

目 录

[致谢](#)

[README](#)

[首页](#)

[project](#)

[Quick Start](#)

[Spring Cloud Config](#)

[Spring Cloud Netflix](#)

[Spring Cloud Bus](#)

[Spring Cloud Cloud Foundry Service Broker](#)

[Spring Cloud Cloud Foundry Service Broker](#)

[Spring Cloud Consul](#)

[Spring Cloud Security](#)

[Spring Cloud Sleuth](#)

[Spring Cloud Data Flow](#)

[Spring Cloud Stream](#)

[Spring Cloud Stream Modules](#)

[Spring Cloud Task](#)

[Spring Cloud Zookeeper](#)

[Spring Cloud for Amazon Web Services](#)

[Spring Cloud Connectors](#)

[Spring Cloud CLI](#)

[Spring Cloud翻译文档](#)

[Spring Cloud Eureka](#)

[声明式REST客户端Feign](#)

[Spring Cloud Bus](#)

[Spring Cloud Config](#)

[Spring Cloud sleuth](#)

[Spring Cloud stream](#)

[Spring Cloud zuul](#)

[Spring Cloud ribbon](#)

[源码分析](#)

致谢

当前文档《Spring Cloud中文文档》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-22。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/spring-cloud-docs>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

README

- [spring-cloud-docs](#)
 - [来源\(书栈小编注\)](#)

spring-cloud-docs

`docs.springcloud.cn`文档

来源(书栈小编注)

<https://github.com/SpringCloud/spring-cloud-docs>

首页

- [Spring Cloud中国社区](#)
 - [为什么要发起Spring Cloud中国社区](#)
 - [spring cloud目前国内使用情况](#)
 - [贡献文档](#)
 - [捐赠社区发展](#)

Spring Cloud中国社区

欢迎来到，Spring Cloud中国社区文档地址，欢迎贡献优质博客和翻译文档

为什么要发起Spring Cloud中国社区

Spring Cloud发展到2016年，国内关注的人越来越多，但是相应学习交流的平台和材料比较分散，不利于学习交流，因此Spring Cloud中国社区应运而生。于是Spring Cloud中国社区由许进联合翟永超，周立发起Spring cloud中国社区，是国内首个Spring Cloud构建微服务架构的交流社区。我们致力于为Spring Boot或Spring Cloud技术人员提供分享和交流的平台，推动Spring Cloud在中国的普及和应用。 欢迎CTO、架构师、开发者等，在这里学习与交流使用Spring Cloud的实战经验。 目前QQ群人数:2000+, 微信群:600+

Spring Cloud中国社区精英群:415028731

Spring cloud中国社区QQ群: 470962790(已满)

Spring Cloud中国社区官网:<http://springcloud.cn>

Spring Cloud中国社区论坛:<http://bbs.springcloud.cn>

Spring Cloud中国社区文档:<http://docs.springcloud.cn>

spring cloud目前国内使用情况

1. 中国联通子公司
http://flp.baidu.com/feedland/video/?entry=box_searchbox_feed&id=144115189637730162&from=timeline&isappinstalled=0
2. 上海米么金服
3. 指点无限（北京）科技有限公司

4. 易保软件 目前在定制开发中
<http://www.ebaotech.com/cn/>
5. 广州简法网络
6. 深圳睿云智合科技有限公司
持续交付产品基于Spring Cloud研发 <http://www.wise2c.com>
7. 猪八戒网, 目前调研中
8. 上海云首科技有限公司
9. 华为
整合netty进来用rpc 包括nerflix那套东西 需要注意的是sleuth traceid的传递需要自己写。tps在物理机上能突破20w
0. 东软
1. 南京云帐房网络科技有限公司
2. 四众互联(北京)网络科技有限公司
3. 深圳摩令技术科技有限公司
4. 广州万表网
5. 视觉中国
6. 上海秦苍信息科技有限公司-买单侠
7. 爱油科技(大连)有限公司
[爱油科技基于SpringCloud的微服务实践](#)
数据在统计之中, 会一直持续更新, 敬请期待!

贡献文档

欢迎大家贡献, 优质的博客文章和翻译文档。

捐赠社区发展

如果你觉得, Spring Cloud中国社区还可以, 为了更好的发展, 你可以捐赠社区, 点击下面的打赏捐赠, 捐赠的钱将用于社区发展和线下meeting up。 支付宝账号:Software_King@qq.com

project

- [Quick Start](#)
- [Spring Cloud Config](#)
- [Spring Cloud Netflix](#)
- [Spring Cloud Bus](#)
- [Spring Cloud Cloud Foundry Service Broker](#)
- [Spring Cloud Cloud Foundry Service Broker](#)
- [Spring Cloud Consul](#)
- [Spring Cloud Security](#)
- [Spring Cloud Sleuth](#)
- [Spring Cloud Data Flow](#)
- [Spring Cloud Stream](#)
- [Spring Cloud Stream Modules](#)
- [Spring Cloud Task](#)
- [Spring Cloud Zookeeper](#)
- [Spring Cloud for Amazon Web Services](#)
- [Spring Cloud Connectors](#)
- [Spring Cloud CLI](#)

Quick Start

- [Spring Cloud](#)
 - [Quick Start](#)
 - [Features](#)
 - [Main Projects](#)
 - [Release Trains](#)
 - [Sample Projects](#)

Spring Cloud

Spring Cloud为开发人员提供了工具，用以快速的在分布式系统中建立一些通用方案（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线，一次性令牌，全局锁，领导选举，分布式会话，集群状态）。协调分布式系统有固定样板模型，使用Spring Cloud开发人员可以快速地搭建基于实现了这些模型的服务和应用程序。他们将在任何分布式环境中工作，包括开发人员自己的笔记本电脑，裸机数据中心，和管理的平台，如云计算。

For full documentation visit spring.io.

Quick Start

基于Spring Boot构建Spring Cloud，可以在类路径中自动引入提升应用程序性能的一组类库。您可以利用默认配置来快速启动，然后当您需要时，您可以配置或扩展以创建自定义解决方案。

发布版的版本号要在`artifact:spring-cloud-dependencies`

中明确使用，其他的版本标签会从parent中获取，你可以使用`dependencyManagement`去做版本依赖管理，下面是使用最新版`config client`和`eureka`的配置用例。

```

1. <parent>
2.   <groupId>org.springframework.boot</groupId>
3.   <artifactId>spring-boot-starter-parent</artifactId>
4.   <version>1.3.5.RELEASE</version>
5. </parent>
6. <dependencyManagement>
7.   <dependencies>
8.     <dependency>
9.       <groupId>org.springframework.cloud</groupId>
10.      <artifactId>spring-cloud-dependencies</artifactId>
11.      <version>Brixton.SR1</version>
12.      <type>pom</type>
13.      <scope>import</scope>
14.    </dependency>

```



```

15.     </dependencies>
16. </dependencyManagement>
17. <dependencies>
18.     <dependency>
19.         <groupId>org.springframework.cloud</groupId>
20.         <artifactId>spring-cloud-starter-config</artifactId>
21.     </dependency>
22.     <dependency>
23.         <groupId>org.springframework.cloud</groupId>
24.         <artifactId>spring-cloud-starter-eureka</artifactId>
25.     </dependency>
26. </dependencies>

```

Features

Spring Cloud 侧重提供良好的开箱即用体验

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

Spring Cloud 提供一个发布方法，通常你获得很多特性仅是由于一个classpath的变化或注解，下面是一个discovery client的例子

```

1. @SpringBootApplication
2. @EnableDiscoveryClient
3. public class Application {
4.     public static void main(String[] args) {
5.         SpringApplication.run(Application.class, args);
6.     }
7. }

```

Main Projects

- [Spring Cloud Config](#)

利用git集中管理程序的配置。配置资源直接映射到Spring的Environment，另一方面如果有需要也可以被非Spring应用使用。

- [Spring Cloud Netflix](#)

集成了许多Netflix的开源软件(Eureka, Hystrix, Zuul, Archaius, etc)

- [Spring Cloud Bus](#)

一个事件总线，利用分布式消息将服务和服务实例连接在一起，用于在一个集群中传播状态的变化，比如配置更改事件。

- [Spring Cloud for Cloud Foundry](#)

利用Pivotal Cloudfoundry集成你的应用程序，提供了一个服务发现的实现也使得它很容易实现SSO和OAuth2保护资源，并创建一个Cloudfoundry服务代理。

- [Spring Cloud Cloud Foundry Service Broker](#)

为建立管理云托管服务的服务代理提供了一个起点。

- [Spring Cloud Cluster](#)

基于Zookeeper, Redis, Hazelcast, Consul实现的领导选举和平民状态模式的抽象和实现。

- [Spring Cloud Consul](#)

基于Hashicorp Consul实现的服务发现和配置管理。

- [Spring Cloud Security](#)

在Zuul代理中为OAuth2 rest客户端和认证头转发提供负载均衡

- [Spring Cloud Sleuth](#)

Spring Cloud 应用的分布式追踪系统，和Zipkin, HTrace, ELK兼容。

- [Spring Cloud Data Flow](#)

一个云本地程序和操作模型，组成数据微服务在一个结构化的平台上。

- [Spring Cloud Stream](#)

基于 Redis, Rabbit, Kafka实现的消息微服务，简单声明模型用以在Spring Cloud应用中收发消息。

- [Spring Cloud Stream Modules](#)

Spring Cloud Stream Modules 可用于创建消息驱动的微服务。

- [Spring Cloud Task](#)

短生命周期的微服务，为SpringBooot应用简单声明添加功能和非功能特性。

- [Spring Cloud Zookeeper](#)

服务发现和配置管理基于Apache Zookeeper。

- [Spring Cloud for Amazon Web Services](#)

易与托管的亚马逊网络服务的集成。它提供了一个方便的方式来与AWS提供的服务使用知名的Spring语法和API进行交互，如消息或缓存API。开发人员可以在托管服务周围建立他们的应用程序，而不必关心基础设施或维护。

- [Spring Cloud Connectors](#)

便于PaaS应用在各种平台上连接到后端像数据库和消息经纪服务。

- [Spring Cloud Starters](#)

SpringBoot风格starter项目，用以简化Spring Cloud客户端的依赖管理。（项目已经终止并且在Angel.SR2后的版本和其他项目合并）

- [Spring Cloud CLI](#)

Spring Boot CLI 插件用Groovy快速的创建Spring Cloud组件应用。

Release Trains

Spring Cloud 是由自主项目组成的，原则上采用不同的发行节奏。管理投资组合的BOM（物料清单）见下文。发布历程有名称，而不是版本，以避免与子项目混乱。名字是伦敦地铁站的名字的字母序列排序（这样你就可以知道时间顺序），“Angel”是第一个版本，“Brixton”是第二。当子项目的发行版本中累计了大量危机的bug，或者有一个严重的bug需要让每个人可见，发布列车将推出“服务发布（service releases）”结尾的名字“.SRX”，其中“X”是一个数字。

Release train contents:

Component	Angel.SR6	Brixton.SR1	Brixton.BUILD-SNAPSHOT
spring-cloud-aws	1.0.4.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-bus	1.0.3.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT

spring-cloud-cli	1.0.6.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-commons	1.0.5.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-config	1.0.4.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-netflix	1.0.7.RELEASE	1.1.2.RELEASE	1.1.3.BUILD-SNAPSHOT
spring-cloud-security	1.0.3.RELEASE	1.1.0.RELEASE	1.1.1.BUILD-SNAPSHOT
spring-cloud-starters	1.0.6.RELEASE		
spring-cloud-cloudfoundry		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-cluster		1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-consul		1.0.1.RELEASE	1.0.2.BUILD-SNAPSHOT
spring-cloud-sleuth		1.0.1.RELEASE	1.0.2.BUILD-SNAPSHOT
spring-cloud-stream		1.0.2.RELEASE	1.0.3.BUILD-SNAPSHOT
spring-cloud-zookeeper		1.0.1.RELEASE	1.0.2.BUILD-SNAPSHOT
spring-boot	1.2.8.RELEASE	1.3.5.RELEASE	1.3.5.RELEASE
spring-cloud-stream-app-starters*			1.0.0.BUILD-SNAPSHOT
spring-cloud-task*			1.0.0.BUILD-SNAPSHOT

(*) 这些项目在他们被发布之前还不是 Brixton 的一部分

Angel基于Spring Boot 1.2.x, 某些部分和1.3.x不兼容。Brixton基于Spring Boot 1.3.x 尽可能的兼容1.2.x, 一些基于Angel的库和基于Angel的大部分应用可以很好的运行在Brixton上, 但spring-cloud-security 1.0.x 用到的OAuth2特性将需要全部修改 (他们大多搬到Spring Boot1.3.0)。

使用你的依赖管理工具来控制版本。如果你正在使用Maven, 记住第一个版本宣布获胜, 所以申报材料清单的顺序, 与第一个通常是最新的 (例如, 如果你想使用Spring Boot 1.3.6启动Brixton.RELEASE, 把启动BOM放在第一)。同样的规则适用于Gradle, 如果你使用Spring的依赖管理插件。

NOTE: starting after Brixton.M4 the release train contains a `spring-cloud-starter-dependencies` as well as the `spring-cloud-starter-parent`. Use the parent as you would the `spring-boot-starter-parent` (if you are using Maven). If you only need dependency management,

the “dependencies” version is a BOM-only version of the same thing (it just contains dependency management and no plugin declarations or direct references to Spring or Spring Boot). If you are using the Spring Boot parent POM, then you can use the BOM from Spring Cloud.

Sample Projects

[Config Server](#)

[Service Registry](#)

[Circuit Breaker Dashboard](#)

[Business Application](#)(Customers and Stores)

[OAuth2 Authorization Server](#)

[OAuth2 SSO Client](#)

[Integration Test Samples](#)

Spring Cloud Config

- [Spring Cloud Config](#)
 - [Features](#)
 - [Quick Start](#)
 - [Sample Projects](#)

Spring Cloud Config

Spring Cloud Config在分布式系统中为外部配置提供服务端和客户端支持。通过Config Server你可以集中管理应用程序的外部配置文件。客户端和服务端的键值对概念与 `Spring Environment` 和 `PropertySource` 相同，所以十分适合Spring应用，另一方面可以适用于任何语言的应用程序。作为一个应用程序将通过部署管道从开发到测试到生产，你可以管理这些环境的配置，可以确定应用都需要运行时迁移。服务器的后端存储的默认实现使用Git，所以容易支持标记版本的配置环境，以及一系列管理工具访问内容。它很容易添加替代的实现，并将它们插入到Spring配置中。

For full documentation visit [spring cloud config](#).

Features

Spring Cloud Config Server features:

- HTTP API的外部资源配置（名称-值对，或等效的YAML内容）
- 对属性值（对称或非对称）进行加密和解密
- 使用 `@EnableConfigServer` 注解嵌入Spring Boot应用

Config Client features (for Spring applications):

- 绑定到配置服务器，并使用远程属性源初始化Spring环境
- 对属性值（对称或非对称）进行加密和解密

Quick Start

项目中使用 `spring-cloud-config` 推荐基于一个依赖管理系统—下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```
1. <dependencyManagement>
2.   <dependencies>
3.     <dependency>
4.       <groupId>org.springframework.cloud</groupId>
```

```

5.         <artifactId>spring-cloud-config</artifactId>
6.         <version>1.1.1.RELEASE</version>
7.         <type>pom</type>
8.         <scope>import</scope>
9.     </dependency>
10. </dependencies>
11. </dependencyManagement>
12. <dependencies>
13.     <dependency>
14.         <groupId>org.springframework.cloud</groupId>
15.         <artifactId>spring-cloud-starter-config</artifactId>
16.     </dependency>
17. </dependencies>

```

只要classpath中包含Spring Boot Actuator和Spring Config Client, Spring Boot应用将尝试连接配置服务 `http://localhost:8888` (`spring.cloud.config.uri` 默认值)

```

1. @Configuration
2. @EnableAutoConfiguration
3. @RestController
4. public class Application {
5.
6.     @Value("${config.name}")
7.     String name = "World";
8.
9.     @RequestMapping("/")
10.    public String home() {
11.        return "Hello " + name;
12.    }
13.
14.    public static void main(String[] args) {
15.        SpringApplication.run(Application.class, args);
16.    }
17.
18. }

```

范例中 `config.name` 的值 (或任何其他值) 可以来自本地配置或从远程配置服务器。配置服务器将优先默认。在应用程序中看 `/env` 端点, 看 `configServer` 资源文件。

要想运行你的服务, 需要依赖 `spring-cloud-config-server` 并且使用 `@EnableConfigServer` 注解。如果设置 `spring.config.name=configserver`, 应用将在8888端口启动, 数据来自样本库。你需要 `spring.cloud.config.server.git.uri` 为您自己的需求找到配置数据 (默认情况下它是一个Git仓库, 并且可以是一个本地文件路径 `file:..`)

Sample Projects

Config Server

Config Clients

Spring Cloud Netflix

- [Spring Cloud Netflix](#)
 - [Features](#)
 - [Quick Start](#)
 - [Sample Projects](#)

Spring Cloud Netflix

Spring Cloud Netflix提供了Netflix公司的开源软件（OSS）的整合，Spring Boot应用通过自动配置和绑定环境和其他spring模型风格。用一个简单的注释，你可以在你的应用中快速启用常见模式配置，并构建大型分布式系统，这些组件是经过Netflix公司生产环境考验的。该模式包括服务发现（Eureka）、断路器（Hystrix），智能路由（Zuul）和客户端负载均衡（Ribbon）..

For full documentation visit [spring cloud netflix](#).

Features

Spring Cloud Netflix features:

- 服务发现：可以在Eureka中注册实例，客户端可以发现使用Spring管理bean实例
- 服务发现：嵌入式Eureka服务可以通过声明java配置来创建
- 断路器：Hystrix客户端可以通过方法上的注解来创建
- 断路器：嵌入Hystrix仪表盘通过声明java配置
- 声明REST客户端：伪装创建一个动态接口装饰，使用JAX-RS或Spring MVC注解
- 客户端负载均衡：Ribbon
- 外部配置：Spring Environment和Archaius（配置管理API）搭建起一座桥梁。（使Netflix组件的本地配置能够使用Spring Boot习俗）
- 路由器和过滤器：自动注册Zuul的过滤器，和一个简单配置约定创建反向代理

Quick Start

项目中使用 `spring-cloud-netflix` 推荐基于一个依赖管理系统 — 下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```
1. <dependencyManagement>
2.   <dependencies>
3.     <dependency>
4.       <groupId>org.springframework.cloud</groupId>
5.       <artifactId>spring-cloud-netflix</artifactId>
```

```

6.         <version>1.1.2.RELEASE</version>
7.         <type>pom</type>
8.         <scope>import</scope>
9.     </dependency>
10. </dependencies>
11. </dependencyManagement>
12. <dependencies>
13.     <dependency>
14.         <groupId>org.springframework.cloud</groupId>
15.         <artifactId>spring-cloud-starter-eureka</artifactId>
16.     </dependency>
17. </dependencies>

```

只要classpath中包含Spring Cloud Netflix 和 Eureka Core, 所有应用了

`@EnableEurekaClient` 注解的Spring Boot应用将尝试连接Eureka服务
<http://localhost:8761> (`eureka.client.serviceUrl.defaultZone` 默认值)

```

1. @Configuration
2. @EnableAutoConfiguration
3. @EnableEurekaClient
4. @RestController
5. public class Application {
6.
7.     @RequestMapping("/")
8.     public String home() {
9.         return "Hello World";
10.    }
11.
12.    public static void main(String[] args) {
13.        SpringApplication.run(Application.class, args);
14.    }
15.
16. }

```

要运行你自己的服务, 需要使用 `spring-cloud-starter-eureka-server` 依赖和 `@EnableEurekaServer` 注解。

Sample Projects

[Eureka Server](#)

[Eureka Clients](#)

Spring Cloud Bus

- [Spring Cloud Bus](#)
 - [Quick Start](#)
 - [Sample Projects](#)

Spring Cloud Bus

Spring Cloud Bus 通过一个轻量级消息代理连接分布式系统的节点。这可以用于广播状态更改（如配置更改）或其他管理指令。当前唯一的实现方式是通过一个AMQP代理作为消息传输，但相同的基本特征（传输上的一些依赖）是其他传输的路线图

For full documentation visit [spring cloud bus](#).

Quick Start

项目中使用 `spring-cloud-bus` 推荐基于一个依赖管理系统 — 下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```

1. <dependencyManagement>
2.     <dependencies>
3.         <dependency>
4.             <groupId>org.springframework.cloud</groupId>
5.             <artifactId>spring-cloud-bus-parent</artifactId>
6.             <version>1.1.1.BUILD-SNAPSHOT</version>
7.             <type>pom</type>
8.             <scope>import</scope>
9.         </dependency>
10.    </dependencies>
11. </dependencyManagement>
12. <dependencies>
13.     <dependency>
14.         <groupId>org.springframework.cloud</groupId>
15.         <artifactId>spring-cloud-starter-bus-amqp</artifactId>
16.     </dependency>
17. </dependencies>
18. <repositories>
19.     <repository>
20.         <id>spring-snapshots</id>
21.         <name>Spring Snapshots</name>
22.         <url>https://repo.spring.io/libs-snapshot</url>
23.         <snapshots>
24.             <enabled>true</enabled>

```

```

25.         </snapshots>
26.     </repository>
27. </repositories>

```

只要classpath中包含AMQP和RabbitMQ，Spring Boot应用将尝试连接RabbitMQ服务 `http://localhost:5672` (`spring.rabbitmq.addresses` 默认值)

```

1. @Configuration
2. @EnableAutoConfiguration
3. @RestController
4. public class Application {
5.
6.     @RequestMapping("/")
7.     public String home() {
8.         return "Hello World";
9.     }
10.
11.     public static void main(String[] args) {
12.         SpringApplication.run(Application.class, args);
13.     }
14.
15. }

```

Sample Projects

Bus Clients

Spring Cloud Cloud Foundry Service Broker

- [Spring Cloud Cloud Foundry](#)
 - [Quick Start](#)

Spring Cloud Cloud Foundry

For full documentation visit [spring cloud cloud foundry](#).

Quick Start

Spring Cloud Cloud Foundry Service Broker

- [Spring Cloud Cloud Foundry Service Broker](#)
 - [Quick Start](#)

Spring Cloud Cloud Foundry Service Broker

For full documentation visit [spring cloud cloud foundry service broker](#).

Quick Start

Spring Cloud Consul

- [Spring Cloud Consul](#)
 - [Features](#)
 - [Quick Start](#)
 - [Sample Projects](#)

Spring Cloud Consul

Spring Cloud Consul提供了Consul的整合，Spring Boot应用通过自动配置和绑定环境和其他spring模型风格。用一个简单的注释，你可以在你的应用中快速启用常见模式配置，并构建大型分布式系统，这些组件是经过Netflix公司生产环境考验的。该模式包括服务发现，分布式配置和控制总线。

For full documentation visit [spring cloud consul](#).

Features

Spring Cloud Consul features:

- 服务发现：实例可以用Consul代理端注册，客户端可以发现使用Spring管理bean实例。
- 支持Ribbon，利用Spring Cloud Netflix实现客户端负载均衡。
- 支持Zuul，利用Spring Cloud Netflix实现动态路由和过滤。
- 分布式配置：利用Consul的Key/Value存储。
- 控制总线：使用Consul事件处理分布式控制事件

Quick Start

项目中使用 `spring-cloud-consul` 推荐基于一个依赖管理系统 — 下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```
1. <dependencies>
2.   <dependency>
3.     <groupId>org.springframework.cloud</groupId>
4.     <artifactId>spring-cloud-starter-consul-all</artifactId>
5.   </dependency>
6. </dependencies>
7.
8. <dependencyManagement>
9.   <dependencies>
```



```

10.         <dependency>
11.             <groupId>org.springframework.cloud</groupId>
12.             <artifactId>spring-cloud-consul-dependencies</artifactId>
13.             <version>1.0.1.RELEASE</version>
14.             <type>pom</type>
15.             <scope>import</scope>
16.         </dependency>
17.     </dependencies>
18. </dependencyManagement>

```

只要classpath中包含 Spring Cloud Consul和Consul API, 所有应用了

`@EnableDiscoveryClient` 注解的Spring Boot应用将尝试连接Consul的代理服

务 `http://localhost:8500` (`spring.cloud.consul.host` and `spring.cloud.consul.port` 默认值)

```

1. @Configuration
2. @EnableAutoConfiguration
3. @EnableDiscoveryClient
4. @RestController
5. public class Application {
6.
7.     @RequestMapping("/")
8.     public String home() {
9.         return "Hello World";
10.    }
11.
12.    public static void main(String[] args) {
13.        SpringApplication.run(Application.class, args);
14.    }
15.
16. }

```

本地Consul代理的运行. 参见[Consul agent documentation](#).

Sample Projects

[Consul Sample](#)

Spring Cloud Security

- [Spring Cloud Security](#)
 - [Features](#)
 - [Quick Start](#)
 - [Sample Projects](#)

Spring Cloud Security

Spring Cloud Security提供了一组原语义，用最小的代价来创建安全的应用和服务。通过统一管理中心，将应用自己授权给大型协作系统、远程组件。他在Cloud Foundry平台中也非常易用。基于 Spring Boot 和 Spring Security OAuth2我们可以快速的实现统一登录、令牌传递、令牌交换。

For full documentation visit [spring cloud security](#).

Features

Spring Cloud Security features:

- 在Zuul proxy中传递SSO tokens
- 资源服务器之间的传递tokens
- Feign客户端拦截器行为，如OAuth2RestTemplate (fetching tokens)
- 在Zuul proxy配置下游认证

Quick Start

项目中使用 `spring-cloud-security` 推荐基于一个依赖管理系统—下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```

1. <dependencies>
2.     <dependency>
3.         <groupId>org.springframework.cloud</groupId>
4.         <artifactId>spring-cloud-security</artifactId>
5.         <version>1.1.4.BUILD-SNAPSHOT</version>
6.     </dependency>
7. </dependencies><repositories>
8.     <repository>
9.         <id>spring-snapshots</id>
10.        <name>Spring Snapshots</name>
11.        <url>https://repo.spring.io/libs-snapshot</url>

```

```

12.         <snapshots>
13.             <enabled>true</enabled>
14.         </snapshots>
15.     </repository>
16. </repositories>

```

如果你的应用启动了 Spring Cloud Zuul 反向代理（使用了`@EnableZuulProxy`注解），然后你就可以向下转发OAuth2访问令牌到Zuul代理的服务中。因此，在单点登录增强就像下面这样简单

```

1. @SpringBootApplication
2. @EnableOAuth2Sso
3. @EnableZuulProxy
4. class Application {
5.
6. }

```

并且他将（除了登录用户和抓取令牌）批准认证令牌下游到 `/proxy/*` services，如果这些服务执行了 `@EnableResourceServer` 注解他们就会在标准头中得到一个有效的令牌

Sample Projects

- [SSO](#)
- [Auth Server](#)
- [SSO Groovy](#)
- [Resource server](#)

Spring Cloud Sleuth

- [Spring Cloud Sleuth](#)
 - [Features](#)
 - [Quick Start](#)
 - [Sample Projects](#)

Spring Cloud Sleuth

Spring Cloud Sleuth为Spring Cloud提供了分布式追踪方案，借用了Dapper，Zipkin和HTrace。对于大多数用户来说Sleuth应该是看不见的，与外部系统的相互作用是自动的。您可以简单地在日志中捕获数据，或将数据发送到远程收集服务。

For full documentation visit [spring cloud sleuth](#).

Features

Span是一个基本单位，例如，发送一个RPC是一个新的Span。Span是由一个64位的SpanID和一个64位的traceID组成，Span也有其他数据，如描述，键值注释，SpanID，processID（通常是IP地址）。Span有开始和停止，并且跟踪他们的时间信息。一旦你创建一个Span，你必须在未来的某一点停止它。一组Span形成一个树状结构称为一个跟踪，例如，如果运行一个分布式大数据存储，则可能由一个放请求形成一个跟踪。

Spring Cloud Sleuth features:

- 在Slf4J的MDC中添加traceId和spanId，所以在日志汇总处你可以提取trace或span信息。
- 提供了一个抽象数据模型：traces，spans (forming a DAG)，annotations，key-value annotations。轻易的基于HTrace，和Zipkin (Dapper)兼容。
- Spring应用通用的入口和出口工具(servlet filter, rest template, scheduled actions, message channels, zuul filters, feign client)。
- 如果激活 `spring-cloud-sleuth-zipkin`，应用程序将生成并通过HTTP收集兼容Zipkin的traces。默认情况下发送到本地的Zipkin，可以通过 `spring.zipkin.[host,port]` 去配置正确的地址。

Quick Start

项目中使用 `spring-cloud-sleuth` 推荐基于一个依赖管理系统—下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```
1. <dependencyManagement>
```

```

2.     <dependencies>
3.         <dependency>
4.             <groupId>org.springframework.cloud</groupId>
5.             <artifactId>spring-cloud-sleuth</artifactId>
6.             <version>1.1.0.BUILD-SNAPSHOT</version>
7.             <type>pom</type>
8.             <scope>import</scope>
9.         </dependency>
10.    </dependencies>
11. </dependencyManagement>
12. <dependencies>
13.     <dependency>
14.         <groupId>org.springframework.cloud</groupId>
15.         <artifactId>spring-cloud-starter-sleuth</artifactId>
16.     </dependency>
17. </dependencies><repositories>
18.     <repository>
19.         <id>spring-snapshots</id>
20.         <name>Spring Snapshots</name>
21.         <url>https://repo.spring.io/libs-snapshot</url>
22.         <snapshots>
23.             <enabled>true</enabled>
24.         </snapshots>
25.     </repository>
26. </repositories>

```

只要classpath中包含Spring Cloud Sleuth, Spring Boot应用将产生trace数据。

```

1. @SpringBootApplication
2. @RestController
3. public class Application {
4.
5.     private static Logger log = LoggerFactory.getLogger(DemoController.class);
6.
7.     @RequestMapping("/")
8.     public String home() {
9.         log.info("Handling home");
10.        return "Hello World";
11.    }
12.
13.    public static void main(String[] args) {
14.        SpringApplication.run(Application.class, args);
15.    }
16.
17. }

```

运行这个程序并进入主页, 你将会看到日志中的traceId和spanId, 如果这个程序调用了另一个

(例如RestTemplate) 它将在headers中发送跟踪数据，如果接收器是一个Sleuth应用你会持续看到trace。

NOTE: instead of logging the request in the handler explicitly, you could set
`logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG`

NOTE: Set `spring.application.name=bar` *(for instance) to see the service name as well as the trace and span ids.*

Sample Projects

[Simple HTTP app](#) that calls back to itself

[Using Zipkin](#) to collect traces

[Messaging with Spring Integration](#)

Spring Cloud Data Flow

- [Spring Cloud Data Flow](#)
 - [Quick Start](#)

Spring Cloud Data Flow

Spring Cloud Data Flow是一种原生云业务流程服务，在现代运行时组合数据的微服务。开发人员可以创建和编排数据管道常见的用例如数据采集，实时分析，数据导入/导出

For full documentation visit [spring cloud data flow](#).

Quick Start

Spring Cloud Stream

- [Spring Cloud Stream](#)
 - [Quick Start](#)
 - [Sample Projects](#)
 - [Related Projects](#)

Spring Cloud Stream

Spring Cloud Stream是一个构建消息驱动的微服务框架。Spring Cloud Stream构建在Spring Boot之上用以创建DevOps友好的微服务，并且Spring Integration提供了和消息代理的连接。Spring Cloud Stream提供消息代理的自用配置，引入发布订阅的语义概念，引入不同的中间件厂商通用的消费组和分区，这些自用配置提供了创建流处理应用的基础。

添加`@EnableBinding`注解在你的程序中，被`@StreamListener`修饰的方法可以立即连接到消息代理，你将收到流处理事件。

For full documentation visit [spring cloud stream](#).

Quick Start

项目中使用 `spring-cloud-stream` 推荐基于一个依赖管理系统 — 下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```
1. <dependencyManagement>
2.     <dependencies>
3.         <dependency>
4.             <groupId>org.springframework.cloud</groupId>
5.             <artifactId>spring-cloud-stream-dependencies</artifactId>
6.             <version>1.0.2.RELEASE</version>
7.             <type>pom</type>
8.             <scope>import</scope>
9.         </dependency>
10.    </dependencies>
11. </dependencyManagement>
12. <dependencies>
13.     <dependency>
14.         <groupId>org.springframework.cloud</groupId>
15.         <artifactId>spring-cloud-stream</artifactId>
16.     </dependency>
17.     <dependency>
18.         <groupId>org.springframework.cloud</groupId>
```



```

19.         <artifactId>spring-cloud-starter-stream-kafka</artifactId>
20.     </dependency>
21. </dependencies>

```

只要classpath中包含 Spring Cloud Stream和Spring Cloud Stream binder，并且被 `@EnableBinding` 修饰，应用将通过总线绑定一个外部代理（Rabbit MQ或Kafka，取决于你的选择）。示例应用：

```

1. @SpringBootApplication
2. @EnableBinding(Source.class)
3. public class StreamdemoApplication {
4.
5.     public static void main(String[] args) {
6.         SpringApplication.run(StreamdemoApplication.class, args);
7.     }
8.
9.     @Bean
10.    @InboundChannelAdapter(value = Source.OUTPUT)
11.    public MessageSource<String> timerMessageSource() {
12.        return () -> new GenericMessage<>(new SimpleDateFormat().format(new Date()));
13.    }
14.
15. }

```

确定应用运行的时候Kafka同时运行，你可以看 `kafka-console-consumer.sh` kafka提供的实用工具，用来监控消息发送。

Sample Projects

[Source](#)

[Sink](#)

[Transformer](#)

[Multi-binder](#)

[RxJava Processor](#)

Related Projects

[Spring Cloud Stream Applications](#)

Spring Cloud Data Flow

Spring XD

Spring Cloud Stream Modules

- [Spring Cloud Stream Applications](#)
 - [Quick Start](#)

Spring Cloud Stream Applications

For full documentation visit [spring cloud stream applications](#).

Quick Start

Spring Cloud Task

- [Spring Cloud Task](#)
 - [Quick Start](#)

Spring Cloud Task

For full documentation visit [spring cloud task](#).

Quick Start

Spring Cloud Zookeeper

- [Spring Cloud Zookeeper](#)
 - [Features](#)
 - [Quick Start](#)

Spring Cloud Zookeeper

Spring Cloud Zookeeper提供了Zookeeper的整合，Spring Boot应用通过自动配置和绑定环境和其他spring模型风格。用一个简单的注释，你可以在你的应用中快速启用常见模式配置，并构建大型分布式系统。该模式包括服务发现和分布式配置。

For full documentation visit [spring cloud zookeeper](#).

Features

Spring Cloud Zookeeper features:

- 服务发现：可以在Zookeeper中注册实例，客户端可以发现使用Spring管理bean实例
- 支持Ribbon，客户端负载均衡
- 支持Zuul，动态路由和过滤器
- 分布式配置：利用Zookeeper存取数据

Quick Start

项目中使用 `spring-cloud-zookeeper` 推荐基于一个依赖管理系统 — 下面的代码段可以被复制和粘贴到您的构建。需要帮助吗？看看我们基于[Maven](#)和[Gradle](#)构建的入门指南。

```

1. <dependencyManagement>
2.   <dependencies>
3.     <dependency>
4.       <groupId>org.springframework.cloud</groupId>
5.       <artifactId>spring-cloud-zookeeper-dependencies</artifactId>
6.       <version>1.0.1.RELEASE</version>
7.       <type>pom</type>
8.       <scope>import</scope>
9.     </dependency>
10.  </dependencies>
11. </dependencyManagement>
12. <dependencies>
13.   <dependency>
14.     <groupId>org.springframework.cloud</groupId>

```

```

14.         <artifactId>spring-cloud-zookeeper-discovery</artifactId>
15.     </dependency>
16. </dependencies>

```

只要classpath中包含Spring Cloud Zookeeper, Apache Curator和Zookeeper Java客户端, 所有应用了 `@EnableDiscoveryClient` 注解的Spring Boot应用将尝试连接Zookeeper服务 `http://localhost:2181` (`zookeeper.connectString` 默认值)

```

1. @Configuration
2. @EnableAutoConfiguration
3. @EnableDiscoveryClient
4. @RestController
5. public class Application {
6.
7.     @RequestMapping("/")
8.     public String home() {
9.         return "Hello World";
10.    }
11.
12.    public static void main(String[] args) {
13.        SpringApplication.run(Application.class, args);
14.    }
15.
16. }

```

本地Zookeeper服务必须运行, 参见[Zookeeper](#)文档。

Spring Cloud for Amazon Web Services

- [Spring Cloud for Amazon Web Services](#)
 - [Quick Start](#)

Spring Cloud for Amazon Web Services

For full documentation visit [spring cloud for amazon web services](#).

Quick Start

Spring Cloud Connectors

- [Spring Cloud Connectors](#)
 - [Quick Start](#)

Spring Cloud Connectors

For full documentation visit [spring cloud connectors](#).

Quick Start

Spring Cloud CLI

- [Spring Cloud CLI](#)
 - [Quick Start](#)

Spring Cloud CLI

For full documentation visit [spring cloud cli](#).

Quick Start

Spring Cloud翻译文档

- [Spring Cloud Eureka](#)
- [声明式REST客户端Feign](#)
- [Spring Cloud Bus](#)
- [Spring Cloud Config](#)
- [Spring Cloud sleuth](#)
- [Spring Cloud stream](#)
- [Spring Cloud zuul](#)
- [Spring Cloud ribbon](#)

Spring Cloud Eureka

- 服务发现：Eureka客户端
 - [向Eureka注册服务](#)
 - [Eureka Server的身份验证](#)
 - [状态页和健康信息指示器](#)
 - [使用HTTPS](#)
 - [健康检查](#)
 - [Eureka元数据说明](#)
 - [使用EurekaClient对象](#)
 - [使用Spring的DiscoveryClient对象](#)
 - [为什么注册一个服务这么慢？](#)
- 服务发现：Eureka服务端
 - [高可用，Zone 和 Region](#)
 - [Standalone模式](#)
 - [“伙伴”感知](#)
 - [使用IP地址](#)

译者：王鸿飞 / brucewhf@gmail.com

Eureka学习文档资料：

- [Netflix Eureka详细文档](#)
- [Spring Cloud中对Eureka的介绍](#)

Spring Cloud Netflix提供了对Netflix开源项目的集成，使得我们可以以Spring Boot编程风格使用Netflix旗下相关框架。你只需要在程序中添加注解，就能使用成熟的Netflix组件来快速实现分布式系统的常见架构模式。这些模式包括服务发现(Eureka)，断路器(Hystrix)，智能路由(Zuul)和客户端负载均衡(Ribbon)。

服务发现：Eureka客户端

服务发现是微服务架构中的一项核心服务。如果没有该服务，我们就只能为每一个服务调用者手工配置可用服务的地址，这不仅繁琐而且非常容易出错。Eureka包括了服务端和客户端两部分。服务端可以做到高可用集群部署，每一个节点可以自动同步，有相同的服务注册信息。

向Eureka注册服务

当客户端向Eureka注册自己时会提供一些元信息，如主机名、端口号、获取健康信息的url和主页等。Eureka通过心跳连接判断服务是否在线，如果心跳检测失败超过指定时间，对应的服务通常就会被移出可用服务列表。

译者注：向Eureka Server注册过的服务会每30秒向Server发送一次心跳连接，Server会根据心跳数据更新该服务的健康状态并复制到其他Server中。如果超过90秒没有收到该服务的心跳数据，则Server会将该服务移出列表。参考文档：<https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>

Eureka Client代码示例：

```

1. @Configuration
2. @ComponentScan
3. @EnableAutoConfiguration
4. @EnableEurekaClient
5. @RestController
6. public class Application {
7.
8.     @RequestMapping("/")
9.     public String home() {
10.         return "Hello world";
11.     }
12.
13.     public static void main(String[] args) {
14.         new SpringApplicationBuilder(Application.class).web(true).run(args);
15.     }
16.
17. }
```

(其实就是个普通的Spring Boot应用)。在这个例子中我们显式的使用了 `@EnableEurekaClient` 注解，如果你只添加了Eureka相关依赖(即依赖中没有 `@EnableEurekaClient` 的定义)，可以使用 `@EnableDiscoveryClient` 注解达到同样的效果。除此之外你必须指定一下Eureka服务器的地址：

application.yml

```

1. eureka:
2.   client:
3.     serviceUrl:
4.       defaultZone: http://localhost:8761/eureka/
```

其中，`defaultZone` 的作用是给没有指定 `Zone` 的客户端一个默认的Eureka地址。

译者注：客户端可以在配置文件中指定当前服务属于哪一个 `Zone`，如果没有指定，则属于默认 `Zone`。

默认的应用名(Service ID)、主机名和端口号分别对应配置信息中

的 `${spring.application.name}`、`${spring.application.name}` 和 `${server.port}` 参数。

使用 `@EnableEurekaClient` 注解后当前应用会同时变成一个Eureka服务端实例(它会注册自身)和Eureka客户端(可以查询当前服务列表)，与此相关的配置都在以 `eureka.instance.*` 开头的参数下。只要你指定了 `spring.application.name` 参数，那么就可以放心的使用默认参数而不需要修改任何配置。

要查看更详细的参数，请参阅[EurekaInstanceConfigBean](#)和[EurekaClientConfigBean](#)。

Eureka Server的身份验证

如果客户端的 `eureka.client.serviceUrl.defaultZone` 参数值(即Eureka Server的地址)中包含 `HTTP Basic Authentication` 信息，如 `[http://user:password@localhost:8761/eureka]` (`http://user:password@localhost:8761/eureka`)，那么客户端就会自动使用该用户名、密码信息与Eureka服务端进行验证。如果你需要更复杂的验证逻辑，你必须注册一个 `DiscoveryClientOptionalArgs` 组件，并将 `ClientFilter` 组件注入，在这里定义的逻辑会在每次客户端向服务端发起请求时执行。

由于Eureka的限制，Eureka不支持单节点身份验证。

状态页和健康信息指示器

Eureka应用的状态页和健康信息默认的url为 `/info` 和 `/health`，这与 `Spring Boot Actuator` 中对应的Endpoint是重复的，因此你必须进行修改：

```
1. eureka:
2.   instance:
3.     statusPageUrlPath: ${management.context-path}/info
4.     healthCheckUrlPath: ${management.context-path}/health
```

客户端通过这些URL获取数据，并根据这些数据来判断是否可以向某个服务发起请求。

使用HTTPS

你可以指定 `EurekaInstanceConfig` 类中的 `eureka.instance.[nonSecurePortEnabled,securePortEnabled]=[false,true]` 属性来指定是否使用HTTPS。当配置使用HTTPS时，Eureka Server会返回以 `https` 开头的服务地址。

即使配置了使用HTTPS，Eureka的主页依然是以普通 HTTP 方式访问的。你需要手动添加一些配置来将这些页面也通过HTTPS保护起来：

```
1. eureka:
2.   instance:
3.     statusPageUrl: https://${eureka.hostname}/info
4.     healthCheckUrl: https://${eureka.hostname}/health
5.     homePageUrl: https://${eureka.hostname}/
```

注意，`eureka,hostname` 是Eureka原生属性，只有新版本的Eureka才支持该属性。你也可以用Spring EL表达式代

替: `${eureka.instance.hostName}`

如果你的应用前端部署了代理, 并且SSL的终点是此代理服务器, 那么你就需要在应用中解析 `forwarded` 请求头。如果你在配置文件中添加了 `X-Forwarded-*` 相关参数, *Spring Boot*中的嵌入式*Tomcat*会自动解析该请求头。一种表明你没有处理好 `forwarded` 请求头的迹象就是你的应用渲染出的HTML页面中链接显示的是错误的主机名和端口号。

健康检查

默认情况下, Eureka通过客户端发来的心跳包来判断客户端是否在线。如果你不显式指定, 客户端在心跳包中不会包含当前应用的健康数据(由*Spring Boot Actuator*提供)。这意味着只要客户端启动时完成了服务注册, 那么该客户端在主动注销之前在Eureka中的状态会永远是 `UP` 状态。我们可以通过配置修改这一默认行为, 即在客户端发送心跳包时会带上自己的健康信息。这样做的后果是只有当该服务的状态是 `UP` 时才能被访问, 其它的任何状态都会导致该服务不能被调用。

```
1. eureka:
2.   client:
3.     healthcheck:
4.       enabled: true
```

如果你想对健康检查有更细粒度的控制, 你可以自己实现 `com.netflix.appinfo.HealthCheckHandler` 接口。

以下内容翻译自Eureka官方手册:

Eureka客户端会每隔30s向服务端发送心跳包以告知服务端当前客户端没有挂掉。对于Client来说, 服务Server超过90s没有收到该Client的心跳数据, Server就会把该Client移出服务列表。最好不要修改30s的默认心跳间隔, 因为Server会使用这个时间数值来判断是否出现了大面积故障。(译者: 意思是比如Eureka默认2分钟收不到心跳就认为网络出了故障, 你如果把这个心跳间隔改成了3分钟, 那就出问题了。)

Eureka元数据说明

我们有必要花一些时间来了解一下Eureka的元数据, 这样就可以添加一些自定义的数据以适应特定的业务场景。像主机名、IP地址、端口号、状态页url和健康检查url都是Eureka定义的标准元数据。这些元数据会被保存在Eureka Server的注册信息中, 客户端会读取这些数据来向需要调用的服务直接发起连接。你可以使用以 `eureka.instance.metadataMap` 开头的参数来添加你自定义的元数据, 所有客户端都会读取到该信息。通过这种方式你能给客户端自定义一些行为。

使用EurekaClient对象

当添加了 `@EnableDiscoveryClient` 或 `@EnableEurekaClient` 注解后, 你就可以在应用中使用 `EurekaClient` 对象来获取服务列表:

```
1. @Autowired
```

```

2. private EurekaClient discoveryClient;
3.
4. public String serviceUrl() {
5.     InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);
6.     return instance.getHomePageUrl();
7. }

```

不要在 `@PostConstruct` 或 `@Scheduled` 方法中使用 `EurekaClient` 。在 `ApplicationContext` 还没有完全启动时使用该对象会发生错误。

使用Spring的DiscoveryClient对象

你没有必要直接使用Netflix原生的 `EurekaClient` 对象，在此基础上做一些封装使用起来会更方便。Spring Cloud支持 `Feign` 和 `Spring RestTemplate` ，它们都可以使用服务的逻辑名而不是URL地址来查询服务。如果想给 `Ribbon` 手工指定服务列表，你可以将 `<client>.ribbon.listOfServers` 属性设为逗号分隔的物理地址或主机名，参数中的 `client` 是服务id，即服务名。

你可以使用Spring提供的 `DiscoveryClient` 对象从而代码不会与Eureka紧耦合：

```

1. @Autowired
2. private DiscoveryClient discoveryClient;
3.
4. public String serviceUrl() {
5.     List<ServiceInstance> list = discoveryClient.getInstances("STORES");
6.     if (list != null && list.size() > 0 ) {
7.         return list.get(0).getUri();
8.     }
9.     return null;
10. }

```

为什么注册一个服务这么慢？

服务的注册涉及到心跳连接，默认为每30秒一次。只有当Eureka服务端和客户端本地缓存中的服务元数据相同时这个服务才能被其它客户端发现，这需要3个心跳周期。你可以通过参数 `eureka.instance.leaseRenewalIntervalInSeconds` 调整这个时间间隔来加快这个过程。在生产环境中你最好使用默认值，因为Eureka内部的某些计算依赖于该时间间隔。

服务发现：Eureka服务端

添加 `spring-cloud-starter-eureka-server` ，主类代码示例如下：

```

1. @SpringBootApplication
2. @EnableEurekaServer
3. public class Application {
4.
5.     public static void main(String[] args) {
6.         new SpringApplicationBuilder(Application.class).web(true).run(args);
7.     }
8.
9. }

```

服务启动后，Eureka有一个带UI的主页，注册信息可以通过 `/eureka/*` 下的URL获取到。

高可用，Zone 和 Region

Eureka把所有注册信息都放在内存中，所有注册过的客户端都会向Eureka发送心跳包来保持连接。客户端会有一份本地注册信息的缓存，这样就不需要每次远程调用时都向Eureka查询注册信息。

默认情况下，Eureka服务端自身也是个客户端，所以需要指定一个Eureka Server的URL作为“伙伴”(peer)。如果你没有提供这个地址，Eureka Server也能正常启动工作，但是在日志中会有大量关于找不到peer的错误信息。

Standalone模式

只要Eureka Server进程不会挂掉，这种集Server和Client于一身和心跳包的模式能让 Standalone(单台)部署的Eureka Server非常容易进行灾难恢复。在 Standalone 模式中，可以通过下面的配置来关闭查找“伙伴”的行为：

```

1. server:
2.     port: 8761
3.
4. eureka:
5.     instance:
6.         hostname: localhost
7.     client:
8.         registerWithEureka: false
9.         fetchRegistry: false
10.    serviceUrl:
11.        defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

```

注意，`serviceUrl` 中的地址的主机名要与本地主机名相同。

“伙伴”感知

Eureka Server可以通过运行多个实例并相互指定为“伙伴”的方式来达到更高的高可用性。实际上这就是默认设置，你只需要指定“伙伴”的地址就可以了：

```
1. ---
2. spring:
3.   profiles: peer1
4. eureka:
5.   instance:
6.     hostname: peer1
7.   client:
8.     serviceUrl:
9.       defaultZone: http://peer2/eureka/
10.
11. ---
12. spring:
13.   profiles: peer2
14. eureka:
15.   instance:
16.     hostname: peer2
17.   client:
18.     serviceUrl:
19.       defaultZone: http://peer1/eureka/
```

在上面这个例子中，我们通过使用不同 `profile` 配置的方式可以在本地运行两个Eureka Server。你可以通过修改 `/etc/host` 文件，使用上述配置在本地测试伙伴感特性。

你可以同时启动多个Eureka Server，并通过伙伴配置使之围成一圈(相邻两个Server互为伙伴)，这些Server中的注册信息都是同步的。If the peers are physically separated (inside a data centre or between multiple data centres) then the system can in principle survive split-brain type failures.

使用IP地址

有些时候你可能更倾向于直接使用IP地址定义服务而不是使用主机名。

把 `eureka.instance.preferIpAddress` 参数设为 `true` 时，客户端在注册时就会使用自己的ip地址而不是主机名。

声明式REST客户端Feign

- 声明式REST客户端：Feign
 - 覆盖Feign的默认配置
 - Feign对Hystrix的支持
 - Feign对Hystrix Fallback的支持
 - Feign对继承的支持
 - Feign对压缩的支持
 - Feign的日志

译者：王鸿飞 / brucewhf@gmail.com

声明式REST客户端：Feign

Feign是一个声明式Web Service客户端。使用Feign能让编写Web Service客户端更加简单，它的使用方法是定义一个接口，然后在上面添加注解，同时也支持 `JAX-RS` 标准的注解。Feign也支持可拔插式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和 `HttpMessageConverters`。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

主类示例：

```
1. @Configuration
2. @ComponentScan
3. @EnableAutoConfiguration
4. @EnableEurekaClient
5. @EnableFeignClients
6. public class Application {
7.
8.     public static void main(String[] args) {
9.         SpringApplication.run(Application.class, args);
10.    }
11.
12. }
```

`StoreClient.java` :

```
1. @FeignClient("stores")
2. public interface StoreClient {
3.     @RequestMapping(method = RequestMethod.GET, value = "/stores")
4.     List<Store> getStores();
5.
6.     @RequestMapping(method = RequestMethod.POST, value = "/stores/{storeId}", consumes =
    "application/json")
```

```

7.     Store update(@PathVariable("storeId") Long storeId, Store store);
8. }

```

`@FeignClient` 注解中的 `stores` 属性可以是一个任意字符串(译者注: 如果与 *Eureka* 组合使用, 则 `stores` 应为 *Eureka* 中的服务名), Feign用它来创建一个Ribbon负载均衡器。你也可以通过 `url` 属性来指定一个地址(可以是完整的URL, 也可以是一个主机名)。标注了 `@FeignClient` 注解的接口在 `ApplicationContext` 中的Bean实例名是这个接口的全限定名, 同时这个Bean还有一个别名, 为Bean名 + `FeignClient`。在本例中, 可以使用 `@Qualifier("storesFeignClient")` 来注入该组件。

如果classpath中有Ribbon, 上面的例子中Ribbon Client会想办法查找 `stores` 服务的IP地址。如果Eureka也在classpath中, 那么Ribbon会从Eureka的注册信息中查找。如果你不想用Eureka, 你也可以在配置文件中直接指定一组服务器地址。

覆盖Feign的默认配置

Spring Cloud对Feign的封装中一个核心的概念就是客户端要有一个名字。每一个客户端随时可以向远程服务发起请求, 并且每个服务都可以像使用 `@FeignClient` 注解一样指定一个名字。Spring Cloud会将所有的 `@FeignClient` 组合在一起创建一个新的 `ApplicationContext`, 并使用 `FeignClientsConfiguration` 对Clients进行配置。配置中包括编码器、解码器和一个 `feign.Contract`。

Spring Cloud允许你通过 `configuration` 属性完全控制Feign的配置信息, 这些配置比 `FeignClientsConfiguration` 优先级要高:

```

1. @FeignClient(name = "stores", configuration = FooConfiguration.class)
2. public interface StoreClient {
3.     //...
4. }

```

在这个例子中, `FooConfiguration` 中的配置信息会覆盖掉 `FeignClientsConfiguration` 中对应的配置。

注意: `FooConfiguration` 虽然是个配置类, 但是它不应该被主上下文(`ApplicationContext`)扫描到, 否则该类中的配置信息就会被应用于所有的 `@FeignClient` 客户端(本例中 `FooConfiguration` 中的配置应该只对 `StoreClient` 起作用)。

注意: `serviceId` 属性已经被弃用了, 取而代之的是 `name` 属性。

在先前的版本中指定了 `url` 属性时 `name` 是可选属性，现在无论什么时候 `name` 都是必填属性。

`name` 和 `url` 属性也支持占位符：

```
1. @FeignClient(name = "${feign.name}", url = "${feign.url}")
2. public interface StoreClient {
3.     //...
4. }
```

Spring Cloud Netflix为Feign提供了以下默认的配置Bean：（下面最左侧是Bean的类型，中间是Bean的name，右侧是类名）

- `Decoder` `feignDecoder`: `ResponseEntityDecoder`（这是对 `SpringDecoder` 的封装）
- `Encoder` `feignEncoder`: `SpringEncoder`
- `Logger` `feignLogger`: `Slf4jLogger`
- `Contract` `feignContract`: `SpringMvcContract`
- `Feign.Builder` `feignBuilder`: `HystrixFeign.Builder`

下列Bean默认情况下Spring Cloud Netflix并没有提供，但是在应用启动时依然会从上下文中查找这些Bean来构造客户端对象：

- `Logger.Level`
- `Retryer`
- `ErrorDecoder`
- `Request.Options`
- `Collection<RequestInterceptor>`

如果想要覆盖Spring Cloud Netflix提供的默认配置Bean，需要

在 `@FeignClient` 的 `configuration` 属性中指定一个配置类，并提供想要覆盖的Bean即可：

```
1. @Configuration
2. public class FooConfiguration {
3.     @Bean
4.     public Contract feignContract() {
5.         return new feign.Contract.Default();
6.     }
7.
8.     @Bean
9.     public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
10.        return new BasicAuthRequestInterceptor("user", "password");
11.    }
12. }
```

本例子中，我们用 `feign.Contract.Default` 代替了 `SpringMvcContract`，并添加了一个 `RequestInterceptor`。以这种方式做的配置会在所有的 `@FeignClient` 中生效。

Feign对Hystrix的支持

如果Hystrix在classpath中，Feign会默认将所有方法都封装到断路器中。Returning a `com.netflix.hystrix.HystrixCommand` is also available。这样一来你就可以使用Reactive Pattern了。（调用 `.toObservable()` 或 `.observe()` 方法，或者通过 `.queue()` 进行异步调用）。

将 `feign.hystrix.enabled=false` 参数设为 `false` 可以关闭对Hystrix的支持。

如果想只关闭指定客户端的Hystrix支持，创建一个 `Feign.Builder` 组件并标注为 `@Scope(prototype)`：

```
1. @Configuration
2. public class FooConfiguration {
3.     @Bean
4.     @Scope("prototype")
5.     public Feign.Builder feignBuilder() {
6.         return Feign.builder();
7.     }
8. }
```

Feign对Hystrix Fallback的支持

Hystrix支持 `fallback` 的概念，即当断路器打开或发生错误时执行指定的失败逻辑。要为指定的 `@FeignClient` 启用Fallback支持，需要在 `fallback` 属性中指定实现类：

```
1. @FeignClient(name = "hello", fallback = HystrixClientFallback.class)
2. protected interface HystrixClient {
3.     @RequestMapping(method = RequestMethod.GET, value = "/hello")
4.     Hello iFailSometimes();
5. }
6.
7. static class HystrixClientFallback implements HystrixClient {
8.     @Override
9.     public Hello iFailSometimes() {
10.         return new Hello("fallback");
11.     }
12. }
```

注意：Feign对Hystrix Fallback的支持有一个限制：对于返

回 `com.netflix.hystrix.HystrixCommand` 或 `rx.Observable` 对象的方法，`fallback`不起作用。

Feign对继承的支持

Feign可以通过Java的接口支持继承。你可以把一些公共的操作放到父接口中，然后定义子接口继承之：

UserService.java

```
1. public interface UserService {
2.
3.     @RequestMapping(method = RequestMethod.GET, value = "/users/{id}")
4.     User getUser(@PathVariable("id") long id);
5. }
```

UserResource.java

```
1. @RestController
2. public class UserResource implements UserService {
3.
4. }
```

UserClient.java

```
1. package project.user;
2.
3. @FeignClient("users")
4. public interface UserClient extends UserService {
5.
6. }
```

注意：在服务的调用端和提供端共用同一个接口定义是不明智的，这会将调用端和提供端的代码紧紧耦合在一起。同时在SpringMVC中会有问题，因为请求参数映射是不能被继承的。

Feign对压缩的支持

你可能会想要对请求/响应数据进行Gzip压缩，指定以下参数即可：

```
1. feign.compression.request.enabled=true
2. feign.compression.response.enabled=true
```

也可以添加一些更细粒度的配置：

```
1. feign.compression.request.enabled=true
2. feign.compression.request.mime-types=text/xml,application/xml,application/json
3. feign.compression.request.min-request-size=2048
```

上面的3个参数可以让你选择对哪种请求进行压缩，并设置一个最小请求大小的阈值。

Feign的日志

每一个 `@FeignClient` 都会创建一个 `Logger`，`Logger` 的名字就是接口的全限定名。Feign的日志配置参数仅支持 `DEBUG`：

application.properties

```
1. logging.level.project.user.UserClient: DEBUG
```

`Logger.Level` 对象允许你为指定客户端配置想记录哪些信息：

- `NONE`，不记录任何信息，默认值。
- `BASIC`，记录请求方法、请求URL、状态码和用时。
- `HEADERS`，在 `BASIC` 的基础上再记录一些常用信息。
- `FULL`：记录请求和响应报文的全部内容。

将 `Level` 设置为 `FULL` 的示例如下：

```
1. @Configuration
2. public class FooConfiguration {
3.     @Bean
4.     Logger.Level feignLoggerLevel() {
5.         return Logger.Level.FULL;
6.     }
7. }
```

Spring Cloud Bus

- [Spring Cloud Bus](#)
 - [Quick Start](#)
 - [Addressing an Instance](#)
 - [Addressing all instances of a service](#)
 - [Application Context ID must be unique](#)
 - [Customizing the Message Broker](#)
 - [Tracing Bus Events](#)
 - [Broadcasting Your Own Events](#)

Spring Cloud Bus

Spring Cloud Bus 通过一个轻量级消息代理连接分布式系统的节点。这可以用于广播状态更改（如配置更改）或其他管理指令。当前唯一的实现方式是通过一个AMQP代理作为消息传输，但相同的基本特征（传输上的一些依赖）是其他传输的路线图

Quick Start

如果Spring Cloud Bus在类路径里它将通过Spring Boot autconfiguration启动。所有你需要做的就是使 `spring-cloud-starter-bus-amqp` 或 `spring-cloud-starter-bus-kafka` 添加到你的依赖管理中，Spring Cloud将完成剩下的事。确定broker (RabbitMQ or Kafka) 可用：如果在本地运行你不需要做任何事，但如果在远程运行需要配置broker信息，如Rabbit

application.yml

```
1. spring:
2.   rabbitmq:
3.     host: mybroker.com
4.     port: 5672
5.     username: user
6.     password: secret
```

bus目前支持向所有的监听节点发送消息，或向所有特定服务节点（注册到Eureka中的）发送消息。更多的选择标准可能在未来增加（ie. only service X nodes in data center Y, etc...）。在 `/bus/*` 命名空间下有一些http的endpoints，目前有两个实现，第一个 `/bus/env`，发送键/值对来更新每个节点Spring Environment，第二个 `/bus/refresh`，重载每个应用程序的配置，就好像他们都被发现通过 `/refresh` endpoint。

NOTE

*Bus*的starters涵盖Rabbit和Kafka，因为这两个比较常见，另一方面Spring Cloud Stream十分灵活，binder将结合spring-cloud-bus进行工作。

Addressing an Instance

HTTP endpoints 接受 "destination" 参数, 如: `/bus/refresh?destination=customers:9000` , "destination" 参数是 `ApplicationContext` 的ID, 如果一个Bus实例拥有一个ID, 然后它将处理消息, 并且所有其他实例将忽略它。Spring Boot 在 `ContextIdApplicationContextInitializer` 设置这个ID, ID的组成默认是由 `spring.application.name` , `active profiles` , `server.port`

Addressing all instances of a service

`destination`参数采用Spring PathMatcher (分隔符 `:`) 确认某个实例是否将处理消息, 使用上面的例子`/bus/refresh?destination=customers:*` 将对象的所有实例的“customers”服务的配置和端口设置为ApplicationContext的ID

Application Context ID must be unique

bus 试图消除处理一个事件两次, 一次从原来的applicationevent消除, 一次从队列消除。要做到这一点, 它检查发送应中的用程序上下文ID和当前应用程序上下文ID, 如果多个服务实例具有相同的应用程序上下文ID, 事件将不被处理。在本地机器上运行, 每个服务将在一个不同的端口, 这将是应用程序上下文ID的一部分。云计算提供了一个索引来区分。Cloud Foundry提供了一个索引来区分。确保应用程序上下文ID是唯一的, 对每一个服务实例设置 `spring.application.index` 唯一。例如, 在框架配置`application.properties` (或`bootstrap.properties`如果使用`configserver`) 中设置 `spring.application.index=${INSTANCE_INDEX}`

Customizing the Message Broker

Spring Cloud Bus 通过 Spring Cloud Stream 广播消息从而获得信息的流动, 你只需要在类路径中选择你绑定的实现。bus有方便的starters, AMQP (RabbitMQ) 和 Kafka (`spring-cloud-starter-bus-[amqp,kafka]`)。一般来说, Spring Cloud Stream依靠Spring Boot autoconfiguration规则配置中间件, 所以例如AMQP代理地址是可以通过 `spring.rabbitmq.*` 属性配置改变的。Spring Cloud Bus有少数本地配置 `spring.cloud.bus.*` (例如 `spring.cloud.bus.destination` 是使用的体中间件的主题的名称)。通常情况下, 默认值就足够了。

Tracing Bus Events

Bus events (RemoteApplicationEvent的子类) 可以通过设置

`spring.cloud.bus.trace.enabled=true` 追踪, 如果你这样做的话, Spring Boot

TraceRepository (如果存在) 将显示每个事件发送和每个服务实例所有的acks。例如(/trace endpoint):

```

1. {
2.   "timestamp": "2015-11-26T10:24:44.411+0000",
3.   "info": {
4.     "signal": "spring.cloud.bus.ack",
5.     "type": "RefreshRemoteApplicationEvent",
6.     "id": "c4d374b7-58ea-4928-a312-31984def293b",
7.     "origin": "stores:8081",
8.     "destination": "*:*"
9.   }
10. },
11. {
12.   "timestamp": "2015-11-26T10:24:41.864+0000",
13.   "info": {
14.     "signal": "spring.cloud.bus.sent",
15.     "type": "RefreshRemoteApplicationEvent",
16.     "id": "c4d374b7-58ea-4928-a312-31984def293b",
17.     "origin": "customers:9000",
18.     "destination": "*:*"
19.   }
20. },
21. {
22.   "timestamp": "2015-11-26T10:24:41.862+0000",
23.   "info": {
24.     "signal": "spring.cloud.bus.ack",
25.     "type": "RefreshRemoteApplicationEvent",
26.     "id": "c4d374b7-58ea-4928-a312-31984def293b",
27.     "origin": "customers:9000",
28.     "destination": "*:*"
29.   }
30. }

```

这个trace显示了来自 `customers:9000` 的 `RefreshRemoteApplicationEvent`, 广播到所有的服务, 通过 `customers:9000` 和 `stores:8081` 收到 (acked)

处理自己的ACK信号你可以为AckRemoteApplicationEvent添加@EventListener, 在添加SentApplicationEvent类型, 或者你可以进入TraceRepository, 从那里开采数据。

NOTE

所有Bus应用都可以追踪acks, 在做复杂查询的数据中央服务中这么做是有用的, 或将其转发到一个专门的跟踪服务。

Broadcasting Your Own Events

Bus可以支持任何RemoteApplicationEvent的事件类型，但默认传输JSON和反序列化器需要提前知道哪些类型将被使用。定义一个新类型需要在org.springframework.cloud.bus.event的子目录下，可以使用@JsonTypeName注解在你的定制类上或依赖于默认的策略，使用类的简单的名字。请注意，生产者和消费者都需要访问类定义。

Spring Cloud Config

- [Spring Cloud Config](#)
 - [Quick Start](#)
 - [Client Side Usage](#)
- [Spring Cloud Config Server](#)
 - [Environment Repository](#)
 - [Health Indicator](#)
 - [Security](#)
 - [Encryption and Decryption](#)
 - [Key Management](#)
 - [Creating a Key Store for Testing](#)
 - [Using Multiple Keys and Key Rotation](#)
 - [Serving Encrypted Properties](#)
- [Serving Alternative Formats](#)
- [Serving Plain Text](#)
- [Embedding the Config Server](#)
- [Push Notifications and Spring Cloud Bus](#)
- [Spring Cloud Config Client](#)
 - [Config First Bootstrap](#)
 - [Discovery First Bootstrap](#)
 - [Config Client Fail Fast](#)
 - [Config Client Retry](#)
 - [Locating Remote Configuration Resources](#)
 - [Security](#)

Spring Cloud Config

Quick Start

Client Side Usage

Spring Cloud Config Server

Environment Repository

Health Indicator

Security

Encryption and Decryption

Key Management

Creating a Key Store for Testing

Using Multiple Keys and Key Rotation

Serving Encrypted Properties

Serving Alternative Formats

Serving Plain Text

Embedding the Config Server

Push Notifications and Spring Cloud Bus

许多源代码库提供者（如GitHub，gitlab或bitbucket）会通过webhook通知您在代码库的变化。你可以通过供应商的用户界面配置你的webhook，一个URL和一系列你感兴趣的事件。比如Github会通过POST请求webhook，body中包含提交的一些信息，请求头“X-Github-Event”等于“push”。如果你添加了依赖 `spring-cloud-config-monitor` 并在你的Config Server中激活了Spring Cloud Bus，一个“/monitor” endpoint会被启用。

当webhook被触发，Config Server认为发生了改变，并将针对应用程序发送一个 `RefreshRemoteApplicationEvent`。变化检测可以考虑，但默认情况下它只是在寻找application name匹配的文件的改变（例如“foo.properties”是针对“foo”应用，“application.properties”是针对所有应用）。该策略，如果你想重写的行为是 `PropertyPathNotificationExtractor`，这个类接受请求头和请求体作为参数，返回一个路径改变的列表。

默认配置基于Github，Gitlab或Bitbucket。除了来自Github，Gitlab或Bitbucket的JSON通知，您可以触发一个更改通知，向/monitor发起POST请求，参数是 `path={name}`，这将广播匹配“{name}”的应用（可以包含通配符）

NOTE

当 `spring-cloud-bus` 在 *Config Server* 的服务端和客户端被激活时，`RefreshRemoteApplicationEvent` 才会被传播

NOTE

默认的配置也检测到本地的 *Git* 仓库的文件系统的变化（这种情况下不适用 *webhook*，只要你编辑一个配置文件的更新就会被发送）

Spring Cloud Config Client

Config First Bootstrap

Discovery First Bootstrap

Config Client Fail Fast

Config Client Retry

Locating Remote Configuration Resources

Security

Spring Cloud sleuth

- Spring Cloud Sleuth
 - 术语
 - 目的
 - 基于Zipkin的分布式追踪
 - Log相关
 - JSON Logback with Logstash
 - 添加进工程
 - 仅Sleuth(log收集)
 - 通过HTTP使用基于Zipkin的Sleuth
 - 通过SpringCloudStream使用Sleuth+Zipkin
 - Spring Cloud Sleuth Stream Zipkin Collector
 - 特性
 - 抽样
 - Instrumentation
 - Span生命周期
 - Creating and closing spans
 - Continuing spans
 - Creating spans with an explicit parent
 - 命名spans
 - @SpanName注解
 - toString()方法
 - 定制化
 - Spring Integration
 - HTTP
 - Example
 - Custom SA tag in Zipkin
 - Spring Data as Messages
 - Zipkin Consumer
 - Custom Consumer
 - 度量
 - Integrations
 - Runnable and Callable
 - Hystrix
 - RxJava
 - HTTP integration
 - HTTP client integration
 - Feign
 - 异步通信
 - Messaging

■ Zuul

译者：楠倏之语 / xc8609@126.com

原文地址：[Spring Cloud Sleuth使用简介](#)

绝大部分出自原文，部分做了微调。

Spring Cloud Sleuth

SpringCloudSleuth提供了分布式追踪的解决方案。

术语

SpringCloudSleuth 借用了 Dapper 的术语

- **Span:** 基本工作单元，例如，在一个新建的span中发送一个RPC等同于发送一个回应请求给RPC，span通过一个64位ID唯一标识，trace以另一个64位ID表示，span还有其他数据信息，比如摘要、时间戳事件、关键值注释(tags)、span的ID、以及进度ID(通常是IP地址) span在不断的启动和停止，同时记录了时间信息，当你创建了一个span，你必须在未来的某个时刻停止它。

`root span` 的SpanID和TraneID相等。

- **Trace:** 一系列spans组成的一个树状结构，例如，如果你正在跑一个分布式大数据工程，你可能需要创建一个trace。
- **Annotation:** 用来及时记录一个事件的存在，一些核心annotations用来定义一个请求的开始和结束
 - cs: Client Sent - 客户端发起一个请求，这个anotion描述了这个span的开始
 - sr: Server Received - 服务端获得请求并准备开始处理它，如果将其sr减去cs时间戳便可得到网络延迟
 - ss: Server Sent - 注解表明请求处理的完成(当请求返回客户端)，如果ss减去sr时间戳便可得到服务端需要的处理请求时间
 - cr: Client Received - 表明span的结束，客户端成功接收到服务端的回复，如果cr减去cs时间戳便可得到客户端从服务端获取回复的所有所需时间

将Span和Trace在一个系统中使用Zipkin注解的过程图形化：



每个颜色的注解表明一个span(总计7个spans，从A到G)，如果在注解中有这样的信息：

1. **Trace Id** = X
2. **Span Id** = D

3. Client Sent

这就表明当前span将**Trace-Id**设置为X，将**Span-Id**设置为D，同时它还表明了**Client Sent**事件。

spans 的parent/child关系图形化：



目的

下面章节的例子是基于上图描述的场景

基于Zipkin的分布式追踪

总计10个spans，如果在Zipkin中查看traces将看到如下图：



但如果你选取一个特殊的trace你将看到7个spans：



当选取一个特殊*trace*时你会看到合并的*spans*，这意味着如果有两个*spans*使用客户端接收发送/服务端接收发送注解发送至*Zipkin*时，他们将表现为一个单独的*span*

在展示Span和Trace图形化的图片中有20个颜色标签，Zipkin又是如何接收10个spans的呢？

- 2个span A标签表明span的开始和结束，接近结束时一个单独的span发送给Zipkin
- 4个span B标签实际上是一个有4个注解的单独span，然而这个span是由两个分离的实例组成的，一个由 service 1发出，一个由service 2发出，因此实际上两个span实例是发送到Zipkin并在那合并
- 2个span C标签表明span的开始和结束，接近结束时一个单独的span发送给Zipkin
- 4个span D标签实际上是一个有4个注解的单独span，然而这个span是由两个分离的实例组成的，一个由 service 2发出，一个由service 3发出，因此实际上两个span实例是发送到Zipkin并在那合并
- 2个span E标签表明span的开始和结束，接近结束时一个单独的span发送给Zipkin
- 4个span F标签实际上是一个有4个注解的单独span，然而这个span是由两个分离的实例组成的，一个由 service 2发出，一个由service 4发出，因此实际上两个span实例是发送到Zipkin并在那合并
- 2个span G标签表明span的开始和结束，接近结束时一个单独的span发送给Zipkin

因此1个span来自A, 2个span来自B, 1个span来自C, 2个span来自D, 1个span来自E, 2个span来自F, 1个来自G, 总计10个spans。

Zipkin中的依赖图:



Log相关

当使用trace id为 `2485ec27856c56f4` 抓取这四个应用的log时, 会获得如下输出:

```
1. service1.log:2016-02-26 11:15:47.561 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true]
68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Hello from service1.
Calling service2
2. service2.log:2016-02-26 11:15:47.710 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true]
68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Hello from service2.
Calling service3 and then service4
3. service3.log:2016-02-26 11:15:47.895 INFO [service3,2485ec27856c56f4,1210be13194bfe5,true]
68060 --- [nio-8083-exec-1] i.s.c.sleuth.docs.service3.Application : Hello from service3
4. service2.log:2016-02-26 11:15:47.924 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true]
68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from
service3 [Hello from service3]
5. service4.log:2016-02-26 11:15:48.134 INFO [service4,2485ec27856c56f4,1b1845262ffba49d,true]
68061 --- [nio-8084-exec-1] i.s.c.sleuth.docs.service4.Application : Hello from service4
6. service2.log:2016-02-26 11:15:48.156 INFO [service2,2485ec27856c56f4,9aa10ee6fbde75fa,true]
68059 --- [nio-8082-exec-1] i.s.c.sleuth.docs.service2.Application : Got response from
service4 [Hello from service4]
7. service1.log:2016-02-26 11:15:48.182 INFO [service1,2485ec27856c56f4,2485ec27856c56f4,true]
68058 --- [nio-8081-exec-1] i.s.c.sleuth.docs.service1.Application : Got response from
service2 [Hello from service2, response from service3 [Hello from service3] and from service4
[Hello from service4]]
```

如果你使用log集合工具例如Kibana、Splunk等, 你可以看到事件的发生信息, Kibana的例子如下:




以下是Logstash的Grok模式:

```
1. filter {
2.     # pattern matching logback pattern
3.     grok {
4.         match => { "message" => "%{TIMESTAMP_ISO8601:timestamp}\s+{%{LOGLEVEL:severity}}\s+\
[%{DATA:service},{DATA:trace},{DATA:span},{DATA:exportable}]\s+{%{DATA:pid}}---\s+[%\
{DATA:thread}]\s+{%{DATA:class}}\s+:\s+{%{GREEDYDATA:rest}}" }
5.     }
6. }
```

JSON Logback with Logstash

为了方便获取Logstash，通常保存log在JSON文件中而不是text文件中，配置方法如下：

依赖建立

- 确保Logback在classpath中(ch.qos.logback:logback-core)
- 增加LogstashLogback编码 - version 4.6的例子: net.logstash.logback  4.6

Logback建立

以下是一个Logback配置的例子(文件名称 `logback-spring.xml`)：

- 使用JSON格式记录应用信息到build/\${spring.application.name}.json文件
- 有两个添加注释源- console和标准log文件
- 与之前章节使用相同的log模式

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <configuration>
3.     <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
4.
5.     <springProperty scope="context" name="springAppName" source="spring.application.name"/>
6.     <!-- Example for logging into the build folder of your project -->
7.     <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>
8.
9.     <property name="CONSOLE_LOG_PATTERN"
10.         value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p})
11.             %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]){yellow}
12.             %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan}
13.             %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>
14.
15.     <!-- Appender to log to console -->
16.     <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
17.         <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
18.             <!-- Minimum logging level to be presented in the console logs -->
19.             <level>INFO</level>
20.         </filter>
21.         <encoder>
22.             <pattern>${CONSOLE_LOG_PATTERN}</pattern>
23.             <charset>utf8</charset>
24.         </encoder>
25.     </appender>
26.
27.     <!-- Appender to log to file -->
28.     <appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">
29.         <file>${LOG_FILE}</file>

```

```

27.         <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
28.             <fileNamePattern>${LOG_FILE}_%d{yyyy-MM-dd}.gz</fileNamePattern>
29.             <maxHistory>7</maxHistory>
30.         </rollingPolicy>
31.         <encoder>
32.             <pattern>${CONSOLE_LOG_PATTERN}</pattern>
33.             <charset>utf8</charset>
34.         </encoder>
35.     </appender>
36.
37.     <!-- Appender to log to file in a JSON format -->
38.     <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
39.         <file>${LOG_FILE}.json</file>
40.         <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
41.             <fileNamePattern>${LOG_FILE}.json_%d{yyyy-MM-dd}.gz</fileNamePattern>
42.             <maxHistory>7</maxHistory>
43.         </rollingPolicy>
44.         <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
45.             <providers>
46.                 <timestamp>
47.                     <timeZone>UTC</timeZone>
48.                 </timestamp>
49.                 <pattern>
50.                     <pattern>
51.                         {
52.                             "severity": "%level",
53.                             "service": "${springAppName:-}",
54.                             "trace": "%X{X-B3-TraceId:-}",
55.                             "span": "%X{X-B3-SpanId:-}",
56.                             "exportable": "%X{X-Span-Export:-}",
57.                             "pid": "${PID:-}",
58.                             "thread": "%thread",
59.                             "class": "%logger{40}",
60.                             "rest": "%message"
61.                         }
62.                     </pattern>
63.                 </pattern>
64.             </providers>
65.         </encoder>
66.     </appender>
67.
68.     <root level="INFO">
69.         <!--<appender-ref ref="console"/>-->
70.         <appender-ref ref="logstash"/>
71.         <!--<appender-ref ref="flatfile"/>-->
72.     </root>
73. </configuration>

```

使用 `logback-spring.xml` 需要将 `spring.application.name` 在 `bootstrap` 中配置，而不是配置在 `application` 中，否则logback客户端将不能读取到配置值。

添加进工程

仅Sleuth(log收集)

如果仅需要Spring Cloud Sleuth而不需要Zipkin集成，只需要增加 `spring-cloud-starter-sleuth` 模块到你工程中，maven方式如下

```

1. <dependencyManagement> (1)
2.     <dependencies>
3.         <dependency>
4.             <groupId>org.springframework.cloud</groupId>
5.             <artifactId>spring-cloud-dependencies</artifactId>
6.             <version>Brixton.RELEASE</version>
7.             <type>pom</type>
8.             <scope>import</scope>
9.         </dependency>
10.    </dependencies>
11. </dependencyManagement>
12.
13. <dependency> (2)
14.     <groupId>org.springframework.cloud</groupId>
15.     <artifactId>spring-cloud-starter-sleuth</artifactId>
16. </dependency>

```

1. 为了不手动添加版本号，更好的方式是通过Spring BOM添加dependencymanagement
2. 添加依赖到 `spring-cloud-starter-sleuth`

通过HTTP使用基于Zipkin的Sleuth

如果你需要Sleuth和Zipkin，只需要添加 `spring-cloud-starter-zipkin` 依赖

```

1. <dependencyManagement> (1)
2.     <dependencies>
3.         <dependency>
4.             <groupId>org.springframework.cloud</groupId>
5.             <artifactId>spring-cloud-dependencies</artifactId>
6.             <version>Brixton.RELEASE</version>
7.             <type>pom</type>
8.             <scope>import</scope>
9.         </dependency>

```

```

10.     </dependencies>
11. </dependencyManagement>
12.
13. <dependency> (2)
14.     <groupId>org.springframework.cloud</groupId>
15.     <artifactId>spring-cloud-starter-zipkin</artifactId>
16. </dependency>

```

1. 为了不手动添加版本号，更好的方式是通过Spring BOM添加dependencymanagement
2. 添加依赖到 `spring-cloud-starter-zipkin`

通过SpringCloudStream使用Sleuth+Zipkin

```

1. <dependencyManagement> (1)
2.     <dependencies>
3.         <dependency>
4.             <groupId>org.springframework.cloud</groupId>
5.             <artifactId>spring-cloud-dependencies</artifactId>
6.             <version>Brixton.RELEASE</version>
7.             <type>pom</type>
8.             <scope>import</scope>
9.         </dependency>
10.    </dependencies>
11. </dependencyManagement>
12.
13. <dependency> (2)
14.     <groupId>org.springframework.cloud</groupId>
15.     <artifactId>spring-cloud-sleuth-stream</artifactId>
16. </dependency>
17. <dependency> (3)
18.     <groupId>org.springframework.cloud</groupId>
19.     <artifactId>spring-cloud-starter-sleuth</artifactId>
20. </dependency>
21. <!-- EXAMPLE FOR RABBIT BINDING -->
22. <dependency> (4)
23.     <groupId>org.springframework.cloud</groupId>
24.     <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
25. </dependency>

```

1. 为了不手动添加版本号，更好的方式是通过Spring BOM添加dependencymanagement
2. 添加依赖到 `spring-cloud-sleuth-stream`
3. 添加依赖到 `spring-cloud-starter-sleuth`
4. 添加一个binder(e.g.Rabbit binder)来告诉Spring Cloud Stream应该绑定什么

Spring Cloud Sleuth Stream Zipkin Collector

启动一个Spring Cloud Sleuth Stream Zipkin收集器只需要添加 `spring-cloud-sleuth-zipkin-stream` 依赖

```

1. <dependencyManagement> (1)
2.   <dependencies>
3.     <dependency>
4.       <groupId>org.springframework.cloud</groupId>
5.       <artifactId>spring-cloud-dependencies</artifactId>
6.       <version>Brixton.RELEASE</version>
7.       <type>pom</type>
8.       <scope>import</scope>
9.     </dependency>
10.  </dependencies>
11. </dependencyManagement>
12.
13. <dependency> (2)
14.   <groupId>org.springframework.cloud</groupId>
15.   <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
16. </dependency>
17. <dependency> (3)
18.   <groupId>org.springframework.cloud</groupId>
19.   <artifactId>spring-cloud-starter-sleuth</artifactId>
20. </dependency>
21. <!-- EXAMPLE FOR RABBIT BINDING -->
22. <dependency> (4)
23.   <groupId>org.springframework.cloud</groupId>
24.   <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
25. </dependency>

```

1. 为了不手动添加版本号，更好的方式是通过Spring BOM添加dependencymanagement
2. 添加依赖到 `spring-cloud-sleuth-zipkin-stream`
3. 添加依赖到 `spring-cloud-starter-sleuth`
4. 添加一个binder (e.g. Rabbit binder) 来告诉Spring Cloud Stream应该绑定什么

之后只需要在你的主类中添加`@EnableZipkinStreamServer`注解

```

1. package example;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5. import org.springframework.cloud.sleuth.zipkin.stream.EnableZipkinStreamServer;
6.
7. @SpringBootApplication
8. @EnableZipkinStreamServer
9. public class ZipkinStreamServerApplication {
10.

```

```

11.     public static void main(String[] args) throws Exception {
12.         SpringApplication.run(ZipkinStreamServerApplication.class, args);
13.     }
14.
15. }

```

特性

- 添加trace和spanid到Slf4J MDC，然后就可以从一个给定的trace或span中提取所有的log，例如

```

1. 2016-02-02 15:30:57.902 INFO [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
2. 2016-02-02 15:30:58.372 ERROR [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 --- [nio-8081-exec-3] ...
3. 2016-02-02 15:31:01.936 INFO [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 --- [nio-8081-exec-4] ...

```

注意MDC中的 `[appname, traceId, spanId, exportable]` :

- - spanId - the id of a specific operation that took place
 - appname - the name of the application that logged the span
 - traceId - the id of the latency graph that contains the span
 - exportable - whether the log should be exported to Zipkin or not. When would you like the span not to be exportable? In the case in which you want to wrap some operation in a Span and have it written to the logs only.
- 在通常的分布式追踪数据模型上提供一种抽象模型：traces、spans(生成一个DAG)、annotations、key-value annotations。基于HTrace是较为宽松的，但Zipkin(Dapper)更具兼容性
- Sleuth记录时间信息来帮助延迟分析，使用Sleuth可以精确找到应用中延迟的原因，Sleuth不会log太多，因此不会导致你的应用挂掉
 - propagates structural data about your call-graph in-band, and the rest out-of-band
 - includes opinionated instrumentation of layers such as HTTP
 - includes sampling policy to manage volume
 - can report to a Zipkin system for query and visualization
- 使用Spring应用装备出入口点(servletfilter、async endpoints、rest template、scheduled actions、messagechannels、zuul filters、feign client)

- Sleuth包含默认逻辑通过http或messaging boundaries来加入一个trace, 例如, http传播通过Zipkin-compatible request headers工作, 这个传播逻辑定义和定制是通过SpanInjector和SpanExtractor
- 实现提供简单的接受或放弃span
- If spring-cloud-sleuth-zipkin then the app will generate and collect Zipkin-compatible traces. By default it sends them via HTTP to a Zipkin server on localhost (port 9411). Configure the location of the service using spring.zipkin.baseUrl.
- If spring-cloud-sleuth-stream then the app will generate and collect traces via Spring Cloud Stream. Your app automatically becomes a producer of tracer messages that are sent over your broker of choice (e.g. RabbitMQ, Apache Kafka, Redis).

IMPORTANT

If using Zipkin or Stream, configure the percentage of spans exported using `spring.sleuth.sampler.percentage` (default 0.1, i.e. 10%). Otherwise you might think that Sleuth is not working cause it's omitting some spans.

NOTE

the SLF4J MDC is always set and logback users will immediately see the trace and span ids in logs per the example above. Other logging systems have to configure their own formatter to get the same result. The default is `logging.pattern.level` set to `%clr(%5p)`
`%clr([${spring.application.name:},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}])`
`{yellow}` (this is a Spring Boot feature for logback users). This means that if you're not using SLF4J this pattern WILL NOT be automatically applied.

抽样

在分布式追踪时, 数据量可能会非常大, 因此抽样就变得非常重要(通常不需要导出所有的spans以得到事件发生原貌), Spring Cloud Sleuth有一个 `Sampler` 策略, 即用户可以控制抽样算法, Samplers不会停止正在生成的span id(相关的), 但他们会阻止tags和events附加和输出, 默认策略是当一个span处于活跃状态会继续trace, 但新的span会一直处于不输出状态, 如果所有应用都使用这个sampler, 你会在logs中看到traces, 但不会出现在任何远程仓库。测试状态资源都是充足的, 并且你只使用logs的话他就是你需要的全部(e.g. 一个ELK集合), 如果输出span数据到Zipkin或Spring Cloud Stream, 有 `AlwaysSampler` 输出所有数据和 `PercentageBasedSampler` 采样spans确定的一部分。

如果使用spring-cloud-sleuth-zipkin或spring-cloud-sleuth-stream, `PercentageBasedSampler`是默认的, 你可以使用`spring.sleuth.sampler.percentage`配置输出

通过创建一个bean定义就可以新建一个sampler

1. `@Bean`

```

2. public Sampler defaultSampler() {
3.     return new AlwaysSampler();
4. }

```

Instrumentation

Spring Cloud Sleuth自动装配所有Spring应用，因此你不用做任何事来让他工作，装配是使用一系列技术添加的，例如对于一个servlet web应用我们使用一个Filter，对于SpringIntegration我们使用 `ChannelInterceptors`。

用户可以使用span tags定制关键字，为了限制span数据量，一般一个HTTP请求只会被少数元数据标记，例如status code、host以及URL，用户可以通过配置 `spring.sleuth.keys.http.headers` (一系列头名称)添加request headers。

tags仅在Sampler允许其被收集和输出时工作(默认情况其不工作，因此不会有在不配置的情况下收集过多数据的意外危险出现)

Currently the instrumentation in Spring Cloud Sleuth is eager - it means that we're actively trying to pass the tracing context between threads. Also timing events are captured even when sleuth isn't exporting data to a tracing system. This approach may change in the future towards being lazy on this matter.

Span生命周期

通过Trace接口的方式可以在Span上进行如下操作：

- start - 当打开一个span时，其名字被指定且开始时间戳被记录
- close - span已经结束(span的结束时间已被记录)并且如果span是输出的，他将是Zipkin合适的收集项，span在当前线程也将被移除
- continue - span的一个新实例将被创建，然而他将是正是正在运行的span的一个复制体
- detach - span不会停止或关闭，他只会被从当前线程中移除
- create with explicit parent - 建立一个新的span并设置一个明确的parent给他

Creating and closing spans

使用Tracer接口可以手动新建spans

```

1. // Start a span. If there was a span present in this thread it will become
2. // the `newSpan`'s parent.
3. Span newSpan = this.tracer.createSpan("calculateTax");
4. try {
5.     // ...
6.     // You can tag a span

```

```

7.     this.tracer.addTag("taxValue", taxValue);
8.     // ...
9.     // You can log an event on a span
10.    newSpan.logEvent("taxCalculated");
11. } finally {
12.     // Once done remember to close the span. This will allow collecting
13.     // the span to send it to Zipkin
14.    this.tracer.close(newSpan);
15. }

```

在例子中我们可以看到如何新建一个span实例，假设在当前线程中已经有一个span，那么新建的线程将会是这个线程的parent。

IMPORTANT

新建span后要记得清除他！如果你想要将一个span发送给Zipkin，不要忘记关闭他。

Continuing spans

有时你不需要新建一个span但你想持续使用，这种情况的例子可能如下(当然实际依赖于使用情况):

- **AOP** - 如果在实际应用前已经有一个span新建可用，那么就不需要新建一个span
- **Hystrix** - 对于当前处理流程而言，执行Hystrix操作是最为合理的一部分，实际上只有技术实现细节的话，不必将他作为分离的部分反映在tracing中

span的持续实例等同于正在运行的:

```

1. Span continuedSpan = this.tracer.continueSpan(spanToContinue);
2. assertThat(continuedSpan).isEqualTo(spanToContinue);

```

可以使用Tracer接口延续一个span

```

1. // let's assume that we're in a thread Y and we've received
2. // the `initialSpan` from thread X
3. Span continuedSpan = this.tracer.continueSpan(initialSpan);
4. try {
5.     // ...
6.     // You can tag a span
7.     this.tracer.addTag("taxValue", taxValue);
8.     // ...
9.     // You can log an event on a span
10.    continuedSpan.logEvent("taxCalculated");
11. } finally {
12.     // Once done remember to detach the span. That way you'll
13.     // safely remove it from the current thread without closing it

```

```

14.     this.tracer.detach(continuedSpan);
15. }

```

IMPORTANT

新建一个span后记得清除他！如果有些工作在一个线程(*e.g.* *thread X*)中已经结束并且他在等待另外的线程(*e.g.* *Y, Z*)结束时，不要忘记分离span，在线程*Y, Z*中的spans在他们工作结束时也应被分离，结果收集完成时thread X中的span应该被关闭

Creating spans with an explicit parent

如果你想新建一个span并且提供一个明确的parent给他，假设span的parent在一个thread中，而你想在另一个thread中新建span，Tracer接口的startSpan命令就是你需要的。

```

1. // let's assume that we're in a thread Y and we've received
2. // the `initialSpan` from thread X. `initialSpan` will be the parent
3. // of the `newSpan`
4. Span newSpan = this.tracer.createSpan("calculateCommission", initialSpan);
5. try {
6.     // ...
7.     // You can tag a span
8.     this.tracer.addTag("commissionValue", commissionValue);
9.     // ...
10.    // You can log an event on a span
11.    newSpan.logEvent("commissionCalculated");
12. } finally {
13.    // Once done remember to close the span. This will allow collecting
14.    // the span to send it to Zipkin. The tags and events set on the
15.    // newSpan will not be present on the parent
16.    this.tracer.close(newSpan);
17. }

```

IMPORTANT

记得在新建这样的span后关闭他，否则你在你的log中看到大量的相关warning，更糟糕的是你的span不会正常关闭，这样的话就无法被Zipkin收集

命名spans

为span命名是很重要的工作，span名称必须描述了一个操作名称，名称必须要简明(*e.g.* 不包括标识符)。

Since there is a lot of instrumentation going on some of the span names will be artificial like:

- controller-method-name when received by a Controller with a methodName controllerMethodName
- async for asynchronous operations done via wrappedCallable and Runnable
- `@Scheduled` annotated methods will return the simple name of the class

Fortunately, for the asynchronous processing you can provide explicit naming

@SpanName 注解

可以使用 `@SpanName` 注解明确命名 span

```
1. @SpanName("calculateTax")
2. class TaxCountingRunnable implements Runnable {
3.
4.     @Override public void run() {
5.         // perform logic
6.     }
7. }
```

在这种情况下，使用下面的方式便命名一个 span 为 calculateTax

```
1. Runnable runnable = new TraceRunnable(tracer, spanNamer, new TaxCountingRunnable());
2. Future<?> future = executorService.submit(runnable);
3. // ... some additional logic ...
4. future.get();
```

toString() 方法

为 Runnable 或 Callable 建立分离的 classes 是非常少见的，一般建立这些 classes 的匿名实例，你不能注解这些 classes 除非 override，如果没有 `@SpanName` 注解，我们将会检查 class 是否使用传统的 `toString()` 方法实现

执行这些代码将新建一个名为 calculateTax 的 span：

```
1. Runnable runnable = new TraceRunnable(tracer, spanNamer, new Runnable() {
2.     @Override public void run() {
3.         // perform logic
4.     }
5.
6.     @Override public String toString() {
7.         return "calculateTax";
8.     }
9. });
```

```

10. Future<?> future = executorService.submit(runnable);
11. // ... some additional logic ...
12. future.get();

```

定制化

使用 `SpanInjector` 和 `SpanExtractor` 你可以定制化span的新建和传播。

当前有两种built-in方法来在进程间传递tracing信息：

- 通过SpringIntegration
- 通过HTTP

span id是从Zipkin-compatible(B3)头中提取的(不论Message或HTTP头)，以此来开始或加入一个存在的trace，trace信息被注入到输出请求中，这样后面的步骤就可以提取他。

Spring Integration

对于Spring Integration，存在beans负责span从Message的创建和使用tracing信息装配MessageBuilder

```

1. @Bean
2. public SpanExtractor<Message> messagingSpanExtractor() {
3.     ...
4. }
5.
6. @Bean
7. public SpanInjector<MessageBuilder> messagingSpanInjector() {
8.     ...
9. }

```

用户可以使用自己的实现来override他，或者添加 `@Primary` 注解到你的bean定义

HTTP

对于HTTP，存在beans负责span从HttpServletRequest的创建和使用tracing信息装配HttpServletResponse。

```

1. @Bean
2. public SpanExtractor<HttpServletRequest> httpServletRequestSpanExtractor() {
3.     ...
4. }

```

```

5.
6. @Bean
7. public SpanInjector<HttpServletResponse> httpServletResponseSpanInjector() {
8.     ...
9. }

```

用户可以使用自己的实现来override他，或者添加 `@Primary` 注解到你的bean定义

Example

对比传统的兼容Zipkin，tracingHTTP头名有以下格式

- traceid - correlationId
- spanid - mySpanId

以下是一个 `SpanExtractor` 的例子

```

1. static class CustomHttpServletRequestSpanExtractor
2.     implements SpanExtractor<HttpServletRequest> {
3.
4.     @Override
5.     public Span joinTrace(HttpServletRequest carrier) {
6.         long traceId = Span.hexToId(carrier.getHeader("correlationId"));
7.         long spanId = Span.hexToId(carrier.getHeader("mySpanId"));
8.         // extract all necessary headers
9.         Span.SpanBuilder builder = Span.builder().traceId(traceId).spanId(spanId);
10.        // build rest of the Span
11.        return builder.build();
12.    }
13. }

```

以下 `SpanInjector` 将被建立

```

1. static class CustomHttpServletResponseSpanInjector
2.     implements SpanInjector<HttpServletResponse> {
3.
4.     @Override
5.     public void inject(Span span, HttpServletResponse carrier) {
6.         carrier.addHeader("correlationId", Span.idToHex(span.getTraceId()));
7.         carrier.addHeader("mySpanId", Span.idToHex(span.getSpanId()));
8.         // inject the rest of Span values to the header
9.     }
10. }

```

并且你可以这样注册他们

```

1. @Bean
2. @Primary
3. SpanExtractor<HttpServletRequest> customHttpServletRequestSpanExtractor() {
4.     return new CustomHttpServletRequestSpanExtractor();
5. }
6.
7. @Bean
8. @Primary
9. SpanInjector<HttpServletResponse> customHttpServletResponseSpanInjector() {
10.    return new CustomHttpServletResponseSpanInjector();
11. }

```

Custom SA tag in Zipkin

Sometimes you want to create a manual Span that will wrap a call to an external service which is not instrumented. What you can do is to create a span with the `peer.service` tag that will contain a value of the service that you want to call. Below you can see an example of a call to Redis that is wrapped in such a span.

```

1. org.springframework.cloud.sleuth.Span newSpan = tracer.createSpan("redis");
2. try {
3.     newSpan.tag("redis.op", "get");
4.     newSpan.tag("lc", "redis");
5.     newSpan.logEvent(org.springframework.cloud.sleuth.Span.CLIENT_SEND);
6.     // call redis service e.g
7.     // return (SomeObj) redisTemplate.opsForHash().get("MYHASH", someObjKey);
8. } finally {
9.     newSpan.tag("peer.service", "redisService");
10.    newSpan.tag("peer.ipv4", "1.2.3.4");
11.    newSpan.tag("peer.port", "1234");
12.    newSpan.logEvent(org.springframework.cloud.sleuth.Span.CLIENT_RECV);
13.    tracer.close(newSpan);
14. }

```

Remember not to add both `peer.service` tag and the SA tag! You have to add only `peer.service`.

Spring Data as Messages

可以通过Spring Cloud Stream来积累和发送span数据，配置时需要包含spring-cloud-sleuth-streamjar为依赖且增加一个Channel Binder实现方式(e.g. spring-cloud-starter-stream-rabbit对应RabbitMQ或spring-cloud-starter-stream-kafka对应

Kafka), 使用payload格式Spans将自动把你的app变为一个信息生产者

Zipkin Consumer

有一种特殊而又便利的注解方式, 即为span数据建立一个信息消费者, 并将他推到一个Zipkin SpanStore中

```

1. @SpringBootApplication
2. @EnableZipkinStreamServer
3. public class Consumer {
4.     public static void main(String[] args) {
5.         SpringApplication.run(Consumer.class, args);
6.     }
7. }
```

这种应用将通过Spring Cloud Stream Binder监听不论何种方式传输的span数据(e.g. 包括spring-cloud-starter-stream-rabbit对应RabbitMQ, 和对应Redis和Kafka的类似starter存在), 如果添加以下UI依赖

```

1. <groupId>io.zipkin.java</groupId>
2. <artifactId>zipkin-autoconfigure-ui</artifactId>
```

你将启动一个Zipkin server应用, 他将通过端口9411访问UI和api。

默认SpanStore是in-memory的(适合于demos且启动迅速), 你可以添加MySQL和spring-boot-starter-jdbc到你的系统环境并通过配置激活JDBC SpanStore。例如:

```

1. spring:
2.   rabbitmq:
3.     host: ${RABBIT_HOST:localhost}
4.   datasource:
5.     schema: classpath:/mysql.sql
6.     url: jdbc:mysql://${MYSQL_HOST:localhost}/test
7.     username: root
8.     password: root
9.   # Switch this on to create the schema on startup:
10.    initialize: true
11.    continueOnError: true
12.   sleuth:
13.     enabled: false
14.   zipkin:
15.     storage:
16.       type: mysql
```

`@EnableZipkinStreamServer`也使用`@EnableZipkinServer`注解，因此进程也会显示标准Zipkin服务终端以通过HTTP收集span，且可以通过Zipkin Web UI查询

Custom Consumer

使用spring-cloud-sleuth-stream且绑定SleuthSink可以很方便的实现定制消费者。例子：

```
1. @EnableBinding(SleuthSink.class)
2. @SpringBootApplication(exclude = SleuthStreamAutoConfiguration.class)
3. @MessageEndpoint
4. public class Consumer {
5.
6.     @ServiceActivator(inputChannel = SleuthSink.INPUT)
7.     public void sink(Spans input) throws Exception {
8.         // ... process spans
9.     }
10. }
```

上述的消费者应用明确排除`SleuthStreamAutoConfiguration`，因此他不会给自己发消息，但这是可选的(你可能想要trace请求到消费者app)

度量

当前Spring Cloud Sleuth记录非常简单的spans metrics，使用Spring Boot的metrics support来计算接收丢弃的span数量，当有span发送给Zipkin时，接收span的数量就会增加，如果有错误发生，丢弃span数量就会增加。

Integrations

Runnable and Callable

如果你要将你的逻辑包裹在Runnable或Callable中，足够将这些classes放到他们的Sleuth代表中。

Example for Runnable:

```
1. Runnable runnable = new Runnable() {
2.     @Override
3.     public void run() {
4.         // do some work
5.     }
```

```

6.
7.     @Override
8.     public String toString() {
9.         return "spanNameFromToStringMethod";
10.    }
11. };
12. // Manual `TraceRunnable` creation with explicit "calculateTax" Span name
13. Runnable traceRunnable = new TraceRunnable(tracer, spanNamer, runnable, "calculateTax");
14. // Wrapping `Runnable` with `Tracer`. The Span name will be taken either from the
15. // `@SpanName` annotation or from `toString` method
16. Runnable traceRunnableFromTracer = tracer.wrap(runnable);

```

Example for Callable:

```

1. Callable<String> callable = new Callable<String>() {
2.     @Override
3.     public String call() throws Exception {
4.         return someLogic();
5.     }
6.
7.     @Override
8.     public String toString() {
9.         return "spanNameFromToStringMethod";
10.    }
11. };
12. // Manual `TraceCallable` creation with explicit "calculateTax" Span name
13. Callable<String> traceCallable = new TraceCallable<>(tracer, spanNamer, callable,
14.     "calculateTax");
14. // Wrapping `Callable` with `Tracer`. The Span name will be taken either from the
15. // `@SpanName` annotation or from `toString` method
16. Callable<String> traceCallableFromTracer = tracer.wrap(callable);

```

这种方式你可以保证一个新的Span在每次执行时新建和关闭。

Hystrix

传统并发策略

我们以将所有的 `Callable` 实例置入到他们的Sleuth代表 - `TraceCallable` 的方式来记录一个传统的 `HystrixConcurrencyStrategy`，策略的打开或延续一个span取决于在Hystrix操作被调用前tracing是否在工作，为了使传统Hystrix并发策略无效可以设置 `spring.sleuth.hystrix.strategy.enable` 为false。

手动操作设置

假设你有以下HystrixCommand:

```

1. HystrixCommand<String> hystrixCommand = new HystrixCommand<String>(setter) {
2.     @Override
3.     protected String run() throws Exception {
4.         return someLogic();
5.     }
6. };

```

为了传递tracing信息你必须将同样的逻辑置于 `HystrixCommand` 的Sleuth版本中，也就是 `TraceCommand`：

```

1. TraceCommand<String> traceCommand = new TraceCommand<String>(tracer, traceKeys, setter) {
2.     @Override
3.     public String doRun() throws Exception {
4.         return someLogic();
5.     }
6. };

```

RxJava

我们记录了一个典型的RxJavaSchedulersHook，他将所有Action0实例置入到他们的Sleuth代表-TraceAction中，hook打开或延续一个span取决于Action被安排前tracing是否已经在工作，为了使RxJavaSchedulersHook无效可设置 `spring.sleuth.rxjava.schedulers.hook.enabled` 为 false。

You can define a list of regular expressions for thread names, for which you don't want a Span to be created. Just provide a comma separated list of regular expressions in the `spring.sleuth.rxjava.schedulers.ignoredthreads` property.

HTTP integration

将 `spring.sleuth.web.enabled` 配置值设置为false可以使这章中的特征方法无效

HTTP Filter

通过TraceFilter，所有抽样输入的请求都会归结到span的创建，span的名称为“http+请求发送的路径”，例如，如果请求发送到/foo/bar，名称即为http:/foo/bar，你可以配置通过 `spring.sleuth.web.skipPattern`，那些URIs将被过滤掉，如果你在环境中添加了ManagementServerProperties，你的contextPath值会附加到过滤配置上。

HandlerInterceptor

由于需要span名称的精确，我们使用一个TraceHandlerInterceptor来置入一个存在的

HandlerInterceptor或直接添加到存在的HandlerInterceptors列表中，TraceHandlerInterceptor添加一个特殊的请求属性给HttpServletRequest，如果TraceFilter没有看到属性，他会建立一个“fallback”span，这是一个建立在服务端的附加的span，此时trace在UI中可以正确的显示。

Async Servlet support

If your controller returns a Callable or a WebAsyncTask Spring Cloud Sleuth will continue the existing span instead of creating a new one.

HTTP client integration

同步RestTemplate

我们注入一个RestTemplate拦截器来保证所有的tracing信息被发送到请求端，每当一个请求被生成，一个新的span将被创建，他会在接收应答后关闭，为了限制同步RestTemplate只需要设置 `spring.sleuth.web.client.enabled` 为false。

你必须注册一个RestTemplate为bean以使得拦截器可以注入，如果你使用一个新的关键字建立一个RestTemplate实例，instrumentation将无法工作

异步RestTemplate

传统的instrumentation是通过发送接收请求来建立关闭span的，你可以通过注册你的bean来定制ClientHttpRequestFactory和AsyncClientHttpRequestFactory，记得使用tracing compatible实现方式(e.g. 不要忘记将ThreadPoolTaskScheduler置入一个TraceAsyncListenableTaskExecutor)，传统请求工厂例子如下：

```

1. @EnableAutoConfiguration
2. @Configuration
3. public static class TestConfiguration {
4.
5.     @Bean
6.     ClientHttpRequestFactory mySyncClientFactory() {
7.         return new MySyncClientHttpRequestFactory();
8.     }
9.
10.    @Bean
11.    AsyncClientHttpRequestFactory myAsyncClientFactory() {
12.        return new MyAsyncClientHttpRequestFactory();
13.    }
14. }
```

通过设置 `spring.sleuth.web.async.client.enabled` 为false可以限制 `AsyncRestTemplate`，使默认的 `TraceAsyncClientHttpRequestFactoryWrapper` 无效可以设

置 `spring.sleuth.web.async.client.factory.enabled` 为false，如果你不想创建 `AsyncRestClient`，设置 `spring.sleuth.web.async.client.template.enabled` 为false。

Feign

默认Spring Cloud Sleuth通过 `TraceFeignClientAutoConfiguration` 提供feign的集成，你可以设置 `spring.sleuth.feign.enabled` 为false来使他无效，如果这样设置那么所有feign相关的装配都无法发生。

通过FeignBeanPostProcessor feign装配的部分结束，可以设置

置 `spring.sleuth.feign.processor.enabled` 为false来是他无效化，如果你这样设置，Spring Cloud Sleuth将不会装配任何你的传统feign组件，所有默认装配保持原有状态。

异步通信

@Async注解方法

在Spring Cloud Sleuth中，我们装配异步关联组件以使得tracing信息可以在threads间传递，你可以通过设置 `spring.sleuth.async.enabled` 值为false来使其无效化。

如果你使用@Async来注解你的方法，我们将自动建立一个新的span：

- span名称将是注解方法名
- span将被标注为方法类名和方法名

@Scheduled注解方法

在Spring Cloud Sleuth中，我们装配scheduled执行方法以使得tracing信息可以在threads间传递，你可以通过设置 `spring.sleuth.scheduled.enabled` 值为false来使其无效化。

如果你使用 `@Scheduled` 来注解你的方法，我们将自建立一个新的span：

- span名称将是注解方法名
- span将被标注为方法类名和方法名

如果在一些@Scheduled注解类中你想跳过span新建过程，可以设置

置 `spring.sleuth.scheduled.skipPattern` 为一个指定的表达式，这将匹配 `@Scheduled` 注解类的完整描述名称。

Executor, ExecutorServiceand ScheduledExecutorService

我们提供了LazyTraceExecutor, TraceableExecutorService和TraceableScheduledExecutorService。每当一个新的任务被提交、调用或scheduled时，这些实现会建立新的spans。

以下是当使用CompletableFuture时如何用TraceableExecutorService传递tracing信息：

```

1. CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {
2.     // perform some logic
3.     return 1_000_000L;
4. }, new TraceableExecutorService(executorService,
5.     // 'calculateTax' explicitly names the span - this param is optional
6.     tracer, traceKeys, spanNamer, "calculateTax"));

```

Messaging

Spring Cloud Sleuth集成了Spring Integration。他会建立span来发布或订阅事件，设置 `spring.sleuth.integration.enabled` 为false可以使Spring Integration无效。

Spring Cloud Sleuth到1.0.4版本前都是使用消息传递时发送无效tracing头，这些头和在HTTP(包含 -)发送的名称时一样的，为了在1.0.4版本的向后兼容目的，我们开始发送所有有效和无效的头，请更新到1.0.4，因为在Spring Cloud Sleuth 1.1中我们将会移除对分离头的支持。

从1.0.4后可以明确设置 `spring.sleuth.integration.patterns` 模式来提供你想要包含的tracing信道名称，默认所有的信道已被包含在内。

Zuul

我们注册Zuul过滤器来传播tracing信息(请求头使用tracing数据填满)，可以设置 `spring.sleuth.zuul.enabled` 为false来关闭Zuul服务。

Spring Cloud stream

- Spring Cloud Stream
 - 简介
 - 主要概念
 - 应用模型
 - Fat JAR
 - 绑定抽象
 - 持久化发布 / 订阅支持
 - 消费者组
 - 持久性
 - 分区支持
 - 编程模型
 - 声明和绑定通道
 - 通过 `@EnableBinding` 触发绑定
 - `@Input` 与 `@Output`
 - 访问绑定通道
 - 生产和消费消息
 - 聚合
 - RxJava 支持
 - 绑定器
 - 生产者与消费者
 - Binder SPI
 - Binder Detection
 - Classpath Detection
 - Multiple Binders on the Classpath
 - Connecting to Multiple Systems
 - Binder configuration properties
 - Implementation strategies
 - Kafka Binder
 - RabbitMQ Binder
 - 配置管理
 - SpringCloudStream配置项
 - Binding配置项
 - SpringCloudStream的bindings配置
 - Consumer properties
 - Producer Properties
 - Binder-Specific Configuration
 - Rabbit-Specific Settings
 - RabbitMQ Binder Properties
 - RabbitMQ Consumer Properties

- Rabbit Producer Properties
- Kafka-Specific Settings
 - Kafka Binder Properties
 - Kafka Consumer Properties
 - Kafka Producer Properties
- Content Type and Transformation
 - MIME types
 - MIME types and Java types
 - @StreamListener and Message Conversion
- 应用程序间通信
 - 连接多个应用程序实例
 - 实例索引和实例数
 - 分区
 - 配置Output Bindings
 - 配置Input Bindings
- Testing
- 健康指示器
- 例子
- Getting Started

Spring Cloud Stream

这一节将更详细地介绍如何使用SpringCloudStream工作，它涵盖主题了创建和运行Stream应用。

简介

SpringCloudStream是一个构建消息驱动的微服务框架。SpringCloudStream构建在SpringBoot之上用以创建工业级的应用程序，并且Spring Integration提供了和消息代理的连接。SpringCloudStream提供几个厂商消息中间件个性化配置，引入发布订阅、消费组和分区的语义概念。

添加@EnableBinding注解在你的程序中，被@StreamListener修饰的方法可以立即连接到消息代理接收流处理事件。下面是一个简单的接收外部消息的接收器应用程序

```

1. @SpringBootApplication
2. public class StreamApplication {
3.
4.     public static void main(String[] args) {
5.         SpringApplication.run(StreamApplication.class, args);
6.     }
7. }
8.

```

```

9. @EnableBinding(Sink.class)
10. public class TimerSource {
11.
12.     ...
13.
14.     @StreamListener(Sink.INPUT)
15.     public void processVote(Vote vote) {
16.         votingService.recordVote(vote);
17.     }
18. }

```

`@EnableBinding`注解使用一个或者多个接口作为参数（本例子中，参数是单独的Sink接口），接口中可以定义输入或输出的channels，SpringCloudStream定义三个接

□ `Source` ， `Sink` ， `Processor` ， 你也可以自定义接口。

下面是Sink接口的定义：

```

1. public interface Sink {
2.     String INPUT = "input";
3.
4.     @Input(Sink.INPUT)
5.     SubscribableChannel input();
6. }

```

`@Input` 定义了一个接收消息的输入channel，`@Output` 定义了一个发布消息的输出channel，这两个注解支持一个参数作为channel名称，如果没有设置参数则注解修饰的方法名将被设置为channel名称。

SpringCloudStream会为你创建一个接口的实现，你可以通过自动装配在应用中使用它，如下例：

```

1. @RunWith(SpringJUnit4ClassRunner.class)
2. @SpringApplicationConfiguration(classes = StreamApplication.class)
3. @WebAppConfiguration
4. @DirtiesContext
5. public class StreamApplicationTests {
6.
7.     @Autowired
8.     private Sink sink;
9.
10.    @Test
11.    public void contextLoads() {
12.        assertNotNull(this.sink.input());
13.    }
14. }

```

主要概念

SpringCloudStream提供了很多抽象和基础组件来简化消息驱动型微服务应用。包含以下内容：

- Spring Cloud Stream的应用模型
- 绑定抽象
- 持久化发布 / 订阅支持
- 消费者组支持
- 分片支持 (Partitioning Support)
- 可插拔API

应用模型

SpringCloudStream由一个中立的中间件内核组成。SpringCloudStream会注入输入和输出的channels，应用程序通过这些channels与外界通信，而channels则是通过一个明确的中间件Binder与外部brokers连接。



Fat JAR

SpringCloudStream可以在ide中运行一个单独的实例用来测试，如果要部署在生产环境中则可以通过Maven或者Gradle提供的Spring Boot工具创建可执行JAR（胖JAR）

绑定抽象

SpringCloudStream提供对Kafka, Rabbit MQ, Redis, 和Gemfire的Binder实现。Spring Cloud Stream还包括了一个TestSupportBinder，TestSupportBinder预留一个未更改的channel以便于直接地、可靠地和channels通信。您可以使用可扩展的API来编写自己的Binder。

SpringCloudStream使用SpringBoot做配置，绑定抽象使得SpringCloudStream应用可以灵活地连接到中间件。比如，开发者可以在运行时动态的选择channels连接的目标（可以是kafka topics或RabbitMQ exchanges）。这样的配置可以通过外部配置或任何SpringBoot支持的形式（包括应用参数，环境变量和application.yml或application.properties文件）。简介一节提到的sink例子中，将 `spring.cloud.stream.bindings.input.destination` 设置成 `raw-sensor-data` 程序会从命名为 `raw-sensor-data` 的kafka主题中读取数据，或者从一个绑定到 `raw-sensor-data` 的rabbitmq交换机的队列中读取数据。

SpringCloudStream能自动发现并使用类路径中的binder，您可以很容易地以相同的代码使用不同类型的中间件：只需要在build的时候引入不同的binder。对于更复杂的情况，你可以引入多个binders并选择使用哪一个，甚至可以在运行时根据不同的channels选择不同的binder。

持久化发布 / 订阅支持

应用间通信遵照发布-订阅模型，数据通过共享的topics进行广播，下图显示了SpringCloudStream应用交互的典型部署。



数据被发送到一个公共的目标 `raw-sensor-data`，在目标中，数据分别被两个独立的微服务加工，一个微服务计算平均窗口时间，另一个将原始数据存储到HDFS。为了处理数据，两个微服务在运行时声明这个topic作为他们的输入源。

发布-订阅通信模型降低了生产者和消费者的复杂性，并允许新的应用程序被添加到拓扑结构，而不会破坏现有的流程。例如，下游的平均计算应用程序，您可以添加一个应用程序，该应用程序计算最高温度用来显示和监控。然后您可以再添加一个基于相同数据流的，解释故障检测的另一个应用程序。通过共同的topics做沟通相比点对点的队列更能减少微服务间的耦合。

发布订阅不是一个新概念，SpringCloudStream在你的应用中提供一个额外的手段供你选择。通过使用本地中间件支持，SpringCloudStream简化了不同平台上的发布订阅模型。

消费者组

虽然发布-订阅模型可以很容易地通过共享topics连接应用程序，但创建一个应用多实例的的扩张能力同等重要。当这样做时，应用程序的不同实例被放置在一个竞争的消费者关系中，其中只有一个实例将处理一个给定的消息。

SpringCloudStream利用消费者组定义这种行为（这种分组类似于Kafka consumer groups，灵感也来源于此），每个消费者通过 `spring.cloud.stream.bindings.input.group` 指定一个组名称，以下图所示的消费者为例，应分别设

置 `spring.cloud.stream.bindings.input.group=hdfsWrite` 和 `spring.cloud.stream.bindings.input.group=average`。



所有订阅指定topics的组都会收到发布数据的一份副本，但是每一个组内只有一个成员会收到该消息。默认情况下，当一个组没有指定时，SpringCloudStream将分配给一个匿名的、独立的只有一个成员的消费组，该组与所有其他组都处于一个发布-订阅关系中。

持久性

SpringCloudStream一致性模型中，消费者组订阅是持久的，也就是说一个绑定的实现确保组的订阅者是持久的。一旦组中至少有一个成员创建了订阅，这个组就会收到消息，即使组中所有的应用都被停止了，组仍然会收到消息。

匿名订阅是非持久的，一些**binder**的实现（如：*RabbitMQ*），可以创建非持久化（*non-durable*）组订阅

在一般情况下，将应用绑定到给定目标的时候，最好指定一个消费者组，当扩展一个SpringCloudStream应用时，必须为每个输入bindings指定一个消费组，这防止了应用程序的实例接收重复的消息（除非该行为是需要的，这是不寻常的）。

分区支持

SpringCloudStream支持在一个应用程序的多个实例之间数据分区，在分区的情况下，物理通信介质（例如，topic代理）被视为多分区结构。一个或多个生产者应用程序实例将数据发送给多个消费应用实例，并保证共同的特性的数据由相同的消费者实例处理。

SpringCloudStream提供了一个通用的抽象，用于统一方式进行分区处理，因此分区可以用于自带分区的代理（如kafka）或者不带分区的代理（如rabbitemq）



分区在有状态处理中是一个很重要的概念，其重要性体现在性能和一致性上，要确保所有相关数据被一并处理，例如，在时间窗平均计算的例子中，给定传感器测量结果应该都由同一应用实例进行计算。

如果要设置分区处理方案，需要配置数据生产端点和数据消费端点

编程模型

本节介绍SpringCloudStream的编程模型，SpringCloudStream提供了一些预定义的注解，用于绑定输入和输出channels，以及如何监听channels。

声明和绑定通道

通过 `@EnableBinding` 触发绑定

将`@EnableBinding`注解添加到应用的配置类，就可以把一个spring应用转换成SpringCloudStream应用，`@EnableBinding`注解本身就包含`@Configuration`注解，会触发SpringCloudStream 基本配置。

```
1. ...
2. @Import(...)
3. @Configuration
4. @EnableIntegration
5. public @interface EnableBinding {
6.     ...
7.     Class<?>[] value() default {};
```

```
8. }
```

`@EnableBinding`注解可以接收一个或多个接口类作为参数，后者包含代表了可绑定构件（一般来说是消息通道）的方法

在`SpringCloudStream1.0`中，仅有的可绑定构件是`Spring Messaging` `MessageChannel` 以及它的扩展 `SubscribableChannel` 和 `PollableChannel` 。未来版本会使用相同的机制扩展对其他类型构件的支持。在本文档中，会继续引用`channels`。

`@Input` 与 `@Output`

一个`SpringCloudStream`应用可以有任意数目的input和output通道，后者通过 `@Input` 和 `@Output` 注解在接口中定义。

```
1. public interface Barista {
2.
3.     @Input
4.     SubscribableChannel orders();
5.
6.     @Output
7.     MessageChannel hotDrinks();
8.
9.     @Output
10.    MessageChannel coldDrinks();
11. }
```

使用这个接口作为`@EnableBinding`的参数，将触发三个bound channels的创建，后者的分别被命名为 `orders` ， `hotDrinks` ， `coldDrinks`

```
1. @EnableBinding(Barista.class)
2. public class CafeConfiguration {
3.
4.     ...
5. }
```

定制通道名字

使用 `@Input` 和 `@Output` 注解，您可以为该channel指定一个自定义的channel名称，如下面的示例所示：

```
1. public interface Barista {
2.     ...
3.     @Input("inboundOrders")
4.     SubscribableChannel orders();
5. }
```

在这个例子中，创建的绑定channel将被命名为inboundorders。

Source , **Sink** , **Processor**

最常见的场景中，包含一个输入通道或者包含一个输出通道或者二者都包含，SpringCloudStream提供了三个开箱即用的预定义接口。

Source 用于有单个输出（outbound）通道的应用。

```
1. public interface Source {
2.
3.     String OUTPUT = "output";
4.
5.     @Output(Source.OUTPUT)
6.     MessageChannel output();
7.
8. }
```

Sink 用于有单个输入（inbound）通道的应用。

```
1. public interface Sink {
2.
3.     String INPUT = "input";
4.
5.     @Input(Sink.INPUT)
6.     SubscribableChannel input();
7.
8. }
```

Processor 用于单个应用同时包含输入和输出通道的情况。

```
1. public interface Processor extends Source, Sink {
2. }
```

SpringCloudStream对这些接口不提供特殊的处理，仅提供开箱即用的特性。

访问绑定通道

注入已绑定接口

对于每一个绑定的接口，SpringCloudStream将产生一个实现接口的bean，调用这个生成类的 **@Input** 或 **@Output** 方法，会返回一个相应的channel。

下面的例子中，当hello被调用时输出channel会发送一个消息，在注入的Source上提供唤醒output()来检索到目标通道

```

1. @Component
2. public class SendingBean {
3.
4.     private Source source;
5.
6.     @Autowired
7.     public SendingBean(Source source) {
8.         this.source = source;
9.     }
10.
11.     public void sayHello(String name) {
12.         source.output().send(MessageBuilder.withPayload(body).build());
13.     }
14. }

```

直接注入到通道

绑定的通道也可以直接注入

```

1. @Component
2. public class SendingBean {
3.
4.     private MessageChannel output;
5.
6.     @Autowired
7.     public SendingBean(MessageChannel output) {
8.         this.output = output;
9.     }
10.
11.     public void sayHello(String name) {
12.         output.send(MessageBuilder.withPayload(body).build());
13.     }
14. }

```

如果channel的名字是在注解中指定的，那么请使用这个名字，而不是使用方法名。如下：

```

1. public interface CustomSource {
2.     ...
3.     @Output("customOutput")
4.     MessageChannel output();
5. }

```

该通道将被注入，如下面的示例所示：

```

1. @Component

```



```

2. public class SendingBean {
3.
4.     @Autowired
5.     private MessageChannel output;
6.
7.     @Autowired @Qualifier("customOutput")
8.     public SendingBean(MessageChannel output) {
9.         this.output = output;
10.    }
11.
12.    public void sayHello(String name) {
13.        customOutput.send(MessageBuilder.withPayload(body).build());
14.    }
15. }

```

生产和消费消息

可以使用Spring Integration的注解或者SpringCloudStream的 `@StreamListener` 注解来实现一个SpringCloudStream应用。 `@StreamListener` 注解模仿其他spring消息注解（例如 `@MessageMapping` , `@JmsListener` , `@RabbitListener` 等），但是它增加了内容类型管理和类型强制特性。

原生Spring Integration支持

SpringCloudStream是基于Spring Integration的，所以完全的继承了后者的基础设施以及构件本身，例如，可以将 `Source` 的output通道连接到一个 `MessageSource`

```

1. @EnableBinding(Source.class)
2. public class TimerSource {
3.
4.     @Value("${format}")
5.     private String format;
6.
7.     @Bean
8.     @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay = "${fixedDelay}",
9.         maxMessagesPerPoll = "1"))
10.    public MessageSource<String> timerMessageSource() {
11.        return () -> new GenericMessage<>(new SimpleDateFormat(format).format(new Date()));
12.    }
13. }

```

或者你可以在transformer中使用处理器的channels:

```

1. @EnableBinding(Processor.class)
2. public class TransformProcessor {
3.     @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)

```

```

4.     public Object transform(String message) {
5.         return message.toUpperCase();
6.     }
7. }

```

使用 `@StreamListener` 进行自动内容类型处理

作为原生Spring Integration的补充，SpringCloudStream提供了自己的 `@StreamListener` 注解，该注解模仿spring的其它消息注解（如 `@MessageMapping`，`@JmsListener`，`@RabbitListener` 等）。`@StreamListener` 注解提供了一种更简单的模型来处理输入消息，尤其是处理包含内容类型管理和类型强制的用例的情况。

SpringCloudStream提供了一个扩展的 `MessageConverter` 机制，该机制提供绑定通道实现数据处理，本例子中，数据会分发给带 `@StreamListener` 注解的方法。下面例子展示了处理外部 `Vote` 事件的应用：

```

1. @EnableBinding(Sink.class)
2. public class VoteHandler {
3.
4.     @Autowired
5.     VotingService votingService;
6.
7.     @StreamListener(Sink.INPUT)
8.     public void handle(Vote vote) {
9.         votingService.record(vote);
10.    }
11. }

```

`@StreamListener` 和Spring Integration的 `@ServiceActivator` 是有区别的，区别体现在当输入消息内容头为application/json的字符串的时候，`@StreamListener`的MessageConverter机制会使用contentType头将string解析为Vote对象。

和其他Spring Messaging方法一样，方法参数可以被如下注解修饰，`@Payload`，`@Headers`和`@Header`

对于那些有返回数据的方法，必须使用`@SendTo`注解来指定返回数据的输出绑定目标。

```

1. @EnableBinding(Processor.class)
2. public class TransformProcessor {
3.
4.     @Autowired
5.     VotingService votingService;
6.
7.     @StreamListener(Processor.INPUT)
8.     @SendTo(Processor.OUTPUT)
9.     public VoteResult handle(Vote vote) {

```

```

10.     return votingService.record(vote);
11. }
12. }

```

在RabbitMQ中，内容类型头可以由外部应用设定。SpringCloudStream支持他们作为一个扩展的内部协议，用于任何类型的运输（包括运输，如Kafka，不能正常支持headers）

聚合

SpringCloudStream可以支持多种应用聚合，直接连接他们的输入和输出channel，并避免通过代理交换消息的额外成本，截止1.0版本，聚合只支持以下类型的应用程序：

- sources：带有名为output的单一输出channel的应用。典型情况下，该应用带有包含一个以下类型的绑定

```
org.springframework.cloud.stream.messaging.Source
```

- sinks：带有名为input的单一输入channel的应用。典型情况下，该应用带有包含一个以下类型的绑定

```
org.springframework.cloud.stream.messaging.Sink
```

- processors：带有名为input的单一输入channel和带有名为output的单一输出channel的应用。典型情况下，该应用带有包含一个以下类型的绑定

```
org.springframework.cloud.stream.messaging.Processor
```

可以通过创建一个相互关联的应用的序列将他们聚合在一起，其中一个序列元素的输出通道连接到下一个其中一个元素的输出通道连接到下一个元素的输入通道元素的输入通道，序列可以由一个source或者一个processor开始，可以包含任意数目的processors，且必须由processors或者sink结束。

根据开始和结束元素的特性，序列可以有一个或者多个可绑定的channels，如下：

- 如果序列由source开始，sink结束，应用之间直接通信并且不会绑定通道
- 如果序列由processor开始，它的输入通道会变成聚合的input通道并进行相应的绑定
- 如果序列由processor结束，它的输出通道会变成聚合的output通道并进行相应的绑定

使用AggregateApplicationBuilder功能类来实现聚合，如下例子所示。考虑一个包含source, processor和sink的工程，它们可以示包含在工程中，或者包含在工程的依赖中。

```

1. @SpringBootApplication
2. @EnableBinding(Sink.class)
3. public class SinkApplication {
4.
5.     private static Logger logger = LoggerFactory.getLogger(SinkModuleDefinition.class);
6.
7.     @ServiceActivator(inputChannel=Sink.INPUT)
8.     public void loggerSink(Object payload) {

```

```

9.         logger.info("Received: " + payload);
10.     }
11. }

```

```

1. @SpringBootApplication
2. @EnableBinding(Processor.class)
3. public class ProcessorApplication {
4.
5.     @Transformer
6.     public String loggerSink(String payload) {
7.         return payload.toUpperCase();
8.     }
9. }

```

```

1. @SpringBootApplication
2. @EnableBinding(Source.class)
3. public class SourceApplication {
4.
5.     @Bean
6.     @InboundChannelAdapter(value = Source.OUTPUT)
7.     public String timerMessageSource() {
8.         return new SimpleDateFormat().format(new Date());
9.     }
10. }

```

每一个配置可用于运行一个独立的组件，在这个例子中，它们可以这样实现聚合：

```

1. @SpringBootApplication
2. public class SampleAggregateApplication {
3.
4.     public static void main(String[] args) {
5.         new AggregateApplicationBuilder()
6.             .from(SourceApplication.class).args("--fixedDelay=5000")
7.             .via(ProcessorApplication.class)
8.             .to(SinkApplication.class).args("--debug=true").run(args);
9.     }
10. }

```

序列的开始组件作为from()方法的参数，序列的结束组件作为to()方法的参数，中间处理器作为via()方法的参数，同一类型的处理器可以链在一起（例如，可以使用不同配置的管道传输方式）。对于每一个组件，编译器可以为Spring Boot提供运行时参数。

RxJava 支持

RxJava 是一个响应式编程框架，SpringCloudStream通过 `RxJavaProcessor` 可以支持RxJava的 processor，参见 `spring-cloud-stream-rxjava`

```
1. public interface RxJavaProcessor<I, O> {
2.     Observable<O> process(Observable<I> input);
3. }
```

`RxJavaProcessor`（观察者设计模式）收到观察得到的对象`Observable`作为输入，相当于数据流的输入装载器。在启动时调用`process`方法来设置数据流。

用`@EnableRxJavaProcessor`修饰在你的处理方法上，就可以启用基于RxJava的处理
器。`@EnableRxJavaProcessor`包含了`@EnableBinding(Processor.class)`注解并可以创建
`Processor`，如下：

```
1. @EnableRxJavaProcessor
2. public class RxJavaTransformer {
3.
4.     private static Logger logger = LoggerFactory.getLogger(RxJavaTransformer.class);
5.
6.     @Bean
7.     public RxJavaProcessor<String,String> processor() {
8.         return inputStream -> inputStream.map(data -> {
9.             logger.info("Got data = " + data);
10.            return data;
11.        })
12.        .buffer(5)
13.        .map(data -> String.valueOf(avg(data)));
14.    }
15.
16.    private static Double avg(List<String> data) {
17.        double sum = 0;
18.        double count = 0;
19.        for(String d : data) {
20.            count++;
21.            sum += Double.valueOf(d);
22.        }
23.        return sum/count;
24.    }
25. }
```

实施RxJava处理器，处理流程中的异常特别重要，未捕获的异常将被视为`errors`，并会结束`Observable`，中断了处理流程。

绑定器

SpringCloudStream提供绑定抽象用于与外部中间件中的物理目标进行连接。本章主要介绍Binder SPI背后的主要概念，主要组件以及实现细节。

生产者与消费者



任何往通道中发布消息的组件都可称作生产者。通道可以通过代理的Binder实现与外部消息代理进行绑定。调用`bindProducer()`方法，第一个参数是代理名称，第二个参数是本地通道目标名称（生产者向本地通道发送消息），第三个参数包含通道创建的适配器的属性信息（比如：分片key表达式）。

任何从通道中接收消息的组件都可称作消费者。与生产者一样，消费者通道可以与外部消息代理进行绑定。调用`bindConsumer()`方法，第一个参数是目标名称，第二个参数提供了消费者组的名称。每个组都会收到生产中发出消息的副本（即，发布-订阅语义），如果有多个消费者绑定相同的组名称，消息只会由一个消费者消费（即，队列语义）

Binder SPI

Binder Detection

Classpath Detection

Multiple Binders on the Classpath

Connecting to Multiple Systems

Binder configuration properties

Implementation strategies

Kafka Binder

RabbitMQ Binder

配置管理

SpringCloudStream 支持通用的配置以及bindings和binders的配置，一些binders允许

binding属性用来支持中间件的特定功能。

SpringCloudStream配置项

spring.cloud.stream.instanceCount

应用程序的部署实例的数量。如果使用卡夫卡则会设置分区。

Default: 1

spring.cloud.stream.instanceIndex

应用程序的部署实例的数量，大小介于0 ~ (instanceCount-1)，用于kafka寻找分区。在Cloud Foundry中会自动设置

Default: 1

spring.cloud.stream.dynamicDestinations

A list of destinations that can be bound dynamically (for example, in a dynamic routing scenario). If set, only listed destinations can be bound.

Default: empty

spring.cloud.stream.defaultBinder

The default binder to use, if multiple binders are configured

Default: empty

spring.cloud.stream.overrideCloudConnectors

This property is only applicable when the cloud profile is active and Spring Cloud Connectors are provided with the application. If the property is false (the default), the binder will detect a suitable bound service (e.g. a RabbitMQ service bound in Cloud Foundry for the RabbitMQ binder) and will use it for creating connections (usually via Spring Cloud Connectors). When set to true, this property instructs binders to completely ignore the bound services and rely on Spring Boot properties (e.g. relying on the spring.rabbitmq.* properties provided in the environment for the RabbitMQ binder). The typical usage of this property is to be nested in a customized environment when connecting to multiple systems.

Default: false

Binding配置项

配置格式为 `spring.cloud.stream.bindings.<channelName>.<property>=<value>` , `<channelName>` 是配置的频道名称 (e.g., output for a Source), 下面的介绍中省略 `spring.cloud.stream.bindings.<channelName>.` 前缀, 只关注属性参数

SpringCloudStream的bindings配置

下面的配置对于input bindings和output bindings都有效, 且前缀是 `spring.cloud.stream.bindings.<channelName>.`

destination

绑定中间件的目的 (e.g., the RabbitMQ exchange or Kafka topic)。如果channel绑定的是消费者, 那么可以绑定多个目的, 用逗号分隔。如果不设置则channel名称会替代这个值。

group

channel的消费者组, 仅对inbound bindings有效。

Default: null (暗示一个匿名消费者)

contentType

The content type of the channel.

Default: null (so that no type coercion is performed).

binder

The binder used by this binding. See Multiple Binders on the Classpath for details.

Default: null (the default binder will be used, if one exists).

Consumer properties

下面的配置仅对input bindings有效, 且前缀是 `spring.cloud.stream.bindings.<channelName>.consumer.`

concurrency

The concurrency of the inbound consumer.

Default: 1

partitioned

Whether the consumer receives data from a partitioned producer.

Default: false

headerMode

When set to raw, disables header parsing on input. Effective only for messaging middleware that does not support message headers natively and requires header embedding. Useful when inbound data is coming from outside Spring Cloud Stream applications.

Default: embeddedHeaders.

maxAttempts

The number of attempts of re-processing an inbound message.

Default: 3.

backOffInitialInterval

The backoff initial interval on retry.

Default: 1000.

backOffMaxInterval

The maximum backoff interval.

Default: 10000.

backOffMultiplier

The backoff multiplier.

Default: 2.0.

Producer Properties

下面的配置仅对output bindings有效，且前缀是 `spring.cloud.stream.bindings.`
`<channelName>.producer.`

partitionKeyExpression

A SpEL expression that determines how to partition outbound data. If set, or if `partitionKeyExtractorClass` is set, outbound data on this channel will be partitioned, and `partitionCount` must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See

Partitioning Support.

Default: null.

partitionKeyExtractorClass

A PartitionKeyExtractorStrategy implementation. If set, or if partitionKeyExpression is set, outbound data on this channel will be partitioned, and partitionCount must be set to a value greater than 1 to be effective. The two options are mutually exclusive. See Partitioning Support.

Default: null.

partitionSelectorClass

A PartitionSelectorStrategy implementation. Mutually exclusive with partitionSelectorExpression. If neither is set, the partition will be selected as the hashCode(key) % partitionCount, where key is computed via either partitionKeyExpression or partitionKeyExtractorClass.

Default: null.

partitionSelectorExpression

A SpEL expression for customizing partition selection. Mutually exclusive with partitionSelectorClass. If neither is set, the partition will be selected as the hashCode(key) % partitionCount, where key is computed via either partitionKeyExpression or partitionKeyExtractorClass.

Default: null.

partitionCount

The number of target partitions for the data, if partitioning is enabled. Must be set to a value greater than 1 if the producer is partitioned. On Kafka, interpreted as a hint; the larger of this and the partition count of the target topic is used instead.

Default: 1.

requiredGroups

A comma-separated list of groups to which the producer must ensure message delivery even if they start after it has been created (e.g., by pre-creating durable queues in RabbitMQ).

headerMode

When set to raw, disables header embedding on output. Effective only for messaging middleware that does not support message headers natively and requires header embedding. Useful when producing data for non-Spring Cloud Stream applications.

Default: embeddedHeaders.

Binder-Specific Configuration

Rabbit-Specific Settings

RabbitMQ Binder Properties

RabbitMQ Consumer Properties

Rabbit Producer Properties

Kafka-Specific Settings

Kafka Binder Properties

Kafka Consumer Properties

Kafka Producer Properties

Content Type and Transformation

MIME types

MIME types and Java types

[@StreamListener](#) and Message Conversion

应用程序间通信

连接多个应用程序实例

SpringCloudStream使SpringBoot应用连接消息系统变得容易，典型情况是多应用管道的创作，微服务通过这个管道彼此发送数据。

为了实现TimeSource应用的数据发送给LogSink应用，你可以通过配置相同的目的地名字来绑定他们。

TimeSource的配置如下

```
spring.cloud.stream.bindings.output.destination=ticktock
```

LogSink的配置如下

```
spring.cloud.stream.bindings.input.destination=ticktock
```

实例索引和实例数

当水平扩展SpringCloudStream应用时，每个实例都能收到消息，这个消息是关于本应用运行的实例数量和每个实例自己的索引值。利

用 `spring.cloud.stream.instanceCount` 和 `spring.cloud.stream.instanceIndex` 就能做到上面的所述的功能。例如：如果有三个HDFS的sink application，这三个实例都设置了 `spring.cloud.stream.instanceCount=3`，并且又分别设置了 `spring.cloud.stream.instanceIndex` 的值为0, 1, 2。

当SpringCloudStream应用通过SpringCloudDataFlow部署，这些参数会自动配置。如果是独立部署，那这些参数必须被正确配置。默认情况

下， `spring.cloud.stream.instanceCount=1`， `spring.cloud.stream.instanceIndex=0`

水平扩展扩展的案例中，正确的配置这两个参数对于访问分区的行为十分重要，并且这两个参数需要确定的binders(e.g., the Kafka binder)，上述是为了保证数据能被正确分配在多个消费端实例。

分区

配置Output Bindings

output binding的配置是用于发送分区数据，配

置 `partitionKeyExpression` 或 `partitionKeyExtractorClass` 以及 `partitionCount`。例如，下面是一个有效的和典型的配置：

```
1. spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload.id
2. spring.cloud.stream.bindings.output.producer.partitionCount=5
```

基于上述的配置，数据将被用下述逻辑发送到目标分区。

分区key的值是基于partitionKeyExpression计算得出的，用于每个消息被发送至分区的输出channel，partitionKeyExpression是spring EL表达式用以提取分区键。

如果SpEL不能满足你的需求，你可以通过 `partitionKeyExtractorClass` 设置一个自定义的类去计算分区的key值，这个类需要实现 `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` 接口，通常情况下SpEL是够用的，更复杂的情况才会用到自定义的策略。

一旦消息的key被算出，分区选择器将会确定目标分区值，这个值介于0 和 `partitionCount - 1` 之间，默认的算法，在大多数情况下适用，是基于公式的 `key.hashCode() % partitioncount`。

额外的属性可以被配置为更高级的情况，如下面的章节所述。

Kafka binder使用 `partitionCount` 做创建topic的线索利用给定的分区数（这个数是 `partitionCount` 与 `minPartitionCount` 的最大值）。当为binder配置 `minPartitionCount`，为应用配置 `partitionCount` 的时候你要小心，两者较大的值将会被使用。如果一个topic已经存在与小分区数的kafka中，并且 `autoAddPartitions` 是被禁用的（默认如此），那么binder将启动失败，如果 `autoAddPartitions` 是启用的则会自动添加新分区。如果topic已经存于大分区数的kafka（比 `minPartitionCount` 和 `partitionCount` 的值都大），这个存在的分区将会被使用。

配置Input Bindings

通过配置分区属性来接收分区中的数据，如下面的示例：

1. `spring.cloud.stream.bindings.input.consumer.partitioned=true`
2. `spring.cloud.stream.instanceIndex=3`
3. `spring.cloud.stream.instanceCount=5`

`instanceCount` 表示应用实例的总数，`instanceIndex` 在多个实例中必须唯一，并介于0~（`instanceCount-1`）之间。实例的索引可以帮助每个实例确定唯一的接收数据的分区，正确的设置这两个值十分重要，用来确保所有的数据被消耗，以及应用实例接收相互排斥的数据集。

使用多实例进行分区数据处理是一个复杂设置，SpringCloudDataFlow可以显著的简化过程，通过正确的填写输入和输出值，以及信任运行时提供的instance索引和instance数量信息

Testing

健康指示器

SpringCloudStream提供binders健康指示器，他以binders名字注册，可以由 `management.health.binders.enabled` 开控制启动或停止

例子

[spring-cloud-stream-samples](#)

Getting Started

Spring Cloud zuul

- [Zuul: 路由器和过滤器](#)
 - [如何引入Zuul](#)
 - [内置Zuul 反向代理](#)
 - [Zuul Http Client](#)
 - [Cookies and Sensitive Headers](#)
 - [忽略Headers](#)
 - [The Routes 访问点](#)
 - [抑制模式和本地转发](#)
 - [通过Zuul上传文件](#)
 - [Plain Embedded Zuul](#)
 - [失效 Zuul 过滤器](#)
 - [Providing Hystrix Fallbacks For Routes](#)

贡献者: Alex Wei

Zuul: 路由器和过滤器

路由是微服务体系不可或缺的一部分，例如 `/` 可以映射到Web 应用，`/api/users` 可以映射的用户服务，`api/shop` 可以映射到商店服务。Zuul是一个基于JVM的路由器，同时也是一个服务端负载均衡。

它由Netflix公司开源。

Netflix使用Zuul做如下事情：

- 认证鉴权。
- 审查
- 压力测试
- 金丝雀测试
- 动态路由
- 服务迁移
- 负载剪裁
- 安全
- 静态应答处理

Zuul允许使用任何支持JVM的语言来建立规则和过滤器，内置了对Java和Groovy的支持。

如何引入Zuul

为了在自己的project中引入zuul，仅仅需要使用spring-cloud-starter-zuul就OK啦。详细的

内容参照 [Spring Cloud Project page](#) 。

内置Zuul 反向代理

当一个UI应用想要通过代理调用后端服务时，Spring Cloud通过一个内置的Zuul代理，从而减轻一些通用案例的开发。这个特性对于前端用户交互通过代理访问后端服务是有用的，避免了为所有后端服务单独建立CORS和认证相关管理等事情。在Spring Boot的main类上面添加`@EnableZuulProxy`注解，它就可以转发本地的调用到适用的服务上面。

按照惯例，一个服务ID为“users”的服务也将收到来自于/users的请求。代理使用Ribbon（一种客户端负载均衡机制）定位一个服务实例，所有迭代请求在hystrix command（服务异常断路机制）被执行，以致于一旦失败，Hystrix metrics（服务异常断路的监控机制）将能够呈现出来。一旦断路器被打开，代理将不再尝试与这个服务联系。

为了跳过一些自动增加的服务，可以设置zuul.ignored-services的值，使其符合想要忽略的服务列表的ID模式。如果一个服务匹配这种忽略的模式，但是被显示的配置到了Routes map里的话，它又不能被忽略。例如：

```
1. application.yml
2. zuul:
3.   ignoredServices: '*'
4.   routes:
5.     users: /myusers/**
```

在这个例子里，除了“users”，所有的服务都会被忽略。

为了扩充或者改变代理路由，你可以增加如下显示的配置：

```
1. application.yml
2. zuul:
3.   routes:
4.     users: /myusers/**
```

这意味着来自于 `"/myusers"` 的http请求将被转发到 `"users"` 服务上（例如 `"/myusers/101"` 被转发到 `"/101"`）。

为了得到更细粒度的控制，你可以指定具体的路由到服务的标识上：

```
1. zuul:
2.   routes:
3.     users:
4.       path: /myusers/**
5.       serviceId: users_service
```


这意味着来自于 `"/myusers"` 的http请求被转发到 `"users_service"` 这个服务上。路由必须有一个符合Ant模式的“path”，那么 `"/myusers/**"` 就仅仅匹配第一级，而 `"/myusers/**"` 能够层次化匹配。

后端服务的定位可以按照服务ID或者url来识别。 例如：

```
1. application.yml
2. zuul:
3.   routes:
4.     users:
5.       path: /myusers/**
6.       url: http://example.com/users_service
```

这些简单的url路由是不支持HystrixCommand 和Ribbon负载均衡的。为了实现这一点，需要指定service路由，并且为这个Service配置Ribbon客户端(当前需要在Ribbon失效Eureka的支持)。例如：

```
1. application.yml
2. zuul:
3.   routes:
4.     users:
5.       path: /myusers/**
6.       serviceId: users
7. ribbon:
8.   eureka:
9.     enabled: false
10. users:
11.   ribbon:
12.     listOfServers: example.com,google.com
```

你可以使用正则匹配来建立ServiceId和路由之间的默契。使用正则表达式命名组来从服务ID中提取变量，注入他们到一个路由模式里。

```
1. ApplicationConfiguration.java
2. @Bean
3. public PatternServiceRouteMapper serviceRouteMapper() {
4.     return new PatternServiceRouteMapper(
5.         "(?<name>^.+)-(?<version>v.+)$",
6.         "${version}/${name}");
7. }
```

这意味着，一个服务ID为“myusers-v1”将被映射到路由 `"/v1/myusers/**"` 上。任何正则表达式都会被接受，但是所有的名称组必须同时出现在servicePattern和routePattern上。如果servicePattern不匹配Service Id，将使用默认行为。在上面的例子中， 服务ID

是“myusers”的服务会被映射到路由：`"/myusers/**"`（不检测版本）。这个特性默认是不可用的，并且仅仅适用于已经被发现（应该是已经注册）的服务。

为了给所有的映射增加前缀，可以设置`zuul.prefix`，例如`/api`。默认情况下，在请求被转发前，这个代理前缀会被从请求中去除掉。你也可以按照独立的路由来控制这个功能的切换，例如：

```
1. application.yml
2.  zuul:
3.    routes:
4.      users:
5.        path: /myusers/**
6.        stripPrefix: false
```

这个例子中，请求 `"/myusers/101"` 将被转发到 `"users"` 服务的 `"/myusers/101"` 上。

`Zuul.routes`条目实际上是绑定到一个类型为`ZuulProperties`的对象上。如果查找这个对象的属性集，你会发现有一个“retryable”的标志位。将这个标志位设置为`true`，Ribbon的客户端将自动重试失败的请求。（如果需要，可以使用Ribbon客户端配置更改这个操作的参数）。

默认情况下，`X-Forwarded-Host` header会被增加到转发的请求里。 如果不要这个功能，需要设置 `zuul.addProxyHeaders = false`。

一个带有`@EnableZuulProxy`的应用能够作为独立服务器。如果设置一个默认的路由（`"/"`），例如 `examplezuul.route.home: /` 将路由所有的请求到“home” service。

如果需要更细粒度的控制一些路由模式的忽略，可以指定一个特殊的忽略模式。 这些模式在路由定位过程的开始被计算。这也意味着前缀应该被包含在模式里来保证匹配。忽略模式跨越所有服务和取代所有其他路由规范。

```
1. application.yml
2. zuul:
3.   ignoredPatterns: /**/admin/**
4.   routes:
5.     users: /myusers/**
```

这意味着所有的类似 `"/myusers/101"` 的请求将被转发到 `"users"` 服务的 `"/101"` 上。但包含 `"/admin/"` 将不被解析。

注意： 如果你需要你的路由配置有顺序，需要使用YAML文件，`properties file`将流失预订的顺序。

```
1. application.yml
2. zuul:
3.   routes:
4.     users:
```

```

5.     path: /myusers/**
6.     legacy:
7.     path: /**

```

如果使用 `properties file`，`legacy`路径可能会跑到`users`路径前面，使得`users`路径不可达。

Zuul Http Client

zuul默认使用的HTTP Client是Apache HTTP Client，代替了过时的Ribbon RestClient。如果想使用 RestClient或者okhttp3.OkHttpClient，设置 `ribbon.restclient.enabled=true`或者`ribbon.okhttp.enabled=true`。

Cookies and Sensitive Headers

同一系统在服务间共享headers是可以做到的。但是，你或许不想一些敏感的headers向下泄露到一个外部服务。你可以在路由配置里指定一个忽略的headers列表。Cookies在浏览器端有明确的语义，所以作为一个特殊的角色，总是被视为很敏感。如果客户端是浏览器，因为cookies的混杂，也会给使用下游服务的用户造成问题。

如果你对服务的设计很小心，例如如果仅仅一个下游服务设置了cookies，那么你也可能让他们一直从后端流窜到调用端。同时，如果你的代理设置了cookies，并且所有后端服务是同一系统的一部分。它可以很自然的分享他们（类似于使用Spring Session来link他们到一些共享状态）。另外，任何由下游服务设置/获取的cookies，对于调用者来说，不一定特别有用。因为，推荐你为路由“Set-Cookie”和设置cookies到header时，不要作为domain的一部分。即使路由是domain的一部分，也要尝试认真思考是否允许cookies在服务 and 代理间流动。

敏感性headers能够在每个route里用逗号分隔进行配置。

```

1. application.yml
2.   zuul:
3.     routes:
4.     users:
5.       path: /myusers/**
6.       sensitiveHeaders: Cookie,Set-Cookie,Authorization
7.       url: https://downstream

```

敏感性的headers也能用 `zuul.sensitiveHeaders`进行全局设置。如果route上设置了 `sensitiveHeaders`，将覆盖全局设置。

忽略Headers

除了route上的敏感性headers之外，在与下游服务交互期间，你可以设置zuul.ignoredHeaders来忽略headers（请求和应答）。默认，这个属性是空的。除非Spring Security不在classpath，否则他们将初始化一组由Spring Security指定的 "security" headers。这个假设是，下游服务也可能增加headers。如果当Spring Security不在classpath，我们不想废弃那些security headers，可以设置zuul.ignoreSecurityHeaders为false。如果你不激活Spring Security的Security response headers或者希望下游服务提供这些值，这可能是有用的。

The Routes 访问点

如果你使用 `@EnableZuulProxy` 和Spring Boot Actuator，你将能得到一个/routes的访问点。GET这个访问点将返回映射的route列表。POST这个访问点将强制刷新存在的routes。

抑制模式和本地转发

当迁移一个存在的应用或者API时，有一个通用的模式是“抑制”那个旧的访问点，慢慢的用不同的实现替换他们。Zuul代理就是一个有用的工具。你可以使用它重定向网络访问到新的访问点。例如如下配置：

```
1. application.yml
2.   zuul:
3.     routes:
4.       first:
5.         path: /first/**
6.         url: http://first.example.com
7.       second:
8.         path: /second/**
9.         url: forward:/second
10.      third:
11.        path: /third/**
12.        url: forward:/3rd
13.      legacy:
14.        path: /**
15.        url: http://legacy.example.com
```

这个例子中，路径是/first/的请求被提取到一个带有外部url的新的服务。路径/second/和/third/**被转发到本地服务来处理。而剩下的不符合以上模式的请求才被“legacy”处理。

通过Zuu1上传文件

使用Zuu1（`@EnableZuulProxy`），可以使用代理路径上传文件，但这仅仅对于尽可能小的文件。

对于大的文件，有一个叫做 `"/zuul/**"` 的可替代的路径可以绕过Spring DispatcherServlet（为了避免多重处理）。例如，如果设置 `zuul.routes.customers=/customers/**`，你可以POST一个大的文件给 `"/zuul/customers/**"`。这个Servlet路径经由zuul.servletPath被暴露。极端情况下，如果代理的route利用的是Ribbon负载均衡，大的文件也需要提高超时的设置。

例如：

```
1. application.yml
2.   hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
3.   ribbon:
4.     ConnectTimeout: 3000
5.     ReadTimeout: 60000
```

注意，对于流式处理大的文件，需要在请求中使用分块编码（一些浏览器默认不这样做）。例如，使用命令行：

```
1. $ curl -v -H "Transfer-Encoding: chunked" \
2.   -F "file=@mylarge.iso" localhost:9999/zuul/simple/file
```

Plain Embedded Zuul

也可以运行一个没有代理的Zuul服务器，或者选择性的开启一部分代理平台。如果使用 `@EnableZuulServer`（不是 `@EnableZuulProxy`），任何增加到应用里的，扩展自ZuulFilter的beans都将自动被安装。但如果他们被标注 `@EnableZuulProxy`，没有任何代理过滤器被自动安装。

这种情况下，zuul服务器里的routes仍然通过 `"zuul.routes.*"` 来指定，但是没有服务发现，也没有代理。因此，`"serviceId"`和`"url"`设置都被忽略。例如：

```
1. application.yml
2.   zuul:
3.     routes:
4.       api: /api/**
```

将映射所有的 `"/api/**"` 到Zuul过滤器链。

失效 Zuul 过滤器

Zuul默认包含了大量的ZuulFilter Beans用于代理或者服务模式。可以参照filters包来查找有哪些可用的过滤器。如果你想失效一个，简单的设置 `zuul.<SimpleClassName>.<filterType>.disable=true` 就可以。依照惯例，包中的过滤器都是Zuul的顾虑器类型。例如，为了失效 `org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter`，设置

`zuul.SendResponseFilter.post.disable=true` 就可以啦。

Providing Hystrix Fallbacks For Routes

当Zuul中一个给定route的断路器跳闸时，可以通过建立一个 `ZuulFallbackProvider` 的扩展Bean来提供一个fallback应答。在这个Bean里，你需要指定Route ID，并提供一个 `ClientHttpResponse` 作为Fallback返回。下面是一个非常简单的 `ZuulFallbackProvider` 实现。

```
class MyFallbackProvider implements ZuulFallbackProvider {
    @Override
    public String getRoute() {
        return "customers";
    }
}
```

```
1. @Override
2. public ClientHttpResponse fallbackResponse() {
3.     return new ClientHttpResponse() {
4.         @Override
5.         public HttpStatus getStatusCode() throws IOException {
6.             return HttpStatus.OK;
7.         }
8.
9.         @Override
10.        public int getRawStatusCode() throws IOException {
11.            return 200;
12.        }
13.
14.        @Override
15.        public String getStatusText() throws IOException {
16.            return "OK";
17.        }
18.
19.        @Override
20.        public void close() {
21.
22.        }
23.
24.        @Override
25.        public InputStream getBody() throws IOException {
26.            return new ByteArrayInputStream("fallback".getBytes());
27.        }
28.
29.        @Override
30.        public HttpHeaders getHeaders() {
```

```
31.         HttpHeaders headers = new HttpHeaders();
32.         headers.setContentType(MediaType.APPLICATION_JSON);
33.         return headers;
34.     }
35. };
36. }
```

```
}
```

这时，route的配置看起来像：

```
1. zuul:
2.   routes:
3.     customers: /customers/**
```

Spring Cloud ribbon

- 客户端负载均衡Ribbon
 - 怎么引入Ribbon
 - 定制化Ribbon Client
 - 使用properties定制化Ribbon Client
 - 一起使用Ribbon和Eureka
 - Example: 没有Eureka, 如何使用Ribbon
- Example: 在Ribbon停止Eureka可用
 - 直接使用Ribbon API

客户端负载均衡Ribbon

Ribbon是一个客户端负载均衡器，能够给HTTP和TCP客户端带来灵活的控制。 Feign已经使用了Ribbon，因此如果你正在使用@FeignClient，那么本部分的说明同样试用。

Ribbon的核心概念是命名的客户端。每一个负载均衡器都是一系列工作在一起的组件的一部分，并用于按需联系远程服务。你可以给这一系列一个名字（例如使用@FeignClient注解）。Spring Cloud使用RibbonClientConfiguration为每一个命名的客户端建立一个新系列为满足ApplicationContext的需求。 这包括一个ILoadBalancer， 一个RestClient和一个ServerListFilter。

怎么引入Ribbon

为了引入Ribbon到你的工程，可以使用org.springframework.cloud组的starter.artifact id是spring-cloud-starter-ribbon。可以参照Spring Cloud Project的细节来设置你的构建系统。

定制化Ribbon Client

你可以使用外部的属性 `<client>.ribbon.*` 来配置一些Ribbon Client，除了能够使用Spring Boot的配置文件以外，它和使用原生的Netflix APIS没什么不同。原生选项可以在CommonClientConfigKey中被检查到。

通过使用@RibbonClient声明额外的配置（在RibbonClientConfiguration上面），Spring Cloud也让你全面的控制Client。 例如：

```
1. @Configuration
2. @RibbonClient(name = "foo", configuration = FooConfiguration.class)
```



```

3.     public class TestConfiguration {
4.     }

```

这个例子中，Client除了包括了已经存在于RibbonClientConfiguration的组件，也包括了自定义的FooConfiguration（一般后者会重载前者）。

警告: *FooConfiguration* 不能被@ComponentScan 在main application context。这样的话，它将被所有@RibbonClients共享。如果你使用 @ComponentScan (or @SpringBootApplication)，你需要避免它被包括其中。(例如：放它到一个独立的，无重叠的包里，或者指明不被@ComponentScan扫描)。

Spring Cloud Netflix为ribbon提供了如下的Beans(BeanType beanName: ClassName):

- IClientConfig ribbonClientConfig: DefaultClientConfigImpl
- IRule ribbonRule: ZoneAvoidanceRule
- IPing ribbonPing: NoOpPing
- ServerList ribbonServerList: ConfigurationBasedServerList
- ServerListFilter ribbonServerListFilter: ZonePreferenceServerListFilter
- ILoadBalancer ribbonLoadBalancer: ZoneAwareLoadBalancer

建立一个如上类型的bean，放置到@RibbonClient配置（如上FooConfiguration），允许你重载其中的任意一个。例如：

```

1. @Configuration
2. public class FooConfiguration {
3.     @Bean
4.     public IPing ribbonPing(IClientConfig config) {
5.         return new PingUrl();
6.     }
7. }

```

如上将用PingUrl代替NoOpPing。

使用properties定制化Ribbon Client

从1.2.0开始，Spring Cloud Netflix支持使用properties来定制化Ribbon clients。这就允许你在不同的环境，在启动时改变举止。

支持的属性如下（加上.ribbon.的前缀）：

- NFLoadBalancerClassName：实现ILoadBalancer
- NFLoadBalancerRuleClassName：实现IRule
- NFLoadBalancerPingClassName：实现IPing
- NIWSServerListClassName：实现ServerList
- NIWSServerListFilterClassName：实现ServerListFilter

在这些属性中定义的Classes优先于在`@RibbonClient(configuration=MyRibbonConfig.class)`定义的beans和由Spring Cloud Netflix提供的缺省值。

例如，为一个叫user的服务设置IRule，可以进行如下设置：

```
1. users:
2.   ribbon:
3.     NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule
```

一起使用Ribbon和Eureka

当Eureka被和Ribbon一起联合使用时，`ribbonServerList`被一个叫做`DiscoveryEnabledNIWSServerList`的扩展重载。这个扩展汇聚了来自于Eureka的服务列表。`IPing`的接口也被`NIWSDiscoveryPing`代替，用于委托Eureka来测定服务是否UP。缺省的`ServerListThe`是一个`DomainExtractingServerList`，目的是使得物理上的元数据可以用于负载均衡器，而不必要使用AWA AMI的元数据。缺省情况下，服务List将根据“zone”信息被创建。（远程客户端设置`eureka.instance.metadataMap.zone`）。如果这个没有设置，它可以使用来自于服务hostname的domain name作为zone的代理（取决于`approximateZoneFromHostname`是否被设置）。一旦那个zone的信息可获得，他可以在`ServerListFilter`被使用。缺省情况下，由于使用`ZonePreferenceServerListFilter`，它被用于定位一个和Client相同zone的server。默认情况下，client的zone是和远程实例一样的，通过`eureka.instance.metadataMap.zone`来设置。

注意：正统的“archaius”方式是通过配置属性“`@zone`”来设置Client zone。如果它是有效的，Spring Cloud使用这个优先于其他设置。（注意，这个key必须在YAML configuration被引）。

如果zone数据没有其他来源，基于Client的配置将会有个猜测。（as opposed to the instance configuration）。我们称为`eureka.client.availabilityZones`。它是一个从region名到zones列表的map，并且抽出第一个zone 作为实例自己的region（例如the `eureka.client.region`）

Example：没有Eureka，如何使用Ribbon

Eureka是一个实用的方式，它抽象了远程服务的发现机制，可以使你不用在Client中硬编码URLs。但如果你不打算使用它，Ribbon和Feign仍然是相当可用的。假设你已经声明了一个`@RibbonClient`为一个“Stores”的服务。Eureka没有被使用。Ribbon Client会缺省到一个已配置的Server列表。你可以提供配置像：

```
1. stores:
2.   ribbon:
3.     listOfServers: example.com,google.com
```

Example：在Ribbon停止Eureka可用

设置 `ribbon.eureka.enabled = false` ，可以在Ribbon中停止对Eureka的使用

```
1. ribbon:
2.   eureka:
3.     enabled: false
```

直接使用Ribbon API

可以直接使用LoadBalancerClient。例如：

```
1. public class MyClass {
2.     @Autowired
3.     private LoadBalancerClient loadBalancer;
4.
5.     public void doStuff() {
6.         ServiceInstance instance = loadBalancer.choose("stores");
7.         URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(),
8. instance.getPort()));
9.         // ... do something with the URI
10.    }
```

源码分析

- [Spring Cloud源码分析系列](#)
 - [Spring Cloud Eureka源码分析](#)
 - [Spring Cloud Eureka服务续约\(Renew\)源码分析](#)
 - [Spring Cloud Eureka服务注册源码分析](#)
 - [Spring Cloud中@EnableEurekaClient源码分析](#)
 - [Spring Cloud Eureka服务下线\(Cancel\)源码分析](#)

Spring Cloud源码分析系列

Spring Cloud Eureka源码分析

[Spring Cloud Eureka服务续约\(Renew\)源码分析](#)

[Spring Cloud Eureka服务注册源码分析](#)

[Spring Cloud中@EnableEurekaClient源码分析](#)

[Spring Cloud Eureka服务下线\(Cancel\)源码分析](#)
