**Minik8s 验收报告**

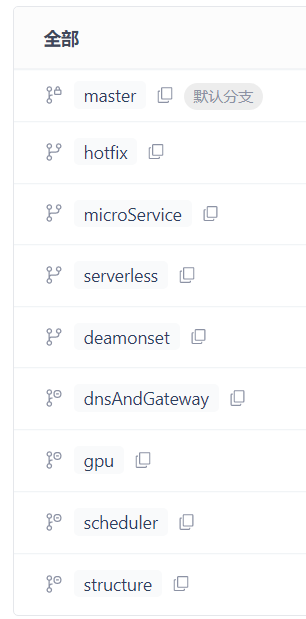
Minik8s 是一个类似于 Kubernetes 的迷你容器编排工具，能够在多机上对满足 CRI 接口的容器进行管理。支持容器生命周期管理、动态伸缩、自动扩容等基本功能，并且基于 Minik8s 实现了 Serverless 平台集成。

1. **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% |

1. **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：

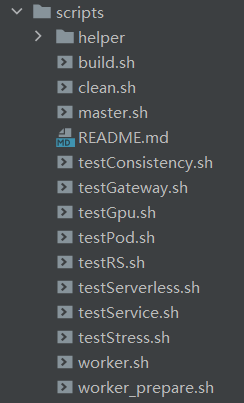
1. 以master分支为基线，创建serverless分支。
2. 在serverless分支上进行开发和单元测试。
3. 通过PR将serverless分支并入hotfix分支。
4. 拉取hotfix分支，检查serverless功能与其他功能的集成是否存在问题。
5. 如果有问题，在hotfix分支上进行修改。
6. 确认无误后，通过PR将hotfix分支并入master分支。
7. master分支成为新的基线。

* 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。



1. **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。



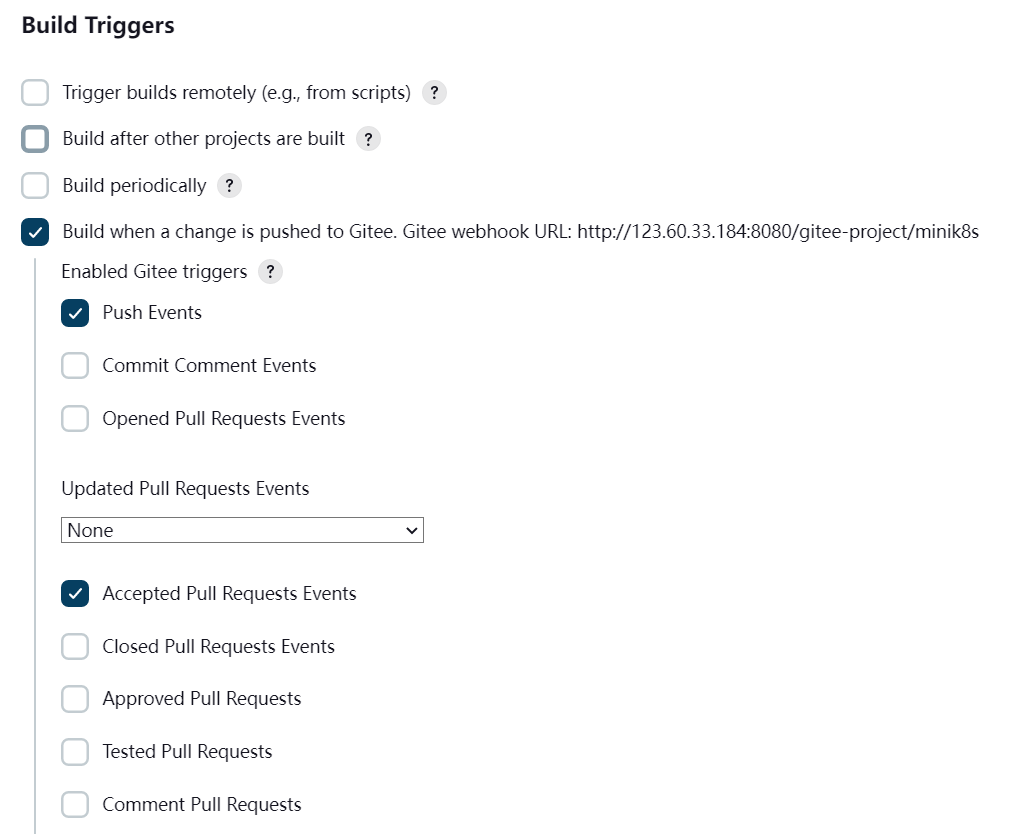
* CI/CD的实现方式：

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/。

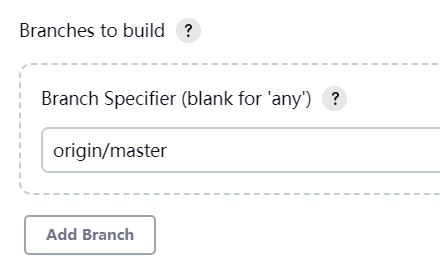
以下是一个完整的CI/CD工作流：

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。



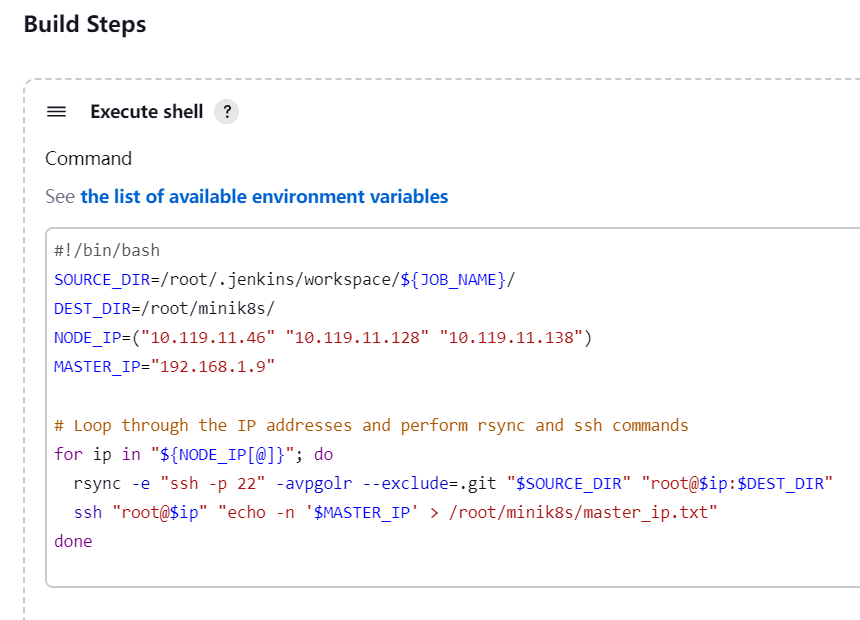


注意到这里我们设置了只有master分支才会触发构建。这就保证了我们的服务器上运行的Minik8s始终是稳定版本。

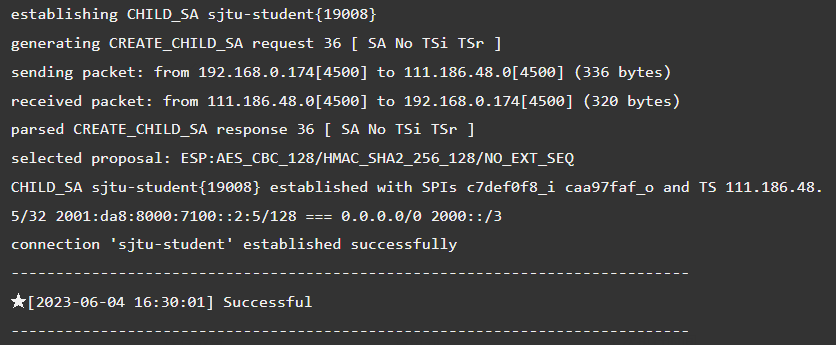


1. Jenkins根据脚本把最新代码部署到三台服务器上。

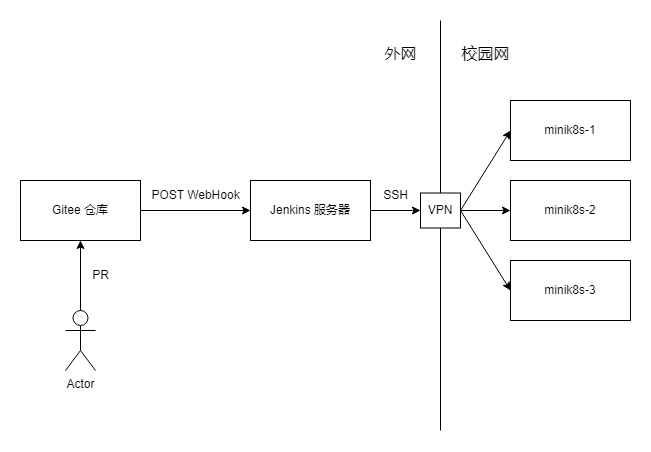




由于三台服务器使用的是校园网，而Jenkins所在的华为云服务器在校外，所以华为云服务器需要连接交大VPN。这也是我们把Jenkins部署在华为云服务器上的原因，通过这个服务器连接外网（Gitee）和校园网（三台服务器）。

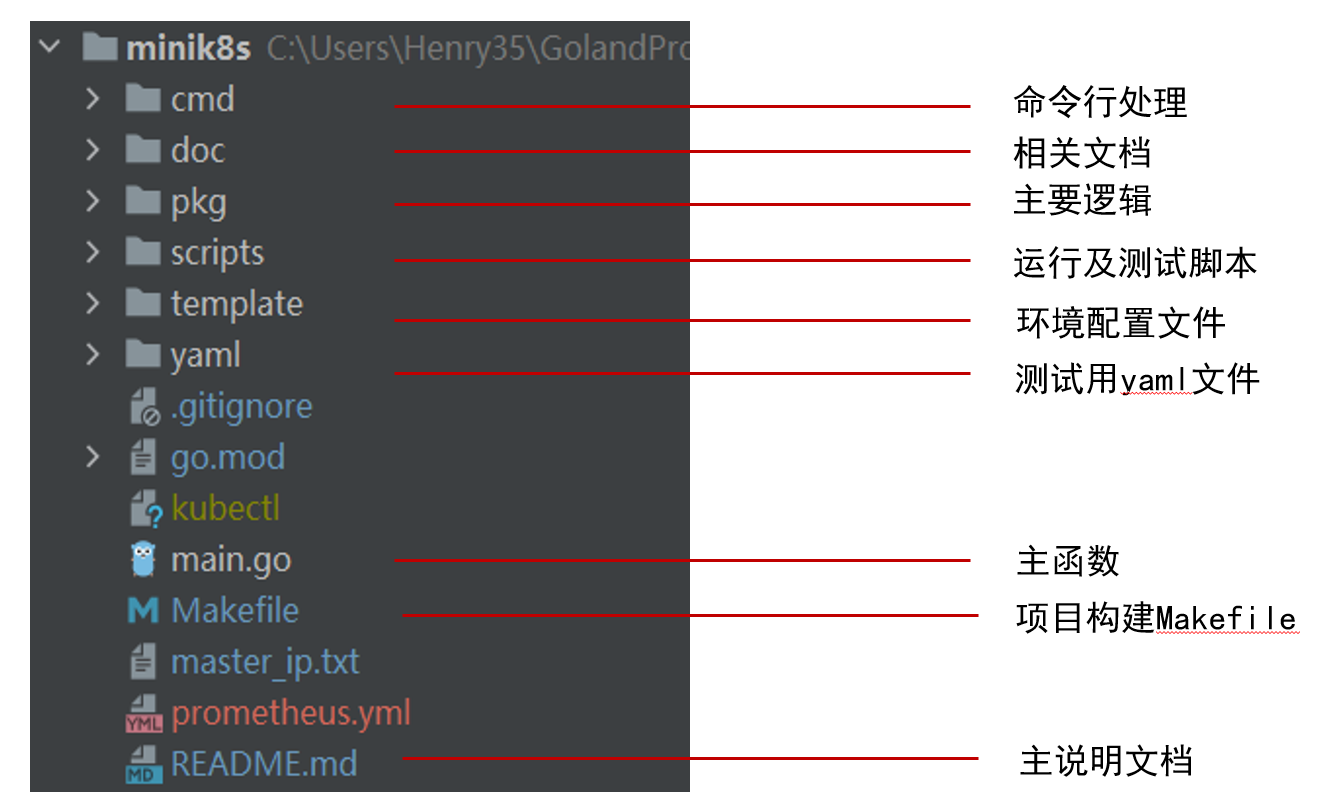


整体架构图如下：

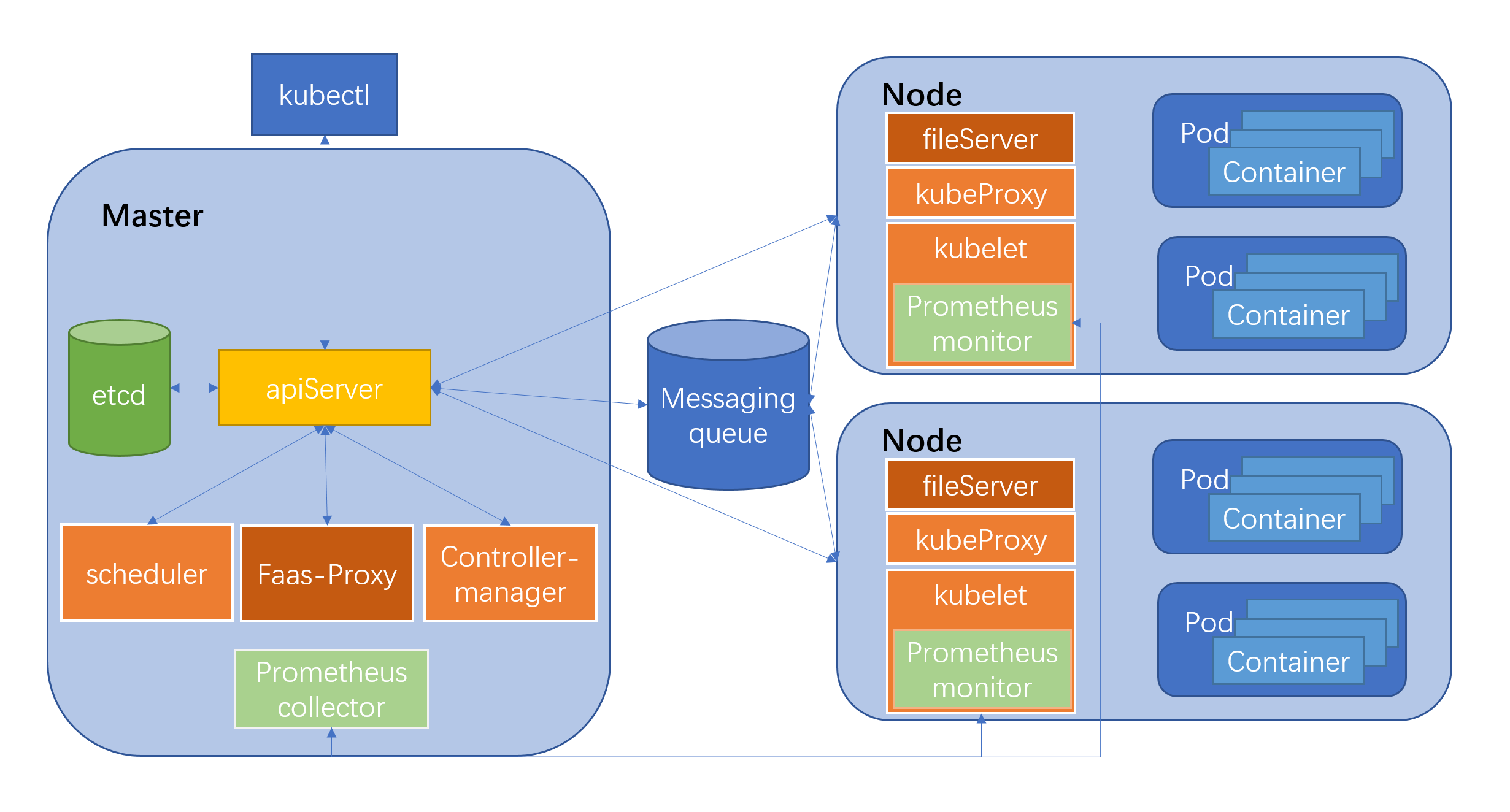


1. **项目整体架构与软件栈**

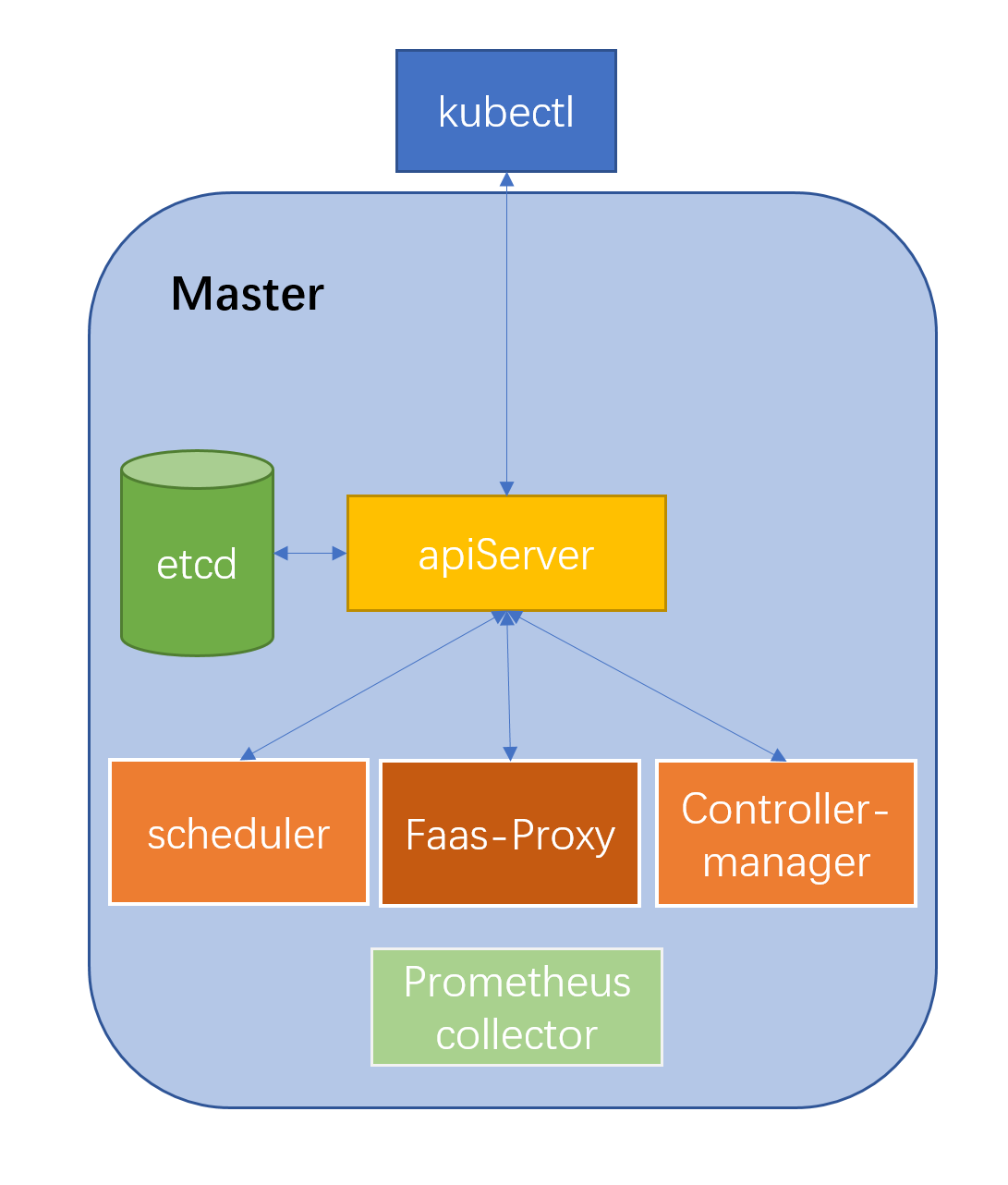
* 项目的文件结构：



* 项目的整体架构：

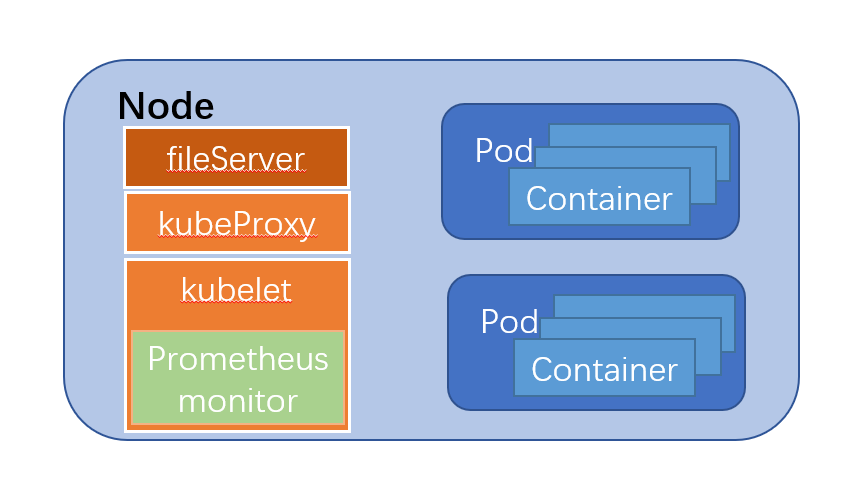


* Master节点上的组件及其作用：



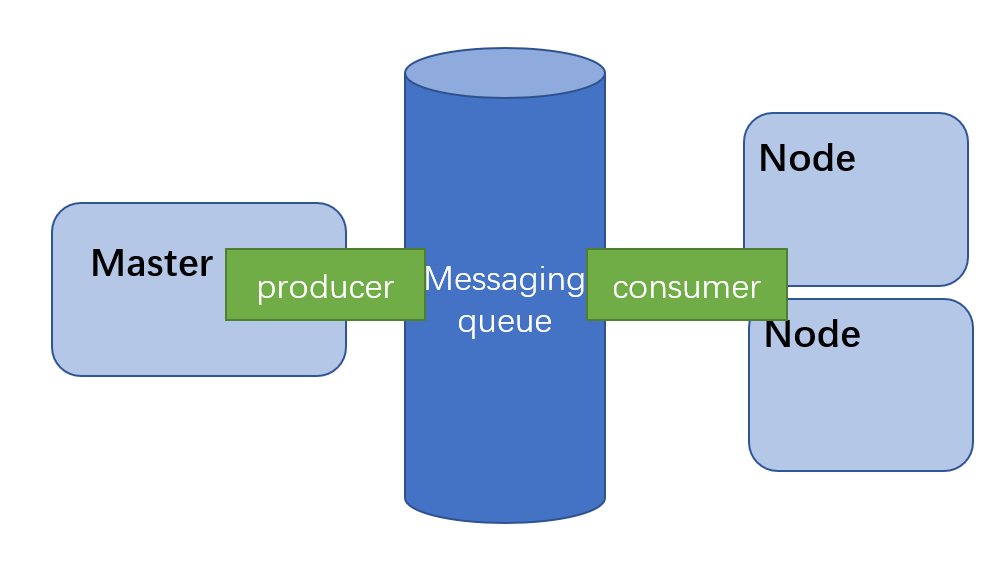
1. etcd：存储集群的所有需要持久化的数据。
2. apiServer：集群操作和CRUD的入口，采用现有echo框架，支持restful服务。
3. scheduler：负责集群资源调度，当前支持的策略有round-robin和根据Node资源以及Pod反亲和性调度。
4. Controller-manager：负责维护集群的实际状态与预期状态一致。
5. Faas-Proxy：负责接受并转发serverless的请求，支持流量统计，缓存。
6. Prometheus collector：通过Prometheus采集Node的监控数据。

* 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。

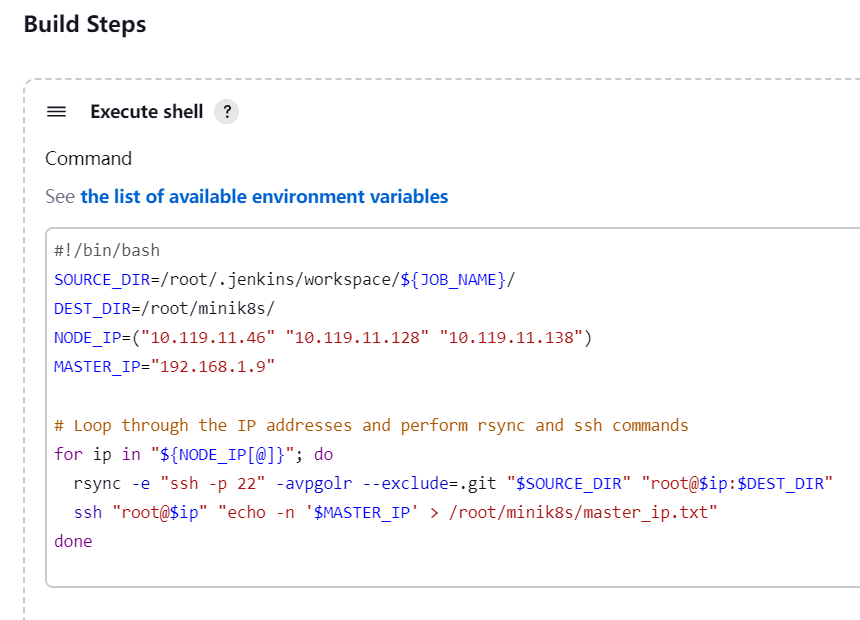
* 项目使用的软件栈与开源组件：

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
7. **项目的所有功能、使用方法和实现方式**
8. **多机部署与调度**

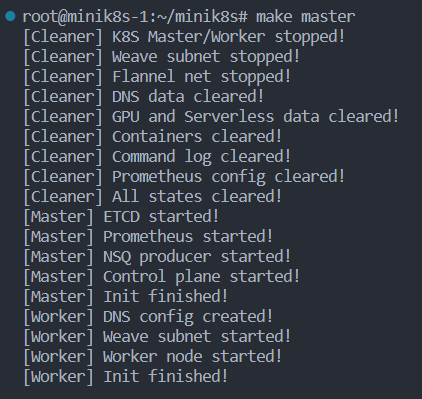
在视频1-2\_NodeAndPod\_V2.mp4中已经介绍了如何通过kubectl启动Master和Worker节点，以及如何查看节点的运行情况。

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：

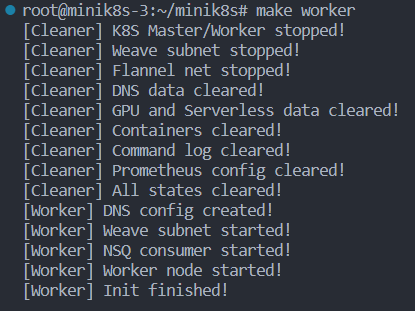
1. 首先需要保证服务器上已经安装了Docker，weave和flannel。
2. 在每个服务器上，修改master\_ip.txt的内容为Master服务器的IP地址。（事实上在CI/CD的部署脚本中已经完成了这一步）



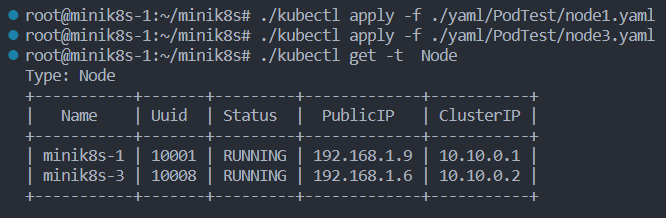
1. 在Master服务器上运行make master，等待初始化完毕。



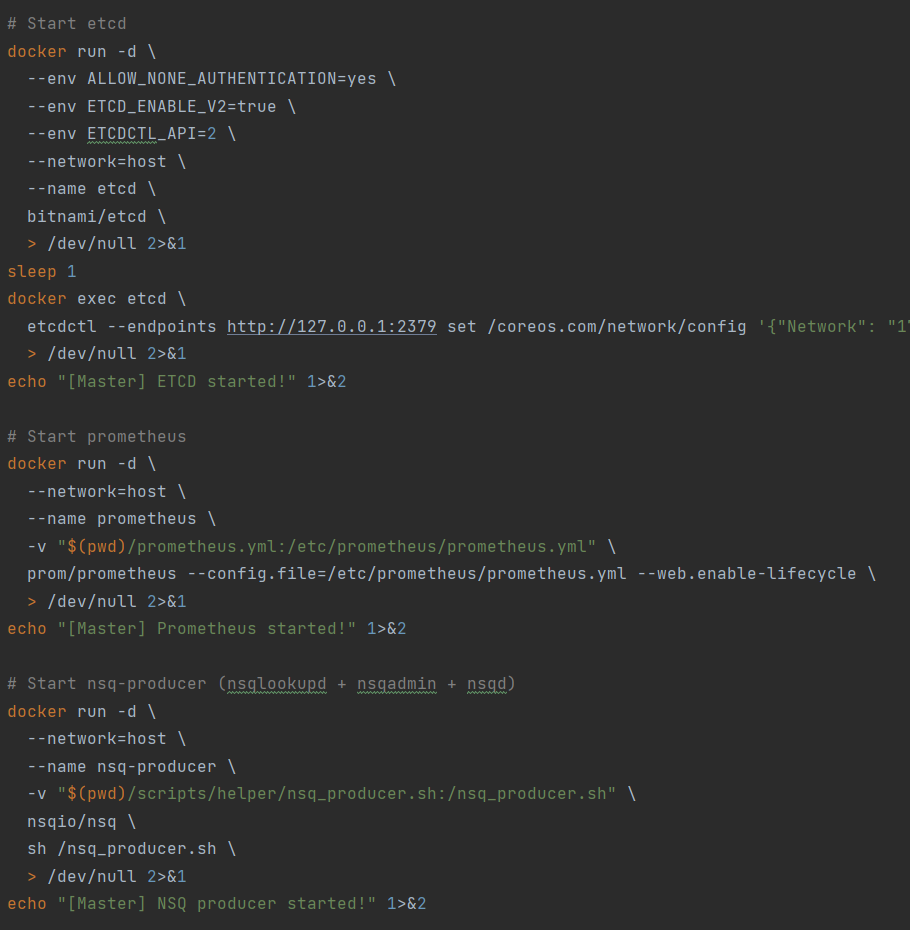
1. 在Worker服务器上运行make worker，等待初始化完毕。



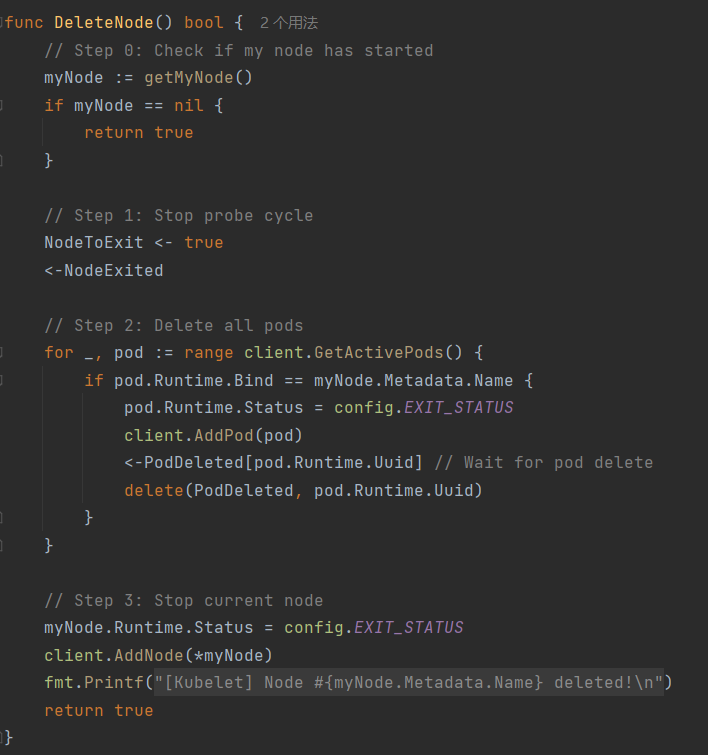
1. 接下来就可以通过kubectl来启动和查看Node了。

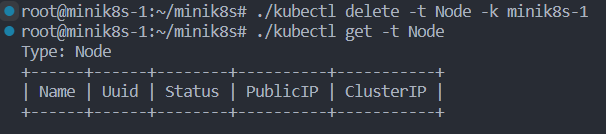


在make master和make worker的过程中，我们启动了一些基础设施，例如etcd, nsq, Prometheus等。这些基础设施都实现了容器化，如此用户只需要安装Docker和CNI插件即可运行Minik8s，十分方便。



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。





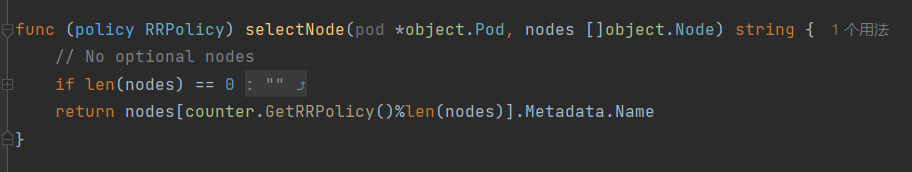
然后是多机调度的实现。首先Scheduler要筛选出可选的Node，在本项目中，我们考虑了nodeSelector和CPU/内存资源的限制：



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：

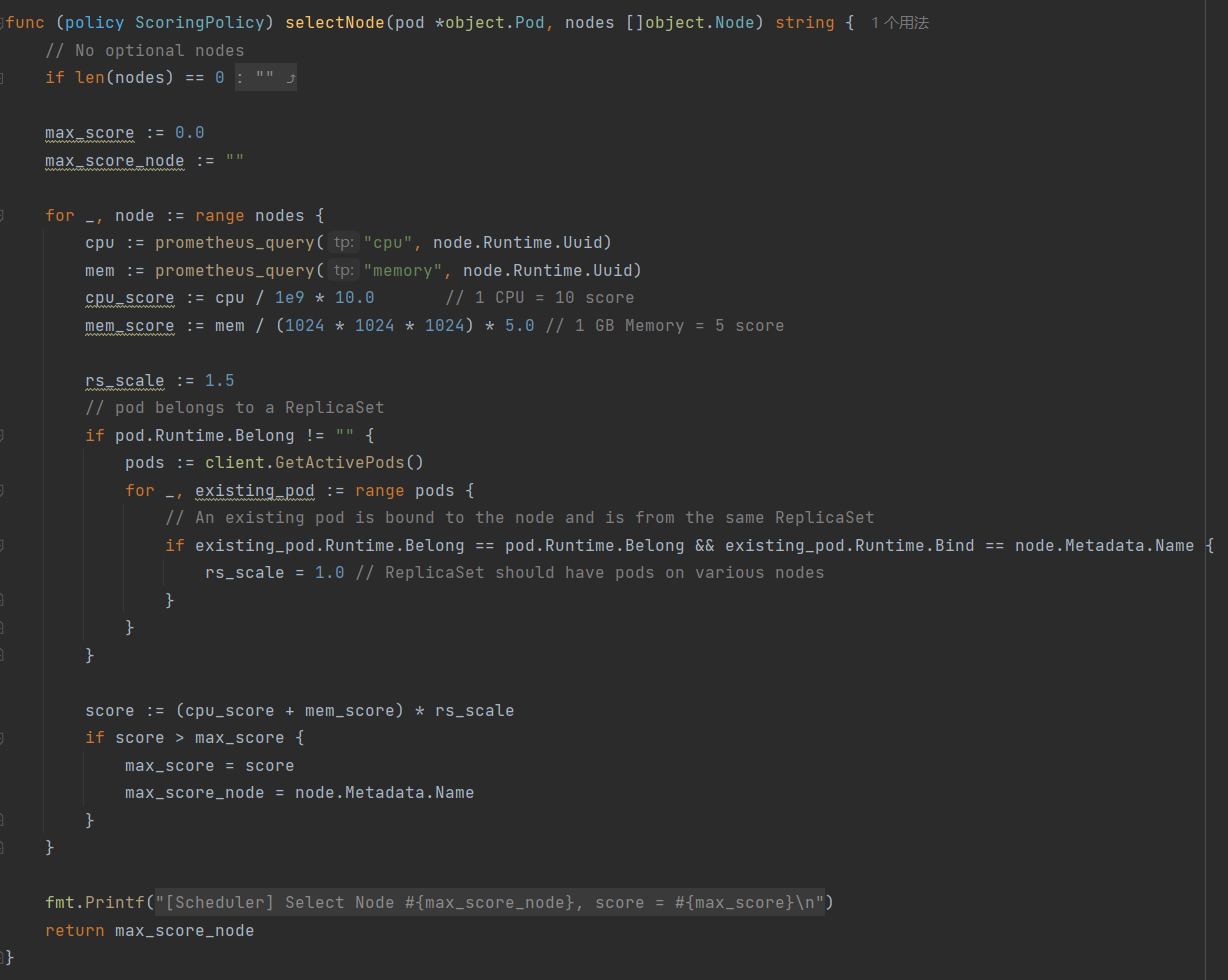
1. Round Robin策略

按顺序依次选择每个可选的Node。



1. 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**。

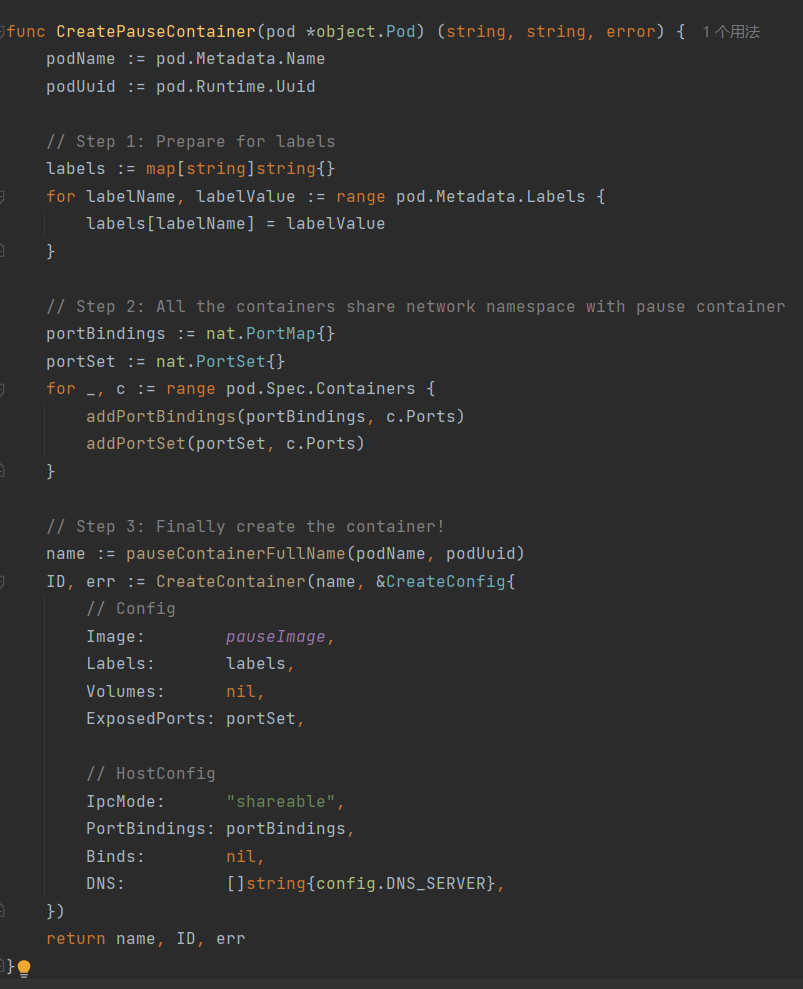


1. **Pod抽象**

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。

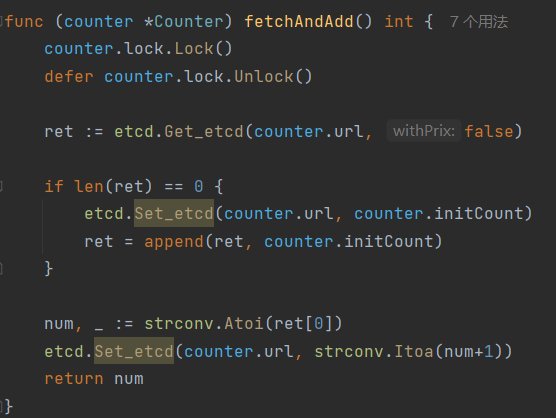


通常情况下，PAUSE容器的端口绑定就是其他容器的端口绑定的并集。但是在ReplicaSet的Pod Spec中不能指定HostPort（否则可能会端口冲突），所以如果没有指定HostPort，就使用随机的空闲端口作为HostPort。

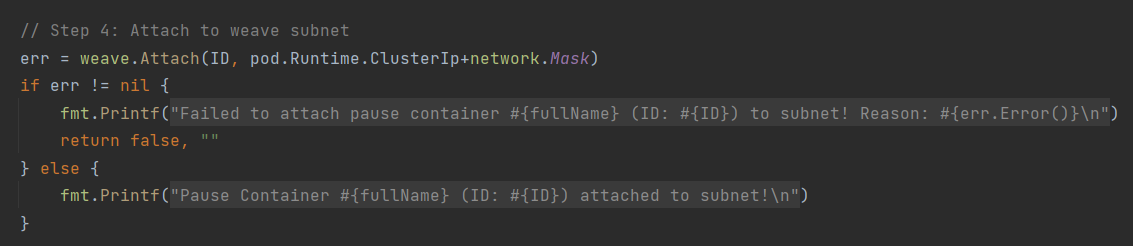


* Pod的Cluster IP是如何分配的？

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

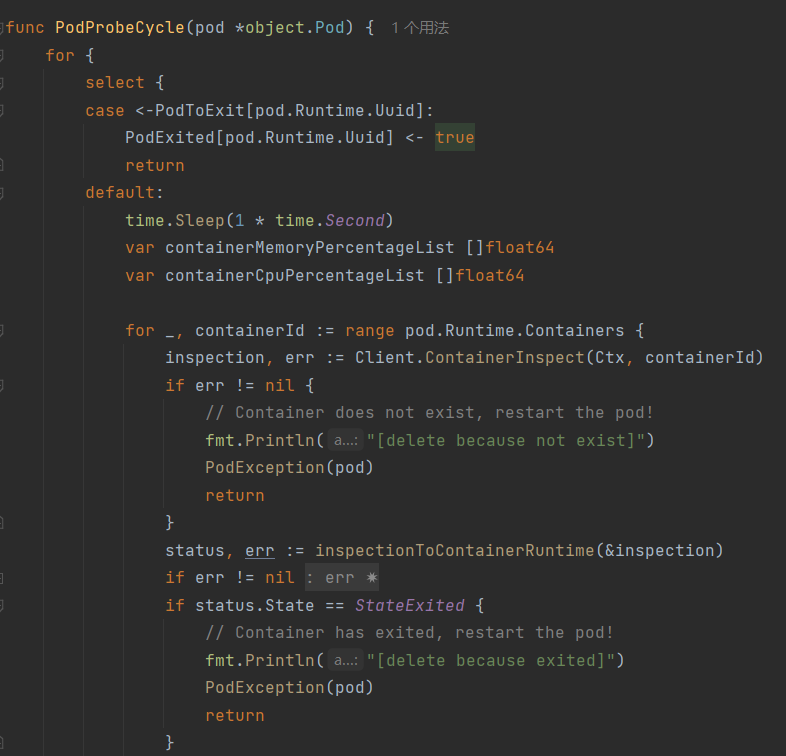


在分配完Cluster IP后，还需要执行weave attach <PAUSE容器ID> <Cluster IP/Mask>来将这个Pod加入到weave子网。然后在集群中就可以用<Cluster IP : containerPort>来访问Pod内部的容器了。

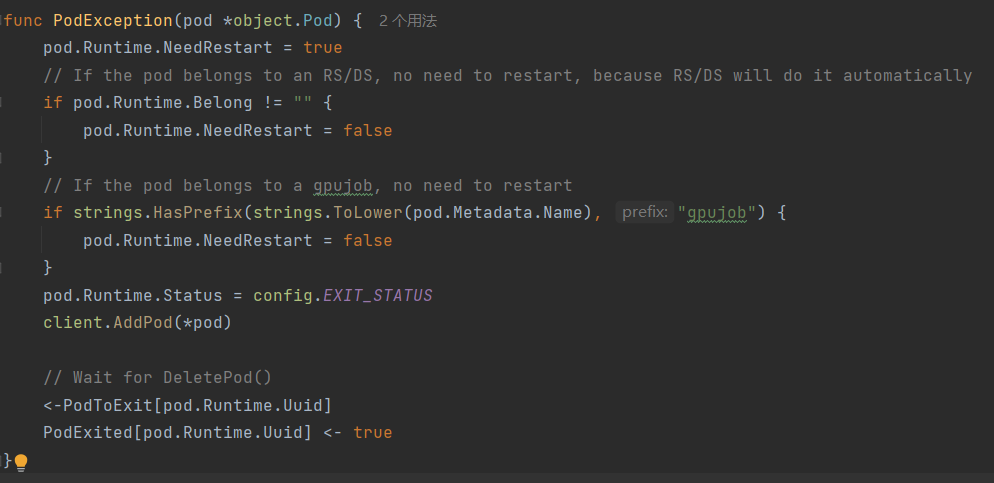


* Pod内的容器异常退出后是如何重启的？

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。

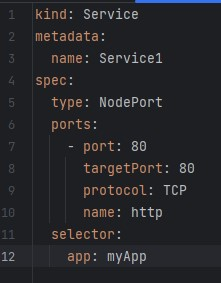


PodException函数会将pod.Runtime.NeedRestart设为true，由此让Kubelet重启这个Pod。但如果这个Pod属于一个ReplicaSet或者DaemonSet就不需要重启了（只删除不重启），因为ReplicaSet或者DaemonSet的controller会自行管理它们的Pod。



1. **Service抽象**

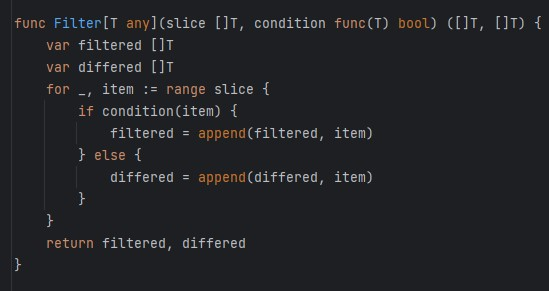
验收指南中与Service相关的功能要求已经全部在视频3-4\_RSAndService\_V3\_part1.mp4与视频3-4\_RSAndService\_V3\_part2.mp4中介绍过了，这里补充一下Service的yaml文件以及Service的具体实现。



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。

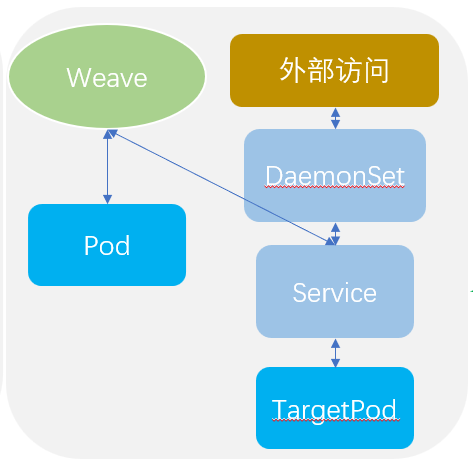


文本

描述已自动生成

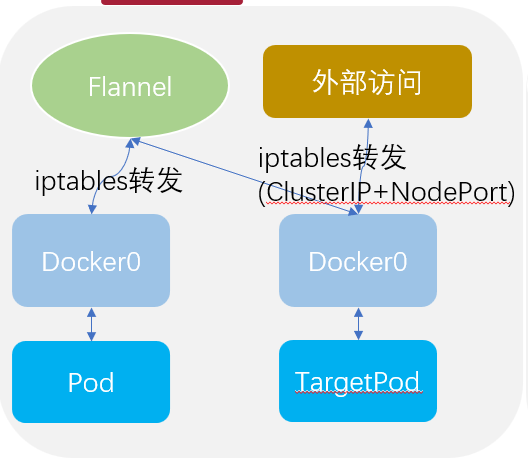
1. Nginx模式：

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。



1. 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。



1. Microservice模式：

此部分放在10. Microservice中讲解。

图示

描述已自动生成

1. 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。

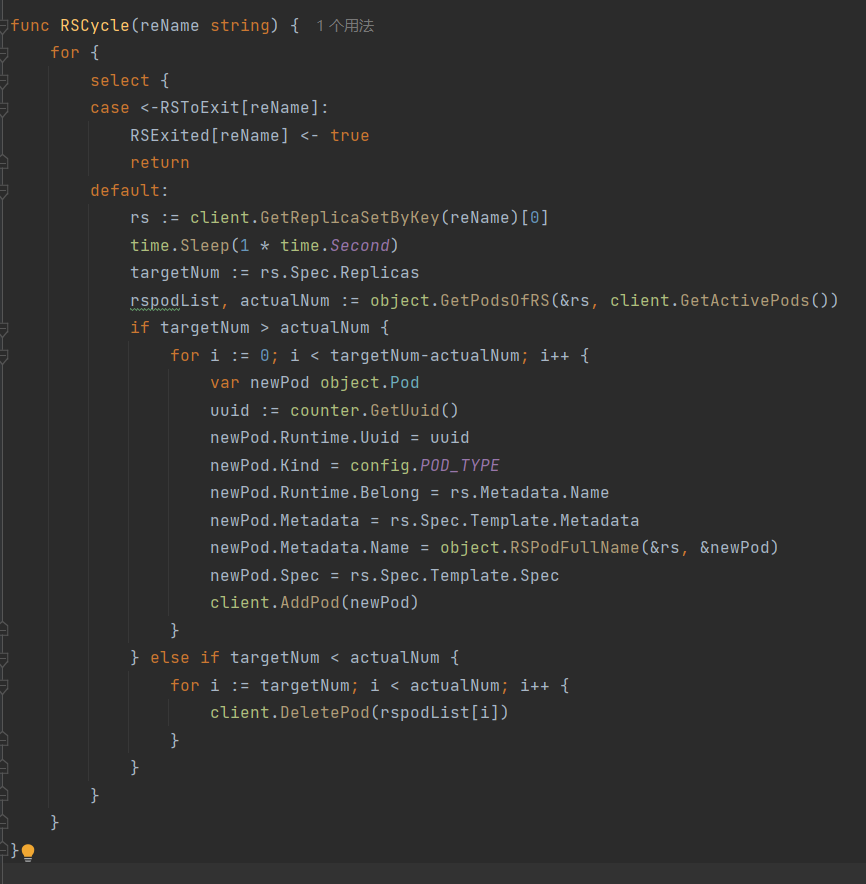
1. 其他：

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。

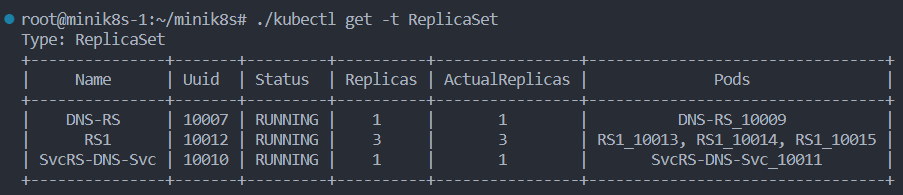
1. **ReplicaSet抽象**

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。

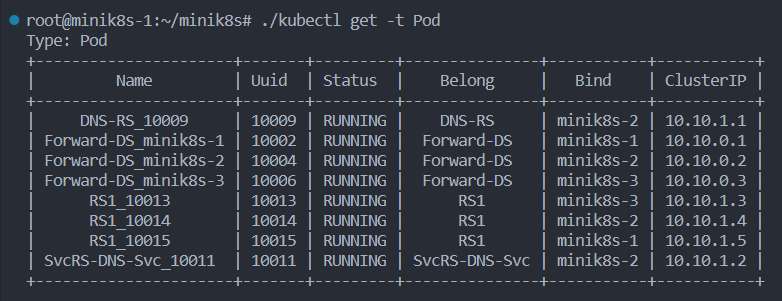
ReplicaSet的核心功能就是保证有指定数量的Pod在运行，因此ReplicaSet Controller需要定期检查集群中的Pod数量，如果Pod数量不足就启动新的Pod，如果Pod数量过多就删除部分Pod。



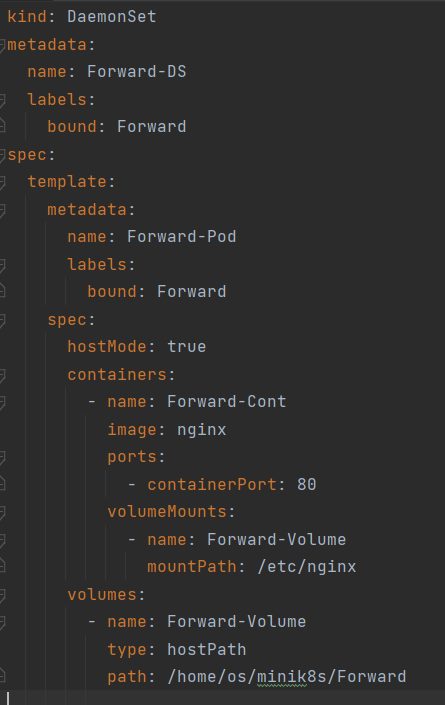
通过kubectl get可以看到ReplicaSet的预期replica数和实际replica数。

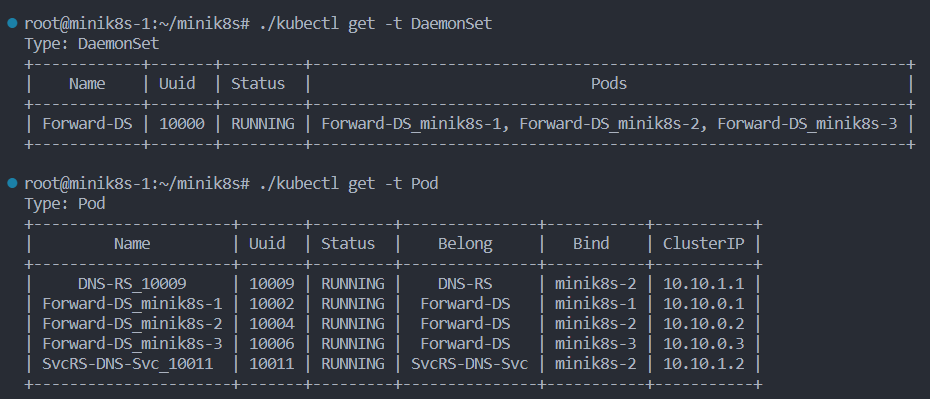


由于Scheduler的综合评分策略给没有replica的Node额外奖励50%的分数，所以ReplicaSet的Pod一般可以均匀分布到不同Node上。

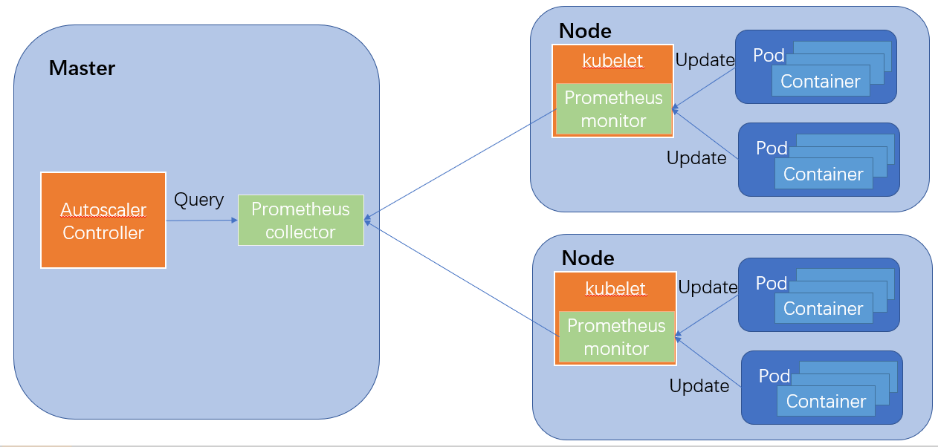


然后除了ReplicaSet，我们还额外实现了DaemonSet抽象。通过DaemonSet，我们可以在集群中的每个Node上运行一个Pod，这样可以支持基于Nginx策略的NodePort Service。（这里的hostMode: true表明这个Pod应该与主机共享Network Namespace）





1. **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。

1. **DNS与转发**

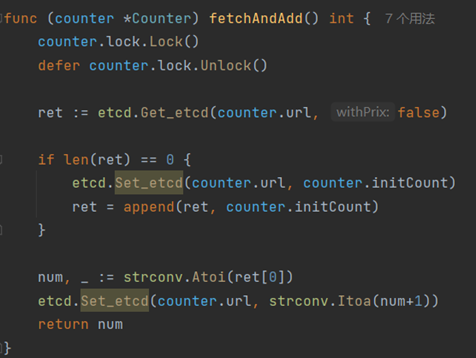
验收指南中与DNS与转发相关的验收要求已经全部在视频6-7\_DnsAndGateway\_V2.mp4中介绍过了，这里补充一下DNS与转发的具体实现。

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。

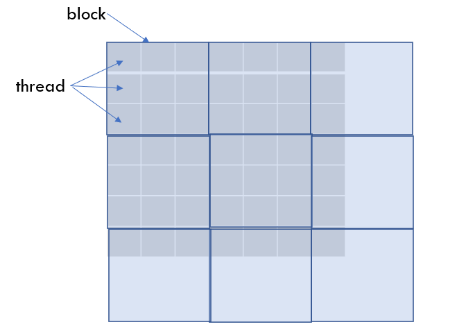
1. **控制面容错**

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。

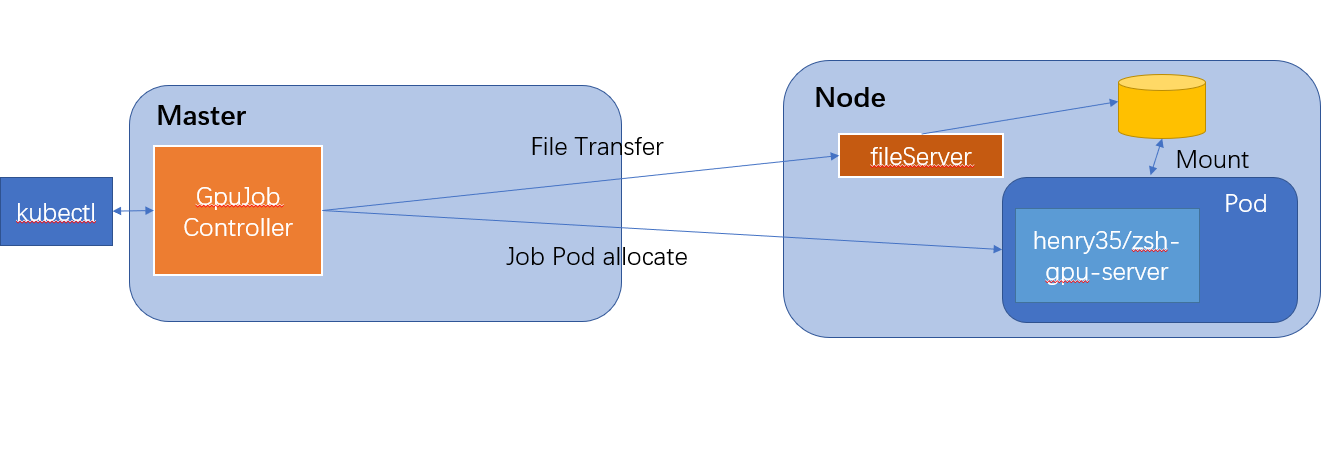
****

1. **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。

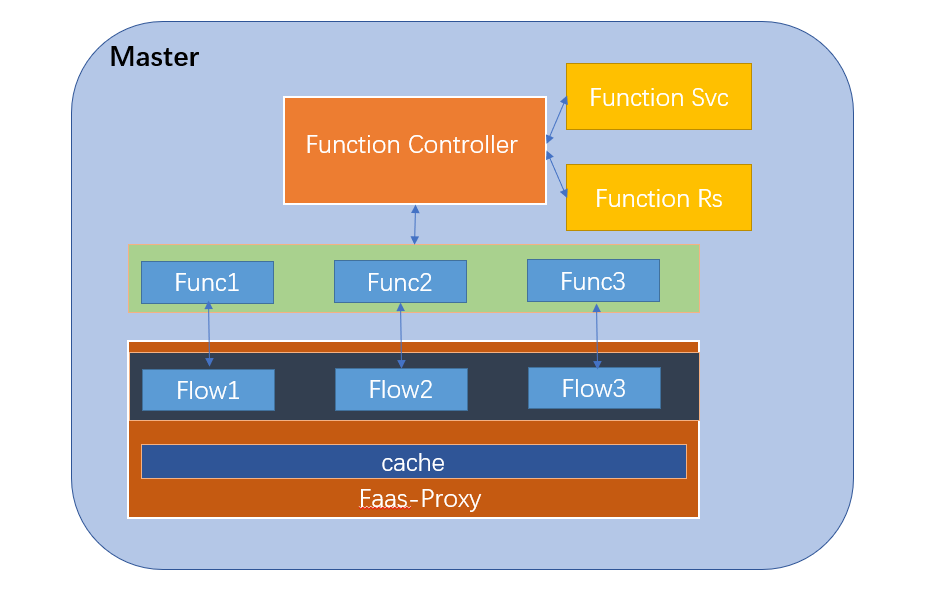


**GPU的具体实现：**

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。

1. **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。

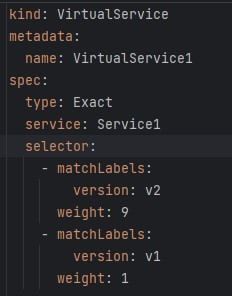
Function Proxy为每个已注册的函数运行一个协程，来动态管理函数的流量。当对函数的请求到来时，它会统一被Function Proxy拦截并进行代理，增加流量，并根据流量进行Function副本的扩容；同理，当长期没有请求到来时，也会自动将副本数降至0。此外，它自身也能通过缓存机制更快速的返回目标函数的结果。Function Proxy支持单请求也支持work-flow。当work-flow到达时，它会自动根据DAG执行循环和路径选择，最后将结果返回用户。

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。

1. **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。



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。

