饿了么技术往事（上）

原创 黄晓路（脉坤） 阿里巴巴中间件 2020-10-23

收录于话题

#饿了么 2 #成长&思考 8

作为一个互联网创业公司，饿了么从初创到壮大，在移动互联网时代，业务量和技术团队的体量经历了10倍增长，这其中的经历，是互联网领域许多创业公司技术团队的一个缩影。在这里把我们成长过程中的体会和教训记录下来。

饿了么的技术体系，经历了以下四个阶段：

1. 核心系统 All in one 的早期架构；
2. 以系统领域化拆分、业务系统和中间件等基础设施分离为基础的全面服务化的架构；
3. 随着自动化平台、容器调度体系成熟，治理从传统运维向 DevOps 转变的基础设施体系；
4. 多数据中心体系基础上的Cloud Ready架构成型。

在这期间，业务的快速增长，大大小小的事故，组织结构的变化、融合，团队的工程师文化和技术栈背景，不同技术理念的冲突，交织在一起，互相冲击、影响着架构变迁。

第一阶段：All in one

这是饿了么技术体系早期的样子，技术联合创始人带着一群 Geek，从0到1，搭建了最早的技术体系，支撑了百万级的单量。

这个阶段，业务一路狂奔，技术拼命追赶业务，唯快不破。

技术栈以 Python 为主，兼有部分 PHP 的系统，单机多应用的混布模式，应用发布、系统运维基本上是开发工程师敲命令行完成。核心的业务系统，商户、用户、交易都共享一个 codebase ，建立在一个名为 zeus 的系统下。短时间内业务飞速增长，在线数据库已经支撑不了 ETL 的需求，大数据体系、数仓开始建立。

大数据所在的上海数据中心，在线业务系统所在的北京数据中心开始搭建，这两个数据中心见证了我们架构的变迁，直到后来整体上云，最早的时候，技术团队40多人，有时候需要创始人跑到机房里面搬服务器。

系统跟不上业务发展速度的时候，核心系统经历过一些间歇性宕机的尴尬阶段。一些刚刚开始开拓的业务系统，也经历了系统刚上线就连续宕机，不得不临时放慢业务的阶段。但是这个过程也有收获，很多开发工程师线上排障能力非常强，脚本玩得很溜。

这个时期的工程师经常一人身兼数职，前端、后端、开发、测试、运维部署，对业务的理解也很深刻，甚至身兼技术和产品的角色。

收获和教训——文化的重要性

早期饿了么有一个鲜明的特点，就是大家都非常有担当，开放且包容。推卸、逃避责任的事情很少发生。虽然当时很多事故回过头来看比较严重，但是，组织对技术人员成长中的错误也相对包容。整个团队比较扁平，上下级之间的技术争论是常有的事情，但是都能就技术论技术。

饿了么的工程师文化氛围还是挺强烈的：工程师想着服务器的资源利用率能不能再压榨一下；在决定大规模使用Redis之前，会去读Redis的源码；很多方案，都是找个吧台和白板快速讨论，快速达成共识，落地上线。可能也是这个氛围，吸引到很多相同味道的人，形成了技术团队的文化。技术团队创始人最初形成的文化能延续下来，是这个团队从最初的几十人发展到上千人，还保有凝聚力和执行力的基础。

第二阶段：拆迁和基建

技术系统架构影响了业务交付效率，那么就需要重构甚至重建系统；如果是组织结构的不合理，阻碍了系统架构的迭代，成为业务发展瓶颈，那么就需要调整组织结构。

打个比方，如果说业务系统是赛车，那么基础设施就是跑道。基础设施也是这个阶段我们建设的重点，为承接将来业务快速增长打下基础。

这个阶段我们面临几个问题：

1. Python为主的技术栈，现有的工程师单兵作战能力都很强，但是当时市场上的兵源严重不足。

2. All in one的系统，各业务领域没有划分，业务模块之间代码交错，影响到了交付效率，需要给业务快速发展松绑。

3. 基础设施和业务系统开发没有分开，身兼数职的开发工程师，在基础设施运维、中间件开发、前后端业务系统开发各个方面，各有长短。

4. 传统的手工上线、部署、运维、监控模式 —— SSH到服务器上手工执行脚本，效率低下，事故发生时恢复时间长，发布后难以回滚。

成建制

随着业务量迅速增长，业务系统日渐复杂，技术团队也随之扩张，当时人才市场上 Java 工程师还是比 Python 的工程师来源多，有更大的选择余地，因此，以 Java 为主的多个领域开始逐步壮大，形成了 Python 和 Java 两大技术栈体系。

大前端、移动端、多个领域的后端业务应用、运维、中间件、风控安全、大数据、项目管理等多个角色分工，不同角色的工程师做自己擅长的事情。在这个阶段，系统和组织形成了业务开发、运维、共享组件及中间件几个体系架构。

业务系统

1. 业务领域划分

单体架构的系统开始按照领域拆分。All in one 的系统，通过划分业务领域，各个领域的技术骨干认领掉所负责的领域，组织结构相应调整，才很艰难的完成了划分。导购、搜索推荐、营销、交易、金融、公共服务、商户商品、商户履约、客服，新建的物流运单、分流、调度等系统，大数据数仓，逐步识别出自己领域、子领域及相应模块。在这个过程中，也有一些骨干在接手所负责领域后，没有在第一时间重视人力资源，导致交付能力不足，成为业务发展的瓶颈。从技术骨干再到技术团队负责人这一转变过程中，很容易被忽略的就是团队的人员结构。

2. 系统拆分

随着各自领域内的需求快速迭代，系统也迅速膨胀，相互之间的依赖、领域边界也开始变得错综复杂。原来可以“闭环”的，现在需要交互了，甚至原来可以直接动其他领域的代码提PR，访问别人的数据库，现在都不行了。从单体到领域切分服务，改变原有的思维方式，导致很多不适应，也走了一些弯路：导购域为了性能，自己落了一份商户商品数据缓存供商品查询，从而需要理解商户端领域的业务，订阅这类主数据的变更，而商户端的这部分数据就没有办法收口，缓存的新鲜度保障、底层数据结构变更、系统重构都比较麻烦；交易域麻烦也不少，一些领域为了不依赖交易，自己冗余了大部分的订单数据；物流履约领域则是下游存在多份运单数据冗余，导致领域职责没有收口，带来很多一致性问题，系统的复杂度也随之增加，系统交互和沟通成本也上升了。

而另一个极端，则是系统拆得过散，频繁的互相调用依赖，把本该高内聚的系统拆成低内聚了。订单和物流都经历了过度的异步化带来的痛苦，故障恢复时间过长，复杂性和排障成本增加。这段时间，领域边界的分歧也是一个头疼的事情。

体会和教训——康威定律、技术文化

订单履约体系里面，有一个隶属于商户域的团队，负责推单系统，该系统主要职责是承接商户呼叫配送，并将订单推送到物流运单中心。因为想减少对订单系统的依赖，以防订单系统挂了，无法履约，开发团队冗余了很多订单的数据，为此，要同时考虑订单和运单的正向、逆向，自身的可用性，系统也被设计得比较复杂。

在这个过程中，一旦有项目涉及到物流和推单系统交互，两个团队就经常发生领域边界的分歧，涉及到一些从订单取数的场景，物流团队认为，“这部分逻辑我不应该理解，你们应该从上游系统取到推给我们”，而这个隶属于商户域的团队则认为，“这部分不是推单的领域内的数据，而且推单系统自己也用不到，物流需要应该自己去取”。

类似问题反复出现，反复讨论，消耗掉大家很多时间，而且可见的将来还会发生。链路过长也带来稳定性隐患。最终，负责订单域的团队，来负责推单系统，订单领域内的逻辑可以闭环，这个问题就迎刃而解了。

所以，涉及到两个领域边界的时候，一旦相似问题反复出现，我们可能要考虑一下康威定律。

关于争论：系统设计阶段的激烈争论是非常合理的，充分的讨论会大大减少方案出现硬伤的概率，开发阶段也少受返工之苦。技术讨论围绕技术合理性，就事论事的展开，不要因为技术以外原因推脱或者权威说得算，讨论完大家才能很坦然的接受决定，最重要的是，参与的工程师都能充分理解最终方案的

利弊和取舍，落地不会有偏差，出了问题不推诿。这也是很多团队技术氛围吸引人的一个地方，文化不是口号，是日常的这些细节和实践。

运营体系

1. 团队

负责软件交付的业务运维、负责底层操作系统和硬件交付的系统运维、负责数据库的DBA、稳定性保障团队，相继成立。

2. 监控告警

集群实例的数量急剧扩张，登录到服务器上查看日志的运维模式已经不现实，也被基于遥测的监控体系所替代。随着7*24小时的NOC团队（故障应急响应）建立，基于ELK的监控体系也搭建起来，将核心指标投到了监控墙上。

体会和教训：监控告警机制的意义

互联网应用到了一定复杂度以后，庞大的集群，尤其容器化之后，IP动态分配，日志数量巨大，日志数据繁杂而离散，监控和排障需要借助的是和卫星排障一样的思路——遥测。日志系统要支持聚合和查询，监控需要实时收集采样各种指标、监控面板可以随时查看系统当前健康状况、告警机制第一时间发现系统冒烟。

有一次出现了一个问题，从监控面板上看一切正常，后来发现根因是一个int32 溢出的bug，导致下单失败，但是，为什么监控看不出来？因为代码里面把异常吞掉了，调用返回成功，而我们核心指标用的是下单接口的成功调用量指标和一些异常指标。

经过关键指标的治理，我们的监控面板关注三类核心指标：

- 业务指标 —— 实时关注业务整体的健康状况，从这个指标可以很直观的看到，业务的受损程度、时长和影响面，比如单量什么时候掉的、掉了多少比例、掉了多久，关键页面访问的成功率怎么样。对于上面订单的案例而言，需要在下单成功后打点（由负责订单落库的实现逻辑来完成），确保真实反映业务状况。需要注意的是这类指标通常涉及到敏感的业务信息，因此需要做一些处理。

- 应用指标 —— 关注应用实时的健康状况，应用本身及直接上下游的调用量、时延、成功率、异常等。为了安全起见，应该注意敏感系统信息不露出。特别是涉及到金融、安全领域的业务系统。

- 系统指标 —— 关注中间件、操作系统层面的实时健康状况，Network Input/Output、CPU load & Utilization、Memory Usage等。故障发生时，这些指标通常会先后发生异常，需要关注哪个是因哪个是果，避免被误导。

当然以上还不够，监控还有一些需要关注的地方，随着业务的发展，对我们的监控系统带来更多挑战，后面一个阶段，监控体系会有翻天覆地的变化。

这个事故给我们的另一个教训就是，关键路径和非关键路径的隔离：那个int32 溢出的bug是在非关键路径上触发的，当时还没有梳理出来关键路径，所以非关键路径故障影响到关键路径的事故时有发生。随着关键路径的梳理，后续服务降级的能力也才开始逐步建设起来。

3. 业务运维

业务运维团队担负起了很多业务系统运行时环境的初始化工作，比如虚拟机、HAProxy、Nginx、Redis、RabbitMQ、MySQL这些组件的初始化工作，容量评估工作。稳定性保障工作也是主要职责之一。这些分工让开发工程师可以更专心于业务系统的交付，但也是一把双刃剑，这是后话了。

4. 系统运维

随着专业运维团队的建立，系统从物理机迁移到了虚拟机上，单机单应用（Service Instance per VM）是这个阶段的主要部署模式。硬件设备的运维，网络规划，慢慢都更专业规范起来，CMDB也开始构建。

5. DBA/DA

数据库的运营统一收口到这个团队，负责数据库容量规划，可靠性保障，索引优化等等，交付了包括数据库监控体系在内的很多数据库运营工具产品，与此同时，他们也参与到业务系统的数据架构设计和评估选型当中。

6. 制度

故障等级定义、架构评审机制、全局项目机制也相继出炉。制度的建立、执行和以人为本，三件事情，从来难统一，得不到人的认可，则执行会打折，背离制度设立初衷，所以，制度也需要迭代。制度是底线，制度覆盖不到的地方靠团队文化来维持，但是，如果把文化当制度来执行，就得不偿失了。

体会和教训——架构师到底要做什么？

架构师是一个角色，而不是职级。这个角色的职责包含但不仅仅限于以下方面：

- 业务系统的技术方案设计和迭代规划
- 非功能需求的定义和方案设计、技术选型
- 现有架构的治理、领域边界的划分、设计原则与现实（技术债）的取舍和平衡
- 架构未来的演进

但是，如果不够深入，以上很难落地。从事前设计，到事中交付阶段的跟进，事后线上系统运营及反馈、持续的优化迭代，都需要架构师充分主导或者参与。

敏捷开发，设计是很容易被忽略掉的一环，制度上，我们通过组织由运维、中间件、业务开发、数据库、安全风控各领域的架构师组成的评审会议，在算力和基础设施资源申请通过前，审核评估设计方案。

实际上，这个上线前的“事前预防”也还是有局限性，因为这个时候设计方案已经出来了，虽然设计文档评审前已经提前提交，但是没有能够参与到整个生命周期，深入程度有限，只能做到兜底。更好的机制是每个团队都有架构师的角色，在设计过程中充分参与，这才是真正的“事前”。

所以，很长一段时间，跨领域的重点项目或者整个技术中心的项目，架构师才会主导或者相对深入的介入，而日常迭代，在设计方案、领域边界各方有重大分歧的时候，架构师往往是被动的卷入，变成居委会大妈的角色。覆盖的领域太广，架构师这个时候成为了瓶颈。而业务系统的架构恰恰是由日常的迭代逐步构建而成。后来全局架构组成立，架构师才真正开始分领域深入到更多业务的日常交付当中：

- 设计方案的负责人，作为Owner，为方案负责，并跟进交付，有权力有义务；

- 构建影响力。除了日常和开发工程师一起并肩交付，没有捷径，这个需要比较长的时间和项目的积累，很难一蹴而就，这个过程中如果得到一线工程师和开发经理的认可，影响力就会逐渐形成，有更强的推动力，反之，则会逐步丧失影响力。架构师如果没有影响力，一切无从谈起。因为组织上，架构师不一定在一线开发工程师或者开发经理的汇报线上。中立客观的态度，开放的心智，也是构建影响力的关键。影响力是架构师 Leadership 的体现。

- 稳定性的保障，永远是架构师的核心关注点之一，稳定性指标是架构师必须要背负的。其中包括交付质量、可用性、弹性、系统容量等等，不深入到具体领域，很难去提供保障。为此，架构师需要有调动资源保障稳定性的权力。

- 技术文化的影响，架构规划的贯彻，除了常规的宣讲、分享以外，更为重要的是日常项目交付、技术讨论过程中的潜移默化。设计思想、设计原则、技术文化认同，这些不是说教能形成的，是在一个个项目迭代、一次次线上排障、一场场事故复盘会中达到共识的。

在以上几点的基础上，才能够在全局上担负起架构的职责，推进架构的演进。敬请期待下周内容：饿了么全面服务化的架构实现和挑战；容器化实践与 DevOps 转变等。

作者：黄晓路（花名：脉坤），2015年10月加入饿了么，负责全局架构的工作。

收录于话题 #成长&思考·8个

下一篇 · 语雀的技术架构演进之路

阅读原文

喜欢此内容的人还喜欢

云原生消息、事件、流超融合平台——RocketMQ 5.0 初探

阿里巴巴中间件