Springboot cloud 快速入手

引入springcloud的原因

服务之间松耦合，每个服务高度自制，可以单独运行，服务之间采用轻量通信协议的高度可持续部署的分布式架构

松耦合：一个是异步通信，restfulApi调用，一方修改不影响另一方

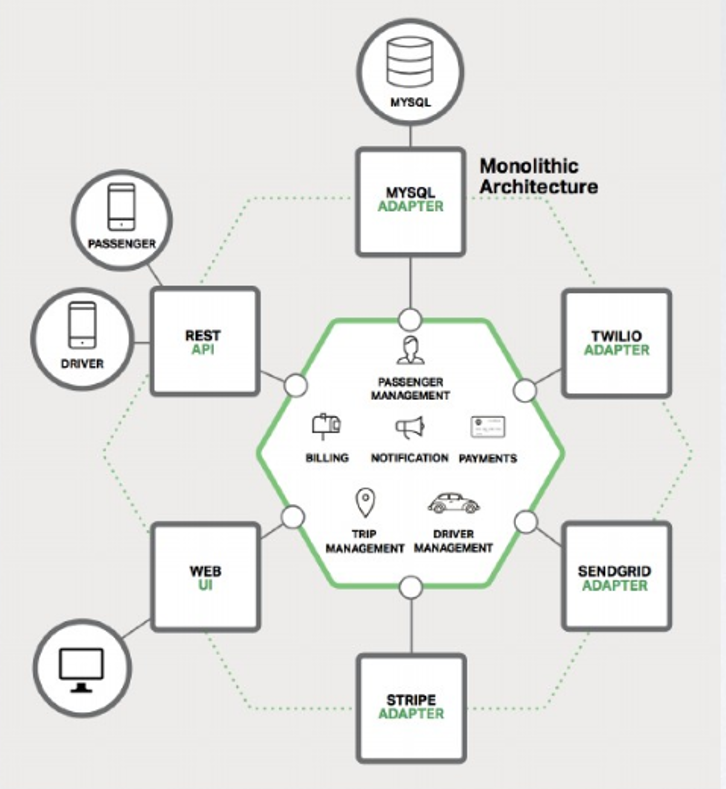
高度自治：每一个服务的状态单独可控，可以独立运行

轻量通信协议： 通过http协议进行服务之间的调用，mq中间件，fegin和cxf等都是基于http协议的

可持续部署：可以频繁部署，基础是可以自动化，一键部署

分布式：每一个模块（功能）作为一个单独的系统，便于提高这个模块（功能）高可用行 ，当这个模块（功能）遇到性能瓶颈时，能够更加简单的进行优化和进行集群部署

单体架构：巨石型



单体架构：

1高度集成，过于复杂

2打包部署周期长

3不能使用自动化部署测试等

4数据库不好拆分

微服务架构：

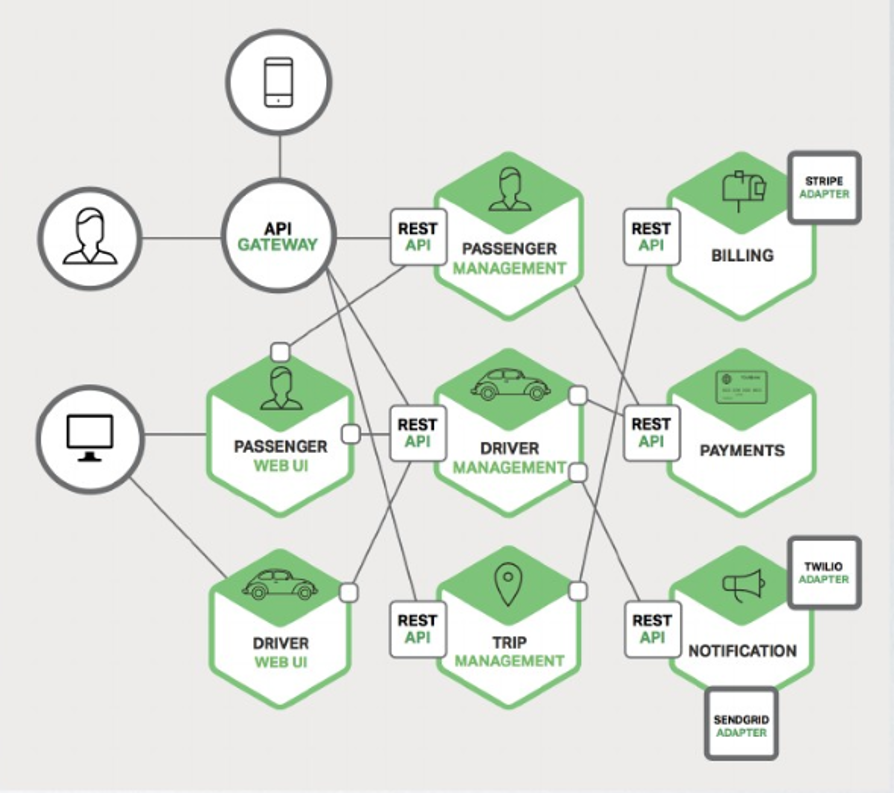
1.应用开发简单

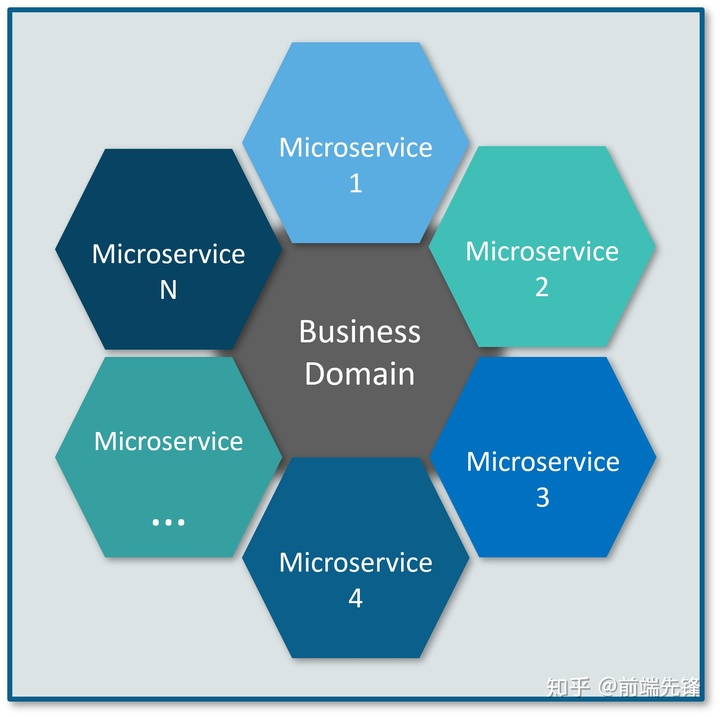
2.易于对应用程序进行大规模的更改

3.测试相对简单直观

4.部署简单明了

5.横向扩展容易





# **dubbo与cloud的区别**

dubbo是基于rpc协议，基于接口的远程调用；不能跨平台

Cloud是http协议的，restful风格的，可以实现跨平台调用

rpc协议是基于更底层的TCP协议，数据不需要通过http协议包装，实践性能更好。

****2.使用方式****

dubbo一般是xml配置的方式，cloud是boot基于注解的

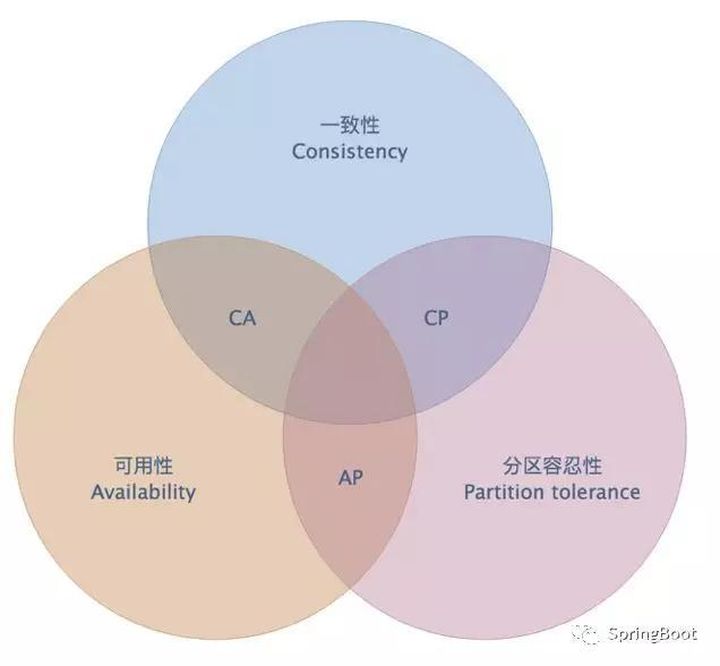
****3.注册发现****

dubbo使用的是zookeeper，在分布式系统中，zookeeper更加关注一致性，和容错性；当zookeeper主节点挂掉会有一个30~120s的选举时间，期间服务是不可用的。（牺牲高可用）

cloud的注册中心使用的是EureKa,更加关注的是可用性，和分区容错，多个eureka不存在单点问题（牺牲数据一致性，某个服务不可用，可能导致数据丢失）；

微服务架构cpa理论：

技术选型：



1.2 CP结合

CP结合就是保证一致性和分区容忍性，放弃可用性。Zookerper就是追求强一致性，放弃了可用性，还有跨行转账，一次转账请求要等待双方银行系统都完成整个事务才能完成。(当有master节点宕机，leader选举 中间30-120秒不可用状态)

1.3 AP结合

AP结合就是保证可用性和分区容忍性，放弃一致性。这是分布式系统设计时的选择 Eureka（

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)）

所以一般Eureka客户端也就是 微服务的任意一个server 会向多个Eureka服务端进行注册

# 微服务、Springcloud 和springboot的 三者之间的关系

springboot的特点是专注于快速方便的开发单个个体服务 ，为开发者减去了很多繁琐的配置，springboot也可以单独使用

springcloud是关注全局的微服务协调整理治理框架，包将boot的单体微服务整合并管理起来，含了一堆基于springboot的技术框架，所以springcloud依赖于springboot

微服务的五大组件

Eureka、feign、ribbon、zuu(gateway) 、config

先认识当个组件， 在具体的项目中 了解具体模块怎么开发

# Eureka：

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

# Register(服务注册)：把自己的IP和端口注册给Eureka。 Renew(服务续约)：发送心跳包，每30秒发送一次。告诉Eureka自己还活着。 Cancel(服务下线)：当provider关闭时会向Eureka发送消息，把自己从服务列表中删除。防止consumer调用到不存在的服务。 Get Registry(获取服务注册列表)：获取其他服务列表。 Replicate(集群中数据同步)：eureka集群中的数据复制与同步。 Make Remote Call(远程调用)：完成服务的远程调用。

# Ribbon：

负载均衡是指将负载分摊到多个执行单元上，

Ribbon的 实现逻辑就是 ：将负载均衡逻辑以代码的形式封装到服务消费者的客户端上（核心是通过loadBalanceder 决定将调用哪个服务的提供者），服务消费者客户端（通过Eureka ）维护了一份服务提供者的信息列表，有了信息列表，通过负载均衡策略将请求分摊给多个服务提供者，从而达到负载均衡的目的。

# Hystrix:

在分布式系统中，服务与服务之间的依赖错综复杂， 一种不可避免的情况就是某些服务会出现故障，导致依赖于它们的其他服务出现远程调度的线程阻塞。Hystrix提供了熔断器功能，能够阻止分布式系统中出现联动故障。Hystrix 是通过隔离服务的访问点阻止联动故障的，并提供了故障的解决方案，从而提高了整个分布式系统的弹性。

Hystrix的设计原则：

* + - 防止单个服务的故障耗尽整个服务的Servlet 容器（例如Tomcat ）的线程资源。
    - 快速失败机制，如果某个服务出现了故障，则调用该服务的请求快速失败，而不是线程等待。
    - 提供回退（ fallback ）方案，在请求发生故障时，提供设定好的回退方案。
    - 使用熔断机制，防止故障扩散到其他服务。
    - 提供熔断器的监控组件Hystrix Dashboard，可以实时监控熔断器的状态。

Hystrix的工作机制：

首先，当服务的某个API 接口的失败次数在一定时间内小于设定的阀值时，熔断器处于关闭状态，该API 接口正常提供服务。当该API 接口处理请求的失败次数大于设定的阀值时，

Hystrix 判定该API 接口出现了故障，打开熔断器，这时请求该API 接口会执行快速失败的逻辑（即fallback 回退的逻辑），不执行业务逻辑，请求的线程不会处于阻塞状态。

处于打开状态的熔断器， 一段时间后会处于半打开状态，并将一定数量的请求执行正常逻辑。剩余的请求会执行快速失败，若执行正常逻辑的请求失败了，则熔断器继续打开；

若成功了，则将熔断器关闭。这样熔断器就具有了自我修复的能力。

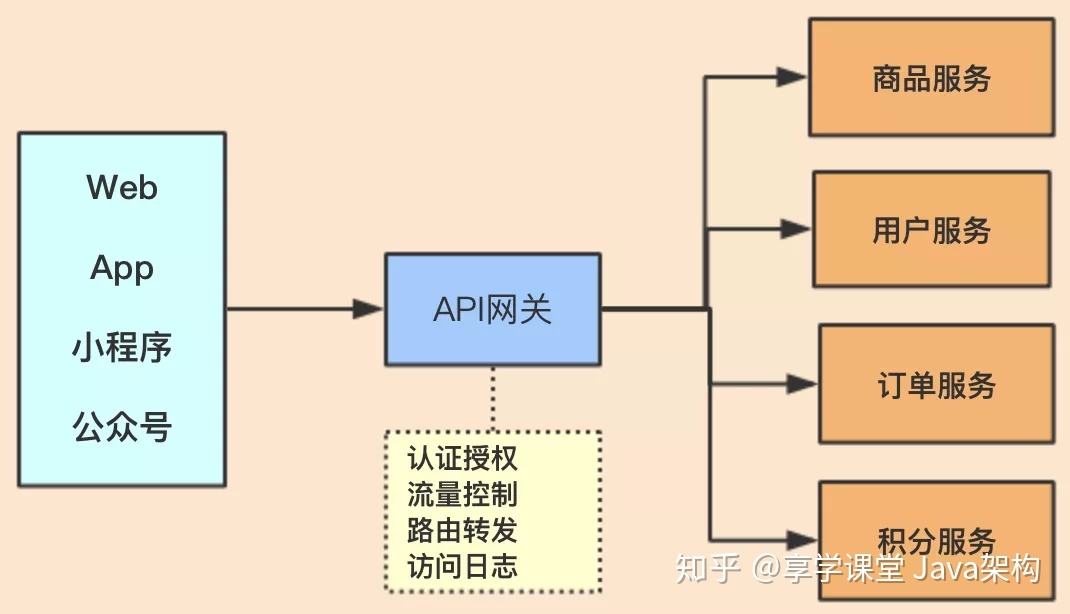
# Config:

分为config服务端和config客户端，服务端 提供配置文件（可以存在于本地项目也可以存在于git仓库）在搭配自动化部署工具（比如jenkins）和 Spring Cloud Bus，mqd等，实现 代码提交之后，或者修改配置文件之后 自动通知客户端，进行重启，或者热部署 实现自动化部署的效果

# Zuul：由于使用的是阻塞式的 API，不支持长连接所使用gateway 替换

# Gateway:

网关的作用:

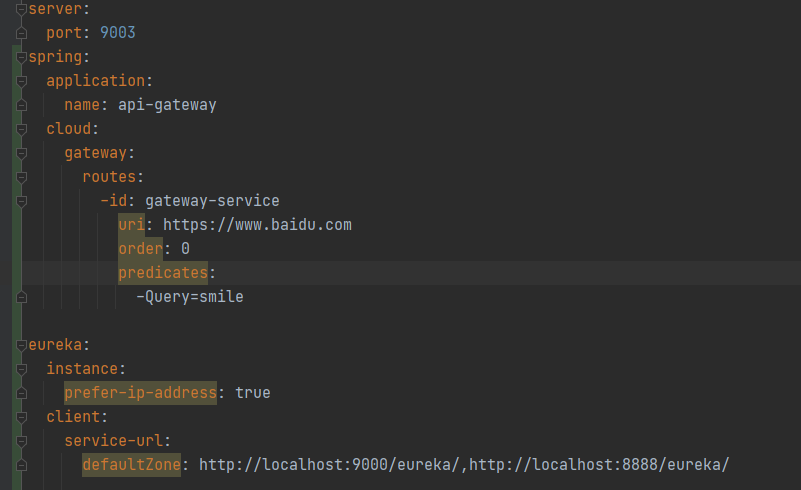


gateway对比zuul多依赖了spring-webflux，在spring的支持下，功能更强大，内部实现了限流、负载均衡等，扩展性也更强，但同时也限制了仅适合于Spring Cloud套件

（1）基于 Spring Framework 5，Project Reactor 和 Spring Boot 2.0

（2）集成 Hystrix 断路器

（3）Predicates 和 Filters 作用于特定路由，易于编写的 Predicates 和 Filters

* 这是一个 Java 8 的 Predicate，可以使用它来匹配来自 HTTP 请求的任何内容，例如 headers 或参数。**断言的**输入类型是一个 ServerWebExchange(Predicate 就是为了实现一组匹配规则，方便让请求过来找到对应的 Route 进行处理)。
* 
* 
* 和Zuul的过滤器在概念上类似，可以使用它拦截和修改请求，并且对上游的响应，进行二次处理。过滤器为org.springframework.cloud.gateway.filter.GatewayFilter类的实例。

1. 具备一些网关的高级功能：动态路由、限流、路径重写

* 网关配置的基本组成模块，和Zuul的路由配置模块类似。一个**Route模块**由一个 ID，一个目标 URI，一组断言和一组过滤器定义。如果断言为真，则路由匹配，目标URI会被访问。

# Feign：

feign 是一个伪Java Http 客户端， Feign 不做任何的请求处理。Feign 通过处理注解生成Request 模板，从而简化了Http API 的开发。

开发人员可以使用注解的方式定制Request API模板。在发送HttpRequest 请求之前， Feign 通过处理注解的方式替换掉Request 模板中的参数，生成真正的Request ，并交给Java Http 客户端去处理。利用这种方式，开发者只需要关注Feign注解模板的开发，而不用关注Http 请求本身，

简化了Http 请求的过程，使得Http请求变得简单和容易理解。

远端调用有webService 、restTemplate、httpClient都可以实现，Feign的声明式调用一个是写法上 更加统一 一致，可以轻松的结合ribbon和hystrix 实现负载均衡和服务降级

总结：

对于后期我们自己对指挥平台要二次开发：

微服务拆分的目的是 专注于业务逻辑，当我们掌握了，各个服务之间的调用和通信方式，剩下的就是业务逻辑了，业务逻辑，先设计好模块和功能等，剩下的就是编码。

后续我们的项目如何整合到springcloud。 首先保证后续开发的功能正常（功能拆分比的合理性较重要），然后配置到位，然后是静态资源到 我们网关的接口规则的统一（后台接口），然后当我们 了解比如说 路由网关规则，和远程调用规则，有了统一的接口，后续的整合 并不难

微服务平台的建设：

后续服务状态的监控、治理和日志收集分析 ，以及自动化配置，构建等 一系列的基础服务需要先搭建起来。

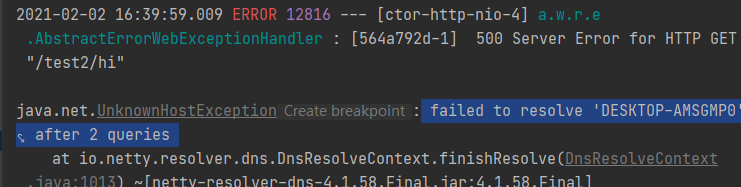
### **踩坑指南**

### **spring cloud config 不读取 bootstrap.properties/yml**



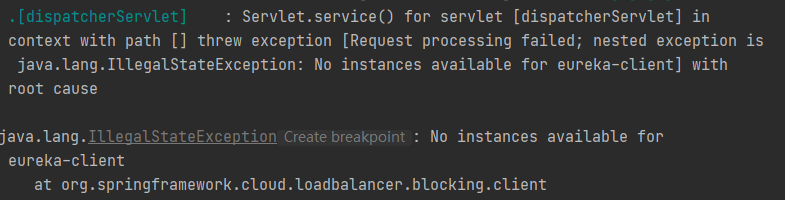
# **总结 还是那句话springboot cloud中 大部分问题 都是版本问题**

**GateWay 请求不到主机地址**

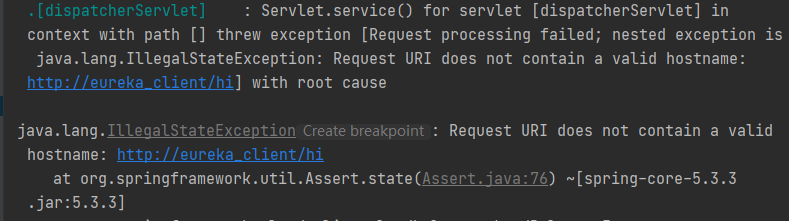
****

**解决办法 ： 目标server开启ip注册**

**eureka.instance.prefer-ip-address= true**

****

**Application.name 这个是首字母大写，按照驼峰命名写法 造成的**

****

**这个是按照规则了 但是 用 \_ 替换了 - 造成的**

**总结Application.name 命名 一定要规范 所有字母小写，单词之间 英文 小 - 隔开 不然的话 明明写的一模一样就是调用不到服务，这个是通过serviceId 调用的服务的**