

# K8s 基础入门和实践

---

[K8s 是什么](#)

[K8s 和 Docker](#)

[K8s 大致的样子](#)

[K8s 整体架构图](#)

[主节点](#)

[工作节点](#)

[K8s 里常用的控制器，以及对象的三要素](#)

[对象定义的要三要素](#)

[常用的控制器对象](#)

[Deployment](#)

[StatefulSet](#)

[Service](#)

[ClusterIP Service](#)

[NodePort Service](#)

[Ingress](#)

[DaemonSet](#)

[K8s 本机环境安装](#)

[从零开始部署一个Web APP到 K8s](#)

[把 Web 程序打包成容器镜像](#)

[配置应用的Pod 和 Deployment](#)

[用 Service 暴露服务](#)

[用 Ingress 代理服务](#)

[最后](#)

这里的内容是我自己总结的K8s入门知识和实践练习，在公司做过几次分享，反响还不错，这里是我当时准备的手稿，上课的时候是做了一个课程PPT，不过是公司的模版就不再公开了。

## K8s 是什么

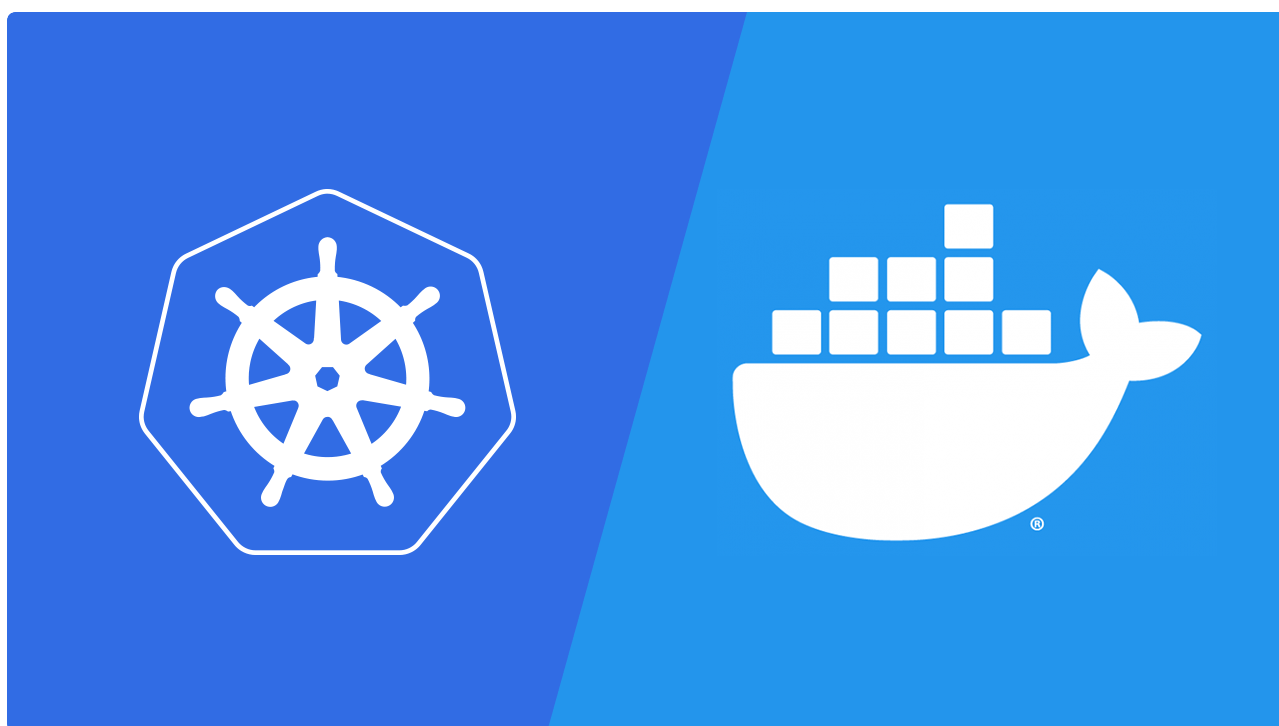
Kubernetes（单词太长，后面用 K8s 代替）是一个基于容器技术的分布式架构方案，它源自 Google 内部大规模集群管理系统——Borg，自 2015 年开源后得到开源社群的全力支援，IBM、惠普、微软、RedHat 等业界巨头纷纷加入，成为后来的 CNCF 组织（Cloud Native Computing Foundation，云原生计算基金会）首个毕业的项目。

K8s 具备完善的集群管理能力，包括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、内建负载均衡器、故障发现和自我修复能力、服务滚动升级和线上扩容、可扩展的资源自动调度机制、多粒度的资源配额管理能力。还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

总结一句话：很牛，很给力，是一套结合了容器编排和集群调度管理的大规模分布式系统解决方案。

## K8s 和 Docker

Docker 和 K8s 这两个经常一起出现，两者的 Logo 看着也有一定联系一个是背上驮着集装箱的鲸鱼一个是船的舵轮。



不过两者不能放在一个维度上讨论，Docker 是当前流行的 Linux 容器解决方案，利用 Namespaces、Cgroups 以及联合文件系统UnionFS 实现了同一主机上容器进程间的相互隔离。

- NameSpaces: 隔离进程，让进程只能访问到本命名空间里的挂载目录、PID、NetWork 等资源
- Cgroups: 限制进程能使用的计算机系统各项资源的上限，包括 CPU、内存、磁盘、网络带宽等等
- 联合文件系统UnionFS: 保存一个操作系统的所有文件和目录，在它基础之上添加应用运行依赖的文件。创建容器进程的时候给进程指定Mount Namespace 把镜像文件挂载到容器里，用 chroot 把进程的 Root目录切换到挂载的目录里，从而让容器进程各自拥有独立的操作系统目录。

而 K8s 是拥有容器编排能力的集群管理解决方案，可以按照应用的定义调度各个运行着应用组件 Docker 容器，但是 Docker 并不是 K8s 对容器的唯一选择，K8s 的 容器运行时支持对接多种容器，比如CoreOS公司的Rkt容器（之前称为Rocket，现更名为Rkt）。

Docker 公司也推出过自己的容器编排工具 Docker Swarm，但是在生产上几乎没人使用，之前有专门做容器云的前同事还说过，Docker Swam 在生产用非常之坑。

Docker Swarm 没有流行起来的深层次的原因就不深究了，从一些IT媒体的报道看，可能的原因是

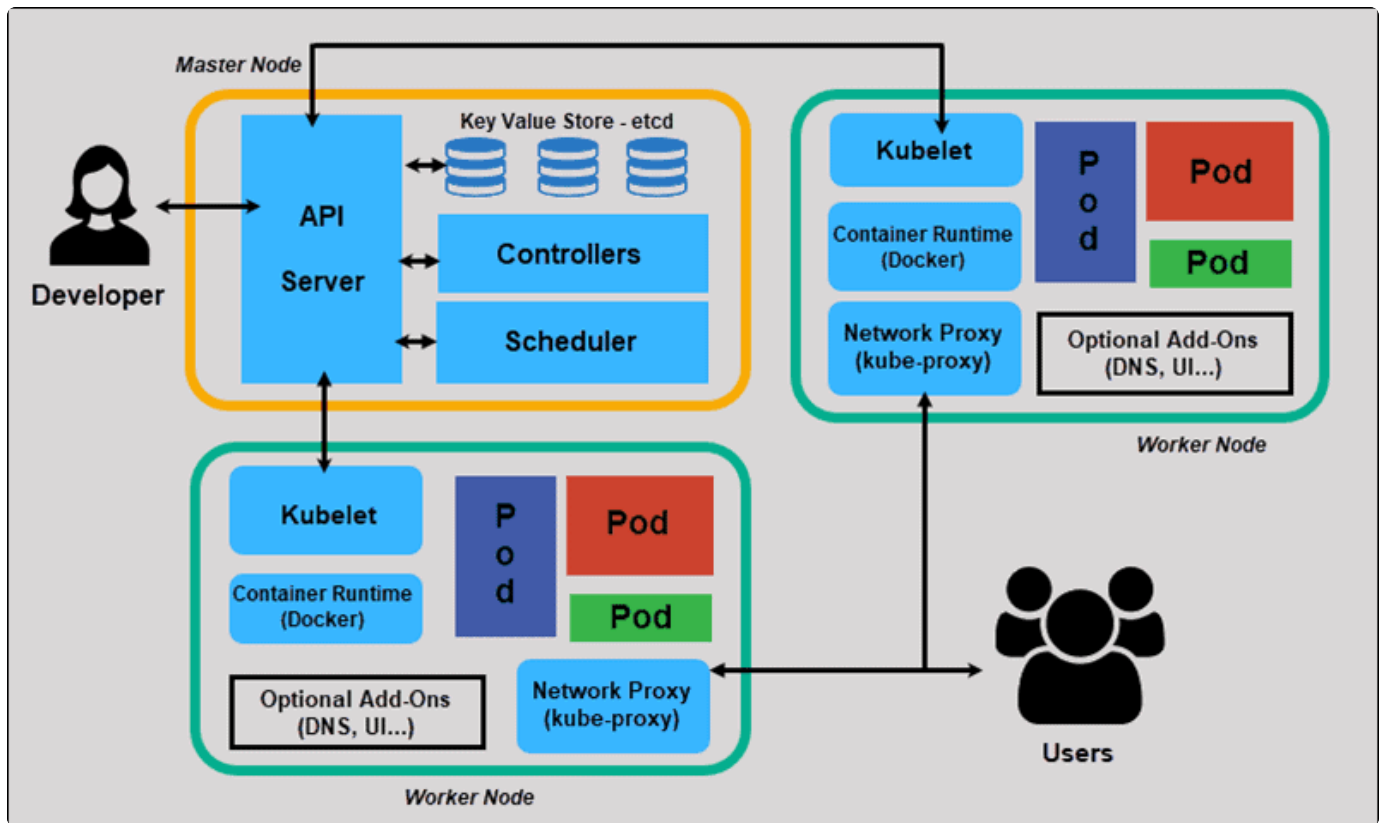
1> 跟 Docker 深度绑定，国外人对集权主义非常反感

2> Docker 公司在大规模集群管理上的经验不足，不像谷歌那样能高屋建瓴地给出好的解决方法

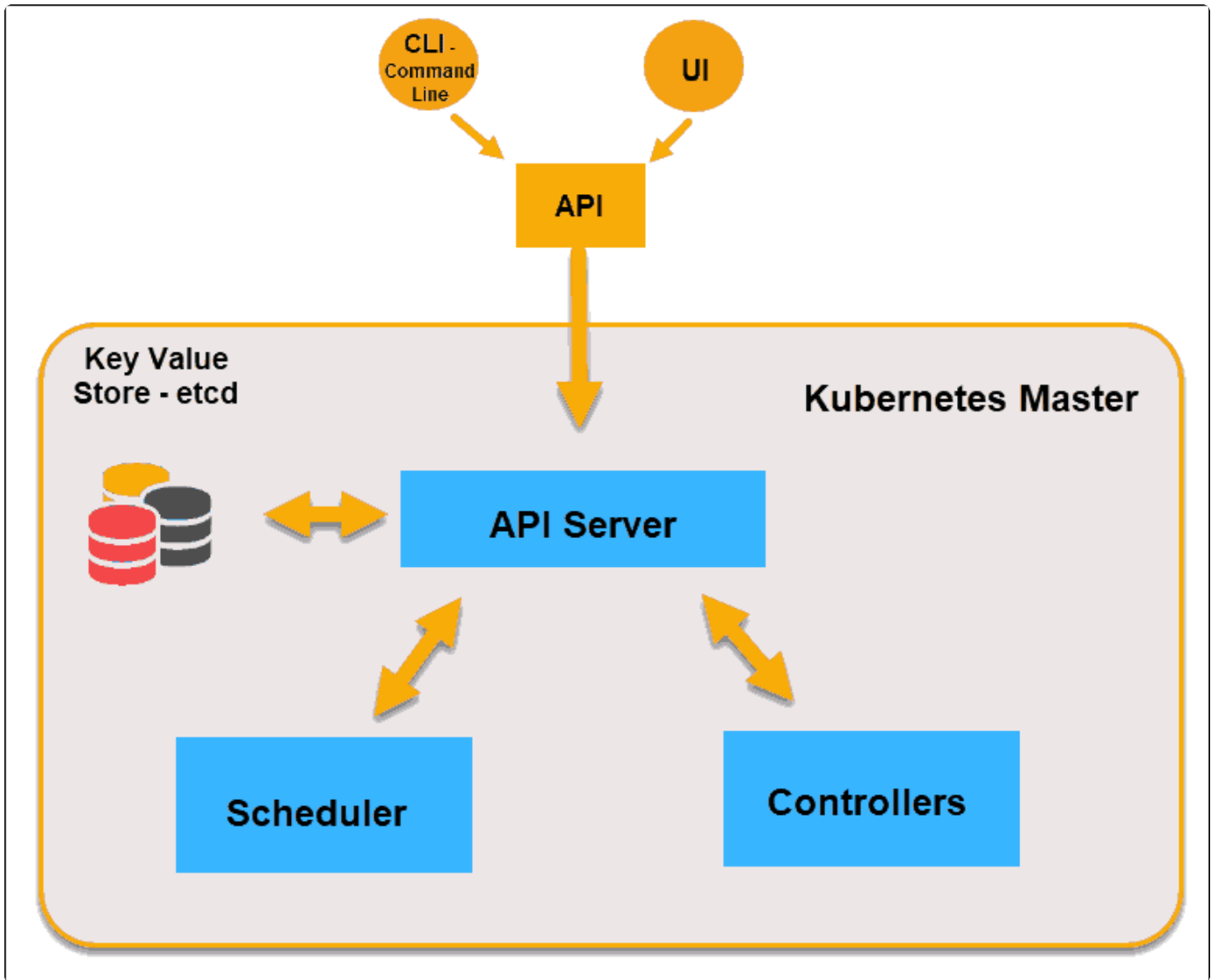
## K8s 大致的样子

K8s 集群采用Master / Work Node（最初称为Minion，后改名Node）的结构，Master Node（主节点）控制整个集群，Work Node（从节点）为集群提供计算能力。使用者可以通过命令行或者 Web 控制台页面的方式来操作集群。

## K8s 整体架构图



主节点



Master 节点是 K8s 集群的大脑，负责向外开放集群的 API，调度和管理整个集群。集群至少要有有一个 Master 节点，如果在生产环境中要达到高可用，还需要配置 Master 集群。

Master 主要包含 API Server、Scheduler、Controller 三个组成部分，以及用作存储的 etcd。用来储存整个集群的状态。

- etcd：由 CoreOS 开发，是一个高可用、强一致性的键值存储，为 Kubernetes 集群提供储存服务，类似于 zookeeper。它会存储集群的整个配置和状态。主节点通过查询 etcd 以检查节点，容器的现状。
- API Server：kubernetes 最重要的核心元件之一，提供资源操作的唯一入口（其他模块通过 API Server 查询或修改资源对象，只有 API Server 才能直接操作 etcd），并提供认证、授权、访问控制、API 注册和发现等机制。
- Scheduler：负责资源的调度，按照预定的调度策略将 Pod（k8s 中调度的基本单位）调度到相应的 Node 上，这里说的 Node 就是 Work Node，当然如果是只有一个节点的集群，Master 也会同时作

为 Work Node。

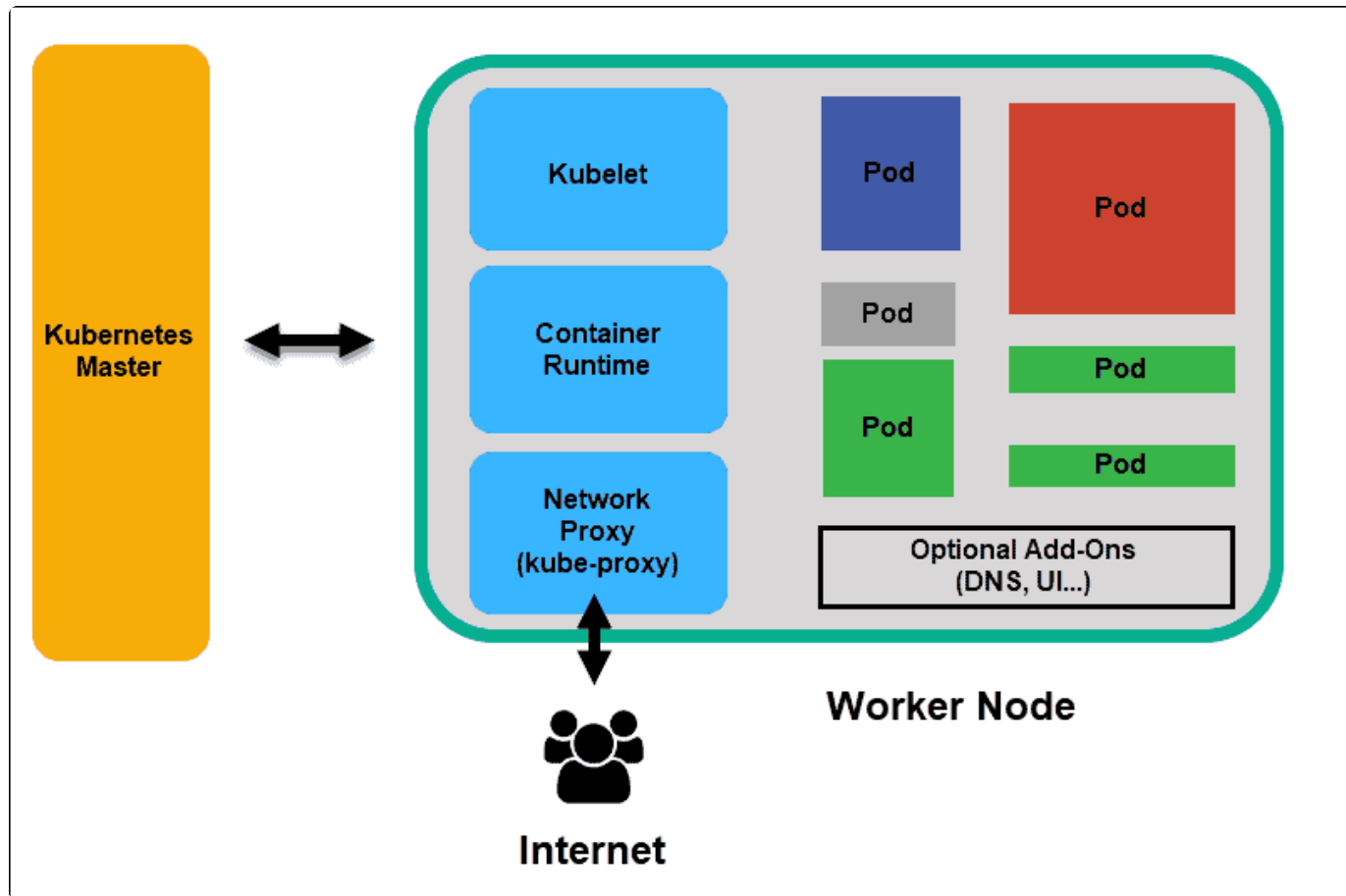
- **Controllers:** 通过 API Server 查询要控制的资源对象的预期状态，它检查其管控的对象的当前状态，确保它们始终处于预期的工作状态，它们的工作包括比如故障检测、自动扩充、减少、滚动更新等。

我们能接触到的控制器有，下面这些，具体干什么的，放到后面介绍完工作节点再说

- Deployment
- StatuefulSet
- Service
- DaemonSet
- Ingress

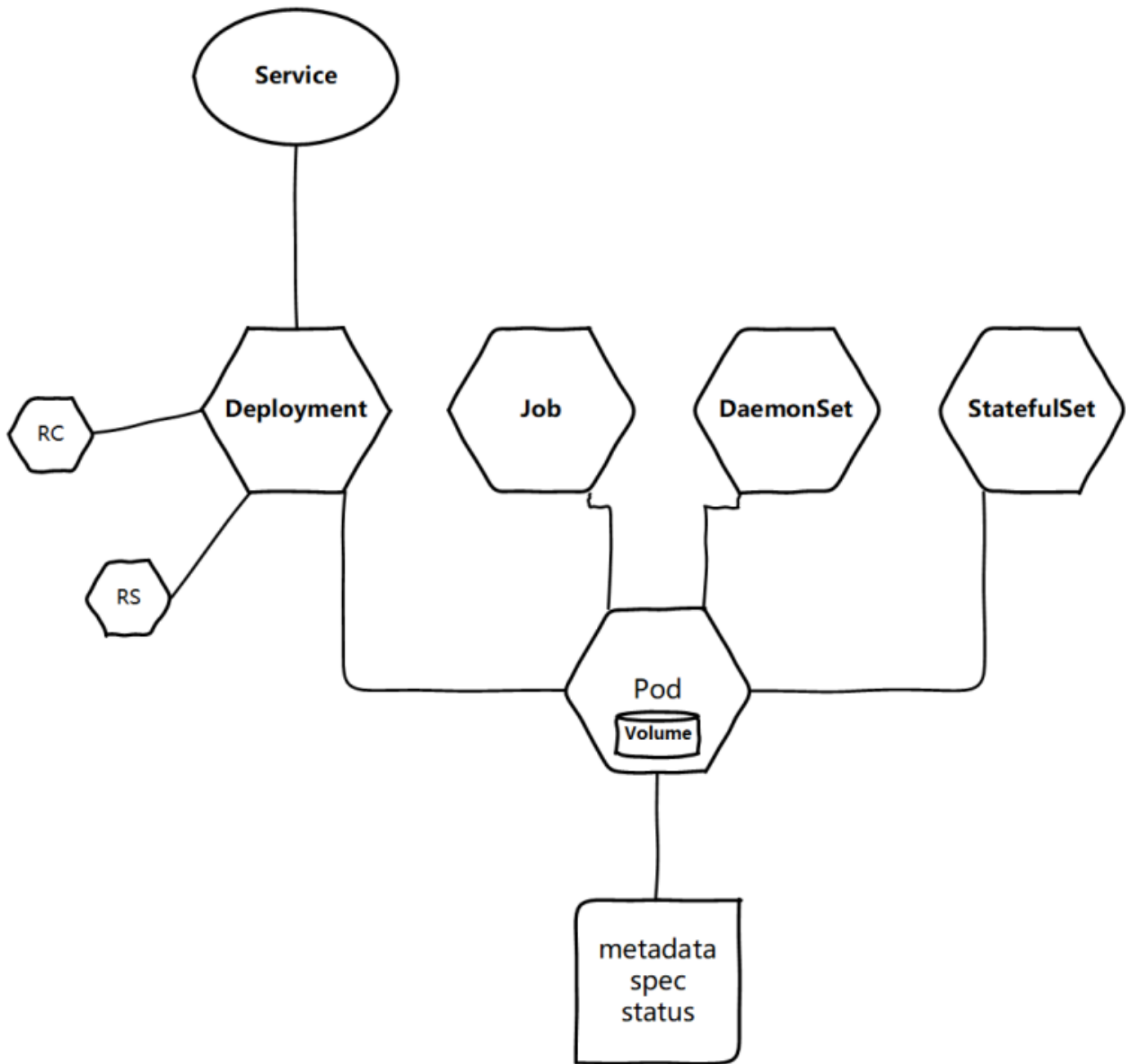
## 工作节点

K8s 集群的工作节点，可以是物理机也可以是虚拟机器。Node 上运行的主要 K8s 组件有kubelet、kube-proxy、Container Runtime 、Pod 等。



- kubelet：K8s 集群的每个工作节点上都会运行一个 kubelet 程序 维护容器的生命周期，它叫接收并执行Master 节点发来的指令，管理节点上的 Pod 及 Pod 中的容器。同时也负责Volume（CVI）和网络（CNI）的管理。每个 kubelet 程序会在 API Server 上注册节点自身的信息，定期向Master 节点汇报自身节点的资源使用情况，并通过cAdvisor监控节点和容器的资源。通过运行 kubelet，节点将自身的 CPU，RAM 和存储等计算机资源编程集群的一部分，相当于是放进了集群统一的资源管理池中，交由 Master 统一调配。
- Container Runtime：容器运行时负责与容器实现进行通信，完成像容器镜像库中拉取镜像，然后启动和停止容器等操作，引入容器运行时另外一个原因是让 K8s 的架构与具体的某一个容器实现解耦，不光是 Docker 能运行在 K8s 之上，同样也让K8s 的发展按自己的节奏进行，想要运行在我的生态里的容器，请实现我的CRI（Container Runtime Interface），Container Runtime 只负责调用CRI 里定义的方法完成容器管理，不单独执行 docker run 之类的操作。这个也是K8s 发现Docker 制约了它的发展在 1.5 后引入的。
- Pod：Pod 是 K8s 中的最小调度单元。我们的应用程序运行在容器里，而容器又被分装在 Pod 里。一个 Pod 里可以有多个容器，也可以有多个容器。没有统一的标准，是单个还是多个，看要运行的应用程序的性质。不过一个 Pod 里只有一个主容器，剩下的都是辅助主容器工作的。比如做服务网格 Istio 的 Envoy 网关，就是放在Pod的辅助容器运行来实现流量控制的。这就是 K8s 的容器设计模式里最常用的一种模式：sidecar。顾名思义，sidecar 指的就是我们可以在一个Pod中，启动一个辅助容器，来完成一些独立于主进程（主容器）之外的工作。
- kube-proxy：为集群提供内部的服务发现和负载均衡，监听 API Server 中 Service 控制器和它后面挂的 endpoint 的变化情况，并通过 iptables 等方式来为 Service 的虚拟IP、访问规则、负载均衡。

## K8s 里常用的控制器，以及对象的三要素



对于 K8s 这个系统来说也是一切皆对象，对象是系统中持久化的实体，它使用对象来表示几种中各种资源，以及他们在集群中的状态。

## 对象定义的要三要素

Pod是K8s 里的最小调度单元，控制器则是管理 Pod 用的，他们在 K8s 系统中都是通过相应的对象来表示的。要创建 / 更改对象时，我们只需要把他们的声明文件（YAML）提交给 K8s 的 API Server，集群就会自动帮我们创建、更新对象。提交给集群的声明YAML 文件中，会写清楚，对象的类型、元信息、以及预期在集群里预期的状态。这就构成了K8s 对象定义的要三要素。



- Kind : 对象种类
- metadata: 对象的元信息, 名字、标签、uid、所在的命名空间等等
- spec: 技术规范, 技术说明: 描述对象的技术细节、各个资源要设置的细节不一样, 以及提交者期望对象达到的理想状态。PS: 所有预期状态的定义都是声明式的 (Declarative) 的而不是命令式 (Imperative), 在分布式系统中的好处是稳定, 不怕丢操作或执行多次。比如设定期望 3 个运行 Nginx 的 Pod, 执行多次也还是一个结果, 而给副本数加1的操作就不是声明式的, 执行多次结果就错了

给大家做个演示, 下面是一个 Pod 对象的定义:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx-pod
spec:
  containers:
    - name: nginx
      image: nginx:1.7.9
      ports:
        - containerPort: 80
```

这是Pod的, 作为一个被调度的单元它里边并没有定义期望的状态, 我们再来看一下控制器 Deployment 的定义, Deployment 是创建、滚动更新 Pod 的控制器, 后面会讲。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx-dp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      labels:
        app: nginx-pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

## 常用的控制器对象

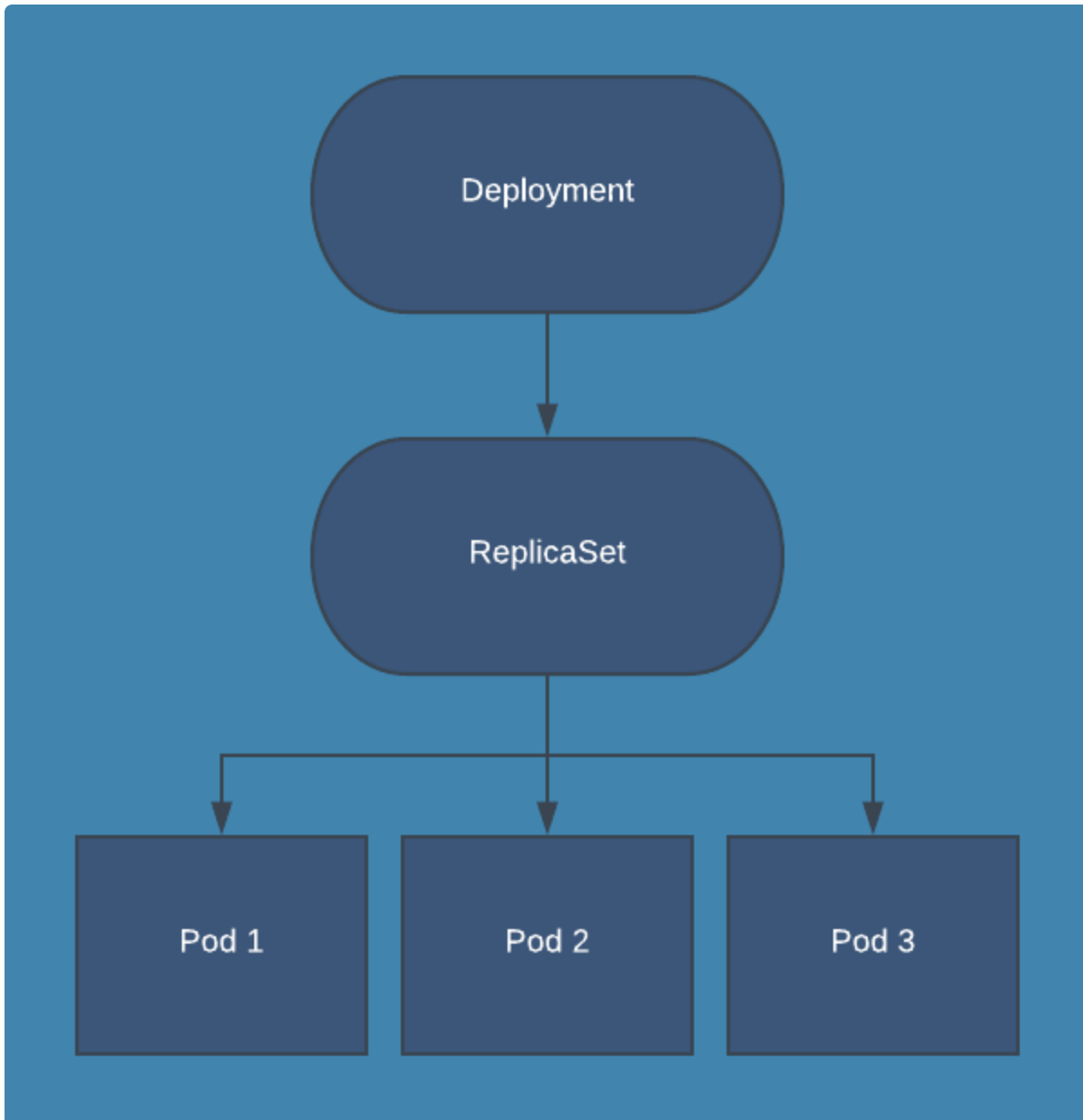
- Deployment
- StatuefulSet
- Service
- DaemonSet
- Ingress

## Deployment

Deployment 控制器用来管理无状态应用的，创建、水平扩容/缩容、滚动更新、健康检查等。为啥叫无状态应用呢，就是它的创建和滚动更新是不保证顺序的，这个特征就特别适合管控运行着 Web 服务的 Pod，因为一个 Web 服务重启、更新并不依赖副本的顺序。不像 MySQL 这样的服务，肯定是要先启动主节点再启动从节点才行。

那种情况我们应该用 StatefulSet 控制器。

Deployment 是一个复合型的控制器，它包装了一个叫做 ReplicaSet -- 副本集的控制器。ReplicaSet 管理正在运行的Pod数量，Deployment 在其之上实现 Pod 滚动更新，对Pod的运行状况进行健康检查以及回滚更新的能力。他们三者之间的关系可以用下面这种图表示。



回滚更新是 Deployment 在内部记录了 ReplicaSet 每个版本的对象，要回滚就直接把生效的版本切回原来的ReplicaSet 对象

ReplicaSet 和 Pod 的定义其实是包含在 Deployment 对象的定义中的，我们再把上一个 Deployment 定义的例子拿过来看一下。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx-dp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      labels:
        app: nginx-pod
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

定义文件里的 `replicas: 3` 代表的就是我期望一个拥有三个副本 Pod 的副本集，而 `template` 这个 YAML 定义也叫做 Pod 模板，意思就是副本集的 Pod，要按照这个样板创建出来。

所以这里有一点需要注意，虽然我们知道了 Pod 是 K8s 里最小的调度单元，ReplicaSet 控制器可以控制 Pod 的可用数量始终保持在想要数量。但是在 K8s 里我们却不应直接定义和操作他们俩。对这两种对象的所有操作都应该通过 Deployment 来执行。这么做最主要的好处是能控制 Pod 的滚动更新。

Deployment 里边有很多配置允许我们制定 Pod 的滚动更新，比如允许在更新期间多创建出来多少个副本，以及最多容忍多少个副本不可用等等。这个滚动更新我们就先不细究了，只要先记住一点，Deployment 对 Pod 的滚动更新是先创建新 Pod，逐步的用新创建的 Pod 替换掉老版本的 Pod。

这里给大家来个动图演示一下 Deployment 对 Pod 的滚动更新。

<https://segmentfault.com/img/bVcMpH5>

## StatefulSet

StatefulSet，是在Deployment的基础上扩展出来的控制器。使用Deployment时多数时候你不会在意Pod的调度方式。但当你需要调度有拓扑状态的应用时，就需要关心Pod的部署顺序、对应的持久化存储、Pod 在集群内拥有固定的网络标识（即使重启或者重新调度后也不会变）这些内容，这个时候，就需要 StatefulSet 控制器实现调度目标。

Web 服务一般不用这个控制器部署，所以关于 StatefulSet 的内容，在这次的课上就不再继续细说了。

## Service

Service 是另一个我们必用到的控制器对象，因为在K8s 里 Pod 的 IP 是不固定的，所以 K8s 通过 Service 对象向应用程序的客户端提供一个静态/稳定的网络地址，另外因为应用程序往往是由多个Pod 副本构成，Service还可以为它负责的 Pod 提供负载均衡的功能。

每个 Service 都具有一个ClusterIP和一个可以解析为该IP的DNS名，并且由这个 ClusterIP 向 Pod 提供负载均衡。

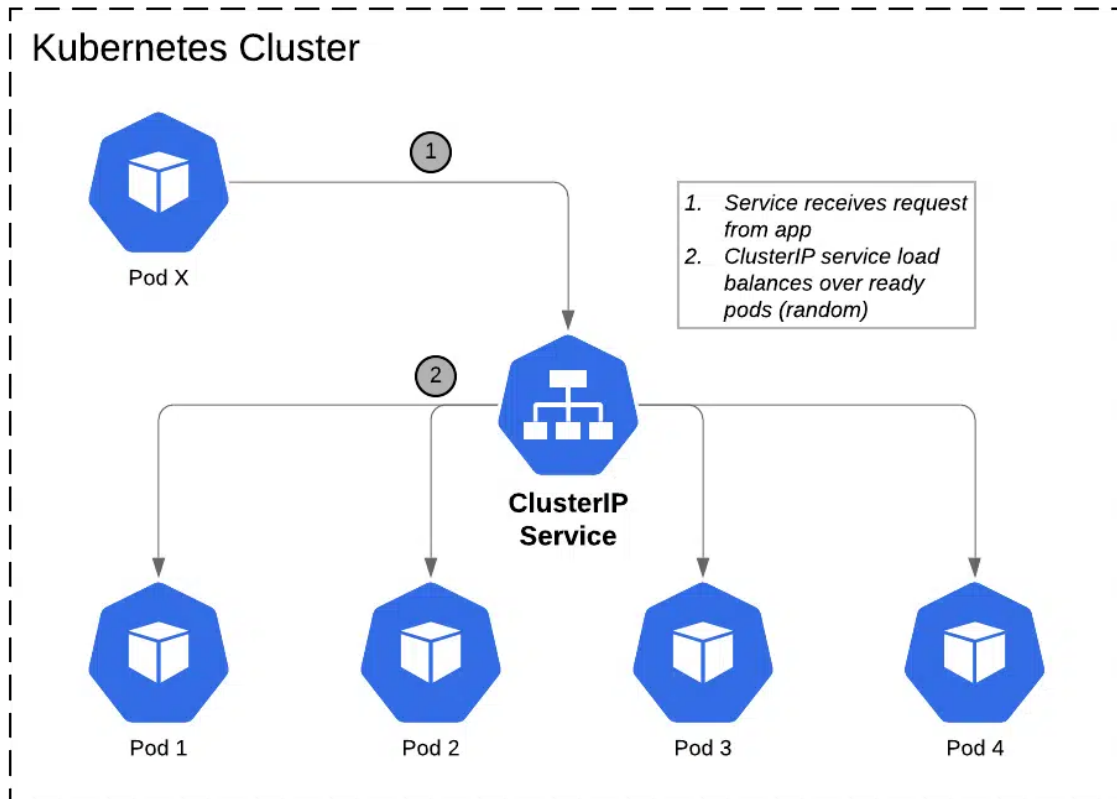
Service 控制器也是靠着 Pod 的标签在集群里筛选到自己要代理的 Pod，被选中的 Pod 叫做 Service 的端点 (EndPoint)

看一下 Service 对象定义的模板：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
spec:
  # type: NodePort, 这个不配置，svc默认只能在集群内访问
  selector:
    app: nginx-pod
  ports:
    - name: default
      protocol: TCP
      port: 80
      targetPort: 80
```

## ClusterIP Service

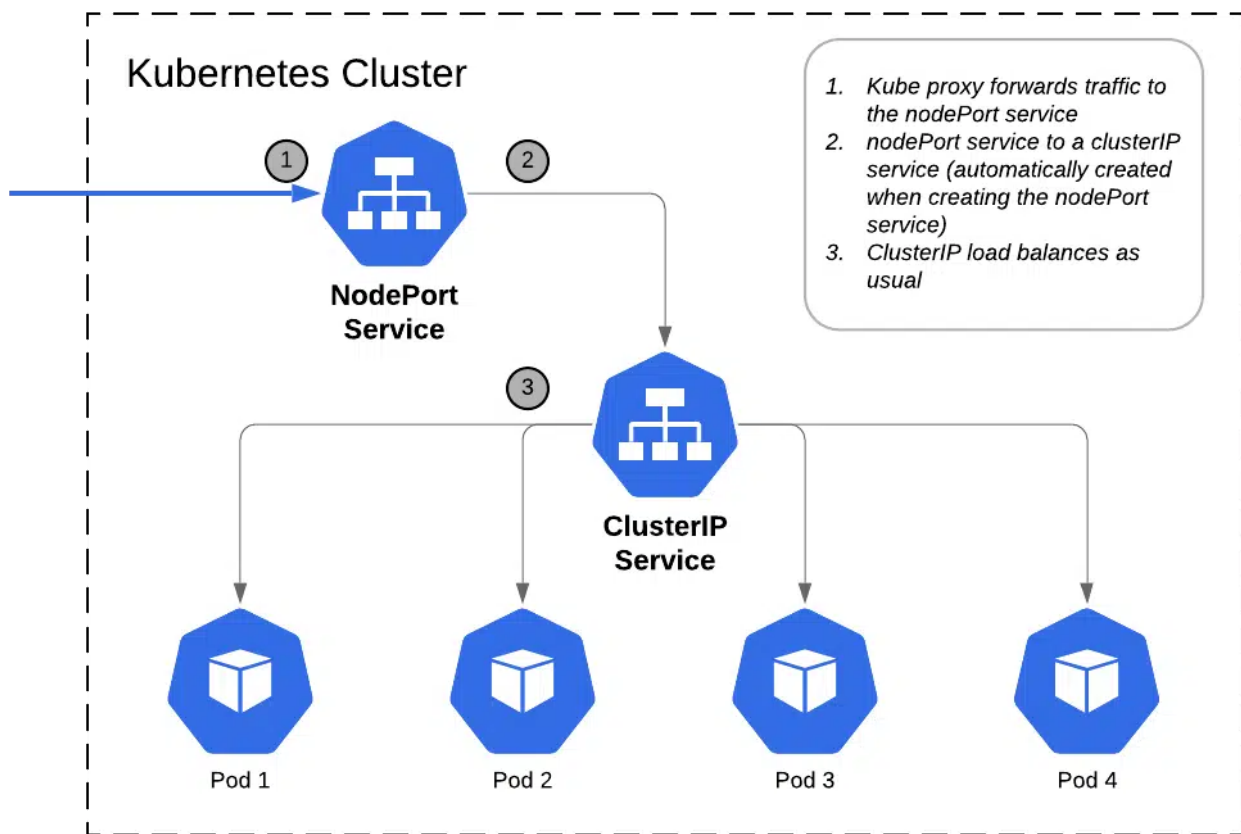
这是默认的Service类型，会将Service对象通过一个内部IP暴露给集群内部，这种类型的Service只能够在集群内部使用<ClusterIP>:<port>访问。



## NodePort Service

会在每个 Node 的一个指定的固定端口上暴露Service，与此同时还会自动创建一个ClusterIP类型的Service，NodePort类型的Service会将集群外部的请求路由给ClusterIP类型的Service。

使用<NodeIP>:<NodePort>访问NodePort类型的Service，NodePort的端口范围为30000–32767。

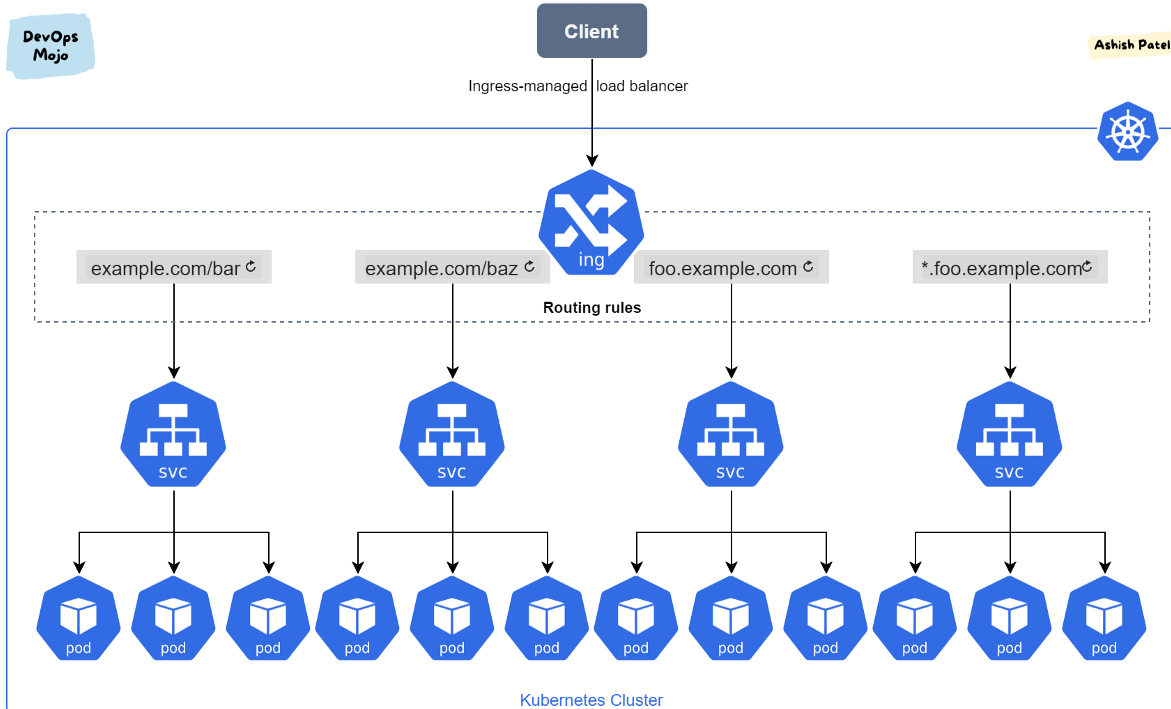


**NodePort** 类型的 **Service** 是向集群外暴露服务的最原始方式，也是最好让人理解的。优点是简单，好理解，通过IP+端口的方式就能访问，不过它的缺点也很明显。

- 每向外暴露一个服务都要占用所有Node的一个端口，如果多了难以管理。
- NodePort的端口区间固定，只能使用30000—32767间的端口。
- 如果Node的IP发生改变，负载均衡代理需要跟着改后端端点IP才行。

## Ingress

Ingress 在 K8s 集群里的角色是给 Service 充当反向代理。它可以位于多个 Service 的前端，给这些 Service 充当“智能路由”或者集群的入口点。



使用 Ingress 对象前需要先安装 Ingress-Controller, 像阿里云、亚马逊 AWS 他们的 K8s 企业服务都会提供自己的Controller, 对于自己搭建的集群, 通常使用nginx-ingress作为控制器, 它使用 NGINX 服务器作为反向代理, 访问 Ingress 的流量按规则路由给集群内部的Service。

下面看一下 Ingress 的配置

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  rules:
    - host: nginx.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: nginx-svc
              servicePort: 80
```



正常的生产环境，因为Ingress是公网的流量入口，所以压力比较大肯定需要多机部署。一般会在集群里单独出几台Node，只用来跑Ingress-Controller，可以使用daemonSet的让节点创建时就安装上Ingress-Controller，在这些Node上层再做一层负载均衡，把域名的DNS解析到负载均衡IP上。

## DaemonSet

这个控制器不常用，主要保证每个 Node上 都且只有一个 DaemonSet 指定的 Pod 在运行。当然可以定义多个不同的 DaemonSet 来运行各种基础软件的 Pod。

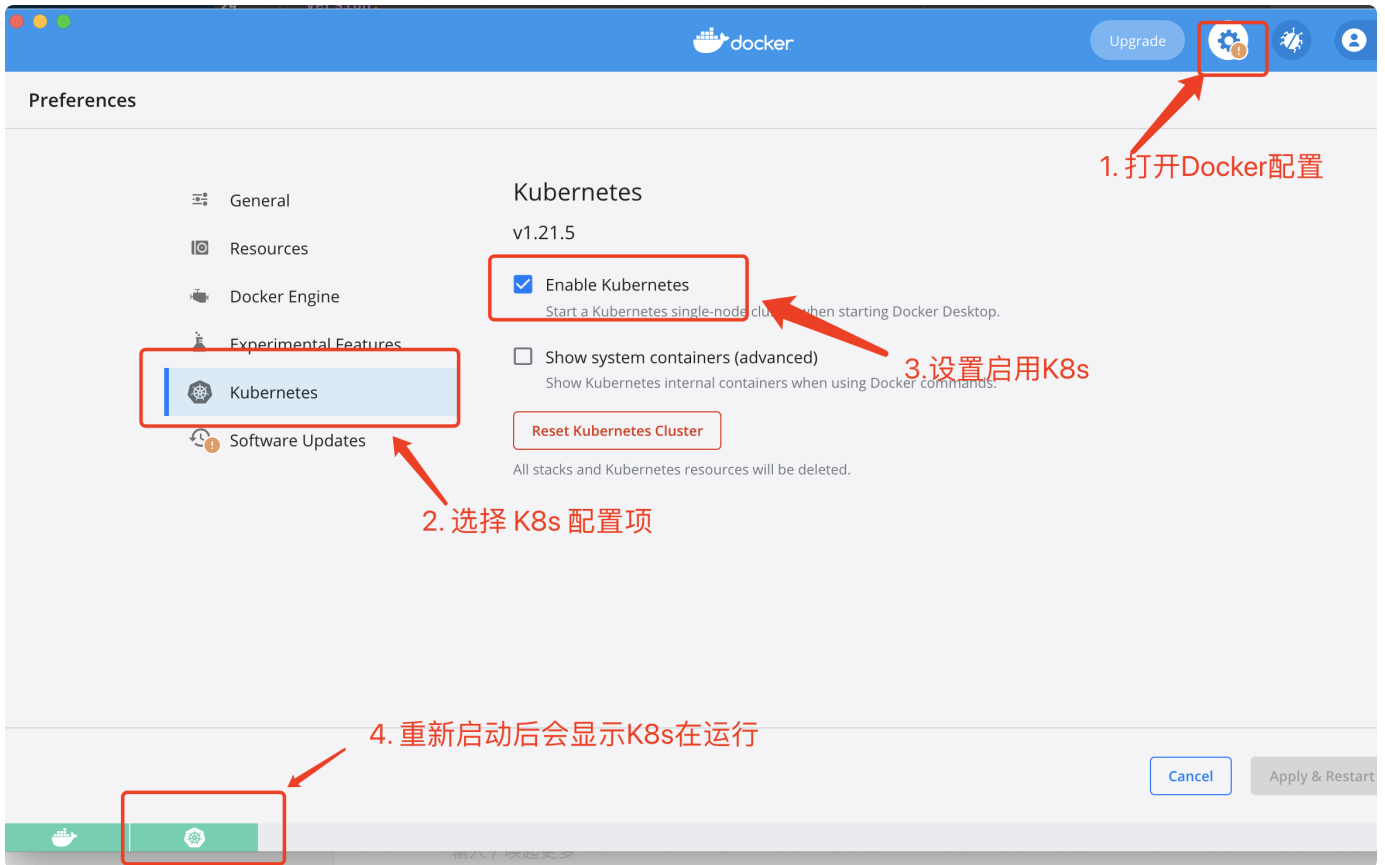
比如新建节点的网络配置、或者是每个节点上收集日志的组件往往是靠 DaemonSet 来保证的，他会在集群创建时优先于其他组件执行，为的是做好集群的基础设施建设。

## K8s 本机环境安装

可选的软件：

- minikube。
- Kind。
- Docker 桌面应用自带的 K8s 集群。

下面是用 Docker 桌面应用启用K8s的步骤：



## 从零开始部署一个Web APP到 K8s

下面演示一下怎么把一个Web应用部署到 K8s 集群上运行。因为 K8s 是基于容器技术的分布式架构方案，所以首先我们需要把要部署的应用程序打包到容器镜像里。

这里会把我们上面理论部分的知识点再串一遍，主要有这么几个步骤：

- 把 Web 程序打包成容器镜像
- 使用上一步打包的镜像，创建应用的Pod
- 用 Deployment 调度应用
- 使用 Service 暴露应用
- 通过 Ingress 代理应用

## 把 Web 程序打包成容器镜像

首先看下面这个简单的程序

```

package main

import (
    "fmt"
    "net/http"
    "os"
)

func indexHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello World")
    hostname, _ := os.Hostname()
    fmt.Fprintf(w, "Hello, you are visiting host: %s", hostname)
}

func main() {
    http.HandleFunc("/", indexHandler)
    fmt.Println("Server starting on port:8080...")
    http.ListenAndServe(":8080", nil)
}

```

这里是用 Go 程序起了一个特别简单的 HTTP Server，访问 "/" 路径后像页面上打印一行文字，告诉访问者他正在访问的网页的主机地址。

下面是打包镜像用的 Dockerfile

```

FROM golang:alpine as build
RUN apk --no-cache add tzdata
WORKDIR /app
ADD . /app
RUN CGO_ENABLED=0 GOOS=linux go build -o demoapp .

FROM scratch as final
COPY --from=build /app/demoapp .
COPY --from=build /usr/share/zoneinfo /usr/share/zoneinfo
ENV TZ=Asia/Shanghai
CMD ["/demoapp"]

```

- 打包应用镜像

```
1 docker build -t registry.cn-hangzhou.aliyuncs.com/docker-study-lab/simple-app-go:v0.1 .
```


- 上传到镜像仓库

```
1 docker push registry.cn-hangzhou.aliyuncs.com/docker-study-lab/simple-app-go:v0.1
```

## 配置应用的Pod 和 Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app-go
  template:
    metadata:
      labels:
        app: app-go
    spec:
      containers:
        - name: go-app-container
          image: registry.cn-hangzhou.aliyuncs.com/docker-study-lab/simple-app-go:v0.2
          resources:
            limits:
              memory: "50Mi"
              cpu: "50m"
          ports:
            - containerPort: 8080
```

## 用 Service 暴露服务



```
apiVersion: v1
kind: Service
metadata:
  name: app-svc
spec:
  type: NodePort
  selector:
    app: app-go
  ports:
    - name: http
      protocol: TCP
      nodePort: 30088
      port: 80
      targetPort: 8080
```

## 用 Ingress 代理服务

使用 Ingress 前需要先安装 Ingress Controller，这里我们使用开源的 Ingress-Nginx

- 安装 Ingress nginx Controller: <https://kubernetes.github.io/ingress-nginx/deploy/>

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ing
  annotations:
    kubernetes.io/ingress.class: "nginx"
    ingress.kubernetes.io/enable-access-log: "true"
spec:
  rules:
    - host: app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app-svc
                port:
                  number: 80
```

## 最后

这里总结了一下 K8s 常用的入门知识以及相关的实践操作，只能算是一个非常初级的入门，还有其他很多非常高级的特性能让我们控制 K8s 对应用的各种调度动作。

现在用 K8s 作为基建的公司越来越多，掌握 K8s 对我们做应用架构设计绝对是有好处的，希望这篇入门文章能给你的K8 学习之旅创造一个好的开端，后面还有很多知识需要我们继续探索。

这里推荐几个比较好的学习资源：

- Deployment 滚动更新配置详解：<https://www.blumatador.com/blog/kubernetes-deployments-rolling-update-configuration>
- K8s 基础教程：<https://www.bmc.com/blogs/what-is-kubernetes/>

- K8是中文指南/云原生应用架构手册: <https://jimmysong.io/kubernetes-handbook/>