nju - 云原生大作业说明文档

项目地址： https://github.com/maoding1/cloud-native-2023

成员及分工：

姓名	学号	分工
毛丁	211098325	springboot基础功能+限流+暴露prometheus接口，jenkins pipeline编写,jmeter压测,prometheus+grafana监控+文档编写
刘克典	211230043	springboot基础功能+限流，测试prometheus接口，jenkins pipeline构建测试，服务接口测试，文档编写
林奥	211180234	springboot基础功能+限流，测试prometheus接口，jenkins pipeline编写，文档编写

springboot基础功能实现

1. 导入依赖

除了springboot项目的一些maven依赖外，关于json对象和限流需要导入以下依赖：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.67_noneautotype2</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1-jre</version>
</dependency>
```

其中json对象使用了alibaba的fastjson库，而限流使用的是Google开源工具包Guava。

2.构建项目 实现接口和初步限流

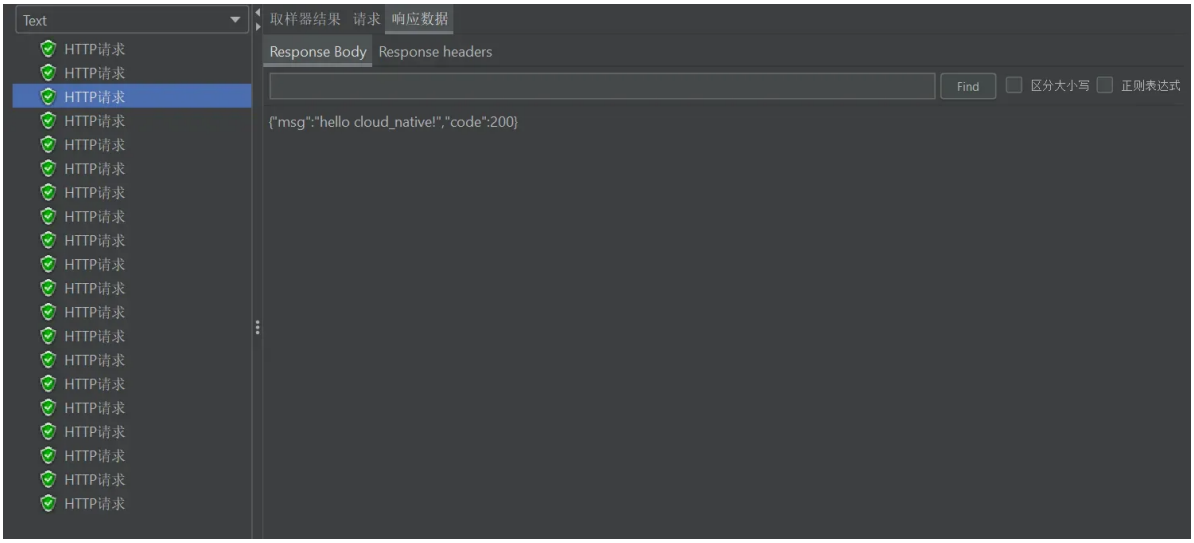
```java
@RestController
public class HelloController {
    /* 为了方便测试，限流策略设为1秒钟2个令牌,正式提交时应改为每秒100个令牌**/
    1 usage
    private final RateLimiter limiter = RateLimiter.create(2.0);
    @RequestMapping("/hello")
    public String hello() {
        JSONObject jsonObject = new JSONObject();
        //令牌桶算法实现流量控制 五百毫秒内每拿到令牌则失败
        boolean tryAcquire = limiter.tryAcquire( timeout: 500, TimeUnit.MILLISECONDS);
        if (tryAcquire) {
            jsonObject.put("code", 200);
            jsonObject.put("msg", "hello cloud_native!");
        } else {
            jsonObject.put("code", 429);
            jsonObject.put("msg", "too many requests");
        }

        return jsonObject.toJSONString();
    }
}
```

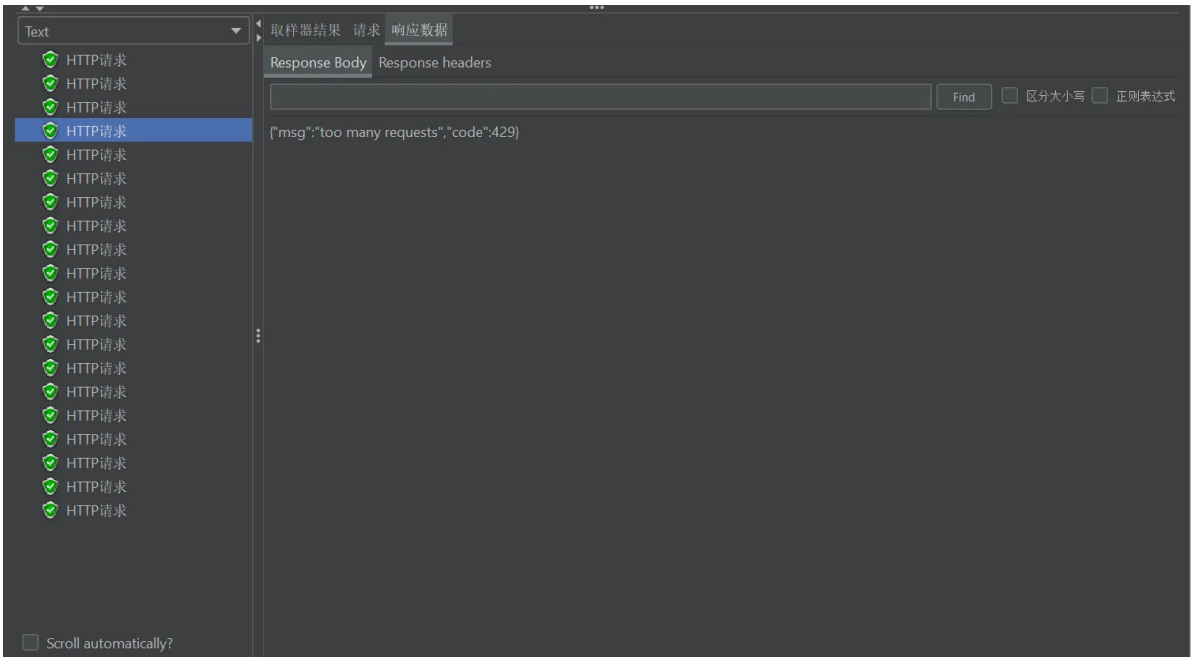
3. 对基本功能和限流功能的测试

使用jmeter工具对接口进行测试，测试的url为localhost:8080/hello

线程组设置：启动20个线程在一秒内循环一次，即一秒内对url发送20次请求

结果如下：有正常被接受的请求，也有因为限流策略被拒绝的请求





构建时单元测试

在/test目录下加入测试类HelloControllerTest:

```
package com.example.hello;

import com.google.common.util.concurrent.RateLimiter;
import org.json.JSONException;
import org.json.JSONObject;
import org.junit.Before;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.junit.MockitoJUnitRunner;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.concurrent.TimeUnit;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;
```

```java
@RunWith(MockitoJUnitRunner.class)
@SpringBootTest
public class HelloControllerTests {

    @Mock
    private RateLimiter rateLimiter;

    @InjectMocks
    private HelloController helloController;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testHelloWithTokenAcquired() throws JSONException {
        // Mock the rateLimiter.tryAcquire() method to return true
        when(rateLimiter.tryAcquire(500, TimeUnit.MILLISECONDS)).thenReturn(true);

        // Call the hello() method
        helloController.SetRateLimiter(rateLimiter);
        String response = helloController.hello();

        // Verify the response
        JSONObject jsonObject = new JSONObject(response);
        assertEquals(200, jsonObject.getInt("code"));
        assertEquals("hello cloud_native!", jsonObject.getString("msg"));
    }

    @Test
    public void testHelloWithTokenNotAcquired() throws JSONException {
        // Mock the rateLimiter.tryAcquire() method to return false
        when(rateLimiter.tryAcquire(500, TimeUnit.MILLISECONDS)).thenReturn(false);

        // Call the hello() method
        helloController.SetRateLimiter(rateLimiter);
        String response = helloController.hello();

        // Verify the response
        JSONObject jsonObject = new JSONObject(response);
        assertEquals(429, jsonObject.getInt("code"));
        assertEquals("too many requests", jsonObject.getString("msg"));
    }
}
```

在pipeline打包时使用 mvn clean test package插入执行单元测试步骤

本地执行测试结果如下：

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.479 s - in com.example.hello.HelloControllerTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  7.779 s
[INFO] ------------------------------------------------------------------------
```

实现接口访问指标，并暴露给Prometheus

maven中加入如下依赖：

```
<!-- Spring Boot Actuator -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>

        <!-- Prometheus Java Client -->
        <dependency>
            <groupId>io.prometheus</groupId>
            <artifactId>simpleclient_spring_boot</artifactId>
            <version>0.11.0</version>
        </dependency>
<dependency>
            <groupId>io.micrometer</groupId>
            <artifactId>micrometer-registry-prometheus</artifactId>
        </dependency>
```

在项目配置中加入如下配置，暴露指标：

```
# 启用Actuator的所有端点
management.endpoints.web.exposure.include=*

# 配置Prometheus指标端点
management.endpoint.metrics.enabled=true
management.endpoint.metrics.path=/actuator/prometheus
```

bonus：对实例进行统一限流

使用tomcat完成相关功能

下载与安装：

```
wget https://downloads.apache.org/tomcat/tomcat-9/v9.0.52/bin/apache-tomcat-
9.0.52.tar.gz
tar -xzf apache-tomcat-9.0.52.tar.gz
```

在tomcat安装目录下的conf/server.xml中进行如下配置：

```
<Connector port="30034" protocol="HTTP/1.1"
           connectionTimeout="5000" maxThreads="100"
           redirectPort="30034" />
```

- `port`：指定连接器监听的端口号，这里设置为30034，表示后文中springboot应用启动时集群暴露给外界的端口。
- `protocol`：指定连接器使用的协议，这里使用的是HTTP/1.1。
- `connectionTimeout`：定义连接的超时时间，即如果连接在指定的时间内没有活动，则会被关闭。这里设置为5000毫秒（5秒）。
- `maxThreads`：定义了最大线程数，即Tomcat容器可以同时处理的最大请求数量。当并发请求数超过此限制时，新的请求将排队等待处理。这里设置为100，表示最大线程数为100。
- `redirectPort`：指定重定向端口，当连接器接收到安全请求（例如HTTPS）时，它会将请求重定向到指定的端口。这里设置为30034，表示不进行重定向。

或者使用k8s中的VirtualService对象对service进行统一限流：

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: nju-34-virtualService
spec:
  hosts:
  - cloud-native-34-svc
  http:
  - route:
    - destination:
        host: cloud-native-34-svc
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
      tcp:
        maxConnections: 1
    httpReqTimeout: 3s
    maxConnections: 1
    maxRequestsPerConn: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

DevOps功能实现

docker镜像构建

dockerfile文件如下：

```dockerfile
# 使用基础的Java 18镜像
FROM openjdk:18

# 设置工作目录
WORKDIR /app

# 将编译后的Spring Boot JAR文件复制到容器中
COPY ./target/hello-0.0.1-SNAPSHOT.jar app.jar

EXPOSE 8080
# 设置启动命令
CMD ["java", "-jar", "app.jar"]
```

k8s deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: project-group34
```

```
  namespace: nju34
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cloud-native-34
  template:
    metadata:
      labels:
        app: cloud-native-34
    spec:
      containers:
        - name: group34-containers
          image: harbor.edu.cn/nju34/34_images:VERSION
          resources:
            requests:
              memory: 50Mi
              cpu: 50m
      #需要提前创建secret资源对象，用于从私有仓库拉取镜像，否则容器会创建失败
      imagePullSecrets:
        - name: nju34
```

k8s service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: cloud-native-34-svc
  namespace: nju34 #规定命名空间为nju34
spec:
  type: NodePort
  selector:
    app: cloud-native-34 #与depolyment中容器的标签匹配
    ports:
      - nodePort: 30034 # 外部访问端口
        port: 8888 #集群内部端口
        targetPort: 8080 #所有流量最终路由到的端口
```

servermonitor.yaml

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: project-group34-monitor
  namespace: nju34
  labels:
    app: cloud-native-34
spec:
  namespaceSelector:
    matchNames:
      - nju34
  selector:
    matchLabels:
      app: cloud-native-34
```

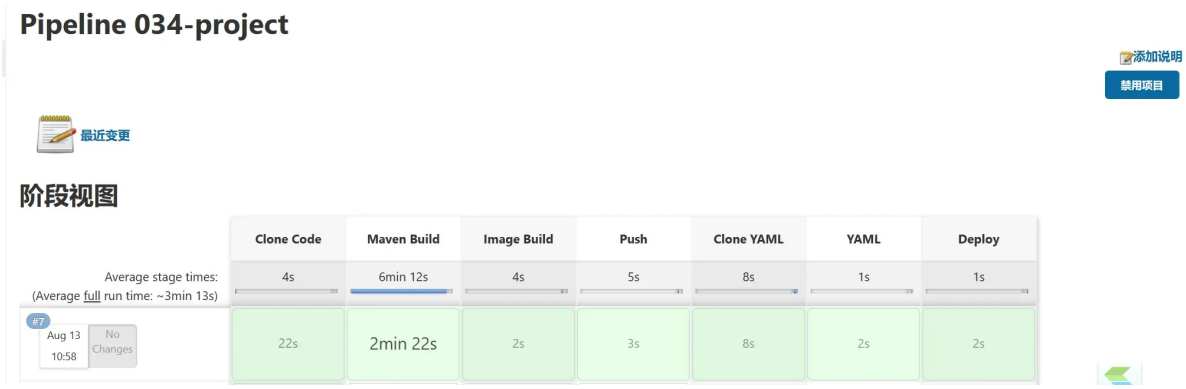
```
  endpoints:
    - port: http
      interval: 15s
      path: /actuator/prometheus
```

jenkins pipeline

```
pipeline{
    agent none
    stages {
        stage('Clone Code') {
            agent {
                label 'master'
            }
            steps {
                echo "1.Git Clone Code"
                sh 'curl "http://p.nju.edu.cn/portal_io/logout"'
                sh 'curl "http://p.nju.edu.cn/portal_io/login?
username=xxx&password=xxx"'
                git url: "https://gitee.com/md2002/cloud-native-2023.git"
             # 设定了github仓库的地址，由于服务器无法访问外网，因此配置了github仓库与gitee仓库的
同步
            }
        }

        stage('Maven Build') {
            agent {
                docker {
                    image 'maven:latest'
                    args ' -v /home/nju34:/home/nju34'
                }
            }
            steps {
                echo "2.Maven Build Stage"
                sh 'mvn clean install package \'-Dmaven.test.skip=true\''
            }
        }

        stage('Image Build') {
            agent {
                label 'master'
            }
            steps {
                echo "3.Image Build Stage"
                sh 'docker build -f Dockerfile --build-arg jar_name=target/hello-
0.0.1-SNAPSHOT.jar -t 34_images:${BUILD_ID} .'
                sh 'docker tag 34_images:${BUILD_ID}
harbor.edu.cn/nju34/34_images:${BUILD_ID}'
            }
        }

        stage('Push') {
            agent {
                label 'master'
            }
```

```
            steps {
                echo "4.Push Docker Image Stage"
                sh "docker login --username=nju34 harbor.edu.cn -p nju342023"
                sh 'docker push harbor.edu.cn/nju34/34_images:${BUILD_ID}'
            }
        }
    }
}
node('slave'){
    container('jnlp-kubectl'){
        stage('Clone YAML'){
            echo "5.Git Clone YAML to Slave"
            sh 'curl "http://p.nju.edu.cn/portal_io/logout"'
            sh 'curl "http://p.nju.edu.cn/portal_io/login?username=xxx&password=xxx"'
            git url: "https://gitee.com/md2002/cloud-native-2023.git"
        }
        stage('YAML'){
            echo"6.Change YAML File Stage"
            sh 'sed -i "s#{VERSION}#${BUILD_ID}#g"   deployment.yaml'
            sh 'sed -i "s#{VERSION}#${BUILD_ID}#g"   service.yaml'
        }
        stage('Deploy'){
            echo "7.Deploy To K8s Stage"
            sh "kubectl apply -f deployment.yaml -n nju34"
            sh "kubectl apply -f service.yaml -n nju34"
        }
    }
}
```

jenkins部署成功截图：



使用kubectl查看部署情况并使用curl命令测试：

```
nju34@host-172-29-4-18:~
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS C:\Users\13584> ssh nju34@172.29.4.18
nju34@172.29.4.18's password:
Last login: Mon Aug 14 18:31:32 2023 from 172.29.48.239
[nju34@host-172-29-4-18 ~]$ kubectl get deployment -n nju34
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
project-group34   2/2     1            2           31h
[nju34@host-172-29-4-18 ~]$ kubectl get service -n nju34
NAME                 TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
cloud-native-34-svc   NodePort   10.105.242.171   <none>        8888:30034/TCP   31h
[nju34@host-172-29-4-18 ~]$ curl localhost:30034/hello
{"msg":"hello cloud_native!","code":200}[nju34@host-172-29-4-18 ~]$
```

扩容场景

prometheus查看监控指标

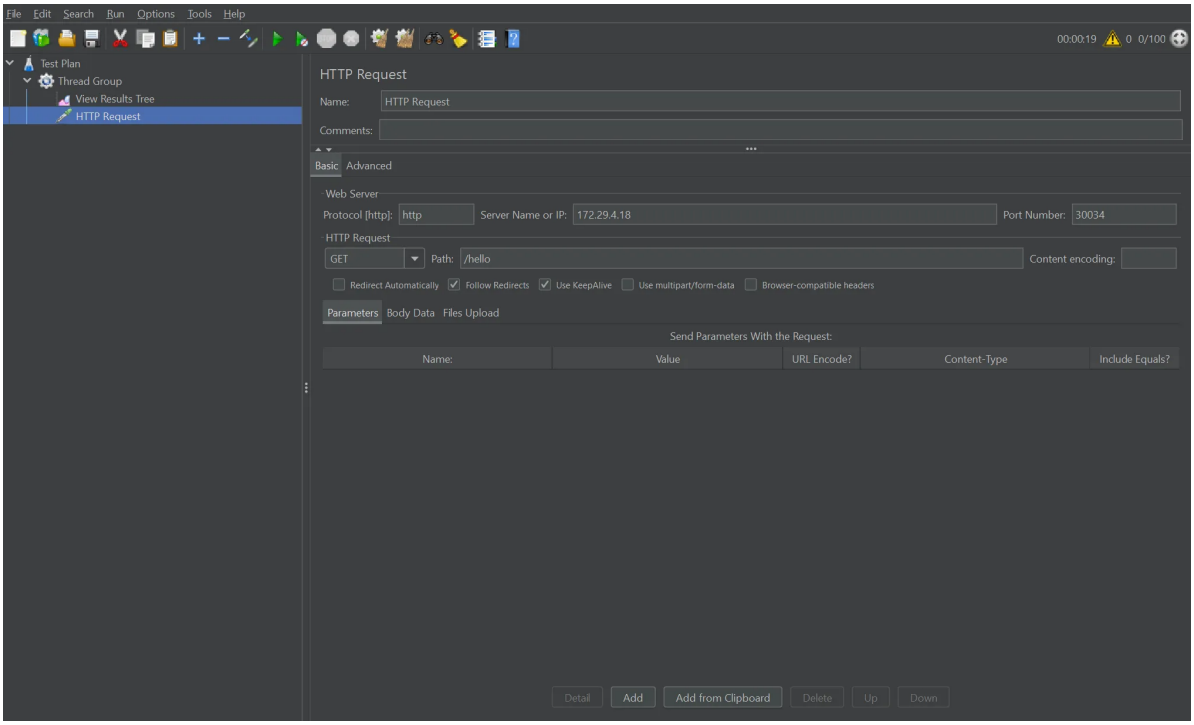
创建servicemonitor对象后通过服务发现机制就能自动开始监控服务信息了：

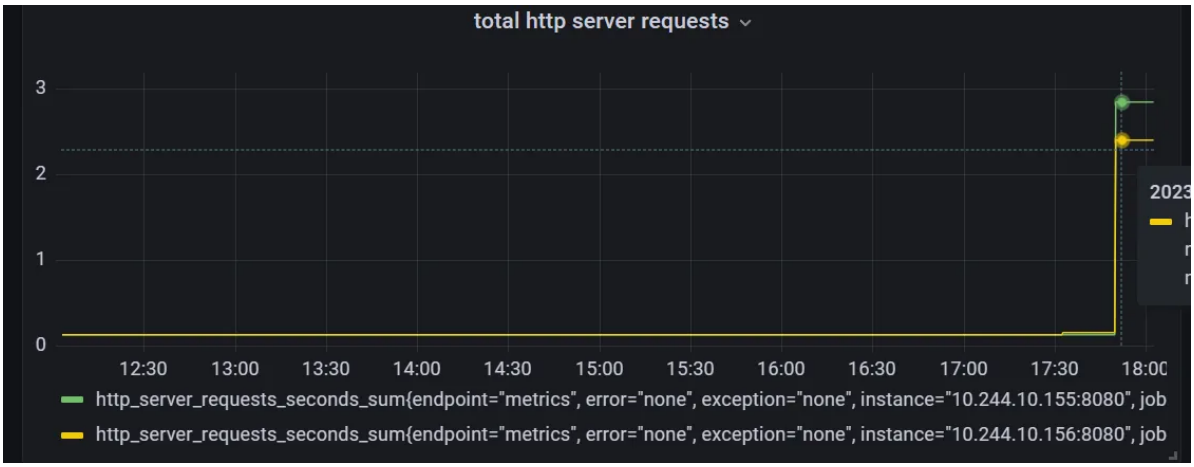
serviceMonitor/nju34/project-group34-monitor/0 (2/2 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.244.10.155:8080/actuator/prometheus	UP	endpoint="metrics" instance="10.244.10.155:8080" job="cloud-native-34-svc" namespace="nju34" pod="project-group34-dddcb8455-zlj6" service="cloud-native-34-svc"	32.149s ago	8.795ms	
http://10.244.10.156:8080/actuator/prometheus	UP	endpoint="metrics" instance="10.244.10.156:8080" job="cloud-native-34-svc" namespace="nju34" pod="project-group34-dddcb8455-bmmlr" service="cloud-native-34-svc"	25.741s ago	9.047ms	

Grafana 监控大屏

使用jmeter对服务的/hello启动一秒内100次的请求：





可以看到这里曲线上升了一段(与上图相比少了两条斜线，因为那两条斜线是prometheus使用http协议定期对/atuator/prometheus端口请求数据，这里取消了对这个路径http服务的监控)

使用k8s命令手工扩容，并再次观察Grafana的监控数据

使用命令`kubectl scale deployment project-group34 --replicas=3`将pod的数量从2扩容到3，

但是由于当前用户没有权限：Error from server (Forbidden): deployments.apps "project-group34" is forbidden: User "nju34" cannot patch resource "deployments/scale" in API group "apps" in the namespace "nju34"

所以在jenkins中将deployment中定义的pod数量从2改到3后重新apply。

这里由于集群资源紧张，用时1天后pod仍然处于containner creating阶段，遂未成功。

```
[nju34@host-172-29-4-18 ~]$ kubectl get pod -n nju34
NAME                                READY   STATUS              RESTARTS   AGE
project-group34-5bd84869c8-97kp7    0/1     ContainerCreating   0          6h14m
project-group34-8578868b4f-j9rlq    0/1     ContainerCreating   0          31h
project-group34-dddcb8455-bmmlr     1/1     Running             0          32h
project-group34-dddcb8455-zljr6     1/1     Running             0          32h
```

bonus： Auto Scale

可以通过 kubectl create 命令创建一个 HPA 对象，

此外，也可以使用简便的命令 kubectl autoscale 来创建 HPA 对象。 例如，在此项目中使用 kubectl autoscale deployment project-group34 --min=2 --max=5 --cpu-percent=50 将会为名为 project-group34 的 deployment对象创建一个 HPA 对象， 目标 CPU 使用率为 50%，副本数量配置为 2 到 5 之间。不过由于权限的原因 未能成功创建hpa对象。