Kubernetes 架构及内部原理

1. 概述

Kubernetes（常简称为 K8s）是一个开源平台，用于自动化部署、扩展和管理容器化应用程序。它提供了一个以容器为中心的管理环境，能够有效地处理工作负载和服务，并促进声明式配置和自动化。Kubernetes 的设计目标是提供一个健壮、可扩展且高可用的平台，以应对现代云原生应用的需求。

Kubernetes 集群由控制平面（Control Plane）和一组工作机器（称为节点，Node）组成，这些节点运行容器化应用程序。每个集群至少需要一个工作节点才能运行 Pod。工作节点托管构成应用程序工作负载的 Pod。控制平面管理集群中的工作节点和 Pod。在生产环境中，控制平面通常在多台计算机上运行，并且集群通常运行多个节点，以提供容错性和高可用性。

本文档将详细介绍 Kubernetes 的核心架构、各个组件的内部原理，并提供相关的代码示例及来源说明。

2. 控制平面组件 (Control Plane Components)

控制平面组件负责对集群做出全局决策（例如调度），以及检测和响应集群事件（例如，当 Deployment 的 `replicas` 字段未满足时启动一个新的 Pod）。控制平面组件可以在集群中的任何机器上运行。然而，为了简化，设置脚本通常会在同一台机器上启动所有控制平面组件，并且不在该机器上运行用户容器。

2.1. kube-apiserver

功能: API 服务器是 Kubernetes 控制平面的一个组件，它暴露 Kubernetes API。API 服务器是 Kubernetes 控制平面的前端。它是所有 RESTful 操作的中心枢纽，所有其他组件都通过它进行通信。

内部原理: `kube-apiserver` 设计为水平扩展，即通过部署更多实例来扩展。您可以运行多个 `kube-apiserver` 实例并在这些实例之间平衡流量。它负责验证和配置 API 对象的数据，例如 Pod、Service、Deployment 等。API Server 还会处理 REST 操作，并提供集群状态的持久化存储。

代码示例及来源: `kube-apiserver` 的源代码位于 Kubernetes 项目的 `cmd/kube-apiserver` 目录下。以下是其 `main.go` 文件的部分结构，展示了其启动和初始化过程：

```go
// cmd/kube-apiserver/apiserver.go
package main

import (
    "k8s.io/kubernetes/cmd/kube-apiserver/app"
    "k8s.io/component-base/cli"
)

func main() {
    cmd := app.NewAPIServerCommand()
    cli.Run(cmd)
}
```

此代码片段展示了 `kube-apiserver` 的入口点，它调用 `app.NewAPIServerCommand()` 来创建并配置 API 服务器的命令行命令，然后通过 `cli.Run()` 执行该命令。更详细的实现细节，包括 API 注册、请求处理、认证授权等，可以在 `k8s.io/kubernetes/cmd/kube-apiserver/app` 包中找到 [1]。

2.2. etcd

功能: `etcd` 是一个一致且高可用的键值存储，用作 Kubernetes 所有集群数据的后端存储。它存储了集群的配置数据、状态数据以及元数据。

内部原理: Kubernetes 使用 `etcd` 来存储集群的期望状态（desired state）和实际状态（actual state）。所有对集群状态的修改都通过 `kube-apiserver` 写入 `etcd`。`etcd` 采用 Raft 一致性算法来保证数据在分布式环境中的一致性和高可用性。如果您的 Kubernetes 集群使用 `etcd` 作为其后端存储，请务必制定数据备份计划 [1]。

代码示例及来源: `etcd` 是一个独立的开源项目，其源代码托管在 GitHub 上：`https://github.com/etcd-io/etcd`。Kubernetes 通过 `kube-apiserver` 与 `etcd` 进行交互，通常使用 `etcd` 客户端库。以下是一个简化的 `etcd` 客户端操作示例（非 Kubernetes 内部代码，但展示了与 `etcd` 交互的方式）：

```go
// 示例：使用 etcd 客户端库进行操作
package main

import (
    "context"
    "fmt"
    "go.etcd.io/etcd/client/v3"
    "time"
)

func main() {
    cli, err := clientv3.New(clientv3.Config{
```

```
        Endpoints:   []string{"localhost:2379"},
        DialTimeout: 5 * time.Second,
    })
    if err != nil {
        // handle error!
    }
    defer cli.Close()

    ctx, cancel := context.WithTimeout(context.Background(),
time.Second)
    _, err = cli.Put(ctx, "mykey", "myvalue")
    cancel()
    if err != nil {
        // handle error!
    }

    ctx, cancel = context.WithTimeout(context.Background(),
time.Second)
    resp, err := cli.Get(ctx, "mykey")
    cancel()
    if err != nil {
        // handle error!
    }
    for _, ev := range resp.Kvs {
        fmt.Printf("%s: %s\n", ev.Key, ev.Value)
    }
}
```

Kubernetes 内部与 `etcd` 的交互逻辑可以在 `k8s.io/apiserver/pkg/storage/etcd3` 包中找到，例如 `store.go` 文件中定义了如何将 API 对象存储到 `etcd` 中 [2]。

2.3. kube-scheduler

功能: 控制平面组件，负责监视新创建的没有分配节点的 Pod，并为它们选择一个节点来运行。

内部原理: 调度器在调度决策中考虑的因素包括：单个和集体资源需求、硬件/软件/策略约束、亲和性和反亲和性规范、数据局部性、工作负载间干扰以及截止日期。调度过程通常分为两个阶段：**过滤**（Filtering）和**评分**（Scoring）。过滤阶段会找出所有满足 Pod 调度要求的节点，评分阶段则会根据一系列预设的优先级函数对这些节点进行打分，最终选择得分最高的节点 [1]。

代码示例及来源: `kube-scheduler` 的源代码位于 Kubernetes 项目的 `cmd/kube-scheduler` 目录下。其核心调度逻辑在 `k8s.io/kubernetes/pkg/scheduler` 包中实现。以下是其 `main.go` 文件的部分结构：

```go
// cmd/kube-scheduler/scheduler.go
package main

import (
    "k8s.io/kubernetes/cmd/kube-scheduler/app"
    "k8s.io/component-base/cli"
)

func main() {
    cmd := app.NewSchedulerCommand()
    cli.Run(cmd)
}
```

调度器的具体实现，包括各种调度算法和策略，可以在 `k8s.io/kubernetes/pkg/scheduler/framework` 包中找到 [3]。

2.4. kube-controller-manager

功能: 控制平面组件，运行控制器进程。逻辑上，每个控制器都是一个独立的进程，但为了降低复杂性，它们都被编译成一个二进制文件并在一个进程中运行。

内部原理: 控制器管理器负责运行各种控制器，这些控制器通过 API Server 监视集群的共享状态，并尝试将当前状态更改为期望状态。例如：

- **节点控制器 (Node controller)**: 负责在节点出现故障时进行通知和响应。
- **Job 控制器 (Job controller)**: 监视代表一次性任务的 Job 对象，然后创建 Pod 来运行这些任务直到完成。
- **EndpointSlice 控制器 (EndpointSlice controller)**: 填充 EndpointSlice 对象（提供 Service 和 Pod 之间的链接）。
- **ServiceAccount 控制器 (ServiceAccount controller)**: 为新的命名空间创建默认的 ServiceAccount。

代码示例及来源: `kube-controller-manager` 的源代码位于 Kubernetes 项目的 `cmd/kube-controller-manager` 目录下。其 `main.go` 文件结构与 `kube-apiserver` 和 `kube-scheduler` 类似。各种控制器的具体实现可以在 `k8s.io/kubernetes/pkg/controller` 包中找到，例如 `node`、`job`、`serviceaccount` 等子目录 [4]。

2.5. cloud-controller-manager

功能: Kubernetes 控制平面组件，嵌入了云提供商特定的控制逻辑。云控制器管理器允许您将集群连接到云提供商的 API，并将与该云平台交互的组件与仅与集群交互的组件分离。

内部原理: `cloud-controller-manager` 只运行特定于您的云提供商的控制器。如果您在自己的本地环境或个人电脑上的学习环境中运行 Kubernetes，则集群没有云控制器管理器。

与 `kube-controller-manager` 一样，`cloud-controller-manager` 将几个逻辑上独立的控制循环组合成一个二进制文件，并作为一个进程运行。您可以水平扩展（运行多个副本）以提高性能或帮助容忍故障。以下控制器可能具有云提供商依赖性：

- **节点控制器**: 用于检查云提供商，以确定节点在停止响应后是否已在云中删除。
- **路由控制器**: 用于在底层云基础设施中设置路由。
- **服务控制器**: 用于创建、更新和删除云提供商负载均衡器。

代码示例及来源: `cloud-controller-manager` 的源代码位于 Kubernetes 项目的 `cmd/cloud-controller-manager` 目录下。其内部实现依赖于具体的云提供商，例如 AWS、GCP、Azure 等。相关代码可以在 `k8s.io/cloud-provider` 包中找到 [5]。

3. 节点组件 (Node Components)

节点组件在每个节点上运行，维护运行中的 Pod 并提供 Kubernetes 运行时环境。

3.1. kubelet

功能: 在集群中每个节点上运行的代理。它确保容器在 Pod 中运行。`kubelet` 接收通过各种机制提供的 PodSpec，并确保 PodSpec 中描述的容器正在运行且健康。`kubelet` 不管理非 Kubernetes 创建的容器。

内部原理: `kubelet` 通过与容器运行时（如 containerd、CRI-O）交互来管理 Pod 的生命周期，包括创建、启动、停止和删除容器。它还负责向 API Server 报告节点的状态、Pod 的状态以及资源使用情况。`kubelet` 还会定期执行健康检查，以确保容器的正常运行 [1]。

代码示例及来源: `kubelet` 的源代码位于 Kubernetes 项目的 `cmd/kubelet` 目录下。其核心逻辑在 `k8s.io/kubernetes/pkg/kubelet` 包中实现。以下是其 `main.go` 文件的部分结构：

```go
// cmd/kubelet/kubelet.go
package main

import (
    "k8s.io/kubernetes/cmd/kubelet/app"
    "k8s.io/component-base/cli"
)

func main() {
    cmd := app.NewKubeletCommand()
    cli.Run(cmd)
}
```

`kubelet` 的详细实现，包括 Pod 同步、容器管理、CRI 接口调用等，可以在 `k8s.io/kubernetes/pkg/kubelet/kubelet.go` 文件中找到 [6]。

3.2. kube-proxy (可选)

功能: `kube-proxy` 是一个网络代理，在集群中的每个节点上运行，实现 Kubernetes Service 概念的一部分。`kube-proxy` 维护节点上的网络规则。这些网络规则允许从集群内部或外部的网络会话与您的 Pod 进行网络通信。

内部原理: `kube-proxy` 使用操作系统的包过滤层（如果存在且可用）。否则，`kube-proxy` 会自行转发流量。它支持多种代理模式，如 `iptables`、`ipvs` 等，通过这些模式实现 Service 的负载均衡和网络转发。如果您使用的网络插件本身实现了 Service 的包转发，并提供了与 `kube-proxy` 等效的行为，则无需在集群节点上运行 `kube-proxy` [1]。

代码示例及来源: `kube-proxy` 的源代码位于 Kubernetes 项目的 `cmd/kube-proxy` 目录下。其核心逻辑在 `k8s.io/kubernetes/pkg/proxy` 包中实现。以下是其 `main.go` 文件的部分结构：

```go
// cmd/kube-proxy/proxy.go
package main

import (
    "k8s.io/kubernetes/cmd/kube-proxy/app"
    "k8s.io/component-base/cli"
)

func main() {
    cmd := app.NewProxyCommand()
    cli.Run(cmd)
}
```

`kube-proxy` 的具体实现，包括各种代理模式的切换和规则配置，可以在 `k8s.io/kubernetes/pkg/proxy/iptables` 或 `k8s.io/kubernetes/pkg/proxy/ipvs` 等子包中找到 [7]。

3.3. 容器运行时 (Container Runtime)

功能: 容器运行时是使 Kubernetes 能够有效运行容器的基础组件。它负责管理 Kubernetes 环境中容器的执行和生命周期。

内部原理: Kubernetes 支持多种容器运行时，例如 `containerd`、`CRI-O` 以及任何其他实现 Kubernetes CRI (Container Runtime Interface) 的运行时。容器运行时负责从镜像仓库拉取容器镜像、解压镜像、创建容器沙箱、运行容器进程、管理容器的资源隔离（CPU、内存、

网络等）以及容器的生命周期事件。CRI 是 Kubernetes 和容器运行时之间通信的接口规范 [1]。

代码示例及来源: 容器运行时是独立的软件项目，例如 `containerd` 的源代码位于 `https://github.com/containerd/containerd`，`CRI-O` 的源代码位于 `https://github.com/cri-o/cri-o`。Kubernetes 通过 CRI 接口与这些运行时进行交互。CRI 的接口定义可以在 `k8s.io/cri-api` 包中找到 [8]。

4. 插件 (Addons)

插件使用 Kubernetes 资源（DaemonSet、Deployment 等）来实现集群功能。由于这些插件提供集群级别的功能，因此插件的命名空间资源属于 `kube-system` 命名空间。

4.1. DNS

功能: 尽管其他插件并非严格必需，但所有 Kubernetes 集群都应该有集群 DNS，因为许多示例都依赖它。集群 DNS 是一个 DNS 服务器，除了您环境中的其他 DNS 服务器外，它还为 Kubernetes 服务提供 DNS 记录。Kubernetes 启动的容器会自动将此 DNS 服务器包含在其 DNS 搜索中。

内部原理: Kubernetes DNS 通常由 CoreDNS 或 Kube-DNS 实现。它通过监视 Kubernetes Service 和 Pod 的创建、更新和删除事件，动态地生成 DNS 记录。当 Pod 需要解析 Service 名称时，它会向集群 DNS 服务器发送请求，集群 DNS 服务器会返回 Service 对应的 IP 地址，从而实现服务发现 [1]。

代码示例及来源: CoreDNS 的源代码位于 `https://github.com/coredns/coredns`。Kubernetes 中 CoreDNS 的部署配置通常通过 YAML 文件定义，例如：

```yaml
# 示例：CoreDNS Deployment 部分配置
apiVersion: apps/v1
kind: Deployment
metadata:
  name: coredns
  namespace: kube-system
spec:
  template:
    spec:
      containers:
      - name: coredns
        image: k8s.gcr.io/coredns/coredns:1.8.0
        args: [".", "-conf", "/etc/coredns/Corefile"]
        volumeMounts:
        - name: config-volume
          mountPath: /etc/coredns
      volumes:
```

```
        - name: config-volume
          configMap:
            name: coredns
```

此示例展示了 CoreDNS 作为 Deployment 部署在 `kube-system` 命名空间中，并挂载了一个 ConfigMap 作为其配置文件。CoreDNS 的具体实现和插件机制可以在其官方文档和源代码中找到 [9]。

4.2. Web UI (Dashboard)

功能: Dashboard 是一个通用的、基于 Web 的 Kubernetes 集群 UI。它允许用户管理和排除集群中运行的应用程序以及集群本身的问题。

内部原理: Kubernetes Dashboard 通过与 `kube-apiserver` 交互来获取集群的各种资源信息（如 Pod、Deployment、Service、Node 等），并以可视化的方式展示给用户。它还提供了一些基本的操作功能，如部署应用程序、扩展工作负载、查看日志等。Dashboard 通常通过 Service 和 Ingress 对外暴露 [1]。

代码示例及来源: Kubernetes Dashboard 的源代码位于 `https://github.com/kubernetes/dashboard`。其部署通常通过官方提供的 YAML 文件进行：

```
# 示例：部署 Kubernetes Dashboard
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
```

此命令会部署 Dashboard 及其相关的 ServiceAccount、ClusterRoleBinding 等资源。Dashboard 的前端和后端实现细节可以在其 GitHub 仓库中找到 [10]。

4.3. 容器资源监控 (Container Resource Monitoring)

功能: 容器资源监控将容器的通用时间序列指标记录到中央数据库中，并提供一个用于浏览这些数据的 UI。

内部原理: Kubernetes 生态系统中常用的监控解决方案是 Prometheus 和 Grafana。Prometheus 负责从各个组件（如 `kubelet`、`cAdvisor`、`Node Exporter` 等）收集指标数据，并将其存储在时间序列数据库中。Grafana 则负责从 Prometheus 中查询数据，并以图表的形式进行可视化展示，提供丰富的仪表盘功能 [1]。

代码示例及来源: Prometheus 的源代码位于 `https://github.com/prometheus/prometheus`，Grafana 的源代码位于 `https://github.com/grafana/grafana`。Kubernetes 中部署 Prometheus 和 Grafana 通常使用 Prometheus Operator 或 Helm Charts。以下是一个简化的 Prometheus 配置示例：

```yaml
# 示例：Prometheus ServiceMonitor 部分配置
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: kube-apiserver
  namespace: default
spec:
  endpoints:
  - port: https
    scheme: https
    tlsConfig:
      insecureSkipVerify: true
    bearerTokenFile: /var/run/secrets/kubernetes.io/
serviceaccount/token
  selector:
    matchLabels:
      k8s-app: kube-apiserver
```

此示例展示了 Prometheus Operator 如何通过 ServiceMonitor 资源来发现并抓取 `kube-apiserver` 的指标。Prometheus 和 Grafana 的详细配置和使用方法可以在其官方文档中找到 [11], [12]。

4.4. 集群级别日志 (Cluster-level Logging)

功能: 集群级别日志机制负责将容器日志保存到具有搜索/浏览界面的中央日志存储中。

内部原理: 常见的 Kubernetes 日志解决方案是 ELK Stack (Elasticsearch, Logstash, Kibana) 或 EFK Stack (Elasticsearch, Fluentd, Kibana)。日志代理（如 Fluentd 或 Filebeat）部署在每个节点上，负责收集容器的日志文件，并将其发送到 Elasticsearch 进行存储和索引。Kibana 提供了一个 Web 界面，用于搜索、分析和可视化日志数据 [1]。

代码示例及来源: Elasticsearch 的源代码位于 `https://github.com/elastic/elasticsearch`，Logstash 的源代码位于 `https://github.com/elastic/logstash`，Kibana 的源代码位于 `https://github.com/elastic/kibana`。Fluentd 的源代码位于 `https://github.com/fluent/fluentd`。以下是一个简化的 Fluentd DaemonSet 配置示例：

```yaml
# 示例：Fluentd DaemonSet 部分配置
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
spec:
  template:
    spec:
```

```
      containers:
      - name: fluentd
        image: fluent/fluentd-kubernetes-daemonset:v1.16-debian-
elasticsearch8
        env:
          - name: FLUENT_ELASTICSEARCH_HOST
            value: "elasticsearch.default.svc.cluster.local"
          - name: FLUENT_ELASTICSEARCH_PORT
            value: "9200"
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

此示例展示了 Fluentd 作为 DaemonSet 部署在每个节点上，并挂载了宿主机的日志目录以收集容器日志。ELK/EFK Stack 的详细配置和使用方法可以在其官方文档中找到 [13], [14], [15]。

4.5. 网络插件 (Network plugins)

功能: 网络插件是 Kubernetes 集群网络功能的核心。它们负责为 Pod 提供网络连接，并实现 Pod 之间的通信以及 Pod 与外部世界的通信。

内部原理: Kubernetes 采用 CNI (Container Network Interface) 规范来管理网络插件。CNI 插件负责为 Pod 分配 IP 地址、配置网络接口、设置路由规则以及实现网络策略。常见的 CNI 插件包括 Calico、Flannel、Cilium 等。这些插件通常通过 DaemonSet 部署在每个节点上，以确保每个节点都具备网络功能 [1]。

代码示例及来源: CNI 规范的源代码位于 `https://github.com/containernetworking/cni`。各种 CNI 插件的源代码可以在其各自的 GitHub 仓库中找到，例如 Calico 的源代码位于 `https://github.com/projectcalico/calico`。以下是一个简化的 Calico DaemonSet 配置示例：

```
# 示例：Calico DaemonSet 部分配置
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: calico-node
```

```
    namespace: kube-system
spec:
  template:
    spec:
      containers:
      - name: calico-node
        image: calico/node:v3.26.1
        env:
          - name: CALICO_NODENAME
            valueFrom:
              fieldRef:
                fieldPath: spec.nodeName
          - name: CALICO_IP_AUTODETECTION_METHOD
            value: "can-reach=8.8.8.8"
        securityContext:
          privileged: true
        volumeMounts:
        - name: lib-modules
          mountPath: /lib/modules
          readOnly: true
        - name: var-run-calico
          mountPath: /var/run/calico
      volumes:
      - name: lib-modules
        hostPath:
          path: /lib/modules
      - name: var-run-calico
        hostPath:
          path: /var/run/calico
```

此示例展示了 Calico 作为 DaemonSet 部署在每个节点上，并配置了必要的环境变量和卷挂载。网络插件的详细配置和使用方法可以在其官方文档中找到 [16]。

5. 总结

Kubernetes 通过其精心设计的控制平面和节点组件，以及丰富的插件生态系统，提供了一个强大而灵活的容器编排平台。控制平面负责集群的全局管理和决策，而节点组件则负责容器的实际运行和维护。各种插件则进一步扩展了 Kubernetes 的功能，使其能够满足各种复杂的应用场景需求。理解这些组件的内部原理对于有效管理和排查 Kubernetes 集群问题至关重要。

6. 参考文献

[1] Kubernetes 官方文档. Cluster Architecture. https://kubernetes.io/docs/concepts/architecture/ [2] Kubernetes GitHub 仓库. `k8s.io/apiserver/pkg/storage/etcd3`. https://github.com/kubernetes/apiserver/tree/master/pkg/storage/etcd3 [3] Kubernetes GitHub 仓库. `k8s.io/kubernetes/pkg/scheduler/framework`. https://github.com/

kubernetes/kubernetes/tree/master/pkg/scheduler/framework [4] Kubernetes GitHub 仓库. `k8s.io/kubernetes/pkg/controller`. https://github.com/kubernetes/kubernetes/tree/master/pkg/controller [5] Kubernetes GitHub 仓库. `k8s.io/cloud-provider`. https://github.com/kubernetes/cloud-provider [6] Kubernetes GitHub 仓库. `k8s.io/kubernetes/pkg/kubelet/kubelet.go`. https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/kubelet.go [7] Kubernetes GitHub 仓库. `k8s.io/kubernetes/pkg/proxy`. https://github.com/kubernetes/kubernetes/tree/master/pkg/proxy [8] Kubernetes GitHub 仓库. `k8s.io/cri-api`. https://github.com/kubernetes/cri-api [9] CoreDNS GitHub 仓库. https://github.com/coredns/coredns [10] Kubernetes Dashboard GitHub 仓库. https://github.com/kubernetes/dashboard [11] Prometheus GitHub 仓库. https://github.com/prometheus/prometheus [12] Grafana GitHub 仓库. https://github.com/grafana/grafana [13] Elasticsearch GitHub 仓库. https://github.com/elastic/elasticsearch [14] Logstash GitHub 仓库. https://github.com/elastic/logstash [15] Kibana GitHub 仓库. https://github.com/elastic/kibana [16] Calico GitHub 仓库. https://github.com/projectcalico/calico