实验八：云上的应用开发、部署和运维

实验目的

实验内容

实验操作步骤

一、云应用开发

二、Dockerfile制作镜像

0、安装Docker

1、 编译出可执行文件(在此以go语言为例)

2、 编写Dockerfile

3、 使用Dockerfile制作镜像

4、 查看制作成功的镜像

5、测试镜像是否能够正常使用

三、容器化部署于Kubernetes平台

1、在进行本步骤前，可选择阅读并了解Kubernetes，以及其控制器模式、声明式API的概念，可参考课...

2、使用Deployment控制器管理服务，于是编写Deployment的YAML配置文件如下：

3、使用kubelet命令工具的kubelet apply命令部署Deployment：

4、使用kubelet get/ kubelet describe查看应用部署状态：

5、常见问题可查于：

四、使用Service实现服务发现

1、 在进行本步骤前，可选择阅读并了解Service的概念，可参考：

2、 编写Service的YAML配置文件如下：

3、 使用kubelet apply部署该Service：

5、 使用Cluster IP，测试该Service的可用性

6、 查看Service所有的endpoints

五、横向扩缩容

六、Prometheus监控平台的使用

0、 在进行本步骤前，需阅读并了解prometheus、Exporter、Counter指标、Hisogram指标等概念。可...

1、在集群中部署prometheus

2、 修改业务逻辑代码，增加prometheus Exporter

3、本地测试Exporter是否编写成功

4、“发布”新的版本（带有metrics的版本）

5、测试通过service访问 exporter是否正常

7、查看prometheus配置文件中配置的target

8、浏览器直接打开prometheus

参考材料：

实验目的

1、熟悉何为云原生应用
2、熟悉云原生应用的生命周期：开发、部署、版本更新、服务注册、服务发现、扩缩容、监控。
3、熟悉Dockerfile的编写
4、熟悉Kubernetes容器平台的基本使用及其基本概念
5、熟悉Service的概念，了解其原理及使用
6、了解扩缩容的概念
7、了解Prometheus监控平台，学会编写简单的Exporter，学会采集Exporter的监控指标。

实验内容

本周的实验需要完成一个云上服务的开发部署及运维。

1、首先自行编写云服务业务逻辑（有Golang示例）
2、编写Dockerfile制作镜像
3、使应用容器化部署到Kubernetes平台上
4、制作Service对象实现服务注册与服务发现
5、使用HPA组件完成服务的横向或扩缩容
6、在服务内编写prometheus的Exporter，实现新版本发布，最终完成监控指标的抓取。

实验操作步骤

一、云应用开发

该步骤同学可使用任何编程语言，自行开发任意功能的云服务，业务逻辑不需特别复杂。
实验目的不在于学习开发，重点是学习后续的Docker工具、Kubernetes平台的使用以及基于二者之上的运维工作的流程和步骤。
Golang语言编写的示例：

实验均以https://github.com/tjrone/example为例，该仓库内包含示例的所有相关内容，包括代码、配置文件等。

制作镜像前，先测试在本地运行情况

二、Dockerfile制作镜像

在进行本步骤前，需提前阅读并了解容器、镜像等概念，可参考课件、和后续的参考资料

0、安装Docker

具体安装步骤：https://www.runoob.com/docker/ubuntu-docker-install.html

1、 编译出可执行文件(在此以go语言为例)

```
root@master:~/example/without_metrics# go build -o example
root@master:~/example/without_metrics# ls
example  main.go
```

2、 编写Dockerfile

2.1 设置基础镜像

```
1 FROM ubuntu
```

2.2 将可执行文件放入镜像中（此处可以这样做的前提是静态链接，无需在运行中动态链接）
根据自己的Dockfile和可执行文件的相对位置来写，这里只是示例

```
2 ADD example /
```

2.3 设置环境变量（后来实验能看出效果）

```
3 ENV Num 10
```

2.4 设置暴露的端口

```
4 EXPOSE 5565
```

2.5 设置容器内执行的命令

```
5 ENTRYPOINT ["./example"]
```

3、 使用Dockerfile制作镜像

使用docker build命令：
–t 为镜像命名 –t name：tag(version)

```
root@master:~/example/without_metrics# docker build -t example:latest .
Sending build context to Docker daemon  7.459MB
Step 1/5 : FROM ubuntu
 ---> ccc6e87d482b
Step 2/5 : ADD example /
 ---> bdcf4e9934d0
Step 3/5 : ENV Num 10
 ---> Running in f2f1ae7371d4
Removing intermediate container f2f1ae7371d4
 ---> 62e7023dd6db
Step 4/5 : EXPOSE 5565
 ---> Running in de10681fb41f
Removing intermediate container de10681fb41f
 ---> 64d4b95f397f
Step 5/5 : ENTRYPOINT ["./example"]
 ---> Running in f8244bb3fd9e
Removing intermediate container f8244bb3fd9e
 ---> db2ee681cf16
Successfully built db2ee681cf16
Successfully tagged example:latest
```

4、 查看制作成功的镜像

```
root@master:~/example/deployment# docker image ls
REPOSITORY                                  TAG                 IMAGE ID            CREATED             SIZE
example                                     latest              fa247d8fa2c3        9 minutes ago       8.67MB
```

5、测试镜像是否能够正常使用

```
root@master:~/example/without_metrics# docker run -itd -p 33333:5565 example:latest
25ae2002a9d9622d71d68c0be4e8cadd22c26c96cef78361905a2869bc1ca5ba
```

-p 33333:9090 即把主机的33333端口映射到容器的9090端口

测试结果：可反映环境变量生效

```
root@master:~/example/without_metrics# curl 127.0.0.1:33333/abc
there is env Num. Computation successed
```

三、容器化部署于Kubernetes平台

1、在进行本步骤前，可选择阅读并了解Kubernetes，以及其控制器模式、声明式API的概念，可参考课件。

2、使用Deployment控制器管理服务，于是编写Deployment的YAML配置文件如下：

```
1 apiVersion: apps/v1                    //此处为Deployment对象的版本信息
```

```
kind: Deployment
metadata:
  name: example-service        //Deployment的名称
spec:
  replicas: 2                  //副本数，即Pod的个数
  selector:
    matchLabels:
      app: example             //即控制标记为 “ app: example ” 的Pod
  template:
    metadata:
      labels:
        app: example           //标记Pod
    spec:
      containers:
        - name: back-end             //容器名
          image: example                      //容器使用的镜像
          imagePullPolicy: Never         //由于本地已使用Dockerfile制作好镜像，于是不需下载
          ports:
            - containerPort: 5565      //
```

3、使用kubelet命令工具的kubelet apply命令部署Deployment：

```
root@master:~/example/deploy# kubectl apply -f deployment.yaml
deployment.apps/example-service created
```

4、使用kubelet get/ kubelet describe查看应用部署状态：

```
root@master:~/example/deploy# kubectl get deployments example-service
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
example-service   2/2     2            2           2m52s
```

```
root@master:~/example/deploy# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
coffee-6cbd8b965c-8dj9d             1/1     Running   0          43d
coffee-6cbd8b965c-dk4mn             1/1     Running   0          43d
dns-test                            1/1     Running   0          71d
etcd-operator-85f7494fcf-95qv4      1/1     Running   14         75d
example-service-cb57cc5c4-bp4cm     1/1     Running   0          4s
example-service-cb57cc5c4-bz92f     1/1     Running   0          4s
hamster-6d7dd457c6-sgplq            1/1     Running   1          88d
hamster-6d7dd457c6-wdjrs            1/1     Running   1          88d
tea-588dbb89d5-9ggtn                1/1     Running   0          43d
tea-588dbb89d5-bcvrq                1/1     Running   0          43d
tea-588dbb89d5-vt8mt                1/1     Running   0          43d
web-0                               1/1     Running   0          72d
web-1                               1/1     Running   0          72d
```

5、常见问题可查于：

1、镜像拉取失败：集群有多台节点，检查调度所至节点上，是否已经存在所需镜像

```
root@master:~/example/deploy# kubectl get pod
NAME                                READY   STATUS             RESTARTS   AGE
coffee-6cbd8b965c-8dj9d             1/1     Running            0          43d
coffee-6cbd8b965c-dk4mn             1/1     Running            0          43d
dns-test                            1/1     Running            0          71d
etcd-operator-85f7494fcf-95qv4      1/1     Running            14         75d
example-service-7b89f56675-vp5q4    0/1     ImagePullBackOff   0          43s
example-service-7b89f56675-wbbtj    0/1     ImagePullBackOff   0          43s
hamster-6d7dd457c6-sgplq            1/1     Running            1          88d
hamster-6d7dd457c6-wdjrs            1/1     Running            1          88d
```

2、kubectl apply 失败：检查yaml格式及其内容是否正确

```
root@master:~/example/deploy# kubectl apply -f deployment.yaml
error: error validating "deployment.yaml": error validating data: ValidationError(Deployment.metadata): unknown field "spec"
```

3、出现Crash，查看原因发现：

```
1 exec user process caused "no such file or directory
```

基础镜像问题，建议使用Ubuntu作为基础镜像。

四、使用Service实现服务发现

1、 在进行本步骤前，可选择阅读并了解Service的概念，可参考：

📎 37 找到容器不容易：Service、DNS与服务发现.html

2、 编写Service的YAML配置文件如下：

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
  - name: default
    protocol: TCP
    port: 80
    targetPort: 5565
```

3、 使用kubelet apply部署该Service：

```
root@master:~/example/deploy# kubectl apply -f service.yaml
service/example-service created
```

4、 查看该Service，并查看Kubernetes为Service分配的Cluster IP

```
root@master:~/example/without_metrics# kubectl get svc
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
example-service   ClusterIP   10.111.101.163   <none>        80/TCP           100m
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP          105d
nginx             ClusterIP   None             <none>        80/TCP           75d
prometheus        NodePort    10.107.190.103   <none>        9090:32126/TCP   3d2h
```

5、 使用Cluster IP，测试该Service的可用性

```
root@master:~/example/deploy# curl 10.111.101.163/abc
there is env Num. Computation successed
```

6、 查看Service所有的endpoints

```
root@master:~/example/deploy# kubectl get endpoints example-service
NAME              ENDPOINTS                                   AGE
example-service   192.168.1.90:5565,192.168.1.91:5565   103m
```

五、横向扩缩容

横向pod 自动伸缩是指由控制器管理的pod 副本数量的自动伸缩。它由Horizontal 控制器执行，我们通过创建一个HorizontalpodAutoscaler（ HPA ）资源来启用和配置Horizontal 控制器。该控制器周期性检查pod，计算满足HPA资源所配置的目标数值所需的副本数量，进而调整目标资源（如Deployment 、ReplicaSet 、ReplicationController 、StatefulSet 等）的replicas 字段。

自动伸缩的过程可以分为三个步骤：

1）获取被伸缩资源对象所管理的所有pod 度量。

2）计算使度量数值到达（或接近）所指定目标数值所需的pod 数量。

3）更新被伸缩资源的replicas 字段。

基于CPU使用率的自动扩缩容介绍：

我们现在来看看如何创建一个HPA ，并让它基于CPU 使用率来伸缩pod 。你将创建一个Deployment。但正如我们讨论的，你需要确保Deployment所创建的所有pod 都指定了CPU 资源请求，这样才有可能实现自动伸缩。你需要给Deployment 的pod 模板添加一个CPU 资源请求，如以下代码所示。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kubia
spec:
  replicas: 3
  template:
    metadata:
      name: kubia
      labels:
        app: kubia
    spec:
      containers:
      - image: luksa/kubia:v1
        name: nodejs
        resources:
          requests:
            cpu: 100m
```

手动设置（初始）想要的副本数为 3

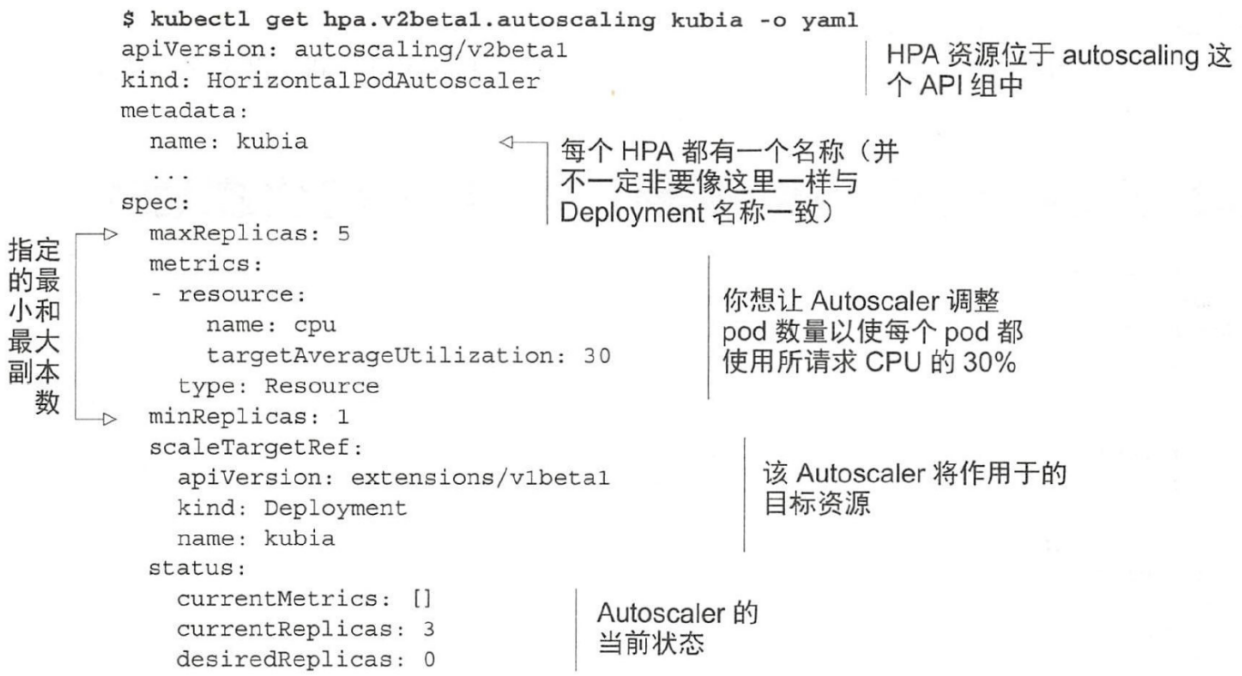
运行 kubia:v1 镜像

每个 pod 请求 100 毫核的 CPU

这就是一个正常的Deployment 对象——现在还没有启用自动伸缩。它会运行3个实例的kubia NodeJS 应用，每个实例请求100 毫核的CPU 。创建了Deployment 之后，为了给它的pod 启用横向自动伸缩， 需要创建一个HorizontalpodAutoscaler（ HPA ）对象，并把它指向该Deployment 。可以给HPA准备YAML manifest，但有个办法更简单——还可以用kubectl autoscale 命令：

```
$ kubectl autoscale deployment kubia --cpu-percent=30 --min=1 --max=5
deployment "kubia" autoscaled
```

这会帮你创建HPA对象，并将叫作kubia的Deployment 设置为伸缩目标。你还设置了pod 的目标CPU使用率为30% ，指定了副本的最小和最大数量。Autoscaler 会持续调整副本的数量以使CPU 使用率接近30% ， 但它永远不会调整到少于一个或多于五个。



注意 HPA 资源存在多个版本：新的 autoscaling/v2beta1 和旧的 autoscaling/v1。此处请求的是新版资源。

基于CPU使用率的自动扩缩容演示：

【注】本步骤为了方便演示，不再使用golang编写的example程序，而是简单使用一个nginx镜像。

首先准备好两个配置文件清单：1）nginx-deployment.yaml，2）nginx-svc.yaml，并在K8s集群中通过命令创建它们，如下图所示。

`nginx-deployment.yaml`

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30023
  selector:  #标签选择器
    app: nginx
```

创建deployment、Service对象

```
root@ljl-virtual-machine:/home/ljl# kubectl create -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
root@ljl-virtual-machine:/home/ljl# kubectl create -f nginx-svc.yaml
service/nginx created
```

成功创建完两个资源对象后我们就成功启动nginx服务了。如遇到API版本报错问题请见博客：https://kubernetes.io/blog/2019/07/18/api-deprecations-in-1-16/

下面检查nginx Pods的运行状态，如状态为running则表示已正常运行。

```
[root@ljl-virtual-machine]# kubectl get deployments.apps
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3                  3               109s

[root@ljl-virtual-machine]# kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE
```

```
  IP            NODE         NOMINATED NODE   READINESS GATES
7 nginx-deployment-6f6d9b887f-57dvw   1/1     Running   0          119s
  10.254.1.15   k8s-node-1   <none>           <none>
8 nginx-deployment-6f6d9b887f-bk5nl   1/1     Running   0          119s
  10.254.2.14   k8s-node-2   <none>           <none>
9 nginx-deployment-6f6d9b887f-lbr27   1/1     Running   0          119s
  10.254.1.16   k8s-node-1   <none>           <none>
```

接下来输入指令创建HPA操作。说明扩缩容操作目的是将pod数量控制在2-8之间，期望的cpu使用率不超过80%。

```
1 [root@ljl-virtual-machine]# kubectl autoscale deployment nginx-deploy
  ment --max=8 --min=2 --cpu-percent=80
2 deployment "nginx-deployment" autoscaled
```

最后我们测试一下水平扩缩容，如下所示。我们强制让pod副本数降到1个，但HPA会根据之前的配置以及当前cpu的使用率决定pod副本的数量，因此HPA会将pod副本变为2。
以下演示即随时间副本数量的变化过程：

```
 1 [root@ljl-virtual-machine]# kubectl scale deployment nginx-deploymen
   t --replicas=1
 2 deployment "nginx-deployment" scaled
 3 [root@ljl-virtual-machine]# kubectl get pod -o wide
 4 NAME                                READY     STATUS        RESTARTS
   AGE
 5 nginx-deployment-4019830974-01j1n   1/1       Running       0
   14m
 6 nginx-deployment-4019830974-2x9j4   1/1       Terminating   0
   14m
 7 nginx-deployment-4019830974-pxvb9   1/1       Terminating   0
   14m
 8 [root@ljl-virtual-machine]#
 9 [root@ljl-virtual-machine]# kubectl get pod -o wide
10 NAME                                READY     STATUS        RESTARTS
   AGE
11 nginx-deployment-4019830974-01j1n   1/1       Running       0
```

```
   14m
12 nginx-deployment-4019830974-2x9j4   1/1       Terminating   0
   14m
13 nginx-deployment-4019830974-pxvb9   1/1       Terminating   0
   14m
14 [root@ljl-virtual-machine]# kubectl get pod -o wide
15 NAME                                READY     STATUS    RESTARTS   A
   GE
16 nginx-deployment-4019830974-01j1n   1/1       Running   0          1
   4m
17 [root@ljl-virtual-machine]#
18 [root@ljl-virtual-machine]# kubectl get pod -o wide
19 NAME                                READY     STATUS              RE
   STARTS   AGE
20 nginx-deployment-4019830974-01j1n   1/1       Running             0
   14m
21 nginx-deployment-4019830974-bg35z   0/1       ContainerCreating   0
   2s
22 [root@ljl-virtual-machine]# kubectl get pod -o wide
23 NAME                                READY     STATUS    RESTARTS   A
   GE
24 nginx-deployment-4019830974-01j1n   1/1       Running   0          1
   4m
25 nginx-deployment-4019830974-bg35z   1/1       Running   0          7
   s
```

六、Prometheus监控平台的使用

0、 在进行本步骤前，需阅读并了解prometheus、Exporter、Counter指标、Hisogram指标等概念。可参考资料：📎 48 Prometheus、Metrics Server与Kubernetes监控体系.html

1、在集群中部署prometheus

（1）为prometheus准备配置文件prometheus.config.yml

```yaml
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
```

```
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval:     15s
      evaluation_interval: 15s
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
        - targets: ['localhost:9090']
```

(2) 由于prometheus要与API server交互，因此为其准备权限配置文件prometheus.rbac.yml

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [""]
    resources:
      - nodes
      - nodes/proxy
      - services
      - endpoints
      - pods
    verbs: ["get", "list", "watch"]
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs: ["get", "list", "watch"]
  - nonResourceURLs: ["/metrics"]
    verbs: ["get"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: default
```

```yaml
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: default
```

(3) 正式部署prometheus，以Deployment方式部署于K8s集群内

```yaml
apiVersion: v1
kind: "Service"
metadata:
  name: prometheus
  labels:
    name: prometheus
spec:
  ports:
    - name: prometheus
      protocol: TCP
      port: 9090
      targetPort: 9090
  selector:
    app: prometheus
  type: NodePort
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    name: prometheus
  name: prometheus
spec:
```

```
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      serviceAccount: prometheus
      containers:
        - name: prometheus
          image: prom/prometheus:v2.2.1
          command:
            - "/bin/prometheus"
          args:
            - "--config.file=/etc/prometheus/prometheus.yml"
          ports:
            - containerPort: 9090
              protocol: TCP
          volumeMounts:
            - mountPath: "/etc/prometheus"
              name: prometheus-config
      volumes:
        - name: prometheus-config
          configMap:
            name: prometheus-config
```

2、 修改业务逻辑代码，增加prometheus Exporter

【注】以下代码片段截图，仅供方便理解，并不完整，需查看https://github.com/tjrone/example中完整的golang示例程序，其他编程语言处理逻辑类似。

（1）此处定义两个指标，分别为请求次数和请求的处理时延：

```go
var (
    requestCount = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name:      "request_total",
            Help:      "Number of request processed by this service.",
        }, []string{},
    )

    requestLatency = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name:      "request_latency_seconds",
            Help:      "Time spent in this service.",
            Buckets:   []float64{0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 30.0, 60.0, 120.0, 300.0},
        }, []string{},
    )
)
```

（2）在原业务逻辑代码中进行**打点**，提取数据：

即在业务逻辑处理前为请求计数器作+1操作，并标记当前时间，在处理后记录所需处理时间。

```go
func index(w http.ResponseWriter, r *http.Request) {
    timer:=metrics.NewAdmissionLatency()
    metrics.RequestIncrease()
    num:=os.Getenv( key: "Num")
    if num==""{
        Fibonacci( n: 10)
    }else{
        numInt,_:=strconv.Atoi(num)
        Fibonacci(numInt)
    }
    w.Write([]byte("success"))
    log.Println( v...: "success")
    timer.Observe()
}
```

（3）将记录好的指标，以URL方式暴露

```go
func Register() {
    prometheus.MustRegister(requestCount)
    prometheus.MustRegister(requestLatency)
}
```

```go
func main(){
    http.HandleFunc( pattern: "/abc", index)
    http.Handle( pattern: "/metrics", promhttp.Handler())
    metrics.Register()
    err := http.ListenAndServe( addr: ":5565",  handler: nil) // 设置监听的端口
    if err != nil {
        log.Fatal( v...: "ListenAndServe: ", err)
    }
}
```

3、本地测试Exporter是否编写成功

访问三次127.0.0.1:5565/abc

然后访问127.0.0.1:5565/metrics， 可见如下则编写成功：

```
# TYPE request_latency_seconds histogram
request_latency_seconds_bucket{le="0.01"} 3
request_latency_seconds_bucket{le="0.02"} 3
request_latency_seconds_bucket{le="0.05"} 3
request_latency_seconds_bucket{le="0.1"} 3
request_latency_seconds_bucket{le="0.2"} 3
request_latency_seconds_bucket{le="0.5"} 3
request_latency_seconds_bucket{le="1"} 3
request_latency_seconds_bucket{le="2"} 3
request_latency_seconds_bucket{le="5"} 3
request_latency_seconds_bucket{le="10"} 3
request_latency_seconds_bucket{le="20"} 3
request_latency_seconds_bucket{le="30"} 3
request_latency_seconds_bucket{le="60"} 3
request_latency_seconds_bucket{le="120"} 3
request_latency_seconds_bucket{le="300"} 3
request_latency_seconds_bucket{le="+Inf"} 3
request_latency_seconds_sum 0.00054015
request_latency_seconds_count 3
```

4、“发布”新的版本（带有metrics的版本）

（1）重新编译

```
root@node2:~/example/metrics_version# go build -o example
```

(2)重新制作镜像，取了一个新的tag

```
root@node2:~/example/metrics_version# docker build -t example:0.2 .
Sending build context to Docker daemon  11.77MB
Step 1/5 : FROM ubuntu
 ---> ccc6e87d482b
Step 2/5 : ADD example /
 ---> 570ff5f09821
Step 3/5 : ENV Num 10
 ---> Running in 6e8c97d9b7d7
Removing intermediate container 6e8c97d9b7d7
 ---> 3f85665e263f
Step 4/5 : EXPOSE 5565
 ---> Running in 233a69d3557c
Removing intermediate container 233a69d3557c
 ---> 302c074c573c
Step 5/5 : ENTRYPOINT ["./example"]
 ---> Running in e81ce68ac1c2
Removing intermediate container e81ce68ac1c2
 ---> 39699dac3546
Successfully built 39699dac3546
Successfully tagged example:0.2
```

(3) 更新deployment的YAML配置，更新其中的镜像

```
root@master:~/example# kubectl edit deployments example-service
deployment.extensions/example-service edited
```

(4) 使用Pod IP测试metrics exporter是否正常

先查看Pod IP

```
NAME                               READY   STATUS    RESTARTS   AGE    IP              NODE    NOMINATED NODE   READINESS GATE
S
etcd-operator-85f7494fcf-95qv4     1/1     Running   14         78d    192.168.1.46    node2   <none>           <none>
example-service-6fc78ff896-8dg82   1/1     Running   0          12s    192.168.1.93    node2   <none>           <none>
example-service-6fc78ff896-hx9cw   1/1     Running   0          14s    192.168.1.92    node2   <none>           <none>
hamster-6d7dd457c6-sgplq           1/1     Running   1          92d    192.168.1.45    node2   <none>           <none>
hamster-6d7dd457c6-wdjrs           1/1     Running   1          92d    192.168.1.44    node2   <none>           <none>
prometheus-567d6d47d-qfsh4         1/1     Running   0          3d2h   192.168.1.83    node2   <none>           <none>
```

开始测试：正常显示各种metrics

```
root@master:~/example/deploy# curl 192.168.1.93:5565/metrics
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 8
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.4"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
```

5、测试通过service访问 exporter是否正常

a) 先访问三次服务

```
root@master:~/example/deploy# kubectl get svc
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
example-service   ClusterIP   10.111.101.163   <none>        80/TCP           109m
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP          105d
nginx             ClusterIP   None             <none>        80/TCP           75d
prometheus        NodePort    10.107.190.103   <none>        9090:32126/TCP   3d2h
root@master:~/example/deploy# curl 10.111.101.163/abc
there is env Num. Computation successed
root@master:~/example/deploy# curl 10.111.101.163/abc
there is env Num. Computation successed
root@master:~/example/deploy# curl 10.111.101.163/abc
there is env Num. Computation successed
```

b) 测试指标中是否包含所需信息(多执行)

```
root@master:~# curl 10.111.101.163/metrics
```

测试成功结果包含如下：

```
# HELP request_latency_seconds Time spent in this service.
# TYPE request_latency_seconds histogram
request_latency_seconds_bucket{le="0.01"} 1
request_latency_seconds_bucket{le="0.02"} 1
request_latency_seconds_bucket{le="0.05"} 1
request_latency_seconds_bucket{le="0.1"} 1
request_latency_seconds_bucket{le="0.2"} 1
request_latency_seconds_bucket{le="0.5"} 1
request_latency_seconds_bucket{le="1"} 1
request_latency_seconds_bucket{le="2"} 1
request_latency_seconds_bucket{le="5"} 1
request_latency_seconds_bucket{le="10"} 1
request_latency_seconds_bucket{le="20"} 1
request_latency_seconds_bucket{le="30"} 1
request_latency_seconds_bucket{le="60"} 1
request_latency_seconds_bucket{le="120"} 1
request_latency_seconds_bucket{le="300"} 1
request_latency_seconds_bucket{le="+Inf"} 1
request_latency_seconds_sum 2.7307e-05
request_latency_seconds_count 1
# HELP request_total Number of request processed by this service.
# TYPE request_total counter
request_total 1
```

```
# HELP request_latency_seconds Time spent in this service.
# TYPE request_latency_seconds histogram
request_latency_seconds_bucket{le="0.01"} 2
request_latency_seconds_bucket{le="0.02"} 2
request_latency_seconds_bucket{le="0.05"} 2
request_latency_seconds_bucket{le="0.1"} 2
request_latency_seconds_bucket{le="0.2"} 2
request_latency_seconds_bucket{le="0.5"} 2
request_latency_seconds_bucket{le="1"} 2
request_latency_seconds_bucket{le="2"} 2
request_latency_seconds_bucket{le="5"} 2
request_latency_seconds_bucket{le="10"} 2
request_latency_seconds_bucket{le="20"} 2
request_latency_seconds_bucket{le="30"} 2
request_latency_seconds_bucket{le="60"} 2
request_latency_seconds_bucket{le="120"} 2
request_latency_seconds_bucket{le="300"} 2
request_latency_seconds_bucket{le="+Inf"} 2
request_latency_seconds_sum 5.5468999999999994e-05
request_latency_seconds_count 2
# HELP request_total Number of request processed by this service.
# TYPE request_total counter
request_total 2
```

【注】此测试步骤会出现以下问题：
由于example-service后端的endpoints有两个（deployment中设置了replicas=2），所以a）步骤可能会负载均衡至2个pod进行处理。因此，我们测试a)中访问了3次，可能会出现“0+3，1+2”等情况。
步骤b)中，可检验几次，只要最终结果加和为3即可

7、查看prometheus配置文件中配置的target

```
root@master:~/example/deploy# vi prometheus.config.yml
```

配置中有如下的target。如kubernetes-endpoints 这个target，即将所有的endpoints作为拉取目标

```
- job_name: 'kubernetes-nodes'
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: node

- job_name: 'kubernetes-service'
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: service

- job_name: 'kubernetes-endpoints'
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: endpoints

- job_name: 'kubernetes-pods'
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role: pod
```

8、浏览器直接打开prometheus

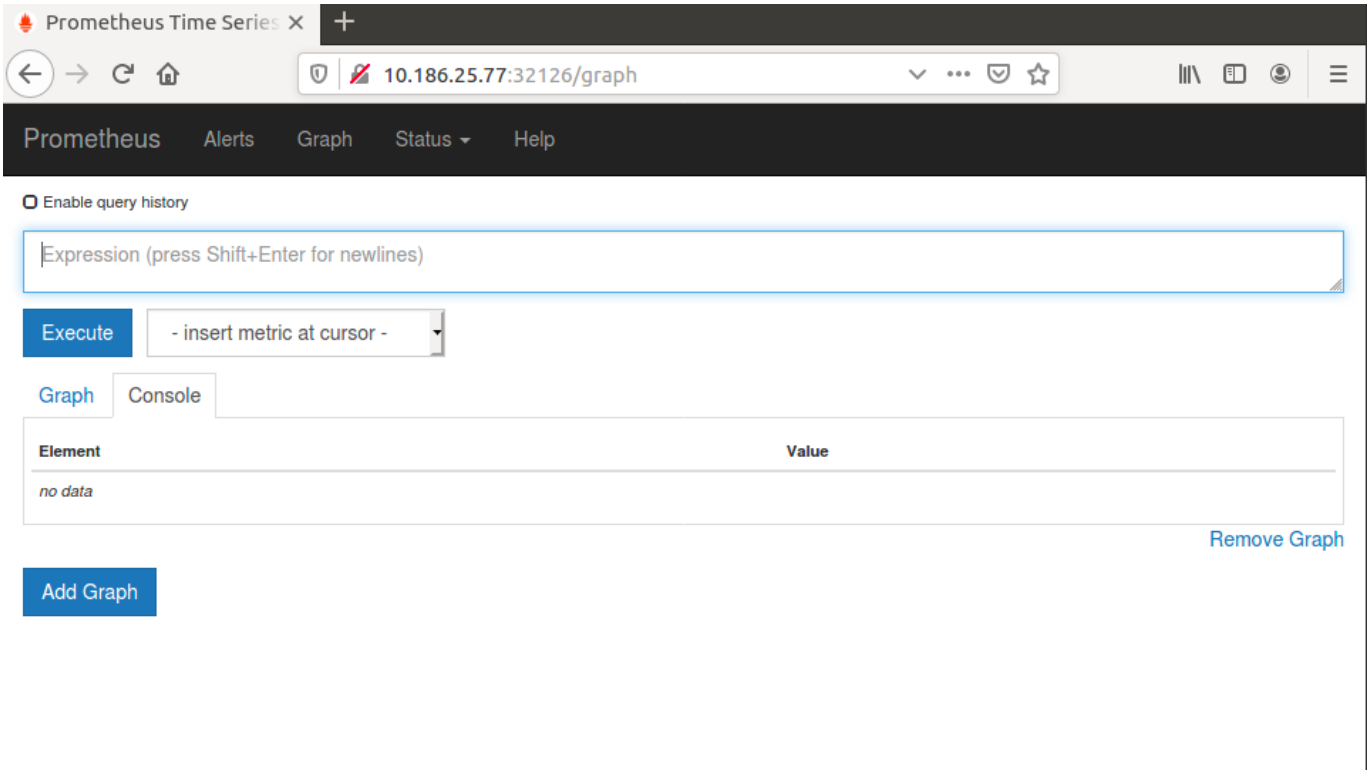
查看prometheus部署时，部署好的NodePort类型的Service

```
root@master:~/example/deploy# kubectl get svc
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
example-service   ClusterIP   10.111.101.163   <none>        80/TCP           133m
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP          105d
nginx             ClusterIP   None             <none>        80/TCP           75d
prometheus        NodePort    10.107.190.103   <none>        9090:32126/TCP   3d3h
```

查看内网IP是多少

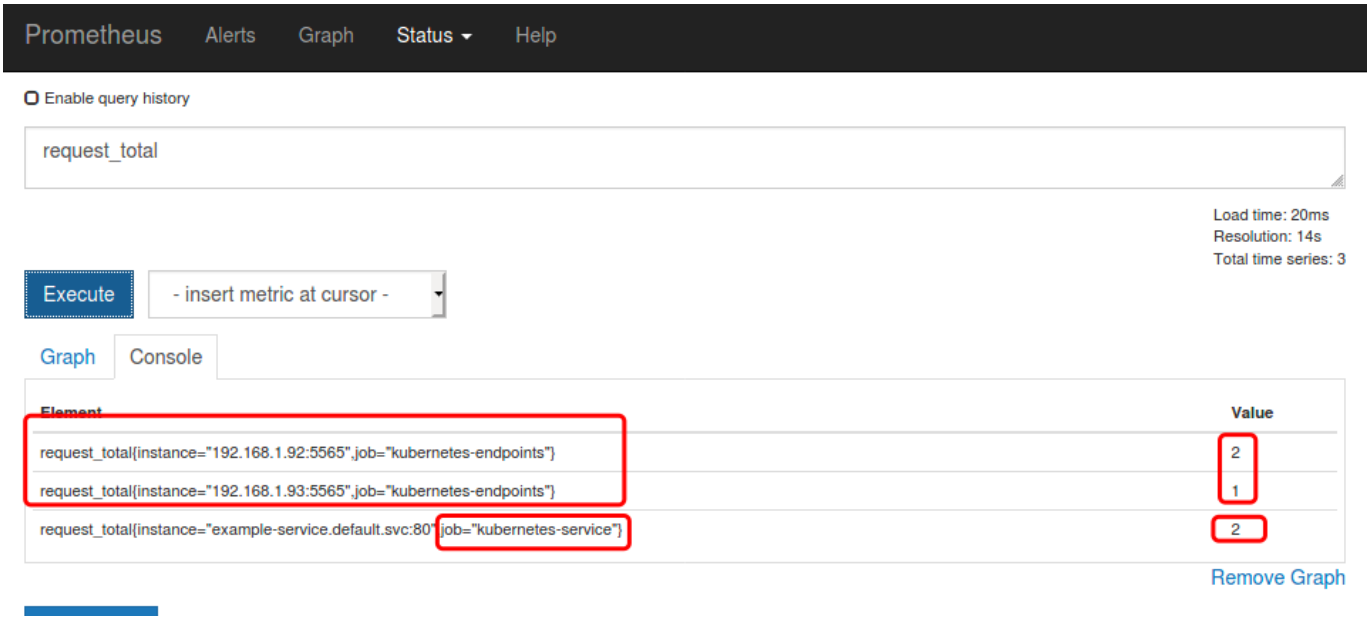
```
root@master:~/example/deploy# ifconfig
cali0b509f65c17 Link encap:Ethernet  HWaddr ee:ee:ee:ee:ee:ee
          inet6 addr: fe80::ecee:eeff:feee:eeee/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15162343 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15414048 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1318413624 (1.3 GB)  TX bytes:5961621666 (5.9 GB)

cali2d90edc2c4e Link encap:Ethernet  HWaddr ee:ee:ee:ee:ee:ee
          inet6 addr: fe80::ecee:eeff:feee:eeee/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15165719 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15414285 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1317770455 (1.3 GB)  TX bytes:5961664234 (5.9 GB)

docker0   Link encap:Ethernet  HWaddr 02:42:2f:a5:2e:d3
          inet addr:172.17.0.1  Bcast:172.17.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:2fff:fea5:2ed3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:29 errors:0 dropped:0 overruns:0 frame:0
          TX packets:37 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1735 (1.7 KB)  TX bytes:2712 (2.7 KB)

eth0      Link encap:Ethernet  HWaddr fa:16:3e:02:32:6a
          inet addr:10.186.25.77  Bcast:10.186.25.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fe02:326a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:91204329 errors:0 dropped:0 overruns:0 frame:0
          TX packets:88686598 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:23038578574 (23.0 GB)  TX bytes:49580699983 (49.5 GB)
```

由于，我实验环境的服务器没有Desktop，在同一局域网，我开了一台有Desktop的服务器，打开如下界面：



输入metrics的名字request_total:

可以看到：



观察上图，可见配置成功！

参考材料：

1、什么是容器？什么是镜像？

📎 05 白话容器基础（一）：从进程说开去.html

📎 06 白话容器基础（二）：隔离与限制.html

📎 07 白话容器基础（三）：深入理解容器镜像.html

📎 08 白话容器基础（四）：重新认识Docker容器.html

2、部署kubernetes

📎 10 Kubernetes一键部署利器：kubeadm.html📎 11 从0到1：搭建一个完整的Kubernetes集群.html

3、prometheus介绍：

📎 48 Prometheus、Metrics Server与Kubernetes监控体系.html

4、Service介绍

📎 37 找到容器不容易：Service、DNS与服务发现.html