**电商前端交易型系统设计原则**

摘自：<http://blog.csdn.net/qq_26562641/article/details/53004566>

从毕业到现在已经快7年开发经验了，做过基础用户系统、积分商城、偷菜游戏、论坛、博客等等；也一个人全栈开发在线视频网站（http://sishuok.com/），也开发过几万、几十万、几千万、几个亿不同量级的系统，踩过不少坑，也学到许多经验。

设计了一些系统，也有了一些自己的观点，个人认为设计系统要因场景因时间而异，一个系统不是一下子就设计的非常完美，在有限的资源情况下一定是先解决当下最核心的问题，并预测/发现未来可能出现的问题，一步步解决最痛点的问题。也就是说系统设计是不断迭代的过程，在迭代中发现问题修复问题；即满足需求的系统是不断迭代优化出来的，不是一下子就架构的非常完美，这是一个持续的过程，个人不相信完美架构银弹。不过如果一开始就有好的基础系统设计，未来可以更容易达到一个比较满意的目标。

在设计系统时应该多思考墨菲定律：

1、任何事都没有表面看起来那么简单；

2、所有的事都会比你预计的时间长；

3、会出错的事总会出错；

4、如果你担心某种情况发生，那么它就更有可能发生。

但是也要思考80/20法则，在系统设计初期将有限的资源用到刀刃上，我们的目标是系统满足现有需求并能支持未来需求。

在持续开发系统过程中前辈们也总结了很多设计原则/经验；而我个人也有幸运用了一些经验/原则。设计原则是系统发展初期或进化过程中根据自己系统特征匹配使用的，如果刚开始不是核心问题请不要复杂化系统设计。

## 拆分

在系统设计初期，是做一个大而全的系统还是进行按功能拆分系统这个需要进行权衡；比如做私塾在线时本身用户量/交易量不会特别大，而且开发就我一个人，资源有限，那就没必要对系统拆分（比如拆分商品、订单等等），就是做一个大而全的系统。而比如设计一个京东秒杀系统，预测到一旦上线量会非常大，而且投入的资源还是蛮充足的，这种情况下就要考虑进行按功能拆分系统。

笔者遇到的拆分主要有如下几种情况：

* 系统维度：按照系统功能/业务拆分，比如商品系统、购物车、结算、订单系统等等；
* 功能维度：对一个系统进行功能再拆分，比如优惠券系统，可以拆分为后台券创建系统、领券系统、用券系统等；
* 读写维度：根据读写比例特征进行拆分，比如商品系统，交易的各个系统都会读取，读大于写，因此就可以进行拆分：商品写服务、商品读服务；读服务可以考虑全量缓存提升性能；比如写的量太大，需要考虑分库分表；还有些聚合读取的场景，如商品详情页，请考虑数据异构拆分系统，将分散在多处的数据聚合到一处存储，提升读的性能和可靠性；
* AOP维度：根据访问特征，按照AOP进行拆分，比如商品详情页，可以分为CDN、页面渲染系统；CDN就是一个AOP系统；
* 模块维度：比如按照基础或者代码维护特征进行拆分，如基础模块：分库分表、数据库连接池等等；还有如代码维护一般按照三层架构（Web、Service、DAO）进行划分。

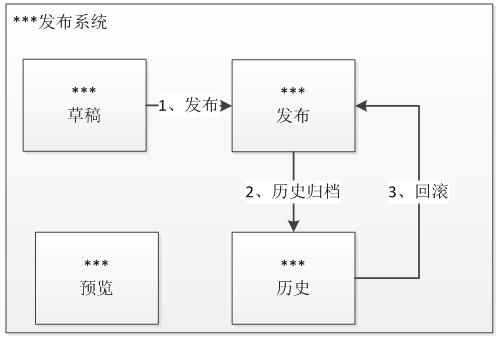
## 服务化

首先判断是不是只需要简单的单点远程服务调用即可，如果单机扛不住了需要集群，是不是可以在客户端注册多台机器，使用Nginx进行负载均衡即可解决；如果随着调用方越来越多，就要考虑使用服务自动注册和发现（如Dubbo使用zookeeper）；还要考虑服务的分组/隔离，比如有的系统访问量太大导致把整个服务打挂，因此需要为不同的调用方提供不同的服务分组，隔离访问；后期还会随着调用量的增加还要考虑如服务的限流、黑白名单等等。还有一些细节需要注意，如超时时间、重试机制、服务路由（能动态切换不同的分组）、故障补偿等等，这些都会影响到服务的质量。

总结为进程内服务--->单点远程服务--->集群手动注册服务--->自动注册和发现服务---->服务的分组/隔离/路由---->限流/黑白名单。

## 数据版本化，可回滚

在设计时考虑是否需要进行数据的版本化，数据维护出问题是否需要回滚。比如商品的维护是不是需要版本化。我们目前有一些非常重要的系统需要对数据进行版本化并且支持可回滚。整体设计类似于下图设计：



## 流程可定义

如果接触过保险业务，会发现不同的保险理赔服务是不一样的，因此我们在系统设计时就设计了一套理赔流程服务。而承保流程和理赔流程是分离，然后进行关联，从而可以复用一些理赔流程并提供个性化的理赔流程。

## 状态与状态机

在设计交易订单系统时，会存在正向状态（待付款、待发货、已发货、完成）和逆向状态（取消、退款）等，正向状态和逆向状态应该根据自己系统的特征来决定是不是需要分离存储。

另外还有订单状态的变迁，比如待支付、已支付待发货、待收货、完成的迁移；要考虑是不是需要使用状态机来驱动状态的变更和后续流程节点操作。

还要考虑并发状态修改问题，同时对同一个订单只存在一个修改；状态变更的有序问题，状态变更消息的先到后到问题，如支付成功消息和用户取消消息的时间差。

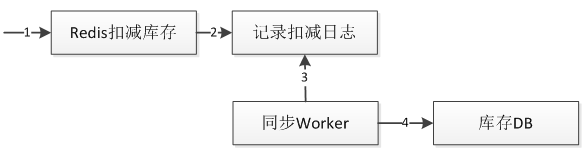
## 消息队列

消息队列，用来解耦一些不需要同步调用的服务或者订阅一些自己系统关心的变化；使用消息队列可以实现服务解耦（一对多消费）、异步、缓冲（削峰）等。比如电商系统中的交易订单数据，该数据有非常多的系统关心并订阅，比如订单生产系统、定期送系统、订单风控系统等等；如果订阅者太多，那么订阅单个消息队列就会成为瓶颈，此时需要考虑对消息队列进行多个镜像复制。

## 大流量缓冲持久化

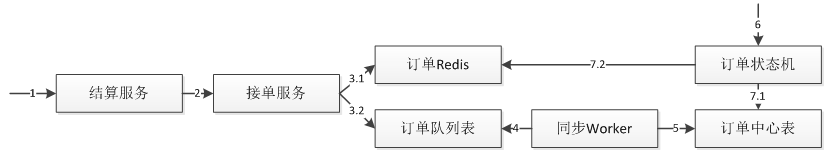
在电商搞大促时，此时的系统流量会高于正常流量的几倍甚至几十倍，此时就要进行一些特殊的设计来保证系统平稳度过这段时期；而解决的手段很多，一般都是牺牲强一致性，而是保证最终一致性即可。

比如扣减库存，可以考虑这样设计：



直接在Redis中扣减，然后记录下扣减日志，通过Worker去同步到DB。

还有如交易订单系统，可以考虑这样设计：



首先结算服务调用订单接单服务将订单存储到：订单Redis和订单队列表，订单队列表可以按照需求水平扩展N个表，通过队列缓冲表提升接单的能力；然后通过同步Worker同步到订单中心表；假设用户支付了订单，订单状态机会驱动状态变更，此时可能订单队列表的订单还没有同步到订单中心表，此时状态机就要根据实际情况进行重试。

如果用户查看单个订单详情可以直接从订单Redis就能查到；但如果查询订单列表需要考虑订单Redis和列表的合并。

同步Worker在设计时需要考虑并发处理和重复处理的问题，单机串行扫描处理（每台Worker只扫描其中的一部分表）还是集群处理（Map-Reduce），另外需要考虑是否需要对订单队列表添加相关字段：处理人（哪个应用正在处理）和处理状态（正在处理、已处理、处理失败）。

## 数据校对

在使用了消息异步机制的场景下，可能存在消息的丢失，需要考虑进行数据校对和修正来保证数据一致性和完整性。可以通过Worker定期去扫描原始表进行补偿，扫描周期根据实际场景进行定义。

## 数据异构化

订单分库分表一般按照订单ID进行分，那么如果要查询某个用户的订单列表就需要聚合N个表的数据然后返回，这样会导致订单表的读性能很低；此时需要对订单表进行异构，异构一套用户订单表，按照用户ID进行分库分表；另外还需要考虑对历史订单数据进行归档处理。

还一种异构场景，如商品详情页，因为数据来源太多，影响服务稳定性的因素就太多了，因此最好的办法是把使用到的数据进行异构存储，形成数据闭环；提升服务的性能和稳定性。而有些数据异构的意义不大，如库存价格可以考虑异步加载，或者并发请求合并。

## 后台系统操作可反馈

在我接触过的很多系统，很多场景都需要反馈，比如修改了某些内容想预览看看最终效果，即想得到一些反馈；还有一些是规则系统，希望看到这些规则在系统数据下的反馈。因此在设计后台系统请考虑效果的可预览、可反馈。

## 后台系统审批化

对于有些重要的后台功能需要设计审批流，比如调整价格，并对操作进行日志记录从而保证操作可追溯、可审计。

## 防重设计

比如结算页需要考虑重复提交，还有如下单扣减库存时需要防止重复扣减库存。解决方案可以考虑防重KEY、防重表。而有些场景如重复支付，如有的电商网站同时支持微信支付、京东支付，渠道不一样是无法防止重复支付的，但是系统设计时需要将支付的每笔情况记录下。比如下图是我在京东使用京东支付和微信支付模拟的重复支付之后进行退款的支付明细：



## 幂等设计

在交易系统中经常会用到消息，而现有消息中间件基本不保证不发生重复消息的消费；因此需要业务系统考虑在重复消息消费时进行幂等处理。还有如使用第三方支付时，第三方支付会进行异步回调，因此也要考虑做好回调的幂等处理。

## 文档&注释

我接触的一些系统是完全没有文档、代码没有注释的，完全都是人传人；这将导致后来人接手很痛苦，而且对有些代码是完全不敢改动的，比如有些代码完全是因为业务的一些特殊情况而写的，可以说是没有注释是完全不懂为什么那么做的。因此在一个系统发展的一开始就应该有文档库（设计架构、设计思想、数据字典/业务流程、现有问题）、业务代码/特殊需求有注释。

## 备份

包括代码和人员。代码主要提交到代码仓库进行管理和备份，代码仓库应该至少具备多版本的功能。人员备份指的是一个系统至少应该有两名开发了解，即假设其中一名离职了也不会出现新人接手之后手忙脚乱事故频发的状况。还有一些是“核心人员”，写着系统的核心代码，被认为是“不可替代的”，这种情况也是尽可能的让他带一名兄弟一起开发核心代码（业务系统），即使离职也还是可以努力下克服困难的。

## 最后

本文只是整理了一小部分原则，还有很多好的原则无法在一篇文章中全部阐述，比如可回滚（系统出问题时第一时间应该回滚处理，必要情况下摘除并保留一台问题机器进行问题排查）、有损服务（故障功能降级/屏蔽、部分人可用、部分系统可用）、灰度发布（功能只对部分人开发，从而保证假设出问题只是影响一小部分人）等等，每一个原则都可以写一篇文章好好阐述。

前端交易型系统本身是非常复杂的，以上原则只是笔者在实际开发时遇到过并使用的一些原则，而还有很多好的原则和经验是可以借鉴的，如果您有好的想法欢迎整理成文章分享给更多的人。另外笔者对支付/结算、供应链、库房生产等部分也不熟悉，只进行了前端交易系统的一些原则的总结，也希望更多人加入进来来完善设计原则库。