id，成单接口的入参是购物车id。购物车id本身是全局唯一的，所以订单系统只要检查购物车id是否被处理过就可以了。

而结算服务需要做两个动作：使用购物车id去生成订单id，清除购物车，这两个动作调用购物车服务和订单服务，构成了分布式事务。一般为了高并发，是不做事务的，采用后台修数来应对错误。

共 8 条评论 >

6



L

2020-03-01

我有个疑问，在提交订单的时候，一种是先拿着订单号去查库，让业务代码校验是否存在，另一种是直接利用库表主键唯一约束抛异常，这两种处理方式哪种性能更好？

共 4 条评论 >

5



aoe

2020-02-27

老师，您在”幂等创建订单的流程“的时序图中：1. 下单；1.1. 生成订单号, 按这个逻辑是不能防止重复提交。

因为：下单后会走一个完整的生成订单流程

留言中”业余爱好者“的思路是正确的

作者回复：“业余爱好者”同学相比课中的方案，多了一步在Redis中验证订单号，我理解，这种设计一般是出于安全的考虑，确保提交的订单号确实是我们生成的，而不是来自外部的攻击。

真正的幂等或者说是防重，还是要依靠数据库的唯一约束。

也欢迎你说一下你的理解。

共 6 条评论 >

5



划过天空阿忠

2020-03-19

下单接口的参数做签名，放redis。且每个签名做setnx判断，并且设定时长，规定时间内有重复的就是重复下单



👍 4



陈迪

2020-02-29

第一个防重复创建订单的解决方案中有个疑点，唯一的订单号是在进入“创建订单页面”时创建，用户如果对这个页面不小心同时开了多个tab页，每次打开一个tab页都会生成个新订单号，那就可以提交多个“一模一样”的订单了，而且用户本意应该是一个订单。老师怎么看待这个问题？

作者回复: 这种情况我们确实没提到。如果用户打开了多个tab页，只会产生多个订单号，这个时候还没有生成新订单呢。

只有他在每个tab也都点“下单”按钮，才会产生多个订单。这种情况不存在“误操作”的可能了。

共 8 条评论 >

👍 3