SCM Refactoring Project

Based on Spring Boot and Spring Cloud

易果集团 云象供应链 - 李波涛

libotao@yiguo.com

Agenda

- Spring Boot/Spring Cloud overview；
- Why choose Spring Boot & Cloud?
- What is Cloud Native Application?
- 12 factors
- SCM Refactoring Project goal;
- SCM architecture based on Spring Cloud;
- Spring Cloud Core Components
  - ✓ Eureka – Service Discovery
  - ✓ Hystrix Circuit Breaker
  - ✓ Monitoring: Hystrix Dashboard
  - ✓ Ribbon – Client Side Load Balancer
  - ✓ Zuul – API Gateway
  - ✓ Distributed tracing with Sleuth and Zipkin
- Spring Cloud & .Net Core
- References

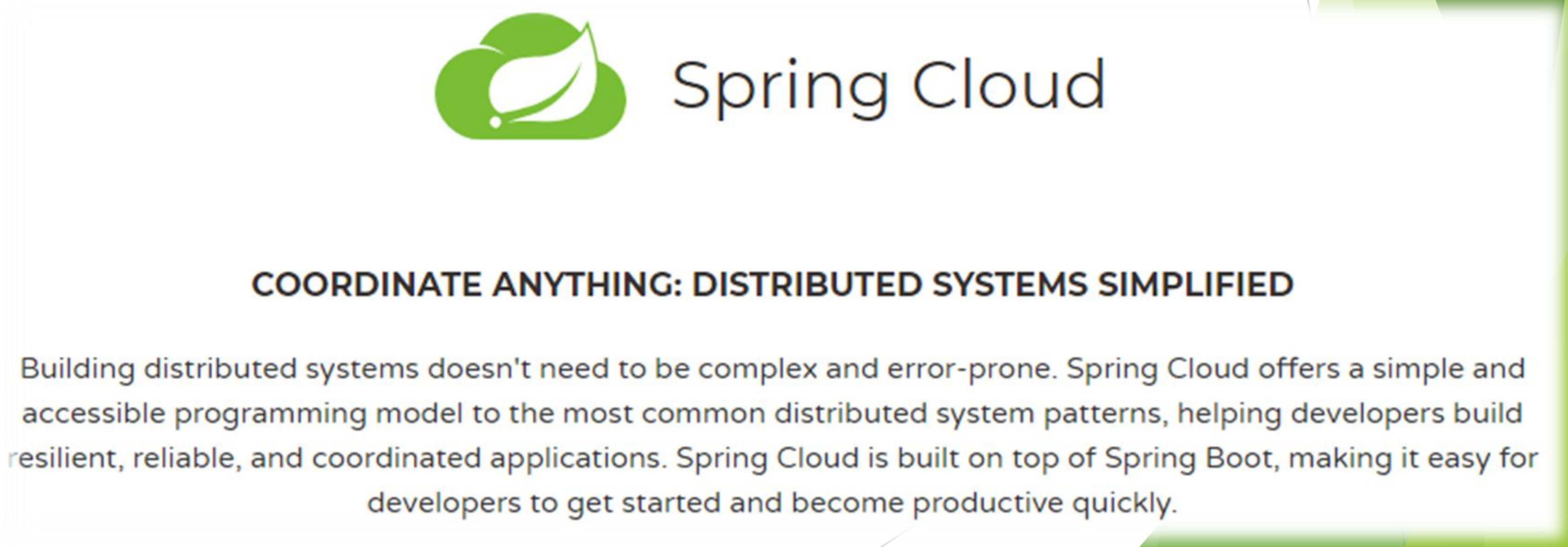
1
Spring Boot & Spring Cloud

What is Spring Boot?

- Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。
- Spring Boot 简化了基于 Spring 的应用开发，通过少量的代码就能创建一个独立的、产品级别的 Spring 应用。
- 简而言之，简化配置，面向微服务开发，提升开发效率。

What is Spring Cloud?

- Spring Cloud 虽然带有 “Cloud”字样，但不是云计算解决方案；
  - 是适合部署在云计算平台；
- 是在 Spring Boot 基础上构建的，用于快速构建分布式微服务架构；
- 非常适合在 Docker 或者 PaaS 上部署。


Spring Cloud

COORDINATE ANYTHING: DISTRIBUTED SYSTEMS SIMPLIFIED

Building distributed systems doesn't need to be complex and error-prone. Spring Cloud offers a simple and accessible programming model to the most common distributed system patterns, helping developers build resilient, reliable, and coordinated applications. Spring Cloud is built on top of Spring Boot, making it easy for developers to get started and become productive quickly.

Spring Cloud features

1. 开箱即用，可以提升开发效率；
2. 文档丰富，社区活跃，遇到问题比较容易获得支持；
3. 为微服务架构提供了完整的解决方案；
4. 轻量级组件，Spring Cloud 整合的组件大多比较轻量级，并且可插拔；
5. 组件丰富、功能齐全，比如提供了服务注册发现、断路器、微服务网关、负载均衡、数据监控、配置中心等等；

Spring 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 Boot和Spring Cloud 关系

- Spring Boot 简化配置，可以基于 Spring Boot 快速开发单个微服务应用。Spring Cloud 是一个基于 Spring Boot 实现的微服务治理项目集。
- Spring Boot 专注于快速、方便开发单个微服务；Spring Cloud 关注全局的微服务治理框架。
- Spring Boot和Spring Cloud 构成Java微服务实践的最佳落地方案。

Spring Cloud is built on top of Spring Boot

Spring Boot	Eureka Server	Spring Cloud-Microservices
• 注解（Annotation） • @RestController • @SpringBootApplication	• 注解（Annotation ） • @SpringBootApplication • @EnableEurekaServer • Dependencies and configuration	• 注解（Annotation ） • @RestController • @SpringBootApplication @EnableEurekaClient • Dependencies and configuration

Spring Cloud is built on top of Spring Boot, making it easy for developers to get started and become productive quickly.

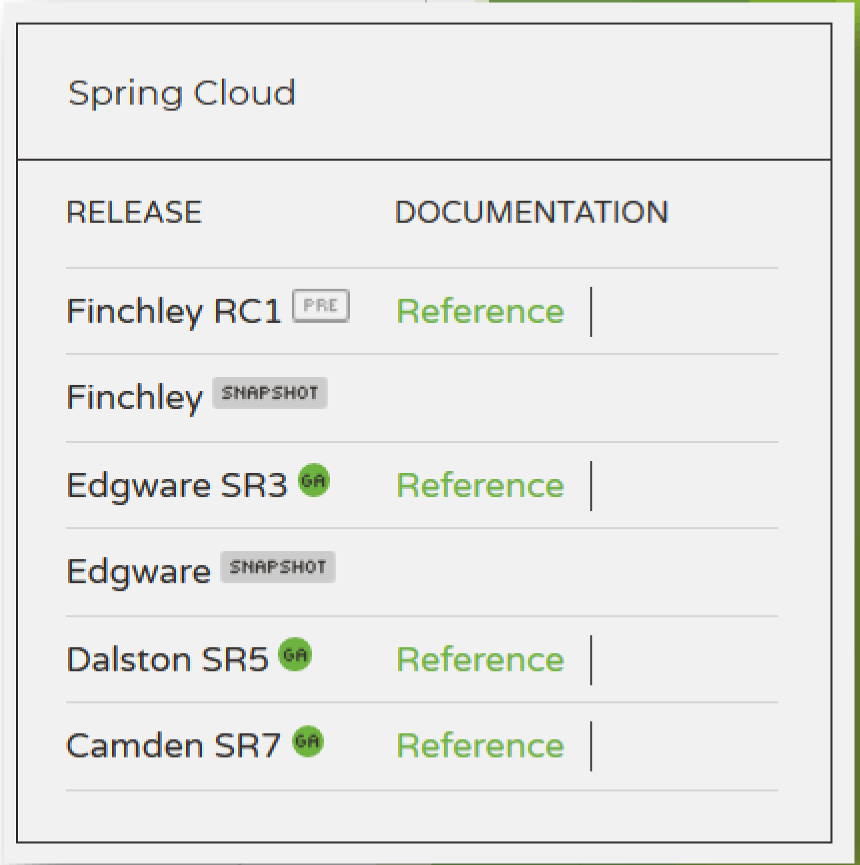
Why choose Spring Cloud?

1. Spring Cloud 分布式微服务完整解决方案，它包含了
   - ✓ 灵活的网关支持，方便对服务进行裁剪和聚合；
   - ✓ 支持客户端负载均衡和服务端负载均衡；
   - ✓ 单个接口级别的熔断器控制，可以通过熔断机制控制服务节点，从而对延迟和故障提供更强大的容错能力，防止雪崩效应；
   - ✓ 全链路跟踪支持；

Why choose Spring Cloud?

2. Spring Cloud 是现在Java领域最流行的开源的微服务解决方案，由Spring 官方持续更新升级维护，稳定性，可靠性有保证；

3. Spring Cloud 学习成本低，对业务代码无侵入性，开发人员只关注自己的业务即可，提升团队开发效率；



Spring Cloud

RELEASE	DOCUMENTATION
Finchley RC1 PRE	Reference
Finchley SNAPSHOT	
Edgware SR3 GA	Reference
Edgware SNAPSHOT	
Dalston SR5 GA	Reference
Camden SR7 GA	Reference

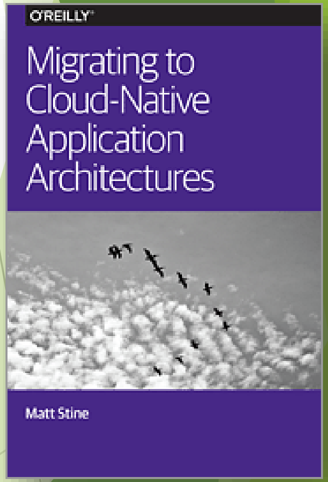
2
Cloud Native Application

What is Cloud-Native Application?

- A cloud-native application is a software application that is specifically built for cloud computing and virtualization environments. Cloud-native applications are designed, developed and deployed in such a way that they reap the maximum functionality and services of a cloud computing and virtualization infrastructure.

- 2015 年，Pivotal公司的Matt Stine提出Cloud Native这一概念；
- Matt Stine：free ebook - 《Migrating to Cloud Native Application Architectures》
- 中文版《迁移到云原生应用架构》：

https://github.com/rootsongjc/migrating-to-cloud-native-application-architectures

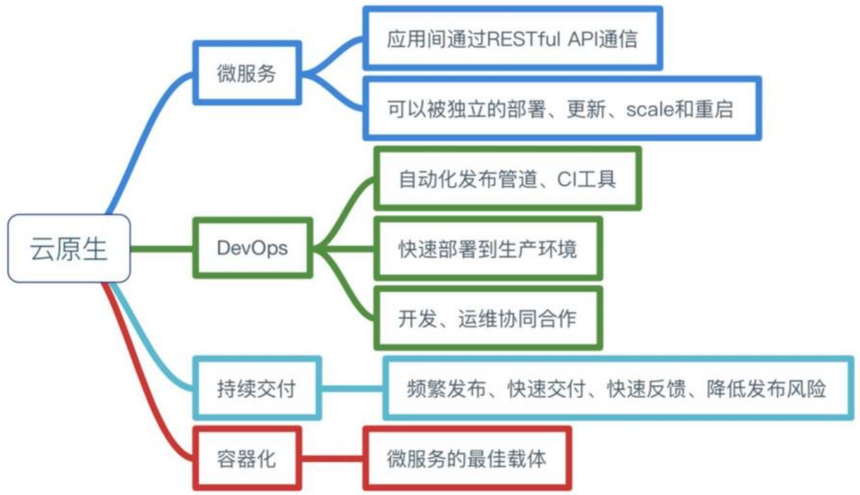


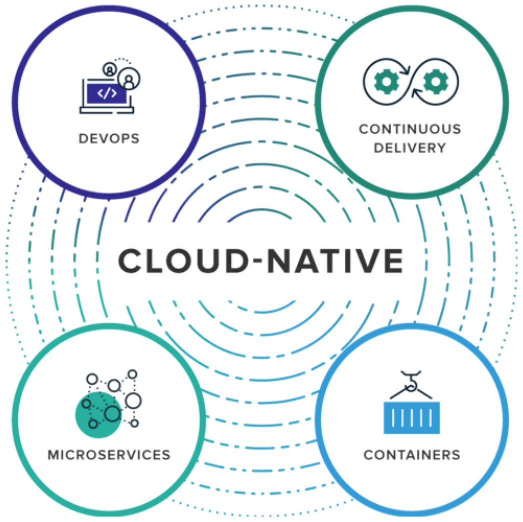
Cloud-native apps differ from traditional apps

- Cloud-native apps use an elastic infrastructure.
- Cloud-native apps need to provision instances of themselves through an API.
- Cloud-native apps must be able to scale up and down at a rapid rate, with the ability to offer thousands or even hundreds of thousands of nodes or instances.
- Cloud-native apps are autonomic in nature, with the ability to detect and work around failures, and if one or more nodes are lost, new nodes are spun up extremely quickly.

4 features of Cloud-Native applications

- 容器化封装(Docker)：以容器为基础，简化云原生应用程序的维护。在容器中运行应用程序和进程，并作为应用程序部署的独立单元，实现高水平资源隔离。
- 服务编排(Kubernetes)：通过集中式的编排调度系统来动态的管理和调度。
- 面向微服务化(Microservices Arch.)：明确服务间的依赖，互相解耦。😃
- 持续交付(Continuous Delivery)：频繁发布，快速交付。





12 factors-云原生应用开发指导原则

公有云PaaS的先驱Heroku于2012年提出：

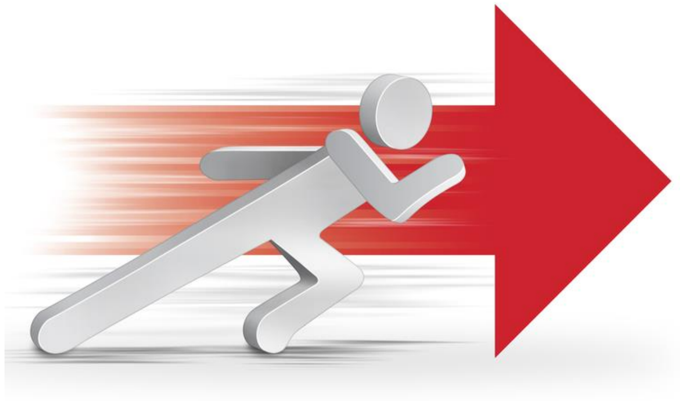
- 基准代码
- 显式声明依赖关系
- 在环境中存储配置
- 把后端服务当作附加资源
- 严格分离构建、发布和运行
- 无状态进程
- 通过端口绑定提供服务
- 通过进程模型进行扩展
- 快速启动和优雅终止
- 开发环境与线上环境等价
- 日志作为事件流
- 管理进程-以一次性进程(one-off process) 的形式运行后台管理任务

https://12factor.net/
https://12factor.net/zh_cn/

THE TWELVE-FACTOR APP

Spring Cloud and Cloud-Native app

- Spring Cloud - 应“云”而生，All in One的集合，快速搭建分布式微服务架构；
- 创建符合12 factors要素的cloud-native app；
- Spring Cloud 提供了一套创建cloud-native app的解决方案；



Spring Boot and Spring Cloud Accelerate Cloud-Native Java Applications Development.

Summary of Cloud-native apps

- 是一种软件开发方式?
- 在持续交付和价值驱动的软件开发领域...
- 鼓励方便快捷有效地实施最佳实践...
- ...的一种软件开发方式。
- ...也是一套技术体系和一套方法论。

面向云平台部署运行而设计开发的应用。真正的云化不仅是基础设施和平台的事情，应用也要做出改变，实现云化的应用 — 包括架构、开发方式、部署和维护等等都要做出改变，真正发挥云的弹性、动态调度、自动伸缩……特性。

Cloud Native is a style of application development that encourages easy adoption of best practices in the areas of continuous delivery and value-driven development.

Come back to SCM project

- SCM’s technical debt
- SCM refactoring project goal
- SCM Architecture based on Spring Cloud
- Core components from Spring Cloud

3

SCM

Current SCM’s Technical Debt

当前最大问题：SCM、WMS系统深度耦合；

- SCM 对WMS DB 直连访问；
- 对WMS DLL直接引用；
- 对WMS老框架的API调用；
- 其他奇葩耦合……

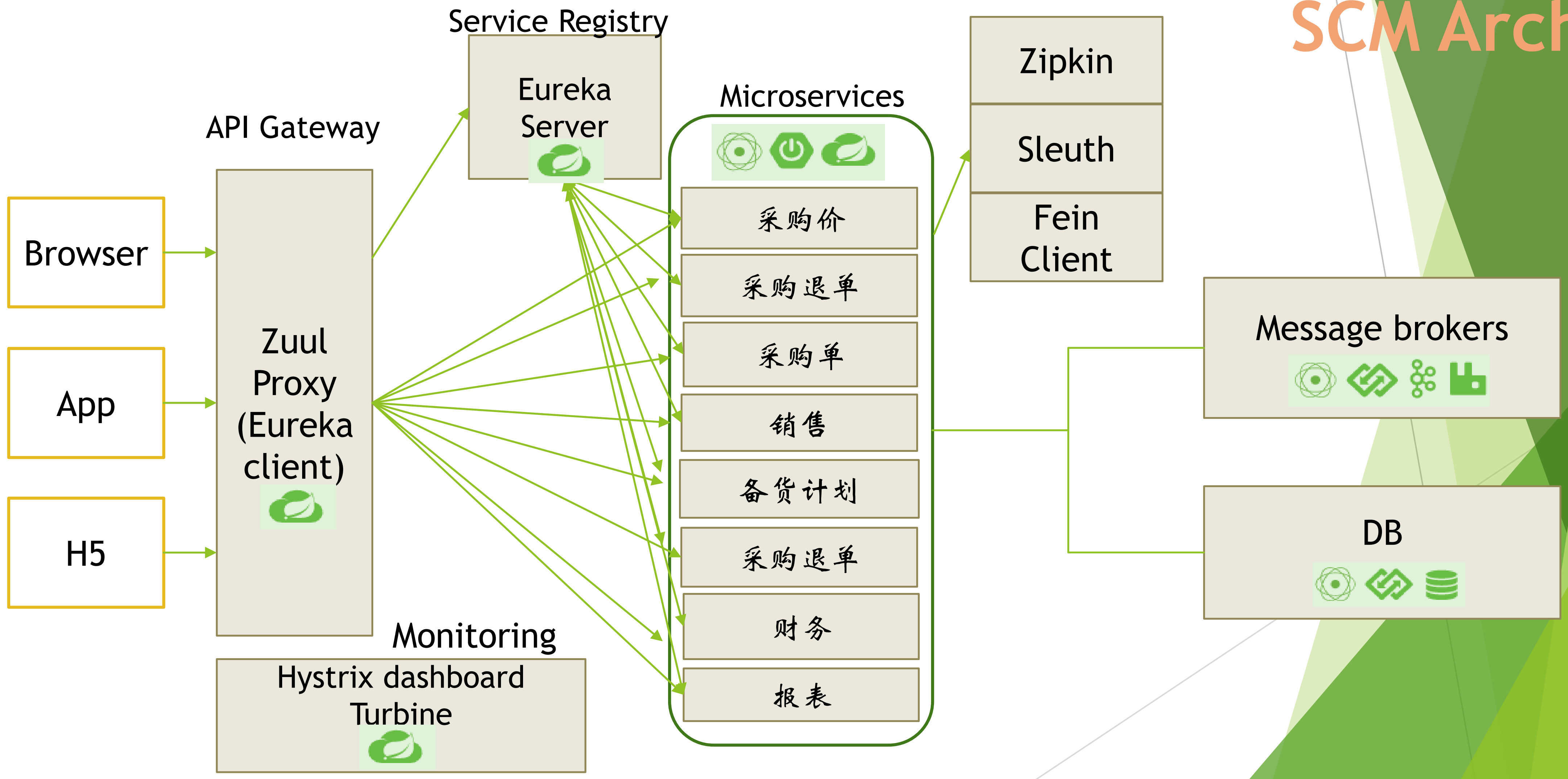
SCM Refactoring Project

Project Scope

- 和WMS DB 彻底解耦（这个是SCM重构项目重点）；
- 货品对接从CMS切换到 GMS（依赖货品）；
- 数据库从SQL Server切换到MySql；
- 从.NET 技术栈切换到Java技术栈；

2018年IT团队：左手支持业务发展，右手实现系统解耦。

SCM Architecture based on Spring Cloud



SCM Arch.

Core Components

- Eureka – Service Discovery
- Hystrix Circuit Breaker
- Monitoring: Hystrix Dashboard
- Ribbon – Client Side Load Balancer
- Zuul – API Gateway
- Distributed tracing with Sleuth and Zipkin

NETFLIX

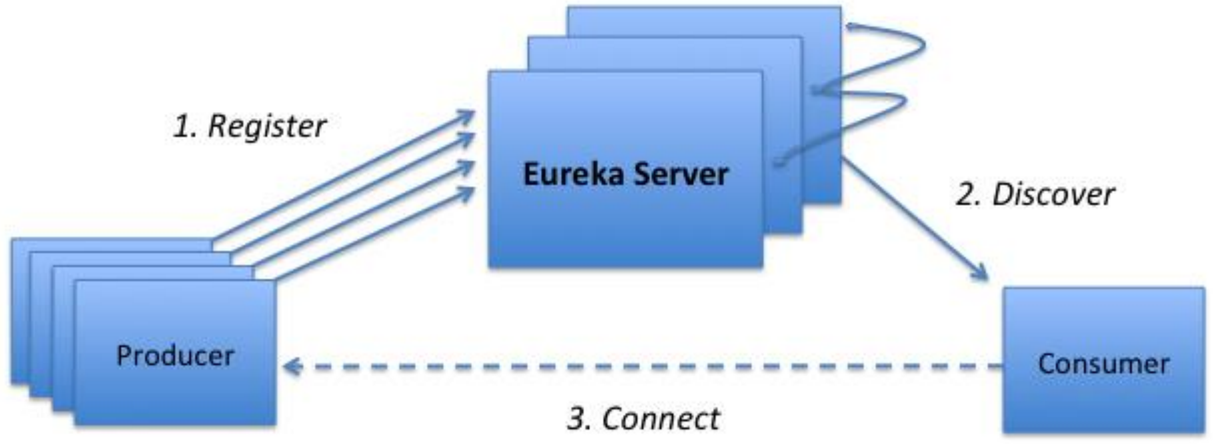
https://cloud.spring.io/spring-cloud-netflix/

4

Spring Cloud

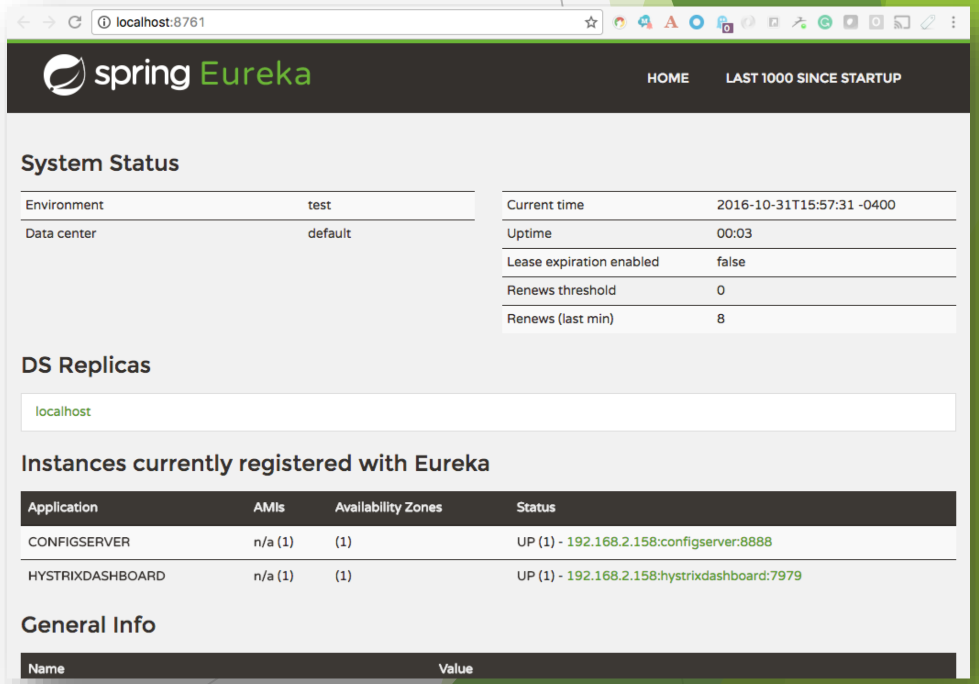
Eureka – Service Discovery

- Eureka 架构图



- Service Producer
- Service Consumer
- Eureka Server

Eureka Server控制台



System Status

Environment	test
Data center	default

Current time	2016-10-31T15:57:31 -0400
Uptime	00:03
Lease expiration enabled	false
Renews threshold	0
Renews (last min)	8

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIGSERVER	n/a (1)	(1)	UP (1) - 192.168.2.158:configserver:8888
HYSTRIXDASHBOARD	n/a (1)	(1)	UP (1) - 192.168.2.158:hystrixdashboard:7979

General Info

Name	Value

Hystrix – Circuit Breaker

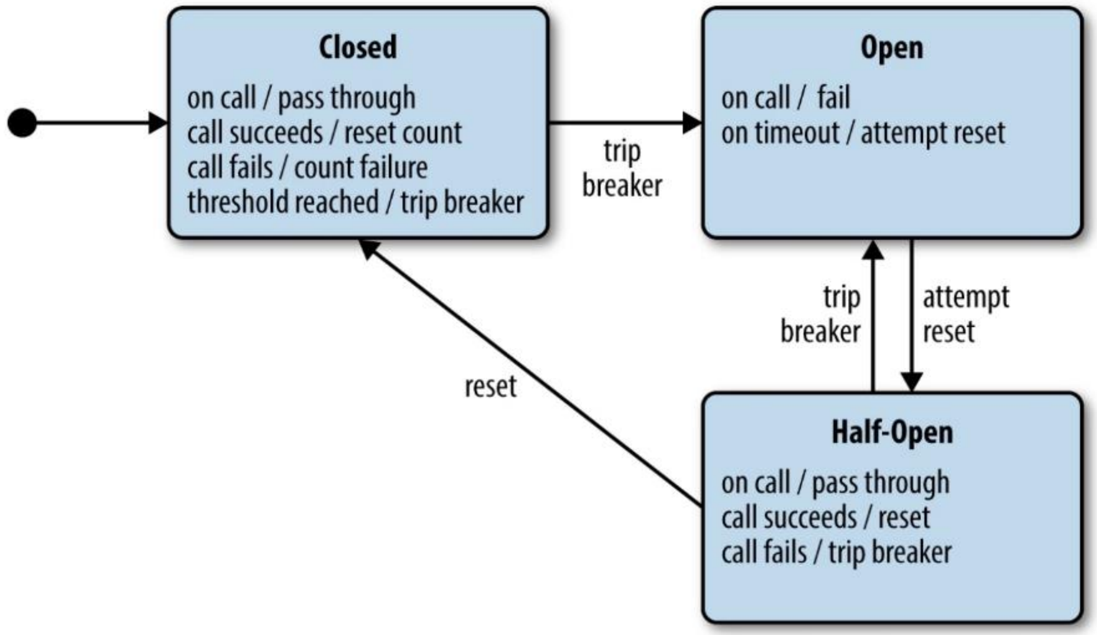
Hystrix的功能

- Hystrix 也是Netflix中的一个组件（框架）；
- 在网络发生延迟或者故障时，对客户端进行保护；
- 通过延迟阈值或容错逻辑，来防止级联故障；
- 提供可回退操作（fallback method）来实现容错；
- 可支持实时监控、报警等等；

- @EnableCircuitBreaker
- @EnableHystrixDashboard

当服务的依赖被确定为不健康时，使用熔断器来断开对该服务与其依赖的远程调用，就像电路熔断器，可以防止电力使用过度，发生火灾或者设备损坏一样。

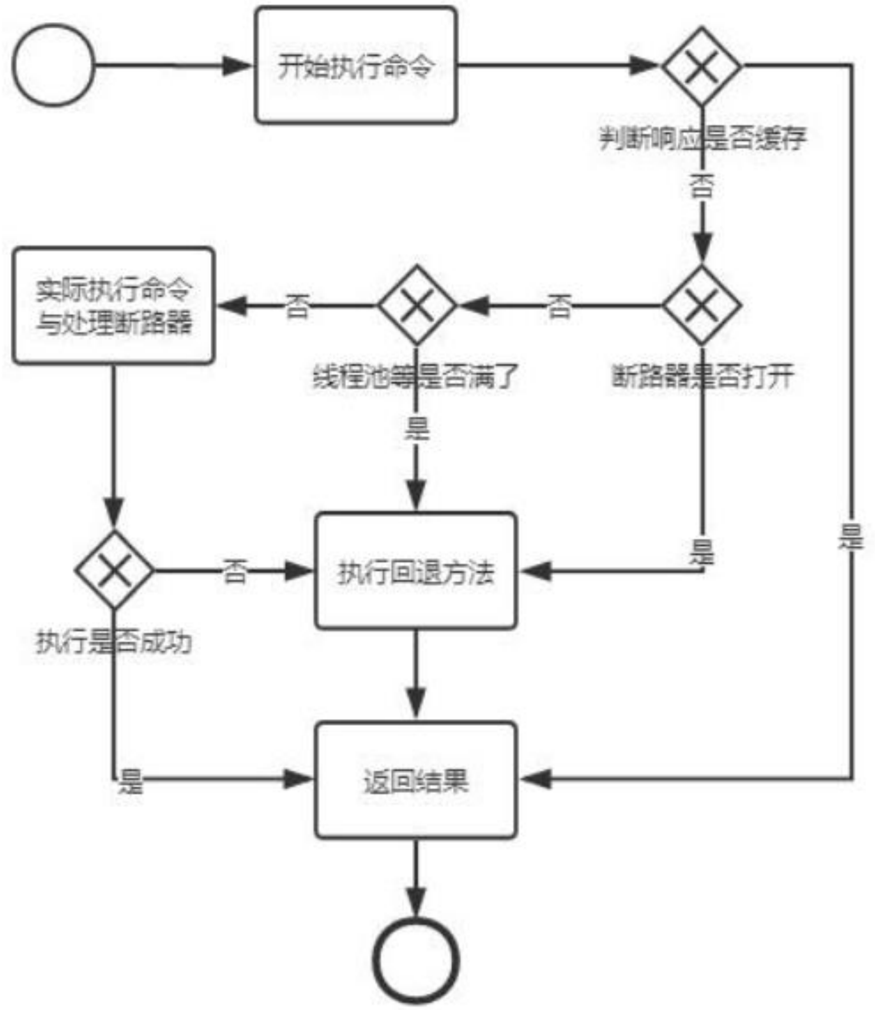
Circuit Breaker State Machine



熔断器状态机

- 当其处于关闭状态时，服务调用将直接传递给依赖关系。如果任何一个调用失败，则计入这次失败。当故障计数在指定时间内达到指定的阈值时，熔断器进入打开状态。
- 在熔断器为打开状态时，所有调用都会失败。
- 在预定时间段之后，线路转变为“半开”状态。在这种状态下，调用再次尝试远程依赖组件。成功的调用将熔断器转换回关闭状态，而失败的调用将熔断器返回到打开状态。

Hystrix execution process



第一步： 开始执行命令。

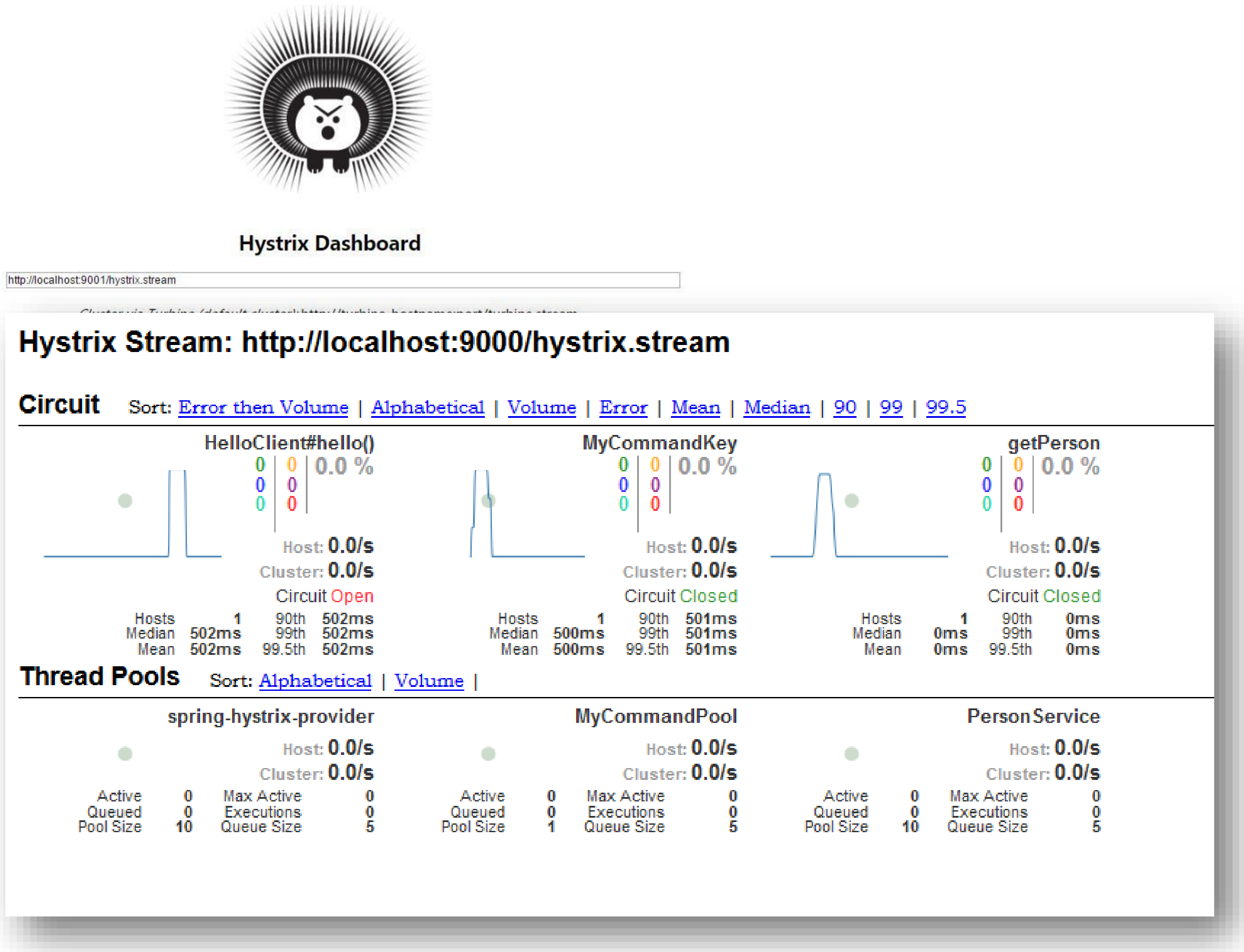
第二步： 判断是否打开了缓存@CacheResult， 打开了缓存就直接查找缓存并返回结果。

第三步： 判断断路器是否打开， 如果打开了， 就表示链路不可用， 直接执行回退方法，响应用户请求。

第四步： 判断线程池、 信号量（计数器） 等条件， 例如像线程池超负荷， 则执行回退方法， 否则， 就去执行命令的请求 。

第五步： 执行命令， 计算是否需要对断路器进行处理，执行完成后， 如果满足一定条件， 则需要开启断路器。如果执行成功， 则返回结果， 反之则执行回退。

Monitoring: Hystrix Dashboard



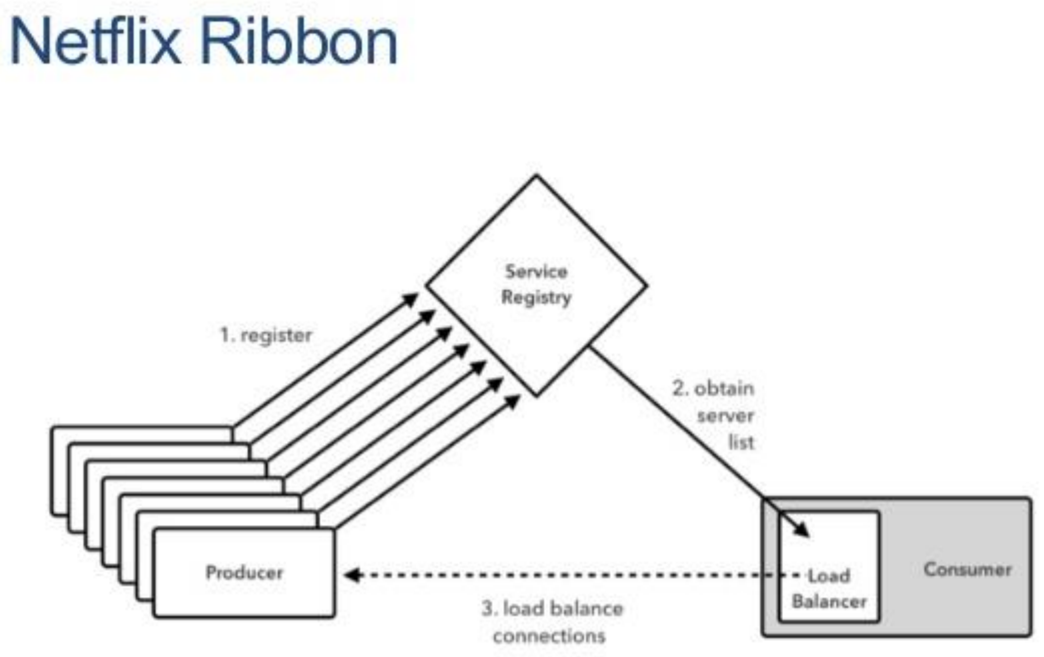
Hystrix Dashboard

http://localhost:9001/hystrix.stream

```
Hystrix Stream: http://localhost:9000/hystrix.stream

Circuit    Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

            HelloClient#hello()                 MyCommandKey                        getPerson
            0 | 0 | 0.0 %                       0 | 0 | 0.0 %                       0 | 0 | 0.0 %
            0 | 0                               0 | 0                               0 | 0
            0 | 0                               0 | 0                               0 | 0
            Host: 0.0/s                         Host: 0.0/s                         Host: 0.0/s
            Cluster: 0.0/s                      Cluster: 0.0/s                      Cluster: 0.0/s
            Circuit Open                        Circuit Closed                      Circuit Closed
   Hosts      1    90th  502ms     Hosts      1    90th  501ms     Hosts     1    90th   0ms
   Median 502ms    99th  502ms     Median 500ms    99th  501ms     Median  0ms    99th   0ms
   Mean   502ms  99.5th  502ms     Mean   500ms  99.5th  501ms     Mean    0ms  99.5th   0ms

Thread Pools    Sort: Alphabetical | Volume |

            spring-hystrix-provider             MyCommandPool                       PersonService
            Host: 0.0/s                         Host: 0.0/s                         Host: 0.0/s
            Cluster: 0.0/s                      Cluster: 0.0/s                      Cluster: 0.0/s
   Active     0  Max Active  0         Active     0  Max Active  0         Active     0  Max Active  0
   Queued     0  Executions  0         Queued     0  Executions  0         Queued     0  Executions  0
   Pool Size 10  Queue Size  5         Pool Size  1  Queue Size  5         Pool Size 10  Queue Size  5
```

- Hystrix-dashboard是一款实时监控的工具，通过Hystrix Dashboard，我们可以在直观地看到各Hystrix Command的请求响应时间，请求成功率等数据
- 如果监控整个集群的情况，可以使用Turbine框架；

Ribbon – Client Side Load Balancer

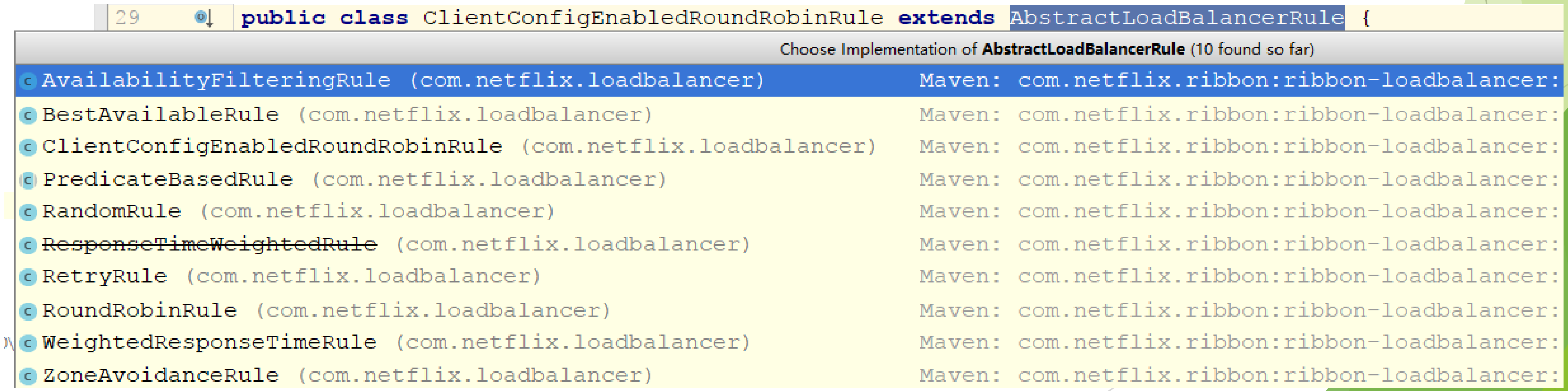


- 负载均衡是对系统的高可用、网络压力的缓解和处理能力扩容的重要手段之一；
- Spring Cloud Ribbon 是一个基于Http和TCP的客户端负载均衡组件，基于Netflix Ribbon实现；
- 不像服务注册中心、API网关那样独立部署，但是它几乎存在于每个微服务的基础设施中；
  - RestTemplate+Ribbon 实现负载均衡；
  - 声明式服务调用Feign内部也使用了ribbon做负载均衡；
  - 默认Zuul网关也使用Ribbon定位服务注册中的实例；

Ribbon Load Balancer’s Rule

Ribbon 提供了几种负载均衡策略：

- RoundRobinRule：轮询策略，Ribbon以轮询的方式选择服务器，这个是默认值；
- RandomRule：随机选择，也就是说Ribbon会随机从服务器列表中选择一个进行访问；
- BestAvailableRule：最大可用策略，即先过滤出故障服务器后，选择一个当前并发请求数最小的；
- WeightedResponseTimeRule：带有加权的轮询策略，对各个服务器响应时间进行加权处理，然后在采用轮询的方式来获取相应的服务器；
- AvailabilityFilteringRule：可用过滤策略，先过滤出故障的或并发请求大于阈值一部分服务实例，然后再以线性轮询的方式从过滤后的实例清单中选出一个；
- ZoneAvoidanceRule：区域感知策略，先使用主过滤条件（区域负载器），选择最优区域；然后根据Server的可用性选择Server；最后对满足条件的服务器则使用RoundRobinRule(轮询方式)选择一个服务器实例。



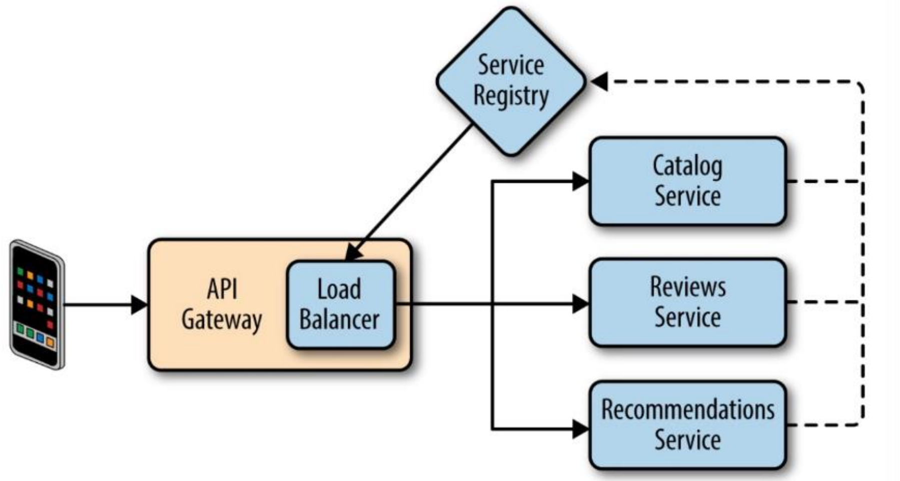
```
public class ClientConfigEnabledRoundRobinRule extends AbstractLoadBalancerRule {
```

Choose Implementation of **AbstractLoadBalancerRule** (10 found so far)

```
AvailabilityFilteringRule (com.netflix.loadbalancer)             Maven: com.netflix.ribbon:ribbon-loadbalancer:
BestAvailableRule (com.netflix.loadbalancer)                     Maven: com.netflix.ribbon:ribbon-loadbalancer:
ClientConfigEnabledRoundRobinRule (com.netflix.loadbalancer)     Maven: com.netflix.ribbon:ribbon-loadbalancer:
PredicateBasedRule (com.netflix.loadbalancer)                    Maven: com.netflix.ribbon:ribbon-loadbalancer:
RandomRule (com.netflix.loadbalancer)                            Maven: com.netflix.ribbon:ribbon-loadbalancer:
ResponseTimeWeightedRule (com.netflix.loadbalancer)              Maven: com.netflix.ribbon:ribbon-loadbalancer:
RetryRule (com.netflix.loadbalancer)                             Maven: com.netflix.ribbon:ribbon-loadbalancer:
RoundRobinRule (com.netflix.loadbalancer)                        Maven: com.netflix.ribbon:ribbon-loadbalancer:
WeightedResponseTimeRule (com.netflix.loadbalancer)              Maven: com.netflix.ribbon:ribbon-loadbalancer:
ZoneAvoidanceRule (com.netflix.loadbalancer)                     Maven: com.netflix.ribbon:ribbon-loadbalancer:
```

Zuul – API Gateway

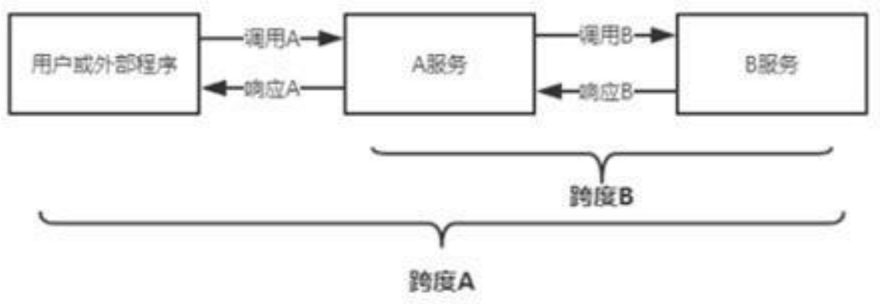
- 动态路由：将请求动态路由到不同的服务集群；
  - Zuul也有负载均衡的功能，它是针对外部请求做负载均衡；
- 负载分配：设置请求的处理能力，删除超出限制的请求；
- 身份验证和安全性：对需要身份验证的资源，拒绝非法请求；
- 聚合微服务，简化客户端的开发，同时提升性能；



Distributed tracing with Sleuth and Zipkin

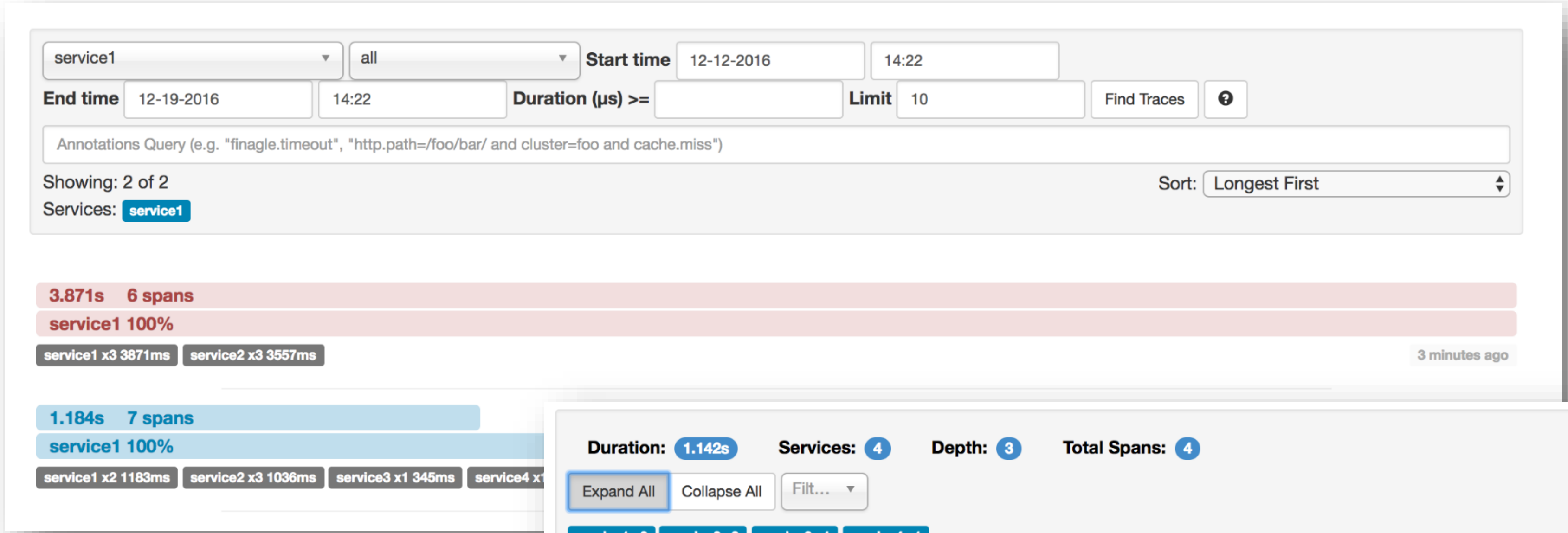
集群中的服务、节点数量增多，且存在复杂的依赖关系，如何有效管理；

- Sleuth – 跟踪微服务的调用过程；
  - 借鉴Google Dapper的设计，了解Trace/Span概念；
- Trace：表示整个跟踪的过程，从用户发起请求，到最终的响应。一次跟踪包含多个跨度，这些跨度以树状结构进行保存。
- Span：跨度 -- 假设用户向A服务发送请求，A服务又要调用B服务，那么此时将会产生两个跨度：用户调用A服务、A服务调用B服务。



Zipkin UI

- Zipkin - 分布式跟踪系统，用来收集、管理微服务产生的数据；



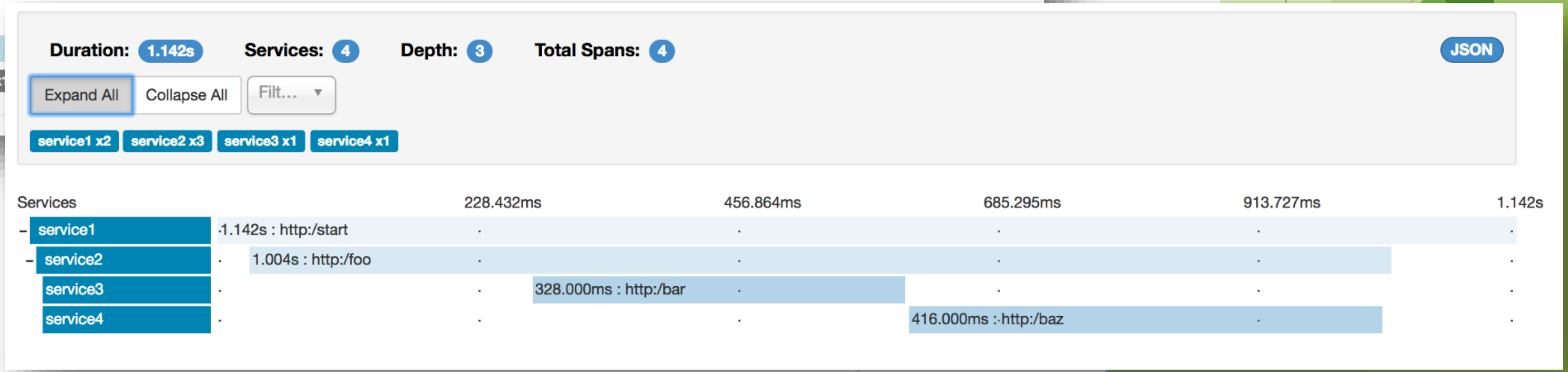
service1 | all | Start time 12-12-2016 14:22
End time 12-19-2016 14:22 | Duration (μs) >= | Limit 10 | Find Traces

Annotations Query (e.g. "finagle.timeout", "http.path=/foo/bar/ and cluster=foo and cache.miss")

Showing: 2 of 2 Sort: Longest First
Services: service1

3.871s 6 spans
service1 100%
service1 x3 3871ms | service2 x3 3557ms | 3 minutes ago

1.184s 7 spans
service1 100%
service1 x2 1183ms | service2 x3 1036ms | service3 x1 345ms | service4 x1



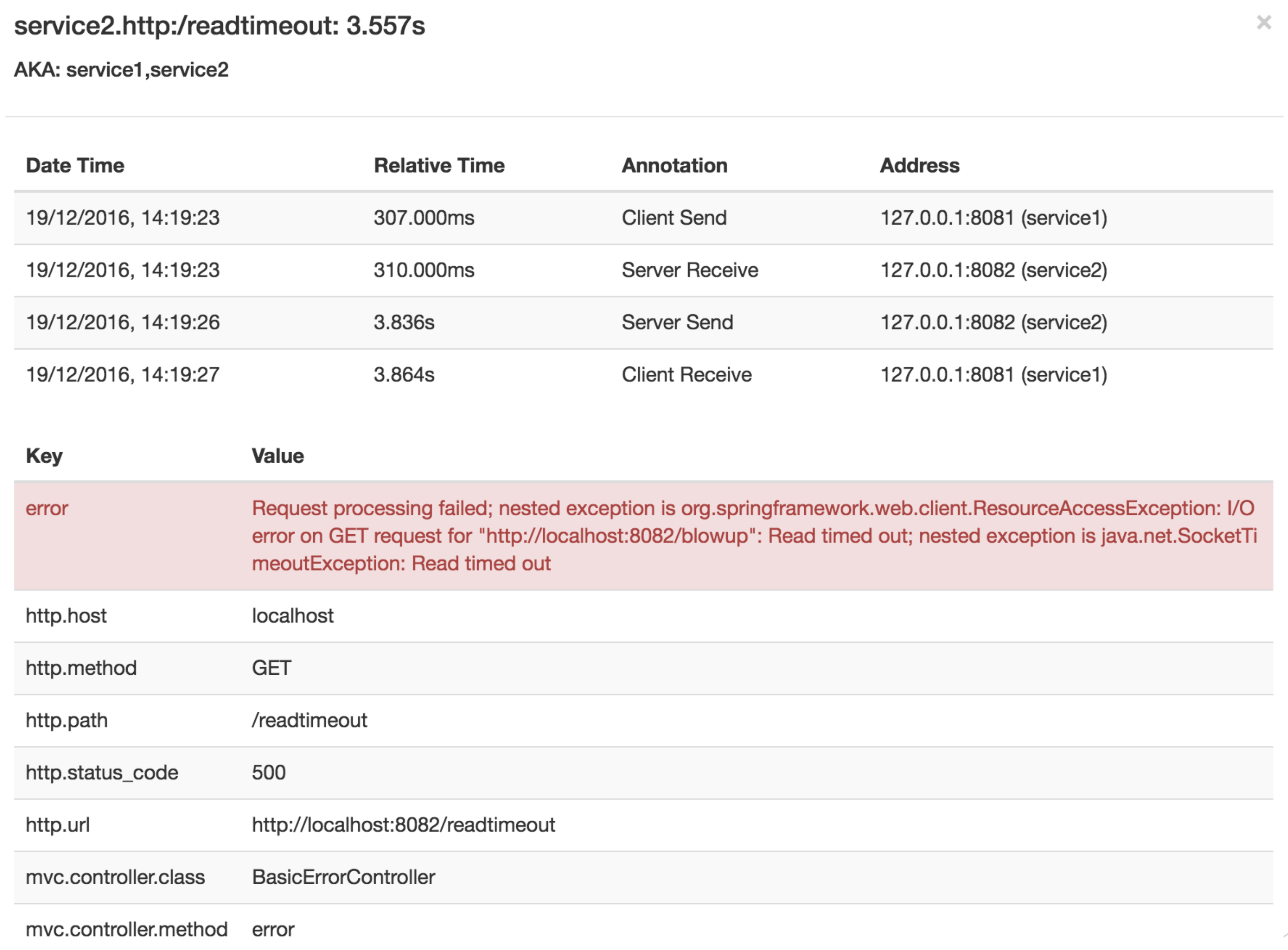
Duration: 1.142s Services: 4 Depth: 3 Total Spans: 4 JSON

Expand All | Collapse All | Filt...

service1 x2 | service2 x3 | service3 x1 | service4 x1

```
Services      228.432ms     456.864ms     685.295ms     913.727ms     1.142s
- service1    1.142s : http:/start
  - service2    1.004s : http:/foo
    service3                328.000ms : http:/bar
    service4                              416.000ms : http:/baz
```

Span-跨度详细信息



service2.http:/readtimeout: 3.557s

AKA: service1,service2

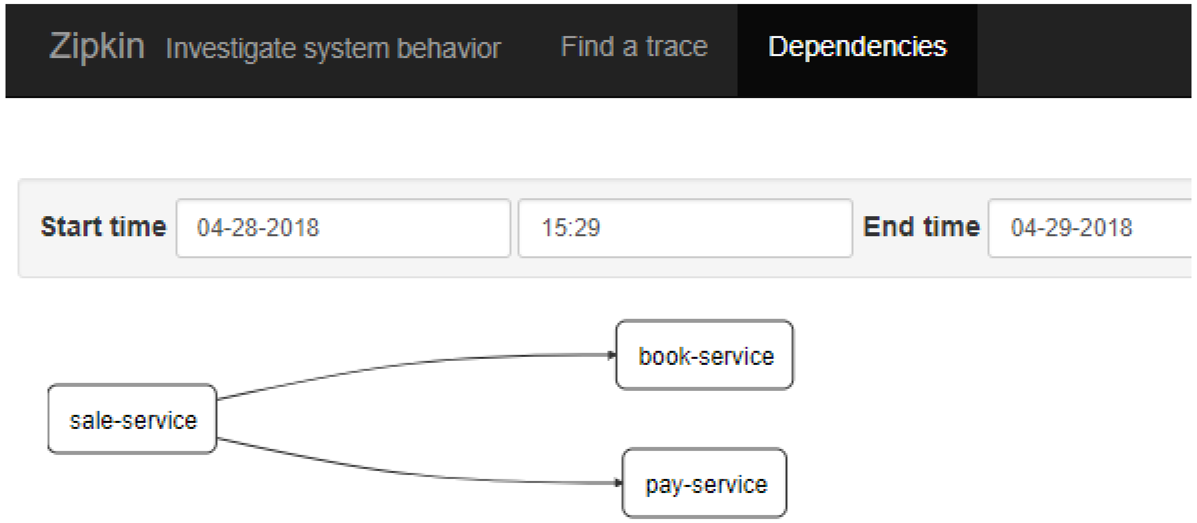
Date Time	Relative Time	Annotation	Address
19/12/2016, 14:19:23	307.000ms	Client Send	127.0.0.1:8081 (service1)
19/12/2016, 14:19:23	310.000ms	Server Receive	127.0.0.1:8082 (service2)
19/12/2016, 14:19:26	3.836s	Server Send	127.0.0.1:8082 (service2)
19/12/2016, 14:19:27	3.864s	Client Receive	127.0.0.1:8081 (service1)

Key	Value
error	Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://localhost:8082/blowup": Read timed out; nested exception is java.net.SocketTimeoutException: Read timed out
http.host	localhost
http.method	GET
http.path	/readtimeout
http.status_code	500
http.url	http://localhost:8082/readtimeout
mvc.controller.class	BasicErrorController
mvc.controller.method	error

- CS - Client Sent 客户端发送请求；
- SR - Server Received 服务端收到请求；
- SS - Server Send 服务端完成请求的处理，并对客户端做出响应；
- CR - Client Received 客户端收到响应；
- 整个Span跨度结束。

Dependency graph in Zipkin

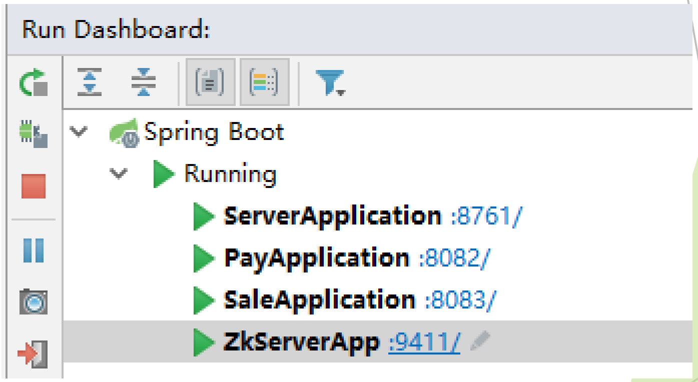
- 微服务的依赖关系



构建Zipkin服务器

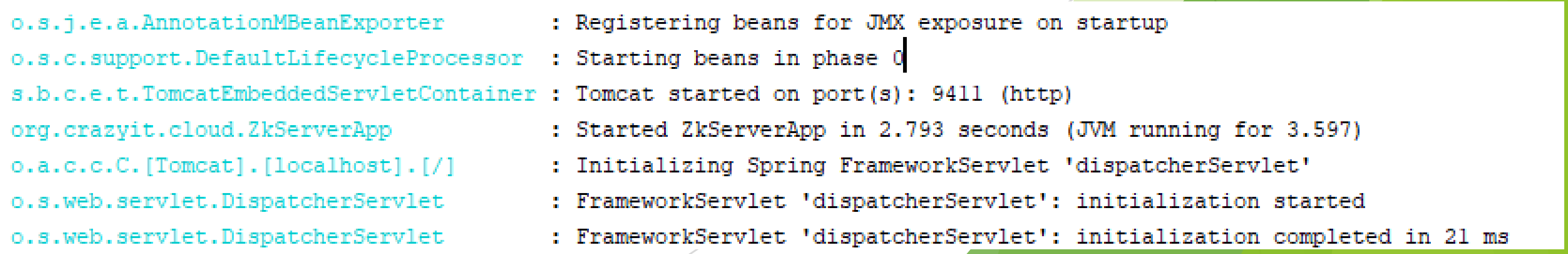
- 基于Spring Boot构建Zipkin服务器；
- 启动类配置@EnableZipkinServer；
- 默认端口：9411；
- Dependency依赖；

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
    <scope>runtime</scope>
</dependency>
```



Run Dashboard:

```
Spring Boot
  Running
    ServerApplication :8761/
    PayApplication :8082/
    SaleApplication :8083/
    ZkServerApp :9411/
```



```
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
o.s.c.support.DefaultLifecycleProcessor  : Starting beans in phase 0
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9411 (http)
org.crazyit.cloud.ZkServerApp            : Started ZkServerApp in 2.793 seconds (JVM running for 3.597)
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 21 ms
```

配置微服务连接Zipkin Server

- 微服务添加依赖-Dependency

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

- 微服务配置Zipkin 服务器

```
spring:
  application:
    name: sale-service
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      percentage: 1.0
```

Spring Cloud & .Net Core

- Steeltoe – Pivotal 开源项目；
- 支持.Net Core和.NET Framework4.0 使用Spring Cloud快速构建微服务框架；
- Spring Cloud搭建底层微服务框架，.Net Core来编写业务逻辑；

steeltoe
by Pivotal

http://steeltoe.io/docs/

.Net
Core

Spring Cloud+.Net Core搭建微服务架构

http://www.cnblogs.com/longxianghui/p/7561259.html

- spring cloud+dotnet core搭建微服务架构：服务注册（一）
- spring cloud+dotnet core搭建微服务架构：服务发现（二）
- spring cloud+dotnet core搭建微服务架构： Api网关（三）
- spring cloud+dotnet core搭建微服务架构：配置中心（四）

References

- http://spring.io/ 官网
- http://start.spring.io/
  - SPRING INITIALIZR快速创建Spring Boot项目
- https://pivotal.io/cloud-native
- http://dockone.io/article/2991
  - 云原生概念
- https://12factor.net/ 12要素（或原则）
- https://github.com/rootsongjc/migrating-to-cloud-native-application-architectures
  - 迁移到云原生应用架构
- http://cloud.spring.io/spring-cloud-static/Edgware.SR3/single/spring-cloud.html Spring Cloud 文档

THANKS