spring by Pivotal

Spring Cloud Alibaba 参考文档

Jim Fang, Jing Xiao, Mercy Ma, Xiaolong Zuo, Bingting Peng, Yuxin Wang

1. 介绍
2. 依赖管理
3. Spring Cloud Alibaba Nacos Discovery
    3.1. 服务注册/发现: Nacos Discovery
    3.2. 如何引入 Nacos Discovery 进行服务注册/发现
    3.3. 一个使用 Nacos Discovery 进行服务注册/发现并调用的例子
        3.3.1. Nacos Server 启动
        3.3.2. Provider 应用
        3.3.3. Consumer 应用
    3.4. Nacos Discovery 对外暴露的 Endpoint
    3.5. 关于 Nacos Discovery Starter 更多的配置项信息
4. Spring Cloud Alibaba Nacos Config
    4.1. 如何引入 Nacos Config 进行配置管理
    4.2. 快速开始
        4.2.1. Nacos 服务端初始化
        4.2.2. 客户端使用方式
    4.3. 基于 DataId 为 yaml 的文件扩展名配置方式
    4.4. 支持配置的动态更新
    4.5. 支持profile粒度的配置
    4.6. 支持自定义 namespace 的配置
    4.7. 支持自定义 Group 的配置
    4.8. 支持自定义扩展的 Data Id 配置
    4.9. 配置的优先级
    4.10. Nacos Config 对外暴露的 Endpoint
    4.11. 完全关闭 Nacos Config 的自动化配置
    4.12. 关于 Nacos Config Starter 更多的配置项信息

5. Spring Cloud Alibaba Sentinel
5.1. Sentinel 介绍
5.2. 如何使用 Sentinel
Sentinel 控制台
5.2.2. 配置控制台信息
5.3. OpenFeign 支持
5.4. RestTemplate 支持
5.5. 动态数据源支持
5.6. Zuul 支持
5.7. Spring Cloud Gateway 支持
5.8. Sentinel 对外暴露的 Endpoint
5.9. 关于 Sentinel Starter 更多的配置项信息
6. Spring Cloud Alibaba Dubbo
6.1. 简介
6.2. 功能
6.3. Reference 说明
7. Spring Cloud Alibaba RocketMQ Binder
7.1. RocketMQ 介绍
7.2. RocketMQ 基本使用
7.3. Spring Cloud Stream 介绍
7.4. 如何使用 Spring Cloud Alibaba RocketMQ Binder #
7.5. Spring Cloud Alibaba RocketMQ Binder 实现
7.6. MessageSource 支持
7.7. 配置选项
7.7.1. RocketMQ Binder Properties
7.7.2. RocketMQ Consumer Properties
7.7.3. RocketMQ Provider Properties
7.8. 阿里云 MQ 服务
8. Spring Cloud AliCloud ANS
8.1. 如何引入 Spring Cloud AliCloud ANS
8.2. 使用ANS进行服务注册
8.3. 启动注册中心
8.3.1. 启动轻量版配置中心
8.3.2. 使用云上注册中心

9. Spring Cloud AliCloud ACM
9.1. 如何引入 Spring Cloud AliCloud ACM
9.2. 使用 ACM 进行配置管理
9.2.1. 启动配置中心
使用轻量版配置中心
使用阿里云配置中心
9.2.2. 在配置中心添加配置
9.2.3. 启动应用验证
9.3. 更改配置文件扩展名
9.4. 动态更新
9.5. Profile 粒度的配置
9.6. 自定义配置中心超时时间
9.7. 自定义 Group 的配置
9.8. 共享配置
9.9. Actuator 监控
10. Spring Cloud AliCloud OSS
10.1. 如何引入 Spring Cloud AliCloud OSS
10.2. 如何使用 OSS API
10.2.1. 配置 OSS
10.2.2. 引入 OSS API
10.3. 与 Spring 框架的 Resource 结合
10.4. 采用 STS 授权
10.5. 更多客户端配置
11. Spring Cloud AliCloud SchedulerX
11.1. 如何引入 Spring Cloud AliCloud SchedulerX
11.2. 启动SchedulerX任务调度
11.3. 编写一个简单任务
11.4. 对任务进行调度
11.5. 生产环境使用
12. Spring Cloud AliCloud SMS
12.1. 如何引入 Spring Cloud AliCloud SMS
12.2. 如何使用 SMS API
12.2.1. 配置 SMS
12.2.2. 引入 SMS API

12.3. SMS Api 的高级功能
13. Spring Cloud Alibaba Sidecar
13.1. 术语
13.1.1. 异构微服务
13.1.2. ``完美整合"的三层含义
13.2. Why or Why not?
13.2.1. 为什么要编写Alibaba Sidecar?
13.2.2. 为什么不用Service Mesh?
13.3. 原理
13.4. 要求
13.5. 使用示例
13.5.1. 示例代码（以Nacos服务发现为例）
13.5.2. 异构微服务
13.5.3. 测试
测试1：Spring Cloud微服务完美调用异构微服务
测试2：异构微服务完美调用Spring Cloud微服务
13.6. Alibaba Sidecar优缺点分析

1.介绍

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。

依托 Spring Cloud Alibaba，您只需要添加一些注解和少量配置，就可以将 Spring Cloud 应用接入阿里分布式应用解决方案，通过阿里中间件来迅速搭建分布式应用系统。

目前 Spring Cloud Alibaba 提供了如下功能:

1. **服务限流降级**：支持 WebServlet、WebFlux, OpenFeign、RestTemplate、Dubbo 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
2. **服务注册与发现**：适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持。
3. **分布式配置管理**：支持分布式系统中的外部化配置，配置更改时自动刷新。
4. **Rpc服务**：扩展 Spring Cloud 客户端 RestTemplate 和 OpenFeign，支持调用 Dubbo RPC 服务

5. **消息驱动能力**：基于 Spring Cloud Stream 为微服务应用构建消息驱动能力。
6. **分布式事务**：使用 @GlobalTransactional 注解， 高效并且对业务零侵入地解决分布式事务问题。
7. **阿里云对象存储**：阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
8. **分布式任务调度**：提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker（schedulerx-client）上执行。
9. **阿里云短信服务**：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

Spring Cloud Alibaba 也提供了丰富的 examples。

---

2. 依赖管理

Spring Cloud Alibaba BOM 包含了它所使用的所有依赖的版本。

如果您是 Maven Central 用户，请将我们的 BOM 添加到您的 pom.xml 中的 <dependencyManagement> 部分。 这将允许您省略任何Maven依赖项的版本，而是将版本控制委派给BOM。

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.1.1.BUILD-SNAPSHOT</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

在下面的章节中，假设您使用的是 Spring Cloud Alibaba bom，相关 starter 依赖将不包含版本号。

---

3. Spring Cloud Alibaba Nacos Discovery

Nacos 是一个 Alibaba 开源的、易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

使用 Spring Cloud Alibaba Nacos Discovery，可基于 Spring Cloud 的编程模型快速接入 Nacos 服务注册功能。

3.1. 服务注册/发现: Nacos Discovery

服务发现是微服务架构体系中最关键的组件之一。如果尝试着用手动的方式来给每一个客户端来配置所有服务提供者的服务列表是一件非常困难的事，而且也不利于 服务的动态扩缩容。Nacos Discovery 可以帮助您将服务自动注册到 Nacos 服务端并且能够动态感知和刷新某个服务实例的服务列表。除此之外，Nacos Discovery 也将服务实例自身的一些元数据信息-例如 host，port, 健康检查URL，主页等内容注册到 Nacos。Nacos 的获取和启动方式可以参考 Nacos 官网。

3.2. 如何引入 Nacos Discovery 进行服务注册/发现

如果要在您的项目中使用 Nacos 来实现服务注册/发现，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alibaba-nacos-discovery` 的 starter。

XML

```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

3.3. 一个使用 Nacos Discovery 进行服务注册/发现并调用的例子

Nacos Discovery 适配了 Netflix Ribbon，可以使用 RestTemplate 或 OpenFeign 进行服务的调用。

3.3.1. Nacos Server 启动

具体启动方式参考 Nacos 官网。

Nacos Server 启动后，进入 http://ip:8848 查看控制台(默认账号名/密码为 nacos/nacos):

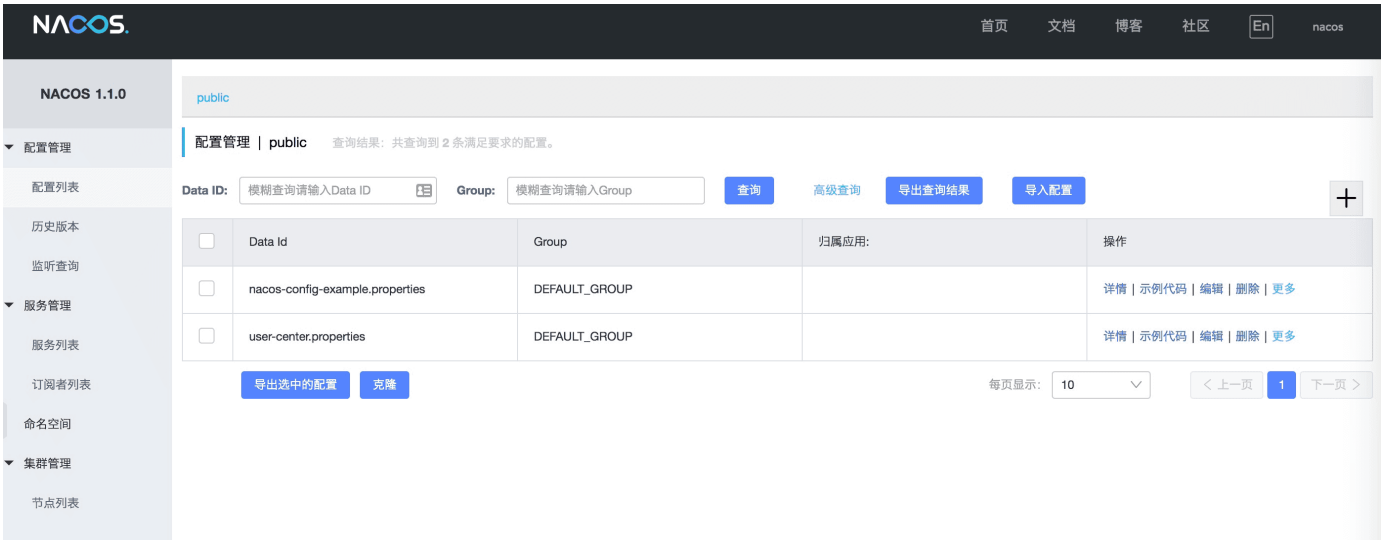

Figure 1. Nacos Dashboard

关于更多的 Nacos Server 版本，可以从 release 页面 下载最新的版本。

3.3.2. Provider 应用

以下步骤向您展示了如何将一个服务注册到 Nacos。

- pom.xml的配置。一个完整的 pom.xml 配置如下所示：

pom.xml

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mave
    <modelVersion>4.0.0</modelVersion>

    <groupId>open.source.test</groupId>
    <artifactId>nacos-discovery-test</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>nacos-discovery-test</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>${spring.boot.version}</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring.cloud.version}</version>
```

```
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <dependency>
                <groupId>com.alibaba.cloud</groupId>
                <artifactId>spring-cloud-alibaba-dependencies</artifactId>
                <version>${spring.cloud.alibaba.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>

        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

- application.properties 配置。一些关于 Nacos 基本的配置也必须在 application.properties(也可以是application.yaml)配置，如下所示：

application.properties

```
PROPERTIES

server.port=8081
spring.application.name=nacos-provider
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
management.endpoints.web.exposure.include=*
```

如果不想使用 Nacos 作为您的服务注册与发现，可以将 `spring.cloud.nacos.discovery` 设置为 `false` 。

- 启动 Provider 示例。如下所示：

JAVA

```java
@SpringBootApplication
@EnableDiscoveryClient
public class NacosProviderDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(NacosProviderDemoApplication.class, args);
    }

    @RestController
    public class EchoController {
        @GetMapping(value = "/echo/{string}")
        public String echo(@PathVariable String string) {
            return "Hello Nacos Discovery " + string;
        }
    }
}
```

这个时候你就可以在 Nacos的控制台上看到注册上来的服务信息了。

3.3.3. Consumer 应用

Consumer 应用可能还没像启动一个 Provider 应用那么简单。因为在 Consumer 端需要去调用 Provider 端提供的REST 服务。例子中我们使用最原始的一种方式， 即显示的使用 LoadBalanceClient 和 RestTemplate 结合的方式来访问。 pom.xml 和 application.properties 的配置可以参考 1.2 小结。启动一个 Consumer应用的示例代码如下所示：

通过带有负载均衡的RestTemplate 和 FeignClient 也是可以访问的。

JAVA

```java
@SpringBootApplication
@EnableDiscoveryClient
public class NacosConsumerApp {

    @RestController
    public class NacosController{

        @Autowired
        private LoadBalancerClient loadBalancerClient;
        @Autowired
        private RestTemplate restTemplate;

        @Value("${spring.application.name}")
        private String appName;

        @GetMapping("/echo/app-name")
        public String echoAppName(){
```

```
            //使用 LoadBalanceClient 和 RestTemplate 结合的方式来访问
            ServiceInstance serviceInstance = loadBalancerClient.choose("nacos-provider");
            String url = String.format("http://%s:%s/echo/%s",serviceInstance.getHost(),service
            System.out.println("request url:"+url);
            return restTemplate.getForObject(url,String.class);
        }

    }

    //实例化 RestTemplate 实例
    @Bean
    public RestTemplate restTemplate(){

        return new RestTemplate();
    }

    public static void main(String[] args) {

        SpringApplication.run(NacosConsumerApp.class,args);
    }
}
```

这个例子中我们注入了一个 LoadBalancerClient 的实例，并且手动的实例化一个 RestTemplate，同时将 `spring.application.name` 的配置值 注入到应用中来， 目的是调用 Provider 提供的服务时，希望将当前配置的应用名给显示出来。

在启动 Consumer 应用之前请先将 Nacos 服务启动好。具体启动方式可参考 Nacos 官网。

启动后，访问 Consumer 提供出来的 `http://ip:port/echo/app-name` 接口。我这里测试启动的 port 是 8082。访问结果如下所示：

```
访问地址：http://127.0.0.1:8082/echo/app-name
访问结果：Hello Nacos Discovery nacos-consumer
```

3.4. Nacos Discovery 对外暴露的 Endpoint

Nacos Discovery 内部提供了一个 Endpoint, 对应的 endpoint id 为 `nacos-discovery` 。

Endpoint 暴露的 json 中包含了两种属性:

1. subscribe: 显示了当前服务有哪些服务订阅者
2. NacosDiscoveryProperties: 当前应用 Nacos 的基础配置信息

这是 Endpoint 暴露的 json 示例:

JSON


```
{
  "subscribe": [
    {
      "jsonFromServer": "",
      "name": "nacos-provider",
      "clusters": "",
      "cacheMillis": 10000,
      "hosts": [
        {
          "instanceId": "30.5.124.156#8081#DEFAULT#nacos-provider",
          "ip": "30.5.124.156",
          "port": 8081,
          "weight": 1.0,
          "healthy": true,
          "enabled": true,
          "cluster": {
            "serviceName": null,
            "name": null,
            "healthChecker": {
              "type": "TCP"
            },
            "defaultPort": 80,
            "defaultCheckPort": 80,
            "useIPPort4Check": true,
            "metadata": {

            }
          },
          "service": null,
          "metadata": {

          }
        }
      ],
      "lastRefTime": 1541755293119,
      "checksum": "e5a699c9201f5328241c178e804657e11541755293119",
      "allIPs": false,
      "key": "nacos-provider",
      "valid": true
    }
  ],
  "NacosDiscoveryProperties": {
    "serverAddr": "127.0.0.1:8848",
    "endpoint": "",
    "namespace": "",
    "logName": "",
    "service": "nacos-provider",
    "weight": 1.0,
    "clusterName": "DEFAULT",
    "metadata": {

    },
    "registerEnabled": true,
    "ip": "30.5.124.201",
    "networkInterface": "",
    "port": 8082,
    "secure": false,
    "accessKey": "",
    "secretKey": ""
  }
}
```

3.5. 关于 Nacos Discovery Starter 更多的配置项信息

更多关于 Nacos Discovery Starter 的配置项如下所示:

配置项	Key	默认值	说明
服务端地址	spring.cloud.nacos.discovery.server-addr		Nacos Server 启动监听的ip地址和端口
服务名	spring.cloud.nacos.discovery.service	${spring.application.name}	注册的服务名
权重	spring.cloud.nacos.discovery.weight	1	取值范围 1 到 100，数值越大，权重越大
网卡名	spring.cloud.nacos.discovery.network-interface		当IP未配置时，注册的IP为此网卡所对应的IP地址，如果此项也未配置，则默认取第一块网卡的地址
注册的IP地址	spring.cloud.nacos.discovery.ip		优先级最高
注册的端口	spring.cloud.nacos.discovery.port	-1	默认情况下不用配置，会自动探测
命名空间	spring.cloud.nacos.discovery.namespace		常用场景之一是不同环境的注册的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等
AccessKey	spring.cloud.nacos.discovery.access-key		当要上阿里云时，阿里云上面的一个云账号名
SecretKey	spring.cloud.nacos.discovery.secret-key		当要上阿里云时，阿里云上面的一个云账号密码
Metadata	spring.cloud.nacos.d		使用Map格式配置，

	iscovery.metadata		用户可以根据自己的需要自定义一些和服务相关的元数据信息
日志文件名	spring.cloud.nacos.discovery.log-name		
集群	spring.cloud.nacos.discovery.cluster-name	DEFAULT	Nacos集群名称
接入点	spring.cloud.nacos.discovery.endpoint		地域的某个服务的入口域名，通过此域名可以动态地拿到服务端地址
是否集成Ribbon	ribbon.nacos.enabled	true	一般都设置成true即可
是否开启Nacos Watch	spring.cloud.nacos.discovery.watch.enabled	true	可以设置成false来关闭 watch

4. Spring Cloud Alibaba Nacos Config

Nacos 是一个 Alibaba 开源的、易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

使用 Spring Cloud Alibaba Nacos Config，可基于 Spring Cloud 的编程模型快速接入 Nacos 配置管理功能。

4.1. 如何引入 Nacos Config 进行配置管理

如果要在您的项目中使用 Nacos 来实现配置管理，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alibaba-nacos-config` 的 starter。

XML

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

4.2. 快速开始

Nacos Config 使用 DataId 和 GROUP 确定一个配置。

下图表示 DataId 使用 `myDataid`，GROUP 使用 `DEFAULT_GROUP`， 配置格式为 Properties 的一个配置项:

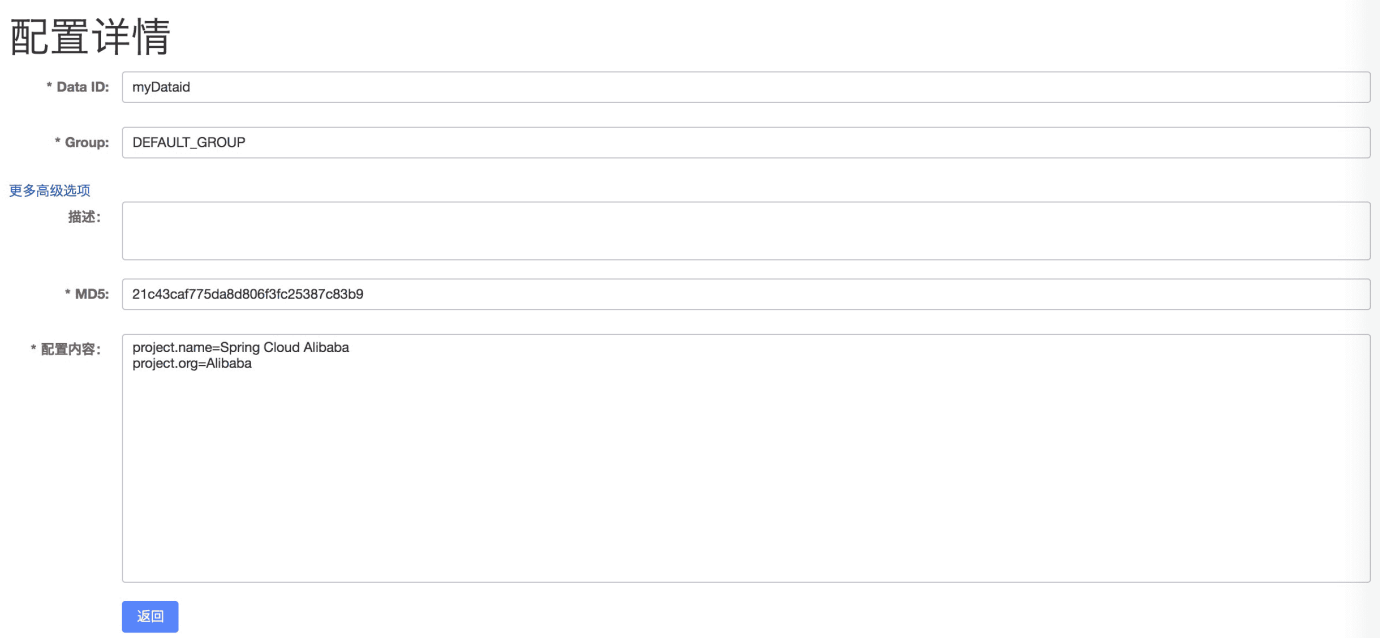

Figure 2. Nacos Config Item

4.2.1. Nacos 服务端初始化

具体启动方式参考 Spring Cloud Alibaba Nacos Discovery 小节的 "Nacos Server 启动" 章节。

Nacos Server 启动完毕后，添加如何配置:

```
Data ID:    nacos-config.properties

Group   :   DEFAULT_GROUP

配置格式:    Properties

配置内容:    user.name=nacos-config-properties
            user.age=90
```

注意DataId是以 properties(默认的文件扩展名方式)为扩展名。

4.2.2. 客户端使用方式

如果要在您的项目中使用 Nacos 来实现应用的外部化配置，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alibaba-nacos-config` 的 starter。

XML

```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

现在创建一个标准的 Spring Boot 应用。

JAVA

```java
@SpringBootApplication
public class NacosConfigApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApplication.run(ConfigApplica
        String userName = applicationContext.getEnvironment().getProperty("user.name");
        String userAge = applicationContext.getEnvironment().getProperty("user.age");
        System.err.println("user name :"+userName+"; age: "+userAge);
    }
}
```

在运行此 NacosConfigApplication 之前， 必须使用 `bootstrap.properties` 配置文件来配置 Nacos Server 地址，例如：

bootstrap.properties

PROPERTIES

```properties
# DataId 默认使用 `spring.application.name` 配置跟文件扩展名结合(配置格式默认使用 properties), G
spring.application.name=nacos-config
spring.cloud.nacos.config.server-addr=127.0.0.1:8848
```

注意当你使用域名的方式来访问 Nacos 时， `spring.cloud.nacos.config.server-addr` 配置的方式为 `域名:port` 。 例如 Nacos 的域名为abc.com.nacos，监听的端口为 80，则 `spring.cloud.nacos.config.server-addr=abc.com.nacos:80` 。 注意 80 端口不能省略。

启动这个 Example，可以看到如下输出结果：

```
2018-11-02 14:24:51.638  INFO 32700 --- [main] c.a.demo.provider.ConfigApplication    : Started
user name :nacos-config-properties; age: 90
2018-11-02 14:24:51.688  INFO 32700 --- [-127.0.0.1:8848] s.c.a.AnnotationConfigApplicationCont
```

4.3. 基于 DataId 为 yaml 的文件扩展名配置方式

Nacos Config 除了支持 properties 格式以外，也支持 yaml 格式。这个时候只需要完成以下两步：

1、在应用的 bootstrap.properties 配置文件中显示的声明 DataId 文件扩展名。如下所示

bootstrap.properties

YAML

```
spring.cloud.nacos.config.file-extension=yaml
```

2、在 Nacos 的控制台新增一个DataId为yaml为扩展名的配置，如下所示：

```
Data ID:        nacos-config.yaml

Group   :       DEFAULT_GROUP

配置格式:        YAML

配置内容:        user.name: nacos-config-yaml
               user.age: 68
```

这两步完成后，重启测试程序，可以看到如下输出结果。

```
2018-11-02 14:59:00.484  INFO 32928 --- [main] c.a.demo.provider.ConfigApplication:Started Conf
user name :nacos-config-yaml; age: 68
2018-11-02 14:59:00.529  INFO 32928 --- [-127.0.0.1:8848] s.c.a.AnnotationConfigApplicationCont
```

4.4. 支持配置的动态更新

Nacos Config 默认支持配置的动态更新，启动 Spring Boot 应用测试的代码如下：

JAVA

```
@SpringBootApplication
public class ConfigApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApplication.run(ConfigApplica
        while(true) {
            //当动态配置刷新时，会更新到 Enviroment中，因此这里每隔一秒中从Enviroment中获取配置
            String userName = applicationContext.getEnvironment().getProperty("user.name");
            String userAge = applicationContext.getEnvironment().getProperty("user.age");
            System.err.println("user name :" + userName + "; age: " + userAge);
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

如下所示，当变更user.name时，应用程序中能够获取到最新的值：

```
user name :nacos-config-yaml; age: 68
user name :nacos-config-yaml; age: 68
user name :nacos-config-yaml; age: 68
2018-11-02 15:04:25.069  INFO 32957 --- [-127.0.0.1:8848] o.s.boot.SpringApplication
2018-11-02 15:04:25.070  INFO 32957 --- [-127.0.0.1:8848] s.c.a.AnnotationConfigApplicationCont
2018-11-02 15:04:25.071  INFO 32957 --- [-127.0.0.1:8848] s.c.a.AnnotationConfigApplicationCont
//从 Enviroment 中 读取到更改后的值
user name :nacos-config-yaml-update; age: 68
user name :nacos-config-yaml-update; age: 68
```

你可以通过配置 `spring.cloud.nacos.config.refresh.enabled=false` 来关闭动态刷新

4.5. 支持profile粒度的配置

Nacos Config 在加载配置的时候，不仅仅加载了以 DataId 为 `${spring.application.name}.${file-extension:properties}` 为前缀的基础配置，还加载了DataId为 `${spring.application.name}-${profile}.${file-extension:properties}` 的基础配置。在日常开发中如果遇到多套环境下的不同配置，可以通过Spring 提供的 `${spring.profiles.active}` 这个配置项来配置。

PROPERTIES

```
spring.profiles.active=develop
```

${spring.profiles.active} 当通过配置文件来指定时必须放在 bootstrap.properties 文件中。

Nacos 上新增一个DataId为：nacos-config-develop.yaml的基础配置，如下所示：

```
Data ID:        nacos-config-develop.yaml
Group   :       DEFAULT_GROUP
配置格式:        YAML
配置内容:        current.env: develop-env
```

启动 Spring Boot 应用测试的代码如下：

JAVA

```java
@SpringBootApplication
public class ConfigApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApplication.run(ConfigApplica
        while(true) {
            String userName = applicationContext.getEnvironment().getProperty("user.name");
            String userAge = applicationContext.getEnvironment().getProperty("user.age");
            //获取当前部署的环境
            String currentEnv = applicationContext.getEnvironment().getProperty("current.env");
            System.err.println("in "+currentEnv+" enviroment; "+"user name :" + userName + "; a
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

启动后，可见控制台的输出结果：

```
in develop-env enviroment; user name :nacos-config-yaml-update; age: 68
2018-11-02 15:34:25.013  INFO 33014 --- [ Thread-11] ConfigServletWebServerApplicationContext :
```

如果需要切换到生产环境，只需要更改 `${spring.profiles.active}` 参数配置即可。如下所示：

PROPERTIES

```properties
spring.profiles.active=product
```

同时生产环境上 Nacos 需要添加对应 DataId 的基础配置。例如，在生成环境下的 Naocs 添加了 DataId为：nacos-config-product.yaml的配置：

```
Data ID:        nacos-config-product.yaml

Group   :       DEFAULT_GROUP

配置格式:        YAML

配置内容:        current.env: product-env
```

启动测试程序，输出结果如下：

```
in product-env enviroment; user name :nacos-config-yaml-update; age: 68
2018-11-02 15:42:14.628  INFO 33024 --- [Thread-11] ConfigServletWebServerApplicationContext :
```

此案例中我们通过 `spring.profiles.active=<profilename>` 的方式写死在配置文件中，而在真

正的项目实施过程中这个变量的值是需要不同环境而有不同的值。这个时候通常的做法是通过 `-Dspring.profiles.active=<profile>` 参数指定其配置来达到环境间灵活的切换。

4.6. 支持自定义 namespace 的配置

Nacos 内部有 Namespace 的概念:

> 用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

在没有明确指定 `${spring.cloud.nacos.config.namespace}` 配置的情况下， 默认使用的是 Nacos 上 Public 这个namespae。如果需要使用自定义的命名空间，可以通过以下配置来实现：

```properties
spring.cloud.nacos.config.namespace=b3404bc0-d7dc-4855-b519-570ed34b62d7
```

该配置必须放在 bootstrap.properties 文件中。此外 `spring.cloud.nacos.config.namespace` 的值是 namespace 对应的 id，id 值可以在 Nacos 的控制台获取。并且在添加配置时注意不要选择其他的 namespae，否则将会导致读取不到正确的配置。

4.7. 支持自定义 Group 的配置

在没有明确指定 `${spring.cloud.nacos.config.group}` 配置的情况下， 默认使用的是 DEFAULT_GROUP 。如果需要自定义自己的 Group，可以通过以下配置来实现：

```properties
spring.cloud.nacos.config.group=DEVELOP_GROUP
```

该配置必须放在 bootstrap.properties 文件中。并且在添加配置时 Group 的值一定要和 `spring.cloud.nacos.config.group` 的配置值一致。

4.8. 支持自定义扩展的 Data Id 配置

Nacos Config 从 0.2.1 版本后，可支持自定义 Data Id 的配置。关于这部分详细的设计可参考 这里。一个完整的配置案例如下所示：

```properties
spring.application.name=opensource-service-provider
spring.cloud.nacos.config.server-addr=127.0.0.1:8848

# config external configuration
# 1、Data Id 在默认的组 DEFAULT_GROUP,不支持配置的动态刷新
spring.cloud.nacos.config.ext-config[0].data-id=ext-config-common01.properties

# 2、Data Id 不在默认的组，不支持动态刷新
spring.cloud.nacos.config.ext-config[1].data-id=ext-config-common02.properties
spring.cloud.nacos.config.ext-config[1].group=GLOBALE_GROUP

# 3、Data Id 既不在默认的组，也支持动态刷新
spring.cloud.nacos.config.ext-config[2].data-id=ext-config-common03.properties
spring.cloud.nacos.config.ext-config[2].group=REFRESH_GROUP
spring.cloud.nacos.config.ext-config[2].refresh=true
```

可以看到:

- 通过 `spring.cloud.nacos.config.ext-config[n].data-id` 的配置方式来支持多个 Data Id 的配置。
- 通过 `spring.cloud.nacos.config.ext-config[n].group` 的配置方式自定义 Data Id 所在的组，不明确配置的话，默认是 DEFAULT_GROUP。
- 通过 `spring.cloud.nacos.config.ext-config[n].refresh` 的配置方式来控制该 Data Id 在配置变更时，是否支持应用中可动态刷新， 感知到最新的配置值。默认是不支持的。

> 多个 Data Id 同时配置时，他的优先级关系是 `spring.cloud.nacos.config.ext-config[n].data-id` 其中 n 的值越大，优先级越高。

> `spring.cloud.nacos.config.ext-config[n].data-id` 的值必须带文件扩展名，文件扩展名既可支持 properties，又可以支持 yaml/yml。 此时 `spring.cloud.nacos.config.file-extension` 的配置对自定义扩展配置的 Data Id 文件扩展名没有影响。

通过自定义扩展的 Data Id 配置，既可以解决多个应用间配置共享的问题，又可以支持一个应用有多个配置文件。

为了更加清晰的在多个应用间配置共享的 Data Id ，你可以通过以下的方式来配置：

```properties
spring.cloud.nacos.config.shared-dataids=bootstrap-common.properties,all-common.properties
spring.cloud.nacos.config.refreshable-dataids=bootstrap-common.properties
```

可以看到：

- 通过 `spring.cloud.nacos.config.shared-dataids` 来支持多个共享 Data Id 的配置，多个之间用逗号隔开。
- 通过 `spring.cloud.nacos.config.refreshable-dataids` 来支持哪些共享配置的 Data Id 在配置变化时，应用中是否可动态刷新， 感知到最新的配置值，多个 Data Id 之间用逗号隔开。如果没有明确配置，默认情况下所有共享配置的 Data Id 都不支持动态刷新。

通过 `spring.cloud.nacos.config.shared-dataids` 来支持多个共享配置的 Data Id 时， 多个共享配置间的一个优先级的关系我们约定：按照配置出现的先后顺序，即后面的优先级要高于前面。

通过 `spring.cloud.nacos.config.shared-dataids` 来配置时，Data Id 必须带文件扩展名，文件扩展名既可支持 properties，也可以支持 yaml/yml。 此时 `spring.cloud.nacos.config.file-extension` 的配置对自定义扩展配置的 Data Id 文件扩展名没有影响。

`spring.cloud.nacos.config.refreshable-dataids` 给出哪些需要支持动态刷新时，Data Id 的值也必须明确给出文件扩展名。

4.9. 配置的优先级

Nacos Config 目前提供了三种配置能力从 Nacos 拉取相关的配置

- A: 通过 `spring.cloud.nacos.config.shared-dataids` 支持多个共享 Data Id 的配置
- B: 通过 `spring.cloud.nacos.config.ext-config[n].data-id` 的方式支持多个扩展 Data Id 的配置
- C: 通过内部相关规则(应用名、应用名+ Profile)自动生成相关的 Data Id 配置

当三种方式共同使用时，他们的一个优先级关系是:A < B < C

4.10. Nacos Config 对外暴露的 Endpoint

Nacos Config 内部提供了一个 Endpoint, 对应的 endpoint id 为 `nacos-config` 。

Endpoint 暴露的 json 中包含了三种属性:

1. Sources: 当前应用配置的数据信息
2. RefreshHistory: 配置刷新的历史记录
3. NacosConfigProperties: 当前应用 Nacos 的基础配置信息

这是 Endpoint 暴露的 json 示例:

JSON


```
{
    "NacosConfigProperties": {
        "serverAddr": "127.0.0.1:8848",
        "encode": null,
        "group": "DEFAULT_GROUP",
        "prefix": null,
        "fileExtension": "properties",
        "timeout": 3000,
        "endpoint": null,
        "namespace": null,
        "accessKey": null,
        "secretKey": null,
        "contextPath": null,
        "clusterName": null,
        "name": null,
        "sharedDataids": "base-common.properties,common.properties",
        "refreshableDataids": "common.properties",
        "extConfig": null
    },
    "RefreshHistory": [{
        "timestamp": "2019-07-29 11:20:04",
        "dataId": "nacos-config-example.properties",
        "md5": "7d5d7f1051ff6571e2ec9f90887d9d91"
    }],
    "Sources": [{
        "lastSynced": "2019-07-29 11:19:04",
        "dataId": "common.properties"
    }, {
        "lastSynced": "2019-07-29 11:19:04",
        "dataId": "base-common.properties"
    }, {
        "lastSynced": "2019-07-29 11:19:04",
        "dataId": "nacos-config-example.properties"
    }]
}
```


4.11. 完全关闭 Nacos Config 的自动化配置

通过设置 spring.cloud.nacos.config.enabled = false 来完全关闭 Spring Cloud Nacos Config

4.12. 关于 Nacos Config Starter 更多的配置项信息

更多关于 Nacos Config Starter 的配置项如下所示:

配置项	Key	默认值	说明
服务端地址	spring.cloud.nacos.config.server-addr		Nacos Server 启动监听的ip地址和端口
配置对应的 DataId	spring.cloud.nacos.config.name		先取 prefix，再去 name，最后取 spring.application.name
配置对应的 DataId	spring.cloud.nacos.config.prefix		先取 prefix，再去 name，最后取 spring.application.name
配置内容编码	spring.cloud.nacos.config.encode		读取的配置内容对应的编码
GROUP	spring.cloud.nacos.config.group	DEFAULT_GROUP	配置对应的组
文件扩展名	spring.cloud.nacos.config.fileExtension	properties	配置项对应的文件扩展名，目前支持 properties 和 yaml(yml)
获取配置超时时间	spring.cloud.nacos.config.timeout	3000	客户端获取配置的超时时间(毫秒)
接入点	spring.cloud.nacos.config.endpoint		地域的某个服务的入口域名，通过此域名可以动态地拿到服务端地址
命名空间	spring.cloud.nacos.config.namespace		常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等
AccessKey	spring.cloud.nacos.config.accessKey		当要上阿里云时，阿里云上面的一个云账号名
SecretKey	spring.cloud.nacos.		当要上阿里云时，阿里云

	config.secretKey		上面的一个云账号密码
Nacos Server 对应的 context path	spring.cloud.nacos.config.contextPath		Nacos Server 对外暴露的 context path
集群	spring.cloud.nacos.config.clusterName		配置成Nacos集群名称
共享配置	spring.cloud.nacos.config.sharedDataids		共享配置的 DataId，"," 分割
共享配置动态刷新	spring.cloud.nacos.config.refreshableDataids		共享配置中需要动态刷新的 DataId，"," 分割
自定义 Data Id 配置	spring.cloud.nacos.config.extConfig		属性是个集合，内部由 `Config` POJO 组成。`Config` 有 3 个属性，分别是 `dataId`，`group` 以及 `refresh`

5. Spring Cloud Alibaba Sentinel

5.1. Sentinel 介绍

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。 Sentinel 以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。

Sentinel 具有以下特征:

- **丰富的应用场景**： Sentinel 承接了阿里巴巴近 10 年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、实时熔断下游不可用应用等。
- **完备的实时监控**： Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至 500 台以下规模的集群的汇总运行情况。
- **广泛的开源生态**： Sentinel 提供开箱即用的与其它开源框架/库的整合模块，例如与 Spring Cloud、Dubbo、gRPC 的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。

- **完善的 SPI 扩展点**： Sentinel 提供简单易用、完善的 SPI 扩展点。您可以通过实现扩展点，快速的定制逻辑。例如定制规则管理、适配数据源等。

5.2. 如何使用 Sentinel

如果要在您的项目中引入 Sentinel，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alibaba-sentinel` 的 starter。

XML

```
[source,yaml]
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

下面这个例子就是一个最简单的使用 Sentinel 的例子:

JAVA

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(ServiceApplication.class, args);
    }

}

@RestController
public class TestController {

    @GetMapping(value = "/hello")
    @SentinelResource("hello")
    public String hello() {
        return "Hello Sentinel";
    }

}
```

@SentinelResource 注解用来标识资源是否被限流、降级。上述例子上该注解的属性 'hello' 表示资源名。

@SentinelResource 还提供了其它额外的属性如 `blockHandler`， `blockHandlerClass`， `fallback` 用于表示限流或降级的操作，更多内容可以参考 Sentinel注解支持。

以上例子都是在 WebServlet 环境下使用的，Sentinel 目前已经支持 WebFlux，需要配合 `spring-boot-starter-webflux` 依赖触发 sentinel-starter 中 WebFlux 相关的自动化配置。

JAVA

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
```

```
        SpringApplication.run(ServiceApplication.class, args);
    }

}

@RestController
public class TestController {

    @GetMapping("/mono")
    @SentinelResource("hello")
    public Mono<String> mono() {
    return Mono.just("simple string")
            .transform(new SentinelReactorTransformer<>("otherResourceName"));
    }

}
```

Sentinel 控制台

Sentinel 控制台提供一个轻量级的控制台，它提供机器发现、单机资源实时监控、集群资源汇总，以及规则管理的功能。您只需要对应用进行简单的配置，就可以使用这些功能。

注意: 集群资源汇总仅支持 500 台以下的应用集群，有大概 1 - 2 秒的延时。

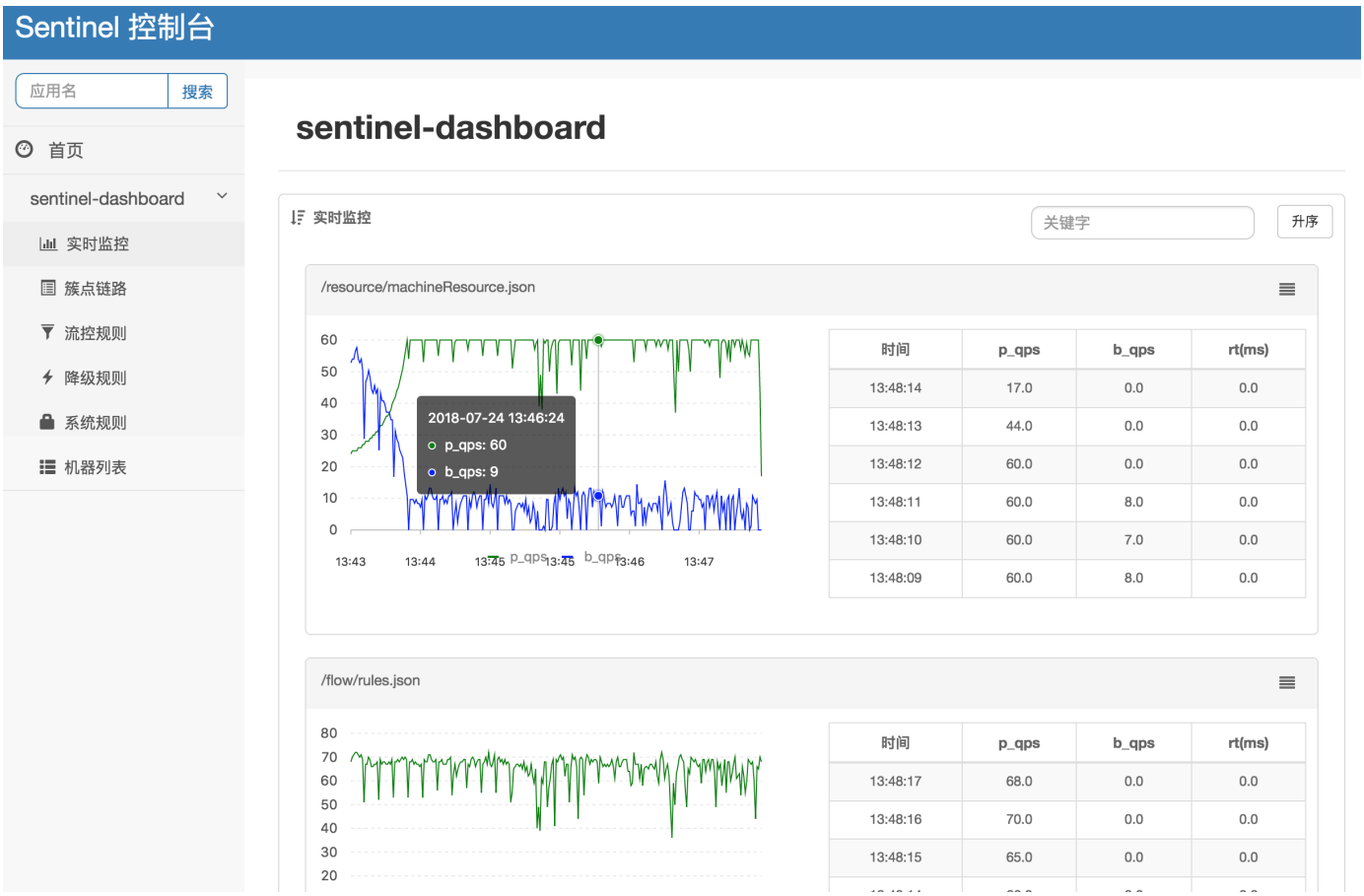

Figure 3. Sentinel Dashboard

开启该功能需要3个步骤：

获取控制台

您可以从 release 页面 下载最新版本的控制台 jar 包。

您也可以从最新版本的源码自行构建 Sentinel 控制台：

- 下载 控制台 工程
- 使用以下命令将代码打包成一个 fat jar: `mvn clean package`

启动控制台

Sentinel 控制台是一个标准的 SpringBoot 应用，以 SpringBoot 的方式运行 jar 包即可。

```shell
java -Dserver.port=8080 -Dcsp.sentinel.dashboard.server=localhost:8080 -Dproject.name=sentinel-
```

如若8080端口冲突，可使用 `-Dserver.port=新端口` 进行设置。

5.2.2. 配置控制台信息

application.yml

```
spring:
  cloud:
    sentinel:
      transport:
        port: 8719
        dashboard: localhost:8080
```

这里的 `spring.cloud.sentinel.transport.port` 端口配置会在应用对应的机器上启动一个 Http Server，该 Server 会与 Sentinel 控制台做交互。比如 Sentinel 控制台添加了1个限流规则，会把规则数据 push 给这个 Http Server 接收，Http Server 再将规则注册到 Sentinel 中。

更多 Sentinel 控制台的使用及问题参考： Sentinel控制台

5.3. OpenFeign 支持

Sentinel 适配了 OpenFeign 组件。如果想使用，除了引入 `sentinel-starter` 的依赖外还需要 2 个步骤：

- 配置文件打开 sentinel 对 feign 的支持： `feign.sentinel.enabled=true`
- 加入 `openfeign starter` 依赖使 `sentinel starter` 中的自动化配置类生效：

XML

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

这是一个 `FeignClient` 的简单使用示例：

JAVA

```java
@FeignClient(name = "service-provider", fallback = EchoServiceFallback.class, configuration = F
public interface EchoService {
    @GetMapping(value = "/echo/{str}")
    String echo(@PathVariable("str") String str);
}

class FeignConfiguration {
    @Bean
    public EchoServiceFallback echoServiceFallback() {
        return new EchoServiceFallback();
    }
}

class EchoServiceFallback implements EchoService {
    @Override
    public String echo(@PathVariable("str") String str) {
        return "echo fallback";
    }
}
```

Feign 对应的接口中的资源名策略定义：httpmethod:protocol://requesturl。`@FeignClient` 注解中的所有属性，Sentinel 都做了兼容。

`EchoService` 接口中方法 `echo` 对应的资源名为 `GET:http://service-provider/echo/{str}`。

5.4. RestTemplate 支持

Spring Cloud Alibaba Sentinel 支持对 `RestTemplate` 的服务调用使用 Sentinel 进行保护，在构造 `RestTemplate` bean的时候需要加上 `@SentinelRestTemplate` 注解。

JAVA

```java
@Bean
@SentinelRestTemplate(blockHandler = "handleException", blockHandlerClass = ExceptionUtil.class
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

`@SentinelRestTemplate` 注解的属性支持限流(`blockHandler` , `blockHandlerClass`)和降级(`fallback` , `fallbackClass`)的处理。

其中 `blockHandler` 或 `fallback` 属性对应的方法必须是对应 `blockHandlerClass` 或 `fallbackClass` 属性中的静态方法。

该方法的参数跟返回值跟 `org.springframework.http.client.ClientHttpRequestInterceptor#interceptor` 方法一致，其中参数多出了一个 `BlockException` 参数用于获取 Sentinel 捕获的异常。

比如上述 `@SentinelRestTemplate` 注解中 `ExceptionUtil` 的 `handleException` 属性对应的方法声明如下：

```java
public class ExceptionUtil {
    public static ClientHttpResponse handleException(HttpRequest request, byte[] body, ClientHt
        ...
    }
}
```

> 应用启动的时候会检查 `@SentinelRestTemplate` 注解对应的限流或降级方法是否存在，如不存在会抛出异常

`@SentinelRestTemplate` 注解的限流(`blockHandler` , `blockHandlerClass`)和降级(`fallback` , `fallbackClass`)属性不强制填写。

当使用 `RestTemplate` 调用被 Sentinel 熔断后，会返回 `RestTemplate request block by sentinel` 信息，或者也可以编写对应的方法自行处理返回信息。这里提供了 `SentinelClientHttpResponse` 用于构造返回信息。

Sentinel RestTemplate 限流的资源规则提供两种粒度：

- `httpmethod:schema://host:port/path` ： 协议、主机、端口和路径
- `httpmethod:schema://host:port` ： 协议、主机和端口

> 以 `https://www.taobao.com/test` 这个 url 并使用 GET 方法为例。对应的资源名有两种粒度，分别是 `GET:https://www.taobao.com` 以及 `GET:https://www.taobao.com/test`

5.5. 动态数据源支持

`SentinelProperties` 内部提供了 `TreeMap` 类型的 `datasource` 属性用于配置数据源信息。

比如配置 4 个数据源：

```properties
spring.cloud.sentinel.datasource.ds1.file.file=classpath: degraderule.json
spring.cloud.sentinel.datasource.ds1.file.rule-type=flow

#spring.cloud.sentinel.datasource.ds1.file.file=classpath: flowrule.json
#spring.cloud.sentinel.datasource.ds1.file.data-type=custom
#spring.cloud.sentinel.datasource.ds1.file.converter-class=org.springframework.cloud.alibaba.cl
#spring.cloud.sentinel.datasource.ds1.file.rule-type=flow

spring.cloud.sentinel.datasource.ds2.nacos.server-addr=localhost:8848
spring.cloud.sentinel.datasource.ds2.nacos.data-id=sentinel
spring.cloud.sentinel.datasource.ds2.nacos.group-id=DEFAULT_GROUP
spring.cloud.sentinel.datasource.ds2.nacos.data-type=json
spring.cloud.sentinel.datasource.ds2.nacos.rule-type=degrade

spring.cloud.sentinel.datasource.ds3.zk.path = /Sentinel-Demo/SYSTEM-CODE-DEMO-FLOW
spring.cloud.sentinel.datasource.ds3.zk.server-addr = localhost:2181
spring.cloud.sentinel.datasource.ds3.zk.rule-type=authority

spring.cloud.sentinel.datasource.ds4.apollo.namespace-name = application
spring.cloud.sentinel.datasource.ds4.apollo.flow-rules-key = sentinel
spring.cloud.sentinel.datasource.ds4.apollo.default-flow-rule-value = test
spring.cloud.sentinel.datasource.ds4.apollo.rule-type=param-flow
```

这种配置方式参考了 Spring Cloud Stream Binder 的配置，内部使用了 `TreeMap` 进行存储，comparator 为 `String.CASE_INSENSITIVE_ORDER` 。

> d1, ds2, ds3, ds4 是 `ReadableDataSource` 的名字，可随意编写。后面的 `file` ， `zk` ， `nacos` ， `apollo` 就是对应具体的数据源。 它们后面的配置就是这些具体数据源各自的配置。

每种数据源都有两个共同的配置项： `data-type` 、 `converter-class` 以及 `rule-type` 。

`data-type` 配置项表示 `Converter` 类型，Spring Cloud Alibaba Sentinel 默认提供两种内置的值，分别是 `json` 和 `xml` (不填默认是json)。 如果不想使用内置的 `json` 或 `xml` 这两种 `Converter` ，可以填写 `custom` 表示自定义 `Converter` ，然后再配置 `converter-class` 配置项，该配置项需要写类的全路径名(比如 `spring.cloud.sentinel.datasource.ds1.file.converter-class=org.springframework.cloud.alibaba.cloud.examples.JsonFlowRuleListConverter`)。

`rule-type` 配置表示该数据源中的规则属于哪种类型的规则(`flow` ， `degrade` ， `authority` ， `system` , `param-flow` , `gw-flow` , `gw-api-group`)。

当某个数据源规则信息加载失败的情况下，不会影响应用的启动，会在日志中打印出错误信息。

默认情况下，xml 格式是不支持的。需要添加 `jackson-dataformat-xml` 依赖后才会自动生效。

关于 Sentinel 动态数据源的实现原理，参考： 动态规则扩展

5.6. Zuul 支持

参考 Sentinel 网关限流

若想跟 Sentinel Starter 配合使用，需要加上 `spring-cloud-alibaba-sentinel-gateway` 依赖，同时需要添加 `spring-cloud-starter-netflix-zuul` 依赖来让 `spring-cloud-alibaba-sentinel-gateway` 模块里的 Zuul 自动化配置类生效：

XML

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

5.7. Spring Cloud Gateway 支持

参考 Sentinel 网关限流

若想跟 Sentinel Starter 配合使用，需要加上 `spring-cloud-alibaba-sentinel-gateway` 依赖，同时需要添加 `spring-cloud-starter-gateway` 依赖来让 `spring-cloud-alibaba-sentinel-gateway` 模块里的 Spring Cloud Gateway 自动化配置类生效：

XML

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

5.8. Sentinel 对外暴露的 Endpoint

Sentinel 内部提供了一个 Endpoint, 对应的 endpoint id 为 `sentinel` 。

Endpoint 暴露的 json 中包含了多种属性:

1. appName: 应用名
2. logDir: 日志所在目录
3. logUsePid: 日志文件名是否带上进程id
4. blockPage: 限流 block 之后跳转的页面
5. metricsFileSize: metrics 文件的大小
6. metricsFileCharset: metrics 文件对应的字符集
7. totalMetricsFileCount: metrics 最多保留的文件数
8. consoleServer: sentinel dashboard 地址
9. clientIp: 客户端 ip
10. heartbeatIntervalMs: 客户端跟 dashboard 的心跳间隔时间
11. clientPort: 客户端需要暴露的端口跟 dashboard 进行交互
12. coldFactor: 冷启动因子
13. filter: CommonFilter 相关的属性， 比如 order, urlPatterns 以及 enable
14. datasource: 客户端配置的数据源信息
15. rules: 客户端生效的规则，内部含有 flowRules, degradeRules, systemRules, authorityRule, paramFlowRule

这是 Endpoint 暴露的 json 示例:

JSON

```
{
    "blockPage": null,
```

```
"appName": "sentinel-example",
"consoleServer": "localhost:8080",
"coldFactor": "3",
"rules": {
    "flowRules": [{
        "resource": "GET:http://www.taobao.com",
        "limitApp": "default",
        "grade": 1,
        "count": 0.0,
        "strategy": 0,
        "refResource": null,
        "controlBehavior": 0,
        "warmUpPeriodSec": 10,
        "maxQueueingTimeMs": 500,
        "clusterMode": false,
        "clusterConfig": null
    }, {
        "resource": "/test",
        "limitApp": "default",
        "grade": 1,
        "count": 0.0,
        "strategy": 0,
        "refResource": null,
        "controlBehavior": 0,
        "warmUpPeriodSec": 10,
        "maxQueueingTimeMs": 500,
        "clusterMode": false,
        "clusterConfig": null
    }, {
        "resource": "/hello",
        "limitApp": "default",
        "grade": 1,
        "count": 1.0,
        "strategy": 0,
        "refResource": null,
        "controlBehavior": 0,
        "warmUpPeriodSec": 10,
        "maxQueueingTimeMs": 500,
        "clusterMode": false,
        "clusterConfig": null
    }]
},
"metricsFileCharset": "UTF-8",
"filter": {
    "order": -2147483648,
    "urlPatterns": ["/*"],
    "enabled": true
},
"totalMetricsFileCount": 6,
"datasource": {
    "ds1": {
        "file": {
            "dataType": "json",
            "ruleType": "FLOW",
            "converterClass": null,
            "file": "...",
            "charset": "utf-8",
            "recommendRefreshMs": 3000,
            "bufSize": 1048576
        },
        "nacos": null,
        "zk": null,
        "apollo": null,
```

```
            "redis": null
        }
    },
    "clientIp": "30.5.121.91",
    "clientPort": "8719",
    "logUsePid": false,
    "metricsFileSize": 52428800,
    "logDir": "...",
    "heartbeatIntervalMs": 10000
}
```


5.9. 关于 Sentinel Starter 更多的配置项信息

下表显示当应用的 `ApplicationContext` 中存在对应的Bean的类型时，会进行自动化设置：

存在Bean的类型	操作	作用
`UrlCleaner`	`WebCallbackManager.setUrlCleaner(urlCleaner)`	资源清理(资源（比如将满足 /foo/:id 的 URL 都归到 /foo/* 资源下）)
`UrlBlockHandler`	`WebCallbackManager.setUrlBlockHandler(urlBlockHandler)`	自定义限流处理逻辑
`RequestOriginParser`	`WebCallbackManager.setRequestOriginParser(requestOriginParser)`	设置来源信息

Spring Cloud Alibaba Sentinel 提供了这些配置选项

配置项	含义	默认值
`spring.application.`	Sentinel项目名	

name or project.name		
spring.cloud.sentinel.enabled	Sentinel自动化配置是否生效	true
spring.cloud.sentinel.	是否提前触发 Sentinel 初始化	false

eager		
spring.cloud.sentinel.transport.port	应用与Sentinel控制台交互的端口，应用本地会起一个该端口占用的HttpServer	8719
spring.cloud.sentinel.transport.dashbo	Sentinel 控制台地址	

ard		
spring.cloud.sentinel.transport.heartbeat-interval-ms	应用与Sentinel控制台的心跳间隔时间	
spring.cloud.sentinel.	此配置的客户端IP将被注册到 Sentinel Server 端	

transport.client-ip		
spring.cloud.sentinel.filter.order	Servlet Filter的加载顺序。Starter内部会构造这个filter	Integer.MIN_VALUE
spring.cloud.sentinel.filt	数据类型是数组。表示Servlet Filter的url pattern集合	/*

er.url-patterns		
spring.cloud.sentinel.filter.enabled	Enable to instance CommonFilter	true
spring.cloud.sentinel.metric	metric文件字符集	UTF-8

.charset		
spring.cloud.sentinel.metric.file-single-size	Sentinel metric 单个文件的大小	
spring.cloud.sentinel.metr	Sentinel metric 总文件数量	

ic.file-total-count		
spring.cloud.sentinel.log.dir	Sentinel 日志文件所在的目录	
spring.cloud.sentinel.log.swit	Sentinel 日志文件名是否需要带上pid	false

ch-pid		
spring.cloud.sentinel.servlet.block-page	自定义的跳转 URL，当请求被限流时会自动跳转至设定好的 URL	
spring.cloud.sentinel.flow.cold-	https://github.com/alibaba/Sentinel/wiki/%E9%99%90%E6%B5%81--%E5%86%B7%E5%90%AF%E5%8A%A8[冷启动因子]	3

factor		
spring.cloud.sentinel.zuul.order.pre	SentinelZuulPreFilter 的 order	10000
spring.cloud.sentinel.zuul.order.post	SentinelZuulPostFilter 的 order	1000

spring.cloud.sentinel.zuul.order.error	SentinelZuulErrorFilter 的 order	-1
spring.cloud.sentinel.scg.fallback.mode	Spring Cloud Gateway 熔断后的响应模式(选择 `redirect` or `response`)	

spring.cloud.sentinel.scg.fallback.redirect	Spring Cloud Gateway 响应模式为 'redirect' 模式对应的重定向 URL	
spring.cloud.sentinel.scg.fallback.r	Spring Cloud Gateway 响应模式为 'response' 模式对应的响应内容	

esponse-body		
spring.cloud.sentinel.scg.fallback.response-status	Spring Cloud Gateway 响应模式为 'response' 模式对应的响应码	429
spring.cloud.senti	Spring Cloud Gateway 响应模式为 'response' 模式对应的 content-type	application/json

nel.scg.fallback.content-type		

请注意。这些配置只有在 Servlet 环境下才会生效，RestTemplate 和 Feign 针对这些配置都无法生效

6. Spring Cloud Alibaba Dubbo

6.1. 简介

Dubbo Spring Cloud 基于 Dubbo Spring Boot 2.7.3[1] 和 Spring Cloud 2.x 开发，无论开发人员是 Dubbo 用户还是 Spring Cloud 用户， 都能轻松地驾驭，并以接近“零”成本的代价使应用向上迁移。Dubbo Spring Cloud 致力于简化 Cloud Native 开发成本，提高研发效能以及提升应用性能等目的。

6.2. 功能

由于 Dubbo Spring Cloud 构建在原生的 Spring Cloud 之上，其服务治理方面的能力可认为是 Spring Cloud Plus， 不仅完全覆盖 Spring Cloud 原生特性[5]，而且提供更为稳定和成熟的实现，特性比对如下表所示：

功能组件	Spring Cloud	Dubbo Spring Cloud

分布式配置（Distributed configuration）	Git、Zookeeper、Consul、JDBC	Spring Cloud 分布式配置 + Dubbo 配置中心[6]
服务注册与发现（Service registration and discovery）	Eureka、Zookeeper、Consul	Spring Cloud 原生注册中心[7] + Dubbo 原生注册中心[8]
负载均衡（Load balancing）	Ribbon（随机、轮询等算法）	Dubbo 内建实现（随机、轮询等算法 + 权重等特性）
服务熔断（Circuit Breakers）	Spring Cloud Hystrix	Spring Cloud Hystrix + Alibaba Sentinel[9] 等
服务调用（Service-to-service calls）	Open Feign、`RestTemplate`	Spring Cloud 服务调用 + Dubbo `@Reference`
链路跟踪（Tracing）	Spring Cloud Sleuth[10] + Zipkin[11]	Zipkin、opentracing 等

6.3. Reference 说明

[1]: 从 2.7.0 开始，Dubbo Spring Boot 与 Dubbo 在版本上保持一致

[2]: Preview releases of Spring Cloud Alibaba are available: 0.9.0, 0.2.2, and 0.1.2 - https://spring.io/blog/2011/04/11/preview-releases-of-spring-cloud-alibaba-are-available-0-9-0-0-2-2-and-0-1-2

[3]: 目前最新的 Spring Cloud “F” 版的版本为： `Finchley.SR2` - https://cloud.spring.io/spring-cloud-static/Finchley.SR2/single/spring-cloud.html

[4]: 当前Spring Cloud “G” 版为 `Greenwich.RELEASE`

[5]: Spring Cloud 特性列表 - https://cloud.spring.io/spring-cloud-static/Greenwich.RELEASE/single/spring-cloud.html#_features

[6]: Dubbo 2.7 开始支持配置中心，可自定义适配 - http://dubbo.apache.org/zh-cn/docs/user/configuration/config-center.html

[7]: Spring Cloud 原生注册中心，除 Eureka、Zookeeper、Consul 之外，还包括 Spring Cloud Alibaba 中的 Nacos

[8]: Dubbo 原生注册中心 - http://dubbo.apache.org/zh-cn/docs/user/references/registry/introduction.html

[9]: Alibaba Sentinel：Sentinel 以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性 - https://github.com/alibaba/Sentinel/wiki/%E4%BB%8B%E7%BB%8D ，目前 Sentinel 已被 Spring Cloud 项目纳为 Circuit Breaker 的候选实现 - https://spring.io/blog/2011/04/8/introducing-spring-cloud-circuit-breaker

[10]:Spring Cloud Sleuth - https://spring.io/projects/spring-cloud-sleuth

[11]: Zipkin - https://github.com/apache/incubator-zipkin

---

7. Spring Cloud Alibaba RocketMQ Binder

7.1. RocketMQ 介绍

RocketMQ 是一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。同时，广泛应用于多个领域，包括异步通信解耦、企业解决方案、金融支付、电信、电子商务、快递物流、广告营销、社交、即时通信、移动应用、手游、视频、物联网、车联网等。

具有以下特点：

- 能够保证严格的消息顺序
- 提供丰富的消息拉取模式
- 高效的订阅者水平扩展能力
- 实时的消息订阅机制
- 亿级消息堆积能力

7.2. RocketMQ 基本使用

- 下载 RocketMQ

下载 RocketMQ最新的二进制文件，并解压

解压后的目录结构如下：

```
apache-rocketmq
├── LICENSE
├── NOTICE
├── README.md
├── benchmark
├── bin
```

```
├── conf
└── lib
```

- 启动 NameServer

```bash
nohup sh bin/mqnamesrv &
tail -f ~/logs/rocketmqlogs/namesrv.log
```

- 启动 Broker

```bash
nohup sh bin/mqbroker -n localhost:9876 &
tail -f ~/logs/rocketmqlogs/broker.log
```

- 发送、接收消息

发送消息：

```bash
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
```

发送成功后显示： `SendResult [sendStatus=SEND_OK, msgId= …`

接收消息：

```bash
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

接收成功后显示： `ConsumeMessageThread_%d Receive New Messages: [MessageExt…`

- 关闭 Server

```bash
sh bin/mqshutdown broker
sh bin/mqshutdown namesrv
```

7.3. Spring Cloud Stream 介绍

Spring Cloud Stream 是一个用于构建基于消息的微服务应用框架。它基于 SpringBoot 来创建具有生产级别的单机 Spring 应用，并且使用 `Spring Integration` 与 Broker 进行连接。

Spring Cloud Stream 提供了消息中间件配置的统一抽象，推出了 publish-subscribe、consumer groups、partition 这些统一的概念。

Spring Cloud Stream 内部有两个概念：Binder 和 Binding。

- Binder: 跟外部消息中间件集成的组件，用来创建 Binding，各消息中间件都有自己的 Binder 实现。

比如 `Kafka` 的实现 `KafkaMessageChannelBinder`， `RabbitMQ` 的实现 `RabbitMessageChannelBinder` 以及 `RocketMQ` 的实现 `RocketMQMessageChannelBinder`。

- Binding: 包括 Input Binding 和 Output Binding。

Binding 在消息中间件与应用程序提供的 Provider 和 Consumer 之间提供了一个桥梁，实现了开发者只需使用应用程序的 Provider 或 Consumer 生产或消费数据即可，屏蔽了开发者与底层消息中间件的接触。

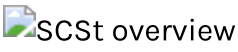

Figure 4. Spring Cloud Stream

使用 Spring Cloud Stream 完成一段简单的消息发送和消息接收代码：

JAVA

```
MessageChannel messageChannel = new DirectChannel();

// 消息订阅
((SubscribableChannel) messageChannel).subscribe(new MessageHandler() {
    @Override
    public void handleMessage(Message<?> message) throws MessagingException {
        System.out.println("receive msg: " + message.getPayload());
    }
});

// 消息发送
messageChannel.send(MessageBuilder.withPayload("simple msg").build());
```

这段代码所有的消息类都是 `spring-messaging` 模块里提供的。屏蔽具体消息中间件的底层实现，如果想用更换消息中间件，在配置文件里配置相关消息中间件信息以及修改 binder 依赖即可。

Spring Cloud Stream 底层基于这段代码去做了各种抽象。

7.4. 如何使用 Spring Cloud Alibaba RocketMQ Binder

如果要在您的项目中引入 RocketMQ Binder，需要引入如下 maven 依赖：

XML

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rocketmq</artifactId>
</dependency>
```

或者可以使用 Spring Cloud Stream RocketMQ Starter：

XML

```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rocketmq</artifactId>
</dependency>
```

7.5. Spring Cloud Alibaba RocketMQ Binder 实现

这是 Spring Cloud Stream RocketMQ Binder 的实现架构:

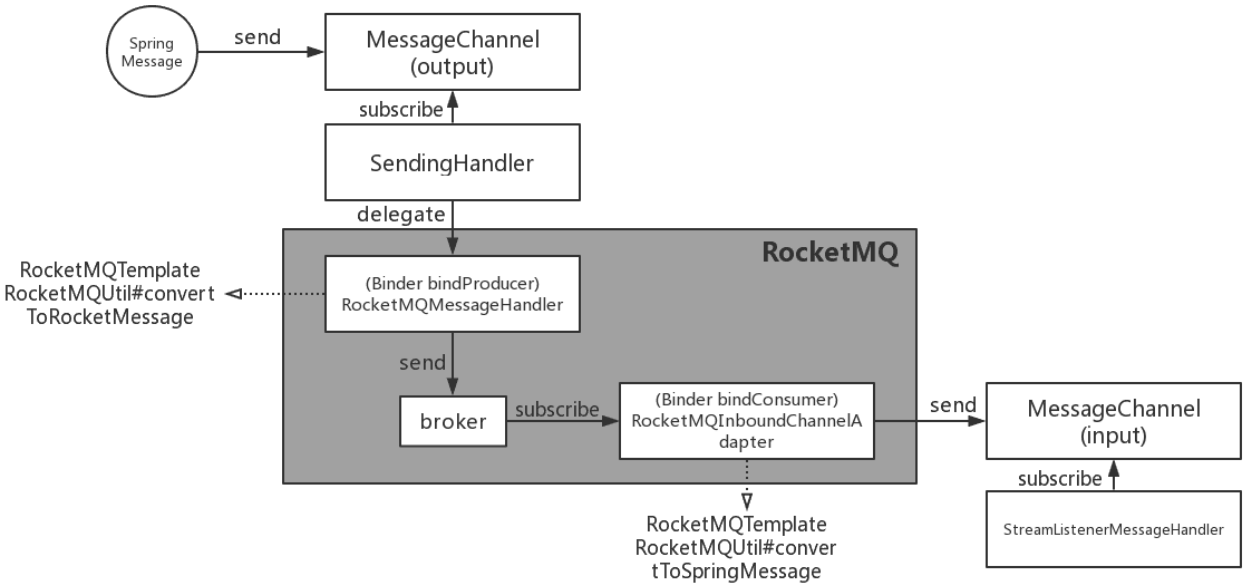

Figure 5. SCS RocketMQ Binder

RocketMQ Binder 的实现依赖于 RocketMQ-Spring 框架。

RocketMQ-Spring 框架是 RocketMQ 与 Spring Boot 的整合，RocketMQ Spring 主要提供了 3 个特性:

1. 使用 `RocketMQTemplate` 用来统一发送消息，包括同步、异步发送消息和事务消息
2. `@RocketMQTransactionListener` 注解用来处理事务消息的监听和回查
3. `@RocketMQMessageListener` 注解用来消费消息

RocketMQ Binder 的核心类 RocketMQMessageChannelBinder 实现了 Spring Cloud Stream 规范，内部构建会 `RocketMQInboundChannelAdapter` 和 `RocketMQMessageHandler` 。

`RocketMQMessageHandler` 会基于 Binding 配置构造 `RocketMQTemplate` ， `RocketMQTemplate` 内部会把 `spring-messaging` 模块内 `org.springframework.messaging.Message` 消息类转换成 RocketMQ 的消息

类 `org.apache.rocketmq.common.message.Message`，然后发送出去。

`RocketMQInboundChannelAdapter` 也会基于 Binding 配置构造 `RocketMQListenerBindingContainer`，`RocketMQListenerBindingContainer` 内部会启动 RocketMQ `Consumer` 接收消息。

> 在使用 RocketMQ Binder 的同时也可以配置 rocketmq.** 用于触发 RocketMQ Spring 相关的 AutoConfiguration

目前 Binder 支持在 `Header` 中设置相关的 key 来进行 RocketMQ Message 消息的特性设置。

比如 `TAGS`、`DELAY`、`TRANSACTIONAL_ARG`、`KEYS`、`WAIT_STORE_MSG_OK`、`FLAG` 表示 RocketMQ 消息对应的标签，

JAVA

```
MessageBuilder builder = MessageBuilder.withPayload(msg)
    .setHeader(RocketMQHeaders.TAGS, "binder")
    .setHeader(RocketMQHeaders.KEYS, "my-key")
    .setHeader(MessageConst.PROPERTY_DELAY_TIME_LEVEL, "1");
Message message = builder.build();
output().send(message);
```

7.6. MessageSource 支持

SCS RocketMQ Binder 支持 `MessageSource`，可以进行消息的拉取，例子如下：

JAVA

```
@SpringBootApplication
@EnableBinding(MQApplication.PolledProcessor.class)
public class MQApplication {

  private final Logger logger =
      LoggerFactory.getLogger(MQApplication.class);

  public static void main(String[] args) {
    SpringApplication.run(MQApplication.class, args);
  }

  @Bean
  public ApplicationRunner runner(PollableMessageSource source,
        MessageChannel dest) {
    return args -> {
      while (true) {
        boolean result = source.poll(m -> {
          String payload = (String) m.getPayload();
          logger.info("Received: " + payload);
          dest.send(MessageBuilder.withPayload(payload.toUpperCase())
              .copyHeaders(m.getHeaders())
              .build());
        }, new ParameterizedTypeReference<String>() { });
```

```
            if (result) {
              logger.info("Processed a message");
            }
            else {
              logger.info("Nothing to do");
            }
            Thread.sleep(5_000);
          }
        };
      }

      public static interface PolledProcessor {

        @Input
        PollableMessageSource source();

        @Output
        MessageChannel dest();

      }

    }
```

7.7. 配置选项

7.7.1. RocketMQ Binder Properties

spring.cloud.stream.rocketmq.binder.name-server

RocketMQ NameServer 地址(老版本使用 namesrv-addr 配置项)。

Default: `127.0.0.1:9876` .

spring.cloud.stream.rocketmq.binder.access-key

阿里云账号 AccessKey。

Default: null.

spring.cloud.stream.rocketmq.binder.secret-key

阿里云账号 SecretKey。

Default: null.

spring.cloud.stream.rocketmq.binder.enable-msg-trace

是否为 Producer 和 Consumer 开启消息轨迹功能

Default: `true` .

spring.cloud.stream.rocketmq.binder.customized-trace-topic

消息轨迹开启后存储的 topic 名称。

Default: `RMQ_SYS_TRACE_TOPIC` .

7.7.2. RocketMQ Consumer Properties

下面的这些配置是以 `spring.cloud.stream.rocketmq.bindings.<channelName>.consumer.` 为前缀的 RocketMQ Consumer 相关的配置。

enable

是否启用 Consumer。

默认值: `true` .

tags

Consumer 基于 TAGS 订阅，多个 tag 以 `||` 分割。

默认值: empty.

sql

Consumer 基于 SQL 订阅。

默认值: empty.

broadcasting

Consumer 是否是广播消费模式。如果想让所有的订阅者都能接收到消息，可以使用广播模式。

默认值: `false` .

orderly

Consumer 是否同步消费消息模式。

默认值: `false` .

delayLevelWhenNextConsume

异步消费消息模式下消费失败重试策略：

- -1,不重复，直接放入死信队列
- 0,broker 控制重试策略
- >0,client 控制重试策略

  默认值: `0` .

suspendCurrentQueueTimeMillis

同步消费消息模式下消费失败后再次消费的时间间隔。

默认值: `1000` .

7.7.3. RocketMQ Provider Properties

下面的这些配置是以 `spring.cloud.stream.rocketmq.bindings.<channelName>.producer.` 为前缀的 RocketMQ Producer 相关的配置。

enable

是否启用 Producer。

默认值: `true` .

group

Producer group name。

默认值: empty.

maxMessageSize

消息发送的最大字节数。

默认值: `8249344` .

transactional

是否发送事务消息。

默认值: `false` .

sync

是否使用同步得方式发送消息。

默认值: `false` .

vipChannelEnabled

是否在 Vip Channel 上发送消息。

默认值: `true` .

sendMessageTimeout

发送消息的超时时间(毫秒)。

默认值: `3000` .

compressMessageBodyThreshold

消息体压缩阀值(当消息体超过 4k 的时候会被压缩)。

默认值: `4096` .

retryTimesWhenSendFailed

在同步发送消息的模式下，消息发送失败的重试次数。

默认值: `2` .

retryTimesWhenSendAsyncFailed

在异步发送消息的模式下，消息发送失败的重试次数。

默认值: `2` .

retryNextServer

消息发送失败的情况下是否重试其它的 broker。

默认值: `false` .

7.8. 阿里云 MQ 服务

使用阿里云 MQ 服务需要配置 AccessKey、 SecretKey 以及云上的 NameServer 地址。

0.1.2 & 0.2.2 & 0.9.0 才支持该功能

```properties
spring.cloud.stream.rocketmq.binder.access-key=YourAccessKey
spring.cloud.stream.rocketmq.binder.secret-key=YourSecretKey
spring.cloud.stream.rocketmq.binder.name-server=NameServerInMQ
```

topic 和 group 请以 实例id% 为前缀进行配置。比如 topic 为 "test"，需要配置成 "实例id%test"

Figure 6. NameServer 的获取(配置中请去掉 http:// 前缀)

8. Spring Cloud AliCloud ANS

ANS（Application Naming Service） 是隶属于阿里云 EDAS 产品的组件， Spring Cloud AliCloud ANS 提供了 Spring Cloud 规范下商业版的服务注册与发现，可以让用户方便的在本地开发，同时也可以运行在云环境里。

> 目前 EDAS 已经支持直接部署 Nacos Discovery 应用

8.1. 如何引入 Spring Cloud AliCloud ANS

如果要在您的项目中引入 ANS，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alicloud-ans` 的 starter。

XML

```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alicloud-ans</artifactId>
</dependency>
```

8.2. 使用ANS进行服务注册

当客户端引入了 Spring Cloud AliCloud ANS Starter 以后，服务的元数据会被自动注册到注册中心，比如IP、端口、权重等信息。客户端会与服务端保持心跳，来证明自己可以正常提供服务。

以下是一个简单的应用示例。

JAVA

```java
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class ProviderApplication {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
```

```
}
```

既然服务会被注册到注册中心，那么肯定需要配置注册中心的地址，在 application.properties 中，还需要配置上以下地址。

```properties
# 应用名会被作为服务名称使用，因此会必选
spring.application.name=ans-provider
server.port=18081
# 以下就是注册中心的IP和端口配置
spring.cloud.alicloud.ans.server-list=127.0.0.1
spring.cloud.alicloud.ans.server-port=8080
```

此时没有启动注册中心，启动应用会报错，因此在应用启动之前，应当首先启动注册中心。

8.3. 启动注册中心

ANS 使用的注册中心有两种，一种是完全免费的轻量版配置中心，主要用于开发和本地调试，一种是云上注册中心，ANS 依托于阿里云 EDAS 产品提供服务注册的功能。通常情况下，可以使用轻量版配置中心作为开发和测试环境，使用云上的 EDAS 作为灰度和生产环境。

8.3.1. 启动轻量版配置中心

轻量版配置中心的下载和启动方式可参考 这里

只需要进行第1步（下载轻量配置中心）和第2步（启动轻量配置中心）即可，第3步（配置 hosts）在与 ANS 结合使用时，不需要操作。

启动完轻量版配置中心以后，直接启动 ProviderApplication ，即可将服务注册到轻量版配置中心，由于轻量版配置中心的默认端口是8080，因此你可以打开 http://127.0.0.1:8080 ，点击左侧"服务列表"，查看注册上来的服务。

8.3.2. 使用云上注册中心

使用云上注册中心，可以省去服务端的维护工作，同时稳定性也会更有保障。当使用云上注册中心时，代码部分和使用轻量配置中心并没有区别，但是配置上会有一些区别。

以下是一个简单的使用云上配置中心的配置示例。

```properties
# 应用名会被作为服务名称使用，因此是必选
spring.application.name=ans-provider
# 端口配置自由配置即可
server.port=18081
# 以下就是注册中心的IP和端口配置，因为默认就是127.0.0.1和8080，因此以下两行配置也可以省略
spring.cloud.alicloud.ans.server-mode=EDAS
spring.cloud.alicloud.access-key=你的阿里云AK
spring.cloud.alicloud.secret-key=你的阿里云SK
spring.cloud.alicloud.edas.namespace=cn-xxxxx
```

server-mode 的默认值为 LOCAL，如果要使用云上注册中心，则需要更改为 EDAS 。

access-key 和 secret-key 则是阿里云账号的 AK/SK，需要首先注册阿里云账号，然后登陆 阿里云AK/SK管理页面，即可看到 AccessKey ID 和 Access Key Secret，如果没有的话，需要点击"创建AccessKey"按钮创建。

namespace 是阿里云 EDAS 产品的概念，用于隔离不同的环境，比如测试环境和生产环境。要获取 namespace 需要 开通 EDAS 服务，按量计费模式下开通是免费的，开通以后进入 EDAS控制台，即可看到对应的 namespace，比如 cn-hangzhou。

> EDAS 提供应用托管服务，如果你将应用托管到 EDAS，那么 EDAS 将会自动为你填充所有配置。

9. Spring Cloud AliCloud ACM

Spring Cloud AliCloud ACM 是阿里云提供的商业版应用配置管理(Application Configuration Management) 产品 在 Spring Cloud 应用侧的客户端实现，且目前完全免费。

使用 Spring Cloud AliCloud ACM，可基于 Spring Cloud 的编程模型快速接入 ACM 配置管理功能。

> 目前 EDAS 已经支持直接部署 Nacos Config 应用

9.1. 如何引入 Spring Cloud AliCloud ACM

如果要在您的项目中引入 ACM，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alicloud-acm` 的 starter。

XML

```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alicloud-acm</artifactId>
</dependency>
```

9.2. 使用 ACM 进行配置管理

当客户端引入了 Spring Cloud AliCloud ACM Starter 以后，应用启动时会自动从配置管理的服务端获取配置信息，并注入到 Spring 的 Environment 中。

以下是一个简单的应用示例。

JAVA

```java
@SpringBootApplication
public class ProviderApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApplication.run(ProviderAppli
        String userName = applicationContext.getEnvironment().getProperty("user.name");
        String userAge = applicationContext.getEnvironment().getProperty("user.age");
        System.err.println("user name :"+userName+"; age: "+userAge);
    }
}
```

在从配置中心服务端获取配置信息之前，还需要配置服务端的地址，在 bootstrap.properties 中，还需要配置以下信息。

PROPERTIES

```properties
# 必选，应用名会被作为从服务端获取配置 key 的关键词组成部分
spring.application.name=acm-config
server.port=18081
# 以下就是配置中心服务端的IP和端口配置
spring.cloud.alicloud.acm.server-list=127.0.0.1
spring.cloud.alicloud.acm.server-port=8080
```

此时没有启动配置中心，启动应用会报错，因此在应用启动之前，应当首先启动配置中心。

9.2.1. 启动配置中心

ACM 使用的配置中心有两种，一种是本地运行的轻量版配置中心，主要用于开发和本地调试，一种是阿里云产品 ACM。通常情况下，可以使用轻量版配置中心作为开发和测试环境，使用云上的 ACM 作为灰度和生产环境。

使用轻量版配置中心

轻量版配置中心的下载和启动方式可参考 这里

> 只需要执行文档中的第1步 (下载轻量配置中心) 和第2步 (启动轻量配置中心)。

使用阿里云配置中心

使用云上 ACM ，可以省去服务端的维护工作，同时稳定性也会更有保障。当使用云上配置中心时，代码部分和使用轻量配置中心并没有区别，但是配置上会有一些区别。

以下是一个简单的使用云上配置中心的配置示例，配置详情需要在 ACM控制台查询

```properties
# 应用名会被作为从服务端获取配置 key 的关键词组成部分，因此是必选
spring.application.name=acm-config
# 端口配置自由配置即可
server.port=18081
# 以下就是配置中心的IP和端口配置
spring.cloud.alicloud.acm.server-mode=EDAS
spring.cloud.alicloud.access-key=你的阿里云AK
spring.cloud.alicloud.secret-key=你的阿里云SK
spring.cloud.alicloud.acm.endpoint=acm.aliyun.com
spring.cloud.alicloud.acm.namespace=你的 ACM namespace，需要在 ACM 控制台查询
```

> EDAS 提供应用托管服务，如果你将应用托管到 EDAS，那么 EDAS 将会自动为你填充所有与业务无关的配置。

9.2.2. 在配置中心添加配置

1. 启动好轻量版配置中心之后，在控制台中添加如下的配置。

```
Group:      DEFAULT_GROOUP

DataId:     acm-config.properties
```

```
Content:    user.name=james
            user.age=18
```

DataId 的格式为 `{prefix}.{file-extension}` ,prefix 默认从配置 spring.application.name 中取值，file-extension 默认的值为 "properties"。

9.2.3. 启动应用验证

启动这个Example，可以在控制台看到打印出的值正是我们在轻量版配置中心上预先配置的值。

```
user name :james; age: 18
```

9.3. 更改配置文件扩展名

spring-cloud-starter-alicloud-acm 中 DataId 默认的文件扩展名是 properties。除去 properties 格式之外，也支持 yaml 格式。 支持通过 spring.cloud.alicloud.acm.file-extension 来配置文件的扩展名，yaml 格式可以配置成 `yaml` 或 `yml` 。

修改文件扩展名后，在配置中心中的 DataID 以及 Content 的格式都必须做相应的修改。

9.4. 动态更新

spring-cloud-starter-alicloud-acm 默认支持配置的动态更新，当您在配置中心修改配置的内容时，会发布 Spring 中的 RefreshEvent 事件。 带有 @RefreshScope 和 @ConfigurationProperties 注解的类会自动刷新。

你可以通过配置 spring.cloud.alicloud.acm.refresh.enabled=false 来关闭动态刷新。

9.5. Profile 粒度的配置

spring-cloud-starter-alicloud-acm 在加载配置的时候，首先会加载 DataId 为{spring.application.name}.{file-extension}的配置，当 spring.profiles.active 中配置有内容时，还会依次去加载 spring.profile 对应的内容， DataId 的格式为{spring.application.name}-{profile}.{file-extension}的配置，且后者的优先级高于前者。

spring.profiles.active 属于配置的元数据，所以也必须配置在 bootstrap.properties 或 bootstrap.yaml 中。比如可以在 bootstrap.properties 中增加如下内容。

```
spring.profiles.active={profile-name}
```

Note: 也可以通过 JVM 参数 -Dspring.profiles.active=develop 或者 --spring.profiles.active=develop 这类优先级更高的方式来配置，只需遵循 Spring Boot 规范即可。

9.6. 自定义配置中心超时时间

ACM Client 与 Server 通信的超时时间默认是 3000ms，可以通过 `spring.cloud.alicloud.acm.timeout` 来修改超时时间，单位为 ms 。

9.7. 自定义 Group 的配置

在没有明确指定 `{spring.cloud.alicloud.acm.group}` 配置的情况下， 默认使用的是 DEFAULT_GROUP 。如果需要自定义自己的 Group，可以通过以下配置来实现：

PROPERTIES

```
spring.cloud.alicloud.acm.group=DEVELOP_GROUP
```

该配置必须放在 bootstrap.properties 文件中。并且在添加配置时 Group 的值要和 `spring.cloud.alicloud.acm.group` 的配置值一致。

9.8. 共享配置

ACM 提供了一种多个应用之间共享配置中心的同一个配置的推荐方式，供多个应用共享一些配置时使用，您在使用的时候需要添加在 bootstrap 中添加一个配置项 `spring.application.group` 。

PROPERTIES

```
spring.application.group=company.department.team
```

这时应用在获取上文提到的自身所独有的配置之前，会先依次从这些 DataId 去获取，分别是 company:application.properties, company.department:application.properties, company.department.team:application.properties。 然后，还会从 {spring.application.group}:{spring.application.name}.{file-extension} 中获取，越往后优先级越高，最高的仍然是应用自身所独有的配置。

> 共享配置中 DataId 默认后缀为 properties，可以通过 spring.cloud.alicloud.acm.file-extension 配置. `{spring.application.group}:{spring.application.name}.{file-extension}` 。

> 如果设置了 `spring.profiles.active` ，DataId 的格式还支持 `{spring.application.group}:{spring.application.name}-{spring.profiles.active}.{file-extension}` 。优先级高于 `{spring.application.group}:{spring.application.name}.{file-extension}`

9.9. Actuator 监控

ACM 对应的 Actuator 监控地址为 `/acm` ，其中 `config` 代表了 ACM 元数据配置的信息，`runtime.sources` 对应的是从 ACM 服务端获取的配置的信息及最后刷新时间，`runtime.refreshHistory` 对应的是动态刷新的历史记录。

---

10. Spring Cloud AliCloud OSS

OSS（Object Storage Service）是阿里云的一款对象存储服务产品， Spring Cloud AliCloud OSS 提供了Spring Cloud规范下商业版的对象存储服务，提供简单易用的API，并且支持与 Spring 框架中 Resource 的整合。

10.1. 如何引入 Spring Cloud AliCloud OSS

如果要在您的项目中引入 OSS，使用 group ID 为 `org.springframework.cloud` 和 artifact ID 为 `spring-cloud-starter-alicloud-oss` 的 starter。

XML

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alicloud-oss</artifactId>
</dependency>
```

10.2. 如何使用 OSS API

10.2.1. 配置 OSS

使用 Spring Cloud AliCloud OSS 之前，需要在 application.properties 中加入以下配置。

PROPERTIES

```
spring.cloud.alicloud.access-key=你的阿里云AK
spring.cloud.alicloud.secret-key=你的阿里云SK
spring.cloud.alicloud.oss.endpoint=***.aliyuncs.com
```

access-key 和 secret-key 是阿里云账号的AK/SK，需要首先注册阿里云账号，然后登陆 阿里云AK/SK管理页面 ，即可看到 AccessKey ID 和 Access Key Secret ，如果没有的话，需要点击"创建AccessKey"按钮创建。

endpoint可以到 OSS 的 官方文档中查看，根据所在的 region ，填写对应的 endpoint 即可。

10.2.2. 引入 OSS API

Spring Cloud Alicloud OSS 中的 OSS API 基于阿里云官方OSS SDK提供，具备上传、下载、查看等所有对象存储类操作API。

一个简单的使用 OSS API 的应用如下。

JAVA

```
@SpringBootApplication
public class OssApplication {

    @Autowired
    private OSS ossClient;

    @RequestMapping("/")
    public String home() {
        ossClient.putObject("bucketName", "fileName", new FileInputStream("/your/local/file/pat
        return "upload success";
    }

    public static void main(String[] args) throws URISyntaxException {
        SpringApplication.run(OssApplication.class, args);
    }

}
```

在上传文件之前，首先需要 注册阿里云账号 ，如果已经有的话，请 开通OSS服务。

进入 OSS控制台，点击左侧"新建Bucket"，按照提示创建一个Bucket，然后将bucket名称替换掉上面代码中的"bucketName"，而"fileName"取任意文件名，"/your/local/file/path"取任意本地文件路

径，然后 curl http://127.0.0.1:端口/ 即可上传文件，可以到 OSS控制台查看效果。

更多关于 OSS API 的操作，可以参考 OSS官方SDK文档。

10.3. 与 Spring 框架的 Resource 结合

Spring Cloud AliCloud OSS 整合了 Spring 框架的 Resource 规范，可以让用户很方便的引用 OSS 的资源。

一个简单的使用 Resource 的例子如下。

JAVA

```java
@SpringBootApplication
public class OssApplication {

    @Value("oss://bucketName/fileName")
    private Resource file;

    @GetMapping("/file")
    public String fileResource() {
        try {
            return "get file resource success. content: " + StreamUtils.copyToString(
                file.getInputStream(), Charset.forName(CharEncoding.UTF_8));
        } catch (Exception e) {
            return "get resource fail: " + e.getMessage();
        }
    }

    public static void main(String[] args) throws URISyntaxException {
        SpringApplication.run(OssApplication.class, args);
    }

}
```

以上示例运行的前提是，在 OSS 上需要有名为"bucketName"的Bucket，同时在该Bucket下，存在名为"fileName"的文件。

10.4. 采用 STS 授权

Spring Cloud AliCloud OSS 除了 AccessKey/SecretKey 的授权方式以外，还支持 STS 授权方式。STS 是临时访问令牌的方式，一般用于授权第三方，临时访问自己的资源。

作为第三方，也就是被授权者，只需要配置以下内容，就可以访问临时被授权的资源。

PROPERTIES

```properties
spring.cloud.alicloud.oss.authorization-mode=STS
spring.cloud.alicloud.oss.endpoint=***.aliyuncs.com
```

```
spring.cloud.alicloud.oss.sts.access-key=你被授权的AK
spring.cloud.alicloud.oss.sts.secret-key=你被授权的SK
spring.cloud.alicloud.oss.sts.security-token=你被授权的ST
```

其中 spring.cloud.alicloud.oss.authorization-mode 是枚举类型，此时填写 STS ，代表采用 STS 的方式授权。 endpoint可以到 OSS 的 官方文档中查看，根据所在的 region ， 填写对应的 endpoint 即可。

access-key、secret-key和security-token需要由授权方颁发，如果对 STS 不了解的话，可以参考 STS官方文档。

10.5. 更多客户端配置

除了基本的配置项以外， Spring Cloud AliCloud OSS 还支持很多额外的配置，也是在 application.properties 文件中。

以下是一些简单的示例。

PROPERTIES

```
spring.cloud.alicloud.oss.authorization-mode=STS
spring.cloud.alicloud.oss.endpoint=***.aliyuncs.com
spring.cloud.alicloud.oss.sts.access-key=你被授权的AK
spring.cloud.alicloud.oss.sts.secret-key=你被授权的SK
spring.cloud.alicloud.oss.sts.security-token=你被授权的ST

spring.cloud.alicloud.oss.config.connection-timeout=3000
spring.cloud.alicloud.oss.config.max-connections=1000
```

如果想了解更多的配置项，可以参考 OSSClient配置项 的末尾表格。

> 通常情况下，都需要将 OSSClient配置项 末尾表格中的参数名更换成"-"连接，且所有字母小写。例如 ConnectionTimeout，对应 connection-timeout。

11. Spring Cloud AliCloud SchedulerX

SchedulerX（分布式任务调度） 是隶属于阿里云EDAS产品的组件， Spring Cloud AliCloud SchedulerX 提供了在Spring Cloud的配置规范下，分布式任务调度的功能支持。SchedulerX可提供秒级、精准、高可靠、高可用的定时任务调度服务，并支持多种类型的任务调度，如简单单机任务、简单多机任务、脚本任务以及网格任务。

11.1. 如何引入 Spring Cloud AliCloud SchedulerX

如果要在您的项目中引入 SchedulerX，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alicloud-schedulerX` 的 starter。

XML

```xml
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alicloud-schedulerX</artifactId>
</dependency>
```

11.2. 启动SchedulerX任务调度

当客户端引入了 Spring Cloud AliCloud SchedulerX Starter 以后，只需要进行一些简单的配置，就可以自动初始化SchedulerX的任务调度服务。

以下是一个简单的应用示例。

JAVA

```java
@SpringBootApplication
public class ScxApplication {

    public static void main(String[] args) {
        SpringApplication.run(ScxApplication.class, args);
    }

}
```

在application.properties中，需要加上以下配置。

PROPERTIES

```properties
server.port=18033
# 其中cn-test是SchedulerX的测试区域
spring.cloud.alicloud.scx.group-id=***
spring.cloud.alicloud.edas.namespace=cn-test
```

在获取group-id之前，需要首先 注册阿里云账号 ，然后 开通EDAS服务 ，并 开通分布式任务管理组件 。

其中group-id的获取，请参考 这里。

在创建group的时候，要选择"测试"区域。

11.3. 编写一个简单任务

简单任务是最常用的任务类型，只需要实现 ScxSimpleJobProcessor 接口即可。

以下是一个简单的单机类型任务示例。

JAVA

```java
public class SimpleTask implements ScxSimpleJobProcessor {

    @Override
    public ProcessResult process(ScxSimpleJobContext context) {
        System.out.println("-----------Hello world---------------");
        ProcessResult processResult = new ProcessResult(true);
        return processResult;
    }

}
```

11.4. 对任务进行调度

进入 SchedulerX任务列表 页面，选择上方"测试"区域，点击右上角"新建Job"，创建一个Job，即如下所示。

TEXT

```
Job分组：测试——***-*-*-****
Job处理接口：org.springframework.cloud.alibaba.cloud.examples.SimpleTask
类型：简单Job单机版
定时表达式：默认选项——0 * * * * ?
Job描述：无
自定义参数：无
```

以上任务类型选择了"简单Job单机版"，并且制定了Cron表达式为"0 * * * * ?"，这意味着，每过一分钟，任务将会被执行且只执行一次。

更多任务类型，请参考 SchedulerX官方文档。

11.5. 生产环境使用

以上使用的都是SchedulerX的"测试"区域，主要用于本地调试和测试。

在生产级别，除了上面的group-id和namespace以外，还需要一些额外的配置，如下所示。

PROPERTIES

```properties
server.port=18033
# 其中cn-test是SchedulerX的测试区域
spring.cloud.alicloud.scx.group-id=***
spring.cloud.alicloud.edas.namespace=***
# 当应用运行在EDAS上时，以下配置不需要手动配置。
spring.cloud.alicloud.access-key=***
spring.cloud.alicloud.secret-key=***
```

```
# 以下配置不是必须的，请参考SchedulerX文档
spring.cloud.alicloud.scx.domain-name=***
```

其中group-id与之前的获取方式一样，namespace则是从EDAS控制台左侧"命名空间"列表中获取命名空间ID。

group-id必须创建在namespace当中。

access-key以及secret-key为阿里云账号的AK/SK信息，如果应用在EDAS上部署，则不需要填写这两项信息，否则请前往 安全信息管理获取。

domain-name并不是必须的，具体请参考 SchedulerX官方文档。

---

12. Spring Cloud AliCloud SMS

短信服务（Short Message Service）是阿里云为用户提供的一种通信服务的能力。 Spring Cloud AliCloud SMS 实现了与 SMS 的简单集成，提供更为简单易用的 API，可以基于 Spring Cloud Alibaba SMS 来快速的接入阿里云的 SMS 服务。

12.1. 如何引入 Spring Cloud AliCloud SMS

如果要在您的项目中引入 SMS，使用 group ID 为 `com.alibaba.cloud` 和 artifact ID 为 `spring-cloud-starter-alicloud-sms` 的 starter。

XML

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alicloud-sms</artifactId>
</dependency>
```

12.2. 如何使用 SMS API

12.2.1. 配置 SMS

使用 Spring Cloud AliCloud SMS 之前，需要在 application.properties 中加入以下配置。

PROPERTIES

```
spring.cloud.alicloud.access-key=你的阿里云 AK
spring.cloud.alicloud.secret-key=你的阿里云 SK
```

access-key 和 secret-key 是阿里云账号的 AK/SK，需要首先注册阿里云账号，然后登陆 阿里云AK/SK管理页面，即可看到 AccessKey ID 和 Access Key Secret，如果没有的话，需要点击"创建AccessKey"按钮创建。

12.2.2. 引入 SMS API

Spring Cloud Alicloud SMS 中的 SMS API 基于阿里云官方 SMS SDK ,提供具备单个短信发送、多个短信批量发送、短信查询、短信消息(短信回执消息 和 上行短信消息) 类型操作API。

一个简单的使用 SMS API 发送短信的应用如下。

JAVA

```java
@SpringBootApplication
public class SmsApplication {

    @Autowired
    private ISmsService smsService;

    /**
     * 短信发送 Example
     * @param code
     * @return
     */
    @RequestMapping("/batch-sms-send.do")

    public SendBatchSmsResponse batchsendCheckCode(
            @RequestParam(name = "code") String code) {

        // 组装请求对象-具体描述见控制台-文档部分内容
        SendSmsRequest request = new SendSmsRequest();
        // 必填:待发送手机号
        request.setPhoneNumbers("152******");
        // 必填:短信签名-可在短信控制台中找到
        request.setSignName("******");
        // 必填:短信模板-可在短信控制台中找到
        request.setTemplateCode("******");
        // 可选:模板中的变量替换JSON串,如模板内容为"【企业级分布式应用服务】,您的验证码为${code}
        request.setTemplateParam("{\"code\":\"" + code + "\"}");
        SendSmsResponse sendSmsResponse ;
        try {
            sendSmsResponse = smsService.sendSmsRequest(request);
        }
        catch (ClientException e) {
            e.printStackTrace();
            sendSmsResponse = new SendSmsResponse();
        }
        return sendSmsResponse ;
    }

    public static void main(String[] args) throws URISyntaxException {

        SpringApplication.run(SmsApplication.class, args);
    }

}
```

在发送短信之前，首先需要 注册阿里云账号 ，如果已经有的话，请 开通 SMS 服务。

更多关于 SMS 发送短信的步骤，可以参考 SMS 官方 短信发送API(SendSms)---JAVA 文档。

> 由于早期的 SMS sdk 版本的问题，如果短信发送失败，请将代码中含有明确指定 MethodType 为 POST 的这行代码给删除掉。如果还有问题，请第一时间联系我们。

12.3. SMS Api 的高级功能

Spring Cloud Alicloud SMS 封装的 API 接口为了降低学习的成本，尽量保持和官网提供的 API 以及 Example 保持一致。

- 批量短信发送

参考以下的 Example ，来快速开发一个具有批量短信发送的功能。在 Controller 中或者新建一个 Controler 新增如下代码：

JAVA

```
/**
 * 批量短信发送 Example
 * @param code
 * @return
 */
@RequestMapping("/batch-sms-send.do")
public SendBatchSmsResponse batchsendCheckCode(
        @RequestParam(name = "code") String code) {
    // 组装请求对象
    SendBatchSmsRequest request = new SendBatchSmsRequest();
    // 使用 GET 提交
    request.setMethod(MethodType.GET);
    // 必填:待发送手机号。支持JSON格式的批量调用，批量上限为100个手机号码,批量调用相对于单条调用
    request.setPhoneNumberJson("[\"177********\",\"130********\"]");
    // 必填:短信签名-支持不同的号码发送不同的短信签名
    request.setSignNameJson("[\"*******\",\"*******\"]");
    // 必填:短信模板-可在短信控制台中找到
    request.setTemplateCode("******");
    // 必填:模板中的变量替换JSON串,如模板内容为"亲爱的${name},您的验证码为${code}"时,此处的值为
    // 友情提示:如果JSON中需要带换行符,请参照标准的JSON协议对换行符的要求,比如短信内容中包含\r\n
    request.setTemplateParamJson(
            "[{\"code\":\"" + code + "\"},{\"code\":\"" + code + "\"}]");
    SendBatchSmsResponse sendSmsResponse ;
    try {
        sendSmsResponse = smsService
                .sendSmsBatchRequest(request);
        return sendSmsResponse;
    }
    catch (ClientException e) {
        e.printStackTrace();
```

```
            sendSmsResponse =  new SendBatchSmsResponse();
        }
        return sendSmsResponse ;
    }
```

这里设置请求的 MethodType 为 GET，和官网给出的例子还有些不一样。这是因为依赖的阿里云 POP API 版本不一致导致不兼容的问题，设置为 GET 即可。

更多的参数说明可 参考这里

- 短信查询

参考以下的 Example ，可以快速开发根据某个指定的号码查询短信历史发送状态。在 Controller 中或者新建一个 Controler 新增如下代码：

JAVA

```
/**
 *
 * 短信查询 Example
 * @param telephone
 * @return
 */
@RequestMapping("/query.do")
public QuerySendDetailsResponse querySendDetailsResponse(
        @RequestParam(name = "tel") String telephone) {
    // 组装请求对象
    QuerySendDetailsRequest request = new QuerySendDetailsRequest();
    // 必填-号码
    request.setPhoneNumber(telephone);
    // 必填-短信发送的日期 支持30天内记录查询（可查其中一天的发送数据），格式yyyyMMdd
    request.setSendDate("20190103");
    // 必填-页大小
    request.setPageSize(10L);
    // 必填-当前页码从1开始计数
    request.setCurrentPage(1L);
    try {
        QuerySendDetailsResponse response = smsService.querySendDetails(request);
        return response;
    }
    catch (ClientException e) {
        e.printStackTrace();
    }

    return new QuerySendDetailsResponse();
}
```

更多的参数说明，可 参考这里

- 短信回执消息

通过订阅 SmsReport 短信状态报告，可以获知每条短信的发送情况，了解短信是否达到终端用户的状态与相关信息。这些工作已经都被 Spring Cloud AliCloud SMS 封装在内部了。你只需要完成以下两步即可。

1、在 `application.properties` 配置文件中(也可以是 application.yaml)配置 SmsReport 的队列名称。

application.properties

```properties
spring.cloud.alicloud.sms.report-queue-name=Alicom-Queue-********-SmsReport
```

2、 实现 SmsReportMessageListener 接口，并初始化一个 Spring Bean 。

```java
/**
 * 如果需要监听短信是否被对方成功接收，只需实现这个接口并初始化一个 Spring Bean 即可。
 */
@Component
public class SmsReportMessageListener
        implements org.springframework.cloud.alicloud.sms.SmsReportMessageListener {

    @Override
    public boolean dealMessage(Message message) {
        // 在这里添加你的处理逻辑

        //do something

        System.err.println(this.getClass().getName() + "; " + message.toString());
        return true;
    }
}
```

更多关于 Message 的消息体格式可 参考这里。

- 上行短信消息

通过订阅SmsUp上行短信消息，可以获知终端用户回复短信的内容。这些工作也已经被 Spring Cloud AliCloud SMS 封装好了。你只需要完成以下两步即可。

1、 在 `application.properties` 配置文件中(也可以是 application.yaml)配置 SmsReport 的队列名称。

application.properties

```properties
spring.cloud.alicloud.sms.up-queue-name=Alicom-Queue-********-SmsUp
```

2、实现 SmsUpMessageListener 接口，并初始化一个 Spring Bean 。

JAVA

```java
/**
 * 如果发送的短信需要接收对方回复的状态消息，只需实现该接口并初始化一个 Spring Bean 即可。
 */
@Component
public class SmsUpMessageListener
        implements org.springframework.cloud.alicloud.sms.SmsUpMessageListener {

    @Override
    public boolean dealMessage(Message message) {
        // 在这里添加你的处理逻辑

        //do something

        System.err.println(this.getClass().getName() + "; " + message.toString());
        return true;
    }
}
```

更多关于 Message 的消息体格式可 参考这里。

---

13. Spring Cloud Alibaba Sidecar

`Spring Cloud Alibaba Sidecar` 是一个用来快速**完美整合** Spring Cloud 与 **异构微服务** 的框架，灵感来自 Spring Cloud Netflix Sidecar 。目前支持的服务发现组件：

- Nacos
- Consul

13.1. 术语

13.1.1. 异构微服务

非Spring Cloud应用，统称异构微服务。比如你的遗留项目，或者非JVM应用。

13.1.2. ``完美整合''的三层含义

- 享受服务发现的优势
- 有负载均衡
- 有断路器

13.2. Why or Why not?

13.2.1. 为什么要编写Alibaba Sidecar?

原因有两点：

- Spring Cloud子项目 `Spring Cloud Netflix Sidecar` 是可以快速整合异构微服务的。然而，Sidecar只支持使用Eureka作为服务发现，**如果使用其他服务发现组件就抓瞎了。**
- **Sidecar是基于Zuul 1.x的**，Spring Cloud官方明确声明，未来将会逐步淘汰Zuul。今年早些时候，我有给Spring Cloud官方提出需求，希望官方实现一个基于Spring Cloud Gateway的新一代Sidecar，然而官方表示并没有该计划。详见：https://github.com/spring-cloud/spring-cloud-gateway/issues/735

既然没有，索性自己写了。

13.2.2. 为什么不用Service Mesh?

- 目前Mesh主要使用场景在Kubernetes领域（Istio、Linkerd 2等，大多将Kubernetes作为First Class支持，虽然Istio也可部署在非Kubernetes环境），而目前业界，Spring Cloud应用未必有试试Mesh的环境；
- 使用Alibaba Sidecar一个小组件就能解决问题了（核心代码不超过200行），引入整套Mesh方案，颇有点屠龙刀杀黄鳝的意思。

13.3. 原理

- Alibaba Sidecar根据配置的异构微服务的IP、端口等信息，**将异构微服务的IP/端口注册到服务发现组件上** 。
- Alibaba Sidecar实现了 **健康检查** ， Alibaba Sidecar会定时检测异构微服务是否健康。如果发现异构微服务不健康，Alibaba Sidecar会自动将代表异构微服务的Alibaba Sidecar实例下线；如果异构微服务恢复正常，则会自动上线。最长延迟是30秒，详见 `Alibaba SidecarChecker#check` 。

13.4. 要求

- 【必须】你的异构微服务需使用HTTP通信。这一点严格来说不算要求，因为Spring Cloud本身就是基于HTTP的；
- 【可选】如果微服务配置了 `sidecar.health-check-url` ，则表示开启健康检查，此时，你的异构微服务需实现健康检查（可以是空实现，只要暴露一个端点，返回类似 `{"status": "UP"}` 的字符串即可）。

13.5. 使用示例

- 如使用Nacos作为服务发现组件，详见`spring-cloud-alibaba-examples/spring-cloud-alibaba-sidecar-examples/spring-cloud-alibaba-sidecar-nacos-example`
- 如使用Consul作为服务发现组件，详见`spring-cloud-alibaba-examples/spring-cloud-alibaba-sidecar-examples/spring-cloud-alibaba-sidecar-nacos-example`

13.5.1. 示例代码（以Nacos服务发现为例）

- 加依赖：

```xml
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-sidecar</artifactId>
</dependency>
```

- 写配置：

```yaml
server:
  port: 8070
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    gateway:
      discovery:
        locator:
          enabled: true
  application:
    name: node-service
sidecar:
  # 异构微服务的IP
  ip: 127.0.0.1
  # 异构微服务的端口
  port: 8060
  # 异构微服务的健康检查URL
  health-check-url: http://localhost:8060/health.json
management:
endpoint:
    health:
      show-details: always
```

配置比较简单，就是把Alibaba Sidecar注册到Nacos上，然后添加了几行Alibaba Sidecar的配置。

13.5.2. 异构微服务

我准备了一个NodeJS编写的简单微服务。

```javascript
var http = require('http');
var url = require("url");
```

```
var path = require('path');

// 创建server
var server = http.createServer(function(req, res) {
    // 获得请求的路径
    var pathname = url.parse(req.url).pathname;
    res.writeHead(200, { 'Content-Type' : 'application/json; charset=utf-8' });
    // 访问http://localhost:8060/，将会返回{"index":"欢迎来到首页"}
    if (pathname === '/') {
        res.end(JSON.stringify({ "index" : "欢迎来到首页" }));
    }
    // 访问http://localhost:8060/health，将会返回{"status":"UP"}
    else if (pathname === '/health.json') {
        res.end(JSON.stringify({ "status" : "UP" }));
    }
    // 其他情况返回404
    else {
        res.end("404");
    }
});
// 创建监听，并打印日志
server.listen(8060, function() {
    console.log('listening on localhost:8060');
});
```

13.5.3. 测试

测试1：Spring Cloud微服务完美调用异构微服务

为你的Spring Cloud微服务整合Ribbon，然后构建 `http://node-service/` ，**就可以请求到异构微服务的** / 了。

示例：

Ribbon请求 `http://node-service/` 会请求到 `http://localhost:8060/` ，以此类推。

至于断路器，正常为你的Spring Cloud微服务整合Sentinel或者Hystirx、Resilience4J即可 。

测试2：异构微服务完美调用Spring Cloud微服务

由于Alibaba Sidecar基于Spring Cloud Gateway，而网关自带转发能力。

示例：

如果你有一个Spring Cloud微服务叫做 `spring-cloud-microservice` ，那么NodeJS应用只需构建 `http://localhost:8070/spring-cloud-microservice/` ，**Alibaba Sidecar就会把请求转发到** `spring-cloud-microservice` **的** / 。

而Spring Cloud Gateway是整合了Ribbon的，所以实现了负载均衡；Spring Cloud Gateway还可以整合Sentinel或者Hystirx、Resilience4J，所以也带有了断路器。

13.6. Alibaba Sidecar优缺点分析

Alibaba Sidecar的设计和Sidecar基本一致，优缺点和Sidecar的优缺点也是一样的。

优点：

- 接入简单，几行代码就可以将异构微服务整合到Spring Cloud生态
- 不侵入原代码

缺点：

- 每接入一个异构微服务实例，都需要额外部署一个Alibaba Sidecar实例，增加了部署成本（虽然这个成本在Kubernetes环境中几乎可以忽略不计（只需将Alibaba Sidecar实例和异构微服务作为一个Pod部署即可））；
- 异构微服务调用Spring Cloud微服务时，本质是把Alibaba Sidecar当网关在使用，经过了一层转发，性能有一定下降。