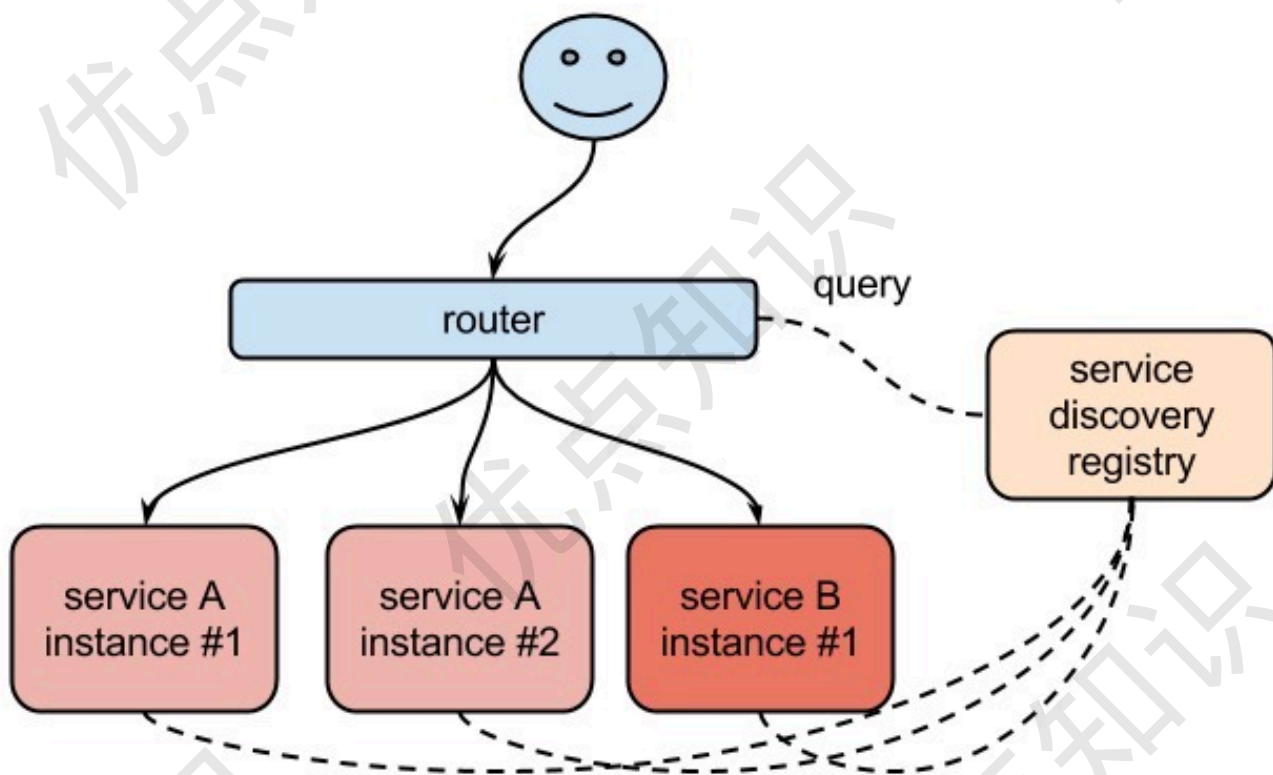


# Service

我们前面的课程中学习了一些常用控制器的基本用法，我们也了解到 Pod 的生命是有限的，死亡过后不会复活了。然后我们知道可以用 ReplicaSet 和 Deployment 来动态的创建和销毁 Pod，每个 Pod 都有自己的 IP 地址，但是如果 Pod 重建了的话那么他的 IP 很有可能也就变化了。这就会带来一个问题：比如我们有一些后端的 Pod 集合为集群中的其他应用提供 API 服务，如果我们在前端应用中把所有的这些后端的 Pod 的地址都写死，然后以某种方式去访问其中一个 Pod 的服务，这样看上去是可以工作的，对吧？但是如果这个 Pod 挂掉了，然后重新启动起来了，是不是 IP 地址非常有可能就变了，这个时候前端就极大可能访问不到后端的服务了。

遇到这样的问题该怎么解决呢？在没有使用 Kubernetes 之前，我相信可能很多同学都遇到过这样的问题，不一定是 IP 变化的问题，比如我们在部署一个 WEB 服务的时候，前端一般部署一个 Nginx 作为服务的入口，然后 Nginx 后面肯定就是挂载的这个服务的大量后端服务，很早以前我们可能是去手动更改 Nginx 配置中的 upstream 选项，来动态改变提供服务的数量，到后面出现了一些服务发现的工具，比如 Consul、ZooKeeper 还有我们熟悉的 etcd 等工具，有了这些工具过后我们就可以只需要把我们的服务注册到这些服务发现中心去就可以，然后让这些工具动态的去更新 Nginx 的配置就可以了，我们完全不用去手工的操作了，是不是非常方便。



同样的，要解决我们上面遇到的问题是不是实现一个服务发现的工具也可以解决？没错的，当我们 Pod 被销毁或者新建过后，我们可以把这个 Pod 的地址注册到这个服务发现中心去就可以，但是这样的话我们的前端应用就不能直接去连接后台的 Pod 集合了，应该连接到一个能够做服务发现的中间件上面，对吧？

为解决这个问题 Kubernetes 就为我们提供了这样的一个对象 - **Service**，Service 是一种抽象的对象，它定义了一组 Pod 的逻辑集合和一个用于访问它们的策略，其实这个概念和微服务非常类似。一个 Service 下面包含的 Pod 集合是由 Label Selector 来决定的。

比如我们上面的例子，假如我们后端运行了 3 个副本，这些副本都是可以替代的，因为前端并不关心它们使用的是哪一个后端服务。尽管由于各种原因后端的 Pod 集合会发送变化，但是前端却不需要知道这些变化，也不需要自己用一个列表来记录这些后端的服务，Service 的这种抽象就可以帮我们达到这种解耦的目的。

## 三种 IP

在继续往下学习 Service 之前，我们需要先弄明白 Kubernetes 系统中的三种 IP，因为经常有同学混乱。

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

首先，Node IP 是 Kubernetes 集群中节点的物理网卡 IP 地址（一般为内网），所有属于这个网络的服务器之间都可以直接通信，所以 Kubernetes 集群外要想访问 Kubernetes 集群内部的某个节点或者服务，肯定得通过 Node IP 进行通信（这个时候一般是通过外网 IP 了）

然后 Pod IP 是每个 Pod 的 IP 地址，它是网络插件进行分配的，前面我们已经讲解过

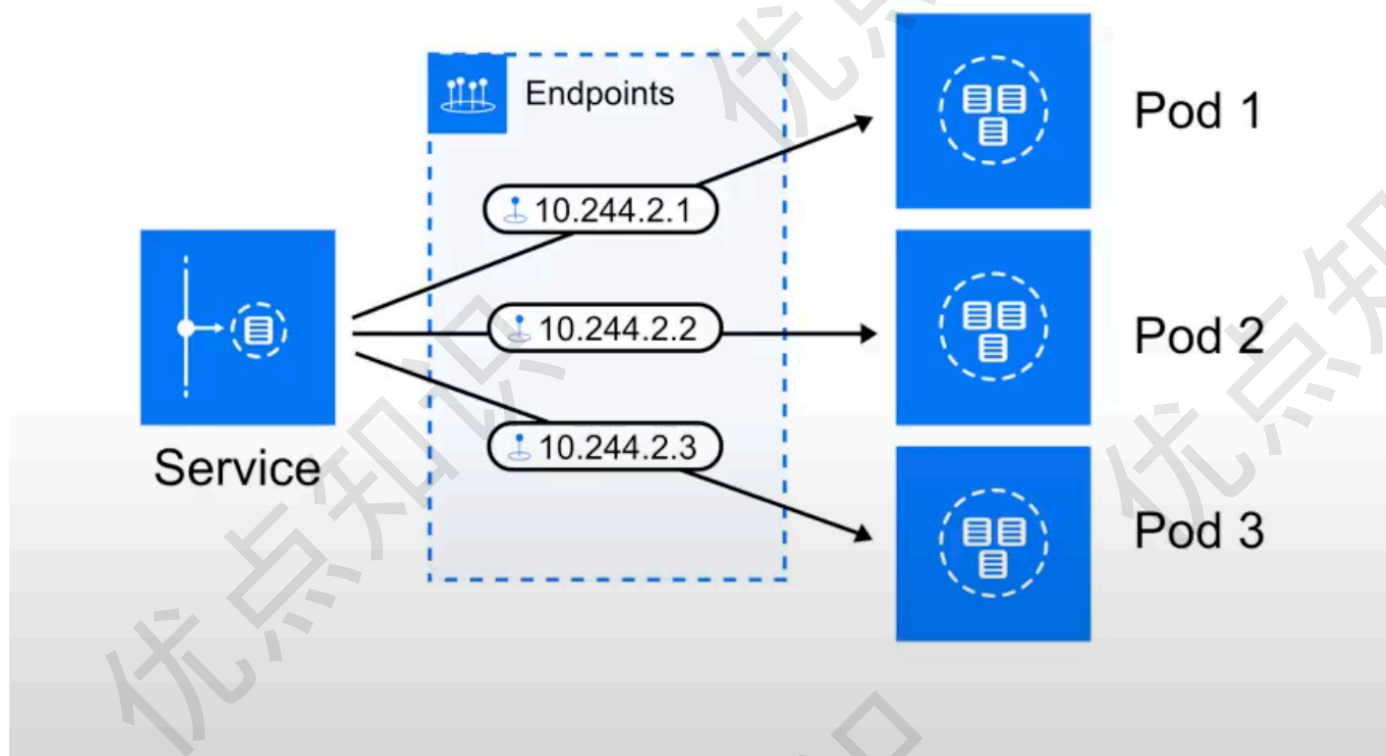
最后 Cluster IP 是一个虚拟的 IP，仅仅作用于 Kubernetes Service 这个对象，由 Kubernetes 自己来进行管理和分配地址。

## 定义 Service

定义 Service 的方式和我们前面定义的各种资源对象的方式类型，例如，假定我们有一组 Pod 服务，它们对外暴露了 8080 端口，同时都被打上了 `app=myapp` 这样的标签，那么我们就可以像下面这样来定义一个 Service 对象：

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      name: myapp-http
```

然后通过的使用 `kubectl create -f myservice.yaml` 就可以创建一个名为 myservice 的 Service 对象，它会将请求代理到使用 TCP 端口为 8080，具有标签 `app=myapp` 的 Pod 上，这个 Service 会被系统分配一个我们上面说的 Cluster IP，该 Service 还会持续的监听 selector 下面的 Pod，会把这些 Pod 信息更新到一个名为 myservice 的 **Endpoints** 对象上去，这个对象就类似于我们上面说的 Pod 集合了。

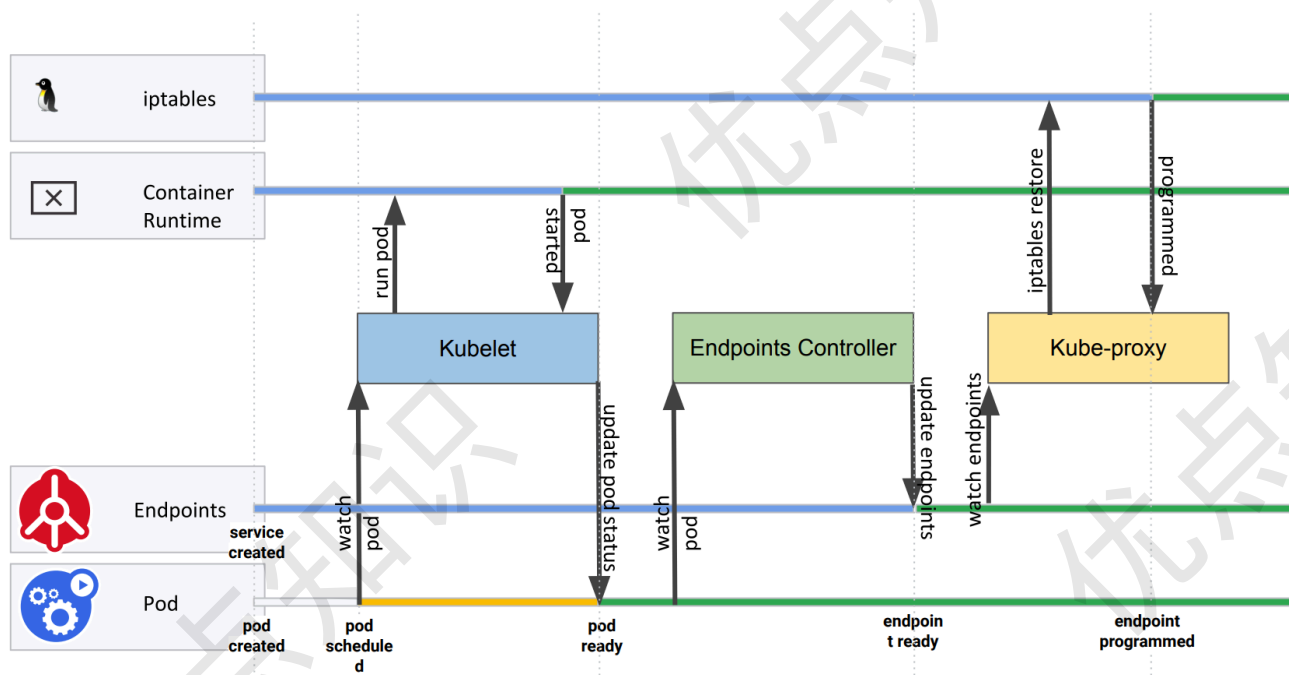


需要注意的是，Service 能够将一个接收端口映射到任意的 targetPort。默认情况下，targetPort 将被设置为与 port 字段相同的值。可能更有趣的是，targetPort 可以是一个字符串，引用了 backend Pod 的一个端口的名称。因实际指派给该端口名称的端口号，在每个 backend Pod 中可能并不相同，所以对于部署和设计 Service，这种方式会提供更大的灵活性。

另外 Service 能够支持 TCP 和 UDP 协议，默认是 TCP 协议。

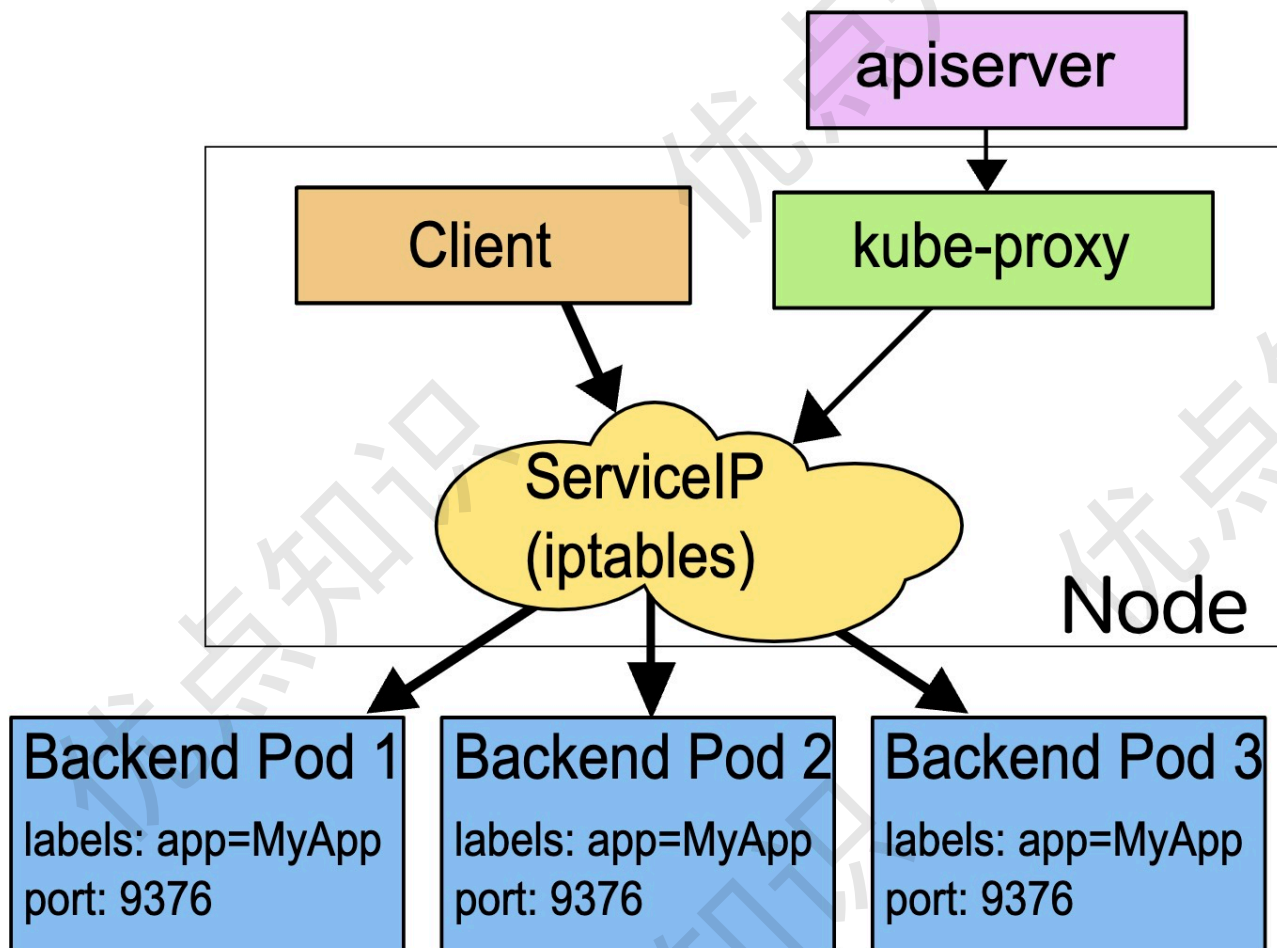
## kube-proxy

前面我们讲到过，在 Kubernetes 集群中，每个 Node 会运行一个 kube-proxy 进程，负责为 Service 实现一种 VIP（虚拟 IP，就是我们上面说的 clusterIP）的代理形式，现在的 Kubernetes 中默认是使用的 iptables 这种模式来代理。



## iptables

这种模式，kube-proxy 会 watch apiserver 对 Service 对象和 Endpoints 对象的添加和移除。对每个 Service，它会添加上 iptables 规则，从而捕获到达该 Service 的 clusterIP（虚拟 IP）和端口的请求，进而将请求重定向到 Service 的一组 backend 中的某一个 Pod 上面。我们还可以使用 **Pod readiness 探针** 验证后端 Pod 可以正常工作，以便 iptables 模式下的 kube-proxy 仅看到测试正常的后端，这样做意味着可以避免将流量通过 kube-proxy 发送到已知失败的 Pod 中，所以对于线上的应用来说一定要做 readiness 探针。



iptables 模式的 kube-proxy 默认的策略是，随机选择一个后端 Pod。

比如当创建 backend Service 时，Kubernetes 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。假设 Service 的端口是 1234，该 Service 会被集群中所有的 kube-proxy 实例观察到。当 kube-proxy 看到一个新的 Service，它会安装一系列的 iptables 规则，从 VIP 重定向到 **per-Service** 规则。该 **per-Service** 规则连接到 **per-Endpoint** 规则，该 **per-Endpoint** 规则会重定向（目标 NAT）到后端的 Pod。

## ipvs

除了 iptables 模式之外，kubernetes 也支持 ipvs 模式，在 ipvs 模式下，kube-proxy watch Kubernetes 服务和端点，调用 **netlink** 接口相应地创建 IPVS 规则，并定期将 IPVS 规则与 Kubernetes 服务和端点同步。该控制循环可确保 IPVS 状态与所需状态匹配。访问服务时，IPVS 将流量定向到后端 Pod 之一。

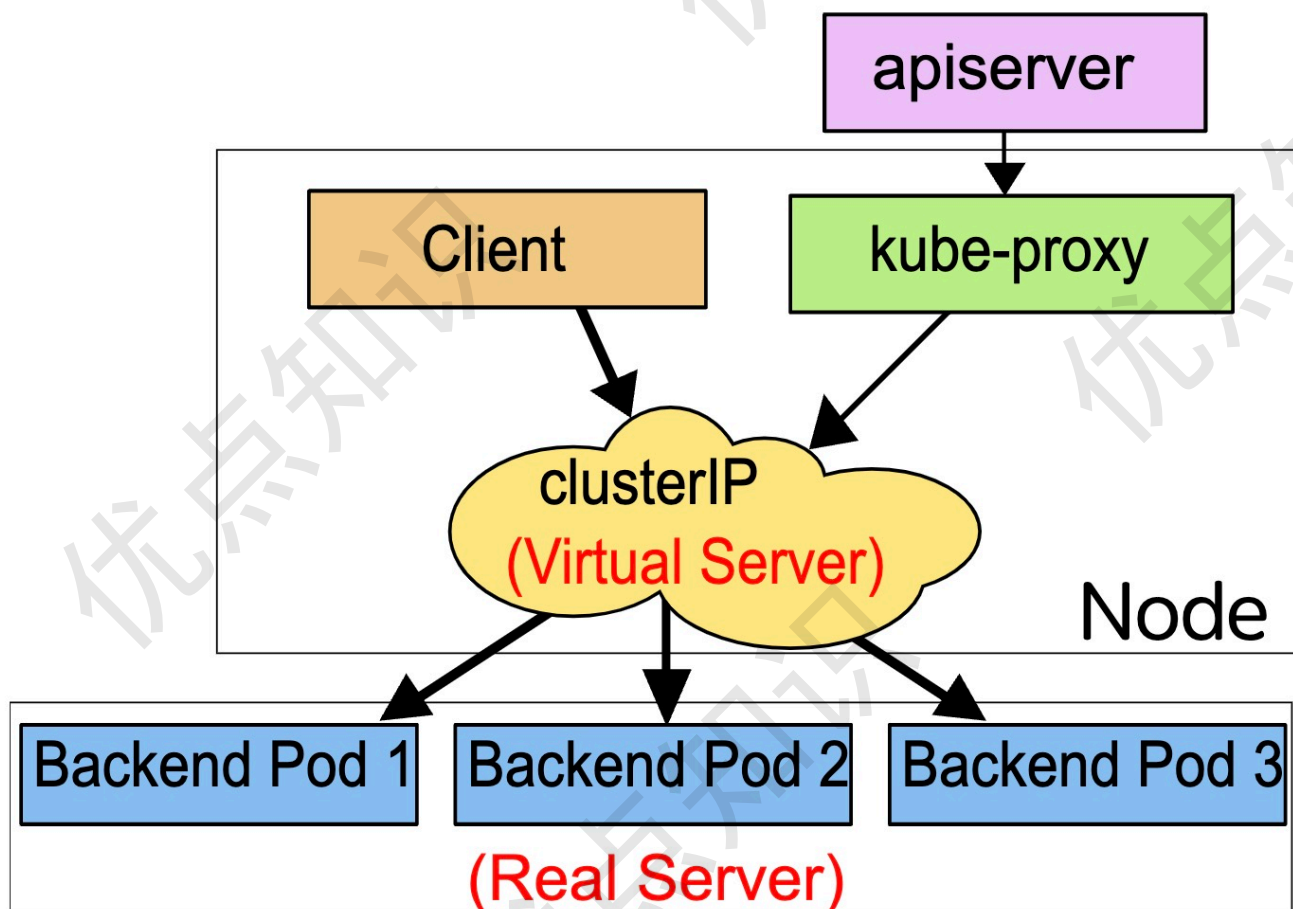
IPVS 代理模式基于类似于 iptables 模式的 netfilter 钩子函数，但是使用**哈希表**作为基础数据结构，并且在内核空间中工作。所以与 iptables 模式下的 kube-proxy 相比，IPVS 模式下的 kube-proxy 重定向通信的延迟要短，并且在同步代理规则时具有更好的性能。与其他代理模式相比，IPVS 模式还支持更高的网络流量吞吐量。所以对于较大规模的集群会使用 ipvs 模式的 kube-proxy，只需要满足节点上运行 ipvs 的条件，然后我们就可以直接将 kube-proxy 的模式修改为 ipvs，如果不满足运行条件会自动降级为 iptables 模式，现在都推荐使用 ipvs 模式，可以大幅度提高 Service 性能。

IPVS 提供了更多选项来平衡后端 Pod 的流量，默认是 **rr**，有如下一些策略：

- rr: round-robin
- lc: least connection (smallest number of open connections)
- dh: destination hashing
- sh: source hashing

- sed: shortest expected delay
- nq: never queue

不过现在只能整体修改策略，可以通过 kube-proxy 中配置 `-ipvs-scheduler` 参数来实现，暂时不支持特定的 Service 进行配置。



我们也可以实现基于客户端 IP 的会话亲和性，可以将 `service.spec.sessionAffinity` 的值设置为 "ClientIP"（默认值为 "None"）即可，此外还可以通过适当设置 `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` 来设置最大会话停留时间（默认值为 10800 秒，即 3 小时）：

```
apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...
```

Service 只支持两种形式的会话亲和性服务：None 和 ClientIP，不支持基于 cookie 的会话亲和性，这是因为 Service 不是在 HTTP 层面上工作的，处理的是 TCP 和 UDP 包，并不关心其中的载荷内容，因为 cookie 是 HTTP 协议的一部分，Service 并不知道它们，所有会话亲和性不能基于 Cookie。



## Service 使用

我们在定义 Service 的时候可以指定一个自己需要的类型的 Service，如果不指定的话默认是 **ClusterIP** 类型。

我们可以使用的服务类型如下：

- **ClusterIP**: 通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的服务类型。
- **NodePort**: 通过**每个 Node 节点**上的 IP 和静态端口（NodePort）暴露服务。NodePort 服务会路由到 ClusterIP 服务，这个 ClusterIP 服务会自动创建。通过请求 **NodeIp:NodePort**，可以从集群的外部访问一个 NodePort 服务。
- **LoadBalancer**: 使用云提供商的负载均衡器，可以向外部暴露服务。外部的负载均衡器可以路由到 NodePort 服务和 ClusterIP 服务，这个需要结合具体的云厂商进行操作。
- **ExternalName**: 通过返回 **CNAME** 和它的值，可以将服务映射到 **externalName** 字段的内容（例如，foo.bar.example.com）。

### NodePort 类型

如果设置 type 的值为 "NodePort", Kubernetes master 将从给定的配置范围内（默认: 30000-32767）分配端口，每个 Node 将从该端口（每个 Node 上的同一端口）代理到 Service。该端口将通过 Service 的 **spec.ports[\*].nodePort** 字段被指定，如果不指定的话会自动生成一个端口。

需要注意的是，Service 将能够通过 **spec.ports[].nodePort** 和 **spec.clusterIp:spec.ports[].port** 而对外可见。

接下来我们来给大家创建一个 NodePort 的服务来访问我们前面的 Nginx 服务：

```
# service-nodeport-demo.yaml
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  selector:
    app: myapp
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      name: myapp-http
```

创建该 Service：

```
❏ → kubectl apply -f service-nodeport-demo.yaml
```

然后我们可以查看 Service 对象信息：

```
❏ → kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	46d
myservice	NodePort	10.96.209.180	<none>	80:31696/TCP	10s

我们可以看到 `myservice` 的 `TYPE` 类型已经变成了 `NodePort`，后面的 `PORT(S)` 部分也多了一个 `31696` 的映射端口。

## ExternalName

`ExternalName` 是 `Service` 的特例，它没有 `selector`，也没有定义任何的端口和 `Endpoint`。对于运行在集群外部的服务，它通过返回该外部服务的别名这种方式来提供服务。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

当访问地址 `my-service.prod.svc.cluster.local`（后面服务发现的时候我们会再深入讲解）时，集群的 `DNS` 服务将返回一个值为 `my.database.example.com` 的 `CNAME` 记录。访问这个服务的工作方式与其它的相同，唯一不同的是重定向发生在 `DNS` 层，而且不会进行代理或转发。如果后续决定要将数据库迁移到 `Kubernetes` 集群中，可以启动对应的 `Pod`，增加合适的 `Selector` 或 `Endpoint`，修改 `Service` 的 `type`，完全不需要修改调用的代码，这样就完全解耦了。

除了可以直接通过 `externalName` 指定外部服务的域名之外，我们还可以通过自定义 `Endpoints` 来创建 `Service`，前提是 `clusterIP=None`，名称要和 `Service` 保持一致，如下所示：

```
apiVersion: v1
kind: Service
metadata:
  name: etcd-k8s
  namespace: kube-system
  labels:
    k8s-app: etcd
spec:
  type: ClusterIP
  clusterIP: None
  ports:
    - name: port
      port: 2379
---
apiVersion: v1
kind: Endpoints
metadata:
  name: etcd-k8s # 名称必须和 Service 一致
  namespace: kube-system
  labels:
    k8s-app: etcd
subsets:
  - addresses:
    - ip: 10.151.30.57 # Service 将连接重定向到 endpoint
```



```
ports:
  - name: port
    port: 2379 # endpoint 的目标端口
```

上面这个服务就是将外部的 etcd 服务引入到 Kubernetes 集群中来。

## externalIPs

Service 的属性中还有一个 `externalIPs` 的属性，从 Kubernetes 官网文档可以看到该属性的相关描述：

如果外部的 IP 路由到集群中一个或多个 Node 上，Kubernetes Service 会被暴露给这些 `externalIPs`。通过外部 IP（作为目的 IP 地址）进入到集群，传到 Service 的端口上的流量，将会被路由到 Service 的 Endpoint 上，`externalIPs` 不会被 Kubernetes 管理，它属于集群管理员的职责范畴。

这里最重要的一点就是确保使用哪个 IP 来访问 Kubernetes 集群，使用外部 IP Service 类型，我们可以将 Service 绑定到连接集群的 IP。

参考文档：<https://www.fadhil-blog.dev/blog/kubernetes-external-ip/>。

## Endpoints 与 Endpointslices

我们已经知道在 Service 创建时，Kubernetes 会根据 Service 关联一个 Endpoints 资源，若 Service 没有定义 selector 字段，将不会自动创建 Endpoints。Endpoints 是 Kubernetes 中的一个资源对象，存储在 etcd 中，用来记录一个 Service 对应一组 Pod 的访问地址，一个 Service 只有一个 Endpoints 资源，Endpoints 资源会去观测 Pod 集合，只要服务中的某个 Pod 发生变更，Endpoints 就会进行同步更新。

比如现在我们部署如下所示 3 个副本的 httpbin 测试应用：

```
# httpbin-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
spec:
  replicas: 3
  selector:
    matchLabels:
      app: httpbin
  template:
    metadata:
      labels:
        app: httpbin
    spec:
      containers:
        - name: httpbin
          image: kennethreitz/httpbin:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
```

```

      name: http
---
apiVersion: v1
kind: Service
metadata:
  name: httpbin
spec:
  selector:
    app: httpbin
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 80
  type: ClusterIP

```

在该应用中我们定义了 3 个副本，然后创建了一个 Service 对象来关联这些 Pod，直接应用该资源清单即可：

```

❖ → kubectl apply -f httpbin-deploy.yaml
❖ → kubectl get pods -l app=httpbin -o wide

```

NAME	NOMINATED NODE	READY	STATUS	RESTARTS	AGE	IP	NODE
httpbin-75d9685444-9v9x2	<none>	1/1	Running	0	7m42s	10.244.1.8	demo-worker
httpbin-75d9685444-fsftf	<none>	1/1	Running	0	7m42s	10.244.1.7	demo-worker
httpbin-75d9685444-vvx2c	<none>	1/1	Running	0	7m42s	10.244.2.7	demo-worker2

```

❖ → kubectl get svc httpbin

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
httpbin	ClusterIP	10.96.158.93	<none>	80/TCP	2m46s

```

❖ → kubectl get endpoints httpbin

```

NAME	ENDPOINTS	AGE
httpbin	10.244.1.7:80,10.244.1.8:80,10.244.2.7:80	6m46s

```

❖ → kubectl get endpoints httpbin -oyaml
apiVersion: v1
kind: Endpoints
metadata:
  annotations:
    endpoints.kubernetes.io/last-change-trigger-time: "2023-01-17T07:18:49Z"
  creationTimestamp: "2023-01-17T07:12:54Z"
  name: httpbin
  namespace: default
  resourceVersion: "965173"
  uid: cafa3f6c-86fa-47df-a849-cc0954d2897f
subsets:
- addresses:
  - ip: 10.244.1.7
    nodeName: demo-worker
    targetRef:

```

```
kind: Pod
name: httpbin-75d9685444-fsftf
namespace: default
uid: d62e09c4-62fb-48d1-be93-f4184a3714fa
- ip: 10.244.1.8
  nodeName: demo-worker
  targetRef:
    kind: Pod
    name: httpbin-75d9685444-9v9x2
    namespace: default
    uid: 2faf6f38-82fb-425c-a3ec-abd080e6cdd6
- ip: 10.244.2.7
  nodeName: demo-worker2
  targetRef:
    kind: Pod
    name: httpbin-75d9685444-vvx2c
    namespace: default
    uid: 0f848cef-e405-4d35-847c-8752df3d3573
ports:
- name: http
  port: 80
  protocol: TCP
```

由于我们这里创建了一个 Service 对象，所以也会自动创建一个对应的 Endpoints 对象，只有当 Pod 正常运行后才会被包含到 Endpoints 对象中去，所以往往如果是线上应用我们都是**强烈推荐**为 Pod 配置上 **readiness probe** 的，当应用还未就绪的时候就不会进入 Endpoints，也就不会对外提供服务了，当应用下线后就从该对象中摘掉。

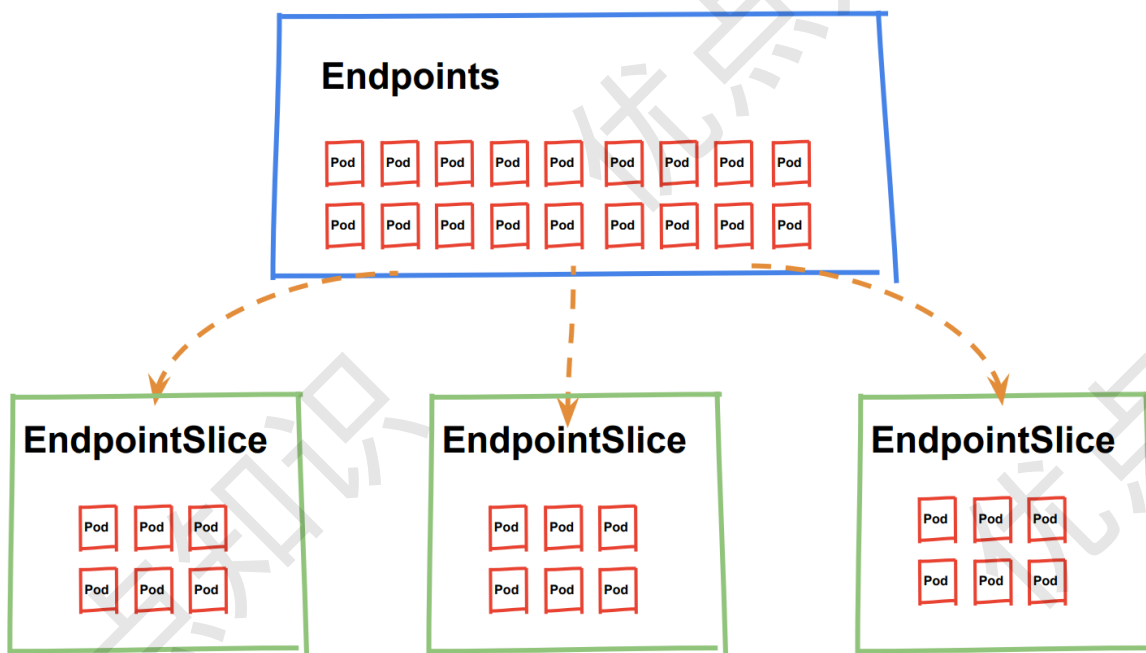
从上述示例可以看到，Endpoints 中的所有网络端点，分别对应了每个 Pod 的 IP 地址，也就是上面对象中的 **subsets** 里面的数据。

### Endpoints 的不足之处

但实际上 Endpoints 也有它的一些不足之处，比如：

- Kubernetes 限制单个 Endpoints 对象中可以容纳的端点数量。当一个服务有超过 1000 个后端端点时，Kubernetes 会截断 Endpoints 对象中的数据，这种情况下，Kubernetes 选择最多 1000 个可能的后端端点来存储到 Endpoints 对象中，并在 Endpoints 中配置上 **endpoints.kubernetes.io/over-capacity: truncated** 注解。如果后端 Pod 的数量低于 1000，控制平面会移除该注解。
- 一个 Service 只有一个 Endpoints 资源，这意味着它需要为支持相应服务的每个 Pod 存储 IP 等网络信息。这导致 Endpoints 资源变的十分巨大，其中一个端点发生了变更，将会导致整个 Endpoints 资源更新。当业务需要进行频繁端点更新时，一个巨大的 API 资源被相互传递，而这会影响到 Kubernetes 组件的性能，并且会产生大量的网络流量和额外的处理。

所以当你的应用规模达到了上千规模的话，我们就不能使用原有的这种方案了，一个新的对象 **Endpointslices** 就出现了，Endpointslices 为 Endpoints 提供了一种可伸缩和可拓展的替代方案，缓解处理大量网络端点带来的性能问题，还能对一些诸如拓扑路由的额外功能提供一个可扩展的平台，该特性在 Kubernetes v1.21+ 的版本中已提供支持。

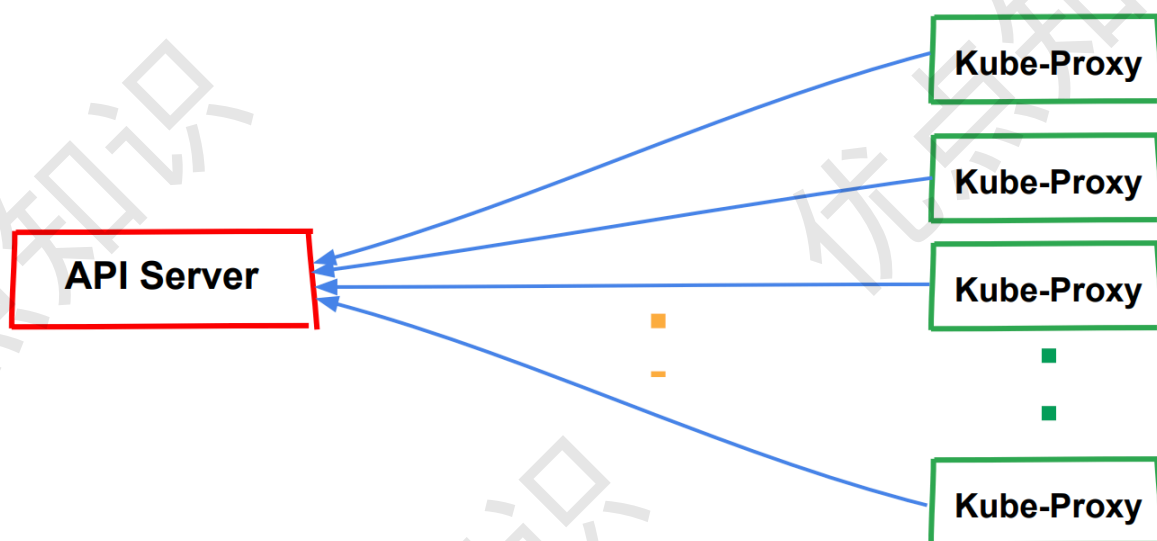


默认情况下，控制面创建和管理的 **EndpointSlice** 将包含不超过 100 个端点，但是我们可以使用 **kube-controller-manager** 的 **--max-endpoints-per-slice** 标志设置此值，其最大值为 1000。

### 为什么需要 Endpointslices?

为了说明这些问题的严重程度，这里举一个简单的例子。假设我们有一个 2000 个 Pod 副本的服务，它最终生成的 **Endpoints** 资源对象会很大（比如 1.5MB（**etcd** 具有最大请求大小限制 1.5MB），还会截断成 1000），在生产环境中，如果该服务发生滚动更新或节点迁移，那么 **Endpoints** 资源将会频繁变更，当该列表中的某个网络端点发生了变化，那么就要将完整的 **Endpoint** 资源分发给集群中的每个节点。如果我们在一个具有 3000 个节点的大型集群中，这会是个很大的问题，每次更新将跨集群发送 4.5GB 的数据（ $1.5\text{MB} \times 3000$ ，即 **Endpoint 大小 \* 节点个数**），并且每次端点更新都要发送这么多数据。想象一下，如果进行一次滚动更新，共有 2000 个 Pod 全部被替换，那么传输的数据量将会是 TB 级数据。这不进对集群内的网络带宽浪费巨大，而对 Master 的冲击非常大，会影响 Kubernetes 整体的性能。

## GET /api/v1/endpoints?watch=true&...



如果使用了 Endpointslices, 假设一个服务后端有 2000 个 Pod, 我们可以让每个 Endpointslices 存储 100 个端点, 最终将获得 20 个 Endpointslices。添加或删除 Pod 时, 只需要更新其中 1 个 Endpointslice 资源即可, 这样操作后, 可扩展性和网络可伸缩有了很大的提升。

比起在流量高峰时, 服务为了承载流量, 扩容出大量的 Pod, Endpoints 资源会被频繁更新, 两个使用场景的差异就变得非常明显。更重要的是, 既然服务的所有 Pod IP 都不需要存储在单个资源中, 那么我们也就不必担心 etcd 中存储的对象的大小限制。

所以我们可以了解到 Endpoints 和 Endpointslice 两种资源的不同之处了。

- Endpoints 适用场景:
  - 有弹性伸缩需求, Pod 数量较少, 传递资源不会造成大量网络流量和额外处理。
  - 无弹性伸缩需求, Pod 数量不会太多。哪怕 Pod 数量是固定, 但是总是要滚动更新或者出现故障的。
- Endpointslice 适用场景:
  - 有弹性需求, 且 Pod 数量较多 (几百上千)。
  - Pod 数量很多 (几百上千), 因为 Endpoints 网络端点最大数量限制为 1000, 所以超过 1000 的 Pod 必须得用 Endpointslice。

## EndpointSlice 使用

EndpointSlice 控制器会 Watch Service 和 Pod 来自动创建和更新 EndpointSlices 对象, 然后 kube-proxy 同样会 Watch Service 和 EndpointSlices 对象来更新 iptables 或 ipvs proxy 规则。

由于默认情况下, 控制面创建和管理的 EndpointSlice 将包含不超过 100 个端点, 为了测试方便, 我们这里将其修改为 5, 将 `kube-controller-manager` 的 `--max-endpoints-per-slice` 标志设置为 5, 最大值为 1000。

比如我们现在创建如下的资源对象:

```
# httpbin-deploy-slice.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
spec:
  replicas: 17
  selector:
    matchLabels:
      app: httpbin
  template:
    metadata:
      labels:
        app: httpbin
    spec:
      containers:
        - name: httpbin
          image: kennethreitz/httpbin:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              name: http
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: httpbin
spec:
  selector:
    app: httpbin
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 80
  type: ClusterIP
```

直接应用即可：

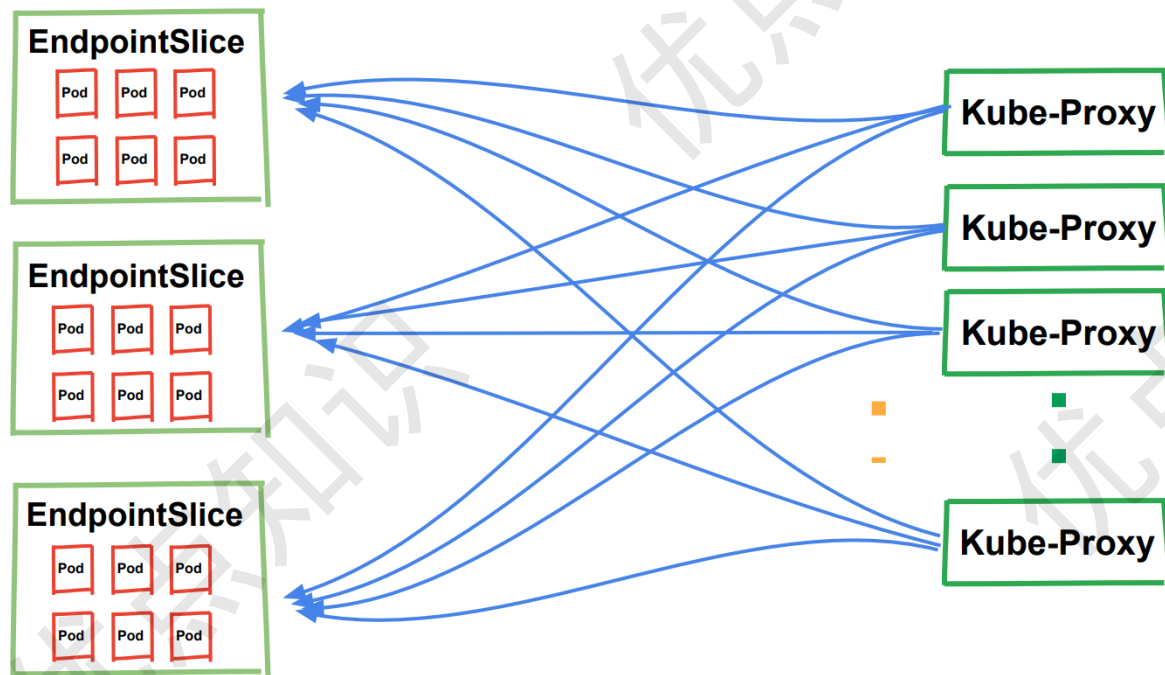
```
❖ → kubectl apply -f httpbin-deploy-slice.yaml
```

当 Pod 和 Service 创建后便会自动创建 EndpointSlices 对象了：

```
❖ → kubectl get svc httpbin
NAME      TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
httpbin   ClusterIP   10.96.8.67   <none>        80/TCP     61m
❖ → kubectl get endpointslice
NAME                ADDRESSTYPE  PORTS    ENDPOINTS
AGE
httpbin-8hts5       IPv4         80       10.244.2.185,10.244.1.183,10.244.2.184 + 2 more ...
61m
httpbin-c26qh       IPv4         80       10.244.1.180,10.244.2.182,10.244.1.177 + 2 more ...
61m
httpbin-gvbh8       IPv4         80       10.244.1.178,10.244.1.181,10.244.2.186 + 2 more ...
61m
httpbin-whvr4       IPv4         80       10.244.1.184,10.244.1.185
61m
```

由于我们这里配置的每个 EndpointSlice 最多有 5 个 Endpoint，一共 17 个副本，所以这里自动创建了 4 个 EndpointSlices 对象，当然现在 kube-proxy 组件会 Watch 所有的这个 4 个对象。





但是当我们更新一个 Pod 的时候只有包含该 Pod 的 EndpointSlice 对象变更，该对象中包含的 Endpoints 列表非常少，这就大大提升了性能。

我们可以查看任意一个 EndpointSlice 对象，如下所示：

```
❖ → kubectl get endpointslice httpbin-whvr4 -oyaml
```

```
addressType: IPv4
apiVersion: discovery.k8s.io/v1
endpoints:
- addresses:
  - 10.244.1.184
  conditions:
    ready: true
    serving: true
    terminating: false
  nodeName: demo-worker
  targetRef:
    kind: Pod
    name: httpbin-75d9685444-wl4w6
    namespace: default
    uid: 337f10f2-31b2-43d1-a5b5-4a6643580aea
- addresses:
  - 10.244.1.185
  conditions:
    ready: true
    serving: true
    terminating: false
  nodeName: demo-worker
  targetRef:
    kind: Pod
```

```

name: httpbin-75d9685444-jbbzh
namespace: default
uid: b08eea07-2b99-41c3-913e-1bc03424339c
kind: EndpointSlice
metadata:
  annotations:
    endpoints.kubernetes.io/last-change-trigger-time: "2023-01-17T08:33:21Z"
  creationTimestamp: "2023-01-17T08:33:23Z"
  generateName: httpbin-
  generation: 2
  labels:
    endpointslice.kubernetes.io/managed-by: endpointslice-controller.k8s.io
    kubernetes.io/service-name: httpbin
name: httpbin-whvr4
namespace: default
ownerReferences:
- apiVersion: v1
  blockOwnerDeletion: true
  controller: true
  kind: Service
  name: httpbin
  uid: 4e0d09cb-2942-49ad-9766-67e1e7352d12
resourceVersion: "983677"
uid: 2671c164-818f-44b3-92c5-30a297b7766d
ports:
- name: http
  port: 80
  protocol: TCP

```

从上面可以看到 **EndpointSlice** 对象中通过 **endpoints** 属性来保存 Endpoint 数据，每个 Endpoint 中包含 Pod IP、节点名以及关联的 Pod 对象信息，其中还包括一个 **conditions** 属性。

```

addresses:
- 10.244.1.185
conditions:
  ready: true
  serving: true
  terminating: false
nodeName: demo-worker
targetRef:
  kind: Pod
  name: httpbin-75d9685444-jbbzh
  namespace: default
  uid: b08eea07-2b99-41c3-913e-1bc03424339c

```

EndpointSlice 存储了可能对使用者有用的、有关端点的状态，分别是 **ready**、**serving** 和 **terminating**。

- Ready (就绪) : ready 是映射 Pod 的 Ready 状况的。对于处于运行中的 Pod，它的 Ready 状况被设置为 True，应该将此 EndpointSlice 状况也设置为 true。出于兼容性原因，当 Pod 处于终止过程中，ready 永远不会为 true。
- Serving (服务中) : serving 状况与 ready 状况相同，不同之处在于它不考虑终止状态。如果 EndpointSlice

API 的使用者关心 Pod 终止时的就绪情况，就应检查此状况。

- Terminating (终止中) : terminating 是表示端点是否处于终止中的状况，对于 Pod 来说，这是设置了删除时间戳的 Pod。

此外 EndpointSlice 中的每个端点都可以包含一定的拓扑信息，拓扑信息包括端点的位置，对应节点、可用区的信息，这些信息体现为 EndpointSlices 的如下端点字段：

- nodeName - 端点所在的 Node 名称
- zone - 端点所处的可用区。

为了确保多个实体可以管理 EndpointSlice 而且不会相互产生干扰，Kubernetes 定义了标签 `endpointslice.kubernetes.io/managed-by`，用来标明哪个实体在管理某个 EndpointSlice。端点切片控制器会在自己所管理的所有 EndpointSlice 上将该标签值设置为 `endpointslice-controller.k8s.io`，管理 EndpointSlice 的其他对象也应该为此标签设置一个唯一值。

此外还有一个地址类型 AddressType: IPv4、IPv6、FQDN(全限定域名)。

关于 Service 的使用还有很多其他特性，比如 LoadBalancer，我们将在后面的课程中慢慢接触到。