

# 目录

正文第一部分 实验内容.....	2
一、实验主要目标.....	2
二、实验环境及相关配置.....	2
2.1 ECS 弹性云服务器各项参数 .....	2
2.2 环境版本信息.....	2
2.3 集群网络信息.....	3
三、实验步骤.....	3
3.1 Kubernetes 搭建过程.....	3
3.2 OpenFaas 平台搭建.....	11
3.3 函数部署示例.....	12
3.4 K8s & OpenFaaS 部署过程中所遇到的 bug 及解决方法.....	15
3.5 监控平台搭建.....	21
3.6 测试函数编写 & 参数修改方法.....	31
3.7 性能测试方法.....	34
四、实验结果分析.....	35
4.1 更改参数规模.....	35
4.2 修改线程数.....	36
4.3 资源限制实验（使用计算密集型 float-operation 函数） .....	37
正文第二部分 知识点分析.....	46
五、知识点分析.....	46
5.1 Kubernetes 架构、Master Node & Worker Node .....	46
5.2 K8s 基础概念：Pod & Service & Controller.....	49
5.3 OpenFaaS 层次框架 & 工作流程 .....	50
5.4 OpenFaaS 函数请求处理流程 & 源码解读 .....	53
5.5 Prometheus & PromQL.....	57
5.6 OpenFaaS 自动伸缩原理 & 源码解读 .....	58
5.7 K8s 集群资源限制 .....	61
5.8 K8s 任务调度 .....	62
5.9 压力测试.....	63
5.10 阿姆达尔定律.....	64
小组分工.....	65
心得体会与收获.....	66

# 正文第一部分 实验内容

## 一、实验主要目标

学习 K8s、OpenFaaS 的基础概念与原理。利用华为云虚拟机构建三个及以上的节点搭建 K8s 集群和 OpenFaaS 平台；配置 Node Exporter + Prometheus + Grafana 集群监控平台；运用所学知识设计实验对系统进行测试，通过观察实验数据，分析系统性能并尝试找出瓶颈。体会 Serverless 架构与传统架构的不同，验证所学习的原理知识。

## 二、实验环境及相关配置

### 2.1 ECS 弹性云服务器各项参数

本实验使用 3 台配置相同的弹性云服务器 ECS：

计费模式	按需计费
区域	北京四
可用区	可用区 1
CPU 架构	x86
规格	2vCPU 4GiB 64bit
系统	CentOS 7.6
系统盘	100GiB
弹性公网 IP 设置	按流量计费
宽带	10 Mbit/s

表 2-1-1

### 2.2 环境版本信息

Kubernetes 版本：三个节点均为 1.20.5
Docker-hub 版本：server 为 18.09.7 client 为 20.10.9
Faas-cli 版本： 0.13.10

## 2.3 集群网络信息

	公网 ip	内网 ip
Master-0001	119.3.164.247	192.168.0.213
Slave-0001	124.70.110.214	192.168.0.132
Slave-0002	124.70.25.183	192.168.0.194

表 2-3-1

安全组: full-Access, 即放通所有端口

子网网段: 192.168.0.0/24

## 三、实验步骤

### 3.1 Kubernetes 搭建过程

Master 和 slave 节点均需要安装 Kubernetes, 安装步骤相同, 均如下所示:

首先修改 hosts 文件, 添加各个节点的名称和内网 ip, 主节点修改后如下, 从节点类似:

```
1:1      localhost      localhost.localdomain  localhost6      localhost6.localdomain6
192.168.0.213  localhost      localhost.localdomain  localhost4      localhost4.localdomain4
127.0.0.1    master-0001    master-0001
192.168.0.213  master-0001
~
```

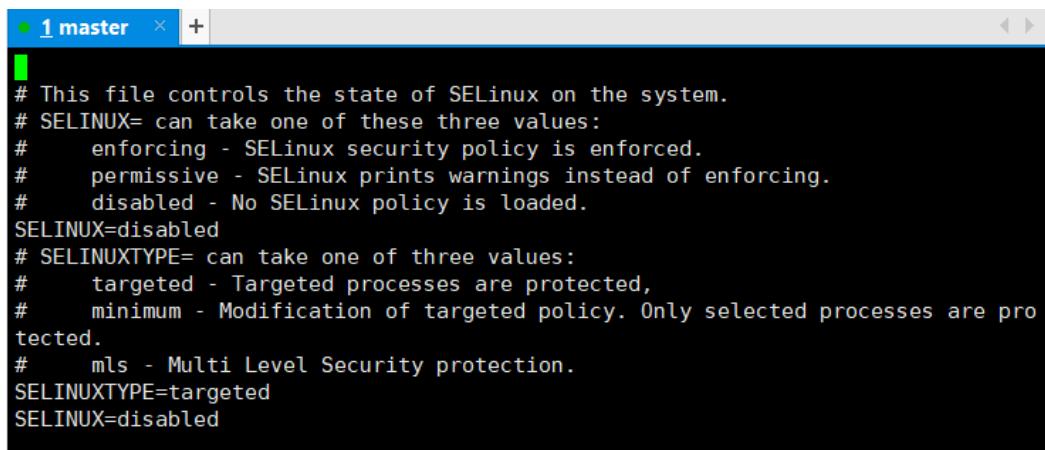
图 3-1-1

关闭防火墙:

```
[root@master-0001 ~]# systemctl stop firewalld
[root@master-0001 ~]# systemctl disable firewalld
[root@master-0001 ~]#
```

图 3-1-2

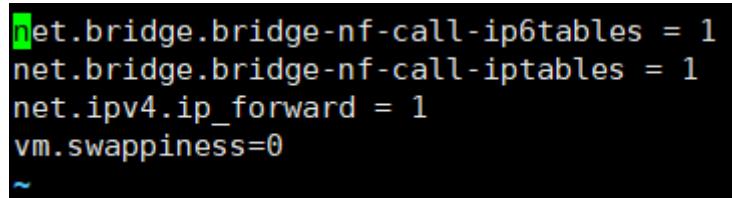
禁用 SELINUX:



```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - No SELinux policy is loaded.
SELINUX=disabled
# SELINUXTYPE= can take one of three values:
#       targeted - Targeted processes are protected,
#       minimum - Modification of targeted policy. Only selected processes are protected.
#       mls - Multi Level Security protection.
SELINUXTYPE=targeted
SELINUX=disabled
```

图 3-1-3

在/etc/sysctl.d 目录下创建 k8s.conf 文件，并添加如下内容：



```
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
vm.swappiness=0
~
```

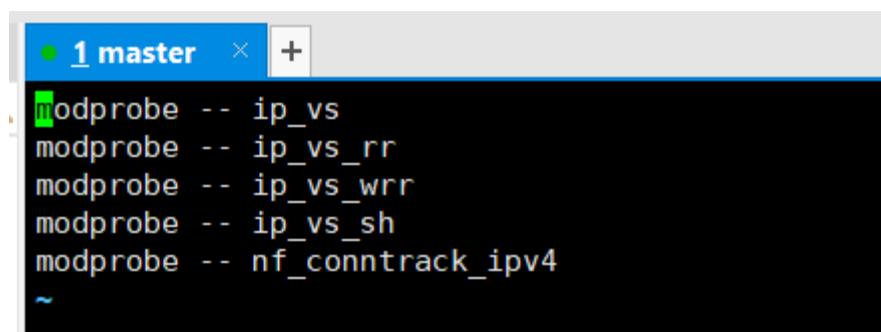
图 3-1-4

运行如下命令使其生效：

```
modprobe br_netfilter
sysctl -p /etc/sysctl.d/k8s.conf
```

然后安装 ipvs，这个会在 kube-proxy 中被使用：

在创造/etc/sysconfig/modules 中 ipvs.modules 文件，并添加如下内容：



```
modprobe -- ip_vs
modprobe -- ip_vs_rr
modprobe -- ip_vs_wrr
modprobe -- ip_vs_sh
modprobe -- nf_conntrack_ipv4
~
```

图 3-1-5

使上述配置文件生效

```
[root@master-0001 modules]# chmod 755 /etc/sysconfig/modules/ipvs.modules
[root@master-0001 modules]# bash /etc/sysconfig/modules/ipvs.modules
[root@master-0001 modules]# lsmod | grep -e ip_vs -e nf_conntrack_ipv4
nf_conntrack_ipv4      15053  0
nf_defrag_ipv4        12729  1 nf_conntrack_ipv4
ip_vs_sh              12688  0
ip_vs_wrr              12697  0
ip_vs_rr              12600  0
ip_vs                  145458  6 ip_vs_rr,ip_vs_sh,ip_vs_wrr
nf_conntrack           139264  2 ip_vs,nf_conntrack_ipv4
libcrc32c              12644  2 ip_vs,nf_conntrack
[root@master-0001 modules]#
```

图 3-1-6

然后安装 ipvs 用到的 ipset 和 ipvsadm:

```
[root@master-0001 modules]# yum install ipset
Loaded plugins: fastestmirror
Determining fastest mirrors
base                                         | 3.6 kB     00:00
epel                                         | 4.7 kB     00:00
extras                                         | 2.9 kB     00:00
updates                                         | 2.9 kB     00:00
(1/7): base/7/x86_64/group_gz               | 153 kB     00:00
(2/7): epel/x86_64/group_gz                 | 96 kB     00:00
(3/7): epel/x86_64/updateinfo               | 1.0 MB     00:00
(4/7): extras/7/x86_64/primary_db           | 232 kB     00:00
(5/7): updates/7/x86_64/primary_db          | 7.1 MB     00:09
(6/7): epel/x86_64/primary_db               | 6.9 MB     00:14
(7/7): base/7/x86_64/primary_db             | 6.1 MB     00:16
Package ipset-7.1-1.el7.x86_64 already installed and latest version
Nothing to do
```

图 3-1-7

```
[root@master-0001 modules]# yum install ipset
Loaded plugins: fastestmirror
Determining fastest mirrors
base                                         | 3.6 kB     00:00
epel                                         | 4.7 kB     00:00
extras                                         | 2.9 kB     00:00
updates                                         | 2.9 kB     00:00
(1/7): base/7/x86_64/group_gz               | 153 kB     00:00
(2/7): epel/x86_64/group_gz                 | 96 kB     00:00
(3/7): epel/x86_64/updateinfo               | 1.0 MB     00:00
(4/7): extras/7/x86_64/primary_db           | 232 kB     00:00
(5/7): updates/7/x86_64/primary_db          | 7.1 MB     00:09
(6/7): epel/x86_64/primary_db               | 6.9 MB     00:14
(7/7): base/7/x86_64/primary_db             | 6.1 MB     00:16
Package ipset-7.1-1.el7.x86_64 already installed and latest version
Nothing to do
```

图 3-1-8

再需要安装 docker

```
[root@master-0001 modules]# yum-config-manager --add-repo \
>      https://download.docker.com/linux/centos/docker-ce.repo
Loaded plugins: fastestmirror
adding repo from: https://download.docker.com/linux/centos/docker-ce.repo
grabbing file https://download.docker.com/linux/centos/docker-ce.repo to /etc/yum.repos.d/docker-ce.repo
repo saved to /etc/yum.repos.d/docker-ce.repo
[root@master-0001 modules]#
```

图 3-1-9

查看最新 docker 版本:

```
[root@master-0001 home]# yum list docker-ce.x86_64 --showduplicates |sort -r
Loaded plugins: fastestmirror
Installed Packages
docker-ce.x86_64           3:20.10.6-3.el7          docker-ce-stable
docker-ce.x86_64           3:20.10.5-3.el7          docker-ce-stable
docker-ce.x86_64           3:20.10.4-3.el7          docker-ce-stable
docker-ce.x86_64           3:20.10.3-3.el7          docker-ce-stable
docker-ce.x86_64           3:20.10.2-3.el7          docker-ce-stable
docker-ce.x86_64           3:20.10.1-3.el7          docker-ce-stable
docker-ce.x86_64           3:20.10.0-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.9-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.8-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.7-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.6-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.5-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.4-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.3-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.2-3.el7          docker-ce-stable
docker-ce.x86_64           3:19.03.15-3.el7         docker-ce-stable
docker-ce.x86_64           3:19.03.14-3.el7         docker-ce-stable
docker-ce.x86_64           3:19.03.1-3.el7          docker-ce-stable
docker-ce.x86_64           3:18.09.7-3.el7          docker-ce-stable
```

图 3-1-10

安装对应版本并启动

```
Installed:
  docker-ce.x86_64 3:18.09.7-3.el7

Dependency Installed:
  audit-libs-python.x86_64 0:2.8.5-4.el7
  checkpolicy.x86_64 0:2.5-8.el7
  container-selinux.noarch 2:2.119.2-1.911c772.el7_8
  containerd.io.x86_64 0:1.4.4-3.1.el7
  docker-ce-cli.x86_64 1:20.10.6-3.el7
  docker-scan-plugin.x86_64 0:0.7.0-3.el7
  libcgroup.x86_64 0:0.41-21.el7
  libsemanage-python.x86_64 0:2.5-14.el7
  policycoreutils-python.x86_64 0:2.5-34.el7
  python-IPy.noarch 0:0.75-6.el7
  setools-libs.x86_64 0:3.3.8-4.el7

Complete!
[root@master-0001 modules]#
```

图 3-1-11

然后修改 docker cgroup driver 为 systemd:

```
[{"exec-opts": ["native.cgroupdriver=systemd"]}]
```

图 3-1-12

修改完成后重启 docker 并验证：

```
[root@master-0001 home]# docker info | grep Cgroup
Cgroup Driver: systemd
```

图 3-1-13

至此 docker 安装完成。

然后安装 kubernetes。在/etc/yum.repos.d 目录下添加 kubernetes.repo 文件，为镜像添加国内阿里云的源：

```
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg https://mirrors
.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
~
~
~
```

图 3-1-14

使用如下命令进行安装

```
yum makecache fast
```

```
yum install -y kubelet kubeadm kubectl
```

安装完成：

```
Installed:
  kubeadm.x86_64 0:1.21.0-0 kubelet.x86_64 0:1.21.0-0 kubectl.x86_64 0:1.21.0-0

Dependency Installed:
  conntrack-tools.x86_64 0:1.4.4-7.el7
  cri-tools.x86_64 0:1.13.0-0
  kubernetes-cni.x86_64 0:0.8.7-0
  libnetfilter_cthelper.x86_64 0:1.0.0-11.el7
  libnetfilter_cttimeout.x86_64 0:1.0.0-7.el7
  libnetfilter_queue.x86_64 0:1.0.2-2.el7_2
  socat.x86_64 0:1.7.3.2-2.el7

Complete!
[root@master-0001 modules]#
```

图 3-1-15

再关闭 swap 分区，在/etc/sysctl.d/k8s.conf 的最后一行添加如下内容：

```
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
vm.swappiness=0
```

图 3-1-16

使用下面命令应用配置：

sysctl -p /etc/sysctl.d/k8s.conf

然后启动 kubeadm 的服务 kubelet.service

```
[root@master-0001 modules]# systemctl enable kubelet.service
Created symlink from /etc/systemd/system/multi-user.target.wants/kubelet.service
to /usr/lib/systemd/system/kubelet.service.
[root@master-0001 modules]#
```

图 3-1-17

此时 kubeadm 已经安装完成，下面使用 kubeadm 安装 Kubernetes。

首先查看可用的版本：（安装时最新可用版本为 1.20.5）

```
[root@master-0001 modules]# kubeadm config images list
k8s.gcr.io/kube-apiserver:v1.21.0
k8s.gcr.io/kube-controller-manager:v1.21.0
k8s.gcr.io/kube-scheduler:v1.21.0
k8s.gcr.io/kube-proxy:v1.21.0
k8s.gcr.io/pause:3.4.1
k8s.gcr.io/etcd:3.4.13-0
k8s.gcr.io/coredns/coredns:v1.8.0
[root@master-0001 modules]#
```

图 3-1-18

然后创建 sh 脚本用于拉取镜像，脚本内容如下：

```
images=(
    kube-apiserver:v1.20.5
    kube-controller-manager:v1.20.5
    kube-scheduler:v1.20.5
    kube-proxy:v1.20.5
    pause:3.2
    etcd:3.4.13-0
    coredns:1.7.0
)
for imageName in ${images[@]}; do
    docker pull registry.cn-hangzhou.aliyuncs.com/google_containers/$imageName
    docker tag registry.cn-hangzhou.aliyuncs.com/google_containers/$imageName k8s.gcr.io/$imageName
done
```

图 3-1-19

然后使用下面命令进行镜像拉取：

```
bash /home/docker_pull_kube.sh
```

拉取完成后新建一个配置文件 kubeadm.yaml，写入如下内容：

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 192.168.0.213
  bindPort: 6443
nodeRegistration:
  taints:
    - effect: PreferNoSchedule
      key: node-role.kubernetes.io/master
---
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
kubernetesVersion: v1.20.5
networking:
  podSubnet: 10.244.0.0/16
~
```

图 3-1-20

(集群中从节点至此已经搭建完成)

最后使用下面命令来初始化 kubernetes

```
kubeadm init --config kubeadm.yaml --ignore-preflight-errors=Swap
```

初始化完成后安装 flannel 插件：

```
[root@master-0001 home]# curl -O https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
% Total    % Received % Xferd  Average Speed   Time     Time   Current
          Dload  Upload Total Spent   Left Speed
100  4821  100  4821    0     0  6563      0 --:--:-- --:--:-- --:--:--  6568
[root@master-0001 home]#
```

图 3-1-21

```
[root@master-0001 home]# kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
podsecuritypolicy.policy/psp.flannel.unprivileged configured
clusterrole.rbac.authorization.k8s.io/flannel unchanged
clusterrolebinding.rbac.authorization.k8s.io/flannel unchanged
serviceaccount/flannel unchanged
configmap/kube-flannel-cfg unchanged
daemonset.apps/kube-flannel-ds configured
[root@master-0001 home]#
```

图 3-1-22

此时 kubernetes 已经搭建完成，在三个节点上分别进行上述操作（从节点不需要初始化集群和安装 flannel 插件）。

然后需要将从节点加入到主节点中，在主节点初始化完成后，在最后会有加入本节点的命令，如果命令失效，可以使用 `kubeadm token create --print-join-command` 指令重新生成加入密钥。

```
[root@master-0001 home]# kubeadm token create --print-join-command
kubeadm join 192.168.0.213:6443 --token bogr3z.858varbte5ev16n5 --discovery-token-ca-cert-hash sha256:245c272287adafa8de826d5d6e4c40e41836f2a03ab9fd6f0ab6a46250733076
[root@master-0001 home]#
```

图 3-1-23

在从节点命令行输入上述指令即可加入集群。

```
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm ku
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/ku
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

[root@slave-0001 home]#
```

图 3-1-24

加入集群后可以在主节点使用 kubectl 指令查看 pod 是否正常运行：

```
[root@master-0001 home]# kubectl get pod -n kube-system
NAME                               READY   STATUS    RESTARTS   AGE
coredns-74ff55c5b-2tp6r           1/1    Running   21          21d
coredns-74ff55c5b-6mg4q           1/1    Running   21          21d
etcd-master-0001                  1/1    Running   22          21d
kube-apiserver-master-0001        1/1    Running   24          21d
kube-controller-manager-master-0001 1/1    Running   23          21d
kube-flannel-ds-5xhxr             1/1    Running   25          21d
kube-flannel-ds-6nfnw              1/1    Running   20          21d
kube-flannel-ds-k565g              1/1    Running   9           12d
kube-proxy-gqjq9                  1/1    Running   9           12d
kube-proxy-m2xv5                  1/1    Running   19          21d
kube-proxy-nfjml                  1/1    Running   21          21d
kube-scheduler-master-0001        1/1    Running   22          21d
```

图 3-1-25

使用 kubectl get nodes 指令来查看集群已有节点：

```
[root@master-0001 home]# kubectl get nodes
NAME      STATUS   ROLES
master-0001 Ready    control-plane, master
slave-0001 Ready    <none>
slave-0002 Ready    <none>
[root@master-0001 home]#
```

图 3-1-26

至此，三节点集群已成功搭建，下面在这个 kubernetes 集群的基础上进行 openfaas 的搭建。

## 3.2 OpenFaas 平台搭建

首先从 github 上 clone faas-nets 文件，然后 cd 到对应目录中。  
执行下列指令，创建 openfaas 和 openfaas-fn 两个 namespace：

```
kubectl apply -f ./namespaces.yml
```

查看是否成功创建：

```
[root@master-0001 home]# kubectl get namespaces
NAME      STATUS  AGE
default   Active  21d
kube-node-lease  Active  21d
kube-public  Active  21d
kube-system  Active  21d
openfaas   Active  21d
openfaas-fn Active  21d
[root@master-0001 home]#
```

图 3-2-1

然后给 openfaas 创建用户和密码，这里用户和密码均为 admin：

```
[root@master-0001 home]# kubectl -n openfaas create secret generic basic-auth \
> --from-literal=basic-auth-user=admin \
> --from-literal=basic-auth-password=admin
```

图 3-2-2

然后使用 kubectl apply -f ./yaml/ 安装 openfaas 的所有组件：

```
[root@master-0001 faas-nets]# cd faas-nets
[root@master-0001 faas-nets]# kubectl apply -f ./yaml/
configmap/alertmanager-config created
deployment.apps/alertmanager created
service/alertmanager created
deployment.apps/basic-auth-plugin created
service/basic-auth-plugin created
serviceaccount/openfaas-controller created
role.rbac.authorization.k8s.io/openfaas-controller created
role.rbac.authorization.k8s.io/openfaas-profiles created
rolebinding.rbac.authorization.k8s.io/openfaas-controller created
rolebinding.rbac.authorization.k8s.io/openfaas-profiles created
Warning: apiextensions.k8s.io/v1beta1 CustomResourceDefinition is deprecated in v1.16+, unavailable in v1.22; use apiextensions.k8s.io/v1 CustomResourceDefinition
customresourcedefinition.apiextensions.k8s.io/profiles.openfaas.com created
deployment.apps/gateway created
service/gateway-external created
service/gateway created
deployment.apps/nats created
service/nats created
customresourcedefinition.apiextensions.k8s.io/profiles.openfaas.com configured
configmap/prometheus-config created
deployment.apps/prometheus created
serviceaccount/openfaas-prometheus created
role.rbac.authorization.k8s.io/openfaas-prometheus created
role.rbac.authorization.k8s.io/openfaas-prometheus-fn created
rolebinding.rbac.authorization.k8s.io/openfaas-prometheus created
rolebinding.rbac.authorization.k8s.io/openfaas-prometheus-fn created
service/prometheus created
deployment.apps/queue-worker created
[root@master-0001 faas-nets]#
```

图 3-2-3

验证 openfaas 下的 pod 是否都已安装成功：

```

[root@master-0001 faas-netes]# kubectl get pod -n openfaas
NAME                      READY   STATUS    RESTARTS   AGE
alertmanager-5c857b6674-jf598  1/1    Running   0          8m34s
basic-auth-plugin-85d885557c-g5ztl  1/1    Running   0          8m34s
gateway-6ffb6bd755-7xhnv        2/2    Running   0          8m34s
nats-5fdf6476f-lpvqx          1/1    Running   0          8m33s
prometheus-96fd58985-x87xq     1/1    Running   0          8m32s
queue-worker-7cbc9f8688-7phrt  1/1    Running   0          8m33s
[root@master-0001 faas-netes]#

```

图 3-2-4

此时 openfaas 已经实际搭建完成，为了正常部署函数和使用，需要再去下载 faas-cli 工具。

由于直接下载过慢，我们先将 faas-cli 下载到本地，再传到服务器中。然后将 faas-cli 复制一份到/usr/local/bin 目录下，便于使用。

### 3.3 函数部署示例

我们部署的函数采用从 dockerhub 拉取镜像的方式，所以需要先注册登录 dockerhub 账号：

```

[root@master-0001 home]# docker login -u willowd -p [REDACTED]
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[root@master-0001 home]#

```

图 3-3-1

然后创建一个 function 目录，用于存放函数。

在 function 中使用如下命令创建一个模板函数：

faas-cli new --lang python hello-python

```

[root@master-0001 functions]# faas-cli new --lang python hello-python
2021/04/17 20:09:25 No templates found in current directory.
2021/04/17 20:09:25 Attempting to expand templates from https://github.com/openfaas/templates.git
2021/04/17 20:09:33 Fetched 13 template(s) : [csharp dockerfile go java11 java11
-vert-x node node12 node14 php7 python python3 python3-debian ruby] from https://
github.com/openfaas/templates.git
Folder: hello-python created.

[REDACTED FOLDER STRUCTURE]

Function created in folder: hello-python
Stack file written: hello-python.yml
[root@master-0001 functions]#

```

图 3-3-2

修改 yaml 中的镜像地址：

```
version: 1.0
provider:
  name: openfaas
  gateway: http://192.168.0.213:31112
functions:
  hello-python:
    lang: python
    handler: ./hello-python
    image: willowd/hello-python:latest
```

图 3-3-3

然后分别使用下面两条命令生成和 push 镜像：

```
faas-cli build -f ./hello-python.yml
```

```
faas-cli push -f ./hello-python.yml
```

结果如下：

```
Step 26/31 : RUN chmod +x app/app.py & chmod +x 777 /home/app/python
--> Using cache
--> 77359b1ae8b7
Step 27/31 : USER app
--> Using cache
--> 098f210e7de5
Step 28/31 : ENV fprocess="python index.py"
--> Using cache
--> ad9b68780761
Step 29/31 : EXPOSE 8080
--> Using cache
--> 0bf9d7ee2bbf
Step 30/31 : HEALTHCHECK --interval=3s CMD [ -e /tmp/.lock ] || exit 1
--> Using cache
--> 4f5c80d9d89f
Step 31/31 : CMD ["fwatchdog"]
--> Using cache
--> 8503b6e11f01
Successfully built 8503b6e11f01
Successfully tagged willowd/hello-python:latest
Image: willowd/hello-python:latest built.
[0] < Building hello-python done in 0.18s.
[0] Worker done.

Total build time: 0.18s
[root@master-0001 functions]#
```

图 3-3-4

```
[root@master-0001 functions]# faas-cli push -f ./hello-python.yml
[0] > Pushing hello-python [willowd/hello-python:latest].
The push refers to repository [docker.io/willowd/hello-python]
b1cb4fb77170: Mounted from zqyoung/hello-python
4d244b3c1e16: Mounted from zqyoung/hello-python
ef2897930bca: Mounted from zqyoung/hello-python
e1a1cb97f42c: Mounted from zqyoung/hello-python
e4276c6f66fa: Pushed
d50e1548636d: Mounted from zqyoung/hello-python
da9368b40da8: Mounted from zqyoung/hello-python
5270ab2a4768: Mounted from zqyoung/hello-python
c2e1497002a9: Mounted from zqyoung/hello-python
b3c3e5ed3fa7: Mounted from zqyoung/hello-python
98876204d0b2: Mounted from zqyoung/hello-python
32ec5d125ccb: Mounted from zqyoung/hello-python
62b2c6bcafbc: Mounted from zqyoung/hello-python
879c0d8666e3: Layer already exists
20a7b70bd12f: Layer already exists
3fc750b41be7: Layer already exists
beee9f30bc1f: Layer already exists
latest: digest: sha256:1b33db70b9d29f8e059bc7bb798000ff5e2e517be8594d52c94dd28508c3a227 size: 4074
[0] < Pushing hello-python [willowd/hello-python:latest] done.
[0] Worker done.
```

图 3-3-5

将函数部署之前需要使用 faas-cli 登录 openfaas 平台，在登录之前需要先设置网关。使用如下命令设置网关为 master 的私网 ip:

```
[root@master-0001 functions]# export OPENFAAS_URL=http://192.168.0.213:31112
[root@master-0001 functions]#
```

图 3-3-6

然后使用 faas-cli login -u admin -p admin 登录:

```
[root@master-0001 functions]# faas-cli login -u admin -p admin
WARNING! Using --password is insecure, consider using: cat ~/faas_pass.txt | faas-cli login -u user --password-stdin
Calling the OpenFaaS server to validate the credentials...
WARNING! You are not using an encrypted connection to the gateway, consider using HTTPS.
credentials saved for admin http://192.168.0.213:31112
[root@master-0001 functions]#
```

图 3-3-7

登录成功后可以使用 kubectl deploy -f ./hello-python.yml 进行部署，部署成功后可以在网页中看到这个函数。

网页的登录如下，使用公网 ip+31112 端口登录:

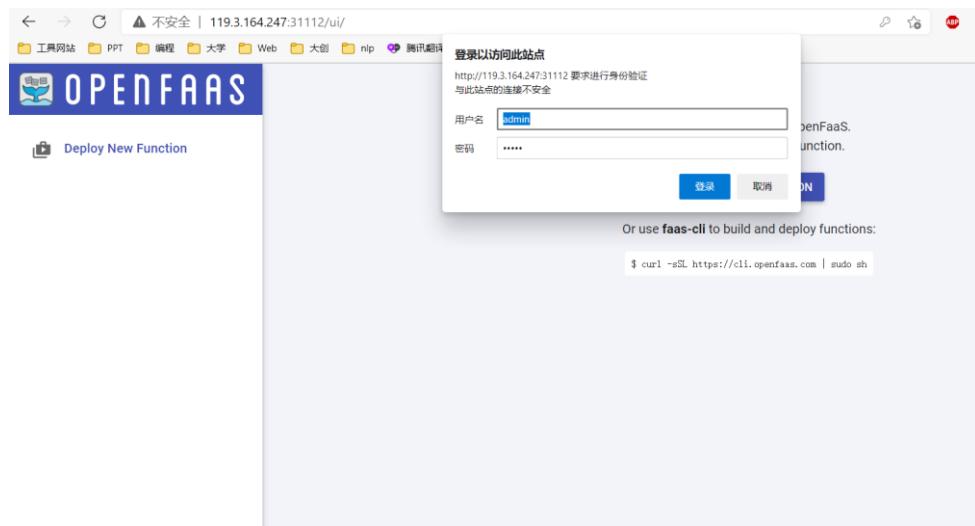


图 3-3-8

可以看到已经存在了 hello-python 函数

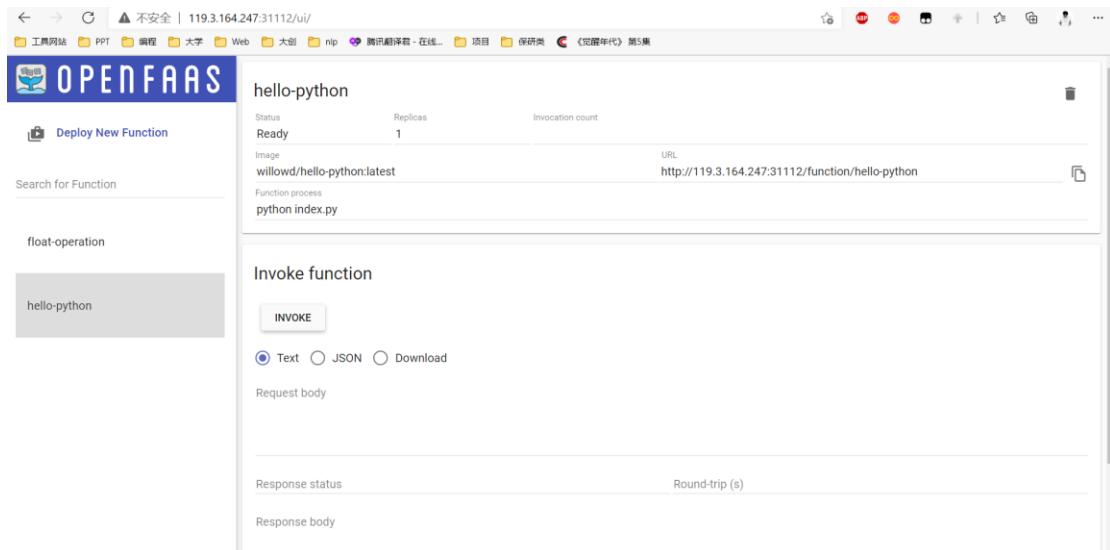


图 3-3-9

同时，也可以直接在网页端使用 ui 来创建函数：

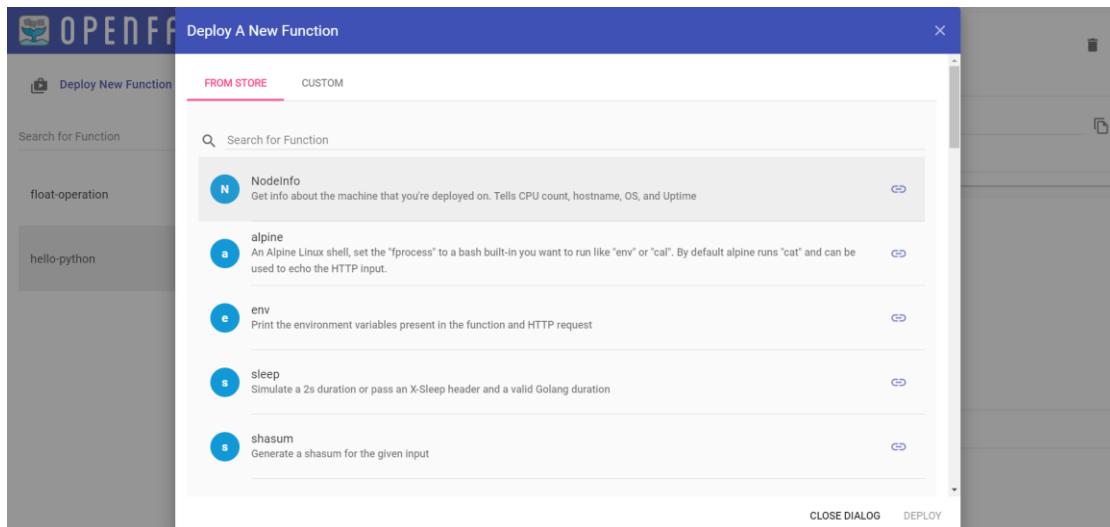


图 3-3-10

## 3.4 K8s & OpenFaaS 部署过程中所遇到的 bug 及解决方法

### 3.4.1 安装 docker

安装 docker 时不可以直接使用 `yum install docker`，否则会在后续 openfaas 下载函数模板时遇到错误。应该按照上述流程，查看 docker 的最新可用版本，使用上述命令安装对应最新版本。

### 3.4.2 拉取 kubernetes 镜像

拉取 kubernetes 镜像时，需要先查看最新可用版本；将 docker\_pull\_buke.sh 中的版本号修改为对应的版本，修改时需要将 kube-apiserver, kubescheduler, kube-controller-manager 等镜像的版本都修改为同样的最新版本号。

### 3.4.3 flannel 插件安装

安装 flannel 插件时需要修改镜像源，将镜像源换为国内的阿里源，否则无法正常下载安装。

### 3.4.4 pod 一直处于创建状态

部署 openfaas 时，一开始 pod 长时间处于 ContainerCreating 状态，无法创建成功。借助 describe 语句查看具体的镜像状态，报错信息如下：

冬 3-4-1

报错信息显示：从节点的 cni0 网卡错误，将这个网卡删除后重新构建，构建完成后所有 pod 均可正常创建。

### 3.4.5 faas-cli 登录问题

使用 faas-cli 登录前需要设置 OPENFAAS\_URL，如果不设置的话默认网关为 127.0.0.1:8080，faas-cli 不能正确登录和部署函数。设置时可以将网关设置为 127.0.0.1:31112 或者私网 ip+31112 端口，并且这个网关需要与函数 yaml 配置文件中的网关相同，否则也不能正确部署。

```

version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:31112
functions:
  float-operation:
    lang: python3
    handler: ./float-operation
    image: willowd/float-operation

```

图 3-4-2

### 3.4.6 跨用户节点加入集群

在不同用户的 ECS 服务器在安装好 kubeadm 之后，从节点不能加入主节点所在的集群。调试过程中发现，从节点和主节点之间不能 ping 通，判断是网络问题。

借助华为云官方文档，我们了解到不同账户的服务器内网默认是不互通的，需要进行配置。并且由于我们的两个账号在创建服务器时选择了北京四大区的不同可用区，所以需要通过建立“VPC 对等连接”来实现服务器内网的互通。

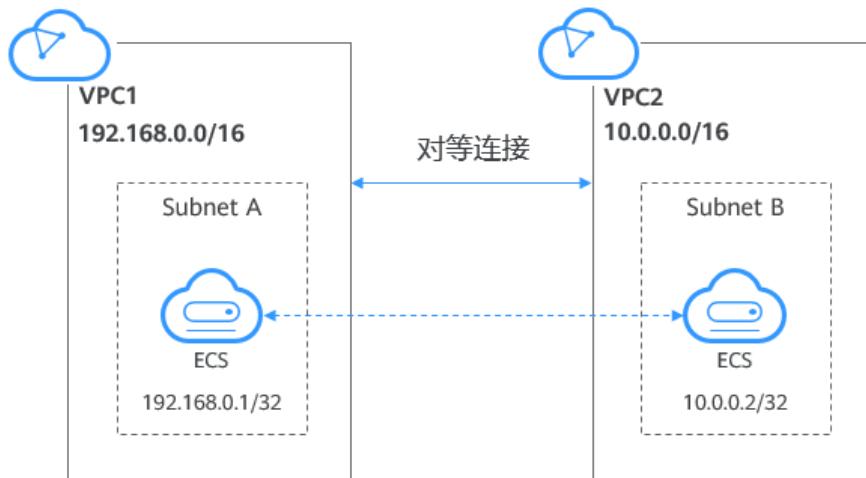


图 3-4-3

建立对等连接之后，可以使用私有 ip 地址在两个 VPC 之间进行通信，就像两个 VPC 在同一个网络中一样，但是不同区域的 VPC 之间不能创立对等连接。并且，有重叠子网网段的 VPC 建立的对等连接可能是无效的，我们在实际配置时也发现了不同账号的两台服务器，在网段都是 192.168.0.0/16 时，不能建立有效地对等连接。可用的对等连接建立步骤如下：

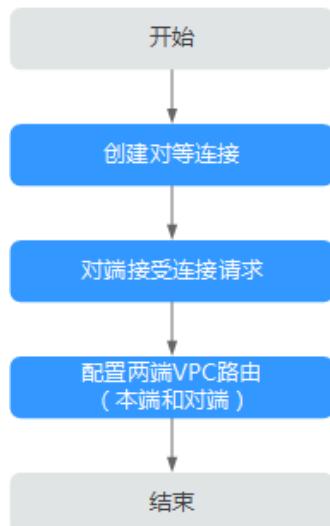


图 3-4-4

登录从节点账户的华为云，按照下图所示依次找到创建对等连接的按钮：

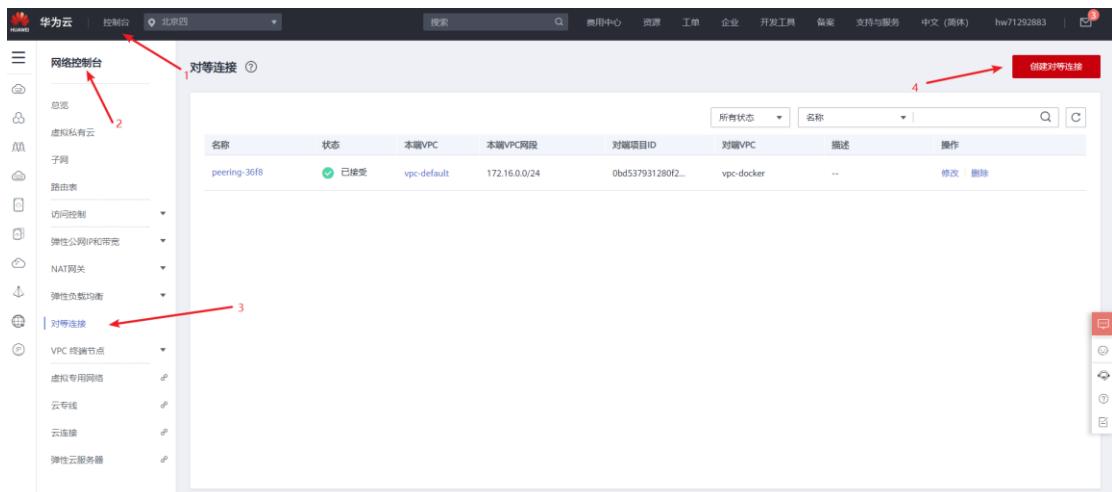


图 3-4-5

然后选择“其他账户”：

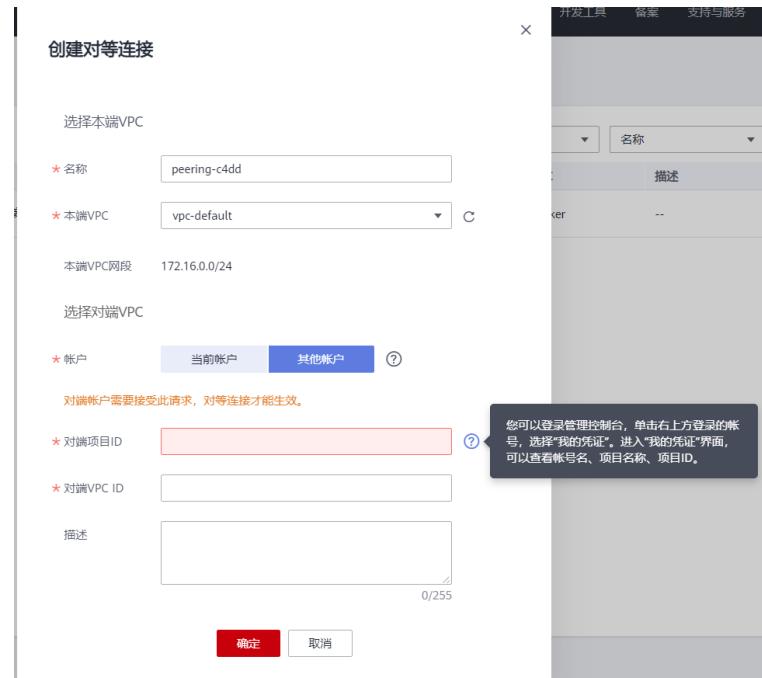


图 3-4-6

对端项目 ID 和对端 VPC ID 按照提示，需要登录主节点账号的华为云进行查看。在正确填写信息后点击确定，这时已经向主节点账号发送了对等连接请求。然后登录主节点账号，进入控制台的对等连接窗口（和上图步骤相同），查看待接受的对等连接请求，点击接受。对等连接接受之后，双方还需要配置对等连接的路由。具体步骤如下：

在上述对等连接窗口，点击已接受的对等连接项目：



图 3-4-7

然后点击路由表：



图 3-4-8

按照下图步骤依次点击和添加信息：

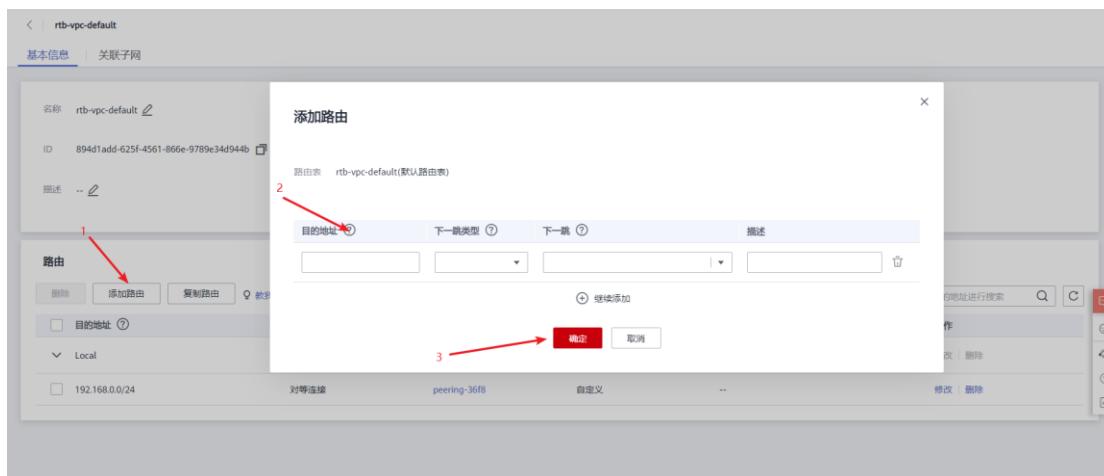


图 3-4-9

下图便是一个已经添加好的路由信息



图 3-4-10

在添加路由时，如果两个子网有重叠，则不能正常添加：

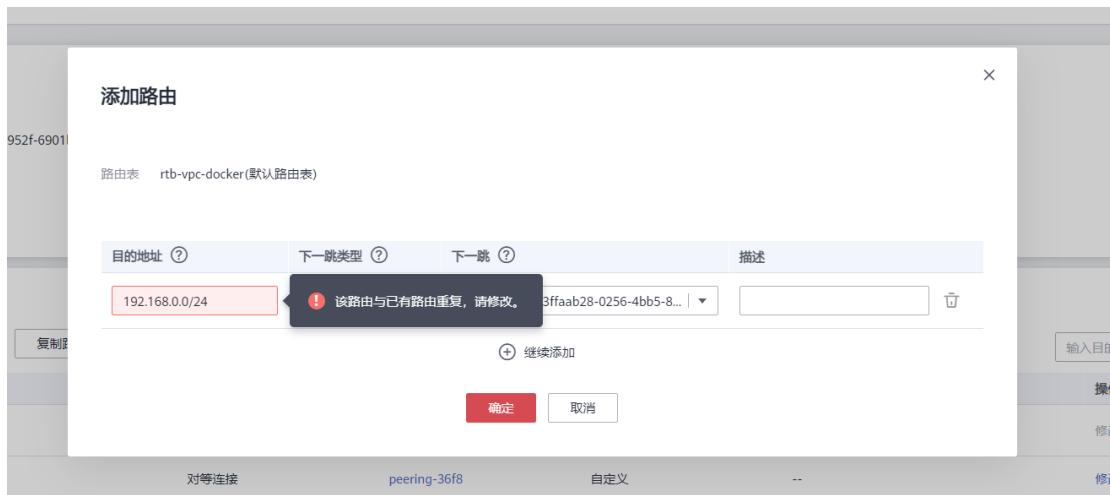


图 3-4-11

在主从节点账号均正确完成了上述路由的配置之后，两台服务器便可互相 ping 通，从节点也可以通过密钥指令来加入集群。

## 3.5 监控平台搭建

### 3.5.1 Prometheus 搭建

由于 openfaas 在搭建过程中已经进行了 Prometheus 的配置和搭建，所以并不需要从头开始在 k8s 上建立 Prometheus 监控系统。如下图所示，完成 openfaas 搭建后，相关组件中已经包含了 Prometheus。

```
[root@master-0001 faas-netes]# kubectl get pod -n openfaas
  NAME           READY   STATUS    RESTARTS   AGE
  alertmanager-5c857b6674-jf598   1/1    Running   0          8m34s
  basic-auth-plugin-85d885557c-g5ztl   1/1    Running   0          8m34s
  gateway-6ffb6bd755-7xhnv   2/2    Running   0          8m34s
  nats-5fdf6476f-lpvqx   1/1    Running   0          8m33s
  prometheus-96fd58985-x87xq   1/1    Running   0          8m32s
  queue-worker-7cbc9f8688-7phrt   1/1    Running   0          8m33s
  [root@master-0001 faas-netes]#
```

图 3-5-1

在此基础上，只需要将 Prometheus 暴露于外部端口进行访问即可。

首先，查看 Prometheus service 的类型和状态，具体命令为：

Kubectl describe svc prometheus -n openfaas

结果如下：

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
alertmanager	ClusterIP	10.102.161.247	<none>	9093/TCP	39h
basic-auth-plugin	ClusterIP	10.102.236.145	<none>	8080/TCP	39h
gateway	ClusterIP	10.101.56.175	<none>	8080/TCP	39h
gateway-external	NodePort	10.100.68.253	<none>	8080:31112/TCP	39h
grafana	NodePort	10.102.189.60	<none>	3000:30634/TCP	37h
nats	ClusterIP	10.111.15.38	<none>	4222/TCP	39h
prometheus	ClusterIP	10.107.58.30	<none>	9090/TCP	39h

图 3-5-2

可以看到，虽然 openfaas 自带组件中的 Prometheus 已经创建了相应的 pod、deployment、svc，正常运行于节点内部的 9090 端口，但是其运行模式为 clusterIP，没有绑定外部访问端口，不能通过弹性公网 IP 进行访问。

根据 Kubernetes 架构可知，k8s 中的 services 有三种类型，Cluster-IP、NodePort 和 LoadBalancer。其中，Cluster 仅使用一个集群内部 IP，仅可被集群内部的节点访问。而 NodePort 则可以在集群内部 IP 的基础上，开放相应的服务端口供外部访问。故在此，我们将 Prometheus 服务的类型变更为 NodePort 即可。

具体命令为：

`kubectl expose pods prometheus-96fd58985-x87xq --type=NodePort`

此命令将 Prometheus 服务对应的 pod 额外暴露出一个 NodePort 类型的服务，命令执行结果如下：

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
alertmanager	ClusterIP	10.102.161.247	<none>	9093/TCP	39h
basic-auth-plugin	ClusterIP	10.102.236.145	<none>	8080/TCP	39h
gateway	ClusterIP	10.101.56.175	<none>	8080/TCP	39h
gateway-external	NodePort	10.100.68.253	<none>	8080:31112/TCP	39h
grafana	NodePort	10.102.189.60	<none>	3000:30634/TCP	37h
nats	ClusterIP	10.111.15.38	<none>	4222/TCP	39h
prometheus	ClusterIP	10.107.58.30	<none>	9090/TCP	39h
prometheus-96fd58985-x87xq	NodePort	10.105.185.123	<none>	9090:30459/TCP	2m19s

图 3-5-3

可以看到，prometheus 服务已经可以通过外部端口 30459 进行访问。

Targets					
kubernetes-pods (1/1 up) <a href="#">show less</a>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.244.1.45:8082/metrics	UP	app="gateway" instance="10.244.1.45:8082" job="kubernetes-pods" kubernetes_namespace="openfaas" kubernetes_pod_name="gateway-4f6fd6d755-7xhew" pod_template_hash="6ff6bd755"	1.514s ago	1.507ms	

node_exporter_metrics (1/1 up) <a href="#">show less</a>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.0.213:9100/metrics	UP	instance="192.168.0.213:9100" job="node_exporter_metrics"	402ms ago	8.468ms	

prometheus (1/1 up) <a href="#">show less</a>					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	4.367s ago	7.382ms	

图 3-5-4

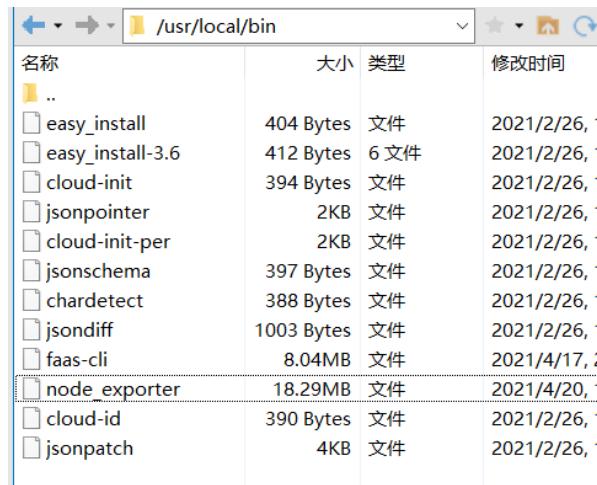
### 3.5.2 安装 Node-exporter

Openfaas 组件中自带的 Prometheus 仅可对 openfaas 函数级别的指标进行监控，如函数的调用次数等，并不能对 k8s 集群的系统级指标，如 CPU、内存、磁盘 IO 等情况进行监控。由于 openfaas 自带的 Prometheus 不能满足我们实验的需求，我们将借助 Prometheus 官方开发 Node-exporter 组件进行系统级资源的监控。

首先，下载 node-exporter 的可执行文件，解压并移动到/usr/local/bin 目录下，具体命令为：

```
wget
https://github.com/prometheus/node_exporter/releases/download/v0.18.1/node_exporter-0.18.0.linux-amd64.tar.gz
tar xvf node_exporter-0.18.0.linux-amd64.tar.gz
mv node_exporter-0.18.0.linux-amd64 /usr/local/bin/node_exporter
```

如下图所示：



名称	大小	类型	修改时间
..			
easy_install	404 Bytes	文件	2021/2/26, 1
easy_install-3.6	412 Bytes	文件	2021/2/26, 1
cloud-init	394 Bytes	文件	2021/2/26, 1
jsonpointer	2KB	文件	2021/2/26, 1
cloud-init-per	2KB	文件	2021/2/26, 1
jsonschema	397 Bytes	文件	2021/2/26, 1
chardetect	388 Bytes	文件	2021/2/26, 1
jsondiff	1003 Bytes	文件	2021/2/26, 1
faas-cli	8.04MB	文件	2021/4/17, 2
node_exporter	18.29MB	文件	2021/4/20, 1
cloud-id	390 Bytes	文件	2021/2/26, 1
jsonpatch	4KB	文件	2021/2/26, 1

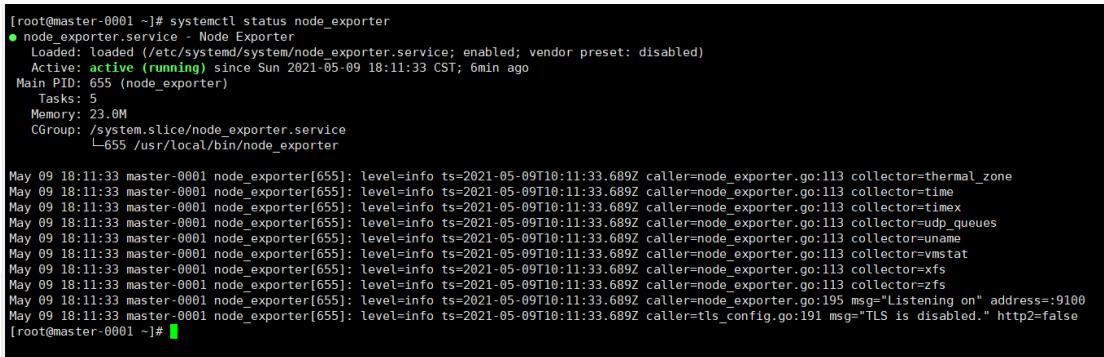
图 3-5-5

接着，创建名为 service.yaml 的配置文件，用于创建 node-exporter 系统服务，文件内容如下：

```
[Unit]
Description=node_exporter
Documentation=https://prometheus.io/
After=network.target
[Service]
Type=simple
User=prometheus
ExecStart=/usr/local/bin/node_exporter
Restart=on-failure
```

接着，启动 node-exporter 服务，具体命令为：

```
systemctl start node_exporter
systemctl status node_exporter
结果如下：
```

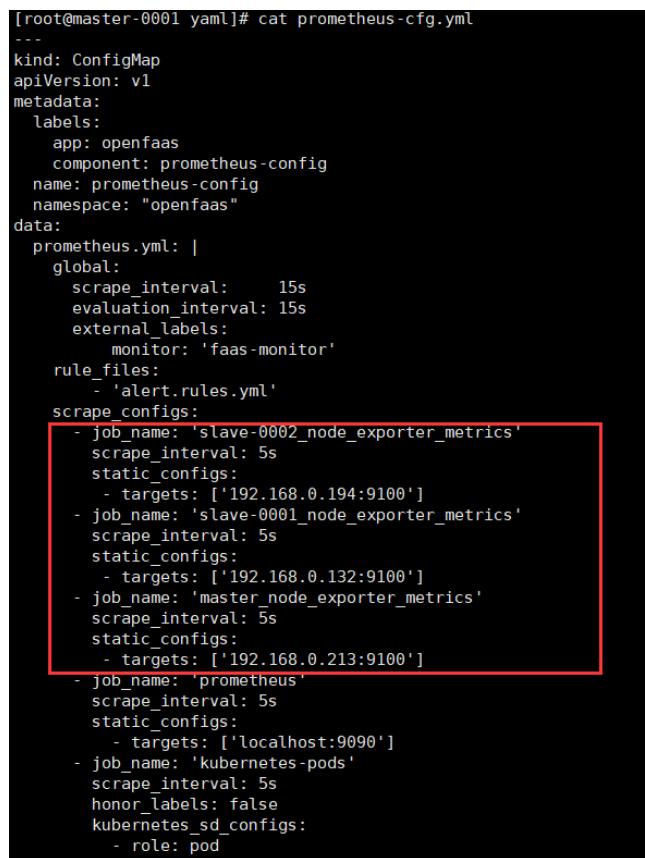


```
[root@master-0001 ~]# systemctl status node_exporter
● node_exporter.service - Node Exporter
  Loaded: loaded (/etc/systemd/system/node_exporter.service; enabled; vendor preset: disabled)
  Active: active (running) since Sun 2021-05-09 18:11:33 CST; 6min ago
    Main PID: 655 (node_exporter)
      Tasks: 5
     Memory: 23.0M
      CGroup: /system.slice/node_exporter.service
              └─655 /usr/local/bin/node_exporter

May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=thermal_zone
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=time
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=timex
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=udp_queues
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=uname
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=vmstat
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=xfs
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:113 collector=zfs
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=node_exporter.go:195 msg="Listening on" address=:9100
May 09 18:11:33 master-0001 node_exporter[655]: level=info ts=2021-05-09T10:11:33.689Z caller=tls_config.go:191 msg="TLS is disabled." http2=false
[root@master-0001 ~]#
```

图 3-5-6

此时，node-exporter 已经可以通过默认端口 9100 被 Prometheus 抓取。  
此后，修改位于/home/faas-netes/yaml 目录下的 prometheus-cfg.yml 配置文件，在 Prometheus 中添加对 node-exporter 的监控任务，具体内容如下：



```
[root@master-0001 yaml]# cat prometheus-cfg.yml
---
kind: ConfigMap
apiVersion: v1
metadata:
  labels:
    app: openfaas
    component: prometheus-config
  name: prometheus-config
  namespace: "openfaas"
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
      evaluation_interval: 15s
      external_labels:
        monitor: 'faas-monitor'
    rule_files:
      - 'alert.rules.yml'
    scrape_configs:
      - job_name: 'slave-0002_node_exporter_metrics'
        scrape_interval: 5s
        static_configs:
          - targets: ['192.168.0.194:9100']
      - job_name: 'slave-0001_node_exporter_metrics'
        scrape_interval: 5s
        static_configs:
          - targets: ['192.168.0.132:9100']
      - job_name: 'master_node_exporter_metrics'
        scrape_interval: 5s
        static_configs:
          - targets: ['192.168.0.213:9100']
      - job_name: 'prometheus'
        scrape_interval: 5s
        static_configs:
          - targets: ['localhost:9090']
      - job_name: 'kubernetes-pods'
        scrape_interval: 5s
        honor_labels: false
        kubernetes_sd_configs:
          - role: pod
```

图 3-5-7

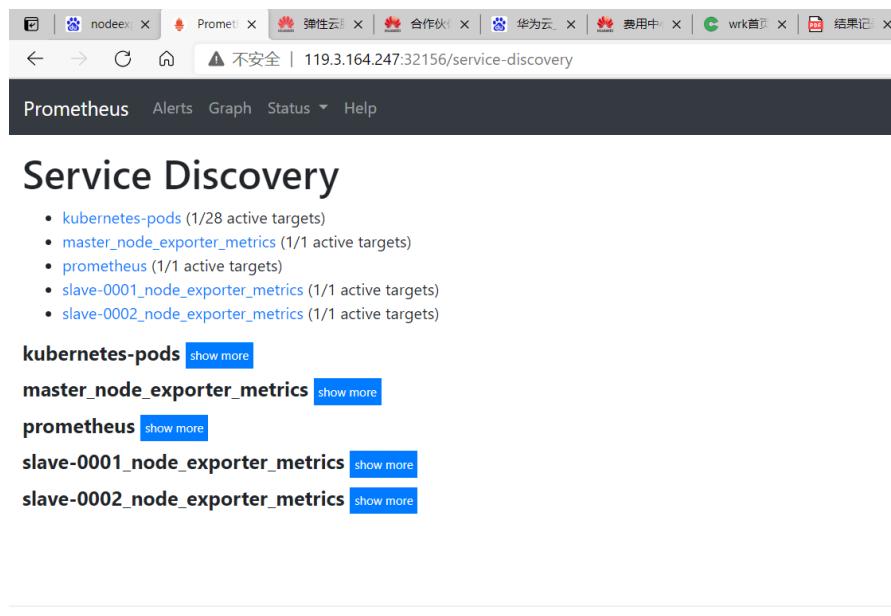
最后，应用新的配置文件，并删除旧的 Prometheus 服务所用的 pod，k8s 集群将自动按照新的配置文件拉取镜像，新的镜像即可对 node-exporter 的 metrics

进行监控。具体命令为：

```
kubectl apply -f prometheus-cfg.yml
```

```
kubectl delete pod prometheus-xxxxxxx-xxxxx(具体的镜像编号) -n openfaas
```

此时，再次打开 Prometheus，node-exporter 相关的 metrics 已经可以被监控：



The screenshot shows the Prometheus Service Discovery interface. The top navigation bar includes tabs for Prometheus, Alerts, Graph, Status, and Help. The main title is "Service Discovery". Below the title, a list of active targets is displayed:

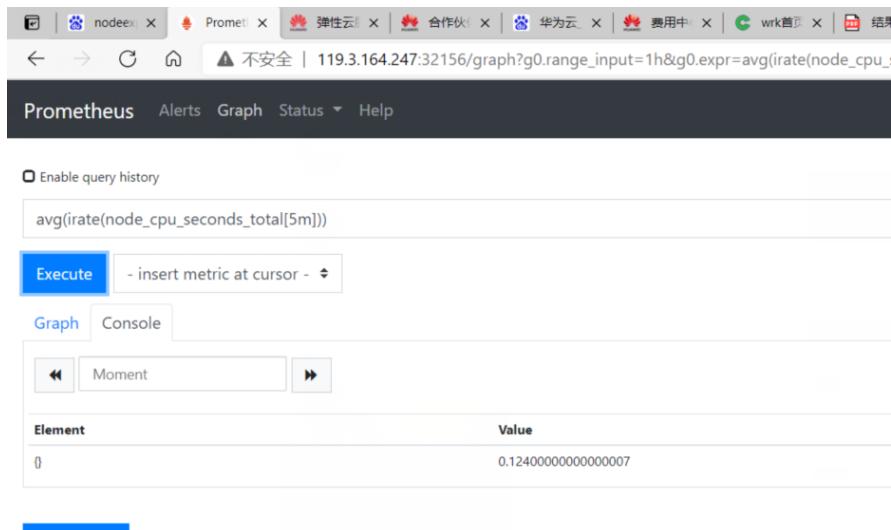
- [kubernetes-pods](#) (1/28 active targets)
- [master\\_node\\_exporter\\_metrics](#) (1/1 active targets)
- [prometheus](#) (1/1 active targets)
- [slave-0001\\_node\\_exporter\\_metrics](#) (1/1 active targets)
- [slave-0002\\_node\\_exporter\\_metrics](#) (1/1 active targets)

Below this list are five buttons, each corresponding to one of the metrics listed above, with "show more" links:

- [kubernetes-pods](#) [show more](#)
- [master\\_node\\_exporter\\_metrics](#) [show more](#)
- [prometheus](#) [show more](#)
- [slave-0001\\_node\\_exporter\\_metrics](#) [show more](#)
- [slave-0002\\_node\\_exporter\\_metrics](#) [show more](#)

图 3-5-8

可通过相关查询语句进行系统级别的指标的监控：



The screenshot shows the Prometheus Graph interface. The top navigation bar includes tabs for Prometheus, Alerts, Graph, Status, and Help. The main area contains a query input field with the following content:

```
avg(irate(node_cpu_seconds_total[5m]))
```

Below the query field is an "Execute" button and a dropdown menu labeled "- insert metric at cursor -". The interface is divided into two tabs: "Graph" (selected) and "Console". The "Graph" tab shows a single data series with the following data:

Element	Value
{} (empty)	0.12400000000000007

At the bottom of the interface is a blue "Add Graph" button.

图 3-5-9

### 3.5.3 Grafana 配置

下载 Grafana:

```
git clone https://github.com/bibinwilson/kubernetes-grafana.git
```

执行:

```
kubectl create -f deployment.yaml
```

```
kubectl create -f service.yaml
```

指定端口: 30634

登录 Grafana: <http://119.3.164.247:30634/login>, 初始用户和密码均为 admin。

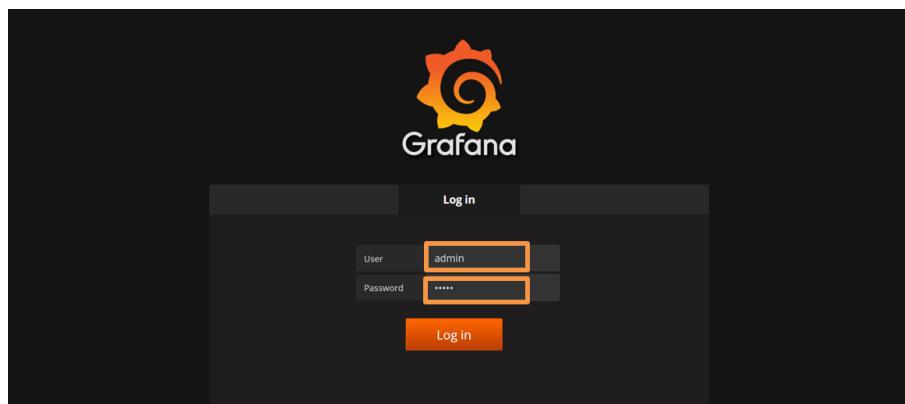


图 3-5-10

新建 data source, 连接 Prometheus。

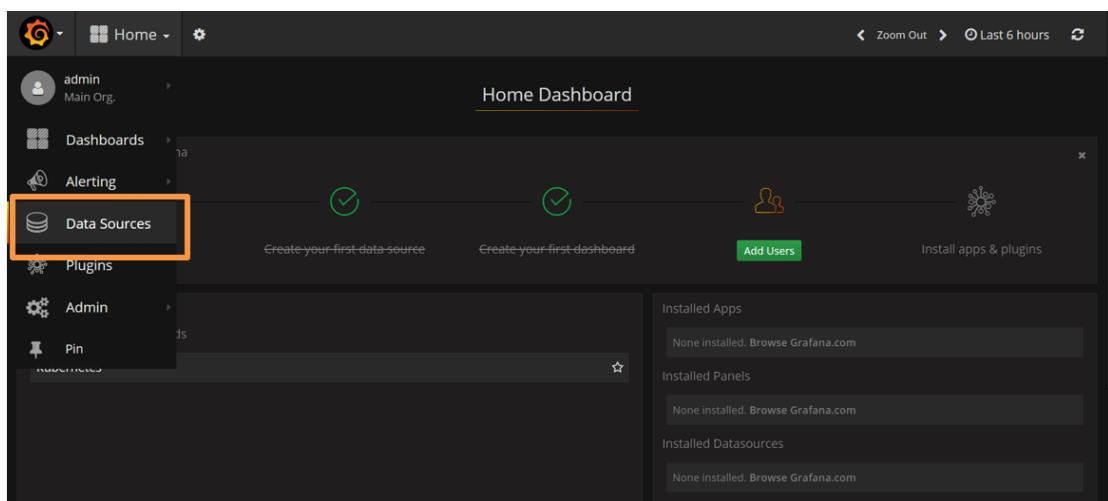


图 3-5-11

在设置中, 类型选择“Prometheus”, URL 填写对应的 9090 端口, 名称自定义。其他选项默认即可。

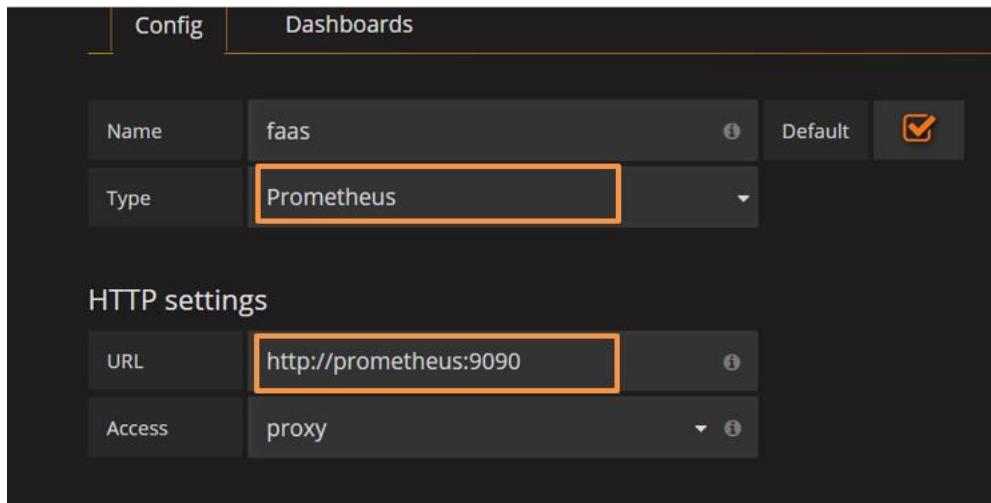


图 3-5-12

新建 Dashboards，自定义编辑图表。以 Master CPU 使用率图表的编辑为例：

Metrics 选项中选择预先设定好的 Prometheus 数据源（faas）；然后 Add Query，使用 PromQL 编写查询语句、指定监控的 job。

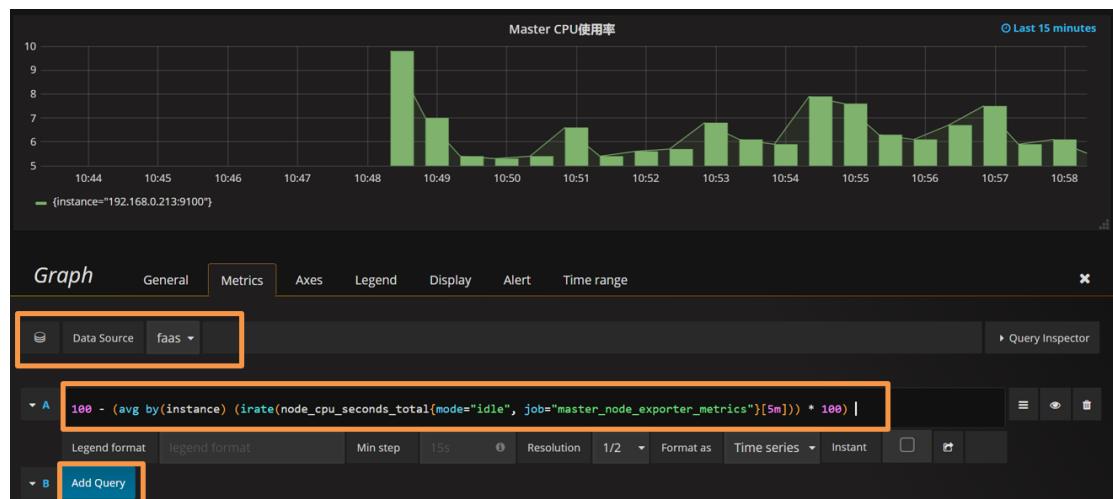


图 3-5-13

Time Range 选项中，可以调整横轴的时间范围，根据实验要求进行设定。

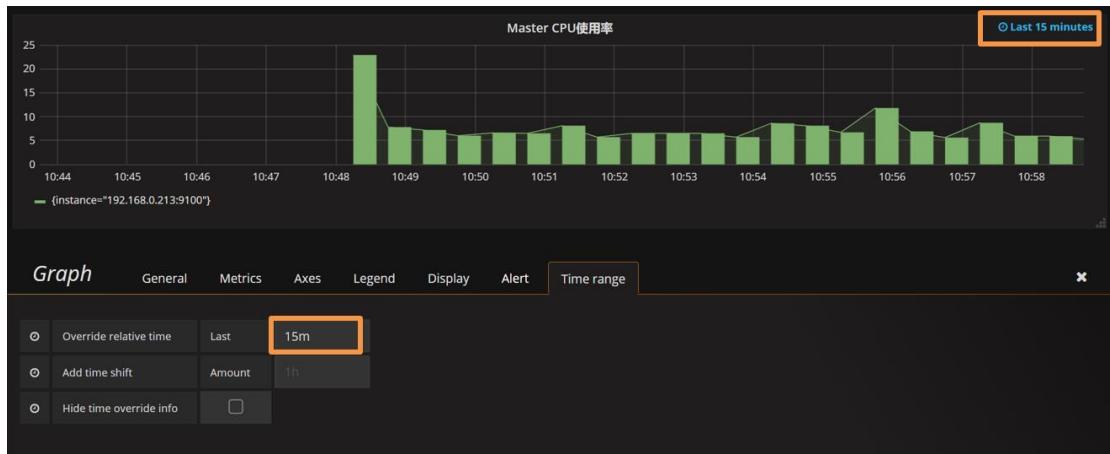


图 3-5-14

如果存在多个需要监控的 device，还需要增加右侧的图例。

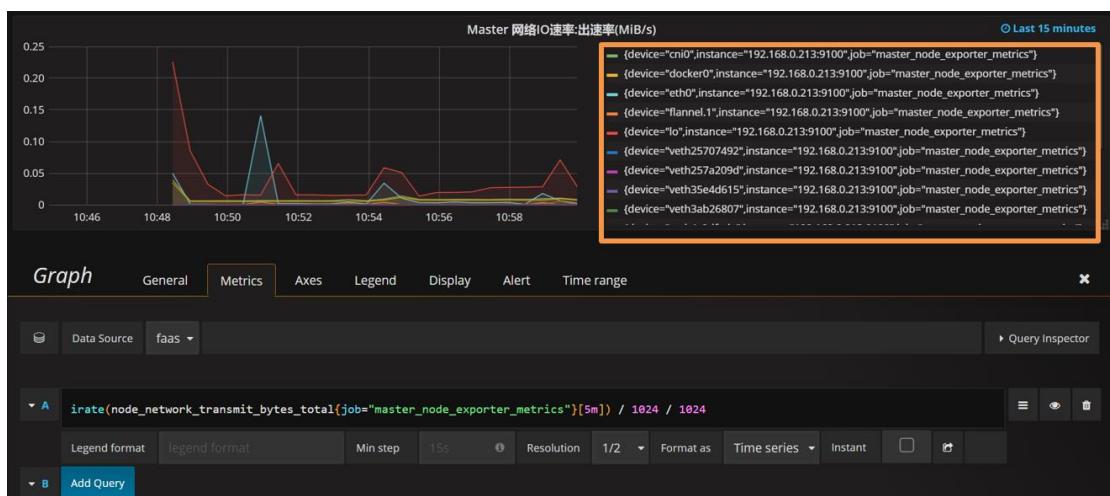


图 3-5-15

平台监控内容：各节点 CPU 使用率、磁盘、内存、磁盘 IO 读写、网络 IO 速率：



图 3-5-16



图 3-5-17

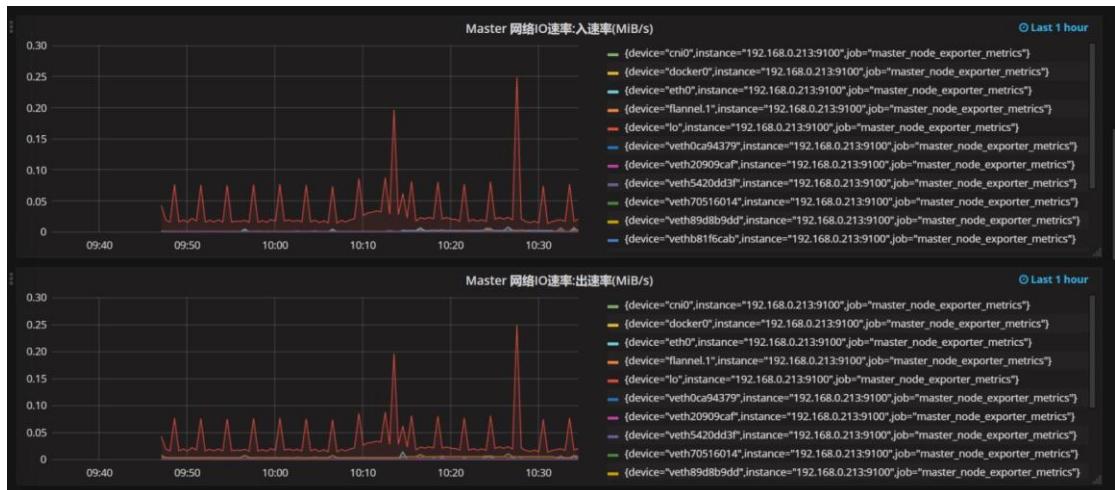


图 3-5-18



图 3-5-19

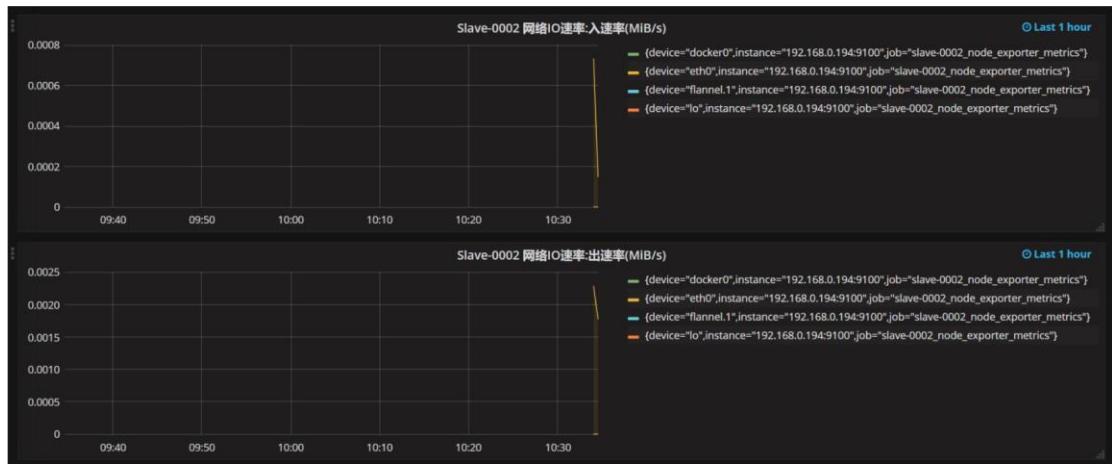


图 3-5-20

Grafana 监控平台配置完成后，可以将仪表盘导出为 JSON 文件，以便之后使用。

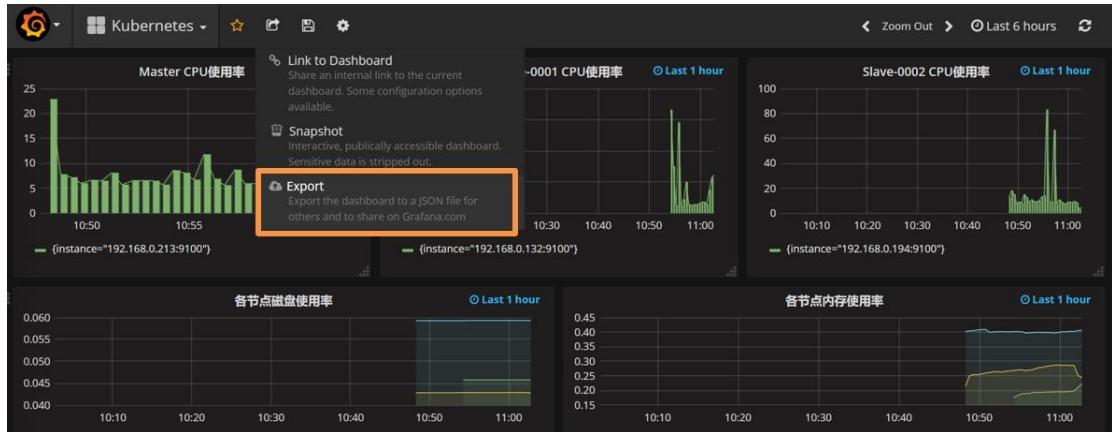


图 3-5-21

下次使用时，可以直接选择 import 之前保存的 JSON 文件。

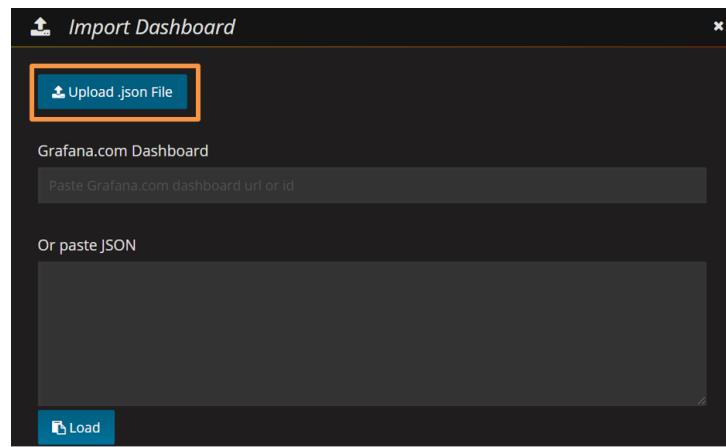


图 3-5-22

此外, Grafana.com 提供了大量的仪表盘模板, 可以提供监控平台设计参考。

### 3.6 测试函数编写 & 参数修改方法

除了示例函数 `hello-python` 之外, 实验中部署了两个用于测试的函数, 分别为进行浮点运算的 `float-operation` 和进行矩阵运算的 `matmul`。

函数的整体开发流程与上述的 `hello-python` 示例函数类似, 即使用模板新建函数文件、修改函数的 `handler` 文件和 `yaml` 配置文件、生成镜像、推送镜像、部署函数。

#### 3.6.1 Float-Operation 函数

这个函数用于测试浮点数的运算, 返回值为函数的计算时间。输入参数为计算规模 `N`, 函数会进行 `N` 次循环, 每次循环分别进行 `sin`, `cos` 和 `sqrt` 的运算。具体代码如下所示:

```
1  import math
2  from time import time
3
4  def float_operation(N):
5      start=time()
6      for i in range(0,N):
7          sin_=math.sin(i)
8          cos_=math.cos(i)
9          sqrt_=math.sqrt(i)
10     latency=time()-start
11     return latency
12
13 def handle(req):
14     """handle a request to the function
15     Args:
16         req (str): request body
17     """
18     n=int(req)
19     latency=float_operation(n)
20     return "latency: "+str(latency)
21     #return str(n)
```

图 3-6-1

不进行其他实验操作时, `yaml` 配置文件也只需要修改镜像来源和网关地址, 默认配置如下:

```

1  version: 1.0
2  provider:
3      name: openfaas
4      gateway: http://127.0.0.1:31112
5  functions:
6      float-operation:
7          lang: python3
8          handler: ./float-operation
9          image: willowd/float-operation
10

```

图 3-6-2

### 3.6.2 Matmul 函数

这个函数旨在测试矩阵运算的性能，返回值同样为计算时间。输入参数为矩阵规模  $N$ ，参数会初始化两个  $N \times N$  大小的矩阵，并令这两个矩阵进行相乘运算。

正常情况下，python 进行矩阵运算应该使用 numpy 库，openfaas 也提供了第三方库的使用方法，即将库名放入 requirements.txt 中。

但是在具体安装时，由于 openfaas 默认使用库的源镜像，导致安装失败，也不方便修改 pip 的镜像源，所以这里的矩阵相乘使用纯 python 代码暴力实现。具体代码如下：

```

1  import random
2  from time import time
3  import json
4
5  def matmul(n):
6      A=[[random.randint(0,100) for i in range(n)] for j in range(n)]
7      B=[[random.randint(0,100) for i in range(n)] for j in range(n)]
8      C=[[0 for i in range(n)] for j in range(n)]
9      start=time()
10     for i in range(n):
11         for j in range(n):
12             for k in range(n):
13                 C[i][j]=C[i][j]+A[i][k]*B[k][j]
14     latency=time()-start
15     return latency
16
17 def handle(req):
18     """Handle a request to the function
19     Args:
20         req (str): request body
21     """
22     json_req=json.loads(req)
23     n=int(json_req['n'])
24     result=matmul(n)
25     #return 'latency: '+str(result)
26     return result

```

图 3-6-3

yaml 配置文件默认情况下和上面 float-operation 相同，但还可以额外添加

其他信息，如资源限制和镜像设置。

若要设置副本数，可以在 yaml 中加入 labels 选项，通过以下三个参数来指定镜像数：

com.openfaas.scale.min: 指定最少的副本数，默认为 1  
com.openfaas.scale.max: 指定最大的副本数，默认为 20  
com.openfaas.scale.zero: 指定是否可以冷启动，如果为 true，则在不触发时，函数副本会变为 0. (OpenFaaS Pro 才能使用)  
com.openfaas.scale.factor: 指定每次扩容时增加的副本数，如果为 100，会直接增加到最大副本数。所以可以将 yaml 配置修改如下：

```
1  version: 1.0
2  provider:
3    name: openfaas
4    gateway: http://127.0.0.1:31112
5  functions:
6    matmul:
7      lang: python3
8      handler: ./matmul
9      image: willowd/matmul
10   label:
11     com.openfaas.scale.min: 2
12     com.openfaas.scale.max: 10
```

图 3-6-4

这表明函数最少有 2 个副本，最多可以有 10 个副本。（自动扩容的机制会在知识点部分详细阐述）

除了副本数，还可以在 yaml 文件中设置资源限制。使用 limits 标签可以限制该函数副本能够使用的最大资源数，可以限制 CPU 和内存。CPU 项设置为 1000m 代表可以使用 100% 的 CPU 时间，设置为 100m 代表可以使用 10% 的 CPU 时间，以此类推；内存则是直接指定可以使用的最大内存数，如果设置为 1024Mi 就代表这个函数副本最大可以使用 1GiB 的内存。

同时还可以使用 requests 标签来设置最小资源数，requests 标签内的值代表这个函数一般情况下会使用的资源数，OpenFaaS 会至少满足 requests 中所请求的资源。CPU 和内存的设置方法和上面相同。

下面是一个 yaml 配置：

```
1  version: 1.0
2  provider:
3    name: openfaas
4    gateway: http://127.0.0.1:31112
5  functions:
6    matmul:
7      lang: python3
8      handler: ./matmul
9      image: willowd/matmul
10   limits:
11     cpu: 100m
12     memory: 512Mi
13   requests:
14     cpu: 50m
15     memory: 62Mi
```

图 3-6-5

这里面说明了每个函数副本至少需要 5% 的 CPU 时间和 62MiB 的内存大小，同时这个副本最多不能够使用超过 10% 的 CPU 时间和 512MiB 的内存。

使用这个资源限制方法，在后续的实验和分析中我们设计了对应实验来探究 OpenFaaS 平台的计算方法和调度方法。

## 3.7 性能测试方法

在搭建好的 OpenFaaS 平台上，我们利用编写的函数测试了不同实例个数、资源限制、负载均衡等情况或机制下的集群性能。

具体地，我们利用编写好的浮点类型的计算密集型函数，在部署时改变其实例个数并对其进行 CPU 时间和内存使用上的限制，在此基础上借助 wrk 压力测试工具测试访问延迟、并借助 Prometheus + Grafana 监控平台观察记录各节点硬件资源的使用情况。

### 3.7.1 压力测试

可以使用 wrk 压力测试工具模拟并发的访问情形。wrk 是一个比较先进的 HTTP 压力测试工具，该工具使用多线程进行并发访问的模拟，在使用时可以指定线程数量、连接数量、持续时间等参数。

使用 wrk 进行访问的具体命令如下：

```
wrk -t 1 -c 1 -d 5s --latency -s post.lua
http://192.168.0.213:31112/function/float-operation
```

平均延迟、延迟分布、每秒请求数等统计结果如下图所示：

```
Running 5s test @ http://192.168.0.213:31112/function/float-operation
  1 threads and 1 connections
  Thread Stats      Avg      Stdev     Max  +/- Stdev
    Latency    46.11ms    2.12ms  59.32ms  95.37%
    Req/Sec    21.60      4.22    30.00    80.00%
  Latency Distribution
    50% 45.55ms
    75% 46.36ms
    90% 47.11ms
    99% 56.76ms
  108 requests in 5.00s, 26.75KB read
  Requests/sec: 21.58
  Transfer/sec: 5.34KB
```

图 3-7-1

### 3.7.2 OpenFaaS 资源限制实验

为了进一步探究实例数量、资源限制情况对于集群负载情况和 openfaas 函数

服务的性能影响和原理，我们设计了 openfaas 资源限制实验，来探究单个实例条件下资源限制对于函数性能的影响（可使用最大 CPU、可使用最大内存容量），以及多个实例条件下资源限制对于函数性能的影响（实例数、负载均衡），通过绘制关系曲线和绘制加速比曲线，得到结论。（具体实验数据详见实验结果分析部分）

## 四、实验结果分析

### 4.1 更改参数规模

限制线程数为 1，连接数为 1，改变参数规模。结果数据记录如下：

Float-operation								
参数规模	Latency				Req/Sec			
	avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev
10	46.11	2.12	59.32	95.37%	21.6	4.22	30	80%
100	47.01	1.81	62.61	95.28%	21.2	3.28	30	88%
10000	49.93	1.4	56.6	85%	20	2.02	30	96%

Matmul								
参数规模	Latency				Req/Sec			
	avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev
50	47.85	2.61	62.01	89.00%	20.9	3.21	30	89%
100	46.81	1.336	56.74	84.04%	21.27	3.68	30	85%

表 4-1-1

实时监控平台情况如下：（以 Float-operation 参数规模-10 为例）

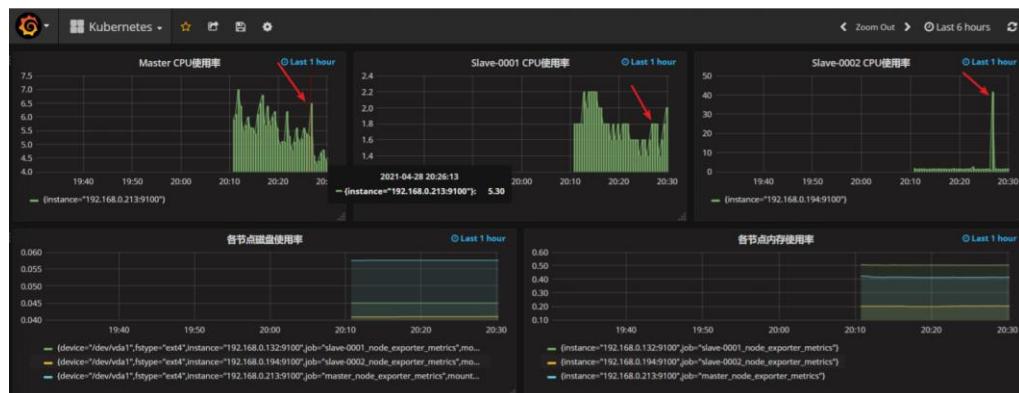


图 4-1-1

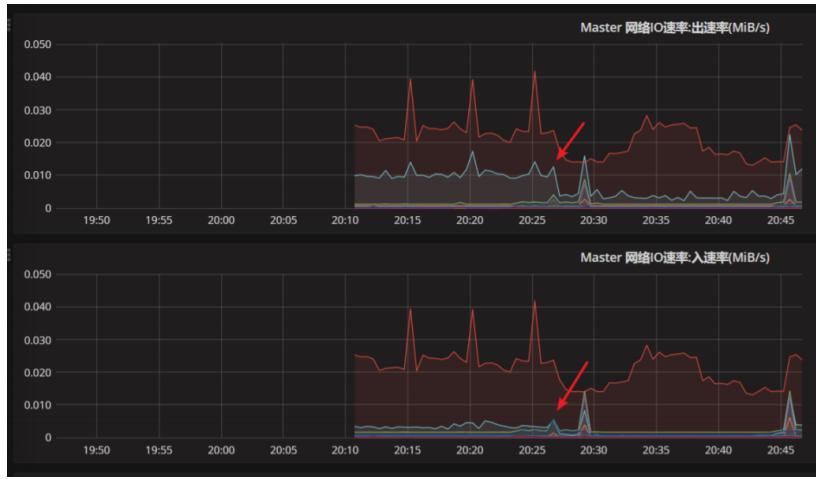


图 4-1-2

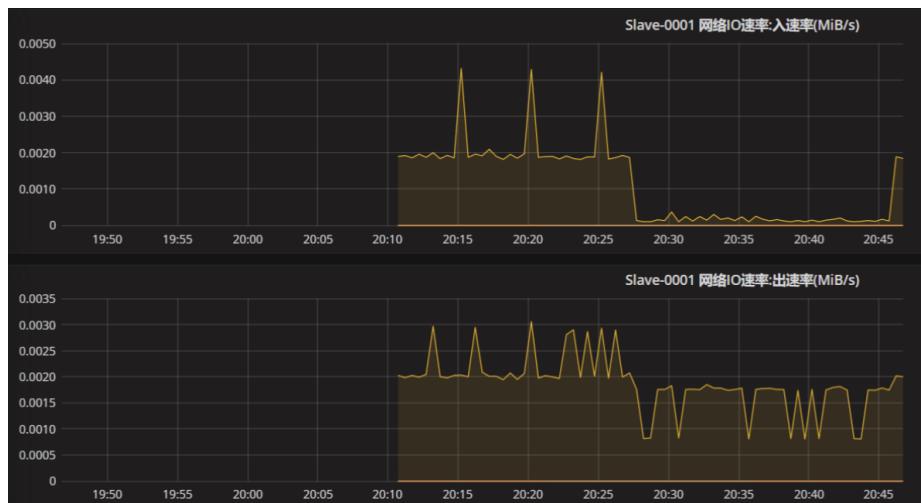


图 4-1-3

可以看出：随着函数运算的参数规模的扩大，平均延迟与每秒完成请求数大致呈上升趋势，符合实际。且在函数执行期间，CPU 使用率有明显的上升峰值。同时能明显观察到节点间的网络互通。

## 4.2 修改线程数

保持参数规模不变，修改连接数和线程数，数据记录如下：

Float-operation								
线程数	Latency				Req/Sec			
	avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev
4	95.35	22.19	154.54	66.83%	10.67	3.18	20	86%
8	193.4	38.93	312.83	73.63%	5.58	2.1	10	71%
Matmul								
线程数	Latency				Req/Sec			
	avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev
4	675.63	130.95	860.09	66.67%	1.15	0.36	2	85%
8	1.38	415.91	1.97	56.00%	0.24	0.52	2	80%

表 4-2-1

实时监控平台情况如下：（以 Float-operation 函数 线程数-4 为例）

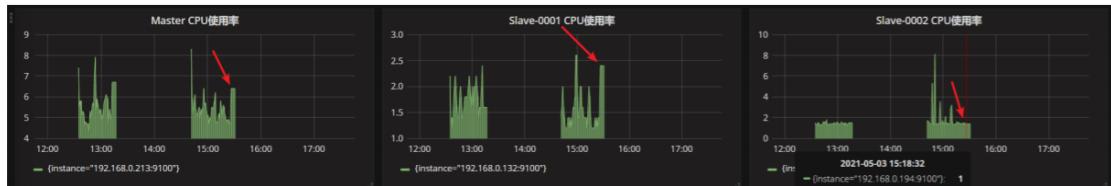


图 4-2-1

可以看出：随着线程数的增加（扩倍），平均延迟与每秒完成请求数均有显著下降，性能提升明显。且在函数执行期间，master 与 slave-0001 的 CPU 使用率有明显的上升峰值，slave-0002 的 CPU 使用率几乎没有波动。

据我们分析：这是因为一般情况下 K8s 调度器会尽量的使得属于同一服务的实例运行在不同的节点上。但是 k8s 调度算法还会综合考虑系统可支配资源等多种因素，并不会严格的平均分配。

### 4.3 资源限制实验（使用计算密集型 float-operation 函数）

#### 4.3.1 单个实例条件下资源限制对于函数性能的影响

首先，限制实例个数为 1，改变实例可使用的最大 CPU 时间和内存容量，观察并记录平均延迟、单线程每秒完成请求数等指标的变化情况。

- 限制可使用的最大 CPU（千分之核心）

限制实例个数为 1，可使用最大内存容量为 1024，改变实例可使用的最大 CPU。实验数据记录如下：

实例数	CPU	Mem	N	线程数	Latency				Req/Sec				Latency Distribution				
					avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev	99%	90%	75%	50%	
	1000	1024	10000	1	50.09	3.26	80.61	96.97%	19.8	2.47	30	94%	80.61	51.38	50.1	49.6	
1	750	1024	10000	1	65.05	11.63	82.54	60.53%	15.2	5.03	20	52%	82.54	75.22	73.7	72.59	
1	500	1024	10000	1	98.4	6.62	101.17	98%	9.98	0.14	10	98%	101.2	100.3	99.9	99.41	
1	250	1024	10000	1	199.82	3.11	202.51	96%	5	0	5	100%	202.5	201.5	201	200.2	
1	100	1024	10000	1	520.72	41.12	593.42	77.78%	1.56	0.53	2	55.56%	593.4	593.4	503	502.2	
1	50	1024	10000	1	1140	70.64	1200		0	0	0		1.2s	1.2s	1.2s		

表 4-3-1

据上述实验数据，绘制可使用最大 CPU 与平均延迟、每秒完成请求数的关系图：

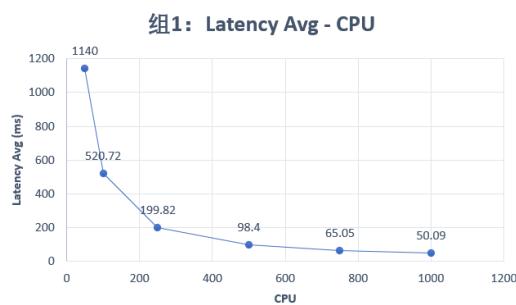


图 4-3-1(a)

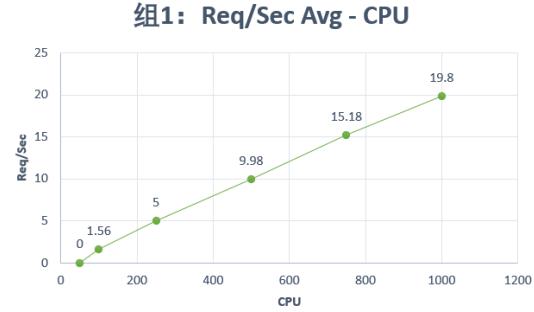


图 4-3-1(b)

可以看出：随着可使用 CPU 的提高，平均延迟逐渐降低。从图 4-3-1(a)中可知，可使用的 CPU 为  $50/1000=5\%$  时，平均延迟为 1140ms；而在可使用 CPU 提高至  $1000/1000=100\%$  时，平均延迟降至 50.09ms。也就是说，CPU 可使用时间提高 20 倍，平均延迟降至 1/20，两者大致呈现为反比例关系。

随着 CPU 可使用时间的提高，每秒完成请求数逐渐增加。从图 4-3-1(b)中可知，可使用的 CPU 为  $100/1000=10\%$  时，平均每秒请求数为 1.56；而在可使用 CPU 提高至  $1000/1000=100\%$  时，平均每秒请求数升至 19.8。也就是说，CPU 可使用时间提高 10 倍，每秒完成请求数提高 10 倍，两者大致呈现为线性关系。

### ● 限制可使用的最大内存容量

限制实例个数为 1，可使用的最大 CPU 时间为  $1000/1000 = 100\%$ ，改变实例可使用最大内存容量。实验数据记录如下：

实例数	CPU	Mem	N	线程数	Latency				Req/Sec				Latency Distribution				
					avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev	99%	90%	75%	50%	
	1000	1024	10000	1	50.17	1.21	57.44	85.86%	19.8	1.41	20	98.00%	57.44	51.22	50.59	49.99	
1	1000	512	10000	1	50.23	1.23	54.97	73.74%	19.8	1.41	20	98.00%	54.97	52.16	50.99	49.88	
1	1000	256	10000	1	50.45	1.35	55.32	74.75%	19.8	1.41	20	98.00%	55.32	52.52	51.18	50.07	
1	1000	128	10000	1	49.51	0.88	52.59	70.30%	20.2	2.47	30	94.00%	52.02	50.58	50.00	49.38	
1	1000	64	10000	1	50.12	1.21	55.64	82.83%	19.8	3.19	30	90.00%	55.64	51.76	50.73	49.72	
1	1000	32	10000	1	50.12	2.06	68.08	95.96%	19.8	1.41	20	98.00%	68.08	51.01	50.33	49.83	
1	1000	16	10000	1	51.02	1.63	59.19	79.59%	19.6	1.98	20	96.00%	59.19	53.02	51.80	50.70	

表 4-3-2

据上述数据，绘制最大内存容量与平均延迟、每秒完成请求数的关系图：

组2: Latency Avg - Mem

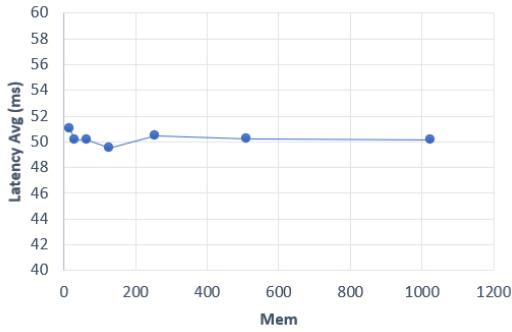


图 4-3-2(a)

组2: Req/Sec Avg - Mem

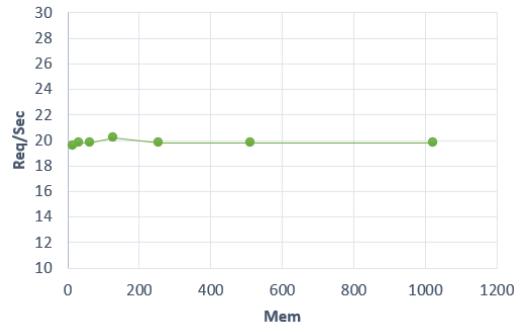


图 4-3-2(b)

可以看出：随着实例可使用最大内存容量的提高，平均延迟基本维持不变。从图 4-3-2(a)中可知，实例可使用最大内存容量为 16、32、64、128、256、512 和 1024 时，平均延迟均在 50ms 左右波动。由此可见，提高实例可使用最大内存容量的方法，并不能降低延迟和提升集群在处理 float-operation 函数请求时的性能。

同样，随着实例可使用最大内存容量的提高，每秒完成请求数基本维持不变。从图 4-3-2(b)中可知，实例可使用最大内存容量为 16、32、64、128、256、512 和 1024 时，平均每秒请求数均在 20 左右波动。由此可见，提高实例可使用最大内存容量的方法，并不能增加每秒请求数，提升集群在处理 float-operation 函数请求时的性能。

总的来说，对于 float-operation 函数，最大内存容量的设置不影响集群性能表现，这是因为该函数是计算密集型函数 (**CPU-bound**)，性能瓶颈是 CPU 资源。

### 4.3.2 多个实例条件下资源限制对于函数性能的影响

接着，我们多次改变函数的实例个数和资源限制，研究其对函数的实际运行实例个数、负载均衡情况以及性能的影响。

#### ● 实例数

限制可使用的最大 CPU 为 1000 (100%)，Mem 为 512，线程数为 1，改变实例数。实验数据记录如下：

组3: 单线程换实例数

实例数	CPU	Mem	N	线程数	Latency				Req/Sec				Latency Distribution			
					avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev	99%	90%	75%	50%
1	1000	512	10000	1	50.1	1.14	55.45	79.00%	20	1.41	20	98.00%	55.45	51.7	50.5	49.81
2	1000	512	10000	1	50.03	1.51	57.99	89.00%	20	1.41	20	98.00%	57.99	51.5	50.4	49.64
4	1000	512	10000	1	51.98	3.27	70.62	91.67%	19	2.74	20	92.00%	70.62	54.4	52.8	51.29
8	1000	512	10000	1	50.34	1.64	58.95	83.84%	20	2.42	20	98.00%	58.95	52.9	50.7	49.89
16	1000	512	10000	1	50.46	2.13	62.31	88.89%	20	2.47	30	94.00%	62.31	53.1	50.8	49.91

表 4-3-3

根据上述实验数据，分别绘制实例数与平均延迟、每秒完成请求数的关系图：

组3: Latency Avg - 实例数(单线程)

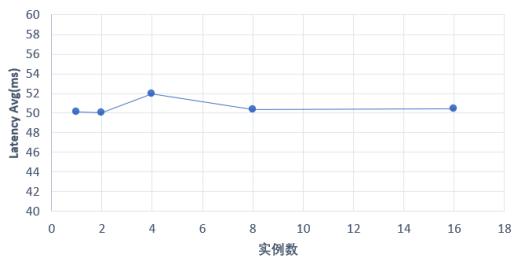


图 4-3-3(a)

组3: Req/Sec - 实例数(单线程)

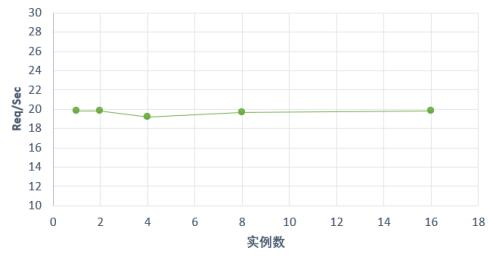


图 4-3-3(b)

可以看出: 在单线程条件下, 实例数的增加, 平均延迟和每秒完成请求数均基本维持不变。这是因为该单线程访问的时候, 计算任务只能分配到一个实例上去, 增加实例数无效。

### ● 多个实例条件下的资源限制与负载均衡

实验数据记录如下:

组4

设置实例	实际镜像	CPU	Mem	N	线程数	Latency			Req/Sec				Latency Distribution				
						avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev	99%	90%	75%	50%
1	1	1000	1024	50	1000	1980	821.05	2890	78.26%	0.29	0.81	5	92.30%	2830	2700	2620	2520
3	3	1000	1024	50	1000	839.07	245.49	1380	72.96%	0.89	0.84	5	61.10%	1320	1130	1020	852.3
6	3	1000	1024	50	1000	819.78	281.76	1470	69.14%	0.91	1.13	10	95.10%	1440	1160	1030	817.9
6	6	500	512	50	1000	658.5	558.93	2230	63.55%	3.67	4.54	20	77.10%	2100	1490	1050	534
9	9	100	512	50	1000	515.32	635.68	3870	86.29%	5.35	5.06	20	82.30%	3020	1410	675.7	251.2
12	11	50	64	50	1000	623.85	686.43	4320	84.35%	4.6	4.41	20	54.50%	2940	1640	869	364.5
15	12	3	8	50	1000	414.01	416.27	1940	81.55%	5.27	4.9	20	82.20%	1690	1130	619.9	236.4
18	13	1	1	50	1000	312.57	393.37	1990	84.24%	6.99	5.81	20	66.60%	1700	917	364.1	118.4
21	13	1	1	50	1000	365.19	415.62	1880	84.26%	5.81	5.05	20	82.20%	1850	999	528.4	169.3

表 4-3-4(a)

设置实例数	实际镜像数	master	slave-0001	slave-0002	timeout
1	1	0	0	1	0
3	3	1	1	1	0
6	3	1	1	1	0
6	6	1	3	2	0
9	9	1	5	3	0
12	11	1	5	5	0
15	12	1	5	6	0
18	13	1	5	7	3
21	13	1	5	7	14

表 4-3-4(b)

计算不同实际镜像数下的平均延迟加速比:

实际镜像数	平均延迟	加速比
1	1980	1.00
3	839.07	2.36
6	658.5	3.01
9	515.32	3.84
11	623.85	3.17
12	414.01	4.78
13	312.57	6.33

表 4-3-4(c)

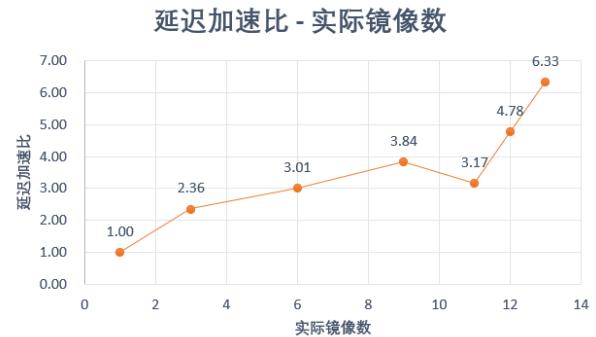


图 4-3-4

实例数	CPU	Mem	N	线程数	Latency				Req/Sec				Latency Distribution				
					avg(ms)	std(ms)	max(ms)	stdev	avg	std	max	stdev	99%	90%	75%	50%	
					4	1000	512	10000	1000	43.17	18.01	55.73	83.48%	23	11.29	80	84.00%
4	500	512	10000	1000	53.31	11.62	104.68	93.55%	18.6	5.35	30	70.00%	104.7	52.59	51.1	50.04	
4	250	512	10000	1000	91.8	59.45	202.63	85.48%	11.8	4.74	20	67.50%	202.6	194.2	107	77.64	
4	125	512	10000	1000	316.83	124.08	403.52	80.00%	3.87	3.27	10	80.00%	403.5	403.2	402	397.1	
4	100	512	10000	1000	463.76	75.53	592.17	80.00%	1.7	0.48	2	70.00%	592.2	592.2	503	501	
4	50	512	10000	1000	624.15	230.84	998.32	75.00%	1.25	0.71	2	50%	998.3	998.3	998	502.2	

表 4-3-5

据表 4-3-5 的数据, 绘制可使用最大 CPU 与平均延迟、每秒完成请求数的关系图:

组5: Latency Avg - CPU

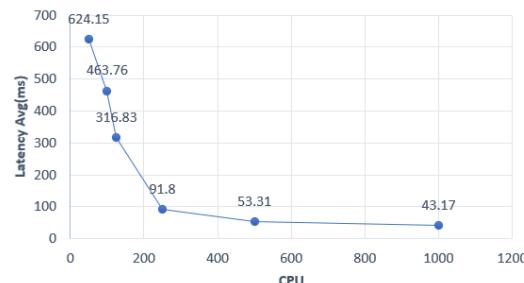


图 4-3-5(a)

组5: Req/Sec Avg - CPU

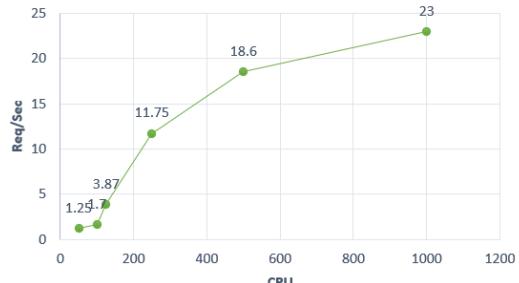


图 4-3-5(b)

可以看出:

对比单个实例条件下增加实例数的实验, 组 4 将线程数设置为 1000, 所以计算任务可以分配到多个实例上去, 观察延迟加速比-实际镜像数图表, 可以发现, 随着实例数的增加, 加速比显著提升, 符合实际。

由组 5 可知, 多实例条件下, 控制可使用的最大 CPU 时间的实验结论大致与单实例条件相同, 唯一不同的是每秒完成请求数不再与最大可使用 CPU 呈明显的线性关系。

函数实际运行的实例数量可能并不是部署时指定的数量, 由于系统级别的资源限制, 可能会导致实际处于 running 状态可参与计算的实例个数远小于指定的实例个数。如表中设置实例为 15、18 和 21 时的情况, 此时, 查看 k8s 集群 pod 的状态如下:

```
node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From           Message
  ----  -----  ----  ----
  Warning FailedScheduling 115s default-scheduler 0/3 nodes are available: 3 Insufficient cpu.
  Warning FailedScheduling 115s default-scheduler 0/3 nodes are available: 3 Insufficient cpu.
[root@master-0001 functions]#
```

图 4-3-6

这是由于实例创建时 `requests` 指定的系统资源超过了实际可使用的资源余量，导致该镜像并没有正常运行。

在创建实例时降低其请求的系统资源量，可以增加实际运行的实例数量。如组 4 表中所示，设置实例数从 3 增至 6 时，尽管指定了运行 6 个实例，但是由于资源限制仅有 3 个实际运行，导致平均延迟和线程每秒完成请求数基本不变。

创建函数实例时，函数实例具体运行于哪个节点，是由 k8s 调度算法所决定的。参照知识点“（九）资源调度”，k8s 的默认调度器会根据优先级算法尽可能地进行负载的均衡，一般情况下会尽量的使得属于同一服务的实例运行在不同的节点上。但是 k8s 调度算法还会综合考虑系统可支配资源等多种因素，并不会严格的平均分配。

例如，如组 4 所示，在设置实例数从 6 增加到 12 时（实际镜像数从 6 增至 11），新增加的实例近似均匀的分配在了 `slave-0001` 和 `slave-0002` 两个节点上，这是由于 `master` 节点的内存有超过一半的容量用于维持整个集群的运行，余量较小所致。

下图展示了 Prometheus 中三个节点的内存使用情况，可见，即使是在集群空闲没有计算任务的情况下(CPU 处于谷值，如 23: 00)，`master` 节点依然有大约 60% 的内存处于占用状态，这很可能是 k8s 调度算法不愿意将新的实例分配到 `master` 的原因。

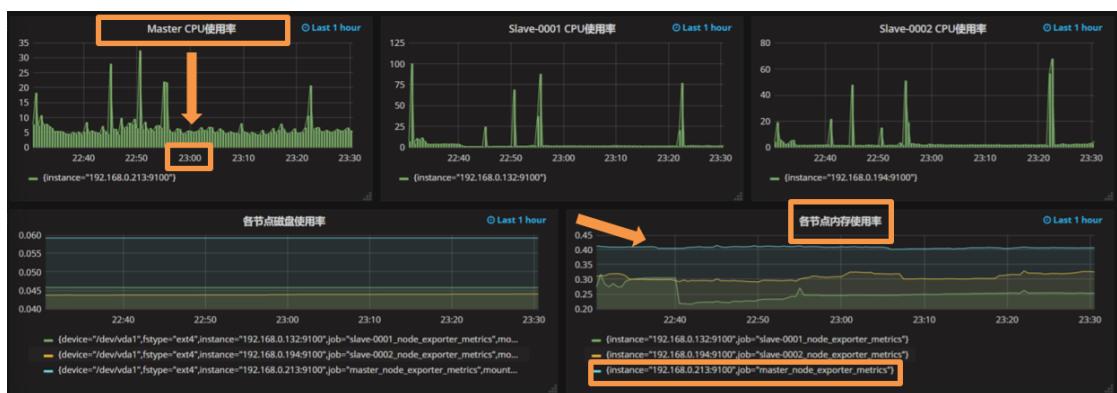


图 4-3-7

尽管可以通过降低实例可使用的最大资源来使得更多的实例处于运行状态，提高函数在高并发条件下的性能。但是实例的可使用资源并不能无限制的降低，如组 4 所示，设置实例数为 18、21 时，CPU 和 Mem 被限制在 1，而实际镜像分别为 12、13、13。

由此可见，即使实例可使用的资源被指定为最低，也无法在运行更多的实例了。这可能是由于，每个实例的本质是一个镜像，在操作系统级别即为一个进程，

进程本身的所需的内存大小并不能无限制的降低，其必然存在一个最小的下限，这限制了实例数的进一步增加。

任务规模恒定的条件下，函数的加速比并不是随着指定的实例数增加，而是随着可运行的实例数增加，最大加速比约等于整个集群的核数。这是由于每个实例在操作系统级别的本质是一个进程，增加可运行的实例等同于使用多个进程来处理并发的访问请求。本实验中最大加速比约为 6 左右，符合阿姆达尔定律中“可并行部分趋近于 1 时，加速比的上限约等于核数”的规律，与理论相符。

最后，实验整体结论如下：

- a) 适当增加实例数可以降低并发访问延迟，提高加速比，提升集群处理计算密集型任务的性能。要最高效地利用 CPU，计算密集型任务同时进行的数量应当等于 CPU 的核心数。
- b) 适当的进行资源限制、调整负载均衡情况，可以提高实际运行的实例数，达到更高的计算效率。

#### 4.4.3 性能测试中遇到的问题

##### 1) 压力测试中的线程池限制问题

起初在进行压力测试时，我们借助 wrk 压力测试工具，指定线程数和连接数为 1000，从单一节点向集群发送函数调用请求。然而 wrk 的统计结果中存在大量的 timeout 情况，且不同参数设置下的平均延迟基本不变，测试结果存在异常。

如下图：

```
[root@master-0001 functions]# wrk -t 1000 -c 1000 -d 5s --latency -s post.lua http://192.168.0.213:31112/function/float-operation
Running 5s test @ http://192.168.0.213:31112/function/float-operation
 1000 threads and 1000 connections
 Thread Stats      Avg      Stdev     Max  +/- Stdev
  Latency    1.15s   419.94ms   1.84s   73.08%
  Req/Sec    0.12      0.33     1.00   87.50%
Latency Distribution
  50%   1.01s
  75%   1.55s
  90%   1.80s
  99%   1.84s
  88 requests in 5.10s, 21.66KB read
  Socket errors: connect 0, read 0, write 0, timeout 62
  Non-2xx or 3xx responses: 1
Requests/sec:    17.24
Transfer/sec:   4.24KR
[root@master-0001 functions]#
```

图 4-4-1

我们猜测是由于参数设置不合理所致。查询压力测试相关资料了解到，wrk 工具的并发模拟基于多线程实现，测试时进程启动多个线程切换来模拟并发的访问。然而根据操作系统和体系结构相关知识，每个进程在同一时刻只会有一个线程处于 running 状态，过多的线程数必然会导致频繁的线程上下文切换过程，降低实际的性能。此外，同一个进程所能启动的线程数量受到线程池规模大小的限制，线程数量规模过大必然会导致线程的排队等待，这将使得延迟大大增加。

在我们的资源限制实验中，我们希望访问延迟的瓶颈是 openfaas 的函数计算过程，而不是发送端的线程切换和排队等待过程。因此，我们缩小线程为

50、并发数为 50 进行后续的实验，此时随着实例数和资源限制的改变，延迟变化明显。

## 2) k8s 负载不均衡问题

起初，我们调节 openfaas 函数实例的资源限制，访问延迟和其他指标变化不明显，并且镜像频繁崩溃，出现 Non-2xx or 3xx responses 的错误。

如下图所示：

```
[root@master-0001 functions]# wrk -t 2 -c 2 -d 60s --latency -s post.lua http://119.3.164.247:31112/function/float-operation
Running 1m test @ http://119.3.164.247:31112/function/float-operation
2 threads and 2 connections
  Thread Stats      Avg      Stdev     Max  +/- Stdev
    Latency      0.00us      0.00us    0.00us  -nan%
    Req/Sec      0.00      0.00      0.00  100.00%
  Latency Distribution
    50%      0.00us
    75%      0.00us
    90%      0.00us
    99%      0.00us
  8 requests in 1.00m, 2.30KB read
  Socket errors: connect 0, read 0, write 0, timeout 8
  Non-2xx or 3xx responses: 8
  Requests/sec:      0.13
  Transfer/sec:    39.28B
```

图 4-4-2

借助 Prometheus 和 k8s 的监控信息，我们发现此时所有的实例都被分配到了单个节点上。如下图所示：



图 4-4-3

```
[root@master-0001 functions]# kubectl get pod -n openfaas-fn -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP           NODE   NOMINATED NODE   READINESS GATES
float-operation-548c849596-22fpr  1/1   Running   1   79m  10.244.2.61  slave-0002  <none>  <none>
float-operation-548c849596-418g2  1/1   Running   1   80m  10.244.2.58  slave-0002  <none>  <none>
float-operation-548c849596-55xtj  1/1   Running   1   79m  10.244.2.65  slave-0002  <none>  <none>
float-operation-548c849596-5c82p  1/1   Running   1   83m  10.244.2.63  slave-0002  <none>  <none>
float-operation-548c849596-6cprv  1/1   Running   2   79m  10.244.2.57  slave-0002  <none>  <none>
float-operation-548c849596-7pnqb  1/1   Running   1   78m  10.244.2.55  slave-0002  <none>  <none>
float-operation-548c849596-866gm  1/1   Running   1   79m  10.244.2.62  slave-0002  <none>  <none>
float-operation-548c849596-9n5t8  1/1   Running   1   79m  10.244.2.54  slave-0002  <none>  <none>
float-operation-548c849596-cznj2  1/1   Running   1   79m  10.244.2.67  slave-0002  <none>  <none>
float-operation-548c849596-d4ttr  1/1   Running   1   79m  10.244.2.71  slave-0002  <none>  <none>
float-operation-548c849596-jzgtf  1/1   Running   1   89m  10.244.2.59  slave-0002  <none>  <none>
float-operation-548c849596-s56k8  1/1   Running   1   78m  10.244.2.68  slave-0002  <none>  <none>
float-operation-548c849596-s9q85  1/1   Running   2   79m  10.244.2.66  slave-0002  <none>  <none>
float-operation-548c849596-tqmgr  1/1   Running   1   78m  10.244.2.60  slave-0002  <none>  <none>
float-operation-548c849596-vx7bz  1/1   Running   1   78m  10.244.2.72  slave-0002  <none>  <none>
float-operation-548c849596-x2nmk  1/1   Running   2   78m  10.244.2.56  slave-0002  <none>  <none>
[root@master-0001 functions]#
```

图 4-4-4

继续深入查找原因，我们发现这是由于每个实例的系统资源限制设置过小导致的(CPU: 1m/Memory: 1Mi)。K8s 集群的默认调度算法会综合考虑多种评价指标，进行实时动态的任务调度。一般情况下，默认调度算法会综合考虑节点的剩余资源量和反亲和性等多个指标，但是由于我们实验之初设置的资源限制过小，导致集群直接将所有的任务调度到了同一个节点去。

在适当的提高实例可使用的最大资源后，重新部署，此时 k8s 默认调度算法已经不再将任务全部调度至同一节点，多个节点的系统资源都得到了有效的

利用。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
float-operation-546d9d6f66-mwtd	1/1	Running	0	8m48s	10.244.2.136	slave-0002	<none>	<none>
float-operation-5574cf44f6-4lf8k	1/1	Running	0	31m	10.244.1.166	slave-0001	<none>	<none>
float-operation-55bb48774c-7vzkm	0/1	Pending	0	3m13s	<none>	<none>	<none>	<none>
float-operation-55ff998f5b-4sr8x	1/1	Running	0	30m	10.244.2.133	slave-0002	<none>	<none>
float-operation-55ff998f5b-vbx5j	1/1	Running	0	30m	10.244.1.167	slave-0001	<none>	<none>
float-operation-5787b1f4f4-zw552	1/1	Running	0	35m	10.244.1.164	slave-0001	<none>	<none>
float-operation-6886794f5d-tbct7	1/1	Running	0	42m	10.244.0.295	master-0001	<none>	<none>
float-operation-6886794f5d-wx929	1/1	Running	0	44m	10.244.2.131	slave-0002	<none>	<none>
float-operation-6f6f866777-ljrgb	1/1	Running	0	37m	10.244.1.163	slave-0001	<none>	<none>
float-operation-6f6f866777-nfvms	1/1	Running	0	37m	10.244.2.132	slave-0002	<none>	<none>
float-operation-74bd76d494-hn4sn	1/1	Running	0	25m	10.244.2.134	slave-0002	<none>	<none>
float-operation-74bd76d494-smr77	1/1	Running	0	25m	10.244.2.135	slave-0002	<none>	<none>
float-operation-7797686444-rqt86	0/1	Pending	0	3m50s	<none>	<none>	<none>	<none>
float-operation-79995f6b89-62d47	0/1	Pending	0	2s	<none>	<none>	<none>	<none>
float-operation-79995f6b89-kc8rx	0/1	Pending	0	3m50s	<none>	<none>	<none>	<none>
float-operation-79995f6b89-lb8dg	0/1	Pending	0	4m5s	<none>	<none>	<none>	<none>
float-operation-79995f6b89-m6wcr	0/1	Pending	0	2s	<none>	<none>	<none>	<none>
float-operation-79995f6b89-rjdcn	1/1	Running	0	4m11s	10.244.2.137	slave-0002	<none>	<none>
float-operation-79995f6b89-rpcfm	0/1	Pending	0	3m50s	<none>	<none>	<none>	<none>
float-operation-7b4b57dd4-c679w	0/1	Pending	0	2s	<none>	<none>	<none>	<none>
float-operation-7b6fc5b5b8d-lbnrf	1/1	Running	0	35m	10.244.1.165	slave-0001	<none>	<none>
float-operation-f794b7f58-wgg4p	0/1	Pending	0	38s	<none>	<none>	<none>	<none>

图 4-4-5

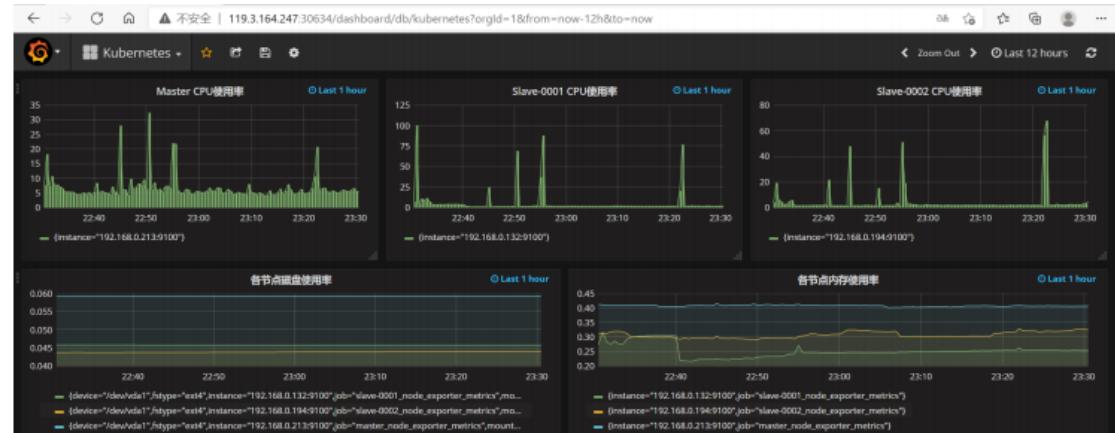


图 4-4-6

## 正文第二部分 知识点分析

### 五、知识点分析

#### 5.1 Kubernetes 架构、Master Node & Worker Node

##### 5.1.1 K8s 架构

Kubernetes 属于主从分布式架构，主要由 Master Node 和 Worker Node 组成，以及包括客户端命令行工具 kubectl 和其它附加项。

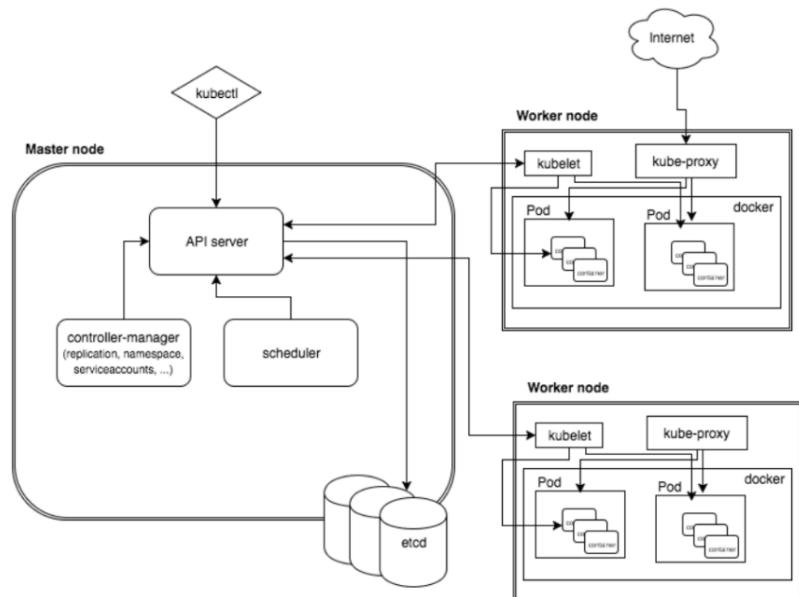


图 1-1\*

Master Node 作为控制节点，对集群进行调度管理；Master Node 由 API Server、Scheduler、Cluster State Store 和 Controller-Manger Server 所组成；

Worker Node 作为真正的工作节点，运行业务应用的容器；Worker Node 包含 kubelet、kube proxy 和 Container Runtime；

kubectl：用于通过命令行与 API Server 进行交互，而对 Kubernetes 进行操作，实现在集群中进行各种资源的增删改查等操作。

### 5.1.2 Master Node

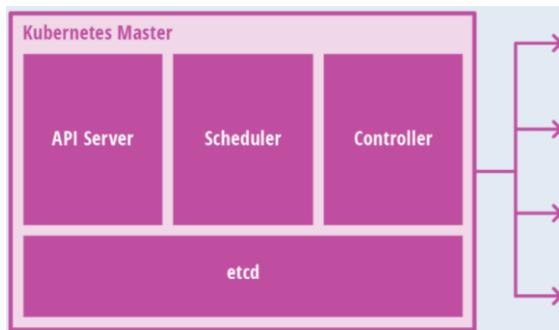


图 1-2\*

#### 1) API Server (API 服务器)

API Server 主要用来处理 REST 的操作，确保它们生效，并执行相关业务逻辑，以及更新 etcd (或者其他存储) 中的相关对象。

#### 2) Cluster state store (集群状态存储)

Kubernetes 默认使用 etcd 作为集群整体存储。集群的所有状态都存储在 etcd 实例中，并具有监控的能力，因此当 etcd 中的信息发生变化时，就能够快速的通知集群中相关的组件。

#### 3) Controller-Manager Server (控制管理服务器)

Controller-Manager Server 用于执行大部分的集群层次的功能，它既执行生命周期功能(例如：命名空间创建和生命周期、事件垃圾收集、已终止垃圾收集、级联删除垃圾收集、node 垃圾收集)，也执行 API 业务逻辑 (例如：pod 的弹性扩容)

#### 4) Scheduler (调度器)

Scheduler 组件为容器自动选择运行的主机。依据请求资源的可用性，服务请求的质量等约束条件，scheduler 监控未绑定的 pod，并将其绑定至特定的 node 节点。

### 5.1.3 Worker Node

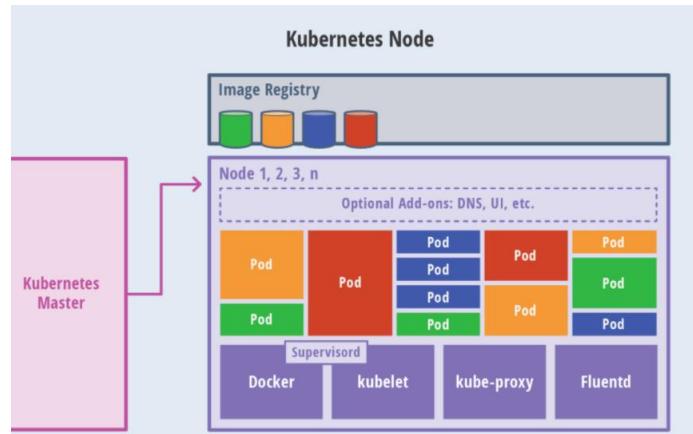


图 1-3\*

1) Kubelet

kubelet 负责管理 pods 和它们上面的容器，images 镜像、volumes、etc。

2) kube-proxy

kube proxy 负责为 Pod 创建代理服务；引到访问至服务；并实现服务到 Pod 的路由和转发，以及通过应用的负载均衡。每一个 Node 都会运行一个 kube-proxy，kube proxy 通过 iptables 规则引导访问至服务 IP，并将重定向至正确的后端应用，通过这种方式 kube-proxy 提供了一个高可用的负载均衡解决方案。

3) Container Runtime

每一个 Node 都会运行一个 Container Runtime，其负责下载镜像和运行容器。kubelet 使用 Unix socket 之上的 gRPC 框架与容器运行时进行通信，kubelet 作为客户端，而 CRI shim 作为服务器。

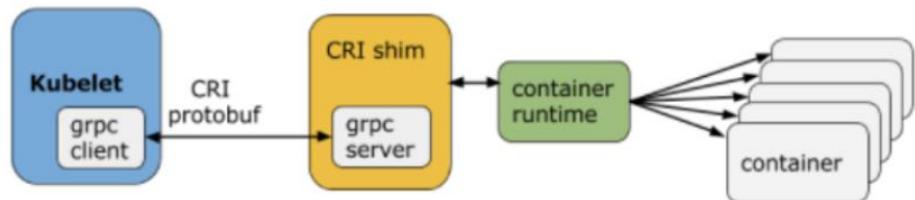


图 1-4\*

4) Fluentd-elasticsearch

提供集群日志采集、存储与查询

## 5.2 K8s 基础概念：Pod & Service & Controller

### 5.2.1 Pod

Pod 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。Pod 的设计理念是支持多个容器在一个 Pod 中共享网络地址和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务

### 5.2.2 服务 (Service)

一个 Pod 只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的 IP 启动一个新的 Pod，因此不能以确定的 IP 和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在 K8s 集群中，客户端需要访问的服务就是 Service 对象。每个 Service 会对应一个集群内部有效的虚拟 IP，集群内部通过虚拟 IP 访问一个服务。

### 5.2.3 控制器 (Controller)

Kubernetes 中内建了很多 controller (控制器)，这些相当于一个状态机，用来控制 Pod 的具体状态和行为。控制器又被称为工作负载，pod 通过控制器实现应用的运维，比如伸缩、升级等。

**Deployment:** 适合无状态的服务部署适合部署无状态的应用服务，用来管理 pod 和 replicaset，具有上线部署、副本设定、滚动更新、回滚等功能，还可提供声明式更新，例如只更新一个新的 Image。

**StatefullSet:** 适合有状态的服务部署适合部署有状态应用。解决 Pod 的独立生命周期，保持 Pod 启动顺序和唯一性。稳定，唯一的网络标识符，持久存储（例如：etcd 配置文件，节点地址发生变化，将无法使用）。

**DaemonSet:** 一次部署，所有的 node 节点都会部署例如一些典型的应用场景：运行集群存储 daemon，例如在每个 Node 上运行 glusterd、ceph；在每个 Node 上运行日志收集 daemon，例如 fluentd、logstash；在每个 Node 上运行监控 daemon，例如我们在实验中用到的 **Prometheus Node Exporter**。  
**Job:** 一次性的执行任务用来控制批处理型任务。Job 会创建一个或者多个 Pods，跟踪记录成功完成的 Pods 个数。当数量达到指定的成功个数阈值时，任务结束。删除 Job 的操作会清除所创建的全部 Pods。

**Cronjob:** 周期性的执行任务，不需要持续后台运行

DaemonSet 与 Deployment 的区别在于：Deployment 部署的副本 Pod 会分布在各个 Node 上，每个 Node 都可能运行好几个副本；DaemonSet 的不同之处在于：每个 Node 上最多只能运行一个副本。

## 5.3 OpenFaaS 层次框架 & 工作流程

### 5.3.1 层次框架

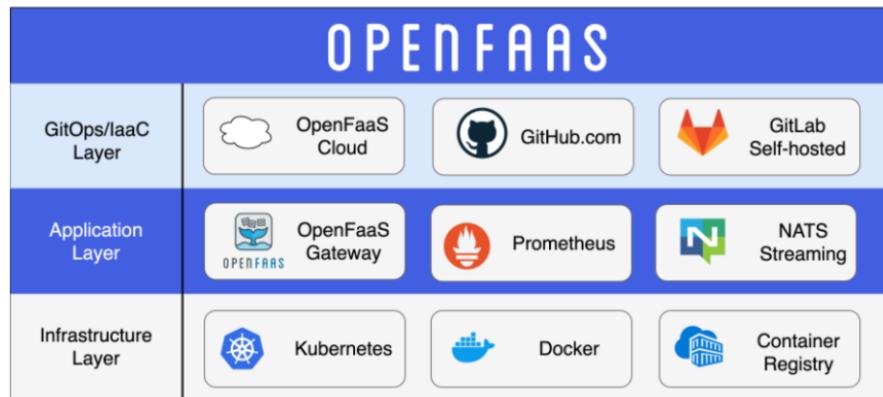


图 3-1\*

推荐的部署 OpenFaaS 平台都是 Kubernetes。OpenFaaS Cloud 建立在 OpenFaaS 的基础上，可通过 GitHub.com 或自行托管的 GitLab 交付。GitOpsNATS 提供异步执行和排队 Prometheus 提供指标并通过 AlertManager 启用自动缩放容器注册表包含每个可以通过 API 部署在 OpenFaaS 上的不可变构件。

### 5.3.2 总体工作流程示意

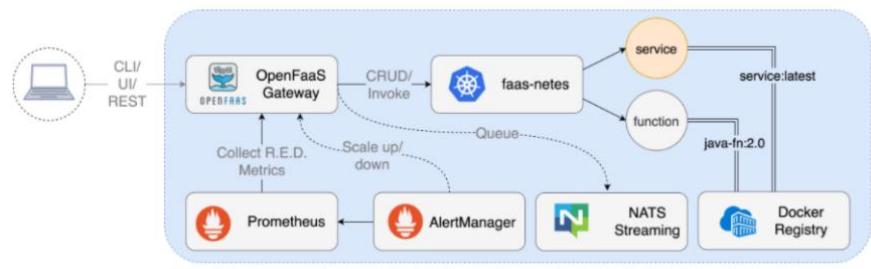


图 3-2\*

可以通过其 REST API, CLI 或 UI 来访问网关。所有服务或功能都会暴露默认路由，但是自定义域也可以用于每个端点。

### 5.3.3 Gateway

OpenFaaS 的 Gateway 是一个 golang 实现的请求转发的网关，在这个网关服务中，主要有以下几个功能：UI、部署函数、监控、自动伸缩。

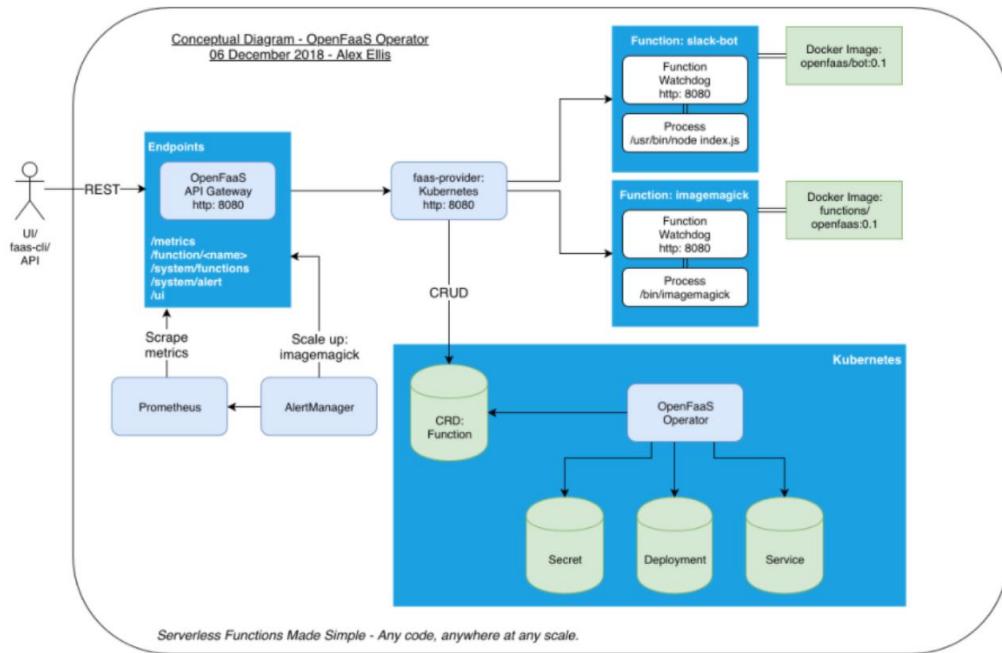


图 3-3\*

从图中可以发现，当 Gateway 作为一个入口，当 CLI 或者 web 页面发来要部署或者调用一个函数的时候，Gateway 会将请求转发给 Provider，同时会将监控指标发给 Prometheus。AlterManager 会根据需求，调用 API 自动伸缩函数。

Gateway 是 OpenFaaS 最为重要的一个组件。Gateway 本质上就是一个 rest 转发服务，一个一个的 handler，每个模块之间的耦合性不是很高，可以很容易的去拆卸，自定义实现相应的模块。

### 5.3.4 Watchdog & Of-Watchdog

watchdog 提供了一个外部世界和函数之间的非托管的通用接口。它的工作是收集从 API 网关来的 HTTP 请求，然后调用程序。watchdog 是一个小型的 Golang 服务。下图展示了它是如何工作的：

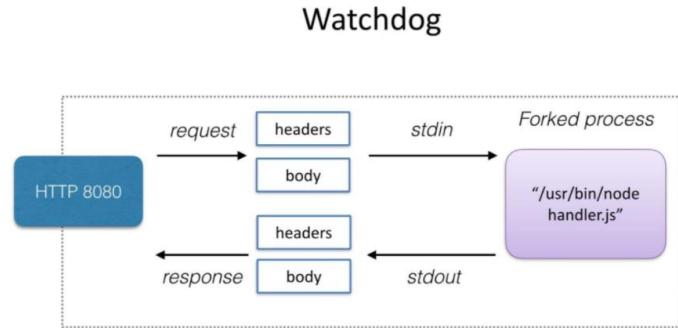


图 3-4\*

它相当于一个小型的 web 服务，可以为每个传入的 HTTP 请求分配所需要的进程。

每个函数都需要嵌入这个二进制文件并将其作为 ENTRYPOINT 或 CMD，实际上是把它作为容器的初始化进程。一旦进程被创建分支，watchdog 就会通过 stdin 传递 HTTP 请求并从 stdout 中读取 HTTP 响应。这意味着程序无需知道 web 和 HTTP 的任何信息。

of-watchdog 项目于 2017 年 10 月启动，该组件适合在生产中使用，并且是 openfaas GitHub 组织的一部分。of-watchdog 是函数和微服务之间的反向代理。

of-watchdog 的默认模式是 http 模式，部署服务时 of-watchdog 会 fork 一个进程（假设为进程 A），进程 A 会监听一个端口，of-watchdog 收到的所有请求都会转发到进程 A 监听的端口，HTTP 模式的官方架构图如下：

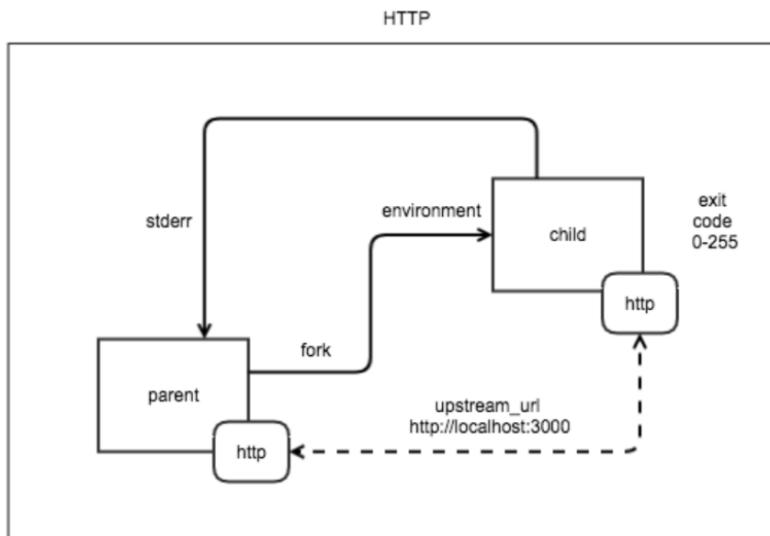


图 3-5\*

如图所示，代码是在右侧的 child 里面执行的，这个 child 从 3000 端口收到 of-watchdog 转发过来的外部请求，然后内部处理掉

Watchdog & 和 Of-Watchdog 的主要区别在于：经典的看门狗会为每个请求派生一个进程，以提供最高级别的可移植性，但是 of-watchdog 启用了一种 http 模式，在该模式下，该进程可以重复使用，以抵消分叉的延迟。

## 5.4 OpenFaaS 函数请求处理流程 & 源码解读

### 5.4.1 访问 openfaas 的 gateway 的 service。

对于一个函数请求而言，其由一个客户端发出，首先访问的是 openfaas 的 gateway 的 service，通过 iptables 将对于 Service 的访问路由到 pod 所连接的网卡。

如下图所示：

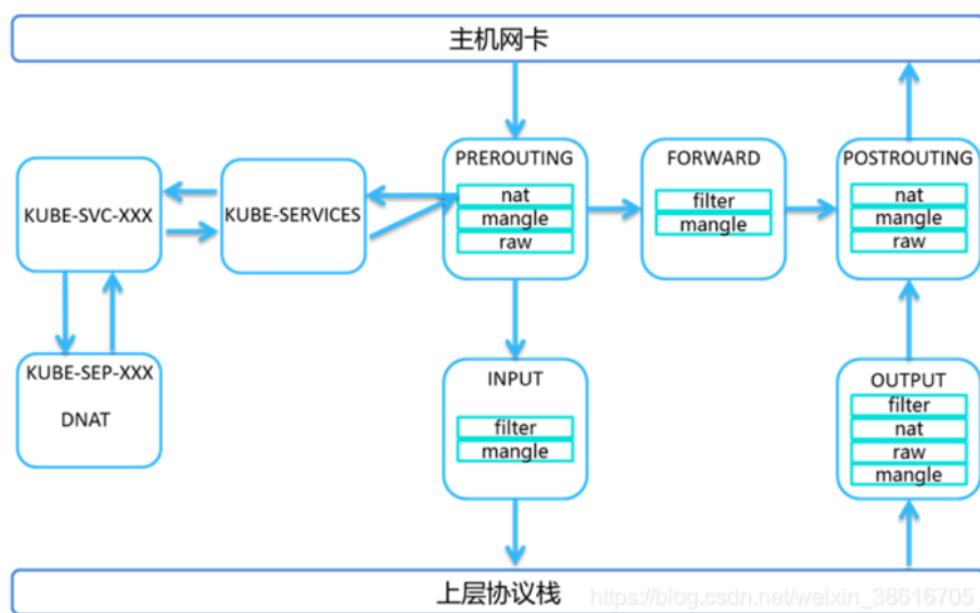


图 4-1\*

### 5.4.2 pod 的连接

在多节点集群中，请求进行 DNAT 之后会被转发到 flannel 网卡上，然后由一个 flanneld 的程序进行处理（也就是路由，flanneld 通过 etcd 储存了一个集群的网络信息，通过此进行路由），接着 flanneld 会根据路由选择的结果将请求转发到目的 pod 所在机器上，而在目的 pod 所在机器上，请求先是被转发到 flannel 网卡上

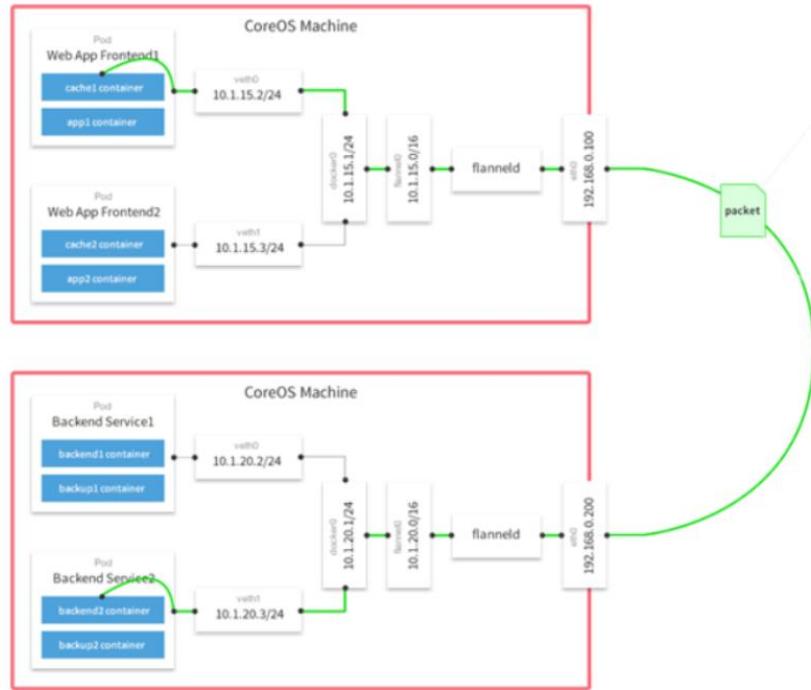


图 4-2\*

找到了相应的 pod 后，接下来需要发送给具体的容器。

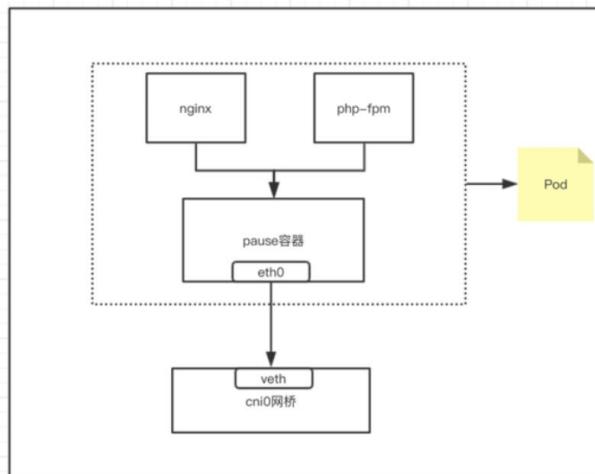


图 4-3\*

在每个 pod 之中，都有一个 pause 容器，这是 kubernetes 创建的系统容器，pause 容器会创建网络命名空间并将直接的网络桥接到 cni0 网桥（通过 veth pair），然后所有的用户容器都会共享这个网络命名空间，即共享一块网卡。那么请求转发的流程就是直接在 cni0 通过 veth pair 传向了 pod 内的 eth0，接着容器就可以直接接收网卡 eth0 上的信息了。

### 5.4.3 Gateway 中的请求处理

在 gateway 中存在处理函数请求的是 handler，下面将深入至 OpenFaaS 源码内部进行 Gateway 请求处理的分析

**代理转发：**Gateway 本身不做任何和部署发布函数的事情，它只是作为一个代理，把请求转发给相应的 Provider 去处理，所有的请求都要通过这个网关

#### 1) 同步函数转发

主要转发的 API 有：RoutelessProxy、ListFunctions、DeployFunction、DeleteFunction、UpdateFunction。

1. faasHandlers.RoutelessProxy = handlers.MakeForwardingProxyHandler(reverseProxy, fo  
rwardingNotifiers, urlResolver)
2. faasHandlers.ListFunctions = handlers.MakeForwardingProxyHandler(reverseProxy, for  
wardingNotifiers, urlResolver)
3. faasHandlers.DeployFunction = handlers.MakeForwardingProxyHandler(reverseProxy, f  
orwardingNotifiers, urlResolver)
4. faasHandlers.DeleteFunction = handlers.MakeForwardingProxyHandler(reverseProxy, fo  
rwardingNotifiers, urlResolver)
5. faasHandlers.UpdateFunction = handlers.MakeForwardingProxyHandler(reverseProxy, fo  
rwardingNotifiers, urlResolver)

MakeForwardingProxyHandler()的三个参数作用如下：

**Proxy:** 这是一个 http 的客户端，作者把这个客户端抽成一个类，然后使用该类的 NewHTTPClientReverseProxy 方法创建实例，这样就简化了代码，不用每次都得写一堆相同的配置。

**Notifiers:** 这个其实是要打印的日志，这里是一个 HTTPNotifier 的接口。而在这个 MakeForwardingProxyHandler 中其实有两个实现类，一个是 LoggingNotifier，一个是 PrometheusFunctionNotifier，分别用来打印和函数 http 请求相关的日志以及和 Prometheus 监控相关的日志。

**baseURLResolver:** 这个就是 Provider 的 url 地址。

MakeForwardingProxyHandler 中主要做了三件事：解析要转发的 url；调用 forwardRequest 方法转发请求；打印日志。

#### 2) 异步函数转发

如果是异步函数，Gateway 就作为一个发布者，将函数放到队列里。

MakeQueuedProxy 方法就是做这件事的：

- 1) 读取请求体
- 2) 将 X-Callback-Url 参数从参数中 http 的 header 中读出来
- 3) 实例化用于异步处理的 Request 对象
- 4) 调用 canQueueRequests.Queue(req)，将请求发布到队列中

#### 5.4.5. 函数 pod 内部的请求处理

接下来 gateway 就会把请求转发给具体的 function 所对应的 Service，这里 Service 的访问流程如前所述，但是也有一点需要注意，因为每个 function 都会对应多个 pod 副本，那么 Service 在进行 iptables 处理的时候就需要选择一个具体的 pod 来转发请求，这个时候就会有一个负载均衡的问题，而 Kubernetes 在这里的处理是随机选择一个 pod 进行转发。

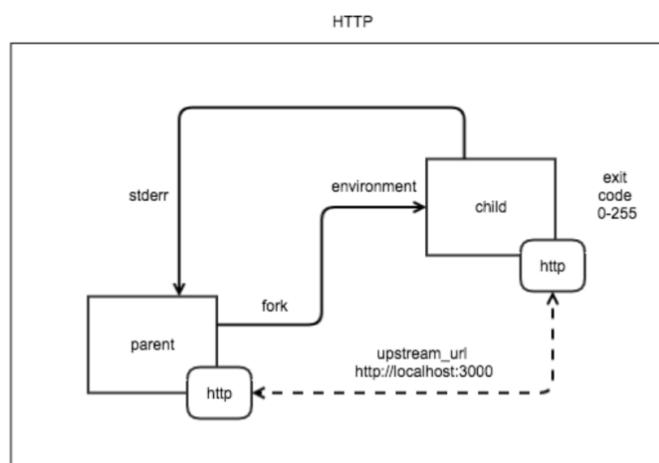


图 4-4\*

在 Http 的方式下当这个 watchdog 启动时，父进程会 fork 一个子进程（也就是用户提交的函数程序），这个函数程序不是简单的函数逻辑，也要包含一个 Http 服务器用来处理 Http 请求，且这个函数程序会一直运行，监听容器网卡上的端口以获得请求。

在我们的实验中，函数的部署通过 **of-watchdog** 实现。对这一知识点的了解使得我们在配置过程中通过对 **of-watchdog** 镜像换源解决了配置过程中遇到的错误。

## 5.5 Prometheus & PromQL

### 5.5.1 Prometheus 架构设计

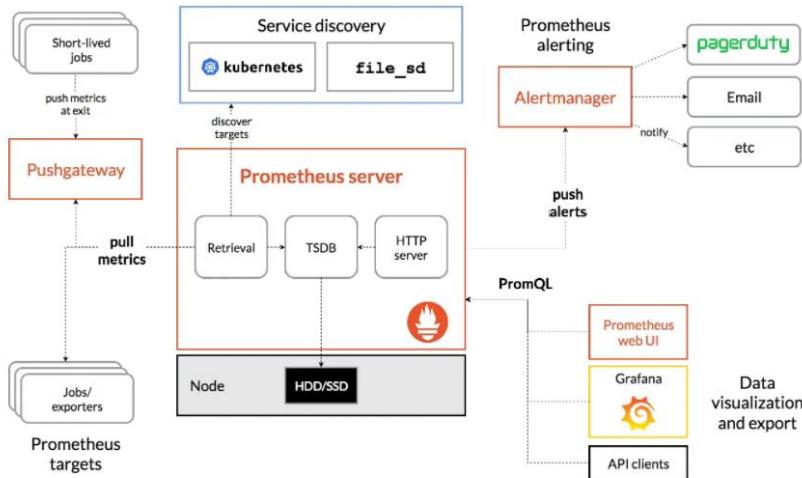


图 5-1\*

prometheus 生态系统由多个组件组成，其中许多组件是可选的。

1) **prometheus server:**

主要获取和存储时间序列数据，并且对外提供数据查询支持，需要定时从 Exporter 暴露的 HTTP 服务获取监控数据即可。

2) **exporters:**

主要是作为 agent 收集数据发送到 prometheus server，不同的数据收集由不同的 exporters 实现，如监控主机有 node-exporters，mysql 有 MySQL server exporters。在本次实验中，主要使用 node exporter 获取到所在主机大量的运行数据，典型的包括 CPU、内存，磁盘、网络等等监控样本。

3) **pushgateway:**

允许短暂和批处理的 jobs 推送它们的数据到 prometheus；由于这类工作的存在时间不够长，所以需要他们主动将数据推送到 pushgateway，然后由 pushgateway 将数据发送的 prometheus。

4) **alertmanager:**

实现 prometheus 的告警功能。

5) **Prometheus UI:**

提供了快速验证 PromQL 以及临时可视化支持的能力，而在大多数场景下引入监控系统通常还需要构建可以长期使用的监控数据可视化面板（Dashboard）。这时用户可以考虑使用第三方的可视化工具如 Grafana，Grafana 是一个开源的可视化平台，并且提供了对 Prometheus 的完整支持。

### 5.5.2 PromQL

Prometheus 通过指标名称 (metrics name) 以及对应的一组标签 (labelset) 唯一定义一条时间序列。指标名称反映了监控样本的基本标识，而 label 则在这个基本特征上为采集到的数据提供了多种特征维度。用户可以基于这些特征维度过滤，聚合，统计从而产生新的计算后的一条时间序列。

在我们的实验中，我们使用了下列查询语句用于 **Prometheus** 平台的搭建：

各节点磁盘使用率：

```
node_filesystem_size_bytes{fstype=~"xfs|ext4"} - node_filesystem_free_bytes  
node_filesystem_size_bytes
```

各节点内存使用率：

```
node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes) /  
node_memory_MemTotal_bytes
```

各节点磁盘 IO 读速率(MiB/s)：

```
irate(node_disk_read_bytes_total{}[5m]) / 1024 / 1024
```

各节点磁盘 IO 写速率(MiB/s)：

```
irate(node_disk_written_bytes_total{}[5m]) / 1024 / 1024
```

某节点网络 IO 速率:入速率(MiB/s)：

```
irate(node_network_receive_bytes_total{job="节点名称  
_node_exporter_metrics"}[5m]) / 1024 / 1024
```

某节点网络 IO 速率:出速率(MiB/s)：

```
irate(node_network_transmit_bytes_total{job="节点名称  
_node_exporter_metrics"}[5m]) / 1024 / 1024
```

某节点 CPU 使用率：

```
100 - (avg by(instance) (irate(node_cpu_seconds_total{mode="idle", job="节点名称  
_node_exporter_metrics"}[5m])) * 100)
```

## 5.6 OpenFaaS 自动伸缩原理 & 源码解读

### 5.6.1 自动伸缩指标需求

自动伸缩是 OpenFaaS 的一大特点，触发自动伸缩主要是根据不同的指标需求。OpenFaaS 附带了一个自动伸缩的规则，这个规则是在 AlertManager 配置文件中定义。AlertManager 从 Prometheus 中读取使用情况（每秒请求数），然后在满足一定条件时向 Gateway 发送警报。可以通过删除 AlertManager，或者将部署扩展的环境变量设置为 0，来禁用此方式。此外，还可以通过向函数添加标签，在部署时设置最小（初始）和最大副本数。

自动伸缩相关参数如下：

com.openfaas.scale.min 默认是 1，代表最小的实例数

com.openfaas.scale.max 默认是 20，代表最大实例数

com.openfaas.scale.factor 默认是 20%，这是每次扩容的时候，新增实例的百分比，若是 100% 的话，会瞬间飙升到副本数的最大值。

## 5.6.2 scaleService 函数

Gateway 的源码中，scaleService 是真正处理伸缩服务的函数：

```
1. func scaleService(alert requests.PrometheusInnerAlert, service ServiceQuery) error {
2.     var err error
3.     serviceName := alert.Labels.FunctionName
4.
5.     if len(serviceName) > 0 {
6.         queryResponse, getErr := service.GetReplicas(serviceName)
7.         if getErr == nil {
8.             status := alert.Status
9.
10.            newReplicas := CalculateReplicas(status, queryResponse.Replicas, uint64(queryResponse.MaxReplicas), queryResponse.MinReplicas, queryResponse.ScalingFactor)
11.
12.            log.Printf("[Scale] function=%s %d => %d.\n", serviceName, queryResponse.Replicas, newReplicas)
13.            if newReplicas == queryResponse.Replicas {
14.                return nil
15.            }
16.
17.            updateErr := service.SetReplicas(serviceName, newReplicas)
18.            if updateErr != nil {
19.                err = updateErr
20.            }
21.        }
22.    }
23.    return err
24. }
```

从代码中可以看到，scaleService 做了三件事：

获取现在的副本数；

计算新的副本数：新副本数的计算方法是根据 com.openfaas.scale.factor 计算步长，`step := uint64((float64(maxReplicas) / 100) * float64(scalingFactor))`；

设置为新的副本数。

### 5.6.3 MakeScalingHandler 函数

在调用函数的时候，用的路由是：/function/:name。如果环境变量里有配置 scale\_from\_zero 为 true，先用 MakeScalingHandler()方法对 proxyHandler 进行一次包装。

MakeScalingHandler 接受参数主要是：

next: 就是下一个 httpHandlerFunc, 中间件都会有这样一个参数

config: ScalingConfig 的对象:

源码如下：

- ```
•    // ScalingConfig for scaling behaviours
•    type ScalingConfig struct {
•        MaxPollCount uint // 查到的最大数量
•        FunctionPollInterval time.Duration // 函数调用时间间隔
•        CacheExpiry time.Duration // 缓存过期时间
•        ServiceQuery ServiceQuery // 外部服务调用的一个接口
•    }
```

这个 MakeScalingHandler 中间件主要做了如下的事情：先从 FunctionCache 缓存中获取该函数的基本信息，从这个缓存可以拿到每个函数的副本数量。为了加快函数的启动速度，如果缓存中可以获该得函数，且函数的副本数大于 0，满足条件，return 即可。如果不满足上一步，就会调用 SetReplicas 方法设置副本数，并更新 FunctionCache 的缓存。

- ```
•     // MakeScalingHandler creates handler which can scale a function from
•     // zero to 1 replica(s).
•
•     func MakeScalingHandler(next http.HandlerFunc, upstream http.HandlerFunc, config ScalingConfig) http.HandlerFunc {
•         cache := FunctionCache{
•             Cache: make(map[string]*FunctionMeta),
•             Expiry: config.CacheExpiry,
•         }
•
•         return func(w http.ResponseWriter, r *http.Request) {
•             functionName := getServiceName(r.URL.String())
•
•             if serviceQueryResponse, hit := cache.Get(functionName); hit && serviceQueryResponse.AvailableReplicas > 0 {
•                 next.ServeHTTP(w, r)
•             }
•             return
•         }
•     }
• }
```

```

    }
    queryResponse, err := config.ServiceQuery.GetReplicas(functionName)
    cache.Set(functionName, queryResponse)
    // 省略错误处理
    if queryResponse.AvailableReplicas == 0 {
        minReplicas := uint64(1)
        if queryResponse.MinReplicas > 0 {
            minReplicas = queryResponse.MinReplicas
        }
        err := config.ServiceQuery.SetReplicas(functionName, minReplicas)
        // 省略错误处理代码
        for i := 0; i < int(config.MaxPollCount); i++ {
            queryResponse, err := config.ServiceQuery.GetReplicas(functionName)
            cache.Set(functionName, queryResponse)
            // 省略错误处理
            time.Sleep(config.FunctionPollInterval)
        }
    }
    next.ServeHTTP(w, r)
}
}

```

在我们的实验中根据本知识点，禁用自动缩放后手动设置实例数来实现资源限制实验。

## 5.7 K8s 集群资源限制

资源限制是用户可以向 Kubernetes 提供的诸多配置之一，它意味着两点：工作负载运行需要哪些资源；最多允许消费多少资源。第一点对于调度器而言十分重要，因为它要以此选择合适的节点。第二点对于 Kubelet 非常重要，每个节点上的守护进程 Kubelet 负责 Pod 的运行健康状态。

资源限制是通过每个容器 containerSpec 的 resources 字段进行设置的，它是 v1 版本的 ResourceRequirements 类型的 API 对象。每个指定了"limits"和"requests"的对象都可以控制对应的资源。目前只有 CPU 和内存两种资源。

例如，在 yaml 文件中：

```

resources:
  requests:
    cpu: 50m
    memory: 50Mi
  limits:
    cpu: 100m
    memory: 100Mi

```

图 7-1\*

这个容器通常情况下，需要 5% 的 CPU 时间和 50MiB 的内存（requests），同时最多允许它使用 10% 的 CPU 时间和 100MiB 的内存（limits）。我会对 requests 和 limits 的区别做进一步讲解，但是一般来说，在调度的时候 requests 比较重要，在运行时 limits 比较重要。

在我们的实验中，依据具体设置的资源限制数额，实例创建时 requests 指定的系统资源可能会超过了实际可使用的资源余量，导致该镜像并没有正常运行。

## 5.8 k8s 任务调度

调度器 Scheduler 是 Kubernetes 的重要组件之一。其作用是要将待调度的 Pod 依据某调度策略调度到最适合它运行的节点上运行。这里涉及到三个对象：待调度的 Pod、调度策略、待部署的节点队列。

kube-scheduler 的整个调度流程分为两个阶段：预选策略（Predicates）和优选策略（Priorities）。

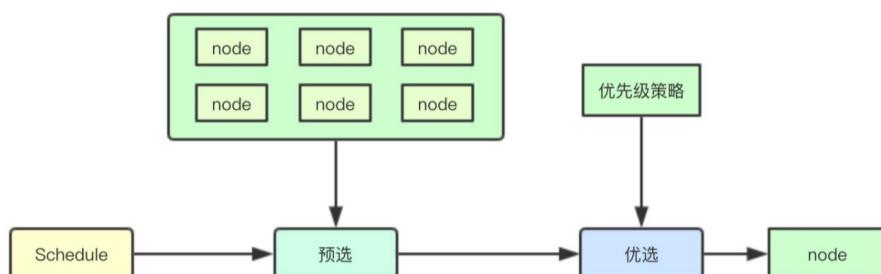


图 8-1\*

预选阶段（Predicates）：输入是所有节点，输出是满足预选条件的节点。kube-scheduler 根据预选策略过滤掉不满足策略的 Nodes。。

优选阶段（Priorities）：输入是预选阶段筛选出的节点，优选会根据优先策略为通过预选的 Nodes 进行打分排名，选择得分最高的 Node。资源越富裕、负载越小的 Node 可能具有越高的排名。

实际上，调度的过程就是在回答两个问题：候选有哪些？其中最适合的是哪个？值得一提的是，如果在预选阶段没有节点满足条件，Pod 会一直在 Pending 状态直到出现满足的节点，在此期间调度器会不断的进行重试。这一现象在我们的实验过程中经常出现，在资源限制实验部分，经常可以看到大量处于 Pending 状态的 pod。

调度器相当于一个黑盒子，输入待调度的 Pod 及待部署的节点队列，则会输出一个 Pod 与某节点的绑定信息。也就是说，经过调度器的调度，一个 Pod 就会与一个特定的节点相关联。接下来这个绑定信息则会通过 APIServer 这个入口组件输入给 Etcd 存储起来。Etcd 是一个强一致、高可用的服务发现存储仓库，里面存储了 Kubernetes 的许多重要信息。接下来与该待调度 Pod 绑定的节点上的

Kubelet 会通过 APIServer 监听 Etcd，发现自己所在节点被绑定了，那么就会依据 Pod 的信息开始进行容器在该节点的创建工作。

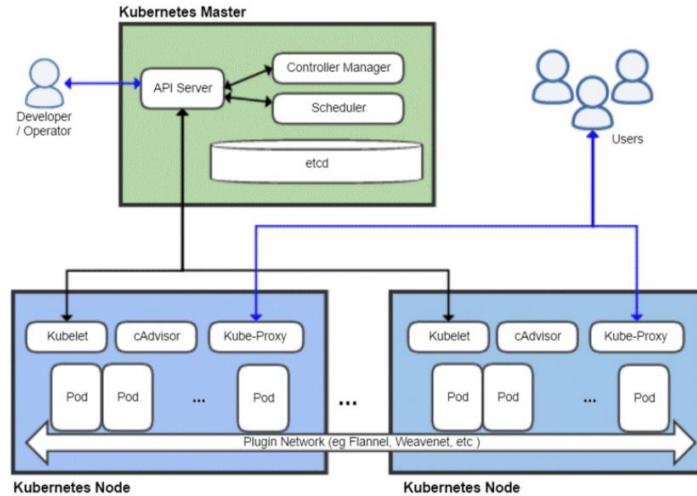


图 8-2\*

## 5.9 压力测试

压力测试指的是在对服务器资源进行一定的限制条件下，逐步增加访问压力，在此过程中观察服务器的行为变化，从而分析并找出性能瓶颈和威胁服务器稳定运行的因素。Wrk 是一款简单的 http 开源压力测试工具，可以利用多线程模拟高并发的访问情况。在使用过程中可以通过参数指定压力测试使用的线程数、并发连接数以及测试持续时间。工具将在测试结束后自动地给出常用压力测试指标的数据统计结果，如延迟分布情况、平均延迟、延迟的标准差等等。

在我们的实验中，压力测试主要使用了延迟作为服务器性能的度量指标，我们通过进行资源限制、实例数调整、负载均衡调整观察了不同参数规模、并发数下的服务器运行状态。并得出了提高服务器性能的合理建议。

就我们的体验而言，压力测试的最主要作用在于保证系统在各类情况下的稳定性，找出隐蔽的性能瓶颈。这些性能瓶颈可能并非我们直觉所想的 CPU、内存空闲情况，还可能是网络传输速率、线程池大小、CPU 的核数、负载均衡情况等等。

此外，在压力测试中需要尤其注意的是，相关参数的设置不可盲目设置。例如线程数的设置一般为发起测试主机 CPU 核数的整数倍，不可过大。因为整个测试进程同一时刻在操作系统中只会有一个 running 状态的线程，**线程数过大只会导致进程频繁的进行线程上下文切换**，导致测试主机压力下降，测试结果出现偏差。

## 5.10 阿姆达尔定律

G.M.Amdahl 在 1967 年提出了 Amdahl's law，针对并行处理的 scalability 给出了一个模型，指出使用并行处理的提速由问题的可并行的部分所决定。这个模型为并行计算系统的设计者提供了指导。其形式如下：

$$Speedup_{Amdahl} = \frac{1}{(1-f) + \frac{f}{m}}$$

其中， $f$  为问题中可被并行处理的部分的比例， $m$  为并行处理机的数量， $Speedup$  为并行后相比串行时的提速。

在本实验中，并发的函数访问分别被 openfaas 平台上运行于不同节点的各个实例进行处理。由于每个实例在操作系统层面的本质为一个进程，所以多个实例处理并发的访问本质上是一种**基于进程的任务级并行计算过程**。由于在我们的实验中，并发数大于实例数且任务均为计算密集型任务，仅需要极少的 IO 和网络传输时延，故整个任务中可以被并行处理的部分近似等于 1。

根据阿姆达尔定律，当  $f$  取 1 时，加速比近似等于总的 CPU 核数。

根据 4.3 中的实验结果，加速比最大为 6.3，这与阿姆达尔定律相符。

**最后，各知识点和实验部分的联系总结如下：**

K8s 架构和基础概念（知识点 1-3）相关知识点贯穿于整个实验过程中，Prometheus 搭建、函数部署等均涉及这部分知识。（正文第一部分第三节第 1 点、第 2 点）

OpenFaaS 架构和请求流程（知识点 4）的相关知识主要在函数服务部署过程中体现。（正文第一部分第三节第 5 点、第 6 点，第四节）

prometheus 监控（知识点 5、6）相关知识主要在 Prometheus + Grafana 监控平台的搭建中涉及。（正文第一部分第三节第 4 点）

OpenFaaS 自动缩放，调度，资源限制（知识点 7、8、9）主要在单实例/多实例的资源限制实验中涉及。（正文第一部分第三节、第四节）

## 小组分工

	张庆阳	董威龙	丁若萌
Week1	调研服务器测试指标与方法	调研 Serverless 计算模型及其优缺点	学习 OpenFaaS 部署流程及函数编写
Week2	完成 OpenFaaS 平台部署		学习 OpenFaaS 平台函数开发与性能检测方法
Week3	完成第二个 OpenFaaS 平台的搭建, 记录过程	安装并配置 node-exporter, Prometheus 和 Grafana, 搭建监控平台	
Week4	完成跨区域的 K8s 集群搭建; 确定性能测试变量	在 OpenFaaS 平台上完成了浮点/矩阵函数的相关测试	完成跨区域的 K8s 集群搭建; 配置 grafana 仪表盘
Week5	编写函数, 更改参数规模、线程数进行压力测试; 资源限制测试		知识点撰写、数据图表处理
Week6	实验内容汇总, 撰写大作业报告		

## 心得体会与收获

本次实验中，本小组成员协力合作，接力推进，完成了 openfaas 搭建和性能测试任务。在整个过程中，由于首次接触 openfaas、k8s 集群、压力测试相关内容，我们遇到了很多困难，也收获了许多教训和经验。

实验体会方面：在 Prometheus 配置过程中，由于我们使用了跨区域的节点连接，k8s 中 node-exporter 和 Prometheus 服务状态均为正常的 running 状态，但是通过指定端口却无法访问对应的服务。没有任何 debug 信息的情况下，我们的实验一度陷入停滞状态。最后，我们仔细分析配置流程，向华为在线工程师进行了咨询，最终将问题锁定在 vpc 网络配置上，最终解决了 bug。

在压力测试过程中，由于没有深入了解压力测试工具参数的含义和 k8s 实例调度算法就贸然开展实验，导致结果与预期严重不符，且不合理的参数一度导致集群的崩溃。最后，我们查阅 wrk 测试原理和 k8s 调度，借助 Prometheus 平台监控的硬件使用情况，将问题锁定在线程数过大和集群负载不均衡上，通过修改实验参数成功的解决了问题，并得到了与预期相符的实验数据，这令我们获得了巨大的成就感。

实验收获方面：首先，我们认识到 openfaas 函数式编程与传统的 CS 架构有着根本的不同，这种 Function as service 的云计算服务模式可以根据实际的并发情况进行动态的资源调度和负载均衡。例如在我们的实验中，可以通过 Prometheus 监控与 alert 组件连接，利用 openfaas 的 auto-scale 机制进行实例数量的自动调整。这使得业务在访问高峰期可以有更低的访问延迟、更高的 qps，在访问低谷期可以将计算资源回收到云平台中，与传统服务器架构相比更加灵活可靠。与此同时，k8s 的任务调度算法也会根据各节点的系统资源使用情况进行负载均衡。正如老师所言：serverless 架构确实代表着云计算未来十年的发展方向。

此外，在整个实验中，我们深刻认识到操作系统、体系结构、数据结构、计算机网络等专业基础课的重要性。尽管 openfaas 函数式服务技术日新月异，但是其底层的许多设计和调度算法依然可以从操作系统课程中窥见其思想本源。K8s 的调度算法设计原则体现了体系结构中“通过冗余提高可靠性”、“通过并行提高性能”等多种经典的设计思想，而最终我们的实验数据也和并行计算中的阿姆达尔定律相符合。这些经典的理论和设计思想值得我在今后的研究学习中继续深入理解和体会。

最后，感谢老师和助教同学在整个实验过程中给予的帮助。