Mini-kubernetes 验收文档

hjk，tyc，lsh

2022.6

一.目录

二. 系统架构概述 2
三. 依赖和库 4
四. 演示视频 4
五. 版本管理、开发流程和工程源代码结构 5
1. 源代码仓库和分支 5
2. CICD 6
3. 软件测试方法 6
4. 新功能开发流程 7
六. Etcd 及消息广播和传递机制 7
七. Node 的加入、注册和删除 7
1. Node 的加入 7
2. Node 的 heartbeat 8
3. Node 的删除 8
八. Pod 创建、管理、删除和内部通信 8
1. Docker 接口的实现 8
2. Pod 的创建和删除 9
3. Pod 的管理 9
4. Pod 中 localhost 通信、volume 共享和重点参数实现 9
九. Kube-proxy、Pod 间通信、service 和跨集群通信 10
1. 跨集群使用浮动 IP 通信的实现 10
2. Pod 与 Pod 通信的实现 11
3. Service 的创建和删除 11
4. Kube-proxy、Service 通信的实现和负载均衡策略 11
5. Service 在 Pod replica 创建和删除时的更新 12
6. Service 与 pod 匹配机制、selector 对多 Pod 的支持 13
十. Controller、Deployment 的管理、AutoScaler 的管理和动态扩容策略 13
1. Deployment Controller 13

2. AutoScaler Controller 13
3. AutoScaler Controller 的水平扩容/缩容策略 13
十一. Scheduler 和多机调度策略 14
1. Scheduler 的工作流程 14
2. Scheduler 的调度策略 14
3. Scheduler 的初始化 15
十二. Gateway 和 DNS 功能实现 15
1. 域名解析的实现 15
2. Gateway 的实现 16
3. DNS 功能的创建、运行和删除 16
十三. Activer 和 serverless 功能实现 16
1. 函数的创建和删除 16
2. http trigger、冷启动、负载均衡和动态扩缩容（包括 scale-to-0） 17
3. 函数的更新 18
4. StateMachine(workflow)的定义和实现 18
5. Activer 的实现 19
十四. GPU 功能实现 19
十五. 容错和控制面重启 20
十六. 分工和贡献概览 21
十七. 附录一：kubectl 交互格式和报错处理 21
1. Kubectl 提供的命令接口及其功能 21
2. Kubectl 的异常处理和报错 24
十八. 附录二：function 源代码环境支持和代码规范 24
十九. 附录三：pipeline 流水线 CICD 脚本 25
二十. 附录四：ETCD 中的存储结构 26

二. 系统架构概述

Mini-kubernetes 主要包括如下组件：

- **Kubectl**：部署在 master 节点

负责接收用户指令，将其解析后发送给 ApiServer，并解析 ApiServer 的 response 在控制台输出，支持的命令和 yaml 文件格式仿照 kubernetes 定义

- **ApiServer**：部署在 master 节点

(1) 提供一系列 RESTful 接口供 kubectl(和其他组件和插件，如 Activer)调用，对指令做

具体处理

(2) 提供 mini-kubernetes image factory 功能，在基础镜像的基础上自动生成镜像和对应的 Pod 和 Service 并部署

- **Controller**：部署在 master 节点

包括三部分，实现为 3 个 goRoutines：

（1） Node Controller：负责接收 kubelet 的 heartbeat，设置 node 的状态(Ready/NotReady)

（2） Deployment Controller：负责监控 Deployment 中 replica 的数量和运行情况：在 Pod replica 数量减少/Fail 时补充 replica

（3） Autoscaler Controller：负责监控 AutoScaler 中的 replica 数量、运行情况和资源用量，根据相关指标和用户设置完成动态水平扩容/缩容

- **Scheduler**：部署在 master 节点

负责将 Pod replica（在 mini-kubernetes 中常使用 PodInstance）调度到节点处，提供多种调度策略，可以在编译时/yaml 中指定

- **CoreDNS**：部署在 master 节点

负责解析域名，使用开源实现

- **Activer**：可以部署在任意节点和多实例部署，目前部署在 master 节点

负责处理 serverless function 和 state machine 的 http trigger，调用 apiServer 提供的接口对 function 做冷启动和动态伸缩

- **Kubelet**：每个 node 对应一个 kubelet

使用 GoRoutines 实现如下工作流：

（1） Pod Routine：Watch Etcd，对节点上 Pod 的增删指令做对应的处理

（2） Heartbeat Routine：定时向 node controller 发送 heartbeat

（3） Pod Check Routine：定时获取当前节点上正常运行的容器(通过 Cadvisor client)，从而判断容器是否健康运行

（4） Resource Monitor：定时获取当前节点上各容器占用的资源（CPU, Memory, 对 IO 和网络有预留接口， 通过 Cadvisor client）和 node 节点空闲的物理资源（CPU, Memory）,统计各 Pod replica 占用的资源，将 Pod replica 占用的资源和 node 空闲的资源存储进 etcd 中

- **Kube-proxy**：每个 node 节点各运行一个

提供统一的代理服务，根据映射关系将对 service 的访问转发到对应的 pod 上。监听 API server 中 service 和 endpoint 的变化情况，并通过 iptables 来为 service 配置负载均衡策略（这里我们采用了 round robin 的方式）

- **Cadvisor**：每个 node 对应一个

用于监控节点上的全部 docker container，获取 ID 列表及每个 Container 的资源使用(CPU 和 memory)

- **Etcd**：每个运行 mini-kubernetes 节点（包括 master 和 node）的物理机上各运行一个

充当 mini-kubernetes 的元数据持久化容器，消息队列，消息广播工具和 Watch 工具

此外，Mini-kubernetes 内置如下几种基础镜像用于自动生成相应的 pod 和 service

- **Gateway**：内置 Spring Cloud Zuul 框架和 maven、Java 环境，用户定义 Gateway 时由 apiServer-image factory 填入路由配置信息并编译，重新打包上传得到新的镜像
- **Serverless_template**：内置 Django 框架和 Python 环境，用户定义函数时服务时由 spiserver-image factory 填入函数内容后拉取依赖，重新打包上传得到新的镜像

- **GPU Job Uploader**：内置 ssh 和 Python http client，用户定义 GPU Job 时由 spiserver-image factory 填入 cuda 程序，Makefile 和 Slurm 定义后打包得到新的镜像，可以自动上传和提交 gpu 任务并用轮询的方式判断任务完成情况，在任务完成后下载结果并输出至用户指定的文件目录，并通知 apiServer 结束 Job.

三. 依赖和库

Mini-kubernetes 使用 Go 语言开发，在开发预定义基础容器镜像时使用了 python，Java 和 C，在开发过程中使用了如下开源库和开源组件：

- CoreDNS
- Cadvisor
- Etcd
- Docker client
- Go 后端框架 Echo(github.com/labstack/echo)
- Go 定时任务库 cron(github.com/robfig/cron)
- Go 命令行开发框架 cli(github.com/urfave/cli)
- Json, yaml 解析相关基础库（encording/json，gopkg.in/yaml）
- 网络相关基础库（net/http, github.com/monaco-io/request）
- 系统相关基础库(os，syscall)
- 其他 Go 提供的基础库(fmt, sync, math, time, strconv, bytes, strings, log, bufio, io)
- （Java）Spring Cloud Zuul(Gateway 基础镜像中提供 gateway 服务)
- （Python）Django(functional 基础镜像中提供网络服务)

没有使用 CNI 插件

四. 演示视频

Mini-kubernetes 实现了文档要求的所有功能。

编号	视频内容	视频链接
1	对应验收文档 1 和 2 演示多机（1master+3worker）部署 minik8s，创建 pod，验证 localhost、volume，多机调度和删除 pod，describe/get pod	video - OneDrive (sharepoint.com)中的“1.node&pod.mp4”
2	对应验收文档 3 和 4 演示部署 deployment，多机分配 replica，在 deployment 的基础上部署 service，验证 service 的多机可访问性，验证 service 的负载均衡策略，deployment 中 pod 停止工作后自动补充，停止工作的 pod 被 service 自动剔除，自动补充的 pod 被 service 自动加入，deployment 和 service 的删除和 describe/get	video - OneDrive (sharepoint.com)中的“2.service&deployment.mp4”
3&3+	对应验收文档 5	video - OneDrive

	演示 auto-scaler 的部署,通过改变 CPU 和 memory 的负载（cpu_boom 和 memory_boom）演示扩容和缩容，并演示 auto-scaler 扩容和缩容后建立在其上的 service 仍可正常多机访问，auto-scaler 的删除、describe/get	(sharepoint.com) 中的“3.autoscaler_mem.mp4”和“3+.autoscaler_cpu.mp4”
4	对应验收文档 6 演示创建 DNS，在物理机上通过“域名+路径”访问 service，在容器内部通过“域名+路径”访问 service，DNS 的 get/describe 和删除	video - OneDrive (sharepoint.com) 中的“4.dns.mp4”
5	对应验收文档 7 演示控制面崩溃后容器仍然正常运行，控制面重启后容器和 service 仍可正常访问并在 kubectl 中查看状态	video - OneDrive (sharepoint.com) 中的“5.control_plane_restart.mp4”
6	对应验收文档 8 演示提交矩阵乘法和矩阵加法来创建 GPU JOB，GPU JOB 的 get/describe，GPU JOB 的自动停止和输出结果，GPU JOB 的 delete	video - OneDrive (sharepoint.com) 中的“6.gpu.mp4”
6+	对应验收文档 8，起佐证作用 演示 GPU JOB 在容器内部运行的全过程，包括任务的上传、提交、运行、判断结束、结果下载、删除和发送结果	video - OneDrive (sharepoint.com) 中的“6+.gpu-in_docker_view.mp4”
7	对应验收文档 Serverless 自选功能的 1、3、4 演示 function 的创建(仿照 AWS，minik8s 中对应 activity 类型)、get/describe、冷启动、http trigger（即 invoke）、扩容至大于 1 实例，scale-to-0，更新(update)和删除	video - OneDrive (sharepoint.com) 中的“7.function.mp4”
8	对应验收文档 Serverless 自选功能的 2 演示 workflow（仿照 AWS，minik8s 命名为 state machine）的创建，get/describe，http trigger 及其在不同输入下的不同 workflow（根据条件进入不同分支），以及 state machine 的删除	video - OneDrive (sharepoint.com) 中的“8.statemachine.mp4”

五. 版本管理、开发流程和工程源代码结构

1. 源代码仓库和分支

本项目使用 Git 管理，远端仓库地址为 https://gitee.com/jkhe/mini-kubernetes
本仓库的 Git 分支及其作用为：

- master：项目主线，保存已经完成和通过测试后通过 pr 合并审查的分支的功能和组件代码；
- hjk/tyc/lsh：各组员分别对应以名字缩写命名的 Git 分支，用于保存自己代码修改的提交，在完成一个功能（如一个组件、一个功能或修改一个 bug）并进行测试后通过提交 pull request，经过审查后并入主线。

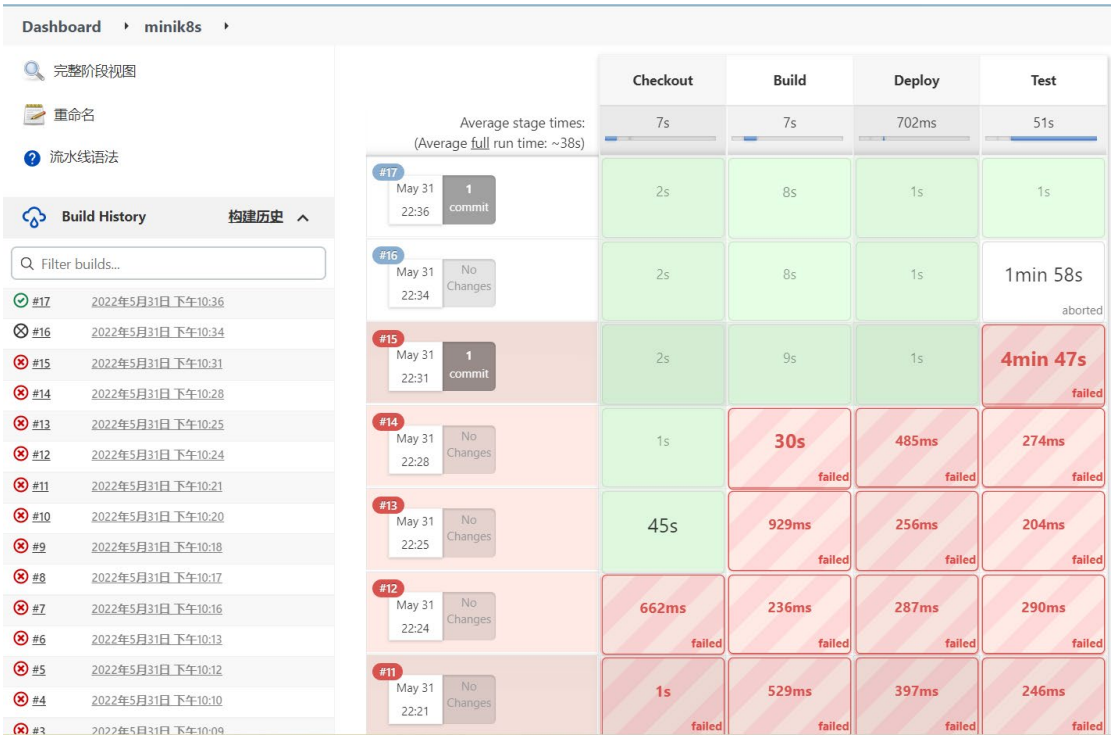
2. CICD

本项目使用 Jenkins 进行 CICD，在项目的测试服务器上部署了 pipeline 服务，在 Git 发生 push 和 pr 时进行如下 pipeline 步骤：

- 更新本地仓库
- 单元测试：运行仓库/test 目录下的单元测试
- 编译测试和部署：编译各组件并将可指定文件放置在指定位置
- 启动集群：启动集群中各组件实现部署

相关的 pipeline 脚本见附录，单元测试、编译、启动集群的具体操作写在仓库根目录下的 Makefile 中，脚本中通过 make+参数调用

CICD 运行记录：



3. 软件测试方法

在开发过程中，要求每个重点函数在开发完成后进行单元测试，每个功能/组件加入集群后进行系统测试和回归测试

- 单元测试：测试单独的功能点和函数（如解析各 yaml 定义为 go struct 的函数，docker exec 的函数等），为简化开发流程只测试复杂度高的重点函数，对应的测试代码位于/test 目录下，单元测试使用功能测试（黑盒测试）和白盒测试（如分支和条件覆盖）相结合的方法进行
- 系统测试和回归测试：测试集群运行状态下功能（如 Pod 的创建和删除等功能点）的

运行，既测试新加入的功能可以正常运行，又测试之前实现的功能不受影响。测试脚本位于/presentation 下，系统测试和回归测试通过运行具体功能的脚本的方式进行，测试用例经过多次迭代和修改其中一部分最终作为答辩演示时使用的测试用例

4. 新功能开发流程

- 负责人提出具体功能实现细节和对系统结构的修改，小组讨论后确定实现方式
- 编写代码，编码过程中对复杂度高和功能重要的重点函数进行单元测试，编码过程中可能对实现方式做讨论和修改
- 代码编写完成后进行编译测试、系统测试和回归测试，测试已经完成的功能不受影响，新加入的功能可以正常运行，过程中可能对实现方式做讨论和修改
- 测试完成后提出 pr，组长在审查（主要是阅读代码统一编码风格和运行系统测试脚本）通过后并入主线
- 功能或接口比较复杂时，负责人需要编写说明文件供参考（放置在/doc 目录下）

六. Etcd 及消息广播和传递机制

不同于 kubernetes 使用消息队列进行消息广播和传递，Mini-kubenetes 使用 Etcd 既作为控制面数据持久化容器，又充当消息队列。Etcd 运行在集群的各个节点上，除 kube-proxy 外的全部组件都会连接 etcd 的 client.

Mini-kubenetes 集群中各组件把自己的所有重要数据都存进 etcd 中，这使得几乎所有组件都实现为无状态，方便了控制面重启功能的实现。

消息队列通过 Watch Etcd 中的某个 key 的方式实现，例如，要通知 kubelet 对本 Node 上的 Pod 做增删，只需要修改该 Node 对应的 per node pod replica ID list。kubelet 会 watch 该 list，如果该 list 有 Pod replica ID 增加，kubelet 会从 etcd 中取出该 ID 对应的 pod replica 的详细信息并在节点上创建对应的 Pod，同理，如果该 list 有 Pod replica ID 删除，kubelet 会在本 node 上删除对应的 Pod 并在 etcd 中更新该 ID 对应的 Pod replica 的状态和信息。这大大减少了集群中消息通知的复杂度，只需要通过在 Etcd 中维护完整的集群状态记录即可实现对集群状态的更新和控制，kubelet 等节点会不断同步 Etcd 中的集群状态到物理世界(如创建和删除 Pod 实体)，etcd 中维护的数据相当于时 Mini-kubernetes 集群的数字抽象。

Mini-kubenetes 通过设计 Etcd 中各字段的修改者和读取者，保证各字段读写在不加锁的情况下不会出现并发冲突，此外，通过设计字段存储顺序的方式保证了 Etcd 中数据的一致性和崩溃一致性，如首先持久化一个抽象，再将此抽象的索引（如 ID）加入全局 list。

Etcd 中的存储结构和字段请见附录.

七. Node 的加入、注册和删除

Worker Node 抽象的管理和 Worker Node 状态的维护由 Node Controller 负责。

1. Node 的加入

要启动 Mini-kubernetes 集群，用户需要手动启动 Master Node，之后再各 Worker Node 上启动 kubelet，启动 kubelet 时用户需要指定 Node 的 name 和 master 节点的

注册 url，Worker Node 加入集群的过程如下

- Node 上的 kubelet 将自己的 name、浮动 IP 等信息包含在注册请求中发送给 Node Controller
- Node Controller 收到注册请求后将这些基本信息记录在 etcd 当中，之后 Node Controller 为每个 node 分配唯一的 nodeID 和 subnet IP 区间，其中 nodeID 会被加入到 etcd 中的 node ID list 中（scheduler watch 此条目来获取目前集群中的全部 Node），这些数据（nodeID 和 subnet IP 区间）会作为注册请求的 response 返回给 kubelet。
- Node 上的 kubelet 根据该 subnet IP 区间创建自己的 miniK8S-bridge 网桥，我们的 miniK8S 集群中的网络通信，大部分就是依托于自建的 miniK8S-bridge 网桥实现的。

2. Node 的 heartbeat

集群中所有的 kubelet 需要每 1min 向 Node Controller 通过 http POST 的形式发送 heartbeat，Node Controller 会根据 heartbeat 更新存储在本地内存中的最近一次收到 heartbeat 的时间。每隔三分钟 Node Controller 会检查各 Node 的最近一次收到 heartbeat 的时间，如果距今超过 3min 则将该 Node 的状态由 Ready 改为 NotReady，**Scheduler 不会调度 Pod 到状态为 NotReady 的节点上，该节点上所有 Pod 的状态也将被设为 Unknown(表示失联)。**下次检查时如果最近一次收到 heartbeat 的时间符合条件（距今小于 3min），Node Controller 会将该 Node 的状态重新设置为 Ready

3. Node 的删除

用户可通过 kubectl 向 apiserver 发送 http 请求，要求将某一 node 从集群中移除。Apiserver 收到 http 请求后，将其 node ID 信息从 Node ID List 中移除（因此之后不会再有 Pod replica 调度到该 Node 上），并在 etcd 中将存储的该 node 信息删除。之后，master 和该 node 分别断开与彼此的 vxlan 隧道连接。

删除 Node 意味着 Node Controller 不会再检查 Node 的状态（Ready/NotReady），Scheduler 不会再将 Pod replica 调度到该 Node 之上，集群中 IP table 的修改也不会同步到该 Node 上，由于各 Node 与被删除 Node 的 vxlan 隧道被断开，该 Node 和其他 Node 的跨节点通信也无法进行。但是 Mini-kubernetes 集群不会删除在该 Node 上运行的 Pod replica，只会在 etcd 状态中将这些 Pod replica 的状态设置为 FAILED，为避免正在运行的任务受到删除 Node 的影响，集群管理者应保证删除 Node 时该 Node 上没有正在运行的 Pod，或删除 Node 后将受到删除 Node 影响的任务重新部署。

八. Pod 创建、管理、删除和内部通信

1. Docker 接口的实现

我们通过对 Docker client 库的相关函数和类型进行封装，实现了对容器（包括创建、启动、停止、删除、重启、查看、配置启动命令、将宿主机文件 copy 进容器等）、镜像（制作、打上 tag、上传、拉取等）和容器网络（创建、查看、删除等）的相关操作。

2. Pod 的创建和删除

不同于传统 K8S 中的 pod，考虑到后续 replicas 的存在，我们在这里定义了两种结构体：pod 和 podInstance（也会被称为 Pod replica，包含 Pod replica 的具体信息，如运行的节点和状态）。具体地，前者是 yaml 文件解析的直接结果，可以看作是后者的元数据；后者相当于传统 K8S 当中的 pod，可以看作是前者的具体实现。每个 podInstance 只能由一个 pod 创建而来，而一个 pod 可以用来创建多个 pod-Instance。

（1）Pod 的创建

- ApiServer：将 kubectl 发送来的 pod 和根据 pod 新创建得到的 podInstance 写入到 etcd 当中，并将该 podInstance 的 ID 添加到 etcd 中的 pod replica ID list 当中。
- Scheduler 通过 Watch Etcd 中的 pod replica ID list 发现有 Pod 被创建，将其调度到某个 Node 上运行，具体细节在第九节中详细说明.
- Kubelet： Node watch 在 etcd 中自己所处 node 的 per node pod replica ID list，发现 Pod replica ID 增加时使用 replica ID 从 etcd 中取出 pod replica 的详细信息，并在本 Node 上创建 Pod，具体操作为分别创建 Pod 中包含的各 Docker Container 并将其 ID 记录在 Etcd 中 PodInstance 的对应位置中

（2） Pod 的删除

- Apiserver：先查找是否存在该 podInstance，若不存在直接返回 404，通知 kubectl 该 podInstance 不存在；若存在，从 etcd 中的 pod replica ID list 中删除该 Pod replica 的 ID
- Scheduler：发现 etcd 中的 pod replica ID list 中该 Pod replica ID 被删除后，在该 Pod 所在 Node 的 per node pod replica ID list 中删除该 pod replica 的 ID
- Kubelet：发现本 Node 的 per node pod replica ID list 中该 Pod replica 的 ID 被删除后，在本节点删除该 Pod 对应的各 Docker 容器，并删除该 PodInstance 在 etcd 中对应的数据

3. Pod 的管理

Kubelet 在创建 Pod 后，会监控器运行状态并更新到 Etcd 中供 controller 使用，对容器的监控使用开源组件 CAdvisor 进行

- Kubelet 每隔 30s 通过 CAdvisor 获取当前物理机上正常运行的所有 Docker Container 的 ID，并和该节点上各 Pod replica 对应的 Docker Container 的 ID 对比，如果发现某 Pod 中的某个 Container 不在正常运行的 Docker Container 的 ID list 中，则将该 Pod replica 的 status 改为 FAILED 并持久化到 Etcd 中（kubectl 可以通过 get 查看， Deployment Controller 会借此判断各 replica 是否正常运行）
- Kubelet 每隔 30s 通过 CAdvisor 获取当前物理机上正常运行的所有 Docker Container 占用的资源，并和该节点上各 Pod replica 对应的 Docker Container 的 ID 对比，计算出该节点上各 Pod replica 占用的资源并存储在 etcd 中（AutoScaler Controller 会使用此记录计算各 Pod replica 平均资源占用并决定是否进行扩容/缩容）

4. Pod 中 localhost 通信、volume 共享和重点参数实现

（1） localhost 通信：每个 pod 中都会被默认创建一个 pause 容器，pod 内其他容器都通过 network=container 的网络模式共享该 pause 容器的网络地址空间。因而同一 pod 内

的所有容器便可以通过 localhost 实现互相通信

（2） PID、IPC 命名空间共享：在/etc/docker/daemon.json 里写入"default-ipc-mode": "shareable"来允许 IPC 命名空间可被容器共享。然后同样通过（1）中类似的模式指定来指定 pause 父容器的 PID、IPC 命名空间被同一 pod 内其他容器共享

（3） volume 共享：我们采用的方式是，在 Docker client 提供的 Mount 结构体中，指定容器的 mount 类型为 bind，从而将 host 机器的目录 mount 到 container 中。这样一来，不仅 pod 内容器可以实现文件共享，容器和物理机之间也可以实现文件共享

（4） 资源用量限制：通过配置 Docker client 提供的 Resources 结构体参数予以限制

（5） 命令与参数：通过配置 Docker client 提供的 Config 结构体参数来实现。yaml 文件中的 spec - containers – command 为容器的启动命令列表，将覆盖容器中的 Entrypoint，对应 Dockefile 中的 ENTRYPOINT，我们将其赋给 Config 中的 Entrypoint 字段；spec - containers – args 为容器的启动命令参数列表，对应 Dockerfile 中 CMD 参数，我们将其赋给 Config 中的 Args 字段

（6） 容器端口的暴露和映射：通过配置 Docker client 提供的 Config 结构体和 HostConfig 结构体的参数来实现

九. Kube-proxy、Pod 间通信、service 和跨集群通信

1. 跨集群使用浮动 IP 通信的实现

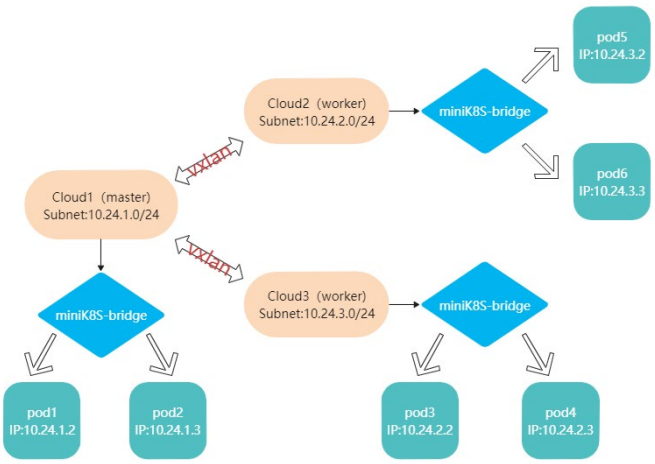
在新的 node 加入集群后，会尝试使用 open-vswitch 与当前存在于集群中的 master node 建立起 miniK8S-bridge 网桥之间的 vxlan 隧道，从而能够访问到其他 node 上的 pod。具体地，我们通过 open-vswitch 新建网桥 br0，将 miniK8S-bridge 和 br0 连接起来，然后将对应特定 IP 的 vxlan 隧道挂在 br0 上。

当 master 发现 etcd 中 node 的 ID list 发生变化时（通过 watch etcd 中的 node ID list 实现），也会根据其增减状况来尝试搭建和删除与其他 node 之间的 vxlan 隧道。

Vxlan 隧道原理解释：隧道是一个逻辑上的概念，在 vxlan 模型中并没有具体的物理实体向对应。隧道可以看做是一种虚拟通道，vxlan 通信双方认为自己是在直接通信，并不知道底层网络的存在。从整体来说，每个 vxlan 网络像是为通信的机器搭建了一个单独的通信通道，不同节点上的容器能够通过隧道直连。

如果将 vxlan 比作“网线”的话，假设 hostA 连着 hostB，hostB 连着 hostC，那么 hostC 上的 container 自然可以通过 hostB 找到 hostA 上的 container。所以我们在 master 和其余所有 node 之间建立起双向的 vxlan 隧道，就可以实现集群内通过任意浮动 IP 跨 node 通信。

对于三台跨子网机器而言，其网络抽象可以用下图表示：



2. Pod 与 Pod 通信的实现

（1）相同 node 内的 pod：每个 node 在向 Node Controller 注册后会被返回一个不与其他 Node 重叠的 subnet IP 区间，node 据此创建自己的 miniK8S-bridge 网桥，此后创建的所有 pod 的 IP 都位于该 subnet 区间中，各 Pod 的网络都接入此网桥中。具体地，每个 pod 中都会被默认创建一个 pause 容器通过 network=bridge 的网络模式接入到 miniK8S-bridge 网桥中，被分配到网桥中的唯一 IP，pod 内其他容器都通过 network=container 的网络模式共享该 pause 容器的网络地址空间。同一 node 内的所有 pod 便可以根据 pod IP 经由 miniK8S-bridge 网桥自由进行通信。

（2）node 的 pod 通信：由于每个向 Node Controller 注册的 node 都会被分配一个不与其他 Node 重叠的 subnet IP 区间，而每个 node 中创建的 pod 所具有的 IP 也都处于这不相重叠的地址空间内，所以这样就保证了集群内 pod IP 的唯一性。不同物理机上的 pod 之间便可以根据彼此的 pod IP 实现跨 node 通信。为了实现这一点，我们还需要在路由表里添加规则，使得对其余 node 的 subnet 地址空间的访问（即对 IP 处于该地址空间内 pod 的访问）都会被转发到本机的 miniK8S-bridge 网桥上，进而通过 vxlan 隧道到达目标 pod 所在 node 的 miniK8S-bridge 中，并最终被发送到目标 pod 的 IP 上。

3. Service 的创建和删除

Apiserver 收到 Kubectl 发送的创建 Service 的请求后，会先尝试在 etcd 中根据 name 查找 service，若发现同名 Service，则直接返回 404 告知 Kubectl 不可重复创建。否则进入创建 service 的流程：

- ApiServer 在 etcd 中查找符合该 Service 映射条件的全部 pod replica，将其 IP 和 port 信息添加到 Service 的信息当中，再将 Service 的信息存储到 etcd 中。
- Apiserver 通过 http 请求告知所有 node 上的 Kube-proxy 更新 iptables。

Service 的删除流程为通知 kube-proxy 清除创建时对 IPtable 的修改并删除 etcd 中存储的与此相关的信息。

4. Kube-proxy、Service 通信的实现和负载均衡策略

为了实现 Service 的功能需求，我们在 iptables 内新建了 mK8S-SERVICES 链，负责处理对于 Service 的访问，并把抵达 mK8S-SERVICES 链的相应转发规则加入到

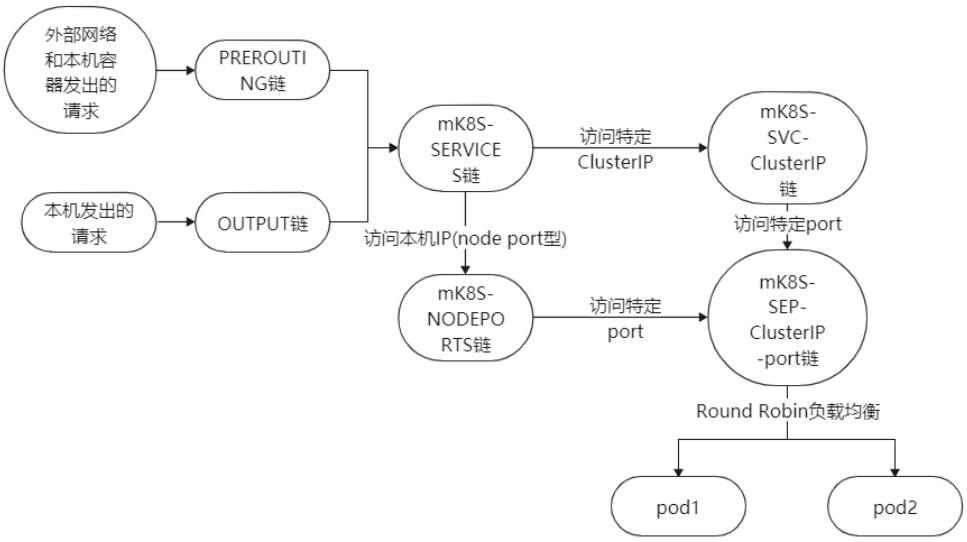
PREROUTING（处理外部网络和本机容器发出的请求）和 OUTPUT（处理本机发出的请求）两条链中。

对于每一个 Service，我们都依据它的 ClusterIP（集群中唯一）创建一个 mK8S-SVC-ClusterIP 链，并在 mK8S-SERVICES 中添加规则，使得所有对该 ClusterIP 的访问都被转发到这条链上。

接着，对于该 Service 的每一个 port，我们也都创建一个 mK8S-SEP-ClusterIP-port 链，并在 mK8S-SVC-ClusterIP 链中添加规则，使得所有对该 ClusterIP 这一 port 的访问都被转发到这条链上。在这条链上，我们使用 round robin 的方式实现负载均衡，将请求转发到所有涉及到的目标 pod 上。这样，我们就实现了访问 ClusterIP 类型 Service 的通信。

NodePort 是在 ClusterIP 的基础上实现的，简而言之就是通过修改 iptable 的方式将对于物理机 port 的访问转发到该 port 对应的 NodePort service 的 ClusterIP 上。为了支持 NodePort 类型的 Service，我们还创建了 mK8S-NODEPORTS 链，并在 mK8S-SERVICES 链末尾添加规则：只要 dst IP 是 node 本机 IP 的话（–dst-type LOCAL），就跳转到 mK8S-NODEPORTS 链做进一步判定。在 mK8S-NODEPORTS 中添加规则，根据yaml 中指定的 nodeport 和 service port 的对应关系，将对于暴露给外界的特定 NodePort 的访问都转发到对应的 mK8S-SEP-ClusterIP-port 链上进行处理。这样，我们就实现了访问 nodeport 类型 Service 的通信。

通过 IP table 处理网络请求的工作流如下图所示：



5. Service 在 Pod replica 创建和删除时的更新

当新的 pod 创建和删除时，ApiServer 都会去 etcd 中找出所有符合该 pod 映射条件的 service，更新其信息并存储在 etcd 中。

而当集群内的 service 发生更新（新增、删除、作为 endpoint 的 pod IP 发生改变）时，都会由 ApiServer 同步 etcd 中的 service 数据，同时 ApiServer 还会发送 http 请求，通知每个 node 上的 kube-proxy 更新本机的 iptables。我们的 Service（包括 ClusterIP 和 NodePort 类型的 Service）通信和相关的负载均衡策略均通过修改 iptables 实现。

通过这种方式，我们保证了 Service 的转发调度可根据 pod 的加入删除而动态调整。

6. Service 与 pod 匹配机制、selector 对多 Pod 的支持

在确定 service 中包含的 Pod 时，我们将 pod 的 metadata.label 字段与 service 的 selector.name 字段进行比较，若二者相同，即说明该 pod 符合该 service 的映射条件，接下来我们再处理它们端口的匹配关系。Service 的 PortsBinding 结构体记录了特定 Service 端口与特定容器端口的映射关系，以及需要转发到的容器 IP List。因而即使 Service 的一个端口映射到了多个 pod 内容器的同一端口上，我们同样可以实现转发机制。

我们在 Service 当中维护了 PortsBinding 列表，以记录多个不同的端口映射关系。因而，即使具备相同 metadata.label 的不同 pod 与 Service 上的各个端口存在不同的映射关系，我们都能很好地处理。

每次 Service 建立时都会遍历 Pod 列表来确定需要被包含在其中的符合条件的 Pod replica，每次有新 Pod replica 创建也会遍历全部 Service 来确定 Pod replica 是否需要加入 Service，因此 service 可以支持多个 Pod 的多个 replica，以及不断进行扩容缩容等删减更新操作的 Pod replica。

十. Controller、Deployment 的管理、AutoScaler 的管理和动态扩容策略

Controller 负责管理集群中的 Node、Deployment 的 AutoScaler，以三个 go routine 的形式分别实现了 Node Controller(已述)、Deployment Controller 和 AutoScaler Controller。

Controller 创建 Pod replica 和删除 Pod replica 均调用 apiserver 创建 Pod 和删除 Pod 的接口，此接口在创建 Pod replica 和删除 Pod replica 的同时会将 Pod replica 加入符合条件的 service 或将其从所处的 service 中删除。

1. Deployment Controller

Deployment Controller 主要有如下任务

- Watch etcd 中的 all deployment name list，当 deployment 新增时从取出其详细信息，调用创建 Pod replica 的接口创建对应数量的 Pod replica，当 deployment 删除时删除其对应的所有 Pod replica，并更新 Etcd 中 deployment 的状态
- 每隔 30s 检查一次 Deployment 的各 replica 是否正常运行（与 kubelet 配合），删除状态为 FAILED 的 Pod replica 并创建新的 Pod replica 补充

2. AutoScaler Controller

AutoScaler Controller 主要有如下任务

- Watch etcd 中的 all autoScaler name list，当 autoScaler 新增时从取出其详细信息，调用创建 Pod replica 的接口创建最小数量的 Pod replica，当 autoScaler 删除时删除其对应的所有 Pod replica，并更新 Etcd 中 autoScaler 的状态
- 每隔 30s 检查一次 autoScaler 的各 replica 是否正常运行（与 kubelet 配合），计算其平均资源用量（资源提供 CPU 和 Memory 两种可选项）并判断是否应该扩容/缩容

3. AutoScaler Controller 的水平扩容/缩容策略

AutoScaler Controller 的水平扩容/缩容策略可以概括为如下几点：低于最小平均值的

80%缩容，高于最大平均值的 120%缩容

- AutoScaler 创建时只创建最小数目的 Pod replica 作为启动时的初始 Pod replica
- 每 30s 检查一次各 replica 的运行情况和平均资源用量（与 kubelet 合作）
- 如果正常运行的 Pod replica 数目小于最小 replica 数目，则直接扩容至最小 replica 数目
- 如果各 Pod replica 的平均 CPU/Memory 用量小于最小平均值的 80%，则进行缩容，删除集群中占用资源最少的 Pod replica（首先，占用资源最少代表删除它影响的正在处理的请求数目最少，删除它对业务影响最小；其次，删除占用资源最少的 Pod replica 对提高平均资源用量的效果最显著）
- 如果各 Pod replica 的平均 CPU/Memory 用量大于最小平均值的 120%，则进行扩容，再创建一个 Pod replica

之所以每 30s 检查一次，是因为我们任务一个新的 pod 至少需要 30s 才能进入最大吞吐量处理请求的状态，这时对其资源用量进行分析才是有意义的。

之所以扩容和缩容的阈值分别对用户定义的最小资源平均用量和最大资源平均用量向下和向上偏移，是为了防止因为在并发请求数目波动的波峰波谷取样引起 Pod replica 数目的无意义波动，最大限度维持 Pod replica 数目的稳定，提高集群中 Pod replica 的利用率。

十一. Scheduler 和多机调度策略

1. Scheduler 的工作流程

Scheduler 掌管着将 Pod replica 调度到集群中合适的机器上的工作和向对应 kubelet 发送删除 Pod replica 命令的任务，具体工作流程如下：

- Scheduler 通过 Watch 接口 Watch Etcd 中的 all pod replica ID list，这是整个集群中正在运行的全部 Pod replica 的 ID 的集合
- 发现 pod replica ID list 有 Pod replica ID 增加时，代表有新的 Pod replica 需要调度，Scheduler 根据调度策略将其调度到某个 Node 运行，具体调度策略后述，通知 kubelet 的方法为在该 Node 对应的 per node pod replica ID list 中加入该 Pod replica 的 ID（kubelet 会 watch 所在 Node 对应的 per node pod replica ID list，并在其发生变动时同步更新到本地运行的 Pod，主要是创建和删除 Pod）。如果调度失败则将其加入到 fail to schedule pod replica ID list 中并持久化到 etcd，Scheduler 会每 30s 对该 list 中的 Pod replica 重新调度一次，直到存在合适的 Node 为止。
- 发现 pod replica ID list 有 Pod replica ID 被删除时，代表有 Pod replica 需要被删除，Scheduler 通过查看该 Pod replica 存储在 etcd 中的数据（其中有其运行在哪个 Node）确定其运行的 Node 并通知该 Node 将其删除，具体方法为在该 Node 对应的 per node pod replica ID list 中删除该 Pod replica 的 ID

2. Scheduler 的调度策略

Scheduler 的调度依据主要有如下几类信息：

- Pod 在定义时用户指定的 NodeSelector 的 With 字段（必须和哪个 Pod 运行在同一 Node）和 NotWith 字段（必须不和哪个 Pod 运行在同一节点）
- 用户定义的 Pod 的 Container 的 Resource 的 Request 字段，将该 Pod 中全部

Container 的该字段加和可以得到 Pod 运行所需的最小资源（主要是 CPU 和内存）

- 各物理节点的现存资源和使用情况，主要是物理节点的 CPU 空闲率、CPU 核数、空闲内大小和现在运行的 Pod 数目

Scheduler 的调度流程为

- 以全部 Node 的集合为起始
- 经过 With Filter 和 NotWith Filter 以及 ReSource Filter 筛选出符合 Pod 的 With 和 NottWith 要求以及最小空闲资源要求的节点
- 如果此时 Node 集合为空，则调度失败
- 如果 Node 集合非空，则选择最合适节点

选择最合适节点的依据有如下两种，可以在编译时确定

（1） 选择空闲硬件资源最丰富的节点：即选择

$$\text{空闲内存大小(MB)} + 1000 \times \left(1 - \text{CPU 平均占用率}\right) \times \text{CPU 核数}$$

最大的节点

（2） 选择现运行 Pod 数目最少的节点

在生产环境下，使用（1）号策略更有利于为 Pod 运行提供充足的资源，但在答辩验收环境下由于和 Master 节点运行在同一物理机上的 Worker Node 硬件资源显著少于其他 Worker Node，为避免该 Node 无 Pod 运行，答辩演示时使用了（2）号选择策略。

3. Scheduler 的初始化

合适的初始化流程是控制面重启的关键，Scheduler 需要在启动时需要初始化本地元数据，主要有如下几项

- 已经完成调度的 Pod replica ID list，只有和已经完成调度的 Pod replica ID list 对比才能知道需要调度和删除哪个 Pod replica，已经完成调度的 Pod replica ID list 通过从 etcd 中查询各 Node 的 per node pod replica ID list 并取并集来初始化，该 List 之后会在每次完成 Pod replica 调度或删除后更新
- 现有全部 Node 及各 Node 上的正在运行的 Pod replica ID，前者通过读取 etcd 中的 node ID list 来初始化，并会通过 watch 此 node ID list 实时同步，后者通过查询各 Node 的 per node pod replica ID list 来实现，并会在调度和删除过程中实时同步

此外，Scheduler 中还缓存了各 Pod replica 的概要信息来方便调度时减少 etcd 查询次数，主要是各 Pod replica 对应的 Pod Name 和 Pod Label，也会在启动时初始化并实时同步。

十二. Gateway 和 DNS 功能实现

1. 域名解析的实现

DNS 利用开源的 CoreDNS 组件实现域名解析，CoreDNS 启动后监听在 Master 节点的 53 端口。ApiServer 根据 yaml 文件的配置，将特定的域名经处理后作为 key，生成的 Gateway pod 的 IP 地址作为 value，存入 etcd 中。此后对于该域名的所有访问都将经由 coredns 转发到 Gateway pod 中进行处理。

为了使集群的 DNS 服务器为 CoreDNS，需要对云主机的配置修改：

- 在 master 的/etc/resolv.conf 中第一行写入 nameserver 127.0.0.1，在其余 node 的

/etc/resolv.conf 中第一行写入 nameserver + master 的浮动 IP，从而指定 DNS 解析的服务器。

- 在每台云主机的/etc/hosts 中第一行的 127.0.0.1 localhost 后写上本台云主机的名称，这是为了避免出现“unable to resolve host cloud1: Temporary failure in name resolution”类似的报错

2. Gateway 的实现

在 DNS 服务里，Gateway 的作用为根据用户的定义，将不同路径的请求分别转发到对应的 service 的 ClusterIP。

在 Mini-kubernetes 中 Gateway 以 Pod 的形式运行在集群中， DNS 会将域名解析为对应的 Gateway Pod 的 Pod IP，Gateway Pod 会将请求分别重定向到对应的 service 的 ClusterIP。

创建 Gateway Pod 的流程如下：

- ApiServer 根据 yaml 中 path 和 serviceName 的对应关系，结合 Etcd 中存储的 Service 和 ClusterIP 的对应关系，确定请求的 path 和 ClusterIP 的对应关系，并据此生成定义重定向关系的 yaml 文件（由于 Gateway 镜像的基础镜像使用的框架为 Spring Cloud Zuul，该 yaml 文件格式与 Zuul 的配置文件格式匹配)。
- ApiServer 中的 Image Factory 拉取基础镜像 zuul，并运行；之后使用 docker copy 接口将上一步确定的 yaml 配置文件写入预定路径，并调用 docker exec 接口完成对 Maven 工程（Zuul）的编译和打包，此时 Container 内已经有了可以正常运行的 Gateway 应用
- ApiServer 使用 docker commit 和 docker push 接口提交该 Container 为新的 Docker Image
- Apiserver 生成运行该 Docker Image 的 Pod，并复用 ApiServer 创建 Pod 的接口在集群中部署运行此 Gateway Pod。

3. DNS 功能的创建、运行和删除

- **创建：** Apiserver 在获取到 kubectl 发送来的 DNS 信息后，从 etcd 中查找出相应的 Service 并补充在 DNS 的路径解析信息当中，然后将完善后的 DNS 信息存储在 etcd 中，接着创建 Gateway pod
- **运行：** 当 master 上的 pod 或用户进行域名访问时，由于配置本机为域名服务器，运行在 53 端口监听 TCP 请求的 CoreDNS 会直接进行处理；当其他 node 上的 pod 或用户进行域名访问时，会根据/etc/resolv.conf 里的配置将 master 作为域名服务器。node 向 master 发送域名解析请求，然后由 master 上的 CoreDNS 处理。CoreDNS 将经处理后的域名作为 key，从 etcd 中查找出对应 Gateway pod 的 IP 地址。之后请求将发往 Gateway pod，Gateway pod 会根据配置文件里的配置将访问请求转发到特定的 Service 里。

十三. Activer 和 serverless 功能实现

1. 函数的创建和删除

用户定义函数时需要提供函数的源代码文件路径(源代码格式规范和代码环境支持见附

录，用户在源代码中可以使用环境提供的预先解析的请求体和请求 url 中附带的参数)，依赖文件路径（对于 Python 而言主要是 Requirments.txt)，以及函数的版本（为 update 准备)

ApiSever 收到用户提交的函数定义文件后：

(1) 首先制作函数镜像，ApiServer 运行预先准备好的 python_serverless_tmplate 镜像，使用包装了 docker copy 的 go 接口将 python 源代码和依赖文件写入预定义的位置，之后使用包装了 docker exec 的 go 接口运行 pip 命令在容器中拉取依赖，依赖拉取结束后，ApiServer 使用包装了 docker commit 和 docker push 的 go 接口提交该容器制作新的镜像（该镜像名字为 function 名字+版本号+UUID)，之后停止和移除该容器（有关 docker 接口的封装请见前文)

(2) 之后，ApiServer 自动生成运行该镜像的 Pod 定义文件并将其存储在 etcd 中，并自动生成可以访问到该类 Pod 80 端口的 ClusterIP 型 service 的定义，并在集群中运行该 service(由于此时集群中没有 service 对应的 Pod replica，该 service 目前无法提供正常访问，只是形式上部署)，之后，ApiServer 将 function 实例的详细信息记录在 etcd 中，并将 function 的 name 记录在 etcd 的 function name list 中（此时 watch 该 list 的 Activer 收到 function 创建的消息，并在内存中创建用于记录 function 并发访问数目的 map)，此时 function 的 create 完成

(3) ApiServer 创建用于 http trigger 的 url，其 ip 和端口为 function 分配到的 Activer 的 IP 和端口，具体为：activerIP:activerPort/function/`functionName`，Activer 在物理机浮动 IP 的端口运行，集群外也可以使用该 url 发起 http trigger，http trigger 的发起方式与发送普通 http 请求没有区别，开发人员使用 curl 或 PostMan 等 http 发送工具测试。Kubectl 会将这个 url 输出给用户。

Function 的删除使用需要 ApiServer 删除集群中正在运行的全部 Pod Replica，并在 Etcd 的 function name list 中删除 function 的 Name(Watch 该 list 的 Activer 会删除内存中用于记录的并发访问数的 map 条目)，之后删除 function 对应的 Service、function 和对应 Pod 的定义文件。删除 Pod replica 和 Service 复用了 Pod 和 Service 的删除接口。

2. http trigger、冷启动、负载均衡和动态扩缩容（包括 scale-to-0)

Activer 处理 http trigger 的流程为：

- 记录 function 访问（用于根据并发访问数动态伸缩)
- 查看集群中有无 Pod replica 运行，如果没有，启动冷启动流程
- 向 function 对应的 service 的 ClusterIP 发送转发 http trigger 的 http 请求，并将结果返回给用户

Function 在创建时并没有 Pod replica 在集群中运行，此时 function 的自动生成的 Docker Image 和 Pod 定义文件已经存储在 etcd 中，当 http trigger 到来时，Activer 接收到该 http trigger 并查找集群中运行的 Replica 数目，如果 replica 数目为 0 则 function 开始冷启动，Activer 复用 Apiserver 创建 Pod 的接口，使用预定义的 Pod 定义文件，要求 ApiServer 创建一个运行 function 定义的服务的 Pod，如上所述，这个 Pod 会自动加入 service 中。

Activer 会记录并发访问的次数，以此为依据进行动态扩容和缩容，动态扩容的缩容的策略为：**快速扩容、缓慢缩容**。Activer 每 30s 对各 function 进行一次动态实例数目调整，调整的目标为前 30s 内对应 function 收到的 http trigger 数目除以预定义的值后向上取整（在演示环境下，为方便演示扩容至>2 实例，这个值为 5，生产环境下可以调整至更大)，

如果现有实例数目大于目标值则进行慢速缩容，不管目标值为多少，每次缩容仅删除一个 Pod 实例，删除 Pod 实例复用 ApiServer 的 Pod 删除接口；如果实例数目小于目标值，则进行快速扩容，直接扩容至目标值。通过快速扩容和慢速缩容，集群可以以最快速度满足客户端要求，并最大限度利用已经创建的实例，防止因为并发请求数波动引起集群内 Pod replica 数目不断波动带来的开销，使集群内 Pod replica 的数目在波动的并发请求数下仍然保持相对稳定。新创建的和删除的 Pod 都会同步加入对应的 service 或从 service 中删除。

在负载均衡方面，Activer 并不直接负责 Pod replica 之间的负载均衡，function 只访问对应 service 的 ClusterIP，由 service 做不同 Pod 之间的负载均衡，这复用了 service 已经实现的基础功能，降低了集群复杂度。

由动态扩缩容策略可知，若一段时间内某 function 没有 http trigger，其对应的 Pod replica 将以每 30s 减少一个的速度逐步减少到 0，实现 scale-to-0。

3. 函数的更新

Mini-kubernetes 为用户提供了两种 function update 接口：

- Soft-Update：ApiServer 使用新版 function 定义重新创建 Docker Image 和 Pod 定义文件来替换原有的，之后扩容时创建的 Pod replica 均为新版 function，但已经运行在集群中的 Pod replica 不会删除，通过 function 的扩缩容将其逐步淘汰。Soft-Update 在更新函数过程中不会停止函数服务，适合接口向后兼容的函数的更新。
- Hard-Update：ApiServer 会使用新版 function 定义重新创建 Docker Image 和 Pod 定义文件来替换原有的，并删除集群中运行的旧版 function 实例，之后新版 function 重新进行冷启动流程。Hard-Update 相当于 Delete+Create，它一次性停止集群内全部旧版 function 的更新，但代价是函数服务会停止运行一段时间，适合因为漏洞和缺陷需要应急更新函数和接口不向后兼容的函数的更新。

4. StateMachine(workflow)的定义和实现

StateMachine 仿照 AWS StateMachine 的定义格式，每个 function 和分支判断都被定义为一个 State，workflow 在 State 中不断传递，如果 State 时 function(对应 Task)则以上个 State 返回 response body 作为输入对该 function 发起 http trigger(复用了 function 的 http trigger 接口)，如果 State 是分支判断则使用上个 State 的 response body 进行判断并以该 response body 作为本 State 的输出，状态机以用户定义的`StartAt`状态开始，向各 State 的`NextState`传递（Task 有一个，Choice 的每个分支各对应一个），在定义为`EndState`的 State 结束并将 response body 返回给用户。与形式化理论中的状态机工作过程类似。

StateMachine 同样使用绑定在物理机浮动 IP 和 Port 上的 url 进行 http trigger，url 为 activerIP:activerPort/stateMachine/`stateMachineName`。

创建 StateMachine 的流程十分简单，ApiServer 只检查 StateMachine 中是否存在起始和终止状态、各属性为 Task 的 State 是否已经创建对应的 function，之后将 StateMachine 的定义文件存储在 etcd 中即可，在 http trigger 发生后，Activer 会根据 StateMachine 的 Name 取出 StateMachine 的详细信息并以此为地图处理 http trigger。

StateMachine 的删除同样也只需要删除 etcd 中存储的 StateMachine 定义即可。

5. Activer 的实现

Activer 由以下三个线程组成：

- function name list Watcher：负责 watch function name list，在其发生变动（增加和删除 function）时同步更新（增加和删除）内存中记录用于记录并发访问数的 map 的条目。
- 主服务线程：运行在 Activer 对应的物理端口上，接收和处理 function 的 stateMachine 的 http trigger，为提高并发性，每个 http trigger 的处理都创建新的线程来处理后返回。
- 扩缩容线程：每 30s 唤醒一次，执行 function 对于的 Pod Replica 的动态扩容缩容，扩容缩容策略已述。

十四. GPU 功能实现

Mini-kubernetes 实现 GPU 任务上传和运行功能的方式为 Job，创建一个 GPU JOB 的流程如下：

- 用户通过 Kubectl 发送 create 命令并提供 GPU JOB 定义文件，文件中包括 GPU JOB 的名字，.cu 源代码的位置，Makefile 的位置，slurm 相关的参数（如需要 CPU 和 GPU 的数目、提交到的集群的名称）以及输出结果的目录（为了避免 Kubectl 由于执行 GPU JOB 长时间阻塞，ApiServer 在创建对应的 Pod 后立刻返回，用户可以使用 GPU JOB 的 get 接口查询 GPU JOB 的运行情况，如是否完成和是否出错，任务完成后，ApiServer 会在用户定义的输出结果目录下创建 result.out 和 error.err 文件分别记录 GPU JOB 运行的结果和报错）
- ApiService 的 Image Factory 根据定义文件生成该任务对应的 Docker Image，具体流程为：
  （1） 拉取和运行 Docker 基础镜像：gpu_job_uploader
  （2） 根据 yaml 定义生成 slurm 文件，将其使用 docker copy 接口写入 docker 容器中的预定位置，读取用户指定的源代码(.cu)文件和 Makefile 文件，使用 docker copy 接口写入到 docker 容器中的预定位置，将 yaml 的任务的名字、master 用于接收 GPU JOB 运行结束和运行结果的 http 请求的 url 写入 docker 容器的预定义位置
  （3） 使用 docker commit 和 docker push 上传修改后的容器，新镜像命名为 GPU JOB 的 name+UUID，之后停止和移除该 docker 容器
- ApiServer 将 GPU JOB 的定义数据和状态（尚未运行和拉取镜像阶段为 PENDING，正在运行阶段为 RUNNING，运行成功并结束状态为 SUCCESS，运行失败状态为 FAILED，由于网络故障失联状态为 UNKNOWN）存进 etcd 中。
- 在创建好 GPU JOB 对应的 Docker Image 后， Apiserver 生成可以运行该 docker 容器的 Pod 定义文件，并复用 Pod 的 create 接口在集群中运行之。

之后进入容器运行阶段，使用容器基础镜像内预置的 sh 脚本进行，具体流程如下：

- 容器使用 scp 指令将容器中的.cu 源代码文件、Makefile 文件、slurm 文件上传到 slurm 对应集群的任务上传节点，为了便于多个 GPU 任务同时执行，不同任务的目录名不同，目录名具体为：GPU JOB 的名字+UUID(GPU JOB 的名字在制作镜像时已经写在容器中的预定位置)，这保证了集群中 GPU JOB 可以并行上传和运行
- 上传结束后，容器使用 SSH 命令（容器内置了 SSH 无密码登录的环境）运行 sbatch

命令向 slurm 调度系统提交 GPU 任务

- 之后容器通过轮询的方式判断 GPU 任务是否执行结束，容器每 5s 通过 SSH 向该 GPU 任务对应的目录发送 ls 命令获取结果，如果 ls 命令获取的结果中包含 result.out 和 error.err 文件代表任务已经执行完毕（这两个文件记录了 GPU 任务执行的输出和报错，任务上传时目录下不包含这两个文件，slurm 只在任务结束后将结果写入该文件），多次测试发现这样判断任务是否结束最为准确，使用 squeue 或 sacct 等命令获取的数据有可能因为不同步而造成误判。
- 发现命令结束完毕后使用 scp 命令将结果文件(result.out 和 error.err)下载到 docker 容器中
- 使用 http 请求向 Master 接收 GPU JOB 完成信号的 url(url 在制作镜像时已经写在容器中的预定位置)，该信号为 http POST 请求，其中包含了 result.out 和 error.err 中读取的内容

此时 Pod 的任务已经完成，接下来 ApiServer 收到任务完成信号后执行后续工作:

- Apiserver 将结果和报错分别输出到用户定义的结果输出目录的 result.out 和 error.err 文件中，并将 GPU JOB 的状态标记为 SUCCESS，此时用户可以查看 GPU JOB 的运行结果
- ApiServer 之后复用 Pod 的删除接口删除执行 GPU JOB 任务的 Pod

GPU JOB 的删除流程为若 GPU JOB 对应的 Pod 没有运行结束被自动删除则先删除该 Pod，之后删除 etcd 中 GPU JOB 对应的数据.

GPU JOB 的实现通过不同 GPU JOB 单独在一个 Pod 中运行和不同 GPU JOB 上传到服务器的不同目录下实现了 GPU JOB 的并行运行，通过 Pod 主动向 ApiServer 发送任务完成信号实现了 Job 的主动停止和结果的传递。

十五. 容错和控制面重启

为方便控制面重启，Mini-Kubernetes 中的每个组件都实现为无状态，具体而言:

对于 controller、scheduler、ApiServer 而言，其作用为根据 etcd 中一些持久化数据的变化（Controller、Scheduler）或 http 指令（ApiServer）对其他数据做修改，即传递和同步 etcd 中存储的状态的变化。如 Deployment Controller 通过监视 etcd 中 Deployment 的各 Pod replica 的运行状态，根据运行状态决定删除故障 replica（在 etcd 中的 all Pod replica list 中删除该 replica 的 ID）和添加新的 replica（创建 Pod replica 将其信息存储进 etcd，并在 all Pod replica list 中加入该 replica 的 ID）；Scheduler 通过监视 all Pod replica list 的变化将 Pod replica 调度到对应 node（在 etcd 中该 node 对应的 per node pod replica ID list 中加入该 pod replica 的 ID）或删除 all Pod replica list 中删除的 Pod replica（在 etcd 中从 node 对应的 per node pod replica ID list 中删除该 pod replica 的 ID）。这类组件只要做好启动时的初始化工作（初始化的目的为确定已经完成的任务，相当于建立组件的本地缓存，从而得到尚待完成的任务，如 scheduler 初始化时需要查询各 node 上已经调度的 Pod replica 从而获得已经完成调度的 pod replica ID list），和运行中对集群状态改写的持久化工作（及时存储进 etcd）即可实现控制面重启。

对于 Kubelet 等组件而言，其作用为将 etcd 中的状态更新同步到物理机的应用层（如创建和删除 Pod），其启动时需要做的初始化工作为确定已经完成的同步工作，如对于 kubelet 而言，通过读取 etcd 中的信息（具体为所在 node 的 per node pod replica ID list 和 list 中各 Pod replica 的运行状态）可以得到所在集群中现有的 Pod 及其运行情况，使用初始化后的 Pod replica ID list 和 etcd 中对应的 per node pod replica ID list 对比可以得

知需要创建和删除的 Pod replica 有哪些。

对于 Kubectl、kube-proxy 等组件而言，其作用为流式处理发送对应监听端口/shell 的请求，如 Kubectl 解析 shell 命令并调用 ApiServer 接口，kube-proxy 根据 ApiServer 的 http 命令改写本地 shell 接口，这些组件可以不初始化即可运行。

总之，控制面重启通过及时持久化对集群状态的更新和在组件启动时初始化以探测集群状态来实现。

十六. 分工和贡献概览

具体分工请见各次迭代计划

组员	分工	贡献度/%
***	***	***
***	***	***
***	***	***

十七. 附录一：kubectl 交互格式和报错处理

1. Kubectl 提供的命令接口及其功能

1.1 Pod

- kubectl create -f xxx.yaml
  根据配置文件创建 pod 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe pod xxx.yaml/yyy
  查看 pod 信息，可以通过 pod 的名称或 pod 配置文件来确定 pod。
- kubectl get pods
  查看所有 pod 的简要信息，具体格式如下：

```
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
service-1   ClusterIP   10.119.11.111   <none>        8080/TCP    35.358121ms
service-2   NodePort    10.119.11.140   <none>        8080/TCP    2562047h47m16.854775807s
service-2   NodePort    10.119.11.140   <none>        7070/TCP    2562047h47m16.854775807s
```

- kubectl delete pod yyy
  根据 pod 名称删除对应 pod。

1.2 Service（包括 ClusterIP 和 Nodeport 两种类型）

- kubectl create -f xxx.yaml
  根据配置文件创建 service 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe service xxx.yaml/yyy
  查看 service 信息，可以通过 service 的名称或 service 配置文件来确定 service。
- kubectl get services
  查看所有 service 的简要信息，具体格式如下：

```
NAME        READY     STATUS    RESTARTS   AGE
test-pod-1  0/1       0         0          2562047h47m16.854775807s
test-pod-2  0/1       0         0          2562047h47m16.854775807s
```

- kubectl delete service yyy
  根据 service 名称删除对应 service。

1.3 DNS

- kubectl create -f xxx.yaml
  根据配置文件创建 dns 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe dns xxx.yaml/yyy
  查看 dns 信息，可以通过 dns 的名称或 dns 配置文件来确定 dns。
- kubectl delete dns xxx.yaml/yyy
  删除某 DNS 任务
- kubectl get dns
  查看所有 dns 的简要信息，具体格式如下：

```
All dns' information is as follows
NAME            HOST            PATH            SERVICE-NAME    PORT
test-dns        example.com     /               service-1       80
test-dns        example.com     /temp           service-2       80
```

1.4 Deployment

- kubectl create -f xxx.yaml
  根据配置文件创建 deployment 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe deployment xxx.yaml/yyy
  查看 deployment 信息，可以通过名称或配置文件来确定 deployment。
- kubectl get deployment
  查看所有 deployment 的简要信息，具体格式如下：

```
All deployments' information is as follows
NAME            READY           UpToDate   AVAILABLE  AGE
deployment-hjk  0/3             0          0          84.946946ms
```

- kubectl delete deployment yyy
  根据 deployment 名称删除对应 deployment。

1.5 Autoscaler

- kubectl create -f xxx.yaml
  根据配置文件创建 autoscaler 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe autoscaler xxx.yaml/yyy
  查看 autoscaler 信息，可以通过名称或配置文件来确定 autoscaler。
- kubectl get autoscaler
  查看所有 autoscaler 的简要信息，具体格式如下：

```
All autoscalers' information is as follows
NAME            MIN-PODS   MAX-PODS   REPLICAS   AGE
mynginx         2          10         0          89.745656ms
```

- kubectl delete autoscaler yyy
  根据 autoscaler 名称删除对应 autoscaler。

1.6 GPUJob

- kubectl create -f xxx.yaml
  根据配置文件创建 gpujob 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe gpujob xxx.yaml/yyy
  查看 gpujob 信息，可以通过名称或配置文件来确定 gpujob。
- kubectl get gpujob

查看所有 gpujob 的简要信息，字段包括:
NAME POD-NODE POD-STATUS POD-ID POD-STIME

1.7 Function

- kubectl create -f xxx.yaml
  根据配置文件创建 function 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe function xxx.yaml/yyy
  查看 function 信息，可以通过名称或配置文件来确定 function。
- kubectl get function
  查看所有 function 的简要信息，字段包括:
  NAME VERSION URL

- kubectl update soft -f xxx.yaml
  根据 function 配置文件软更新 function。
- kubectl update hard -f xxx.yaml
  根据 function 配置文件硬更新 function。
- kubectl delete function yyy
  根据 function 名称删除对应 function。

1.8 Statemachine

- kubectl create -f xxx.yaml
  根据配置文件创建 statemachine 资源，xxx.yaml 为 yaml 文件位置。
- kubectl describe statemachine yyy
  查看 statemachine 信息，可以通过名称来确定 statemachine。
- kubectl get statemachine
  查看所有 statemachine 的简要信息，字段包括:
  NAME STARTAT URL

- kubectl delete statemachine yyy
  根据 statemachine 名称删除对应 statemachine。

1.9 node

- kubectl get node
  查看所有 node 的简要信息，字段包括:
  NAME IP STATUS ID
- kubectl delete node yyy
  根据 node 名称删除对应 node。

1.10 其它

- kubectl
  显示所有可用的 kubectl 命令。

```
NAME:
   kubectl - Command line tool to communicate with apiserver.

USAGE:
   kubectl_test.test [global options] command [command options] [arguments...]

VERSION:
   0.0.0

COMMANDS:
   hello     Test
   create    Create resources according to xxx.yaml
   get       Get resources state
   delete    Delete resources according to xxx.yaml or name
   describe  Describe resources according name
   help, h   Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --help, -h     show help
   --version, -v  print the version
```

2. Kubectl 的异常处理和报错

2.1 未找到文件或地址

```
>kubectl create -f /home/test.yaml
2022/06/01 01:00:00 open /home/test.yaml: no such file or directory
```

2.2 命令行参数错误或缺失

```
>kubectl create dfs
2022/06/01 00:59:25 You need to specify directory
```

2.3 yaml 文件格式错误

```
>kubectl create -f ../test/yaml_test/test.yaml
2022/06/01 01:14:45 yaml: unmarshal errors:
  line 6: cannot unmarshal !!str `obeject` into def.NodeSelector
  line 21: cannot unmarshal !!str `int` into uint16
  line 22: cannot unmarshal !!str `int` into uint16
```

2.4 API server 返回值为空

例如输入 kubectl get pods，而 API server 返回值为空，输出"No pod exists"。

2.5 API server 返回状态错误

直接输出 API server 返回的错误信息

十八. 附录二：function 源代码环境支持和代码规范

1. function 源代码中应该有一个定义为 handler(env)的函数，Mini-Kubelet 通过调用该函数处理请求完成函数服务
2. handler 函数的 env 为函数运行的参数环境，具体而言：
   - env.url 为请求 url 的解析结果，可以使用 env.url.get('参数名'，默认值)的方法获取请求 url 中的参数
   - env.body 为请求体的解析结果可以使用 env.body['字段名']的方法获取请求体中的字

段

3. 对于 handler 中返回的结果，基础执行环境会将其转为 json 格式作为响应返回。

十九. 附录三：pipeline 流水线 CICD 脚本

```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                dir("/home/cicd/mini-kubernetes"){
git credentialsId: '*', url:'https://gitee.com/jkhe/mini-kubernetes.git'
                }
            }
        }
        stage('Build') {
            steps {
                dir("/home/cicd/mini-kubernetes/"){
                    sh 'make master'
                    sh 'make node'
                }
            }
        }
        stage('Deploy') {
            steps {
                dir("/home/cicd/mini-kubernetes/"){
                    sh 'make run_master'
                    sh 'make run_node'
                }
            }
        }
        stage('Test') {
            steps {
                dir("/home/cicd/mini-kubernetes/"){
                    sh 'make test'
                }
            }
        }
    }
}
```

二十. 附录四：ETCD 中的存储结构

```
.etcd____//meta data:
     |____{pod meta}(per pod)
     |____{pod instance meta}(per pod replica)
     |____{service meta}(per service)
     |____{function meta}(per function)
     |____{stateMachine meta}(per stateMachine)
     |____{deployment meta}(per deployment)
     |____{autoscaler meta}(per autoscaler)
     |____{gpuJob meta}(per gpuJob)
     |____{node meta}(per node)
     |____{dns meta}(per dns)
     |____{node resource record}
     |     (per node, record node's available CPU & memory)
     |____{podInstance resource record}
     |     (per pod, record node's CPU & memory usage)
     |// list:
     |____node ID list(all node's ID)
     |____{pod repilca ID list}
     |     (per pod, record the pod's replicas' ID)
     |____function name list(all function's name)
     |____deployment name list(all deployment's name)
     |____horizontal pod autoscaler name list(all autoscaler's name)
     |____{node pod replica ID list}
     |     (per node, record the ID of pod replicas running on the node)
     |____pod replica ID list(all pod replica's ID)
```