一、Java 基础 1.JDK 动态代理和 CGLIB 动态代理的区别"JDK 动态代理只能对实现了接口的类生成代理，而不能针对类。CGLIB 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法。因为是继承，所以该类或方法最好不要声明成 final。"2.静态代理和动态代理的区别静态代理中代理类在编译期就已经确定，而动态代理则是 JVM 运行时动态生成，静态代理的效率相对动态代理来说相对高一些，但是静态代理代码冗余大，一单需要修改接口，代理类和委托类都需要修改。3.ArrayList 和 LinkedList 有什么区别？"ArrayList 和 LinkedList 的差别主要来自于 Array 和 LinkedList 数据结构的不同。ArrayList 是基于数组实现的，LinkedList 是基于双链表实现的。另外 LinkedList 类不仅是 List 接口的实现类，可以根据索引来随机访问集合中的元素，除此之外，LinkedList 还实现了 Deque 接口，Deque 接口是 Queue 接口的子接口，它代表一个双向队列，因此 LinkedList 可以作为双向队列，栈（可以参见 Deque 提供的接口方法）和 List 集合使用，功能强大。因为 Array 是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的，可以直接返回数组中 index 位置的元素，因此在随机访问集合元素上有较好的性能。Array 获取数据的时间复杂度是 O(1),但是要插入、删除数据却是开销很大的，因为这需要移动数组中插入位置之后的的所有元素。相对于 ArrayList，LinkedList 的随机访问集合元素时性能较差，因为需要在双向列表中找到要 index 的位置，再返回；但在插入，删除操作是更快的。因为 LinkedList 不像 ArrayList 一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是 ArrayList 最坏的一种情况，时间复杂度是 O(n)，而 LinkedList 中插入或删除的时间复杂度仅为 O(1)。ArrayList 在插入数据时还需要更新索引（除了插入数组的尾部）。LinkedList 需要更多的内存，因为 ArrayList 的每个索引的位置是实际的数据，而 LinkedList 中的每个节点中存储的是实际的数据和前后节点的位置。"4.重写和重载的区别"重写是子类对父类的允许访问的方法的实现过程进行重新编写,返回值和形参都不能改变。即外壳不变，核心重写！重写的好处在于子类可以根据需要，定义特定于自己的行为。也就是说子类能够根据需要实现父类的方法。重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。重载(overloading)是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。"5.Java8 的接口新增了哪些特性？增加了 default 方法和 static 方法，这 2 种方法可以有方法体。6.抽象类和接口（Java7）的区别"抽象类可以提供成员方法的实现细节，而接口中只能存在 publicabstract 方法；抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 publicstaticfinal 类型的；接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；一个类只能继承一个抽象类，而一个类却可以实现多个接口。"7.为什么要有 hashCode"我们以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode：当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 equals()方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入搜查成功。如果不同的话，就会重新散列到其他位置。(摘自我的 Java 启蒙书《Headfirstjava》第二版)。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。hashCode()与 equals()的相关规定如果两个对象相等，则 hashcode 一定也是相同的两个对象相等，对两个对象分别调用 equals 方法都返回 true 两个对象有相同的 hashcode 值，它们也不一定是相等的因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖 hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）对象的相等与指向他们的引用相等，两者有什么不同？对象的相等比的是内存中存放的内容是否相等而引用相等比较的是他们指向的内存地址是否相等。"8.hashCode()介绍"hashCode()的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode()定义在 JDK 的 Object.java 中，这就意味着 Java 中的任何类都包含有 hashCode()函数。散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！(可以快速找到所需要的对象）"9.hashCode 与 equals(重要)"HashSet 如何检查重复两个对象的 hashCode()相同，则 equals()也一定为 true，对吗？hashCode 和 equals 方法的关系面试官可能会问你：“你重写过 hashcode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”"10.Java 中异常分为哪些种类？"按照异常需要处理的时机分为编译时异常(也叫受控异常)也叫 CheckedException 和运行时异常(也叫非受控异常)也叫 UnCheckedException。Java 认为 Checked 异常都是可以被处理的异常，所以 Java 程序必须显式处理 Checked 异常。如果程序没有处理 Checked 异常，该程序在编译时就会发生错误无法编译。这体现了 Java 的设计哲学：没有完善错误处理的代码根本没有机会被执行。对 Checked 异常处理方法有两种：第一种：当前方法知道如何处理该异常，则用 try...catch 块来处理该异常。第二种：当前方法不知道如何处理，则在定义该方法时声明抛出该异常。运行时异常只有当代码在运行时才发行的异常，编译的时候不需要 try...catch。Runtime 如除数是 0 和数组下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。当然如果你有处理要求也可以显示捕获它们。"11.内部类的分类有哪些内部类可以分为四种：成员内部类、局部内部类、匿名内部类和静态内部类。12.什么是内部类？在 Java 中，可以将一个类的定义放在另外一个类的定义内部，这就是内部类。内部类本身就是类的一个属性，与其他属性定义方式一致。13.什么是方法的返回值？返回值的作用是什么？方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果！(前提是该方法可能产生结果)。返回值的作用:接收出结果，使得它可以用于其他的操作！14.静态方法和实例方法有何不同？"静态方法和实例方法的区别主要体现在两个方面：在外部调用静态方法时，可以使用""类名.方法名""的方式，也可以使用""对象名.方法名""的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制"15.静态变量和实例变量区别"静态变量：静态变量由于不属于任何实例对象，属于类的，所以在内存中只会有一份，在类的加载过程中，JVM 只为静态变量分配一次内存空间。实例变量：每次创建对象，都会为每个对象分配成员变量内存空间，实例变量是属于实例对象的，在内存中，创建几次对象，就有几份成员变量。"16.构造方法有哪些特性？"名字与类名相同；没有返回值，但不能用 void 声明构造函数；生成类的对象时自动执行，无需调用。"17.在 Java 中定义一个不做事且没有参数的构造方法的作用 Java 程序在执行子类的构造方法之前，如果没有用 super()来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 super()来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。18.break,continue,return 的区别及作用"break 跳出总上一层循环，不再执行循环(结束当前的循环体)continue 跳出本次循环，继续执行下次循环(结束正在执行的循环进入下一个循环条件)return 程序返回，不再执行下面的代码(结束当前的方法直接返回)"19.static 注意事项"1、静态只能访问静态。2、非静态既可以访问非静态的，也可以访问静态的。"20.static 应用场景"因为 static 是被类的实例对象所共享，因此如果某个成员变量是被所有对象所共享的，那么这个成员变量就应该定义为静态变量。因此比较常见的 static 应用场景有：1、修饰成员变量 2、修饰成员方法 3、静态代码块 4、修饰类【只能修饰内部类也就是静态内部类】5、静态导包"21.static 的独特之处"1、被 static 修饰的变量或者方法是独立于该类的任何对象，也就是说，这些变量和方法不属于任何一个实例对象，而是被类的实例对象所共享。怎么理解“被类的实例对象所共享”这句话呢？就是说，一个类的静态成员，它是属于大伙的【大伙指的是这个类的多个对象实例，我们都知道一个类可以创建多个实例！】，所有的类对象共享的，不像成员变量是自个的【自个指的是这个类的单个实例对象】...我觉得我已经讲的很通俗了，你明白了咩？2、在该类被第一次加载的时候，就会去加载被 static 修饰的部分，而且只在类第一次使用时加载并进行初始化，注意这是第一次用就要初始化，后面根据需要是可以再次赋值的。3、static 变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋值的话，是可以任意赋值的！4、被 static 修饰的变量或者方法是优先于对象存在的，也就是说当一个类加载完毕之后，即便没有创建对象，也可以去访问。"22.static 存在的主要意义"static 的主要意义是在于创建独立于具体对象的域变量或者方法。以致于即使没有创建对象，也能使用属性和调用方法！static 关键字还有一个比较关键的作用就是用来形成静态代码块以优化程序性能。static 块可以置于类中的任何地方，类中可以有多个 static 块。在类初次被加载的时候，会按照 static 块的顺序来执行每个 static 块，并且只会执行一次。为什么说 static 块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在 static 代码块中进行。"23.this 与 super

的区别"super:它引用当前对象的直接父类中的成员（用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）this：它代表当前对象名（在程序中易产生二义性之处，应使用 this 来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用 this 来指明成员变量名）super()和 this()类似,区别是，super()在子类中调用父类的构造方法，this()在本类内调用本类的其它构造方法。super()和 this()均需放在构造方法内第一行。尽管可以用 this 调用一个构造器，但却不能调用两个。this 和 super 不能同时出现在一个构造函数里面，因为 this 必然会调用其它的构造函数，其它的构造函数必然也会有 super 语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。this()和 super()都指的是对象，所以，均不可以在 static 环境中使用。包括：static 变量,static 方法，static 语句块。从本质上讲，this 是一个指向本对象的指针,然而 super 是一个 Java 关键字。"24.super 关键字的用法"super 可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。super 也有三种用法：1.普通的直接引用与 this 类似，super 相当于是指向当前对象的父类的引用，这样就可以用 super.xxx 来引用父类的成员。2.子类中的成员变量或方法与父类中的成员变量或方法同名时，用 super 进行区分 3.引用父类构造函数 super（参数）：调用父类中的某一个构造函数（应该为构造函数中的第一条语句）。this（参数）：调用本类中另一种形式的构造函数（应该为构造函数中的第一条语句）。"25.String 类的常用方法有哪些？"indexof();返回指定字符的的索引。charAt();返回指定索引处的字符。replace();字符串替换。trim();去除字符串两端空格。splt()；字符串分割，返回分割后的字符串数组。getBytes()；返回字符串 byte 类型数组。length()；返回字符串长度。toLowerCase();将字符串转换为小写字母。•toUpperCase();将字符串转换为大写字母。substring();字符串截取。equals();比较字符串是否相等。"26.char 型变量中能否能不能存储一个中文汉字，为什么？char 可以存储一个中文汉字，因为 Java 中使用的编码是 Unicode(不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个 char 类型占 2 个字节（16 比特），所以放一个中文是没问题的。27.是否可以继承 String 类？"String 类是 final 类，不可以被继承。补充：继承 String 本身就是一个错误的行为，对 String 类型最好的重用方式是关联关系（Has-A）和依赖关系（Use-A）而不是继承关系（Is-A）。"28.两个对象值相同(x.equals(y)==true)，但却可有不同的 hashcode，这句话对不对？"不对，如果两个对象 x 和 y 满足 x.equals(y)==true，它们的哈希码（hashcode）应当相同。Java 对于 eqauls 方法和 hashCode 方法是这样规定的：(1)如果两个对象相同（equals 方法返回 true），那么它们的 hashCode 值一定要相同；(2)如果两个对象的 hashCode 相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在 Set 集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。"29.构造器（constructor）是否可被重写（override）？构造器不能被继承，因此不能被重写，但可以被重载。30.谈谈你对多态的理解？多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，不用修改源代码，就可以让引用变量绑定到各种不同的对象上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。31.Java 中实现多态的机制是什么？Java 中的多态靠的是父类或接口定义的引用变量可以指向子类或具体实现类的实例对象，而程序调用的方法在运行期才动态绑定，就是引用变量所指向的具体实例对象的方法，也就是内存里正在运行的那个对象的方法，而不是引用变量的类型中定义的方法。32.new 一个对象的过程和 clone 一个对象的区别？"new 操作符的本意是分配内存。程序执行到 new 操作符时，首先去看 new 操作符后面的类型，因为知道了类型，才能知道要分配多大的内存空间。分配完内存之后，再调用构造函数，填充对象的各个域，这一步叫做对象的初始化，构造方法返回后，一个对象创建完毕，可以把他的引用（地址）发布到外部，在外部就可以使用这个引用操纵这个对象。clone 在第一步是和 new 相似的，都是分配内存，调用 clone 方法时，分配的内存和原对象（即调用 clone 方法的对象）相同，然后再使用原对象中对应的各个域，填充新对象的域，填充完成之后，clone 方法返回，一个新的相同的对象被创建，同样可以把这个新对象的引用发布到外部。"33.深克隆和浅克隆？"浅克隆：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。"34.Java 中为什么要用 clone？在实际编程过程中，我们常常要遇到这种情况：有一个对象 A，在某一时刻 A 中已经包含了一些有效值，此时可能会需要一个和 A 完全相同新对象 B，并且此后对 B 任何改动都不会影响到 A 中的值，也就是说，A 与 B 是两个独立的对象，但 B 的初始值是由 A 对象确定的。在 Java 语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但 clone()方法是其中最简单，也是最高效的手段。35.Java 中操作字符串都有哪些类？它们之间有什么区别？"操作字符串的类有：String、StringBuffer、StringBuilder。String 和 StringBuffer、StringBuilder 的区别在于 String 声明的是不可变的对象，每次操作都会生成新的 String 对象，再将指针指向新的 String 对象，而 StringBuffer、StringBuilder 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 String。StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全的，而 StringBuilder 是非线程安全的，但 StringBuilder 的性能却高于 StringBuffer，所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。"36.Stringstr=“i”和 Stringstr=newString(“1”)一样吗？不一样，因为内存的分配方式不一样。Stringstr="i"的方式 JVM 会将其分配到常量池中，而 Stringstr=newString("i")JVM 会将其分配到堆内存中。37.finalfinallyfinalize 的区别"final 可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。finally 一般作用在 try-catch 代码块中，在处理异常的时候，通常我们将一定要执行的代码方法 finally 代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。finalize 是一个方法，属于 Object 类的一个方法，而 Object 类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用 System.gc()方法的时候，由垃圾回收器调用 finalize()，回收垃圾，一个对象是否可回收的最后判断。"38.final 有什么用？"用于修饰类、属性和方法；被 final 修饰的类不可以被继承被 final 修饰的方法不可以被重写被 final 修饰的变量不可以被改变，被 final 修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的"39.Java 有哪些数据类型"定义：Java 语言是强类型语言，对于每一种数据都定义了明确的具体的数据类型，在内存中分配了不同大小的内存空间。基本数据类型数值型整数类型(byte,short,int,long)浮点类型(float,double)字符型(char)布尔型(boolean)引用数据类型类(class)接口(interface)数组([])"40.什么是 Java 注释"定义：用于解释说明程序的文字 Java 注释的分类单行注释格式：单行注释格式：//注释文字多行注释格式：/注释文字/文档注释格式：/*注释文字/Java 注释的作用在程序中，尤其是复杂的程序中，适当地加入注释可以增加程序的可读性，有利于程序的修改、调试和交流。注释的内容在程序编译的时候会被忽视，不会产生目标代码，注释的部分不会对程序的执行结果产生任何影响。注意事项：多行和文档注释都不能嵌套使用。"41.用最有效率的方法计算 2 乘以 8？2<<342.Math.round(11.5)等于多少？Math.round(-11.5)等于多少？Math.round(11.5)的返回值是 12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加 0.5 然后进行下取整。43.&和&&的区别？"&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是 true 整个表达式的值才是 true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为：username!=null&&!username.equals(""")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会产生 NullPointerException 异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。"44.Java 有没有 goto？goto 是 Java 中的保留字，在目前版本的 Java 中没有使用。（根据 JamesGosling（Java 之父）编写的《TheJavaProgrammingLanguage》一书的附录中给出了一个 Java 关键字列表，其中有 goto 和 const，但是这两个是目前无法使用的关键字，因此有些地方将其称之为保留字，其实保留字这个词应该有更广泛的意义，因为熟悉 C 语言的程序员都知道，在系统类库中使用过的有特殊意义的单词或单词的组合都被视为保留字）45.floatf=3.4;是否正确？不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（downcasting，也称为窄化）会造成精度损失，因此需要强制类型转换 floatf=(float)3.4;或者写成 floatf=3.4F;。46.访问修饰符 public,private,protected,以及不写（默认）时的区别？"饰符当

前类同包子类其他包 public√√√√protected√√√×default√√××private√×××类的成员不写访问修饰时默认为 default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java 中，外部类的修饰符只能是 public 或默认，类的成员（包括内部类）的修饰符可以是以上四种。"47.Java 语言有哪些特点封装、继承、多态、抽象。48.什么是 Java 程序的主类？应用程序和小程序的主类有何不同？一个程序中可以有多个类，但只能有一个类是主类。在 Java 应用程序中，这个主类是指包含 main()方法的类。而在 Java 小程序中，这个主类是一个继承自系统类 JApplet 或 Applet 的子类。应用程序的主类不一定要求是 public 类，但小程序的主类要求必须是 public 类。主类是 Java 程序执行的入口点。49.说下面向对象四大特性封装、继承、多态、抽象。二、JavaIO1.IO 多路复用的底层原理"IO 多路复用的底层原理 IO 多路复用使用两个系统调用(select/poll/epoll 和 recvfrom)，blockingIO 只调用了 recvfrom；select/poll/epoll 核心是可以同时处理多个 connection，而不是更快，所以连接数不高的话，性能不一定比多线程+阻塞 IO 好,多路复用模型中，每一个 socket，设置为 nonblocking,阻塞是被 select 这个函数 block，而不是被 socket 阻塞的。select 机制客户端操作服务器时就会产生这三种文件描述符(简称 fd)：writefds(写)、readfds(读)、和 exceptfds(异常)。select 会阻塞住监视 3 类文件描述符，等有数据、可读、可写、出异常或超时、就会返回；返回后通过遍历 fdset 整个数组来找到就绪的描述符 fd，然后进行对应的 IO 操作。优点：几乎在所有的平台上支持，跨平台支持性好缺点：由于是采用轮询方式全盘扫描，会随着文件描述符 FD 数量增多而性能下降。每次调用 select()，需要把 fd 集合从用户态拷贝到内核态，并进行遍历(消息传递都是从内核到用户空间)默认单个进程打开的 FD 有限制是 1024 个，可修改宏定义，但是效率仍然慢。poll 机制基本原理与 select 一致，只是没有最大文件描述符限制，因为采用的是链表存储 fd。epoll 机制 epoll 之所以高性能是得益于它的三个函数 epoll_create()系统启动时，在 Linux 内核里面申请一个 B+树结构文件系统，返回 epoll 对象，也是一个 fdepoll_ctl()每新建一个连接，都通过该函数操作 epoll 对象，在这个对象里面修改添加删除对应的链接 fd,绑定一个 callback 函数。epoll_wait()轮训所有的 callback 集合，并完成对应的 IO 操作优点：没 fd 这个限制，所支持的 FD 上限是操作系统的最大文件句柄数，1G 内存大概支持 10 万个句柄效率提高，使用回调通知而不是轮询的方式，不会随着 FD 数目的增加效率下降内核和用户空间 mmap 同一块内存实现"2.缓冲区是什么意思？"Buffer 是一个对象，它包含一些要写入或者刚读出的数据。在 NIO 中加入 Buffer 对象，体现了新库与原 I/O 的一个重要区别。在面向流的 I/O 中，您将数据直接写入或者将数据直接读到 Stream 对象中在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问 NIO 中的数据，您都是将它放到缓冲区中。缓冲区实质上是一个数组。通常它是一个字节数组，但是也可以使用其他种类的数组。但是一个缓冲区不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程 ByteBufferCharBufferShortBufferIntBufferLongBufferFloatBufferDoubleBuffer"3.通道是个什么意思？"通道是对原 I/O 包中的流的模拟。到任何目的地(或来自任何地方)的所有数据都必须通过一个 Channel 对象（通道）。一个 Buffer 实质上是一个容器对象。发送给一个通道的所有对象都必须首先放到缓冲区中；同样地，从通道中读取的任何数据都要读到缓冲区中。Channel 是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流。正如前面提到的，所有数据都通过 Buffer 对象来处理。您永远不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。"4.同步、异步、阻塞、非堵塞"同/异、阻/非堵塞的组合，有四种类型，如下表：组合方式性能分析同步非阻塞提升 I/O 性能的常用手段，就是将 I/O 的阻塞改成非阻塞方式，尤其在网络 I/O 是长连接，同时传输数据也不是很多的情况下，提升性能非常有效。这种方式通常能提升 I/O 性能，但是会增加 CPU 消耗，要考虑增加的 I/O 性能能不能补偿 CPU 的消耗，也就是系统的瓶颈是在 I/O 还是在 CPU 上。异步阻塞这种方式在分布式数据库中经常用到，例如在网一个分布式数据库中写一条记录，通常会有一份是同步阻塞的记录，而还有两至三份是备份记录会写到其它机器上，这些备份记录通常都是采用异步阻塞的方式写 I/O。异步阻塞对网络 I/O 能够提升效率，尤其像上面这种同时写多份相同数据的情况。异步非阻塞这种组合方式用起来比较复杂，只有在一些非常复杂的分布式情况下使用，像集群之间的消息同步机制一般用这种 I/O 组合方式。如 Cassandra 的 Gossip 通信机制就是采用异步非阻塞的方式。它适合同时要传多份相同的数据到集群中不同的机器，同时数据的传输量虽然不大，但是却非常频繁。这种网络 I/O 用这个方式性能能达到最高。"5.阻塞与非阻塞阻塞与非阻塞主要是从 CPU 的消耗上来说的，阻塞就是 CPU 停下来等待一个慢的操作完成 CPU 才接着完成其它的事。非阻塞就是在这个慢的操作在执行时 CPU 去干其它别的事，等这个慢的操作完成时，CPU 再接着完成后续的操作。虽然表面上看非阻塞的方式可以明显的提高 CPU 的利用率，但是也带了另外一种后果就是系统的线程切换增加。增加的 CPU 使用时间能不能补偿系统的切换成本需要好好评估。6.同步与异步同步就是一个任务的完成需要依赖另外一个任务时，只有等待被依赖的任务完成后，依赖的任务才能算完成，这是一种可靠的任务序列。要么成功都成功，失败都失败，两个任务的状态可以保持一致。而异步是不需要等待被依赖的任务完成，只是通知被依赖的任务要完成什么工作，依赖的任务也立即执行，只要自己完成了整个任务就算完成了。至于被依赖的任务最终是否真正完成，依赖它的任务无法确定，所以它是不可靠的任务序列。我们可以用打电话和发短信来很好的比喻同步与异步操作。7.什么是 AIOAIO 是 Java1.7 之后引入的包，是 NIO 的升级版本，提供了异步非堵塞的 IO 操作方式，所以人们叫它 AIO（AsynchronousIO），异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作 8.什么是 NIO 是 Java1.4 引入的 java.nio 包，提供了 Channel、Selector、Buffer 等新的抽象，可以构建多路复用的、同步非阻塞 IO 程序，同时提供了更接近操作系统底层高性能的数据操作方式。9.什么是 BIOBIO 就是传统的 java.io 包，它是基于流模型实现的，交互的方式是同步、阻塞方式，也就是说在读入输入流或者输出流时，在读写动作完成之前，线程会一直阻塞在那里，它们之间的调用时可靠的线性顺序。它的有点就是代码比较简单、直观；缺点就是 IO 的效率和扩展性很低，容易成为应用性能瓶颈。10.流一般需要不需要关闭,如果关闭的话在用什么方法,一般要在那个代码块里面关闭比较好，处理流是怎么关闭的，如果有多个流互相调用传入是怎么关闭的？"流一旦打开就必须关闭，使用 close 方法放入 finally 语句块中（finally 语句一定会执行）调用的处理流就关闭处理流多个流互相调用只关闭最外层的流"11.什么是节点流,什么是处理流,它们各有什么用处,处理流的创建有什么特征？"节点流直接与数据源相连，用于输入或者输出处理流：在节点流的基础上对之进行加工，进行一些功能的扩展处理流的构造器必须要传入节点流的子类"12.PrintStream、BufferedWriter、PrintWriter 的比较?"PrintStream 类的输出功能非常强大，通常如果需要输出文本内容，都应该将输出流包装成 PrintStream 后进行输出。它还提供其他两项功能。与其他输出流不同，PrintStream 永远不会抛出 IOException；而是，异常情况仅设置可通过 checkError 方法测试的内部标志。另外，为了自动刷新，可以创建一个 PrintStreamBufferedWriter:将文本写入字符输出流，缓冲各个字符从而提供单个字符，数组和字符串的高效写入。通过 write()方法可以将获取到的字符输出，然后通过 newLine()进行换行操作。BufferedWriter 中的字符流必须通过调用 flush 方法才能将其刷出去。并且 BufferedWriter 只能对字符流进行操作。如果要对字节流操作，则使用 BufferedInputStreamPrintWriter 的 println 方法自动添加换行，不会抛异常，若关心异常，需要调用 checkError 方法看是否有异常发生，PrintWriter 构造方法可指定参数，实现自动刷新缓存（autoflush）"13.字节流和字符流的区别？字节流读取的时候，读到一个字节就返回一个字节；字符流使用了字节流读到一个或多个字节（中文对应的字节数是两个，在 UTF-8 码表中是 3 个字节）时。先去查指定的编码表，将查到的字符返回。字节流可以处理所有类型数据，如：图片，MP3，AVI 视频文件，而字符流只能处理字符数据。只要是处理纯文本数据，就要优先考虑使用字符流，除此之外都用字节流。字节流主要是操作 byte 类型数据，以 byte 数组为准，主要操作类就是 OutputStream、InputStream 字符流处理的单元为 2 个字节的 Unicode 字符，分别操作字符、字符数组或字符串，而字节流处理单元为 1 个字节，操作字节和字节数组。所以字符流是由 Java 虚拟机将字节转化为 2 个字节的 Unicode 字符为单位的字符而成的，所以它对多国语言支持性比较好！如果是音频文件、图片、歌曲，就用字节流好点，如果是关系到中文（文本）的，用字符流好点。在程序中一个字符等于两个字节，java 提供了 Reader、Writer 两个专门操作字符流的类。14.如何实现 java 序列化？序列化的实现，将需要被序列化的类实现 Serializable 接口，该接口没有需要实现的方法，implementsSerializable 只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个 ObjectOutputStream(对象流)对象，接着，使用 ObjectOutputStream 对象的

writeObject(Objectobj)方法就可以将参数为 obj 的对象写出(即保存其状态)，要恢复的话则用输入流。15.什么是 java 序列化？序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题 16.Java 中有几种类型的流？"（1）按照流的方向：输入流（inputStream）和输出流（outputStream）；（2）按照实现功能分：节点流（可以从或向一个特定的地方（节点）读写数据。如 FileReader）和处理流（是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。如 BufferedReader。处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接）；（3）按照处理数据的单位：字节流和字符流。字节流继承于 InputStream 和 OutputStrea，字符流继承于 InputStreamReader 和 OutputStreamWriter。"三、Java 虚拟机 1.如何判断一个常量是废弃常量？运行时常量池主要回收的是废弃的常量。假如在常量池中存在字符串"abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量"abc"就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc"就会被系统清理出常量池。2.程序计数器为什么是私有的?"程序计数器主要有下面两个作用：字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。"3.JRE、JDK、JVM 及 JIT 之间有什么不同？JRE 代表 Java 运行时（Javarun-time），是运行 Java 引用所必须的。JDK 代表 Java 开发工具（Javadevelopmentkit），是 Java 程序的开发工具，如 Java 编译器，它也包含 JRE。JVM 代表 Java 虚拟机（Javavirtualmachine），它的责任是运行 Java 应用。JIT 代表即时编译（JustInTimecompilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。4.JVM 调优命令有哪些？"jps，JVMProcessStatusTool,显示指定系统内所有的 HotSpot 虚拟机进程。jstat，JVMstatisticsMonitoring 是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据。jmap，JVMMemoryMap 命令用于生成 heapdump 文件 jhat，JVMHeapAnalysisTool 命令是与 jmap 搭配使用，用来分析 jmap 生成的 dump，jhat 内置了一个微型的 HTTP/HTML 服务器，生成 dump 的分析结果后，可以在浏览器中查看 jstack，用于生成 java 虚拟机当前时刻的线程快照。jinfo，JVMConfigurationinfo 这个命令作用是实时查看和调整虚拟机运行参数。"5.说一下 JVM 调优的工具？"常用调优工具分为两类,jdk 自带监控工具：jconsole 和 jvisualvm，第三方有：MAT(MemoryAnalyzerTool)、GChisto。jconsole，JavaMonitoringandManagementConsole 是从 java5 开始，在 JDK 中自带的 java 监控和管理控制台，用于对 JVM 中内存，线程和类等的监控。jvisualvm，jdk 自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC 变化等。MAT，MemoryAnalyzerTool，一个基于 Eclipse 的内存分析工具，是一个快速、功能丰富的 Javaheap 分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。GChisto，一款专业分析 gc 日志的工具。"6.介绍一下类文件结构吧！"魔数:确定这个文件是否为一个能被虚拟机接收的 Class 文件。Class 文件版本：Class 文件的版本号，保证编译正常执行。常量池：常量池主要存放两大常量：字面量和符号引用。访问标志：标志用于识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口，是否为 public 或者 abstract 类型，如果是类的话是否声明为 final 等等。当前类索引,父类索引：类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名，由于 Java 语言的单继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0。接口索引集合：接口索引集合用来描述这个类实现了那些接口，这些被实现的接口将按 implents(如果这个类本身是接口的话则是 extends)后的接口顺序从左到右排列在接口索引集合中。字段表集合：描述接口或类中声明的变量、字段包括类级变量以及实例变量，但不包括在方法内部声明的局部变量。方法表集合：类中的方法。属性表集合：在 Class 文件，字段表，方法表中都可以携带自己的属性表集合。"7.如何判断一个类是无用的类?"方法区主要回收的是无用的类，判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。加载该类的 ClassLoader 已经被回收。该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。"8.Java 会存在内存泄漏吗？请简单描述。"内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说，Java 是有 GC 垃圾回收机制的，也就是说，不再被使用的对象，会被 GC 自动回收掉，自动从内存中清除但是，即使这样，Java 也还是存在着内存泄漏的情况，java 导致内存泄露的原因很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是 java 中内存泄露的发生场景。"9.MinorGc 和 FullGC 有什么不同呢？"大多数情况下，对象在新生代中 eden 区分配。当 eden 区没有足够空间进行分配时，虚拟机将发起一次 MinorGC。新生代 GC（MinorGC）:指发生新生代的的垃圾收集动作，MinorGC 非常频繁，回收速度一般也比较快。老年代 GC（MajorGC/FullGC）:指发生在老年代的 GC，出现了 MajorGC 经常会伴随至少一次的 MinorGC（并非绝对），MajorGC 的速度一般会比 MinorGC 的慢 10 倍以上。"10.说一下堆内存中对象的分配的基本策略"eden 区、s0 区、s1 区都属于新生代，tentired 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 XX:MaxTenuringThreshold 来设置。另外，大对象和长期存活的对象会直接进入老年代。"11.对象的访问定位有哪几种方式?"建立对象就是为了使用对象，我们的 Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有使用句柄和直接指针 2 种：句柄：如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。直接指针：如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象的地址。这两种对象访问方式各有优势。使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。"12.说一下 Java 对象的创建过程"1）类加载检查：虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。2）分配内存：在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。选择以上 2 种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是""标记-清除""，还是""标记-整理""（也称作""标记-压缩""），值得注意的是，复制算法内存也是规整的。在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：CAS+失败重试：CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。TLAB：为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配 3）初始化零值：内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。4）设置对象头：初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等信息。这些信息存放在对象头中。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。5）执行 init 方法：在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生

了，但从 Java 程序的视角来看，对象创建才刚开始，init 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 init 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。"13.Java8 为什么要将永久代(PermGen)替换为元空间(MetaSpace)呢？整个永久代有一个 JVM 本身设置固定大小上线，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，并且永远不会出现 java.lang.OutOfMemoryError。你可以使用-XX：MaxMetaspaceSize 标志设置最大元空间大小，默认值为 unlimited，这意味着它只受系统内存的限制。-XX：MetaspaceSize 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。14.说一下堆和栈的区别"1）物理地址堆的物理地址分配对对象是不连续的。因此性能慢些。在 GC 的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩）栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。2）内存分别堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。3）存放的内容堆存放的是对象的实例和数组。因此该区更关注的是数据的存储栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。4）程序的可见度堆对于整个应用程序都是共享、可见的。栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。"15.怎么打破双亲委派模型？打破双亲委派机制则不仅要继承 ClassLoader 类，还要重写 loadClass 和 findClass 方法。16.为什么需要双亲委派模式？在这里，先想一下，如果没有双亲委派，那么用户是不是可以自己定义一个 java.lang.Object 的同名类，java.lang.String 的同名类，并把它放到 ClassPath 中,那么类之间的比较结果及类的唯一性将无法保证，因此，为什么需要双亲委派模型？防止内存中出现多份同样的字节码。17.怎么打出线程栈信息？"输入 jps，获得进程号。top-Hppid 获取本进程中所有线程的 CPU 耗时性能 jstackpid 命令查看当前 java 进程的堆栈状态 jstack-l>/tmp/output.txt 把堆栈信息打到一个 txt 文件，可以使用 fastthread 堆栈定位（fastthread.io）"18.说说你知道的几种主要的 JVM 参数"1）堆栈配置相关-Xmx3550m：最大堆大小为 3550m。-Xms3550m：设置初始堆大小为 3550m。-Xmn2g：设置年轻代大小为 2g。-Xss128k：每个线程的堆栈大小为 128k。-XX:MaxPermSize：设置持久代大小为 16m-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。2）垃圾收集器相关-XX:+UseParallelGC：选择垃圾收集器为并行收集器。-XX:ParallelGCThreads=20：配置并行收集器的线程数-XX:+UseConcMarkSweepGC：设置年老代为并发收集。-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片 3）辅助信息相关-XX:+PrintGC 输出形式:[GC118250K->113543K(130112K),0.0094143secs][FullGC121376K>10414K(130112K),0.0650971secs]-XX:+PrintGCDetails 输出形式:[GC[DefNew:8614K->781K(9088K),0.0123035secs]118250K->113543K(130112K),0.0124633secs][GC[DefNew:8614K->8614K(9088K),0.0000665secs][Tenured:112761K->10414K(121024K),0.0433488secs]121376K->10414K(130112K),0.0436268secs"19.什么是 happen-before 原则？"单线程 happen-before 原则：在同一个线程中，书写在前面的操作 happen-before 后面的操作。锁的 happen-before 原则：同一个锁的 unlock 操作 happen-before 此锁的 lock 操作。volatile 的 happen-before 原则：对一个 volatile 变量的写操作 happen-before 对此变量的任意操作(当然也包括写操作了)。happen-before 的传递性原则：如果 A 操作 happen-beforeB 操作，B 操作 happen-beforeC 操作，那么 A 操作 happen-beforeC 操作。线程启动的 happen-before 原则：同一个线程的 start 方法 happen-before 此线程的其它方法。线程中断的 happen-before 原则：对线程 interrupt 方法的调用 happen-before 被中断线程的检测到中断发送的代码。线程终结的 happen-before 原则：线程中的所有操作都 happen-before 线程的终止检测。对象创建的 happen-before 原则：一个对象的初始化完成先于他的 finalize 方法调用。"20.什么是内存屏障？"内存屏障，也叫内存栅栏，是一种 CPU 指令，用于控制特定条件下的重排序和内存可见性问题。LoadLoad 屏障：对于这样的语句 Load1;LoadLoad;Load2，在 Load2 及后续读取操作要读取的数据被访问前，保证 Load1 要读取的数据被读取完毕。StoreStore 屏障：对于这样的语句 Store1;StoreStore;Store2，在 Store2 及后续写入操作执行前，保证 Store1 的写入操作对其它处理器可见。LoadStore 屏障：对于这样的语句 Load1;LoadStore;Store2，在 Store2 及后续写入操作被刷出前，保证 Load1 要读取的数据被读取完毕。StoreLoad 屏障：对于这样的语句 Store1;StoreLoad;Load2，在 Load2 及后续所有读取操作执行前，保证 Store1 的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。"21.什么是指令重排序？在实际运行时，代码指令可能并不是严格按照代码语句顺序执行的。大多数现代微处理器都会采用将指令乱序执行（out-of-orderexecution，简称 OoOE 或 OOE）的方法，在条件允许的情况下，直接运行当前有能力立即执行的后续指令，避开获取下一条指令所需数据时造成的等待。通过乱序执行的技术，处理器可以大大提高执行效率。而这就是指令重排。22.JVM 中一次完整的 GC 流程是怎样的，对象如何晋升到老年代？"当 Eden 区的空间满了，Java 虚拟机会触发一次 MinorGC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor 区。大对象（需要大量连续内存空间的 Java 对象，如那种很长的字符串）直接进入老年态；如果对象在 Eden 出生，并经过第一次 MinorGC 后仍然存活，并且被 Survivor 容纳的话，年龄设为 1，每熬过一次 MinorGC，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。即长期存活的对象进入老年态。老年代满了而无法容纳更多的对象，MinorGC 之后通常就会进行 FullGC，FullGC 清理整个内存堆–包括年轻代和年老代。MajorGC 发生在老年代的 GC，清理老年区，经常会伴随至少一次 MinorGC，比 MinorGC 慢 10 倍以上。"23.JVM 新生代中为什么要分为 Eden 和 Survivor？"如果没有 Survivor，Eden 区每进行一次 MinorGC，存活的对象就会被送到老年代。老年代很快被填满，触发 MajorGC.老年代的内存空间远大于新生代，进行一次 FullGC 消耗的时间比 MinorGC 长得多,所以需要分为 Eden 和 Survivor。Survivor 的存在意义，就是减少被送到老年代的对象，进而减少 FullGC 的发生，Survivor 的预筛选保证，只有经历 16 次 MinorGC 还能在新生代中存活的对象，才会被送到老年代。设置两个 Survivor 区最大的好处就是解决了碎片化，刚刚新建的对象在 Eden 中，经历一次 MinorGC，Eden 中的存活对象就会被移动到第一块 survivorspaceS0，Eden 被清空；等 Eden 区再满了，就再触发一次 MinorGC，Eden 和 S0 中的存活对象又会被复制送入第二块 survivorspaceS1（这个过程非常重要，因为这种复制算法保证了 S1 中来自 S0 和 Eden 两部分的存活对象占用连续的内存空间，避免了碎片化的发生）"24.什么情况下会发生栈内存溢出？"栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型。如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常，方法递归调用产生这种结果。如果 Java 虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么 Java 虚拟机将抛出一个 OutOfMemory 异常。(线程启动过多)。"25.Java 对象的布局了解过吗？"对象头区域此处存储的信息包括两部分：1、对象自身的运行时数据(MarkWord)，占 8 字节存储 hashCode、GC 分代年龄、锁类型标记、偏向锁线程 ID、CAS 锁指向线程 LockRecord 的指针等，synconized 锁的机制与这个部分(markwork)密切相关，用 markword 中最低的三位代表锁的状态，其中一位是偏向锁位，另外两位是普通锁位。2、对象类型指针(ClassPointer)，占 4 字节对象指向它的类元数据的指针、JVM 就是通过它来确定是哪个 Class 的实例。实例数据区域此处存储的是对象真正有效的信息，比如对象中所有字段的内容对齐填充区域 JVM 的实现 HostSpot 规定对象的起始地址必须是 8 字节的整数倍，换句话来说，现在 64 位的 OS 往外读取数据的时候一次性读取 64bit 整数倍的数据，也就是 8 个字节，所以 HotSpot 为了高效读取对象，就做了""对齐""，如果一个对象实际占的内存大小不是 8byte 的整数倍时，就""补位""到 8byte 的整数倍。所以对齐填充区域的大小不是固定的。"26.Tomcat 是怎么打破双亲委派机制的呢？是通过重写 ClassLoader#loadClass 和 ClassLoader#findClass 实现的。可以看图中的 WebAppClassLoader，它加载自己目录下的.class 文件，并不会传递给父类的加载器。但是，它却可以使用 SharedClassLoader 所加载的类，实现了共享和分离的功能。

27.什么是双亲委派机制？双亲委派机制的意思是除了顶层的启动类加载器以外，其余的类加载器，在加载之前，都会委派给它的父加载器进行加载。这样一层层向上传递，直到祖先们都无法胜任，它才会真正的加载。28.说下有哪些类加载器？"BootstrapClassLoader（启动类加载器）ExtentionClassLoader（扩展类加载器）AppClassLoader（应用类加载器）"29.说说类加载的过程"加载验证准备（为一些类变量分配内存，并将其初始化为默认值）解析（将符号引用替换为直接引用。类和接口、类方法、接口方法、字段等解析）初始化"30.ZGC 收集器中的染色指针有什么用？"染色指针是一种直接将少量额外的信息存储在指针上的技术，可是为什么指针本身也可以存储额外信息呢？在 64 位系统中，理论可以访问的内存高达 16EB（2 的 64 次幂）字节[3]。实际上，基于需求（用不到那么多内存）、性能（地址越宽在做地址转换时需要的页表级数越多）和成本（消耗更多晶体管）的考虑，在 AMD64 架构[4]中只支持到 52 位（4PB）的地址总线和 48 位（256TB）的虚拟地址空间，所以目前 64 位的硬件实际能够支持的最大内存只有 256TB。此外，操作系统一侧也还会施加自己的约束，64 位的 Linux 则分别支持 47 位（128TB）的进程虚拟地址空间和 46 位（64TB）的物理地址空间，64 位的 Windows 系统甚至只支持 44 位（16TB）的物理地址空间。尽管 Linux 下 64 位指针的高 18 位不能用来寻址，但剩余的 46 位指针所能支持的 64TB 内存在今天仍然能够充分满足大型服务器的需要。鉴于此，ZGC 的染色指针技术继续盯上了这剩下的 46 位指针宽度，将其高 4 位提取出来存储四个标志信息。通过这些标志位，虚拟机可以直接从指针中看到其引用对象的三色标记状态、是否进入了重分配集（即被移动过）、是否只能通过 finalize()方法才能被访问到。当然，由于这些标志位进一步压缩了原本就只有 46 位的地址空间，也直接导致 ZGC 能够管理的内存不可以超过 4TB（2 的 42 次幂）。"31.说说 ZGC 垃圾收集器的工作原理"1）内存布局小型 Region（SmallRegion）：容量固定为 2MB，用于放置小于 256KB 的小对象。中型 Region（MediumRegion）：容量固定为 32MB，用于放置大于等于 256KB 但小于 4MB 的对象。大型 Region（LargeRegion）：容量不固定，可以动态变化，但必须为 2MB 的整数倍，用于放置 4MB 或以上的大对象。每个大型 Region 中只会存放一个大对象，这也预示着虽然名字叫作“大型 Region”，但它的实际容量完全有可能小于中型 Region，最小容量可低至 4MB。大型 Region 在 ZGC 的实现中是不会被重分配（重分配是 ZGC 的一种处理动作，用于复制对象的收集器阶段，稍后会介绍到）的，因为复制一个大对象的代价非常高昂。2）染色指针染色指针是一种直接将少量额外的信息存储在指针上的技术，可是为什么指针本身也可以存储额外信息呢？在 64 位系统中，理论可以访问的内存高达 16EB（2 的 64 次幂）字节[3]。实际上，基于需求（用不到那么多内存）、性能（地址越宽在做地址转换时需要的页表级数越多）和成本（消耗更多晶体管）的考虑，在 AMD64 架构[4]中只支持到 52 位（4PB）的地址总线和 48 位（256TB）的虚拟地址空间，所以目前 64 位的硬件实际能够支持的最大内存只有 256TB。此外，操作系统一侧也还会施加自己的约束，64 位的 Linux 则分别支持 47 位（128TB）的进程虚拟地址空间和 46 位（64TB）的物理地址空间，64 位的 Windows 系统甚至只支持 44 位（16TB）的物理地址空间。尽管 Linux 下 64 位指针的高 18 位不能用来寻址，但剩余的 46 位指针所能支持的 64TB 内存在今天仍然能够充分满足大型服务器的需要。鉴于此，ZGC 的染色指针技术继续盯上了这剩下的 46 位指针宽度，将其高 4 位提取出来存储四个标志信息。通过这些标志位，虚拟机可以直接从指针中看到其引用对象的三色标记状态、是否进入了重分配集（即被移动过）、是否只能通过 finalize()方法才能被访问到。当然，由于这些标志位进一步压缩了原本就只有 46 位的地址空间，也直接导致 ZGC 能够管理的内存不可以超过 4TB（2 的 42 次幂）。3）收集过程并发标记（ConcurrentMark）：与 G1、Shenandoah 一样，并发标记是遍历对象图做可达性分析的阶段，前后也要经过类似于 G1、Shenandoah 的初始标记、最终标记（尽管 ZGC 中的名字不叫这些）的短暂停顿，而且这些停顿阶段所做的事情在目标上也是相类似的。与 G1、Shenandoah 不同的是，ZGC 的标记是在指针上而不是在对象上进行的，标记阶段会更新染色指针中的 Marked0、Marked1 标志位。并发预备重分配（ConcurrentPrepareforRelocate）：这个阶段需要根据特定的查询条件统计得出本次收集过程要清理哪些 Region，将这些 Region 组成重分配集（RelocationSet）。重分配集与 G1 收集器的回收集（CollectionSet）还是有区别的，ZGC 划分 Region 的目的并非为了像 G1 那样做收益优先的增量回收。相反，ZGC 每次回收都会扫描所有的 Region，用范围更大的扫描成本换取省去 G1 中记忆集的维护成本。因此，ZGC 的重分配集只是决定了里面的存活对象会被重新复制到其他的 Region 中，里面的 Region 会被释放，而并不能说回收行为就只是针对这个集合里面的 Region 进行，因为标记过程是针对全堆的。此外，在 JDK12 的 ZGC 中开始支持的类卸载以及弱引用的处理，也是在这个阶段中完成的。并发重分配（ConcurrentRelocate）：重分配是 ZGC 执行过程中的核心阶段，这个过程要把重分配集中的存活对象复制到新的 Region 上，并为重分配集中的每个 Region 维护一个转发表（ForwardTable），记录从旧对象到新对象的转向关系。得益于染色指针的支持，ZGC 收集器能仅从引用上就明确得知一个对象是否处于重分配集之中，如果用户线程此时并发访问了位于重分配集中的对象，这次访问将会被预置的内存屏障所截获，然后立即根据 Region 上的转发表记录将访问转发到新复制的对象上，并同时修正更新该引用的值，使其直接指向新对象，ZGC 将这种行为称为指针的“自愈”（Self-Healing）能力。这样做的好处是只有第一次访问旧对象会陷入转发，也就是只慢一次，对比 Shenandoah 的 Brooks 转发指针，那是每次对象访问都必须付出的固定开销，简单地说就是每次都慢，因此 ZGC 对用户程序的运行时负载要比 Shenandoah 来得更低一些。还有另外一个直接的好处是由于染色指针的存在，一旦重分配集中某个 Region 的存活对象都复制完毕后，这个 Region 就可以立即释放用于新对象的分配（但是转发表还得留着不能释放掉），哪怕堆中还有很多指向这个对象的未更新指针也没有关系，这些旧指针一旦被使用，它们都是可以自愈的。并发重映射（ConcurrentRemap）：重映射所做的就是修正整个堆中指向重分配集中旧对象的所有引用，这一点从目标角度看是与 Shenandoah 并发引用更新阶段一样的，但是 ZGC 的并发重映射并不是一个必须要“迫切”去完成的任务，因为前面说过，即使是旧引用，它也是可以自愈的，最多只是第一次使用时多一次转发和修正操作。重映射清理这些旧引用的主要目的是为了不变慢（还有清理结束后可以释放转发表这样的附带收益），所以说这并不是很“迫切”。因此，ZGC 很巧妙地把并发重映射阶段要做的工作，合并到了下一次垃圾收集循环中的并发标记阶段里去完成，反正它们都是要遍历所有对象的，这样合并就节省了一次遍历对象图[9]的开销。一旦所有指针都被修正之后，原来记录新旧对象关系的转发表就可以释放掉了。"32.说说 G1 垃圾收集器的工作原理"优点：指定最大停顿时间、分 Region 的内存布局、按收益动态确定回收集 G1 开创的基于 Region 的堆内存布局是它能够实现这个目标的关键。虽然 G1 也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异：G1 不再坚持固定大小以及固定数量的分代区域划分，而是把连续的 Java 堆划分为多个大小相等的独立区域（Region），每一个 Region 都可以根据需要，扮演新生代的 Eden 空间、Survivor 空间，或者老年代空间。收集器能够对扮演不同角色的 Region 采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果。虽然 G1 仍然保留新生代和老年代的概念，但新生代和老年代不再是固定的了，它们都是一系列区域（不需要连续）的动态集合。G1 收集器之所以能建立可预测的停顿时间模型，是因为它将 Region 作为单次回收的最小单元，即每次收集到的内存空间都是 Region 大小的整数倍，这样可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集。更具体的处理思路是让 G1 收集器去跟踪各个 Region 里面的垃圾堆积的“价值”大小，价值即回收所获得的空间大小以及回收所需时间的经验值，然后在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间（使用参数-XX：MaxGCPauseMillis 指定，默认值是 200 毫秒），优先处理回收价值收益最大的那些 Region，这也就是“GarbageFirst”名字的由来。这种使用 Region 划分内存空间，以及具有优先级的区域回收方式，保证了 G1 收集器在有限的时间内获取尽可能高的收集效率。G1 收集器的运作过程大致可划分为以下四个步骤：·初始标记（InitialMarking）：仅仅只是标记一下 GCRoots 能直接关联到的对象，并且修改 TAMS 指针的值，让下一阶段用户线程并发运行时，能正确地在可用的 Region 中分配新对象。这个阶段需要停顿线程，但耗时很短，而且是借用进行 MinorGC 的时候同步完成的，所以 G1 收集器在这个阶段实际并没有额外的停顿。并发标记（ConcurrentMarking）：从 GCRoot 开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。当对象图扫描完成以后，还要重新处理 SATB 记录下的在并发时有引用变动的对象。最终标记（FinalMarking）：对用户线程做另一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的最后那少量的 SATB 记录。筛选回收（LiveDataCountingandEvacuation）：负责更新 Region 的统计数据，对各个 Region 的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个 Region 构成回收集，然后把决定回收的那一部分 Region 的存活对象复制到空的 Region 中，再清理掉整个旧 Region 的全部空间。这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的。从上述阶段的描述可以看出，G1 收集器除了并发标记外，其余阶段也

是要完全暂停用户线程的。"33.说说 CMS 垃圾收集器的工作原理"Concurrentmarksweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间，和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。CMS 工作机制相比其他的垃圾收集器来说更复杂，整个过程分为以下 4 个阶段：1）初始标记只是标记一下 GCRoots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。2）并发标记进行 GCRoots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。3）重新标记为了修正在并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，仍然需要暂停所有的工作线程。4）并发清除清除 GCRoots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户线程一起并发工作，所以总体上来看 CMS 收集器的内存回收和用户线程是一起并发地执行。"34.你了解过哪些垃圾收集器？"年轻代 Serial 垃圾收集器（单线程，通常用在客户端应用上。因为客户端应用不会频繁创建很多对象，用户也不会感觉出明显的卡顿。相反，它使用的资源更少，也更轻量级。）ParNew 垃圾收集器（多线程，追求降低用户停顿时间，适合交互式应用。）ParallelScavenge 垃圾收集器（追求 CPU 吞吐量，能够在较短时间内完成指定任务，适合没有交互的后台计算。）老年代 SerialOld 垃圾收集器 ParallelOld 垃圾收集器 CMS 垃圾收集器（以获取最短 GC 停顿时间为目标的收集器，它在垃圾收集时使得用户线程和 GC"35.对象都是优先分配在年轻代上的吗？不是。当新生代内存不够时，老年代分配担保。而大对象则是直接在老年代分配。36.GCRoots 有哪些？"GCRoots 是一组必须活跃的引用。用通俗的话来说，就是程序接下来通过直接引用或者间接引用，能够访问到的潜在被使用的对象。GCRoots 包括：Java 线程中，当前所有正在被调用的方法的引用类型参数、局部变量、临时值等。也就是与我们栈帧相关的各种引用。所有当前被加载的 Java 类。Java 类的引用类型静态变量。运行时常量池里的引用类型常量（String 或 Class 类型）。JVM 内部数据结构的一些引用，比如 sun.jvm.hotspot.memory.Universe 类。用于同步的监控对象，比如调用了对象的 wait()方法。JNIhandles，包括 globalhandles 和 localhandles。这些 GCRoots 大体可以分为三大类，下面这种说法更加好记一些：活动线程相关的各种引用。类的静态变量的引用。JNI 引用。有两个注意点：我们这里说的是活跃的引用，而不是对象，对象是不能作为 GCRoots 的。GC 过程是找出所有活对象，并把其余空间认定为“无用”；而不是找出所有死掉的对象，并回收它们占用的空间。所以，哪怕 JVM 的堆非常的大，基于 tracing 的 GC 方式，回收速度也会非常快。"37.JVM 怎么判断一个对象是不是要回收？"引用计数法（缺点是对于相互引用的对象，无法进行清除）可达性分析"38.Java 里有哪些引用类型？"强引用这种引用属于最普通最强硬的一种存在，只有在和 GCRoots 断绝关系时，才会被消灭掉。软引用软引用用于维护一些可有可无的对象。在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。可以看到，这种特性非常适合用在缓存技术上。比如网页缓存、图片缓存等。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。弱引用弱引用对象相比较软引用，要更加无用一些，它拥有更短的生命周期。当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。弱引用拥有更短的生命周期，在 Java 中，用 java.lang.ref.WeakReference 类来表示。它的应用场景和软引用类似，可以在一些对内存更加敏感的系统里采用。虚引用这是一种形同虚设的引用，在现实场景中用的不是很多。虚引用必须和引用队列（ReferenceQueue）联合使用。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。实际上，虚引用的 get，总是返回 null。"39.你熟悉哪些垃圾收集算法？"标记清除（缺点是碎片化）复制算法（缺点是浪费空间）标记整理算法（效率比前两者差）分代收集算法（老年代一般使用“标记-清除”、“标记-整理”算法，年轻代一般用复制算法）"40.字符串常量存放在哪个区域？"字符串常量池，已经移动到堆上（jdk8 之前是 perm 区），也就是执行 intern 方法后存的地方。类文件常量池，constant_pool，是每个类每个接口所拥有的，这部分数据在方法区，也就是元数据区。而运行时常量池是在类加载后的一个内存区域，它们都在元空间。"41.程序计数器有什么作用？"程序计数器是一块较小的内存空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。这里面存的，就是当前线程执行的进度。程序计数器还存储了当前正在运行的流程，包括正在执行的指令、跳转、分支、循环、异常处理等。"42.栈帧里面包含哪些东西？局部变量表、操作数栈、动态连接、返回地址等

四、Kafka1.简述 Follower 副本消息同步的完整流程"首先，Follower 发送 FETCH 请求给 Leader。接着，Leader 会读取底层日志文件中的消息数据，再更新它内存中的 Follower 副本的 LEO 值，更新为 FETCH 请求中的 fetchOffset 值。最后，尝试更新分区高水位值。Follower 接收到 FETCH 响应之后，会把消息写入到底层日志，接着更新 LEO 和 HW 值。Leader 和 Follower 的 HW 值更新时机是不同的，Follower 的 HW 更新永远落后于 Leader 的 HW。这种时间上的错配是造成各种不一致的原因。"2.JavaConsumer 为什么采用单线程来获取消息?"在回答之前，如果先把这句话说出来，一定会加分:JavaConsumer 是双线程的设计。一个线程是用户主线程，负责获取消息;另一个线程是心跳线程，负责向 Kafka 汇报消费者存活情况。将心跳单独放入专属的线程，能够有效地规避因消息处理速度慢而被视为下线的“假死”情况。单线程获取消息的设计能够避免阻塞式的消息获取方式。单线程轮询方式容易实现异步非阻塞式，这样便于将消费者扩展成支持实时流处理的操作算子。因为很多实时流处理操作算子都不能是阻塞式的。另外一个可能的好处是，可以简化代码的开发。多线程交互的代码是非常容易出错的。"3.Controller 发生网络分区(NetworkPartitioning)时，Kafka 会怎么样?"这道题目能够诱发我们对分布式系统设计、CAP 理论、一致性等多方面的思考。不过，针对故障定位和分析的这类问题，我建议你首先言明“实用至上”的观点，即不论怎么进行理论分析，永远都要以实际结果为准。一旦发生 Controller 网络分区，那么，第一要务就是查看集群是否出现“脑裂”，即同时出现两个甚至是多个 Controller 组件。这可以根据 Broker 端监控指标 ActiveControllerCount 来判断。现在，我们分析下，一旦出现这种情况，Kafka 会怎么样。由于 Controller 会给 Broker 发送 3 类请求，即 LeaderAndIsrRequest、StopReplicaRequest 和 UpdateMetadataRequest，因此，一旦出现网络分区，这些请求将不能顺利到达 Broker 端。这将影响主题的创建、修改、删除操作的信息同步，表现为集群仿佛僵住了一样，无法感知到后面的所有操作。因此，网络分区通常都是非常严重的问题，要赶快修复。"4.Kafka 的哪些场景中使用了零拷贝(ZeroCopy)?"ZeroCopy 是特别容易被问到的高阶题目。在 Kafka 中，体现 ZeroCopy 使用场景的地方有两处:基于 mmap 的索引和日志文件读写所用的 TransportLayer。先说第一个。索引都是基于 MappedByteBuffer 的，也就是让用户态和内核态共享内核态的数据缓冲区，此时，数据不需要复制到用户态空间。不过，mmap 虽然避免了不必要的拷贝，但不一定就能保证很高的性能。在不同的操作系统下，mmap 的创建和销毁成本可能是不一样的。很高的创建和销毁开销会抵消 ZeroCopy 带来的性能优势。由于这种不确定性，在 Kafka 中，只有索引应用了 mmap，最核心的日志并未使用 mmap 机制。再说第二个。TransportLayer 是 Kafka 传输层的接口。它的某个实现类使用了 FileChannel 的 transferTo 方法。该方法底层使用 sendfile 实现了 ZeroCopy。对 Kafka 而言，如果 I/O 通道使用普通的 PLAINTEXT，那么，Kafka 就可以利用 ZeroCopy 特性，直接将页缓存中的数据发送到网卡的 Buffer 中，避免中间的多次拷贝。相反，如果 I/O 通道启用了 SSL，那么，Kafka 便无法利用 ZeroCopy 特性了。"5.分区 Leader 选举策略有几种?"分区的 Leader 副本选举对用户是完全透明的，它是由 Controller 独立完成的。你需要回答的是，在哪些场景下，需要执行分区 Leader 选举。每一种场景对应于一种选举策略。当前，Kafka 有 4 种分区 Leader 选举策略。OfflinePartitionLeader 选举：每当有分区上线时，就需要执行 Leader 选举。所谓的分区上线，可能是创建了新分区，也可能是之前的下线分区重新上线。这是最常见的分区 Leader 选举场景。ReassignPartitionLeader 选举：当你手动运行 kafka-reassign-partitions 命令，或者是调用 Admin 的 alterPartitionReassignments 方法执行分区副本重分配时，可能触发此类选举。假设原来的 AR 是[1，2，3]，Leader 是 1，当执行副本重分配后，副本集合 AR 被设置成[4，5，6]，显然，Leader 必须要变更，此时会发生 ReassignPartitionLeader 选举。PreferredReplicaPartitionLeader 选举：当你手动运行 kafka-preferred-replicaelection 命令，或自动触发了 PreferredLeader 选举时，该类策略被激活。所谓的 PreferredLeader，指的是 AR 中的第一个副本。比如 AR 是[3，2，1]，那么，PreferredLeader 就是 3。ControlledShutdownPartitionLeader 选举：当 Broker 正常关闭时，该 Broker 上的所有 Leader 副本都会下线，因此，需要为受影响的分区执行相应的 Leader 选举。这 4 类选举策略的大致思想是类似的，即从 AR 中挑选首个在 ISR 中的副本，作为新 Leader。当然，个别策略有些微小差异。不过，回答到这种程度，应该足以应付面试官了。毕竟，微小差别对选举 Leader 这件事的影响很小。"6.consumer_offsets 是做什么用的?"这是一个内部主题，公开的官网资料很少涉及到。因此，我认为，此题属于面试官炫技一类的题目。你要小心这里的考点:该主题有 3 个重要的知识点，你一定要全部答

出来，才会显得对这块知识非常熟悉。它是一个内部主题，无需手动干预，由 Kafka 自行管理。当然，我们可以创建该主题。它的主要作用是负责注册消费者以及保存位移值。可能你对保存位移值的功能很熟悉，但其实该主题也是保存消费者元数据的地方。千万记得把这一点也回答上。另外，这里的消费者泛指消费者组和独立消费者，而不仅仅是消费者组。Kafka 的 GroupCoordinator 组件提供对该主题完整的管理功能，包括该主题的创建、写入、读取和 Leader 维护等。"7.Kafka 能手动删除消息吗?"其实，Kafka 不需要用户手动删除消息。它本身提供了留存策略，能够自动删除过期消息。当然，它是支持手动删除消息的。因此，你最好从这两个维度去回答。对于设置了 Key 且参数 cleanup.policy=compact 的主题而言，我们可以构造一条的消息发送给 Broker，依靠 LogCleaner 组件提供的功能删除掉该 Key 的消息。对于普通主题而言，我们可以使用 kafka-delete-records 命令，或编写程序调用 Admin.deleteRecords 方法来删除消息。这两种方法殊途同归，底层都是调用 Admin 的 deleteRecords 方法，通过将分区 LogStartOffset 值抬高的方式间接删除消息。"8.LEO、LSO、AR、ISR、HW 都表示什么含义?"LEO:LogEndOffset。日志末端位移值或末端偏移量，表示日志下一条待插入消息的位移值。举个例子，如果日志有 10 条消息，位移值从 0 开始，那么，第 10 条消息的位移值就是 9。此时，LEO=10。LSO:LogStableOffset。这是 Kafka 事务的概念。如果你没有使用到事务，那么这个值不存在(其实也不是不存在，只是设置成一个无意义的值)。该值控制了事务型消费者能够看到的消息范围。它经常与 LogStartOffset，即日志起始位移值相混淆，因为有些人将后者缩写成 LSO，这是不对的。在 Kafka 中，LSO 就是指代 LogStableOffset。AR:AssignedReplicas。AR 是主题被创建后，分区创建时被分配的副本集合，副本个数由副本因子决定。ISR:In-SyncReplicas。Kafka 中特别重要的概念，指代的是 AR 中那些与 Leader 保持同步的副本集合。在 AR 中的副本可能不在 ISR 中，但 Leader 副本天然就包含在 ISR 中。关于 ISR，还有一个常见的面试题目是如何判断副本是否应该属于 ISR。目前的判断依据是:Follower 副本的 LEO 落后 LeaderLEO 的时间，是否超过了 Broker 端参数 replica.lag.time.max.ms 值。如果超过了，副本就会被从 ISR 中移除。HW:高水位值(Highwatermark)。这是控制消费者可读取消息范围的重要字段。一个普通消费者只能“看到”Leader 副本上介于 LogStartOffset 和 HW(不含)之间的所有消息。水位以上的消息是对消费者不可见的。关于 HW，问法有很多，我能想到的最高级的问法，就是让你完整地梳理下 Follower 副本拉取 Leader 副本、执行同步机制的详细步骤。"9.Leader 总是-1，怎么破?"在生产环境中，你一定碰到过“某个主题分区不能工作了”的情形。使用命令行查看状态的话，会发现 Leader 是-1，于是，你使用各种命令都无济于事，最后只能用“重启大法”。但是，有没有什么办法，可以不重启集群，就能解决此事呢?这就是此题的由来。我直接给答案:删除 ZooKeeper 节点/controller，触发 Controller 重选举。Controller 重选举能够为所有主题分区重刷分区状态，可以有效解决因不一致导致的 Leader 不可用问题。我几乎可以断定，当面试官问出此题时，要么就是他真的不知道怎么解决在向你寻求答案，要么他就是在等你说出这个答案。所以，千万别一上来就说“来个重启”之类的话。"10.如何估算 Kafka 集群的机器数量?"这道题目考查的是机器数量和所用资源之间的关联关系。所谓资源，也就是 CPU、内存、磁盘和带宽。通常来说，CPU 和内存资源的充足是比较容易保证的，因此，你需要从磁盘空间和带宽占用两个维度去评估机器数量。在预估磁盘的占用时，你一定不要忘记计算副本同步的开销。如果一条消息占用 1KB 的磁盘空间，那么，在有 3 个副本的主题中，你就需要 3KB 的总空间来保存这条消息。显式地将这些考虑因素答出来，能够彰显你考虑问题的全面性，是一个难得的加分项。对于评估带宽来说，常见的带宽有 1Gbps 和 10Gbps，但你要切记，这两个数字仅仅是最大值。因此，你最好和面试官确认一下给定的带宽是多少。然后，明确阐述出当带宽占用接近总带宽的 90%时，丢包情形就会发生。这样能显示出你的网络基本功。"11.Broker 的 HeapSize 如何设置？"如何设置 HeapSize 的问题，其实和 Kafka 关系不大，它是一类非常通用的面试题目。一旦你应对不当，面试方向很有可能被引到 JVM 和 GC 上去，那样的话，你被问住的几率就会增大。因此，我建议你简单地介绍一下 HeapSize 的设置方法，并把重点放在 KafkaBroker 堆大小设置的最佳实践上。比如，你可以这样回复:任何 Java 进程 JVM 堆大小的设置都需要仔细地进行考量和测试。一个常见的做法是，以默认的初始 JVM 堆大小运行程序，当系统达到稳定状态后，手动触发一次 FullGC，然后通过 JVM 工具查看 GC 后的存活对象大小。之后，将堆大小设置成存活对象总大小的 1.5~2 倍。对于 Kafka 而言，这个方法也是适用的。不过，业界有个最佳实践，那就是将 Broker 的 HeapSize 固定为 6GB。经过很多公司的验证，这个大小是足够且良好的。"12.监控 Kafka 的框架都有哪些?"这道题表面上是考核你对 Leader 和 Follower 区别的理解，但很容易引申到 Kafka 的同步机制上。因此，我建议你主动出击，一次性地把隐含的考点也答出来，也许能够暂时把面试官“唬住”，并体现你的专业性。你可以这么回答:Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时准备应聘 Leader 副本。通常来说，回答到这个程度，其实才只说了 60%，因此，我建议你再回答两个额外的加分项。强调 Follower 副本也能对外提供读服务。自 Kafka2.4 版本开始，社区通过引入新的 Broker 端参数，允许 Follower 副本有限度地提供读服务。强调 Leader 和 Follower 的消息序列在实际场景中不一致。很多原因都可能造成 Leader 和 Follower 保存的消息序列不一致，比如程序 Bug、网络问题等。这是很严重的错误，必须要完全规避。你可以补充下，之前确保一致性的主要手段是高水位机制，但高水位值无法保证 Leader 连续变更场景下的数据一致性，因此，社区引入了 LeaderEpoch 机制，来修复高水位值的弊端。关于“LeaderEpoch 机制”，国内的资料不是很多，它的普及度远不如高水位，不妨大胆地把这个概念秀出来，力求惊艳一把。"13.如何设置 Kafka 能接收的最大消息的大小?"这道题表面上是考核你对 Leader 和 Follower 区别的理解，但很容易引申到 Kafka 的同步机制上。因此，我建议你主动出击，一次性地把隐含的考点也答出来，也许能够暂时把面试官“唬住”，并体现你的专业性。你可以这么回答:Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时准备应聘 Leader 副本。通常来说，回答到这个程度，其实才只说了 60%，因此，我建议你再回答两个额外的加分项。强调 Follower 副本也能对外提供读服务。自 Kafka2.4 版本开始，社区通过引入新的 Broker 端参数，允许 Follower 副本有限度地提供读服务。强调 Leader 和 Follower 的消息序列在实际场景中不一致。很多原因都可能造成 Leader 和 Follower 保存的消息序列不一致，比如程序 Bug、网络问题等。这是很严重的错误，必须要完全规避。你可以补充下，之前确保一致性的主要手段是高水位机制，但高水位值无法保证 Leader 连续变更场景下的数据一致性，因此，社区引入了 LeaderEpoch 机制，来修复高水位值的弊端。关于“LeaderEpoch 机制”，国内的资料不是很多，它的普及度远不如高水位，不妨大胆地把这个概念秀出来，力求惊艳一把。"14.阐述下 Kafka 中的领导者副本(LeaderReplica)和追随者副本(FollowerReplica)的区别"这道题表面上是考核你对 Leader 和 Follower 区别的理解，但很容易引申到 Kafka 的同步机制上。因此，我建议你主动出击，一次性地把隐含的考点也答出来，也许能够暂时把面试官“唬住”，并体现你的专业性。你可以这么回答:Kafka 副本当前分为领导者副本和追随者副本。只有 Leader 副本才能对外提供读写服务，响应 Clients 端的请求。Follower 副本只是采用拉(PULL)的方式，被动地同步 Leader 副本中的数据，并且在 Leader 副本所在的 Broker 宕机后，随时准备应聘 Leader 副本。通常来说，回答到这个程度，其实才只说了 60%，因此，我建议你再回答两个额外的加分项。强调 Follower 副本也能对外提供读服务。自 Kafka2.4 版本开始，社区通过引入新的 Broker 端参数，允许 Follower 副本有限度地提供读服务。强调 Leader 和 Follower 的消息序列在实际场景中不一致。很多原因都可能造成 Leader 和 Follower 保存的消息序列不一致，比如程序 Bug、网络问题等。这是很严重的错误，必须要完全规避。你可以补充下，之前确保一致性的主要手段是高水位机制，但高水位值无法保证 Leader 连续变更场景下的数据一致性，因此，社区引入了 LeaderEpoch 机制，来修复高水位值的弊端。关于“LeaderEpoch 机制”，国内的资料不是很多，它的普及度远不如高水位，不妨大胆地把这个概念秀出来，力求惊艳一把。"15.解释下 Kafka 中位移(offset)的作用"在 Kafka 中，每个主题分区下的每条消息都被赋予了一个唯一的 ID 数值，用于标识它在分区中的位置。这个 ID 数值，就被称为位移，或者叫偏移量。一旦消息被写入到分区日志，它的位移值将不能被修改。答完这些之后，你还可以把整个面试方向转移到你希望的地方。常见方法有以下 3 种:如果你深谙 Broker 底层日志写入的逻辑，可以强调下消息在日志中的存放格式;如果你明白位移值一旦被确定不能修改，可以强调下“LogCleaner 组件都不能影响位移值”这件事情;如果你对消费者的概念还算熟悉，可以再详细说说位移值和消费者位移值之间的区别。"16.什么是消费者组?"Kafka 并没有使用 JDK 自带的 Timer 或者 DelayQueue 来实现延迟的功能，而是基于时间轮自定义了一个用于实现延迟功能的定时器（SystemTimer）。JDK 的 Timer 和 DelayQueue 插入和删除操作的平均时间复杂度为 O(nlog(n))，并不能满足 Kafka 的高性能要求，而基于

时间轮可以将插入和删除操作的时间复杂度都降为 O(1)。时间轮的应用并非 Kafka 独有，其应用场景还有很多，在 Netty、Akka、Quartz、Zookeeper 等组件中都存在时间轮的踪影。底层使用数组实现，数组中的每个元素可以存放一个 TimerTaskList 对象。TimerTaskList 是一个环形双向链表，在其中的链表项 TimerTaskEntry 中封装了真正的定时任务 TimerTask.Kafka 中到底是怎么推进时间的呢？Kafka 中的定时器借助了 JDK 中的 DelayQueue 来协助推进时间轮。具体做法是对于每个使用到的 TimerTaskList 都会加入到 DelayQueue 中。Kafka 中的 TimingWheel 专门用来执行插入和删除 TimerTaskEntry 的操作，而 DelayQueue 专门负责时间推进的任务。再试想一下，DelayQueue 中的第一个超时任务列表的 expiration 为 200ms，第二个超时任务为 840ms，这里获取 DelayQueue 的队头只需要 O(1)的时间复杂度。如果采用每秒定时推进，那么获取到第一个超时的任务列表时执行的 200 次推进中有 199 次属于“空推进”，而获取到第二个超时任务时有需要执行 639 次“空推进”，这样会无故空耗机器的性能资源，这里采用 DelayQueue 来辅助以少量空间换时间，从而做到了“精准推进”。Kafka 中的定时器真可谓是“知人善用”，用 TimingWheel 做最擅长的任务添加和删除操作，而用 DelayQueue 做最擅长的时间推进工作，相辅相成。"17.kafka 如何实现延迟队列？"Kafka 并没有使用 JDK 自带的 Timer 或者 DelayQueue 来实现延迟的功能，而是基于时间轮自定义了一个用于实现延迟功能的定时器（SystemTimer）。JDK 的 Timer 和 DelayQueue 插入和删除操作的平均时间复杂度为 O(nlog(n))，并不能满足 Kafka 的高性能要求，而基于时间轮可以将插入和删除操作的时间复杂度都降为 O(1)。时间轮的应用并非 Kafka 独有，其应用场景还有很多，在 Netty、Akka、Quartz、Zookeeper 等组件中都存在时间轮的踪影。底层使用数组实现，数组中的每个元素可以存放一个 TimerTaskList 对象。TimerTaskList 是一个环形双向链表，在其中的链表项 TimerTaskEntry 中封装了真正的定时任务 TimerTask.Kafka 中到底是怎么推进时间的呢？Kafka 中的定时器借助了 JDK 中的 DelayQueue 来协助推进时间轮。具体做法是对于每个使用到的 TimerTaskList 都会加入到 DelayQueue 中。Kafka 中的 TimingWheel 专门用来执行插入和删除 TimerTaskEntry 的操作，而 DelayQueue 专门负责时间推进的任务。再试想一下，DelayQueue 中的第一个超时任务列表的 expiration 为 200ms，第二个超时任务为 840ms，这里获取 DelayQueue 的队头只需要 O(1)的时间复杂度。如果采用每秒定时推进，那么获取到第一个超时的任务列表时执行的 200 次推进中有 199 次属于“空推进”，而获取到第二个超时任务时有需要执行 639 次“空推进”，这样会无故空耗机器的性能资源，这里采用 DelayQueue 来辅助以少量空间换时间，从而做到了“精准推进”。Kafka 中的定时器真可谓是“知人善用”，用 TimingWheel 做最擅长的任务添加和删除操作，而用 DelayQueue 做最擅长的时间推进工作，相辅相成。"18.Kafka 中是怎么体现消息顺序性的？"kafka 每个 partition 中的消息在写入时都是有序的，消费时，每个 partition 只能被每一个 group 中的一个消费者消费，保证了消费时也是有序的。整个 topic 不保证有序。如果为了保证 topic 整个有序，那么将 partition 调整为 1。"19.为什么 Kafka 不支持读写分离？"在 Kafka 中，生产者写入消息、消费者读取消息的操作都是与 leader 副本进行交互的，从而实现的是一种主写主读的生产消费模型。Kafka 并不支持主写从读，因为主写从读有 2 个很明显的缺点:(1)数据一致性问题。数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间窗口会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中 A 数据的值都为 X，之后将主节点中 A 的值修改为 Y，那么在这个变更通知到从节点之前，应用读取从节点中的 A 数据的值并不为最新的 Y，由此便产生了数据不一致的问题。(2)延时问题。类似 Redis 这种组件，数据从写入主节点到同步至从节点中的过程需要经历网络→主节点内存→网络→从节点内存这几个阶段，整个过程会耗费一定的时间。而在 Kafka 中，主从同步会比 Redis 更加耗时，它需要经历网络→主节点内存→主节点磁盘→网络→从节点内存→从节点磁盘这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。"20.Kafka 中的消息是否会丢失和重复消费？"要确定 Kafka 的消息是否丢失或重复，从两个方面分析入手：消息发送和消息消费。1）消息发送 Kafka 消息发送有两种方式：同步（sync）和异步（async），默认是同步方式，可通过 producer.type 属性进行配置。Kafka 通过配置 request.required.acks 属性来确认消息的生产：0—表示不进行消息接收是否成功的确认；1—表示当 Leader 接收成功时确认；-1—表示 Leader 和 Follower 都接收成功时确认；综上所述，有 6 种消息生产的情况，下面分情况来分析消息丢失的场景：（1）acks=0，不和 Kafka 集群进行消息接收确认，则当网络异常、缓冲区满了等情况时，消息可能丢失；（2）acks=1、同步模式下，只有 Leader 确认接收成功后但挂掉了，副本没有同步，数据可能丢失；2）消息消费 Kafka 消息消费有两个 consumer 接口，Low-levelAPI 和 High-levelAPI：Low-levelAPI：消费者自己维护 offset 等值，可以实现对 Kafka 的完全控制；High-levelAPI：封装了对 parition 和 offset 的管理，使用简单；如果使用高级接口 High-levelAPI，可能存在一个问题就是当消息消费者从集群中把消息取出来、并提交了新的消息 offset 值后，还没来得及消费就挂掉了，那么下次再消费时之前没消费成功的消息就“诡异”的消失了；3）解决办法针对消息丢失：同步模式下，确认机制设置为-1，即让消息写入 Leader 和 Follower 之后再确认消息发送成功；异步模式下，为防止缓冲区满，可以在配置文件设置不限制阻塞超时时间，当缓冲区满时让生产者一直处于阻塞状态；针对消息重复：将消息的唯一标识保存到外部介质中，每次消费时判断是否处理过即可。"21.kafka 的 message 格式是什么样的？"一个 Kafka 的 Message 由一个固定长度的 header 和一个变长的消息体 body 组成。header 部分由一个字节的 magic(文件格式)和四个字节的 CRC32(用于判断 body 消息体是否正常)构成。当 magic 的值为 1 的时候，会在 magic 和 crc32 之间多一个字节的数据：attributes(保存一些相关属性，比如是否压缩、压缩格式等等)；如果 magic 的值为 0，那么不存在 attributes 属性。body 是由 N 个字节构成的一个消息体，包含了具体的 key/value 消息。"22.如果 leadercrash 时，ISR 为空怎么办？"kafka 在 Broker 端提供了一个配置参数：unclean.leader.election,这个参数有两个值：true（默认）：允许不同步副本成为 leader，由于不同步副本的消息较为滞后，此时成为 leader，可能会出现消息不一致的情况。false：不允许不同步副本成为 leader，此时如果发生 ISR 列表为空，会一直等待旧 leader 恢复，降低了可用性。"23.kafkaunclean 配置代表啥？会对 sparkstreaming 消费有什么影响？unclean.leader.election.enable 为 true 的话，意味着非 ISR 集合的 broker 也可以参与选举，这样有可能就会丢数据，sparkstreaming 在消费过程中拿到的 endoffset 会突然变小，导致 sparkstreamingjob 挂掉。如果 unclean.leader.election.enable 参数设置为 true，就有可能发生数据丢失和数据不一致的情况，Kafka 的可靠性就会降低；而如果 unclean.leader.election.enable 参数设置为 false，Kafka 的可用性就会降低。24.kafkaproducer 打数据，ack 为 0，1，-1 的时候代表啥，设置-1 的时候，什么情况下，leader 会认为一条消息 commit 了"1（默认）：数据发送到 Kafka 后，经过 leader 成功接收消息的的确认，就算是发送成功了。在这种情况下，如果 leader 宕机了，则会丢失数据。0：生产者将数据发送出去就不管了，不去等待任何返回。这种情况下数据传输效率最高，但是数据可靠性确是最低的。-1：producer 需要等待 ISR 中的所有 follower 都确认接收到数据后才算一次发送完成，可靠性最高。当 ISR 中所有 Replica 都向 Leader 发送 ACK 时，leader 才 commit，这时候 producer 才能认为一个请求中的消息都 commit 了。"25.kafkaproducer 如何优化打入速度？"增加线程提高 batch.size 增加更多 producer 实例增加 partition 数设置 acks=-1 时，如果延迟增大：可以增大 num.replica.fetchers（follower 同步数据的线程数）来调解；跨数据中心的传输：增加 socket 缓冲区设置以及 OStcp 缓冲区设置。"26.kafka 为什么那么快？"CacheFilesystemCachePageCache 缓存顺序。由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。Zero-copy。零拷技术减少拷贝次数 BatchingofMessages 批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。Pull 拉模式。使用拉模式进行消息的获取消费，与消费端处理能力相符。"27.什么情况下一个 broker 会从 ISR 中被踢出去？leader 会维护一个与其基本保持同步的 Replica 列表，该列表称为 ISR(in-syncReplica)，每个 Partition 都会有一个 ISR，而且是由 leader 动态维护，如果一个 follower 比一个 leader 落后太多，或者超过一定时间未发起数据复制请求，则 leader 将其重 ISR 中移除。28.kafkafollower 如何与 leader 同步数据？Kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。完全同步复制要求 AllAliveFollower 都复制完，这条消息才会被认为 commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，Follower 异步的从 Leader 复制数据，数据只要被 Leader 写入 log 就被认为已经 commit，这种情况下，如果 leader 挂掉，会丢失数据，kafka 使用 ISR 的方式很好的均衡了确保数据不丢失以及吞吐率。Follower 可以批量的从 Leader 复制数据，而且 Leader 充分利用磁盘顺序读以及 sendfile(zerocopy)机制，这样极大的提高复制性能，内部批量写磁盘，大幅减少了 Follower 与 Leader 的消息量差。29.kafka 中的 zookeeper 起到什么作用？可以不用 zookeeper 么？zookeeper 是一个分布式的协调组件，早期版本的 kafka 用 zk 做 meta 信息存储，consumer 的消费状态，group 的管理以及 offset 的值。考虑到 zk 本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了 zookeeper 的作用。

新的 consumer 使用了 kafka 内部的 groupcoordination 协议，也减少了对 zookeeper 的依赖，但是 broker 依然依赖于 ZK，zookeeper 在 kafka 中还用来选举 controller 和检测 broker 是否存活等等。30.kafka 中的 broker 是干什么的？broker 是消息的代理，Producers 往 Brokers 里面的指定 Topic 中写消息，Consumers 从 Brokers 里面拉取指定 Topic 的消息，然后进行业务处理，broker 在中间起到一个代理保存消息的中转站。31.Kafka 中的 ISR、AR 又代表什么？ISR 的伸缩又指什么？"ISR:In-SyncReplicas 副本同步队列 AR:AssignedReplicas 所有副本 ISR 是由 leader 维护，follower 从 leader 同步数据有一些延迟（包括延迟时间 replica.lag.time.max.ms 和延迟条数 replica.lag.max.messages 两个维度,当前最新的版本 0.10.x 中只支持 replica.lag.time.max.ms 这个维度），任意一个超过阈值都会把 follower 剔除出 ISR,存入 OSR（Outof-SyncReplicas）列表，新加入的 follower 也会先存放在 OSR 中。AR=ISR+OSR。"32.为什么要使用 kafka？为什么要使用消息队列？"缓冲和削峰：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka 在中间可以起到一个缓冲的作用，把消息暂存在 kafka 中，下游服务就可以按照自己的节奏进行慢慢处理。解耦和扩展性：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅 topic 的服务消费到，供多个毫无关联的业务使用。健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。"五、Linux1.在 Linux 下如何指定 dns 服务器，来解析某个域名？使用谷歌 DNS 解析百度：dig@8.8.8.8www.baidu.com2.设置 DNS 需要修改哪个配置文件？"全局的配置，可以在/etc/resolv.conf 文件中配置。指定网卡的配置，可以在/etc/sysconfig/network-scripts/ifcfg-eth0 文件中配置。"3.如何禁止服务器被 ping？echo1>/proc/sys/net/ipv4/icmp_echo_ignore_all4.用一条命令显示本机 eth0 网卡的 IP 地址，不显示其它字符？"方法一 ifconfigeth0|grepinet|awk-F ':' '{print$2}'|awk '{print$1}'方法二 ifconfigeth0|grep""inetaddr""|awk-F '[:]+' '{print$4}'方法三 ifconfigeth0|awk-F '[:]+' 'NR==2{print$4}'方法四 ifconfigeth0|sed-n '2p'|sed 's#^.addr:##g'|sed 's#Bc.$##g'方法五 ifconfigeth0|sed-n '2p'|sed-r 's#^.addr:(.)Bc.*$#\1#g'方法六(CENTOS7 也适用)：ipaddr|grepeth0|grepinet|awk '{print$2}'|awk-F '/' '{print$1}' "5.whereis 命令"当你不知道某个命令的位置时可以使用 whereis 命令，下面使用 whereis 查找 ls 的位置：whereisls。当你想查找某个可执行程序的位置，但这个程序又不在 whereis 的默认目录下，你可以使用 B 选项，并指定目录作为这个选项的参数。下面的命令在/tmp 目录下查找 lsmk 命令：whereis-u-B/tmp-flsmk。"6.service 命令"service 命令用于运行 SystemVinit 脚本，这些脚本一般位于/etc/init.d 文件下，这个命令可以直接运行这个文件夹里面的脚本，而不用加上路径。查看服务状态：servicesshstatus。查看所有服务状态：service–status-all。重启服务：servicesshrestart。"7.shutdown 命令"关闭系统并立即关机：shutdown-hnow。10 分钟后关机：shutdown-h+10。重启：shutdown-rnow。重启期间强制进行系统检查：shutdown-Frnow。"8.rpm 命令"使用 rpm 安装 apache：rpm-ivhhttpd-2.2.3-22.0.1.el5.i386.rpm。更新 apache：rpm-uvhhttpd-2.2.3-22.0.1.el5.i386.rpm。卸载/删除 apache：rpm-evhttpd。"9.yum 命令"使用 yum 安装 apache：yuminstallhttpd。更新 apache：yumupdatehttpd。卸载/删除 apache：yumremovehttpd。"10.export 命令"输出跟字符串 oracle 匹配的环境变量：export|grepORCALE。设置全局环境变量：exportORACLE_HOME=/u01/app/oracle/product/10.2.0。"11.unzip 命令"解压*.zip 文件：unziptest.zip。查看*.zip 文件的内容：unzip-ljasper.zip。"12.bzip2 命令"创建*.bz2 压缩文件：bzip2test.txt。解压*.bz2 文件：bzip2-dtest.txt.bz2。"13.gzip 命令"创建一个*.gz 的压缩文件：gziptest.txt。解压*.gz 文件：gzip-dtest.txt.gz。显示压缩的比率：gzip-l*.gz。"14.tar 命令"创建一个新的 tar 文件：tarcvfarchive_name.tardirname/。解压 tar 文件：tarxvfarchive_name.tar。查看 tar 文件：tartvfarchive_name.tar。"15.把当前目录下所有后缀名为.txt 的文件的权限修改为 777？"方式一，使用 xargs 命令：find./-typef-name""*.txt""|xargschmod777 方式二，使用 exec 命令：find./-typef-name""*.txt""-execchmod777{}"16.xargs 命令"将所有图片文件拷贝到外部驱动器：ls*.jpg|xargs-n1-icp{}/external-harddrive/directory。将系统中所有 jpg 文件压缩打包：find/-name*.jpg-typef-print|xargstar-cvzfimages.tar.gz。下载文件中列出的所有 url 对应的页面：caturl-list.txt|xargswget–c。"17.sort 命令"以升序对文件内容排序：sortnames.txt。以降序对文件内容排序：sort-rnames.txt。以第三个字段对/etc/passwd 的内容排序：sort-t:-k3n/etc/passwd|more。"18.diff 命令比较的时候忽略空白符：diff-wname_list.txtname_list_new.txt。19.vim 命令"打开文件并跳到第 10 行：vim+10filename.txt。打开文件跳到第一个匹配的行：vim+/search-termfilename.txt。以只读模式打开文件：vim-R/etc/passwd。"20.打印/etc/passwd 的 1 到 3 行？"使用 sed 命令：sed-n '1,3p' /etc/passwd 使用 awk 命令：awk 'NR>=1&&NR<=3{print$0}'/etc/passwd"21.awk 命令"删除重复行：$awk '!($0inarray){array[$0];print}' temp。打印/etc/passwd 中所有包含同样的 uid 和 gid 的行：awk-F ':' '$3=$4' /etc/passwd。打印文件中的指定部分的字段：awk '{print$2,$5;}' employee.txt。"22.打印/etc/ssh/sshd_config 的第一百行？sed-n '100p' /etc/ssh/sshd_config23.用 sed 命令将指定的路径/usr/local/http 替换成为/usr/src/local/http？echo"/usr/local/http/"|sed 's#/usr/local/#/usr/src/local/#' 24.sed 命令"当你将 Dos 系统中的文件复制到 Unix/Linux 后，这个文件每行都会以\r\n 结尾，sed 可以轻易将其转换为 Unix 格式的文件，使用\n 结尾的文件：sed 's/.$//' filename。反转文件内容并输出：sed-n '1!G;h;p' filename。为非空行添加行号：sed '/./=' thegeekstuff.txt|sed 'N;s/\n//' 。"25.grep 命令"在文件中查找字符串(不区分大小写)：grep-i""the""demo_file。输出成功匹配的行，以及该行之后的三行：grep-A3-i""example""demo_text。在一个文件夹中递归查询包含指定字符串的文件：grep-r""ramesh""*。"26.tail 命令"tail 命令默认显示文件最后的 10 行文本：tailfilename.txt。你可以使用-n 选项指定要显示的行数：tail-nNfilename.txt。你也可以使用-f 选项进行实时查看，这个命令执行后会等待，如果有新行添加到文件尾部，它会继续输出新的行，在查看日志时这个选项会非常有用。你可以通过 CTRL-C 终止命令的执行：tail-flog-file。"27.cp 命令"拷贝 file1 到 file2，并保持文件的权限、属主和时间戳：cp-pfile1file2。拷贝 file1 到 file2，如果 file2 存在会提示是否覆盖：cp-ifile1file2。"28.mv 命令"将 file1 重命名为 file2，如果 file2 存在则提示是否覆盖：mv-ifile1file2。-v 会输出重命名的过程，当文件名中包含通配符时，这个选项会非常方便：mv-vfile1file2。"29.rm 命令"删除文件前先确认：rm-ifilename.txt。在文件名中使用 shell 的元字符会非常有用。删除文件前先打印文件名并进行确认：rm-ifile*。递归删除文件夹下所有文件，并删除该文件夹：rm-rexample。"30.df 命令"显示文件系统的磁盘使用情况，默认情况下 df-k 将以字节为单位输出磁盘的使用量。使用 df-h 选项可以以更符合阅读习惯的方式显示磁盘使用量。使用 df-T 选项显示文件系统类型。"31.ls 命令"以易读的方式显示文件大小(显示为 MB,GB...)：ls-lh。以最后修改时间升序列出文件：ls-ltr。在文件名后面显示文件类型：ls-F。"32.在整个目录树下查找文件“core”，如发现则无需提示直接删除它们？find/-namecore-execrm{}\;33.如何在 home 目录下找出 120 天之前被修改过的文件？"find/home-mtime+120"34.如何在 var 目录下找出 90 天之内未被访问过的文件？find/var!-atime-9035.如何在 usr 目录下找出大小超过 10MB 的文件?find/usr-typef-size+10240k36.find 命令如何使用？"查找指定文件名的文件(不区分大小写)：find-iname""MyProgram.c""。对找到的文件执行某个命令：find-iname""MyProgram.c""-execmd5sum{}\;。查找 home 目录下的所有空文件：find~-empty。"37.vim 有几种工作模式？命令模式，行末模式，编辑模式六、Mybatis1.Mybatis 的一级、二级缓存"（1）一级缓存:基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Sessionflush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。（2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态),可在它的映射文件中配置；（3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear 掉并重新更新，如果开启了二级缓存，则只根据配置判断是否刷新。"2.Mybatis 都有哪些 Executor 执行器？它们之间的区别是什么？"Mybatis 有三种基本的 Executor 执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。SimpleExecutor：每执行一次 update 或 select，就开启一个 Statement 对象，用完立刻关闭 Statement 对象。ReuseExecutor：执行 update 或 select，以 sql

作为 key 查找 Statement 对象，存在就使用，不存在就创建，用完后，不关闭 Statement 对象，而是放置于 Map<String,Statement>内，供下一次使用。简言之，就是重复使用 Statement 对象。BatchExecutor：执行 update（没有 select，JDBC 批处理不支持 select），将所有 sql 都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个 Statement 对象，每个 Statement 对象都是 addBatch()完毕后，等待逐一执行 executeBatch()批处理。与 JDBC 批处理相同。作用范围：Executor 的这些特点，都严格限制在 SqlSession 生命周期范围内。"3.Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？"Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke()方法发现 a.getB()是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 a.getB().getName()方法的调用。这就是延迟加载的基本原理。当然了，不光是 Mybatis，几乎所有的包括 Hibernate，支持延迟加载的原理都是一样的。"4.Mybatis 能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别。"答：能，Mybatis 不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实就是一对一查询，只需要把 selectOne()修改为 selectList()即可；多对多查询，其实就是一对多查询，只需要把 selectOne()修改为 selectList()即可。关联对象查询，有两种实现方式，一种是单独发送一个 sql 去查询关联对象，赋给主对象，然后返回主对象。另一种是使用嵌套查询，嵌套查询的含义为使用 join 查询，一部分列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。那么问题来了，join 查询出来 100 条记录，如何确定主对象是 5 个，而不是 100 个？其去重复的原理是 resultMap 标签内的 id 子标签，指定了唯一确定一条记录的 id 列，Mybatis 根据 id 列值来完成 100 条记录的去重复功能，id 可以有多个，代表了联合主键的语意。同样主对象的关联对象，也是根据这个原理去重复的，尽管一般情况下，只有主对象会有重复记录，关联对象一般不会重复。"5.Mybatis 全局配置文件中有哪些标签?分别代表什么意思?"configuration 配置 properties 属性:可以加载 properties 配置文件的信息 settings 设置：可以设置 mybatis 的全局属性 typeAliases 类型命名 typeHandlers 类型处理器 objectFactory 对象工厂 plugins 插件 environments 环境 environment 环境变量 transactionManager 事务管理器 dataSource 数据源 mappers 映射器"6.说一下 resultMap 和 resultType？"resultmap 是手动提交，人为提交，resulttype 是自动提交 MyBatis 中在查询进行 select 映射的时候，返回类型可以用 resultType，也可以用 resultMap，resultType 是直接表示返回类型的，而 resultMap 则是对外部 ResultMap 的引用，但是 resultType 跟 resultMap 不能同时存在。在 MyBatis 进行查询映射时，其实查询出来的每一个属性都是放在一个对应的 Map 里面的，其中键是属性名，值则是其对应的值。1.当提供的返回类型属性是 resultType 时，MyBatis 会将 Map 里面的键值对取出赋给 resultType 所指定的对象对应的属性。所以其实 MyBatis 的每一个查询映射的返回类型都是 ResultMap，只是当提供的返回类型属性是 resultType 的时候，MyBatis 对自动的给把对应的值赋给 resultType 所指定对象的属性。2.当提供的返回类型是 resultMap 时，因为 Map 不能很好表示领域模型，就需要自己再进一步的把它转化为对应的对象，这常常在复杂查询中很有作用。"7.Mybatis 动态 SQL？"1)传统的 JDBC 的方法，在组合 SQL 语句的时候需要去拼接，稍微不注意就会少少了一个空格，标点符号，都会导致系统错误。Mybatis 的动态 SQL 就是为了解决这种问题而产生的；Mybatis 的动态 SQL 语句值基于 OGNL 表达式的，方便在 SQL 语句中实现某些逻辑；可以使用标签组合成灵活的 sql 语句，提供开发的效率。2)Mybatis 的动态 SQL 标签主要由以下几类：If 语句（简单的条件判断）Choose(when/otherwise),相当于 java 语言中的 switch，与 jstl 中 choose 类似 Trim(对包含的内容加上 prefix，或者 suffix)Where(主要是用来简化 SQL 语句中 where 条件判断，能智能的处理 and/or 不用担心多余的语法导致的错误)Set(主要用于更新时候)Foreach(一般使用在 mybatisin 语句查询时特别有用)"8.Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？不同的 Xml 映射文件，如果配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复 9.如何获取自动生成的(主)键值?"<insertid=”insertname” usegeneratedkeys=”true” keyproperty=”id”>insertintonames(name)values(#{name})</insert>"10.Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？"第一种是使用 resultMap 标签，逐一定义数据库列名和对象属性名之间的映射关系。第二种是使用 sql 列的别名功能，将列的别名书写为对象属性名。有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。"11.当实体类的属性名和表种字段名不一致怎么办?有两种解决方案:可以在 sql 语句给字段名取别名,别名于实体类属性名同名,也可以用 resultMap 来映射字段名和实体类属性名——对应 12.#{}和${}的区别是什么？"{}是预编译处理，${}是字符串替换。Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；Mybatis 在处理{}时，就是把时，就是把{}替换成变量的值。使用#{}可以有效的防止 SQL 注入，提高系统安全性。"13.Mybatis 使用场合?专注于 sql 本身,是一个足够灵活的 dao 层解决方案.,对性能的要求很高,或者需求多变的项目 14.Mybatis 的优缺点?"Mybaits 的优点：（1）基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签，支持编写动态 SQL 语句，并可重用。（2）与 JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；（3）很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）。（4）能够与 Spring 很好的集成；（5）提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。MyBatis 框架的缺点：（1）SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求。（2）SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。MyBatis 框架适用场合：（1）MyBatis 专注于 SQL 本身，是一个足够灵活的 DAO 层解决方案。（2）对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis 将是不错的选择。"15.什么是 Mybatis？"1）mybatis 是一个半 ORM 框架，它内部封装了 JDBC，开发时只需要关乎 sql 语句本身，不需要花费精力去处理驱动，创建连接，创建 1statement 等繁复过程。2）mybatis 可以使用 xml 或注解来配置和映射原生信息。将 pijo 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。3）通过 xm 文件或注解的方式将要执行的各种 statement 配置起来,并通 java 对象和 statement 中 sql 的动态参数进行映射生成最终的 sql 语句,最后由 mybatis 框架执行 sql 并将结果映射 java 对象返回."七、MySQL1.解释 MySQL 外连接、内连接与自连接的区别"先说什么是交叉连接:交叉连接又叫笛卡尔积，它是指不使用任何条件，直接将一个表的所有记录和另一个表中的所有记录一一匹配。内连接则是只有条件的交叉连接，根据某个条件筛选出符合条件的记录，不符合条件的记录不会出现在结果集中，即内连接只连接匹配的行。外连接其结果集中不仅包含符合连接条件的行，而且还会包括左表、右表或两个表中的所有数据行，这三种情况依次称之为左外连接，右外连接，和全外连接。左外连接，也称左连接，左表为主表，左表中的所有记录都会出现在结果集中，对于那些在右表中并没有匹配的记录，仍然要显示，右边对应的那些字段值以 NULL 来填充。右外连接，也称右连接，右表为主表，右表中的所有记录都会出现在结果集中。左连接和右连接可以互换，MySQL 目前还不支持全外连接。"2.Mysql 如何优化 DISTINCT?DISTINCT 在所有列上转换为 GROUPBY，并与 ORDERBY 子句结合使用。3.自增主键最大 ID 记录，MyISAM 和 InnoDB 分别是如何存储的"MyISAM 表把自增主键的最大 ID 记录到数据文件里 InnoDB 表把自增主键的最大 ID 记录到内存中"4.MySQL 主从复制原理流程"主：binlog 线程——记录下所有改变了数据库数据的语句，放进 master 上的 binlog 中；从：io 线程——在使用 startslave 之后，负责从 master 上拉取 binlog 内容，放进自己的 relaylog 中；从：sql 执行线程——执行 relaylog 中的语句；"5.delete、truncate、drop 区别"truncate 和 delete 只删除数据，不删除表结构,drop 删除表结构，并且释放所占的空间。删除数据的速度，drop>truncate>deletedelete 属于 DML 语言，需要事务管理，commit 之后才能生效。drop 和 truncate 属于 DDL 语言，操作立刻生效，不可回滚。使用场合：当你不再需要该表时，用 drop;当你仍要保留该表，但要删除所有记录时，用 truncate;当你要删除部分记录时（alwayswithawhereclause),用 delete。"6.key 和 index 的区别"key 是数据库的物理结构，它包含两层意义和作用，一是约束（偏重于约束和规范数据库的结构完整性），二是索引（辅助查询用的）。包括 primarykey,uniquekey,foreignkey 等 index 是数据库的物理结构，它只是辅助查询的，它创建时会在另外的表空间（mysql 中的

innodb 表空间）以一个类似目录的结构存储。索引要分类的话，分为前缀索引、全文本索引等；"7.MySQL 优化"开启查询缓存，优化查询 explain 你的 select 查询，这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的当只要一行数据时使用 limit1，MySQL 数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据为搜索字段建索引使用 ENUM 而不是 VARCHAR。如果你有一个字段，比如“性别”，“国家”，“民族”，“状态”或“部门”，你知道这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是 VARCHARPreparedStatementsPreparedStatements 很像存储过程，是一种运行在后台的 SQL 语句集合，我们可以从使用 preparedstatements 获得很多好处，无论是性能问题还是安全问题。PreparedStatements 可以检查一些你绑定好的变量，这样可以保护你的程序不会受到“SQL 注入式”攻击垂直分表选择正确的存储引擎"8.行级锁定的缺点"比页级或表级锁定占用更多的内存。当在表的大部分中使用时，比页级或表级锁定速度慢，因为你必须获取更多的锁。如果你在大部分数据上经常进行 GROUPBY 操作或者必须经常扫描整个表，比其它锁定明显慢很多。用高级别锁定，通过支持不同的类型锁定，你也可以很容易地调节应用程序，因为其锁成本小于行级锁定。"9.行级锁定的优点"1、当在许多线程中访问不同的行时只存在少量锁定冲突。2、回滚时只有少量的更改 3、可以长时间锁定单一的行。"10.在 MVCC 并发控制中，读操作可以分成哪几类？"快照读(snapshotread)：读取的是记录的可见版本(有可能是历史版本)，不用加锁（共享读锁 s 锁也不加，所以不会阻塞其他事务的写）当前读(currentread)：读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录"11.MVVC 了解过吗"MySQLInnoDB 存储引擎，实现的是基于多版本的并发控制协议——MVCC(Multi-VersionConcurrencyControl)注：与 MVCC 相对的，是基于锁的并发控制，Lock-BasedConcurrencyControlMVCC 最大的好处：读不加锁，读写不冲突。在读多写少的 OLTP 应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，现阶段几乎所有的 RDBMS，都支持了 MVCC。LBCC：Lock-BasedConcurrencyControl，基于锁的并发控制 MVCC：Multi-VersionConcurrencyControl 基于多版本的并发控制协议。纯粹基于锁的并发机制并发量低，MVCC 是在基于锁的并发控制上的改进，主要是在读操作上提高了并发量。"12.表分区有什么好处？"1、存储更多数据。分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备。和单个磁盘或者文件系统相比，可以存储更多数据 2、优化查询。在 where 语句中包含分区条件时，可以只扫描一个或多个分区表来提高查询效率；涉及 sum 和 count 语句时，也可以在多个分区上并行处理，最后汇总结果。3、分区表更容易维护。例如：想批量删除大量数据可以清除整个分区。4、避免某些特殊的瓶颈，例如 InnoDB 的单个索引的互斥访问，ext3 问价你系统的 inode 锁竞争等。"13.表分区与分表的区别"分表：指的是通过一定规则，将一张表分解成多张不同的表。比如将用户订单记录根据时间成多个表。分表与分区的区别在于：分区从逻辑上来讲只有一张表，而分表则是将一张表分解成多张表。"14.什么是表分区？表分区，是指根据一定规则，将数据库中的一张表分解成多个更小的，容易管理的部分。从逻辑上看，只有一张表，但是底层却是由多个物理分区组成。15.什么情况下应不建或少建索引"1、表记录太少 2、经常插入、删除、修改的表 3、数据重复且分布平均的表字段，假如一个表有 10 万行记录，有一个字段 A 只有 T 和 F 两种值，且每个值的分布概率大约为 50%，那么对这种表 A 字段建索引一般不会提高数据库的查询速度。4、经常和主字段一块查询但主字段索引值比较多的表字段"16.说一说三个范式"第一范式:每个列都不可以再拆分.第二范式:非主键列完全依赖于主键,而不能是依赖于主键的一部分.第三范式:非主键列只依赖于主键,不依赖于其他非主键。在设计数据库结构的时候,要尽量遵守三范式,如果不遵守,必须有足够的理由.比如性能.事实上我们经常会为了性能而妥协数据库的设计。"17.什么是存储过程？有哪些优缺点？"存储过程是一些预编译的 SQL 语句。1、更加直白的理解：存储过程可以说是一个记录集，它是由一些 T-SQL 语句组成的代码块，这些 T-SQL 语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。2、存储过程是一个预编译的代码块，执行效率比较高,一个存储过程替代大量 T_SQL 语句，可以降低网络通信量，提高通信速率,可以一定程度上确保数据安全但是,在互联网项目中,其实是不太推荐存储过程的,比较出名的就是阿里的《Java 开发手册》中禁止使用存储过程,我个人的理解是,在互联网项目中,迭代太快,项目的生命周期也比较短,人员流动相比于传统的项目也更加频繁,在这样的情况下,存储过程的管理确实是没有那么方便,同时,复用性也没有写在服务层那么好。"18.关心过业务系统里面的 sql 耗时吗?统计过慢查询吗?对慢查询都怎么优化过?"在业务系统中,除了使用主键进行的查询,其他的我都会在测试库上测试其耗时,慢查询的统计主要由运维在做,会定期将业务中的慢查询反馈给我们。慢查询的优化首先要搞明白慢的原因是什么?是查询条件没有命中索引?是 load 了不需要的数据列?还是数据量太大?所以优化也是针对这三个方向来的首先分析语句,看看是否 load 了额外的数据,可能是查询了多余的行并且抛弃掉了,可能是加载了许多结果中并不需要的列,对语句进行分析以及重写。分析语句的执行计划,然后获得其使用索引的情况,之后修改语句或者修改索引,使得语句可以尽可能的命中索引。如果对语句的优化已经无法进行,可以考虑表中的数据量是否太大,如果是的话可以进行横向或者纵向的分表。"19.超大分页怎么处理?"超大的分页一般从两个方向上来解决.数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于 selectfromtablewhereage>20limit1000000,10 这种查询其实也是有可以优化的余地的.这条语句需要 load1000000 数据然后基本上全部丢弃,只取 10 条当然比较慢.当时我们可以修改为 selectfromtablewhereidin(selectidfromtablewhereage>20limit1000000,10).这样虽然也 load 了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快.同时如果 ID 连续的好,我们还可以 select*fromtablewhereid>1000000limit10,效率也是不错的,优化的可能性有许多种,但是核心思想都一样,就是减少 load 的数据.从需求的角度减少这种请求....主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止 ID 泄漏且连续被人恶意攻击.解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至 redis 等 k-V 数据库中,直接返回即可.在阿里巴巴《Java 开发手册》中,对超大分页的解决办法是类似于上面提到的第一种。"20.MySQL 的 binlog 有有几种录入格式?分别有什么区别?"有三种格式,statement,row 和 mixed.statement 模式下,记录单元为语句.即每一个 sql 造成的影响会记录.由于 sql 的执行是有上下文的,因此在保存的时候需要保存相关的信息,同时还有一些使用了函数之类的语句无法被记录复制.row 级别下,记录单元为每一行的改动,基本是可以全部记下来但是由于很多操作,会导致大量行的改动(比如 altertable),因此这种模式的文件保存的信息太多,日志量太大.mixed.一种折中的方案,普通操作使用 statement 记录,当无法使用 statement 的时候使用 row.此外,新版的 MySQL 中对 row 级别也做了一些优化,当表结构发生变化的时候,会记录语句而不是逐行记录。"21.varchar(10)和 int(10)代表什么含义?varchar 的 10 代表了申请的空间长度,也是可以存储的数据的最大长度,而 int 的 10 只是代表了展示的长度,不足 10 位以 0 填充.也就是说,int(1)和 int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示。22.如果要存储用户的密码散列,应该使用什么字段进行存储?密码散列,盐,用户身份证号等固定长度的字符串应该使用 char 而不是 varchar 来存储,这样可以节省空间且提高检索效率。23.字段为什么要求定义为 notnull?"MySQL 官网这样介绍:NULLcolumnsrequireadditionalspaceintherowtorecordwhethertheirvaluesareNULL.ForMyISAMtables,eachNULLcolumntakesonebitextra,roundeduptothenearestbyte.null 值会占用更多的字节,且会在程序中造成很多与预期不符的情况。"24.主键使用自增 ID 还是 UUID?"推荐使用自增 ID,不要使用 UUID.因为在 InnoDB 存储引擎中,主键索引是作为聚簇索引存在的,也就是说,主键索引的 B+树叶子节点上存储了主键索引以及全部的数据(按照顺序),如果主键索引是自增 ID,那么只需要不断向后排列即可,如果是 UUID,由于到来的 ID 与原来的大小不确定,会造成非常多的数据插入,数据移动,然后导致产生很多的内存碎片,进而造成插入性能的下降.总之,在数据量大一些的情况下,用自增主键性能会好一些.*图片来源于《高性能 MySQL》:其中默认后缀为使用自增 ID,_uuid 为使用 UUID 为主键的测试,测试了插入 100w 行和 300w 行的性能 image-20210813174937500 关于主键是聚簇索引,如果没有主键,InnoDB 会选择一个唯一键来作为聚簇索引,如果没有唯一键,会生成一个隐式的主键。"25.为什么要尽量设定一个主键?主键是数据库确保数据行在整张表唯一性的保障,即使业务上本张表没有主键,也建议添加一个自增长的 ID 列作为主键.设定了主键之后,在后续的删改查的时候可能更加快速以及确保操作数据范围安全。26.在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?"使用不等于查询列参与了数学运算或者函数在字符串 like 时左边是通配符.类似于’%aaa’当 mysql 分析全表扫描比使用索引快的时候不使用索引当使用联合索引,前面一个条件为范围查询,后面的即使符合最左前缀原则,也无法使用索引."27.MySQL 有哪些日志，分别是什么用处？"mysql 日志一般分为 5 种错误日志：-log-err(记录启动，运行，停止 mysql 时出现的信息)二进制日志：-log-bin（记录所有更改数据的语

句，还用于复制，恢复数据库用）查询日志：-log（记录建立的客户端连接和执行的语句）慢查询日志:-log-slow-queries（记录所有执行超过 long_query_time 秒的所有查询）更新日志:-log-update（二进制日志已经代替了老的更新日志，更新日志在 MySQL5.1 中不再使用）"28.MySQL 中 varchar 与 char 的区别以及 varchar(50)中的 50 代表的涵义"(1)、varchar 与 char 的区别(2)、varchar(50)中 50 的涵义(3)、int（20）中 20 的涵义(4)、mysql 为什么这么设计"29.MySQL 的 redo 日志的刷盘时机"logbuffer 空间不足时事务提交时后台线程不停的刷刷刷正常关闭服务器时做所谓的 checkpoint 时"30.MySQL 的 redo 日志和 undo 日志分别有什么用？"1）redo 作用：保证事务的持久性 MySQL 作为一个存储系统，为了保证数据的可靠性，最终得落盘。但是，又为了数据写入的速度，需要引入基于内存的""缓冲池""。其实不止 MySQL，这种引入缓冲来解决速度问题的思想无处不在。既然数据是先缓存在缓冲池中，然后再以某种方式刷新到磁盘，那么就存在因宕机导致的缓冲池中的数据丢失，为了解决这种情况下的数据丢失问题，引入了 redolog。在其他存储系统，比如 Elasticsearch 中，也有类似的机制，叫 translog。但是一般讨论数据写入时，在 MySQL 中，一般叫事务操作，根据事务的 ACID 特性，如何保证一个事务提交后 Durability 的保证？而这就是 redolog 的作用。当向 MySQL 写用户数据时，先写 redolog，然后 redolog 根据""某种方式""持久化到磁盘，变成 redologfile，用户数据则在""buffer""中(比如数据页、索引页)。如果发生宕机，则读取磁盘上的 redologfile 进行数据的恢复。从这个角度来说，MySQL 事务的持久性是通过 redolog 来实现的。2）undo 作用：实现事务回滚 Undolog 是 InnoDBMVCC 事务特性的重要组成部分。当我们对记录做了变更操作时就会产生 undo 记录，Undo 记录默认被记录到系统表空间(ibdata)中，但从 5.6 开始，也可以使用独立的 Undo 表空间。Undo 记录中存储的是老版本数据，当一个旧的事务需要读取数据时，为了能读取到老版本的数据，需要顺着 undo 链找到满足其可见性的记录。当版本链很长时，通常可以认为这是个比较耗时的操作。大多数对数据的变更操作包括 INSERT/DELETE/UPDATE，其中 INSERT 操作在事务提交前只对当前事务可见，因此产生的 Undo 日志可以在事务提交后直接删除（谁会对刚插入的数据有可见性需求呢!!），而对于 UPDATE/DELETE 则需要维护多版本信息，在 InnoDB 里，UPDATE 和 DELETE 操作产生的 Undo 日志被归成一类，即 update_undo。"31.为什么 InnoDB 一定会生成主键？因为 Innodb 的数据结构是通过聚簇索引组织起来的，如果没有主键的话，通过其他索引回表的时候没法查到相应的数据行。32.InnoDB 如果没有设置主键的话，它内部会怎么处理？"优先使用用户自定义主键作为主键，如果用户没有定义主键，则选取一个 Unique 键作为主键，如果表中连 Unique 键都没有定义的话，则 InnoDB 会为表默认添加一个名为 row_id 的隐藏列作为主键。所以我们从上表中可以看出：InnoDB 存储引擎会为每条记录都添加 transaction_id 和 roll_pointer 这两个列，但是 row_id 是可选的（在没有自定义主键以及 Unique 键的情况下才会添加该列）。这些隐藏列的值不用我们操心，InnoDB 存储引擎会自己帮我们生成的。"33.InnoDB 删除某条记录后，内部会怎么处理？记录头信息里的 delete_mask 标记位设置为 1（表示该记录已删除），同时将记录从记录行链表中断开，并加入到垃圾链表中，垃圾链表的空间后续可以复用。34.InnoDB 主键索引跟非主键索引在数据存储上的差异"主键索引的叶子节点存的是整行数据。在 InnoDB 里，主键索引也被称为聚簇索引（clusteredindex）。非主键索引的叶子节点内容是主键的值。在 InnoDB 里，非主键索引也被称为二级索引（secondaryindex）。"35.InnoDB 的数据是怎么存储的？InnoDB 的主键索引文件上直接存放该行数据，称为聚簇索引，非主索引指向对主键的引用。36.MyIsam 的数据是怎么存储的？"MyIsam 索引的节点中存储的是数据的物理地址（磁道和扇区），在查找数据时，查找到索引后，根据索引节点中的物理地址，查找到具体的数据内容。"37.InnoDB 有聚簇索引吗？MyIsam 呢？"InnoDB 有聚簇索引，主键索引就是聚簇索引。MyIsam 没有聚簇索引，因为他的索引和记录行是分开存储的。"38.什么是聚簇索引？"聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据非聚簇索引：将数据与索引分开存储，索引结构的叶子节点指向了数据的对应行"39.MySQL 索引的类型"聚簇索引、二级（辅助）索引 B 树索引、hash 索引"40.有了解过“回表”的概念吗？什么情况下会出现“回表”？回表就是先通过数据库索引扫描出数据所在的行，再通过行主键 id 取出索引中未提供的数据，即基于非主键索引的查询需要多扫描一棵索引树。当查询的字段在二级索引上没有的时候，就需要“回表”在主键索引上再查一次。41.事务的隔离级别了解过吗？"1、读未提交（ReadUncommited），该隔离级别允许脏读取，其隔离级别最低；比如事务 A 和事务 B 同时进行，事务 A 在整个执行阶段，会将某数据的值从 1 开始一直加到 10，然后进行事务提交，此时，事务 B 能够看到这个数据项在事务 A 操作过程中的所有中间值（如 1 变成 2，2 变成 3 等），而对这一系列的中间值的读取就是未授权读取 2、授权读取也称为已提交读（ReadCommited），授权读取只允许获取已经提交的数据。比如事务 A 和事务 B 同时进行，事务 A 进行+1 操作，此时，事务 B 无法看到这个数据项在事务 A 操作过程中的所有中间值，只能看到最终的 10。另外，如果说有一个事务 C，和事务 A 进行非常类似的操作，只是事务 C 是将数据项从 10 加到 20，此时事务 B 也同样可以读取到 20，即授权读取允许不可重复读取。3、可重复读（RepeatableRead)就是保证在事务处理过程中，多次读取同一个数据时，其值都和事务开始时刻是一致的，因此该事务级别禁止不可重复读取和脏读取，但是有可能出现幻影数据。所谓幻影数据，就是指同样的事务操作，在前后两个时间段内执行对同一个数据项的读取，可能出现不一致的结果。在上面的例子中，可重复读取隔离级别能够保证事务 B 在第一次事务操作过程中，始终对数据项读取到 1，但是在下一次事务操作中，即使事务 B（注意，事务名字虽然相同，但是指的是另一个事务操作）采用同样的查询方式，就可能读取到 10 或 20；4、串行化是最严格的事务隔离级别，它要求所有事务被串行执行，即事务只能一个接一个的进行处理，不能并发执行。"42.说一下什么是事务的 ACID 属性吧"原子性（atomicity)一个事务要么全部提交成功，要么全部失败回滚，不能只执行其中的一部分操作，这就是事务的原子性 2.一致性（consistency)事务的执行不能破坏数据库数据的完整性和一致性，一个事务在执行之前和执行之后，数据库都必须处于一致性状态。如果数据库系统在运行过程中发生故障，有些事务尚未完成就被迫中断，这些未完成的事务对数据库所作的修改有一部分已写入物理数据库，这是数据库就处于一种不正确的状态，也就是不一致的状态隔离性（isolation）事务的隔离性是指在并发环境中，并发的事务时相互隔离的，一个事务的执行不能不被其他事务干扰。不同的事务并发操作相同的数据时，每个事务都有各自完成的数据空间，即一个事务内部的操作及使用的数据对其他并发事务时隔离的，并发执行的各个事务之间不能相互干扰。在标准 SQL 规范中，定义了 4 个事务隔离级别，不同的隔离级别对事务的处理不同，分别是：未授权读取，授权读取，可重复读取和串行化持久性（durability）一旦事务提交，那么它对数据库中的对应数据的状态的变更就会永久保存到数据库中。即使发生系统崩溃或机器宕机等故障，只要数据库能够重新启动，那么一定能够将其恢复到事务成功结束的状态"43.了解过哪些存储引擎？各有什么优缺点？"常用的是 MyISAM 和 InnoDB。InnoDB：支持事务、支持外键、支持行级锁、不支持全文索引 MyISAM：不支持事务、不支持外键、不支持行级锁、支持全文索引"44.在建立索引的时候,都有哪些需要考虑的因素呢?建立索引的时候一般要考虑到字段的使用频率,经常作为条件进行查询的字段比较适合.如果需要建立联合索引的话,还需要考虑联合索引中的顺序.此外也要考虑其他方面,比如防止过多的所有对表造成太大的压力.这些都和实际的表结构以及查询方式有关。45.Hash 索引和 B+树索引有什么区别或者说优劣呢?"首先要知道 Hash 索引和 B+树索引的底层实现原理:hash 索引底层就是 hash 表,进行查找时,调用一次 hash 函数就可以获取到相应的键值,之后进行回表查询获得实际数据.B+树底层实现是多路平衡查找树.对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表查询数据.那么可以看出他们有以下的不同:hash 索引进行等值查询更快(一般情况下),但是却无法进行范围查询.因为在 hash 索引中经过 hash 函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.而 B+树的的所有节点皆遵循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.hash 索引不支持使用索引进行排序,原理同上.hash 索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为 hash 函数的不可预测.AAAA 和 AAAAB 的索引没有相关性.hash 索引任何时候都避免不了回表查询数据,而 B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.hash 索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生 hash 碰撞,此时效率可能极差.而 B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.因此,在大多数情况下,直接选择 B+树索引可以获得稳定且较好的查询速度.而不需要使用 hash 索引."46.索引是个什么样的数据结构呢?索引的数据结构和具体存储引擎的实现有关,在 MySQL 中使用较多的索引有 Hash 索引,B+树索引等,而我们经常使用的 InnoDB 存储引擎的默认索引实现为:B+树索引。47.什么是索引?索引是一种数据结构,可以帮助我们快速的进行数据的查找。八、Netty1.Netty 高性能体现在哪些方面？"1）传输：IO 模型在很大程度上决定了框架的性能，相比于 bio，netty 建议采用异步通信模式，因为 nio 一个线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 IO 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大

的提升。正如代码中所示，使用的是 NioEventLoopGroup 和 NioSocketChannel 来提升传输效率。2）协议：采用什么样的通信协议，对系统的性能极其重要，netty 默认提供了对 GoogleProtobuf 的支持，也可以通过扩展 Netty 的编解码接口，用户可以实现其它的高性能序列化框架。3）线程：netty 使用了 Reactor 线程模型，但 Reactor 模型不同，对性能的影响也非常大，下面介绍常用的 Reactor 线程模型有三种，分别如下：Reactor 单线程模型：单线程模型的线程即作为 NIO 服务端接收客户端的 TCP 连接，又作为 NIO 客户端向服务端发起 TCP 连接，即读取通信对端的请求或者应答消息，又向通信对端发送消息请求或者应答消息。理论上一个线程可以独立处理所有 IO 相关的操作，但一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，即便 NIO 线程的 CPU 负荷达到 100%，也无法满足海量消息的编码、解码、读取和发送，又因为当 NIO 线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了 NIO 线程的负载，最终会导致大量消息积压和处理超时，NIO 线程会成为系统的性能瓶颈。Reactor 多线程模型：有专门一个 NIO 线程用于监听服务端，接收客户端的 TCP 连接请求；网络 IO 操作(读写)由一个 NIO 线程池负责，线程池可以采用标准的 JDK 线程池实现。但百万客户端并发连接时，一个 nio 线程用来监听和接受明显不够，因此有了主从多线程模型。主从 Reactor 多线程模型：利用主从 NIO 线程模型，可以解决 1 个服务端监听线程无法有效处理所有客户端连接的性能不足问题，即把监听服务端，接收客户端的 TCP 连接请求分给一个线程池。因此，在代码中可以看到，我们在 server 端选择的就是这种方式，并且也推荐使用该线程模型。在启动类中创建不同的 EventLoopGroup 实例并通过适当的参数配置，就可以支持上述三种 Reactor 线程模型。"2.说说 Netty 的执行流程？"创建 ServerBootStrap 实例设置并绑定 Reactor 线程池：EventLoopGroup，EventLoop 就是处理所有注册到本线程的 Selector 上面的 Channel 设置并绑定服务端的 channel 创建处理网络事件的 ChannelPipeline 和 handler，网络时间以流的形式在其中流转，handler 完成多数的功能定制：比如编解码 SSl 安全认证绑定并启动监听端口当轮训到准备就绪的 channel 后，由 Reactor 线程：NioEventLoop 执行 pipline 中的方法，最终调度并执行 channelHandler"3.Netty 支持哪些心跳类型设置？readerIdleTime：为读超时时间（即测试端一定时间内未接受到被测试端消息）。writerIdleTime：为写超时时间（即测试端一定时间内向被测试端发送消息）。allIdleTime：所有类型的超时时间。4.Netty 发送消息有几种方式？"Netty 有两种发送消息的方式：直接写入 Channel 中，消息从 ChannelPipeline 当中尾部开始移动；写入和 ChannelHandler 绑定的 ChannelHandlerContext 中，消息从 ChannelPipeline 中的下一个 ChannelHandler 中移动。"5.Netty 中有哪些重要组件？"Channel：Netty 网络操作抽象类，它除了包括基本的 I/O 操作，如 bind、connect、read、write 等。EventLoop：主要是配合 Channel 处理 I/O 操作，用来处理连接的生命周期中所发生的事情。ChannelFuture：Netty 框架中所有的 I/O 操作都为异步的，因此我们需要 ChannelFuture 的 addListener()注册一个 ChannelFutureListener 监听事件，当操作执行成功或者失败时，监听就会自动触发返回结果。ChannelHandler：充当了所有处理入站和出站数据的逻辑容器。ChannelHandler 主要用来处理各种事件，这里的事件很广泛，比如可以是连接、数据接收、异常、数据转换等。ChannelPipeline：为 ChannelHandler 链提供了容器，当 channel 创建时，就会被自动分配到它专属的 ChannelPipeline，这个关联是永久性的。"6.Netty 的心跳机制了解么？"在 TCP 保持长连接的过程中，可能会出现断网等网络异常出现，异常发生的时候，client 与 server 之间如果没有交互的话，它们是无法发现对方已经掉线的。为了解决这个问题,我们就需要引入心跳机制。心跳机制的工作原理是:在 client 与 server 之间在一定时间内没有数据交互时,即处于 idle 状态时,客户端或服务器就会发送一个特殊的数据包给对方,当接收方收到这个数据报文后,也立即发送一个特殊的数据报文,回应发送方,此即一个 PING-PONG 交互。所以,当某一端收到心跳消息后,就知道了对方仍然在线,这就确保 TCP 连接的有效性.TCP 实际上自带的就有长连接选项，本身是也有心跳包机制，也就是 TCP 的选项：SO_KEEPALIVE。但是，TCP 协议层面的长连接灵活性不够。所以，一般情况下我们都是在应用层协议上实现自定义心跳机制的，也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话，核心类是 IdleStateHandler。"7.Netty 的零拷贝了解么？"维基百科是这样介绍零拷贝的：“零复制（英语：Zero-copy；也译零拷贝）技术是指计算机执行操作时，CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。在 OS 层面上的 Zero-copy 通常指避免在用户态(User-space)与内核态(Kernel-space)之间来回拷贝数据。而在 Netty 层面，零拷贝主要体现在对于数据操作的优化。Netty 中的零拷贝体现在以下几个方面：使用 Netty 提供的 CompositeByteBuf 类,可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf,避免了各个 ByteBuf 之间的拷贝。ByteBuf 支持 slice 操作,因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf,避免了内存的拷贝。通过 FileRegion 包装的 FileChannel.tranferTo 实现文件传输,可以直接将文件缓冲区的数据发送到目标 Channel,避免了传统通过循环 write 方式导致的内存拷贝问题。"8.Netty 的应用场景了解么？"Netty 主要用来做网络通信:作为 RPC 框架的网络通信工具：我们在分布式系统中，不同服务节点之间经常需要相互调用，这个时候就需要 RPC 框架了。不同服务节点之间的通信是如何做的呢？可以使用 Netty 来做。比如我调用另外一个节点的方法的话，至少是要让对方知道我调用的是哪个类中的哪个方法以及相关参数吧！实现一个自己的 HTTP 服务器：通过 Netty 我们可以自己实现一个简单的 HTTP 服务器，这个大家应该不陌生。说到 HTTP 服务器的话，作为 Java 后端开发，我们一般使用 Tomcat 比较多。一个最基本的 HTTP 服务器可要以处理常见的 HTTPMethod 的请求，比如 POST 请求、GET 请求等等。实现一个即时通讯系统：使用 Netty 我们可以实现一个可以聊天类似微信的即时通讯系统，这方面的开源项目还蛮多的，可以自行去 Github 找一找。实现消息推送系统：市面上有很多消息推送系统都是基于 Netty 来做的。"9.为什么要用 Netty？"统一的 API，支持多种传输类型，阻塞和非阻塞的。简单而强大的线程模型。自带编解码器解决 TCP 粘包/拆包问题。自带各种协议栈。真正的无连接数据包套接字支持。比直接使用 Java 核心 API 有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。安全性不错，有完整的 SSL/TLS 以及 StartTLS 支持。社区活跃成熟稳定，经历了大型项目的使用和考验，而且很多开源项目都使用到了 Netty，比如我们经常接触的 Dubbo、RocketMQ 等等。"10.Netty 是什么？"Netty 是一个基于 NIO 的 client-server(客户端服务器)框架，使用它可以快速简单地开发网络应用程序。它极大地简化并优化了 TCP 和 UDP 套接字服务器等网络编程,并且性能以及安全性等很多方面甚至都要更好。支持多种协议如 FTP，SMTP，HTTP 以及各种二进制和基于文本的传统协议。用官方的总结就是：Netty 成功地找到了一种在不妥协可维护性和性能的情况下实现易于开发，性能，稳定性和灵活性的方法。"11.UDP 协议会有粘包拆包的问题吗？为什么？"UDP 不会有这个问题。因为 TCP 是“数据流”协议，UDP 是“数据报”协议。UDP 协议的数据包之间是没有联系，而且有明确边界的。"12.了解过粘包拆包吗？为什么会出现粘包拆包？怎么处理粘包拆包？"粘包的主要原因：发送方写入数据套接字缓冲区大小；发送方发送的数据大于协议的 MTU（最大传输单元），不得已必须拆包。如何处理：1、消息长度固定；2、消息之间用分隔符分隔；3、在消息头保留一个字段，用于描述消息的长度。"13.什么是 Reactor 模型？Reactor 的 3 种版本都知道吗？"Reactor 模式究竟是个什么东西呢？这要从事件驱动的开发方式说起。我们知道，对于应用服务器，一个主要规律就是，CPU 的处理速度是要远远快于 IO 速度的，如果 CPU 为了 IO 操作（例如从 Socket 读取一段数据）而阻塞显然是不划算的。好一点的方法是分为多进程或者线程去进行处理，但是这样会带来一些进程切换的开销，试想一个进程一个数据读了 500ms，期间进程切换到它 3 次，但是 CPU 却什么都不能干，就这么切换走了，是不是也不划算？这时先驱们找到了事件驱动，或者叫回调的方式，来完成这件事情。这种方式就是，应用业务向一个中间人注册一个回调（eventhandler），当 IO 就绪后，就这个中间人产生一个事件，并通知此 handler 进行处理。这种回调的方式，也体现了“好莱坞原则”（Hollywoodprinciple）-“Don’tcallus,we’llcallyou”，在我们熟悉的 IoC 中也有用到。看来软件开发真是互通的！好了，我们现在来看 Reactor 模式。在前面事件驱动的例子里有个问题：我们如何知道 IO 就绪这个事件，谁来充当这个中间人？Reactor 模式的答案是：由一个不断等待和循环的单独进程（线程）来做这件事，它接受所有 handler 的注册，并负责先操作系统查询 IO 是否就绪，在就绪后就调用指定 handler 进行处理，这个角色的名字就叫做 Reactor。Reactor 的 3 种版本：单线程模式、多线程模式、主从多线程模式"九、RabbitMQ1.vhost 是什么？起什么作用？vhost 可以理解为虚拟 broker，即 mini-RabbitMQserver。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。2.RabbitMQ 有几种广播类型?"①fanout：所有 bind 到此 exchange 的 queue 都可以接收消息(纯广播，绑定到 RabbitMQ 的接受者都能收到消息);②direct：通过 routingKey 和 exchange 决定的那个唯一的 queue 可以接收消息;③topic：所有符合 routingKey(此时可以是一

个表达式)的 routingKey 所 bind 的 queue 可以接收消息;"3.要保证消息持久化成功的条件有哪些?"①声明队列必须设置持久化 durable 设置为 true。②消息推送投递模式必须设置持久化，deliveryMode 设置为 2(持久)。③消息已经到达持久化交换器。④消息已经到达持久化队列。以上四个条件都满足才能保证消息持久化成功。"4.如何确保消息正确地发送至 RabbitMQ?如何确保消息接收方消费了消息?"1、发送方确认模式①将信道设置成 confirm 模式(发送方确认模式)，则所有在信道上发布的消息都会被指派一个唯一的 ID。②一旦消息被投递到目的队列后，或者消息被写入磁盘后(可持久化的消息)，信道会发送一个确认给生产者(包含消息唯一 ID)。③如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack(notacknowledged，未确认)消息。④发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。2、接收方确认机制①消费者接收每一条消息后都必须进行确认(消息接收和消息确认是两个不同操作)。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。②这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。保证数据的最终一致性。3、下面罗列几种特殊情况①如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。(可能存在消息重复消费的隐患，需要去重)②如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。"5.RabbitMQ 有哪些重要的角色?"RabbitMQ 中重要的角色有：生产者、消费者和代理：①生产者：消息的创建者，负责创建和推送数据到消息服务器;②消费者：消息的接收方，用于处理数据和确认消息;③代理：就是 RabbitMQ 本身，用于扮演“快递”的角色，本身不生产消息，只是扮演“快递”的角色。"6.RabbitMQ 的使用场景有哪些?"①跨系统的异步通信，所有需要异步交互的地方都可以使用消息队列。就像我们除了打电话(同步)以外，还需要发短信，发电子邮件(异步)的通讯方式。②多个应用之间的耦合，由于消息是平台无关和语言无关的，而且语义上也不再是函数调用，因此更适合作为多个应用之间的松耦合的接口。基于消息队列的耦合，不需要发送方和接收方同时在线。在企业应用集成(EAI)中，文件传输，共享数据库，消息队列，远程过程调用都可以作为集成的方法。③应用内的同步变异步，比如订单处理，就可以由前端应用将订单信息放到队列，后端应用从队列里依次获得消息处理，高峰时的大量订单可以积压在队列里慢慢处理掉。由于同步通常意味着阻塞，而大量线程的阻塞会降低计算机的性能。④消息驱动的架构(EDA)，系统分解为消息队列，和消息制造者和消息消费者，一个处理流程可以根据需要拆成多个阶段(Stage)，阶段之间用队列连接起来，前一个阶段处理的结果放入队列，后一个阶段从队列中获取消息继续处理。⑤应用需要更灵活的耦合方式，如发布订阅，比如可以指定路由规则。⑥跨局域网，甚至跨城市的通讯(CDN 行业)，比如北京机房与广州机房的应用程序的通信。"7.RabbitMQ 怎么避免消息丢失？"①消息持久化;②ACK 确认机制;③设置集群镜像模式;④消息补偿机制。"8.RabbitMQ 的消息是怎么发送的？首先客户端必须连接到 RabbitMQ 服务器才能发布和消费消息，客户端和 rabbitserver 之间会创建一个 tcp 连接，一旦 tcp 打开并通过了认证（认证就是你发送给 rabbit 服务器的用户名和密码），你的客户端和 RabbitMQ 就创建了一条 amqp 信道（channel），信道是创建在“真实”tcp 上的虚拟连接，amqp 命令都是通过信道发送出去的，每个信道都会有一个唯一的 id，不论是发布消息，订阅队列都是通过这个信道完成的。9.若 cluster 中拥有某个 queue 的 ownernode 失效了，且该 queue 被声明具有 durable 属性，是否能够成功从其他 node 上重新声明该 queue？不能，在这种情况下，将得到 404NOT_FOUND 错误。只能等 queue 所属的 node 恢复后才能使用该 queue。但若该 queue 本身不具有 durable 属性，则可在其他 node 上重新声明。10.客户端连接到 cluster 中的任意 node 上是否都能正常工作？是的。客户端感觉不到有何不同。11.在单 node 系统和多 node 构成的 cluster 系统中声明 queue、exchange，以及进行 binding 会有什么不同？当你在单 node 上声明 queue 时，只要该 node 上相关元数据进行了变更，你就会得到 Queue.Declare-ok 回应；而在 cluster 上声明 queue，则要求 cluster 上的全部 node 都要进行元数据成功更新，才会得到 Queue.Declare-ok 回应。另外，若 node 类型为 RAMnode 则变更的数据仅保存在内存中，若类型为 disknode 则还要变更保存在磁盘上的数据。12.什么是元数据？元数据分为哪些类型？包括哪些内容？与 cluster 相关的元数据有哪些？元数据是如何保存的？元数据在 cluster 中是如何分布的？"在非 cluster 模式下，元数据主要分为 Queue 元数据（queue 名字和属性等）、Exchange 元数据（exchange 名字、类型和属性等）、Binding 元数据（存放路由关系的查找表）、Vhost 元数据（vhost 范围内针对前三者的名字空间约束和安全属性设置）。在 cluster 模式下，还包括 cluster 中 node 位置信息和 node 关系信息。元数据按照 erlangnode 的类型确定是仅保存于 RAM 中，还是同时保存在 RAM 和 disk 上。元数据在 cluster 中是全 node 分布的。"13.RabbitMQ 有什么优缺点？"优点：解耦、异步、削峰；缺点：降低了系统的稳定性：本来系统运行好好的，现在你非要加入个消息队列进去，那消息队列挂了，你的系统不是呵呵了。因此，系统可用性会降低；增加了系统的复杂性：加入了消息队列，要多考虑很多方面的问题，比如：一致性问题、如何保证消息不被重复消费、如何保证消息可靠性传输等。因此，需要考虑的东西更多，复杂性增大。"14.什么是 RabbitMQ？为什么使用 RabbitMQ？"RabbitMQ 是一款开源的，Erlang 编写的，基于 AMQP 协议的，消息中间件；可以用它来：解耦、异步、削峰。"15.死信队列和延迟队列的使用"死信消息：消息被拒绝（Basic.Reject 或 Basic.Nack）并且设置 requeue 参数的值为 false 消息过期了队列达到最大的长度过期消息：在 rabbitmq 中存在 2 种方可设置消息的过期时间，第一种通过对队列进行设置，这种设置后，该队列中所有的消息都存在相同的过期时间，第二种通过对消息本身进行设置，那么每条消息的过期时间都不一样。如果同时使用这 2 种方法，那么以过期时间小的那个数值为准。当消息达到过期时间还没有被消费，那么那个消息就成为了一个死信消息。队列设置：在队列申明的时候使用 x-message-ttl 参数，单位为毫秒单个消息设置：是设置消息属性的 expiration 参数的值，单位为毫秒延时队列：在 rabbitmq 中不存在延时队列，但是我们可以通过设置消息的过期时间和死信队列来模拟出延时队列。消费者监听死信交换器绑定的队列，而不要监听消息发送的队列。有了以上的基础知识，我们完成以下需求：需求：用户在系统中创建一个订单，如果超过时间用户没有进行支付，那么自动取消订单。分析：1、上面这个情况，我们就适合使用延时队列来实现，那么延时队列如何创建 2、延时队列可以由过期消息+死信队列来时间 3、过期消息通过队列中设置 x-message-ttl 参数实现 4、死信队列通过在队列申明时，给队列设置 x-dead-letter-exchange 参数，然后另外申明一个队列绑定 x-dead-letter-exchange 对应的交换器。"16.如何避免消息重复投递或重复消费？"在消息生产时，MQ 内部针对每条生产者发送的消息生成一个 inner-msg-id，作为去重和幂等的依据（消息投递失败并重传），避免重复的消息进入队列；在消息消费时，要求消息体中必须要有一个 bizId（对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等）作为去重和幂等的依据，避免同一条消息被重复消费。这个问题针对业务场景来答分以下几点：1.比如，你拿到这个消息做数据库的 insert 操作。那就容易了，给这个消息做一个唯一主键，那么就算出现重复消费的情况，就会导致主键冲突，避免数据库出现脏数据。2.再比如，你拿到这个消息做 redis 的 set 的操作，那就容易了，不用解决，因为你无论 set 几次结果都是一样的，set 操作本来就算幂等操作。3.如果上面两种情况还不行，上大招。准备一个第三方介质,来做消费记录。以 redis 为例，给消息分配一个全局 id，只要消费过该消息，将<id,message>以 K-V 形式写入 redis。那消费者开始消费前，先去 redis 中查询有没消费记录即可。"17.如何确保消息接收方消费了消息？"接收方消息确认机制：消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。下面罗列几种特殊情况：如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。(可能存在消息重复消费的隐患，需要根据 bizId 去重）如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。"18.消息怎么路由？"从概念上来说，消息路由必须有三部分：交换器、路由、绑定。生产者把消息发布到交换器上；绑定决定了消息如何从路由器路由到特定的队列；消息最终到达队列，并被消费者接收。消息发布到交换器时，消息将拥有一个路由键（routingkey），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。消息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）。如果能够匹配到队列，则消息会投递到相应队列中；如果不能匹配到任何队列，消息将进入“黑洞”。常用的交换器主要分为一下三种：direct：如果路由键完全匹配，消息就被投递到相应的队列 fanout：如果交换器收到消息，将会广播到所有绑定的队列上 topic：可以使来自不同源头的消息能够到达同一个队列。使用 topic 交换器时，

可以使用通配符，比如：“*”匹配特定位置的任意文本，“.”把路由键分为了几部分，“#”匹配所有规则等。特别注意：发往 topic 交换器的消息不能随意的设置选择键（routing_key），必须是由""."隔开的一系列的标识符组成。"19.消息如何分发？若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。20.消息基于什么传输？由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。十、Redis1.Redis 如何做内存优化？"尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面。"2.Redis 中的管道有什么用？"一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。这就是管道（pipelining），是一种几十年来广泛使用的技术。例如许多 POP3 协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。"3.Redis 和 Redisson 有什么关系？Redisson 是一个高级的分布式协调 Redis 客服端，能帮助用户在分布式环境中轻松实现一些 Java 的对象(Bloomfilter,BitSet,Set,SetMultimap,ScoredSortedSet,SortedSet,Map,ConcurrentMap,List,ListMultimap,Queue,BlockingQueue,Deque,BlockingDeque,Semaphore,Lock,ReadWriteLock,AtomicLong,CountDownLatch,Publish/Subscribe,HyperLogLog)。4.Redis 有哪些适合的场景？"（1）会话缓存（SessionCache）最常用的一种使用 Redis 的情景是会话缓存（sessioncache）。用 Redis 缓存会话比其他存储（如 Memcached）的优势在于：Redis 提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用 Redis 来缓存会话的文档。甚至广为人知的商业平台 Magento 也提供 Redis 的插件。（2）全页缓存（FPC）除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。再次以 Magento 为例，Magento 提供一个插件来使用 Redis 作为全页缓存后端。此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。（3）队列 Reids 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。如果你快速的在 Google 中搜索“Redisqueues”，你马上就能找到大量的开源项目，这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需求。例如，Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去查看。（4）排行榜/计数器 Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（SortedSet）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的 10 个用户–我们称之为“user_scores”，我们只需要像下面一样执行即可：当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：ZRANGEuser_scores010WITHSCORESAgoraGames 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。（5）发布/订阅最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！"5.MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？"redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。其实面试除了考察 Redis，不少公司都很重视高并发高可用的技术，特别是一线互联网公司，分布式、JVM、spring 源码分析、微服务等知识点已是面试的必考题。"6.Redis 集群方案什么情况下会导致整个集群不可用？有 A，B，C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了，那么整个集群就会以为缺少 5501-11000 这个范围的槽而不可用。7.Redis 集群方案应该怎么做？都有哪些方案？"codis 目前用的最多的集群方案，基本和 twemproxy 一致的效果，但它支持在节点数量改变情况下，旧节点数据可恢复到新 hash 节点。rediscluster3.0 自带的集群，特点在于他的分布式算法不是一致性 hash，而是 hash 槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。在业务代码层实现，起几个毫无关联的 redis 实例，在代码层，对 key 进行 hash 计算，然后去对应的 redis 实例操作数据。这种方式对 hash 层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。"8.RedisString 的内部编码有哪些？"int、embstr、raw10000 以下的整数会使用缓存里的 int 常量。长度小于等于 44 字节：embstr 编码长度大于 44 字节：raw 编码"9.用 Redis 做延时队列，具体应该怎么实现？可以使用 Zset 实现。member 是任务描述，score 是执行时间，然后用定时器定时去扫描，一旦有执行时间小于或等于当前时间的任务，就立即执行。10.Redis 在集群种查找 key 的时候，是怎么定位到具体节点的？使用 crc16 算法对 key 进行 hash 将 hash 值对 16384 取模，得到具体的槽位根据节点和槽位的映射信息（与集群建立连接后，客户端可以取得槽位映射信息），找到具体的节点地址去具体的节点找 key 如果 key 不在这个节点上，则 redis 集群会返回 moved 指令，加上新的节点地址给客户端，同时，客户端会刷新本地的节点槽位映射关系如果槽位正在迁移中，那么 redis 集群会返回 asking 指令给客户端，这是临时纠正，客户端不会刷新本地的节点槽位映射关系 11.Redis 的持久化了解过吗？"Redis 持久化有 RDB 和 AOF 这 2 种方式。RDB：将数据库快照以二进制的方式保存到磁盘中。AOF：以协议文本方式，将所有对数据库进行过写入的命令和参数记录到 AOF 文件，从而记录数据库状态。"12.Redis 在什么情况下会触发 key 的回收？2 种情况：1、定时（抽样）清理；2、执行命令时，判断内存是否超过 maxmemory。13.Rediskey 的淘汰策略有哪些？8 种：noeviction，volatile-lru，volatile-lfu，volatile-ttl，volatile-random，allkeylru，allkeys-lfu，allkeys-random14.Redis 事务机制了解过吗？"Redis 事务的概念：Redis 事务的本质是一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。Redis 事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。Redis 事务没有隔离级别的概念：批量操作在发送 EXEC 命令前被放入队列缓存，并不会被实际执行，也就不存在事务内的查询要看到事务里的更新，事务外查询不能看到。Redis 不保证原子性：Redis 中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。Redis 事务的三个阶段：开始事务命令入队执行事务 Redis 事务相关命令：watchkey1key2...:监视一或多个 key,如果在事务执行之前，被监视的 key 被其他命令改动，则事务被打断（类似乐观锁）multi:标记一个事务块的开始（queued）exec:执行所有事务块的命令（一旦执行 exec 后，之前加的监控锁都会被取消掉）discard:取消事务，放弃事务块中的所有命令 unwatch:取消 watch 对所有 key 的监控"15.使用 Redis 统计网站的 UV，应该怎么做？"UV 与 PV 不同，UV 需要去重。一般有 2 种方案：1、用 BitMap。存的是用户的 uid，计算 UV 的时候，做下 bitcount 就行了。2、用布隆过滤器。将每次访问的用户 uid 都放到布隆过滤器中。优点是省内存，缺点是无法得到精确的 UV。但是对于不需要精确知道具体 UV，只需要大概的数量级的场景，是个不错的选择。"16.Redis 中的大 key 怎么处理？"大 key 指的是 value 特别大的 key。比如很长的字符串，或者很大的 set 等等。大 key 会造成 2 个问题：1、数据倾斜，比如某些节点内存占用过高。2、当删除大 key 或者大 key 自动过期的时候，会造成 QPS 突降，因为 Redis 是单线程的缘故。处理方案：可以将一个大 key 进行分片处理，比如：将一个大 set 分成多个小的 set。"17.Redis 中的热 key 怎么处理？"1、对热 key 进行分散处理。比如：在 key 上加上不同的前后缀，缓存多个 key，使得各个 key 分散到不同的节点上。2、采用多级缓存。"18.缓存失效？缓存穿透？缓存雪崩？缓存并发？"缓存失效缓存失效指的是大量的缓存在同一时间失效，到时 DB 的瞬间压力飙升。造成这种现象的原因是，key 的过期时间都设置成一样了。解决方案是，key 的过期时间引入随机因素，比如 5 分钟+随机秒这种方式。缓存穿透缓存穿透是指查询一条数据库和缓存都没有的一条数据，就会一直查询数据库，对数据库的访问压力就会增大，缓存穿透的解决方案，有以下 2 种：缓存空对象：代码维护较简单，但是效果不好。布隆过滤器：代码维护复杂，效果很好。缓存雪崩缓存雪崩是指在某一个时间段，缓存集中过期失效。此刻无数的请求直接绕开缓存，直接请求数据库。造成缓存雪崩的原因，有以下 2 种：reids 宕机。大部分数据失效。对于缓存雪崩的解决方案有以下 2 种：搭建高可用的集群，防止单机的 redis 宕机。设置不同的过期时间，防止同意之间内大量的 key 失效。缓存并发有时候如果网站并发访问高，一个缓存如果失效，可能出现多个进程同时查询 DB，同时设置缓存的情况，如果并发确实很大，这也可能造成 DB

压力过大，还有缓存频繁更新的问题。一般处理方案是在查 DB 的时候进行加锁，如果 KEY 不存在，就加锁，然后查 DB 入缓存，然后解锁；其他进程如果发现有锁就等待，然后等解锁后再查缓存或者进入 DB 查询。"19.Redis 集群如何选择数据库？Redis 集群目前无法做数据库选择，默认在 0 数据库。20.Redis 如何设置密码及验证密码？"设置密码：configsetrequirepass123456 授权密码：auth123456"21.为什么 Redis 需要把所有数据放到内存中？"Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征，如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。在内存越来越便宜的今天，redis 将会越来越受欢迎，如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。"22.Redis 官方为什么不提供 Windows 版本？因为目前 Linux 版本已经相当稳定，而且用户量很大，无需开发 windows 版本，反而会带来兼容性等问题。23.Redis 是单线程还是多线程？"Redis6.0 采用多线程 IO，不过命令的执行还是单线程的。Redis6.0 之前，IO 线程和执行线程都是单线程的。"24.Redis 为什么那么快？"1、内存操作；2、单线程，省去线程切换、锁竞争的开销；3、非阻塞 IO 模型，epoll。"25.一个字符串类型的值能存储最大容量是多少？512M26.Redis 的全称是什么？RemoteDictionaryServer。27.Redis 主要消耗什么物理资源？内存。28.Redis 有哪些数据结构？"Redis 有 5 种基础数据结构，它们分别是：string(字符串)、list(列表)、hash(字典)、set(集合)和 zset(有序集合)。这 5 种是 Redis 相关知识中最基础、最重要的部分。"29.Redis 相比 memcached 有哪些优势？"(1)memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类型(2)redis 的速度比 memcached 快很多(3)redis 可以持久化其数据"30.什么是 Redis？简述它的优缺点？"Redis 本质上是一个 Key-Value 类型的内存数据库，很像 memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 flush 到硬盘上进行保存。因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操作，是已知性能最快的 Key-ValueDB。Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构，此外单个 value 的最大限制是 1GB，不像 memcached 只能保存 1MB 的数据，因此 Redis 可以用来实现很多有用的功能。比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性能消息队列服务，用他的 Set 可以做高性能的 tag 系统等等。另外 Redis 也可以对存入的 Key-Value 设置 expire 时间，因此也可以被当作一个功能加强版的 memcached 来用。Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。"十一、Spring1.@Component 和@Bean 的区别是什么？"1.作用对象不同。@Component 注解作用于类，而@Bean 注解作用于方法。2.@Component 注解通常是通过类路径扫描来自动侦测以及自动装配到 Spring 容器中（我们可以使用@ComponentScan 注解定义要扫描的路径）。@Bean 注解通常是在标有该注解的方法中定义产生这个 bean，告诉 Spring 这是某个类的实例，当我需要用它的时候还给我。3.@Bean 注解比@Component 注解的自定义性更强，而且很多地方只能通过@Bean 注解来注册 bean。比如当引用第三方库的类需要装配到 Spring 容器的时候，就只能通过@Bean 注解来实现。"2.Spring 框架中用到了哪些设计模式？"1.工厂设计模式：Spring 使用工厂模式通过 BeanFactory 和 ApplicationContext 创建 bean 对象。2.代理设计模式：SpringAOP 功能的实现。3.单例设计模式：Spring 中的 bean 默认都是单例的。4.模板方法模式：Spring 中的 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。5.包装器设计模式：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。6.观察者模式：Spring 事件驱动模型就是观察者模式很经典的一个应用。7.适配器模式：SpringAOP 的增强或通知（Advice）使用到了适配器模式、SpringMVC 中也是用到了适配器模式适配 Controller。"3.SpringMVC 的工作原理了解嘛？"1.客户端（浏览器）发送请求，直接请求到 DispatcherServlet。2.DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。3.解析到对应的 Handler（也就是我们平常说的 Controller 控制器）。4.HandlerAdapter 会根据 Handler 来调用真正的处理器来处理请求和执行相对应的业务逻辑。5.处理器处理完业务后，会返回一个 ModelAndView 对象，Model 是返回的数据对象，View 是逻辑上的 View。6.ViewResolver 会根据逻辑 View 去查找实际的 View。7.DispatcherServlet 把返回的 Model 传给 View（视图渲染）。8.把 View 返回给请求者（浏览器）。"4.Spring 中的 bean 生命周期了解过吗？"1.Bean 容器找到配置文件中 SpringBean 的定义。2.Bean 容器利用 JavaReflectionAPI 创建一个 Bean 的实例。3.如果涉及到一些属性值，利用 set()方法设置一些属性值。4.如果 Bean 实现了 BeanNameAware 接口，调用 setBeanName()方法，传入 Bean 的名字。5.如果 Bean 实现了 BeanClassLoaderAware 接口，调用 setBeanClassLoader()方法，传入 ClassLoader 对象的实例。6.如果 Bean 实现了 BeanFactoryAware 接口，调用 setBeanClassFacotory()方法，传入 ClassLoader 对象的实例。7.与上面的类似，如果实现了其他*Aware 接口，就调用相应的方法。8.如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessBeforeInitialization()方法。9.如果 Bean 实现了 InitializingBean 接口，执行 afeterPropertiesSet()方法。10.如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。11.如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcess 对象，执行 postProcessAfterInitialization()方法。12.当要销毁 Bean 的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy()方法。13.当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。"5.Bean 工厂和 Applicationcontexts 有什么区别？Applicationcontexts 提供一种方法处理文本消息，一个通常的做法是加载文件资源（比如镜像），它们可以向注册为监听器的 bean 发布事件。另外，在容器或容器内的对象上执行的那些不得不由 bean 工厂以程序化方式处理的操作，可以在 Applicationcontexts 中以声明的方式处理。Applicationcontexts 实现了 MessageSource 接口，该接口的实现以可插拔的方式提供获取本地化消息的方法。6.ApplicationContext 通常的实现是什么?"FileSystemXmlApplicationContext：此容器从一个 XML 文件中加载 beans 的定义，XMLBean 配置文件的全路径名必须提供给它的构造函数。ClassPathXmlApplicationContext：此容器也从一个 XML 文件中加载 beans 的定义，这里，你需要正确设置 classpath 因为这个容器将在 classpath 里找 bean 配置。WebXmlApplicationContext：此容器加载一个 XML 文件，此文件定义了一个 WEB 应用的所有 bean。"7.SpringAOP 实现原理"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。SpringAOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice 和回调目标对象的方法所组成,并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异，AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。SpringAOP 使用动态代理技术在运行期织入增强代码。使用两种代理机制：基于 JDK 的动态代理（JDK 本身只提供接口的代理）和基于 CGlib 的动态代理。(1)JDK 的动态代理 JDK 的动态代理主要涉及 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。其中 InvocationHandler 只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而 Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。其代理对象必须是某个接口的实现,它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理.只能实现接口的类生成代理，而不能针对类。(2)CGLibCGLib 采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑.它运行期间生成的代理对象是目标类的扩展子类.所以无法通知 final、private 的方法,因为它们不能被覆写.是针对类实现代理,主要是为指定的类生成一个子类，覆盖其中方法。在 spring 中默认。况下使用 JDK 动态代理实现 AOP,如果 proxy-target-class 设置为 true 或者使用了优化策略那么会使用 CGLIB 来创建动态代理.SpringAOP 在这两种方式的实现上基本一样．以 JDK 代理为例，会使用 JdkDynamicAopProxy 来创建代理，在 invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入，最终返回代理对象。"8.有哪些不同类型的 IOC(依赖注入)？"构造器依赖注入：构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。Setter 方法依赖注入：首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用 bean 中的 setter 方法完成 Setter 方法依赖注入。"9.解释自动装配的各种模式？"自动装配提供五种不同的模式供 Spring 容器用来自动装配 beans 之间的依赖注入:no：默认的方式是不进行自动装配，通过手工设置 ref 属性来进行装配 bean。byName：通过参数名自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byName。之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。

byType：通过参数的数据类型自动自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byType。之后容器试图匹配和装配和该 bean 的属性类型一样的 bean。如果有多个 bean 符合条件，则抛出错误。constructor：这个同 byType 类似，不过是应用于构造函数的参数。如果在 BeanFactory 中不是恰好有一个 bean 与构造函数参数相同类型，则抛出一个严重的错误。autodetect：如果有默认的构造方法，通过 construct 的方式自动装配，否则使用 byType 的方式自动装配。"10.Resource 是如何被查找、加载的？"Resource 接口是 Spring 资源访问策略的抽象，它本身并不提供任何资源访问实现，具体的资源访问由该接口的实现类完成——每个实现类代表一种资源访问策略。Spring 为 Resource 接口提供了如下实现类：UrlResource：访问网络资源的实现类。ClassPathResource：访问类加载路径里资源的实现类。FileSystemResource：访问文件系统里资源的实现类。ServletContextResource：访问相对于 ServletContext 路径里的资源的实现类：InputStreamResource：访问输入流资源的实现类。ByteArrayResource：访问字节数组资源的实现类。这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。"11.BeanFactory 和 ApplicationContext 有什么区别？"ApplicationContext 提供了一种解决文档信息的方法，一种加载文件资源的方式(如图片)，他们可以向监听他们的 beans 发送消息。另外，容器或者容器中 beans 的操作，这些必须以 bean 工厂的编程方式处理的操作可以在应用上下文中以声明的方式处理。应用上下文实现了 MessageSource，该接口用于获取本地消息，实际的实现是可选的。相同点：两者都是通过 xml 配置文件加载 bean，ApplicationContext 和 BeanFacotry 相比，提供了更多的扩展功能。不同点：BeanFactory 是延迟加载，如果 Bean 的某一个属性没有注入，BeanFacotry 加载后，直至第一次使用调用 getBean 方法才会抛出异常；而 ApplicationContext 则在初始化自身是检验，这样有利于检查所依赖属性是否注入；所以通常情况下我们选择使用 ApplicationContext。"12.Spring 事务底层原理"划分处理单元——IoC 由于 spring 解决的问题是对单个数据库进行局部事务处理的，具体的实现首先用 spring 中的 IoC 划分了事务处理单元。并且将对事务的各种配置放到了 ioc 容器中（设置事务管理器，设置事务的传播特性及隔离机制）。AOP 拦截需要进行事务处理的类 Spring 事务处理模块是通过 AOP 功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用 TransactionProxyFactoryBean 接口来使用 AOP 功能，生成 proxy 代理对象，通过 TransactionInterceptor 完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。读取 ioc 容器事务配置属性，转化为 spring 事务处理需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为 TransactionAttribute 表示的数据对象。对事务处理实现（事务的生成、提交、回滚、挂起）spring 委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource 数据源支持、hibernate 数据源事务处理支持、JDO 数据源事务处理支持，JPA、JTA 数据源事务处理支持。这些支持都是通过设计 PlatformTransactionManager、AbstractPlatforTransaction 一系列事务处理的支持。为常用数据源支持提供了一系列的 TransactionManager。总结 PlatformTransactionManager 实现了 TransactionInterception 接口，让其与 TransactionProxyFactoryBean 结合起来，形成一个 Spring 声明式事务处理的设计体系。"13.Spring 事务中有哪几种事务传播行为？"在 TransactionDefinition 接口中定义了八个表示事务传播行为的常量。支持当前事务的情况：PROPAGATION_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）。不支持当前事务的情况：PROPAGATION_REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。其他情况：PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 PROPAGATION_REQUIRED。"14.Spring 事务中的隔离级别有哪几种？"在 TransactionDefinition 接口中定义了五个表示隔离级别的常量：ISOLATION_DEFAULT：使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ 隔离级别；Oracle 默认采用的 READ_COMMITTED 隔离级别。ISOLATION_READ_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。ISOLATION_READ_COMMITTED：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生 ISOLATION_REPEATABLE_READ：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。ISOLATION_SERIALIZABLE：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。"15.Spring 事务管理的方式有几种？"1.编程式事务：在代码中硬编码（不推荐使用）。2.声明式事务：在配置文件中配置（推荐使用），分为基于 XML 的声明式事务和基于注解的声明式事务。"16.将一个类声明为 Spring 的 bean 的注解有哪些？"我们一般使用@Autowired 注解去自动装配 bean。而想要把一个类标识为可以用@Autowired 注解自动装配的 bean，可以采用以下的注解实现：1.@Component 注解。通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于哪一个层，可以使用@Component 注解标注。2.@Repository 注解。对应持久层，即 Dao 层，主要用于数据库相关操作。3.@Service 注解。对应服务层，即 Service 层，主要涉及一些复杂的逻辑，需要用到 Dao 层（注入）。4.@Controller 注解。对应 SpringMVC 的控制层，即 Controller 层，主要用于接受用户请求并调用 Service 层的方法返回数据给前端页面。"17.Spring 中的单例 bean 的线程安全问题了解吗？"大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。有两种常见的解决方案：1.在 bean 对象中尽量避免定义可变的成员变量（不太现实）。2.在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。"18.Spring 中的 bean 的作用域有哪些？"1.singleton：唯一 bean 实例，Spring 中的 bean 默认都是单例的。2.prototype：每次请求都会创建一个新的 bean 实例。3.request：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTPrequest 内有效。4.session：每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTPsession 内有效。5.global-session：全局 session 作用域，仅仅在基于 Portlet 的 Web 应用中才有意义，Spring5 中已经没有了。Portlet 是能够生成语义代码（例如 HTML）片段的小型 JavaWeb 插件。它们基于 Portlet 容器，可以像 Servlet 一样处理 HTTP 请求。但是与 Servlet 不同，每个 Portlet 都有不同的会话。"19.谈谈自己对于 SpringAOP 的理解"AOP（Aspect-OrientedProgramming，面向切面编程）能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可扩展性和可维护性。SpringAOP 是基于动态代理的，如果要代理的对象实现了某个接口，那么 SpringAOP 就会使用 JDK 动态代理去创建代理对象；而对于没有实现接口的对象，就无法使用 JDK 动态代理，转而使用 CGlib 动态代理生成一个被代理对象的子类来作为代理。当然也可以使用 AspectJ，SpringAOP 中已经集成了 AspectJ，AspectJ 应该算得上是 Java 生态系统中最完整的 AOP 框架了。使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样可以大大简化代码量。我们需要增加新功能也方便，提高了系统的扩展性。日志功能、事务管理和权限管理等场景都用到了 AOP。"20.SpringAOP 和 AspectJAOP 有什么区别？"SpringAOP 是属于运行时增强，而 AspectJ 是编译时增强。SpringAOP 基于代理（Proxying），而 AspectJ 基于字节码操作（BytecodeManipulation）。SpringAOP 已经集成了 AspectJ，AspectJ 应该算得上是 Java 生态系统中最完整的 AOP 框架了。AspectJ 相比于 SpringAOP 功能更加强大，但是 SpringAOP 相对来说更简单。如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ，它比 SpringAOP 快很多。"21.谈谈自己对于 SpringIOC 的理解"IOC（InversionOfControll，控制反转）是一种设计思想，就是将原本在程序中手动创建对象的控制权，交由给 Spring 框架来管理。IOC 在其他语言中也有应用，并非 Spring 特有。IOC 容器是 Spring 用来实现 IOC 的载体，IOC 容器实际上就是一个 Map(key,value)，Map 中存放的是各种对象。将对象之间的相互依赖关系交给 IOC 容器来管理，并由 IOC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。IOC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。在实际项目中一个 Service 类可能由几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，可能要每次都

搞清楚这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IOC 的话，你只需要配置好，然后在需要的地方引用就行了，大大增加了项目的可维护性且降低了开发难度。Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得用 XML 文件来配置不太好，于是 SprngBoot 注解配置就慢慢开始流行起来。"22.SpringBoot 手动装配有哪几种方式？"使用模式注解@Component 等（Spring2.5+）使用配置类@Configuration 与@Bean（Spring3.0+）使用模块装配@EnableXXX 与@Import（Spring3.1+）其中使用@Component 及衍生注解很常见，咱开发中常用的套路，不再赘述。但模式注解只能在自己编写的代码中标注，无法装配 jar 包中的组件。为此可以使用@Configuration 与@Bean，手动装配组件。但这种方式一旦注册过多，会导致编码成本高，维护不灵活等问题。SpringFramework 提供了模块装配功能，通过给配置类标注@EnableXXX 注解，再在注解上标注@Import 注解，即可完成组件装配的效果。"23.Spring 是怎么解决循环依赖的？"整个 IOC 容器解决循环依赖，用到的几个重要成员：singletonObjects：一级缓存，存放完全初始化好的 Bean 的集合，从这个集合中取出来的 Bean 可以立马返回 earlySingletonObjects：二级缓存，存放创建好但没有初始化属性的 Bean 的集合，它用来解决循环依赖 singletonFactories：三级缓存，存放单实例 Bean 工厂的集合 singletonsCurrentlyInCreation：存放正在被创建的 Bean 的集合 IOC 容器解决循环依赖的思路：初始化 Bean 之前，将这个 BeanName 放入三级缓存创建 Bean 将准备创建的 Bean 放入 singletonsCurrentlyInCreation（正在创建的 Bean）createNewInstance 方法执行完后执行 addSingletonFactory，将这个实例化但没有属性赋值的 Bean 放入二级缓存，并从三级缓存中移除属性赋值&自动注入时，引发关联创建关联创建时，检查“正在被创建的 Bean”中是否有即将注入的 Bean。如果有，检查二级缓存中是否有当前创建好但没有赋值初始化的 Bean。如果没有，检查三级缓存中是否有正在创建中的 Bean。至此一般会有，将这个 Bean 放入二级缓存，并从三级缓存中移除之后 Bean 被成功注入，最后执行 addSingleton，将这个完全创建好的 Bean 放入一级缓存，从二级缓存和三级缓存移除，并记录已经创建了的单实例 Bean"24.Spring 由哪些模块组成?"以下是 Spring 框架的基本模块：

CoremoduleBeanmoduleContextmoduleExpressionLanguagemoduleJDBCmoduleORMmoduleOXMmoduleJavaMessagingService(JMS)moduleTransactionmoduleWebmoduleWeb-ServletmoduleWeb-StrutsmoduleWeb-Portletmodule"25.使用 Spring 框架的好处是什么？"轻量：Spring 是轻量的，基本的版本大约 2MB。控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。面向切面的编程(AOP)：Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。容器：Spring 包含并管理应用中对象的生命周期和配置。MVC 框架：Spring 的 WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。异常处理：Spring 提供方便的 API 把具体技术相关的异常（比如由 JDBC，HibernateorJDO 抛出的）转化为一致的 unchecked 异常。"26.什么是 spring?Spring 是个 java 企业级应用的开源开发框架。Spring 主要用来开发 Java 应用，但是有些扩展是针对构建 J2EE 平台的 web 应用。Spring 框架目标是简化 Java 企业级应用开发，并通过 POJO 为基础的编程模型促进良好的编程习惯。十二、SpringBoot1.SpringBoot2.X 有哪些新特性？与 1.X 有什么区别？"配置变更 JDK 版本升级第三方类库升级响应式 Spring 编程支持 HTTP/2 支持配置属性绑定更多改进与加强"2.保护 SpringBoot 应用有哪些方法？"在生产中使用 HTTPS 使用 Snyk 检查你的依赖关系升级到最新版本启用 CSRF 保护使用内容安全策略防止 XSS 攻击"3.SpringBoot 的核心注解是哪些？他由哪几个注解组成的？"启动类上面的注解是@SpringBootApplication，他也是 SpringBoot 的核心注解，主要组合包含了以下 3 个注解：@SpringBootConfiguration：组合了@Configuration 注解，实现配置文件的功能；@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置的功能：

@SpringBootApplication(exclude={DataSourceAutoConfiguration.class})；@ComponentScan：Spring 组件扫描。"4.SpringBoot 中如何解决跨域问题？"跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此推荐在后端通过（CORS，Crossoriginresourcesharing）来解决跨域问题。这种解决方案并非 SpringBoot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS，现在可以通过实现 WebMvcConfigurer 接口然后重 addCorsMappings 方法解决跨域问题。

@ConfigurationpublicclassCorsConfigimplementsWebMvcConfigurer{@OverridepublicvoidaddCorsMappings(CorsRegistryregistry){registry.addMapping(""/**"").allowedOrigins(""*"").allowCredentials(true).allowedMethods(""GET"",""POST"",""PUT"",""DELETE"",""OPTIONS"").maxAge(3600);}}项目中前后端分离部署，所以需要解决跨域的问题。我们使用 cookie 存放用户登录的信息，在 spring 拦截器进行权限控制，当权限不符合时，直接返回给用户固定的 json 结果。当用户登录以后，正常使用；当用户退出登录状态时或者 token 过期时，由于拦截器和跨域的顺序有问题，出现了跨域的现象。我们知道一个 http 请求，先走 filter，到达 servlet 后才进行拦截器的处理，如果我们把 cors 放在 filter 里，就可以优先于权限拦截器执行。

@ConfigurationpublicclassCorsConfig{@BeanpublicCorsFiltercorsFilter(){CorsConfigurationcorsConfiguration=newCorsConfiguration();corsConfiguration.addAllowedOrigin(""*"");corsConfiguration.addAllowedHeader(""*"");corsConfiguration.addAllowedMethod(""*"");corsConfiguration.setAllowCredentials(true);UrlBasedCorsConfigurationSourceurlBasedCorsConfigurationSource=newUrlBasedCorsConfigurationSource();urlBasedCorsConfigurationSource.registerCorsConfiguration(""/**"",corsConfiguration);returnnewCorsFilter(urlBasedCorsConfigurationSource);}}"5.比较一下 SpringSecurity 和 Shiro 各自的优缺点？"由于 SpringBoot 官方提供了大量的非常方便的开箱即用的 Starter，包括 SpringSecurity 的 Starter，使得在 SpringBoot 中使用 SpringSecurity 变得更加容易，甚至只需要添加一个一来就可以保护所有接口，所以如果是 SpringBoot 项目，一般选择 SpringSecurity。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro 和 SpringSecurity 相比，主要有如下特点：SpringSecurity 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架；SpringSecurity 概念复杂，配置繁琐；Shiro 概念简单、配置简单；SpringSecurity 功能强大；Shiro 功能简单"6.如何实现 SpringBoot 应用程序的安全性？为了实现 SpringBoot 的安全性，使用 spring-boot-starter-security 依赖项，并且必须添加安全配置。它只需要很少代码。配置类将必须扩展 WebSecurityConfigurerAdapter 并覆盖其方法。7.什么是 Swagger？你用 SpringBoot 实现了吗？Swagger 广泛用于可视化 API，使用 SwaggerUI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTfulWeb 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。8.如何使用配置文件通过 SpringBoot 配置特定环境的配置？"配置文件不是设别环境的关键。在下面的例子中，我们将会用到两个配置文件 devprod 缺省的应用程序配置在 application.properties 中。让我们来看下面的例子：

basic.value=truebasic.message=DynamicMessagebasic.number=100application.properties 我们想要为 dev 文件自定义 application.properties 属性。我们需要创建一个名为 application-dev.properties 的文件，并且重写我们想要自定义的属性。application-dev.propertiesbasic.message:DynamicMessageinDEV 一旦你特定配置了配置文件，你需要在环境中设定一个活动的配置文件。有多种方法可以做到这一点：在 VM 参数中使用 Dspring.profiles.active=prod 在 application.properties 中使用 spring.profiles.active=prod"9.如何使用 SpringBoot 部署到不同的服务器？"你需要做下面两个步骤：在一个项目中生成一个 war 文件。将它部署到你最喜欢的服务器（websphere 或者 Weblogic 或者 Tomcatandsoon）。第一步：这本入门指南应该有所帮助：https://spring.io/guides/gs/convert-jar-to-war/第二步：取决于你的服务器。"10.如何在 SpringBoot 中添加通用的 JS 代码？"在源文件夹下，创建一个名为 static 的文件夹。然后，你可以把你的静态的内容放在这里面。例如，myapp.js 的路径是 resources\static\js\myapp.js 你可以参考它在 jsp 中的使用方法：image-20210813203850149 错误：HALbrowsergivesmeunauthorizederror–Fullauthenticaitionisrequiredtoaccessthisresource.该如何来修复这个错误呢？image-20210813203857201 两种方法：方法 1：关闭安全验证 application.propertiesmanagement.security.enabled:FALSE 方法二：在日志中搜索密码并传递至请求标头中"11.什么是嵌入式服务器？我们为什么要使用嵌入式服务器呢?"思考一下在你的虚拟机上

部署应用程序需要些什么。第一步：安装 Java 第二部：安装 Web 或者是应用程序的服务器（Tomat/Wbesphere/Weblogic 等等）第三部：部署应用程序 war 包如果我们想简化这些步骤，应该如何做呢？让我们来思考如何使服务器成为应用程序的一部分？你只需要一个安装了 Java 的虚拟机，就可以直接在上面部署应用程序了，是不是很爽？这个想法是嵌入式服务器的起源。当我们创建一个可以部署的应用程序的时候，我们将会把服务器（例如，tomcat）嵌入到可部署的服务器中。例如，对于一个 SpringBoot 应用程序来说，你可以生成一个包含 EmbeddedTomcat 的应用程序 jar。你就可以想运行正常 Java 应用程序一样来运行 web 应用程序了。嵌入式服务器就是我们的可执行单元包含服务器的二进制文件（例如，tomcat.jar）。"12.为什么我们需要 spring-boot-maven-plugin?"spring-boot-maven-plugin 提供了一些像 jar 一样打包或者运行应用程序的命令。spring-boot:run 运行你的 SpringBooty 应用程序。spring-boot：repackage 重新打包你的 jar 包或者是 war 包使其可执行 spring-boot：start 和 spring-boot：stop 管理 SpringBoot 应用程序的生命周期（也可以说是为了集成测试）。spring-boot:build-info 生成执行器可以使用的构造信息。"13.SpringInitializr 是创建 SpringBootProjects 的唯一方法吗？"不是的。SpringInitiatlizr 让创建 SpringBoot 项目变的很容易，但是，你也可以通过设置一个 maven 项目并添加正确的依赖项来开始一个项目。在我们的 Spring 课程中，我们使用两种方法来创建项目。第一种方法是 start.spring.io。另外一种方法是在项目的标题为“BasicWebApplication”处进行手动设置。手动设置一个 maven 项目这里有几个重要的步骤：在 Eclipse 中，使用文件–新建 Maven 项目来创建一个新项目添加依赖项。添加 maven 插件。添加 SpringBoot 应用程序类。到这里，准备工作已经做好！"14.怎么使用 Maven 来构建一个 SpringBoot 程序？"就像引入其他库一样，我们可以在 Maven 工程中加入 SpringBoot 依赖。然而，最好是从 spring-boot-starter-parent 项目中继承以及声明依赖到 SpringBootstarters。这样做可以使我们的项目可以重用 SpringBoot 的默认配置。继承 spring-boot-starter-parent 项目依赖很简单–我们只需要在 pom.xml 中定义一个 parent 节点：<parent><groupId>org.springframework.boot</groupId><artifactId>spring-boot-starter-parent</artifactId><version>2.1.1.RELEASE</version></parent>我们可以在 Mavencentral 中找到 spring-boot-starter-parent 的最新版本。使用 starter 父项目依赖很方便，但并非总是可行。例如，如果我们公司都要求项目继承标准 POM，我们就不能依赖 SpringBootstarter 了。这种情况，我们可以通过对 POM 元素的依赖管理来处理：<dependencyManagement><dependencies><dependency><groupId>org.springframework.boot</groupId><artifactId>spring-boot-dependencies</artifactId><version>2.1.1.RELEASE</version><type>pom</type><scope>import</scope></dependency></dependencies></dependencyManagement>"15.如何在 SpringBoot 中禁用 Actuator 端点安全性？默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 HttpServletRequest.isUserInRole 方法实施的。我们可以使用来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。16.SpringBoot 中的监视器是什么？Springbootactuator 是 spring 启动框架中的重要功能之一。Springboot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTPURL 访问的 REST 端点来检查状态。17.如何重新加载 SpringBoot 上的更改，而无需重新启动服务器？"这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式 tomcat 将重新启动。SpringBoot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 SpringBoot 上的更改，而无需重新启动服务器。这将消除每次手动部署更改的需要。SpringBoot 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。<dependency><groupId>org.springframework.boot</groupId><artifactId>spring-boot-devtools</artifactId><optional>true</optional></dependency>"18.Spring 和 SpringBoot 有什么不同？"Spring 框架提供多种特性使得 web 应用开发变得更简便，包括依赖注入、数据绑定、切面编程、数据存取等等。随着时间推移，Spring 生态变得越来越复杂了，并且应用程序所必须的配置文件也令人觉得可怕。这就是 SpirngBoot 派上用场的地方了–它使得 Spring 的配置变得更轻而易举。实际上，Spring 是 unopinionated（予以配置项多，倾向性弱）的，SpringBoot 在平台和库的做法中更 opinionated，使得我们更容易上手。这里有两条 SpringBoot 带来的好处：根据 classpath 中的 artifacts 的自动化配置应用程序提供非功能性特性例如安全和健康检查给到生产环境中的应用程序"19.创建一个 SpringBootProject 的最简单的方法是什么？"SpringInitializr 是启动 SpringBootProjects 的一个很好的工具。image-20210813203406199 就像上图中所展示的一样，我们需要做一下几步：登录 SpringInitializr，按照以下方式进行选择：选择 com.in28minutes.springboot 为组选择 studet-services 为组件选择下面的依赖项 WebActuatorDevTools 点击生 GenerateProject 将项目导入 Eclipse。文件–导入–现有的 Maven 项目"20.SpringBoot 有哪些优点？"SpringBoot 的优点有：1、减少开发，测试时间和努力。2、使用 JavaConfig 有助于避免使用 XML。3、避免大量的 Maven 导入和各种版本冲突。4、提供意见发展方法。5、通过提供默认值快速开始开发。6、没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat，Glassfish 或其他任何东西。7、需要更少的配置因为没有 web.xml 文件。只需添加用@Configuration 注释的类，然后添加用@Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将@Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。8、基于环境的配置使用这些属性，您可以将您正在使用的环境传递到应用程序：Dspring.profiles.active={enviornment}。在加载主应用程序属性文件后，Spring 将在（application{environment}.properties）中加载后续的应用程序属性文件。"21.什么是 springboot"用来简化 spring 应用的初始搭建以及开发过程使用特定的方式来进行配置（properties 或 yml 文件）创建独立的 spring 引用程序 main 方法运行嵌入的 Tomcat 无需部署 war 文件简化 maven 配置自动配置 spring 添加对应功能 starter 自动化配置 springboot 来简化 spring 应用开发，约定大于配置，去繁从简，justrun 就能创建一个独立的，产品级别的应用"十三、SpringCloud1.Eureka 和 zookeeper 都可以提供服务注册与发现的功能，两者的区别"Zookeeper 保证了 CP(C：一致性，P：分区容错性)，Eureka 保证了 AP(A：高可用，P：分区容错)1、Zookeeper——当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的信息，但不能容忍直接 down 掉不可用的。也就是说服务注册功能对高可用性要求比较高，但是 zk 会出现这样的一种情况，当 master 节点因为网络故障与其他节点失去联系时，剩余的节点会重新选 leader。问题在于，选取 leader 的时间过长(30~120s)，且选取期间 zk 集群都不可用，这样就会导致选取期间注册服务瘫痪。在云部署的环境下，因网络问题使得 zk 集群失去 master 节点是较大概率会发生的事，虽然服务最终恢复，但是漫长的选择时间导致的注册长期不可用是不能容忍的 2、Eureka 则看明白这一点，因此再设计的优先保证了高可用性。Eureka 各个节点都是平等的，几个节点挂掉不会影响到正常节点的工作，剩余的节点依然可以提供注册和查询服务。而 Eureka 的客户端再向某个 Eureka 注册时如果发现连接失败，则会自动切换至其他节点，只要有一台 Eureka 还在，就能保证注册服务的可用(保证可用性)，只不过查到的信息可能不是最新的(不保证一致性)。除此之外 Eureka 还有一种自我保护机制，如果在 15 分钟内超过 85%的节点都没有正常心跳，那么 Eureka 就认为客户端与注册中心出现了网络故障，此时就会出现以下几种情况：1>、Eureka 不再从注册列表移除因为长时间没收到心跳而应该过期的服务 2>、Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(保证当前节点可用)3>、当网络稳定时，当前实例新的注册信息会被同步到其它节点中 Eureka 还有客户端缓存功能(Eureka 分为客户端程序和服务器端程序两个部分，客户端程序负责向外提供注册与发现服务接口)。所以即便 Eureka 集群中所有节点都失效，或者发生网络分隔故障导致客户端不能访问任何一台 Eureka 服务器；Eureka 服务的消费者任然可以通过 Eureka 客户端缓存来获取所有的服务注册信息。甚至最极端的环境下，所有正常的 Eureka 节点都不对请求产生响应也没有更好的服务器解决方案来解决这种问题时；得益于 Eureka 的客户端缓存技术，消费者服务仍然可以通过 Eureka 客户端查询与获取注册服务信息，这点很重要，因此 Eureka 可以很好的应对网络故障导致部分节点失去联系的情况，而不像 Zookeeper 那样使整个注册服务瘫痪。"2.SpringCloudConfig 可以实现实时刷新吗?springcloudconfig 实时刷新采用 SpringCloudBus 消息总线 3.什么是 SpringCloudBus?"发送端（endpoint）构造事件 event，将其 publish 到 context 上下文中（springcloudbus 有一个父上下文，bootstrap），然后将事件发送到 channel 中（json 串 message），接收端从 channel 中获取到 message，将 message 转为事件 event，

然后将 event 事件 publish 到 context 上下文中，最后接收端（Listener）收到 event，调用服务进行处理。整个流程中，只有发送/接收端从 context 上下文中取事件和发送事件是需要我们在代码中明确写出来的，其它部分都由框架封装完成。"4.如何实现动态 Zuul 网关路由转发？通过 path 配置拦截请求，通过 ServiceId 到配置中心获取转发的服务列表，zuul 内部使用 Ribbon 实现本地负载均衡和转发。5.ZuulFilter 有哪些常用方法？"Run()：过滤器的具体业务逻辑 shouldFilter()：判断过滤器是否有效 filterOrder()：过滤器执行顺序 filterType()：过滤器拦截位置"6.什么是服务雪崩效应?雪崩效应是在大型互联网项目中，当某个服务发生宕机时，调用这个服务的其他服务也会发生宕机，大型项目的微服务之间的调用是互通的，这样就会将服务的不可用逐步扩大到各个其他服务中，从而使整个项目的服务宕机崩溃。7.什么是服务熔断？什么是服务降级？"服务熔断：熔断机制是应对雪崩效应的一种微服务链路保护机制。当某个微服务不可用或者响应时间太长时，会进行服务降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在 SpringCloud 框架里熔断机制通过 Hystrix 实现，Hystrix 会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是 5 秒内调用 20 次，如果失败，就会启动熔断机制。服务降级：服务降级，一般是从整体负荷考虑。就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的 fallback 回调，返回一个缺省值。"8.什么是 zuul?"zuul 是对 SpringCloud 提供的成熟对的路由方案，他会根据请求的路径不同，网关会定位到指定的微服务，并代理请求到不同的微服务接口，他对外隐蔽了微服务的真正接口地址。三个重要概念：动态路由表：Zuul 支持 Eureka 路由，手动配置路由，这俩种都支持自动更新路由定位：根据请求路径，Zuul 有自己的一套定位服务规则以及路由表达式匹配反向代理：客户端请求到路由网关，网关受理之后，在对目标发送请求，拿到响应之后在给客户端 Zuul 的应用场景：对外暴露，权限校验，服务聚合，日志审计等"9.说说 Eureka 的自我保护机制？当一个服务未按时进行心跳续约时，在生产环境下，因为网络延迟等原因，此时就把服务剔除列表并不妥当，因为服务可能没有宕机。Eureka 就会把当前实例的注册信息保护起来，不予剔除。生产环境下这很有效，保证了大多数服务依然可用。但是有可能会造成一些挂掉的服务被剔除有延迟。10.Eureka 的工作原理？"Eureka：服务注册中心（可以是一个集群），对外暴露自己的地址提供者：启动后向 Eureka 注册自己信息（地址，提供什么服务）消费者：向 Eureka 订阅服务，Eureka 会将对应服务的所有提供者地址列表发送给消费者，并且定期更新心跳(续约)：提供者定期通过 http 方式向 Eureka 刷新自己的状态（每 30s 定时向 EurekaServer 发起请求）"11.什么是 NetflixFeign？它的优点是什么？"Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 httpapis，而不考虑其稳定性。在 employeeconsumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。但是我们必须编写大量代码才能执行以下步骤使用功能区进行负载平衡。获取服务实例，然后获取基本 URL。利用 REST 模板来使用服务。前面的代码如下

@ControllerpublicclassConsumerControllerClient{@AutowiredprivateLoadBalancerClientloadBalancer;publicvoidgetEmployee()throwsRestClientException,IOException{ServiceInstanceserviceInstance=loadBalancer.choose(""employeeproducer"");System.out.println(serviceInstance.getUri());之前的代码，有像 NullPointer 这样的例外的机会，并不是最优的。我们将看到如何使用 NetflixFeign 使呼叫变得更加轻松和清洁。如果 NetflixRibbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载平衡。

StringbaseUrl=serviceInstance.getUri().toString();baseUrl=baseUrl+""/employee"";RestTemplaterestTemplate=newRestTemplate();try{response=restTemplate.exchange(baseUrl,ResponseEntity<String>response=null;HttpMethod.GET,getHeaders(),String.class);}catch(Exceptionex){System.out.println(ex);}System.out.println(response.getBody());}}"12.什么是 Hystrix 断路器？我们需要它吗？"由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。如果 firstPagemethod()中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firtsPage 方法，并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会。"13.什么是 Hystrix？它如何实现容错？"Hystrix 是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。思考以下微服务 image-20210814062924353 假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000.这是 hystrix 出现的地方我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employeeconsumer 使用由 employee-consumer 公开的服务。简化图如下所示 image-20210814062949346 现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。"14.SpringCloud 由哪些组件组成？"Eureka：服务注册与发现 Zuul：服务网关 Ribbon：客户端负载均衡 Feign：声明性的 Web 服务客户端 Hystrix：断路器 Config：分布式统一配置管理等 20 几个框架，开源一直在更新"15.服务注册和发现是什么意思？SpringCloud 如何实现？当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。16.使用 SpringCloud 有什么优势？"使用 SpringBoot 开发分布式微服务时，我们面临以下问题与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。冗余-分布式系统中的冗余问题。负载平衡–负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。性能-问题由于各种运营开销导致的性能问题。部署复杂性-Devops 技能的要求。"17.什么是 SpringCloud？springcloud 是一系列框架的有序集合。它利用 springboot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 springboot 的开发风格做到一键启动和部署。十四、Zookeeper1.zookeeper 负载均衡和 nginx 负载均衡区别"zookeeper 不存在单点问题，zab 机制保证单点故障可重新选举一个 leader 只负责服务的注册与发现，不负责转发，减少一次数据交换（消费方与服务方直接通信）需要自己实现相应的负载均衡算法 nginx 存在单点问题，单点负载高数据量大,需要通过 KeepAlived+LVS 备机实现高可用每次负载，都充当一次中间人转发角色，增加网络负载量（消费方与服务方间接通信）自带负载均衡算法"2.Zookeeper 和 Dubbo 的关系？"Zookeeper 的作用：zookeeper 用来注册服务和进行负载均衡，哪一个服务由哪一个机器来提供必需让调用者知道，简单来说就是 ip 地址和服务名称的对应关系。当然也可以通过硬编码的方式把这种对应关系在调用方业务代码中实现，但是如果提供服务的机器挂掉调用者无法知晓，如果不更改代码会继续请求挂掉的机器提供服务。zookeeper 通过心跳机制可以检测挂掉的机器并将挂掉机器的 ip 和服务对应关系从列表中删除。至于支持高并发，简单来说就是横向扩展，在不更改代码的情况通过添加机器来提高运算能力。通过添加新的机器向 zookeeper 注册服务，服务的提供者多了能服务的客户就多了。dubbo：是管理中间层的工具，在业务层到数据仓库间有非常多服务的接入和服务提供者需要调度，dubbo 提供一个框架解决这个问题。注意这里的 dubbo 只是一个框架，至于你架子上放什么是完全取决于你的，就像一个汽车骨架，你需要配你的轮子引擎。这个框架中要完成调度必须要有一个分布式的注册中心，储存所有服务的元数据，你可以用 zk，也可以用别的，只是大家都用 zk。zookeeper 和 dubbo 的关系：Dubbo 的将注册中心进行抽象，它可以外接不同的存储媒介给注册中心提供服务，有 ZooKeeper，Memcached，Redis 等。引入了 ZooKeeper 作为存储媒介，也就把 ZooKeeper 的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时候就需要分流，负载均衡就是为了分流而存在的，一个 ZooKeeper 群配合相应的 Web 应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不够，节点之间的数据和资源需要同步，ZooKeeper 集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动的时候，向 ZooKeeper 上的指定节点/dubbo/${serviceName}/providers 目录下写入自己的 URL 地址，这个操作就完

成了服务的发布。其他特性还有 Mast 选举，分布式锁等。"3.Zookeeper 的典型应用场景"数据发布/订阅负载均衡命名服务分布式协调/通知集群管理 Master 选举分布式锁分布式队列"4.ZAB 和 Paxos 算法的联系与区别？"相同点：两者都存在一个类似于 Leader 进程的角色，由其负责协调多个 Follower 进程的运行 Leader 进程都会等待超过半数的 Follower 做出正确的反馈后，才会将一个提案进行提交 ZAB 协议中，每个 Proposal 中都包含一个 epoch 值来代表当前的 Leader 周期，Paxos 中名字为 Ballot 不同点：ZAB 用来构建高可用的分布式数据主备系统（Zookeeper），Paxos 是用来构建分布式一致性状态机系统。"5.Zookeeper 对节点的 watch 监听通知是永久的吗？为什么不是永久的?"不是。官方声明：一个 Watch 事件是一个一次性的触发器，当被设置了 Watch 的数据发生了改变的时候，则服务器将这个改变发送给设置了 Watch 的客户端，以便通知它们。为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，给网络和服务器造成很大压力。一般是客户端执行 getData(“/节点 A”,true)，如果节点 A 发生了变更或删除，客户端会得到它的 watch 事件，但是在之后节点 A 又发生了变更，而客户端又没有设置 watch 事件，就不再给客户端发送。在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。"6.集群支持动态添加机器吗？"其实就是水平扩容了，Zookeeper 在这方面不太好。两种方式：全部重启：关闭所有 Zookeeper 服务，修改配置之后启动。不影响之前客户端的会话。逐个重启：在过半存活即可用的原则下，一台机器重启不影响整个集群对外提供服务。这是比较常用的方式。3.5 版本开始支持动态扩容。"7.集群最少要几台机器，集群规则是怎样的?集群规则为 2N+1 台，N>0，即 3 台。8.Zookeeper 有哪几种部署模式？单机模式、伪集群模式、集群模式。9.zk 节点宕机如何处理？"Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK 节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。所以 3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)"10.分布式集群中为什么会有 Master？在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，于是就需要进行 leader 选举。11.zookeeper 是如何保证事务的顺序一致性的？zookeeper 采用了全局递增的事务 Id 来标识，所有的 proposal（提议）都在被提出的时候加上了 zxid，zxid 实际上是一个 64 位的数字，高 32 位是 epoch（时期;纪元;世;新时代）用来标识 leader 周期，如果有新的 leader 产生出来，epoch 会自增，低 32 位用来递增计数。当新产生 proposal 的时候，会依据数据库的两阶段过程，首先会向其他的 server 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。12.数据同步"整个集群完成 Leader 选举之后，Learner（Follower 和 Observer 的统称）回向 Leader 服务器进行注册。当 Learner 服务器想 Leader 服务器完成注册后，进入数据同步环节。数据同步流程：(均以消息传递的方式进行）i.Learner 向 Learder 注册 ii.数据同步 iii.同步确认 Zookeeper 的数据同步通常分为四类：直接差异化同步（DIFF 同步）先回滚再差异化同步（TRUNC+DIFF 同步）仅回滚同步（TRUNC 同步）全量同步（SNAP 同步）在进行数据同步前，Leader 服务器会完成数据同步初始化：peerLastZxid：从 learner 服务器注册时发送的 ACKEPOCH 消息中提取 lastZxid（该 Learner 服务器最后处理的 ZXID）minCommittedLog：Leader 服务器 Proposal 缓存队列 committedLog 中最小 ZXIDmaxCommittedLog：Leader 服务器 Proposal 缓存队列 committedLog 中最大 ZXID 直接差异化同步（DIFF 同步）场景：peerLastZxid 介于 minCommittedLog 和 maxCommittedLog 之间先回滚再差异化同步（TRUNC+DIFF 同步）场景：当新的 Leader 服务器发现某个 Learner 服务器包含了一条自己没有的事务记录，那么就需要让该 Learner 服务器进行事务回滚–回滚到 Leader 服务器上存在的，同时也是最接近于 peerLastZxid 的 ZXID 仅回滚同步（TRUNC 同步）场景：peerLastZxid 大于 maxCommittedLog 全量同步（SNAP 同步）场景一：peerLastZxid 小于 minCommittedLog 场景二：Leader 服务器上没有 Proposal 缓存队列且 peerLastZxid 不等于 lastProcessZxid"13.Zookeeper 下 Server 工作状态"服务器具有四种状态，分别是 LOOKING、FOLLOWING、LEADING、OBSERVING。LOOKING：寻找 Leader 状态。当服务器处于该状态时，它会认为当前集群中没有 Leader，因此需要进入 Leader 选举状态。FOLLOWING：跟随者状态。表明当前服务器角色是 Follower。LEADING：领导者状态。表明当前服务器角色是 Leader。OBSERVING：观察者状态。表明当前服务器角色是 Observer。"14.服务器角色"Leader 事务请求的唯一调度和处理者，保证集群事务处理的顺序性集群内部各服务的调度者 Follower 处理客户端的非事务请求，转发事务请求给 Leader 服务器参与事务请求 Proposal 的投票参与 Leader 选举投票 Observer3.3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力处理客户端的非事务请求，转发事务请求给 Leader 服务器不参与任何形式的投票"15.ACL 权限控制机制"1）UGO（User/Group/Others）目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。2）ACL（AccessControlList）访问控制列表包括三个方面：权限模式（Scheme）IP：从 IP 地址粒度进行权限控制 Digest：最常用，用类似于 username:password 的权限标识来进行权限配置，便于区分不同应用来进行权限控制 World：最开放的权限控制方式，是一种特殊的 digest 模式，只有一个权限标识“world:anyone” Super：超级用户授权对象授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器灯。权限 PermissionCREATE：数据节点创建权限，允许授权对象在该 Znode 下创建子节点 DELETE：子节点删除权限，允许授权对象删除该数据节点的子节点 READ：数据节点的读取权限，允许授权对象访问该数据节点并读取其数据内容或子节点列表等 WRITE：数据节点更新权限，允许授权对象对该数据节点进行更新操作 ADMIN：数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关设置操作"16.服务端处理 Watcher 实现"1）服务端接收 Watcher 并存储接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在 WatcherManager 的 WatchTable 和 watch2Paths 中去。2）Watcher 触发以服务端接收到 setData()事务请求触发 NodeDataChanged 事件为例：封装 WatchedEvent 将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个 WatchedEvent 对象查询 Watcher 从 WatchTable 中根据节点路径查找 Watcher 没找到；说明没有客户端在该数据节点上注册过 Watcher 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）3）调用 process 方法来触发 Watcher 这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知。"17.客户端注册 Watcher 实现"调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象标记请求 request，封装 Watcher 到 WatchRegistration 封装成 Packet 对象，发服务端发送 request 收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理请求返回，完成注册。"18.ZookeeperWatcher 机制"Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。工作机制：客户端注册 watcher 服务端处理 watcher 客户端回调 watcherWatcher 特性总结：一次性无论是服务端还是客户端，一旦一个 Watcher 被触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。客户端串行执行客户端 Watcher 回调的过程是一个串行同步的过程。轻量 Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。watcherevent 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 orderingguarantee，即客户端监听事件后，才会感知它所监视 znode 发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。注册 watchergetData、exists、getChildren 触发 watchercreate、delete、setData 当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode 的 existwatch，如果在客

户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。"19.Zookeeper 文件系统 Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。20.ZooKeeper 提供了什么？"1、文件系统 2、通知机制"21.ZooKeeper 是什么？"ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。Zookeeper 保证了如下分布式一致性特性：顺序一致性原子性单一视图可靠性实时性（最终一致性）客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 zxid（ZookeeperTransactionId）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个 zookeeper 最新的 zxid。"22.Zookeeper 怎么实现服务注册？Zookeeper 提供一个多层级的节点命名空间（节点称为 znode）。与文件系统不同的是，这些节点都可以设置关联的数据，而文件系统中只有文件节点可以存放数据而目录节点不行。Zookeeper 为了保证高吞吐和低延迟，在内存中维护了这个树状的目录结构，这种特性使得 Zookeeper 不能用于存放大量的数据，每个节点的存放数据上限为 1M。23.ZookeeperLeader 选举过程是怎样的？"1、文件系统 2、通知机制"24.Zookeeper 是怎么保证数据一致性的？"ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。Zookeeper 保证了如下分布式一致性特性：顺序一致性原子性单一视图可靠性实时性（最终一致性）客户端的读请求可以被集群中的任意一台机器处理，如果读请求在节点上注册了监听器，这个监听器也是由所连接的 zookeeper 机器来处理。对于写请求，这些请求会同时发给其他 zookeeper 机器并且达成一致后，请求才会返回成功。因此，随着 zookeeper 的集群机器增多，读请求的吞吐会提高但是写请求的吞吐会下降。有序性是 zookeeper 中非常重要的一个特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为 zxid（ZookeeperTransactionId）。而读请求只会相对于更新有序，也就是读请求的返回结果中会带有这个 zookeeper 最新的 zxid。"25.Zookeeper 怎么实现分布式锁？"有了 zookeeper 的一致性文件系统，锁的问题变得容易。锁服务可以分为两类，一个是保持独占，另一个是控制时序。对于第一类，我们将 zookeeper 上的一个 znode 看作是一把锁，通过 createznode 的方式来实现。所有客户端都去创建/distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 distribute_lock 节点就释放出锁。对于第二类，/distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 master 一样，编号最小的获得锁，用完删除，依次方便。"26.了解过 Zookeeper 的 ZAB 协议吗？"ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复的原子广播协议。ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。当整个 zookeeper 集群刚刚启动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。"27.Zookeeper 有哪些节点类型？"PERSISTENT-持久节点除非手动删除，否则节点一直存在于 Zookeeper 上 EPHEMERAL-临时节点临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。PERSISTENT_SEQUENTIAL-持久顺序节点基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。EPHEMERAL_SEQUENTIAL-临时顺序节点基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。"十五、多线程 1.你将如何使用 threaddump？你将如何分析 Threaddump？在 UNIX 中你可以使用 kill-3，然后 threaddump 将会打印日志，在 windows 中你可以使用”CTRL+Break”。非常简单和专业的线程面试问题，但是如果他问你怎样分析它，就会很棘手。2.在 Java 中 Lock 接口比 synchronized 块的优势是什么？你需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，你会怎样去实现它？lock 接口在多线程和并发编程中最大的优势是它们为读和写分别提供了锁，它能满足你写像 ConcurrentHashMap 这样的高性能数据结构和有条件的阻塞。Java 线程面试的问题越来越会根据面试者的回答来提问。我强烈建议在你去参加多线程的面试之前认真读一下 Locks，因为当前其大量用于构建电子交易终统的客户端缓存和交易连接空间。3.高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？"1）高并发、任务执行时间短的业务，线程池线程数可以设置为 CPU 核数+1，减少线程上下文的切换 2）并发不高、任务执行时间长的业务要区分开看：a）假如是业务时间长集中在 IO 操作上，也就是 IO 密集型的任务，因为 IO 操作并不占用 CPU，所以不要让所有的 CPU 闲下来，可以加大线程池中的线程数目，让 CPU 处理更多的业务 b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换 c）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考其他有关线程池的文章。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。"4.同步方法和同步块，哪个是更好的选择？"同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：同步的范围越小越好。虽说同步的范围越少越好，但是在 Java 虚拟机中还是存在着一种叫做锁粗化的优化方法，这种方法就是把同步范围变大。这是有用的，比方说 StringBuffer，它是一个线程安全的类，自然最常用的 append()方法是一个同步方法，我们写代码的时候会反复 append 字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着 Java 虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此 Java 虚拟机会将多次 append 方法调用的代码进行一个锁粗化的操作，将多次的 append 的操作扩展"5.Hashtable 的 size()方法中明明只有一条语句”returncount”，为什么还要做同步？"1）同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程 A 在执行 Hashtable 的 put 方法添加数据，线程 B 则可以正常调用 size()方法读取 Hashtable 中当前元素的个数，那读取到的值可能不是最新的，可能线程 A 添加了完了数据，但是没有对 size++，线程 B 就已经读取 size 了，那么对于线程 B 来说读取到的 size 一定是不准确的。而给 size()方法加了同步之后，意味着线程 B 调用 size()方法只有在线程 A 调用 put 方法完毕之后才可以调用，这样就保证了线程安全性 2）CPU 执行代码，执行的不是 Java 代码，这点很关键，一定得记住。Java 代码最终是被翻译成机器码执行的，机器码才是真正可以和硬件电路交互的代码。即使你看到 Java 代码只有一行，甚至你看到 Java 代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句""returncount""假设被翻译成了三句汇编语句执行，一句汇编语句和其机器码做对应，完全可能执行完第一句，线程就切换了。"6.Semaphore 有什么作用？"Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。"7.单例模式的线程安全性"1）饿汉式单例模式的写法：线程安全 2）懒汉式单例模式的写法：非线程安全 3）双检锁单例模式的写法：线程安全"8.Java 中用到的线程调度算法是什么？"抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。"9.Java 中如何获取到线程 dump 文件"死循环、死锁、阻塞、页面打开慢等问题，打线程 dump 是最好的解决问题的途径。所谓线程 dump 也就是线程堆栈，获取到线程堆栈有两步：1）获取到线程的 pid，可以通过使用 jps 命令，在 Linux 环境下还可以使用 ps-ef|grepjava2）打印线程堆栈，

可以通过使用 jstackpid 命令，在 Linux 环境下还可以使用 kill-3pid 另外提一点，Thread 类提供了一个 getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈。"10.什么是线程安全？"如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。这个问题有值得一提的地方，就是线程安全也是有几个级别的：1）不可变像 String、Integer、Long 这些，都是 final 类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用 2）绝对线程安全不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java 中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java 中也有，比方说 CopyOnWriteArrayList、CopyOnWriteArraySet3）相对线程安全相对线程安全也就是我们通常意义上所说的线程安全，像 Vector 这种，add、remove 方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个 Vector、有个线程同时在 add 这个 Vector，99%的情况下都会出现 ConcurrentModificationException，也就是 failfast 机制。4）线程非安全这个就没什么好说的了，ArrayList、LinkedList、HashMap 等都是线程非安全的类，点击这里了解为什么不安全。"11.线程池都有哪几种工作队列？"ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。LinkedBlockingQueue：是一个基于链表结构的阻塞队列，此队列按 FIFO 排序元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法 Executors.newFixedThreadPool()使用了这个队列。SynchronousQueue：是一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 Linked-BlockingQueue，静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。PriorityBlockingQueue：一个具有优先级的无限阻塞队列。"12.说一说几种常见的线程池及适用场景？"FixedThreadPool：可重用固定线程数的线程池。（适用于负载比较重的服务器）FixedThreadPool 使用无界队列 LinkedBlockingQueue 作为线程池的工作队列该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。SingleThreadExecutor：只会创建一个线程执行任务。（适用于需要保证顺序执行各个任务；并且在任意时间点，没有多线程活动的场景。）SingleThreadExecutorl 也使用无界队列 LinkedBlockingQueue 作为工作队列若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。CachedThreadPool：是一个会根据需要调整线程数量的线程池。（大小无界，适用于执行很多的短期异步任务的小程序，或负载较轻的服务器）CachedThreadPool 使用没有容量的 SynchronousQueue 作为线程池的工作队列，但 CachedThreadPool 的 maximumPool 是无界的。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。ScheduledThreadPool：继承自 ThreadPoolExecutor。它主要用来在给定的延迟之后运行任务，或者定期执行任务。使用 DelayQueue 作为任务队列。"13.synchronized 关键字和 volatile 关键字的区别"volatile 关键字是线程同步的轻量级实现，所以 volatile 性能比 synchronized 关键字要好。但是 volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块。多线程访问 volatile 关键字不会发生阻塞，而 synchronized 关键字可能会发生阻塞。volatile 关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。volatile 关键字能保证数据的可见性，但不能保证数据的原子性。synchronized 关键字两者都能保证。"14.什么是线程的阻塞问题？如何解决？"阻塞是用来形容多线程的问题，几个线程之间共享临界区资源，那么当一个线程占用了临界区资源后，所有需要使用该资源的线程都需要进入该临界区等待，等待会导致线程挂起，一直不能工作，这种情况就是阻塞，如果某一线程一直都不释放资源，将会导致其他所有等待在这个临界区的线程都不能工作。当我们使用 synchronized 或重入锁时，我们得到的就是阻塞线程，如论是 synchronized 或者重入锁，都会在试图执行代码前，得到临界区的锁，如果得不到锁，线程将会被挂起等待，知道其他线程执行完成并释放锁且拿到锁为止。解决方法：可以通过减少锁持有时间，读写锁分离，减小锁的粒度，锁分离，锁粗化等方式来优化锁的性能。"15.什么是线程的饥饿问题？如何解决？"饥饿指的是某一线程或多个线程因为某些原因一直获取不到资源，导致程序一直无法执行。如某一线程优先级太低导致一直分配不到资源，或者是某一线程一直占着某种资源不放，导致该线程无法执行等。解决方法：与死锁相比，饥饿现象还是有可能在一段时间之后恢复执行的。可以设置合适的线程优先级来尽量避免饥饿的产生。"16.什么是活锁？活锁体现了一种谦让的美德，每个线程都想把资源让给对方，但是由于机器“智商”不够，可能会产生一直将资源让来让去，导致资源在两个线程间跳动而无法使某一线程真正的到资源并执行，这就是活锁的问题。17.什么是线程安全问题？如何解决？"线程安全问题指的是在某一线程从开始访问到结束访问某一数据期间，该数据被其他的线程所修改，那么对于当前线程而言，该线程就发生了线程安全问题，表现形式为数据的缺失，数据不一致等。线程安全问题发生的条件：1）多线程环境下，即存在包括自己在内存在有多个线程。2）多线程环境下存在共享资源，且多线程操作该共享资源。3）多个线程必须对该共享资源有非原子性操作。线程安全问题的解决思路：1）尽量不使用共享变量，将不必要的共享变量变成局部变量来使用。2）使用 synchronized 关键字同步代码块，或者使用 jdk 包中提供的 Lock 为操作进行加锁。3）使用 ThreadLocal 为每一个线程建立一个变量的副本，各个线程间独立操作，互不影响。"18.为什么我们调用 start()方法时会执行 run()方法，为什么我们不能直接调用 run()方法？"new 一个 Thread，线程进入初始状态；调用 start()方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。start()会执行线程的相应准备工作，然后自动执行 run()方法的内容，这是真正的多线程工作。而直接执行 run()方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。总结：调用 start 方法可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。"19.什么是线程死锁?如何避免死锁?"多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。假如线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。避免死锁的几个常见方法：避免一个线程同时获取多个锁避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。尝试使用定时锁，使用 lock.tryLock(timeout)来代替使用内部锁机制。对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。"20.并发与并行的区别？"并发指的是多个任务交替进行，并行则是指真正意义上的 “同时进行”。实际上，如果系统内只有一个 CPU，使用多线程时，在真实系统环境下不能并行，只能通过切换时间片的方式交替进行，从而并发执行任务。真正的并行只能出现在拥有多个 CPU 的系统中。"21.虚拟机栈和本地方法栈为什么是私有的?"虚拟机栈：每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。本地方法栈：和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。所以，为了保证线程中的局部变量不被别的线程访问到，虚拟机栈和本地方法栈是线程私有的。"22.程序计数器为什么是私有的?"程序计数器主要有下面两个作用：字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。（需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。）所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。"23.什么是线程和进程?"进程：在操作系统中能够独立运行，并且作为资源分配的基本单位。它表示运行中的程序。系统运行一个程序就是一个进程从创建、运行到消亡的过程。线程：是一个比进程更小的执行单位，能够完成进程中的一个功能，也被称为轻量级进程。一个进程在其执行的过程中可以产生多个线程。线程与进程不同的是：同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多。"24.什么是多线程的上下文切换？"即使是单核 CPU 也支持多线程执行代码，CPU 通过给每个线程分配 CPU 时间片来实现这个机制。时间片是 CPU 分配给各个线程的时间，因为时间片非常短，所以 CPU 通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）上下文切换过程中，CPU 会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行 CPU 通过时间片分配算法来循环执行任

务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态从任务保存到再加载的过程就是一次上下文切换"25.什么是自旋锁？"自旋锁是 SMP 架构中的一种 low-level 的同步机制。当线程 A 想要获取一把自旋锁而该锁又被其它线程锁持有时，线程 A 会在一个循环中自旋以检测锁是不是已经可用了。自旋锁需要注意：由于自旋时不释放 CPU，因而持有自旋锁的线程应该尽快释放自旋锁，否则等待该自旋锁的线程会一直在那里自旋，这就会浪费 CPU 时间。持有自旋锁的线程在 sleep 之前应该释放自旋锁以便其它线程可以获得自旋锁。"26.AQS 支持几种同步方式？"1）独占式 2）共享式这样方便使用者实现不同类型的同步组件，独占式如 ReentrantLock，共享式如 Semaphore，CountDownLatch，组合式的如 ReentrantReadWriteLock。总之，AQS 为使用提供了底层支撑，如何组装实现，使用者可以自由发挥。"27.什么是 AQS？"AQS 是 AbustactQueuedSynchronizer 的简称，它是一个 Java 提高的底层同步工具类，用一个 int 类型的变量表示同步状态，并提供了一系列的 CAS 操作来管理这个同步状态。AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 ReentrantLock，Semaphore，其他的诸如 ReentrantReadWriteLock，SynchronousQueue，FutureTask 等等皆是基于 AQS 的。"28.CAS 的问题"1）CAS 容易造成 ABA 问题一个线程 a 将数值改成了 b，接着又改成了 a，此时 CAS 认为是没有变化，其实是已经变化过了，而这个问题的解决方案可以使用版本号标识，每操作一次 version 加 1。在 java5 中，已经提供了 AtomicStampedReference 来解决问题。2）不能保证代码块的原子性 CAS 机制所保证的知识一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证 3 个变量共同进行原子性的更新，就不得不使用 synchronized 了。3）CAS 造成 CPU 利用率增加之前说过了 CAS 里面是一个循环判断的过程，如果线程一直没有获取到状态，cpu 资源会一直被占用。"29.什么是 CAS？"CAS 是 compareandswap 的缩写，即我们所说的比较交换。cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。CAS 操作包含三个操作数——内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的(AtomicInteger,AtomicBoolean,AtomicLong)。"30.CyclicBarrier 和 CountDownLatch 的区别"1）CountDownLatch 简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用 countDown()方法发出通知后，当前线程才可以继续执行。2）cyclicBarrier 是所有线程都进行等待，直到所有线程都准备好进入 await()方法之后，所有线程同时开始执行！3）CountDownLatch 的计数器只能使用一次。而 CyclicBarrier 的计数器可以使用 reset()方法重置。所以 CyclicBarrier 能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。4）CyclicBarrier 还提供其他有用的方法，比如 getNumberWaiting 方法可以获得 CyclicBarrier 阻塞的线程数量。isBroken 方法用来知道阻塞的线程是否被中断。如果被中断返回 true，否则返回 false。"31.线程池的优点？"1）重用存在的线程，减少对象创建销毁的开销。2）可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。3）提供定时执行、定期执行、单线程、并发数控制等功能。"32.创建线程有哪些方式？"1）继承 Thread 类创建线程类 2）通过 Runnable 接口创建线程类 3）通过 Callable 和 Future 创建线程 4）通过线程池创建"33.并发编程三要素？"1）原子性原子性指的是一个或者多个操作，要么全部执行并且在执行的过程中不被其他操作打断，要么就全部都不执行。2）可见性可见性指多个线程操作一个共享变量时，其中一个线程对变量进行修改后，其他线程可以立即看到修改的结果。3）有序性有序性，即程序的执行顺序按照代码的先后顺序来执行。"34.什么是悲观锁？什么是乐观锁？"当我们要对一个数据库中的一条数据进行修改的时候，为了避免同时被其他人修改，最好的办法就是直接对该数据进行加锁以防止并发。这种借助数据库锁机制在修改数据之前先锁定，再修改的方式被称之为悲观并发控制（又名“悲观锁”，PessimisticConcurrencyControl，缩写“PCC”）。之所以叫做悲观锁，是因为这是一种对数据的修改抱有悲观态度的并发控制方式。我们一般认为数据被并发修改的概率比较大，所以需要在修改之前先加锁。乐观锁（OptimisticLocking）是相对悲观锁而言的，乐观锁假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。"35.Java 里的线程有哪些状态？"初始(NEW)：新创建了一个线程对象，但还没有调用 start()方法。运行(RUNNABLE)：Java 线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。线程对象创建后，其他线程(比如 main 线程）调用了该对象的 start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取 CPU 的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得 CPU 时间片后变为运行中状态（running）。阻塞(BLOCKED)：表示线程阻塞于锁。等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。超时等待(TIMED_WAITING)：该状态不同于 WAITING，它可以在指定的时间后自行返回。终止(TERMINATED)：表示该线程已经执行完毕。"36.如何避免“伪共享”？"字节填充（创建变量时，使用字段对其进行填充，避免多个变量被分派到同一个缓存行里）。JDK8 提供了一个 Contended 注解来解决伪共享。"37.“伪共享”出现的原因是什么？因为 CPU 缓存和内存交换数据的单位是缓存行，而同一个缓存行里的多个变量不能同时被多个线程修改。38.了解过什么是“伪共享”吗？CPU 缓存从内存读数据时，是按缓存行读取的，即使只用到一个变量，也要将整行数据进行读取，这行数据量可能包含其他变量。当多个线程同时修改同一个缓存行里的不同变量时，由于同时只能有一个线程在操作，所以相比将每个变量放到不同缓存行里，性能会有所下降。多个线程同时修改了同一个缓存行上的不同变量，由于不能并发修改，所以称为“伪共享”。39.说一下 synchronized 锁升级过程"偏向锁在 JDK1.8 中，其实默认是轻量级锁，但如果设定了-XX:BiasedLockingStartupDelay=0，那在对一个 Object 做 syncronized 的时候，会立即上一把偏向锁。当处于偏向锁状态时，markwork 会记录当前线程 ID。升级到轻量级锁当下一个线程参与到偏向锁竞争时，会先判断 markword 中保存的线程 ID 是否与这个线程 ID 相等，如果不相等，会立即撤销偏向锁，升级为轻量级锁。每个线程在自己的线程栈中生成一个 LockRecord(LR)，然后每个线程通过 CAS(自旋)的操作将锁对象头中的 markwork 设置为指向自己的 LR 的指针，哪个线程设置成功，就意味着获得锁。关于 synchronized 中此时执行的 CAS 操作是通过 native 的调用 HotSpot 中 bytecodeInterpreter.cpp 文件 C++代码实现的，有兴趣的可以继续深挖。升级到重量级锁如果锁竞争加剧(如线程自旋次数或者自旋的线程数超过某阈值，JDK1.6 之后，由 JVM 自己控制该规则)，就会升级为重量级锁。此时就会向操作系统申请资源，线程挂起，进入到操作系统内核态的等待队列中，等待操作系统调度，然后映射回用户态。在重量级锁中，由于需要做内核态到用户态的转换，而这个过程中需要消耗较多时间，也就是""重""的原因之一。"40.ReentrantLock 与 synchronized 的区别"ReentrantLock 有如下特点：可重入 ReentrantLock 和 syncronized 关键字一样，都是可重入锁，不过两者实现原理稍有差别，RetrantLock 利用 AQS 的的 state 状态来判断资源是否已锁，同一线程重入加锁，state 的状态+1;同一线程重入解锁,state 状态-1(解锁必须为当前独占线程，否则异常);当 state 为 0 时解锁成功。需要手动加锁、解锁 synchronized 关键字是自动进行加锁、解锁的，而 ReentrantLock 需要 lock()和 unlock()方法配合 try/finally 语句块来完成，来手动加锁、解锁。支持设置锁的超时时间 synchronized 关键字无法设置锁的超时时间，如果一个获得锁的线程内部发生死锁，那么其他线程就会一直进入阻塞状态，而 ReentrantLock 提供 tryLock 方法，允许设置线程获取锁的超时时间，如果超时，则跳过，不进行任何操作，避免死锁的发生。支持公平/非公平锁 synchronized 关键字是一种非公平锁，先抢到锁的线程先执行。而 ReentrantLock 的构造方法中允许设置 true/false 来实现公平、非公平锁，如果设置为 true，则线程获取锁要遵循""先来后到""的规则，每次都会构造一个线程 Node，然后到双向链表的""尾巴""后面排队，等待前面的 Node 释放锁资源。可中断锁 ReentrantLock 中的 lockInterruptibly()方法使得线程可以在被阻塞时响应中断，比如一个线程 t1 通过 lockInterruptibly()方法获取到一个可重入锁，并执行一个长时间的任务，另一个线程通过 interrupt()方法就可以立刻打断 t1 线程的执行，来获取 t1 持有的那个可重入锁。而通过 ReentrantLock 的 lock()方法或者 Synchronized 持有锁的线程是不会响应其他线程的 interrupt()方法的，直到该方法主动释放锁之后才会响应 interrupt()方法。"41.说说 synchronized 的实现原理在 Java 中，每个对象都隐式包含一个 monitor（监视器）对象，加锁的过程其实就是竞争 monitor 的过程，当线程进入字节码 monitorenter 指令之后，线程将持有 monitor 对象，执行

monitorexit 时释放 monitor 对象，当其他线程没有拿到 monitor 对象时，则需要阻塞等待获取该对象。42.sleep()方法和 wait()方法的区别和共同点?"相同点：两者都可以暂停线程的执行，都会让线程进入等待状态。不同点：sleep()方法没有释放锁，而 wait()方法释放了锁。sleep()方法属于 Thread 类的静态方法，作用于当前线程；而 wait()方法是 Object 类的实例方法，作用于对象本身。执行 sleep()方法后，可以通过超时或者调用 interrupt()方法唤醒休眠中的线程；执行 wait()方法后，通过调用 notify()或 notifyAll()方法唤醒等待线程。"43.Thread.sleep(0)的作用是什么？由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。十六、分布式 1.SOA 和微服务架构有哪些区别?"微服务是在 SOA 的基础上发展而来,从粒度上来说,微服务的粒度要比 SOA 更细.微服务由于粒度更细,所以微服务架构的耦合度相对于 SOA 架构的耦合度更低.微服务的服务规模相较于 SOA 一般要更大,所能承载的并发量也更高."2.BASE 理论了解过吗？BASE 是 BasicallyAvailable(基本可用)Softstate(软状态)Eventuallyconsistent(最终一致性)这几个单词的缩写,是从 CAP 理论发展而来的,其核心思想是:即使无法做到强一致性,但每个应用都可以根据自身特点,采取适当的方式来使系统达到最终一致性.3.如何保障请求执行顺序"一般来说，从业务逻辑上最好设计系统不需要这种顺序的保证，因为一旦引入顺序性保障，会导致系统复杂度的上升，效率会降低，对于热点数据会压力过大等问题。首先使用一致性 hash 负载均衡策略，将同一个 id 的请求都分发到同一个机器上面去处理，比如订单可以根据订单 id。如果处理的机器上面是多线程处理的，可以引入内存队列去处理，将相同 id 的请求通过 hash 到同一个队列当中，一个队列只对应一个处理线程。最好能将多个操作合并成一个操作。"4.分布式系统的接口幂等性设计"唯一 id。每次操作，都根据操作和内容生成唯一的 id，在执行之前先判断 id 是否存在，如果不存在则执行后续操作，并且保存到数据库或者 redis 等。服务端提供发送 token 的接口，业务调用接口前先获取 token,然后调用业务接口请求时，把 token 携带过去,务器判断 token 是否存在 redis 中，存在表示第一次请求，可以继续执行业务，执行业务完成后，最后需要把 redis 中的 token 删除建去重表。将业务中有唯一标识的字段保存到去重表，如果表中存在，则表示已经处理过了版本控制。增加版本号，当版本号符合时，才能更新数据状态控制。例如订单有状态已支付未支付支付中支付失败，当处于未支付的时候才允许修改为支付中等"5.如何设计一个秒杀系统?"前端：在秒杀之前，按钮置灰，并且不给前端真正的请求地址。前端定时请求后端接口，如果到了秒杀时间，则返回给前端真正的地址，前端放开按钮，每次点击后都要等 X 秒才能点击。服务器：服务器用 nginx 做集群、redis 也做集群限流：在秒杀之前，将秒杀数量的令牌存入到 redis 中，可以用 list，每次来请求都去 redis 中取出令牌，如果获取到说明秒杀成功，然后去访问数据库下单，如果没有获取到，则说明商品卖完了。消息中间件：如果秒杀数量比较多，例如上万十万，则秒杀成功之后，将成功的请求放入到 mq 或者 kafka 中间件中，再从消息队列中获取请求。服务降级：为了以防万一，还是要做服务熔断降级。"6.如何防止表单重复提交？"前端。每次点击后都要等 X 秒才能点击。数据库添加唯一索引服务器返回表单页面时，会先生成一个 token 保存于 session 或 redis，当表单提交时候携带 token，如果 token 一致，则执行后续，并将服务器中的 token 删除。"7.分布式 Session 了解过吗？如何实现？"如果不做任何处理的话，用户将出现频繁登录的现象，比如集群中存在 A、B 两台服务器，用户在第一次访问网站时，Nginx 通过其负载均衡机制将用户请求转发到 A 服务器，这时 A 服务器就会给用户创建一个 Session。当用户第二次发送请求时，Nginx 将其负载均衡到 B 服务器，而这时候 B 服务器并不存在 Session，所以就会将用户踢到登录页面。这将大大降低用户体验度，导致用户的流失，这种情况是项目绝不应该出现的。粘性 Session 原理粘性 Session 是指将用户锁定到某一个服务器上，比如上面说的例子，用户第一次请求时，负载均衡器将用户的请求转发到了 A 服务器上，如果负载均衡器设置了粘性 Session 的话，那么用户以后的每次请求都会转发到 A 服务器上，相当于把用户和 A 服务器粘到了一块，这就是粘性 Session 机制。优点简单，不需要对 Session 做任何处理。缺点缺乏容错性，如果当前访问的服务器发生故障，用户被转移到第二个服务器上时，他的 Session 信息都将失效。适用场景发生故障对客户产生的影响较小；服务器发生故障是低概率事件。服务器 Session 复制原理任何一个服务器上的 Session 发生改变，该节点会把这个 Session 的所有内容序列化，然后广播给所有其它节点，不管其他服务器需不需要 Session，以此来保证 Session 同步。优点可容错，各个服务器间 Session 能够实时响应。缺点会对网络负荷造成一定压力，如果 Session 量大的话可能会造成网络堵塞，拖慢服务器性能。实现方式设置 Tomcat 的 server.xml 开启 tomcat 集群功能。在应用里增加信息：通知应用当前处于集群环境中，支持分布式，即在 web.xml 中添加选项。Session 共享机制使用分布式缓存方案比如 Memcached、Redis，但是要求 Memcached 或 Redis 必须是集群。使用 Session 共享也分两种机制，两种情况如下：3.1 粘性 Session 共享机制和粘性 Session 一样，一个用户的 Session 会绑定到一个 Tomcat 上。Memcached 只是起到备份作用。3.2 非粘性 Session 共享机制原理 Tomcat 本身不存储 Session，而是存入 Memcached 中。Memcached 集群构建主从复制架构。优点可容错，Session 实时响应。实现方式用开源的 msm 插件解决 Tomcat 之间的 Session 共享：Memcached_Session_Manager（MSM）Session 持久化到数据库原理拿出一个数据库，专门用来存储 Session 信息。保证 Session 的持久化。优点服务器出现问题，Session 不会丢失。缺点如果网站的访问量很大，把 Session 存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库。Terracotta 实现 Session 复制原理 Terracotta 的基本原理是对于集群间共享的数据，当在一个节点发生变化的时候，Terracotta 只把变化的部分发送给 Terracotta 服务器，然后由服务器把它转发给真正需要这个数据的节点。它是服务器 Session 复制的优化。优点这样对网络的压力就非常小，各个节点也不必浪费 CPU 时间和内存进行大量的序列化操作。把这种集群间数据共享的机制应用在 Session 同步上，既避免了对数据库的依赖，又能达到负载均衡和灾难恢复的效果。"8.正向代理和反向代理的区别"正向代理：发生在客户端，是由用户主动发起的。比如翻墙，客户端通过主动访问代理服务器，让代理服务器获得需要的外网数据，然后转发回客户端。反向代理：发生在服务器端，用户不知道发生了代理。"9.负载均衡的实现方案有哪些？"DNS 解析使用 DNS 作为负载均衡器，会根据负载情况返回不同服务器的 IP 地址。大型网站基本使用了这种方式最为第一级负载均衡手段，然后在内部在第二级负载均衡。修改 MAC 地址使用 LVS（LinuxVirtualServer）这种链路层负载均衡器，根据负载情况修改请求的 MAC 地址。修改 IP 地址在网络层修改请求的目的 IP 地址。HTTP 重定向 HTTP 重定向负载均衡服务器收到 HTTP 请求之后会返回服务器的地址，并将该地址写入 HTTP 重定向响应中返回给浏览器，浏览器收到后再次发送请求。"10.了解过哪些负载均衡算法？"轮询（RoundRobin）轮询算法把每个请求轮流发送到每个服务器上。该算法比较适合每个服务器的性能差不多的场景，如果有性能存在差异的情况下，那么性能较差的服务器可能无法承担多大的负载。加权轮询（WeightedRoundRobbin）加权轮询是在轮询的基础上，根据服务器的性能差异，为服务器赋予一定的权值。最少连接（leastConnections）由于每个请求的连接时间不一样，使用轮询或者加权轮询算法的话，可能会让一台服务器当前连接数过多，而另一台服务器的连接数过少，造成负载不均衡。最少连接算法就是将请求发送给当前最少连接数的服务器上。加权最小连接（WeightedLeastConnection）在最小连接的基础上，根据服务器的性能为每台服务器分配权重，然后根据权重计算出每台服务器能处理的连接数。随机算法（Random）把请求随机发送到服务器上。和轮询算法类似，该算法比较适合服务器性能差不多的场景。"11.TCC 了解过吗？try,commit,cancel 的缩写,try 阶段进行检测,commit 提交执行,只要 try 阶段成功了 commit 就一定会被执行,cancel 业务出现错误时执行,回滚事务,释放资源.12.什么是二阶段提交（2PC）？什么是三阶段提交（3PC）？"二阶段提交 2PC:第一步请求阶段通过协调者来统计表决结果,第二步执行表决后的结果,如果表决的结果是提交,那就提交执行,否则不执行提交.缺点是同步阻塞,而且万一协调者挂了就无法保证 ACID.三阶段提交 3PC:在 2PC 的第一步拆分成了 2 步并且引入了超时机制,解决了 2PC 的痛点.第一步先向参与者发出一个信号,看看大家是否都能提交,如果可以就返回 yes,否则返回 no.第二步 PreCommit 阶段,预提交一下,如果参与者可以完成 commit,就返回 ack 进确认,如果不能则放弃提交本次事务.第三步 doCommit 阶段,进行真正的事务提交."13.分布式事务了解过吗？涉及到多个数据库操作的事务即为分布式事务,目的是为保证分布式系统中的数据一致性.14.什么是 CAP 定理？任何分布式系统都无法同时满足一致性(consistency),可用性(availibity),分区容错性(partitiontolerance)这三项,最多只可同时满足其中的两项。15.雪花算法了解过吗？雪花算法生成的是 Long 类型的 ID，一个 Long 类型占 8 个字节，每个字节占 8 比特，也就是说一个 Long 类型占 64 个比特。雪花 ID 组成结构：正数位（占 1 比特）+时间戳（占 41 比特）+机器 ID（占 5 比特）+数据中心（占 5 比特）+自增值（占 12 比特），总共 64 比特组成的一个 Long 类型。第一个 bit 位（1bit）：Java 中 long 的最高位是符号位代表正负，正数是 0，负数是 1，一般生成 ID 都为正数，所以默认为 0。时间戳部分（41bit）：毫秒级的时间，不建议存当前时间戳，而是用（当前时

间戳–固定开始时间戳）的差值，可以使产生的 ID 从更小的值开始；41 位的时间戳可以使用 69 年，(1L<<41)/(1000L606024365)=69 年工作机器 id（10bit）：也被叫做 workId，这个可以灵活配置，机房或者机器号组合都可以。序列号部分（12bit），自增值支持同一毫秒内同一个节点可以生成 4096 个 ID 十七、计算机网络 1.HTTP 协议包括哪些请求？"GET：对服务器资源的简单请求 POST：用于发送包含用户提交数据的请求 HEAD：类似于 GET 请求，不过返回的响应中没有具体内容，用于获取报头 PUT：传说中请求文档的一个版本 DELETE：发出一个删除指定文档的请求 TRACE：发送一个请求副本，以跟踪其处理进程 OPTIONS：返回所有可用的方法，检查服务器支持哪些方法 CONNECT：用于 ssl 隧道的基于代理的请求"2.在浏览器中输入 url 地址到显示主页的过程"百度好像最喜欢问这个问题。打开一个网页，整个过程会使用哪些协议图解（图片来源：《图解 HTTP》）：image-20210814132940082 总体来说分为以下几个过程:DNS 解析 TCP 连接发送 HTTP 请求服务器处理请求并返回 HTTP 报文浏览器解析渲染页面连接结束具体可以参考下面这篇文章：https://segmentfault.com/a/1190000006879700"3.拥塞控制"在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做到的就是抑制发送端发送数据的速率，以便使接收端来得及接收。为了进行拥塞控制，TCP 发送方要维持一个拥塞窗口(cwnd)的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。TCP 的拥塞控制采用了四种算法，即慢开始、拥塞避免、快重传和快恢复。在网络层也可以使路由器采用适当的分组丢弃策略（如主动队列管理 AQM），以减少网络拥塞的发生。慢开始：慢开始算法的思路是当主机开始发送数据时，如果立即把大量数据字节注入到网络，那么可能会引起网络阻塞，因为现在还不知道网络的符合情况。经验表明，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是由小到大逐渐增大拥塞窗口数值。cwnd 初始值为 1，每经过一个传播轮次，cwnd 加倍。拥塞避免：拥塞避免算法的思路是让拥塞窗口 cwnd 缓慢增大，即每经过一个往返时间 RTT 就把发送放的 cwnd 加 1.快重传与快恢复：在 TCP/IP 中，快速重传和恢复（fastretransmitandrecovery，FRR）是一种拥塞控制算法，它能快速恢复丢失的数据包。没有 FRR，如果数据包丢失了，TCP 将会使用定时器来要求传输暂停。在暂停的这段时间内，没有新的或复制的数据包被发送。有了 FRR，如果接收机接收到一个不按顺序的数据段，它会立即给发送机发送一个重复确认。如果发送机接收到三个重复确认，它会假定确认件指出的数据段丢失了，并立即重传这些丢失的数据段。有了 FRR，就不会因为重传时要求的暂停被耽误。当有单独的数据包丢失时，快速重传和恢复（FRR）能最有效地工作。当有多个数据信息包在某一段很短的时间内丢失时，它则不能很有效地工作。"4.滑动窗口和流量控制 TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。5.TCP 为什么要四次挥手"任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了 TCP 连接。举个例子：A 和 B 打电话，通话即将结束后，A 说“我没啥要说的了”，B 回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”，A 回答“知道了”，这样通话才算结束。上面讲的比较概括，推荐一篇讲的比较细致的文章：https://blog.csdn.net/qzcsu/article/details/72861891"6.TCP 建立连接时为什么要传回 SYN"接收端传回发送端所发送的 SYN 是为了告诉发送端，我接收到的信息确实就是你所发送的信号了。SYN 是 TCP/IP 建立连接时使用的握手信号。在客户机和服务器之间建立正常的 TCP 网络连接时，客户机首先发出一个 SYN 消息，服务器使用 SYN-ACK 应答表示接收到了这个消息，最后客户机再以 ACK(Acknowledgement[汉译:确认字符，在数据通信传输中，接收站发给发送站的一种传输控制字符。它表示确认发来的数据已经接受无误。])消息响应。这样在客户机和服务器之间才能建立起可靠的 TCP 连接，数据才可以在客户机和服务器之间传递。"7.为什么 TCP 要三次握手"三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收是正常的。第一次握手：Client 什么都不能确认；Server 确认了对方发送正常，自己接收正常第二次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：对方发送正常，自己接收正常第三次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：自己发送、接收正常，对方发送、接收正常所以三次握手就能确认双发收发功能都正常，缺一不可。"8.说一说 TCP 的三次握手"在 TCP/IP 协议中，TCP 协议提供可靠的连接服务，连接是通过三次握手进行初始化的。三次握手的目的是同步连接双方的序列号和确认号并交换 TCP 窗口大小信息 image-20210814132622711 核心思想：让双方都证实对方能发收。知道对方能收是因为收到对方的因为收到信息之后发的回应(ACK)。客户端–发送带有 SYN 标志的数据包—一次握手–服务端服务端–发送带有 SYN/ACK 标志的数据包–二次握手–客户端客户端–发送带有带有 ACK 标志的数据包–三次握手–服务端"9.简述 ICMP、TFTP、HTTP、NAT、DHCP 协议"ICMP:因特网控制报文协议。它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息 TFTP：是 TCP/IP 协议族中的一个用来在客户机和服务器之间进行简单的文件传输的协议，提供不复杂、开销不大的文件传输服务 HTTP：超文本传输层协议，是一个属于应用层的面向对象的协议 NAT 协议：网络地址转换接入广域网（WAN）技术，是一种将私有地址转换为合法 IP 地址的转换技术 DHCP 协议：动态主机配置协议，使用 UDP 协议工作。给内部的网络和网络服务供应商自动的分配 IP 地址。RARP 是逆地址解析协议，作用是完成从硬件地址到 IP 地址的映射，RARP 只能用于具有广播能力的网络。封装一个 RARP 的数据包里面有 MAC 地址，然后广播到网络上，当服务器收到请求包后，就查找对应的 MAC 地址的 IP 地址装入到响应报文中发送给请求者。一些常见的端口号及其用途：TCP21 端口：FTP 文件传输服务 TCP23 端口：TELNET 终端仿真服务 TCP25 端口：SMTP 简单邮件传输服务 UDP53 端口：DNS 域名解析服务 TCP80 端口：HTTP 超文本传输服务 TCP109 端口：POP2 邮局协议 2TCP110 端口：POP3 邮局协议版本 3 使用的端口 UDP69 端口：TFTP 简单文件传输协议 3306：Mysql 端口号"10.简述 ARP 地址解析协议工作原理"首先，每个主机会在自己的 ARP 缓冲区简历一个 ARP 列表，以表示 IP 地址和 MAC 地址之间的对应关系。当源主机要发送数据时，首先检查自己的 ARP 列表中是否有对应的目的主机的 MAC 地址，如果有就直接发送数据，如果没有，就向本网段的所有的主机发送 ARP 数据包，该数据包括的内容由：源主机 IP 地址，源主机的 MAC 地址，目的主机的 IP 地址当本网络的所有主机收到 ARP 数据包时，首先检查数据包中的 IP 地址是否是自己的 IP 地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的 IP 和 MAC 地址写入到 ARP 列表中，如果已经存在，则覆盖，然后将自己的 MAC 地址中放入到 ARP 响应包中，告诉源主机自己是它想找的 MAC 地址。源主机接收到 ARP 响应包后，将目的主机的 IP 和 MAC 地址写入到 ARP 列表，并利用此消息发送数据。如果源主机一直没有收到 ARP 响应数据包，表示 ARP 查询失败。广播发送 ARP 请求，单播发送 ARP 响应。"11.简述 IP 地址的分类?"IP 地址分为网络号和主机号，A 类地址的前 8 位是网络地址，B 类地址的前 16 位是网络地址，C 类地址的前 24 位是网络地址。A 类地址：1.0.0.0~126.0.0.0B 类地址：128.0.0.0~191.255.255.255C 类地址：192.0.0.0~223.255.255.255D 类地址：224.0.0.0~239.255.255.255（作为多播使用）E 类地址：保留 A,B,C 是基本类，D、E 类作为多播和保留使用。主机号，全 0 的是网络号，主机号全 1 的是广播地址。"12.说一说 TCP、IP 四层模型 13.你能说一说 OSI 七层模型?14.有哪些私有（保留）地址？"A 类：10.0.0.0–10.255.255.255B 类：172.16.0.0–172.31.255.255C 类：192.168.0.0–192.168.255.255"15.TCP 对应的协议和 UDP 对应的协议"TCP 对应的协议：FTP：定义了文件传输协议，使用 21 端口。Telnet：一种用于远程登陆的端口，使用 23 端口，用户可以以自己的身份远程连接到计算机上，可提供基于 DOS 模式下的通信服务。SMTP：邮件传送协议，用于发送邮件。服务器开放的是 25 号端口。POP3：它是和 SMTP 对应，POP3 用于接收邮件。POP3 协议所用的是 110 端口。HTTP：是从 Web 服务器传输超文本到本地浏览器的传送协议。UDP 对应的协议：DNS：用于域名解析服务，将域名地址转换为 IP 地址。DNS 用的是 53 号端口。SNMP：简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。TFTP(TrivalFileTransferProtocal)，简单文件传输协议，该协议在熟知端口 69 上使用 UDP 服务。"16.请简述 TCP 和 UDP 的区别"TCP 和 UDP 是 OSI 模型中的运输层中的协议。TCP 提供可靠的通信传输，而 UDP 则常被用于让广播和细节控制交给应用的通信传输。两者的区别大致如下：TCP 面向连接，UDP 面向非连接即发送数据前不需要建立链接 TCP 提供可靠的服务（数据传输），UDP

无法保证 TCP 面向字节流，UDP 面向报文 TCP 数据传输慢，UDP 数据传输快 TCP 提供一种面向连接的、可靠的字节流服务在一个 TCP 连接中，仅有两方进行彼此通信，因此广播和多播不能用于 TCPTCP 使用校验和，确认和重传机制来保证可靠传输 TCP 使用累积确认 TCP 使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制"十八、设计模式 1.Spring 当中用到了哪些设计模式？"模板方法模式：例如 jdbcTemplate，通过封装固定的数据库访问比如获取 connection、获取 statement，关闭 connection、关闭 statement 等然后将特殊的 sql 操作交给用户自己实现。策略模式：Spring 在初始化对象的时候，可以选择单例或者原型模式。简单工厂：Spring 中的 BeanFactory 就是简单工厂模式的体现，根据传入一个唯一的标识来获得 bean 对象。工厂方法模式：一般情况下,应用程序有自己的工厂对象来创建 bean.如果将应用程序自己的工厂对象交给 Spring 管理,那么 Spring 管理的就不是普通的 bean,而是工厂 Bean。单例模式：保证全局只有唯一一个对象。适配器模式：SpringAOP 的 Advice 有如下：BeforeAdvice、AfterAdvice、AfterAdvice，而需要将这些增强转为 aop 框架所需的对应的拦截器 MethodBeforeAdviceInterceptor、AfterReturningAdviceInterceptor、ThrowsAdviceInterceptor。代理模式：Spring 的 Proxy 模式在 aop 中有体现，比如 JdkDynamicAopProxy 和 Cglib2AopProxy。装饰者模式：如 HttpServletRequestWrapper，自定义请求包装器包装请求，将字符编码转换的工作添加到 getParameter()方法中。观察者模式：如启动初始化 Spring 时的 ApplicationListener 监听器。"2.Dubbo 源码使用了哪些设计模式？"责任链模式：责任链中的每个节点实现 Filter 接口，然后由 ProtocolFilterWrapper，将所有 Filter 串连起来。Dubbo 的许多功能都是通过 Filter 扩展实现的，比如监控、日志、缓存、安全、telnet 以及 RPC 本身都是。观察者模式：消费者在初始化的时候回调用 subscribe 方法，注册一个观察者，如果观察者引用的服务地址列表发生改变，就会通过 NotifyListener 通知消费者。装饰器模式：比如 ProtocolFilterWrapper 类是对 Protocol 类的修饰。工厂模式：如 ExtenstionLoader.getExtenstionLoader(Protocol.class).getAdaptiveExtenstion()。"3.举出一个例子，在这种情况你会更倾向于使用抽象类，而不是接口？"这是很常用但又是很难回答的设计面试问题。接口和抽象类都遵循”面向接口而不是实现编码”设计原则，它可以增加代码的灵活性，可以适应不断变化的需求。下面有几个点可以帮助你回答这个问题：在一些对时间要求比较高的应用中，倾向于使用抽象类，它会比接口稍快一点。如果希望把一系列行为都规范在类继承层次内，并且可以更好地在同一个地方进行编码，那么抽象类是一个更好的选择。有时，接口和抽象类可以一起使用，接口中定义函数，而在抽象类中定义默认的实现。"4.工厂模式与抽象工厂模式的区别？"首先来看看这两者的定义区别：工厂模式：定义一个用于创建对象的借口，让子类决定实例化哪一个类抽象工厂模式：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类个人觉得这个区别在于产品，如果产品单一，最合适用工厂模式，但是如果有多个业务品种、业务分类时，通过抽象工厂模式产生需要的对象是一种非常好的解决方式。再通俗深化理解下：工厂模式针对的是一个产品等级结构，抽象工厂模式针对的是面向多个产品等级结构的。再来看看工厂方法模式与抽象工厂模式对比：工厂方法模式抽象工厂模式针对的是一个产品等级结构针对的是面向多个产品等级结构一个抽象产品类多个抽象产品类可以派生出多个具体产品类每个抽象产品类可以派生出多个具体产品类一个抽象工厂类，可以派生出多个具体工厂类一个抽象工厂类，可以派生出多个具体工厂类每个具体工厂类只能创建一个具体产品类的实例每个具体工厂类可以创建多个具体产品类的实例"5.给我一个符合开闭原则的设计模式的例子？开闭原则要求你的代码对扩展开放，对修改关闭。这个意思就是说，如果你想增加一个新的功能，你可以很容易的在不改变已测试过的代码的前提下增加新的代码。有好几个设计模式是基于开闭原则的，如策略模式，如果你需要一个新的策略，只需要实现接口，增加配置，不需要改变核心逻辑。一个正在工作的例子是 Collections.sort()方法，这就是基于策略模式，遵循开闭原则的，你不需为新的对象修改 sort()方法，你需要做的仅仅是实现你自己的 Comparator 接口。6.OOP 中的组合、聚合和关联有什么区别？如果两个对象彼此有关系，就说他们是彼此相关联的。组合和聚合是面向对象中的两种形式的关联。组合是一种比聚合更强力的关联。组合中，一个对象是另一个的拥有者，而聚合则是指一个对象使用另一个对象。如果对象 A 是由对象 B 组合的，则 A 不存在的话，B 一定不存在，但是如果 A 对象聚合了一个对象 B，则即使 A 不存在了，B 也可以单独存在。7.适配器模式和代理模式之间有什么不同？这个问题与前面的类似，适配器模式和代理模式的区别在于他们的意图不同。由于适配器模式和代理模式都是封装真正执行动作的类，因此结构是一致的，但是适配器模式用于接口之间的转换，而代理模式则是增加一个额外的中间层，以便支持分配、控制或智能访问。8.适配器模式与装饰器模式有什么区别？"虽然适配器模式和装饰器模式的结构类似，但是每种模式的出现意图不同。适配器模式被用于桥接两个接口，而装饰模式的目的是在不修改类的情况下给类增加新的功能。装饰者模式：动态地将责任附加到对象上，若要扩展功能，装饰者模提供了比继承更有弹性的替代方案。通俗的解释：装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。适配器模式：将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间。适配器模式有三种：类的适配器模式、对象的适配器模式、接口的适配器模式。通俗的说法：适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。举例如下：1、适配器模式//file 为已定义好的文件流 FileInputStreamfileInput=newFileInputStream(file);InputStreamReaderinputStreamReader=newInputStreamReader(fileInput);以上就是适配器模式的体现，FileInputStream 是字节流，而并没有字符流读取字符的一些 api，因此通过 InputStreamReader 将其转为 Reader 子类，因此有了可以操作文本的文件方法。2、装饰者模式 BufferedReaderbufferedReader=newBufferedReader(inputStreamReader);构造了缓冲字符流，将 FileInputStream 字节流包装为 BufferedReader 过程就是装饰的过程，刚开始的字节流 FileInputStream 只有 read 一个字节的方法，包装为 inputStreamReader 后，就有了读取一个字符的功能，在包装为 BufferedReader 后，就拥有了 read 一行字符的功能。"9.适配器模式是什么？什么时候使用？适配器模式（AdapterPattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。适配器模式提供对接口的转换。如果你的客户端使用某些接口，但是你有另外一些接口，你就可以写一个适配去来连接这些接口。10.简述一下你了解的 Java 设计模式（总结）"标星号的为常用设计模式★单例模式：保证某个类只能有一个唯一实例，并提供一个全局的访问点。★简单工厂：一个工厂类根据传入的参数决定创建出那一种产品类的实例。工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。抽象工厂：创建一组相关或依赖对象族，比如创建一组配套的汉堡可乐鸡翅。★建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造，最后再 build。★原型模式：通过复制现有的实例来创建新的实例，减少创建对象成本（字段需要复杂计算或者创建成本高）。★适配器模式：将一个类的方法接口转换成我们希望的另外一个接口。★组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构。（无限层级的知识点树）★装饰模式：动态的给对象添加新的功能。★代理模式：为对象提供一个代理以增强对象内的方法。亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象（Integer 中的少量缓存）。★外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化（比如插座和充电器，他们之间相插是固定的，但是至于插座是插在 220V 还是 110V，充电器是充手机还是 pad 可以自主选择）。★模板方法模式：定义一个算法步骤，每个小步骤由子类各自实现。解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。★策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。★状态模式：允许一个对象根据其内部状态改变而改变它的行为。★观察者模式：被观测的对象发生改变时通知它的所有观察者。备忘录模式：保存一个对象的某个状态，以便在适当的时候恢复对象。中介者模式：许多对象利用中介者来进行交互，将网状的对象关系变为星状的（最少知识原则）。命令模式：将命令请求封装为一个对象，可用于操作的撤销或重做。访问者模式：某种物体的使用方式是不一样的，将不同的使用方式交给访问者，而不是给这个物体。（例如对铜的使用，造币厂造硬币。雕刻厂造铜像，不应该把造硬币和造铜像的功能交给铜自己实现，这样才能解耦）★责任链模式：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。"11.说说你所熟悉或听说过的 j2ee 中的几种常用模式?"IO 流的装饰器模式，Web 过滤器的责任链模式，Spring 的单例模式和工厂模式，Spring 中根据不同配置方式进行初始化的策略模式"12.设计模式的类型"根据设计模式的参考书 DesignPatterns–ElementsofReusableObject-OrientedSoftware（中文译名：设计模式–可复用的面向对象软件元素）中所提到的，总共有 23 种设计模式。这些模式可以分为三大类：创建型模式（CreationalPatterns）、结构型模式（StructuralPatterns）、行为型模式（BehavioralPatterns）。当然，我们还会讨论另一类设计模式：

J2EE 设计模式。序号模式&描述包括 1 创建型模式这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。工厂模式（FactoryPattern）抽象工厂模式（AbstractFactoryPattern）单例模式（SingletonPattern）建造者模式（BuilderPattern）原型模式（PrototypePattern）2 结构型模式这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。适配器模式（AdapterPattern）桥接模式（BridgePattern）过滤器模式（Filter、CriteriaPattern）组合模式（CompositePattern）装饰器模式（DecoratorPattern）外观模式（FacadePattern）享元模式（FlyweightPattern）代理模式（ProxyPattern）3 行为型模式这些设计模式特别关注对象之间的通信。责任链模式（ChainofResponsibilityPattern）命令模式（CommandPattern）解释器模式（InterpreterPattern）迭代器模式（IteratorPattern）中介者模式（MediatorPattern）备忘录模式（MementoPattern）观察者模式（ObserverPattern）状态模式（StatePattern）空对象模式（NullObjectPattern）策略模式（StrategyPattern）模板模式（TemplatePattern）访问者模式（VisitorPattern）4J2EE 模式这些设计模式特别关注表示层。这些模式是由 SunJavaCenter 鉴定的。MVC 模式（MVCPattern）业务代表模式（BusinessDelegatePattern）组合实体模式（CompositeEntityPattern）数据访问对象模式（DataAccessObjectPattern）前端控制器模式（FrontControllerPattern）拦截过滤器模式（InterceptingFilterPattern）服务定位器模式（ServiceLocatorPattern）传输对象模式（TransferObjectPattern）"13.Java 怎么实现单例模式？"懒汉式：懒加载，线程不安全 publicclassSingleton{privatestaticSingletonsingleton;privateSingleton(){}publicstaticSingletongetInstance(){if(singleton==null)singleton=newSingleton();returnsingleton;}}COPY 懒汉式线程安全版：同步效率低 publicclassSingleton{privatestaticSingletonsingleton;privateSingleton(){}publicsynchronizedstaticSingletongetInstance(){if(singleton==null)singleton=newSingleton();returnsingleton;}}COPY 饿汉式：publicclassSingleton{privatestaticSingletonsingleton=newSingleton();privateSingleton(){}publicstaticSingletongetInstance(){returnsingleton;}}COPY 饿汉式变种：publicclassSingleton{privatestaticSingletonsingleton;static{singleton=newSingleton();}privateSingleton(){}publicstaticSingletongetInstance(){returnsingleton;}}COPY 静态内部类方式:利用 JVM 的加载机制，当使用到 SingletonHolder 才会进行初始化。

publicclassSingleton{privateSingleton(){}privatestaticclassSingletonHolder{privatestaticfinalSingletonsingleton=newSingleton();}publicstaticSingletongetInstance(){returnSingletonHolder.singleton;}}COPY 枚举：publicenumSingletons{INSTANCE;//此处表示单例对象里面的各种方法 publicvoidMethod(){}}COPY 双重校验锁:publicclassSingleton{privatevolatilestaticSingletonsingleton;privateSingleton(){}publicstaticSingletongetInstance(){if(singleton==null){synchronized(Singleton.class){if(singleton==null){singleton=newSingleton();}}}returnsingleton;}}"14.设计模式六大原则？"1、开闭原则（OpenClosePrinciple）开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。2、里氏代换原则（LiskovSubstitutionPrinciple）里氏代换原则(LiskovSubstitutionPrincipleLSP)面向对象设计的基本原则之一。里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。——FromBaidu 百科 3、依赖倒转原则（DependenceInversionPrinciple）这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。4、接口隔离原则（InterfaceSegregationPrinciple）这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。5、迪米特法则（最少知道原则）（DemeterPrinciple）为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。6、合成复用原则（CompositeReusePrinciple）原则是尽量使用合成/聚合的方式，而不是使用继承"15.接口是什么？为什么要使用接口而不是直接使用具体类？接口用于定义 API。它定义了类必须得遵循的规则。同时，它提供了一种抽象，因为客户端只使用接口，这样可以有多重实现，如 List 接口，你可以使用可随机访问的 ArrayList，也可以使用方便插入和删除的 LinkedList。接口中不允许写代码，以此来保证抽象，但是 Java8 中你可以在接口声明静态的默认方法，这种方法是具体的。十九、数据结构与算法 1.谈一谈，id 全局唯一且自增，如何实现？"SnowFlake 雪花算法雪花 ID 生成的是一个 64 位的二进制正整数，然后转换成 10 进制的数。64 位二进制数由如下部分组成：snowflakeid 生成规则 1 位标识符：始终是 0，由于 long 基本类型在 Java 中是带符号的，最高位是符号位，正数是 0，负数是 1，所以 id 一般是正数，最高位是 0。41 位时间戳：41 位时间截不是存储当前时间的时间截，而是存储时间截的差值（当前时间截开始时间截)得到的值，这里的的开始时间截，一般是我们的 id 生成器开始使用的时间，由我们程序来指定的。10 位机器标识码：可以部署在 1024 个节点，如果机器分机房（IDC）部署，这 10 位可以由 5 位机房 ID+5 位机器 ID 组成。12 位序列：毫秒内的计数，12 位的计数顺序号支持每个节点每毫秒(同一机器，同一时间截)产生 4096 个 ID 序号"2.什么是 B+树?"什么是 B+树?B 树的变种，拥有 B 树的特点独有特点：节点中的关键字与子树数目相同。关键字对应的子树节点都大于等于该关键字，子树包含该关键字自身。所有关键字都出现在叶节点之中。所有叶节点都有指向下一个叶节点的指针。搜索：只在叶节点搜索。叶子节点保存关键字和对应的数据，非叶节点只保存关键字和指向叶节点的指针，同等关键字数量的 B 树和 B+树，B+树更小。更适合做索引系统，原因：由于叶节点有指针项链，B+树更适合做范围检索。由于非叶节点只保存关键字和指向叶节点的指针，B+树可以容纳更多的关键字，树层数变小，磁盘查询次数更低。B+树的查询效率比较稳定，查询所有关键字的路径相同。（MySQL 索引就提供了 B+树的实现方式）"3.什么是 B 树?"B 树是一种多叉树，也叫多路搜索树，适合用于文件索引上，减少磁盘 IO 次数，子节点存储最大数成为 B 树的阶，图中为 2-3 树。m 阶 B 树特点：非叶节点最多有 m 棵子树。根节点最少有两棵子树，非根非叶节点最少有 m/2 棵子树。非叶节点保存的关键字个数等于该节点子树个数-1。非叶节点保存的关键字大小有序。节点中每个关键字左子树的关键字都小于该该关键字，右子树的关键字都大于该该关键字。所有叶节点都在同一层。查找：对节点关键字进行二分查找。如果找不到，进入对应的子树进行二分查找，如此循环。"4.为什么要设计后缀表达式，有什么好处？后缀表达式又叫逆波兰表达式，逆波兰记法不需要括号来标识操作符的优先级。5.请你讲讲 LRU 算法的实现原理？"①LRU（Leastrecentlyused，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也很高”，反过来说“如果数据最近这段时间一直都没有访问,那么将来被访问的概率也会很低”，两种理解是一样的；常用于页面置换算法，为虚拟页式存储管理服务。②达到这样一种情形的算法是最理想的：每次调换出的页面是所有内存页面中最迟将被使用的；这可以最大限度的推迟页面调换，这种算法，被称为理想页面置换算法。可惜的是，这种算法是无法实现的。为了尽量减少与理想算法的差距，产生了各种精妙的算法，最近最少使用页面置换算法便是其中一个。LRU 算法的提出，是基于这样一个事实：在前面几条指令中使用频繁的页面很可能在后面的几条指令中频繁使用。反过来说，已经很久没有使用的页面很可能在未来较长的一段时间内不会被用到。这个，就是著名的局部性原理——比内存速度还要快的 cache，也是基于同样的原理运行的。因此，我们只需要在每次调换时，找到最近最少使用的那个页面调出内存。"6.如何在一个 1 到 100 的整数数组中找到丢失的数字?"如果是丢了一个数字，用个遍历把这些数字累加求和，然后用（1+100）*100/2 减去这个累加的总和，就是少的那一个数.如果是丢了一些数字，方法一：先 1-100 遍历创建一个字典，key 为 1-100，值默认都为 NO。然后把那一些数字作为一个数组，判断是否包含每一个 key，包含那 key，则那 key 的值改为 YES，最后值为 NO 的数则为缺失的数字方法二：先排序,并创建一个用来装缺失数的空数组，排好序后遍历,最大的数用 101 减，其余用后一个值减去前一个值如果差值不是 1 而是为 n，就把被减数分别加 1 到（n-1）得出的数保存下来就是缺少的数字"7.二分查找了解过吗？"查找思路【a】待查找有序数组序列：1,2,3,4,5,6,7 起始：定义 start=0,end=6,mid=(start+end)/2=(0+6)/2=3,arr[mid]=arr[3]=4【b】假设需要查找""2"",因为 2arr[mid]=arr[3]=4,所以需要将 start 移动

到 mid 右边一个位置，即 start=mid+1=4,此时重新计算 mid=(start+end)/2=(4+6)/2=5,arr[mid]=arr[5]=6,因为 7>arr[mid]=arr[5]=6,所以还是需要将 start 移动到 mid 右边一个位置，即 start=mid+1=5+1=6,此时重新计算 mid=(start+end)/2=(6+6)/2=6.arr[6]=7,此时 arr[mid]=arr[6]=7,刚好等于待查找数字 7，说明成功找到数字""7""所在的位置.【d】假设查找""0"",因为 0<arr[mid]=arr[3]=4,所以需要将 end 移动到 mid 左边一个位置，即 end=mid−1=3−1=2,此时重新计算 mid=(start+end)/2=(0+2)/2=1,arr[mid]=arr[1]=2,因为 0<arr[mid]=arr[1]=2,所以需要将 end 移动到 mid 左边一个位置，即 end=mid−1=1−1=0,此时 mid=(start+end)/2=(0+0)/2=0,arr[mid]=arr[0]=1,因为 0end，即 start 已经大于 end 结束位置，说明没有找到相应的元素 0。算法实现 publicclassBinarySearchUtils{/***根据指定值查找在数组中的位置 **@paramarr 待查找有序数组*@paramvalue 指定值*@return 返回值在数组中对应的下标位置*/publicstaticintbinarySearch(int[]arr,intvalue){//起始位置 intstart=0;//结束位置 intend=arr.length-1;while(true){//计算中间位置下标 intmid=(start+end)/2;//中间值 intmidValue=arr[mid];if(value==midValue){returnmid;}else{//待查找数值比中间值小,需要将 if(midValue>value){end=mid-1;}else{//待查找数值比中间值大,需要将 start=mid+1start=mid+1;}}if(start>end){//start>end,说明未找到相应的元素,返回-1return-1;}}}}"8.数组和链表的区别"1、数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。2、链表恰好相反，链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了。"9.介绍一下，堆排序的原理是什么？"堆排序就是把最大堆堆顶的最大数取出，将剩余的堆继续调整为最大堆，再次将堆顶的最大数取出，这个过程持续到剩余数只有一个时结束。在堆中定义以下几种操作：（1）最大堆调整（Max-Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点。（2）创建最大堆（Build-Max-Heap）：将堆所有数据重新排序，使其成为最大堆。（3）堆排序（Heap-Sort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算"10.如何知道二叉树的深度？"实现二叉树的深度方式有两种，递归以及非递归。①递归实现：为了求树的深度，可以先求其左子树的深度和右子树的深度，可以用递归实现，递归的出口就是节点为空。返回值为 0；②非递归实现：利用层次遍历的算法，设置变量 level 记录当前节点所在的层数，设置变量 last 指向当前层的最后一个节点，当处理完当前层的最后一个节点，让 level 指向+1 操作。设置变量 cur 记录当前层已经访问的节点的个数，当 cur 等于 last 时，表示该层访问结束。层次遍历在求树的宽度、输出某一层节点，某一层节点个数，每一层节点个数都可以采取类似的算法。树的宽度：在树的深度算法基础上，加一个记录访问过的层节点个数最多的变量 max,在访问每层前 max 与 last 比较，如果 max 比较大，max 不变，如果 max 小于 last，把 last 赋值给 max;"11.TreeMap 和 TreeSet 在排序时如何比较元素？Collections 工具类中的 sort()方法如何比较元素？TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元素的 compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap 要求存放的键值对映射的键必须实现 Comparable 接口从而根据键对元素进行排序。Collections 工具类的 sort 方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现 Comparable 接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是 Comparator 接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java 中对函数式编程的支持）。12.什么是算法？"算法简单来说就是解决问题的步骤。在 Java 中，算法通常都是由类的方法来实现的。前面的数据结构，比如链表为啥插入、删除快，而查找慢，平衡的二叉树插入、删除、查找都快，这都是实现这些数据结构的算法所造成的。后面我们讲的各种排序实现也是算法范畴的重要领域。一、算法的五个特征①、有穷性：对于任意一组合法输入值，在执行又穷步骤之后一定能结束，即：算法中的每个步骤都能在有限时间内完成。②、确定性：在每种情况下所应执行的操作，在算法中都有确切的规定，使算法的执行者或阅读者都能明确其含义及如何执行。并且在任何条件下，算法都只有一条执行路径。③、可行性：算法中的所有操作都必须足够基本，都可以通过已经实现的基本操作运算有限次实现之。④、有输入：作为算法加工对象的量值，通常体现在算法当中的一组变量。有些输入量需要在算法执行的过程中输入，而有的算法表面上可以没有输入，实际上已被嵌入算法之中。⑤、有输出：它是一组与“输入”有确定关系的量值，是算法进行信息加工后得到的结果，这种确定关系即为算法功能。二、算法的设计原则①、正确性：首先，算法应当满足以特定的“规则说明”方式给出的需求。其次，对算法是否“正确”的理解可以有以下四个层次：一、程序语法错误。二、程序对于几组输入数据能够得出满足需要的结果。三、程序对于精心选择的、典型、苛刻切带有刁难性的几组输入数据能够得出满足要求的结果。四、程序对于一切合法的输入数据都能得到满足要求的结果。PS：通常以第三层意义的正确性作为衡量一个算法是否合格的标准。②、可读性：算法为了人的阅读与交流，其次才是计算机执行。因此算法应该易于人的理解；另一方面，晦涩难懂的程序易于隐藏较多的错误而难以调试。③、健壮性：当输入的数据非法时，算法应当恰当的做出反应或进行相应处理，而不是产生莫名其妙的输出结果。并且，处理出错的方法不应是中断程序执行，而是应当返回一个表示错误或错误性质的值，以便在更高的抽象层次上进行处理。④、高效率与低存储量需求：通常算法效率值得是算法执行时间；存储量是指算法执行过程中所需要的最大存储空间，两者都与问题的规模有关。前面三点正确性，可读性和健壮性相信都好理解。对于第四点算法的执行效率和存储量，我们知道比较算法的时候，可能会说“A 算法比 B 算法快两倍”之类的话，但实际上这种说法没有任何意义。因为当数据项个数发生变化时，A 算法和 B 算法的效率比例也会发生变化，比如数据项增加了 50%，可能 A 算法比 B 算法快三倍，但是如果数据项减少了 50%，可能 A 算法和 B 算法速度一样。所以描述算法的速度必须要和数据项的个数联系起来。也就是“大 O”表示法，它是一种算法复杂度的相对表示方式，这里我简单介绍一下，后面会根据具体的算法来描述。相对(relative)：你只能比较相同的事物。你不能把一个做算数乘法的算法和排序整数列表的算法进行比较。但是，比较 2 个算法所做的算术操作（一个做乘法，一个做加法）将会告诉你一些有意义的东西；表示(representation)：大 O(用它最简单的形式)把算法间的比较简化为了一个单一变量。这个变量的选择基于观察或假设。例如，排序算法之间的对比通常是基于比较操作(比较 2 个结点来决定这 2 个结点的相对顺序)。这里面就假设了比较操作的计算开销很大。但是，如果比较操作的计算开销不大，而交换操作的计算开销很大，又会怎么样呢？这就改变了先前的比较方式；复杂度(complexity)：如果排序 10,000 个元素花费了我 1 秒，那么排序 1 百万个元素会花多少时间？在这个例子里，复杂度就是相对其他东西的度量结果。然后我们在说说算法的存储量，包括：程序本身所占空间；输入数据所占空间；辅助变量所占空间；一个算法的效率越高越好，而存储量是越低越好。"二十、微服务 1.作为服务注册中心，Eureka 比 Zookeeper 好在哪里?"著名的 CAP 理论指出，一个分布式系统不可能同时满足 C(一致性)、A(可用性)和 P(分区容错性)。由于分区容错性 P 在是分布式系统中必须要保证的，因此我们只能在 A 和 C 之间进行权衡。因此，Zookeeper 保证的是 CP,Eureka 则是 AP。Zookeeper 保证 CP 当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接 down 掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是 zk 会出现这样一种情况，当 master 节点因为网络故障与其他节点失去联系时，剩余节点会重新进行 leader 选举。问题在于，选举 leader 的时间太长，30~120s,且选举期间整个 zk 集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得 zk 集群失去 master 节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。Eureka 保证 APEureka 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)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka 还有一种自我保护机制，如果在 15 分钟内超过 85%的节点都没有正常的心跳，那么 Eureka 就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务 Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)当网络稳定时，当前实例新的注册信息会被同步到其它节点中因此，Eureka 可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像

zookeeper 那样使整个注册服务瘫痪。"2.Eureka 的基本架构是什么？"SpringCloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现(请对比 Zookeeper)。Eureka 采用了 C-S 的设计架构。EurekaServer 作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用 Eureka 的客户端连接到 EurekaServer 并维持心跳连接。这样系统的维护人员就可以通过 EurekaServer 来监控系统中各个微服务是否正常运行。SpringCloud 的一些其他模块（比如 Zuul）就可以通过 EurekaServer 来发现系统中的其他微服务，并执行相关的逻辑。Eureka 包含两个组件：EurekaServer 和 EurekaClientEurekaServer 提供服务注册服务各个节点启动后，会在 EurekaServer 中进行注册，这样 EurekaServer 中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到 EurekaClient 是一个 Java 客户端用于简化 EurekaServer 的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向 EurekaServer 发送心跳(默认周期为 30 秒)。如果 EurekaServer 在多个心跳周期内没有接收到某个节点的心跳，EurekaServer 将会从服务注册表中把这个服务节点移除（默认 90 秒）"3.什么是 Eureka 服务注册与发现？Eureka 是 Netflix 的一个子模块，也是核心模块之一。Eureka 是一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。服务注册与发现对于微服务架构来说是非常重要的，有了服务发现与注册，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件了。功能类似于 dubbo 的注册中心，比如 Zookeeper。4.你所知道的微服务技术栈有哪些？"服务开发 Springboot、Spring、SpringMVC 服务配置与管理 Netflix 公司的 Archaius、阿里的 Diamond 等服务注册与发现 Eureka、Consul、Zookeeper 等服务调用 Rest、RPC、gRPC 服务熔断器 Hystrix、Envoy 等负载均衡 Ribbon、Nginx 等服务接口调用(客户端调用服务的简化工具)Feign 等消息队列 Kafka、RabbitMQ、ActiveMQ 等服务配置中心管理 SpringCloudConfig、Chef 等服务路由（API 网关）Zuul 等服务监控 Zabbix、Nagios、Metrics、Spectator 等全链路追踪 Zipkin，Brave、Dapper 等服务部署 Docker、OpenStack、Kubernetes 等数据流操作开发包 SpringCloudStream（封装与 Redis,Rabbit、Kafka 等发送接收消息）事件消息总线 SpringCloudBus"5.什么是服务熔断，什么是服务降级"服务熔断熔断机制是应对雪崩效应的一种微服务链路保护机制。当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回”错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在 SpringCloud 框架里熔断机制通过 Hystrix 实现。Hystrix 会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是 5 秒内 20 次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand。服务降级其实就是线程池中单个线程障处理，防止单个线程请求时间太长，导致资源长期被占有而得不到释放，从而导致线程池被快速占用完，导致服务崩溃。Hystrix 能解决如下问题：请求超时降级，线程资源不足降级，降级之后可以返回自定义数据线程池隔离降级，分布式服务可以针对不同的服务使用不同的线程池，从而互不影响自动触发降级与恢复实现请求缓存和请求合并"6.请谈谈对 SpringBoot 和 SpringCloud 的理解"SpringBoot 专注于快速方便的开发单个个体微服务。SpringCloud 是关注全局的微服务协调整理治理框架，它将 SpringBoot 开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务 SpringBoot 可以离开 SpringCloud 独立使用开发项目，但是 SpringCloud 离不开 SpringBoot，属于依赖的关系。SpringBoot 专注于快速、方便的开发单个微服务个体，SpringCloud 关注全局的服务治理框架。SpringBoot 可以离开 SpringCloud 独立使用开发项目，但是 SpringCloud 离不开 SpringBoot，属于依赖的关系。"7.微服务之间是如何通讯的？"第一种：远程过程调用（RemoteProcedureInvocation）直接通过远程过程调用来访问别的 service。示例：REST、gRPC、Apache、Thrift 优点：简单，常见。因为没有中间件代理，系统更简单缺点：只支持请求/响应的模式，不支持别的，比如通知、请求/异步响应、发布/订阅、发布/异步响应降低了可用性，因为客户端和服务端在请求过程中必须都是可用的第二种：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。示例：ApacheKafka、RabbitMQ 优点:把客户端和服务端解耦，更松耦合提高可用性，因为消息中间件缓存了消息，直到消费者可以消费支持很多通信机制比如通知、请求/异步响应、发布/订阅、发布/异步响应缺点:消息中间件有额外的复杂性"8.什么是微服务？"微服务架构是一种架构模式或者说是一种架构风格，它提倡将单一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中，服务之间互相协调、互相配合，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（通常是基于 HTTP 的 RESTfulAPI）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。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。从技术维度来说：微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地去耦合,每一个微服务提供单个业务功能的服务，一个服务做一件事，从技术角度看就是一种小而独立的处理过程，类似进程概念，能够自行单独启动或销毁，拥有自己独立的数据库。"9.SpringCloud 和 dubbo 的区别?"（1）服务调用方式 dubbo 是 RPCspringcloudRestApi（2）注册中心,dubbo 是 zookeeperspringcloud 是 eureka，也可以是 zookeeper（3）服务网关,dubbo 本身没有实现，只能通过其他第三方技术整合，springcloud 有 Zuul 路由网关，作为路由服务器，进行消费者的请求分发,springcloud 支持断路器，与 git 完美集成配置文件支持版本控制，事物总线实现配置文件的更新与服务自动装配等等一系列的微服务架构要素。"10.服务注册和发现是什么意思？SpringCloud 如何实现？当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。11.SpringCloud 解决了哪些问题？"与分布式系统相关的复杂性–包括网络问题，延迟开销，带宽问题，安全问题。处理服务发现的能力–服务发现允许集群中的进程和服务找到彼此并进行通信。解决冗余问题–冗余问题经常发生在分布式系统中。负载平衡–改进跨多个计算资源（例如计算机集群，网络链接，中央处理单元）的工作负载分布。减少性能问题–减少因各种操作开销导致的性能问题。"12.单片，SOA 和微服务架构有什么区别？"单片架构类似于大容器，其中应用程序的所有软件组件组装在一起并紧密封装。一个面向服务的架构（SOA）是一种相互通信服务的集合。通信可以涉及简单的数据传递，也可以涉及两个或多个协调某些活动的服务。微服务架构是一种架构风格，它将应用程序构建为以业务域为模型的小型自治服务集合。"13.微服务有哪些特点？"解耦–系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展组件化–微服务被视为可以轻松更换和升级的独立组件业务能力–微服务非常简单，专注于单一功能自治–开发人员和团队可以彼此独立工作，从而提高速度持续交付–通过软件创建，测试和批准的系统自动化，允许频繁发布软件责任–微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品分散治理–重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可以自由选择最有用的工具来解决他们的问题敏捷–微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃"14.微服务有哪些优缺点？"优点：独立的可扩展性，每个微服务都可以独立进行横向或纵向扩展，根据业务实际增长情况来进行快速扩展；独立的可升级性，每个微服务都可以独立进行服务升级、更新，不用依赖于其它服务，结合持续集成工具可以进行持续发布，开发人员就可以独立快速完成服务升级发布流程；易维护性，每个微服务的代码均只专注于完成该单个业务范畴的事情，因此微服务项目代码数量将减少至 IDE 可以快速加载的大小，这样可以提高了代码的可读性，进而可以提高研发人员的生产效率；语言无关性，研发人员可以选用自己最为熟悉的语言和框架来完成他们的微服务项目（当然，一般根据每个公司的实际技术栈需要来了），这样在面对新技术或新框架的选用时，微服务能够更好地进行快速响应；故障和资源的隔离性，在系统中出现不好的资源操作行为时，例如内存泄露、数据库连接未关闭等情况，将仅仅只会影响单个微服务；优化跨团队沟通，如果要完全实践微服务架构设计风格，研发团队势必会按照新的原则来进行划分，由之前的按照技能、职能划分的方式变为按照业务（单个微服务）来进行划分，如此这般团队里将有各个方向技能的研发人员，沟通效率上来说要优于之前按照技能进行划分的组织架构；原生基于“云”的系统架构设计，基于微服务架构设计风格，我们能构建出来原生对于“云”具备超高友好度的系统，与常用容器工具如 Docker 能够很方便地结合，构建持续发布系统与 IaaS、PaaS 平台对接，使其能够方便的部署于各类“云”上，如公用云、私有云以及混合云。缺点：增加了系统复杂性；运维难度增加；本地调用变成 RPC 调用，接口耗时增加；可能会引入分布式事务。"二十一、消息队列 1.为什么使用消息队列？"面试官心理分析其实面试官主要是想看看：第一，你知不知道你们系统里为什么要用消息队列这个东西？不少候选人，说自己项目里用了

Redis、MQ，但是其实他并不知道自己为什么要用这个东西。其实说白了，就是为了用而用，或者是别人设计的架构，他从头到尾都没思考过。没有对自己的架构问过为什么的人，一定是平时没有思考的人，面试官对这类候选人印象通常很不好。因为面试官担心你进了团队之后只会木头木脑的干呆活儿，不会自己思考。第二，你既然用了消息队列这个东西，你知不知道用了有什么好处&坏处？你要是没考虑过这个，那你盲目弄个 MQ 进系统里，后面出了问题你是不是就自己溜了给公司留坑？你要是没考虑过引入一个技术可能存在的弊端和风险，面试官把这类候选人招进来了，基本可能就是挖坑型选手。就怕你干 1 年挖一堆坑，自己跳槽了，给公司留下无穷后患。第三，既然你用了 MQ，可能是某一种 MQ，那么你当时做没做过调研？你别傻乎乎的自己拍脑袋看个人喜好就瞎用了一个 MQ，比如 Kafka，甚至都从没调研过业界流行的 MQ 到底有哪几种。每一个 MQ 的优点和缺点是什么。每一个 MQ 没有绝对的好坏，但是就是看用在哪个场景可以扬长避短，利用其优势，规避其劣势。如果是一个不考虑技术选型的候选人招进了团队，leader 交给他一个任务，去设计个什么系统，他在里面用一些技术，可能都没考虑过选型，最后选的技术可能并不一定合适，一样是留坑。面试题剖析其实就是问问你消息队列都有哪些使用场景，然后你项目里具体是什么场景，说说你在这个场景里用消息队列是什么？面试官问你这个问题，期望的一个回答是说，你们公司有个什么业务场景，这个业务场景有个什么技术挑战，如果不用 MQ 可能会很麻烦，但是你现在用了 MQ 之后带给了你很多的好处。先说一下消息队列常见的使用场景吧，其实场景有很多，但是比较核心的有 3 个：解耦、异步、削峰。解耦看这么个场景。A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃……image-20210814191517000 在这个场景中，A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来？头发都白了啊！如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。image-20210814191530422 总结：通过一个 MQ，Pub/Sub 发布订阅消息这么一个模型，A 系统就跟其它系统彻底解耦了。面试技巧：你需要去考虑一下你负责的系统中是否有类似的场景，就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦，也是可以的，你就需要去考虑在你的项目里，是不是可以运用这个 MQ 去进行系统的解耦。在简历中体现出来这块东西，用 MQ 作解耦。异步再来看一个场景，A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3+300+450+200=953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求，等待个 1s，这几乎是不可接受的。image-20210814191550645 一般互联网类的企业，对于用户直接的操作，一般要求是每个请求都必须在 200ms 以内完成，对用户几乎是无感知的。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3+5=8ms，对于用户而言，其实感觉上就是点个按钮，8ms 以后就直接返回了，爽！网站做得真好，真快 image-20210814191602862 削峰每天 0:00 到 12:00，A 系统风平浪静，每秒并发请求数量就 50 个。结果每次一到 12:00~13:00，每秒并发请求数量突然会暴增到 5k+条。但是系统是直接基于 MySQL 的，大量的请求涌入 MySQL，每秒钟对 MySQL 执行约 5k 条 SQL。一般的 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。但是高峰期一过，到了下午的时候，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。image-20210814191623027 如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在中午高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。image-20210814191637646 这个短暂的高峰期积压是 ok 的，因为高峰期过了之后，每秒钟就 50 个请求进 MQ，但是 A 系统依然会按照每秒 2k 个请求的速度在处理。所以说，只要高峰期一过，A 系统就会快速将积压的消息给解决掉。"2.如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时怎么解决？"（一）、大量消息在 mq 里积压了几个小时了还没解决几千万条数据在 MQ 里积压了七八个小时，从下午 4 点多，积压到了晚上很晚，10 点多，11 点多这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复 consumer 的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。一个消费者一秒是 1000 条，一秒 3 个消费者是 3000 条，一分钟是 18 万条，1000 多万条，所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概 1 小时的时间才能恢复过来。一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：先修复 consumer 的问题，确保其恢复消费速度，然后将现有 cnosumer 都停掉。新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍或者 20 倍的 queue 数量。然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的 consumer 机器来消费消息。（二）、消息队列过期失效问题假设你用的是 rabbitmq，rabbitmq 是可以设置过期时间的，就是 TTL，如果消息在 queue 中积压超过一定的时间就会被 rabbitmq 给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在 mq 里，而是大量的数据会直接搞丢。这个情况下，就不是说要增加 consumer 消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上 12 点以后，用户都睡觉了。这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。也只能是这样了。假设 1 万个订单积压在 mq 里面，没有处理，其中 1000 个订单都丢了，你只能手动写程序把那 1000 个订单给查出来，手动发到 mq 里去再补一次。(三)、消息队列满了怎么搞？如果走的方式是消息积压在 mq 里，那么如果你很长时间都没处理掉，此时导致 mq 都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。"3.各种 MQ 的比较"特性 ActiveMQRabbitMQRocketMQKafka 单机吞吐量万/秒万/秒 10 万/秒 10 万/秒 topic 对吞吐量的影响无无 topic 达到几百/几千个级别，吞吐量会有小幅下降；这是 rocket 的最大优势所以非常适用于支撑大批量 topic 场景 topic 可以达到几十/几百个级别，吞吐量会有大幅下降 kafka 不适用大批量 topic 场景，除非加机器时效性毫秒微秒这是 rabbit 最大优势，延迟低毫秒毫秒可用性高。主从架构高。主从架构非常高。分布式。多副本，不会丢数据，非常高。分布式。不会数据，不会不可用。可靠性有较低概率丢失数据一-经配置优化可达到 0 丢失经配置优化可达到 0 丢失功能特性功能齐全，但已不怎么维护 erlang 开发，并发强，性能极好，延迟低 MQ 功能较为齐全，扩展好功能简单，主要用于大数据实时计算和日志采集，事实标准综上，总结如下：activeMQ 优点：技术成熟，功能齐全，历史悠久，有大量公司在使用缺点：偶尔会有较低概率丢失数据，而且社区已经不怎么维护 5.15.X 版本使用场景：主要用于系统解耦和异步处理，不适用与大数据量吞吐情况。互联网公司很少适用 rabitMQ 优点：吞吐量高，功能齐全，管理界面易用，社区活跃，性能极好，；缺点：吞吐量只是万级，erlang 难以二次开发和掌控；集群动态扩展非常麻烦；使用场景：吞吐量不高而要求低延迟，并且不会频繁调整和扩展的场景。非常适合国内中小型互联网公司适用，因为管理界面非常友好，可以在界面进行配置和优化/集群监控。rocketMQ 优点：支持百千级大规模 topic。吞吐量高（十万级，日处理上百亿）。接口易用。分布式易扩展，阿里支持。java 开发易于掌控缺点：与阿里（社区）存在绑定。不兼容 M 规范。使用场景：高吞吐量 kafka 优点：超高吞吐量，超高可用性和可靠性，分布式易扩展缺点：topic 支持少，MQ 功能简单，消息可能会重复消费影响数据精确度使用场景：超高"4.消息队列积压怎么办"当消费者出现异常，很容易引起队列积压，如果一秒钟 1000 个消息，那么一个小时就是几千万的消息积压，是非常可怕的事情，但是生产线上又有可能会出现；当消息积压来不及处理，rabbitMQ 如果设置了消息过期时间，那么就有可能

由于积压无法及时处理而过期，这消息就被丢失了；解决方法如下：不建议在生产环境使用数据过期策略，一是数据是否丢失无法控制，二是一旦积压就很有可能丢失；建议数据的处理都有代码来控制；当出现消息积压时，做法就是临时扩大 consumer 个数,让消息快速消费,一般都是通过业务逻辑的手段来完成如下:rabbitmq 解决积压范例修复 consumer 代码故隙，确保 consumer 逻辑正确可以消费停止 consumer，开启 10 倍 20 倍的 queue 个数创建一个临时的 consumer 程序，消费积压的 queue，并把消息写入到扩建 10 倍的 queue 中再开启 10 倍 20 倍的 consumer 对新的扩充后队列进行消费这种做法相当于通过物理资源扩充了 10 们来快速消费当消费完成后，需要恢复原有架构，开启原来的 consumer 进行正常消费 kafka 解决范例修复 consumer 代码故障，确保 consumer 逻辑正确可以消费停止 consumer，新建 topic，新建 10 倍 20 倍的 partition 个数创建对应原 topic 的 partition 个数的临时的 consumer 程序，消费原来的 topic，并把消息写入到扩建的新 topic 中再开启对应新 partition 个数的 consumer 对新的 topic 进行消费这种做法相当于通过物理资源扩充了 10 倍来快速消费"5.消息如何保证幂等性例如 kafka 的 offset 可能是消费者批量处理后才提交到 zk，重启后再消费时就可能会收到重复消息，需要消费者在处理消息时做幂等性设计，即先判断是否消费过，把已消费的放到本地缓存或者 redis 中，每次消费时先做个判断即可。6.Kafka 的消息是有序的吗？如果保证 Kafka 消息的顺序性？Kafka 只能保证局部有序，即只能保证一个分区里的消息有序。而其具体实现是通过生产者为每个分区的消息维护一个发送队列，我们需要将保证顺序的消息都发送到同一个分区中。并且由于 Kafka 会同时发送多个消息，所以还需指定 max.in.flight.requests.per.connection 为 1，保证前一个消息发送成功，后一个消息才开始发送。7.使用消息队列，如果处理重复消息？"1）利用数据库的唯一约束实现幂等 2）为更新的数据设置前置条件（CAS）3）记录并检查操作（在发送消息时，给每条消息指定一个全局唯一的 ID，消费时，先根据这个 ID 检查这条消息是否有被消费过，如果没有消费过，才更新数据，然后将消费状态置为已消费。）"8.使用消息队列，怎么确保消息不丢失？在生产阶段，你需要捕获消息发送的错误，并重发消息。在存储阶段，你可以通过配置刷盘和复制相关的参数，让消息写入到多个副本的磁盘上，来确保消息不会因为某个 Broker 宕机或者磁盘损坏而丢失。在消费阶段，你需要在处理完全部消费业务逻辑之后，再发送消费确认。9.消息队列的弊端有哪些？数据延迟；增加系统复杂度；可能产生数据不一致的问题。10.消息队列有哪些应用场景？异步处理、流量控制、服务解耦、消息广播二十二、Docker1.在非 Linux 操作系统平台上如何运行 Docker?"容器化虚拟技术概念可能来源于，在 Linux 内核版本 2.6.24 上加入的对命名空间（namespace）的技术支持特性。容器化进程加入其进程 ID 到其创建的每个进程上并且对每个进程中的系统级调用进行访问控制及审查。其本身是由系统级调用 clone()克隆出来的进程，允许其创建属于自己命名空间的进程实例，而区别于之前的，归属与整个本机系统的进程实例。如果上述在 Linux 系统内核上的技术实现成为可能，那么明显的问题是如何在非 Linux 系统上运行容器化的 Docker。过去，Mac 和 Windows 系统上运行 Docker 容器都使用 Linux 虚拟机（VMs）技术，Docker 工具箱使用的容器运行在 VirtualBox 虚拟机上。现在，最新的情况是，Windows 平台上使用的是 Hyper-V 产品技术，Mac 平台上使用的是 Hypervisor.framework（框架）产品技术。"2.在 Windows 系统上可以运行原生的 Docker 容器吗？"在‘WindowsServer2016’系统上，你可以运行 Windows 的原生容器，微软推出其映像是‘WindowsNanoServer’，一个轻量级的运行在容器中的 Windows 原生系统。您可以在其中布署基于.NET 的应用。译注：结合 Docker 的基本技术原理，参考后面的问题 26 和问题 27，可推测，微软在系统内核上开发了对 Docker 的支持，支持其闭源系统的容器化虚拟技术。但译者认为，Windows 系统本就是闭源紧耦合的系统，好像你在本机上不装.NET 组件，各应用能很好运行似的。何必再弄个容器，浪费资源。这只是译者自己之孔见，想喷就喷！另：WindowsServer2016 版本之后的都可支持这种原生 Docker 技术，如 WindowsServer2018 版。"3.什么是孤儿卷及如何删除它？孤儿卷是未与任何容器关联的卷。在 Dockerv1.9 之前的版本中，删除这些孤儿卷存在很大问题。4.在使用 Docker 技术的产品中如何监控其运行"Docker 在产品中提供如运行统计和 Docker 事件的工具。可以通过这些工具命令获取 Docker 运行状况的统计信息或报告。Dockerstats：通过指定的容器 id 获取其运行统计信息，可获得容器对 CPU，内存使用情况等的统计信息，类似 Linux 系统中的 top 命令。Dockerevents：Docker 事件是一个命令，用于观察显示运行中的 Docker 一系列的行为活动。一般的 Docker 事件有：attach（关联），commit（提交），die（僵死），detach（取消关联），rename（改名），destory（销毁）等。也可使用多个选项对事件记录筛选找到想要的事件信息。"5.Docker 群（Swarm）是什么 DockerSwarm—Docker 群—是原生的 Docker 集群服务工具。它将一群 Docker 主机集成为单一一个虚拟 Docker 主机。利用一个 Docker 守护进程，通过标准的 DockerAPI 和任何完善的通讯工具，DockerSwarm 提供透明地将 Docker 主机扩散到多台主机上的服务。6.如何临时退出一个正在交互的容器的终端，而不终止它？按 Ctrl+p，后按 Ctrl+q，如果按 Ctrl+c 会使容器内的应用进程终止，进而会使容器终止。7.如何清理批量后台停止的容器？使用 dockerrm$（sudodockerps-a-q）8.如何停止所有正在运行的容器？使用 dockerkill$(sudodockerps-q)9.DockerImage 和 DockerLayer(层)有什么不同"Image：一个 DockerImage 是由一系列 Docker 只读层（read-onlyLayer）创建出来的。Layer：在 Dockerfile 配置文件中完成的一条配置指令，即表示一个 Docker 层（Layer）。如下 Dockerfile 文件包含 4 条指令，每条指令创建一个层（Layer）。FROMubuntu:15.04COPY./appRUNmake/appCMDpython/app/app.pyCOPY 重点，每层只对其前一层进行一（某）些进化。"10.有什么方法确定一个 Docker 容器运行状态"使用如下命令行命令确定一个 Docker 容器的运行状态 dockerps–aCOPY 这将列表形式输出运行在主机上的所有 Docker 容器及其运行状态。从这个列表中很容易找到想要的容器及其运行状态。"11.如何使用 Docker 技术创建与环境无关的容器系统？"Docker 技术有三中主要的技术途径辅助完成此需求：存储卷（Volumes）环境变量（Environmentvariable）注入只读（Read-only）文件系统"12.启动 nginx 容器（随机端口映射），并挂载本地文件目录到容器 html 的命令？Dockerrun-d-p--namenginx2-v/home/nginx:/usr/share/nginx/htmlnginx13.容器与主机之间的数据拷贝命令？"Dockercp 命令用于穷奇与主机之间的数据拷贝主机到哦容器：dockercp/www96f7f14e99ab:/www/容器到主机：dockercp96f7f14e99ab:/www/tmp"14.Docker 的常用命令？"命令备注 dockerpull 拉去或更新指定的镜像 dockerpush 将镜像推送到远程仓库 dockerrm 删除容器 dockerrmi 删除镜像 dockerimages 列出所有镜像 dockerps 列出所有容器"15.DockerFile 中的命令 COPY 和 ADD 命令有什么区别？COPY 和 ADD 的区别时 COPY 的 SRC 只能是本地文件，其他用法一致。16.DockerFile 中最常见的指定是什么?"指令备注 FROM 指定基础镜像 LABEL 功能为镜像指定标签 RUN 运行指定命令 CMD 容器启动时要运行的命令"17.Docker 容器有几种状态四种状态：运行、已暂停、重新启动、已退出。18.什么是 Docker 容器 Docker 容器包括应用程序及其所有依赖项，作为操作系统的独立进程运行。19.什么是 Docker 镜像 Docker 镜像是 Docker 容器的源代码，Docker 镜像用于创建容器。使用 build 命令创建镜像。20.Docker 与虚拟机有何不同"Docker 不是虚拟化方法。它依赖于实际实现基于容器的虚拟化或操作系统级虚拟化的其他工具。为此，Docker 最初使用 LXC 驱动程序，然后移动到 libcontainer 现在重命名为 runc。Docker 主要专注于在应用程序容器内自动部署应用程序。应用程序容器旨在打包和运行单个服务，而系统容器则设计为运行多个进程，如虚拟机。因此，Docker 被视为容器化系统上的容器管理或应用程序部署工具。A 容器不需要引导操作系统内核，因此可以在不到一秒的时间内创建容器。此功能使基于容器的虚拟化比其他虚拟化方法更加独特和可取。B 由于基于容器的虚拟化为主机增加了很少或没有开销，因此基于容器的虚拟化具有接近本机的性能。C 对于基于容器的虚拟化，与其他虚拟化不同，不需要其他软件。D 主机上的所有容器共享主机的调度程序，从而节省了额外资源的需求。E 与虚拟机映像相比，容器状态（Docker 或 LXC 映像）的大小很小，因此容器映像很容易分发。F 容器中的资源管理是通过 cgroup 实现的。Cgroups 不允许容器消耗比分配给它们更多的资源。虽然主机的所有资源都在虚拟机中可见，但无法使用。这可以通过在容器和主机上同时运行 top 或 htop 来实现。所有环境的输出看起来都很相似。"21.什么是 DockerDocker 是一个容器化平台，它以容器的形式将您的应用程序及其所有依赖项打包在一起，以确保您的应用程序在任何环境中无缝运行。二十三、Dubbo1.说说 Dubbo 服务暴露的过程。Dubbo 会在 Spring 实例化完 bean 之后，在刷新容器最后一步发布 ContextRefreshEvent 事件的时候，通知实现了 ApplicationListener 的 ServiceBean 类进行回调 onApplicationEvent 事件方法，Dubbo 会在这个方法中调用 ServiceBean 父类 ServiceConfig 的 export 方法，而该方法真正实现了服务的（异步或者非异步）发布。2.Dubbo 的管理控制台能做什么？管理控制台主要包含：路由规则，动态配置，服务降级，访问控制，权重调整，负载均衡，等管理功能。3.Dubbo 必须依赖的包有哪些？Dubbo 必须依赖 JDK，其他为可选。4.服务读写推荐的容错策略是怎样的？"读操作建议使用 Failover 失败自动切换，默认重试两次其他服务器。写操作建议使用 Failfast 快速失败，发一次调用失败就立即报错。"5.如何解决服务调用链过长的问题？Dubbo 可以使用

Pinpoint 和 ApacheSkywalking(Incubator)实现分布式服务追踪，当然还有其他很多方案。6.服务提供者能实现失效踢出是什么原理？服务失效踢出基于 Zookeeper 的临时节点原理。7.Dubbo 如何优雅停机？Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果使用 kill-9PID 等强制关闭指令，是不会执行优雅停机的，只有通过 killPID 时，才会执行。8.Dubbo 支持服务降级吗？Dubbo2.2.0 以上版本支持。9.Dubbo 支持分布式事务吗？目前暂时不支持，后续可能采用基于 JTA/XA 规范实现。10.Dubbo 服务之间的调用是阻塞的吗？"默认是同步等待结果阻塞的，支持异步调用。Dubbo 是基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小，异步调用会返回一个 Future 对象。"11.Dubbo 可以对结果进行缓存吗？可以，Dubbo 提供了声明式缓存，用于加速热门数据的访问速度，以减少用户加缓存的工作量。12.服务上线怎么兼容旧版本？可以用版本号（version）过渡，多个不同版本的服务注册到注册中心，版本号不同的服务相互间不引用。这个和服务分组的概念有一点类似。13.当一个服务接口有多种实现时怎么做？当一个接口有多种实现时，可以用 group 属性来分组，服务提供方和消费方都指定同一个 group 即可。14.Dubbo 支持服务多协议吗？Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。15.注册了多个同一样的服务，如果测试指定的某一个服务呢？可以配置环境点对点直连，绕过注册中心，将以服务接口为单位，忽略注册中心的提供者列表。16.Dubbo 默认使用的是什么通信框架，还有别的选择吗？Dubbo 默认使用 Netty 框架，也是推荐的选择，另外内容还集成有 Mina、Grizzly。17.Dubbo 推荐使用什么序列化框架，你知道的还有哪些？推荐使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。18.Dubbo 启动时如果依赖的服务不可用会怎样？Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，默认 check="true"，可以通过 check="false"关闭检查。19.在 Provider 上可以配置的 Consumer 端的属性有哪些？"1）timeout：方法调用超时 2）retries：失败重试次数，默认重试 2 次 3）loadbalance：负载均衡算法，默认随机 4）actives 消费者端，最大并发调用限制"20.Dubbo 有哪几种配置方式？"1）Spring 配置方式 2）JavaAPI 配置方式"21.Dubbo 默认使用什么注册中心，还有别的选择吗？推荐使用 Zookeeper 作为注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。22.Dubbo 内置了哪几种服务容器？"SpringContainerJettyContainerLog4jContainer"23.Dubbo 需要 Web 容器吗？不需要，如果硬要用 Web 容器，只会增加复杂性，也浪费资源。24.dubbo 都支持什么协议，推荐用哪种？"dubbo://（推荐）rmi://hessian://http://webservice://thrift://memcached://redis://rest://"25.Dubbo 和 Dubbox 有什么区别？Dubbox 是继 Dubbo 停止维护后，当当网基于 Dubbo 做的一个扩展项目，如加了服务可 Restful 调用，更新了开源组件等。26.为什么要用 Dubbo？"因为是阿里开源项目，国内很多互联网公司都在用，已经经过很多线上考验。内部使用了 Netty、Zookeeper，保证了高性能高可用性。使用 Dubbo 可以将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，可用于提高业务复用灵活扩展，使前端应用能更快速的响应多变的市场需求。"27.Dubbo 是什么？Dubbo 是阿里巴巴开源的基于 Java 的高性能 RPC 分布式服务框架，现已成为 Apache 基金会孵化项目。二十四、Elasticsearch1.如何监控 Elasticsearch 集群状态？Marvel 让你可以很简单的通过 Kibana 监控 Elasticsearch。你可以实时查看你的集群健康状态和性能，也可以分析过去的集群、索引和节点指标。2.在并发情况下，Elasticsearch 如果保证读写一致？"（1）可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；（2）另外对于写操作，一致性级别支持 quorum/one/all，默认为 quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。（3）对于读操作，可以设置 replication 为 sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置 replication 为 async 时，也可以通过设置搜索请求参数_preference 为 primary 来查询主分片，确保文档是最新版本。"3.对于 GC 方面，在使用 Elasticsearch 时要注意什么？"（1）倒排词典的索引需要常驻内存，无法 GC，需要监控 datanode 上 segmentmemory 增长趋势。（2）各类缓存，fieldcache,filtercache,indexingcache,bulkqueue 等等，要设置合理的大小，并且要应该根据最坏的情况来看 heap 是否够用，也就是各类缓存全部占满的时候，还有 heap 空间可以分配给其他任务吗？避免采用 clearcache 等“自欺欺人”的方式来释放内存。（3）避免返回大量结果集的搜索与聚合。确实需要大量拉取数据的场景，可以采用 scan&scrollapi 来实现。（4）clusterstats 驻留内存并无法水平扩展，超大规模集群可以考虑分拆成多个集群通过 tribenode 连接。（5）想知道 heap 够不够，必须结合实际应用场景，并对集群的 heap 使用情况做持续的监控。（6）根据监控数据理解内存需求，合理配置各类 circuitbreaker，将内存溢出风险降低到最低"4.在 Elasticsearch 中，是怎么根据一个词找到对应的倒排索引的？"（1）Lucene 的索引过程，就是按照全文检索的基本过程，将倒排表写成此文件格式的过程。（2）Lucene 的搜索过程，就是按照此文件格式将索引进去的信息读出来，然后计算每篇文档打分(score)的过程。"5.客户端在和集群连接时，如何选择特定的节点执行请求？TransportClient 利用 transport 模块远程连接一个 elasticsearch 集群。它并不加入到集群中，只是简单的获得一个或者多个初始化的 transport 地址，并以轮询的方式与这些地址进行通信。6.Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法？"（1）64GB 内存的机器是非常理想的，但是 32GB 和 16GB 机器也是很常见的。少于 8GB 会适得其反。（2）如果你要在更快的 CPUs 和更多的核心之间选择，选择更多的核心更好。多个内核提供的额外并发远胜过稍微快一点点的时钟频率。（3）如果你负担得起 SSD，它将远远超出任何旋转介质。基于 SSD 的节点，查询和索引性能都有提升。如果你负担得起，SSD 是一个好的选择。（4）即使数据中心们近在咫尺，也要避免集群跨越多个数据中心。绝对要避免集群跨越大的地理距离。（5）请确保运行你应用程序的 JVM 和服务器的 JVM 是完全一样的。在 Elasticsearch 的几个地方，使用 Java 的本地序列化。（6）通过设置 gateway.recover_after_nodes、gateway.expected_nodes、gateway.recover_after_time 可以在集群重启的时候避免过多的分片交换，这可能会让数据恢复从数个小时缩短为几秒钟。（7）Elasticsearch 默认被配置为使用单播发现，以防止节点无意中加入集群。只有在同一台机器上运行的节点才会自动组成集群。最好使用单播代替组播。（8）不要随意修改垃圾回收器（CMS）和各个线程池的大小。（9）把你的内存的（少于）一半给 Lucene（但不要超过 32GB!），通过 ES_HEAP_SIZE 环境变量设置。（10）内存交换到磁盘对服务器性能来说是致命的。如果内存交换到磁盘上，一个 100 微秒的操作可能变成 10 毫秒。再想想那么多 10 微秒的操作时延累加起来。不难看出 swapping 对于性能是多么可怕。（11）Lucene 使用了大量的文件。同时，Elasticsearch 在节点和 HTTP 客户端之间进行通信也使用了大量的套接字。所有这一切都需要足够的文件描述符。你应该增加你的文件描述符，设置一个很大的值，如 64,000。"7.说说 Elasticsearch 常用的调优手段？"设计阶段调优（1）根据业务增量需求，采取基于日期模板创建索引，通过 rolloverAPI 滚动索引；（2）使用别名进行索引管理；（3）每天凌晨定时对索引做 force_merge 操作，以释放空间；（4）采取冷热分离机制，热数据存储到 SSD，提高检索效率；冷数据定期进行 shrink 操作，以缩减存储；（5）采取 curator 进行索引的生命周期管理；（6）仅针对需要分词的字段，合理的设置分词器；（7）Mapping 阶段充分结合各个字段的属性，是否需要检索、是否需要存储等。写入调优（1）写入前副本数设置为 0；（2）写入前关闭 refresh_interval 设置为-1，禁用刷新机制；（3）写入过程中：采取 bulk 批量写入；（4）写入后恢复副本数和刷新间隔；（5）尽量使用自动生成的 id。查询调优（1）禁用 wildcard；（2）禁用批量 terms（成百上千的场景）；（3）充分利用倒排索引机制，能 keyword 类型尽量 keyword；（4）数据量大时候，可以先基于时间敲定索引再检索；（5）设置合理的路由机制。其他调优部署调优，业务调优等。上面的提及一部分，面试者就基本对你之前的实践或者运维经验有所评估了。"8.Elasticsearch 中的分析器是什么？"在 ElasticSearch 中索引数据时，数据由为索引定义的 Analyzer 在内部进行转换。分析器由一个 Tokenizer 和零个或多个 TokenFilter 组成。编译器可以在一个或多个 CharFilter 之前。分析模块允许您在逻辑名称下注册分析器，然后可以在映射定义或某些 API 中引用它们。Elasticsearch 附带了许多可以随时使用的预建分析器。或者，您可以组合内置的字符过滤器，编译器和过滤器器来创建自定义分析器。"9.Elasticsearch 中的倒排索引是什么？倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索。倒排索引是一种像数据结构一样的散列图，可将用户从单词导向文档或网页。它是搜索引擎的核心。其主要目标是快速搜索从数百万文件中查找数据。10.什么是 ElasticSearch？Elasticsearch 是一个基于 Lucene 的搜索引擎。它提供了具有 HTTPWeb 界面和无架构 JSON 文档的分布式，多租户能力的全文搜索引擎。Elasticsearch 是用 Java 开发的，根据 Apache 许可条款作为开源发布。11.ElasticSearch 中的分片是什么?"在大多数环境中，每个节点都在单独的盒子或虚拟机上运行。索引–在 Elasticsearch 中，索引是文档的集合。分片-因为 Elasticsearch 是一个分布式搜索引擎，所以索引通常被分割成分布在多个节点上的被称为分片的元素。"12.ElasticSearch 中的集群、节点、索引、文档、类

型是什么？"群集是一个或多个节点（服务器）的集合，它们共同保存您的整个数据，并提供跨所有节点的联合索引和搜索功能。群集由唯一名称标识，默认情况下为“elasticsearch”。此名称很重要，因为如果节点设置为按名称加入群集，则该节点只能是群集的一部分。节点是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。索引就像关系数据库中的“数据库”。它有一个定义多种类型的映射。索引是逻辑名称空间，映射到一个或多个主分片，并且可以有零个或多个副本分片。MySQL=>数据库，ElasticSearch=>索引。文档类似于关系数据库中的一行。不同之处在于索引中的每个文档可以具有不同的结构（字段），但是对于通用字段应该具有相同的数据类型。MySQL=>Databases=>Tables=>Columns/Rows，ElasticSearch=>Indices=>Types=>具有属性的文档。类型是索引的逻辑类别/分区，其语义完全取决于用户。"13.在并发情况下，Elasticsearch 如果保证读写一致？"可以通过版本号使用乐观并发控制，以确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；另外对于写操作，一致性级别支持 quorum/one/all，默认为 quorum，即只有当大多数分片可用时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被认为故障，分片将会在一个不同的节点上重建。对于读操作，可以设置 replication 为 sync(默认)，这使得操作在主分片和副本分片都完成后才会返回；如果设置 replication 为 async 时，也可以通过设置搜索请求参数_preference 为 primary 来查询主分片，确保文档是最新版本。"14.Elasticsearch 对于大数据量（上亿量级）的聚合如何实现？"Elasticsearch 提供的首个近似聚合是 cardinality 度量。它提供一个字段的基数，即该字段的 distinct 或者 unique 值的数目。它是基于 HLL 算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。其特点是：可配置的精度，用来控制内存的使用（更精确 = 更多内存）；小的数据集精度是非常高的；我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。"15.详细描述一下 Elasticsearch 搜索的过程"搜索被执行成一个两阶段过程，我们称之为 QueryThenFetch；在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。每个分片在本地执行搜索并构建一个匹配文档的大小为 from+size 的优先队列。PS：在搜索的时候是会查询 FilesystemCache 的，但是有部分数据还在 MemoryBuffer，所以搜索是近实时的。每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。接下来就是取回阶段，协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分片加载并丰富文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调节点返回结果给客户端。补充：QueryThenFetch 的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档数量较少的时候可能不够准确，DFSQueryThenFetch 增加了一个预查询的处理，询问 Term 和 Documentfrequency，这个评分更准确，但是性能会变差。"16.详细描述一下 Elasticsearch 更新和删除文档的过程"删除和更新也都是写操作，但是 Elasticsearch 中的文档是不可变的，因此不能被删除或者改动以展示其变更；磁盘上的每个段都有一个相应的.del 文件。当删除请求发送后，文档并没有真的被删除，而是在.del 文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并时，在.del 文件中被标记为删除的文档将不会被写入新段。在新的文档被创建时，Elasticsearch 会为该文档指定一个版本号，当执行更新时，旧版本的文档在.del 文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是会在结果中被过滤掉。"17.详细描述一下 Elasticsearch 索引文档的过程。"协调节点默认使用文档 ID 参与计算（也支持通过 routing），以便为路由提供合适的分片。shard=hash(document_id)%(num_of_primary_shards)当分片所在的节点接收到来自协调节点的请求后，会将请求写入到 MemoryBuffer，然后定时（默认是每隔 1 秒）写入到 FilesystemCache，这个从 MomeryBuffer 到 FilesystemCache 的过程就叫做 refresh；当然在某些情况下，存在 MomeryBuffer 和 FilesystemCache 的数据可能会丢失，ES 是通过 translog 的机制来保证数据的可靠性的。其实现机制是接收到请求后，同时也会写入到 translog 中，当 Filesystemcache 中的数据写入到磁盘中时，才会清除掉，这个过程叫做 flush；在 flush 过程中，内存中的缓冲将被清除，内容被写入一个新段，段的 fsync 将创建一个新的提交点，并将内容刷新到磁盘，旧的 translog 将被删除并开始一个新的 translog。flush 触发的时机是定时触发（默认 30 分钟）或者 translog 变得太大（默认为 512M）时；"18.Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了一个 master，另外 10 个选了另一个 master，怎么办？"当集群 master 候选数量不小于 3 个时，可以通过设置最少投票通过数量（discovery.zen.minimum_master_nodes）超过所有候选节点一半以上来解决脑裂问题；当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为 data 节点，避免脑裂问题。"19.Elasticsearch 是如何实现 Master 选举的？"Elasticsearch 的选主是 ZenDiscovery 模块负责的，主要包含 Ping（节点之间通过这个 RPC 来发现彼此）和 Unicast（单播模块包含一个主机列表以控制哪些节点需要 ping 通）这两部分；对所有可以成为 master 的节点（node.master:true）根据 nodeId 字典排序，每次选举每个节点都把自己所知道节点排一次序，然后选出第一个（第 0 位）节点，暂且认为它是 master 节点。如果对某个节点的投票数达到一定的值（可以成为 master 节点数 n/2+1）并且该节点自己也选举自己，那这个节点就是 master。否则重新选举一直到满足上述条件。补充：master 节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；data 节点可以关闭 http 功能。"20.为什么要使用 Elasticsearch?因为在我们的数据，将来会非常多，所以采用以往的模糊查询，模糊查询前置配置，会放弃索引，导致商品查询是全表扫面，在百万级别的数据库中，效率非常低下，而我们使用 ES 做一个全文索引，我们将经常查询的商品的某些字段，比如说商品名，描述、价格还有 id 这些字段我们放入我们索引库里，可以提高查询速度。二十五、Java81.Lambda 表达式的参数列表与 Lambda 箭头运算符有何不同？Lambda 表达式可以一次携带零个，一个或甚至多个参数。另一方面，Lambda 箭头运算符使用图标“->”将这些参数从列表和主体中分离出来。2.是什么使 JavaSE8 优于其他？"JavaSE8 具有以下功能，使其优于其他功能：它编写并行代码。它提供了更多可用的代码。它具有改进的性能应用程序。它具有更易读和简洁的代码。它支持编写包含促销的数据库。"3.什么是 Java8 中的 MetaSpace？它与 PermGenSpace 有何不同？使用 JDK8 时，permGen 空间已被删除。那么现在将元数据信息存储在哪里？此元数据现在存储在本机内存中，称为“MetaSpace”。该内存不是连续的 Java 堆内存。它允许通过垃圾收集，自动调整，元数据并发解除分配来改进 PermGen 空间。4.Lambda 函数的优点"直到 Java8 列表和集合通常由客户端代码从集合中获取迭代器来处理，然后使用它迭代其元素并依次处理每个元素。如果要并行处理不同的元素，那么客户代码而不是集合的责任就是组织它。通过 Java8，可以更轻松地在多个线程上分发集合的处理。集合现在可以在内部组织自己的迭代，将并行化的责任从客户端代码转移到库代码中。更少的代码行。如上所述，用户必须仅以声明方式声明要执行的操作。n>System.out.println（“HelloWorld”+n）;所以用户必须键入减少的代码量。使用 Java8Lambda 表达式可以实现更高的效率。通过使用具有多核的 CPU，用户可以通过使用 lambda 并行处理集合来利用多核 CPU。"5.什么是 Lambda 表达式？"LambdaExpression 可以定义为允许用户将方法作为参数传递的匿名函数。这有助于删除大量的样板代码。Lambda 函数没有访问修饰符（私有，公共或受保护），没有返回类型声明和没有名称。Lambda 表达式允许用户将“函数”传递给代码。所以，与以前需要一整套的接口/抽象类想必，我们可以更容易地编写代码。例如，假设我们的代码具有一些复杂的循环/条件逻辑或工作流程。使用 lambda 表达式，在那些有难度的地方，可以得到很好的解决。"6.解释 Java8 中间操作与终端操作？"流操作可以分为两部分：中间操作-返回另一个 Stream 的中间操作，允许操作以查询的形式连接。终端操作-产生非流，结果如原始值，集合或根本没有值。"7.hashMap 原理，java8 做的改变"从结构实现来讲，HashMap 是数组+链表+红黑树（JDK1.8 增加了红黑树部分）实现的。HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。HashMap 非线程安全。ConcurrentHashMap 线程安全。解决碰撞：当出现冲突时，运用拉链法，将关键词为同义词的结点链接在一个单链表中，散列表长 m，则定义一个由 m 个头指针组成的指针数组 T，地址为 i 的结点插入以 T(i)为头指针的单链表中。Java8 中，冲突的元素超过限制（8），用红黑树替换链表。"8.Java8 中的可选项是什么？Java8 引入了一个新的容器类 java.util.Optional。如果该值可用，它将包装一个值。如果该值不可用，则应返回空的可选项。因此它代表空值，缺失值。这个类有各种实用方法，如 isPresent（），它可以帮助用户避免使用空值检查。由于不直接返回值，而是返回包装器对象，所以用户可以避免空指针异常。9.Java8 支持函数编程是什么意思？"在 Java8 之前，所有东西都是面向对象的。除了原语之外，java 中的所有内容都作为对象存在。对方法/函数的所有调用都是使用对象或类引用进行的。方法/功能本身并不是独立存在的。使用 Java8，引入了函数式编程。所以我们可以使用匿名函数。Java 是一种一流的面向对象语言。除了原始数据类型之外，Java 中的所有内容都是一个对象。即使是一个数组也是一个对象。每个类都创建对象的实例。没有办法只定义一个独立于 Java 的函数/方法。无法将方法作为参数传递或返回该实例的

方法体。"10.抽象类和接口的异同？"抽象类：含有 abstract 修饰符的 class 就算抽象类；它既可以有抽象方法，也可以有普通方法，构造方法，静态方法，但是不能有抽象构造方法和抽象静态方法。且如果其子类没有实现其所有的抽象方法，那么该子类也必须是抽象类；接口：他可以看成是抽象类的一个特例，使用 interface 修饰符；内部结构：jdk7：接口只有常量和抽象方法，无构造器 jdk8：接口增加了默认方法和静态方法，无构造器 jdk9：接口允许以 private 修饰的方法，无构造器共同点：不能实例化；多态方式的一种使用；不同点：抽象类是单继承的，而接口可以多继承（实现）；"11.Java8 新特性简介"代码更少（增加了新语法：Lambda 表达式）强大的 StreamAPI（集合数据的操作）最大化的减少空指针异常：Optional 类的使用接口的新特性注解的新特性集合的底层源码实现新日期时间的 api"二十六、Java 高并发 1.ForkJoin 框架"Fork/Join 框架是 Java7 提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。Fork/Join 框架要完成两件事情：1.任务分割：首先 Fork/Join 框架需要把大的任务分割成足够小的子任务，如果子任务比较大的话还要对子任务进行继续分割 2.执行任务并合并结果：分割的子任务分别放到双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都放在另外一个队列里，启动一个线程从队列里取数据，然后合并这些数据。在 Java 的 Fork/Join 框架中，使用两个类完成上述操作 1.ForkJoinTask:我们要使用 Fork/Join 框架，首先需要创建一个 ForkJoin 任务。该类提供了在任务中执行 fork 和 join 的机制。通常情况下我们不需要直接集成 ForkJoinTask 类，只需要继承它的子类，Fork/Join 框架提供了两个子类：a.RecursiveAction：用于没有返回结果的任务 b.RecursiveTask:用于有返回结果的任务 2.ForkJoinPool:ForkJoinTask 需要通过 ForkJoinPool 来执行任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务(工作窃取算法)。Fork/Join 框架的实现原理 ForkJoinPool 由 ForkJoinTask 数组和 ForkJoinWorkerThread 数组组成，ForkJoinTask 数组负责将存放程序提交给 ForkJoinPool，而 ForkJoinWorkerThread 负责执行这"2.Java 里的阻塞队列"7 个阻塞队列。分别是 ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。DelayQueue：一个使用优先级队列实现的无界阻塞队列。SynchronousQueue：一个不存储元素的阻塞队列。LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。添加元素 Java 中的阻塞队列接口 BlockingQueue 继承自 Queue 接口。BlockingQueue 接口提供了 3 个添加元素方法。add：添加元素到队列里，添加成功返回 true，由于容量满了添加失败会抛出 IllegalStateException 异常 offer：添加元素到队列里，添加成功返回 true，添加失败返回 falseput：添加元素到队列里，如果容量满了会阻塞直到容量不满删除方法 3 个删除方法 poll：删除队列头部元素，如果队列为空，返回 null。否则返回元素。remove：基于对象找到对应的元素，并删除。删除成功返回 true，否则返回 falsetake：删除队列头部元素，如果队列为空，一直阻塞到队列有元素并删除"3.AQS"AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 队列来完成获取资源线程的排队工作。privatevolatileintstate;//共享变量，使用 volatile 修饰保证线程可见性 COPY2 种同步方式：独占式，共享式。独占式如 ReentrantLock，共享式如 Semaphore，CountDownLatch，组合式的如 ReentrantReadWriteLock 节点的状态 CANCELLED，值为 1，表示当前的线程被取消；SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是 unpark；CONDITION，值为-2，表示当前节点在等待 condition，也就是在 condition 队列中；PROPAGATE，值为-3，表示当前场景下后续的 acquireShared 能够得以执行；值为 0，表示当前节点在 sync 队列中，等待着获取锁。模板方法模式 protectedbooleantryAcquire(intarg):独占式获取同步状态，试着获取，成功返回 true，反之为 falseprotectedbooleantryRelease(intarg)：独占式释放同步状态，等待中的其他线程此时将有机会获取到同步状态；protectedinttryAcquireShared(intarg)：共享式获取同步状态，返回值大于等于 0，代表获取成功；反之获取失败；protectedbooleantryReleaseShared(intarg)：共享式释放同步状态，成功为 true，失败为 falseAQS 维护一个共享资源 state，通过内置的 FIFO 来完成获取资源线程的排队工作。该队列由一个一个的 Node 结点组成，每个 Node 结点维护一个 prev 引用和 next 引用，分别指向自己的前驱和后继结点。双端双向链表。独占式:乐观的并发策略 acquirea.首先 tryAcquire 获取同步状态，成功则直接返回；否则，进入下一环节；b.线程获取同步状态失败，就构造一个结点，加入同步队列中，这个过程要保证线程安全；c.加入队列中的结点线程进入自旋状态，若是老二结点（即前驱结点为头结点），才有机会尝试去获取同步状态；否则，当其前驱结点的状态为 SIGNAL，线程便可安心休息，进入阻塞状态，直到被中断或者被前驱结点唤醒。releaserelease 的同步状态相对简单，需要找到头结点的后继结点进行唤醒，若后继结点为空或处于 CANCEL 状态，从后向前遍历找寻一个正常的结点，唤醒其对应线程。共享式:共享式地获取同步状态.同步状态的方法 tryAcquireShared 返回值为 int。a.当返回值大于 0 时，表示获取同步状态成功，同时还有剩余同步状态可供其他线程获取；b.当返回值等于 0 时，表示获取同步状态成功，但没有可用同步状态了；c.当返回值小于 0 时，表示获取同步状态失败。AQS 实现公平锁和非公平锁非公平锁中，那些尝试获取锁且尚未进入等待队列的线程会和等待队列 head 结点的线程发生竞争。公平锁中，在获取锁时，增加了 isFirst(current)判断，当且仅当，等待队列为空或当前线程是等待队列的头结点时，才可尝试获取锁。"4.CopyOnWriteArrayList"CopyOnWriteArrayList:写时加锁，当添加一个元素的时候，将原来的容器进行 copy，复制出一个新的容器，然后在新的容器里面写，写完之后再将原容器的引用指向新的容器，而读的时候是读旧容器的数据，所以可以进行并发的读，但这是一种弱一致性的策略。使用场景：CopyOnWriteArrayList 适合使用在读操作远远大于写操作的场景里，比如缓存。"5.Nginx 多进程模型是如何实现高并发的？"进程数与并发数不存在很直接的关系。这取决取 server 采用的工作方式。如果一个 server 采用一个进程负责一个 request 的方式，那么进程数就是并发数。那么显而易见的，就是会有很多进程在等待中。等什么？最多的应该是等待网络传输。Nginx 的异步非阻塞工作方式正是利用了这点等待的时间。在需要等待的时候，这些进程就空闲出来待命了。因此表现为少数几个进程就解决了大量的并发问题。apache 是如何利用的呢，简单来说：同样的 4 个进程，如果采用一个进程负责一个 request 的方式，那么，同时进来 4 个 request 之后，每个进程就负责其中一个，直至会话关闭。期间，如果有第 5 个 request 进来了。就无法及时反应了，因为 4 个进程都没干完活呢，因此，一般有个调度进程，每当新进来了一个 request，就新开个进程来处理。nginx 不这样，每进来一个 request，会有一个 worker 进程去处理。但不是全程的处理，处理到什么程度呢？处理到可能发生阻塞的地方，比如向上游（后端）服务器转发 request，并等待请求返回。那么，这个处理的 worker 不会这么傻等着，他会在发送完请求后，注册一个事件：“如果 upstream 返回了，告诉我一声，我再接着干”。于是他就休息去了。此时，如果再有 request 进来，他就可以很快再按这种方式处理。而一旦上游服务器返回了，就会触发这个事件，worker 才会来接手，这个 request 才会接着往下走。由于 webserver 的工作性质决定了每个 request 的大部份生命都是在网络传输中，实际上花费在 server 机器上的时间片不多。这是几个进程就解决高并发的秘密所在。webserver 刚好属于网络 io 密集型应用，不算是计算密集型。异步，非阻塞，使用 epoll，和大量细节处的优化。也正是 nginx 之所以然的技术基石。"6.常见的同步工具类？"CountDownLatch：递减计数器闭锁，直到达到某个条件时才放行，多线程可以调用 await 方法一直阻塞，直到计数器递减为零。比如我们连接 zookeeper，由于连接操作是异步的，所以可以使用 countDownLatch 创建一个计数器为 1 的锁，连接挂起，当异步连接成功时，调用 countDown 通知挂起线程；再比如 5V5 游戏竞技，只有房间人满了才可以开始游戏。FutureTask：带有计算结果的任务，在计算完成时才能获取结果，如果计算尚未完成，则阻塞 get 方法。FutureTask 将计算结果从执行线程传递到获取这个结果的线程。Semaphore：信号量，用来控制同时访问某个特定资源的数量，只有获取到许可 acquire，才能够正常执行，并在完成后释放许可，acquire 会一致阻塞到有许可或中断超时。使用信号量可以轻松实现一个阻塞队列。CyclicBarrier：类似于闭锁，它可以阻塞一组线程，只有所有线程全部到达以后，才能够继续执行，so 线程必须相互等待。这在并行计算中是很有用的，将一个问题拆分为多个独立的子问题，当线程到达栅栏时，调用 await 等待，一直阻塞到所有参与线程全部到达，再执行下一步任务。"7.常见的并发容器？"ConcurrentHashMap：使用了分段锁，锁的粒度变得更小，多线程访问时，可能都不存在锁的竞争，所以大大提高了吞吐量。简单对比来看，就好比数据库上用行锁来取代表锁，行锁无疑带来更大的并发。CopyOnWriteArrayList：写入时复制，多线程访问时，彼此不会互相干扰或被修改的线程所干扰，当然 copy 时有开销的，尤其时列表元素庞大，且写入操作频繁时，所以仅当迭代操作远远大于修改操作时，才应该考虑使用。BlockingQueue：阻塞队列提供了可阻塞的 put 和 take 方法，当队列已经满了，那么 put 操作将阻塞到队列可用，当队列为空时，take 操作会阻塞

到队列里有数据。有界的队列是一种强大的资源管理器，可以在程序负荷过载时保护应用，可作为一种服务降级的策略。阻塞队列还提供 offer 操作，当数据无法加入队列时，返回失败状态，给应用主动处理负荷过载带来更多灵活性。"8.死锁的避免与诊断？如果一个线程最多只能获取一个锁，那么就不会发生锁顺序死锁了。如果确实需要获取多个锁，锁的顺序可以按照某种规约，比如两个资源的 id 值，程序按规约保证获取锁的顺序一致。或者可以使用显式的锁 Lock，获取锁的时候设置超时时间，超时后可以重新发起，以避免发生死锁。9.什么是锁顺序死锁？两个线程试图以不同的顺序获得相同的锁，那么可能发发生死锁。比如转账问题，由 from 账户向 to 账户转账，假设每次我们先同步 from 对象，再同步 to 账户，然后执行转账操作，貌似没什么问题。如果这时候 to 账户同时向 from 账户转账，那么两个线程可能要永久等待了。10.数据库死锁？在执行一个事务时可能要获取多个锁，一直持有锁到事务提交，如果 A 事务需要获取的锁在另一个事务 B 中，且 B 事务也在等待 A 事务所持有的锁，那么两个事务之间就会发生死锁。但数据库死锁比较少见，数据库会加以干涉死锁问题，牺牲一个事务使得其他事务正常执行。11.什么是线程调度器(ThreadScheduler)和时间分片(TimeSlicing)？"线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（即最好不要让你的程序依赖于线程的优先级）。"12.有三个线程 T1，T2，T3，怎么确保它们按顺序执行？在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的 join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3 调用 T2，T2 调用 T1)，这样 T1 就会先完成而 T3 最后完成。13.如何测试并发量？可以使用 apache 提供的 ab 工具测试。14.Java 中 Unsafe 类详解"通过 Unsafe 类可以分配内存，可以释放内存；类中提供的 3 个本地方法 allocateMemory、reallocateMemory、freeMemory 分别用于分配内存，扩充内存和释放内存，与 C 语言中的 3 个方法对应。可以定位对象某字段的内存位置，也可以修改对象的字段值，即使它是私有的；挂起与恢复:将一个线程进行挂起是通过 park 方法实现的，调用 park 后，线程将一直阻塞直到超时或者中断等条件出现。unpark 可以终止一个挂起的线程，使其恢复正常。整个并发框架中对线程的挂起操作被封装在 LockSupport 类中，LockSupport 类中有各种版本 pack 方法，但最终都调用了 Unsafe.park()方法。Java 中 Unsafe 类详解"15.进程调度算法"实时系统：FIFO(FirstInputFirstOutput，先进先出算法)，SJF(ShortestJobFirst，最短作业优先算法)，SRTF(ShortestRemainingTimeFirst，最短剩余时间优先算法)。交互式系统：RR(RoundRobin，时间片轮转算法)，HPF(HighestPriorityFirst，最高优先级算法)，多级队列，最短进程优先，保证调度，彩票调度，公平分享调度。"16.同步和异步有何不同，在什么情况下分别使用它们？举例说明"如果数据将在线程间共享。例如：正在写的数据以后可能会被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须进行同步存取当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效。同步交互：指发送一个请求，需要等待返回，然后才能发送下一个请求，有个等待的过程异步交互：指发送一个请求，不需要等待返回，随时可以再发送下一个请求，即不需要等待。区别：一个需要等待，一个不需要等待"17.线程间如何通讯"锁机制：包括互斥锁、条件变量、读写锁互斥锁提供了以排他方式防止数据结构被并发修改的方法读写锁允许多个线程同时读共享数据，而对写操作是互斥的条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。信号量机制：包括无名线程信号量和命名线程信号量信号机制：类似进程间的信号处理线程间的通信目的只要是用于新城同步，所以线程没有像进程通信中的用于数据交换的通信机制。"18.进程间如何通讯"管道(pipe)管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系有名管道(namedpipe)有名管道也是半双工的通信方式，但是它云溪无亲缘关系进程间的通信。信号量(semaphore)信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。消息队列（messagequeue）消息队列里有消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递消息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点信号（signal）信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生共享内存（sharedmemory）共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。套接字（socket）套接口也是一种进程间通信机制，以其他通信机制不同的是，它可用于不同进程间的通信"19.什么是线程进程是表示自愿分配的基本单位。而线程则是进程中执行运算的最小单位，即执行处理机调度的基本单位。通俗来讲：一个程序有一个进程，而一个进程可以有多个线程。20.什么是进程进程是指运行中的应用程序，每个进程都有自己独立的地址空间（内存空间）。比如用户点击桌面的 IE 浏览器，就启动了一个进程，操作系统就会为该进程分配独立的地址空间。当用户再次点击左边的 IE 浏览器，又启动了一个进程，操作系统将为新的进程分配新的独立的地址空间。目前操作系统都支持多进程。二十七、Java 集合 1.Iterator 是什么？Iterator 接口提供遍历任何 Collection 的接口。我们可以从一个 Collection 中使用迭代器方法来获取迭代器实例。迭代器取代了 Java 集合框架中的 Enumeration。迭代器允许调用者在迭代过程中移除元素。2.为何 Map 接口不继承 Collection 接口？"尽管 Map 接口和它的实现也是集合框架的一部分，但 Map 不是集合，集合也不是 Map。因此，Map 继承 Collection 毫无意义，反之亦然。如果 Map 继承 Collection 接口，那么元素去哪儿？Map 包含 key-value 对，它提供抽取 key 或 value 列表集合的方法，但是它不适合“一组对象”规范。"3.为何 Collection 不从 Cloneable 和 Serializable 接口继承？"Collection 接口指定一组对象，对象即为它的元素。如何维护这些元素由 Collection 的具体实现决定。例如，一些如 List 的 Collection 实现允许重复的元素，而其它的如 Set 就不允许。很多 Collection 实现有一个公有的 clone 方法。然而，把它放到集合的所有实现中也是没有意义的。这是因为 Collection 是一个抽象表现。重要的是实现。当与具体实现打交道的时候，克隆或序列化的语义和含义才发挥作用。所以，具体实现应该决定如何对它进行克隆或序列化，或它是否可以被克隆或序列化。在所有的实现中授权克隆和序列化，最终导致更少的灵活性和更多的限制。特定的实现应该决定它是否可以被克隆和序列化。"4.集合框架中的泛型有什么优点？Java1.5 引入了泛型，所有的集合接口和实现都大量地使用它。泛型允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在编译时报错。这避免了在运行时出现 ClassCastException，因为你将会在编译时得到报错信息。泛型也使得代码整洁，我们不需要使用显式转换和 instanceOf 操作符。它也给运行时带来好处，因为不会产生类型检查的字节码指令。5.Java 集合框架是什么？说出一些集合框架的优点？"每种编程语言中都有集合，最初的 Java 版本包含几种集合类：Vector、Stack、HashTable 和 Array。随着集合的广泛使用，Java1.2 提出了囊括所有集合接口、实现和算法的集合框架。在保证线程安全的情况下使用泛型和并发集合类，Java 已经经历了很久。它还包括在 Java 并发包中，阻塞接口以及它们的实现。集合框架的部分优点如下：（1）使用核心集合类降低开发成本，而非实现我们自己的集合类。（2）随着使用经过严格测试的集合框架类，代码质量会得到提高。（3）通过使用 JDK 附带的集合类，可以降低代码维护成本。（4）复用性和可操作性。"6.谈谈对 HashMap 构造方法中初始容量、加载因子的理解"初始容量代表了哈希表中桶的初始数量，即 Entry<K,V>[]table 数组的初始长度。加载因子是哈希表在其容量自动增加之前可以达到多满的一种饱和度百分比，其衡量了一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。"7.HashMap 默认的初始化长度是多少？在 JDK 中默认长度是 16，并且默认长度和扩容后的长度都必须是 2 的幂。8.ArrayList 和 LinkedList 的区别？LinkedList 基于链表的数据结构；ArrayList 基于动态数组的数据结构 LinkedList 在插入和删除数据时效率更高，ArrayList 查询效率更高；9.ArrayList 和 Vector 的区别？"Vector 是线程安全的，ArrayList 是线程不安全的。Vector 在数据满时增长为原来的两倍，而 ArrayList 在数据量达到容量的一半时,增长为原容量的 1.5 倍。"10.ConcurrentHashMap 实现原理"JDK1.7:【数组（Segment）+数组（HashEntry）+链表（HashEntry 节点)】ConcurrentHashMap（分段锁）对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。JDK1.8:Node 数组+链表/红黑树利用 CAS+Synchronized 来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。"11.ConcurrentHashMap 和 Hashtable 的区

别？"ConcurrentHashMap 结合了 HashMap 和 HashTable 二者的优势。HashMap 没有考虑同步，HashTable 考虑了同步的问题。但是 HashTable 在每次同步执行时都要锁住整个结构。ConcurrentHashMap 锁的方式是稍微细粒度的。"12.HashMap 与 HashTable 的区别？"HashMap 没有考虑同步，是线程不安全的；Hashtable 使用了 synchronized 关键字，是线程安全的；HashMap 允许 K/V 都为 null；后者 K/V 都不允许为 null；"13.常见的集合底层实现"ArrayList 底层是数组。LinkedList 底层是双向链表。HashMap 底层与 HashTable 原理相同，Java8 版本以后如果同一位置哈希冲突大于 8 则链表变成红黑树。HashTable 底层是链地址法组成的哈希表（即数组+单项链表组成）。HashSet 底层是 HashMap。LinkedHashMap 底层修改自 HashMap，包含一个维护插入顺序的双向链表。TreeMap 底层是红黑树。LinkedHashSet 底层是 LinkedHashMap。TreeSet 底层是 TreeMap。"14.常见的集合有哪些？"Collection 接口的子接口包括：Set 接口和 List 接口 Map 接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap 以及 Properties 等 Set 接口的实现类主要有：HashSet、TreeSet、LinkedHashSet 等 List 接口的实现类主要有：ArrayList、LinkedList、Stack 以及 Vector 等"15.Comparable 和 Comparator 接口有何区别？"Comparable 和 Comparator 接口被用来对对象集合或者数组进行排序。Comparable 接口被用来提供对象的自然排序，我们可以使用它来提供基于单个逻辑的排序。Comparator 接口被用来提供不同的排序算法，我们可以选择需要使用的 Comparator 来对给定的对象集合进行排序。"16.Collections 类是什么？Java.util.Collections 是一个工具类仅包含静态方法，它们操作或返回集合。它包含操作集合的多态算法，返回一个由指定集合支持的新集合和其它一些内容。这个类包含集合框架算法的方法，比如折半搜索、排序、混编和逆序等。17.队列和栈是什么，列出它们的区别？"栈和队列两者都被用来预存储数据。java.util.Queue 是一个接口，它的实现类在 Java 并发包中。队列允许先进先出（FIFO）检索元素，但并非总是这样。Deque 接口允许从两端检索元素。栈与队列很相似，但它允许对元素进行后进先出（LIFO）进行检索。Stack 是一个扩展自 Vector 的类，而 Queue 是一个接口。"18.BlockingQueue 是什么？Java.util.concurrent.BlockingQueue 是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。BlockingQueue 接口是 Java 集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在 BlockingQueue 的实现类中被处理了。Java 提供了集中 BlockingQueue 的实现，比如 ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue 等。19.哪些集合类提供对元素的随机访问？ArrayList、HashMap、TreeMap 和 HashTable 类提供对元素的随机访问。20.如何决定选用 HashMap 还是 TreeMap？对于在 Map 中插入、删除和定位元素这类操作，HashMap 是最好的选择。然而，假如你需要对一个有序的 key 集合进行遍历，TreeMap 是更好的选择。基于你的 collection 的大小，也许向 HashMap 中添加元素会更快，将 map 换为 TreeMap 进行有序 key 的遍历。21.我们能否使用任何类作为 Map 的 key？"我们可以使用任何类作为 Map 的 key，然而在使用它们之前，需要考虑以下几点：（1）如果类重写了 equals()方法，它也应该重写 hashCode()方法。（2）类的所有实例需要遵循与 equals()和 hashCode()相关的规则。请参考之前提到的这些规则。（3）如果一个类没有使用 equals()，你不应该在 hashCode()中使用它。（4）用户自定义 key 类的最佳实践是使之为不可变的，这样，hashCode()值可以被缓存起来，拥有更好的性能。不可变的类也可以确保 hashCode()和 equals()在未来不会改变，这样就会解决与可变相关的问题了。"22.hashCode()和 equals()方法有何重要性？"HashMap 使用 Key 对象的 hashCode()和 equals()方法去决定 key-value 对的索引。当我们试着从 HashMap 中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同 Key 也许会产生相同的 hashCode()和 equals()输出，HashMap 将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用 hashCode()和 equals()去查找重复，所以正确实现它们非常重要。equals()和 hashCode()的实现应该遵循以下规则：（1）如果 o1.equals(o2)，那么 o1.hashCode()==o2.hashCode()总是为 true 的。（2）如果 o1.hashCode()==o2.hashCode()，并不意味着 o1.equals(o2)会为 true。"23.fail-fast 与 fail-safe 有什么区别？Iterator 的 fail-fast 属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util 包中的所有集合类都被设计为 fail-fast 的，而 java.util.concurrent 中的集合类都为 fail-safe 的。Fail-fast 迭代器抛出 ConcurrentModificationException，而 fail-safe 迭代器从不抛出 ConcurrentModificationException。24.Iterater 和 ListIterator 之间有什么区别？"（1）我们可以使用 Iterator 来遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。（2）Iterator 只可以向前遍历，而 LIstIterator 可以双向遍历。（3）ListIterator 从 Iterator 接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。"25.Enumeration 和 Iterator 接口的区别？"Enumeration 的速度是 Iterator 的两倍，也使用更少的内存。Enumeration 是非常基础的，也满足了基础的需要。但是，与 Enumeration 相比，Iterator 更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。迭代器取代了 Java 集合框架中的 Enumeration。迭代器允许调用者从集合中移除元素，而 Enumeration 不能做到。为了使它的功能更加清晰，迭代器方法名已经经过改善。"二十八、Jenkins1.什么是 BlueOcean"BlueOcean 是 pipeline 的可视化 UI。同时他兼容经典的自由模式的 job。JenkinsPipeline 从头开始设计，但仍与自由式作业兼容，BlueOcean 减少了经典模式下的混乱并为团队中的每个成员增加了清晰度。BlueOcean 的主要特点包括：连续交付（CD）管道的复杂可视化，可以让您快速直观地理解管道状态。管道编辑器–通过引导用户通过直观和可视化的过程来创建管道，从而使管道的创建变得平易近人。个性化以适应团队中每个成员的基于角色的需求。在需要干预和/或出现问题时确定精确度。BlueOcean 显示的标注了关键步骤，促进异常处理和提高生产力。"2.什么是 jenkinsfile?为什么使用 jenkinsfile"Jenkinsfile 是一个文本文件，其中包含 JenkinsPipeline 的定义，并已签入源代码管理虽然用于定义管道的脚本语法和 jenkinsfile 类似，但通常认为在项目中定义管道 Jenkinsfile 并检查源代码管理是最佳实践为所有分支和请求自动创建一个管道构建过程管道上的代码审查/迭代审核追踪管道"3.如何通过 Jenkins 克隆 Git 存储库？如果要通过 Jenkins 克隆 Git 存储库,则必须输入 Jenkins 系统的电子邮件和用户名。切换到作业目录并为此执行”gitconfig”命令。4.如何在 Jenkins 中创建备份和复制文件？如果要创建 Jenkins 设置的备份,只需复制将 Jenkins 的所有设置,构建工件和日志保存在其主目录中的目录。你还可以复制作业目录以克隆或复制作业或重命名目录。5.可以使用哪些命令手动启动 Jenkins？"你可以使用以下任何命令来手动启动 Jenkins：(Jenkins_url)/restart：强制重启,而无需等待构建完成。(Jenkin_url)/safeRestart：允许所有正在运行的构建完成。"6.Jenkins 的优势是什么？"Jenkins 的优势包括：在开发环境的早期阶段,错误跟踪很容易。提供大量的插件支持。对代码的迭代改进。构建失败会在集成阶段进行缓存。对于每个代码提交更改,都会生成一个自动生成报告通知。为了将构建报告的成功或失败通知开发人员,它与 LDAP 邮件服务器集成在一起。实现持续集成的敏捷开发和测试驱动的开发。通过简单的步骤,即可自动完成 maven 发布项目。"7.在 Jenkins 中,什么是持续集成？在软件开发中,多个开发人员或团队在同一个 Web 应用程序的不同部分上工作,因此你必须通过集成所有模块来执行集成测试。为了做到这一点,每天都要对每段代码进行自动化处理,以便对所有代码进行测试。此过程称为连续集成。8.Maven,Ant 和 Jenkins 有什么区别？Maven 和 Ant 是 BuildTechnologies,而 Jenkins 是持续集成工具。9.什么是 Jenkins？Jenkins 是一个用 Java 编写的开源持续集成工具。它跟踪版本控制系统,并在发生更改时启动和监视构建系统。二十九、Kubernetes1.ReplicaSet 和 ReplicationController 之间有什么区别？"ReplicaSet 和 ReplicationController 几乎完全相同。它们都确保在任何给定时间运行指定数量的 pod 副本。不同之处在于复制 pod 使用的选择器。ReplicaSet 使用基于集合的选择器，而 ReplicationController 使用基于权限的选择器。Equity-Based 选择器：这种类型的选择器允许按标签键和值进行过滤。因此，在外行术语中，基于 Equity 的选择器将仅查找与标签具有完全相同短语的 pod。示例：假设您的标签键表示 app=nginx，那么，使用此选择器，您只能查找标签应用程序等于 nginx 的那些 pod。Selector-Based 选择器：此类型的选择器允许根据一组值过滤键。因此，换句话说，基于 Selector 的选择器将查找已在集合中提及其标签的 pod。示例：假设您的标签键在（nginx，NPS，Apache）中显示应用程序。然后，使用此选择器，如果您的应用程序等于任何 nginx，NPS 或 Apache，则选择器将其视为真实结果。"2.什么是 Container 资源监控？对于用户而言，了解应用程序的性能和所有不同抽象层的资源利用率非常重要，Kubernetes 通过在容器，pod，服务和整个集群等不同级别创建抽象来考虑集群的管理。现在，可以监视每个级别，这只是容器资源监视。3.您对云控制器管理器有何了解？CloudControllerManager 负责持久存储，网络路由，从核心 Kubernetes 特定代码中抽象出特定于云的代码，以及管理与底层云服务的通信。它可能会分成几个不同的容器，具体取决于您运行的是哪个云平台，然后它

可以使云供应商和 Kubernetes 代码在没有任何相互依赖的情况下开发。因此，云供应商开发他们的代码并在运行 Kubernetes 时与 Kubernetes 云控制器管理器连接。4.什么是 Ingress 网络，它是如何工作的？"Ingress 网络是一组规则，充当 Kubernetes 集群的入口点。这允许入站连接，可以将其配置为通过可访问的 URL，负载平衡流量或通过提供基于名称的虚拟主机从外部提供服务。因此，Ingress 是一个 API 对象，通常通过 HTTP 管理集群中服务的外部访问，是暴露服务的最有效方式。现在，让我以一个例子向您解释 Ingress 网络的工作。有 2 个节点具有带有 Linux 桥接器的 pod 和根网络命名空间。除此之外，还有一个名为 flannel0（网络插件）的新虚拟以太网设备被添加到根网络中。现在，假设我们希望数据包从 pod1 流向 pod4。因此，数据包将 pod1 的网络保留在 eth0，并进入 veth0 的根网络。然后它被传递给 cbr0，这使得 ARP 请求找到目的地，并且发现该节点上没有人具有目的地 IP 地址。因此，桥接器将数据包发送到 flannel0，因为节点的路由表配置了 flannel0。现在，flannel 守护程序与 Kubernetes 的 API 服务器通信，以了解所有 podIP 及其各自的节点，以创建 podsIP 到节点 IP 的映射。网络插件将此数据包封装在 UDP 数据包中，其中额外的标头将源和目标 IP 更改为各自的节点，并通过 eth0 发送此数据包。现在，由于路由表已经知道如何在节点之间路由流量，因此它将数据包发送到目标节点 2。数据包到达 node2 的 eth0 并返回到 flannel0 以解封装并在根网络命名空间中将其发回。同样，数据包被转发到 Linux 网桥以发出 ARP 请求以找出属于 veth1 的 IP。数据包最终穿过根网络并到达目标 Pod4。"5.你对 Kubernetes 的负载均衡器有什么了解？负载均衡器是暴露服务的最常见和标准方式之一。根据工作环境使用两种类型的负载均衡器，即内部负载均衡器或外部负载均衡器。内部负载均衡器自动平衡负载并使用所需配置分配容器，而外部负载均衡器将流量从外部负载引导至后端容器。6.什么是 etcd？etcd 是用 Go 编程语言编写的，是一个分布式键值存储，用于协调分布式工作。因此，Etcd 存储 Kubernetes 集群的配置数据，表示在任何给定时间点的集群状态。7.你能简要介绍一下 Kubernetes 控制管理器吗？多个控制器进程在主节点上运行，但是一起编译为单个进程运行，即 Kubernetes 控制器管理器。因此，ControllerManager 是一个嵌入控制器并执行命名空间创建和垃圾收集的守护程序。它拥有责任并与 API 服务器通信以管理端点。8.kube-apiserver 和 kube-scheduler 的作用是什么？"kube-apiserver 遵循横向扩展架构，是主节点控制面板的前端。这将公开 Kubernetes 主节点组件的所有 API，并负责在 Kubernetes 节点和 Kubernetes 主组件之间建立通信。kube-scheduler 负责工作节点上工作负载的分配和管理。因此，它根据资源需求选择最合适的节点来运行未调度的 pod，并跟踪资源利用率。它确保不在已满的节点上调度工作负载。"9.您能否介绍一下 Kubernetes 中主节点的工作情况？Kubernetesmaster 控制容器存在的节点和节点内部。现在，这些单独的容器包含在容器内部和每个容器内部，您可以根据配置和要求拥有不同数量的容器。因此，如果必须部署 pod，则可以使用用户界面或命令行界面部署它们。然后，在节点上调度这些 pod，并根据资源需求，将 pod 分配给这些节点。kube-apiserver 确保在 Kubernetes 节点和主组件之间建立通信。10.你对 Kube-proxy 有什么了解？Kube-proxy 可以在每个节点上运行，并且可以跨后端网络服务进行简单的 TCP/UDP 数据包转发。基本上，它是一个网络代理，它反映了每个节点上 KubernetesAPI 中配置的服务。因此，Docker 可链接的兼容环境变量提供由代理打开的群集 IP 和端口。11.KubernetesArchitecture 的不同组件有哪些？KubernetesArchitecture 主要有两个组件–主节点和工作节点。如下图所示，master 和 worker 节点中包含许多内置组件。主节点具有 kube-controller-manager，kubeapiserver，kube-scheduler 等。而工作节点具有在每个节点上运行的 kubelet 和 kubeproxy。12.什么是 Kubelet？这是一个代理服务，它在每个节点上运行，并使从服务器与主服务器通信。因此，Kubelet 处理 PodSpec 中提供给它的容器的描述，并确保 PodSpec 中描述的容器运行正常。13.什么是 Kubectl？Kubectl 是一个平台，您可以使用该平台将命令传递给集群。因此，它基本上为 CLI 提供了针对 Kubernetes 集群运行命令的方法，以及创建和管理 Kubernetes 组件的各种方法。14.什么是 Minikube？Minikube 是一种工具，可以在本地轻松运行 Kubernetes。这将在虚拟机中运行单节点 Kubernetes 群集。15.什么是 Heapster？Heapster 是由每个节点上运行的 Kubelet 提供的集群范围的数据聚合器。此容器管理工具在 Kubernetes 集群上本机支持，并作为 pod 运行，就像集群中的任何其他 pod 一样。因此，它基本上发现集群中的所有节点，并通过机上 Kubernetes 代理查询集群中 Kubernetes 节点的使用信息。16.什么是 Google 容器引擎？GoogleContainerEngine（GKE）是 Docker 容器和集群的开源管理平台。这个基于 Kubernetes 的引擎仅支持在 Google 的公共云服务中运行的群集。17.Kubernetes 如何简化容器化部署？由于典型应用程序将具有跨多个主机运行的容器集群，因此所有这些容器都需要相互通信。因此，要做到这一点，你需要一些能够负载平衡，扩展和监控容器的东西。由于 Kubernetes 与云无关并且可以在任何公共/私有提供商上运行，因此必须是您简化容器化部署的选择。18.什么是 ContainerOrchestration？考虑一个应用程序有 5-6 个微服务的场景。现在，这些微服务被放在单独的容器中，但如果没有容器编排就无法进行通信。因此，由于编排意味着所有乐器在音乐中和谐共处，所以类似的容器编排意味着各个容器中的所有服务协同工作以满足单个服务器的需求。19.Kubernetes 与 Docker 有什么关系？众所周知，Docker 提供容器的生命周期管理，Docker 镜像构建运行时容器。但是，由于这些单独的容器必须通信，因此使用 Kubernetes。因此，我们说 Docker 构建容器，这些容器通过 Kubernetes 相互通信。因此，可以使用 Kubernetes 手动关联和编排在多个主机上运行的容器。20.什么是 Kubernetes？Kubernetes 是一个开源容器管理工具，负责容器部署，容器扩缩容以及负载平衡。作为 Google 的创意之作，它提供了出色的社区，并与所有云提供商合作。因此，我们可以说 Kubernetes 不是一个容器化平台，而是一个多容器管理解决方案。三十、Maven1.依赖的解析机制"当依赖的范围是 system 的时候，Maven 直接从本地文件系统中解析构件。根据依赖坐标计算仓库路径，尝试直接从本地仓库寻找构件，如果发现对应的构件，就解析成功。如果在本地仓库不存在相应的构件，就遍历所有的远程仓库，发现后，下载并解析使用。如果依赖的版本是 RELEASE 或 LATEST，就基于更新策略读取所有远程仓库的元数据文件（groupId/artifactId/maven-metadata.xml），将其与本地仓库的对应元合并后，计算出 RELEASE 或者 LATEST 真实的值，然后基于该值检查本地仓库，或者从远程仓库下载。如果依赖的版本是 SNAPSHOT，就基于更新策略读取所有远程仓库的元数据文件，将它与本地仓库对应的元数据合并，得到最新快照版本的值，然后根据该值检查本地仓库，或从远程仓库下载。如果最后解析得到的构件版本包含有时间戳，先将该文件下载下来，再将文件名中时间戳信息删除，剩下 SNAPSHOT 并使用（以非时间戳的形式使用）。"2.Maven 依赖冲突"每个显式声明的类包都会依赖于一些其它的隐式类包，这些隐式的类包会被 maven 间接引入进来，因而可能造成一个我们不想要的类包的载入，严重的甚至会引起类包之间的冲突。要解决这个问题，首先就是要查看 pom.xml 显式和隐式的依赖类包，然后通过这个类包树找出我们不想要的依赖类包，手工将其排除在外就可以了。例如：<exclusions><exclusion><artifactId>unitils-database</artifactId><groupId>org.unitils</groupId></exclusion></exclusions>"3.什么是 Maven 插件？"Maven 生命周期的每一个阶段的具体实现都是由 Maven 插件实现的。插件通常提供了一个目标的集合，并且可以使用下面的语法执行：mvn[plugin-name]:[goal-name]Maven 提供了下面两种类型的插件：Buildplugins：在构建时执行，并在 pom.xml 的元素中配置。Reportingplugins：在网站生成过程中执行，并在 pom.xml 的元素中配置。下面是一些常用插件的列表：clean：构建之后清理目标文件。删除目标目录。compiler：编译 Java 源文件。surefile：运行 JUnit 单元测试。创建测试报告。jar：从当前工程中构建 JAR 文件。war：从当前工程中构建 WAR 文件。javadoc：为工程生成 Javadoc。antrun：从构建过程的任意一个阶段中运行一个 ant 任务的集合。"4.如何解决 jar 冲突？"遇到冲突的时候第一步，要找到 Maven 加载的到时是什么版本的 jar 包，通过们 mvndependency:tree 查看依赖树，或者使用 IDEAMavenHelper 插件。然后，通过 Maven 的依赖原则来调整坐标在 pom 文件的申明顺序是最好的办法，或者使用将冲突中不想要的 jar 引入的 jar 进行掉。"5.Maven 依赖原则？"1、赖路径最短优先原则。一个项目 Demo 依赖了两个 jar 包，其中 A-B-C-X(1.0)，A-D-X(2.0)。由于 X(2.0)路径最短，所以项目使用的是 X(2.0)。2、pom 文件中申明顺序优先。如果 A-B-X(1.0)，A-C-X(2.0)这样的路径长度一样怎么办呢？这样的情况下，Maven 会根据 pom 文件声明的顺序加载，如果先声明了 B，后声明了 C，那就最后的依赖就会是 X(1.0)。3、覆写优先子 pom 内声明的优先于父 pom 中的依赖。"6.对于一个多模块项目，如果管理项目依赖的版本？"方式一，通过在父模块中声明和，然后让子模块通过元素指定父模块，这样子模块在定义依赖是就可以只定义 groupId 和 artifactId，自动使用父模块的 version，这样统一整个项目的依赖的版本。继承的方式。方式二，使用声明为 import 的依赖，从而引入一个 pom 的的。具体的，可以看看《MavenSpringBOM(billofmaterials)》文章。组合的方式。"7.Maven 版本规则？"Maven 主要是这样定义版本规则的：...。比如说 1.2.3，主版本是 1，次版本是 2，增量版本是 3。主版本，一般来说代表了项目的重大的架构变更，比如说 Maven1 和 Maven2，在架构上已经两

样了，将来的 Maven3 和 Maven2 也会有很大的变化。次版本，一般代表了一些功能的增加或变化，但没有架构的变化，比如说 Nexus1.3 较之于 Nexus1.2 来说，增加了一系列新的或者改进的功能（仓库镜像支持，改进的仓库管理界面等等），但从大的架构上来说，1.3 和 1.2 没什么区别。增量版本，一般是一些小的 bugfix，不会有重大的功能变化。一般来说，在我们发布一次重要的版本之后，随之会开发新的版本。比如说，myapp-1.1 发布之后，就着手开发 myapp-1.2 了。由于 myapp-1.2 有新的主要功能的添加和变化，在发布测试前，它会变得不稳定，而 myapp-1.1 是一个比较稳定的版本，现在的问题是，我们在 myapp-1.1 中发现了一些 BUG（当然在 1.2 中也存在），为了能够在一段时间内修复 BUG 并仍然发布稳定的版本，我们就会用到分支(branch)，我们基于 1.1 开启一个分支 1.1.1，在这个分支中修复 BUG，并快速发布。这既保证了版本的稳定，也能够使 bug 得到快速修复，也不同停止 1.2 的开发。只是，每次修复分支 1.1.1 中的 BUG 后，需要 merge 代码到 1.2 中。"8.Maven 有哪些优点和缺点"1）优点简化了项目依赖管理。当年，多少人被 SSH 整合搞死搞活，很多时候，是因为依赖不完整，或者版本不正确。自从 Maven 出来后，终于可以无痛了~当然，也有一部分功劳是 SpringBoot，这是后话。易于上手，对于新手可能一个 mvncleanpackage 命令就可能满足我们的工作。便于与持续集成工具(Jenkins)整合。便于项目升级，无论是项目本身升级还是项目使用的依赖升级。有助于多模块项目的开发，一个模块开发好后，发布到仓库，依赖该模块时可以直接从仓库更新，而不用自己去编译。Maven 有很多插件，便于功能扩展，比如生产站点，自动发布版本等。2）缺点 Maven 是一个庞大的构建系统，学习难度大。这里的学习，更多指的完整学习。如果基本使用，并不会存在该问题。Maven 采用约定优于配置的策略(conventionoverconfiguration)，虽然上手容易，但是一旦出了问题，难于调试。这个确实，略微痛苦。当依赖很多时，m2eclipse 老是搞得 Eclipse 很卡。使用 IDEA，而不是 Eclipse，完美解决。中国的网络环境差，很多 repository 无法访问，比如 GoogleCode、JBoss 仓库无法访问等。这个也好解决，在中增加阿里巴巴的 Maven 私服，具体可以参见《提高 Maven 速度——Maven 仓库修改成国内阿里巴巴地址》文章。"9.Maven 常用命令"mvnarchetype：create：创建 Maven 项目。mvncompile：编译源代码。mvndeploy：发布项目。mvntest-compile：编译测试源代码。mvntest：运行应用程序中的单元测试。mvnsite：生成项目相关信息的网站。mvnclean：清除项目目录中的生成结果。mvnpackage：根据项目生成的 jar/war 等。mvninstall：在本地 Repository 中安装 jar。mvneclipse:eclipse：生成 Eclipse 项目文件。mvnjetty:run 启动 Jetty 服务。mvntomcat:run：启动 Tomcat 服务。mvncleanpackage-Dmaven.test.skip=true：清除以前的包后重新打包，跳过测试类。用到最多的命令：mvneclipse:clean：清除 Project 中以前的编译的东西，重新再来。mvneclipse:eclipse：开始编译 Maven 的 Project。mvncleanpackage：清除以前的包后重新打包。"10.Maven 规约是什么？"/src/main/java/：Java 源码。/src/main/resource：Java 配置文件，资源文件。/src/test/java/：Java 测试代码。/src/test/resource：Java 测试配置文件，资源文件。/target：文件编译过程中生成的.class 文件、jar、war 等等。pom.xml：配置文件 Maven 要负责项目的自动化构建，以编译为例，Maven 要想自动进行编译，那么它必须知道 Java 的源文件保存在哪里，这样约定之后，不用我们手动指定位置，Maven 能知道位置，从而帮我们完成自动编译。遵循“>>>”。即能进行配置的不要去编码指定，能事先约定规则的不要去进行配置。这样既减轻了劳动力，也能防止出错。"11.你们项目为什么选用 Maven 进行构建？"首先，Maven 是一个优秀的项目构建工具。使用 maven，可以很方便的对项目进行分模块构建，这样在开发和测试打包部署时，效率会提高很多。其次，Maven 可以进行依赖的管理。使用 Maven，可以将不同系统的依赖进行统一管理，并且可以进行依赖之间的传递和继承。"12.Maven 的生命周期"一个项目的构建过成通常包括清理、编译、测试、打包、集成测试、验证、部署等。Maven 从中抽取了一套完善的、易扩展的生命周期。Maven 的生命周期是抽象的，其中的具体任务都交由插件来完成。Maven 为大多数构建任务编写并绑定了默认的插件。Maven 定义了三套生命周期：clean、default、site，每个生命周期都包含了一些阶段（phase）。三套生命周期相互独立，但各个生命周期中的阶段却是有顺序的，且后面的夹断依赖于前面的阶段。执行某个阶段时，其前面的阶段会依顺序执行，但不会触发另外两套生命周期中的任何阶段。"13.Maven 的坐标和依赖"maven 的坐标组成部分 groupId:组织机构 id，org.aptech.hdax，org.springframeworkartifactId:子项目编号，springmvc,spring-test,spring-coreversion:版本号，可以一直迭代，平时项目开发用的是快照版本 0.0.1-SNAPSHOTPackage：jarwarpom 项目依赖的核心概念：框架整合最害怕 jar 包冲突，之前不使用maven，经常出现这个文件。依赖范围 scope 标签进行配置 Compile:默认值，项目打包的时候会把该依赖包打进去 Test:测试依赖，只是在运行测试用例的时候会用到，打包是不打进去的 Provided:提供依赖，类似于 test 传递依赖 a->b,b->c,如果在 a 中导入到 b 的依赖，c 会自动过来依赖调解如果不同的包传递依赖了一个相同的 jar，但是版本不一致原则：最短路径第一声明优先原则排除依赖归类依赖:方便后期的依赖版本升级、降级"14.使用 Maven 好处"Maven 能提供一种项目的配置，配置好的项目，只需要运行一条简单的命令，就能完成重复的，繁琐的构建动作.Maven 能提供一种项目的依赖配置.可以自动的导入项目依赖的 jar,并且自动导入这些 jar 包依赖的第三方的 jar 包.Maven 提供了一种标准的项目目录结构，测试命名规则等项目的最佳实践方案，统一了不同项目的学习成本.maven 的常用命令 mvnclean:清理 mvncompile：编译 mvnpackage：打包 mvntest:测试，自动运行所有的测试用例 mvninstall:安装，将项目打的包安装到本地仓库，其他项目就可以依赖了 mvnjetty:run:运行jetty 插件"15.maven 是什么？"ApacheMaven 是一个软件项目管理和理解工具。基于项目对象模型（POM）的概念，Maven 可以从一个中心信息管理项目的构建，报告和文档。项目构建在 eclipse 中新建一个 WEB 工程。进行编码及编写配置文件对源代码进行编译运行，生成 class 文件打成 war 包，部署至 tomcat"三十一、MongoDB1.为什么 MongoDB 的数据文件很大？MongoDB 采用的预分配空间的方式来防止文件碎片。2.如何理解 MongoDB 中的 GridFS 机制，MongoDB 为何使用 GridFS 来存储文件？GridFS 是一种将大型文件存储在 MongoDB 中的文件规范。使用 GridFS 可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了 BSON 对象有限制的问题。3.在 MongoDB 中什么是副本集在 MongoDB 中副本集由一组 MongoDB 实例组成，包括一个主节点多个次节点，MongoDB 客户端的所有数据都写入主节点(Primary),副节点从主节点同步写入数据，以保持所有复制集内存储相同的数据，提高数据可用性。4.MongoDB 支持存储过程吗？如果支持的话，怎么用？MongoDB 支持存储过程，它是 javascript 写的，保存在 db.system.js 表中。5.在 MongoDb 中什么是索引"索引用于高效的执行查询,没有索引的 MongoDB 将扫描整个集合中的所有文档,这种扫描效率很低,需要处理大量的数据。索引是一种特殊的数据结构,将一小块数据集合保存为容易遍历的形式.索引能够存储某种特殊字段或字段集的值,并按照索引指定的方式将字段值进行排序。"6.“ObjectID”有哪些部分组成一共有四部分组成:时间戳、客户端 ID、客户进程 ID、三个字节的增量计数器 7.为什么在 MongoDB 中使用”ObjectID”数据类型"ObjectID"数据类型用于存储文档 id8.为什么要在 MongoDB 中用”RegularExpression”数据类型"RegularExpression"类型用于在文档中存储正则表达式 9.为什么要在 MongoDB 中用”Code”数据类型"Code"类型用于在文档中存储 JavaScript 代码。10.MongoDB 支持哪些数据类型"StringIntegerDoubleBooleanObjectObjectIDArraysMin/MaxKeysDatetimeCodeRegularExpression 等"11.MongoDB 支持主键外键关系吗"默认 MongoDB 不支持主键和外键关系。用 Mongodb 本身的 API 需要硬编码才能实现外键关联，不够直观且难度较大。"12.为什么要在 MongoDB 中使用分析器 mongodb 中包括了一个可以显示数据库中每个操作性能特点的数据库分析器.通过这个分析器你可以找到比预期慢的查询(或写操作);利用这一信息,比如,可以确定是否需要添加索引。13.MongoDB 中的分片什么意思分片是将数据水平切分到不同的物理节点。当应用数据越来越大的时候，数据量也会越来越大。当数据量增长时，单台机器有可能无法存储数据或可接受的读取写入吞吐量。利用分片技术可以添加更多的机器来应对数据量增加以及读写操作的要求。14.MongoDB 中的命名空间是什么意思?"mongodb 存储 bson 对象在丛集(collection)中，数据库名字和丛集名字以句点连结起来叫做名字空间(namespace)。一个集合命名空间又有多个数据域(extent)，集合命名空间里存储着集合的元数据，比如集合名称，集合的第一个数据域和最后一个数据域的位置等等。而一个数据域由若干条文档(document)组成，每个数据域都有一个头部，记录着第一条文档和最后一条文档的为知，以及该数据域的一些元数据。extent 之间，document 之间通过双向链表连接。索引的存储数据结构是 B 树，索引命名空间存储着对 B 树的根节点的指针。"15.为什么用 MOngoDB？"架构简单没有复杂的连接深度查询能力,MongoDB 支持动态查询。容易调试容易扩展不需要转化/映射应用对象到数据库对象使用内部内存作为存储工作区,以便更快的存取数据。"16.在哪些场景使用 MongoDB"大数据内容管理系统移动端 Apps 数据管理"17.什么是文档(记录)文档由一组 keyvalue 组成。文档是动态模式,这意味着同一集合里的文档不需要有相同的字段和结构。在关系型数据库中 table 中的每一条记录相当于 MongoDB 中的一个文档。18.什么是集

合(表)？"集合就是一组 MongoDB 文档。它相当于关系型数据库（RDBMS）中的表这种概念。集合位于单独的一个数据库中。一个集合内的多个文档可以有多个不同的字段。一般来说，集合中的文档都有着相同或相关的目的。"19.MongoDB 的优势有哪些"面向文档的存储：以 JSON 格式的文档保存数据。任何属性都可以建立索引。复制以及高可扩展性。自动分片。丰富的查询功能。快速的即时更新。"20.什么是 MongoDB？"MongoDB 是一个文档数据库，提供好的性能，领先的非关系型数据库。采用 BSON 存储文档数据。BSON（）是一种类 json 的一种二进制形式的存储格式，简称 BinaryJSON 相对于 json 多了 date 类型和二进制数组。"三十二、Mycat1.你们项目中分片的实现方式是什么？在 rule.xml 中配置 PartitionByMod2.配置文件不会变多，配置的节点主机会变多？不会 3.Mycat 的在分库分表之后，它是怎么支持联表查询的？"使用好 ER 表善用全局表在 sql 上添加注解"4.Mycat 中全局 ID 方案有哪些？程序自定义全局 ID 的方案有哪些？"mycat 的全局 id 方案（1）本地文件方式 sequnceHandlerType=0 配置 sequence_conf.properties 使用 nextvalueforMYCATSEQ_XXX（2）数据库方式 sequnceHandlerType=1 配置 sequence_db_conf.properties 使用 nextvalueforMYCATSEQ_XXX 或者指定 autoIncrement（3）本地时间戳方式 ID=64 位二进制(42(毫秒)+5(机器 ID)+5(业务编码)+12(重复累加)sequnceHandlerType=2 配置 sequence_time_conf.properties 指定 autoIncrement 程序方式（1）Snowflake（2）UUID（3）Redis"5.进行库表拆分时，拆分规则怎么取舍？"1.不存在热点数据时，则使用连续分片 2.存在热点数据时，使用离散分片或者是综合分片 3.离散分片暂时迁移比较麻烦（但是 mycat 给出了数据迁移的脚本，虽然现在还是不是很完美），综合分片占用总机器数量多"6.mycat 分库可以分成 100 个库吗？我们目前项目组分的是 3 个库，我们说一般数据量大的话我们使用的是 mycat 中间件进行分片处理，如果更大的话，我们可以使用 oracle 数据库，如果更大的话可以使用 hadoop 或是云存储数据，不需要 mycat 作为工具手段。衡量的标准是项目有没有对应的硬件设备。如果没有，基本就是使用 mysql 因为搭建一套云环境或者大数据的环境基本都是超大型的公司。比如大数据中的所有的技术，例如 hbase 或者是一大堆的服务器一大堆的网络路由设备或是私有云。或者是一大堆的数据库运维实施人员都是成本 7.搭建 mycat 的核心配置文件有哪些？"schem.xml 配置参数：逻辑库，逻辑表，数据节点。节点主机 rule.xml：分片规则 server.xml：连接 mycat 的用户信息（账号和密码）这里是使用中间件做数据切分，感兴趣的小伙伴还可以了解一下 mysql 的分库分表高可用方案"8.在项目组中，切分后的库从哪里而来？"在开发中是基于原有库创建出来，并且原有库和切分后的库是数据表的设计是保持一致的。dm_order1,dm_order2,dm_order3 这些库是需要和 dm_order 的设计保持一致的！！！！ 附注：所以，切分后的库例如 dm_order1,dm_order2,dm_order3 这些都是有数据库维护团队创建出来的。"9.什么叫混合切分项目组中如果有水平切分，那项目组里的开发方式就叫混合切分。或者项目组里就是单纯的垂直切分。10.Mycat 是什么？Mycat 是基于 MySQL 的数据库中间件，目的是为了降低数据库的压力。三十三、Nginx1.请陈述 stub_status 和 sub_filter 指令的作用是什么？"（1）Stub_status 指令：该指令用于了解 Nginx 当前状态的当前状态，如当前的活动连接，接受和处理当前读/写/等待连接的总数；（2）Sub_filter 指令：它用于搜索和替换响应中的内容，并快速修复陈旧的数据"2.Nginx 常用配置？"Nginx 常用配置？worker_processes8;#工作进程个数 worker_connections65535;#每个工作进程能并发处理（发起）的最大连接数（包含所有连接数）error_log/data/logs/nginx/error.log;#错误日志打印地址 access_log/data/logs/nginx/access.log;#进入日志打印地址 log_formatmain'$remote_addr""$request""'$status$upstream_addr""$request_time"";#进入日志格式##如果未使用 fastcgi 功能的，可以无视 fastcgi_connect_timeout=300;#连接到后端 fastcgi 超时时间 fastcgi_send_timeout=300;#向 fastcgi 请求超时时间(这个指定值已经完成两次握手后向 fastcgi 传送请求的超时时间)fastcgi_rend_timeout=300;#接收 fastcgi 应答超时时间，同理也是 2 次握手后 fastcgi_buffer_size=64k;#读取 fastcgi 应答第一部分需要多大缓冲区，该值表示使用 1 个 64kb 的缓冲区读取应答第一部分(应答头),可以设置为 fastcgi_buffers 选项缓冲区大小 fastcgi_buffers464k;#指定本地需要多少和多大的缓冲区来缓冲 fastcgi 应答请求，假设一个 php 或 java 脚本所产生页面大小为 256kb,那么会为其分配 4 个 64kb 的缓冲来缓存 fastcgi_cacheTEST;#开启 fastcgi 缓存并为其指定为 TEST 名称，降低 cpu 负载,防止 502 错误发生 listen80;#监听端口 server_namerrc.test.jiedaibao.com;#允许域名 root/data/release/rrc/web;#项目根目录 indexindex.phpindex.htmlindex.htm;#访问根文件"3.Nginx 常用命令？"启动 nginx。停止 nginx-sstop 或 nginx-squit。重载配置./sbin/nginx-sreload(平滑重启)或 servicenginxreload。重载指定配置文件.nginx-c/usr/local/nginx/conf/nginx.conf。查看 nginx 版本 nginx-v。检查配置文件是否正确 nginx-t。显示帮助信息 nginx-h。"4.fastcgi 与 cgi 的区别？"1）cgiweb 服务器会根据请求的内容，然后会 fork 一个新进程来运行外部 c 程序（或 perl 脚本…），这个进程会把处理完的数据返回给 web 服务器，最后 web 服务器把内容发送给用户，刚才 fork 的进程也随之退出。如果下次用户还请求改动态脚本，那么 web 服务器又再次 fork 一个新进程，周而复始的进行。2）fastcgiweb 服务器收到一个请求时，他不会重新 fork 一个进程（因为这个进程在 web 服务器启动时就开启了，而且不会退出），web 服务器直接把内容传递给这个进程（进程间通信，但 fastcgi 使用了别的方式，tcp 方式通信），这个进程收到请求后进行处理，把结果返回给 web 服务器，最后自己接着等待下一个请求的到来，而不是退出。综上，差别在于是否重复 fork 进程，处理请求。"5.ngx_http_upstream_module 的作用是什么?ngx_http_upstream_module 用于定义可通过 fastcgi 传递、proxy 传递、uwsgi 传递、memcached 传递和 scgi 传递指令来引用的服务器组。6.在 Nginx 中如何在 URL 中保留双斜线?要在 URL 中保留双斜线，就必须使用 merge_slashes_off；语法:merge_slashes[on/off]；默认值:merge_slasheson；环境:http，server7.在 Nginx 中，如何使用未定义的服务器名称来阻止处理请求?"只需将请求删除的服务器就可以定义为：Server{listen80;server_name “ ”;return444;}这里，服务器名被保留为一个空字符串，它将在没有“主机”头字段的情况下匹配请求，而一个特殊的 Nginx 的非标准代码 444 被返回，从而终止连接。"8.请解释 Nginx 如何处理 HTTP 请求。Nginx 使用反应器模式。主事件循环等待操作系统发出准备事件的信号，这样数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。单个线程可以提供数万个并发连接。9.列举 Nginx 服务器的最佳用途。Nginx 服务器的最佳用法是在网络上部署动态 HTTP 内容，使用 SCGI、WSGI 应用程序服务器、用于脚本的 FastCGI 处理程序。它还可以作为负载均衡器。10.使用“反向代理服务器”的优点是什么?反向代理服务器可以隐藏源服务器的存在和特征。它充当互联网云和 web 服务器之间的中间层。这对于安全方面来说是很好的，特别是当您使用 web 托管服务时。11.Nginx 应用场景？"http 服务器。Nginx 是一个 http 服务可以独立提供 http 服务。可以做网页静态服务器。虚拟主机。可以实现在一台服务器虚拟出多个网站，例如个人网站使用的虚拟机。反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用 nginx 做反向代理。并且多台服务器可以平均分担负载，不会应为某台服务器负载高宕机而某台服务器闲置的情况。nginx 中也可以配置安全管理、比如可以使用 Nginx 搭建 API 接口网关,对每个接口服务进行拦截。"12.Nginx 的优缺点？"优点：占内存小，可实现高并发连接，处理响应快可实现 http 服务器、虚拟主机、方向代理、负载均衡 Nginx 配置简单可以不暴露正式的服务器 IP 地址缺点：动态处理差：nginx 处理静态文件好,耗费内存少，但是处理动态页面则很鸡肋，现在一般前端用 nginx 作为反向代理抗住压力，"13.Nginx 怎么处理请求的？"nginx 接收一个请求后，首先由 listen 和 server_name 指令匹配 server 模块，再匹配 server 模块里的 location，location 就是实际地址 server{#第一个 Server 区块开始，表示一个独立的虚拟主机站点 listen80;#提供服务的端口，默认 80server_namelocalhost;#提供服务的域名主机名 location/{#第一个 location 区块开始 roothtml;#站点的根目录，相当于 Nginx 的安装目录 indexindex.html;#默认的首页文件，多个用空格分开}}"14.为什么要用 Nginx？"跨平台、配置简单、方向代理、高并发连接：处理 2-3 万并发连接数，官方监测能支持 5 万并发，内存消耗小：开启 10 个 nginx 才占 150M 内存，nginx 处理静态文件好，耗费内存少，而且 Nginx 内置的健康检查功能：如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。使用 Nginx 的话还能：节省宽带：支持 GZIP 压缩，可以添加浏览器本地缓存稳定性高：宕机的概率非常小接收用户请求是异步的"15.请解释一下什么是 Nginx？"Nginx，是一个 Web 服务器和反向代理服务器，用于 HTTP、HTTPS、SMTP、POP3 和 IMAP 协议。目前使用的最多的 Web 服务器或者代理服务器，像淘宝、新浪、网易、迅雷等都在使用。Nginx 的主要功能如下：作为 httpserver(代替 Apache，对 PHP 需要 FastCGI 处理器支持)FastCGI：Nginx 本身不支持 PHP 等语言，但是它可以通过 FastCGI 来将请求扔给某些语言或框架处理。反向代理服务器实现负载均衡虚拟主机"三十四、RocketMQ1.任何一台 Broker 突然宕机了怎么办？Broker 主从架构以及多副本策略。Master 收到消息后会同步给 Slave，这样一条

消息就不止一份了，Master 宕机了还有 slave 中的消息可用，保证了 MQ 的可靠性和高可用性。而且 RocketMQ4.5.0 开始就支持了 Dlegder 模式，基于 raft 的，做到了真正意义的 HA。2.再说说 RocketMQ 是如何保证数据的高容错性的?"在不开启容错的情况下，轮询队列进行发送，如果失败了，重试的时候过滤失败的 Broker 如果开启了容错策略，会通过 RocketMQ 的预测机制来预测一个 Broker 是否可用如果上次失败的 Broker 可用那么还是会选择该 Broker 的队列如果上述情况失败，则随机选择一个进行发送在发送消息的时候会记录一下调用的时间与是否报错，根据该时间去预测 broker 的可用时间"3.高吞吐量下如何优化生产者和消费者的性能?"1）开发同一 group 下，多机部署，并行消费单个 Consumer 提高消费线程个数批量消费。消息批量拉取，业务逻辑批量处理。2）运维网卡调优 jvm 调优多线程与 cpu 调优 PageCache"4.如果让你来动手实现一个分布式消息中间件，整体架构你会如何设计实现?"需要考虑能快速扩容、天然支持集群持久化的姿势高可用性数据 0 丢失的考虑服务端部署简单、client 端使用简单"5.RocketMQ 在分布式事务支持这块机制的底层原理?"分布式系统中的事务可以使用 TCC（Try、Confirm、Cancel）、2pc 来解决分布式系统中的消息原子性 RocketMQ4.3+提供分布事务功能，通过 RocketMQ 事务消息能达到分布式事务的最终一致 RocketMQ 实现方式：HalfMessage：预处理消息，当 broker 收到此类消息后，会存储到 RMQ_SYS_TRANS_HALF_TOPIC 的消息消费队列中检查事务状态：Broker 会开启一个定时任务，消费 RMQ_SYS_TRANS_HALF_TOPIC 队列中的消息，每次执行任务会向消息发送者确认事务执行状态（提交、回滚、未知），如果是未知，Broker 会定时去回调在重新检查。超时：如果超过回查次数，默认回滚消息。也就是他并未真正进入 Topic 的 queue，而是用了临时 queue 来放所谓的 halfmessage，等提交事务后才会真正的将 halfmessage 转移到 topic 下的 queue。"6.rocketMQ 的消息堆积如何处理"首先要找到是什么原因导致的消息堆积，是 Producer 太多了，Consumer 太少了导致的还是说其他情况，总之先定位问题。然后看下消息消费速度是否正常，正常的话，可以通过上线更多 consumer 临时解决消息堆积问题。"7.RocketMQ 如何保证消息不丢失？"首先在如下三个部分都可能会出现丢失消息的情况：Producer 端 Broker 端 Consumer 端 1）Producer 端如何保证消息不丢失采取 send()同步发消息，发送结果是同步感知的。发送失败后可以重试，设置重试次数。默认 3 次。producer.setRetryTimesWhenSendFailed(10);集群部署，比如发送失败了的原因可能是当前 Broker 宕机了，重试的时候会发送到其他 Broker 上。2）Broker 端如何保证消息不丢失修改刷盘策略为同步刷盘。默认情况下是异步刷盘的。flushDiskType=SYNC_FLUSH 集群部署，主从模式，高可用。3）Consumer 端如何保证消息不丢失完全消费正常后在进行手动 ack 确认。"8.如何让 RocketMQ 保证消息的顺序消费？"首先多个 queue 只能保证单个 queue 里的顺序，queue 是典型的 FIFO，天然顺序。多个 queue 同时消费是无法绝对保证消息的有序性的。所以总结如下：同一 topic，同一个 QUEUE，发消息的时候一个线程去发送消息，消费的时候一个线程去消费一个 queue 里的消息。"9.消息重复消费如何解决？"影响消息正常发送和消费的重要原因是网络的不确定性。引起重复消费的原因 1）ACK 正常情况下在 consumer 真正消费完消息后应该发送 ack，通知 broker 该消息已正常消费，从 queue 中剔除当 ack 因为网络原因无法发送到 broker，broker 会认为词条消息没有被消费，此后会开启消息重投机制把消息再次投递到 consumer2）消费模式在 CLUSTERING 模式下，消息在 broker 中会保证相同 group 的 consumer 消费一次，但是针对不同 group 的 consumer 会推送多次解决方案 1）数据库表处理消息前，使用消息主键在表中带有约束的字段中 insert2）Map 单机时可以使用 mapConcurrentHashMap->putIfAbsentguavacache3）Redis 分布式锁搞起来。"10.RocketMQ 如何做负载均衡？"通过 Topic 在多 Broker 中分布式存储实现。1）producer 端发送端指定 messagequeue 发送消息到相应的 broker，来达到写入时的负载均衡：提升写入吞吐量，当多个 producer 同时向一个 broker 写入数据的时候，性能会下降消息分布在多 broker 中，为负载消费做准备默认策略是随机选择：producer 维护一个 index 每次取节点会自增 index 向所有 broker 个数取余自带容错策略其他实现：SelectMessageQueueByHashhash 的是传入的 argsSelectMessageQueueByRandomSelectMessageQueueByMachineRoom 没有实现也可以自定义实现 MessageQueueSelector 接口中的 select 方法 2）consumer 端采用的是平均分配算法来进行负载均衡。其他负载均衡算法平均分配策略(默认)(AllocateMessageQueueAveragely)环形分配策略(AllocateMessageQueueAveragelyByCircle)手动配置分配策略(AllocateMessageQueueByConfig)机房分配策略(AllocateMessageQueueByMachineRoom)一致性哈希分配策略(AllocateMessageQueueConsistentHash)靠近机房策略(AllocateMachineRoomNearby)"11.broker 如何处理拉取请求的？"Consumer 首次请求 BrokerBroker 中是否有符合条件的消息如果有响应 Consumer 等待下次 Consumer 的请求如果没有 DefaultMessageStore#ReputMessageService#run 方法 PullRequestHoldService 来 Hold 连接，每个 5s 执行一次检查 pullRequestTable 有没有消息，有的话立即推送每隔 1ms 检查 commitLog 中是否有新消息，有的话写入到 pullRequestTable 当有新消息的时候返回请求挂起 consumer 的请求，即不断开连接，也不返回数据使用 consumer 的 offset"12.消费消息是 push 还是 pull？"RocketMQ 没有真正意义的 push，都是 pull，虽然有 push 类，但实际底层实现采用的是长轮询机制，即拉取方式。broker 端属性 longPollingEnable 标记是否开启长轮询。默认开启。"13.RocketMQ 消费模式有几种？"集群消费一条消息只会被同 Group 中的一个 Consumer 消费多个 Group 同时消费一个 Topic 时，每个 Group 都会有一个 Consumer 消费到数据广播消费消息将对一个 ConsumerGroup 下的各个 Consumer 实例都消费一遍。即即使这些 Consumer 属于同一个 ConsumerGroup，消息也会被 ConsumerGroup 中的每个 Consumer 都消费一次。"14.RocketMQBroker 中的消息被消费后会立即删除吗？不会，每条消息都会持久化到 CommitLog 中，每个 Consumer 连接到 Broker 后会维持消费进度信息，当有消息消费后只是当前 Consumer 的消费进度（CommitLog 的 offset）更新了。15.RocketMQ 中的 Topic 和 JMS 的 queue 有什么区别？queue 就是来源于数据结构的 FIFO 队列。而 Topic 是个抽象的概念，每个 Topic 底层对应 N 个 queue，而数据也真实存在 queue 上的。三十五、Servlet1.init(ServletConfig)方法与异常该方法在执行过程中可以抛出 ServletException 来通知 Web 服务器 Servlet 实例初始化失败。一旦 ServletException 抛出，Web 服务器不会将客户端请求交给该 Servlet 实例来处理，而是报告初始化失败异常信息给客户端，该 Servlet 实例将被从内存中销毁。如果在来新的请求，Web 服务器会创建新的 Servlet 实例，并执行新实例的初始化操作 2.init(ServletConfig)方法执行次数该方法执行在单线程的环境下，因此开发者不用考虑线程安全的问题。3.如何读取 Servlet 的初始化参数？"ServletConfig 中定义了如下的方法用来读取初始化参数的信息：publicStringgetInitParameter(Stringname)参数：初始化参数的名称。返回：初始化参数的值，如果没有配置，返回 null。"4.如何配置 Servlet 的初始化参数？"在 web.xml 中该 Servlet 的定义标记中，比如：<servlet><servlet-name>TimeServlet</servlet-name><servlet-class>com.allanlxf.servlet.basic.TimeServlet</servlet-class><init-param><param-name>user</param-name><param-value>username</param-value></init-param><init-param><param-name>blog</param-name><param-value>http://。。。</param-value></init-param></servlet>配置了两个初始化参数 user 和 blog 它们的值分别为 username 和 http://。。。，这样以后要修改用户名和博客的地址不需要修改 Servlet 代码，只需修改配置文件即可。"5.Request 对象的主要方法"setAttribute(Stringname,Object)：设置名字为 name 的 request 的参数值 getAttribute(Stringname)：返回由 name 指定的属性值 getAttributeNames()：返回 request 对象所有属性的名字集合，结果是一个枚举的实例 getCookies()：返回客户端的所有 Cookie 对象，结果是一个 Cookie 数组 getCharacterEncoding()：返回请求中的字符编码方式 getContentLength()：返回请求的 Body 的长度 getHeader(Stringname)：获得 HTTP 协议定义的文件头信息 getHeaders(Stringname)：返回指定名字的 requestHeader 的所有值，结果是一个枚举的实例 getHeaderNames()：返回所以 requestHeader 的名字，结果是一个枚举的实例 getInputStream()：返回请求的输入流，用于获得请求中的数据 getMethod()：获得客户端向服务器端传送数据的方法 getParameter(Stringname)：获得客户端传送给服务器端的有 name 指定的参数值 getParameterNames()：获得客户端传送给服务器端的所有参数的名字，结果是一个枚举的实例 getParameterValues(Stringname)：获得有 name 指定的参数的所有值 getProtocol()：获取客户端向服务器端传送数据所依据的协议名称 getQueryString()：获得查询字符串 getRequestURI()：获取发出请求字符串的客户端地址 getRemoteAddr()：获取客户端的 IP 地址 getRemoteHost()：获取客户端的名字 getSession([Booleancreate])：返回和请求相关 SessiongetServerName()：获取服务器的名字 getServletPath()：获取客户端所请求的脚本文件的路径 getServerPort()：获取服务器的端口号 removeAttribute(Stringname)：删除请求中的一个属性"6.四种会话跟踪技术"会话作用域 ServletsJSP 页面描述 1）page 否是代表与一个页面相

关的对象和属性。一个页面由一个编译好的 Javaservlet 类（可以带有任何的 include 指令，但是没有 include 动作）表示。这既包括 servlet 又包括被编译成 servlet 的 JSP 页面 2）request 是是代表与 Web 客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个 Web 组件（由于 forward 指令和 include 动作的关系）3）session 是是代表与用于某个 Web 客户机的一个用户体验相关的对象和属性。一个 Web 会话可以也经常会跨越多个客户机请求 4）application 是是代表与整个 Web 应用程序相关的对象和属性。这实质上是跨越整个 Web 应用程序，包括多个页面、请求和会话的一个全局作用域"7.页面间对象传递的方法 equest，session，application，cookie 等 8.什么情况下调用 doGet()和 doPost()？JSP 页面中的 form 标签里的 method 属性为 get 时调用 doGet()，为 post 时调用 doPost()；超链接跳转页面时调用 doGet()9.Servlet 的基本架构

"publicclassServletNameextendsHttpServlet{publicvoiddoPost(HttpServletRequestrequest,HttpServletResponseresponse)throwsServletException,IOException{}publicvoiddoGet(HttpServletRequestrequest,HttpServletResponseresponse)throwsServletException,IOException{}}"10.Servlet 和 JSP 的区别？Servlet 是服务器端的程序，动态生成 html 页面发送到客户端，但是这样程序里会有很多 out.println(),java 与 html 语言混在一起很乱，造成编写逻辑控制的后台工程师和设计前端网页的前端工程师彼此很难独立开展工作，所以后来 sun 公司推出了 JSP，其实 JSP 就是 Servlet，每次运行的时候 JSP 都首先被编译成 servlet 文件，然后再被编译成.class 文件运行。有了 jsp，在 MVC 项目中 servlet 不再负责动态生成页面，转而去负责控制程序逻辑的作用，控制 jsp 与 javabean 之间的流转。其实对 jsp 也有封装的模板工具 velocity 和 freemarker。11.Servlet 的生命周期？"—根据 Servlet 的配置参数 1 来决定实例化时机，没有配置该参数项或者为负数，则第一次访问的时候才会被实例化并调用 init()函数，如果为 0 或者正整数，则服务器启动的时候就会被加载，加载顺序由小到达。Servlet 通过调用 init()方法进行初始化。—客户端请求到达后，Servlet 调用 service()方法来处理客户端的请求。—服务器关闭，或者 Servlet 长时间没有使用，Servlet 通过调用 destroy()方法终止（结束）。—最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。"三十六、Shiro1.SessionManager 会话管理介绍一下"Session 所谓 session，即用户访问应用时保持的连接关系，在多次交互中应用能够识别出当前访问的用户是谁，且可以在多次交互中保存一些数据。Subjectsubject=SecurityUtils.getSubject();Sessionsession=subject.getSession();session.getId();//获取当前 session 的唯一标识 session.getHost();//获取当前 Subject 的主机地址，该地址是通过 HostAuthenticationToken.getHost()提供的 session.getTimeOut();//获取超时时间 session.setTimeOut();//设置超时时间（不设置默认是全局过期时间）session.touch();//更新最后访问时间 session.stop();//销毁 session，当 Subject.logout()时会自动调用 stop 方法来销毁会话。如果在 web 中，调用 javax.servlet.http.HttpSession.invalidate()也会自动调用 shirosession.top 方法进行销毁 shiro 的会话 session.setAttribute（“key”,”123”);//设置 session 属性 session.getAttribute（“key”);//获取 session 属性 session.removeAttribute（“key”);//删除属性注：Shiro 提供的会话可以用于 javaSE/javaEE 环境，不依赖于任何底层容器，可以独立使用，是完整的会话模块。Sessionmanager 会话管理器会话管理器管理着应用中所有 Subject 的会话的创建、维护、删除、失效、验证等工作。是 Shiro 的核心组件，顶层组件 SecurityManager 直接继承了 SessionManager，且提供了 SessionSecurityManager 实现直接把会话管理委托给相应的 SessionManager、DefaultSecurityManager 及 DefaultWebSecurityManager 默认 SecurityManager 都继承了 SessionSecurityManager。Shiro 提供了三个默认实现：DefaultSessionManager：DefaultSecurityManager 使用的默认实现，用于 JavaSE 环境；ServletContainerSessionManager:DefaultWebSecurityManager 使用的默认实现，用于 Web 环境，其直接使用 Servlet 容器的会话；DefaultWebSessionManager：用于 Web 环境的实现，可以替代 ServletContainerSessionManager，自己维护着会话，直接废弃了 Servlet 容器的会话管理。"2.shiro 拦截器的执行流程"基于表单登录拦截器 onPreHandle 主要流程：1）首先判断是否已经登录过了，如果已经登录过了继续拦截器链即可；2）如果没有登录，看看是否是登录请求，如果是 get 方法的登录页面请求，则继续拦截器链（到请求页面），否则如果是 get 方法的其他页面请求则保存当前请求并重定向到登录页面；3）如果是 post 方法的登录页面表单提交请求，则收集用户名/密码登录即可，如果失败了保存错误消息到“shiroLoginFailure”并返回到登录页面；4）如果登录成功了，且之前有保存的请求，则重定向到之前的这个请求，否则到默认的成功页面。任意角色授权拦截器流程：1）首先判断用户有没有任意角色，如果没有返回 false，将到 onAccessDenied 进行处理；2）如果用户没有角色，接着判断用户有没有登录，如果没有登录先重定向到登录；3）如果用户没有角色且设置了未授权页面（unauthorizedUrl），那么重定向到未授权页面；否则直接返回 401 未授权错误码。默认拦截器身份验证相关的 authc 基于表单的拦截器，即验证成功之后才能访问 /=authcauthcBasicBasicHTTP 身份验证拦截器，主要属性：applicationNamelogout 退出/logout=logoutuser 用户拦截器/=useranon 匿名拦截器，一般用于静态资源过滤/static/=anon 授权相关的 roles 角色授权拦截器，主要属性：loginUrl，unauthorizedUrl/admin/=roles[admin]perms 权限授权拦截器/user/=perms[“user:create”]port 端口拦截器，主要属性:port(80)/test=port[80]restrest 风格拦截器 /users=rest[user]，会自动拼接出“user:read,user:create,user:update,user:delete”sslssl 拦截器，只有请求协议是 https 才能通过"3.Realm 域如何使用？"定义 Realm（自定义 Realm 继承 AuthorizingRealm 即可）1）UserRealm 父类 AuthorizingRealm 将获取 Subject 相关信息分成两步：获取身份验证信息（doGetAuthenticationInfo）及授权信息（doGetAuthorizationInfo）2）doGetAuthenticationInfo 获取身份验证相关信息：首先根据传入的用户名获取 User 信息；如果 user 为空，那么抛出没找到账号异常 UnknownAccountExecption；如果 user 找到但却被锁定了抛出锁定异常 LockedAccountException；最后生成 AuthenticationInfo 信息，交给间接父类 AuthenticatingRealm 使用 CredentialsMatcher 进行判断密码是否匹配，如果不匹配将抛出密码错误异常信息 IncorrectCredentialsException；如果密码重试次数太多将抛出超出重试次数异常 ExcessiveAttemptsException；在组装 SimpleAuthenticationInfo 信息时，需要传入：身份信息（用户名）、凭据（密文密码）、盐（username+salt），CredentialsMatcher 使用盐加密传入的明文密码和此处的密文密码进行匹配。3）doGetAuthorizationInfo 获取授权信息：PrincipalCollection 是一个身份集合，因为只用到了一个 Realm，所以直接调用 getPrimaryPrincipal 得到之前传入的用户名即可；然后根据用户名调用 UserService 接口获取角色及权限信息。AuthenticationInfo 的两个作用 1）如果 Realm 是 AuthenticatingRealm 子类，则提供给 AuthenticatingRealm 内部使用的 CredentialsMatcher 进行凭据验证；（如果没有继承它需要在自己的 Realm 中实现验证）；2）提供给 SecurityManager 来创建 Subject（提供身份信息）；"4.Cryptography 加密的过程是这样的？"编码/解码 Shiro 提供了 base64 和 16 进制字符串编码/解码的 API 支持,方便一些编码解码操作 Base64.encodeToString(str.getBytes())编码 Base64.decodeToString(base64Encoded)解码散列算法常见散列算法如 MD5,SHA 等 1）首先创建一个 DfaultHashService,默认使用 SHA-512 算法；2）可以通过 hashAlgorithmName 属性修改算法；3）可以通过 privateSalt 设置一个私盐，其在散列时自动与用户传入的公盐混合产生一个新盐；4）可以通过 generatePublicSalt 属性在用户没有传入公盐的情况下是否生成公盐；5）可以设置 randomNumberGenerator 用于生成公盐；6）可以设置 hashIterations 属性来修改默认加密迭代次数；7）需要构建一个 HashRequest,传入算法、数据、公盐、迭代次数。生成随机数

SecureRandomNumberGeneratorrandomNumberGenerator=newSecureRandomNumberGenerator();randomNumberGenerator.setSeed(“159”.getBytes());Stringhex=randomNumberGenerator.nextBytes().toHex();加密/解密提供对称式加密/解密算法的支持，如 AES、Blowfish 等 PasswordService/CredentialsMatcher 用于提供加密密码及验证密码服务 Shiro 默认提供了 PasswordService 实现 DefaultPasswordService;CredentialsMatcher 实现 PasswordMatcher 及 HashedCredentialsMatcher(更强大)HashedCredentialsMatcher 实现密码验证服务 Shiro 提供了 CredentialsMatcher 的散列实现 HashedCredentialsMatcher,和 PasswordMatcher 不同的是，它只是用于密码验证，且可以提供自己的盐，而不是随机生成盐，且生成密码散列值的算法需要自己写，因为能提供自己的盐"5.Authorization 授权的方式和流程是怎样的？"principals：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。credentials：证明/凭证，即只有主体知道的安全值，如密码/数字证书等。身份认证流程：1）首先调用

Subject.login(token)进行登录，其会自动委托给 SecurityManager，调用之前必须通过 SecurityUtils.setSecurityManager()设置；2）SecurityManager 负责真正的身份验证逻辑；它会委托给 Authenticator 进行身份验证；3）Authenticator 才是真正的身份验证者，shiroapi 中核心的身份认证入口点，此处可以自定义插入自己的实现；4）Authenticator 可能会委托给相应的 AuthenticationStrategy 进行多 Realm 身份验证，默认 ModularRealmAuthenticator 会调用 AuthenticationStrategy 进行多 Realm 身份验证；5）Authenticator 会把相应的 token 传入 Realm，从 Realm 获取身份验证信息，如果没有返回/抛出异常表示身份验证失败了。此处可以配置多个 Realm，将按照相应的顺序及策略进行访问。6）Authenticator 的职责是验证用户账号，是 shiroapi 中身份验证核心的入口点。7）AuthenticationStrategy 认证策略 ModularRealmAuthenticator 默认使用 AtLeastOneSuccessfulStrategy 策略 1>FirstSuccessfulStrategy：只要有一个 Realm 验证成功即可，只返回第一个 Realm 身份验证成功的认证信息，其他的忽略；2>AtLeastOneSuccessfulStrategy：只要有一个 Realm 验证成功即可，和 FirstSuccessfulStrategy 不同，返回所有 Realm 身份验证成功的认证信息；3>AllSuccessfulStrategy：所有 Realm 验证成功才算成功，且返回所有 Realm 身份验证成功的认证信息，如果有一个失败就失败了。"6.说下 Authentication 身份验证的流程"principals：身份，即主体的标识属性，可以是任何东西，如用户名、邮箱等，唯一即可。credentials：证明/凭证，即只有主体知道的安全值，如密码/数字证书等。身份认证流程：1）首先调用 Subject.login(token)进行登录，其会自动委托给 SecurityManager，调用之前必须通过 SecurityUtils.setSecurityManager()设置；2）SecurityManager 负责真正的身份验证逻辑；它会委托给 Authenticator 进行身份验证；3）Authenticator 才是真正的身份验证者，shiroapi 中核心的身份认证入口点，此处可以自定义插入自己的实现；4）Authenticator 可能会委托给相应的 AuthenticationStrategy 进行多 Realm 身份验证，默认 ModularRealmAuthenticator 会调用 AuthenticationStrategy 进行多 Realm 身份验证；5）Authenticator 会把相应的 token 传入 Realm，从 Realm 获取身份验证信息，如果没有返回/抛出异常表示身份验证失败了。此处可以配置多个 Realm，将按照相应的顺序及策略进行访问。6）Authenticator 的职责是验证用户账号，是 shiroapi 中身份验证核心的入口点。7）AuthenticationStrategy 认证策略 ModularRealmAuthenticator 默认使用 AtLeastOneSuccessfulStrategy 策略 1>FirstSuccessfulStrategy：只要有一个 Realm 验证成功即可，只返回第一个 Realm 身份验证成功的认证信息，其他的忽略；2>AtLeastOneSuccessfulStrategy：只要有一个 Realm 验证成功即可，和 FirstSuccessfulStrategy 不同，返回所有 Realm 身份验证成功的认证信息；3>AllSuccessfulStrategy：所有 Realm 验证成功才算成功，且返回所有 Realm 身份验证成功的认证信息，如果有一个失败就失败了。"7.Shiro 有哪些组件？"Authentication：身份认证/登录，验证用户是不是拥有相应的身份；Authorization：授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限；SessionManager：会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中；会话可以是普通 JavaSE 环境的，也可以是如 Web 环境的；Cryptography：加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储；WebSupport：Web 支持，可以非常容易的集成到 Web 环境；Caching：缓存，比如用户登录后，其用户信息、拥有的角色/权限不必每次去查，这样可以提高效率；Concurrency：shiro 支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去；Testing：提供测试支持；RunAs：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问；RememberMe：记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用登录了。记住一点，Shiro 不会去维护用户、维护权限；这些需要我们自己去设计/提供；然后通过相应的接口注入给 Shiro 即可。"8.Shiro 的优点"1、简单的身份验证，支持多种数据源 2、对角色的简单授权，支持细粒度的授权（方法）3、支持一级缓存，以提升应用程序的性能 4、内置基于 POJO 的企业会话管理，适用于 web 及非 web 环境 5、非常简单的 API 加密 6、不跟任何框架绑定，可以独立运行"9.解释下 Shiro 的核心概念：Subject、SecurityManager、Realm"Subject：主体，代表了当前“用户”，这个用户不一定是一个具体的人，与当前应用交互的任何东西都是 Subject，如爬虫、机器人等；即一个抽象概念；所有 Subject 都绑定到 SecurityManager，与 Subject 的所有交互都会委托给 SecurityManager；可以把 Subject 认为是一个门面；SecurityManager 才是实际的执行者。SecurityManager：安全管理器；即所有与安全有关的操作都会与 SecurityManager 交互；且它管理着所有 Subject；可以看出它是 shiro 的核心,SecurityManager 相当于 springmvc 中的 dispatcherServlet 前端控制器。Realm：域，shiro 从 Realm 获取安全数据(如用户、角色、权限)，就是说 SecurityManager 要验证用户身份，那么它需要从 Realm 获取相应的用户进行比较以确定用户身份是否合法；也需要从 Realm 得到用户相应的角色/权限进行验证用户是否能进行操作；可以把 Realm 看成 DataSource，即安全数据源。"10.什么是 shiroShiro 是一个强大易用的 java 安全框架，提供了认证、授权、加密、会话管理、与 web 集成、缓存等功能，对于任何一个应用程序，都可以提供全面的安全服务，相比其他安全框架，shiro 要简单的多。三十七、Tomcat1.Tomcat 一个请求的完整过程"首先 dns 解析 wo.de.tian 机器，一般是 ng 服务器 ip 地址然后 ng 根据 server 的配置，寻找路径为 yy/的机器列表，ip 和端口最后选择其中一台机器进行访问—->下面为详细过程 1)请求被发送到本机端口 8080，被在那里侦听的 CoyoteHTTP/1.1Connector 获得 2)Connector 把该请求交给它所在的 Service 的 Engine 来处理，并等待来自 Engine 的回应 3)Engine 获得请求 localhost/yy/index.jsp，匹配它所拥有的所有虚拟主机 Host4)Engine 匹配到名为 localhost 的 Host（即使匹配不到也把请求交给该 Host 处理，因为该 Host 被定义为该 Engine 的默认主机）5)localhostHost 获得请求/yy/index.jsp，匹配它所拥有的所有 Context6)Host 匹配到路径为/yy 的 Context（如果匹配不到就把该请求交给路径名为”“的 Context 去处理）7)path=”/yy”的 Context 获得请求/index.jsp，在它的 mappingtable 中寻找对应的 servlet8)Context 匹配到 URLPATTERN 为*.jsp 的 servlet，对应于 JspServlet 类 9)构造 HttpServletRequest 对象和 HttpServletResponse 对象，作为参数调用 JspServlet 的 doGet 或 doPost 方法 Context 把执行完了之后的 HttpServletResponse 对象返回给 HostHost 把 HttpServletResponse 对象返回给 EngineEngine 把 HttpServletResponse 对象返回给 ConnectorConnector 把 HttpServletResponse 对象返回给客户 browser"2.如何添加 JMS 远程监控"对于部署在局域网内其它机器上的 Tomcat，可以打开 JMX 监控端口，局域网其它机器就可以通过这个端口查看一些常用的参数（但一些比较复杂的功能不支持），同样是在 JVM 启动参数中配置即可，配置如下：-Dcom.sun.management.jmxremote.ssl=false-Dcom.sun.management.jmxremote.authenticate=false-Djava.rmi.server.hostname=192.168.71.38 设置 JVM 的 JMS 监控监听的 IP 地址，主要是为了防止错误的监听成 127.0.0.1 这个内网地址-Dcom.sun.management.jmxremote.port=1090 设置 JVM 的 JMS 监控的端口-Dcom.sun.management.jmxremote.ssl=false 设置 JVM 的 JMS 监控不实用 SSL-Dcom.sun.management.jmxremote.authenticate=false 设置 JVM 的 JMS 监控不需要认证"3.tomcat 共享 session 如何处理？"目前的处理方式有如下几种：1).使用 Tomcat 本身的 Session 复制功能参考 http://ajita.iteye.com/blog/1715312（Session 复制的配置）方案的有点是配置简单，缺点是当集群数量较多时，Session 复制的时间会比较长，影响响应的效率 2).使用第三方来存放共享 Session 目前用的较多的是使用 memcached 来管理共享 Session，借助于 memcached-sessonmanager 来进行 Tomcat 的 Session 管理参考 http://ajita.iteye.com/blog/1716320（使用 MSM 管理 Tomcat 集群 session）3).使用黏性 session 的策略对于会话要求不太强（不涉及到计费，失败了允许重新请求下等）的场合，同一个用户的 session 可以由 nginx 或者 apache 交给同一个 Tomcat 来处理，这就是所谓的 sessionsticky 策略，目前应用也比较多参考：http://ajita.iteye.com/blog/1848665（tomcatsessionsticky）nginx 默认不包含 sessionsticky 模块，需要重新编译才行（windows 下我也不知道怎么重新编译）优点是处理效率高多了，缺点是强会话要求的场合不合适"4.tomcat 垃圾回收策略调优了解吗？"垃圾回收的设置也是在 catalina.sh 中，调整 JAVA_OPTS 变量。具体设置如下：JAVA_OPTS=""$JAVA_OPTS-Xmx3550m-Xms3550m-Xss128k-XX:+UseParallelGCXX:MaxGCPauseMillis=100""具体的垃圾回收策略及相应策略的各项参数如下：串行收集器（JDK1.5 以前主要的回收方式）-XX:+UseSerialGC:设置串行收集器并行收集器（吞吐量优先）示例：java-Xmx3550m-Xms3550m-Xmn2g-Xss128k-XX:+UseParallelGCXX:MaxGCPauseMillis=100-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。-XX:+UseParallelOldGC：配置年老代垃圾收集方式为

并行收集。JDK6.0 支持对年老代并行收集-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM 会自动调整年轻代大小，以满足此值。-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的 Survivor 区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。并发收集器（响应时间优先）示例：java-Xmx3550m-Xms3550m-Xmn2g-Xss128k–XX:+UseConcMarkSweepGC-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，XX:NewRatio=4 的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn 设置。-XX:+UseParNewGC:设置年轻代为并行收集。可与 CMS 收集同时使用。JDK5.0 以上，JVM 会根据系统配置自行设置，所以无需再设置此值。-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片"5.tomcat 内存调优了解过吗？"内存方式的设置是在 catalina.sh 中，调整一下 JAVA_OPTS 变量即可，因为后面的启动参数会把 JAVA_OPTS 作为 JVM 的启动参数来处理。具体设置如下：JAVA_OPTS=""$JAVA_OPTS-Xmx3550m-Xms3550m-Xss128k-XX:NewRatio=4-XX:SurvivorRatio=4""其各项参数如下：-Xmx3550m：设置 JVM 最大可用内存为 3550M。-Xms3550m：设置 JVM 促使内存为 3550m。此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。-Xmn2g：设置年轻代大小为 2G。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。-Xss128k：设置每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为 1M，以前每个线程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4，年轻代占整个堆栈的 1/5-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6-XX:MaxPermSize=16m:设置持久代大小为 16m。-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。"6.tomcat 如何优化？"1、优化连接配置.这里以 tomcat7 的参数配置为例，需要修改 conf/server.xml 文件，修改连接数，关闭客户端 dns 查询。参数解释：URIEncoding=”UTF-8”:使得 tomcat 可以解析含有中文名的文件的 url，真方便，不像 apache 里还有搞个 mod_encoding，还要手工编译 maxSpareThreads:如果空闲状态的线程数多于设置的数目，则将这些线程中止，减少这个池中的线程总数。minSpareThreads:最小备用线程数，tomcat 启动时的初始化的线程数。enableLookups:这个功效和 Apache 中的 HostnameLookups 一样，设为关闭。connectionTimeout:connectionTimeout 为网络连接超时时间毫秒数。maxThreads:maxThreadsTomcat 使用线程来处理接收的每个请求。这个值表示 Tomcat 可创建的最大的线程数，即最大并发数。acceptCount:acceptCount 是当线程数达到 maxThreads 后，后续请求会被放入一个等待队列，这个 acceptCount 是这个队列的大小，如果这个队列也满了，就直接 refuseconnectionmaxProcessors 与 minProcessors:在 Java 中线程是程序运行时的路径，是在一个程序中与其它控制线程无关的、能够独立运行的代码段。它们共享相同的地址空间。多线程帮助程序员写出 CPU 最大利用率的高效程序，使空闲时间保持最低，从而接受更多的请求。通常 Windows 是 1000 个左右，Linux 是 2000 个左右。useURIValidationHack://我们来看一下 tomcat 中的一段源码：

if(connector.getUseURIValidationHack()){Stringuri=validate(request.getRequestURI());if(uri==null){res.setStatus(400);res.setMessage(“InvalidURI”);thrownewIOException(“InvalidURI”);}else{req.requestURI().setString(uri);//RedoingtheURIdecodingreq.decodedURI().duplicate(req.requestURI());req.getURLDecoder().convert(req.decodedURI(),true);}COPY 可以看到如果把 useURIValidationHack 设成”false”，可以减少它对一些 url 的不必要的检查从而减省开销。enableLookups=”false”：为了消除 DNS 查询对性能的影响我们可以关闭 DNS 查询，方式是修改 server.xml 文件中的 enableLookups 参数值。disableUploadTimeout：类似于 Apache 中的 keeyalive 一样给 Tomcat 配置 gzip 压缩(HTTP 压缩)功能

compression=”on”compressionMinSize=”2048”compressableMimeType=”text/html,text/xml,text/JavaScript,text/css,text/plain”HTTP 压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩并浏览。相对于普通的浏览过程 HTML,CSS,javascript,Text，它可以节省 40%左右的流量。更为重要的是，它可以对动态生成的，包括 CGI、PHP,JSP,ASP,Servlet,SHTML 等输出的网页也能进行压缩，压缩效率惊人。1)compression=”on”打开压缩功能 2)compressionMinSize=”2048”启用压缩的输出内容大小，这里面默认为 2KB3)noCompressionUserAgents=”gozilla,traviata”对于以下的浏览器，不启用压缩 4)compressableMimeType=”text/html,text/xml”压缩类型最后不要忘了把 8443 端口的地方也加上同样的配置，因为如果我们走 https 协议的话，我们将会用到 8443 端口这个段的配置，对吧？<!–enabletomcatssl–><Connectorport=”8443”protocol=”HTTP/1.1”URIEncoding=”UTF-8”minSpareThreads=”25”maxSpareThreads=”75”enableLookups=”false”disableUploadTimeout=”true”connectionTimeout=”20000”acceptCount=”300”maxThreads=”300”maxProcessors=”1000”minProcessors=”5”useURIValidationHack=”false”compression=”on”compressionMinSize=”2048”compressableMimeType=”text/html,text/xml,text/javascript,text/css,text/plain”SSLEnabled=”true”scheme=”https”secure=”true”clientAuth=”false”sslProtocol=”TLS”keystoreFile=”d:/tomcat2/conf/shnlap93.jks”keystorePass=”aaaaaa”/>COPY 好了，所有的 Tomcat 优化的地方都加上了。"7.tomcat 容器是如何创建 servlet 类实例？用到了什么原理？"当容器启动时，会读取在 webapps 目录下所有的 web 应用中的 web.xml 文件，然后对 xml 文件进行解析，并读取 servlet 注册信息。然后，将每个应用中注册的 servlet 类都进行加载，并通过反射的方式实例化。（有时候也是在第一次请求时实例化）在 servlet 注册时加上如果为正数，则在一开始就实例化，如果不写或为负数，则第一次请求实例化。"8.Tomcat 有几种部署方式？"1）直接把 Web 项目放在 webapps 下，Tomcat 会自动将其部署 2）在 server.xml 文件上配置 Context 节点，设置相关的属性即可 3）通过 Catalina 来进行配置:进入到 conf\Catalina\localhost 文件下，创建一个 xml 文件，该文件的名字就是站点的名字。编写 XML 的方式来进行设置。"9.tomcat 有哪几种 Connector 运行模式(优化)？"bio：传统的 Java I/O 操作，同步且阻塞 IO。

maxThreads=""150""//Tomcat 使用线程来处理接收的每个请求。这个值表示 Tomcat 可创建的最大的线程数。默认值 200。可以根据机器的时期性能和内存大小调整，一般可以在 400-500。最大可以在 800 左右。minSpareThreads=""25""—Tomcat 初始化时创建的线程数。默认值 4。如果当前没有空闲线程，且没有超过 maxThreads，一次性创建的空闲线程数量。Tomcat 初始化时创建的线程数量也由此值设置。maxSpareThreads=""75""—一旦创建的线程超过这个值，Tomcat 就会关闭不再需要的 socket 线程。默认值 50。一旦创建的线程超过此数值，Tomcat 会关闭不再需要的线程。线程数可以大致上用“同时在线人数每秒用户操作次数系统平均操作时间”来计算。acceptCount=""100""—指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。默认值 10。如果当前可用线程数为 0，则将请求放入处理队列中。这个值限定了请求队列的大小，超过这个数值的请求将不予处理。connectionTimeout=""20000""—网络连接超时，默认值 20000，单位：毫秒。设置为 0 表示永不超时，这样设置有隐患的。通常可设置为 30000 毫秒。nio：JDK1.4 开始支持，同步阻塞或同步非阻塞 IO。指定使用 NIO 模型来接受 HTTP 请求 protocol=""org.apache.coyote.http11.Http11NioProtocol""指定使用 NIO 模型来接受 HTTP 请求。默认是 BlockingIO，配置为 protocol=""HTTP/1.1""acceptorThreadCount=""2""使用 NIO 模型时接收线程的数目 aio(nio.2)：JDK7 开始支持，异步非阻塞 IO。apr：Tomcat 将以 JNI 的形式调用 ApacheHTTP 服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地提高 Tomcat 对静态文件的处理性能。<!--

<ConnectorconnectionTimeout=""20000""port=""8000""protocol=""HTTP/1.1""redirectPort=""8443""uriEncoding=""utf-8""/>--><!--protocol 启用 nio 模式，(tomcat8 默认使用的是 nio)(apr 模式利用系统级异步 io)--><!--minProcessors 最小空闲连接线程数--><!--maxProcessors 最大连接线程数--><!--acceptCount 允许的最大连接数，应大于等于 maxProcessors--><!--enableLookups 如果为 true,requst.getRemoteHost 会执行 DNS 查找，反向解析 ip 对应域名或主机名-->
<Connectorport=""8080""protocol=""org.apache.coyote.http11.Http11NioProtocol""connectionTimeout=""20000""redirectPort=""8443""maxThreads=""500""minSpareThreads=""100""maxSpareThreads=""200""acceptCount=""200""enableLookups=""false""/>COPY 其他配置 maxHttpHeaderSize=""8192""http 请求头信息的最大程度，超过此长度的部分不予处理。一般 8K。URIEncoding=""UTF-8""指定 Tomcat 容器的 URL 编码格式。disableUploadTimeout=""true""上传时是否使用超时机制 enableLookups=""false""–是否反查域名，默认值为 true。为了提高处理能力，应设置为 falsecompression=""on""打开压缩功能 compressionMinSize=""10240""启用压缩的输出内容大小，默认为 2KBnoCompressionUserAgents=""gozilla,traviata""对于以下的浏览器，不启用压缩 compressableMimeType=""text/html,text/xml,text/javascript,text/css,text/plain""哪些资源类型需要压缩"10.Tomcat 的缺省端口是多少，怎么修改？"1）找到 Tomcat 目录下的 conf 文件夹 2）进入 conf 文件夹里面找到 server.xml 文件 3）打开 server.xml 文件 4）在 server.xml 文件里面找到下列信息
<ConnectorconnectionTimeout=""20000""port=""8080""protocol=""HTTP/1.1""redirectPort=""8443""uriEncoding=""utf-8""/>port=""8080""改成你想要的端口"三十八、编译原理 1.全局优化过程内的全局优化是在一个程序过程（C 语言内称为函数）范围内进行的优化，前面提到的所有方法也都可用到全局优化当中。2.有穷自动机（有限自动机）"为什么是有穷？：状态和输入字母表有穷。作用：是一种识别装置，识别正规文法所定义的语言和正规式所表示的集合。分类：（i）确定的有穷自动机（DFA）（ii）不确定的有穷自动机（NFA）"3.出错处理出错处理：编译过程中，发现源程序有错误（词法错误、语法错误、语义错误），编译程序应报告错误的性质和出错的地点，并将错误所造成的影响限制在尽可能小的范围内，使得源程序的其余部分继续被编译下去。这些工作称为出错处理(errorhandling)。目的是使得编译程序能够继续向下进行分析和处理。4.表格管理程序编译过程中源程序的各种信息被保留在种种不同的表格里，编译各阶段的工作涉及到构造、查找或更新有关的表格，因此需要有表格管理工作。5.目标代码生成把中间代码变换成特定机器上的绝对指令代码或可重定位的指令代码或汇编指令代码,它的工作与硬件系统和指令含义有关.6.中间代码生成"中间代码是一种结构简单，含义明确的记号系统。将源程序生成一种内部表示形式，这种内部表示形式叫中间代码。（四元式就是一种中间代码形式）"7.语义分析"审查源程序有无语义错误，为代码生成阶段收集类型信息（如类型转化，类型匹配，上下文相关性等）。主要是类型相容检查，有以下几种：各种条件表达式的类型是不是 boolean 型？运算符的分量类型是否相容？赋值语句的左右部的类型是否相容？形参和实参的类型是否相容?下标表达式的类型是否为所允许的类型？函数说明中的函数类型和返回值的类型是否一致？V[E]中的 V 是不是变量，而且是数组类型？V.i 中的 V 是不是变量，而且是记录类型？i 是不是该记录的域名？x+f(…)中的 f 是不是函数名？形参个数和实参个数是否一致？每个使用性标识符是否都有声明？有无标识符的重复声明？在语义分析同时产生中间代码，在这种模式下，语义分析的主要功能如下：语义审查在扫描声明部分时构造标识符的符号表在扫描语句部分时产生中间代码"8.语法分析"语法分析是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等等.语法分析程序判断源程序在结构上是否正确在词法分析的基础上，将单词序列（输出）分解成各类语法短语。（赋值语句等语句）语法分析依据语言的语法规则，确定整个输入串是否在语法上正确。Ifelse 是否匹配，无论是 LL(1)文法分析，自顶向下还是自底向上都是要按照给定的语法，判断词法分析的词语是不是满足语法要求（标识符定义是否正确，括号是否匹配，关键字是否正确，标识符，函数定义是否正确，使用的函数或者标识符是否定义等等）"9.词法分析"即对构成源程序的字符流进行扫描然后根据构词规则识别单词(也称单词符号或符号)。所以当单词不符合构词规则时词法分析会报错。读入一个一个的字符（输入），对这些字符进行扫描和分解，从而识别出一个个单词（输出）。"三十九、操作系统 1.守护、僵尸、孤儿进程的概念"守护进程：运行在后台的一种特殊进程，独立于控制终端并周期性地执行某些任务。僵尸进程：一个进程 fork 子进程，子进程退出，而父进程没有 wait/waitpid 子进程，那么子进程的进程描述符仍保存在系统中，这样的进程称为僵尸进程。孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，这些子进程称为孤儿进程。（孤儿进程将由 init 进程收养并对它们完成状态收集工作）"2.系统调用与库函数的区别"系统调用(Systemcall)是程序向系统内核请求服务的方式。可以包括硬件相关的服务(例如，访问硬盘等)，或者创建新进程，调度其他进程等。系统调用是程序和操作系统之间的重要接口。库函数：把一些常用的函数编写完放到一个文件里，编写应用程序时调用，这是由第三方提供的，发生在用户地址空间。在移植性方面，不同操作系统的系统调用一般是不同的，移植性差；而在所有的 ANSIC 编译器版本中，C 库函数是相同的。在调用开销方面，系统调用需要在用户空间和内核环境间切换，开销较大；而库函数调用属于“过程调用”，开销较小。"3.内部碎片与外部碎片分别是什么？"在内存管理中，内部碎片是已经被分配出去的的内存空间大于请求所需的内存空间。外部碎片是指还没有分配出去，但是由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。固定分区存在内部碎片，可变式分区分配会存在外部碎片；页式虚拟存储系统存在内部碎片；段式虚拟存储系统，存在外部碎片为了有效的利用内存，使内存产生更少的碎片，要对内存分页，内存以页为单位来使用，最后一页往往装不满，于是形成了内部碎片。为了共享要分段，在段的换入换出时形成外部碎片，比如 5K 的段换出后，有一个 4k 的段进来放到原来 5k 的地方，于是形成 1k 的外部碎片。"4.用户态和核心态(内核态）之间的区别是什么呢？"权限不一样。用户态的进程能存取它们自己的指令和数据，但不能存取内核指令和数据（或其他进程的指令和数据）。核心态下的进程能够存取内核和用户地址某些机器指令是特权指令，在用户态下执行特权指令会引起错误。在系统中内核并不是作为一个与用户进程平行的估计的进程的集合。"5.用户态切换到内核态的方式有哪些？"系统调用：程序的执行一般是在用户态下执行的，但当程序需要使用操作系统提供的服务时，比如说打开某一设备、创建文件、读写文件（这些均属于系统调用）等，就需要向操作系统发出调用服务的请求，这就是系统调用。异常：当 CPU 在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。外围设备的中断：当外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。"6.中断与系统调用了解吗？"所谓的中断就是在计算机执行程序的过程中，由于出现了某些特殊事情，使得 CPU 暂停对程序的执行，转而去执行处理这一事件的程序。等这些特殊事情处理完之后再回去执行之前的程序。中断一般分为三类：由计算机硬件异常或故障引起的中断，称为内部异常中断；由程序中执行了引起中断的指令而造成的中断，称为软中断（这也是和我们将要说明的系统调用相关的中断）；由外部设备请求引起的中断，称为外部中断。简单来说，对中断的理解就是对一些特殊事情的处理。与中断紧密相连的一个概念就是中断处理程序了。当中断发生的时候，系统需要去对中断进行处理，对这些中断的处理是由操作系统内核中的特定函数进行的，这些处理中断的特定的函数就是我们所说的中断处理程序了。另一个与中断紧密相连的概念就是中断的优先级。中断的优先级说明的是当一个中断正在被处理的时候，处理器能接受的中断的级别。中断的优先级也表明了中断需要被处理的紧急程度。每个中断都有一个对应的优先级，当处理器在处理某一中断的时候，只有比这个中断优先级高的中断可以被处理器接受并且被处理。优先级比这个当前正在被处理的中断优先级要低的中断将会被忽略。典型的中断优先级如下所示：机器错误>时钟>磁盘>网络设备>终端>软件中断在讲系统调用之前，先说下进程的执行在系统上的两个级别：用户级和核心级，也称为用户态和系统态(usermodeandkernelmode)。用户空间就是用户进程所在的内存区域，相对的，系统空间就是操作系统占据的内存区域。用户进程和系统进程的所有数据都在内存中。处于用户态的程序只能访问用户空间，而处于内核态的程序可以访问用户空间和内核空间。"7.页面置换算法了解多少？"操作系统将内存按照页面进行管理，在需要的时候才把进程相应的部分调入内存。当产生缺页中断时，需要选择一个页面写入。如果要换出的页面在内存中被修改过，变成了“脏”页面，那就需要先写会到磁盘。页面置换算法，就是要选出最合适的一个页面，使得置换的效率最

高。页面置换算法有很多，简单介绍几个，重点介绍比较重要的 LRU 及其实现算法。一、最优页面置换算法最理想的状态下，我们给页面做个标记，挑选一个最远才会被再次用到的页面调出。当然，这样的算法不可能实现，因为不确定一个页面在何时会被用到。二、先进先出页面置换算法（FIFO）及其改进这种算法的思想和队列是一样的，该算法总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予淘汰。实现：把一个进程已调入内存的页面按先后次序链接成一个队列，并且设置一个指针总是指向最老的页面。缺点：对于有些经常被访问的页面如含有全局变量、常用函数、例程等的页面，不能保证这些不被淘汰。三、最近最少使用页面置换算法 LRU（LeastRecentlyUsed）根据页面调入内存后的使用情况做出决策。LRU 置换算法是选择最近最久未使用的页面进行淘汰。1.为每个在内存中的页面配置一个移位寄存器。(P165）定时信号将每隔一段时间将寄存器右移一位。最小数值的寄存器对应页面就是最久未使用页面。2.利用一个特殊的栈保存当前使用的各个页面的页面号。每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。因此，栈顶永远是最新被访问的页面号，栈底是最近最久未被访问的页面号。"8.说下对虚拟内存的理解"定义：具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充得一种存储器系统。其逻辑容量由内存之和和外存之和决定。与传统存储器比较虚拟存储器有以下三个主要特征：多次性，是指无需在作业运行时一次性地全部装入内存，而是允许被分成多次调入内存运行。对换性，是指无需在作业运行时一直常驻内存，而是允许在作业的运行过程中，进行换进和换出。虚拟性，是指从逻辑上扩充内存的容量，使用户所看到的内存容量，远大于实际的内存容量。虚拟内存的实现有以下两种方式：请求分页存储管理。请求分段存储管理。"9.动态链接库与静态链接库的区别"静态库静态库是一个外部函数与变量的集合体。静态库的文件内容，通常包含一堆程序员自定的变量与函数，其内容不像动态链接库那么复杂，在编译期间由编译器与链接器将它集成至应用程序内，并制作成目标文件以及可以独立运作的可执行文件。而这个可执行文件与编译可执行文件的程序，都是一种程序的静态创建（staticbuild）。动态库静态库很方便，但是如果我们只是想用库中的某一个函数，却仍然得把所有的内容都链接进去。一个更现代的方法则是使用共享库，避免了在文件中静态库的大量重复。动态链接可以在首次载入的时候执行(load-timelinking)，这是 Linux 的标准做法，会由动态链接器 ld-linux.so 完成，比方标准 C 库(libc.so)通常就是动态链接的，这样所有的程序可以共享同一个库，而不用分别进行封装。动态链接也可以在程序开始执行的时候完成(run-timelinking)，在 Linux 中使用 dlopen()接口来完成（会使用函数指针），通常用于分布式软件，高性能服务器上。而且共享库也可以在多个进程间共享。链接使得我们可以用多个对象文件构造我们的程序。可以在程序的不同阶段进行（编译、载入、运行期间均可），理解链接可以帮助我们避免遇到奇怪的错误。区别：使用静态库的时候，静态链接库要参与编译，在生成执行文件之前的链接过程中，要将静态链接库的全部指令直接链接入可执行文件中。而动态库提供了一种方法，使进程可以调用不属于其可执行代码的函数。函数的可执行代码位于一个.dll 文件中，该 dll 包含一个或多个已被编译，链接并与使用它们的进程分开储存的函数。静态库中不能再包含其他动态库或静态库，而在动态库中还可以再包含其他动态或者静态库。静态库在编译的时候，就将库函数装在到程序中去了，而动态库函数必须在运行的时候才被装载，所以使用静态库速度快一些。"10.介绍一下内存池、进程池、线程池"首先介绍一个概念“池化技术”。池化技术就是：提前保存大量的资源，以备不时之需以及重复使用。池化技术应用广泛，如内存池，线程池，连接池等等。内存池相关的内容，建议看看 Apache、Nginx 等开源 web 服务器的内存池实现。由于在实际应用当做，分配内存、创建进程、线程都会设计到一些系统调用，系统调用需要导致程序从用户态切换到内核态，是非常耗时的操作。因此，当程序中需要频繁的进行内存申请释放，进程、线程创建销毁等操作时，通常会使用内存池、进程池、线程池技术来提升程序的性能。线程池：线程池的原理很简单，类似于操作系统中的缓冲区的概念，它的流程如下：先启动若干数量的线程，并让这些线程都处于睡眠状态，当需要一个开辟一个线程去做具体的工作时，就会唤醒线程池中的某一个睡眠线程，让它去做具体工作，当工作完成后，线程又处于睡眠状态，而不是将线程销毁。进程池与线程池同理。内存池：内存池是指程序预先从操作系统申请一块足够大内存，此后，当程序中需要申请内存的时候，不是直接向操作系统申请，而是直接从内存池中获取；同理，当程序释放内存的时候，并不真正将内存返回给操作系统，而是返回内存池。当程序退出(或者特定时间)时，内存池才将之前申请的内存真正释放。"11.什么是临界资源"在操作系统中，进程是占有资源的最小单位（线程可以访问其所在进程内的所有资源，但线程本身并不占有资源或仅仅占有一点必须资源）。但对于某些资源来说，其在同一时间只能被一个进程所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源。典型的临界资源比如物理上的打印机，或是存在硬盘或内存中被多个进程所共享的一些变量和数据等(如果这类资源不被看成临界资源加以保护，那么很有可能造成丢数据的问题)。对于临界资源的访问，必须是互斥进行。也就是当临界资源被占用时，另一个申请临界资源的进程会被阻塞，直到其所申请的临界资源被释放。而进程内访问临界资源的代码被成为临界区。"12.如何处理死锁问题"忽略该问题。例如鸵鸟算法，该算法可以应用在极少发生死锁的的情况下。为什么叫鸵鸟算法呢，因为传说中鸵鸟看到危险就把头埋在地底下，可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗铃有点像。检测死锁并且恢复。仔细地对资源进行动态分配，使系统始终处于安全状态以避免死锁。通过破除死锁四个必要条件之一，来防止死锁产生。"13.死锁出现的条件？"定义:如果一组进程中的每一个进程都在等待仅由该组进程中的其他进程才能引发的事件,那么该组进程就是死锁的。或者在两个或多个并发进程中，如果每个进程持有某种资源而又都等待别的进程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组进程产生了死锁。通俗地讲，就是两个或多个进程被无限期地阻塞、相互等待的一种状态。产生死锁的必要条件：互斥条件(Mutualexclusion)：资源不能被共享，只能由一个进程使用。请求与保持条件(Holdandwait)：已经得到资源的进程可以再次申请新的资源。非抢占条件(Nopre-emption)：已经分配的资源不能从相应的进程中被强制地剥夺。循环等待条件(Circularwait)：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。"14.一个程序从开始运行到结束的完整过程（四个过程）"预处理：条件编译，头文件包含，宏替换的处理，生成.i 文件。编译：将预处理后的文件转换成汇编语言，生成.s 文件汇编：汇编变为目标代码(机器代码)生成.o 的文件链接：连接目标代码,生成可执行程序"15.说下你知道的调度算法"FIFO 或 FirstCome,FirstServed(FCFS)先来先服务调度的顺序就是任务到达就绪队列的顺序公平、简单(FIFO 队列)、非抢占、不适合交互式未考虑任务特性，平均等待时间可以缩短 ShortestJobFirst(SJF)最短的作业(CPU 区间长度最小)最先调度 SJF 可以保证最小的平均等待时间 ShortestRemainingJobFirst(SRJF)SJF 的可抢占版本，比 SJF 更有优势 SJF(SRJF):如何知道下一 CPU 区间大小？根据历史进行预测:指数平均法优先权调度每个任务关联一个优先权，调度优先权最高的任务注意：优先权太低的任务一直就绪，得不到运行，出现“饥饿”现象 Round-Robin(RR)轮转调度算法设置一个时间片，按时间片来轮转调度（“轮叫”算法）优点:定时有响应，等待时间较短；缺点:上下文切换次数较多时间片太大，响应时间太长；吞吐量变小，周转时间变长；当时间片过长时，退化为 FCFS 多级队列调度按照一定的规则建立多个进程队列不同的队列有固定的优先级（高优先级有抢占权）不同的队列可以给不同的时间片和采用不同的调度方法存在问题 1：没法区分 I/Obound 和 CPUbound 存在问题 2：也存在一定程度的“饥饿”现象多级反馈队列在多级队列的基础上，任务可以在队列之间移动，更细致的区分任务可以根据“享用”CPU 时间多少来移动队列，阻止“饥饿”最通用的调度算法，多数 OS 都使用该方法或其变形，如 UNIX、Windows 等"16.非抢占式调度与抢占式调度的区别是什么？"非抢占式：分派程序一旦把处理机分配给某进程后便让它一直运行下去，直到进程完成或发生进程调度进程调度某事件而阻塞时，才把处理机分配给另一个进程抢占式：操作系统将正在运行的进程强行暂停，由调度程序将 CPU 分配给其他就绪进程的调度方式"17.进程调度的种类有哪些？"高级调度：(High-LevelScheduling)又称为作业调度，它决定把后备作业调入内存运行低级调度：(Low-LevelScheduling)又称为进程调度，它决定把就绪队列的某进程获得 CPU 中级调度：(Intermediate-LevelScheduling)又称为在虚拟存储器中引入，在内、外存对换区进行进程对换"18.进程的通信方式有哪些"进程通信，是指进程之间的信息交换（信息量少则一个状态或数值，多者则是成千上万个字节）。因此，对于用信号量进行的进程间的互斥和同步，由于其所交换的信息量少而被归结为低级通信所谓高级进程通信指：用户可以利用操作系统所提供的一组通信命令传送大量数据的一种通信方式。操作系统隐藏了进程通信的实现细节。或者说，通信过程对用户是透明的高级通信机制可归结为三大类：共享存储器系统（存储器中划分的共享存储区）；实际操作中对应的是“剪贴板”（剪贴板实际上是系统维护管理的一块内存区域）的通信方式，比如举例如下：word 进程按下 ctrl+c，在 ppt 进程按下 ctrl+v，即完成了 word 进程和 ppt 进程之间的通信，复制时将数据放入到剪贴板，粘贴时从剪贴板中取出数据，然后显示在 ppt 窗口上消息传递系统（进程间的数据交换以消息（message）为单位，当今最流行的微内核操作系统中，微内核与服务器之间的通信，无一例外地都采用了消息传递机制。应用举例：邮槽（MailSlot）是基于广播通信体系设计出来的，它采用无连接的不可

靠的数据传输。邮槽是一种单向通信机制，创建邮槽的服务器进程读取数据，打开邮槽的客户机进程写入数据管道通信系统（管道即：连接读写进程以实现他们之间通信的共享文件（pipe 文件，类似先进先出的队列，由一个进程写，另一进程读））。实际操作中，管道分为：匿名管道、命名管道。匿名管道是一个未命名的、单向管道，通过父进程和一个子进程之间传输数据。匿名管道只能实现本地机器上两个进程之间的通信，而不能实现跨网络的通信。命名管道不仅可以在本机上实现两个进程间的通信，还可以跨网络实现两个进程间的通信管道：管道是单向的、先进先出的、无结构的、固定大小的字节流，它把一个进程的标准输出和另一个进程的标准输入连接在一起。写进程在管道的尾端写入数据，读进程在管道的道端读出数据。数据读出后将从管道中移走，其它读进程都不能再读到这些数据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程将一直阻塞。同样地，管道已经满时，进程再试图写管道，在其它进程从管道中移走数据之前，写进程将一直阻塞。信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其它进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段消息队列：是一个在系统内核中用来保存消息的队列，它在系统内核中是以消息链表的形式出现的。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点共享内存：共享内存允许两个或多个进程访问同一个逻辑内存。这一段内存可以被两个或两个以上的进程映射至自身的地址空间中，一个进程写入共享内存的信息，可以被其他使用这个共享内存的进程，通过一个简单的内存读取读出，从而实现了进程间的通信。如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。共享内存是最快的 IPC 方式，它是针对其它进程间通信方式运行效率低而专门设计的。它往往与其它通信机制（如信号量）配合使用，来实现进程间的同步和通信套接字：套接字也是一种进程间通信机制，与其它通信机制不同的是，它可用于不同机器间的进程通信"19.说下你对进程同步的理解"进程同步的主要任务：是对多个相关进程在执行次序上进行协调，以使并发执行的诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性同步机制遵循的原则：空闲让进；忙则等待（保证对临界区的互斥访问）；有限等待（有限代表有限的时间，避免死等）；让权等待（当进程不能进入自己的临界区时，应该释放处理机，以免陷入忙等状态）"20.为什么进程上下文切换比线程上下文切换代价高？"进程切换分两步：切换页目录以使用新的地址空间切换内核栈和硬件上下文对于 linux 来说，线程和进程的最大区别就在于地址空间，对于线程切换，第 1 步是不需要做的，第 2 是进程和线程切换都要做的切换的性能消耗：线程上下文切换和进程上下问切换一个最主要的区别是线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的。这两种上下文切换的处理都是通过操作系统内核来完成的。内核的这种切换过程伴随的最显著的性能损耗是将寄存器中的内容切换出另外一个隐藏的损耗是上下文的切换会扰乱处理器的缓存机制。简单的说，一旦去切换上下文，处理器中所有已经缓存的内存地址一瞬间都作废了。还有一个显著的区别是当你改变虚拟内存空间的时候，处理的页表缓冲（processor’sTranslationLookasideBuffer(TLB)）或者相当的神马东西会被全部刷新，这将导致内存的访问在一段时间内相当的低效。但是在线程的切换中，不会出现这个问题"21.说下进程和线程的联系与区别"进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位进程和线程的关系一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程是操作系统可识别的最小执行和调度单位资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段(代码和常量)，数据段(全局变量和静态变量)，扩展段(堆存储)。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量处理机分给线程，即真正在处理机上运行的是线程线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步进程与线程的区别进程有自己的独立地址空间，线程没有进程是资源分配的最小单位，线程是 CPU 调度的最小单位进程和线程通信方式不同(线程之间的通信比较方便。同一进程下的线程共享数据（比如全局变量，静态变量），通过这些数据来通信不仅快捷而且方便，当然如何处理好这些访问的同步与互斥正是编写多线程程序的难点。而进程之间的通信只能通过进程通信的方式进行。)进程上下文切换开销大，线程开销小一个进程挂掉了不会影响其他进程，而线程挂掉了会影响其他线程对进程进程操作一般开销都比较大，对线程开销就小了"22.说下进程的状态"就绪：进程已处于准备好运行的状态，即进程已分配到除 CPU 外的所有必要资源后，只要再获得 CPU，便可立即执行执行：进程已经获得 CPU，程序正在执行状态阻塞：正在执行的进程由于发生某事件（如 I/O 请求、申请缓冲区失败等）暂时无法继续执行的状态"