最近5年133个Java面试问题列表

Java 面试随着时间的改变而改变。在过去的日子里，当你知道 String 和 StringBuilder 的区别就能让你直接进入第二轮面试，但是现在问题变得越来越高级，面试官问的问题也更深入。 在我初入职场的时候，类似于 Vector 与 Array 的区别、HashMap 与 Hashtable 的区别是最流行的问题，只需要记住它们，就能在面试中获得更好的机会，但这种情形已经不复存在。如今，你将会被问到许多 Java 程序员都没有看过的领域，如 NIO，[设计模式](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8)，成熟的单元测试，或者那些很难掌握的知识，如并发、算法、数据结构及编码。

由于我喜欢研究面试题，因此我已经收集了许多的面试问题，包括许多许多不同的主题。我已经为这众多的问题准备一段时间了，现在我将它们分享给你们。这里面不但包含经典的面试问题，如线程、集合、equals 和 hashcode、socket，而且还包含了 NIO、数组、字符串、Java 8 等主题。

该列表包含了入门级 Java 程序员和多年经验的高级开发者的问题。无论你是 1、2、3、4、5、6、7、8、9 还是 10 年经验的开发者，你都能在其中找到一些有趣的问题。这里包含了一些超级容易回答的问题，同时包含经验丰富的 Java 程序员也会棘手的问题。

当然你们也是非常幸运的，当今有许多好的书来帮助你准备 Java 面试，其中有一本我觉得特别有用和有趣的是 Markham 的 Java 程序面试揭秘（Java Programming Interview Exposed）。 这本书会告诉你一些 Java 和 JEE 面试中最重要的主题，即使你不是准备 Java 面试，也值得一读。

该问题列表特别长，我们有各个地方的问题，所以，答案必须要短小、简洁、干脆，不拖泥带水。因此，除了这一个段落，你只会听到问题与答案，再无其他内容，没有反馈，也没有评价。为此，我已经写好了一些博文，在这些文章中你可以找到我对某些问题的观点，如我为什么喜欢这个问题，这个问题的挑战是什么？期望从面试者那获取到什么样的答案？

这个列表有一点不同，我鼓励你采用类似的方式去分享问题和答案，这样容易温习。我希望这个列表对面试官和候选人都有很好的用处，面试官可以对这些问题上做一些改变以获取新奇和令人惊奇的元素，这对一次好的面试来说非常重要。而候选者，可以扩展和测试 Java 程序语言和平台关键领域的知识。2015 年，会更多的关注并发概念，JVM 内部，32 位 JVM 和 64 JVM的区别，单元测试及整洁的代码。我确信，如果你读过这个庞大的 Java 面试问题列表，无论是电话面试还是面对面的面试，你都能有很好的表现。

Java 面试中的重要话题

除了你看到的惊人的问题数量，我也尽量保证质量。我不止一次分享各个重要主题中的问题，也确保包含所谓的高级话题，这些话题很多程序员不喜欢准备或者直接放弃，因为他们的工作不会涉及到这些。Java NIO 和 JVM 底层就是最好的例子。你也可以将设计模式划分到这一类中，但是越来越多有经验的程序员了解 GOF 设计模式并应用这些模式。我也尽量在这个列表中包含 2015 年最新的面试问题，这些问题可能是来年关注的核心。为了给你一个大致的了解,下面列出这份 Java 面试问题列表包含的主题：

多线程，并发及线程基础  
数据类型转换的基本原则  
垃圾回收（GC）  
Java 集合框架  
数组  
字符串  
GOF 设计模式  
SOLID （**单一功能、开闭原则、里氏替换、接口隔离**以及**依赖反转**）设计原则  
抽象类与接口  
Java 基础，如 equals 和 hashcode  
泛型与枚举  
Java IO 与 NIO  
常用网络协议  
Java 中的数据结构和算法  
正则表达式  
JVM 底层  
Java 最佳实践  
JDBC  
Date, Time 与 Calendar  
Java 处理 XML  
JUnit  
编程

120 大 [Java 面试题](http://forum.jobbole.com/showthread.php/1707)及答案

现在是时候给你展示我近 5 年从各种面试中收集来的 120 个问题了。我确定你在自己的面试中见过很多这些问题，很多问题你也能正确回答。

多线程、并发及线程的基础问题

1）Java 中能创建 volatile 数组吗？  
能，Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。我的意思是，如果改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素，volatile 标示符就不能起到之前的保护作用了。

2）volatile 能使得一个非原子操作变成原子操作吗？  
一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

3）volatile 修饰符的有过什么实践？  
一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

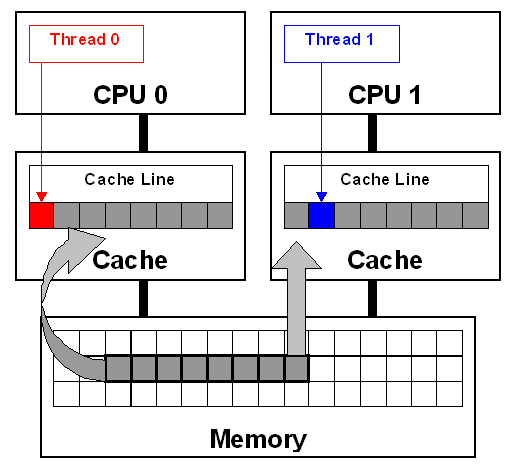
4）volatile 类型变量提供什么保证？([答案](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html))  
volatile 变量提供顺序和可见性保证，例如，JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的，但 volatile 类型的 double 和 long 就是原子的。

5) 10 个线程和 2 个线程的同步代码，哪个更容易写？  
从写代码的角度来说，两者的复杂度是相同的，因为同步代码与线程数量是相互独立的。但是同步策略的选择依赖于线程的数量，因为越多的线程意味着更大的竞争，所以你需要利用同步技术，如锁分离，这要求更复杂的代码和专业知识。

6）你是如何调用 wait（）方法的？使用 if 块还是循环？为什么？([答案](http://javarevisited.blogspot.sg/2015/07/how-to-use-wait-notify-and-notifyall-in.html))  
wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // The standard idiom for using the wait method  synchronized (obj) {  while (condition does not hold)  obj.wait(); // (Releases lock, and reacquires on wakeup)  ... // Perform action appropriate to condition  } |

参见 [Effective Java](http://www.amazon.com/gp/product/B000WJOUPA/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000WJOUPA&linkCode=as2&tag=job0ae-20) 第 69 条，获取更多关于为什么应该在循环中来调用 wait 方法的内容。

7）什么是多线程环境下的伪共享（false sharing）？  
伪共享是多线程系统（每个处理器有自己的局部缓存）中一个众所周知的性能问题。伪共享发生在不同处理器的上的线程对变量的修改依赖于相同的缓存行，如下图所示：  
[](http://jbcdn2.b0.upaiyun.com/2015/11/2bccd7f52a70db95aa72524ef3a55164.gif)

有经验程序员的 Java 面试题

伪共享问题很难被发现，因为线程可能访问完全不同的全局变量，内存中却碰巧在很相近的位置上。如其他诸多的并发问题，避免伪共享的最基本方式是仔细审查代码，根据缓存行来调整你的数据结构。

8）什么是 Busy spin？我们为什么要使用它？  
Busy spin 是一种在不释放 CPU 的基础上等待事件的技术。它经常用于避免丢失 CPU 缓存中的数据（如果线程先暂停，之后在其他CPU上运行就会丢失）。所以，如果你的工作要求低延迟，并且你的线程目前没有任何顺序，这样你就可以通过循环检测队列中的新消息来代替调用 sleep() 或 wait() 方法。它唯一的好处就是你只需等待很短的时间，如几微秒或几纳秒。LMAX 分布式框架是一个高性能线程间通信的库，该库有一个 BusySpinWaitStrategy 类就是基于这个概念实现的，使用 busy spin 循环 EventProcessors 等待屏障。

9）Java 中怎么获取一份线程 dump 文件？  
在 Linux 下，你可以通过命令 kill -3 PID （Java 进程的进程 ID）来获取 Java 应用的 dump 文件。在 Windows 下，你可以按下 Ctrl + Break 来获取。这样 JVM 就会将线程的 dump 文件打印到标准输出或错误文件中，它可能打印在控制台或者日志文件中，具体位置依赖应用的配置。如果你使用Tomcat。

10）Swing 是线程安全的？([答案](http://javarevisited.blogspot.sg/2013/08/why-swing-is-not-thread-safe-in-java-Swingworker-Event-thread.html))  
不是，Swing 不是线程安全的。你不能通过任何线程来更新 Swing 组件，如 JTable、JList 或 JPanel，事实上，它们只能通过 GUI 或 AWT 线程来更新。这就是为什么 Swing 提供 invokeAndWait() 和 invokeLater() 方法来获取其他线程的 GUI 更新请求。这些方法将更新请求放入 AWT 的线程队列中，可以一直等待，也可以通过异步更新直接返回结果。你也可以在参考答案中查看和学习到更详细的内容。

11）什么是线程局部变量？([答案](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html))  
线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java 提供 ThreadLocal 类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

12）用 wait-notify 写一段代码来解决生产者-消费者问题？([答案](http://java67.blogspot.sg/2012/12/producer-consumer-problem-with-wait-and-notify-example.html))  
请参考答案中的示例代码。只要记住在同步块中调用 wait() 和 notify()方法，如果阻塞，通过循环来测试等待条件。

13) 用 Java 写一个线程安全的单例模式（Singleton）？([答案](http://javarevisited.blogspot.in/2012/12/how-to-create-thread-safe-singleton-in-java-example.html))  
请参考答案中的示例代码，这里面一步一步教你创建一个线程安全的 Java 单例类。当我们说线程安全时，意思是即使初始化是在多线程环境中，仍然能保证单个实例。Java 中，使用枚举作为单例类是最简单的方式来创建线程安全单例模式的方式。

14）Java 中 sleep 方法和 wait 方法的区别？([答案](http://java67.blogspot.sg/2012/08/what-are-difference-between-wait-and.html))  
虽然两者都是用来暂停当前运行的线程，但是 sleep() 实际上只是短暂停顿，因为它不会释放锁，而 wait() 意味着条件等待，这就是为什么该方法要释放锁，因为只有这样，其他等待的线程才能在满足条件时获取到该锁。

15）什么是不可变对象（immutable object）？Java 中怎么创建一个不可变对象？([答案](http://javarevisited.blogspot.sg/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html))  
不可变对象指对象一旦被创建，状态就不能再改变。任何修改都会创建一个新的对象，如 String、Integer及其它包装类。详情参见答案，一步一步指导你在 Java 中创建一个不可变的类。

16）我们能创建一个包含可变对象的不可变对象吗？  
是的，我们是可以创建一个包含可变对象的不可变对象的，你只需要谨慎一点，不要共享可变对象的引用就可以了，如果需要变化时，就返回原对象的一个拷贝。最常见的例子就是对象中包含一个日期对象的引用。

数据类型和 Java 基础面试问题

17）Java 中应该使用什么数据类型来代表价格？([答案](http://javarevisited.blogspot.sg/2012/02/java-mistake-1-using-float-and-double.html))  
如果不是特别关心内存和性能的话，使用BigDecimal，否则使用预定义精度的 double 类型。

18）怎么将 byte 转换为 String？([答案](http://javarevisited.blogspot.sg/2014/08/2-examples-to-convert-byte-array-to-String-in-Java.html))  
可以使用 String 接收 byte[] 参数的构造器来进行转换，需要注意的点是要使用的正确的编码，否则会使用平台默认编码，这个编码可能跟原来的编码相同，也可能不同。

19）Java 中怎样将 bytes 转换为 long 类型？  
这个问题你来回答 :-)

20）我们能将 int 强制转换为 byte 类型的变量吗？如果该值大于 byte 类型的范围，将会出现什么现象？  
是的，我们可以做强制转换，但是 Java 中 int 是 32 位的，而 byte 是 8 位的，所以，如果强制转化是，int 类型的高 24 位将会被丢弃，byte 类型的范围是从 -128 到 128。

21）存在两个类，B 继承 A，C 继承 B，我们能将 B 转换为 C 么？如 C = (C) B；([answer](http://javarevisited.blogspot.sg/2012/12/what-is-type-casting-in-java-class-interface-example.html)答案)

22）哪个类包含 clone 方法？是 Cloneable 还是 Object？([答案](http://javarevisited.blogspot.sg/2015/01/java-clone-tutorial-part-2-overriding-with-mutable-field-example.html))  
java.lang.Cloneable 是一个标示性接口，不包含任何方法，clone 方法在 object 类中定义。并且需要知道 clone() 方法是一个本地方法，这意味着它是由 c 或 c++ 或 其他本地语言实现的。

23）Java 中 ++ 操作符是线程安全的吗？(答案)  
23）不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

24）a = a + b 与 a += b 的区别(答案)  
+= 隐式的将加操作的结果类型强制转换为持有结果的类型。如果两这个整型相加，如 byte、short 或者 int，首先会将它们提升到 int 类型，然后在执行加法操作。如果加法操作的结果比 a 的最大值要大，则 a+b 会出现编译错误，但是 a += b 没问题，如下：  
byte a = 127;  
byte b = 127;  
b = a + b; // error : cannot convert from int to byte  
b += a; // ok  
（译者注：这个地方应该表述的有误，其实无论 a+b 的值为多少，编译器都会报错，因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错）

25）我能在不进行强制转换的情况下将一个 double 值赋值给 long 类型的变量吗？([答案](http://java67.blogspot.com/2014/11/how-to-convert-double-to-long-in-java-example.html))  
不行，你不能在没有强制类型转换的前提下将一个 double 值赋值给 long 类型的变量，因为 double 类型的范围比 long 类型更广，所以必须要进行强制转换。

26）3\*0.1 == 0.3 将会返回什么？true 还是 false？(答案)  
false，因为有些浮点数不能完全精确的表示出来。

27）int 和 Integer 哪个会占用更多的内存？(答案)  
Integer 对象会占用更多的内存。Integer 是一个对象，需要存储对象的元数据。但是 int 是一个原始类型的数据，所以占用的空间更少。

28）为什么 Java 中的 String 是不可变的（Immutable）？([answer](http://java67.blogspot.sg/2014/01/why-string-class-has-made-immutable-or-final-java.html)答案)  
Java 中的 String 不可变是因为 Java 的设计者认为字符串使用非常频繁，将字符串设置为不可变可以允许多个客户端之间共享相同的字符串。更详细的内容参见答案。

29）我们能在 Switch 中使用 String 吗？([answer](http://javarevisited.blogspot.sg/2011/08/string-switch-case-jdk7-example.html)答案)  
从 Java 7 开始，我们可以在 switch case 中使用字符串，但这仅仅是一个语法糖。内部实现在 switch 中使用字符串的 hash code。

30）Java 中的构造器链是什么？([answer](http://java67.blogspot.sg/2012/12/how-constructor-chaining-works-in-java.html)答案)  
当你从一个构造器中调用另一个构造器，就是Java 中的构造器链。这种情况只在重载了类的构造器的时候才会出现。

JVM 底层 与 GC（Garbage Collection） 的面试问题

31）64 位 JVM 中，int 的长度是多数？  
Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位。意思就是说，在 32 位 和 64 位 的Java 虚拟机中，int 类型的长度是相同的。

32）Serial 与 Parallel GC之间的不同之处？([答案](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html))  
Serial 与 Parallel 在GC执行的时候都会引起 stop-the-world。它们之间主要不同 serial 收集器是默认的复制收集器，执行 GC 的时候只有一个线程，而 parallel 收集器使用多个 GC 线程来执行。

33）32 位和 64 位的 JVM，int 类型变量的长度是多数？(答案)  
32 位和 64 位的 JVM 中，int 类型变量的长度是相同的，都是 32 位或者 4 个字节。

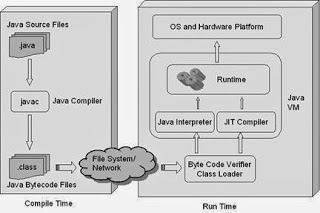
34）Java 中 WeakReference 与 SoftReference的区别？([答案](http://javarevisited.blogspot.sg/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html))  
虽然 WeakReference 与 SoftReference 都有利于提高 GC 和 内存的效率，但是 WeakReference ，一旦失去最后一个强引用，就会被 GC 回收，而软引用虽然不能阻止被回收，但是可以延迟到 JVM 内存不足的时候。

35）WeakHashMap 是怎么工作的？(答案)  
WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就是当 key 对象没有任何引用时，key/value 将会被回收。

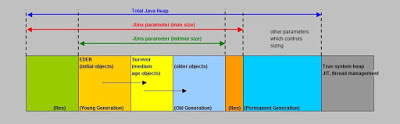
36）JVM 选项 -XX:+UseCompressedOops 有什么作用？为什么要使用？(答案)  
当你将你的应用从 32 位的 JVM 迁移到 64 位的 JVM 时，由于对象的指针从 32 位增加到了 64 位，因此堆内存会突然增加，差不多要翻倍。这也会对 CPU 缓存（容量比内存小很多）的数据产生不利的影响。因为，迁移到 64 位的 JVM 主要动机在于可以指定最大堆大小，通过压缩 OOP 可以节省一定的内存。通过 -XX:+UseCompressedOops 选项，JVM 会使用 32 位的 OOP，而不是 64 位的 OOP。

37）怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位？([答案](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html))  
你可以检查某些系统属性如 sun.arch.data.model 或 os.arch 来获取该信息。

38）32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数？([答案](http://javarevisited.blogspot.sg/2013/04/what-is-maximum-heap-size-for-32-bit-64-JVM-Java-memory.html))  
理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5 GB，Solaris 大约 3GB。64 位 JVM允许指定最大的堆内存，理论上可以达到 2^64，这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB。甚至有的 JVM，如 Azul，堆内存到 1000G 都是可能的。

39）JRE、JDK、JVM 及 JIT 之间有什么不同？([答案](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html))  
JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。JDK 代表 Java 开发工具（Java development kit），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（Java virtual machine），它的责任是运行 Java 应用。JIT 代表即时编译（Just In Time compilation），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。  
[](http://jbcdn2.b0.upaiyun.com/2015/11/4468a2440b48658c08acc50f15c3985b.jpg)

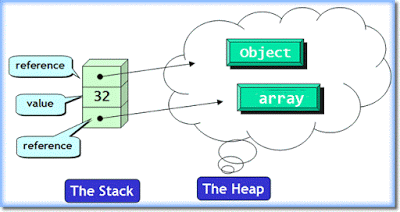
3 年工作经验的 Java 面试题

40）解释 Java 堆空间及 GC？([答案](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html))  
当通过 Java 命令启动 Java 进程的时候，会为它分配内存。内存的一部分用于创建堆空间，当程序中创建对象的时候，就从对空间中分配内存。GC 是 JVM 内部的一个进程，回收无效对象的内存用于将来的分配。  
[](http://jbcdn2.b0.upaiyun.com/2015/11/7ce5bea27bd3671ceab81f17629b395b.jpg)

JVM 底层面试题及答案

41）你能保证 GC 执行吗？(答案)  
不能，虽然你可以调用 System.gc() 或者 Runtime.gc()，但是没有办法保证 GC 的执行。

42）怎么获取 Java 程序使用的内存？堆使用的百分比？  
可以通过 java.lang.Runtime 类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。Runtime.freeMemory() 方法返回剩余空间的字节数，Runtime.totalMemory() 方法总内存的字节数，Runtime.maxMemory() 返回最大内存的字节数。

43）Java 中堆和栈有什么区别？([答案](http://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html))  
JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。  
[](http://jbcdn2.b0.upaiyun.com/2015/11/1495374ba77fdf61435e38755773cc30.gif)

关于内存的的面试问题和答案

Java 基本概念面试题

44）“a==b”和”a.equals(b)”有什么区别？([答案](http://javarevisited.blogspot.sg/2012/12/difference-between-equals-method-and-equality-operator-java.html))  
如果 a 和 b 都是对象，则 a==b 是比较两个对象的引用，只有当 a 和 b 指向的是堆中的同一个对象才会返回 true，而 a.equals(b) 是进行逻辑比较，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

45）a.hashCode() 有什么用？与 a.equals(b) 有什么关系？([答案](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html))  
hashCode() 方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap等等。它与 equals() 方法关系特别紧密。根据 Java 规范，两个使用 equal() 方法来判断相等的对象，必须具有相同的 hash code。

46）final、finalize 和 finally 的不同之处？([答案](http://javarevisited.blogspot.sg/2012/11/difference-between-final-finally-and-finalize-java.html))  
final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。finalize 方法是在对象被回收之前调用的方法，给对象自己最后一个复活的机会，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

47）Java 中的编译期常量是什么？使用它又什么风险？  
公共静态不可变（public static final ）变量也就是我们所说的编译期常量，这里的 public 可选的。实际上这些变量在编译时会被替换掉，因为编译器知道这些变量的值，并且知道这些变量在运行时不能改变。这种方式存在的一个问题是你使用了一个内部的或第三方库中的公有编译时常量，但是这个值后面被其他人改变了，但是你的客户端仍然在使用老的值，甚至你已经部署了一个新的jar。为了避免这种情况，当你在更新依赖 JAR 文件时，确保重新编译你的程序。

Java 集合框架的面试题

这部分也包含数据结构、算法及数组的面试问题

48) List、Set、Map 和 Queue 之间的区别(答案)  
List 是一个有序集合，允许元素重复。它的某些实现可以提供基于下标值的常量访问时间，但是这不是 List 接口保证的。Set 是一个无序集合。

49）poll() 方法和 remove() 方法的区别？  
poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

50）Java 中 LinkedHashMap 和 PriorityQueue 的区别是什么？([答案](http://javarevisited.blogspot.sg/2013/10/what-is-priorityqueue-data-structure-java-example-tutorial.html))  
PriorityQueue 保证最高或者最低优先级的的元素总是在队列头部，但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺序保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。

51）ArrayList 与 LinkedList 的不区别？([答案](http://java67.blogspot.sg/2012/12/difference-between-arraylist-vs-LinkedList-java.html))  
最明显的区别是 ArrrayList 底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构书链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)。更多细节的讨论参见答案。

52）用哪两种方式来实现集合的排序？([答案](http://java67.blogspot.sg/2012/07/sort-list-ascending-descending-order-set-arraylist.html))  
你可以使用有序集合，如 TreeSet 或 TreeMap，你也可以使用有顺序的的集合，如 list，然后通过 Collections.sort() 来排序。

53）Java 中怎么打印数组？([answer](http://java67.blogspot.sg/2014/03/how-to-print-array-in-java-example-tutorial.html)答案)  
你可以使用 Arrays.toString() 和 Arrays.deepToString() 方法来打印数组。由于数组没有实现 toString() 方法，所以如果将数组传递给 System.out.println() 方法，将无法打印出数组的内容，但是 Arrays.toString() 可以打印每个元素。

54）Java 中的 LinkedList 是单向链表还是双向链表？(答案)  
是双向链表，你可以检查 JDK 的源码。在 Eclipse，你可以使用快捷键 Ctrl + T，直接在编辑器中打开该类。

55）Java 中的 TreeMap 是采用什么树实现的？(答案)  
Java 中的 TreeMap 是使用红黑树实现的。

56) Hashtable 与 HashMap 有什么不同之处？([答案](http://java67.blogspot.sg/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html))  
这两个类有许多不同的地方，下面列出了一部分：  
a) Hashtable 是 JDK 1 遗留下来的类，而 HashMap 是后来增加的。  
b）Hashtable 是同步的，比较慢，但 HashMap 没有同步策略，所以会更快。  
c）Hashtable 不允许有个空的 key，但是 HashMap 允许出现一个 null key。  
更多的不同之处参见答案。

57）Java 中的 HashSet，内部是如何工作的？([answer](http://java67.blogspot.sg/2014/01/how-hashset-is-implemented-or-works-internally-java.html)答案)  
HashSet 的内部采用 HashMap来实现。由于 Map 需要 key 和 value，所以所有 key 的都有一个默认 value。类似于 HashMap，HashSet 不允许重复的 key，只允许有一个null key，意思就是 HashSet 中只允许存储一个 null 对象。

58）写一段代码在遍历 ArrayList 时移除一个元素？([答案](http://javarevisited.blogspot.sg/2014/01/ow-to-remove-objects-from-collection-arraylist-java-iterator-traversing.html))  
该问题的关键在于面试者使用的是 ArrayList 的 remove() 还是 Iterator 的 remove()方法。这有一段[示例代码](http://java67.blogspot.com/2015/10/how-to-solve-concurrentmodificationexception-in-java-arraylist.html)，是使用正确的方式来实现在遍历的过程中移除元素，而不会出现 ConcurrentModificationException 异常的示例代码。

59）我们能自己写一个容器类，然后使用 for-each 循环码？  
可以，你可以写一个自己的容器类。如果你想使用 Java 中增强的循环来遍历，你只需要实现 Iterable 接口。如果你实现 Collection 接口，默认就具有该属性。

60）ArrayList 和 HashMap 的默认大小是多数？([答案](http://javarevisited.blogspot.sg/2014/07/java-optimization-empty-arraylist-and-Hashmap-cost-less-memory-jdk-17040-update.html))

在 Java 7 中，ArrayList 的默认大小是 10 个元素，HashMap 的默认大小是16个元素（必须是2的幂）。这就是 Java 7 中 ArrayList 和 HashMap 类的代码片段：

|  |  |
| --- | --- |
| 1  2  3  4  5 | // from ArrayList.java JDK 1.7  private static final int DEFAULT\_CAPACITY = 10;    //from HashMap.java JDK 7  static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16 |

61）有没有可能两个不相等的对象有有相同的 hashcode？  
有可能，两个不相等的对象可能会有相同的 hashcode 值，这就是为什么在 hashmap 中会有冲突。相等 hashcode 值的规定只是说如果两个对象相等，必须有相同的hashcode 值，但是没有关于不相等对象的任何规定。

62）两个相同的对象会有不同的的 hash code 吗？  
不能，根据 hash code 的规定，这是不可能的。

63）我们可以在 hashcode() 中使用随机数字吗？([答案](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html))  
不行，因为对象的 hashcode 值必须是相同的。参见答案获取更多关于 Java 中重写 hashCode() 方法的知识。

64）Java 中，Comparator 与 Comparable 有什么不同？([答案](http://java67.blogspot.sg/2013/08/difference-between-comparator-and-comparable-in-java-interface-sorting.html))  
Comparable 接口用于定义对象的自然顺序，而 comparator 通常用于定义用户定制的顺序。Comparable 总是只有一个，但是可以有多个 comparator 来定义对象的顺序。

65）为什么在重写 equals 方法的时候需要重写 hashCode 方法？([答案](http://javarevisited.blogspot.sg/2015/01/why-override-equals-hashcode-or-tostring-java.html))  
因为有强制的规范指定需要同时重写 hashcode 与 equal 是方法，许多容器类，如 HashMap、HashSet 都依赖于 hashcode 与 equals 的规定。

Java IO 和 NIO 的面试题

IO 是 Java 面试中一个非常重要的点。你应该很好掌握 Java IO，NIO，NIO2 以及与操作系统，磁盘 IO 相关的基础知识。下面是 Java IO 中经常问的问题。

66）在我 Java 程序中，我有三个 socket，我需要多少个线程来处理？

67）Java 中怎么创建 ByteBuffer？

68）Java 中，怎么读写 ByteBuffer ？

69）Java 采用的是大端还是小端？

70）ByteBuffer 中的字节序是什么？

71）Java 中，直接缓冲区与非直接缓冲器有什么区别？([答案](http://javarevisited.blogspot.sg/2015/08/difference-between-direct-non-direct-mapped-bytebuffer-nio-java.html))

72）Java 中的内存映射缓存区是什么？([answer](http://javarevisited.blogspot.sg/2012/01/memorymapped-file-and-io-in-java.html)答案)

73）socket 选项 TCP NO DELAY 是指什么？

74）TCP 协议与 UDP 协议有什么区别？([answer](http://javarevisited.blogspot.com/2014/07/9-difference-between-tcp-and-udp-protocol.html)答案)

75）Java 中，ByteBuffer 与 StringBuffer有什么区别？(答案)

Java 最佳实践的面试问题

包含 Java 中各个部分的最佳实践，如集合，字符串，IO，多线程，错误和异常处理，设计模式等等。

76）Java 中，编写多线程程序的时候你会遵循哪些最佳实践？([答案](http://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html))  
这是我在写Java 并发程序的时候遵循的一些最佳实践：  
a）给线程命名，这样可以帮助调试。  
b）最小化同步的范围，而不是将整个方法同步，只对关键部分做同步。  
c）如果可以，更偏向于使用 volatile 而不是 synchronized。  
d）使用更高层次的并发工具，而不是使用 wait() 和 notify() 来实现线程间通信，如 BlockingQueue，CountDownLatch 及 Semeaphore。  
e）优先使用并发集合，而不是对集合进行同步。并发集合提供更好的可扩展性。

77）说出几点 Java 中使用 Collections 的最佳实践(答案)  
这是我在使用 Java 中 Collectionc 类的一些最佳实践：  
a）使用正确的集合类，例如，如果不需要同步列表，使用 ArrayList 而不是 Vector。  
b）优先使用并发集合，而不是对集合进行同步。并发集合提供更好的可扩展性。  
c）使用接口代表和访问集合，如使用List存储 ArrayList，使用 Map 存储 HashMap 等等。  
d）使用迭代器来循环集合。  
e）使用集合的时候使用泛型。

78）说出至少 5 点在 Java 中使用线程的最佳实践。([答案](http://java67.blogspot.com/2014/01/10-points-about-thread-and-javalangthread-in-java.html))  
这个问题与之前的问题类似，你可以使用上面的答案。对线程来说，你应该：  
a）对线程命名  
b）将线程和任务分离，使用线程池执行器来执行 Runnable 或 Callable。  
c）使用线程池

79）说出 5 条 IO 的最佳实践(答案)  
IO 对 Java 应用的性能非常重要。理想情况下，你不应该在你应用的关键路径上避免 IO 操作。下面是一些你应该遵循的 Java IO 最佳实践：  
a）使用有缓冲区的 IO 类，而不要单独读取字节或字符。  
b）使用 NIO 和 NIO2  
c）在 finally 块中关闭流，或者使用 try-with-resource 语句。  
d）使用内存映射文件获取更快的 IO。

80）列出 5 个应该遵循的 JDBC 最佳实践([答案](http://javarevisited.blogspot.sg/2012/08/top-10-jdbc-best-practices-for-java.html))  
有很多的最佳实践，你可以根据你的喜好来例举。下面是一些更通用的原则：  
a）使用批量的操作来插入和更新数据  
b）使用 PreparedStatement 来避免 SQL 异常，并提高性能。  
c）使用数据库连接池  
d）通过列名来获取结果集，不要使用列的下标来获取。

81）说出几条 Java 中方法重载的最佳实践？([答案](http://javarevisited.blogspot.sg/2013/01/java-best-practices-method-overloading-constructor.html))  
下面有几条可以遵循的方法重载的最佳实践来避免造成自动装箱的混乱。  
a）不要重载这样的方法：一个方法接收 int 参数，而另个方法接收 Integer 参数。  
b）不要重载参数数量一致，而只是参数顺序不同的方法。  
c）如果重载的方法参数个数多于 5 个，采用可变参数。

Date、Time 及 Calendar 的面试题

82）在多线程环境下，SimpleDateFormat 是线程安全的吗？([答案](http://javarevisited.blogspot.sg/2012/03/simpledateformat-in-java-is-not-thread.html))  
不是，非常不幸，DateFormat 的所有实现，包括 SimpleDateFormat 都不是线程安全的，因此你不应该在多线程序中使用，除非是在对外线程安全的环境中使用，如 将 SimpleDateFormat 限制在 ThreadLocal 中。如果你不这么做，在解析或者格式化日期的时候，可能会获取到一个不正确的结果。因此，从日期、时间处理的所有实践来说，我强力推荐 joda-time 库。

83）Java 中如何格式化一个日期？如格式化为 ddMMyyyy 的形式？([答案](http://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html))  
Java 中，可以使用 SimpleDateFormat 类或者 joda-time 库来格式日期。DateFormat 类允许你使用多种流行的格式来格式化日期。参见答案中的示例代码，代码中演示了将日期格式化成不同的格式，如 dd-MM-yyyy 或 ddMMyyyy。

84）Java 中，怎么在格式化的日期中显示时区？([答案](http://java67.blogspot.sg/2013/01/how-to-format-date-in-java-simpledateformat-example.html))

85）Java 中 java.util.Date 与 java.sql.Date 有什么区别？([答案](http://java67.blogspot.sg/2014/02/how-to-convert-javautildate-to-javasqldate-example.html))

86）Java 中，如何计算两个日期之间的差距？([程序](http://javarevisited.blogspot.sg/2015/07/how-to-find-number-of-days-between-two-dates-in-java.html))

87）Java 中，如何将字符串 YYYYMMDD 转换为日期？([答案](http://java67.blogspot.sg/2014/12/string-to-date-example-in-java-multithreading.html))

单元测试 JUnit 面试题

89）如何测试静态方法？(答案)  
可以使用 PowerMock 库来测试静态方法。

90）怎么利用 JUnit 来测试一个方法的异常？([答案](http://javarevisited.blogspot.sg/2013/04/JUnit-tutorial-example-test-exception-thrown-by-java-method.html))

91）你使用过哪个单元测试库来测试你的 Java 程序？(答案)

92）@Before 和 @BeforeClass 有什么区别？([答案](http://javarevisited.blogspot.sg/2013/04/JUnit-tutorial-example-test-exception-thrown-by-java-method.html))

编程和代码相关的面试题

93）怎么检查一个字符串只包含数字？([解决方案](http://java67.blogspot.com/2014/01/java-regular-expression-to-check-numbers-in-String.html))

94）Java 中如何利用泛型写一个 LRU 缓存？(答案<)

95）写一段 Java 程序将 byte 转换为 long？(答案)

95）在不使用 StringBuffer 的前提下，怎么反转一个字符串？([解决方案](http://java67.blogspot.com/2012/12/how-to-reverse-string-in-java-stringbuffer-stringbuilder.htm))

97）Java 中，怎么获取一个文件中单词出现的最高频率？([解决方案](http://java67.blogspot.com/2015/10/java-program-to-find-repeated-words-and-count.html))

98）如何检查出两个给定的字符串是反序的？([解决方案](http://javarevisited.blogspot.sg/2013/03/Anagram-how-to-check-if-two-string-are-anagrams-example-tutorial.html))

99）Java 中，怎么打印出一个字符串的所有排列？([解决方案](http://javarevisited.blogspot.com/2015/08/how-to-find-all-permutations-of-string-java-example.html))

100）Java 中，怎样才能打印出数组中的重复元素？([解决方案](http://javarevisited.blogspot.com/2015/06/3-ways-to-find-duplicate-elements-in-array-java.html))

101）Java 中如何将字符串转换为整数？([解决方案](http://java67.blogspot.com/2015/08/2-ways-to-parse-string-to-int-in-java.html))

102）在没有使用临时变量的情况如何交换两个整数变量的值？([解决方案](http://java67.blogspot.com/2015/08/how-to-swap-two-integers-without-using.html))

关于 OOP 和设计模式的面试题

这部分包含 Java 面试过程中关于 SOLID 的设计原则，OOP 基础，如类，对象，接口，继承，多态，封装，抽象以及更高级的一些概念，如组合、聚合及关联。也包含了 GOF 设计模式的问题。

103）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 Java 8 中你可以在接口声明静态的默认方法，这种方法是具体的。

104）Java 中，抽象类与接口之间有什么不同？([答案](http://javarevisited.blogspot.sg/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html))  
Java 中，抽象类和接口有很多不同之处，但是最重要的一个是 Java 中限制一个类只能继承一个类，但是可以实现多个接口。抽象类可以很好的定义一个家族类的默认行为，而接口能更好的定义类型，有助于后面实现多态机制。关于这个问题的讨论请查看答案。

105）除了单例模式，你在生产环境中还用过什么设计模式？  
这需要根据你的经验来回答。一般情况下，你可以说依赖注入，工厂模式，装饰模式或者观察者模式，随意选择你使用过的一种即可。不过你要准备回答接下的基于你选择的模式的问题。

106）你能解释一下里氏替换原则吗?([答案](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html))

107) 什么情况下会违反迪米特法则？为什么会有这个问题？([答案](http://javarevisited.blogspot.com/2014/05/law-of-demeter-example-in-java.html))  
迪米特法则建议“只和朋友说话，不要陌生人说话”，以此来减少类之间的耦合。

108）适配器模式是什么？什么时候使用？  
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。

109）什么是“依赖注入”和“控制反转”？为什么有人使用？([答案](http://javarevisited.blogspot.sg/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html))

110）抽象类是什么？它与接口有什么区别？你为什么要使用过抽象类？([答案](http://java67.blogspot.sg/2014/06/why-abstract-class-is-important-in-java.html))

111）构造器注入和 setter 依赖注入，那种方式更好？([答案](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html))  
每种方式都有它的缺点和优点。构造器注入保证所有的注入都被初始化，但是 setter 注入提供更好的灵活性来设置可选依赖。如果使用 XML 来描述依赖，Setter 注入的可读写会更强。经验法则是强制依赖使用构造器注入，可选依赖使用 setter 注入。

112）依赖注入和工程模式之间有什么不同？([答案](http://javarevisited.blogspot.sg/2015/06/difference-between-dependency-injection.html))  
虽然两种模式都是将对象的创建从应用的逻辑中分离，但是依赖注入比工程模式更清晰。通过依赖注入，你的类就是 POJO，它只知道依赖而不关心它们怎么获取。使用工厂模式，你的类需要通过工厂来获取依赖。因此，使用 DI 会比使用工厂模式更容易测试。关于这个话题的更详细讨论请参见答案。

113）适配器模式和装饰器模式有什么区别？([答案](http://javarevisited.blogspot.sg/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
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。

114）适配器模式和代理模式之前有什么不同？([答案](http://javarevisited.blogspot.sg/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
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。

115）什么是模板方法模式？(答案)  
模板方法提供算法的框架，你可以自己去配置或定义步骤。例如，你可以将排序算法看做是一个模板。它定义了排序的步骤，但是具体的比较，可以使用 Comparable 或者其语言中类似东西，具体策略由你去配置。列出算法概要的方法就是众所周知的模板方法。

116）什么时候使用访问者模式？(答案)  
访问者模式用于解决在类的继承层次上增加操作，但是不直接与之关联。这种模式采用双派发的形式来增加中间层。

117）什么时候使用组合模式？(答案)  
组合模式使用树结构来展示部分与整体继承关系。它允许客户端采用统一的形式来对待单个对象和对象容器。当你想要展示对象这种部分与整体的继承关系时采用组合模式。

118）继承和组合之间有什么不同？([答案](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html))  
虽然两种都可以实现代码复用，但是组合比继承共灵活，因为组合允许你在运行时选择不同的实现。用组合实现的代码也比继承测试起来更加简单。

119）描述 Java 中的重载和重写？([答案](http://java67.blogspot.sg/2012/09/difference-between-overloading-vs-overriding-in-java.html))  
重载和重写都允许你用相同的名称来实现不同的功能，但是重载是编译时活动，而重写是运行时活动。你可以在同一个类中重载方法，但是只能在子类中重写方法。重写必须要有继承。

120）Java 中，嵌套公共静态类与顶级类有什么不同？([答案](http://javarevisited.blogspot.sg/2012/12/inner-class-and-nested-static-class-in-java-difference.html))  
类的内部可以有多个嵌套公共静态类，但是一个 Java 源文件只能有一个顶级公共类，并且顶级公共类的名称与源文件名称必须一致。

121) OOP 中的 组合、聚合和关联有什么区别？([答案](http://javarevisited.blogspot.sg/2014/02/ifference-between-association-vs-composition-vs-aggregation.html))  
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。

122）给我一个符合开闭原则的设计模式的例子？([答案](http://javarevisited.blogspot.sg/2011/11/great-example-of-open-closed-design.html))  
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。

123）抽象工厂模式和原型模式之间的区别？(答案)

124）什么时候使用享元模式？(答案)  
享元模式通过共享对象来避免创建太多的对象。为了使用享元模式，你需要确保你的对象是不可变的，这样你才能安全的共享。JDK 中 String 池、Integer 池以及 Long 池都是很好的使用了享元模式的例子。

Java 面试中其他各式各样的问题

这部分包含 Java 中关于 XML 的面试题，JDB[C 面试题](http://forum.jobbole.com/showthread.php/1730)，正则表达式面试题，Java 错误和异常及序列化面试题

125）嵌套静态类与顶级类有什么区别？([答案](http://java67.blogspot.sg/2012/10/nested-class-java-static-vs-non-static-inner.html))  
一个公共的顶级类的源文件名称与类名相同，而嵌套静态类没有这个要求。一个嵌套类位于顶级类内部，需要使用顶级类的名称来引用嵌套静态类，如 HashMap.Entry 是一个嵌套静态类，HashMap 是一个顶级类，Entry是一个嵌套静态类。

126）你能写出一个正则表达式来判断一个字符串是否是一个数字吗？([解决方案](http://javarevisited.blogspot.sg/2012/10/regular-expression-example-in-java-to-check-String-number.html))  
一个数字字符串，只能包含数字，如 0 到 9 以及 +、- 开头，通过这个信息，你可以下一个如下的正则表达式来判断给定的字符串是不是数字。

127）Java 中，受检查异常 和 不受检查异常的区别？([答案](http://java67.blogspot.sg/2012/12/difference-between-runtimeexception-and-checked-exception.html))  
受检查异常编译器在编译期间检查。对于这种异常，方法强制处理或者通过 throws 子句声明。其中一种情况是 Exception 的子类但不是 RuntimeException 的子类。非受检查是 RuntimeException 的子类，在编译阶段不受编译器的检查。

128）Java 中，throw 和 throws 有什么区别？([答案](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html))

throw 用于抛出 java.lang.Throwable 类的一个实例化对象，意思是说你可以通过关键字 throw 抛出一个 Error 或者 一个Exception，如：  
throw new IllegalArgumentException(“size must be multiple of 2″)

而throws 的作用是作为方法声明和签名的一部分，方法被抛出相应的异常以便调用者能处理。Java 中，任何未处理的受检查异常强制在 throws 子句中声明。

129）Java 中，Serializable 与 Externalizable 的区别？([答案](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html))  
Serializable 接口是一个序列化 Java 类的接口，以便于它们可以在网络上传输或者可以将它们的状态保存在磁盘上，是 JVM 内嵌的默认序列化方式，成本高、脆弱而且不安全。Externalizable 允许你控制整个序列化过程，指定特定的二进制格式，增加安全机制。

130）Java 中，DOM 和 SAX 解析器有什么不同？([答案](http://javarevisited.blogspot.sg/2011/12/difference-between-dom-and-sax-parsers.html))  
DOM 解析器将整个 XML 文档加载到内存来创建一棵 DOM 模型树，这样可以更快的查找节点和修改 XML 结构，而 SAX 解析器是一个基于事件的解析器，不会将整个 XML 文档加载到内存。由于这个原因，DOM 比 SAX 更快，也要求更多的内存，不适合于解析大 XML 文件。

131）说出 JDK 1.7 中的三个新特性？([答案](http://javarevisited.blogspot.sg/2014/04/10-jdk-7-features-to-revisit-before-you.html))  
虽然 JDK 1.7 不像 JDK 5 和 8 一样的大版本，但是，还是有很多新的特性，如 try-with-resource 语句，这样你在使用流或者资源的时候，就不需要手动关闭，Java 会自动关闭。Fork-Join 池某种程度上实现 Java 版的 Map-reduce。允许 Switch 中有 String 变量和文本。菱形操作符(<>)用于类型推断，不再需要在变量声明的右边申明泛型，因此可以写出可读写更强、更简洁的代码。另一个值得一提的特性是改善异常处理，如允许在同一个 catch 块中捕获多个异常。

132）说出 5 个 JDK 1.8 引入的新特性？([答案](http://javarevisited.blogspot.sg/2014/02/10-example-of-lambda-expressions-in-java8.html))  
Java 8 在 Java 历史上是一个开创新的版本，下面 JDK 8 中 5 个主要的特性：  
Lambda 表达式，允许像对象一样传递匿名函数  
Stream API，充分利用现代多核 CPU，可以写出很简洁的代码  
Date 与 Time API，最终，有一个稳定、简单的日期和时间库可供你使用  
扩展方法，现在，接口中可以有静态、默认方法。  
重复注解，现在你可以将相同的注解在同一类型上使用多次。

133）Java 中，Maven 和 ANT 有什么区别？([答案](http://javarevisited.blogspot.sg/2015/01/difference-between-maven-ant-jenkins-and-hudson.html))  
虽然两者都是构建工具，都用于创建 Java 应用，但是 Maven 做的事情更多，在基于“约定优于配置”的概念下，提供标准的Java 项目结构，同时能为应用自动管理依赖（应用中所依赖的 JAR 文件），Maven 与 ANT 工具更多的不同之处请参见答案。

这就是所有的面试题，如此之多，是不是？我可以保证，如果你能回答列表中的所有问题，你就可以很轻松的应付任何核心 Java 或者高级 Java 面试。虽然，这里没有涵盖 Servlet、JSP、JSF、JPA，JMS，EJB 及其它 Java EE 技术，也没有包含主流的框架如 Spring MVC，Struts 2.0，Hibernate，也没有包含 SOAP 和 RESTful web service，但是这份列表对做 Java 开发的、准备应聘 Java web 开发职位的人还是同样有用的，因为所有的 Java 面试，开始的问题都是 Java 基础和 JDK API 相关的。如果你认为我这里有任何应该在这份列表中而被我遗漏了的 Java 流行的问题，你可以自由的给我建议。我的目的是从最近的面试中创建一份最新的、最优的 Java 面试问题列表。

Java EE 相关的面试题

为了做 Java EE 的朋友，这里列出了一些 web 开发的特定问题，你们可以用来准备 JEE 部分的面试：

10 大 Spring 框架面试题及答案([参见](http://javarevisited.blogspot.sg/2011/09/spring-interview-questions-answers-j2ee.html))  
10 个非常好的 XML 面试问题（Java 程序员）([参见](http://javarevisited.blogspot.sg/2013/01/10-xml-interview-questions-and-answers.html))  
20 个非常好的设计模式面试问题([参见](http://java67.blogspot.com/2012/09/top-10-java-design-pattern-interview-question-answer.html))  
10个最流行的 Struts 面试题（Java 开发者）([参见](http://javarevisited.blogspot.sg/2011/11/struts-interview-questions-answer-j2ee.html))  
20 个 Tibco Rendezvous 及 EMS 的面试题([更多](http://javarevisited.blogspot.sg/2011/01/tibco-rv-interview-question-as-part.html))  
10 个最频繁被问到的 Servlet 面试问题及答案([参见](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html))  
20 个 jQuery 面试问题（Java Web 开发者）([列表](http://javarevisited.blogspot.sg/2015/02/top-16-jquery-interview-questions.html))  
10 个非常好的 Oracle 面试问题（Java 开发者）([参见](http://javarevisited.blogspot.sg/2012/12/top-10-oracle-interview-questions-and-answers-database-sql.html))  
10 大 来自 J2EE 面试中的 JSP 问题([更多](http://javarevisited.blogspot.sg/2011/10/jsp-interview-questions-answers-for.html))  
12 个很好的 RESTful Web Services 面试问题([参见](http://javarevisited.blogspot.sg/2012/01/rest-web-services-framework-interview.html))  
10 大 EJB 面试问题及答案([参见](http://javarevisited.blogspot.sg/2012/03/top-10-ejb-interview-question-and.html))  
10 大 JMS 及 MQ 系列面试题及答案([列表](http://javarevisited.blogspot.sg/2014/03/top-10-websphere-mq-series-interview-questions-answers-active-rabbit.html))  
10 个非常好 Hibernate 面试问题（Java EE 开发者）([参见](http://javarevisited.blogspot.sg/2013/05/10-hibernate-interview-questions-answers-java-j2ee-senior.html))  
10 个非常好的 JDBC 面试题（Java 开发者）([参见](http://javarevisited.blogspot.sg/2012/12/top-10-jdbc-interview-questions-answers.html))  
15 个 Java NIO 和网络面试题及答案([参见](http://javarevisited.blogspot.sg/2014/08/socket-programming-networking-interview-questions-answers-Java.html))  
10 大 XSLT 面试题及答案([更多](http://javarevisited.blogspot.sg/2013/05/10-xslt-or-xml-xsl-transformation-interview-questions-answers-java.html))  
15 个来自 Java 面试的数据结构和算法问题([参见](http://javarevisited.blogspot.com/2013/03/top-15-data-structures-algorithm-interview-questions-answers-java-programming.html))  
10 大 Java 面试难题及答案([参见](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html))  
40 个核心 Java 移动开发面试题及答案([列表](http://java67.blogspot.com/2015/03/top-40-core-java-interview-questions-answers-telephonic-round.html))

**1.JVM相关(包括了各个版本的特性)**   
对于刚刚接触Java的人来说，JVM相关的知识不一定需要理解很深， 对此里面的概念有一些简单的了解即可。不过对于一个有着3年以上Java经验的资深开发者来说，不会JVM几乎是不可接受的。

JVM作为java运行的基础，很难相信对于JVM一点都不了解的人可以把java语言吃得很透。我在面试有超过3年Java经验的开发者的时候， JVM几乎就是一个必问的问题了。当然JVM不是唯一决定技术能力好坏的面试问题，但是可以佐证java开发能力的高低。

在JVM这个大类中，我认为需要掌握的知识有：   
JVM内存模型和结构GC原理，性能调优调优：Thread Dump， 分析内存结构   
class 二进制字节码结构， class loader 体系 ， class加载过程 ， 实例创建过程

方法执行过程：Java各个大版本更新提供的新特性(需要简单了解)

**2.Java的运行(基础必备)**   
这条可能出看很简单，java程序的运行谁不会呢?不过很多时候， 我们只是单纯通过IDE去执行java程序，底层IDE又是如何执行java程序呢?很多人并不了解。

这 个知识点是最最基本的java开发者需要掌握的，初学java，第一个肯定是教你如何在命令行中执行java程序，但是很多人一旦把java学完 了，IDE用上了，就把这个都忘了。为什么强调要知道这个呢，知道了java最纯粹的启动方式之后，你才能在启动出问题的时候，去分析当时启动的目录多少，执行命名如何，参数如何，是否有缺失等。 这样有利于你真正开发中去解决那些奇奇怪怪的可能和环境相关的问题。

在这里需要掌握的知识有：   
javac 编译java文件为 class 文件java 命令的使用， 带package的java类如何在命令行中启动java程序涉及到的各个路径(classpath， java。library。path， java运行的主目录等)

**3.数据类型**   
这条没有什么好多说的，无非就是Java中的基本类型和对象类型的掌握。可以再了解一些JDK如何自动转换方面的知识，包括装箱拆箱等，还要注意避免装箱之后的类型相等的判断   
主要知识点：   
基本类型： int， long， float， double， boolean ······   
对应的对象类型： Integer 等类型到基本类型的转换， 装箱和拆箱Object类型： equals， hashcodeString 类型的特点

**4.对象和实例，对象的创建**   
在这方面，开发者需要了解class和instance的概念以及之间的差别， 这是java面向对象特性的一个基础。

主要知识点有：Class和 Instance 的概念；   
Instance 创建的过程：1. 无继承：分配内存空间， 初始化变量， 调用构造函数；2.有继承：处理静态动作， 分配内存空间， 变量定义为初始值 ， 从基类->子类， 处理定义处的初始化， 执行构造方法；   
需要注意的点：静态属性等从基类->子类进行初始化；默认无参构造方法相关的特性。

**5.访问控制**   
这也是java封装特性的一个基础需要掌握的有：public protected default private 对于class， method， field 的修饰作用

**6.流程控制**   
Java 流程控制的基础，虽然有些语法不一定很常用，但是都需要了解，并且在合适的地方使用它们。需要掌握的有：   
if， switch， loop， for， while 等流程控制的语法

**7.面向对象编程的概念**   
这是一个java的核心概念，对于任何java开发者都需要熟练掌握。Java中很多特性或者说知识点都是和java面向对象编程概念相关的。在我的理解， 一个好的开发者不仅仅需要了解这些特性(知识点)本身，也更需要知道这些对象在java的面向对象编程概念中是如何体现出来的，这样更有利于开发者掌握 java这门开发语言，以及其他面向对象编程的语言。

主要的知识点包括有：面向对象三大特性：封装，继承，多态; 各自的定义概念，有哪些特性体现出 来，各自的使用场景静态多分派，动态单分派的概念重载的概念和使用继承：接口多实现，基类单继承抽象，抽象类，接口多态：方法覆盖的概念和使用接口回调

**8.Static**   
静态属性在java日常开发中也是经常使用，需要了解和 static 关键字相关的用法，还有和其他关键字的配合使用， 如是否可以和 abstract， final 等关键字联合使用。   
主要需要掌握的有：静态属性的定义，使用，以及类加载时如何初始化静态方法的定义和使用静态类的定义和使用静态代码块的定义和初始化时机

**9.基础知识点**   
这里主要罗列一些散落的，没有系统归类的一些java知识点。在日常的开发中用到也不少。

暂时归纳包括有：   
equals ， hashcode ， string/stringbuffer ，final ， finally ， finalize

**10.集合框架**   
这个是一个需要多加掌握的部分，做java开发，可以说没有不用到集合框架的，这很重要。但是这里的知识点并不难，但是对于集合最好要了解内部的实现方式，因为这样有助于你在各个不同的场景选择适合的框架来解决问题，比如有1W个元素的集合，经常要进行contains判断操作，知道了集合的特性或者内部实现，就很容易做出正确的选择。   
这里包括了如下内容(并发相关不包含在内)：集合框架的体系： 基础Collection ，Map具体集合实现的内容， List ，Set ，Map 具体的实现，内部结构， 特殊的方法， 适用场景等集合相关的工具类 Collections 等的用法

**11.异常框架**   
异常在java的开发中可能没有那么被重视。一般遇到异常，直接上抛，或者随便catch一下处理之后对于程序整体运行也没有什么大的影响。不过在企业级设计开发中，异常的设计与处理的好坏，往往就关系着这个系统整体的健壮性。一个好的系统的异常对于开发者来说，处理应该统一，避免各处散落很多异常处理逻辑;对于系统 来说，异常应该是可控的，并且是易于运维的，某些异常出现后，应该有应对的方法，知道如何运维处理，所以虽然异常框架很简单，但是对于整个企业级应用开发来说，异常处理是很重要的，处理好异常就需要了解Java中的异常体系。

这部分需要掌握的知识点主要是：异常的体系：ThrowableExceptionRuntimeExceptionErrorRuntimeException 和 一般 Exception 的区别， 具体处理方法等

**12.Java IO**   
IO 在java中不仅仅是文件读写那么简单，也包括了 socket 网络的读写等等一切的输入输出操作。比如说 标准HTTP请求中Post的内容的读取也是一个输出的过程，等等…

对于IO，Java不仅提供了基本Input、Output相关的api，也提供了一些简化操作的Reader、Writer等api，在某些开发(涉及大量IO操作的项目)中也很重要，一般日常的开发中也会涉及(日志，临时文件的读写等)。

在这中的知识点主要有：基本IO的体系： 包括有InputStream ， OutputStream， Reader/Writer， 文件读取，各种流读取等NIO 的概念， 具体使用方式和使用场景

**13.多线程并发**   
多线程是Java中普遍认为比较难的一块。多线程用好了可以有效提高cpu使用率， 提升整体系统效率， 特别是在有大量IO操作阻塞的情况下;但是它也是一柄双刃剑， 如果用不好，系统非但提升不大，或者没有提升，而且还会带来多线程之间的调试时等问题。在多线程中内容有很多，只是简单说明一下Java中初步使用多线程需要掌握的知识点，以后有机会单独再详细介绍一些高级特性的使用场景。

需要了解的知识点：   
多线程的实现和启动callable 与 runable 区别syncrhoized ，reentrantLock 各自特点和比对线程池future 异步方式获取执行结果concurrent 包lock

**14.网络**   
Java 中也是提供了可以直接操作 TCP协议、UDP协议的API。在需要强调网络性能的情况下，可以直接使用TCP/UDP 进行通讯。在查看Tomcat等的源码中，就可以看到这些相关API的使用情况。不过一般也比较少会直接使用TCP，会使用诸如MINA、Netty这样 的框架来进行处理，因为这个方面的开发涉及不多，所以就不再详细罗列了。

**15.时间日期处理**   
几乎对于每个应用来说，时间日期的处理也是绕不过去的，但是JDK8 之前的时间相关API用法并不友好。在那个时代，可以选择Joda等时间框架。到了JDK8 发布之后，全新的时间API基本融合了其他框架的优点，已经可以很好的直接使用了。

对于Java开发者来说，需要熟练地使用API来对时间和日期做相关的处理。

具体知识点不再罗列，会在以后再写个专门的文章来总结一下JDK8中时间日期API的用法

**16.XML解析/ JSON解析**   
其实这两块内容都不是J2SE里面的内容，但是在日常开发中，和其他程序交互，和配置文件交互，越来越离不开这两种格式的解析。

不过对于一个开发者来说，能够了解一些XML/JSON具体解析的原理和方法，有助于你在各个具体的场景中更好的选择合适你的方式来使得你的程序更有效率和更加健壮。

XML： 需要了解 DOM解析和 SAX解析的基本原理和各自的适用场景JSON： 需要了解一些常用JSON框架的用法， 如 Jackson， FastJson， Gson 等。

**17.Maven的使用**   
Maven 也不是Java里面的内容，但是maven是革命性的，给java开发带来了巨大的便利。从依赖的引入和管理，开发流程的更新和发布产出，乃至版本的更 新，使用maven可以大大简化开发过程中的复杂度，从而节省大量时间。可以说，maven已经成为java开发者的标配了。所以我把maven也作为一 个java开发者对于基础必备的知识点。以后会再放上一些我的一些对于maven使用的经验和技巧等，这里就不再细说了。（Java学习交流QQ群：589809992 我们一起学Java！）

**18.泛型**   
这是JDK5开始引入的新概念，其实是个语法糖，在编写java代码时会有些许便利， 一般的应用或者是业务的开发，只需要简单使用，不一定会用到定义泛型这样的操作， 但是开发一些基础公共组件会使用到，可以在需要的时候再细看这个部分，一般情况下只要会简单使用即可。

**19.标注**   
也是jdk5 之后引入的。spring是个优秀的框架，最开始就以xml作为标准的配置文件。不过到了Spring3 之后，尤其是spring-boot 兴起之后，越来越推崇使用标注来简化xml配置文件了，对于开发者来说，可以节省不少xml配置的时间。但是劣势是在于标注散落在各个类中，不像xml， 可以对所有配置有个全局性的理解和管理，所以还没有办法说完全就取代所有的xml。对于一般开发者，会使用标注即可，一些公共组建的开发者可能会需要了解 标注的定义和实现，可以在具体需要的时候再细看。

**20.RMI**   
RemoteMethodInvocation ，Java语言特有的远程调用接口，使用还是比较简单方便。不过需要跨语言的情况下，就需要使用 webservice 等其他方式来支持。一般来说，程序都不需要使用RMI，不过可以在特定的情况下使用，我就在一个项目中，使用RMI来进行程序远程启动停止的控制。

**21.JNI**   
Java Native Interface，可以允许Java中调用本地接口方法，一般用于C/C++代码的调用。需要注意的是在java中加载so/dll文件的路径问题，本身调用接口并不复杂，但是经常在是否加载了所需的本地接口库中花费较多时间

# Java的21个核心技术点，你知道吗？

[赛阳软件培训中心](https://www.zhihu.com/people/sai-yang-ruan-jian-pei-xun-zhong-xin)[赛阳软件培训中心](https://www.zhihu.com/people/sai-yang-ruan-jian-pei-xun-zhong-xin)

2 个月前

写这篇文章的目的是想总结一下自己这么多年来使用java的一些心得体会，主要是和一些[java基础](http://link.zhihu.com/?target=http%3A//bbs.it-home.org/forum-java-1.html)知识点相关的，所以也希望能分享给刚刚入门的[java程序](http://link.zhihu.com/?target=http%3A//bbs.it-home.org/forum-java-1.html)员和打算入[java开发](http://link.zhihu.com/?target=http%3A//bbs.it-home.org/forum-java-1.html)这个行当的准新手们，希望可以给大家一些经验，能让大家更好学习和使用Java。

　　这次介绍的主要内容是和J2SE相关的部分，另外，会在以后再介绍些J2EE相关的、和Java中各个框架相关的内容。

　　经过这么多年的Java开发，以及结合平时面试Java开发者的一些经验，我觉得对于J2SE方面主要就是要掌握以下的一些内容。

**1. JVM相关(包括了各个版本的特性)**

　　对于刚刚接触Java的人来说，JVM相关的知识不一定需要理解很深，对此里面的概念有一些简单的了解即可。不过对于一个有着3年以上Java经验的资深开发者来说，不会JVM几乎是不可接受的。

　　JVM作为java运行的基础，很难相信对于JVM一点都不了解的人可以把[java语言](http://link.zhihu.com/?target=http%3A//bbs.it-home.org/forum-java-1.html)吃得很透。我在面试有超过3年Java经验的开发者的时候， JVM几乎就是一个必问的问题了。当然JVM不是唯一决定技术能力好坏的面试问题，但是可以佐证java开发能力的高低。

　　在JVM这个大类中，我认为需要掌握的知识有：

JVM内存模型和结构

GC原理，性能调优

调优：Thread Dump， 分析内存结构

class 二进制字节码结构， class loader 体系 ， class加载过程 ， 实例创建过程

方法执行过程

Java各个大版本更新提供的新特性(需要简单了解)

**2. Java的运行（基础必备）**

　　这条可能出看很简单，java程序的运行谁不会呢？不过很多时候， 我们只是单纯通过IDE去执行java程序，底层IDE又是如何执行java程序呢？很多人并不了解。

　　这个知识点是最最基本的java开发者需要掌握的，初学java，第一个肯定是教你如何在命令行中执行java程序，但是很多人一旦把java学完了，IDE用上了，就把这个都忘了。为什么强调要知道这个呢，知道了java最纯粹的启动方式之后，你才能在启动出问题的时候，去分析当时启动的目录多少，执行命名如何，参数如何，是否有缺失等。 这样有利于你真正开发中去解决那些奇奇怪怪的可能和环境相关的问题。

　　在这里需要掌握的知识有：

javac 编译java文件为 class 文件

java 命令的使用， 带package的java类如何在命令行中启动

java程序涉及到的各个路径(classpath， java。library。path， java运行的主目录等)

**3. 数据类型**

　　这条没有什么好多说的，无非就是Java中的基本类型和对象类型的掌握。可以再了解一些JDK如何自动转换方面的知识，包括装箱拆箱等，还要注意避免装箱之后的类型相等的判断

　　主要知识点：

基本类型： int， long， float， double， boolean ， 。。。

对应的对象类型： Integer 等类型到基本类型的转换， 装箱和拆箱

Object类型： equals， hashcode

String 类型的特点

**4. 对象和实例，对象的创建**

　　在这方面，开发者需要了解class和instance的概念以及之间的差别， 这是java面向对象特性的一个基础。主要知识点有：

Class和 Instance 的概念

Instance 创建的过程：

　　1。 无继承：分配内存空间， 初始化变量， 调用构造函数

　　2。 有继承：处理静态动作， 分配内存空间， 变量定义为初始值 ， 从基类->子类， 处理定义处的初始化， 执行构造方法

　　需要注意的点：

　　　　静态属性等从基类->子类进行初始化

　　　　默认无参构造方法相关的特性

**5. 访问控制**

　　这也是java封装特性的一个基础，需要掌握的有：

public protected default private 对于class， method， field 的修饰作用

**6. 流程控制**

　　Java 流程控制的基础， 虽然有些语法不一定很常用，但是都需要了解，并且在合适的地方使用它们。

需要掌握的有：

if， switch， loop， for， while 等流程控制的语法

**7. 面向对象编程的概念**

　　这是一个java的核心概念，对于任何java开发者都需要熟练掌握。Java中很多特性或者说知识点都是和java面向对象编程概念相关的。在我的理解，一个好的开发者不仅仅需要了解这些特性（知识点）本身，也更需要知道这些对象在java的面向对象编程概念中是如何体现出来的，这样更有利于开发者掌握java这门开发语言，以及其他面向对象编程的语言。在这里只是简单罗列了一下，主要的知识点包括有：

面向对象三大特性：封装，继承，多态; 各自的定义概念，有哪些特性体现出来，各自的使用场景

静态多分派，动态单分派的概念

重载的概念和使用

继承：接口多实现，基类单继承

抽象，抽象类，接口

多态：方法覆盖的概念和使用

接口回调

**8. Static**

　　静态属性在java日常开发中也是经常使用，需要了解和 static 关键字相关的用法，还有和其他关键字的配合使用， 如是否可以和 abstract， final 等关键字联合使用。

主要需要掌握的有：

静态属性的定义，使用，以及类加载时如何初始化

静态方法的定义和使用

静态类的定义和使用

静态代码块的定义和初始化时机

**9. 基础知识点**

　　这里主要罗列一些散落的，没有系统归类的一些java知识点。在日常的开发中用到也不少。 这块内容其实还有很多，目前只是暂时归纳了这几个在这里：

　　包括有：

equals ， hashcode ， string/stringbuffer ，final ， finally ， finalize

**10.集合框架**

　　这个是一个需要多加掌握的部分，做java开发，可以说没有不用到集合框架的，这很重要。但是这里的知识点并不难，但是对于集合最好要了解内部的实现方式，因为这样有助于你在各个不同的场景选择适合的框架来解决问题，比如有1W个元素的集合，经常要进行contains判断操作，知道了集合的特性或者内部实现，就很容易做出正确的选择。

　　这里包括了如下内容(并发相关不包含在内)：

集合框架的体系： 基础Collection ，Map

具体集合实现的内容， List ，Set ，Map 具体的实现，内部结构， 特殊的方法， 适用场景等

集合相关的工具类 Collections 等的用法

**11.异常框架**

　　异常在java的开发中可能没有那么被重视。一般遇到异常，直接上抛，或者随便catch一下处理之后对于程序整体运行也没有什么大的影响。不过在企业级设计开发中， 异常的设计与处理的好坏，往往就关系着这个系统整体的健壮性。一个好的系统的异常对于开发者来说，处理应该统一，避免各处散落很多异常处理逻辑；对于系统来说，异常应该是可控的，并且是易于运维的，某些异常出现后，应该有应对的方法，知道如何运维处理，所以虽然异常框架很简单，但是对于整个企业级应用开发来说，异常处理是很重要的，处理好异常就需要了解Java中的异常体系。

　　这部分需要掌握的知识点不多，主要就是：

异常的体系：

Throwable

Exception

RuntimeException

Error

RuntimeException 和 一般 Exception 的区别， 具体处理方法等

**12. Java IO**

　　IO 在java中不仅仅是文件读写那么简单，也包括了 socket 网络的读写等等一切的输入输出操作。比如说 标准HTTP请求中Post的内容的读取也是一个输出的过程，等等…

　　对于IO，Java不仅提供了基本Input、Output相关的api，也提供了一些简化操作的Reader、Writer等api，在某些开发（涉及大量IO操作的项目）中也很重要，一般日常的开发中也会涉及（日志，临时文件的读写等）。

　　在这中的知识点主要有：

基本IO的体系： 包括有InputStream ， OutputStream， Reader/Writer， 文件读取，各种流读取等

NIO 的概念， 具体使用方式和使用场景

**13.多线程并发**

　　多线程是Java中普遍认为比较难的一块。多线程用好了可以有效提高cpu使用率， 提升整体系统效率， 特别是在有大量IO操作阻塞的情况下；但是它也是一柄双刃剑， 如果用不好，系统非但提升不大，或者没有提升，而且还会带来多线程之间的调试时等问题。

　　在多线程中内容有很多，只是简单说明一下Java中初步使用多线程需要掌握的知识点，以后有机会单独再详细介绍一些高级特性的使用场景。

多线程的实现和启动

callable 与 runable 区别

syncrhoized ，reentrantLock 各自特点和比对

线程池

future 异步方式获取执行结果

concurrent 包

lock

..

**14.网络**

　　Java中也是提供了可以直接操作 TCP协议、UDP协议的API。在需要强调网络性能的情况下，可以直接使用TCP/UDP 进行通讯。在查看Tomcat等的源码中，就可以看到这些相关API的使用情况。不过一般也比较少会直接使用TCP，会使用诸如MINA、Netty这样的框架来进行处理，因为这个方面的开发涉及不多，所以就不再详细罗列了。

**15.时间日期处理**

　　几乎对于每个应用来说，时间日期的处理也是绕不过去的，但是JDK8 之前的时间相关API用法并不友好。在那个时代，可以选择Joda等时间框架。到了JDK8 发布之后，全新的时间API基本融合了其他框架的优点，已经可以很好的直接使用了。

　　对于Java开发者来说，需要熟练地使用API来对时间和日期做相关的处理。

　　具体知识点不再罗列，会在以后再写个专门的文章来总结一下JDK8中时间日期API的用法。

**16.XML解析/ JSON解析**

　　其实这两块内容都不是J2SE里面的内容，但是在日常开发中，和其他程序交互，和配置文件交互，越来越离不开这两种格式的解析。

　　不过对于一个开发者来说，能够了解一些XML/JSON具体解析的原理和方法，有助于你在各个具体的场景中更好的选择合适你的方式来使得你的程序更有效率和更加健壮。

　　XML： 需要了解 DOM解析和 SAX解析的基本原理和各自的适用场景

　　JSON： 需要了解一些常用JSON框架的用法， 如 Jackson， FastJson， Gson 等。。

**17.Maven的使用**

　　Maven也不是Java里面的内容，但是maven是革命性的，给java开发带来了巨大的便利。从依赖的引入和管理，开发流程的更新和发布产出，乃至版本的更新，使用maven可以大大简化开发过程中的复杂度，从而节省大量时间。可以说，maven已经成为java开发者的标配了。所以我把maven也作为一个java开发者对于基础必备的知识点。以后会再放上一些我的一些对于maven使用的经验和技巧等，这里就不再细说了。

**18.泛型**

　　这是JDK5开始引入的新概念，其实是个语法糖，在编写[java代码](http://link.zhihu.com/?target=http%3A//bbs.it-home.org/forum-java-1.html)时会有些许便利， 一般的应用或者是业务的开发，只需要简单使用，不一定会用到定义泛型这样的操作， 但是开发一些基础公共组件会使用到，可以在需要的时候再细看这个部分，一般情况下只要会简单使用即可。

**19.标注**

　　也是jdk5 之后引入的。spring是个优秀的框架，最开始就以xml作为标准的配置文件。不过到了Spring3 之后，尤其是 spring-boot 兴起之后，越来越推崇使用标注来简化xml配置文件了，对于开发者来说，可以节省不少xml配置的时间。但是劣势是在于标注散落在各个类中，不像xml，可以对所有配置有个全局性的理解和管理，所以还没有办法说完全就取代所有的xml。对于一般开发者，会使用标注即可，一些公共组建的开发者可能会需要了解标注的定义和实现，可以在具体需要的时候再细看。

**20.RMI**

　　RemoteMethodInvocation ，Java语言特有的远程调用接口，使用还是比较简单方便。不过需要跨语言的情况下，就需要使用 webservice 等其他方式来支持。一般来说，程序都不需要使用RMI，不过可以在特定的情况下使用，我就在一个项目中，使用RMI来进行程序远程启动停止的控制。

**21.JNI**

　　Java Native Interface，可以允许Java中调用本地接口方法，一般用于C/C++代码的调用。需要注意的是在java中加载so/dll文件的路径问题，本身调用接口并不复杂，但是经常在是否加载了所需的本地接口库中花费较多时间。

**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？**

**2. 谈一谈”==“与”equals()"的区别。**

**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？**

**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<K, V>**:  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

更加详细的信息请点击“阅读原文”

**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”。更加详细的分析请点击“阅读原文”。

**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底层实现的更加详细分析请点击“阅读原文”。

**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能够记住插入元素的顺序。

**13. Collection与Collections的区别是什么？**

* Collection是Java集合框架中最基本的接口；
* Collections是Java集合框架提供的一个工具类，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？**

答案是会执行。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**

\*\*

\*\*

更加详细的说明请点击“阅读原文”。

**16. Java面向对象的三个特征与含义**

**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。关于内部类的详细介绍请点击“阅读原文”。

**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”。详细介绍请点击“阅读原文”。

**21. 简述Java中创建新线程的两种方法**

* 继承Thread类（假设为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。

更加详细的介绍请点击“阅读原文”。

**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。更加详细的介绍请点击“阅读原文”。

**24. 给出“生产者-消费者”问题的一种解决方案**

**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。

关于线程池的详细介绍以及实现原理分析请点击“阅读原文”。

**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); }

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“阅读原文”

**34. Java 1.7与1.8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。

关于每个模式具体的介绍请点击“阅读原文”。

**36. JNI的基本用法**

**37. 动态代理的定义、应用场景及原理**

**38. 注解的基本概念与使用**

* 注解可以看作是“增强版的注释”，它可以向编译器、虚拟机说明一些事情。
* 注解是描述Java代码的代码，它能够被编译器解析，注解处理工具在运行时也能够解析注解。注解本身是“被动”的信息，只有主动解析它才有意义。
* 除了向编译器/虚拟机传递信息，我们也可以使用注解来生成一些“模板化”的代码。

**1、面向对象的特征有哪些方面？**   
答：面向对象的特征主要有以下几个方面：   
- 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。   
- 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段（如果不能理解请阅读阎宏博士的《Java与模式》或《设计模式精解》中关于桥梁模式的部分）。   
- 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是封装得足够好的，因为几个按键就搞定了所有的事情）。   
- 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当A系统访问B系统提供的服务时，B系统有多种提供服务的方式，但一切对A系统来说都是透明的（就像电动剃须刀是A系统，它的供电系统是B系统，B系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A系统只会通过B类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2). 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

**2、访问修饰符public,private,protected,以及不写（默认）时的区别？**   
答：

| **修饰符** | **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。

**3、String 是最基本的数据类型吗？**   
答：不是。Java中的基本数据类型只有8个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（reference type），Java 5以后引入的枚举类型也算是一种比较特殊的引用类型。

**4、float f=3.4;是否正确？**   
答: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（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

**5、short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗？**   
答：对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。而short s1 = 1; s1 += 1;可以正确编译，因为s1+= 1;相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

**6、Java有没有goto？**   
答：goto 是Java中的保留字，在目前版本的Java中没有使用。（根据James Gosling（Java之父）编写的《The Java Programming Language》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）

**7、int和Integer有什么区别？**   
答：Java是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。   
Java 为每个原始类型提供了包装类型：   
- 原始类型: boolean，char，byte，short，int，long，float，double   
- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

class AutoUnboxingTest {

public static void main(String[] args) {

Integer a = new Integer(3);

Integer b = 3; // 将3自动装箱成Integer类型

int c = 3;

System.out.println(a == b); // false 两个引用没有引用同一对象

System.out.println(a == c); // true a自动拆箱成int类型再和c比较

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

最近还遇到一个面试题，也是和自动装箱和拆箱有点关系的，代码如下所示：

public class Test03 {

public static void main(String[] args) {

Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;

System.out.println(f1 == f2);

System.out.println(f3 == f4);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是Integer对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。

public static Integer valueOf(int i) {

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

* 1
* 2
* 3
* 4
* 5

IntegerCache是Integer的内部类，其代码如下所示：

/\*\*

\* Cache to support the object identity semantics of autoboxing for values between

\* -128 and 127 (inclusive) as required by JLS.

\*

\* The cache is initialized on first usage. The size of the cache

\* may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.

\* During VM initialization, java.lang.Integer.IntegerCache.high property

\* may be set and saved in the private system properties in the

\* sun.misc.VM class.

\*/

private static class IntegerCache {

static final int low = -128;

static final int high;

static final Integer cache[];

static {

// high value may be configured by property

int h = 127;

String integerCacheHighPropValue =

sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");

if (integerCacheHighPropValue != null) {

try {

int i = parseInt(integerCacheHighPropValue);

i = Math.max(i, 127);

// Maximum array size is Integer.MAX\_VALUE

h = Math.min(i, Integer.MAX\_VALUE - (-low) -1);

} catch( NumberFormatException nfe) {

// If the property cannot be parsed into an int, ignore it.

}

}

high = h;

cache = new Integer[(high - low) + 1];

int j = low;

for(int k = 0; k < cache.length; k++)

cache[k] = new Integer(j++);

// range [-128, 127] must be interned (JLS7 5.1.7)

assert IntegerCache.high >= 127;

}

private IntegerCache() {}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44

简单的说，如果整型字面量的值在-128到127之间，那么不会new新的Integer对象，而是直接引用常量池中的Integer对象，所以上面的面试题中f1==f2的结果是true，而f3==f4的结果是false。

**提醒：**越是貌似简单的面试题其中的玄机就越多，需要面试者有相当深厚的功力。

**8、&和&&的区别？**   
答：&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。

**补充：**如果你熟悉JavaScript，那你可能更能感受到短路运算的强大，想成为JavaScript的高手就先从玩转短路运算开始吧。

**9、解释内存中的栈(stack)、堆(heap)和方法区(method area)的用法。**   
答：通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用JVM中的栈空间；而通过new关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为Eden、Survivor（又可分为From Survivor和To Survivor）、Tenured；方法区和堆都是各个线程共享的内存区域，用于存储已经被JVM加载的类信息、常量、静态变量、JIT编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的100、"hello"和常量都是放在常量池中，常量池是方法区的一部分，。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过JVM的启动参数来进行调整，栈空间用光了会引发StackOverflowError，而堆和常量池空间不足则会引发OutOfMemoryError。

String str = new String("hello");

* 1

上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而"hello"这个字面量是放在方法区的。

**补充1：**较新版本的Java（从Java 6的某个更新开始）中，由于JIT编译器的发展和"逃逸分析"技术的逐渐成熟，栈上分配、标量替换等优化技术使得对象一定分配在堆上这件事情已经变得不那么绝对了。

**补充2**：运行时常量池相当于Class文件常量池具有动态性，Java语言并不要求常量一定只有编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的。

看看下面代码的执行结果是什么并且比较一下Java 7以前和以后的运行结果是否一致。

String s1 = new StringBuilder("go")

.append("od").toString();

System.out.println(s1.intern() == s1);

String s2 = new StringBuilder("ja")

.append("va").toString();

System.out.println(s2.intern() == s2);

* 1
* 2
* 3
* 4
* 5
* 6

**10、Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？**   
答：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。

**11、switch 是否能作用在byte 上，是否能作用在long 上，是否能作用在String上？**   
答：在Java 5以前，switch(expr)中，expr只能是byte、short、char、int。从Java 5开始，Java中引入了枚举类型，expr也可以是enum类型，从Java 7开始，expr还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

**12、用最有效率的方法计算2乘以8？**   
答： 2 << 3（左移3位相当于乘以2的3次方，右移3位相当于除以2的3次方）。

**补充：**我们为编写的类重写hashCode方法时，可能会看到如下所示的代码，其实我们不太理解为什么要使用这样的乘法运算来产生哈希码（散列码），而且为什么这个数是个素数，为什么通常选择31这个数？前两个问题的答案你可以自己百度一下，选择31是因为可以用移位和减法运算来代替乘法，从而得到更好的性能。说到这里你可能已经想到了：31 \* num 等价于(num << 5) - num，左移5位相当于乘以2的5次方再减去自身就相当于乘以31，现在的VM都能自动完成这个优化。

public class PhoneNumber {

private int areaCode;

private String prefix;

private String lineNumber;

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + areaCode;

result = prime \* result

+ ((lineNumber == null) ? 0 : lineNumber.hashCode());

result = prime \* result + ((prefix == null) ? 0 : prefix.hashCode());

return result;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

PhoneNumber other = (PhoneNumber) obj;

if (areaCode != other.areaCode)

return false;

if (lineNumber == null) {

if (other.lineNumber != null)

return false;

} else if (!lineNumber.equals(other.lineNumber))

return false;

if (prefix == null) {

if (other.prefix != null)

return false;

} else if (!prefix.equals(other.prefix))

return false;

return true;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

**13、数组有没有length()方法？String有没有length()方法？**   
答：数组没有length()方法，有length 的属性。String 有length()方法。JavaScript中，获得字符串的长度是通过length属性得到的，这一点容易和Java混淆。

**14、在Java中，如何跳出当前的多重嵌套循环？**   
答：在最外层循环前加一个标记如A，然后用break A;可以跳出多重循环。（Java中支持带标签的break和continue语句，作用有点类似于C和C++中的goto语句，但是就像要避免使用goto一样，应该避免使用带标签的break和continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用，所以这种语法其实不知道更好）

**15、构造器（constructor）是否可被重写（override）？**   
答：构造器不能被继承，因此不能被重写，但可以被重载。

**16、两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？**   
答：不对，如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code）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）。

**补充：**关于equals和hashCode方法，很多Java程序都知道，但很多人也就是仅仅知道而已，在Joshua Bloch的大作《Effective Java》（很多软件公司，《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍，如果你还没看过，那就赶紧去[亚马逊](http://z.cn/)买一本吧）中是这样介绍equals方法的：首先equals方法必须满足自反性（x.equals(x)必须返回true）、对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）、传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）和一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。实现高质量的equals方法的诀窍包括：1. 使用==操作符检查"参数是否为这个对象的引用"；2. 使用instanceof操作符检查"参数是否为正确的类型"；3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；5. 重写equals时总是要重写hashCode；6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

**17、是否可以继承String类？**   
答：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）。

**18、当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？**   
答：是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码，但是在Java中却做不到。

using System;

namespace CS01 {

class Program {

public static void swap(ref int x, ref int y) {

int temp = x;

x = y;

y = temp;

}

public static void Main (string[] args) {

int a = 5, b = 10;

swap (ref a, ref b);

// a = 10, b = 5;

Console.WriteLine ("a = {0}, b = {1}", a, b);

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

说明：Java中没有传引用实在是非常的不方便，这一点在Java 8中仍然没有得到改进，正是如此在Java编写的代码中才会出现大量的Wrapper类（将需要通过方法调用修改的引用置于一个Wrapper类中，再将Wrapper对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从C和C++转型为Java程序员的开发者无法容忍。

**19、String和StringBuilder、StringBuffer的区别？**   
答：Java平台提供了两种类型的字符串：String和StringBuffer/StringBuilder，它们可以储存和操作字符串。其中String是只读字符串，也就意味着String引用的字符串内容是不能被改变的。而StringBuffer/StringBuilder类表示的字符串对象可以直接进行修改。StringBuilder是Java 5中引入的，它和StringBuffer的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。

**面试题1** - 什么情况下用+运算符进行字符串连接比调用StringBuffer/StringBuilder对象的append方法连接字符串性能更好？

**面试题2** - 请说出下面程序的输出。

class StringEqualTest {

public static void main(String[] args) {

String s1 = "Programming";

String s2 = new String("Programming");

String s3 = "Program";

String s4 = "ming";

String s5 = "Program" + "ming";

String s6 = s3 + s4;

System.out.println(s1 == s2);

System.out.println(s1 == s5);

System.out.println(s1 == s6);

System.out.println(s1 == s6.intern());

System.out.println(s2 == s2.intern());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

补充：解答上面的面试题需要清除两点：1. String对象的intern方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与String对象的equals结果是true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；2. 字符串的+操作其本质是创建了StringBuilder对象进行append操作，然后将拼接后的StringBuilder对象用toString方法处理成String对象，这一点可以用javap -c StringEqualTest.class命令获得class文件对应的JVM字节码指令就可以看出来。

**20、重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？**   
答：方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

**面试题：**华为的面试题中曾经问过这样一个问题 - "为什么不能根据返回类型来区分重载"，快说出你的答案吧！

**21、描述一下JVM加载class文件的原理机制？**   
答：JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。   
由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。   
类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM更好的保证了Java平台的安全性，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明：

* Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；
* Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；
* System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。

**22、char 型变量中能不能存贮一个中文汉字，为什么？**   
答：char类型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。

**补充：**使用Unicode意味着字符在JVM内部和外部有不同的表现形式，在JVM内部都是Unicode，当这个字符被从JVM内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以Java中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如InputStreamReader和OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于C程序员来说，要完成这样的编码转换恐怕要依赖于union（联合体/共用体）共享内存的特征来实现了。

**23、抽象类（abstract class）和接口（interface）有什么异同？**   
答：抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是private、默认、protected、public的，而接口中的成员全都是public的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

**24、静态嵌套类(Static Nested Class)和内部类（Inner Class）的不同？**   
答：Static Nested Class是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示。

/\*\*

\* 扑克类（一副扑克）

\* @author 骆昊

\*

\*/

public class Poker {

private static String[] suites = {"黑桃", "红桃", "草花", "方块"};

private static int[] faces = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

private Card[] cards;

/\*\*

\* 构造器

\*

\*/

public Poker() {

cards = new Card[52];

for(int i = 0; i < suites.length; i++) {

for(int j = 0; j < faces.length; j++) {

cards[i \* 13 + j] = new Card(suites[i], faces[j]);

}

}

}

/\*\*

\* 洗牌 （随机乱序）

\*

\*/

public void shuffle() {

for(int i = 0, len = cards.length; i < len; i++) {

int index = (int) (Math.random() \* len);

Card temp = cards[index];

cards[index] = cards[i];

cards[i] = temp;

}

}

/\*\*

\* 发牌

\* @param index 发牌的位置

\*

\*/

public Card deal(int index) {

return cards[index];

}

/\*\*

\* 卡片类（一张扑克）

\* [内部类]

\* @author 骆昊

\*

\*/

public class Card {

private String suite; // 花色

private int face; // 点数

public Card(String suite, int face) {

this.suite = suite;

this.face = face;

}

@Override

public String toString() {

String faceStr = "";

switch(face) {

case 1: faceStr = "A"; break;

case 11: faceStr = "J"; break;

case 12: faceStr = "Q"; break;

case 13: faceStr = "K"; break;

default: faceStr = String.valueOf(face);

}

return suite + faceStr;

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75

测试代码：

class PokerTest {

public static void main(String[] args) {

Poker poker = new Poker();

poker.shuffle(); // 洗牌

Poker.Card c1 = poker.deal(0); // 发第一张牌

// 对于非静态内部类Card

// 只有通过其外部类Poker对象才能创建Card对象

Poker.Card c2 = poker.new Card("红心", 1); // 自己创建一张牌

System.out.println(c1); // 洗牌后的第一张

System.out.println(c2); // 打印: 红心A

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

面试题 - 下面的代码哪些地方会产生编译错误？

class Outer {

class Inner {}

public static void foo() { new Inner(); }

public void bar() { new Inner(); }

public static void main(String[] args) {

new Inner();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

注意：Java中非静态内部类对象的创建要依赖其外部类对象，上面的面试题中foo和main方法都是静态方法，静态方法中没有this，也就是说没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中创建内部类对象，可以这样做：

new Outer().new Inner();

* 1

**25、Java 中会存在内存泄漏吗，请简单描述。**   
答：理论上Java因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被GC回收，因此也会导致内存泄露的发生。例如Hibernate的Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

import java.util.Arrays;

import java.util.EmptyStackException;

public class MyStack<T> {

private T[] elements;

private int size = 0;

private static final int INIT\_CAPACITY = 16;

public MyStack() {

elements = (T[]) new Object[INIT\_CAPACITY];

}

public void push(T elem) {

ensureCapacity();

elements[size++] = elem;

}

public T pop() {

if(size == 0)

throw new EmptyStackException();

return elements[--size];

}

private void ensureCapacity() {

if(elements.length == size) {

elements = Arrays.copyOf(elements, 2 \* size + 1);

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的pop方法却存在内存泄露的问题，当我们用pop方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成OutOfMemoryError。

**26、抽象的（abstract）方法是否可同时是静态的（static）,是否可同时是本地方法（native），是否可同时被synchronized修饰？**   
答：都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如C代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

**27、阐述静态变量和实例变量的区别。**   
答：静态变量是被static修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

**补充：**在Java开发中，上下文类和工具类中通常会有大量的静态成员。

**28、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？**   
答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

**29、如何实现对象克隆？**   
答：有两种方式：   
  1). 实现Cloneable接口并重写Object类中的clone()方法；   
  2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class MyUtil {

private MyUtil() {

throw new AssertionError();

}

@SuppressWarnings("unchecked")

public static <T extends Serializable> T clone(T obj) throws Exception {

ByteArrayOutputStream bout = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(bout);

oos.writeObject(obj);

ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());

ObjectInputStream ois = new ObjectInputStream(bin);

return (T) ois.readObject();

// 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义

// 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

下面是测试代码：

import java.io.Serializable;

/\*\*

\* 人类

\* @author 骆昊

\*

\*/

class Person implements Serializable {

private static final long serialVersionUID = -9102017020286042305L;

private String name; // 姓名

private int age; // 年龄

private Car car; // 座驾

public Person(String name, int age, Car car) {

this.name = name;

this.age = age;

this.car = car;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Car getCar() {

return car;

}

public void setCar(Car car) {

this.car = car;

}

@Override

public String toString() {

return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50

/\*\*

\* 小汽车类

\* @author 骆昊

\*

\*/

class Car implements Serializable {

private static final long serialVersionUID = -5713945027627603702L;

private String brand; // 品牌

private int maxSpeed; // 最高时速

public Car(String brand, int maxSpeed) {

this.brand = brand;

this.maxSpeed = maxSpeed;

}

public String getBrand() {

return brand;

}

public void setBrand(String brand) {

this.brand = brand;

}

public int getMaxSpeed() {

return maxSpeed;

}

public void setMaxSpeed(int maxSpeed) {

this.maxSpeed = maxSpeed;

}

@Override

public String toString() {

return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38

class CloneTest {

public static void main(String[] args) {

try {

Person p1 = new Person("Hao LUO", 33, new Car("Benz", 300));

Person p2 = MyUtil.clone(p1); // 深度克隆

p2.getCar().setBrand("BYD");

// 修改克隆的Person对象p2关联的汽车对象的品牌属性

// 原来的Person对象p1关联的汽车不会受到任何影响

// 因为在克隆Person对象时其关联的汽车对象也被克隆了

System.out.println(p1);

} catch (Exception e) {

e.printStackTrace();

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

**注意：**基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

30、GC是什么？为什么要有GC？   
答：GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显示的垃圾回收调用。   
垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

**补充：**垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：   
- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。   
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。   
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM参数：

* -Xms / -Xmx — 堆的初始大小 / 堆的最大大小
* -Xmn — 堆中年轻代的大小
* -XX:-DisableExplicitGC — 让System.gc()不产生任何作用
* -XX:+PrintGCDetails — 打印GC的细节
* -XX:+PrintGCDateStamps — 打印GC操作的时间戳
* -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小
* -XX:NewRatio — 可以设置老生代和新生代的比例
* -XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布
* -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值
* -XX:TargetSurvivorRatio：设置幸存区的目标使用率

**31、String s = new String("xyz");创建了几个字符串对象？**   
答：两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象。

**32、接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？**   
答：接口可以继承接口，而且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类。

**33、一个".java"源文件中是否可以包含多个类（不是内部类）？有什么限制？**   
答：可以，但一个源文件中最多只能有一个公开类（public class）而且文件名必须和公开类的类名完全保持一致。

**34、Anonymous Inner Class(匿名内部类)是否可以继承其它类？是否可以实现接口？**   
答：可以继承其他类或实现其他接口，在Swing编程和Android开发中常用此方式来实现事件监听和回调。

**35、内部类可以引用它的包含类（外部类）的成员吗？有没有什么限制？**   
答：一个内部类对象可以访问创建它的外部类对象的成员，包括私有成员。

**36、Java 中的final关键字有哪些用法？**   
答：(1)修饰类：表示该类不能被继承；(2)修饰方法：表示方法不能被重写；(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

**37、指出下面程序的运行结果。**

class A {

static {

System.out.print("1");

}

public A() {

System.out.print("2");

}

}

class B extends A{

static {

System.out.print("a");

}

public B() {

System.out.print("b");

}

}

public class Hello {

public static void main(String[] args) {

A ab = new B();

ab = new B();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

答：执行结果：1a2b2b。创建对象时构造器的调用顺序是：先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。

**提示：**如果不能给出此题的正确答案，说明之前第21题Java类加载机制还没有完全理解，赶紧再看看吧。

**38、数据类型之间的转换：**   
**- 如何将字符串转换为基本数据类型？**   
**- 如何将基本数据类型转换为字符串？**   
答：   
- 调用基本数据类型对应的包装类中的方法parseXXX(String)或valueOf(String)即可返回相应基本类型；   
- 一种方法是将基本数据类型与空字符串（""）连接（+）即可获得其所对应的字符串；另一种方法是调用String 类中的valueOf()方法返回相应字符串

**39、如何实现字符串的反转及替换？**   
答：方法很多，可以自己写实现也可以使用String或StringBuffer/StringBuilder中的方法。有一道很常见的面试题是用递归实现字符串反转，代码如下所示：

public static String reverse(String originStr) {

if(originStr == null || originStr.length() <= 1)

return originStr;

return reverse(originStr.substring(1)) + originStr.charAt(0);

}

* 1
* 2
* 3
* 4
* 5

**40、怎样将GB2312编码的字符串转换为ISO-8859-1编码的字符串？**   
答：代码如下所示：

String s1 = "你好";

String s2 = new String(s1.getBytes("GB2312"), "ISO-8859-1");

* 1
* 2

**41、日期和时间：**   
**- 如何取得年月日、小时分钟秒？**   
**- 如何取得从1970年1月1日0时0分0秒到现在的毫秒数？**   
**- 如何取得某月的最后一天？**   
**- 如何格式化日期？**   
答：   
问题1：创建java.util.Calendar 实例，调用其get()方法传入不同的参数即可获得参数所对应的值。Java 8中可以使用java.time.LocalDateTimel来获取，代码如下所示。

public class DateTimeTest {

public static void main(String[] args) {

Calendar cal = Calendar.getInstance();

System.out.println(cal.get(Calendar.YEAR));

System.out.println(cal.get(Calendar.MONTH)); // 0 - 11

System.out.println(cal.get(Calendar.DATE));

System.out.println(cal.get(Calendar.HOUR\_OF\_DAY));

System.out.println(cal.get(Calendar.MINUTE));

System.out.println(cal.get(Calendar.SECOND));

// Java 8

LocalDateTime dt = LocalDateTime.now();

System.out.println(dt.getYear());

System.out.println(dt.getMonthValue()); // 1 - 12

System.out.println(dt.getDayOfMonth());

System.out.println(dt.getHour());

System.out.println(dt.getMinute());

System.out.println(dt.getSecond());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

问题2：以下方法均可获得该毫秒数。

Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis();

Clock.systemDefaultZone().millis(); // Java 8

* 1
* 2
* 3

问题3：代码如下所示。

Calendar time = Calendar.getInstance();

time.getActualMaximum(Calendar.DAY\_OF\_MONTH);

* 1
* 2

问题4：利用java.text.DataFormat 的子类（如SimpleDateFormat类）中的format(Date)方法可将日期格式化。Java 8中可以用java.time.format.DateTimeFormatter来格式化时间日期，代码如下所示。

import java.text.SimpleDateFormat;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.util.Date;

class DateFormatTest {

public static void main(String[] args) {

SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");

Date date1 = new Date();

System.out.println(oldFormatter.format(date1));

// Java 8

DateTimeFormatter newFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");

LocalDate date2 = LocalDate.now();

System.out.println(date2.format(newFormatter));

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

补充：Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括LocalDate、LocalTime、LocalDateTime、Clock、Instant等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。如果不理解这些内容，可以参考我的另一篇文章[《关于Java并发编程的总结和思考》](http://blog.csdn.net/jackfrued/article/details/44499227)。

**42、打印昨天的当前时刻。**   
答：

import java.util.Calendar;

class YesterdayCurrent {

public static void main(String[] args){

Calendar cal = Calendar.getInstance();

cal.add(Calendar.DATE, -1);

System.out.println(cal.getTime());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

在Java 8中，可以用下面的代码实现相同的功能。

import java.time.LocalDateTime;

class YesterdayCurrent {

public static void main(String[] args) {

LocalDateTime today = LocalDateTime.now();

LocalDateTime yesterday = today.minusDays(1);

System.out.println(yesterday);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

**43、比较一下Java和JavaSciprt。**   
答：JavaScript 与Java是两个公司开发的不同的两个产品。Java 是原Sun Microsystems公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而JavaScript是Netscape公司的产品，为了扩展Netscape浏览器的功能而开发的一种可以嵌入Web页面中运行的基于对象和事件驱动的解释性语言。JavaScript的前身是LiveScript；而Java的前身是Oak语言。   
下面对两种语言间的异同作如下比较：   
- 基于对象和面向对象：Java是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象；JavaScript是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象（Object-Based）和事件驱动（Event-Driven）的编程语言，因而它本身提供了非常丰富的内部对象供设计人员使用。   
- 解释和编译：Java的源代码在执行之前，必须经过编译。JavaScript是一种解释性编程语言，其源代码不需经过编译，由浏览器解释执行。（目前的浏览器几乎都使用了JIT（即时编译）技术来提升JavaScript的运行效率）   
- 强类型变量和类型弱变量：Java采用强类型变量检查，即所有变量在编译之前必须作声明；JavaScript中变量是弱类型的，甚至在使用变量前可以不作声明，JavaScript的解释器在运行时检查推断其数据类型。   
- 代码格式不一样。

**补充：**上面列出的四点是网上流传的所谓的标准答案。其实Java和JavaScript最重要的区别是一个是静态语言，一个是动态语言。目前的编程语言的发展趋势是函数式语言和动态语言。在Java中类（class）是一等公民，而JavaScript中函数（function）是一等公民，因此JavaScript支持函数式编程，可以使用Lambda函数和闭包（closure），当然Java 8也开始支持函数式编程，提供了对Lambda表达式以及函数式接口的支持。对于这类问题，在面试的时候最好还是用自己的语言回答会更加靠谱，不要背网上所谓的标准答案。

**44、什么时候用断言（assert）？**   
答：断言在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。一般来说，断言用于保证程序最基本、关键的正确性。断言检查通常在开发和测试时开启。为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式的值为false，那么系统会报告一个AssertionError。断言的使用如下面的代码所示：

assert(a > 0); // throws an AssertionError if a <= 0

* 1

断言可以有两种形式：   
assert Expression1;   
assert Expression1 : Expression2 ;   
Expression1 应该总是产生一个布尔值。   
Expression2 可以是得出一个值的任意表达式；这个值用于生成显示更多调试信息的字符串消息。

要在运行时启用断言，可以在启动JVM时使用-enableassertions或者-ea标记。要在运行时选择禁用断言，可以在启动JVM时使用-da或者-disableassertions标记。要在系统类中启用或禁用断言，可使用-esa或-dsa标记。还可以在包的基础上启用或者禁用断言。

**注意：**断言不应该以任何方式改变程序的状态。简单的说，如果希望在不满足某些条件时阻止代码的执行，就可以考虑用断言来阻止它。

**45、Error和Exception有什么区别？**   
答：Error表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；Exception表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

面试题：2005年摩托罗拉的面试中曾经问过这么一个问题“If a process reports a stack overflow run-time error, what’s the most possible cause?”，给了四个选项a. lack of memory; b. write on an invalid memory space; c. recursive function calling; d. array index out of boundary. Java程序在运行时也可能会遭遇StackOverflowError，这是一个无法恢复的错误，只能重新修改代码了，这个面试题的答案是c。如果写了不能迅速收敛的递归，则很有可能引发栈溢出的错误，如下所示：

class StackOverflowErrorTest {

public static void main(String[] args) {

main(null);

}

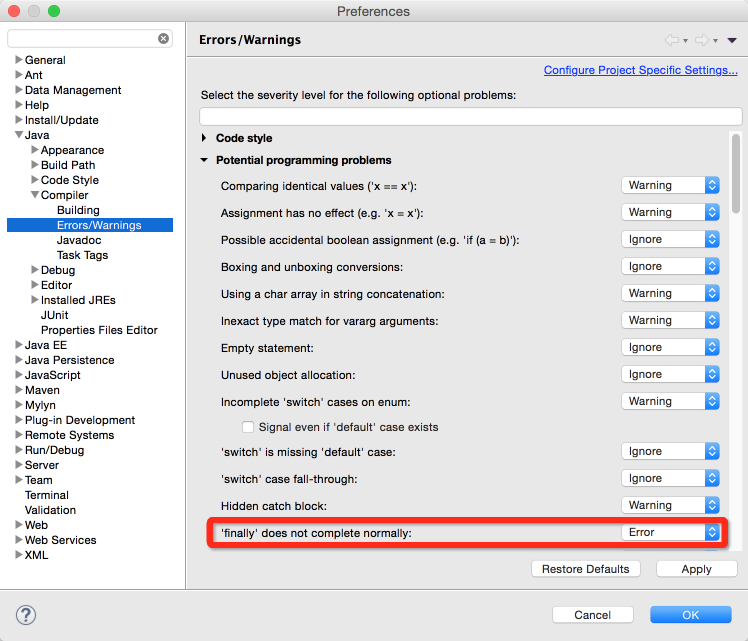
}

* 1
* 2
* 3
* 4
* 5
* 6

**提示：**用递归编写程序时一定要牢记两点：1. 递归公式；2. 收敛条件（什么时候就不再继续递归）。

**46、try{}里有一个return语句，那么紧跟在这个try后的finally{}里的代码会不会被执行，什么时候被执行，在return前还是后?**   
答：会执行，在方法返回调用者前执行。

**注意：**在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是记录下返回值待finally代码块执行完毕之后再向调用者返回其值，然后如果在finally中修改了返回值，就会返回修改后的值。显然，在finally中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java中也可以通过提升编译器的语法检查级别来产生警告或错误，Eclipse中可以在如图所示的地方进行设置，强烈建议将此项设置为编译错误。



**47、Java语言如何进行异常处理，关键字：throws、throw、try、catch、finally分别如何使用？**   
答：Java通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。在Java中，每个异常都是一个对象，它是Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。Java的异常处理是通过5个关键词来实现的：try、catch、throw、throws和finally。一般情况下是用try来执行一段程序，如果系统会抛出（throw）一个异常对象，可以通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；try用来指定一块预防所有异常的程序；catch子句紧跟在try块后面，用来指定你想要捕获的异常的类型；throw语句用来明确地抛出一个异常；throws用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；finally为确保一段代码不管发生什么异常状况都要被执行；try语句可以嵌套，每当遇到一个try语句，异常的结构就会被放入异常栈中，直到所有的try语句都完成。如果下一级的try语句没有对某种异常进行处理，异常栈就会执行出栈操作，直到遇到有处理这种异常的try语句或者最终将异常抛给JVM。

**48、运行时异常与受检异常有何异同？**   
答：异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误，只要程序设计得没有问题通常就不会发生。受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java编译器要求方法必须声明抛出可能发生的受检异常，但是并不要求必须声明抛出未被捕获的运行时异常。异常和继承一样，是面向对象程序设计中经常被滥用的东西，在Effective Java中对异常的使用给出了以下指导原则：   
- 不要将异常处理用于正常的控制流（设计良好的API不应该强迫它的调用者为了正常的控制流而使用异常）   
- 对可以恢复的情况使用受检异常，对编程错误使用运行时异常   
- 避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）   
- 优先使用标准的异常   
- 每个方法抛出的异常都要有文档   
- 保持异常的原子性   
- 不要在catch中忽略掉捕获到的异常

**49、列出一些你常见的运行时异常？**   
答：   
- ArithmeticException（算术异常）   
- ClassCastException （类转换异常）   
- IllegalArgumentException （非法参数异常）   
- IndexOutOfBoundsException （下标越界异常）   
- NullPointerException （空指针异常）   
- SecurityException （安全异常）

**50、阐述final、finally、finalize的区别。**   
答：   
- final：修饰符（关键字）有三种用法：如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。将变量声明为final，可以保证它们在使用中不被改变，被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为final的方法也同样只能使用，不能在子类中被重写。   
- finally：通常放在try…catch…的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。   
- finalize：Object类中定义的方法，Java中允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize()方法可以整理系统资源或者执行其他清理工作。

**51、类ExampleA继承Exception，类ExampleB继承ExampleA。**   
**有如下代码片断：**

try {

throw new ExampleB("b")

} catch（ExampleA e）{

System.out.println("ExampleA");

} catch（Exception e）{

System.out.println("Exception");

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**请问执行此段代码的输出是什么？**   
答：输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，抓取ExampleA类型异常的catch块能够抓住try块中抛出的ExampleB类型的异常）

**面试题** - 说出下面代码的运行结果。（此题的出处是《Java编程思想》一书）

class Annoyance extends Exception {}

class Sneeze extends Annoyance {}

class Human {

public static void main(String[] args)

throws Exception {

try {

try {

throw new Sneeze();

}

catch ( Annoyance a ) {

System.out.println("Caught Annoyance");

throw a;

}

}

catch ( Sneeze s ) {

System.out.println("Caught Sneeze");

return ;

}

finally {

System.out.println("Hello World!");

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

**52、List、Set、Map是否继承自Collection接口？**   
答：List、Set 是，Map 不是。Map是键值对映射容器，与List和Set有明显的区别，而Set存储的零散的元素且不允许有重复元素（数学中的集合也是如此），List是线性结构的容器，适用于按数值索引访问元素的情形。

**53、阐述ArrayList、Vector、LinkedList的存储性能和特性。**   
答：ArrayList 和Vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但性能上较ArrayList差，因此已经是Java中的遗留容器。LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。Vector属于遗留容器（Java早期的版本中提供的容器，除此之外，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已经不推荐使用，但是由于ArrayList和LinkedListed都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可以通过工具类Collections中的synchronizedList方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现）。

**补充：**遗留容器中的Properties类和Stack类在设计上有严重的问题，Properties是一个键和值都是字符串的特殊的键值对映射，在设计上应该是关联一个Hashtable并将其两个泛型参数设置为String类型，但是Java API中的Properties直接继承了Hashtable，这很明显是对继承的滥用。这里复用代码的方式应该是Has-A关系而不是Is-A关系，另一方面容器都属于工具类，继承工具类本身就是一个错误的做法，使用工具类最好的方式是Has-A关系（关联）或Use-A关系（依赖）。同理，Stack类继承Vector也是不正确的。Sun公司的工程师们也会犯这种低级错误，让人唏嘘不已。

**54、Collection和Collections的区别？**   
答：Collection是一个接口，它是Set、List等容器的父接口；Collections是个一个工具类，提供了一系列的静态方法来辅助容器操作，这些方法包括对容器的搜索、排序、线程安全化等等。

**55、List、Map、Set三个接口存取元素时，各有什么特点？**   
答：List以特定索引来存取元素，可以有重复元素。Set不能存放重复元素（用对象的equals()方法来区分元素是否重复）。Map保存键值对（key-value pair）映射，映射关系可以是一对一或多对一。Set和Map容器都有基于哈希存储和排序树的两种实现版本，基于哈希存储的版本理论存取时间复杂度为O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

**56、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？**   
答：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）。   
例子1：

public class Student implements Comparable<Student> {

private String name; // 姓名

private int age; // 年龄

public Student(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Student [name=" + name + ", age=" + age + "]";

}

@Override

public int compareTo(Student o) {

return this.age - o.age; // 比较年龄(年龄的升序)

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

import java.util.Set;

import java.util.TreeSet;

class Test01 {

public static void main(String[] args) {

Set<Student> set = new TreeSet<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)

set.add(new Student("Hao LUO", 33));

set.add(new Student("XJ WANG", 32));

set.add(new Student("Bruce LEE", 60));

set.add(new Student("Bob YANG", 22));

for(Student stu : set) {

System.out.println(stu);

}

// 输出结果:

// Student [name=Bob YANG, age=22]

// Student [name=XJ WANG, age=32]

// Student [name=Hao LUO, age=33]

// Student [name=Bruce LEE, age=60]

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

例子2：

public class Student {

private String name; // 姓名

private int age; // 年龄

public Student(String name, int age) {

this.name = name;

this.age = age;

}

/\*\*

\* 获取学生姓名

\*/

public String getName() {

return name;

}

/\*\*

\* 获取学生年龄

\*/

public int getAge() {

return age;

}

@Override

public String toString() {

return "Student [name=" + name + ", age=" + age + "]";

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class Test02 {

public static void main(String[] args) {

List<Student> list = new ArrayList<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)

list.add(new Student("Hao LUO", 33));

list.add(new Student("XJ WANG", 32));

list.add(new Student("Bruce LEE", 60));

list.add(new Student("Bob YANG", 22));

// 通过sort方法的第二个参数传入一个Comparator接口对象

// 相当于是传入一个比较对象大小的算法到sort方法中

// 由于Java中没有函数指针、仿函数、委托这样的概念

// 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调

Collections.sort(list, new Comparator<Student> () {

@Override

public int compare(Student o1, Student o2) {

return o1.getName().compareTo(o2.getName()); // 比较学生姓名

}

});

for(Student stu : list) {

System.out.println(stu);

}

// 输出结果:

// Student [name=Bob YANG, age=22]

// Student [name=Bruce LEE, age=60]

// Student [name=Hao LUO, age=33]

// Student [name=XJ WANG, age=32]

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36

**57、Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别?**   
答：sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

**补充：**可能不少人对什么是进程，什么是线程还比较模糊，对于为什么需要多线程编程也不是特别理解。简单的说：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，是操作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，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的CPU资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费CPU时间。时下很时髦的[Node.js](https://nodejs.org/)就采用了单线程异步I/O的工作模式。

**58、线程的sleep()方法和yield()方法有什么区别？**   
答：   
① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；   
② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；   
③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；   
④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

**59、当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？**   
答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（**注意不是等待池哦**）中等待对象的锁。

**60、请说出与线程同步以及线程调度相关的方法。**   
答：   
- wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；   
- sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；   
- notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；   
- notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

**提示：**关于Java多线程和并发编程的问题，建议大家看我的另一篇文章[《关于Java并发编程的总结和思考》](http://blog.csdn.net/jackfrued/article/details/44499227)。

补充：Java 5通过Lock接口提供了显式的锁机制（explicit lock），增强了灵活性以及对线程的协调。Lock接口中定义了加锁（lock()）和解锁（unlock()）的方法，同时还提供了newCondition()方法来产生用于线程之间通信的Condition对象；此外，Java 5还提供了信号量机制（semaphore），信号量可以用来限制对某个共享资源进行访问的线程的数量。在对资源进行访问之前，线程必须得到信号量的许可（调用Semaphore对象的acquire()方法）；在完成对资源的访问后，线程必须向信号量归还许可（调用Semaphore对象的release()方法）。

下面的例子演示了100个线程同时向一个银行账户中存入1元钱，在没有使用同步机制和使用同步机制情况下的执行情况。

* 银行账户类：

/\*\*

\* 银行账户

\* @author 骆昊

\*

\*/

public class Account {

private double balance; // 账户余额

/\*\*

\* 存款

\* @param money 存入金额

\*/

public void deposit(double money) {

double newBalance = balance + money;

try {

Thread.sleep(10); // 模拟此业务需要一段处理时间

}

catch(InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

/\*\*

\* 获得账户余额

\*/

public double getBalance() {

return balance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 存钱线程类：

/\*\*

\* 存钱线程

\* @author 骆昊

\*

\*/

public class AddMoneyThread implements Runnable {

private Account account; // 存入账户

private double money; // 存入金额

public AddMoneyThread(Account account, double money) {

this.account = account;

this.money = money;

}

@Override

public void run() {

account.deposit(money);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 测试类：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test01 {

public static void main(String[] args) {

Account account = new Account();

ExecutorService service = Executors.newFixedThreadPool(100);

for(int i = 1; i <= 100; i++) {

service.execute(new AddMoneyThread(account, 1));

}

service.shutdown();

while(!service.isTerminated()) {}

System.out.println("账户余额: " + account.getBalance());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

在没有同步的情况下，执行结果通常是显示账户余额在10元以下，出现这种状况的原因是，当一个线程A试图存入1元的时候，另外一个线程B也能够进入存款的方法中，线程B读取到的账户余额仍然是线程A存入1元钱之前的账户余额，因此也是在原来的余额0上面做了加1元的操作，同理线程C也会做类似的事情，所以最后100个线程执行结束时，本来期望账户余额为100元，但实际得到的通常在10元以下（很可能是1元哦）。解决这个问题的办法就是同步，当一个线程对银行账户存钱时，需要将此账户锁定，待其操作完成后才允许其他的线程进行操作，代码有如下几种调整方案：

* 在银行账户的存款（deposit）方法上同步（synchronized）关键字

/\*\*

\* 银行账户

\* @author 骆昊

\*

\*/

public class Account {

private double balance; // 账户余额

/\*\*

\* 存款

\* @param money 存入金额

\*/

public synchronized void deposit(double money) {

double newBalance = balance + money;

try {

Thread.sleep(10); // 模拟此业务需要一段处理时间

}

catch(InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

/\*\*

\* 获得账户余额

\*/

public double getBalance() {

return balance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 在线程调用存款方法时对银行账户进行同步

/\*\*

\* 存钱线程

\* @author 骆昊

\*

\*/

public class AddMoneyThread implements Runnable {

private Account account; // 存入账户

private double money; // 存入金额

public AddMoneyThread(Account account, double money) {

this.account = account;

this.money = money;

}

@Override

public void run() {

synchronized (account) {

account.deposit(money);

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 通过Java 5显示的锁机制，为每个银行账户创建一个锁对象，在存款操作进行加锁和解锁的操作

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

/\*\*

\* 银行账户

\*

\* @author 骆昊

\*

\*/

public class Account {

private Lock accountLock = new ReentrantLock();

private double balance; // 账户余额

/\*\*

\* 存款

\*

\* @param money

\* 存入金额

\*/

public void deposit(double money) {

accountLock.lock();

try {

double newBalance = balance + money;

try {

Thread.sleep(10); // 模拟此业务需要一段处理时间

}

catch (InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

finally {

accountLock.unlock();

}

}

/\*\*

\* 获得账户余额

\*/

public double getBalance() {

return balance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43

按照上述三种方式对代码进行修改后，重写执行测试代码Test01，将看到最终的账户余额为100元。当然也可以使用Semaphore或CountdownLatch来实现同步。

**61、编写多线程程序有几种实现方式？**   
答：Java 5以前实现多线程有两种实现方法：一种是继承Thread类；另一种是实现Runnable接口。两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

补充：Java 5以后创建线程还有第三种方式：实现Callable接口，该接口中的call方法可以在线程执行结束时产生一个返回值，代码如下所示：

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

class MyTask implements Callable<Integer> {

private int upperBounds;

public MyTask(int upperBounds) {

this.upperBounds = upperBounds;

}

@Override

public Integer call() throws Exception {

int sum = 0;

for(int i = 1; i <= upperBounds; i++) {

sum += i;

}

return sum;

}

}

class Test {

public static void main(String[] args) throws Exception {

List<Future<Integer>> list = new ArrayList<>();

ExecutorService service = Executors.newFixedThreadPool(10);

for(int i = 0; i < 10; i++) {

list.add(service.submit(new MyTask((int) (Math.random() \* 100))));

}

int sum = 0;

for(Future<Integer> future : list) {

// while(!future.isDone()) ;

sum += future.get();

}

System.out.println(sum);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44

**62、synchronized关键字的用法？**   
答：synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。在第60题的例子中已经展示了synchronized关键字的用法。

**63、举例说明同步和异步。**   
答：如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。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率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

**64、启动一个线程是调用run()还是start()方法？**   
答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

**65、什么是线程池（thread pool）？**   
答：在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。   
Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：   
- newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。   
- newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。   
- newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。   
- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。   
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

第60题的例子中演示了通过Executors工具类创建线程池并使用线程池执行线程的代码。如果希望在服务器上使用线程池，强烈建议使用newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

**66、线程的基本状态以及状态之间的关系？**   
答：   


**说明：**其中Running表示运行状态，Runnable表示就绪状态（万事俱备，只欠CPU），Blocked表示阻塞状态，阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

**67、简述synchronized 和java.util.concurrent.locks.Lock的异同？**   
答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

**68、Java中如何实现序列化，有什么意义？**   
答：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。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。   
要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第29题）。

**69、Java中有几种类型的流？**   
答：字节流和字符流。字节流继承于InputStream、OutputStream，字符流继承于Reader、Writer。在java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于Java的I/O需要注意的有两点：一是两种对称性（输入和输出的对称性，字节和字符的对称性）；二是两种设计模式（适配器模式和装潢模式）。另外Java中的流不同于C#的是它只有一个维度一个方向。

**面试题** - 编程实现文件拷贝。（这个题目在笔试的时候经常出现，下面的代码给出了两种实现方案）

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public final class MyUtil {

private MyUtil() {

throw new AssertionError();

}

public static void fileCopy(String source, String target) throws IOException {

try (InputStream in = new FileInputStream(source)) {

try (OutputStream out = new FileOutputStream(target)) {

byte[] buffer = new byte[4096];

int bytesToRead;

while((bytesToRead = in.read(buffer)) != -1) {

out.write(buffer, 0, bytesToRead);

}

}

}

}

public static void fileCopyNIO(String source, String target) throws IOException {

try (FileInputStream in = new FileInputStream(source)) {

try (FileOutputStream out = new FileOutputStream(target)) {

FileChannel inChannel = in.getChannel();

FileChannel outChannel = out.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(4096);

while(inChannel.read(buffer) != -1) {

buffer.flip();

outChannel.write(buffer);

buffer.clear();

}

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

**注意：**上面用到Java 7的TWR，使用TWR后可以不用在finally中释放外部资源 ，从而让代码更加优雅。

**70、写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。**   
答：代码如下：

import java.io.BufferedReader;

import java.io.FileReader;

public final class MyUtil {

// 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)

private MyUtil() {

throw new AssertionError();

}

/\*\*

\* 统计给定文件中给定字符串的出现次数

\*

\* @param filename 文件名

\* @param word 字符串

\* @return 字符串在文件中出现的次数

\*/

public static int countWordInFile(String filename, String word) {

int counter = 0;

try (FileReader fr = new FileReader(filename)) {

try (BufferedReader br = new BufferedReader(fr)) {

String line = null;

while ((line = br.readLine()) != null) {

int index = -1;

while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {

counter++;

line = line.substring(index + word.length());

}

}

}

} catch (Exception ex) {

ex.printStackTrace();

}

return counter;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37

**71、如何用Java代码列出一个目录下所有的文件？**   
答：   
如果只要求列出当前文件夹下的文件，代码如下所示：

import java.io.File;

class Test12 {

public static void main(String[] args) {

File f = new File("/Users/Hao/Downloads");

for(File temp : f.listFiles()) {

if(temp.isFile()) {

System.out.println(temp.getName());

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

如果需要对文件夹继续展开，代码如下所示：

import java.io.File;

class Test12 {

public static void main(String[] args) {

showDirectory(new File("/Users/Hao/Downloads"));

}

public static void showDirectory(File f) {

\_walkDirectory(f, 0);

}

private static void \_walkDirectory(File f, int level) {

if(f.isDirectory()) {

for(File temp : f.listFiles()) {

\_walkDirectory(temp, level + 1);

}

}

else {

for(int i = 0; i < level - 1; i++) {

System.out.print("\t");

}

System.out.println(f.getName());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

在Java 7中可以使用NIO.2的API来做同样的事情，代码如下所示：

class ShowFileTest {

public static void main(String[] args) throws IOException {

Path initPath = Paths.get("/Users/Hao/Downloads");

Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {

@Override

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)

throws IOException {

System.out.println(file.getFileName().toString());

return FileVisitResult.CONTINUE;

}

});

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

**72、用Java的套接字编程实现一个多线程的回显（echo）服务器。**   
答：

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

public class EchoServer {

private static final int ECHO\_SERVER\_PORT = 6789;

public static void main(String[] args) {

try(ServerSocket server = new ServerSocket(ECHO\_SERVER\_PORT)) {

System.out.println("服务器已经启动...");

while(true) {

Socket client = server.accept();

new Thread(new ClientHandler(client)).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class ClientHandler implements Runnable {

private Socket client;

public ClientHandler(Socket client) {

this.client = client;

}

@Override

public void run() {

try(BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));

PrintWriter pw = new PrintWriter(client.getOutputStream())) {

String msg = br.readLine();

System.out.println("收到" + client.getInetAddress() + "发送的: " + msg);

pw.println(msg);

pw.flush();

} catch(Exception ex) {

ex.printStackTrace();

} finally {

try {

client.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51

**注意：**上面的代码使用了Java 7的TWR语法，由于很多外部资源类都间接的实现了AutoCloseable接口（单方法回调接口），因此可以利用TWR语法在try结束的时候通过回调的方式自动调用外部资源类的close()方法，避免书写冗长的finally代码块。此外，上面的代码用一个静态内部类实现线程的功能，使用多线程可以避免一个用户I/O操作所产生的中断影响其他用户对服务器的访问，简单的说就是一个用户的输入操作不会造成其他用户的阻塞。当然，上面的代码使用线程池可以获得更好的性能，因为频繁的创建和销毁线程所造成的开销也是不可忽视的。

下面是一段回显客户端测试代码：

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.util.Scanner;

public class EchoClient {

public static void main(String[] args) throws Exception {

Socket client = new Socket("localhost", 6789);

Scanner sc = new Scanner(System.in);

System.out.print("请输入内容: ");

String msg = sc.nextLine();

sc.close();

PrintWriter pw = new PrintWriter(client.getOutputStream());

pw.println(msg);

pw.flush();

BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));

System.out.println(br.readLine());

client.close();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

如果希望用NIO的多路复用套接字实现服务器，代码如下所示。NIO的操作虽然带来了更好的性能，但是有些操作是比较底层的，对于初学者来说还是有些难于理解。

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.Iterator;

public class EchoServerNIO {

private static final int ECHO\_SERVER\_PORT = 6789;

private static final int ECHO\_SERVER\_TIMEOUT = 5000;

private static final int BUFFER\_SIZE = 1024;

private static ServerSocketChannel serverChannel = null;

private static Selector selector = null; // 多路复用选择器

private static ByteBuffer buffer = null; // 缓冲区

public static void main(String[] args) {

init();

listen();

}

private static void init() {

try {

serverChannel = ServerSocketChannel.open();

buffer = ByteBuffer.allocate(BUFFER\_SIZE);

serverChannel.socket().bind(new InetSocketAddress(ECHO\_SERVER\_PORT));

serverChannel.configureBlocking(false);

selector = Selector.open();

serverChannel.register(selector, SelectionKey.OP\_ACCEPT);

} catch (Exception e) {

throw new RuntimeException(e);

}

}

private static void listen() {

while (true) {

try {

if (selector.select(ECHO\_SERVER\_TIMEOUT) != 0) {

Iterator<SelectionKey> it = selector.selectedKeys().iterator();

while (it.hasNext()) {

SelectionKey key = it.next();

it.remove();

handleKey(key);

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

private static void handleKey(SelectionKey key) throws IOException {

SocketChannel channel = null;

try {

if (key.isAcceptable()) {

ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();

channel = serverChannel.accept();

channel.configureBlocking(false);

channel.register(selector, SelectionKey.OP\_READ);

} else if (key.isReadable()) {

channel = (SocketChannel) key.channel();

buffer.clear();

if (channel.read(buffer) > 0) {

buffer.flip();

CharBuffer charBuffer = CharsetHelper.decode(buffer);

String msg = charBuffer.toString();

System.out.println("收到" + channel.getRemoteAddress() + "的消息：" + msg);

channel.write(CharsetHelper.encode(CharBuffer.wrap(msg)));

} else {

channel.close();

}

}

} catch (Exception e) {

e.printStackTrace();

if (channel != null) {

channel.close();

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.charset.CharacterCodingException;

import java.nio.charset.Charset;

import java.nio.charset.CharsetDecoder;

import java.nio.charset.CharsetEncoder;

public final class CharsetHelper {

private static final String UTF\_8 = "UTF-8";

private static CharsetEncoder encoder = Charset.forName(UTF\_8).newEncoder();

private static CharsetDecoder decoder = Charset.forName(UTF\_8).newDecoder();

private CharsetHelper() {

}

public static ByteBuffer encode(CharBuffer in) throws CharacterCodingException{

return encoder.encode(in);

}

public static CharBuffer decode(ByteBuffer in) throws CharacterCodingException{

return decoder.decode(in);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

**73、XML文档定义有几种形式？它们之间有何本质区别？解析XML文档有哪几种方式？**   
答：XML文档定义分为DTD和Schema两种形式，二者都是对XML语法的约束，其本质区别在于Schema本身也是一个XML文件，可以被XML解析器解析，而且可以为XML承载的数据定义类型，约束能力较之DTD更强大。对XML的解析主要有DOM（文档对象模型，Document Object Model）、SAX（Simple API for XML）和StAX（Java 6中引入的新的解析XML的方式，Streaming API for XML），其中DOM处理大型文件时其性能下降的非常厉害，这个问题是由DOM树结构占用的内存较多造成的，而且DOM解析方式必须在解析文件之前把整个文档装入内存，适合对XML的随机访问（典型的用空间换取时间的策略）；SAX是事件驱动型的XML解析方式，它顺序读取XML文件，不需要一次全部装载整个文件。当遇到像文件开头，文档结束，或者标签开头与标签结束时，它会触发一个事件，用户通过事件回调代码来处理XML文件，适合对XML的顺序访问；顾名思义，StAX把重点放在流上，实际上StAX与其他解析方式的本质区别就在于应用程序能够把XML作为一个事件流来处理。将XML作为一组事件来处理的想法并不新颖（SAX就是这样做的），但不同之处在于StAX允许应用程序代码把这些事件逐个拉出来，而不用提供在解析器方便时从解析器中接收事件的处理程序。

**74、你在项目中哪些地方用到了XML？**   
答：XML的主要作用有两个方面：数据交换和信息配置。在做数据交换时，XML将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从XML文件中还原相关信息进行处理，XML曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被JSON（JavaScript Object Notation）取而代之。当然，目前很多软件仍然使用XML来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在XML文件中，Java的很多框架也是这么做的，而且这些框架都选择了[dom4j](http://www.dom4j.org/)作为处理XML的工具，因为Sun公司的官方API实在不怎么好用。

**补充：**现在有很多时髦的软件（如Sublime）已经开始将配置文件书写成JSON格式，我们已经强烈的感受到XML的另一项功能也将逐渐被业界抛弃。

**75、阐述JDBC操作数据库的步骤。**   
答：下面的代码以连接本机的Oracle数据库为例，演示JDBC操作数据库的步骤。

* 加载驱动。

Class.forName("oracle.jdbc.driver.OracleDriver");

* 1
* 创建连接。

Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");

* 1
* 创建语句。

PreparedStatement ps = con.prepareStatement("select \* from emp where sal between ? and ?");

ps.setInt(1, 1000);

ps.setInt(2, 3000);

* 1
* 2
* 3
* 执行语句。

ResultSet rs = ps.executeQuery();

* 1
* 处理结果。

while(rs.next()) {

System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));

}

* 1
* 2
* 3
* 关闭资源。

finally {

if(con != null) {

try {

con.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

**提示：**关闭外部资源的顺序应该和打开的顺序相反，也就是说先关闭ResultSet、再关闭Statement、在关闭Connection。上面的代码只关闭了Connection（连接），虽然通常情况下在关闭连接时，连接上创建的语句和打开的游标也会关闭，但不能保证总是如此，因此应该按照刚才说的顺序分别关闭。此外，第一步加载驱动在JDBC 4.0中是可以省略的（自动从类路径中加载驱动），但是我们建议保留。

**76、Statement和PreparedStatement有什么区别？哪个性能更好？**   
答：与Statement相比，①PreparedStatement接口代表预编译的语句，它主要的优势在于可以减少SQL的编译错误并增加SQL的安全性（减少SQL注射攻击的可能性）；②PreparedStatement中的SQL语句是可以带参数的，避免了用字符串连接拼接SQL语句的麻烦和不安全；③当批量处理SQL或频繁执行相同的查询时，PreparedStatement有明显的性能上的优势，由于数据库可以将编译优化后的SQL语句缓存起来，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

**补充：**为了提供对存储过程的调用，JDBC API中还提供了CallableStatement接口。存储过程（Stored Procedure）是数据库中一组为了完成特定功能的SQL语句的集合，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。虽然调用存储过程会在网络开销、安全性、性能上获得很多好处，但是存在如果底层数据库发生迁移时就会有很多麻烦，因为每种数据库的存储过程在书写上存在不少的差别。

**77、使用JDBC操作数据库时，如何提升读取数据的性能？如何提升更新数据的性能？**   
答：要提升读取数据的性能，可以指定通过结果集（ResultSet）对象的setFetchSize()方法指定每次抓取的记录数（典型的空间换时间策略）；要提升更新数据的性能可以使用PreparedStatement语句构建批处理，将若干SQL语句置于一个批处理中执行。

**78、在进行数据库编程时，连接池有什么作用？**   
答：由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，释放连接需要进行TCP四次握手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。池化技术在Java开发中是很常见的，在使用线程时创建线程池的道理与此相同。基于Java的开源数据库连接池主要有：[C3P0](http://sourceforge.net/projects/c3p0/)、[Proxool](http://proxool.sourceforge.net/)、[DBCP](http://commons.apache.org/proper/commons-dbcp/)、[BoneCP](https://github.com/wwadge/bonecp)、[Druid](https://github.com/alibaba/druid)等。

**补充：**在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

**79、什么是DAO模式？**   
答：DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。

**80、事务的ACID是指什么？**   
答：   
- 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；   
- 一致性(Consistent)：事务结束后系统状态是一致的；   
- 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态；   
- 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

**补充：**关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在5类问题，包括3类数据读取问题（脏读、不可重复读和幻读）和2类数据更新问题（第1类丢失更新和第2类丢失更新）。

脏读（Dirty Read）：A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 |  | 取出500元余额修改为500元 |
| T5 | 查询账户余额为500元（脏读） |  |
| T6 |  | 撤销事务余额恢复为1000元 |
| T7 | 汇入100元把余额修改为600元 |  |
| T8 | 提交事务 |  |

不可重复读（Unrepeatable Read）：事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务B修改过了。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元修改余额为900元 |
| T6 |  | 提交事务 |
| T7 | 查询账户余额为900元（不可重复读） |  |

幻读（Phantom Read）：事务A重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务B提交的行。

| **时间** | **统计金额事务A** | **转账事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 | 统计总存款为10000元 |  |
| T4 |  | 新增一个存款账户存入100元 |
| T5 |  | 提交事务 |
| T6 | 再次统计总存款为10100元（幻读） |  |

第1类丢失更新：事务A撤销时，把已经提交的事务B的更新数据覆盖了。

| **时间** | **取款事务A** | **转账事务B** |
| --- | --- | --- |
| T1 | 开始事务 |  |
| T2 |  | 开始事务 |
| T3 | 查询账户余额为1000元 |  |
| T4 |  | 查询账户余额为1000元 |
| T5 |  | 汇入100元修改余额为1100元 |
| T6 |  | 提交事务 |
| T7 | 取出100元将余额修改为900元 |  |
| T8 | 撤销事务 |  |
| T9 | 余额恢复为1000元（丢失更新） |  |

第2类丢失更新：事务A覆盖事务B已经提交的数据，造成事务B所做的操作丢失。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元将余额修改为900元 |
| T6 |  | 提交事务 |
| T7 | 汇入100元将余额修改为1100元 |  |
| T8 | 提交事务 |  |
| T9 | 查询账户余额为1100元（丢失更新） |  |

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。   
直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析SQL语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，如下表所示：

| **隔离级别** | **脏读** | **不可重复读** | **幻读** | **第一类丢失更新** | **第二类丢失更新** |
| --- | --- | --- | --- | --- | --- |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITTED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

需要说明的是，事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

**81、JDBC中如何进行事务处理？**   
答：Connection提供了事务处理的方法，通过调用setAutoCommit(false)可以设置手动提交事务；当事务完成后用commit()显式提交事务；如果在事务处理过程中发生异常则通过rollback()进行事务回滚。除此之外，从JDBC 3.0中还引入了Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。   


**82、JDBC能否处理Blob和Clob？**   
答： Blob是指二进制大对象（Binary Large Object），而Clob是指大字符对象（Character Large Objec），因此其中Blob是为存储大的二进制数据而设计的，而Clob是为存储大的文本数据而设计的。JDBC的PreparedStatement和ResultSet都提供了相应的方法来支持Blob和Clob操作。下面的代码展示了如何使用JDBC操作LOB：   
下面以MySQL数据库为例，创建一个张有三个字段的用户表，包括编号（id）、姓名（name）和照片（photo），建表语句如下：

create table tb\_user

(

id int primary key auto\_increment,

name varchar(20) unique not null,

photo longblob

);

* 1
* 2
* 3
* 4
* 5
* 6

下面的Java代码向数据库中插入一条记录：

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

class JdbcLobTest {

public static void main(String[] args) {

Connection con = null;

try {

// 1. 加载驱动（Java6以上版本可以省略）

Class.forName("com.mysql.jdbc.Driver");

// 2. 建立连接

con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123456");

// 3. 创建语句对象

PreparedStatement ps = con.prepareStatement("insert into tb\_user values (default, ?, ?)");

ps.setString(1, "骆昊"); // 将SQL语句中第一个占位符换成字符串

try (InputStream in = new FileInputStream("test.jpg")) { // Java 7的TWR

ps.setBinaryStream(2, in); // 将SQL语句中第二个占位符换成二进制流

// 4. 发出SQL语句获得受影响行数

System.out.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");

} catch(IOException e) {

System.out.println("读取照片失败!");

}

} catch (ClassNotFoundException | SQLException e) { // Java 7的多异常捕获

e.printStackTrace();

} finally { // 释放外部资源的代码都应当放在finally中保证其能够得到执行

try {

if(con != null && !con.isClosed()) {

con.close(); // 5. 释放数据库连接

con = null; // 指示垃圾回收器可以回收该对象

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

**83、简述正则表达式及其用途。**   
答：在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

**说明：**计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，正则表达式就是在进行字符串匹配和处理的时候最为强大的工具，绝大多数语言都提供了对正则表达式的支持。

**84、Java中是如何支持正则表达式操作的？**   
答：Java中的String类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，Java中可以用Pattern类表示正则表达式对象，它提供了丰富的API进行各种正则表达式操作，请参考下面面试题的代码。

**面试题：** - 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

import java.util.regex.Matcher;

import java.util.regex.Pattern;

class RegExpTest {

public static void main(String[] args) {

String str = "北京市(朝阳区)(西城区)(海淀区)";

Pattern p = Pattern.compile(".\*?(?=\\()");

Matcher m = p.matcher(str);

if(m.find()) {

System.out.println(m.group());

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

**说明：**上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的[《正则表达式30分钟入门教程》](http://www.jb51.net/tools/zhengze.html)。

**85、获得一个类的类对象有哪些方式？**   
答：   
- 方法1：类型.class，例如：String.class   
- 方法2：对象.getClass()，例如："hello".getClass()   
- 方法3：Class.forName()，例如：Class.forName("java.lang.String")

**86、如何通过反射创建对象？**   
答：   
- 方法1：通过类对象调用newInstance()方法，例如：String.class.newInstance()   
- 方法2：通过类对象的getConstructor()或getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance("Hello");

**87、如何通过反射获取和设置对象私有字段的值？**   
答：可以通过类对象的getDeclaredField()方法字段（Field）对象，然后再通过字段对象的setAccessible(true)将其设置为可以访问，接下来就可以通过get/set方法来获取/设置字段的值了。下面的代码实现了一个反射的工具类，其中的两个静态方法分别用于获取和设置私有字段的值，字段可以是基本类型也可以是对象类型且支持多级对象操作，例如ReflectionUtil.get(dog, "owner.car.engine.id");可以获得dog对象的主人的汽车的引擎的ID号。

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Modifier;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 反射工具类

\* @author 骆昊

\*

\*/

public class ReflectionUtil {

private ReflectionUtil() {

throw new AssertionError();

}

/\*\*

\* 通过反射取对象指定字段(属性)的值

\* @param target 目标对象

\* @param fieldName 字段的名字

\* @throws 如果取不到对象指定字段的值则抛出异常

\* @return 字段的值

\*/

public static Object getValue(Object target, String fieldName) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

try {

for(int i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(true);

target = f.get(target);

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(true);

return f.get(target);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

/\*\*

\* 通过反射给对象的指定字段赋值

\* @param target 目标对象

\* @param fieldName 字段的名称

\* @param value 值

\*/

public static void setValue(Object target, String fieldName, Object value) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

try {

for(int i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(true);

Object val = f.get(target);

if(val == null) {

Constructor<?> c = f.getType().getDeclaredConstructor();

c.setAccessible(true);

val = c.newInstance();

f.set(target, val);

}

target = val;

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(true);

f.set(target, value);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

}31

**88、如何通过反射调用对象的方法？**   
答：请看下面的代码：

import java.lang.reflect.Method;

class MethodInvokeTest {

public static void main(String[] args) throws Exception {

String str = "hello";

Method m = str.getClass().getMethod("toUpperCase");

System.out.println(m.invoke(str)); // HELLO

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

**89、简述一下面向对象的"六原则一法则"。**   
答：   
- 单一职责原则：一个类只做它该做的事情。（单一职责原则想表达的就是"高内聚"，写代码最终极的原则只有六个字"高内聚、低耦合"，就如同葵花宝典或辟邪剑谱的中心思想就八个字"欲练此功必先自宫"，所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫"因为专注，所以专业"，一个对象如果承担太多的职责，那么注定它什么都做不好。这个世界上任何好的东西都有两个特征，一个是功能单一，好的相机绝对不是电视购物里面卖的那种一个机器有一百多种功能的，它基本上只能照相；另一个是模块化，好的自行车是组装车，从减震叉、刹车到变速器，所有的部件都是可以拆卸和重新组装的，好的乒乓球拍也不是成品拍，一定是底板和胶皮可以拆分和自行组装的，一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这样才能实现软件复用的目标。）   
- 开闭原则：软件实体应当对扩展开放，对修改关闭。（在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。）   
- 依赖倒转原则：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。）   
里氏替换原则：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）   
- 接口隔离原则：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。）   
- 合成聚合复用原则：优先使用聚合或合成关系复用代码。（通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，类与类之间简单的说有三种关系，Is-A关系、Has-A关系、Use-A关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是Has-A关系，合成聚合复用原则想表达的是优先考虑Has-A关系而不是Is-A关系复用代码，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在Java的API中也有不少滥用继承的例子，例如Properties类继承了Hashtable类，Stack类继承了Vector类，这些继承明显就是错误的，更好的做法是在Properties类中放置一个Hashtable类型的成员并且将其键和值都设置为字符串来存储数据，而Stack类的设计也应该是在Stack类中放一个Vector对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。）   
- 迪米特法则：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到"低耦合"，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web开发中作为前端控制器的Servlet或Filter不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）


**90、简述一下你了解的设计模式。**   
答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。   
在GoF的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共23种设计模式，包括：Abstract Factory（抽象工厂模式），Builder（建造者模式），Factory Method（工厂方法模式），Prototype（原始模型模式），Singleton（单例模式）；Facade（门面模式），Adapter（适配器模式），Bridge（桥梁模式），Composite（合成模式），Decorator（装饰模式），Flyweight（享元模式），Proxy（代理模式）；Command（命令模式），Interpreter（解释器模式），Visitor（访问者模式），Iterator（迭代子模式），Mediator（调停者模式），Memento（备忘录模式），Observer（观察者模式），State（状态模式），Strategy（策略模式），Template Method（模板方法模式）， Chain Of Responsibility（责任链模式）。   
面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：   
- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。   
- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。   
- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。   
- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。   
除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

**91、用Java写一个单例类。**   
答：   
- 饿汉式单例

public class Singleton {

private Singleton(){}

private static Singleton instance = new Singleton();

public static Singleton getInstance(){

return instance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 懒汉式单例

public class Singleton {

private static Singleton instance = null;

private Singleton() {}

public static synchronized Singleton getInstance(){

if (instance == null) instance ＝ new Singleton();

return instance;

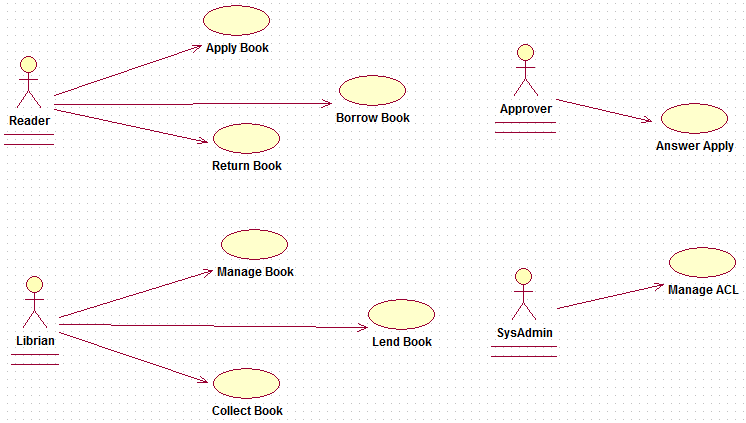
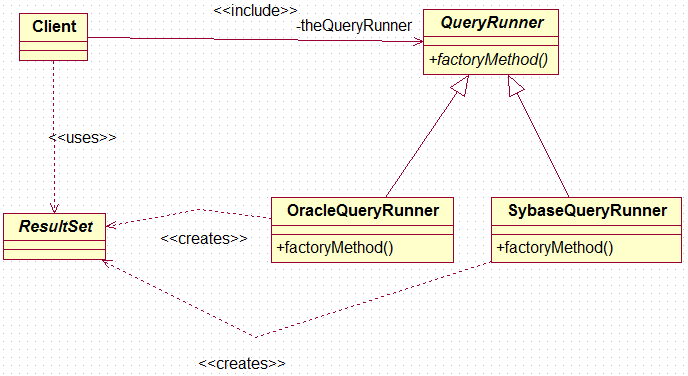
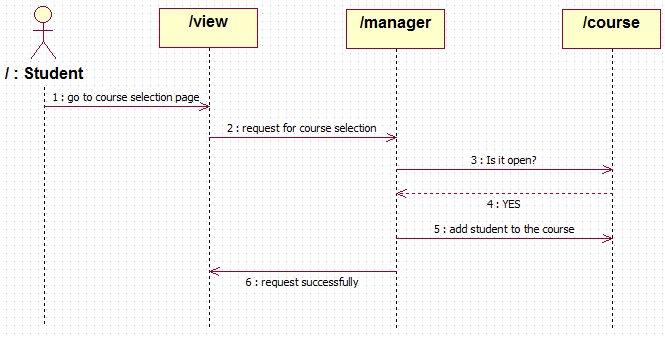
}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

**注意：**实现一个单例有两点注意事项，①将构造器私有，不允许外界通过构造器创建对象；②通过公开的静态方法向外界返回类的唯一实例。这里有一个问题可以思考：Spring的IoC容器可以为普通的类创建单例，它是怎么做到的呢？

**92、什么是UML？**   
答：UML是统一建模语言（Unified Modeling Language）的缩写，它发表于1997年，综合了当时已经存在的面向对象的建模语言、方法和过程，是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用UML可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。

**93、UML中有哪些常用的图？**   
答：UML定义了多种图形化的符号来描述软件系统部分或全部的静态结构和动态结构，包括：用例图（use case diagram）、类图（class diagram）、时序图（sequence diagram）、协作图（collaboration diagram）、状态图（statechart diagram）、活动图（activity diagram）、构件图（component diagram）、部署图（deployment diagram）等。在这些图形化符号中，有三种图最为重要，分别是：用例图（用来捕获需求，描述系统的功能，通过该图可以迅速的了解系统的功能模块及其关系）、类图（描述类以及类与类之间的关系，通过该图可以快速了解系统）、时序图（描述执行特定任务时对象之间的交互关系以及执行顺序，通过该图可以了解对象能接收的消息也就是说对象能够向外界提供的服务）。   
用例图：   
  
类图：   
   
时序图：   


**94、用Java写一个冒泡排序。**   
答：冒泡排序几乎是个程序员都写得出来，但是面试的时候如何写一个逼格高的冒泡排序却不是每个人都能做到，下面提供一个参考代码：

import java.util.Comparator;

/\*\*

\* 排序器接口(策略模式: 将算法封装到具有共同接口的独立的类中使得它们可以相互替换)

\* @author骆昊

\*

\*/

public interface Sorter {

/\*\*

\* 排序

\* @param list 待排序的数组

\*/

public <T extends Comparable<T>> void sort(T[] list);

/\*\*

\* 排序

\* @param list 待排序的数组

\* @param comp 比较两个对象的比较器

\*/

public <T> void sort(T[] list, Comparator<T> comp);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

import java.util.Comparator;

/\*\*

\* 冒泡排序

\*

\* @author骆昊

\*

\*/

public class BubbleSorter implements Sorter {

@Override

public <T extends Comparable<T>> void sort(T[] list) {

boolean swapped = true;

for (int i = 1, len = list.length; i < len && swapped; ++i) {

swapped = false;

for (int j = 0; j < len - i; ++j) {

if (list[j].compareTo(list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

}

}

@Override

public <T> void sort(T[] list, Comparator<T> comp) {

boolean swapped = true;

for (int i = 1, len = list.length; i < len && swapped; ++i) {

swapped = false;

for (int j = 0; j < len - i; ++j) {

if (comp.compare(list[j], list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42

**95、用Java写一个折半查找。**   
答：折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是O(logN)。

import java.util.Comparator;

public class MyUtil {

public static <T extends Comparable<T>> int binarySearch(T[] x, T key) {

return binarySearch(x, 0, x.length- 1, key);

}

// 使用循环实现的二分查找

public static <T> int binarySearch(T[] x, T key, Comparator<T> comp) {

int low = 0;

int high = x.length - 1;

while (low <= high) {

int mid = (low + high) >>> 1;

int cmp = comp.compare(x[mid], key);

if (cmp < 0) {

low= mid + 1;

}

else if (cmp > 0) {

high= mid - 1;

}

else {

return mid;

}

}

return -1;

}

// 使用递归实现的二分查找

private static<T extends Comparable<T>> int binarySearch(T[] x, int low, int high, T key) {

if(low <= high) {

int mid = low + ((high -low) >> 1);

if(key.compareTo(x[mid])== 0) {

return mid;

}

else if(key.compareTo(x[mid])< 0) {

return binarySearch(x,low, mid - 1, key);

}

else {

return binarySearch(x,mid + 1, high, key);

}

}

return -1;

}

}

**说明：**上面的代码中给出了折半查找的两个版本，一个用递归实现，一个用循环实现。需要注意的是计算中间位置时不应该使用(high+ low) / 2的方式，因为加法运算可能导致整数越界，这里应该使用以下三种方式之一：low + (high - low) / 2或low + (high – low) >> 1或(low + high) >>> 1（>>>是逻辑右移，是不带符号位的右移）