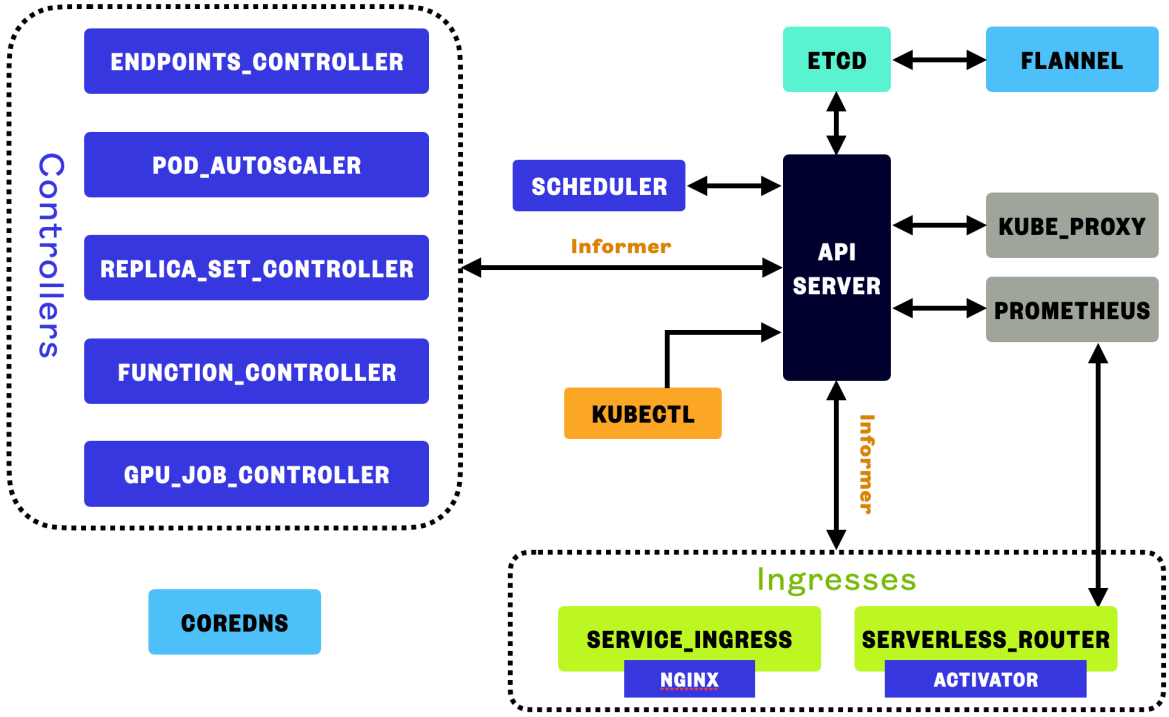
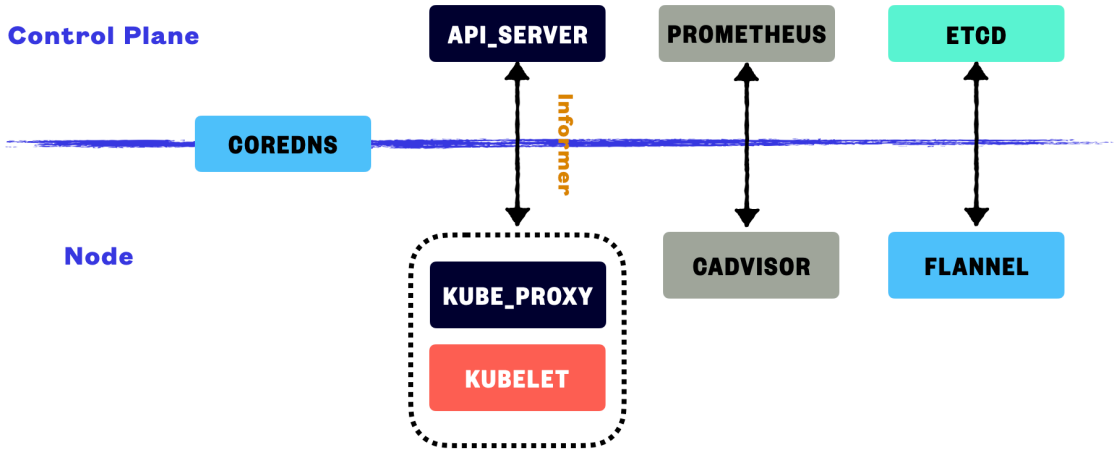
Minik8s验收文档

总体架构图示

本项目架构与K8s大致相同，控制面有以下组件：



worker节点有以下组件：



组件

API Server

API Server的主要职责是与etcd进行交互，为API对象提供REST操作服务，并为集群的共享状态提供前端，其他所有组件都通过该前端进行交互。

REST操作主要包括对API对象的CRUD操作，对API对象的watch，以及提供临时文件的存储。除此之外，API Server还充当请求到node的proxy，例如用户使用kubectl log命令请求获取特定pod或容器的log，则该请求会首先发送到API Server，再由API Server转发到对应的node上，并将结果返回。

API Server使用Axum作为服务器框架搭建。

Kubelet

Kubelet的职责：

- 管理某一个Node上的Pod，包括Pod的创建、删除、调谐和状态更新
- Node的自动注册和状态更新
- 对控制面提供操作Pod的API，包括获取Pod日志和在Pod中执行命令

Kubelet通过Informer接收Pod信息的变化，并由Pod Worker执行相应操作：

- 当新增Pod时，Pod Worker创建Pod并启动。创建Pod时首先会创建Pod的数据目录用于挂载共享卷，之后创建并启动Sandbox（即Pause Container）、并发创建Pod Spec中定义的所有容器，最后更新Pod状态。创建Pod容器时首先会根据定义的Image Pull Policy拉取镜像，之后创建共享卷的目录，再根据Pod Spec生成Docker容器配置，包括镜像、资源限制、挂载的共享卷、启动命令、重启策略等，并使其与Sandbox共享PID、IPC和网络，最后创建容器。更新Pod状态时，通过Docker Inspect命令获取各容器的运行状态、转换为ContainerStatus对象，并根据各容器状态计算Pod Phase。Pod启动直接调用Docker API并发启动各个Pod容器。
- 发现Pod状态更新时，Pod Worker对Pod进行调谐，调谐时根据各容器的运行状态计算出所要执行的调谐操作：如Pod Sandbox被停止，则重新启动；如Sandbox被删除，则清除所有Pod容器并重新创建Pod；对于Pod容器则根据重启策略决定是否重新启动和创建。
- Pod从etcd中被删除后，Pod Worker停止Pod并将Pod移除。

Status Manager负责定期检查Pod的状态，如果状态发生变化则向API Server提交更新。

Node Status Manager负责在Kubelet启动时向API Server注册节点，之后定期检查并更新节点状态。

Scheduler

Scheduler负责将新创建的Pod调度到节点上。

Scheduler默认会将Pod调度到Pod数量最少且状态正常（根据最近一次心跳时间判断）的Node上，从而实现类似Round Robin的调度效果。除此之外，用户可为Node打上标签，比如（cpu=high-performance，gpu=nvdia），在Pod的nodeSelector中使用这些标签以调度到想要的Node上。

Kube Proxy

Kube Proxy通过设置iptables，实现了各节点通过Cluster IP来访问Service。

具体来说，Kube proxy通过Informer与API Server建立连接，根据Service Endpoints的变化来调整iptables的规则。Kube Proxy会在iptables中创建三种Chain：

- KUBE-SERVICES Chain中，每个Service都有一条规则，将目标地址为Cluster IP的流量全部导入对应的KUBE-LB Chain中
- KUBE-LB Chain中，以等概率的方式将流量导入不同的KUBE-EP Chain中
- KUBE-EP Chain中的规则将流量导入相应的Pod IP中。

Kube Proxy使用rust-iptables操作iptables

ReplicaSet Controller

ReplicaSet Controller负责监控和更新ReplicaSet的状态，以及将其调谐到期望状态。

ReplicaSet Controller使用Informer监听ReplicaSet和Pod的状态，当ReplicaSet状态发生变化时对其进行调谐，当Pod被创建、更新和删除时更新其所属ReplicaSet的状态。调谐ReplicaSet时，根据当前Replica数量和期望Replica数量创建或删除Pod，删除Pod时将优先选择处于失败和未就绪状态的Pod以及早先创建的Pod。

Pod Autoscaler

每个节点上都运行着cAdvisor，用于收集Docker容器的资源指标，在Kubelet向API Server注册节点时cAdvisor会被添加到Prometheus的监控对象中。由于在创建Pod容器时添加了对应的标签，因此在Prometheus中可通过标签筛选Pod。

Pod Autoscaler使用Informer监听HPA对象，将新增或更新的HPA对象加入工作队列，定期遍历工作队列对HPA进行调谐，使用Replica Calculator计算期望的Replica数量，并对Replica数量进行限制，使其符合定义的扩缩容速率并减小波动，最后修改目标ReplicaSet的期望Replica数量，具体的扩缩容操作由ReplicaSet Controller根据ReplicaSet Spec完成。

Replica Calculator使用Metrics Client向API Server查询目标ReplicaSet下所有Pod的资源指标，计算资源实际占用总量和Pod Spec中定义的Request总量，根据实际值与目标值（Average Utilization或Average Value）的比值得到扩缩容比率，例如实际值大于目标值时扩缩容比率大于1，即进行扩容操作。在计算的过程中如遇到还未收集到指标的Pod，则对其进行保守估计。

由于瞬时资源指标可能发生突变，为了降低扩缩容过程中Replica数量的波动，Pod Autoscaler记录了稳定窗口（Stabilization Window）中的Replica数量，并将当前的期望Replica数量限制在稳定窗口的最大值和最小值之间。

为了使扩缩容速度符合Spec中的定义，Pod Autoscaler还记录了历史的扩缩容事件，当进行扩散容操作时，首先根据扩缩容事件计算当前Period内已扩容或缩容的Replica数量，推算得到当前Period开始时的Replica数量，进而计算出当前时刻扩缩容的上下限，并对Replica数量进行规制。

GPU

GPU Server

首先需要介绍GPU Server。

GPU Server是单独的程序，功能是从环境变量得到各种配置信息，根据配置信息将相应的文件发送到超算平台，在超算平台上提交任务，并轮询任务的完成情况，最后将任务结果取回。简单来说，GPU Server就是一个单独的负责与超算平台进行交互的程序。GPU Server同样采用rust编写。

GPU Server被提前打包成镜像，作为一个base image。用户上传到minik8s的cuda程序将会被在base image的基础上继续打包为新的镜像。最终的效果是新镜像容器启动时自动启动里面的GPU Server，启动的GPU Server会自动将里面的用户代码文件上传到超算平台，并与超算平台进行交互。最终超算平台上的任务结束并且从超算平台取到结果以后，容器会自己以succeeded状态退出。

GpuJob Controller

GpuJob Controller的作用是监听job的创建与更新，为相应gpu代码打包成镜像，然后为job创建相应的pod，并保证pod按照job预期执行。

GpuJob Controller通过informer监听API Server端job的创建。

监听到job创建以后，controller为打包镜像准备相应的文件。首先根据job的filename字段从API Server发请求取得对应代码文件，然后根据job的配置字段生成对应的.slurm文件，最后生成打包镜像需要的Dockerfile。

生成的Dockerfile可能是这样的（隐去了密码）：

```
FROM minik8s.xyz/gpu_server:latest
COPY . /code/
ENV USERNAME=stu607 PASSWORD=****** JOB_NAME=cuda
ENV CODE_FILE_NAME=gpuveuHG.zip COMPILE_SCRIPT=make
```

这些环境变量将会作为参数，被GPU Server读取。

随后GpuJob Controller使用这些文件生成镜像并push到registry。镜像生成以后，controller为job的template字段填入使用该镜像的pod template，然后向API Server更新job状态。

每次监听到job状态更新，controller都会计算该job的执行状态，并比较是否与job的预期一致，然后进行同步。

GpuJob Controller支持让job预期数量的pod创建并完成，并且还支持指定数量的pod并行存在。逻辑如下：

```
let want_active = min(
    job.spec.completions - status.succeeded,
    job.spec.parallelism,
);
match status.active.cmp(&want_active) {
    Ordering::Less => {
        // Create a new pod, if more pods are needed, they'll be created
later
        self.create_pod(&job).await?;
    },
    Ordering::Greater => {
        // Delete active pods
        let pod_name = self.get_pod_to_delete(&job).await;
        self.delete_pod(pod_name).await?;
```

```
    },
    Ordering::Equal => {
        // Nothing to do
    },
}
```

job.spec.parallelism是job允许的pod并行执行数量。controller通过比较job.spec.completions – status.succeeded与job.spec.parallelism得出want_active，即当前需要多少个active的pod。然后再将其与当前属于该job的active pod数量作比较，最后根据结果创建或者删除pod，保证job执行状态与预期一致。

Function Controller

Function Controller与GpuJob Controller类似，监听function的创建，并为其build image。除了build image，Function Controller还会为function创建replicaset和HPA。最后Function Controller更新状态到API Server。

和上面使用base image不同，Function Controller将提前准备好的function wrapper与用户上传的函数放在一起打包为镜像。

wrapper是这样的：

```
from flask import Flask, request
from handler import handler
app = Flask(__name__)
@app.route("/")
def function():
    print("received request")
    args = {}
    if request.data:
        args = request.get_json()
    print(args)

    return handler(args)
app.run(host='0.0.0.0', port=80)
```

Dockerfile：

```
FROM python:3
WORKDIR /app
RUN pip config set global.index-url https://mirror.sjtu.edu.cn/pypi/web/
simple && pip install flask
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 80
CMD ["python", "server.py"]
```

可以看出，用户上传的代码文件需要包含叫做handler的函数。wrapper为函数提供了底层的访问接口。

Endpoints Controller

Endpoints Controller计算并更新Service的Endpoints。

具体来说，Endpoints Controller通过Informer与API Server建立连接，监听Service与Pod对象的变化，如果Pod的labels与Service的selectors匹配，那么就将该Pod的IP加入Service的Endpoints中。

Service Ingress

Service Ingress是所有带域名Service的流量入口，其使用nginx作为反向代理，将不同host、不同path的http请求导入相应的service中。

具体来说，Service Ingress通过Informer与API Server建立连接，监听Service与Ingress对象的变化，在Nginx中注入以下配置：

- 为每个Ingress的host创建一个nginx的server
- 为每个host中的path在nginx的server中创建一个location对应相应的service

Serverless Router

Serverless Router负责Function的激活和请求转发，并向Prometheus提供Function的扩缩容指标。当收到新请求时，Serverless Router首先从域名中提取出Function或Workflow的名称，处理Function请求时，根据Function对应Service有无Endpoint判断是否处于休眠状态，如果是则将通过将对应ReplicaSet的Replica数设置为1来激活Function，之后等待Function启动后转发请求；如果Function有活跃的示例则直接转发请求，转发请求前还需记录Function请求数量以便向HPA提供扩容指标。Function启动后的扩缩容操作由Pod Autoscaler完成，类似容器资源指标的处理方式，Pod Autoscaler向API Server请求Function指标，根据15秒窗口内每个Pod的实际和目标请求数对ReplicaSet进行扩缩容，Function Scale-to-Zero的时间通过HPA缩容的稳定窗口设定，由于Function HPA的最小Replica数量为0，在经过最少一个稳定窗口时间后，Pod Autoscaler便会将无请求的Function缩容到0。

Workflow的冷启动与Function类似，但为了优化冷启动速度，会在开始时将Workflow中的所有Function并发激活，之后通过模拟执行的方法依次调用Workflow中的Function，将上一状态作为下一Function的参数，并对条件分支进行处理。

Serverless Router基于Hyper（底层Http库）实现。

Kubectl

Kubectl是用户操控Minik8s的命令行工具。

Kubectl使用Clap作为命令行框架，提供了自动补全、命令提示、错误提醒的功能。

除了常用的create、get、describe、patch、delete指令，Kubectl还实现了exec指令远程执行Pod的指令（包括shell），logs指令查看Pod的log，completions指令生成bash、elvish、fish、powershell和zsh的自动补全配置。

常用库

除了在组件介绍中提到的库之外，本项目还主要使用以下库方便开发：

- tokio：Rust异步运行时
- config、dotenv：解析配置文件
- reqwest：HTTP client
- serde、serde_json、serde_yaml：（反）序列化工具

网络

本项目使用Flannel作为网络插件，保证了Pod的IP唯一性和跨节点访问的功能；使用CoreDNS作为全部组件和Pod的DNS服务，将*.minik8s.com的请求导入Service Ingress中，*.func.minik8s.com的请求导入Serverless Router中。

容错

本项目的容错由etcd与Informer的Resync机制来保证：

- etcd持久化数据，使得其它组件崩溃重启时，能读取之前的状态。
- Informer使用WebSocket与API Server建立连接获得etcd有关状态的实时更新，并将数据缓存在本地。当发现本地数据与etcd中的不一致时（比如说组件重启、API Server断连后重连），则会调用Resync函数，使组件的状态再次与etcd中保持一致。
    - Kubelet将所有Pod重新加入Pod Worker的工作队列并进行调谐
    - Kube Proxy重新配置iptables
    - ReplicaSet Controller重新计算并更新所有ReplicaSet的状态，如果ReplicaSet状态发生变化则会重新进行调谐
    - Pod Autoscaler将所有HPA重新加入工作队列并进行调谐
    - Endpoints Controller重新计算Service的Endpoints
    - Service Ingress会重新配置nginx
- 除了整个控制面崩溃重启后能够恢复，我们的Resync机制还能保证以下情况的容错：
    - API Server崩溃重启后，各组件自动重连
    - 单一组件崩溃不会影响其它组件，且该组件重启后会自动恢复其状态

功能示例

演示视频链接（包含验收指南要求的所有功能演示）：https://jbox.sjtu.edu.cn/v/link/view/275cf9ecfd924a3cbd8349fa5c74755e

在答辩时未来得及演示的ReplicaSet在1m53s处，Function动态扩缩容在20m14s处。

各种对象的配置文件示例可参考examples目录。

多机部署

在Master节点中运行master/up.sh启动控制面，该脚本会做以下操作：安装Flannel，启动etcd，启动CoreDNS，将其余控制面组件会运行在docker compose中。

在Worker节点中运行node/up.sh启动Worker节点，该脚本在用户输入Master节点IP后启动Flannel，设置DNS、Docker，启动Kubelet、Kube Proxy。

Node的配置如下：

```
Name: k8s1
Labels:
Last Heartbeat: 2022-06-01 23:38:50.142271029 +08:00
Addresses:
  InternalIP: 192.168.64.50
  Hostname: k8s1
Capacity:
  CPU: 2
  Memory: 8322289KB
Allocatable:
  CPU: 2
  Memory: 7831904KB
System Info:
  Architecture: aarch64
  Machine ID: 61dcbe54db3458381a75bb76f533e29
  Operating System: linux
  OS Image: Linux 20.04 Ubuntu
```

可以使用rkubectl patch命令来更新Node的Labels以供调度。

Kubectl

Kubectl提供了Create, Delete, Describe, Exec, Get, Logs, Patch和Completion八个子命令，可通过help命令获取使用指南：

```
$ rkubectl help logs
rkubectl-logs 0.1.0
Print pod container logs

USAGE:
    rkubectl logs [OPTIONS] <NAME> [CONTAINER_NAME]

ARGS:
    <NAME>              Job or Pod name
    <CONTAINER_NAME>    Container name, optional if there's only one
container

OPTIONS:
    -h, --help           Print help information
    -j
    -t, --tail <TAIL>    Tail option
    -V, --version        Print version information
```

当用户输入的命令不合法时，Kubectl将予以提示：

```
$ rkubectl exec
error: The following required arguments were not provided:
    --command <COMMAND>
    <POD_NAME>
    <CONTAINER_NAME>

USAGE:
    rkubectl exec --command <COMMAND> <POD_NAME> <CONTAINER_NAME>
```

```
For more information try --help
```

Kubectl会校验传入的配置文件格式，如果格式不正确将予以提示：

```
$ rkubectl create -f examples/pods/demo.yaml
Error: Failed to parse file examples/pods/demo.yaml

Caused by:
    missing field `name`
```

此外还可使用Completion子命令生成各类Shell下的Bash Completion自动补全配置。

Pod

Pod实现了验收指南中的所有功能要求，配置和使用示例如下：

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo
  labels:
    app: demo
spec:
  volumes:
  - name: nginx-files
    emptyDir:
  - name: download-files
    emptyDir:

  containers:
  - name: busybox
    image: busybox:latest
    command:
      - ifconfig
    imagePullPolicy: IfNotPresent

  - name: viewer
    image: dplsming/nginx-fileserver:1.0
    ports:
    - containerPort: 80
    volumeMounts:
    - name: nginx-files
      mountPath: /usr/share/nginx/html/files
    - name: download-files
      mountPath: /data

  - name: downloader
    image: dplsming/aria2ng-downloader:1.0
    ports:
    - containerPort: 6800
    - containerPort: 6880
    volumeMounts:
    - name: nginx-files
      mountPath: /usr/share/nginx/html/files
    - name: download-files
      mountPath: /data
    resources:
      limits:
        # 100 milli-CPU
        cpu: 100
        # 128MB
        memory: 134217728
```

创建Pod：

```
rkubectl create -f demo-pod.yaml
```

查看Pod列表：

```
rkubectl get pods
```

查看Pod详情：

```
rkubectl describe pods demo
```

删除Pod：

```
rkubectl delete pods demo
```

除基本要求外，Pod还支持以下配置项：

- 容器镜像的拉取策略（IfNotPresent, Never, Always）
- Pod的重启策略（Never, OnFailure, Always）
- Pod应调度到的Node名称
- Pod应调度到的Node Selector

此外可使用Kubectl查看Pod容器日志，并支持tail功能，例如：

```
rkubectl logs demo busybox -t 10
```

以及在Pod容器中执行命令或开启Shell（按Ctrl-D退出）：

```
rkubectl exec demo nginx -c /bin/bash
```

Service

Service实现了验收指南中的所有功能要求，配置和使用示例如下：

```yaml
# 1. Create a service: rkubectl create -f service.yaml
# 2. Get Cluster IP: rkubectl get services
# 3. Invoke the service by its Cluster IP
kind: Service
metadata:
  name: server-service
spec:
  selector:
    app: server
  ports:
    - port: 80
      targetPort: 80
```

Service的查看和删除命令与Pod类似。

Replica Set

ReplicaSet实现了验收指南中的所有功能要求，配置和使用示例如下：

```yaml
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: server
spec:
  replicas: 3
  selector:
    app: server
```

```yaml
  template:
    metadata:
      name: nginx
      labels:
        app: server
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
```

ReplicaSet的创建、查看和删除命令与Pod类似，此外还可使用Patch命令修改ReplicaSet的目标Replica数量：

```
rkubectl patch -f examples/replicasets/simple.yaml
```

Horizontal Pod Autoscaler

```yaml
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
    type: Resource
    name: CPU
    target:
      averageUtilization: 50
  behavior:
    scaleUp:
      policies:
        - type: Pods
          value: 4
          periodSeconds: 60
      selectPolicy: Max
      stabilizationWindowSeconds: 0
    scaleDown:
      policies:
        - type: Percent
          value: 100
          periodSeconds: 100
      selectPolicy: Max
      stabilizationWindowSeconds: 30
```

HPA实现了验收指南中的所有功能，配置和使用示例如下： 创建HPA前需要先创建目标ReplicaSet和Service，HPA的扩缩容速率中可指定多个策略，并通过Select Policy选择最终的目标值（Min, Max, Disabled）。

HPA的创建、查看和删除命令与Pod类似。

Ingress

Ingress实现了验收指南中的所有功能，配置和使用示例如下：

```
# 1. Create an ingress: rkubectl create -f ingress.yaml
```

```
# 2. Check the ingress: rkubectl get ingresses
# 3. Invoke a request to the ingress, eg.
#    * http get server.minik8s.com/nginx
#    * http get server.minik8s.com/httpd
#    * http get tomcat.minik8s.com
kind: Ingress
metadata:
  name: mix
spec:
  rules:
    - host: server.minik8s.com
      paths:
        - path: /nginx
          service:
            name: nginx-service
            port: 80
        - path: /httpd
          service:
            name: httpd-service
            port: 80
    - host: tomcat.minik8s.com
      paths:
        - path: /
          service:
            name: tomcat-service
            port: 80
```

GPU

CUDA示例

main里面有两个函数，分别测试矩阵加法与矩阵乘法。加法与乘法均直接使用一维N*N数组表示二维(N,N)矩阵。

加法的逻辑大概是首先要为CPU这边使用的变量malloc内存，再为GPU上使用的变量cudaMalloc内存，然后使用cudaMemcpy（第四个参数使用cudaMemcpyHostToDevice）把数据copy到GPU设备使用的内存里，然后调用核函数。核函数用__global__修饰，逻辑是通过block号，block大小，thread号算出自己的全局索引，然后使用这个索引计算自己该计算的内容，这样GPU上的多个线程可以并行执行算各个不同部分，最终得到结果。结果会被用cudaMemcpy（第四个参数使用cudaMemcpyDeviceToHost）从GPU copy回内存中。这里还使用CPU计算了一遍，并比较了两者的时间以及结果是否一样。

乘法的逻辑类似：malloc，cudaMalloc，cudaMemcpy，运行核函数，cudaMemcpy取回结果，最后打印了第一个元素出来。乘法这边核函数取得的索引是作为矩阵的行索引，然后每次会算一行，最终也达到并行的效果。

Makefile作为CUDA程序的编译脚本。

具体示例代码见代码文件（/examples/gpu-code/cuda_example.cu）。

GpuJob

GPU实现了验收指南中的所有功能，配置和使用示例如下：

```
apiVersion: batch/v1
kind: GpuJob
metadata:
```

```
  name: cuda
spec:
  gpuConfig:
    slurmConfig:
      partition: dgx2
      totalCoreNumber: 1
      ntasksPerNode: 1
      cpusPerTask: 6
      gres: gpu:1
      scripts:
        [
          "ulimit -s unlimited",
          "ulimit -l unlimited",
          "module load cuda/10.1.243-gcc-9.2.0",
          "./cuda_example",
        ]
    compileScripts: make
```

- 用户使用kubectl创建GpuJob。minik8s会通过创建pod去执行的方式来完成job。GpuJob Controller会保证GpuJob的执行。

```
rkubectl create -f gpu-job.yaml -c gpu.zip
```

- 用户可使用kubectl get查询job的执行状态：

```
rkubectl get gpu-jobs
NAME     DESIRE     ACTIVE     FAILED     SUCCEEDED
```

根据设定的gpujob.spec.completions（默认为1），将会有指定数量的pod被创建并完成。新的pod会不断被创建并完成，直到SUCCEEDED状态的pod达到指定数量。

根据设定的gpujob.spec.parallelism（默认为1），最多只会有指定数量的pod同时处于ACTIVE状态。

- 用户可使用kubectl log获取job的执行结果。结果除了包含CUDA程序的输出，还会包含一些GPU Server与超算平台的交互信息。

```
rkubectl logs cuda -j
```

Function

Function实现了验收指南中的所有功能，配置和使用示例如下：

```
kind: Function
metadata:
  name: test
spec:
  maxReplicas: 10
  metrics:
    type: Function
    name: test
    target: 30
```

- 用户使用kubectl create命令上传函数。用户需提前准备好代码文件，文件需要是一个zip。

```
rkubectl create -f simple.yaml -c function.zip
```

- 用户可使用curl url的方式通过http访问函数，函数的执行结果会被返回。

```
curl http://test.func.minik8s.com
```

- 用户可使用kubectl patch命令对函数进行更新。运行该命令后，正在运行的pod会被删除，新的函数镜像会覆盖掉原先的函数镜像，当用户再次访问该函数时访问到的就是更新后的函数了。

```
rkubectl patch -f simple.yaml -c function.zip
```

- 用户可使用kubectl delete命令删除函数。执行命令后，该函数相关的service，replicaset，HPA都会被删除。

```
rkubectl delete functions test
```

Workflow

Worflow实现了验收指南中的所有功能，配置和使用示例如下：

```
# 1. Create functions needed by the workflow: add, minus3, guess, right,
wrong
# 2. Create the workflow: rkubectl create -f workflow.yaml
# 3. Get the host of the workflow: rkubeclt get workflows
# 4. Invoke the workflow: http get guess.workflow.func.minik8s.com a:=3 b:=4
kind: Workflow
metadata:
  name: guess
spec:
  startAt: add
  states:
    add:
      type: Task
      resource: add
      next: minus3
    minus3:
      type: Task
      resource: minus3
      next: guess
    guess:
      type: Choice
      rules:
        - type: FieldNumEquals
          field: ans
          content: 10
          next: right
      default: wrong
    right:
      type: Task
      resource: right
    wrong:
      type: Task
      resource: wrong
```

工程

本项目使用Jetbrains Space作为统一开发平台。

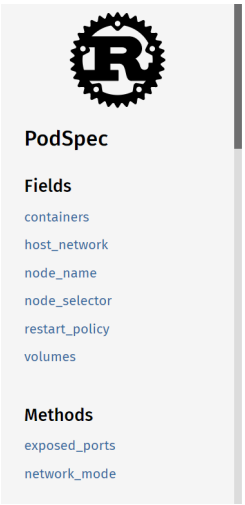
Gitee仓库（仅为Jetbrains Space中仓库的镜像）地址：https://gitee.com/markcty/mini-k8s

- 使用Cargo Workspaces管理Monorepo
- 开发流程遵循Gitflow，Commit规范遵循Conventional Commits
- Code Review共37次，Merge分支35次，每个PR都经过一人或以上审核通过

- 所有代码经过rustfmt格式化，代码构成：

Language	Files	Lines	Blank	Comment	Code
Rust	93	12905	1153	807	10945
YAML	40	868	56	49	763
Bourne Shell	17	743	78	65	600
Toml	12	290	21	0	269
CUDA	1	183	26	2	155
Docker	22	124	0	0	124
Markdown	4	112	27	0	85
Python	7	71	22	0	49
JSON	3	18	0	0	18
PHP	1	7	0	0	7
Makefile	1	5	1	0	4
Plain Text	5	1	0	0	1
Total	206	15327	1384	923	13020

- CI/CD使用Jetbrains Space Automation
  - Cargo Clippy自动化静态代码分析和风格规范
  - Cargo Test运行代码和文档注释中的测试
  - Cargo doc自动生成文档



PodSpec

Fields

containers
host_network
node_name
node_selector
restart_policy
volumes

Methods

exposed_ports
network_mode

Fields

`containers: Vec<Container>`
List of containers belonging to the pod. Containers cannot currently be added or removed. There must be at least one container in a Pod. Cannot be updated.

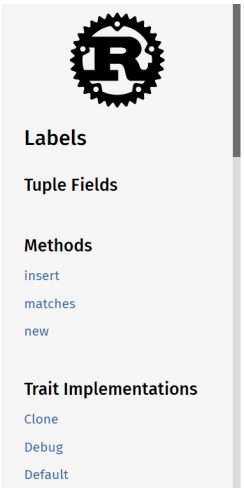
`volumes: Vec<Volume>`
List of volumes that can be mounted by containers belonging to the pod.

`restart_policy: RestartPolicy`
Restart policy for all containers within the pod. One of Always, OnFailure, Never. Default to Always.

`host_network: bool`
Host networking requested for this pod. Use the host's network namespace. If this option is set, the ports that will be used must be specified. Default to false.

`node_selector: Labels`
NodeSelector is a selector which must be true for the pod to fit on a node. Selector which must match a node's labels for the pod to be scheduledon that node.

`node_name: Option<String>`
NodeName is a request to schedule this pod onto a specific node. If it is non-empty, the scheduler simply schedules this pod onto that node, assuming that it fits resource requirements.



Labels

Tuple Fields

Methods

insert
matches
new

Trait Implementations

Clone
Debug
Default

```
impl TryFrom<&'_ String> for Labels
```
source

```
fn try_from(s: &String) -> Result<Labels>
```
source

Parse a string of the form "key1=value1,key2=value2" into a Labels.

Examples

```
use resources::objects::Labels;

let labels = Labels::try_from(&"key1=value1,key2=value2".to_string()).unwrap();
assert_eq!(labels.0.get(&"key1".to_string()), Some(&"value1".to_string()));
assert_eq!(labels.0.get(&"key2".to_string()), Some(&"value2".to_string()));
```

Errors

```
use resources::objects::Labels;

let labels = Labels::try_from(&"key1=value1,key2".to_string());
assert!(labels.is_err());
```

- 自动构建Docker Image发布至minik8s.xyz自建Registry
- 自动构建二进制可执行文件发布至一台文件服务器