饿了么：业务井喷时，订单系统架构这样演进

原创：石佳宁 InfoQ 2016-08-25

> 本文根据石佳宁在InfoQ举办的2016ArchSummit全球架构师（深圳）峰会上的演讲整理而成。

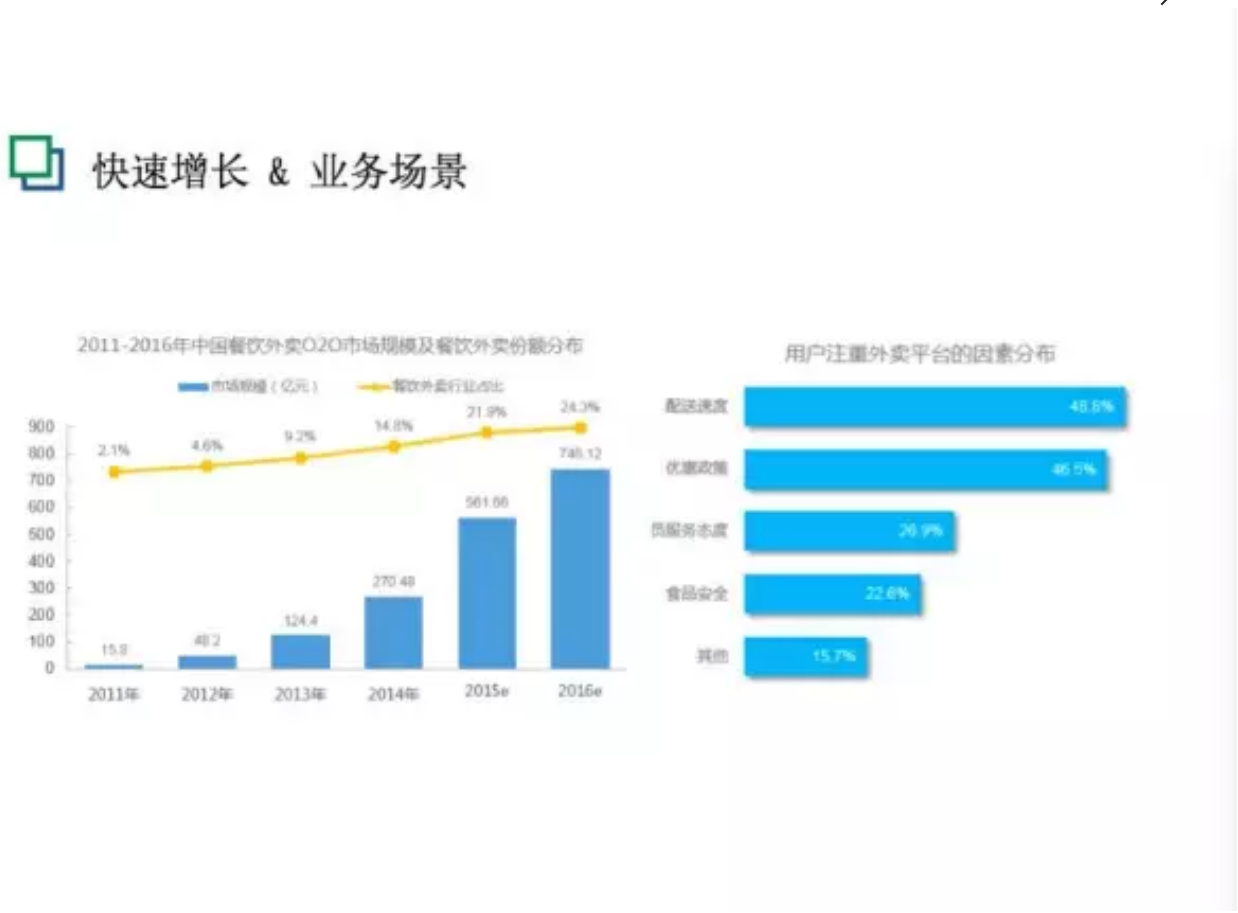
老司机简介

石佳宁，**饿了么后台支撑研发部负责人**，目前任职于饿了么,现任平台研发中心-后台支撑部门负责人,主要负责饿了么外卖订单、统一客服系统、BD销售以及管理工 具、代理商管理平台等系统的设计和研发工作。

先自我介绍一下，我于2014年加入饿了么，那时正是饿了么飞速发展的起始点。我一直从事后台领域的研发，比如BD系统、客服系统和订单系统，现在专注交易架构相关的工作。

今天要讲的内容主要分为两大部分。第一部分是在高速增长和愈加复杂的交易场景下，饿了么订单的服务架构是如何演进的，究竟是什么支撑我们的发展。

快速增长下的业务场景



具体讲之前，我先介绍一下我们的场景，因为**脱离具体的场景所有架构演进没有任何意义。**上面这两个图表不是饿了么的数据，是第三方分析整个外卖市场的数据图。左边的图表是从2011年开始，整个O2O市场以及外卖的份额逐年增加。2013年和2014年的时候发生了比较大的飞跃，饿了么也是在这个时间段订单量开始猛增。右边的图表是用户注重外卖平台的因素分布。

从图中可以看到，用户很在意配送速度，在意交易的时效性。对于O2O或者饿了么订单，交易的要求比传统电商的高，因为交易一般一两个小时就结束了。在2014年初，饿了么订单量只有日均10万单，到2014年底超过百万，这是一个质的飞跃，10万订单的量级和百万订单的量级的要求非常不一样。在2015年突破了日均300万，到今年5月单日峰值突破500万。

快速发展涉及很多问题。我们是一家创业公司，业务发展非常快，可能准备不是很充分，比如说监控、日志、告警、框架、消息、数据库，很多基础设施还在建设之中。在这个过程中出现一些问题是在所难免的，对系统的要求不是不能挂、不能出问题，而是出了问题要第一时间能恢复。这是整个系统架构的前提。

服务架构的演进

图中所示是订单的早期架构图，比较简单。这个架构在2014年的时候支撑了日均10万的订单，是一套很不错的架构，依然在很多系统中完美运行。但是对于后来发展的场景，它已经曝露问题了，比如业务逻辑严重耦合、代码管理很困难，因为数据库都在一起，操作变更很难追溯。

更进一步的是，性能的瓶颈只能是靠服务器去硬抗，从物理架构到逻辑架构，都已经成为业务发展的掣肘了。于是，为了业务的发展，我们做了一些演进的工作。

演进工作的核心就是一个字“拆”，跟“拆”对等的就是分治的思想。怎么拆分呢？面向服务有很多拆分原则。我将拆分过程中最具帮助和指导性的点罗列了以下几条。

- **第一是明确的定义。**之前也确实犯了一些错误，为了拆而拆。其实我们需要更明确，什么才算是一个服务？服务一定具有非常独立的技术能力或者业务能力，而且一定意义上能够很抽象。

- **第二是松耦合。**最基本的松耦合就是Customer的消费不依赖于Provider的某一个特定实现，这样服务器的内部变更不会影响外部消费，消费者可以切换到其他服务能力的提供方，这是最基本的松耦合。还有时间上的松耦合或者位置上的松耦合，我们希望的松耦合是消费方和服务方是可以分离的。

- **第三是基于领域的认知，这对于整个产品起到非常大的作用。**因为当时整个饿了么所有系统是在一起的，基于领域的认知，在面向用户的维度和面向商户的维度做了切分，还有基于交易链路做了切分。

- **第四是单一职责和关注分离。**简单说，我们希望一个服务或者一个模块拥有单一的能力，而不是承担过多的职责，否则责任不清晰，导致能力也不清晰。

- **最后一点是可被验证的结果。**在订单拆分的过程中我们犯了一些错误，当时认为这样拆分是没有问题的，但是过一、两个月，并没有带来效率和能力的提升，反而是跨团队的要求越来越多，能力要求也越来越多。这时候可能是拆错了。如果是一个好的拆分一定有利于发展，拆分之后的发展是更迅速的。

基于这几条原则，我们对饿了么的整体服务做拆分之后，如上图所示，架构就有了一些变化，看起来跟刚才架构区别不大。把Order Service做了分离。当时拆分虽然比较垂直，但是用户、商户、金融、订单等还是有一些横向交互。

一个接口有一个非常明确的Owner，一个表、一个库也能保证仅有单一的操作方，让我感受比较直接的是，为服务的治理奠定了基础，以后可以针对某项特定业务做一些降级、熔断，以及单独的监控。拆分实际上是让各自模块的掌控力变得更强了，对业务起到更好的支撑作用。

这时每个部门或者每个团队都负责自己独立的领域，代码和数据都拆分完毕是不是就可以了？但是后来发现好像还不对。因为虽然大的领域上确实已经干净了，但是在小的领域上依然问题很多，订单并不仅仅只有一张表，一个单一的模块，其实还有很多复杂的内容。

在一些技术工作上，这些问题曝露得并不是那么明显，那时候大家对于一些领域认知或者业务边界的认识还是模糊的，没有人界定这些。但是当更进一步地去发展一个领域的时候，还是会有职责不清晰或者能力模糊的地方。我们思考，还要基于业务进行更细腻的规划。

于是我们把订单本身做了一些业务层次的拆分，拆分之前首先要确认订单到底在整个系统中，尤其是交易系统、O2O系统中承担什么角色，担负什么职责。

在这个思考过程中，我们的认知大概是以下四点。

第一，订单是整个交易链路的核心，围绕了一些相关服务，比如金额计算服务、催单服务、售中异常服务等，我们希望这些服务之间有明确的区别。

第二，订单实时处理是整个链路的中心，我们将这个过程定义得尽量简洁。一笔交易中，订单被推进得越复杂，说明系统设计得越复杂，出问题的概率也会越高。所以我们希望订单核心流程非常简单、轻薄，把复杂的东西剥离出来，把简单和复杂明确成两个部分。

第三，考虑到交易的时效性和异常场景越来越复杂，**将交易分成正向交易流程和逆向交易流程两个部分**。正向交易流程，99%的订单会根据这个流程走完生命周期；逆向交易流

程，比如说退单要求时效性比较低，处理会牵扯多方业务可能很复杂，所以通过一个逆向的交易流程来解决。

第四，能够在功能和业务上独立的部分，尽可能抽象为单独的模块或服务。简单来说，比如催单的服务，它其实对交易链路无法起到推进作用，它只是一个动作或者附带服务，我们把它单独抽象出来，为后面的发展做出铺垫。

基于这些之后，我们对订单进行完整的认知，对订单的服务架构和业务架构做成图中的样子，大概是三层。下面一层是基本数据；中间层是正向逆向的流程、最核心的状态和最关联的交易链上耦合的服务；上层是用户服务、商户服务，包括跟交易链相关的，比如饿了么最近推出的“准时达”的服务。

我们同时对其他服务模块也做了演进。一些是之前设计的不合理，如图所示是当时缓存服务的逻辑架构，节点比较多。简单解释一下最初的做法: 提交订单的时候清除缓存，获取订单的时候如果没有缓存的话，会通过消息机制来更新缓存。中间还有一个Replicator，起到重复合并的作用。

后来我们发现，本来可以轻量级实现的内容，但是用了相对复杂的实现，链路长，组件多，受网络影响非常大。一旦一个节点缓存数据不一致，感知会比较困难，尤其是业务体量大的时候。

业务体量小的时候同时处理的量并不多，问题曝露并不明显，但是体量变大的时候，这个设计立刻带来很多困扰。所以我们对缓存做了简化，就是把不必要的内容砍掉，做一个最基本的缓存服务。

这是一个最基本的缓存的套路，在数据库更精准的情况下更新缓存，如果从DB获取不到就从缓存获取。这个架构虽然简单了，但是效率比之前高很多，之前数据库和缓存之间延迟在200毫秒左右，而这个简单实现延迟控制在10毫秒以内。

之前订单最大的瓶颈是在数据库，我们主要做了DAL中间层组件。图中这个中间件对我们影响非常大，日均300万单的时候数据库量比较大，引入DAL中间件做什么呢？有几个作用：数据库管理和负载均衡以及读写分离，水平分表对用户和商户两个维度做评估，为用户存储至少半年以上的数据。解决了数据库的瓶颈，系统整体负载能力提升了很多。

这张图说明了订单具体改造的时候DAL中间件起的作用，有读写分离端口、绑定主库端口、水平分表、限流削峰以及负载均衡等功能。

监控和告警的峰值非常明显，午间和晚间两个高峰，其他时间流量相对平缓。下面主要讲三个部分。

第一，对于订单而言，吞吐量是最需要重点关注的指标。一开始对业务指标的感知并不是特别清晰，就在某一个接口耗费了很多时间。后来发现一些很小BD的问题不太容易从小接口感知到，但是从业务方面感知就比较明显，所以就更多关注业务指标的控制。

第二，通常我们重视系统指标，而容易忽视业务指标，其实业务指标更能反映出隐晦的问题。

第三，目前我们致力于基于监控和数据学习的过载保护和业务自动降级。虽然现在还没有完全做好，但是已经能感觉到一些效果。如果商户长时间不接单，用户会自动取消订单，自动取消功能的开关目前是人工控制的，我们更希望是系统来控制。

比如说有大量订单取消了，有可能是接单功能出了问题，就需要临时关闭这个功能，如果还是依靠人来做，往往已经来不及，这时候就应该基于数据的学习，让系统自动降级这个功能。

当做完这一切，订单的架构就变成了上面这个样子。我们把整个Service集群做了分组，有面向用户的、面向商户的，还有物流和其他方面的。

Design for failure

就订单系统而言主要有以下四个内容。第一是消息广播补偿，第二是主流程补偿，第三是灾备，第四是随机故障测试系统。

首先是消息广播补偿。对于订单来说，MQ是非常核心的基础组件，如果它出现问题，一些订单处理就会受影响。为了避免这种情况发生，我们做了一个补偿的内容，这个补偿其实很简单，就是在订单状态发生消息变化的时候，我们会同时落一份消息数据，目前会存储最近一小时的消息。

如果MQ系统或者集群当前有问题或者抖动，消息广播补偿可以起到一个备线的作用。消息广播补偿不能应付所有问题，但是对于订单系统的稳定和健壮而言还是非常有用的。

第二是主流程的补偿。什么是主流程？就是交易的正向流程。99%的交易都会是正向的，就是下单、付款，顺利吃饭。在这个过程中，只要用户有通过饿了么吃饭的意向，就尽全力一定让他完成最终的交易，不要因为系统的原因影响到他。

主流程主要是针对链路本身出问题的情况，以最大程度保证交易的进行，也是对主要链路的保护。

比如有一次出现这个问题：用户已经支付过了，但订单没有感受到这个结果，订单显示还在待支付，当时支付服务本应该把结果推送过来，订单就可以继续往前走，但是系统在那里卡住了，这对用户就是比较差的体验。

所以后来我们做了主流程的补偿，以确保交易的信息链路一直完整。我们的原则是，对订单的各个状态变更进行推送或拉取，保证最终的一致性，链路和介质要独立于原流程。我们从两个方面来解决这个问题。

在部署方面，把提供补偿功能的服务和主服务分开部署，依赖的服务也需要使用独立实例，以保证高可用性。在效果方面，用户支付成功前的所有信息都应该尽量入库，可以对支付、待接单、接单等一系列环节都可以做补偿。

这是主流程补偿的图，最大的关联方就是支付和商户，支付就是代表用户。商户有推送订单信息，支付也有推送订单信息，如果出现问题，补偿服务可以拉取结果，订单甚至可以自动接单。这个补偿经过多次的演变，目前依然在运作，对于一些比较特殊的情况还是很有用的，可以在第一时间处理问题，保证交易的完成。

第三是灾备。目前订单系统做了一个比较简单的灾备，就是两个机房的切换。切换的时候是全流量切换的，我们会把流量从A机房切到B机房。订单的主要操作是在切换的过程中要进行修复数据。

比如，一些订单开始是在A机房，被切换到B机房去操作，这就可能会造成两个机群数据不一致的情况，所以会专门对信息做补全，当一笔交易切换到另一个机房后，我们要确保短时间内将数据对比并修复完成，当然主要还是确保数据最终一致。

第四是随机故障测试系统。左图是Netflix的猴子家族，右图是我们做的Kennel系统，一个是猴子窝，一个是狗窝。大家对猴子家族了解吗？Netflix现在几乎把所有内容都部署在云上，对系统和架构的要求很高，他们可以随时破坏一些节点，以测试是否能依然为用户提供服务。

我们也参考他们的做法，有很大的启发，**避免失败最好的办法就是经常失败。**饿了么的发展速度比非常快，技术还不完善，设计也会有缺口。我个人觉得，一个好的系统或者好的设计不是一开始被大牛设计出来的，一定是随着发展和演进逐渐被迭代出来的。

参考了Netflix的猴子家族，我们研发了自己的Kennel系统。猴子家族主要是针对节点的攻击，我们的Kennel主要是对网络、内存等做了调整，还结合自己的服务，对应用和接口也做了一些攻击。攻击分两部分。

第一部分是物理层面的，我们可以对指定节点IO做攻击，或者把CPU打到很高；对于服务和接口而言，可以把某个接口固定增加500毫秒或者更要的响应延迟，这样做的目的是什

么？在整个链路中，我们希望架构设计或者节点都是高可用的，高可用就需要被测试，通过大量的测试人为攻击节点或者服务，来看预先设计好的那些措施或者补偿的能力是不是真的有用。

整个Kennel的设计是，首先会有一个控制中心来做总的调度，配置模块可以配置各种计划，可以控制CPU或者网络丢包等，可以设置在每周六8-10am的某个时间点攻击系统十五分钟。它还有一些操作模块，比如执行计划模块、任务执行模块、节点管理模块、执行记录模块等。

Kennel有四个主要的作用。

首先，帮助我们发现链路中隐蔽的缺陷，将小概率事件放大。比如说缓存不一致的问题，之前极少出现，一旦出现之后，处理手段比较缺乏，那就可以通过Kennel来模拟。网络的抖动是很随机的，那么Kennel可以在某个时间段专门进行模拟，把小概率事件放大。如果怀疑某个地方出了问题，可以通过它来测试是不是真的能查出问题。

第二，重大功能可以在发布之前通过其进行测试，迫使你更深入地设计和编码。通过模拟流量或者线上流量回放，来检验系统运行是否如你设计那样工作，比如监控的曲线或者告警以及相关服务之间的依赖等。

第三，我们做了很多失败的准备和设计，要看到底会不会起作用、起多大作用。可以通过Kennel进行校验，在某个时间通过随机手段攻击相关服务，服务方不知道具体的攻击内

容，这时原本设定的监控告警，降级熔断等措施有没有及时起作用就是一个很好的校验。同时还可以检验之前准备的容错或者补偿措施是否能按照预期工作。

第四，需要验证FailOver的设计，只有验证通过才可以依靠。所有的设计都是经历了一次一次的失败，一些设计原以为有用，但是真实问题发生时并没有起到作用。真正有意义的FailOver设计一定是经过验证的。

【百川解码】8月27日20:00-21:00，百川解码带你一起和技术大咖直接交流，一起聊聊那些年热修复带来的坑，解析最IN的阿里多平台热修复方案！

扫码加入直播交流群，戳阅读原文了解更多详情

▽

延展阅读（点击标题）：

- 程序员简易成长指南：从菜鸟码农到架构师
- 拍摄纸牌屋的Netflix为何要迁移数据库入云？
- 过去十年，编程语言领域有什么重要进展？

喜欢我们的会点赞，爱我们的会分享！

阅读原文