k8s基础与相关测试点

1. Docker的简介

官方的解释：容器就是将软件打包成标准化单元，以用于开发、交付和部署。

1.1 图解物理机、虚拟机与容器

物理机

虚拟机：

容器：

> 通过上面这三张抽象图，我们可以大概可以通过类比概括出： **容器虚拟化的是操作系统而不是硬件，容器之间是共享同一套操作系统资源的。虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统。因此容器的隔离级别会稍低一些。**

1.2 docker核心概念

1. **镜像（Image）一个特殊的文件系统**：类似于虚拟机中的镜像，是一个包含有文件系统的面向Docker引擎的只读模板。任何应用程序运行都需要环境，而镜像就是用来提供这种运行环境的。例如一个Ubuntu镜像就是一个包含Ubuntu操作系统环境的模板，同理在该镜像上装上Apache软件，就可以称为Apache镜像。
2. **容器（Container）镜像运行时的实体**：类似于一个轻量级的沙盒，可以将其看作一个极简的Linux系统环境（包括root权限、进程空间、用户空间和网络空间等），以及运行在其中的应用程序。Docker引擎利用容器来运行、隔离各个应用。容器是镜像创建的应用实例，可以创建、启动、停止、删除容器，各个容器之间是是相互隔离的，互不影响。注意：镜像本身是只读的，容器从镜像启动时，Docker在镜像的上层创建一个可写层，镜像本身不变。
3. **仓库（Repository）集中存放镜像文件的地方**：类似于代码仓库，这里是镜像仓库，是Docker用来集中存放镜像文件的地方。注意与注册服务器（Registry）的区别：注册服务器是存放仓库的地方，一般会有多个仓库；而仓库是存放镜像的地方，一般每个仓库存放一类镜像，每个镜像利用tag进行区分，比如Ubuntu仓库存放有多个版本（12.04、14.04等）的Ubuntu镜像。

1.3 基础使用方法

镜像：

- 获取镜像：docker pull IMAGE_NAME
- 列举镜像：docker images
- 删除镜像：docker rmi IMAGE-NAME

容器：

- 运行容器：docker run -d --name NAME
- 列举所有容器：docker ps -a
- 删除容器：docker rm CONTAINER_NAME

打包镜像：

- 容器打包为镜像：docker commit -p CONTAINER:TAG
- 镜像打包为文件：docker save -o FILE_NAME IMAGE_NAME
- 加载文件为镜像：docker load -i FILE_NAME

镜像仓库：

注意改镜像名

docker push

docker pull

容器挂载卷：

docker run -d --name web1 -v /home:/data bx:1.2

容器通信：

- 通过docker网桥互连

创建一个网桥：docker network create -d bridge my-net

创建一个依赖该网桥的容器：docker run -it --rm --network my-net --name t1 busybox

- 共享容器ip

创建一个基础容器：docker run -it --rm --name t3 busybox

使用上一个容器的ip：docker run -it --rm --network container:t3 --name t4 busybox

- 使用宿主机ip

docker run -it --rm --network host --name t5 busybox

- 端口映射

docker run -itd --name t6 -p 8088:80 busybox

Dockerfile

> Dockerfile 是一个文本格式的配置文件，用户可以使用 Dockerfile 快速创建自定义的镜像

```
FROM busybox:latest

ENV CONTENT="Busybox httpd server"

COPY ./Dockerfile /root
ADD ./kubernetes.tar.gz /root
# ADD url /root
# WORKDIR
# ENTRYPOINT
# USER
HEALTHCHECK --interval=5s --timeout=3s \
  CMD wget -O - -q http://127.0.0.1 || exit 1
# ...

RUN mkdir /data/html -p \
    && echo $CONTENT >> /data/html/index.html

VOLUME /data

EXPOSE 80

CMD ["httpd","-f","-h","/data/html"]

# docker build -t bx:1.2 .
# docker run -itd --rm --name web1 -p 9003:80 bx:1.2

```

2. kubenet简介

2.1 Kubernetes是什么

> Kubernetes是容器集群管理系统，是一个开源的平台，可以实现容器集群的自动化部署、自动扩缩容、维护等功能

通过Kubernetes你可以：

- 快速部署应用
- 快速扩展应用
- 无缝对接新的应用功能
- 节省资源，优化硬件资源的使用

  我们的目标是促进完善组件和工具的生态系统，以减轻应用程序在公有云或私有云中运行的负担

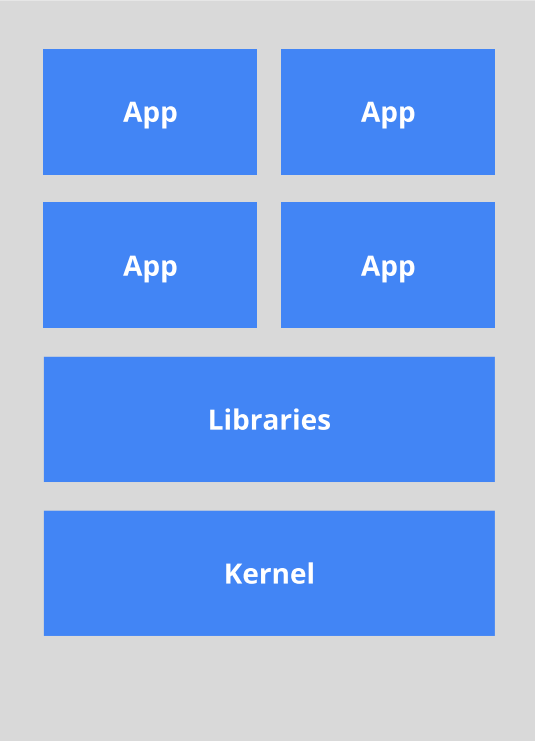
2.2 Kubernetes 特点

- **可移植**: 支持公有云，私有云，混合云，多重云（multi-cloud）
- **可扩展**: 模块化, 插件化, 可挂载, 可组合
- **自动化**: 自动部署，自动重启，自动复制，自动伸缩/扩展

2.3 Why containers?

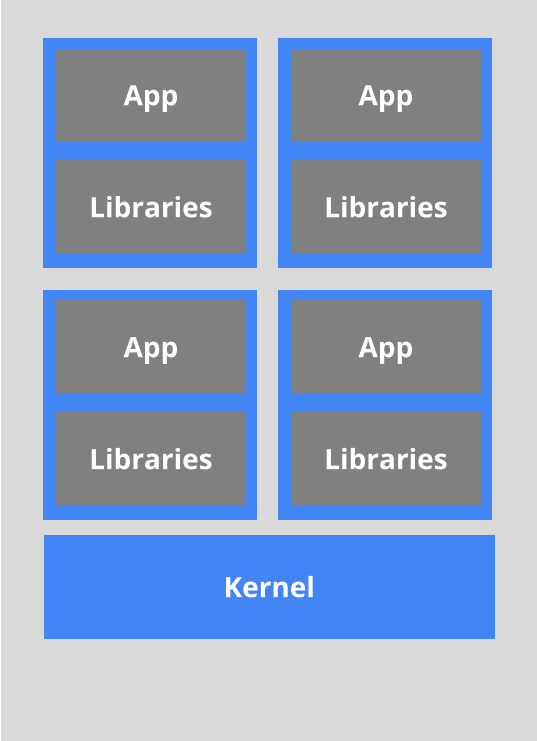
为什么要使用容器？通过以下两个图对比:

The old way: Applications on host



Heavyweight, non-portable
Relies on OS package manager

The new way: Deploy containers



Small and fast, portable
Uses OS-level virtualization

传统的应用部署方式是通过插件或脚本来安装应用。这样做的缺点是应用的运行、配置、管理、所有生存周期将与当前操作系统绑定，这样做并不利于应用的升级更新/回滚等操作，当然也可以通过创建虚机的方式来实现某些功能，但是虚拟机非常重，并不利于可移植性。

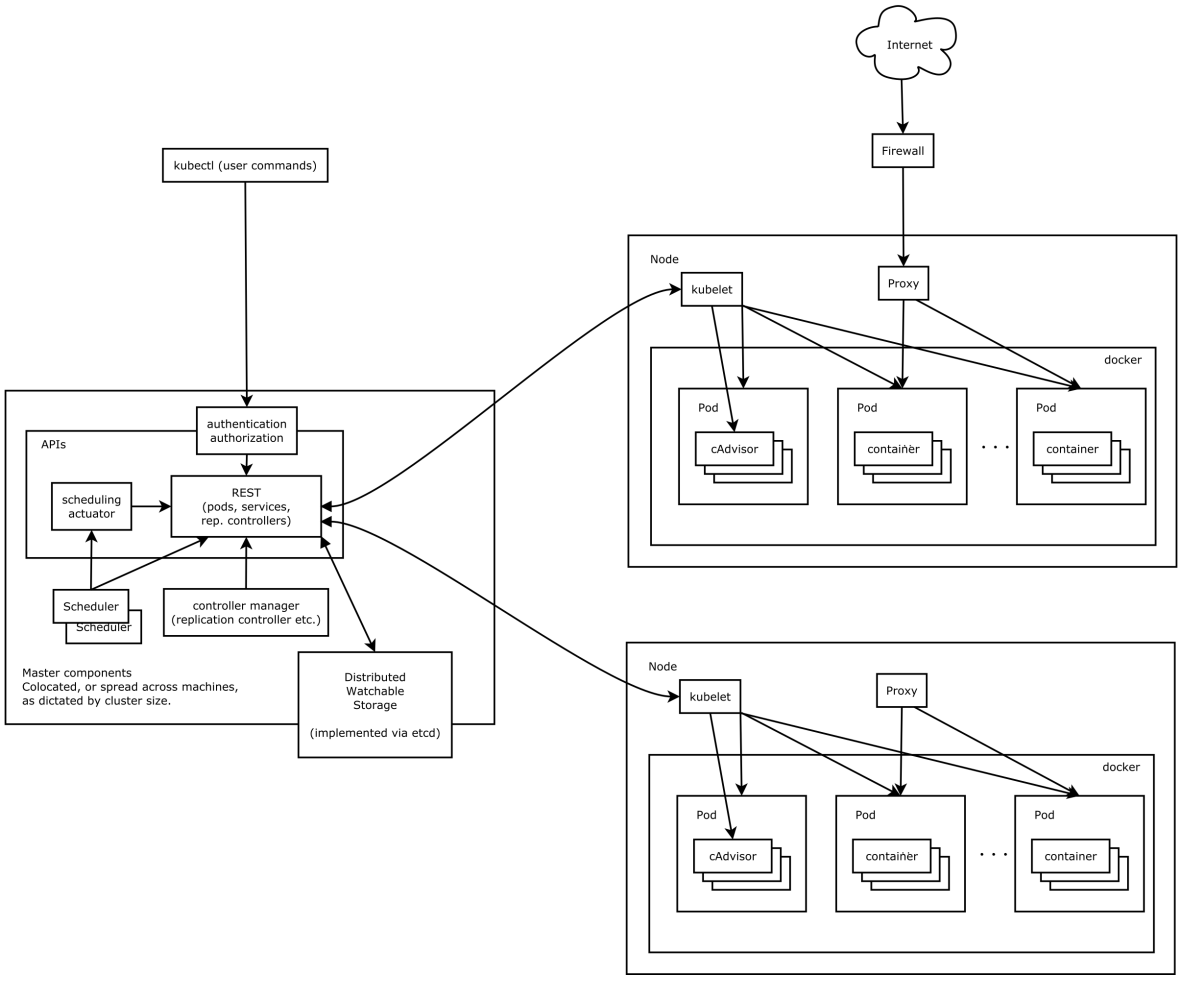
新的方式是通过部署容器方式实现，每个容器之间互相隔离，每个容器有自己的文件系统 ，容器之间进程不会相互影响，能区分计算资源。相对于虚拟机，容器能快速部署，由于容器与底层设施、机器文件系统解耦的，所以它能在不同云、不同版本操作系统间进行迁移。

容器占用资源少、部署快，每个应用可以被打包成一个容器镜像，每个应用与容器间成一对一关系也使容器有更大优势，使用容器可以在build或release 的阶段，为应用创建容器镜像，因为每个应用不需要与其余的应用堆栈组合，也不依赖于生产环境基础结构，这使得从研发到测试、生产能提供一致环境。类似地，容器比虚机轻量、更“透明”，这更便于监控和管理。最后，

容器优势总结：

- **快速创建/部署应用**：与VM虚拟机相比，容器镜像的创建更加容易。
- **持续开发、集成和部署**：提供可靠且频繁的容器镜像构建/部署，并使用快速和简单的回滚(由于镜像不可变性)。
- **开发和运行相分离**：在build或者release阶段创建容器镜像，使得应用和基础设施解耦。
- **开发，测试和生产环境一致性**：在本地或外网（生产环境）运行的一致性。
- **云平台或其他操作系统**：可以在 Ubuntu、RHEL、 CoreOS、on-prem、Google Container Engine或其它任何环境中运行。
- **Loosely coupled，分布式，弹性，微服务化**：应用程序分为更小的、独立的部件，可以动态部署和管理。
- **资源隔离**
- **资源利用**：更高效

2.4 基础架构



在这张系统架构图中，我们把服务分为运行在工作节点上的服务和组成集群级别控制板的服务。

Kubernetes节点有运行应用容器必备的服务，而这些都是受Master的控制。

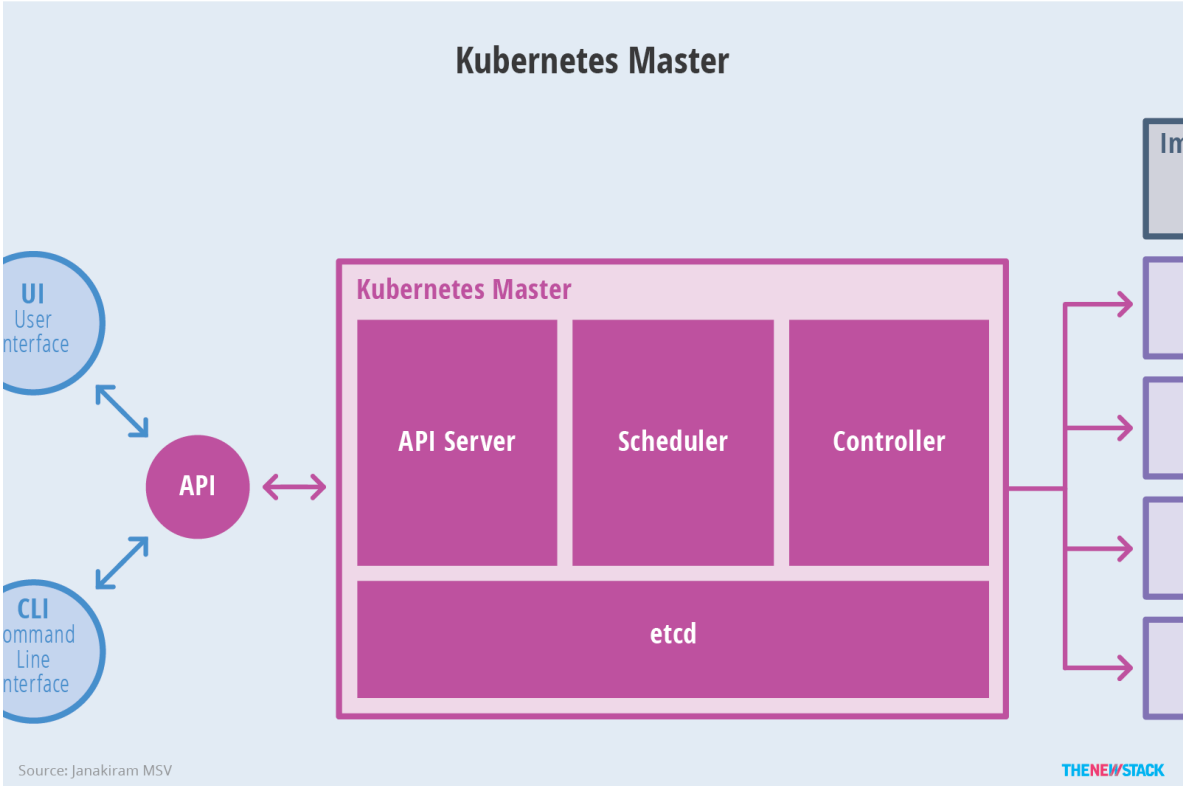
每次个节点上当然都要运行Docker。Docker来负责所有具体的映像下载和容器运行。

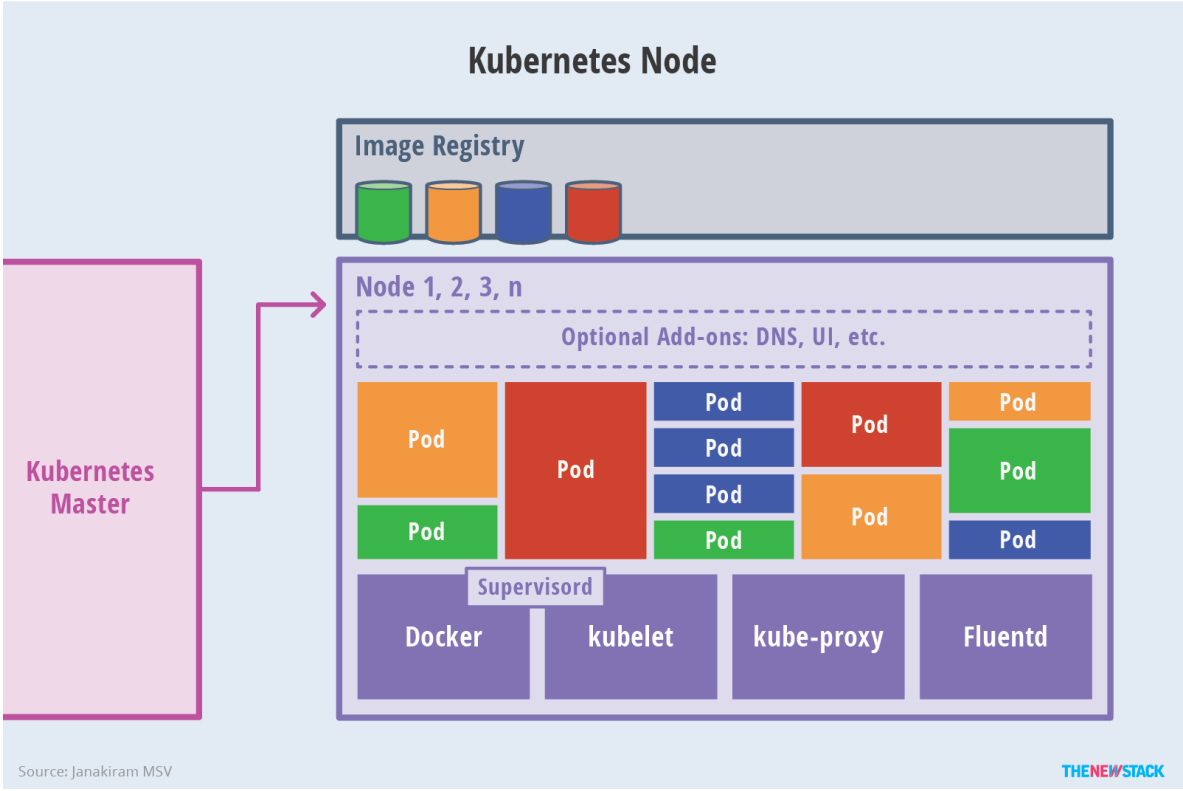
Kubernetes主要由以下几个核心组件组成：

- etcd保存了整个集群的状态；
- apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
- controller manager负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
- kubelet负责维护容器的生命周期，同时也负责Volume（CVI）和网络（CNI）的管理；
- Container runtime负责镜像管理以及Pod和容器的真正运行（CRI）；
- kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

除了核心组件，还有一些推荐的Add-ons：

- kube-dns负责为整个集群提供DNS服务
- Ingress Controller为服务提供外网入口
- Heapster提供资源监控
- Dashboard提供GUI
- Federation提供跨可用区的集群
- Fluentd-elasticsearch提供集群日志采集、存储与查询





3. 部署应用

3.1 创建你的 Node.js 应用

首先要写这个程序。保存代码到 “hellonode/” 文件夹下的server.js

```
var http = require('http');
var handleRequest = function(request, response) {
  console.log('Received request for URL: ' + request.url);
  response.writeHead(200);
  response.end('Hello World!');
};
var www = http.createServer(handleRequest);
www.listen(8080);
```

现在运行这个简单的命令

`node server.js`

3.2 创建一个 Docker 容器镜像

也在`hellonode/`文件夹中 创建一个叫`Dockerfile`的文件. 一个 Dockerfile 定制你想要构建的镜像。 Docker 容器镜像可以基于其他镜像进行拓展，所以我们这个镜像将要基于已经存在的 Node 镜像来构建

```
FROM node  # 必须为第一个命令，指定基础镜像
EXPOSE 8080 # 暴露的端口
COPY server.js .  # 拷贝文件到指定目录
CMD node server.js # 容器开启后的执行命令
```

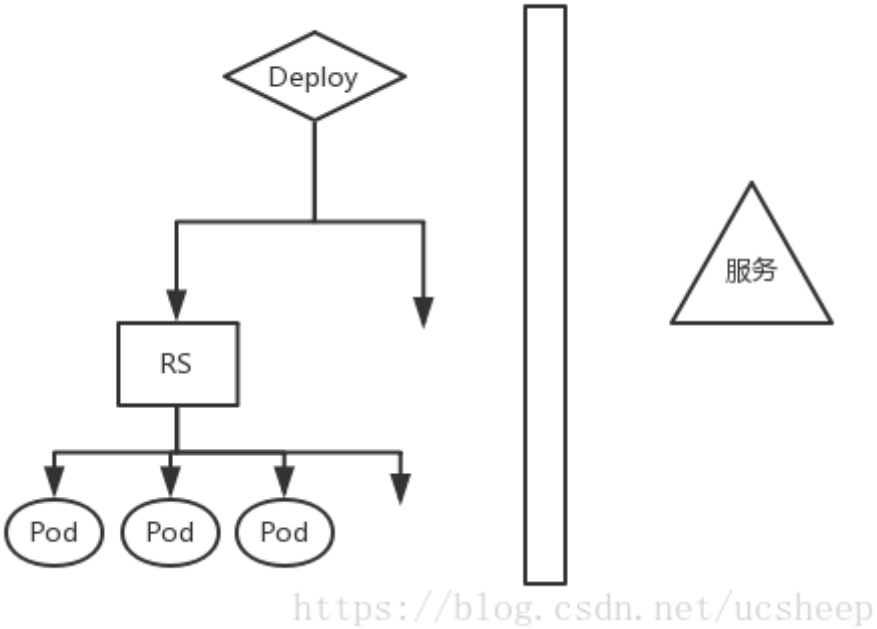
运行`docker build`来构建你的容器

`docker build -t ahpp1001/hello-node:v1 .`

推送镜像到dockerhub（镜像名称前缀改成用户名）

```
docker push ahpp1001/hello-node:v1
```

3.3 Pod,Deployment,ReplicaSet,Service之间关系



> deploy控制RS，RS控制Pod，这一整套，向外提供稳定可靠的Service。

3.4 部署nginx到k8s上（直接通过镜像名创建）

第一步：在master 节点上创建一个deployment

```
kubectl create deployment nginx --image=nginx
```

查看效果

```
kubectl get deployments
```

第二步：创建一个service

```
kubectl create service nodeport nginx --tcp 80:80
```

查看效果

```
kubectl get svc
```

删除

```
kubectl delete deployments/nginx services/nginx
```

3.5 使用yaml文件创建Deployment

使用yaml文件好处

- 便捷性：你将不再需要添加大量的参数到命令行中执行命令
- 可维护性：YAML文件可以通过源头控制，可以跟踪每次的操作
- 灵活性：通过YAML你将可以创建比命令行更加复杂的结构

YAML语法规则

- 大小写敏感
- 使用缩进表示层级关系
- **缩进时不允许使用Tab键，只允许使用空格**
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- "#" 表示注释，从这个字符一直到行尾，都会被解析器忽略

YAML两种结构类型

- Lists

```
args
  - sleep
  - "1000"
  - message
  - "Bring back Firefly!"
```

在JSON格式中，它将表示如下：

```
{
   "args": ["sleep", "1000", "message", "Bring back Firefly!"]
}
```

- Maps

```
---
apiVersion: v1
kind: Pod
metadata:
  name: rss-site
  labels:
    app: web
```

在JSON格式中，它将表示如下：

```
{
   "apiVersion": "v1",
   "kind": "Pod",
   "metadata": {
                "name": "rss-site",
                "labels": {
                            "app": "web"
                          }
              }
}
```

创建一个名为deployment.yaml的yaml文件,内容如下:

```
---
apiVersion: extensions/v1beta1  # 必选 指定api版本 kubectl api-versions
kind: Deployment # 必选 指定创建资源的角色/类型
metadata: # 必选 资源的元数据/属性
  name: kube-busybox # 必选 资源的名字，在同一个namespace中必须唯一
spec:  # Pod中容器的详细定义
```

```
  replicas: 2  # 副本数
  template: # 定义 Pod 的模板
    metadata: # 定义 Pod 的元数据
      labels: # 标签
        app: web
    spec: # 描述 Pod 的规格
      containers:
        - name: busybox-web  # 容器的名字
          image: ahpp1001/bx:1.2 # 容器使用的镜像
          ports:
            - containerPort: 80 # 容器对外开放的端口
```

创建一个deployment

`kubectl create -f deployment.yaml`

查看

`kubectl get pods -o wide`

检查

`kubectl describe pod/POD_NAME`

`curl ip`

3.6 使用yaml文件创建Service（NodePort）

为什么使用service?

Pod是有生命周期的，使用凡人皆有一死来描述pod很贴切，当一个工作节点(node)销毁时，节点上运行的pods也会被销毁， ReplicationController会动态地在其他节点上创建Pod来保持应用程序的运行，每一个Pod都有一个独立的IP地址，甚至是同一个节点上的Pod，可以看出Pod的IP是动态的，它随Pod的创建而创建，随Pod的销毁而消失，这就引出一个问题：如果由一组Pods组合而成的集群来提供服务，那如何访问这些Pods呢?

Kubenetes的Service就是用来解决这个问题的。一个**Service可以看作一组提供相同服务的Pods的对外访问接口**，Service作用于哪些Pods是通过label selector来定义的，这些Pods能被Service访问，Pod之间的发现和路由（如应用中的前端和后端组件）由Kubernetes Service处理。

Service有四种type: ClusterIP(默认）、NodePort、LoadBalancer、ExternalName. 其中NodePort和LoadBalancer两类型的Services可以对外提供服务。

这里使用yaml文件来创建NodePort类型的Service，service.yaml文件内容如下:

```
---
apiVersion: v1
kind: Service
metadata:
  name: kube-busybox-service
  labels:
    name: kube-busybox-service
spec:
  type: NodePort       #这里代表是NodePort类型的
  ports:
  - port: 80           #这里的端口和clusterIP(10.97.114.36)对应，即10.97.114.36:80,供内部访问。
    targetPort: 80  #端口一定要和container暴露出来的端口对应
    protocol: TCP
```

```
    nodePort: 32143    # 所有的节点都会开放此端口，此端口供外部调用。
  selector:
    app: web           #这里选择器一定要选择容器的标签
```

创建service：

`kubectl create -f service.yaml`

查看

`kubectl get services`

还另一种创建service的方式，更快更便捷，即使用expose命令来创建service

`kubectl expose deployment kube-node --type=NodePort`

3.7 使用yaml文件创建Service（LoadBalancer）

继续expose命令创建loadBalancer类型的service，命令如下：

`kubectl expose deployment kube-node --type=LoadBalancer`

也可以使用yaml文件来创建，service-lb.yaml文件如下：

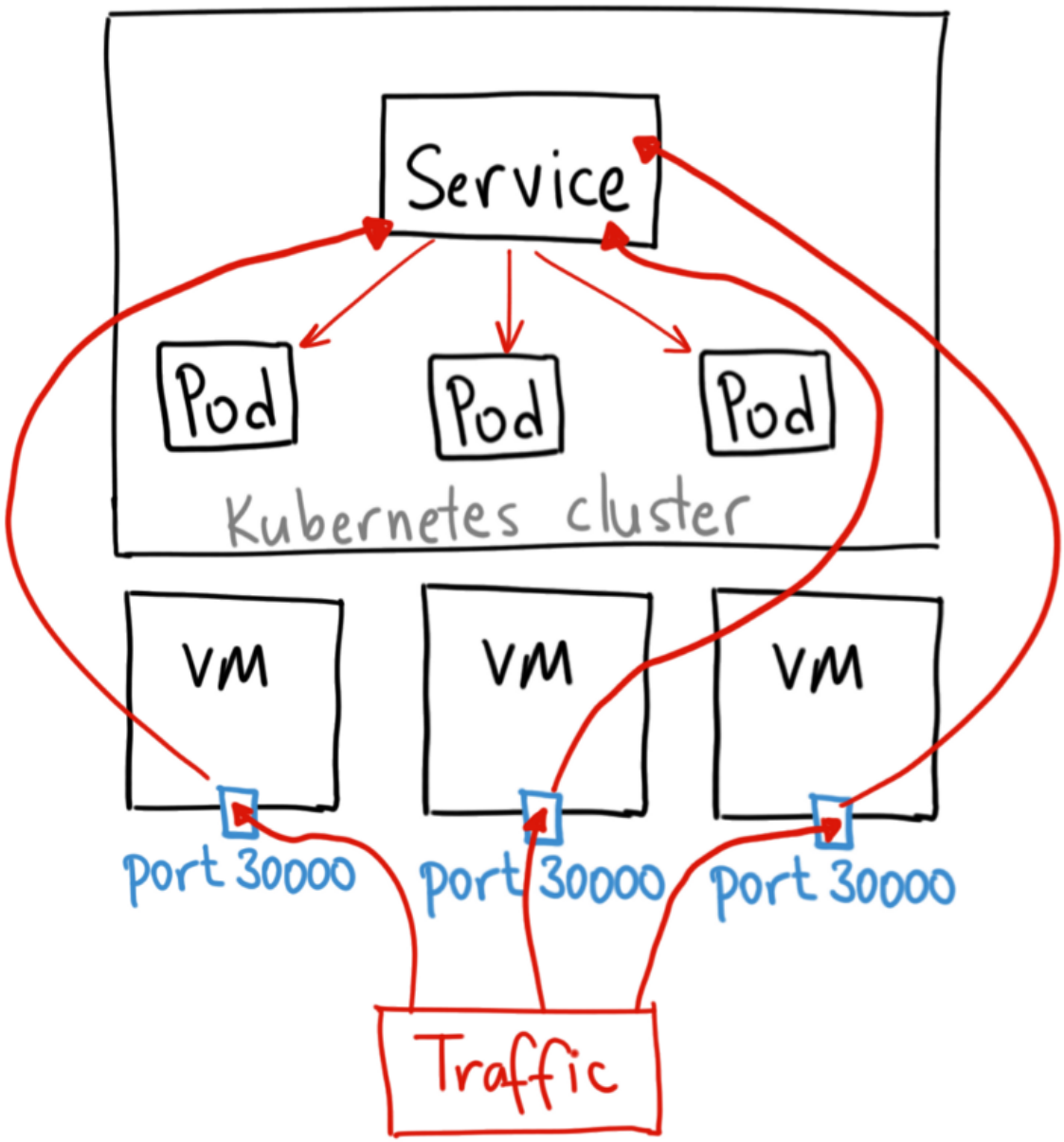
```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: kube-busybox-service-lb
  labels:
    name: kube-busybox-service-lb
spec:
  type: LoadBalancer
  clusterIP: 10.1.0.13
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    nodePort: 32145
  selector:
    app: web
status:
  loadBalancer:
    ingress:
    - ip: 192.168.50.130     #这里是云服务商提供的负载匀衡器的IP地址
```

执行命令：

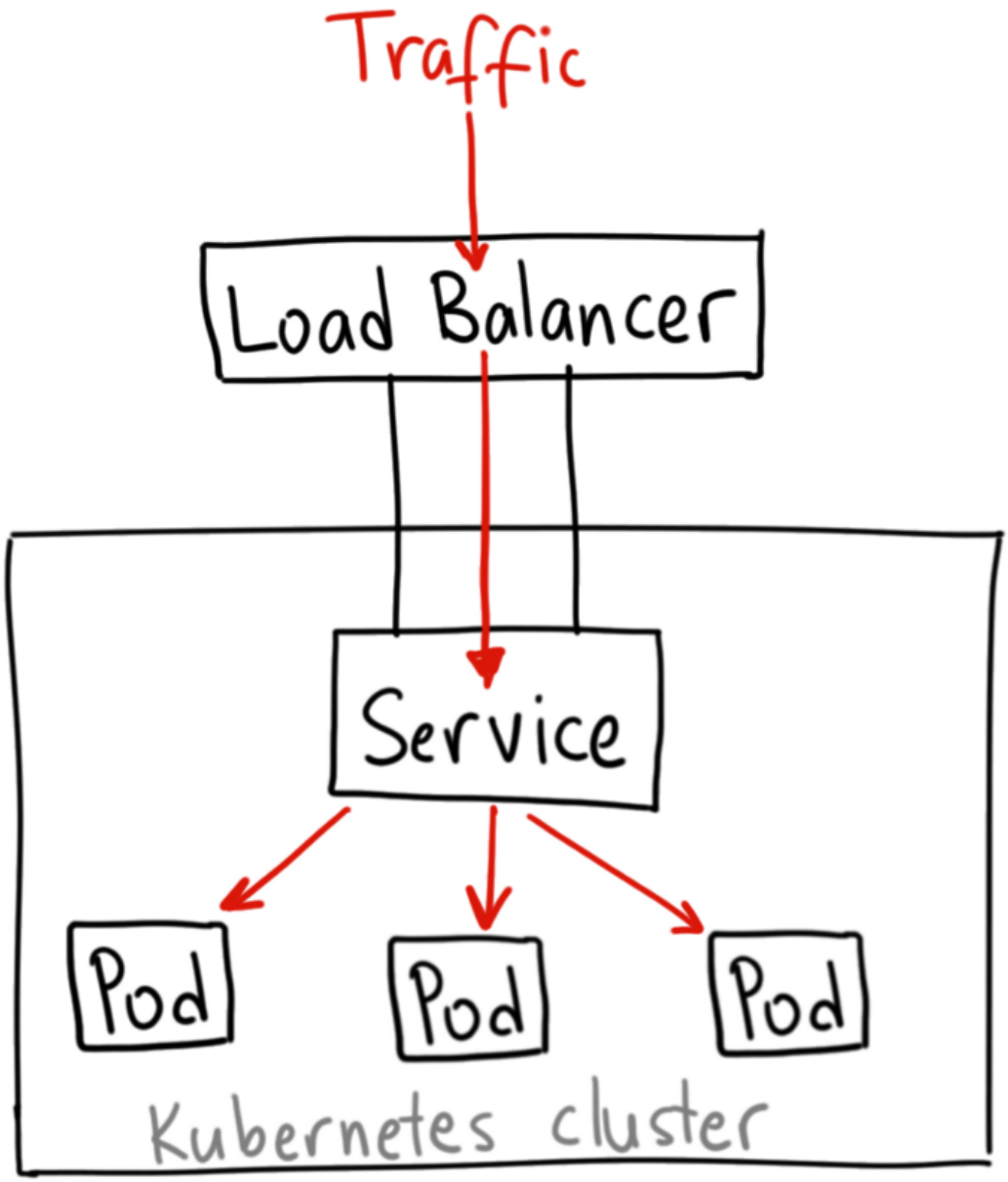
`kubectl create -f service-lb.yaml`

NodePort和LoadBalancer区别：

> NodePort 服务是引导外部流量到你的服务的最原始方式。NodePort，正如这个名字所示，在所有节点（虚拟机）上开放一个特定端口，任何发送到该端口的流量都被转发到对应服务



> LoadBalancer 服务是暴露服务到 internet 的标准方式。在 GKE 上，这种方式会启动一个 Network Load Balancer，它将给你一个单独的 IP 地址，转发所有流量到你的服务。



4. 测试点

- 部署
  - 离线部署
  - 查看容器运行状态
  - 检查功能是否正常
- 稳定性和高可用
  - 节点离线
  - 服务崩溃
- 扩容和缩容
- 更新升级
  - 研发过程中的迭代升级
  - 客户版本升级
- 资料备份迁移

5. 常见运维方法

5.1 排查Pods的故障

检查Pod的问题首先应该了解Pod所处的状况。下面这个命令能够获得Pod当前的状态和近期的事件列表：

```
$ kubectl describe pods ${POD_NAME}
```

确认清楚在Pod以及其中每一个容器的状态，是否都处于`Runing`？通过观察容器的已运行时间判断它是否刚刚才重新启动过？

根据不同的运行状态，用户应该采取不同的调查措施。

- **Pod始终处于Pending状态**

如果Pod保持在`Pending`的状态，这意味着它无法被正常的调度到一个节点上。通常来说，这是由于某种系统资源无法满足Pod运行的需求。观察刚才`kubectl describe`命令的输出内容，其中应该包括了能够判断错误原因的消息。常见的原因有以下这些：

系统没有足够的资源：你也许已经用尽了集群中所有的CPU或内存资源。对于这种情况，你需要清理一些不在需要的Pod，调整它们所需的资源量，或者向集群中增加新的节点。在计算资源文档有更详细的说明。

用户指定了hostPort：通过hostPort用户能够将服务暴露到指定的主机端口上，但这样会限制Pod能够被调度运行的节点。在大多数情况下，hostPort配置都是没有必要的，用户应该采用Service的方式暴露其对外的服务。如果用户确实必须使用hostPort的功能，那么此Pod最多只能部署到和集群节点相同的数目。

- **Pod始终处于Waiting状态**

    - 请确保正确书写了镜像的名称
    - 请检查所需镜像是否已经推送到了仓库中
    - 手工的在节点上运行一次`docker pull <镜像名称>`检测镜像能否被正确的拉取下来

Pod处在`Waiting`的状态，说明它已经被调度分配到了一个工作节点，然而它无法在那个节点上运行。同样的，在刚才`kubectl describe`命令的输出内容中，应该包含有更详细的错误信息。最经常导致Pod始终`Waiting`的原因是无法下载所需的镜像（译者注：由于国内特殊的网络环境，这类问题出现得特别普遍）。用户可以从下面三个方面进行排查：

- **Pod一直崩溃或运行不正常**

首先，查看正在运行容器的日志。

```
$ kubectl logs <Pod名称> <Pod中的容器名称>
```

如果容器之前已经崩溃过，通过以下命令可以获得容器前一次运行的日志内容。

```
$ kubectl logs --previous <Pod名称> <Pod中的容器名称>
```

此外，还可以使用`exec`命令在指定的容器中运行任意的调试命令。

```
$ kubectl exec <Pod名称> -c <Pod中的容器名称> -- <任意命令> <命令参数列表...>
```

值得指出的是，对于只有一个容器的Pod情况，`-c <Pod中的容器名称>`这个参数是可以省略的。

例如查看一个运行中的Cassandra日志文件内容，可以参考下面这个命令：

```
$ kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

要是上面的这些办法都不奏效，你也可以找到正在运行该Pod的主机地址，然后使用SSH登陆进去检测。但这是在确实迫不得已的情况下才会采用的措施，通常使用Kubernetes暴露的API应该足够获得所需的环境信息。因此，如果当你发现自己不得不登陆到主机上去获取必要的信息数据时，不妨到Kubernetes的GitHub页面中给我们提一个功能需求（Feature Request），在需求中附上详细的使用场景以及为什么当前Kubernetes所提供的工具不能够满足需要。

- **Pod在运行但没有工作**

如果Pod没有按照预期的功能运行，有可能是由于在Pod描述文件中（例如你本地机器的`mypod.yaml`文件）存在一些错误，这些配置中的错误在Pod时创建并没有引起致命的故障。这些错误通常包括Pod描述的某些元素嵌套层级不正确，或是属性的名称书写有误（这些错误属性在运行时会被忽略掉）。举例来说，如果你把`command`属性误写为了`commnd`，Pod仍然会启动，但用户所期待运行的命令则不会被执行。

对于这种情况，首先应该尝试删掉正在运行的Pod，然后使用`--validate`参数重新运行一次。继续之前的例子，当执行`kubectl create --validate -f mypod.yaml`命令时，被误写为`commnd`的`command`指令会导致下面这样的错误：

```
I0805 10:43:25.129850   46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973   46757 schema.go:129] this may be a false alarm, see https://github.com/kubernetes/kubernetes/issues/6842
pods/mypod
```

下一件事是检查当前apiserver运行Pod所使用的Pod描述文件内容是否与你想要创建的Pod内容（用户本地主机的那个yaml文件）一致。比如，执行`kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml`命令将正在运行的Pod描述文件内容导出来保存为`mypod-on-apiserver.yaml`，并将它和用户自己的Pod描述文件`mypod.yaml`进行对比。由于apiserver会尝试自动补全一些缺失的Pod属性，在apiserver导出的Pod描述文件中有可能比本地的Pod描述文件多出若干行，这是正常的，但反之如果本地的Pod描述文件比apiserver导出的Pod描述文件多出了新的内容，则很可能暗示着当前运行的Pod使用了不正确的内容。

5.2 排查Replication Controllers的故障

Replication Controllers的逻辑十分直白，它的作用只是创建新的Pod副本，仅仅可能出现的错误便是它无法创建正确的Pod，对于这种情况，应该参考上面的『排查Pods的故障』部分进行检查。

也可以使用`kubectl describe rc <控制器名称>`来显示与指定Replication Controllers相关的事件信息。

5.3 排查Services的故障

Service为一系列后端Pod提供负载均衡的功能。有些十分常见的故障都可能导致Service无法正常工作，以下将提供对调试Service相关问题的参考。

首先，检查Service连接的服务提供端点（endpoint），对于任意一个Service对象，apiserver都会为其创建一个端点资源（译者注：即提供服务的IP地址和端口号）。

这个命令可以查看到Service的端口资源：

```
$ kubectl get endpoints <Service名称>
```

请检查这个命令输出端点信息中的端口号与实际容器提供服务的端口号是否一致。例如，如果你的Service使用了三个Nginx容器的副本（replicas），这个命令应该输出三个不同的IP地址的端点信息。

- **服务没有端点信息**

如果刚刚的命令显示Service没有端点信息，请尝试通过Service的选择器找到具有相应标签的所有Pod。假设你的Service描述选择器内容如下：

```
...
spec:
  - selector:
     name: nginx
     type: frontend
```

可以使用以下命令找出相应的Pod：

```
$ kubectl get pods --selector=name=nginx,type=frontend
```

找到了符合标签的Pod后，首先确认这些被选中的Pod是正确，有无错选、漏选的情况。

如果被选中的Pod没有问题，则问题很可能出在这些Pod暴露的端口没有被正确的配置好。要是一个Service指定了`containerPort`，但被选中的Pod并没有在配置中列出相应的端口，它们就不会出现在端点列表中。

请确保所用Pod的`containerPort`与Service的`containerPort`配置信息是一致的。

- **网络流量没有正确的转发**

如果你能够连接到Service，但每次连接上就立即被断开，同时Service的端点列表内容是正确的，很可能是因为Kubernetes的kube-proxy服务无法连接到相应的Pod。

请检查以下几个方面：

- Pod是否在正常工作？从每个Pod的自动重启动次数可以作为有用的参考信息，前面介绍过的Pod错误排查也介绍了更详细的方法
- 能够直接连接到Pod提供的服务端口上吗？不妨获取到Pod的IP地址，然后尝试直接连接它，以验证Pod本身十分运行正确。
- 容器中的应用程序是否监听在Pod和Service中配置的那个端口？Kubernetes不会自动的映射端口号，因此如果应用程序监听在8080端口，务必保证Service和Pod的`containerPort`都配置为了8080。

> 如果上述的这些步骤还不足以解答你所遇到的问题，也就是说你已经确认了相应的Service正在运行，并且具有恰当的端点资源，相应的Pod能够提供正确的服务，集群的DNS域名服务没有问题，IPtable的防火墙配置没有问题，kube-proxy服务也都运转正常。