```
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 24095  bytes 2133648 (2.0 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 24095  bytes 2133648 (2.0 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

可以看出，有一个 docker0 网桥和一个本地地址的网络端口。现在部署一下我们在前面准备的 RC/Pod 配置文件，看看发生了什么：

```
    # kubectl create -f frontend-controller.yaml
    replicationcontrollers/frontend
    #
    # kubectl get pods
    NAME              READY     STATUS    RESTARTS   AGE    NODE
    frontend-4o11g   1/1       Running   0          11s    192.168.1.130
```

可以看到一些有趣的事情。Kubernetes 为这个 Pod 找了一个主机 192.168.1.130（Node2）来运行它。另外，这个 Pod 还获得了一个在 Node2 的 docker0 网桥上的 IP 地址。我们登录到 Node2 上看看发生了什么事情：

```
    # docker ps
    CONTAINER ID      IMAGE       COMMAND      CREATED      STATUS      PORTS      NAMES
    37b193a4c633      kubeguide/example-guestbook-php-redis    "/bin/sh -c /run.sh"
32 seconds ago      Up 26 seconds      k8s_php-redis.6ad3289e_frontend-n9n1m_
development_813e2dd9-8149-11e5-823b-000c2921ba71_af6dd859
    6d1b99cff4ae      google_containers/pause:latest    "/pause"    35 seconds ago
Up 28 seconds      0.0.0.0:80->80/tcp  k8s_POD.855eeb3d_frontend-4t52y_development_
813e3870-8149-11e5-823b-000c2921ba71_2b66f05e
```

在 Node2 上现在运行了两个容器。在我们的 RC/Pod 定义文件中仅仅包含了一个，那么这第 2 个是从哪里来的呢？第 2 个看起来运行的是一个叫作 google_containers/pause:latest 的镜像，而且这个容器已经有端口映射到它上面了，为什么是这样呢？让我们深入容器内部去看一下具体原因。使用 Docker 的“inspect”命令来查看容器的详细信息，特别要关注容器的网络模型。

```
    # docker inspect 6d1b99cff4ae | grep NetworkMode
          "NetworkMode": "bridge",
    # docker inspect 37b193a4c633 | grep NetworkMode
          "NetworkMode": "container:6d1b99cff4ae537689ce87d7528f4ba9dbb40ae
711ecc0a5b3f7c39ff5e5e495",
```

有趣的结果是，在查看完每个容器的网络模型后，我们可以看到这样的配置：我们检查的第 1 个容器是运行了“google_containers/pause:latest”镜像的容器，它使用了 Docker 默认的网络模型 bridge；而我们检查的第 2 个容器，也就是在我们 RC/Pod 中定义运行的 php-redis 容器，使用了非默认的网络配置和映射容器的模型，指定了映射目标容器为“google_containers/ pause:latest”。

我们一起来仔细思考一下这个过程，为什么 Kubernetes 要这么做呢？首先，一个 Pod 内的所有容器都需要共用同一个 IP 地址，这就意味着一定要使用网络的容器映射模式。然而，为什么不能只启动第 1 个 Pod 中的容器，而将第 2 个 Pod 内的容器关联到第 1 个容器呢？我们认为 Kubernetes 从两个方面来考虑这个问题：首先，如果 Pod 有超过两个容器的话，则连接这些容器可能不容易；其次，后面的容器还要依赖第 1 个被关联的容器，如果第 2 个容器关联到第 1 个容器，且第 1 个容器死掉的话，第 2 个也将死掉。启动一个基础容器，然后将 Pod 内的所有容器都连接到它上面会更容易一些。因为我们只需要为基础的这个 Google_containers/pause 容器执行端口映射规则，这也简化了端口映射的过程。所以我们的 Pod 的网络模型类似于图 3.32。

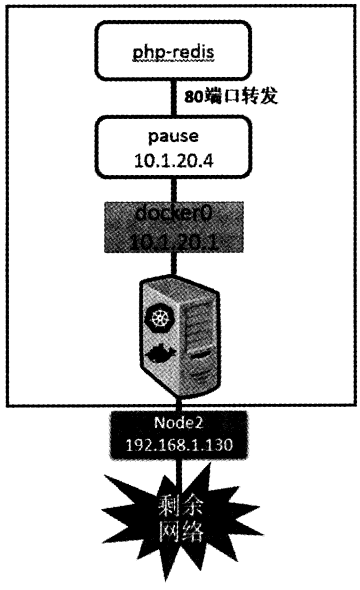

图 3.32　启动 Pod 后网络模型

在这种情况下，实际 Pod 的 IP 数据流的网络目标都是这个 google_containers/pause 容器。图 3.32 有点儿取巧地显示了是 google_containers/pause 容器将端口 80 的流量转发给了相关的容器。而 Pause 只是逻辑上的，并没有真的这么做。实际上另外的 Web 容器直接监听了这些端口，和 google_containers/pause 容器共享了同一个网络堆栈。这就是为什么 Pod 内部实际容器的端口映射都显示到 google_containers/pause 容器上了。我们可以通过 docker port 命令来检验一下：

```
# docker ps
CONTAINER ID        IMAGE
37b193a4c633        kubeguide/example-guestbook-php-redis
6d1b99cff4ae        google_containers/pause:latest
#
# docker port 6d1b99cff4ae
```

```
80/tcp -> 0.0.0.0:80
```

综上所述，google_containers/pause 容器实际上只是负责接管这个 Pod 的 Endpoint，它实际上并没有做更多的事情。那么 Node 呢，它需要将数据流传给 google_containers/pause 容器吗？我们来检查一下 Iptables 的规则，看看有什么发现：

```
# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24 17:15:01 2015
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
:KUBE-NODEPORT-CONTAINER - [0:0]
:KUBE-NODEPORT-HOST - [0:0]
:KUBE-PORTALS-CONTAINER - [0:0]
:KUBE-PORTALS-HOST - [0:0]
-A PREROUTING -m comment --comment "handle ClusterIPs; NOTE: this must be before
the NodePort rules" -j KUBE-PORTALS-CONTAINER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -m comment --comment "handle service
NodePorts; NOTE: this must be the last rule in the chain" -j KUBE-NODEPORT-CONTAINER
-A OUTPUT -m comment --comment "handle ClusterIPs; NOTE: this must be before the
NodePort rules" -j KUBE-PORTALS-HOST
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT -m addrtype --dst-type LOCAL -m comment --comment "handle service
NodePorts; NOTE: this must be the last rule in the chain
-A POSTROUTING -s 10.1.20.0/24 ! -o docker0 -j MASQUERADE
-A KUBE-PORTALS-CONTAINER -d 20.1.0.1/32 -p tcp -m comment --comment
"default/kubernetes:" -m tcp --dport 443 -j REDIRECT --to-ports 60339
-A KUBE-PORTALS-HOST -d 20.1.0.1/32 -p tcp -m comment --comment
"default/kubernetes:" -m tcp --dport 443 -j DNAT --to-destination 192.168.1.131:60339
COMMIT
# Completed on Thu Sep 24 17:15:01 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24 17:15:01 2015
*filter
:INPUT ACCEPT [1131:377745]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [1246:209888]
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER -d 172.17.0.19/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 5000
-j ACCEPT
```

```
COMMIT
# Completed on Thu Sep 24 17:15:01 2015
```

上面的这些规则并没有应用到我们刚刚定义的 Pod。当然，Kubernetes 会给每一个 Kubernetes 的节点提供一些默认的服务，上面的规则就是 Kubernetes 的默认服务需要的。关键是，我们没有看到任何 IP 伪装的规则，并且没有任何指向 Pod 10.1.20.4 的内部方向的端口映射。

第 2 步：发布一个服务

我们已经了解了 Kubernetes 如何处理最基本的元素 Pod 的连接问题，接下来看一下它是如何处理 Service 的。Service 允许我们在多个 Pod 之间抽象一些服务，而且，服务可以通过提供在同一个 Service 的多个 Pod 之间的负载均衡机制来支持水平扩展。我们再次将环境初始化，删除刚刚创建的 RC/Pod 来确保集群是空的：

```
# kubectl stop rc frontend
replicationcontroller/frontend
#
# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS
#
# kubectl get services
NAME        LABELS                                    SELECTOR   IP(S)      PORT(S)
kubernetes   component=apiserver,provider=kubernetes    <none>     20.1.0.1
443/TCP
#
# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

然后准备一个名称为 frontend 的 Service 配置文件：

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  ports:
  - port: 80
#    nodePort: 30001
  selector:
    name: frontend
#  type:
#    NodePort
```

然后在 Kubernetes 集群中定义这个服务：

```
# kubectl create -f frontend-service.yaml
services/frontend
# kubectl get services
NAME       LABELS                SELECTOR              IP(S)          PORT(S)
frontend   name=frontend         name=frontend         20.1.244.75    80/TCP
kubernetes    component=apiserver,provider=kubernetes   <none>    20.1.0.1
443/TCP
```

服务正确创建后，可以看到 Kubernetes 集群已经为这个服务分配了一个虚拟 IP 地址 20.1.244.75，这个 IP 地址是在 Kubernetes 的 Portal Network 中分配的。而这个 Portal Network 的地址范围则是我们在 Kubmaster 上启动 API 服务进程时，使用--service-cluster-ip-range=xx 命令行参数指定的：

```
# cat /etc/kubernetes/apiserver
......
# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=20.1.0.0/16"
......
```

这个 IP 段可以是任何段，只要不和 docker0 或者物理网络的子网冲突就可以。选择任意其他网段的原因是这个网段将不会在物理网络和 docker0 网络上进行路由。这个 Portal Network 针对每一个 Node 都有局部的特殊性，实际上它存在的意义是让容器的流量都指向默认网关（也就是 docker0 网桥）。在继续实验前，先登录到 Node1 上看一下我们定义服务后发生了什么变化。首先检查一下 Iptables/Netfilter 的规则：

```
# iptables-save
......
-A KUBE-PORTALS-CONTAINER -d 20.1.244.75/32 -p tcp -m comment --comment "default/
frontend:" -m tcp --dport 80 -j REDIRECT --to-ports 59528
-A KUBE-PORTALS-HOST -d 20.1.244.75/32 -p tcp -m comment --comment "default/
kubernetes:" -m tcp --dport 80 -j DNAT --to-destination 192.168.1.131:59528
......
```

第 1 行是挂在 PREROUTING 链上的端口重定向规则，所有的进流量如果满足 20.1.244.75:80，则都会被重定向到端口 33761。第 2 行是挂在 OUTPUT 链上的目标地址 NAT，做了和上述第 1 行规则类似的工作，但针对的是当前主机生成的外出流量。所有主机生成的流量都需要使用这个 DNAT 规则来处理。简而言之，这两个规则使用了不同的方式做了类似的事情，就是将所有从节点生成的发送给 20.1.244.75:80 的流量重定向到本地的 33761 端口。

到此为止，目标为 Service IP 地址和端口的任何流量都将被重定向到本地的 33761 端口。这个端口连到哪里去了呢？这就到了 kube-proxy 发挥作用的地方了。这个 kube-proxy 服务给每一个新创建的服务关联了一个随机的端口号，并且监听那个特定的端口，为服务创建相关的负载均衡对象。在我们的实验中，随机生成的端口刚好是 33761。通过监控 Node1 上的 Kubernetes-Service 的日志，在创建服务时，我们可以看到下面的记录：

```
    2612 proxier.go:413] Opened iptables from-containers portal for service "default/
frontend:" on TCP 20.1.244.75:80
    2612 proxier.go:424] Opened iptables from-host portal for service "default/
frontend:" on TCP 20.1.244.75:80
```

现在我们知道，所有的流量都被导入 kube-proxy。现在我们需要它完成一些负载均衡的工作。创建 Replication Controller 并观察结果，下面是 Replication Controller 的配置文件：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
      - name: php-redis
        image: kubeguide/example-guestbook-php-redis
        env:
        - name: GET_HOSTS_FROM
          value: env
        ports:
        - containerPort: 80
#         hostPort: 80
```

在集群发布上述配置文件后，等待并观察，确保所有 Pod 都运行起来了：

```
# kubectl create -f frontend-controller.yaml
replicationcontrollers/frontend
#
# kubectl get pods -o wide
NAME             READY    STATUS     RESTARTS   AGE       NODE
frontend-64t8q   1/1      Running    0          5s        192.168.1.130
frontend-dzqve   1/1      Running    0          5s        192.168.1.131
frontend-x5dwy   1/1      Running    0          5s        192.168.1.129
```

现在所有的 Pod 都运行起来了，Service 将会对匹配到标签为“name=frontend”的所有 Pod 进行负载分发。因为 Service 的选择匹配所有的这些 Pod，所以我们的负载均衡将会对这 3 个 Pod 进行分发。现在我们做实验的环境如图 3.33 所示。

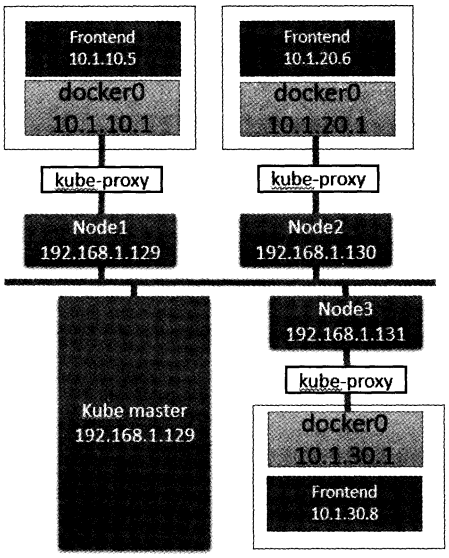

图 3.33　*启动服务后的结构*

Kubernetes 的 kube-proxy 看起来只是一个夹层，但实际上它只是在 Node 上运行的一个服务。上述重定向规则的结果就是针对目标地址为服务 IP 的流量，将 Kubernetes 的 kube-proxy 变成了一个中间的夹层。

为了查看具体的重定向动作，我们会使用 tcpdump 来进行网络抓包操作。首先，安装 tcpdump：

```
yum -y install tcpdump
```

安装完成后，登录 Node1，运行 tcpdump 命令：

```
tcpdump -nn -q -i eno16777736 port 80
```

需要捕获物理服务器以太网接口的数据包，Node1 机器上的以太网接口名字叫作 eno16777736。

再打开第 1 个窗口运行第 2 个 tcpdump 程序，不过我们需要一些额外的信息去运行它，即挂接在 docker0 桥上的虚拟网卡 Veth 的名字。我们看到只有一个 frontend 容器在 Node1 主机上运行，所以可以使用简单的“ip addr”命令来查看唯一的“Veth”网络接口：

```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
```

```
    2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
       link/ether 00:0c:29:47:6e:2c brd ff:ff:ff:ff:ff:ff
       inet 192.168.1.129/24 brd 192.168.1.255 scope global eno16777736
          valid_lft forever preferred_lft forever
       inet6 fe80::20c:29ff:fe47:6e2c/64 scope link
          valid_lft forever preferred_lft forever
    3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
       link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
       inet 10.1.10.1/24 brd 10.1.10.255 scope global docker0
          valid_lft forever preferred_lft forever
       inet6 fe80::5484:7aff:fefe:9799/64 scope link
          valid_lft forever preferred_lft forever
    12: veth0558bfa: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
docker0 state UP
       link/ether 86:82:e5:c8:5a:9a brd ff:ff:ff:ff:ff:ff
       inet6 fe80::8482:e5ff:fec8:5a9a/64 scope link
          valid_lft forever preferred_lft forever
```

复制这个接口的名字，在第 2 个窗口中运行 tcpdump 命令。

```
tcpdump -nn -q -i veth0558bfa host 20.1.244.75
```

同时运行这两个命令，并且将窗口并排放置，以便同时看到两个窗口的输出：

```
# tcpdump -nn -q -i eno16777736 port 80
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eno16777736, link-type EN10MB (Ethernet), capture size 65535 bytes

# tcpdump -nn -q -i veth0558bfa host 20.1.244.75
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0558bfa, link-type EN10MB (Ethernet), capture size 65535 bytes
```

好了，我们已经在同时捕获两个接口的网络包了。这时再启动第 3 个窗口，运行一个“docker exec”命令来连接到我们的“frontend”的容器内部（你可以先执行 docker ps 来获得这个容器的 ID）：

```
# docker ps
CONTAINER ID        IMAGE                                   ......
268ccdfb9524        kubeguide/example-guestbook-php-redis       ......
6a519772b27e        google_containers/pause:latest              ......
```

执行命令进入容器内部：

```
#docker exec -it 268ccdfb9524 bash
# docker exec -it 268ccdfb9524 bash
root@frontend-x5dwy:/#
```

一旦进入运行的容器内部，我们就可以通过 Pod 的 IP 地址来访问服务了。使用 curl 来尝试访问服务：

```
curl 20.1.244.75
```

在使用 curl 访问服务时，将在抓包的两个窗口内看到：

```
20:19:45.208948 IP 192.168.1.129.57452 > 10.1.30.8.8080: tcp 0
20:19:45.209005 IP 10.1.30.8.8080 > 192.168.1.129.57452: tcp 0
20:19:45.209013 IP 192.168.1.129.57452 > 10.1.30.8.8080: tcp 0
20:19:45.209066 IP 10.1.30.8.8080 > 192.168.1.129.57452: tcp 0

20:19:45.209227 IP 10.1.10.5.35225 > 20.1.244.75.80: tcp 0
20:19:45.209234 IP 20.1.244.75.80 > 10.1.10.5.35225: tcp 0
20:19:45.209280 IP 10.1.10.5.35225 > 20.1.244.75.80: tcp 0
20:19:45.209336 IP 20.1.244.75.80 > 10.1.10.5.35225: tcp 0
```

这些信息说明了什么问题呢？让我们在网络图上用实线标出第 1 个窗口中网络抓包信息的含义（物理网卡上的网络流量），并用虚线标出第 2 个窗口中网络抓包信息的含义（docker0 网桥上的网络流量），如图 3.34 所示。

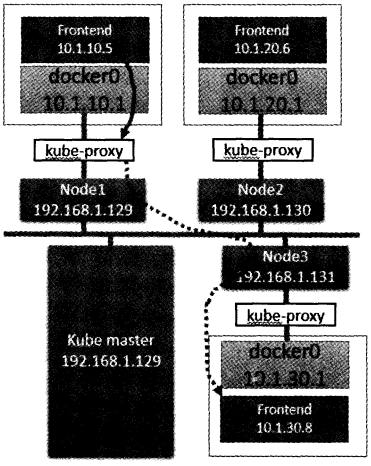

图 3.34　数据流动情况图 1

注意，图 3.34 中，虚线绕过了 Node3 的 kube-proxy，这么做是因为 Node3 上的 kube-proxy 没有参与这次网络交互。换句话说，Node1 的 kube-proxy 服务直接和负载均衡到的 Pod 进行网络交互。

在查看第 2 个捕获包的窗口时，我们能够站在容器的视角看这些流量。首先，容器尝试使用 20.1.244.75:80 打开 TCP 的 Socket 连接。同时，我们还可以看到从服务地址 20.1.244.75 返回的数据。从容器的视角来看，整个交互过程都是在服务之间进行的。但是在查看一个捕获包的窗口时（上面的窗口），我们可以看到物理机之间的数据交互，可以看到一个 TCP 连接从 Node1 的物理地址（192.168.1.129）发出，直接连接到运行 Pod 的主机 Node3（192.168.1.131）。总而言之，Kubernetes 的 kube-proxy 作为一个全功能的代理服务器管理了两个独立的 TCP 连接：一

个是从容器到 kube-proxy；另一个是从 kube-proxy 到负载均衡的目标 Pod。

如果我们清理一下捕获的记录，再次运行 curl，则还可以看到网络流量被负载均衡转发到另一个节点 Node2 上了。

```
20:19:45.208948 IP 192.168.1.129.57485 > 10.1.20.6.8080: tcp 0
20:19:45.209005 IP 10.1.20.6.8080 > 192.168.1.129.57485: tcp 0
20:19:45.209013 IP 192.168.1.129.57485 > 10.1.20.6.8080: tcp 0
20:19:45.209066 IP 10.1.20.6.8080 > 192.168.1.129.57485: tcp 0

20:19:45.209227 IP 10.1.10.5.38026 > 20.1.244.75.80: tcp 0
20:19:45.209234 IP 20.1.244.75.80 > 10.1.10.5.38026: tcp 0
20:19:45.209280 IP 10.1.10.5.38026> 20.1.244.75.80: tcp 0
20:19:45.209336 IP 20.1.244.75.80 > 10.1.10.5.38026: tcp 0
```

这一次，Kubernetes 的 Proxy 将选择运行在 Node2（10.1.20.1）上面的 Pod 作为负载均衡的目的。网络流动图如图 3.35 所示。

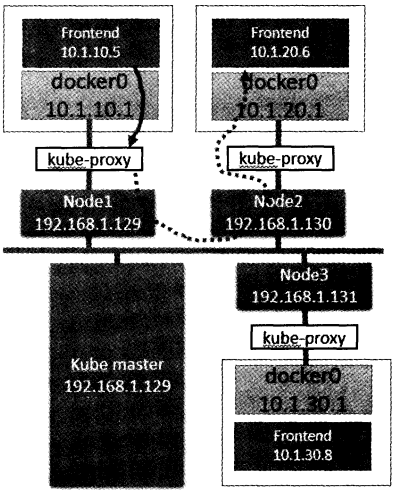

图 3.35　数据流动情况图 2

到这里，你肯定已经知道另外一个可能的负载均衡的路由结果了吧。