一、Java 基础

1. == 和 equals 的区别？

- ‘==’ 是用来比较两个变量（基本类型和对象类型）的值是否相等的， 如果两个变量是基本类型的，那很容易，直接比较值就可以了。如果两个变量是对象类型的，那么它还是比较值，只是它比较的是这两个对象在栈中的引用（即地址）。

- 对象是放在堆中的，栈中存放的是对象的引用（地址），由此可见 ‘==’ 是对栈中的值进行比较的。如果要比较堆中对象的内容是否相同，那么就要重写 equals 方法了。

- Object 类中的 equals 方法就是用 ‘==’ 来比较的，所以如果没有重写 equals 方法（string 是重写了 equal 方法的），equals 和 == 是等价的。通常我们会重写 equals 方法，让 equals 比较两个对象的内容，而不是比较对象的引用（地址）因为往往我们觉得比较对象的内容是否相同比比较对象的引用（地址）更有意义。

2. 拆箱和装箱分别是什么？分别应用在什么场景？

- 拆箱和装箱：拆箱是把包装类型变成基本类型；装箱是把基本类型变成包装类型。

- 应用场景：当== 两边是基本类型和包装类型的时候，包装类型会拆箱，然后==会比较基本类型的两个值；当 equals 方法的参数是基本类型的时候，基本类型会装箱，然后 equals 方法会先比较两个包装类型的类型，在比较值。

3. String str="abc" 与 String str=new String (“abc”) 一样吗？为什么？

- 使用 String 直接赋值：String str = “abc”：可能创建一个或者不创建对象。如果”abc” 在字符串池中不存在，会在 java 字符串池中创建一个 String 对象（”abc”），然后 str 指向这个内存地址，无论以后用这种方式创建多少个值为”abc” 的字符串对象，始终只有一个内存地址被分配；如果 “abc” 在字符串池中存在，str 直接指向这个内存地址。

- 使用 new String 创建字符串：String str = new String(“abc”)：至少会创建一个对象，也有可能创建两个。因为用到 new 关键字，肯定会在堆中创建一个 String 对象，如果字符池中已经存在”abc”，则不会在字符串池中创建一个 String 对象，如果不存在，则会在字符串常量池中也创建一个对象。注意，此时二者地址不相同。

- String 拼接字符串：字符串拼接又分为变量拼接和已知字符串拼接。只要拼接内容存在变量，那么该拼接后的新变量就是在堆内存中新建的一个对象实体。但是不会在字符常量池中创建新对象。

- String.intern()：当调用 intern 方法时，如果常量池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到常量池中，并返回此 String 对象的引用。

4. String str=“abc” 和 String str=new String(“abc”); 产生几个对象？ String str = new String("hello")+new String("123");产生了几个对象？

- 前者 1 或 0， 后者 2 或 1，最后一个是 3 或 4 或 5

- 先看字符串常量池，如果字符串常量池中没有，都在常量池中创建一个，如果有，前者直接引用，后者在堆内存中还需创建一个 “abc” 实例对象。

5. 字符常量池的位置？字符常量池存储的内容？

- 字符串常量池的位置，jdk1.7 从方法区移至堆中。

- 字符串常量池存储的是字符串字面量（注意，这里的字面量指的就是 String 的对象，因为 String 本身是一个类，而字符常量池在 jdk1.8 的时候放在堆中，所以存储的内容全都是对象）

6. 给定三个变量：i1、i2、i3。Integer i1 = 120；Integer i2 = 120；int i3= 120；i1 和 i2 一样吗？i1 和 i3 呢？为什么？如果把 120 换成 130 呢，i1,12,i3 的关系又如何，为什么？

- integer 包装类和其他包装类不同之处，对于范围在-128 到 127 之间的整数，integer 创建对象的时候是直接从缓存读取的，这个时候直接从缓存得的的内容地址是保持一致的，所以两个 integer 对象地址是相同的，但是不在这个范围的时候，地址就不在相同的。对于 integer 变量和 int 变量，通过==比较，符合拆箱应用场景，integer 会直接拆箱，所以无论什么范围，integer 和 int，都是直接比较值的。

7. throw 和 throws 的区别？

throws 声明：

- 如果一个方法内部的代码会抛出检查异常（checked exception），而方法自己又没有完全处理掉，则 javac 保证你必须在方法的签名上使用 throws 关键字声明这些可能抛出的异常，否则编译不通过。

- throws 是另一种处理异常的方式，它不同于 try…catch…finally，throws 仅仅是将函数中可能出现的异常向调用者声明，而自己则不具体处理。

throw 异常抛出语句：

- 程序员也可以通过 throw 语句手动显式的抛出一个异常。throw 语句的后面必须是一个异常对象。

- throw 语句必须写在函数中，执行 throw 语句的地方就是一个异常抛出点，它和由 JRE 自动形成的异常抛出点没有任何差别。

- 在异常处理中，try 语句要捕获的是一个异常对象，那么此异常对象也可以自己抛出。

8. try-catch-finally 中哪个部分可以省略？

- finally 可以省略，其他都不可以省略

9. try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？

- 在 try 块中即便有 return，break，continue 等改变执行流的语句，finally 也会执行。

- try…catch…finally 中的 return 只要能执行，就都执行了，他们共同向同一个内存地址（假设地址是 0×80）写入返回值，后执行的将覆盖先执行的数据，而真正被调用者取的返回值就是最后一次写入的。finally 中的 return 会覆盖 try 或者 catch 中的返回值。

- finally 中的 return 和方法的 return 不能共存，因为 finally 是异常处理的最后执行的部分，finally 必定执行，内部的 return 跟方法最后的 return 存在冲突。

10. 常见的异常类有哪些？

- error 错误类：StackOverFlowError、OutOfMemoryError

- exception 类：IOException（FileNotFindException）、RuntimeException（NullPointException）

11. java 异常的执行流程？

在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。

- 抛出异常：当一个方法出现错误引发异常时，方法创建异常对象并交付运行时系统，异常对象中包含了异常类型和异常出现时的程序状态等异常信息。运行时系统负责寻找处置异常的代码并执行。

- 捕捉异常：在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler）。潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合。当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适的异常处理器。运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适的异常处理器，则运行时系统终止。同时，意味着 Java 程序的终止。

12. 为什么要使用克隆？如何实现对象克隆？

Java 中的对象拷贝 (Object Copy) 指的是将一个对象的所有属性（成员变量）拷贝到另一个有着相同类类型的对象中去。 在程序中拷贝对象是很常见的，主要是为了在新的上下文环境中复用现有对象的部分或全部数据。

Java 中的对象拷贝主要分为：浅拷贝 (Shallow Copy)、深拷贝 (Deep Copy)。

13. 深拷贝和浅拷贝区别是什么？

浅拷贝只能应用于成员变量全是基本类型的对象，而深拷贝可以应用成员变量是对象的对象。

一般步骤是（浅复制）：

- 被复制的类需要实现 Clonenable 接口（不实现的话在调用 clone 方法会抛出 CloneNotSupportedException 异常) 该接口为标记接口 (不含任何方法)

- 覆盖 clone () 方法，。方法中调用 super.clone () 方法得到需要的复制对象，注意该方法需要处理异常。

深拷贝的方式有两种：

- 第一种：与通过重写 clone 方法实现浅拷贝的基本思路一样，只需要为对象图的每一层的每一个对象都实现 Cloneable 接口并重写 clone 方法，最后在最顶层的类的重写的 clone 方法中调用所有的 clone 方法即可实现深拷贝。简单的说就是：每一层的每个对象都进行浅拷贝 = 深拷贝。

- 第二种：结合序列化来解决这个问题，先把对象序列化，然后再反序列化成对象，该对象保证每个引用都是崭新的。这个就形成了多个引用，原引用和反序列化之后的引用不在相同，具体实现：

14. 值传递和引用传递的区别是什么？

- 值传递：在方法的调用过程中，实参把它的实际值传递给形参，此传递过程就是将实参的值复制一份传递到函数中，这样如果在函数中对该值（形参的值）进行了操作将不会影响实参的值。因为是直接复制，所以这种方式在传递大量数据时，运行效率会特别低下。

- 引用传递：弥补了值传递的不足，如果传递的数据量很大，直接复过去的话，会占用大量的内存空间，而引用传递就是将对象的地址值传递过去，函数接收的是原始值的首地址值。在方法的执行过程中，形参和实参的内容相同，指向同一块内存地址，也就是说操作的其实都是源数据，所以方法的执行将会影响到实际对象。

15. 什么是 java 序列化？什么情况下需要序列化？如何避免序列化对象中的属性序列化？

- 在 Java 中，我们可以通过多种方式来创建对象，并且只要对象没有被回收我们都可以复用此对象。但是，创建出来的这些对象都存在于 JVM 中的堆（stack）内存中，只有 JVM 处于运行状态的时候，这些对象才可能存在。一旦 JVM 停止，这些对象也就随之消失；但是在真实的应用场景中，我们需要将这些对象持久化下来，并且在需要的时候将对象重新读取出来，Java 的序列化可以帮助我们实现该功能。

- 对象序列化机制（object serialization）是 java 语言内建的一种对象持久化方式，通过对象序列化，可以将对象的状态信息保存为字节数组，并且可以在有需要的时候将这个字节数组通过反序列化的方式转换成对象，对象的序列化可以很容易的在 JVM 中的活动对象和字节数组（流）之间进行转换。

- Java 类通过实现 java.io.Serializable 接口来启用序列化功能，未实现此接口的类将无法将其任何状态或者信息进行序列化或者反序列化。可序列化类的所有子类型都是可以序列化的。序列化接口没有方法或者字段，仅用于标识可序列化的语义。在 JAVA 中，对象的序列化和反序列化被广泛的应用到 RMI（远程方法调用）及网络传输中。

属性避免序列化：

- 静态数据不能被序列化，因为静态数据不在堆内存中，而是在静态方法区中

- 将不需要序列化的属性前添加关键字 transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中。

16. 什么是反射？反射的应用场景？

- JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

获取类对象的三种方式：

- 通过 object 类的 getClass () 函数，由于 object 是根类，每一个类都有这个函数
- 每一个类（包括基本数据类型，注意这里基本数据类型不用转成包装类）都有一个 class 属性，静态属性，通过类名直接访问
- 通过 Class 类的静态方法 forName(String className)

应用：

- 反射是很多框架的基础
- 通过反射运行配置文件
- 通过反射越过泛型检查

17. 代理模式有什么用？应用场景是什么？

- 代理（Proxy）模式是结构型的设计模式之一，它可以为其他对象提供一种代理（Proxy）以控制对这个对象的访问。所谓代理，是指具有与被代理的对象具有相同的接口的类，客户端必须通过代理与被代理的目标类交互。

应用场景：

- 需要控制对目标对象的访问。
- 需要对目标对象进行方法增强。如：添加日志记录，计算耗时等。
- 需要延迟加载目标对象。

18. 动态代理的实现方式都有什么？那种实现效率高？

实现动态代理的两种方式： JDK 动态代理和 Gglib 动态代理

- JDK 动态代理是实现了被代理对象的接口， Cglib 是继承了被代理对象。
- JDK 和 Cglib 都是在运行期生成字节码，JDK 是直接写 Class 字节码，Cglib 使用 ASM 框架写 Class 字节码，Cglib 代理实现更复杂，生成代理类比 JDK 效率低。
- JDK 调用代理方法，是通过反射机制调用，Cglib 是通过 FastClass 机制直接调用方法，Cglib 执行效率更高。

19. 动态代理是什么？与静态代理的区别在于？

Java 中的静态代理要求代理主题 (ProxySubject) 和真实主题 (RealSubject) 都实现同一个接口 (Subject)。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，一但需要修改接口，代理类和委托类都需要修改。

20. 抽象类和接口的区别？普通类和抽象类有哪些区别？

抽象类和普通类的唯一区别就在于是否实例化

抽象类和接口：

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

21. 抽象类必须要有抽象方法吗？抽象类能使用 final 修饰吗？

- 抽象类中可以有普通方法，也可以有抽象方法
- 但是有抽象方法的类一定是抽象类，但是抽象类中不一定有抽象方法。

- abstract 和 final 不共存，final 不能继承
- abstract 和 static 不共存 ，因为 abstract 不能修饰属性

22. 局部内部类和匿名内部类为什么只能访问 final 的局部变量？

- 第一个问题：内部类当中可以调用外部类当中的属性和方法，但是当外部类的方法结束时，局部变量就会被销毁了，内部类对象访问了一个不存在的变量。为了解决这个问题，就将局部变量复制了一份作为内部类的成员变量，这样当局部变量死亡后，内部类仍可以访问它，实际访问的是局部变量的”copy”。这样就好像延长了局部变量的生命周期。

- 第二个问题：将局部变量复制为内部类的成员变量时，必须保证这两个变量是一样的，也就是如果我们在内部类中修改了成员变量，方法中的局部变量也得跟着改变，但是将局部变量设置为 final，对它初始化后，就不让你再去修改这个变量，就保证了内部类的成员变量和方法的局部变量的一致性。这实际上也是一种妥协。

23. 什么是多态？

- 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，必须在由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，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，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。

24. BIO、NIO、AIO 有什么区别？

- BIO (Blocking I/O): 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。

- NIO (New I/O): NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel , Selector，Buffer 等抽象。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

- AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2, 它是异步非阻塞的 IO 模型。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。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备

好时，得到通知，接着就由这个线程自行进行 IO 操作， IO 操作本身是同步的。

25. 同步、异步、阻塞、非阻塞？

- 同步，就是在发出一个调用时，在没有得到结果之前，该调用就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由调用者主动等待这个调用的结果。
- 异步则是相反，调用在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在调用发出后，被调用者通过状态、通知来通知调用者，或通过回调函数处理这个调用。
- 阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

26. java 中 IO 流分为几种？

- 按照流的流向分，可以分为输入流和输出流；
- 按照操作单元划分，可以划分为字节流和字符流；
- InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

27. final 在 java 中有什么作用？抽象类能使用 final 修饰吗？

- final 类不能被继承（final 类内的 method 自动为 final，但不包括属性）
- final 方法可以被继承但不能被 override
- final 属性不能被重新赋值（可以被继承，但不可以修改）定义时可以初始化，也可以不初始化，而在语句块中初始化或者构造函数中初始化（最晚要在构造函数中初始化，只能初始化一次），final 定义的成员变量可以在代码块 (类变量则静态代码块，实例变量普通代码块) 里初始化
- final 属性只能人为赋值一次，继承与父类的 final 属性不能被修改
- final 可以修饰局部变量表示局部常量（方法级局部变量（形参及局部变量，方法体级局部变量）或块级局部变量）

28. java 中的 Math.round (-1.5) 等于多少？

- 这些方法的作用于它们的英文名称的含义相对应，例如：ceil 的英文意义是天花板，该方法就表示向上取整，Math.ceil（11.3）的结果为 12，Math.ceil (-11.6) 的结果为 - 11；floor 的英文是地板，该方法就表示向下取整，Math.floor (11.6) 的结果是 11，Math.floor (-11.4) 的结果 - 12；最难掌握的是 round 方法，他表示“四舍五入”，算法为 Math.floor (x+0.5)，即将原来的数字加上 0.5 后再向下取整，所以，Math.round (11.5) 的结果是 12，Math.round (-11.5) 的结果为 - 11.

- Math.round (-1.5) = -1

29. 管道的类型？

管道类型：

- 普通管道（PIPE）：通常有两种限制，一是单工，即只能单向传输；二是血缘，即常用于父子进程间（或有血缘关系的进程间）。

- 流管道（s_pipe）：去除了上述的第一种限制，实现了双向传输。

- 命名管道（name_pipe）：去除了上述的第二种限制，实现了无血缘关系的不同进程间通信。

30. 什么是半双工？什么是全双工？

- 半双工，虽然可以双向，但是同一时间只能有一个方向传输，

- 全双工，同一时间可以双向传输

31. 在多线程下选用什么处理大规模字符串？

- String 是 final 类不能被继承且为字符串常量，而 StringBuilder 和 StringBuffer 均为字符串变量。String 对象一旦创建便不可更改，而后两者是可更改的，它们只能通过构造函数来建立对象，且对象被建立以后将在内存中分配内存空间，并初始保存一个 null，通过 append 方法向 StringBuffer 和 StringBuilder 中赋值。

- String（线程安全）：不可变类（Immutable Class）是一旦被实例化就不会改变自身状态（或值）的类。String 就是一种典型的不可变类。（使用字符串自带的函数改变 string 内容都是相当于创建一个新的 string，即 new String）。不可变类的主要用途是在多线程环境下确保对象的线程安全。

- StringBuffer（线程安全的）：StringBuffer 中大部分方法由 synchronized 关键字修饰，

在必要时可对方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致，所以是线程安全的。

- StringBuilder（非线程安全）：StringBuilder 的方法不能保证线程安全性。StringBuilder 是 JDK1.5 新增的，该类提供一个与 StringBuffer 兼容的 API，但不能保证同步，所以在性能上较高。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同。

32. Java 中的设计原则？

- 开闭原则（Open Close Principle）：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。

- 里氏代换原则（Liskov Substitution Principle）：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。

- 依赖倒转原则（Dependence Inversion Principle）：针对接口编程，依赖于抽象而不依赖于具体。

- 接口隔离原则（Interface Segregation Principle）：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：降低依赖，降低耦合。

- 迪米特法则（最少知道原则）（Demeter Principle）：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。

- 合成复用原则（Composite Reuse Principle）{} 原则是尽量使用合成 / 聚合的方式，而不是使用继承

33. 什么是组合？什么是聚合？

- 聚合用来表示 “拥有” 关系或者整体与部分的关系

- 组合用来表示一种强得多的 “拥有” 关系

比如 A 类中包含 B 类的一个引用 b，当 A 类的一个对象消亡时，b 这个引用所指向的对象也同时消亡（没有任何一个引用指向它，成了垃圾对象），这种情况叫做组合，反之 b 所指向的对象还会有另外的引用指向它，这种情况叫聚合。

- 组合方式一般会这样写：A 类的构造方法里创建 B 类的对象，也就是说，当 A 类的一个对象产生时，B 类的对象随之产生，当 A 类的这个对象消亡时，它所包含的 B 类的对象也随之消亡。

- 聚合方式则是这样：A 类的对象在创建时不会立即创建 B 类的对象，而是等待一个外界的对象传给它，给它的这个对象不是 A 类创建的。

34. 说一下你熟悉的设计模式？

- 观察者模式
- 装饰者模式
- 代理模式

35. 简单工厂和抽象工厂有什么区别？

- 工厂模式：定义一个用于创建对象的接口，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

36. 装饰者模式和适配器模式以及代理模式的区别？

装饰者和代理模式的区别

- 装饰者模式关注的是对象的动态添加功能。代理模式关注的是对对象的控制访问，对它的用户隐藏对象的具体信息。

装饰者模式和适配器模式

- 装饰者模式和被装饰的类要实现同一个接口，或者装饰类是被装饰的类的子类。适配器模式和被适配的类具有不同的接口。
- 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。

37. 说出几个在 JDK 库中使用的设计模式？

- 装饰器设计模式（Decorator design pattern）被用于多个 Java IO 类中。单例模式

（Singleton pattern）用于 Runtime，Calendar 和其他的一些类中。工厂模式（Factory pattern）被用于各种不可变的类如 Boolean，像 Boolean.valueOf，观察者模式（Observer pattern）被用于 Swing 和很多的事件监听中。

38. jdk1.8 新增的变化

- Hashmap 红黑树，扩容
- concurrentHashmap 红黑树，进一步减少锁的粒度
- Jvm 常量池的变化，字符常量池在 jdk1.7 从方法区脱离放入堆中，方法区则在 jdk1.8 中变成元空间。

二、JVM

1. 说一下 jvm 的主要组成部分？及其作用？

- Class Loader 类加载器：类加载器的作用是加载类文件到内存，比如编写一个 HelloWord.java 程序，然后通过 javac 编译成 class 文件，那怎么才能加载到内存中被执行呢？Class Loader 承担的就是这个责任，当然，不可能随便建立一个.class 文件就能被加载的，Class Loader 加载的 class 文件是有格式要求。

- Execution Engine 执行引擎：Class Loader 只管加载，只要符合文件结构就加载，至于说能不能运行，则不是它负责的，那是由 Execution Engine 负责的。执行引擎也叫做解释器 (Interpreter)，负责解释命令，提交操作系统执行。

- Native Interface 本地接口：本地接口的作用是融合不同的编程语言为 Java 所用，它的初衷是融合 C/C++ 程序，Java 诞生的时候是 C/C++ 横行的时候，要想立足，必须有一个聪明的、睿智的调用 C/C++ 程序，于是就在内存中专门开辟了一块区域处理标记为 native 的代码。

- Runtime data area 运行数据区：运行数据区是整个 JVM 的重点。

2. jvm 运行时数据区组成？

- 堆：线程公有，存放对象。

- 栈：线程私有，由栈帧组成，保存函数的一些信息。

- PC 计数器：代码指令行号计数器

- 方法区（持久代、元数据区）

3. 堆栈的区别？

- 堆：Java 内存管理中最大的一块，用来存储对象实例和数组，线程共享。

- 栈：分为虚拟机栈和本地方法栈，这里主要指的是虚拟机栈。虚拟机栈由一个个栈帧组成，每一个栈帧又由局部变量表、动态链接（对象引用）、方法出口等信息组成，线程私有。

4. 运行时数据区哪些是线程共享，哪些是线程私有？

- 线程私有：栈、PC 计数器

- 线程共享：堆、方法区（元空间）

5. Java 中成员变量、局部变量、静态变量、常量分别存储在那些内存区域中？

- 成员变量：堆

- 局部变量：
  - 局部变量是基本类型（8 种基本类型）：栈
  - 局部变量是对象：对象实例存储在堆中，对象引用存储在栈中

- 静态变量（static 修饰）：方法区（元数据区）

- 常量（final 修饰）：方法区（元数据区）

6. 说一下类加载的执行过程？

- 加载：把 class 文件加载到内存中

- 链接（又分为：验证、准备、解析）
  - 验证：验证加载到内存的文件是否符合格式要求
  - 准备：给类变量（static 修饰的变量）赋初值（系统默认值）。
  - 解析：把符合引用转变成直接引用

- 初始化：按照继承关系的顺序给 静态变量赋初值（代码中设定的值）

7. Java 中都有哪些加载器？

- 启动类加载器、扩展类加载器、系统类加载器、用户自定义类加载器

- 启动类加载器将<JAVA_HOME>/lib 下的核心类库加载到内存中，扩展类加载器将<JAVA_HOME>/lib/ext 下的类库加载到内存中，系统类加载器将当前类和第三方类库加载到内存中

- 扩展类加载器和系统类加载器都是继承自 Java 的 ClassLoader 抽象类，用户可以直接使用的类加载器。

- 启动类加载器，Java 程序不可用直接使用，虚拟机默认使用 null 代表启动类加载器

- 三者存在继承关系，系统类加载器调用 classloader 的构造器把父类设置为扩展类加载器，扩展类加载器调用 classloader 把父类设置为 null（启类动加载器）

8. 什么是双亲委派模型？

- 当一个类加载器接收到一个类加载的任务时，不会立即展开加载，而是将加载任务委托给它的父类加载器去执行，每一层的类都采用相同的方式，直至委托给最顶层的启动类加载器为止。如果父类加载器无法加载委托给它的类（它的搜索范围中没有找到所需要加载的类），便将类的加载任务退回给下一级类加载器去执行加载。

- 好处：保证类加载的全局唯一性，加载的类随着加载器一起具备了优先级关系。

- 实现：先判断该类是否被加载过，如果未被加载就调用父类的 loadClass()方法，如果父类为 null，则使用启动类加载器作为加载器。如果父类加载失败，抛出 notfound 异常，则调用自己的 findClass 进行加载。简单概括：自底向上检查类是否被加载，自顶向下尝试加载类。

9. 反射中，Class.forName() 和 ClassLoader.loadClass()区别

- Class.forName() 执行的是类加载过程的链接和初始化。需要整个类完全加载到内存中，以获取该类的信息。

- ClassLoader.loadClass() 执行的只是类加载过程中的第一步，加载过程。loadClass 方法是在双亲委派中调用，此时判断类由哪一个类加载器加载，因此类还未加载到内存中。

10. 说一下对象创建的过程？

- 类加载检查：在执行到 new 命令时，查看 new 后面的参数是否正确定位到常量池中的符号引用，并且该符号引用是否被正确的加载、连接和初始化。

- 内存分配：为对象分配内存，有指针碰撞和空闲列表两种方式。
  - 指针碰撞：适用于内存完整，无碎片。内存使用和未使用中间由指针隔开，分配的时候指针移动相应的位置。
  - 空闲列表：适用于内存不完整，有碎片。维护一个列表，列表记录可用的内存，使用时分配给对象一个足够的内存空间并更新列表

- 初始化零值：将对象分配到的空间初始化零值（不包括对象头）

- 设置对象头：对象头信息包括哈希码、GC 分代信息、元数据信息、对象是哪个类的示例等

- 执行 init 方法：给对象设置程序值，执行构造方法。

11. 对象有哪几部分构成？虚拟机如何访问对象？

对象在内存中的组成部分（对象在内存的划分、对象的内存布局）：

- 对象头：
  - 第一：元数据、GC 分代、哈希值等自身运行信息；
  - 第二：类型指针，确定属于哪个类的实例

- 实例数据：对象真正存储的有效信息，定义的各种字段的值。
- 对齐填充：没有实际意义，JVM 内存地址需要 8 字节的整数倍

虚拟机如何访问对象

- 句柄：堆中专门划分一个区域作为句柄池，虚拟机栈存储是堆中句柄的地址，句柄存储的是对象的实例数据和类型数据的地址

- 直接指针：栈中的引用直接指向的就是对象的实例数据和类型数据的地址。

- 对比：直接指针避免二次寻址；使用句柄，在对象移动时，只修改句柄地址而不用改变引用的地址。

12. java 中都有哪些引用类型？

- 强引用：new 语句产生的都是强引用，虚拟机不会主动去回收，即便内存溢出，也是会抛出异常而不是回收强引用。可以将对象=null，从而在垃圾回收器在下一次回收。

- 软引用：在内存足够时，不会回收，在内存不足时，会回收。常用于缓存技术。
- 弱引用：垃圾回收器遇到该引用就会回收，常与引用队列一起使用
- 虚引用：最弱的引用，在对象被 JVM 回收之后收到一个系统通知，用于追踪垃圾回收过程，必须与应用队列一起使用。

13. 怎么判断对象是否可以被回收？

先判断对象是否为不可用：

- 引用计数法：引用+1，引用失效-1，对象引用为 0，则不可在使用；无法解决对象互相循环引用的问题。
- 可达性分析：选用某些对象作为 GC Roots 节点，从 Roots 节点向下搜索，形成一个引用链，所有不在引用链上的对象都是不可使用的对象

然后对不可用对象进行判断是否回收，进行两次标记（两次判断）

- 第一次标记：所有没在 GC Roots 引用链上的对象都进行第一次标记，所有执行过 finalize 方法或者未覆盖 finalize 方法的对象，直接回收；除此之外的标记过一次的对象都进入到 F-Queue 队列中。
- 对 F-Queue 中的对象进行二次标记，如果在 finalize 方法中，对象又重新加入引用链，则不回收，否则就进行回收。

14. 内存泄露和内存溢出分别是什么？什么原因造成？如何避免？

- 内存泄露：本应被回收的对象因为其他对象的引用而不能被回收，从而在堆中寄存，造成内存泄露，长周期对象持有短周期对象的引用会造成。
- 内存溢出：无法为对象分配足够的内存，对象申请过多

避免：

- 不要在循环中创建对象
- 不要一次调用过多数据
- 大量字符串使用 StringBuffer 或者 StringBuilder

- 方法区很少进行垃圾回收，尽量避免申请常量和静态变量

15. 给对象分配内存如何保证线程安全？

- CAS+失败重试：CAS 是乐观锁，每次都假设成功，在执行，失败就重试，找到成功。保证更新的原子性

- TLAB：为每一个线程，预先在 Eden 区域分配一块内存，对象先分配到这里，当该区域内存不足或用完之后，使用 CAS+失败重试机制

16. 说一下 jvm 有哪些垃圾回收算法？

标记是通过 可达性分析法，以 GC Roots 对象为根节点 把引用链上的对象标记

- 标记-清除 算法：分标记和清除两个部分，清除就是把未标记的对象清除掉。效率不高，并且产生空间碎片。

- 复制算法：将内存分成两个部分，一个用于存储对象，一个不使用。回收的时候，把使用的内存中的对象全部复制到不使用的内存中，然后清除使用的内存，并交换二者的角色。不会产生空间碎片但是浪费一半内存。不适用对象存活较多的情况。

- 标记-整理 算法：先把对象标记处理，再把对象全部压缩到内存的边界，再把边界外的内存清空。适用于对象存活较多的情况。

17. 说一下 jvm 有哪些垃圾回收器？

- serial 回收器：单线程回收器，回收的时候会暂停所有用户线程 stop the world，复制算法，新生代

- parnew 回收器：serial 的多线程版，其他的跟 serial 一致，复制算法，新生代

- parallel scavenge 回收器：多线程版，对 CPU 资源敏感，注重吞吐量，复制算法，新生代

- serial old 回收器：serial 的老年代版本，标记-整理 算法，老年代

- parallel old 回收器：parallel 的老年代版本，标记-整理算法，老年代

- CMS 回收器：注重用户体验，以获取最短回收停顿时间为目的，第一款并发收集器。采用标记-清除算法，老年代

- G1 回收器：面向服务器。低停顿，可预测的停顿时间模型，作用整个堆，把内存区域划分为一个个 region，不会产生空间碎片

18. 详细介绍一下 CMS 垃圾回收器？

- 注重用户体验，以获取最短回收停顿时间为目的，第一款并发收集器。采用标记-清除算法，老年代

- 运行过程：
  - 初始标记：直接与 GC Roots 连接的对象，暂停其他用户程序
  - 并发标记：并发进行，GC Roots 追踪引用链与用户程序一起执行
  - 重新标记：修正因程序运行而导致的对象标记的变化，暂停其他用户程序
  - 并发清除：清除标记的对象，与用户程序一起执行

- 优点：并发、低停顿

- 缺点：对 CPU 资源敏感；无法处理浮动垃圾；产生空间碎片。

19. 新生代垃圾回收器和老生代垃圾回收器都有哪些？有什么区别？

- 新生代收集器：Serial、ParNew、Parallel Scavenge

- 老年代收集器：CMS、Serial Old、Parallel Old

- 整堆收集器：G1

- 区别：算法、单线程多线程、侧重方面（吞吐量、停顿时间、效率）

20. 简述分代垃圾回收器是怎么工作的？

- 新生代和老年代的对象数量和存活时间不同，所以通常是分代使用不同的垃圾回收器。

- 少量对象存活，适合复制算法：在新生代中，每次 GC 时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成 GC。

- 大量对象存活，适合用标记 - 清理 / 标记 - 整理：在老年代中，因为对象存活率高、没有额外空间对他进行分配担保，就必须使用“标记 - 清理”/“标记 - 整理”算法进行 GC。

21. G1 为什么能建立可预测的停顿时间模型？

- G1 收集器的工作范围是整个 Java 堆。在使用 G1 收集器时，它将整个 Java 堆划分为多个大小相等的独立区域（Region）。虽然也保留了新生代、老年代的概念，但新生代和老年代不再是相互隔离的，他们都是一部分 Region（不需要连续）的集合。G1 跟踪各个 Region 里面的垃圾堆积的大小，对各个 Region 的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。这样就保证了在有限的时间内可以获取尽可能高的收集效率。

22. Minor Gc 和 Full GC 有什么不同呢？

- minor GC 是针对新生代，major GC 是清理老年代，Full GC 是清理整个堆

23. 什么对象会进入老年代

- 大对象直接进入老年代
- 长期存活的对象直接进入老年代

24. 如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？

- 不会立刻进行垃圾回收，而是在下一次进行回收，因为垃圾回收是优先度比较低的线程，设置为 null 之后不会立即执行垃圾回收，而是在垃圾回收执行到的时候，进行回收。

25. 常量池都包括哪些内容？常量池的位置？

Java 中有三个常量池：字符串常量池、运行时常量池、class 常量池

- 字符常量池：全局字符串池里的内容是在类加载完成，经过验证，准备阶段之后在堆中生成字符串对象实例，也就是编译期间字符常量池已经创建完成。
- class 文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池 (constant pool table)——注意这里的 class 常量池是存放在 class 文件中，而不是在 jvm 中，用于存放编译器生成的各种字面量 (Literal) 和符号引用 (Symbolic References)。 字面量就是我们所说的常量概念，符号引用就是在类加载阶段的解析阶段，把符号引用替换成直接引用。
- jvm 在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，jvm 就会将 class 常量池中的内容存放到运行

时常量池中，运行时常量池也是每个类都有一个。

- 关于 class 常量池和运行时常量池可以这么理解：在 java 文件编译时，类的字面量和符号引用，会存放在 class 类常量池，但是当类文件加载到 jvm 之后，class 常量池的内容就被会加载到运行时常量池。

常量池的位置

- 在 JDK1.7 之前运行时常量池逻辑包含字符串常量池存放在方法区，此时 hotspot 虚拟机对方法区的实现为 永久代

- 在 JDK1.7 字符串常量池被从方法区拿到了堆中，这里没有提到运行时常量池，也就是说字符串常量池被单独拿到堆，运行时常量池剩下的东西还在方法区 ，也就是 hotspot 中的永久代

- 在 JDK1.8 hotspot 移除了永久代用元空间 (Metaspace) 取而代之，这时候 字符串常量池还在堆 ，运行时常量池还在方法区 ，只不过方法区的实现从永久代变成了元空间 (Metaspace)

三、容器类和集合框架

1. java 容器都有哪些？

容器分为三大类：

- List：有序可重复集合，ArrayList,LinkedList,Vector,Stack

- Set：无序不可重复集合,HashSet,TreeSet,LinkedHashSet

- Map：关系映射，HashMap,TreeMap,LinkedHashMap,HashTable

2. Collection 和 Collections 有什么区别？

- Collection 是容器接口，是 List 和 Set 的根接口

- Collections 是工具类，提供处理集合的各种方法，

3. List、Set、Map 之间的区别是什么？

- List：有序可重复集合，ArrayList,LinkedList,Vector,Stack
- Set：无序不可重复集合,HashSet,TreeSet,LinkedHashSet
- Map：关系映射，HashMap,TreeMap,LinkedHashMap,HashTable

4. 如何决定使用 HashMap 还是 TreeMap？

- HashMap 效率高，存取数据快，但是数据无序
- TreeMap 效率低，但是存储数据是排序的，可以取到最大和最小值

5. ArrayList 和 LinkedList 的区别是什么？

- ArrayList 动态数组，索引方便，插入不方便
- LinkedList 链表，索引不方便，插入方便

6. ArrayList 和 Vector 的区别是什么？

- ArrayList 线程不安全，扩容 *1.5
- Vector 方法使用 synchronized 关键字。线程安全，扩容*2，效率比 ArrayList 低，适合存大数据并且要求线程安全

7. 说一下 HashMap 的实现原理？

- 底层实现：hashmap 底层实现是：数组+链表+红黑树（链表元素大于 8，并且数组长度大于 64 的时候，链表会转化为红黑树）

8. java 如何判断 HashMap 中的元素是否相等？添加的元素是自定义类的时候，需要注意什么？

java 中：

- 两个元素 hashcode 不相等，则二者一定不 equals
- 两个元素 equals，则认为二者 hashcode 相等

判断相等：

- Java 的所有的类都会继承 object，object 类有两个函数，一个是 hashcode（根据直接地址返回一个整型值，object 是直接返回地址），一个是 equals（判断两个对象是否相等，object 底层实现是==，还是比较地址），java 用这两个函数判断 hashmap 的元素是否相等，具体过程是：hashcode 与数字长度-1 进行进行与运算获得数组下标，然后利用 equals 判断是否有相等元素，没有则插入，有则跳过。

自定义类：

- 如果添加的元素 key 是自定义类，那么需要重写该类的 hashcode 和 equals 方法，因为 hashcode 是返回直接地址，对于对象来说，直接地址都是不相同的，例如建立两个内容相同的对象，但是对于 hashmap 来说，这两个对象就是不相同的，同样的，对于 equal 也是，底层实现是==，也是对直接地址判断。

9. HashMap 为什么引入红黑树？

- 原有的数组+链表的方式，存在极端情况，即数组很短，链表很长，这样的查询效率接近 O(n)，引入红黑树，查询时间变为 O（logn），大大提高查询效率

10. JDK1.8 的 HashMap 有哪些优化？

- 引入红黑树

- 扩容的时候不需要重新计算 hashcode，采用与运算获得数组下标，当数组大小变为 2 倍的时候，获得的数组下标从后 n 位变成了 n+1 位，多出来的 bit 位，是 0 就不变，是 1 就加 oldcap。

- 扩容的链表不在逆序

11. 请写出 HashMap 的添加操作和扩容操作的代码

put 函数伪代码：

- 判断键值对数组 tab [] 是否为空或为 null，否则以默认大小 resize ()；

- 根据键值 key 计算 hash 值得到插入的数组索引 i，如果 tab [i]==null，直接新建节点添加，否则转入 下一步

- 判断当前数组中处理 hash 冲突的方式为链表还是红黑树 (check 第一个节点类型即可), 分别处理

- 最后 ++size，并判断此时的 size 是否大于阈值，大于则需要扩容。

resize 函数伪代码：

- 当数组为 null 的时候，会创建新的数组

- 当数组不为空，会把容量和阈值均 * 2，并创建一个容量为之前二倍的数组，然后把原有数组的数据都转移到新数组。

12. HashMap 和 Hashtable 有什么区别？HashMap 和 HashSet 呢？

HashMap 和 Hashtable 区别：

- hashmap 线程不安全、hashtable 线程安全
- hashmap 继承自 abstractmap、hashtable 继承 dictionary
- hashmap 允许存储 null 键值（存一个 null 键和多个 value 是 null）、hashtable 不允许
- hashtable 直接使用 hashcode，hashmap 用扰动函数处理 hashcode（hashcode 右移 16 位在与原 hashcode 异或操作，为了减少 hashcode 冲突）
- hashmap 初始容量 16，hashtable11
- hashmap 扩容 old*2；hashtable 扩容 old*2+1

hashset：

- 底层实现是 hashmap，因为 hashset 内部元素不能相同，所以存储方式为 hashmap 的 key 值。

- 需要重写 hashcode 函数和 equals 函数，通过 hashcode 直接定位到地址，在通过 equals 比较对象。

- 元素无序

13. final 关键字用于什么场景？

- 字段不可修改，引用不可修改地址

- 方法不能重写

- 类不能继承

包含 final 域的对象的引用和读这个 final 域，不能重排序；构造函数对 final 域的写入和这个对象的引用被赋值，不能重排序。

使用场景：

- 不可改变域

- 多线程使用场景，使用 final 关键字或者：synchronized、volatile、锁

14. ConcurrentHashMap 如何实现线程同步？

- hashmap 的线程安全版，引入 segment，每一个 segment 都是线程安全的，相当于一个 hashtable，因此，ConcurrentHashMap 也不允许出现 null。这样就把整个类锁变成了局部锁，用哪一个 segment 就给哪一个 segment 加锁。减少竞争，提高效率。

对于 jdk1.8 的改进：

- 取消的 segment，转而采用数组元素作为锁。把锁的粒度从多个 node 变成一个 node，进一步减少锁竞争
- 链表大于 8 的时候转化为红黑树

实现线程同步：元素 Node，字段修饰为 final 和 volatile，采用乐观锁 CAS，和分而治之的思想

- put 操作和初始化操作：
  - volatile 字段，标识位，表示当前是否有线程在初始化，volatile 字段保证了所有线程的可见。
  - CAS 机制，保证只有一个线程能够初始化

- size()/判断大小
  - 首先通过 CAS 机制，如果没有线程竞争，直接递增 count，
  - 失败就初始化桶，每一个桶并发的记录（同样是 CAS 机制，最大程度利用并发），如果桶计数频繁失败就扩容桶。

15. Map 遍历的两种方式？

- keyset 和 entryset，前者是获得 key 的集合（），后者是获得 key-value 的集合，返回的都是 set 视图，因为 set 有迭代器 iterator 可以用，通过 iterator.next 来遍历。

- 推荐使用 entrySet () 方法，效率较高。对于 keySet 其实是遍历了 2 次，一次是转为 iterator，一次就是从 HashMap 中取出 key 所对于的 value。而 entryset 只是遍历了第一次，它把 key 和 value 都放到了 entry 中，所以快了。

16. 哪些集合类是线程安全的？

- Vector：就比 Arraylist 多了个同步化机制（线程安全）。
- Hashtable：就比 Hashmap 多了个线程安全。
- ConcurrentHashMap: 是一种高效但是线程安全的集合。
- Stack：栈，也是线程安全的，继承于 Vector。

17. Iterator 怎么使用？有什么特点？Iterator 的 fail-fast 属性是什么？

- Iterator 是一种设计模式，一种轻量级对象，
- Iterator 使用的时候需要集合调用重写的 iterator 方法，返回一个 iterator 对象，通过该对象遍历集合元素。只有 collection 接口继承了 iterator 接口，map 接口没有。
- 正在被迭代的集合对象被删除时，必须使用迭代器的删除方法。
- 所有的 java.util 包中的集合类都被设计为 fail-fast，集合与迭代器一起作用，保证能否安全的删除集合中元素。如果正在被迭代的对象发生结构上的改变时，集合中的两个字段（集合修改次数和迭代器对集合的修改次数）就会不一样，就会报 ConcurrentModificationException 异常。

18. Iterator 和 ListIterator 有什么区别？

- Iterator 是 List 和 Set 的迭代器
- ListIterator 是 List 的迭代器
- Iterator 单向，ListIterator 双向，ListIterator 继承自 Iterator，并且实现了更多功能，添加替换等。

19. Arrays 和 Collections 的常用方法

- Arrsys: 是数组的工具类，提供了对数组操作的工具方法。
- Collections: 是集合对象的工具类，提供了操作集合的工具方法。

其中 Arrays 和 Collections 中所有的方法都为静态的，不需要创建对象，直接使用类名调用即可。

Arrays 常用方法：

- sort() 排序
- asList() 将数组转化为 arraylist
- toString() 返回 数组中元素以逗号分隔，两边是[] 的字符串

Collections 常用方法

- sort() 排序
- reverse() 反转顺序
- synchronizedMap、synchronizedSet、synchronizedList 使线程安全

20. Array 和 ArrayList 有何区别？

- Arrays 是数组的工具类

- Array 是数组，声明时必须说明大小和类型

- ArrayList 是动态数组，不必声明大小，数组大小动态扩展，在不使用泛型的情况下都不必声明类型。

补充：

- 泛型只是在编译期间生效，运行期间会被擦除，保证类型安全，消除了强制类型转换，带来性能上的收益。

21. 在 Queue 中 poll()和 remove()有什么区别？

- poll () 和 remove () 都将移除并且返回对头，但是在 poll () 在队列为空时返回 null，而 remove () 会抛出 NoSuchElementException 异常。

- peek () 和 element () 都将在不移除的情况下返回队头，但是 peek () 方法在队列为空时返回 null，调用 element () 方法会抛出 NoSuchElementException 异常。

- add () 和 offer () 都是向队列中添加一个元素。但是如果想在一个满的队列中加入一个新元素，调用 add () 方法就会抛出一个 unchecked 异常，而调用 offer () 方法会返回 false。

22. 怎么确保一个集合不能被修改？

方式一：final：

- final 修饰的类不能被继承，
- final 修饰的方法不能被重写，
- final 修饰的变量不能修改

这里需要特别注意，变量是基本类型的时候，值不可修改，变量是对象的时候，引用不可修改。集合的值是可以不断添加，但是当引用改变的时候就会报错，所以使用 final 只能让集合的引用不可修改，而值还是可以修改和添加。

方式二：使用 Collections. unmodifiableCollection (Collection c) 方法来创建一个只读集合，这样改变集合的任何操作都会抛出 Java. lang. UnsupportedOperationException 异常。

四、多线程

1. 什么是锁消除？什么是锁粗化？

锁消除：

- 对数据进行逃逸分析。对象实例都是存在于线程共享的堆中的，即便是局部变量的对象，也是存在于堆中，但是局部变量对象的引用是存在于方法栈中的，方法栈是线程私有，线程之间彼此不可见，当对于这样的引用进行加锁和释放锁的时候，其实是没有必要的，因为数据是不会逃逸出去，比如说 StringBuffer 的 append 方法，是 synchronized 修饰的同步方法，虚拟机检测到这类情况不会发生数据逃逸就会在运行的时候消除锁，从而提高性能。

锁粗化：

- 原则上需要尽量缩小加锁的范围，把需要同步的代码的范围尽量缩小，这样其他线程可以尽快的获得锁。

- 但是如果频繁对同一个对象进行加锁和解锁的操作，频繁的同步互斥操作会造成不必要的性能损耗；对于这种情况，需要把加锁提前到对象的第一个操作之前，解锁释放到最后一个对象，例子：StringBuffer 对象变成全部局部，此时变成堆的线程共享。

2. 乐观锁有哪几种？主要思想是什么？

- 乐观锁：偏向锁、轻量级锁、自旋锁

- 乐观锁的主要思想是：CAS（compare and swap）

- CAS 是一种思想，并不是一种锁，它是一种原子操作，通过比较传入的值来判断是否更新，一样则更新，不一样则失败。

3. 多线程锁的升级原理是什么（锁膨胀）？

锁的级别从低到高：

无锁—偏向锁—轻量级锁—重量级锁

- 无锁：不加锁，所有线程都可以对资源进行访问，失败则重试
- 偏向锁：适用于必须同步但是只有一个线程执行的代码块，虚拟机检测到同步的代码块只有一个线程持续执行，就会在后续加上偏向锁。减少非竞争条件的同步代码，提升性能。
- 轻量级锁：只有一个线程执行同步代码块的时候会加上偏向锁，当第二个线程加入竞争，偏向锁就会升级会轻量级锁，第二个线程不会阻塞，而是通过自旋等待第一个线程释放锁。第二个线程自旋（也就是 CPU 空转）可以避免用户态和内核态的转换，提升性能。
- 重量级锁：当第三个线程加入竞争，轻量级锁就会升级为重量级锁，线程会阻塞等待，直到线程释放锁。

4. 什么是死锁？如何预防死锁？死锁和活锁的区别是什么？

死锁：多个线程因竞争资源而造成的互相等待，没有外力作用则一直等待下去。

死锁条件：

- 互斥
- 非抢占
- 请求和保持
- 循环等待

活锁：死锁和活锁的表现都是程序不在运行，但是死锁中所有线程都是出于阻塞状态的，活锁中所有线程都是活的状态，是在运行，不断的 try，但是不在继续执行，在做无用功。

5. 并行和并发有什么区别？同步、异步、阻塞、非阻塞有什么区别？同步等同于阻塞吗？

并行与并发：

- 并行：宏观上和微观上都是多个线程一起执行，需要的是多个处理器
- 并发：微观是多个线程快速轮换从而宏观上看是多个线程一起执行，但是同一时间只有一个线程执行，只需要一个处理器就可以实现。

同步和异步是线程之间的概念，对应的是调用者和被调用者，侧重的是消息通知

- 同步：调用者调用方法之后，只有等待被调用者返回结果才能继续执行。
- 异步：调用者调用方法之后，不需要等待被调用者返回结果才能继续执行。方法完成之后会有相应的通知提示调用者结果返回。

阻塞和非阻塞是线程本身的概念，侧重是本身的状态

- 线程未得到结果返回，会被挂起，结果返回之后才会继续执行。
- 线程未得到结果返回，不会被挂机，依旧继续执行。

综上所述，同步和异步仅仅是关注的消息如何通知的机制，而阻塞与非阻塞关注的是等待消息通知时的状态。

- 同步非阻塞：当前线程 A 需要等待调用方法的的线程 B 返回结果，才会继续执行任务 C，但是未阻塞的线程 A 可以继续执行任务 D。
- 异步阻塞：虽然不必等待结果返回，但是当前线程依旧被挂起

6. 线程和进程的区别？线程有哪些状态？

- 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。

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；对进程进程操作一般开销都比较大，对线程开销就小了
- 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。

线程状态

- 创建：创建一个线程
- 就绪：有执行资格，并未执行
- 运行：正在运行的线程
- 阻塞：sleep 或者 wait 方法，线程被挂起
- 结束

7. 守护线程是什么？

- 线程分为用户线程和守护线程，守护线程是在后台提供通用服务的线程，二者没有本质区别，唯一的区别就是在于虚拟机的结束。当所有用户线程全部结束，那么虚拟机就会结束，即便还有守护线程在运行。new thread 创建的线程都是用户线程，可以调用 setDaemon 方法把用户线程变成守护线程。垃圾回收线程就是一个守护线程。

8. 创建线程有哪几种方式？线程的 run () 和 start () 有什么区别？

创建线程的方式：

- 继承 thread 类，重写 run 方法
- 实现 runnable 接口，重写 run 方法，作为 thread 类的参数
- 实现 callable 接口，仅限线程池

run 方法和 start 方法区别：

- start 方法是启动线程的方法，调用线程的 start 方法之后，线程就会进入就绪状态，
- run 方法是线程实现逻辑的方法，调用线程的 run 方式并不会启动线程，只是相当于普通的方法调用

9. ThreadLocal 是什么？有哪些使用场景？

- ThreadLocal 是线程的局部变量， 是每一个线程所单独持有的，其他线程不能对其进行访问。当使用 ThreadLocal 维护变量的时候 为每一个使用该变量的线程提供一个独立的变量副本，即每个线程内部都会有一个该变量，这样同时多个线程访问该变量并不会彼此相互影响，因此他们使用的都是自己从内存中拷贝过来的变量的副本， 这样就不存在线程安全问题，也不会影响程序的执行性能。最常见的 ThreadLocal 使用场景为：数据库连接、 Session 管理等

- synchronized 对线程共享的数据加锁，让线程以有序的方式访问该数据。ThreadLocal 是为每一个线程在内存中开辟一份私有空间，保证数据安全。synchronized 是以时间换空间，ThreadLocal 是以空间换时间

10. sleep 和 wait 方法有什么区别？notify () 和 notifyAll () 有什么区别？

sleep 方法和 wait 方法的区别

- sleep 是 Thread 中的方法，wait 是 Object 中的方法
- sleep 必须指定时间，wait 可以不指定时间，不指定代表当前线程阻塞到持有锁的线程完成任务
- sleep 可以写在任何地方，wait 只能写在同步代码块中
- sleep 释放 CPU 的使用权，但是不释放锁；wait 释放 CPU 的使用权，同时也释放锁。

notify 和 notifyall 方法

- notify 方法唤醒，正在等待调用对象的锁的阻塞线程，如果有多个这样的线程，就会随机唤醒。
- notifyall，唤醒所有，正在等待调用对象的锁的阻塞线程。

11. 现在有 T1、T2、T3 三个线程，你怎样保证 T2 在 T1 执行完后执行，T3 在 T2 执行完后执行？

- 可以使用 thread 类的 join 方法，

- 使用 thread 的优先级不行，因为优先级不能保证严格按照这样的顺序。

12. volatile 有什么用？synchronized 和 volatile 的区别是什么？

volatile 关键字的作用：

- 内存可见性，修饰的变量发生改变之后对所有线程立即可见
- 禁止指令重排序

volatile 的底层是通过内存屏障实现的，第一个作用是禁止指令重排。内存屏障另一个作用是强制更新一次不同 CPU 的缓存。

synchronized 看作重量级的锁，而 volatile 看作轻量级的锁 。 synchronized 使用的锁的层面是在 JVM 层面， 虚拟机处理字节码文件实现相关指令。 volatile 底层使用多核处理器实现的 lock 指令，更底层，消耗代价更小。

13. synchronized 和 Lock 有什么区别？

- 实现层面不一样。synchronized 是 Java 关键字， JVM 层面 实现加锁和释放锁；Lock

是一个接口，在代码层面实现加锁和释放锁，（但是 Lock 的底层 CAS 乐观锁比 synchronized 更底层，是 CPU 原语，属于操作系统层面的）

- 是否自动释放锁。synchronized 在线程代码执行完或出现异常时自动释放锁；Lock 不会自动释放锁，需要再 finally {} 代码块显式地中释放锁

- 是否一直等待。synchronized 会导致线程拿不到锁一直等待；Lock 可以设置尝试获取锁或者获取锁失败一定时间超时。

- 获取锁成功是否可知。synchronized 无法得知是否获取锁成功；Lock 可以通过 tryLock 获得加锁是否成功

- 功能复杂性。synchronized 加锁可重入、不可判断、非公平；Lock 可重入、可判断、可公平和不公平、细分读写锁提高效率。

14. Lock 实现锁的底层原理？

- Lock 锁的底层实现是 AQS，AQS（AbstractQuenedSynchronizer ），抽象的队列式同步器，除了 java 自带的 synchronized 关键字之外的锁机制。

- AQS 是将每一条请求共享资源的线程封装成一个 CLH 锁队列（该队列是一个双向链表，没有实现）的一个结点（Node），将暂时获取不到锁的线程加入到队列中来实现锁的分配。AQS 基于 CLH 队列，用 volatile 修饰共享变量 state，线程通过 CAS 去改变状态符，成功则获取锁成功，失败则进入等待队列，等待被唤醒。

Lock 的实现过程

- lock 的存储结构：一个 int 类型状态值（用于锁的状态变更），一个双向链表（用于存储等待中的线程）

- lock 获取锁的过程：本质上是通过 CAS 来获取状态值修改，如果当场没获取到，会将该线程放在线程等待链表中。

- lock 释放锁的过程：修改状态值，调整等待链表。

可以看到在整个实现过程中，lock 大量使用 CAS + 自旋。因此根据 CAS 特性，lock 建议使用在低锁冲突的情况下。目前 java1.6 以后，官方对 synchronized 做了大量的锁优化（偏向锁、自旋、轻量级锁）。因此在非必要的情况下，建议使用 synchronized 做同步操作。

15. 说一下 atomic 的原理？

- Atomic 包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进

行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

- Atomic 系列的类中的核心方法都会调用 unsafe 类中的几个本地方法。因此 atomic 证原子性就是通过：自旋 + CAS（乐观锁）

- Lock 类和 Atomic 包底层实现都是通过 CAS + 自旋的方式解决多线程同步问题。Atomic 在竞争激烈时能维持常态，比 lock 性能好，但是只能同步一个变量。

16. 乐观锁的实现方式？

版本号机制

- 一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加一。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

CAS 算法

即 compare and swap（比较与交换），它是一条 CPU 并发原语。是一种原子操作，也是一种乐观锁的实现思想。CAS 算法涉及到三个操作数

- 需要读写的内存值 V

- 进行比较的值 A

- 拟写入的新值 B

当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

17. 说一下 synchronized 底层实现原理？

synchronized 是由一对 monitorenter/monitorexit 指令实现的，monitor 对象是同步的基本实现单元。在 JVM 处理字节码会出现相关指令。

- 代码块的同步：利用 monitorenter 和 monitorexit 这两个字节码指令。它们分别位于同步代码块的开始和结束位置。当 jvm 执行到 monitorenter 指令时，当前线程试图获取 monitor 对象的所有权，如果未加锁或者已经被当前线程所持有，就把锁的计数器 + 1；当执行 monitorexit 指令时，锁计数器 - 1；当锁计数器为 0 时，该锁就被释放了。如果获取 monitor 对象失败，该线程则会进入阻塞状态，直到其他线程释放锁。

- 方法级的同步：是隐式的，即无需通过字节码指令来控制的，它实现在方法调用和返回操作之中。JVM 可以从方法常量池中的方法表结构 (method_info Structure) 中的 ACC_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会 检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有 monitor（虚拟机规范中用的是管程一词），然后再执行方法，最后再方法完成 (无论是正常完成还是非正常完成) 时释放 monitor。

18. 线程池都有哪些状态？

- RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。线程池的初始化状态是 RUNNING。线程池被一旦被创建，就处于 RUNNING 状态，并且线程池中的任务数为 0。

- SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。调用线程池的 shutdown () 方法时，线程池由 RUNNING -> SHUTDOWN。

- STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。调用线程池的 shutdownNow () 方法时，线程池由 (RUNNING or SHUTDOWN) -> STOP。

- TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated ()。因为 terminated () 在 ThreadPoolExecutor 类中是空的，所以用户想在线程池变为 TIDYING 时进行相应的处理；可以通过重载 terminated () 函数来实现。

- TERMINATED：线程池处在 TIDYING 状态时，执行完 terminated () 之后，就会由 TIDYING -> TERMINATED。

注意：

- 当线程池在 SHUTDOWN 状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。

- 当线程池在 STOP 状态下，线程池中执行的任务为空时，就会由 STOP -> TIDYING。

19. 线程池中 submit () 和 execute () 方法有什么区别？

- execute () 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。execute () 方法输入的任务是一个 Runnable 类的实例。

- submit () 方法用于提交需要返回值的任务。线程池会返回一个 future 类型的对象，通过这个 future 对象可以判断任务是否执行成功，并且可以通过 future 的 get () 方法来获取返回值，submit () 方法输入的任务是一个 Callable 类的实例。

20. 创建线程池有哪几种方式？

- 快捷创建线程池很简单，只需要调用 Executors 中相应的静态工厂方法即可
- 使用 ThreadPoolExecutor 创建线程池

21. 创建线程池的各个参数代表的含义？

- corePoolSize（线程池的基本大小）
- runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列
  - 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：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：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 LinkedBlockingQueue，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。
- maximumPoolSize（线程池最大数量）
- ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
- RejectedExecutionHandler（饱和策略 \ 拒绝策略）
  - AbortPolicy：直接抛出异常（默认策略）。
  - CallerRunsPolicy：只用调用者所在线程来运行任务。
  - DiscardOldestPolicy：丢弃队列里最老的一个任务，尝试为当前提交的任务腾出位置 。
  - DiscardPolicy：不处理，丢弃掉。
- liveTime（线程活动保持时间）

五、计算机网络

1. OSI 七层模型？HTTP 协议对应第几层？IP 协议呢？

- 物理层：通过媒介传输比特，确定机械及电气规范（比特 Bit）
- 数据链路层：将比特组装成帧和点到点的传递（帧 Frame）
- 网络层：负责数据包从源到宿的传递和网际互连（包 Packet）
- 传输层：提供端到端的可靠报文传递和错误恢复（段 Segment）
- 会话层：建立、管理和终止会话（会话协议数据单元 SPDU）
- 表示层：对数据进行翻译、加密和压缩（表示协议数据单元 PPDU）
- 应用层：允许访问 OSI 环境的手段（应用协议数据单元 APDU）

http 协议对应的是应用层，ip 协议对应的是网络层

2. 从一个 URL 到获取页面的过程？

- 浏览器查询 DNS，获取域名对应的 IP 地址：具体过程包括浏览器搜索自身的 DNS 缓存、搜索操作系统的 DNS 缓存、读取本地的 Host 文件和向本地 DNS 服务器进行查询等。
- 浏览器获得域名对应的 IP 地址以后，浏览器向服务器请求建立链接，发起 TCP 三次握手；
- TCP/IP 链接建立起来后，浏览器向服务器发送 HTTP 请求；
- 服务器接收到这个请求，并根据路径参数映射到特定的请求处理器进行处理，并将处理结果及相应的视图返回给浏览器；
- 浏览器解析并渲染视图，若遇到对 js 文件、css 文件及图片等静态资源的引用，则重复上述步骤并向服务器请求这些资源；浏览器根据其请求到的资源、数据渲染页面，最终向用户呈现一个完整的页面。

3. session 的实现原理？cookie 的原理？

- Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器。服务器检查该 Cookie，以此来辨认用户状态。服务器还可以根据需要修改

Cookie 的内容。

- 如果说 Cookie 机制是通过检查客户身上的“通行证”来确定客户身份的话，那么 Session 机制就是通过检查服务器上的“客户明细表”来确认客户身份。Session 相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。使用上比 Cookie 简单一些，相应的也增加了服务器的存储压力。

- Cookie 通过在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份。

4. session 和 cookie 的关系？禁用 cookie 后对 session 的影响？

Session 的实现常常依赖于 Cookie 机制。一般默认情况下，在会话中，服务器存储 session 的 sessionid 是通过 cookie 存到浏览器里。如果浏览器禁用了 cookie，浏览器请求服务器无法携带 sessionid，服务器无法识别请求中的用户身份，session 失效。但是可以通过其他方法在禁用 cookie 的情况下，可以继续使用 session。

- 通过 url 重写，把 sessionid 作为参数追加的原 url 中，后续的浏览器与服务器交互中携带 sessionid 参数。

- 服务器的返回数据中包含 sessionid，浏览器发送请求时，携带 sessionid 参数。

- 通过 Http 协议其他 header 字段，服务器每次返回时设置该 header 字段信息，浏览器中 js 读取该 header 字段，请求服务器时，js 设置携带该 header 字段。

5. forward 和 redirect 的区别？

- forward 是服务器请求资源，服务器直接访问目标地址的 URL, 把那个 URL 的响应内容读取过来，然后把这些内容再发给浏览器。浏览器根本不知道服务器发送的内容从哪里来的，所以它的地址栏还是原来的地址.

- redirect 是服务端根据逻辑，发送一个状态码 ，告诉浏览器重新去请求那个地址。所以地址栏显示的是新的 URL.

6. 内网和外网 IP 地址的区别？ABC 三类 IP 地址的划分

- 外网 IP 是全世界唯一的 IP 地址，仅分配给一个网络设备。公网 IP 地址全世界仅分配给一个网络设备（比如你在家拨号，分配给你一个 IP 地址吧，那个地址是唯一的，你用你机器做个网站，别人访问你的 IP 地址就可以连接到你的机器）

- 内网 IP 局域网，网线都是连接在同一个 交换机上面的，也就是说它们的 IP 地址是由

交换机或者路由器进行分配的。内网用户的电脑都是经过交换机和路由器之后才能连到外网。Internet 上的用户也无法直接访问到内网用户。不同内网的内网地址可以相同。

IP 地址 = 网络地址 + 主机地址。A 类前 8 位是（0+7 位网络地址）,B 类前 16 位是（10+14 位网络地址），C 类前 24 位是（110+21 位网络地址）。

- A 类地址：以 0 开头， 第一个字节范围：0~127（1.0.0.0 – 126.255.255.255）；
- B 类地址：以 10 开头， 第一个字节范围：128~191（128.0.0.0 – 191.255.255.255）；
- C 类地址：以 110 开头， 第一个字节范围：192~223（192.0.0.0 – 223.255.255.255）；

7. 网关和子网掩码的关系是什么？

- 子网掩码：用来判断任意两台计算机的 ip 地址是否属于同一子网络的根据
- 网关实质上是一个在不同子段网路中传输数据的设备。比如有网络 A 和网络 B，若二者子网掩码不同，即二者不属于同一个子网络。在没有路由器的情况下，两个网络之间是不能进行 TCP/IP 通信的
- 子网掩码相同，不需要网关即可通讯，子网掩码不同，需要网关才能通讯

8. MAC 地址和 IP 地址的关系是什么？

- MAC 地址是硬件地址，定位全球唯一主机机器，在网络底层的物理传输过程中，是通过物理地址来识别主机的，它一定是全球唯一的，应数据链路层。
- IP 地址是网络拓扑地址，定位全球唯一网络结构中的主机。对应网路层
- 可以通过身份证（MAC）和电话（IP）来理解

9. 什么是 DNS 服务器？

- DNS 是指：域名服务器 (Domain Name Server)。在 Internet 上域名与 IP 地址之间是一一对应的，域名虽然便于人们记忆，但机器之间只能互相认识 IP 地址，它们之间的转换工作称为域名解析，域名解析需要由专门的域名解析服务器来完成，DNS 就是进行域名解析的服务器 。

10. IP 如何映射到 MAC 地址的？

使用的是 ARP （Address Resolution Protocol：地址解析协议），ARP 协议位于网络层。

工作原理：

- 首先，每个主机都会在自己的 ARP 缓冲区中建立一个 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 ARP 列表中是否有对应 IP 地址的目的主机的 MAC 地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送 ARP 数据包。

- 当本网络的所有主机收到该 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。

- 如果目标 IP 与自己不在同一个网段，这种情况需要将包发给默认网关，所以主要获取网关的 MAC 地址。

11. TCP 是如何保证可靠传输数据的？

保证可靠稳定的传输最主要是通过：
- 拥塞控制
- 流量控制
- ARQ 协议

除此之外还有：超时传送、丢弃重复、校验和、分割合适数据包

拥塞控制：
- 慢启动：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小；
- 拥塞避免：拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍，这样拥塞窗口按线性规律缓慢增长。
- 快重传：快重传要求接收方在收到一个 失序的报文段 后就立即发出 重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。
- 快恢复：快重传配合使用的还有快恢复算法，当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半，但是接下去并不执行慢开始算法：因为如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。

流量控制
- TCP 连接的每一方都有固定大小的缓冲空间，TCP 的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。（TCP 利用滑动窗口实现流量控制）

ARQ 协议

- 停止等待 ARQ 协议：停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认（回复 ACK）。如果过了一段时间（超时时间后），还是没有收到 ACK 确认，说明没有发送成功，需要重新发送，直到收到确认后再发下一个分组；在停止等待协议中，若接收方收到重复分组，就丢弃该分组，但同时还要发送确认；
- 连续 ARQ 协议：连续 ARQ 协议可提高信道利用率。发送方维持一个发送窗口，凡位于发送窗口内的分组可以连续发送出去，而不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了。

12. TCP 和 UDP 的区别？

- TCP：面向连接、面向字节流、一对一； UDP：无连接、面向报文、一对一、一对多、多对多。
- TCP 的优点：可靠，稳定.
- TCP 的缺点：慢，效率低，占用系统资源高，易被攻击
- UDP 的优点：快，比 TCP 稍安全 UDP 没有 TCP 的握手、确认、窗口、重传、拥塞控制等机制，UDP 是一个无状态的传输协议
- UDP 的缺点： 不可靠，不稳定

13. TCP 三次握手和四次挥手的过程？

三次握手：(我要和你建立链接，你真的要和我建立链接么，我真的要和你建立链接，成功)

- 第一次握手：客户端发送 syn 包 (syn=x) 到服务器，并进入 SYN_SEND 状态，等待服务器确认；
- 第二次握手：服务器收到 syn 包，必须确认客户的 ACK（ack=x+1），同时自己也发送一个 SYN 包（syn=y），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；
- 第三次握手：客户端收到服务器的 SYN＋ACK 包，向服务器发送确认包 ACK (ack=y+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

四次挥手：(我要和你断开链接；好的，断吧。我也要和你断开链接；好的，断吧)

- 第一次挥手：客户端主动关闭方发送一个 FIN，用来关闭客户端到服务端的数据传送，也就是客户端告诉服务端：我已经不会再给你发数据了 (当然，在 fin 包之前发送出去

的数据，如果没有收到对应的 ack 确认报文，客户端依然会重发这些数据)，但是，此时客户端还可以接受数据。

- 第二次挥手：服务端收到 FIN 包后，发送一个 ACK 给客户端，确认序号为收到序号 + 1（与 SYN 相同，一个 FIN 占用一个序号）。

- 第三次挥手：服务端发送一个 FIN，用来关闭服务端到客户端的数据传送，也就是告诉客户端，我的数据也发送完了，不会再给你发数据了。

- 第四次挥手：客户端收到 FIN 后，发送一个 ACK 给服务端，确认序号为收到序号 + 1，至此，完成四次挥手。

14. TCP 为什么需要三次握手？只进行两次会出现什么问题？

- 客户端发出的连接请求报文并未丢失，而是在某个网络节点长时间滞留了，以致延误到链接释放以后的某个时间才到达 Server。这是，Server 误以为这是 Client 发出的一个新的链接请求，于是就向客户端发送确认数据包，同意建立链接。若不采用“三次握手”，那么只要 Server 发出确认数据包，新的链接就建立了。由于 client 此时并未发出建立链接的请求，所以其不会理睬 Server 的确认，也不与 Server 通信；而这时 Server 一直在等待 Client 的请求，这样 Server 就白白浪费了一定的资源。若采用“三次握手”，在这种情况下，由于 Server 端没有收到来自客户端的确认，则就会知道 Client 并没有要求建立请求，就不会建立链接。

15. TCP 第三次握手失败的情况 TCP 是如何处理的？

第三次失败，只有客户端处于成功状态（因为第 2 次服务器返回了 ACK），服务器端没有接收到客户端的 ACK。这要分几种情况讨论：

- 客户端发出的 ACK 丢失了，发出的 下一个数据包 没有丢失，则服务端接收到下一个数据包（这个数据包里也会带上 ACK 信息），能够进入正常的 ESTABLISHED 状态

- 如果服务端和客户端都没有数据发送，或者服务端想发送数据（但是发不了，因为没有收到客户端的 ACK），服务器都会有定时器发送第二步 SYN+ACK 数据包，如果客户端再次发送 ACK 成功，建立连接。

- 如果一直不成功，服务器肯定会有超时设置，超时之后会给客户端发 RTS 报文，进入 CLOSED 状态，防止 SYN 洪泛攻击。

16. 为什么连接的时候是三次握手，关闭的时候却是四次握手？

- TCP 是全双工模式，关闭连接时，当主机 B 收到主机 A 的 FIN 报文时，仅仅表示主机 A 不再发送数据了但是还能接收数据。此时，主机 B 也未必全部数据都发送给 A

了，所以 B 可以立即 close；也可以发送一些数据给 A 后，再发送 FIN 报文给对方来表示同意现在关闭连接，因此，主机 BACK 和 FIN 一般都会分开发送。

17. http1.0 与 http1.1 的区别？什么是 keep-alive 模式？

- 缓存处理，在 HTTP1.0 中主要使用 header 里的 If-Modified-Since,Expires 来做为缓存判断的标准，HTTP1.1 则引入了更多的缓存控制策略例如 Entity tag，If-Unmodified-Since, If-Match, If-None-Match 等更多可供选择的缓存头来控制缓存策略。

- 带宽优化及网络连接的使用，HTTP1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

- 错误通知的管理，在 HTTP1.1 中新增了 24 个错误状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

- Host 头处理，在 HTTP1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。HTTP1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有 Host 头域会报告一个错误（400 Bad Request）。

- 长连接，HTTP 1.1 支持长连接（Persistent-Connection）和请求的流水线（Pipelining）处理，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟，在 HTTP1.1 中默认开启 Connection： keep-alive，一定程度上弥补了 HTTP1.0 每次请求都要创建连接的缺点。

Keep-Alive 模式

- 当使用 Keep-Alive 模式（又称持久连接、连接重用）时，Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接。启用 Keep-Alive 模式肯定更高效，性能更高。因为避免了建立 / 释放连接的开销。因此最好能维持一个长连接，可以用一个长连接来发多个请求。

18. 简单说一下 http2.0？

- HTTP2.0 使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比 HTTP1.1 大了好几个数量级。

- 当然 HTTP1.1 也可以多建立几个 TCP 连接，来支持处理更多并发的请求，但是创建

TCP 连接本身也是有开销的。TCP 连接有一个预热和保护的过程，先检查数据是否传送成功，一旦成功过，则慢慢加大传输速度。因此对应瞬时并发的连接，服务器的响应就会变慢。所以最好能使用一个建立好的连接，并且这个连接可以支持瞬时并发的请求。

19. 什么是幂等性？http 的方法是否都符合幂等性？若不符合，怎么避免？

- 幂等性：HTTP 方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。说白了就是，同一个请求，发送一次和发送 N 次效果是一样的！

HTTP 方法：

- GET 方法用于获取资源，不应有副作用，所以是幂等的。
- DELETE 方法用于删除资源，有副作用，但它应该满足幂等性。
- PUT 方法用于创建或更新操作，有副作用，与 DELETE 相同，对同一资源无论调用一次还是多次，其副作用是相同的，因此也满足幂等性。
- POST 方法与 PUT 方法的区别主要在于幂等性，POST 不具备幂等性，因为 POST 请求每次都会创建一个文件，而 PUT 方法会在服务器验证是否有 ENTITY，若有则更新该 ENTITY 而不是重新创建。

POST 方法避免非幂等性，当我们因为反复刷新浏览器导致多次提交表单，多次发出同样的 POST 请求，导致远端服务器重复创建出了资源。

- 第一、对应的后端 WebService 一定要做到幂等性，
- 第二、服务器端收到 POST 请求，在操作成功后必须返回状态码 302 重定向到另外一个页面，这样即使用户刷新页面，url 已经变更，不需要进行 post 请求，也不会重复提交表单。

20. https 与 http 的区别？

Http 协议运行在 TCP 之上，明文传输，客户端与服务器端都无法验证对方的身份； HTTPS 是运行在 SSL/TLS 之上的 HTTP 协议，SSL/TLS 运行在 TCP 之上。HTTPS 是添加了加密和认证机制的 HTTP。二者之间存在如下不同：

- 端口不同：Http 与 Https 使用不同的连接方式，用的端口也不一样，前者是 80，后者是 443；
- 资源消耗：和 HTTP 通信相比，Https 通信会由于加减密处理消耗更多的 CPU 和内存资源；

- 开销：Https 通信需要证书，而证书一般需要向认证机构购买

21. https 加密的过程？

- 浏览器使用 Https 的 URL 访问服务器，建立 SSL 链接。
- 服务器接收到 SSL 链接后，发送非对称加密的公钥 A 给浏览器。
- 浏览器生成随机数，作为对称加密的密钥 B。
- 浏览器使用服务器返回的公钥 A，对自己生成的对称加密密钥 B 进行加密，得到密钥 C。
- 浏览器将密钥 C 发送给服务器
- 服务器使用自己的非对称加密私钥 D 对接受的密钥 C 进行解密，得到对称加密密钥 B。
- 浏览器和服务器之间使用密钥 B 作为对称加密密钥进行通信。

注意：非对称加密之所以不安全，因为客户端不知道这把公钥是不是属于服务器的。

22. https 是否存在安全问题？如何避免？

当服务器发送公钥给客户端， 中间人截获公钥 ，将 中间人自己的公钥 冒充服务器的公钥发送给客户端。之后客户端会用 中间人的的公钥 来加密自己生成的 对称密钥 。然后把加密的密钥发送给服务器，这时中间人又把密钥截取，中间人用自己的私钥把加密的密钥进行解密，解密后中间人就能获取 对称加密的密钥 。

避免方式：

- 一个拥有公信力、大家都认可的认证中心，数字证书认证机构。
- 服务器在给客户端传输公钥的过程中，会把公钥和服务器的个人信 息通过 hash 算法生成 信息摘要 。
- 为了 防止信息摘要被调换 ，服务器会采用 CA 提供的私钥 对信息摘要进行加密来形成 数字签名 。最后会把原来没 Hash 算法之前的个人信息、公钥及、数字签名合并在一起，形成数字证书。
- 客户端拿到 数字证书 之后，使用 CA 提供的公钥对数字证书里的数字签名进行解密

来得到信息摘要，然后对数字证书里服务器的公钥及个人信息进行 Hash 得到 另一份信息摘要 。最后将两份信息摘要对比，如果一样则证明是服务器，否则就是中间人。

23. get 方法和 post 方法的区别？

- GET 方法从服务器获取资源，POST 是向服务器发送数据
- GET 浏览器回退是无害的，而 POST 会再次提交请求。
- GET 产生的 URL 地址可以被书签收藏，并且被浏览器缓存，而 POST 不能书签收藏也不能缓存。
- GET 只能进行 URL 编码，而 POST 支持多种编码方式。
- GET 参数通过 URL 传递，并且长度有限制，而 POST 放在 request body 并且长度没有限制。并且，正因为这个原因， GET 比 POST 更不安全，因为参数暴露在 URL 中。

二者还有一个显著区别：GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包。

- 对于 GET 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应 200（返回数据）；
- 而对于 POST，浏览器先发送 header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）。

24. 什么 XSS 攻击？如何预防？

跨站脚本攻击

- 存储型 XSS，也叫持久型 XSS，主要是将 XSS 代码发送到服务器（不管是数据库、内存还是文件系统等。），然后在下次请求页面的时候就不用带上 XSS 代码了。用户输入的带有恶意脚本的数据存储在服务器端。当浏览器请求数据时，服务器返回脚本并执行。最典型的就是留言板 XSS。
- 反射型 XSS，也叫非持久型 XSS，把用户输入的数据“反射”给浏览器。通常是，用户点击链接或提交表单时，攻击者向用户访问的网站注入恶意脚本。XSS 代码出现在请求 URL 中，作为参数提交到服务器，服务器解析并响应。响应结果中包含 XSS 代码，最后浏览器解析并执行。从概念上可以看出，反射型 XSS 代码是首先出现在 URL 中的，然后需要服务端解析，最后需要浏览器解析之后 XSS 代码才能够攻击。

预防：

- 内容安全策略 (CSP)
  - 入参字符过滤
  - 出参进行编码
  - 入参长度限制

- HttpOnly 阻止 Cookie 劫持攻击，服务器端 Set-Cookie 字段设置 HttpOnly 参数，这样可以避免 Cookie 劫持攻击。这时候，客户端的 Document.cookie API 无法访问带有 HttpOnly 标记的 Cookie，但可以设置 cookie。

25. 什么是 CSRF 攻击？如何预防？

跨站请求伪造：CSRF 就是利用用户的登录态发起恶意请求，借助用户的 Cookie 骗取服务器的信任。

如何预防

- Referer Check，在 HTTP 头中有一个字段叫做 Referer，它记录了该 HTTP 请求的来源地址。
- token 验证：在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，若请求无 token 或者 token 不正确，则认为可能是 CSRF 攻击而拒绝该请求。
- 验证码：验证码会强制用户必须与应用进行交互，才能完成最终请求

26. 什么是 DDoS 攻击？如何预防

DDoS 攻击：全称 Distributed Denial of Service，中文意思为“分布式拒绝服务”，就是利用大量合法的分布式服务器对目标发送请求，从而导致正常合法用户无法获得服务。通俗点讲就是利用网络节点资源如：IDC 服务器、个人 PC、手机、智能设备、打印机、摄像头等对目标发起大量攻击请求，从而导致服务器拥塞而无法对外提供正常服务，只能宣布 game over。

- 资源消耗类攻击：资源消耗类是比较典型的 DDoS 攻击，最具代表性的包括：Syn Flood、Ack Flood、UDP Flood。这类攻击的目标很简单，就是通过大量请求消耗正常的带宽和协议栈处理资源的能力，从而达到服务端无法正常工作的目的。

- 服务消耗性攻击：相比资源消耗类攻击，服务消耗类攻击不需要太大的流量，它主要是针对服务的特点进行精确定点打击，如 web 的 CC 攻击，数据服务的检索，文件服务的下载等。这类攻击往往不是为了拥塞流量通道或协议处理通道，它们是让服务端始终处理高消耗型的业务的忙碌状态，进而无法对正常业务进行响应

DDoS 的防护系统本质上是一个基于资源较量和规则过滤的智能化系统

- 资源隔离：资源隔离可以看作是用户服务的一堵防护盾。

- 用户规则：根据流量类型、请求频率、数据包特征、正常业务之间的延时间隔等，判断用户是否为攻击行为。

- 大数据智能分析

27. 什么是 SQL 注入攻击？如何预防？

SQL 注入（SQLi）是一种注入攻击，它通过将任意 SQL 代码插入数据库查询，使攻击者能够完全控制 Web 应用程序后面的数据库服务器。攻击者可以使用 SQL 注入漏洞绕过应用程序安全措施；可以绕过网页或 Web 应用程序的身份验证和授权，并检索整个 SQL 数据库的内容；还可以使用 SQL 注入来添加，修改和删除数据库中的记录。

- 带内注入：依赖于攻击者修改应用程序发送的 SQL，以及浏览器中显示的错误和返回的信息。
- 盲注入：推理 SQL 注入（根据：HTTP 响应中的详细信息，某些用户输入的空白网页以及数据库响应某些用户输入需要多长时间）
- 带外注入：攻击者会制作 SQL 语句，这些语句在呈现给数据库时会触发数据库系统创建与攻击者控制的外部服务器的连接。

预防 SQL 注入：

- 不要使用动态 SQL
- 不要将敏感数据保留在纯文本中
- 限制数据库权限和特权
- 避免直接向用户显示数据库错误
- 对访问数据库的 Web 应用程序使用 Web 应用程序防火墙（WAF）
- 将数据库更新为最新的可用修补程序

六、数据库

1. 什么是超键？什么是主键？二者有什么关系？

- 超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

- 候选键：最小超键，即没有冗余元素的超键。

- 主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

- 只能能唯一标识元组的属性集就是超键，所以，超键不唯一，并且，超键可以是一个属性也可以是多个属性，同时候选键是最少的超键，那么候选键也是不唯一的，但是一个元组，主键是唯一的，是从候选键中选择一个作为主键。

2. 数据库的三范式是什么？

- 第一范式：(确保每列保持原子性) 所有字段值都是不可分解的原子值。

- 第二范式：(确保表中的每列都和主键相关) 在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

- 第三范式：(确保每列都和主键列直接相关，而不是间接相关) 数据表中的每一列数据都和主键直接相关，而不能间接相关。

3. char 和 varchar 的区别是什么？

- char：存储定长数据很方便，CHAR 字段上的索引效率级高，必须在括号里定义长度，可以有默认值，比如定义 char (10)，那么不论你存储的数据是否达到了 10 个字节，都要占去 10 个字节的空间（自动用空格填充），且在检索的时候后面的空格会隐藏掉，所以检索出来的数据需要记得用 trim 函数去过滤空格。

- varchar：存储变长数据，但存储效率没有 CHAR 高，必须在括号里定义长度，可以有默认值。保存数据的时候，不进行空格自动填充，而且如果数据存在空格时，当值保存和检索时尾部的空格仍会保留。另外，varchar 类型的实际长度是它的值的实际长度 + 1，这一个字节用于保存实际使用了多大的长度。

4. delete 和 truncate 有什么区别？谁效率更好？

- DELETE 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。

- TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

- 当表被 TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而 DELETE 操作不会减少表或索引所占用的空间。 drop 语句将表所占用的空间全释放掉。TRUNCATE 效率更高一点。

5. 存储过程和函数的区别？

- 存储过程是一个预编译的 SQL 语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。函数每次执行都需要编译一次。

- 存储过程中可以使用 try-catch 块和事务，而函数中不可以

- 函数有且只有一个输入参数和一个返回值，而存储过程没有这个限制

- 函数可以被存储过程调用而存储过程不可以被函数调用

6. 视图的操作会对基本表产生影响吗？

- 视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。对视图的修改会影响基本表。它使得我们获取数据更容易，相比多表查询。
- 当用户试图修改视图的某些信息时，数据库必须把它转化为对基本表的某些信息的修改，对于简单的视图来说，这是很方便的，但是，对于比较复杂的试图，可能是不可修改的。

7. count（*）和 count（列名）谁的效率更高？

- count (*) 对行的数目进行计算，包含 NULL

- count (column) 对特定的列的值具有的行数进行计算，不包含 NULL 值。

- count (1) 这个用法和 count (*) 的结果是一样的。

- count (1) 和 count (*) 效率差不多，二者都快于 count (column)

8. 索引是什么？

- 在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。
- 索引的底层结构是 B 树、B + 树和 hash 结构。

9. 索引的分类？索引失效条件？

分类：

- 主键索引 （PRIMARY KEY）
- 唯一索引 （UNIQUE）
- 普通索引 （INDEX）
- 组合索引 （INDEX）
- 全文索引 （FULLTEXT）

索引失效条件：

- 在 where 子句中使用！= 或 <> 操作符
- 在 where 子句中使用 or 来连接条件，当连接的字段有字段没有索引时，将导致所有字段的索引失效
- 在 where 子句字段进行 null 值判断，
- 在 where 子句中 like 的模糊匹配以 % 开头
- 在 where 子句中对有索引的字段进行表达式或函数操作
- 如果执行引擎估计使用全表扫描要比使用索引快，则不使用索引

10. 索引优化方式？

创建：

- 在经常需要搜索的列上，可以加快搜索的速度；
- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；
- 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
- 在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度。

避免创建：

- 对于那些在查询中很少使用或者参考的列不应该创建索引。

- 对于那些只有很少数据值的列也不应该增加索引。
- 对于那些定义为 text, image 和 bit 这种数据量很大的数据类型的列不应该增加索引。
- 当修改性能远远大于检索性能时，不应该创建索引

11. 怎么验证 mysql 的索引是否满足需求？

用数据库再带的命令 explain 查看语句中索引是否启动，

- type: 主要衡量该检索的性能（all 代表全表扫描，index 代表全索引）
- key: 显示 Mysql 实际决定使用的键（索引），null 代表未走索引。

12. 索引的底层结构是什么？说说各种的特点和缺点？

B 树的定义：

- 根节点至少有 2 个孩子，至多有 m 个孩子。
- 除了根节点以外，所有内部节点至少有 m/ 2（向上取整）个孩子，至多有 m 个孩子。
- 节点内部关键字 = 孩子数 - 1，并且内部关键字是有序的。
- 所有外部节点位于同一层上。

B 树和 B+树区别：

- B 树，每个节点都存储 key 和 data，所有节点组成这棵树，并且叶子节点指针为 null，叶子结点不包含任何关键字信息。
- B + 树，所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接，所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。(而 B 树的非终节点也包含需要查找的有效信息)
- Hash 索引无法范围查询，无法模糊查询，无法排序操作，不支持联合索引最左匹配，无法避免表扫描。但是等值查询具有极高效率。
- B + 的磁盘读写代价更低，B+tree 的查询效率更加稳定

13. 什么是事务？

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

14. 事务的 ACID 特性？

- 原子性（Atomicity）：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

- 一致性（Consistency）：事务开始前和结束后，数据库的完整性约束没有被破坏。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

- 隔离性（Isolation）：隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

- 持久性（Durability）：持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

15. 事务并发会造成的问题？

- 脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据

- 不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果因此本事务先后两次读到的数据因更新结果会不一致。不可重复读的重点是修改。同样的条件， 你读取过的数据， 再次读取出来发现值不一样了

- 幻读：幻读发生在当两个完全相同的查询执行时，第二次查询所返回的结果集跟第一个查询不相同。幻读的重点在于新增或者删除。同样的条件， 第 1 次和第 2 次读出来的记录数不一样

16. 事务的隔离级别？

- Read uncommitted：读未提交，顾名思义，就是一个事务可以读取另一个未提交事务的数据。会造成脏读。

- Read committed：读提交，顾名思义，就是一个事务要等另一个事务提交后才能读取数据。若有事务对数据进行更新（UPDATE）操作时，读操作事务要等待这个更新操作事务提交后才能读取数据，可以解决脏读问题。但会造成不可重复读。

- Repeatable read：重复读，就是在开始读取数据（事务开启）时，不再允许修改操作在同一个事务里，SELECT 的结果是事务开始时时间点的状态，因此，同样的 SELECT 操作读到的结果会是一致的。但是，会有幻读现象。

- Serializable 序列化，Serializable 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

17. 说一下乐观锁和悲观锁？说一下 mysql 的行锁和表锁？

- 悲观锁: 先获取锁，再进行业务操作。即 “悲观” 的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。通常来讲在数据库上的悲观锁需要数据库本身提供支持，即通过常用的 select … for update 操作来实现悲观锁。

- 乐观锁的特点先进行业务操作，不到万不得已不去拿锁。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，那么当前正在提交的事务会进行回滚。

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。

- 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

行锁又分为读锁和写锁：

- 共享锁：也称读锁或 S 锁。如果事务 T 对数据 A 加上共享锁后，则其他事务只能对 A 再加共享锁，不能加排它锁。获准共享锁的事务只能读数据，不能修改数据。

- 排它锁：也称独占锁、写锁或 X 锁。如果事务 T 对数据 A 加上排它锁后，则其他事务不能再对 A 加任何类型的锁。获得排它锁的事务即能读数据又能修改数据。

18. 事务的隔离级别和加锁的关系？

- 读未提交
- 事务读不阻塞其他事务读和写，事务写阻塞其他事务写但不阻塞读。
- 可以通过写操作加 “持续 - X 锁” 实现。

- 读已提交
- 事务读不会阻塞其他事务读和写，事务写会阻塞其他事务读和写。
- 可以通过写操作加 “持续 -X” 锁，读操作加 “临时 -S 锁” 实现。

- 可重复读
- 事务读会阻塞其他事务事务写但不阻塞读，事务写会阻塞其他事务读和写。
- 可以通过写操作加 “持续 -X” 锁，读操作加 “持续 -S 锁” 实现。

- 串行化
- “行级锁” 做不到，需使用 “表级锁”。

19. 两种常见的数据库引擎？分别具有什么特点？

InnoDB 和 MyISAM 是数据库最常见的两种存储引擎。

- InnoDB 支持事务，支持行锁（读写锁），适合频繁性修改操作和需要安全性的应用；MyISAM 不支持行锁，不支持事务，适合查询和插入的应用。因为行锁的读写锁只能适用于修改，插入和删除都是表锁。

- 如果你的应用程序对查询性能要求较高，就要使用 MyISAM 了。MyISAM 的性能更优，占用的存储空间少

- 如果你的应用程序一定要使用事务，毫无疑问你要选择 INNODB 引擎。但要注意，INNODB 的行级锁是有条件的。在 where 条件没有使用主键时，照样会锁全表。比如 DELETE FROM mytable 这样的删除语句。

- 现在一般都是选用 innodb 了，主要是 MyISAM 的全表锁，读写串行问题，并发效率锁表，效率低，MyISAM 对于读写密集型应用一般是不会去选用的。

20. 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 MySQL 数据库，又插入了一条数据，此时 id 是几？

- 一般情况下，我们创建的数据库表引擎是 InnoDB，如果新增一条记录（不重启 mysql 的情况下），这条记录的 id 是 8；但是如果重启（文中提到的）MySQL 的话，这条记录的 ID 是 6。因为 InnoDB 表只把自增主键的最大 ID 记录到内存中，所以重启数据库或者对表 OPTIMIZE 操作，都会使最大 ID 丢失。

- 但是，如果我们使用表的类型是 MyISAM，那么这条记录的 ID 就是 8。因为 MyISAM 表会把自增主键的最大 ID 记录到数据文件里面，重启 MYSQL 后，自增主键的最大 ID 也不会丢失。

- 注：如果在这 7 条记录里面删除的是中间的几个记录（比如删除的是 3,4 两条记录），重启 MySQL 数据库后，insert 一条记录后，ID 都是 8。因为内存或者数据库文件存储都是自增主键最大 ID

21. 什么是主从复制？什么是读写分离？

在实际的生产环境中，对数据库的读和写都在同一个数据库服务器中，是不能满足实际需求的。无论是在安全性、高可用性还是高并发等各个方面都是完全不能满足实际需求的。因此，通过主从复制的方式来同步数据，再通过读写分离来提升数据库的并发负载能力。

主从复制的过程：

- 在每个事务更新数据完成之前，master 在二进制日志记录这些改变。写入二进制日志完成后，master 通知存储引擎提交事务。

- Slave 将 master 的 binary log 复制到其中继日志。首先 slave 开始一个工作线程（I/O），I/O 线程在 master 上打开一个普通的连接，然后开始 binlog dump process。binlog dump process 从 master 的二进制日志中读取事件，如果已经跟上 master，它会睡眠并等待 master 产生新的事件，I/O 线程将这些事件写入中继日志。

- Sql slave thread（sql 从线程）处理该过程的最后一步，sql 线程从中继日志读取事件，并重放其中的事件而更新 slave 数据，使其与 master 中的数据一致，只要该线程与 I/O 线程保持一致，中继日志通常会位于 os 缓存中，所以中继日志的开销很小。

22. 数据库从哪几方面进行调优？

- 数据库设计 — 三大范式、字段、表结构
- 数据库索引
- 存储过程 (模块化编程，可以提高速度)
- 分表分库 (水平分割，垂直分割)
- 主从复制、读写分离
- SQL 调优

23. 索引优化方向？

一般来说，应该在这些列上创建索引：

- 在经常需要搜索的列上，可以加快搜索的速度；
- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；
- 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连

续的；

- 在经常需要排序的列（group by 或者 order by）上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
- 在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度。
- 总结就是：唯一、不为空、经常被查询的字段。

对于有些列不应该创建索引：

- 对于那些在查询中很少使用或者参考的列不应该创建索引。
- 对于那些只有很少数据值的列也不应该增加索引。
- 对于那些定义为 text, image 和 bit 这种数据量很大的数据类型的列不应该增加索引。
- 当修改性能远远大于检索性能时，不应该创建索引。修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

索引失效：

在以下这些情况种，执行引擎将放弃使用索引而进行全表扫描

- 在 where 子句中使用！= 或 <> 操作符
- 在 where 子句中使用 or 来连接条件，当连接的字段有字段没有索引时，将导致所有字段的索引失效
- 在 where 子句字段进行 null 值判断，
- 在 where 子句中 like 的模糊匹配以 % 开头
- 在 where 子句中对索引进行表达式运算或函数操作
- 如果执行引擎估计使用全表扫描要比使用索引快，则不使用索引

24. mysql 问题排查都有哪些手段？怎么验证 mysql 的索引是否满足需求？

SQL 调优最常见的方式是，由自带的慢查询日志或者开源的慢查询系统定位到具体的出问题的 SQL，然后使用 explain、profile 等工具来逐步调优，最后经过测试达到效果后上线。

开启慢查询

- slow_query_log 慢查询开启状态。
- slow_query_log_file 慢查询日志存放的位置（这个目录需要 MySQL 的运行帐号的可写权限，一般设置为 MySQL 的数据存放目录）。
- long_query_time 查询超过多少秒才记录。

分析慢查询 explain

在该语句之前加上 explain 再次执行，explain 会在查询上设置一个标志，当执行查询时，这个标志会使其返回关于在执行计划中每一步的信息，而不是执行该语句。

explain 通常用于查看索引是否生效， explain 执行语句返回的重要字段：

- type：显示是搜索方式（全表扫描 all 或者索引扫描 index）
- key：使用的索引字段，未使用则是 null

七、操作系统

1. 什么是僵尸进程？什么是孤儿进程？有什么危害？

僵尸进程：

- 一个进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或者 waitpid 获取子进程的状态信息，那么子进程的进程描述符等一系列信息还会保存在系统中。这种进程称之为僵尸进程。
- “僵尸”进程是一个早已死亡的进程，但在进程表（processs table）中仍占了一个位置（slot）。由于进程表的容量是有限的，所以，defunct 进程不仅占用系统的内存资源，影响系统的性能，而且如果其数目太多，还会导致系统瘫痪。

处理僵尸进程：

- 改写父进程，在子进程死后要为它收尸。具体做法是接管 SIGCHLD 信号。子进程死后，会发送 SIGCHLD 信号给父进程，父进程收到此信号后，执行 waitpid () 函数为子进程收尸。这是基于这样的原理：就算父进程没有调用 wait，内核也会向它发送 SIGCHLD 消息，尽管默认处理是忽略，如果想响应这个消息，可以设置一个处理函数。
- 把父进程杀掉。父进程死后，僵尸进程成为”孤儿进程”，过继给 1 号进程 init，init 始终会负责清理僵尸进程．它产生的所有僵尸进程也跟着消失。

孤儿进程：

- 父进程运行结束，但子进程还在运行 (未运行结束) 的子进程就称为孤儿进程。
- 孤儿进程最终会被 init 进程 (进程号为 1) 所收养，因此 init 进程此时变成孤儿进程的父进程，并由 init 进程对它们完成状态收集工作。

2. CPU 的上下文切换有几种？系统中断进行了几次上下文切换？

上下文切换 (Context Switch) 是一种将 CPU 资源从一个进程分配给另一个进程的机制。操

作系统需要先存储当前进程的状态 (包括内存空间的指针，当前执行完的指令等等)，再读入下一个进程的状态，然后执行此进程。

CPU 的上下文切换分三种：进程上下文切换、线程上下文切换、中断上下文切换。

- 系统调用过程中也会发生 CPU 上下文切换。CPU 寄存器会先保存用户态的状态，然后加载内核态相关内容。系统调用结束之后，CPU 寄存器要恢复原来保存的用户态，继续运行进程。所以，一次系统调用，发生两次 CPU 上下文切换。

- 进程是由内核管理和调度的，进程的切换只能发生在内核态。进程上下文切换与系统调用的不同在于，进程的调用会保存用户空间的虚拟内存，全局变量等信息，但是系统调用的上下文则不会，因为其未发生进程的变化。

- 内核中的任务调度实际是在调度线程，进程只是给线程提供虚拟内存、全局变量等资源。线程上下文切换时，共享相同的虚拟内存和全局变量等资源不需要修改。而线程自己的私有数据，如栈和寄存器等，上下文切换时需要保存。

3. 进程的通信方式？效率最高的通信方式是什么？

- 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。

- 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。

- 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。

4. 进程调度算法有几种？应用最广泛的是什么？

FIFO 或 First Come, First Served (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。

Shortest Job First (SJF) 最短作业优先

- 最短的作业 (CPU 区间长度最小) 最先调度。
- 由于作业的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。
- SJF 可以保证最小的平均等待时间。

Shortest Remaining Job First (SRJF) 最短剩余作业优先

- 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。
- 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。

多级反馈队列算法详解：

- 进程在进入待调度的队列等待时，首先进入优先级最高的 Q1 等待。
- 首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q1,Q2,Q3 三个队列，只有在 Q1 中没有进程等待时才去调度 Q2，同理，只有 Q1,Q2 都为空时才会去调度 Q3。
- 对于同一个队列中的各个进程，按照时间片轮转法调度。比如 Q1 队列的时间片为 N，那么 Q1 中的作业在经历了 N 个时间片后若还没有完成，则进入 Q2 队列等待，若 Q2 的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。
- 在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU 马上分配给新到达的作业（抢占式）。

5. 进程和线程的区别？

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。

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；对进程进程操作一般开销都比较大，对线程开销就小了

- 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。

6. 虚拟内存的作用与特性？

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。

- 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。

当每个进程创建的时候，内核会为进程分配 4G 的虚拟内存，当进程还没有开始运行时，这只是一个内存布局。实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data 段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射就好（叫做存储器映射）。这个时候数据和代码还是在磁盘上的。当运行到对应的程序时，进程去寻找页表，发现页表中地址没有存放在物理内存上，而是在磁盘上，于是发生缺页异常，于是将磁盘上的数据拷贝到物理内存中。

7. 虚拟内存的实现方式？分别有何种缺陷？

方式：

- 请求分页存储管理。将虚拟内存空间和物理内存空间皆划分为大小相同的页面，如 4KB、8KB 或 16KB 等，并以页面作为内存空间的最小分配单位，一个程序的一个页面可以存放在任意一个物理页面里。页是信息的物理单位

- 请求分段存储管理。将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。段是信息的逻辑单位

分页存储的缺点：产生内部碎片
分段存储的缺点：产生外部碎片

采用段页式管理就是将程序分为多个逻辑段，在每个段里面又进行分页，即将分段和分页组合起来使用。这样做的目的就是想同时获得分段和分页的好处，但又避免了单独分段或单独分页的缺陷。

8. 页面置换算法？

- 最优页面置换算法（往后看）：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

- 先进先出页面置换算法（FIFO）及其改进（往前看）：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。

- 最近最少使用页面置换算法 LRU（往前看）：总是选择在最近一段时间内最久不用的页面予以淘汰。 即淘汰最近最长时间未访问过的页面。

9. 什么是中断？产生中断的方式？

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（广义）是激活操作系统的唯一方式。

产生方式：

- 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。

10. 什么是系统调用？

- 操作系统提供的函数就被称为系统调用（system call）。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。

11. 什么会导致用户态陷入内核态？

- 系统调用：操作系统提供的函数就被称为系统调用（system call）。

- 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。

12. 陷阱和中断的区别？

- 陷阱指令可以使执行流程从用户态陷入内核并把控制权转移给操作系统，使得用户程序可以调用内核函数和使用硬件从而获得操作系统所提供的服务，比如用视频软件放一个电影，视频软件就发出陷阱使用显示器和声卡从而访问硬件。

- 中断是由外部事件导致并且它发生的时间是不可预测的，这一点和陷阱不同。外部事件主要是指时钟中断，硬件中断等。CPU 决定切换到另一个进程运行，就会产生一个时钟中断，切换到下一个进程运行。

13. 互斥和同步的关系？

- 互斥某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

- 同步是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

实现同步和互斥的方式：

- 临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。
- 互斥量：为协调共同对一个共享资源的单独访问而设计的。
- 信号量：为控制一个具有有限数量用户资源而设计。
- 事 件：用来通知线程有一些事件已发生，从而启动后继任务的开始。

14. 死锁产生的条件？

- 互斥条件 (Mutual exclusion)：资源不能被共享，只能由一个进程使用。

- 请求与保持条件 (Hold and wait)：已经得到资源的进程可以再次申请新的资源。

- 非抢占条件 (No pre-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 (Circular wait)：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。

15. 信号量的 PV 实现？

```
struct    semaphore {
   int count;
   queueType queue;
};

// P 操作，进程请求分配一个资源
void semWait(semaphore s){
   s.count--;
   if(s.count < 0){
      /*把当前进程插入队列*/
      /*阻塞当前进程*/
   }
}

// V 操作，进程释放一个资源
void semSignal(semaphore s){
   s.count++;
   if(s.count <= 0){
      /*把进程 p 从队列中移除*/
      /*把进程 p 插入就绪队列*/
   }

}
```

总结：V 原语操作的本质在于：一个进程使用完临界资源后，释放临界资源，使 S 加 1，以通知其它的进程，这个时候如果 S<=0，表明有进程阻塞在该类资源上，因此要从阻塞队列里唤醒一个进程来“转手”该类资源；如果资源数 count>0，代表阻塞队列没有进程阻塞，就不需要唤醒操作。

16. 生产者消费者的代码实现？

```
semaphore s = 1;                    // 临界区互斥信号量
semaphore empty = n;                  // 空闲缓冲区
semaphore full = 0;                  // 满缓冲区

void producter(){
   while(true){
      semWait(empty);
      semWait(s);
      /*把数据送到缓冲区*/
      semSignal(s);
```

```
        semSignal(full);
    }
}

void consumer(){
    while(true){
        semWait(full);
        semWait(s);
        /*从缓冲区取出数据*/
        semSignal(s);
        semSignal(empty);
    }
}
```

17. 如何动态查看服务器日志文件？

- 命令 tail -f 文件 可以对某个文件进行动态监控，例如 tomcat 的日志文件， 会随着程序的运行，日志会变化，可以使用 tail -f catalina-2016-11-11.log 监控 文 件的变化

18. 如何打包压缩文件？

Linux 中的打包文件一般是以.tar 结尾的，压缩的命令一般是以.gz 结尾的。而一般情况下打包和压缩是一起进行的，打包并压缩后的文件的后缀名一般.tar.gz。

命令：tar -zcvf 打包压缩后的文件名 要打包压缩的文件

- z：调用 gzip 压缩命令进行压缩
- c：打包文件
- v：显示运行过程
- f：指定文件名

解压压缩包：命令：tar [-xvf] 压缩文件，其中：x：代表解压

19. 修改 /test 下的 aaa.txt 的权限为属主有全部权限，属主所在的组有读写权限， 其他用户只有读的权限？

- 修改文件 / 目录的权限的命令：chmod

- 修改 /test 下的 aaa.txt 的权限为属主有全部权限，属主所在的组有读写权限， 其他用户只有读的权限：chmod u=rwx,g=rw,o=r aaa.txt

20. 查找 text.txt 文件中的 abc 的位置？

- grep 要搜索的字符串 要搜索的文件 --color：搜索命令，–color 代表高亮显示
- grep “abc” text.txt

21. 在 /home 目录下查找以.txt 结尾的文件名？

- find /home -name “*.txt”

22. 普通文件 IO 页缓存需要复制几次？具体过程是什么？

- Linux 的缓存 IO 机制中，操作系统会将 IO 的数据缓存在文件系统的页缓存中，也就是说，数据会先被拷贝到操作系统内核的缓冲区，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。
- 所有的文件内容的读取（无论一开始是命中页缓存还是没有命中页缓存）最终都是直接来源于页缓存。当将数据从磁盘复制到页缓存之后，还要将页缓存的数据通过 CPU 复制到 read 调用提供的缓冲区中，这就是普通文件 IO 需要的两次复制数据复制过程。其中第一次是通过 DMA 的方式将数据从磁盘复制到页缓存中;第二次是将数据从页缓存复制到进程自己的的地址空间对应的物理内存中。

23. 常见的 IO 模型？

- 同步阻塞 IO（blocking IO）：当用户进程调用了 recvfrom 这个系统调用，kernel 就开始了 IO 的第一个阶段：准备数据，对于网络 IO 来说，很多时候数据在一开始还没有到达。这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区中是需要一个过程的。而在用户进程这边，整个进程会被阻塞。
- 同步非阻塞 IO（nonblocking IO）当用户进程发出 read 操作时，如果 kernel 中的数据还没有准备好，那么它并不会 block 用户进程，而是立刻返回一个 error。从用户进程角度讲 ，它发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦 kernel 中的数据准备好了，并且又再次收到了用户进程的 system call，那么它马上就将数据拷贝到了用户内存，然后返回。
- IO 多路复用（IO multiplexing） I/O 多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。可达到在同一个线程内同时处理多个 IO 请求的目的。select，poll，epoll 都是 IO 多路复用的机制。但 select，poll，epoll 本质上都是同步 I/O，因为他们都需

要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是同步阻塞的。

- 异步 IO （asynchronous IO）用户进程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从 kernel 的角度，当它受到一个 asynchronous read 之后，首先它会立刻返回，所以不会对用户进程产生任何 block。然后，kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel 会给用户进程发送一个 signal，告诉它 read 操作完成了。

24. IO 多路复用的 select、poll 和 epoll 函数的区别？

- select 函数监视文件描述符，调用后 select 函数会阻塞，直到有描述符就绪，或者超时，函数返回，当 select 函数返回后，就可以遍历描述符，找到就绪的描述符。select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制

- 没有最大限制（但是数量过大后性能也是会下降）。和 select 函数一样，poll 返回后，需要轮询来获取就绪的描述符。

- epoll 是 Linux 内核为处理大批量文件描述符而作了改进的 poll，是 Linux 下多路复用 IO 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率。另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合就行了。