概念入门：

CAP理论,

C 数据的一致性 不是强一致

A 一定时间内一定有数据返回 error 提示信息

P 服务容错性 一个系统中，我们的服务允许某一块出错。不影响系统整体的运行。

Dubbo –cp

Netflix – AP

微服务:

1. 什么是微服务:

小，专注于做好一件事。

完成微服务，我们需要做到：

1. 高度的自治性.独立的开发，部署，发布
2. 需要异构性。在协作的过程中，我们需要去思考不同服务最适合的技术（方案）。
3. 弹性。容错性
4. 扩展。
5. 简化部署。
6. 可组合性。
7. 监控和日志。定位问题。
8. 微服务不是万金油

Spring cloud

官网：<https://spring.io/projects/spring-cloud>

<https://springcloud.cc/>

项目创建地址:<https://start.spring.io/>

\*注册中心（非常重要）:

1.什么是注册中心

注册中心可以说是微服务架构中的”通讯录“，它记录了服务和服务地址的映射关系。在分布式架构中，服务会注册到这里，当服务需要调用其它服务时，就到这里找到服务的地址，进行调用。

1. 为什么需要注册中心

在分布式系统中，我们不仅仅是需要在注册中心找到服务和服务地址的映射关系这么简单，我们还需要考虑更多更复杂的问题：

服务注册后，如何被及时发现？

服务宕机后，如何及时下线？

服务如何有效的水平扩展？

服务发现时，如何进行路由？

服务异常时，如何进行降级？

注册中心如何实现自身的高可用？

这里问题的解决都依赖于注册中心。简单看，注册中心的功能有点类似于DNS服务器或者负载均衡器，而实际上，注册中心作为微服务的基础组件，可能要更加复杂，也需要更多的灵活性和时效性。

在spring cloud中注册中心有eureka，consul，zookeeper，nacos等.

我们这堂课就是以eureka来了解和认知注册中心的:

假设现在我们去设计一个eureka，我们要怎么去分析：

1:注册流程

自动装配:spring-cloud-netflix-eureka-server.jar META-INFO/Spring.factors **EurekaServerAutoConfiguration**

spring-cloud-netflix-eureka-client.jar META-INFO/Spring.factors **EurekaClientAutoConfiguration EurekaDiscoveryClientConfiguration**

**分析：注册流程，我们要怎么去做?**

1. **加载文件，并获取配置文件信息.我们有三份配置文件**

Server配置信息 server 信息: EurekaServerConfigBean#EurekaServerConfig

Client配置信息 instance信息: EurekaClientAutoConfiguration#EurekaApplicationInfo()#create()\*在这个方法里传入了一个接口，它的实现类: EurekaInstanceConfigBean

Client配置信息 client: EurekaClientAutoConfiguration# eurekaClientConfigBean

New 一个eurekaClientConfigBean 这里定义了client的配置

1. 实例化一个client端这样我才能注册
2. 注册前，我要确定这个client是否先被注册了，不能重复注册。

EurekaClientAutoConfiguration#eurekaClient()\*实例化客户端client跟踪

DiscoveryClient 找定时任务，然后向下看，会有一句提示:启动所有定时任务，跟踪进去，找到heartbeat—然后会看到renew，renew规则：如果当前实例是404状态，就去注册，如果不是，返回true。

2：注册中心，怎么接收请求

注册中心的请求:

1. 心跳请求(刷新，注册，心跳)
2. 存储服务实例

3:服务实例怎么存储。

courrentHashmap中

4:服务中心是怎么实现的高可用？

高可用这一块我们先去理解一下：**对等**. P2P 对等实例来实现的高可用 (peer to peer)

老规矩，我们继续去寻找:EurekaServerAutoConfiguration下去寻找我们要的东西.

我们会看到eurekaServerContext() 这个方法.通过方法名，我们猜都能猜出来这个就是初始化上下文容器。

那么我们来看一下它在初始化干了什么？它new 了一个对象 进入这个对象继续深入:

我们可以在这里看到，这个类初始化的时候会加载一个对象peerEurekaNodes.start()

位置DefaultEurekaServerContext#peerEurekaNodes 那么我们进入它的start里面去继续看，

我们会发现它实际上是调用了updatePeerEurekaNodes(resolvePeerUrls) 这个resolvePeerUrls我们可以看到，它实际上就是我们通过实例获取到的zone，也就是我们自己定义的集群地址的集合.

回头继续看updatePeerEurekaNodes()方法, 里面会调用一个共有变量newPeerUrls --伙伴节点地址。它会在这里定义两个Set，一个是toAdd()，一个是toshutdown()，这里面我们可以看，它是去维护了新加入的实例，以及需要移除的实例，然后根据一系列的操作判断，把已经终止的实例移除，再把新加入的实例写入。

那么我们也许会有疑问，如果说我们在运行期某个节点出现了故障，我们该怎么办？我们会发现PeerEurekaNodes#start()方法下做了一个定时任务，这个定时任务，会不断的维护我们的eureka列表。

5：服务中心怎么同步数据？怎么防止循环传播？

同步数据，我觉得我们就需要来看一下它是怎么维护实例的。跟进一下

PeerAwareInstanceRegistryImpl这个类我们之前看到它怎么注册的，我们可以继续跟进一下它的注册流程.

进入到PeerAwareInstanceRegistryImpl之后，我们可以看到，它在注册结束后，还调用了replicateToPeers 复制数据给其他节点，我们进入这个方法去看一下

我们看到了非常熟悉的peerEurekaNodes,它维护了我们集群下的服务节点。所以我们的同步方法就是在遍历我们的集群节点，然后将相应的数据进行同步.(说白了，就是把相应实例再一次注册给不同的服务端)

这样做有没有问题？我们去思考一下，其实是有的，我们把数据给到了其他节点，那么其他节点能不能接收到我们当前的节点是新加的，还是复制过来的呢？我们继续看代码--它还定义了一个isReplication作为标志，如果是复制过来的实例，就让它为true，这就表示了它是一个复制过来的数据，防止出现死循环。

6：erueka客户端关闭，eureka的下架

1：cancelScheduledTasks();关停线程池

2:判断我们的当前客户端确实注册在了eureka上面，就去把它的当前状态改为DOWN。

3：调用unregister();然后去调用了cancel()方法将服务删除。

7：服务异常下剔除。

进入initEurekaServerContext会看到 registry.openForTraffic(applicationInfoManager, registryCount);这段代码引用了openForTraffic()方法

进入方法，我们继续去追发现它是一个接口，我们去查看实现类会找到InstanceRegistry下的实现，然后发现它实际上是调用了super()方法，所以我们去找到它的父类，看看这里是怎么实现的。

1.updateRenewsPerMinThreshold（）这里会去调用一个公式

当前发送过服务的数量X(60/发送心跳的间隔)X定义的阈值

然后他在这里对我们的instance做了一系列的处理，最后去调用了postInit（）方法

进入postinit方法，会发现evictionTaskRef.set(new EvictionTask())初始化了一个EvictionTask。

到这个EvictionTask里面去看，我们会看到EvictionTask的run方法中调用了一个evict（）方法。

这个evict方法就有说法了。这里就是在处理相关的服务的东西.首先evict方法会去判断一下isLeaseExpirationEnabled（） 也就是是否实现了自我保护机制？如果我们实现了自我保护机制，

可以去看一下，如果没用启用，它会直接返回一个true，如果启用了，这里需要满足以下条件，

服务实例个数大于0，每分钟的心跳需要大于我们的阈值，而我们的阈值就是

当前发送过服务的数量X(60/发送心跳的间隔)X定义的阈值，假设我们服务已经关停，那么这里的第二个条件一定不会满足，所以会返回一个false。所以这里就会直接return。不去剔除任何服务。所以，在自我保护机制开启下，eureka不会去主动剔除服务。

回到evict方法里，继续去看非自我保护下，我们会看到evict方法定义了一个集合。将已经没有发送心跳的实例写入到expiredLeases这个集合中去。

通过expiredLeases集合，调用internalCancel（）会开始去一个个去删除存在于这个集合中的客户端。

8：自我保护机制原理？我们在什么情况下去选择它？

我们继续进入PeerAwareInstanceRegistryImpl下的 isLeaseExpirationEnabled()我们可以看到，它去读取了一个配置项isSelfPreservationModeEnabled 这个就是我们的自我保护机制的开启,自我保护说的是什么意思？保护的是在节点上的服务，不是第一时间被剔除，默认是超过了90s,再去剔除服务。那么这样，会出现一个问题，当我们去访问时，它有可能拿到的是一个不正常的节点，从而调用失败，所以我们才需要降级和熔断机制。

**this**.**numberOfRenewsPerMinThreshold** = (**int**) (**this**.**expectedNumberOfClientsSendingRenews** \* (60.0 / **serverConfig**.getExpectedClientRenewalIntervalSeconds())  
 \* **serverConfig**.getRenewalPercentThreshold());

1min =60s

1 \* (60/30).\*0.85 = 1.7