

Spring Cloud Alibaba 致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用微服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务。

1.Spring cloud alibaba和Spring cloud

Spring cloud是一系列分布式框架的集合，基于Spring Boot开发，是一种规范，相当于Java中的接口，而Spring cloud Alibaba相当于是Spring cloud的一种实现。

作用：将不同的分布式组件，以Springboot的风格进行集成整合，需要哪个，就以Springboot的方式引入

2.Spring cloud Alibaba和boot以及Spring cloud的关系

Spring boot ==》 spring cloud ==》 spring cloud alibaba (Spring cloud alibaba不能直接使用，需要先整合Cloud)

要注意这三个框架的版本关系，去[Spring cloud alibaba Github](#)页面查看

2023/1/5版本

使用阿里云的Spring boot生成工具，Spring boot的版本为2.6.11，则cloud版本如下

```
<dependencyManagement>
  <dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dependencies -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>2021.0.4</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/com.alibaba.cloud/spring-cloud-alibaba-dependencies -->
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>2021.0.4.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

相关分布式组件的版本

每个 Spring Cloud Alibaba 版本及其自身所适配的各组件对应版本如下表所示（注意，Spring Cloud Dubbo 从 2021.0.1.0 起已被移除出主干，不再随主干演进）：

Spring Cloud Alibaba Version	Sentinel Version	Nacos Version	RocketMQ Version	Dubbo Version	Seata Version
2022.0.0.0-RC	1.8.6	2.2.1-RC	4.9.4	~	1.6.1
2.2.9.RELEASE	1.8.5	2.1.0	4.9.4	~	1.5.2
2021.0.4.0	1.8.5	2.0.4	4.9.4	~	1.5.2
2.2.8.RELEASE	1.8.4	2.1.0	4.9.3	~	1.5.1

3.服务治理

服务治理=服务注册+服务发现
Spring cloud alibaba使用Nacos作为服务注册发现框架

简单的Nacos使用

配置Maven依赖

服务注册和服务发现都需要添加Nacos的依赖

```
1 <dependency>
2     <groupId>com.alibaba.cloud</groupId>
3     <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
4     <version>2021.0.4.0</version>
5 </dependency>
```

服务注册

服务注册仅需要配置应用的名字和Nacos服务的url

```
1 spring.application.name=provide
2
3 spring.cloud.nacos.discovery.server-addr=localhost:8848
4 server.port=8082
```

服务发现

服务发现调用Spring boot自动注册的Nacos客户端进行访问

```
1 @RestController
2 public class ConsumerController {
3     @Autowired
4     private DiscoveryClient discoveryClient;
5     @GetMapping("/")
6     public List<ServiceInstance> instances(){
7         List<ServiceInstance> provide = this.discoveryClient.getInstances("provide");
8         return provide;
9     }
10 }
```

结果

如下图是启动了两个Provide服务时，服务发现的结果

```
[{"serviceId":"provide","instanceId":null,"host":"192.168.1.10","port":8082,"secure":false,"metadata":
{"nacos.instanceId":null,"nacos.weight":"1.0","nacos.cluster":"DEFAULT","nacos.ephemeral":"true","nacos.healthy":"true","preserved.register.source":"SPI
{"serviceId":"provide","instanceId":null,"host":"192.168.1.10","port":8081,"secure":false,"metadata":
{"nacos.instanceId":null,"nacos.weight":"1.0","nacos.cluster":"DEFAULT","nacos.ephemeral":"true","nacos.healthy":"true","preserved.register.source":"SPI
```

服务调用示例

服务提供方

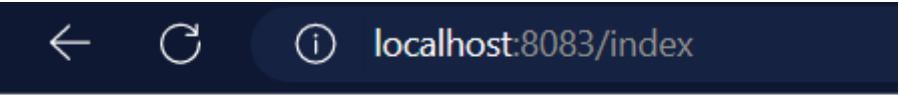
```
1 @RestController
2 public class ProviderController {
3     @Value("${server.port}")
4     private String port;
5
6     @GetMapping("/index")
7     public String index(){
8         return this.port;
9     }
10 }
```

服务消费方

```

1 @RestController
2 @Slf4j
3 public class ConsumerController {
4     @Autowired
5     private DiscoveryClient discoveryClient;
6     @Autowired
7     private RestTemplate restTemplate;
8     @GetMapping("/index")
9     public String index(){
10         //随机拿到Provide的示例
11         List<ServiceInstance> provide = this.discoveryClient.getInstances("provide");
12         int index = ThreadLocalRandom.current().nextInt(provide.size());
13         ServiceInstance instance = provide.get(index);
14         String url = instance.getUri()+"/port";
15         //调用一个
16         log.info("调用了端口是: "+instance.getPort());
17         return "调用了端口为: "+instance.getPort()+"的服务, 结果
为: "+restTemplate.getForObject(url,String.class);
18     }
19 }

```



调用了端口为: 8082的服务, 结果为: 8082

使用负载均衡

Spring Cloud 2020版本以后, 默认移除了对Netflix的依赖, 其中就包括Ribbon, 官方默认推荐使用Spring Cloud Loadbalancer 正式替换Ribbon, 并成为了Spring Cloud负载均衡器的唯一实现, 因此要在原有依赖的基础上添加 下面的Loadbalancer依赖。

此时, 默认为轮询调用服务

```

1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-loadbalancer</artifactId>

```

```
4     <version>3.1.3</version>
5 </dependency>
```

服务消费方修改代码如下

RestTemplate 添加 @LoadBalanced注解

```
1
2 @org.springframework.context.annotation.Configuration
3 public class Configuration {
4     @Bean
5     @LoadBalanced
6     public RestTemplate restTemplate(){
7         return new RestTemplate();
8     }
9 }
```

修改控制器方法，直接使用服务名调用

```
1 @GetMapping("/index")
2 public String index(){
3     return this.restTemplate.getForObject("
4     http://provide/index
5     ",String.class);
6 }
```

4.Sentinel 服务限流降级

主要是为了解决**雪崩效应**（一个服务宕机，导致其他消费该服务的服务一直尝试消费，进而也宕机，如同葫芦娃救爷爷一样，一个接着一个宕机）

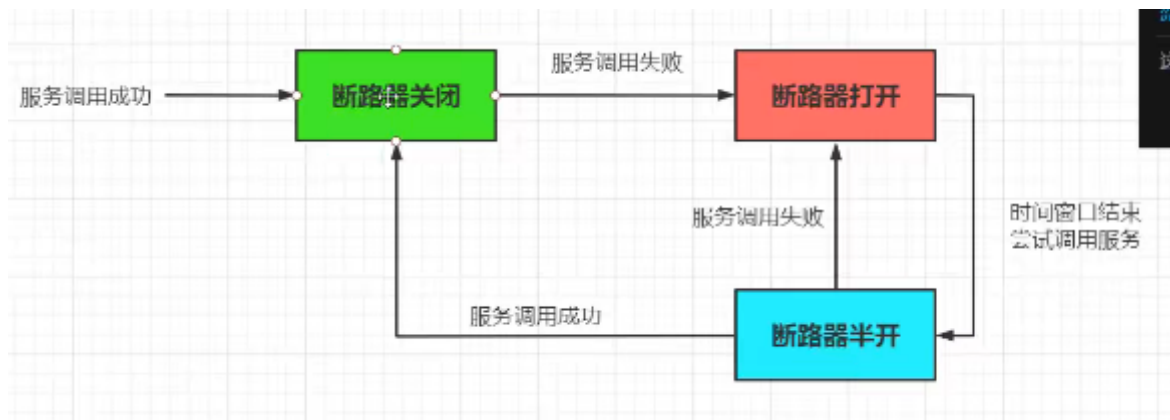
- 降级

放弃一些功能，保证整个系统能平稳运行

- 限流

通过对并发访问进行限速。最简单的方式，把多余的请求直接拒绝掉

- 熔断



Sentinel可以对一个资源进行下面四种操作

+ 流控
+ 熔断
+ 热点
+ 授权

+ 流控
+ 熔断
+ 热点
+ 授权

共 5 条记录 每页 16 条记录

Sentinel流控规则

首先添加配置

```

1 <dependency>
2   <groupId>com.alibaba.cloud</groupId>
3   <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
4   <version>2021.0.4.0</version>
5 </dependency>
6 <dependency>
7   <groupId>org.springframework.boot</groupId>
8   <artifactId>spring-boot-starter-actuator</artifactId>
9 </dependency>

```

然后启动Sentinel服务，修改Provide的配置

新增

其中前两条是配置spring-boot-starter-actuator的

```

1 management.server.port=8081
2 management.endpoints.web.exposure.include=*
3 spring.cloud.sentinel.transport.dashboard=localhost:8080
4

```

在Sentinel控制台添加如下配置

编辑流控规则

X

资源名

/index

针对来源

default

阈值类型

☒ QPS

☐ 并发线程数

单机阈值

1

是否集群

☐

流控模式

☒ 直接

☐ 关联

☐ 链路

流控效果

☒ 快速失败

☐ Warm Up

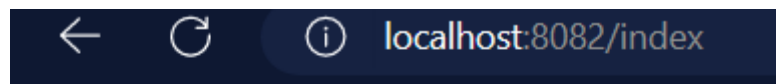
☐ 排队等待

关闭高级选项

保存

取消

此时/index这个服务，每秒只能访问一次，因为设置了QPS为1
若多次访问，结果如下



Blocked by Sentinel (flow limiting)

流控模式的区别

按照上上图，流控模式有三种，链路有点复杂，跳过

- 直接

直接就是直接限流当前的url

- 关联

关联是指当另一个路径设置QPS，但是限制的是当前的路径，比如下面的/root达到QPS为1时，/root并不会被限制，但是/index会被限制

编辑流控规则

资源名

/index

针对来源

default

阈值类型

☒ QPS

☐ 并发线程数

单机阈值

1

是否集群

☐

流控模式

☐ 直接

☒ 关联

☐ 链路

关联资源

/root

流控效果

☒ 快速失败

☐ Warm Up

☐ 排队等待

关闭高级选项

- 链路

链路简单说明就是前面两种只能控制Controller层，但是链路可以控制更深的层次，比如Service层

Sentinel熔断规则

没看懂

如下图为慢调用比例，大概效果为，在1秒内（统计时长），响应时间小于0.1毫秒（RT）的请求超过6次时（1+最小请求数的5次），则会发生10秒钟的熔断（熔断时长）

编辑熔断规则

资源名	<input type="text" value="/index"/>		
熔断策略	<input checked="" type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input type="radio"/> 异常数		
最大 RT	<input type="text" value="0.1"/>	比例阈值	<input type="text" value="0"/>
熔断时长	<input type="text" value="10"/> s	最小请求数	<input type="text" value="5"/>
统计时长	<input type="text" value="1000"/> ms		

保存

取消

除了慢调用比例，还有异常比例和异常输，这两个好理解

- 异常比例 == 发生异常的次数和请求数的比例
- 异常数 == 发生异常的次数

热点规则

其实就是流控规则的另类，热点规则是对Controller的方法的参数进行控制

先添加如下方法

```
1 @GetMapping("/hot")
2 @SentinelResource("hot")
3 public String hot(@RequestParam(value = "num1",required = false)Integer
   num1,@RequestParam(value = "num2",required = false)Integer num2){
4     return num1+"-"+num2;
5 }
```

再添加热点规则，如下就是对第一个参数进行限流，也就是上面方法的num1参数，主要是带上这个参数访问这个路径超过QPS为1的话，就会被限流

新增热点规则

资源名

hot

限流模式

QPS 模式

参数索引

0

单机阈值

1

统计窗口时长

2

秒

是否集群

☐

新增并继续添加

新增

取消

授权规则

相当于权限控制，需要另外配置类支持，略过

RocketMQ

首先在linux服务器上安装RocketMQ 4.9.4

RocketMQ是绿色安装，在官网下载压缩包并解压后，进入RocketMQ文件夹的bin目录，修改配置文件

```
hth@XTZJ-20211125UP:~/rocketmq-all-4.9.4-bin-release/bin$ ls
README.md      export.sh      mqbroker.numanode0  mqnamesrv.cmd   play.cmd        runserver.sh
cachedog.sh    mqadmin       mqbroker.numanode1  mqshutdown      play.sh         setcache.sh
cleancache.sh  mqadmin.cmd  mqbroker.numanode2  mqshutdown.cmd  runbroker.cmd  startfsrv.sh
cleancache.v1.sh mqbroker     mqbroker.numanode3  nohup.out       runbroker.sh   tools.cmd
dledger        mqbroker.cmd mqnamesrv           os.sh           runserver.cmd  tools.sh
hth@XTZJ-20211125UP:~/rocketmq-all-4.9.4-bin-release/bin$
```

分别修改runserver.sh和runbroker.sh的jvm启动内存

runbroker.sh 256m

```

82 bin
83 choose_gc_log_directory
84
85 JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m"
86 choose_gc_options
87 JAVA_OPT="${JAVA_OPT} -XX:-OmitStackTraceInFastThrow

```

runserver.sh

```

Is.sh org.apache.rocketmq.example.quickstart.Producer
choose_gc_options()
{ run -d --name rocketmq-dashboard -e "JAVA_OPTS=-Drocketmq.namesrv.addr=127.0.0.1:9876
# Example of JAVA_MAJOR_VERSION value : '1', '9', '10', '11',
# '1' means releases befor Java 9
JAVA_MAJOR_VERSION=$(("$JAVA" -version 2>&1 | sed -r -n 's/.*
if [ -z "$JAVA_MAJOR_VERSION" ] || [ "$JAVA_MAJOR_VERSION" -1
JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m -Xmn128m -X

```

启动 NameServer

nohup ./bin/mqnamesrv &

启动 Broker

nohup ./mqbroker -n localhost:9876 &

安装 RocketMQ 控制台

```

1 docker run -d --name rocketmq-dashboard -e "JAVA_OPTS=-Drocketmq.namesrv.addr=<服务器
ip>:9876" -p 8099:8080 -t apacherocketmq/rocketmq-dashboard:latest

```

简单的生产消费

添加依赖

```

1 <dependency>
2     <groupId>org.apache.rocketmq</groupId>
3     <artifactId>rocketmq-spring-boot-starter</artifactId>
4     <version>2.2.2</version>
5 </dependency>

```

生产者代码

```

2 import org.apache.rocketmq.client.producer.DefaultMQProducer;
3 import org.apache.rocketmq.client.producer.SendResult;
4 import org.apache.rocketmq.common.message.Message;
5 public class MQProducer {
6     public static void main(String[] args)
7         throws Exception {
8 //创建消息生产者
9         DefaultMQProducer producer = new DefaultMQProducer("myproducer-group");
10 //设置NameServer
11         producer.setNamesrvAddr("172.20.158.18:9876");
12 //启动生产者
13         producer.start();
14 //构建消息对象
15         Message message = new Message("myTopic","myTag",("Test MQ").getBytes());
16 //发送消息
17         SendResult result = producer.send(message, 1000);
18         System.out.println(result);
19 //关闭生产者
20         producer.shutdown();
21     }
22 }

```

消费者代码

```

1 package com.hth.provide.Test;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;
5 import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;
6 import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;
7 import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;
8 import org.apache.rocketmq.client.exception.MQClientException;
9 import org.apache.rocketmq.common.message.MessageExt;
10 import java.util.List;
11 @Slf4j
12 public class MQConsumer {
13     public static void main(String[] args) throws MQClientException {
14         //创建消息消费者
15         DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("myconsumer-group");

```

```
16 //设置NameServer
17     consumer.setNamesrvAddr("172.20.158.18:9876");
18 //指定订阅的主题和标签
19     consumer.subscribe("myTopic", "*");
20 //回调函数
21     consumer.registerMessageListener(new MessageListenerConcurrently(){
22         @Override
23         public ConsumeConcurrentlyStatus
24             consumeMessage(List<MessageExt> list, ConsumeConcurrentlyContext
consumeConcurrentlyContext) {
25             log.info("Message=>{}", list);
26             return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
27         }
28     });
29 //启动消费者
30     consumer.start();
31 }
32 }
```

Cloud Alibaba整合RocketMQ