每个工程师都应该了解的：系统拆分

2017-12-15 朱赟

每个工程师都应该了解的：系统拆分

朱赟

00:02 / 16:25

四年前，我加入了风头正劲的 Square 公司。两年前，我又加入了涨势甚猛的 Airbnb 公司。在我加入的时候，这两个公司都有上百名的工程师，网站和主要产品的核心功能也已齐备。

两家创业公司从 0 到 1 的创业过程我并没有亲身经历，但是两次都恰好经历了公司从 1 到 N 的扩张过程和业务拆分的过程。

今天我就和你聊聊，公司从 1 到 N 发展过程中的系统拆分问题。

创业初期的代码现状

在 Square 刚刚起步的时候，整个产品都是基于 Ruby on Rails 构建的，所有的产品和功能代码几乎都在一个代码库里。

等到我进入 Square 的时候，有一些服务已经从 Ruby 代码中分离出来了，形成了单独的 Java 或者 Ruby 服务，然而大部分功能还是在一大块 Ruby 代码里。

当时，几乎所有的工程师每天都在这一份基准代码（Code Base）里写程序。虽然有严格的代码审核过程和规范的开发流程，但是，不同功能的代码模块会产生交叉影响，不同工程师改动的模块会有重合或牵连，所以，系统还是会时不时出现问题。

那时候， Square 的做法是：在周五对本周所有的代码进行代码审查（Code Review），通过审查之后，把修改合并到主分支，然后再发布到生产环境。

这种做法虽然可以避免产生人为错误，但是非常不灵活，比如，每周只有周五有一次机会将改进的代码部署到线上。

可以想象一下，一百多名工程师，就算只有三分之一的人在这个代码池子里改代码，一周累积下来，已经有不少的改动了。

于是，当时 Square 有个系统管理员组（Sysops），专门负责每周五的部署。我也是工作近半年的时候，因为表现不错才被荣幸地“选拔”进了这个“特别行动小组”，承担部署的重任。

那么说，每次的部署是一幅什么样的场景呢？

部署开始的时候，一正一副两位工程师正襟危坐，多个显示器同时打开，进行各种指标监控。

工程师先将在测试环境中测试无误的代码部署到若干生产机器上，进行灰度发布，这就意味着有一部分用户的访问量会调用新代码。如果监控没有发现异常的话，再进行全量发布，这周修改的代码就会被部署到几百台机器上。

一旦出现异常，监控系统就开始各种红色告警，工程师们会立刻扔下手中的可乐或者咖啡，进入备战状态，停止系统，进行数据回滚、排查问题、修复，从头开始把流程再来一遍，直到代码安全地部署到线上并能够正常运行为止。

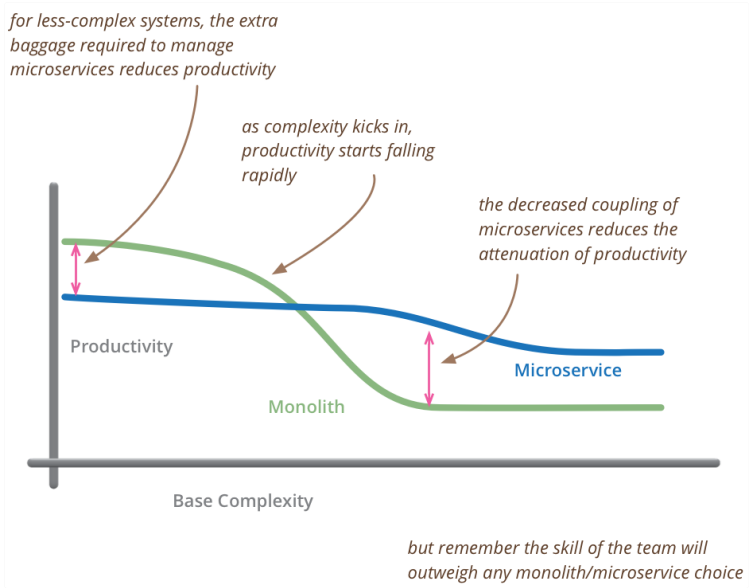
随后的两年，我们进行了细致的业务拆分，等到我离开 Square 的时候，大部分可以独立出来的服务都已经拆分出来，很多系统可以分别部署和上线，也就再没有了那种激动人心的周五上线日。

Airbnb 的情况也差不多，我刚加入的时候，代码状态甚至更原始一些。不同的是，Airbnb 没有一周只能部署一次代码的规矩，所有的工程师只要准备好了就可以做部署上线。

这样做的优点是可以快速迭代，每次部署的代码改动也很小，缺点是几乎任何时候都有人在部署代码。时时的部署也就意味着，红色告警随时可能在身边响起。

为什么系统需要进行业务拆分

为什么会出现这种情况呢？我在文稿中给大家放了一张图，图例很好地阐述了效率和复杂度的关系。



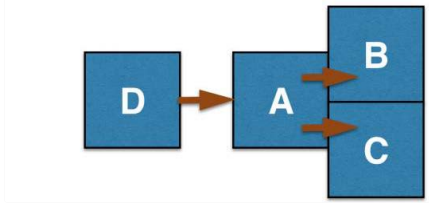
图的 X 轴代表了基本复杂度（Base Complexity），Y 轴代表了生产效率。我们可以看出，当一个公司规模很小的时候，基本复杂度相对较小，所以单一代码库（Monolith）的效率就会高。

然而，随着公司业务的扩展，访问量的增加，其基本复杂度就会逐步升高，达到某一个临界点后，微服务（Microservice）的效率就远远高于单一代码库。关于微服务，这里就不做详述了，极客时间会发布专门的微服务知识产品。

为了解决效率和复杂度的问题，无论是在 Square 还是 Airbnb，我都有一大部分时间花在了业务拆分上。下面，我就和你聊聊这几年做业务拆分的一些心得和踩过的坑。

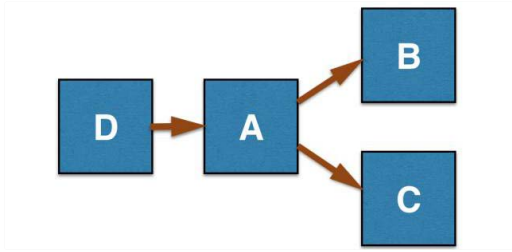
业务拆分并不像看起来那么简单

我们从一个例子谈起，比如你有一个功能模块，大概可以分成四部分。其中模块 A 连接一个外部模块 D，A 输出的结果，会被模块 B 和 模块 C 分别调用。



针对这样的模块，我们可以做一个集成测试（Integration Test），在模拟（mock）D 的情况下，测试 A、B、C 是不是可以正确运行。

如果有人修改了模块 A 的返回值，但忘了修改模块 B 和 C 的接口，测试就会立刻失败，不会存在因为忘了修改接口而测试通过的可能。一旦通过了集成测试，所有的改动会在一次部署中同时展现（Rollout） 或者回滚（Rollback），非常容易控制。



随着业务的发展，A、B、C 三个功能被拆分成三个独立的服务（Service），各自保存在不同的代码库，或者是同一个代码库不同的服务容器（Service Container）里。

这样的话，测试用例就不能综合测试这三部分的功能了，只能模拟相互的请求（Request）或响应（Response），如果在开发环境下联调测试，则需要本地建立这三个服务。

根据每个公司开发环境的成熟度，这一步可能很简单，也可能耗掉你几个小时，才能让不同服务在本地正常运行，并且需要通过 RPC 相互调用。

RPC 就是远程过程调用的意思。有远程调用，就会用到本地桩和过程调用，这涉及了本地多服务的配置，过程繁复，不小心就会引入错误，测试成本也会随之增加。

如果程序员在改动的时候并没有按照正常流程进行测试，尤其是一些 很小的 或者 不相干的 改动，一旦部署上线，系统就可能出现各种各样的问题。

就算一切顺利，有一天，A 修改了自己的接口，RPC 调用中请求的一个字段（Field）从 integer 变成 string 类型。

如果 A、B、C 还在一起的时候，我们在代码库里把三者的相应类型都改了就好；但是，现在 A、B、C 都是独立的服务，可以独立地部署，这事就有点麻烦了，我们很难保证 A、B、C 的部署总是完全同步。

有经验的读者知道，我们为接口做个向后兼容（Backward Compatibility）就好了，只要：

1. 先改 A 的接口，让它接受 integer 也接受 string，如果请求是 integer，先做一下转换，然后发布这个改动；
2. 修改 B 和 C 的接口，响应从 integer 变成 string 类型，发布这个改动；
3. 等到 A、B、C 的新代码都稳定了，再修改 A 的接口，只接受 string 类型的参数，发布这个变化，我们就完成了所有接口的改动。

这样就没问题了么？并没有这么简单。

因为 A 还有其他代码，所以在上面的第二步之后，你有可能发现 A 的代码有一个问题，需要将线上的代码回滚到之前的某个版本。这时候 B 和 C 的接口已经是 string 类型了，而 A 只接受 integer，然后，线上就是频繁请求报错。

当然，这里举的是一个简单的例子，我们可以通过延长第一步的兼容时间来避免出现类似问题，但是，实际工作中的改动不会是这么简单的依赖关系，或者没有约束关系，所以，服务之间无缝修改接口，是一个需要非常小心的问题。

业务拆分时的注意事项

系统拆分后的痛远远不止于此。就我自己的经历，大概有下面的这些感受。

测试会变得异常复杂

因为模块被独立出来之后，并没有办法很方便地写出集成测试用例。一个做法是模拟出所有接口的请求和响应，但实际上大部分时候根本没法测试跨服务的改动，这种做法多少有点自欺欺人的味道。

另一个方法就是在本地配置好所有的服务，用真实的服务响应来测试。但是撇开本地设置多服务的复杂度，保证本地服务一直是最新代码，同样也是一件麻烦的事。

尤其是同步开发的工程师变多以后，可能你正在测的服务没有问题，但是在你做测试的同时，已经有同事对你刚刚测试的服务做改动推送到了主分支上。

测试的复杂度，几乎是软件工程中的万恶之源。当每个小改动都让测试变得耗时耗力时，就难保没有偷工减料的员工，大家揣着“我的改动应该没问题”的侥幸心理，不去做完整测试，就把自己的代码合并进主分支。

尤其是大部分这么做的改动都没有问题，时间一长，侥幸心理一再滋长，人们直接合并代码的胆子也越来越大，终于有一天会把生产系统彻底搞挂。

针对这个问题，我和在 Google 工作的朋友交流过。Google 或者 Facebook 这样的大公司里，整个系统做得相当成熟，测试环境做得非常完美。

每个服务都对应设置了在线的测试服务，写集成测试极其方便，或者把服务做成开箱即用，工程师可以一次性地建立所有的本地服务进行联调和测试，但是，对于大部分创业公司来说，很难达到这个水准。

与接口相关的改动需要大量协调

这一点也很容易理解。比如我们要把一部分代码从一个服务迁移到另一个服务，或者修改 API 的协议，那么，所有不同服务的维护者都需要在代码里增加向前或者向后的兼容性，对代码进行保护。

同时代码的上线顺序和修改顺序也息息相关，我们需要做一张检查列表（Checklist），考虑各种可能性，精确地按照顺序执行。一旦发生代码回滚，可能又要重来一遍。

这个过程就会涉及方方面面的人、事和代码修改，过程十分繁复。

报错的处理

因为程序不在一起了，当异常发生的时候，我们就得不到完整的异常堆栈信息（Exception Stack），只能追踪到某个服务的接口处，于是 Debug 变得很难。你还需要去另一个服务的日志里去找，看看那个时间点从你这里发出去的请求到底发生了些什么事，然后才能进一步定位问题。

好的程序员在写服务的时候知道要把异常信息封装后层层传播出去，并最终暴露到接口的 4XX 响应里，这样，调用方就可以在堆栈信息里看到具体的出错信息。

如果有的程序员没有这么做，就很容易出现“无语问苍天”的感觉，比如你线上的服务出了问题，到日志系统 Kibana 里一看，只有下面这行错误信息，你是不是会很崩溃？

```
Error! HTTP 400 response from http://another-microservice.com/update
```

日志的完整性

系统拆分了，日志系统也会分离，不仅系统调试变得困难，一些基于日志产生的事件流（Event Stream）机制，也会变得难以处理。这意味着，想要真正从日志里获取完整有用的信息，就需要将不同服务的日志一起取出来进行分析和处理。

这种需求并不是所有的应用都需要，因为我们是做支付的，经常需要一个事务的完整审计线索（Audit trail），也就是一条告诉我们“每个相关的变化是谁做的，什么时候做的（who did what and when for every change involved）”的特殊日志。

这件事以前处理起来非常复杂，现在倒是有了比较标准的解决方案，就是一个共享的消息总线（Message Bus），比如 Kafka，有了日志就分门别类的扔到消息总线里处理，然后再进行分析。

超时设置

为了保证用户体验，我们常常在系统里做一些超时设置（Timeout），比如一个请求从终端设备发过来，我们希望用户最多等待 5 秒，超过 5 秒就会放弃请求并返回相应的结果通知用户。系统拆分之后，我们可以做一个全局的超时设置，让所有的服务都使用这个全局变量。

这一切看起来很美好，但稍不注意就会出现这样那样的问题。由于服务都是独立开发的，如果某一个服务的实现没有使用 5 秒的全局变量，我们就不知道这个服务到底超时多久才会返回结果，或者是否有超时的设置。

另外，根据某些服务的性质不同，我们希望尽可能地给出最合理的延时设置。还有些请求会经历多次跨服务的调用，一旦同时出现超时，就会进行叠加，超时设置就完全不可控了。

为了避免这些情况出现，就需要增加流程和规范，并且在进行系统拆分的时候进行宏观的设计和考虑。系统拆分会为我们带来灵活性，同样也会增加其他成本。

关于代码自由

记得以前看到过一句话，当每个人都有绝对自由的时候，这个世界就没有自由可言了。拆分之后每个服务的实现都可以自主选择自己的语言，自己的数据存储方式，自己的代码风格。

短期来说，这种做法可以让程序员的效率极大地提高，但是在同一个公司里，当各种各样的服务变成一场技术秀的时候，不论是维护还是稳定性都会受到极大的挑战。于是，这时便会有人扮演清道夫的角色，开始搞服务的标准化。

另外，独立服务的开发周期相对较短，往往一两个工程师几周时间就可以写出一个新的服务，这样系统里会出现数不清的服务，有的服务由于人员离职等原因没人维护了，有的服务被重写了，有的服务要退休了，为了管理这些服务，我们还需要一个服务编排和管理系统。

系统拆分之路漫漫，吾等将上下而求索。

如何去判断系统是不是到了必须进行拆分和服务化的临界点

写到这里可能有读者会问，这篇文章中你介绍了大代码库的弊端，也写了很多系统拆分和服务化需要注意的问题，那么，你到底想告诉我什么呢？

做为一个亲身经历过两种架构的工程师，我想说的是：系统拆分并不是做一道单选题，在进行系统拆分和服务化之前，我们需要综合考虑各种因素，找到平衡点。

1. 你的业务量是否足够大，逻辑是否足够复杂以至于必须进行系统拆分。水平扩展是不是已经不起作用了？代码的相互影响、部署时间过长真的是系统的切肤之痛么？如果答案都是肯定的，那么你就应该进行系统拆分了。
2. 对于服务化的架构，你的开发人员多少经验，能否正确驾驭而不是让本文中提到的问题成为拦路虎么？
3. 系统拆分是一个“从一到多容易，从多到一困难”的过程，这个过程几乎是不可逆的。一旦你三分天下，想再一统江山就没那么容易了。所以在做拆分计划的时候，一定要慎之又慎。

系统拆分是一个实践性很强的工作，并无一定之规，只有亲自参与了这个过程，才会有更深入的体会；在这个过程中，你的架构能力也会产生一个质的跃迁。

文章的最后，我来总结一下今天分享的内容。

今天的文章较长，涉及的内容也比较多：

第一点，我谈到了创业公司初期，代码的构建状况以及遇到的相关问题；

第二点，我解释了为什么随着业务的发展，我们会进行系统拆分；

第三点，我提醒了业务拆分并不像看起来那么简单，我们需要时刻去注意细节；

第四点，我分析了在进行系统拆分和服务化的过程中，需要注意哪些问题；

第五点，我讲解了如何去判断一个系统是不是到了必须进行拆分和服务化的临界点。

希望这些内容对走在创业路上的技术人有所帮助。

你有系统拆分的经验和故事么，可以在留言中告诉我，我们一起讨论，也欢迎你把这篇文章转发给你工作中的伙伴，我们一起成长。再见。

刘剑

2017-12-15

技术管理课，讲这个稍微有点过于基础哦（包括前面的数据库片）。建议多讲讲团队、激励、培训、招聘、绩效等方面呀。

刘剑

2017-12-19

对于业务拆分的原则之一是：服务边界内的业务能力职责单一化，不是完成同一业务能力的模型不放在同一个上下文中。

至于拆分的手段，我们用的是

1.绞杀模式，就是在遗留系统外围，将新功能用新的方式构建为新的服务。随着时间的推移，新的服务逐渐“绞杀”完老的系统。对于那些老旧庞大难以更改的遗留系统，推荐采用绞杀者模式。

2.修缮者模式就像修房子和修路一样，将老旧待修缮的部分进行隔离，用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍能协同功能。App版本兼容上也多用此模式

我们业务拆分的原则是：“旧的不变，新的创建，一步切换，旧的再见”

whhbbq

2018-01-09

开发环境的成熟度、调试难度、日志查看、接口超时、异常处理，安姐列出的都是干货，都是系统拆分随之而来绕不过去的痛点。去年经历了公司系统的微服务化和一些模块的重构，看完文章后特别有感触。特别是修改代码后，有时本地需要起好多服务才能调试，一直是个痛点。公司只有一套公共使用的环境，上面部署的都是最新的主干代码。安姐提到google和facebook的开发环境比较成熟，他们是如何做到开箱即用呢？能否针对这些痛点，写写解决的方案？谢谢！

顾金鑫

2017-12-15

灰度发布的时候，新旧的数据库 schema 不一样，怎么数据迁移咧？新老系统同时存在，也就意味着会对库有两种写法...这可如何是好

zhengfc

2017-12-15

拆起来容易，合起来成为一体就有挑战了

simaopig

2017-12-15

不要为了拆分而拆分，视业务实际痛点，人员实际水平综合考虑

泽

2018-03-24

系统拆分是任何到发展到中后期公司必须经历的过程，前期因为商业模式试错抢占市场等会快速上线快速迭代，前期公司也没多少牛人，各方面都是业务优先，在不知道自己能活多久前提下，谈什么服务化技术优化都是扯淡。随时公司发展一般2-3年还没死，随着业务量越来越大，系统增加新功能、系统维护成本越来越高，系统变得越来越不稳定，DB一直挑战极限，这个阶段重构服务化是必须介入了，拆分和服务化的具体问题文中大体都介绍了，至于什么原则来进行服务化，如何去确定服务边界，如果确定上下文大小，哪些功能该放在一个服务中，这些需要好好看看DDD-领域模型设计。什么阶段做什么样的事，遇到事了不怕事，事后会发现这些不过如此。

张伟波

2017-12-15

系统拆分最佳时机是否是软件设计时，如果系统已经上线，因为没有拆分带来了大量的弊端，比如迭代发版总出各种各样的连带问题，是不是要考虑下架构的问题了？

bluze

2018-01-04

这篇文章写的非常好，从望京一路看到西二旗。解答了很多困惑。

Dylan

2017-12-30

公司刚成立，我们现在的代码库就是一个大的代码库，工程师人少，每个客户端，后台，前段也就一两个工程师，所以目前每次部署变更正式环境还是可控的，而且业务逻辑也还没复杂到说要进行拆分的时候～看了作者的观点后，还是要准备着以后可能面临的这些拆分问题～

walt

2017-12-27

Airbnb是开发人员写测试用例，执行测试吗？上线前不回归测试吗？

mattlin

2017-12-19

喜欢这种平易近人的文章 ��

00

2017-12-18

最近浏览了后台架构，了解了下thrift、kafuka。看了安姐的分享，顿悟到thrift的无缝衔接。自己做的是桌面，了解系统的架构总觉是纸上谈兵。安姐的文章通俗易懂，因为有场景，即便是没从事过相关工作，也会理解相对深刻一些。
3ks

碰上这样的朋友

2017-12-18

团队构架对拆分也会很大影响

天之炽

2017-12-16

能不能着重讲讲招人

Edward

2017-12-15

朱老师，有开源的灰度发布工具和基于微服务的自动化测试工具推荐不