Minik8s 验收报告

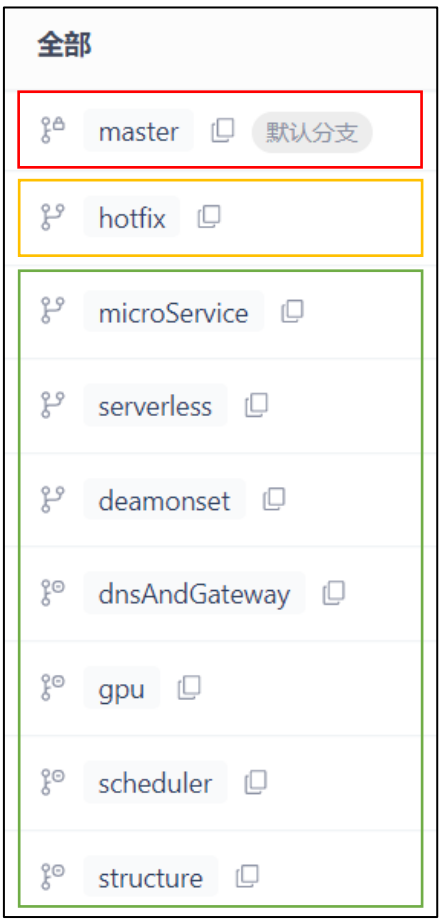
Minik8s 是一个类似于 Kubernetes 的迷你容器编排工具，能够在多机上对满足 CRI 接口的容器进行管理。支持容器生命周期管理、动态伸缩、自动扩容等基本功能，并且基于 Minik8s 实现了 Serverless 平台集成。

一、 成员分工及贡献比例

姓名	分工	贡献度
杨景凯（组长）	Service 抽象，DNS 与转发，Microservice	33%
彭逸帆	Pod 抽象，ReplicaSet 抽象，多机部署与调度，控制面容错，CI/CD	33%
张世昊	动态伸缩，GPU 应用，Serverless	33%

二、 项目开发过程简介

- Gitee 仓库地址：https://gitee.com/ethereal-O/k8s.git
- 仓库共有 9 个分支：



- 所有分支可分为三类：

1) **master 分支**是 Minik8s 的稳定版本，可以随时发布。 master 分支不接受 push，只接受来自 hotfix 分支的并入。并且只有 master 分支会触发 CI/CD。

2) **hotfix 分支**接受各个功能性分支的并入请求。由于各个功能在集成时可能出现新的问题，因此需要在 hotfix 分支上快速解决这些问题，待确认无误后会并入 master 分支。

3) 其余分支如 serverless, gpu 都是实现并测试某个具体功能的**功能性分支**，最终会并入 hotfix 分支。

- 以下是一个完整的工作流，以 serverless 功能的开发为例：

1) 以 master 分支为基线，创建 serverless 分支。

2) 在 serverless 分支上进行开发和单元测试。

3) 通过 PR 将 serverless 分支并入 hotfix 分支。

4) 拉取 hotfix 分支，检查 serverless 功能与其他功能的集成是否存在问题。

5) 如果有问题，在 hotfix 分支上进行修改。

6) 确认无误后，通过 PR 将 hotfix 分支并入 master 分支。

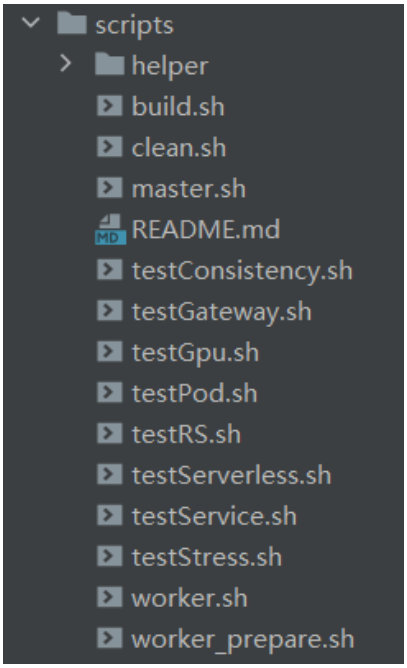
7) master 分支成为新的基线。

- 项目采用的软件测试方法：

1) **代码走查**：在每周组会中，开发某一功能的组员需要向其余两位组员逐条讲解代码并描述功能的设计、框架、实现思路。其余两位组员对其中的细节和可能存在的问题进行质询，从而保证软件质量。

2) **同行审查**：在分支合并的过程中，开发某一功能的组员需要指派另一名组员作为审查人员和测试人员，从而对功能的实现和缺陷进行进一步的测试和检查，提高软件质量。

3) **回归测试**：在 hotfix 分支上，开发某一功能的组员需要运行以往的所有测试脚本，以保证各个功能在集成时没有出现新的问题。

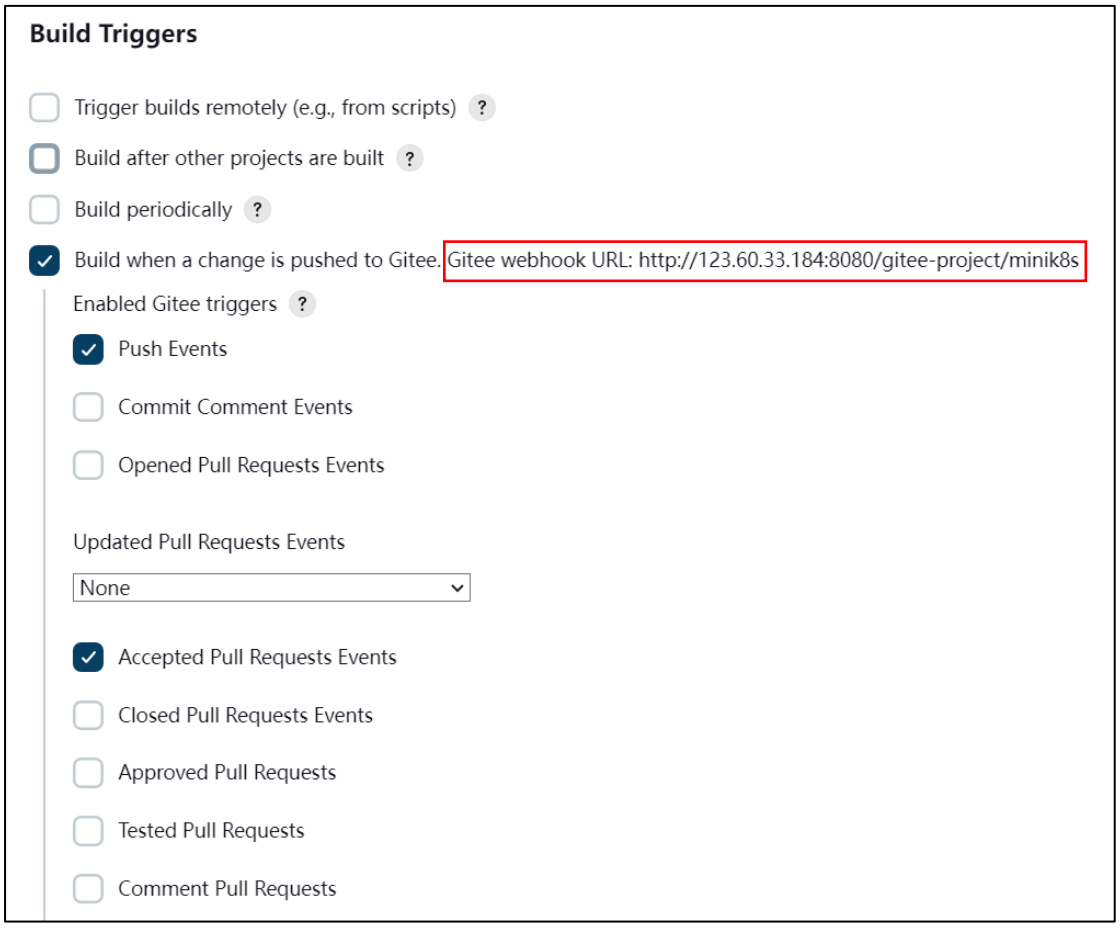


- CI/CD 的实现方式：

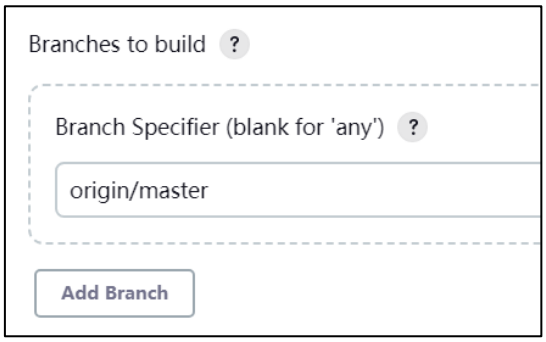
  本项目采用 Jenkins 实现 CI/CD，Jenkins 部署在一个华为云服务器上，其公网 URL 为 http://123.60.33.184:8080/。

  以下是一个完整的 CI/CD 工作流：

1) 用户通过 PR 将 hotfix 分支并入 master 分支。
2) master 分支发生更新，Gitee 向 Jenkins 的 WebHook 发送 POST 请求，通知 Jenkins 拉取最新代码并开始构建。



注意到这里我们设置了只有 master 分支才会触发构建。这就保证了我们的服务器上运行的 Minik8s 始终是稳定版本。



3) Jenkins 根据脚本把最新代码部署到三台服务器上。

名称	镜像	内网IP	浮动IP
minik8s-3		192.168.1.6	10.119.11.138
minik8s-2		192.168.1.4	10.119.11.128
minik8s-1		192.168.1.9	10.119.11.46

Build Steps

Execute shell ?

Command

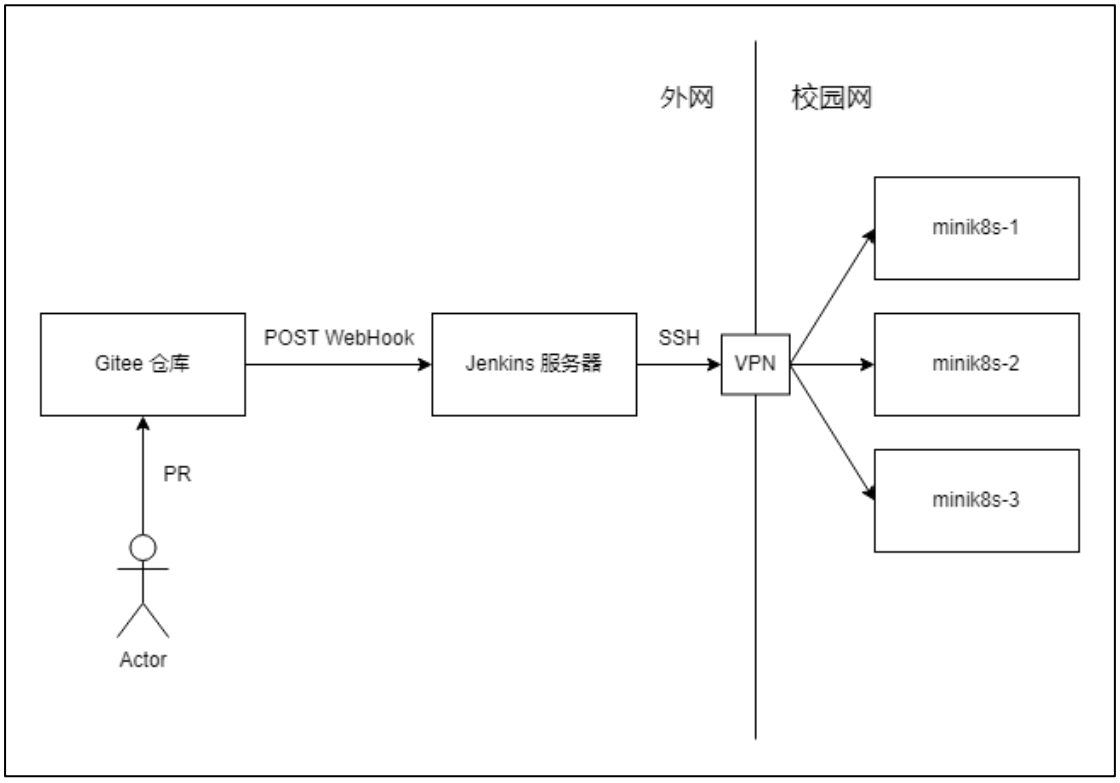
See **the list of available environment variables**

```
#!/bin/bash
SOURCE_DIR=/root/.jenkins/workspace/${JOB_NAME}/
DEST_DIR=/root/minik8s/
NODE_IP=("10.119.11.46" "10.119.11.128" "10.119.11.138")
MASTER_IP="192.168.1.9"

# Loop through the IP addresses and perform rsync and ssh commands
for ip in "${NODE_IP[@]}"; do
  rsync -e "ssh -p 22" -avpgolr --exclude=.git "$SOURCE_DIR" "root@$ip:$DEST_DIR"
  ssh "root@$ip" "echo -n '$MASTER_IP' > /root/minik8s/master_ip.txt"
done
```

由于三台服务器使用的是校园网，而 Jenkins 所在的华为云服务器在校外，所以华为云服务器需要连接交大 VPN。这也是我们把 Jenkins 部署在华为云服务器上的原因，通过这个服务器连接外网（Gitee）和校园网（三台服务器）。

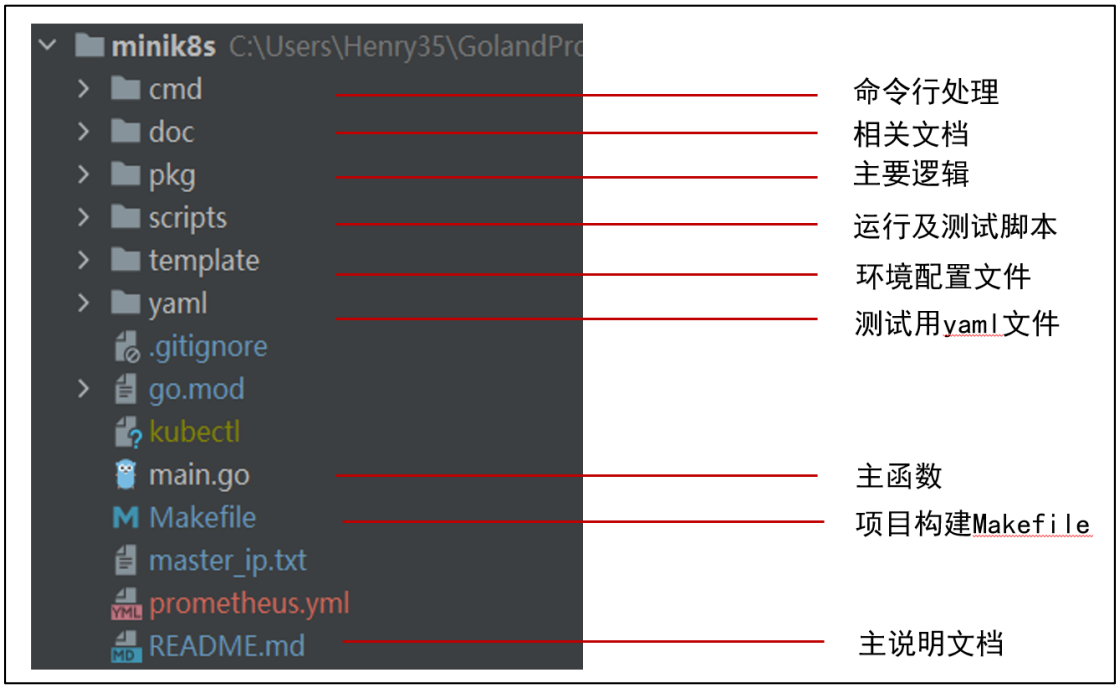
```
establishing CHILD_SA sjtu-student{19008}
generating CREATE_CHILD_SA request 36 [ SA No TSi TSr ]
sending packet: from 192.168.0.174[4500] to 111.186.48.0[4500] (336 bytes)
received packet: from 111.186.48.0[4500] to 192.168.0.174[4500] (320 bytes)
parsed CREATE_CHILD_SA response 36 [ SA No TSi TSr ]
selected proposal: ESP:AES_CBC_128/HMAC_SHA2_256_128/NO_EXT_SEQ
CHILD_SA sjtu-student{19008} established with SPIs c7def0f8_i caa97faf_o and TS 111.186.48.
5/32 2001:da8:8000:7100::2:5/128 === 0.0.0.0/0 2000::/3
connection 'sjtu-student' established successfully
--------------------------------------------------------------------------------
★[2023-06-04 16:30:01] Successful
--------------------------------------------------------------------------------
```

整体架构图如下:

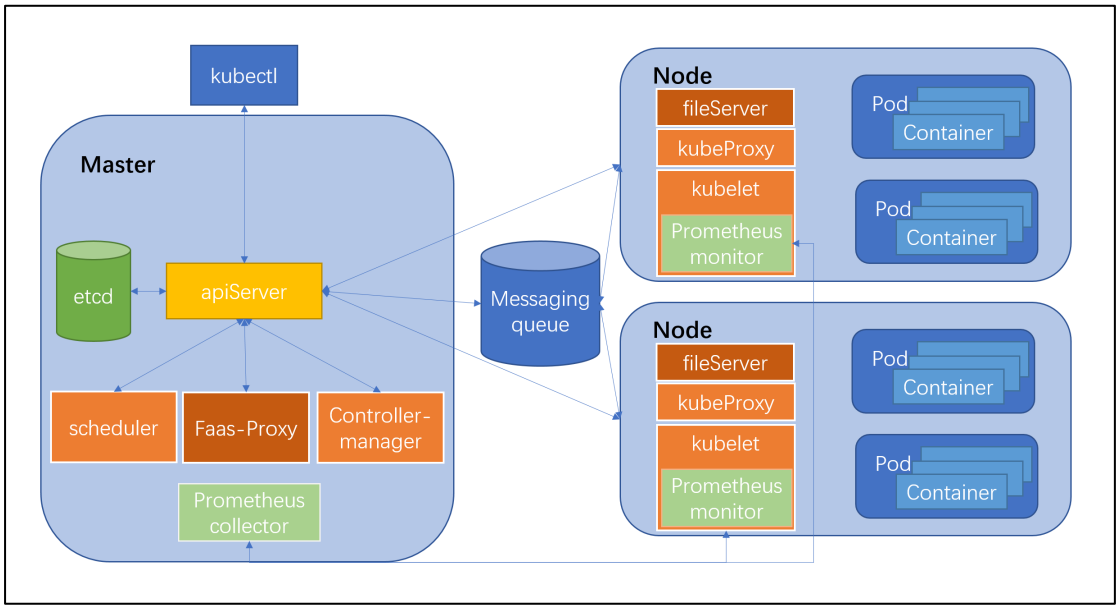


三、 项目整体架构与软件栈

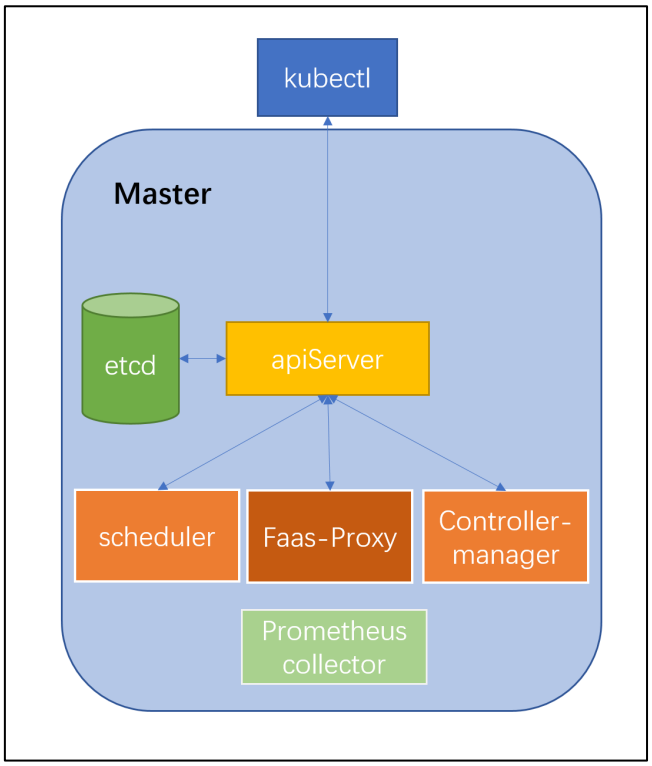
- 项目的文件结构:



- 项目的整体架构:

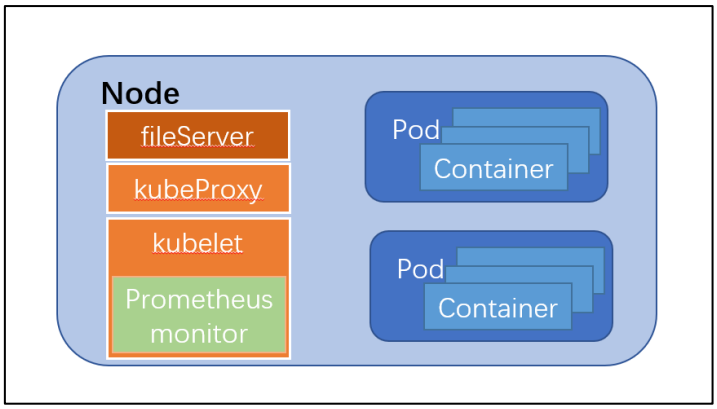


- Master 节点上的组件及其作用:



1) etcd：存储集群的所有需要持久化的数据。
2) apiServer：集群操作和 CRUD 的入口，采用现有 echo 框架，支持 restful 服务。
3) scheduler: 负责集群资源调度, 当前支持的策略有 round-robin 和根据 Node 资源以及 Pod 反亲和性调度。
4) Controller-manager：负责维护集群的实际状态与预期状态一致。
5) Faas-Proxy：负责接受并转发 serverless 的请求，支持流量统计，缓存。
6) Prometheus collector：通过 Prometheus 采集 Node 的监控数据。

- Worker 节点上的组件及其作用：

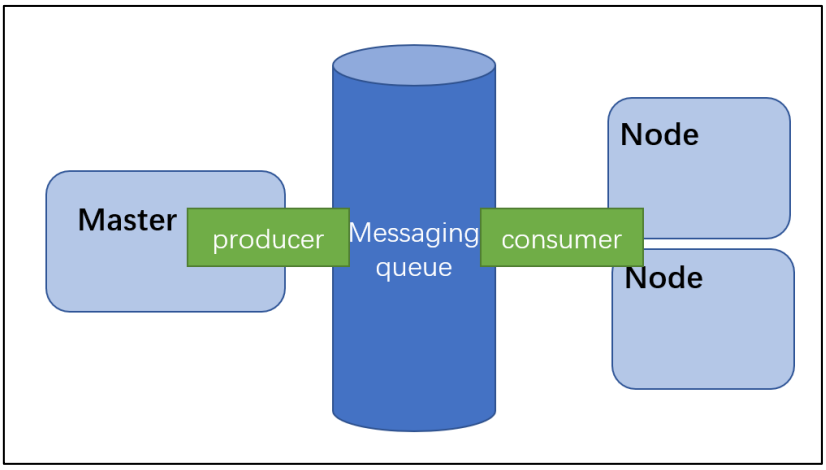


1) kubelet：负责维护 Pod 和容器的生命周期，负责 Node 的启动，维护与删除，同时也通过内置的 Prometheus monitor 向 Master 节点提供 Node 和 Pod 的资源使用情况。

2) kubeProxy：负责提供集群内部的服务发现和负载均衡，目前提供三套服务方案：基于 Nginx 的反向代理与负载均衡，基于 Iptables 的请求转发，基于 Sidecar 的拦截分发。

3) fileServer：负责节点之间的文件传输，包括 GPU 配置文件，CUDA 文件，函数文件等。

- 节点间的消息传递机制——消息队列：



Producer 运行在 Master 节点上，负责上传数据。Consumer 运行在 Worker 节点上，负责接收数据。不同节点共享 topic，但拥有独立的 channel。

- 项目使用的软件栈与开源组件：

1) 数据存储：etcd

2) 消息队列：nsq

3) Web 框架：echo

4) 容器运行时：Docker

5) 指标监控：Prometheus

6) CNI 插件：weave 和 flannel

四、 项目的所有功能、使用方法和实现方式

1. 多机部署与调度

在视频 1-2_NodeAndPod_V2.mp4 中已经介绍了如何通过 kubectl 启动 Master 和 Worker 节点，以及如何查看节点的运行情况。

但是因为时间所限，没有演示节点的环境初始化过程，因此将在文档中加以补充：

1) 首先需要保证服务器上已经安装了 Docker，weave 和 flannel。

2) 在每个服务器上，修改 master_ip.txt 的内容为 Master 服务器的 IP 地址。

（事实上在 CI/CD 的部署脚本中已经完成了这一步）

Build Steps

Execute shell ?

Command

See **the list of available environment variables**

```
#!/bin/bash
SOURCE_DIR=/root/.jenkins/workspace/${JOB_NAME}/
DEST_DIR=/root/minik8s/
NODE_IP=("10.119.11.46" "10.119.11.128" "10.119.11.138")
MASTER_IP="192.168.1.9"

# Loop through the IP addresses and perform rsync and ssh commands
for ip in "${NODE_IP[@]}"; do
  rsync -e "ssh -p 22" -avpgolr --exclude=.git "$SOURCE_DIR" "root@$ip:$DEST_DIR"
  ssh "root@$ip" "echo -n '$MASTER_IP' > /root/minik8s/master_ip.txt"
done
```

3) 在 Master 服务器上运行 make master，等待初始化完毕。

```
root@minik8s-1:~/minik8s# make master
[Cleaner] K8S Master/Worker stopped!
[Cleaner] Weave subnet stopped!
[Cleaner] Flannel net stopped!
[Cleaner] DNS data cleared!
[Cleaner] GPU and Serverless data cleared!
[Cleaner] Containers cleared!
[Cleaner] Command log cleared!
[Cleaner] Prometheus config cleared!
[Cleaner] All states cleared!
[Master] ETCD started!
[Master] Prometheus started!
[Master] NSQ producer started!
[Master] Control plane started!
[Master] Init finished!
[Worker] DNS config created!
[Worker] Weave subnet started!
[Worker] Worker node started!
[Worker] Init finished!
```

4) 在 Worker 服务器上运行 make worker，等待初始化完毕。

```
root@minik8s-3:~/minik8s# make worker
[Cleaner] K8S Master/Worker stopped!
[Cleaner] Weave subnet stopped!
[Cleaner] Flannel net stopped!
[Cleaner] DNS data cleared!
[Cleaner] GPU and Serverless data cleared!
[Cleaner] Containers cleared!
[Cleaner] Command log cleared!
[Cleaner] Prometheus config cleared!
[Cleaner] All states cleared!
[Worker] DNS config created!
[Worker] Weave subnet started!
[Worker] NSQ consumer started!
[Worker] Worker node started!
[Worker] Init finished!
```

5) 接下来就可以通过 kubectl 来启动和查看 Node 了。

```
root@minik8s-1:~/minik8s# ./kubectl apply -f ./yaml/PodTest/node1.yaml
root@minik8s-1:~/minik8s# ./kubectl apply -f ./yaml/PodTest/node3.yaml
root@minik8s-1:~/minik8s# ./kubectl get -t  Node
Type: Node
+-----------+-------+---------+-------------+-----------+
|   Name    | Uuid  | Status  |  PublicIP   | ClusterIP |
+-----------+-------+---------+-------------+-----------+
| minik8s-1 | 10001 | RUNNING | 192.168.1.9 | 10.10.0.1 |
| minik8s-3 | 10008 | RUNNING | 192.168.1.6 | 10.10.0.2 |
+-----------+-------+---------+-------------+-----------+
```

在 make master 和 make worker 的过程中，我们启动了一些基础设施，例如 etcd, nsq, Prometheus 等。这些基础设施都实现了容器化，如此用户只需要安装 Docker 和 CNI 插件即可运行 Minik8s，十分方便。

```
# Start etcd
docker run -d \
  --env ALLOW_NONE_AUTHENTICATION=yes \
  --env ETCD_ENABLE_V2=true \
  --env ETCDCTL_API=2 \
  --network=host \
  --name etcd \
  bitnami/etcd \
  > /dev/null 2>&1
sleep 1
docker exec etcd \
  etcdctl --endpoints http://127.0.0.1:2379 set /coreos.com/network/config '{"Network": "1
  > /dev/null 2>&1
echo "[Master] ETCD started!" 1>&2

# Start prometheus
docker run -d \
  --network=host \
  --name prometheus \
  -v "$(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml" \
  prom/prometheus --config.file=/etc/prometheus/prometheus.yml --web.enable-lifecycle \
  > /dev/null 2>&1
echo "[Master] Prometheus started!" 1>&2

# Start nsq-producer (nsqlookupd + nsqadmin + nsqd)
docker run -d \
  --network=host \
  --name nsq-producer \
  -v "$(pwd)/scripts/helper/nsq_producer.sh:/nsq_producer.sh" \
  nsqio/nsq \
  sh /nsq_producer.sh \
  > /dev/null 2>&1
echo "[Master] NSQ producer started!" 1>&2
```

在 Node 启动后，可以通过 kubectl delete 命令将 Node 删除。这样会使该 Node 上的所有 Pod 也一并被删除。

```go
func DeleteNode() bool {  2个用法
    // Step 0: Check if my node has started
    myNode := getMyNode()
    if myNode == nil {
        return true
    }

    // Step 1: Stop probe cycle
    NodeToExit <- true
    <-NodeExited

    // Step 2: Delete all pods
    for _, pod := range client.GetActivePods() {
        if pod.Runtime.Bind == myNode.Metadata.Name {
            pod.Runtime.Status = config.EXIT_STATUS
            client.AddPod(pod)
            <-PodDeleted[pod.Runtime.Uuid] // Wait for pod delete
            delete(PodDeleted, pod.Runtime.Uuid)
        }
    }

    // Step 3: Stop current node
    myNode.Runtime.Status = config.EXIT_STATUS
    client.AddNode(*myNode)
    fmt.Printf("[Kubelet] Node #{myNode.Metadata.Name} deleted!\n")
    return true
}
```

```
root@minik8s-1:~/minik8s# ./kubectl delete -t Node -k minik8s-1
root@minik8s-1:~/minik8s# ./kubectl get -t Node
Type: Node
+------+------+--------+----------+-----------+
| Name | Uuid | Status | PublicIP | ClusterIP |
+------+------+--------+----------+-----------+
```

然后是多机调度的实现。首先 Scheduler 要筛选出可选的 Node，在本项目中，我们考虑了 nodeSelector 和 CPU/内存资源的限制：

```go
func getOptionalNodes(pod *object.Pod) []object.Node {  1 个用法
    nodes := client.GetActiveNodes()

    var optional_nodes []object.Node
    for _, node := range nodes {
        // Filter by labels
        labelOk := true
        for key, value := range pod.Spec.NodeSelector {
            nodeValue, ok := node.Metadata.Labels[key]
            if !ok || value != nodeValue {
                labelOk = false
                break
            }
        }
        if !labelOk {
            continue
        }

        // Filter by resources
        cpu := prometheus_query( tp: "cpu", node.Runtime.Uuid)
        mem := prometheus_query( tp: "memory", node.Runtime.Uuid)
        for _, container := range pod.Spec.Containers {
            cpu -= float64(resource.ConvertCpuToBytes(container.Limits.Cpu))
            mem -= float64(resource.ConvertMemoryToBytes(container.Limits.Memory))
        }
        if cpu > 0 && mem > 0 {
            optional_nodes = append(optional_nodes, node)
        }
    }

    return optional_nodes
}
```

然后要从可选的 Node 中挑选出最佳的 Node，我们实现了两种调度策略:

1) Round Robin 策略

按顺序依次选择每个可选的 Node。

```go
func (policy RRPolicy) selectNode(pod *object.Pod, nodes []object.Node) string {  1 个用法
    // No optional nodes
    if len(nodes) == 0 { return "" }
    return nodes[counter.GetRRPolicy()%len(nodes)].Metadata.Name
}
```

2) 综合评分策略

Kubelet 内置的 Prometheus monitor 可以向 Master 节点提供 Node 的资源使用情况。因此 Scheduler 可以根据 Node 的空闲资源量进行打分，在本项目中，**每有 1 个空闲的 CPU 核计 10 分，每有 1GB 空闲内存计 5 分**。除此以外，为了鼓励让 ReplicaSet 的 Pod 均匀分布到不同 Node 上，我们给没

有 replica 的 Node **额外奖励 50%的分数**。

```go
func (policy ScoringPolicy) selectNode(pod *object.Pod, nodes []object.Node) string {  1 个用法
    // No optional nodes
    if len(nodes) == 0 { return "" }

    max_score := 0.0
    max_score_node := ""

    for _, node := range nodes {
        cpu := prometheus_query( tp: "cpu", node.Runtime.Uuid)
        mem := prometheus_query( tp: "memory", node.Runtime.Uuid)
        cpu_score := cpu / 1e9 * 10.0       // 1 CPU = 10 score
        mem_score := mem / (1024 * 1024 * 1024) * 5.0 // 1 GB Memory = 5 score

        rs_scale := 1.5
        // pod belongs to a ReplicaSet
        if pod.Runtime.Belong != "" {
            pods := client.GetActivePods()
            for _, existing_pod := range pods {
                // An existing pod is bound to the node and is from the same ReplicaSet
                if existing_pod.Runtime.Belong == pod.Runtime.Belong && existing_pod.Runtime.Bind == node.Metadata.Name {
                    rs_scale = 1.0 // ReplicaSet should have pods on various nodes
                }
            }
        }

        score := (cpu_score + mem_score) * rs_scale
        if score > max_score {
            max_score = score
            max_score_node = node.Metadata.Name
        }
    }

    fmt.Printf("[Scheduler] Select Node #{max_score_node}, score = #{max_score}\n")
    return max_score_node
}
```

2. Pod 抽象

验收指南中与 Pod 相关的验收要求已经全部在视频 1-2_NodeAndPod_V2.mp4 中介绍过了，这里补充一些实现细节：

- Pod 内部是怎么共享 Network Namespace 的？

  和 K8S 的实现方式类似，每个 Pod 内部除了用户定义的容器外，还包括一个 PAUSE 容器，其他容器都与 PAUSE 容器共享 Network Namespace。由此实现 Pod 内部共享 Network Namespace。

```go
func CreatePauseContainer(pod *object.Pod) (string, string, error) {  1个用法
    podName := pod.Metadata.Name
    podUuid := pod.Runtime.Uuid

    // Step 1: Prepare for labels
    labels := map[string]string{}
    for labelName, labelValue := range pod.Metadata.Labels {
        labels[labelName] = labelValue
    }

    // Step 2: All the containers share network namespace with pause container
    portBindings := nat.PortMap{}
    portSet := nat.PortSet{}
    for _, c := range pod.Spec.Containers {
        addPortBindings(portBindings, c.Ports)
        addPortSet(portSet, c.Ports)
    }

    // Step 3: Finally create the container!
    name := pauseContainerFullName(podName, podUuid)
    ID, err := CreateContainer(name, &CreateConfig{
        // Config
        Image:        pauseImage,
        Labels:       labels,
        Volumes:      nil,
        ExposedPorts: portSet,

        // HostConfig
        IpcMode:      "shareable",
        PortBindings: portBindings,
        Binds:        nil,
        DNS:          []string{config.DNS_SERVER},
    })
    return name, ID, err
}
```

通常情况下，PAUSE 容器的端口绑定就是其他容器的端口绑定的并集。但是在 ReplicaSet 的 Pod Spec 中不能指定 HostPort（否则可能会端口冲突），所以如果没有指定 HostPort，就使用随机的空闲端口作为 HostPort。

```go
func addPortBindings(portBindings nat.PortMap, ports []object.Port) {  1 个用法
    for _, port := range ports {
        // Protocol not assigned, default is tcp
        if port.Protocol == "" {
            port.Protocol = "tcp"
        }
        // HostIP not assigned, default is "0.0.0.0"
        if port.HostIP == "" {
            port.HostIP = "0.0.0.0"
        }
        // HostPort not assigned, default is random available port
        if port.HostPort == 0 {
            randomPort, err := network.GetAvailablePort()
            if err != nil {

            }
            fmt.Printf("Using random available port #{randomPort}\n")
            port.HostPort = randomPort
        }

        // Finally bind them!
        containerPort, err := nat.NewPort(port.Protocol, strconv.Itoa(port.ContainerPort))
        if err != nil {

        }
        portBindings[containerPort] = []nat.PortBinding{{
            HostIP:   port.HostIP,
            HostPort: strconv.Itoa(port.HostPort),
        }}
    }
}
```

- Pod 的 Cluster IP 是如何分配的?

  Pod 的 Cluster IP 是由一个 atomic counter 进行分配的, 起始值为 10.10.1.1。

```go
func (counter *Counter) fetchAndAdd() int {  7 个用法
    counter.lock.Lock()
    defer counter.lock.Unlock()

    ret := etcd.Get_etcd(counter.url,  withPrix: false)

    if len(ret) == 0 {
        etcd.Set_etcd(counter.url, counter.initCount)
        ret = append(ret, counter.initCount)
    }

    num, _ := strconv.Atoi(ret[0])
    etcd.Set_etcd(counter.url, strconv.Itoa(num+1))
    return num
}
```

  在分配完 Cluster IP 后，还需要执行 weave attach <PAUSE 容器 ID> <Cluster IP/Mask>来将这个 Pod 加入到 weave 子网。然后在集群中就可以用<Cluster IP : containerPort>来访问 Pod 内部的容器了。

```
// Step 4: Attach to weave subnet
err = weave.Attach(ID, pod.Runtime.ClusterIp+network.Mask)
if err != nil {
    fmt.Printf("Failed to attach pause container #{fullName} (ID: #{ID}) to subnet! Reason: #{err.Error()}\n")
    return false, ""
} else {
    fmt.Printf("Pause Container #{fullName} (ID: #{ID}) attached to subnet!\n")
}
```

- Pod 内的容器异常退出后是如何重启的?

  Kubelet 在启动完一个 Pod 后, 会再启动一个协程用来监测 Pod 的运行状态, 如果发现 Pod 内的容器异常退出了，就会触发 PodException 函数。

```
func PodProbeCycle(pod *object.Pod) {  1 个用法
    for {
        select {
        case <-PodToExit[pod.Runtime.Uuid]:
            PodExited[pod.Runtime.Uuid] <- true
            return
        default:
            time.Sleep(1 * time.Second)
            var containerMemoryPercentageList []float64
            var containerCpuPercentageList []float64

            for _, containerId := range pod.Runtime.Containers {
                inspection, err := Client.ContainerInspect(Ctx, containerId)
                if err != nil {
                    // Container does not exist, restart the pod!
                    fmt.Println( a...: "[delete because not exist]")
                    PodException(pod)
                    return
                }
                status, err := inspectionToContainerRuntime(&inspection)
                if err != nil { err *
                if status.State == StateExited {
                    // Container has exited, restart the pod!
                    fmt.Println( a...: "[delete because exited]")
                    PodException(pod)
                    return
                }
```

PodException 函数会将 pod.Runtime.NeedRestart 设为 true，由此让 Kubelet 重启这个 Pod。但如果这个 Pod 属于一个 ReplicaSet 或者 DaemonSet 就不需要重启了（只删除不重启），因为 ReplicaSet 或者 DaemonSet 的 controller 会自行管理它们的 Pod。

```go
func PodException(pod *object.Pod) {  2个用法
    pod.Runtime.NeedRestart = true
    // If the pod belongs to an RS/DS, no need to restart, because RS/DS will do it automatically
    if pod.Runtime.Belong != "" {
        pod.Runtime.NeedRestart = false
    }
    // If the pod belongs to a gpujob, no need to restart
    if strings.HasPrefix(strings.ToLower(pod.Metadata.Name),  prefix: "gpujob") {
        pod.Runtime.NeedRestart = false
    }
    pod.Runtime.Status = config.EXIT_STATUS
    client.AddPod(*pod)

    // Wait for DeletePod()
    <-PodToExit[pod.Runtime.Uuid]
    PodExited[pod.Runtime.Uuid] <- true
}
```

3. Service 抽象

验收指南中与 Service 相关的功能要求已经全部在视频 3-4_RSAndService_V3_part1.mp4 与视频 3-4_RSAndService_V3_part2.mp4 中介绍过了，这里补充一下 Service 的 yaml 文件以及 Service 的具体实现。

```yaml
kind: Service
metadata:
  name: Service1
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: http
  selector:
    app: myApp
```

首先，Service 分为三种模式。三者在 Master 端的处理是具有相同之处的。

1) Master 端处理共同点：

Master 首先会监听 Service 的注册请求，并产生相应的 RuntimeService 对象，每个对象通过 ticker，不断轮询 Pod 的状态，以更新 Pod 实例。为了更好地泛化函数，我们使用了 Go 泛型实现了一些基本的 slice 库，例如 filter 等。同时，为了区分是 Service 的动态更新或用户手动更新还是 etcd 的 at least once 策略，我们使用 Go 泛型，利用哈希来区别，实

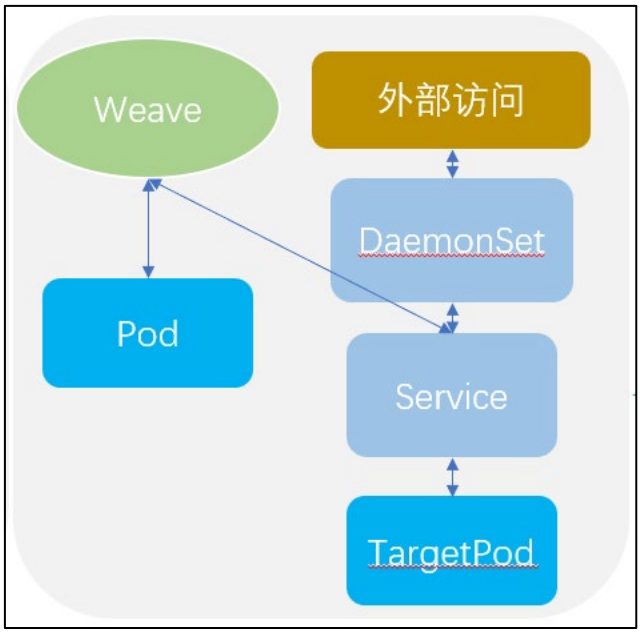
现 Service 版本迭代与更新。

```go
func Filter[T any](slice []T, condition func(T) bool) ([]T, []T) {
    var filtered []T
    var differed []T
    for _, item := range slice {
        if condition(item) {
            filtered = append(filtered, item)
        } else {
            differed = append(differed, item)
        }
    }
    return filtered, differed
}
```

```go
func MD5[T any](origin T) (result string) {
    transfer, err := json.Marshal(origin)
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    res := md5.Sum(transfer)
    result = hex.EncodeToString(res[:])
    return result
}
```

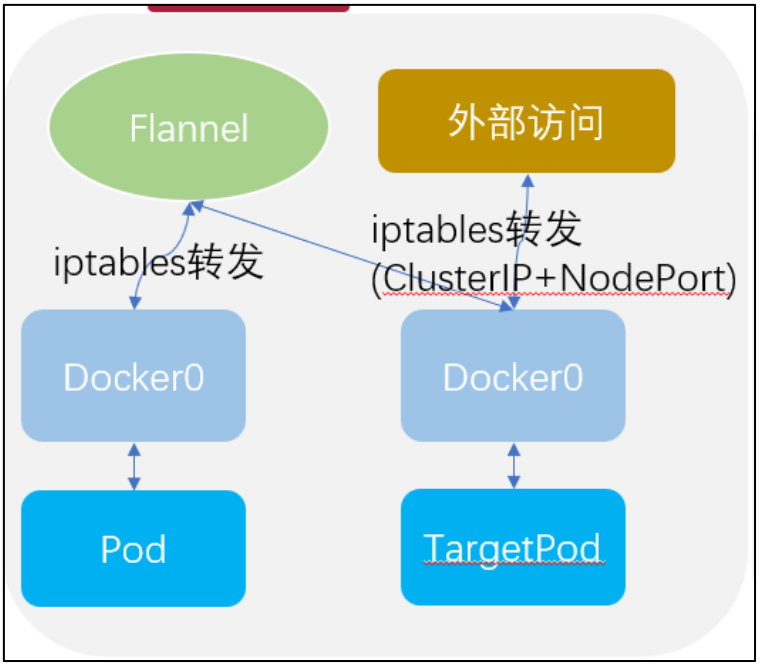
2) Nginx 模式：

Nginx 模式是目前的默认模式，这是因为其能够完全实现容器化，与主机实现更好的解耦，同时清理与维护更加方便。对于每个 Service 的 ClusterIP 模式，我们会产生一个 Nginx 容器，这个容器与主机建立 Volume 映射，这样我们可以直接修改主机的文件并执行 reload 即可更新。我们通过 weave attach 将 Service 的 ClusterIP 赋予给 Nginx 容器，这样即可通过 weave 网桥来找到 Service。而对于 NodePort 模式，则通过 DaemonSet 产生的 Nginx 来实现，思路类似，不再赘述。值得注意的是，为了能够从 weave 网卡发送数据，ClusterIP 必须在 weave 的子网内。



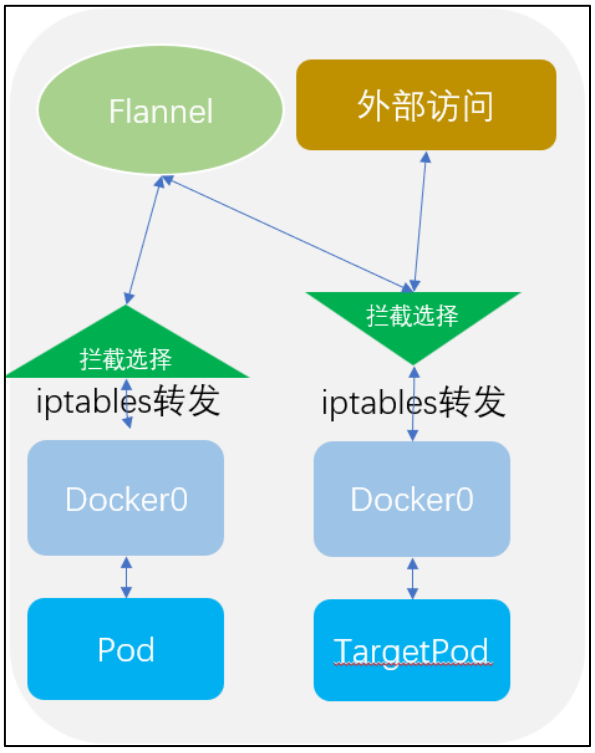
3) Iptables 模式：

Iptables 模式不需要启动 Nginx 容器，因此启动速度快。这次需要维护 Iptables 链表，分为 KUBE_PROXY_PARENT 链表，Service-xxx 的 single-service 链表，Pod-xxx 的 single-pod 链表，以及位于 single-pod 链表下的 NAT 规则。分别作为根链表(插入到 OUTPUT 与 POSTROUTING 下)、负责一个 Service 的一个端口、负责一个 Pod、负责一条转发规则。在 Service-xxx 链表下，实现了负载均衡。对于 NodePort 模式，只需要多在根链表下插入一条指向 single-service 的规则即可。值得注意的是，这次必须通过 Docker 网桥发送信息，因此 ClusterIP 必须不在 weave 的子网内。



4) Microservice 模式:

此部分放在 10. Microservice 中讲解。



5) 对比:

我们对比了 Nginx 模式下与 Iptables 模式下的访问时延。本来以为是 Nginx 模式会明显慢于 Iptables 模式, 但是两者大体类似, 分别为 0.13ms

与 0.12ms（基于 1000 次测试）。这是因为通过 Nginx 模式是使用 weave 的网络，而 weave 是两层的网络，而 flannel 是三层，并且跨主机的访问需要访问 master 的 etcd 服务来确定子 Docker 网络的目的地址，而 etcd 由于目前的容器化部署在 Docker 内，因此大体抵消掉了 Nginx 的转发。

6) 其他：

为了使得 Worker 节点的启动与 Service 的配置时间无关（即可先配置 Service 再启动 Worker 节点），在启动时会获取并更新所有其他节点已经部署完成的 Service。

4. ReplicaSet 抽象

验收指南中与 ReplicaSet 相关的验收要求已经全部在视频 3-4_RSAndService_V3_part1.mp4 中介绍过了，这里补充一下 ReplicaSet 的具体实现。

ReplicaSet 的核心功能就是保证有指定数量的 Pod 在运行，因此 ReplicaSet Controller 需要定期检查集群中的 Pod 数量，如果 Pod 数量不足就启动新的 Pod，如果 Pod 数量过多就删除部分 Pod。

```go
func RSCycle(reName string) {  1个用法
    for {
        select {
        case <-RSToExit[reName]:
            RSExited[reName] <- true
            return
        default:
            rs := client.GetReplicaSetByKey(reName)[0]
            time.Sleep(1 * time.Second)
            targetNum := rs.Spec.Replicas
            rspodList, actualNum := object.GetPodsOfRS(&rs, client.GetActivePods())
            if targetNum > actualNum {
                for i := 0; i < targetNum-actualNum; i++ {
                    var newPod object.Pod
                    uuid := counter.GetUuid()
                    newPod.Runtime.Uuid = uuid
                    newPod.Kind = config.POD_TYPE
                    newPod.Runtime.Belong = rs.Metadata.Name
                    newPod.Metadata = rs.Spec.Template.Metadata
                    newPod.Metadata.Name = object.RSPodFullName(&rs, &newPod)
                    newPod.Spec = rs.Spec.Template.Spec
                    client.AddPod(newPod)
                }
            } else if targetNum < actualNum {
                for i := targetNum; i < actualNum; i++ {
                    client.DeletePod(rspodList[i])
                }
            }
        }
    }
}
```

通过 kubectl get 可以看到 ReplicaSet 的预期 replica 数和实际 replica 数。

```
root@minik8s-1:~/minik8s# ./kubectl get -t ReplicaSet
Type: ReplicaSet
+---------------+-------+---------+----------+---------------+---------------------------------+
|     Name      | Uuid  | Status  | Replicas | ActualReplicas |              Pods               |
+---------------+-------+---------+----------+---------------+---------------------------------+
|    DNS-RS     | 10007 | RUNNING |    1     |       1       |          DNS-RS_10009           |
|     RS1       | 10012 | RUNNING |    3     |       3       | RS1_10013, RS1_10014, RS1_10015 |
| SvcRS-DNS-Svc | 10010 | RUNNING |    1     |       1       |       SvcRS-DNS-Svc_10011       |
+---------------+-------+---------+----------+---------------+---------------------------------+
```

由于 Scheduler 的综合评分策略给没有 replica 的 Node 额外奖励 50%的分数，所以 ReplicaSet 的 Pod 一般可以均匀分布到不同 Node 上。

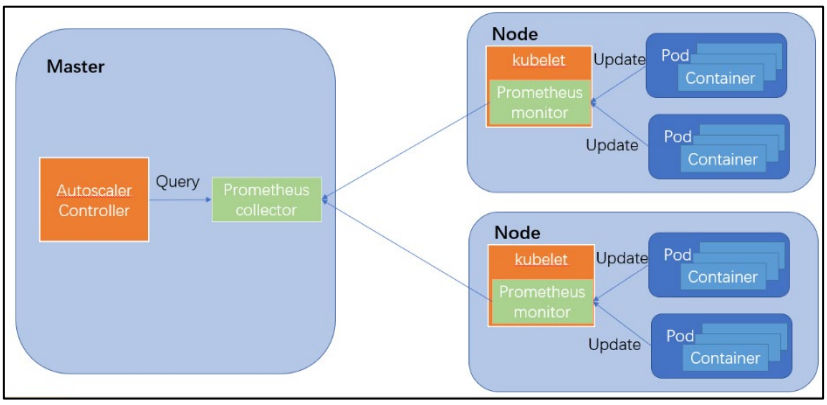
```
root@minik8s-1:~/minik8s# ./kubectl get -t Pod
Type: Pod
+----------------------+-------+---------+---------------+-----------+-----------+
|         Name         | Uuid  | Status  |    Belong     |   Bind    | ClusterIP |
+----------------------+-------+---------+---------------+-----------+-----------+
|     DNS-RS_10009     | 10009 | RUNNING |    DNS-RS     | minik8s-2 | 10.10.1.1 |
| Forward-DS_minik8s-1 | 10002 | RUNNING |  Forward-DS   | minik8s-1 | 10.10.0.1 |
| Forward-DS_minik8s-2 | 10004 | RUNNING |  Forward-DS   | minik8s-2 | 10.10.0.2 |
| Forward-DS_minik8s-3 | 10006 | RUNNING |  Forward-DS   | minik8s-3 | 10.10.0.3 |
|      RS1_10013       | 10013 | RUNNING |      RS1      | minik8s-3 | 10.10.1.3 |
|      RS1_10014       | 10014 | RUNNING |      RS1      | minik8s-2 | 10.10.1.4 |
|      RS1_10015       | 10015 | RUNNING |      RS1      | minik8s-1 | 10.10.1.5 |
| SvcRS-DNS-Svc_10011  | 10011 | RUNNING | SvcRS-DNS-Svc | minik8s-2 | 10.10.1.2 |
+----------------------+-------+---------+---------------+-----------+-----------+
```

然后除了 ReplicaSet, 我们还额外实现了 DaemonSet 抽象。通过 DaemonSet, 我们可以在集群中的每个 Node 上运行一个 Pod，这样可以支持基于 Nginx 策略的 NodePort Service。（这里的 hostMode: true 表明这个 Pod 应该与主机共享 Network Namespace）

```yaml
kind: DaemonSet
metadata:
  name: Forward-DS
  labels:
    bound: Forward
spec:
  template:
    metadata:
      name: Forward-Pod
      labels:
        bound: Forward
    spec:
      hostMode: true
      containers:
        - name: Forward-Cont
          image: nginx
          ports:
            - containerPort: 80
          volumeMounts:
            - name: Forward-Volume
              mountPath: /etc/nginx
      volumes:
        - name: Forward-Volume
          type: hostPath
          path: /home/os/minik8s/Forward
```

```
root@minik8s-1:~/minik8s# ./kubectl get -t DaemonSet
Type: DaemonSet
+------------+-------+---------+-------------------------------------------------------------------------+
|    Name    | Uuid  | Status  |                                  Pods                                   |
+------------+-------+---------+-------------------------------------------------------------------------+
| Forward-DS | 10000 | RUNNING | Forward-DS_minik8s-1, Forward-DS_minik8s-2, Forward-DS_minik8s-3 |
+------------+-------+---------+-------------------------------------------------------------------------+

root@minik8s-1:~/minik8s# ./kubectl get -t Pod
Type: Pod
+---------------------+-------+---------+--------------+-----------+-----------+
|        Name         | Uuid  | Status  |    Belong    |   Bind    | ClusterIP |
+---------------------+-------+---------+--------------+-----------+-----------+
|     DNS-RS_10009    | 10009 | RUNNING |    DNS-RS    | minik8s-2 | 10.10.1.1 |
| Forward-DS_minik8s-1 | 10002 | RUNNING |  Forward-DS  | minik8s-1 | 10.10.0.1 |
| Forward-DS_minik8s-2 | 10004 | RUNNING |  Forward-DS  | minik8s-2 | 10.10.0.2 |
| Forward-DS_minik8s-3 | 10006 | RUNNING |  Forward-DS  | minik8s-3 | 10.10.0.3 |
| SvcRS-DNS-Svc_10011 | 10011 | RUNNING | SvcRS-DNS-Svc | minik8s-2 | 10.10.1.2 |
+---------------------+-------+---------+--------------+-----------+-----------+
```

5. 动态伸缩



动态伸缩主要基于 Autoscaler-Controller 实现，使用了 Prometheus 作为指标监控的工具。

每个 Worker 节点启动 Prometheus 监控服务器并暴露 9080 端口供 Master 节点监控。Worker 上每个 Pod 启动后就会创建一个协程，固定时间间隔将 Pod 的资源使用情况更新到该 Worker 的资源监控服务器中。本项目中支持的监控指标为 CPU 和内存，采用的指标类型为 Gauge，指标的关键字段为资源的类型（此处为 Pod），资源的唯一标识符 UUID 以及其监控指标。

Master 节点需要配置并从对应的 Worker 节点资源监控服务器中获取信息。它本身在 9090 端口运行 Prometheus 的监控服务，并在配置文件中配置监控的目标Node节点的9080端口作为目标。Autoscaler-Controller会为每个Autoscaler

单元运行一个协程，每隔一段时间自动根据 Autoscaler 的配置，查看其管理的 ReplicaSet 的 Pod，并通过 UUID 以及监控指标作为关键字段，查看资源占用情况。然后根据策略实时扩缩容，对 ReplicaSet 的副本数进行更新，之后 RsController 会自动对 Pod 进行控制。

6. DNS 与转发

验收指南中与 DNS 与转发相关的验收要求已经全部在视频 6-7_DnsAndGateway_V2.mp4 中介绍过了，这里补充一下 DNS 与转发的具体实现。

DNS 与转发基于运行 Nginx 和 CoreDNS 的 Pod，在 Master 启动时就会启动相应的 ReplicaSet 与 Service。在 Worker 启动时，会获取 DNS 的状态并更新 Nginx，CoreDNS 映射的文件与主机的 Hosts 文件，使得 DNS 与转发和 Worker 的启动顺序无关。

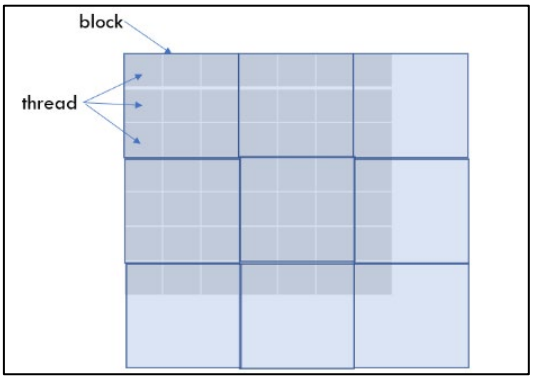
7. 控制面容错

由于 etcd 存储了集群的所有需要持久化的数据，而 etcd 本身是高容错的，所以不需要进行太多的额外处理即可实现控制面容错，效果可见视频 6-7_DnsAndGateway_V2.mp4 的末尾。

但有一点需要注意：**atomic counter 的值需要被持久化**，否则控制面重启后创建的 Pod 会分配到重复的 Cluster IP。

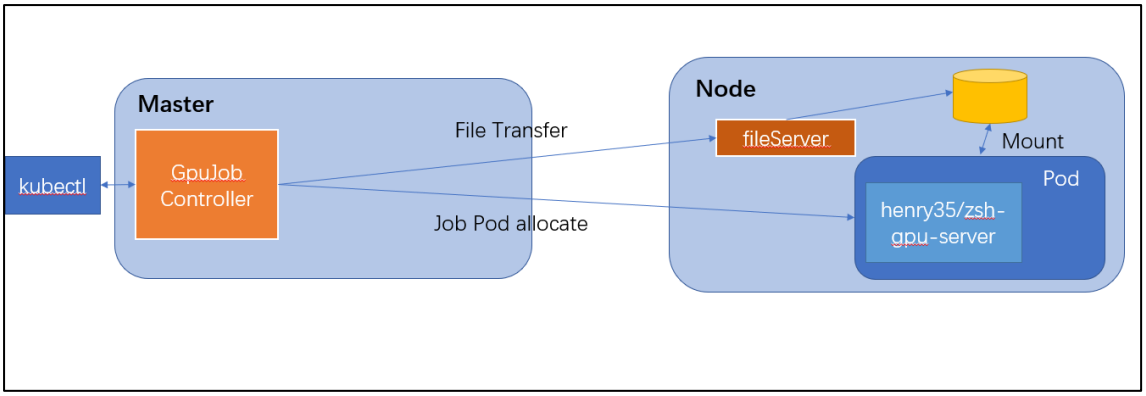
```
func (counter *Counter) fetchAndAdd() int {  7 个用法
    counter.lock.Lock()
    defer counter.lock.Unlock()

    ret := etcd.Get_etcd(counter.url,  withPrix: false)

    if len(ret) == 0 {
        etcd.Set_etcd(counter.url, counter.initCount)
        ret = append(ret, counter.initCount)
    }

    num, _ := strconv.Atoi(ret[0])
    etcd.Set_etcd(counter.url, strconv.Itoa(num+1))
    return num
}
```

8. GPU 应用



GPU 矩阵加法与矩阵乘法的并行化思路：

首先将待计算的矩阵按一定大小分为若干个 block 块，不足的地方采用空白补齐。每个 block 块可以并行计算。同时，每个 block 内又分为了若干 thread，每个 thread 可以计算矩阵中的一个单元，他们也可以并行计算。需要注意的是，在计算时，采用了一维数组保存与传递矩阵，因此要进行矩阵实际二维坐标与传递数组一维坐标的变换，并忽略补足的部分。

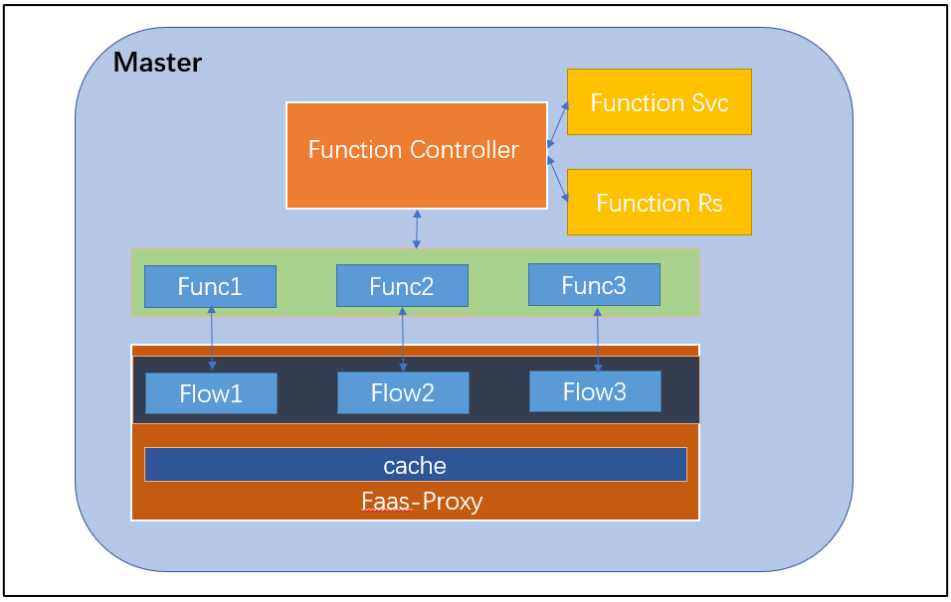


GPU 的具体实现：

当用户注册 GPU 的任务后，便会交由 GpuJob-Controller 管理。其主要负责将 GPU 的相关文件通过 fileServer 传递到对应的 Node 节点，并且在该节点上创建一个包含专用于处理 GPU 计算任务的容器的 Pod。该 Pod 会挂载 GPU 的相关文件并进行任务处理。GpuJob-Controller 之后便会通过监控指定任务 Pod 来更新 Job 的状态以及任务结果。

此处负责处理 GPU 应用任务的镜像为自制镜像 henry35/zsh-gpu-server:4.0，其基于 Ubuntu 基础镜像创建，并且包含了 sshd 等基础服务。该容器启动后便会执行 server.sh 脚本，通过 ssh 与 scp 向超算服务器传输计算文件并等待获取计算结果，执行完毕后容器便会自动终止。此外，加入了任务文件发送失败，结果文件获取超时，文件更新等边界情况的处理。

9. Serverless



注册函数后，Function-Controller 便会基于 fileServer 进行函数文件的传输，并且进行函数 Pod 的注册——此处并不是直接启动 Pod，而是为每个函数注册 ReplicaSet 以及 Service。ReplicaSet 的初始副本数为 0，后续根据请求调整副本数；Service 则能起到负载均衡的作用。此外，Function-Controller 也会为每个已注册的函数运行一个协程，它会根据对应 Service 和 ReplicaSet 的状态动态更改函数的运行时状态。

Function Proxy 为每个已注册的函数运行一个协程，来动态管理函数的流量。当对函数的请求到来时，它会统一被 Function Proxy 拦截并进行代理，增加流量，并根据流量进行 Function 副本的扩容；同理，当长期没有请求到来时，也会自动将副本数降至 0。此外，它自身也能通过缓存机制更快速的返回目标函数的结果。Function Proxy 支持单请求也支持 work-flow。当 work-flow 到达时，它会自动根据 DAG 执行循环和路径选择，最后将结果返回用户。

此处负责处理函数应用任务的镜像为自制镜像 henry35/serverless:2.0,其基于 Ubuntu 基础镜像创建，并且包含了 python，pip 等基础运行环境。该容器启动后便会自动执行 import.sh 脚本，读取配置文件并导入相应的依赖包。之后通过

flask 启动内置 server，根据请求的函数动态导入模块和相应函数进行编译。

10. Microservice

架构图在 3. Service 抽象中。与预期文档要求不同，我们并不是通过向 Master 提交请求来开启或关闭 Sidecar，而是直接修改 Service 模式，这样做的原因一方面是因为我们的附加功能并不是 Microservice，另一方面通过向 Master 提交请求会修改所有已经配置好的 Service，因此直接修改 Service 的模式会是一个比较方便且代码复用较多的选择。首先通过 Iptables 完成对所有规则的配置，将其转发至特定端口（目前是 15001 与 15006），允许特例 group_id 为 0 的用户（即 root）穿透规则。然后监听端口，使用 socket 复制完成转发。通过 VirtualService 抽象，可以定义其转发分配，进而实现灰度发布。

VirtualService 的 yaml 文件如下所示。

```
kind: VirtualService
metadata:
  name: VirtualService1
spec:
  type: Exact
  service: Service1
  selector:
    - matchLabels:
        version: v2
      weight: 9
    - matchLabels:
        version: v1
      weight: 1
```

通过 labels 的选择，可以进一步对 Service 进行划分，实现同 Service 下的不同 label 的 Pod 具有不同的分配比例。当前可以使用的 VirtualService 模式有 Exact 模式（完全匹配 selector）、Regular 模式（通过正则匹配）、Prefix 模式（通过前缀匹配）。

通过 Sidecar 拦截器输出的计数，我们可以发现已经正确的完成了流量按比例控制。

```
$ ./scripts/testMicroService.sh
[TestMicroService] Pod1 and Pod2 started!
[TestMicroService] Service1 started!
[TestMicroService] VirtualService1 started!
[TestMicroService] Testing curl once...
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
[TestMicroService] Testing curl 100 times...
     92 selecting ip 172.17.15.6 for service 100.100.20.1
      9 selecting ip 172.17.15.5 for service 100.100.20.1
```