```
3375b86fb253db75 replicas: 2
    Updating redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 2
    At end of loop: redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 2
    At beginning of loop: redis-master replicas: 0, redis-master-ea866a5d2c08588c
3375b86fb253db75 replicas: 3
    Updating redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb253db
75 replicas: 3
    At end of loop: redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 3
    Update succeeded. Deleting old controller: redis-master
    Renaming redis-master-ea866a5d2c08588c3375b86fb253db75 to redis-master
    redis-master
```

可以看到，kubectl 通过新建一个新版本 Pod，停掉一个旧版本 Pod，逐步迭代来完成整个 RC 的更新。

更新完成后，查看 RC：

```
    $ kubectl get rc
    CONTROLLER     CONTAINER(S)   IMAGE(S)           SELECTOR        REPLICAS
    redis-master   master         kubeguide/redis-master:2.0            deployment=
ea866a5d2c08588c3375b86fb253db75,name=redis-master,version=v1   3
```

可以看到，kubectl 给 RC 增加了一个 key 为“deployment”的 Label（这个 key 的名字可通过--deployment-label-key 参数进行修改），Label 的值是 RC 的内容进行 Hash 计算后的值，相当于签名，这样就能很方便地比较 RC 里的 Image 名字及其他信息是否发生了变化，它的具体作用可以参见第 6 章的源码分析。

如果在更新过程中发现配置有误，则用户可以中断更新操作，并通过执行 kubectl rolling-update–rollback 完成 Pod 版本的回滚：

```
    $ kubectl rolling-update redis-master --image=kubeguide/redis-master:2.0 --rollback
    Found existing update in progress (redis-master-fefd9752aa5883ca4d53013a7b
583967), resuming.
    Found desired replicas.Continuing update with existing controller redis-master.
    At beginning of loop: redis-master-fefd9752aa5883ca4d53013a7b583967 replicas:
0, redis-master replicas: 3
    Updating redis-master-fefd9752aa5883ca4d53013a7b583967 replicas: 0, redis-master
replicas: 3
    At end of loop: redis-master-fefd9752aa5883ca4d53013a7b583967 replicas: 0,
redis-master replicas: 3
    Update succeeded. Deleting redis-master-fefd9752aa5883ca4d53013a7b583967
    redis-master
```

到此，可以看到 Pod 恢复到更新前的版本了。

2.5　深入掌握 Service

Service 是 Kubernetes 最核心的概念，通过创建 Service，可以为一组具有相同功能的容器应用提供一个统一的入口地址，并且将请求进行负载分发到后端的各个容器应用上。本节对 Service 的使用进行详细说明，包括 Service 的负载均衡、外网访问、DNS 服务的搭建、Ingress 7 层路由机制等。

2.5.1　Service 定义详解

yaml 格式的 Service 定义文件的完整内容如下：

```
apiVersion: v1           // Required
kind: Service            // Required
metadata:                // Required
  name: string           // Required
  namespace: string      // Required
  labels:
    - name: string
  annotations:
    - name: string
spec:                    // Required
  selector: []           // Required
  type: string           // Required
  clusterIP: string
  sessionAffinity: string
  ports:
  - name: string
    protocol: string
    port: int
    targetPort: int
    nodePort: int
  status:
    loadBalancer:
      ingress:
        ip: string
        hostname: string
```

对各属性的说明如表 2.17 所示。

表 2.17　对 Service 的定义文件模板的各属性的说明

属性名称	取值类型	是否必选	取值说明
version	string	Required	v1
kind	string	Required	Service
metadata	object	Required	元数据
metadata.name	string	Required	Service 名称，需符合 RFC 1035 规范
metadata.namespace	string	Required	命名空间，不指定系统时将使用名为“default”的命名空间
metadata.labels[]	list		自定义标签属性列表
metadata.annotation[]	list		自定义注解属性列表
spec	object	Required	详细描述
spec.selector[]	list	Required	Label Selector 配置，将选择具有指定 Label 标签的 Pod 作为管理范围
spec.type	string	Required	Service 的类型，指定 Service 的访问方式，默认为 ClusterIP。 ClusterIP：虚拟的服务 IP 地址，该地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置的 iptables 规则进行转发。 NodePort：使用宿主机的端口，使能够访问各 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务。 LoadBalancer：使用外接负载均衡器完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，并同时定义 nodePort 和 clusterIP，用于公有云环境
spec.clusterIP	string		虚拟服务 IP 地址，当 type=ClusterIP 时，如果不指定，则系统进行自动分配，也可以手工指定；当 type=LoadBalancer 时，则需要指定
spec.sessionAffinity	string		是否支持 Session，可选值为 ClientIP，默认为空。 ClientIP：表示将同一个客户端（根据客户端的 IP 地址决定）的访问请求都转发到同一个后端 Pod
spec.ports[]	list		Service 需要暴露的端口列表
spec.ports[].name	string		端口名称
spec.ports[].protocol	string		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.ports[].port	int		服务监听的端口号
spec.ports[].targetPort	int		需要转发到后端 Pod 的端口号
spec.ports[].nodePort	int		当 spec.type=NodePort 时，指定映射到物理机的端口号
Status	object		当 spec.type=LoadBalancer 时，设置外部负载均衡器的地址，用于公有云环境
status.loadBalancer	object		外部负载均衡器
status.loadBalancer.ingress	object		外部负载均衡器
status.loadBalancer.ingress.ip	string		外部负载均衡器的 IP 地址
status.loadBalancer.ingress.hostname	string		外部负载均衡器的主机名

2.5.2　Service 基本用法

一般来说，对外提供服务的应用程序需要通过某种机制来实现，对于容器应用最简便的方式就是通过 TCP/IP 机制及监听 IP 和端口号来实现。例如，我们定义一个提供 Web 服务的 RC，由两个 tomcat 容器副本组成，每个容器通过 containerPort 设置提供服务的端口号为 8080：

webapp-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: webapp
spec:
  replicas: 2
  template:
    metadata:
      name: webapp
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: tomcat
        ports:
        -containerPort: 80
```

创建该 RC：

```
# kubectl create -f webapp-rc.yaml
replicationcontroller "webapp" created
```

获取 Pod 的 IP 地址：

```
# kubectl get pods -l app=webapp -o yaml | grep podIP
    podIP: 172.17.1.4
    podIP: 172.17.1.3
```

访问这两个 Pod 提供的 Tomcat 服务：

```
# curl 172.17.1.3:8080
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
......
# curl 172.17.1.4:8080
<!DOCTYPE html>
<html lang="en">
    <head>
```

```
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
......
```

直接通过 Pod 的 IP 地址和端口号可以访问容器应用，但是 Pod 的 IP 地址是不可靠的，例如当 Pod 所在的 Node 发生故障时，Pod 将被 Kubernetes 重新调度到另一台 Node 进行启动，Pod 的 IP 地址将发生变化。更重要的是，如果容器应用本身是分布式的部署方式，通过多个实例共同提供服务，就需要在这些实例的前端设置一个负载均衡器来实现请求的分发。Kubernetes 中的 Service 就是设计出来用于解决这些问题的核心组件。

以前面创建的 webapp 应用为例，为了让客户端应用能够访问到两个 Tomcat Pod 实例，需要创建一个 Service 来提供服务。Kubernetes 提供了一种快速的方法，即通过 kubectl expose 命令来创建 Service：

```
# kubectl expose rc webapp
service "webapp" exposed
```

查看新创建的 Service，可以看到系统为它分配了一个虚拟的 IP 地址（ClusterIP），而 Service 所需的端口号则从 Pod 中的 containerPort 复制而来：

```
# kubectl get svc
NAME         CLUSTER-IP        EXTERNAL-IP    PORT(S)     AGE
webapp      169.169.235.79     <none>         8080/TCP    3s
```

接下来，我们就可以通过 Service 的 IP 地址和 Service 的端口号访问该 Service 了：

```
# curl 169.169.235.79:8080
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8" />
      <title>Apache Tomcat/8.0.35</title>
......
```

这里，对 Service 地址 169.169.235.79:8080 的访问被自动负载分发到了后端两个 Pod 之一：172.17.1.3:8080 或 172.17.1.4:8080。

除了使用 kubectl expose 命令创建 Service，我们也可以通过配置文件定义 Service，再通过 kubectl create 命令进行创建。例如对于前面的 webapp 应用，我们可以设置一个 Service，代码如下：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
```

```
  - port: 8081
    targetPort: 8080
  selector:
    app: webapp
```

Service 定义中的关键字段是 ports 和 selector。本例中 ports 定义部分指定了 Service 所需的虚拟端口号为 8081，由于与 Pod 容器端口号 8080 不一样，所以需要再通过 targetPort 来指定后端 Pod 的端口号。selector 定义部分设置的是后端 Pod 所拥有的 label：app=webapp。

创建该 Service 并查看其 ClusterIP 地址：

```
# kubectl create -f webapp-svc.yaml
service "webapp" created

# kubectl get svc
NAME         CLUSTER-IP        EXTERNAL-IP    PORT(S)     AGE
webapp       169.169.28.190    <none>         8081/TCP    3s
```

通过 Service 的 IP 地址和 Service 的端口号进行访问：

```
# curl 169.169.28.190:8081
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
......
```

同样，对 Service 地址 169.169.28.190:8081 的访问被自动负载分发到了后端两个 Pod 之一：172.17.1.3:8080 或 172.17.1.4:8080。目前 Kubernetes 提供了两种负载分发策略：RoundRobin 和 SessionAffinity，具体说明如下。

- RoundRobin：轮询模式，即轮询将请求转发到后端的各个 Pod 上。
- SessionAffinity：基于客户端 IP 地址进行会话保持的模式，即第 1 次将某个客户端发起的请求转发到后端的某个 Pod 上，之后从相同的客户端发起的请求都将被转发到后端相同的 Pod 上。

在默认情况下，Kubernetes 采用 RoundRobin 模式进行路由选择，但我们也可以通过将 service.spec.sessionAffinity 设置为“ClientIP”来启用 SessionAffinity 策略，这样，同一个客户端发来的请求就会建立一个 Session，并且对应到后端固定的某个 Pod 上了。

在某些应用场景中，开发人员希望自己控制负载均衡的策略，不使用 Service 提供的默认负载均衡的功能，Kubernetes 通过 Headless Service 的概念来实现这种功能，即不给 Service 设置 ClusterIP（无入口 IP 地址），而仅通过 Label Selector 将后端的 Pod 列表返回给调用的客户端。例如：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
  clusterIP: None
  selector:
    app: nginx
```

该 Service 没有虚拟的 ClusterIP 地址，对其进行访问将获得具有 Label“app=nginx”的全部 Pod 列表，然后客户端程序需要实现自己的负载分发策略，再确定访问具体哪一个后端的 Pod。

在某些环境中，应用系统需要将一个外部数据库作为后端服务进行连接，或将另一个集群或 Namespace 中的服务作为服务的后端，这时可以通过创建一个无 Label Selector 的 Service 来实现：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

通过该定义创建的是一个不带标签选择器的 Service，即无法选择后端的 Pod，系统不会自动创建 Endpoint，因此需要手动创建一个和该 Service 同名的 Endpoint，用于指向实际的后端访问地址。创建 Endpoint 的配置文件内容如下：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
- addresses:
  - IP: 1.2.3.4
  ports:
  - port: 80
```

如图 2.16 所示，访问没有标签选择器的 Service 和带有标签选择器的 Service 一样，请求将会被路由到由用户手动定义的后端 Endpoint 上。

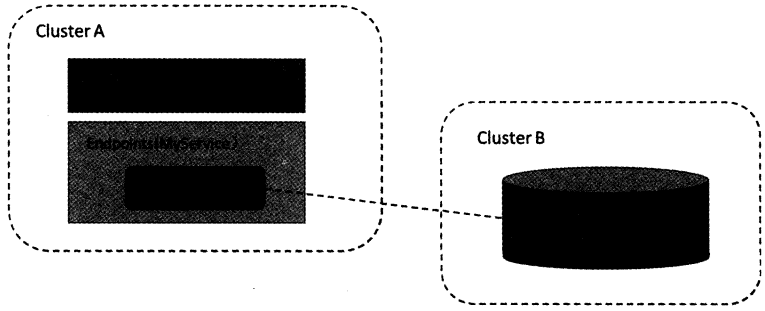

图 2.16　Service 指向外部服务

有时，一个容器应用也可能提供多个端口的服务，所以在 Service 的定义中也可以相应地设置为多个端口。在下面的例子中，Service 设置了两个端口号，并且为每个端口号进行了命名：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
  - port: 8080
    targetPort: 8080
    name: web
  - port: 8005
    targetPort: 8005
    name: management
  selector:
    app: webapp
```

另一个例子是两个端口号使用了不同的 4 层协议，即 TCP 或 UDP：

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
  - name: dns
    port: 53
```

```
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

2.5.3 集群外部访问 Pod 或 Service

由于 Pod 和 Service 是 Kubernetes 集群范围内的虚拟概念，所以集群外的客户端系统无法通过 Pod 的 IP 地址或者 Service 的虚拟 IP 地址和虚拟端口号访问到它们。为了让外部客户端可以访问这些服务，可以将 Pod 或 Service 的端口号映射到宿主机，以使得客户端应用能够通过物理机访问容器应用。

1. 将容器应用的端口号映射到物理机

（1）通过设置容器级别的 hostPort，将容器应用的端口号映射到物理机上：

pod-hostport.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  containers:
  - name: webapp
    image: tomcat
    ports:
    - containerPort: 8080
      hostPort: 8081
```

通过 kubectl create 命令创建这个 Pod：

```
# kubectl create -f pod-hostport.yaml
pod "webapp" created
```

通过物理机的 IP 地址和 8081 端口号访问 Pod 内的容器服务：

```
# curl 192.168.18.3:8081
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
......
```

（2）通过设置 Pod 级别的 hostNetwork=true，该 Pod 中所有容器的端口号都将被直接映射到物理机上。设置 hostNetwork=true 时需要注意，在容器的 ports 定义部分如果不指定 hostPort，则默认 hostPort 等于 containerPort，如果指定了 hostPort，则 hostPort 必须等于 containerPort 的值。

pod-hostnetwork.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  hostNetwork: true
  containers:
  - name: webapp
    image: tomcat
    imagePullPolicy: Never
    ports:
    - containerPort: 8080
```

创建这个 Pod：

```
# kubectl create -f pod-hostnetwork.yaml
pod "webapp" created
```

通过物理机的 IP 地址和 8080 端口号访问 Pod 内的容器服务：

```
# curl 192.168.18.4:8080
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
......
```

2. 将 Service 的端口号映射到物理机

（1）通过设置 nodePort 映射到物理机，同时设置 Service 的类型为 NodePort：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
```

```
    nodePort: 8081
  selector:
    app: webapp
```

创建这个 Service：

```
# kubectl create -f webapp-svc-nodeport.yaml
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet, you may
need to set up firewall rules for the service port(s) (tcp:8081) to serve traffic.

See http://releases.k8s.io/release-1.3/docs/user-guide/services-firewalls.md
for more details.
service "webapp" created
```

系统提示信息说明：由于要使用物理机的端口号，所以需要在防火墙上做好相应的配置，以使得外部客户端能够访问到该端口。

通过物理机的 IP 地址和 nodePort 8081 端口号访问服务：

```
# curl 192.168.18.3:8081
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Apache Tomcat/8.0.35</title>
......
```

同样，对该 Service 的访问也将被负载分发到后端的多个 Pod 上。

（2）通过设置 LoadBalancer 映射到云服务商提供的 LoadBalancer 地址。这种用法仅用于在公有云服务提供商的云平台上设置 Service 的场景。在下面的例子中，status.loadBalancer.ingress.ip 设置的 146.148.47.155 为云服务商提供的负载均衡器的 IP 地址。对该 Service 的访问请求将会通过 LoadBalancer 转发到后端 Pod 上，负载分发的实现方式则依赖于云服务商提供的 LoadBalancer 的实现机制。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
    nodePort: 30061
```

```
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 146.148.47.155
```

2.5.4　DNS 服务搭建指南

根据第 1 章对 Service 概念的说明，为了能够通过服务的名字在集群内部进行服务的相互访问，需要创建一个虚拟的 DNS 服务来完成服务名到 ClusterIP 的解析。本节将对如何搭建 DNS 服务进行详细说明。

Kubernetes 提供的虚拟 DNS 服务名为 skydns，由四个组件组成。

（1）etcd：DNS 存储。

（2）kube2sky：将 Kubernetes Master 中的 Service（服务）注册到 etcd。

（3）skyDNS：提供 DNS 域名解析服务。

（4）healthz：提供对 skydns 服务的健康检查功能。

图 2.17 描述了 Kubernetes DNS 服务的总体架构。

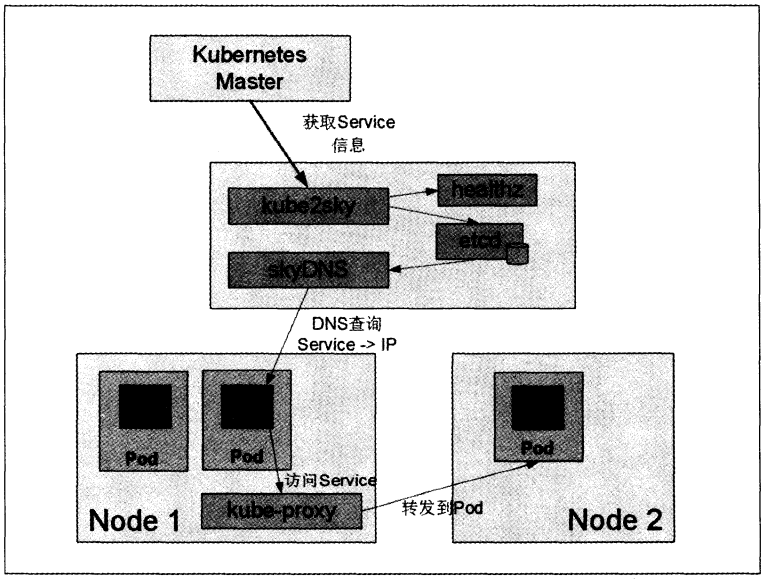

图 2.17　Kubernetes DNS 服务的总体架构

1. skydns 配置文件说明

skydns 服务由一个 RC 和一个 Service 的定义组成，分别由配置文件 skydns-rc.yaml 和 skydns-svc.yaml 定义。

skydns 的 RC 配置文件 skydns-rc.yaml 的内容如下，包含了 4 个容器的定义：

skydns-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v11
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    version: v11
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v11
  template:
    metadata:
      labels:
        k8s-app: kube-dns
        version: v11
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: etcd
        image: gcr.io/google_containers/etcd-amd64:2.2.1
        resources:
          limits:
            cpu: 100m
            memory: 50Mi
          requests:
            cpu: 100m
            memory: 50Mi
        command:
        - /usr/local/bin/etcd
        - -data-dir
        - /tmp/data
        - -listen-client-urls
        - http://127.0.0.1:2379,http://127.0.0.1:4001
        - -advertise-client-urls
        - http://127.0.0.1:2379,http://127.0.0.1:4001
```

```
    - -initial-cluster-token
    - skydns-etcd
    volumeMounts:
    - name: etcd-storage
      mountPath: /tmp/data
  - name: kube2sky
    image: gcr.io/google_containers/kube2sky-amd64:1.15
    resources:
      limits:
        cpu: 100m
        # Kube2sky watches all pods.
        memory: 50Mi
      requests:
        cpu: 100m
        memory: 50Mi
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        scheme: HTTP
      initialDelaySeconds: 60
      timeoutSeconds: 5
      successThreshold: 1
      failureThreshold: 5
    readinessProbe:
      httpGet:
        path: /readiness
        port: 8081
        scheme: HTTP
      # we poll on pod startup for the Kubernetes master service and
      # only setup the /readiness HTTP server once that's available.
      initialDelaySeconds: 30
      timeoutSeconds: 5
    args:
    # command = "/kube2sky"
    - --kube-master-url=http://192.168.18.3:8080
    - --domain=cluster.local
  - name: skydns
    image: gcr.io/google_containers/skydns:2015-10-13-8c72f8c
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 100m
        memory: 50Mi
    args:
```

```
        # command = "/skydns"
        - -machines=http://127.0.0.1:4001
        - -addr=0.0.0.0:53
        - -ns-rotate=false
        - -domain=cluster.local
        ports:
        - containerPort: 53
          name: dns
          protocol: UDP
        - containerPort: 53
          name: dns-tcp
          protocol: TCP
      - name: healthz
        image: gcr.io/google_containers/exechealthz:1.0
        resources:
          # keep request = limit to keep this container in guaranteed class
          limits:
            cpu: 10m
            memory: 20Mi
          requests:
            cpu: 10m
            memory: 20Mi
        args:
        - -cmd=nslookup kubernetes.default.svc.cluster.local 127.0.0.1 >/dev/null
        - -port=8080
        ports:
        - containerPort: 8080
          protocol: TCP
      volumes:
      - name: etcd-storage
        emptyDir: {}
      dnsPolicy: Default  # Don't use cluster DNS.
```

需要修改的几个配置参数如下。

（1）kube2sky 容器需要访问 Kubernetes Master，需要配置 Master 所在物理主机的 IP 地址和端口号，本例中设置参数--kube_master_url 的值为 http://192.168.18.3:8080。

（2）kube2sky 容器和 skydns 容器的启动参数--domain，设置 Kubernetes 集群中 Service 所属的域名，本例中为“cluster.local”。启动后，kube2sky 会通过 API Server 监控集群中全部 Service 的定义，生成相应的记录并保存到 etcd 中。kube2sky 为每个 Service 生成以下两条记录。

- <service_name>.<namespace_name>.<domain>。
- <service_name>.<namespace_name>.svc.<domain>。

（3）skydns 的启动参数-addr=0.0.0.0:53 表示使用本机 TCP 和 UDP 的 53 端口提供服务。

skydns 的 Service 配置文件 skydns-svc.yaml 的内容如下：

```
skydns-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

注意，skydns 服务使用的 clusterIP 需要我们指定一个固定的 IP 地址，每个 Node 的 kubelet 进程都将使用这个 IP 地址，不能通过 Kubernetes 自动分配。

另外，这个 IP 地址需要在 kube-apiserver 启动参数--service-cluster-ip-range 指定的 IP 地址范围内。

在创建 skydns 容器之前，先修改每个 Node 上 kubelet 的启动参数。

2. 修改每台 Node 上的 kubelet 启动参数

修改每台 Node 上 kubelet 的启动参数，加上以下两个参数。

- --cluster_dns=169.169.0.100：为 DNS 服务的 ClusterIP 地址。
- --cluster_domain=cluster.local：为 DNS 服务中设置的域名。

然后重启 kubelet 服务。

3. 创建 skydns RC 和 Service

通过 kubectl create 完成 skydns 的 RC 和 Service 的创建：

```
# kubectl create -f skydns-rc.yaml
```

```
# kubectl create -f skydns-svc.yaml
```

查看 RC、Pod 和 Service，确保容器成功启动：

```
# kubectl get rc --namespace=kube-system
NAME                          DESIRED        CURRENT       AGE
kube-dns-v11                   1              1            1d

# kubectl get pods --namespace=kube-system
NAME                          READY          STATUS        RESTARTS        AGE
kube-dns-v11-6d1wu              4/4            Running       0               1d

# kubectl get services --namespace=kube-system
NAME                CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
kube-dns            169.169.0.100   <none>        53/UDP,53/TCP   1d
```

然后，我们为 redis-master 应用创建一个 Service：

redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    name: redis-master
```

查看创建好的 redis-master service：

```
# kubectl get services
NAME                CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
redis-master         169.169.8.10   <none>        6379/TCP        1h
```

可以看到，系统为 redis-master 服务分配了一个虚拟 IP 地址：169.169.8.10。

到此，在 Kubernetes 集群内的虚拟 DNS 服务就搭建好了。在需要访问 redis-master 的应用中，仅需要配置上 redis-master Service 的名称和服务的端口号，就能够访问到 redis-master 应用了，让我们回顾一下 redis-slave 应用需要访问 redis-master 的配置内容：

redis-slave 镜像的启动脚本/run.sh 的内容为：

```
if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
  redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST} 6379
else
  redis-server --slaveof redis-master 6379
```

```
fi
```

在使用 DNS 模式的情况下，redis-slave 配置的 Master 地址为：redis-master:6379。通过服务名进行配置，能够极大地简化客户端应用对后端服务变化的感知，包括服务虚拟 IP 地址的变化、服务后端 Pod 的变化等，对应用程序的微服务架构实现提供了强有力的支撑。

4. 通过 DNS 查找 Service

接下来使用一个带有 nslookup 工具的 Pod 来验证 DNS 服务是否能够正常工作：

busybox.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: gcr.io/google_containers/busybox
    command:
      - sleep
      - "3600"
```

运行 kubectl create -f busybox.yaml 完成创建。

在该容器成功启动后，通过 kubectl exec <container_id> nslookup 进行测试：

```
# kubectl exec busybox -- nslookup redis-master
Server:    169.169.0.100
Address 1: 169.169.0.100

Name:      redis-master
Address 1: 169.169.8.10
```

可以看到，通过 DNS 服务器 169.169.0.100 成功找到了名为“redis-master”服务的 IP 地址：169.169.8.10。

如果某个 Service 属于不同的命名空间，那么在进行 Service 查找时，需要带上 namespace 的名字。下面以查找 kube-dns 服务为例：

```
# kubectl exec busybox -- nslookup kube-dns.kube-system
Server:    169.169.0.100
Address 1: 169.169.0.100

Name:      kube-dns.kube-system
Address 1: 169.169.0.100
```

如果仅使用“kube-dns”来进行查找，则将会失败：

```
nslookup: can't resolve 'kube-dns'
```

5. DNS 服务的工作原理解析

让我们看看 DNS 服务背后的工作原理。

（1）kube2sky 容器应用通过调用 Kubernetes Master 的 API 获得集群中所有 Service 的信息，并持续监控新 Service 的生成，然后写入 etcd 中。

查看 etcd 中存储的 Service 信息：

```
# kubectl exec kube-dns-v8-5tpm2 -c etcd --namespace=kube-system etcdctl ls
/skydns/local/cluster
/skydns/local/cluster/default
/skydns/local/cluster/svc
/skydns/local/cluster/kube-system
```

可以看到在 skydns 键下面，根据我们配置的域名（cluster.local）生成了 local/cluster 子键，接下来是 namespace（default 和 kube-system）和 svc（下面也按 namespace 生成子键）。

查看 redis-master 服务对应的键值：

```
# kubectl exec kube-dns-v8-5tpm2 -c etcd --namespace=kube-system etcdctl get /
skydns/local/cluster/default/redis-master
{"host":"169.169.8.10","priority":10,"weight":10,"ttl":30,"targetstrip":0}
```

可以看到，redis-master 服务对应的完整域名为 redis-master.default.cluster.local，并且其 IP 地址为 169.169.8.10。

（2）根据 kubelet 启动参数的设置（--cluster_dns），kubelet 会在每个新创建的 Pod 中设置 DNS 域名解析配置文件/etc/resolv.conf 文件，在其中增加了一条 nameserver 配置和一条 search 配置：

```
nameserver 169.169.0.100
search default.svc.cluster.local svc.cluster.local cluster.local localdomain
```

通过名字服务器 169.169.0.100 访问的实际上就是 skydns 在 53 端口上提供的 DNS 解析服务。

（3）最后，应用程序就能够像访问网站域名一样，仅仅通过服务的名字就能访问到服务了。

仍然以 redis-slave 为例，假设已经启动了 redis-slave Pod，登录 redis-slave 容器进行查看，可以看到其通过 DNS 域名服务找到了 redis-master 的 IP 地址 169.169.8.10，并成功建立了连接。

2.5.5　Ingress：HTTP 7 层路由机制

根据前面对 Service 的使用说明，我们知道 Service 的表现形式为 IP:Port，即工作在 TCP/IP 层。而对于基于 HTTP 的服务来说，不同的 URL 地址经常对应到不同的后端服务或者虚拟服务器（Virtual Host），这些应用层的转发机制仅通过 Kubernetes 的 Service 机制是无法实现的。Kubernetes v1.1 版本中新增的 Ingress 将不同 URL 的访问请求转发到后端不同的 Service，实现 HTTP 层的业务路由机制。在 Kubernetes 集群中，Ingress 的实现需要通过 Ingress 的定义与 Ingress Controller 的定义结合起来，才能形成完整的 HTTP 负载分发功能。

图 2.18 显示了一个典型 HTTP 层路由的例子。

- 对 http://mywebsite.com/api 的访问将被路由到后端名为“api”的 Service。
- 对 http://mywebsite.com/web 的访问将被路由到后端名为“web”的 Service。
- 对 http://mywebsite.com/doc 的访问将被路由到后端名为“doc”的 Service。

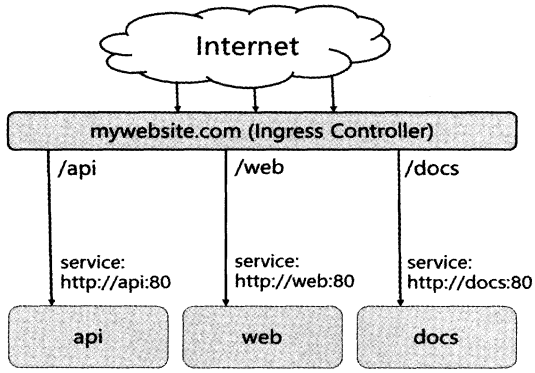

图 2.18　Ingress 示例

1. 创建 Ingress Controller

在定义 Ingress 之前，需要先部署 Ingress Controller，以实现为所有后端 Service 提供一个统一的入口。Ingress Controller 需要实现基于不同 HTTP URL 向后转发的负载分发规则，通常应该根据应用系统的需求进行个性化实现。如果公有云服务商能够提供该类型的 HTTP 路由 LoadBalancer，则也可设置其为 Ingress Controller。

在 Kubernetes 中，Ingress Controller 将以 Pod 的形式运行，监控 apiserver 的/ingress 接口（在 1.3 版本中为/apis/extensions/v1beta1/namespaces/<namespace_name>/ingresses 接口）后端的 backend services，如果 service 发生变化，则 Ingress Controller 应自动更新其转发规则。

在下面的例子中，我们使用 Nginx 来实现一个 Ingress Controller，需要实现的基本逻辑如下。

（1）监听 apiserver，获取全部 ingress 的定义。

（2）基于 ingress 的定义，生成 Nginx 所需的配置文件/etc/nginx/nginx.conf。

（3）执行 nginx -s reload 命令，重新加载 nginx.conf 配置文件的内容，

基于 Go 语言的代码实现如下：

```
for {
    rateLimiter.Accept()
    ingresses, err := ingClient.List(labels.Everything(), fields.Everything())
    if err != nil || reflect.DeepEqual(ingresses.Items, known.Items) {
        continue
    }
    if w, err := os.Create("/etc/nginx/nginx.conf"); err != nil {
        log.Fatalf("Failed to open %v: %v", nginxConf, err)
    } else if err := tmpl.Execute(w, ingresses); err != nil {
        log.Fatalf("Failed to write template %v", err)
    }
    shellOut("nginx -s reload")
}
```

我们可以通过直接下载谷歌提供的 nginx-ingress 镜像来创建 Ingress Controller：

nginx-ingress-rc.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-ingress
  labels:
    app: nginx-ingress
spec:
  replicas: 1
  selector:
    app: nginx-ingress
  template:
    metadata:
      labels:
        app: nginx-ingress
    spec:
      containers:
      - image: gcr.io/google_containers/nginx-ingress:0.1
        name: nginx
        ports:
        - containerPort: 80
          hostPort: 80
```

这里，该 Nginx 应用设置了 hostPort，即它将容器应用监听的 80 端口号映射到物理机，以使得客户端应用可以通过 URL 地址“http://物理机 IP:80”来访问该 Ingress Controller。

通过 kubectl create 命令创建该 RC：

```
# kubectl create -f nginx-ingress-rc.yaml
replicationcontroller "nginx-ingress" created

# kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
nginx-ingress-mrwtz   1/1       Running   0          2s
```

2. 定义 Ingress

为 mywebsite.com 定义 Ingress，设置到后端 Service 的转发规则：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: mywebsite-ingress
spec:
  rules:
  - host: mywebsite.com
    http:
      paths:
      - path: /web
        backend:
          serviceName: webapp
          servicePort: 80
```

这个 Ingress 的定义说明对目标 URL http://mywebsite.com/web 的访问将被转发到 Kubernetes 的一个 Service 上：webapp:80。

创建该 Ingress：

```
# kubectl create -f ingress.yaml
ingress "mywebsite-ingress" created

# kubectl get ingress
NAME              HOSTS          ADDRESS   PORTS     AGE
mywebsite-ingress   mywebsite.com           80        17s
```

在该 Ingress 成功创建后，登录 nginx-ingress Pod，查看其自动生成的 nginx.conf 配置文件内容：

```
events {
  worker_connections 1024;
}
http {
  server {
    listen 80;
    server_name mywebsite.com;    # Ingress 中定义的虚拟 host 名
```

```
    resolver 127.0.0.1;

    location /web {                  # Ingress 中定义的路径 /web
      proxy_pass http://webapp;      # service 名
    }
  }
}
```

3. 访问 http://mywebsite.com/web

由于 Ingress Controller 设置了 hostPort，所以我们可以通过其所在的物理机对其进行访问。可以在物理机上设置 mywebsite.com 对应的 IP 地址，也可以通过 curl --resolve 进行指定：

```
$ curl --resolve mywebsite.com:80:192.168.18.3 mywebsite.com/foo
```

将获得 Kubernetes Service“webapp:80”提供的主页。

4. Ingress 的发展路线

当前的 Ingress 还是 beta 版本，在 Kubernetes 的后续版本中将增加至少以下功能。

- 支持更多 TLS 选项，例如 SNI、重加密等。
- 支持 L4 和 L7 负载均衡策略（目前只支持 HTTP 层的规则）。
- 支持更多的转发规则（目前仅支持基于 URL 路径的），例如重定向规则、会话保持规则等。