1.Spring Boot&Spring Cloud&Docker

**1. SpringBoot基本应用**

**1.1 约定优于配置**

Spring Boot是所有基于Spring开发的项目的起点。Spring Boot的设计是为了让你尽可能快的跑起来Spring应用程序并且尽可能减少你的配置文件。

约定优于配置(Convention over Configuration)，又称按约定编程，是一种软件设计范式。

本质上是说，系统、类库或框架应该假定合理的默认值，而非要求提供不必要的配置。比如说模型中有一个名为User的类，那么数据库中对应的表就会默认命名为user。只有在偏离这一个约定的时候，例如想要将该表命名为person，才需要写有关这个名字的配置。

约定优于配置简单来理解，就是遵循约定

**1.2 SpringBoot概念**

1.2.1 Spring优缺点分析

**优点**：Spring是Java企业版(Java Enterprise Edition，JEE，也称J2EE)的轻量级代替品。无需开发重量级的Enterprise Java Bean(EJB)，Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的Java对象(Plain Old Java Object，POJO)实现了EJB的功能

**缺点**：1、Spring的组件代码是轻量级的，但它的配置却是重量级的。2、项目的依赖管理也是一件耗时耗力的事情

1.2.2 Spring Boot解决上述spring问题

SpringBoot对上述Spring的缺点进行的改善和优化，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑 业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短了项目周期。

**起步依赖**：起步依赖本质上是一个Maven项目对象模型(Project Object Model，POM)，定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。简单的说，起步依赖就是将具备某种功能的依赖坐标打包到一起，并提供一些默认的功能。

**自动配置：**springboot的自动配置，指的是springboot，会自动将一些配置类的bean注册进ioc容器，我们可以需要的地方使用@autowired或者@resource等注解来使用它。“自动”的表现形式就是我们只需要引我们想用功能的包，相关的配置我们完全不用管，springboot会自动注入这些配置bean，我们直接使用这些bean即可

springboot: 简单、快速、方便地搭建项目；对主流开发框架的无配置集成；极大提高了开发、部署效率

**1.3 Spring Boot案例**

所有的springBoot项目都会直接或者间接的继承spring-boot-starter-parent

1.指定项目的编码格式为UTF-8

2.指定JDK版本为1.8（可以修改）

3.对项目依赖的版本进行管理，当前项目再引入其他常用的依赖时就需要再指定版本号，避免版本冲突的问题

4.默认的资源过滤和插件管理

**@SpringBootApplication注解：**标识当前类为SpringBoot项目的启动类

SpringBoot的启动类通常放在二级包中，因为SpringBoot项目在做包扫描，会扫描启动类所在的包及其子包下的所有内容。

**1.4 单元测试与热部署**

1.4.1 单元测试

开发中，每当完成一个功能接口或业务方法的编写后，通常都会借助单元测试验证该功能是否正确。Spring Boot对项目的单元测试提供了很好的支持，在使用时，需要提前在项目的pom.xml文件中添加spring-boot-starter-test测试依赖启动器，可以通过相关注解实现单元测试

1.4.2 热部署

热部署：在修改完代码之后，不需要重新启动容器，就可以实现更新。

使用步骤：

1）添加SpringBoot的热部署依赖启动器：spring-boot-devtools

2）开启Idea的自动编译： Build下Compiler中，勾选“Build project automatically”

3）开启Idea的在项目运行中自动编译的功能，不同版本idea不同，网上查询

**1.5 全局配置文件**

全局配置文件能够对一些默认配置值进行修改。Spring Boot使用application.properties或者application.yaml的文件作为全局配置文件，该文件存放在src/main/resource目录或者类路径的/config，一般会选择resource目录。

Spring Boot配置文件的命名及其格式：application.properties；application.yaml；**application.yml（推荐）**

1.5.1 application.properties配置文件

使用Spring Initializr方式构建Spring Boot项目时，会在resource目录下自动生成一个空的application.properties文件，Spring Boot项目启动时会自动加载application.properties文件。我们可以在application.properties文件中定义Spring Boot项目的相关属性，当然，这些相关属性可以 是系统属性、环境变量、命令参数等信息，也可以是自定义配置文件名称和位置。配置格式：**key=value**

**@ConfigurationProperties(prefix = "person")**注解的作用是将配置文件中以person开头的属性值通过setXxx()方法注入到实体类对应属性中

@Component注解的作用是将当前注入属性值的Person类对象作为Bean组件放到Spring容器中，只有这样才能被@ConfigurationProperties注解进行赋值

中文乱码问题解决：

调整文件编码格式：Editor->File Encodings中所有的编码都改为UTF-8

设置Tomcat及Http编码(配置文件中):

#解决中文乱码

server.tomcat.uri-encoding=UTF-8

spring.http.encoding.force=true

spring.http.encoding.charset=UTF-8

spring.http.encoding.enabled=true

1.5.2 application.yaml配置文件

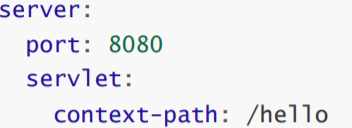
YAML文件格式是Spring Boot支持的一种JSON文件格式，相较于传统的Properties配置文件，YAML文件以数据为核心，是一种更为直观且容易被电脑识别的数据序列化格式。application.yaml配置文件的工作原理和application.properties是一样的，只不过yaml格式配置文件看起来更简洁一些。

**YAML文件的扩展名可以使用.yml或者.yaml。**

application.yml文件使用 “**key:（空格）value**”格式配置属性，使用缩进控制**层级关系**。

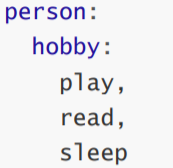
针对不同数据类型的属性值，介绍一下YAML：

（1）value值为普通数据类型(例如数字、字符串、布尔等)：当YAML配置文件中配置的属性值为普通数据类型时，可以直接配置对应的属性值，同时对于字符串类型的属性值，不需要额外添加引号



（2）value值为数组和单列集合：当YAML配置文件中配置的属性值为数组或单列集合类型时，主要有两种书写方式：**缩进式写法和行内式写法。**

其中，缩进式写法还有两种表示形式

上述代码中，在YAML配置文件中通过两种缩进式写法对person对象的单列集合（或数组）类型的爱好hobby赋值为play、read和sleep。其中一种形式为“**-（空格）属性值**”，另一种形式为多个属性值之前加**英文逗号分隔（注意，最后一个属性值后不要加逗号）。**

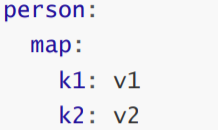
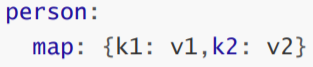
行内式写法



通过上述示例对比发现，YAML配置文件的行内式写法更加简明、方便。另外，包含属性值的中括号“[]”还可以进一步省略，在进行属性赋值时，程序会自动匹配和校对

（3）value值为Map集合和对象：当YAML配置文件中配置的属性值为Map集合或对象类型时，YAML配置文件格式同样可以分为两种书写方式：**缩进式写法和行内式写法**。

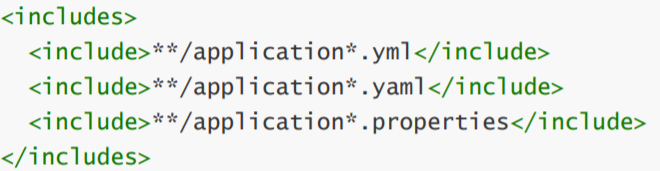
缩进式写法和对应的行内式写法

在YAML配置文件中，配置的属性值为Map集合或对象类型时，缩进式写法的形式按照YAML文件格式编写即可，而行内式写法的属性值要用大括号“{}”包含。

**1.6 配置文件属性值的注入**

配置文件的优先级如下： 从低到高



**SpringBoot的三种配置文件是可以共存的，**但有相同配置时，application.properties配置文件会覆盖application.yaml配置文件，所以application.properties优先级最高

如果配置属性是Spring Boot已有属性，例如服务端口server.port，那么Spring Boot内部会自动扫描并读取这些配置文件中的属性值并覆盖默认属性。

如果配置的属性是用户自定义属性，例如刚刚自定义的Person实体类属性，还必须在程序中注入这些配置属性方可生效。

Spring Boot支持多种注入配置文件属性的方式，使用注解@ConfigurationProperties和 @Value注入属性

1.6.1 使用@ConfigurationProperties注入属性

@Component和@ConfigurationProperties(prefix = “person”)注解共同快速、方便地将配置文件中的自定义属性值批量注入到某个Bean对象的多个对应属性中。

1.6.2 使用@Value注入属性

@Value注解是Spring框架提供的，用来读取配置文件中的属性值并逐个注入到Bean对象的对应属性中，Spring Boot框架从Spring框架中对@Value注解进行了默认继承，所以在Spring Boot框架中还可以使用该注解读取和注入配置文件属性值。

使用@Component和@Value注入Person实体类的id属性。其中，@Value不仅可以将配置文件的属性注入Person的id属性，还可以直接给id属性赋值，这点是@ConfigurationProperties不支持的

**1.7 自定义配置**

spring Boot免除了项目中大部分的手动配置，对于一些特定情况，我们可以通过修改全局配置文件以适应具体生产环境，可以说，几乎所有的配置都可以写在application.yml文件中，Spring Boot会自动加载全局配置文件从而免除我们手动加载的烦恼。但是，如果我们自定义配置文件，Spring Boot是无法识别这些配置文件的，此时就需要我们手动加载。

1.7.1 使用@PropertySource加载配置文件

@PropertySource注解用于指定自定义配置文件的具体位置和名称

如果需要将自定义配置文件中的属性值注入到对应类的属性中，可以使用@ConfigurationProperties或者@Value注解进行属性值注入

@PropertySource("classpath:test.properties")注解指定了自定义配置文件的位置和名称，此示例表示自定义配置文件为classpath类路径下的test.properties文件；

1.7.2 使用@Configuration编写自定义配置类

在Spring Boot框架中，推荐使用配置类的方式向容器中添加和配置组件

在Spring Boot框架中，通常使用@Configuration注解定义一个配置类，Spring Boot会自动扫描和识别配置类，从而替换传统Spring框架中的XML配置文件。

当定义一个配置类后，还需要在类中的方法上使用@Bean注解进行组件配置，将方法的返回对象注入到Spring容器中，并且组件名称默认使用的是方法名，当然也可以使用@Bean注解的name或value属性自定义组件的名称

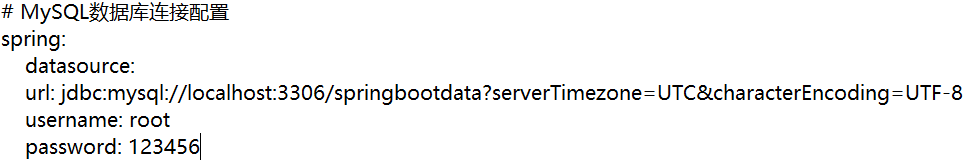
**2. SpringBoot数据访问**

**2.1 Spring Boot整合MyBatis**

MyBatis是一款优秀的持久层框架，Spring Boot官方虽然没有对MyBatis进行整合，但是MyBatis团队自行适配了对应的启动器，进一步简化了使用MyBatis进行数据的操作

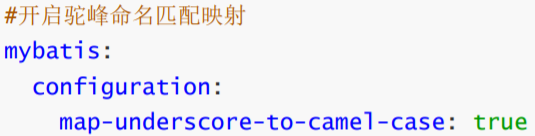
因为Spring Boot框架开发的便利性，所以实现Spring Boot与数据访问层框架(例如MyBatis)的整合非常简单，主要是引入对应的依赖启动器，并进行数据库相关参数设置即可

**在application.yml中配置Mysql数据库连接：**

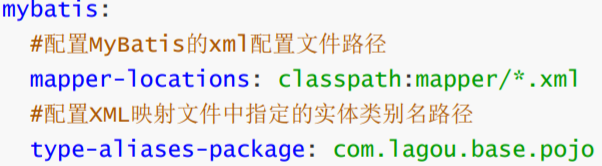


**在Spring Boot项目启动类上添加mapper扫描注解：@MapperScan("xxx")**

为了解决上述由于驼峰命名方式造成的表字段值无法正确映射到类属性的情况，可以在Spring Boot全局配置文件application.properties中添加开启驼峰命名匹配映射配置



**注：如果mybatis使用注解的方式，以上配置即可完成，如果使用配置文件的方式开发，还需要配置在application.yml文件中添加MyBatis映射文件路径的配置，同时需要添加实体类别名映射路径**



**2.2 Spring Boot整合Redis**

添加Redis依赖包



在application.yml中配置redis数据库连接信息



**3. SpringBoot视图技术**

**3.1 支持的视图技术**

前端模板引擎技术的出现，使前端开发人员无需关注后端业务的具体实现，只关注自己页面的呈现效果即可，并且解决了前端代码错综复杂的问题、实现了前后端分离开发。Spring Boot框架对很多常用的模板引擎技术（如：FreeMarker、Thymeleaf、Mustache等）提供了整合支持

Spring Boot不太支持常用的JSP模板，并且没有提供对应的整合配置，这是因为使用嵌入式Servlet容器的Spring Boot应用程序对于JSP模板存在一些限制 ：

(1) 在Jetty和Tomcat容器中，Spring Boot应用被打包成war文件可以支持JSP。但Spring Boot默认使用嵌入式Servlet容器以JAR包方式进行项目打包部署，这种JAR包方式不支持JSP。

(2) 如果使用Undertow嵌入式容器部署Spring Boot项目，也不支持JSP模板。（Undertow是红帽公司开发的一款基于NIO的高性能Web嵌入式服务器）

(3) Spring Boot默认提供了一个处理请求路径“/error”的统一错误处理器，返回具体的异常信息。使用JSP模板时，无法对默认的错误处理器进行覆盖，只能根据Spring Boot要求在指定位置定制错误页面。

**3.2 Thymeleaf**

3.2.1 Thymeleaf语法

在HTML页面上使用Thymeleaf标签，Thymeleaf 标签能够动态地替换掉静态内容，使页面动态展示。



上述代码中，“xmlns:th="http://www.thymeleaf.org"“ 用于引入Thymeleaf模板引擎标签，使用关键字“th”标注标签是Thymeleaf模板提供的标签，其中，“th:href”用于引入外联样式文件，“th:text”用于动态显示标签文本内容。

**常用标签**

|  |  |
| --- | --- |
| **th:标签** | **说明** |
| th:insert | 布局标签，替换内容到引入的文件 |
| th:replace | 页面片段包含（类似JSP中的include标签） |
| th:each | 元素遍历（类似JSP中的c:forEach标签） |
| th:if | 条件判断，如果为真 |
| th:unless | 条件判断，如果为假 |
| th:switch | 条件判断，进行选择性匹配 |
| th:case | 条件判断，进行选择性匹配 |
| th:value | 属性值修改，指定标签属性值 |
| th:href | 用于设定链接地址 |
| th:src | 用于设定链接地址 |
| th:text | 用于指定标签显示的文本内容 |

Thymeleaf模板引擎提供了多种**标准表达式语法**

**1．变量表达式 ${...}**

变量表达式${...}主要用于获取上下文中的变量值，示例代码如下：

**<p th:text="${title}">这是标题</p>**

示例使用了Thymeleaf模板的变量表达式${...}用来动态获取P标签中的内容，**如果当前程序没有启动或者当前上下文中不存在title变量，该片段会显示标签默认值“这是标题”**；**如果当前上下文中存在title变量并且程序已经启动，当前P标签中的默认文本内容将会被title变量的值所替换**，从而达到模板引擎页面数据动态替换的效果

Thymeleaf为变量所在域提供了一些**内置对象**，具体如下所示

|  |  |
| --- | --- |
| **内置对象** | **说明** |
| #ctx | 上下文对象 |
| #vars | 上下文变量 |
| #locale | 上下文区域设置 |
| #request | (仅限Web Context)HttpServletRequest对象 |
| #response | (仅限Web Context)HttpServletResponse对象 |
| #session | (仅限Web Context)HttpSession对象 |
| #servletContext | (仅限Web Context)ServletContext对象 |

结合上述内置对象的说明，假设要在Thymeleaf模板引擎页面中动态获取当前国家信息，可以使用#locale内置对象，示例代码如下

The locale country is: **<spanth:text="${#locale.country}">US</span>**

上述代码中，使用th:text="${#locale.country}"动态获取当前用户所在国家信息，其中标签内默认内容为US（美国），程序启动后通过浏览器查看当前页面时，Thymeleaf会通过浏览器语言设置来识别当前用户所在国家信息，从而实现动态替换

**2．选择变量表达式 \*{...}**

选择变量表达式和变量表达式用法类似，一般用于从被选定对象而不是上下文中获取属性值，如果没有选定对象，则和变量表达式一样，示例代码如下

<div th:object="${book}">

<p>titile:<span **th:text="\*{title}"**>标题</span>.</p>

</div>

\*{title} 选择变量表达式获取当前指定对象book的title属性值。

**3．消息表达式 #{...}**

消息表达式#{...}主要用于Thymeleaf模板页面国际化内容的动态替换和展示，使用消息表达式#{...}进行国际化设置时，还需要提供一些国际化配置文件。

**4．链接表达式 @{...}**

链接表达式@{...}一般用于页面跳转或者资源的引入，在Web开发中占据着非常重要的地位，并且使用也非常频繁，示例代码如下：

<a **th:href="@{http://localhost:8080/details(orderId=${o.id})}"**>view</a>

<a **th:href="@{/order/details(orderId=${o.id},pid=${p.id})}"**>view</a>

上述代码中，链接表达式@{...}分别编写了绝对链接地址和相对链接地址。在有参表达式中，需要按照@{路径(参数名称=参数值，参数名称=参数值...)}的形式编写，同时该参数的值可以使用变量表达式来传递动态参数值

**5．片段表达式 ~{...}**

片段表达式~{...}用来标记一个片段模板，并根据需要移动或传递给其他模板。其中，最常见的用法是使用th:insert或th:replace属性插入片段，示例代码如下：

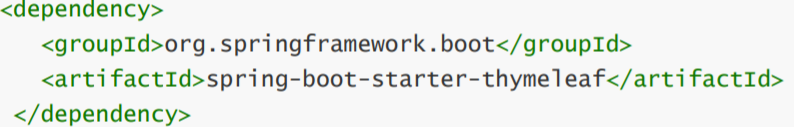
<div **th:insert="~{thymeleafDemo::title}"**></div>

上述代码中，使用th:insert属性将title片段模板引用到该

标签中。thymeleafDemo为模板名称，Thymeleaf会自动查找“/resources/templates/”目录下的thymeleafDemo模板，title为片段名称

3.2.2 基本使用

引入Thymeleaf依赖

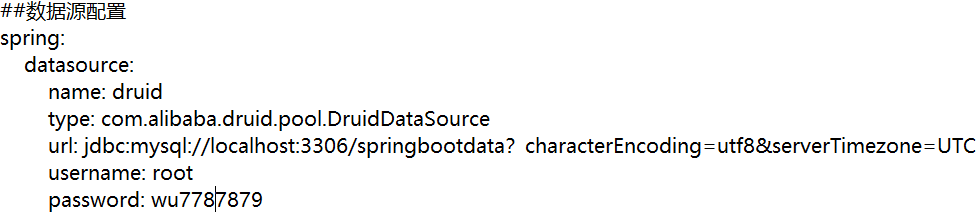


在全局配置文件中配置Thymeleaf模板的一些参数。一般Web项目都会使用下列配置



上述配置中，spring.thymeleaf.cache表示是否开启Thymeleaf模板缓存，默认为true，在开发过程中通常会关闭缓存，保证项目调试过程中数据能够及时响应；spring.thymeleaf.prefix指定了 Thymeleaf模板页面的存放路径，默认为classpath:/templates/；spring.thymeleaf.suffix指定了 Thymeleaf模板页面的名称后缀，默认为.html

**补：如果连接数据库要使用druid连接池，需要先引入druid的依赖，在application.yml配置文件中配置如下内容：**



**4. Spring Boot项目部署**

添加打包组件，使用maven的 package命令打包；部署运行：java -jar 包名



2.Spring Cloud微服务

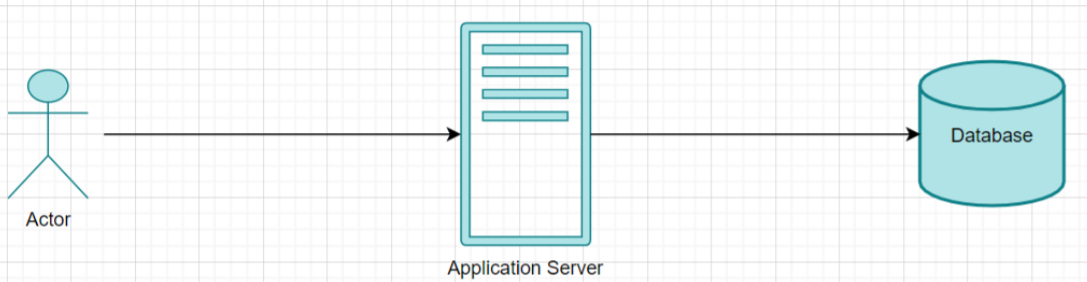
**第一部分 微服务架构**

**第 1 节 互联网应用架构演进**

1）单体应用架构

在诞生之初，拉钩用户量，数据规模都比较小，项目的所有功能模块都放在一个工程中编码、编译、打包并且部署在一个tomcat容器中的架构模式就是单体应用架构，这样的架构既简单实用，便于维护，成本又低，成为了那个时代的主流架构方式





**优点：**

**高效开发**：项目前期开发节奏快，退队成员少的时候可以快速迭代

**架构简单**：MVC架构，只需要借助IDE开发、调试即可

**易于测试**：只需要通过单元测试或者浏览器完成

**易于部署**：打包成单一可执行的jar或者war包放在容器中启动

单体架构的应用比较容易部署、测试，在项目的初期，单体应用可以很好地运行。然而，随着需求的不断增加，越来越多的人加入开发团队，代码库也在飞速地膨胀。慢慢地，单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，维护成本越来越高。

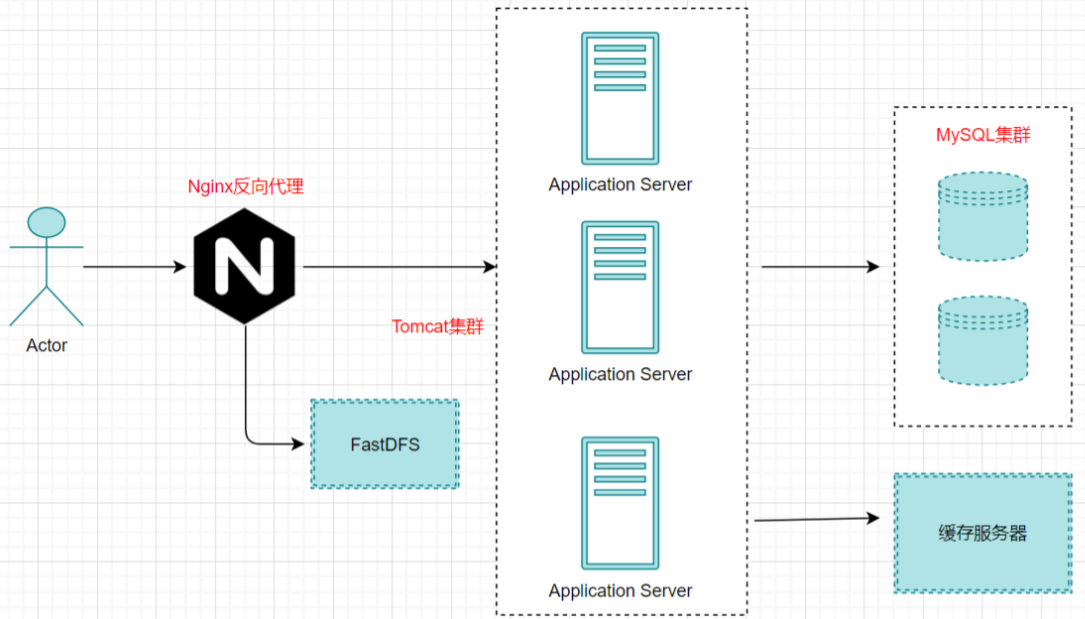
**缺点**：

**可靠性差**：某个应用Bug，如死循环、内存溢出等，可能会导致整个应用的崩溃

**复杂性高**：以一个百万行级别的单体应用为例，整个项目包含的模块多、模块的边界模糊、依赖关系不清晰、代码质量参差不齐、混乱地堆砌在一起。使得整个项目非常复杂。

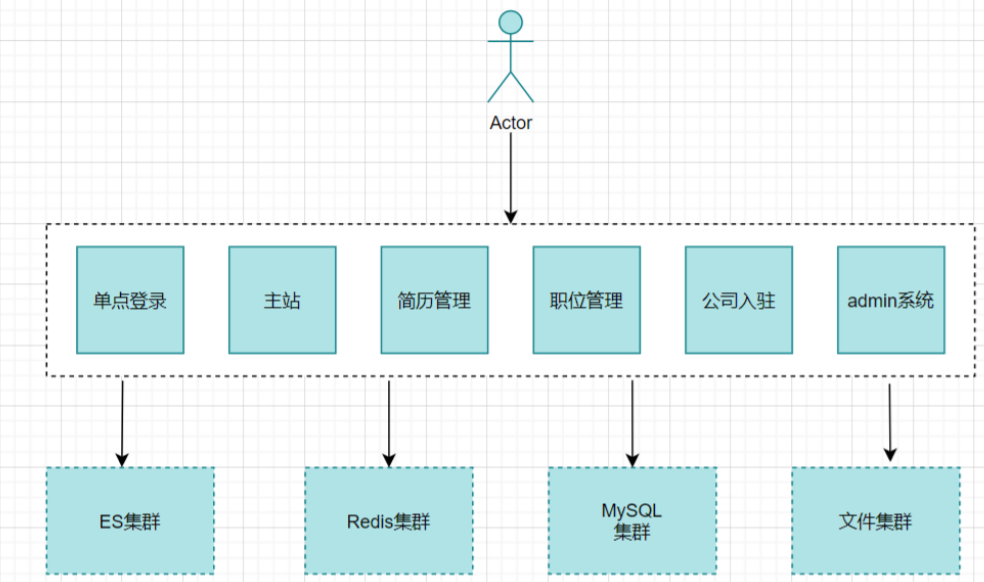
**扩展能力受限**：单体应用只能作为一个整体进行扩展，无法根据业务模块的需要进行伸缩。例如，应用中有的模块是计算密集型的，它需要强劲的CPU；有的模块则是IO密集型的，需要更大的内存。由于这些模块部署在一起，不得不在硬件的选择上做出妥协。

业务量上涨之后，单体应用架构进一步丰富变化，比如应用集群部署、使用Nginx进行负载均衡、增加缓存服务器、增加文件服务器、数据库集群并做读写分离等，通过以上措施增强应对高并发的能力、应对一定的复杂业务场景，但依然属于单体应用架构。



2）垂直应用架构

为了避免上⾯提到的那些问题，开始做模块的垂直划分，做垂直划分的原则是基于拉勾现有的业务特性来做，核心目标标第⼀个是为了业务之间互不影响，第⼆个是在研发团队的壮⼤后为了提⾼效率，减少组件之间的依赖。



**优点**

系统拆分实现了流量分担，解决了并发问题

可以针对不同模块进⾏优化

⽅便⽔平扩展，负载均衡，容错率提⾼

系统间相互独⽴，互不影响，新的业务迭代时更加⾼效

**缺点**

服务之间相互调⽤，如果某个服务的端⼝或者ip地址发⽣改变，调⽤的系统得⼿动改变

搭建集群之后，实现负载均衡⽐较复杂，如：内⽹负载，在迁移机器时会影响调⽤⽅的路由，导致线上故障

服务之间调⽤⽅式不统⼀，基于 httpclient、 webservice，接⼝协议不统⼀

服务监控不到位：除了依靠端⼝、进程的监控，调⽤的成功率、失败率、总耗时等等这些监控指标是没有的

3）SOA应用架构

在做了垂直划分以后，模块随之增多，维护的成本在也变⾼，⼀些通⽤的业务和模块重复的越来越多，为了解决上⾯提到的接⼝协议不统⼀、服务⽆法监控、服务的负载均衡，引⼊了阿⾥巴巴开源的Dubbo ，⼀款⾼性能、轻量级的开源Java RPC框架，可以和Spring框架无缝集成。它提供了三⼤核⼼能⼒：⾯向接⼝的远程⽅法调⽤，智能容错和负载均衡，以及服务⾃动注册和发现。

SOA (Service-Oriented Architecture)，即面向服务的架构。根据实际业务，把系统拆分成合适的、独立部署的模块，模块之间相互独立（通过Webservice/Dubbo等技术进行通信）。

**优点：分布式、松耦合、扩展灵活、可重用。**

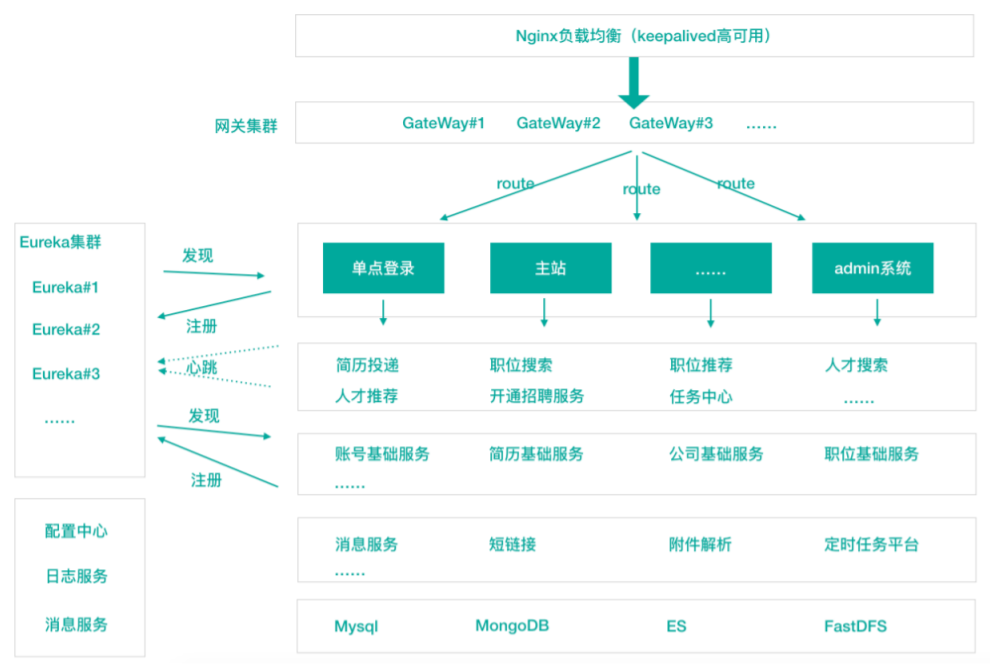
**缺点：服务抽取粒度较大、服务调用方和提供方耦合度较高（接口耦合度）**



4）微服务应用架构

微服务架构可以说是SOA架构的一种拓展，这种架构模式下它拆分粒度更小、服务更独立。把应用拆分成为一个个微小的服务，不同的服务可以使用不同的开发语言和存储，服务之间往往通过Restful等轻量级通信。微服务架构关键在于微小、独立、轻量级通信。

微服务是在 SOA 上做的升华粒度更加细致，微服务架构强调的⼀个重点是业务需要彻底的组件化和服务化



微服务架构和SOA架构很明显的一个区别就是**服务拆分粒度的不同**，但是对于拉勾的架构发展来说，我们所看到的SOA阶段其实服务拆分粒度相对来说已经比较细了（超前哦！），所以上述拉勾SOA 到拉勾微服务，从服务拆分上来说变化并不大，只是引入了相对完整的新一代Spring Cloud微服务技术。自然，上述我们看到的都是拉勾架构演变的阶段结果，每一个阶段其实都经历了很多变化，拉勾的服务拆分其实也是走过了从粗到细，并非绝对的一步到位。

**第 2 节 微服务架构体现的思想及优缺点**

微服务架构设计的核心思想就是“**微**”，拆分的粒度相对比较小，这样的话单一职责、开发的**耦合度就会降低**、微小的功能可以独立部署扩展、**灵活性强**，**升级改造影响范围小**。

**微服务架构的优点**: 微服务架构和微服务

微服务很小，便于特定业务功能的聚焦

微服务很小，每个微服务都可以被一个小团队单独实施(开发、测试、部署上线、运维)，团队合作一定程度解耦，便于实施敏捷开发

微服务很小，便于重用和模块之间的组装

微服务很独立，那么不同的微服务可以使用不同的语言开发，松耦合

微服务架构下，我们更容易引入新技术

**微服务架构的缺点**

微服务架构下，分布式复杂难以管理，当服务数量增加，管理将越加复杂；

微服务架构下，分布式链路跟踪难等；

**第 3 节 微服务架构中的核心概念**

**服务注册与服务发现**

服务注册：服务提供者将所提供服务的信息（服务器IP和端口、服务访问协议等）注册/登记到注册中心

服务发现：服务消费者能够从注册中心获取到较为实时的服务列表，然后根究一定的策略选择一个服务访问

**负载均衡**

负载均衡即将请求压力分配到多个服务器（应用服务器、数据库服务器等），以此来提高服务的性能、可靠性

**熔断**

熔断即断路保护。微服务架构中，如果下游服务因访问压力过大而响应变慢或失败，上游服务为了保护系统整体可用性，可以暂时切断对下游服务的调用。这种牺牲局部，保全整体的措施就叫做熔断。

**链路追踪**

微服务架构越发流行，一个项目往往拆分成很多个服务，那么一次请求就需要涉及到很多个服务。不同的微服务可能是由不同的团队开发、可能使用不同的编程语言实现、整个项目也有可能部署在了很多服务器上（甚至百台、千台）横跨多个不同的数据中心。所谓链路追踪，就是对一次请求涉及的很多个服务链路进行日志记录、性能监控

**API 网关**

微服务架构下，不同的微服务往往会有不同的访问地址，客户端可能需要调用多个服务的接口才能完成一个业务需求，如果让客户端直接与各个微服务通信可能出现：

1）客户端需要调用不同的url地址，增加了维护调用难度

2）在一定的场景下，也存在跨域请求的问题（前后端分离就会碰到跨域问题，原本我们在后端采用Cors就能解决，现在利用网关，那么就放在网关这层做好了）

3）每个微服务都需要进行单独的身份认证

那么，API网关就可以较好的统一处理上述问题，API请求调用统一接入API网关层，由网关转发请求。API网关更专注在安全、路由、流量等问题的处理上（微服务团队专注于处理业务逻辑即可），它的功能比如：z

1）统一接入（路由）

2）安全防护（统一鉴权，负责网关访问身份认证验证，与“访问认证中心”通信，实际认证业务逻辑交移“访问认证中心”处理）

3）黑白名单（实现通过IP地址控制禁止访问网关功能，控制访问）

4）协议适配（实现通信协议校验、适配转换的功能）

5）流量管控（限流）

6）长短链接支持

7）容错能力（负载均衡）

**第二部分 Spring Cloud 综述**

**第 1 节 Spring Cloud 是什么**

Spring Cloud是一**系列框架的有序集合**。它**利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发**，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot的开发风格做到一键启动和部署。S**pring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。**

**Spring Cloud是一系列框架的有序集合（Spring Cloud是一个规范）**

**开发服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等**

**利用Spring Boot的开发便利性简化了微服务架构的开发（自动装配）**

这里，我们需要注意，**Spring Cloud其实是一套规范**，是一套用于构建微服务架构的规范，而不是一个可以拿来即用的框架（所谓规范就是应该有哪些功能组件，然后组件之间怎么配合，共同完成什么事情）。在这个规范之下第三方的Netflix公司开发了一些组件、Spring官方开发了一些框架/组件，包括第三方的阿里巴巴开发了一套框架/组件集合Spring Cloud Alibaba，这些才是Spring Cloud规范的实现。

Netflix搞了一套 ，简称SCN

阿里巴巴在之前的基础上搞出了一堆微服务组件,Spring Cloud Alibaba（SCA）

**第 2 节 Spring Cloud 解决什么问题**

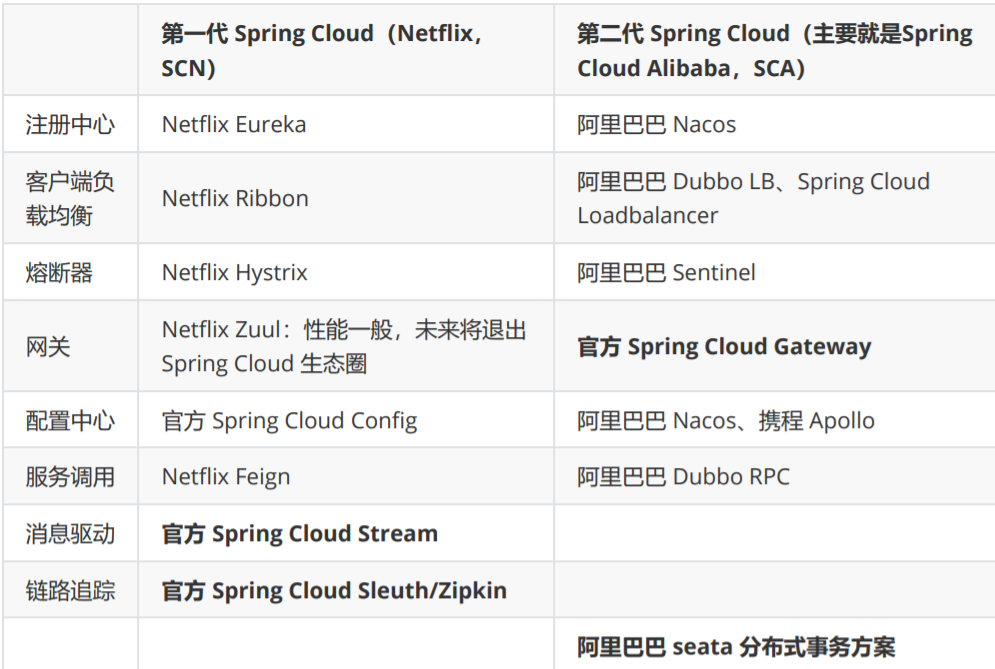
Spring Cloud 规范及实现意图要解决的问题其实就是微服务架构实施过程中存在的一些问题，比如微服务架构中的**服务注册发现问题、网络问题（比如熔断场景）、统一认证安全授权问题、负载均衡问题、链路追踪等问题**。

**第 3 节 Spring Cloud 架构**

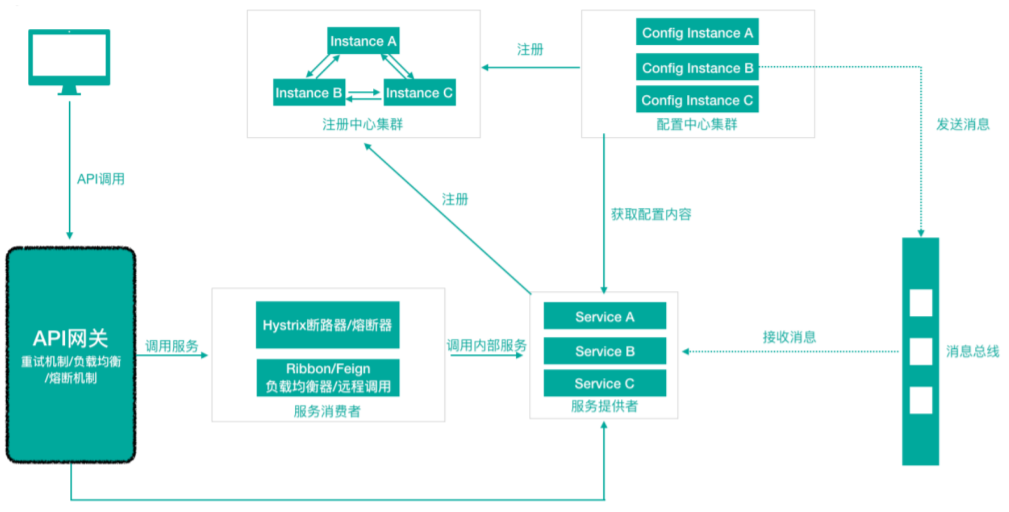
Spring Cloud是一个微服务相关规范，这个规范意图为搭建微服务架构提供一站式服务，**采用组件（框架）化机制**定义一系列组件，各类组件针对性的处理微服务中的特定问题，这些组件 共同来构成**Spring Cloud微服务技术栈**

**3.1 Spring Cloud 核心组件**

Spring Cloud 生态圈中的组件，按照发展可以分为第一代 Spring Cloud组件和第二代 Spring Cloud组件。



**3.2 Spring Cloud 体系结构（组件协同工作机制）**



Spring Cloud中的各组件协同工作，才能够支持一个完整的微服务架构。比如

（1）注册中心负责服务的注册与发现，很好将各服务连接起来

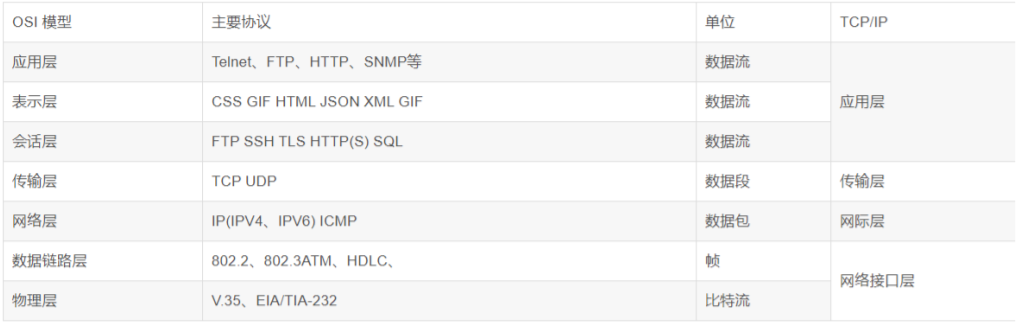
（2）API网关负责转发所有外来的请求

（3）断路器负责监控服务之间的调用情况，连续多次失败进行熔断保护。

（4）配置中心提供了统一的配置信息管理服务,可以实时的通知各个服务获取最新的配置信息

**第 4 节 Spring Cloud 与 Dubbo 对比**

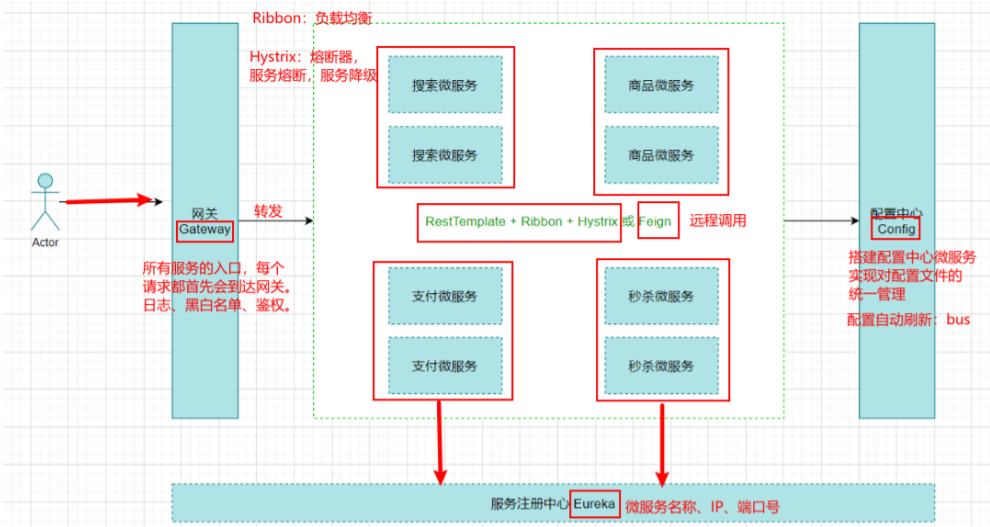
Dubbo是阿里巴巴公司开源的一个高性能优秀的服务框架，基于RPC调用，对于目前使用率较高的Spring Cloud Netflix来说，它是基于HTTP的，所以效率上没有Dubbo高，但问题在于Dubbo体系的组件不全，不能够提供一站式解决方案，比如服务注册与发现需要借助于Zookeeper等实现，而Spring Cloud Netflix则是真正的提供了一站式服务化解决方案，且有Spring大家族背景。



**第 5 节 Spring Cloud 与 Spring Boot 的关系**

Spring Cloud只是利用了Spring Boot的特点，让我们能够快速的实现微服务组件开发，否则不使用Spring Boot的话，我们在使用Spring Cloud时，每一个组件的相关Jar包都需要我们自己导入配置以及需要开发人员考虑兼容性等各种情况。所以Spring Boot是我们快速把Spring Cloud微服务技术应用起来的一种方式。

**第四部分 第一代 Spring Cloud 核心组件**



从形式上来说，Feign一个顶三，Feign = RestTemplate + Ribbon + Hystrix

**第 1 节 Eureka服务注册中心**

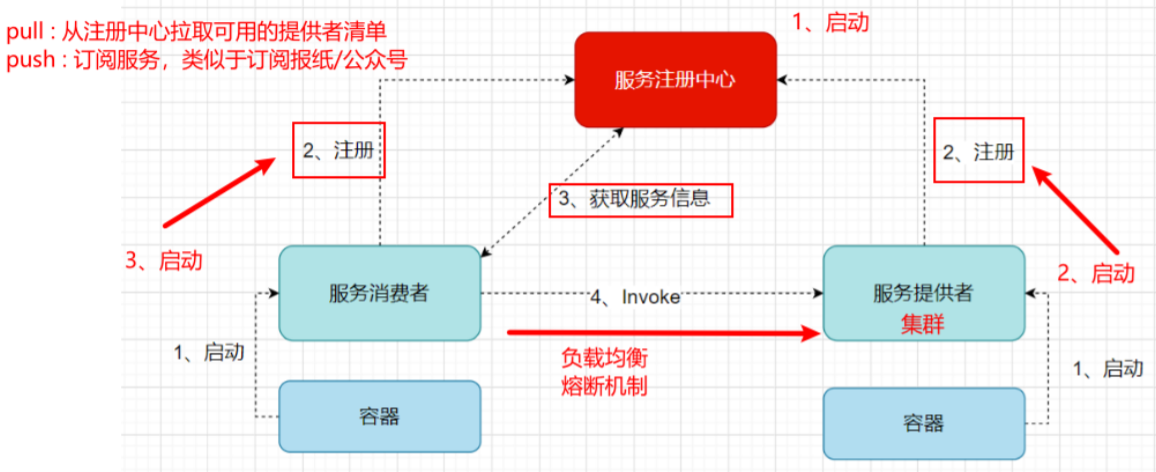
常用的服务注册中心：Eureka、Nacos、Zookeeper、Consul

**1.1 关于服务注册中心**

**注意：服务注册中心本质上是为了解耦服务提供者和服务消费者。**

对于任何一个微服务，原则上都应存在或者支持多个提供者（比如商品微服务部署多个实例），这是由微服务的**分布式属性决定的**。

1.1.1注册中心实现原理



分布式微服务架构中，服务注册中心用于存储服务提供者地址信息、服务发布相关的属性信息，消费者通过主动查询和被动通知的方式获取服务提供者的地址信息，而不再需要通过硬编码方式得到提供者的地址信息。消费者只需要知道当前系统发布了那些服务，而不需要知道服务具体存在于什么位置，这就是透明化路由。

1）服务提供者启动

2）服务提供者将相关服务信息主动注册到注册中心

3）服务消费者获取服务注册信息：

pull模式：服务消费者可以主动拉取可用的服务提供者清单

push模式：服务消费者订阅服务（当服务提供者有变化时，注册中心也会主动推送更新后的服务清单给消费者）

4）服务消费者直接调用服务提供者

另外，注册中心也需要完成服务提供者的健康监控，当发现服务提供者失效时需要及时剔除；

1.1.2主流服务中心对比

**Zookeeper**

Dubbo + Zookeeper

Zookeeper它是一个分布式服务框架，是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。

简单来说zookeeper本质 = 存储 + 监听通知。

Zookeeper用来做服务注册中心，主要是因为它具有节点变更通知功能，只要客户端监听相关服务节点，服务节点的所有变更，都能及时的通知到监听客户端，这样作为调用方只要使用Zookeeper的客户端就能实现服务节点的订阅和变更通知功能了，非常方便。另外，Zookeeper可用性也可以，因为只要半数以上的选举节点存活，整个集群就是可用的，最少节点数为3。

**Eureka**

Eureka由Netflix开源，并被Pivatal集成到SpringCloud体系中，它是基于 RestfulAPI 风格开发的服务注册与发现组件。

**Consul**

Consul是由HashiCorp基于Go语言开发的支持多数据中心分布式高可用的服务发布和注册服务软件，采用Raft算法保证服务的一致性，且支持健康检查。

**Nacos**

Nacos是一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。简单来说Nacos就是**注册中心 + 配置中心的组合**，帮助我们解决微服务开发必会涉及到的服务注册与发现，服务配置，服务管理等问题。Nacos 是 Spring Cloud Alibaba 核心组件之一，**负责服务注册与发现，还有配置**。

|  |  |  |  |
| --- | --- | --- | --- |
| 组件名 | 语言 | CAP | 对外暴露接口 |
| Eureka | Java | AP（自我保护机制，保证可用） | HTTP |
| Consul | Go | CP | HTTP /DNS |
| Zookeeper | Java | CP | 客户端 |
| Nacos | Java | 支持AP/CP切换 | HTTP |

CAP定理又称CAP原则，指的是在一个分布式系统中，**Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性）**，最多只能同时三个特性中的两个，三者不可兼得。

P：分区容错性：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务（一定的要满足的）

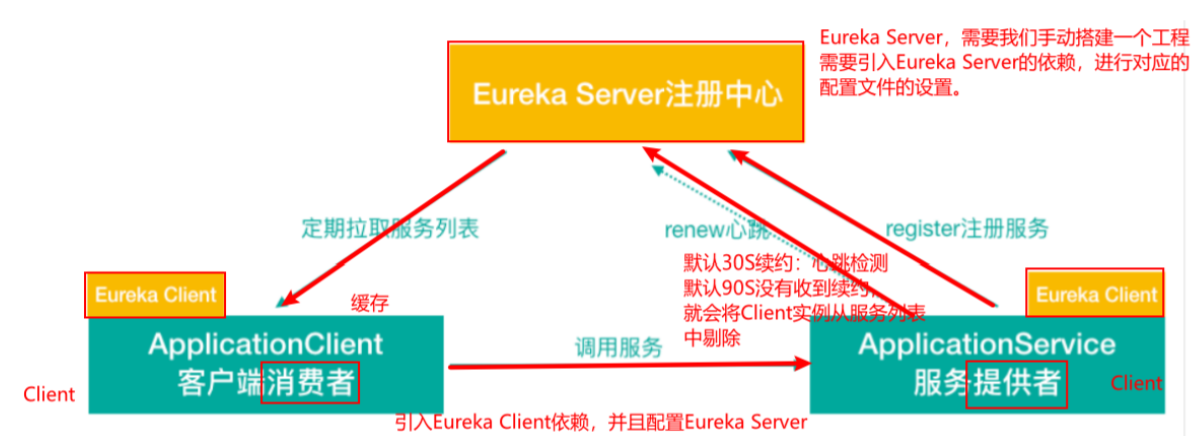
C：数据一致性：all nodes see the same data at the same time

A：高可用：Reads and writes always succeed

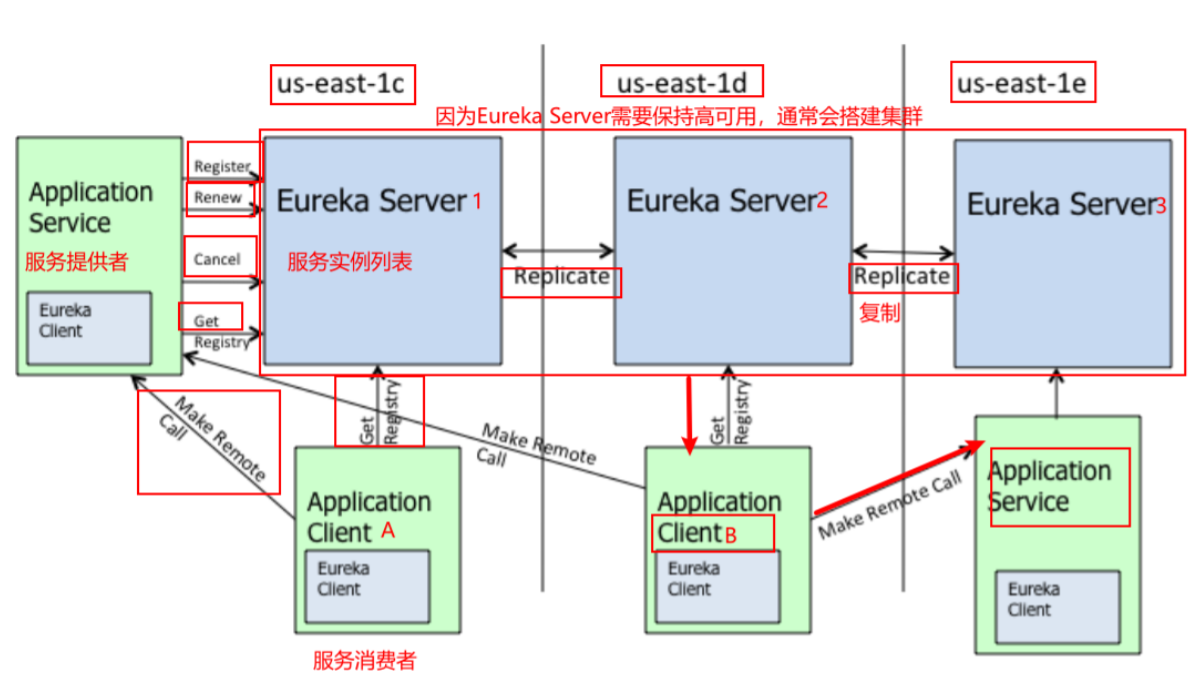
**CAP不可能同时满足三个，要么是AP，要么是CP**

**1.2 服务注册中心组件 Eureka**

Eureka 基础架构



Eureka 交互流程及原理



Eureka包含两个组件：Eureka Server和 Eureka Client，Eureka Client是一个Java客户端，用于简化与Eureka Server的交互；Eureka Server提供服务发现的能力，各个微服务启动时，会通过Eureka Client向Eureka Server进行注册自己的信息（例如网络信息），Eureka Server会存储该服务的信息；

（1）图中Application Service作为服务提供者向Eureka Server中注册服务，Eureka Server接受到注册事件会在集群和分区中进行数据同步，Application Client作为消费端（服务消费者）可以从Eureka Server中获取到服务注册信息，进行服务调用。

（2）微服务启动后，会周期性地向Eureka Server发送心跳（**默认周期为30秒**，**默认Eureka Server90S会将还没有续约的给剔除**）以续约自己的信息

（3）Eureka Server在一定时间内没有接收到某个微服务节点的心跳，Eureka Server将会注销该微服务节点（**默认90秒**）

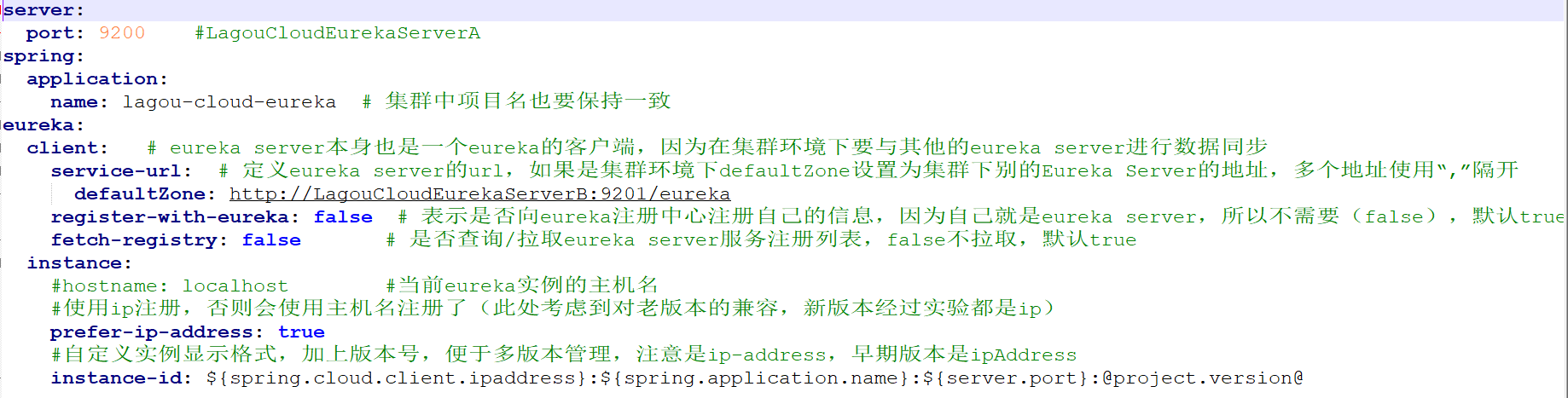
（4）每个Eureka Server同时也是Eureka Client，多个Eureka Server之间通过复制的方式完成服务注册列表的同步

（5）Eureka Client会缓存Eureka Server中的信息。即使所有的Eureka Server节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者

**Eureka通过心跳检测、健康检查和客户端缓存等机制，提高系统的灵活性、可伸缩性和高可用性。**

**1.3 搭建Eureka Server服务注册中心**

在eureka server的application.yml配置文件中配置如下内容：



在eureka server的启动类上添加**@EnableEurekaServer注解，声明是一个eureka服务**

**1.4 搭建Eureka Client，将服务注册到Eureka Server注册中心**

在eureka client的application.yml配置文件中配置如下内容：



在eureka client的启动类上添加**@EnableDiscoveryClient 或者@EnableEurekaClient声明这是一个服务客户端**

**@EnableDiscoveryClient：将服务注册到任意的注册中心**

**@EnableEurekaClient：将服务注册到eureka注册中心**

**1.5 Eureka细节详解**

1.5.1 Eureka元数据详解

Eureka的元数据有两种：标准元数据和自定义元数据。

标准元数据：主机名、IP地址、端口号等信息，这些信息都会被发布在服务注册表中，用于服务之间的调用。

自定义元数据：可以使用**eureka.instance.metadata-map**配置，符合**KEY/VALUE**的存储格式。这些元数据可以在远程客户端中访问。

1.5.2 Eureka客户端详解

服务提供者(也是Eureka客户端)要向EurekaServer注册服务，并完成服务续约等工作

**服务注册详解（服务提供者）**

1）当我们导入了eureka-client依赖坐标，配置Eureka服务注册中心地址

2）服务在启动时会向注册中心发起注册请求，携带服务元数据信息

3）Eureka注册中心会把服务的信息保存在Map中。

**服务续约详解（服务提供者）**

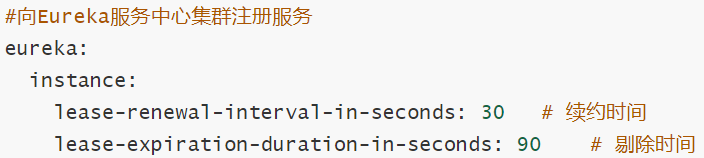
服务每隔30秒会向注册中心续约(心跳)一次（也称为报活），如果没有续约，租约在90秒后到期，然后服务会被失效。每隔30秒的续约操作我们称之为心跳检测

Eureka Client：30S续约一次，在Eureka Server更新自己的状态 (Client端进行配置)

Eureka Server：90S还没有进行续约，将该微服务实例从服务注册表（Map）剔除 (Client端进行配置)

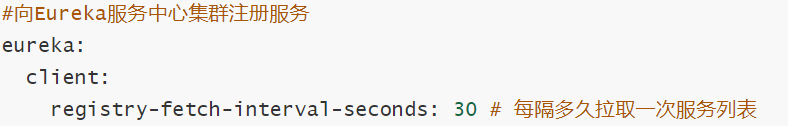
Eureka Client：30S拉取服务最新的注册表并缓存到本地 (Client端进行配置)

往往不需要我们调整这两个配置



**获取服务列表（服务注册表）详解（服务消费者）**

每隔30秒服务会从注册中心中拉取一份服务列表，这个时间可以通过配置修改。往往不需要我们调整



1）服务消费者启动时，从 EurekaServer服务列表获取只读备份，缓存到本地

2）每隔30秒，会重新获取并更新数据

3）每隔30秒的时间可以通过配置eureka.client.registry-fetch-interval-seconds修改

1.5.3 Eureka服务端详解

**服务下线：**

1）当服务正常关闭操作时，会发送服务下线的REST请求给EurekaServer。

2）服务中心接受到请求后，将该服务置为下线状态

**失效剔除：**

Eureka Server会定时（间隔值是eureka.server.eviction-interval-timer-in-ms，默认60s）进行检查，如果发现实例在在一定时间（此值由客户端设置的eureka.instance.lease-expiration-duration-inseconds定义，默认值为90s）内没有收到心跳，则会注销此实例。

**自我保护机制：**

**自我保护模式正是一种针对网络异常波动的安全保护措施，使用自我保护模式能使Eureka集群更加的 健壮、稳定的运行。**

自我保护机制的工作机制是：**如果在15分钟内超过85%的客户端节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，Eureka Server自动进入自我保护机制**，此时会出现以下几种情况：

1. Eureka Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。

2. Eureka Server仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上，保证当前节点依然可用。

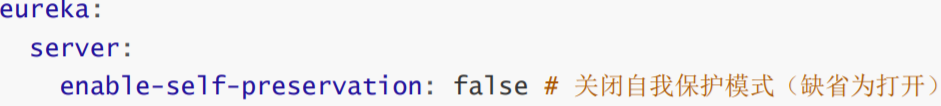
3. 当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中

因此Eureka Server可以很好的应对因网络故障导致部分节点失联的情况，而不会像ZK那样如果有一半不可用的情况会导致整个集群不可用而变成瘫痪。

**为什么会有自我保护机制？**

默认情况下，如果Eureka Server在一定时间内（默认90秒）没有接收到某个微服务实例的心跳，Eureka Server将会移除该实例。但是当网络分区故障发生时，微服务与Eureka Server之间无法正常通信，而微服务本身是正常运行的，此时不应该移除这个微服务，所以引入了自我保护机制。

我们在单机测试的时候很容易满足心跳失败比例在 15 分钟之内低于85%，这个时候就会触发Eureka 的保护机制，一旦开启了保护机制（**默认开启**），则服务注册中心维护的服务实例就不是那么准确了，此时我们通过修改Eureka Server的配置文件来关闭保护机制，这样可以确保注册中心中不可用的实例被 及时的剔除（**不推荐**）。



**经验：建议生产环境打开自我保护机制**

**第 2 节 Ribbon负载均衡**

**2.1 关于负载均衡**

负载均衡一般分为**服务器端负载均衡和客户端负载均衡**

所谓服务器端负载均衡，比如Nginx、F5这些，请求到达服务器之后由这些负载均衡器根据一定的 算法将请求路由到目标服务器处理。

所谓客户端负载均衡，比如我们要说的Ribbon，服务消费者客户端会有一个服务器地址列表，调用方在请求前通过一定的负载均衡算法选择一个服务器进行访问，负载均衡算法的执行是在请求客户端进行。

Ribbon是Netflix发布的负载均衡器。Eureka一般配合Ribbon进行使用，Ribbon利用从Eureka中读取到服务信息，在调用服务提供者提供的服务时，会根据一定的算法进行负载。

**Ribbon负载均衡的实现就是在生成RestTemplate实例的方法上添加@LoadBalanced注解即可，生成RestTemplate对象的方法写在启动类中，使用@Bean注解将对象交给IOC容器进行管理**

RestTemplate对象就是发送请求



代码中第二个参数是返回值类型的字节码文件

**2.2 Ribbon负载均衡策略**

|  |  |
| --- | --- |
| RoundRobinRule：轮询策略 | 默认超过10次获取到的server都不可用，会返回一个空的server |
| RandomRule：随机策略 | 如果随机到的server为null或者不可用的话，会while不停的循环选取 |
| RetryRule：重试策略 | 一定时限内循环重试。默认继承RoundRobinRule，也支持自定义注入，RetryRule会在每次选取之后，对选举的server进行判断，是否为null，是否alive，并且在500ms内会不停的选取判断。而 RoundRobinRule失效的策略是超过10次，RandomRule是没有失效时间的概念，只要serverList没都挂。 |
| BestAvailableRule：最小连接数策略 | 遍历serverList，选取出可用的且连接数最小的一个server。该算法里面有一个LoadBalancerStats的成员变量，会存储所有server的运行状况和连接数。如果选取到的server为null，会调用RoundRobinRule重新选取。 |
| AvailabilityFilteringRule：可用过滤策略 | 扩展了轮询策略，会先通过默认的轮询选取一个server，再去判断该zserver是否超时可用，当前连接数是否超限，都成功再返回。 |
| ZoneAvoidanceRule：区域权衡策略（**默认策略**） | 扩展了轮询策略，继承了2个过滤器：ZoneAvoidancePredicate和AvailabilityPredicate，除了过滤超时和链接数过多的server，还会过滤掉不符合要求的zone区域里面的所有节点， 在一个区域/机房内的服务实例中轮询。**先过滤再轮询** |

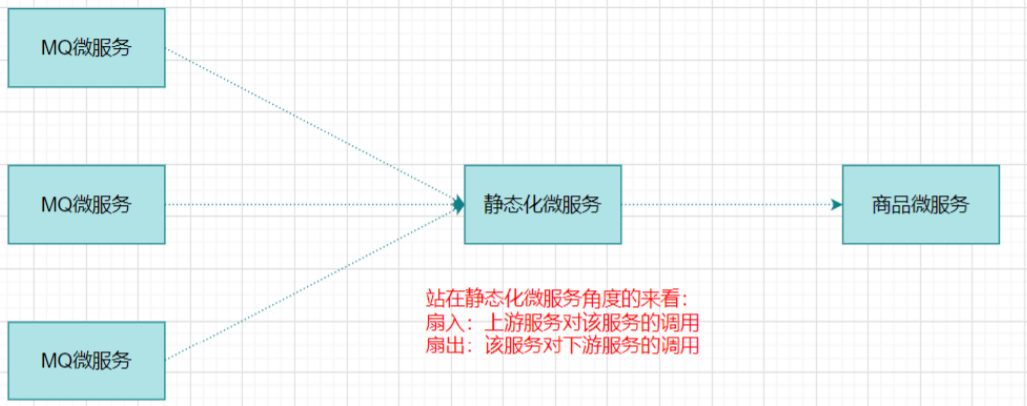
**修改负载均衡策略：**



**第 3 节Hystrix熔断器**

**3.1 微服务中的雪崩效应**

服务雪崩效应：是一种因“服务提供者的不可用”（原因）导致“服务调用者不可用”（结果），并将不可用逐渐放大的现象。



扇入：代表着该微服务被调用的次数，扇入大，说明该模块复用性好

扇出：该微服务调用其他微服务的个数，扇出大，说明业务逻辑复杂

**扇入大是一个好事，扇出大不一定是好事**

在微服务架构中，一个应用可能会有多个微服务组成，微服务之间的数据交互通过远程过程调用完成。这就带来一个问题，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其它的微服务，这就是所谓的“扇出”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“雪崩效应”。

如图中所示，最下游商品微服务响应时间过长，大量请求阻塞，大量线程不会释放，会导致服务器资源耗尽，最终导致上游服务甚至整个系统瘫痪。

**形成原因：**

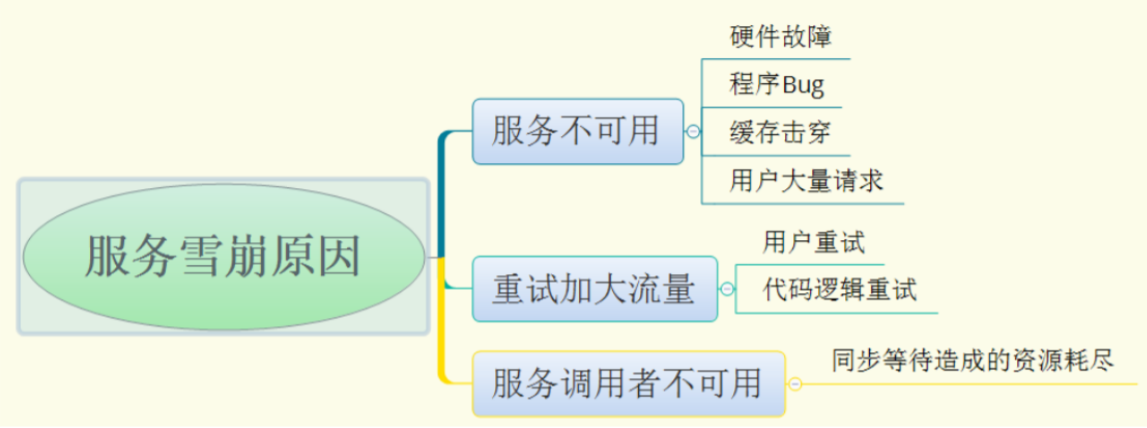
服务雪崩的过程可以分为三个阶段：

1. 服务提供者不可用

2. 重试加大请求流量

3. 服务调用者不可用

服务雪崩的每个阶段都可能由不同的原因造成：



**3.2 雪崩效应解决方案**

**服务熔断**

熔断机制是应对雪崩效应的一种微服务链路保护机制。我们在各种场景下都会接触到熔断这两个字。当扇出链路的某个微服务不可用或者响应时间太长时，熔断该节点微服务的调用，进行服务的降级，快速返回错误的响应信息。当检测到该节点微服务调用响应正常后，恢复调用链路。

**注意：**

1）服务熔断重点在“**断**”，切断对下游服务的调用

2）服务熔断和服务降级往往是一起使用的，Hystrix就是这样。

**服务降级**

通俗讲就是整体资源不够用了，先将一些不关紧的服务停掉（调用我的时候，给你返回一个预留的值，也叫做兜底数据），待渡过难关高峰过去，再把那些服务打开。

服务降级一般是从整体考虑，就是当某个服务熔断之后，服务器将不再被调用，此刻客户端可以自己准备一个本地的fallback回调，返回一个缺省值，这样做，虽然服务水平下降，但好歹可用，比直接挂掉要强。

**服务限流**

服务降级是当服务出问题或者影响到核心流程的性能时，暂时将服务屏蔽掉，待高峰或者问题解决后再打开；但是有些场景并不能用服务降级来解决，比如秒杀业务这样的核心功能，这个时候可以结合服务限流来限制这些场景的并发/请求量

限流措施也很多，比如：限制总并发数（比如数据库连接池、线程池）；限制瞬时并发数（如nginx限制瞬时并发连接数）；限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit\_req模块，限制每秒的平均速率）；限制远程接口调用速率、限制MQ的消费速率等

**3.3 Hystrix简介**

]Hystrix(豪猪)，宣言“defend your application”是由Netflix开源的一个延迟和容错库，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。Hystrix主要通过以下几点实现延迟和容错。

包裹请求：使用HystrixCommand包裹对依赖的调用逻辑。 页面静态化微服务方法 （@HystrixCommand 添加Hystrix控制）

跳闸机制：当某服务的错误率超过一定的阈值时，Hystrix可以跳闸，停止请求该服务一段时间。

资源隔离：Hystrix为每个依赖都维护了一个小型的线程池(舱壁模式)。如果该线程池已满，发往该依赖的请求就被立即拒绝，而不是排队等待，从而加速失败判定。

监控：Hystrix可以近乎实时地监控运行指标和配置的变化，例如成功、失败、超时、以及被拒绝的请求等。

回退机制：当请求失败、超时、被拒绝，或当断路器打开时，执行回退逻辑。回退逻辑由开发人员自行提供，例如返回一个缺省值。

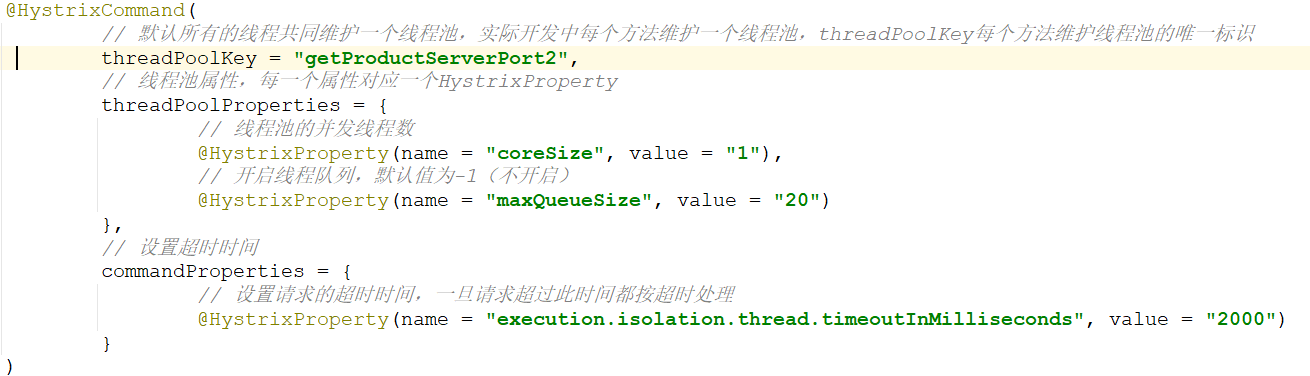
自我修复：断路器打开一段时间后，会自动进入“半开”状态（探测服务是否可用，如还是不可用，再次退回打开状态）。

**3.4 Hystrix应用**

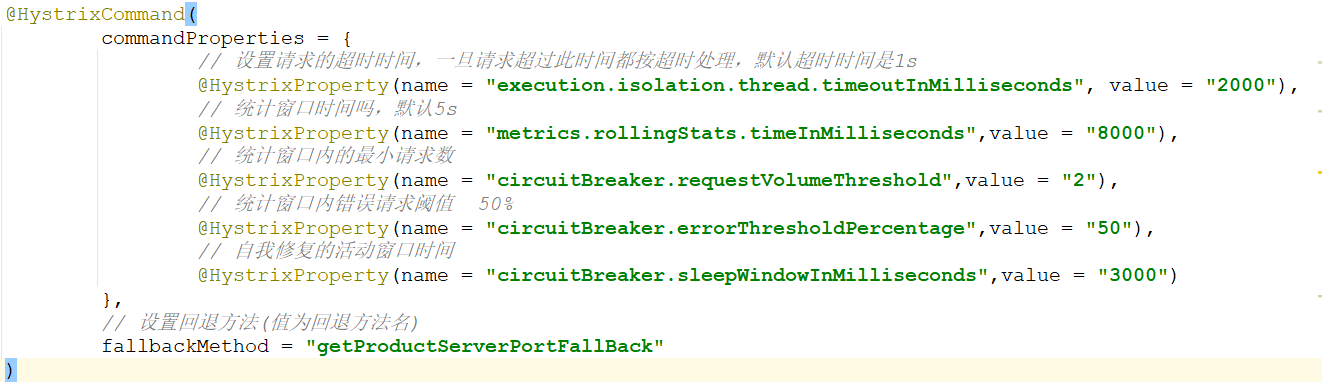
3.4.1.熔断处理

在启动类上添加**@EnableCircuitBreaker注解**开启熔断

在服务降级处理方法使用**@HystrixCommand**的**fallbackMethod**属性关联到服务降 级处理方法



3.4.2降级处理



回退方法：当请求发生熔断时的补救措施

**注意：1.方法的形参和原方法的形参保持一致；2.方法的返回值和原方法的返回值保持一致**

**3.5 Hystrix舱壁模式**

即：线程池隔离策略

如果不进行任何设置，所有熔断方法使用一个Hystrix线程池（10个线程），那么这样的话会导致问题，这个问题并不是扇出链路微服务不可用导致的，而是我们的线程机制导致的，如果方法A的请求把10个线程都用了，方法2请求处理的时候压根都没法去访问B，因为没有线程可用，并不是B服务不可用。

为了避免问题服务请求过多导致正常服务无法访问，Hystrix 不是采用增加线程数，而是单独的为每一个控制方法创建一个线程池的方式，这种模式叫做“舱壁模式"，也是线程隔离的手段。

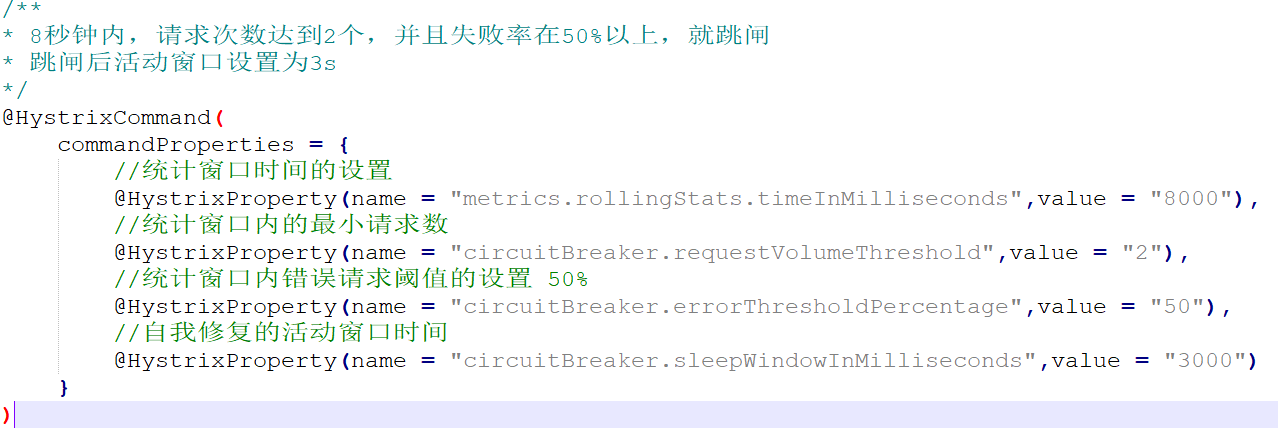
**3.6 Hystrix工作流程与高级应用**



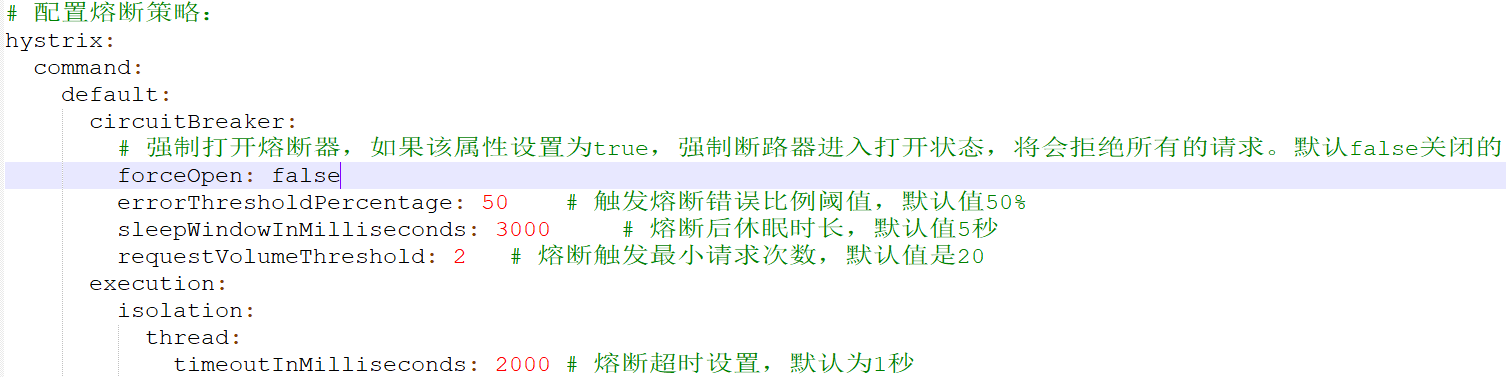
1）当调用出现问题时，开启一个时间窗（10s）

2）在这个时间窗内，统计调用次数是否达到最小请求数？如果没有达到，则重置统计信息，回到第1步；如果达到了，则统计失败的请求数占所有请求数的百分比，是否达到阈值？如果达到，则跳闸（不再请求对应服务）；如果没有达到，则重置统计信息，回到第1步

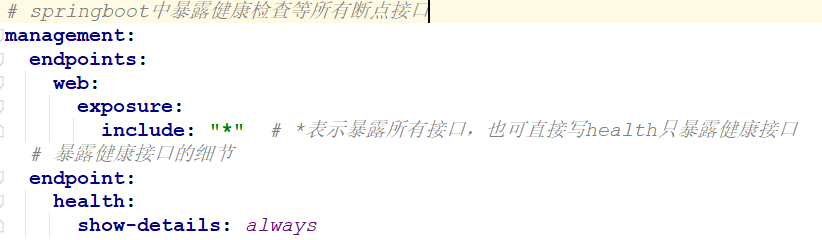
3）如果跳闸，则会开启一个活动窗口（默认5s），每隔5s，Hystrix会让一个请求通过,到达那个问题服务，看是否调用成功，如果成功，重置断路器回到第1步，如果失败，回到第3步



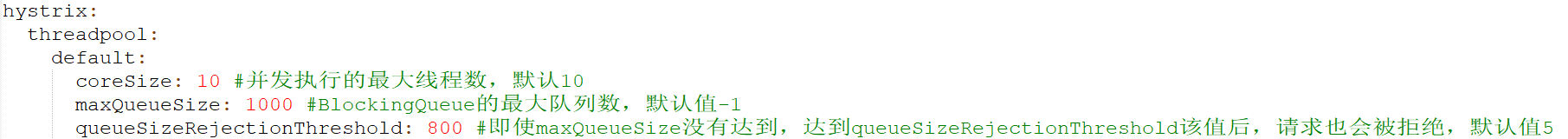
我们上述通过注解进行的配置也可以配置在配置文件中:



基于springboot的健康检查观察跳闸状态（自动投递微服务暴露健康检查细节）



**Hystrix 线程池队列配置**



**第 4 节 Feign远程调用组件**

在之前的案例中，服务消费者调用服务提供者的时候使用RestTemplate技术。

**4.1 Feign简介**

**Feign是Netflix开发的一个轻量级RESTful的HTTP服务客户端（用它来发起请求，远程调用的）**，**是以Java接口注解的方式调用Http请求**，而不用像Java中通过封装HTTP请求报文的方式直接调用， Feign被广泛应用在Spring Cloud 的解决方案中。

类似于Dubbo，服务消费者拿到服务提供者的接口，然后像调用本地接口方法一样去调用，实际发出的是远程的请求。

Feign可帮助我们更加便捷，优雅的调用HTTP API：不需要我们去拼接url然后呢调用restTemplate的api，在SpringCloud中，使用Feign非常简单，创建一个接口（在消费者--服务调用方这一端），并在接口上添加一些注解，代码就完成了

SpringCloud对Feign进行了增强，使Feign支持了SpringMVC注解（OpenFeign）

**本质：封装了Http调用流程，更符合面向接口化的编程习惯，类似于Dubbo的服务调用**

**4.2 Feign配置应用**

在服务调用者工程（消费）创建接口（添加注解）

（效果）Feign = RestTemplate+Ribbon+Hystrix

在消费者工程引入openfgign的启动依赖

在消费者工程的启动类上添加**@EnableFeignClients注解声明开启feign**

**注意：此时去掉Hystrix熔断的支持注解@EnableCircuitBreaker即可包括引入的依赖，因为Feign会自动引入**

在消费者微服务中创建Feign接口，接口上使用**@FeignClient(name = "xxx")，表示此接口时是一个feign接口，name属性的值就是服务提供者的名称（在配置文件中配置的），接口中的抽象方法与服务提供者的方法一摸一样，请求的路径为服务提供者controller中的方法**

**注意：**

1）@FeignClient注解的name属性用于指定要调用的服务提供者名称，和服务提供者yml文件中spring.application.name保持一致

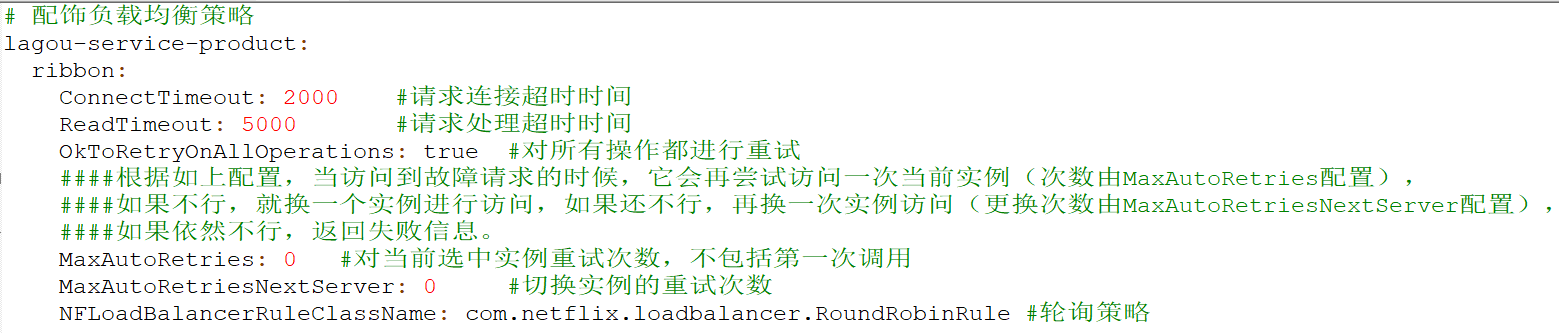
2）接口中的接口方法，就好比是远程服务提供者Controller中的Hander方法（只不过如同本地调用了），那么在进行参数绑定的时，可以使用@PathVariable、@RequestParam、@RequestHeader 等，这也是OpenFeign对SpringMVC注解的支持，但是需要注意value必须设置，否则会抛出异常

3) @FeignClient(name = "lagou-service-product")，name在消费者微服务中只能出现一次。（升级Spring Boot 2.1.0 Spring Cloud Greenwich.M1 版本后，在2个Feign接口类内定义相同的名字， **@FeignClient(name = 相同的名字就会出现报错，在之前的版本不会提示报错），所以最好将调用一个微服务的信息都定义在一个Feign接口中。**

**4.3 Feign对负载均衡的支持**

Feign本身已经集成了Ribbon依赖和自动配置，因此我们不需要额外引入依赖，可以通过ribbon.xx来进行全局配置,也可以通过服务名.ribbon.xx来对指定服务进行细节配置配置

Feign默认的请求处理超时时长1s，有时候我们的业务确实执行的需要一定时间，那么这个时候，我们就需要调整请求处理超时时长，Feign自己有超时设置，如果配置Ribbon的超时，则会以Ribbon的为准



**4.4 Feign对熔断器的支持**

在Feign客户端工程配置文件（application.yml）中开启Feign对熔断器的支持



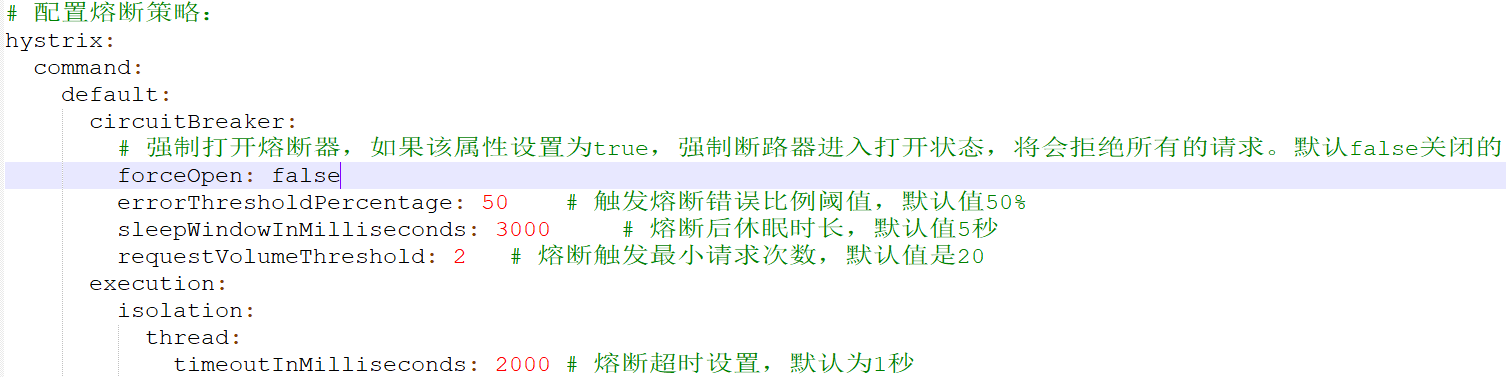
Feign的超时时长设置那其实就上面Ribbon的超时时长设置

Hystrix超时设置（就按照之前Hystrix设置的方式就OK了）

**注意：**

1）开启Hystrix之后，Feign中的方法都会被进行一个管理了，一旦出现问题就进入对应的回退逻辑处理

2）针对超时这一点，当前有两个超时时间设置（Feign/hystrix），**熔断的时候是根据这两个时间的最小值来进行的**，即处理时长超过最短的那个超时时间了就熔断进入回退降级逻辑



**回退降级类需要实现之前定义的feign接口，并且在此类上使用@Component注解**将其实例交给spring管理

在feign接口上的@FeignClient注解中添加fallback属性，值为实现feign接口的实现类的字节码文件**@FeignClient(name = "lagou-service-product",fallback = ProductFeignFallBack.class)**

**4.5 Feign对请求压缩和响应压缩的支持**

Feign 支持对请求和响应进行GZIP压缩，以减少通信过程中的性能损耗。通过下面的参数 即可开启请求与响应的压缩功能：



**第 5 节 GateWay网关组件**

网关：微服务架构中的重要组成部分

局域网中就有网关这个概念，局域网接收或者发送数据出去通过这个网关，比如用Vmware虚拟机软件搭建虚拟机集群的时候，往往我们需要选择IP段中的一个IP作为网关地址。

**5.1 GateWay简介**

Spring Cloud GateWay是Spring Cloud的一个全新项目，目标是取代Netflix Zuul，它基于Spring5.0+SpringBoot2.0+WebFlux（基于高性能的Reactor模式响应式通信框架Netty，异步非阻塞模 型）等技术开发，性能高于Zuul，官方测试，GateWay是Zuul的1.6倍，旨在为微服务架构提供一种简单有效的统一的API路由管理方式。

Spring Cloud GateWay不仅提供统一的路由方式（反向代理）并且基于 Filter(定义过滤器对请求过滤，完成一些功能) 链的方式提供了网关基本的功能，例如：鉴权、流量控制、熔断、路径重写、日志监控等。

**5.2 GateWay核心概念**

Spring Cloud GateWay天生就是异步非阻塞的，基于Reactor模型（同步非阻塞的I/O多路复用机制）

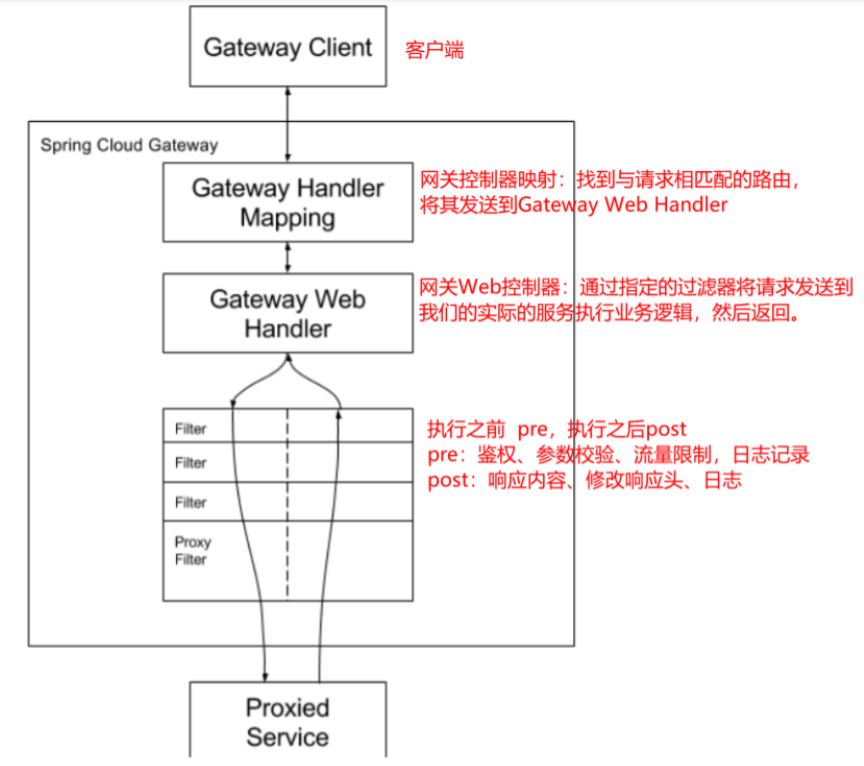
一个请求—>网关根据一定的条件匹配—匹配成功之后可以将请求转发到指定的服务地址；而在这个过程中，我们可以进行一些比较具体的控制（限流、日志、黑白名单）

**路由(route)**： 网关最基础的部分，也是网关比较基础的工作单元。路由由一个ID、一个目标URL(最终路由到的地址)、一系列的断言(匹配条件判断)和Filter过滤器(精细化控制)组成。如果断言为true，则匹配该路由。

**断言(predicates)**：参考了Java8中的断言java.util.function.Predicate，开发人员可以匹配Http请求中的所有内容(包括请求头、请求参数等)(类似于nginx中的location匹配一样)，如果断言与请求相匹配则路由。

**过滤器(filter)**：一个标准的Spring webFilter，使用过滤器，可以在请求之前或者之后执行业务逻辑。

**5.3 GateWay如何工作**



Spring 官方介绍：

客户端向Spring Cloud GateWay发出请求，然后在GateWay Handler Mapping中找到与请求相匹配的路由，将其发送到GateWay Web Handler；Handler再通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前（pre）或者之后（post）执行业务逻辑。

Filter在“pre”类型过滤器中可以做参数校验、权限校验、流量监控、日志输出、协议转换等，在“post”类型的过滤器中可以做响应内容、响应头的修改、日志的输出、流量监控等。

**5.4 GateWay应用**

**注意：GateWay不要引入starter-web模块，需要引入web-flux**（**类似于SpringMVC**）

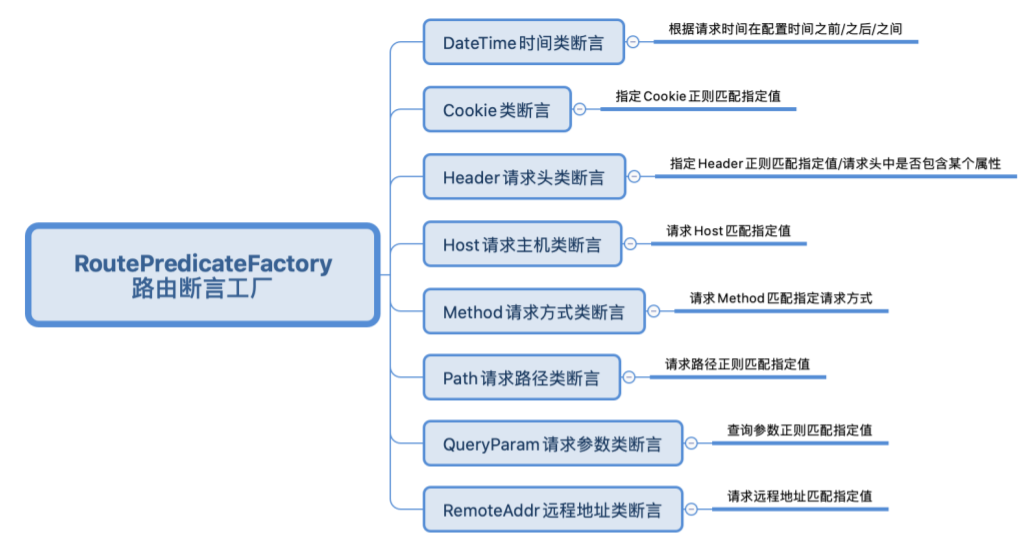




**注意：动态路由设置时，uri以 lb: //开头（lb代表从注册中心获取服务），后面是需要转发到的服务名称**

**5.5 GateWay路由规则详解**

Spring Cloud GateWay 帮我们内置了很多 Predicates功能，实现了各种路由匹配规则（通过Header、请求参数等作为条件）匹配到对应的路由。



**时间点后匹配**



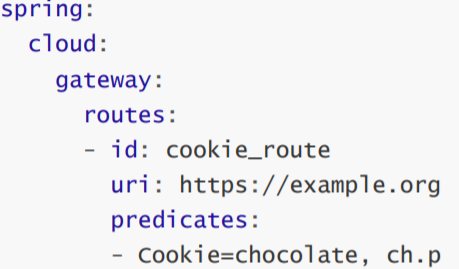
**时间点前匹配**



**时间区间匹配**



**指定Cookie正则匹配指定值**

****

**指定Header正则匹配指定值**



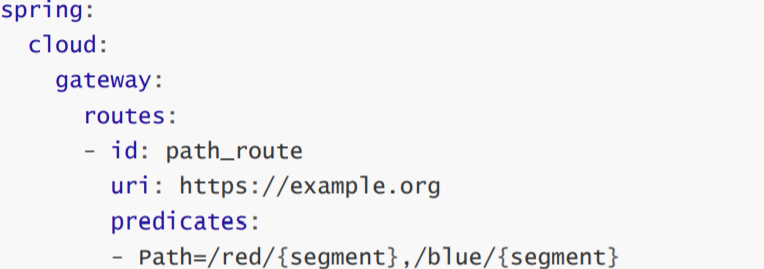
**请求Host匹配指定值**



**请求Method匹配指定请求方式**



**请求路径正则匹配**



**请求包含某参数**



**请求包含某参数并且参数值匹配正则表达式**



**远程地址匹配**



**5.6 GateWay过滤器**

5.6.1 GateWay过滤器简介

从过滤器生命周期（影响时机点）的角度来说，主要有两个pre和post：

|  |  |
| --- | --- |
| pre | 这种过滤器在请求被路由之前调用。我们可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。 |
| post | 这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。 |

从过滤器类型的角度，Spring Cloud GateWay的过滤器分为GateWayFilter和GlobalFilter两种

|  |  |
| --- | --- |
| GateWayFilter | 应用到单个路由路由上 |
| GlobalFilter | 应用到所有的路由上 |

如Gateway Filter可以去掉url中的占位后转发路由

**注意：GlobalFilter全局过滤器是程序员使用比较多的过滤器**

5.6.2 自定义全局过滤器实现IP访问限制（黑白名单）

自定义GateWay全局过滤器时，我们实现**GlobalFilter接口**即可，通过全局过滤器可以实现黑白名单、限流等功能。重写public Mono filter(ServerWebExchange exchange, GatewayFilterChain chain) 过滤器核心方法；**exchange参数封装了request和response对象的上下文，chain 网关过滤器链（包含全局过滤器和单路由过滤器）**

自定义GateWay全局过滤器时实现**Ordered接口**可以指定过滤器的优先级，重写getOrder方法的返回值表示当前过滤器的顺序(优先级)，数值越小，优先级越高

**5.8 GateWay高可用**

GateWay的高可用很简单：可以启动多个GateWay实例来实现高可用，在GateWay的上游使用Nginx等负载均衡设备进行负载转发以达到高可用的目的。在nginx的配置文件中配置如下：



**第 6 节 Spring Cloud Config 分布式配置中心**

**6.1 分布式配置中心应用场景**

单体应用架构，配置信息的管理、维护并不会显得特别麻烦，手动操作就可以，因为就一个工程；

微服务架构，因为我们的分布式集群环境中可能有很多个微服务，我们不可能一个一个去修改配置然后重启生效，在一定场景下我们还需要在运行期间动态调整配置信息，比如：根据各个微服务的负载情况，动态调整数据源连接池大小，我们希望配置内容发生变化的时候，微服务可以自动更新。

1）集中配置管理，一个微服务架构中可能有成百上千个微服务，所以集中配置管理是很重要的（一次修改、到处生效）

2）不同环境不同配置，比如数据源配置在不同环境（开发dev,测试test,生产prod）中是不同的

3）运行期间可动态调整。例如，可根据各个微服务的负载情况，动态调整数据源连接池大小等配置修改后可自动更新

4）如配置内容发生变化，微服务可以自动更新配置

那么，我们就需要对配置文件进行**集中式管理**，这也是分布式配置中心的作用

**6.2 Spring Cloud Config**

6.2.1 Config简介

Spring Cloud Config是一个分布式配置管理方案，包含了Config Server端和Config Client端两个部分。

Server 端：提供配置文件的存储、以接口的形式将配置文件的内容提供出去，通过使用 @EnableConfigServer注解在 Spring boot 应用中非常简单的嵌入

Client 端：通过接口获取配置数据并初始化自己的应用

6.2.2 Config分布式配置应用

**说明：Config Server是集中式的配置服务，用于集中管理应用程序各个环境下的配置。 默认使用 Git存储配置文件内容，也可以SVN。**

比如，我们要对“静态化微服务或者商品微服务”的application.yml进行管理（区分开发环境（dev）、测试环境(test)、生产环境(prod)）

1）登录GitHub

2）上传yml配置文件，命名规则如下：

{application}-{profile}.yml 或者 {application}-{profile}.properties

其中，application为应用名称，profile指的是环境（用于区分开发环境，测试环境、生产环境等）

3）构建Config Server统一配置中心

**在Config Server的启动类上使用注解@EnableConfigServer开启配置中心服务器功能**

**在Config Server的application.yml中配置git如下：**



4）构建Config Client客户端

**application.yml修改为bootstrap.yml配置文件**

bootstrap.yml是系统级别的，优先级比application.yml高，应用启动时会检查这个配置文件， 在这个配置文件中指定配置中心的服务地址，会自动拉取所有应用配置并且启用。

（主要是把与统一配置中心连接的配置信息放到bootstrap.yml）



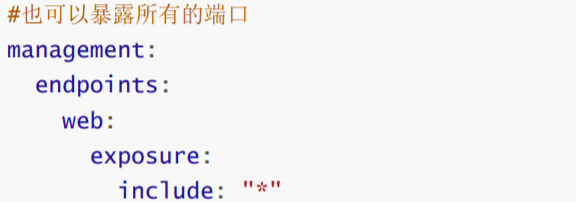
**6.3 Config配置手动刷新**

不用重启微服务，只需要手动的做一些其他的操作（访问一个地址/refresh）刷新，之后再访问即可

此时，客户端取到了配置中心的值，但当我们修改GitHub上面的值时，服务端（Config Server）能实时获取最新的值，但客户端（Config Client）读的是缓存，无法实时获取最新值。Spring Cloud已z经为我们解决了这个问题，那就是客户端使用post去触发refresh，获取最新数据。

1）Client客户端添加依赖springboot-starter-actuator

2）Client客户端bootstrap.yml中添加配置



3）Client客户端使用到配置信息的类上添加**@RefreshScope注解表示手动刷新**

4）手动向**Client**客户端发起**POST**请求，http://host:port**/actuator/refresh**，刷新配置信息

**6.4 Config配置自动更新**

在微服务架构中，我们可以结合消息总线（Bus）实现分布式配置的自动更新（Spring Cloud Config + Spring Cloud Bus）实现一次通知，处处生效

6.4.1 消息总线Bus

所谓消息总线Bus，即我们经常会使用MQ消息代理构建一个共用的Topic，通过这个Topic连接各个微服务实例，MQ广播的消息会被所有在注册中心的微服务实例监听和消费。**换言之就是通过一个主题连接各个微服务，打通脉络。**

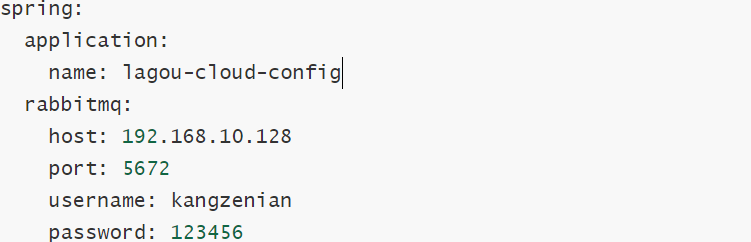
Spring Cloud Bus（基于MQ的，支持RabbitMq/Kafka）是Spring Cloud中的消息总线方案，Spring Cloud Config + Spring Cloud Bus结合可以实现配置信息的自动更新。

6.4.2 Spring Cloud Config + Spring Cloud Bus 实现自动刷新

MQ消息代理，我们还选择使用RabbitMQ，ConfigServer和ConfigClient都添加都消息总线的支持 以及与RabbitMq的连接信息

1）Config Server服务端和客户端添加消息总线支持bus-amqp

2）Config Server和客户端添加配置



3）Config Server微服务暴露端口

4）重启各个服务，更改配置之后，向**配置中心服务端发送post请求**，各个客户端配置即可自动刷新 http://host:port/actuator/bus-refresh

如果我只想定向更新，最后面跟上要定向刷新的实例的 服务名:端口号即可

<http://localhost:9006/actuator/bus-refresh/lagou-service-page:9100>

**注意：自动刷新也需要在Client客户端使用到配置信息的类上添加@RefreshScope注解**

**第五部分 第二代 Spring Cloud 核心组件（SCA）**

SpringCloud是若干个框架的集合，包括spring-cloud-config、spring-cloud-bus等近 20个子项目，提供了服务治理、服务网关、智能路由、负载均衡、断路器、监控跟踪、分布式消息队列、配置管理等领域的解决方案。Spring Cloud通过Spring Boot风格的封装，屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、容易部署的分布式系统开发工具包。一般来说，Spring Cloud包含以下组件，主要以Netflix开源为主。

同Spring Cloud一样，Spring Cloud Alibaba也是一套微服务解决方案，包含开发分布式应用微服务的必需组件，方便开发者通过Spring Cloud编程模型轻松使用这些组件来开发分布式应用服务。依托Spring Cloud Alibaba，您只需要添加一些注解和少量配置，就可以将Spring Cloud应用接入阿里微服务解决方案，通过阿里中间件来迅速搭建分布式应用系统。

**阿里开源组件**

Nacos：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

Sentinel：把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。

RocketMQ：开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。

Dubbo：这个就不用多说了，在国内应用非常广泛的一款高性能Java RPC框架。

Seata：阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。

Arthas：开源的Java动态追踪工具，基于字节码增强技术，功能非常强大。

**阿里商业化组件**

作为一家商业公司，阿里巴巴推出Spring Cloud Alibaba，很大程度上市希望通过抢占开发者生态，来帮助推广自家的云产品。所以在开源社区，夹带了不少私货，阿里商业化组件，整体易用性和稳定性还是很高的。

Alibaba Cloud ACM：一款在分布式架构环境中对应用配置进行集中管理和推送的应用配置中心产品。

Alibaba Cloud OSS：阿里云对象存储服务（Object Storage Service，简称OSS），是阿里云提供的云存储服务。

Alibaba Cloud SchedulerX：阿里中间件团队开发的一款分布式任务调度产品，提供秒级、精准的定时（基于Cron表达式）任务调度服务。

**集成 Spring Cloud 组件**

Spring Cloud Alibaba作为整套的微服务解决组件，只依靠目前阿里的开源组件是不够的，更多的是集成当前的社区组件，所以Spring Cloud Alibaba可以集成Zuul，GateWay等网关组件，也可继承 Ribbon、OpenFeign等组件。

**第 1 节 Nacos 服务注册和配置中心**

Nacos（Dynamic Naming and Configuration Service）是阿里巴巴开源的一个针对微服务架构中服务发现、配置管理和服务管理平台。

Nacos就是注册中心+配置中心的组合（Nacos=Eureka + Config + Bus）

官网：https://nacos.io 下载地址：https://github.com/alibaba/Nacos

**Nacos功能特性**

服务发现与健康检查

动态配置管理

动态DNS服务

服务和元数据管理（管理平台的角度，nacos也有一个ui页面，可以看到注册的服务及其实例信息（元数据信息）等），动态的服务权重调整、动态服务优雅下线，都可以去做

**1.2 Nacos 单例服务部署**

下载解压安装包，执行命令启动（我们使用最近比较稳定的版本 nacos-server-1.2.0.tar.gz）

linux/mac：sh startup.sh -m standalone

windows：cmd startup.cmd

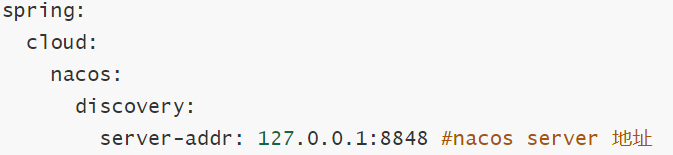
访问nacos控制台：http://127.0.0.1:8848/nacos/#/login 或者 <http://127.0.0.1:8848/> nacos/index.html（默认端口8848，账号和密码 nacos/nacos）

**1.3 微服务注册到Nacos**

1. 在父pom中引入SCA依赖

2. 在商品服务提供者工程中引入nacos客户端依赖，必须删除eureka-client依赖

3. application.yml修改，添加nacos配置信息



保护阈值：可以设置为0-1之间的浮点数，它其实是一个比例值（当前服务健康实例数/当前服务总实例数）

一般流程下，nacos是服务注册中心，服务消费者要从nacos获取某一个服务的可用实例信息，对于服务实例有健康/不健康状态之分，nacos在返回给消费者实例信息的时候，会返回健康实例。这个时候在一些高并发、大流量场景下会存在一定的问题

如果服务A有100个实例，98个实例都不健康了，只有2个实例是健康的，如果nacos只返回这两个健康实例的信息的话，那么后续消费者的请求将全部被分配到这两个实例，流量洪峰到来，2个健康的实例也扛不住了，整个服务A 就扛不住，上游的微服务也会导致崩溃，产生雪崩效应。

保护阈值的意义在于：当服务A健康实例数/总实例数 < 保护阈值 的时候，说明健康实例真的不多了，这个时候保护阈值会被触发（状态true）

nacos将会把该服务所有的实例信息（健康的+不健康的）全部提供给消费者，消费者可能访问到不健康的实例，请求失败，但这样也比造成雪崩要好，牺牲了一些请求，保证了整个系统的一个可用。

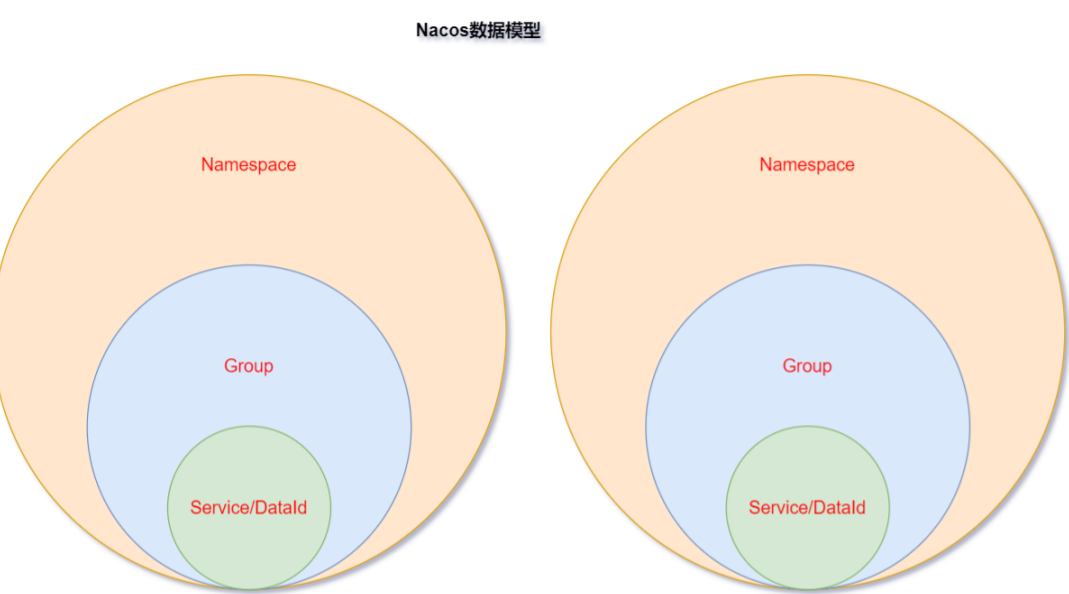
注意：阿里内部在使用nacos的时候，也经常调整这个保护阈值参数。

**1.4 负载均衡**

Nacos客户端引入的时候，会关联引入Ribbon的依赖包，我们使用OpenFiegn的时候也会引入Ribbon的依赖，Ribbon包括Hystrix都按原来方式进行配置即可

**1.5 Nacos 数据模型（领域模型）**

Namespace命名空间、Group分组、集群这些都是为了进行归类管理，把服务和配置文件进行归类，归类之后就可以实现一定的效果，比如隔离



Namespace：命名空间，对不同的环境进行隔离，比如隔离开发环境、测试环境和生产环境

Group：分组，将若干个服务或者若干个配置集归为一组，通常习惯一个系统归为一个组（拉勾招聘、拉勾猎头、拉勾教育）

Service：某一个服务，比如商品微服务

DataId：配置集或者可以认为是一个配置文件

**Namespace + Group + Service 如同 Maven 中的GAV坐标，GAV坐标是为了锁定Jar，而这里是为了锁定服务**

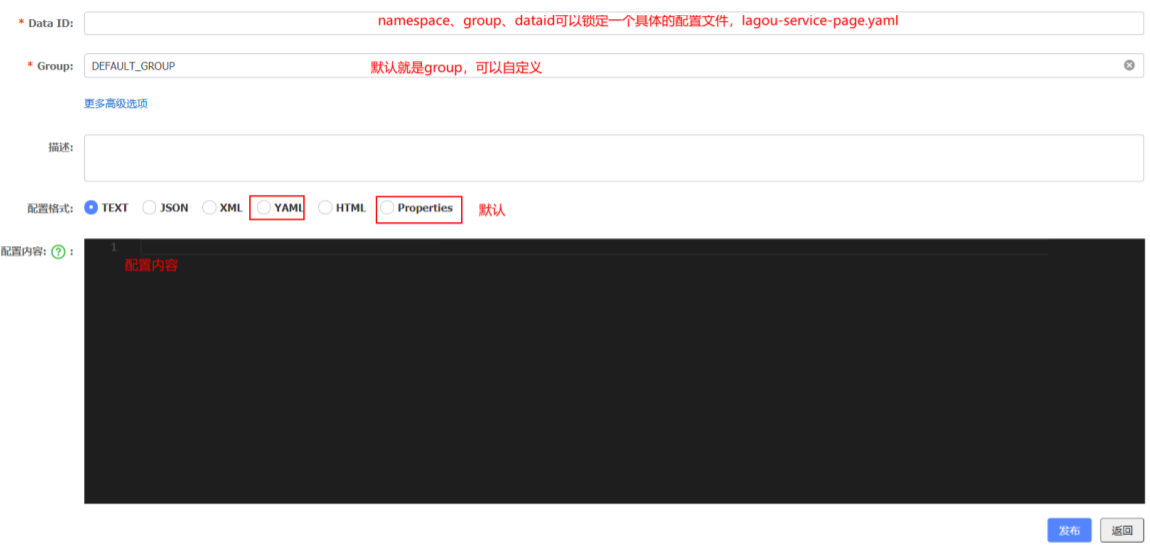
**Namespace + Group + DataId 如同 Maven 中的GAV坐标，GAV坐标是为了锁定Jar，而这里是为了锁定配置文件**

Nacos抽象出了Namespace、Group、Service、DataId等概念，具体代表什么取决于怎么用（非常灵活），推荐用法如下

|  |  |
| --- | --- |
| Namespace | 代表不同的环境，如开发dev、测试test、生产环境prod |
| Group | 代表某项目，比如拉勾云项目 |
| Service | 某个项目中具体xxx服务 |
| DataId | 某个项目中具体的xxx配置文件 |

**1.6 Nacos 配置中心**

1.6.1 Nacos Server添加配置



1.6.2 微服务中开启 Nacos 配置管理

1）添加依赖alibaba-nacos-config，将配置文件名改为bootstrap.yml

2）微服务中如何锁定 Nacos Server 中的配置文件（dataId）

通过 Namespace + Group + dataId 来锁定配置文件，Namespace不指定就默认public，Group不指定就默认 DEFAULT\_GROUP

**dataId的完整格式: ${prefix}-${spring.profile.active}.${file-extension}**

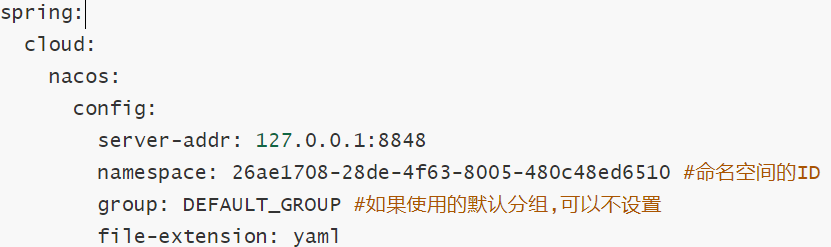
prefix 默认为 spring.application.name 的值，也可以通过配置项

spring.cloud.nacos.config.prefix 来配置。

spring.profile.active 即为当前环境对应的 profile。 **注意：当 spring.profile.active为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 ${prefix}.${file-extension}**

file-exetension 为配置内容的数据格式，可以通过配置项

spring.cloud.nacos.config.file-extension 来配置。目前只支持properties和yaml类型。



3）通过 Spring Cloud 原生注解 @RefreshScope 实现配置自动更新

4）一个微服务希望从配置中心Nacos server中获取多个dataId的配置信息，可以的，扩展多个 dataId



**第 2 节 SCA Sentinel 分布式系统的流量防卫兵**

**2.1 Sentinel 介绍**

Sentinel是一个面向云原生微服务的流量控制、熔断降级组件。

替代Hystrix，针对问题：服务雪崩、服务降级、服务熔断、服务限流

独立可部署Dashboard/控制台组件（其实就是一个jar文件，直接运行即可）

减少代码开发，通过UI界面配置即可完成细粒度控制

Sentinel 分为两个部分:

核心库：（Java客户端）不依赖任何框架/库，能够运行于所有 Java 运行时环境，同时对Dubbo/Spring Cloud等框架也有较好的支持。

控制台：（Dashboard）基于Spring Boot开发，打包后可以直接运行，不需要额外的 Tomcat 等应用容器。

Sentinel 具有以下特征:

**丰富的应用场景**：Sentinel承接了阿里巴巴近10年的双十一大促流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息削峰填谷、集群流量控制、实时熔断下游不可用应用等。

**完备的实时监控**：Sentinel 同时提供实时的监控功能。您可以在控制台中看到接入应用的单台机器秒级数据，甚至500台以下规模的集群的汇总运行情况。

**广泛的开源生态**：Sentinel提供开箱即用的与其它开源框架/库的整合模块，例如与Spring Cloud、Dubbo的整合。您只需要引入相应的依赖并进行简单的配置即可快速地接入 Sentinel。

**完善的SPI扩展点**：Sentinel提供简单易用、完善的SPI扩展接口。您可以通过实现扩展接口来快速地定制逻辑。例如定制规则管理、适配动态数据源等。

**2.2 Sentinel 部署**

下载地址：https://github.com/alibaba/Sentinel/releases 我们使用v1.7.1

启动：java -jar sentinel-dashboard-1.7.1.jar &

用户名/密码：sentinel/sentinel

**2.3 服务改造**

引入依赖: alibaba-sentinel

bootstrap.yml修改

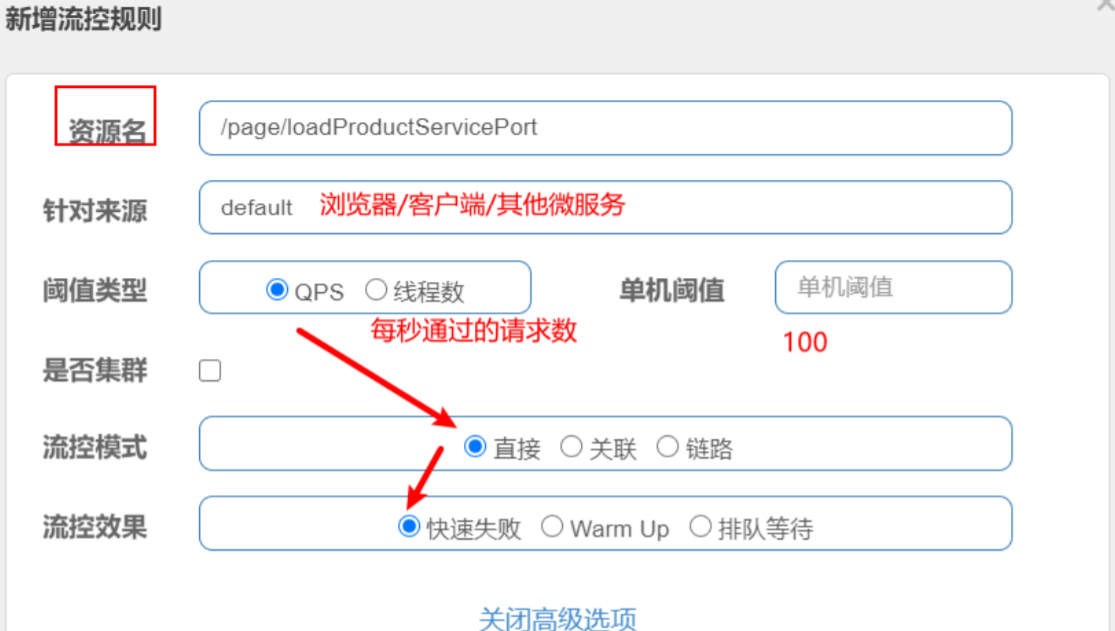


**2.4 Sentinel 关键概念**

|  |  |
| --- | --- |
| 资源 | 它可以是Java应用程序中的任何内容，例如，由应用程序提供的服务，或由应用程序调用的其它应用提供的服务，甚至可以是一段代码。**我们请求的API接口就是资源** |
| 规则 | 围绕资源的实时状态设定的规则，可以包括流量控制规则、熔断降级规则以及系统保护规则。所有规则可以动态实时调整。 |

**2.5 Sentinel 流量规则模块**

2.5 Sentinel 流量规则模块



资源名：默认请求路径

针对来源：Sentinel可以针对调用者进行限流，填写微服务名称，默认default（不区分来源）

阈值类型/单机阈值

QPS：（每秒钟请求数量）当调用该资源的QPS达到阈值时进行限流

线程数：当调用该资源的线程数达到阈值的时候进行限流（线程处理请求的时候，如果说业务逻辑执行时间很长，流量洪峰来临时，会耗费很多线程资源，这些线程资源会堆积，最终可能造成服务不可用，进一步上游服务不可用，最终可能服务雪崩）

是否集群：是否集群限流

流控模式：

直接：资源调用达到限流条件时，直接限流

关联：关联的资源调用达到阈值时候限流自己

链路：只记录指定链路上的流量

流控效果：

快速失败：直接失败，抛出异常

Warm Up：根据冷加载因子（默认3）的值，从阈值/冷加载因子，经过预热时长，才达到设置的QPS阈值

排队等待：匀速排队，让请求匀速通过，阈值类型必须设置为QPS，否则无效

**流控模式之关联限流**

关联的资源调用达到阈值时候限流自己，比如用户注册接口，需要调用身份证校验接口（往往身份证校验接口），如果身份证校验接口请求达到阈值，使用关联，可以对用户注册接口进行限流。

**流控模式之链路限流**

链路指的是请求链路（调用链：A-->B--C，D-->E-->C）

链路模式下会控制该资源所在的调用链路入口的流量。需要在规则中配置入口资源，即该调用链路入口的上下文名称。

**流控效果之Warm up**

当系统长期处于空闲的情况下，当流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮，比如电商网站的秒杀模块。

通过 Warm Up 模式（预热模式），让通过的流量缓慢增加，经过设置的预热时间以后，到达系统处理请求速率的设定值。

Warm Up模式默认会从设置的QPS阈值的1/3开始慢慢往上增加至QPS设置值。

**流控效果之排队等待**

排队等待模式下会严格控制请求通过的间隔时间，即请求会匀速通过，允许部分请求排队等待，通常用于消息队列削峰填谷等场景。需设置具体的超时时间，当计算的等待时间超过超时时间时请求就会被拒绝。

很多流量过来了，并不是直接拒绝请求，而是请求进行排队，一个一个匀速通过（处理），请求能等就等着被处理，不能等（等待时间>超时时间）就会被拒绝

例如，QPS 配置为5，则代表请求每200 ms才能通过一个，多出的请求将排队等待通过。超时时间代表最大排队时间，超出最大排队时间的请求将会直接被拒绝。排队等待模式下，QPS 设置值不要超过1000（请求间隔 1 ms）。

**2.6 Sentinel 降级规则模块**

流控是对外部来的大流量进行控制，熔断降级的视角是对内部问题进行处理。

Sentinel 降级会在调用链路中某个资源出现不稳定状态时（例如调用超时或异常比例升高），对这个资源的调用进行限制，让请求快速失败，避免影响到其它的资源而导致级联错误。当资源被降级后，在接下来的降级时间窗口之内，对该资源的调用都自动熔断，**这里的降级其实是Hystrix中的熔断**。

**策略:** Sentinel不会像Hystrix那样放过一个请求尝试自我修复，就是明明确确按照时间窗口来，熔断触发后，时间窗口内拒绝请求，时间窗口后就恢复。



**RT（平均响应时间 ）**

当1s内持续进入>=5 个请求，平均响应时间超过阈值（以ms为单位），那么在接下的时间窗口（以s为单位）之内，对这个方法的调用都会自动地熔断（抛出DegradeException）。注意Sentinel默认统计的RT上限是4900 ms，超出此阈值的都会算作4900 ms，若需要变更此上限可以通过启动配置项-Dcsp.sentinel.statistic.max.rt=xxx来配置。

**异常比例**

当资源的每秒请求量>= 5，并且每秒异常总数占通过量的比值超过阈值之后，资源进入降级状态，即在接下的时间窗口（以s为单位）之内，对这个方法的调用都会自动地返回。异常比率的阈值范围是 [0.0, 1.0]，代表0% - 100%。

**异常数**

当资源近1分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若timeWindow小于60s，则结束熔断状态后仍可能再进入熔断状态，时间窗口>= 60s。