越大），他的评分影响力就越大：

```
    combinedScores := map[string]int{}
        for _, priorityConfig := range priorityConfigs {
            weight := priorityConfig.Weight
            // skip the priority function if the weight is specified as 0
            if weight == 0 {
                continue
            }
            priorityFunc := priorityConfig.Function
            prioritizedList, err := priorityFunc(pod, podLister, minionLister)
            if err != nil {
                return algorithm.HostPriorityList{}, err
            }
            for _, hostEntry := range prioritizedList {
                combinedScores[hostEntry.Host] += hostEntry.Score * weight
            }
        }
        for host, score := range combinedScores {
            glog.V(10).Infof("Host %s Score %d", host, score)
            result = append(result, algorithm.HostPriority{Host: host, Score: score})
        }
        return result, nil
```

接下来，我们看看系统初始化加载的默认的 Predicate 与 Priorities 有哪些，通过追踪代码，我们发现默认加载的代码位于 plugin/pkg/scheduler/algorithmprovider/default/default.go 的 init 函数里：

```
    func init() {
      factory.RegisterAlgorithmProvider(factory.DefaultProvider, defaultPredicates(),
defaultPriorities())
       // EqualPriority is a prioritizer function that gives an equal weight of one
to all minions
       // Register the priority function so that its available
       // but do not include it as part of the default priorities
       factory.RegisterPriorityFunction("EqualPriority", scheduler. EqualPriority, 1)
    }
```

跟踪进去后，我们看到系统默认加载的 predicates 有如下几种：

- PodFitsResources；
- MatchNodeSelector；
- HostName。

而默认加载的 priorities 则有如下几种：

- LeastRequestedPriority；
- BalancedResourceAllocation；

◎　ServiceSpreadingPriority。

从上述这些信息来看，Kubernetes 默认的调度指导原则是尽量均匀分布 Node 到不同的 Node 上，并且确保各个 Node 上的资源利用率基本保持一致，也就是说如果你有 100 台机器，则可能每个机器都被调度到，而不是只有其中的 20%被调度到，哪怕每台机器都只利用了不到 10%的资源，这不正是所谓的“韩信点兵，多多益善”么？

接下来我们以服务亲和性这个默认没有加载的 Predicate 为例，看看 Kubernetes 是如何通过 Policy 文件注册加载它的。下面是我们定义的一个 Policy 文件：

```
{
   "kind" : "Policy",
   "version" : "v1",
   "predicates" : [
      ......
      {"name" : "RegionZoneAffinity", "argument" : {"serviceAffinity" :
{"labels" : ["region", "zone"]}}}
   ],
   "priorities" : [
      ......
      {"name" : "RackSpread", "weight" : 1, "argument" : {"serviceAnti
Affinity" : {"label" : "rack"}}}
   ]
}
```

首先，这个文件被映射成 api.Policy 对象（plugin/pkg/scheduler/api/types.go）。下面是其结构体定义：

```
type Policy struct {
   api.TypeMeta `json: ",inline"`
   // Holds the information to configure the fit predicate functions
   Predicates []PredicatePolicy `json: "predicates"`
   // Holds the information to configure the priority functions
   Priorities []PriorityPolicy `json: "priorities"`
}
```

我们看到 policy 文件中的 predicates 部分被映射为 PredicatePolicy 数组：

```
type PredicatePolicy struct {
   Name string `json: "name"`
   Argument *PredicateArgument 'json: "argument"'
}
```

而 PredicateArgument 的定义如下，包括服务亲和性的相关属性 ServiceAffinity：

```
type PredicateArgument struct {
   ServiceAffinity *ServiceAffinity `json: "serviceAffinity"`
   LabelsPresence *LabelsPresence `json: "labelsPresence"`
}
```

策略文件被映射为 api.Policy 对象后，PredicatePolicy 部分的处理逻辑则交给下面的函数进行处理（plugin/pkg/scheduler/factory/plugin.go）：

```
    func RegisterCustomFitPredicate(policy schedulerapi.PredicatePolicy) string {
        var predicateFactory FitPredicateFactory
        var ok bool
        validatePredicateOrDie(policy)
        // generate the predicate function, if a custom type is requested
        if policy.Argument != nil {
            if policy.Argument.ServiceAffinity != nil {
                predicateFactory = func(args PluginFactoryArgs) algorithm.
FitPredicate {
                    return predicates.NewServiceAffinityPredicate(
                        args.PodLister,
                        args.ServiceLister,
                        args.NodeInfo,
                        policy.Argument.ServiceAffinity.Labels,
                    )
                }
            } else if policy.Argument.LabelsPresence != nil {
                predicateFactory = func(args PluginFactoryArgs) algorithm.
FitPredicate {
                    return predicates.NewNodeLabelPredicate(
                        args.NodeInfo,
                        policy.Argument.LabelsPresence.Labels,
                        policy.Argument.LabelsPresence.Presence,
                    )
                }
            }
```

在上面的代码中，当 ServiceAffinity 属性不空时，就会调用 predicates.NewServiceAffinityPredicate 方法来创建一个处理服务亲和性的 FitPredicate，随后被加载到全局的 predicateFactory 中生效。

最后，genericScheduler.Schedule 方法才是真正实现 Pod 调度的方法，我们看看这段完整代码：

```
    func (g *genericScheduler) Schedule(pod *api.Pod, minionLister algorithm.
MinionLister) (string, error) {
        minions, err := minionLister.List()
        if err != nil {
            return "", err
        }
        if len(minions.Items) == 0 {
            return "", ErrNoNodesAvailable
        }

        filteredNodes, failedPredicateMap, err := findNodesThatFit(pod, g.pods,
g.predicates, minions)
        if err != nil {
```

```
            return "", err
        }

        priorityList, err := PrioritizeNodes(pod, g.pods, g.prioritizers, algorithm.
FakeMinionLister(filteredNodes))
        if err != nil {
            return "", err
        }
        if len(priorityList) == 0 {
            return "", &FitError{
                Pod:              pod,
                FailedPredicates: failedPredicateMap,
            }
        }

        return g.selectHost(priorityList)
    }
```

这段代码已经简单得不能再简单了，因为该干的活都已经被 predicates 与 priorities 干完了！架构之美，就在于程序逻辑分解得恰到好处，每个组件各司其职，从而化繁为简，使得主体流程清晰直观，犹如行云流水，一气呵成。

向谷歌大神们致敬！

### 6.4.3　设计总结

与之前的 Kubernetes API Server 和 Kubernetes Controller Mangager 对比，Kubernetes Scheduler Server 的设计和代码显得更为“精妙”。项目中引入 ratelimit 组件来解决 Pod 调度的流控问题的做法，既大大简化了代码量，又体现了大神们的气度。

Kubernetes Scheduler Server 的一个关键设计目标是“插件化”，以方便 Cloud Provider 或者个人用户根据自己的需求进行定制，本节我们围绕其中最为关键的“FitPredicate 与 PriorityFunction”对其设计做一个总结。如图 6.7 所示，在 plugin.go 中采用了全局变量的 Map 变量记录了系统当前注册的 FitPredicate 与 PriorityFunction，其中 fitPredicateMap 和 priorityFunctionMap 分别存放 FitPredicateFactory 与 PriorityConfigFactory（包含了 PriorityFunctionFactory 的一个引用）中。可以看出，这里的设计采用了标准的工厂模式，factory.PluginFactoryArgs 这个数据结构可以认为是一个上下文环境变量，它提供给 PluginFactory 必要的数据访问接口，比如获取一个 Node 的详细信息并获取一个 Pod 上的所有 Service 信息等，这些接口可以被某些具体的 FitPredicate 或 PriorityFunction 使用，以实现特定的功能，如图 6.7 所示的 predicates.PodFitsPods 和 priorities.LeastRequestedPriority 就分别使用了上述接口。

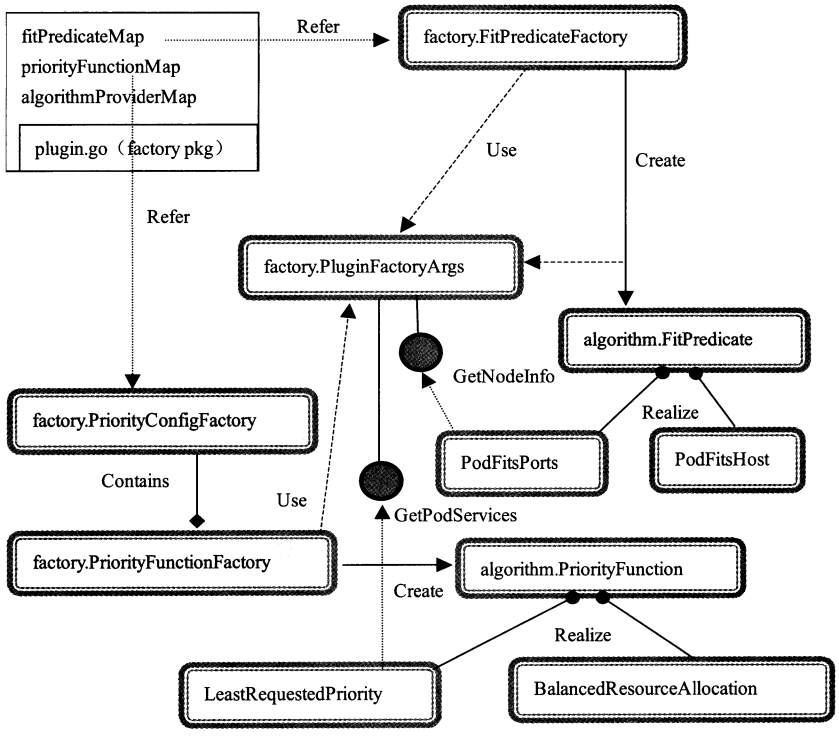


图 6.7　Kubernetes Scheduler Server 调度策略相关设计示意图

我们注意到 PluginFactoryArgs 的接口都是 Kubernetes 的资源访问接口，那么问题就来了，为何不直接用 Kubernetes RestClient API 访问呢？一个主要的原因是如果这样做，则增加了插件开发者开发和调测的难度，因为开发者需要再去学习和掌握 RestClient；另外一个原因是效率的问题，如果大家都采用框架提供的“标准方法”查询资源，那么框架可以实现很多优化，比较容易缓存；最后一个原因则与之前我们分析的“Assumed Pod”有关，即查询当前已经调度过的 Pod 列表是有其特殊性的，PluginFactoryArgs 中的 PodLister 方法就是引用了 ConfigFactory 的 PodLister。

algorithmProviderMap 这个全局变量则保存了一组命名的调度策略配置文件（Algorithm ProviderConfig），其实就是一组 FitPredicate 与 PriorityFunction 的集合，其定义如下：

```
type AlgorithmProviderConfig struct {
   FitPredicateKeys     util.StringSet
   PriorityFunctionKeys util.StringSet
}
```

它的作用是预配置和自定义调度规则，Kubernetes Scheduler Server 默认加载了一个名为“DefaultProvider”的调度策略配置，通过定义和加载不同的调度规则配置文件，我们可以改变默认的调度策略，比如我们可以定义两组规则文件：其中一个命名为“function_test_cfg”，面向

功能测试，调度原则是尽量在最少的机器上调度 Pod 以节省资源；另外一个则命名为 performance_test_cfg”，面向性能测试，调度原则是尽可能使用更多的机器，以测试系统性能。

顺便提一下，笔者认为在 Kubernetes Scheduler Server 中关于 PredicateArgument/Priority Argument 的设计并不好，这里没有将 Predicate 的属性通用化，比如采用 key-value 这种模式，因此导致 Policy 文件格式与 Predicate/Priority 关联之间的强耦合性，增加了代码理解的困难性，之前分析的 Policy 文件中服务亲和性的 Predicate 的加载逻辑即反映了这个问题，笔者深信，未来版本中大神们会认真考虑重构问题。

至此，Master 节点上的进程的源码都已经分析完毕，我们发现这些进程所做的事情，归根到底就是两件事：Pod 调度+智能纠错，这也是为什么这些进程所在的节点被称为“Master”，因为它们高高在上，运筹帷幄。虽然“Master”从不深入底层微服私访，但也的确鞠躬尽瘁、日理万机，计算机的世界果然比我们人类的世界要单纯、高效很多，真心希望人工智能的发展不会让它们的世界也变得扑朔迷离。

## 6.5　kubelet 进程源码分析

kubelet 是运行在 Minion 节点上的重要守护进程，是工作在一线的重要“工人”，它才是负责“实例化”和“启动”一个具体的 Pod 的幕后主导，并且掌管着本节点上的 Pod 和容器的全生命周期过程，定时向 Master 汇报工作情况。此外，kubelet 进程也是一个“Server”进程，它默认监听 10250 端口，接收并执行远程（Master）发来的指令。

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。

### 6.5.1　进程启动过程

kubelet 进程的入口类源码位置如下：

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

入口 main()函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewKubeletServer()
    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)
    }
}
```

我们已经是第 4 次“遇见”这样的代码风格了，代码的颜值匹配度高达 99%，这至少说明一点：谷歌在源码一致性方面做得很好，*N* 多人写的代码看起来就好像出自一个人之手。我们先来看看 KubeletServer 这个结构体所包括的属性吧，这些属性可以分为以下几组。

1）基本配置

- KubeConfig：kubelet 默认配置文件路径。
- Address、Port、ReadOnlyPort、CadvisorPort、HealthzPort、HealthzBindAddress：为 kubelet 绑定监听的地址，包括自身 Server 的地址，Cadvisor 绑定的地址，以及自身健康检查服务的绑定地址等。
- RootDirectory、CertDirectory：kubelet 默认的工作目录（/var/lib/kubelet），用于存放配置及 VM 卷等数据，CertDirectory 用于存放证书目录。

2）管理 Pod 和容器相关的参数

- PodInfraContainerImage：Pod 的 infra 容器的镜像名称，谷歌被屏蔽的时候可以换成自己的私有仓库的镜像名。
- CgroupRoot：可选项，创建 Pod 的时候所使用的顶层的 cgroop 名字（Root Cgroup）。
- ContainerRuntime、DockerDaemonContainer、SystemContainer：这三个参数分别表示选择什么容器技术（Docker 或者 RKT）、Docker Daemon 容器的名字及可选的系统资源容器名称，用来将所有非 kernel 的、不在容器中的进程放入此容器中。

3）同步和自动运维相关的参数

- SyncFrequency、FileCheckFrequency、HTTPCheckFrequency：Pod 容器同步周期、当前运行的容器实例分别与 Kubernetes 注册表中的信息、本地的 Pod 定义文件及以 HTTP 方式提供信息的数据源进行对比同步。
- RegistryPullQPS、RegistryBurst：从注册表拉取待创建的 Pod 列表时的流控参数。
- NodeStatusUpdateFrequency：kubelet 多久汇报一次当前 Node 的状态。
- ImageGCHighThresholdPercent、ImageGCLowThresholdPercent、LowDiskSpace ThresholdMB：分别是 Image 镜像占用磁盘空间的高低水位阈值及本机磁盘最小空闲容量，当可用容量低于这个容量时，所有新 Pod 的创建请求会被拒绝。

- MaxContainerCount、MaxPerPodContainerCount：分别是 maximum-dead-containers 与 maximum-dead-containers-per-container，表示保留多少个死亡容器的实例在磁盘上，因为每个实例都会占用一定的磁盘，所以需要控制，默认是 MaxContainerCount 为 100，MaxPerPodContainerCount 为 2，即每个容器保留最多两个死亡实例，每个 Node 保留最多 100 个死亡实例。

只要分析一下上述 KubeletServer 结构体的关键属性，我们就可以得到这样一个推论：kubelet 进程的“工作量”还是很饱满的，一点都不比 Master 上的 API Server、Controll Manager、Scheduer 做得少。

在继续下面的代码分析之前，我们先要理解这里的一个重要概念“Pod Source”，它是 kubelet 用于获取 Pod 定义和描述信息的一个“数据源”，kubelet 进程查询并监听 Pod Source 来获取属于自己所在节点的 Pod 列表，当前支持三种 Pod Source 类型。

- Config File：本地配置文件作为 Pod 数据源。
- Http URL：Pod 数据源的内容通过一个 HTTP URL 方式获取。
- Kubernetes API Server：默认方式，从 API Server 获取 Pod 数据源。

进程根据启动参数创建了 KubeletServer 以后，调用 KubeletServer 的 run 方法，进入启动流程，在流程的一开始首先设置了自身进程的 oom_adj 参数（默认为-900），这是利用了 Linux 的 OOM 机制，当系统发生 OOM 时，oom_adj 的值越小，越不容易被系统 Kill 掉。

```
if err := util.ApplyOomScoreAdj(0, s.OOMScoreAdj); err != nil {
    glog.Warning(err)
}
```

为什么在之前的 Master 节点进程上都没有见到这个调用，而在 kubelet 进程上却看到这段逻辑？答案很简单，因为 Master 节点不运行 Pod 和容器，主机资源通常是稳定和宽裕的，而 Minion 节点由于需要运行大量的 Pod 和容器，因此容易产生 OOM 问题，所以这里要确保“守护者”不会因此而被系统 Kill 掉。

由于 kubelet 会跟 API Server 打交道，所以接下来创建了一个 Rest Client 对象来访问 API Server。随后，启动进程构造了 cAdvisor 来监控本地的 Docker 容器，cAdvisor 具体的创建代码则位于 pkg/kubelet/cadvisor/cadvisor_linux.go 里，引用了 github.com/google/cadvisor 这个同样属于谷歌开源的项目。

接着，初始化 CloudProvider，这是因为如果 Kubernetes 运行在某个运营商的 Cloud 环境中，则很多环境和资源都需要从 CloudProvider 中获取，比如在创建 Pod 的过程中可能需要知道某个 Node 的真实主机名。

虽然容器可以绑定宿主机的网络空间，但若不当使用会导致系统安全漏洞，所以

KubeletServer 中的 HostNetworkSources 的属性用来控制哪些 Pod 允许绑定宿主机的网络空间，默认是都禁止绑定。举例说明，比如设置 HostNetworkSources=api,http，则表明当一个 Pod 的定义源来自 Kubernetes API Server 或者某个 HTTP URL 时，则允许此 Pod 绑定到宿主机的网络空间。下面这行代码即上述处理逻辑中的一小部分：

```
    hostNetworkSources, err :=
kubelet.GetValidatedSources(strings.Split(s.HostNetworkSources, ","))
```

接下来加载数字证书，如果没有提供证书和私钥，则默认创建一个自签名的 X509 证书并保存到本地。下一步，创建一个 Mounter 对象，用来实现容器的文件系统挂载功能。

接下来的这段代码根据指定了 DockerExecHandlerName 参数的值，确定 dockerExecHandler 是采用 Docker 的 exec 命令还是 nsenter 来实现，默认采用了 Docker 的 exec 这种本地方式，Docker 从 1.3 版本开始提供了 exec 指令，为进入容器内部提供了更好的手段。

```
    var dockerExecHandler dockertools.ExecHandler
       switch s.DockerExecHandlerName {
       case "native":
          dockerExecHandler = &dockertools.NativeExecHandler{}
       case "nsenter":
          dockerExecHandler = &dockertools.NsenterExecHandler{}
       default:
          log.Warningf("Unknown Docker exec handler %q; defaulting to native",
s.DockerExecHandlerName)
          dockerExecHandler = &dockertools.NativeExecHandler{}
       }
```

运行至此，程序构造了一个 KubeletConfig 结构体，90%的变量与之前的 KubeletServer 一样，这让代码长度增加了 20 多行！定睛一看，源码上有 TODO 注释：“它应该可能被合并到 KubeletServer 里……”，目测注释是另外一个大神添加的，这让笔者陷入了深深的思考：难道谷歌的绩效考评系统中也有恶俗的代码行数考核指标？

KubeletConfig 创建好以后作为参数调用 RunKubelet (&kcfg, nil)方法，程序运行到这里，才真正进入流程的核心步骤。下面这段代码表明 kubelet 会把自己的事件通知 API Server：

```
    eventBroadcaster := record.NewBroadcaster()
        kcfg.Recorder = eventBroadcaster.NewRecorder(api.EventSource{Component:
"kubelet", Host: kcfg.NodeName})
        eventBroadcaster.StartLogging(glog.V(3).Infof)
        if kcfg.KubeClient != nil {
            glog.V(4).Infof("Sending events to api server. ")
            eventBroadcaster.StartRecordingToSink(kcfg.KubeClient.Events(""))
        } else {
            glog.Warning("No api server defined - no events will be sent to API server.
")
        }
```

接下来，启动进程进入关键函数 createAndInitKubelet 中，这里首先创建一个 PodConfig 对象，并根据启动参数中 Pod Source 参数是否提供，来创建相应类型的 Pod Source 对象，这些 PodSource 在各种协程中运行，拉取 Pod 信息并汇总输出到同一个 Pod Channel 中等待 kubelet 处理。创建 PodConfig 的具体代码如下：

```
    func makePodSourceConfig(kc *KubeletConfig) *config.PodConfig {
        // source of all configuration
        cfg := config.NewPodConfig(config.PodConfigNotificationSnapshotAndUpdates,
kc.Recorder)

        // define file config source
        if kc.ConfigFile != "" {
            glog.Infof("Adding manifest file: %v", kc.ConfigFile)
            config.NewSourceFile(kc.ConfigFile, kc.NodeName, kc.FileCheckFrequency,
cfg.Channel(kubelet.FileSource))
        }

        // define url config source
        if kc.ManifestURL != "" {
            glog.Infof("Adding manifest url: %v", kc.ManifestURL)
            config.NewSourceURL(kc.ManifestURL, kc.NodeName, kc.HTTPCheckFrequency,
cfg.Channel(kubelet.HTTPSource))
        }
        if kc.KubeClient != nil {
            glog.Infof("Watching apiserver")
            config.NewSourceApiserver(kc.KubeClient, kc.NodeName, cfg.Channel
(kubelet.ApiserverSource))
        }
        return cfg
    }
```

然后，创建一个 kubelet 并宣告它的诞生：

```
    k, err = kubelet.NewMainKubelet(….)
    k.BirthCry()
```

接着，触发 kubelet 开启垃圾回收协程以清理无用的容器和镜像，释放磁盘空间，下面是其代码片段：

```
    // Starts garbage collection threads.
    func (kl *Kubelet) StartGarbageCollection() {
        go util.Forever(func() {
            if err := kl.containerGC.GarbageCollect(); err != nil {
                glog.Errorf("Container garbage collection failed: %v", err)
            }
        }, time.Minute)

        go util.Forever(func() {
```

```
            if err := kl.imageManager.GarbageCollect(); err != nil {
                glog.Errorf("Image garbage collection failed: %v", err)
            }
        }, 5*time.Minute)
    }
```

createAndInitKubelet 方法创建 kubelet 实例以后，返回到 RunKubelet 方法里，接下来调用 startKubelet 方法，此方法首先启动一个协程，让 kubelet 处理来自 PodSource 的 Pod Update 消息，然后启动 Kubelet Server，下面是具体代码：

```
func startKubelet(k KubeletBootstrap, podCfg *config.PodConfig, kc *KubeletConfig) {
    // start the kubelet
    go util.Forever(func() { k.Run(podCfg.Updates()) }, 0)

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

至此，kubelet 进程启动完毕。

### 6.5.2 关键代码分析

6.5.1 节里，我们分析了 kubelet 进程的启动流程，大致明白了 kubelet 的核心工作流程就是不断从 Pod Source 中获取与本节点相关的 Pod，然后开始“加工处理”，所以，我们先来分析 Pod Source 部分的代码。前面我们提到，kubelet 可以同时支持三类 Pod Source，为了能够将不同的 Pod Source“汇聚”到一起统一处理，谷歌特地设计了 PodConfig 这个对象，其代码如下：

```
type PodConfig struct {
   pods *podStorage
   mux  *config.Mux

   // the channel of denormalized changes passed to listeners
   updates chan kubelet.PodUpdate

   // contains the list of all configured sources
```

```
    sourcesLock sync.Mutex
    sources     util.StringSet
}
```

其中，sources 属性包括了当前加载的所有 Pod Source 类型，sourcesLock 是 source 的排他锁，在新增 Pod Source 的方法里使用它来避免共享冲突。

当 Pod 发生变动时，例如 Pod 创建、删除或者更新，相关的 Pod Source 就会产生对应的 PodUpdate 事件并推送到 Channel 上。为了能够统一处理来自多个 Source 的 Channel，谷歌设计了 config.Mux 这个“聚合器”，它负责监听多路 Channel，当接收到 Channel 发来的事件以后，交给 Merger 对象进行统一处理，Merger 对象最终把多路 Channel 发来的事件合并写入 updates 这个汇聚 Channel 里等待处理。

下面是 config.Mux 的结构体定义，其属性 sources 为一个 Channel Map，key 是对应的 Pod Source 的类型：

```
type Mux struct {
    // Invoked when an update is sent to a source.
    merger Merger
    // Sources and their lock.
    sourceLock sync.RWMutex
    // Maps source names to channels
    sources map[string]chan interface{}
}
```

我们继续深入分析 config.Mux 的工作过程，前面提到，kubelet 在启动过程中在 makePod SourceConfig 方法里创建了一个 PodConfig 对象，并且根据启动参数来决定要加载哪些类型的 Pod Source，在这个过程中调用了下述方法来创建一个对应的 Channel：

```
func (c *PodConfig) Channel(source string) chan<- interface{} {
    c.sourcesLock.Lock()
    defer c.sourcesLock.Unlock()
    c.sources.Insert(source)
    return c.mux.Channel(source)
}
```

而 Channel 具体的创建过程则在 config.Mux 里，Channel 创建完成后被加入 config.Mux 的 sources 里并且启动一个协程开始监听消息，代码如下：

```
func (m *Mux) Channel(source string) chan interface{} {
    if len(source) == 0 {
        panic("Channel given an empty name")
    }
    m.sourceLock.Lock()
    defer m.sourceLock.Unlock()
    channel, exists := m.sources[source]
    if exists {
```

```
            return channel
        }
        newChannel := make(chan interface{})
        m.sources[source] = newChannel
        go util.Forever(func() { m.listen(source, newChannel) }, 0)
        return newChannel
    }
```

config.Mux 的上述 listen 方法很简单，就是监听新创建的 Channel，一旦发现 Channel 上有数据就交给 Merger 进行处理：

```
func (m *Mux) listen(source string, listenChannel <-chan interface{}) {
    for update := range listenChannel {
        m.merger.Merge(source, update)
    }
}
```

我们先来看看 Pod Source 是如何发送 PodUpdate 事件到自己所在的 Channel 上的，在 6.5.1 节中我们所见到的下面这段代码创建了一个 Config File 类型的 Pod Source：

```
// define file config source
    if kc.ConfigFile != "" {
        glog.Infof("Adding manifest file: %v", kc.ConfigFile)
        config.NewSourceFile(kc.ConfigFile, kc.NodeName, kc.FileCheckFrequency,
cfg.Channel(kubelet.FileSource))
    }
```

在 NewSourceFile 方法里启动了一个协程，每隔指定的时间（kc.FileCheckFrequency）就执行一次 SourceFile 的 run 方法，在 run 方法里所调用的主体逻辑是下面的函数：

```
func (s *sourceFile) extractFromPath() error {
    path := s.path
    statInfo, err := os.Stat(path)
    if err != nil {
        if !os.IsNotExist(err) {
            return err
        }
        // Emit an update with an empty PodList to allow FileSource to be marked
as seen
        s.updates <- kubelet.PodUpdate{[]*api.Pod{}, kubelet.SET, kubelet.
FileSource}
        return fmt.Errorf("path does not exist, ignoring")
    }

    switch {
    case statInfo.Mode().IsDir():
        pods, err := s.extractFromDir(path)
        if err != nil {
            return err
```

```
            }
            s.updates <- kubelet.PodUpdate{pods, kubelet.SET, kubelet.FileSource}

        case statInfo.Mode().IsRegular():
            pod, err := s.extractFromFile(path)
            if err != nil {
                return err
            }
            s.updates <-kubelet.PodUpdate{[]*api.Pod{pod},kubelet.SET, kubelet.
FileSource}

        default:
            return fmt.Errorf("path is not a directory or file")
        }

        return nil
    }
```

看一眼上面的代码，我们就大致明白了 Config File 类型的 Pod Source 是如何工作的：它从指定的目录中加载多个 Pod 定义文件并转换为 Pod 列表或者加载单个 Pod 定义文件并转换为单个 Pod，然后生成对应的全量类型的 PodUpdate 事件并写入 Channel 中去。这里笔者也发现了代码命名的一个疏漏之处，SourceFile 的 updates 属性其实应该被命名为 update。其他两种 Pod Source 类型的代码解析就不在这里提及了。

接下来我们分析 Merger 对象，PodConfig 里的 Merger 对象其实是一个 config.podStorage 实例，它同时是 PodConfig 的 pods 属性的一个引用。podStorage 的源码位于 pkg/kubelet/config/config.go 里，其定义如下：

```
type podStorage struct {
    podLock sync.RWMutex
    // map of source name to pod name to pod reference
    pods map[string]map[string]*api.Pod
    mode PodConfigNotificationMode
    // ensures that updates are delivered in strict order
    // on the updates channel
    updateLock sync.Mutex
    updates    chan<- kubelet.PodUpdate
    // contains the set of all sources that have sent at least one SET
    sourcesSeenLock sync.Mutex
    sourcesSeen    util.StringSet
    // the EventRecorder to use
    recorder record.EventRecorder
}
```

我们看到 podStorage 的关键属性解释如下。

（1）pods：类型是 Map，存放每个 Pod Source 上拉过来的 Pod 数据，是 podStorage 当前保存“全量 Pod”的地方。

（2）updates：它就是 PodConfig 里的 updates 属性的一个引用。

（3）mode：表明 podStorage 的 Pod 事件通知模式，有以下几种。

- PodConfigNotificationSnapshot：全量快照通知模式。
- PodConfigNotificationSnapshotAndUpdates：全量快照+更新 Pod 通知模式（代码中创建 podStorage 实例时采用的模式）。
- PodConfigNotificationIncremental：增量通知模式。

podStorage 实现的 Merge 接口的源码如下：

```
func (s *podStorage) Merge(source string, change interface{}) error {
    s.updateLock.Lock()
    defer s.updateLock.Unlock()
    adds, updates, deletes := s.merge(source, change)
    // deliver update notifications
    switch s.mode {
    case PodConfigNotificationSnapshotAndUpdates:
        if len(updates.Pods) > 0 {
            s.updates <- *updates
        }
        if len(deletes.Pods) > 0 || len(adds.Pods) > 0 {
           s.updates<- kubelet.PodUpdate{s.MergedState().([]*api.Pod), kubelet.
SET, source}
        }
     //省略无关的 Case 逻辑
    }
    return nil
}
```

在上述 Merge 过程中，先调用内部函数 merge，将 Pod Soucre 的 Channel 上发来的 PodUpdate 事件分解为对应的新增、更新及删除等三类 PodUpdate 事件，然后判断是否有更新事件，如果有，则直接写入汇总的 Channel 中（podStorage.updates），然后调用 MergedState 函数复制一份 podStorage 的当前全量 Pod 列表，以此产生一个全量的 PodUpdate 事件并写入汇总的 Channel 中，从而实现了多 Pod Source Channel 的“汇聚”逻辑。

分析完 Merger 过程以后，我们接下来看看是什么对象，以及如何消费这个汇总的 Channel。在上一节提到，在 kubelet 进程启动的过程中调用了 startKubelet 方法，此方法首先启动一个协程，让 kubelet 处理来自 PodSource 的 Pod Update 消息，即下面这行代码：

```
go util.Forever(func() { k.Run(podCfg.Updates()) }, 0)
```

其中，PodConfig 的 Updates()方法返回了前面我们所说的汇总 Channel 变量的一个引用，下面是 kubelet 的 Run（updates ＜ -chan PodUpdate)方法的代码：

```
    func (kl *Kubelet) Run(updates <-chan PodUpdate) {
       if kl.logServer == nil {
          kl.logServer = http.StripPrefix("/logs/",
http.FileServer(http.Dir("/var/log/")))
       }
       if kl.kubeClient == nil {
          glog.Warning("No api server defined - no node status update will be sent. ")
       }
       // Move Kubelet to a container.
       if kl.resourceContainer != "" {
          err := util.RunInResourceContainer(kl.resourceContainer)
          if err != nil {
             glog.Warningf("Failed to move Kubelet to container %q: %v", kl.
resourceContainer, err)
          }
          glog.Infof("Running in container %q", kl.resourceContainer)
       }
       if err := kl.imageManager.Start(); err != nil {
     kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
ImageManager %v", err)
          glog.Errorf("Failed to start ImageManager, images may not be garbage
collected: %v", err)
       }
       if err := kl.cadvisor.Start(); err != nil {
          kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
CAdvisor %v", err)
         glog.Errorf("Failed to start CAdvisor, system may not be properly monitored:
%v", err)
       }
       if err := kl.containerManager.Start(); err != nil {
          kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
ContainerManager %v", err)
          glog.Errorf("Failed to start ContainerManager, system may not be properly
isolated: %v", err)
       }
       if err := kl.oomWatcher.Start(kl.nodeRef); err != nil {
          kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
OOM watcher %v", err)
          glog.Errorf("Failed to start OOM watching: %v", err)
       }
```

```
    go util.Until(kl.updateRuntimeUp, 5*time.Second, util.NeverStop)
        // Run the system oom watcher forever.
        kl.statusManager.Start()
        kl.syncLoop(updates, kl)
    }
```

上述代码首先启动了一个 HTTP File Server 来远程获取本节点的系统日志，接下来根据启动参数的设置来决定是否在指定的 Docker 容器中启动 kubelet 进程（如果成功，则将本进程转移到指定的容器中），然后分别启动 Image Manager（负责 Image GC）、cAdvisor（Docker 性能监控）、Container Manager（Container GC）、OOM Watcher（OOM 监测）、Status Manager（负责同步本节点上 Pod 的状态到 API Server 上）等组件，最后进入 syncLoop 方法中，无限循环调用下面的 syncLoopIteration 方法：

```
    func (kl *Kubelet) syncLoopIteration(updates <-chan PodUpdate, handler
SyncHandler) {
        kl.syncLoopMonitor.Store(time.Now())
        if !kl.containerRuntimeUp() {
            time.Sleep(5 * time.Second)
            glog.Infof("Skipping pod synchronization, container runtime is not up. ")
            return
        }
        if !kl.doneNetworkConfigure() {
            time.Sleep(5 * time.Second)
            glog.Infof("Skipping pod synchronization, network is not configured")
            return
        }
        unsyncedPod := false
        podSyncTypes := make(map[types.UID]SyncPodType)
        select {
        case u, ok := <-updates:
            if !ok {
                glog.Errorf("Update channel is closed. Exiting the sync loop. ")
                return
            }
            kl.podManager.UpdatePods(u, podSyncTypes)
            unsyncedPod = true
            kl.syncLoopMonitor.Store(time.Now())
        case <-time.After(kl.resyncInterval):
            glog.V(4).Infof("Periodic sync")
        }
        start := time.Now()
        // If we already caught some update, try to wait for some short time
        // to possibly batch it with other incoming updates.
        for unsyncedPod {
            select {
            case u := <-updates:
```

```
                kl.podManager.UpdatePods(u, podSyncTypes)
                kl.syncLoopMonitor.Store(time.Now())
            case <-time.After(5 * time.Millisecond):
                // Break the for loop.
                unsyncedPod = false
            }
        }
        pods, mirrorPods := kl.podManager.GetPodsAndMirrorMap()
        kl.syncLoopMonitor.Store(time.Now())
        if err := handler.SyncPods(pods, podSyncTypes, mirrorPods, start); err !=
nil {
            glog.Errorf("Couldn't sync containers: %v", err)
        }
        kl.syncLoopMonitor.Store(time.Now())
    }
```

在上述代码中，如果从 Channel 中拉取到了 PodUpdate 事件，则先调用 podManager 的 UpdatePods 方法来确定此 PodUpdate 的同步类型，并将结果放入 podSyncTypes 这个 Map 中，同时为了提升处理效率，在代码中增加了持续循环拉取 PodUpdate 数据直到 Channel 为空为止（超时判断）的一段逻辑。在方法的最后，调用 SyncHandler 接口来完成 Pod 同步的具体逻辑，从而实现了 PodUpdate 事件的高效批处理模式。

SyncHandler 在这里就是 kubelet 实例本身，它的 SyncPods 方法比较长，其主要逻辑如下。

- 将传入的全量 Pod，与 statusManager 中当前保存的 Pod 集合进行对比，删除 statusManager 中当前已经不存在的 Pod（孤儿 Pod）。
- 调用 kubelet 的 admitPods 方法以过滤掉不适合本节点创建的 Pod。此方法首先过滤掉状态为 Failed 或者 Succeeded 的 Pod；接着过滤掉不适合本节点的 Pod，比如 Host Port 冲突、Node Label 的约束不匹配及 Node 的可用资源不足等情况；最后检查磁盘的使用情况，如果磁盘的可用空间不足，则过滤掉所有 Pod。
- 对上述过滤后的 Pod 集合中的每一个 Pod 调用 podWorkers 的 UpdatePod 方法，而此方法内部创建了一个 Pod 的 workUpdate 事件并发布到该 Pod 对应的一个 Work Channel 上（podWorkers.podWorkers）。
- 对于已经删除或不存在的 Pod，通知 podWorkers 删除相关联的 Work Channel（workUpdate）。
- 对比 Node 当前运行中的 Pod 及目标 Pod 列表，“杀掉”多余的 Pod，并且调用 Docker Runtime（Docker Deamon 进程）API，重新获取当前运行中的 Pod 列表信息。
- 清理“孤儿”Pod 所遗留的 PV 和磁盘目录。

要真正理解 Pod 是怎么在 Node 上“落地”的，还要继续深入分析上述第 3 步的代码。首先我们看看对 workUpdatc 这个结构体的定义：

```
type workUpdate struct {
    pod *api.Pod
    // The mirror pod of pod; nil if it does not exist.
    mirrorPod *api.Pod
    // Function to call when the update is complete.
    updateCompleteFn func()
    updateType SyncPodType
}
```

其中的属性 pod 是当前要操作的 Pod 对象，mirrorPod 则是对应的镜像 Pod，下面是对它的解释：

“对于每个来自非 API Server Pod Source 上的 Pod，kubelet 都在 API Server 上注册一个几乎“一模一样”的 Pod，这个 Pod 被称为 mirrorPod，这样一来，就将不同的 Pod Source 上的 Pod 都“统一”到了 kubelet 的注册表上，从而统一了 Pod 生命周期的管理流程。”

workUpdate 的 updateCompleteFn 属性是一个回调函数，work 完成后会执行此回调函数，在上述第 3 步中，此函数用来计算该 work 的调度时延指标。

对于每个要同步的 Pod，podWorkers 会用一个长度为 1 的 Channel 来存放其对应的 workUpdate，而属性 lastUndeliveredWorkUpdate 则存放最近一个待安排执行的 workUpdate，这是因为一个 Pod 的前一个 workUpdate 正在执行的时候，可能会有一个新的 PodUpdate 事件需要处理。理解了这个过程后，再来看 podWorkers 的定义，就不难了：

```
type podWorkers struct {
    // Protects all per worker fields.
    podLock sync.Mutex
    podUpdates map[types.UID]chan workUpdate
    isWorking map[types.UID]bool
    lastUndeliveredWorkUpdate map[types.UID]workUpdate
    runtimeCache kubecontainer.RuntimeCache
    syncPodFn syncPodFnType
  recorder record.EventRecorder
}
```

下面这个函数就是第 3 步里产生 workUpdate 事件并放入到 podWorkers 的对应 Channel 的方法的源码：

```
    func (p *podWorkers) UpdatePod(pod *api.Pod, mirrorPod *api.Pod, updateComplete
func()) {
        uid := pod.UID
        var podUpdates chan workUpdate
        var exists bool
        updateType := SyncPodUpdate
        p.podLock.Lock()
        defer p.podLock.Unlock()
        if podUpdates, exists = p.podUpdates[uid]; !exists {
```

```
            podUpdates = make(chan workUpdate, 1)
            p.podUpdates[uid] = podUpdates
            updateType = SyncPodCreate
            go func() {
                defer util.HandleCrash()
                p.managePodLoop(podUpdates)
            }()
        }
        if !p.isWorking[pod.UID] {
            p.isWorking[pod.UID] = true
            podUpdates <- workUpdate{
                pod:            pod,
                mirrorPod:      mirrorPod,
                updateCompleteFn: updateComplete,
                updateType:     updateType,
            }
        } else {
            p.lastUndeliveredWorkUpdate[pod.UID] = workUpdate{
                pod:            pod,
                mirrorPod:      mirrorPod,
                updateCompleteFn: updateComplete,
                updateType:     updateType,
            }
        }
    }
```

上面的代码会调用 podWorkers 的 managePodLoop 方法来处理 podUpdates 队列，这里主要是获取必要的参数，最终处理又转手交给 syncPodFn 方法去处理。下面是 managePodLoop 的源码：

```
    func (p *podWorkers) managePodLoop(podUpdates <-chan workUpdate) {
        var minRuntimeCacheTime time.Time
        for newWork := range podUpdates {
            func() {
    defer p.checkForUpdates(newWork.pod.UID, newWork.updateCompleteFn)
    if err := p.runtimeCache.ForceUpdateIfOlder(minRuntimeCacheTime); err != nil {
        glog.Errorf("Error updating the container runtime cache: %v", err)
                return
            }
        pods, err := p.runtimeCache.GetPods()
        if err != nil {
            glog.Errorf("Error getting pods while syncing pod: %v", err)
                return
            }
            err = p.syncPodFn(newWork.pod, newWork.mirrorPod,
         kubecontainer.Pods(pods).FindPodByID(newWork.pod.UID), newWork.
updateType)
            if err != nil {
```

```
    glog.Errorf("Error syncing pod %s, skipping: %v", newWork.pod.UID, err)
    p.recorder.Eventf(newWork.pod, "failedSync", "Error syncing pod, skipping: %v",
err)
                    return
                }
                minRuntimeCacheTime = time.Now()
                newWork.updateCompleteFn()
            }()
        }
    }
```

追踪podWorkers的构造函数调用过程，可以发现syncPodFn函数其实就是kubelet的syncPod方法，这个方法的代码量有点儿多，主要逻辑如下。

（1）根据系统配置中的权限控制，检查 Pod 是否有权在本节点运行，这些权限包括 Pod 是否有权使用 HostNetwork（还记得之前分析的代码么？由 Pod Source 类型决定）、Pod 中的容器是否被授权以特权模式启动（privileged mode）等，如果未被授权，则删除当前运行中的旧版本的 Pod 实例并返回错误信息。

（2）创建 Pod 相关的工作目录、PV 存放目录、Plugin 插件目录，这些目录都以 Pod 的 UID 为上一级目录。

（3）如果 Pod 有 PV 定义，则针对每个 PV 执行目录的 mount 操作。

（4）如果是 SyncPodUpdate 类型的 Pod，则从 Docker Runtime 的 API 接口查询获取 Pod 及相关容器的最新状态信息。

（5）如果 Pod 有 imagePullSecrets 属性，则在 API Server 上获取对应的 Secret。

（6）调用 Container Runtime 的 API 接口方法 SyncPod，实现 Pod“真正同步”的逻辑。

（7）如果 Pod Source 不来自 API Server，则继续处理其关联的 mirrorPod。

- 如果 mirrorPod 跟当前 Pod 的定义不匹配，则它会被删除。
- 如果 mirrorPod 还不存在（比如新创建的 Pod），则会在 API Server 上新建一个。

Kubernetes 中 Container Runtime 的默认实现是 Dockers，对应类是 dockertools.DockerManager，其源码位于 kg/kubelet/dockertools/manager.go 里，在上述 kubelet.syncPod 方法中所调用的 DockerManager 的 SyncPod 方法实现了下面的逻辑。

- 判断一个 Pod 实例的哪些组成部分需要重启：包括 Pod 的 infra 容器是否发生变化（如网络模式、Pod 里运行的各个容器的端口是否发生变化）；Pod 里运行的容器是否发生变化；用 Probe 检测容器的状态以确定容器是否异常等。
- 根据 Pod 实例重启结果的判断，如果需要重启 Pod 的 infra 容器，则先 Kill Pod 然后启

动 Pod 的 infra 容器，设定好网络，最后启动 Pod 里的所有 Container；否则就先 Kill 那些需要重启的 Container，然后重新启动它们。注意，如果是新创建的 Pod，则因为找不到 Node 上对应的 Pod 的 infra 容器，所以会被当作重启 Pod 的 infra 容器的逻辑来实现创建过程。

DockerManager 创建 Pod 的 infra 容器的逻辑在 createPodInfraContainer 方法里，大体逻辑如下。

- 如果 Pod 的网络不是 HostNetwork 模式，则搜集 Pod 所有容器的 Port 作为 infra 容器所要暴露的 Port 列表。
- 如果 infra 容器的 Image 目前不存在，则尝试拉取 Image。
- 创建 infra 的 Container 对象并且启动 runContainerInPod 方法。
- 如果容器定义有 Lifecycle，并且 PostStart 回调方法被设置了，就会触发此方法的调用，如果调用失败则 Kill 容器并返回。
- 创建一个软连接文件指向容器的日志文件，此软连接文件名包括 Pod 的名称、容器的名称及容器的 ID，这样的目的是让 ElasticSearch 这样的搜索技术容易索引和定位 Pod 日志。
- 如果此容器是 Pod infra 容器，则设置其 OOM 参数低于标准值，使得它比其他容器具备更强的“抗灾”能力。
- 修改 Docker 生成的容器的 resolv.conf 文件，增加 ndots 参数并默认设置为 5，这是因为 Kubernetes 默认假设的域名分割长度是 5，例如_dns._udp.kube-dns.default.svc。

上述逻辑中所调用的 runContainerInPod 是 DockerManager 的核心方法之一，不管是创建 Pod 的 infra 容器还是 Pod 里的其他容器，都会通过此方法使得容器被创建和运行。以下是其主要逻辑。

- 生成 Container 必要的环境变量和参数，比如 ENV 环境变量、Volume Mounts 信息、端口映射信息、DNS 服务器信息、容器的日志目录、parent cgGroup 等。
- 调用 runContainer 方法完成 Docker Container 实例的创建过程，简单地说，就是完成 Docker create container 命令行所需的各种参数的构造过程，并通过程序来调用执行。
- 构造 HostConfig 对象，主要参数有目录映射、端口映射等、cgGroup 的设定等，简单地说，就是完成了 Docker start container 命令行所需的必要参数的构造过程，并通过程序来调用执行。

在上述逻辑中，runContainer 与 startContainer 的具体实现都是靠 DockerManager 中的 dockerClient 对象完成的，它实现了 DockerInterface 接口，dockerClient 的创建过程在 pkg/kubelet/dockertools/docker.go 里，下面是这段代码：

```
func ConnectToDockerOrDie(dockerEndpoint string) DockerInterface {
```

```
        if dockerEndpoint == "fake:// " {
            return &FakeDockerClient{
                VersionInfo: docker.Env{"ApiVersion=1.18"},
            }
        }
        client, err := docker.NewClient(getDockerEndpoint(dockerEndpoint))
        if err != nil {
            glog.Fatalf("Couldn't connect to docker: %v", err)
        }
        return client
    }
```

这里的 dockerEndpoint 是本节点上的 Docker Deamon 进程的访问地址，默认是 unix:///var/run/docker.sock，在上述代码中使用了来自开源项目 https://github.com/fsouza/go-dockerclient 提供的 Docker Client，它也是 Go 语言实现的一个用 HTTP 访问 Docker Deamon 提供的标准 API 的客户端框架。

我们来看看 dockerClient 创建容器的具体代码（CreateContainer）：

```
    func (c *Client) CreateContainer(opts CreateContainerOptions) (*Container, error) {
        path := "/containers/create? " + queryString(opts)
        body, status, err := c.do(
            "POST",
            path,
            doOptions{
                data: struct {
                    *Config
                    HostConfig *HostConfig `json: "HostConfig,omitempty" yaml:
"HostConfig,omitempty"`
                }{
                    opts.Config,
                    opts.HostConfig,
                },
            },
        )
        if status == http.StatusNotFound {
            return nil, ErrNoSuchImage
        }
        if err != nil {
            return nil, err
        }
        var container Container
        err = json.Unmarshal(body, &container)
        if err != nil {
            return nil, err
        }
        container.Name = opts.Name
```

```
    return &container, nil
}
```

上述代码其实就是通过调用标准的 Docker Rest API 来实现功能的，我们进入 docker.Client 的 do 方法里可以看到更多详情，例如输入参数转换为 JSON 格式的数据、DockerAPI 版本检查及异常处理等逻辑，最有趣的是：在 dockerEndpoint 是 unix 套接字的情况下，会先建立套接字连接，然后在这个连接上创建 HTTP 连接。

至此，我们分析了 kubelet 创建和同步 Pod 实例的整个流程，简单总结如下。

- 汇总：先将多个 Pod Source 上过来的 PodUpdate 事件汇聚到一个总的 Channel 上去。
- 初审：分析并过滤掉不符合本节点的 PodUpdate 事件，对满足条件的 PodUpdate 则生成一个 workUpdate 事件，交给 podWorkers 处理。
- 接待：podWorkers 对每个 Pod 的 workUpdate 事件排队，并且负责更新 Cache 中的 Pod 状态，而把具体的任务转给 kubelet 去处理（syncPod 方法）。
- 终审：kubelet 对符合条件的 Pod 进一步审查，例如检查 Pod 是否有权在本节点运行，对符合审查的 Pod 开始着手准备工作，包括目录创建、PV 创建、Image 获取、处理 Mirror Pod 问题等，然后把“皮球”踢给了 DockerManager。
- 落地：任务抵达 DockerManager 之后，DockerManager 尽心尽责地分析每个 Pod 的情况，以决定这个 Pod 究竟是新建、完全重启还是部分更新的。给出分析结果以后，剩下的就是 dockerClient 的工作了。

好复杂的设计！原来非业务流程的代码理解起来也会如此折磨人，真心不知道谷歌当初是怎么设计和实现它的，目测国内 P8 水平的一帮大牛们天天加班到 9 点钟，也难以交付这样的 Code。

在继续下面的分析之前，留一个小小的思考给聪明的读者：Pod Source 上发来的 Pod 删除的事件，是在哪里处理的？

接下来我们继续分析 kubelet 进程的另外一个重要功能是如何实现的，即定期同步 Pod 状态信息到 API Server 上。先来看看 Pod 状态的数据结构定义：

```
type PodStatus struct {
    Phase      PodPhase       `json: "phase,omitempty"`
    Conditions []PodCondition `json: "conditions,omitempty"`
    Message string `json: "message,omitempty"`
    Reason string `json: "reason,omitempty"`
    HostIP string `json: "hostIP,omitempty"`
    PodIP  string `json: "podIP,omitempty"`
    StartTime *util.Time `json: "startTime,omitempty"`
    ContainerStatuses []ContainerStatus
}
// PodStatusResult is a wrapper for PodStatus returned by kubelet that can be
```

```
encode/decoded
    type PodStatusResult struct {
       TypeMeta   `json: ",inline"`
       ObjectMeta `json: "metadata,omitempty"`
       Status PodStatus `json: "status,omitempty"`
    }
```

Pod 的状态（Phase）有 5 种：运行中（PodRunning）、等待中（PodPending）、正常终止（PodSucceeded）、异常停止（PodFailed）及未知状态（PodUnknown），最后一种状态很可能是由于 Pod 所在主机的通信问题导致的。从上面的定义可以看到 Pod 的状态同时包括它里面运行的 Container 的状态，另外给出了导致当前状态的原因说明、Pod 的启动时间等信息。PodStatusResult 则是 Kubernete API Server 提供的 Pod Status API 接口中用到的 Wrapper 类。

通过之前的代码研读，我们发现在 Kubernetes 中大量使用了 Channel 和协程机制来完成数据的高效传递和处理工作，在 kubelet 中更是大量使用了这一机制，实现 Pod Status 上报的 kubelet.statusManager 也是如此，它用一个 Map（podStatuses）保存了当前 kubelet 中所有 Pod 实例的当前状态，并且声明了一个 Channel（podStatusChannel）来存放 Pod 状态同步的更新请求（podStatuses），Pod 在本地实例化和同步的过程中会引发 Pod 状态的变化，这些变化被封装为 podStatusSyncRequest 放入 Channel 中，然后被异步上报到 API Server，这就是 statusManager 的运行机制。

下面是 statusManager 的 SetPodStatus 方法，先比较缓存的状态信息，如果状态发生变化，则触发 Pod 状态，生成 podStatusSyncRequest 并放到队列中等待上报：

```
func (s *statusManager) SetPodStatus(pod *api.Pod, status api.PodStatus) {
    podFullName := kubecontainer.GetPodFullName(pod)
    s.podStatusesLock.Lock()
    defer s.podStatusesLock.Unlock()
    oldStatus, found := s.podStatuses[podFullName]
    // ensure that the start time does not change across updates.
    if found && oldStatus.StartTime != nil {
        status.StartTime = oldStatus.StartTime
    }
    if status.StartTime.IsZero() {
        if pod.Status.StartTime.IsZero() {
            // the pod did not have a previously recorded value so set to now
            now := util.Now()
            status.StartTime = &now
        } else {
            status.StartTime = pod.Status.StartTime
        }
    }
    if !found || !isStatusEqual(&oldStatus, &status) {
        s.podStatuses[podFullName] = status
```