美团点评技术团队

美团团购订单系统优化记

思诚 ·2016-12-27 15:58

团购订单系统简介

美团团购订单系统主要作用是支撑美团的团购业务，为上亿美团用户购买、消费提供服务保障。2015年初时，日订单量约400万~500万，同年七夕订单量达到800万。

目标

作为线上S级服务，稳定性的提升是我们不断的追求。尤其像七夕这类节日，高流量，高并发请求不断挑战着我们的系统。发现系统瓶颈，并有效地解决，使其能够稳定高效运行，为业务增长提供可靠保障是我们的目标。

优化思路

2015年初的订单系统，和团购其它系统如商品信息、促销活动、商家结算等强耦合在一起，约50多个研发同时在同一个代码库上开发，按不同业务结点全量部署，代码可以互相修改，有冲突在所难免。同时，对于订单系统而言，有很多问题，架构方面不够合理清晰，存储方面问题不少，单点较多，数据库连接使用不合理，连接打满频发等等。

针对这些问题，我们按紧迫性，由易到难，分步骤地从存储、传输、架构方面对订单系统进行了优化。

具体步骤

1. 存储优化

订单存储系统之前的同事已进行了部分优化，但不够彻底，且缺乏长远规划。具体表现在有分库分表行为，但没有解决单点问题，分库后数据存储不均匀。
此次优化主要从水平、垂直两个方面进行了拆分。垂直方面，按业务进行了分库，将非订单相关表迁出订单库；水平方面，解决了单点问题后进行了均匀拆库。

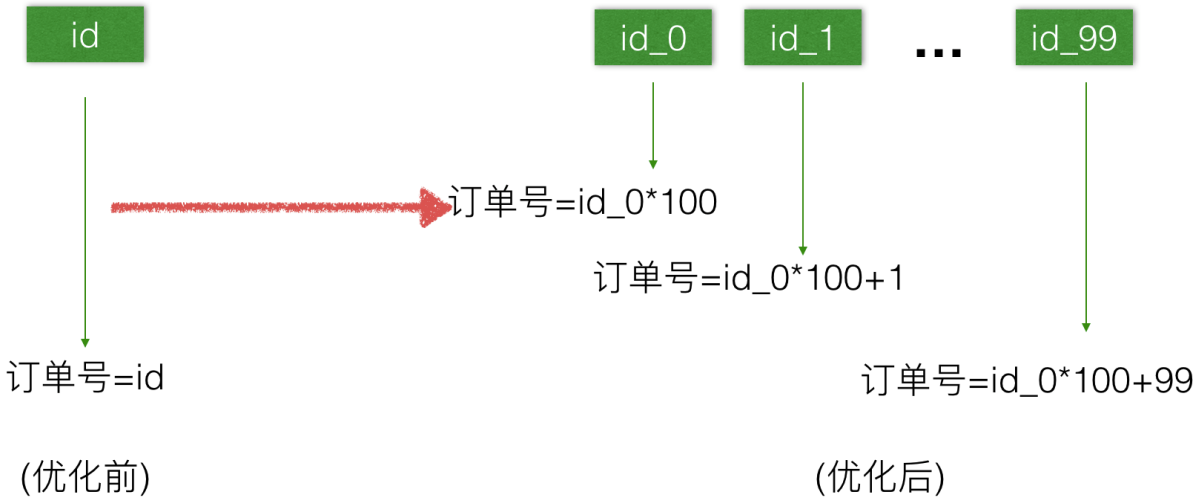
这里主要介绍一下ID分配单点问题：

系统使用一张表的自增来得到订单号，所有的订单生成必须先在这里insert一条数据，得到订单号。分库后，库的数量变多，相应的故障次数变多，但由于单点的存在，故障影响范围并未相应的减少，使得全年downtime上升，可用性下降。

针对ID分配单点问题，考虑到数据库表分配性能的不足，调研了Tair、Redis、Snowflake等ID分配器，同时也考虑过将ID区间分段，多点分配。

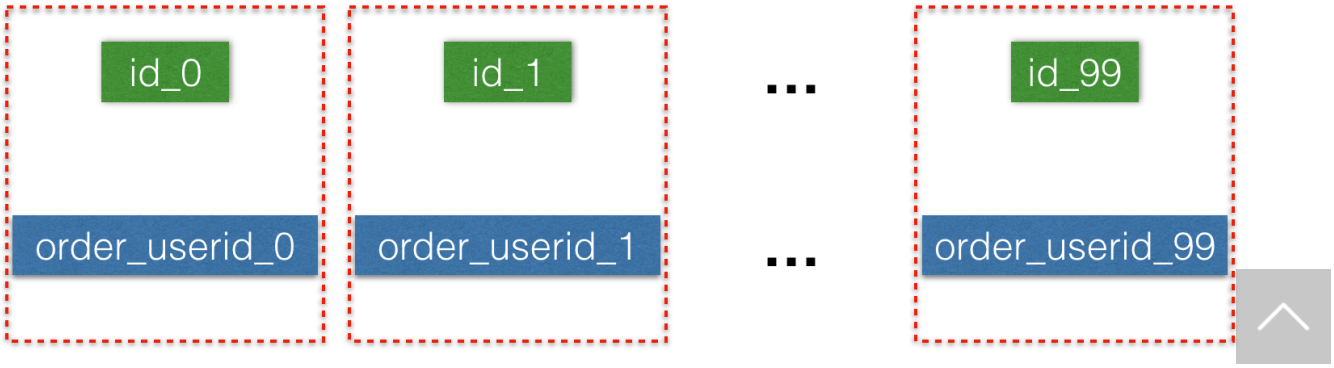
但最后没有使用这些方案，主要原因是ID分配对系统而言是强依赖服务，在分布式系统中，增加这样一个服务，整体可用性必然下降。 我们还是从数据库入手，进行改良，方案如下。

如下图，由原来一个表分配改为100张表同时分配，业务逻辑上根据不同的表名执行一个简单的运算得到最终的订单号。



ID与用户绑定：对订单系统而言，每个用户有一个唯一的userid，我们可以根据这个userid的末2位去对应的id_x表取订单号，例如userid为10086的用户去id_86表取到值为42，那订单号就42*100+86=4286。

将订单内容根据userid模100分表后如下图：

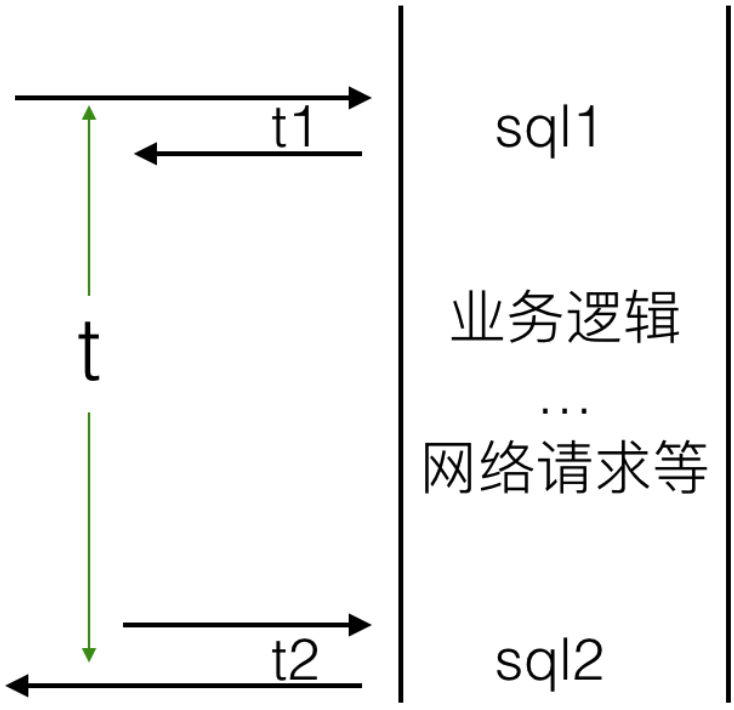


通过看上面的技巧，我们发现订单根据“userid取模”分表和根据“订单号取模”来分表结果是一样的，因为后两位数一样。到此，分库操作就相当简单容易了，极限情况下分成100个库，每个库两个表。同一个用户的请求一定在同一个库完成操作，达到了完全拆分。

注：一般情况下，订单数据分表都是按userid进行的，因为我们希望同一个用户的数据存储在一张表中，便于查询。当给定一个订单号的时候，我们无法判别这个订单在哪个分表，所以大多数订单系统同时维护了一个订单号和userid的关联关系，先根据订单号查到userid，再根据userid确定分表进而查询得到内容。在这里，我们通过前面的技巧发现，订单号末二位和userid一样，给定订单号后，我们就直接知道了分表位置，不需要维护关联表了。给定订单号的情况下，单次查询由原来2条SQL变为1条，查询量减少50%，极大提升了系统高并发下性能。

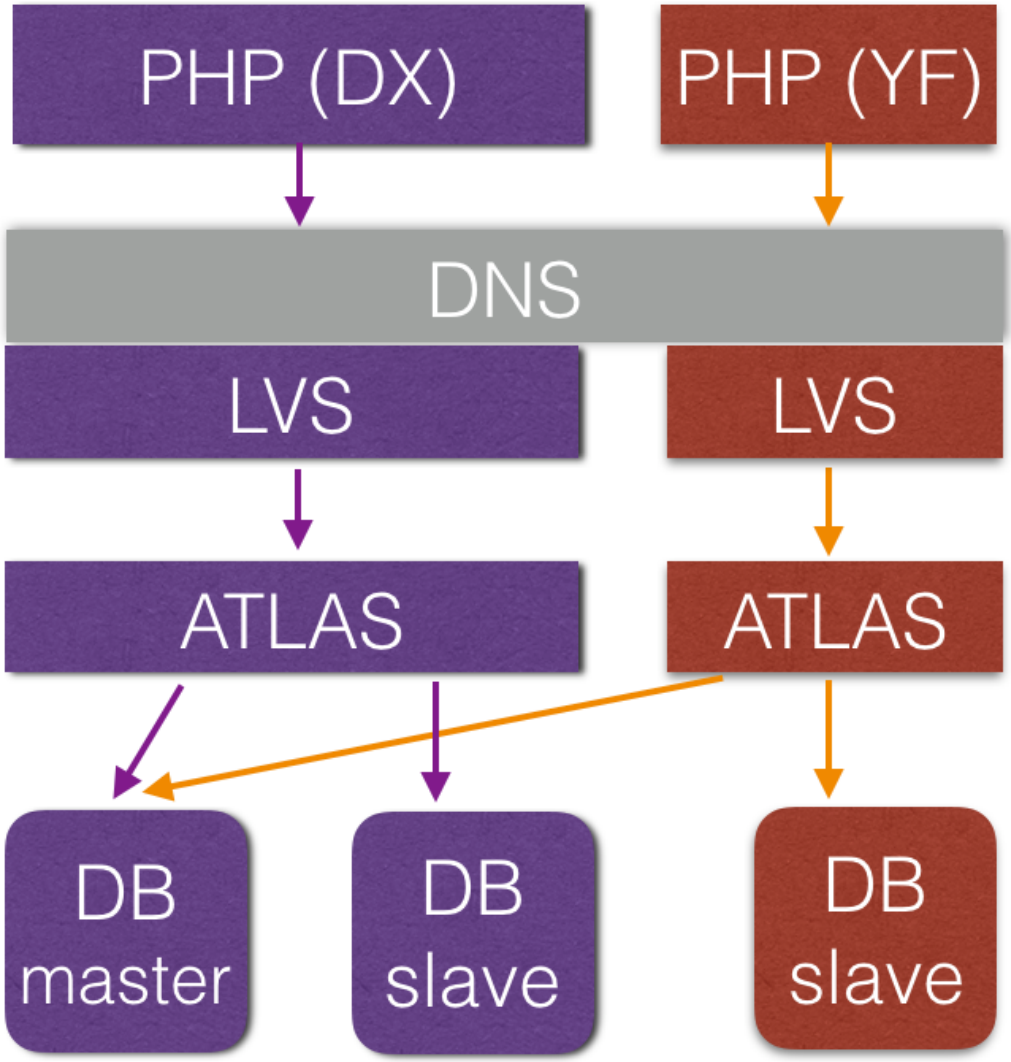
2. 传输优化

当时订单业务主要用PHP编码，直连数据库。随着前端机器的增多，高流量下数据库的连接数频繁报警，大量连接被闲置占用，因此也发生过数次故障。另一方面，数据库IP地址硬编码，数据库故障后上下线操作需要研发人员改代码上线配合，平均故障处理时间(MTTR)达小时级。
如下图：

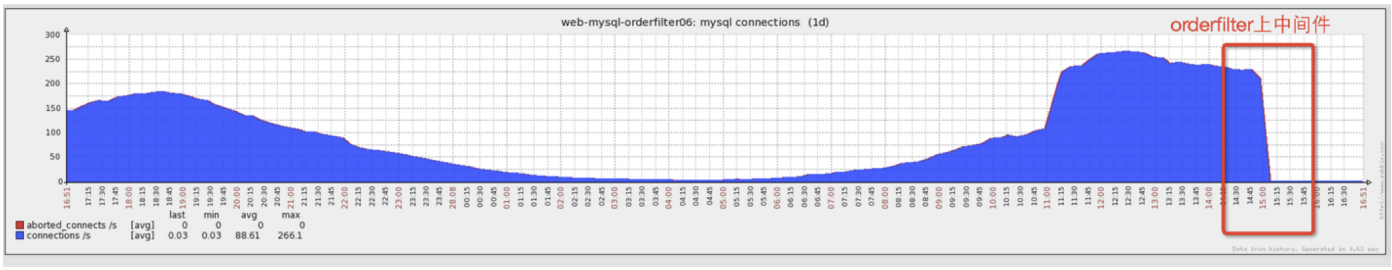


在整个业务流程中，只有执行SQL的t1和t2时间需要数据库连接，其余时间连接资源应该释放出来供其它请求使用。现有情况是连接持有时间为t，很不合理。如果在代码中显式为每次操作分别建立并释放资源，无疑增大了业务代码的复杂度，并且建立和释放连接的开销变得不可忽略。最好的解决办法是引入连接池，由连接池管理所有的数据库连接资源。

经过调研，我们引入了DBA团队的Atlas中间件，解决了上述问题。



有了中间件后，数据库的连接资源不再如以前频繁地创建、销毁，而是和中间件保持动态稳定的数量，供业务请求复用。下图是某个库上线中间件后，数据库每秒新增连接数的监控。



同时，Atlas所提供的自动读写分离也减轻了业务自主择库的复杂度。数据库机器的上下线通过Atlas层热切换，对业务透明。

3. 架构优化

经过前面两步的处理，此时的订单系统已比较稳定，但仍然有一些问题需要解决。如前面所述，50多个开发人员共享同一个代码仓库，开发过程互相影响，部署时需要全量发布所有机器，耗时高且成功率偏低。

在此基础上，结合业界主流实践，我们开始对订单系统进行微服务化改造。服务化其实早已是很热门的话题，最早有Amazon的服务化改造，并且收益颇丰，近年有更多公司结合自身业务所进行的一些案例。当然也有一些反思的声音，如Martin Fowler所说，要搞微服务，你得“Tall enough”。

我们搞微服务，是否tall enough呢，或者要进行微服务化的话，有什么先决条件呢？结合业内大牛分享以及我自己的理解，我认为主要有以下三方面：

- DevOps：开发即要考虑运维。架构设计、开发过程中必须考虑好如何运维，一个大服务被拆成若干小服务，服务注册、发现、监控等配套工具必不可少，服务治理能力得达标。
- 服务自演进：大服务被拆成小服务后，如何划清边界成为一个难题。拆的太细，增加系统复杂度；太粗，又达不到预期的效果。所以整个子服务的边界也应该不断梳理完善、细化，服务需要不断演进。
- 团队与架构对齐：服务的拆分应该和团队人员配置保持一致，团队人员如何沟通，设计出的服务架构也应一样，这就是所谓康威定律。

公司层面，美团点评平台主要基于Java生态，在服务治理方面已有较完善的解决方案。统一的日志收集、报警监控，服务注册、服务发现、负载均衡等等。如果继续使用PHP语言做服务化，困难重重且与公司技术发展方向不符，所以我们果断地换语言，使用Java对现有的订单系统进行升级改造。使用公司基础设施后，业务开发人员需要考虑的，就只剩下服务的拆分与人员配置了，在这个过程中还需考虑开发后的部署运维。

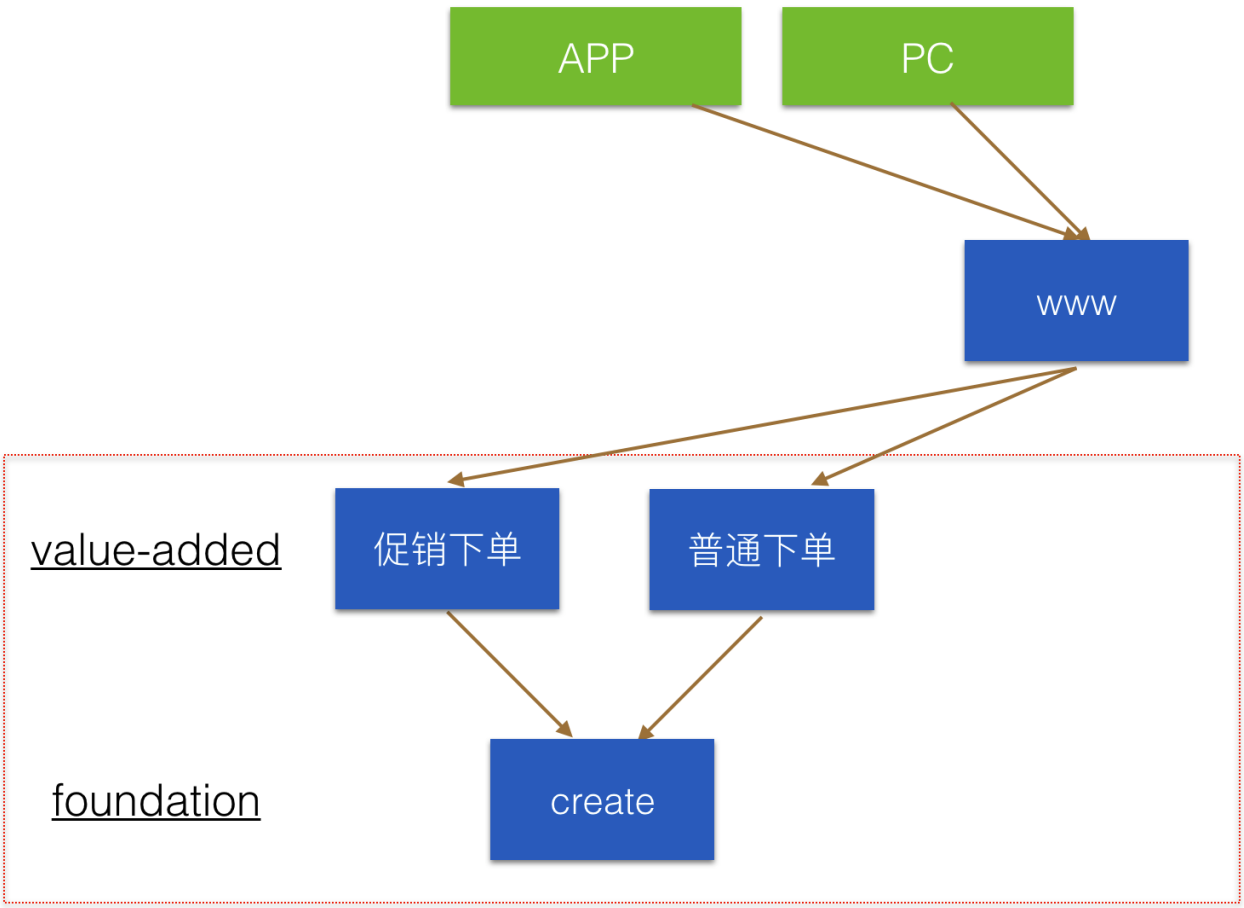
结合业务实际情况，订单核心部分主要分为三块：下单、查询和发券。

下单部分

由易到难，大体经过如下两次迭代过程：

第一步：新造下单系统，分为二层结构，foundation这层主要处理数据相关，不做业务逻辑。通过这一层严格控制与数据库的连接，SQL的执行。在foundation的上层，作为下单逻辑处理层，在这里我们部署了物理隔离的两套系统，分别作为普通订单请求和促销订单（节日大促等不稳定流量）请求服务。

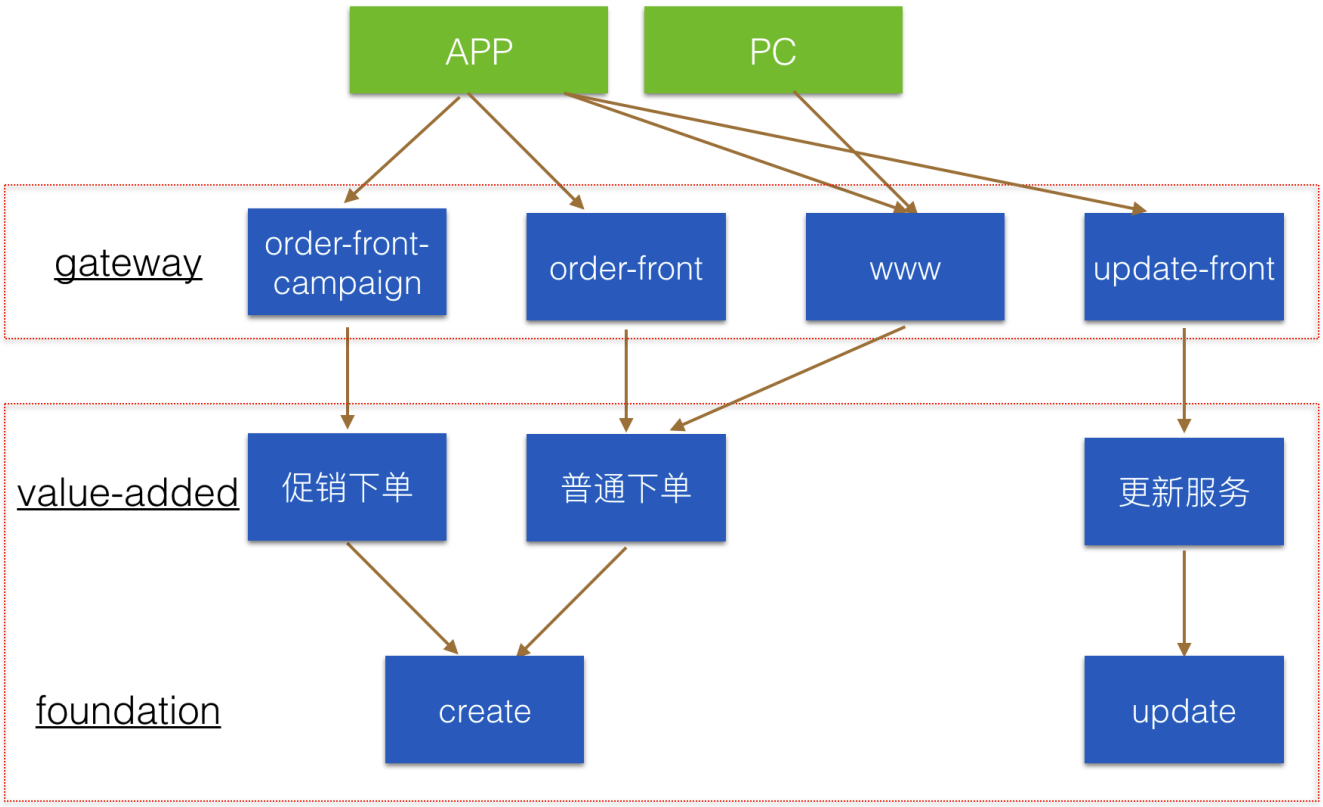
通过从原系统www不断切流量，完成下单服务全量走新系统，www演变为一个导流量的接入层。



(下单服务)

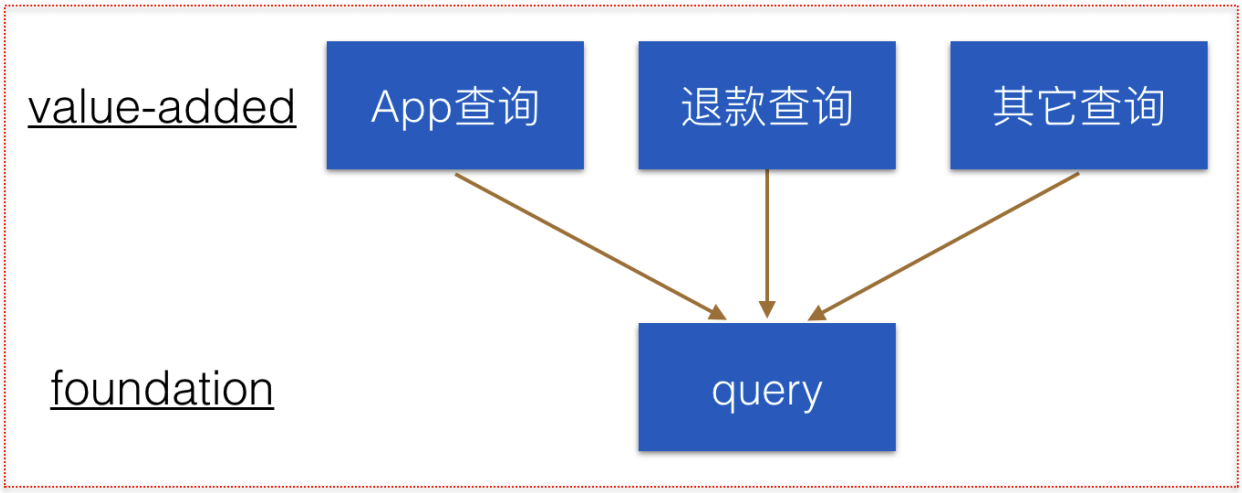
第二步：在上述基础上，分别为正常下单和促销下单开发了front层服务，完成基本的请求接入和数据校验，为这两个新服务启用新的域名URI。在这个过程中，我们推动客户端升级开发，根据订单发起时是否有促销活动或优惠券，访问不同的URI地址，从源头上对促销和非促流量进行了

隔离。

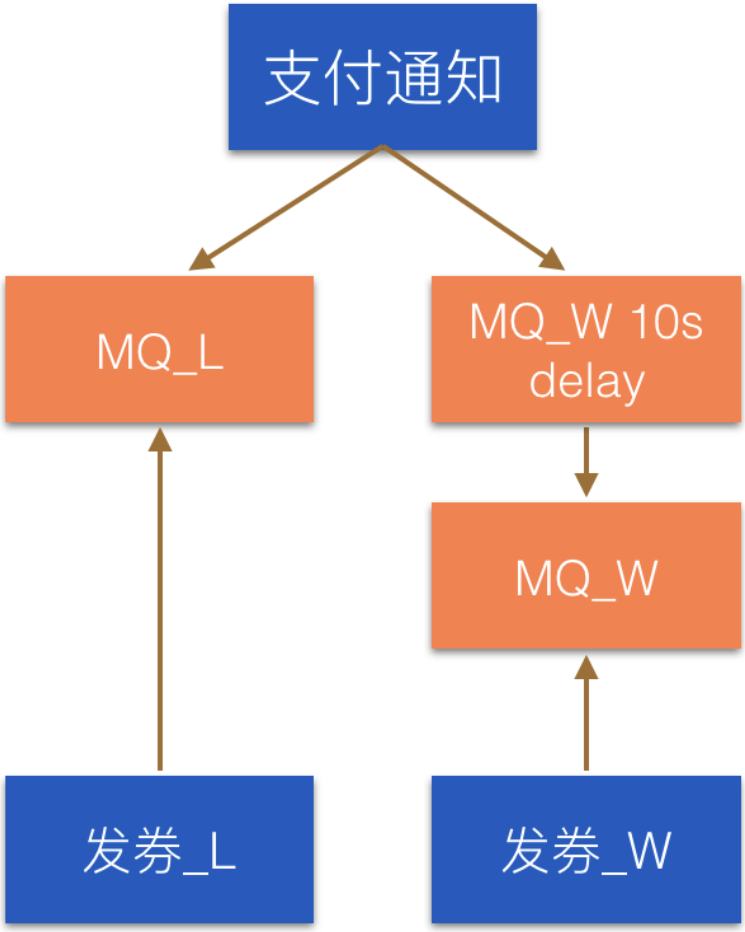


查询部分：

和下单部分类似，分为两层结构，上层根据不同业务请求和重要性进行了物理隔离。



发券部分：

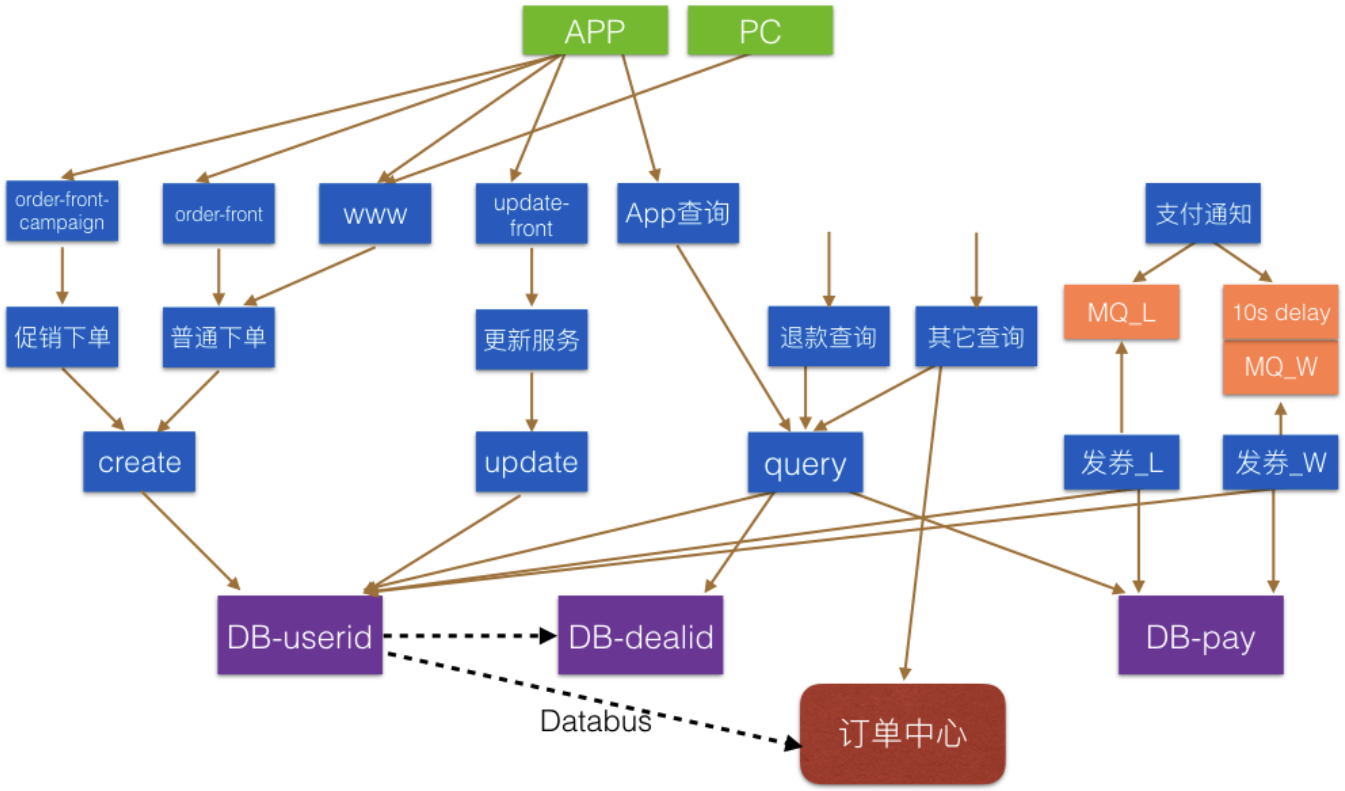


纵观发券业务历史上的一些故障原因，主要集中在两点：
一是消息队列本身出问题，连不上，数据不能投递，消费者取不到消息。
二是个别脏数据问题，消费者不断重试、失败，造成队列堵塞。

针对上述问题，我们设计了如图所示架构，搭建两组消息队列，互相独立。支付通知分别向L队列和W队列的一个10秒延时队列投递消息，只要有一个投递成功即可。

- 消息到达L队列后，迅速被发券L服务消费。发券L服务拿到消息后，先ack消息，再尝试进行发券，不论成功或失败，仅一次。
- 与此同时，相同的消息到达W的10秒延时队列后，经过10秒时间，被投递到MQ_W队列，被发券W服务拿到。发券W服务先检查此消息关联的订单是否已成功发券，若非，尝试进行发券，并完成一系列兜底策略，如超过30分钟自动退款等。

去掉一些细节部分，全景如下：



稳定性保障

目前，订单系统服务化已完成，从上述模块部署图中可以看出，架构设计中充分考虑了隔离、降级等容灾措施。具体从以下几个方面说明：

1. 开发、测试。相比于原来大一统的系统，彼此代码耦合、无法进行测试，服务化后，各个模块单独开发部署，依赖便于mock，单元测试很容易进行。同时我们搭建了稳定的线下环境，便于回归功能。
2. 蓝绿发布。这是无停机发布常见的一种方法，指的是系统的两个版本，蓝色的表示已经在生产上运行的版本，绿色表示即将发布的新版本。首先将两套版本的系统都启动起来，现有的用户请求连接的还是旧的蓝色版本，而新的绿色版本启动起来后，观察有没有异常，如果没有问题的话，再将现有的用户请求连接到新的绿色版本。目前线上服务发布均采用蓝绿发布流程，对用户无感知。
3. 多机房部署。按照整体规划，订单系统主要以一个机房为主，另一个机房作为辅助，按照2：1比例进行部署，提升机房故障容灾能力。
4. 促销与非促购买隔离。如上述下单部署架构图，我们推动App方，对于促销和非促流量，从源头上区别访问地址，达到物理隔离，做到互不影响。
5. 全流程去单点。除去数据库主库外，全流量无单点，弱化对消息队列的依赖，使用Databus进行数据异步复