# **1.[面向过程](https://so.csdn.net/so/search?q=%E9%9D%A2%E5%90%91%E8%BF%87%E7%A8%8B&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42025798/article/details/_blank) & 面向对象**

**面向过程思想：  
  步骤清晰简单、第一步做什么、第二步做什么  
  面向过程适合处理一些较为简单的问题**

**[面向对象](https://so.csdn.net/so/search?q=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42025798/article/details/_blank)思想  
  物以类聚，分类的思维模式，思考问题首先会解决问题需要哪些分类，然后对这些分类进行单独思考。最后，才对某个分类下的细节进行面向过程的思索。**

**什么是面向对象？  
  面向对象编程（OOP）  
  面向对象编程的本质就是：以类的方式组织代码，以对象的方式组织（封装）数据；**

**三大特性：  
  1.封装  
  2.继承  
  3.[多态](https://so.csdn.net/so/search?q=%E5%A4%9A%E6%80%81&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42025798/article/details/_blank)**

**从认识论角度考虑是先有对象后有类。对象，是具体的事物。类是抽象的，是对对象的抽象  
  从代码运行角度考虑是先有类后有对象。类是对象的模板。**

# **2.[构造方法](https://so.csdn.net/so/search?q=%E6%9E%84%E9%80%A0%E6%96%B9%E6%B3%95&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42025798/article/details/_blank)**

**使用new 关键字创建的时候，除了分配内存空间之外，还会给创建好的对象进行默认的初始化以及对类中构造器的调用。**

**构造器的特点  
  1.必须和类的名字相同  
  2.必须没有返回值类型的声明，也不能写void  
  3.一旦定义了有参构造，无参构造必须显示定义**

**构造器作用：  
  1.使用new 关键字必须要有构造器  
  2.用来初始化值**

# **3.封装**

**该露的露，该藏的藏  
  程序设计要追求“高内聚、低耦合”。高内聚就是类的内部数据操作细节自己完成，不允许外部干涉；低耦合：仅暴露少量的方法给外部使用。**

# **4.继承**

**继承的本质是对某一批类的抽象，从而实现对现实世界更好的建模。  
extends 的意思是 扩展，子类是父类的扩展。  
Java中 类只有单继承，没有多继承！（一个儿子只能有一个爸爸，但是一个爸爸可以有多个儿子）**

**在Java中，所有的类都默认继承Object类，都默认直接或者间接继承Object类**

**super注意点：  
  1.super调用父类的构造方法，必须在构造方法中的第一行  
  2.super 必须只能出现在子类的方法或者构造方法中！  
  3.super 和 this 不能同时调用构造方法**

**super Vs this**

**代表的对象不同：  
  this：本身调用者这个对象  
  super：代表父类对象的应用  
  
  
前提  
  this：没有继承也可以使用  
  super：只能在继承条件才可以使用  
  
  
构造方法：  
  this(); 本类的构造  
  super(); 父类的构造！**

# **5.重写**

**需要有继承关系，子类重写父类的方法！  
子类的方法必须和父类必要一致，方法体不同**

**前提**

1. **方法名必须相同**
2. **参数列表必须相同**
3. **修饰符：范围可以扩大，单不能缩小： private（小） -》 public（大）**
4. **抛出的异常：范围可以被缩小，但不能扩大：  
   ClassNofoundException（小） --》 Exception（大）**

**静态方法和非静态方法区别很大！  
静态方法调用跟左边有关**

# **6.多态**

**super(); 即同一方法可以根据发送对象的不同而采用多种不同行为方式  
  super(); 一个对象的实际类型是固定的，可以指向的引用类型就不确定了**

**public static void main(String[] args) {**

**// 能调用的方法都是自己的和父类的**

**Student s1 = new Student();**

**//不能调用子类独有的方法**

**Person s2 = new Student();**

**Object s3 = new Student();}**

**子类重写父类的方法，执行子类的方法。  
编译看左边，运行看右边。  
多态是方法的多态，属性没有多态性。**

**多态的条件**

1. **继承关系**
2. **父类引用指向子类对象**
3. **方法重写**

**ClassCaseException 类型转换异常**

**以下情况方法不能重写  
  [static](https://so.csdn.net/so/search?q=static&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42025798/article/details/_blank) 方法：属于类，它不属于实例  
  final 常量 ：在常量池里面的  
  private ：父类私有的方法**

# **7.static 详解**

**静态方法、静态属性。**

**非静态方法可以调用静态方法，静态方法只能调用静态方法。**

**静态随着类的加载而加载。**

**静态代码块，随着类的加载而调用，永远只执行一次。**

**static{**

**}**

**匿名代码块，在构造器之前调用。每次创建对象的时候都会执行{**

**}**

# **8.抽象类**

**abstract 可以用来修饰方法也可以修饰类，如果修饰方法，那么该方法就是抽象方法。如果修饰类，那么该类就是抽象类。**

**抽象类中可以没有抽象方法，但是有抽象方法的类必须声明为抽象类。**

**抽象类，不能使用new 关键字创建对象，它是用来让子类继承的。  
抽象方法，只有方法的声明，没有方法的实现，它是用来让子类实现的。**

**子类继承抽象类，必须实现所有的抽象方法，否则该子类也要声明为抽象类。**

**抽象类中可以写普通方法  
抽象方法必须在抽象类中**

# **9.接口**

**普通类：只有具体实现  
抽象类：具体实现和规范（抽象方法）都有  
接口：只有规范！自己无法写方法~专业的约束！**

**接口就是规范，定义的就是一组规则，体现了显示世界中“如果你是。。。就必须能。。。”的思想。**

1. **如果你是天使，则必须能飞**
2. **如果你是汽车，则必须能跑**
3. **如果你是好人，就必须干掉坏人**
4. **如果你是坏人，就必须欺负好人**

**作用：  
  1.约束  
  2.定义一些方法，让不同的人实现  
  3.方法都是public abstract  
  4.常量都是 public static final  
  5.接口中没有构造方法（它都不是类）  
  6.可以实现多个接口  
  7.必须重写接口中的方法**

# **10.[内部类](https://so.csdn.net/so/search?q=%E5%86%85%E9%83%A8%E7%B1%BB&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_42025798/article/details/_blank)**

**内部类就是在一个类的内部定义一个类，比如，A类中定义一个B类，那么B类相对A类来说就称为内部类，而A类相对于B类来说就是外部类了。**

**内部类分类**

1. **成员内部类**
2. **静态内部类**
3. **局部内部类**
4. **匿名内部类**

**成员内部类**

**public class Outer {**

**private int id;**

**public void out(){**

**System.out.println("这是外部类方法");**

**}**

**class Inner{**

**public void in(){**

**System.out.println("这是内部类的方法");**

**}**

**}**

**}//测试**

**public static void main(String[] args) {**

**Outer outer = new Outer();**

**outer.out();**

**Outer.Inner inner = new Outer().new Inner();**

**inner.in();}**

**静态内部类**

**\*/public class Outer {**

**private int id;**

**public void out() {**

**System.out.println("这是外部类方法");**

**}**

**static class Inner {**

**public void in() {**

**System.out.println("这是内部类方法");**

**}**

**}**

**}//测试public static void main(String[] args) {**

**Outer outer = new Outer();**

**outer.out();**

**Outer.Inner inner = new Outer.Inner();**

**inner.in();}**

**局部内部类**

**public class Outer {**

**private int id;**

**public void method() {**

**class Inner {**

**public void in() {**

**System.out.println("这是局部内部类");**

**}**

**}**

**Inner inner = new Inner();**

**inner.in();**

**}}//测试public static void main(String[] args) {**

**Outer outer = new Outer();**

**outer.method();}**

**控制反转IoC（Inversion of Control），是一种设计思想，DI（依赖注入）是实现IoC的一种方法，也有人认为DI只是IoC的另一种说法。没有IoC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建转移给第三方，个人认为所谓控制反转就是：获得依赖对象的方式反转了。**

**采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的。**

**控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入（Dependency Injection，DI）**

**对象的创建权转移到了Spring框架，这种思想称为控制反转Inversion of Control**

**如果是框架帮你找这些依赖对象，按一定规则提供给你，称之为依赖注入**

**单例模式**

**只要是单例模式就一定要构造器（与类同名）私有**

**无继承的情况下的执行顺序：总结一下优先级：静态代码块 > 构造代码块 > 构造器**

**有继承的情况下的执行顺序：总结一下优先级：父类静态代码块 > 子类静态代码块 > 父类构造代码块 > 父类构造器 > 子类构造代码块 > 子类构造器**

**1、面试官：公司使用什么 RPC 框架？，可以介绍一下 RPC 的工作原理吗？说你最熟悉的一个。**

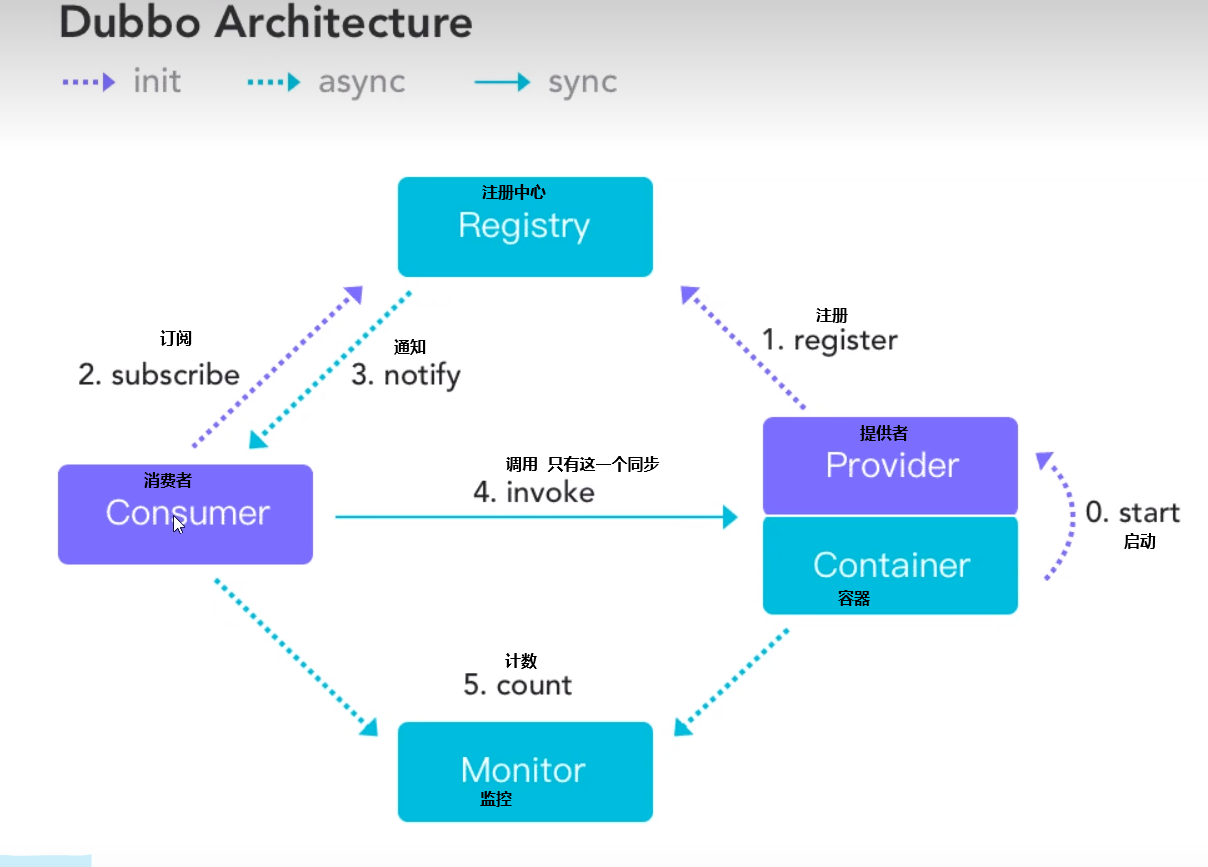
答：RPC 指远程过程调用，是一种进程间通信方式，他是一种技术的思想，而不是规范。它允许程序调用另一个地址空间(通常是共享网络的另一台机器上)的过程或函数，而不用程序员显式编码这个远程调用的细节。即程序员无论是调用本地的还是远程的函数，本质上编写的调用代码基本相同。

也就是说两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。为什么要用RPC呢?就是无法在一个进程内，甚至一个计算机内通过本地调用的方式完成的需求，比如不同的系统间的通讯，甚至不同的组织间的通讯，由于计算能力需要横向扩展，需要在多台机器组成的集群上部署应用。RPC就是要像调用本地的函数一样去调远程函数;

使用最广泛的 Spring Cloud，基于 Spring Boot 特性整合了开源行业中优秀的组件，整体对外提供了一套在微服务架构中服务治理的解决方案。

国内开源的框架中，使用比较广泛的有阿里的 Dubbo，后来捐献给了 Apache。还有腾讯的 Tars 框架，还有 Thrift 框架，也有基于 Thrift 二次开发的 RPC 框架，比如美团的 Mtthrift。

这些 RPC 大致原理基本都是一样的。（这个时候，跟面试官要纸和笔，画图解释 RPC 原理）



RPC两个核心模块:通讯，序列化。

**2、面试官：服务启动的时候服务基本信息被注册到注册中心，如果服务提供者挂了，注册中心如何知道服务不可用了呢？**

**答：服务掉线分为主动下线和心跳检测**

比如服务由于发版时，在重启之前先主动通知注册中心：我要重启了，有流量进来先不要分给我，让别的机器服务，等我重启成功后在放流量进来，或者是在管理后台手动直接摘掉机器，这个是主动下线。

心跳检测是处理服务非正常下线（如断电断网）的情况，这个时候如果注册中心不知道该服务已经掉线，一旦被其调用就会带来问题。为了避免出现这样的情况，注册中心增加一个心跳检测功能，它会对服务提供者（Provider）进行心跳检测，比如每隔 30s 发送一个心跳，如果三次心跳结果都没有返回值，就认为该服务已下线，赶紧更新 Consumer 的服务列表，告诉 Consumer 调用别的机器。

问题分析： 阐述了服务端挂了注册中心如何感知的问题，你以为此问题已经完事儿了？还没有，你成功给自己挖了个坑，面试官可能继续深挖，服务提供者（Provider）挂了注册中心能解决，那注册中心自己就不挂了吗？三连问继续。

**3、面试官：如果注册中心挂了，比如你用的是 Zookeeper，如果 Zookeeper 挂了，那服务之间还能相互调用吗？**

**答：首先注册中心挂掉也要分两种情况，如果数据库挂了，ZK 还是能用的，因为 ZK 会缓存注册机列表在缓存里**。

其次 ZK 本身就是一个集群的，一台机器挂了，ZK 会选举出集群中的其他机器作为 Master 继续提供服务，如果整个集群都挂了也没问题，因为调用者本地会缓存注册中心获取的服务列表。省略和注册中心的交互，Consumer 和 Provider 采用直连方式，这些策略都是可配置的。

4、面试官：你对 RPC 了解的很透彻，那你能否自己写一个 RPC 框架？可以简答描述下思路也行。

答：

客户端 invoke 方法编写，使用 JDK 的动态代理技术，客户端调用远程服务方法时调用的是 InvocationHandler 的 invoke 方法。

客户端 Filter 方法编写，完善的 RPC 框架少不了监控、路由、降级、鉴权等功能。

创建 Socket，在 Filter 方法中实现 Client.write 方法，其逻辑为从连接池（ChannelPool）中获取连接，然后将数据写进 Channel。

实现数据序列化、压缩，目的减少网络传输的数据量，向服务端发送 request 数据，这里可以使用 Netty 异步通讯框架。

服务端收到客户端发过的消息后，从 Channel 中将消息读出来之前，也会先经反序列化解压。

请求就到了服务端 Filter 中。请求依次经过监控、鉴权方法。

根据客户端传递来的服务信息和参数，通过反射调用相应的业务服务并拿到业务处理结果。然后在 ResponseFilter 中将返回结果写入 Channel。

服务端序列化、压缩等，发送给客户端。

客户端收到消息后，经过客户端反序列化、解压缩，后交给 ResponseThreadPoolProcessor 线程池处理。

ResponseThreadPoolProcessor 收到消息后，就将结果返回给之前的方法调用，整个调用请求就结束了。

深入分析

已经有 http 协议接口，或者说 RestFul 接口，为什么还要使用 RPC 技术？

**在接⼝不多的情况下**，使用 http 确实是一个明智的选择，比如在初创企业，我们不确定业务能顺利开展下去，可能面临随时倒闭，开发人员也不足，这个时候使用简洁高效的技术，先把东西做出来是最明智的选择，无需一步登天。

**系统与系统交互较少的情况下**，使用 http 协议优点显而易见：开发简单、测试也比较直接、部署方便，利用现成的 http 协议进行系统间通讯，如果业务真的慢慢做大，系统也慢慢扩大，RPC 框架的好处就显示出来 了，⾸先 RPC 支持长链接，通信不必每次都要像 http 一样去重复 3 次握⼿，减少了网络开销。

其次就是 RPC 框架一般都有注册中心模块，有完善的监控管理功能，服务注册发现、服务下线、服务动态扩展等都方便操作，服务化治理效率大大提高。

基于 TCP 协议实现的 RPC，能更灵活地对协议字段进行定制，相比 http 能减少网络传输字节数，降低网络开销（握手）提高性能。实现更大的吞吐量和并发数，但是需要更多的关注底层复杂的细节， 对开发人员的要求也高，增加开发成本。

RPC 工作原理总结：

Provider：服务提供方，CS 模型中的 Server。

Consumer： 调用远程服务服务消费方，CS 模型中的 Client。

Registry：服务注册与发现的服务管理中心。

Monitor：统计服务的调用次数和调用时间的监控中心。

Container：服务运行容器，如 jetty。

RPC 执行过程总结：

服务容器负责启动，加载，运行服务提供者。

服务提供者在启动时，向注册中心注册自己提供的服务，暴露自己的 IP 和端口信息。

服务消费者在启动时，向注册中心订阅自己所需的服务。

注册中心返回服务提供者列表给消费者，如果有变更，注册中心将基于长连接推送给数据消费者。

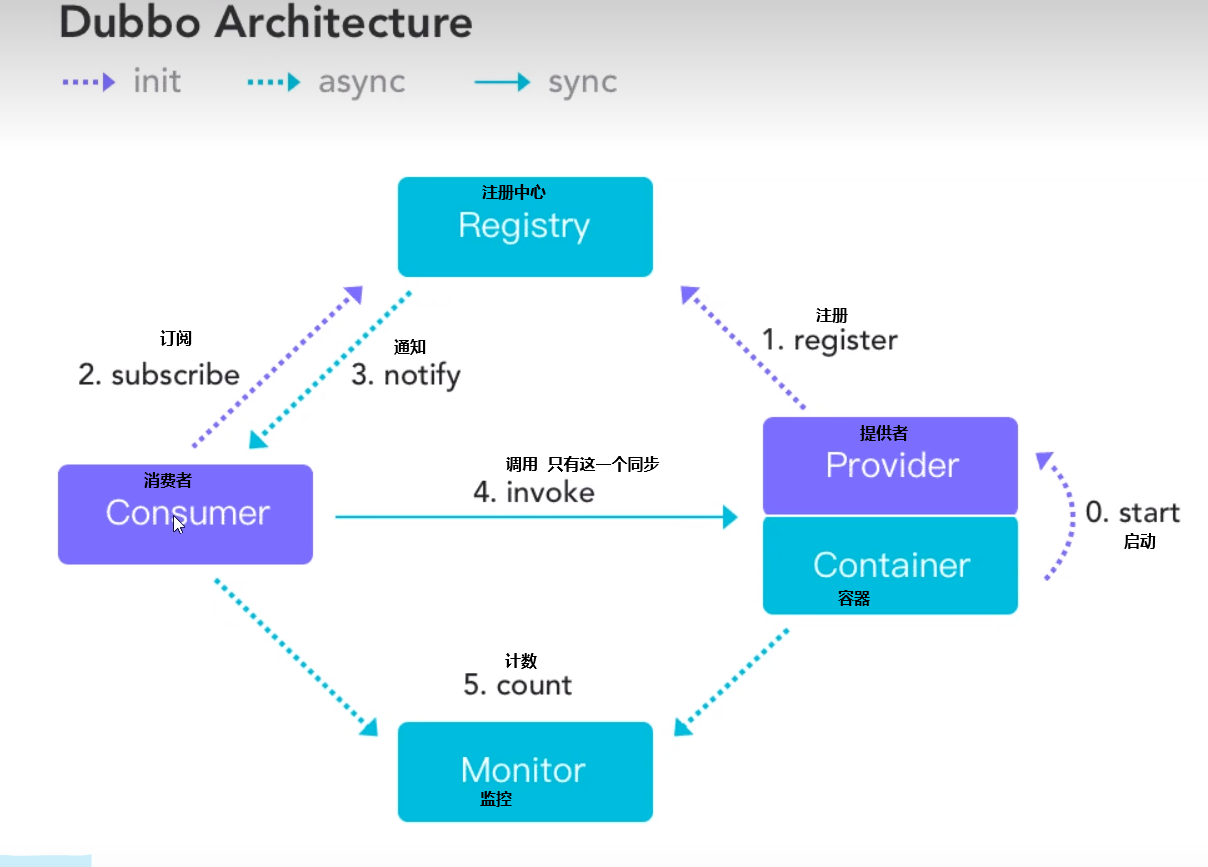
服务消费者，从提供这地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另外一台服务调用。

服务消费者和提供者，在内存中累计调用次数和调用时间，定时发送一次统计数据到监控中心。

Springcloud五大  
 服务注册与发现——Netflix Eureka  
 负载均衡：  
 客户端负载均衡——Netflix Ribbon  
 服务端负载均衡：——Feign(其也是依赖于Ribbon，只是将调用方式RestTemplete 更改成Service 接口)  
 断路器——Netflix Hystrix   
 服务网关——Netflix Zuul 狂神的这套视频看完了，建议补充学习gateway  
 分布式配置——Spring Cloud Config  
  
  
### cloud是基础，eureka是服务注册和发现，consumer是消费者去消费provider里的东西，消费方式就是Feign和Ribbon，feign接口消费，ribbonRest消费  
### Zuul代理 + 路由 + 过滤 三大功能 ，服务进来先走zuul路由网关  
  
### 熔断：代码写在提供方，相当于处理、返回异常；降级：代码写在消费方，提供方关闭服务后，提示、返回异常信息  
  
### 负载均衡分两种，进程式和集中式。进程式是客户端对同名微服务的调用方式依照什么规则。Zuul是对所有微服务做统筹。负载均衡对同名  
  
### Zuul 服务最终还是会注册进 Eureka  
  
   
#### 1.1、 什么是微服务？  
 微服务架构是将复杂臃肿的单体应用进行细粒度的服务化拆分，  
 每个拆分出来的服务各自独立打包部署，  
 并交由小团队进行开发和运维，  
 从而极大地提高了应用交付的效率，  
 并被各大互联网公司所普遍采用。

【微服务架构是一种架构模式，或者说是一种架构风格， 将**单一的应用程序划分成一组小的服务**，每个服务运行在其**独立的自己的进程内**，**服务之间互相协调，互相配置**，为用户提供最终价值。服务之间采用**轻量级的通信机制**互相沟通，每个服务都围绕着具体的业务进行构建，并且**能够被独立的部署到生产环境**中，另外，应尽量避免统一的，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，可以有一个非常**轻量级的集中式管理来协调这些服务**，**可以使用不同的语言来编写服务，也可以使用不同的数据存储**;】

#### 1.2 、微服务之间是如何独立通讯的？  
 同步通信：dubbo通过RPC远程过程调用、springcloud通过 REST接口json调用等。  
 异步：消息队列，如：RabbitMq、ActiveMq、Kafka等。  
  
#### 1.3 、SpringCloud 和 Dubbo有那些区别？  
 1、dubbo由于是二进制的传输，占用带宽会更少。  
 2、springCloud是http协议传输，带宽会比较多，同时使用http协议一般会使用JSON报文，消耗会更大。  
 3、dubbo的开发难度较大，原因是dubbo的jar包依赖问题，很多大型工程无法解决。  
 4、springcloud的接口协议约定比较自由且松散，需要有强有力的行政措施来限制接口无序升级。  
 5、dubbo的注册中心可以选择zk、redis等多种，springcloud的注册中心，只能用eureka或者自研。  
  
#### 1.4 、SpringBoot 和 SpringCloud，请谈谈你对他们的理解  
 springboot和springcloud是渐进式的关系,一个用来构建微服务，一个用来协调微服务  
  
#### 1.5 、什么是服务熔断？什么是服务降级？  
 服务熔断：当下游的服务因为某种原因突然变得不可用或响应过慢，  
 上游服务为了保证自己整体服务的可用性，不再继续调用目标服务，直接返回，快速释放资源。  
 如果目标服务情况好转则恢复调用。  
  
 那么，什么是服务降级呢？  
 这里有两种场景:  
 当下游的服务因为某种原因响应过慢，下游服务主动停掉一些不太重要的业务，释放出服务器资源，增加响应速度！  
 当下游的服务因为某种原因不可用，上游主动调用本地的一些降级逻辑，避免卡顿，迅速返回给用户！  
 其实乍看之下，很多人还是不懂熔断和降级的区别!  
 其实应该要这么理解:  
 服务降级有很多种降级方式！如开关降级、限流降级、熔断降级!  
 服务熔断属于降级方式的一种！  
  
### 1.6 、微服务的优缺点分别是什么？说下你在项目开发中遇到的坑  
 优点：  
 松耦合，聚焦单一业务功能，无关开发语言，团队规模降低。在开发中，不需要了解多有业务，  
 只专注于当前功能，便利集中，功能小而精。微服务一个功能受损，对其他功能影响并不是太大，可以快速定位问题。  
 微服务只专注于当前业务逻辑代码，不会和 html、css 或其他界面进行混合。可以灵活搭配技术，独立性比较舒服。  
   
 缺点：  
 随着服务数量增加，管理复杂，部署复杂，服务器需要增多，服务通信和调用压力增大，运维工程师压力增大，  
 人力资源增多，系统依赖增强，数据一致性，性能监控。  
  
### 1.7 、你所知道的微服务技术栈有哪些？列举一二  
 Spring Cloud NetFlix(一站式) 、Apache Dubbo+Zookeeper: 半自动、SpringCloud Alibaba  
  
### 1.8、 Eureka和Zookeeper都可以提供服务注册与发现的功能，请说说两者的区别  
 Eureka是AP，保证一致性，Zookeeper是CP，保证可用性

1. 服务很多，客户端该怎么访问? gateway路由网关 ； Zuul代理 + 路由 + 过滤 三大功能 ，服务进来先走zuul路由网关
2. 这么多服务?服务之间如何通信? 同步通信：dubbo通过RPC远程过程调用、springcloud通过 REST接口json调用等。 异步：消息队列，如：RabbitMq、ActiveMq、Kafka等。
3. 这么多服务?如何治理? Eureka ；Nacos 注册中心。提供者先注册信息到注册中心，消费者到注册中心订阅数据，数据变化注册中出会通知消费者
4. 
5. 服务挂了怎么办? Hystrix 自动降级、熔断

**SpringcloudAlibaba**

## 主要功能：  
- \*\*服务限流降级:\*\* 默认支持WebServlet、webFlux，OpenFeign、RestTemplate、Spring Cloud Gateway,Zuul，Dubbo和RocketMQ限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。  
- \*\*服务注册与发现:\*\* 适配Spring Cloud服务注册与发现标准，默认集成了Ribbon的支持。  
- \*\*分布式配置管理:\*\* 支持分布式系统中的外部化配置，配置更改时自动刷新。   
- \*\*消息驱动能力:\*\* 基于Spring Cloud Stream为微服务应用构建消息驱动能力。  
- \*\*分布式事务:\*\* 使用@GlobalTransactional注解，高效并且对业务零侵入地解决分布式事务问题。  
- \*\*阿里云对象存储:\*\* 阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。  
- \*\*分布式任务调度:\*\* 提供秒级、精准、高可靠、高可用的定时(基于Cron表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有Worker (schedulerx-client)上执行。  
- \*\*阿里云短信服务:\*\* 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。  
  
## 主要组件：  
- \*\*sentinel:\*\* 把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。  
- \*\*Nacos:\*\* 一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。  
- \*\*RocketMQ:\*\* 一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。  
- \*\*Dubbo:\*\* Apache Dubbo是一款高性能Java RPC框架。  
- \*\*Seata:\*\* 阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。（本次没用到）  
- \*\*Alibaba Cloud ACM:\*\* 一款在分布式架构环境中对应用配置进行集中管理和推送的应用配置中心产品。  
- \*\*Alibaba Cloud OSS:\*\* 阿里云对象存储服务(Object Storage Service，简称OSS)，是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。  
- \*\*阿里云短信服务:\*\* 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。  
- \*\*Alibaba Cloud SchedulerX:\*\* 阿里中间件团队开发的一款分布式任务调度产品，提供秒级、精准、高可靠、高可用的定时(基于Cron表达式)任务调度服务。  
- \*\*Alibaba Cloud SMS:\*\* 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。