

Kubernetes权威指南读书笔记

申恒恒 | 中科院云计算中心大数据研究院

Dec 7th, 2018

INTRODUCTION

- 基于容器技术的分布式架构的领先方案
 - 容器编排解决方案
 - 容器编排
 - Borg
 - Google 内部使用的大规模集群管理系统
 - Kubernetes可以看作是Borg的一个开源版本
 - k8s的贡献在于简化或者直接省略底层复杂的代码和功能模块，是业务开发人员更加关注业务本身，提升了开发效率，降低了后期的运维难度和运维成本.
 - 此外，Kubernetes平台对于现有的编程语言、编程框架、中间件没有任何侵入性，对现有的系统易于改造升级并迁移到现有的系统中.
- 2015.04 开源

集群管理能力：多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制，以及多粒度的资源配额管理能力.



Borg Motivation





Refer: john wikes, Cluster management at Google with Borg - coping with scale, 2016, goto; conference

Image by Connie Zhou

User View

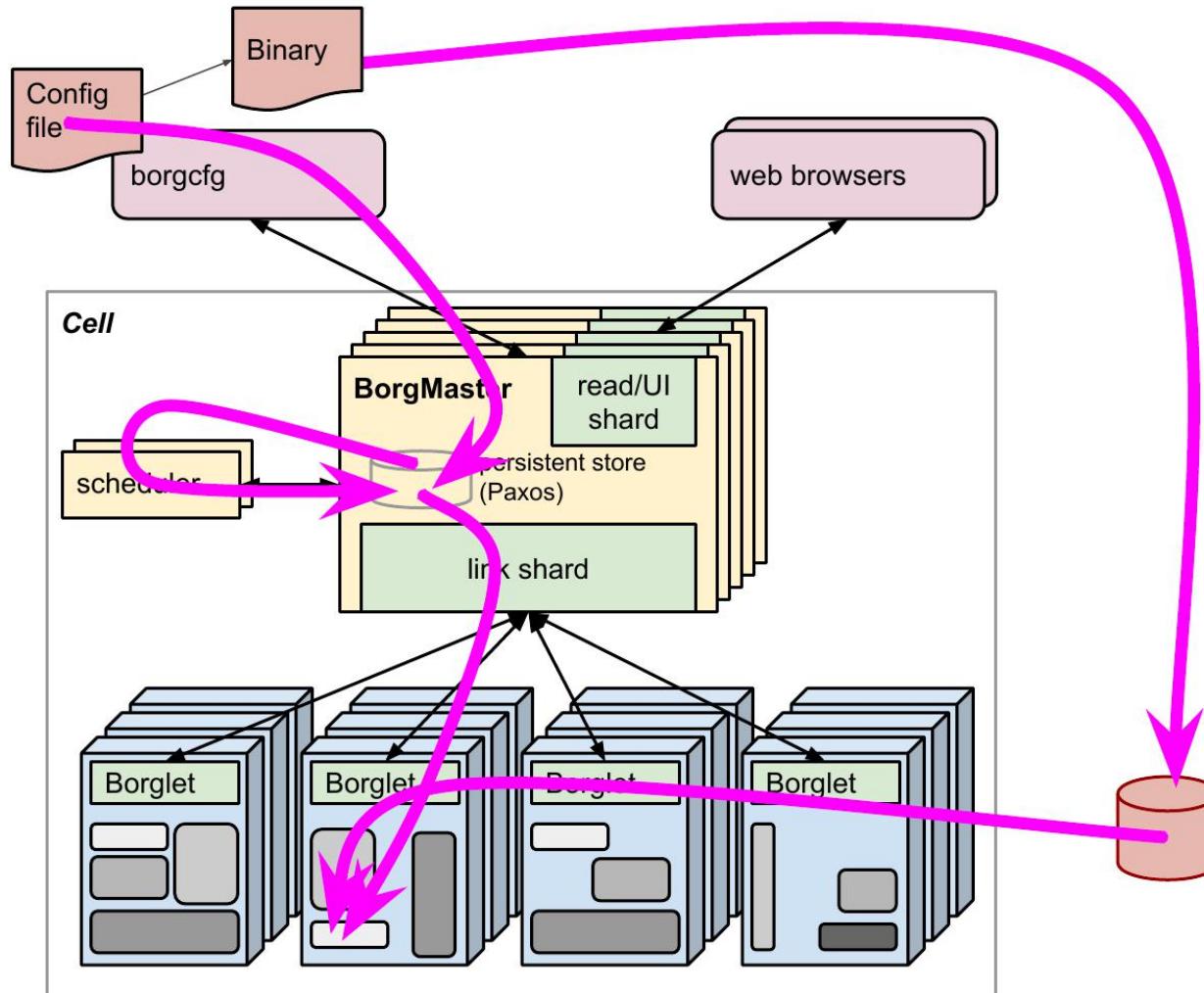
```
job hello_world = {
    runtime = { cell = 'ic' }                      // Cell (cluster) to run in
    binary = '.../hello_world_webserver'           // Program to run
    args = { port = '%port%' }                     // Command line parameters
    requirements = {                             // Resource requirements (optional)
        ram = 100M
        disk = 100M
        cpu = 0.1
    }
    replicas = 10000      // Number of tasks
}
```



Refer to: john wikes, Cluster management at Google with Borg - coping with scale, 2016, goto; conference

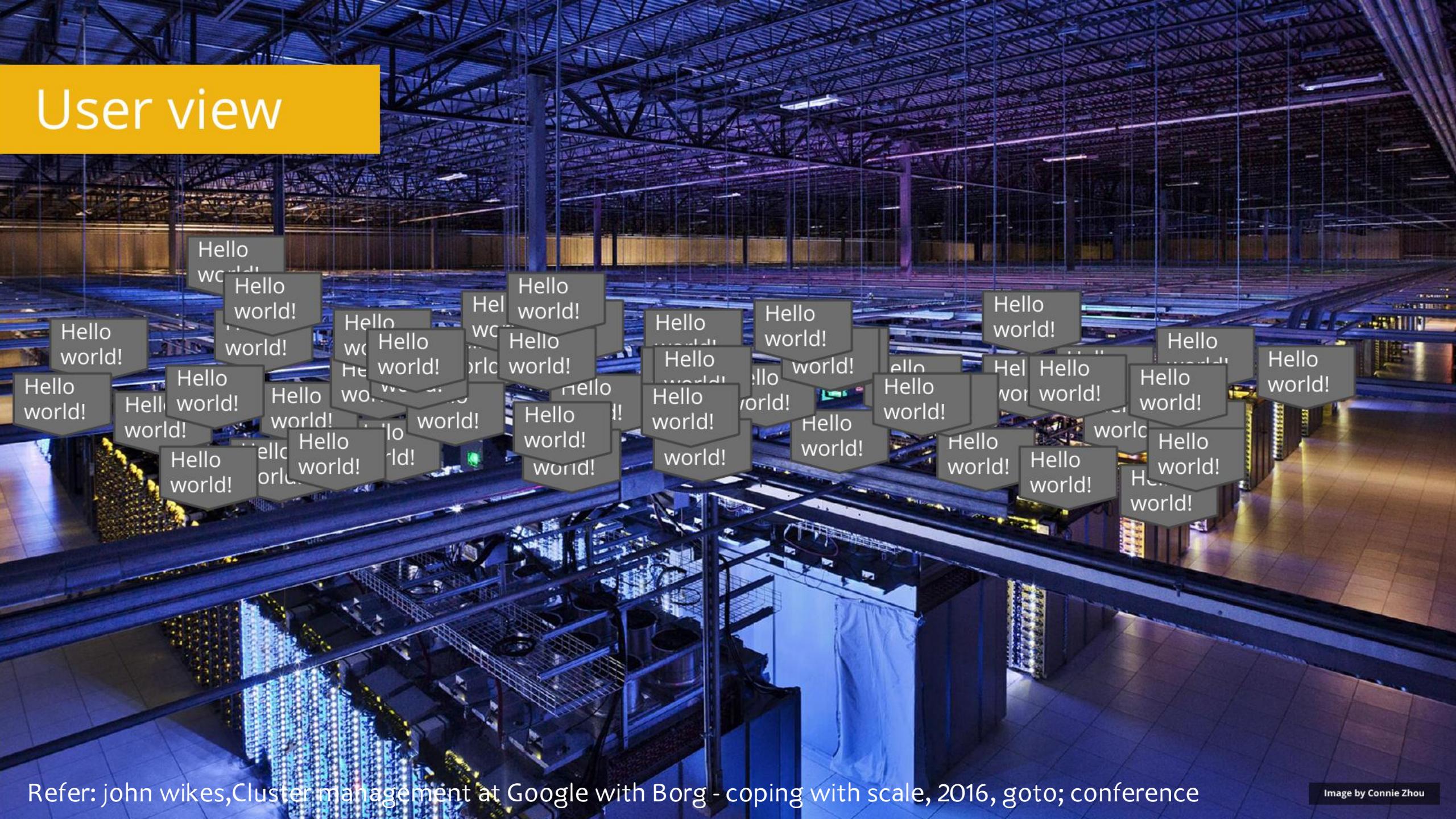
User View

What just happened??



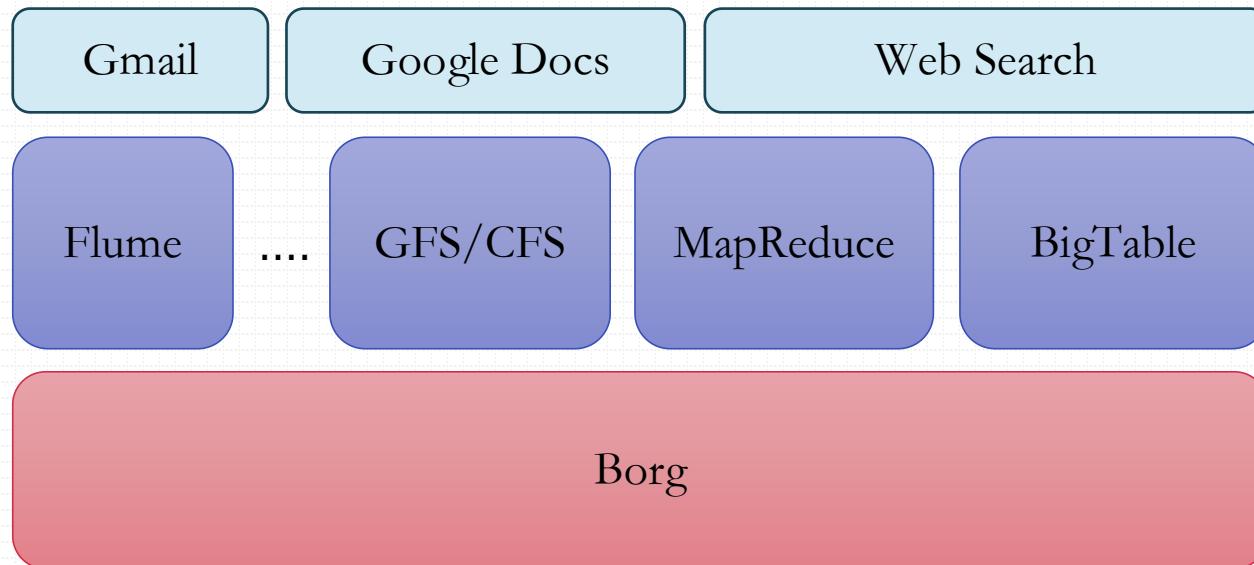
Refer to: john wikes, Cluster management at Google with Borg - coping with scale, 2016, goto; conference

User view

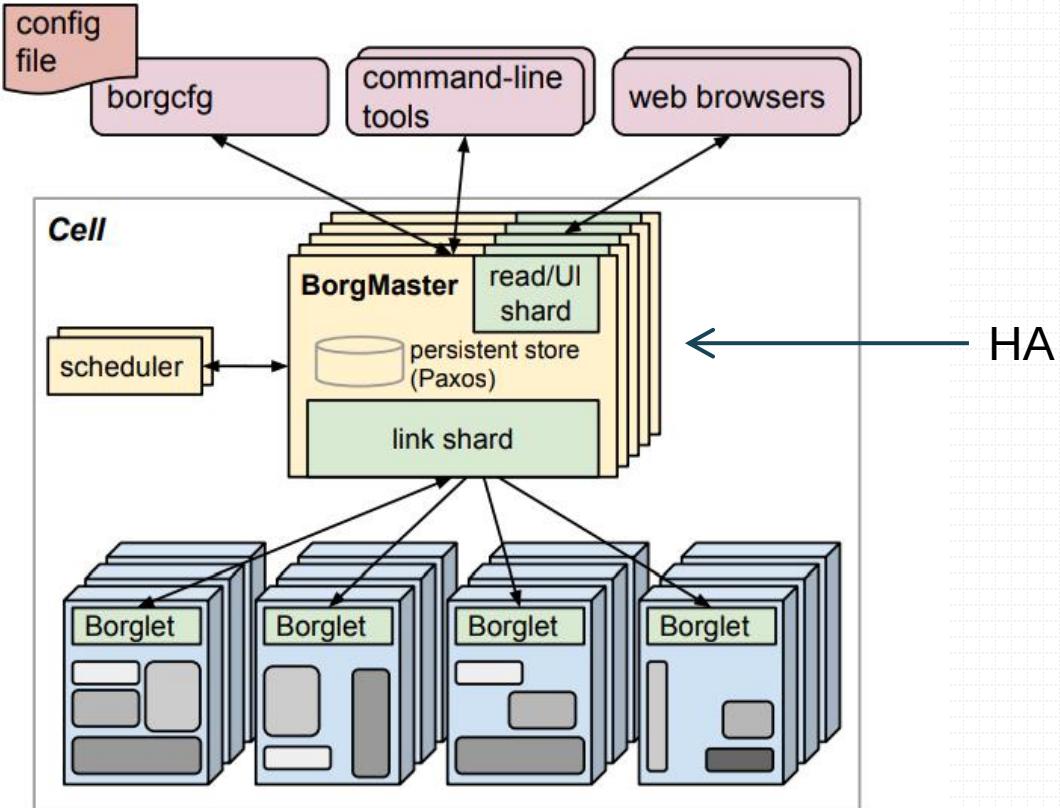


Paper review: Large-scale cluster management at Google with Borg

- Borg的三个重要的特性：
 - 隐藏了资源管理和故障处理的细节，从而使用户只关注应用开发；
 - 具有非常高的可靠性和可用性，并支持执行相同操作的应用程序；
 - 有效地在数万台机器上运行工作负载.
- Google 内部的使用情况



The high-level Architecture of Borg



- 每个集群叫一个Cell，分别对应一个BorgMaster.
 - 左图中含有5个BorgMaster，为了HA
 - 这5个BorgMaster依赖于Paxos的存储
- BorgMaster管理很多机器.
- 每个机器上会有一个Borglet, BorgMaster定期向Borglet沟通，“需要在这个机器上做什么”、“需要安装什么软件”
 - Borglet不主动向BorgMaster沟通，防止BorgMaster因为请求过多宕掉.
 - BorgMaster跟Borglet沟通的桥梁中间加了一层link shard，很大一个作用是只把Borglet的变化传给BorgMaster，这样可以减少沟通的成本和BorgMaster处理的成本。



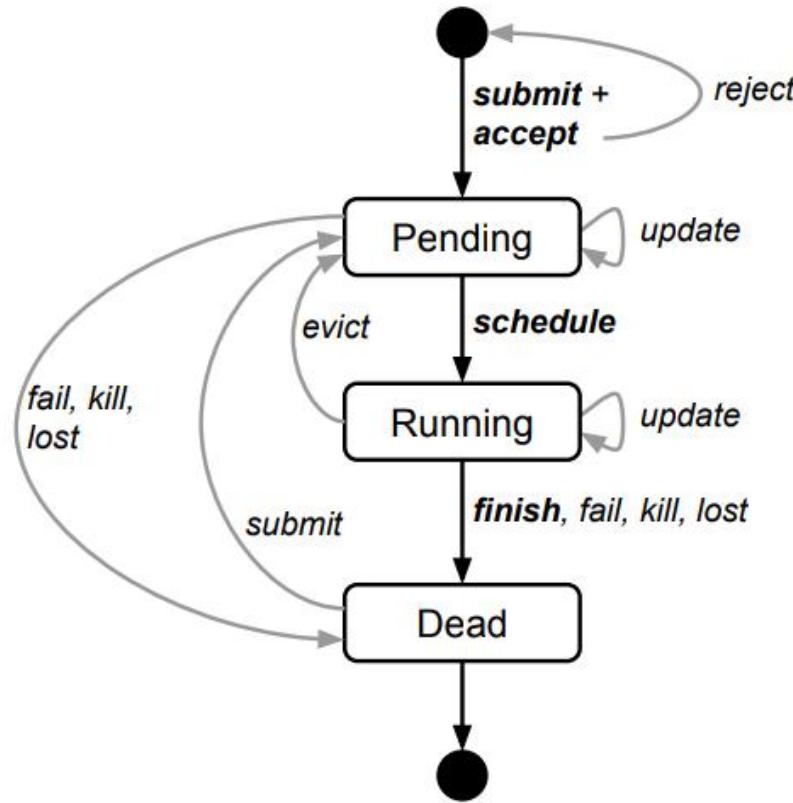
Components in Borg

- 单元 (Cell) —— 将多个机器的集合视为一个单元。单元通常包括1万台服务器，但如果有必要的话也可以增加这个数字，它们各自具有不同的CPU、内存、磁盘容量等等。
- 集群 —— 一般来说包含了一个大型单元，有时也会包含一些用于特定目的的小单元，其中有些单元可以用做测试。一个集群通常来说限制在一个数据中心大楼里，集群中的所有机器都是通过高性能的网络进行连接的。一个网站可以包含多个大楼和集群。
- Job —— 是一种在单元的边界之内进行执行的活动。这些job可以附加一些需求信息，例如CPU、OS、公开的IP等等。Job之间可以互相通信，用户或监控job也可以通过RPC方式向某个job发送命令。
- Task —— 一个job可以一个或多个任务组成，这些任务在同一个可执行进行中运行。这些任务通常是直接在硬件上运行的，而不是在虚拟环境中运行，以避免虚拟化的成本。任务的程序是静态链接的，以避免在运行时进行动态链接。
- 分配额 (alloc) —— 专门为一个或多个任务所保留的机器资源集。分配额能够与运行于其上的任务一起被转移到一个不同的机器上。一个分配额集表示为某个job保留的资源，并且分布在多台机器上。
- Borglet —— 一个运行在每台机器上的代理。
- BorgMaster —— 一个控制器进程，它在单元的级别上运行，并保存着所有Borglet上的状态数据。BorgMaster将job发送到某个队列中以执行。BorgMaster和它的数据将会进行五次复制，数据将被持久化在一个Paxos存储系统中。所有的BorgMaster中有一个领导者。
- 调度器 —— 对队列进行监控，并根据每个机器的可用资源情况对job进行调度。



Task in Borg

- Borg Job 可以限制 Task 运行在满足特定条件的 Machine
 - prod job (例如 Gmail、GoogleDocs、BigTable 之类的长期运行服务，这些服务的响应延迟很短，最多几百毫秒)
 - batch job (批处理作业)
- Task 的生命周期



Name	Owner
Task	Task
Task	Task

Job Structure

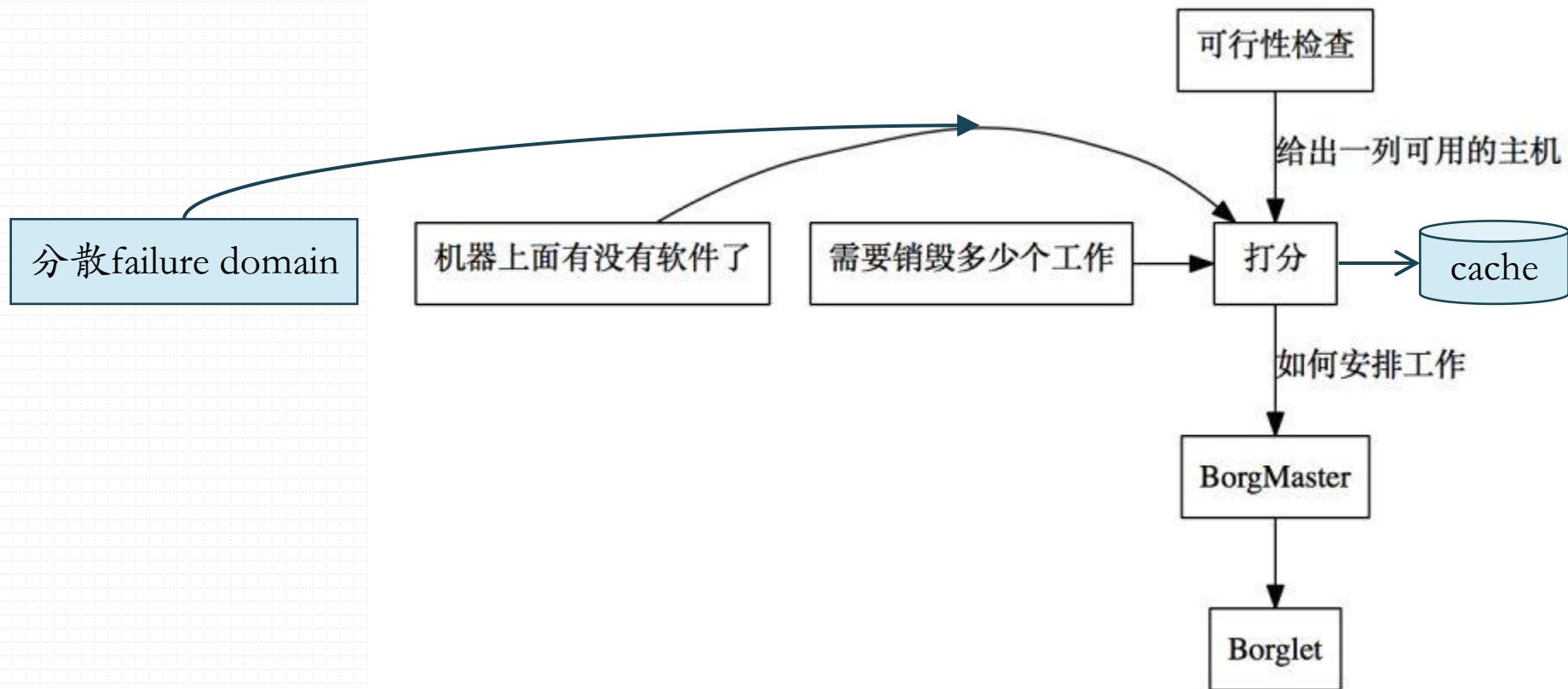


Borg中的一些概念

- 资源配置 (alloc) : 每个工作可以告知系统需要给我预留多少资源，这样系统可以确保不会太过激进的去叠加
- 优先级 (priority) : 可以告诉borg你的工作的优先级，高优先级的程序可以踢走低优先级的程序
- 整体工作配额 (quota) : 这个部分是在工作发送给borg的时候决定要不要接受这个工作，具体quota是在capacity planning的时候做的，是一个项目资源占比的问题而不是一个软件问题。
- 如何发送工作给Borg: 每个程序发现Borg是通过Chubby里面的文件的，文件会写明每个 BorgMaster对应的cell跟它的网络信息。之后就是一般的RPC。



Borg 中的调度



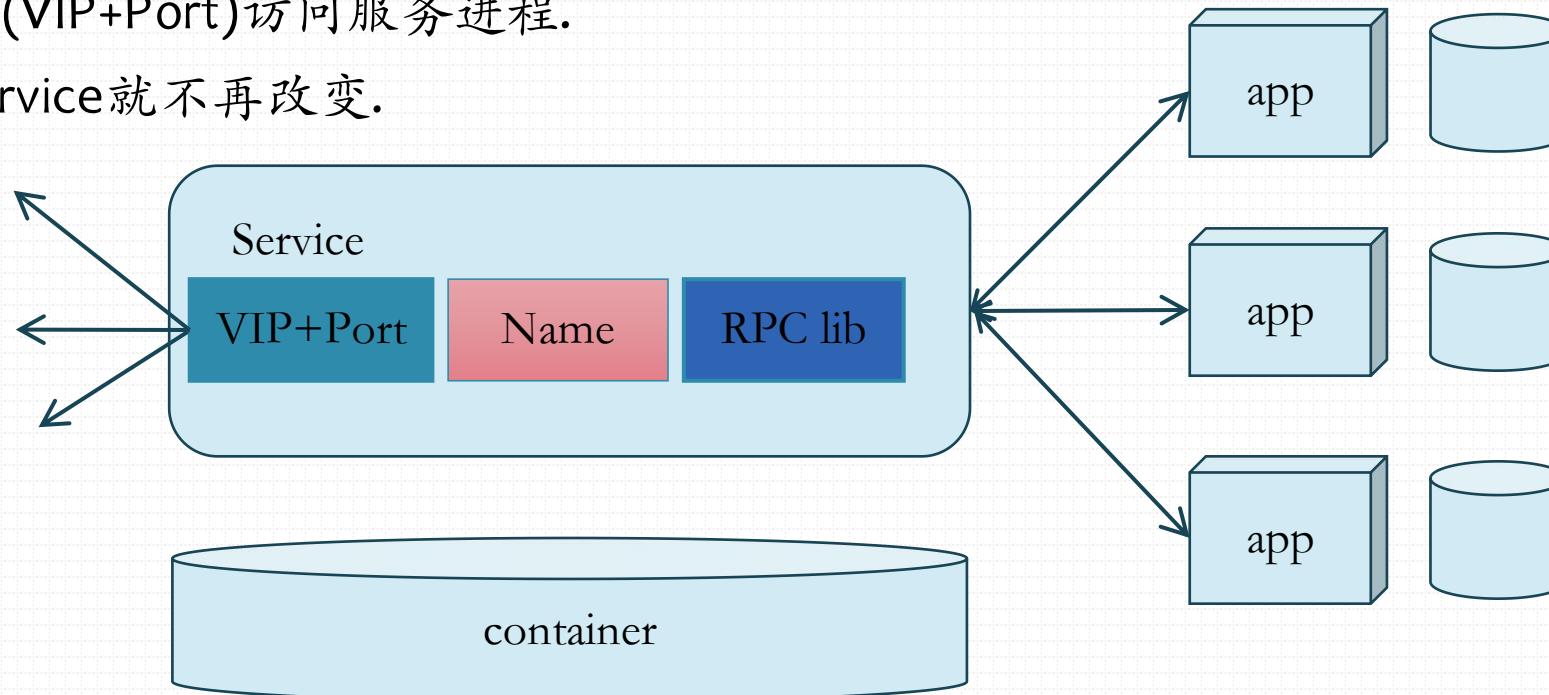
Borg 总结

- Borg的最基础单位是job，没有办法在job下面做更复杂的分配，所以很多内部用户需要hack一些奇怪的东西来表达topology。这个在Kubernetes里面解决了，可以用label。
- Borg假设一个host只能有一个IP，这样导致Borg会把port当成一个资源来分配。Kubernetes里面可以有多个IP了。
- Borg为了大用户做了很多优化，所有API非常复杂，但是一般用户很多API根本用不到。这个Kubernetes做的还是不错的。
- 资源配置单位是非常重要的，Kubernetes里面叫Pod
- 集群控制不只是task management，Kubernetes支持服务的load balancing跟naming
- 优良的debugging工具是非常重要的，Kubernetes基本复制了所有Borg的debugging能力
- BorgMaster就跟整个集群的kernel一样，未来需要做很多kernel能做而Borg不能做的事情



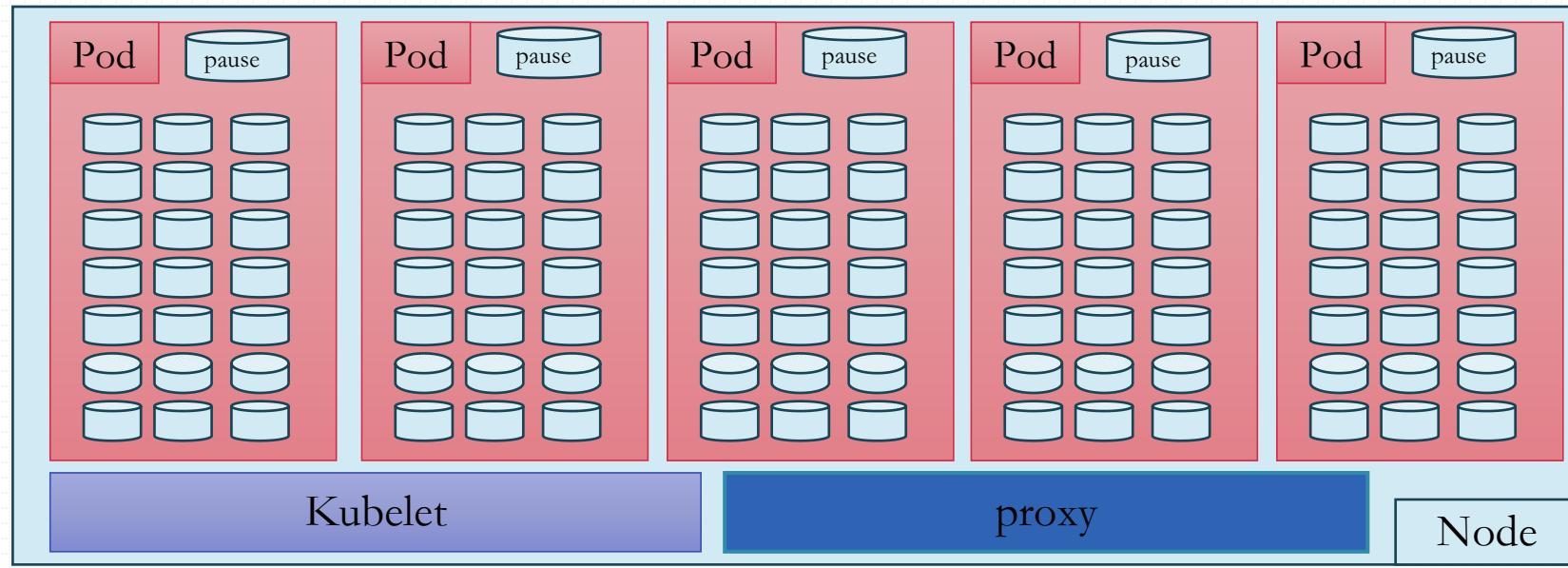
kubernets 基本知识

- Service(分布式集群架构的核心)
 - 唯一名字(比如: mysql-service)
 - 虚拟ip (Service IP、VIP或者Cluster IP) 和端口号 → (VIP+Port)
 - 具有RPC功能
 - 被映射到具有一系列功能的容器应用上.
- 通过Socket (VIP+Port)访问服务进程.
- 一旦创建Service就不再改变.



Service

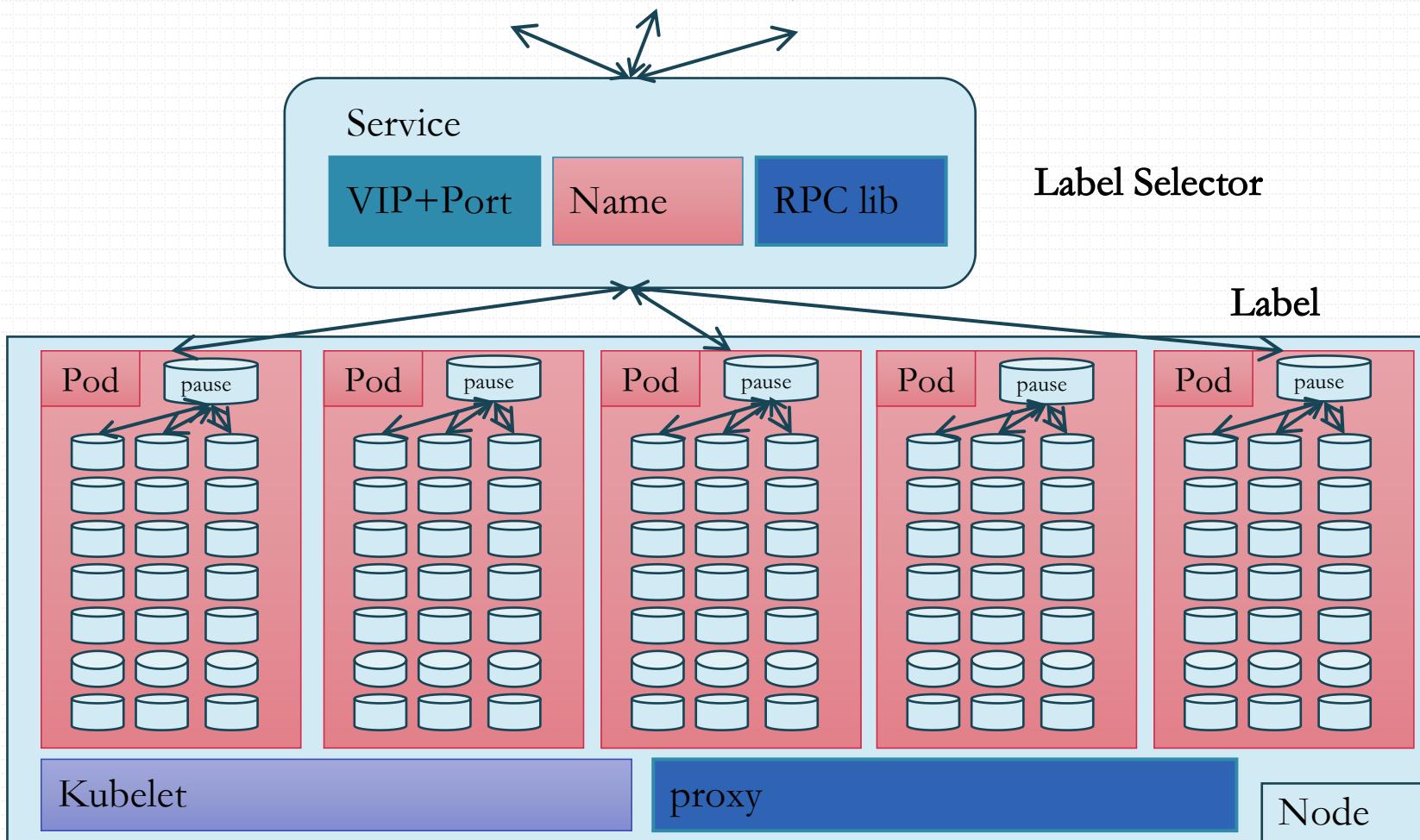
- 容器提供隔离功能，意味着将为Service提供服务的这组进程放入容器中.
- 引入Pod对象
 - 将每个服务进程包装到Pod中，使其成为Pod中的容器；
 - 怎么建立Pod与Service之间的联系？
 - Pod → Label（标签）、Service → Label Selector（标签选择器）
 - 比如：运行MySQL的Pod贴上name=mysql的标签，相应的Service的标签选择器的选择条件为name=mysql
- Pod（最小的运行单元）



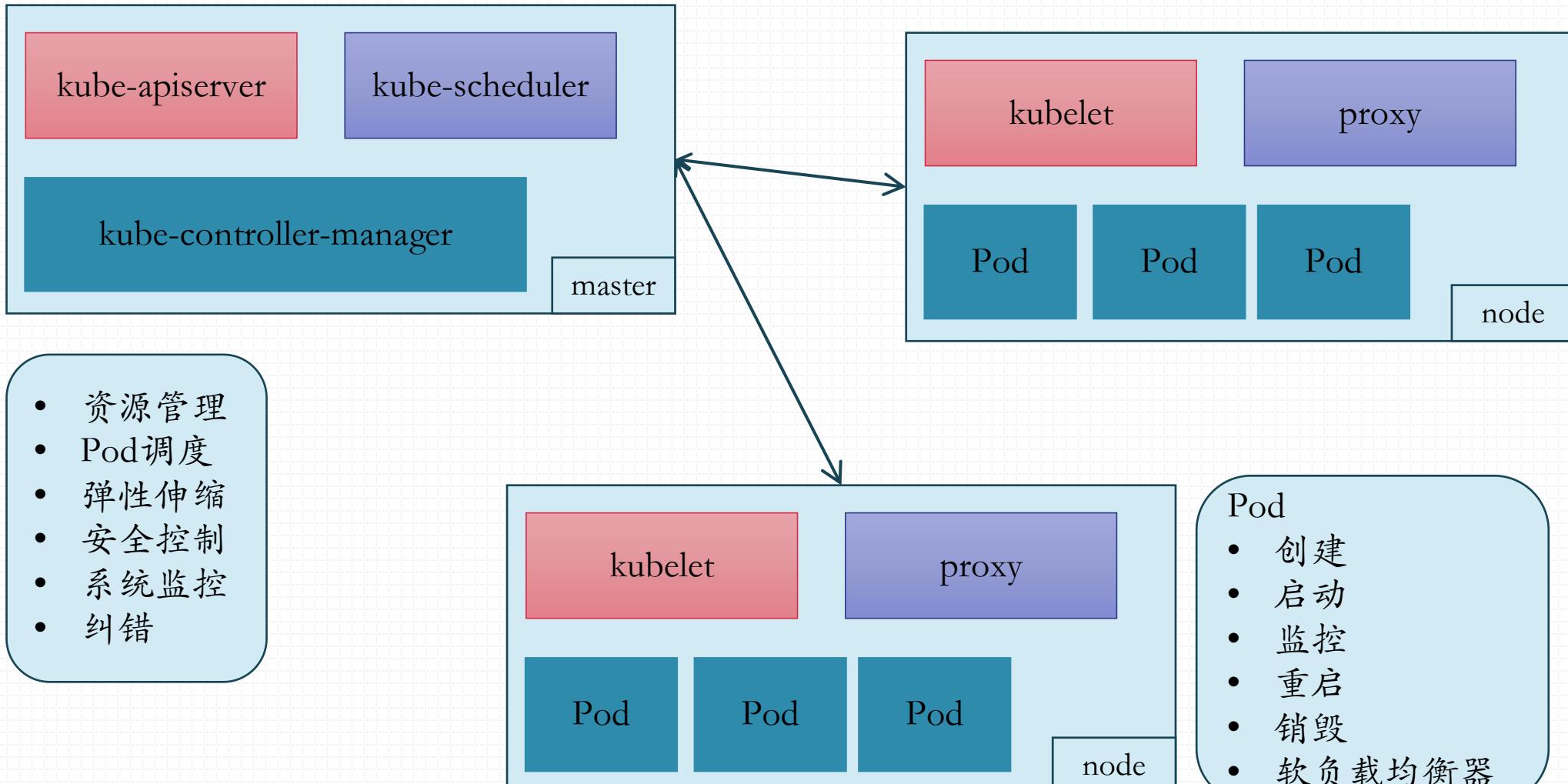
Pod + Service

- Pause 容器

- 同一组Pod内的业务容器共享组内Pause容器的网络栈和Volume挂载卷
- 设计：将一组密切相关的服务进程放入同一个Pod中



Master/Node Architecture



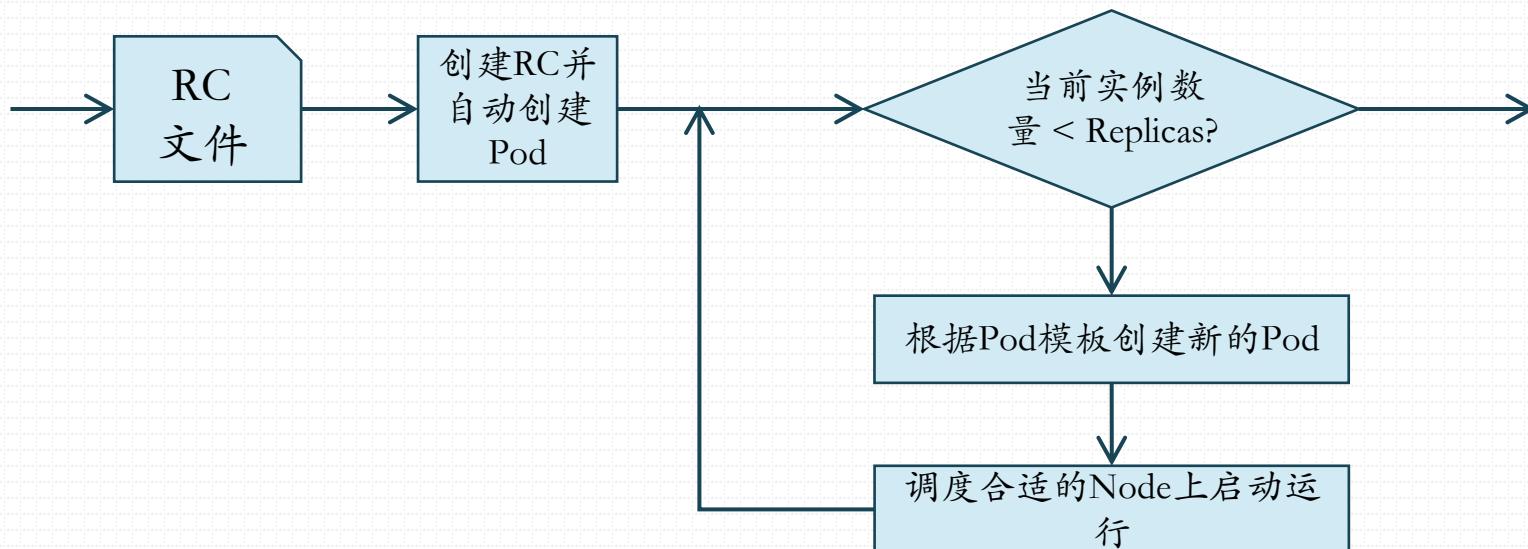
服务扩容/服务升级

- 服务扩容涉及以下：
 - 资源分配
 - 实例部署
 - 启动
- 传统的IT行业解决此类问题：
 - 人工手动解决（运维压力）
 - 索性不升级
- 在Kubernetes中，引入RC（Replication Controller）
 - 只需要在service关联的Pod上创建一个RC即可完成弹性扩容和升级任务

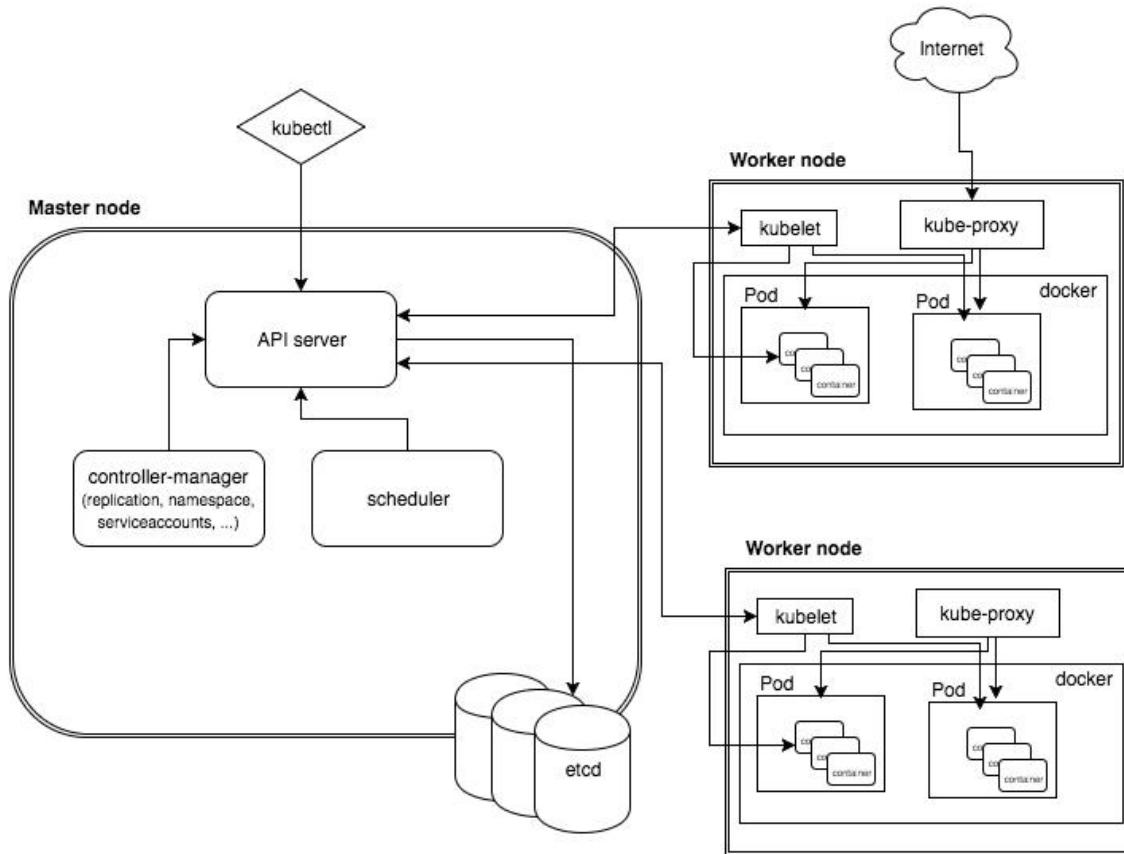


Replication Controller

- RC 定义文件
 - 目标Pod的定义
 - 目标Pod需要运行的副本数量 (Replicas)
 - 要监控的目标Pod的标签 (Label)



测试环境安装 (centos7)



(1) 关闭 CentOS 自带的防火墙服务:

```
# systemctl disable firewalld  
# systemctl stop firewalld
```

(2) 安装 etcd 和 Kubernetes 软件 (会自动安装 Docker 软件):

```
# yum install -y etcd kubernetes
```

(3) 安装好软件后, 修改两个配置文件 (其他配置文件使用系统默认的配置参数即可)。

① Docker 配置文件为 /etc/sysconfig/docker, 其中 OPTIONS 的内容设置为:

```
OPTIONS='--selinux-enabled=false --insecure-registry gcr.io'
```

② Kubernetes apiserver 配置文件为 /etc/kubernetes/apiserver, 把 --admission_control 参数中的 ServiceAccount 删除。

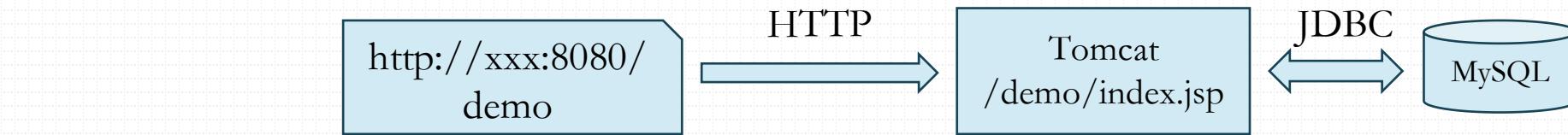
(4) 按顺序启动所有的服务:

```
# systemctl start etcd  
# systemctl start docker  
# systemctl start kube-apiserver  
# systemctl start kube-controller-manager  
# systemctl start kube-scheduler  
# systemctl start kubelet  
# systemctl start kube-proxy
```

至此, 一个单机版的 Kubernetes 集群环境就安装启动完成了。



From `Hello World` ...



```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
    app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "123456"
```

副本控制器 RC
RC 的名称，全局唯一
Pod 副本期待数量
符合目标的 Pod 拥有此标签
根据此模板创建 Pod 的副本（实例）
Pod 副本拥有的标签，对应 RC 的 Selector
Pod 内容器的定义部分
容器的名称
容器对应的 Docker Image
容器暴露的端口号
注入到容器内的环境变量

图 1.2 RC 的定义和解说图

- **kind属性**: 表明资源对象的类型
- **spec节**: 表示RC的相关属性的定义
 - spec.selector是RC的pod标签选择器
 - spec.replicas表示运行的Pod的实例数量
- **spec.template节**: 表示RC定义的Pod实例模板
 - 若当前运行的Pod实例<replicas，则按照定义的Pod模板创建
- **spec.template.metadata.labels**: 指定了当前Pod的标签，这里的labels必须要和spec.selector一样。



```
[root@bigdata example]# kubectl create -f mysql-rc.yaml
replicationcontroller "mysql" created
```

```
[root@bigdata example]# kubectl get rc
```

NAME	DESIRED	CURRENT	READY	AGE
mysql	1	1	1	26m

创建的RC名字
metadata.name

期望的副本数
spec.replicas

创建完成的副本数
spec.replicas

```
[root@bigdata example]# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-ck1n1	1/1	Running	0	3m

Pod实例的名字

Pod的状态



创建 Service

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
    - port: 3306
  selector:
    app: mysql
```

表明是 Kubernetes Service
Service 的全局唯一名称
Service 提供服务的端口号
Service 对应的 Pod 拥有这里定义的标签

图 1.3 Service 的定义和解说图

- **spec.selector:** 确定那些Pod副本实例对应到本服务.
- **spec.ports:** 表示开放那些虚端口
 - 比如: **port**属性定义了Service的虚端口

```
[root@bigdata example]# kubectl create -f mysql-svc.yaml
service "mysql" created
```

```
[root@bigdata example]# kubectl get svc
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
mysql     10.254.56.251  <none>        3306/TCP  6m
```

↑
VIP



```
[root@bigdata example]# kubectl describe svc mysql
Name:           mysql
Namespace:      default
Labels:          <none>
Selector:        app=mysql
Type:            ClusterIP
IP:              10.254.56.251
Port:            <unset> 3306/TCP
Endpoints:      172.17.0.2:3306
Session Affinity: None
No events.
```



启动tomcat应用

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myweb
spec:
  replicas: 5
  selector:
    app: myweb
  template:
    metadata:
      labels:
        app: myweb
    spec:
      containers:
        - name: myweb
          image: kubeguide/tomcat-app:v1
          ports:
            - containerPort: 8080
          env:
            - name: MYSQL_SERVICE_HOST
              value: 'mysql'
            - name: MYSQL_SERVICE_PORT
              value: '3306'
```

RC

```
apiVersion: v1
kind: Service
metadata:
  name: myweb
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 30001
  selector:
    app: myweb
```

Service

- 此Service采用了NodePort的外网访问方式
 - 可以通过30001端口访问到myweb（对应虚端口8080）

```
[root@bigdata example]# kubectl create -f myweb-rc.yaml
replicationcontroller "myweb" created
```

```
[root@bigdata example]# kubectl create -f myweb-svc.yaml
service "myweb" created
```

```
[root@bigdata example]# kubectl get services
NAME           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes     10.254.0.1     <none>          443/TCP       1h
mysql          10.254.56.251   <none>          3306/TCP      32m
myweb          10.254.146.101   <nodes>        8080:30001/TCP 8m
```

```
[root@bigdata example]# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
mysql-ck1n1    1/1     Running   0          1h
myweb-2qgjc    1/1     Running   0          12m
myweb-bqvfn   1/1     Running   0          12m
myweb-khz4t    1/1     Running   0          12m
myweb-pffvb    1/1     Running   0          12m
myweb-tp6vq    1/1     Running   0          12m
```



Let's Test

```
[root@bigdata example]# kubectl describe services myweb
Name:           myweb
Namespace:      default
Labels:         <none>
Selector:       app=myweb
Type:          NodePort
IP:            10.254.146.101
Port:          <unset> 8080/TCP
NodePort:       <unset> 30001/TCP
Endpoints:     172.17.0.3:8080,172.17.0.4:8080,172.17.0.5:8080 + 2 more...
Session Affinity: None
No events.
[root@bigdata example]# curl http://172.17.0.3:8080/
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
    <link href="favicon.ico" rel="icon" type="image/x-icon" />
    <link href="favicon.ico" rel="shortcut icon" type="image/x-icon" />
    <link href="tomcat.css" rel="stylesheet" type="text/css" />
  </head>

  <body>
    <div id="wrapper">
      <div id="navigation" class="curved container">
        <span id="nav-home"><a href="http://tomcat.apache.org/">Home</a></span>
```

The screenshot shows a web page with the title "Congratulations!!". Below it is a table with two columns: "Name" and "Level(Score)". The table contains six rows, each with a name and a score of 100. An "Add..." button is located above the table.

Name	Level(Score)
google	100
docker	100
teacher	100
PE	100
our team	100
me	100

图 1.4 通过浏览器访问 Tomcat 应用



DEMO



总结

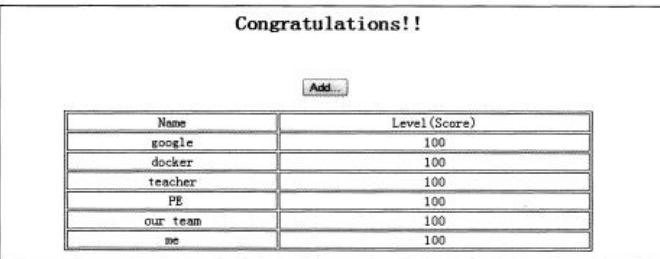
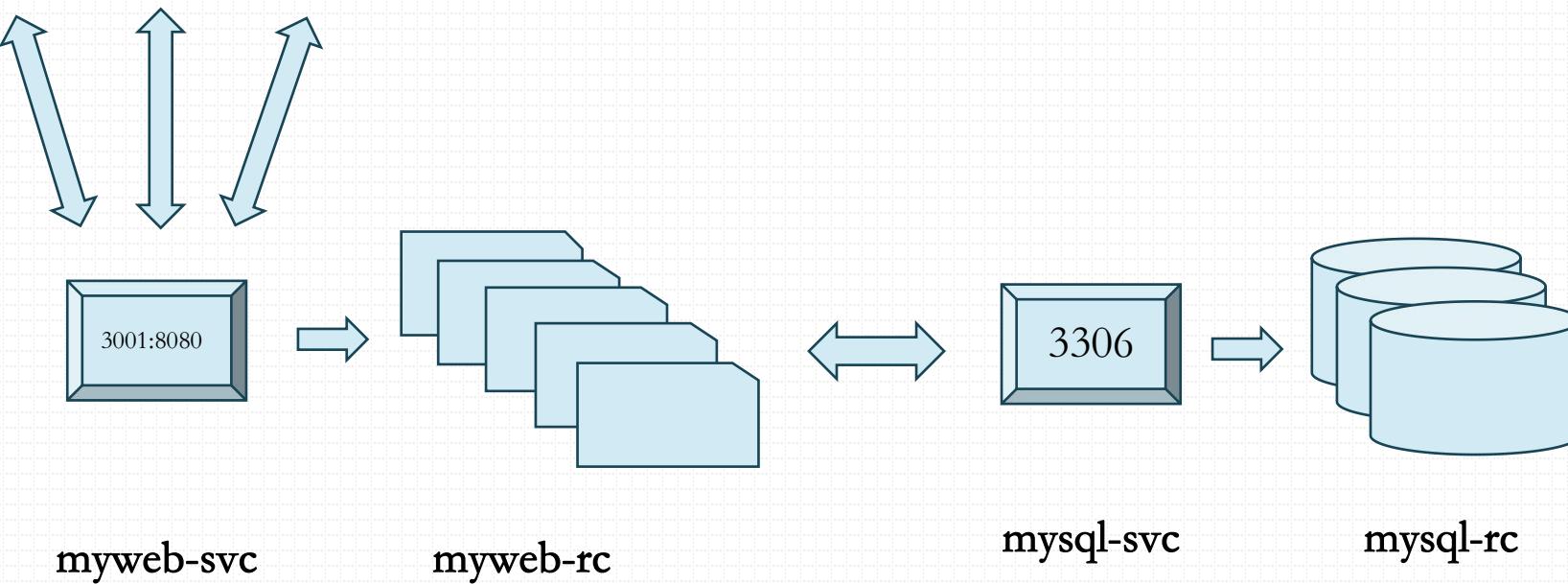
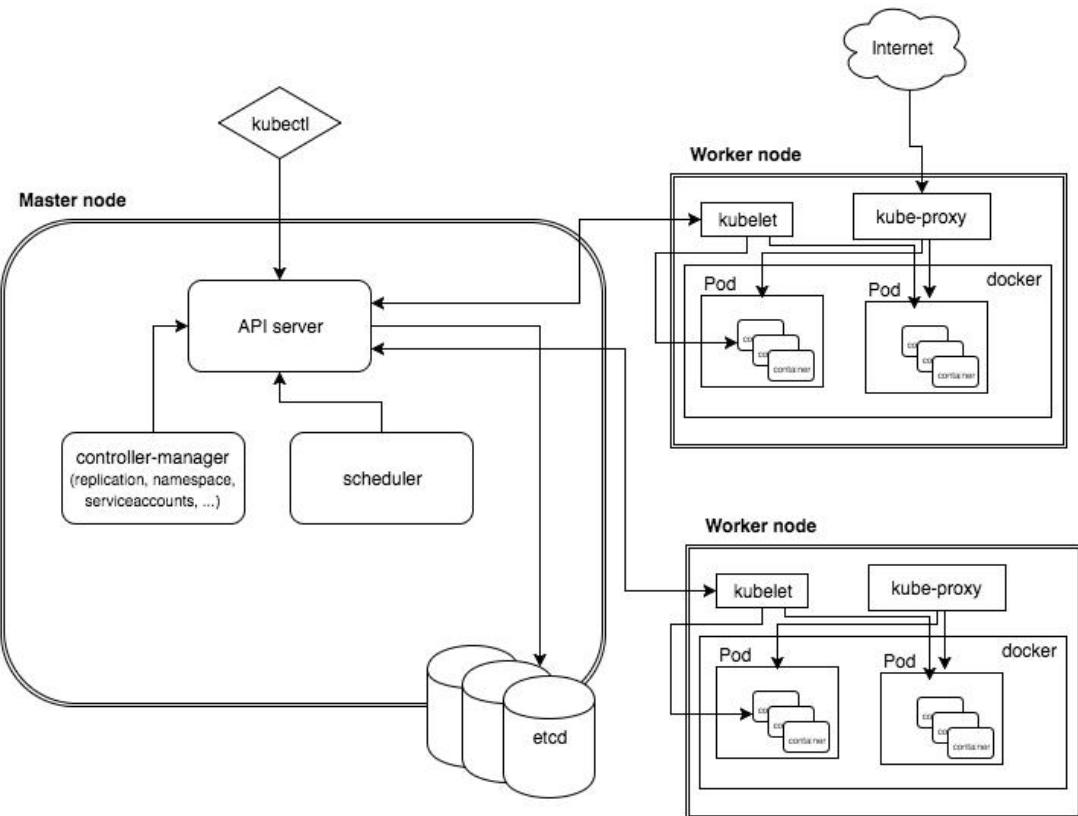


图 1.4 通过浏览器访问 Tomcat 应用



基本概念和术语



Master

- **kube-apiserver:** 提供REST接口的关键进程，资源的增删改查等操作的唯一入口，也是集群控制的入口进程；
- **kube-controller-manager:** 所有资源对象的自动化控制中心；
- **kube-scheduler:** 负责资源（Pod）调度；
- **etcd Server进程:** 存储所有资源对象

Node

- **kube-kubelet:** 负责Pod对应的容器的创建、起停等任务、同时与Master密切协作，实现集群管理；
- **kube-proxy:** 实现Service间的通信和负载均衡机制；
- **Docker Engine:** 负责本机的容器的创建和管理



Node

```
[root@bigdata example]# kubectl describe node 127.0.0.1
Name:           127.0.0.1
Role:           beta.kubernetes.io/arch=amd64
Labels:         beta.kubernetes.io/os=linux
                kubernetes.io/hostname=127.0.0.1
Taints:        <none>
CreationTimestamp: Sun, 07 Oct 2018 14:30:37 +0800
Phase:          Pending
Conditions:
  Type        Status  LastHeartbeatTime           LastTransitionTime        Reason                               Message
  ----        -----  ---------------------           ---------------------        ----
  OutOfDisk   False   Sun, 07 Oct 2018 17:01:55 +0800  Sun, 07 Oct 2018 14:30:37 +0800  KubeletHasSufficientDisk
  MemoryPressure   False   Sun, 07 Oct 2018 17:01:55 +0800  Sun, 07 Oct 2018 14:30:37 +0800  KubeletHasSufficientMemory
  DiskPressure   False   Sun, 07 Oct 2018 17:01:55 +0800  Sun, 07 Oct 2018 14:30:37 +0800  KubeletHasNoDiskPressure
  Ready        True    Sun, 07 Oct 2018 17:01:55 +0800  Sun, 07 Oct 2018 14:30:47 +0800  KubeletReady
Addresses:      127.0.0.1,127.0.0.1,127.0.0.1
Capacity:
  alpha.kubernetes.io/nvidia-gpu:  0
  cpu:                          56
  memory:                       131931160Ki
  pods:                         110
Allocatable:
  alpha.kubernetes.io/nvidia-gpu:  0
  cpu:                          56
  memory:                       131931160Ki
  pods:                         110
System Info:
  Machine ID:      49c08e003c514acf813ed869071eede5
  System UUID:     F23DBF99-D9EE-5A87-1F27-38D54775374D
  Boot ID:         c70b029f-fcd6-446d-a53c-816429ce68dc
  Kernel Version:  3.10.0-327.el7.x86_64
  OS Image:        CentOS Linux 7 (Core)
  Operating System: linux
  Architecture:   amd64
  Container Runtime Version: docker://1.13.1
  Kubelet Version: v1.5.2
  Kube-Proxy Version: v1.5.2
  ExternalID:     127.0.0.1
Non-terminated Pods:
  Namespace       Name            CPU Requests  CPU Limits   Memory Requests  Memory Limits
  ----           ----           -----          -----        -----          -----
  default        mysql-ck1n1      0 (0%)       0 (0%)      0 (0%)        0 (0%)
  default        myweb-2qgjc      0 (0%)       0 (0%)      0 (0%)        0 (0%)
  default        myweb-bqvfn      0 (0%)       0 (0%)      0 (0%)        0 (0%)
  default        myweb-khz4t      0 (0%)       0 (0%)      0 (0%)        0 (0%)
  default        myweb-pffvb      0 (0%)       0 (0%)      0 (0%)        0 (0%)
  default        myweb-tp6vq      0 (0%)       0 (0%)      0 (0%)        0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.
  CPU Requests  CPU Limits   Memory Requests  Memory Limits
  -----          -----        -----          -----
  0 (0%)         0 (0%)      0 (0%)        0 (0%)
No events.
```

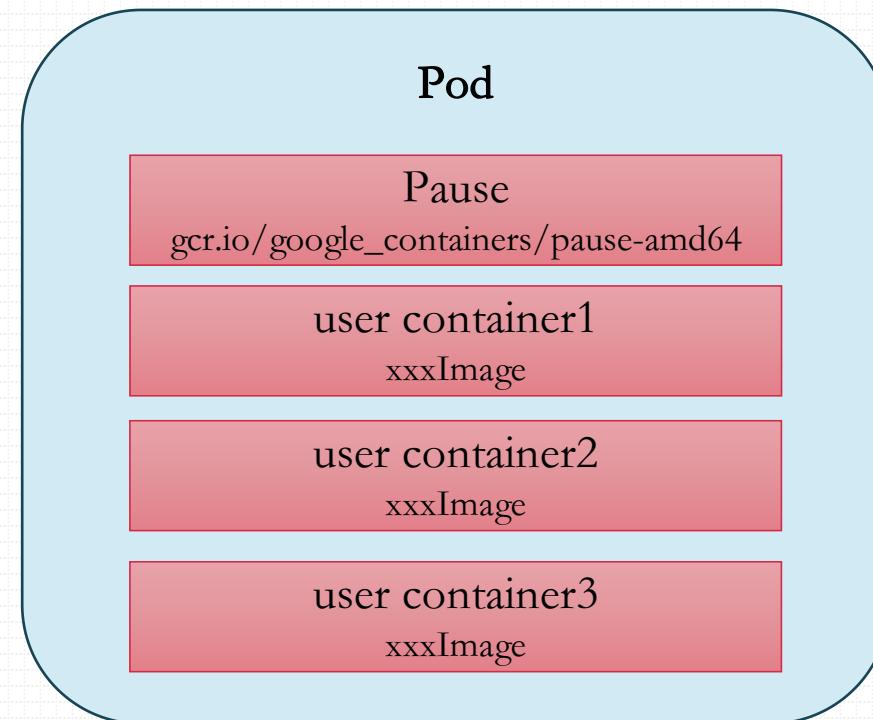
- Node基本信息：名称、角色、标签、创建时间
- Node当前运行状态
- Node的主机地址和主机名
- Node上的资源总量
- Node可分配资源量
- 主机系统信息
- 当前正在运行的Pod列表
- 已经分配的资源使用概况
- Node相关Event信息



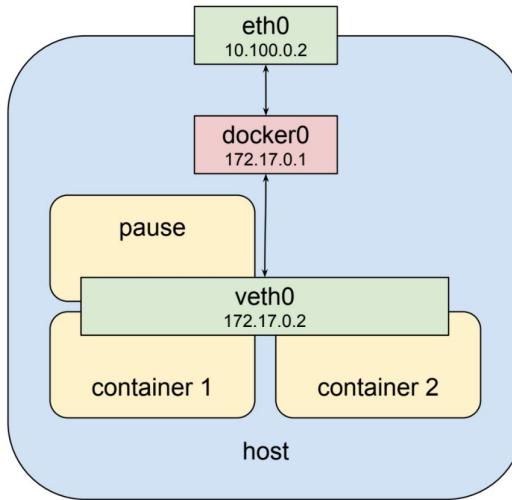
Pod - Why Pods??

- 当一组容器作为一个单元时，难以对“整体”简单的进行判断及有效的进行行动
 - 引入业务无关、不宜死亡的Pause容器作为Pod的根容器
- Pod里的多个业务容器共享Pause容器的IP，共享Pause容器挂接的Volume
 - 简化业务容器间的通信，解决文件共享

- Pod可以看作是容器的小集合
- 在同一台机器里可以运行多个Pod
 - 共享资源
- 每个Pod有一个IP (Pod IP)
- Pod内部的容器共享网络 Namespace
 - IP Address
 - localhost



Pod Networking



- Pod内部的容器共享Pod的IP地址
- 同一集群内部的任意两个Pod之间可以通过TCP/IP直接通信
 - 通常需要虚拟二层网络技术实现（Flannel、Openvswitch）
- 一个Pod的容器与另外主机的Pod容器能够直接通信

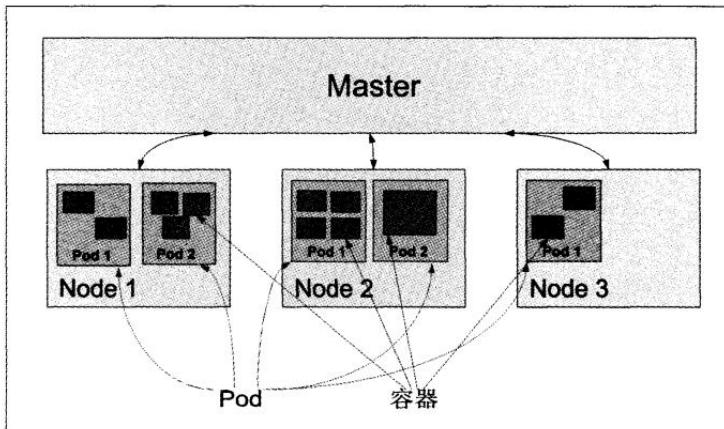


图 1.7 Pod、容器与 Node 的关系

Pod

- 静态Pod：不存储在etcd里，而是存储在某个具体Node的一个具体文件中
- 普通Pod：普通的Pod一旦创建就会被放入etcd中存储，随后让Master调度到合适的Node上并绑定，随后该Pod被对应的Node上的kubelet进程实例化成一组Docker容器并启动起来

Pod 定义

```
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
    name: myweb
spec:
  containers:
  - name: myweb
    image: kubeguide/tomcat-app:v1
    ports:
    - containerPort: 8080
    env:
      - name: MYSQL_SERVICE_HOST
        value: 'mysql'
      - name: MYSQL_SERVICE_PORT
        value: '3306'
```



拾遗

- Kubernetes Volume
 - 可以用分布式文件系统GFS实现后端的存储功能
 - 定义在Pod之上，然后被各个容器挂载到自己的文件系统
 - Volume vs. Docker Volumn
- Event (排查故障重要参考)

Events:						
FirstSeen	LastSeen	Count	From	SubobjectPath	Type	Reason
Message						
-----	-----	-----	-----	-----	-----	-----
-----	-----	-----	-----	-----	-----	-----
10h	12m	32	{kubelet k8s-node-1} spec.containers{kube2sky}			
Warning	Unhealthy	Liveness probe failed: Get http://172.17.1.2:8080/healthz: net/http: request canceled (Client.Timeout exceeded while awaiting headers)				



资源配额

- CPU配额
 - 比较奢侈，不敢造次！
 - 通常以千分之一的CPU配额作为最小单位，使用m表示（一般一个容器的CPU配额为100m~300m）
- 在k8s中，一个计算资源进行配额限定需要设定以下两个参数
 - Request: 该资源的最小申请量，系统必须满足要求
 - Limits: 资源允许使用的最大量，不能被突破，若容器超过，可能会被kill掉并重启

```
spec:  
  containers:  
    - name: db  
      image: mysql  
      resources:  
        requests:  
          memory: "64Mi"  
          cpu: "250m"  
        limits:  
          memory: "128Mi"  
          cpu: "500m"
```



Label

- Label是一个key=value的键值对，其中key和value需要用户指定
 - Label可以附加到各种资源对象上
 - Node、Pod、Service、RC等
 - Label可以在资源对象定义时确定，也可以在对象创建后动态添加或者删除
 - Label主要用于实现多维度的资源分组管理，方便进行资源分配、调度、配置部署等管理
 - 给某个资源对象打了标签后，可以利用标签选择器来查询和筛选出拥有某些Label的资源对象
-
- ◎ 版本标签: "release" : "stable", "release" : "canary"...
 - ◎ 环境标签: "environment": "dev", "environment": "qa", "environment": "production"
 - ◎ 架构标签: "tier" : "frontend", "tier" : "backend", "tier" : "middleware"
 - ◎ 分区标签: "partition" : "customerA", "partition" : "customerB"...
 - ◎ 质量管控标签: "track" : "daily", "track" : "weekly"



Label Selector

- 有两种Label Selector的表达式
 - 基于等式(=, !=)
 - name=redis-slave: 匹配所有具有标签name=redis-slave的资源对象
 - name!=redis-slave: 匹配所有不具有标签name=redis-slave的资源对象
 - 基于集合(in, not in)
 - name in (redis-master, redis-slave): 匹配所有具有标签name=redis-slave或者name=redis-master的资源对象
 - name not in (php-frontend): 匹配所有不具有标签name=php-frontend的资源对象
- 可以通过多个label selector表达式的组合实现更复杂的条件选择
 - 多个表达式之间使用“,”进行分隔即可，AND

Label Selector 在 Kubernetes 中的重要使用场景有以下几处。

- ◎ kube-controller 进程通过资源对象 RC 上定义的 Label Selector 来筛选要监控的 Pod 副本的数量，从而实现 Pod 副本的数量始终符合预期设定的全自动控制流程。
- ◎ kube-proxy 进程通过 Service 的 Label Selector 来选择对应的 Pod，自动建立起每个 Service 到对应 Pod 的请求转发路由表，从而实现 Service 的智能负载均衡机制。
- ◎ 通过对某些 Node 定义特定的 Label，并且在 Pod 定义文件中使用 NodeSelector 这种标签调度策略，kube-scheduler 进程可以实现 Pod “定向调度”的特性。



RC 高级部分

- 自动扩容

```
kubectl scala rc redis-slave --replicas=3
```

- 滚动升级

- 传统IT行业的应用升级：

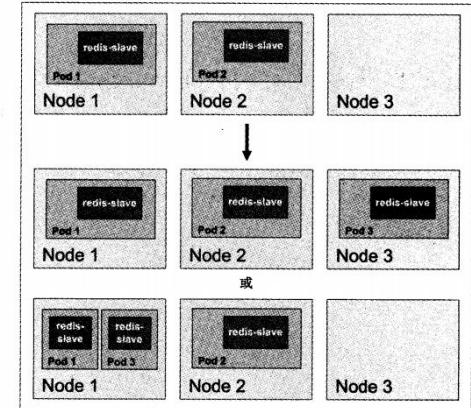
- BUILD一个新的Docker镜像，使用新的镜像版本来替代旧的版本达到目的

- 希望平滑的升级过程

- 旧版本的Pod每次停止一次，同时创建一个新版本的Pod，始终保持运行的Pod数目始终不变

- kubernetes实现了滚动升级

```
kubectl rolling-update xxx -f xxx-v2.yaml
```



下一代RC：Replica Set

- Replica Set

- 不仅支持等式的Label Selector而且支持集合的Label Selector
- 而RC仅支持等式的Label Selector

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
```

- 在大多数情况下，我们通过定义一个 RC 实现 Pod 的创建过程及副本数量的自动控制。
- RC 里包括完整的 Pod 定义模板。
- RC 通过 Label Selector 机制实现对 Pod 副本的自动控制。
- 通过改变 RC 里的 Pod 副本数量，可以实现 Pod 的扩容或缩容功能。
- 通过改变 RC 里 Pod 模板中的镜像版本，可以实现 Pod 的滚动升级功能。



Deployment

- 更好的解决Pod的编排问题
- 在Deployment内部引入Replica Set.
- 可以看做RC的一次升级，两者相似度超过90%.
- Deployment相对于RC的一个最大升级就是我们可以随时知道当前Pod“部署”的进度.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
```

```
apiVersion: v1
kind: ReplicaSet
metadata:
  name: nginx-repset
```

Deployment vs. ReplicaSet

- 创建一个 Deployment 对象来生成对应的 Replica Set 并完成 Pod 副本的创建过程。
- 检查 Deployment 的状态来看部署动作是否完成（Pod 副本的数量是否达到预期的值）。
- 更新 Deployment 以创建新的 Pod（比如镜像升级）。
- 如果当前 Deployment 不稳定，则回滚到一个早先的 Deployment 版本。

应用场景



```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
        - name: tomcat-demo
          image: tomcat
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080

```

kubectl create -f tomcat-deployment.yaml



```

[root@bigdata example]# kubectl get deployments
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
frontend   1        1        1          1          51s

```

- **DESIRED**: Pod副本的期望值，即Deployment里定义的replicas；
- **CURRENT**: 当前Replica的值，实际上是Deployment创建的Replica Set里的Replica值，这个值会一直增加，直到达到DESIRED为止；
- **UP-TO-DATE**: 最新版本的Pod的副本数量，用于指示在滚动升级的过程中；
- **AVAILABLE**: 当前集群中可用的Pod副本数量，即集群中当前存活的Pod数量.

```

[root@bigdata example]# kubectl get rs
NAME      DESIRED  CURRENT  READY  AGE
frontend-141477217  1        1        1        2m

```

```

[root@bigdata example]# kubectl get pods
NAME      READY  STATUS  RESTARTS  AGE
frontend-141477217-h0v0r  1/1    Running  0        4m

```

HPA (Horizontal Pod Autoscaling)

- Pod横向自动扩容，也是一种Kubernetes资源对象
 - 通过追踪分析RC控制的所有目标Pod的负载变化情况来确定是否需要针对性的调整目标Pod的副本数
- Pod负载度量指标
 - CPUUtilizationPercentage
 - 应用程序自定义的度量指标 (QPS/TPS)

- CPUUtilizationPercentage是一个算数平均值，目标Pod所有副本自身的CPU利用率的平均值；
 - CPU利用率=CPU使用量/Pod Request
 - CPUUtilizationPercentage计算过程使用的Pod的CPU使用量通常是1分钟内的平均值
 - 若要查询，需要部署Heapster扩展组件

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 90
```

```
[root@bigdata example]# kubectl autoscale deployment php-apache --cpu-percent=90 --min=1 --max=10
```



再谈Service（最核心的资源对象之一）

- 微服务
- 搞懂这几个访问方式：
 - Pod IP + ContainerPort
 - 负载均衡器机制
 - Service IP/Cluster IP + Port
- Service IP一旦创建，在Service的生命周期内不会改变

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
  - port: 8080
  selector:
    tier: frontend
```

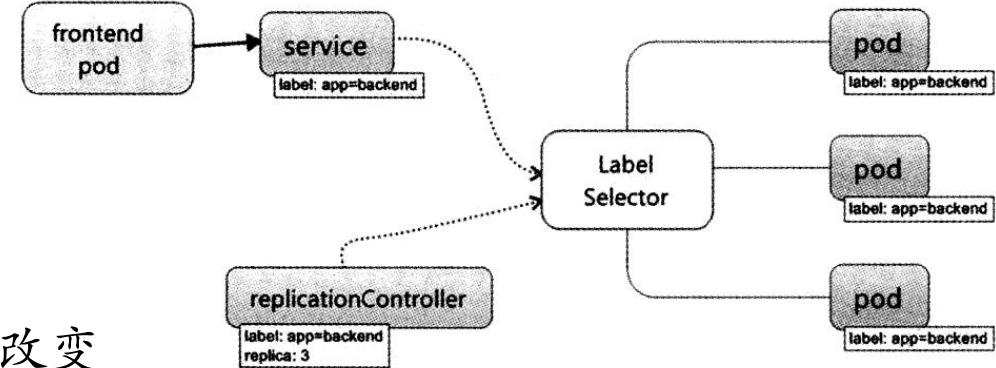


图 1.14 Pod、RC 与 Service 的关系

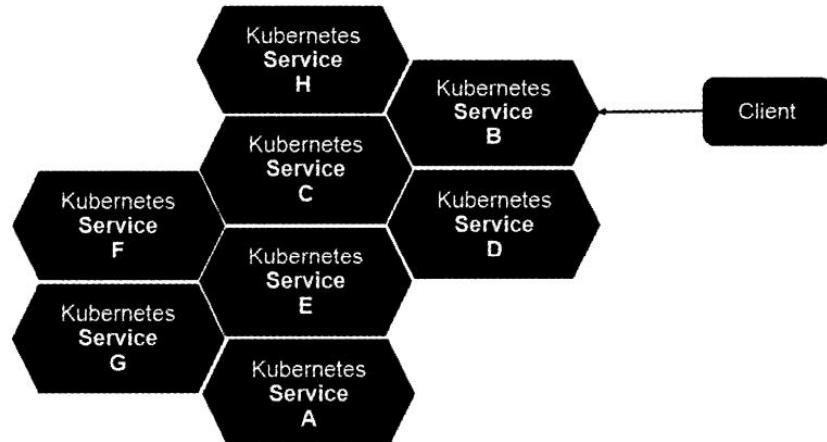
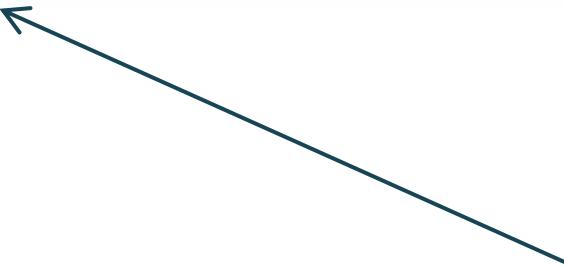


图 1.15 Kubernetes 所提供的微服务网格架构



```
# kubectl get endpoints  
NAME           ENDPOINTS      AGE  
kubernetes     192.168.18.131:6443   15d  
tomcat-service 172.17.1.3:8080    1m
```

```
# kubectl get svc tomcat-service -o yaml  
apiVersion: v1  
kind: Service  
metadata:  
  creationTimestamp: 2016-07-21T17:05:52Z  
  name: tomcat-service  
  namespace: default  
  resourceVersion: "23964"  
  selfLink: /api/v1/namespaces/default/services/tomcat-service  
  uid: 61987d3c-4f65-11e6-a9d8-000c29ed42c1  
spec:  
  clusterIP: 169.169.65.227  
  ports:  
    - port: 8080  
      protocol: TCP  
      targetPort: 8080  
  selector:  
    tier: frontend  
  sessionAffinity: None  
  type: ClusterIP  
status:  
  loadBalancer: {}
```



- spec.ports的定义中，targetPort属性用于确定提供该服务的容器所暴露的端口号；
- port属性定义了Service的虚端口
- 若在定义服务时，没有指定target Port，则默认Target与port相同。



kubernetes中的服务发现机制

- 最早采用了Linux环境变量的方式
 - 每个Service生成一些对应的Linux环境变量，并在每个Pod的容器在启动时，自动注入这些环境变量

```
TOMCAT_SERVICE_SERVICE_HOST=169.169.41.218
TOMCAT_SERVICE_SERVICE_PORT_SERVICE_PORT=8080
TOMCAT_SERVICE_SERVICE_PORT_SHUTDOWN_PORT=8005
TOMCAT_SERVICE_SERVICE_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP_PORT=8005
TOMCAT_SERVICE_PORT=tcp://169.169.41.218:8080
TOMCAT_SERVICE_PORT_8080_TCP_ADDR=169.169.41.218
TOMCAT_SERVICE_PORT_8080_TCP=tcp://169.169.41.218:8080
TOMCAT_SERVICE_PORT_8080_TCP_PROTO=tcp
TOMCAT_SERVICE_PORT_8080_TCP_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP=tcp://169.169.41.218:8005
TOMCAT_SERVICE_PORT_8005_TCP_ADDR=169.169.41.218
TOMCAT_SERVICE_PORT_8005_TCP_PROTO=tcp
```

- 目前采用了Add-On 增值包，引入DNS系统，将服务名作为DNS域名，这样程序就可以直接使用service name 建立通信连接



外部系统访问Service的问题

- 三种IP

- Node IP: Node节点的IP地址；
- Pod IP: Pod的IP地址；
- Cluster IP: Service 的IP地址.

	类型	能否直接通信	通信方式	备注
Node IP	真实	能	Node IP	<ul style="list-style-type: none">• 每个节点的真实的物理网卡地址
Pod IP	虚拟	否	Pod IP所在的虚拟二层网络	<ul style="list-style-type: none">• 虚拟的二层网络• 不同的Node的Pod之间可以直接通信
Cluster IP	虚拟	否	Cluster IP + Cluster Port	<ul style="list-style-type: none">• 单独的ClusterIP不具备TCP/IP通信的基础• Cluster IP无法被PING• Cluster IP仅仅作用于Service对象，由k8s管理和分配IP



Node Port

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 31002
  selector:
    tier: frontend
```

Node IP + nodePort

- NodePort并未完全解决外部访问Service
 - 负载均衡问题

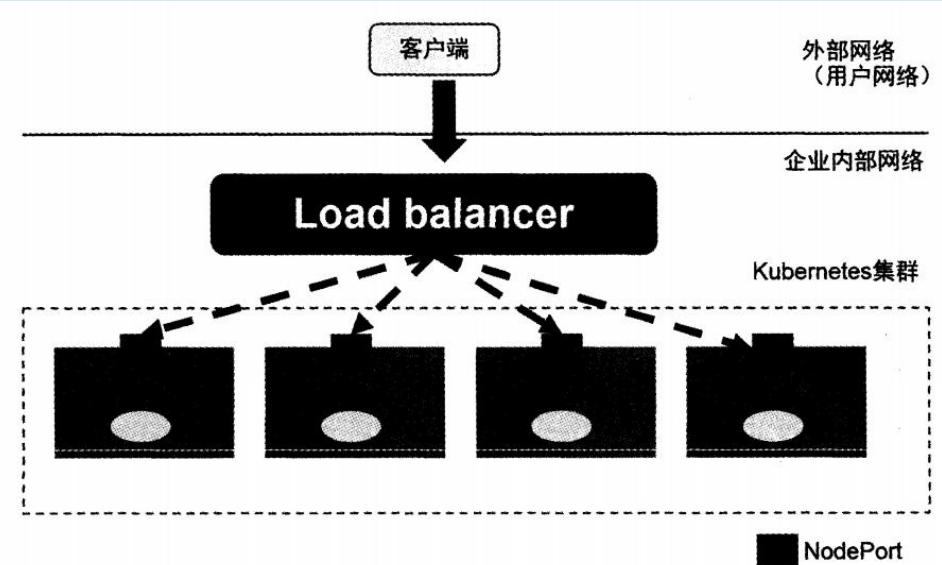


图 1.17 NodePort 与 Load balancer



Volume(存储卷)

1. emptyDir

- ◎ 临时空间，例如用于某些应用程序运行时所需的临时目录，且无须永久保留。
- ◎ 长时间任务的中间过程 CheckPoint 的临时保存目录。
- ◎ 一个容器需要从另一个容器中获取数据的目录（多容器共享目录）。

2. hostPath

- ◎ 容器应用程序生成的日志文件需要永久保存时，可以使用宿主机的高速文件系统进行存储。
- ◎ 需要访问宿主机上 Docker 引擎内部数据结构的容器应用时，可以通过定义 hostPath 为宿主机 /var/lib/docker 目录，使容器内部应用可以直接访问 Docker 的文件系统。

3. gcePersistentDisk

4. awsElasticBlockStore

5. NFS

6. iscsi



- ◎ **flocker**: 使用 Flocker 来管理存储卷。
- ◎ **glusterfs**: 使用开源 GlusterFS 网络文件系统的目录挂载到 Pod 中。
- ◎ **rbd**: 使用 Linux 块设备共享存储 (Rados Block Device) 挂载到 Pod 中。
- ◎ **gitRepo**: 通过挂载一个空目录，并从 GIT 库 clone 一个 git repository 以供 Pod 使用。
- ◎ **secret**: 一个 secret volume 用于为 Pod 提供加密的信息，你可以将定义在 Kubernetes 中的 secret 直接挂载为文件让 Pod 访问。secret volume 是通过 tmpfs (内存文件系统) 实现的，所以这种类型的 volume 总是不会持久化的。



PV (Persistent Volume)

- 网络存储
- PV与Volume的区别
 - PV 只能是网络存储，不属于任何 Node，但可以在每个 Node 上访问。
 - PV 并不是定义在 Pod 上的，而是独立于 Pod 之外定义。
 - PV 目前只有几种类型：GCE Persistent Disks、NFS、RBD、iSCSI、AWS ElasticBlockStore、GlusterFS 等。
- NFS类型PV的一个yaml定义文件

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /somepath
    server: 172.17.0.2
```

- **ReadWriteOnce**: 读写权限，并且只能被单个Node挂载；
- **ReadOnlyMany**: 只读权限，允许被多个Node挂载；
- **ReadWriteMany**: 读写权限，允许被多个Node挂载



Namespace(命名空间)

- 使用Namespace实现多租户的资源隔离；

```
# kubectl get namespaces
```

NAME	LABELS	STATUS
default	<none>	Active

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

```
# kubectl get pods --namespace=development
```

NAME	READY	STATUS	RESTARTS	AGE
busybox	1/1	Running	0	1m

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: development
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      name: busybox
```



总结

- Kubernetes的前身Borg集群管理系统
- Kubernetes的核心组件与资源对象
 - Master/Node
 - service
 - RC/RS/Deployment
 - Pod
 - Namespace
 - Volumes
 - HPA等等...
- 下一步计划
 - 二进制安装Kubernetes
 - kubernetes的命令行工具使用
 - 深入Pod
 - 深入Service
 - Kubernetes的网络机制



A large blue graphic of a sailboat is positioned on the left side of the slide. It has a white hull and two blue sails. The boat is angled towards the right.

Thank you!!!

@rh01

Dec 7th, 2018

