获取Kubernetes Proxy API 接口数据：

ReplicationController作用：

3.1 Kubernetes API Server 原理分析

总体来看，Kubernetes API Server 的核心功能是提供了 Kubernetes 各类资源对象（如 Pod、RC、Service 等）的增、删、改、查及 Watch 等 HTTP Rest 接口，成为集群内各个功能模块之间数据交互和通信的中心枢纽，是整个系统的数据总线和数据中心。除此之外，它还有以下一些功能特性。

（1）是集群管理的 API 入口。

（2）是资源配额控制的入口。

（3）提供了完备的集群安全机制。

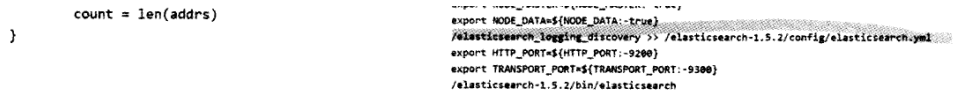

图 3.1 应用程序编程访问 API Server

在上述使用场景中，Pod 中的进程如何知道 API Server 的访问地址呢？答案很简单，因为 Kubernetes API Server 本身也是一个 Service，它的名字就是“kubernetes”，并且它的 Cluster IP 地址是 Cluster IP 地址池里的第 1 个地址！另外，它所服务的端口是 HTTPS 端口 443，通过 kubectl get service 命令可以确认这一点：

```
# kubectl get service
NAME            CLUSTER-IP       EXTERNAL-IP    PORT(S)           AGE
kubernetes      169.169.0.1      <none>         443/TCP           30d
```

第 2 种使用场景：开发基于 Kubernetes 的管理平台。比如调用 Kubernetes API 来完成 Pod、Service、RC 等资源对象的图形化创建和管理界面，此时可以使用 Kubernetes 及各开源社区为开发人员提供的各种语言版本的 Client Library。我们会在后面介绍通过编程方式访问 API Server 的一些细节技术。

获取Kubernetes Proxy API 接口数据：

3.1.2 独特的 Kubernetes Proxy API 接口

前面我们说过，Kubernetes API Server 最主要的 REST 接口是资源对象的增、删、改、查，除此之外，它还提供了一类很特殊的 REST 接口——Kubernetes Proxy API 接口，这类接口的作用是代理 REST 请求，即 Kubernetes API Server 把收到的 REST 请求转发到某个 Node 上的 kubelet 守护进程的 REST 端口上，由该 Kubelet 进程负责响应。

首先，我们来说说 Kubernetes Proxy API 里关于 Node 的相关接口，该接口的 REST 路径为 /api/v1/proxy/nodes/{name}，其中{name}为节点的名称或 IP 地址，包括以下几个具体接口：

- `/api/v1/proxy/nodes/{name}/pods/` #列出指定节点内所有 Pod 的信息
- `/api/v1/proxy/nodes/{name}/stats/` #列出指定节点内物理资源的统计信息
- `/api/v1/proxy/nodes/{name}/spec/` #列出指定节点的概要信息

例如当前 Node 节点的名字为 k8s-node-1，用下面的命令即可获取该节点上所有运行中的 Pod：

```
# curl localhost:8080/api/v1/proxy/nodes/k8s-node-1/pods
```

需要说明的是：这里获取的 Pod 的信息数据来自 Node 而非 etcd 数据库，所以两者可能在某些时间点会有偏差。此外，如果 kubelet 进程在启动时包含--enable-debugging-handlers =true 参数，那么 Kubernetes Proxy API 还会增加下面的接口：

- `/api/v1/proxy/nodes/{name}/run` #在节点上运行某个容器
- `/api/v1/proxy/nodes/{name}/exec` #在节点上的某个容器中运行某条命令
- `/api/v1/proxy/nodes/{name}/attach` #在节点上 attach 某个容器
  `/api/v1/proxy/nodes/{name}/portForward` #实现节点上的 Pod 端口转发
- `/api/v1/proxy/nodes/{name}/logs` #列出节点的各类日志信息，例如 tallylog、lastlog、wtmp、ppp/、rhsm/、audit/、tuned/和 anaconda/等
- `/api/v1/proxy/nodes/{name}/metrics` #列出和该节点相关的 Metrics 信息
- `/api/v1/proxy/nodes/{name}/runningpods` #列出节点内运行中的 Pod 信息
- `/api/v1/proxy/nodes/{name}/debug/pprof` #列出节点内当前 Web 服务的状态，包括 CPU 占用情况和内存使用情况等

接下来，我们来说说 Kubernetes Proxy API 里关于 Pod 的相关接口，通过这些接口，我们可以访问 Pod 里某个容器提供的服务（如 Tomcat 在 8080 端口服务）：

- `/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*}` #访问 Pod 的某个服务接口
- `/api/v1/proxy/namespaces/{namespace}/pods/{name}` #访问 Pod
- `/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*}` #访问 Pod 的某个服务接口
- `/api/v1/namespaces/{namespace}/pods/{name}/proxy` #访问 Pod

在上面的 4 个接口里，后面两个接口的功能与前面两个完全一样，只是写法不同。下面我们用第 1 章的 Java Web 例子中的 Tomcat Pod 来说明上述 Proxy 接口的用法。

```
# curl http://localhost:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/
```

我们也可以在浏览器中访问上面的地址，比如 Master 节点的 IP 地址是 192.168.18.131，我们在浏览器中输入 http://192.168.18.131:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/，就能够访问 Tomcat 首页了；而如果输入/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/demo，就能访问 Tomcat 中 Demo 应用的页面了。

看到这里，你可能明白 Pod 的 Proxy 接口的作用和意义了：在 Kubernetes 集群之外访问某个 Pod 容器的服务（HTTP 服务）时，可以用 Proxy API 实现，这种场景多用于管理目的，比如逐一排查 Service 的 Pod 副本，检查哪些 Pod 的服务存在异常问题。

最后我们说说 Service，Kubernetes Proxy API 也有 Service 的 Proxy 接口，其接口定义与 Pod 的接口定义基本一样：/api/v1/proxy/namespaces/{namespace}/services/{name}。比如，我们想访问 myweb 这个 Service，则可以在浏览器里输入 http://192.168.18.131:8080/api/v1/proxy/namespaces/default/services/myweb/demo/。

http://localhost:8080/api/v1/proxy/namespaces/default/pods/myweb-gasdfs

http://localhost:8080

3.2 Controller Manager 原理分析

Controller Manager 作为集群内部的管理控制中心，负责集群内的 Node、Pod 副本、服务端点（Endpoint）、命名空间（Namespace）、服务账号（ServiceAccount）、资源定额（ResourceQuota）等的管理，当某个 Node 意外宕机时，Controller Manager 会及时发现此故障并执行自动化修复流程，确保集群始终处于预期的工作状态。

ReplicationController作用：

我们总结一下 Replication Controller 的职责，如下所述。

（1）确保当前集群中有且仅有 *N* 个 Pod 实例，*N* 是 RC 中定义的 Pod 副本数量。

（2）通过调整 RC 的 spec.replicas 属性值来实现系统扩容或者缩容。

（3）通过改变 RC 中的 Pod 模板（主要是镜像版本）来实现系统的滚动升级。

最后，我们总结一下 Replication Controller 的典型使用场景，如下所述。

（1）重新调度（Rescheduling）。如前面所提及的，不管你想运行 1 个副本还是 1000 个副本，副本控制器都能确保指定数量的副本存在于集群中，即使发生节点故障或 Pod 副本被终止运行等意外状况。

（2）弹性伸缩（Scaling）。手动或者通过自动扩容代理修改副本控制器的 spec.replicas 属性值，非常容易实现扩大或缩小副本的数量。

（3）滚动更新（Rolling Updates）。副本控制器被设计成通过逐个替换 Pod 的方式来辅助服务的滚动更新。推荐的方式是创建一个新的只有一个副本的 RC，若新的 RC 副本数量加 1，则旧的 RC 的副本数量减 1，直到这个旧的 RC 的副本数量为零，然后删除该旧的 RC。通过上述模式，即使在滚动更新的过程中发生了不可预料的错误，Pod 集合的更新也都在可控范围内。

NodeController工作流程图：

Node 节点的相关控制功能，Node Controller 的核心工作流程如图 3.4 所示。

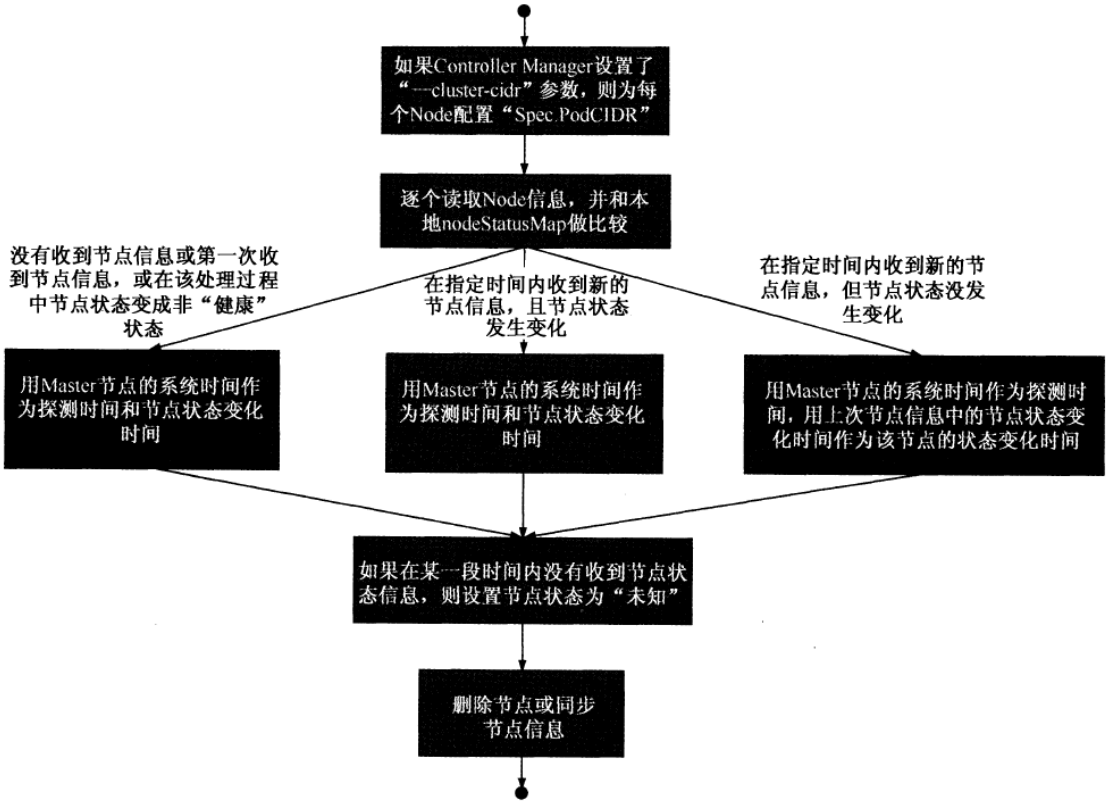

图 3.4　Node Controller 流程图

ResourceQuotaController：资源配额管理

3.2.3　ResourceQuota Controller

作为完备的企业级的容器集群管理平台，Kubernetes 也提供了资源配额管理（ResourceQuota Controller）这一高级功能，资源配额管理确保了指定的资源对象在任何时候都不会超量占用系统物理资源，避免了由于某些业务进程的设计或实现的缺陷导致整个系统运行紊乱甚至意外宕机，对整个集群的平稳运行和稳定性有非常重要的作用。

目前 Kubernetes 支持如下三个层次的资源配额管理。

（1）容器级别，可以对 CPU 和 Memory 进行限制。

（2）Pod 级别，可以对一个 Pod 内所有容器的可用资源进行限制。

（3）Namespace 级别，为 Namespace（多租户）级别的资源限制，包括：

- Pod 数量；
- Replication Controller 数量；
- Service 数量；
- ResourceQuota 数量；
- Secret 数量；
- 可持有的 PV（Persistent Volume）数量。

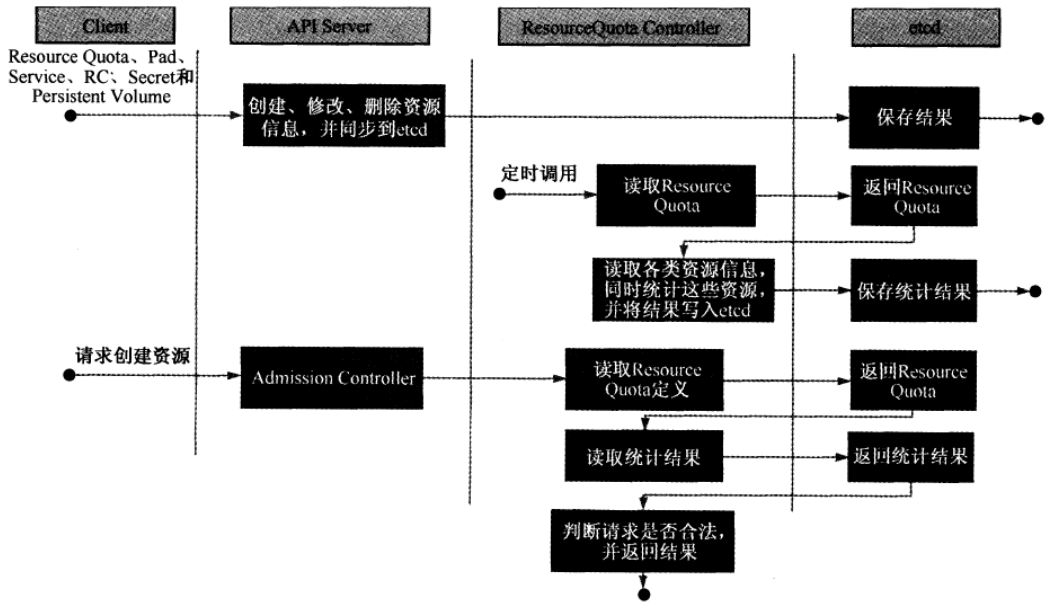

图 3.5　ResourceQuota Controller 流程图

NameSpace Controller:

3.2.4　Namespace Controller

用户通过 API Server 可以创建新的 Namespace 并保存在 etcd 中，Namespace Controller 定时通过 API Server 读取这些 Namespace 信息。如果 Namespace 被 API 标识为优雅删除（通过设置删除期限，即 DeletionTimestamp 属性被设置），则将该 NameSpace 的状态设置成“Terminating”并保存到 etcd 中。同时 Namespace Controller 删除该 Namespace 下的 ServiceAccount、RC、Pod、Secret、PersistentVolume、ListRange、ResourceQuota 和 Event 等资源对象。

当 Namespace 的状态被设置成“Terminating”后，由 Admission Controller 的 NamespaceLifecycle 插件来阻止为该 Namespace 创建新的资源。同时，在 Namespace Controller 删除完该 Namespace 中的所有资源对象后，Namespace Controller 对该 Namespace 执行 finalize 操作，删除 Namespace 的 spec.finalizers 域中的信息。

如果 Namespace Controller 观察到 Namespace 设置了删除期限，同时 Namespace 的 spec.finalizers 域值是空的，那么 Namespace Controller 将通过 API Server 删除该 Namespace 资源。

ServiceController与EndpointController

3.2.5　Service Controller 与 Endpoint Controller

我们先说说 Endpoints Controller，在这之前，让我们先看看 Service、Endpoints 与 Pod 的关系，如图 3.6 所示，Endpoints 表示了一个 Service 对应的所有 Pod 副本的访问地址，而 Endpoints Controller 就是负责生成和维护所有 Endpoints 对象的控制器。

它负责监听 Service 和对应的 Pod 副本的变化，如果监测到 Service 被删除，则删除和该 Service 同名的 Endpoints 对象；如果监测到新的 Service 被创建或者修改，则根据该 Service 信息获得相关的 Pod 列表，然后创建或者更新 Service 对应的 Endpoints 对象。如果监测到 Pod 的事件，则更新它所对应的 Service 的 Endpoints 对象（增加、删除或者修改对应的 Endpoint 条目）。

那么，Endpoints 对象是在哪里被使用的呢？答案是每个 Node 上的 kube-proxy 进程，kube-proxy 进程获取每个 Service 的 Endpoints，实现了 Service 的负载均衡功能。在后面的章节中我们会深入讲解这部分内容。

接下来我们说说 Service Controller 的作用，它其实是属于 Kubernetes 集群与外部的云平台之间的一个接口控制器。Service Controller 监听 Service 的变化，如果是一个 LoadBalancer 类型的 Service（externalLoadBalancers=true），则 Service Controller 确保外部的云平台上该 Service 对应的 LoadBalancer 实例被相应地创建、删除及更新路由转发表（根据 Endpoints 的条目）。

3.3 Scheduler 原理分析

我们在前面深入分析了 Controller Manager 及它所包含的各个组件的运行机制。本节我们将继续对 Kubernetes 中负责 Pod 调度的重要功能模块——Kubernetes Scheduler 的工作原理和运行机制做深入分析。

Kubernetes Scheduler 在整个系统中承担了“承上启下”的重要功能，“承上”是指它负责接收 Controller Manager 创建的新 Pod，为其安排一个落脚的“家”——目标 Node；“启下”是指安置工作完成后，目标 Node 上的 kubelet 服务进程接管后继工作，负责 Pod 生命周期中的“下半生”。

具体来说，Kubernetes Scheduler 的作用是将待调度的 Pod（API 新创建的 Pod、Controller Manager 为补足副本而创建的 Pod 等）按照特定的调度算法和调度策略绑定（Binding）到集群中的某个合适的 Node 上，并将绑定信息写入 etcd 中。在整个调度过程中涉及三个对象，分别是：待调度 Pod 列表、可用 Node 列表，以及调度算法和策略。简单地说，就是通过调度算法调度为待调度 Pod 列表的每个 Pod 从 Node 列表中选择一个最适合的 Node。

随后，目标节点上的 kubelet 通过 API Server 监听到 Kubernetes Scheduler 产生的 Pod 绑定事件，然后获取对应的 Pod 清单，下载 Image 镜像，并启动容器。完整的流程如图 3.7 所示。

3.4：kubelet运行机制：

1：节点管理：--apiserviers告诉位置；--kubeconfig告诉证书；--cloud-provider如何从IaaS读取

2：Pod管理：三种获取方式：文件、http端点、api server

3：容器健康检查:通过LivenessProbe探针来查看：execAction 内部检查、TCPSockertAction访问TCP端口检查；HTTPGetAction获取返回码200-400

4：cAdvisor资源健康：等同于日志记录功能，提供restApi暴露这些数据供pod进行统计查看

3.4 kubelet 运行机制分析

在 Kubernetes 集群中，在每个 Node 节点（又称 Minion）上都会启动一个 kubelet 服务进程。该进程用于处理 Master 节点下发到本节点的任务，管理 Pod 及 Pod 中的容器。每个 kubelet 进程会在 API Server 上注册节点自身信息，定期向 Master 节点汇报节点资源的使用情况，并通过 cAdvisor 监控容器和节点资源。

3.4.1 节点管理

节点通过设置 kubelet 的启动参数“--register-node”，来决定是否向 API Server 注册自己。如果该参数的值为 true，那么 kubelet 将试着通过 API Server 注册自己。在自注册时，kubelet 启动时还包含下列参数。

- --api-servers：告诉 kubelet API Server 的位置。
- --kubeconfig：告诉 kubelet 在哪儿可以找到用于访问 API Server 的证书。
- --cloud-provider：告诉 kubelet 如何从云服务商（IaaS）那里读取到和自己相关的元数据。

当前每个 kubelet 被授予创建和修改任何节点的权限。但是在实践中，它仅仅创建和修改自己。将来，我们计划限制 kubelet 的权限，仅允许它修改和创建其所在节点的权限。如果在集群运行过程中遇到集群资源不足的情况，则用户很容易通过添加机器及运用 kubelet 的自注册模式

3.4.2 Pod 管理

kubelet 通过以下几种方式获取自身 Node 上所要运行的 Pod 清单。

（1）文件：kubelet 启动参数“--config”指定的配置文件目录下的文件（默认目录为“/etc/kubernetes/manifests/”）。通过--file-check-frequency 设置检查该文件目录的时间间隔，默认为 20 秒。

（2）HTTP 端点（URL）：通过“--manifest-url”参数设置。通过--http-check-frequency 设置检查该 HTTP 端点数据的时间间隔，默认为 20 秒。

（3）API Server：kubelet 通过 API Server 监听 etcd 目录，同步 Pod 列表。

所有以非 API Server 方式创建的 Pod 都叫作 Static Pod。kubelet 将 Static Pod 的状态汇报给 API Server，API Server 为该 Static Pod 创建一个 Mirror Pod 和其相匹配。Mirror Pod 的状态将真实反映 Static Pod 的状态。当 Static Pod 被删除时，与之相对应的 Mirror Pod 也会被删除。在本章中我们只讨论通过 API Server 获得 Pod 清单的方式。kubelet 通过 API Server Client 使用 Watch 加 List 的方式监听“/registry/nodes/$当前节点的名称”和“/registry/pods”目录，将获取的信息同步到本地缓存中。

kubelet 监听 etcd，所有针对 Pod 的操作将会被 kubelet 监听到。如果发现有新的绑定到本节点的 Pod，则按照 Pod 清单的要求创建该 Pod。

如果发现本地的 Pod 被修改，则 kubelet 会做出相应的修改，比如删除 Pod 中的某个容器时，则通过 Docker Client 删除该容器。

如果发现删除本节点的 Pod，则删除相应的 Pod，并通过 Docker Client 删除 Pod 中的容器。

kubelet 读取监听到的信息，如果是创建和修改 Pod 任务，则做如下处理。

（1）为该 Pod 创建一个数据目录。

（2）从 API Server 读取该 Pod 清单。

（3）为该 Pod 挂载外部卷（External Volume）。

（4）下载 Pod 用到的 Secret。

（5）检查已经运行在节点中的 Pod，如果该 Pod 没有容器或 Pause 容器（“kubernetes/pause”镜像创建的容器）没有启动，则先停止 Pod 里所有容器的进程。如果在 Pod 中有需要删除的容器，则删除这些容器。

（6）用“kubernetes/pause”镜像为每个 Pod 创建一个容器。该 Pause 容器用于接管 Pod 中所有其他容器的网络。每创建一个新的 Pod，kubelet 都会先创建一个 Pause 容器，然后创建其他容器。“kubernetes/pause”镜像大概为 200KB，是一个非常小的容器镜像。

（7）为 Pod 中的每个容器做如下处理。

- 为容器计算一个 hash 值，然后用容器的名字去查询对应 Docker 容器的 hash 值。若查找到容器，且两者的 hash 值不同，则停止 Docker 中容器的进程，并停止与之关联的 Pause 容器的进程；若两者相同，则不做任何处理。
- 如果容器被终止了，且容器没有指定的 restartPolicy（重启策略），则不做任何处理。
- 调用 Docker Client 下载容器镜像，调用 Docker Client 运行容器。

3.4.3 容器健康检查

Pod 通过两类探针来检查容器的健康状态。一个是 LivenessProbe 探针，用于判断容器是否健康，告诉 kubelet 一个容器什么时候处于不健康的状态。如果 LivenessProbe 探针探测到容器不健康，则 kubelet 将删除该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含 LivenessProbe 探针，那么 kubelet 认为该容器的 LivenessProbe 探针返回的值永远是“Success”；另一类是 ReadinessProbe 探针，用于判断容器是否启动完成，且准备接收请求。如果 ReadinessProbe 探针检测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 IP 地址的 Endpoint 条目。

kubelet 定期调用容器中的 LivenessProbe 探针来诊断容器的健康状况。LivenessProbe 包含以下三种实现方式。

（1）ExecAction：在容器内部执行一个命令，如果该命令的退出状态码为 0，则表明容器健康。

（2）TCPSocketAction：通过容器的 IP 地址和端口号执行 TCP 检查，如果端口能被访问，则表明容器健康。

（3）HTTPGetAction：通过容器的 IP 地址和端口号及路径调用 HTTP Get 方法，如果响应的状态码大于等于 200 且小于等于 400，则认为容器状态健康。

3.6.1 API Server 认证

我们知道，Kubernetes 集群中所有资源的访问和变更都是通过 Kubernetes API Server 的 REST API 来实现的，所以集群安全的关键点就在于如何识别并认证客户端身份（Authentication），以及随后访问权限的授权（Authorization）这两个关键问题，本节我们讲解前一个问题。

我们知道，Kubernetes 集群提供了 3 种级别的客户端身份认证方式。

- 最严格的 HTTPS 证书认证：基于 CA 根证书签名的双向数字证书认证方式。
- HTTP Token 认证：通过一个 Token 来识别合法用户。
- HTTP Base 认证：通过用户名+密码的方式认证。

首先，我们说说 HTTPS 证书认证的原理。

CA认证：

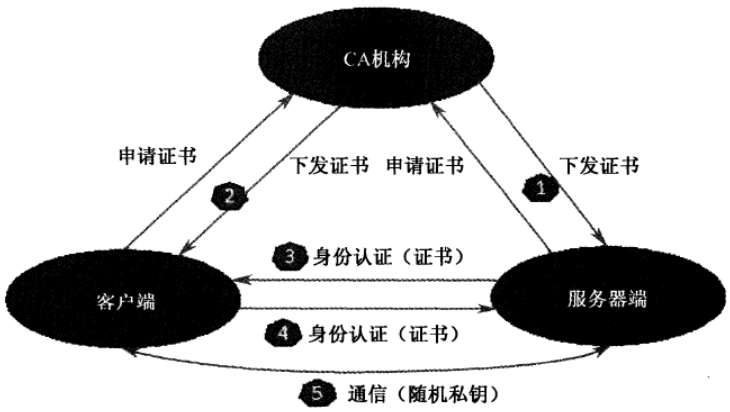

图 3.11　CA 认证流程

HTTP Token 的认证是用一个很长的特殊编码方式的并且难以被模仿的字符串——Token 来表明客户身份的一种方式。在通常情况下，Token 是一个很复杂的字符串，比如我们用私钥签名一个字符串后的数据就可以当作一个 Token。此外，每个 Token 对应一个用户名，存储在 API Server 能访问的一个文件中。当客户端发起 API 调用请求时，需要在 HTTP Header 里放入 Token，这样一来，API Server 就能识别合法用户和非法用户了。

最后，我们说说 HTTP Base 认证。

我们知道，HTTP 协议是无状态的，浏览器和 Web 服务器之间可以通过 Cookie 来进行身份识别。桌面应用程序(比如新浪桌面客户端、SkyDrive 客户端、命令行程序)一般不会使用 Cookie，那么它们与 Web 服务器之间是如何进行身份识别的呢？这就用到了 HTTP Base 认证，这种认证方式是把“用户名+冒号+密码”用 BASE64 算法进行编码后的字符串放在 HTTP Request 中的 Header Authorization 域里发送给服务端，服务端收到后进行解码，获取用户名及密码，然后进行用户身份的鉴权过程。

API Server授权

3.6.2　API Server 授权

对合法用户进行授权（Authorization）并且随后在用户访问时进行鉴权，是权限与安全系统的重要一环。简单地说，授权就是授予不同的用户不同的访问权限，API Server 目前支持以下几种授权策略（通过 API Server 的启动参数“--authorization_mode”设置）。

- AlwaysDeny。
- AlwaysAllow。
- ABAC。

其中，AlwaysDeny 表示拒绝所有的请求，该配置一般用于测试；AlwaysAllow 表示接收所有的请求，如果集群不需要授权流程，则可以采用该策略，这也是 Kubernetes 的默认配置；ABAC（Attribute-Based Access Control）为基于属性的访问控制，表示使用用户配置的授权规则去匹配用户的请求。

为了简化授权的复杂度，对于 ABAC 模式的授权策略，Kubernetes 仅有下面四个基本属性。

ABAC

◎ 被访问对象所属的 Namespace。

当我们为 API Server 启用 ABAC 模式时，需要指定授权策略文件的路径和名字（--authorization_policy_file=SOME_FILENAME），授权策略文件里的每一行都是一个 Map 类型的 JSON 对象，被称为“访问策略对象”，我们可以通过设置“访问策略对象”中的如下属性来确定具体的授权行为。

◎ user（用户名）：为字符串类型，该字符串类型的用户名来源于 Token 文件或基本认证文件中的用户名字段的值。

◎ readonly（只读标识）：为布尔类型，当它的值为 true 时，表明该策略允许 GET 请求通过。

◎ resource（资源）：为字符串类型，来自于 URL 的资源，例如“Pods”。

◎ namespace（命名空间）：为字符串类型，表明该策略允许访问某个 Namespace 的资源。

例如，我们要实现如下访问控制。

（1）允许用户 alice 做任何事情

（2）kubelet 只能访问 Pod 的只读 API。

（3）kubelet 能读和写 Event 对象。

（4）用户 bob 只能访问 myNamespace 中的 Pod 的只读 API。

则满足上述要求的授权策略文件的内容写法如下：

```
{"user":"alice"}
{"user":"kubelet", "resource": "pods", "readonly": true}
{"user":"kubelet", "resource": "events"}
{"user":"bob", "resource": "pods", "readonly": true, "ns": " myNamespace "}
```

3准入控制：

```
ceAccount,ResourceQuota
```

大部分准入控制器都比较容易理解，我们接下来着重介绍 SecurityContextDeny、ResourceQuota 及 LimitRanger 这三个准入控制器。

1）SecurityContextDeny

Security Context 是运用于容器的操作系统安全设置（uid、gid、capabilities、SELinux role 等）。Admission Control 的 SecurityContextDeny 插件的作用是，禁止创建设置了 Security Context 的 Pod，例如包含下面这些配置项的 Pod：

```
spec.containers.securityContext.seLinuxOptions
spec.containers.securityContext.runAsUser
```

2）ResourceQuota

准入控制器 ResourceQuota 不仅能够限制某个 Namespace 中创建资源的数量，而且能够限制某个 Namespace 中被 Pod 所请求的资源总量。该准入控制器和资源对象 ResourceQuota 一起实现了资源配额管理。

3）LimitRanger

准入控制器 LimitRanger 的作用类似于上面的 ResourceQuota 控制器，针对 Namespace 资源的每个个体（Pod 与 Container 等）的资源配额。该插件和资源对象 LimitRange 一起实现资源限制管理。