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

目录

开源组件

Nacos Config

Spring Cloud Alibaba Nacos Config

Nacos 提供用于存储配置和其他元数据的 key/value 存储，为分布式系统中的外部化配置提供服务器端和客户端支持。使用 Spring Cloud Alibaba Nacos Config，您可以在 Nacos Server 集中管理你 Spring Cloud 应用的外部属性配置。

Spring Cloud Alibaba Nacos Config 是 Config Server 和 Client 的替代方案，客户端和服务器上的概念与 Spring Environment 和 PropertySource 有着一致的抽象，在特殊的 bootstrap 阶段，配置被加载到 Spring 环境中。当应用程序通过部署管道从开发到测试再到生产时，您可以管理这些环境之间的配置，并确保应用程序具有迁移时需要运行的所有内容。

快速开始

Nacos 服务端初始化

1、启动 Nacos Server。启动方式可见 Nacos 官网

2、启动好 Nacos 之后，在 Nacos 添加如下的配置：

```
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
```

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

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。

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

现在就可以创建一个标准的 Spring Boot 应用。

```
@SpringBootApplication
public class ProviderApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(ProviderApplication.class, args);
        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 Example 之前， 必须使用 bootstrap.properties 配置文件来配置 Nacos Server 地址，例如：

bootstrap.properties

```
spring.application.name=nacos-config
spring.cloud.nacos.config.server-addr=127.0.0.1:8848
```

Note	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

	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.ProviderApplication    : Started ProviderApplication in
14.645 seconds (JVM running for 15.139)
user name :nacos-config-properties; age: 90
2018-11-02 14:24:51.688  INFO 32700 --- [-127.0.0.1:8848]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@a
8c5e74: startup date [Fri Nov 02 14:24:51 CST 2018]; root of context
hierarchy
2018-11
```

基于 dataid 为 yaml 的文件扩展名配置方式

spring-cloud-starter-alibaba-nacos-config 对于 yaml 格式也是完美支持的。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

```
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.ProviderApplication:Started ProviderApplication in 14.183
seconds (JVM running for 14.671)
user name :nacos-config-yaml; age: 68
```

```
2018-11-02 14:59:00.529  INFO 32928 --- [-127.0.0.1:8848]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@2
65a478e: startup date [Fri Nov 02 14:59:00 CST 2018]; root of context
hierarchy
```

支持配置的动态更新

spring-cloud-starter-alibaba-nacos-config 也支持配置的动态更新，启动 Spring Boot 应用测试的代码如下：

```
@SpringBootApplication
public class ProviderApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(ProviderApplication.class, args);
        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               : Started application in 0.144
seconds (JVM running for 71.752)
2018-11-02 15:04:25.070  INFO 32957 --- [-127.0.0.1:8848]
s.c.a.AnnotationConfigApplicationContext : Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@1
```

```
0c89124: startup date [Fri Nov 02 15:04:25 CST 2018]; parent:
org.springframework.context.annotation.AnnotationConfigApplicationContext@6
520af7
2018-11-02 15:04:25.071  INFO 32957 --- [-127.0.0.1:8848]
s.c.a.AnnotationConfigApplicationContext : Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@6
520af7: startup date [Fri Nov 02 15:04:24 CST 2018]; root of context
hierarchy
//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
```

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

可支持 profile 粒度的配置

spring-cloud-starter-alibaba-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。

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

Note	${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：

```
@SpringBootApplication
public class ProviderApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(ProviderApplication.class, args);
        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 + "; age: " + userAge);
            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 : Closing
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServ
erApplicationContext@6f1c29b7: startup date [Fri Nov 02 15:33:57 CST 2018];
parent:
org.springframework.context.annotation.AnnotationConfigApplicationContext@6
3355449
```

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：

```
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 : Closing
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServ
erApplicationContext@6aa8e115: startup date [Fri Nov 02 15:42:03 CST 2018];
parent:
org.springframework.context.annotation.AnnotationConfigApplicationContext@1
9bb07ed
```

Note	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。

支持自定义 namespace 的配置

首先看一下 Nacos 的 Namespace 的概念， Nacos 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：

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

Note	该配置必须放在 bootstrap.properties 文件中。此外 spring.cloud.nacos.config.namespace 的 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 namespae，否则将会导致读取不到正确的配置。

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：

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

Note	该配置必须放在 bootstrap.properties 文件中。并且在添加配置时 Group 的值一和 `spring.cloud.nacos.config.group` 的配置值一致。

支持自定义扩展的 Data Id 配置

Spring Cloud Alibaba 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：

```
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.extension-configs[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.extension-configs[1].data-id=ext-config-
common02.properties
spring.cloud.nacos.config.extension-configs[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.extension-configs[2].data-id=ext-config-
common03.properties
spring.cloud.nacos.config.extension-configs[2].group=REFRESH_GROUP
spring.cloud.nacos.config.extension-configs[2].refresh=true
```

可以看到:

- 通过 `spring.cloud.nacos.config.extension-configs[n].data-id` 的配置方式来支持多个 Data Id 的配置。
- 通过 `spring.cloud.nacos.config.extension-configs[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.extension-configs[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。

Note	多个 Data Id 同时配置时，他的优先级关系是 `spring.cloud.nacos.config.exter configs[n].data-id` 其中 n 的值越大，优先级越高。
Note	`spring.cloud.nacos.config.extension-configs[n].data-id` 的值必须带文件扩展名，文件扩展支持 properties，又可以支持 yaml/yml。此时 `spring.cloud.nacos.config.file-extension` 的配定义扩展配置的 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：

```
# 配置支持共享的 Data Id
spring.cloud.nacos.config.shared-configs[0].data-id=common.yaml

# 配置 Data Id 所在分组，缺省默认 DEFAULT_GROUP
spring.cloud.nacos.config.shared-configs[0].group=GROUP_APP1

# 配置 Data Id 在配置变更时，是否动态刷新，缺省默认 false
spring.cloud.nacos.config.shared-configs[0].refresh=true
```

可以看到：

- 通过 `spring.cloud.nacos.config.shared-configs[n].data-id` 来支持多个共享 Data Id 的配置。

- 通过 `spring.cloud.nacos.config.shared-configs[n].group` 来配置自定义 Data Id 所在的组，不明确配置的话，默认是 DEFAULT_GROUP。
- 通过 `spring.cloud.nacos.config.shared-configs[n].refresh` 来控制该 Data Id 在配置变更时，是否支持应用中动态刷新，默认 false。

配置的优先级

Spring Cloud Alibaba 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-configs[n].data-id` 支持多个共享 Data Id 的配置
- B: 通过 `spring.cloud.nacos.config.extension-configs[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

完全关闭配置

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

Nacos Discovery

Spring Cloud Alibaba Nacos Discovery

该项目通过自动配置以及其他 Spring 编程模型的习惯用法为 Spring Boot 应用程序在服务注册与发现方面提供和 Nacos 的无缝集成。 通过一些简单的注解，您可以快速来注册一个服务，并使用经过双十一考验的 Nacos 组件来作为大规模分布式系统的服务注册中心。

服务注册发现: Nacos Discovery Starter

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 Starter 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 Starter 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-注册到 Nacos 。Nacos 的获取和启动方式可以参考 Nacos 官网。

如何引入 Nacos Discovery Starter

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

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

启动一个 **Provider** 应用

以下步骤向您展示了如何将一个服务注册到 Nacos。

1. pom.xml 的配置。一个完整的 pom.xml 配置如下所示：

```
<?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-instance"

        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <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>
```

1. 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

```
server.port=8081

spring.application.name=nacos-producer

spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848

management.endpoints.web.exposure.include=*
```

Note	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 f

1. 启动 Provider 示例。如下所示：

```
@SpringBootApplication

@EnableDiscoveryClient

public class NacosProviderDemoApplication {



    public static void main(String[] args) {

        SpringApplication.run(NacosProducerDemoApplication.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。

Note	再启动 Provider 应用之前 请先将 Nacos 服务启动。具体启动方式可参考 Nacos 官网。

服务的 EndPoint

spring-cloud-starter-alibaba-nacos-discovery 在实现的时候提供了一个

EndPoint,EndPoint 的访问地址为 http://ip:port/actuator/nacos-discovery。

EndPoint 的信息主要提供了两类:

```
1、subscribe：显示了当前有哪些服务订阅者

2、NacosDiscoveryProperties：显示了当前服务实例关于 Nacos 的基础配置
```

一个服务实例访问 EndPoint 的信息如下所示：


```
{
  "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-producer",
    "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": ""
 }
}
```

启动一个 Consumer 应用

Consumer 的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 RestTemolate 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：

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

```
@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 RestTemolate 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(),serviceInsta
nce.getPort(),appName);
            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。

Note	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
```

关于 Nacos Starter 更多的配置项信息

更多关于 spring-cloud-starter-alibaba-nacos-discovery 的 starter 配置项如下所示:

配置项	Key	默认值	
服务端地址	spring.cloud.nacos.discovery.server-addr	无	Na Se 启

配置项	Key	默认值	
			的址
服务名	spring.cloud.nacos.discovery.service	${spring.application.name}	给服
服务分组	spring.cloud.nacos.discovery.group	DEFAULT_GROUP	设所组
权重	spring.cloud.nacos.discovery.weight	1	取1 10值权
网卡名	spring.cloud.nacos.discovery.network-interface	无	当配注IP网应地果未则第卡
注册的 IP 地址	spring.cloud.nacos.discovery.ip	无	优高
注册的端口	spring.cloud.nacos.discovery.port	-1	默下置动
命名空间	spring.cloud.nacos.discovery.namespace	无	常之同注分例测

配置项	Key	默认值	
			和 境 （ 置 务 等
AccessKey	spring.cloud.nacos.discovery.access-key	无	当 里 阿 面 云
SecretKey	spring.cloud.nacos.discovery.secret-key	无	当 里 阿 面 云 码
Metadata	spring.cloud.nacos.discovery.metadata	无	使 格 置 可 自 要 一 务 元 息
日志文件名	spring.cloud.nacos.discovery.log-name	无	
集群	spring.cloud.nacos.discovery.cluster-name	DEFAULT	配 Na 群
接入点	spring.cloud.nacos.discovery.enpoint	UTF-8	地 个 入 名 此 以

配置项	Key	默认值	
			拿 端
是否集成 Ribbon	ribbon.nacos.enabled	true	一 置 tr 可
是否开启 Nacos Watch	spring.cloud.nacos.discovery.watch.enabled	true	可 成 来 wa

Sentinel

Spring Cloud Alibaba Sentinel

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。

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。

```
<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:

```
@SpringBootApplication
public class Application {

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

@Service
public class TestService {

    @SentinelResource(value = "sayHello")
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}

@RestController
public class TestController {

    @Autowired
    private TestService service;

    @GetMapping(value = "/hello/{name}")
    public String apiHello(@PathVariable String name) {
        return service.sayHello(name);
    }
}
```

`@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 `sayHello` 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（注意有方法签名要求），更多内容可以参考 Sentinel 注解支持文档。若不配置 `blockHandler`、`fallback` 等函数，则被流控降级时方法会直接抛出对应的 BlockException；若方法未定义 `throws BlockException` 则会被 JVM 包装一层 `UndeclaredThrowableException`。

注：一般推荐将 `@SentinelResource` 注解加到服务实现上，而在 Web 层直接使用 Spring Cloud Alibaba 自带的 Web 埋点适配。Sentinel Web 适配同样支持配置自定义流控处理逻辑，参考 相关文档。

以上例子都是在 Web Servlet 环境下使用的。Sentinel 目前已经支持 Spring 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。

```
@SpringBootApplication
public class Application {

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

}

@RestController
public class TestController {

    @GetMapping("/mono")
    public Mono<String> mono() {
         return Mono.just("simple string");
    }
```

```
}
```

当 Spring WebFlux 应用接入 Sentinel starter 后，所有的 URL 就自动成为 Sentinel 中的埋点资源，可以针对某个 URL 进行流控。

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 1. 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 Spring Boot 应用，以 Spring Boot 的方式运行 jar 包即可。

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

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

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 控制台添加了一个限流规则，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 控制台文档 以及 Sentinel FAQ

Feign 支持

Sentinel 适配了 Feign 组件。如果想使用，除了引入 `spring-cloud-starter-alibaba-sentinel` 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 `spring-cloud-starter-openfeign` 依赖使 Sentinel starter 中的自动化配置类生效：

```
<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：

```
@FeignClient(name = "service-provider", fallback =
EchoServiceFallback.class, configuration = FeignConfiguration.class)
public interface EchoService {
    @RequestMapping(value = "/echo/{str}", method = RequestMethod.GET)
    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";
    }
}
```

Note	Feign 对应的接口中的资源名策略定义：httpmethod:protocol://requesturl。@FeignClient 注解中的
	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}。

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。

```
@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：

```
public class ExceptionUtil {
    public static ClientHttpResponse handleException(HttpRequest request, byte[] body, ClientHttpRequestExecution execution, BlockException exception) {
        ...
    }
}
```

Note	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

Note	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，是 GET:https://www.taobao.com 以及 GET:https://www.taobao.com/test

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：

```
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=com.alibaba.cloud.examples.JsonFlowRuleListConverter
#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 。

Note	ds1, ds2, ds3, ds4 是 ReadableDataSource 的名字，可随意编写。后面的 file ， zk ， nacos , apollo 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。注意数据源的依赖要单独引 如 sentinel-datasource-nacos)。

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=com.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)。

Note	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。
Note	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。
Note	如果规则加载没有生效，有可能是 jdk 版本导致的，请关注 759 issue 的处理。

关于 Sentinel 动态数据源的实现原理，参考： 动态规则扩展

Zuul 支持

参考 Sentinel 网关限流文档

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：

```
<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>
```

同时请将 spring.cloud.sentinel.filter.enabled 配置项置为 false（若在网关流控控制台上看到了 URL 资源，就是此配置项没有置为 false）。

Spring Cloud Gateway 支持

参考 Sentinel 网关限流文档

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：

```
<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>
```

同时请将 `spring.cloud.sentinel.filter.enabled` 配置项置为 false（若在网关流控控制台上看到了 URL 资源，就是此配置项没有置为 false）。

Endpoint 支持

在使用 Endpoint 特性之前需要在 Maven 中添加 `spring-boot-starter-actuator` 依赖，并在配置中允许 Endpoints 的访问。

- Spring Boot 1.x 中添加配置 `management.security.enabled=false`。暴露的 endpoint 路径为 `/sentinel`

- Spring Boot 2.x 中添加配置 `management.endpoints.web.exposure.include=*`。暴露的 endpoint 路径为 `/actuator/sentinel`

Sentinel Endpoint 里暴露的信息非常有用。包括当前应用的所有规则信息、日志目录、当前实例的 IP，Sentinel Dashboard 地址，Block Page，应用与 Sentinel Dashboard 的心跳频率等等信息。

配置

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	
`UrlCleaner`	`WebCallbackManager.setUrlCleaner(urlCleaner)`	资 (资 （ 满 /f 的 都 /f 资 下
`UrlBlockHandler`	`WebCallbackManager.setUrlBlockHandler(urlBlockHandler)`	自 流 辑
`RequestOriginParser`	`WebCallbackManager.setRequestOriginParser(requestOriginParser)`	设 信

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

配置项	含义	默认值
spring.application.name or project.name	Sentinel 项目名	
spring.cloud.sentinel.enabled	Sentinel 自动化配置是否生效	true
spring.cloud.sentinel.eager	是否提前触发 Sentinel 初始化	false
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.dashboard	Sentinel 控制台地址	
spring.cloud.sentinel.transport.heartbeat-interval-ms	应用与 Sentinel 控制台的心跳间隔时间	
spring.cloud.sentinel.transport.client-ip	此配置的客户端 IP 将被注册到 Sentinel Server 端	
spring.cloud.sentinel.filter.order	Servlet Filter 的加载顺序。Starter 内部会构造这个 filter	Integer.MI
spring.cloud.sentinel.filter.url-patterns	数据类型是数组。表示 Servlet Filter 的 url pattern 集合	/*
spring.cloud.sentinel.filter.enabled	Enable to instance CommonFilter	true
spring.cloud.sentinel.metric.charset	metric 文件字符集	UTF-8
spring.cloud.sentinel.metric.file-single-size	Sentinel metric 单个文件的大小	

配置项	含义	默认值
spring.cloud.sentinel.metric.file-total-count	Sentinel metric 总文件数量	
spring.cloud.sentinel.log.dir	Sentinel 日志文件所在的目录	
spring.cloud.sentinel.log.switch-pid	Sentinel 日志文件名是否需要带上 pid	false
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-factor	WarmUp 模式中的 冷启动因子	3
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 流控处理逻辑 (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.response-body	Spring Cloud Gateway 响应模式为 'response' 模式对应的响应内容	

配置项	含义	默认值
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.sentinel.scg.fallback.content-type	Spring Cloud Gateway 响应模式为 'response' 模式对应的 content-type	applicatio
Note	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	

RocketMQ

Spring Cloud Alibaba RocketMQ Binder

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

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

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

- 启动 Broker

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

- 发送、接收消息

发送消息：

```
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：

```
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

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

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 1. 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：

```
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。

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：

```
<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：

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

Spring Cloud Alibaba RocketMQ Binder 实现

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。

Note	在使用 RocketMQ Binder 的同时也可以配置 rocketmq.** 用于触发 RocketMQ Spring 相关 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，

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

MessageSource 支持

目前 RocketMQ 已经支持 MessageSource，可以进行消息的拉取，例子如下：

```
@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();

  }

}
```

配置选项

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`.

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`.

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`.

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。

Note	0.1.2 & 0.2.2 & 0.9.0 及以上才支持该功能

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

Note	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 2. NameServer 的获取(配置中请去掉 http:// 前缀)

Dubbo Spring Cloud

简介

Dubbo Spring Cloud 基于 Dubbo Spring Boot 2.7.1[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。

Dubbo Spring Cloud 首个 Preview Release，随同 Spring Cloud Alibaba `0.2.2.RELEASE` 和 `0.9.0.RELEASE` 一同发布[2]， 分别对应 Spring Cloud Finchley[3] 与 Greenwich[4] (下文分别简称为 “F” 版 和 “G” 版) 。

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]

功能组件	Spring Cloud	Dubbo Spring Cloud
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

示例代码

---

[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