```
    - name: slave
      image: kubeguide/guestbook-redis-slave
      env:
      - name: GET_HOSTS_FROM
        value: env
      ports:
      - containerPort: 6379
```

在容器的配置部分设置了一个环境变量 GET_HOSTS_FROM=env，意思是从环境变量中获取 redis-master 服务的 IP 地址信息。

redis-slave 镜像中的启动脚本/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
```

在创建 redis-slave Pod 时，系统将自动在容器内部生成之前已经创建好的 redis-master service 相关的环境变量，所以 redis-slave 应用程序 redis-server 可以直接使用环境变量 REDIS_MASTER_SERVICE_HOST 来获取 redis-master 服务的 IP 地址。

如果在容器配置部分不设置该 env，则将使用 redis-master 服务的名称“redis-master”来访问它，这将使用 DNS 方式的服务发现，需要预先启动 Kubernetes 集群的 skydns 服务，详见 2.5.4 节的说明。

运行 kubectl create 命令：

```
$ kubectl create -f redis-slave-controller.yaml
Replicationcontrollers "redis-slave" created
```

运行 kubectl get 命令查看 RC：

```
$ kubectl get rc
NAME              DESIRED   CURRENT   AGE
redis-master        1         1         1h
redis-slave         2         2         1h
```

查看 RC 创建的 Pod，可以看到有两个 redis-slave Pod 在运行：

```
$ kubectl get pods
NAME                READY      STATUS     RESTARTS   AGE
redis-master-b03io  1/1        Running    0          1h
redis-slave-10ahl   1/1        Running    0          1h
redis-slave-c5y10   1/1        Running    0          1h
```

然后创建 redis-slave 服务。类似于 redis-master 服务，与 redis-slave 相关的一组环境变量也将在后续新建的 frontend Pod 中由系统自动生成。

配置文件 redis-slave-service.yaml 的内容如下：

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

运行 kubectl 创建 Service：

```
$ kubectl create -f redis-slave-service.yaml
services/redis-slave
```

通过 kubectl 查看创建的 Service：

```
$ kubectl get services
NAME              CLUSTER-IP        EXTERNAL-IP   PORT(S)    AGE
frontend          169.169.167.153   <nodes>       80/TCP     25m
redis-master      169.169.208.57    <none>        6379/TCP   25m
redis-slave       169.169.78.102    <none>        6379/TCP   25m
```

2.3.3　创建 frontend RC 和 Service

类似地，定义 frontend 的 RC 配置文件——frontend-controller.yaml，内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
      - name: frontend
```

```
        image: kubeguide/guestbook-php-frontend
        env:
        - name: GET_HOSTS_FROM
          value: env
        ports:
        - containerPort: 80
```

在容器的配置部分设置了一个环境变量 GET_HOSTS_FROM=env，意思是从环境变量中获取 redis-master 和 redis-slave 服务的 IP 地址信息。

容器镜像名为 kubeguide/guestbook-php-frontend，该镜像中所包含的 PHP 的留言板源码（guestbook.php）如下：

```
<?
set_include_path('.:/usr/local/lib/php');
error_reporting(E_ALL);
ini_set('display_errors', 1);
require 'Predis/Autoloader.php';
Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
  $host = 'redis-master';
  if (getenv('GET_HOSTS_FROM') == 'env') {
    $host = getenv('REDIS_MASTER_SERVICE_HOST');
  }
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);

    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
  } else {
    $host = 'redis-slave';
    if (getenv('GET_HOSTS_FROM') == 'env') {
      $host = getenv('REDIS_SLAVE_SERVICE_HOST');
    }
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);

    $value = $client->get($_GET['key']);
```

```
    print('{"data": "' . $value . '"}');
  }
} else {
  phpinfo();
} ?>
```

这段 PHP 代码的意思是，如果是一个 set 请求（提交留言），则会连接到 redis-master 服务进行写数据操作，其中 redis-master 服务的虚拟 IP 地址是用之前提过的从环境变量中获取的方式得到的，端口使用默认的 6379 端口号（当然，也可以使用环境变量'REDIS_MASTER_SERVICE_PORT'的值）；如果是 get 请求，则会连接到 redis-slave 服务进行读数据操作。

可以看到，如果在容器配置部分不设置 env “GET_HOSTS_FROM”，则将使用 redis-master 或 redis-slave 服务名来访问这两个服务，这将使用 DNS 方式的服务发现，需要预先启动 Kubernetes 集群的 skydns 服务，详见 2.5.4 节的说明。

运行 kubectl create 命令创建 RC：

```
$ kubectl create -f frontend-controller.yaml
replicationcontrollers "frontend" created
```

查看已创建的 RC：

```
$ kubectl get rc
NAME           DESIRED   CURRENT   AGE
frontend       3         3         1h
redis-master   1         1         1h
redis-slave    2         2         1h
```

再查看生成的 Pod：

```
$ kubectl get pods
NAME                 READY     STATUS    RESTARTS   AGE
redis-master-b03io   1/1       Running   0          1h
redis-slave-10ahl    1/1       Running   0          1h
redis-slave-c5y10    1/1       Running   0          1h
frontend-4o11g       1/1       Running   0          1h
frontend-u9aq6       1/1       Running   0          1h
frontend-yga1l       1/1       Running   0          1h
```

最后创建 frontend Service，主要目的是使用 Service 的 NodePort 给 Kubernetes 集群中的 Service 映射一个外网可以访问的端口，这样一来，外部网络就可以通过 NodeIP+NodePort 的方式访问集群中的服务了。

服务定义文件 frontend-service.yaml 的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
```

```
  labels:
    name: frontend
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30001
  selector:
    name: frontend
```

这里的关键点是设置 type=NodePort 并指定一个 NodePort 的值，表示使用 Node 上的物理机端口提供对外访问的能力。需要注意的是，spec.ports.NodePort 的端口号范围可以进行限制（通过 kube-apiserver 的启动参数--service-node-port-range 指定），默认为 30000～32767，如果指定为可用 IP 范围之外的其他端口号，则 Service 的创建将会失败。

运行 kubectl create 命令创建 Service：

```
$ kubectl create -f frontend-service.yaml
Services "frontend" created
```

通过 kubectl 查看创建的 Service：

```
$ kubectl get services
NAME              CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
frontend          169.169.167.153  <nodes>       80/TCP     25m
redis-master      169.169.208.57   <none>        6379/TCP   25m
redis-slave       169.169.78.102   <none>        6379/TCP   25m
```

2.3.4 通过浏览器访问 frontend 页面

经过上面的三个步骤就搭建好了 Guestbook 留言板系统，总共包括 3 个应用的 6 个实例，都运行在 Kubernetes 集群中。打开浏览器，在地址栏输入 http://虚拟机 IP:30001/，将看到如图 2.6 所示的网页，并且看到网页上有一条留言——“Hello World!”。

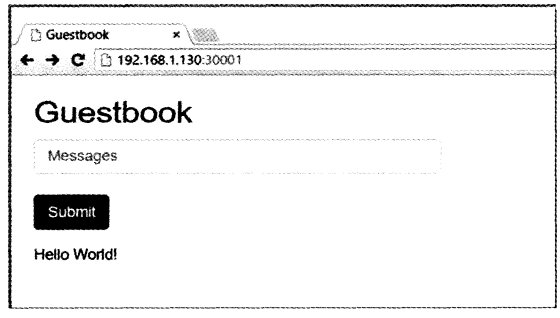

图 2.6 通过浏览器访问留言板网页

尝试输入一条新的留言“Hi Kubernetes!”，单击 Submit 按钮，网页将会在原留言的下方显示新的留言，说明这条留言已经被成功加入 Redis 数据库中了，如图 2.7 所示。

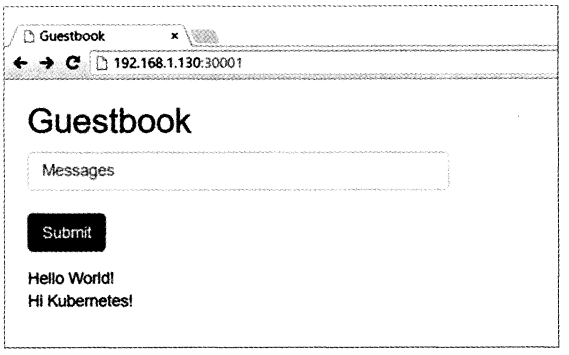

图 2.7　在留言板网页添加新的留言

通过 Guestbook 示例，可以看到 Kubernetes 强大的应用管理功能，用户仅需通过几个简单的 YAML 配置就能完成复杂系统的搭建，并能够通过 Kubernetes 自动实现服务发现和负载均衡。接下来，让我们深入 Pod 的应用、配置、调度管理及服务的应用，开始 Kubernetes 应用管理之旅。

2.4　深入掌握 Pod

本节将对 Kubernetes 如何发布和管理应用进行详细说明和示例，主要包括 Pod 和容器的使用、Pod 的控制和调度管理、应用配置管理等内容。

2.4.1　Pod 定义详解

yaml 格式的 Pod 定义文件的完整内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: string
  namespace: string
  labels:
    - name: string
  annotations:
    - name: string
spec:
```

```
containers:
- name: string
  image: string
  imagePullPolicy: [Always | Never | IfNotPresent]
  command: [string]
  args: [string]
  workingDir: string
  volumeMounts:
  - name: string
    mountPath: string
    readOnly: boolean
  ports:
  - name: string
    containerPort: int
    hostPort: int
    protocol: string
  env:
  - name: string
    value: string
  resources:
    limits:
      cpu: string
      memory: string
    requests:
      cpu: string
      memory: string
  livenessProbe:
    exec:
      command: [string]
    httpGet:
      path: string
      port: number
      host: string
      scheme: string
      httpHeaders:
      - name: string
        value: string
    tcpSocket:
      port: number
    initialDelaySeconds: 0
    timeoutSeconds: 0
    periodSeconds: 0
    successThreshold: 0
    failureThreshold: 0
```

```
  securityContext:
    privileged: false
restartPolicy: [Always | Never | OnFailure]
nodeSelector: object
imagePullSecrets:
- name: string
hostNetwork: false
volumes:
- name: string
  emptyDir: {}
  hostPath:
    path: string
  secret:
    secretName: string
    items:
    - key: string
      path: string
  configMap:
    name: string
    items:
    - key: string
      path: string
```

对各属性的详细说明如表 2.13 所示。

表 2.13　对 Pod 定义文件模板中各属性的详细说明

属性名称	取值类型	是否必选	取值说明
version	String	Required	版本号，例如 v1
kind	String	Required	Pod
metadata	Object	Required	元数据
metadata.name	String	Required	Pod 的名称，命名规范需符合 RFC 1035 规范
metadata.namespace	String	Required	Pod 所属的命名空间，默认为“default”
metadata.labels[]	List		自定义标签列表
metadata.annotation[]	List		自定义注解列表
Spec	Object	Required	Pod 中容器的详细定义
spec.containers[]	List	Required	Pod 中的容器列表
spec.containers[].name	String	Required	容器的名称，需符合 RFC 1035 规范
spec.containers[].image	String	Required	容器的镜像名称

续表

属性名称	取值类型	是否必选	取值说明
spec.containers[].imagePullPolicy	String		获取镜像的策略，可选值包括：Always、Never、IfNotPresent，默认值为 Always。 Always：表示每次都尝试重新下载镜像。 IfNotPresent：表示如果本地有该镜像，则使用本地的镜像，本地不存在时下载镜像。 Never：表示仅使用本地镜像
spec.containers[].command[]	List		容器的启动命令列表，如果不指定，则使用镜像打包时使用的启动命令
spec.containers[].args[]	List		容器的启动命令参数列表
spec.containers[].workingDir	String		容器的工作目录
spec.containers[].volumeMounts[]	List		挂载到容器内部的存储卷配置
spec.containers[].volumeMounts[].name	String		引用 Pod 定义的共享存储卷的名称，需使用 volumes[]部分定义的共享存储卷名称
spec.containers[].volumeMounts[].mountPath	String		存储卷在容器内 Mount 的绝对路径，应少于 512 个字符
spec.containers[].volumeMounts[].readOnly	Boolean		是否为只读模式，默认为读写模式
spec.containers[].ports[]	List		容器需要暴露的端口号列表
spec.containers[].ports[].name	String		端口的名称
spec.containers[].ports[].containerPort	Int		容器需要监听的端口号
spec.containers[].ports[].hostPort	Int		容器所在主机需要监听的端口号，默认与 containerPort 相同。设置 hostPort 时，同一台宿主机将无法启动该容器的第 2 份副本
spec.containers[].ports[].protocol	String		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.containers[].env[]	List		容器运行前需设置的环境变量列表
spec.containers[].env[].name	String		环境变量的名称
spec.containers[].env[].value	String		环境变量的值
spec.containers[].resources	Object		资源限制和资源请求的设置，详见第 5 章的说明
spec.containers[].resources.limits	Object		资源限制的设置
spec.containers[].resources.limits.cpu	String		CPU 限制，单位为 core 数，将用于 docker run --cpu-shares 参数
spec.containers[].resources.limits.memory	String		内存限制，单位可以为 MiB/GiB 等，将用于 docker run --memory 参数
spec.containers[].resources.requests	Object		资源限制的设置

续表

属性名称	取值类型	是否必选	取值说明
spec.containers[].resources.requests.cpu	String		CPU 请求，单位为 core 数，容器启动的初始可用数量
spec.containers[].resources.requests.memory	String		内存请求，单位可以为 MiB、GiB 等，容器启动的初始可用数量
spec.volumes[]	List		在该 Pod 上定义的共享存储卷列表
spec.volumes[].name	String		共享存储卷的名称，在一个 Pod 中每个存储卷定义一个名称，应符合 RFC 1035 规范。容器定义部分的 containers[].volumeMounts[].name 将引用该共享存储卷的名称。 volume 的类型包括：emptyDir、hostPath、gcePersistentDisk、awsElasticBlockStore、gitRepo、secret、nfs、iscsi、glusterfs、persistentVolumeClaim、rbd、flexVolume、cinder、cephfs、flocker、downwardAPI、fc、azureFile、configMap、vsphereVolume，可以定义多个 volume，每个 volume 的 name 保持唯一。本节讲解 emptyDir、hostPath、secret、configMap 这 4 种 volume，其他类型 volume 的设置方式详见第 1 章的说明
spec.volumes[].emptyDir	Object		类型为 emptyDir 的存储卷，表示与 Pod 同生命周期的一个临时目录，其值为一个空对象：emptyDir: {}
spec.volumes[].hostPath	Object		类型为 hostPath 的存储卷，表示挂载 Pod 所在宿主机的目录，通过 volumes[]. hostPath.path 指定
spec.volumes[].hostPath.path	String		Pod 所在主机的目录，将被用于容器中 mount 的目录
spec.volumes[].secret	Object		类型为 secret 的存储卷，表示挂载集群预定义的 secret 对象到容器内部
spec.volumes[].configMap	Object		类型为 configMap 的存储卷，表示挂载集群预定义的 configMap 对象到容器内部
spec.volumes[].livenessProbe	Object		对 Pod 内各容器健康检查的设置，当探测无响应几次之后，系统将自动重启该容器。可以设置的方法包括：exec、httpGet 和 tcpSocket。对一个容器仅需设置一种健康检查方法
spec.volumes[].livenessProbe.exec	Object		对 Pod 内各容器健康检查的设置，exec 方式
spec.volumes[].livenessProbe.exec.command[]	String		exec 方式需要指定的命令或者脚本
spec.volumes[].livenessProbe.httpGet	Object		对 Pod 内各容器健康检查的设置，HTTPGet 方式。需指定 path、port

续表

属性名称	取值类型	是否必选	取值说明
spec.volumes[].livenessProbe.tcpSocket	Object		对 Pod 内各容器健康检查的设置，tcpSocket 方式
spec.volumes[].livenessProbe.initialDelaySeconds	Number		容器启动完成后进行首次探测的时间，单位为秒
spec.volumes[].livenessProbe.timeoutSeconds	Number		对容器健康检查的探测等待响应的超时时间设置，单位为秒，默认为 1 秒。超过该超时时间设置，将认为该容器不健康，将重启该容器
spec.volumes[].livenessProbe.periodSeconds	Number		对容器健康检查的定期探测时间设置，单位为秒，默认为 10 秒探测一次
spec.restartPolicy	String		Pod 的重启策略，可选值为 Always、OnFailure，默认值为 Always。 Always：Pod 一旦终止运行，则无论容器是如何终止的，kubelet 都将重启它。 OnFailure：只有 Pod 以非零退出码终止时，kubelet 才会重启该容器。如果容器正常结束（退出码为 0），则 kubelet 将不会重启它。 Never：Pod 终止后，kubelet 将退出码报告给 Master，不会再重启该 Pod
spec.nodeSelector	Object		设置 NodeSelector 表示将该 Pod 调度到包含这些 label 的 Node 上，以 key:value 格式指定
spec.imagePullSecrets	Object		Pull 镜像时使用的 secret 名称，以 name:secretkey 格式指定
spec.hostNetwork	Boolean		是否使用主机网络模式，默认为 false。如果设置为 true，则表示容器使用宿主机网络，不再使用 Docker 网桥，该 Pod 将无法在同一台宿主机上启动第 2 个副本

2.4.2 Pod 的基本用法

在对 Pod 的用法进行说明之前，有必要先对 Docker 容器中应用的运行要求进行说明。

在使用 Docker 时，可以使用 docker run 命令创建并启动一个容器。而在 Kubernetes 系统中对长时间运行容器的要求是：其主程序需要一直在前台执行。如果我们创建的 Docker 镜像的启动命令是后台执行程序，例如 Linux 脚本：

```
nohup ./start.sh &
```

则在 kubelet 创建包含这个容器的 Pod 之后运行完该命令，即认为 Pod 执行结束，将立刻销毁该 Pod。如果为该 Pod 定义了 ReplicationController，则系统将会监控到该 Pod 已经终止，之后根

据 RC 定义中 Pod 的 replicas 副本数量生成一个新的 Pod。而一旦创建出新的 Pod，就将在执行完启动命令后，陷入无限循环的过程中。这就是 Kubernetes 需要我们自己创建的 Docker 镜像以一个前台命令作为启动命令的原因。

对于无法改造为前台执行的应用，也可以使用开源工具 Supervisor 辅助进行前台运行的功能。Supervisor 提供了一种可以同时启动多个后台应用，并保持 Supervisor 自身在前台执行的机制，可以满足 Kubernetes 对容器的启动要求。关于 Supervisor 的安装和使用，请参考官网 http://supervisord.org 的文档说明。

接下来对 Pod 对容器的封装和应用进行说明，Pod 的基本用法为：Pod 可以由 1 个或多个容器组合而成。

在上一节 Guestbook 的例子中，名为 frontend 的 Pod 只由一个容器组成：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  containers:
  - name: frontend
    image: kubeguide/guestbook-php-frontend
    env:
    - name: GET_HOSTS_FROM
      value: env
    ports:
    - containerPort: 80
```


这个 frontend Pod 在成功启动之后，将启动 1 个 Docker 容器。

另一种场景是，当 frontend 和 redis 两个容器应用为紧耦合的关系，应该组合成一个整体对外提供服务时，则应将这两个容器打包为一个 Pod，如图 2.8 所示。

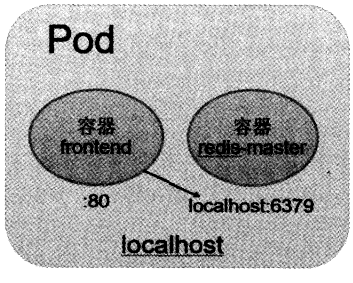

图 2.8　包含两个容器的 Pod

配置文件 frontend-localredis-pod.yaml 如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-php
  labels:
    name: redis-php
spec:
  containers:
  - name: frontend
    image: kubeguide/guestbook-php-frontend:localredis
    ports:
    - containerPort: 80
  - name: redis
    image: kubeguide/redis-master
    ports:
    - containerPort: 6379
```

属于一个 Pod 的多个容器应用之间相互访问时仅需要通过 localhost 就可以通信，使得这一组容器被“绑定”在了一个环境中。

在 Docker 容器 kubeguide/guestbook-php-frontend:localredis 的 PHP 网页中，直接通过 URL 地址“localhost:6379”对同属于一个 Pod 内的 redis-master 进行访问。guestbook.php 的内容如下：

```
<?
set_include_path('.:/usr/local/lib/php');
error_reporting(E_ALL);
ini_set('display_errors', 1);
require 'Predis/Autoloader.php';
Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
  $host = 'localhost';
  if (getenv('REDIS_HOST') && strlen(getenv('REDIS_HOST')) > 0 ) {
    $host = getenv('REDIS_HOST');
  }
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);

    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
  } else {
```

```
    $host = 'localhost';
    if (getenv('REDIS_HOST') && strlen(getenv('REDIS_HOST')) > 0 ) {
      $host = getenv('REDIS_HOST');
    }
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => $host,
      'port'   => 6379,
    ]);

    $value = $client->get($_GET['key']);
    print('{"data": "' . $value . '"}');
  }
} else {
  phpinfo();
} ?>
```

运行 kubectl create 命令创建该 Pod：

```
$ kubectl create -f frontend-localredis-pod.yaml
pod "redis-php" created
```

查看已创建的 Pod：

```
# kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
redis-php   2/2       Running   0          10m
```

可以看到 READY 信息为 2/2，表示 Pod 中的两个容器都成功运行了。

查看这个 Pod 的详细信息，可以看到两个容器的定义及创建的过程（Event 事件信息）：

```
# kubectl describe pod redis-php
Name:          redis-php
Namespace:     default
Node:          k8s/192.168.18.3
Start Time:    Thu, 28 Jul 2016 12:28:21 +0800
Labels:        name=redis-php
Status:        Running
IP:            172.17.1.4
Controllers:   <none>
Containers:
  frontend:
    Container ID:
docker://ccc8616f8df1fb19abbd0ab189a36e6f6628b78ba7b97b1077d86e7fc224ee08
    Image:                  kubeguide/guestbook-php-frontend:localredis
    Image ID:
docker://sha256:d014f67384a11186e135b95a7ed0d794674f7ce258f0dce47267c3052a0d0fa9
    Port:                   80/TCP
    State:                  Running
```

```
      Started:              Thu, 28 Jul 2016 12:28:22 +0800
    Ready:                  True
    Restart Count:          0
    Environment Variables:      <none>
  redis:
    Container ID:
docker://c0b19362097cda6dd5b8ed7d8eaaaf43aeeb969ee023ef255604bde089808075
    Image:                  kubeguide/redis-master
    Image ID:
docker://sha256:405a0b586f7ebeb545ec65be0e914311159d1baedccd3a93e9d3e3b249ec5cbd
    Port:                   6379/TCP
    State:                  Running
      Started:              Thu, 28 Jul 2016 12:28:23 +0800
    Ready:                  True
    Restart Count:          0
    Environment Variables:      <none>
Conditions:
  Type        Status
  Initialized   True
  Ready       True
  PodScheduled  True
Volumes:
  default-token-97j21:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-97j21
QoS Tier:       BestEffort
Events:
  FirstSeen   LastSeen   Count   From    SubobjectPath   Type   Reason   Message
  ---------   --------   -----   ----    -------------   -----  ------   -------
  18m         18m          1      {default-scheduler }     Normal
Scheduled   Successfully assigned redis-php to k8s-node-1
  18m         18m          1      {kubelet k8s-node-1}
spec.containers{frontend}      Normal        Pulled          Container image
"kubeguide/guestbook-php-frontend:localredis" already present on machine
  18m         18m          1      {kubelet k8s-node-1}
spec.containers{frontend}      Normal        Created         Created container
with docker id ccc8616f8df1
  18m         18m          1      {kubelet k8s-node-1}
spec.containers{frontend}      Normal        Started         Started container
with docker id ccc8616f8df1
  18m         18m          1      {kubelet k8s-node-1}
spec.containers{redis}         Normal        Pulled          Container image
"kubeguide/redis-master" already present on machine
  18m         18m          1      {kubelet k8s-node-1}
spec.containers{redis}         Normal        Created         Created container
with docker id c0b19362097c
  18m         18m          1      {kubelet k8s-node-1}
```

```
spec.containers{redis}          Normal          Started             Started container
with docker id c0b19362097c
```

2.4.3　静态 Pod

静态 Pod 是由 kubelet 进行管理的仅存在于特定 Node 上的 Pod。它们不能通过 API Server 进行管理，无法与 ReplicationController、Deployment 或者 DaemonSet 进行关联，并且 kubelet 也无法对它们进行健康检查。静态 Pod 总是由 kubelet 进行创建，并且总是在 kubelet 所在的 Node 上运行。

创建静态 Pod 有两种方式：配置文件或者 HTTP 方式。

1）配置文件方式

首先，需要设置 kubelet 的启动参数“--config”，指定 kubelet 需要监控的配置文件所在的目录，kubelet 会定期扫描该目录，并根据该目录中的.yaml 或.json 文件进行创建操作。

假设配置目录为/etc/kubelet.d/，配置启动参数：--config=/etc/kubelet.d/，然后重启 kubelet 服务。

在目录/etc/kubelet.d 中放入 static-web.yaml 文件，内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    name: static-web
spec:
  containers:
  - name: static-web
    image: nginx
    ports:
    - name: web
      containerPort: 80
```

等待一会儿，查看本机中已经启动的容器：

```
# docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS      PORTS       NAMES
2292ea231ab1    nginx    "nginx -g 'daemon off"   1 minute ago    1m
k8s_static-web.68ee0075_static-web-k8s-node-1_default_78c7efddebf191c949cbb7aa22
a927c8_401b96d0
```

可以看到一个 Nginx 容器已经被 kubelet 成功创建了出来。

到 Master 节点查看 Pod 列表，可以看到这个 static pod：

```
# kubectl get pods
NAME                READY       STATUS        RESTARTS   AGE
static-web-node1    1/1         Running       0          5m
```

由于静态 Pod 无法通过 API Server 直接管理，所以在 Master 节点尝试删除这个 Pod，将会使其变成 Pending 状态，且不会被删除。

```
# kubectl delete pod static-web-node1
pod "static-web-node1" deleted

# kubectl get pods
NAME                READY     STATUS        RESTARTS   AGE
static-web-node1    0/1       Pending       0          1s
```

删除该 Pod 的操作只能是到其所在 Node 上，将其定义文件 static-web.yaml 从/etc/kubelet.d 目录下删除。

```
# rm /etc/kubelet.d/static-web.yaml
# docker ps
// 无容器正在运行。
```

2.4.4 Pod 容器共享 Volume

在同一个 Pod 中的多个容器能够共享 Pod 级别的存储卷 Volume。Volume 可以定义为各种类型，多个容器各自进行挂载操作，将一个 Volume 挂载为容器内部需要的目录，如图 2.9 所示。

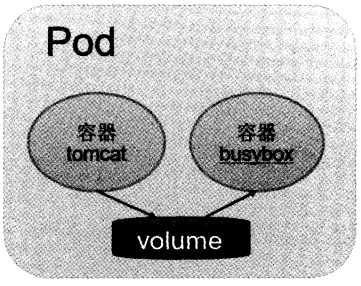

图 2.9 Pod 中多个容器共享 volume

在下面的例子中，Pod 内包含两个容器：tomcat 和 busybox，在 Pod 级别设置 Volume “app-logs”，用于 tomcat 向其中写日志文件，busybox 读日志文件。

配置文件 pod-volume-applogs.yaml 的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
```

```
    name: volume-pod
  spec:
    containers:
    - name: tomcat
      image: tomcat
      ports:
      - containerPort: 8080
      volumeMounts:
      - name: app-logs
        mountPath: /usr/local/tomcat/logs
    - name: busybox
      image: busybox
      command: ["sh", "-c", "tail -f /logs/catalina*.log"]
      volumeMounts:
      - name: app-logs
        mountPath: /logs
    volumes:
    - name: app-logs
      emptyDir: {}
```

这里设置的 Volume 名为 app-logs，类型为 emptyDir（也可以设置为其他类型，详见第 1 章对 Volume 概念的说明），挂载到 tomcat 容器内的/usr/local/tomcat/logs 目录，同时挂载到 logreader 容器内的/logs 目录。tomcat 容器在启动后会向/usr/local/tomcat/logs 目录中写文件，logreader 容器就可以读取其中的文件了。

logreader 容器的启动命令为 tail -f /logs/catalina*.log，我们可以通过 kubectl logs 命令查看 logreader 容器的输出内容：

```
# kubectl logs volume-pod -c busybox
.............
    29-Jul-2016 12:55:59.626 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application
directory /usr/local/tomcat/webapps/manager
    29-Jul-2016 12:55:59.722 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web
application directory /usr/local/tomcat/webapps/manager has finished in 96 ms
    29-Jul-2016 12:55:59.740 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["http-apr-8080"]
    29-Jul-2016 12:55:59.794 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["ajp-apr-8009"]
    29-Jul-2016 12:56:00.604 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in 4052 ms
```

这个文件即为 tomcat 生成的日志文件/usr/local/tomcat/logs/catalina.<date>.log 的内容。登录 tomcat 容器进行查看：

```
# kubectl exec -ti volume-pod -c tomcat -- ls /usr/local/tomcat/logs
```

```
    catalina.2016-07-29.log      localhost_access_log.2016-07-29.txt
    host-manager.2016-07-29.log  manager.2016-07-29.log

    # kubectl exec -ti volume-pod -c tomcat -- tail
/usr/local/tomcat/logs/catalina.2016-07-29.log
    ............
    29-Jul-2016 12:55:59.722 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web
application directory /usr/local/tomcat/webapps/manager has finished in 96 ms
    29-Jul-2016 12:55:59.740 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["http-apr-8080"]
    29-Jul-2016 12:55:59.794 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["ajp-apr-8009"]
    29-Jul-2016 12:56:00.604 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in 4052 ms
```

2.4.5 Pod 的配置管理

应用部署的一个最佳实践是将应用所需的配置信息与程序进行分离，这样可以使得应用程序被更好地复用，通过不同的配置也能实现更灵活的功能。将应用打包为容器镜像后，可以通过环境变量或者外挂文件的方式在创建容器时进行配置注入，但在大规模容器集群的环境中，对多个容器进行不同的配置将变得非常复杂。Kubernetes v1.2 版本提供了一种统一的集群配置管理方案——ConfigMap。本节对 ConfigMap 的概念和用法进行详细描述。

1. ConfigMap：容器应用的配置管理

ConfigMap 供容器使用的典型用法如下。

（1）生成为容器内的环境变量。

（2）设置容器启动命令的启动参数（需设置为环境变量）。

（3）以 Volume 的形式挂载为容器内部的文件或目录。

ConfigMap 以一个或多个 key:value 的形式保存在 Kubernetes 系统中供应用使用，既可以用于表示一个变量的值（例如 apploglevel=info），也可以用于表示一个完整配置文件的内容（例如 server.xml=<?xml...>...）

可以通过 yaml 配置文件或者直接使用 kubectl create configmap 命令行的方式来创建 ConfigMap。

2. ConfigMap 的创建：yaml 文件方式

下面的例子 cm-appvars.yaml 描述了将几个应用所需的变量定义为 ConfigMap 的用法：

```
cm-appvars.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appvars
data:
  apploglevel: info
  appdatadir: /var/data
```

执行 kubectl create 命令创建该 ConfigMap：

```
$kubectl create -f cm-appvars.yaml
configmap "cm-appvars" created
```

查看创建好的 ConfigMap：

```
# kubectl get configmap
NAME          DATA      AGE
cm-appvars    2         3s

# kubectl describe configmap cm-appvars
Name:         cm-appvars
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
appdatadir:   9 bytes
apploglevel:  4 bytes

# kubectl get configmap cm-appvars -o yaml
apiVersion: v1
data:
  appdatadir: /var/data
  apploglevel: info
kind: ConfigMap
metadata:
  creationTimestamp: 2016-07-28T19:57:16Z
  name: cm-appvars
  namespace: default
  resourceVersion: "78709"
  selfLink: /api/v1/namespaces/default/configmaps/cm-appvars
  uid: 7bb2e9c0-54fd-11e6-9dcd-000c29dc2102
```

下面的例子 cm-appconfigfiles.yaml 描述了将两个配置文件 server.xml 和 logging.properties 定义为 ConfigMap 的用法，设置 key 为配置文件的别名，value 则是配置文件的全部文本内容：

```
cm-appconfigfiles.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appconfigfiles
data:
  key-serverxml: |
    <?xml version='1.0' encoding='utf-8'?>
    <Server port="8005" shutdown="SHUTDOWN">
      <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
      <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
      <Listener className=
"org.apache.catalina.core.JreMemoryLeakPreventionListener" />
      <Listener className=
"org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
      <Listener className=
"org.apache.catalina.core.ThreadLocalLeakPreventionListener" />
      <GlobalNamingResources>
        <Resource name="UserDatabase" auth="Container"
                  type="org.apache.catalina.UserDatabase"
                  description="User database that can be updated and saved"
                  factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
                  pathname="conf/tomcat-users.xml" />
      </GlobalNamingResources>

      <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
                   connectionTimeout="20000"
                   redirectPort="8443" />
        <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
        <Engine name="Catalina" defaultHost="localhost">
          <Realm className="org.apache.catalina.realm.LockOutRealm">
            <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                   resourceName="UserDatabase"/>
          </Realm>
          <Host name="localhost"  appBase="webapps"
                unpackWARs="true" autoDeploy="true">
            <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
                   prefix="localhost_access_log" suffix=".txt"
                   pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />

          </Host>
```

```
          </Engine>
        </Service>
      </Server>
    key-loggingproperties: "handlers
      =1catalina.org.apache.juli.FileHandler, 2localhost.org.apache.juli.
FileHandler,
      3manager.org.apache.juli.FileHandler, 4host-manager.org.apache.juli.
FileHandler,
      java.util.logging.ConsoleHandler\r\n\r\n.handlers= 1catalina.org.apache.
juli.FileHandler,

java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apache.juli.FileHandler.level
      = FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHandler.prefix
      = catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
      = ${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
      = FINE\r\n3manager.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
      = manager.\r\n\r\n4host-manager.org.apache.juli.FileHandler.level =
FINE\r\n4host-manager.org.apache.juli.FileHandler.directory
      = ${catalina.base}/logs\r\n4host-manager.org.apache.juli.FileHandler.
prefix =
      host-manager.\r\n\r\njava.util.logging.ConsoleHandler.level = FINE\r\
njava.util.logging.ConsoleHandler.formatter
      = java.util.logging.SimpleFormatter\r\n\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
handlers
      = 2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/manager].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].handlers
      = 3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/host-manager].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager].handlers
      = 4host-manager.org.apache.juli.FileHandler\r\n\r\n"
```

执行 kubectl create 命令创建该 ConfigMap：

```
$kubectl create -f cm-appconfigfiles.yaml
configmap "cm-appconfigfiles" created
```

查看创建好的 ConfigMap：

```
# kubectl get configmap cm-appconfigfiles
NAME              DATA      AGE
```

```
cm-appconfigfiles   2       14s

# kubectl describe configmap cm-appconfigfiles
Name:         cm-appconfigfiles
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
key-loggingproperties:  1809 bytes
key-serverxml:          1686 bytes
```

查看已创建的 ConfigMap 的详细内容，可以看到两个配置文件的全文：

```
# kubectl get configmap cm-appconfigfiles -o yaml
apiVersion: v1
data:
  key-loggingproperties: "handlers = 1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
       3manager.org.apache.juli.FileHandler, 4host-manager.org.apache.juli.
FileHandler,
       java.util.logging.ConsoleHandler\r\n\r\n.handlers = 1catalina.org.apache.
juli.FileHandler,
       java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apache.juli.
FileHandler.level
       = FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHandler.prefix
       = catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
       = ${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
       = FINE\r\n3manager.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
       = manager.\r\n\r\n4host-manager.org.apache.juli.FileHandler.level =
FINE\r\n4host-manager.org.apache.juli.FileHandler.directory
       = ${catalina.base}/logs\r\n4host-manager.org.apache.juli.FileHandler.
prefix =
       host-manager.\r\n\r\njava.util.logging.ConsoleHandler.level = FINE\r\njava.
util.logging.ConsoleHandler.formatter
       = java.util.logging.SimpleFormatter\r\n\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].level
       = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
handlers
       = 2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/manager].level
       = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].handlers
```

```
       = 3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/host-manager].level
       = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager].handlers
       = 4host-manager.org.apache.juli.FileHandler\r\n\r\n"
     key-serverxml: |
       <?xml version='1.0' encoding='utf-8'?>
       <Server port="8005" shutdown="SHUTDOWN">
         <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
         <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
         <Listener className="org.apache.catalina.core.
JreMemoryLeakPreventionListener" />
         <Listener className="org.apache.catalina.mbeans.
GlobalResourcesLifecycleListener" />
         <Listener className="org.apache.catalina.core.
ThreadLocalLeakPreventionListener" />
         <GlobalNamingResources>
           <Resource name="UserDatabase" auth="Container"
                   type="org.apache.catalina.UserDatabase"
                   description="User database that can be updated and saved"
                   factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
                   pathname="conf/tomcat-users.xml" />
         </GlobalNamingResources>

         <Service name="Catalina">
           <Connector port="8080" protocol="HTTP/1.1"
                    connectionTimeout="20000"
                    redirectPort="8443" />
           <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
           <Engine name="Catalina" defaultHost="localhost">
             <Realm className="org.apache.catalina.realm.LockOutRealm">
               <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                      resourceName="UserDatabase"/>
             </Realm>
             <Host name="localhost"  appBase="webapps"
                   unpackWARs="true" autoDeploy="true">
               <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
                      prefix="localhost_access_log" suffix=".txt"
                      pattern="%h %l %u %t &quot;%r&quot; %s %b" />

             </Host>
           </Engine>
         </Service>
       </Server>
   kind: ConfigMap
```

```
metadata:
  creationTimestamp: 2016-07-29T00:52:18Z
  name: cm-appconfigfiles
  namespace: default
  resourceVersion: "85054"
  selfLink: /api/v1/namespaces/default/configmaps/cm-appconfigfiles
  uid: b30d5019-5526-11e6-9dcd-000c29dc2102
```

3. ConfigMap 的创建：kubectl 命令行方式

不使用 yaml 文件，直接通过 kubectl create configmap 也可以创建 ConfigMap，可以使用参数--from-file 或--from-literal 指定内容，并且可以在一行命令中指定多个参数。

（1）通过--from-file 参数从文件中进行创建，可以指定 key 的名称，也可以在一个命令行中创建包含多个 key 的 ConfigMap，语法为：

```
# kubectl create configmap NAME --from-file=[key=]source --from-file=[key=]source
```

（2）通过--from-file 参数从目录中进行创建，该目录下的每个配置文件名都被设置为 key，文件的内容被设置为 value，语法为：

```
# kubectl create configmap NAME --from-file=config-files-dir
```

（3）--from-literal 从文本中进行创建，直接将指定的 key#=value#创建为 ConfigMap 的内容，语法为：

```
# kubectl create configmap NAME --from-literal=key1=value1 --from-literal=
key2=value2
```

下面对这几种用法举例说明。

例如，当前目录下含有配置文件 server.xml，可以创建一个包含该文件内容的 ConfigMap：

```
# kubectl create configmap cm-server.xml --from-file=server.xml
configmap "cm-server.xml" created

# kubectl describe configmap cm-server.xml
Name:         cm-server.xml
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
server.xml:   6458 bytes
```

假设 configfiles 目录下包含两个配置文件 server.xml 和 logging.properties，创建一个包含这两个文件内容的 ConfigMap：