1. Java 中的原始数据类型都有哪些，它们的大小及对应的封装类是什么？

- **boolean**
  boolean 数据类型非 true 即 false。这个数据类型表示 1 bit，但是它的大小并没有精确定义。
  《Java 虚拟机规范》中如是说：“虽然定义了 boolean 这种数据类型，但是只对它提供了非常有限的支持。在 Java 虚拟机中没有任何供 boolean 值专用的字节码指令，Java 语言表达式所操作的 boolean 值，在编译之后都使用 Java 虚拟机中的 int 数据类型来代替，而 boolean 数组将会被编码成 Java 虚拟机的 byte 数组，每个元素 boolean 元素占 8 位”。这样我们可以得出 **boolean 类型单独使用是 4 个字节，在数组中又是 1 个字节。**那虚拟机为什么要用 int 来代替 boolean 呢？为什么不用 byte 或 short，这样不是更节省内存空间吗？实际上，使用 int 的原因是，对于当下 32 位的 CPU 来说，一次进行 32 位的数据交换更加高效。
  综上，我们可以知道：官方文档对 boolean 类型没有给出精确的定义，《Java 虚拟机规范》给出了“单独时使用 4 个字节，boolean 数组时 1 个字节”的定义，具体还要看虚拟机实现是否按照规范来，所以 1 个字节、4 个字节都是有可能的。这其实是一种时空权衡。
  boolean 类型的封装类是 Boolean。
- byte——1 byte——Byte
- short——2 bytes——Short
- int——4 bytes——Integer
- long——8 bytes——Long
- float——4 bytes——Float
- double——8 bytes——Double
- char——2 bytes——Character

2. 谈一谈”==“与”equals()"的区别。

《Think in Java》中说：“关系操作符生成的是一个 boolean 结果，它们计算的是操作数的值之间的关系”。
"=="判断的是两个对象的内存地址是否一样，适用于原始数据类型和枚举类型（它们的变量存储的是值本身，而引用类型变量存储的是引用）；equals 是 Object 类的方法，Object 对它的实现是比较内存地址，我们可以重写这个方法来自定义“相等”这个概念。比如类库中的 String、Date 等类就对这个方法进行了重写。
综上，对于枚举类型和原始数据类型的相等性比较，应该使用"=="；对于引用类型的相等性比较，应该使用 equals 方法。

3. Java 中的四种引用及其应用场景是什么？

强引用: 通常我们使用 new 操作符创建一个对象时所返回的引用即为强引用

软引用: 若一个对象只能通过软引用到达，那么这个对象在内存不足时会被回收，可用于图片缓存中，内存不足时系统会自动回收不再使用的 Bitmap

弱引用: 若一个对象只能通过弱引用到达，那么它就会被回收（即使内存充足），同样可用于图片缓存中，这时候只要 Bitmap 不再使用就会被回收

虚引用: 虚引用是 Java 中最“弱”的引用，通过它甚至无法获取被引用的对象，它存在的唯一作用就是当它指向的对象回收时，它本身会被加入到引用队列中，这样我们可以知道它指向的对象何时被销毁。

4. object 中定义了哪些方法？

clone(), equals(), hashCode(), toString(), notify(), notifyAll(), wait(), finalize(), getClass()

5. hashCode 的作用是什么？

请参见散列表的基本原理与实现

6. ArrayList, LinkedList, Vector 的区别是什么？

ArrayList: 内部采用数组存储元素，支持高效随机访问，支持动态调整大小

LinkedList: 内部采用链表来存储元素，支持快速插入/删除元素，但不支持高效地随机访问

Vector: 可以看作线程安全版的 ArrayList

7. String, StringBuilder, StringBuffer 的区别是什么？

String: 不可变的字符序列，若要向其中添加新字符需要创建一个新的 String 对象

StringBuilder: 可变字符序列，支持向其中添加新字符（无需创建新对象）

StringBuffer: 可以看作线程安全版的 StringBuilder

8. Map, Set, List, Queue、Stack 的特点及用法。

Map: Java 中存储键值对的数据类型都实现了这个接口，表示“映射表”。支持的两个核心操作是 *get(Object key)*以及 *put(K key, V value)*，分别用来获取键对应的值以及向映射表中插入键值对。

Set: 实现了这个接口的集合类型中不允许存在重复的元素，代表数学意义上的“集合”。它所支持的核心操作有 *add(E e)*, *remove(Object o)*, *contains(Object o)*，分别用于添加元素，删除元素以及判断给定元素是否存在于集中。

List: Java 中集合框架中的列表类型都实现了这个接口，表示一种有序序列。支持 *get(int index)*, *add(E e)*等操作。

Queue: Java 集合框架中的队列接口，代表了“先进先出”队列。支持 *add(E element)*, *remove()*等操作。

Stack: Java 集合框架中表示堆栈的数据类型，堆栈是一种“后进先出”的数据结构。支持 *push(E item)*, *pop()*等操作。

更详细的说明请参考官方文档，对相关数据结构不太熟悉的同学可以参考《算法导论》或其他相关书籍。

9. HashMap 和 HashTable 的区别

HashTable 是线程安全的，而 HashMap 不是

HashMap 中允许存在 null 键和 null 值，而 HashTable 中不允许

10. HashMap 的实现原理
简单的说，HashMap 的底层实现是“基于拉链法的散列表”。详细分析请参考深入解析 HashMap、HashTable

11. ConcurrentHashMap 的实现原理
ConcurrentHashMap 是支持并发读写的 HashMap，它的特点是读取数据时无需加锁，写数据时可以保证加锁粒度尽可能的小。由于其内部采用“分段存储”，只需对要进行写操作的数据所在的“段”进行加锁。关于 ConcurrentHashMap 底层实现的详细分析请参考 Java 并发编程：并发容器之 ConcurrentHashMap

12. TreeMap, LinkedHashMap, HashMap 的区别是什么？

HashMap 的底层实现是散列表，因此它内部存储的元素是无序的；

TreeMap 的底层实现是红黑树，所以它内部的元素的有序的。排序的依据是自然序或者是创建 TreeMap 时所提供的比较器（Comparator）对象。

LinkedHashMap 可以看作能够记住插入元素的顺序的 HashMap。

13. Collection 与 Collections 的区别是什么？

- Collection 是 Java 集合框架中的基本接口；
- Collections 是 Java 集合框架提供的一个工具类，其中包含了大量用于操作或返回集合的静态方法。

14. 对于“try-catch-finally”，若 try 语句块中包含“return”语句，finally 语句块会执行吗？
会执行。只有两种情况 finally 块中的语句不会被执行：**

-

调用了 *System.exit()*方法；

-
-

JVM“崩溃”了。

-

15. Java 中的异常层次结构

Java 中的异常层次结构如下图所示：

我们可以看到 Throwable 类是异常层级中的基类。Error 类表示内部错误，这类错误使我们无法控制的；Exception 表示异常，RuntimeException 及其子类属于未检查异常，这类异常包括 ArrayIndexOutOfBoundsException、NullPointerException 等，我们应该通过条件判断等方式语句避免未检查异常的发生。IOException 及其子类属于已检查异常，编译器会检查我们是否为所有可能抛出的已检查异常提供了异常处理器，若没有则会报错。对于未检查异常，我们无需捕获（当然 Java 也允许我们捕获，但我们应该做的事避免未检查异常的发生）。
复制代码

16. Java 面向对象的三个特征与含义

三大特征：封装、继承、多态。详细介绍请戳 Java 面向对象三大特性

17. Override, Overload 的含义与区别

Override 表示“重写”，是子类对父类中同一方法的重新定义

Overload 表示“重载”，也就是定义一个与已定义方法名称相同但签名不同的新方法**

18. 接口与抽象类的区别

接口是一种约定，实现接口的类要遵循这个约定；

抽象类本质上是一个类，使用抽象类的代价要比接口大。

接口与抽象类的对比如下：

- 抽象类中可以包含属性，方法（包含抽象方法与有着具体实现的方法），常量；接口只能包含常量和方法声明。
-
-

抽象类中的方法和成员变量可以定义可见性（比如 public、private 等）；而接口中的方法只能为 public（缺省为 public）。

-
-

一个子类只能有一个父类（具体类或抽象类）；而一个接口可以继承一个多个接口，一个类也可以实现多个接口。

-
-

子类中实现父类中的抽象方法时，可见性可以大于等于父类中的；而接口实现类中的接口 方法的可见性只能与接口中相同（public）。

-

19. 静态内部类与非静态内部类的区别

静态内部类不会持有外围类的引用，而非静态内部类会隐式持有外围类的一个引用。

20. Java 中多态的实现原理

所谓多态，指的就是父类引用指向子类对象，调用方法时会调用子类的实现而不是父类的实现。多态的实现的关键在于“动态绑定”。详细介绍请戳 Java 动态绑定的内部实现机制

21. 简述 Java 中创建新线程的两种方法

继承 Thread 类（假设子类为 MyThread），并重写 run()方法，然后 new 一个 MyThread 对象并对其调用 start()即可启动新线程。

实现 Runnable 接口（假设实现类为 MyRunnable），而后将 MyRunnable 对象作为参数传入 Thread 构造器，在得到的 Thread 对象上调用 start()方法即可。

22. 简述 Java 中进行线程同步的方法

volatile: Java Memory Model 保证了对同一个 volatile 变量的写 happens before 对它的读；

synchronized: 可以来对一个代码块或是对一个方法上锁，被“锁住”的地方称为临界区，进入临界区的线程会获取对象的 monitor，这样其他尝试进入临界区的线程会因无法获取 monitor 而被阻塞。由于等待另一个线程释放 monitor 而被阻塞的线程无法被中断。

ReentrantLock: 尝试获取锁的线程可以被中断并可以设置超时参数。

23. 简述 Java 中具有哪几种粒度的锁

Java 中可以对类、对象、方法或是代码块上锁。

24. 给出“生产者-消费者”问题的一种解决方案

使用阻塞队列：

```
public class BlockingQueueTest {
  private int size = 20;
  private ArrayBlockingQueue blockingQueue = new
ArrayBlockingQueue<>(size);
  public static void main(String[] args) {
    BlockingQueueTest test = new BlockingQueueTest();
    Producer producer = test.new Producer();
    Consumer consumer = test.new Consumer();
    producer.start(); consumer.start();
  }

  class Consumer extends Thread{
    @Override
    public void run() {
      while(true){
        try {
          //从阻塞队列中取出一个元素
          queue.take();
          System.out.println("队列剩余" + queue.size() + "个元素");
        } catch (InterruptedException e) {
        }
      }
    }
  }

  class Producer extends Thread{
```

```
    @Override
    public void run() {
      while (true) {
        try {
          //向阻塞队列中插入一个元素
          queue.put(1);
          System.out.println("队列剩余空间：" + (size - queue.size()));
        } catch (InterruptedException e) {
        }
      }
    }
  }
}
```
复制代码

25. ThreadLocal 的设计理念与作用

ThreadLocal 的作用是提供线程内的局部变量，在多线程环境下访问时能保证各个线程内的 ThreadLocal 变量各自独立。也就是说，每个线程的 ThreadLocal 变量是自己专用的，其他线程是访问不到的。ThreadLocal 最常用于以下这个场景：多线程环境下存在对非线程安全对象的并发访问，而且该对象不需要在线程间共享，但是我们不想加锁，这时候可以使用 ThreadLocal 来使得每个线程都持有一个该对象的副本。

26. concurrent 包的整体架构

27. ArrayBlockingQueue, CountDownLatch 类的作用

CountDownLatch: 允许线程集等待直到计数器为 0。适用场景: 当一个或多个线程需要等待指定数目的事件发生后再继续执行。

ArrayBlockingQueue: 一个基于数组实现的阻塞队列，它在构造时需要指定容量。当试图向满队列中添加元素或者从空队列中移除元素时，当前线程会被阻塞。通过阻塞队列，我们可以按以下模式来工作：工作者线程可以周期性的将中间结果放入阻塞队列中，其它线程可以取出中间结果并进行进一步操作。若工作者线程的执行比较慢（还没来得及向队列中插入元素），其他从队列中取元素的线程会等待它（试图从空队列中取元素从而阻塞）；若工作者线程执行较快（试图向满队列中插入元素），则它会等待其它线程取出元素再继续执行。

28. wait()，sleep() 的区别

wait(): Object 类中定义的实例方法。在指定对象上调用 wait 方法会让当前线程进入等待状态（前提是当前线程持有该对象的 monitor），此时当前线程会释放相应对象的 monitor，这样一来其它线程便有机会获取这个对象的 monitor 了。当其它线程获取了这个对象的 monitor 并进行了所需操作时，便可以调用 notify 方法唤醒之前进入等待状态的线程。

sleep(): Thread 类中的静态方法，作用是让当前线程进入休眠状态，以便让其他线程有机会执行。进入休眠状态的线程不会释放它所持有的锁。

29. 线程池的用法与优势

优势: 实现对线程的复用，避免了反复创建及销毁线程的开销；使用线程池统一管理线程可以减少并发线程的数目，而线程数过多往往会在线程上下文切换上以及线程同步上浪费过多时间。

用法: 我们可以调用 ThreadPoolExecutor 的某个构造方法来自己创建一个线程池。但通常情况下我们可以使用 Executors 类提供给我们的静态工厂方法来更方便的创建一个线程池对象。创建了线程池对象后，我们就可以调用 submit 方法提交任务到线程池中去执行了；线程池使用完毕后我们要记得调用 shutdown 方法来关闭它。

30. for-each 与常规 for 循环的效率对比

关于这个问题我们直接看《Effective Java》给我们做的解答：

for-each 能够让代码更加清晰，并且减少了出错的机会。
下面的惯用代码适用于集合与数组类型：

```
for (Element e : elements) {
   doSomething(e);
}
```
复制代码

使用 for-each 循环与常规的 for 循环相比，并不存在性能损失，即使对数组进行迭代也是如此。实际上，在有些场合下它还能带来微小的性能提升，因为它只计算一次数组索引的上限。

31. 简述 Java IO 与 NIO 的区别

Java IO 是面向流的，这意味着我们需要每次从流中读取一个或多个字节，直到读取完所有字节；NIO 是面向缓冲的，也就是说会把数据读取到一个缓冲区中，然后对缓冲区中的数据进行相应处理。

Java IO 是阻塞 IO，而 NIO 是非阻塞 IO。

Java NIO 中存在一个称为选择器（selector）的东西，它允许你把多个通道（channel）注册到一个选择器上，然后使用一个线程来监视这些通道：若这些通道里有某个准备好可以开始进行读或写操作了，则开始对相应的通道进行读写。而在等待某通道变为可读/写期间，请求对通道进行读写操作的线程可以去干别的事情。

32. 反射的作用与原理

反射的作用概括地说是运行时获取类的各种定义信息，比如定义了哪些属性与方法。原理是通过类的 class 对象来获取它的各种信息。

33. Java 中的泛型机制

关于泛型机制的详细介绍请直接戳 Java 核心技术点之泛型

34. Java 7 与 Java 8 的新特性

这里有两篇总结的非常好的：

Java 7 的新特性

Java 8 的新特性

35. 常见设计模式

所谓“设计模式”，不过是面向对象编程中一些常用的软件设计手法，并且经过实践的检验，这些设计手法在各自的场景下能解决一些需求，因此它们就成为了如今广为流传的”设计模式“。也就是说，正式因为在某些场景下产生了一些棘手的问题，才催生了相应的设计模式。明确了这一点，我们在学习某种设计模式时要充分理解它产生的背景以及它所解决的主要矛盾是什么。

常用的设计模式可以分为以下三大类：

创建型模式: 包括工厂模式（又可进一步分为简单工厂模式、工厂方法模式、抽象工厂模式）、建造者模式、单例模式。

结构型模式: 包括适配器模式、桥接模式、装饰模式、外观模式、享元模式、代理模式。

行为型模式: 包括命令模式、中介者模式、观察者模式、状态模式、策略模式。