促进软件开发领域知识与创新的传播

架构师 ARCHITECT

推荐文章 | Article

Docker背后的容器集群管理

微服务架构综述

专题 | Topic

多范式编程语言 - 以Swift为例

MySQL Cluster支持200M的QPS

观点 | Opinion

创业公司的产品开发与团队管理

InfoQueue

2015年08月

Docker背后的容器集群管理
从Borg到Kubernetes

2015年4月，传闻许久的Borg论文总算出现在了Google Research的页面上。虽然传言Borg作为G家的“老”项目一直是槽点满满，而且本身的知名度和影响力也应该比不上当年的“三大论文”，但是同很多好奇的小伙伴一样，笔者还是饶有兴趣地把这篇“非典型”论文拜读了一番。

微服务架构综述

微服务架构（Microservice Architect）是一种架构模式，它提倡将单块架构的应用划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。

多范式编程语言
以Swift为例

Swift即支持面向对象编程范式，也支持函数式编程范式，同时还支持泛型编程。Swift支持多种编程范式是由它的目标决定的。

Oracle专家谈
MySQL Cluster如何支持200M的QPS

Andrew Morgan是Oracle MySQL首席产品经理。近日，他撰文介绍了MySQL Cluster如何支持200M的QPS。

创业公司的产品开发与团队管理

创业公司由于管理层次简单，很少受到官僚作风的困扰，但也不能想当然地认为万事大吉，高枕无忧了。其实创业公司也有创业公司的局限性，这些局限性常常被忽略，从而影响到产品的开发。

架构师 8月刊

本期主编　郭蕾

流程编辑　丁晓昀

发行人　霍泰稳

联系我们

提供反馈　feedback@cn.infoq.com

商务合作　sales@cn.infoq.com

内容合作　editors@cn.infoq.com

马晓宇 环信CTO，18年的老程序员,先后从事过IC设计软件，短信网关、电信网管、中间件、手机操作系统和手机App的研发。

卷首语 扁平的组织结构

技术团队的管理目标是打造高效的自组织团队，也就是老子提出的“无为而治”。做为一个从业 20 年的老程序员，从自身出发，我深切知道程序员喜欢什么样的管理方式，喜欢怎样的团队氛围。在环信，我们希望顺其自然，发挥工程师的热情和创造力，每个成员都能做到自我实现，最终整个团队被业界认同。

扁平的组织结构

我们研发团队有 50 多人，没有一个专门的管理岗位。作为 CTO，我的大多数时间在技术上面，团队负责人会承担一部分管理工作，但更多是靠工程师的自我驱动，自我管理。

结果导向的日常管理

Google 只招聘最聪明的人，这些人能很好的管理个人时间。做为创业公司，我们招聘最努力的人，因为这样的工程师很珍惜时间和成长的机会。团队里有些人每天下午才出现，有些同事在备战马拉松比赛，有酷爱高山滑雪的，也有业余足球运动员。努力工作，精彩生活，结果导向是我们日常管理的原则。

严格的个人成长

Work for fun, constantly improve, and big rewards ahead. 在团队的宽松氛围中，其实我们对工程师的个人成长要求很高：对年轻工程师我们破格使用，希望尽快成长；对高级工程师， 要求技术深度，成为某个方面的专家；对技术负责人，要求更多业界交流，才能打造技术领先产品，同时建立自己的社区影响力。

美好的远程办公

年轻人希望到世界各地体验生活；有小孩的技术中坚天天忧虑北京的雾霾想着移民；40 多岁的同事希望每年回老家工作一段时间陪陪父母。面对这些美好的期待，我们在不断尝试协同工具和管理方法来提高团队内部配合、团队间协作，及团队和客户的沟通。希望一两年内能成为一支高效远程开发团队。到那时，团队成员每年回来参加 InfoQ 大会学习最新技术，也做为每年的团队聚会。

无为而无所不为，希望大家一起探索，把越来越多的团队，打造成工程师的“乐土”。

Docker背后的容器集群管理 从Borg到Kubernetes

作者 张磊

2015年4月，传闻许久的Borg论文总算出现在了Google Research的页面上。虽然传言Borg作为G家的“老”项目一直是槽点满满，而且本身的知名度和影响力也应该比不上当年的“三大论文”，但是同很多好奇的小伙伴一样，笔者还是饶有兴趣地把这篇“非典型”论文拜读了一番。

1. Borg在讲什么故事

其实，如果这篇论文发表在两三年前，Borg 的关注度恐怕真没有今天这么高。作为一篇本质上是关于数据中心利用率的半工程、半研究性的成果，这个领域的关注人群来自大厂的运维部以及系统部的技术同僚可能要占很大的比例。而对于绝大多数研究人员，普通开发者，甚至包括平台开发者而言，Borg 论文本身的吸引力应该说都是比较有限的。

不过，一旦我们把 Borg 放到当前这个时间点上来重新审视，这篇本该平淡的论文就拥有了众多深层意义。当然，这一切绝非偶然，从 2013 年末以 Docker 为代表的容器技术的迅速兴起，2014 年 Google 容器管理平台 Kubernetes 和 Container Engine 的强势扩张，再到如今由 Mesos 一手打造的 DCOS（数据中心操作系统）概念的炙手可热。容器技术令人咋舌的进化速度很快就将一个曾经并不需要被大多数开发人员关注的问题摆到了台面：

我们应该如何高效地抽象和管理一个颇具规模的服务集群？

这正是 Borg 全力阐述的核心问题。

说得更确切一点，当我们逐步接纳了以容器为单位部署和运行应用之后，运维人员终于可以从无休止的包管理，莫名其妙的环境差异，繁杂重复的批处理和任务作业的中稍微回过一点神来，开始重新审视自己手中的物理资源的组织和调度方式：即我们能不能将容器看作传统操作系统的进程，把所有的服务器集群抽象成为统一的 CPU、内存、磁盘和网络资源，然后按需分配给任务使用呢？

所以，作为《Docker 背后的技术解析》系列文章的特别篇，笔者将和读者一起从 Borg 出发，结合它的同源项目 Kubernetes 中尝试探索一下这个问题的答案。

2. Borg的核心概念

同大多数 PaaS、云平台类项目宣称的口号一样，Borg 最基本的出发点还是“希望能让开发者最大可能地把精力集中在业务开发上”，而不需要关心这些代码制品的部署细节。不过，另一方面，Borg 非常强调如何对一个大规模的服务器集群做出更合理的抽象，使得开发者可以像对待一台 PC 一样方便地管理自己的所有任务。这与 Mesos 现在主推的观点是一致的，同时也是 Borg 同 PaaS 类项目比如 Flynn、Deis、Cloud Foundry 等区别开来的一个主要特征：即 Borg，以及 Kubernetes 和 Mesos 等，**都不是一个面向应用的产物**。

什么叫面向应用？

就是以应用为中心。系统原生为用户提交的制品提供一系列的上传、构建、打包、运行、绑定访问域名等接管运维过程的功能。这类系统一般会区分”应用“和”服务“，并且以平台自己定义的方式为”应用“（比如 Java 程序）提供具体的”服务“（比如 MySQL 服务）。面向应用是 PaaS 的一个很重要的特点。

另一方面，Borg 强调的是**规模**二字。文章通篇多次强调了 Google 内部跑在 Borg 上的作业数量、以及被 Borg 托管的机器数量之庞大。比如我们传统认知上的“生产级别集群”在文章中基本上属于 Tiny 的范畴，而 Borg 随便一个 Medium 的计算单元拿出来都是一家中大型企业数据中心的规模（10K 个机器）。这也应证了淘宝毕玄老大曾经说过的：“规模绝对是推动技术发展的最关键因素”。

Borg 里服务器的划分如下：Site = 一组数据中心（Cluster），Cluster = 一组计算单元（Cell），Cell = 一组机器。其中计算单元（Cell）是最常用的集群类别。

2.1 Job，Task

既然 Borg 不关心“应用”和“服务”的区别，也不以应用为中心，那么它需要接管和运行的作业是什么？

是 **Job**。

Borg 文章里对 Job 的定义很简单，就是多个任务（Task）的集合，而所谓 Task 就是跑在 Linux 容器里的应用进程了。这样看起来 Job 是不是就等同于 Kubernetes 里的 Pod（容器组）呢？

其实不然。Job 映射到 Kubernetes 中的话，其实等同于用户提交的“应用”，至于这个应用运行了几个副本 Pod，每个 Pod 里又运行着哪些容器，用户并不需要关心。用户只知道，我们访问这个服务，应该返回某个结果，就够了。

举个例子，因为高可用等原因，用户常常会在 Kubernetes 里创建并启动若干个一模一样的 Pod（这个功能是通过 Kubernetes 的 Replication Controller 实现的）。这些一模一样的 Pod“副本”的各项配置和容器内容等都完全相同，他们抽象成一个逻辑上的概念就是 Job。

由于 Job 是一个逻辑上的概念，Borg 实际上负责管理和调度的实体就是 Task。用户的 submit、kill、update 操作能够触发 Task 状态机从 Pending 到 Running 再到 Dead 的的转移，这一点论文里有详细的图解。

值得一提的是，作者还强调了Task是通过先SIGTERM，一定时间后后再SIGKILL的方式来被杀死的，所以Task在被杀死前有一定时间来进行“清理，保存状态，结束正在处理的请求并且拒绝新的请求”的工作。

2.2 Alloc

Borg中，真正与Pod对应的概念是Alloc。

Alloc的主要功能，就是在一台机器上“划”一块资源出来，然后一组Task就可以运行在这部分资源上。这样，“超亲密”关系的Task就可以被分在同一个Alloc里，比如一个“Tomcat应用”和它的“logstash服务”。

Kubernetes中Pod的设计与Alloc如出一辙：属于同一个Pod的Docker容器共享Network Namepace和volume，这些容器使用localhost来进行通信，可以共享文件，任何时候都会被当作一个整体来进行调度。

所以，Alloc和Pod的设计其实都是在遵循“一个容器一个进程”的模型。经常有人问，我该如何在Docker容器里跑多个进程？其实，这种需求最好是通过类似Pod这种方法来解决：每个进程都跑在一个单独的容器里，然后这些容器又同属于一个Pod，共享网络和指定的volume。这样既能满足这些进程之间的紧密协作（比如通过localhost互相访问，直接进行文件交换），又能保证每个进程不会挤占其他进程的资源，它们还能作为一个整体进行管理和调度。如果没有Kubernetes的话，Pod可以使用“Docker in Docker”的办法来模拟，即使用一个Docker容器作为Pod，真正需要运行的进程作为Docker容器嵌套运行在这个Pod容器中，这样它们之间互不干涉，又能作为整体进调度。

另外，Kubernetes实际上没有Job这个说法，而是直接以Pod和Task来抽象用户的任务，然后使用相同的**Label**来标记同质的Pod副本。这很大程度是因为在Borg中Job Task Alloc的做法里，会出现“交叉”的情况，比如属于不同Job的Task可能会因为“超亲密”关系被划分到同一个Alloc中，尽管此时Job只是个逻辑概念，这还是会给系统的管理带来很多不方便。

2.3 Job的分类

Borg中的Job按照其运行特性划分为两类：LRS（Long Running Service）和batch jobs。

上述两种划分在传统的PaaS中也很常见。LRS类服务就像一个“死循环”，比如一个Web服务。它往往需要服务于用户或者其它组件，故对延时敏感。当然论文里Google举的LRS例子就要高大上不少，比如Gmail、Google Docs。

而batch jobs类任务最典型的就是Map-Reduce的job，或者其它类似的计算任务。它们的执行往往需要持续一段时间，但是最终都会停止，用户需要搜集并汇总这些job计算得到的结果或者是job出错的原因。所以Borg在Google内部起到了YARN和Mesos的角色，很多项目通过在Borg之上构建framework来提交并执行任务。Borg里面还指出，batch job对服务器瞬时的性能波动是不敏感的，因为它不会像LRS一样需要立刻响应用户的请求，这一点可以理解。

比较有意思的是，Borg中大多数LRS都会被赋予高优先级并划分为生产环境级别的任务（prod），而batch job则会被赋予低优先级（non-prod）。在实际环境中，prod任务会被分配和占用大部分的CPU和内存资源。正是由于有了这样的划分，Borg的“资源抢占”模型才得以实现，即prod任务可以占用non-prod任务的资源，这一点我们后面会专门说明。

对比Kubernetes，我们可以发现在LRS上定义上是与Borg类似的，但是目前Kubernetes却不能支持batch job：因为对应的Job Controller还没有实现。这意味着当前Kubernetes上一个容器中的任务执行完成退出后，会被Replication Controller无条件重启。Kubernetes尚不能按照用户的需求去搜集和汇总这些任务执行的结果。

2.4 优先级和配额

前面已经提到了 Borg 任务优先级的存在，这里详细介绍一下优先级的划分。

Borg 中把优先级分类为监控级、生产级、批任务级、尽力级（也叫测试级）。其中监控级和生产级的任务就是前面所说的 prod 任务。为了避免在抢占资源的过程中出现级联的情况触发连锁反应（A抢占B,B抢占C,C再抢占D），Borg 规定 prod 任务不能互相抢占。

如果说优先级决定了当前集群里的任务的重要性，配额则决定了任务是否被允许运行在这个集群上。

尽管我们都知道，对于容器来说，CGroup 中的配额只是一个限制而并非真正割据的资源量，但是我们必须为集群设定一个标准来保证提交来任务不会向集群索要过分多的资源。Borg 中配额的描述方法是：该用户的任务在一段时间内在某一个计算单元上允许请求的最大资源量。需要再次重申，配额一定是任务提交时就需要验证的，它是任务合法性的一部分。

既然是配额，就存在超卖的情况。在 Borg 中，允许被超卖的是 non-prod 的任务，即它们在某个计算单元上请求的资源可能超出了允许的额度，但是在允许超卖的情况下它们仍然有可能被系统接受（虽然很可能由于资源不足而暂时进入 Pending 状态）。而优先级最高的任务则被 Borg 认为是享有无限配额的。

与 Kubernetes 类似的是，Borg 的配额也是管理员静态分配的。Kubernetes 通过用户空间（namespace）来实现了一个简单的多租户模型，然后为每一个用户空间指定一定的配额，比如：

```
001 apiVersion: v1beta3
002 kind: ResourceQuota
003 metadata:
004   name: quota
005 spec:
006   hard:
007     cpu: “20”
008     memory: 10Gi
009     pods: “10”
010     replicationcontroll
011 ers: “20”
012     resourcequotas: “1”
013     services: “5”
```

到这里，我们有必要多说一句。像 Borg、Kubernetes 以及 Mesos 这类项目，它们把系统中所有需要对象都抽象成了一种“资源”保存在各自的分布式键值存储中，而管理员则使用如上所示的“资源描述文件”来进行这些对象的创建和更新。这样，整个系统的运行都是围绕着“资源”的增删改查来完成的，各组件的主循环遵循着“检查对象”、“对象变化”、“触发事件”、“处理事件”这样的周期来完成用户的请求。这样的系统有着一个明显的特点就是它们一般都没有引入一个消息系统来进行事件流的协作，而是使用“ectd”或者“Zookeeper”作为事件系统的核心部分。

2.5 名字服务和监控

与 Mesos 等不同，Borg 中使用的是自家的一致性存储项目 Chubby 来作为分布式协调组件。这其中存储的一个重要内容就是为每一个 Task 保存了一个 DNS 名字，这样当 Task 的信息发生变化时，变更能够通过 Chubby 及时更新到 Task 的负载均衡器。这同 Kubernetes 通过 Watch 监视 etcd 中 Pod 的信息变化来更新服务代理的原理是一样的，但是由于使用了名为“Service”的服务代理机制（Service 可以理解为能够自动更新的负载均衡组件），Kubernetes 中默认并没有内置名字服务来进行容器间通信（但是提供了插件式的 DNS 服务供管理员选用）。

在监控方面，Borg 中的所有任务都设置了一个健康检查 URL，一旦 Borg 定期访问某个 Task 的 URL 时发现返回不符合预期，这个 Task 就会被重启。这个过程同 Kubernetes 在 Pod 中设置 health_check 是一样的，比如下面这个例子。

这种做法的一个小缺点是 Task 中服务的开发者需要自己定义好这些 /healthzURL 和对应的响应逻辑。当然，另一种做法是可以在容器里内置一些“探针”来完成很多健康检查工作而做到对用户的开发过程透明。

```
001 apiVersion: v1beta3
002 kind: Pod
003 metadata:
004   name: pod-with-
    healthcheck
005 spec:
006   containers:
007     - name: nginx
008       image: nginx
009       # defines the
    health checking
010       livenessProbe:
011         # an http probe
012         httpGet:
013           path: /_
    status/healthz
014           port: 80
015         # length of
    time to wait for a
    pod to initialize
016         # after pod
    startup, before
    applying health
    checking
017
    initialDelaySeconds:
    30
018         timeoutSeconds:
    1
019       ports:
020         -
    containerPort: 80
```

这种做法的一个小缺点是 Task 中服务的开发者需要自己定义好这些 /healthzURL 和对应的响应逻辑。当然，另一种做法是可以在容器里内置一些“探针”来完成很多健康检查工作而做到对用户的开发过程透明。

除了健康检查，Borg 对日志的处理也很值得借鉴。Borg 中 Task 的日志会在 Task 退出后保留一段时间，方便用户进行调试。相比之下目前大多数 PaaS 或者类似项目的容器退出后日志都会立即被删除（除非用户专门做了日志存储服务）。

最后，Borg 轻描淡写地带过了保存 event 做审计的功能。这其实与 Kubernetes 的 event 功能也很类似，比如 Kube 的一条 event 的格式类似于：**发生时间 结束时间 重复次数 资源名称 资源类型 子事件 发起原因 发起者 事件日志**。

3. Borg的架构与设计

Borg 的架构与 Kubernetes 的相似度很高，在每一个 Cell（工作单元）里，运行着少量 Master 节点和大量 Worker 节点。其中，Borgmaster 负责响应用户请求以及所有资源对象的调度管理；而每个工作节点上运行着一个称为 Borglet 的 Agent，用来处理来自 Master 的指令。这样的设计与 Kubernetes 是一致的，Kubernetes 这两种节点上的工作进程分别是：

```
001 Master:
002 apiserver, controller-
      manager, scheduler
003 Minion:
004 kube-proxy, kubelet
```

虽然我们不清楚 Borg 运行着的工作进程有哪些，但单从功能描述里面我们不难推测到至少在 Master 节点上两者的工作进程应该是类似的。不过，如果深入到论文中的细节的话，我们会发现 Borg 在 Master 节点上的工作要比 Kubernetes 完善很多。

3.1 Borgmaster

首先，Borgmaster 由一个独立的 scheduler 和主 Borgmaster 进程组成。其中，主进程负责响应来自客户端的 RPC 请求，并且将这些请求分为“变更类”和“只读”类。

在这一点上 Kubernetes 的 apiserver 处理方法类似，kuber 的 API 服务被分为“读写”（GET，POST，PUT，DELETE）和“只读”（GET）两种，分别由 6443 和 7080 两个不同的端口负责响应，并且要求“读写”端口 6443 只能以 HTTPS 方式进行访问。同样，Kubernetes 的 scheduler 也是一个单独的进程。

但是，相比 Kubernetes 的单点 Master，Borgmaster 是一个由五个副本组成的集群。每一个副本都在内存中都保存了整个 Cell 的工作状态，并且使用基于 Paxos 的 Chubby 项目来保存这些信息和保证信息的一致性。Borgmaster 中的 Leader 是也是集群创建的时候由 Paxos 选举出来的，一旦这个 Leader 失败，Chubby 将开始新一轮的选举。论文中指出，这个重选举到恢复正常的过程一般耗时 10s，但是在比较大的 Cell 里的集群会由于数据量庞大而延长到一分钟。

更有意思的是，Borgmaster 还将某一时刻的状态通过定时做快照的方式保存成了 checkpoint 文件，以便管理员回滚 Borgmaster 的状态，从而进行调试或者其他的分析工作。基于上述机制，Borg 还设计了一个称为 Fauxmaster 的组件来加载 checkpoint 文件，从而直接进入某时刻 Borgmaster 的历史状态。再加上 Fauxmaster 本身为 kubelet 的接口实现了“桩”，

所以管理员就可以向这个Fauxmaster发送请求来模拟该历史状态数据下Borgmaster的工作情况，重现当时线上的系统状况。这个对于系统调试来说真的是非常有用。此外，上述Fauxmaster还可以用来做容量规划，测试Borg系统本身的变更等等。这个Fauxmaster也是论文中第一处另我们眼前一亮的地方。

上述些特性使得Borg在Master节点的企业级特性上明显比Kubernetes要成熟得多。当然，值得期待的是Kube的高可用版本的Master也已经进入了最后阶段，应该很快就能发布了。

3.2 Borg的调度机制

用户给Borg新提交的任务会被保存在基于Paxos的一致性存储中并加入到等待队列。Borg的scheduler会异步地扫描这个队列中的任务，并检查当前正在被扫描的这个任务是否可以运行在某台机器上。上述扫描的顺序按照任务优先级从高到低来Round-Robin，这样能够保证高优先级任务的可满足性，避免“线头阻塞”的发生（某个任务一直不能完成调度导致它后面的所有任务都必须进行等待）。每扫描到一个任务，Borg即使用调度算法来考察当前Cell中的所有机器，最终选择一个合适的节点来运行这个任务。

此算法分两阶段：

第一，可行性检查。这个检查每个机器是所有符合任务资源需求和其它约束（比如指定的磁盘类型），所以得到的结果一般是个机器列表。需要注意的是在可行性检查中，一台机器“资源是否够用”会考虑到抢占的情况，这一点我们后面会详细介绍。

第二，打分。这个过程从上述可行的机器列表中通过打分选择出分数最高的一个。

这里重点看**打分过程**。Borg设计的打分标准有如下几种：

1. 尽量避免发生低优先级任务的资源被抢占；如果避免不了，则让被抢占的任务数量最少、优先级最低；
2. 挑选已经安装了任务运行所需依赖的机器；
3. 使任务尽量分布在不同的高可用域当中；
4. 混合部署高优先级和低优先级任务，这样在流量峰值突然出现后，高优先级可以抢占低优先级的资源（这一点很有意思）。

Borg其实曾经使用过E-PVM模型（简单的说就是把所有打分规则按照一定算法综合成一种规则）来进行打分的。但是这种调度的结果是任务最终被平均的分散到了所有机器上，并且每台机器上留出了一定的空闲空间来应对压力峰值。这直接造成了整个集群资源的碎片化。

与上述做法的相反的是另一个极端，即尽量让所有的机器都填满。但是这将导致任务不能很好的应对突发峰值。而且Borg或者用户对于任务所需的资源配额的估计往往不是很准确，尤其是对于batch job来说，它们所请求的资源量默认是很少的（特别是CPU资源）。所以在这种调度策略下batch job会很容易被填充在狭小的资源缝隙中，这时一旦遇到压力峰值，不仅batch job会出问题，与它运行在同一台机器上的LRS也会遭殃。

而Borg采用的是“混部加抢占”的模式，这种做法集成了上述两种模型的优点：兼顾公平性和利用率。这其中，LRS和batch job的混部以及优先级体系的存在为资源抢占提供了基础。这样，Borg在“可行性检查”阶段就可以考虑已经在此机器上运行的任务的资源能被抢占多少。如果算上可以抢占的这部分资源后此机器可以满足待调度任务的需求的话，任务就会被认为“可行”。接下，Borg会按优先级低到高“kill”这台机器上的任务直到满足待运行任务的需求，这就是抢占的具体实施过程。当然，被“kill”的任务会重新进入了调度队列，等待重新调度。

另一方面Borg也指出在任务调度并启动的过程中，安装依赖包的过程会构成80%的启动延时，所以调度器会优先选择已经安装好了这些依赖的机器。这让我想起来以前使用VMware开发的编排系统BOSH时，它的每一个Job都会通过spec描述自己依赖

哪些包，比如 GCC。所以当时为了节省时间，我们会在部署开始前使用脚本并发地在所有目标机器上安装好通用的依赖，比如 Ruby、GCC 这些，然后才开始真正的部署过程。 事实上，Borg 也有一个类似的包分发的过程，而且使用的是类似 BitTorrent 的协议。

这时我们回到 Kubernetes 上来，不难发现它与 Borg 的调度机制还比较很类似的。这当然也就意味着 Kubernetes 中没有借鉴传说中的 Omega 共享状态调度（反倒是 Mesos 的 Roadmap 里出现了类似”乐观并发控制“的概念）。

Kubernetes 的调度算法也分为两个阶段：

“Predicates 过程”：筛选出合格的 Minion，类似 Borg 的“可行性检查”。这一阶段 Kubernetes 主要需要考察一个 Minion 的条件包括：

- 容器申请的主机端口是否可用；
- 其资源是否满足 Pod 里所有容器的需求（仅考虑 CPU 和 Memory，且没有抢占机制）；
- volume 是否冲突；
- 是否匹配用户指定的 Label；
- 是不是指定的 hostname。

“Priorities 过程”：对通过上述筛选的 Minon 打分，这个打分的标准目前很简单：

- 选择资源空闲更多的机器；
- 属于同一个任务的副本 Pod 尽量分布在不同机器上。

从调度算法实现上差异中，我们可以看到 Kubernetes 与 Borg 的定位有着明显的不同。Borg 的调度算法中资源抢占和任务混部是两个关键点，这应是考虑到了这些策略在 Google 庞大的机器规模上所能带来的巨大的成本削减。所以 Borg 在算法的设计上强调了混部状态下对资源分配和任务分布的优化。而 Kubernetes 明显想把调度过程尽量简化，其两个阶段的调度依据都采用了简单粗暴的硬性资源标准，而没有支持任何抢占策略，也没有优先级的说法。当然，有一部分原因是开源项目的用户一般都喜欢定制自己的调度算法，从这一点上来说确实是“less is more”。总之，最终的结果是尽管保留了 Borg 的影子（毕竟作者很多都是一伙人），Kubernetes 调度器的实现上却完全是另外一条道路，确切的说更像 Swarm 这种偏向开发者的编排项目。

此外，还有一个非常重要的因素不得不提，那就是 Docker 的镜像机制。Borg 在 Google 服役期间所使用的 Linux 容器虽然应用极广且规模庞大，但核心功能还是 LXC 的变体或者强化版，强调的是隔离功能。这一点从它的开源版项目 lmctfy 的实现，以及论文里提到需要考虑任务依赖包等细节上我们都可以推断出来。可是 Docker 的厉害之处就在于直接封装了整个Job的运行环境，这使得 Kubernetes 在调度时可以不必考虑依赖包的分布情况，并且可以使用 Pod 这样的“原子容器组”而不是单个容器作为调度单位。当然，这也提示了我们将来进行 Docker 容器调度时，其实也可以把镜像的分布考虑在内：比如事先在所有工作节点上传基础镜像；在打分阶段优先选择任务所需基础镜像更完备的节点。

如果读者想感受一下没有镜像的 Docker 容器是什么手感，不妨去试用一下 DockerCon 上刚刚官宣的 runc 项目（https://github.com/opencontainers/runc）。runc 完全一个 libcontainer 的直接封装，提供所有的 Docker 容器必备功能，但是没有镜像的概念（即用户需要自己指定 rootfs 环境），这十分贴近 lmctfy 等仅专注于隔离环境的容器项目。

3.3 Borglet

离开了 Borgmaster 节点，我们接下来看一下工作节点上的 Borglet 组件，它的主要工作包括：

启停容器，进行容器失败恢复，通过 kernel 参数操作和管理 OS 资源，清理系统日志，收集机器状态供 Borgmaster 及其他监控方使用。

这个过程中，Borgmaster 会通过定期轮询来检查机器的状态。这种主动 poll 的做法好处是能够大量 Borglet 主动汇报状态造

成流量拥塞，并且能防止“恢复风暴”（比如大量失败后恢复过来的机器会在同段一时间不停地向 Borgmaster 发送大量的恢复数据和请求，如果没有合理的拥塞控制手段，者很可能会阻塞整个网络或者直接把 master 拖垮掉）。一旦收到汇报信息后，充当 leader 的 Borgmaster 会根据这些信息更新自己持有的 Cell 状态数据。

这个过程里，集群 Borgmaster 的“优越性”再次得到了体现。Borgmaster 的每个节点维护了一份无状态的“链接分片（link shard）”。每个分片只负责一部分 Borglet 机器的状态检查，而不是整个 Cell。而且这些分片还能够汇集并 diif 这些状态信息，最后只让 leader 获知并更新那些发生了变化的数据。这种做法有效地降低了 Borgmaster 的工作负载。

当然，如果一个 Borglet 在几个 poll 周期内都没有回应，他就会被认为宕机了。原本运行在整个节点上的任务容器会进入重调度周期。如果此期间 Borglet 与 master 的通信恢复了，那么 master 会请求杀死那些被重调度的任务容器，以防重复。Borglet 的运行并不需要依赖于 Borgmaster，及时 master 全部宕机，任务依然可以正常运行。

与 Borg 相比，Kubernetes 则选择了方向相反的状态汇报策略。当一个 kubelet 进程启动后，它会主动将自己注册给 master 节点上的 apiserver。接下来，kubelet 会定期向 apiserver 更新自己对应的 node 的信息，如果一段时间内没有更新，则 master 就会认为此工作节点已经发生故障。上述汇报信息的收集主要依赖于每个节点上运行的 CAdvisor 进程，而并非直接与操作系统进行交互。

事实上，不止 kubelet 进程会这么做。Kubernetes 里的所有组件协作，都会采用主动去跟 apiServer 建立联系，进而通过 apiserver 来监视、操作 etcd 的资源来完成相应的功能。

举个例子，用户向 apiserver 发起请求表示要创建一个 Pod，在调度器选择好了某个可用的 minion 后 apiserver 并不会直接告诉 kubelet 说我要在这个机器上创建容器，而是会间接在 etcd 中创建一个“boundPod”对象（这个对象的意思是我要在某个 kubelet 机器上绑定并运行某个 Pod）。与此同时，kubelet 则定时地主动检查有没有跟自己有关的“boundPod”，一旦发现有，它就会按照这个对象保存的信息向 Docker Daemon 发起创建容器的请求。

这正是 Kubernetes 设计中“一切皆资源”的体现，即所有实体对象，消息等都是作为 etcd 里保存起来的一种资源来对待，其他所有协作者要么通过监视这些资源的变化来采取动作，要么就是通过 apiserver 来对这些资源进行增删改查。

所以，我们可以把 Kubernetes 的实现方法描述为“面向 etcd 的编程模式”。这也是 Kubernetes 与 Borg 设计上的又一个不同点，说到底还是规模存在的差异：即 Kubernetes 认为它管理的集群中不会存在那么多机器同时向 apiserver 发起大量的请求。这也从另一个方面表现出了作者们对 etcd 响应能力还是比较有信心的。

3.4 可扩展性

这一节里与其说在 Borg 的可扩展性，倒不如说在讲它如何通过各种优化实现了更高的可扩展性。

首先是对 Borgmaster 的改进。最初的 Borgmaster 就是一个同步循环，在循环过程中顺序进行用户请求响应、调度、同 Borglet 交互等动作。所以 Borg 的第一个改进就是将调度器独立出来，从而能够同其他动作并行执行。改进后的调度器使用 Cell 集群状态的缓存数据来不断重复以下操作：

从 Borgmaster 接受集群的状态变化；
更新本地的集群状态缓存数据；
对指定的 Task 执行调度工作；
将调度结果告诉 Borgmaster。
这些操作组成了调度器的完整工作周期。

其次，Borgmaster 上负责响应只读请求和同 Borglet 进行交互的进程也被独立出来，通过职责的单一性来保证各自的执行效率。这些进程会被分配在 Borgmaster 的不同副本节点上来进一步提高效率，只负责同本副本节点所管理

的那部分 Worker 节点进行交互。

最后是专门针对调度器的优化。

缓存机器的打分结果。毕竟每次调度都给所有机器重新打一次分确实很无聊。只有当机器信息或者 Task 发生了变化（比如任务被从这个机器上调度走了）时，调度器缓存的机器分数才会发生更新。而且，Borg 会忽略那些不太明显的资源变化，减少缓存的更新次数。

划分 Task 等价类。Borg 的调度算法针对的是一组需求和约束都一样的 Task（等价类）而不是单个 Task 来执行的。

随机选择一组机器来做调度。这是很有意思的一种做法，即 Borg 调度器并不会把 Cell 里的所有机器拿过来挨个进行可行性检查，而是不断地随机挑选一个机器来检查可行性，判断是否通过，再挑选下一个，直到通过筛选的机器达到一定的数目。然后再在这些通过筛选的机器集合里进行打分过程。这个策略与著名的 Sparrow 调度器的做法很类似。

这些优化方法大大提高了 Borg 的工作效率，作者在论文中指出在上述功能被禁掉，有些原来几百秒完成的调度工作需要几天才能完全完成。

4. 可用性

Borg 在提高可用性方面所做的努力与大多数分布式系统的做法相同。比如：

- 自动重调度失败的任务；
- 将同一 Job 的不同任务分布在不同的高可用域；
- 在机器或者操作系统升级的过程中限制允许的任务中断的次数和同时中断的任务数量；
- 保证操作的幂等性，这样当客户端失败时它可以放心的发起重试操作；
- 当一台机器失联后，任务重调度的速度会被加以限制，因为 Borg 不能确定失联的原因是大规模的机器失败（比如断电），还是部分网络错误；
- 任务失败后，在一段时间内在本地磁盘保留日志及其他关键数据，哪怕对应的任务已经被杀死或者调度到其他地方了。

最后也是最重要的，Borglet 的运行不依赖于 master，所以哪怕控制节点全部宕机，用户提交的任务依然正常运行。

在这一部分，Kubernetes 也没有特别的设计。毕竟，在任务都已经容器化的情况下，只要正确地处理好容器的调度和管理工作，任务级别高可用的达成并不算十分困难。

至此，论文的前四章我们就介绍完了。通过与 Kubernetes 的实现作比较，我们似乎能得到一个“貌合神离”的结论。即 Kubernetes 与 Borg 从表面上看非常相似：相同的架构，相似的调度算法，当然还有同一伙开发人员。但是一旦我们去深入一些细节就会发现，在某些重要的设计和实现上，Borg 似乎有着和 Kubernetes 截然不同的认识：比如完全相反的资源汇报方向，复杂度根本不在一个水平上的 Master 实现（集群 VS 单点），对 batch job 的支持（Kubernetes 目前不支持 batch job），对于任务优先级和资源抢占的看法，等等。

这些本来可以照搬的东西，为什么在 Kubernetes 又被重新设计了一遍呢？在本文的第二部分，我们将一步步带领读者领悟造成这些差异的原因，即：资源回收和利用率优化。敬请关注。

参考文献

http://research.google.com/pubs/pub43438.html

https://github.com/googlecloudplatform/kubernetes

解析微服务架构（二）微服务架构综述

作者 王磊

在解析微服务架构(一)　单块架构系统以及其面临的挑战中，我们谈到了随着市场的快速发展，业务的不断扩大，单块架构应用面临着越来越多的挑战，其改造与重构势在必行。

微服务的诞生

微服务架构（Microservice Architect）是一种架构模式，它提倡将单块架构的应用划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。

微服务架构虽然诞生的时间并不长，但其在各种演讲、文章、书籍上所出现的频率已经让很多人意识到它对软件架构领域所带来的影响。

背景

其实，微服务的诞生并非偶然。它是互联网高速发展，敏捷、精益、持续交付方法论的深入人心，虚拟化技术与 DevOps 文化的快速发展以及传统单块架构无法适应快速变化等多重因素的推动下所诞生的产物。（见图 1）

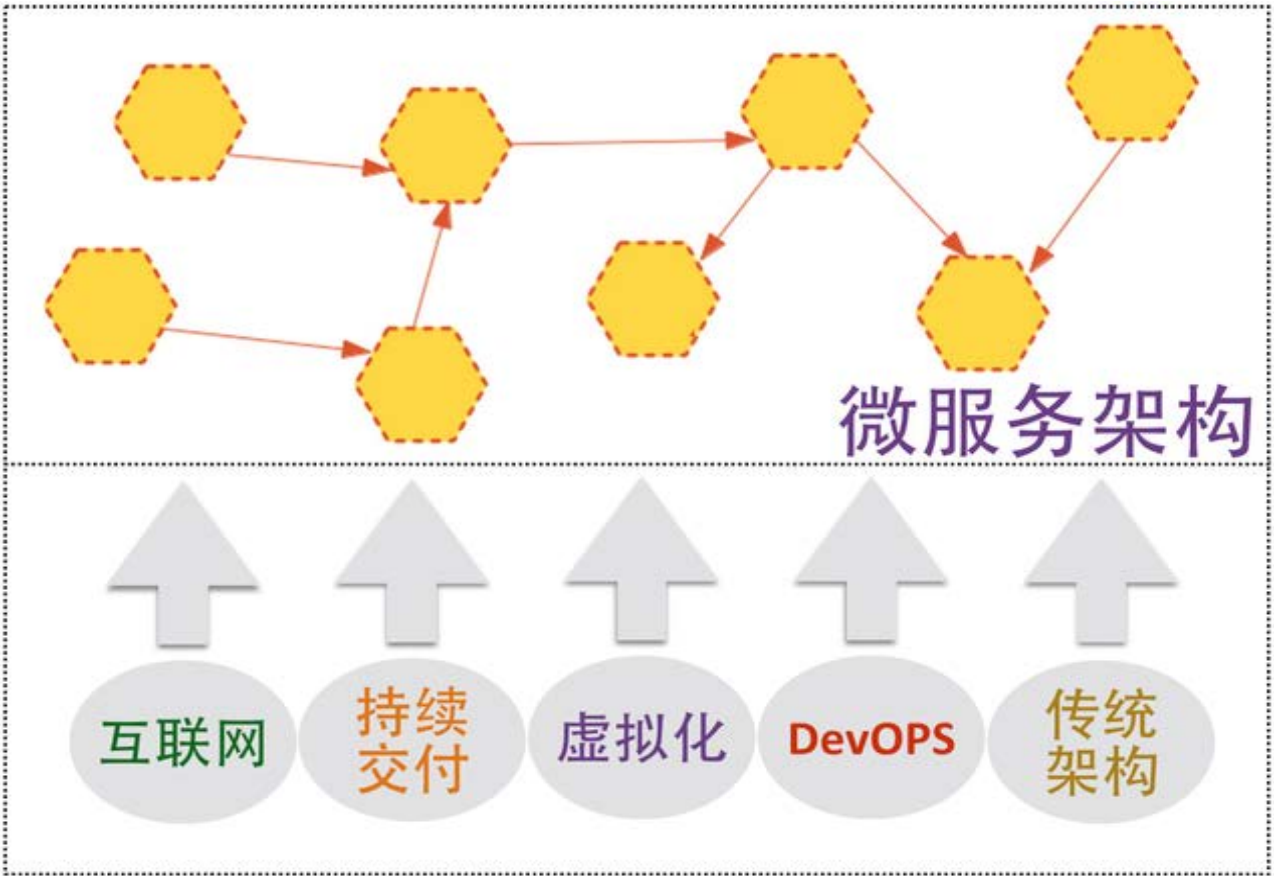


图1

1. 互联网行业的快速发展

过去的十年中，互联网对我们的生活产生了翻天覆地的变化。购物、打车、订餐、支付，甚至美甲、洗车等，想到的，想不到的活动都可以通过互联网完成，越来越多的传统行业公司也开始依赖互联网技术打造其核心竞争优势。互联网时代的产品通常有两类特点：需求变化快和用户群体庞大。在这种情况下，如何从系统架构的角度出发，构建灵活、易扩展的系统，快速应对需求的变化；同时，随着用户量的增加，如何保证系统的可伸缩性、高可用性，成为系统架构面临的挑战。

2. 敏捷、精益方法论的深入人心

纵观 IT 行业过去的十年，敏捷、精益、持续交付等价值观、方法论的提出以及实践，让很多组织意识到应变市场变化、提高响应力的重要性。精益创业（Lean Startup）帮助组织分析并建立最小可实行产品（Minimum Viable Product），通过迭代持续改进；敏捷方法帮助组织消除浪费，通过反馈不断找到正确的方向；持续交付帮助组织构建更快、更可靠、可频繁发布的交付机制。经过这些方法论以及实践的推行和尝试后，从宏观上而言，大部分组织已经基本上形成了一套可遵循、可参考、可实施的交付体系。这时候，逐渐完善并改进各个细节的需求就会更加强烈。所谓细节，就是类似如何找到灵活性高、扩展性好的架构方式、如何用更有效的技术、工具解决业务问题等。

3. 虚拟化技术与DevOps文化的快速发展

虚拟化技术和基础设施自动化 (Infrastructure As Code) 的快速发展极大的简化了基础设施的创建、配置以及系统的安装和部署。譬如云平台的成熟以及像 Chef、Puppet、Ansible 等工具的使用，让更多的基础设施能够通过自动化的方式动态创建。同时，容器化技术的发展以及 Docker 的出现，更是将虚拟化技术推向了一个史无前例的高潮。另外，DevOps 文化的推行打破了传统开发与运维之间的壁垒，帮助组织形成更高效的、开发与运维高度协作的交付团队。这些技术与文化的

快速发展，极大程度上解决了传统环境创建难、配置难以及‘最后一公里’的部署难、交付难等问题，成为推动微服务诞生、发展的重要因素之一。

4. 单块架构系统面临的挑战

几年前我们熟悉的传统 IT 系统，也可以称之为单块架构系统，是以技术分层，譬如逻辑层、数据层等。但随着用户需求个性化、产品生命周期变短、市场需求不稳定等因素的出现，单块架构系统面临着越来越多的挑战。因此，如何找到一种更有效的、更灵活、更适应当前互联网时代需求的系统架构方式，成为大家关注的焦点。

所以说，微服务的诞生决不是偶然，是多重因素推动下的必然产物。

微服务与SOA

SOA简述

早在 1996 年，Gartner 就提出面向服务架构（SOA）。SOA 阐述了“对于复杂的企业 IT 系统，应按照不同的、可重用的粒度划分，将功能相关的一组功能提供者组织在一起为消费者提供服务”，其目的是为了解决企业内部不同 IT 资源之间无法互联而导致的信息孤岛问题。

2002 年，SOA 被称作 " 现代应用开发领域最重要的课题之一 "，其正在帮助企业从资源利用的角度出发，将 IT 资源整合成可操作的、基于标准的服务，使其能被重新组合和应用。

但是，由于 SOA 本身的广义性以及抽象性，在其诞生的相当长一段时间内，人们对 SOA 存在着不同的认知和理解。（见图 2）

直到 2000 年左右，ESB、WebService、SOAP 等这类技术的出现，才使得 SOA 渐渐落地。同时，更多的厂商像 IBM、Oracle 等也分别提出基于 SOA 的解决方案或者产品。

微服务与SOA

实际上，微服务架构并不是一个全新的概念。仔细分析 SOA 的概念，就会发现，其和我们今天所谈到的微服务思想几乎一致。那在 SOA 诞生这么多年后，为什么又提出了微服务架构呢？

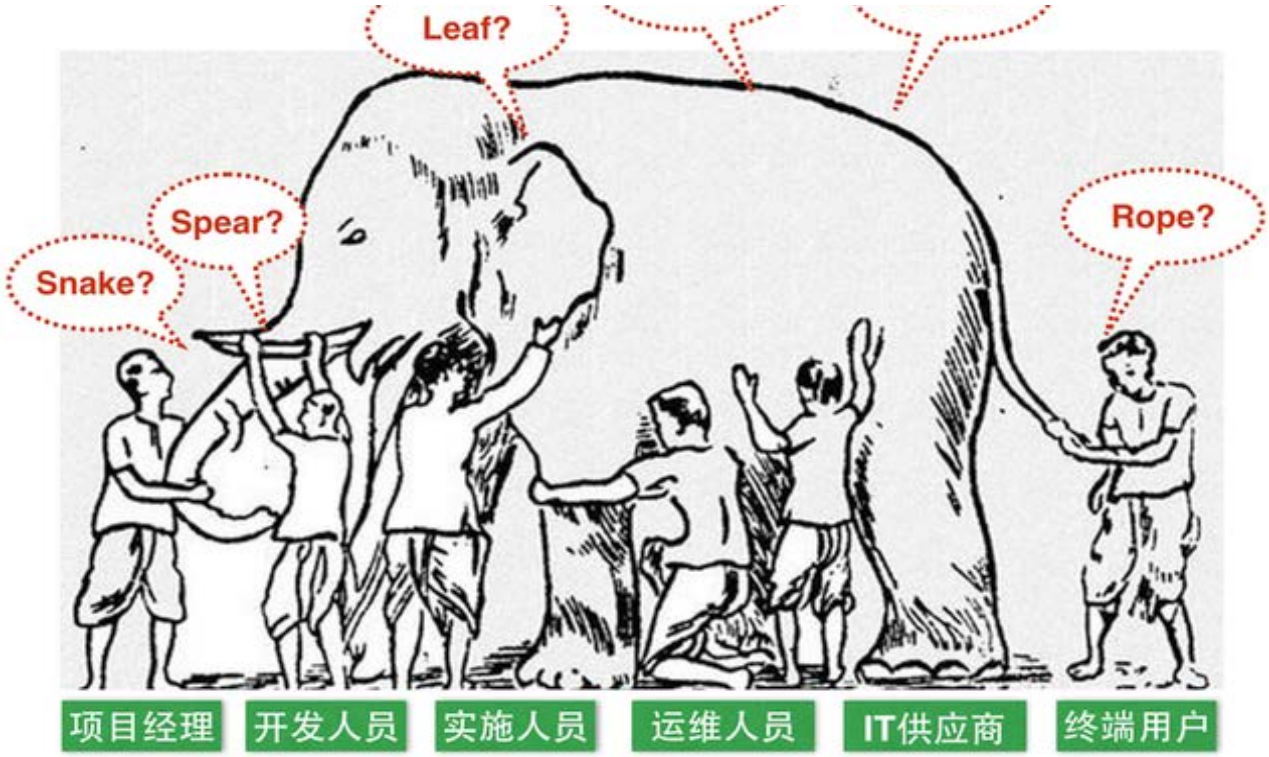


图2

鉴于过去十几年互联网行业的高速发展，以及敏捷、持续集成、持续交付、DevOps，云技术等的深入人心，服务架构的开发、测试、部署以及监控等，相比我们提到的传统的 SOA 实现，已经大相径庭，主要区别如下表所示。

SOA 实现	微服务架构实现
企业级，自顶向下开展实施	团队级，自底向上开展实施
服务由多个子系统组成，粒度大	一个系统被拆分成多个服务，粒度细
企业服务总线，集中式的服务架构	无集中式总线，松散的服务架构
集成方式复杂（ESB/WS/SOAP）	集成方式简单（HTTP/REST/JSON）
单块架构系统，相互依赖，部署复杂	服务都能独立部署

相比传统 SOA 的服务实现方式，微服务更具有灵活性、可实施性以及可扩展性，其强调的是一种独立测试、独立部署、独立运行的软件架构模式。

微服务架构的定义

其实，即便了解了上面的介绍，也很难对微服务下一个准确的定义。就像 NoSQL，我们谈论了好几年的 NoSQL，知道 NoSQL 代表着什么样的含义，也可以根据不同的应用场景选择不同的 NoSQL 数据库，但是我们还是很难对它下一个准确的定义。类似的，关于什么是‘函数式编程’，也或多或少存在同样的窘境。我们可以轻松的选择不同的函数式编程语言，可以轻松的写出函数式编程风格的代码，但很难对什么是函数式编程下一个准确的定义。

实际上，从业界的讨论来看，微服务本身并没有一个严格的定义。不过，ThoughtWorks 的首席科学家，马丁 – 福勒先生对微服务的这段描述，似乎更加具体、贴切，通俗易懂。

> 微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相沟通（通常是基于 HTTP 协议的 RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

总结下来，微服务架构中的核心部分包括以下几点：

- 小，且专注于做件事情
- 独立的进程中
- 轻量级的通信机制
- 松耦合、独立部署

总结

随着市场的快速发展，业务的不断扩大，单块架构应用面临着越来越多的挑战，其改造与重构势在必行。而微服务架构的诞生，是互联网高速发展，虚拟化技术应用以及持续交付、DevOps 深入人心的综合产物。随着用户需求个性化、产品生命周期变短，微服务架构是未来软件软件架构朝着灵活性、扩展性、伸缩性以及高可用性发展的必然方向。同时，以 Docker 为代表的容器虚拟化技术的盛行，将大大降低微服务实施的成本，为微服务落地以及大规模使用提供了坚实的基础和保障。

参考文献

http://microservices.io/
http://martinfowler.com/articles/microservices.html

在线观看演讲视频

使用微服务架构改造企业核心业务系统的实践

讲师：**王磊**

多范式编程语言——以Swift为例

作者 郭麟

Swift的编程范式

编程范式是程序语言背后的思想。代表了程序语言的设计者认为程序应该如何被构建和执行。常见的编程范式有：过程式、面向对象、函数式、泛型编程等。

一些编程语言是专门为某种特定范式设计的，例如，C 语言是过程式编程语言；Smalltalk 和 Java 是较纯粹的面向对象编程语言；Haskell、Scheme、Clojure 是函数式编程语言。

另外一些编程语言和编程范式的关系并不一一对应，如 Python、Scala、Groovy 同时支持面向对象和一定程度上的函数式编程。Swift 也是支持多种编程范式的编程语言。

由于代表了语言背后的思想，编程范式很大程度上决定了语言会呈现为何种面貌。用不着深入学习，仅仅浏览代码，就能发现 Scala 和 Swift 很类似，这是因为它们支持的编程范式是类似的；Scheme 和 Swift 看起来就相差很远，这是因为它们支持的编程范式很不一样。对于理解一门编程语言而言，相对于语言的语法和编写经验，理解语言的编程范式更重要。因为，就像看一本书，琢磨作者如何用词，如何构建章节是很重要，但更重要的是理解书所要表达的思想。

Swift 即支持面向对象编程范式，也支持函数式编程范式，同时还支持泛型编程。Swift 支持多种编程范式是由它的目标决定的。Swift 创造的初衷就是提供一门实用的工业语言。不同于 Haskell 这类出自大学和研究机构的具有学术性质的编程语言。苹果推出 Swift 时就带着着明确的商业目的：Mac OS 和 iOS 系统的主要编程语言 Objective-C 已显老态，Swift 将使得苹果系统的开发者拥有一门更现代的编程语言，从而促进苹果整个生态圈的良性发展。

Swift 的设计和开发无不体现着“实用的工业语言”这一目标。这决定了 Swift 无法做极端的语言实验，它需要在理智地面对现实的基础上，谨慎地寻求突破。这就决定了 Swift 需要继承历史遗产，在照顾现在大多数程序员的现实需求基础上，面向未来有所发展。

面向对象

面向对象编程的核心概念是继承，多态，和封装。以对象构建程序的基本单元的面向对象编程语言中，继承提供了一种复用代码的方法；多态提供了更高的抽象能力，使得我们可以设计出更通用的程序；封装提供一种使用代码更为便捷安全的机制。Swift 拥有以上所有的面向对象特性。所以，Swift 是一门完备的面向对象编程语

言。

Swift 继承了 Objective-C 面向对象方面的主要特性，提供以类为主的封装和继承机制。但给予了结构体（Struct）和枚举（Enum）更丰富的面向对象特征，使它们可以用于封装更为复杂的对象。另外，相对于 Objective-C，Swift 是一门更为安全的语言。

单继承，多协议

在继承上，Swift 不同于 C++ 可以继承一个或者若干个类，而类似于 Objective-C 和 Java，只能单继承。但 Swift 可以实现多个协议（Java 中对应的是接口 Interface）。这在一定程度上弥补了没有多继承的局限，同时又避免了多继承难以控制的缺陷。

除了实现协议，Swift 还可以实现多个扩展（Extension）。扩展是一种向已有的类，枚举或者结构体添加新功能的方法。扩展和 Objective-C 中的分类（Category）类似，但与 Objective-C 中的分类不同的是，Swift 中的扩展没有名字。

更强大的结构体，枚举

C++ 和 Java 等大部分面向对象编程语言主要以类（Class）作为实现面向对象的基本结构。Swift 则赋予了结构体（Struct）和枚举（Enum）更多的面向对象特征，使结构体和枚举也能承担部分数据封装工作。在其他一些语言需要用类来解决的场景中，Swift 可以使用结构体和枚举类型，而且更为合适。例如，Swift 的 Array 和 Dictionary 是用结构体实现，而不是用类实现的，这不同于大多数编程语言。

Swift 的结构体和枚举可以像类一样，完成下列事情：

- 定义属性
- 定义方法
- 拥有构造器
- 可以被扩展（Extension）
- 可以遵守协议（Protocol）

在封装这一点上，结构体和枚举几乎和类完全一致。不同的地方是，结构体和枚举是不能继承或者被继承的。所以，这两种数据类型也就没有多态性。

总结一下，Swift 中的类和其他面向对象编程语言的类一样是面向对象语言的核心概念，具有面向对象的基本特征。Swift 的结构体和枚举拥有比其他面向对象编程语言更多的面向对象特性，可以封装更复杂的对象。但不可继承，也就没有了多态性。

更多的值类型，而不是引用类型

结构体，枚举与类的另外一个区别是：结构体和枚举是值类型，而类是引用类型。

值类型在赋值和作为函数参数被传递时，实际上是在进行复制，操作的是对象的拷贝。Swift 中有大量值类型，包括 Number，String，Array，Dictionary，Tuple，Struct 和 Enum 等。

引用类型在赋值和作为函数参数被传递时，传递的是对象的引用，而并不是对象的拷贝。这些引用都指向同一个实例。对这些引用的操作，都将影响同一个实例。

在 Swift 中区分值类型和引用类型是为了将可变的对象和不可变的数据区分开来。可变的对象，使用引用类型；不可变的数据，使用值类型。值类型的数据，可以保证不会被意外修改。值类型的数据传递给函数，函数内部可以自由拷贝，改变值，而不用担心产生副作用。在多线程环境下，多个线程同时运行，可能会意外错误地修改数据，这常常会是一种难以调试的 bug。而使用值类型，你可以安全地在线程间传递数据，因为值类型传递是拷贝，所以无需在线程间同步数据变化。这就可以保证代码线程环境下的安全性。

结构体是值类型，暗示了结构体应该主要用于封装数据。例如，三维坐标系中的点 Point，代表几何形状的大小的 Size 等。而类是引用类型，意味着类应该用于封装具有状态的，可以继承的对象。例如，人，动物等。

Swift 中，Array、Dictionary、String 都是值类型，它们的行为就像 C 语言中的 Int 一样。你可以像使用 Int 一样简单安全地使用 Array，而不用考虑深度拷贝之类烦人问题。Swift 增强了对值类型的支持，鼓励我们使用值类型。因为值类型更安全。更多地使用值类型，将有助于我们写出行为更可预测，更安全的代码。

类型安全语言

Swift 是强类型语言，这意味着 Swift 禁止错误类型的参数继续运算。例如，你不能让 String 和 Float 相加。这与 C# 和 Java 一致；而与 C 和 Javascript 这类弱类型语言不一样。

Swift 是静态类型语言，这意味着 Swift 中变量是在编译期进行类型检查的。编译时，编译器会尽力找出包括类型错误在内的相关错误。例如，String 和 Int 相加这种类型运算错误，编译器在编译时就能告诉你，而不会在运行时才报错。这与 C# 和 Java 一致；而与 Python 和 Ruby 这类动态类型语言不一样。

Swift 不允许不正确的类型运算或类型转换发生，所以 Swift 是类型安全的。

Swift 支持类型推导，并且有一个相当不错的类型推导器。大部分情况下，你都不用声明类型，编译器可以根据上下文为你推导出变量的类型。

安全的初始化过程

Swift 中类（包括结构体和枚举）的初始化过程类似于 Java 的设计。Swift 有一类特别的方法，被作为初始化方法，它们没有 func 前缀，而是以 init 为方法名。这不同于 Objective-C 中的初始化方法只是一个普通的方法。对于初始化方法的特殊处理可以在语言机制上保证初始化方法只被调用一次。这种机制在 Objective-C 中是不存在的，在 Objective-C 中，初始化方法就像其它的普通方法一样，可以被多次调用。

Swift 中初始化方法必须保证所有

实例变量都被初始化。Swift 初始化方法要求特殊的初始化顺序。先保证当前类的实例变量被初始化，再调用父类的初始化方法完成父类实例变量的初始化。

Swift 保证了初始化方法只会被调用一次，同时所有的实例变量都会被初始化。这使得 Swift 初始化过程很安全。

安全的重写

Swift 提供了重写（Overriding）保护机制。如果要重写基类的方法，就必须在子类的重写方法前加上 overriding 关键字。这么做是向编译器声明你想提供一个重写版本。编译器会确认，基类里是否存在具有相同方法定义的方法。如果，基类中没有相同的方法定义，编译器就会报错。另一方面，如果没有加上 overriding 关键字的方法和基类的某个方法的定义相同，编译器也会报错，以防止意外的重写行为。这样就能从两方面保证重写行为的正确性。

Optionals

Swift 中的 Optionals 让我们能够更安全地应对有可能存在，也有可能不存在的值。在 Objective-C 里我们主要依靠文档来了解一个 API 是否会返回 nil。Optionals 则让我们将这份责任交给了类型系统。如果 API 的返回值声明为 Optional，就表示它可以是 nil。如果它不是 Optional，就表示它不可能是 nil。

在 Swift 中，类型后面加问号声明 Optional 类型，以及感叹号！对 Optional 类型拆包都只是语法糖。Optionals 其实是由枚举实现的。

也就是说，Optional 其实是一种枚举类型。我们通过语言的类型系统来明确可能为 nil 的情况。这比 Objective-C 中使用文档来说明要安全得多。

```
001 enum Optional<T>
      : Reflectable,
      NilLiteralConvertible
      {
002     case None
003     case Some(T)
004     //...
005 }
```

面向对象编程总结

现在绝大部分程序员的工作语言仍然是面向对象编程语言。大部分流行的现代编程语言都会允许你创建对象。面向对象编程语言易于建模。因为，对象和类似乎很容易和现实世界中的事物和概念对应。但编程实践表明，任何东西都成为对象并不是一件好事情。举一个 Java 中的蹩脚例子：Java 中只有对象才能作为参数传入函数（当然还有原始类型 Primitive Type），所以为了将函数作为参数传递给另一个函数，需要将函数包裹在一个对象中，通常会使用一个匿名类（这也是 Java 中，监听器 Listener 通常的实现方法），而这个类不会有其他作用，只是为了满足 Java 一切皆为对象的设计，从而通过编译。

Java 拥有纯粹的面向对象概念。它从设计之初，就希望以一切皆为对象的纯对象模型来为世界建模。但发展到现在，Java 中加入了越来越多非对象的东西。引入了闭包，从而获得了函数式编程中的一级函数；引入泛型，从而获得了参数化的类型。这可能暗示了，这个世界是如此丰富多彩，使用单一模型为世界建模并不会成功。

Swift 在追求统一纯粹的编程范式这一点上并不固执。Swift 完整地支持面向对象编程，拥有完备的面向对象基础概念。这使得熟悉面向对象编程的程序员学习和使用 Swift 的成本降低了。Java 或者 Objective-C 程序员对 Swift 的很多概念会觉得很熟悉。对他们而言，学习 Swift 并不困难，很快就能将 Swift 投入到实际生产之中。

同时，Swift 还一定程度上支持函数式编程风格。在适合函数式编程的场景下，同时程序员又拥有函数式编程的思维和能力时，可以使用 Swift 以函数式的编程方法改善生产力。这将在下一章详细介绍。

函数式编程

函数式编程是一种以数学函数为程序语言建模的核心的编程范式。它将计算机运算视为数学函数计算，并且避免使用程序状态以及可变对象。函数式编程思想主要有两点：

- 以函数为程序语言建模核心
- 避免状态和可变性

函数是函数式编程的基石。函数式编程语言的代码就是由一个个函数组合而成的。编写函数式语言的过程就是设计函数的过程。大规模程序由成千上万的函数组成，为了有效的组合这些函数。函数式编程语言，会尽量避免状态，避免可变对象。没有可变的状态，就使得函数式语言中的函数变为了纯函数。纯函数更容易模块化，更容易理解，对于复用是友好的。

函数

函数式编程的核心是函数，函数是“头等公民”。这就像面向对象语言的主要抽象方法是类，函数式编程语言中的主要抽象方法是函数。Swift 中的函数具有函数式语言中的函数的所有特点。你可以很容易地使用 Swift 写出函数式风格的代码。

高阶函数，一级函数

高阶函数，指可以将其他函数作为参数或者返回结果的函数。

一级函数，进一步扩展了函数的使用范围，使得函数成为语言中的“头等公民”。这意味函数可在任何其他语言构件（比如变量）出现的地方出现。可以说，一级函数是更严格的高阶函数。

Swift 中的函数都是一级函数，当然也都是高阶函数。

前文中举过 Java 中为了将函数作为参数传递给另外一个函数，需要将函数包裹在一个多余的匿名

类中的蹩脚例子。Swift 函数都是一级函数，可以直接将函数作为参数传递给另外一个函数。这就避免了 Java 里出现的这种多余的匿名类。

闭包

闭包是一个会对它内部引用的所有变量进行隐式绑定的函数。也可以说，闭包是由函数和与其相关的引用环境组合而成的实体。

函数实际上是一种特殊的闭包。

Objective-C 在后期加入了对闭包支持。闭包是一种一级函数。通过支持闭包，Objective-C 拓展其语言表达能力。但是如果与 Swift 的闭包语法相比，Objective-C 的闭包会显得有些繁重复杂。以下示例显示了 Swift 闭包语言的简洁和优雅：

```
001 let r = 1...3
002 let t = r.map { (i:
      Int) -> Int in
003     return i * 2
004 }
```

该例中，map 函数遍历了数组，用作为函数参数被传入的闭包处理了数组里的所有元素，并返回了一个处理过的新数组。例子中可以看到，Swift 中使用 {} 来创建一个匿名闭包。使用 in 来分割参数和返回类型。在很多情况下，由于存在类型推导，可以省略类型声明。

不变性

在介绍 Swift 的不变性之前，先讨论一下 Haskell 这门纯函数式语言。这将有助于我们对于不变性有更深刻的理解。

简单而言，Haskell 没有变量。这是因为，Haskell 追求更高级别的抽象，而变量其实是对一类低级计算机硬件：存储器空间（寄存器，内存）的抽象。变量存在的原因，可以视为计算机语言进化的遗迹，比如在初期直接操作硬件的汇编语言中，需要变量来操作存储过程。而在计算机出现之前，解决数学计算问题都是围绕构建数学函数。数学中，不存在计算机语言中这种需要重复赋值的变量。

Haskell 基于更抽象的数学模型。使用 Haskell 编程只需专注于设计数据之间的映射关系。而在数学上，表示两个数据之间映射关系的实体就是函数。这使得编写 Haskell 代码和设计数学函数的过程是一致的，Haskell 程序员的思路也更接近数学的本质。Haskell 摒弃了变量的同时，也抛弃了循环控制。这是因为没有变量，也就没有了控制循环位置的循环变量。这也很好理解。回忆一下我们在学习计算机之前的数学课程中，也无需使用到 for 这类概念。我们还是使用函数处理一个序列到另外一个序列的转换。

不变性导致另外一个结果，就是纯函数。没有可变的状态，没有可变对象，就使得函数式语言中的函数变为了纯函数。纯函数即没有副作用的函数，无论多少次执行，相同的输入就意味着相同的输出。一个纯函数的行为并不取决于全局变量、数据库的内容或者网络连接状态。纯代码天然就是模块化的：每个函数都是自包容的，并且都带有定义良好的接口。纯函数具有非常好的特性。它意味着理解起来更简单，更容易组合，测试起来更方便，线程安全性。

```
001 //变量
002 var mutable
003 //不变量
004 let immutable = 1
```

Swift 提供了一定程度的不变性支持。在 Swift 中，可以使用 var 声明普通的变量，也可以使用 let 快捷方便地声明不变量。
Swift 区分 var 和 let 是为了使用编译器来强制这种区分。Swift 中声明了不变量，就必须在声明时同时初始化，或者在构造器中初始化。除这两个地方之外，都无法再改变不变量。Swift 中鼓励使用不变量。因为，使用不变量更容易写出容易理解，容易测试，松耦合的代码。

不变性有诸多好处。

- 更高层次的抽象。程序员可以以更接近数学的方式思考问题。
- 更容易理解的代码。由于不存在副作用，无论多少次执行，相同的输入就意味着相同的输出。纯函数比有可变状态的函数和对象理解起来要容易简单得多。你无需再担心对象的某个状态的改变，会对它的某个行为（函数）产生影响。
- 线程安全的代码。这意味着多线程环境下，运行代码没有同步问题。它们也不可能因为异常的发生而处于无法预测的状态中。

不像 Haskell 这种纯函数式编程语言只能申明不可变量，Swift 提供变量和不可变量两种申明方式。程序员可以自由选择：在使用面向对象编程范式时，可以使用变量。在需要的情况下，Swift 也提供不变性的支持。

惰性求值

惰性计算是函数式编程语言的一个特性。惰性计算的表达式不在它被绑定到变量之后就立即求值，而是在该值被取用的时候求值。惰性计算有如下优点。

首先，你可以用它们来创建无限序列这样一种数据类型。因为直到需要时才会计算值，这样就可以使用惰性集合模拟无限序列。
第二，减少了存储空间。因为在真正需要时才会发生计算。所以，节约了不必要的存储空间。
第三，减少计算量，产生更高效的代码。因为在真正需要时才会发生计算。所以，节约那部分没有使用到的值的计算时间。例如，寻找数组中第一个符合某个条件的值。找到了之后，数组里该值之后的值都可以不必计算了。
纯函数式编程语言，如 Haskell 中是默认进行惰性求值的。所以，Haskell 被称为惰性语言。而大多数编程语言如 Java、C++ 求值都是严格的，或者说是及早求值。Swift 默认是严格求值，也就是每一个表达式都需要求值，而不论这个表达式在实际中是否确实需要求值。但是，Swift 也提供了支

持惰性求值的语法。在需要惰性时，需要显式声明。这为开发者在 Swift 中使用惰性提供了条件。

下面的例子展示了将默认是严格求值的数组变为惰性序列：

```
001 let r = 1...3
002 let seq = lazy(r).map {
003     (i: Int) -> Int in
004     println("mapping \
      (i)")
005     return i * 2
006 }
007 for i in seq {
008     println(i)
009 }
```

将获得如下结果：

```
001 mapping 1
002 2
003 mapping 2
004 4
005 mapping 3
006 6
```

结果显示 seq 是一个惰性序列。它的值只有在需要时才会真正发生计算。

函数式编程总结

函数式编程语言并不年轻，它的历史和面向对象编程一样悠久。1958 年被创造出来的 Lisp 是最古老的函数式编程语言。它比 C 语言年代更为久远。但直到最近，函数式编程思想才逐渐被重视。几乎所有新发明的编程语言都或多或少受到了函数式编程思想的影 响。Python、Scala、Groovy、Swift 都有一级函数，闭包。使得你可以将函数直接传给另外一个函数，函数也能够以返回值形式被另一个函数返回。消除状态，提供不变性的好处越来越多被接受，Scala、Groovy、Swift 都提供了声明不可变对象的方法，以支持编写更趋近于函数式风格的代码。

函数编程语言有其优秀的地方，也许将来会成为一个重要的编程范式。但是，函数式编程的重要性可能更多会间接地体现在影响其他编程语言的发展上。未来，可能很难出现一门主要以函数式编程范式设计的主流编程语言。如同 Java 这样的以单一编程范式（面向对象）构建，而成为主流的编程语言的机会应该不会太多了。如同 Haskell 这样追求纯粹的函数式编程语言，更多的可能只是一个偏学术的语言实验。

容我再重复一次上一节提到的理由：这个世界是如此丰富多彩，使用单一模式为世界建模可能并不会成功。当然，这类预测常常会被打破。如果，将来计算机领域出现了能解决所有问题的统一范式，我将很乐意再次学习和讨论它。但如果仅仅讨论现状的话，我们仍然不得不面对一个分裂和折衷的世界。

Swift 并不是一门主要以函数式编程范式构建的语言，它更多的是借鉴融合了函数式编程一些优秀思想（更灵活强大的函数，不变性的优点）。Swift 在大多数的场景下，仍然主要会以面向对象编程语言的面目出现。因为，作为另一门面向对象编程语言 Objective-C 的继任者，Swift 需要继承 Objective-C 的遗产：Cocoa。我们现在写 Swift 代码，大部分时候还是在 Cocoa 框架之上，可以说 Cocoa 就是 Swift 的标准库。在一个主要以面向对象语言编写的框架中写代码，最合适的思维方式仍然会是面向对象的。Cocoa 可以说是 Swift 得以在高起点出发的基础，也可以说其发生胎换骨变化的阻碍。

Swift 对函数式编程的支持，使得程序员多了一种选择。Swift 并不强迫程序员一定要以面向对象的方法思维。在场景合适的情况下，程序员可以选择使用函数式风格编写代码。如果确实是合适的场景，就能够改善生产力。

面向对象与函数式编程

如果，我们按语言范式给现在流行的语言分类，支持面向对象的编程语言应该会是最长的队伍。现在大部分流行的现代编程语言都是面向对象的，它们都会允许你创建对象。但同时，你会发现比较流行的几个编程语言，Python、Scala 甚至 Java 都或多或少都受到了函数式编程语言的影响。它们都引入一些函数式编程的概念，可以在一定程度上编写出具有函数式风格的代码。

在熟悉了类面向对象编程语言之后，再接触函数式编程语言，常常会觉得耳目一新，甚至隐约觉得函数式语言会是救世良方。那我们是否应该就此彻底转向函数式编程语言呢？使用 Haskell 来拯救世界？

面向对象编程语言在大规模实践之后，我们确实更深刻地了解了它们的缺点（例如，难以编写多线程环境下的软件应用；继承并不是代码复用的好方法）。函数式语言也确实有不少优点，有些优点恰恰就能解决面向对象语言的问题（纯函数十分适应多线程环境，纯函数天生就是模块化的，对于代码复用十分友好）。但是，函数式编程也许也存在某些问题。而这些问题，可能要在更大规模的业界实践之后才会暴露出来。现在我们已经认识到，单纯以对象为世界建模是有困难的。那么以数学模型来为世界建模可能也并不会好到哪里去。而可以确信的是，它们都有自己各自擅长的领域和环境。我们仍然还无法使用某种单一的编程范式来解决所有问题。

更大的现实是无数企业已经在面向对象编程语言上做了巨大的投资，即使现在面向对象编程已经暴露出一些问题，而函数式编程又呈现出不少能解决这些问题的优点，任何一个谨慎的人都不会，也不可能马上抛弃面向对象编程，彻底全面地转向函数式编程语言。

现实的选择是支持面向对象编程的同时，提供函数式的支持。这样，在大部分面向对象游刃有余的地方，仍然可以使用面向对象的方法。而在适合函数式编程的地方，而你又拥有函数式编程的思维和能力时，还可以采用函数式的编程方法改善生产力。

Swift 就是这样一个现实的选择。完善的面向对象支持，使 Swift 继承了 Objective-C 遗留下来的丰厚遗产。在 Swift 中使用

Objective-C 对象并不复杂。如果，你遇到一个对多线程安全性有要求的场景，需要使用函数式风格编写这部分代码，这在 Swift 中也是很轻松的。

泛型编程

泛型编程是另外一个有趣的话题。泛型为程语言提供了更高层级的抽象，即参数化类型。换句话说，就是把一个原本特定于某个类型的算法或类当中的类型信息抽象出来。这个抽象出来的概念在 C++ 的 STL（Standard Template Library）中就是模版（Template）。STL 展示了泛型编程的强大之处，一出现就成为了 C++ 的强大武器。除 C++ 之外，C#、Java、Haskell 等编程语言也都引入了泛型概念。

泛型编程是一个稍微局部一些的概念，它仅仅涉及如何更抽象地处理类型。这并不足以支撑起一门语言的核心概念。我们不会听到一个编程语言是纯泛型编程的，而没有其他编程范式。但正因为泛型并不会改变程序语言的核心，所以在大多数时候，它可以很好地融入到其他编程范式中。C++、Scala、Haskell 这些风格迥异的编程语言都支持泛型。泛型编程提供了更高的抽象层次，这意味着更强的表达能力。这对大部分编程语言来说都是一道美味佐餐美酒。

在 Swift 中，泛型得到广泛使用，许多 Swift 标准库是通过泛型代码构建出来的。例如 Swift 的数组和字典类型都是泛型集合。这样的例子在 Swift 中随处可见。

泛型函数

Swift 函数支持泛型。泛型函数通过将函数参数和返回值定义为泛型类型，使得函数可以作用于任何适合的类型。代码段 1 是一个简单的泛型函数。

泛型类型

除了泛型函数之外，Swift 还可以自定义泛型类，泛型结构体和泛型枚举。这样的泛型类型可以作用于任何类型，其用法和 Swift 提供的 Array 和 Dictionary 相同。

代码1：

```
001 func swapTwoValues<T>(inout a: T, inout b: T) {
002     let temporaryA = a
003     a = b
004     b = temporaryA
005 }
```

代码2：

```
001 // 定义一个泛型结构体
002 struct Stack<T> {
003     var items = [T]()
004
005     mutating func push(item: T) {
006         items.append(item)
007     }
008
009     mutating func pop() -> T {
010         return items.removeLast()
011     }
012 }
013
014 // 使用一个泛型结构体
015 var stackOfStrings = Stack<String>()
016 stackOfStrings.push("uno")
```

代码3：

```
001 protocol GeneratorType {
002     typealias Element
003     mutating func next() -> Element?
004 }
```

用一个栈（Stack）的例子展示泛型结构体的定义和使用。泛型枚举和泛型类的定义和使用方法是相同的。（见代码 2）

泛型类型参数 T 被用在了三个地方：

- 创建数组 items 时，指定了 items 中可以存储的数据类型；
- 指定了函数 push 的参数类型；
- 指定了函数 pop 的返回值类型。

泛型协议

对于协议，Swift 中没有提供类似结构体或类那样的方法来定义泛型协议。但我们可以使用 typealias 关键字定义该协议的关联类型，这样一定程度上可以模拟泛型协议的效果，例子见代码 3。

实现该协议的类必须定义一个别名为 Element 的关联类型。这和泛型的概念异曲同工，一定程度上实现了泛型协议。

泛型约束

在泛型的编程实践中，我们会遇到一些需要对泛型类型做进一步约束的场景。类型约束为泛型参数指定了一个类型，或者要求其实现某个特定的协议。比如，“:”意味着泛型参数指代的对象需要遵守 Equatable 协议。

类型约束对泛型参数的类型做了一定约束，可以强制要求泛型参数代表的类型遵守某个协议。而 where 语句可以更进一步对类型约束中声明的泛型参数所需要遵守的协议作出更详细的要求。where 语句也可以对协议的关联类型作进一步约束。比如，你可以要求两个泛型参数所遵守的协议的关联类型是相同的。

泛型编程总结

总体而言，Swift 提供了全面的泛型编程语法，让程序员可以写出抽象层次更高，更为灵活的代码，在避免了重复代码的同时，又能拥有良好的类型安全性。

总结

最后总结一下，Swift 是一门典型的多范式编程语言，支持面向对象是为了继承面向对象编程丰厚的成果；支持函数式编程，是为了探索新的可能；支持泛型编程，则是一道美味的佐餐美酒。

Swift 允许程序员在大部分时候，轻松地继续使用面向对象编程；而在适合函数式编程的场景下，同时程序员又拥有函数式编程的思维和能力时，还可以使用 Swift 以函数式的编程方法改善生产力；以及任何时候程序员都可以在Swift中使用泛型，提高抽象层次。

参考文档

- Blog: airspeedvelocity
- Apple's Swift blog
- Blog: objc.io
- Apple's Document “Swift Programming Language”

Swift是一门典型的多范式编程语言，支持面向对象是为了继承面向对象编程丰厚的成果；支持函数式编程，是为了探索新的可能；支持泛型编程，则是一道美味的佐餐美酒。

Oracle专家谈MySQL Cluster如何支持200M的QPS

作者 谢丽

Andrew Morgan是Oracle MySQL首席产品经理。 近日，他撰文介绍了MySQL Cluster如何支持200M的QPS。

MySQL Cluster简介

MySQL Cluster是一个实时可扩展且符合ACID的事务型内存数据库。该数据库有高达99.999%的可用性和低廉的开源软件总拥有成本。在设计方面，它采用了一种分布式、多主节点的架构，消除了单点故障，能够在商用硬件上横向扩展，并借助“自动分片（auto-sharding）”功能为通过SQL和NoSQL接口访问数据的读/写密集型工作负载提供服务。

最初，MySQL Cluster被设计成一个嵌入式的电信数据库，用于网内应用程序，需要具备运营商级的可用性和实时性能。之后，其功能随着新功能集的增加迅速增强，其应用领域随之也扩展到了本地或云上的Web、移动和企业应用程序，包括：大规模OLTP、实时分析、电子商务（库存管理、购物车、支付处理、订单追踪）、在线游戏、金融交易（欺诈检测）、移动与微支付、会话管理&缓存、流式推送、分析及推荐、内容管理与交付、通信与在线感知服务、订阅者/用户信息管理与权益等。

MySQL Cluster体系结构

在MySQL Cluster内部，总共有三种类型的节点为应用程序提供服务。下面是一张MySQL Cluster体系结构简图，其中包含6个节点组，共12个数据节点（见图1）。

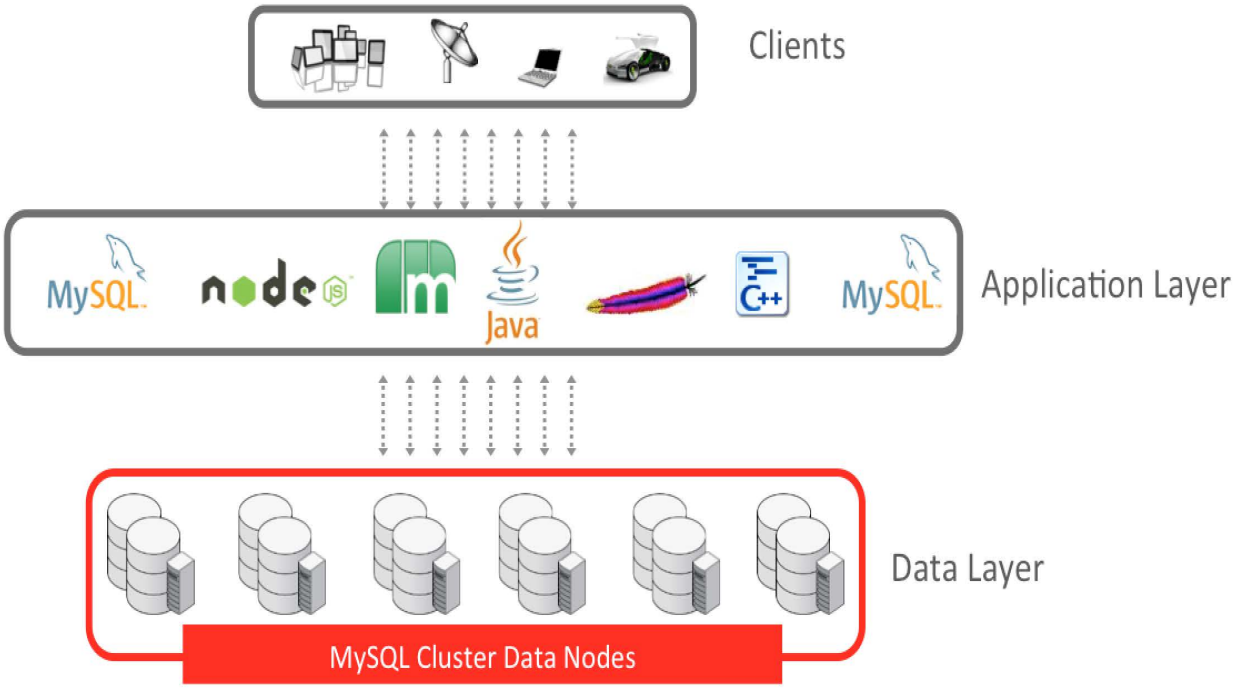


图1

数据节点是 MySQL Cluster 的主要节点。它们提供如下功能：内存内及基于磁盘的数据存储与管理、表的自动“分片（sharding）”及按用户定义分区、数据节点间数据同步复制、事务与数据检索、自动故障恢复、自我修复（故障解决后自动重新同步）。

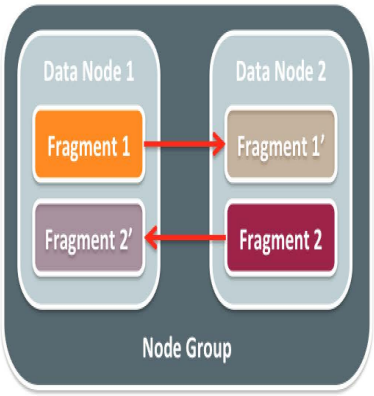
表会自动跨数据节点分区，每个数据节点都是一个可以接受写操作的主节点。这使得写密集型工作负载很容易在节点之间分配，而且对于应用程序而言，这个过程是透明的。

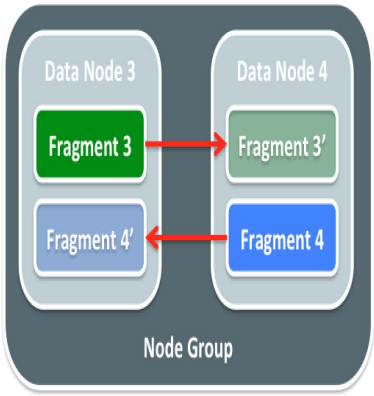
MySQL Cluster 采用了一种无资源共享的体系结构（比如不使用共享磁盘）存储和分发数据，并同步生成至少一个数据副本，如果某个数据节点出现故障，则总是有另一个数据节点存储了同样的信息，使得请求和事务可以继续而不被中断。任何在数据节点故障期间短暂中断（亚秒级）的事务都可以回滚并重新执行。

MySQL Cluster 允许用户选择如何存储数据：全部在内存中或者部分在磁盘上（仅限于未索引数据）。内存内存储对于经常变化的数据（活动工作集）而言尤其有用。存储在内存中的数据会定期地（本地检查点）写入本地磁盘，并在所有数据节点之间协调，这样，MySQL Cluster 可以从系统完全失效（比如停电）的情况下恢复过来。基于磁盘的存储可以用于存储性能要求不那么严格的数据，其数据集大于可用内存。与其它大多数数据库服务器一样，为了提高性能，MySQL Cluster 使用页缓存将经常使用的、基于磁盘存储的数据缓存在数据节点的内存中。

应用节点提供从应用逻辑到数据节点的连接。应用程序可以使用 SQL 访问数据库，通过一台或多个 MySQL 服务器对存储在 MySQL Cluster 中的数据执行 SQL 接口的功能。当访问 MySQL 服务器时，可以使用任何一种标准的 MySQL 连接器，这使用户有许多种访问技术可选择。NDB API 是其中一个可选的方案。这是一个基于 C++ 的高性能接口，可以提供额外控制、更好的实时行为及更高的吞吐能力。NDB API 还提供了一个层，使 NoSQL 接口可以绕过 SQL 层直接访问 MySQL Cluster，降低了延迟，提高了开发灵活性。现有接口包括 Java、JPA、Memcached、JavaScript 与 Node.js、HTTP/REST（借助 Apache Module）。所有应用节点都可以访问所有数据节点的数据，所以，它们即使出现故障也不会导致服务中断，因为应用程序只要使用剩下的节点就可以了。

管理节点负责向 MySQL Cluster 中的所有节点发布集群配置信息以及节点管理。管理节点在启动、向集群加入节点及系统重新配置时使用。管理节点关闭和重启不会影响数据节点和应用节点的运行。在默认情况下，在遇到导致“集群分裂（split-brain）”或网络分区的网络故障时，管理节点还提供仲裁服务。





通过透明分片实现可扩展性

任何表的行都可以透明地分成多个分区 / 片段。对于每一个片段，都会有一个单独的数据节点保存它所有的数据，并处理所有针对那些数据的读写操作。每个数据节点还有一个伙伴节点，它们共同组成了一个节点组；伙伴节点存储了那个片段的第二个副本以及一个它自己原有的片段。MySQL Cluster 使用同步两段提交协议确保事务提交的变化同时存储到两个数据节点。

MySQL Cluster 默认使用表的主键作为“分片键（shard key）”，并对分片键执行 MD5 散列，从而选择数据应该存储的片段 / 分区。如果一个事务或查询需要访问多个数据节点的数据，那么其中一个数据节点将承担事务协调器的角色，并将工作委派给其它所需的数据节点；结果会在提供给应用程序前合并。需要注意的是，事务或查询可以连接来自多个分片和多个表的数据，这与传统的、实现了分片机制的 NoSQL 数据存储相比是一个巨大的优势。（见图 2）

当单个节点就可以满足高强度查询 / 事务的数据操作需求时，就实现了最理想的（线性）扩展（因为这减少了数据节点间消息传递的网络延迟）。要做到这一点，应用程序应该清楚地知道数据分布——这实际上就是说定义模式的人可以指定用作分片键的列。比如，上图中的表使用了由 user-id 和服务名组合而成的主键；如果只使用 user-id 作为分片键，那么表中特定用户的所有行将会总是存储在同一个片段中。更为强大之处在于，如果其它表中也使用了同样的 user-id 列，并将其设定为分片键，那么所有表中特定用户的数据都会存储在同一个片段中，那个用户的查询 / 事务就可以由单个数据节点处理。

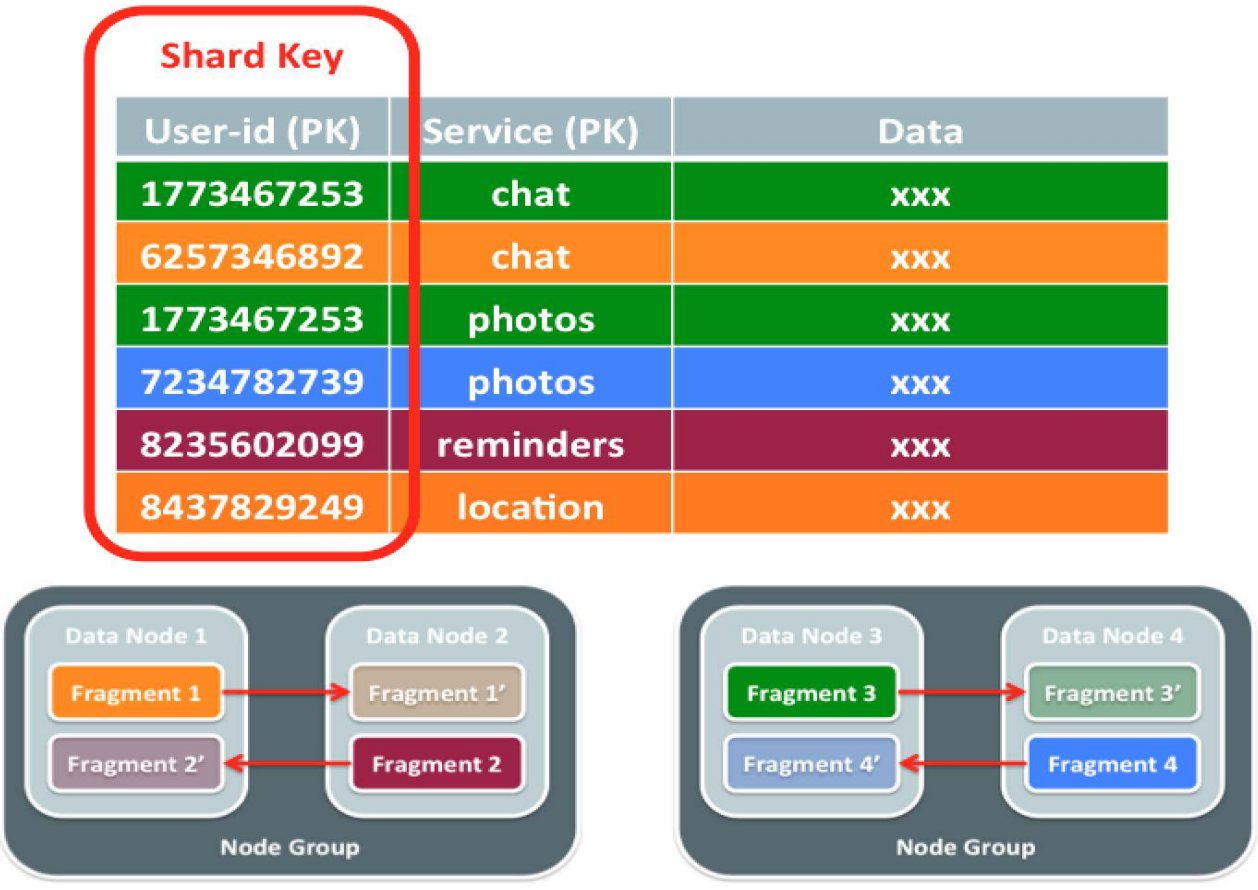


图2

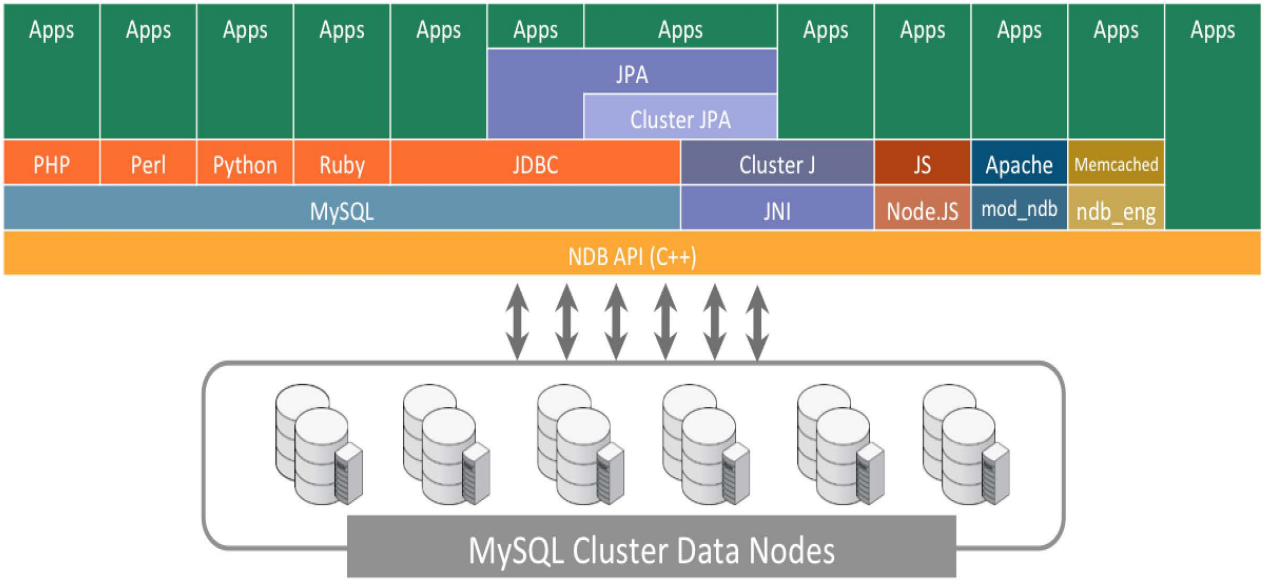


图3

利用NoSQL API最大限度地提高数据访问速度

MySQL Cluster 提供了许多种数据访问方式；最常用的方法是 SQL，但从下图可以看出，还有许多原生 API 可供应用程序从数据库直接读 / 写数据，避免了向 SQL 转换并传递给 MySQL 服务器的低效和开发复杂度。目前，MySQL Cluster 提供了面向 C++、Java、JPA、JavaScript/Node.js、HTTP 及 Memcached 协议的 API。

基准测试：每秒2亿次查询

根据设计，MySQL Cluster 用于处理以下两种工作负载：

OLTP（在线事务处理）：内存优化型表可以提供次毫秒级的低延迟以及极高水平的 OLTP 工作负载并发能力，并且仍然可以提供良好的稳定性；此外，它们也能够用于基于磁盘存储的表。

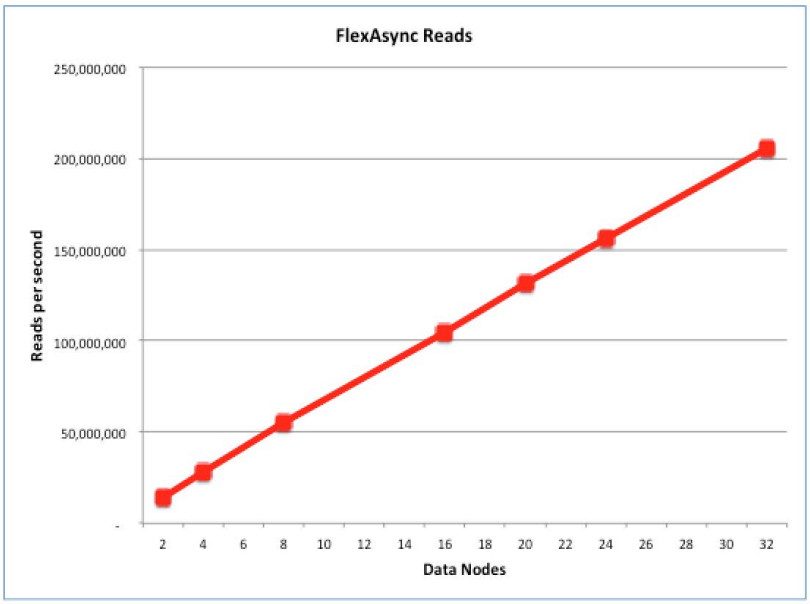
即时搜索：MySQL Cluster 提高了执行表扫描时可以使用的并发数，极大地提高了未索引列的搜索速度。

话虽如此，MySQL Cluster 旨在处理 OLTP 工作负载方面达到最佳，特别是在以并发方式发送大量查询 / 事务请求的情况下。为此，他们使用 flexAsynch 基准测试，测量更多数据节点加入集群后 NoSQL 访问性能的提升。

在该基准测试中，每个数据节点运行在一个专用的 56 线程 Intel E5-2697 v3（Haswell）机器上。上图显示了在数据节点从 2 增加到 32（注意：MySQL Cluster 目前最多支持 48 个数据节点）的过程中吞吐量的变化。从中可以看出，吞吐量呈线性增长，在 32 个数据节点时，达到了每秒 2 亿次 NoSQL 查询。

读者可以登录 MySQL Cluster 基准测试页面，查看关于这次测试的最新结果及更详细的描述。

每秒 2 亿次查询的基准测试是在 MySQL Cluster 7.4（最新的正式版本）上得出的，关于该版本的更多信息请查看这里。



杂谈：创业公司的产品开发与团队管理

作者 贾彦民

一般来说，创业公司规模小，人员少，没有大公司的官僚作风。而官僚作风是很害人的东西，记得在大公司时，本来一言　　而决的一点小事，常常因为害怕承担失败的责任，或参与者（如管项目或管人的经理们）有意的要突出自己的影响力或存在感，在各色人等中往来穿梭，仿佛煞有介事，经过无数次的会议讨论却议而不决。而软件工程师们为配合这样的戏码常常被搞得焦头烂额，无可适从。创业公司由于管理层次简单，很少受到官僚作风的困扰，但也不能想当然地认为万事大吉，高枕无忧了。其实创业公司也有创业公司的局限性，这些局限性常常被忽略，从而影响到产品的开发，兹总结如下。

创业不一定志高。人们往往认为创业公司旨在 IPO，对自己的产品计划往往雄心勃勃。其实并不尽然，有人就认为，创业公司的资源配备不能和大公司相提并论，因此，做出的产品比大公司差些也是理所当然的。而创业公司产品的优势主要在于低廉的价格。听起来似乎有道理，其实是在为自身的懒惰或能力上的缺陷寻找借口。就如同国家足球队那样没有出息，踢输了，或怪草皮太硬，或怪裁判不公，或怪状态不佳，或怪对手太强。在竞争如此激烈的市场，一个创业公司如果做不出最好的产品，为用户提供实实在在的价值，既没有生存的可能，也没有生存的必要。因此，三军可夺其帅，匹夫不可夺其志也，唯有如此，创业公司才有成功的可能性。

简单不一定高效。有些创业公司不太注重项目制定计划，或者不太善于制定项目计划。没有完善可行的项目计划，就更谈不到项目状态和进度的检查和管控，常常是做到哪里算哪里。当然看起来很简单，高效却是未必。大家不清楚什么时间做什么事，也不知道事情的轻重缓急。每个人似乎都在忙忙叨叨，而究竟忙些什么，却是糊里糊涂。临近项目结束时，发现还差很多，于是加班加点赶进度，哪里还顾得上产品质量。

人少不一定沟通流畅。软件开发当然需要团队合作，无论团队大小，项目经理、产品经理、

UX 设计师、开发工程师、测试工程师需要准确有效地沟通协调。对于新的功能：产品经理一定要讲明白它的作用和价值，开发工程师才有信心做好；开发工程师要和测试工程师一起检查主要的功能测试点，制定测试计划；项目经理要确保开发计划让所有人都清楚地知道，并协调任务之间的相互依赖。创业公司人少事多，有时候身兼数职，若没有有意识的沟通，大家都埋头于自己的工作，个人自扫门前雪，莫管他人瓦上霜，就可能出现这样的情况，事情做完了，才突然很惊讶地发现完全搞错了。年轻不一定朝气蓬勃。曾国藩认为：“军事是喜朝气，最忌暮气，惰则皆暮气也。”创业阶段的公司本应该是披荆斩棘，锐意进取，表现出勃勃的生机。但创业的艰辛往往超出创业者的想象，不断的挫败侵蚀着创业者的意志，从而直接影响民心士气。如果面对困难既没有愿景，也没有解决之道，由失败引起的沮丧就可能变成不可逆转的暮气，最终压垮整个公司。

正如有什么样的人民就有什么样的政府，产品的品质归根结底取决于做产品的人，有什么样的工程师（开发和测试）、设计师、架构师、产品经理，就有什么样的产品。因此，保证产品品质最有效的办法莫过于打造一支高水平的团队，而团队建设很大程度上取决于领导的艺术。

让我们来看看从历史中可以学到什么。秦失其鹿，天下人共逐之。最后的竞争在最有实力的两个对手刘邦和项羽之间展开，当然最终的结果是屡败屡战的刘邦完胜屡战屡胜的项羽。历史学家们从个人性格、政治智慧、军事才能、战略战术等各个角度对胜负之数做了全方位的分析，我认为决定最终胜负的最重要的原因当属创业**团队的建设**。刘邦的团队网罗了当时顶尖的一流人才，项羽当初的创业团队也毫不逊色。项羽拥有战国贵族的血统，接受过良好的教育，“见人恭敬慈爱，言语呕呕，人有疾病，涕泣分食饮”，更尊范增为亚父。而平民出身流氓习性难改的刘邦却慢而侮人，甚至于向儒生的帽子里面撒尿。但是在项羽实力鼎盛的时候，一些了不起的大牛如陈平、韩信之辈却选择去项王而从沛公游，何也？原因其实很简单，刘邦长期在黑社会中混迹的经历以及在革命斗争的实践中锻炼出了高度的智慧和不凡的见识。使他有能力明白团队中大牛们的高明见解，闻弦歌而知雅意，并且不断激发团队的想象力和创造力。这其实就是所谓的**领导力**，也是刘邦“不能将兵，而善将将”的秘诀。而项羽虽然个人素质过硬，具有很高的军事天分，巨鹿一战，大破秦军，“召见诸侯将，入辕门，无不膝行而前，莫敢仰视”。但他毕竟“too young, too simple, sometimes naive”，不是不尊重别人的建议，而是缺乏君临天下而应有的领导力，稍微高深一点的意见便不能理解。让我们来看一个具体的例子，当有人劝霸王定都关中的时候，他却说：“富贵不归故乡，如衣锦夜行，谁知之者？”如此幼稚可笑的话，真叫人瞠目结舌，怪不得人家骂他沐猴衣冠，竖子不足与谋。而面对同样的建议，尽管“左右大臣皆山东人，多劝上都雒阳”，但是刘邦却知道这个意见的重要价值，排除众议，定都关中，为后世弭平七国之乱起到了至关重要的作用。所以，一流的人才都离开项羽，跳槽到刘邦那里，把好的**主意**讲给听得懂的人了。

关于这一点，Steve Jobs 讲得更明白：

“For most things in life, the range between best and average is 30% or so. The best airplane flight, the best meal, they may be 30% better than your average one. What I saw with Woz was somebody who was fifty times better than the average engineer. He could have meetings in his head. The Mac team was an attempt to build a whole team like that, A players. People said they wouldn’t get along, they’d hate working with each other. But I realized that A players like to work with A players, they just didn’t like working with C players.”

一流的人才愿意和一流的人才一起工作，因为他们之间有共同语言，相互理解；而一流的人才却不愿意和三流的人才一起工作，因为三流的人才根本不能明白一流的人才的想法和意图，讨论问

题时总是缠夹不清，糊糊涂涂，无可无不可。另外，刘邦比之项羽还有另一个重要的优点，那就是毫不吝啬的激励措施，刘邦“使人攻城略地，因以与之，与天下同其利”，而项羽“至使人有功当封爵者，印刓敝，忍不能予。”所以，百战百胜的项羽不能保证最后决战的胜利。垓下一战，只落得霸王别姬，英雄气短，天数耶？人事耶？历史事件常常会惊人的相似，相似的剧本在三国时期再次上演。这次的主角换成了雄才大略的曹操和刚愎自用的袁绍。袁绍榆木疙瘩一样简单的大脑也同样不能理解高明的见解，一而再，再而三地失去良机。在官渡之战中被实力远不如自己的曹操击败，从此一蹶不振。

历史的经验更加验证了那句俗话，兵熊熊一个，将熊熊一窝。做领导的不一定十八般兵器，样样精通，但一定要加强修养，提高领导力，这是打造一支优秀的开发团队的必要条件。产品的开发是百分之百的智力游戏和创造性活动，每个工程师的最重要的贡献是 TA 的创造力和想象力。而领导要有足够的智慧和见识用来领悟他们高明的建议，并想方设法利用各种激励措施把他们的想象力和创造力最大限度地激发出来。相反，一个比较“熊”的领导形式上搞再多的头脑风暴，对所谓的创新也一样的于事无补。我曾经历过这样一个产品开发团队，几乎所有的人对产品的目标和价值都不甚了了，甚至说不明白产品是什么，也不知道如何与同类型的产品做区分。很多人建议要把这些问题好好讨论清楚，但是团队的领导认为先做起来看，在开发中逐步加深认识，也许一下子所有的问题都清楚了，最不济大家也可以在开发中练练手。最终的结果可想而知，产品不可避免地失败了，刚刚招聘成立的部门也全部解散了。这不仅是公司的损失，而且影响到每一位员工的职业生涯，同时也浪费了宝贵的生产力。这说明如果领导把握不住产品的正确方向，即使开发团队努力再多也不过是徒劳无益，与成功南辕北辙，渐行渐远。

大公司设计的那些官僚而笨拙的开发管理流程不是创业公司可以承受之重。其实，最好的管理就是没有管理。无为而治，所有人都在生产第一线，写代码，做测试。仿佛是一条由机器人操控的自动生产线，所有的开发任务都按部就班进行，顺利得像流水一样。没有任何人力、物力浪费在管理工作上，当然生产效率是最高的。我想，这是一种可望而不可即的理想状况。因为，产品开发本身不是流水线式的作业，存在许多不确定因素，比如对于每一项开发任务，几乎不可能正确地估计出需要的时间，产品的需求随时可能发生变化，任务之间存在各种依赖关系。所以，我们需要把开发工作划分为具体的可操作的开发任务，估计每项任务所需的时间，根据需求的变化确定任务的优先级，协调任务之间的进度等，所有这些都属于管理工作的范畴。因此，我们只能尽量减少管理，而做不到完全没有管理的理想状态。为了下文叙述的方便，我们把管理产品开发的人统称为开发经理，可能的角色包括产品经理，项目经理，技术主管等。开发经理应努力克制自己，尽量减少因为管理对工程师的工作带来的干扰以及时间与精力的开销，让工程师专注于创造性的工作。开发经理要多做功课，精简管理的流程，没有必要开的会就不要去开，能够开短会的就不要开长会，小会能够解决问题的就不要开大会。记得在某大公司时，开发经理们为了同步项目的进度和状态，早上工程师开始工作之前，每个开发小组都要花一刻钟到半个小时的时间开一个例会，报告自己昨天的工作和今天的计划。另外，还有数不清的各种各样的项目状况汇报会。这当然方便了开发经理了解项目的状况，但却浪费了工程师的时间和精力，去一遍又一遍地应付开发经理们重复且无聊问题。

对于创业公司而言，管理要做到尽量简单，只要每个人都有事做，清楚地知道什么时间做什么事，并保证每件事的质量，就基本达到管理的目的了。开发经理要像一个在场边指挥的足球教练，能够及时发现团队或个人的问题和缺陷，并采取有效措施加以补救，不能等到形势严重了还浑然不觉，而当团队运行顺利时，开发经理就应该像空气一样自动消失；开发经理要像一个高明的调酒师，调酒师知道各种调料的风

格味道，并能按照恰当比例制成美酒佳酿，开发经理也要清楚团队中每个工程师的性情，优点和缺点，取长补短，合理搭配组合，高效率、高质量地完成任务；开发经理要像一个拉拉队长，激发工程师的创造力和想象力，鼓舞士气，传播正能量；开发经理要像一个后勤部长，凡是和产品开发不直接相关的杂事，都能搞定。

为了追求简单高效，照搬大公司的产品开发流程固然不可取。但对于行之有效的方法也断不能一概拒之门外。比如，每个人都要清楚地知道产品开发计划，规范测试的方法和流程等。当然有些方法要经过改造才能适用于创业公司。比如代码审核，很多程序员认为是在给自己的工作挑刺，仿佛被批斗一样，感觉很不舒服，甚至有些抗拒。若从另外一个角度来看，代码审核就是另外不同的情景。当程序员对应用的需求不太明确时，代码审核可以用来帮助理清需求；当程序员对自己的代码没有信心时，代码审核可以用来帮助发现问题，修正错误；当程序员写出优雅的算法时，代码审核可以用来展示 TA 的奇思妙想。所以，代码审核绝不应该是挑战程序员，而是帮助程序员。这一点同程序员解释清楚了，那么，程序员就会主动要求审核自己的代码了。

QCon 全球软件开发大会2015

International Software Development Conference

旧金山 上海 伦敦 北京 圣保罗 东京 纽约 里约热内卢

2015.10.15-17 上海·光大会展中心国际大酒店 www.qconshanghai.com

主题演讲:

畅销书《番茄工作法图解》作者

Staffan Nöteberg

Azul Systems联合创始人兼CTO

Gil Tene

Uber首席系统架构师, Voxer联合创始人

Matt Ranney

演讲专题:

机器学习
安全与风控
技术创业
开源文化面面观
建设高效团队
运维之痛
容器与云计算
移动开发新趋势
高可用架构
新兴大数据处理
互联网产品案例研究
新时代的前端
Java优化面面观
再谈软件交付
编程语言选型与实战

[上海站]

8折抢票中...

节省1360元
优惠截至8月16日

5人以上团购 更多给力优惠

Brought by Geekbang. 极客邦科技 InfoQ

www.qconferences.com

InfoQ 中文站　2015迷你书

架构师 月刊 2015年7月

本期内容推荐：Docker、CoreOS握手言和，共同制定容器标准，WWDC总结：开发者需要知道的iOS 9 SDK新特性，NGINX引入线程池性能提升9倍，深入浅出ES6：生成器 Generators，深入浅出React：React开发神器Webpack，姜宁谈红帽绩效考核：不关心员工具体做什么。

开源启示录 第一季

开源软件的未来在于建立一个良性循环，以参与促进繁荣，以繁荣促进参与。在这里，我们为大家呈现本期迷你书，在揭示些许开源软件规律的之外，更希望看到有更多人和企业参与到开源软件中来。

顶尖技术团队访谈录 第二季

本次的《中国顶尖技术团队访谈录》·第二季挑选的九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。

云生态专刊

《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”。