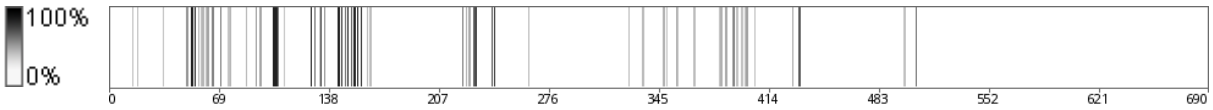
PaperPass旗舰版检测报告
简明打印版

比对结果(相似度):

总　体：10% (总体相似度是指本地库、互联网的综合对比结果)
本地库： 5% (本地库相似度是指论文与学术期刊、学位论文、会议论文、图书数据库的对比结果)
期刊库： 4% (期刊库相似度是指论文与学术期刊库的比对结果)
学位库： 2% (学位库相似度是指论文与学位论文库的比对结果)
会议库： 0% (会议库相似度是指论文与会议论文库的比对结果)
图书库： 1% (图书库相似度是指论文与图书库的比对结果)
互联网： 5% (互联网相似度是指论文与互联网资源的比对结果)

编　号：5B0529186F8719WUT
版　本：旗舰版
标　题：微服务
作　者：思路
长　度：20197字符(不计空格)
句子数：690 句
时　间：2018-5-23 16:40:56
比对库：学术期刊、学位论文、会议论文、书籍数据、互联网资源
查真伪：http://www.paperpass.com/check

句子相似度分布图:



本地库相似资源列表(学术期刊、学位论文、会议论文、书籍数据):

1.相似度： 1% 篇名：《电信运营商单体架构到微服务架构转型设计思路》
来源：学术期刊 《通信世界》 2017年25期
2.相似度： 1% 篇名：《微服务、单体应用以及NoOps》
来源：学术期刊 《电脑编程技巧与维护》 2016年11期
3.相似度： 1% 篇名：《基于容器的微服务架构的浅析》
来源：学术期刊 《信息系统工程》 2016年3期
4.相似度： 1% 篇名：《基于微服务架构的互联网+农业平台设计》
来源：学术期刊 《通信管理与技术》 2017年2期

互联网相似资源列表：

1.相似度： 4% 标题：《基于Spring Boot和Spring Clo...》
https://blog.csdn.net/jye666/article/details/54948924
2.相似度： 3% 标题：《spring cloud快速入门教程（一）总章...》
https://blog.csdn.net/tang86100/article/details/78890083
3.相似度： 3% 标题：《微框架之一：从零开始，轻松搞定SpringC...》
https://www.cnblogs.com/hyhnet/p/5626421.html
4.相似度： 2% 标题：《Spring Cloud--Honghu Clo...》

https://www.cnblogs.com/itspring/p/8978888.html
5.相似度： 2% 标题：《Spring Cloud -- 主要成员 - C...》
https://blog.csdn.net/u010926176/article/details/78814337
6.相似度： 2% 标题：《第5章 Spring Boot自动配置原理 - ...》
https://www.jianshu.com/p/346cac67bfcc
7.相似度： 1% 标题：《springcloud(一)：大话Spring ...》
https://blog.csdn.net/ityouknow/article/details/71077213
8.相似度： 1% 标题：《Spring Cloud - 简书》
https://www.jianshu.com/p/46fd0faecac1
9.相似度： 1% 标题：《从 Spring Cloud 开始，聊聊微服务架...》
http://hao.caibaojian.com/49229.html
10.相似度： 1% 标题：《单体应用与微服务优缺点辨析 - 程序园》
http://www.voidcn.com/article/p-bosscnxg-tr.html
11.相似度： 1% 标题：《《Spring Boot 实战：从0到1》第3章...》
https://blog.csdn.net/universsky2015/article/details/77965781

全文简明报告:

摘要

微服务的概念由Martin Fowler于2012年创造性的提出。 微服务提出是应用开发架构层面的创新，微服务的设计理念围绕在业务的模块分割、 应用的自动化部署、终端智能体验以及语言和数据的分散控制提出的一种新特性。 随着现在云计算技术发展和服务需求的增长，微服务设计思想也越来越多的受到了开发人员的关注。 微服务的理论也是对一套真的服务的虚拟化，云服务的设计思想。 尽管微服务这种开发思想很多开发人员持有不同的争议，但不可否认微服务这种系统架构模式正在为企业和个人提供高效开发能力，与良好的应用架构能力。

关键词 微服务 分布式 SpringCloud

微服务与单体服务

作为我们平常我们所开发系统中，最长听说的设计模式就是是MVC等即业务逻辑数据业务分离，但是这种设计模式再怎么划分业务。 对于日常的单体应用程序开。 针对于Java项目来说一个Web项目的最终产物就是一个运作在服务器的WAR文件。 我们那些划分的模块在源码阶段划分的不同逻辑在编译完成之后都会汇总到一个文件之中。 然后我们把项目部署在Tomcat 等servlet容器或者其他Java web服务器上，而另外一些Java应用会被打包成自包含的JAR格式。 但归根到底我们所开发系统是在一个应用服务器或者说是在一个进程中运行，我们称其这样的项目为单体应用。

{42%：微服务顾名思义，就是一个服务所占的资源或者应用大小相对传统单体应用十分微小。} 但其意思不是我们的整体应用的缩水，微服务的正真的影响是单个应用服务的简化，但整体上来说是对原有功能不做修改。 微服务真正讲究的是服务划分，区别于单体应用的模块划分，微服务的划分更加具体更加形象。 {42%：微服务架构的设计思想不是简简单单的开发一个复杂庞大的单个系统，而是将系统的原先的每个模块分解查分为单个简单的独立应用程序。} 系统的运行不再是单个进程内的模块调用，而是一个应用程序群，拆分后的应用程序通过 HTTP/ IP协议相互链接， 进而零散的子服务链接成一个完整的应用系统。

为什么选择微服务

单体服务的灵活问题：

单体应用程序每次修改都必须将修改后的整个应用重新编译打包部署。 协同开发时，我们只能在全部功能开发结束之后才能进行系统的部署应用。 这样有时后会延迟对客户交付系统的效率。 另一点单体应用在比较庞大情况下，系统的部署重启会有较大的时间消耗。故在在一些要求实时性非常高的系统面前这是一个不小的挑战； 另外一点是受技术限制，对于单体应用的开发，在开发时几乎只能选择一种开发技术，很难根据场景的不同选择合适的技术。 而且要使用的IDE 尽可能统一，这是对每个开发人员无形的技术限制。

单体服务技术陷阱：

在软件开发中“ 不坏不修 ”的思想在软件开发行业是非常常见，复杂单体应用这种情况尤为突出， 因为你不知道动了这一个模块会牵连到其他模块。 代码编完成后很难动态的去变更，因为应用程序的现存的一段逻辑代码在将来可能会以一种意外的方式去使用它。{41%：这种未来不可知使用可能对未来的应用是一种致命的错误。} 所有单体应用随着运行时间的增加，业务功能的不断添加、再加上开发人员变更， 技术陷阱会越来越多，这对应用程序的安全性与稳定性是一个潜在的威胁。

单体应用优点：

开发人员熟知：

单体服务从产生到现在经历了漫长岁月，以作为常识被开发人员接受，且现在大多数的开发工具都是针对传统的单体应用开发所设计。 有着完善的开发与调试环境支持。

编译产品简单：

单体系统编译后产品相当简答，可以简单的编译为单一文件。 进行人员之间共享。

系统易于测试：

单体应用在开发层面易于进行系统内的功能测试，在系统部署完成之后，系统全流程测试相对微服务架构系统来说，只有自身运行成功。 系统测试即完成。 不必依赖其他服务模块。

容易部署：

{55%：只需将单个打包文件复制到Web服务器中进行部署。} 几乎没有其他额外操作。

便于共享：

{97%：单个归档文件包含所有功能，便于在团队之间以及不同的部署阶段之间共享；}

易于系统测试：

{100%：单体应用一旦部署，所有的服务或特性就都可以使用了，这简化了测试过程，因为没有额外的依赖，每项测试都可以在部署完成后立刻开始；}

二、微服务六大优势

{46%：微服务架构相对于传统的单体应用对比在大系统常见的问题上面优势非常明显：}

复杂度可控：

{42%：微服务架构的每个子应用服务都是原本单体应用的具体的模块。} {45%：在模块划分的同时更好的降低了原系统的功能间耦合度。} 微服务中的每个应用都是对原先模块的更加具体逻辑的代码实现这样会有更加清晰的逻辑思路。 {41%：由于每个微服务架构中的每个应用相对原单体应用在体积、复杂度，均有明显降低。} {50%：所以单独的子服务更加易于管理，这样系统易于保持高可维护性和开发效率。}

独立部署：

微服务系统更大的差异性是模块不再是依赖与主应用程序在微服务系统中每个微服务应用都是一个独立的应用服务。 {60%：都是运行在不同的进程之中，所以每个微服务可以单独部署。} 当某个微服务一个提供者发生功能变更时无需对整个系统进行大规模的修改，仅需要对修改的服务进行系统维护即可。 可以理解为微服务系统架构的应用与单体应用 一个是动车与高铁。 微服务每个模块均有单独运行的能力而单体应用竟能在一个容器中跑，所有微服务架构使得单个系统的开发效率和质量有巨大的提升。

技术选型灵活：

{62%：在微服务架构下，技术选择不再是唯一。} 开发人员可以有更多的选择项，可以根据项目需求或者自身原因灵活变迁。 再加上微服务中每个子服务只是系统的小模块所以当需要对现有技术升级或者更换时带来的技术风险并不很高。 在允许情况下 我们可以将原有PHP的微服务组成模块替换为JAVA或者Node去实现也是未尝不可。

容错能力：

{41%：传统单体架构下的应用是在单一进程中运行，一个故障的发生可能应用程序中很快的传播，导致全局应用不可用。} {45%：然而在微服务架构下，每个应用是在不同的进程中运行，加之单个应用一般比较简单所以一个故障会被很容易隔离在单个服务之内。} 对整个系统的影响可以降到很低水平。 加之良好的架构体系。 可以大大的提升整个系统的容错能力。

系统扩展性：

相对于单体架构的横向扩展，通过模块之间的相互引用来实现应用扩展，微服务架构扩张更加轻松， 每个扩展就是一个单独的应用，不存在扩展依赖主应用的情况。

三、微服务三大挑战

固有的复杂性：

微服务架构系统有利必有弊。 {43%：相对与单体应用微服务应用属于分布式架构。} 分布式带来其固有的问题。 相对与单体应用进程之间的服务调用，微服务的远程过程调用（ RPC）或是服务之间的信息交互更加有技术困难， 对于开发人员来说这又是后可能涉及代码层次的调整来保证信息交互的完整，安全，及时。

微服务模块复杂性：

微服务的灵活扩展在有时后会是一个弊端，相对与单体服务虽有模块均在一个应用中，在如何修改也是对一个应用修改。 {53%：而对于微服务架构系统每个模块是一个独立的应用。} 在对系统架构变更时会有很大的变迁比如，假设在一个项目案例，使用微服务架构来构建，

项目内服务甲依赖服务乙，　但此时服务乙又对服务丙依赖，像这样来说项目的内的关系复杂性会很大的提升，对我们日常的开发维护是不利于的不良架构，　{50%：这种架构使服务之间的耦合性大大提升。}　比如，开发者更新服务丙，接下来是乙，最后才是甲，这无形是一种限制。　然而很幸运的是，在良好的微服务架构体系下许多改变要做到的是仅仅影响一个服务，向上文描述的这样的多级关联是不常见的。

SpringCloud　介绍

SpringCloud　是什么？

SpringClound　直译为中文为Spring云，恰似和符合这两年来流行的云技术。　但仅仅这样以为就错了，下面是SpringCloud官网为我们提供的英文版本的解释：

中文大致上是

{95%：Spring　Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如：}　{97%：配置中心，服务注册与发现，熔断器，路由，代理，控制总线，一次性令牌，全局锁，leader选举，分布式　会话，集群状态等）。}　{91%：　分布式系统的协调导致锅炉板模式，使用Spring　Cloud开发人员可以快速站出实现这些模式的服务和应用程序。}{64%：　他们可以在任何分布式环境中运行良好，包括开发人员自己的笔记本电脑，裸机数据中心和托管平台，如Cloud　Foundry（Cloud　Foundry是VMware推出云平台）。}

下面是SpringCloud　的架构基本图：

SpringCloud概念定义

Spring　Cloud是一套微服务框架而非云服务提供者，与阿里的　Dubbo框架等提供的远程过程调用框架，　{40%：　Spring　Cloud为开发者提供的整套的关于分布式系统问题的解决方案。}

Spring　Cloud非自己独立存在的框架它是对微服务基础框架　Netflix等多个开源组件进行了二次封装，　同时又实现了和云平台和原有的　Spring　Boot等成熟开发框架的集成后的产物。

Spring　Cloud的项目分析

Sping　Cloud是作为Spring组织的一个顶级项目，

下面是搜索Spring　Web开发中常见的顶级项目列表如下：

Spring　IO　platform:

主要用于系统部署方面，是可集成的模块，对于简单的构建现代化应用的版本平台来说是不错的选择，　通常我们使用　Maven等项目管理工具时，我们引入　Spring时就会作为依赖为我们自动引入。

Spring　Boot:

SpringBoot旨、时如今　Spring开源产品中最热的一款，它的目的简化原有SpringFramework的繁琐的配置，　为我们提供简单高效的开发框架，其中内置了比如Tomcat等几款　Servlet容器。　在项目发布时可以之间选择使用内置容器进行运行。

Spring Framework:

通常开发者最为了解的作为， Spring最早的开源产品，为开发人员提供全套开源的Java/ Java EE全功能栈应用程序框架， 其作为 Spring其他系列的基础而存在。

Spring XD： {100%：是一种运行时环境（服务器软件，非开发框架），组合spring技术，如spring batch、spring boot、spring data，采集大数据并处理。}

Spring Data：

{68%：是Spring为我们封装的一个简单数据操作的工具包，其中封装了对多种数据及数据库的操作相关技术，包括：} 常见的关系型jdbc、内存型Redis、文件型MongoDB等。

Spring Batch：

{41%：数据批处理框架，在日志等类似的数据处理方面的便捷工具。} {82%：其功能包括任务调度、系统日志记录/跟踪等。}

Spring Security： {56%：首先其基于 Spring框架，为企业应用安全系统提供声明式的安全访问控制，原为 Spring二级项目 Acegi Security，} 但随后其重要性别越来越发现呈现为 Spring现在的顶级项目。

6.4、Spring Cloud的子项目

Spring Cloud包含由很多子项目组成，如：

Spring Cloud Config：

提供项目常用的配置文件的管理工具，另外它支持使用 Git来存储配置文件，同样支持应用配置文件的外部化存储， 我们可以使用客户端来配置项目信息、加解密配置文件内容。实现项目配置信息脱离项目而存在。

Spring Cloud Bus： {100%：事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。}

Spring Cloud Netflix： {84%：针对多种Netflix组件提供的开发工具包，其中包括Eureka、Hystrix、Zuul、Archaius 注册中心 断路器等。}

Spring Cloud Sleuth： {63%：Spring提供系统日志收集模块，其为我们封装了包括Dapper，Zipkin和HTrace等操作。}

Spring Cloud Data Flow： {100%：大数据操作工具，通过命令行方式操作数据流。}

Spring Cloud Security： {71%：安全工具包集成SpringSecurity ，为你的应用程序添加安全控制，主要是指OAuth2。}

Spring Cloud Consul： {100%：封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。}

Spring Cloud Stream： {100%：数据流操作开发包，封装了

与Redis，Rabbit、Kafka等发送接收消息。}

Spring Cloud CLI: {85%：其基于 Spring Boot CLI，可以使开发者以命令行方式快速建立云组件。}

项目组织架构图 一般为

使用SpringCloud构建微服务应用

组件依赖

{42%: Spring 作为一个全球知名开源框架，为企业和个人在应用开发提供方便高效的项目构架工具。} Spring框架的的优势之一是其基于模块开发，除了基础的个别模块是构建项目的必须的，其他模块可以完全做到热插拔是动态的组合。 {43%：正是因为Spring分层架构允许开发者选择使用哪一个组件，Spring 项目可以动态开发Java EE 范围的任何模块，并且可以很好的和Spring core模块很好的匹配。} 不但如此，我们也可以开发相应的中间键来轻松结合那些非Spring社区的产品如Struts 等。

SpringCloud的实现

SpringCloud作为 SpringBoot的继承（ SpringBoot实在 Spring框架基础上的创新）推荐使用 Maven或者 Gradle来构建项目(极不推荐下载相应的 jar包来构建 SpringCloud依赖特别繁杂， 很难构建)自动引入相应的依赖。 这里推荐使用Maven作为项目的构建工具其实Gradle也不错 暂不演示。

这里是 SpringCloud官方推荐提供的1.5.2. RELEASE稳定版本(2.+官方现在是官方首推荐) Pom. xml的配置在以后会填加相应的功能添加删除新的组件 Pom依赖。

[parent]

[groupId]org.springframework.boot[/groupId]

[artifactId]spring-boot-starter-parent[/artifactId]

[version]1.5.12.RELEASE[/version]

[relativePath /]

[! -- Spring Cloud 主板--]

[/parent]

[dependencies]

[! ---指定依赖--]

[dependency]

[groupId]org.springframework.boot[/groupId]

[artifactId]spring-boot-starter-web[/artifactId]

[/dependency]

[dependency]

[groupId]org.springframework.boot[/groupId]

[artifactId]spring-boot-starter-test[/artifactId]

[scope]test[/scope]

[/dependency]

[/dependencies]

[build]

[! ---Maven 编译配置--]

[plugins]

[plugin]

[groupId]org.springframework.boot[/groupId]

[artifactId]spring-boot-maven-plugin[/artifactId]

[/plugin]

[plugin]

[groupId]org.apache.maven.plugins[/groupId]

[artifactId]maven-compiler-plugin[/artifactId]

[configuration]

[source]1.8[/source]

[target]1.8[/target]

[/configuration]

[/plugin]

[plugin]

[groupId]org.springframework.boot[/groupId]

[artifactId]spring-boot-maven-plugin[/artifactId]

[configuration]

[jvmArguments]-Dfile.encoding=UTF-8[/jvmArguments]

[/configuration]

[/plugin]

[/plugins]

[defaultGoal]compile[/defaultGoal]

[/build]

项目配置

Spring Boot 是由 Pivotal 团队提供的Spring顶级项目框架，其设计目的是用来简化原来的Spring构建项目的繁琐。 SpringBoot推荐开发人员使用了 Java来进行配置摒弃开发人员习惯的 XML配置文件，并且 SpringBoot事先为我们封装了大量的开发人员常用的配置， {63%：从而使开发人员不再需要配置繁琐的配置文件。}

SpringBoot现在正日益成为许多中小企业开发众多的开发人员首选的框架。 {45%：成为蓬勃发展的快速应用开发领域领导者。}

Spring Boot可以为我们创建独立的 Spring 应用程序另外SpringBoot内置嵌入的Tomcat，故我们可以不用打包JavaEE项目成 WAR 文件，直接在内置Servle容器中运行

{49%：另外SpringBoot 为开发人员简化 Maven 配置自动配置 Spring提供生产就绪型功能，如指标，健康检查和外部配置开箱即用，没有代码生成，也无需 XML 配置。}

3.特性理解

SpringBoot无需 XML配置主推 JAVA配置主要的配置信息使用 properties或者YAML配置文件来配置，另外 SpringBoot也提供了一些在大型项目中开发人员常见的非功能需求特性， {73%：如内置服务器、安全、指标，健康检测等功能。}

{85%：Spring Boot 并不是对 Spring 功能上的增强，而是提供了一种快速使用 Spring 的方式。}

SpringCloud是基于SpringBoot的继承开发，故在这方面SpringCloud继承SpringBoot的优点

SpringCloud的配置与SpringBoot无明显差异SpringBoot的自动化配置原理，

其实SpringCloud 仅仅是对基于SpringBoot为主题加之其他成熟开源产品的 的基本封装 。 但与此同时SpringCloud 为我们提供了一些新的特性。 如启动类注解，我们除了可以使用@ SpringBootApplication也可以使用@ SpringCloudApplication但擦看源码看出此注解只是简单的对原注解的简单的组合， 在创新性上没有体现，区别主要体现在 Cloud的注解除了依赖 SpringBootApplication后还依赖服务服务注册中心客户端和断路器的注解所以如果没有添加其注解的 Maven其他模块依赖我们可以使用仅@ SpringBootApplication来启动项目

通过使用@ EnableAutoConfiguration注解在 SpringBoot的启动类时，启用Spring应用程序时会自动为我们配置绝大多数 Spring相关的上下文的自动配置， 因为这

个注解会导入一个 EnableAutoConfigurationImportSelector的类，而这个类会主动去读取在 classpath下所有的 spring. factories配置文件且 key值为EnableAutoConfiguration对应的全限定名的值。

{74%： spring. factories配置文件里边里面配置的那些类，主要作用是告诉Spring Boot这个启动类所需要加载的那些 xxxAutoConfiguration类（其实这个类的作用就是定义 Bean，} 注册 Bean，就像我们 xml配置 bean一样），也就是你真正的要自动注册的那些 bean或功能。 {82%：然后，我们实现一个spring.factories指定的类，标上@Configuration注解，这样就形成了一个启动类}

服务注册与发现

Eureka使用

微服务常使用不仅仅提供Eureka这一种，模块作为 Springcloud的服务注册中心，于此同时还有ZooKeeper等其他的注册中心可以选择。 但是 ZK但是有相应的缺陷故不常使用（相应的问题此处不做具体解释），服务发现框架 Eureka原是有 Netflix维护开发的一款开源产品， 其支持常见的 HTTP连接，支持使用 REST风格的链接，在整个系统中其到服务注册发现以及注册服务的信息维护作用。 spring-cloud-netflix将它集成在SpringCloud的封装产物，为SpringCloud提供微服务发现功能。

使用及其简单 首先引入Eureka的Maven依赖

服务端依赖：

[dependency]

[groupId]org.springframework.cloud[/groupId]

[artifactId]spring-cloud-starter-eureka-server[/artifactId]

[/dependency]

后在其启动类注解在启动类使用@EnableEurekaServer 表示我开启Eureka的功能

@EnableEurekaServer

@SpringBootApplication

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

{44%：最后在配置文件中配置我们的Eureka 服务器相关关键性信息}

#=====================================

```
#  配置项目名已经端口
#=======================================
spring:
application:
name:  eureak-server
server:
port:  890
#=======================================
#  配置eureka的基本信息
#=======================================
eureka:
instance:
hostname:  localhost
client:
serviceUrl:
defaultZone:  http:  //${eureka.instance.hostname}:  890/eureka/
#  defaultZone:  http:  //${eureka.instance.hostname}:  891/eureka/
fetch-registry:  false  #
register-with-eureka:  false  #  但Eureka  不需要开启  集群是开启
server:
#  本地调试关闭保护机制
enable-self-preservation:  true  #  默认注册地址
```

那么Eureka Server 构建就完成 接下来SpringCloud 为我们完成接下来所有的事情

在客户端相似的配置

引入客户端依赖：

```
[dependency]
```

[groupId]org.springframework.cloud[/groupId]

[artifactId]spring-cloud-starter-eureka[/artifactId]

[/dependency]

在启动类使用注解 : @EnableEurekaClient 标记开启服务注册

```
@SpringBootApplication
@EnableEurekaClient
public class ServiceHiApplication {
public static void main(String[] args) {
SpringApplication.run(ServiceHiApplication.class, args);
}
}
```

相应配置文件：

```
eureka:
client:
serviceUrl:
defaultZone: http: //localhost: 890/eureka/
```

访问服务器地址结果如图:

Eureka原理

spring-cloud-eureka 它基于原有的nettfix-eureka进行二次封装，简化了配置，使其无障碍的在SpringCloud 框架中使用。 {43%：其主要作用是担任在微服务架构中的各个微服务的注册检查。} 相应信息的推送等。

EurekaServer都会有两个功能： 一个是服务注册、另一个是服务发现。 项目中通常会有一到多个个注册中心，系统中每个服务单元都要向EurekaServer注册自己信息。EurekaServer会将新的服务实例信息发送到其它依赖此微服务的消费者那里。

下面是一个简单的Eureka 功能图

服务注册

系统中的每个服务在启动的同时会自动将本身的服务信息推送的指定的服务注册中心上去，{43%：接下来服务注册中心收到服务提供者信息后，会将注册信息存储在一个双层结构的Map类型对象中，} 内层的 key是具体服务的实例名，外层的 key存储的是服务名。

服务同步

多数情况下会对服务中心进行集群防止某一个注册中心宕机后整个系统无法使用。 原理很简单就是一个服务注册者向其中一注册中心注册后，得到此注册信息的注册中心会向集群内的其它注册中心转发此服务注册者的注册信息， 从而实现在所有注册中心之间服信息同步共享.即使一个注册中心宕机后其他注册中心也会提供信息注与拉去访问。

服务续约

从常理来说什么事情都不会说永恒不变。 在服务注册完成成后，服务可能会因为自身或外界停止服务。 那么注册中心就不应该再维护我这服务的注册信息。 所有服务注册完成后，服务提供者与与注册中心存在一个周期性的信息交互使 Eureka Server(注册中心)知晓自己可以正常运行， 如果服务提供者长时间未和 Eureka Serve保持心跳通信，那么 EurekaServer将会把服务实例从服务列表中剔除。

服务下线

{46%：服务下线是服务提供者主动向 EurekaServer发起的服务信息提出请求，当服务实例不再提供服务时，} 首先它应该发送一条消息给注册中心告诉注册中心自己的服务将不能使用， EurekaServer收到信息后， {41%：将会把该服务实例标志为不可以，并此服务信息推送给其他注册中心或者服务提供者。}

获取服务

单体应用模块之间的调用时基于进程内之间调用，起到管控作用的是操作系统，无需外界进行相应的管理。 但微服务之间模块之间时服务与服务之间即进程之间调用。 注册服务甲的某项行为时依赖与服务乙时，此时就需要注册中心作中间媒介，服务甲作为服务消费者，应该发送一条包含请求调用服务相关的请求信息给注册中心，注册中心收到信息后会在自己维护的服务注册列表寻找合适的服务提供乙， {49%：将乙的信息作为返回信息返回给服务消费者服务甲。} {41%：接下来甲收到想要的信息，根据信息内容去找到真实服务提供者乙。}

自我保护

当一个注册服务长时间与注册中心进行心跳信息交互，即可认为这个注册服务出现问题那么即使剔除失效服务是必要的， 这样可以告诉这个失效的注册服务的消费者不要再访问这个错误的服务，但是在这之间应该还有一个逻辑。 当统计的注册服务在运行期间，心跳成功的比例低于一定的比例（可以设置心跳阈值），那么注册中心应该不能简简单单的删除全部的注册服务， 因为这可能涉及自身网络或者其他问题，简单的剔除未及时心跳交互的微服务。 可能是个错误。 Eureka为我们提供了一个策略就是会将当前服务实例的注册信息保护起来，并不是剔除这些实例。 这种情况就有一定的风险。 那就是万一是真的大面积注册服务提供者出现宕机，那么消费者获取的服务提供者的信息就是有误的。 {43%：这是调用服务提供者的服务就是有问题，所有客户端的容错机制是很必要的，} 这也时候 SpringCloud为我们提供吃的另一项便捷功能在下面将会提供介绍。 相比来说SpeingCloud提供功能是人性化的。 Eureka的安全机制是可关闭的不想使用自我保护机制，使用者仅仅在配置文件修改属性设置为 false即可， 这样可以确保心跳不及时或者有问题的服务注册者会及时被剔除。

源码分析

Eureka分为Client 与Server，也就是注册中心与服务提供者注册中心有JVM语言编写，Client根据不同的提供者的技术语言不同自行实现。 此处仅仅分析JAVA实现的Client;

JAVA版Eureka Client启动过程

相对于其他版本JAVA版EurekaClient 由于SpringCLoud 维护团队以及把它完全封装在SpringCloud套件中了。 仅需要我们引入相应是 Maven依赖，然后配置好EurekaServer的地址并在启动类上添加@ EnableDiscoveryClient注解， 这样在启动时才会去加载运行 Eureka相关的代码包括创建相关接口， 如关于服务注册，续约，以及下线等。 这些接口是用jersey发布的restful接口，资源类都在com.netflix.eureka.resources包下。

负载均衡

什么是负载均衡？

{40%：负载均衡(Load Balance)是现在应用系统中常用的模块，尤其在分布式应用系统设计中总要的考虑的因素之一，} {47%：通过负载均衡器可以将客户端请求/数据均匀分摊到后台服务器上去执行。}

下面常见的负载均衡方案：

为什么要负载均衡？

{49%：解决服务器高并发与大数据传输问题}

在某些特定的网站或者特定时期应用的应用高并发，海量数据的传输时。 我们可以利用高性能的服务器、数据库存储、物理存储设备、编程语言选择(比如NodeJs，Python，Go，)等方法来解决。 但是单个机器的物理服务器在物理性能已经无法解决技术问题。 {53%：服务器之间的负载均衡就是解决此问题的之一。}

{66%：解决分布式系统的统一入口与单点故障问题}

在原有的单体应用，几乎不存在之类问题，在单体应用被分解后，部署到不同的服务器上，可以实现系统的分布式化。 {41%：系统分布式化可以提升系统的稳定性与负载能力，但系统分布式仅仅是解决了系统的分布式。} 但系统分布式化也引入了登陆入口的问题，每个应用系统都应该有其登陆入口。 为了解决此问题，我们通常采用登陆入口冗余的策略，即将相同的应用部署到不同的服务器上， {41%：而后我们可以在集群应用前面使用负载均衡设备或软件，实现流量分发。}

3. 问题的终极解决方案

{41%：负载均衡的存在是解决单独服务器存在的资源分配不均衡问题，通过负载均衡设备可以将应用请求均衡的分摊到不同的设备上去。} {51%：既解决了资源分配不均衡有提高了设备的利用率。} 是解决应用的高性能、设备的高可用、系统扩展性(水平伸缩)的最终解决方案。

Spring Cloud 的负载均衡实现

客户端负载均衡： Spring Cloud Ribbon

{42%：Spring Cloud 工具包为我们提供Ribbon 是一个基于TCP/IP的协议的客户端负载均衡工具，它是基于Netflix Ribbon 的二次封装。} Ribbon的地位不同于服务注册中心、I网关等具有独立部署能力，而RIbbon模块作为微服务的最基本的插件而存在。 同时也包括前面的提供的声明式服务调用也是基于其实现的。 理解Ribbon模块对Spring Cloud来讲非常的重要，正是因为负载均衡对于一些大流量高并发系统提高其高可用和对网络压力的缓解与处理能力提升有显著效果。 Ribbon被作为最基础的模块被集成到SpringCloud之中，SpringCloud中通常采用了注解的方式来实现负载均衡。

客户端负载均衡示意图

SpringCloud对与此模块做了很人性化的设计，无需开发人员作任何配置即可使用，我们只需Eureka的依赖时候Ribbon作为依赖的模块就为我们自动引入。 且jar包引入会触发Spring Cloud在Eureka中关于对Ribbon的自动配置功能。

Feign实现Rest 调用

什么是Restful?

REST中文译为性状态转移（Representational State Transfer）。 这是Roy Fielding（Http协议的发起者之一） 在其博士论文提出的一观点，首先说明RestFul 并不是一种技术或者理论。 它仅仅对开发人员一种技术规范 对于任何符合REST的约束规范则，那么我们就可以称其为RESTful应用。 REST规范现在越来越在许多语言框架或者的万络交互中被广泛使用， REST的火热并不是因为它的技术上的突破， 而是他在理念上给人们打开了一个新的大门， Rest规范使用可以更好地使用现有 HTTP标准中的一些准则。虽然REST提出者的原本理念是REST应用范围不仅仅是在HTTP协议，但从现在来说Rest实现仅仅在HTTP协议。 依据现在RESTful架规范规定，REST充分使用HTTP协议设计之初的方法规范及充分利用HTTP除了POST，GET之外的其他方法，如： PUT，DELETE，PATCH等其他方法。

下面是这些方法设计之初的功能：

GET： 数据获取及从服务器获取出资源。

POST： 客户端向服务器推送资源。

PUT： {49%：客户端向服务器端更新资源但是要求客户端要提供完整的数据资源。}

PATCH： PATCH的功能与PUT类似但区别是PATCH并不要求提供完整的数据资源仅提交更新后的数据即可。

DELETE： {74%：客户端向服务器发送删除资源的请求。}

Feign 是什么？

Feign是一中注解式的网络请求客户端，它的使用简化我们对HTTP请求的使用复杂性使用开发者写Http客户端变得更简单。

Feign具有低侵略性，支持使用Feign 自身注解和JAX-RS注解，另外也可以使用SpringMVC的注解支持。 默认实现了对Ribbon的集成，无障碍与注册中心结合，同时为开发者实现了负载均衡。

SpringCloud 实现开启Fegin

首先 添加SpringCloud整合后Feigin的Maven依赖

[dependency]

[groupId]org.springframework.cloud[/groupId]

[artifactId]spring-cloud-starter-feign[/artifactId]

[/dependency]

在启动类上添加@EnableFeignClients注解开启SpringCloud对Feign的支持

```
@EnableDiscoveryClient
@EnableFeignClients
@SpringBootApplication
public class WebApplication {
public static void main(String[] args) {
logger.debug(”SERVER is Running”);
SpringApplication.run(WebApplication.class, args);
}
}
```

下面解释编写Feign访问接口即可：

```
@FeignClient(name = ”JAVA-PROVIDER”)
public interface SettingFeignInterface {
@RequestMapping(value = ”/updateNavs”, method = RequestMethod.POST)
int updateNav(@RequestParam(”nav”) String nav);
}
```

Feign注解常用属性：

name： 用于标识FeignClient将要访问的名称，通常服务中我们使用Ribbon 那么name属性将对应的是服务在EurekaServer上注册的实例名称

Value： 指定服务提供者的别名

url： 用于手动指定@FeignClient调用的URL地址，一般推荐用于调试

decode404： 调出现用服务资源不存在即常见的404错误时，该字段位true你将不会抛

出FeignException

fallback: 指定发生错误的回调类的名称，当调用远程接口失败或超时时，会调用对应接口的容错逻辑，

path: 定义FeignClient被调用是会被自动添加统一前缀

Feign的调用

```
@Autowired
private SettingFeignInterface settingFeign;
@RequestMapping(”/updateNavs”)
@ResponseBody
public CommonMessage updateNav(Navigation navigation) {
int result =
settingFeign.updateNav(JSONObject.toJSON(navigation).toString());
CommonMessage message = new CommonMessage();
if (result ] 0) {
message.setResult(true);
message.setMessage(”SUCCESS”);
}
return message;
}
```

使用Spring的IOC 自动注入和即可通过定义过的接口调用接口中定义的方法。 Feign会自动帮我们生成相应的接口的代理子类并通过SpringIOC自动传递过来。

服务容错性解决

为什么要服务容错

https: //tech.meituan.com/service-fault-tolerant-pattern.html

容错机制的设计是每个成熟系统中不可或缺的功能，每个系统，系统中的每个模块每个方法都有可能出现我们非预期的使用而产生异常， 错误产生是不可避免的而且无法做的详细设计处理，故容错处理模式也无法做到统一，所以开发人员要根据不同场景做出自己的选择，容错的目的是为了让服务提供者的故障不太大影响用户的使用体验，比如数据提供出现问题，我们可以给予友好的界面提示并且暂时禁用此功能，而不应该影响整个系统的其它正常模块的使用或者是整个系统完全崩溃。 这样也是系统设计模式也是符合系统设计要求健壮性要求使系统可以应对这个错误，并且具有一定恢复能力，如数据提供故障在可能在一段时间后修复

后， 系统对应的服务应该具有相应的感知能力是禁用掉的功能恢复正常使用.

{42%：熔断机制在微服务系统中尤其显得重要，相对于单体应用的单服务，微服务的架构系统中子服务之间存在着更为复杂的依赖关系，} 尤其处底层的服务几乎被所有的上层服务多所依赖，底层服务的故障失效对与微服务系统来说带来的是雪崩效应， 一个服务的失效牵涉到多个服务不可用，进一步导致整个系统崩溃不可用， 对于这个问题，服务容错处理方案的重要性和不可或缺型就显得格外重要了。

Spring Cloud 提供的容错方案

在Spring Cloud 中Hystrix模块为开发人员事先实现包括了线程隔离、断路器等一系列的服务容错能力。 Hystrix同样是基于 Netflix的开源产品二次封装 Hystrix实现的，该产品功能通过控制远程系统、服务和其他系统节点， {66%：从而对那些高延迟或者故障节点提供更强大的容错能力。} Hystrix模块具备了容错常用手段，包括对服务降级、熔断、以及线程隔离保护、访问请求缓存以及请求合并以及服务监控等功能。

SpringCloud开启容错模块首先需要引入相应的hystrix依赖。 如使用Maven 时的jar包依赖如下所示：

[dependency]

[groupId]org.springframework.cloud[/groupId]

[artifactId]spring-cloud-starter-hystrix[/artifactId]

[version]1.4.4.RELEASE[/version]

[/dependency]

同时在服务的启动类使用@EnableCircuitBreaker注解，开启断路器能力代码如下：

// 扫描 自定义 监听器 Servlet

@ServletComponentScan

@EnableDiscoveryClient

@EnableFeignClients

@EnableCircuitBreaker

@SpringBootApplication

public class WebApplication {

private static Logger logger = Logger.getLogger(WebApplication.class);

@Bean

@LoadBalanced

```
public RestTemplate restTemplate() {
return new RestTemplate();
}
public static void main(String[] args) {
logger.debug(”服务启动”);
SpringApplication.run(WebApplication.class, args);
}
}
```

前面我们事先说明， SpringCloud在为我们封装各个模块的同时，又为我创造了新的特性@ SpringCloudApplication标签是 对@ SpringBootApplication@ EnableCircuitBreaker@ EnableDiscoveryClient的集成简化注解使用量的同时使应该包括服务的注册发现以及容错的能力。

接下来指定相应的容错方法，这时候引入路前文我们提到注解@HystrixCommand(fallbackMethod = ”toMianError”) 的fallbackMethod 指定了方法调用出现问题后的回调方法如下文

```
// 2018/5/18 漏电保护器
@HystrixCommand(fallbackMethod = ”toMianError”)
// 201805 16 获取远端NODEjs 数据 首页的 巨幕信息
@RequestMapping(value = ”/”)
public String toMian(Model model, HttpServletRequest req) throws JSONException {
// 获取远端 数据 (NOde )
String result = temp.postForObject(nodeJsUrl + ”/main”, null, String.class);
Gson gson = new Gson();
JSONArray jsons = new JSONArray(result);
Main main = gson.fromJson(jsons.getJSONObject(0).toString(), Main.class);
List[ServiceInstance] serviceInstances = discoveryClient.getInstances(nodeJsNme);
for (ServiceInstance serviceInstance : serviceInstances) {
```

```
main.setM_bg_img(
”http: //” + serviceInstance.getHost() + ”: ” +
serviceInstance.getPort() + ”/” + main.getM_bg_img());
;
}
model.addAttribute(”json”, main);
return ”main”;
}
// 首页 获取信息 出错时 的方法
public String toMianError(Model model, HttpServletRequest req)
throws JSONException {
Main main = new Main();
main.setM_bt_desc(sorryDesc);
main.setM_bt_title(sorryTitle);
main.setM_bt_url(sorryUrl);
main.setM_title(sorryTitle);
main.setM_content(sorryDesc);
model.addAttribute(”json”, main);
return ”main”;
}
```

API网关服务

API网关是什么？

API网关是一个提供单独且统一的API入口的设备或者应用。 它用于访问内部一个或多个API接口。 它的通常作用位置时整个系统服务最外层的的入口。 用于充当外层服务与内部服务接口的代理角色。

微服务API网关的优势

保护内部的敏感信息避免暴露相关安全信息

API网关主要作用就是代理服务，通过服务代理功能即服务内部接口与 API网关代理的绑定， 对于调用者网关的接口是公开透明的，但真实的与内部微服务 API别隐蔽。 且

我们可以动态的绑定网关与内部的服务绑定，这样就实现了重构和裁切微服务的功能同时不会影响外部绑定的客户端的能力。

为服务增加额外的安全层

通常为了安全 API网关与外部使用安全协议进行交互，优点是能够有效保护信息的安全性， 而内部服务在隔离的安全环境使用明文传输，优点是传输效率高。 这样的架构能够抵御常见以及的安全攻击包括常见的SQL注入，XSS，会话欺骗和DDoS拒绝式服务工具攻击。

可以支持混合通讯协议

不同于外部API一般使用HTTP，内部数据的传输格式可以根据自身来定制。 相对与外界与API网关使用通常是HTTP协议，但内部服务之间是不直接与外界进行交互，所以内部服务之间可以指定一个合适的交互方式。

降低微服务的复杂度

系统微服务化带来的不仅仅好处，还有相应的问题。 单体系统认证以为其服务的单一，一次认证一般即可。 但微服务化后，系统是有众多服务组成，原先思路是为系统冗余，每个服务均添加认证模块。 但单来另一个问题是认证频率过高，在一定程度对系统的性能是一种损害你，冗余模块也是对系统资源的浪费。 由此 API网关为我提供了新的思路，最为系统唯一与外界交互的通道，相对与外部系统的安全不可知性， 内部系统通常会运行在安全的局域网内，所以完全可以将校验服务交给网关。 这样一来，性能上与资源利用上是很大的提升。

SpringCloud 使用服务网关Zuul

引入Maven依赖

[! -- https:
//mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-zuul
--]

[dependency]

[groupId]org.springframework.cloud[/groupId]

[artifactId]spring-cloud-starter-zuul[/artifactId]

[version]1.4.4.RELEASE[/version]

[/dependency]

在Application中开启Zuul

@SpringBootApplication

//开启Zuul代理

@EnableZuulProxy

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

配置配置文件 将自己作为一个服务注册到注册中心，接下来就可以通过访问ZUUL的url后添加对应的注册在注册中心的实例的名称来访问代理服务

```
server:
port: 880
spring:
application:
name: GATEWAY-ZUUL
eureka:
client:
service-url:
defaultZone: http: //localhost: 890/eureka/
instance:
prefer-ip-address: true
```

服务容错

代理服务也不能保证被代理的服务正常的运行， 前文的容错在任何模块都是必须的，所以 SpringCloud的维护团队实现考虑到把 Hystrix集成模块已经写到工具包中， 我们仅需如上文引入 Hystrix的依赖后。 开发人员只需配置错误发生的回调方法即可：

```
@Component
public class MyFallbackProvider implements FallbackProvider {
@Override
public String getRoute() {
// 表明是为哪个微服务提供回退，*表示为所有微服务提供回退
return ”*”;
```

```
}
@Override
public ClientHttpResponse fallbackResponse(Throwable cause) {
if (cause instanceof HystrixTimeoutException) {
return response(HttpStatus.GATEWAY_TIMEOUT);
} else {
return this.fallbackResponse();
}
}
@Override
public ClientHttpResponse fallbackResponse() {
return this.response(HttpStatus.INTERNAL_SERVER_ERROR);
}
private ClientHttpResponse response(final HttpStatus status) {
return new ClientHttpResponse() {
@Override
public HttpStatus getStatusCode() throws IOException {
return status;
}
@Override
public int getRawStatusCode() throws IOException {
return status.value();
}
@Override
public String getStatusText() throws IOException {
return status.getReasonPhrase();
}
```

```
@Override

public void close() {

}

@Override

public InputStream getBody() throws IOException {

return new ByteArrayInputStream(”我们的后台出问题了就是那种要杀程序员祭天的问题”.getBytes());

}

@Override

public HttpHeaders getHeaders() {

// headers设定

HttpHeaders headers = new HttpHeaders();

MediaType mt = new MediaType(”application”, ”json”, Charset.forName(”UTF-8”));

headers.setContentType(mt);

return headers;

}

};

}

}
```