表 2.7　对 kubelet 启动参数的详细说明

参数名和取值示例	说　明
--address=0.0.0.0	绑定主机 IP 地址，默认为 0.0.0.0 表示使用全部网络接口
--allow-privileged[=false]	是否允许以特权模式启动容器，默认为 false
--api-servers=[]	API Server 地址列表，以 ip:port 格式表示，以逗号分隔
--application-metrics-count-limit=100	为每个容器保存的性能指标的最大数量，默认为 100
--boot-id-file="/proc/sys/kernel/random/boot_id"	以逗号分隔的文件列表，使用第 1 个存在 book-id 的文件
--cadvisor-port=4194	本地 cAdvisor 监听的端口号，默认为 4194
--cert-dir="/var/run/kubernetes"	TLS 证书所在的目录，默认为/var/run/kubernetes。如果设置了--tls-cert-file 和--tls-private-key-file，则该设置将被忽略
--cgroup-root=""	为 pods 设置的 root cgroup，如果不设置，则将使用容器运行时的默认设置
--chaos-chance=0	随机产生客户端错误的概率，仅用于测试，默认为 0，即不产生
--cloud-config=""	云服务商的配置文件路径
--cloud-provider="auto-detect"	云服务商的名称，默认将自动检测，设置为空表示无云服务商
--cluster-dns=""	集群内 DNS 服务的 IP 地址
--cluster-domain=""	集群内 DNS 服务所用域名
--config=""	kubelet 配置文件的路径或目录名
--configure-cbr0[=false]	设置为 true 表示 kubelet 将会根据 Node.Spec.PodCIDR 的值来配置 cbr0
--container-hints="/etc/cadvisor/container_hints.json"	容器 hints 文件所在的全路径
--container-runtime="docker"	容器类型，目前支持 Docker、rkt，默认为 docker
--containerized[=false]	将 kubelet 运行在容器中，仅供测试使用，默认为 false
--cpu-cfs-quota[=true]	设置为 true 表示启用 CPU CFS quota，用于设置容器的 CPU 限制
--docker-endpoint="unix:///var/run/docker.sock"	Docker 服务的 Endpoint 地址，默认为 unix:///var/run/docker.sock
--docker-env-metadata-whitelist=""	Docker 容器需要使用的环境变量 key 列表，以逗号分隔
--docker-exec-handler="native"	进入 Docker 容器中执行命令的方式，支持 native、nsenter，默认为 native
--docker-only[=false]	设置为 true，表示仅报告 Docker 容器的统计信息而不再报告其他统计信息
--docker-root="/var/lib/docker"	Docker 根目录的全路径，不再使用，将通过 docker info 获取该信息
--enable-controller-attach-detach[=true]	设置为 true 表示启用 Attach/Detach Controller 进行调度到该 Node 的 volume 的 attach 与 detach 操作，同时禁用 kubelet 执行 attach、detach 的操作
--enable-custom-metrics[=false]	设置为 true 表示启用采集自定义性能指标
--enable-debugging-handlers=false	设置为 true 表示提供远程访问本节点容器的日志、进入容器执行命令等相关 Rest 服务
--enable-load-reader[=false]	设置为 true 表示启用 CPU 负载的 reader
--enable-server[=true]	启动 kubelet 上的 http rest server，此 server 提供了获取本节点上运行的 Pod 列表、Pod 状态和其他管理监控相关的 Rest 接口

续表

参数名和取值示例	说　明
--event-burst=10	临时允许的 Event 记录突发的最大数量，默认为 10，当 event-qps>0 时生效
--event-qps=5	设置大于 0 的值表示限制每秒能创建出的 Event 数量，设置为 0 表示不限制
--event-storage-age-limit="default=0"	保存 Event 的最大时间。按事件类型以 key=value 的格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default”表示所有事件的类型
--event-storage-event-limit="default=0"	保存 Event 的最大数量。按事件类型以 key=value 格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default”表示所有事件的类型
--eviction-hard=""	触发 Pod Eviction 操作的一组硬门限设置，例如可用内存<1Gi
--eviction-max-pod-grace-period=0	终止 Pod 操作给 Pod 自行停止预留的时间，单位为秒。时间到达时，将触发 Pod Eviction 操作。默认值为 0，设置为负数表示使用 Pod 中指定的值
--eviction-pressure-transition-period=5m0s	kubelet 在触发 Pod Eviction 操作之前等待的最长时间，默认为 5 分钟
--eviction-soft=""	触发 Pod Eviction 操作的一组软门限设置，例如可用内存<1.5Gi，与 grace-period 一起生效，当 Pod 的响应时间超过 grace-period 后进行触发
--eviction-soft-grace-period=""	触发 Pod Eviction 操作的一组软门限等待时间设置，例如 memory.available=1m30s
--exit-on-lock-contention[=false]	设置为 true 表示当有文件锁存在时 kubelet 也可以退出
--experimental-flannel-overlay[=false]	实验性功能，用于 kubelet 启动时自动支持 flannel 覆盖网络，默认值为 false
--experimental-nvidia-gpus=0	本节点上 NVIDIA GPU 的数量，目前仅支持 0 或 1，默认为 0
--file-check-frequency=20s	在 File Source 作为 Pod 源的情况下，kubelet 定期重新检查文件变化的时间间隔，文件发生变化后，kubelet 重新加载更新的文件内容
--global-housekeeping-interval=1m0s	全局 housekeeping 的时间间隔，默认为 1 分钟
--google-json-key=""	用于谷歌的云平台 Service Account 进行用于鉴权的 JSON key
--hairpin-mode="promiscuous-bridge"	设置 hairpin 模式，表示 kubelet 设置 hairpin NAT 的方式。该模式允许后端 Endpoint 在访问其本身 Service 时能够再次 loadbalance 回自身。可选项包括 promiscuous-bridge、hairpin-veth 和 none
--healthz-bind-address=127.0.0.1	healthz 服务监听的 IP 地址，默认为 127.0.0.1，设置为 0.0.0.0 表示监听全部 IP 地址
--healthz-port=10248	本地 healthz 服务监听的端口号，默认为 10248
--host-ipc-sources="*"	kubelet 允许 Pod 使用宿主机 ipc namespace 的列表，以逗号分隔，默认为“*”
--host-network-sources="*"	kubelet 允许 Pod 使用宿主机 network 的列表，以逗号分隔，默认为“*”
--host-pid-sources="*"	kubelet 允许 Pod 使用宿主机 pid namespace 的列表，以逗号分隔，默认为“*”
--hostname-override=""	设置本 Node 在集群中的主机名，不设置将使用本机 hostanme
--housekeeping-interval=10s	对容器做 housekeeping 操作的时间间隔，默认为 10 秒
--http-check-frequency=20s	HTTP URL Source 作为 Pod 源的情况下，kubelet 定期检查 URL 返回的内容是否发生变化的时间周期，作用同 file-check-frequency 参数
--image-gc-high-threshold=90	镜像垃圾回收上限，磁盘使用空间达到该百分比时，镜像垃圾回收将持续工作
--image-gc-low-threshold=80	镜像垃圾回收下限，磁盘使用空间在达到该百分比之前，镜像垃圾回收将不启用
--kube-api-burst=10	发送到 API Server 的每秒请求数量，默认为 10

续表

参数名和取值示例	说　明
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=5	与 API Server 通信的 QPS 值，默认为 5
--kube-reserved=	kubernetes 系统预留的资源配置，以一组 ResourceName=ResourceQuantity 格式表示，例如 cpu=200m,memory=150G。目前仅支持 CPU 和内存的设置，详见 http://releases.k8s.io/HEAD/docs/user-guide/compute-resources.md，默认为空
--kubeconfig="/var/lib/kubelet/kubeconfig"	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--kubelet-cgroups=""	kubelet 运行所需的 cgroups 名称
--lock-file=""	kubelet 使用的 lock 文件，Alpha 版本
--log-cadvisor-usage[=false]	设置为 true 表示将 cAdvisor 容器的使用情况进行日志记录
--low-diskspace-threshold-mb=256	本 Node 最低磁盘可用空间，单位 MB。当磁盘空间低于该阈值，kubelet 将拒绝创建新的 Pod，默认值为 256MB
--machine-id-file="/etc/machine-id,/var/lib/dbus/machine-id"	用于查找 machine-id 的文件列表，使用找到的第 1 个值，默认从/etc/machine-id,/var/lib/dbus/machine-id 文件中去查找
--manifest-url=""	为 HTTP URL Source 源类型时，kubelet 用来获取 Pod 定义的 URL 地址，此 URL 返回一组 Pod 定义
--manifest-url-header=""	访问 menifest URL 地址时使用的 HTTP 头信息，以 key:value 格式表示
--master-service-namespace="default"	Master 服务的命名空间，默认为 default
--max-open-files=1000000	kubelet 打开的最大文件数量，默认为 1000 000
--max-pods=110	kubelet 能运行的最大 Pod 数量，默认为 110 个 Pod
--maximum-dead-containers=240	在本 Node 上保留的已停止容器的最大数量，由于停止的容器也会消耗磁盘空间，所以超过该上限以后，kubelet 会自动清理已停止的容器以释放磁盘空间，默认为 240
--maximum-dead-containers-per-container=2	以 Pod 为单位可以保留的已停止的（属于同一 Pod 的）容器集的最大数量
--minimum-container-ttl-duration=1m0s	已停止的容器在被清理之前的最小存活时间，例如 300ms、10s 或 2h45m，超过此存活时间的容器将被标记为可被 GC 清理，默认值为 1 分钟
--minimum-image-ttl-duration=2m0s	不再使用的镜像在被清理之前的最小存活时间，例如 300ms、10s 或 2h45m，超过此存活时间的镜像被标记为可被 GC 清理，默认值为两分钟
--network-plugin=""	自定义的网络插件的名字，Pod 的生命周期中相关的一些事件会调用此网络插件进行处理，为 Alpha 测试版功能
--network-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/net/exec/"	扫描网络插件的目录，为 Alpha 测试版功能
--node-ip=""	设置本 Node 的 IP 地址
--node-labels=	kubelet 注册本 Node 时设置的 Labels，label 以 key=value 的格式表示，多个 label 以逗号分隔，为 Alpha 测试版功能
--node-status-update-frequency=10s	kubelet 向 Master 汇报 Node 状态的时间间隔，默认值为 10 秒。与 controller-manager 的—node-monitor-grace-period 参数共同起作用
--non-masquerade-cidr="10.0.0.0/8"	kubelet 向该 IP 段之外的 IP 地址发送的流量将使用 *IP Masquerade* 技术

续表

参数名和取值示例	说　明
--oom-score-adj=-999	kubelet 进程的 oom_score_adj 参数值，有效范围为[-1000, 1000]
--outofdisk-transition-frequency=5m0s	触发磁盘空间耗尽操作之前的等待时间，默认为 5 分钟
--pod-cidr=""	用于给 Pod 分配 IP 地址的 CIDR 地址池，仅在单机模式中使用。在一个集群中，kubelet 会从 API Server 中获取 CIDR 设置
--pod-infra-container-image="gcr.io/google_containers/pause-amd64:3.0"	用于 Pod 内网络命名空间共享的基础 pause 镜像
--pods-per-core=0	该 kubelet 上每个 core 可运行的 Pod 数量。最大值将被 max-pods 参数限制。默认值为 0 表示不做限制
--port=10250	kubelet 服务监听的本机端口号，默认为 10250
--read-only-port=10255	kubelet 服务监听的“只读”端口号，默认为 10255，设置为 0 表示不启用
--really-crash-for-testing=false	设置为 true 表示发生 panics 情况时崩溃，仅用于测试
--reconcile-cidr[=true]	根据 API Server 指定的 CIDR 重排 Node 的 CIDR 地址，如果 register-node 或 configure-cbr0 设置为 false，则表示不启用。默认值为 true
--register-node[=true]	将本 Node 注册到 API Server，默认值为 true
--register-schedulable[=true]	将本 Node 状态标记为 schedulable，设置为 false 表示通知 Master 本 Node 不可进行调度。默认值为 true
--registry-burst=10	最多同时拉取镜像的数量，默认值为 10
--registry-qps=5	在 Pod 创建过程中容器的镜像可能需要从 Registry 中拉取，由于拉取镜像的过程中会消耗大量带宽，因此可能需要限速，此参数与 registry-burst 一起用来限制每秒拉取多少个镜像，默认不限速，如果设置为 5，则表示平均每秒允许拉取 5 个镜像
--resolv-conf="/etc/resolv.conf"	命名服务配置文件，用于容器内应用的 DNS 解析，默认为/etc/resolv.conf
--rkt-api-endpoint="localhost:15441"	rkt API 服务的 URL 地址，--container-runtime='rkt'时生效
--rkt-path=""	rkt 二进制文件的路径，不指定的话从环境变量$PATH 中查找，--container-runtime='rkt'时生效
--root-dir="/var/lib/kubelet"	kubelet 运行根目录，将保持 Pod 和 volume 的相关文件，默认为/var/lib/kubelet。
--runonce=false	设置为 true 表示创建完 Pod 之后立即退出 kubelet 进程，与--api-servers 和 --enable-server 参数互斥
--runtime-cgroups=""	为容器 runtime 设置的 cgroup
--runtime-request-timeout=2m0s	除了长时间运行的 request，对其他 request 的超时时间设置，包括 pull、logs、exec、attach 等操作。当超时时间到达时，请求会被杀掉，抛出一个错误并会重试。默认值为两分钟
--seccomp-profile-root="/var/lib/kubelet/seccomp"	seccomp 配置文件目录，默认为/var/lib/kubelet/seccomp
--serialize-image-pulls[=true]	按顺序挨个 pull 镜像。建议 Docker 低于<1.9 版本或使用 Aufs storage backend 时设置为 true，详见 issue #10959
--storage-driver-buffer-duration=1m0s	将缓存数据写入后端存储的时间间隔，默认为 1 分钟

续表

参数名和取值示例	说　明
--storage-driver-db="cadvisor"	后端存储的数据库名称，默认为 cadvisor
--storage-driver-host="localhost:8086"	后端存储的数据库连接 URL 地址，默认为 localhost:8086
--storage-driver-password="root"	后端存储的数据库密码，默认为 root
--storage-driver-secure[=false]	后端存储的数据库是否用安全连接，默认为 false
--storage-driver-table="stats"	后端存储的数据库表名，默认为 stats
--storage-driver-user="root"	后端存储的数据库用户名，默认为 root
--streaming-connection-idle-timeout=4h0m0s	在容器中执行命令或者进行端口转发的过程中会产生输入、输出流，这个参数用来控制连接空闲超时而关闭的时间，如果设置为“5m”，则表示连接超过 5 分钟没有输入、输出的情况下就被认为是空闲的，而会被自动关闭。默认为 4 小时
--sync-frequency=1m0s	同步运行中容器的配置的频率，默认为 1 分钟
--system-cgroups=""	kubelet 为运行非 kernel 进程设置的 cgroups 名称
--system-reserved=	系统预留的资源配置，以一组 ResourceName=ResourceQuantity 格式表示，例如 cpu=200m,memory=150G。目前仅支持 CPU 和内存的设置，详见 http://releases.k8s.io/HEAD/docs/user-guide/compute-resources.md，默认为空
--tls-cert-file=""	包含 x509 证书的文件路径，用于 HTTPS 认证
--tls-private-key-file=""	包含 x509 与 tls-cert-file 对应的私钥文件路径
--volume-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/volume/exec/"	搜索第三方 volume 插件的目录，为 Alpha 测试版功能
--volume-stats-agg-period=1m0s	kubelet 计算所有 Pod 和 volume 的磁盘使用情况聚合值的时间间隔，默认为 1 分钟。设置为 0 表示不启用该计算功能

6. kube-proxy 启动参数

kube-proxy 的启动参数详细说明见表 2.8。

表 2.8　kube-proxy 的参数表

参数名和取值示例	说　明
--bind-address=0.0.0.0	kube-proxy 绑定主机的 IP 地址，默认为 0.0.0.0 表示绑定所有 IP 地址
--cleanup-iptables[=false]	设置为 true 表示清除 iptables 规则后退出
--cluster-cidr=""	集群中 Pod 的 CIDR 地址范围，用于桥接集群外部流量到内部。用于公有云环境
--config-sync-period=15m0s	从 API Server 更新配置的时间间隔，默认为 15 分钟，必须大于 0
--conntrack-max=0	跟踪 NAT 连接的最大数量，默认值为 0 表示不限制
--conntrack-max-per-core=32768	跟踪每个 CPU core 的 NAT 连接的最大数量，默认值为 32768，仅当 conntrack-max 设置为 0 时生效
--conntrack-tcp-timeout-established=24h0m0s	建立 TCP 连接的超时时间，默认为 24 小时，设置为 0 表示无限制

续表

参数名和取值示例	说　明
--healthz-bind-address=127.0.0.1	healthz 服务绑定主机 IP 地址，默认为 127.0.0.1，设置为 0.0.0.0 表示使用所有 IP 地址
--healthz-port=10249	healthz 服务监听的主机端口号，默认为 10249
--hostname-override=""	设置本 Node 在集群中的主机名，不设置将使用本机 hostanme
--iptables-masquerade-bit=14	iptables masquerade 的 fwmark 位设置，有效范围为[0, 31]
--iptables-sync-period=30s	刷新 iptables 规则的时间间隔，例如 5s、1m、2h22m，默认为 30 秒，必须大于 0
--kube-api-burst=10	发送到 API Server 的每秒发请求数量，默认为 10
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=5	与 API Server 通信的 QPS 值，默认为 5
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--masquerade-all[=false]	设置为 true 表示使用纯 iptables 代理，所有网络包都将做 SNAT 转换
--master=""	API Server 的地址
--oom-score-adj=-999	kube-proxy 进程的 oom_score_adj 参数值，有效范围为[-1000,1000]
--proxy-mode=	代理模式，可选项为 iptables 或 userspace，默认为 iptables，转发速度更快。当操作系统 kernel 版本或 iptables 版本不够新时，将自动降级为 userspace 模式
--proxy-port-range=	进行 Service 代理的本地端口号范围，格式为 begin-end，含两端，未指定则采用随机选择的系统可用的端口号
--udp-timeout=250ms	保持空闲 UDP 连接的时间，例如 250ms、2s，默认值为 250ms，必须大于 0，仅当 proxy-mode=userspace 时生效

2.1.7　Kubernetes 集群网络配置方案

在多个 Node 组成的 Kubernetes 集群内，跨主机的容器间网络互通是 Kubernetes 集群能够正常工作的前提条件。Kubernetes 本身并不会对跨主机容器网络进行设置，这需要额外的工具来实现。除了谷歌公有云 GCE 平台提供的网络设置，一些开源的工具包括 flannel、Open vSwitch、Weave、Calico 等都能够实现跨主机的容器间网络互通。本节将对常用的 flannel、Open vSwitch 和直接路由三种配置进行详细说明。

1. flannel（覆盖网络）

flannel 采用覆盖网络（Overlay Network）模型来完成对网络的打通，本节对 flannel 的安装和配置进行详细说明。

1）安装 etcd

由于 flannel 使用 etcd 作为数据库，所以需要预先安装好 etcd，详见 2.1.2 节的说明。

2）安装 flannel

需要在每台 Node 上都安装 flannel。flannel 软件的下载地址为 https://github.com/coreos/flannel/releases。将下载的压缩包 flannel-<version>-linux-amd64.tar.gz 解压，把二进制文件 flanneld 和 mk-docker-opts.sh 复制到/usr/bin（或其他 PATH 环境变量中的目录），即可完成对 flannel 的安装。

3）配置 flannel

此处以使用 systemd 系统为例对 flanneld 服务进行配置。编辑服务配置文件/usr/lib/systemd/system/flanneld.service：

```
[Unit]
Description=flanneld overlay address etcd agent
After=network.target
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
ExecStart=/usr/bin/flanneld -etcd-endpoints=${FLANNEL_ETCD} $FLANNEL_OPTIONS

[Install]
RequiredBy=docker.service
WantedBy=multi-user.target
```

编辑配置文件/etc/sysconfig/flannel，设置 etcd 的 URL 地址：

```
# flanneld configuration options

# etcd url location.  Point this to the server where etcd runs
FLANNEL_ETCD="http://192.168.18.3:2379"

# etcd config key.  This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"
```

在启动 flanneld 服务之前，需要在 etcd 中添加一条网络配置记录，这个配置将用于 flanneld 分配给每个 Docker 的虚拟 IP 地址段。

```
# etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16" }'
```

4）由于 flannel 将覆盖 docker0 网桥，所以如果 Docker 服务已启动，则停止 Docker 服务。

5）启动 flanneld 服务：

```
# systemctl restart flanneld
```

6）设置 docker0 网桥的 IP 地址：

```
# mk-docker-opts.sh -i
# source /run/flannel/subnet.env
# ifconfig docker0 ${FLANNEL_SUBNET}
```

完成后确认网络接口 docker0 的 IP 地址属于 flannel0 的子网：

```
# ip addr
flannel0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1472
      inet 10.1.10.0  netmask 255.255.0.0  destination 10.1.10.0
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.1.10.1  netmask 255.255.255.0  broadcast 10.1.10.255
```

7）重新启动 Docker 服务：

```
# systemctl restart docker
```

到此就完成了 flannel 覆盖网络的设置。

使用 ping 命令验证各 Node 上 docker0 之间的相互访问。例如在 Node1（docker0 IP=10.1.10.1）机器上 ping Node2 的 docker0（docker0's IP=10.1.30.1），通过 flannel 能够成功连接到其他物理机的 Docker 网络：

```
$ ping 10.1.30.1
PING 10.1.30.1 (10.1.30.1) 56(84) bytes of data.
64 bytes from 10.1.30.1: icmp_seq=1 ttl=62 time=1.15 ms
64 bytes from 10.1.30.1: icmp_seq=2 ttl=62 time=1.16 ms
64 bytes from 10.1.30.1: icmp_seq=3 ttl=62 time=1.57 ms
```

我们也可以在 etcd 中查看到 flannel 设置的 flannel0 地址与物理机 IP 地址的对应规则：

```
# etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/10.1.10.0-24
/coreos.com/network/subnets/10.1.20.0-24
/coreos.com/network/subnets/10.1.30.0-24

# etcdctl get /coreos.com/network/subnets/10.1.10.0-24
{"PublicIP": "192.168.1.129"}
# etcdctl get /coreos.com/network/subnets/10.1.20.0-24
{"PublicIP": "192.168.1.130"}
# etcdctl get /coreos.com/network/subnets/10.1.30.0-24
{"PublicIP": "192.168.1.131"}
```

2. Open vSwitch（虚拟交换机）

以两个 Node 为例，目标网络拓扑如图 2.2 所示。

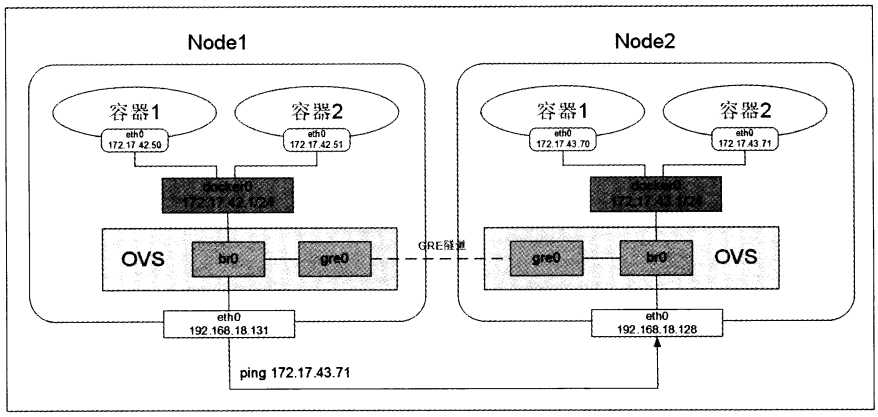

图 2.2　通过 Open vSwitch 打通网络

首先，确保节点 192.168.18.128 的 Docker0 采用 172.17.43.0/24 网段，而 192.168.18.131 的 Docker0 采用 172.17.42.0/24 网段，对应参数为 docker daemon 的启动参数--bip 设置的值。

Open vSwitch 的安装和配置方法如下。

1）在两个 Node 上安装 ovs

```
# yum install openvswitch-2.4.0-1.x86_64.rpm
```

禁止 selinux，配置后重启 Linux：

```
# vi /etc/selinux/config
SELINUX=disabled
```

查看 Open vSwitch 的服务状态，应该启动两个进程：ovsdb-server 与 ovs-vswitchd。

```
# service openvswitch status
ovsdb-server is running with pid 2429
ovs-vswitchd is running with pid 2439
```

查看 Open vSwitch 的相关日志，确认没有异常：

```
# more /var/log/messages |grep openv
Nov  2 03:12:52 docker128 openvswitch: Starting ovsdb-server [  OK  ]
Nov  2 03:12:52 docker128 openvswitch: Configuring Open vSwitch system IDs
[  OK  ]
Nov  2 03:12:52 docker128 kernel: openvswitch: Open vSwitch switching datapath
Nov  2 03:12:52 docker128 openvswitch: Inserting openvswitch module [  OK  ]
```

注意上述操作需要在两个节点机器上分别执行完成。

2）创建网桥和 GRE 隧道

接下来需要在每个 Node 上建立 ovs 的网桥 br0，然后在网桥上创建一个 GRE 隧道连接对端网桥，最后把 ovs 的网桥 br0 作为一个端口连接到 docker0 这个 Linux 网桥上（可以认为是交

换机互联），这样一来，两个节点机器上的 docker0 网段就能互通了。

下面以节点机器 192.168.18.131 为例，具体的操作步骤如下。

（1）创建 ovs 网桥：

```
# ovs-vsctl add-br br0
```

（2）创建 GRE 隧道连接对端，remote_ip 为对端 eth0 的网卡地址：

```
# ovs-vsctl add-port br0 gre1 -- set interface gre1 type=gre
option:remote_ip=192.168.18.128
```

（3）添加 br0 到本地 docker0，使得容器流量通过 OVS 流经 tunnel：

```
# brctl addif docker0 br0
```

（4）启动 br0 与 docker0 网桥：

```
# ip link set dev br0 up
# ip link set dev docker0 up
```

（5）添加路由规则。由于 192.168.18.128 与 192.168.18.131 的 docker0 网段分别为 172.17.43.0/24 与 172.17.42.0/24，这两个网段的路由都需要经过本机的 docker0 网桥路由，其中一个 24 网段是通过 OVS 的 GRE 隧道到达对端的，因此需要在每个 Node 上添加通过 docker0 网桥转发的 172.17.0.0/16 段的路由规则：

```
# ip route add 172.17.0.0/16 dev docker0
```

（6）清空 Docker 自带的 Iptables 规则及 Linux 的规则，后者存在拒绝 icmp 报文通过防火墙的规则：

```
# iptables -t nat -F; iptables -F
```

在 192.168.18.131 上完成上述步骤后，在 192.168.18.128 节点执行同样的操作，注意，GRE 隧道里的 IP 地址要改为对端节点（192.168.18.131）的 IP 地址。

配置完成后，192.168.18.131 的 IP 地址、docker0 的 IP 地址及路由等重要信息显示如下：

```
# 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
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
qlen 1000
    link/ether 00:0c:29:55:5e:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.18.131/24 brd 192.168.18.255 scope global dynamic eth0
       valid_lft 1369sec preferred_lft 1369sec
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether a6:15:c3:25:cf:33 brd ff:ff:ff:ff:ff:ff
4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
```

```
state UNKNOWN
       link/ether 92:8d:d0:a4:ca:45 brd ff:ff:ff:ff:ff:ff
    5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
       link/ether 02:42:44:8d:62:11 brd ff:ff:ff:ff:ff:ff
       inet 172.17.42.1/24 scope global docker0
         valid_lft forever preferred_lft forever
```

同样，192.168.18.128 节点的重要信息如下：

```
    # 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
    2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
qlen 1000
       link/ether 00:0c:29:e8:02:c7 brd ff:ff:ff:ff:ff:ff
       inet 192.168.18.128/24 brd 192.168.18.255 scope global dynamic eth0
         valid_lft 1356sec preferred_lft 1356sec
    3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
       link/ether fa:6c:89:a2:f2:01 brd ff:ff:ff:ff:ff:ff
    4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
state UNKNOWN
       link/ether ba:89:14:e0:7f:43 brd ff:ff:ff:ff:ff:ff
    5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
       link/ether 02:42:63:a8:14:d5 brd ff:ff:ff:ff:ff:ff
       inet 172.17.43.1/24 scope global docker0
         valid_lft forever preferred_lft forever
```

3）两个 Node 上容器之间的互通测试

首先，在 192.168.18.128 节点上 ping 192.168.18.131 上的 docker0 地址：172.17.42.1，验证网络互通性：

```
    # ping 172.17.42.1
    PING 172.17.42.1 (172.17.42.1) 56(84) bytes of data.
    64 bytes from 172.17.42.1: icmp_seq=1 ttl=64 time=1.57 ms
    64 bytes from 172.17.42.1: icmp_seq=2 ttl=64 time=0.966 ms
    64 bytes from 172.17.42.1: icmp_seq=3 ttl=64 time=1.01 ms
    64 bytes from 172.17.42.1: icmp_seq=4 ttl=64 time=1.00 ms
    64 bytes from 172.17.42.1: icmp_seq=5 ttl=64 time=1.22 ms
    64 bytes from 172.17.42.1: icmp_seq=6 ttl=64 time=0.996 ms
```

下面我们通过 tshark 抓包工具来分析流量走向。首先，在 192.168.18.128 节点上监听 br0 上是否有 GRE 报文，执行下面的命令，我们发现 br0 上并没有 GRE 报文：

```
    # tshark -i  br0 -R ip proto GRE
    tshark: -R without -2 is deprecated. For single-pass filtering use -Y.
    Running as user "root" and group "root". This could be dangerous.
```

```
Capturing on 'br0'
^C
```

而在 eth0 上抓包，则发现了 GRE 封装的 ping 包报文通过，说明 GRE 是在承载网的物理网上完成的封包过程：

```
# tshark -i  eth0 -R ip proto GRE
tshark: -R without -2 is deprecated. For single-pass filtering use -Y.
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
  1   0.000000  172.17.43.1 -> 172.17.42.1  ICMP 136 Echo (ping) request
id=0x0970, seq=180/46080, ttl=64
  2   0.000892  172.17.42.1 -> 172.17.43.1  ICMP 136 Echo (ping) reply
id=0x0970, seq=180/46080, ttl=64 (request in 1)
2   3   1.002014  172.17.43.1 -> 172.17.42.1  ICMP 136 Echo (ping) request
id=0x0970, seq=181/46336, ttl=64
  4   1.002916  172.17.42.1 -> 172.17.43.1  ICMP 136 Echo (ping) reply
id=0x0970, seq=181/46336, ttl=64 (request in 3)
4   5   2.004101  172.17.43.1 -> 172.17.42.1  ICMP 136 Echo (ping) request
id=0x0970, seq=182/46592, ttl=64
```

至此，基于 OVS 的网络搭建成功，由于 GRE 是点对点隧道通信方式，所以如果有多个 Node，则需要建立 $N \times (N-1)$ 条 GRE 隧道，即所有 Node 组成一个网状的网络，实现全网互通。

3. 直接路由

通过在每个 Node 上添加到其他 Node 上 docker0 的静态路由规则，就可以将不同物理机的 docker0 网桥互联互通。图 2.3 描述了在两个 Node 之间打通网络的情况。

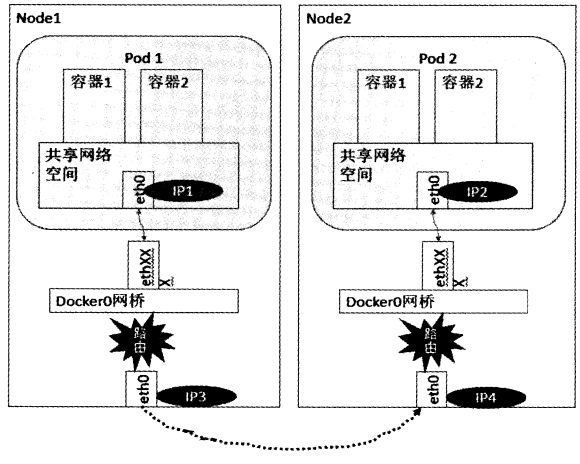

图 2.3　以直接路由方式实现 Pod 到 Pod 的通信

使用这种方案，只需要在每个 Node 的路由表中增加到对方 docker0 的静态路由转发规则。

例如 Pod1 所在 docker0 网桥的 IP 子网是 10.1.10.0，Node 地址为 192.168.1.128；而 Pod2 所在 docker0 网桥的 IP 子网是 10.1.20.0，Node 地址为 192.168.1.129。

在 Node1 上用 route add 命令增加一条到 Node2 上 docker0 的静态路由规则：

```
route add -net 10.1.20.0 netmask 255.255.255.0 gw 192.168.1.129
```

同样，在 Node2 上增加一条到 Node1 上 docker0 的静态路由规则：

```
route add -net 10.1.10.0 netmask 255.255.255.0 gw 192.168.1.128
```

在 Node1 上通过 ping 命令验证到 Node2 上 docker0 的网络连通性。这里 10.1.20.1 为 Node2 上 docker0 网桥自身的 IP 地址。

```
$ ping 10.1.20.1
PING 10.1.20.1 (10.1.20.1) 56(84) bytes of data.
64 bytes from 10.1.20.1: icmp_seq=1 ttl=62 time=1.15 ms
64 bytes from 10.1.20.1: icmp_seq=2 ttl=62 time=1.16 ms
64 bytes from 10.1.20.1: icmp_seq=3 ttl=62 time=1.57 ms
......
```

可以看到，路由转发规则生效，Node1 可以直接访问到 Node2 上的 docker0 网桥，进一步也可以访问到属于 docker0 网段的容器应用了。

不过，集群中机器的数量通常可能很多。假设有 100 台服务器，那么就需要在每台服务器上手工添加到另外 99 台服务器 docker0 的路由规则。为了减少手工操作，可以使用 Quagga 软件来实现路由规则的动态添加。Quagga 软件的主页为 http://www.quagga.net。

除了在每台服务器上安装 Quagga 软件并启动，还可以使用 Quagga 容器来运行（例如 index.alauda.cn/georce/router）。在每台 Node 上下载该 Docker 镜像：

```
$ docker pull index.alauda.cn/georce/router
```

在运行 Quagga 容器之前，需要确保每个 Node 上 docker0 网桥的子网地址不能重叠，也不能与物理机所在的网络重叠，这需要网络管理员的仔细规划。

下面以 3 个 Node 为例，每台 Node 的 docker0 网桥的地址如下（前提是 Node 物理机的 IP 地址不是 10.1.X.X 地址段）：

```
Node 1: # ifconfig docker0 10.1.10.1/24
Node 2: # ifconfig docker0 10.1.20.1/24
Node 3: # ifconfig docker0 10.1.30.1/24
```

然后在每个 Node 上启动 Quagga 容器。需要说明的是，Quagga 需要以--privileged 特权模式运行，并且指定--net=host，表示直接使用物理机的网络：

```
$ docker run -itd --name=router --privileged --net=host
index.alauda.cn/georce/router
```

启动成功后，Quagga 会相互学习来完成到其他机器的 docker0 路由规则的添加。

一段时间后，在 Node1 上使用 route -n 命令来查看路由表，可以看到 Quagga 自动添加了两条到 Node2 和到 Node3 上 docker0 的路由规则。

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.128   0.0.0.0         UG    0      0        0 eth0
10.1.10.0       0.0.0.0         255.255.255.0   U     0      0        0 docker0
10.1.20.0       192.168.1.129   255.255.255.0   UG    20     0        0 eth0
10.1.30.0       192.168.1.130   255.255.255.0   UG    20     0        0 eth0
```

在 Node2 上查看路由表，可以看到自动添加了两条到 Node1 和 Node3 上 docker0 的路由规则。

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.129   0.0.0.0         UG    0      0        0 eth0
10.1.20.0       0.0.0.0         255.255.255.0   U     0      0        0 docker0
10.1.10.0       192.168.1.128   255.255.255.0   UG    20     0        0 eth0
10.1.30.0       192.168.1.130   255.255.255.0   UG    20     0        0 eth0
```

至此，所有 Node 上的 docker0 都可以互联互通了。

2.2 kubectl 命令行工具用法详解

kubectl 作为客户端 CLI 工具，可以让用户通过命令行的方式对 Kubernetes 集群进行操作。本节对 kubectl 的子命令和用法进行详细说明。

2.2.1 kubectl 用法概述

kubectl 命令行的语法如下：

```
$ kubectl [command] [TYPE] [NAME] [flags]
```

其中，command、TYPE、NAME、flags 的含义如下。

（1）command：子命令，用于操作 Kubernetes 集群资源对象的命令，例如 create、delete、describe、get、apply 等。

（2）TYPE：资源对象的类型，区分大小写，能以单数形式、复数形式或者简写形式表示。例如以下 3 种 TYPE 是等价的。

```
$ kubectl get pod pod1
```

```
$ kubectl get pods pod1
$ kubectl get po pod1
```

（3）NAME：资源对象的名称，区分大小写。如果不指定名称，则系统将返回属于 TYPE 的全部对象的列表，例如$ kubectl get pods 将返回所有 Pod 的列表。

（4）flags：kubectl 子命令的可选参数，例如使用“-s”指定 apiserver 的 URL 地址而不用默认值。

kubectl 可操作的资源对象类型如表 2.9 所示。

表 2.9　kubectl 可操作的资源对象类型

资源对象的名称	缩　写
componentstatuses	cs
daemonsets	ds
deployments	
events	ev
endpoints	ep
horizontalpodautoscalers	hpa
ingresses	ing
jobs	
limitranges	limits
nodes	no
namespaces	ns
pods	po
persistentvolumes	pv
persistentvolumeclaims	pvc
resourcequotas	quota
replicationcontrollers	rc
secrets	
serviceaccounts	
services	svc

在一个命令行中也可以同时对多个资源对象进行操作，以多个 TYPE 和 NAME 的组合表示，示例如下。

◎ 获取多个 Pod 的信息：

```
$ kubectl get pods pod1 pod2
```

◎ 获取多种对象的信息：

```
$ kubectl get pod/pod1 rc/rc1
```

◎ 同时应用多个 yaml 文件，以多个-f file 参数表示：

```
$ kubectl get pod -f pod1.yaml -f pod2.yaml
$ kubectl create -f pod1.yaml -f rc1.yaml -f service1.yaml
```

2.2.2 kubectl 子命令详解

kubectl 的子命令非常丰富，涵盖了对 Kubernetes 集群的主要操作，包括资源对象的创建、删除、查看、修改、配置、运行等。详细的子命令如表 2.10 所示。

表 2.10 kubectl 子命令详解

子 命 令	语 法	说 明
annotate	kubectl annotate [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--resource-version=version] [flags]	添加或更新资源对象的 annotation 信息
api-versions	kubectl api-versions [flags]	列出当前系统支持的 API 版本列表，格式为“group/version”
apply	kubectl apply -f FILENAME [flags]	从配置文件或 stdin 中对资源对象进行配置更新
attach	kubectl attach POD -c CONTAINER [flags]	附着到一个正在运行的容器上
autoscale	kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=MINPODS] --max=MAXPODS [--cpu-percent=CPU] [flags]	对 Deployment、ReplicaSet 或 ReplicationController 进行水平自动扩缩容的设置
cluster-info	kubectl cluster-info [flags] kubectl cluster-info [command]	显示集群信息
completion	kubectl completion SHELL [flags]	输出 shell 命令的执行结果码（bash 或 zsh）
config	kubectl config SUBCOMMAND [flags] kubectl config [command]	修改 kubeconfig 文件
convert	kubectl convert –f FILENAME [flags]	转换配置文件为不同的 API 版本
cordon	kubectl cordon NODE [flags]	将 Node 标记为 unschedulable，即“隔离”出集群调度范围
create	kubectl create –f FILENAME [flags] kubectl create [command]	从配置文件或 stdin 中创建资源对象
delete	kubectl delete ([-f FILENAME] | TYPE [(NAME | -l label | --all)]) [flags]	根据配置文件、stdin、资源名称或 label selector 删除资源对象
describe	kubectl describe (-f FILENAME | TYPE [NAME_PREFIX | /NAME | -l label]) [flags]	描述一个或多个资源对象的详细信息
drain	kubectl drain NODE [flags]	首先将 Node 设置为 unschedulable，然后删除该 Node 上运行的所有 Pod，但不会删除不由 apiserver 管理的 Pod
edit	kubectl edit (RESOURCE/NAME | -f FILENAME) [flags]	编辑资源对象的属性，在线更新

续表

子 命 令	语　法	说　明
exec	kubectl exec POD [-c CONTAINER] -- COMMAND [args...] [flags]	执行一个容器中的命令
explain	kubectl explain RESOURCE [flags]	对资源对象属性的详细说明
expose	kubectl expose (-f FILENAME | TYPE NAME) [--port=port] [--protocol=TCP|UDP] [--target-port=number-or-name] [--name=name] [--external-ip=external-ip-of-service] [--type=type] [flags]	将已经存在的一个 RC、Service、Deployment 或 Pod 暴露为一个新的 Service
get	kubectl get [(-o|--output=)json|yaml|wide|go-template=...|go-template-file=...|jsonpath=...|jsonpath-file=...] (TYPE [NAME | -l label] | TYPE/NAME ...) [flags]	显示一个或多个资源对象的概要信息
label	kubectl label [--overwrite] (-f FILENAME | TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--resource-version=version] [flags]	设置或更新资源对象的 labels
logs	kubectl logs [-f] [-p] POD [-c CONTAINER] [flags]	屏幕打印一个容器的日志
namespace	kubectl namespace [namespace] [flags]	已被 kubectl config set-context 替代
patch	kubectl patch (-f FILENAME | TYPE NAME) -p PATCH [flags]	以 merge 形式对资源对象的部分字段的值进行修改
port-forward	kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT [...[LOCAL_PORT_N:]REMOTE_PORT_N] [flags]	将本机的某个端口号映射到 Pod 的端口号，通常用于测试工作
proxy	kubectl proxy [--port=PORT] [--www=static-dir] [--www-prefix=prefix] [--api-prefix=prefix] [flags]	将本机某个端口号映射到 apiserver
replace	kubectl replace -f FILENAME [flags]	从配置文件或 stdin 替换资源对象
rolling-update	kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --image=NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC) [flags]	对 RC 进行滚动升级
rollout	kubectl rollout SUBCOMMAND [flags] kubectl rollout [command]	对 Deployment 进行管理，可用操作包括：history、pause、resume、undo、status
run	kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [--command] -- [COMMAND] [args...] [flags]	基于一个镜在 Kubernetes 集群上启动一个 Deployment
scale	kubectl scale [--resource-version=version] [--current-replicas=count] --replicas=COUNT (-f FILENAME | TYPE NAME) [flags]	扩容、缩容一个 Deployment、ReplicaSet、RC 或 Job 中 Pod 的数量
set	kubectl set SUBCOMMAND [flags] kubectl set [command]	设置资源对象的某个特定信息，目前仅支持修改容器的镜像
taint	kubectl taint NODE NAME KEY_1=VAL_1:TAINT_EFFECT_1 ... KEY_N=VAL_N:TAINT_EFFECT_N [flags]	设置 Node 的 taint 信息，用于将特定的 Pod 调度到特定的 Node 的操作，为 Alpha 版本功能
uncordon	kubectl uncordon NODE [flags]	将 Node 设置为 schedulable
version	kubectl version [flags]	打印系统的版本信息

2.2.3 kubectl 参数列表

kubectl 命令行的公共启动参数如表 2.11 所示。

表 2.11 kubectl 命令行公共参数

参数名和取值示例	说　明
--alsologtostderr[=false]	设置为 true 表示将日志输出到文件的同时输出到 stderr
--as=""	设置本次操作的用户名
--certificate-authority=""	用于 CA 授权的 cert 文件路径
--client-certificate=""	用于 TLS 的客户端证书文件路径
--client-key=""	用于 TLS 的客户端 key 文件路径
--cluster=""	设置要使用的 kubeconfig 中的 cluster 名
--context=""	设置要使用的 kubeconfig 中的 context 名
--insecure-skip-tls-verify[=false]	设置为 true 表示跳过 TLS 安全验证模式，将使得 HTTPS 连接不安全
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--log-backtrace-at=:0	记录日志每到“file:行号”时打印一次 stack trace
--log-dir=""	日志文件路径
--log-flush-frequency=5s	设置 flush 日志文件的时间间隔
--logtostderr[=true]	设置为 true 表示将日志输出到 stderr，不输出到日志文件
--match-server-version[=false]	设置为 true 表示客户端版本号需要与服务端一致
--namespace=""	设置本次操作所在的 namespace
--password=""	设置 apiserver 的 basic authentication 的密码
-s, --server=""	设置 apiserver 的 URL 地址，默认为 localhost:8080
--stderrthreshold=2	在该 threshold 级别之上的日志将输出到 stderr
--token=""	设置访问 apiserver 的安全 token
--user=""	指定 kubeconfig 用户名
--username=""	设置 apiserver 的 basic authentication 的用户名
--v=0	glog 日志级别
--vmodule=	glog 基于模块的详细日志级别

每个子命令（如 create、delete、get 等）还有特定的 flags 参数，可以通过$ kubectl [command] --help 命令进行查看。

2.2.4 kubectl 输出格式

kubectl 命令可以用多种格式对结果进行显示，输出的格式通过-o 参数指定：

```
$ kubectl [command] [TYPE] [NAME] -o=<output_format>
```

根据不同子命令的输出结果，可选的输出格式如表 2.12 所示。

表 2.12　kubectl 命令输出格式列表

输出格式	说　明
-o=custom-columns=<spec>	根据自定义列名进行输出，以逗号分隔
-o=custom-columns-file=<filename>	从文件中获取自定义列名进行输出
-o=json	以 JSON 格式显示结果
-o=jsonpath=<template>	输出 jsonpath 表达式定义的字段信息
-o=jsonpath-file=<filename>	输出 jsonpath 表达式定义的字段信息，来源于文件
-o=name	仅输出资源对象的名称
-o=wide	输出额外信息。对于 Pod，将输出 Pod 所在的 Node 名
-o=yaml	以 yaml 格式显示结果

常用的输出格式示例如下。

（1）显示 Pod 的更多信息：

```
$ kubectl get pod <pod-name> -o wide
```

（2）以 yaml 格式显示 Pod 的详细信息：

```
$ kubectl get pod <pod-name> -o yaml
```

（3）以自定义列名显示 Pod 的信息：

```
$ kubectl get pod <pod-name>
-o=custom-columns=NAME:.metadata.name,RSRC:.metadata.resourceVersion
```

（4）基于文件的自定义列名输出：

```
$ kubectl get pods <pod-name> -o=custom-columns-file=template.txt
```

template.txt 文件的内容为：

```
NAME                RSRC
metadata.name       metadata.resourceVersion
```

输出结果为：

```
NAME          RSRC
pod-name      52305
```

另外，还可以将输出结果按某个字段排序，通过--sort-by 参数以 jsonpath 表达式进行指定：

```
$ kubectl [command] [TYPE] [NAME] --sort-by=<jsonpath_exp>
```

例如，按照名字进行排序：

```
$ kubectl get pods --sort-by=.metadata.name
```

2.2.5 kubectl 操作示例

本节对一些常用的 kubectl 操作进行示例。

1. 创建资源对象

根据 yaml 配置文件一次性创建 service 和 rc：

```
$ kubectl create -f my-service.yaml -f my-rc.yaml
```

根据<directory>目录下所有.yaml、.yml、.json 文件的定义进行创建操作：

```
$ kubectl create -f <directory>
```

2. 查看资源对象

查看所有 Pod 列表：

```
$ kubectl get pods
```

查看 rc 和 service 列表：

```
$ kubectl get rc,service
```

3. 描述资源对象

显示 Node 的详细信息：

```
$ kubectl describe nodes <node-name>
```

显示 Pod 的详细信息：

```
$ kubectl describe pods/<pod-name>
```

显示由 RC 管理的 Pod 的信息：

```
$ kubectl describe pods <rc-name>
```

4. 删除资源对象

基于 pod.yaml 定义的名称删除 Pod：

```
$ kubectl delete -f pod.yaml
```

删除所有包含某个 label 的 Pod 和 service：

```
$ kubectl delete pods,services -l name=<label-name>
```

删除所有 Pod：

```
$ kubectl delete pods --all
```

5. 执行容器的命令

执行 Pod 的 date 命令，默认使用 Pod 中的第 1 个容器执行：

```
$ kubectl exec <pod-name> date
```

指定 Pod 中某个容器执行 date 命令：

```
$ kubectl exec <pod-name> -c <container-name> date
```

通过 bash 获得 Pod 中某个容器的 TTY，相当于登录容器：

```
$ kubectl exec -ti <pod-name> -c <container-name> /bin/bash
```

6. 查看容器的日志

查看容器输出到 stdout 的日志：

```
$ kubectl logs <pod-name>
```

跟踪查看容器的日志，相当于 tail -f 命令的结果：

```
$ kubectl logs -f <pod-name> -c <container-name>
```

2.3　Guestbook 示例：Hello World

在对 Kubernetes 的容器应用进行详细说明之前，让我们先通过一个由 3 个微服务组成的留言板（Guestbook）系统的搭建，对 Kubernetes 对容器应用的基本操作和用法进行初步介绍。本章后面的章节将基于该案例和其他示例，进一步深入 Pod、RC、Service 等核心对象的用法和技巧，对 Kubernetes 的应用管理进行全面讲解。

Guestbook 留言板系统将通过 Pod、RC、Service 等资源对象搭建完成，成功启动后在网页中显示一条“Hello World”留言。其系统架构是一个基于 PHP+Redis 的分布式 Web 应用，前端 PHP Web 网站通过访问后端的 Redis 来完成用户留言的查询和添加等功能。同时 Redis 以 Master+Slave 的模式进行部署，实现数据的读写分离能力。

留言板系统的部署架构如图 2.4 所示。Web 层是一个基于 PHP 页面的 Apache 服务，启动 3 个实例组成集群，为客户端（例如浏览器）对网站的访问提供负载均衡。Redis Master 启动 1 个实例用于写操作（添加留言），Redis Slave 启动两个实例用于读操作（读取留言）。Redis Master 与 Slave 的数据同步由 Redis 具备的数据同步机制完成。

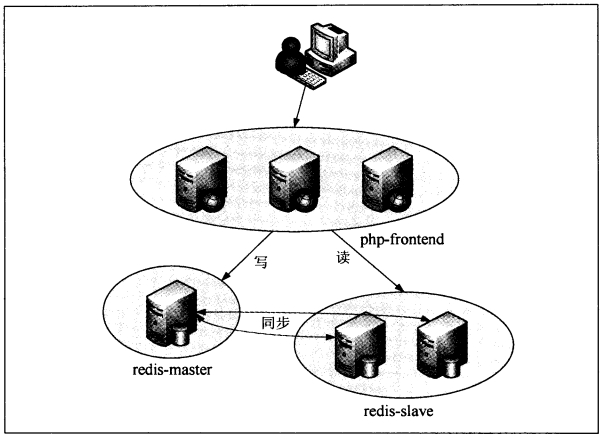

图 2.4　留言板的系统部署架构图

在本例中将要用到 3 个 Docker 镜像，下载地址为 https://hub.docker.com/u/kubeguide/。

- redis-master：用于前端 Web 应用进行“写”留言操作的 Redis 服务，其中已经保存了一条内容为“Hello World!”的留言。
- guestbook-redis-slave：用于前端 Web 应用进行“读”留言操作的 Redis 服务，并与 Redis-Master 的数据保持同步。
- guestbook-php-frontend：PHP Web 服务，在网页上展示留言的内容，也提供一个文本输入框供访客添加留言。

如图 2.5 所示为 Hello World 案例所采用的 Kubernetes 部署架构，这里 Master 与 Node 的服务处于同一个虚拟机中。通过创建 redis-master 服务、redis-slave 服务和 php-frontend 服务来实现整个系统的搭建。

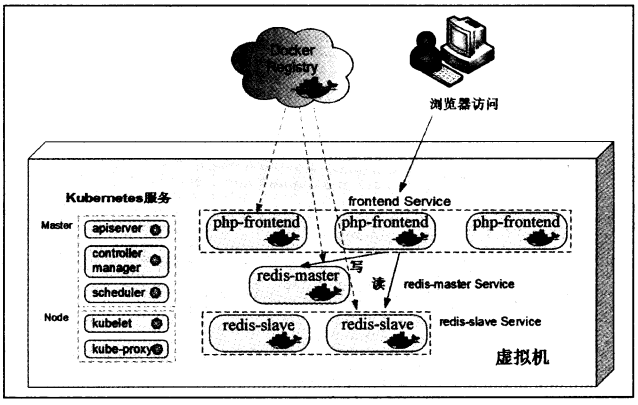

图 2.5　Kubernetes 部署架构图

2.3.1　创建 redis-master RC 和 Service

我们可以先定义 Service，然后定义一个 RC 来创建和控制相关联的 Pod，或者先定义 RC 来创建 Pod，然后定义与之关联的 Service，这里我们采用后一种方法。

首先为 redis-master 创建一个名为 redis-master 的 RC 定义文件 redis-master-controller.yaml。yaml 的语法类似于 PHP 的语法，对于空格的个数有严格的要求，详见 http://yaml.org。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
      containers:
      - name: master
        image: kubeguide/redis-master
        ports:
        - containerPort: 6379
```

其中，kind 字段的值为“ReplicationController”，表示这是一个 RC；spec.selector 是 RC 的 Pod 选择器，即监控和管理拥有这些标签（Label）的 Pod 实例，确保当前集群上始终有且仅有 replicas 个 Pod 实例在运行，这里我们设置 replicas=1 表示只运行一个（名为 redis-master 的）Pod 实例，当集群中运行的 Pod 数量小于 replicas 时，RC 会根据 spec.template 段定义的 Pod 模板来生成一个新的 Pod 实例，labels 属性指定了该 Pod 的标签，注意，这里的 labels 必须匹配 RC 的 spec.selector，否则此 RC 就会陷入“只为他人做嫁衣”的悲惨世界中，永无翻身之时。

创建好 redis-master-controller.yaml 文件以后，我们在 Master 节点执行命令：kubectl create -f <config_file>，将它发布到 Kubernetes 集群中，就完成了 redis-master 的创建过程：

```
$ kubectl create -f redis-master-controller.yaml
replicationcontroller "redis-master" created
```

系统提示“redis-master”表示创建成功。然后我们用 kubectl 命令查看刚刚创建的 redis-master：

```
$ kubectl get rc
```

```
NAME              DESIRED   CURRENT   AGE
redis-master       1         1        5m
```

接下来运行 kubectl get pods 命令来查看当前系统中的 Pod 列表信息，我们看到一个名为 redis-master-xxxxx 的 Pod 实例，这是 Kubernetes 根据 redis-master 这个 RC 的定义自动创建的 Pod。RC 会给每个 Pod 实例在用户设置的 name 后补充一段 UUID，以区分不同的实例。由于 Pod 的调度和创建需要花费一定的时间，比如需要一定的时间来确定调度到哪个节点上，以及下载 Pod 的相关镜像，所以一开始我们看到 Pod 的状态将显示为 Pending。当 Pod 成功创建完成以后，状态会被更新为 Running。如果 Pod 一直处于 Pending 状态，则请参看第 5 章的查错说明。

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

redis-master Pod 已经创建并正常运行了，接下来我们就创建一个与之关联的 Service（服务）定义文件（文件名为 redis-master-service.yaml），完整的内容如下：

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

其中 metadata.name 是 Service 的服务名（ServiceName），spec.selector 确定了选择哪些 Pod，本例中的定义表明将选择设置过 name=redis-master 标签的 Pod。port 属性定义的是 Service 的虚拟端口号，targetPort 属性指定后端 Pod 内容器应用监听的端口号。

运行 kubectl create 命令创建该 service：

```
$ kubectl create -f redis-master-service.yaml
service "redis-master" created
```

系统提示“service "redis-master" created”表示创建成功。然后运行 kubectl get 命令可以查看到刚刚创建的 service：

```
$ kubectl get services
NAME              CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
redis-master      169.169.208.57    <none>       6379/TCP   13m
```

注意到 redis-master 服务被分配了一个值为 169.169.208.57 的虚拟 IP 地址，随后，Kubernetes 集群中其他新创建的 Pod 就可以通过这个虚拟 IP 地址+端口 6379 来访问这个服务了。在本例中将要创建的 redis-slave 和 frontend 两组 Pod 都将通过 169.169.208.57:6379 来访问 redis-master 服务。

但由于 IP 地址是在服务创建后由 Kubernetes 系统自动分配的，在其他 Pod 中无法预先知道某个 Service 的虚拟 IP 地址，因此需要一个机制来找到这个服务。为此，Kubernetes 巧妙地使用了 Linux 环境变量（Environment Variable），在每个 Pod 的容器里都增加了一组 Service 相关的环境变量，用来记录从服务名到虚拟 IP 地址的映射关系。以 redis-master 服务为例，在容器的环境变量中会增加下面两条记录：

```
REDIS_MASTER_SERVICE_HOST=169.169.144.74
REDIS_MASTER_SERVICE_PORT=6379
```

于是，redis-slave 和 frontend 等 Pod 中的应用程序就可以通过环境变量 REDIS_MASTER_SERVICE_HOST 得到 redis-master 服务的虚拟 IP 地址，通过环境变量 REDIS_MASTER_SERVICE_PORT 得到 redis-master 服务的端口号，这样就完成了对服务地址的查询功能。

2.3.2　创建 redis-slave RC 和 Service

现在我们已经成功启动了 redis-master 服务，接下来我们继续完成 redis-slave 服务的创建过程。在本案例中会启动 redis-slave 服务的两个副本，每个副本上的 Redis 进程都与 redis-master 进行数据同步，与 redis-master 共同组成了一个具备读写分离能力的 Redis 集群。留言板的 PHP 网页将通过访问 redis-slave 服务来读取留言数据。与之前的 redis-master 服务的创建过程一样，首先创建一个名为 redis-slave 的 RC 定义文件 redis-slave-controller. yaml，完整内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 2
  selector:
    name: redis-slave
  template:
    metadata:
      labels:
        name: redis-slave
    spec:
      containers:
```