Minik8s验收文档

项目总体架构

软件栈

Containerd：项目使用的容器运行时，用来实现容器的管理，创建，删除等

CNI插件和Flannel插件：用于实现Pod网络通信

Cobra：用于实现Kubectl命令行

Gin：用 Go (Golang) 编写的高性能 HTTP web 框架，用于实现HTTP通信

IPVS：用于实现Service，通过ipvsadm工具配置虚拟IP到对应的endpoints(真正提供服务的Pod)的转发规则

CoreDNS：用于实现DNS功能中的第一步域名解析，与etcd搭配使用

Nginx：用于DNS功能中第二步根据不同子路径转发到不同的服务

Prometheus：用于实现日志与监控，监控集群内的各个指标

Grafana：为集群监控提供更丰富的可视化方式

Docker Registry：使用docker registry容器实现serverless中的镜像管理

buildkit：使用buildkit进行镜像构建

各组件简介

Kubectl：Minik8s的命令行工具

Apiserver：Minik8s的中枢，负责管理集群的apiobject，开放http接口，代理所有对apiobject的操作

Scheduler：负责Pod的调度

ControllerManager：管理一系列控制器组件，向控制器提供统一的ListWatch管理

Serverless Gateway：代理对Serverless的访问，管理Serverless对象的生命周期

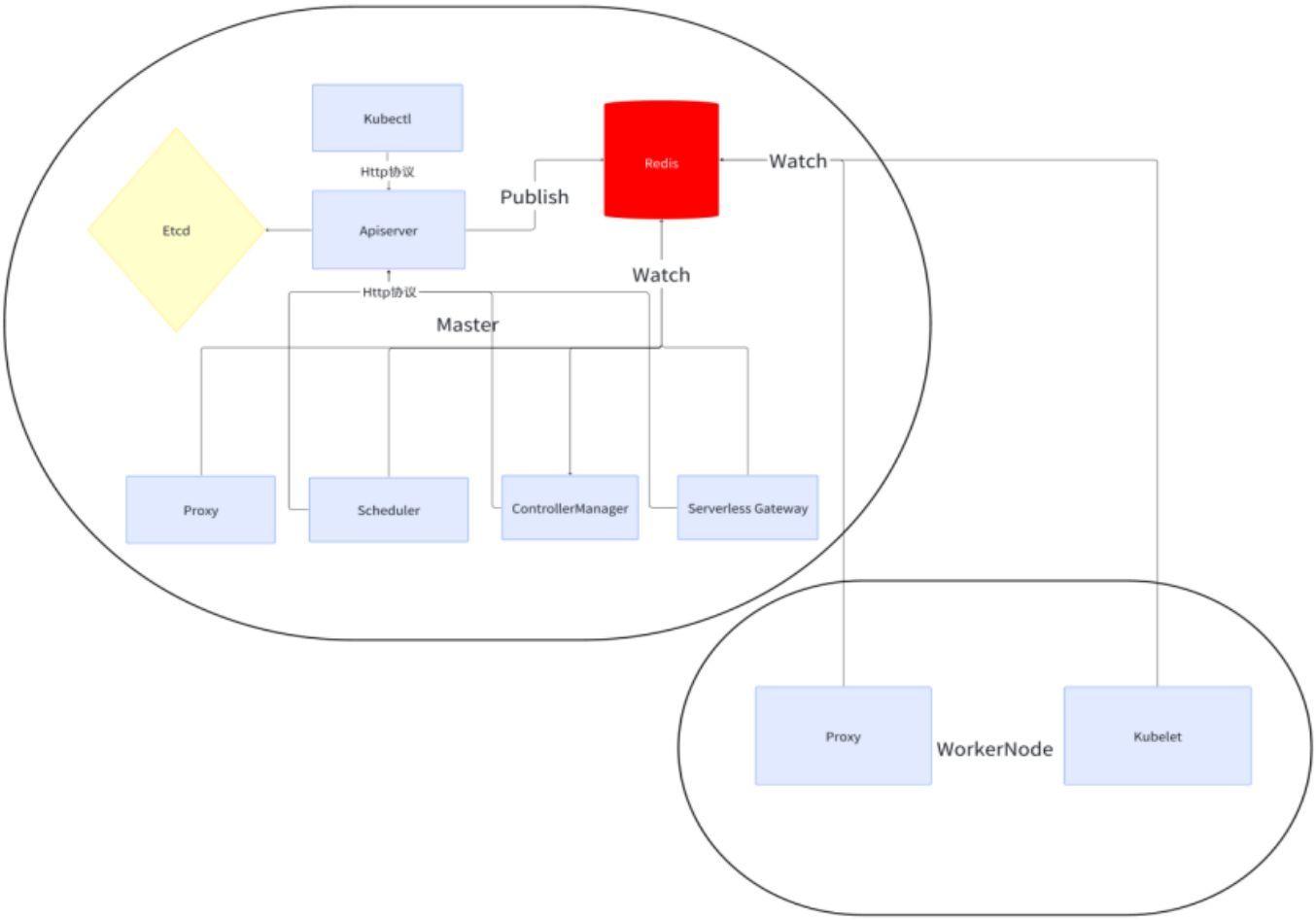
Proxy：运行在各个节点上，监听Service和对应Endpoints的创建与删除，执行ipvs指令

Kubelet：负责管理节点信息与Pod

Etcd：存储整个项目的持久化数据，仅能由Apiserver修改

Redis：作为消息中间件，负责异步消息的通知

系统架构



各组件通过http请求apiserver操作etcd中的apiobject，apiserver根据apiobject的变动发布消息到redis，由redis通知订阅了对应消息的组件。整体架构如图，详细介绍可见功能与实现具体介绍一节。

项目成员分工

林隽乐

- 功能：Pod管理、Node管理、GPU
- 组件：kubelet，ctlmgr框架，node controller
- 环境：CI/CD环境配置，部署脚本、编译脚本编写，Serverless 镜像基础文件，Serverless Demo
- 贡献度：1/3

石远康

- 功能：Service，DNS，Prometheus
- 组件：kubeproxy，service controller，dns组件
- 环境：containerd安装配置、集群网络环境搭建、cni/flannel配置、dns服务搭建、私有镜像服务搭建
- 贡献度：1/3

诸柏铭

- ReplicaSet，HPA，持久化存储
- 组件：apiserver框架，kubectl框架，sl_gtw，scheduler，replicaset controller，pvc controller，hpa controller
- 贡献度：1/3

项目信息

项目gitee路径：https://gitee.com/unit-1112/minik8s.git

分支介绍

主要分支：

- main：用于发布产品，develop分支上的代码经过测试评审，被认为达到产品质量之后，会被合并到main分支上。
- develop：用于开发产品，其他分支上开发的feature基本具备完整的功能后，提交PR，经过审核合并到develop分支上。

Feature分支：

- pod：试水用的分支，初步完成pod的api对象，确定项目文件结构
- kubelet：开发kubelet组件，以及与pod管理相关的功能
- controllerManager：开发控制器管理器ctlmgr
- service：开发service相关功能，包括service的api对象和kubeproxy组件、service controller
- mount：为pod引入mount能力
- pvc/pv：两个分支共同完成pv和pvc的api对象，以及通过pv/pvc为容器绑定nfs的能力
- container_stats：为pod引入CPU和内存占用数据监控能力
- replicaset：实现replicaset功能，实现replicaset controller组件、replicaset的api对象
- hpa：实现hpa功能，实现hpa controller组件、hpa的api对象
- nodeWatchdog：实现node上下线监控，实现node controller组件
- prometheus：实现通过prometheus监控集群状态的功能
- deploy：实现CI/CD设计，编写部署脚本
- serverless：实现serverless相关的功能，包括组件、Demo
- gpu：实现GPU任务提交功能
- docs：编写文档

新功能开发流程

我们的新功能开发流程是从develop创建对应的功能分支，由对应的组员完成开发后，进行测试，提交PR，经过小组其他组员审核后合入develop分支。

CI/CD

我们的CICD是借助Gitlab和Gitlab Runner完成的。

我们在集群子网内的一个节点上部署了我们自己的私有Gitlab。每次将commit推送至Gitlab后，Gitlab就会将CI/CD任务发布给Runner执行。整个CI/CD流程是这样的：

- 构建Build：执行make，构建我们在Makefile中定义好的产物。包括编译kubelet, apiserver, scheduler, ctlmgr等go程序，复制执行脚本、实例yamls文件到构建产物文件夹。
- 部署Deploy：执行部署脚本 `scripts/deploy_to_master.sh`和 `scripts/deploy_to_worker.sh`，根据配置，通过SSH，将发送构建产物发送到集群。

CI/CD的流程定义在.gitlab-ci.yml文件中。

测试方法

我们的测试主要分为单元测试和系统测试。

单元测试方面，出于时间考虑，我们没有为每个功能点设计单元测试，而是针对组件的关键功能设计测试。例如apiserver的http请求处理，redis信息监听发布，pod的创建、删除、获取状态。针对这些关键功能，我们利用go的test系统设计可以通过go test执行的测试代码，通过手动执行的方式测试验证对应的功能正确。

系统测试方面，我们设计了部署脚本和CI/CD流程，以及大量的系统功能测试用例，具体体现为apiobjects/example下的yaml文件。每次推送commit触发CI/CD任务并成功执行后，项目就会被部署到集群，随后可以由我们利用提前设计好的用例对整个系统的功能进行测试。

功能与实现具体介绍

Kubectl功能

我们项目Kubectl的实现基于go开源的项目cobra，通过导入cobra库完成了对一系列指令的参数指定，本项目在root指令下共有8个指令，分别是apply，get，describe,sched,workflow,delete,func，event。具体指令的功能与案例我们在下面的表格中进行展示。

指令类型	使用例子	指令功能
apply	./kubectl apply -f pod.yaml	用于引入对应apiobject对象的yaml文件并解析
get	./kubectl get pod (-n default)	获取在指定namespace（添加-n并加上namespace，缺省时默认为default）下所有apiobject的信息
describe	./kubectl get pod pod1(-n default)	详细描述一个pod对象的具体信息
sched	./kubectl sched MininumCpuStrategy	用于指定scheduler的调度策略(默认为random，还支持最小CPU利用率和内存使用量)
func	./kubectl func create -f ./function-example.yaml ./kubectl func delete -f function-example	对应serverless的function对象，负责function实例的创建和销毁
delete	./kubectl delete pod xxx (-n default)	通过对象名和命名空间用于删除具体apiobject类型的某一个对象
workflow	./kubectl workflow create -f ./example.json ./kubectl delete workflow example	对应serverless的workflow对象，负责其创建和销毁
event	./kubectl event create -f ./event.json ./kubectl event delete event	负责事件的绑定和解绑，如定时触发某个函数实例等等

用户可以通过对应的指令在master节点上引入，销毁，查看对象，调整策略等等。具体使用时可以通过对应指令加上 --help来给出该指令的用途，以及使用例子。

DNS

主要功能

Minik8s中的DNS主要功能：

支持用户通过yaml配置文件对Service的域名和路径进行配置，使得集群内的用户可以直接通过域名与路径的80端口而不是虚拟IP来访问映射至其下的其他Service的特定端口。同时，集群内的Pod可以通过域名与路径的80端口访问到该Service。同时支持同一个域名下的多个子路径path对应多个Service。

DNS配置文件的详细格式

```
kind: Dns
apiVersion: v1
name: dns-test1
namespace: default
host: minik8s.com
paths:
  - service: dns-service
    pathName: path1
    port: 22222
  - service: dns-service2
    pathName: path2
    port: 34567
```

具体实现思路

DNS的配置是通过yaml文件进行的，在具体操作的时候将其映射为一个 `apiobject` (`DNSRecord`)，支持的字段及其含义和要求文档中的基本一致

当相应的yaml文件被解析后，会生成一个 `DNSRecord`对象，该对象的host和nginx server ip的对应会被存储到etcd中，从而实现了域名到ip的映射，并通过 `coreDNS`实现了域名解析的动态加载

而nginx server ip下不同path到具体的service的映射则是通过nginx的配置文件实现的，具体的配置文件示例如下

```
worker_processes  5;  ## Default: 1
error_log  ./error.log debug;
pid        ./nginx.pid;
worker_rlimit_nofile 8192;

events {
  worker_connections  4096;  ## Default: 1024
}
http {

    server {
        listen 192.168.1.12:80;
        server_name minik8s.com;
```

```
        location /path1/ {
            access_log /var/log/nginx/access.log;
            proxy_pass http://10.10.0.1:22222/;
        }

        location /path2/ {
            access_log /var/log/nginx/access.log;
            proxy_pass http://10.10.0.2:34567/;
        }

    }

}
```

当DNSRecord被更新后，nginx的配置文件也会同步热更新，从而实现了不同path到不同service的映射

根据域名定位到IP和port的过程

1. 通过域名找到nginx server ip

以上面的nginx配置文件为例，我们要访问 `http://minik8s.com:80/path1`，首先会通过 `coreDNS`将 `minik8s.com`解析为 `nginx` server ip `192.168.1.12`，这个过程是通过 `coreDNS`自动读取etcd中存储的host和固定的nginx server ip并解析来实现的

2. 根据不同的path找到service

在nginx中根据location的path名字找到对应的service ip和端口，比如在上面的例子中是 `10.10.0.1:22222`，这个过程是通过nginx实现的

DNS注册的过程

通过yaml文件注册，kubectl接收到注册指令以后，发送http请求给apiserver, apiserver将配置文件转换成dnsrecord数据结构存储到etcd中，同时存储对应的host和固定的nginx server ip到etcd中，以供coreDNS来解析域名，最后读取etcd中所有已经创建的dnsrecord,对nginx配置进行热更新

```
#DNS创建
./kubectl apply -f dnsrecord.yaml

#查看已经创建的DNS
./kubectl get dns

#删除DNS
./kubectl delete dns [对应DNS的NAME]
```

coreDNS

CoreDNS只需在master节点上运行，是在控制平面上运行的。

> coreDNS对于DNS记录的存储

CoreDNS内部包含一个DNS记录存储后端。该后端被称为CoreDNS的存储插件。CoreDNS存储插件的主要作用是管理DNS记录的存储和检索，例如将DNS记录存储在etcd、Consul或文件系统中，并在必要时从这些后端中检索记录。当客户端查询DNS记录时，CoreDNS将首先从存储插件中检索记录，然后将记录返回给客户端。如果记录不存在，则返回一个相应的错误。此外，存储插件还支持动态DNS更新，允许客户端通过API向CoreDNS添加、删除和修改DNS记录。

不同子路径对应不同的service

使用**nginx**做反向代理，将不同的path路由到不同的ip+port

Prometheus

Prometheus功能

Minik8s使用Promethus实现简单的日志和监控功能，以提高系统的可观测性。具体的功能如下：

1. 实现对集群中各节点的资源的监控。Minik8s能够采集集群中每个节点(主机)的运行指标，如CPU、内存、网络等信息，并通过Promethus UI显示各节点的资源。
2. 实现对用户程序的日志监控。用户可以通过Promethus Client Library在自己的程序中自定义需要采集的指标，主动暴露相应的信息。创建一个Pod运行该程序，集群中的Promethus能够监控到这些用户自定义的指标，并显示在Promethus UI。
3. 集群中Promethus的自动服务发现。集群中的Promethus自动发现Node的加入和退出、Pod的创建和删除，在不重启Prometheus服务的情况下动态地发现需要监控的Target实例信息。
    - 当新的节点加入集群时，集群中的Promethus可以自动监控新加入的节点。当节点退出集群时，自动停止对该节点的监控。
    - 当用户创建一个Pod，且其中的程序主动暴露自定义的Promethus指标时，集群中的Promethus可以自动监控Pod程序中自定义的指标。当Pod被删除时，自动停止对这些用户自定义指标的监控。
4. 利用Grafana为集群监控提供更丰富的可视化展示方式。

实现思路

1. 首先在master节点上部署Prometheus服务器
2. 在每个node上部署node_exporter
3. 在master节点的配置文件中加入node的IP和固定端口，具体配置如下

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The
default is every 1 minute.
  # scrape_timeout is set to the global default (10s).
```

```
# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
          # - alertmanager:9093

# Load rules once and periodically evaluate them according to the
global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any
timeseries scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "workers" #定义名称

    static_configs:
      - targets: ["192.168.1.14:9100"] //两个初始workers节点,9100为node固
定的端口
      - targets: ["192.168.1.15:9100"]
```

4. 对于要监听的Pod来说，我们要求用户在创建相应的Pod时要在labels中加入“log: prometheus”，同时在容器的端口中指定某个开放的Port的name为prometheus，这样prometheus_controller就会自动将这个Pod加入配置文件中

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The
default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
```

```
          # - alertmanager:9093

# Load rules once and periodically evaluate them according to the
global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any
timeseries scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]

  - job_name: "workers" #定义名称

    static_configs:
      - targets: ["192.168.1.14:9100"] //两个初始workers节点,9100为node固
定的端口
      - targets: ["192.168.1.15:9100"]
      - targets: ["10.10.17.100:2112"] //自己创建的需要监听的Pod
```

5. 通过PrometheusController来自动发现和修改需要监听的对象的变化，包括node和pod

   PrometheusController会定期获取集群中所有的node并筛选出需要监听的Pod，然后和正在监听的node数组和Pod数组进行比较，如果发现有变化，就对Prometheus服务进行热更新（修改配置文件，然后执行reload），以此来实现自动监控

Service和Kubeproxy

Service功能

Service支持多个pod的通信，对外提供service的虚拟ip。用户能够通过虚拟ip访问Service，由minik8s将请求具体转发至对应的pod，使用IPVS控制流量的转发。Service可以看作一组pod的前端代理。Service通过selector筛选包含的pod，并将发往service的流量通过随机/round robin等策略负载均衡到这些Pod上。

在符合selector筛选条件的Pod更新时（如Pod加入和Pod被删除），service会动态更新（如将被删除的Pod移出管理和将新启动的Pod纳入管理）。

用户能够通过yaml配置文件来创建service

Service配置文件格式如下：

```
apiVersion: v1
kind: Service
metadata:
  name: HelloService
  namespace: default
spec:
  selector:
    app: hello
  type: ClusterIP
  ports:
    - name: HelloPort
      protocol: TCP
      port: 12345 # 对外暴露的端口
      targetPort: hello # 转发的端口的名字，pod对应的端口名字
```

NodePort Service

service在不同节点上开相同的端口，用户以同一端口号访问任意一个node都可以访问到该服务。

NodePort类型Service的配置文件

```
apiVersion: v1
kind: Service
metadata:
  name: HelloService2
  namespace: default
spec:
  selector:
    app: hello
  type: NodePort
  ports:
    - name: HelloPort
      protocol: TCP
      port: 23456 # 对外暴露的端口
      targetPort: hello # 转发的端口的名字，pod对应的端口名字
```

实现思路

Kubeproxy

Kubeproxy运行在各个节点上， 主要是监听master节点上的svccontroller，当监听到service或对应的endpoint变化时，修改本地的ipvs规则。同时Kubeproxy还会定期检查节点上已经创建的service规则和etcd中存储的规则，保持一致性

Master节点上的ServiceController

ServiceController会监听apiserver，当监听到service创建时，会为Service分配一个“持久化的”集群内的IP（Service的IP是持久化的，就是Service对应的Pod挂了也不会变），然后筛选符合条件的Pod，创建对应的

endpoints，将Service信息和对应的endpoints信息发给apiserver，apiserver将这些信息都存储到etcd中，最后通知各个节点上的kubeproxy创建相应的IPVS规则，最终实现访问Service的虚拟IP转发到相应Pod上的功能

ServiceController同时也会监听Pod状态的变化（创建，删除等），并检查变化的Pod符合哪些已经创建的Service，然后修改相应的信息（包括etcd中存储的service对应的endpoints信息，各个节点上的ipvs规则等）

Service注册过程

通过yaml文件注册，kubectl接收到注册指令以后，发送http请求给apiserver, apiserver将配置文件转换成service数据结构，通过Redis发送给ServiceController，ServiceController监听到Redis中有关Service创建的信息，读取Service信息，为Service分配“持久化的”集群内唯一的ClusterIP，然后筛选符合条件的Pod，创建对应的endpoint数据结构，最后通过http将完整的Service信息和endpoint信息发送给apiserver，apiserver将这些信息都存储到etcd中，然后通知各个节点上的kubeproxy创建相应的IPVS规则。

```
#service创建
./kubectl apply -f service.yaml

#查看已经创建的service
./kubectl get service

#删除service
./kubectl delete service [对应service的NAME]
```

ListWatch的实现

除了通过http请求访问Apiserver调用对应的handler函数之外，我们还使用了redis数据库作为消息中间件来进行信息的异步转发。ListWatch的Watch函数能够监听某个topic主题上对应的message，一旦有新的message便会交给Watch函数之中绑定的handler函数进行对应处理，Publish函数则负责将新的message公布到对应topic上以便为所有Watch的对象监听并调用对应handler。

Apiserver的实现

Apiserver的实现主要是使用了当下主流的用 Go 语言编写的 Web 框架gin，在本项目中我们在master节点的8080端口上运行Apiserver实例，它可以响应controller，kubelet，serverless模块转发而来的http请求，并根据http请求的对应方法（GET,POST）以及路由转移到对应的handler进行处理。比较重要的一点是我们的Apiserver负责绝大部分与etcd的通讯，在对应handler函数中负责更新，删除，添加，查看etcd中对应的api对象的运行状态，spec规范，卷绑定等各种信息。不仅如此，Apiserver在这些handler中还负责为所有global类中定义的topic来publish新的message信息，方便对应watch这些topic的对象进行后续处理。

Scheduler的实现

本项目实现的scheduler一共支持三种调度策略，第一种是随机选择一个健康的节点并将pod部署到其上，另外两种分别是选择当前cpu占用率较少的节点和选择当前内存占用率较少的节点。策略的选择可以通过kubectl对应的sched指令来进行对应的设置。Scheduler的整体思路便是通过Watch监听Apiserver处Publish的关于pod的信息，收到对应信息后会将信息解析，根据信息中pod的行为（只有创建和更新时才会执行），来为对应的pod实现node节点的调度。调度方法也很简单，首先会从Apiserver处获得所有的node信息，根据Pod中的NodeSelector（没有就忽略）挑选处合适的所有node，然后再根据Scheduler中的策略（就是上面的三种策略）进一步选出唯一一个合适的node，并创建一个node和pod绑定的对象（称为NodePodBinding，后面简称

binding）发送给Apiserver并存储在Etcd中，后面的Kubelet会根据binding的信息是否匹配（如binding的node是不是当前节点）来实现pod的最终创建。除此之外，scheduler还会定时检查binding的情况，如果某个节点挂掉了，所有在etcd中与这个node相关的binding都会被删除，scheduler可以监听到这一事件的发生，这个时候scheduler就会重新调度这个pod到另一个节点上运行。

ReplicasetController的实现

我们项目中的ReplicasetController会Watch对应Apiserver上Publish的与Replicaset相关的topic，一旦有新的信息Msg便会获取并交给对应的WatchHandler来处理。Msg中包含replicaset的行为有四种，分别是创建，更新，删除，扩容。对于每一个具体的replicaset我们会有一个对应的worker协程，worker节点总体做如下工作，它会定时获取与这个replicaset相关的已经处于running状态的pod数量，并且与预期的replicas数量进行比对，大就delete一个pod，小就create一个pod，直到两者相等处于平衡（pod数量是一个个增加或减少的）。同时我们也会在replicaset中保存与它相关的pod的平均CPU占用率和内存使用量，用于后面我们的hpa controller来进行相应计算来为它进行对应扩容。回到ReplicasetController，如果Msg中replicaset的行为是创建，那么我们会为它创建一个新的worker协程，如果是更新，那么它会找到对应的worker协程，更新它的target（也就是预期的replicaset），如果是删除，那么便会删除对应的worker协程。最后如果是扩容，会修改target中replicaset的replicas数量，并且马上进行一次同步操作（这里的同步操作指的就是worker协程中让running的pod数量与预期replicas数量同步的操作）。ReplicasetController会定期从Apiserver获取所有的pod信息，方便在worker协程中筛选出与该协程对应replicaset相关的pod。

HpaController的实现

与ReplicasetController类似，HpaController同样会Watch对应Apiserver上与HPA相关的topic，一旦有新的信息Msg便会获取并交给对应的WatchHandler来处理。Msg中包含HPA的行为有三种，更新，创建，删除。同样我们对于每一个HPA会有一个对应的worker协程。如果Msg中HPA的行为是创建，那么我们会为它创建一个新的worker协程，如果是更新，那么它会找到对应的worker协程，并且更新它的target（也就是新的HPA），如果是删除，我们也会删除对应的worker协程。每一个协程会定时处理一个同步函数，定时锁定的时间取决于我们在HPA对应yaml文件中设定的scaleInterval属性，也就是扩缩容的速度。我们在同步函数中会获取该HPA绑定的replicaset，因为我们的replicaset中存储了与它相关的pod的平均CPU占用率和内存使用量。我们会将它与我们在HPA的yaml文件中的metrics字段下设定的预期CPU占用率或是内存使用量进行一个比较（具体的比较方法是将replicaset中对应的资源量除以metrics中规定的目标资源量，然后乘上当前replicaset中replicas的量，计算出的结果取整再与min replicas和max replicas取中间数作为扩缩容后replicaset中replicas的量。），将更新replicas的replicaset通过http请求返回Apiserver再由Apiserver来Publish对应replicaset的scale信息，再由ReplicasetController中Watch函数对应的WatchHandler来进一步处理scale请求，最后达到一个replicaset的扩缩容的效果。

PV和PVC（持久化存储）的实现

我们的PV和PVC的实现是通过PVController来控制的。在本项目中我们选择了其中一个节点作为专门使用的nfs服务器。因为PVC的创建有两种，分为静态和动态。我们先来说明静态的实现方法，静态需要用户手动apply一个pv，接着再apply一个pvc来实现与对应的pv进行绑定。一个手动apply的pv需要在对应的yaml文件中写下绑定到nfs服务器的对应路径，然后当我们再apply完这个pv后再apply一个pvc与它做对应绑定时，PVController会通过http请求从Apiserver中获取当前所有的pv，然后根据pvc中要求的capacity和storageclass两个字段匹配到对应的满足条件的pv（pvc要求的capacity需要小于pv的capacity且两者storageclass是一样的），之后两者的状态都会由available变为bound，表示互相绑定，如果pvc没能找到满足的pv，它就会动态创建（这个稍后再讲）。如果一个pod想要通过volume来mount对应的pvc，它会先查找绑定的pvc（通过名字），然后根据pvc再解析到它bound的pv，然后根据pv中的绑定到的nfs服务器的路径将该路径绑定到本地节点的/mnt/.k8s-volume中一个自己创建的文件夹下（该文件夹的创建会根据pod的uuid，name以及namespace这几个参数共

同创建，在kubelet创建pod时完成）。然后再将该文件夹路径绑定到pod中container要求的volumeMount下，以此来完成一个PVC的绑定。PVC的动态创建相比于静态创建，因为开始时pv是不存在的，是由pvc去生成对应的pv，所以会通过pvc（uuid和name）生成一个对应的pv，并且创建对应的pv的目录在nfs服务器的/home/nfs下。之后pod通过volume来mount对应的pvc的流程与上面静态pvc的方式是一样的，最后能够完成pod与对应动态PVC的绑定。

Serverless

可以通过kubectl管理function、workflow和event（见kubectl一节）。

可以通过function的yaml文件定义创建镜像时需要额外执行的指令，函数文件目录的路径。在构建镜像时，minik8s会将额外指令贴在dockerfile的最后，并将函数文件目录下的所有文件拷贝进镜像中的指定路径`/function`，用户只需要满足提交的目录下包含一个`func.py`文件，且文件中有一个名为main函数，接收一个参数的函数即可。

集群利用建好的镜像创建pod，并将pod放在replica set的管理下，通过service开放pod的服务，然后由serverless gateway对集群外开放，代理对function和workflow的访问，同时根据函数的访问热度控制replica set的replica数量。

用户可以通过workflow的json文件定义workflow，在其中定义分支节点和函数节点，执行到函数节点时会调用对应的function，执行分支节点的时候会根据条件进入对应的下一个节点，分支的条件支持的变量有布尔值、整型、浮点数、字符串。

用户可以通过event的json文件定义事件，目前支持定时事件，可以按照cron的格式定义定时事件触发的频率和时间、触发的workflow，在定时事件触发后，minik8s会调用对应workflow进行执行。

Kubelet与Node监控

kubelet主要做两件事：管理pod，上报node状态。

kubelet会监听binding的创建和删除：当binding创建时，检查新binding是否是将pod绑定到自己的节点，如果是，则在本节点上创建pod。删除也是同理。

kubelet会定期发送health report到apiserver，apiserver会将health report信息发布到redis。node controller会监听health report信息，对于每一个运行中的node，node controller都会维护一个超时计数，收到health report会清空对应的计数器，如果计数超过一个给定值，node controller会认为节点已经下线，从而通知apiserver对节点相关信息进行清理，删除对应的binding，以便集群中原来绑定到下线节点的pod可以重新被调度到正常工作的节点上。

kubelet的health report信息还包括节点上运行的pod的信息，apiserver会利用这些信息更新pod的状态，并发布pod状态更新的消息。诸如hpa controller、service controller的组件会监听这个信息。对hpa来说，其会根据pod的资源占用信息，对replica set的副本数量进行控制；对service controller来说，其会根据pod的ip变动，修改对应service的endpoint信息。