1. Kubernetes基础

1.1 主要用途和特性

Kubernetes（k8s）可以看做是谷歌严格保密十几年的秘密武器---Borg的一个开源版本。Borg是谷歌一个久负盛名的内部使用的大规模集群管理系统，基于容器技术实现资源管理的自动化及跨数据中心资源利用最大化。Kubernetes是自动化容器操作的开源集群管理平台，可以实现容器部署、调度和节点集群间扩展等操作。Docker是Kubernetes内部使用的低级别组件，Kubernetes不仅仅支持Docker，还支持Rocket。

Kubernetes有以下特性：

- 自动化容器的部署和复制
- 随时扩展或收缩容器规模
- 将容器组织成组，并且提供容器间的负载均衡
- 很容易地升级应用程序容器的新版本
- 提供容器弹性，如果容器失效就替换它
- 资源配额管理，确保指定对象在任何时候都不会超量占用系统资源
- 高可用（HA）
- 集群监控
- ...

实际上，使用Kubernetes只需一个部署文件，一条命令就可以部署多层容器（前端，后台等）的完整集群：

```
$ kubectl create -f single-config-file.yaml
```

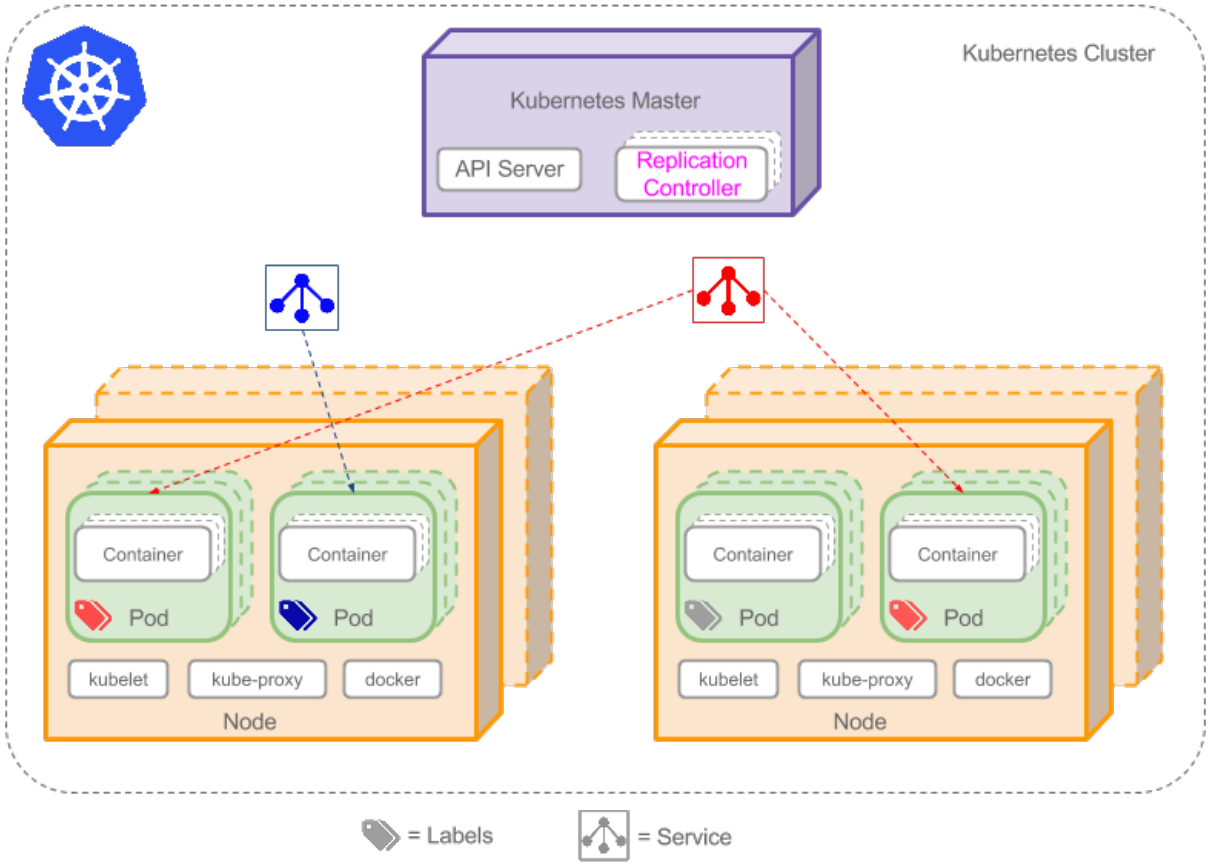
kubectl是和Kubernetes API交互的命令行程序。现在介绍一些核心概念。

1.2 基本概念

Kubernetes中，Node、Pod、Replication Controller、Service等概念都可以看做一种资源对象，通过Kubernetes提供的kubectl工具或者API调用进行操作，并保存在etcd中。

1.2.1 Cluster--集群

集群是一组节点，这些节点可以是物理服务器或者虚拟机，之上安装了Kubernetes组件。典型的Kubernetes架构图如下图所示，注意该图为了强调核心概念有所简化，但是逻辑关系更清晰。



上图可以看到以下KUbernetes组件，使用特别的图标表示Service和Label：

- Kubernetes Master（Kubernetes主节点）
- Node（节点）
- Pod
- Container（容器）
- Label( )（标签）
- Replication Controller（复制控制器）
- Service（ ）（服务）

1.2.2 Node

Node是Kubernetes集群中相对于Master而言的主机，在较早版本中被称作Minion。在Node上运行的而服务进程包括Kubelet、kube-proxy和docker daemon。

Node的信息如下：

- Node地址： 主机的IP地址或者Node ID
- Node运行状态：包括Pending、Running、Terminated三种状态。
- Node Condition:描述Running状态Node的运行条件，目前只有一种条件--Ready,表示Node处于健康状态，可以接收从Master发来的创建Pod的指令。
- Node 系统容量：描述Node可用的系统资源，包括CPU、内存数量、最大可调度Pod数量等。
- 其它信息：Kuebrnetes/Docker版本号、操作系统信息等...

查看node详细信息示例：

```
kubectl get nodes
kubectl describe node <nodename>
```

Node的管理：Kubernetes只是创建node对象，然后对其进行一系列健康检查，包括可以是否可以连通、是否可以创建Pod等。若检查未能通过，则

标记为不可用(Not Ready)。可使用NodeController实现集群范围内Node信息的同步及单个Node的生命周期管理。

Node自注册：通过kubelet进程启动参数配置实现自注册，需指定参数如下：

```
- --apiserver=:apiserver的地址
- --register-node=: true
-  ...
```

kubelet会向apiserver注册自己。自注册是Kubernetes推荐的Node管理方式。

1.2.3 Pod

Pod是Kubernetes的最基本的操作单元，包含一个或多个紧密相关的容器。Pod在Node上被创建、启动或者销毁。**一个Pod中的多个应用容器通常是紧密耦合的，并且共享一组资源：**

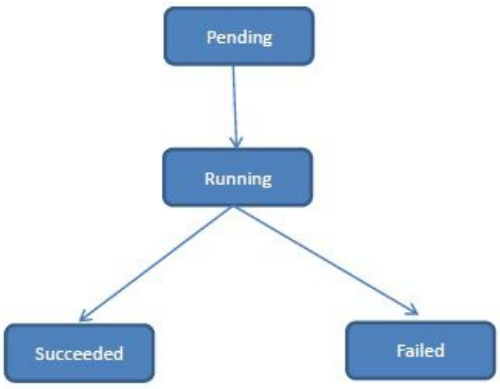
- PID Namespace：Pod中的不同应用程序可以看到其他应用程序的进程PID。
- Network Namespace：Pod中的多个容器能够访问同一个IP和端口范围，IP per Pod。
- UTS Namespace：Pod中的多个容器共享一个主机名。
- Volumes（共享存储卷）：Pod中的各个容器可以访问在Pod级别定义的Volumes。

Pod定义：Pod定义通过Yaml或json格式的配置文件完成，其中kind为Pod,spec中包含了对Containers的定义，可以定义多个容器。简单的pod定义文件如下所示：

```
//redis-slave-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:  # specification of the pod's contents
  restartPolicy: Never
  containers:
  - name: slave
    image: kubeguide/guestbook-redis-slave
    ports:
    - containerPort: 6379
    env:
    - name: GET_HOSTS_FROM
      value: env
```

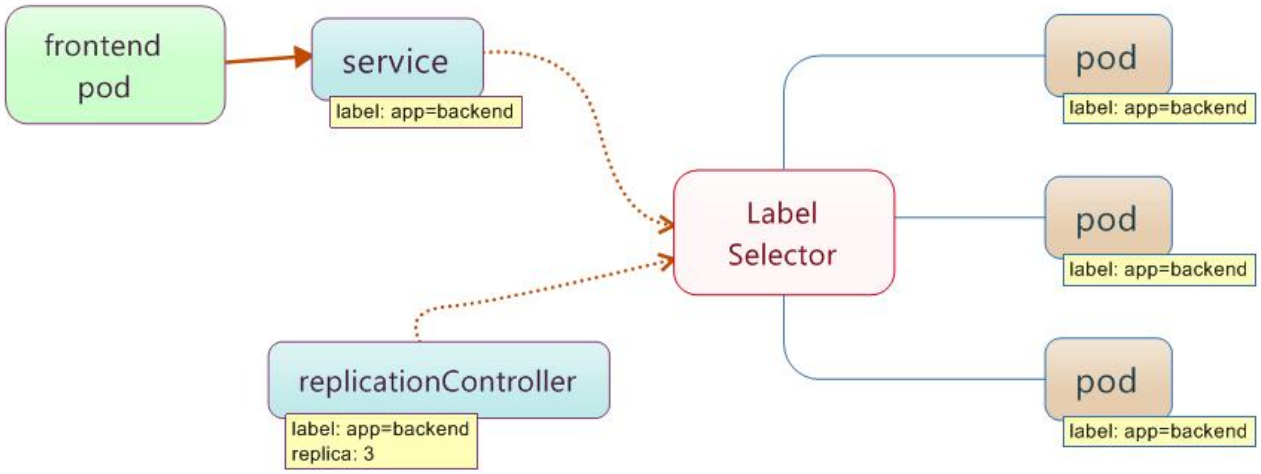
Pod的生命周期：通过ymal/json模板定义-->分配到Node上运行-->Pod所含容器运行结束后Pod也结束。
整个过程Pod处于以下四种状态之一：

- Pending：Pod定义正确，提交至Master，但是容器还未完全创建（Pod调度、镜像下载/运行等耗时）。
- Running：Pod已被分配到某个Node上，且其包含的所有Container镜像均已创建完成并正确运行。
- Succeeded：Pod中所有容器都成功结束，且不会被重启，这是Pod的一种最终状态。
- Failed：Pod中所有容器都结束可，但至少有一个容器是以失败结束的。这也是Pod的一种最终状态。



1.2.4 Lable

Label( )是KUbernetes系统中的核心概念，其本质是attach到各种对象（如Pod、Servcie、RC、Node等）上的键/值对，用来传递用户定义的属性。比如，可以创建一个"tier"和“app”标签，通过Label（tier=frontend, app=myapp）来标记前端Pod容器，使用Label（tier=backend, app=myapp）标记后台Pod。然后可以使用Selectors选择带有特定Label的Pod，并且将Service或者Replication Controller应用到上面。



Label选择器有两种，分别是:

- **Equality-based选择器**

```
#Equality-based Label selector
#匹配Label具有environment key且等于production的对象
environment = production
#匹配具有tier key，但是值不等于frontend的对象
tier != frontend
#kubernetes使用AND逻辑，第三条匹配production但不是frontend的对象。
environment = production，tier != frontend
```

- **Set-based选择器**

```
#Set-based Label selector
#选择具有environment key，而且值是production或者qa的label附加的对象
environment in (production, qa)
#选择具有tier key，但是其值不是frontend和backend
tier notin (frontend, backend)
#选则具有partition key的对象，不对value进行校验
partition
```

> 使用Label可以给对象创建多组标签，Service、RC等组件则通过Label Selector来选择对象范围。Label和Label Selector共同构成了Kubernetes系统中最核心的应用模型，使得被管理对象能够被精确地分组管理。

1.2.5 Replication Controller

Replication用于解决以下几个问题：

- 是否手动创建Pod？
- 如果想要创建同一个容器的多份拷贝，需要一个个分别创建出来么？
- 能否将Pods划到逻辑组里？

Replication Controller定义

当创建Replication Controller时，需要指定以下内容：

- Pod模板（spec.template）：用来创建Pod副本的模板。
- Label(spec.selector):Replication Controller需要监控的Pod的标签。
- spec.replicas:创建的pod副本的数量。

一个简单的Replication Controller定义模板文件实例如下：

```
//redis-slave-controller.yaml
apiVersion: v1
kind: ReplicationController  //定义类型：ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 3   //创建的pod副本的数量
  selector:
    name: redis-slave  //Replication Controller需要监控标签为“name = redis-slave”Label的Pod
  template:     //Pod 副本模板定义
    metadata:  //Pod的元数据
      labels:
        name: redis-slave
    spec:
      containers:   //Pod内运行的容器信息
      - name: slave
        image: kubeguide/guestbook-redis-slave  //容器镜像
        ports:
        - containerPort: 6379
        env:
        - name: GET_HOSTS_FROM
          value: env
```

Replication Controller确保任意时间都有指定数量（spec.replicas）的Pod“副本”在运行。如果为某个Pod创建了Replication Controller并且指定3个副本，它会创建3个Pod，并且持续监控它们。如果某个Pod不响应，那么Replication Controller会替换它，保持总数为3.如下面的动画所示：

如果之前不响应的Pod恢复了，现在就有4个Pod了，那么Replication Controller会将其中一个终止保持总数为3。如果在运行中将副本总数改为5，Replication Controller会立刻启动2个新Pod，保证总数为5。还可以按照这样的方式缩小Pod。

Replication Controller主要功能：

- **Rescheduling**： Replication Controller会确保Kubernetes集群中指定的pod副本(replicas)在运行， 即使在节点出错时。
- **Scaling**： 通过修改Replication Controller的副本(replicas)数量来水平扩展或者缩小运行的pods。
- **Rolling updates**：Replication Controller的设计原则使得可以一个一个地替换pods来滚动更新服务。

1.2.6 Service

Kubernetes中Replication Controller可以现在已经创建多个Pod的replicas，而且每个Pod都会被分配一个IP地址,但是IP地址会随着Pod的销毁而消失。在此背景下，Service提出用于解决以下问题：

- 如果有一组Pod组成一个集群来提供服务，那么如何访问这些Pods？
- 在这些Pod副本上如何均衡负载呢？

一个Service可以看做一组提供相同服务的Pod的对外访问接口。Service作用于哪些Pods是通过Label Selector定义的。

Service 定义
Service的定义亦使用Yaml或Json格式的配置文件完成。以redis-slave服务的定义为例：

```
//redis-slave-service.yaml
apiVersion: v1
kind: Service //定义类型：Service
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  selector:
```

```
    name: redis-slave  //选择具有“name = redis-slave”Label的Pod
  ports:
  - port: 6379  //容器监听的端口号
```

通过该定义，Kubernetes会创建一个名为“redis-slave”的服务，spec.selector的定义表示该Service会选择具有“name = redis-slave”Label的Pod。

自动创建Endpoints：

Kubernetes会根据Service的定义创建出与Pod同名的Endpoint对象，以建立起Service与后端Pod的对应关系。Endpoint对象主要由Pod的IP地址和容器需要监听的端口号列表组成，通过`kubectl get endpoints`命令可以查看，显示为IP:port格式。（kube-proxy就是从Endpoint列表中选择服务后端Pod的）。

Pod的IP地址和Service的Cluster IP地址

- Pod的IP地址：Docker Daemon根据docker0网桥的IP 地址段（`--bip=...`）进行分配
- Service的Cluster IP地址：Kubernetes系统中的虚拟IP地址，由系统动态分配。

Service的Cluster IP地址比Pod的IP地址相对稳定：

- Service创建时即分配一个 Cluster IP地址，在销毁该Service之前，Cluster IP地址均不会变化。
- Pod生命周期较短，销毁后再次创建会分配一个新的Pod IP地址。

内部访问Service：

Kubernetes在一个集群内创建的对象或者在代理集群节点上发出访问的客户端我们称之为**内部使用者**。要把服务暴露给内部使用者，Kubernetes支持两种方式：环境变量和DNS。

- **环境变量**：当kubelet在该Pod的所有容器中为当前运行的Service设置一系列环境变量，这样Pod就可以通过环境变量访问这些Service了。 不常用。
- **DNS**：通过Service name找到Service，DNS返回的查找结果是Cluser IP，并通过Cluster访问后端Pod（kube-proxy实现，后面介绍）。推荐使用。

下述动画展示了Service的功能。

外部访问Service：

Kubernetes支持两种对外服务的Service的type定义：Nodeport和LoadBalancer，通过spec.type指定。

- **NodePort**：在定义Service时指定spec.type=NodePort，并指定spec.ports.nodePort的值，系统就会在Kubernetes的集群中的每个Node上打开一个主机的真实端口号。能够访问Node的外部客户端均可通过访问spec.ports.nodePort指定的端口号访问到集群内部的Service。 以phpfrontend为例,其定义文件如下：

  ```
  //php-frontend-service.yml
  apiVersion: v1
  kind: Service
  metadata:
    name: frontend
    labels:
      name: frontend
  spec:
    selector:
      name: frontend
    type: NodePort  //指定对外服务类型为NodePort方式
    ports:
    - port: 80
      nodePort: 30001 //指定Node打开的真实端口。
  ```

  > 若php-frontend Pod运行在多个主机上，则可以通过其中任意一个主机访问该php-frontend Service,访问方式为：NodeIP:nodePort

- **LoadBalancer**：定义Service时指定spec.type=LoadBalancer，同时需要指定LoadBalancer的IP地址。使用LoadBalance需同时指定Service的nodePort和cluserIP。之后，该Service的访问请求将会通过LoadBalancer转发到后端Pod上去。

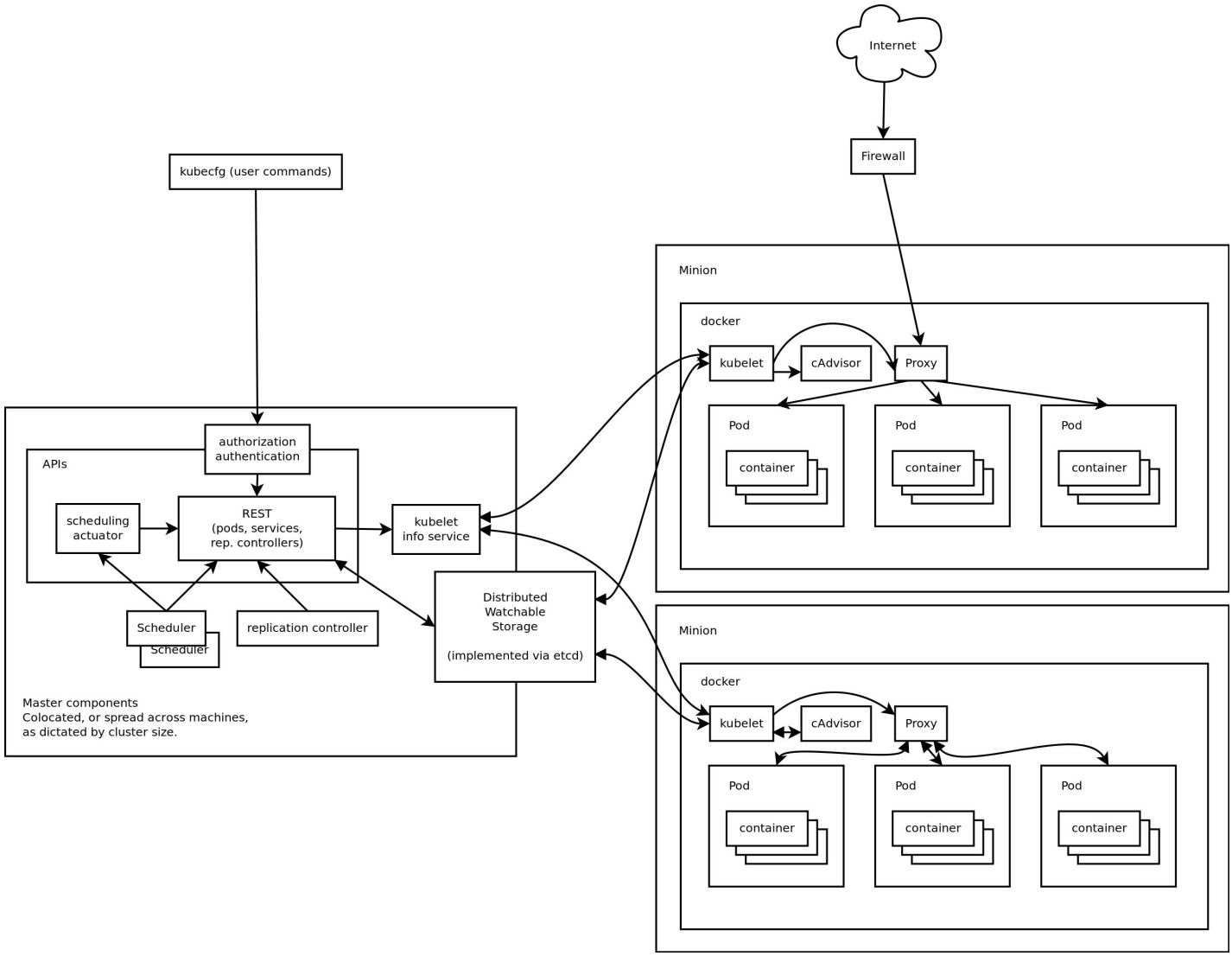
  ```
  //loadbalancer.json
  {
          "kind": "Service",
  ```

```
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
        },
    "spec": {
        "selector": {
        "app": "MyApp"
        },
        "ports": [
            {
                "protocol": "TCP",
                "port": 80,                    #  service port， clusterIP：port,内部访问
                "targetPort": 9376,       #  pod中某个容器的Port， Kube-proxy-->container
                "nodePort": 30061           #  指定nodePort： NodeIP:nodePort,外部访问
            }
        ],
        "clusterIP": "10.0.171.239",        #  指定clusterIP
        "type": "LoadBalancer"             #  指定对外服务类型为LoadBalancer方式
    },
    "status": {
        "loadBalancer": {
            "ingress": [
                {
                    "ip": "146.148.47.155"  #  指定LoadBalancer的IP地址
                }
            ]
        }
    }
}
```

1.3 基本组件构成及功用

Kubenetes整体框架如下图，Kubernetes集群中的主机按功能划分为Master节点和Node节点，并根据其职责运行不同的Kubernetes组件。

下图是官方给出的完整的架构图：



master运行三个组件：

- **apiserver**
- **controller-manager**
- **scheduler**

Node(or minion)运行两个组件：

- **kubelet**
- **proxy**

1.3.1 API Server

API Server作为kubernetes系统的入口，封装了核心对象的增删改查操作，以RESTFul接口方式提供给外部客户和内部组件调用。

Kubernetes API Server功能：

- Kubernetes中各种REST对象（如Pod、RC、Service、Namespace及Node等）的数据通过该API接口被提交到后端的持久化存储（etcd）中。
- Kubernetes集群中的各部件之间通过该API接口交互，实现解耦合。
- kubectl命令也是通过访问该API接口实现其强大的管理功能。

swagger-ui API在线查询功能：

Swagger UI是一款第三方REST API文档在线自动生成和功能测试软件。为了方便查阅API接口的详细定义，Kubernetes使用了swagger-ui提供API在线查询功能，Kubernetes开发团队会定期更新、生成UI及文档。访问方式有如下两种：

- 官网：http://kubernetes.io/third_party/swagger-ui/

- Master节点：http://172.21.101.102:8080/swagger-ui/

API Server访问：http://172.21.101.102:8080/

1.3.2 controller-manager

Controller Manager作为集群内部的管理控制中心，负责集群内的Node、Pod副本、服务端点（Endpoint）、命名空间（NameSpace）、服务账号（ServiceAccount）、资源定额（ResourceQuota)等的管理并执行自动修复流程，确保集群处于预期的工作状态。比如出现某个Node意外宕机时，Controller会在集群的其他节点上自动补齐Pod副本。

Controller Manager内部主要包含以下控制器：

- **Replication Controller:**确保在任何时候集群中一个RC所关联的Pod都保持一定数量的Pod副本处于正常运行状态。
- **Node Controller:**负责发现、管理和监控集群中的各个Node节点。
- **ResourceQuota Controller：**确保指定对象任何时候都不会超量占用系统资源、避免了由于某些业务进程的设计或实现的缺陷导致整个系统进行紊乱甚至意外宕机，对整个吸引的平稳运行和稳定性具有非常重要的作用。
- **NameSpace Controller:**定时通过API Server读取NameSpace信息，并可根据API标识优雅删除NameSpace。
- **ServiceAccount Controller:**安全相关控制器，在Controller Manager启动时被创建。监听Service Account的删除事件和NameSpace的创建修改事件。
- **Token Controller：**安全相关控制器，负责监听Service Account和Secret的创建、修改和删除事件，并根据事件的不同做不同的处理。
- **Service Controller:**负责监听Service的变化。
- **Endpoint Controller:**通过Store缓存Service和Pod信息，它监控Service和Pod的变化。

Kubernetes集群中，每个Controller就是一个操作系统，它通过API Server监控系统的共享状态，并尝试着将系统状态从“现有状态”修正到“期望状态”。

1.3.3 Scheduler

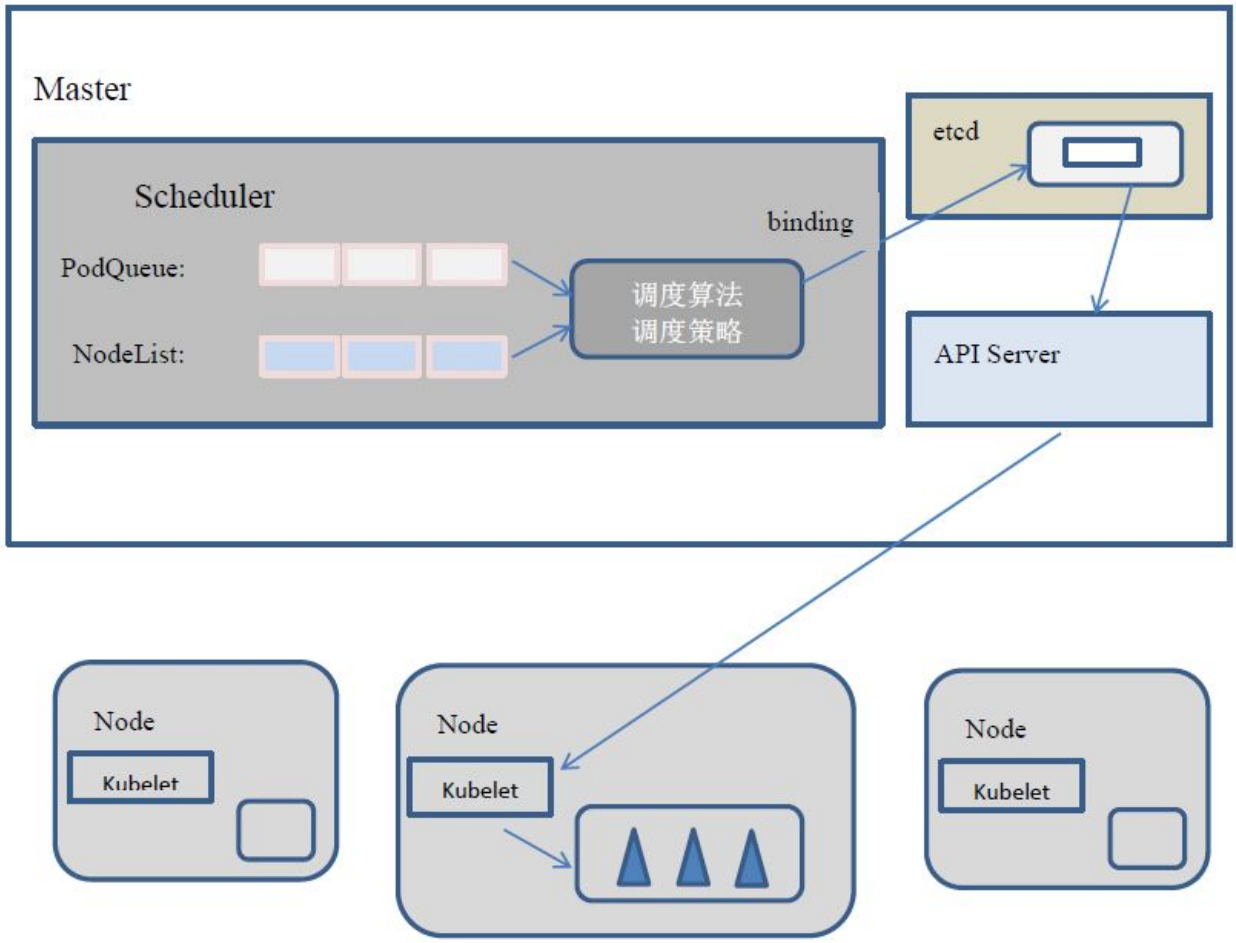
Kubernetes Scheduler的作用是将带调度的Pod(API新创建的Pod、Controller Manager为补足副本而创建的Pod等)按照特定的调度算法和调度策略绑定（binding）到集群中的某个合适的Node上，并将绑定信息写入etcd中。

在整个调度过程中涉及三个对象，分别是：

- 待调度Pod列表
- 可用Node列表
- 调度算法和策略。

Scheduler通过调度算法调度为待调度Pod列表的每个Pod从Node列表中选择一个最合适的Node。随后，目标节点上的Kubelet通过API Server监听到Kubernetes Scheduler产生的Pod绑定事件，然后获取相应的Pod清单，下载Image镜像，并启动容器。

完整的流程如下图所示：



Kubernetes Scheduler当前提供的默认调度流程分为以下两步：
（1）预选调度过程：即遍历所有目标Node，筛选出符合要求的候选节点。为此Kubernetes内置了多种预选策略（xxx Predicates）供用户选择。
（2）确定最优节点：在第一步的基础上，采用优选策略(xxx Priority)计算每个候选节点的积分，积分最高者胜出。

1.3.4 Kubelet

Kubernetes集群中，每个Node节点上都会启动一个Kubelet服务进程。该进程用于处理Master节点下发到本节点的任务，管理Pod及Pod中的容器。每个Kubelet进程会在API Server上注册节点自身信息，定期向Master节点汇报节点资源的使用情况，并通过cAdvise监控容器和节点资源。

Kubelet功能如下：

- 节点管理：Kubelet在启动时通过API Server注册节点信息，并定时向API Server发送节点新消息，API Server在接收到这些信息后，将这些信息写入etcd。
- Pod管理：Kubelet可以获取自身Node上所要运行的Pod清单；所有针对Pod的操作都会被Kubelet监听到。接收apiserver的HTTP请求，汇报pod的运行状态。
- 容器健康检查：
  - LivenessProbe探针：用于判断容器是否健康，告诉Kubelet一个容器什么时候处于不健康的状态。
  - ReadinessProbe探针：用于判断容器是否启动完成，且准备接受请求。
- cAdvisor资源监控：开源的分析容器资源使用率和性能特性的代理工具，即监控agent。http://172.21.101.103:4194

1.3.5 Kube-Proxy

KUbernetes集群中的每个Node上都运行一个Kube-Proxy进程。该kube-proxy进程： - 可以看做是一个Service的透明代理兼负载均衡器。 - 核心功能是是将某个Service的访问转发到后端的多个Pod实例上。

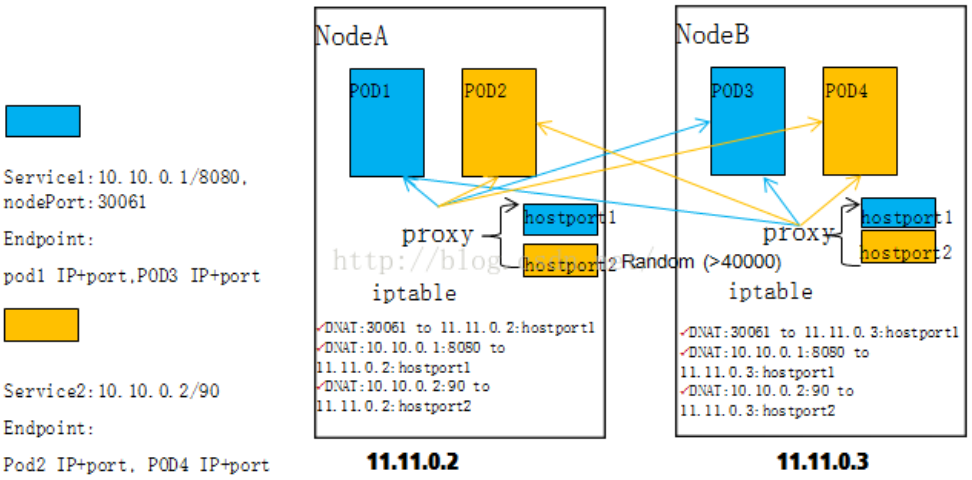
基本原理：

- 对于每一个TCP类型的Kubernetes Service，kube-proxy都会在本地Node上建立一个Socket Server负责可接受请求，然后默认采用Round Robin负载均衡算法均匀发送到后端某个Pod上。

- Kubernetes可以通过修改Service的service.spec.session.sessionAffinity参数的值实现回话保持特性的定向转发，如果设置的值为“ClientIP”，则将来自同一个ClientIP的请求都转发到同一个后端Pod上。

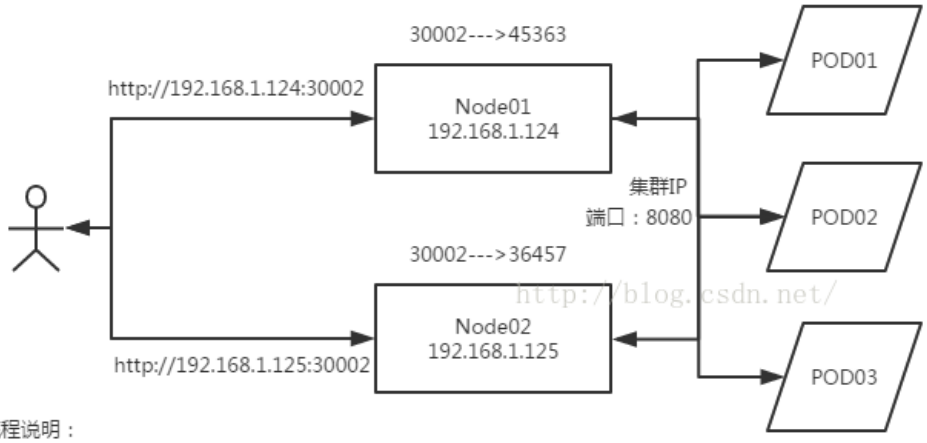
实例原理讲解：

- **实例1：内部服务访问-:port**



- **实例2：外部服务访问-:nodePort**

以nodePort的方式为例，说明iptables的规则

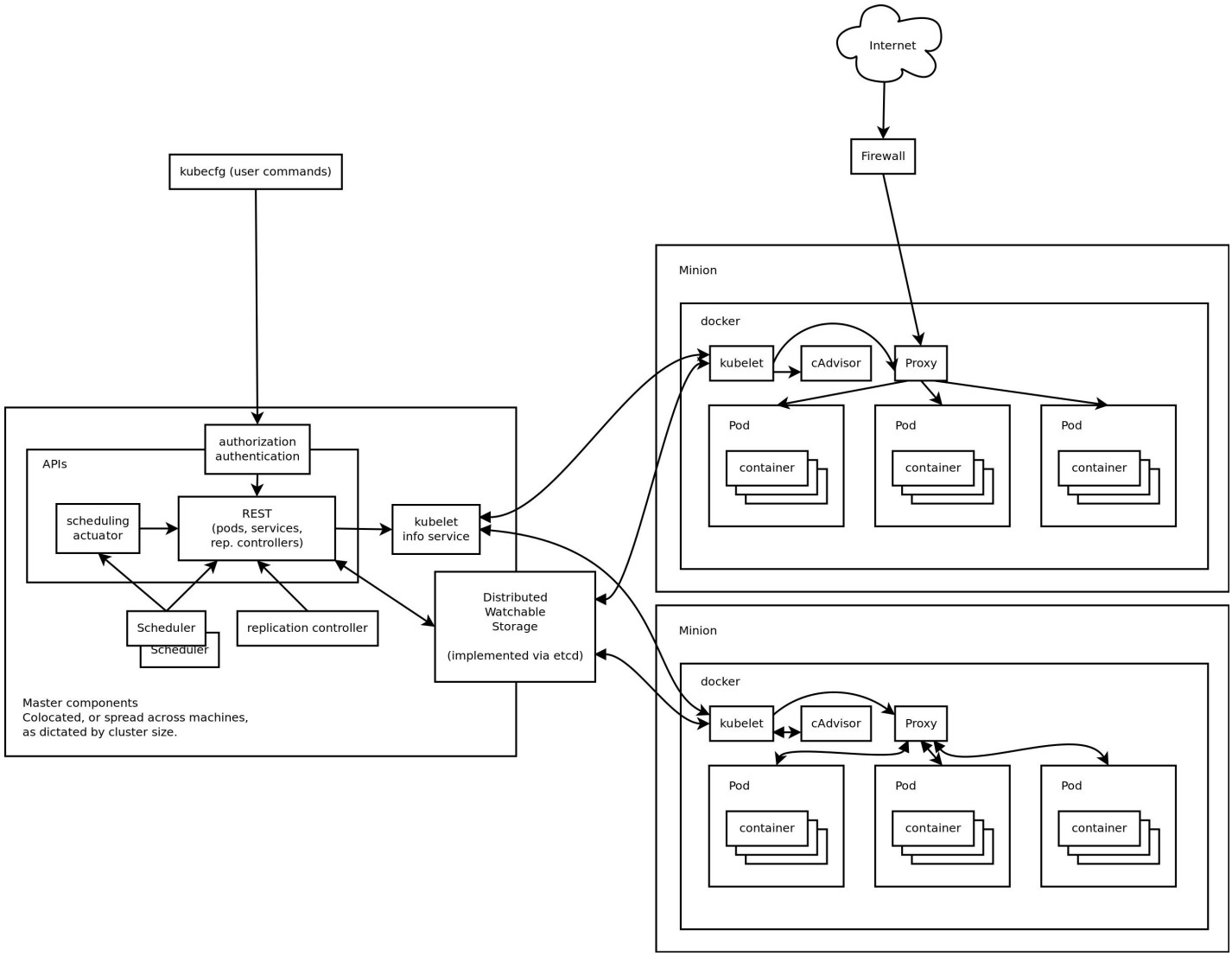


图解：
1、service 对外访问端口为30002
2、45363、36457为kube-proxy分配到本机的随机端口
3、8080为pod真实服务的端口

流程说明：
1、外部访问service 192.168.1.124:30002，然后根据iptables的规则，重定向到45587。外部client启动随机端口与45587连接。
tcp6 0 0 192.168.1.124:45587 192.168.1.121:35845 ESTABLISHED 13101/kube-proxy
2、kube-proxy会在本机启动一个随机端口，与service分配的pod完成连接。
tcp 0 0 192.168.1.124:40614 172.18.0.9:80 ESTABLISHED 13101/kube-proxy

node相当于一个反向代理的意思
外部client<--------->node<--------->service(pod)

1.4 工作方式



1.4.1 kubectl命令行

kubectl是KUbernetes提供的命令行工具(CLI)，用于可直接通过kubectl以命令行的方式同集群交互。

常用命令包括：

- `kubectl --help`：可获得命令的帮助信息
- `kubectl get`: get命令用于获取集群的一个或一些resource信息
- `kubectl describe`: describe获得的是resource集群相关的信息
- `kubectl create`: 用于根据文件或输入创建集群resource
- `kubectl replace`: 用于对已有资源进行更新、替换
- `kubectl patch`: 在容器运行时，直接对容器进行修改
- `kubectl delete`: 根据resource名或label删除resource
- `kubectl logs`: 用于显示pod运行中，容器内程序输出到标准输出的内容
- `kubectl rolling-update`: 已经部署并且正在运行的业务的滚动升级
- `kubectl scale`: 用于程序在负载加重或缩小时副本进行扩容或缩小
- ...

1.4.2 直接使用APIServer交互

kubernetes通过kube-apiserver作为整个集群管理的入口。Apiserver是整个集群的主管理节点，用户通过Apiserver配置和组织集群，同时集群中各个节点同etcd存储的交互也是通过Apiserver进行交互。

Apiserver实现了一套RESTfull的接口，用户可以直接使用API同Apiserver交互。例如可以基于Java语言开发的开发项目OSGI或Fabric8，使用Java程序访问Kubernetes。事实上，kubectl命令也是通过和APIServer交互实现管理Kubernetes集群的目的。

1.5 基本安装步骤

详见POC文档，“CentOS 7 Kubernetes 安装指南”

1.6 使用演示

- 演示一： Guestbook-Example
- 演示二： Kubernetes Dashboard
- 演示三： Kubernetes DNS
- 演示四： Kubernetes 集群监控
- 演示五： KUbernetes 日志监控