```
            s.podStatusChannel <- podStatusSyncRequest{pod, status}
        } else {
            glog.V(3).Infof("Ignoring same status for pod %q, status: %+v", kubeletUtil.
FormatPodName(pod), status)
        }
    }
```

下面是在 Pod 实例化的过程中，kubelet 过滤掉不合适本节点 Pod 所调用的上述方法的代码，类似的调用还有不少：

```
    func (kl *Kubelet) handleNotFittingPods(pods []*api.Pod) []*api.Pod {
        fitting, notFitting := checkHostPortConflicts(pods)
        for _, pod := range notFitting {
            reason := "HostPortConflict"
            kl.recorder.Eventf(pod, reason, "Cannot start the pod due to host port
conflict. ")
            kl.statusManager.SetPodStatus(pod, api.PodStatus{
                Phase:   api.PodFailed,
                Reason:  reason,
                Message: "Pod cannot be started due to host port conflict"})
        }
        fitting, notFitting = kl.checkNodeSelectorMatching(fitting)
        for _, pod := range notFitting {
            reason := "NodeSelectorMismatching"
            kl.recorder.Eventf(pod, reason, "Cannot start the pod due to node selector
mismatch. ")
            kl.statusManager.SetPodStatus(pod, api.PodStatus{
                Phase:   api.PodFailed,
                Reason:  reason,
                Message: "Pod cannot be started due to node selector mismatch"})
        }
        fitting, notFitting = kl.checkCapacityExceeded(fitting)
        for _, pod := range notFitting {
            reason := "CapacityExceeded"
            kl.recorder.Eventf(pod, reason, "Cannot start the pod due to exceeded
capacity. ")
            kl.statusManager.SetPodStatus(pod, api.PodStatus{
                Phase:   api.PodFailed,
                Reason:  reason,
                Message: "Pod cannot be started due to exceeded capacity"})
        }
        return fitting
    }
```

最后，我们看看 statusManager 是怎么把 Channel 的数据上报到 API Server 上的，这是通过 Start 方法开启一个协程无限循环执行 syncBatch 方法来实现的，下面是 syncBatch 的代码：

```
    func (s *statusManager) syncBatch() error {
```

```
        syncRequest := <-s.podStatusChannel
        pod := syncRequest.pod
        podFullName := kubecontainer.GetPodFullName(pod)
        status := syncRequest.status

        var err error
        statusPod := &api.Pod{
            ObjectMeta: pod.ObjectMeta,
        }
    statusPod, err = s.kubeClient.Pods(statusPod.Namespace).Get(statusPod.Name)
        if err == nil {
      statusPod.Status = status
            _, err = s.kubeClient.Pods(pod.Namespace).UpdateStatus(statusPod)
            // TODO: handle conflict as a retry, make that easier too.
            if err == nil {
                glog.V(3).Infof("Status for pod %q updated successfully", kubeletUtil.
FormatPodName(pod))
                return nil
            }
        }
        go s.DeletePodStatus(podFullName)
        return fmt.Errorf("error updating status for pod %q: %v",
kubeletUtil.FormatPodName(pod), err)
    }
```

这段代码首先从 Channel 中拉取一个 syncRequest，然后调用 API Server 接口来获取最新的 Pod 信息，如果成功，则继续调用 API Server 的 UpdateStatus 接口更新 Pod 状态，如果调用失败则删除缓存的 Pod 状态，这将触发 kubelet 重新计算 Pod 状态并再次尝试更新。

说完了 Pod 流程，我们接下来再一起深入分析 Kubernetes 中的容器探针（Probe）的实现机制。我们知道，容器正常不代表里面运行的业务进程能正常工作，比如程序还没初始化好，或者配置文件错误导致无法正常服务，还有诸如数据库连接爆满导致服务异常等各种意外情况都有可能发生，面对这类问题，cAdvisor 就束手无策了，所以 kubelet 引入了容器探针技术，容器探针按照作用划分为以下两种。

- ReadinessProbe：用来探测容器中的用户服务进程是否处于“可服务状态”，此探针不会导致容器被停止或重启，而是导致此容器上的服务被标识为不可用，Kubernetes 不会发送请求到不可用的容器上，直到它们可用为止。
- LivenessProbe：用来探测容器服务是否处于“存活状态”，如果服务当前被检测为 Dead，则会导致容器重启事件发生。

下面是探针相关的结构定义：

```
type Probe struct {
```

```
    Handler
    InitialDelaySeconds  int64
    TimeoutSeconds  int64
}
type Handler struct {
    // One and only one of the following should be specified.
    Exec *ExecAction
    HTTPGet *HTTPGetAction
    TCPSocket *TCPSocketAction
}
```

从上面的定义来看，探针可以通过执行容器中的一个命令、发起一个指向容器内部的 HTTP Get 请求或者 TCP 连接来确定容器内部是否正常工作。

上面的代码属于 API 包中的一部分，只是用来描述和存储容器上的探针定义，而真正的探针实现代码则位于 pkg/kubelet/prober/prober.go 里，下面是对 prober.Probe 的定义：

```
type Prober interface {
    Probe(pod *api.Pod, status api.PodStatus, container api.Container, containerID
string, createdAt int64) (probe.Result, error)
}
```

上述接口方法表示对一个 Container 发起探测并返回其结果。prober.Probe 的实现类为 prober.prober，其结构定义如下：

```
type prober struct {
    exec   execprobe.ExecProber
    http   httprobe.HTTPProber
    tcp    tcprobe.TCPProber
    runner kubecontainer.ContainerCommandRunner
    readinessManager *kubecontainer.ReadinessManager
    refManager      *kubecontainer.RefManager
    recorder        record.EventRecorder
}
```

其中 exec、http、tcp 三个变量分别对应三种探测类型的“探头”，它们已经各自实现了相应的逻辑。比如下面这段代码是 HTTP 探头的核心逻辑，即连接一个 URL 发起 GET 请求：

```
func DoHTTPProbe(url *url.URL, client HTTPGetInterface) (probe.Result, string,
error) {
    res, err := client.Get(url.String())
    if err != nil {
        // Convert errors into failures to catch timeouts.
        return probe.Failure, err.Error(), nil
    }
    defer res.Body.Close()
    b, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return probe.Failure, "", err
```

```
        }
        body := string(b)
        if res.StatusCode >= http.StatusOK && res.StatusCode < http.StatusBadRequest {
            glog.V(4).Infof("Probe succeeded for %s, Response: %v", url.String(),
*res)
            return probe.Success, body, nil
        }
        glog.V(4).Infof("Probe failed for %s, Response: %v", url.String(), *res)
        return probe.Failure, body, nil
    }
```

prober.prober 中的 runner 则是 exec 探头的执行器，因为后者需要在被检测的容器中执行一个 cmd 命令：

```
    func (p *prober) newExecInContainer(pod *api.Pod, container api.Container,
containerID string, cmd []string) exec.Cmd {
        return execInContainer{func() ([]byte, error) {
            return p.runner.RunInContainer(containerID, cmd)
        }}
    }
```

实际上 p.runner 就是之前我们分析过的 DockerManager，下面是 RunInContainer 的源码：

```
    func (dm *DockerManager) RunInContainer(containerID string, cmd []string)
([]byte, error) {
        // If native exec support does not exist in the local docker daemon use nsinit.
        useNativeExec, err := dm.nativeExecSupportExists()
        if err != nil {
            return nil, err
        }
        if !useNativeExec {
            glog.V(2).Infof("Using nsinit to run the command %+v inside container
%s", cmd, containerID)
            return dm.runInContainerUsingNsinit(containerID, cmd)
        }
        glog.V(2).Infof("Using docker native exec to run cmd %+v inside container
%s", cmd, containerID)
        createOpts := docker.CreateExecOptions{
            Container:    containerID,
            Cmd:          cmd,
            AttachStdin:  false,
            AttachStdout: true,
            AttachStderr: true,
            Tty:          false,
        }
        execObj, err := dm.client.CreateExec(createOpts)
        if err != nil {
            return nil, fmt.Errorf("failed to run in container - Exec setup failed
```

```
- %v", err)
        }
        var buf bytes.Buffer
        startOpts := docker.StartExecOptions{
            Detach:      false,
            Tty:         false,
            OutputStream: &buf,
            ErrorStream:  &buf,
            RawTerminal:  false,
        }
        err = dm.client.StartExec(execObj.ID, startOpts)
        if err != nil {
            glog.V(2).Infof("StartExec With error: %v", err)
            return nil, err
        }
        ticker := time.NewTicker(2 * time.Second)
        defer ticker.Stop()
        for {
            inspect, err2 := dm.client.InspectExec(execObj.ID)
            if err2 != nil {
                glog.V(2).Infof("InspectExec %s failed with error: %+v", execObj.
ID, err2)
                return buf.Bytes(), err2
            }
            if !inspect.Running {
                if inspect.ExitCode != 0 {
                    glog.V(2).Infof("InspectExec %s exit with result %+v", execObj.
ID, inspect)
                    err = &dockerExitError{inspect}
                }
                break
            }
            <-ticker.C
        }

        return buf.Bytes(), err
    }
```

Docker 自 1.3 版本开始支持使用 Exec 指令（以及 API 调用）在容器内执行一个命令，我们看看上述过程中使用的 dm.client.CreateExec 方法是如何实现的：

```
    func (c *Client) CreateExec(opts CreateExecOptions) (*Exec, error) {
        path := fmt.Sprintf("/containers/%s/exec", opts.Container)
        body, status, err := c.do("POST", path, doOptions{data: opts})
        if status == http.StatusNotFound {
            return nil, &NoSuchContainer{ID: opts.Container}
        }
```

```
        if err != nil {
            return nil, err
        }
        var exec Exec
        err = json.Unmarshal(body, &exec)
        if err != nil {
            return nil, err
        }
        return &exec, nil
    }
```

我们看到，这是标准的 Docker API 的调用方式，跟之前看到的创建容器的调用代码很相似。现在我们再回头看看 prober.prober 是怎么执行 ReadinessProbe/ LivenessProbe 的检测逻辑的：

```
    func (pb *prober) Probe(pod *api.Pod, status api.PodStatus, container api.
Container, containerID string, createdAt int64) (probe.Result, error) {
        pb.probeReadiness(pod, status, container, containerID, createdAt)
        return pb.probeLiveness(pod, status, container, containerID, createdAt)
    }
```

这段代码先调用容器的 ReadinessProbe 进行检测，并且在 readinessManager 组件中记录容器的 Readiness 状态，随后调用容器的 LivenessProbe 进行检测，并返回容器的状态，在检测过程中如果发现状态为失败或者异常状态，则会连续检测 3 次：

```
    func (pb *prober) runProbeWithRetries(p *api.Probe, pod *api.Pod, status api.
PodStatus, container api.Container, containerID string, retries int) (probe.Result,
string, error) {
        var err error
        var result probe.Result
        var output string
        for i := 0; i < retries; i++ {
            result, output, err = pb.runProbe(p, pod, status, container, containerID)
            if result == probe.Success {
                return probe.Success, output, nil
            }
        }
        return result, output, err
    }
```

比较意外的是 prober.prober 探针检测容器状态的方法目前只在一处被调用到，位于方法 DockerManager.computePodContainerChanges 里：

```
    result, err := dm.prober.Probe(pod, podStatus, container, string(c.ID), c.
Created)
            if err != nil {
                // TODO(vmarmol): examine this logic.
                glog.V(2).Infof("probe no-error: %q", container.Name)
                containersToKeep[containerID] = index
```

```
            continue
        }
        if result == probe.Success {
            glog.V(4).Infof("probe success: %q", container.Name)
            containersToKeep[containerID] = index
            continue
        }
        glog.Infof("pod %q container %q is unhealthy (probe result: %v), it will
be killed and re-created. ", podFullName, container.Name, result)
        containersToStart[index] = empty{}
    }
```

只有没有发生任何变化的 Pod 才会执行一次探针检测，若检测状态为失败，则会导致重启事件发生。

本节最后，我们再来简单分析下 kubelet 中的 Kubelet Server 的实现机制，下面是 kubelet 进程启动过程中启动 Kubelet Server 的源码入口：

```
// start the kubelet server
    if kc.EnableServer {
        go util.Forever(func() {
            k.ListenAndServe(net.IP(kc.Address), kc.Port, kc.TLSOptions, kc.
EnableDebuggingHandlers)
        }, 0)
}
```

在上述代码调用的过程中，创建了一个类型为 kubelet.Server 的 HTTP Server 并在本地监听：

```
handler := NewServer(host, enableDebuggingHandlers)
    s := &http.Server{
        Addr:           net.JoinHostPort(address.String(), strconv.FormatUint
(uint64(port), 10)),
        Handler:        &handler,
        ReadTimeout:    5 * time.Minute,
        WriteTimeout:   5 * time.Minute,
        MaxHeaderBytes: 1 << 20,
    }
    if tlsOptions != nil {
        s.TLSConfig = tlsOptions.Config
        glog.Fatal(s.ListenAndServeTLS(tlsOptions.CertFile, tlsOptions. KeyFile))
    } else {
        glog.Fatal(s.ListenAndServe())
    }
```

在 kubelet.Server 的构造函数里加载如下 HTTP Handler：

```
func (s *Server) InstallDefaultHandlers() {
    healthz.InstallHandler(s.mux,
        healthz.PingHealthz,
```

```
        healthz.NamedCheck("docker", s.dockerHealthCheck),
        healthz.NamedCheck("hostname", s.hostnameHealthCheck),
        healthz.NamedCheck("syncloop", s.syncLoopHealthCheck),
    )
    s.mux.HandleFunc("/pods", s.handlePods)
    s.mux.HandleFunc("/stats/", s.handleStats)
    s.mux.HandleFunc("/spec/", s.handleSpec)
}
```

上述 Handler 分为两组：首先是健康检查，包括 kubelet 进程自身的心跳检查、Docker 进程的健康检查、kubelet 所在主机名检测、Pod 同步的健康检查等；然后是获取当前节点上运行期信息的接口，例如获取当前节点上的 Pod 列表、统计信息等。下面是 hostnameHealthCheck 的实现逻辑，它检查 Pod 两次同步之间的时延，而这个时延则在之前提到的 kubelet 的 syncLoopIteration 方法中进行更新：

```
func (s *Server) syncLoopHealthCheck(req *http.Request) error {
    duration := s.host.ResyncInterval() * 2
    minDuration := time.Minute * 5
    if duration < minDuration {
        duration = minDuration
    }
    enterLoopTime := s.host.LatestLoopEntryTime()
    if !enterLoopTime.IsZero() && time.Now().After(enterLoopTime.Add(duration)) {
        return fmt.Errorf("Sync Loop took longer than expected. ")
    }
    return nil
}
```

handlePods 的 API 则从 kubelet 中获取当前“绑定”到本节点的所有 Pod 的信息并返回：

```
func (s *Server) handlePods(w http.ResponseWriter, req *http.Request) {
    pods := s.host.GetPods()
    data, err := encodePods(pods)
    if err != nil {
        s.error(w, err)
        return
    }
    w.Header().Add("Content-type", "application/json")
    w.Write(data)
}
```

如果 kubelet 运行在 Debug 模式，则加载更多的 HTTP Handler：

```
func (s *Server) InstallDebuggingHandlers() {
    s.mux.HandleFunc("/run/", s.handleRun)
    s.mux.HandleFunc("/exec/", s.handleExec)
    s.mux.HandleFunc("/portForward/", s.handlePortForward)
```

```
    s.mux.HandleFunc("/logs/", s.handleLogs)
    s.mux.HandleFunc("/containerLogs/", s.handleContainerLogs)
    s.mux.Handle("/metrics", prometheus.Handler())
    // The /runningpods endpoint is used for testing only.
    s.mux.HandleFunc("/runningpods", s.handleRunningPods)

    s.mux.HandleFunc("/debug/pprof/", pprof.Index)
    s.mux.HandleFunc("/debug/pprof/profile", pprof.Profile)
    s.mux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
}
```

这些 HTTP Handler 的实现并不复杂，所以在这里就不再一一介绍了。

6.5.3　设计总结

在研读 kubelet 源码的过程中，你经常会有“山穷水尽疑无路，柳暗花明又一村”的感觉，是因为在它的设计中大量运用了 Channel 这种异步消息机制，加之为了测试的方便，又将很多重要的处理函数做成接口类，只有找到并分析这些接口的具体实现类，才能明白整个流程。这对于习惯了面向对象语言的程序员而言，有一种一夜回到解放前的感觉。

因为 kubelet 的功能比较多，所以我们在此仅以 Pod 同步的主流程为例，进行一个设计总结，图 6.8 是 kubelet 主流程相关的设计示意图，为了更加清晰地展示整个流程，我们特意将 kubelet Kernel、Docker System 与其他部分分离开来，并且省略了部分非核心对象和数据结构。

首先，config.PodConfig 创建一个或多个 Pod Source，在默认情况下创建的是 API source，它并没有创建新的数据结构，而是使用之前介绍的 cache.Reflector 结合 cache.UndeltaStore，从 Kubernetes API Server 上拉取 Pod 数据放入内部的 Channel 上，而内部的 Channel 收到 Pod 数据后会调用 podStorage 的 Merge 方法实现多个 Channel 数据的合并，产生 kubelet.PodUpdate 消息并写入 PodConfig 的汇总 Channel 上，随后 PodUpdate 消息进入 kubelet Kernel 中进行下一步处理。

kubelet.kubelet 的 syncLoop 方法监听 PodConfig 的汇总 Channel，过滤掉不合适的 PodUpdate 并把符合条件的放入 SyncPods 方法中，最终为每个符合条件的 Pod 产生一个 kubelet.workUpdate 事件并放入 podWorkers 的内部工作队列上，随后调用 podWorkers 的 managePodLoop 方法进行处理。podWorkers 在处理流程中调用了 DockerManager 的 SyncPod 方法，由此 DockerManager 接班，在进行了必要的 Pod 周边操作后，对于需要重启或者更新的容器，DockerManager 则交给 docker.Client 对象去执行具体的动作，后者通过调用 Dockers Engine 的 API Service 来实现具体功能。

在 Pod 同步的过程中会产生 Pod 状态的变更和同步问题，这些是交由 kubelet.statusManager 实现的，它在内部也采用了 Channel 的设计方式。

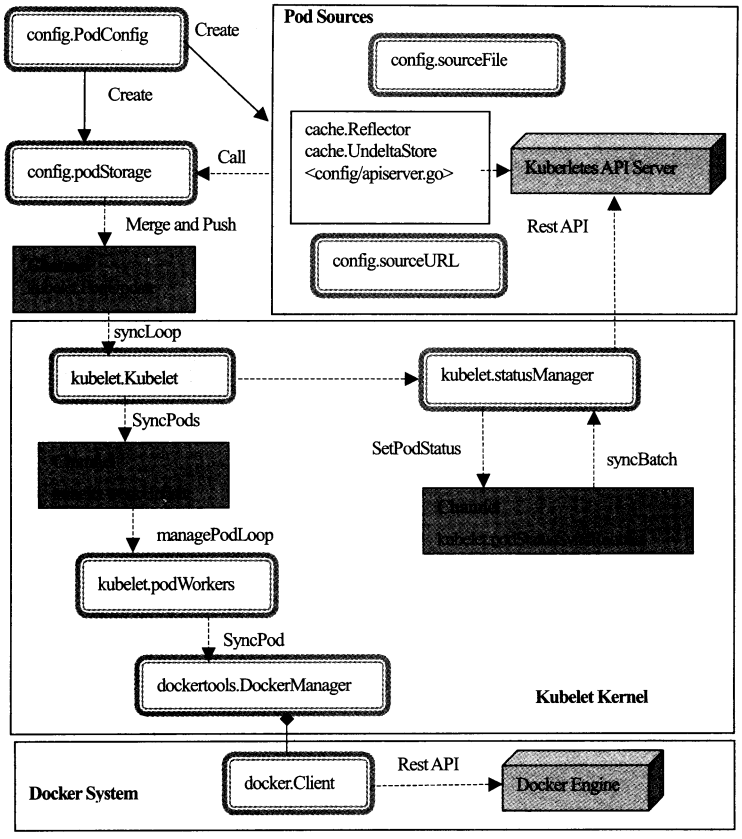

图 6.8　kubelet 主流程相关的设计示意图

6.6　kube-proxy 进程源码分析

kube-proxy 是运行在 Minion 节点上的另外一个重要的守护进程，你可以把它当作一个 HAProxy，它充当了 Kubernetes 中 Service 的负载均衡器和服务代理的角色。下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.6.1　进程启动过程

kube-proxy 进程的入口类源码位置如下：

```
github/com/GoogleCloudPlatform/kubernetes/cmd/kube-proxy/proxy.go
```

入口 main()函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewProxyServer()
    s.AddFlags(pflag.CommandLine)

    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()

    verflag.PrintAndExitIfRequested()

    if err := s.Run(pflag.CommandLine.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

上述代码构造了一个 ProxyServer，然后调用它的 Run 方法启动运行。首先我们看看 NewProxyServer 的代码：

```
func NewProxyServer() *ProxyServer {
    return &ProxyServer{
        BindAddress:       util.IP(net.ParseIP("0.0.0.0")),
        HealthzPort:       10249,
        HealthzBindAddress: util.IP(net.ParseIP("127.0.0.1")),
        OOMScoreAdj:       -899,
        ResourceContainer:  "/kube-proxy",
    }
}
```

在上述代码中，ProxyServer 绑定本地所有 IP（0.0.0.0）对外提供代理服务，而提供健康检查的 HTTP Server 则默认绑定本地的回环 IP，说明后者仅用于在本节点上访问，如果需要开发管理系统进行远程管理，则可以设置参数 healthz-bind-address 为 0.0.0.0 来达到目的。另外，从代码中看，ProxyServer 还有一个重要属性可以调整：PortRange（对应命令行参数为 proxy-port-range），它用来限定 ProxyServer 使用哪些本地端口作为代理端口，默认是随机选择。

ProxyServer 的 Run 方法流程如下。

- 设置本进程的 OOM 参数 OOMScoreAdj，保证系统 OOM 时，kube-proxy 不会首先被系统删除，这是因为 kube-proxy 与 kubelet 进程一样，比节点上的 Pod 进程更重要。
- 让自己的进程运行在指定的 Linux Container 中，这个 Container 的名字来自 ProxyServer.ResourceContainer，如上所述，默认为/kube-proxy，比较重要的一点是这个 Container 具备所有设备的访问权。

- 创建 ServiceConfig 与 EndpointsConfig，它们与之前 kubelet 中的 PodConfig 的作用和实现机制有点像，分别负责监听和拉取 API Server 上 Service 与 Service Endpoints 的信息，并通知给注册到它们上的 Listener 接口进行处理。
- 创建一个 round-robin 轮询机制的 load balancer（LoadBalancerRR），它用来实现 Service 的负载均衡转发逻辑，它也是前面创建的 EndpointsConfig 的一个 Listener。
- 创建一个 Proxier，它负责建立和维护 Service 的本地代理 Socket，它也是前面创建的 ServiceConfig 的一个 Listener。
- 创建一个 config.SourceAPI，并启动两个协程，通过 Kubernetes Client 来拉取 Kubernetes API Server 上的 Service 与 Endpoint 数据，然后分别写入之前定义的 ServiceConfig 与 EndpointsConfig 的 Channel 上，从而触发整个流程的驱动。
- 本地绑定健康检查的 HTTP Server 提供服务。
- 进入 Proxier 的 SyncLoop 方法里，该方法周期性地检查 Iptables 是否设置正常、服务的 Portal 是否正常开启，以及清除 load balancer 上的过期会话。

从启动流程看，kube-proxy 进程的参数比较少，它所做的事情也是比较单一的，没有 kubelet 进程那么复杂，在下一节我们会深入分析其关键代码。

6.6.2 关键代码分析

从上一节 kube-proxy 的启动流程来看，它跟 kubelet 有相似的地方，即都会从 Kubernetes API Server 拉取相关的资源数据并在本地节点上完成“深加工”，其拉取资源的做法，第一眼看上去与 kubelet 相似，但实际上有稍微不同的实现思路，这说明作者另有其人。

由于 ServiceConfig 与 EndpointsConfig 实现机制是完全一样的，只不过拉取的资源不同，所以我们这里仅对前者做深入分析。首先从 ServiceConfig 结构体开始：

```
type ServiceConfig struct {
   mux     *config.Mux
   bcaster *config.Broadcaster
   store   *serviceStore
}
```

ServiceConfig 也使用了 mux(config.Mux)，它是一个多 Channel 的多路合并器，之前 kubelet 的 PodConfig 也用到了它。下面是 ServiceConfig 的构造函数：

```
func NewServiceConfig() *ServiceConfig {
   updates := make(chan struct{})
   store := &serviceStore{updates: updates, services:
make(map[string]map[types.NamespacedName]api.Service)}
```

```
    mux := config.NewMux(store)
    bcaster := config.NewBroadcaster()
    go watchForUpdates(bcaster, store, updates)
    return &ServiceConfig{mux, bcaster, store}
}
```

从上述代码来看，store 是 serviceStore 的一个实例。它作为 config.Mux 的 Merge 接口的实现，负责处理 config.Mux 的 Channel 上收到的 ServiceUpdate 消息并更新 store 的内部变量 services，后者是一个 Map，存放了最新同步到本地的 api.Service 资源，是 Service 的全量数据。下面是 Merge 方法的逻辑：

```
func (s *serviceStore) Merge(source string, change interface{}) error {
    s.serviceLock.Lock()
    services := s.services[source]
    if services == nil {
        services = make(map[types.NamespacedName]api.Service)
    }
    update := change.(ServiceUpdate)
    switch update.Op {
    case ADD:
        glog.V(4).Infof("Adding new service from source %s : %+v", source, update.
Services)
        for _, value := range update.Services {
            name := types.NamespacedName{value.Namespace, value.Name}
            services[name] = value
        }
    case REMOVE:
        glog.V(4).Infof("Removing a service %+v", update)
        for _, value := range update.Services {
            name := types.NamespacedName{value.Namespace, value.Name}
            delete(services, name)
        }
    case SET:
        glog.V(4).Infof("Setting services %+v", update)
        // Clear the old map entries by just creating a new map
        services = make(map[types.NamespacedName]api.Service)
        for _, value := range update.Services {
            name := types.NamespacedName{value.Namespace, value.Name}
            services[name] = value
        }
    default:
        glog.V(4).Infof("Received invalid update type: %v", update)
    }
    s.services[source] = services
    s.serviceLock.Unlock()
    if s.updates != nil {
        s.updates <- struct{}{}
```

```
    }
    return nil
}
```

serviceStore 同时是 config.Accessor 接口的一个实现，MergedState 接口方法返回之前 Merge 最新的 Service 全量数据。

```
func (s *serviceStore) MergedState() interface{} {
    s.serviceLock.RLock()
    defer s.serviceLock.RUnlock()
    services := make([]api.Service, 0)
    for _, sourceServices := range s.services {
        for _, value := range sourceServices {
            services = append(services, value)
        }
    }
    return services
}
```

上述方法在哪里被用到了呢？就在之前提到的 NewServiceConfig 方法里：

```
go watchForUpdates(bcaster, store, updates)
```

一个协程监听 serviceStore 的 updates(Channel)，在收到事件以后就调用上述 MergedState 方法，将当前最新的 Service 数组通知注册到 bcaster 上的所有 Listener 进行处理。下面分别给出了 watchForUpdates 及 Broadcaster 的 Notify 方法的源码：

```
    func watchForUpdates(bcaster *config.Broadcaster, accessor config.Accessor,
updates <-chan struct{}) {
        for true {
            <-updates
            bcaster.Notify(accessor.MergedState())
        }
    }
    func (b *Broadcaster) Notify(instance interface{}) {
        b.listenerLock.RLock()
        listeners := b.listeners
        b.listenerLock.RUnlock()
        for _, listener := range listeners {
            listener.OnUpdate(instance)
        }
    }
```

上述逻辑的精巧设计之处在于，当 ServiceConfig 完成 Merge 调用后，为了及时通知 Listener 进行处理，就产生一个“空事件”并写入 updates 这个 Channel 中，另外监听此 Channel 的协程就及时得到通知，触发 Listener 的回调动作。ServiceConfig 这里注册的 Listener 是 proxy.Proxier 对象，我们以后会继续分析它的回调函数 OnUpdate 是如何使用 Service 数据的。

接下来，我们看看 ServiceUpdate 事件是怎么生成并传递到 ServiceConfig 的 Channel 上的。在 kube-proxy 启动流程中有调用 config.NewSourceAPI 函数，其内部生成了一个 servicesReflector 对象：

```
type servicesReflector struct {
    watcher          ServicesWatcher
    services         chan<- ServiceUpdate
    resourceVersion   string
    waitDuration      time.Duration
    reconnectDuration time.Duration
}
```

其中 services 这个 Channel 是用来写入 ServiceUpdate 事件的，它是 ServiceConfig 的 Channel (source string)方法所创建并返回的 Channel，它写入数据后就会被一个协程立即转发到 ServiceConfig 的 Channel 里。下面这段代码完整地揭示了上述逻辑：

```
func (c *ServiceConfig) Channel(source string) chan ServiceUpdate {
    ch := c.mux.Channel(source)
    serviceCh := make(chan ServiceUpdate)
    go func() {
        for update := range serviceCh {
            ch <- update
        }
        close(ch)
    }()
    return serviceCh
}
```

servicesReflector 中的 watcher 用来从 API Server 上拉取 Service 数据，它是 client.Services (api.NamespaceAll)返回的 client.ServiceInterface 实例对象的一个引用，属于标准的 Kubernetes client 包。在 config.NewSourceAPI 的方法里，启动了一个协程周期性地调用 watcher 的 list 与 Watch 方法获取数据，然后转换成 ServiceUpdate 事件，写入 Channel 中。下面是关键源码：

```
func (s *servicesReflector) run(resourceVersion *string) {
    if len(*resourceVersion) == 0 {
        services, err := s.watcher.List(labels.Everything())
        if err != nil {
            glog.Errorf("Unable to load services: %v", err)
            // TODO: reconcile with pkg/client/cache which doesn't use reflector.
            time.Sleep(wait.Jitter(s.waitDuration, 0.0))
            return
        }
        *resourceVersion = services.ResourceVersion
        // TODO: replace with code to update the
        s.services <- ServiceUpdate{Op: SET, Services: services.Items}
    }
```

```
        watcher, err := s.watcher.Watch(labels.Everything(), fields.Everything(),
*resourceVersion)
        if err != nil {
            glog.Errorf("Unable to watch for services changes: %v", err)
            if !client.IsTimeout(err) {
                // Reset so that we do a fresh get request
                *resourceVersion = ""
            }
            time.Sleep(wait.Jitter(s.waitDuration, 0.0))
            return
        }
        defer watcher.Stop()
        ch := watcher.ResultChan()
        s.watchHandler(resourceVersion, ch, s.services)
    }
```

在上面的代码中，初始时资源版本变量 resourceVersion 为空，于是会执行 Service 的全量拉取动作（watcher.List），之后 Watch 资源会开始发生变化（watcher.Watch）并将 Watch 的结果（一个 Channel 保持了 Service 的变动数据）也转换为对应的 ServiceUpdate 事件并写入 Channel 中。另外，当拉取数据的调用发生异常时，resourceVersion 恢复为空，导致重新进行全量资源的拉取动作。这种自修复能力的编程设计足以见证谷歌大神们的深厚编程功力；另外，笔者认为 kube-proxy 这里的 ServiceConfig 的设计实现思路和代码要比 kubelet 中的好一点，虽然两个作者都是顶尖高手。

接下来才开始进入本节的重点，即服务代理的实现机制分析。首先，我们从代码中的 load balance 组件说起。下面是 kube-proxy 中定义的 Load Balancer 接口：

```
    type LoadBalancer interface {
      NextEndpoint(service ServicePortName, srcAddr net.Addr) (string, error)
      NewService(service ServicePortName, sessionAffinityType api.ServiceAffinity,
stickyMaxAgeMinutes int) error
     CleanupStaleStickySessions(service ServicePortName)
    }
```

LoadBalancer 有 3 个接口，其中 NextEndpoint 方法用于给访问指定 Service 的新客户端请求分配一个可用的 Endpoint 地址；NewService 用来添加一个新服务到负载均衡器上；CleanupStaleStickySessions 则用来清理过期的 Session 会话。目前 kube-proxy 只实现了一个基于 round-robin 算法的负载均衡器，它就是 proxy. LoadBalancerRR 组件。

LoadBalancerRR 采用了 affinityState 这个结构体来保存当前客户端的会话信息，然后在 affinityPolicy 里用一个 Map 来记录（属于某个 Service 的）所有活动的客户端会话，这是它实现 Session 亲和性的负载均衡调度的基础。

```
    type affinityState struct {
       clientIP string
```

```
    //clientProtocol  api.Protocol //not yet used
    //sessionCookie   string      //not yet used
    endpoint string
    lastUsed time.Time
}
type affinityPolicy struct {
    affinityType api.ServiceAffinity
    affinityMap  map[string]*affinityState // map client IP -> affinity info
    ttlMinutes   int
}
```

balancerState 用来记录一个 Service 的所有 Endpoint（数组）、当前所使用的 Endpoint 的 index，以及对应的所有活动的客户端会话（affinityPolicy）。其定义如下：

```
type balancerState struct {
    endpoints []string // a list of "ip:port" style strings
    index     int      // current index into endpoints
    affinity  affinityPolicy
}
```

有了上面的认识，再看 LoadBalancerRR 的构造函数就简单多了，它内部用一个 map 记录每个服务的 balancerState 状态，当然初始化时还是空的：

```
func NewLoadBalancerRR() *LoadBalancerRR {
    return &LoadBalancerRR{
        services: map[ServicePortName]*balancerState{},
    }
}
```

LoadBalancerRR 的 NewService 方法代码很简单，就是在它的 services 里增加一个记录项，用户端的会话超时时间 ttlMinutes 默认为 3 小时，下面是相关源码：

```
    func (lb *LoadBalancerRR) NewService(svcPort ServicePortName, affinityType
api.ServiceAffinity, ttlMinutes int) error {
        lb.lock.Lock()
        defer lb.lock.Unlock()
        lb.newServiceInternal(svcPort, affinityType, ttlMinutes)
        return nil
    }
    func (lb *LoadBalancerRR) newServiceInternal(svcPort ServicePortName, affinityType
api.ServiceAffinity, ttlMinutes int) *balancerState {
        if ttlMinutes == 0 {
            ttlMinutes = 180
        }
        if _, exists := lb.services[svcPort]; !exists {
            lb.services[svcPort] = &balancerState{affinity:
*newAffinityPolicy(affinityType, ttlMinutes)}
            glog.V(4).Infof("LoadBalancerRR service %q did not exist, created",
svcPort)
```

```
        } else if affinityType != "" {
            lb.services[svcPort].affinity.affinityType = affinityType
        }
        return lb.services[svcPort]
    }
```

我们在前面提到过 ServiceConfig 同步并监听 API Server 上的 api.Service 的数据变化，然后调用 Listener（proxy.Proxier 是 ServiceConfig 唯一注册的 Listener）的 OnUpdate 接口完成通知。而上述 NewService 就是在 proxy.Proxier 的 OnUpdate 方法里被调用的，从而实现了 Service 自动添加到 LoadBalancer 的机制。

我们再来看 LoadBalancerRR 的 NextEndpoint 方法，它实现了经典的 round-robin 负载均衡算法。NextEndpoint 方法首先判断当前服务是否有保持会话（sessionAffinity）的要求，如果有，则看当前请求是否有连接可用：

```
    if sessionAffinityEnabled {
            // Caution: don't shadow ipaddr
            var err error
            ipaddr, _, err = net.SplitHostPort(srcAddr.String())
            if err != nil {
                return "", fmt.Errorf("malformed source address %q: %v", srcAddr.
String(), err)
            }
            sessionAffinity, exists := state.affinity.affinityMap[ipaddr]
            if exists && int(time.Now().Sub(sessionAffinity.lastUsed).Minutes()) <
state.affinity.ttlMinutes {
                // Affinity wins.
                endpoint := sessionAffinity.endpoint
                sessionAffinity.lastUsed = time.Now()
                glog.V(4).Infof("NextEndpoint for service %q from IP %s with
sessionAffinity %+v: %s", svcPort, ipaddr, sessionAffinity, endpoint)
                return endpoint, nil
            }
        }
```

如果服务无须会话保持、新建会话及会话过期，则采用 round-robin 算法得到下一个可用的服务端口，如果服务有会话保持需求，则保存当前的会话状态：

```
    // Take the next endpoint.
        endpoint := state.endpoints[state.index]
        state.index = (state.index + 1) % len(state.endpoints)
        if sessionAffinityEnabled {
            var affinity *affinityState
            affinity = state.affinity.affinityMap[ipaddr]
            if affinity == nil {
                affinity = new(affinityState) //&affinityState{ipaddr, "TCP", "",
endpoint, time.Now()}
```

```
            state.affinity.affinityMap[ipaddr] = affinity
        }
        affinity.lastUsed = time.Now()
        affinity.endpoint = endpoint
        affinity.clientIP = ipaddr
        glog.V(4).Infof("Updated affinity key %s: %+v", ipaddr, state.affinity.
affinityMap[ipaddr])
    }
    return endpoint, nil
```

接下来我们看看 Service 的 Endpoint 信息是如何添加到 LoadBalancerRR 上的？答案很简单，类似之前我们分析过的 ServiceConfig。kube-proxy 也设计了一个 EndpointsConfig 来拉取和监听 API Server 上的服务的 Endpoint 信息，并调用 LoadBalancerRR 的 OnUpdate 接口完成通知，在这个方法里，LoadBalancerRR 完成了服务访问端口的添加和同步逻辑。

我们先来看看 api.Endpoints 的定义：

```
type EndpointAddress struct {
    IP string
    TargetRef *ObjectReference
}
type EndpointPort struct {
    Name string
    Port int
    Protocol Protocol
}
type EndpointSubset struct {
    Addresses []EndpointAddress
    Ports     []EndpointPort
}
type Endpoints struct {
    TypeMeta   `json: ",inline"`
    ObjectMeta `json: "metadata,omitempty"`
    Subsets []EndpointSubset
}
```

一个 EndpointAddress 与 EndpointPort 对象可以组成一个服务访问地址，而在 EndpointSubset 对象里则定义了两个单独的 EndpointAddress 与 EndpointPort 数组而不是“服务访问地址”的一个列表。初看这样的定义你可能会觉得很奇怪，为什么没有设计一个 Endpoint 结构？这里的深层次原因在于，Service 的 Endpoint 信息来源于两个独立的实体：Pod 与 Service，前者负责提供 IP 地址即 EndpointAddress，而后者负责提供 Port 即 EndpointPort。由于在一个 Pod 上可以运行多个 Service，而一个 Service 也通常跨越多个 Pod，于是就产生了一个“笛卡尔乘积”的 Endpoint 列表，这就是 EndpointSubset 的设计灵感。

举例说明，对于如下表示的 EndpointSubset：

```
{
  Addresses: [{"ip": "10.10.1.1"}, {"ip": "10.10.2.2"}],
  Ports: [{"name": "a", "port": 8675}, {"name": "b", "port": 309}]
}
```

会产生如下 Endpoint 列表：

```
a: [ 10.10.1.1:8675, 10.10.2.2:8675 ],
b: [ 10.10.1.1:309, 10.10.2.2:309 ]
```

LoadBalancerRR 的 OnUpdate 方法里循环对每个 api.Endpoints 进行处理，先把它转化为一个 Map，Map 的 Key 是 EndpointPort 的 Name 属性（代表一个 Service 的访问端口）；而 Value 则是 hostPortPair 的一个数组，hostPortPair 其实就是之前缺失的 Endpoint 结构体，包括一个 IP 地址与端口属性，即某个服务在一个 Pod 上的对应访问端口。

```
    portsToEndpoints := map[string][]hostPortPair{}
           for i := range svcEndpoints.Subsets {
               ss := &svcEndpoints.Subsets[i]
               for i := range ss.Ports {
                   port := &ss.Ports[i]
                   for i := range ss.Addresses {
                       addr := &ss.Addresses[i]
                       portsToEndpoints[port.Name] = append(portsToEndpoints
[port.Name], hostPortPair{addr.IP, port.Port})
                       // Ignore the protocol field - we'll get that from the Service
objects.
                   }
               }
           }
```

下一步，针对 portsToEndpoints 进行循环处理。对于每个记录，判断是否已经在 services 中存在，并做出相应的更新或跳过的逻辑，最后删除那些已经不在集合中的端口，完成整个同步逻辑。下面是相关代码：

```
    for portname := range portsToEndpoints {
              svcPort := ServicePortName{types.NamespacedName{svcEndpoints.Namespace,
svcEndpoints.Name}, portname}
               state, exists := lb.services[svcPort]
               curEndpoints := []string{}
               if state != nil {
                   curEndpoints = state.endpoints
               }
               newEndpoints := flattenValidEndpoints(portsToEndpoints[portname])

               if !exists || state == nil || len(curEndpoints) != len(newEndpoints)
|| !slicesEquiv(slice.CopyStrings(curEndpoints), newEndpoints) {
                   glog.V(1).Infof("LoadBalancerRR: Setting endpoints for %s to %+v",
svcPort, newEndpoints)
```

```
                lb.updateAffinityMap(svcPort, newEndpoints)
                // OnUpdate can be called without NewService being called externally.
                // To be safe we will call it here.  A new service will only be created
                // if one does not already exist.  The affinity will be updated
                // later, once NewService is called.
                state = lb.newServiceInternal(svcPort, api.ServiceAffinity(""), 0)
                state.endpoints = slice.ShuffleStrings(newEndpoints)

                // Reset the round-robin index.
                state.index = 0
            }
            registeredEndpoints[svcPort] = true
        }
    }
    // Remove endpoints missing from the update.
    for k := range lb.services {
        if _, exists := registeredEndpoints[k]; !exists {
            glog.V(2).Infof("LoadBalancerRR: Removing endpoints for %s", k)
            delete(lb.services, k)
        }
    }
```

LoadBalancerRR 的代码总体来说还是比较简单的，它主要被 kube-proxy 中的关键组件 proxy.Proxier 所使用，后者用到的主要数据结构为 proxy.serviceInfo，它定义和保存了一个 Service 的代理过程中的必要参数和对象。下面是其定义：

```
type serviceInfo struct {
    portal            portal
    protocol          api.Protocol
    proxyPort         int
    socket            proxySocket
    timeout           time.Duration
    nodePort          int
    loadBalancerStatus  api.LoadBalancerStatus
    sessionAffinityType api.ServiceAffinity
    stickyMaxAgeMinutes int
    // Deprecated, but required for back-compat (including e2e)
    deprecatedPublicIPs []string
}
```

serviceInfo 的各个属性解释如下。

- portal：用于存放服务的 Portal 地址，即 Service 的 Cluster IP（VIP）地址与端口。
- protcal：服务的 TCP，目前是 TCP 与 UDP。
- socket、proxyPort：socket 是 Proxier 在本机上为该服务打开的代理 Socket；proxyPort 则是这个代理 Socket 的监听端口。

- timeout：目前只用于 UDP 的 Service，表明服务“链接”的超时时间。
- nodePort：该服务定义的 NodePort。
- loadBalancerStatus：在 Cloud 环境下，如果存在由 Cloud 服务提供者提供的负载均衡器（软件或硬件）用作 Kubernetes Service 的负载均衡，则这里存放这些负载均衡器的 IP 地址。
- sessionAffinityType：该服务的负载均衡调度是否保持会话。
- stickyMaxAgeMinutes：即前面说的 Session 过期时间。
- deprecatedPublicIPs：已过期、废弃的服务的 Public IP 地址。

理解了 serviceInfo，我们再来看 Proxier 的数据结构：

```
type Proxier struct {
   loadBalancer  LoadBalancer
   mu           sync.Mutex // protects serviceMap
   serviceMap    map[ServicePortName]*serviceInfo
   portMapMutex  sync.Mutex
   portMap       map[portMapKey]ServicePortName
   numProxyLoops int32
   listenIP      net.IP
   iptables      iptables.Interface
   hostIP        net.IP
   proxyPorts    PortAllocator
}
```

Proxier 用一个 Map 维护了每个服务的 serviceInfo 信息，同时为了快速查询和检测服务端口是否有冲突，比如定义了两个一样端口的服务，又设计了一个 portMap，其 Key 为服务的端口信息（portMapKey 由 port 和 protocol 组合而成），value 为 ServicePortName。Proxier 的 listenIP 为 Proxier 监听的本节点 IP，它在这个 IP 上接收请求并做转发代理。由于每个服务的 proxySocket 在本节点监听的 Port 端口默认是系统随机分配的，所以使用 PortAllocator 来分配这个端口。另外，Service 的 Portal 与 NodePort 是通过 Linux 防火墙机制来实现的，因此这里引用了 Iptables 的组件完成相关操作。

要想理解 Proxier 中使用 Iptables 的方式，首先我们要弄明白 Kubernetes 中 Service 访问的一些网络细节。先来看看图 6.9，这是一个外部应用通过 NodePort（TCP：//NodeIP:NodePort）来访问 Service 时的网络流量示意图。访问流量进入节点网卡 eth0 后，到达 Iptables 的 PREROUTING 链，通过 KUBE-NODEPORT-CONTAINER 这个 NAT 规则被转发到 kube-proxy 进程上该 Service 对应的 Proxy 端口，然后由 kube-proxy 进程进行负载均衡并且将流量转发到 Service 所在 Container 的本地端口。

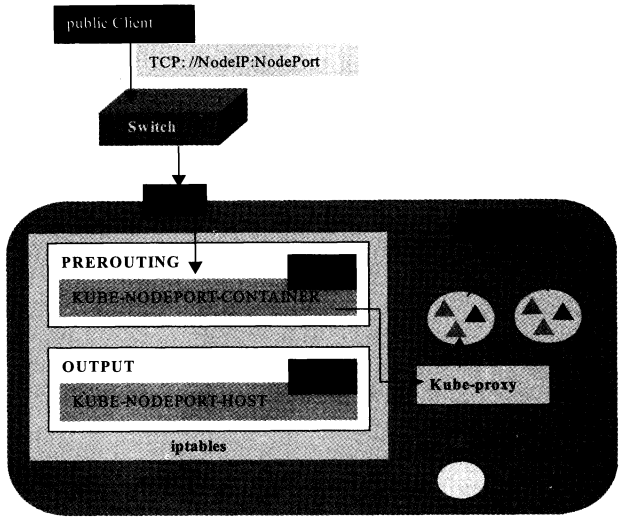

图 6.9　外部应用通过 NodePort 访问 Service 的网络流量示意图

根据 Iptables 的机制，本地进程发起的流量会经过 Iptables 的 OUTPUT 链，于是 kube-proxy 在这里也增加了相同作用的 NAT 规则：KUBE-NODEPORT-HOST。这样一来，如果本地容器内的进程以 NodePort 方式来访问 Service，则流量也会被转发到 kube-proxy 上，虽然以这种方式访问的情况比较少见。

服务之间通过 Service Portal 方式访问的流量转发机制跟 NodePort 方式在本质上是一样的，也是通过NAT，如图6.10所示。当Service A用Service B的Portal地址去访问时，流量经过Iptables的 OUTPUT 链经 NAT 规则 KUBE-PORTALS-HOST 的转换被转发到 kube-proxy 上，然后被转发给 Service B 所在的容器。

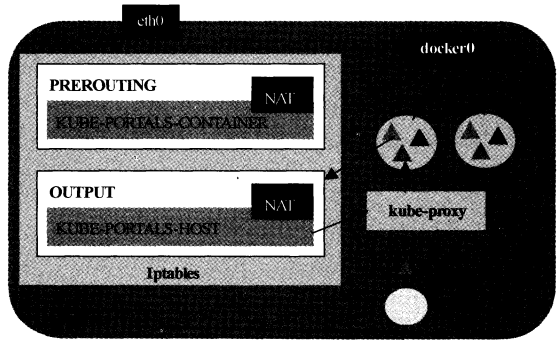

图 6.10　以 Service Portal 方式访问 Service 的流量示意图

Proxier 在创建 Iptables 的 PREROUTING 链中的 NAT 转发规则时，有一些特殊性，源码作者在代码中做了如下注释：

“这是一个复杂的问题。

如果 Proxy 的 Proxier.listenIP 设置为 0.0.0.0，即绑定到所有端口上，那么我们采用 REDIRECT 这种方式进行流量转发，因为这种情况下，返回的流量与进入的流量使用同一个网络端口，这就满足了 NAT 的规则。其他情况则采用 DNAT 转发流量，但 DNAT 到 127.0.0.1 时，流量会消失，这似乎是 Iptables 的一个众所周知的问题，所以这里不允许 Proxy 绑定到 localhost 上。”

现在再看下面这段代码就容易理解了，用来生成 KUBE-NODEPORT-CONTAINER 这条 NAT 规则：

```
    func (proxier *Proxier) iptablesContainerNodePortArgs(nodePort int, protocol
api.Protocol, proxyIP net.IP, proxyPort int, service ServicePortName) []string {
        args := iptablesCommonPortalArgs(nil, nodePort, protocol, service)
        if proxyIP.Equal(zeroIPv4) || proxyIP.Equal(zeroIPv6) {
            // TODO: Can we REDIRECT with IPv6?
            args = append(args, "-j", "REDIRECT", "--to-ports", fmt.Sprintf("%d",
proxyPort))
        } else {
            // TODO: Can we DNAT with IPv6?
            args = append(args, "-j", "DNAT", "--to-destination", net. JoinHostPort
(proxyIP.String(), strconv.Itoa(proxyPort)))
        }
        return args
    }
```

弄明白 Proxier 中关于 Iptables 的事情之后，我们来研究分析下 Proxier 如何在 OnUpdate 方法里为每个 Service 建立起对应的 Proxy 并完成同步工作。首先，在 OnUpdate 方法里创建一个 map（activeServices）来标识当前所有 alive 的 Service，key 为 ServicePortName，然后对 OnUpdate 参数里的 Service 数组进行循环，判断每个 Service 是否需要进行新建、变更或者删除操作，对于需要新建或者变更的 Service，先用 PortAllocator 获取一个新的未用的本地代理端口，然后调用 addServiceOnPort 方法创建一个 ProxySocket 用于实现此服务的代理，接着调用 openPortal 方法添加 iptables 里的 NAT 映射规则，最后调用 LoadBalancer 的 NewService 方法把该服务添加到负载均衡器上。OnUpdate 方法的最后一段逻辑是处理已经被删除的 Service，对于每个要被删除的 Service，先删除 Iptables 中相关的 NAT 规则，然后关闭对应的 proxySocket，最后释放 ProxySocket 占用的监听端口并将该端口“还给” PortAllocator。

从上面的分析中，我们看到 addServiceOnPort 是 Proxier 的核心方法之一。下面是该方法的源码：

```
    func (proxier *Proxier) addServiceOnPort(service ServicePortName, protocol
api.Protocol, proxyPort int, timeout time.Duration) (*serviceInfo, error) {
        sock, err := newProxySocket(protocol, proxier.listenIP, proxyPort)
        if err != nil {
            return nil, err
        }
```

```
        _, portStr, err := net.SplitHostPort(sock.Addr().String())
        if err != nil {
            sock.Close()
            return nil, err
        }
        portNum, err := strconv.Atoi(portStr)
        if err != nil {
            sock.Close()
            return nil, err
        }
        si := &serviceInfo{
            proxyPort:          portNum,
            protocol:           protocol,
            socket:             sock,
            timeout:            timeout,
            sessionAffinityType: api.ServiceAffinityNone, // default
            stickyMaxAgeMinutes: 180,                     // TODO: paramaterize this
in the API.
        }
        proxier.setServiceInfo(service, si)

        glog.V(2).Infof("Proxying for service %q on %s port %d", service, protocol,
portNum)
        go func(service ServicePortName, proxier *Proxier) {
            defer util.HandleCrash()
            atomic.AddInt32(&proxier.numProxyLoops, 1)
            sock.ProxyLoop(service, si, proxier)
            atomic.AddInt32(&proxier.numProxyLoops, -1)
        }(service, proxier)

        return si, nil
    }
```

在上述代码中，先创建一个 ProxySocket，然后创建一个 serviceInfo 并添加到 Proxier 的 serviceMap 中，最后启动一个协程调用 ProxySocket 的 ProxyLoop 方法，使得 ProxySocket 进入 Listen 状态，开始接收并转发客户端请求。

kube-proxy 中的 ProxySocket 有两个实现，其中一个是 tcpProxySocket，另外一个是 udpProxySocket，二者的工作原理都一样，它们的工作流程就是为每个客户端 Socket 请求创建一个到 Service 的后端 Socket 连接，并且“打通”这两个 Socket，即把客户端 Socket 发来的数据“复制”到对应的后端 Socket 上，然后把后端 Socket 上服务响应的数据写入客户端 Socket 上去。

以 tcpProxySocket 为例，我们先看看它是如何完成 Service 后端连接创建过程的：

```
    func tryConnect(service ServicePortName, srcAddr net.Addr, protocol string,
proxier *Proxier) (out net.Conn, err error) {
```