这个项目是做什么的?

为什么要做这个项目?

项目是不是跟着视频做的?

数据库

- 表是怎么设计的?
- 数据库的索引是怎么加的?
- MySQL 某条语句执行的好慢，访问数据库超时了，你怎么排查?
- 那假如找到慢的 SQL 了怎么优化呢?

用户

- 用户管理是怎么实现的?
- 有没有可能攻击者盗用sessionid绕过身份认证？假如说浏览器禁用cookie，sessionid只能重写在URL中，攻击者发起会话固定攻击怎么办?
- 浏览器突然关闭了，你又如何让用户自动退出呢?
- 怎么保证用户密码的安全?
- 怎么防止别人用脚本批量注册刷掉网站?
- 当多点部署服务器的时候，你在服务器a上登录，服务器b如何得到用户登录信息？如果每个服务器都有该用户的session，如果有很大的用户量，假如500万用户是每个服务器存了8g的session信息，如果再来500万，岂不是各个服务器均要扩容到16G，即每个服务器都需扩容容量，怎么优化？如果在多个实例之间处理同一个用户的登录状态?

商品

- 商品管理是怎么实现的?
- 商品信息假如字段越来越多，而且流量上来了怎么办?

订单-支付

- 订单管理是怎么实现的?
- 支付系统是怎么实现的?
- 为什么支付系统要独立出来?
- 怎么防止重复下单?
- 如何解决客户的恶意下单问题?
- 如果用户支付成功，但是看到订单的状态还是未支付，于是又去支付了一次，会重复支付吗?
- 订单id是怎么生成的？怎么保证唯一性?
- 用户下单这个时刻，正好赶上商品调价怎么处理?
- 假如支付完了微信一直不回调怎么办?
- 怎么防止超卖?
- 减库存成功了，但是生成订单失败了，该怎办?

缓存

- 购物车管理是怎么实现的?
- 为什么用Redis不直接用MySQL？用Redis丢数据了怎么办?
- Redis挂了怎么办?

这个项目是做什么的?

该项目基于SpringBoot的SSM框架，数据库采用MySQL+Redis，消息中间件采用RocketMQ的仿电商系统+通用型支付系统的双系统项目。

完整的业务流程包括，用户注册登录，查看商品列表，可以点击查看商品详细信息。找到想购买的商品加入购物车。添加商品成功查看详情，就跳到购物车界面。接着去结算。这里要选收货地址，收货地址可以添加，也可以删除，然后点击去结算订单。这个时候已经是提交下单成功了。

下单跟支付是两个步骤，下单成功，但是还没有支付。接着我们选择使用支付宝，你使用微信也是可以的。点击选择支付宝。它会跳到一个界面是支付宝的付款页，打开手机上的支付宝APP扫码。扫完了之后支付。支付完成页面它自动跳转，跳到了订单列表页。这是我们刚刚下单的商品。付款状态变成了已付款。支付完成之后，我们会收到支付结果的通知，通知是异步的。通知的消息是由微信或者支付宝发起的，对我们的支付系统发起异步请求。支付系统收到消息之后，再经过消息队列MQ，把这个支付成功的消息告诉其他的业务更改订单状态为已支付。

为什么要做这个项目?

因为我们学校的JAVAWEB课设需要用JAVA技术栈来完成一个后端项目，然后我自己对微电商支付这块比较感兴趣，并且这也是一个非常好的思考学习技术的机会，所以就想通过这个项目锻炼自己的业务开发的能力，并借此提升自己使用JAVA语言、数据库、框架和中间件的水平。

项目是不是跟着视频做的?

不是跟着视频。自己学了相应的知识，所以就想着通过做一个项目的形式把他们整合起来，以此来锻炼自己学过的一些知识，不过一开始没做过啥项目，就想着通过 GitHub 搜索一些项目教程，或者看看一些视频教程看看别人都是怎么做的，然后在基于自己的一个理解，去做这个项目。

数据库

表是怎么设计的?

首先我考虑的是表关系，需要结合电商支付的场景结合自己生活中购买商品的场景去淘宝下单或者是点个外卖，自然而然就想到有这些信息设计来进行想象：

- **首先想一下需要哪几张表.....**

首先有用户下单，所以需要一个**用户表**。

用户做什么事？用户要下单，所以有一个**订单表**。

用户下单前首先要购买商品，所以有一个**商品表**。

好。有了商品，我们自然可以想到。商品要分类，有一个**分类表**。

分类用户把商品先加入购物车，然后再下单。购物车我们这里设计直接用**redis**保存就好了，不用存到数据库，所以是不用为它建一个表的。

下单了之后要支付，所以有一个**支付表**。

支付这个订单自然而然应该还有一个订单详情，一个订单里面可以包含多个商品，所以我们设计一个**订单详情表**。

用户下单的时候要填地址，所以还有一张**收货地址****表****。收货地址肯定是跟着人走的。收货地址，所以放到用户这边。

- **其次想一下表与表之间的关系.....**

用户和收货地址。一个用户可以有多个收货地址——一对多的关系。

分类和商品。一个分类，下面有多个商品——一对多的关系。

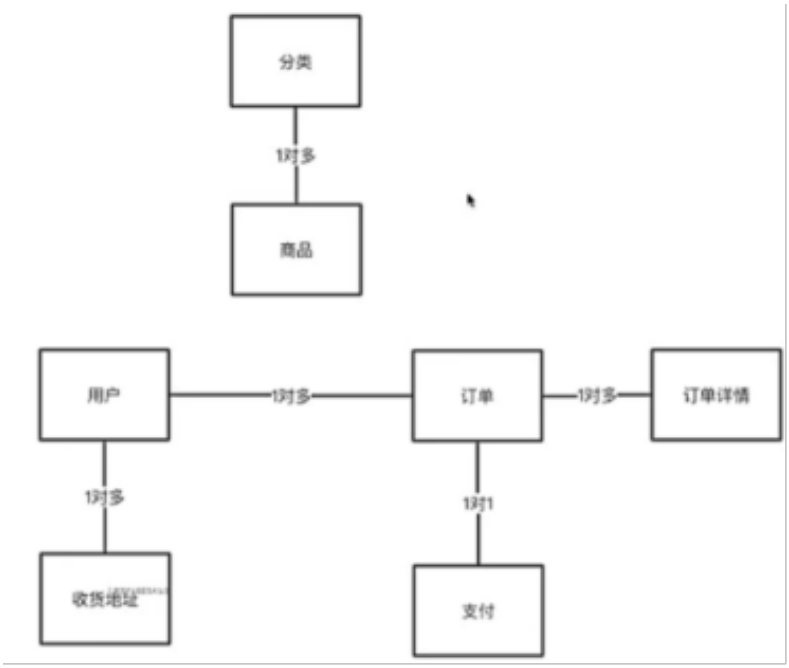
订单和订单详情。一个订单有多个订单详情——一对多的关系。

订单和支付。一个订单只能支付一次——一对一的关系。

用户和商品，商品和订单他们有什么对应关系吗？其实没什么对应关系。

用户和订单是有对应关系的，一个用户可以有多个订单——一对多的关系。

商品和订单的关系，一个商品可以被多个用户下多个订单。那一个订单它又可以买多个商品，相当于是多对多关系，所以我们没必要把它强行关联。



然后结合了表的关系，我就开始考虑表的结构，字段的设计了：

首先是用户表

第一个字段是ID自增，然后是用户名，密码。密码这里我们使用MD5加密，接下来是邮箱手机。用户可能会忘记密码，所以我们设置了一个找回密码的问题和答案。还有一个字段是角色，他是管理员还是普通用户。最后两个字段是创建和更新时间分类表。每个表的时间戳两个字段。主要是为了以后排查业务问题，用的创建时间和更新时间。我们每张表里面都有这两个字段。

用户表结构

```
CREATE TABLE `mall_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '用户表id',
  `username` varchar(50) NOT NULL COMMENT '用户名',
  `password` varchar(50) NOT NULL COMMENT '用户密码，MD5加密',
  `email` varchar(50) DEFAULT NULL,
  `phone` varchar(20) DEFAULT NULL,
  `question` varchar(100) DEFAULT NULL COMMENT '找回密码问题',
  `answer` varchar(100) DEFAULT NULL COMMENT '找回密码答案',
  `role` int(4) NOT NULL COMMENT '角色0-管理员,1-普通用户',
  `create_time` datetime NOT NULL COMMENT '创建时间',
  `update_time` datetime NOT NULL COMMENT '最后一次更新时间',
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_name_unique` (`username`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

其次是分类表

第一个字段是ID自增，注意第二个字段parentID父类ID，当ID等于0的时候说明是根结点。分类我们这里是支持多级分类的，这个分类其实是一个树状的结构。

分类表结构

```
CREATE TABLE `mall_category` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '类别Id',
  `parent_id` int(11) DEFAULT NULL COMMENT '父类别id当id=0时说明是根节
  `name` varchar(50) DEFAULT NULL COMMENT '类别名称',
  `status` tinyint(1) DEFAULT '1' COMMENT '类别状态1-正常,2-已废弃',
  `sort_order` int(4) DEFAULT NULL COMMENT '排序编号,同类展示顺序,数值
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

再接下来是商品表

表里面价格这个字段price，它的类型是decimal类型。价格数据库里面设置都用decimal类型，20逗号2。表示的是最大整数位支持18位，小数位两位。

产品表结构

```
CREATE TABLE `mall_product` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '商品id',
  `category_id` int(11) NOT NULL COMMENT '分类id,对应mall_category表的
  `name` varchar(100) NOT NULL COMMENT '商品名称',
  `subtitle` varchar(200) DEFAULT NULL COMMENT '商品副标题',
  `main_image` varchar(500) DEFAULT NULL COMMENT '产品主图,url相对地址
  `sub_images` text COMMENT '图片地址,json格式,扩展用',
  `detail` text COMMENT '商品详情',
  `price` decimal(20,2) NOT NULL COMMENT '价格,单位-元保留两位小数',
  `stock` int(11) NOT NULL COMMENT '库存数量',
  `status` int(6) DEFAULT '1' COMMENT '商品状态.1-在售 2-下架 3-删除',
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

再看支付表

自增ID订单号，platform就是支付平台，1是支付宝，2微信下面。platform number是第三方支付平台给我们的一个订单号，每次发起支付时自己有一个订单号，请求支付平台之后，他会返回给我们一个支付平台的订单号。下一个字段是支付状态。

支付信息表结构

```
CREATE TABLE `mall_pay_info` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) DEFAULT NULL COMMENT '用户id',
  `order_no` bigint(20) DEFAULT NULL COMMENT '订单号',
  `pay_platform` int(10) DEFAULT NULL COMMENT '支付平台:1-支付宝,
  `platform_number` varchar(200) DEFAULT NULL COMMENT '支付宝支
  `platform_status` varchar(20) DEFAULT NULL COMMENT '支付宝支付
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

然后是订单表

可以看到最下面paymenttime，starttime，endtime，closetime很多个时间，支付时间、发货时间、交易完成，交易关闭好多个时间。搞这么多时间是原因一可能是要求前端界面就要展示这些时间，那他要展示，我必须得存着，不然怎么拿得出这些数据呢？原因二是方便排查问题。假如有一个用户投诉过来了，说那个订单很久没有收到货，那你肯定要过来看时间。这到底是卡在哪一步了，是吧？原因三是为了数据分析。我想知道我们平台从用户支付到最终发出获取，这段时间平均大概要多久呢？这时候你要把所有的数据给他查出来，做个统计。

另一个问题，这四个字段，这四个时间你是怎么想出来的呢？为什么要设定这四个字段？其实是根据订单的状态来的。想想刚开始下单的时候是新订单，新订单我记录一个下单时间，下单时间其实createtime已经有了。接着之后用户会去支付，所以有一个支付完成的时间。下完单之后，用户也可以不支付，他可能会取消，那么取消就有一个交易关闭的时间。总之你可以根据订单状态一旦有变化，就记录对应的变化。

订单表结构

```
CREATE TABLE `mall_order` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '订单id',
  `order_no` bigint(20) DEFAULT NULL COMMENT '订单号',
  `user_id` int(11) DEFAULT NULL COMMENT '用户id',
  `shipping_id` int(11) DEFAULT NULL,
  `payment` decimal(20,2) DEFAULT NULL COMMENT '实际付款金额,单位是
  `payment_type` int(4) DEFAULT NULL COMMENT '支付类型,1-在线支付',
  `postage` int(10) DEFAULT NULL COMMENT '运费,单位是元',
  `status` int(10) DEFAULT NULL COMMENT '订单状态:0-已取消-10-未付款
  `payment_time` datetime DEFAULT NULL COMMENT '支付时间',
  `send_time` datetime DEFAULT NULL COMMENT '发货时间',
  `end_time` datetime DEFAULT NULL COMMENT '交易完成时间',
  `close_time` datetime DEFAULT NULL COMMENT '交易关闭时间',
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL COMMENT '更新时间',
  PRIMARY KEY (`id`),
  UNIQUE KEY `order_no_index` (`order_no`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

然后是订单明细表

开始也说了，每下一个订单我可能包含多个商品。所以我们需要一个明细表来记录，明细表里面有ordernumber就是订单号了。再往下是商品ID、商品的名称、商品图片以及生成订单时商品的单价。为什么要存这么多呢？只存一个商品ID不就行了吗？有了商品ID剩下的商品名称、图片都可以去商品表里面查。现在存这么多，不是占用空间吗？这个空间还非占不可。想想商品的名称，还有图片以及它的单价，这些都是可变的。拿商品价格来说，昨天卖三毛钱一件的，明天可能卖五毛钱一件。

所以我肯定要记录用户购买那一瞬间这些商品属性的值是多少，这一点非常重要。

表关联的时候，一定要想到数据是不是会变化的。如果是会变化的那是不是应该考虑给他做一个存档？

订单明细表结构

```
CREATE TABLE `mall_order_item` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '订单子表id',
  `user_id` int(11) DEFAULT NULL,
  `order_no` bigint(20) DEFAULT NULL,
  `product_id` int(11) DEFAULT NULL COMMENT '商品id',
  `product_name` varchar(100) DEFAULT NULL COMMENT '商品名称',
  `product_image` varchar(500) DEFAULT NULL COMMENT '商品图片地址',
  `current_unit_price` decimal(20,2) DEFAULT NULL COMMENT '生成订单
  `quantity` int(10) DEFAULT NULL COMMENT '商品数量',
  `total_price` decimal(20,2) DEFAULT NULL COMMENT '商品总价,单位是元
  `create_time` datetime DEFAULT NULL,
  `update_time` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `order_no_index` (`order_no`) USING BTREE,
  KEY `order_no_user_id_index` (`user_id`,`order_no`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

下面是收货地址表

收货地址这个就很普通了，省份城市邮政编码详细地址，这个没有什么特别要注意的。

收货地址表结构

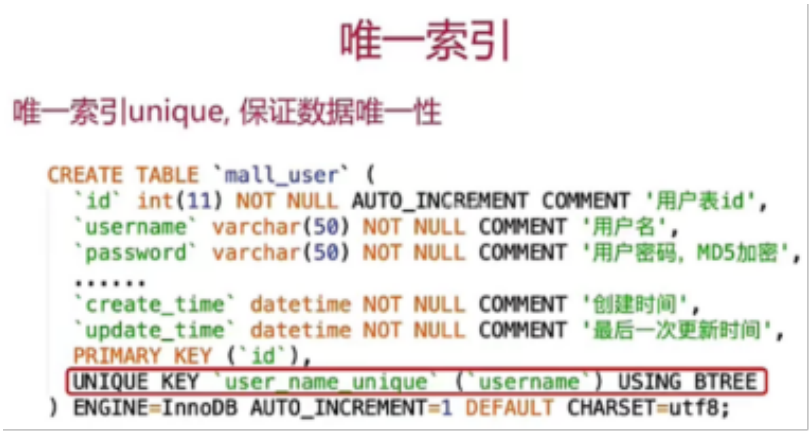
```
CREATE TABLE `mall_shipping` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) DEFAULT NULL COMMENT '用户id',
  `receiver_name` varchar(20) DEFAULT NULL COMMENT '收货姓名',
  `receiver_phone` varchar(20) DEFAULT NULL COMMENT '收货固定电话',
  `receiver_mobile` varchar(20) DEFAULT NULL COMMENT '收货移动电话',
  `receiver_province` varchar(20) DEFAULT NULL COMMENT '省份',
  `receiver_city` varchar(20) DEFAULT NULL COMMENT '城市',
  `receiver_district` varchar(20) DEFAULT NULL COMMENT '区/县',
  `receiver_address` varchar(200) DEFAULT NULL COMMENT '详细地址',
  `receiver_zip` varchar(6) DEFAULT NULL COMMENT '邮编',
  `create_time` datetime DEFAULT NULL,
  `update_time` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

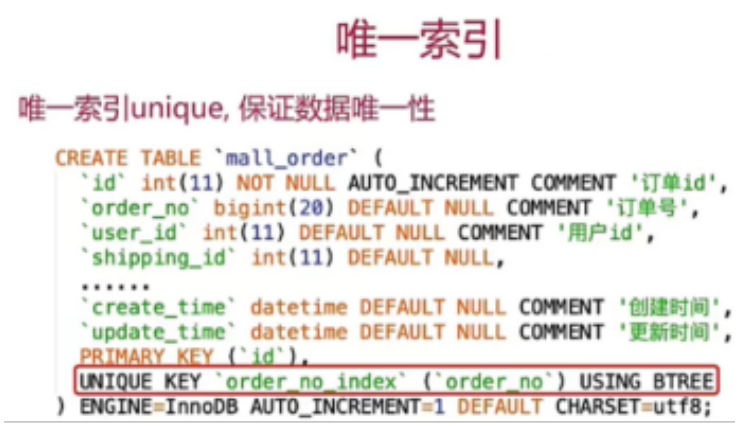
数据库的索引是怎么加的?

唯一索引。唯一它的作用就是保证数据的唯一性。

user 表，user 表这个username 我们就对它做了一个唯一索引用户名。在整张表里面是唯一的，不能重复设置了。唯一索引之后，往用户表里面写相同的用户名，他是写不进去的。

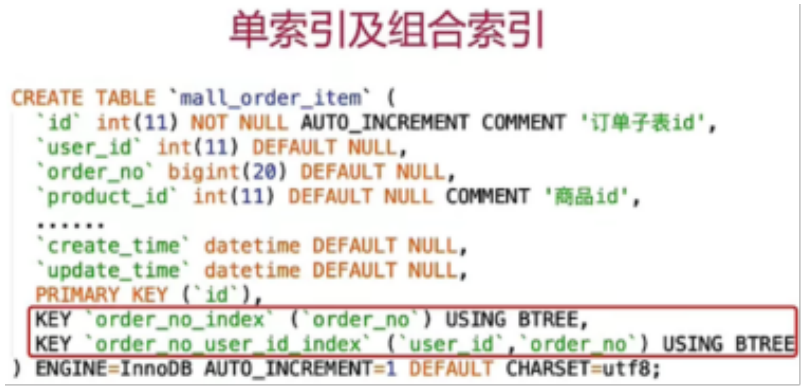
还有order表payinfo表也一样，订单号肯定是唯一的。这些我们知道它一定是唯一的数据，我们就在数据库里面设置唯一索引，避免数据重复。





在订单明细表里对ordernumber订单号字段做了一个索引。

那我怎么知道应该对哪些字段设置索引呢？我们知道索引是为了加快查询速度。应该想到来查这个表的时候会用什么字段来查？也就是 while 条件会写什么字段。目前我们会想到的就是用订单号来查.



MySQL 某条语句执行的好慢，访问数据库超时了，你怎么排查?

1. 根据故障时段，推断出故障是跟哪个功能有关。
2. 根据系统能在流量峰值过后自动恢复这一现象，排除是其他后台服务被大量请求打死的可能性（内存溢出、栈溢出或者进程直接挂掉）。然后可以把排查问题的重点

放到 MySQL 上。
3. 观察 MySQL 的 CPU 利用率，假如很高的话一般来说就是SQL导致的，去分析慢查询日志（慢 SQL 的日志中，会有这样一些信息：SQL、执行次数、执行时长。）找到可能的烂SQL。
4. 修改比较烂的SQL，或者适当做做缓存，再来看慢查询日志还有没有那个SQL，没有了说明改对了。
5. 再查看 MySQL的 CPU 利用率，如果还不正常就继续观察 CPU 利用率曲线变化规律，推断可能的问题比如缓存使用不当刷新得太慢或者其他原因，这个时候就想办法去对症下药了做针对性地预防和改进。

针对慢SQL，为了让整个系统更加健壮，不至于因为某一个小的失误，就导致全站无法访问，可以这样设计：

1. **上线一个定时监控和杀掉慢 SQL 的脚本**。这个脚本每分钟执行一次，检测上一分钟内，有没有执行时间超过一分钟（这个阈值可以根据实际情况调整）的慢 SQL，如果发现，直接杀掉这个会话。这样可以有效地避免一个慢 SQL 拖垮整个数据库的悲剧。即使出现慢 SQL，数据库也可以在至多 1 分钟内自动恢复，避免数据库长时间不可用。代价是，可能会有些功能，之前运行是正常的，这个脚本上线后，就会出现问题。但是，这个代价还是值得付出的，并且，可以反过来督促开发人员更加小心，避免写出慢 SQL。
2. **做某个模块相应的降级方案**。比如首页，做一个简单静态页面的首页，只要包含商品搜索栏、大的品类和其他顶级功能模块入口的链接就可以了。在 Nginx 上做一个策略，如果请求首页数据超时的时候，直接返回这个静态的首页作为替代。这样后续即使首页再出现任何的故障，也可以暂时降级，用静态首页替代。至少不会影响到用户使用其他功能。
3. **优化缓存置换策略。**

前两个方法容易实施，不需要对系统做很大的改造，并且效果也立竿见影。

那假如找到慢的 SQL 了怎么优化呢?

第一：**索引优化**。使用索引避免全表扫描，毕竟SQL执行速度的快慢关键还是语句需要扫描数据的行数，同时尽量只获取需要的列。避免索引失效，特定业务 可以设置联合索引让需要查询返回的列都在索引中避免回表操作。

第二：**排序优化。**排序也是可能完成慢SQL的因素，尤其是数据量大，需要使用外部排序的时候又可以与磁盘IO性能扯上关系等，常见的问题还有limit m,n m很大又无法使用索引的时候，可以通过查询排序数据里面最小的。

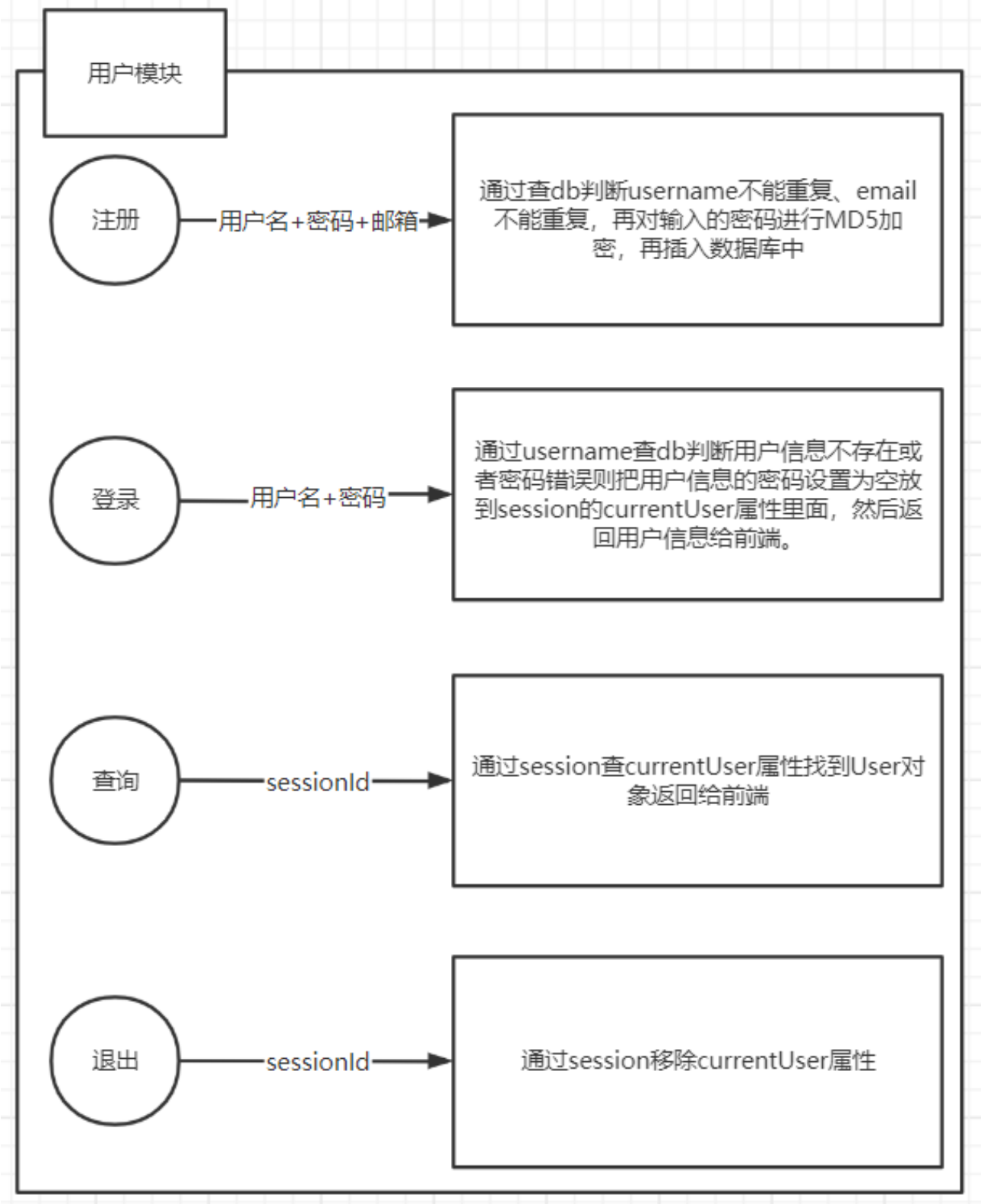
第三：**join优化**。多表联合查询的时候，尽量使用小表驱动大表。

第四：**避免大事务**。将大事务复杂的SQL，拆分成多个小的SQL单个表执行，获取的结果在程序中进行封装，尽量减小事务粒度，减少锁表的时间。

最后假如 SQL 本身没问题了，但是还是慢，那就可能是数据库自己慢导致的 SQL 慢了。首先看看数据库参数设置有没有问题，其次还有数据量到达一定规模后，单机性能容易受限导致数据库响应慢；读写分离，从库提供读服务，错误的认为从库只需要提供查询服务采用了达不到性能指标的机器，其实是主库承受的数据更新压力，从库一个不落的都要承受，还要更多的提供查询服务。

用户

用户管理是怎么实现的?



因为HTTP协议是无状态无连接的协议，服务端对于客户端每次发送的请求都认为它是一个新的请求，上一次会话和下一次会话是没有联系的。因为它无法保存登录状态，所以从协议本身来说，它不适合用来做会话管理。

因此，我们会使用一个上层应用去实现我们的会话管理功能。这个应用可以在切换页面时保持登录状态，并且对用户是透明的，这样就使得我们能在短时间内再次访问一个登录过的页面，就会保持登录状态。

我们采用的是基于session的认证：

登录的时候首先根据用户名查询用户，如果查不到或者查询出来密码比对发现错误就返回错误；查到了密码还正确就是登录成功，把用户数据信息在密码清空后再放到session的currentUser属性里，也放用户信息到响应体里返回。退出登录的时候再清空。

Web应用服务器会给用户配置一个sessionid，并将它存储在服务器内存中，之后再把这个sessionid发送给用户。

下次在进行需要登录的操作时前端发送请求的时候在请求头里带上sessionId，服务器的拦截器拦截判断根据这个session里面是否有用户信息，有说明已经登录过了；没有则返回错误。

有没有可能攻击者盗用sessionid绕过身份认证？假如说浏览器禁用cookie，sessionid只能重写在URL中，攻击者发起会话固定攻击怎么办？

攻击者首先访问一个需要登录的网站，获取到 Web 应用返回的 sessionid 信息。由于攻击者没有账户密码，所以只能通过发送一个诱骗信息url上带着这个sessionid给受害者，使得受害者用这个 sessionid 实现登录操作。这样攻击者的 sessionid 就通过了验证，使得攻击者再次用这个 sessionid 信息访问被攻击网站时，可以直接通过保持登录的认证。

解决：

只要在用户登录时重置 session(session.invalidate()方法)，然后把登录信息保存到新的session中即可。

为了安全考虑，Web应用通常会给 sessionid 设置一个过期时间，使得 sessionid 仅在某个时间段内有效。

浏览器突然关闭了，你又如何让用户自动退出呢？

\1. 浏览器关闭的时候，发送请求到后台调用退出接口，根据sessionid来删除当前会话的用户信息或者把cookie给干死，session没有对应的sessionid等于废了。

\2. 设置session的有效期比如30分钟。写一个监听类，session超时的时候会自动删除session对应的用户数据；

怎么保证用户密码的安全?

每个用户需要一个盐值，密码要加一个盐值再进行加密。

这样相同的密码可以生成不同的hash值，这样可以加大黑客的力度。或者加密算法MD5可以换成SHA安全hash算法，安全等级更高，但是加密速度更慢。

怎么防止别人用脚本批量注册刷掉网站?

可以加一个通过邮件、短信向用户发送验证码的环节，验证码存在redis里，用户输入验证码和redis里的判断，成功才能注册。

对于短信验证码这种开放接口，程序逻辑内需要有防刷逻辑。好的防刷逻辑是，对正常使用的用户毫无影响，只有疑似异常使用的用户才会感受到。对于短信验证码，有如下 4 种可行的方式来防刷。

第一种方式，控制相同手机号的发送次数和发送频次。

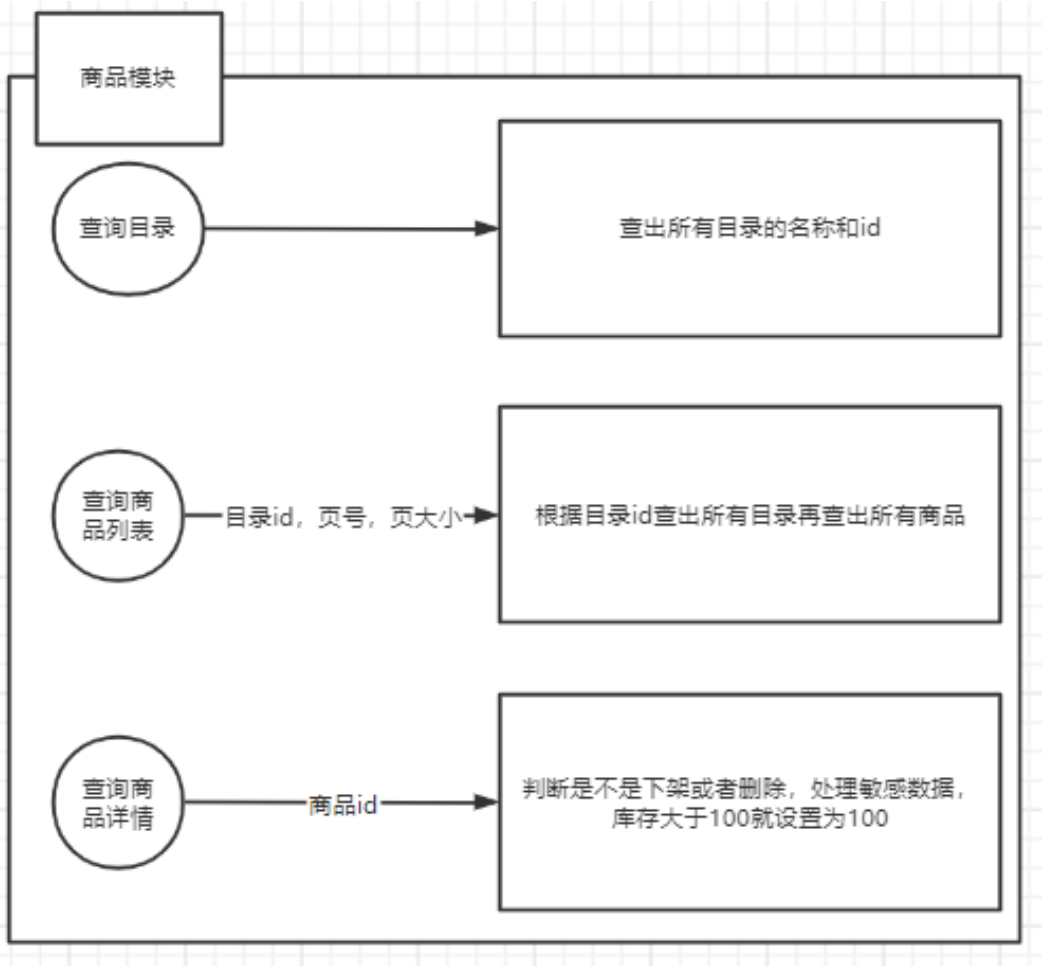
第二种方式，增加前置图形验证码。

当多点部署服务器的时候，你在服务器a上登录，服务器b如何得到用户登录信息？如果每个服务器都有该用户的session，如果有很大的用户量，假如500万用户是每个服务器存了8g的session信息，如果再来500万，岂不是各个服务器均要扩容到16G，即每个服务器都需扩容容量，怎么优化？如果在多个实例之间处理同一个用户的登录状态?

JWT、分布式session

商品

商品管理是怎么实现的?



商品信息假如字段越来越多，而且流量上来了怎么办?

无法一个系统解决，需要分而治之

从存储层面，数据区分为：固定结构数据、非固定结构数据、富媒体数据；

从读取层面，将数据分为：经常变化数据、非经常变化数据；

\1. 从数据存储到哪的角度：

- 固定结构数据：

商品主标题、副标题、价格，等商品最基本、最主要的信息（任何商品都有的属性）

存储到：MySQL中

- 非固定结构数据：

商品参数，不同类型的商品，参数基本完全不一样。电脑的内存大小、手机的屏幕尺寸、酒的度数、口红的色号等等。

存储到不需要固定结构的存储：MongoDB或者MySQL的json字段中

- 富媒体数据：

商品的主图、详情介绍图片、视频等富媒体数据

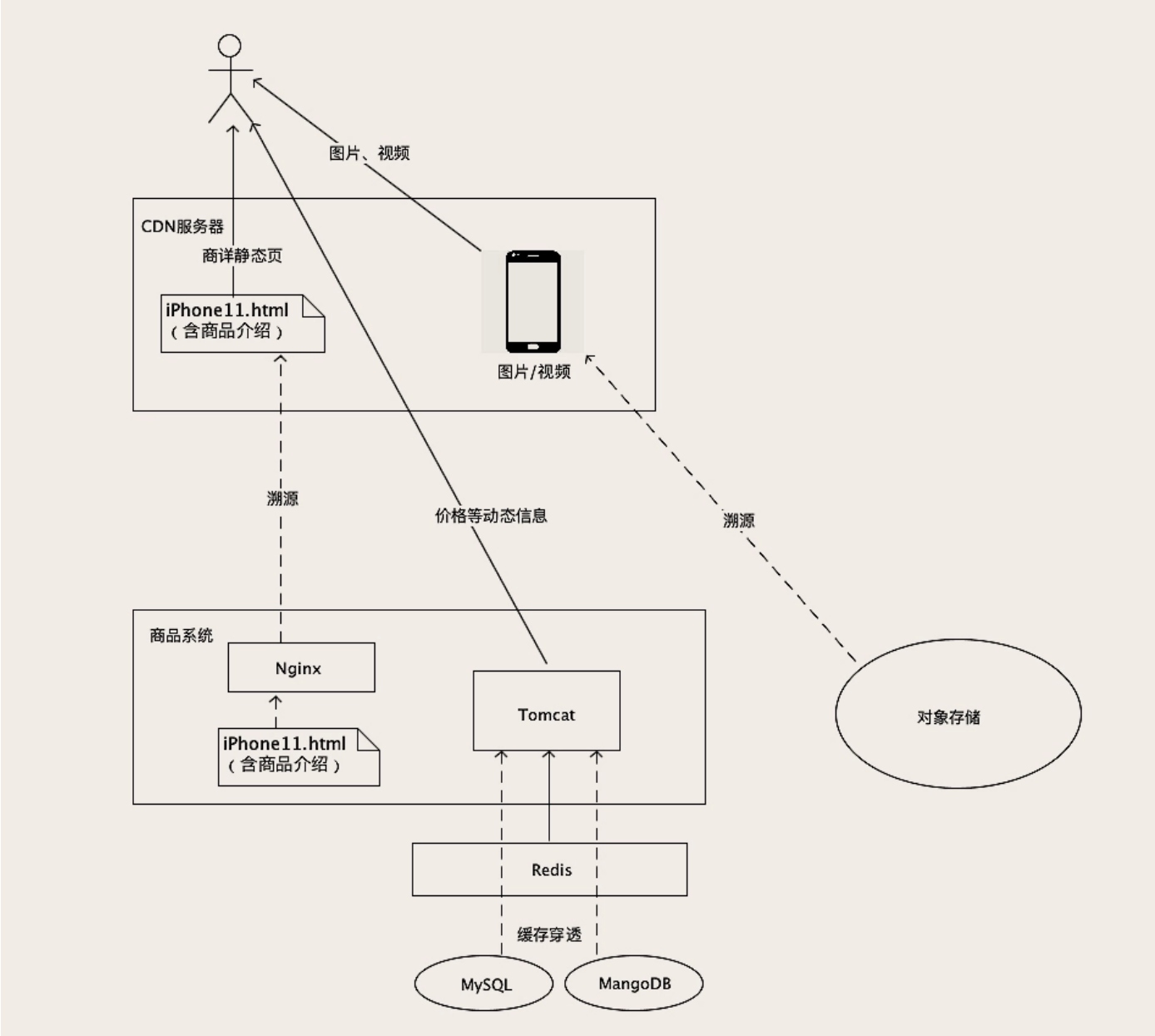
存储到：对象存储。并且通过客户端直接调用对象存储的API，得到媒体资源在对象存储中的ID或者URL之后，将ID或者URL提交到MySQL中。对象存储自带CDN加速服务，响应时间快。

\2. 从数据读取角度：

- 存储到MySQL中的数据，需要设计一层缓存层比如Redis，应对高并发读

- 对于不经常变动的数据：

可以交给前端静态化加速处理



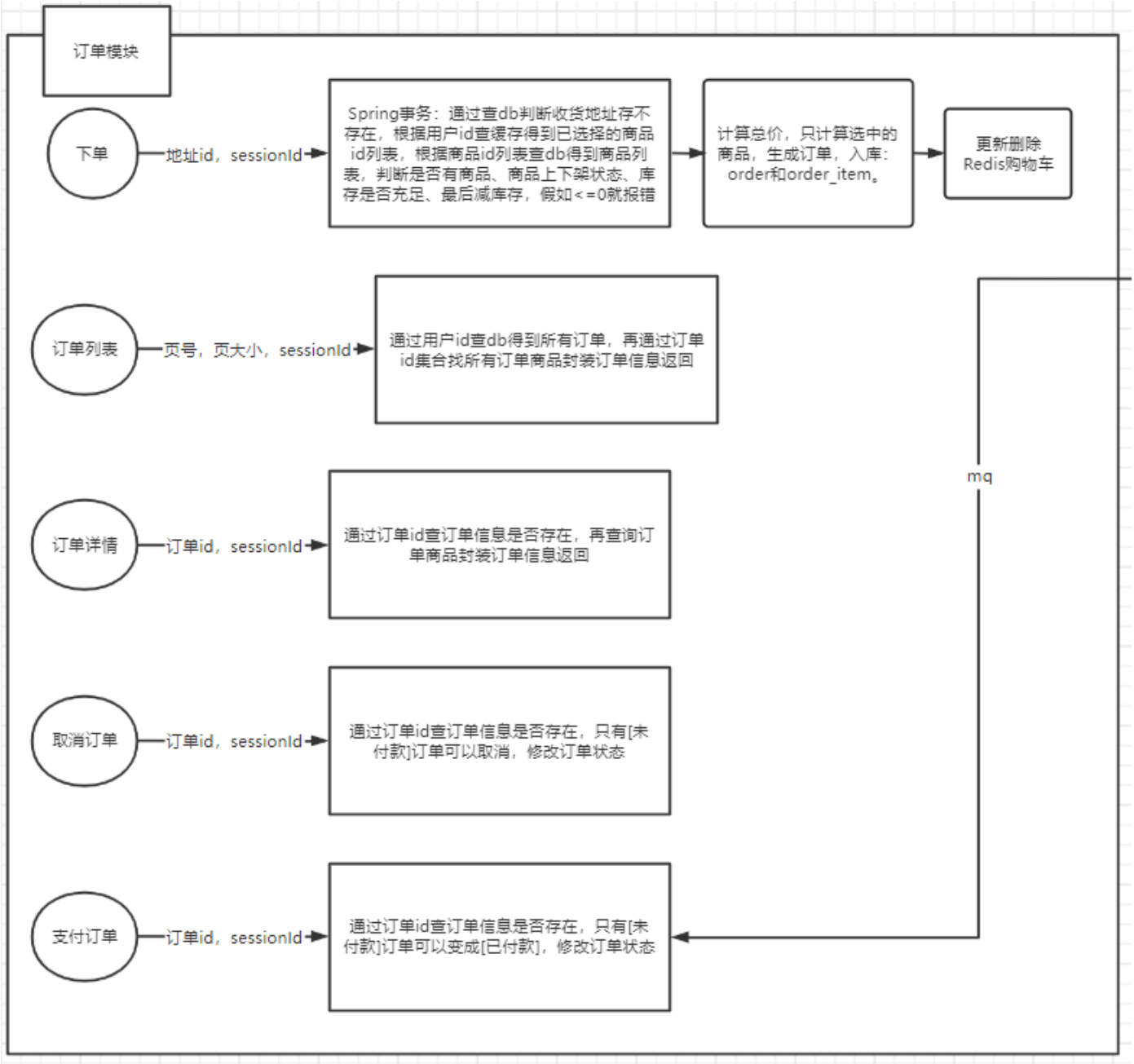
数据量最大的图片、视频和商品介绍都是从离用户最近的 CDN 服务商获取的，速度快，节约带宽。真正打到商品系统的请求，就是价格这些需要动态获取的商品信息，一般做一次 Redis 查询就可以了，基本不会有流量打到 MySQL 中。这样一个商品系统的存储的架构，把大部分请求都转移到了又便宜速度又快的 CDN 服务器上，可以用很少量的服务器和带宽资源，抗住大量的并发请求。

（MongoDB 最大的特点就是，它的“表结构”是不需要事先定义的，其实，在 MongoDB 中根本没有表结构。由于没有表结构，它支持你把任意数据都放在同一张表里，你甚至可以在一张表里保存商品数据、订单数据等这些结构完全不同的数据。并且，还能支持按照数据的某个字段进行查询。它是怎么做到的呢？MongoDB 中的每一行数据，在存储层就是简单地被转化成 BSON 格式后存起来，这个 BSON 就是一种更紧凑的 JSON。所以，即使在同一张表里面，它每一行数据的结构都可以是不一样的。当然，这样的灵活性也是有代价的，MongoDB 不支持 SQL，多表联查和复杂事务比较孱弱，不太适合

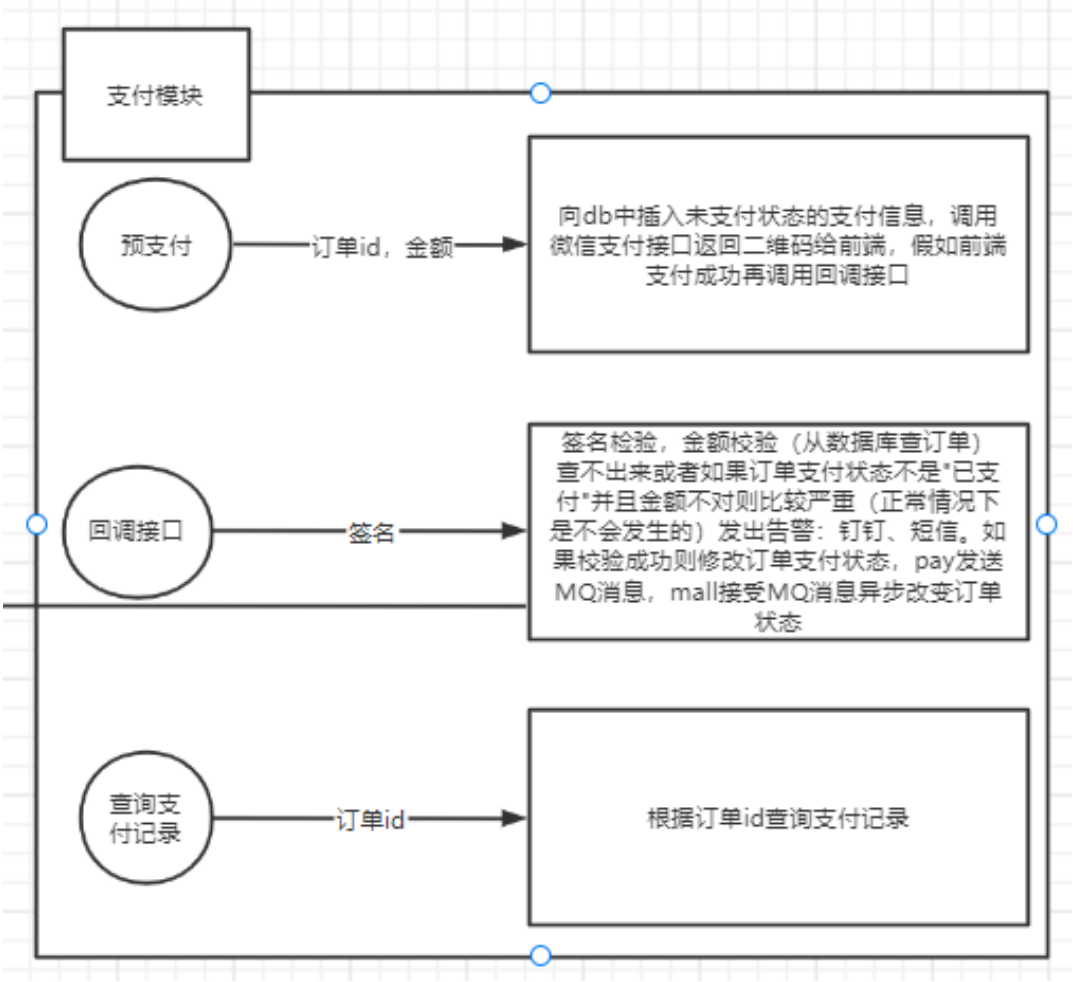
存储一般的数据。但是，对于商品参数信息，数据量大、数据结构不统一，这些 MongoDB 都可以很好的满足。我们也不需要事务和多表联查，MongoDB 简直就是为了保存商品参数量身定制的一样。MySQL最新支持json了所以其实不用MongoDB也行了）

订单-支付

订单管理是怎么实现的?



支付系统是怎么实现的?



为什么支付系统要独立出来?

把业务和支付解耦开来。今天电商系统需要支付，明天活动系统需要支付，你只需要跳转到支付系统就可以发起支付。在新增的业务系统的情况下，支付系统不需要动一行的代码，这就是解耦的优势。

怎么防止重复下单?

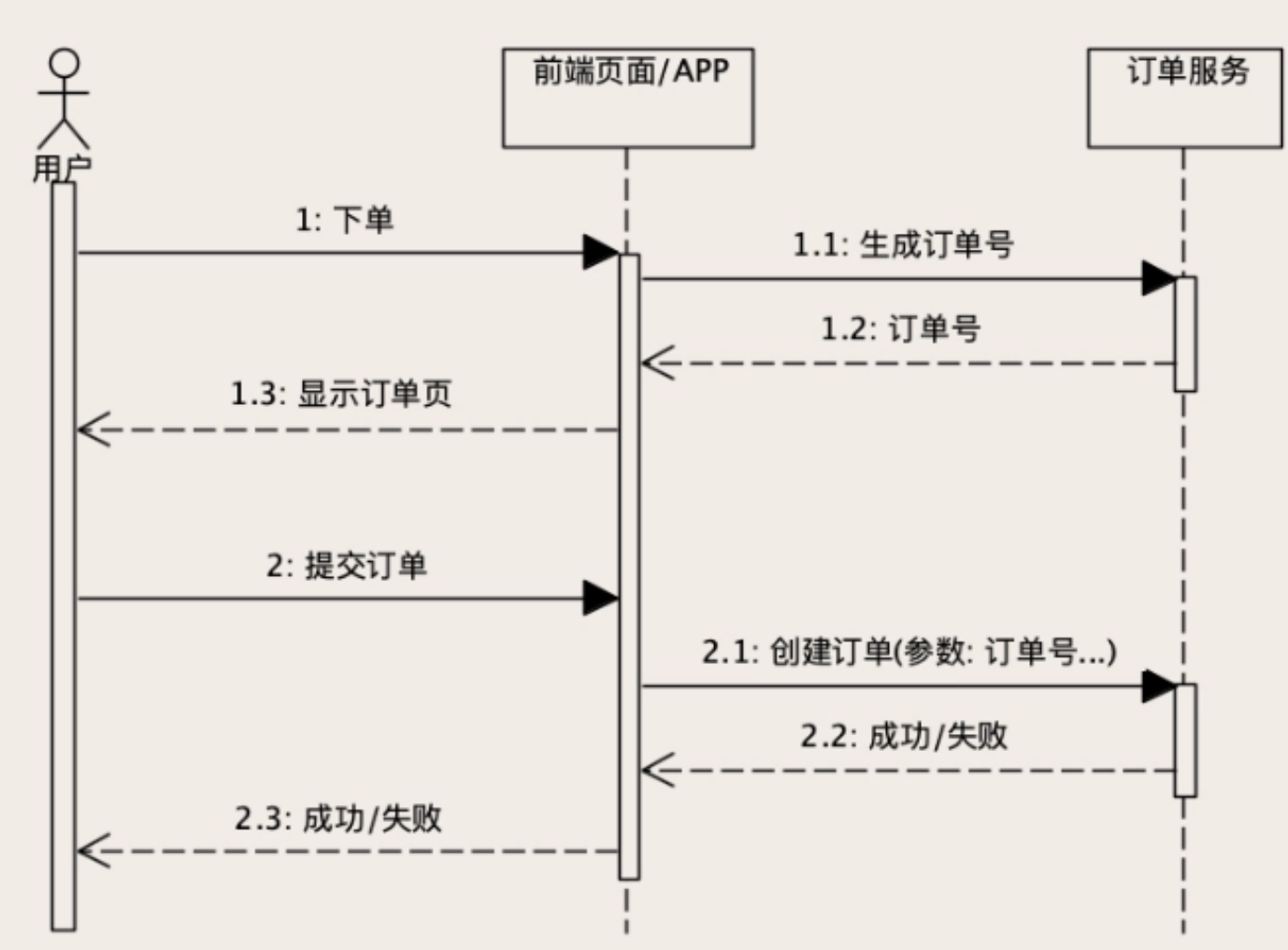
假如说，用户点击“创建订单”的按钮时手一抖，点了两下，浏览器发了两个 HTTP 请求。

首先前端页面上应该防止用户重复提交表单，不过这个不可靠，因为网络错误会导致重传，很多非用户操作的部分都会有自动重试机制。

解决办法是，让你的订单服务具备幂等性：

- 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。



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。

唯一索引这个是兜底的方案，一般配合上层的幂等。

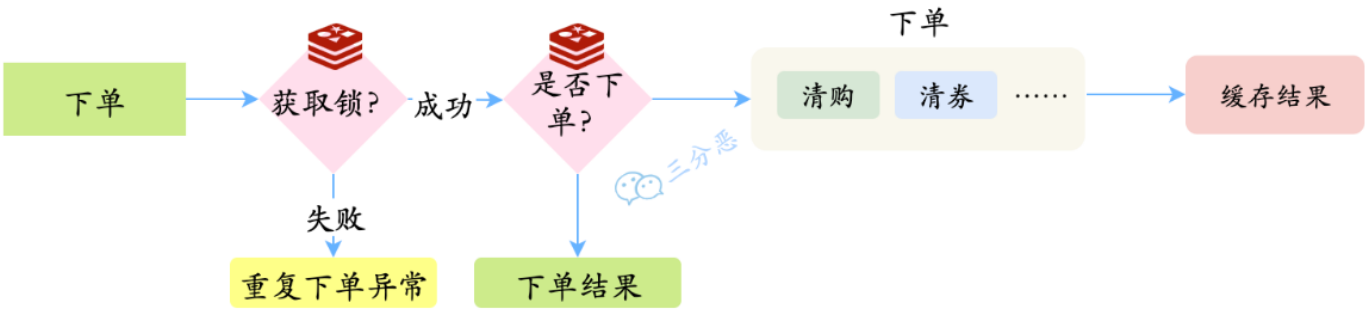
- 上层可以配合利用Redis防重

另外一个办法，就是下单请求的时候要加锁了，通常我们的服务都是集群部署，所以一般都是用Redis实现分布式锁。

大概的逻辑：

就是以requestId为维度，进行加锁，如果获取锁失败，就抛一个自定义的重复下单异常。

如果获取到锁，先check一下，是否已经下单，为了提高性能，下单完成后，也把下单的结果放在Redis缓存　　里。



- 前端接口也可以做一下防重，但是不可靠。

如何解决客户的恶意下单问题?

封IP，nginx中有一个设置，单个IP访问频率和次数多了之后有一个拉黑操作。

如果用户支付成功，但是看到订单的状态还是未支付，于是又去支付了一次，会重复支付吗?

用户再次发起支付请求的时候，支付系统会带着要支付的订单的ID来生成支付信息入库，支付信息中的订单ID是加了唯一索引的，只要是同一个订单是没办法进行重复支付的。

订单id是怎么生成的？怎么保证唯一性?

如果单纯是生成全局唯一ID方法有很多，比如小规模系统完全可以用MySQL的sequence表，可以每次请求从sequence表里拿一组id，放到内存，用完了再拿。或者Redis单线程原子递增操作来生成。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全局唯一ID。也可以用uuid，不过容易导致数据库页分离，这个也可以解决，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，这样表空间还是紧密的，其他关联订单表仍然以订单号作外键，不过不用订单号作主键的话导致的就是多走一次索引，而且还会影响插入更新性能，各有优劣。

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。

同时还得尽可能的短。

用户下单这个时刻，正好赶上商品调价怎么处理?

假如说商品可以单独下单的话。首先，商品系统需要保存包含价格的商品基本信息的历史数据，对每一次变更记录一个自增的版本号。在下单的请求中，不仅要带上SKUID，还要带上版本号。订单服务以请求中的商品版本对应的价格来创建订单，就可以避免“下单时突然变价”的问题了。 但是，这样改正之后会产生一个很严重的系统漏洞：黑客有可能会利用这个机制，以最便宜的历史价格来下单。所以，我们在下单之前需要增加一个检测逻辑：请求中的版本号只能是当前版本或者上一个版本，并且使用上一个版本要有一个时间限制，比如说调价5秒之后，就不再接受上一个版本的请求。这样就可以避免这个调价漏洞了。

对于我们系统下单前必须先把商品加入购物车，所以如果购物车看到一个价格，下单时会重新检查当前价格和最新价格，如果发生变更，一般会在下单页面给出提醒，重新刷新当前页面来解决。

假如支付完了微信一直不回调怎么办?

起定时任务或者延迟消息，对待支付订单主动查询支付状态进行补偿。

怎么防止超卖?

使用了MySQL的事务，默认的隔离级别，因为读是快照读MVCC，所以加了排它锁来当前读，然后事务提交锁才会释放，通过悲观锁解决的超卖问题。同时对死锁问题进行了优化，加锁是按照商品id排序之后加锁的。

可以将库存放到Redis，然后利用Lua原子性地检查库存之后再更新扣减。

减库存成功了，但是生成订单失败了，该怎办?

非分布式的系统中使用Spring提供的事务功能即可。

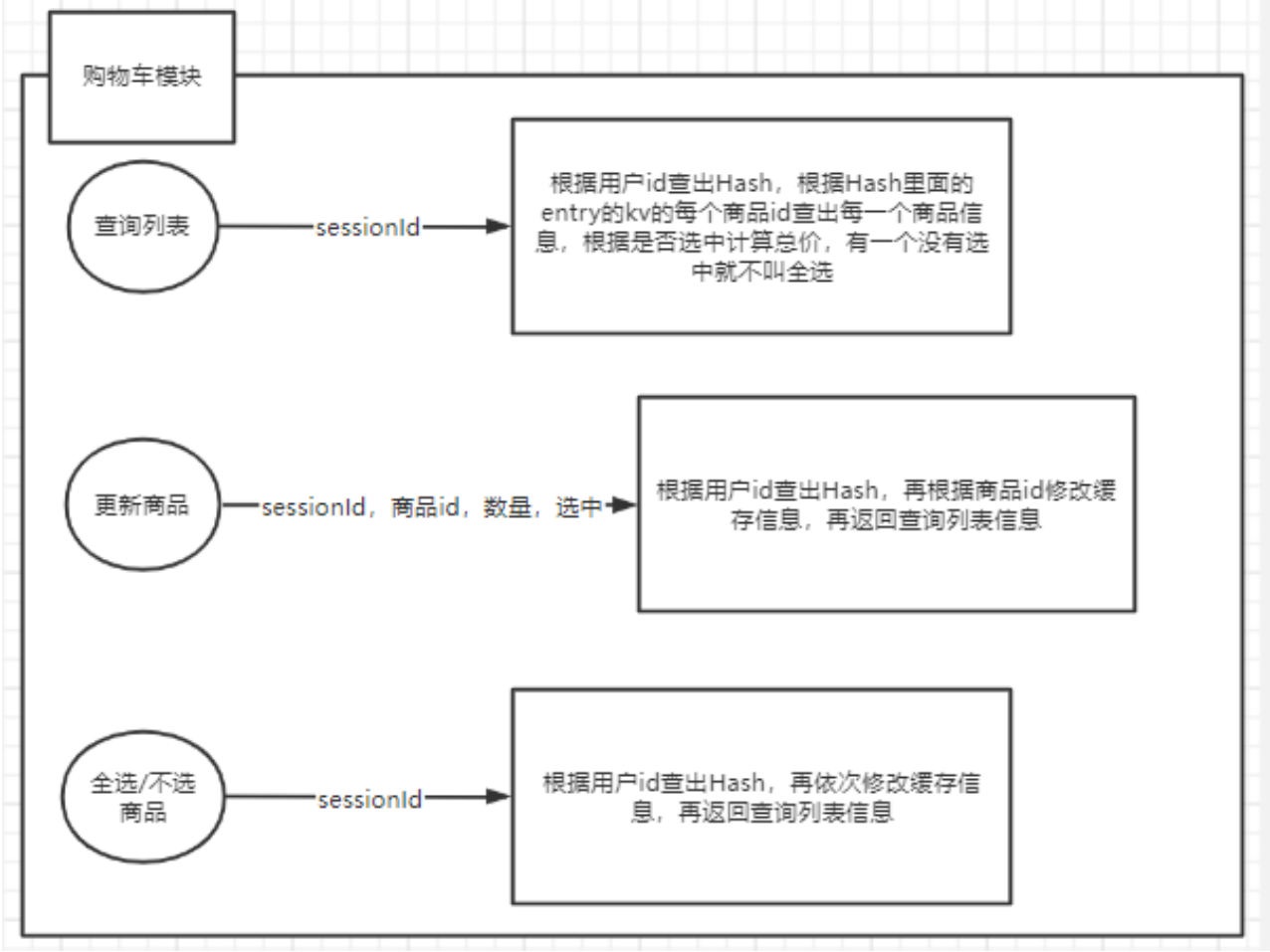
缓存

购物车管理是怎么实现的?

在项目中基于缓存Redis实现了高性能购物车。

缓存的数据类型用的是hash，首先redis的key是用户id，value是hash，hash里面的key是商品id，value是商品id、商品在购物车中的数量、商品是否选中的结构体的Json串。

<用户id,<商品id,<商品id,商品num,商品selected>>>。



为什么用Redis不直接用MySQL？用Redis丢数据了怎么办?

因为购物车是写多读多的场景，可以用Redis的读写缓存模式来实现，读写都基于redis来操作能减少IO次数加快数据操作的性能，提升用户体验，同时还能减少对Mysql数据库的压力。

如果说真的怕丢数据的话要也可以用MySQL给Redis提供一个备份功能，主存储还是用Redis，比如写完Redis之后通过MQ异步备份到MySQL，这样MySQL就起到一个备份的作用，同时MySQL备份了数据假如Redis挂了MySQL可以提供降级服务保证可用性。

Redis挂了怎么办?

如果项目中的Redis挂掉，如何减轻数据库的压力？（滴滴）（华为）

组redis集群，主从模式、哨兵模式、集群模式。

主从模式中：如果主机宕机，使用slave of no one 断开主从关系并且把从机升级为主机。

哨兵模式中：自动监控master / slave的运行状态，基本原理是：**心跳机制+投票裁决。**

每个sentinel会向其它sentinel、master、slave定时发送消息（哨兵定期给主或者从和slave发送ping包，正常则响应pong，**ping和pong就叫心跳机制**），以确认对方是否“活”着，如果发现对方在指定时间内未回应，则暂时主观认为对方已挂。

若master被判断死亡之后，通过**选举算法**，从剩下的slave节点中选一台升级为master。并自动修改相关配置。

MQ

为什么要使用消息队列?

解耦

A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来？头发都白了啊！

如果使用MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。

通过一个MQ，Pub/Sub 发布订阅消息这么一个模型，A 系统就跟其它系统彻底解耦了。

支付系统模块，调用了电商系统模块，互相之间的调用有可能以后增加了业务之后会很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用MQ给它异步化解耦，也是可以的。

异步

一般互联网类的企业，对于用户直接的操作，一般要求是每个请求都必须在短时间内完成，这样对用户才是无感知的。

如果使用 MQ，那么支付系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms，对于用户而言，其实感觉上就是点个按钮，8ms 以后就直接返回了，爽！网站做得真好，真快！

异步化提升用户体验，同时消息队列能保证消息的可靠；并且对比同步的方式，同步的方式会卡住用户，还可能有比如浏览器不小心关闭或者用户关闭，导致页面跳转不了也就无法修改订单数据。

削峰

每天 0:00 到 12:00，A 系统风平浪静，每秒并发请求数量就 50 个。结果每次一到 12:00 ~ 13:00 ，每秒并发请求数量突然会暴增到 5k+ 条。但是系统是直接基于 MySQL 的，大量的请求涌入 MySQL，每秒钟对 MySQL 执行约 5k 条 SQL。

一般的 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。

但是高峰期一过，到了下午的时候，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。

如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在中午高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。

这个短暂的高峰期积压是 ok 的，因为高峰期过了之后，每秒钟就 50 个请求进 MQ，但是 A 系统依然会按照每秒 2k 个请求的速度在处理。所以说，只要高峰期一过，A 系统就会快速将积压的消息给解决掉。

系统使用了消息队列，我们就能够对流量更好的把控。简单来说就是能够扛住高并发大流量。

Mq怎么选型的?

我们主要调研了几个主流的mq，kafka、rabbitmq、rocketmq

选型我们主要基于以下几个点去考虑：

1、由于我们系统的qps压力比较大，所以性能是首要考虑的要素。

Ka和Ro都是ms级别，Ra是us级别。

2、开发语言，由于我们的开发语言是java，主要是为了方便二次开发。

Ka是Scala和Ro都是Java，Ra是Erlang

3、对于高并发的业务场景是必须的，所以需要支持分布式架构的设计。

Ka和Ro都支持分布式架构，Ra是主从架构

4、功能全面，由于不同的业务场景，可能会用到顺序消息、事务消息等。

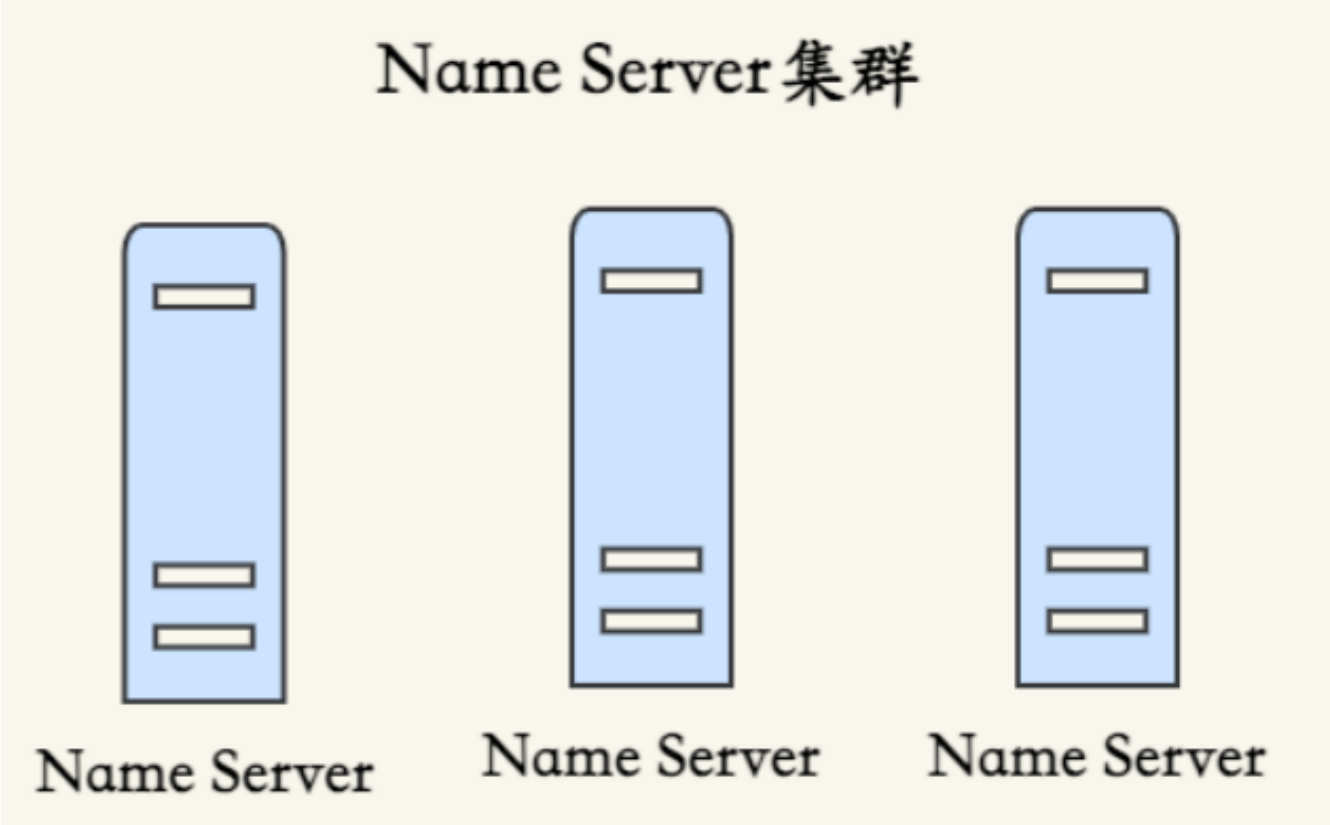
Ka只支持主要Mq功能比较单一，Ro还支持顺序、事务消息等

我们的系统是面向用户的C端系统，未来可能具有一定的并发量，对性能也有比较高的要求，所以选择了低延迟、吞吐量比较高，可用性比较好的RocketMQ。

如何保证消息队列的高可用?

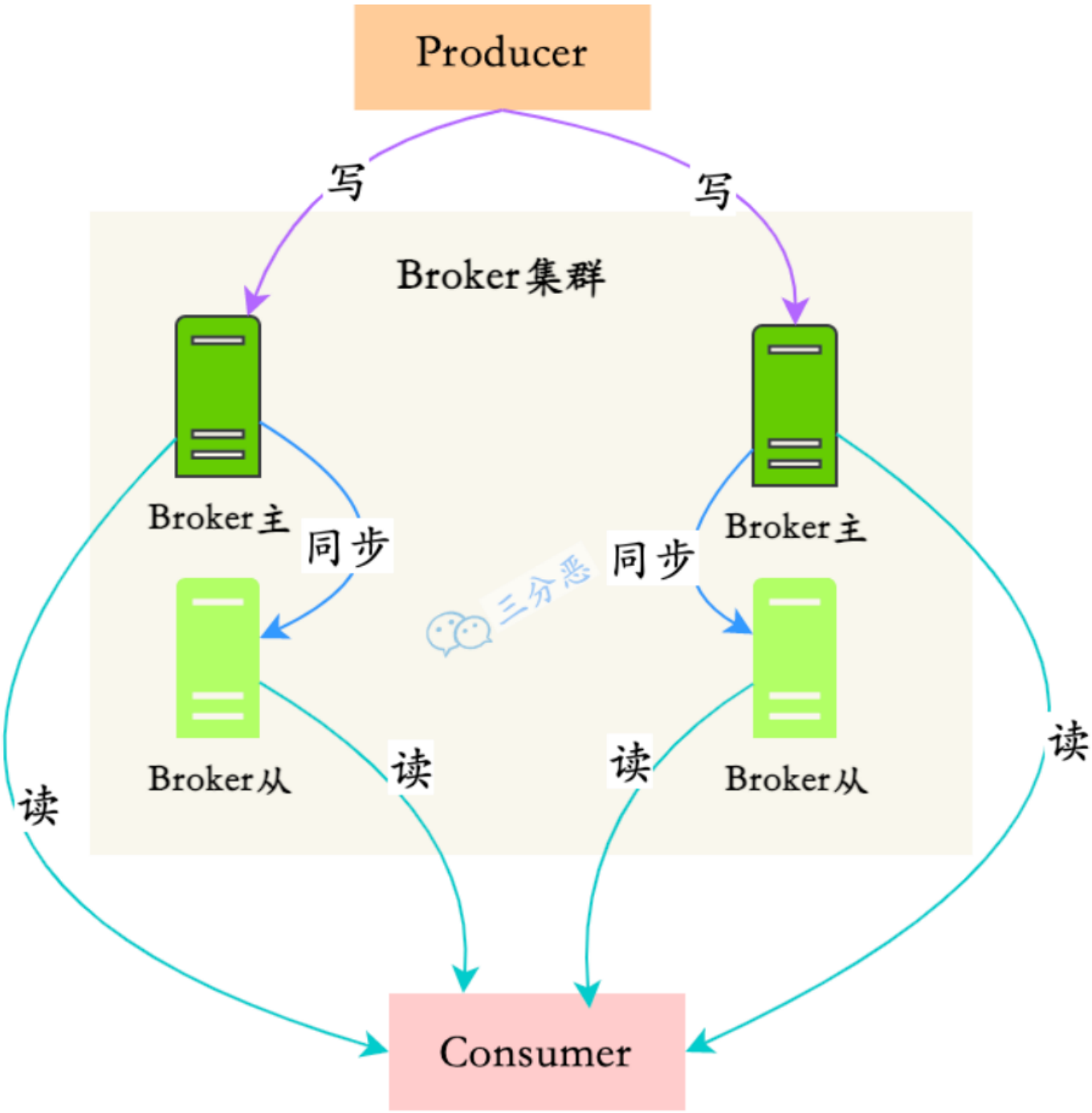
NameServer高可用

NameServer因为是无状态，且不相互通信的，所以只要集群部署就可以保证高可用。



Broker高可用****

RocketMQ的高可用主要是在体现在**Broker的读和写的高可用**，Broker的高可用是通过集群 和 主从 实现的。



Broker可以配置两种角色：Master和Slave，Master角色的Broker支持读和写，Slave角色的Broker只支持读，Master会向Slave同步消息。也就是说Producer只能向Master角色的Broker写入消息，Cosumer可以从Master和Slave角色的Broker读取消息。

读的高可用：

Consumer 的配置文件中，并不需要设置是从 Master 读还是从 Slave读，当 Master 不可用或者繁忙的时候， Consumer 的读请求会被自动切换到从 Slave。有了自动切换 Consumer 这种机制，当一个 Master 角色的机器出现故障后，Consumer 仍然可以从 Slave 读取消息，不影响 Consumer 读取消息，这就实现了读的高可用。

写的高可用：

如何达到发送端写的高可用性呢？在创建 Topic 的时候，把 Topic 的多个 MessageQueue 创建在多个 Broker 组上（相同 Broker 名称，不同 brokerId 机器组成 Broker组），这样当 Broker 组的 Master 不可用后，其他组Master 仍然可用，Producer 仍然可以发送消息 RocketMQ。

如果机器资源不足，需要把 Slave 转成 Master ，

旧版本：人工切换

人工切换需要改配置把Slave配置成Master，重启MQ，会造成这段时间的MQ不可用，可用性比较差。

新版本：自动切换

Dledger机制可以实现主从自动切换，一旦Master挂掉，多个Slave可以使用Dledger机制重新选举出一个Master这就实现了写的高可用。

主从数据不一致咋整？其实引入raft之后从就不用于读了，只是起到一个备份的作用，读写都集中在主节点。

Raft选举：当Master 挂掉后，Slave 们会进行选举，具体逻辑大概是： Slave 都先投自己，发现平票，大家随机休眠，先唤醒的那个节点直接投自己，并传播给其他节点，其他节点收到投票后，会进行是否可以接受它的投票，判断可以则给他也投一票，判断不可以就拒绝，给自己投票！（具体的判断逻辑为: 对方的同步进度是否比自己晚；投票轮次是否为同一次等）

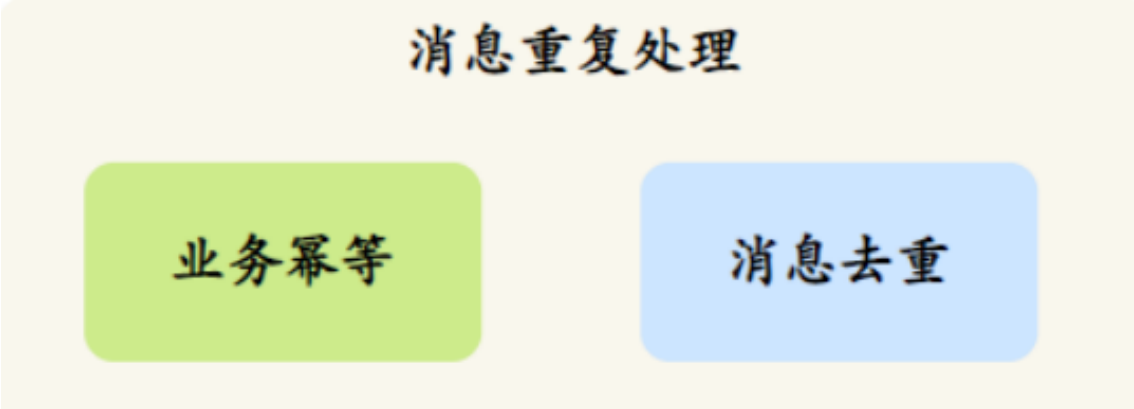
生产者、消费者高可用

集群部署，这样保证一台机器挂了，还有其他机器可以继续工作。

如何处理消息重复?

对分布式消息队列来说，同时做到确保一定投递和不重复投递是很难的，就是所谓的“有且仅有一次”。 RocketMQ择了确保一定投递，保证消息不丢失，但有可能造成消息重复。 比如消费者消费消息时本地事务执行但是提交offset时宕机导致mq给消费者重复推送消息；再比如生产者发送消息到mq时发送成功未获取到响应然后进行消息发送重试导致消息发送多次。

处理消息重复问题，主要由业务端自己保证，主要的方式有两种：业务幂等和消息去重。



业务幂等：第一种是保证消费逻辑的幂等性，也就是多次调用和一次调用的效果是一样的。这样一来，不管消息消费多少次，对业务都没有影响。

比如你的业务逻辑就是写Redis，那没问题了，反正每次都是 set，天然幂等性。

再比如我们项目电商系统消费mq消息修改订单状态，只有未支付才能变成已支付，配合数据库的悲观锁就天然具有业务幂等，假如订单状态不是未支付，就没有办法更新订单状态了，直接抛异常。

消息去重：第二种是业务端，对重复的消息就不再消费了。这种方法，需要保证每条消息都有一个惟一的编号，通常是业务相关的，比如订单号，消费的记录需要落库，而且需要保证和消息确认这一步的原子性。

比如可以建立一个消费记录表，拿到这个消息做数据库的insert操作。给这个消息做一个唯一主键或者唯一约束，那么就算出现重复消费的情况，就会导致主键冲突，那么就不再处理这条消息。

再比如让生产者发送每条数据的时候，里面加一个全局唯一的id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那就别处理了，保证别重复处理相同的消息。

假如是插入操作，兜底的方案比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

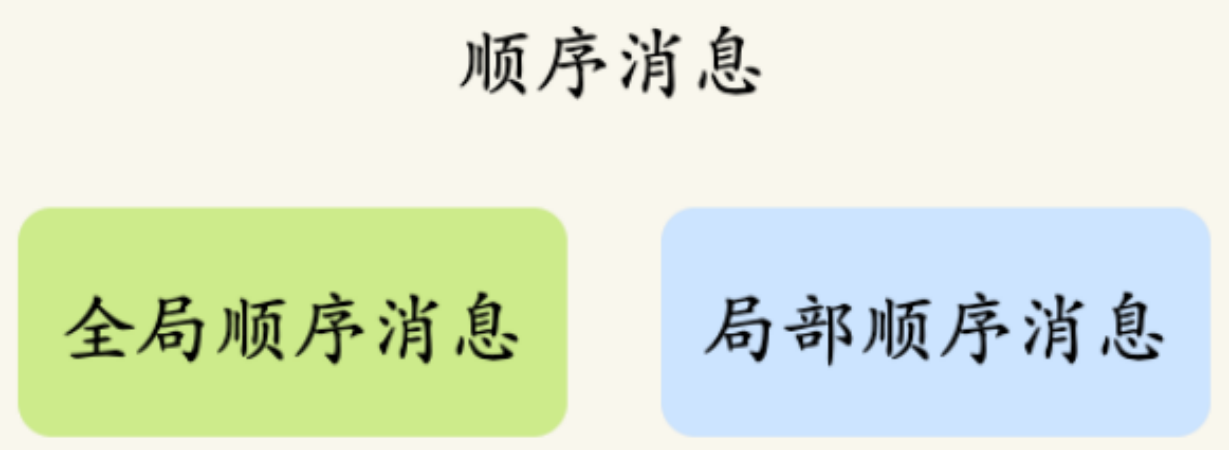
如何保证消息的顺序性?

为什么基于**MQ来传输数据会出现消息乱序？**

我们原本有顺序的消息，完全可能会分发到不同的MessageQueue中去，然后不同机器上部署的Consumer可能会用混乱的顺序从不同的MessageQueue里获取消息然后处理。

顺序消息

顺序消息是指消息的消费顺序和产生顺序相同



顺序消息分为全局顺序消息和部分顺序消息，全局顺序消息指某个 Topic 下的所有消息都要保证顺序；

部分顺序消息只要保证每一组消息被顺序消费即可，比如订单消息，只要保证同一个订单 ID 个消息能按顺序消费即可。

部分顺序消息

部分顺序消息相对比较好实现，生产端需要做到把同 ID 的消息发送到同一个 Message Queue ； 在消费过程中，要做到从同一个Message Queue读取的消息顺序处理——消费端不能并发处理顺序消息，这样才能达到部分有序。

全局顺序消息

RocketMQ 默认情况下不保证顺序，比如创建一个 Topic ，默认八个写队列，八个读队列，这时候一条消息可能被写入任意一个队列里；在数据的读取过程中，可能有 多个 Consumer ，每个 Consumer 也可能启动多个线程并行处理，所以消息被哪个 Consumer 消费，被消费的顺序和写人的顺序是否一致是不确定的。要保证全局顺序消息， 需要先把 Topic 的读写队列数设置为 一，然后Producer Consumer 的并发设置，也要是一。简单来说，为了保证整个 Topic全局消息有序，

只能消除所有的并发处理，各部分都设置成单线程处理 ，这时候就完全牺牲

RocketMQ的高并发、高吞吐的特性了。

另外一点是消费失败，不能直接进重试队列，消费失败需要等待一会重新消费，会导致后面的消息阻塞。

一. 读写队列，是在路由时使用

在消息发送时，根据写队列个数返回路由信息，而消息消费时按照读队列个数返回路由信息。

二. 在物理文件层面，只有写队列才会创建文件

举个例子：写队列个数是8，设置的读队列个数是4. 这个时候，会创建8个文件夹，代表0 1 2 3 4 5 6 7，但在消息消费时，路由信息只返回4，在具体拉取消息时，就只会消费0 1 2 3这4个队列中的消息，4 5 6 7中的信息压根就不会被消费。

反过来，如果写队列个数是4，读队列个数是8，在生产消息时只会往0 1 2 3中生产消息，消费消息时则会从0 1 2 3 4 5 6 7所有的队列中消费，当然 4 5 6 7中压根就没有消息 ，假设ConsumerGroup有两个消费者，事实上只有第一个消费者在真正的消费消息(0 1 2 3)，第二个消费者压根就消费不到消息，空转。

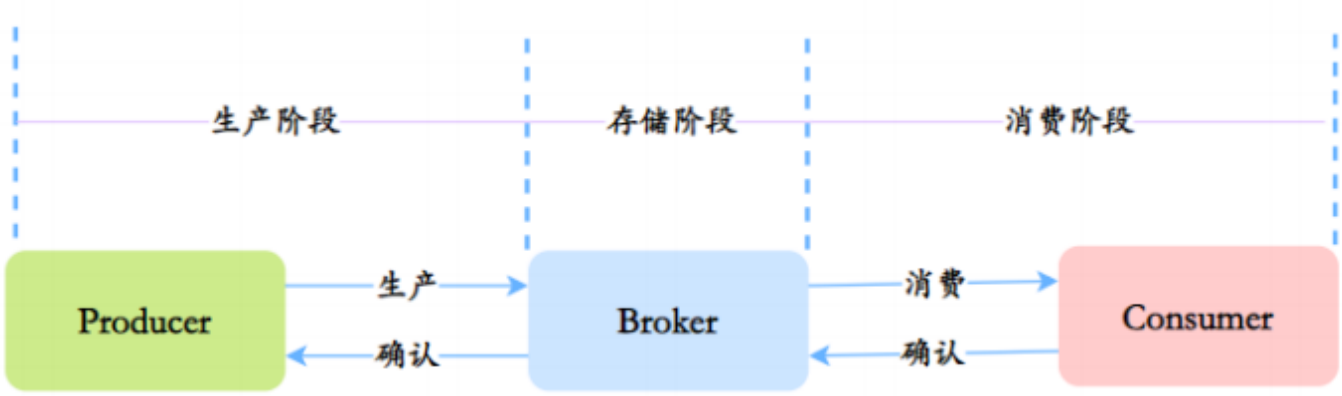
三. 只有readQueueNums>=writeQueueNums,程序才能正常进行

最佳实践是readQueueNums=writeQueueNums。那rocketmq为什么要区分读写队列呢？直接强制readQueueNums=writeQueueNums，不就没有问题了吗？rocketmq设置读写队列数的目的在于方便队列的缩容和扩容。

如何处理消息丢失的问题?

消息可能在哪些阶段丢失呢？可能会在这三个阶段发生丢失：生产阶段、存储阶段、消费阶段。

所以要从这三个阶段考虑：



生产阶段丢失的可能：

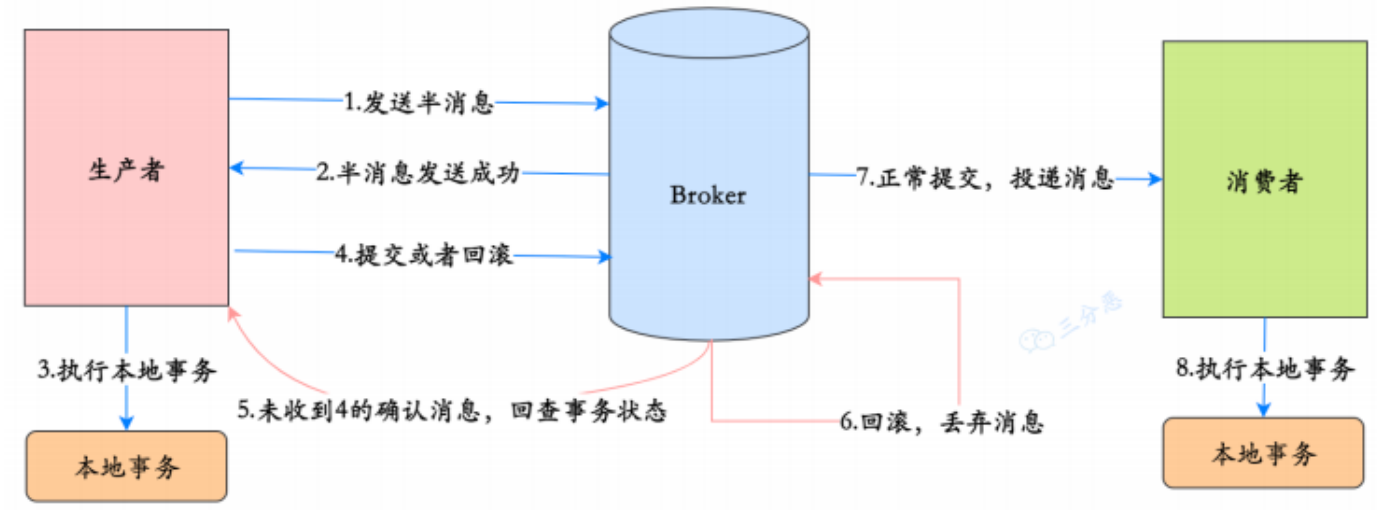
生产者发送消息时由于网络故障或broker的master节点宕机导致消息丢失。

生产阶段可靠性方案：

1、事务消息机制（最优解，更快，避免大量重试）

半消息：是指暂时还不能被 Consumer 消费的消息，Producer 成功发送到 Broker 端的消息，但是此消息被标记为 “暂不可投递” 状态，只有等 Producer 端执行完本地事务后经过二次确认了之后，Consumer 才能消费此条消息。

依赖半消息，可以实现分布式消息事务，其中的关键在于二次确认以及消息回查：



1、Producer 向 broker 发送半消息，broker会把half消息写入特定的半消息主题里去。

2、写入完成后会给 Producer 返回响应，消费者是看不到的；

3、Producer 端收到响应后判断，消息写入成功，Producer 端就开始执行本地事务；写入失败会进行回滚。

4、正常情况本地事务执行完成，Producer 向 Broker 发送 Commit/Rollback，如果是 Commit，Broker 端将半消息标记为正常消息，Consumer 可以消费，如果是 Rollback，Broker 丢弃此消息。

5、异常情况，Broker 端迟迟等不到二次确认。在一定时间后，会查询所有的半消息，然后到 Producer 端查询半消息的执行情况，查15次没查到默认回滚。

6、Producer 端查询本地事务的状态，根据事务的状态提交 commit/rollback 到 broker 端提交投递或者丢弃回滚消息。

7、Consumer 端消费到消息之后，执行本地事务，执行本地事务。

2、同步发送消息 + 重试+ 备忘录模式/回滚

在发送消息代码里用try/catch捕获异常，假如有异常了就可以重试，一定次数之后还是失败，就说明此时可能Broker都彻底崩溃了，此时可以把消息顺序写入到持久化存储中去或者直接回滚掉。

之后还要不停尝试检查Broker是否恢复，一旦发现Broker恢复了，搞一个异步线程可以把之前持久化存储的消息都按照顺序查询出来发送到Broker里去，这样才能保证消息不会因为Broker彻底崩溃而丢失。

存储阶段丢失的可能：

消息到达broker，broker作为的一个jvm进程突然崩溃掉，page cache里的数据是os管理的，如果是这样是不会丢的。

只有是当使用异步刷盘时可能消息对应的commitLog还在page cache中未刷新到磁盘，此时物理机宕机了，重启导致page cache中数据丢失；也可能是消息从page cache刷到磁盘了，磁盘损坏导致的丢失。

存储阶段可靠性方案：

1、刷盘策略调整为同步刷盘

Producer发送消息后等Broker收到消息，写入page cache再写入磁盘，收到响应了才行。

2、Broker主从同步复制模式

对Broker使用主从模式，让一个Master有一个Slave去同步它的数据，一条消息写入成功必须是让Slave也写入成功，同步复制模式可以保证即使Master宕机，消息肯定在Slave中有备份，保证了消息不会丢失。

消费阶段可能的问题：

消息保存到Broker，消费者消费消息时未完成就发ACK让Broker以为消息消费成功了跳到了下一个offset。

消费阶级可靠性方案：

1、手动提交offset+自动故障转移

消费者成功消费才返回成功响应把offset提交到Broker。

如果未消费完宕机，则消费者回滚本地事务，不返回offset给Broker，Broker感知到消费者未完成消费，会Rebalance重新分配MessageQueue对应的消费者，自动将消息发送给消费者组内其他消费者。

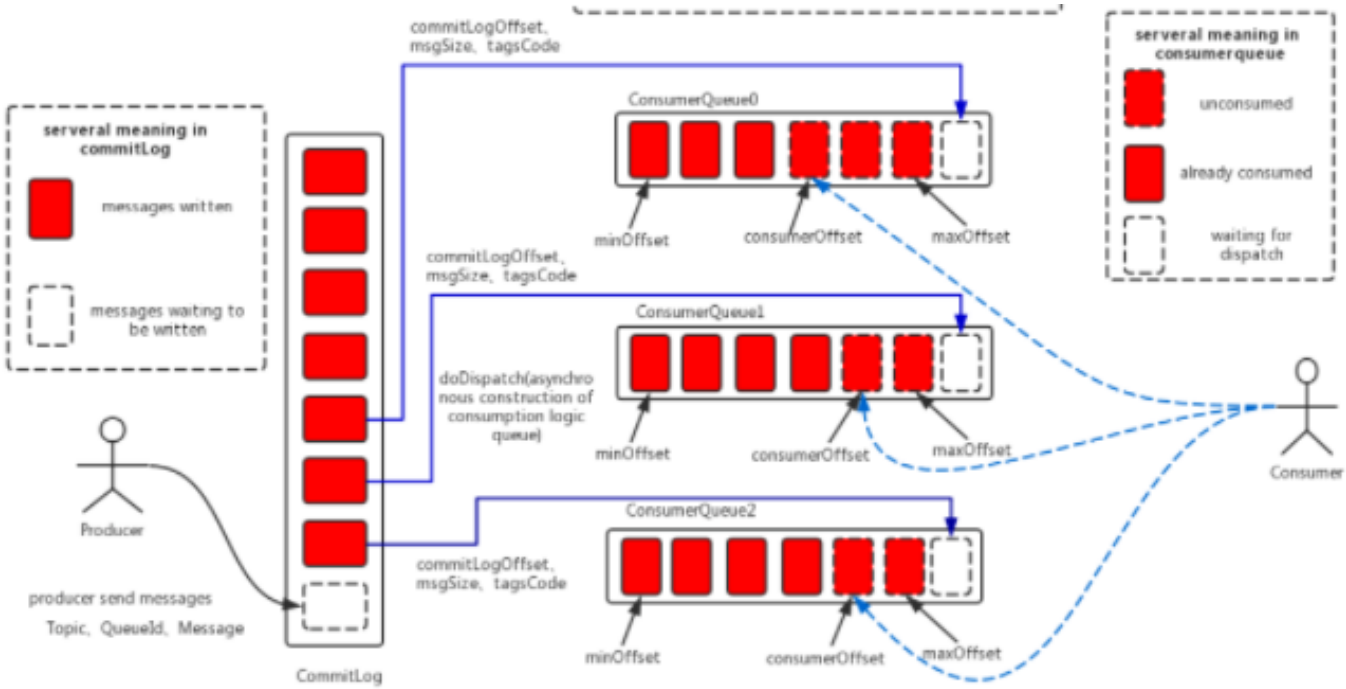
说一下RocketMQ组成，每个角色作用是啥?

NameServer：负责维护元数据/提供路由服务的注册中心

（无状态，每个Broker启动都得向所有的NameServer进行注册，保证每个NameServer都会有一份集群中所有存活的Broker的信息。生产者和消费者通过TCP长链接定时从NameServer拉取Broker信息。Broker通过TCP长链接每隔一段时间向NameServer发送一次心跳，NameServer运行一次定时任务，定期检测Broker有没有超时没有发送心跳，如果没发就说明挂掉了）

Broker：负责消息存储/转发的数据节点

（Broker 内部维护着一个个 Consumer Queue，用来存储消息的索引，真正存储消息的地方是 CommitLog日志文件



）

Producer：负责发送消息的生产者，由用户实现

（RocketMQ 提供了三种方式发送消息：同步、异步和单向

同步发送：同步发送指消息发送方发出数据后会在收到接收方发回响应之后才发下一个数据包。一般用于重要通知消息，例如重要通知邮件、营销短信。

异步发送：异步发送指发送方发出数据后，不等接收方发回响应，接着发送下个数据包，一般用于可能链路耗时较长而对响应时间敏感的业务场景，例如用户视频上传后通知启动转码服务。

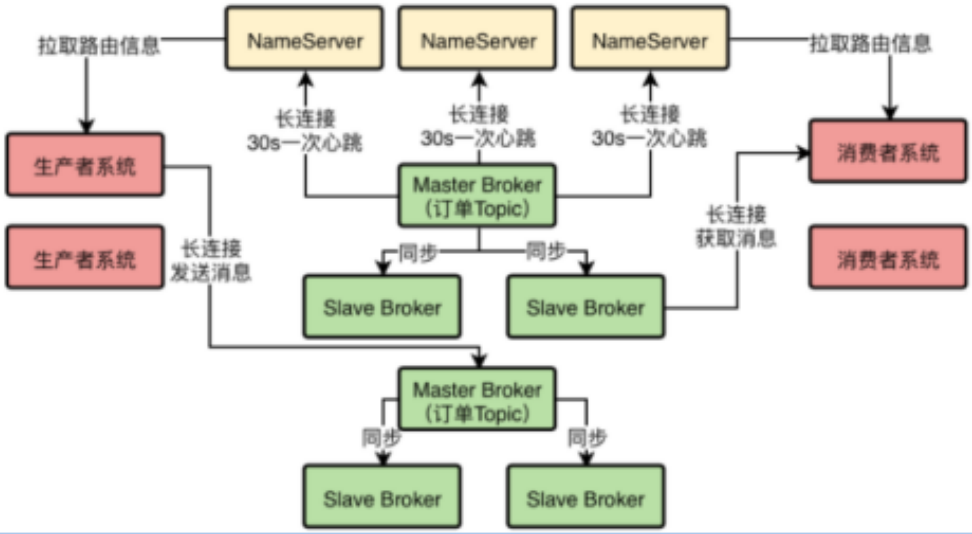
单向发送：单向发送是指只负责发送消息而不等待服务器回应且没有回调函数触发，适用于某些耗时非常短但对可靠性要求并不高的场景，例如日志收集。）

Consumer：负责消费消息的消费者，由用户实现

（支持PUSH和PULL两种消费模式，支持集群消费和广播消费，提供实时的消息订阅机制。

Pull：拉取型消费者（Pull Consumer）主动从消息服务器拉取信息，只要批量拉取到消息，用户应用就会启动消费过程，所以 Pull 称为主动消费型。

Push：推送型消费者（Push Consumer）封装了消息的拉取、消费进度和其他的内部维护工作，将消息到达时执行的回调接口留给用户应用程序来实现。所以 Push 称为被动消费类型，但其实从实现上看还是从消息服务器中拉取消息，不同于 Pull 的是 Push 首先要注册消费监听器，当监听器处触发后才开始消费消息。）



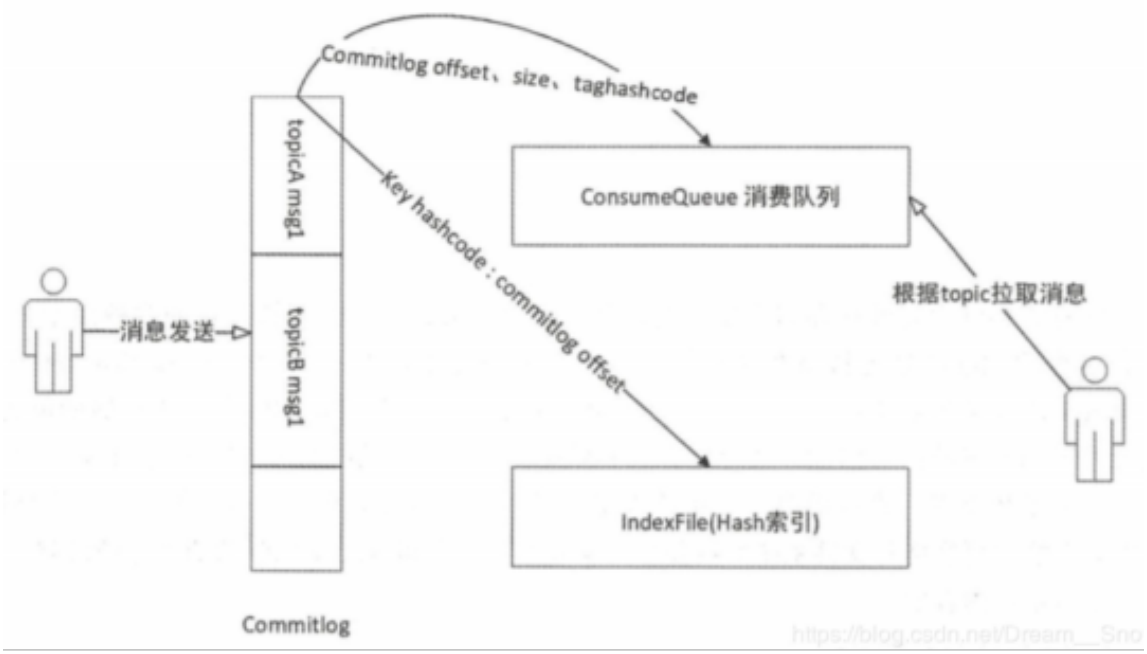
Topic和MessageQueue和ConsumeQueue和CommitLog和Broker是怎样一个联系?

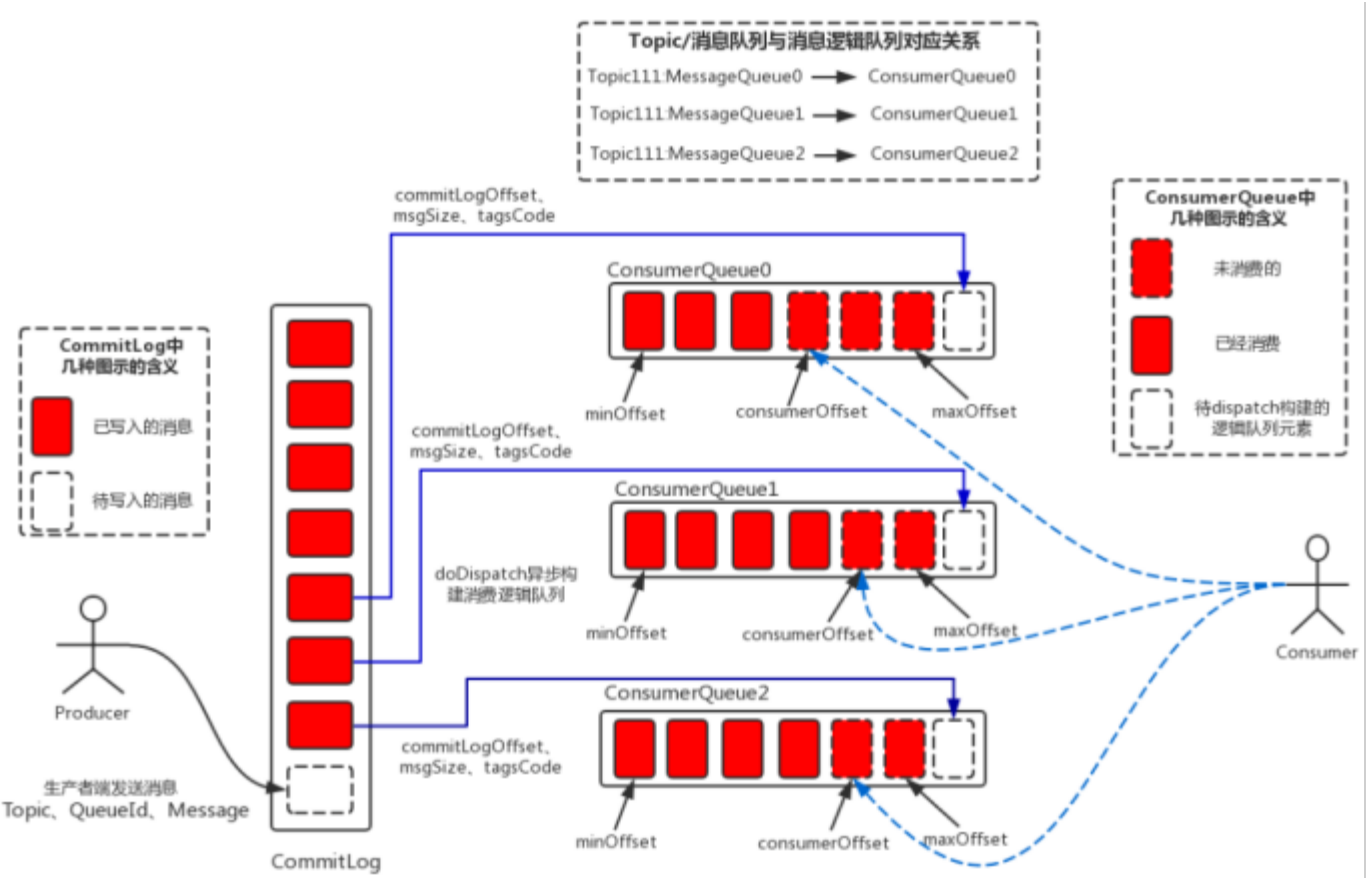
topic表示某一类数据逻辑上的集合，生产者生产数据要分类放入，消费者消费数据也要分类获取，根据的都是topic类型。

messageQueue表示topic的一个分片数据，一个messageQueue对应着多个consumeQueue文件，consumeQueue相当于是messageQueue分片数据所在CommitLog文件里面的偏移量索引的文件。

broker是真正存放数据消息的地方，包括CommitLog和consumeQueue文件。一个broker对应一个CommitLog，CommitLog是真正存放消息数据的地方，一个broker的多个consumeQueue文件都对应同一个CommitLog文件。

topic因为messageQueue分片了，不同的分片分散在多个broker机器上，能实现海量存储消息数据，多台机器也可以抗下更高的并发写入消息数据;而一个messageQueue只能够被一个消费者消费，所以一台机器上多个messageQueue又可以有多个消费者并发消费，加快消费速度；消费者通过consumeQueue又可加快对CommitLog中messageQueue消息的检索。





如何解决消息队列的延时以及过期失效问题?

mq 中的消息过期失效了

假设你用的是 RabbitMQ，RabbtiMQ 是可以设置过期时间的，也就是 TTL。如果消息在 queue 中积压超过一定的时间就会被 RabbitMQ 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是**大量的数据会直接搞丢**。

这个情况下，就不是说要增加 consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是**批量重导**，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上 12 点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。

假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。

怎么处理消息积压?

消费者消费消息过慢，或者消费者由于自身原因消费失败怎么办?

分情况讨论：

消费过慢

可以提高消费者的并行度，部署更多的consumer机器，Topic的MessageQueue得是有对应的增加，因为如果你的consumer机器有5台，然后MessageQueue只有4个，那么意味着有一个consumer机器是获取不到消息的。

然后就是可以增加consumer的线程数量，一台consumer机器上的消费线程越多，消费的速度就越快。

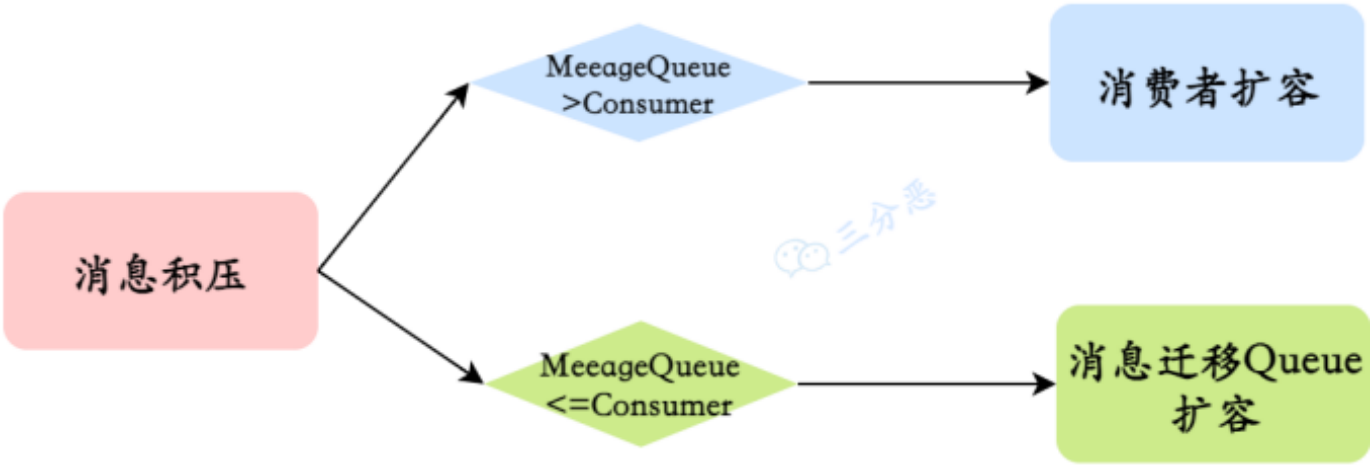
此外，还可以开启消费者的批量消费功能，一次性批量处理一些数据。

消费故障

1. 跳过非重要消息：

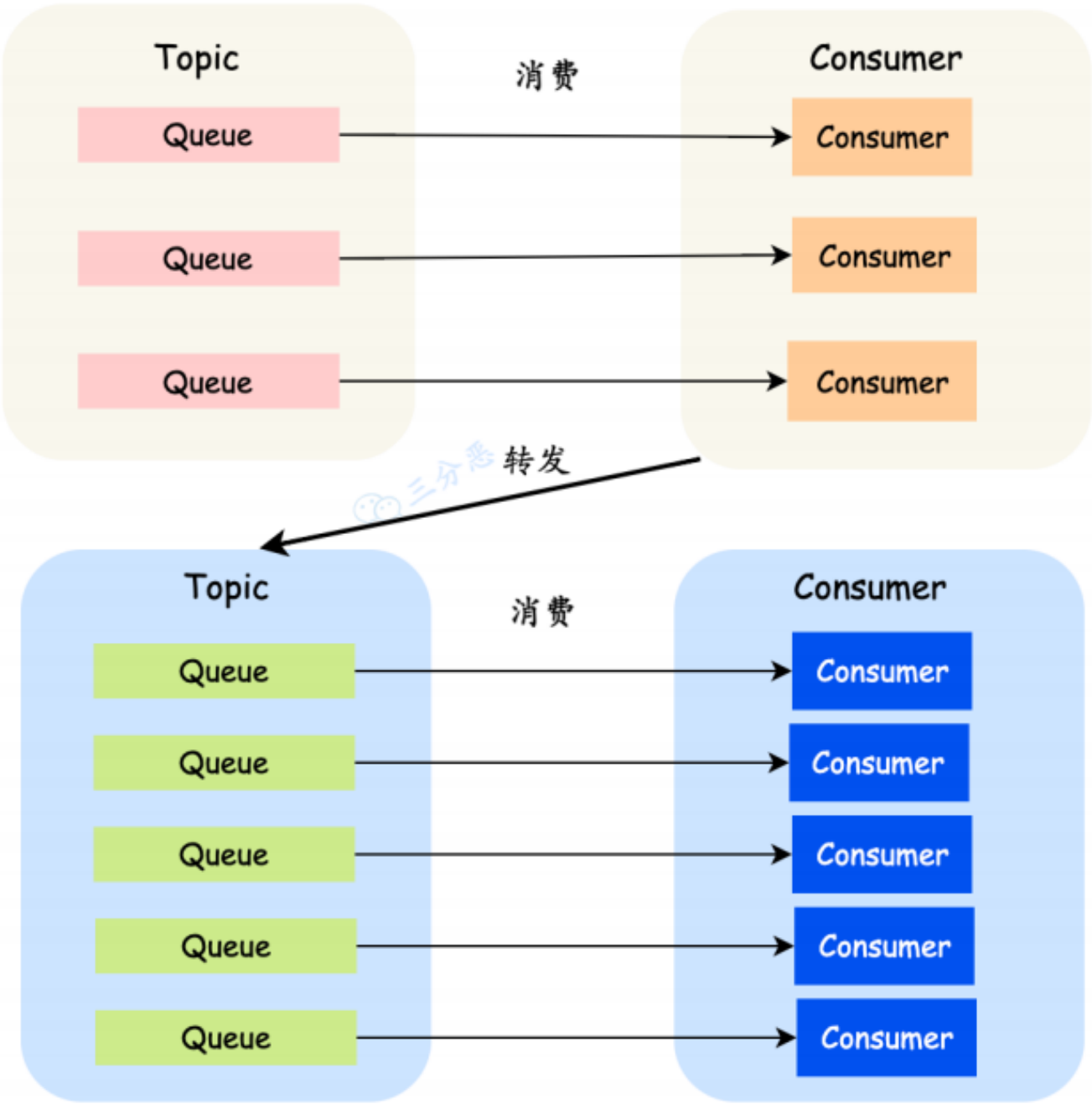
如果这些消息你是允许丢失的，可以把消息全部丢弃

1. 提高消费能力：



加消费者机器（MessageQueue比消费者多，不改代码），临时申请多台机器多个部署消费者系统的实例，然后消费者系统同时消费，每个人消费一个MessageQueue的消息。处理完百万积压的消息之后，就可以下线多余的机器了。

加消费者机器（MessageQueue少，得改代码），这个时候就没办法扩容消费者系统了，因为你加再多的消费者系统，还是只有几个MessageQueue，没法并行消费。所以此时往往是临时修改那消费者系统的代码，让他们获取到消息不是正常去处理，而是直接把消息写入一个新的Topic，这个速度是很快的，因为仅仅是转发一下，不用业务处理。然后新的Topic有更多个MessageQueue，然后再部署更多台临时增加的消费者系统，去消费新的Topic，消费完之后恢复原状。

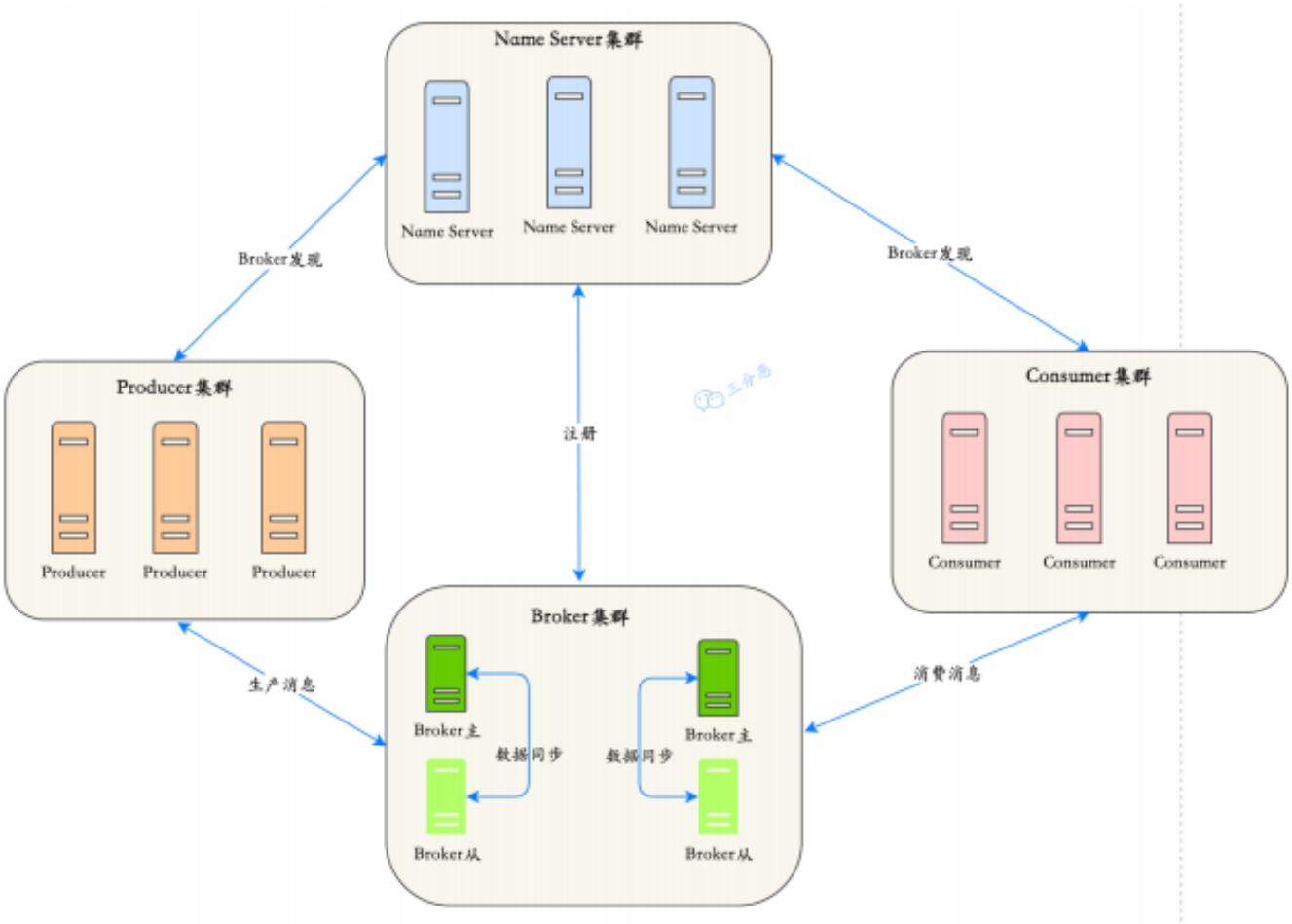


1. 批量方式消费：

某些业务流程如果支持批量方式消费，则可以很大程度上提高消费吞吐量。

1. 优化每条消息消费过程，提升消费者的硬件配置或者改善消息消费的处理逻辑。

RoctetMQ基本架构了解吗?



RocketMQ 一共有四个部分组成：

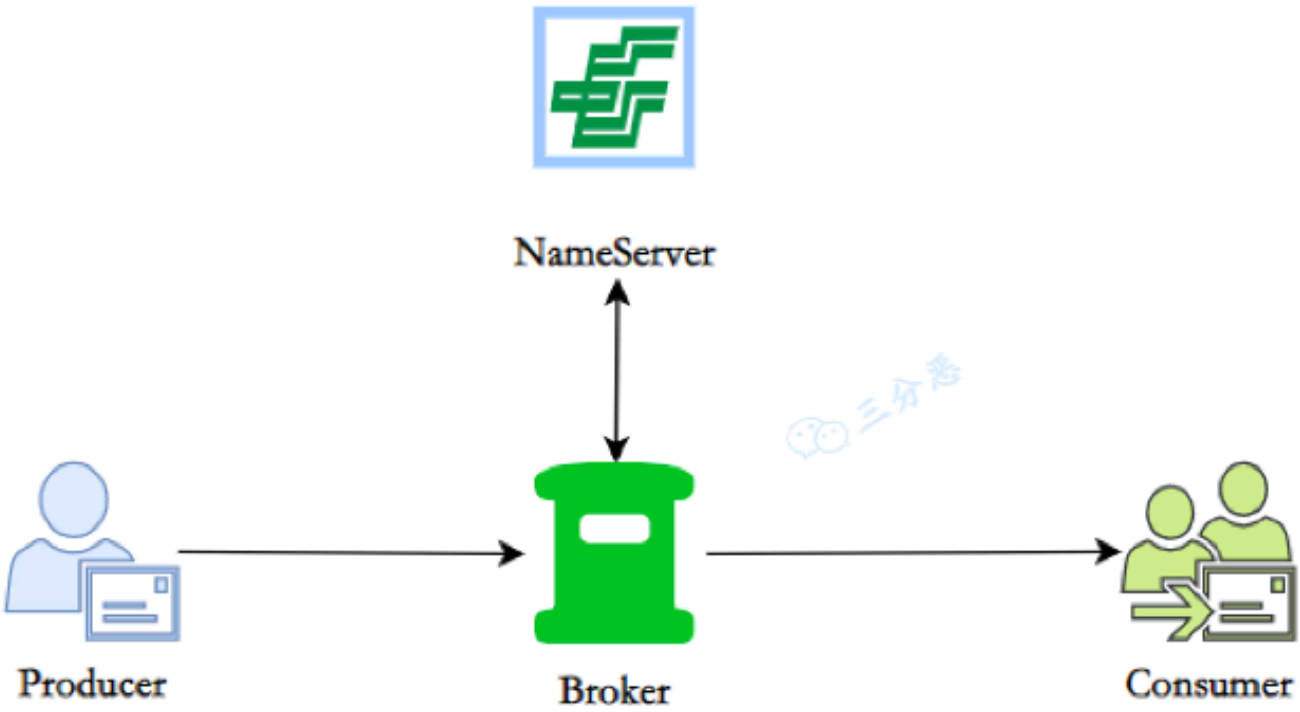
NameServer ， Broker ， Producer 生产者， Consumer 消费者，它们对应了：发现、发、

存、收，为了保证高可用， 一般每一部分都是集群部署的。

那能介绍一下这四部分吗?

类比一下我们生活的邮政系统——邮政系统要正常运行，离不开下面这四个角色， 一是发信者，二是收信者， 三是负责暂存传输的邮局，

四是负责协调各个地方邮局的管理机构。 对应到 RocketMQ 中，这四个角色就是 Producer 、 Consumer 、 Broker 、 NameServer



NameServer

NameServer 是一个无状态的服务器， 角色类似于 Kafka使用的 Zookeeper，但比 Zookeeper 更轻量。

特点：

每个 NameServer 结点之间是相互独立，彼此没有任何信息交互。

Nameserver 被设计成几乎是无状态的，通过部署多个结点来标识自己是一个伪集群，Producer 在发送消息前从 NameServer 中获取 Topic 的路由信息也就是发往哪个 Broker， Consumer 也会定时从 NameServer 获取 Topic 的路由信息， Broker 在启动时会向 NameServer 注册，并定时进行心跳连接，且定时同步维护的 Topic 到 NameServer 。

功能主要有两个：

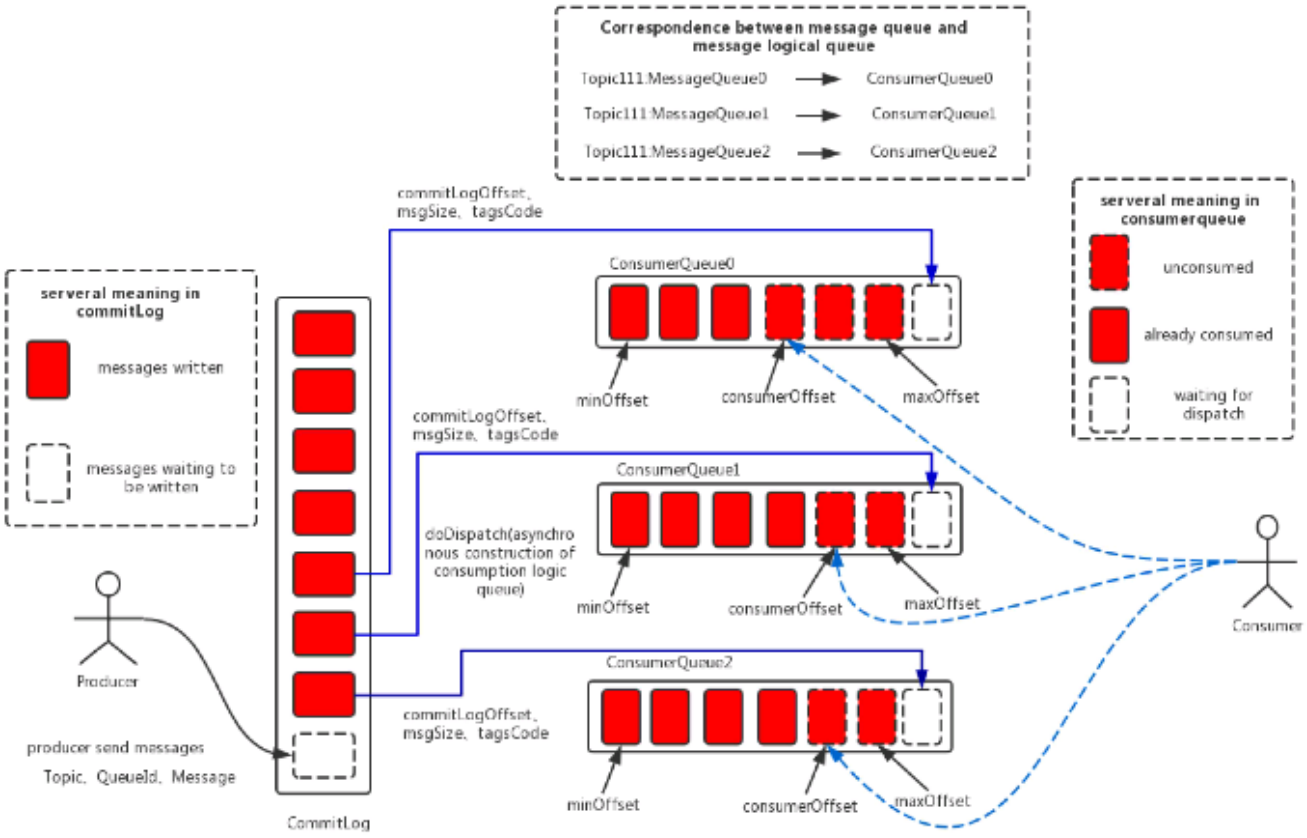
1、和Broker 结点保持长连接。

2、维护 Topic 的路由信息。

Broker

消息存储和中转角色，负责存储和转发消息。

Broker 内部维护着一个个 Consumer Queue，用来存储消息的索引，真正存储消息的地方是 CommitLog（日志文件）。



单个 Broker 与所有的 Nameserver 保持着长连接和心跳，并会定时将 Topic 信息同步到 NameServer，和 NameServer 的通信底层是通过 Netty 实现的。

Producer

消息生产者，业务端负责发送消息，由用户自行实现和分布式部署。

Producer由用户进行分布式部署，消息由Producer通过多种负载均衡模式发送到Broker集群，发送低延时，支持快速失败。

RocketMQ 提供了三种方式

发送消息：同步、异步和单向

同步发送： 同步发送指消息发送方发出数据后会在收到接收方发回响应之后才发下一个数据包。 一般用于重要通知消息，例如重要通知邮件、营销短信。

异步发送： 异步发送指发送方发出数据后，不等接收方发回响应，接着发送下个数据包， 一般用于可能链路耗时较长而对响应时间敏感的业务场景，例如用户视频上传后通知启动转码服务。

单向发送： 单向发送是指只负责发送消息而不等待服务器回应且没有回调函数触发，适用于某些耗时非常短但对可靠性要求并不高的场景，例如日志收集。

Consumer

消息消费者，负责消费消息， 一般是后台系统负责异步消费。

Consumer 也由用户部署，支持PUSH和PULL两种消费模式，支持集群消费和广播消费，提供实时的消息订阅机制 。

Pull： 拉取型消费者主动从消息服务器拉取Topic和MessageQueue信息，自己记录下一次的offset，只要批量拉取到消息，用户应用就会启动消费过程，所以 Pull 称为主动消费型。

Push： 推送型消费者封装了消息的拉取、消费进度和其他的内部维护工作，将消息到达时执行的回调接口留给用户应用程序来实现。所以Push 称为被动消费类型，但其实从实现上看还是从消息服务器中拉取消息，不同于 Pull 的是 Push 首先要注册消费监听器，当监听器处触发后才开始消费消息。

如何实现消息过滤

有两种方案：

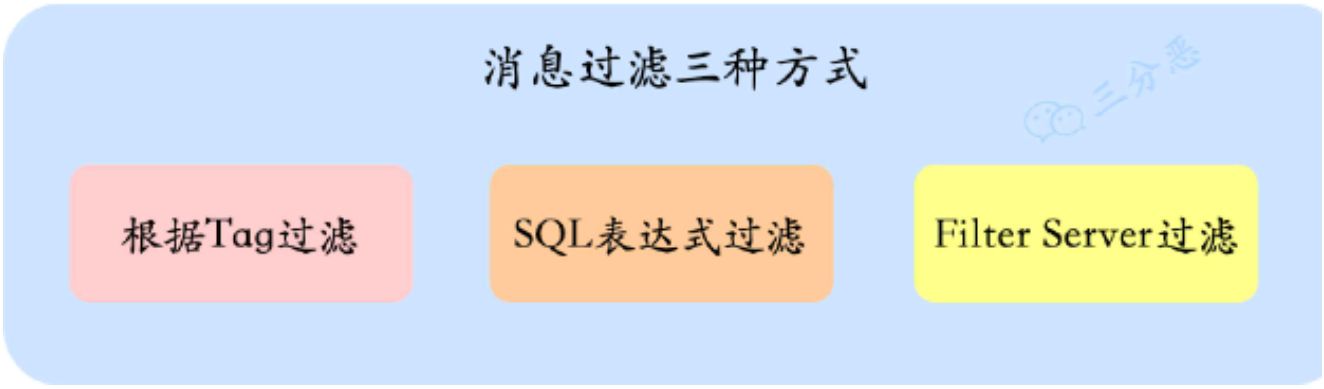
一种是在 Broker 端按照 Consumer 的去重逻辑进行过滤，这样做的好处是避免了无用的消息传

输到 Consumer 端，缺点是加重了 Broker 的负担，实现起来相对复杂。

另一种是在 Consumer 端过滤，比如按照消息设置的 tag 去重，这样的好处是实现起来简单，缺点是有大量无用的消息到达了 Consumer 端

只能丢弃不处理。

一般采用Cosumer端过滤，如果希望提高吞吐量，可以采用Broker过滤。对消息的过滤有三种方式：



根据Tag过滤：这是最常见的一种，用起来高效简单

SQL表达式过滤：更加灵活

Filter Server 方式：最灵活，也是最复杂的一种方式，允许用户自定义函数进行过滤

延时消息了解吗?

电商的订单超时自动取消，就是一个典型的利用延时消息的例子，用户提交了一个订单，就可以发送一个延时消息， 1h后去检查这个订单

的状态，如果还是未付款就取消订单释放库存。

RocketMQ是支持延时消息的，只需要在生产消息的时候设置消息的延时级别

RocketMQ怎么实现延时消息的?

简单，八个字： 临时存储 + 定时任务 。

Broker收到延时消息了，会先发送到主题（SCHEDULE_TOPIC_XXXX）的相应时间段的Message Queue中，然后通过一个定时任务轮询这些队列，到期后，把消息投递到目标Topic的队列中，然后消费者就可以正常消费这些消息。

死信队列知道吗?

死信队列用于处理无法被正常消费的消息，即死信消息。

当一条消息初次消费失败， 消息队列 RocketMQ 会自动进行消息重试 ； 达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时，消息队列 RocketMQ 不会立刻将消息丢弃，而是将其发送到该消费者对应的特殊队列中，该特殊队列称为死信队列 。

死信消息的特点：

不会再被消费者正常消费。

有效期与正常消息相同，均为 3 天， 3 天后会被自动删除。因此，需要在死信消息产生后的 3 天内及时处理。

死信队列的特点：

一个死信队列对应一个 Group ID ， 而不是对应单个消费者实例。

如果一个 Group ID 未产生死信消息，消息队列 RocketMQ 不会为其创建相应的死信队列。

一个死信队列包含了对应 Group ID 产生的所有死信消息，不论该消息属于哪个Topic 。

RocketMQ 控制台提供对死信消息的查询、导出和重发的功能。

说一下RocketMQ的整体工作流程?

简单来说，RocketMQ是一个分布式消息队列，也就是 消息队列 + 分布式系统 。

作为消息队列，它是 发 - 存 - 收 的一个模型，对应的就是Producer、Broker、Cosumer；

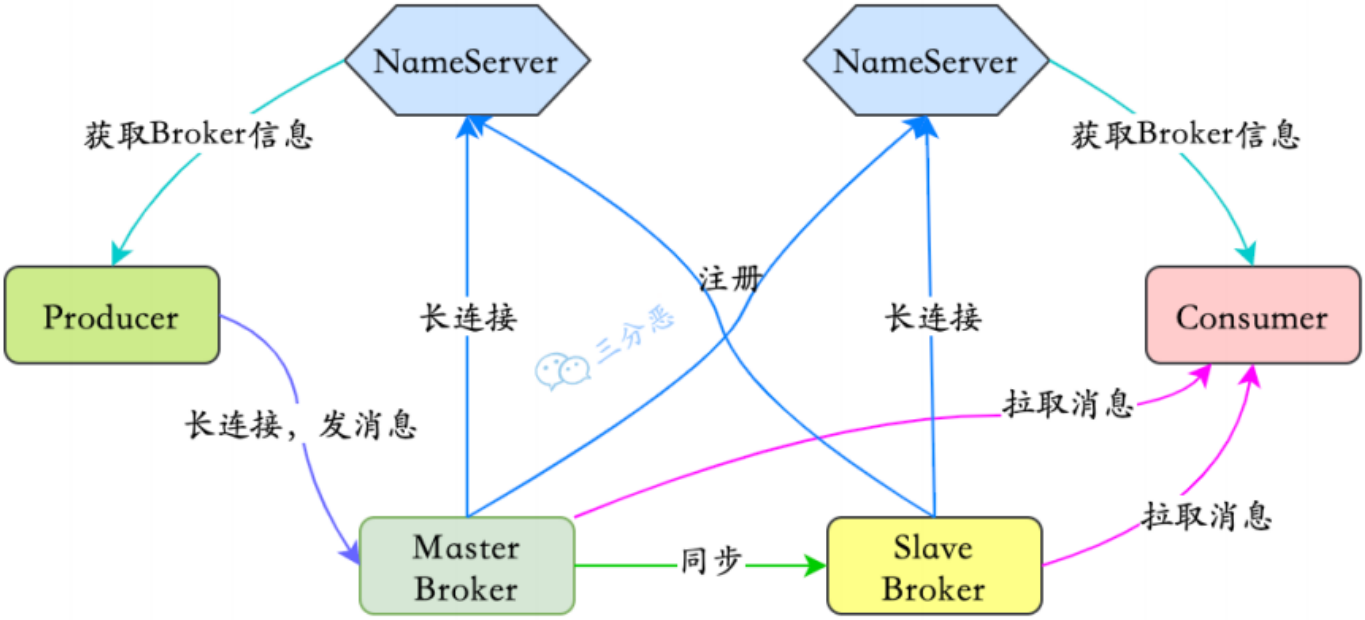
作为分布式系统，它要有服务端、客户端、注册中心，对应的就是 Broker、Producer/Consumer、NameServer 。

所以我们看一下它主要的工作流程：RocketMQ由NameServer注册中心集群、Producer生产者集群、Consumer消费者集群和若干Broker（RocketMQ进程）组成：

1. Broker在启动的时候去向所有的NameServer注册，并保持长连接，每30s发送一次心跳

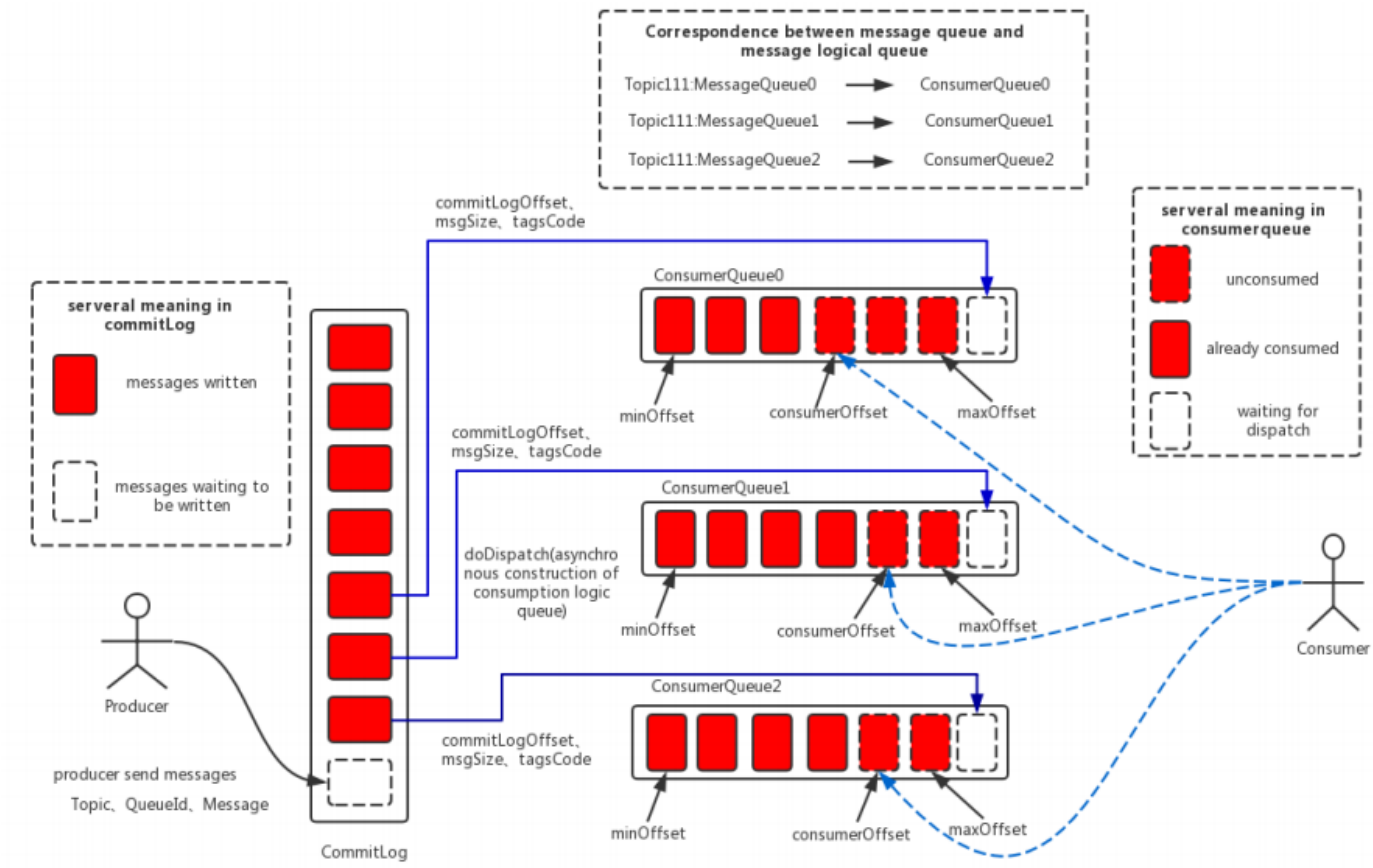
2. Producer在发送消息的时候从NameServer获取Broker服务器地址，根据负载均衡算法选择一台服务器　来发送消息

3. Conusmer消费消息的时候同样从NameServer获取Broker地址，然后主动拉取消息来消费



Broker是怎么保存数据的呢?

RocketMQ主要的存储文件包括CommitLog文件、ConsumeQueue文件、Indexfile文件。



CommitLog：消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容，消息内容不是定长的。消息主要是顺序写入日志文件，当文件满了，写入下一个文件。

ConsumeQueue：消息消费队列，引入的目的主要是提高消息消费的性能，由于RocketMQ是基于主题topic的订阅模式，消息消费是针对主题进行的，如果要遍历commitlog文件中根据topic检索消息是非常低效的。 Consumer即可根据ConsumeQueue来查找待消费的消息。其中，ConsumeQueue（逻辑消费队列）作为消费消息的索引，保存了指定Topic下的队列消息在CommitLog中的起始物理偏移量offset，消息大小size和消息Tag的HashCode值。

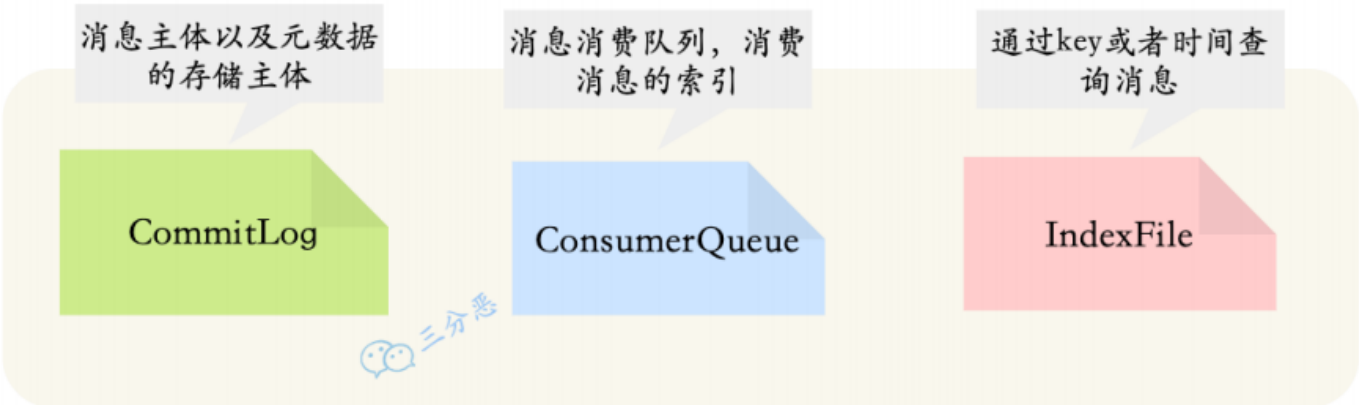
IndexFile：IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。IndexFile的底层存储设计为在文件系统中实现HashMap结构，故RocketMQ的索引文件其底层实现为hash索引。

总结：

RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。

RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。

只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此，Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。这里，RocketMQ的具体做法是，使用Broker端的后台服务线程不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和 IndexFile（索引文件）数据。



说说RocketMQ怎么对文件进行读写的?

RocketMQ对文件的读写巧妙地利用了操作系统的一些高效文件读写方式——

PageCache 、 顺序读写 、 零拷贝 。

- PageCache、顺序读取

在RocketMQ中，ConsumeQueue逻辑消费队列存储的数据较少，并且是顺序读取，在page cache机制的预读取作用下，Consume Queue文件的读性能几乎接近读内存，即使在有消息堆积情况下也不会影响性能。而对于CommitLog消息存储的日志数据文件来说，读取消息内容时候会产生较多的随机访问读取，严重影响性能。如果选择合适的系统IO调度算法，比如设置调度算法为“Deadline”，随机读的性能也会有所提升。页缓存（PageCache)是OS对文件的缓存，用于加速对文件的读写。一般来说，程序对文件进行顺序读写的速度几乎接近于内存的读写速度，主要原因就是由于OS使用PageCache机制对读写访问操作进行了性能优化，将一部分的内存用作PageCache。对于数据的写入，OS会先写入至Cache内，随后通过异步的方式由内核线程将Cache内的数据刷盘至物理磁盘上。对于数据的读取，如果一次读取文件时出现未命中PageCache的情况，OS从物理磁盘上访问读取文件的同时，会顺序对其他相邻块的数据文件进行预读取。

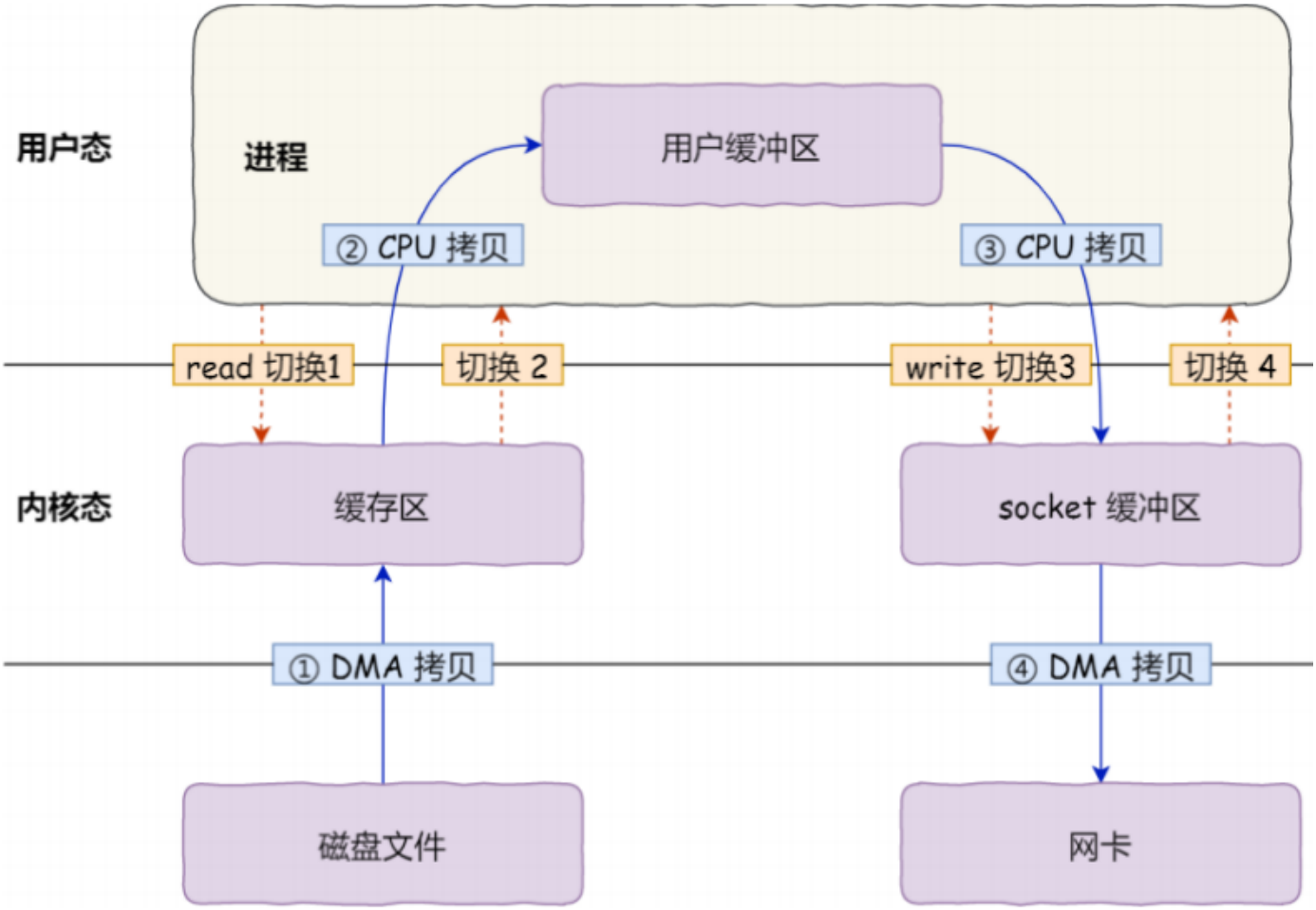
- 零拷贝

另外，RocketMQ主要通过MappedByteBuffer对文件进行读写操作。其中，利用了NIO中的FileChannel模型将磁盘上的物理文件直接映射到用户态的内存地址中（这种Mmap的方式减少了传统IO，将磁盘文件数据在操作系统内核地址空间的缓冲区，和用户应用程序地址空间的缓冲区之间来回进行拷贝的性能开销），将对文件的操作转化为直接对内存地址进行操作，从而极大地提高了文件的读写效率（正因为需要

使用内存映射机制，故RocketMQ的文件存储都使用定长结构来存储，方便一次将整个文件映射至内存）。

说说什么是零拷贝?

在操作系统中，使用传统的方式，数据需要经历几次拷贝，还要经历用户态/内核态切换。



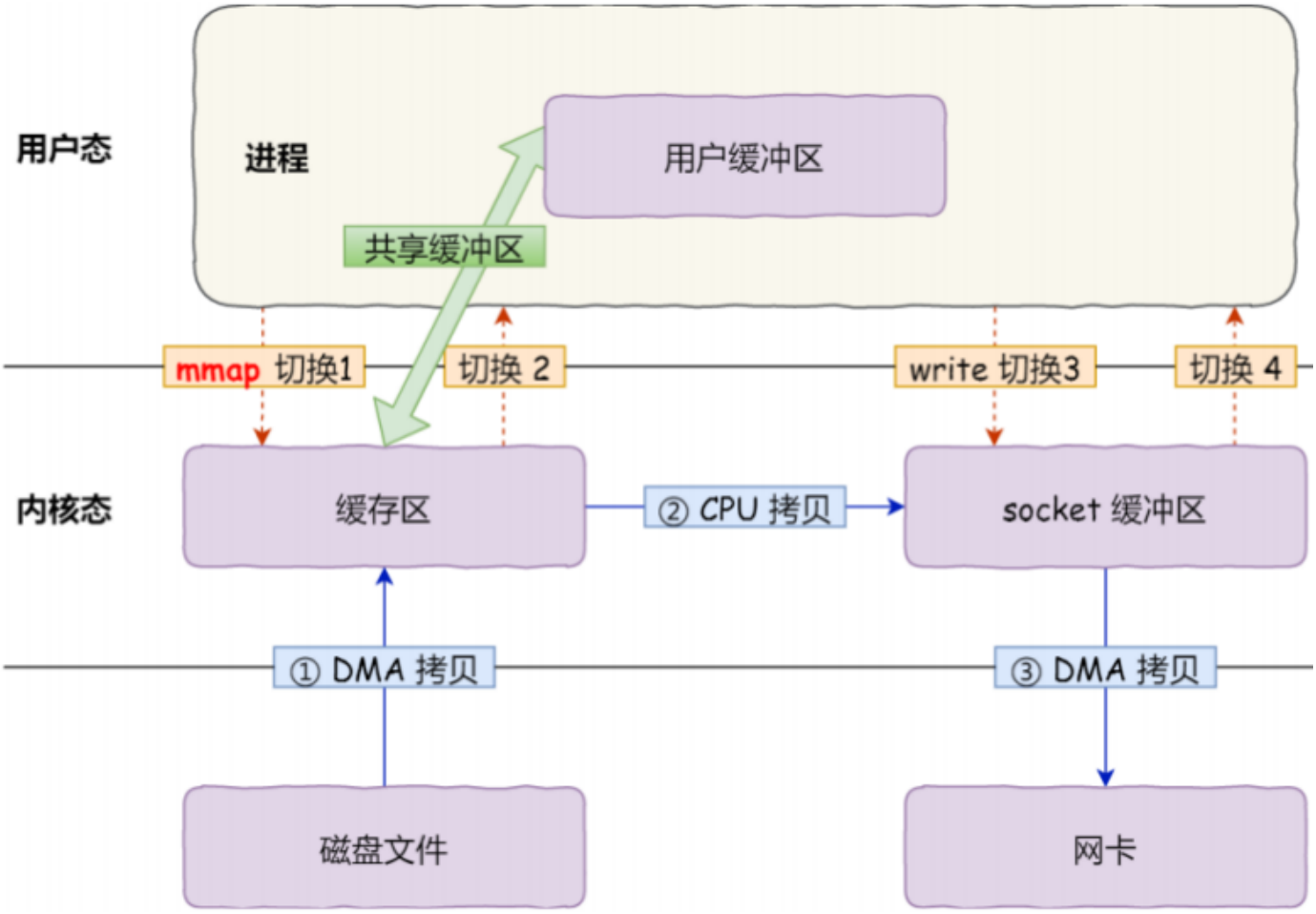
1. 从磁盘复制数据到内核态内存；

2. 从内核态内存复制到用户态内存；

3. 然后从用户态内存复制到网络驱动的内核态内存；

4. 最后是从网络驱动的内核态内存复制到网卡中进行传输。

所以，可以通过零拷贝的方式，减少用户态与内核态的上下文切换和内存拷贝的次数，用来提升I/O的性能。零拷贝比较常见的实现方式是**mmap**，这种机制在Java中是通过MappedByteBuffer实现的。



消息刷盘怎么实现的呢?

RocketMQ提供了两种刷盘策略：同步刷盘和异步刷盘

同步刷盘：在消息达到Broker的内存之后，必须刷到commitLog日志文件中才算成功，然后返回Producer数据已经发送成功。

异步刷盘：异步刷盘是指消息达到Broker内存后就返回Producer数据已经发送成功，会唤醒一个线程去将数据持久化到CommitLog日志文件中。

Broker 在消息的存取时直接操作的是内存（内存映射文件），这可以提供系统的吞吐量，但是无法避免机器掉电时数据丢失，所以需要持久化到磁盘中。

刷盘的最终实现都是使用**NIO**中的 MappedByteBuffer.force() 将映射区的数据写入到磁盘，如果是同步刷盘的话，在**Broker**把消息写到**CommitLog**映射区后，就会等待写入完成。

异步而言，只是唤醒对应的线程，不保证执行的时机。

RocketMQ消息长轮询了解吗?

所谓的长轮询，就是Consumer 拉取消息，如果对应的 Queue 如果没有数据，Broker 不会立即返回，而是把 PullReuqest hold起来，等待 queue 有了消息后，或者长轮询阻塞时间到了，再重新处理该 queue 上的所有 PullRequest。