

# Spring Cloud 学习笔记

## 学习资料

Spring Cloud 中文翻译网站:

1. <https://www.bookstack.cn/books/spring-cloud-docs>
2. <http://docs.springcloud.cn/>
3. <http://docs.springcloud.cn/user-guide/eureka/>

## 什么是集群

**计算机集群**简称集群，是一种计算机系统，它通过一组松散集成的计算机软件/硬件连接起来高度紧密地协作完成计算工作。在某种意义上，他们可以被看作是一台计算机。集群系统中的**单个计算机通常称为节点**，通常通过局域网连接，但也有其它的可能连接方式。集群计算机通常用来改进单个计算机的计算速度和/或可靠性。一般情况下集群计算机比单个计算机，比如工作站或超级计算机性能价格比要高得多

集群技术特点:

1. 通过多台计算机完成同一个工作，达到更高的效率。
2. 两机或多机内容、工作过程等完全一样。如果一台死机，另一台可以起作用

## 什么是分布式

分布式系统是一组计算机，通过网络相互连接传递消息与通信后并协调它们的行为而形成的系统。组件之间彼此进行交互以实现一个共同的目标。

一个业务分拆多个子业务，部署在不同的服务器上

集群和分布式并不冲突，可以有分布式集群

## CAP 理论

### C: 数据一致性(consistency)

所有节点拥有数据的最新版本

### A: 可用性(availability)

数据具备高可用性

### P: 分区容错性(partition-tolerance)

容忍网络出现分区，分区之间网络不可达。

只要是分布式系统，那很有可能会出现一种情况：因为一些故障，使得有些节点之间不连通了，整个网络就分成了几块区域。

数据就散布在了这些不连通的区域中，这就叫分区

1. 如果允许当前用户注册一个账户，此时注册的记录数据只会在节点一和节点二或者节点二和节点三同步，因为节点一和节点三的记录不能同步的。这种情况其实就是选择了可用性(availability)，抛弃了数据一致性(consistency)
2. 如果不允许当前用户注册一个账户(就是要等到节点一和节点三恢复通信)。节点一和节点三一旦恢复通信，我们就可以保证节点拥有的数据是最新版本。这种情况其实就是抛弃了可用性(availability)，选择了数据一致性(consistency)

一般我们说的分布式系统，**P: 分区容错性(partition-tolerance)**这个是必需的，这是客观存在的。CAP 是无法完全兼顾的，从上面的例子也可以看出，我们可以选 AP，也可以选 CP。但是，要注意的是：不是说选了 AP，C 就完全抛弃了。不是说选了 CP，A 就完全抛弃了！

在 CAP 理论中，C 所表示的一致性**是强一致性**(每个节点的数据都是最新版本)，其实一致性还有其他级别的：

1. **弱一致性**：弱一致性是相对于强一致性而言，它不保证总能得到最新的值；
2. **最终一致性(eventual consistency)**：放宽对时间的要求，在被调完成操作响应后的某个时间点，被调多个节点的数据最终达成一致

所以，CAP 理论定义的其实是在容忍网络分区的条件下，“强一致性”和“极致可用性”无法同时达到。

## 为什么需要 SpringCloud?

从分布式/微服务的角度而言：就是把我们的项目，分解成多个小的模块。这些小的模块组合起来，完成功能。

拆分出多个模块以后，就会出现各种各样的问题，而 SpringCloud 提供了一整套的解决方案！

SpringCloud 的基础功能：

- **服务治理： Spring Cloud Eureka**
- **客户端负载均衡： Spring Cloud Ribbon**
- **服务容错保护： Spring Cloud Hystrix**
- **声明式服务调用： Spring Cloud Feign**
- **API 网关服务： Spring Cloud Zuul**
- **分布式配置中心： Spring Cloud Config**



## ZHE RFC

这个 RFC(RFC 是一种文件，记录了各种互联网协议)

SpringCloud 的高级功能：

1. 消息总线： **Spring Cloud Bus**
2. 消息驱动的微服务： **Spring Cloud Stream**
3. 分布式服务跟踪： **Spring Cloud Sleuth**

1. Eureka：实际上在整个过程中维护者每个服务的生命周期。每一个服务都要被注册到 Eureka 服务器上，这里被注册到 Eureka 的服务又称为 Client。Eureka 通过心跳来确定服务是否正常。Eureka 只做请求转发。同时 Eureka 是支持集群的呦!!!
2. Zuul：类似于网关，反向代理。为外部请求提供统一入口。
3. Ribbon/Feign：可以理解为调用服务的客户端。
4. Hystrix：断路器，服务调用通常是深层的，一个底层服务通常为多个上层服务提供服务，那么如果底层服务失败则会造成大面积失败，Hystrix 就是就调用失败后触发定义好的处理方法，从而更友好的解决出错。也是微服务的容错机制。

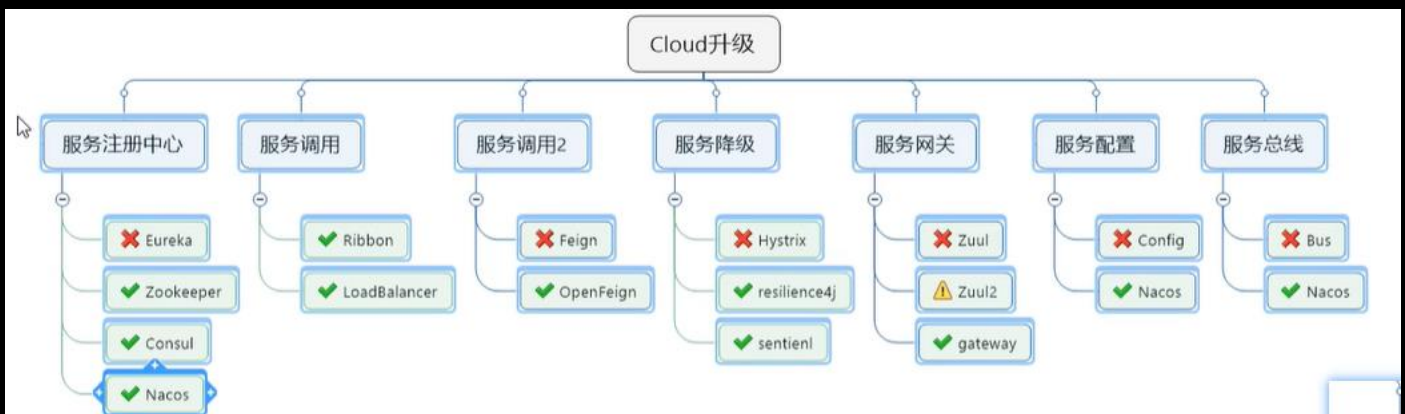
分布式服务组件大全：

1. Spring Cloud Config：配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git 以及 Subversion。
2. Spring Cloud Bus：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。
3. Eureka：云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。
4. Hystrix：熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。
5. Zuul：Zuul 是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。
6. Archaius：配置管理 API，包含一系列配置管理 API，提供动态类型化属性、线程安全配置操

作、轮询框架、回调机制等功能。

7. Consul: 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。
8. Spring Cloud for Cloud Foundry: 通过 Oauth2 协议绑定服务到 CloudFoundry, CloudFoundry 是 VMware 推出的开源 PaaS 云平台。
9. Spring Cloud Sleuth: 日志收集工具包, 封装了 Dapper 和 log-based 追踪以及 Zipkin 和 HTrace 操作, 为 Spring Cloud 应用实现了一种分布式追踪解决方案。
10. Spring Cloud Data Flow: 大数据操作工具, 作为 Spring XD 的替代产品, 它是一个混合计算模型, 结合了流数据与批量数据的处理方式。
11. Spring Cloud Security: 基于 Spring Security 的安全工具包, 为你的应用程序添加安全控制。
12. Spring Cloud Zookeeper: 操作 Zookeeper 的工具包, 用于使用 Zookeeper 方式的服务发现和配置管理。
13. Spring Cloud Stream: 数据流操作开发包, 封装了与 Redis、Rabbit、Kafka 等发送接收消息。
14. Spring Cloud CLI: 基于 Spring Boot CLI, 可以让你以命令行方式快速建立云组件。
15. Ribbon: 提供云端负载均衡, 有多种负载均衡策略可供选择, 可配合服务发现和断路器使用。
16. Turbine: Turbine 是聚合服务器发送事件流数据的一个工具, 用来监控集群下 Hystrix 的 Metrics 情况。
17. Feign: Feign 是一种声明式、模板化的 HTTP 客户端。
18. Spring Cloud Task: 提供云端计划任务管理、任务调度。
19. Spring Cloud Connectors: 便于云端应用程序在各种 PaaS 平台连接到后端, 如: 数据库和消息代理服务。
20. Spring Cloud Cluster: 提供 Leadership 选举, 如: Zookeeper, Redis, Hazelcast, Consul 等常见状态模式的抽象和实现。
21. Spring Cloud Starters: Spring Boot 式的启动项目, 为 Spring Cloud 提供开箱即用的依赖管理。

## 分布式组件推荐



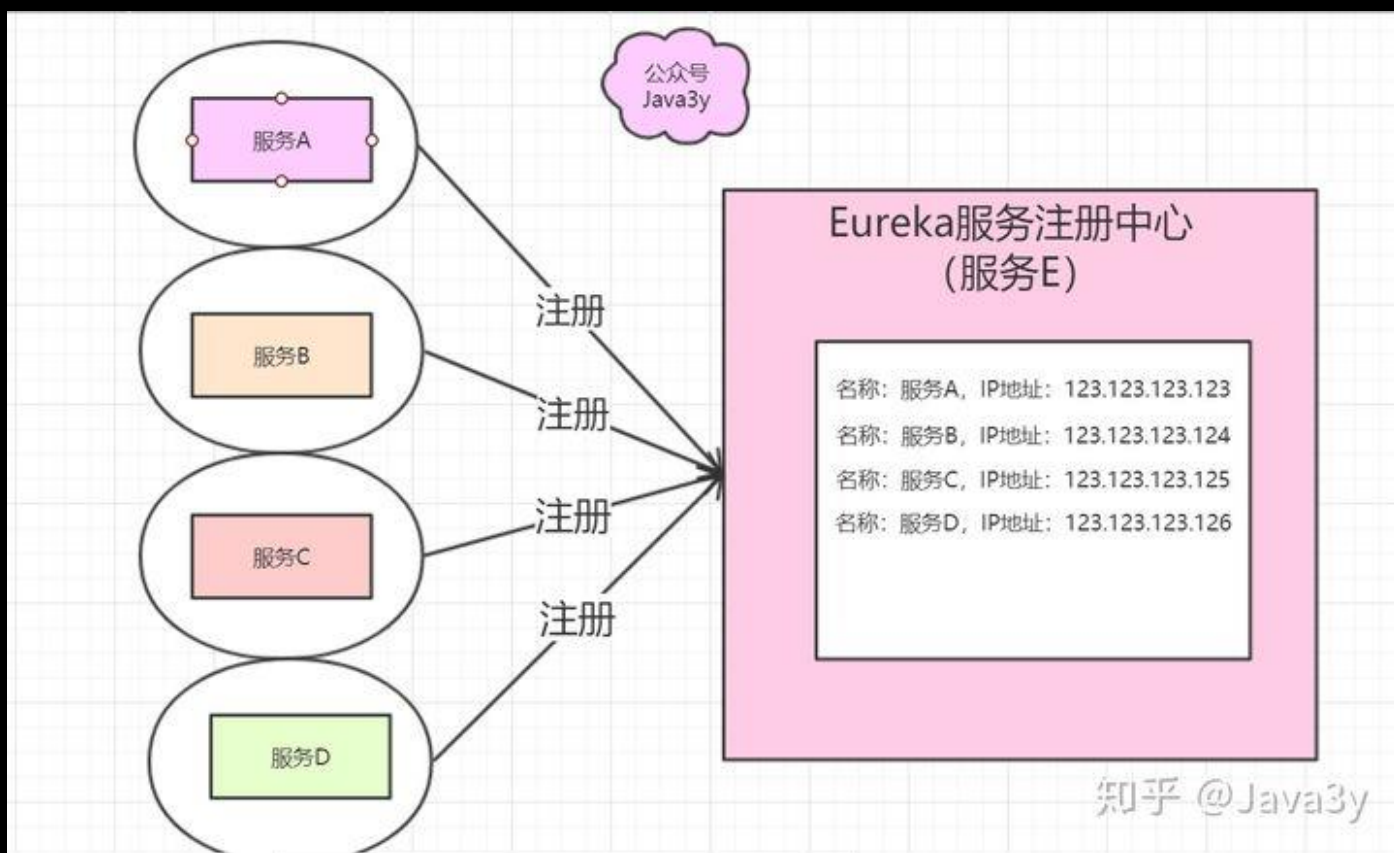
Eureka

为什么需要 eureka?

在分布式中，各个节点之间需要通信，通信需要 ip 地址或者域名，这是一种强耦合，一旦有一个服务的地址变更，就会导致众多服务需要做出变更，为了解耦，引出了 eureka。

Eureka 的思路是这样的：

创建一个 E 服务，将 A、B、C、D 四个服务的信息都注册到 E 服务上，E 服务维护这些已经注册进来的信息



A、B、C、D 四个服务都可以拿到 Eureka(服务 E)那份注册清单。A、B、C、D 四个服务互相调用不再通过具体的 IP 地址，而是**通过服务名来调用**！

- 拿到注册清单--->注册清单上有服务名--->自然就能够拿到服务具体的位置了(IP)。
- 其实简单来说就是：代码中通过服务名找到对应的 IP 地址(IP 地址会变，但服务名一般不会变)

服务注册、服务发现

注册中心，服务的注册与发现

服务端：

1、引入服务端启动器：eureka-server

2、添加配置

spring.application.name 服务名

eureka.client.service-url.defaultZone http://localhost:10086/eureka

剔除无效连接的间隔时间 eureka.server.eviction-interval-timer-in-ms

关闭自我保护 eureka.server.enable-self-preservation

3、@EnableEurekaServer 开启 eureka 服务端功能



客户端:

1、引入启动器: `eureka-client`

2、添加配置

`spring.application.name`

`eureka.client.service-url.defaultZone`

心跳时间 `eureka.instance.lease-renewal-interval-in-seconds`

过期时间 `eureka.instance.lease-expiration-duration-in-seconds`

是否注册给 eureka 容器 `eureka.client.register-with-eureka`

是否拉取服务列表 `eureka.client.fetch-registry`

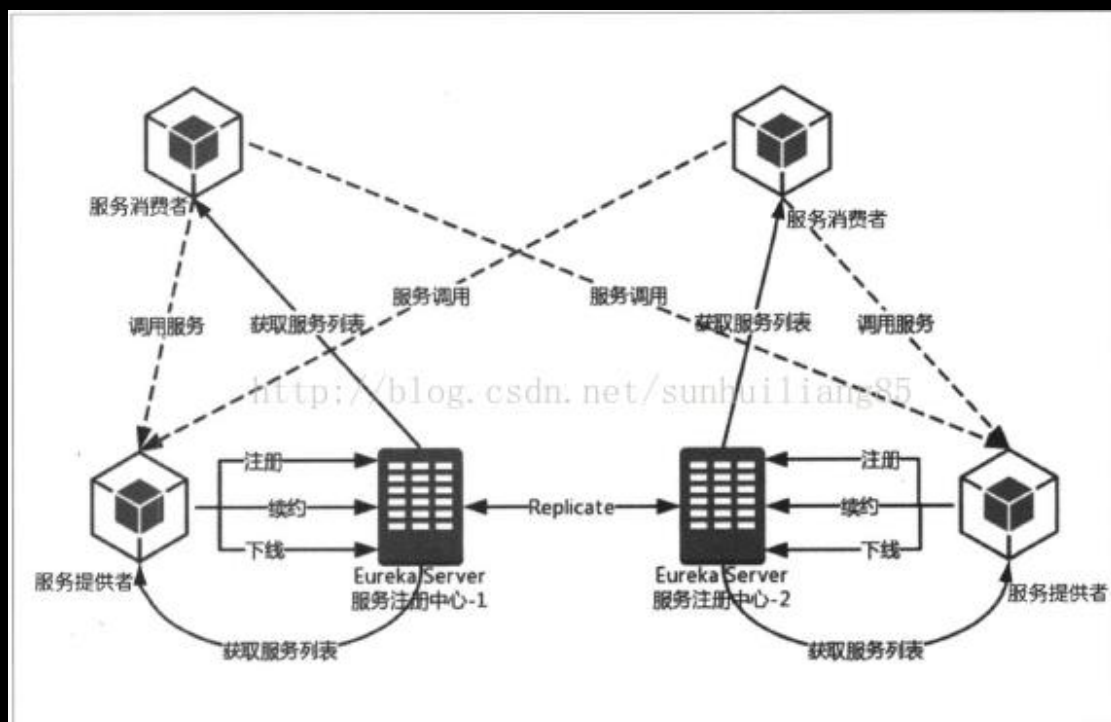
拉取服务的间隔时间 `eureka.client.registry-fetch-interval-seconds`

3、启用 eureka 客户端: `@EnableDiscoveryClient`

## Eureka 服务治理体系

**服务治理**是微服务架构中最为核心和基础的模块，它主要用来实现各个微服务实例的**自动化注册和发现**。

为了避免 eureka 出现单点故障，使用多个 eureka 服务器——**eureka 集群**。



Eureka 工作流程:

1. Eureka Server 启动成功，等待服务端注册。在启动过程中如果配置了集群，集群之间定时通过 Replicate 同步注册表，每个 Eureka Server 都存在独立完整的服务注册表信息
2. Eureka Client 启动时根据配置的 Eureka Server 地址去注册中心注册服务
3. Eureka Client 会每 30s 向 Eureka Server 发送一次心跳请求，证明客户端服务正常

(心跳机制)

4. 当 Eureka Server 90s 内没有收到 Eureka Client 的心跳，注册中心则认为该节点失效，会注销该实例
5. 单位时间内 Eureka Server 统计到有大量的 Eureka Client 没有上送心跳，则认为可能为网络异常，进入自我保护机制，不再剔除没有上送心跳的客户端
6. 当 Eureka Client 心跳请求恢复正常之后，Eureka Server 自动退出自我保护模式
7. Eureka Client 定时全量或者增量从注册中心获取服务注册表，并且将获取到的信息缓存到本地
8. 服务调用时，Eureka Client 会先从本地缓存找寻调取的服务。如果获取不到，先从注册中心刷新注册表，再同步到本地缓存
9. Eureka Client 获取到目标服务器信息，发起服务调用
10. Eureka Client 程序关闭时向 Eureka Server 发送取消请求，Eureka Server 将实例从注册表中删除

### ● 服务注册

在服务治理框架中，通常都会构建一个注册中心，每个服务单元向注册中心登记自己提供的服务，包括服务的主机与端口号、服务版本号、通讯协议等一些附加信息。注册中心按照服务名分类组织服务清单，同时还需要以心跳检测的方式去监测清单中的服务是否可用，若不可用需要从服务清单中剔除，以达到排除故障服务的效果。

### ● 服务发现

在服务治理框架下，服务间的调用不再通过指定具体的实例地址来实现，而是通过服务名发起请求调用实现。服务调用方通过服务名从服务注册中心的服务清单中获取服务实例的列表清单，通过指定的负载均衡策略取出一个服务实例位置来进行服务调用。

## Ribbon

负载均衡组件

配置负载均衡策略：

```
<service-provider (服务名)>.ribbon.NFLoadBalancerRuleClassName:  
com.netflix.loadbalancer.RandomRule (负载均衡策略全路径)
```

@LoadBalanced 在 RestTemplate 的 @Bean 方法上

## hystrix

降级：

1、引入 hystrix 启动器

2、添加配置，超时时间的配置：

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 6000
```

3、启用熔断组件 @EnableCircuitBreaker

组合注解：@SpringCloudApplication，包含了 @SpringBootApplication 和 @EnableDiscoveryClient

局部熔断：返回值和参数列表和被熔断的方法一致

在被熔断的方法上 @HystrixCommand(fallbackMethod="熔断方法名")

全局熔断：返回值和被熔断的方法返回值一致，不能有参数列表

在类上 @DefaultProperties(defaultFallback="全局熔断方法名")

在具体的被熔断方法上 @HystrixCommand

熔断：

1、close：关闭状态

所有请求都正常访问

2、open：打开状态

所有请求都无法访问

触发：连续失败的比例大于 50%或者失败次数不少于 20

维持 5s 的休眠时间

3、half open：半开状态

释放部分请求通过，如果正常就进入 close 状态，如果不正常就进入 open 状态

触发：休眠时间之后



## 客户端弹性模式

- 远程服务发生错误或表现不佳导致的问题：客户端长时间等待调用返回
- 客户端弹性模式要解决的重点：让客户端免于崩溃。
- 目标：让客户端快速失败，而不消耗数据库连接或线程池之类的宝贵资源，防止远程服务的问题向客户端上游传播。

ervice C 阻塞，那么 B，A 都会阻塞

### 4 种客户端弹性模式：

- 客户端负载均衡 (client load banlance) 模式

Ribbon 提供的负载均衡器，帮助发现问题，并删除实例(使用心跳机制)

Ribbon 是 Netflix 发布的负载均衡器，它有助于控制 HTTP 和 TCP 客户端的行为。为 Ribbon 配置服务提供者地址列表后，Ribbon 就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon 默认为我们提供了很多的负载均衡算法，例如轮询、随机等。当然，我们也可为 Ribbon 实现自定义的负载均衡算法。

- 断路器模式(Circuit Breaker Patten)

监视调用失败的次数，快速失败

如果一个服务调用另一个服务多次皆失败，下次调用直接判为失败，过一段时间后则不会判失败而是尝试

- 后备 (fallback) 模式

远程服务调用失败，执行替代代码路径

如：更换数据库、返回默认值

- 舱壁隔离模式(Bulkhead Isolation Pattern)

线程池充当服务的舱壁

把某些请求数限制在某一个范围，以留出充足的空间供其他服务使用

## Hystrix

- Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.
- google翻译：Hystrix是一个延迟和容错库 旨在隔离对远程系统，服务和第三方库的访问点，停止级联故障，并在不可避免发生故障的复杂分布式系统中实现弹性。

spring-cloud-starter-hystrix

hystrix-javanica

Step2 启动类加注解: @EnableCircuitBreaker

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableCircuitBreaker
public class Application {
```

Step3 用断路器包装远程资源调用, 方法加注解: @HystrixCommand

```
@HystrixCommand
public Cargo getCargo(Long customerId, Long cargoId) {
    System.out.println("=====CargoService.getCargo Correlation id:");
    // 可能超时, 抛出com.netflix.hystrix.exception.HystrixRuntimeException
    randomlyRunLong();

    Cargo cargo = cargoRepository.findOne(cargoId);
    cargo.setProduct(config.getExampleProduct());

    Customer customer = getCustomer(customerId);

    cargo.setName(customer.getName());
    cargo.setAddress(customer.getAddress());
    cargo.setCity(customer.getCity());
    cargo.setEmail(customer.getEmail());

    return cargo;
}
```

超时抛出异常  
默认1秒

Step4 默认 1 秒 超时, 超时 会 抛 异 常 :  
com.netflix.hystrix.exception.HystrixRuntimeException

自定义超时时间:

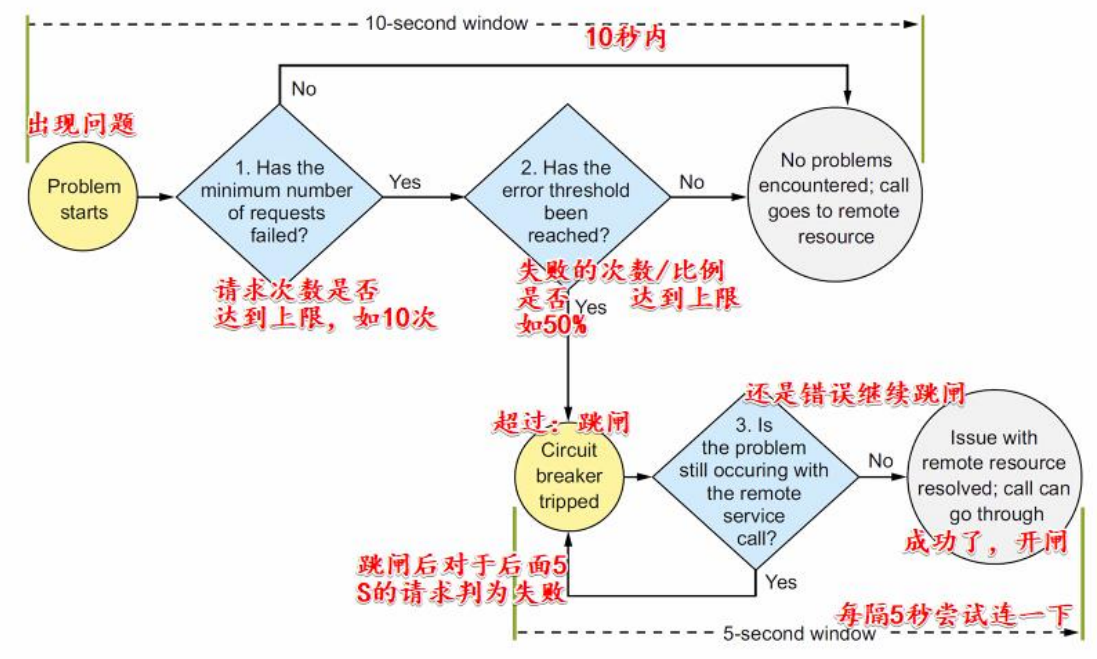
```
@HystrixCommand(
    commandProperties = {@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "12000")}
)
```

超时后定义为失败

备用方法:

```
@HystrixCommand(fallbackMethod = "buildFallbackGetCargo")
```

## 断路器模式(Circuit Breaker Patten)



## @HystrixCommand注解配置

属性名称	默认值
fallbackMethod	None
threadPoolKey	None
threadPoolProperties	None
coreSize	10
maxQueueSize	-1
circuitBreaker.requestVolumeThreshold	20
circuitBreaker.errorThresholdPercentage	50
circuitBreaker.sleepWindowInMilliseconds	5000
metricsRollingStats.timeInMilliseconds	10000
metricsRollingStats.numBuckets	10

## feign

为了简化我们的开发, Spring Cloud Feign 出现了! 它基于 Netflix Feign 实现, 整合了 Spring

Cloud Ribbon 与 Spring Cloud Hystrix, 除了整合这两者的强大功能之外, 它还提 供了**声明式的服务调用**(不再通过 RestTemplate)。

Feign 是一种**声明式**、模板化的 HTTP 客户端。在 Spring Cloud 中使用 Feign, 我们可以做到**使用 HTTP 请求远程服务时能与调用本地方法一样的编码体验**, 开发者完全感知不到这是远程方法, 更感知不到这是个 HTTP 请求。

1、引入 feign 的启动器

2、开启熔断: `feign.hystrix.enable=true`

3、开启 feign 的功能: `@EnableFeignClients`

4、实现: 定义一个接口, 使用注解 `@FeignClient(value="服务名", fallback=实现类.class)`, 方法上的注解使用的都是 springMVC 的注解

## zuul

网关组件, 有路由和过滤器功能

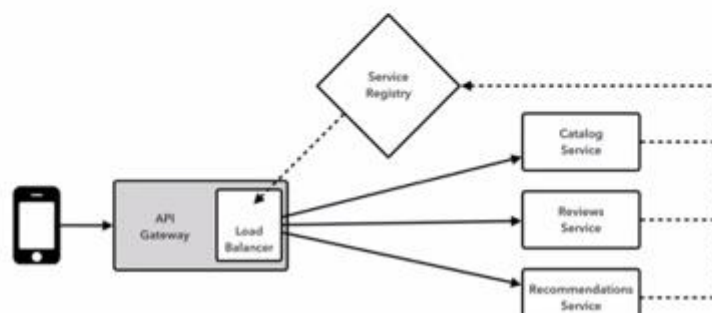
- 服务网关位于服务客户端和相应的服务实例之间
- 所有服务调用（内部和外部）都应流经服务网关
- 服务网关提供的能力
  - ✓ 静态路由
  - ✓ 动态路由
  - ✓ 验证和授权
  - ✓ 度量数据收集和日志记录

客户端只关注网关在那里, 把请求给网关, 网关代理访问 (Eureka 不会代理调用, 只告诉目标服务地址, 自己调用)

网关本身就是一个微服务  
可以使用 Zuul 开发

## Zuul

- Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.
- 将应用程序中的所有服务的路由映射到一个URL
- 过滤器



路由能力：将应用中所有的服务路由映射到一个 URL

## 使用Zuul

- pom文件添加依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```
- 启动类添加注解

```
@EnableZuulProxy
```
- 配置zuul与Eureka通信

```
Eureka、Ribbon
```

Zuul 三种过滤器

1. 前置
2. 后置
3. 路由

## 尚硅谷 Spring Cloud

B 站课程: <https://www.bilibili.com/video/BV18E411x7eT?p=1>

概述

课程内容: **Springcloud + Springcloud Alibaba**

(spring cloud 部分组件停更, 但是也非常优秀)

大概 14 个组件技术

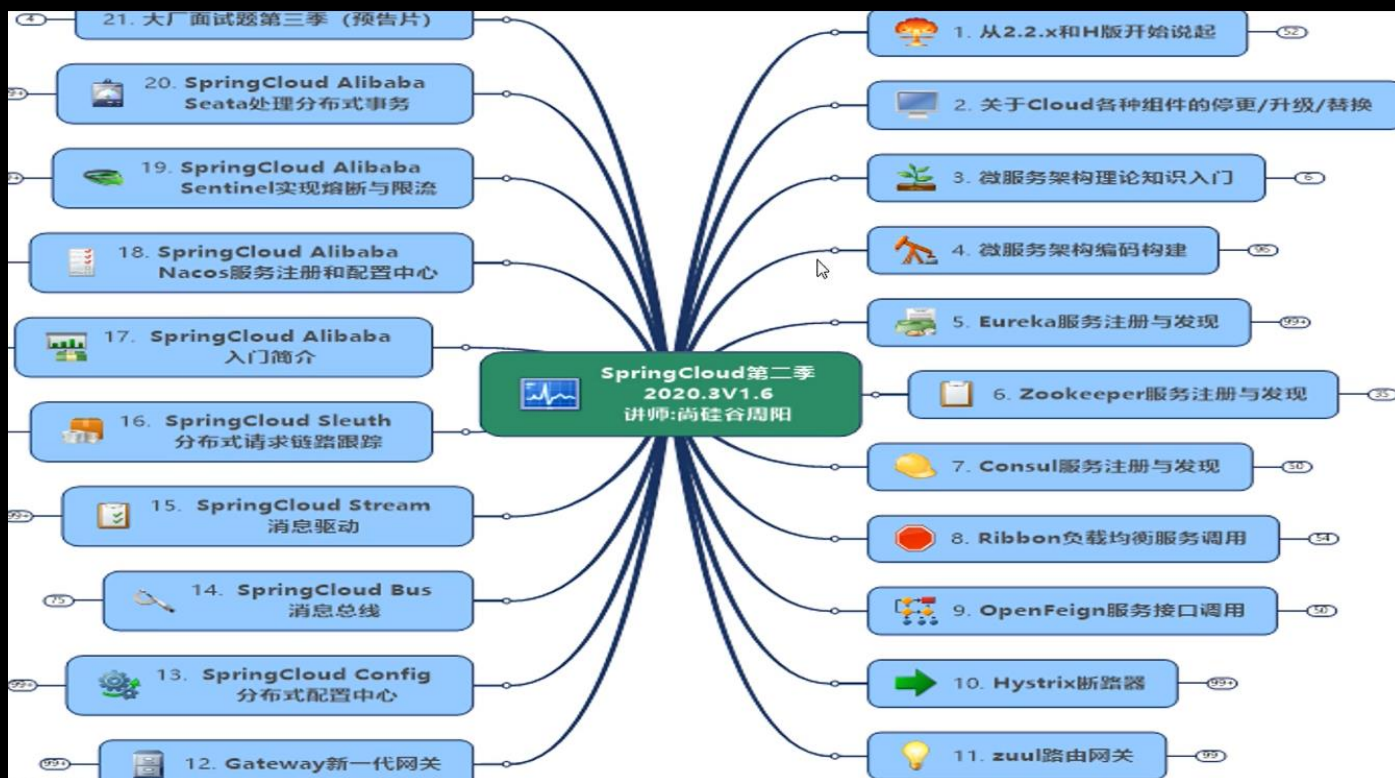
8-10 天

学习思想、代码

技术要求:

Java8+git+maven+nginx+RabbitMQ+springboot

21 章:



“语言少了, 思想就出来了”

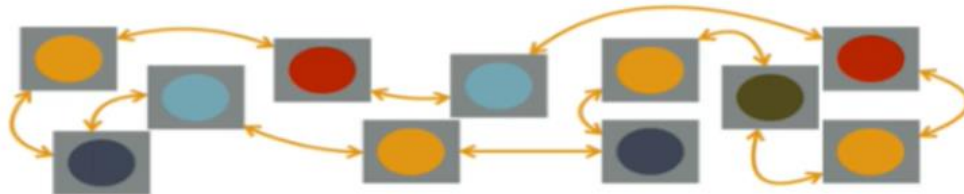
微服务理论



# Microservices - the new architectural style

Martin Fowler, Mar 2014

The **microservice architectural style** is an approach to developing a **single application** as a suite of **small services**, each running in its **own process** and communicating with **lightweight mechanisms** often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

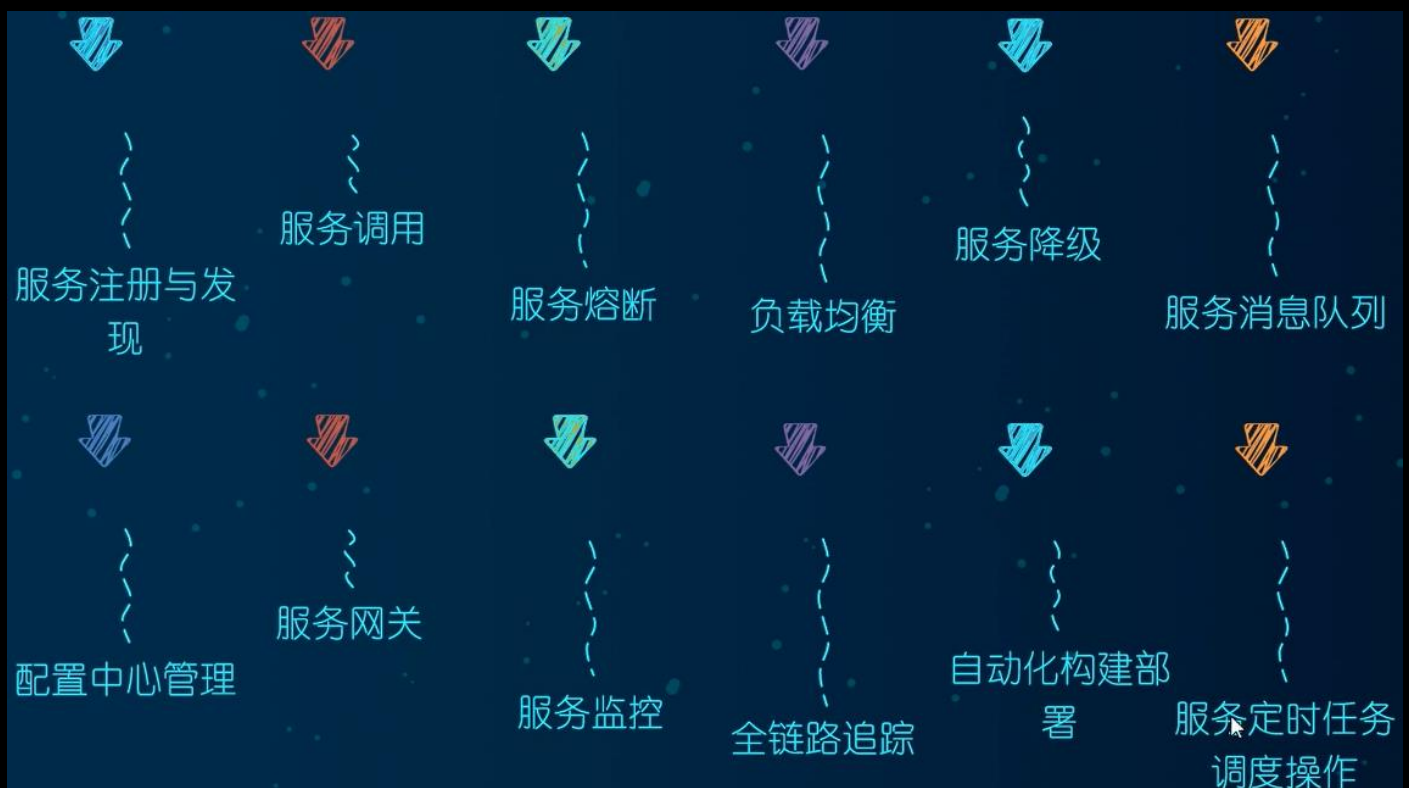


微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于HTTP协议的RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

直白地说，以SpringBoot开发的微小功能模块

基于分布式的微服务架构


















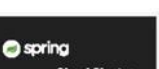
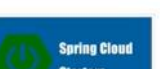


组件/模块：



Spring cloud = 分布式微服务架构的一站式解决方案，是多种微服务架构落地技术的集合体，俗称微服务全家桶。

## Spring Cloud集成相关优质项目推荐

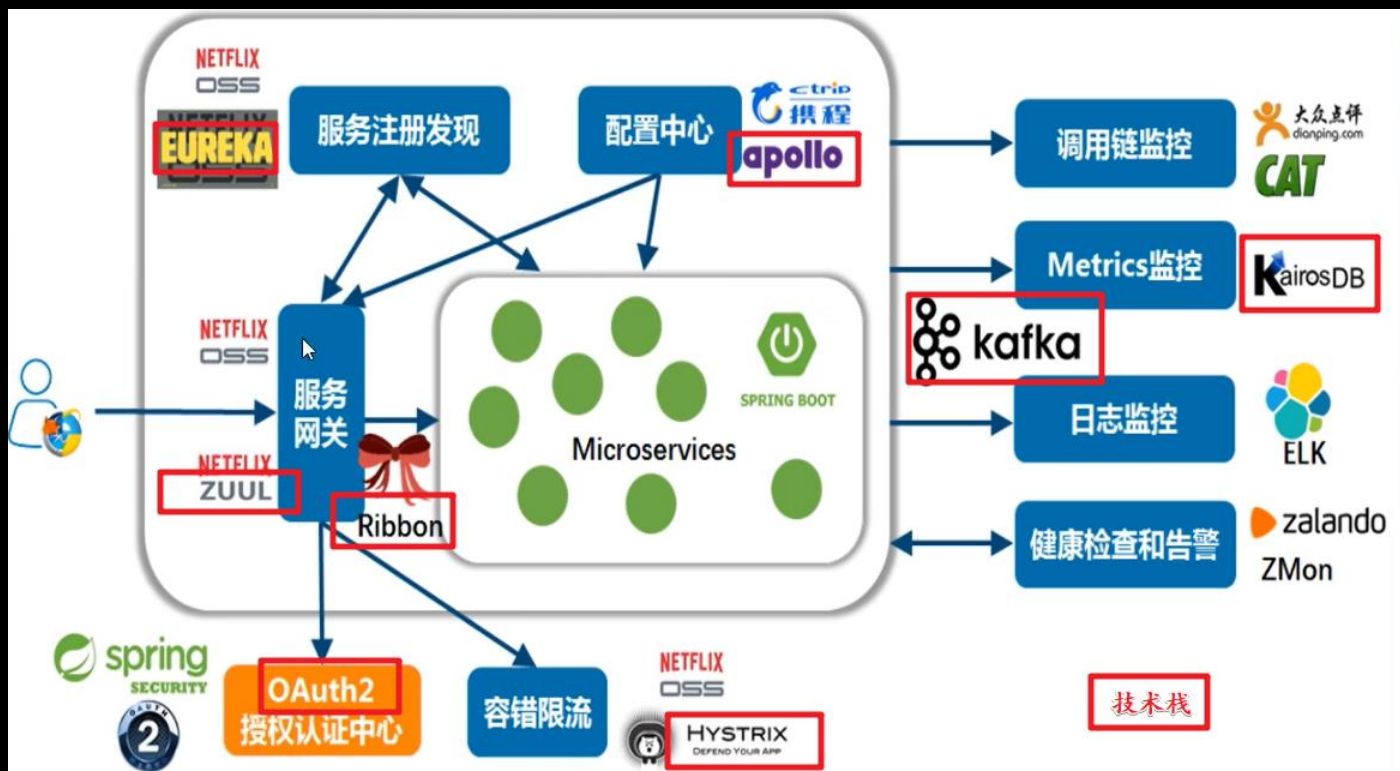
这些项目是Spring Cloud官方项目或是对Spring Cloud进行了有益的补充以及基于Spring Cloud最佳实践。

 <b>Spring Cloud Config</b> Spring 配置管理工具包，让您可以把配置放到远程服务器端，集中化管理您的配置。目前支持本地存储、Git以及Subversion。	 <b>Spring Cloud Bus</b> Spring 事件、消息总线，用于有集群（例如，配置变化事件）中传播状态变化。可与Spring Cloud Config联合实现部署。	 <b>Eureka</b> Netflix 云端服务发现，一个基于REST的服务，用于定位服务，以实现云端中间层服务发现和故障转移。	 <b>Hystrix</b> Netflix 熔断器，容错管理工具，旨在通过增强微服务控制服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。	 <b>Zuul</b> Netflix Zuul是在云平台上提供动态路由、监控、弹性、安全等边缘服务的框架。Zuul相当于设备Netflix流应用的Web网站后端所有请求的网关。	 <b>Archaius</b> Netflix 配置管理API，包含一系列配置管理API，提供动态类型化属性，线程安全配置操作、轮询框架、前缀机制等功能。	 <b>Consul</b> HashiCorp 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。
 <b>Spring Cloud Sleuth</b> Spring 日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTTPTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。	 <b>Spring Cloud Data Flow</b> Pivotal 大数据操作工具，作为Spring XD的替代产品，它是一个混合计算模型，结合了流数据与批处理数据的方式。	 <b>Spring Cloud Security</b> Spring 基于spring security的安全工具包，为您的应用程序添加安全控制。	 <b>Spring Cloud Zookeeper</b> Spring 操作Zookeeper的工具包，用于使用zookeeper方式的开发发现和配置管理。	 <b>Spring Cloud Stream</b> Spring 数据流操作开发包，封装了与Redis, Rabbit, Kafka等发送接收消息。	 <b>Spring Cloud CLI</b> Spring 基于Spring Boot CLI，可以让您以命令行方式快速建立云组件。	 <b>Ribbon</b> Netflix 提供负载均衡均衡，有多种负载均衡策略可供使用，可配合服务发现和断路器使用。
 <b>Feign</b> OpenFeign Feign是一种声明式、模板化的HTTP客户端。	 <b>Spring Cloud Task</b> Spring 提供云端计划任务管理、任务调度。	 <b>Spring Cloud Connectors</b> Spring 便于云端应用程序在各种PaaS平台连接到后端，如：数据库和信息代理服务。	 <b>Spring Cloud Cluster</b> Spring 提供Leadership选举，如：Zookeeper, Redis, Hazelcast, Consul等常见状态模式的抽象和实现。	 <b>Spring Cloud Starters</b> Pivotal Spring Boot式的启动项目，为Spring Cloud提供开箱即用的依赖管理。	 <b>Spring Cloud for Cloud Foundry</b> Pivotal 通过CloudFoundry协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。	 <b>Turbine</b> Netflix Turbine是聚合服务器发送事件流数据的一个工具，用来监控集群Hystrix的metrics情况。

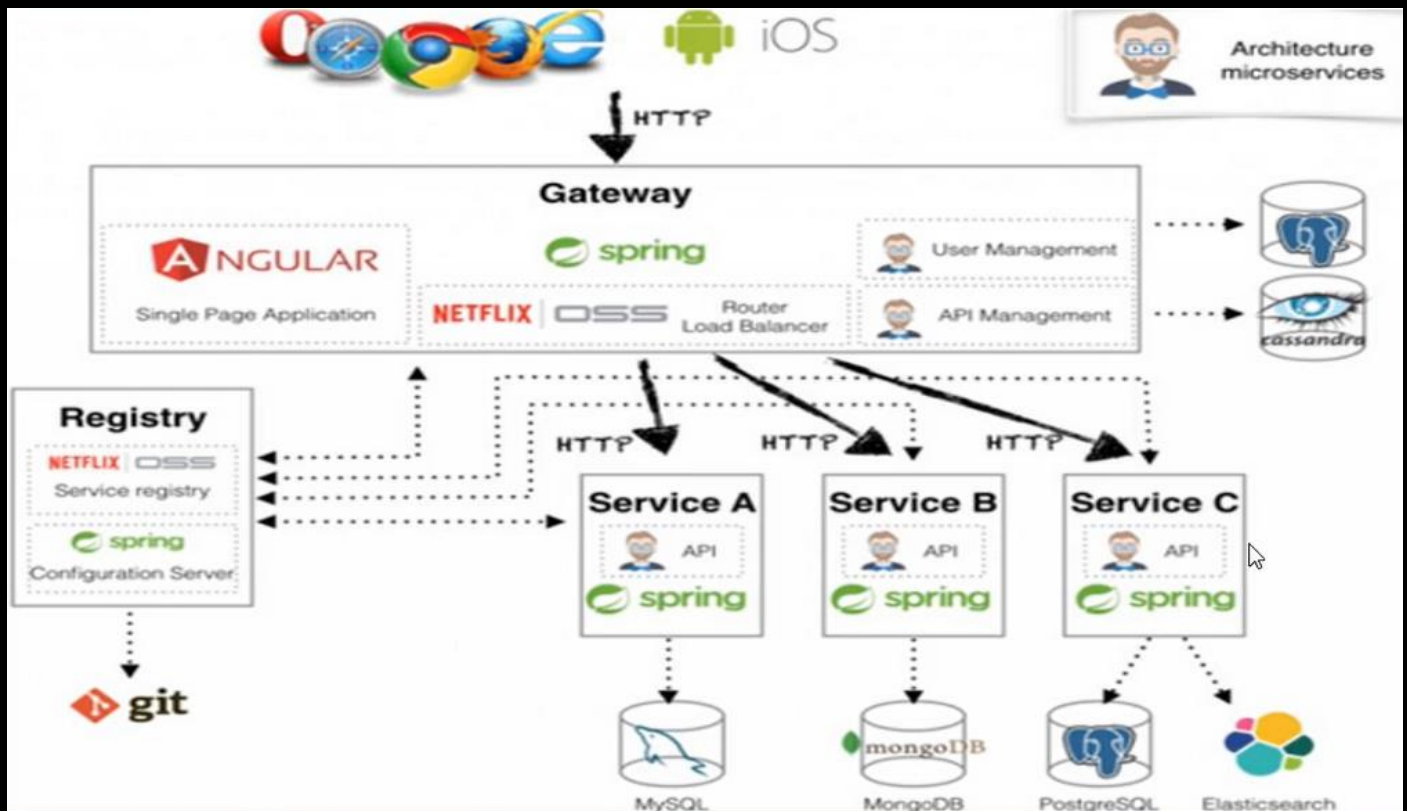








2020/2 后很多组件都有停更和变化



## Boot 和 Cloud 版本选型

Spring Boot 强烈建议升级到 2.0

Spring Cloud 的版本是通过字母命名的：伦敦地铁站名称

- Angel
- Brixton
- Camden
- Dalston
- Edgware
- Finchley
- Greenwich
- Hoxton

Boot 和 Cloud 的依赖关系：

Table 1. Release train Spring Boot compatibility

Release Train	Boot Version
Hoxton	2.2.x
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

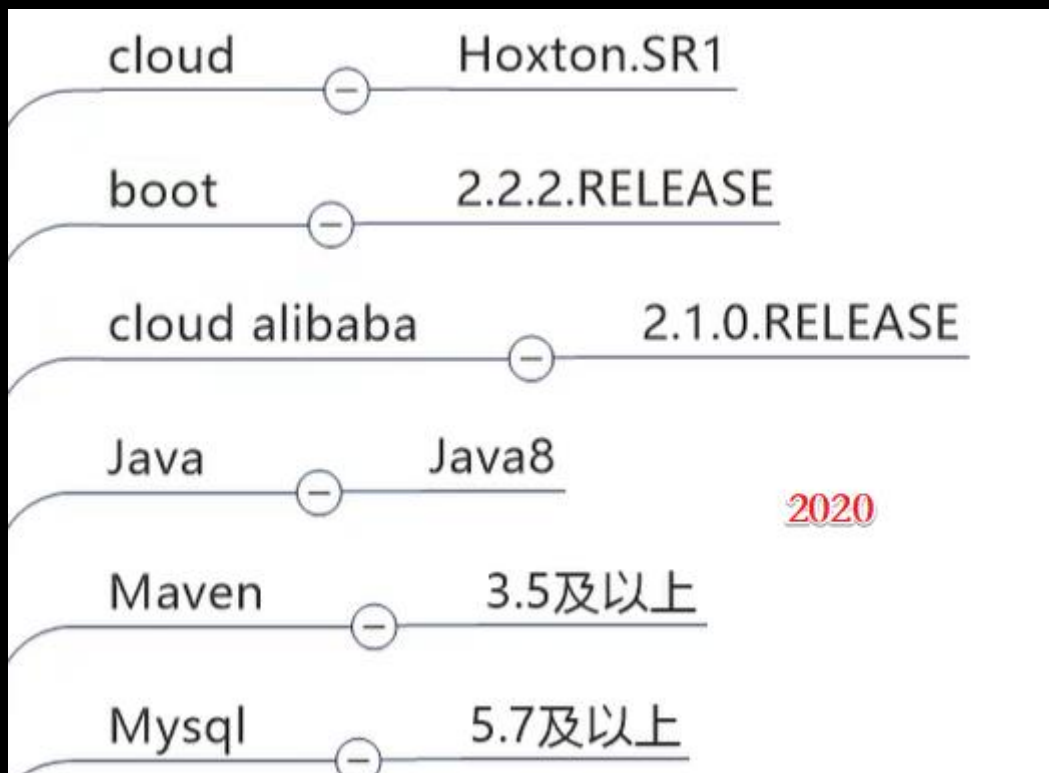
技术选型的官方说明:

<https://start.spring.io/actuator/info>

```
"spring-cloud": {
  "Finchley.M2": "Spring Boot >=2.0.0.M3 and <2.0.0.M5",
  "Finchley.M3": "Spring Boot >=2.0.0.M5 and <=2.0.0.M5",
  "Finchley.M4": "Spring Boot >=2.0.0.M6 and <=2.0.0.M6",
  "Finchley.M5": "Spring Boot >=2.0.0.M7 and <=2.0.0.M7",
  "Finchley.M6": "Spring Boot >=2.0.0.RC1 and <=2.0.0.RC1",
  "Finchley.M7": "Spring Boot >=2.0.0.RC2 and <=2.0.0.RC2",
  "Finchley.M9": "Spring Boot >=2.0.0.RELEASE and <=2.0.0.RELEASE",
  "Finchley.RC1": "Spring Boot >=2.0.1.RELEASE and <2.0.2.RELEASE",
  "Finchley.RC2": "Spring Boot >=2.0.2.RELEASE and <2.0.3.RELEASE",
  "Finchley.SR4": "Spring Boot >=2.0.3.RELEASE and <2.0.999.BUILD-SNAPSHOT",
  "Finchley.BUILD-SNAPSHOT": "Spring Boot >=2.0.999.BUILD-SNAPSHOT and <2.1.0.M3",
  "Greenwich.M1": "Spring Boot >=2.1.0.M3 and <2.1.0.RELEASE",
  "Greenwich.SR5": "Spring Boot >=2.1.0.RELEASE and <2.1.13.BUILD-SNAPSHOT",
  "Greenwich.BUILD-SNAPSHOT": "Spring Boot >=2.1.13.BUILD-SNAPSHOT and <2.2.0.M4",
  "Hoxton.SR1": "Spring Boot >=2.2.0.M4 and <2.2.5.BUILD-SNAPSHOT",
  "Hoxton.BUILD-SNAPSHOT": "Spring Boot >=2.2.5.BUILD-SNAPSHOT and <2.3.0.M1"
}
```

由 Cloud 决定 Boot 的版本



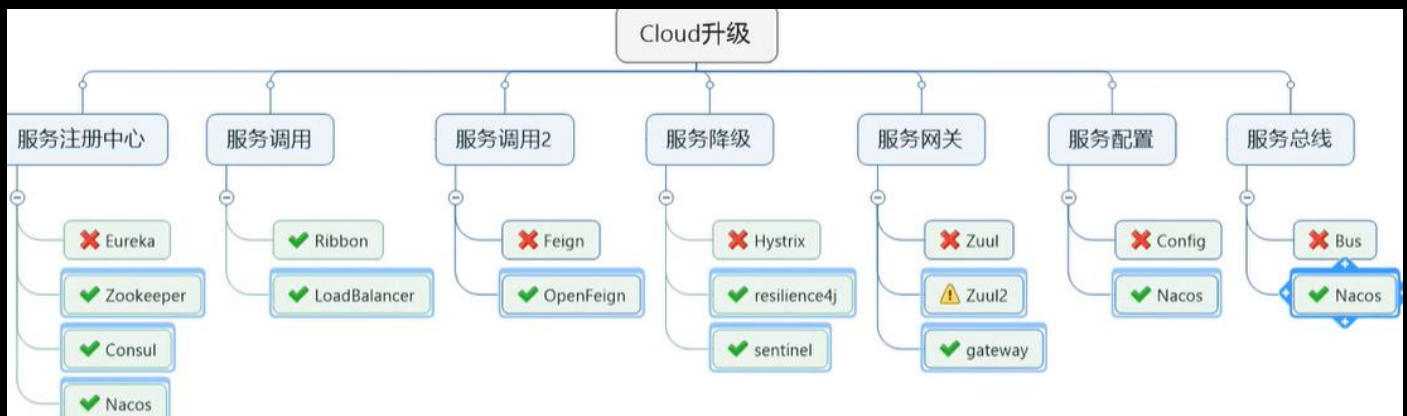


## Cloud 组件停更说明

### Cloud 升级:

- 服务注册中心:
  - ◆ Eureka: No, 停更
  - ◆ Zookeeper: Yes
  - ◆ Consul: Go 语言写的, 不推荐
  - ◆ Nacos: 阿里, 百万级流量验证, 推荐
- 服务调用
  - Ribbon: 还可以用
  - LoadBalancer: Yes
- 服务调用 2
  - Feign: 停更, 不再用了
  - OpenFeign
- 服务降级
  - Hystrix: 仍在用, 但是不太推荐
  - Resilience4j: 国外, 国内比较少
  - Sentinel: 阿里, 强烈推荐
- 服务网关
  - Zuul: 不用, netflix 的
  - Gateway: spring 的, 主流
- 服务配置
  - Config: 不在使用
  - Apollo: 携程的
  - Nacos: 阿里, 推荐
- 服务总线

- Bus: 不推荐
- Nacos



Spring Cloud中文文档

<https://www.bookstack.cn/read/spring-cloud-docs/docs-index.md>

## 父工程建立

约定>配置>编码

父工程:

Project

- Module
- Module2
- ...

```
<!-- 子模块继承之后, 提供作用: 锁定版本+子module不用写groupId和version -->
<dependencyManagement>
  <dependencies>
    <!--spring boot 2.2.2-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.2.2.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!--spring cloud Hoxton.SR1-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Hoxton.SR1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!--spring cloud alibaba 2.1.0.RELEASE-->
```

dependencyManagement

通常会在一个组织或者项目的最顶层的父POM 中看到dependencyManagement 元素。

<pre>&lt;dependencyManagement&gt; &lt;dependencies&gt; &lt;dependency&gt; &lt;groupId&gt;mysql&lt;/groupId&gt; &lt;artifactId&gt;mysql-connector-java&lt;/artifactId&gt; &lt;version&gt;5.1.2&lt;/version&gt; &lt;/dependency&gt; ... &lt;/dependencies&gt;</pre>	<pre>&lt;dependencies&gt; &lt;dependency&gt; &lt;groupId&gt;mysql&lt;/groupId&gt; &lt;artifactId&gt;mysql-connector-java&lt;/artifactId&gt; &lt;/dependency&gt; &lt;/dependencies&gt;</pre>
---	---

父POM

子POM: 不用指定版本号

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.3.7.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
  <version>2.2.4.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <version>2.2.4.RELEASE</version>
  <scope>runtime</scope>
  <optional>true</optional>
</dependency>

<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>1.3.2</version>
</dependency>

<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid-spring-boot-starter</artifactId>
  <version>1.2.5</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.22</version>
</dependency>

<dependency>
  <groupId>org.projectlombok</groupId>
```

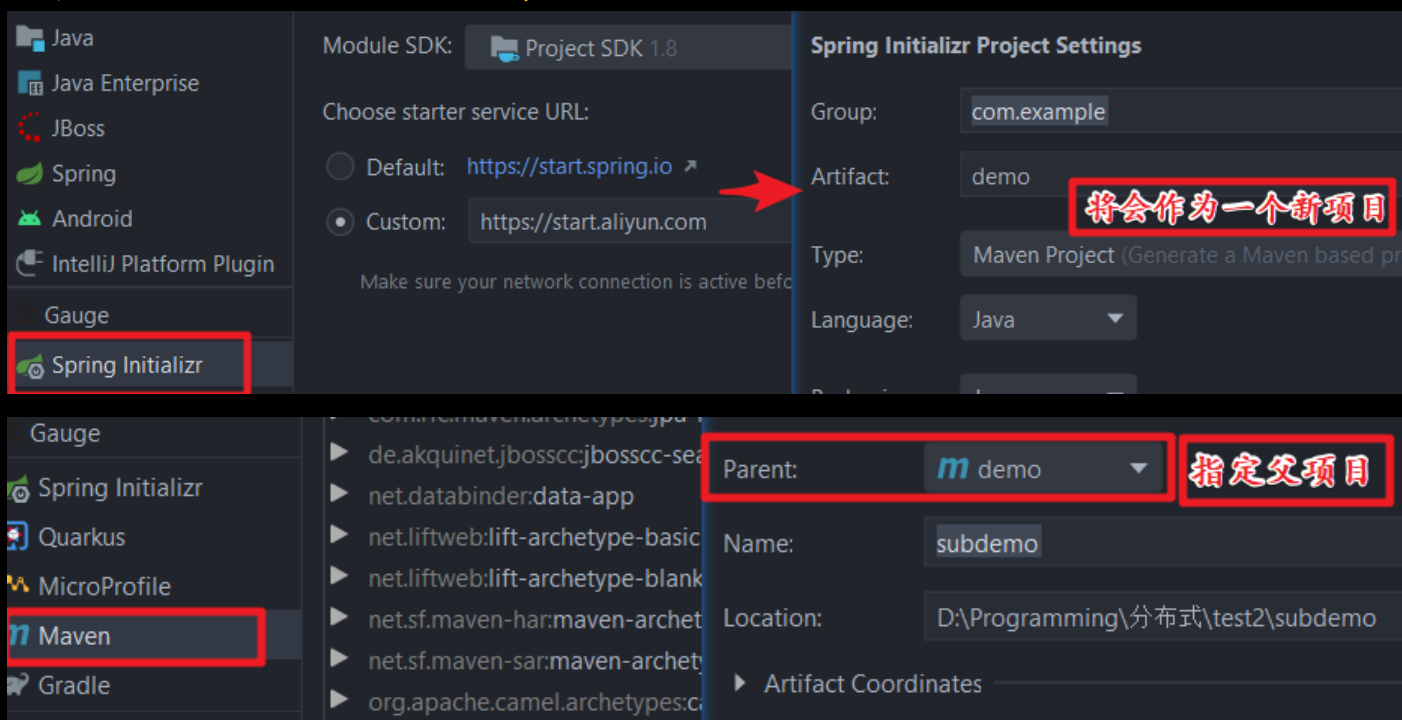
```
<artifactId>lombok</artifactId>
<version>1.18.16</version>
</dependency>
```

Yml:

```
server:
  port: 8080
spring:
  application:
    name: cloud-payment-service
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    url:
jdbc:mysql://localhost:3306/springcloud_payment?characterEncoding=utf8&serverTimezone=UTC
  username: root
  password: root
  driver-class-name: com.mysql.cj.jdbc.Driver
mybatis:
  mapper-locations: classpath:mapper/*Mapper.xml
  type-aliases-package: cn.edw.springcloudpayment.dao
```

IDEA 构建项目子模块的注意事项:

- 对于一个父项目,如果要构建子项目,子项目不能选择 **Spring** 项目,必须选择 **Maven** 项目,否则那个子项目将会替代原本的父项目,原来父项目的代码将会被清空。



## 热部署 DevTools

开启步骤:

- 添加到 devtools 依赖  
Spring-boot-devtools
- 添加插件

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <fork>true</fork>
        <addResources>true</addResources>
      </configuration>
    </plugin>
  </plugins>

```

- 开启自动编译

- ☒ Automatically show first error in editor
- ☒ Display notification on build completion
- ☒ Build project automatically
- ☒ Compile independent modules in parallel
- ☒ Rebuild module on dependency change

- 开启

press `ctrl+shift+Alt+/,` and search for the registry. In the Registry, enable :

- ☒ `compiler.automake.allow.when.app.running`

Key	
<code>compiler.automake.allow.when.app.running</code>	<input checked="" type="checkbox"/>
<code>actionSystem.assertFocusAccessFromEdt</code>	<input checked="" type="checkbox"/>

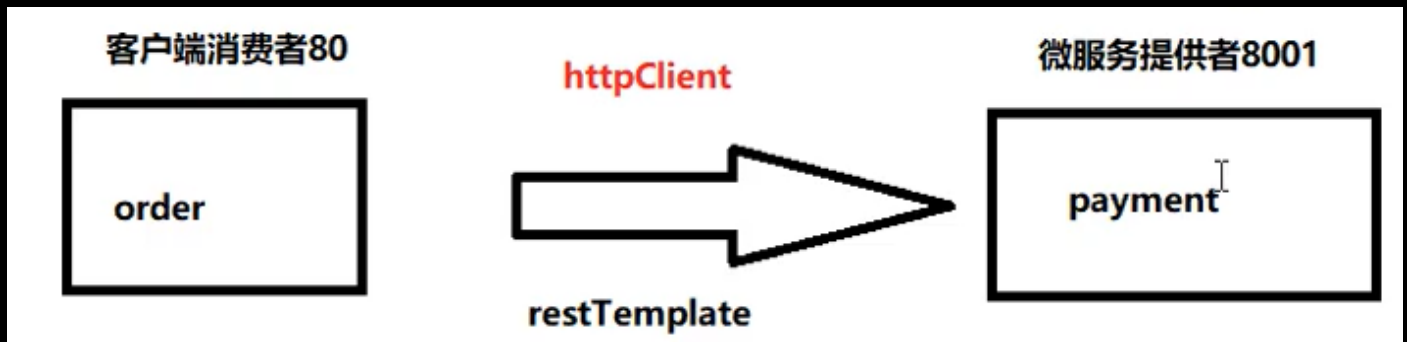
开发工具，部署时去掉

## 支付模块

传统的 SSM 架构，详情见代码

## 消费者订单模块





## RestTemplate

RestTemplate 提供了多种便捷访问远程 HTTP 服务的方法，是一种简单的访问 Restful 服务模板类，是 Spring 提供的用于访问 REST 服务的客户端模板工具类。

[docs.spring.io/spring-framework/docs/5.2.2.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html](https://docs.spring.io/spring-framework/docs/5.2.2.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html)

使用 restTemplate 访问 restful 接口非常的简单粗暴无脑。  
(url, requestMap, ResponseBean.class) 这三个参数分别代表  
REST 请求地址、请求参数、HTTP 响应转换被转换成的对象类型。

Spring 中需要注入到容器，于是需要 config:

```
@Configuration
public class ApplicationContextConfig
{
    @Bean
    public RestTemplate getRestTemplate()
    {
        return new RestTemplate();
    }
}
```

```
public class OrderController
{
    @Resource
    private RestTemplate restTemplate;
}
```



```

@RestController
@RequestMapping("/consumer")
public class OrderController {
    private static final String PAYMENT_URL = "http://localhost:8080";

    @Resource
    private RestTemplate restTemplate;

    @PostMapping("/payment/create")
    public Response create(@RequestBody Payment payment){
        // POST: URL, Body, Response Class
        return restTemplate.postForObject(url: PAYMENT_URL+"/payment/create",payment,Response.class);
    }

    @GetMapping("/payment/get")
    public Response getPaymentByID(@PathParam("id") int id){
        return restTemplate.getForObject(url: PAYMENT_URL+"/payment/get?id="+id,Response.class);
    }
}

```

使用RestTemplate请求其他服务

如果有众多的项目，使用 IDEA 的 run dashboard

## 工程重构

上面两个微服务都有一样的实体类 po——各微服务相同的部分需要提取出来

构建一个新的 module——commons，用于存储公共的类/代码

```

▶ cloud-api-commons
▶ cloud-consumer-order
▶ cloud-provider-payment [payment]

```

把 commons module 打包，发布，供其他模块使用

注意 groupId 和 artifactId

## Hutool

好用的工具类：Hutool

Hutool 是一个小而全的Java工具类库，通过静态方法封装，降低相关API的学习成本，提高工作效率，使Java拥有函数式语言般的优雅，让Java语言也可以“甜甜的”-->

```

<dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>5.1.0</version>
</dependency>

```

1. 日期与字符串转换
2. 文件操作
3. 转码与反转码
4. 随机数生成
5. 压缩与解压
6. 编码与解码
7. CVS 文件操作
8. 缓存处理
9. 加密解密
10. 定时任务
11. 邮件收发
12. 二维码创建
13. FTP 上传与下载
14. 图形验证码生成
15. ...

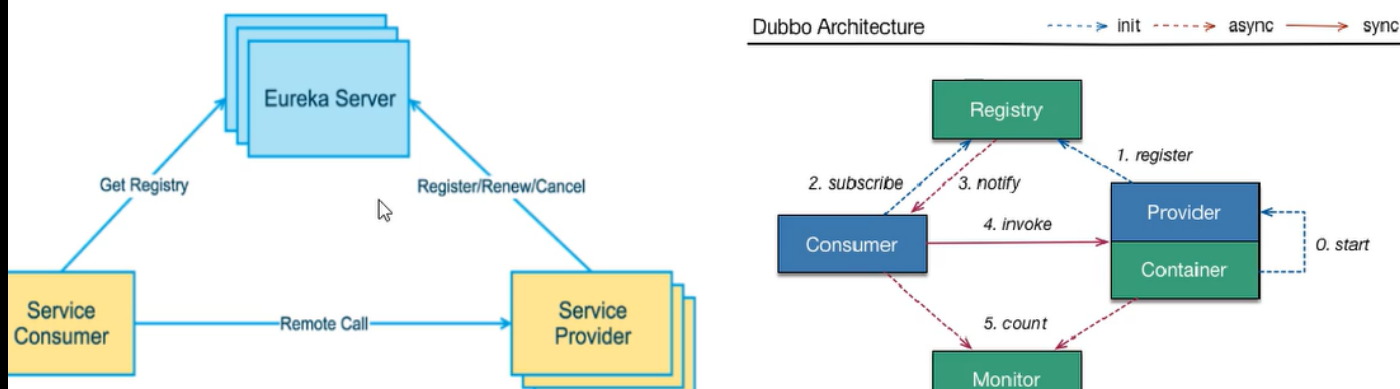
## Eureka 基础知识

什么是**服务治理**？

Spring Cloud 封装了 Netflix 开发的 Eureka 模块来实现 **服务治理**

在传统的 RPC 远程调用框架中，管理每个服务与服务之间依赖关系比较复杂，管理比较复杂，所以需要**使用服务治理，管理服务之间的依赖关系，可以实现服务调用、负载均衡、容错等，实现服务的发现与注册。**

下左图是Eureka系统架构，右图是Dubbo的架构，请对比



Eureka采用了CS的设计架构，Eureka Server 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接。这样系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。**心跳检测**

在服务注册与发现中，有一个**注册中心**，当服务器启动的时候，会把当前自己服务器的信息 比如 服务地址通讯地址等以别名方式注册到注册中心上。另一方（消费者|服务提供者），以该别名的方式去注册中心上获取到实际的服务通讯地址，然后再实现本地RPC调用RPC远程调用框架核心设计思想：在于注册中心，因为使用注册中心管理每个服务与服务之间的一个依赖关系(服务治理概念)。在任何rpc远程框架中，都会有一个注册中心(存放服务地址相关信息(接口地址))

Eureka包含两个组件：Eureka Server和Eureka Client

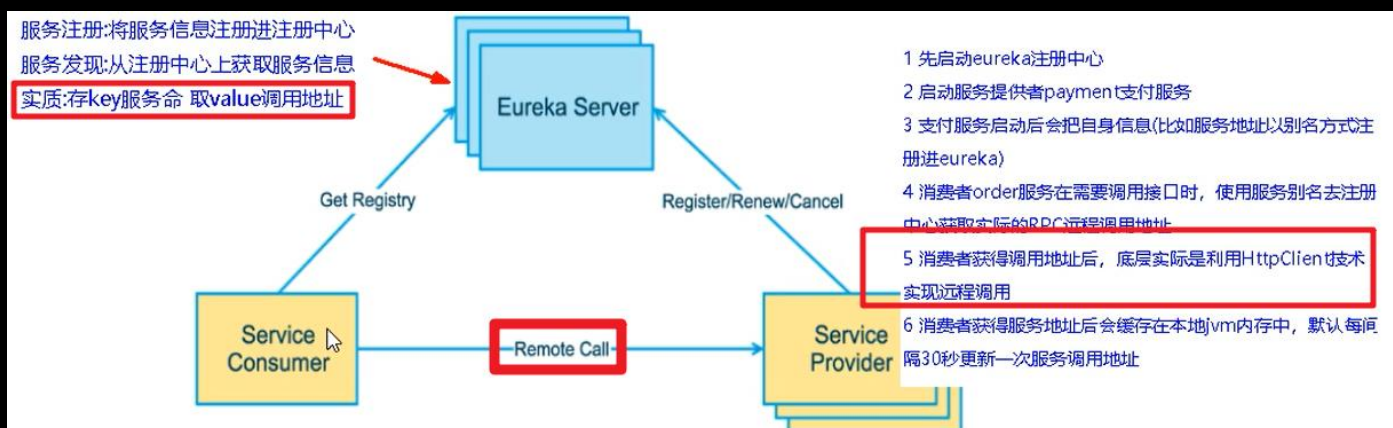
Eureka Server提供服务注册服务

各个微服务节点通过配置启动后，会在EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观看到。

EurekaClient通过注册中心进行访问

是一个Java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳(默认周期为30秒)。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除(默认90秒)

## Eureka 工作原理



Eureka Server：注册中心服务端

注册中心服务端主要对外提供了三个功能：

### 1. 服务注册

服务提供者启动时，会通过 Eureka Client 向 Eureka Server 注册信息，Eureka Server 会存储该服务的信息，Eureka Server 内部有二层缓存机制来维护整个注册表

### 2. 提供注册表

服务消费者在调用服务时，如果 Eureka Client 没有缓存注册表的话，会从 Eureka Server 获取最新的注册表

### 3. 同步状态

Eureka Client 通过注册、心跳机制和 Eureka Server 同步当前客户端的状态。

Eureka Client：注册中心客户端

Eureka Client 是一个 Java 客户端，用于简化与 Eureka Server 的交互。Eureka Client 会拉取、更新和缓存 Eureka Server 中的信息。因此当所有的 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，但是当服务有更改的时候会出现信息不一致。

Register：服务注册

服务的提供者，将自身注册到注册中心，服务提供者也是一个 Eureka Client。当 Eureka Client 向 Eureka Server 注册时，它提供自身的元数据，比如 IP 地址、端口，运行状况指示符 URL，主页等。

### Renew: 服务续约

Eureka Client 会每隔 30 秒发送一次心跳来续约。通过续约来告知 Eureka Server 该 Eureka Client 运行正常, 没有出现问题。默认情况下, 如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的续约, Server 端会将实例从其注册表中删除, 此时间可配置, 一般情况不建议更改。

服务续约任务的调用间隔时间, 默认为 30 秒

```
eureka.instance.lease-renewal-interval-in-seconds=30
```

服务失效的时间, 默认为 90 秒。

```
eureka.instance.lease-expiration-duration-in-seconds=90
```

### Cancel: 服务下线

Eureka Client 在程序关闭时向 Eureka Server 发送取消请求。发送请求后, 该客户端实例信息将从 Eureka Server 的实例注册表中删除。该下线请求不会自动完成, 它需要调用以下内容:

```
DiscoveryManager.getInstance().shutdownComponent();
```

### GetRegistry: 获取注册列表信息

Eureka Client 从服务器获取注册表信息, 并将其缓存在本地。客户端会使用该信息查找其他服务, 从而进行远程调用。该注册列表信息定期 (每 30 秒钟) 更新一次。每次返回注册列表信息可能与 Eureka Client 的缓存信息不同, Eureka Client 自动处理。

## 自我保护机制

默认情况下, 如果 Eureka Server 在一定的 90s 内没有接收到某个微服务实例的心跳, 会注销该实例。但是在微服务架构下服务之间通常都是跨进程调用, 网络通信往往会面临着各种问题, 比如微服务状态正常, 网络分区故障, 导致此实例被注销。

固定时间内大量实例被注销, 可能会严重威胁整个微服务架构的可用性。为了解决这个问题, Eureka 开发了自我保护机制

Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%, 如果低于 85%, Eureka Server 即会进入自我保护机制。

Eureka Server 触发自我保护机制后, 页面会出现提示

```
EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.
```

Eureka Server 进入自我保护机制, 会出现以下几种情况:

1. (1 Eureka 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. (2 Eureka 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(即保证当前节点依然可用)
3. (3 当网络稳定时, 当前实例新的注册信息会被同步到其它节点中



Eureka 自我保护机制是为了防止误杀服务而提供的一个机制。当个别客户端出现心跳失联时，则认为客户端的问题，剔除掉客户端；当 Eureka 捕获到大量的心跳失败时，则认为可能是网络问题，进入自我保护机制；当客户端心跳恢复时，Eureka 会自动退出自我保护机制。

通过在 Eureka Server 配置如下参数，开启或者关闭保护机制，生产环境建议打开：

```
eureka.server.enable-self-preservation=true
```

```
eureka:
  server:
    #关闭自我保护机制，保证不可用服务被及时踢除
    enable-self-preservation: false
    eviction-interval-timer-in-ms: 2000
```

属于 CAP 中的 AP，舍去了一致性，追求可用性和分区容错性

## 单机版 Eureka

IDEA生成eurekaServer端服务注册中心  
类似物业公司

11

EurekaClient端cloud-provider-payment8001

将注册进EurekaServer成为服务提供者provider，类似尚硅谷学校对外提供授课服务

EurekaClient端cloud-consumer-order80

将注册进EurekaServer成为服务消费者consumer，类似来尚硅谷上课消费的各位同学

依赖：

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka-server</artifactId>
<version>1.4.7.RELEASE</version>
</dependency> OR
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

springcloud 更新换代比较快，可能 1.5 可以使用，到了 2.0 就不用了。所以做项目或者练习时要看清自己使用的版本。1.5 版本使用 spring-cloud-starter-eureka-server 还是没问题的。2.0 以上建议使用 **spring-cloud-starter-netflix-eureka-server**。

依赖实际使用中可能产生冲突：

**spring-cloud-starter-netflix-eureka-server** 不要引入 **servlet-api**

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
```

```

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
<version>2.2.6.RELEASE</version>

<exclusions>
  <exclusion>
    <artifactId>servlet-api</artifactId>
    <groupId>javax.servlet</groupId>
  </exclusion>
</exclusions>
</dependency>

```

不需要其他业务代码，只需要注解**@EnableEurekaServer**

```

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

```

**@EnableEurekaServer**


Eureka 的配置：

```

server:
  port: 8001 (Eureka 本身是一个 SpringBoot 服务)
eureka:
  instance:
    hostname: localhost
  client:
    # 不向注册中心注册自己
    register-with-eureka: false
    # false 表示自己端就是注册中心，职责就是维护服务实例，不需要检索服务
    fetch-registry: false
    # 与 Eureka Server 交互的服务都需要这个地址
    service-url:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

```

在端口就可以看到 Eureka 服务：


HOME
LAST 1000 SINCE STARTUP

### System Status

Environment	test	Current time	2021-03-04T14:33:17 +0800
Data center	default	Uptime	00:01

## Eureka 配置

<https://www.jianshu.com/p/0395cc887d82>

eureka.instance.instance-id: 修改实例名称



# EurekaClient

Eureka Server 启动后，需要把服务 EurekaClient 入驻进去。

使用步骤：

## 1. 引入依赖：spring-cloud-starter-netflix-eureka-client (注意是 Client)

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
  <version>2.2.6.RELEASE</version>

  <exclusions>
    <exclusion>
      <artifactId>servlet-api</artifactId>
      <groupId>javax.servlet</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

## 2. 配置

```
spring:
  application:
    name: cloud-payment-service #是入驻 Eureka 中的后的名字，需要注意
```

```
eureka:
  client:
    # 向注册中心注册自己
    register-with-eureka: true
    # 是否从 EurekaServer 抓取已有的注册信息，默认 true。
    # 单节点无所谓，集群必须设计为 true 才能配置 Ribbon 负载均衡
    fetch-registry: true
    # 与 Eureka Server 交互的服务都需要这个地址
    service-url:
      defaultZone: http://localhost:8001/eureka
```

## 3. 启动类注解：@EnableEurekaClient

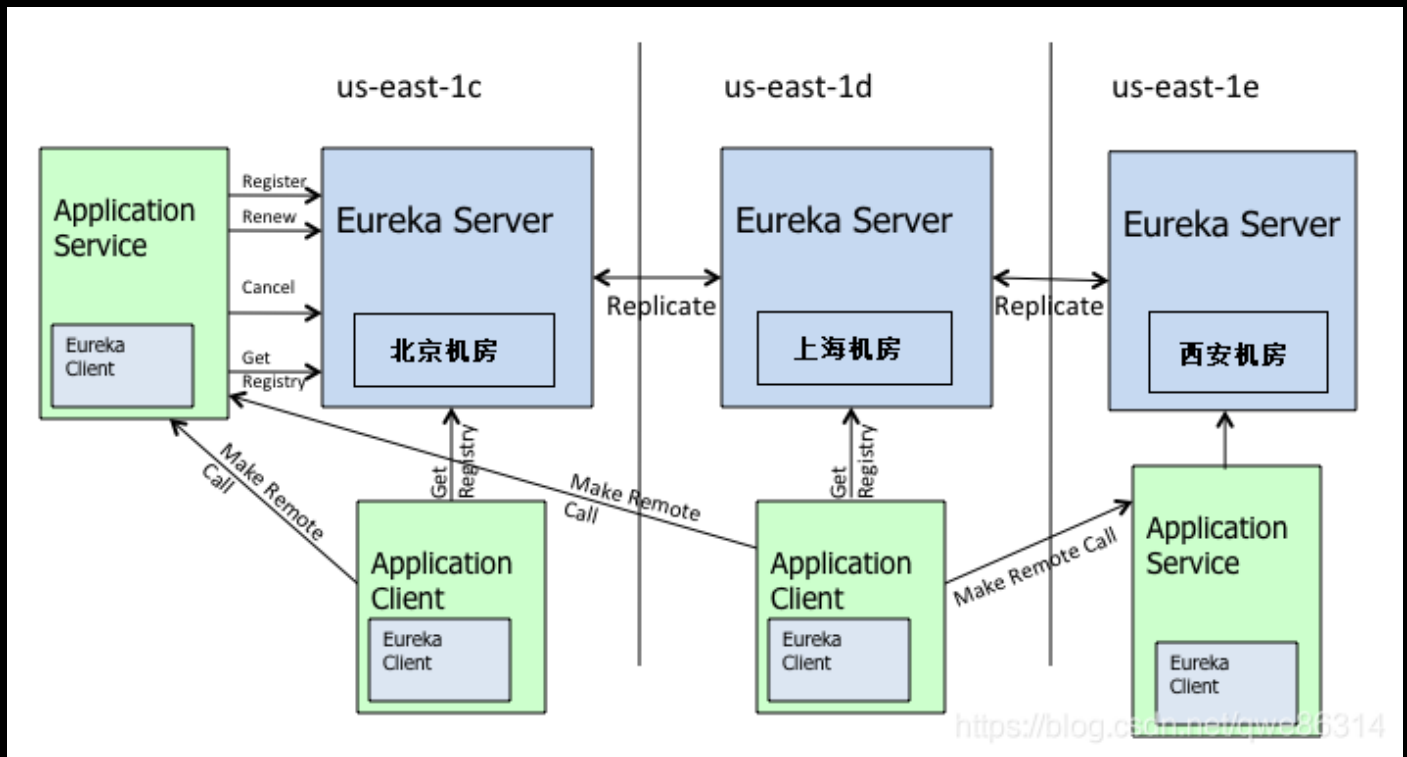
```
@SpringBootApplication
@EnableEurekaClient
public class PaymentApplication {
    public static void main(Str:
}
```

## 4. 查看 Eureka 上的实例

Application	AMIs	Availability Zones	Status
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - 172.31.119.71:cloud-payment-service:8080

# Eureka 集群原理

单机 Eureka 不是高可用的  
原理：互相注册、相互守望



集群相互之间通过 **Replicate** 来同步数据：节点之间相互注册

**Eureka Server** 集群之间的状态是采用异步方式同步的，所以不保证节点间的状态一定是一致的，不过基本能保证最终状态是一致的。

### Eureka 分区

Eureka 提供了 **Region** 和 **Zone** 两个概念来进行分区，这两个概念均来自于亚马逊的 **AWS**：

**region**：可以理解为地理上的不同区域，比如亚洲地区，中国区或者深圳等等。没有具体大小的限制。根据项目具体的情况，可以自行合理划分 **region**。

**zone**：可以简单理解为 **region** 内的具体机房，比如说 **region** 划分为深圳，然后深圳有两个机房，就可以在此 **region** 之下划分出 **zone1**、**zone2** 两个 **zone**。

## Eureka 集群搭建

1. 建立两个（多个）Eureka Server

2. 修改 host 映射

由于现在是本机模拟集群，所以 localhost 不能为多个集群使用，需要修改一下域名映射。

修改 hosts 文件：C:\Windows\System32\drivers\etc\hosts

ipconfig /flushdns

3. 改配置 yml

```
eureka:
  instance:
    # 改主机映射后, 集群环境下不能使用localhost
    hostname: eureka8002.com
  client:
    # 不向注册中心注册自己
    register-with-eureka: false
    # false表示自己端就是注册中心, 职责就是维护服务实例, 不需要检索服务
    fetch-registry: false
    # 与Eureka Server交互的服务都需要这个地址
    service-url:
      # defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
      # 其他集群
      defaultZone: http://eureka8001.com:8001/eureka/
```

#### 4. Client 注册到所有集群

虽然注册到一个集群, 集群间也会复制, 但是还是存在单点故障可能。  
把所有 Eureka Server 地址全部写到 defaultZone 即可, 逗号分隔

```
service-url:
  #defaultZone: http://localhost:8001/eureka
  defaultZone: http://eureka8001.com:8001/eureka,http://eureka8002.com:8002/eureka
```

#### 5. 查看集群: DS Replicas 表示集群内的其他成员

DS Replicas

eureka8002.com

eureka8001.com

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - 172.31.119.71:cloud-order-service:8080
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - 172.31.119.71:cloud-payment-service:80

DS Replicas

eureka8001.com

eureka8001.com

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - 172.31.119.71:cloud-order-service:8080
CLOUD-PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - 172.31.119.71:cloud-payment-service:80

## 服务提供者集群配置

服务提供者应该也是集群, 以保证最大的高可用性

操作步骤:

1. 复制 2 个相同的服务提供者，这里是 cloud-provider-payment 并且分布在不同的端口(部署时是不同服务器) 注意他们的服务名是一样的

```
application:  
  name: cloud-payment-service
```

为了在 consumer 端调用时可以看到使用的那个服务实例，可以在 cloud-payment-service 的返回数据中说明端口：

```
@Value("${server.port}")  
private int serverPort;  
  
return Response.buildSuccess("ok,server  
port="+serverPort,paymentDao.create(payment));
```

2. 查看 Eureka 实例

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status	多个服务提供者实例
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - 172.31.119.71:cloud-order-service:80	
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - 172.31.119.71:cloud-payment-service:8081 , 172.31.119.71:cloud-payment-service:8080	

3. 修改服务消费端 cloud-order-service

cloud-order-service 本来是通过 RestTemplate 访问 payment 的，但是现在由于 payment 有多个实例可选，于是可以让 RestTemplate 自动选择——负载均衡，默认是 Ribbon 策略。

首先，修改 RestTemplate 访问的 HTTP 地址：

```
public class OrderController {  
    // private static final String PAYMENT_URL = "http://localhost:8080";  
    // 服务访问地址变更为 Eureka 上的服务名 而不关系具体是哪个实例  
    private static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";  
}
```

注意这个表示： http://服务名

其次，在 RestTemplate 配置中，加入@LoadBalanced

```
@Bean  
// 负载均衡  
@LoadBalanced  
public RestTemplate getRestTemplate() { return new RestTemplate(); }
```

4. OK，现在实现了轮询的负载均衡。

## spring-boot-starter-actuator

actuator 是监控系统健康情况的工具

```
<dependency>
|   <groupId>org.springframework.boot</groupId>
|   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
|   <groupId>org.springframework.boot</groupId>
|   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

最好一起导入

<https://www.jianshu.com/p/481134c3fab7>

## Eureka 服务发现 Discovery

每一个服务都可以将自己的相关信息暴露出去，共其他服务发现、使用

```
@Resource
private org.springframework.cloud.client.discovery.DiscoveryClient discoveryClient;

@GetMapping("/discovery")
public Response discovery(){
    return Response.buildSuccess(msg: "ok",discoveryClient);
}
```

启动类添加注解@EnableDiscoveryClient

## SpringCloud 与 Dubbo 区别

	Dubbo	SpringCloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务监控	Dubbo-monitor	Spring Boot Admin
熔断器	不完善	Spring Cloud Netflix Hystrix
服务网关	无	Spring Cloud Netflix Zuul
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task
信息总线	无	Spring Cloud Bus

最大的区别：Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession



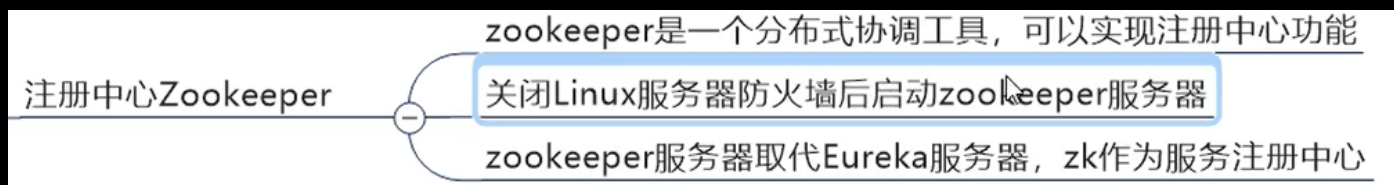
序列化完成 RPC 通信。而 **SpringCloud** 是基于 **Http 协议+rest 接口调用远程过程的通信**，相对来说，Http 请求会有更大的报文，占的带宽也会更多。但是 REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适，至于注重通信速度还是方便灵活性，具体情况具体考虑。

背景区别：Dubbo 是来源于阿里团队，SpringCloud 是来源于 Spring 团队，Spring 广泛遍布全球各种企业开发中，可以确保 SpringCloud 的后续更新维护，Dubbo 虽然来自国内顶尖的阿里团队，但是曾经被阿里弃用停更，但是后来阿里又低调重启维护。

定位区别：Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spirng、Spirng Boot 的优势之上，两个框架在开始目标就不一致，Dubbo 定位服务治理、Spirng Cloud 是一个生态。因此可以大胆地判断，Dubbo 未来会在服务治理方面更为出色，而 Spring Cloud 在微服务治理上面无人能敌。

## Zookeeper 服务注册与发现

Eureka 停更不停用，不建议使用了，使用 Zookeeper 可以替代



使用步骤：

- 依赖

```
<!-- SpringBoot整合zookeeper客户端 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
</dependency>
```

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
  <!-- 先排除自带的zookeeper3.5.3-->
  <exclusions>
    <exclusion>
      <groupId>org.apache.zookeeper</groupId>
      <artifactId>zookeeper</artifactId>
    </exclusion>
  </exclusions>
</dependency>

```

- 配置

```

#服务别名----注册zookeeper到注册中心名称
spring:
  application:
    name: cloud-provider-payment
  cloud:
    zookeeper:
      connect-string: 192.168.111.144:2181

```

- 主启动类
- @EnableDiscoveryClient

## Consul 服务注册与发现

### 第 31 讲

Apache Dubbo | 'dʌbʊ | 提供了六大核心能力：面向接口代理的高性能 RPC 调用，智能容错和负载均衡，服务自动注册和发现，高度可扩展能力，运行期流量调度，可视化的服务治理与运维。