

Coding-Job：从研发到生产的容器化融合实践

Coding-Job：从研发到生产的容 器化融合实践

Coding-Job, a fusion of dockernized Prod & Dev
environments

xh4n3@Coding

大家好，我是来自 Coding 的基础架构工程师，我有幸在 Coding 参与了 Coding-Job 这个容器化的编排平台的研发。大家对 Coding 可能有所了解，Coding 是一个一站式开发平台，具有代码托管，任务管理，产品演示和 WebIDE 等功能。我们在去年也推出了码市众包平台，帮助项目需求方找到高质的开发者，帮助我们开发者找到合适的项目。整体功能看起来比较复杂且较为分散。

Coding 架构的演进

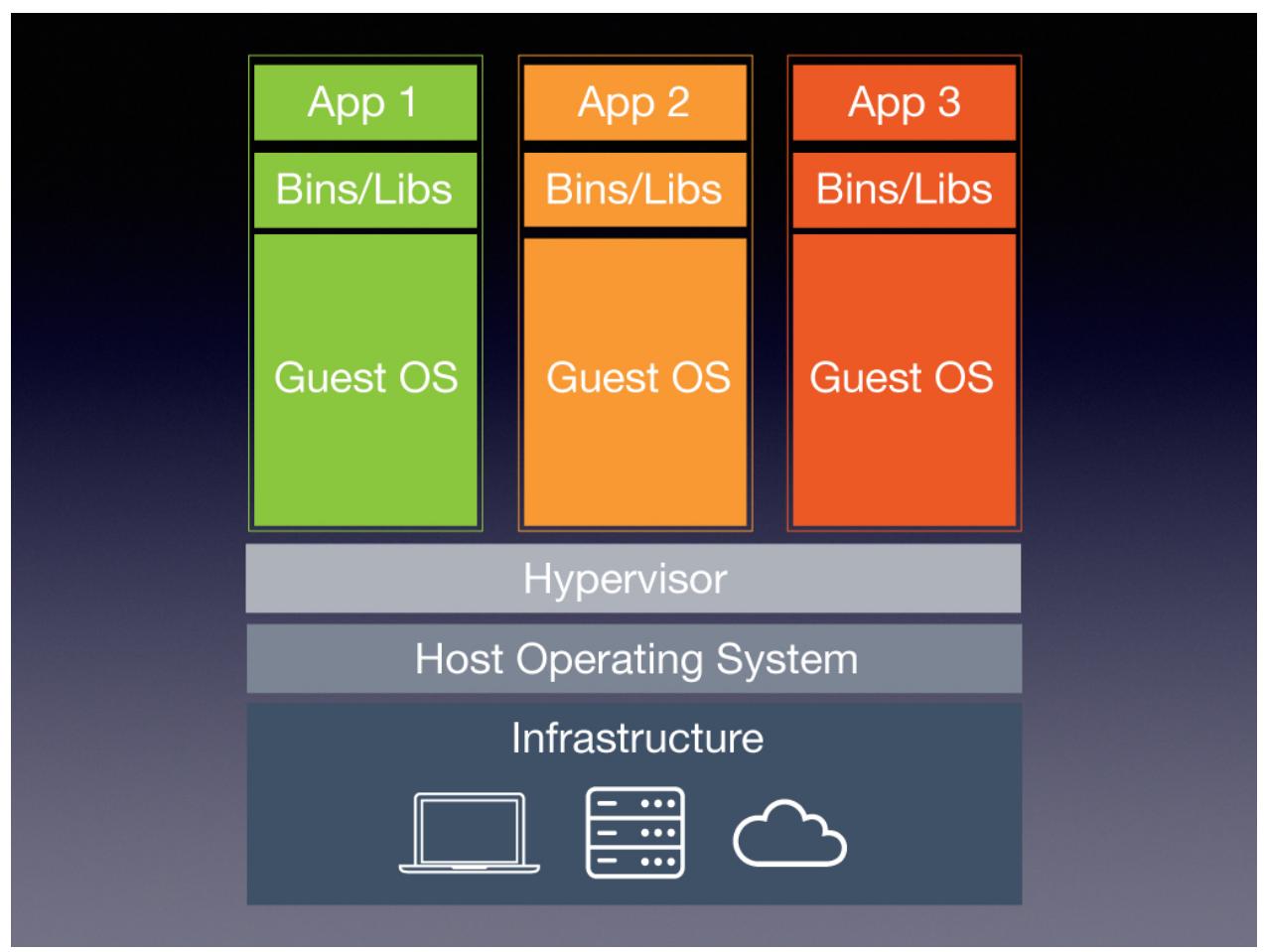
结构复杂的单体系统 > 微服务 > Docker 化 > 自动化

Coding 的复杂的网站系统也不是一天建成的，这是我们 Coding 的架构演进流程，我们的起初也是一个结构复杂的单体系统，后来才采用了各种策略去改进它。那么怎么评判一个系统复不复杂，个人觉得看两个指标，一个就是运维人员用多久时间可以把新的代码部署上线。比如说我之前所在的创业团队，每次部署都是晚上九点以后很少访问量的时候进行，部署当天晚上吃饭时还说半小时就差不多可以更新上线了。最后发现四五个小时过去了还没有上线。为什么呢？可能有一些前端的样式问题，有些配置文件没有做好。为什么会找出这些问题？很多情况下是因为，我们线上和线下状态的不统一。这是其中一个指标，还有一个指标就是，一个新同事来到公司以后，要多长时间可以把整个系统的开发环境部署起来。比如说我是一个后端程序员，得先要装一个虚拟机，Nginx 服务器，数据库，具体语言的编译环境，以及运行 npm install 来安装一些前端的代码库，这些操作，加上国内有奇葩的网络问题，我们通常会耗费半个到一个工作日之久。而用了 Coding-Job 来部署本地开发环境，这个时间可以降低到半个小时。一个复杂的单体系统会拉长启动时间，增加代码理解难度，测试也会变得更加困难。

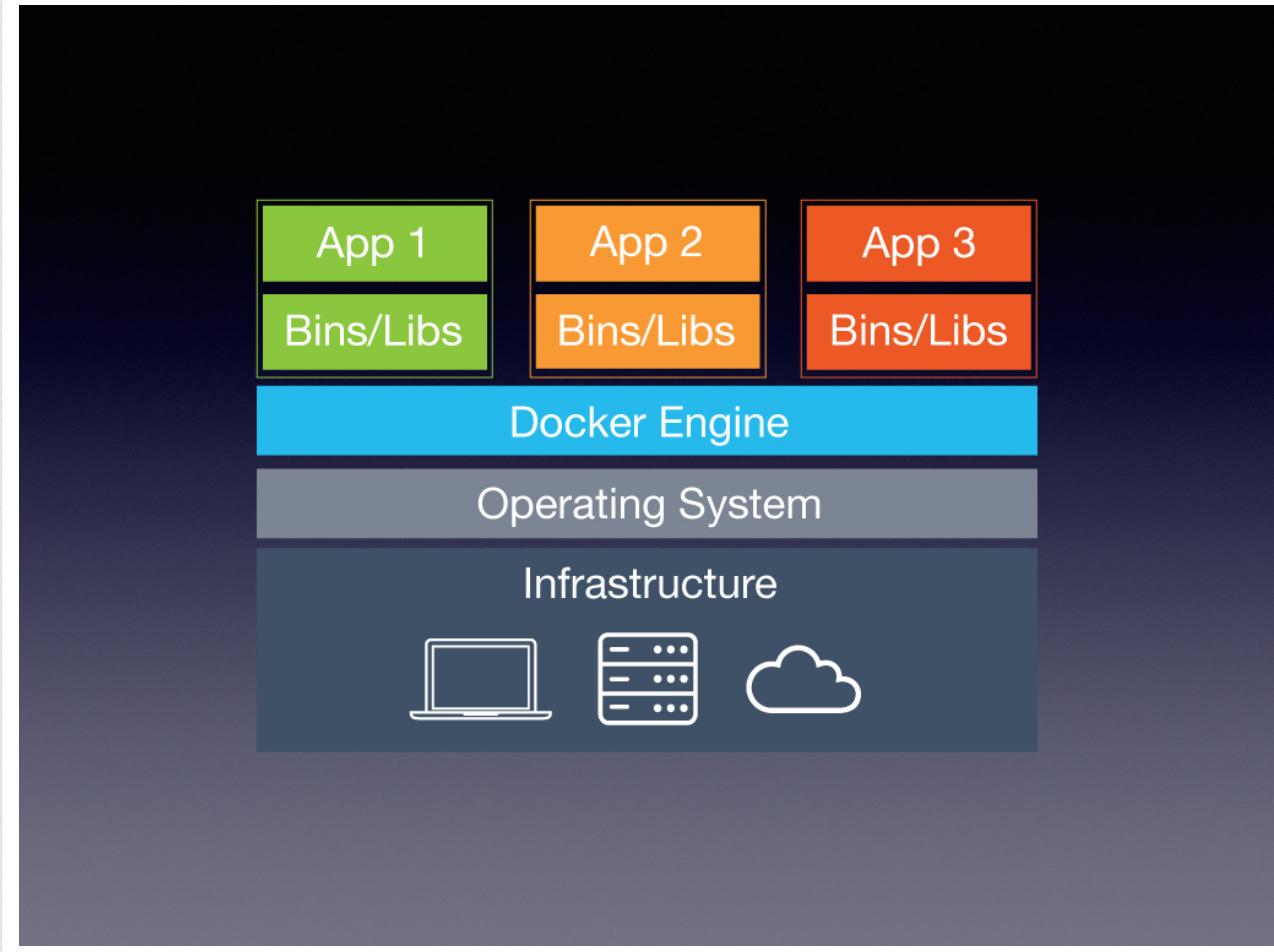
Coding 网站一开始是一个简单的 Java War 包，编译、打包、上传，再到上线，还算是一个比较简单的操作。随着业务的快速发展，比如说有短信功能，有邮件、短信推送，这些东西如何把它融合在一起，是一件比较棘手的事情。前两年微服务也比较火，我们也采用了微服务的架构，允许开发者用他最拿手的框架和语言，选一个微服务来做。比如说短信，短信模块写好了，接口文档写好，发给写 Coding 后台的同事，

然后一下功夫就完事儿了。

这种方式使我们的代码能够跟的上 Coding 迅猛的业务发展。但我们突然发现一个问题，就感觉全公司已经没有人会清楚的记得每个微服务是怎么编译、配置和启动的了。于是就想，有没有这样一种东西，跟黑盒子一样，可以让我们把一堆代码放到里面去。不用管黑盒子是怎么配置运行的，把它放到线上去，让它能跑起来，代码在里面怎么配置是盒子的事情，我外面环境脱胎换骨，也不影响盒子里面的正常运行。这个黑盒子，好像就可以解决我们的问题，而这，其实就是虚拟化技术。在 2015 年初 Docker 刚火起来的时候，我们研究了当时传统方案虚拟机和 Docker 技术，最后采用了 Docker。



图中是传统方案-虚拟机的架构图，我们在买了主机（Infrastructure）装了宿主机操作系统（Host Operating System）以后，想要实现虚拟化技术，就要安装一套虚拟机管理软件（Hypervisor），比如 VMware vSphere，它可以允许跑三个黑盒子，其实就是虚拟机，在三个虚拟机里面分别跑三个操作系统（GuestOS），而在这三个操作系统之上，我们才运行我们真正想要运行的东西，即 App1、App2、App3 这三个微服务。那么想回来，为什么我们要奇奇怪怪地启动三个虚拟机呢？为什么要将大量地物理资源耗费在可有可无的上层操作系统上面呢？



人们终于想清楚了这件事情，就开始着手研究轻量级的虚拟化技术。Linux 内核的命名空间（Name Space）和 控制组 Cgroups（Control Groups）等技术应运而生，实现了进程间命名空间的隔离，网络环境的隔离，和进程资源控制。Docker 正是依靠这些技术突然火热了起来。我们在用 Docker 跑微服务的时候，图中的虚拟机管理软件被替换成了一个 Docker Engine，每个 App 跑在 Docker 容器中，容器与宿主机共享一个操作系统，它能节省较多的物理资源，启动速度也由虚拟机的分钟级达到秒级，I/O 性能也接近直接运行与宿主机上。

说到命名空间，大家都知道 Linux 只会有一个 PID 为 1 的进程 init，有了命名空间以后，在每个进程命名空间里面，PID 列表是相互隔离的，在容器里面，也可以有一个 PID 为 1 的进程，这就实现了进程间命名空间的隔离。而控制组，它是可以做到对每个容器所占用的物理资源的控制，比如指定 CPU 的使用，块设备的读写速度。结合这两种技术，我们可以做到的是让 Docker 容器在保持一定性能的同时，尽可能地在隔离性上面接近虚拟机。



于是我们就将 Coding 的微服务一个一个地迁移到 Docker 内，以容器的方式部署运行，然后你会以为是这样一幅场景。



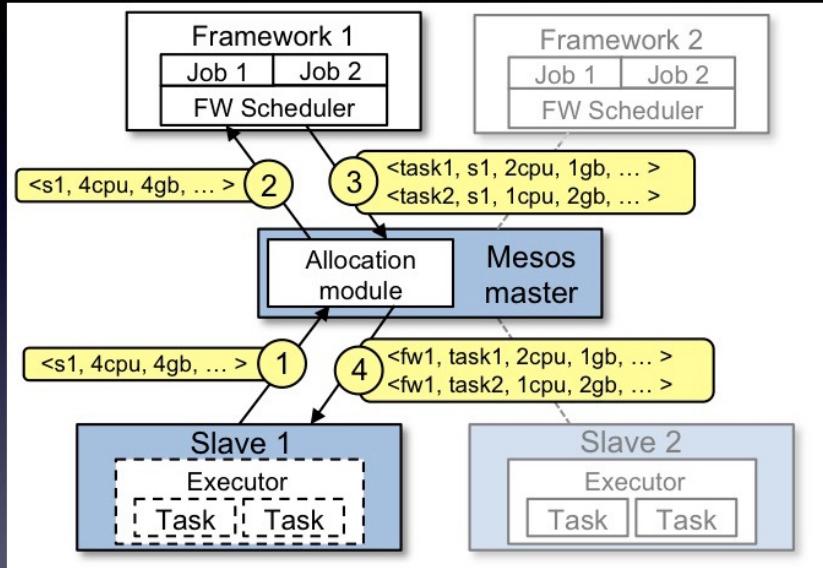
但事实上可能是这样的。在 Coding 我们的大大小小的微服务五十余个，就像鸡蛋不能放在一个篮子里一样，容器也不能被放在同一台云主机上面，而是整整齐齐地分开来放，不然怎么能叫分布式呢？我们的微服务，对于文件系统，对于彼此的网络还存在一些依赖性，像一个蜘蛛网一样串在一起，对于微服务所允许的主机位置和相应的主机配置都是有要求的。所以是需要一个中心化的东西去帮我们去存放每个服务的配置以及它们运行的代码甚至 Docker 镜像。而这个中心化的东西所做的事情就是编排，就好像我们在听音乐会上的时候，台上的指挥家一样，它告诉这台云主机做什么任务，以什么配置运行这个任务，用什么代码来执行它，然后再告诉另外一台机器，又做什么任务等等。



Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.

为此我们研究了两款业界比较火的两款开源容器管理框架，也就是这个中心化的东西。

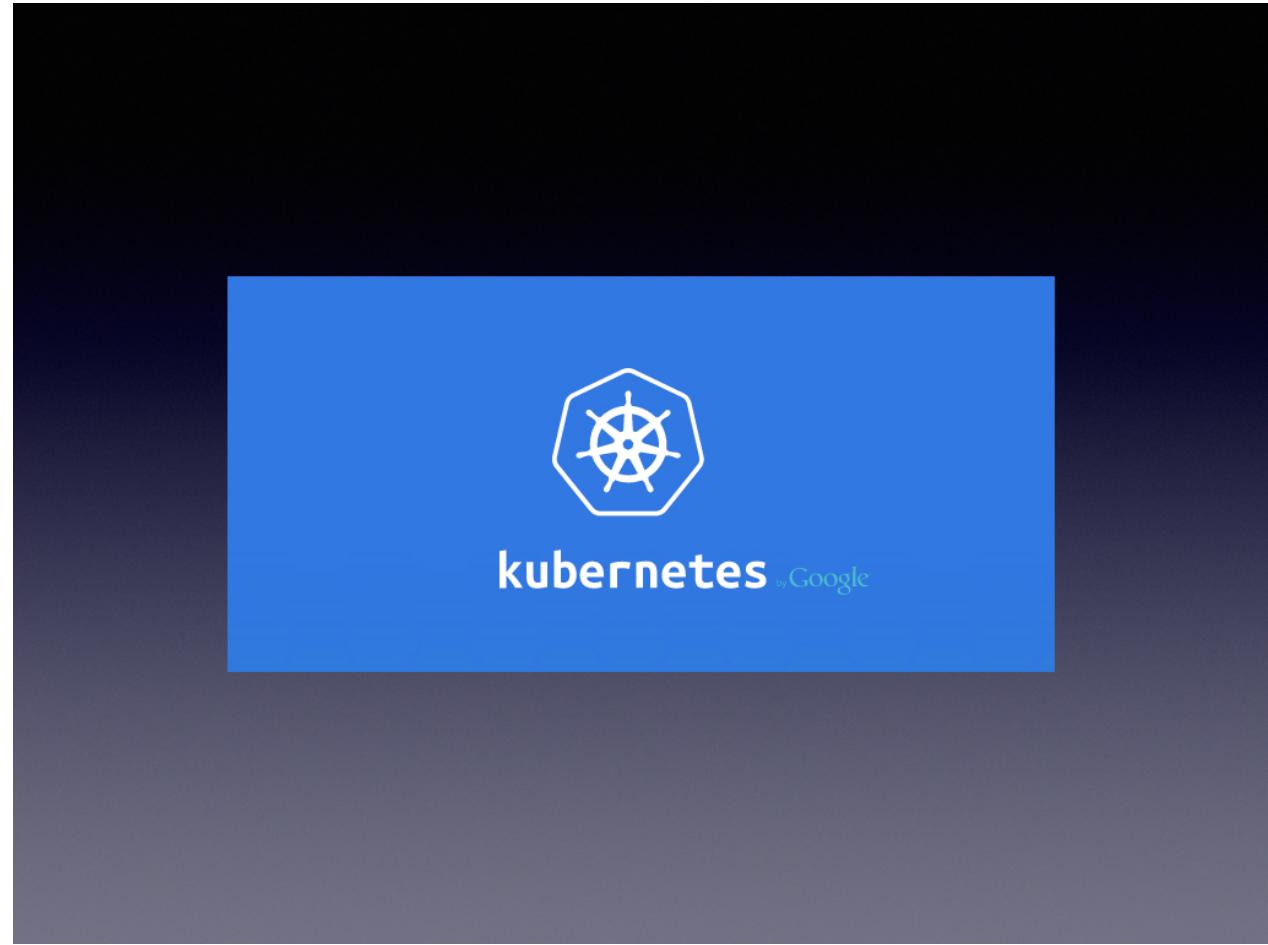
Apache Mesos 是一个用于集群的操作系统，它会把一个集群所具有的所有物理资源抽象成一台计算机供用户使用，比如你的集群里有一百台服务器，每台服务器一个 CPU，那直白的来说 Apache Mesos 能给你一台具有一个一百个 CPU 的电脑。



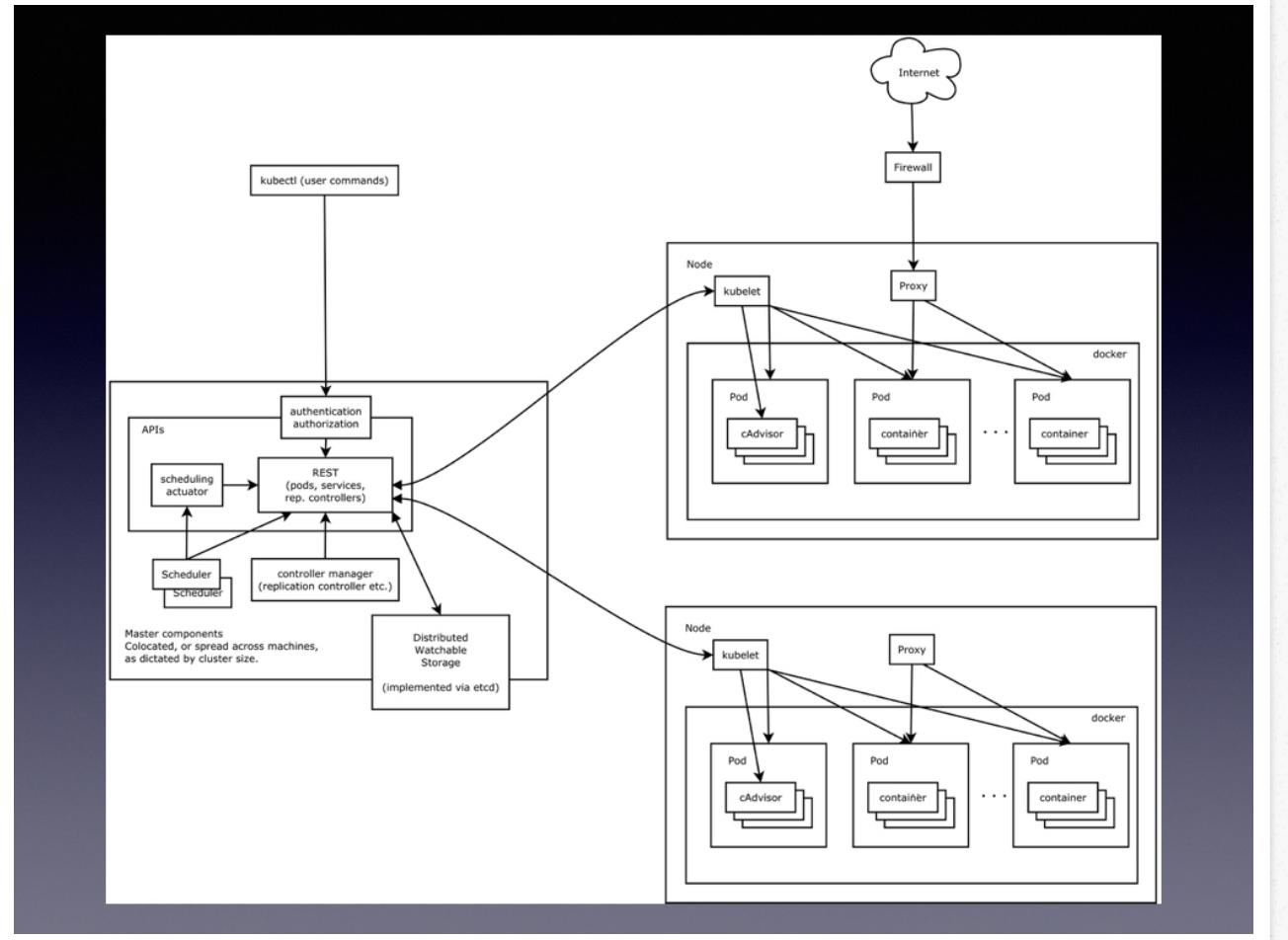
Resource Allocation

Mesos 做的比较好一点的是物理资源的分配。图中可以看到 Mesos 具有 Framework 的概念，它等同于我们通常所说的应用程序，每个 Framework 由调度器（Scheduler）和执行器（Executor）两部分组成。调度器负责调度任务，执行器负责执行任务。大家可以看到图中间部分有 Mesos Master，图下方有几个 Mesos Slave，Master 是管事的，相当于包工头，而 Slave 是奴隶，负责干活儿的。这些 Slave 其实就是我们的云主机，每一个 Slave 就是一台主机，这边可以看到一个物理资源分配的流程，首先 Slave1 发现它有 4GB 内存和 4CPU 的空闲物理资源，于是它向 Mesos Master 汇报，询问它有没有任务可以做，然后 Mesos Master 会询问当前可用的 Framework1 的调度器有没有可以分配给 Slave1 的活儿，调度器按照空闲物理资源取出了两个任务回应给 Mesos Master，然后 Mesos Master 又转发给 Slave1，Slave1 又开始干活了。当然此时大家会发现 Slave1 上面还有 1GB 内存和一个 CPU 是空闲的，那么它可以通过 Mesos Master 请求 Framework2 去请求任务来做。

所以 Mesos 能带给我们的的好处，一是高效，我们通过对主机的物理资源量化以及对任务的资源需求量化，使得多个任务能在同一台主机上运行，并充分利用其物理资源，二是可扩展性，Mesos 提供五十多种 Framework 帮助我们处理多种多样的任务。



另外一款名叫 Kubernetes 的 Docker 容器集群管理框架也特别火热，它借鉴了谷歌的 Borg，为容器化系统提供了资源调度，部署运行，服务发现等各种功能。它更加倾向于容器的编排，具有系统自愈功能，我们来看一下它的架构。



我先讲架构图，左边是一个控制的节点，右边是一台台的 Slave，我在左上角用 kubectl 提交一个作业，让 kubernetes 帮你把作业分配一个 Slave 上面去。在 Kubernetes 里面，调度任务的单位是 Pod，中文是豆荚。就像豆荚一样，里面是有很多豆子的，那这些豆子是什么呢？这些豆子其实是我们的 Docker 容器，这些容器共享一个豆荚，在 Kubernetes 里面就是一个共享容器组的概念。所有在一个 Pod 里面的 Docker 容器都是共享命名空间的，这解决了一些特殊场景的需求。因为使用 Docker 的最佳实践是在每一个容器里面只做一件事，不把 Docker 容器当做虚拟机来用。而这意味着有些时候，比如 B 进程需要监测（watch） A 进程的进程号（PID）或者和 A 进程进行 IPC 通信。如果是一个容器一个命名空间，这显然是不能直接实现的，而这就是 Pod 的应用场景。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

Pod Definition

那么我们现在来看一下一个 Pod 是怎么来定义，这边所示的 Pod Definition 是我们通过 kubectl 向 Kubernetes 提交作业的配置文件。在 Pod 中有多个 Container, 这里它定义了一个 Container 是 Nginx 以及 Nginx 的一些配置，例如镜像名和容器端口。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-controller
spec:
  replicas: 2
  # selector identifies the set of Pods that this
  # replication controller is responsible for managing
  selector:
    app: nginx
  # podTemplate defines the 'cookie cutter' used for creating
  # new pods when necessary
  template:
    metadata:
      labels:
        # Important: these labels need to match the selector above
        # The api server enforces this constraint.
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Replication Controller

接下来我们讲一下它的自愈系统，kubernetes 通过副本控制器（Replication Controller）来实现一个或者多个 Pod 的运行的控制。上图是我们向 Replication Controller 提交的配置文件，Template 字段代表了这个 Controller 使用的 Pod 的模板，而 Replicas 字段代表了我们希望 kubernetes 用这个 Pod 模板产生多少个 Pod 同时运行，这涉及到 kubernetes 里面一个叫理想状态（desired state）的概念。提交了这个配置文件以后，如果有任何一个 Pod 因为某种原因 Down 掉了，那么原先应该运行三个的副本只剩下两个，而 kubernetes 会想办法达到理想状态，因此它会启动一个新的副本，最终变成三个副本，这个就是 kubernetes 的自愈系统。

You'll never know...



那么听了 Mesos 的资源管理和 kubernetes 的自愈管理以后，我觉得它们做的已经相当成熟了，然而想要把它们投入到生产当中，可能还是会遇到一些坑的。比如想用 Mesos，那么首先要考虑我们 Coding 通常所运行的服务是一些常驻服务，不是批处理任务，所以需要额外安装 Marathon 这样的长期运行任务管理框架（Framework）来做到这件事情。若是在生产环境中遇到性能问题，怎么去调优也是一个棘手的事。而在 kubernetes 中，我们并不想让自愈系统替我们把服务恢复在任何一台机器上，目前 Coding 还是有着一些对机器和运行位置有着苛刻要求的微服务。除此之外，这两款框架也在迅速的开发迭代中，文档也仍需补充，所以投入到生产中，我们认为为时过早。我们考虑到使用原生的 Docker API 就可以做到大部分我们想要用到的功能，于是我们就开始自己开发了一套容器编排平台，Coding-Job。

Coding-Job

- 预定义所有任务配置
- Service/Job/Task

```
service: "core"
jobs:
  - name: "core-nginx"
    tasks:
      - name: "core-nginx"
        image: "core-nginx:48b58c1-dirty"
        run_on_host: "host-1"
        port: 0
        docker_config:
          docker_hostconfig:
            Binds:
              - "/data:/data/:rw"
  - name: "coding-email"
    tasks:
      - name: "coding-email"
```

Coding-Job 的任务配置分成三层，Service、Job 和 Task，图中的 Service 是核心（core）服务，说明它是被众多服务所依赖的。每个 Job 定义了一件要做的事情，而每个 Job 要具体被分配去做的时候，会细分为 Task，每个 Task 可以使用不同的配置，运行在不同的机器上。

Coding-Job CLI

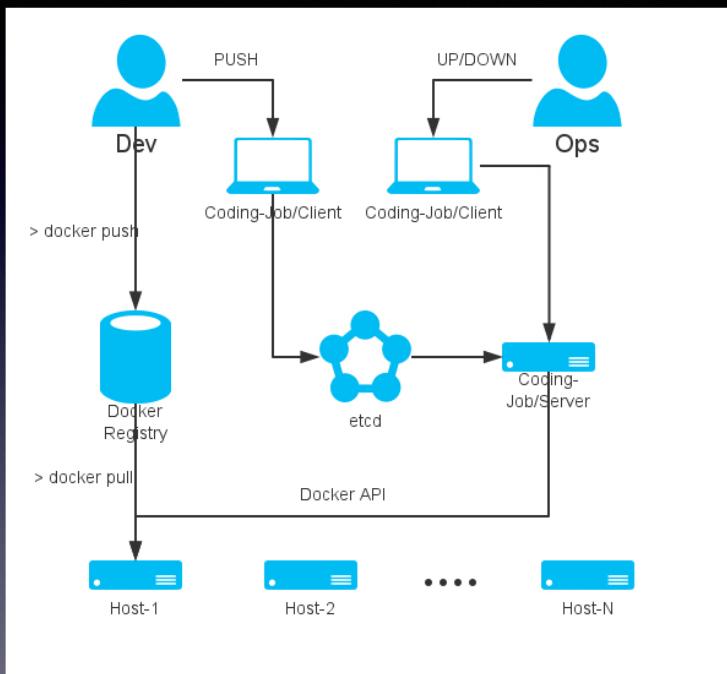
- UP / DOWN / UPDATE / RELOAD / PUSH / LIST

Coding-Job 是 C/S 架构的，在客户端，有命令行版的操作。分别是 Push（推送配置文件操作， UP/DOWN/UPDATE 这些对 Job 的启动、停止和按照配置更新 Job 和 List 功能列出所有的任务现在运行状况。

Coding-Job Web

- STATUS / INSPECT / LOG / HISTORY

此外 Coding-Job 的服务器端会展现一个 WebUI 界面，我们可以通过 WebUI 来获知每个容器的运行状态，通过 INSPECT 操作来提供容器的具体信息（与 docker inspect 一致），LOG 功能供查看容器 LOG，History 是显示每个 JOB 的历史容器的配置。此外，Coding-Job 服务端会定时请求各 Slave 系统资源使用及 Docker 运行数据，这些数据也会在 WebUI 上显示出来。



Coding-Job Topology

这是 Coding-Job 的使用架构图，Host-1、Host-2 这些是 Slave 机器。开发者在拉取最新的代码以后，用预先提供的打包命令生成一个 docker image 文件，通过 docker push 把镜像上传到私有 Docker Registry 中。同时开发者会根据最新的代码标签更新任务配置文件，通过 Coding-Job 客户端 PUSH 到 Coding-Job 服务端所使用的 etcd 中。这个 etcd 起到了存储任务配置信息的作用，由于它是可监测变化的存储（watchable storage），在收到新的任务配置信息后，Coding-Job 服务端就可以各种方式通知到运维人员（Ops）。运维人员如果确认要更新线上代码，调用 Coding-Job 客户端的 UP 或者 UPDATE 命令，就可以让 Slave 机器开始下载新的代码镜像来运行，于是更新就完成了。

- 一分钟发布新组件
- 五分钟启动整个 Coding
- 透明的线上环境

那么好处是什么呢？首先是，一分钟发布新组件（服务）不再是梦，在此基础上，历史容器的功能加上每次更新后任务配置提交到 Git 仓库中，允许我们追溯每个组件的历史线上版本。其次，我们可以利用它在公司内网环境内五分钟启动一个跟线上几乎一致的 Staging 环境，这个 Staging 可被用于新功能的内测。最后一个优点，是我们在使用 Coding-Job 的过程中，开发者不再对线上环境一头雾水，而是全透明的，可以知道生产环境是怎么组建的。在运维检查整站状态的时候，可以通过 WebUI 获得一些编排意见，甚至这个只读的 WebUI 可以开放到公网上，让所有人都可以看到 Coding 的服务提供状态。

References

- <http://mesos.apache.org/documentation/latest/architecture/>
- <https://github.com/kubernetes/kubernetes/blob/release-1.1/docs/design/architecture.md>

