```
        for _, retryTimeout := range endpointDialTimeout {
            endpoint, err := proxier.loadBalancer.NextEndpoint(service, srcAddr)
            if err != nil {
                glog.Errorf("Couldn't find an endpoint for %s: %v", service, err)
                return nil, err
            }
            glog.V(3).Infof("Mapped service %q to endpoint %s", service, endpoint)
            outConn, err := net.DialTimeout(protocol, endpoint, retryTimeout*time.
Second)
            if err != nil {
                if isTooManyFDsError(err) {
                    panic("Dial failed: " + err.Error())
                }
                glog.Errorf("Dial failed: %v", err)
                continue
            }
            return outConn, nil
        }
        return nil, fmt.Errorf("failed to connect to an endpoint. ")
    }
```

在上述方法里，首先调用 loadBalancer.NextEndpoint 方法获取服务的下一个可用 Endpoint 地址，然后调用标准网络库中的方法建立到此地址的连接，如果连接失败，则会重新尝试，间隔时间指数增加（参见 endpointDialTimeout 的值）。

在后端 Service 的连接建立以后，proxyTCP 方法就会启动两个协程，通过调用 Go 标准库 io 里的 Copy 方法把输入流的数据写入输出流，从而完成前后端连接的数据转发功能。此外，proxyTCP 方法会阻塞，直到前后端两个连接的数据流都关闭（或结束）才会返回。下面是其源码：

```
    func proxyTCP(in, out *net.TCPConn) {
        var wg sync.WaitGroup
        wg.Add(2)
        glog.V(4).Infof("Creating proxy between %v <-> %v <-> %v <-> %v",
            in.RemoteAddr(), in.LocalAddr(), out.LocalAddr(), out.RemoteAddr())
        go copyBytes("from backend", in, out, &wg)
        go copyBytes("to backend", out, in, &wg)
        wg.Wait()
        in.Close()
        out.Close()
    }
```

这里我们留一个问题，kube-proxy 会在当前节点上为每个 Service 都建立一个代理么？不管本节点上是否有该 Service 对应的 Pod？

6.6.3　设计总结

从之前的启动流程和代码分析来看，kube-proxy 的设计和实现还是比较精巧和紧凑的，它的流程只有一个：从 Kubernetes API Server 上同步 Service 及其 Endpoint 信息，为每个 Service 建立一个本地代理以完成具备负载均衡能力的服务转发功能。图 6.11 给出了 kube-proxy 的总体设计示意图，为了清晰地表明整个业务流程和数据传递方向，这里省去了一些非关键的结构体和对象。app.ProxyServer 创建了一个 config.SourceAPI 的结构体，用于拉取 Kubernetes API Server 上的 Service 与 Endpoint 配置信息，分别由 config.servicesReflector 与 config.endpointsReflector 这两个对象来实现，它们各自通过相应的 Kubernetes Client API 来拉取数据并且生成对应的 Update 信息放入 Channel 中，最终 Channel 中的 Service 数据到达 proxy.Proxier 上，proxy.Proxier 为每个 Service 建立一个 proxySocket 实现服务代理并且在 iptables 上创建相关的 NAT 规则，然后在 LoadBalancer 组件上开通该服务的负载均衡功能；而 Channel 中的 Endpoints 数据则被发送

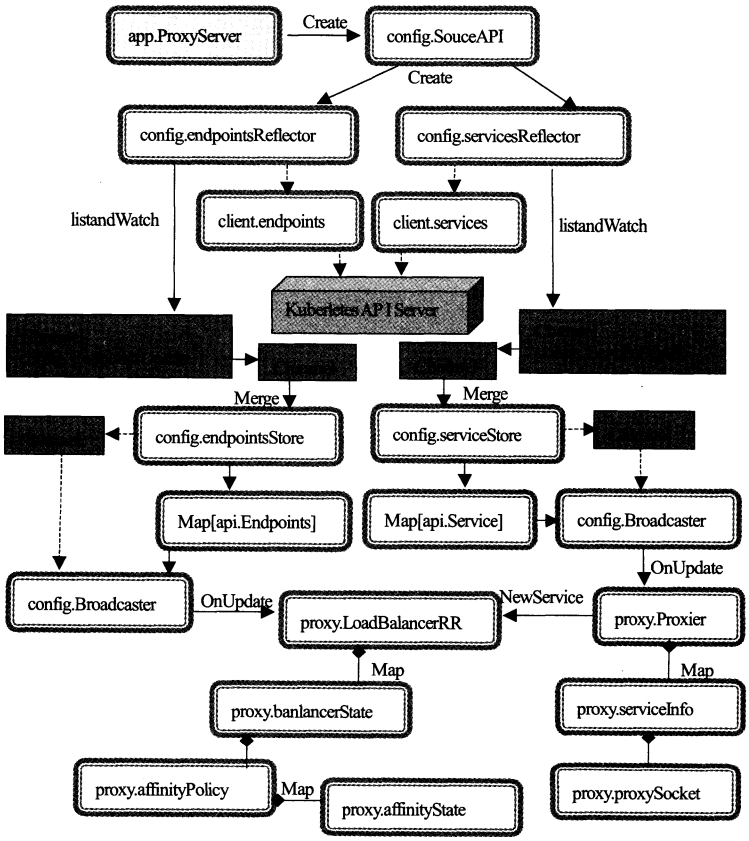

图 6.11　与 kubelet 总体相关的设计示意图

到 proxy.LoadBalancerRR 组件，用于给每个服务建立一个负载均衡的状态机，每个服务用 banlancerState 结构体来保存该服务可用的 Endpoint 地址及当前的会话状态 affinityPolicy，对于需要保存会话状态的服务，affinityPolicy 用一个 Map 来存储每个客户的会话状态 affinityState。

6.7 kubectl 进程源码分析

kubectl 与之前的 Kubernetes 进程不同，它不是一个后台运行的守护进程，而是 Kubernetes 提供的一个命令行工具（CLI），它提供了一组命令来操作 Kubernetes 集群。

kubectl 进程的入口类源码位置如下：

```
github/com/GoogleCloudPlatform/kubernetes/cmd/kubectl/kubectl.go
```

入口 main()函数的逻辑很简单：

```
func main() {
runtime.GOMAXPROCS(runtime.NumCPU())
cmd := cmd.NewKubectlCommand(cmdutil.NewFactory(nil), os.Stdin, os.Stdout,
os.Stderr)
if err := cmd.Execute(); err != nil {
os.Exit(1)
}
}
```

上述代码通过 NewKubectlCommand 方法创建了一个具体的 Command 命令并调用它的 Execute 方法执行，这是工厂模式结合命令模式的一个经典设计案例。从 NewKubectlCommand 的源码中可以看到，kubectl 的 CLI 命令框架使用了 GitHub 开源项目（https://github. com/spf13/cobra），下面是该框架中对 Command 的定义：

```
type Command struct {
   Use string // The one-line usage message.
   Short string // The short description shown in the 'help' output.
   Long string // The long message shown in the 'help <this-command>' output.
   Run func(cmd *Command, args []string) // Run runs the command.
}
```

实现一个具体 Command 就只要实现 Command 的 Run 函数即可，下面是其官方网页给出的一个 Echo 命令的例子：

```
var cmdEcho = &cobra.Command{
       Use:   "echo [string to echo] ",
       Short: "Echo anything to the screen",
       Long:  `echo is for echoing anything back.
       Echo works a lot like print, except it has a child command.
       `,
```

```
        Run: func(cmd *cobra.Command, args []string) {
            fmt.Println("Print: " + strings.Join(args, " "))
        },
    }
```

由于大多数 kubectl 的命令都需要访问 Kubernetes API Server，所以 kubectl 设计了一个类似命令的上下文环境的对象——util.Factory 供 Command 对象使用。

在接下来的几个章节中，我们对 kubectl 中的几个典型 Command 的源码逐一解读。

6.7.1　kubectl create 命令

kubectl create 命令通过调用 Kubernetes API Server 提供的 Rest API 来创建 Kubernetes 资源对象，例如 Pod、Service、RC 等，资源的描述信息来自-f 指定的文件或者来自命令行的输入流。下面是创建 create 命令的相关源码：

```
func NewCmdCreate(f *cmdutil.Factory, out io.Writer) *cobra.Command {
    var filenames util.StringList
    cmd := &cobra.Command{
        Use:     "create -f FILENAME",
        Short:   "Create a resource by filename or stdin",
        Long:    create_long,
        Example: create_example,
        Run: func(cmd *cobra.Command, args []string) {
            cmdutil.CheckErr(ValidateArgs(cmd, args))
            cmdutil.CheckErr(RunCreate(f, out, filenames))
        },
    }
    usage := "Filename, directory, or URL to file to use to create the resource"
    kubectl.AddJsonFilenameFlag(cmd, &filenames, usage)
    cmd.MarkFlagRequired("filename")
    return cmd
}
```

AddJsonFilenameFlag 方法限制 filename 参数（-f）的文件名后缀只能是 json、yaml 或者 yml 中的一种，并且将参数值填充到 filenames 这个 Set 集合中，随后被 Command 的 Run 函数中的 RunCreate 方法所引用，后者就是 kubectl create 命令的核心逻辑所在。

RunCreate 方法使用到了 resource.Builder 对象，它是 kubectl 中的一处复杂设计，采用了 Visitor 的设计模式，kubectl 的很多命令都用到了它。Builder 的目标是根据命令行输入的资源相关的参数，创建针对性的 Visitor 对象来获取对应的资源，最后遍历相关的所有 Visitor 对象，触发用户指定的 VisitorFun 回调函数来处理每个具体的资源，最终完成资源对象的业务处理逻辑。由于涉及的资源参数有各种情况，所以导致 Builder 的代码很复杂。以下是 Builder 所能操作的

各种资源参数：

- 通过输入流提供具体的资源描述；
- 通过本地文件内容或者 HTTP URL 的输出流来获取资源描述；
- 文件列表提供多个资源描述；
- 指定资源类型，通过查询 Kubernetes API Server 来获取相关类型的资源；
- 指定资源的 selector 条件如 cluster-service=true，查询 Kubernetes API Server 来获取相关的资源；
- 指定资源的 namespace 来查询符合条件的相关资源。

下面是 resource.Builder 的定义：

```
type Builder struct {
   mapper *Mapper
   errs []error
   paths  []Visitor
   stream bool
   dir    bool
   selector  labels.Selector
   selectAll bool
   resources []string
   namespace string
   names     []string
   resourceTuples []resourceTuple
   defaultNamespace bool
   requireNamespace bool
   flatten bool
   latest  bool
   requireObject bool
   singleResourceType bool
   continueOnError   bool
   schema validation.Schema
}
```

其实 Builder 很像一个 SQL 查询条件的生成器，里面包括了各种“查询”条件，在指定不同的查询条件时，会生成不同的 Visitor 接口来处理这些查询条件，最后遍历所有 Visitor，就得到最终的“查询结果”。Builder 返回的 Result 对象里也包括 Visitor 对象及可能的最终资源列表等信息，由于资源查询存在各种情况，所以 Result 也提供了多种方法，比如还包括了 Watch 资源变化的方法。

RunCreate 方法里先创建了一个 Builder，设置各种必要参数，然后调用 Builder 的 Do 方法，返回一个 Result，代码如下：

```
schema, err := f.Validator()
mapper, typer := f.Object()
   r := resource.NewBuilder(mapper, typer, f.ClientMapperForCommand()).
       Schema(schema).
       ContinueOnError().
       NamespaceParam(cmdNamespace).DefaultNamespace().
       FilenameParam(enforceNamespace, filenames...).
       Flatten().
       Do()
```

其中，schema 对象用来校验资源描述是否正确，比如有没有缺少字段或者属性的类型错误等；mapper 对象用来完成从资源描述信息到资源对象的转换，用来在 REST 调用过程中完成数据转换；FilenameParam 是这里唯一指定 Builder 的资源参数的方法，即把命令行传入的 filenames 参数作为资源参数；Flatten 方法则告诉 Builder，这里的资源对象其实是一个数组，需要 Builder 构造一个 FlattenListVisitor 来遍历 Visit 数组中的每个资源项目；Do 方法则返回一个 Rest 对象，里面包括与资源相关的 Visitor 对象。

下面是 NamespaceParam 方法的源码，主要逻辑为调用 Builder 的 Builder.Stdin、Builder.URL 或 Builder.Path 方法来处理不同类型的资源参数，这些方法会生成对应的 Visitor 对象并加入 Builder 的 Visitor 数组里（paths 属性）。

```
func (b *Builder) FilenameParam(enforceNamespace bool, paths ...string) *Builder {
for _, s := range paths {
   switch {
    case s == "-":
    b.Stdin()
    case strings.Index(s, "http:// ") == 0 || strings.Index(s, "https:// ") == 0:
     url, err := url.Parse(s)
    if err != nil {
b.errs = append(b.errs, fmt.Errorf("the URL passed to filename %q is not valid:
%v", s, err))
    continue
    }
    b.URL(url)
    default:
    b.Path(s)
    }
    }
if enforceNamespace {
  b.RequireNamespace()
    }
return b
}
```

不管是标准输入流、URL，还是文件目录或者文件本身，这里处理资源的 Visitor 都是

StreamVisitor 这个实现（FileVisitor 与 FileVisitorForSTDIN 是 StreamVisitor 的一个 Wrapper）。下面是 StreamVisitor 的 Visit 接口代码：

```
func (v *StreamVisitor) Visit(fn VisitorFunc) error {
    d := yaml.NewYAMLOrJSONDecoder(v.Reader, 4096)
    for {
        ext := runtime.RawExtension{}
        if err := d.Decode(&ext); err != nil {
            if err == io.EOF {
                return nil
            }
            return err
        }
        ext.RawJSON = bytes.TrimSpace(ext.RawJSON)
        if len(ext.RawJSON) == 0 || bytes.Equal(ext.RawJSON, []byte("null")) {
            continue
        }
        if err := ValidateSchema(ext.RawJSON, v.Schema); err != nil {
            return err
        }
        info, err := v.InfoForData(ext.RawJSON, v.Source)
        if err != nil {
            if v.IgnoreErrors {
                fmt.Fprintf(os.Stderr, "error: could not read an encoded object from
%s: %v\n", v.Source, err)
                glog.V(4).Infof("Unreadable: %s", string(ext.RawJSON))
                continue
            }
            return err
        }
        if err := fn(info); err != nil {
            return err
        }
    }
}
```

在上述代码中，首先从输入流中解析具体的资源对象，然后创建一个 Info 结构体进行包装（转换后的资源对象存储在 Info 的 Object 属性中），最后再用这个 Info 对象作为参数调用回调函数 VisitorFunc，从而完成整个逻辑流程。下面是 RunCreate 方法里调用 Builder 的 Visit 方法触发 Visitor 执行时的源码，可以看到这里的 VisitorFunc 所做的事情是通过 Rest Client 发起 Kubernetes API 调用，把资源对象写入资源注册表里：

```
err = r.Visit(func(info *resource.Info) error {
        data, err := info.Mapping.Codec.Encode(info.Object)
        if err != nil {
            return cmdutil.AddSourceToErr("creating", info.Source, err)
```

```
            }
            obj, err := resource.NewHelper(info.Client, info.Mapping).Create(info.
Namespace, true, data)
            if err != nil {
                return cmdutil.AddSourceToErr("creating", info.Source, err)
            }
            count++
            info.Refresh(obj, true)
            printObjectSpecificMessage(info.Object, out)
            fmt.Fprintf(out, "%s/%s\n", info.Mapping.Resource, info.Name)
            return nil
        })
```

6.7.2　rolling-update 命令

kubectl rolling-update 命令负责滚动更新（升级）RC（ReplicationController），下面是创建对应 Command 的源码：

```
    func NewCmdRollingUpdate(f *cmdutil.Factory, out io.Writer) *cobra.Command {
        cmd := &cobra.Command{
            Use: "rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] -image
=NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC) ",
            // rollingupdate is deprecated.
            Aliases: []string{"rollingupdate"},
            Short:   "Perform a rolling update of the given ReplicationController. ",
            Long:    rollingUpdate_long,
            Example: rollingUpdate_example,
            Run: func(cmd *cobra.Command, args []string) {
                err := RunRollingUpdate(f, out, cmd, args)
                cmdutil.CheckErr(err)
            },
        }
        cmd.Flags().String("update-period", updatePeriod, `Time to wait between
updating pods. Valid time units are "ns", "us" (or "µs"), "ms", "s", "m", "h".`)
```

此处省去一些命令参数添加的非关键代码：

```
        cmdutil.AddPrinterFlags(cmd)
        return cmd
    }
```

从上述代码中我们看到 rolling-update 命令的执行函数为 RunRollingUpdate，在分析这个函数之前，我们先了解下 rolling-update 执行过程中的一个关键逻辑。

rolling update 动作可能由于网络超时或者用户等得不耐烦等原因被中断，因此我们可能会重复执行一条 rolling-update 命令，目的只有一个，就是恢复之前的 rolling update 动作。为了实

现这个目的，rolling-update 程序在执行过程中会在当前 rolling-update 的 RC 上增加一个 Annotation 标签——kubectl.kubernetes.io/next-controller-id，标签的值就是下一个要执行的新 RC 的名字。此外，对于 Image 升级这种更新方式，还会在 RC 的 Selector 上（RC.Spec.Selector）贴一个名为 deploymentKey 的 Label，Label 的值是 RC 的内容进行 Hash 计算后的值，相当于签名，这样就能很方便地比较 RC 里的 Image 名字（以及其他信息）是否发生了变化。

RunRollingUpdate 执行逻辑的第 1 步：确定 New RC 对象及建立起 Old RC 到 New RC 的关联关系。下面我们以指定的 Image 参数进行 rolling update 的方式为例，看看代码是如何实现这段逻辑的。下面是相关源码：

```
    if len(image) != 0 {
            keepOldName = len(args) == 1
            newName := findNewName(args, oldRc)
            if newRc, err = kubectl.LoadExistingNextReplicationController(client,
cmdNamespace, newName); err != nil {
                return err
            }
            if newRc != nil {
                fmt.Fprintf(out, "Found existing update in progress (%s), resuming.\
n", newRc.Name)
            } else {
               newRc, err = kubectl.CreateNewControllerFromCurrentController(client,
cmdNamespace, oldName, newName, image, deploymentKey)
                if err != nil {
                    return err
                }
            }
            // Update the existing replication controller with pointers to the 'next'
controller
            // and adding the <deploymentKey> label if necessary to distinguish it from
the 'next' controller.
            oldHash, err := api.HashObject(oldRc, client.Codec)
            if err != nil {
                return err
            }
            oldRc, err = kubectl.UpdateExistingReplicationController(client, oldRc,
cmdNamespace, newRc.Name, deploymentKey, oldHash, out)
            if err != nil {
                return err
            }
        }
```

在代码里，findNewName 方法查询新 RC 的名字，如果在命令行参数中没有提供新 RC 的名字，则从 Old RC 中根据 kubectl.kubernetes.io/next-controller-id 这个 Annotation 标签找新 RC 的名字并返回，如果新 RC 存在则继续使用，否则调用 CreateNewControllerFromCurrentController

方法创建一个新 RC，在新 RC 的创建过程中设定 deploymentKey 的值为自己的 Hash 签名，方法源码如下：

```
    func CreateNewControllerFromCurrentController(c *client.Client, namespace, oldName,
newName, image, deploymentKey string) (*api.ReplicationController, error) {
        // load the old RC into the "new" RC
        newRc, err := c.ReplicationControllers(namespace).Get(oldName)
        if err != nil {
            return nil, err
        }
        if len(newRc.Spec.Template.Spec.Containers) > 1 {
            // TODO: support multi-container image update.
        return nil, goerrors.New("Image update is not supported for multi-container
pods")
        }
        if len(newRc.Spec.Template.Spec.Containers) == 0 {
            return nil, goerrors.New(fmt.Sprintf("Pod has no containers! (%v) ",
newRc))
        }
        newRc.Spec.Template.Spec.Containers[0].Image = image
        newHash, err := api.HashObject(newRc, c.Codec)
        if err != nil {
            return nil, err
        }
        if len(newName) == 0 {
            newName = fmt.Sprintf("%s-%s", newRc.Name, newHash)
        }
        newRc.Name = newName
        newRc.Spec.Selector[deploymentKey] = newHash
        newRc.Spec.Template.Labels[deploymentKey] = newHash
        // Clear resource version after hashing so that identical updates get different
hashes.
        newRc.ResourceVersion = ""
        return newRc, nil
    }
```

在 Image rolling update 的流程中确定新的 RC 以后，调用 UpdateExistingReplicationController 方法，将旧 RC 的 kubectl.kubernetes.io/next-controller-id 设置为新 RC 的名字，并且判断旧 RC 是否需要设置或更新 deploymentKey，具体代码如下：

```
    func UpdateExistingReplicationController(c client.Interface, oldRc *api.
ReplicationController, namespace, newName, deploymentKey, deploymentValue string,
out io.Writer) (*api.ReplicationController, error) {
        SetNextControllerAnnotation(oldRc, newName)
        if _, found := oldRc.Spec.Selector[deploymentKey]; !found {
            return AddDeploymentKeyToReplicationController(oldRc, c, deploymentKey,
deploymentValue, namespace, out)
```

```
        } else {
        // If we didn't need to update the controller for the deployment key, we still
need to write
            // the "next" controller.
            return c.ReplicationControllers(namespace).Update(oldRc)
        }
    }
```

通过上面的逻辑，新 RC 被确定并且旧 RC 到新 RC 的关联关系也被建立好了，接下来如果 dry-run 参数为 true，则仅仅打印新旧 RC 的信息然后返回。如果是正常的 rolling update 动作，则创建一个 kubectl.RollingUpdater 对象来执行具体任务，任务的参数则放在 kubectl.RollingUpdaterConfig 中，相关源码如下：

```
    updateCleanupPolicy := kubectl.DeleteRollingUpdateCleanupPolicy
        if keepOldName {
            updateCleanupPolicy = kubectl.RenameRollingUpdateCleanupPolicy
        }
        config := &kubectl.RollingUpdaterConfig{
            Out:           out,
            OldRc:         oldRc,
            NewRc:         newRc,
            UpdatePeriod:  period,
            Interval:      interval,
            Timeout:       timeout,
            CleanupPolicy: updateCleanupPolicy,
        }
```


其中 out 是输出流（屏幕输出）；UpdatePeriod 是执行 rolling update 动作的间隔时间；Interval 与 Timeout 组合使用，前者是每次拉取 polling controller 状态的间隔时间，而后者则是对应的（HTTP REST 调用）超时时间。CleanupPolicy 确定升级结束后的善后策略，比如 DeleteRollingUpdateCleanupPolicy 表示删除旧的 RC，而 RenameRollingUpdateCleanupPolicy 则表示保持 RC 的名字不变（改变新 RC 的名字）。

RollingUpdater 的 Update 方法是 rolling update 的核心，它以上述 config 对象作为参数，其核心流程是每次让新 RC 的 Pod 副本数量加 1，同时旧 RC 的 Pod 副本数量减 1，直到新 RC 的 Pod 副本数量达到预期值同时旧 RC 的 Pod 副本数量变为零为止，在这个过程中由于新旧 RC 的 Pod 副本数量一直在变动，所以需要一个地方记录最初不变的那个 Pod 副本数量，这里就是 RC 的 Annotation 标签——kubectl.kubernetes.io/desired-replicas。

下面这段源码就是“贴标签”的过程：

```
    fmt.Fprintf(out, "Creating %s\n", newName)
            if newRc.ObjectMeta.Annotations == nil {
                newRc.ObjectMeta.Annotations = map[string]string{}
            }
```

```
        newRc.ObjectMeta.Annotations[desiredReplicasAnnotation] = fmt.Sprintf
("%d", desired)
        newRc.ObjectMeta.Annotations[sourceIdAnnotation] = sourceId
        newRc.Spec.Replicas = 0
        newRc, err = r.c.CreateReplicationController(r.ns, n
```

下面这段源码便是“江山代有才人出，一代新人换旧人”的生动画面：

```
    for newRc.Spec.Replicas < desired && oldRc.Spec.Replicas != 0 {
        newRc.Spec.Replicas += 1
        oldRc.Spec.Replicas -= 1
        fmt.Printf("At beginning of loop: %s replicas: %d, %s replicas: %d\n",
            oldName, oldRc.Spec.Replicas,
            newName, newRc.Spec.Replicas)
        fmt.Fprintf(out, "Updating %s replicas: %d, %s replicas: %d\n",
            oldName, oldRc.Spec.Replicas,
            newName, newRc.Spec.Replicas)
        newRc, err = r.scaleAndWait(newRc, retry, waitForReplicas)
        if err != nil {
            return err
        }
        time.Sleep(updatePeriod)
        oldRc, err = r.scaleAndWait(oldRc, retry, waitForReplicas)
        if err != nil {
            return err
        }
        fmt.Printf("At end of loop: %s replicas: %d, %s replicas: %d\n",
            oldName, oldRc.Spec.Replicas,
            newName, newRc.Spec.Replicas)
    }
    // delete remaining replicas on oldRc
    if oldRc.Spec.Replicas != 0 {
        fmt.Fprintf(out, "Stopping %s replicas: %d -> %d\n",
            oldName, oldRc.Spec.Replicas, 0)
        oldRc.Spec.Replicas = 0
        oldRc, err = r.scaleAndWait(oldRc, retry, waitForReplicas)
        if err != nil {
            return err
        }
    }
    // add remaining replicas on newRc
    if newRc.Spec.Replicas != desired {
        fmt.Fprintf(out, "Scaling %s replicas: %d -> %d\n",
            newName, newRc.Spec.Replicas, desired)
        newRc.Spec.Replicas = desired
        newRc, err = r.scaleAndWait(newRc, retry, waitForReplicas)
        if err != nil {
            return err
```

```
            }
        }
```

上述方法里的 scaleAndWait 方法调用了 kubectl.ReplicationControllerScaler 的 Scale 方法，Scale 方法先通过 Rest API 调用 Kubernetes API Server 更新 RC 的 Pod 副本数量，然后循环拉取 RC 信息，直到超时或者 RC 同步状态完成。下面是判断 RC 同步状态是否完成的函数，来自 client 包（pkg/client/conditions.go）。

```
    func ControllerHasDesiredReplicas(c Interface, controller *api.ReplicationController)
wait.ConditionFunc {
        desiredGeneration := controller.Generation
        return func() (bool, error) {
            ctrl, err := c.ReplicationControllers(controller.Namespace).Get
(controller.Name)
            if err != nil {
                return false, err
            }
            return ctrl.Status.ObservedGeneration >= desiredGeneration &&
ctrl.Status.Replicas == ctrl.Spec.Replicas, nil
        }
    }
```

rolling-update 是 kubectl 所有命令中最为复杂的一个，从它的功能和流程来看，完全可以被当作一个 Job 并放到 kube-controller-manager 上实现，客户端仅仅发起 Job 的创建及 Job 状态查看等命令即可，未来 Kubernetes 的版本是否会这样重构，我们拭目以待。