促进软件开发领域知识与创新的传播

架构师

ARCHITECT

热点 | Hot

谈谈后端业务系统的微服务化改造

观点 | Opinion

为什么我不再使用MVC框架

推荐文章 | Article

新浪微博混合云架构实践挑战

分布式MySQL集群方案的探索与思考

特别专栏 | Column

AWS S3：不仅仅是存储

Geekbang 极客邦科技 | InfoQ

2016年6月

ArchSummit
全球架构师峰会2016

2016

2016年7月15日-16日

深圳站

扫描二维码了解大会

卷首语：写在 InfoQ 十周年

作者 **魏星**

2016 年 5 月 23 日是 InfoQ 十周岁的生日，同时距 AWS S3 发布也刚好十年。十年里，随着云计算大数据的蓬勃发展，企业基础设施和系统架构发生了翻天覆地的变化。《架构师》作为 InfoQ 中国的招牌作品在过去的几年里已经赢得了大家的广泛赞誉。

架构师是一个充满挑战的职业，知识面的积累往往决定着一个架构师的架构能力。一个优秀的软件架构师，首先一定是一个出色的程序员。但，不是每一个程序员都能够成为一个架构师——这是开发界广为流传的一个论调。随着一切都趋向全堆栈、多种语言化、网络即平台等方向发展，架构师这个角色在企业中的地位越来越重要。

有人的地方就有江湖，有江湖的地方就会有纷争。如同语言之争，每当我看到有人为某些架构、框架的优劣争论不休时，我都会想起遥远的 C/S 与 B/S 之争，甚至 Vim 和 Emacs 之争。我们也看到有很多项目、很牛的架构师、很好的团队最终失败的案例。《软件设计精要与模式》作者张逸在接受 InfoQ 采访时曾说，评价一个架构的优劣的方法之一是，把每一个功能、非功能性的因素扩大化，再看这个架构会不会出问题。诚哉斯言。

张逸的观点是说，架构的设计要面向业务未来，而未来是不断发展变化的。根据 Gartner 的报告数据，云化是未来所有业务的趋势，云服务的基础设施和系统架构跟传统 IT、CT 有着太多不同。今天还是微服务、分布式架构，明天可能是无服务、或者其他什么架构。这时，落在架构师肩上的担子更重了，不但要承载业务的增长，还要兼顾技术发展的趋势。在接下来的日子里，希望《架构师》能继续陪你走下去。

CONTENTS / 目录

架构师

2016 年 6 月刊

本期主编　魏　星

流程编辑　丁晓昀

发行人　霍泰稳

提供反馈　feedback@cn.infoq.com

商务合作　sales@cn.infoq.com

内容合作　editors@cn.infoq.com

案 / 例 / 剖 / 析 · 实 / 践 / 驱 / 动

CNUT
Connect Container Community

CNUTCon 2016
全球容器技术大会

2016年9月9日－10日　　中国 · 北京　喜来登长城饭店

6折优惠进行中，5人团购更划算

主办方 Geekbang. 极客邦科技　InfoQ

谈谈后端业务系统的微服务化改造

作者 张旭

1、篇首语

业务系统是任何一个用户产品的必须组成，充当着一个门面的角色，用户的输入就是这个系统需要维护的，数据存取是整个系统的核心。例如，广告业务系统的输入是广告主的投放约束、定向条件，微博业务系统的输入是短文字、图片等。

在应用发展初期或者规模不大的情况下，有非常简单的实现方案，LNMP、JSP、PyWeb 都是你能随口说出来的词，如果用某种架构方式来描述，那就可以称做单体模式（Monolithic，Martin Flower 大神所提出的，后面还会介绍），而这些技术都是单体模式的成熟解决方案，它们可以使你工作在“应用层”的最顶端，各种中间件、框架能让你省心地干好业务，开发人员可以通过“模块化”的手段来管理业务系统的复杂度，使他们之间解耦、复用。简单来说，这个单体就是如下这种层次划分。

```
表示层        前端（HTML+CSS+JS）
  |             |
逻辑层      业务系统（PHP、Java、Python 是常用的语言）
  |             |
数据层        数据库（MySQL）
```

看起来很简单，对吧。诚然，业务系统在这里面还需要做很多，比如

缓存、SQL 优化、数据分区、系统安全工作，当然还有对于代码的维护和重构。这种工作方式可以很好的工作几年、甚至十年，但是如果业务发展非常快，在系统复杂度、业务规模、参与人数、代码腐化程度都不断上升的情况下，你会发现整个项目正陷于泥潭，PM/RD/QA/OP 经常抱怨：

"改个小功能，怎么要拉这么多模块？"

"拉模块也就罢了，改的地方多，编译太慢了。"

"慢也就算了，关键不知道怎么改，这代码太丑陋了，算了，为了满足 PM 的排期需要，凑合来吧。"

"凑合来了，QA 发现 bug，返工再返工，延期再延期。"

"上线了，oh my god，报 case 了，性能有问题，原来是依赖的模块访问数据库用了 for 循环 select。"

透过现象看本质，我总结了来看就这三点问题：

1、业务逻辑复杂耦合，开发维护成本高。

系统复杂度、规模、参与人数都和腐化程度成正比，单纯的靠模块化，后期来看会存在个别模块成为”大怪物“，臃肿不堪，粒度过粗，难以复用。

2、交付效率和质量低。

在敏捷和持续集成方法论、实践大行其道的现今，迭代的按期交付率、单测覆盖率都不尽如人意，线上问题频发。

3、非功能需求不达标。

非功能需求指标特指性能、可用性、可扩展性等方面，代码的腐化和缺少维护、重构，以及没有代码洁癖的人污染下，必然导致性能逐渐下降，甚至出现不同资源竞争的短板效应，造成整个系统 crash，同时一个大怪物的伸缩性较差，不能随意横向扩展某个细分功能点。

我想任何人做架构都需要秉承“业务需求决定技术演化路线”的思路，

那么这些暴露出来的现状和问题都驱动系统去转型，在系统和人之间找到一种最佳的合作模式，匹配已有的生产力和生成关系。正如软件开发学泰斗 Kent Beck 所说的：

“Design is there to enable you to keep changing the software easily in the long term”，即“变化发生时设计被破坏的程度”

放眼业界，面对复杂的、大规模的、多人协作完成的业务系统，如何管理好这个复杂度，有很多方法，模块化、OSGI、传统服务化 SOA 等等，当今最佳实践的趋势还是服务化，而微服务是最近火热起来的概念，有些人肯定觉得这不就是炒作嘛，但是不管黑猫白猫能抓耗子就是好猫，所以解决问题是重点，不要刻意去批评一些名字，那么本文要重点介绍的就是——如何做微服务化转型和改造。

在接下来讲之前，要重点声明，本文并不是推崇服务化，不鼓励单体模式，相反而是相当肯定和支持单体模式，它模块依赖简单、一个发布包、部署于一个容器都使得构建应用非常的简单，在体量还不大的情况下，首先应该解决的是搭建好一套绝对稳定的基于模块化的平台，待体量逐渐长大，再去根据实际需要进行拆分，团队也随之变化（facebook 的单体持续了非常长的时间，一是人员素质高，二就是基础平台建设的非常好）。再下个阶段体量大到饱受单体模式之苦，也应该先建设平台化服务，建设好之后，先按照大的粒度进行拆分，逐步再微服务化，否则，直接上服务化、甚至下面要说的微服务都是非常危险的，鲜有成功案例。

2. 微服务化改造

做改造一般要经历三个步骤：

第一、技术选型决策

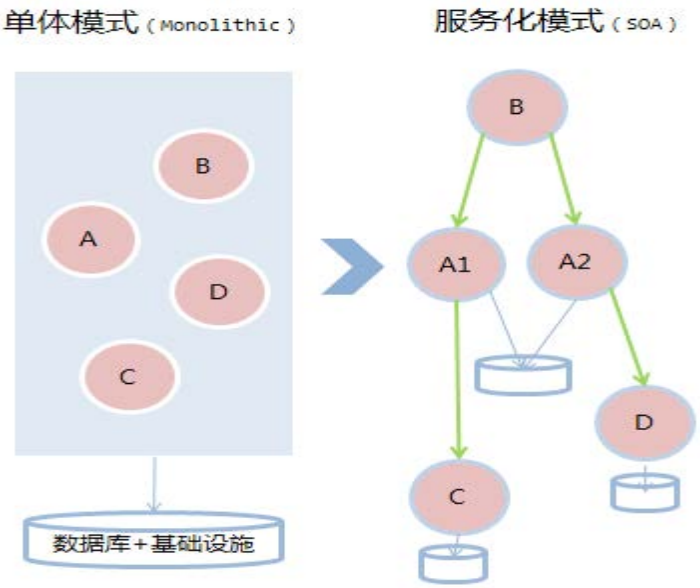

图 1

第二、架构设计规划

第三、落地实施应用。

下面依次展开三个部分，重点介绍前两个步骤，有了这两个，落地应用也就顺水推舟的好做了。

2.1 技术选型决策

2.1.1 选择微服务化方式

选择服务化，众所周知就是 SOA 嘛，这是一种架构风格，重点在原则、理念、方法论等高思维层次上，对于工具、框架、解决方案没有做强制限制，ESB、websercie（基于 WSDL 和 SOAP/BPEL）这两种是企业中流行的，也是过去一直引领 SOA 的技术领头羊，但是随着互联网应用的发展，在敏捷快速迭代、高可用、高性能、高并发等方面要求越来越高，传统的 SOA 并不适合这种场景。那么，现在的互联网流行的实现方式是什么呢？一种最佳的实践方式就是微服务化，见图 1（Micro Service）。

微服务就是一种 SOA 的实现方式而已，更加侧重于在服务的细分演

化，是指引服务的具体落地方案层面的一种实践方式。过去很多互联网公司在实践，你可以把淘宝的 dubbo、HSF，navi-rpc 服务化框架看做比传统 SOA 更适用、更贴近微服务化实现的服务化框架，依赖这些框架可以方便的做服务化。这个趋势被 Martin Flower 大神所发现，并且提出了，你们这些不都是在做“微服务”嘛。

Martin 对微服务这个术语（terminology）的解释是：

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

简而言之，微服务化就是以一系列小的服务来开发支撑一个应用的方法论，服务独立在自己的进程中，通过轻量级通信机制交互（通常是走 HTTP 协议）。这些服务是围绕着业务上的组织结构来构建的，全自动的、独立部署。几乎看不到中心化的服务管理基础设施，可以使用不同的编程语言和数据存储技术来实现不同的服务。

在简单的一种理解来自于一本书《Building Microservices》（Sam Newman, O’Reilly Media），Microservices are small, autonomous services that work together. 微服务化就是一堆小而自治的服务，让他们一起工作起来。

相比于传统的 SOA，Martin 的总结特点可以参考他的博客还有视频（Youtube 链接），一共是 9 个特点，这里不想赘述，而是说说我个人的理解，微服务化的特点下面详述。

1、模块即服务

微服务中的组件在逻辑或者物理层次更趋于细分，这个细分不是极致的，而是一种粒度适中的选择，通常这些组件在前期可以是一些模块，但是当需要时，例如业务上需要拆分独立，或者非功能需求上需要扩容等，都可以灵活的拆解出来。这个特点非常重要，因为业务系统中模块化实践，随着软件规模的变大，很容易绕过障碍而使得不同模块耦合、依赖关系复杂，这种纪律性很难保证，从而削弱模块化的结构、降低了团队的生产力（敏捷开发和持续交付越来越难做，部署起来太庞大了大家的开发士气不高，而且痛苦），很快的这个模块就会变成一个大杂烩，而服务可以做到天然的壁垒，仅仅通过交换契约（通常是 API 或者 proto）来做交互，这是一个演化的过程，不仅有利于分而治之，到达复用的目的，同时老系统也可以灰度的改造剥离。

2、独立自治

这意味着服务是独立开发，独立测试，独立发布，独立部署，独立运维的，某个细分团队负责整个生命周期管理，这就是“康威定律”的通俗解释，官方解释是“一个组织的设计成果，其结构往往对应于这个组织中的沟通结构”，服务的规划不就是多人、跨团队协作的沟通模式嘛。好处在于摒弃原来的火车模型（所有模块一起发布部署），拥抱独立快跑，这也更好的支持了敏捷和持续集成的方法实践。同时去除了牵一发而动全身的问题，单一职责的来进行修改需求或者重构一个点，开发和构建方便，不影响整个产品的功能，一个 bug 不会 crash 掉整个产品，针对不同的类型，区分计算密集型还是 I/O 密集型，区分业务上更好使用关系型还是 NoSQL，区分 2/8 原则、即 80% 经常修改的服务独立出来自成一家，区分短板功能、针对瓶颈可以做水平扩展、避免资源竞争，甚至可以区分技术

栈、突破语言限制。最后，这也是和第一点遥相呼应的，独立的服务可以实现非常大程度上的复用，服务之间依赖轻量级的接口，而不是模块。

3、去中心化的数据管理

在单体模式中，一个应用面对一套数据库，数据库可以按照物理拆分，进行 shard 分区，或者按照逻辑库隔离，不同的业务路由到不同的库，同时做一主多从等结构上的设计，这些原则在微服务中仍然适用，只不过微服务在服务拆分的同时，也需要将数据库分离，独立的服务维护独立的数据库，这对数据库也是减负，同时技术选型、SLA 保证都会区分开来，把精力留给那些重要的业务数据库，进行分级的对待，而不能像以前一样一视同仁，一个不重要的逻辑库的 bad SQL 慢查询，阻塞了其他正常的查询，这是完全可以避免的。

4、轻量级的通信协议

传统的 SOA 使用 ESB 或者 Webservice 这种重量级的解决方案，微服务推荐使用一些更轻的解决方案，要通用性，可以用 Restful 架构，走 HTTP 通道，支持 Json 序列化协议；要高性能，可以考虑一些高性能的 I/O 模型，例如 epoll、Actor 等，可以直接走 tcp 通道，使用 protobuf 序列化协议，同时保持长连接（这里有一个例子 Navi-pbrpc 就是一个这样的具体落地框架）；要异步，用可靠持久的 RabbitMQ 或者高性能的 ZeroMQ 来做 P2P、Pub/Sub、广播 broadcast 消息通信。而这种轻量级还需要体现在代码调用中，模块化直接通过函数、方法调用即可，服务化后能不能在 API 层面做到无侵入，无缝的切换、简单配置，这些都是服务化框架要支持的。

5、为失败设计

服务化调用从进程内 in-process 的调用，转变为跨进程的分布式调用，这种由分布式特性引起的天然不可靠性，需要变为相对可控。也就

是服务间的通信要假设不会成功，为失败处理。异常的传递，能否透过RPC，在调用方本地还原，就像函数、方法调用一样？一个点、或者服务的处理错误率到了一个阈值，为了不影响整个产品，要做错误隔离，可以考虑熔断（circut break）、舱壁隔离模式、限流、回退等手段，最后还有一个幂等性问题，重试的调用会不会对业务造成影响，这个要具体问题具体分析了。

6、基础交付设施自动化

这个特点是整个微服务中的最大亮点，包括持续集成 CI、持续交付 CD 和 PaaS 平台的结合。微服务在细分的背景下，在 project 结构，物理结构上都提高了一个复杂度，如果还要做到提高软件交付链路的整体效率，就需要在基础交付上做一个大的转变，因此 DevOps 文化，让每个人都参与交付，在规范的流程和标准的交付（例如，标准的容器）下，同时在 PaaS 服务提供商的帮助下，完成一个服务的自动部署发布。

任何事情都是两面的，有好的优点，当然会存在弊端，微服务的缺点我的理解如下：

1. 分布式调用造成的性能、延迟问题。（可以采取的措施包括粒度适中、批量、高性能RPC、异步通信等）
2. 可靠性不好保证。（刚才提到的为失败设计可以解决）
3. 数据一致性难以保证。（看各自的业务，确保最终一致性即可，实际上大多数互联网产品很少不用事务；但是我目前所工作的商业产品领域，是需要事务的，除了不推荐的两段式提交，还可以引入仲裁者、补偿措施来解决分布式事务问题，问题可以单独开一篇文章介绍了，这里就不展开了。）
4. 整体复杂度提升。服务多、依赖多、调用多、契约如何管理、监

控如何做，调用链上怎么确定哪个点有问题，服务的SLA保障、性能、错误率、告警、这么多服务如何集成测试、交付容器如何上线等等问题。（通过服务治理可以有效降低微服务化复杂造成的低效，转为推动工程生产力的高效进化，同时基础交付设施的自动化可以加速研发效率，选择了微服务就等于选择了成本优先战略，投入的成本都是为了未来业务的更好发展，避免J-curve曲线式的研发模式，只有在体量大，基础设施包括服务化框架、治理能力完善的基础上，加上流程、规范以及工具和技能的辅助下，才可以真正发挥服务化的威力，否则只有自讨苦吃）

2.1.2 微服务化框架和治理模型

架构方式、原则达成了共识，你再往下看。虚的说完了来点干货，来介绍下我所在 team 实践的微服务化，最核心的就是微服务化框架，业界流行的阿里的框架 dubbo 以及淘宝内部的 HSF，Navi-rpc 都可以看做微服务化框架的雏形，加上服务治理中心的管理、基础交付设施的保障就可以构成完整的一套微服务框架。我们的框架（暂时仅内部使用）整体架构如下图，他由服务发布者、调用者和治理中心三者组成，属于标准的协调者模式（见图 2）。

生产者中服务逻辑在 Spring 或者 Guice 等 IoC 框架的 bean 中，由 IoC 容器托管，为了符合模块即服务的思想，在框架层级实现了一套可插拔组件的引擎，去实现组件的扫描，需要暴露服务的发布出来，依赖别的服务的，通过字节码技术生成 Rpc 调用代理 Stub，形成了一个基于组件的容器，通过 JSR315 这个规范的 SPI 实现对接到 J2EE 容器，下面的消费者结构相同。

在服务启动后，首先会第一步注册自己到服务治理中心，上传自己的

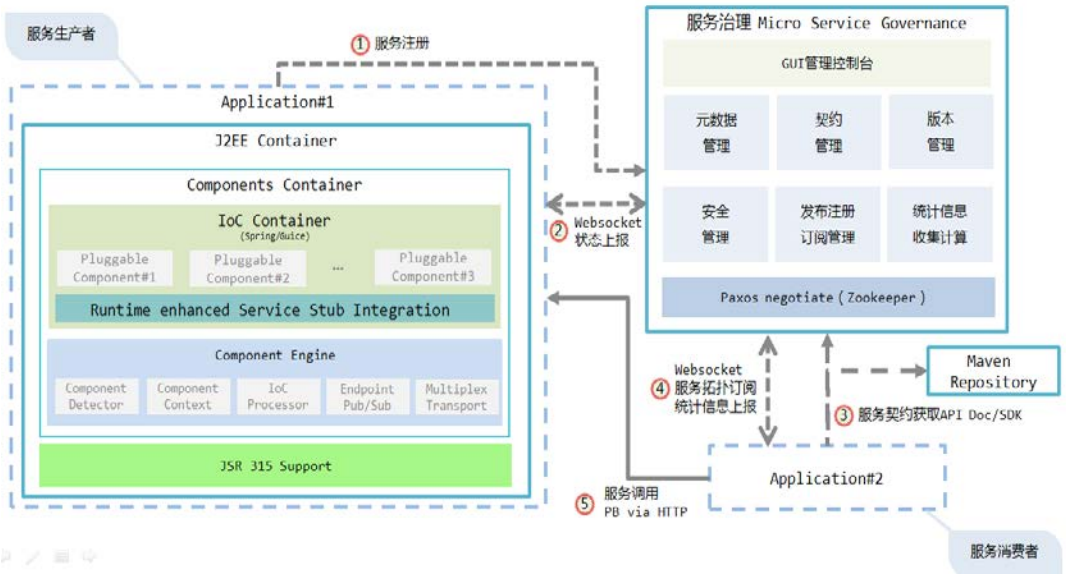

图 2

契约、版本上去，治理中心如果通过检查就发布出去，之后和治理中心通过长连接协议（我们采用 websocket，因为现成、简单）做一个订阅发布的通道，可以供收集状态，推送服务 Endpint 的变更；服务消费者可以去治理中心或者 Maven 仓库获取契约、SDK，治理中心推送 Endpoint 下来，供路由进行 Rpc 调用，通过消费者也通过长连接协议来进行状态和统计信息的上报，供治理中心进行分析决策和反馈。

服务治理中心为了保证高可用性，通常使用 Zookeeper 这个流行的开源的基于 Paxos 的方案，当然最近渐渐流行起来的 kebernetes 的 etcd 是基于 Raft 的集群共享数据、也可以做服务的发现的解决方案。

随着这种分布式调用越来越频繁，就需要服务治理能力越来越强，否则就是一张混乱的、无序的 Rpc 调用的网，无法管理复杂度。

这里建立了服务治理的模型，在下图中的服务治理中心来实现，模型从这样几个角度来考虑如何治理服务，包括通信、契约、版本、监控、安全、交付等角度，依托服务治理中心，有了这套基础设施保驾护航，服务

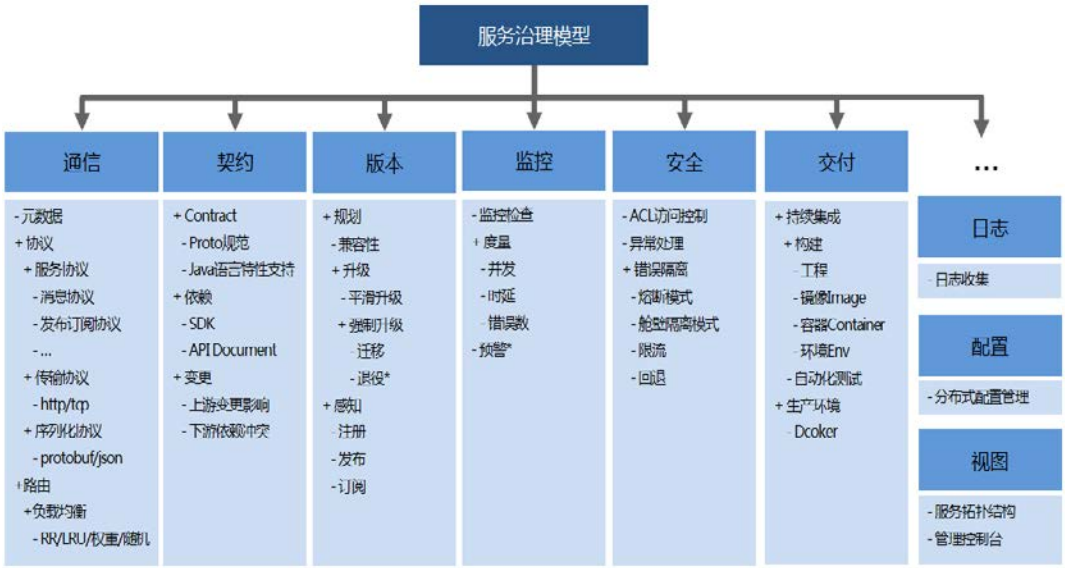

图 3

化就可以真正做到提高研发效率、提供优雅的开发体验（见图 3）。

在基础交付设施自动化上，如下图所示，体现在自动化、容器化交付这个流程中，在平台化的背景下把团队思维转换为 DevOps 式的，依托 Docker 和 k8s 完成了 PaaS 平台的对接，同时和 QA 一起协作完成持续交付流程的建立（见图 4）。

2.2 架构设计规划

这里所指的架构，特指组织、服务的架构设计，非部署和代码架构。

下面我要介绍的，都是扣题，是已有系统的服务化改造，是一个已经存在的、复杂的、体量大的业务系统。

做架构设计规划，主要分为步骤：

1. 整体架构设计
2. 业务领域抽象、建模
3. 服务规划与层次划分
4. 服务内流程、数据、契约（接口）定义和技术选型。

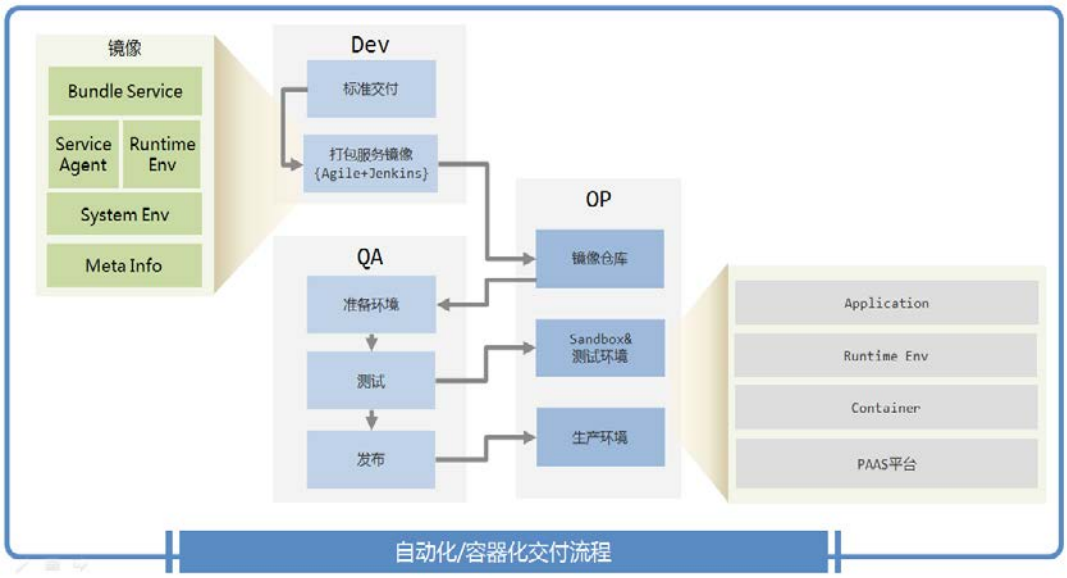

图 4

这里主要介绍前三个步骤，第四个偏向于个例，同时需要强结合业务需求、特点分析解决，这里不做详细展开。

2.2.1 整体架构设计

还记得文章开说所说的单体模型吗？在一个复杂的、规模大的业务系统中，使用微服务化方式实现，就需要从上到下的来做整体架构，下面这张图是我所在的商业产品的业务端到检索端的架构图（见图 5）。

共分为 5 个层次。

第一层，模块化组装，是各个投放产品的门面，各个投放产品可以通过搭积木式的方式，组装下层服务，就可以完成一个面向用户的功能，最常见的 SpringMVC 技术、Java 设计模式中的 facade 模式就属于应用到这一层的一些点。

第二层，计算服务层，服务化也就是在这个层次上展开的，每一个小圆圈都是一个微服务，这是整个服务化的核心，各个服务圈出来的都是一个个服务簇，比如投放管理一个簇，报告报表一个簇。

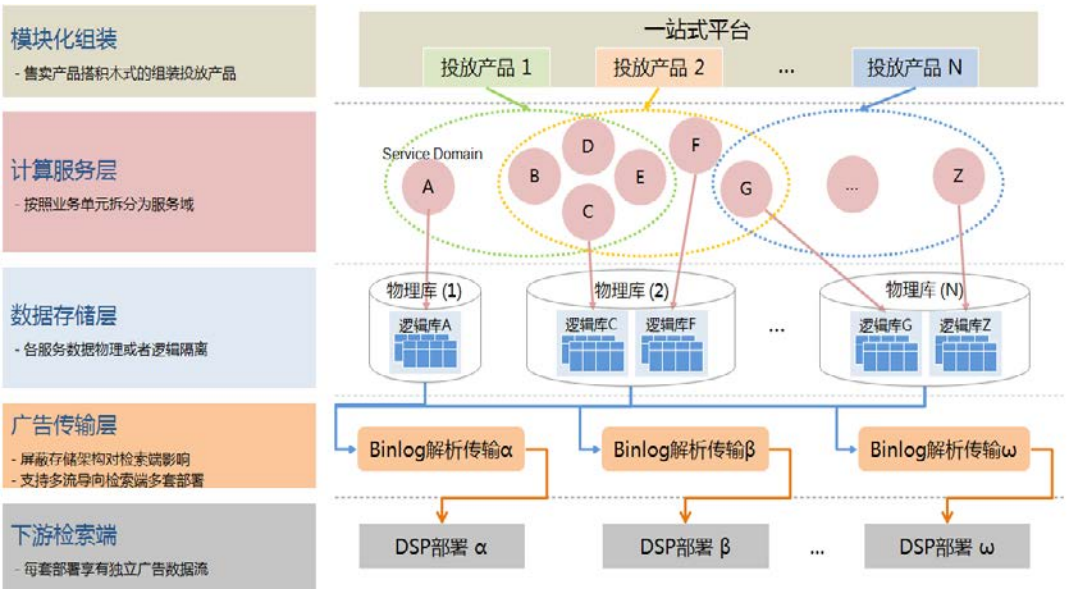

图 5

第三层，数据存储层，会针对各个业务拆分，按照物理库或者逻辑库进行隔离。

第四层，广告传输层，将多 shard 的 MySQL 写入的广告增量实时传输到检索端，形成一条增量流（incremental data stream），我们通过模拟为 MySQL 的一个从库来捕获解析 binlog 实现，将 binlog 增量映射为语言级别的抽象类型，供下游使用，下面一层就是一个数据接收方，其他的还包括一些 MQ 订阅方（如导入 kafka、RMQ、ZeorMQ 等），HDFS 存储等，这样就形成了业务系统的数据快速、高效、实时传输的目的。

第五层，检索端。是广告投放系统的核心，根据媒体环境、用户特征匹配最佳的广告，进行创意的投放，你所看到了图片、H5、flash 广告都是这套系统响应的，可以做到千人前面，最佳化广告主 ROI 与用户体验的折衷。

2.2.2 业务领域抽象建模

技术是为业务服务的，没有了业务，纯粹的讲技术都是纸上谈兵，解决问题是所有技术的出发点，微服务化也不例外。服务于业务，就需要对

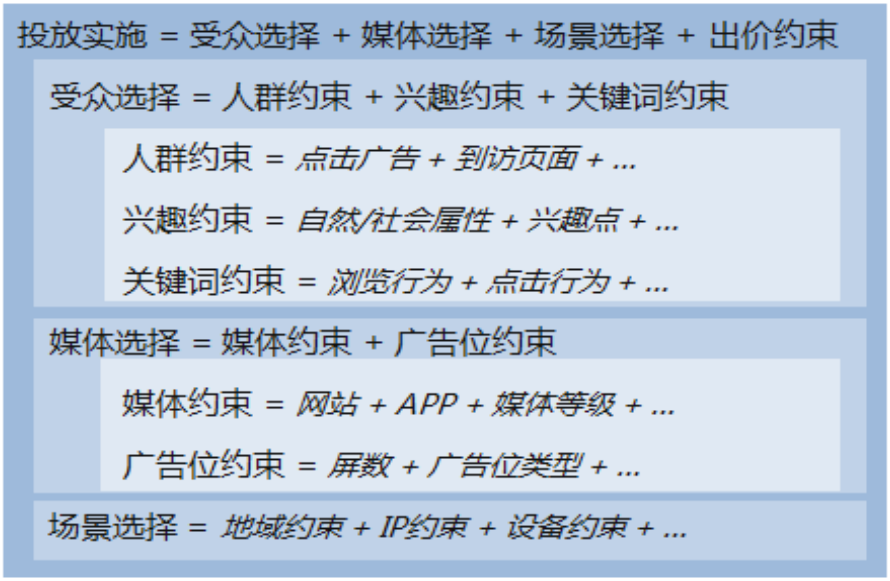

图 6

业务有深刻的理解，技术才能形成良好的输出。

有了前一步的整体架构规划，下一步就是计算服务层中的微服务如何规划的问题，这部分最为复杂，需要深入到产品业务中。拍脑袋规划当然可以，这叫做经验直觉主义，我认为经验主义缺少规范化的表达和标准化的设计，面对未来的修改需求，其架构的生命力不会很强。所应该站在更高的视角上尝试解决，首先就是要规范化需求表达，下图就是一个投放实施的表达，使用巴克斯范式（BNF 范式）表达，将投放实施分为受众、媒体、场景等定向的选择，每种定向又分为多个约束条件，逐层深入，这个规范是所有已有产品的萃取，在新产品的打造中需要遵守的，一般会和产品经理一起打造。（见图 6）

然后各个投放产品进行的功能矩阵划分的标准化设计，以这些为基础，就可以有理有据的进行服务规划，抽象分解出来的服务域高内聚，职责非常清晰，服务内的实体也是建模的，如图 7 所示，每个包都是一个微服务。

2.2.3 服务规划与层次划分

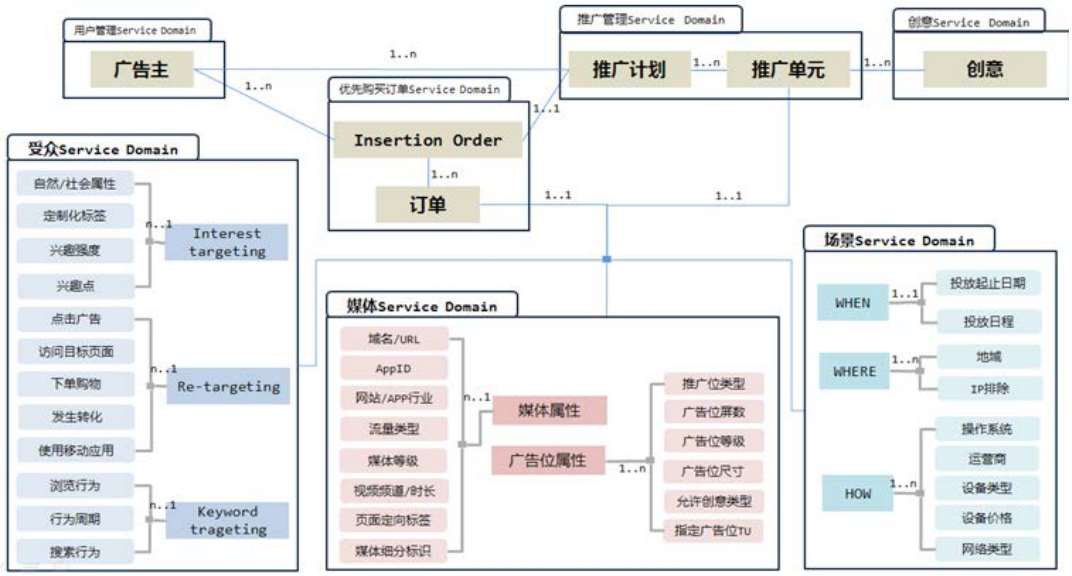

图 7

基于对业务的抽象分解，在计算服务层内部，就可以进行更加细分的层次规划，先是垂直拆分为展现层、计算层、数据资源 3 大纵层，核心的计算层又细分为 3 个层次，包括业务流程处理层，通过组装下层服务完成功能；业务逻辑组件是自包含，跨产品线、高度复用的组件；下面公共服务组件是一些通用服务。然后水平划分为多个服务簇。如图 8 所示。

按照之前的服务规划，将各个微服务安置其中，最上层的 Web-UI 和 API 服务负责和前端以及客户端（安卓或者 iOS）API 打交道，中间例如推广管理作为一个业务流程处理组件的工作流，可以调用下面的微服务进行组织，完成一个投放流程的业务场景。所有这些服务都是通过分布式服务化框架来进行通信、治理的。

2.3 落地实施应用

下面是一个已有产品改造的案例，比如一个报表服务簇，过去是一个大单体，现在按照服务化的架构，进行拆分，最为核心的就是中间这个 sync-report 服务，它从 olap engine 中查询数据，然后通过 merge 字面

图 8

数据，提供排序，过滤，分页功能。围绕 sync-report 抽取了多个不同维度的缓存，保证了核心报表服务的高性能，同时上层，不管是 Web-UI 还是 api，都复用 sync-report，这样上层就会很薄，不用再管那些复杂的查询逻辑，sync-report 作为标准、规范的技术解决方案，做到了统一复用与专职专用，加速了研发效率和交付。（图 9）

3. 篇后语

本文所提倡的微服务，是结合作者所在团队自身业务特点来说的，适合自身的场景，是建立在团队人员素质到了，有成熟的基础设施和框架、中间件辅助，流程也规范，包括 CI、敏捷等，团队都做好了准确去做这个转变，有足够的能力来实施，微服务化也就是水到渠成的事了。相反，小团队在前期或者野蛮生长时期，不宜选择微服务，不但影响效率还带来额外的复杂度。成长型或者大公司，有成熟的流程、规范、基础设施、平台等，要想在整条交付链路上加速，就需要投入更多的资源保障微服务化，一切自动化了，能治理了，回头看来这一切就都是值得的，远期收益非常

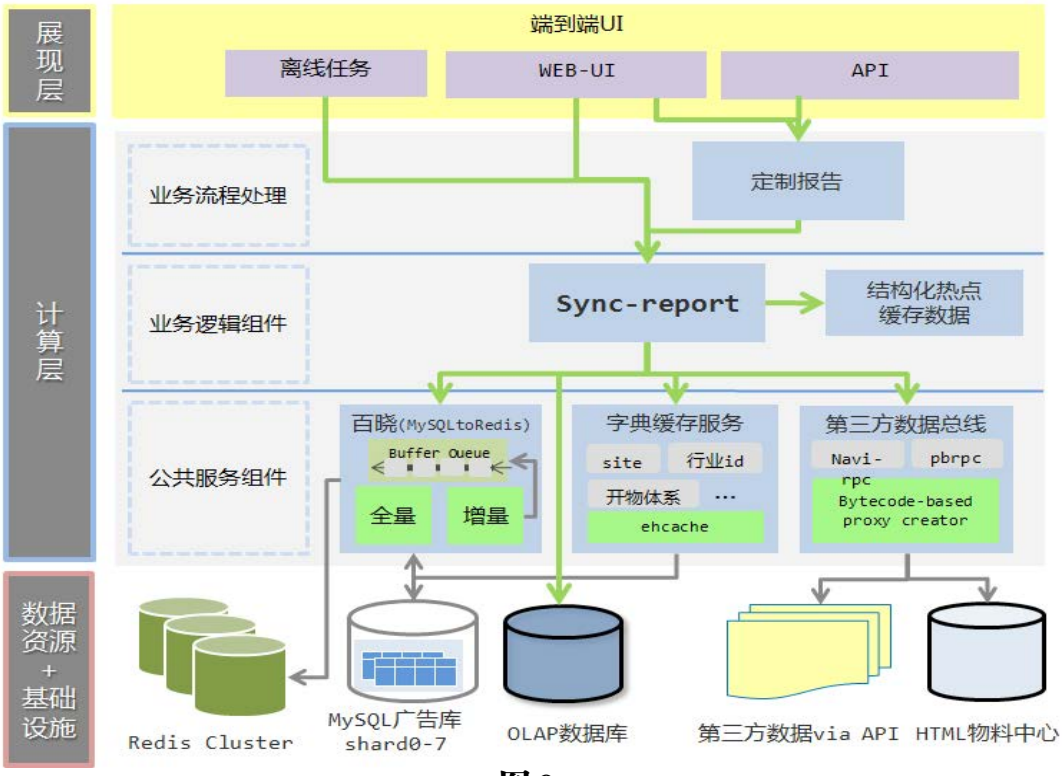

图 9

可观。

最后要说的是，架构只是标准、骨骼，对微服务的讨论不应该让我们忘记了更重要的问题，驱动软件项目成功和失败的重要因素。软因素如团队中人的素质，以及他们如何彼此合作、沟通，这都会对是否使用微服务有很大的影响。在纯技术层面上来讲，应该把重点放在干净的代码、完善到位的测试，并持续关注架构的演化进步，这才是一个软件工程师的根本职责。

新浪微博混合云架构实践挑战——不可变基础设施

作者 蒋生武

【编者按】《微博混合云架构》专栏是 InfoQ 向新浪微博技术团队的系列约稿，本专栏包含 8 篇内容，详细阐述以 DCP 设计理念为指导思想的混合云架构实践。本文是该系列的第二篇，主要讲解了在新浪微博业务背景下 DCP 的不可变基础设施服务。

概述

不可变，顾名思义就是一旦创建，便不再修改。这听起来和我们常见的诉求——灵活的部署环境，似乎背道而驰。对于 DCP 中的基础设施而言，这并不矛盾，因为创建后就没有必要去修改了。整个基础环境作为一种版本化管理的资源，和代码类似。运行时如果有一定要变更的部分，那么直接更新源码，重新创建资源即可。老版的环境依然存在，可以在需要时用于回滚。

为了做到“不可变”，有两个必要条件：

- 能够快速获取所需的资源；
- 业务应用和基础资源之间相互独立。

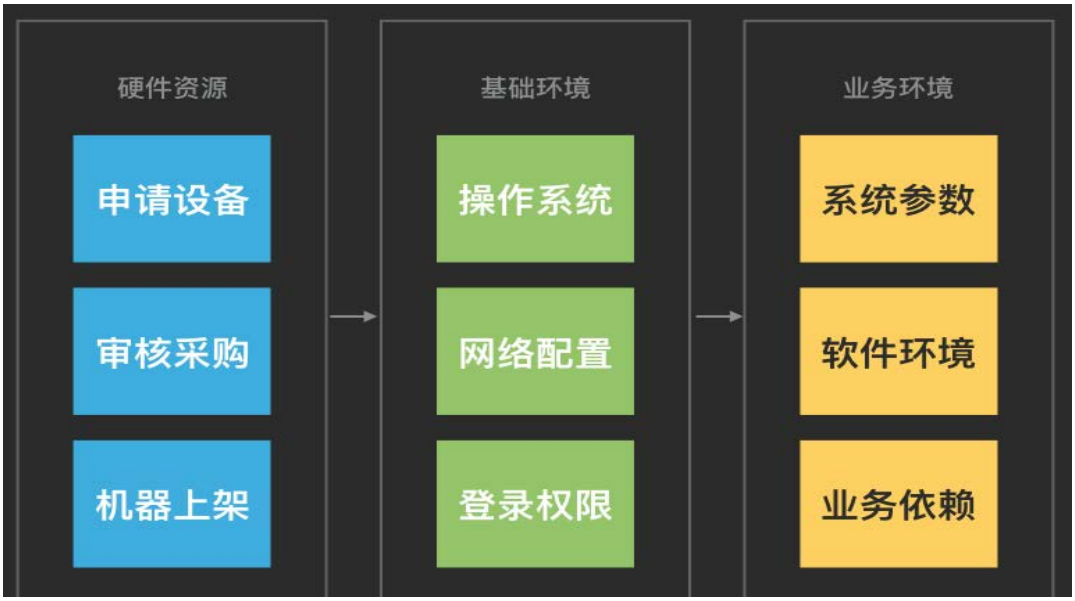

图 1

第一点很容易理解，如果重建创建的过程太慢，比如几个小时甚至数天，那将无法满足线上业务的需求，反而极大提高了运维成本；第二点在容器化技术出现之前，是相当难做到的。业务应用大量依赖本地环境的配置、文件以及缓存等，耦合过于严重导致不同的上层业务需要不同的运行环境，迫使我们对基础设施进行复杂的异构。利用 Docker 技术，可以将业务应用运行时的绝大部分依赖都打包到容器中。这样一来，需要维护的基础环境数量大大减少，便于统一管理，也能够支持更多的业务方来接入。

本文将主要介绍新浪微博混合云架构上的不可变基础设施服务。

微博私有云的传统基础设施

正如概述中所说，私有云的设备申请和环境部署流程过于繁杂，如下图 1 所示。

整个周期可能需要几周到数月之久，一旦有设备损坏，又需要进入另一个同样冗长的报修流程中。而短时间补充计算资源的方式只有从非核心业务中下线服务器挪过来顶上，费时也费力。但是私有云的一些优势是毋

庸置疑的：安全性高、可控度高，相比以虚拟机为基础的公有云性能和稳定性更好。

基础设施方面，传统私有云面临这样几个困难：

- 不同业务部门的环境需求差异非常大，甚至操作系统版本都不同，难以统一管理；
- 随着运行时间增加，即使最初两台主机环境相同，到后来也无法保证一致性，最坏的情况可能造成业务应用部署失败；
- 环境的回滚几乎不可能，只有重装操作系统。

在向混合云架构转变的过程中，我们主要处理的也就是这些问题。

混合云上的基础设施服务

微博混合云的资源分为两个部分：内网主机和阿里云 ECS，我们的目标是对上层业务提供统一、不可变的基础环境。但是如果彻底不可变，反而带来了不小的麻烦。还有人提出拒绝 SSH，目的也是如此。试想仅仅需要临时改变一个系统参数，就必须完整构建一次环境，并且重新部署，这是令人难以接受的。

因此我们决定将不可变的范围缩小，那些耗时相对较长，修改频率又很低的模块纳入不可变范围，这些模块的变更需要重新制作整个环境；而另一些模块放到实时的配置管理系统中，以便在需要的时候进行更新。这样既能保证构建速度，又不带来过高的变更成本的方案，同时解决上面的问题。

正视差异化

不同业务对环境的不同需求是客观存在的，理论上也不可能完全抹平差异，因此我们的服务必须支持各类用户定制自己的环境，自下而上包括

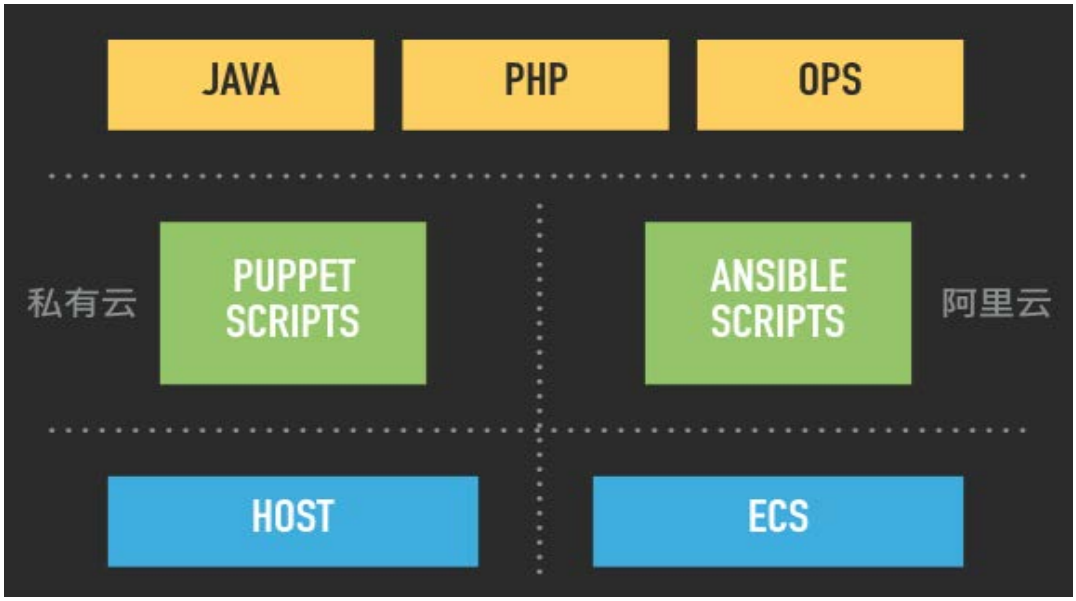

图 2

三部分：主机环境、配置管理脚本和基础容器（见图 2）。

最下层是我们认为不可变的部分，而中间的配置是可能变更的模块，最上面是一些标准的容器，包括面向业务的 JAVA 和 PHP，也包括面向运维的 OPS。这样明确各层的职责，便于分别进行管理。

环境一致性

一致性包括两个方面：

- 资源在使用过程中，环境保持不变：内网主机随着时间推移，会不可避免地被修改，因此我们制作了一个运维使用的基础Docker镜像，即上面的OPS镜像，包括一些工具软件和业务脚本，并将docker.sock挂载进去。这样就可以在容器内部操作Docker Daemon，而其他修改也仅仅只在容器内部，不会污染主机环境。公有云的使用场景原则上不会发生环境漂移的现象，毕竟每次新建的资源使用时间不会超过12小时，同时使用上述运维镜像，更加没有必要操作宿主机。

- 重新创建的和已经在运行的资源环境一致：即使配置未变更，也可能出现不一致的问题，尤其在使用外部软件仓库的情况下，比如脚本中yum update或者yum install docker类似的命令。因此我们使用了内部搭建的软件仓库，可自己对软件的更新频率和范围进行控制，同时也要求脚本中所有安装软件的命令带上版本号，如yum install docker-1.6.2。

基础资源版本化

基础设施即服务，服务即代码，自然可以进行版本化管理。

可以看到图 3 中以 CentOS 7 发行版加基础配置为起点，类似 Git 中分支的概念，衍生出两个不同的基础环境 A 和 B，而从 A 又产生了新的分支 C。每种环境都有自己的版本，在必要的时候可以回滚。需要注意的一点是，各环境的第一个版本在制作时我们需要进行更全面的测试，因为它是后面环境变更的基础，万一出问题，并没有更老的版本可供回滚。

架构演进

初试：Docker Machine

去年微博刚启动混合云项目时，我们首先尝试的是 Docker Machine——一个官方出品的 Docker 环境构建工具。当时这个工具才刚推出不久，还不能直接创建阿里云 ECS，也无法对 CentOS 进行构建，于是我们自行开发了对这两者的支持。它是一个命令行工具，原生支持创建 Swarm 集群，使用起来构建环境只需要简单几步：

- docker-machine create
- 配置自定义的系统环境
- 部署服务

对于个人使用者而言，Docker Machine 非常方便，但是对于混合云项目的需求，有几点它都难以满足：

- Docker启动参数无法完全自定义，当时只能配置少数几个参数，如--insecure-registry。
- 命令行对应的几个函数都是不可导出的，因此无法通过API调用，只能进行二进制依赖。
- 官方宣布不支持CentOS 6.x及之前的版本，而微博内部还有一小部分CentOS 6.5的服务器。
- Docker的安装使用的官方脚本，超时问题比较严重。
- 不够稳定，当时还处于开发阶段，版本迭代过于频繁。

其实在初步尝试过程中，前 4 点都通过修改源码解决了，但那就没办法再合入官方的代码，不是长久之计。因此放弃了 Docker Machine，使用了自己设计的另一套方案。

改进：VM Image & Ansible

前面提到，我们将所有的模块分为不可变和可变两个部分，其实就分别对应了这个方案中的虚拟机镜像和配置管理。利用阿里云 ECS 提供的功能，把耗时长、变更很少的模块直接打到虚机镜像中，而可能变化的模块使用 Ansible 进行管理，能够随时执行。两部分共同构成了一套版本化的环境。如果变更的部分需要被长期使用，通过 Ansible 即时下发后也能够同步到镜像，保证下次构建环境时直接生效。整体设计如图 3 所示。

我们在阿里云 API 的基础上，提供了一个快捷制作镜像的服务，它使用了 Ansible，用户只需要编写 yml 描述文件，即可生成相应的 ECS 镜像。类似于 Dockerfile，环境描述文件也支持 FROM 语法，对应的值就是

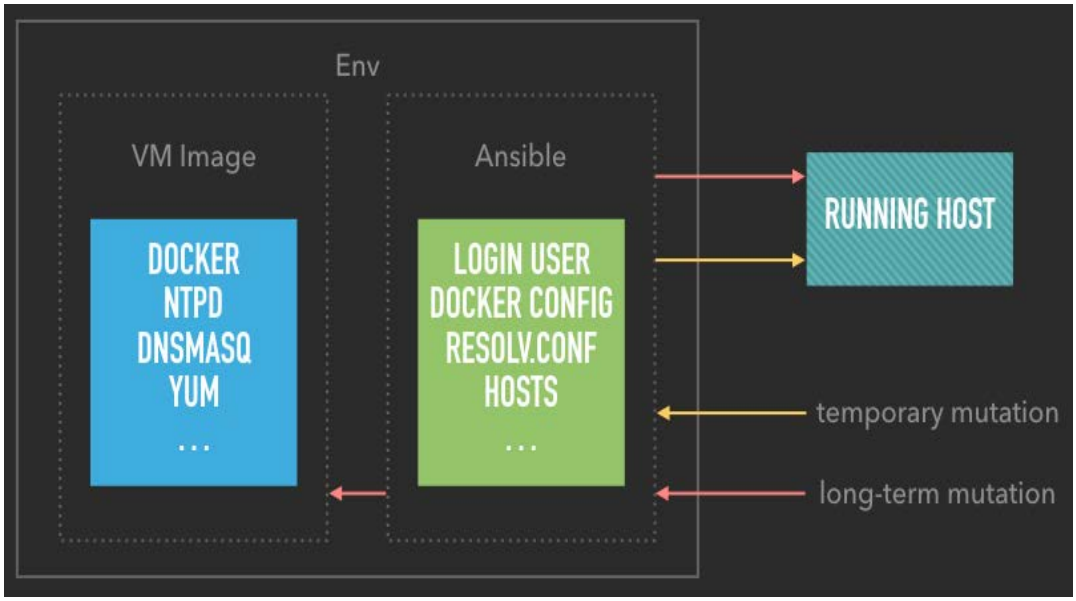

图 3

上层镜像的名字和版本号，再加上 INCLUDE 和 ROLE，分别定义需要执行的 playbook 和 role，例如：

```
001 pFROM weibo_plat_init:1.0
002 INCLUDE user.yml
003 INCLUDE config.yml
004 ROLE hongbao
```

在使用ECS镜像的过程中，有两点也是需要注意的：

- 使用镜像创建ECS后，ntpd的配置会恢复为阿里云默认值，如果使用自定义的需要再次修改。
- 如果在构建镜像时挂载了数据盘，生成镜像前需要删除/etc/fstab中自动挂载的配置，否则该镜像将由于找不到第二块硬盘的分区而无法启动。

另外我们选用 Ansible 作为配置管理工具，主要因为它架构简单、依赖少，只要有 SSH 便可以工作；学习曲线平滑，上手比较容易；模块也足够丰富，能够满足需求。它被广泛诟病的是性能问题，我们对此进行了一

些优化：

- 开启SSH参数，pipelining和ControlPersist；
- 优化playbook，除去非必要模块，减少远程依赖；
- callback模块异步化，Ansible自身会等callback结束才执行后续命令，这点可以大大降低高并发时的耗时；
- 将需要网络传输的操作，如安装软件包、拉取Docker镜像等，内置到ECS镜像中。

最后，从阿里云 ECS 启动完成到初始化结束的用时被缩短到 1 分钟之内。

基础设施发布

准备工作完成之后，就是环境发布了。我们对于公有云的使用是用于弹性扩容，所以发布也主要是针对阿里云 ECS。每次发布可以选择一个 VM 镜像和一套 Ansible 脚本进行构建。整体流程为：

- 在DCP申请阿里云资源；
- 根据选择的VM镜像创建ECS；
- 待ECS启动完成后下发Ansible命令进行初始化；
- 交付上层业务使用。

Ansible Scheduler

上述第三步的命令的批量发布使用了 Ansible Scheduler 来调度。但由于 Ansible 本身不提供 HTTP 接口，因此我们基于它的 Python API 进行了开发。

Scheduler 本身是无状态的，可任意水平扩容，主要负责分发初始化命令，可控制并发度、步长等参数。每个 ECS 镜像都对应 Ansible 的

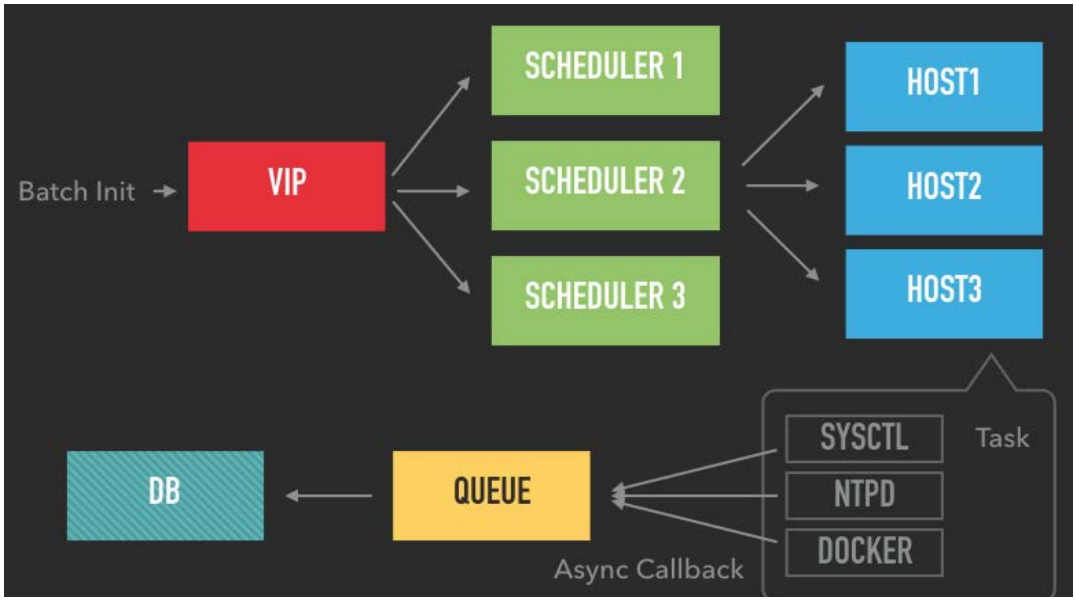

图 4

一个 role，在一台主机上执行的 role 我们称为 Task，其中又包含很多 Action。在每个 Action 开始和结束时，都会触发我们自定义的回调模块，异步将消息写入队列，最后存储到数据库中。DCP 的界面上即可显示所有主机环境初始化的情况，包括执行了哪些模块、耗时和成功与否。（见图 4）

部署公共组件

除了线上应用，DCP 中这整套基础设施构建的系统，也在为其他公共组件提供服务：

```
001 FROM weibo_plat_init:1.0
002
003 INCLUDE user.yml
004 INCLUDE config.yml
005 ROLE hongbao
```

在大规模扩容中，这些组件也是需要支持弹性伸缩的。以 DNS 为例，为避免解析请求跨机房，我们在阿里云上常规部署了两台 DNS 服务。同时由于所有扩容的 ECS 都是新建的，不存在 DNS 缓存，业务首次启动会有大量域名解析的请求，也需要保证 DNS 服务的高可用。这里我们使用了阿里

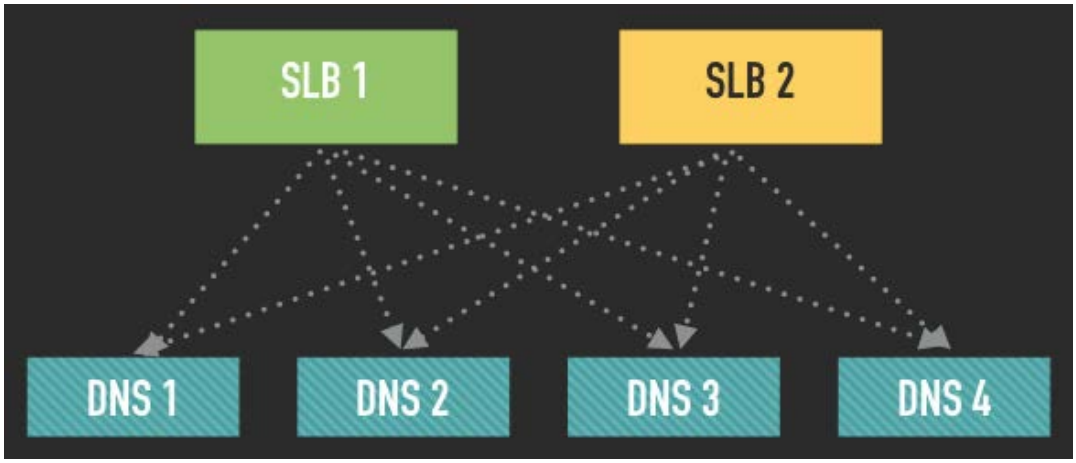

图 5

云的内网 SLB 服务，作为 VIP 对 DNS 做负载均衡。两个 SLB 分别指向所有的 DNS 服务器，设置 UDP 转发，并且增加健康检查，便于摘除不可用节点。（见图 5）

基础监控

由于构建了统一的环境，整套基础监控也是标准化的，包括 CPU、内存、硬盘、网络、Docker 进程等等。其中对于专线带宽的监控是相当重要的一个部分，跨云的专线没法和自有机房的线路相比，不论是稳定性和带宽使用，都需要实时监控。

首先我们根据网络资源的分配在阿里云建立 VPC 时选择一个独立的网段，如 10.85.0.0/16，然后每台 ECS 创建后都会安装专线带宽监控的软件，只要过滤掉所有在上述网段内部传输的流量，就是穿越专线的流量，同时也定时监测自有机房到阿里云的网络延迟。最后将监控数据推送到中心存储节点，通过 grafana 进行展示，见图 6。

专线监控精确到单机，因此也可以自行设置聚合逻辑，从而统计各业务使用的总体带宽。

图 6

总结

基础设施的快速构建为新浪微博混合云的弹性扩容提供了强有力的支持，不可变在很多时候是简单而高效的，但不是绝对的。实际应用中，我们也需要将基础环境中长期稳定和可能变更的模块分别处理，从而在构建速度和变更成本之间取得平衡。

分布式 MySQL 集群方案的探索与思考

作者 张成远

背景

数据库作为一个非常基础的系统，任何一家互联网公司都会使用，数据库产品也很多，有 Oracle、SQL Server 、MySQL、PostgeSQL、MariaDB 等，像 SQLServer/Oracle 这类数据库在初期可以帮业务搞定很多棘手的事情，我们可以花更多的精力在业务本身的发展上，但众所周知也得交不少钱。

涉及到钱的事情在公司发展壮大以后总是会回来重新审视这个事情的，在京东早期发展的过程中确实有一些业务的数据就是直接存在 oracle 或者 sqlserver 中。

后来随着业务的发展以及数据量访问量的不断增加及成本等方面的考虑，从长远考虑需要把这些业务用免费的 MySQL 来存，但单机的 MySQL 往往无法直接抗住这些业务，自然而然的我们就需要考虑引入分布式的 MySQL 解决方案帮助业务去 SQLServer/Oracle 以及支撑未来的发展。

方案选型对比及京东实现方案

说到分布式 MySQL 的解决方案一般来说解决方案主要就两种，客户端

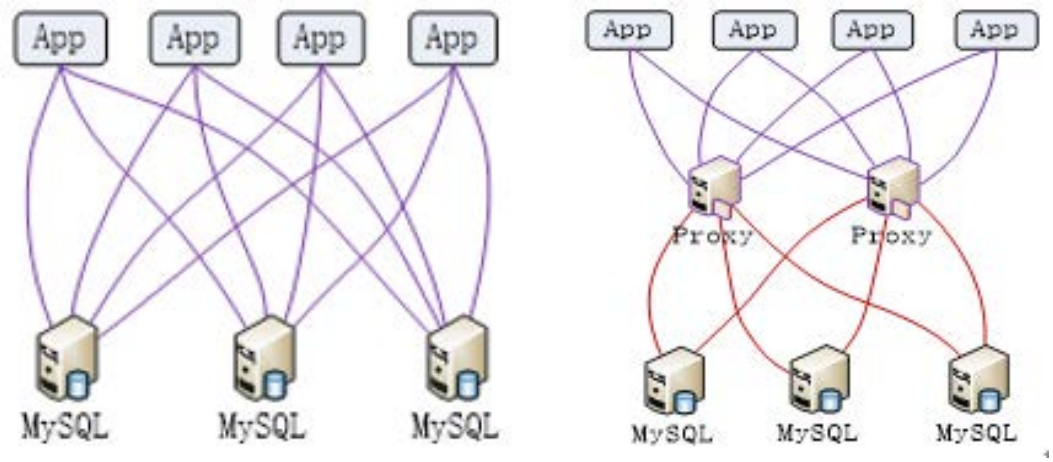

图 1

的方案或者中间代理的方案，如图 1 所示。

这两种方案各有各的优缺点：客户端的方案是指会给业务提供一个专门的客户端的包，这种方案在实现上会更容易一点，如果公司需要快速出一个相对通用的解决方案，客户端的方案可以优先考虑。

客户端方案需要为不同的语言提供不同的客户端的包，这点有所局限。客户端方案只需要走一段网络，理论上性能会更好一点。

客户端方案对业务有侵入，有一些系统部署及实现方面的可能可以控制得更好，但对业务本身不友好，客户端包升级等方面比较麻烦。

中间代理的方案是指采用一个兼容 MySQL 协议的代理的方式，业务可以使用任何语言的 MySQL 客户端的包，对业务本身无侵入的，这种方案相对来说是最友好的。

中间代理方案开发难度上来说门槛会更高一点，需要考虑前后端的东西，尤其是与 MySQL 端交互时自己解析协议的情况下会更复杂一些。中间代理方案多走一段 TCP，对性能理论上会有一些影响。

上述两种方案有一个非常重要的因素没有提及，在实际生产环境中面临一个非常现实的问题是 MySQL 能支持的连接数是有限的。以 MySQL5.5

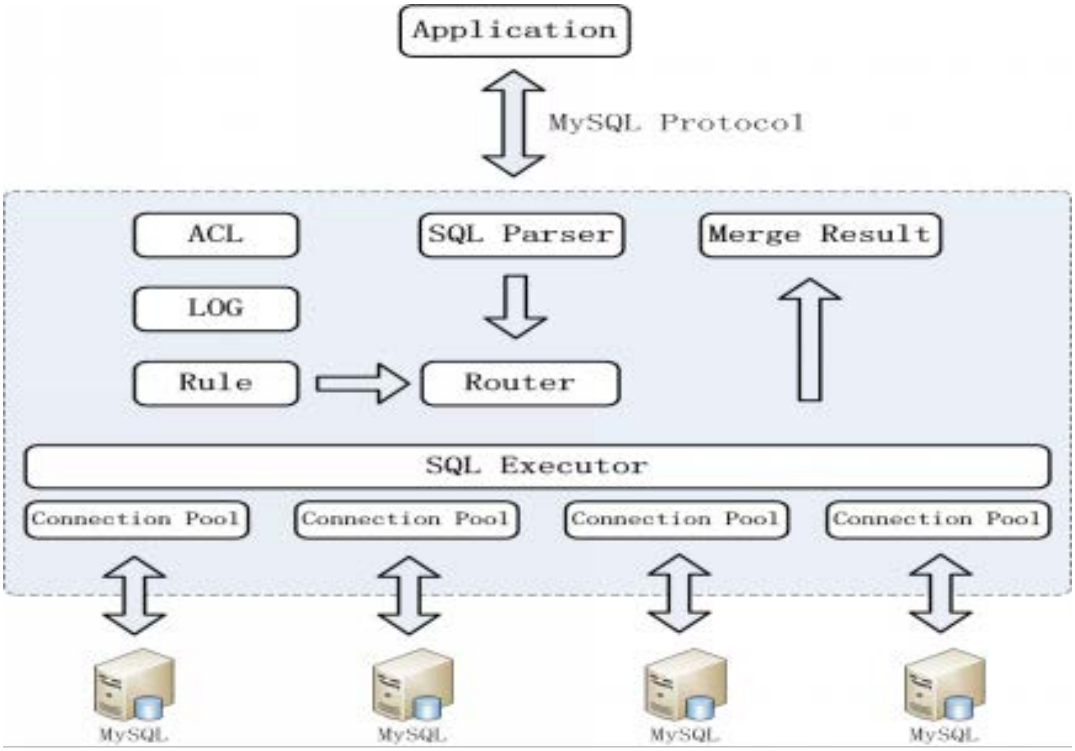

图 2

来说假设一个 MySQL 实例配置 1000 个连接，业务应用实例部署了 100 个，每个应用实例的数据库连接池配置 20 个，采用客户端方案这个 MySQL 实例都没法正常工作了。

大多数情况下并不是每个应用实例的每条连接都是活跃的，中间代理的方案可以很好的解决这个问题，应用实例可以有很多连接打到代理上，代理只需要维护较少的与 MySQL 的连接即可满足需求，代理与 MySQL 之间的连接会被业务打过来的访问重复使用。

另外关于多走一次 TCP 对性能的影响，从我们的实际经验来看其实可以忽略不计，业务实例一多优先遇到的是 MySQL 连接数的问题，从这个角度来说中间代理的方案会更优。

我们采用的就是中（见图 2）。

间代理的方案，京东的分布式 MySQL 方案由很多部分组成，有

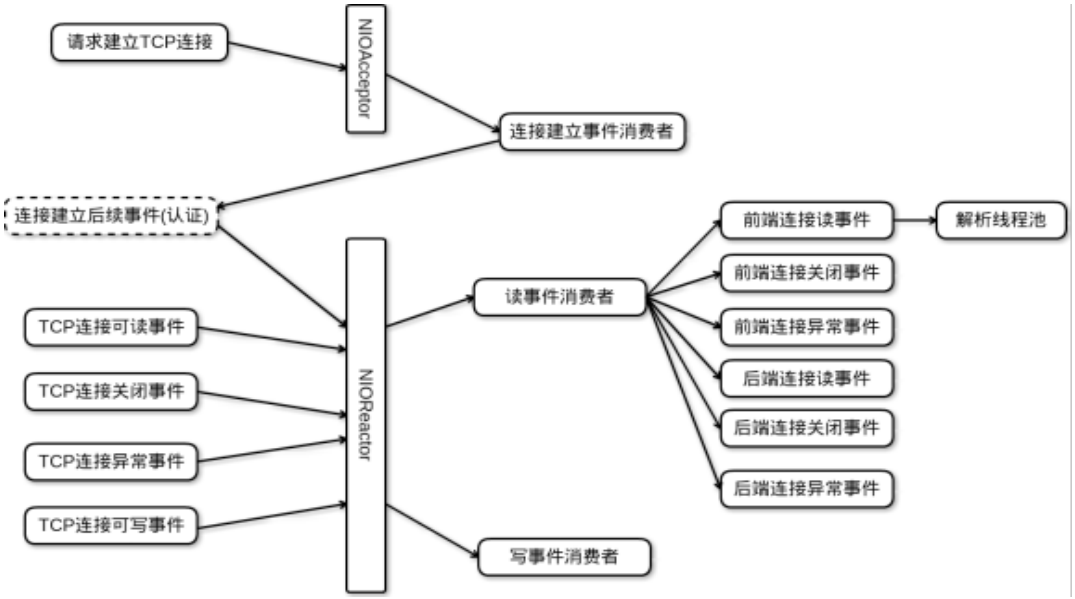

图 3

JManager、 JProxy、 JTransfer、JMonitor、JConsole、MySQL，在实际部署的时候还涉及到 LVS 以及域名系统等。

JManager 是中心管理节点，这个节点负责统一管理系统的元信息，元信息包括路由信息、权限管理信息、资源相关的信息等。

JProxy 就是一个兼容 MySQL 协议的代理，负责把客户端发送过来的 SQL 按照路由规则发送到相应的数据库节点上，再把返回的结果进行合并并返回给客户端。JProxy 在启动的时候会先去 JManager 中拉取相关的元信息，并在自己的内存中维护一份，平时使用的时候只用自身内存维护的这一份就可以了。

JProxy 的内部实现原理图 3 所示。

JTransfer 是在线迁移系统，我们针对业务的数据进行拆分以后，比如某个 MySQL 实例上有 32 个库，等到业务数据量继续增大以后在这个实例上就放不下了，我们就需要往整个集群中加 MySQL 实例，将之前的 32 个库中一部分迁移到这个新增加的实例上，如何在不停业务的情况完成数

据的在线迁移就是 JTransfer 这个系统来保证的。

JConsole 系统可以理解为将多个业务的中心管理节点整合起来的一个后台管理控制系统，这个系统可以与每个 JManager 交互。在具体使用的时候，业务方需要申请创建库表、拆分规则、什么权限、对哪些 IP 授权，我们会通过 JConsole 系统与 JManager 交互完成元数据的配置。

JMonitor 系统会将各个业务的 jproxy 以及 MySQL 相关的信息采集起来，整合到一起形成一个统一的监控系统，完成对系统的全面详尽的监控。

网络模型

JProxy 作为一个非常典型的代理服务，程序本身的性能非常关键，具体在实现的时候我们参考了 Nginx 的网络模型。

大家都知道 Nginx 的性能非常高，根据机器核数配置相应的 worker 数就可以，每个 worker 可以理解为围绕一个 epoll 把前后端的连接以完全基于事件驱动的方式串在一起，避免了上下文切换避免了锁等待等各种可能阻塞或者耗时的操作。

同样的网络模型也可以参考一下 Redis 的实现，redis 虽然不像 nginx 需要考虑前后端连接的处理，但 redis 的模型也是一种非常类似的经典的实现方式。

JProxy 整个网络模型如图所示，采用一个全局的 nioacceptor 以及多个 nioreactor，由 nioacceptor 统一 accept 连接，之后把连接分给某个 nioreactor。

nioreactor 可以理解为底层就是一个 epoll（java nio 实现），前后端的连接都是注册在这个 epoll 上，我们只需要根据事件是读事件或写事件调用相应的回调函数即可。这种模型的特点是系统几乎没有太多的上

下文切换，而且性能很高。

基于事件驱动的网络模型的好处是性能很高，但问题也很明显，编写时复杂度非常高，一条 SQL 发送过来到收到结果的上下文被切成很多片段，同一时刻有来自很多不同上下文的不同的片段要处理，全程只有一个进（线）程来处理这些片段（暂且假设 NIOReactor 只配置成一个），所以在实现的过程中要求把所有的细节都考虑非常周全，一旦某个片段的处理有阻塞或者耗时，整个程序都将阻塞，个人觉得这种编程方式有点反人类思维。

关于分布式事务的思考

另外关于分布式事务的支持也是一个大家可能比较感兴趣的点，基于 MySQL 的方式来做分布式数据库的时候分布式事务是不可能满足严格的分布式事务语义的。

数据库事务有 ACID 四个属性，分别是原子性、一致性、隔离性、持久性。

原子性 (Atomicity) 的意思是整个事务最终只能是要么成功要么失败，不能存在中间状态，如果发生错误了就需要回滚回去，就像这个事务从来没有执行过一样。

一致性 (Consistency) 是指系统要处于一个一致的状态，不能因为并发事务的多少影响到系统的一致性，举个典型的例子就是转帐的情况，假设有 ABC 三个帐号各有 100 元，那么不管这三个帐号之间怎么转账，整个系统总的额度是 300 元这一点是应该是不变的。其实 ACID 里的一致性更多的是应用程序需要考虑的问题，和分布式系统里的 CAP 里的一致性完全不是一个概念。

隔离性 (Isolation)，本质上是解决并发执行的事务如何保证数据库

状态是正确的，抽象描述叫可串行化，就是并发的事务在执行的时候效果要求达到看起来像是一个个事务串行执行的效果。有冲突的事务之间的隔离性如果保证不了会引起前面的一致性（consistency）也无法满足。

每个事务包含多个动作，这些动作如果按照事务本身的顺序依次执行就是所谓的串行执行，这些动作也可以重新排列，排列完以后的动作如果效果可以等价于事务串行执行的效果我们就叫做可串行化调度。

实际实现的时候往往采用的是冲突可串行化，这个条件比可串行化要求会更高一点，规定了一些读写顺序规定了一些访问冲突的情况规定了哪些情况两个事物的动作可以调换哪些是不可以的，可以理解为冲突可串行化是可串行化的充分条件。

持久性（Durability），在事务完成以后所有的修改可以持久的保存在数据库中，一般会采用 WAL 的方式，会把操作提前记录到日志中来保证即使操作还没有刷到磁盘就宕机的情况下有日志可以恢复。介绍完事务的 ACID 属性以后，我们再来分析为什么基于 MySQL 无法提供严格的分布式事务语义的支持。

如果客户端发送的 SQL 只涉及到一个节点，那自然是可以保证事务的，但是如果客户端发送的 SQL 涉及到两个及以上节点的 SQL，那就无法保证事务语义了。

原因主要是两个，一是原子性无法保证，另一个是隔离性无法保证。在一个节点 commit 成功以后，在另外的节点 commit 失败了，这个事务就处在一个中间状态，此时原子性被打破。

引起的另一个问题就是隔离性，这个事务的一部分提交了，另一部分未提交，此时该事务正常是不该被读取到的，但是提交成功的部分会被其他事务读到，此时就无法保证隔离性了。

另外就算是涉及多个节点的操作都是成功的，理论上来说也是无法保证隔离性的。因为假设 A 事务的一个节点先 commit 成功，其他的节点后 commit 成功，而此时 B 事务在读取的时候可能会读取到了 A 事务最早 commit 成功的那部分内容，却没有读到后来 commit 成功的内容，此时依然无法保证隔离性。

更本质一点的原因是 MySQL 的事务都是每个实例维护自身的事务 ID，而基于 MySQL 集群的分布式方案没有一个全局的事务 ID 来标识每个 MySQL 实例上的事务以及全局事务的元信息的管理，所以无法做到严格的分布式事务语义。

但实际上绝大多数业务对这个需求未必那么强烈，因为绝大多数的业务逻辑都是可以拆分的，拆成一个个只落在一个分库里的操作在绝大多数场景下是完全可行的，而且拆分完以后也会更可控，所以这个问题在我们支撑业务的过程中也不是一个特别大的问题。

生产环境监控很关键

在实际生产环境中有很多方面都非常重要，高可用高可靠可扩展等，但是除了这些之外还有一个非常关键的是监控。

一个再健壮再牛 x 的系统都需要配备完善的监控系统，监控系统是生产环境中非常重要的一道防线，没有监控的系统就像是在裸奔，线上突发状况很多完善的监控系统可以做到第一时间发现问题及时定位以及解决问题。

物理机监控

我们在生产环境中会对系统所在物理机进行监控，京东有一个专门的物理机监控系统，可以监控包括 CPU、内存、网卡、TCP 连接数、磁盘使

用情况、机器 load 等很多基础指标，针对这些指标可以设置相应的报警阈值，当超过一定阈值时会以邮件及短信的方式报警。

存活监控

但物理机的监控对于具体的系统的来说是远远不够的，我们还需要关注很多系统本身的信息，首先要有存活监控，这是最基本的。一个系统在线上运行的时候服务本身宕掉一定要求是可以第一时间监控到的。

但除了物理机监控以外，还有一个非常关键的是存活监控。系统的一切前提是可以活着，我们在每个模块都会提供相应的 http 接口，接入公司的统一监控平台，一旦有异常统一监控平台会及时通知相应的负责人。

系统内部状态可视化监控

除了活下来以后，如何活得更好也是很关键的，所以我们还有专门针对分布式 MySQL 集群的 JMonitor 系统，该系统会整合各个模块的内部详细状态信息，包括慢查询、用户访问情况以及数据分布情况等。

一句话一个稳定健壮的系统一定要配备相应的完善的监控系统。今天我的分享就是这些，主要就是介绍一些分布式 MySQL 的相关方案以及京东是怎么做的，讨论了一下分布式事务的问题，最后是一小部分生产实践经验。

张成远，京东资深架构师，ArchSummit 北京专题讲师《Mariadb 原理与实现》，开源项目 speedy 作者。毕业于东北大学，硕士阶段研究分布式数据库相关方向，2012 年加入京东数据库系统研发团队。擅长高性能服务器开发、分布式数据库、分布式存储 / 缓存等大规模分布式系统架构。目前负责京东分布式数据库系统的架构与研发工作，主要关注云数据库及分布式数据库相关领域。

Q&A

问题 1：请介绍下分布式事务保证数据最终一致性的具体方案例子。

回答：首先分布式事务涉及到的一致性和CAP中一致性是两个概念，事务ACID属性中的一致性不涉及最终一致性，对于关系型数据库中事务的概念，我的理解都是强一致的（通过原子性和隔离型保证）。只有涉及到某一个节点（内容是相同的情况）多副本之间的复制问题才会涉及到弱一致性或者最终一致性(CAP中C)的问题。而分布式事务本身如果保证了原子性和隔离性，数据库层面就提供了一致性保证，其余的是应用逻辑层面保证。如果问的是数据库主从复制之间的一致性问题，这个事情本质上和事务（ACID的C）的一致性就没有关系了，所以这个问题本身可能有待商榷。

问题 2：分布式事务如何支持，现在可以支持多大规模的集群。

回答：基于Mysql的分布式集群方案无法保证严格的分布式事务语义，但是在实际使用的时候看业务情况，如果事务之间不怎么冲突的情况下也是ok的，如果可以改成只涉及一个分库的情况下那就绕开分布式事务的问题了。另外支持的集群，我们其实是根据业务来划分资源的，目前整体资源不能说特别大，千台规模。

问题3：JProxy是否可以支持所有复杂sql查询，主要是夸库的关联查询，具体内部逻辑可否介绍下？

回答：我们目前不支持夸库关联查询，从业务层面来解决。因为大表之间分库以后如果要支持跨库关联查询的话，作为一个OLTP系统在实际生产环境估计就没法用了。

问题4：请介绍下MySQL实际应用中主从复制的方案，以及主从的数据差异会在什么程度？

回答：这个其实更多的是主从之间关注的问题，一般会采用基于mix的模式。另外主从差异这个不同业务不一样，加上严格的监控，正常访问的情况下一般不会出现延迟，但是如果涉及到业务倒数据或者突增的访问量可能会引起延迟，所以这个不太好参考，如果有异常我们都会第一时间及时介入处理。

问题5：长时间SQL不会造成堵塞吗？

回答：主要看这条SQL具体是做什么的，如果是抽数据，就正常抽就

可以了。如果有阻塞基本都是因为在MySQL端因为锁冲突等原因造成阻塞，最终可能是这个事务被abort掉或者最终抢到锁成功做完了这个事务。

问题6：请介绍一下JTransfers的工作机制，以及实现过程中最难的部分。

回答：迁移确实是比较刺手的一件事情，要考虑的细节很多。大体的步骤是：我们提交迁移计划，指定什么时间开始迁移，到时间点以后JTransfer就会自动迁移。JTransfer一开始是dump源分库的数据，然后将这些数据恢复到目标实例上，但是在这个期间业务是正常访问的，需要将增量数据迁移完，所以会有追增量过程。当增量追到一定程度，我们会阻塞这个库的访问，最后将剩余的少量数据迁移完。因为最后剩余数据量不多的时候，阻塞过程其实很短暂，所以对业务影响非常小。

最难的部分是：整个迁移过程中的路由变更，要保证路由变更的过程中数据不能写花，且变更以后的路由要准确的推送到JProxy中，由JManager和多个JProxy之间在变更路由的时候采用类似两阶段提交的协议，从而保证路由的变更是正确的。

问题7：可以分享一下JProxy的并发性能优化，以及JProxy中间状态的异常与恢复机制吗？谢谢！

回答：并发性能优化我们主要是通过采用基于事件驱动的网络模型，这种方式的特点是避免上下文切换避免锁的开销，但是代价的话刚才也说了需要考虑得非常周全，把一个上下文切成很多片段，不太符合人类思维。

JProxy中间件状态的异常与恢复机制这个我不是太理解什么含义哈，我的理解是如果jproxy运行过程中访问出异常了怎么处理，如果是某个连接过来的sql出了问题我们的做法是将整个连接涉及到的资源都关闭，把该次查询涉及到的前后端资源清理干净，这样就不会影响到其他客户端的访问。正常来说不应该出现这种情况，所以也需要完善的日志信息以及监控信息。

为什么我不再使用 MVC 框架

作者 Jean-Jacques Dubray 译者 张卫滨

在我最近的工作中，最让人抓狂的就是为前端开发人员设计 API。我们之间的对话大致就是这样的：

开发人员：这个页面上有数据元素x,y,z…，你能不能为我创建一个API，响应格式为{x: , y:, z: }

我：好吧

我甚至没有进行进一步的争论。项目结束时会积累大量的 API，这些 API 与经常发生变化的页面是关联在一起的，按照“设计”，只要页面改变，相应的 API 也要随之变化，而在此之前，我们甚至对此毫不知情，最终，由于形成因素众多且各平台之间存在些许差异，必须创建非常多的 API 来满足这些需求。Sam Newman 甚至将这种制度化的过程称之为 BFF 模式，这种模式建议为每种设备、平台当然还包含 APP 版本开发特定的 API。Daniel Jacobson 在接受 InfoQ 的采访时曾指出，Netflix 颇为勉强地将“体验式 API”与“临时 API（Ephemeral API）”划上了等号。 唉……

几个月前，我开始思考是什么造成了如今的这种现象，该做些什么来应对它，这个过程使我开始质疑应用架构中最强大的理念，也就是 MVC，我感受到了函数式反应型编程（reactive）的强大威力，这个过程致力于流程的简化，并试图消除我们这个行业在生产率方面的膨胀情绪。我相信

你会对我的发现感兴趣的。

在每个用户界面背后，我们都在使用 MVC 模式，也就是模型 - 视图 - 控制器（Model-View-Controller）。MVC 发明的时候，Web 尚不存在，当时的软件架构充其量是胖客户端在原始网络中直接与单一数据库会话。但是，几十年之后，MVC 依然在使用，持续地用于 OmniChannel 应用的构建。

Angular 2 正式版即将发布，在这个时间节点重估 MVC 模式及各种 MVC 框架为应用架构带来的贡献意义重大。

我第一次接触到 MVC 是在 1990 年，当时 NeXT 刚刚发布 Interface Builder（让人惊讶的是，如今这款软件依然发挥着重大的作用）。当时，我们感觉 Interface Builder 和 MVC 是一个很大的进步。在 90 年代末期，MVC 模式用到了 HTTP 上的任务中（还记得 Struts 吗？），如今，就各个方面来讲，MVC 是所有应用架构的基本原则。

MVC 的影响十分深远，以致于 React.js 在介绍他们的框架时都委婉地与其划清界限：“React 实现的只是 MVC 中视图（View）的部分”。

当我去年开始使用 React 的时候，我感觉它在某些地方有着明显的不同：你在某个地方修改一部分数据，不需要显式地与 View 和 Model 进行交互，整个 UI 就能瞬间发生变化（不仅仅是域和表格中的值）。这也就是说，我很快就对 React 的编程模型感到了失望，在这方面，我显然并不孤独。我分享一下 Andre Medeiros 的观点：

React 在很多方面都让我感到失望，它主要是通过设计不佳的 API 来引导程序员 […] 将多项关注点混合到一个组件之中。

作为服务端的 API 设计者，我的结论是没有特别好的方式将 API 调用组织到 React 前端中，这恰恰是因为 React 只关注 View，在它的编程模型中根本不存在控制器。

到目前为止，Facebook 一直致力于在框架层面弥合这一空白。React 团队起初引入了 Flux 模式，不过它依然令人失望，最近 Dan Abramov 又提倡另外一种模式，名为 Redux，在一定程度上来讲，它的方向是正确的，但是在将 API 关联到前端方面，依然比不上我下面所介绍的方案。

Google 发布过 GWT、Android SDK 还有 Angular，你可能认为他们的工程师熟知何为最好的前端架构，但是当你阅读 Angular 2 设计考量的文章时，便会不以为然，即便在 Google 大家也达成这样的共识，他们是这样评价之前的工作成果的：

Angular 1 并不是基于组件的理念构建的。相反，我们需要将控制器与页面上各种［元素］进行关联（attach），其中包含了我们的自定义逻辑。根据我们自定义的指令如何对其进行封装（是否包含 isolate scope？），scope 会进行关联或继续往下传递。

基于组件的 Angular 2 看起来能简单一点吗？其实并没有好多少。Angular 2 的核心包本身就包含了 180 个语义（semantics），整个框架的语义已经接近 500 个，这是基于 HTML5 和 CSS3 的。谁有那么多时间学习和掌握这样的框架来构建 Web 应用呢？当 Angular 3 出现的时候，情况又该是什么样子呢？

在使用过 React 并了解了 Angular 2 将会是什么样子之后，我感到有些沮丧：这些框架都系统性地强制我使用 BFF“页面可替换模式（Screen Scraping）”模式，按照这种模式，每个服务端的 API 要匹配页面上的数据集，不管是输入的还是输出的。

此时，我决定“让这一切见鬼去吧”。我构建了一个 Web 应用，没有使用 React、没有使用 Angular 也没有使用任何其他的 MVC 框架，通过这种方式，我看一下是否能够找到一种在 View 和底层 API 之间进行更好协

图 1：用于生成站点 Slider 组件 HTML 的函数

作的方式。

就 React 来讲，我最喜欢的一点在于 Model 和 View 之间的关联关系。React 不是基于模板的，View 本身没有办法请求数据（我们只能将数据传递给 View），看起来，针对这一点进行探索是一个很好的方向。

如果看得足够长远的话，你会发现 React 唯一的目的就是将 View 分解为一系列（纯粹的）函数和 JSX 语法：

```
<V params={M}/>
```

它实际上与下面的格式并没有什么差别：

```
V = f( M )
```

例如，我当前正在从事项目的 Web 站点 Gliiph 就是使用这种函数构建的，见图 1。

这个函数需要使用 Model 来填充数据，见图 2。

如果用简单的 JavaScript 函数就能完成任务，我们为什么还要用 React 呢？

虚拟 DOM（virtual-dom）？如果你觉得需要这样一种方案的话（我

```
home.logo = home.imageDir+'logo-2.png' ;
home.menuCallToAction = { label: "Join our beta program", link: "http://gliiph.com" , button: "Sign up"};
home.sliders = {} ;
home.sliders.id = 'home' ;
home.sliders.frames = [
    {image: home.imageDir+"slider-1.jpg", title: "Renovate<br>your<br>Website", caption: "Our platform helps you migrate your most valued conten
   ,{image: home.imageDir+"slider-2.jpg", title: "Protect<br>your<br>Assets", caption: "Lock down admin pages, throttle or block unwanted traffi
   ,{image: home.imageDir+"slider-3.jpg", title: "Delight<br>your<br>Visitors", caption: "We store your assets in S3 and use the latest caching
  ] ;
```

图 2：支撑 slider 的 Model

并不确定有很多的人需要这样），其实有这样的可选方案，我也期望开发出更多的方案。

GraphQL ？并不完全如此。不要因为 Facebook 大量使用它就对其产生误解，认为它一定是对你有好处的。GraphQL 仅仅是以声明的方式来创建视图模型。强制要求 Model 匹配 View 会给你带来麻烦，而不是解决方案。React 团队可能会觉得使用“客户端指定查询（Client-specified queries）”是没有问题的（就像反应型团队中那样）：

GraphQL 完全是由 View 以及编写它们的前端工程师的需求所驱动的。[…] 另一方面，GraphQL 查询会精确返回客户端请求的内容，除此之外，也就没什么了。

GraphQL 团队没有关注到 JSX 语法背后的核心思想：用函数将 Model 与 View 分离。与模板和“前端工程师所编写的查询”不同，函数不需要 Model 来适配 View。

当 View 是由函数创建的时候（而不是由模板或查询所创建），我们就可以按需转换 Model，使其按照最合适的形式来展现 View，不必在 Model 的形式上添加人为的限制。

例如，如果 View 要展现一个值 v，有一个图形化的指示器会标明这个值是优秀、良好还是很差，我们没有理由将指示器的值放到 Model 中：函数应该根据 Model 所提供的 v 值，来进行简单的计算，从而确定指示器的值。

现在，把这些计算直接嵌入到 View 中并不是什么好主意，使 View-Model 成为一个纯函数也并非难事，因此当我们需要明确的 View-Model 时，就没有特殊的理由再使用 GraphQL 了：

V = f(vm(M))

作为深谙 MDE 之道的人，我相信你更善于编写代码，而不是元数据，不管它是模板还是像 GraphQL 这样的复杂查询语言。

这个函数式的方式能够带来多项好处。首先，与 React 类似，它允许我们将 View 分解为组件。它们创建的较为自然的界面允许我们为 Web 应用或 Web 站点设置“主题”，或者使用不同的技术来渲染 View（如原生的方式）。函数实现还有可能增强我们实现反应型设计的方式。

在接下来的几个月中，可能会出现开发者交付用 JavaScript 函数包装的基于组件的 HTML5 主题的情况。这也是最近这段时间，在我的 Web 站点项目中，我所采用的方式，我会得到一个模板，然后迅速地将其封装为 JavaScript 函数。我不再使用 WordPress。基本上花同等的工夫（甚至更少），我就能实现 HTML5 和 CSS 的最佳效果。

这种方式也需要在设计师和开发人员之间建立一种新型的关系。任何人都可以编写这些 JavaScript 函数，尤其是模板的设计人员。人们不需要学习绑定方法、JSX 和 Angular 模板的语法，只掌握简单的 JavaScript 核心函数就足以让这一切运转起来。

有意思的是，从反应型流程的角度来说，这些函数可以部署在最合适的地方：在服务端或在客户端均可。

但最为重要的是，这种方式允许在 View 与 Model 之间建立最小的契约关系，让 Model 来决定如何以最好的方式将其数据传递给 View。让 Model 去处理诸如缓存、懒加载、编配以及一致性的问题。与模板和

GraphQL 不同，这种方式不需要从 View 的角度来直接发送请求。

既然我们有了一种方式将 Model 与 View 进行解耦，那么下一个问题就是：在这里该如何创建完整的应用模型呢？“控制器”该是什么样子的？为了回答这个问题，让我们重新回到 MVC 上来。

苹果公司了解 MVC 的基本情况，因为他们在上世纪 80 年代初，从 Xerox PARC“偷来了”这一模式，从那时起，他们就坚定地实现这一模式，见图 3。

Andre Medeiros 曾经清晰地指出，这里核心的缺点在于， MVC 模式是“交互式的”（这与反应型截然不同）。在传统的 MVC 之中，Action（Controller）将会调用 Model 上的更新方法，在成功（或出错）之时会确定如何更新 View。他指出，其实并非必须如此，这里还有另外一种有效的、反应型的处理方式，我们只需这样考虑，Action 只应该将值传递给 Model，不管输出是什么，也不必确定 Model 该如何进行更新。

那核心问题就变成了：该如何将 Action 集成到反应型流程中呢？如果你想理解 Action 的基础知识的话，那么你应该看一下 TLA+。TLA 代表的是“Action 中的逻辑时序（Temporal Logic of Actions）”，这是由 Dr. Lamport 所提出的学说，他也因此获得了图灵奖。在 TLA+ 中，Action 是纯函数：

```
data’ = A (data)
```

我真的非常喜欢 TLA+ 这个很棒的理念，因为它强制函数只转换给定的数据集。

按照这种形式，反应型 MVC 看起来可能就会如下所示：

```
V = f( M.present( A(data) ) )
```

这个表达式规定当 Action 触发的时候，它会根据一组输入（例如用

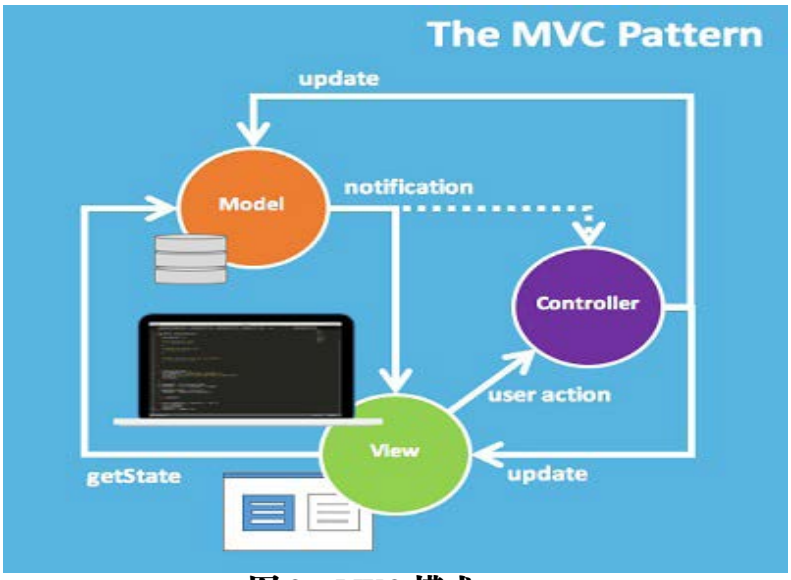

图 3：MVC 模式

户输入）计算一个数据集，这个数据是提交到 Model 中的，然后会确定是否需要以及如何对其自身进行更新。当更新完成后，View 会根据新的 Model 状态进行更新。反应型的环就闭合了。Model 持久化和获取其数据的方式是与反应型流程无关的，所以，它理所应当地“不应该由前端工程师来编写”。不必因此而感到歉意。

再次强调，Action 是纯函数，没有状态和其他的副作用（例如，对于 Model，不会包含计数的日志）。

反应型 MVC 模式很有意思，因为除了 Model 以外，所有的事情都是纯函数。公平来讲，Redux 实现了这种特殊的模式，但是带有 React 不必要的形式，并且在 reducer 中，Model 和 Action 之间存在一点不必要的耦合。Action 和接口之间是纯粹的消息传递。

这也就是说，反应型 MVC 并不完整，按照 Dan 喜欢的说法，它并没有扩展到现实的应用之中。让我们通过一个简单的样例来阐述这是为什么。

假设我们需要实现一个应用来控制火箭的发射：一旦我们开始倒计时，

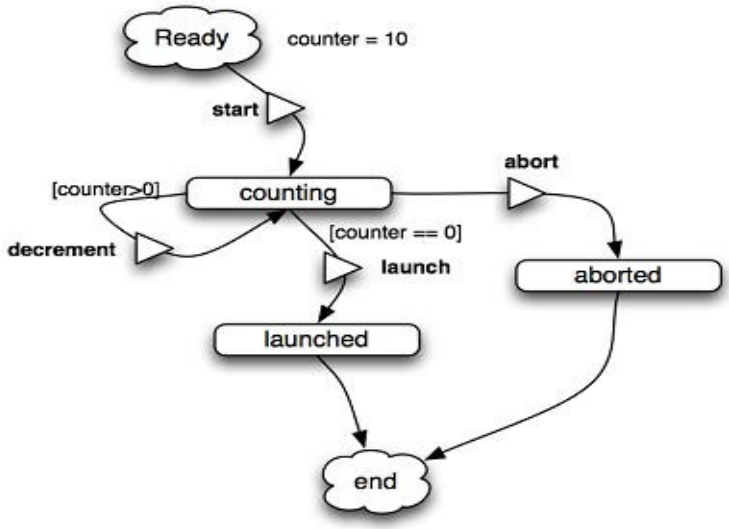

图 4：火箭发射的状态机

系统将会递减计数器（counter），当它到达零的时候，会将 Model 中所有未定的状态设置为规定值，火箭的发射将会进行初始化。

这个应用有一个简单的状态机，见图 4。

其中 decrement 和 launch 都是“自动”的 Action，这意味着我们每次进入（或重新进入）counting 状态时，将会保证进行转换的评估，如果计数器的值大于零的话，decrement Action 将会继续调用，如果值为零的话，将会调用 launchAction。在任何的时间点都可以触发 abort Action，这样的话，控制系统将会转换到 aborted 状态。

在 MVC 中，这种类型的逻辑将会在控制器中实现，并且可能会由 View 中的一个计时器来触发。

这一段至关重要，所以请仔细阅读。我们已经看到，在 TLA+ 中，Action 没有副作用，只是计算结果的状态，Model 处理 Action 的输出并对其自身进行更新。这是与传统状态机语义的基本区别，在传统的状态机中，Action 会指定结果状态，也就是说，结果状态是独立于 Model 的。在 TLA+ 中，所启用的 Action 能够在状态表述（也就是 View）中进行触发，

这些 Action 不会直接与触发状态转换的行为进行关联。换句话说，状态机不应该由连接两个状态的元组（S1, A, S2）来进行指定，传统的状态机是这样做的，它们元组的形式应该是（Sk, Ak1, Ak2,…），这指定了所有启用的 Action，并给定了一个状态 Sk，Action 应用于系统之后，将会计算出结果状态，Model 将会处理更新。

当我们引入“state”对象时，TLA+ 提供了一种更优秀的方式来对系统进行概念化，它将 Action 和 view（仅仅是一种状态的表述）进行了分离。我们样例中的Model如下所示：

```
001 model = {
002         counter:  ,
003         started:  ,
004         aborted:  ,
005         launched:
006 }
```

系统中四个（控制）状态分别对应于 Model 中如下的值：

```
001 ready = {counter: 10, started: false, aborted: false, launched:
      false }
002 counting = {counter: [0..10], started: true, aborted: false,
      launched: false }
003 launched = {counter: 0, started: true, aborted: false, launched:
      true}
004 aborted = {counter: [0..10], started: true, aborted: true, launched:
      false}
```

这个 Model 是由系统的所有属性及其可能的值所指定的，状态则指定了所启用的 Action，它会给定一组值。这种类型的业务逻辑必须要在某个地方进行实现。我们不能指望用户能够知道哪个 Action 是否可行。在这方面，没有其他的方式。不过，这种类型的业务逻辑很难编写、调试和维护，在没有语义对其进行描述时，更是如此，比如在 MVC 中就是这样。

让我们为火箭发射的样例编写一些代码。从 TLA+ 角度来讲，next-action 断言在逻辑上会跟在状态渲染之后。当前状态呈现之后，下一

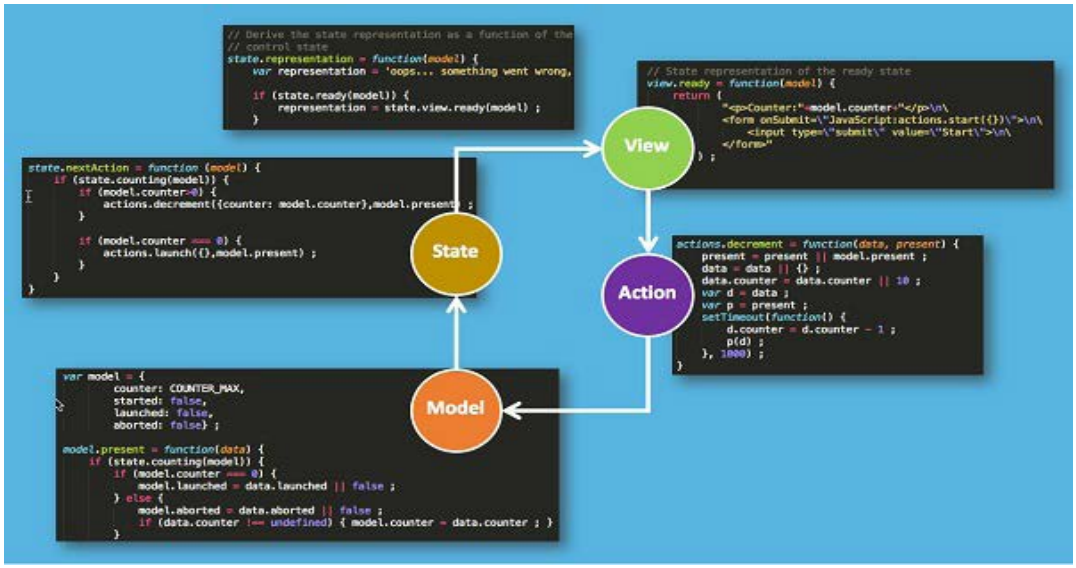

图 5：火箭发射器的实现

步就是执行 next-action 断言，如果存在的话，将会计算并执行下一个 Action，这个 Action 会将其数据交给 Model，Model 将会初始化新状态的表述，以此类推，见图 5。

需要注意的是，在客户端 / 服务器架构下，当自动 Action 触发之后，我们可能需要使用像 WebSocket 这样的协议（或者在 WebSocket 不可用的时候，使用轮询机制）来正确地渲染状态表述。

我曾经使用 Java 和 JavaScript 编写过一个很轻量级的开源库，它使用 TLA+ 特有的语义来构造状态对象，并提供了样例，这些样例使用 WebSocket、轮询和队列实现浏览器 / 服务器交互。在火箭发射器的样例中可以看到，我们并非必须要使用那个库。一旦理解了如何编写，状态实现的编码相对来讲是很容易的。

对于要引入的新模式来说，我相信我们已经具备了所有的元素，这个新模式作为 MVC 的替代者，名为 SAM 模式（状态 - 行为 - 模型，State-Action-Model），它具有反应型和函数式的特性，灵感来源于 React.js 和 TLA+。

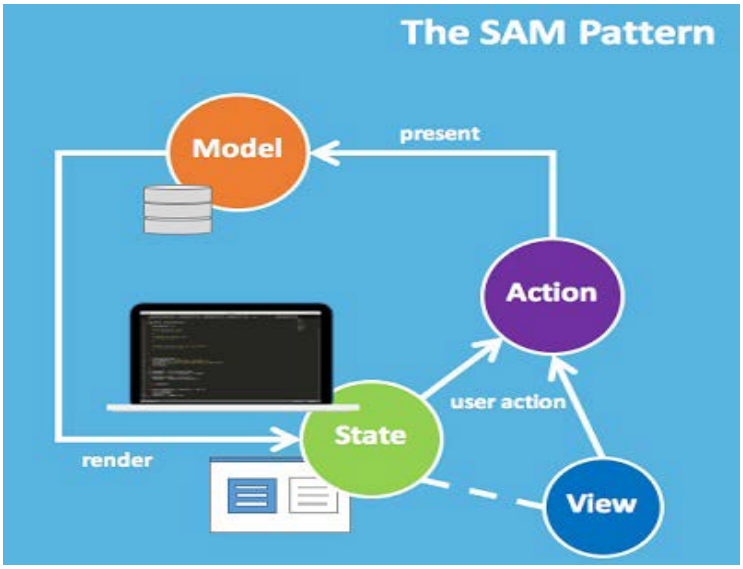

图 6：状态 – 行为 – 模型（SAM）模式

SAM 模式可以通过如下的表达式来进行描述：

V = S(vm(M.present(A(data))), nap(M))

它表明在应用一个 Action A 之后，View V 可以计算得出，Action 会作为 Model 的纯函数。

在 SAM 中，A（Action）、vm（视图 - 模型，view-model）、nap（next-action 断言）以及 S（状态表述）必须都是纯函数。在 SAM 中，我们通常所说的“状态”（系统中属性的值）要完全局限于 Model 之中，改变这些值的逻辑在 Model 本身之外是不可见的。

随便提一下，next-action 断言，即 nap() 是一个回调，它会在状态表述创建完成，并渲染给用户时调用（见图 6）。

模式本身是独立于任何协议的（可以不费什么力气就能在 HTTP 上实现）和客户端 / 服务器拓扑结构的。

SAM 并不意味着我们必须要使用状态机的语义来获取 View 的内容。如果 Action 是由 View 触发的，那 next-action 断言就是一个空函数。不过，这可能是一个很好的实践，它清晰暴露了底层状态机的控制状态，因

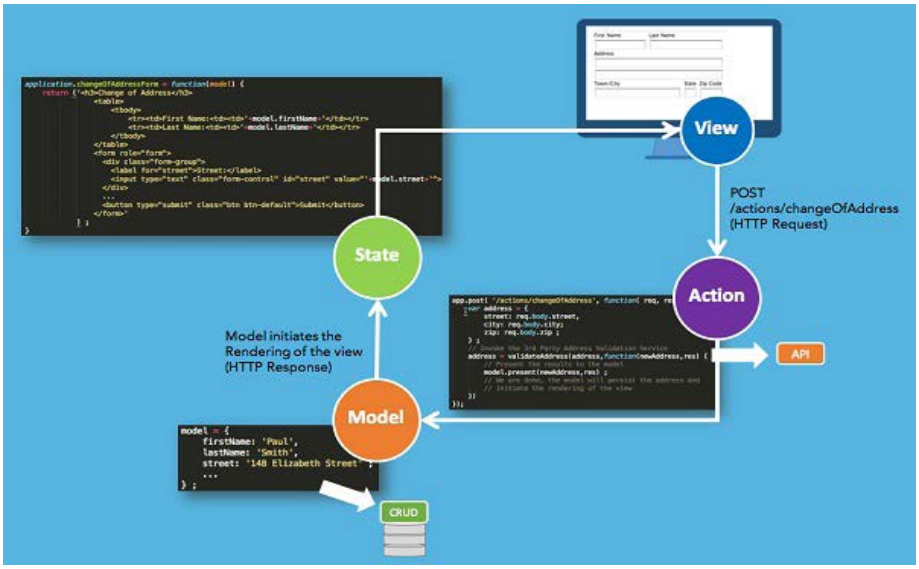

图 7：“修改地址”的实现

为根据（控制）状态的不同，View 看起来可能也是不同的。

另一方面，如果你的状态机涉及到自动化的 Action，那么 Action 和 Model 都不可能做到纯粹的不包含 next-action 断言：有些 Action 将会变得有状态，或者 Model 必须要触发 Action，而这本来并不是它的角色。顺便提一下，也许并不那么直观，状态对象并没有持有任何的“状态”，它同样也是纯函数，它会渲染 View 并计算 next-action 断言，这两者都来源于 Model 的属性值。

这种新模式的好处在于，它清晰地将 CRUD 操作从 Action 中分离了出来。Model 负责它的持久化，将会通过 CRUD 操作来实现，通过 View 是无法进行访问的。尤其是，View 永远不会处于“获取”数据的位置，View 所能做的唯一的事情就是请求系统中当前的状态表述并通过触发 Action 初始化一个反应型流程。

Action 仅仅代表了一种具有权限的通道，以此来建议 Model 该怎样进行变更。它们本身（在 Model 方面）并没有什么副作用。如果必要的话，

Action 会调用第三方的 API（同样，对 Model 没有副作用），比如说，修改地址的 Action 可能会希望调用地址校验服务，并将服务返回的地址提交到 Model 中。

图 7 就是“修改地址”Action 该如何进行实现，它会调用地址校验的 API。

模式中的元素，包括 Action 和 Model，可以进行自由地组合：

函数组合

data’ = A(B(data))

端组合（Peer）（相同的数据集可以提交给两个Model）

M1.present(data’)

M2.present(data’)

父子组合（父Model控制的数据集提交给子Model）

```
M1.present(data’,M2)

function present(data, child) {

            // 执行更新
            …
            // 同步Model

            child.present(c(data))

}
```

发布 / 订阅组合

M1.on(“topic”, present)

M2.on(“topic”, present)

或

M1.on(“data”, present)

M2.on(“data”, present)

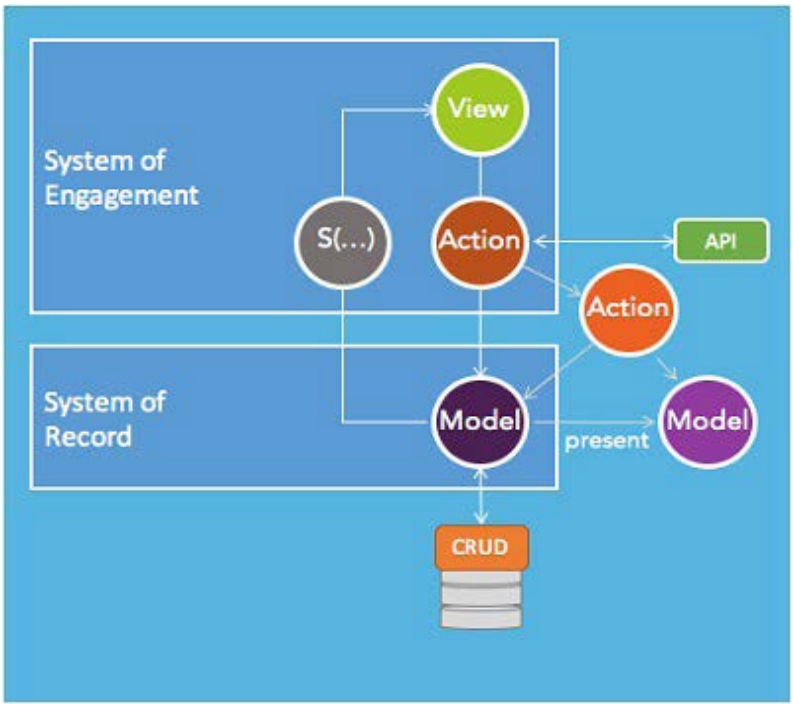

图 8：SAM 组合模型

有些架构师可能会考虑到 System of Record 和 Systems of Engagement，这种模式有助于明确这两层的接口（图 8），Model 会负责与 systems of record 的交互。

整个模式本身也是可以进行组合的，我们可以实现运行在浏览器中的 SAM 实例，使其支持类似于向导（wizard）的行为（如 ToDo 应用），它会与服务器端的 SAM 进行交互，见图 9。

请注意，里层的 SAM 实例是作为状态表述的一部分进行传送的，这个状态表述是由外层的实例所生成的。

会话检查应该在 Action 触发之前进行（图 10）。SAM 能够启用一项很有意思的组合，在将数据提交给 Model 之前，View 可以调用一个第三方的 Action，并且要为其提供一个 token 和指向系统 Action 的回调，这个第三方 Action 会进行授权并校验该调用的合法性。从 CQRS 的角度来讲，这个模式没有对查询（Query）和命令（Command）做特殊的区分，但是底层的实现需要进行这种区分。搜索或查询“Action”只是简单地传递一组

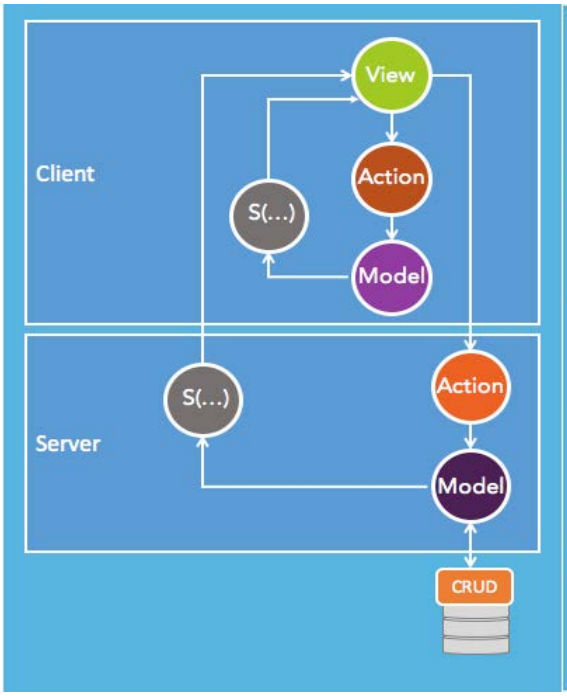

图 9：SAM 实例组合

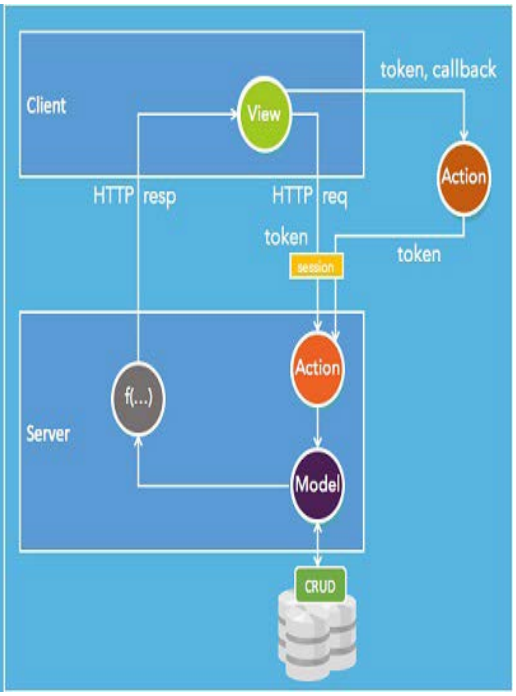

图 10：借助 SAM 实现会话管理

参数到 Model 中。我们可以采用某种约定（如下划线前缀）来区分查询和命令，或者我们可以在 Model 上使用两个不同的 present 方法：

```
001  { _name : '/^[a]$/i' } // 名字以A或a开头
002  { _customerId: '123' } // id=123的customer
```

Model 将会执行必要的操作以匹配查询，更新其内容并触发 View 的渲染。类似的约定可以用于创建、更新或删除 Model 中的元素。在将 Action 的输出传递给 Model 方面，我们可以实现多种方式（数据集、事件、Action……）。每种方式都会有其优势和不足，最终这取决于个人偏好。我更喜欢数据集的方式。

在异常方面，与 React 类似，我们预期 Model 会以属性值的形式保存异常信息（这些属性值可能是由 Action 提交的，也可能是 CRUD 操作返回的）。在渲染状态表述的时候，会用到属性值，以展现异常信息。

在缓存方面，SAM 在状态表述层提供了缓存的选项。直观上来看，缓存这些状态表述函数的结果能够实现更高的命中率，因为我们现在是在组件 / 状态层触发缓存，而不是在 Action/ 响应层。

该模式的反应型和函数式结构使得功能重放（replay）和单元测试变得非常容易。

SAM 模式完全改变了前端架构的范式，因为根据 TLA+ 的基础理念，业务逻辑可以清晰地描述为：

- Action是纯函数
- CRUD操作放在Model中
- 状态控制自动化的Action

作为 API 的设计者，从我的角度来讲，这种模式将 API 设计的责任推到了服务器端，在 View 和 Model 之间保持了最小的契约。

Action 作为纯函数，能够跨 Model 重用，只要某个 Model 能够接受 Action 所对应的输出即可。我们可以期望 Action 库、主题（状态表述）甚至 Model 能够繁荣发展起来，因为它们现在能够独立地进行组合。

借助 SAM 模式，微服务能够非常自然地支撑 Model。像 Hivepod.io 这样的框架能够插入进来，就像它本来就在这层似得。

最为重要的是，这种模式像 React 一样，不需要任何的数据绑定或模板。

随着时间的推移，我希望能够推动浏览器永久添加虚拟 DOM 的特性，新的状态表述能够通过专有 API 直接进行处理。

我发现这个旅程将会带来一定的革新性：在过去的几十年中，面向对象似乎无处不在，但它已经一去不返了。我现在只能按照反应型和函数式来进行思考。我借助 SAM 所构建的东西及其构建速度都是前所未有的。另外，我能够关注于 API 和服务的设计，它们不再遵循由前端决定的模式。

AWS 系列：S3 不仅仅是存储

作者 陈天

【编者按】本文系 InfoQ 中文站向陈天的约稿，这是 AWS 系列文章的第二篇。以后会有更多文章刊出，但并无前后依赖的关系，每篇都自成一体。读者若要跟随文章来学习 AWS，应该至少注册了一个 AWS 账号，事先阅读过当期所介绍服务的简介，并在 AWS management console 中尝试使用过该服务。否则，阅读的效果不会太好。在这篇文章里，介绍了尝试用 S3 创建公司内部的文件服务器，保存员工私人 / 共享文件，并以类似 Dropbox 的方式双向同步。

S3 介绍

S3 是 AWS 最早发布的诸多服务之一，用作可信存储。所谓可信，AWS 给出的概念是：「在指定年度内为对象提供 99.999999999% 的持久性和高达 99.99% 的可用性」，换句话说就是任何存储于 S3 的数据基本不可能丢失，在一个年度内，不超过 1 小时（3153.6s）的宕机时间。除此之外，S3 还提供如下特性：

跨区域复制：只需要简单的配置，存储于 S3 中的数据会自动复制到选定的不同区域中。当你的数据对象的收集分散在不同的区域，而处理集中在某些区域时非常有用。

- 事件通知：当数据对象上传到 Amazon S3 中或从中删除的时候会发送事件通知。事件通知可使用 SQS 或 SNS 进行传送，也可以直接发送到 AWS Lambda 进行处理。比如说，上传到S3的图片的resize。
- 版本控制：数据对象可以启用版本控制，这样你就可以很方便地进行回滚。对于应用开发者来说，这是个特别有用的特性。
- 加密：S3的访问本身是支持 SSL（HTTPS）的，保证传输的安全，对于数据本身，你可以通过Server side encryption（AES256）来加密存储在S3的数据。
- 访问管理：通过 IAM/VPC 可以控制S3的访问粒度，你甚至可以控制一个bucket（S3对数据的管理单元，一个bucket类似于一组数据的根目录）里面的每个folder，甚至每个文件的访问权限。
- 可编程：可以使用 AWS SDK 进行客户端或者服务端的开发。
- 成本监控和控制：S3 有几项管理和控制成本的功能，包括管理成本分配的添加存储桶标签和接收账单警报的 Amazon Cloud Watch 集成。
- 灵活的存储选项：除了 S3 Standard，还有低成本的 Standard - Infrequent Access 选项可用于非频繁访问数据，存储的价格大概是 Standard 的 2/5。至于那些访问不了多少次的冷数据（如1年前的Log），可以存储在Glacier中，价格在 Standard 的 1/4（1T $7/月），缺点是需要几个小时来恢复数据（估计是存放于离线的磁带中）。

基本用法

S3 的用户可以使用 AWS management console 来创建 bucket（类比文件系统的根目录），以及 bucket 内部的目录树，并上传文件，但这不

是使用 S3 的最佳方式。日常的主要操作应该使用 AWS CLI 和 AWS SDK 完成。

AWS CLI

安装 AWS CLI 可以使用 pip / brew 等安装工具，不再详述。AWS CLI 是 AWS 官方提供的 CLI 工具，简单好用，我会另行撰文深度介绍 AWS CLI。AWS CLI 目前不支持命令和参数的自动补全，从 AWS re:invent 2015 透露出来的信息，其团队在做一些自动补全的尝试，未来会变得更加人性化。如果你想现在就用得更舒服一些，可以使用 sAws。

使用 AWS CLI 操作 S3 非常简单，创建 / 删除 bucket 可以使用 aws s3api：

```
$ aws s3api create-bucket --bucket <name>
$ aws s3api delete-bucket --bucket <name>
```

如果要像一般的文件系统一样操作 S3，可以使用 aws s3 命令：

```
$ aws s3 ls
$ aws s3 cp
$ aws s3 rm
```

此外，aws s3 还提供了 sync，方便本地文件和 S3 上的文件互相 sync，比如我本地用 pandoc 编译出了 markdown 撰写的 reveal.js 的 slides，可以这样同步到 S3：

```
$ aws s3 sync ./output s3://eng-assets/slides
```

AWS SDK

AWS SDK 提供了对几乎所有主流语言的支持，在程序里使用 S3，一般的流程是：

- 创建AWS connection（这一步需要用到你的 access key）。
- 使用connection 创建 S3 对象。

- 使用S3 API进行各种API操作，比如创建bucket，上传文件等。

这里列一个 JavaScript 的例子：

```
001 const aws = require('aws-sdk');
002 const Promise = require("bluebird");
003
004 const s3 = Promise.promisifyAll(new aws.S3());
005 s3.createBucketAsync({Bucket: 'test-myBucket'}).then(function() {
006   var params = {Bucket: 'test-myBucket', Key: 'myKey', Body:
      'Hello!'};
007   s3.putObjectAsync(params).then(function(data) {
008     console.log('successfully uploaded data');
009   }).error(function(err) {
010     console.log(err);
011   })
012 });
```

使用 S3 的典型场景

S3 的一些典型使用场景如下：

- 存储用户上传的文件，如头像，照片，视频等静态内容。
- 当作一个的key value store，承担简单的数据库服务功能。
- 数据备份。
- 静态网站的托管：你可以对一个bucket使能Web Hosting。

我们简单介绍一下 S3 实现静态网站托管，然后以一个例子讲述如何使用 S3 实现一个能最大程度保证数据安全同时又价格低廉的团队内部的文件服务器。

使用 S3 实现安全的静态网站托管

经常使用 GitHub 的朋友对 GitHub pages 服务一定不会陌生，你只要把各种静态网站生成工具的生成的目标放入 gh-pages 的 branch，GitHub pages 就会帮你做静态网站的托管。得益于如今越来越强大的 JavaScript 和各种 API，静态网站其实早已脱离了展示 HTML 的基本范畴。

GitHub pages 有一个缺点就是，只要你使用，它就是开放的，无法

变成一个私有网站，存放公司内部的私密文件。公司内部的一些私有内容，比如：

- 使用 reveal.js 生成的 slides。
- 使用 new relic 生成的各种嵌入式报表和图表。
- 使用 JavaScript + AWS SDK 做的各种内部工具（由于 AWS SDK 提供了 JavaScript SDK，所以你可以用静态网站的方式访问数据库等服务，实现 server less 的效果）。

你无论如何都不会想将其暴露给外界。这个时候，GitHub pages 就不适用，我们可以使用 S3 Web Hosting + IAM policy 来完成。

使能 S3 Web Hosting 是件很简单的事情，只需在 AWS console 中，为对应的 bucket 打开这个选项即可，然后添加如下 IAM policy：

```
001 {
002   “Version”: “2012-10-17”,
003   “Statement”: [
004     {
005       “Sid”: “PublicReadGetObject”,
006       “Effect”: “Allow”,
007       “Principal”: “*”,
008       “Action”: “s3:GetObject”,
009       “Resource”: “arn:AWS:s3:::team-assets/*”
010     }
011   ]
012 }
```

S3 Web Hosting 会告诉你一个用于访问的域名，你也可将你自己的私有域名指定一个 CNAME 指向该域。这样配置下来，只要域名和要访问的文件夹没有暴露，文件内容就是安全的。适用于安全等级不高的内容。

如果需要更高的安全级别，可以配合 VPC + IAM policy。一般而言，使用 VPC 的用户，都会将 VPC 设置成私有网络（比如 10.0.0.x 的网络），然后在网络边界配置一台 VPN 服务器，用于内外网的交互。任何用户要访问内网，必须先接入 VPN。我们可以设置用于 Web Hosting 的 S3 的 bucket 的 IAM 仅允许 VPN 服务器的 IP 访问，如下：

```
{
  “Version”: “2012-10-17”,
  “Statement”: [
    {
      “Sid”: “PublicReadGetObject”,
      “Effect”: “Allow”,
      “Principal”: “*”,
      “Action”: “s3:GetObject”,
      “Resource”: “arn:AWS:s3:::team-assets/*”,
      “Condition” :  {
        “IpAddress” : {
          “aws:SourceIp” : [“5.5.5.5/32”]
        }
      }
    }
  ]
}
```

那么，只用当用户接入 VPN 之后，才能访问 Web Hosting 的域名下的文件，进一步提高了安全性。当然，由于不是在路由层面控制访问，所以没办法防止 ip spoofing，还是有一些潜在风险的，不过风险不大。（攻击者需要知道要访问的文件所在的域名和路径，并且知道仅允许哪个源 IP 访问，进而进行 IP spoofing，而公网上 IP spoofing 的难度很大，基本上所有的路由器都会做 reverse route check。）

使用 S3 实现文件服务器

很多公司都会为员工提供私人和共享的文件存储。比如作为一个用户，我可以把我的私人文件存放在：fileserver://home/tyrchen/* 下，把一些共享文件存放在 fileserver://public/tyrchen/* 下。为了能够安全的存储这些文件，公司的 IT 部门一般会使用昂贵的 SAN（Storage Area Network）来保证一定程度的 SLA（Service Level Agreement），同时，还要做各种各样的备份（和恢复）。如果我们使用 S3 来实现类似的文件服务器，其代价和未来的维护成本会小得多。此外，我们还可以做一些额外的开发，使得文件服务器的使用体验类似于 Dropbox。

大致的想法是这样的：

新员工入职后会为其在 S3 上建立 home folder，用来保存重要的私

人文件和共享文件。

员工电脑的本地文件中会有一个目录 corp-fs-box，里面包含三个子目录：

- private：存放任意文件，私有，会自动sync到私人目录，别人无法访问。
- photos：存放各种媒体文件，公开，会自动sync到共享目录，并生成合适的尺寸放在供Web访问的S3 bucket中。

员工只要在本地目录中存放文件，就会按照上述规则自动同步，类似Dropbox。

解决思路：

创建两个 S3 bucket：corp-fs-team 和 corp-fs-web。corp-fs-web 打开 Web Hosting 功能。

使用 IAM policy 来设置 home folder 的权限。

使用 aws s3sync 来同步文件夹：

- 对本地 corp-fs-box/private 里的文件，同步到 S3://corp-fs-team/home/{AWS:username}/ 中。这个目录只有当前用户可以访问，其他用户不能访问。
- 对本地 corp-fs-box/pub/photos 里的文件，同步到 S3:web//corp-fs-team/pub/photos 中。这个目录任何用户都可以访问并修改。

S3 配置 Events，使得对于 S3://corp-fs-team/pub/photos/{AWS:username}/ 的任何更新行为（添加 / 删除）都会触发 lambda 函数。

lambda 函数扫描上传的文件，如果是 *.jpg 或者 *.mp4 / *.mov，则将其进行 resize / transcoding 等处理，并将编译的结果放在 S3://corp-fs-web/pub/photos/{AWS:username}/* 下，供内网的用户浏览。

涉及的 IAM policy 如下：

```
{
  “Version”: “2012-10-17”,
  “Statement”: [{
      “Sid”: “AllowGroupToSeeBucketList”,
      “Action”: [“s3:ListAllMyBuckets”, “s3:GetBucketLocation”],
      “Effect”: “Allow”,
      “Resource”: [“arn:AWS:s3:::*”]
    }, {
      “Sid”: “AllowRootLevelList”,
      “Action”: [“s3:ListBucket”],
      “Effect”: “Allow”,
      “Resource”: [“arn:AWS:s3:::corp-fs-team”],
      “Condition”: {
        “StringEquals”: {
          “s3:prefix”: [“”, “home/”],
          “s3:delimiter”: [“/”]
        }
      }
    }, {
      “Sid”: “AllowListForUserPrefix”,
      “Action”: [“s3:ListBucket”],
      “Effect”: “Allow”,
      “Resource”: [“arn:AWS:s3:::corp-fs-team”],
      “Condition”: {
        “StringLike”: {
          “s3:prefix”: [“home/${AWS:username}/*”]
        }
      }
    }, {
      “Sid”: “AllowUserFullAccessToUserPrefix”,
      “Action”: [“s3:*”],
      “Effect”: “Allow”,
      “Resource”: [
        “arn:AWS:s3:::corp-fs-team/home/${AWS:username}”,
        “arn:AWS:s3:::corp-fs-team/home/${AWS:username}/*”
      ]
    }
  ]
}
```

以及访问 pub 目录的 IAM policy（见下页）。

具体的 lambda 函数不在本文讨论的范围之内。

除此之外，我们还需要一个类似于 Dropbox 的客户端软件来监控本地目录（S3 目录）的更改，以便在合适的时候进行同步。思路如下：

客户端软件做成一个开机启动的 daemon。

随时监控本地目录 corp-fs-box/* 和 S3 bucket 的修改，并按上述规则同步。

```
{
  “Version”: “2012-10-17”,
  “Statement”: [{
      “Sid”: “AllowPublicLevelList”,
      “Action”: [“s3:ListBucket”],
      “Effect”: “Allow”,
      “Resource”: [“arn:AWS:s3:::corp-fs-team-bucket”],
      “Condition”: {
        “StringLike”: {
          “s3:prefix”: [“pub/*”]
        }
      }
    }, {
      “Sid”: “AllowUserFullAccessToPublicPrefix”,
      “Action”: [“s3:*”],
      “Effect”: “Allow”,
      “Resource”: [
        “arn:AWS:s3:::corp-fs-team-bucket/pub”,
        “arn:AWS:s3:::corp-fs-team-bucket/pub/*”
      ]
    }
  ]
}
```

由于涉及的目录都是个人目录，不太会产生冲突（除非同一用户在多个 device 下载没有 sync 的前提下修改同一文件。所以在这里，为简单起见，我们可以不涉及到 diff / merge，简单遵循 last writer wins 进行处理就可以了。另外 S3 自带 versioning，也可以使能这一功能，保存历史版本，在冲突发生的时候，让用户选择。

小结

S3 是一个非常强大的文件服务，如果使用得当，可以带来非常大的收益，建议大家多多深入研究。AWS 的很多服务，如 Elastic Beanstalk，Elastic Transcoder，CloudFormation 实际上都在使用 S3 作为服务的关键一环。

第2季

ChinaTech Day

中国技术开放日·美国站

强势回归！

2016年11月5-14日 美国硅谷

限额预约报名中

中国技术开放日

探索、发现、学习

与技术大咖同行，探访顶尖科技公司，参与全球顶尖技术大会，展示中国技术力量，探索创新之源！

Brought By | Geekbang. 极客邦科技

2015年网站参考 | http://www.chinatechday.com/2015us/

架构师 月刊 2016年5月

本期主要内容：如何在云平台构建大规模分布式系统，携程移动App架构优化之旅，10亿红包从天而降，揭秘微信摇一摇背后的技术细节，云计算时代来了，没有狂欢盛宴只有整个IT业的呜咽

云生态专刊 06

《云生态专刊》是InfoQ为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。

顶尖技术团队访谈录 第五季

本次的《中国顶尖技术团队访谈录》·第五季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。

架构师特刊 推荐系统

推荐系统在各种系统中广泛使用，推荐算法则是其中最核心的技术点，InfoQ策划了系列文章来为读者深入介绍。本书是这个系列文章合集的理论部分。