# 微服务

CAP原则又称CAP定理，指的是在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance）

## 1.1 服务治理

**Consul强一致性(C)带来的是：**

1. 服务注册相比Eureka会稍慢一些。因为Consul的raft协议要求必须过半数的节点都写入成功才认为注册成功.
2. Leader挂掉时，重新选举期间整个consul不可用。保证了强一致性但牺牲了可用性。

**Eureka保证高可用(A)和最终一致性：**

1. 服务注册相对要快，因为不需要等注册信息replicate到其他节点，也不保证注册信息是否replicate成功
2. 当数据出现不一致时，虽然A, B上的注册信息不完全相同，但每个Eureka节点依然能够正常对外提供服务，这会出现查询服务信息时如果请求A查不到，但请求B就能查到。如此保证了可用性但牺牲了一致性。

其他方面，eureka就是个servlet程序，跑在servlet容器中; Consul则是go编写而成。

### 1.1.1 Spring Cloud Eureka

1.1.1.1 概述

Spring Cloud Eureka是Spring Cloud Netflix项目下的服务治理模块。而Spring Cloud Netflix项目是Spring Cloud的子项目之一，主要内容是对Netflix公司一系列开源产品的包装，它为Spring Boot应用提供了自配置的Netflix OSS整合。通过一些简单的注解，开发者就可以快速的在应用中配置一下常用模块并构建庞大的分布式系统。它主要提供的模块包括：服务发现（Eureka），断路器（Hystrix），智能路由（Zuul），客户端负载均衡（Ribbon）等。

1.1.1.2 重要注解

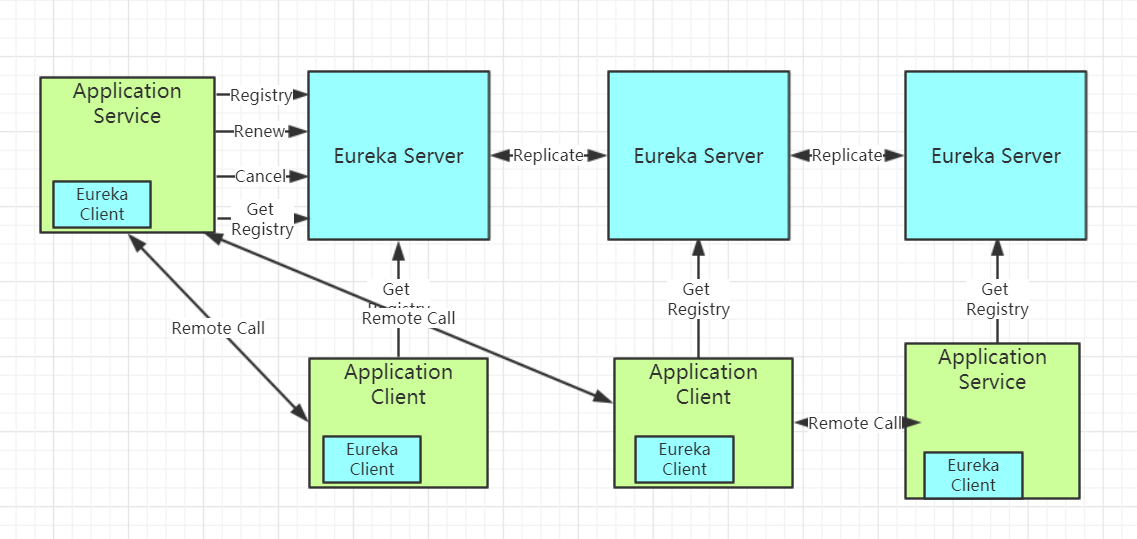
@EnableEurekaServer // 注册服务中心

@EnableEurekaClient // 服务提供端

1.1.1.3 服务治理机制

Eureka Client：一个Java客户端，用于简化与 Eureka Server 的交互（通常就是微服务中的客户端和服务端）

Eureka Server：提供服务注册和发现的能力（通常就是微服务中的注册中心）



**服务注册与续约：**首先服务注册到eureka server上，注册之后服务提供者会维护一个心跳，用来告诉eureka server “我还OK”，也就是续约Renew，心跳默认三十秒检测一次。

**服务剔除：**当服务实例正常下线时，服务实例会告诉eureka server “我要下线了”，将服务状态改为 down。有时服务会出现非正常下线，eureka server 在启动时会去创建一个定时任务检测服务提供者，剔除服务，检测时间默认90秒。

**自我保护：**eureka server 会去将心跳在15分钟内是否低于85%，如果出现eureka会将他们保护起来，让这些实例不会过期，这样客户端就很容易拿到不存在的实例，这就需要容错机制，客户端重试、断路器等机制。

**总结：**

Eureka Client负责：

服务注册、服务续约、服务正常下线、查询eureka server中服务实例列表。

Eureka Server 负责：

服务非正常下线剔除、服务保护。

1.1.1.4 Zone

Eureka 有一个 Region 和 Zone 的概念，你可以理解为现实中的大区（Region）和机房（Zone）

Eureka Client 在启动时需要指定 Zone，它会优先请求自己 Zone 的 Eureka Server 获取注册列表同样的，Eureka Server 在启动时也需要指定 Zone，如果没有指定的话，其会默认使用 defaultZone.

#### 1.1.1.5 服务配置

**微服务client，配置文件：**

##eureka

eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka/

eureka.instance.instance-id=${spring.application.name}:${server.port}

# 设置微服务调用地址为IP优先（缺省为false）

eureka.instance.prefer-ip-address=true

# 心跳时间，即服务续约间隔时间（缺省为30s）

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

# 发呆时间，即服务续约到期时间（缺省为90s）

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

**注册中心配置文件：**

## server

server.port=8081

##eureka

#指定环境

eureka.environment=work

# 设置是否将自己作为客户端注册到注册中心（缺省true）

# 这里为不需要（查看@EnableEurekaServer注解的源码，会发现它间接用到了@EnableDiscoveryClient）

eureka.client.register-with-eureka=false

# 设置是否从注册中心获取注册信息（缺省true）

# 因为这是一个单点的EurekaServer，不需要同步其它EurekaServer节点的数据，故设为false

eureka.client.fetch-registry=false

eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka/

#是否开启自我保护模式，默认为true。

eureka.server.enable-self-preservation=true

#续期时间，即扫描失效服务的间隔时间（缺省为60\*1000ms）

eureka.server.eviction-interval-timer-in-ms=10000 # 此处不开启缓存，上方配置开启一个即可

# use-read-only-response-cache: false

# 指定每分钟需要收到的续约次数的阈值，默认值就是：0.85

renewal-percent-threshold: 0.85

# 续约频率提高，默认：30

leaseRenewalIntervalInseconds: 10

### 1.1.2 Consul

#### 1.1.2.1 概述

Consul是基于GO语言开发的开源工具，主要面向分布式，服务化的系统提供服务注册、服务发现和配置管理的功能。Consul的功能都很实用，其中包括：服务注册/发现、健康检查、Key/Value存储、多数据中心和分布式一致性保证等特性。Consul本身只是一个二进制的可执行文件，所以安装和部署都非常简单，只需要从官网下载后，在执行对应的启动脚本即可。

#### 1.1.2.2 Consul特性

**基础特性：**

**1.服务注册/发现**  
为什么微服务架构下就需要做服务注册和服务发现呢？微服务的目标就是要将原来大一统的系统架构，拆分成细粒度的按功能职责分成的小系统，这样就会出现很多小的系统，部署的节点也会随之增加。试想一下，如果没有一个统一的服务组件来管理各系统间的列表，微服务架构是很难落地实现的。  
Consul提供的服务注册/发现功能在数据强一致性和分区容错性上都有非常好的保证，但在集群可用性下就会稍微差一些（相比Euerka来说）。

**2.数据强一致性保证**  
Consul采用了一致性算法Raft来保证服务列表数据在数据中心中各Server下的强一致性，这样能保证同一个数据中心下不管某一台Server Down了，请求从其他Server中同样也能获取的最新的服务列表数据。数据强一致性带来的副作用是当数据在同步或者Server在选举Leader过程中，会出现集群不可用。

**3.多数据中心**  
Consul支持多数据中心(Data Center),多个数据中心之间通过Gossip协议进行数据同步。多数据中心的好处是当某个数据中心出现故障时，其他数据中心可以继续提供服务，提升了可用性。

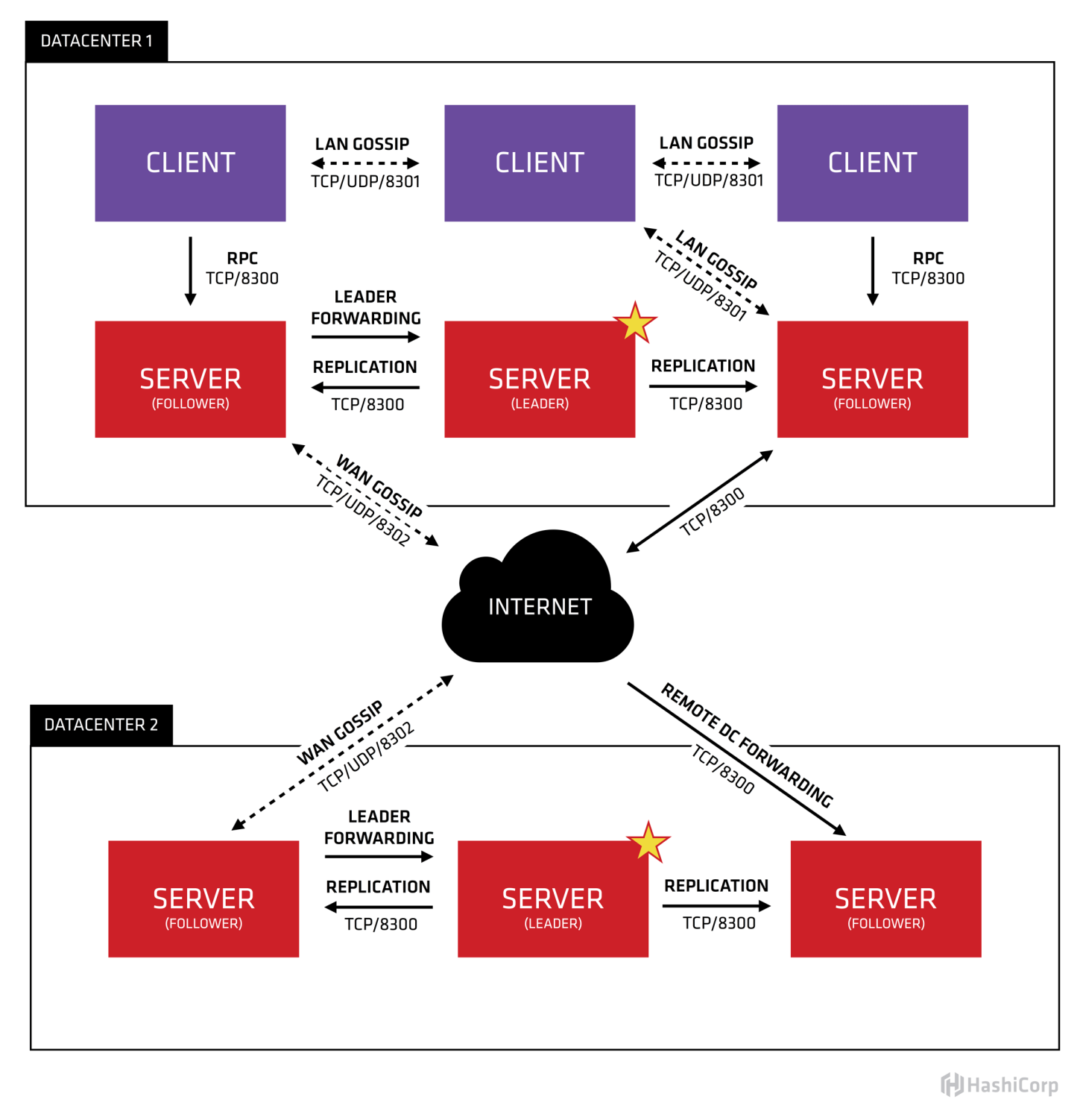
**4.健康检查**  
Consul支持基本硬件资源方面的检查，如：CPU、内存、硬盘等

**5.Key/Value存储**  
Consul支持Key/Value存储功能，可以将Consul作为配置中心使用，可以将一些公共配置信息配置到Consul，然后通过Consul提供的 HTTP API来获取对应Key的Value。

**高级特性：**

**1.HTTP API**  
**2.ACL**

#### 1.1.2.3 Consul工作模式



从上图可以看到，Consul中包括的3种不同的角色：Client、Server、Server-Leader。还有一个在图上没有标出来的角色Agent，一共4个角色。

· Agent  
1.是一个守护线程  
2.跟随Consul应用启动而启动  
3.负责检查、维护节点同步

· Client  
1.转发所有请求给Server  
2.无状态，不持久化数据  
3.参与LAN Gossip的健康检查

· Server  
1.持久化数据  
2.转发请求给Server-Leader  
3.参与Server-Leader选举  
4.通过WAN Gossip，与其他数据中心交换数据

· Server-Leader  
1.响应RPC请求  
2.服务列表数据同步给Server

#### 1.1.2.4 Consul工作原理

##### **服务注册的方式**

Consul服务注册有两种方式：HTTP API & JSON配置文件  
方式一：HTTP API

http://{ip}:8500/v1/agent/service/register/:service

方式二：JSON 配置文件



启动Consul增加启动参数-config-dir

nohup ./consul agent -dev -config-dir=/consul-conf/service.json &

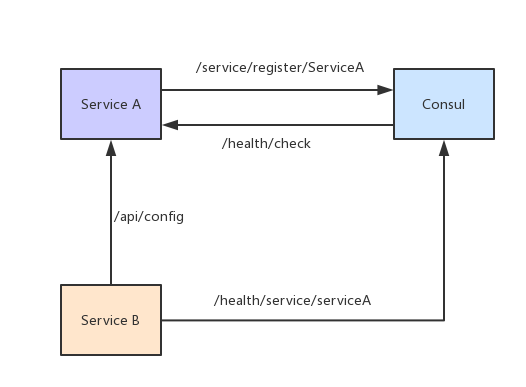
##### **服务发现的方式**

服务发现的方式同时有两种：HTTP API & DNS Agent  
方式一：HTTP API

获取某个service下健康的服务列表信息  
<http://{ip}:8500/v1/health/service/:service>

方式二：DNS Agent

##### **工作流程**



## 1.2 幂等设计

幂等定义：

系统中的多次操作，不管多少次，都应该产生一样的效果，或返回一样的效果。

比如实际的业务请求为创建一个活动，理论上需要根据业务形态开发幂等创建活动的接口，这样在相同参数调用接口多次创建活动时，只可以创建成功一次。

由于查询天生的是幂等请求，所以针对于查询场景可以不做业务角度的幂等约束，查询幂等的约束多是针对于资源控制，安全防刷，流控来做的。

#### 保证幂等策略

幂等需要通过**唯一的业务单号**来保证。也就是说相同的业务单号，认为是同一笔业务。使用这个唯一的业务单号来确保，后面多次的相同的业务单号的处理逻辑和执行效果是一致的。 下面以支付为例，在不考虑并发的情况下，实现幂等很简单：①先查询一下订单是否已经支付过，②如果已经支付过，则返回支付成功；如果没有支付，进行支付流程，修改订单状态为‘已支付’。

#### 防重复提交策略

上述的保证幂等方案是分成两步的，第②步依赖第①步的查询结果，无法保证原子性的。在高并发下就会出现下面的情况：第二次请求在第一次请求第②步订单状态还没有修改为‘已支付状态’的情况下到来。既然得出了这个结论，余下的问题也就变得简单：把查询和变更状态操作加锁，将并行操作改为串行操作。

##### 乐观锁

如果只是更新**已有**的数据，没有必要对业务进行加锁，设计表结构时使用乐观锁，一般通过version来做乐观锁，这样既能保证执行效率，又能保证幂等。例如： UPDATE tab1 SET col1=1,version=version+1 WHERE version=#version# 不过，乐观锁存在失效的情况，就是常说的ABA问题，不过如果version版本一直是自增的就不会出现ABA的情况。（从网上找了一张图片很能说明乐观锁，引用过来，出自[Mybatis对乐观锁的支持](http://www.voidcn.com/blog/liyantianmin/article/p-5038695.html" \t "https://www.cnblogs.com/firstdream/p/_blank)）

##### 防重表

使用订单号orderNo做为去重表的唯一索引，每次请求都根据订单号向去重表中插入一条数据。第一次请求查询订单支付状态，当然订单没有支付，进行支付操作，无论成功与否，执行完后更新订单状态为成功或失败，删除去重表中的数据。后续的订单因为表中唯一索引而插入失败，则返回操作失败，直到第一次的请求完成（成功或失败）。**可以看出防重表作用是加锁的功能。**

##### 分布式锁

这里使用的防重表可以使用分布式锁代替，比如Redis。订单发起支付请求，支付系统会去Redis缓存中查询是否存在该订单号的Key，如果不存在，则向Redis增加Key为订单号。查询订单支付已经支付，如果没有则进行支付，支付完成后删除该订单号的Key。通过Redis做到了分布式锁，只有这次订单订单支付请求完成，下次请求才能进来。相比去重表，将放并发做到了缓存中，较为高效。思路相同，**同一时间只能完成一次支付请求**。

##### token令牌

这种方式分成两个阶段：申请token阶段和支付阶段。 第一阶段，在进入到提交订单页面之前，需要订单系统根据用户信息向支付系统发起一次申请token的请求，支付系统将token保存到Redis缓存中，为第二阶段支付使用。 第二阶段，订单系统拿着申请到的token发起支付请求，支付系统会检查Redis中是否存在该token，如果存在，表示第一次发起支付请求，删除缓存中token后开始支付逻辑处理；如果缓存中不存在，表示非法请求。 实际上这里的token是一个信物，支付系统根据token确认，你是你妈的孩子。不足是需要系统间交互两次，流程较上述方法复杂。

##### 支付缓冲区

把订单的支付请求都快速地接下来，一个快速接单的缓冲管道。后续使用异步任务处理管道中的数据，过滤掉重复的待支付订单。优点是同步转异步，高吞吐。不足是不能及时地返回支付结果，需要后续监听支付结果的异步返回。

# Java基础