3.微服务间如何通信的?

第一种：远程过程调用，即RPC（Remote Procedure Invocation）直接通过远程过程调用来访问别的service，如：REST、Apache。

第二种：消息使用异步消息来做服务间通信。服务间通过消息管道来交换消息，从而通信。如：Apache Kafka、RabbitMQ

使用过哪些微服务组件？分别说说其功能作用。

Spring Cloud Eureka：服务注册与发现
Spring Cloud Zuul：服务网关
Spring Cloud Ribbon：客户端负载均衡
Spring Cloud Feign：声明性的Web服务客户端
Spring Cloud Hystrix：断路器
Spring Cloud Confifig：分布式统一配置管理
等20几个框架，开源一直在更新

共有多少种设计模式？使用过哪些设计模式?

共有23种。

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

常用的Spring注解有哪些?

一.用于创建对象的注解

作用：和在xml配置文件中编写一个<bean>标签实现的功能一样。

1.@Component : 用于把当前类对象存入Spring容器中。

属性：value --- 用于指定bean的id。如果不写该属性，id的默认值是当前类名，且首字母改为小写。

2.@Controller : 一般用在表现层。

3.@Service : 一般用在业务层。

4.@Repository : 一般用在持久层(dto层)。

备注：其中2，3，4注解的作用和属性与@Component 一模一样，他们三个是Spring框架为我们提供的三层架构使用的注解，使我们对三层对象更加清晰。

二.用于注入数据的注解

作用：和在xml配置文件中的<bean>标签中写一个<property>标签的功能一样。

1.@Autowired ： 自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功。可以作用在变量或者方法上。

2.@Qualifier ： 在按照类型注入的基础之上再按照名称注入，它在给类成员注入时要和@Autowired配合使用，但是在给方法参数注入是可以单独使用。

3.@Resource : 直接按照bean的id注入，可以独立使用。

属性：name --- 用于指定bean的id。

备注：以上三个注入都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实现。另外，集合类型的注入只能通过xml来实现。

4.@Value : 用于注入基本类型和String类型的数据。

属性：value --- 用于指定数据的值，它可以使用Spring中的SpEL(也就是Spring中的el表达式)。SpEL的写法：${表达式}

5.@RequestMapping：将特定的Http请求方法映射到控制器中的特定类/方法上。

三.用于改变作用范围的注解

作用：和在xml配置文件中的<bean>标签中使用scope属性实现的功能一样。

1.@Scope : 用于指定bean的作用范围。

属性：value --- 指定范围的取值。常用取值：singleton(单例)和prototype(多例)。

四.和生命周期相关的注解

作用：和在xml配置文件中的<bean>标签中使用init-method和destory-method属性实现的功能一样。

1.@PreDestory : 用于指定销毁方法。

2.@PostConstruct : 用于指定初始化方法。

五.Spring新注解

1.@Configuration : 用于指定当前类是一个配置类。

2.@ComponentScan : 用于通过注解指定Spring在创建容器时要扫描的包。

3.@Bean : 用于把当前方法的返回值作为bean对象存入Spring的IOC容器中。

属性：name --- 用于指定bean的id。当不写时，默认值为当前方法的名称。

4.@Import : 用于导入其他的配置类。

属性：value --- 用于指定其他配置类的字节码。当我们使用@Import注解时，有@Import注解的类就是父配置类。

5.@PropertySource：用于指定properties文件的位置。

属性：value --- 指定文件的名称和路径。关键字classpath表示类路径下。

1. 谈谈IOC和AOP

IOC(Inversion of Control)：依赖注入，控制反转

(1). IoC（Inversion of Control）是指容器控制程序对象之间的关系，而不是传统实现中，由程序代码直接操控。控制权由应用代码中转到了外部容器，控制权的转移是所谓反转。 对于Spring而言，就是由Spring来控制对象的生命周期和对象之间的关系；IoC还有另外一个名字——“依赖注入（Dependency Injection）”。从名字上理解，所谓依赖注入，即组件之间的依赖关系由容器在运行期决定，即由容器动态地将某种依赖关系注入到组件之中。

(2). 在Spring的工作方式中，所有的类都会在spring容器中登记，告诉spring这是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。

(3). 在系统运行中，动态的向某个对象提供它所需要的其他对象。

(4). 依赖注入的思想是通过反射机制实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中。 总而言之，在传统的对象创建方式中，通常由调用者来创建被调用者的实例，而在Spring中创建被调用者的工作由Spring来完成，然后注入调用者，即所谓的依赖注入or控制反转。 注入方式有两种：依赖注入和设置注入；

IoC的优点：降低了组件之间的耦合，降低了业务对象之间替换的复杂性，使之能够灵活的管理对象

依赖注入的三种方式：

（1）构造器；（2）setter；（3）注解方式；

AOP:

(1). AOP面向方面编程基于IoC，是对OOP的有益补充；

(2). AOP利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了 多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的 逻辑或责任封装起来，比如日志记录，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

(3). AOP代表的是一个横向的关 系，将“对象”比作一个空心的圆柱体，其中封装的是对象的属性和行为；则面向方面编程的方法，就是将这个圆柱体以切面形式剖开，选择性的提供业务逻辑。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹，但完成了效果。

(4). 实现AOP的技术，主要分为两大类：

一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

(5). Spring实现AOP：JDK动态代理和CGLIB代理

1. Redis缓存如何保持数据的一致性

读数据的时候首先去 Redis 中读取，没有读到再去 MySQL 中读取，读取都数据更新到 Redis 中作为下一次的缓存。

写数据的时候会产生数据不一致的问题。无论是先写入 Redis 再写入 MySQL 中，还是先写入 MySQL 再写入 Redis 中，这两步操作都不能保证原子性，所以会出现 Redis 和 MySQL 中数据不一致的问题。

无论采取何种方式都不能保证强一致性，如果对 Redis 中的数据设置了过期时间，能够保证最终一致性，对架构的优化只能降低发生的概率，不能从根本上避免不一致性。

更新缓存的两种方式：删除失效缓存、更新缓存

更新缓存和数据库有两种顺序：先数据库后缓存、先缓存后数据库两两组合，分为四种更新策略

1.数据库类别

关系型数据库：mysql、oracle、sqlserver

非关系型数据库：redis、memcache、mogodb、hadoop，NoSql

2.事务特性（ACID）：

事务是对**数据库中一系列操作进行统一的回滚或者提交的操作，主要用来保证数据的完整性和一致性。**

(1)原子性（Atomicity）：原子性是指一个事务中的操作，要么全部成功，要么全部失败，如果失败，就回滚到事务开始前的状态。

(2)一致性（Consistency）：一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。那转账举栗子，A账户和B账户之间相互转账，无论如何操作，A、B账户的总金额都必须是不变的。

(3)隔离性（Isolation）：隔离性是当多个用户 并发的 访问数据库时，如果操作同一张表，数据库则为每一个用户都开启一个事务，且事务之间互不干扰，也就是说事务之间的并发是隔离的。再举个栗子，现有两个并发的事务T1和T2，T1要么在T2开始前执行，要么在T2结束后执行，如果T1先执行，那T2就在T1结束后在执行。关于数据的隔离性级别，将在后文讲到。

(4)持久性（Durability）：持久性就是指如果事务一旦被提交，数据库中数据的改变就是永久性的，即使断电或者宕机的情况下，也不会丢失提交的事务操作。

1. 抽象类和接口有什么区别?

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	子类使用**extends**关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。	子类使用关键字**implements**来实现接口。它需要提供接口中所有声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常Java类的区别	除了你不能实例化抽象类之外，它和普通Java类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有**public**、**protected**和**default**这些修饰符	接口方法默认修饰符是**public**。你不可以使用其它修饰符。
main方法	抽象方法可以有main方法并且我们可以运行它	接口没有main方法，因此我们不能运行它。
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
添加新方法	如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。

2. 谈谈java的垃圾回收机制

1. 常见的垃圾回收算法

标记清除法、引用计数法、可达性分析

2. 垃圾回收区域

Java垃圾回收只针对堆和方法区的内存。程序计数器、虚拟机栈、本地方法栈随线程而生，随线程而灭，因此不用管。

3. 为什么出现Future机制

常见的两种创建线程的方式。一种是直接继承Thread，另外一种就是实现Runnable接口。

这两种方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。

从Java 1.5开始，就提供了Callable和Future，通过它们可以在任务执行完毕之后得到任务执行结果。Future模式的核心思想是能够让主线程将原来需要同步等待的这段时间用来做其他的事情。（因为可以异步获得执行结果，所以不用一直同步等待去获得执行结果）上图简单描述了不使用Future和使用Future的区别，不使用Future模式，主线程在invoke完一些耗时逻辑之后需要等待，这个耗时逻辑在实际应用中可能是一次RPC调用，可能是一个本地IO操作等。B图表达的是使用Future模式之后，我们主线程在invoke之后可以立即返回，去做其他的事情，回头再来看看刚才提交的invoke有没有结果。

4. HashSet 怎么保证元素不重复的?

public boolean add(E e) { return map.put(e, PRESENT)==null; }元素值作为的是 map 的 key，map 的 value 则是 PRESENT 变量，这个变量只作为放入 map 时的一个占位符而存在，所以没什么实际用处。其实，这时候答案已经出来了：HashMap 的 key 是不能重复的，而这里HashSet 的元素又是作为了 map 的 key，当然也不能重复了。

5. 创建线程的几种方式?

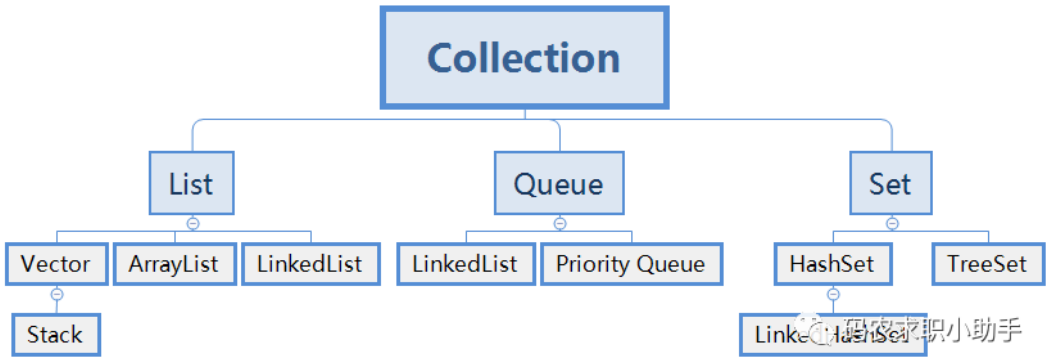
(1)继承 Thread 类创建线程；

(2)实现 Runnable 接口创建线程；

(3)通过 Callable 和 Future 创建线程；
(4)通过线程池创建线程。

6. Java 中常用的容器有哪些?

常见容器主要包括 Collection 和 Map 两种，Collection 存储着对象的集合，而 Map 存储着键值对（两个对象）的映射表



Collection
Set
TreeSet：基于红黑树实现，支持有序性操作，例如：根据一个范围查找元素的操作。但是查找效率不如 HashSet，HashSet 查找的时间复杂度为 O(1)，TreeSet 则为 O(logN)。
HashSet：基于哈希表实现，支持快速查找，但不支持有序性操作。并且失去了元素的插入顺序信息，也就是说使用 Iterator 遍历 HashSet 得到的结果是不确定的。
LinkedHashSet：具有 HashSet 的查找效率，且内部使用双向链表维护元素的插入顺序。
List
ArrayList：基于动态数组实现，支持随机访问。
Vector：和 ArrayList 类似，但它是线程安全的（这里需要注意：vector的单个操作时原子性的，也就是线程安全的。但是如果两个原子操作复合而来，这个组合的方法是非线程安全的，需要使用锁来保证线程安全，具体可以看这篇文章：Vection的非线程安全操作
LinkedList：基于双向链表实现，只能顺序访问，但是可以快速地在链表中间插入和删除元素。不仅如此，LinkedList 还可以用作栈、队列和双向队列。 **Queue**
LinkedList：可以用它来实现双向队列。
PriorityQueue：基于堆结构实现，可以用它来实现优先队列。
Map
TreeMap：基于红黑树实现。 HashMap：基于哈希表实现。 HashTable：和 **HashMap** 类似，但它是线程安全的，这意味着同一时刻多个线程可以同时写入
HashTable 并且不会导致数据不一致。它是遗留类，不应该去使用它。现在可以使用 ConcurrentHashMap 来支持线程安全，并且 ConcurrentHashMap 的效率会更高，因为 ConcurrentHashMap 引入了分段锁。 LinkedHashMap：使用双向链表来维护元素的顺序，顺序为插入顺序或者最近最少使用（LRU）顺序。

7. ArrayList 的扩容机制?

当使用 add 方法的时候首先调用 ensureCapacityInternal 方法，传入 size+1 进去，检查是否需要扩充 elementData 数组的大小；
newCapacity = 扩充数组为原来的 1.5 倍(不能自定义)，如果还不够，就使用它指定要扩充的大小 minCapacity ，然后判断 minCapacity 是否大于 MAX_ARRAY_SIZE(Integer.MAX_VALUE - 8)，如果大于，就取 Integer.MAX_VALUE；
扩容的主要方法：grow；
ArrayList 中 copy 数组的核心就是 System.arraycopy 方法，将 original 数组的所有数据复制到 copy 数组中，这是一个本地方法。

8. Hashmap 底层原理?

在JDK1.6，JDK1.7中，HashMap采用位桶+链表实现，即使用链表处理冲突，同一hash值的链表都存储在一个链表里。但是当位于一个桶中的元素较多，即hash值相等的元素较多时，通过key值依次查找的效率较低。而JDK1.8中，HashMap采用位桶+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。

首先有一个每个元素都是链表（可能表述不准确）的数组，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，同一各链表上的Hash值是相同的，所以说数组存放的是链表。而当链表长度太长时，链表就转换为红黑树，这样大大提高了查找的效率。

即HashMap的原理图是：