链滴

Java 基础知识点

作者：Gao-Eason

原文链接：https://ld246.com/article/1649668122432

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20210509.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">

<h2 id="基本概念">基本概念</h2>
<h3 id="什么是Java">什么是 Java</h3>
**Java 是一门面向对象编程语言，不仅吸收了 C++ 语言的各种优点，还摒弃了 C++ 里难以理
的多继承、指针等概念，因此 Java 语言具有功能强大和简单易用两个特征。Java 语言作为静态面向
象编程语言的代表，极好地实现了面向对象理论，允许程序员以优雅的思维方式进行复杂的编程 **</
>
<h3 id="Java-SE--Java-EE--Java-ME">Java SE 、Java EE 、Java ME</h3>

Java SE（J2SE，Java 2 Platform Standard Edition，标准版）**
**Java SE 以前称为 J2SE。它允许开发和部署在桌面、服务器、嵌入式环境和实时环境中
用的 Java 应用程序。Java SE 包含了支持 Java Web 服务开发的类，并为 Java EE 和 Java ME 提供
础。
Java EE（J2EE，Java 2 Platform Enterprise Edition，企业版）**
**Java EE 以前称为 J2EE。企业版本帮助开发和部署可移植、健壮、可伸缩且安全的服务
端 Java 应用程序。Java EE 是在 Java SE 的基础上构建的，它提供 Web 服务、组件模型、管理和通
API，可以用来实现企业级的面向服务体系结构（service-oriented architecture，SOA）和 Web2.0
应用程序。2018 年 2 月，Eclipse 宣布正式将 JavaEE 更名为 JakartaEE
Java ME（J2ME，Java 2 Platform Micro Edition，微型版）**
**Java ME 以前称为 J2ME。Java ME 为在移动设备和嵌入式设备（比如手机、PDA、电
机顶盒和打印机）上运行的应用程序提供一个健壮且灵活的环境。Java ME 包括灵活的用户界面、健
的安全模型、许多内置的网络协议以及对可以动态下载的连网和离线应用程序的丰富支持。基于 Java
ME 规范的应用程序只需编写一次，就可以用于许多设备，而且可以利用每个设备的本机功能。</str
ng>

<h3 id="Java的特点有哪些">Java 的特点有哪些</h3>
Java 语言是一种分布式的面向对象语言，具有面向对象、平台无关性、简单性、解释
行、多线程、安全性等很多特点，下面针对这些特点进行逐一介绍。
1. 面向对象
Java 是一种面向对象的语言，它对对象中的类、对象、继承、封装、多态、接口、包
均有很好的支持。为了简单起见，Java 只支持类之间的单继承，但是可以使用接口来实现多继承。使
Java 语言开发程序，需要采用面向对象的思想设计程序和编写代码。
2. 平台无关性
平台无关性的具体表现在于，Java 是“一次编写，到处运行（Write Once，Run any
Where）”的语言，因此采用 Java 语言编写的程序具有很好的可移植性，而保证这一点的正是 Java
的虚拟机机制。在引入虚拟机之后，Java 语言在不同的平台上运行不需要重新编译。
Java 语言使用 Java 虚拟机机制屏蔽了具体平台的相关信息，使得 Java 语言编译的程
只需生成虚拟机上的目标代码，就可以在多种平台上不加修改地运行。
3. 简单性
Java 语言的语法与 C 语言和 C++ 语言很相近，使得很多程序员学起来很容易。对 Jav
来说，它舍弃了很多 C++ 中难以理解的特性，如操作符的重载和多继承等，而且 Java 语言不使用
针，加入了垃圾回收机制，解决了程序员需要管理内存的问题，使编程变得更加简单。</
>
4. 解释执行
Java 程序在 Java 平台运行时会被编译成字节码文件，然后可以在有 Java 环境的操作
统上运行。在运行文件时，Java 的解释器对这些字节码进行解释执行，执行过程中需要加入的类在连
阶段被载入到运行环境中。
5. 多线程
Java 语言是多线程的，这也是 Java 语言的一大特性，它必须由 Thread 类和它的子
来创建。Java 支持多个线程同时执行，并提供多线程之间的同步机制。任何一个线程都有自己的 run()
方法，要执行的方法就写在 run() 方法体内。
6. 分布式

Java 语言支持 Internet 应用的开发，在 Java 的基本应用编程接口中就有一个网络应编程接口，它提供了网络应用编程的类库，包括 URL、URLConnection、Socket 等。Java 的 RIM 制也是开发分布式应用的重要手段。
7. 健壮性
Java 的强类型机制、异常处理、垃圾回收机制等都是 Java 健壮性的重要保证。对指的丢弃是 Java 的一大进步。另外，Java 的异常机制也是健壮性的一大体现。
8. 高性能
Java 的高性能主要是相对其他高级脚本语言来说的，随着 JIT（Just in Time）的发展 Java 的运行速度也越来越高。
9. 安全性
Java 通常被用在网络环境中，为此，Java 提供了一个安全机制以防止恶意代码的攻击除了 Java 语言具有许多的安全特性以外，Java 还对通过网络下载的类增加一个安全防范机制，分配同的名字空间以防替代本地的同名类，并包含安全管理机制。
Java 语言的众多特性使其在众多的编程语言中占有较大的市场份额，Java 语言对对象支持和强大的 API 使得编程工作变得更加容易和快捷，大大降低了程序的开发成本。Java 的“一次写，到处执行”正是它吸引众多商家和编程人员的一大优势。
<h3 id="JDK和JRE和JVM的区别">JDK 和 JRE 和 JVM 的区别</h3>
1. JDK
JDK（Java SE Development Kit），Java 标准的开发，提供了编译、运行 Java 程序所需要的各种工具和资源，包括了 Java 编译器、Java 运行时环境、以及常用的 Java 类库等。
2. JRE
JRE（Java Runtime Environment），Java 运行时境，用于解释执行 Java 的字节码文件**。普通用户只需要安装 JRE 来运行 Java 程序即可，而作为一名程序员必须安装 JDK，来编译、调试程序。**
3. JVM
JVM（Java Virtual Mechinal），Java 虚拟机，是 JRE 的一部分。它是整个 Java 实现跨平台的核心，负责解释执行字节文件，是可运行 Java 字节码文件的虚拟计算机。所有平台上的 JVM 向编译器提供相同的接口，而编器只需要面向虚拟机，生成虚拟机能识别的代码，然后由虚拟机来解释执行。
当使用 Java 编译器编译 Java 程序时，生成的是与平台无关的字节码，这些字节码只向 JVM。也就是说JVM 是运行 Java 字节码的虚拟机。
不同平台的 JVM 是不同的，但是他们都提供了相同的接口。JVM 是 Java 程序跨平台关键部分，只要为不同平台实现了相同的虚拟机，编译后的 Java 字节码就可以在该平台上运行。
为什么要采用字节码：
<blockquote>
在 Java 中，JVM 可以理解的代码就叫做 <code>字节码</code>（即 Java 源代码经过虚拟机编译器编译后扩展名为 <code>.class</code> 文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不操作系统的计算机上运行。
</blockquote>
什么是跨平台：
<blockquote>
所谓跨平台性，是指 java 语言编写的程序，一次编译后，可以在多个系统平台上运行
实现原理：Java 程序是通过 java 虚拟机在系统平台上运行的，只要该系统可以安装应的 java 虚拟机，该系统就可以运行 java 程序。
</blockquote>
Java 程序从源代码到运行需要三步：

com/file/2022/04/solo-fetchupload-6237621879298842152-4884662a.png?imageView2/2/interlace/1/format/webp">
4. 总结

JDK 用于开发，JRE 用于运行 java 程序； 如果只是运行 Java 程序，可以只安装 JRE 无序安装 JDK。
JDk 包含 JRE，JDK 和 JRE 中都包含 JVM。
JVM 是 Java 编程语言的核心并且具有平台独立性。

<h3 id="什么是跨平台性-原理是什么">什么是跨平台性？原理是什么</h3>

所谓跨平台性，是指 java 语言编写的程序，一次编译后，可以在多个系统平台上运行
实现原理：Java 程序是通过 java 虚拟机在系统平台上运行的，只要该系统可以安装应的 java 虚拟机，该系统就可以运行 java 程序。

<h3 id="Oracle-JDK-和-OpenJDK-的对比">Oracle JDK 和 OpenJDK 的对比</h3>

Oracle JDK 版本将每三年发布一次，而 OpenJDK 版本每三个月发布一次；
OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是 OpenJDK 的一个实，并不是完全开源的；
Oracle JDK 比 OpenJDK 更稳定。OpenJDK 和 Oracle JDK 的代码几乎相同，但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择 Oracle JDK 因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；
在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；
Oracle JDK 不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版获得支持来获取最新版本；
Oracle JDK 根据二进制代码许可协议获得许可，而 OpenJDK 根据 GPL v2 许可获得可。

<h3 id="Java和C--的区别">Java 和 C++ 的区别</h3>
我知道很多人没学过 C++，但是面试官就是没事喜欢拿咱们 Java 和 C++ 比呀！没法！！！就算没学过 C++，也要记下来！

都是面向对象的语言，都支持封装、继承和多态
Java 不提供指针来直接访问内存，程序内存更加安全
Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口以多继承。
Java 有自动内存管理机制，不需要程序员手动释放无用内存

<h3 id="什么Java注释">什么 Java 注释</h3>
定义**：用于解释说明程序的文字**
分类

单行注释**
**** 格式： // 注释文字**
多行注释**
**** 格式： /* 注释文字 /*
文档注释**
**** 格式：/** 注释文字 /*

作用

在程序中，尤其是复杂的程序中，适当地加入注释可以增加程序的可读性，有利于程的修改、调试和交流。注释的内容在程序编译的时候会被忽视，不会产生目标代码，注释的部分不会程序的执行结果产生任何影响。

注意事项：** 多行和文档注释都不能嵌套使用。**</>
<h3 id="Java程序是如何执行的">Java 程序是如何执行的</h3>
我们日常的工作中都使用开发工具（IntelliJ IDEA 或 Eclipse 等）可以很方便的调试序，或者是通过打包工具把项目打包成 jar 包或者 war 包，放入 Tomcat 等 Web 容器中就可以正常行了，但你有没有想过 Java 程序内部是如何执行的？其实不论是在开发工具中运行还是在 Tomcat 运行，Java 程序的执行流程基本都是相同的，它的执行流程如下：

先把 Java 代码编译成字节码，也就是把 .java 类型的文件编译成 .class 类型的文件。个过程的大致执行流程：Java 源代码 -> 词法分析器 -> 语法分析器 -> 语义分析器 -> 符码生成器 -> 最终生成字节码，其中任何一个节点执行失败就会造成编译失败；
把 class 文件放置到 Java 虚拟机，这个虚拟机通常指的是 Oracle 官方自带的 Hotspot JVM；
Java 虚拟机使用类加载器（Class Loader）装载 class 文件；
类加载完成之后，会进行字节码效验，字节码效验通过之后 JVM 解释器会把字节码翻成机器码交由操作系统执行。但不是所有代码都是解释执行的，JVM 对此做了优化，比如，以 Hotspt 虚拟机来说，它本身提供了 JIT（Just In Time）也就是我们通常所说的动态编译器，它能够在运行将热点代码编译为机器码，这个时候字节码就变成了编译执行。Java 程序执行流程图如下：

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile com/file/2022/04/solo-fetchupload-1619352028290462726-4620e22d.png?imageView2/2/int rlace/1/format/webp">
<h2 id="基础部分">基础部分</h2>
<h3 id="instanceof-关键字的作用">instanceof 关键字的作用</h3>
instanceof 严格来说是 Java 中的一个双目运算符，用来测试一个对象是否为一个类实例，用法为：
<pre><code class="highlight-chroma"> boolean result = obj instanceof Class
</code></pre>
其中 obj 为一个对象，Class 表示一个类或者一个接口，当 obj 为 Class 的对象，或是其直接或间接子类，或者是其接口的实现类，结果 result 都返回 true，否则返回 false。

注意：编译器会检查 obj 是否能转换成右边的 class 类型，如果不能转换则直接报错如果不能确定类型，则通过编译，具体看运行时定。
<pre><code class="highlight-chroma"> int i = 0;
 System.out println(i instanceof Integer);//编译不通过 i必须是引用类型，不能是基本类型
 System.out println(i instanceof Object);//编译不通过
</code></pre>
<pre><code class="highlight-chroma"> Integer integer = new Integer(1);
 System.out println(integer instanceof Integer);//true
</code></pre>

<pre><code class="highlight-chroma"> //false　　,在 JavaSE规范 中对 instanceof 运算符的规定就是：如果 obj 为 null，那么返回 false。
 System.out println(null instanceof Object);
</code></pre>
<h3 id="---和-equals-的区别是什么-">== 和 equals 的区别是什么？</h3>
"=="
对于基本类型和引用类型 == 的作用效果是不同的，如下所示：

基本类型：比较的是值是否相同；
引用类型：比较的是引用是否相同；

<pre><code class="highlight-chroma"> String x = "string";
 String y = string";
 String z = ew String("string");
 System.out println(x==y); // true
 System.out println(x==z); // false
 System.out println(x.equals(y)); // true
 System.out println(x.equals(z)); // true
</code></pre>
<blockquote>
因为 x 和 y 指向的是同一个引用，所以 == 也是 true，而 new String()方法则重写开了内存空间，所以 == 结果为 false，而 equals 比较的一直是值，所以结果都为 true。
</blockquote>
equals
equals 本质上就是 ==，只不过 String 和 Integer 等重写了 equals 方法，把它变成值比较。看下面的代码就明白了。
首先来看默认情况下 equals 比较一个有相同值的对象，代码如下：
<pre><code class="highlight-chroma"> class Cat {
     public Cat(String name) {
         this.name = name;
    }

     private String name;

     public String getName() {
         return name;
    }

     public void setName(String name) {
         this.name = name;
     }
 }

 Cat c1 = new Cat("叶痕秋");
 Cat c2 = new Cat("叶痕秋");
 System.out.println(c1.equals(c2)); // false
</code></pre>
输出结果出乎我们的意料，竟然是 false？这是怎么回事，看了 equals 源码就知道了源码如下：
<pre><code class="highlight-chroma"> public boolean equals(Object obj) {
       return (this == obj);
 }
</code></pre>
原来 equals 本质上就是 ==。
那问题来了，两个相同值的 String 对象，为什么返回的是 true？代码如下：
<pre><code class="highlight-chroma"> String s1 = new String("叶子");
 String s2 = new String("叶子");
 System.out.println(s1.equals(s2)); // true
</code></pre>
同样的，当我们进入 String 的 equals 方法，找到了答案，代码如下：
<pre><code class="highlight-chroma">public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n--

= 0) {
                if (v1[i]
= v2[i])
                    return
false;
                i++;
            }
            return true

        }
    }
    return false;
}
</code></pre>
原来是 String 重写了 Object 的 equals 方法，把引用比较改成了值比较。
/p>
总结
== 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；而 eq
als 默认情况下是引用比较，只是很多类重新了 equals 方法，比如 String、Integer 等把它变成了值
较，所以一般情况下 equals 比较的是值是否相等。
<h3 id="什么是java序列化-如何实现java序列化-或者请解释Serializable接口的作用-">
么是 java 序列化，如何实现 java 序列化？或者请解释 Serializable 接口的作用。</h3>
我们有时候将一个 java 对象变成字节流的形式传出去或者从一个字节流中恢复成一个 j
va 对象，例如，要将 java 对象存储到硬盘或者传送给网络上的其他计算机，这个过程我们可以自己
代码去把一个 java 对象变成某个格式的字节流再传输。
但是，jre 本身就提供了这种支持，我们可以调用 <code>OutputStream<
code> 的 <code>writeObject</code> 方法来做，如果要让 java
我们做，要被传输的对象必须实现 <code>serializable</code> 接口，这样，j
vac 编译时就会进行特殊处理，编译的类才可以被 <code>writeObject</code> <stron
>方法操作，这就是所谓的序列化。需要被序列化的类必须实现 <code>Serializable</co
e> 接口，该接口是一个 mini 接口，其中没有需要实现方法，implements Serializable
是为了标注该对象是可被序列化的。
例如，在 web 开发中，如果对象被保存在了 Session 中，tomcat 在重启时要把 Sess
on 对象序列化到硬盘，这个对象就必须实现 Serializable 接口。如果对象要经过分布式系统进行网
传输，被传输的对象就必须实现 Serializable 接口。
<h3 id="Hashcode的作用">Hashcode 的作用</h3>
java 的集合有两类，一类是 List，还有一类是 Set。前者有序可重复，后者无序不重
。当我们在 set 中插入的时候怎么判断是否已经存在该元素呢，可以通过 equals 方法。但是如果元
太多，用这样的方法就会比较满。
于是有人发明了哈希算法来提高集合中查找元素的效率。 这种方式将集合分成若干个
储区域，每个对象可以计算出一个哈希码，可以将哈希码分组，每组分别对应某个存储区域，根据一
对象的哈希码就可以确定该对象应该存储的那个区域。
hashCode 方法可以这样理解：它返回的就是根据对象的内存地址换算出的一个值。
样一来，当集合要添加新的元素时，先调用这个元素的 hashCode 方法，就一下子能定位到它应该放
的物理位置上。如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了
如果这个位置上已经有元素了，就调用它的 equals 方法与新元素进行比较，相同的话就不存了，不
同就散列其它的地址。这样一来实际调用 equals 方法的次数就大大降低了，几乎只需要一两次。</st
ong>
<h3 id="两个对象的-hashCode---相同--那么-equals---也一定为-true吗-">两个对象的
ashCode() 相同， 那么 equals() 也一定为 true 吗？</h3>
不对，两个对象的 hashCode() 相同，equals() 不一定 true。
代码示例：
<pre><code class="highlight-chroma"><span class="highlight
cl">String str1 = "keep";

String str2 = "brother";
System. out. println(String. format("str1：%d | str2：%d", str1. hashCode(),str2. hashCode()));
System. out. println(str1. equals(str2));
</code></pre>
执行结果:
<pre><code class="highlight-chroma">str1：1179395 | str2：1179395

false
</code></pre>
代码解读：很显然“keep”和“brother”的 hashCode() 相同，然而 equals() 则为 false，因为在散列表中，hashCode() 相等即两个键值对的哈希值相等，然而哈希值相等，并不一定能出键值对相等。
<h3 id="泛型常用特点">泛型常用特点</h3>
泛型是 Java SE 1.5 之后的特性， 《Java 核心技术》中对泛型的定义是：
<blockquote>
 “泛型” 意味着编写的代码可以被不同类型的对象所重用。
</blockquote>
 “泛型” ，顾名思义， “泛指的类型” 。我们提供了泛指的概念，但具体执行的时候可以有具体的规则来约束，比如我们用的非常多的 ArrayList 就是个泛型类，ArrayList 作为集合可以放各种元素，如 Integer, String，自定义的各种类型等，但在我们使用的时候通过具体的规则来约束如我们可以约束集合中只存放 Integer 类型的元素，如
<pre><code class="highlight-chroma">List<Integer> iniData = new ArrayList<>()
</code></pre>
<h3 id="使用泛型的好处-">使用泛型的好处？</h3>
以集合来举例，使用泛型的好处是我们不必因为添加元素类型的不同而定义不同类型集合，如整型集合类，浮点型集合类，字符串集合类，我们可以定义一个集合来存放整型、浮点型，符串型数据，而这并不是最重要的，因为我们只要把底层存储设置了 Object 即可，添加的数据全部可向上转型为 Object。 更重要的是我们可以通过规则按照自己的想法控制存储的数据类型。
<h2 id="数据类型">数据类型</h2>
<h3 id="Java有哪些数据类型">Java 有哪些数据类型</h3>
Java中有 8 种基本据类型，分别为：

6 种数字类型 （四个整数形，两个浮点型）：byte、short、int、long、float、double
1 种字符类型**：char**
1 种布尔型**：boolean。**

byte：

byte 数据类型是 8 位、有符号的，以二进制补码表示的整数；
最小值是 -128（-2^7）；
最大值是 127（2^7-1）；
默认值是 0；

byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 in
类型的四分之一；
例子：byte a = 100，byte b = -50。

short：

short 数据类型是 16 位、有符号的以二进制补码表示的整数
最小值是 -32768（-2^15）</st
ong>；
最大值是 32767（2^15 - 1）</
trong>；
Short 数据类型也可以像 byte 那样节省空间。一个 short 变量是 int 型变量所占空间
二分之一；
默认值是 0；</stro
g>
例子：short s = 1000，short r = -20000。

int：

int 数据类型是 32 位、有符号的以二进制补码表示的整数；
最小值是 -2,147,483,648（-2^31）</stro
g>；
最大值是 2,147,483,647（2^31 - 1）</st
ong>；
一般地整型变量默认为 int 类型；
默认值是 0 ；</stro
g>
例子：int a = 100000, int b = -200000。

long：

注意：Java 里使用 long 类型的数据一定要在数值后面加上 L，否则将作为
型解析
long 数据类型是 64 位、有符号的以二进制补码表示的整数；
最小值是 -9,223,372,036,854,775,808（-2^63）</st
ong>；
最大值是 9,223,372,036,854,775,807（2^63 -1）</s
rong>；
这种类型主要使用在需要比较大整数的系统上；
默认值是 0L；</str
ng>
例子： long a = 100000L，Long b = -200000L。**
**"L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辩。所以最好大写。
/strong>

float：

float 数据类型是单精度、32 位、符合 IEEE 754 标准的浮点数；
float 在储存大型浮点数组的时候可节省内存空间；
默认值是 0.0f；</st
ong>
浮点数不能用来表示精确的值，如货币；
例子：float f1 = 234.5f。

double：

double 数据类型是双精度、64 位、符合 IEEE 754 标准的浮点数；
浮点数的默认类型为 double 类型；
double 类型同样不能表示精确的值，如货币；
默认值是 0.0d；
例子：double d1 = 123.4。

char：

char 类型是一个单一的 16 位 Unicode 字符；
最小值是 \u0000（为 0）；
最大值是 \uffff（即 65535）；
char 数据类型可以储存任何字符；
例子：char letter = 'A';（单引号）

boolean：

boolean 数据类型表示一位的信息；
只有两个取值：true 和 false；
这种类型只作为一种标志来记录 true/false 情况；
默认值是 false；
例子：boolean one = true。

这八种基本类型都有对应的包装类分别为：Byte、Short、Integer、Long、loat、Double、Character、Boolean
<table>
<thead>
<tr>
<th>类型名称</th>
<th>字节、位数</th>
<th>最小值</th>
<th>最大值</th>
<th>默认值</th>
<th>例子</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 字节</td>
<td>1 字节，8 位</td>
<td>-128（-2^7）</td>
<td>127（2^7-1）</td>
<td>0</td>
<td>byte a = 100，byte b = -50</td>
</tr>
<tr>
<td>short 短整型</td>
<td>2 字节，16 位</td>

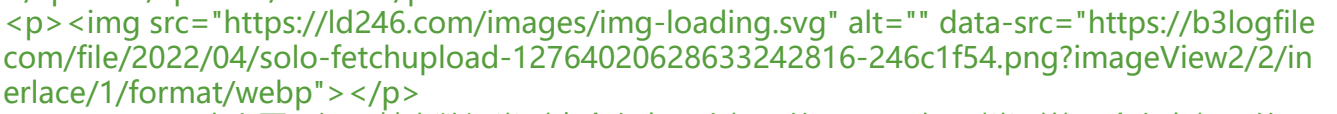
<td>-32768（-2^15）</td>
<td>32767（2^15 - 1）</td>
<td>0</td>
<td>short s = 1000，short r = -20000</td>
</tr>
<tr>
<td>int 整形</td>
<td>4 字节，32 位</td>
<td>-2,147,483,648（-2^31）</td>
<td>2,147,483,647（2^31 - 1）</td>
<td>0</td>
<td>int a = 100000, int b = -200000</td>
</tr>
<tr>
<td>lang 长整型</td>
<td>8 字节，64 位</td>
<td>-9,223,372,036,854,775,808（-2^63）</td>
<td>9,223,372,036,854,775,807（2^63 -1）</td>
<td>0L</td>
<td>long a = 100000L，Long b = -200000L</td>
</tr>
<tr>
<td>double 双精度</td>
<td>8 字节，64 位</td>
<td></td>
<td>double 类型同样不能表示精确的值，如货币</td>
<td>0.0d</td>
<td>double d1 = 123.4</td>
</tr>
<tr>
<td>float 单精度</td>
<td>4 字节，32 位</td>
<td>在储存大型浮点数组的时候可节省内存空间</td>
<td>不同统计精准的货币值</td>
<td>0.0f</td>
<td>float f1 = 234.5f</td>
</tr>
<tr>
<td>char 字符</td>
<td>2 字节，16 位</td>
<td>\u0000（即为 0）</td>
<td>\uffff（即为 65,535）</td>
<td>可以储存任何字符</td>
<td>char letter = 'A';</td>
</tr>
<tr>
<td>boolean 布尔</td>
<td>返回 true 和 false 两个值</td>
<td>这种类型只作为一种标志来记录 true/false 情况；</td>
<td>只有两个取值：true 和 false；</td>
<td>false</td>
<td>boolean one = true</td>
</tr>
</tbody>

</table>
<h3 id="Java中引用数据类型有哪些-它们与基本数据类型有什么区别-">Java 中引用数据型有哪些，它们与基本数据类型有什么区别？</h3>
引用数据类型分 3 种：类，接口，数组；
简单来说,只要不是基本数据类型.都是引用数据类型。 那他们有什么不同呢
1、从概念方面来说
1,基本数据类型:变量名指向具体的数值
2,引用数据类型:变量名不是指向具体的数值,而是指向存数据的内存地址,.也及时 hash
2、从内存的构建方面来说**(内存中,有堆内存和栈存两者)**
1,基本数据类型:被创建时,在栈内存中会被划分出一定的内存,并将数值存储在该内存中./strong>
2,引用数据类型:被创建时,首先会在栈内存中分配一块空间,然后在堆内存中也会分配一具体的空间用来存储数据的具体信息,即 hash 值,然后由栈中引用指向堆中的对象地址.</p

举个例子
<pre><code class="highlight-chroma">//基本数据类型作为方法参数被调用
public class Main{
    public static void main(String[] args){
        //基本数据类

        int i = 1;
        int j = 1;
        double d = 12;

        //引用数据类

        String str = "ello";
        String str1=Hello";
    }
}
</code></pre>

由上图可知，基本数据类型中会存在两个相同的 1，而引用型类型就不会存在相同的据。**
**假如"hello"的引用地址是 xxxxx1，声明 str 变量并其赋值"hello"实际上就是让 str 变量用了"hello"的内存地址，这个内存地址就存储在堆内存中，是不会改变的，当再次声明变量 str1 也赋值为"hello"时，此时就会在堆内存中查询是否有"hello"这个地址，如果堆内存中已经存在这个地了，就不会再次创建了，而是让 str1 变量也指向 xxxxx1 这个地址，如果没有的话，就会重新创建一地址给 str1 变量。
从使用方面来说
1,基本数据类型:判断数据是否相等，用=和!=判断。*
****2,引用数据类型:判断数据是否相等，用 equals()方法,**=**和!=是比较数值的。而 equals()方法比较内存地址的。**

补充：数据类型选择的原则

如果要表示整数就使用 int，表示小数就使用 double；
如果要描述日期时间数字或者表示文件（或内存）大小用 long；
如果要实现内容传递或者编码转换使用 byte；
如果要实现逻辑的控制，可以使用 booleam；
如果要使用中文，使用 char 避免中文乱码；
如果按照保存范围：byte < int < long < double;

<h3 id="--Java的四种引用-强弱软虚---">**Java 的四种引用，强弱软虚 **</h3>

强引用
强引用是平常中使用最多的引用，强引用在程序内存不足（OOM）的时候也不会被回收，用方式：
<pre><code class="highlight-chroma">String str = new String("str");
</code></pre>

软引用
**软引用在程序内存不足时，会被回收，使用方式： **
<pre><code class="highlight-chroma">// 注意：wrf这个引用也是强引用，它是指向SoftReference这个对象的，
// 这里的软引用指是指向new String("str")的引用，也就是SoftReference类中T
SoftReference<String> wrf = new SoftReference<String>(new String("str"));
</code></pre>
** 可用场景： 创建缓存的时候，创建的对象放进缓存中，当内存不足时，JVM 就会回收早先创的对象。**

弱引用
弱引用就是只要 JVM 垃圾回收器发现了它，就会将之回收，使用方式：
<pre><code class="highlight-chroma">WeakReference<String> wrf = new WeakReference<String>(str);
</code></pre>
** 可用场景： Java 源码中的*<code>java.util.WeakHashMap</code> 中的 <code>key</code> 就是使用弱引用，我的理解就是，一旦我不需要某个引用，JVM 会自动帮我处理它，这样我就不需做其它操作。

虚引用
虚引用的回收机制跟弱引用差不多，但是它被回收之前，会被放入 <code>ReferenceQueue</code> 中。注意哦，其它引用是被 JVM 回收后才被传入 <code>ReferenceQueue</code> 中的。由于这个机制，所以虚引用大多被用于引用销毁前的处工作。还有就是，虚引用创建的时候，必须带有 <code>ReferenceQueue</code>**，用例子： **
<pre><code class="highlight-chroma">PhantomReference<String> prf = new PhantomReference<String>(new String(str"), new ReferenceQueue<>());
</code></pre>

可用场景： 对象销毁前的一些操作，比如说资源释放等。**<code>Object.finalize()</cod
>**虽然也可以做这类动作，但是这个方式即不安全又低效 **

** 上诉所说的几类引用，都是指对象本身的引用，而不是指</str
ng> <code>Reference</code> 的四个子类的引用(<code>SoftReference<
code> 等)。 **
<h3 id="自动装箱与拆箱">自动装箱与拆箱</h3>

装箱**：将基本类型用它们对应的引用类型包装起来
**
拆箱**：将包装类型转换为基本数据类型；**

<h3 id="int-和-Integer-有什么区别">int 和 Integer 有什么区别</h3>

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
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
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
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</
trong>

<h3 id="Integer-a--127-与-Integer-b---127相等吗">Integer a= 127 与 Integer b = 1
7 相等吗</h3>

对于对象引用类型：==比较的是对象的内存地址。
对于基本数据类型：==比较的是值。

<code>如果整型字面量的值在-128到127之间，那么自动装箱时不会new新的Integer对象，而
直接引用常量池中的Integer对象，超过范围 a1==b1的结果是false</code>
<pre><code class="highlight-chroma"><span class="highlight
cl">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.print
n(a == b); // false 两个引用没有引用同一对象
  System.out.print
n(a == c); // true a自动拆箱成int类型再和c比较
  System.out.print
n(b == c); // true

    Integer a1 = 12
;

```
</span></span><span class="highlight-line"><span class="highlight-cl">    Integer b1 = 12
;
</span></span><span class="highlight-line"><span class="highlight-cl">    System.out.print
n(a1 == b1); // false
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    Integer a2 = 12
;
</span></span><span class="highlight-line"><span class="highlight-cl">    Integer b2 = 12
;
</span></span><span class="highlight-line"><span class="highlight-cl">    System.out.print
n(a2 == b2); // true
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h3 id="Java中的自动装箱与拆箱"><strong>Java 中的自动装箱与拆箱</strong></h3>
<p><strong><strong>什么是自动装箱拆箱？</strong></strong></p>
<p><strong>从下面的代码中就可以看到装箱和拆箱的过程</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">//自动装箱
</span></span><span class="highlight-line"><span class="highlight-cl">Integer total = 99;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">//自定拆箱
</span></span><span class="highlight-line"><span class="highlight-cl">int totalprim = tot
l;
</span></span></code></pre>
<p><strong><strong>装箱就是自动将基本数据类型转换为包装器类型；拆箱就是自动将包装器类
转换为基本数据类型。</strong></strong></p>
<blockquote>
<p>**在 Java SE5 之前，自动装箱要这样写：Integer i = **<code>new</code> <code>Integer
</code> <strong>10``);</strong></p>
</blockquote>
<p><strong>对于 Java 的自动装箱和拆箱，我们看看源码编译后的 class 文件，其实装箱调用包装
的 valueOf 方法，拆箱调用的是 Integer.Value 方法，下面就是变编译后的代码：</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2022/04/solo-fetchupload-15532215651249386565-f19f2058.png?imageView2/2/in
erlace/1/format/webp"></p>
<p><strong><strong>常见面试一：</strong></strong></p>
<p><strong>这段代码输出什么？</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Main {
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        Integer i1 =
00;
</span></span><span class="highlight-line"><span class="highlight-cl">        Integer i2 =
00;
</span></span><span class="highlight-line"><span class="highlight-cl">        Integer i3 =
00;
</span></span><span class="highlight-line"><span class="highlight-cl">        Integer i4 =
00;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln(i1==i2);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
```

```
ntln(i3==i4);
    }
}
```

答案是:

```
true
false
```

为什么会出现这样的结果？输出结果表明 i1 和 i2 指向的是同一个对象，而 i3 和 i4 向的是不同的对象。此时只需一看源码便知究竟，下面这段代码是 Integer 的 valueOf 方法的具体实 ：

```
public static Integer valueOf(int i) {
        if(i &gt;= -12 &amp;&amp; i &lt;= IntegerCache.high)
            return InterCache.cache[i + 128];
        else
            return new Integer(i);
    }
private static class IntegerCache {
        static final int high;
        static final Integer cache[];

        static {
            final int low = -128;

            // high value may be configured by property
            int h = 127

            if (integerCacheHighPropValue != null) {
                // Use Long.decode here to avoid invoking methods that
                // require Integer's autoboxing cache to be initialized
                int i = Long.decode(integerCacheHighPropValue).intValue();
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - -low);
            }
            high = h;
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">            cache = n
w Integer[(high - low) + 1];
</span></span><span class="highlight-line"><span class="highlight-cl">            int j = low;
</span></span><span class="highlight-line"><span class="highlight-cl">            for(int k =
; k &amp;lt; cache.length; k++)
</span></span><span class="highlight-line"><span class="highlight-cl">                cache[k]
= new Integer(j++);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        private Integ
rCache() {}
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span></code></pre>
<p><strong>从这 2 段代码可以看出，在通过 valueOf 方法创建 Integer 对象的时候，如果数值在[
128,127]之间，便返回指向 IntegerCache.cache 中已经存在的对象的引用；否则创建一个新的 Integ
r 对象。</strong></p>
<p><strong>上面的代码中 i1 和 i2 的数值为 100，因此会直接从 cache 中取已经存在的对象，所以
i1 和 i2 指向的是同一个对象，而 i3 和 i4 则是分别指向不同的对象。</strong></p>
<p><strong><strong>常见面试二：</strong></strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Main {
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        Double i1 =
00.0;
</span></span><span class="highlight-line"><span class="highlight-cl">        Double i2 =
00.0;
</span></span><span class="highlight-line"><span class="highlight-cl">        Double i3 =
00.0;
</span></span><span class="highlight-line"><span class="highlight-cl">        Double i4 =
00.0;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln(i1==i2);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln(i3==i4);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>输出结果为：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">false
</span></span><span class="highlight-line"><span class="highlight-cl">false
</span></span></code></pre>
<p><strong>原因很简单，在某个范围内的整型数值的个数是有限的，而浮点数却不是。</strong
</p>
<h3 id="为什么要有包装类型-"><strong>为什么要有包装类型？</strong></h3>
<p><strong><strong>让基本数据类型也具有对象的特征</strong></strong></p>
<table>
<thead>
<tr>
<th><strong>基本类型</strong></th>
```

<th>包装器类型</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
</tbody>
</table>
为了让基本类型也具有对象的特征，就出现了包装类型（如我们在使用集合类型 Collection 时就一定要使用包装类型而非基本类型）因为容器都是装 object 的，这是就需要这些基本类型包装器类了。
自动装箱：<code>new Integer(6);</code>，底层调用:<code>Integer.valueOf(6)</code>
自动拆箱: <code>int i = new Integer(6);</code>，底层调用 <code>i.intValue();</code> 方法实现。
<pre><code class="highlight-chroma">Integer i = 6;
Integer j = 6;
System.out.println(i==j);
</code></pre>
答案在下面这段代码中找：
<pre><code class="highlight-chroma">public static Integer valueOf(int i) {

    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
</code></pre>
二者的区别：

声明方式不同：基本类型不使用 new 关键字，而包装类型需要使用 new 关键字来在堆中分配存储空间**；**
存储方式及位置不同：基本类型是直接将变量值存储在栈中，而包装类型是将对象放堆中，然后通过引用来使用；
初始值不同：基本类型的初始值如 int 为 0，boolean 为 false，而包装类型的初始值为 null；
使用方式不同：基本类型直接赋值直接使用就好，而包装类型在集合如 Collection、Map 时会使用到。

<h3 id="a-a-b与a--b有什么区别吗-">a=a+b 与 a+=b 有什么区别吗?</h3>
<code>+=</code>**操作符会进行隐式自动类型转换,此处 a+=b 隐式的将加操作的结果类型制转换为持有结果的类型,而 a=a+b 则不会自动进行类型转换.如：**
<pre><code class="highlight-chroma">byte a = 127;
byte b = 127;
b = a + b; // 报编错误:cannot convert from int to byte
b += a;
</code></pre>
以下代码是否有错,有的话怎么改？
<pre><code class="highlight-chroma">short s1= 1;
s1 = s1 + 1;
</code></pre>
有错误.short 类型在进行运算时会自动提升为 int 类型,也就是说 <code>s+1</code> 的运算结果是 int 类型,而 s1 是 short 类型,此时编译器会报错.

正确写法：
<pre><code class="highlight-chroma">short s1= 1;
s1 += 1;
</code></pre>
<code>+=</code> 操作符会对右边的表达式结果强转匹配左边的数据类型,所以没错
<h3 id="float-f-3-4-是否正确">float f=3.4;是否正确</h3>
**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;
<h3 id="short-s1---1--s1---s1---1-有错吗-short-s1---1--s1----1-有错吗">short s1 = 1 s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗</h3>

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);

中有隐含的强制类型转换。

<h3 id="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-">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，将会出现么现象？</h3>
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，如果制转化，int 类型的高 24 位将会被丢弃，因为 byte 类型的范围是从 -128 到 127
<h2 id="常用关键字">常用关键字</h2>
<h3 id="final-在-Java-中有什么作用-">final 在 Java 中有什么作用？</h3>
final 作为 Java 中的关键字可以用于三个地方。用于修饰类、类属性和类方法。</strog>
特征：凡是引用 final 关键字的地方皆不可修改！
(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）。
<h3 id="访问修饰符-public-private-protected-以及不写-默认-时的区别">访问修饰符 public,private,protected,以及不写（默认）时的区别</h3>

定义**：Java 中，可以使用访问修饰符来保护对类变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。**
分类

private** ：在同一类内可见。使用对象：变量、方。 注意：不能修饰类（外部类）**
default** (即缺省，什么也不写，不使用任何关键字) 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。**
protected** ：对同一包内的类和所有子类可见。使对象：变量、方法。 注意：不能修饰类（外部类）。**
public** : 对所有类可见。使用对象：类、接口、变、方法**

访问修饰符图
<h3 id="final有哪些用法-">final 有哪些用法?</h3>
final 也是很多面试喜欢问的地方,但我觉得这个问题很无聊,通常能回答下以下 5 点就错了:

被 final 修饰的类不可以被继承
被 final 修饰的方法不可以被重写
被 final 修饰的变量不可以被改变.如果修饰引用,那么表示引用不可变,引用指向的内容变.
被 final 修饰的方法,JVM 会尝试将其内联,以提高运行效率
被 final 修饰的常量,在编译阶段会存入常量池中.

除此之外,编译器对 final 域要遵守的两个重排序规则更好:
在构造函数内对一个 final 域的写入,与随后把这个被构造对象的引用赋值给一个引用量,这两个操作之间不能重排序 初次读一个包含 final 域的对象的引用,与随后初次读这个 final 域,这两操作之间不能重排序.
<h3 id="final-有什么用-">final 有什么用？</h3>
用于修饰类、属性和方法

被 final 修饰的类不可以被继承
被 final 修饰的方法不可以被重写

被 final 修饰的变量不可以被改变，被 final 修饰不可变的是变量的引用，而不是引用向的内容，引用指向的内容是可以改变的

<h3 id="static都有哪些用法-">static 都有哪些用法?</h3>
所有的人都知道 static 关键字这两个基本的用法:静态变量和静态方法.也就是被 static 所修饰的变量/方法都属于类的静态资源,类实例所共享.
除了静态变量和静态方法之外,static 也用于静态块,多用于初始化操作:
<pre><code class="highlight-chroma">public calss PreCache{
    static{
        //执行相关操

    }
}
</code></pre>
此外 static 也多用于修饰内部类,此时称之为静态内部类.
最后一种用法就是静态导包,即 <code>import static</code>.mport static 是在 JDK 1.5 之后引入的新特性,可以用来指定导入某个类中的静态资源,并且不需要使类名,可以直接使用资源名,比如:
<pre><code class="highlight-chroma">import static java.lang.Math.*;

public class Test{

    public static vo d main(String[] args){
        //System.out println(Math.sin(20));传统做法
        System.out.pr ntln(sin(20));
    }
}
</code></pre>
<h3 id="static存在的主要意义">static 存在的主要意义</h3>

static 的主要意义是在于创建独立于具体对象的域变量或者方法。strong>以致于即使没有创建对象，也能使用属性和调用方法！
static 关键字还有一个比较关键的作用就是 用来形成态代码块以优化程序性能。static 块可以置于类中的任何地方，类可以有多个 static 块。在类初次被加载的时候，会按照 static 块的顺序来执行每个 static 块，并且只执行一次。
为什么说 static 块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行次。因此，很多时候会将一些只需要进行一次的初始化操作都放在 static 代码块中进行。

<h3 id="static的独特之处">static 的独特之处</h3>

1、被 static 修饰的变量或者方法是独立于该类的任何对象，也就是说，这些变量和方不属于任何一个实例对象，而是被类的实例对象所共享**。**

<blockquote>
怎么理解 “被类的实例对象所共享” 这句话呢？就是说，一个类的静态成员，它是属

大伙的【大伙指的是这个类的多个对象实例，我们都知道一个类可以创建多个实例！】，所有的类对共享的，不像成员变量是自个的【自个指的是这个类的单个实例对象】...我觉得我已经讲的很通俗了你明白了咩？</p>
</blockquote>

2、在该类被第一次加载的时候，就会去加载被 static 修饰的部分，而且只在类第一次用时加载并进行初始化，注意这是第一次用就要初始化，后面根据需要是可以再次赋值的。
3、static 变量值在类加载的时候分配空间，以后创建类对象的时候不会重新分配。赋的话，是可以任意赋值的！
4、被 static 修饰的变量或者方法是优先于对象存在的，也就是说当一个类加载完毕之，即便没有创建对象，也可以去访问。

<h3 id="static注意事项">static 注意事项</h3>

1、静态只能访问静态。
2、非静态既可以访问非静态的，也可以访问静态的。

<h3 id="static和final区别">static 和 final 区别</h3>
<table>
<thead>
<tr>
<th>关键词</th>
<th>修饰物</th>
<th>影响</th>
</tr>
</thead>
<tbody>
<tr>
<td>final</td>
<td>变量</td>
<td>分配到常量池中，程序不可改变其值</td>
</tr>
<tr>
<td>final</td>
<td>方法</td>
<td>子类中将不能被重写</td>
</tr>
<tr>
<td>final</td>
<td>类</td>
<td>不能被继承</td>
</tr>
<tr>
<td>static</td>
<td>变量</td>
<td>分配在内存堆上，引用都会指向这一个地址而不会重新分配内存</td>
</tr>
<tr>
<td>static</td>
<td>方法块</td>
<td>虚拟机优先加载</td>
</tr>
<tr>

<td>static</td>
<td>类</td>
<td>可以直接通过类来调用而不需要 new</td>
</tr>
</tbody>
</table>
<h3 id="this关键字的用法">this 关键字的用法</h3>

this 是自身的一个对象，代表对象本身，可以理解为：指向对象本身的一个指针。</st
ong>

this 的用法在 java 中大体可以分为 3 种：

1.普通的直接引用，this 相当于是指向当前对象本身。
2.形参与成员名字重名，用 this 来区分：

<pre><code class="highlight-chroma"><span class="highlight
cl">public Person(String name, int age) {
    this.name = na
e;
    this.age = age;
}
</code></pre>

3.引用本类的构造函数

<pre><code class="highlight-chroma"><span class="highlight
cl">class Person{
    private String n
me;
    private int age;

    public Person()

    }

    public Person(S
ring name) {
        this.name =
ame;
    }
    public Person(S
ring name, int age) {
        this(name);
        this.age = ag
;
    }
}
</code></pre>

<h3 id="super关键字的用法">super 关键字的用法</h3>

<p>super 可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己近的一个父类。</p>
<p>super 也有三种用法：</p>

1.普通的直接引用

<p>** 与 this 类似，super 相当于是指向当前对象的父类的引用，这样就可以用 super.xxx 来引用类的成员。**</p>

<strong>2.子类中的成员变量或方法与父类中的成员变量或方法同名时，用 super 进行区分</st
ong>

<pre><code class="highlight-chroma"><span class="highlight
cl">   class Person{
   protected
tring name;

   public Per
on(String name) {
   this.na
e = name;
   }

   }

   class Student
extends Person{
   private Str
ng name;

   public Stu
ent(String name, String name1) {
   super(n
me);
   this.na
e = name1;
   }

   public void
getInfo(){
   System.
ut.println(this.name);   //Child
   System.
ut.println(super.name);   //Father
   }

   }

   public class T
st {
   public stat
c void main(String[] args) {
<span class="highlight-line"><span class="highlight-cl">   Student
1 = new Student("Father","Child");

            s1.getInf
();

        }
    }
</code></pre>
** 3.引用父类构造函数**

super（参数）：调用父类中的某一个构造函数（应该为构造函数中的第一条语句）。
/strong>
this（参数）：调用本类中另一种形式的构造函数（应该为构造函数中的第一条语句）

<h3 id="-和--的区别">&和&&的区别</h3>

&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
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
，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
。

<code>注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。</code>
<h3 id="break--continue--return-的区别及作用">break ,continue ,return 的区别及作
</h3>

break 跳出总上一层循环，不再执行循环(结束当前的循环体)
continue 跳出本次循环，继续执行下次循环(结束正在执行的循环 进入下一个循环条件

return 程序返回，不再执行下面的代码(结束当前的方法 直接返回)

<h2 id="异常处理">异常处理</h2>
<h3 id="Java异常简介">Java 异常简介</h3>
Java 异常是 Java 提供的一种识别及响应错误的一致性机制。
Java 异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优
，并提高程序健壮性。在有效使用异常的情况下，异常能清晰的回答 what, where, why 这 3 个问题
异常类型回答了“什么”被抛出，异常堆栈跟踪回答了“在哪”抛出，异常信息回答了“为什么”会
出。
<h3 id="Java异常架构">Java 异常架构</h3>
<img src="https://ld246.com/images/img-loading.svg" alt="image-20210814134319791"
data-src="https://b3logfile.com/file/2022/04/solo-fetchupload-1102754489267873709 6-f0c1
0e7.jpeg?imageView2/2/interlace/1/format/webp">
<h3 id="Error-和-Exception-有什么区别-">Error 和 Exception 有什么区别？</strong
</h3>
****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
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
示如果程序运行正常，从不会发生的情况。
<h3 id="阐述-final-finally-finalize-的区别">阐述 final、finally、finalize 的区别</stron
></h3>
1、final：** 修饰符（关键字）有三种用法：如果一
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。
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，

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。**
/p>
2、finally：** 通常放在 try...catch...的后面构造总
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，
以将释放外部资源的代码写在 finally 块中.**
3、finalize：** Object 类中定义的方法，Java 中允
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
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。**
<h3 id="列出一些你常见的运行时异常-">列出一些你常见的运行时异常？</h
>

<code>ClassCastException</code>（类转换异常）
<code>IndexOutOfBoundsException</code>**（数组越界）**
<code>NullPointerException</code>**（空指针异常）**
<code>ArrayStoreException</code>（数据存储异常，操作数组是类型不一致）</
trong>
<code>BufferOverflowException</code>
<code>ArithmeticException</code>** 算术异常**
<code>IllegalArgumentException</code> 非法参数异常
<code>SecurityException</code>** 安全异常**

<h3 id="什么是受检异常">什么是受检异常</h3>
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
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
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
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：
1、** 不要将异常处理用于正常的控制流（设计良好
API 不应该强迫它的调用者为了正常的控制流而使用异常）**
2、** 对可以恢复的情况使用受检异常，对编程错误
用运行时异常**
3、** 避免不必要的使用受检异常（可以通过一些状
检测手段来避免异常的发生）**
4、** 优先使用标准的异常**
5、** 每个方法抛出的异常都要有文档**
6、** 保持异常的原子性**
7、** 不要在 catch 中忽略掉捕获到的异常**
<h3 id="Excption与Error包结构">Excption 与 Error 包结构</h3>
Java 可抛出(Throwable)的结构分为三种类型：被检
的异常(CheckedException)，运
时异常(RuntimeException)，错误(
rror)。
1、运行时异常
定义：
RuntimeException 及其子类都被称为运行时异常。
特点：
Java 编译器不会检查它。也就是说，当程序中可能出现这类异常时，倘若既"没有通过
hrows 声明抛出它"，也"没有用 try-catch 语句捕获它"，还是会编译通过。例如，除数为零时产生的
rithmeticException 异常，数组越界时产生的 IndexOutOfBoundsException 异常，fail-fast 机制
生的 ConcurrentModificationException 异常（java.util 包下面的所有的集合类都是快速失败的，
快速失败”也就是 fail-fast，它是 Java 集合的一种错误检测机制。当多个线程对集合进行结构上的

变的操作时，有可能会产生 fail-fast 机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程 1、线程 2），线程 1 通过 Iterator 在遍历集合 A 中的元素，在某个时候线程 2 修改了集合 A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制，这个错叫并发修改异常。Fail-safe，java.util.concurrent 包下面的所有的类都是安全失败的，在遍历过程中，如果已经遍历的数组上的内容变化，迭代器不会抛出 ConcurrentModificationException 异常。如果未遍历的数组上的内容发生了变，则有可能反映到迭代过程中。这就是 ConcurrentHashMap 迭代器弱一致的表现。ConcurrentHashMap 的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到使用锁，甚至是全局锁，这就与 Hashtable 和同步的 HashMap 一样了。）等，都属于运行时异常。</p>
<p>2、被检查异常</p>
<p>定义：** Exception 类本身，以及 Exception 的子中除了"运行时异常"之外的其它子类都属于被检查异常。**</p>
<p>特点 ： ** Java 编译器会检查它。 此类异常，要么过 throws 进行声明抛出，要么通过 try-catch 进行捕获处理，否则不能通过编译。例如，CloneNotupportedException 就属于被检查异常。**</p>
<p>当通过 clone()接口去克隆一个对象，而该对象对应的类没有实现 Cloneable 接口，会抛出 CloneNotSupportedException 异常。被检查异常通常都是可以恢复的。 如：</>
<p><code>IOException</code></p>
<p><code>FileNotFoundException</code></p>
<p><code>SQLException</code></p>
<p>被检查的异常适用于那些不是因程序引起的错误情况，比如：读取文件时文件不存在发的 <code>FileNotFoundException</code>。然而，不被检查的异常通常是由于糟糕的编程引起的，比如：在对象引用时没有确保对象非空而引起的 <code>NullPinterException</code>。</p>
<p>3、错误</p>
<p>定义 : Error 类及其子类。</p>
<p>特点 : 和运行时异常一样，编译器也不会对错误进行检查。</p>
<p>当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程本身无法修复这些错误的。例如，VirtualMachineError 就属于错误。出现这种错误会导致程序终止行。OutOfMemoryError、ThreadDeath。</p>
<p>Java 虚拟机规范规定 JVM 的内存分为了好几块，比如堆，栈，程序计数器，方法区</p>
<h3 id="try---里有一个return语句-那么紧跟在这个try后的finally--里的code会不会被执行-什么时被执行-在return前还是后-">try {}里有一个 return 语句，那么紧跟在这个 try 后的 finally{里的 code 会不会被执行，什么时候被执行，在 return 前还是后?</h3>
<p>我们知道 finally{}中的语句是一定会执行的，那么这个可能正常脱口而出就是 return 前，return 之后可能就出了这个方法了，鬼知道跑哪里去了，但更准确的应该是在 return 中间执行请看下面程序代码的运行结果：</p>
<pre><code class="highlight-chroma">public classTest {
   public static vod main(String[]args) {
   System.out.prntln(newTest().test());;
   }
   static int test()
   {
   intx = 1;
   try
   {
   return x;
<span class="highlight-line"><span class="highlight-cl">   }

```
</span></span><span class="highlight-line"><span class="highlight-cl">      finally
</span></span><span class="highlight-line"><span class="highlight-cl">      {
</span></span><span class="highlight-line"><span class="highlight-cl">          ++x;
</span></span><span class="highlight-line"><span class="highlight-cl">      }
</span></span><span class="highlight-line"><span class="highlight-cl">  }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>执行结果如下：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">1
</span></span></code></pre>
<p><strong>运行结果是 1，为什么呢？主函数调用子函数并得到结果的过程，好比主函数准备一空罐子，当子函数要返回结果时，先把结果放在罐子里，然后再将程序逻辑返回到主函数。所谓返回就是子函数说，我不运行了，你主函数继续运行吧，这没什么结果可言，结果是在说这话之前放进罐里的。</strong></p>
<h3 id="运行时异常与一般异常有何异同-"><strong>运行时异常与一般异常有何异同？</strong></h3>
<p><strong>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可能遇到的异常，是一种常见运行错误。java 编译器要求方法必须声明抛出可能发生的非运行时异常但是并不要求必须声明抛出未被捕获的运行时异常。</strong></p>
<h3 id="error和exception有什么区别-"><strong>error 和 exception 有什么区别?</strong></h>
<p><strong>error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可指望程序能处理这样的情况。exception 表示一种设计或实现问题。也就是说，它表示如果程序运行常，从不会发生的情况。</strong></p>
<h3 id="简单说说Java中的异常处理机制的简单原理和应用-"><strong>简单说说 Java 中的异常处机制的简单原理和应用。</strong></h3>
<p><strong><strong>异常是指 java 程序运行时（非编译）所发生的非正常情况或错误</strong</strong>**，与现实生活中的事件很相似，现实生活中的事件可以包含事件发生的时间、地点、人、情节等信息，可以用一个对象来表示，Java 使用面向对象的方式来处理异常，它把程序中发生的每异常也都分别封装到一个对象来表示的，该对象中包含有异常的信息。**</p>
<p>** Java 对异常进行了分类，不同类型的异常分别用不同的 Java 类表示，所有异常的根类为 java.ang.Throwable。**</p>
<p><strong>Throwable 下面又派生了两个子类：</strong></p>
<ul>
<li><strong>Error 和 Exception，Error 表示应用程序本身无法克服和恢复的一种严重问题，程序有奔溃了，例如，说内存溢出和线程死锁等系统问题。</strong></li>
<li><strong>Exception 表示程序还能够克服和恢复的问题，其中又分为系统异常和普通异常：</stong></li>
</ul>
<p><strong>系统异常是软件本身缺陷所导致的问题，也就是软件开发人员考虑不周所导致的问题软件使用者无法克服和恢复这种问题，但在这种问题下还可以让软件系统继续运行或者让软件挂掉，如，数组脚本越界（ArrayIndexOutOfBoundsException），空指针异常（NullPointerException、类转换异常（ClassCastException）；</strong></p>
<p><strong>普通异常是运行环境的变化或异常所导致的问题，是用户能够克服的问题，例如，网断线，硬盘空间不够，发生这样的异常后，程序不应该死掉。</strong></p>
<p><strong>java 为系统异常和普通异常提供了不同的解决方案，编译器强制普通异常必须 try..catch 处理或用 throws 声明继续抛给上层调用方法处理，所以普通异常也称为 checked 异常，而系统异可以处理也可以不处理，所以，编译器不强制用 try..catch 处理或用 throws 声明，所以系统异常也为 unchecked 异常。</strong></p>
<h2 id="面向对象"><strong>面向对象</strong></h2>
<h3 id="面向对象和面向过程的区别"><strong>面向对象和面向过程的区别</strong></h3>
<ul>
<li><strong><strong>面向过程</strong></strong>**：**</li>
```

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发，性能是最重要的因素。
缺点：没有面向对象易维护、易复用、易扩展
面向对象**：**

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以计出低耦合的系统，使系统更加灵活、更加易于维护
缺点：性能比面向过程低

<code>面向过程是具体化的，流程化的，解决一个问题，你需要一步一步的分析，一步一步的现。</code>
<code>面向对象是模型化的，你只需抽象出一个类，这是一个封闭的盒子，在这里你拥有数据拥有解决问题的方法。需要什么功能直接使用就可以了，不必去一步一步的实现，至于这个功能是如实现的，管我们什么事？我们会用就可以了。</code>
<code>面向对象的底层其实还是面向过程，把面向过程抽象成类，然后封装，方便我们使用的是面向对象了。</code>
<h3 id="面向对象的特征有哪些方面">面向对象的特征有哪些方面</h3>
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节是什么。**
封装把一个对象的属性私有化，同时提供些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。
继承是使用已存在的类的定义作为基础建新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承类。通过使用继承我们能够非常方便地复用以前的代码。
关于继承如下 3 点请记住：
子类拥有父类非 private 的属性和方法。
子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
子类可以用自己的方式实现父类的方法。（以后介绍）。
多态**：父类或接口定义的引用变量可以指向子类或体实现类的实例对象。提高了程序的拓展性。在 Java 中有两种形式可以实现多态：继承（多个子类同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。**

<h3 id="Java多态的理解">Java 多态的理解</h3>

多态是继封装、继承之后，面向对象的第三大特性。
多态现实意义理解：

现实事物经常会体现出多种形态，如学生，学生是人的一种，则一个具体的同学张三是学生也是人，即出现两种形态。
Java 作为面向对象的语言，同样可以描述一个事物的多种形态。如 Student 类继承了erson 类，一个 Student 的对象便既是 Student，又是 Person。

多态体现为父类引用变量可以指向子类对象。
前提条件：必须有子父类关系。

注意：在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法

多态的定义与使用格式

定义格式：父类类型 变量名=new 子类类型();
<h3 id="什么是多态机制-Java语言是如何实现多态的-">什么是多态机制？Java 语言是如实现多态的？</h3>

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代，让程序可以选择多个运行状态，这就是多态性。
多态分为编译时多态和运行时多态。其中编辑时多态是静态的，主要是指方法的重载它是根据参数列表的不同来区分不同的函数，通过编辑之后会变成两个不同的函数，在运行时谈不上态。而运行时多态是动态的，它是通过动态绑定来实现的，也就是我们所说的多态性。</l>

多态的实现

Java 实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。
重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。/strong>
向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技调用父类的方法和子类的方法。

<code>只有满足了上述三个条件，我们才能够在同一个继承结构中使用统一的逻辑实现代码处不同的对象，从而达到执行不同的行为。</code>
<code>对于Java而言，它多态的实现机制遵循一个原则：当超类对象引用变量引用子类对象时被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在类中定义过的，也就是说被子类覆盖的方法。</code>
<h3 id="面向对象五大基本原则是什么-可选-">面向对象五大基本原则是什么（可选）</h3>

单一职责原则 SRP(Single Responsibility Principle)**
**** 类的功能要单一，不能包罗万象，跟杂货铺似的。**
开放封闭原则 OCP(Open - Close Principle)**
**** 一个模块对于拓展是开放的，对于修改是封闭的，想要增加功能热烈欢迎，想要修改，哼，一万不乐意。**
里式替换原则 LSP(the Liskov Substitution Principle LSP)**
**** 子类可以替换父类出现在父类能够出现的任何地方。比如你能代表你爸去你姥姥家干活。哈哈~~*
依赖倒置原则 DIP(the Dependency Inversion Principle DIP)**
**** 高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现具体实现应该依赖于抽象。就是你出国要说你是中国人，而不能说你是哪个村子的。比如说中国人是象的，下面有具体的 xx 省，xx 市，xx 县。你要依赖的抽象是中国人，而不是你是 xx 村的。**
接口分离原则 ISP(the Interface Segregation Principle ISP)**
**** 设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。就比如一个手机拥有打电，看视频，玩游戏等功能，把这几个功能拆分成不同的接口，比在一个接口里要好的多。**

<h3 id="抽象类和接口的对比">抽象类和接口的对比</h3>

抽象类是用来捕捉子类的通用特性的。接口是抽象方法的集合。
从设计层面来说，抽象类是对类的抽象，是一种模板设计，接口是行为的抽象，是一行为的规范。

相同点

接口和抽象类都不能实例化
都位于继承的顶端，用于被其他实现或继承
都包含抽象方法，其子类都必须覆写这些抽象方法

不同点
<table>
<thead>
<tr>
<th>参数</th>
<th>抽象类</th>
<th>接口</th>
</tr>
</thead>
<tbody>
<tr>
<td>声明</td>
<td>抽象类使用 abstract 关键字声明</td>
<td>接口使用 interface 关键字声明</td>
</tr>
<tr>
<td>实现</td>
<td>子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽类中所有声明的方法的实现</td>
<td>子类使用 implements 关键字来实现接口。它需要提供接口中所有声明的方法的实现/strong></td>
</tr>
<tr>
<td>构造器</td>
<td>抽象类可以有构造器</td>
<td>接口不能有构造器</td>
</tr>
<tr>
<td>访问修饰符</td>
<td>抽象类中的方法可以是任意访问修饰符</td>
<td>接口方法默认修饰符是 public。并且不允许定义为 private 或者 protected</strong
</td>
</tr>
<tr>
<td>多继承</td>
<td>一个类最多只能继承一个抽象类</td>
<td>一个类可以实现多个接口</td>
</tr>
<tr>
<td>字段声明</td>
<td>抽象类的字段声明可以是任意的</td>

<td>接口的字段默认都是 static 和 final 的</td>
</tr>
</tbody>
</table>
备注**：Java8 中接口中引入默认方法和静态方法，此来减少抽象类和接口之间的差异。**
<code>现在，我们可以为接口提供默认实现的方法了，并且不用强制子类来实现它。</code>

接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：

行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量少用抽象类。
选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用的功能。

<h3 id="普通类和抽象类有哪些区别-">普通类和抽象类有哪些区别？</h3>

普通类不能包含抽象方法，抽象类可以包含抽象方法。
抽象类不能直接实例化，普通类可以直接实例化。

<h3 id="抽象类能使用-final-修饰吗-">抽象类能使用 final 修饰吗？</h3>

不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承，这样彼就会产生矛盾，所以 final 不能修饰抽象类

<h3 id="创建一个对象用什么关键字-对象实例与对象引用有何不同-">创建一个对象用什关键字？对象实例与对象引用有何不同？</h3>

new 关键字，new 创建对象实例（对象实例在堆内存中），对象引用指向对象实例（象引用存放在栈内存中）。一个对象引用可以指向 0 个或 1 个对象（一根绳子可以不系气球，也可以一个气球）;一个对象可以有 n 个引用指向它（可以用 n 条绳子系住一个气球）

<h4 id="构造器-constructor-是否可被重写-override-">构造器（constructor）是否可重写（override）</h4>

构造器不能被继承，因此不能被重写，但可以被重载。

<h4 id="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-">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？</h4>

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，而者实现的是运行时的多态性。
重载：发生在同一个类中，方法名相同参数列表不同（参数类型不同、个数不同、顺不同），与方法返回值和访问修饰符无关，即重载的方法不能根据返回类型进行区分
重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛出的常小于等于父类，访问修饰符大于等于父类（里氏代换原则）；如果父类方法访问修饰符为 private 子类中就不是重写。

<h2 id="类-变量-方法">类、变量、方法</h2>

<h3 id="什么是内部类-">什么是内部类？</h3>

在 Java 中，可以将一个类的定义放在另外一个类的定义内部，这就是内部类**。内部类本身就是类的一个属性，与其他属性定义方一致。**

<h3 id="内部类的分类有哪些">内部类的分类有哪些</h3>
<code>内部类可以分为四种：**成员内部类、局部内部类、匿名内部类和静态内部类**。</code>
<h5 id="静态内部类">静态内部类</h5>

定义在类内部的静态类，就是静态内部类。

```
public class Outer {

    private static int radius = 1;

    static class StaticInner {

        public void visit() {

            System.out.println("visit outer static  variable:" + radius);

        }

    }

}
```


静态内部类可以访问外部类所有的静态变量，而不可访问外部类的非静态变量；静态部类的创建方式，<code>new 外部类.静态内部类()</code>，如下：

```
Outer.StaticInner inner = new Outer.StaticInner();

inner.visit();
```


<h5 id="成员内部类">成员内部类</h5>

定义在类内部，成员位置上的非静态类，就是成员内部类。

```
public class Outer {

    private static int radius = 1;

    private int cont =2;

     class Inner {

        public void visit() {

            System.out.println("visit outer static  variable:" + radius);
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">            System.o
t.println("visit outer   variable:" + count);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
</li>
<li><strong>成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公有。成
内部类依赖于外部类的实例，它的创建方式</strong> <code>外部类实例.new 内部类()</code><s
rong>，如下：</strong>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  Outer outer = new Outer();
</span></span><span class="highlight-line"><span class="highlight-cl">  Outer.Inner inner
= outer.new Inner();
</span></span><span class="highlight-line"><span class="highlight-cl">  inner.visit();
</span></span></code></pre>
</li>
</ul>
<h5 id="局部内部类"><strong>局部内部类</strong></h5>
<ul>
<li><strong>定义在方法中的内部类，就是局部内部类。</strong>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  public class Outer {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private  int ou
_a = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">    private static i
t STATIC_b = 2;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public void tes
FunctionClass(){
</span></span><span class="highlight-line"><span class="highlight-cl">        int inner_c
3;
</span></span><span class="highlight-line"><span class="highlight-cl">        class Inner {
</span></span><span class="highlight-line"><span class="highlight-cl">            private vo
d fun(){
</span></span><span class="highlight-line"><span class="highlight-cl">                System
out.println(out_a);
</span></span><span class="highlight-line"><span class="highlight-cl">                System
out.println(STATIC_b);
</span></span><span class="highlight-line"><span class="highlight-cl">                System
out.println(inner_c);
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        Inner  inner
= new Inner();
</span></span><span class="highlight-line"><span class="highlight-cl">        inner.fun();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d testStaticFunctionClass(){
</span></span><span class="highlight-line"><span class="highlight-cl">        int d =3;
</span></span><span class="highlight-line"><span class="highlight-cl">        class Inner {
</span></span><span class="highlight-line"><span class="highlight-cl">            private vo
d fun(){
```

                // System.out.println(out_a); 编译错误，定义在静态方法中的局部类不可以访问外部类的实例变量
System.out.println(STATIC_b);
                System.out.println(d);
            }
        }
        Inner inner = new Inner();
        inner.fun();
    }
  }
</code></pre>

定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的部类只能访问外部类的静态变量和方法。局部内部类的创建方式，在对应方法内，<code>new 内部类()</code>，如下：
<pre><code class="highlight-chroma"> public static void testStaticFunctionClass(){
    class Inner {
    }
    Inner inner = new Inner();
  }
</code></pre>

<h5 id="匿名内部类">匿名内部类</h5>

匿名内部类就是没有名字的内部类，日常开发中使用的比较多。
<pre><code class="highlight-chroma"> public class Outer {

    private void test(final int i) {
        new Service() {
            public void method() {
                for (int j = 0; j < i; j++) {
                    System.out.println("匿名内部类");
                }
            }
        }.method();
    }
  }
  //匿名内部类必继承或实现一个已有的接口
interface Service{
    void method();
  }

</code></pre>

除了没有名字，匿名内部类还有以下特点：

匿名内部类必须继承一个抽象类或者实现一个接口。
匿名内部类不能定义任何静态成员和静态方法。
当所在的方法的形参需要被匿名内部类使用时，必须声明为 final。
匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。</trong>

匿名内部类创建方式：
<pre><code class="highlight-chroma"> new 类/接口{
    //匿名内部类实部分
}
</code></pre>

<h3 id="内部类的优点">内部类的优点</h3>
<code>我们为什么要使用内部类呢？因为它有以下优点：</code>

一个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！

内部类不为同一包的其他类所见，具有很好的封装性；
内部类有效实现了“多重继承”，优化 java 单继承的缺陷。
匿名内部类可以很方便的定义回调。

<h3 id="内部类有哪些应用场景">内部类有哪些应用场景</h3>

一些多算法场合
解决一些非面向对象的语句块。
适当使用内部类，使得代码更加灵活和富有扩展性。
当某个类除了它的外部类，不再被其他的类使用时。

<h3 id="局部内部类和匿名内部类访问局部变量的时候-为什么变量必须要加上final-">局内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上 final？</h3>

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上 final 呢？它内原理是什么呢？先看这段代码：
<pre><code class="highlight-chroma"> public class Outer {

      void outMethod(){
          final int a = 0;
          class Inner {
              void innerMethod(){
                  System.out.println(a);
              }

```
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
</li>
<li><strong>以上例子，为什么要加 final 呢？是因为</strong><strong><strong>生命周期不致</strong></strong>**， 局部变量直接存储在栈中，当方法执行结束后，非 final 的局部变量就销毁。而局部内部类对局部变量的引用依然存在，如果局部内部类要调用局部变量时，就会出错。加了final，可以确保局部内部类使用的变量与外层的局部变量区分开，解决了这个问题。**</li>
</ul>
<h3 id="内部类相关-看程序说出运行结果"><strong>内部类相关，看程序说出运行结果</strong></h3>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public class Outer {
</span></span><span class="highlight-line"><span class="highlight-cl">    private int age = 12;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    class Inner {
</span></span><span class="highlight-line"><span class="highlight-cl">        private int ag = 13;
</span></span><span class="highlight-line"><span class="highlight-cl">        public void pr
nt() {
</span></span><span class="highlight-line"><span class="highlight-cl">            int age =
4;
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("局部变量：" + age);
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("内部类变量：" + this.age);
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("外部类变量：" + Outer.this.age);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">        Outer.Inner i = new Outer().new Inner();
</span></span><span class="highlight-line"><span class="highlight-cl">        in.print();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>运行结果：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">局部变量：14
</span></span><span class="highlight-line"><span class="highlight-cl">内部类变量：13
</span></span><span class="highlight-line"><span class="highlight-cl">外部类变量：12
</span></span></code></pre>
<h3 id="成员变量与局部变量的区别有哪些"><strong>成员变量与局部变量的区别有哪些</strong></h3>
<ul>
<li><strong>变量：在程序执行的过程中，在某个范围内其值可以发生改变的量。从本质上讲，变其实是内存中的一小块区域</strong></li>
<li><strong>成员变量：方法外部，类内部定义的变量</strong></li>
```

局部变量：类的方法中的变量。
成员变量和局部变量的区别

作用域

成员变量：针对整个类有效。
局部变量：只在某个范围内有效。(一般指的就是方法,语句体内)

存储位置

成员变量：随着对象的创建而存在，随着对象的消失而消失，存储在堆内存中。
局部变量：在方法被调用，或者语句被执行的时候存在，存储在栈内存中。当方法调完，或者语句结束后，就自动释放。

生命周期

成员变量：随着对象的创建而存在，随着对象的消失而消失
局部变量：当方法调用完，或者语句结束后，就自动释放。

初始值

成员变量：有默认初始值。
局部变量：没有默认初始值，使用前必须赋值。

<h3 id="在Java中定义一个不做事且没有参数的构造方法的作用">在 Java 中定义一个不事且没有参数的构造方法的作用</h3>

Java 程序在执行子类的构造方法之前，如果没有用 super()来调用父类特定的构造方，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子的构造方法中又没有用 super()来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造法。

<h3 id="在调用子类构造方法之前会先调用父类没有参数的构造方法-其目的是-">在调用类构造方法之前会先调用父类没有参数的构造方法，其目的是？</h3>

帮助子类做初始化工作。

<h3 id="一个类的构造方法的作用是什么-若一个类没有声明构造方法-改程序能正确执行吗-为什么-">一个类的构造方法的作用是什么？若一个类没有声明构造方法，改程序能正确执行吗？为么？</h3>

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方也会有默认的不带参数的构造方法。

<h3 id="构造方法有哪些特性-">构造方法有哪些特性？</h3>

名字与类名相同；
没有返回值，但不能用 void 声明构造函数；
生成类的对象时自动执行，无需调用。

<h3 id="静态变量和实例变量区别">静态变量和实例变量区别</h3>

静态变量： 静态变量由于不属于任何实例对象，属于类的，所以在内存中只会有一份在类的加载过程中，JVM 只为静态变量分配一次内存空间。
实例变量： 每次创建对象，都会为每个对象分配成员变量内存空间，实例变量是属于例对象的，在内存中，创建几次对象，就有几份成员变量。

<h3 id="静态变量与普通变量区别">静态变量与普通变量区别</h3>

static 变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。
还有一点就是 static 成员变量的初始化顺序按照定义的顺序进行初始化。</i>

<h3 id="静态方法和实例方法有何不同-">静态方法和实例方法有何不同？</h>
<code>静态方法和实例方法的区别主要体现在两个方面：</code>

在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法）而不允许访问实例成员变量和实例方法；实例方法则无此限制

<h3 id="在一个静态方法内调用一个非静态成员为什么是非法的-">在一个静态方法内调一个非静态成员为什么是非法的？</h3>

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变，也不可以访问非静态变量成员。

<h3 id="什么是方法的返回值-返回值的作用是什么-">什么是方法的返回值？返回值的作是什么？</h3>

方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果！（前提是该法可能产生结果）。返回值的作用:接收出结果，使得它可以用于其他的操作！

<h3 id="什么是内部类--">什么是内部类？</h3>

在 Java 中，可以将一个类的定义放在另外一个类的定义内部，这就是<strng>内部类**。内部类本身就是类的一个属性，与其他属性定义方一致。**

<h3 id="内部类的分类有哪些-">内部类的分类有哪些</h3>
<code>内部类可以分为四种：**成员内部类、局部内部类、匿名内部类和静态内部类**。</cod>
<h5 id="静态内部类-">静态内部类</h5>

定义在类内部的静态类，就是静态内部类。
<pre><code class="highlight-chroma"> public class Outer {

      private static i t radius = 1;

      static class Stat clnner {

```
</span></span><span class="highlight-line"><span class="highlight-cl">        public void
isit() {
</span></span><span class="highlight-line"><span class="highlight-cl">            System.o
t.println("visit outer static  variable:" + radius);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
</li>
<li><strong>静态内部类可以访问外部类所有的静态变量，而不可访问外部类的非静态变量；静态
部类的创建方式，</strong><code>new 外部类.静态内部类()</code><strong>，如下：</stron
>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  Outer.StaticInner inner = new Outer.StaticInner();
</span></span><span class="highlight-line"><span class="highlight-cl">  inner.visit();
</span></span></code></pre>
</li>
</ul>
<h5 id="成员内部类-"><strong>成员内部类</strong></h5>
<ul>
<li><strong>定义在类内部，成员位置上的非静态类，就是成员内部类。</strong>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  public class Outer {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private static
nt radius = 1;
</span></span><span class="highlight-line"><span class="highlight-cl">    private int co
nt =2;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    class Inner {
</span></span><span class="highlight-line"><span class="highlight-cl">        public void
isit() {
</span></span><span class="highlight-line"><span class="highlight-cl">            System.o
t.println("visit outer static  variable:" + radius);
</span></span><span class="highlight-line"><span class="highlight-cl">            System.o
t.println("visit outer   variable:" + count);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
</li>
<li><strong>成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公有。成
内部类依赖于外部类的实例，它的创建方式</strong> <code>外部类实例.new 内部类()</code><s
rong>，如下：</strong>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  Outer outer = new Outer();
</span></span><span class="highlight-line"><span class="highlight-cl">  Outer.Inner inner
= outer.new Inner();
</span></span><span class="highlight-line"><span class="highlight-cl">  inner.visit();
</span></span></code></pre>
</li>
</ul>
<h5 id="局部内部类-"><strong>局部内部类</strong></h5>
<ul>
```

定义在方法中的内部类，就是局部内部类。
<pre><code class="highlight-chroma"> public class Outer {

      private int ou
_a = 1;
      private static i
t STATIC_b = 2;

      public void tes
FunctionClass(){
          int inner_c
3;
          class Inner {
              private vo
d fun(){
                  System
out.println(out_a);
                  System
out.println(STATIC_b);
                  System
out.println(inner_c);
              }
          }
          Inner inner
= new Inner();
          inner.fun();
      }
      public static vo
d testStaticFunctionClass(){
          int d =3;
          class Inner {
              private vo
d fun(){
                  // Syst
m.out.println(out_a); 编译错误，定义在静态方法中的局部类不可以访问外部类的实例变量
System.out.println
STATIC_b);
                  System
out.println(d);
              }
          }
          Inner inner
= new Inner();
          inner.fun();
      }
  }
</code></pre>

定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的
部类只能访问外部类的静态变量和方法。局部内部类的创建方式，在对应方法内，<code
new 内部类()</code>，如下：
<pre><code class="highlight-chroma"> public static void testStaticFunctionClass(){

```
</span></span><span class="highlight-line"><span class="highlight-cl">      class Inner {
</span></span><span class="highlight-line"><span class="highlight-cl">      }
</span></span><span class="highlight-line"><span class="highlight-cl">      Inner  inner = 
ew Inner();
</span></span><span class="highlight-line"><span class="highlight-cl">   }
</span></span></code></pre>
</li>
</ul>
<h5 id="匿名内部类-"><strong>匿名内部类</strong></h5>
<ul>
<li><strong>匿名内部类就是没有名字的内部类，日常开发中使用的比较多。</strong>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  public class Outer {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">      private void te
t(final int i) {
</span></span><span class="highlight-line"><span class="highlight-cl">          new Service(
 {
</span></span><span class="highlight-line"><span class="highlight-cl">              public vo
d method() {
</span></span><span class="highlight-line"><span class="highlight-cl">                  for (int
 = 0; j &amp;lt; i; j++) {
</span></span><span class="highlight-line"><span class="highlight-cl">                      Syst
m.out.println("匿名内部类" );
</span></span><span class="highlight-line"><span class="highlight-cl">                  }
</span></span><span class="highlight-line"><span class="highlight-cl">              }
</span></span><span class="highlight-line"><span class="highlight-cl">          }.method();
</span></span><span class="highlight-line"><span class="highlight-cl">      }
</span></span><span class="highlight-line"><span class="highlight-cl">  }
</span></span><span class="highlight-line"><span class="highlight-cl">  //匿名内部类必
继承或实现一个已有的接口
</span></span><span class="highlight-line"><span class="highlight-cl">interface Service{
</span></span><span class="highlight-line"><span class="highlight-cl">      void method();
</span></span><span class="highlight-line"><span class="highlight-cl">  }
</span></span></code></pre>
</li>
<li><strong>除了没有名字，匿名内部类还有以下特点：</strong>
<ul>
<li><strong>匿名内部类必须继承一个抽象类或者实现一个接口。</strong></li>
<li><strong>匿名内部类不能定义任何静态成员和静态方法。</strong></li>
<li><strong>当所在的方法的形参需要被匿名内部类使用时，必须声明为 final。</strong></li>
<li><strong>匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。</
trong></li>
</ul>
</li>
<li><strong>匿名内部类创建方式：</strong>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  new 类/接口{
</span></span><span class="highlight-line"><span class="highlight-cl">    //匿名内部类实
部分
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
</li>
</ul>
```

<h3 id="内部类的优点-">内部类的优点</h3>
<code>我们为什么要使用内部类呢？因为它有以下优点：</code>

一个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！

内部类不为同一包的其他类所见，具有很好的封装性；
内部类有效实现了“多重继承”，优化 java 单继承的缺陷。
匿名内部类可以很方便的定义回调。

<h3 id="内部类有哪些应用场景-">内部类有哪些应用场景</h3>

一些多算法场合
解决一些非面向对象的语句块。
适当使用内部类，使得代码更加灵活和富有扩展性。
当某个类除了它的外部类，不再被其他的类使用时。

<h3 id="局部内部类和匿名内部类访问局部变量的时候-为什么变量必须要加上final--">
部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上 final？</h3>

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上 final 呢？它内
原理是什么呢？先看这段代码：
<pre><code class="highlight-chroma"><span class="highlight
cl"> public class Outer {

      void outMeth
d(){
          final int a =
0;
          class Inner {
              void inne
Method(){
                  System
out.println(a);
              }
          }
      }
  }
</code></pre>

以上例子，为什么要加 final 呢？是因为生命周期不
致**，局部变量直接存储在栈中，当方法执行结束后，非 final 的局部变量就
销毁。而局部内部类对局部变量的引用依然存在，如果局部内部类要调用局部变量时，就会出错。加了
final，可以确保局部内部类使用的变量与外层的局部变量区分开，解决了这个问题。**

<h3 id="内部类相关-看程序说出运行结果-">内部类相关，看程序说出运行结果
</h3>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class Outer {
    private int age
 12;

    class Inner {
        private int ag
 = 13;

```
</span></span><span class="highlight-line"><span class="highlight-cl">        public void pr
nt() {
</span></span><span class="highlight-line"><span class="highlight-cl">            int age =
4;
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("局部变量：" + age);
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("内部类变量：" + this.age);
</span></span><span class="highlight-line"><span class="highlight-cl">            System.out
println("外部类变量：" + Outer.this.age);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args) {
</span></span><span class="highlight-line"><span class="highlight-cl">        Outer.Inner i
 = new Outer().new Inner();
</span></span><span class="highlight-line"><span class="highlight-cl">        in.print();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>运行结果：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">局部变量：14
</span></span><span class="highlight-line"><span class="highlight-cl">内部类变量：13
</span></span><span class="highlight-line"><span class="highlight-cl">外部类变量：12
</span></span></code></pre>
<h2 id="文件-I-O流"><strong>文件、I/O 流</strong></h2>
<h3 id="Java-中有几种类型的流-"><strong>Java 中有几种类型的流？</strong></h3>
<p><strong>从输入输出方面来讲：  Java 中有输入流和输出流</strong></p>
<p><strong>从流的编码方式上来讲：  Java 中有字节流和字符流</strong></p>
<p><strong>对于字节流而言:主要继承的抽象类为 InputStream 和 OutputStream</strong></p

<p><strong>对于字符流而言:主要继承的抽象类为 InputStreamReader 和 OutputStreamReder</
trong></p>
<h3 id="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-"><st
ong>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。</stro
g></h3>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">  public int countWords(String file, String find) throws Exception {
</span></span><span class="highlight-line"><span class="highlight-cl">        int count = 0;
</span></span><span class="highlight-line"><span class="highlight-cl">        Reader in =
ew FileReader(file);
</span></span><span class="highlight-line"><span class="highlight-cl">        int c;
</span></span><span class="highlight-line"><span class="highlight-cl">        while ((c = in.
ead()) != -1) {
</span></span><span class="highlight-line"><span class="highlight-cl">            while (c ==
find.charAt(0)) {
</span></span><span class="highlight-line"><span class="highlight-cl">                for (int i
= 1; i &amp;lt; find.length(); i++) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    c = in.
ead();
</span></span><span class="highlight-line"><span class="highlight-cl">                    if (c !
```

```
 find.charAt(i)){
                    bre
k;
                }
                if (i =
 find.length() - 1){
                    co
nt++;
                }
            }
        }
    }
    return count;
}
```

Java 中怎么创建 ByteBuffer？

```
byte[] bytes = new byte[10];

ByteBuffer buf =
yteBuffer.wrap(bytes);
```

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：

1、 ** 使用有缓冲区的 IO 类，而不要单独读取字节字符。**

2、 ** 使用 NIO 和 NIO2**

3、 ** 在 finally 块中关闭流，或者使用 try-with-resource 语句。**

4、 ** 使用内存映射文件获取更快的 IO。**

Java 中 IO 流分为几种？

- **按照流的流向分，可以分为输入流和输出流；**
- **按照操作单元划分，可以划分为字节流和字符流；**
- **按照流的角色划分为节点流和处理流。**

Java Io流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存非常紧密的联系， Java I0流的40多个类都是从如下4个抽象类基类中派生出来的。

- **InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流**
- **OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。**

按操作方式分类结构图：

按操作对象分类结构图：

谈谈 Java IO 里面的常见类，字节流，字符流、接口、实现类、方法阻塞

输入流就是从外部文件输入到内存，输出流主要是从内存输出到文件。

**IO 里面常见的类，第一印象就只知道 IO 流中有很多类，IO 流主要分为字符流和字节流

字符流中有抽象类 InputStream 和 OutputStream，它们的子类 FileInputStream，FileOutputStream,BufferedOutputStream 等。
字符流 BufferedReader 和 Writer 等。都实现了 Closeable, Flushable, Appendable 这些接口。程序中的输入输出都是以流的形式保存的，流中保存的实际上全都是字节文件。**
**java 中的阻塞式方法是指在程序调用改方法时，必须等待输入数据可用或者检测到输入束或者抛出异常，否则程序会一直停留在该语句上，不会执行下面的语句。比如 read()和 readLine()法。
<h3 id="字符流和字节流有什么区别-">字符流和字节流有什么区别？</h3>
字节流用于操作包含 ASCII 字符的文件。JAVA 也支持其他的字符如 Unicode，为了取包含 Unicode 字符的文件，JAVA 语言引入了字符流。ASCII 作为 Unicode 的子集，对于英语字的文件，可以使用字节流也可以使用字符流。
<h3 id="字节流和字符流-你更喜欢哪一个-">字节流和字符流，你更喜欢哪一个？</h3>
更喜欢使用字符流。许多在字符流中存在的特性，字节流中不存在。比如使用 BufferedReader 而不是 BufferedInputStream 或 DataInputStream，它其中包含一个 <code>readLine()</code>** 方法用于读取文本行；又比如 BufferedWriter 流中有一个独特的向文件写入一换行符的方法‘newLine()’用来读取下一行，但是在字节流中我们需要做额外的操作。**
<h3 id="System-out-println--是什么-">System.out.println()是什么？</h3>
println 是 PrintStream 的一个方法。out 是一个静态 PrintStream 类型的成员变量，ystem 是一个 java.lang 包中的类，用于和底层的操作系统进行交互。
<h3 id="什么是Filter流-">什么是 Filter 流？</h3>
1、** Filter Stream 是一种 IO 流。****
2、 Filter 流的主要作用是：对存在的流增加一些额外的功能，像给目标件增加源文件中不存在的行数，或者增加拷贝的性能。**
Filter Stream 是一种 IO 流。Filter 流的主要作用是：对存在的流增加一些额外的功，像给目标文件增加源文件中不存在的行数，或者增加拷贝的性能。在 java.io 包中主要由 4 个可用的filter Stream 组成。两个字节 filter stream，两个字符 filter stream。分别是：FilterInputStream、ilterOutputStream、FilterReader 和 FilterWriter。
<h3 id="有哪些可用的Filter流-">有哪些可用的 Filter 流？</h3>
在 java.io 包中主要由 4 个可用的 filter Stream 组成。两个字节 filter stream，两个符 filter stream。分别是：<code>FilterInputStream</code>、<ode>FilterOutputStream</code>、<code>FilterReader</code> 和 <code>FilterWriter</code>。
<h3 id="有哪些Filter流的子类-">有哪些 Filter 流的子类？</h3>
1、** <code>LineNumberInputStrea</code>：给目标文件增加行号。****
2、 <code>DataInputStream</code>：有些特殊的方法如：readInt()、readDouble()和 readLine()等可以一次的读取一个 int, double 和一个 string 类型的数据。****
3、 <code>BufferedInptStream</code>：增加性能。****
4、 <code>PushbackInutStream</code>：推送要求的字节到系统中。**
<h3 id="在文件拷贝的时候-哪一种流可用于提升更多的性能-">在文件拷贝的时候，哪一种流可用于提升更多的性能？</h3>
1、** 在字节流的时候，使用 BufferedInputStream 和 BufferedOutputStream。****
2、 在字符流的时候，使用 BufferedReader 和 BufferedWriter。**
<h3 id="Java中流类的超类---均为抽象类-主要由哪些组成---">Java 中流类的</h3>

strong>超类**（均为抽象类）主要由哪些组成？**</h3>
1、** <code>java.io.InputStream</code>（字节输入流）****
2、 <code>java.io.OutputStream</code>（字节输出流）****
3、 <code>java.io.Reader</code>（字符输入流）****
4、 <code>java.io.Writer</code>（字符输出流）**
<h3 id="FileInputStream和FileOutputStream是什么-">FileInputStream 和 FileOutputtream 是什么？</h3>
1、** 这是在拷贝文件操作的时候，经常用到的两个。****
2、 在处理小文件的时候，它们的能表现还不错，在大文件的时候，最好使用**<code>BufferedInputStream</code>（或 <code>BufferedReader</code>）和 <code>BufferedOutputStream</code>（或 <code>BufferedWriter</code>）
<h3 id="BIO-NIO-AIO-有什么区别-">BIO、NIO、AIO 有什么区别？</h3>
关于 Java 中的 IO，网络通讯模型主要分三种：IO、NIO 和 AIO。

适用场景：
BIO，** 适用于连接较少，对服务器资源消耗很大，是编程简单。是同步阻塞的。****
举例：你到餐馆点餐，然后在那儿等着，什么也做不了，只要饭还没有好，就要须等着
NIO， 使用于连接数量比较多且连时间比较短的架构，比如聊天服务器，编程比较复杂。是同步非阻塞的****
举例：你到餐馆点完餐，然后就可以去玩儿了，玩一会儿就回饭馆问一声，饭好没。
AIO， 适用于连接数量多而且连接间长的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂。是异步非阻塞的。****
**举例：饭馆打电话给你说，我们知道你的位置，待会儿给您送来，你安心的玩儿就可以。类似于外卖。
1、IO（同步阻塞）
传统的网络通讯模型，就是 BIO，同步阻塞 IO， 其实就是服务端创建一个 ServerSocket， 然后就是客户端用一个 Socket 去连接服务端的那个 ServerSocket， ServerSocket 接收到了一的连接请求就创建一个 Socket 和一个线程去跟那个 Socket 进行通讯。接着客户端和服务端就进行塞式的通信，客户端发送一个请求，服务端 Socket 进行处理后返回响应，在响应返回前，客户端那就阻塞等待，上门事情也做不了。 这种方式的缺点， 每次一个客户端接入，都需要在服务端创建一线程来服务这个客户端，这样大量客户端来的时候，就会造成服务端的线程数量可能达到了几千甚至万，这样就可能会造成服务端过载过高，最后崩溃死掉。
2、NIO（同步非阻塞，JDK1.4）
NIO 是一种同步非阻塞 IO, 基于 Reactor 模型来实现的。其实相当于就是一个线程处大量的客户端的请求，通过一个线程轮询大量的 channel，每次就获取一批有事件的 channel，然后每个请求启动一个线程处理即可。这里的核心就是非阻塞，就那个 selector 一个线程就可以不停轮询hannel，所有客户端请求都不会阻塞，直接就会进来，大不了就是等待一下排着队而已。这里面优化IO 的核心就是，一个客户端并不是时时刻刻都有数据进行交互，没有必要死耗着一个线程不放，所客户端选择了让线程歇一歇，只有客户端有相应的操作的时候才发起通知，创建一个线程来处理请求

3、AIO（NIO2，异步非阻塞，JDK1.7）
对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线自行进行 IO 操作，IO 操作本身是同步的。但是对 AIO 来说，则更加进了一步，它不是在 IO 准备好再通知线程，而是在 IO 操作已经完成后，再给线程发出通知。因此 AIO 是不会阻塞的，此时我们的务逻辑将变成一个回调函数，等待 IO 操作完成后，由系统自动触发。
与 NIO 不同，当进行读写操作时，只须直接调用 API 的 read 或 write 方法即可。这种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入 read 方法的缓区，并通知应用程序；对于写操作而言，当操作系统将 write 方法传递的流写入完毕时，操作系统主通知应用程序。 即可以理解为，read/write 方法都是异步的，完成后会主动调用回调函数。 在 JDK1. 中，这部分内容被称作 NIO.2，主要在 Java.nio.channels 包下增加了下面四个异步通道：</strong

AsynchronousSocketChannel
AsynchronousServerSocketChannel
AsynchronousFileChannel
AsynchronousDatagramChannel

** AIO 是异步非阻塞 IO，基于 Proactor 模型实现。 每个连接发送过来的请求，都会绑定一个uffer，然后通知操作系统去完成异步的读，这个时间你就可以去做其他的事情，等到操作系统完成读后，就会调用你的接口，给你操作系统异步读完的数据。这个时候你就可以拿到数据进行处理，将数往回写，在往回写的过程，同样是给操作系统一个 Buffer，让操作系统去完成写，写完了来通知你。俩个过程都有 buffer 存在，数据都是通过 buffer 来完成读写。,**
<h3 id="讲讲NIO">讲讲 NIO</h3>
看了一些文章，传统的 IO 流是阻塞式的，会一直监听一个 ServerSocket，在调用 rea 等方法时，他会一直等到数据到来或者缓冲区已满时才返回。
调用 accept 也是一直阻塞到有客户端连接才会返回。每个客户端连接过来后，服务都会启动一个线程去处理该客户端的请求。并且多线程处理多个连接。每个线程拥有自己的栈空间并占用一些 CPU 时间。每个线程遇到外部未准备好的时候，都会阻塞掉。阻塞的结果就是会带来大量进程上下文切换。**
对于 NIO，它是非阻塞式，核心类：
1、 Buffer 为所有的原始类型提供 (uffer)缓存支持。****
2、 Charset 字符集编码解码解决案****
3、 Channel 一个新的原始 I/O 抽，用于读写 Buffer 类型，通道可以认为是一种连接，可以是到特定设备，程序或者是网络的连接。**
/p>
<h3 id="Files的常用方法都有哪些-">Files 的常用方法都有哪些？</h3>

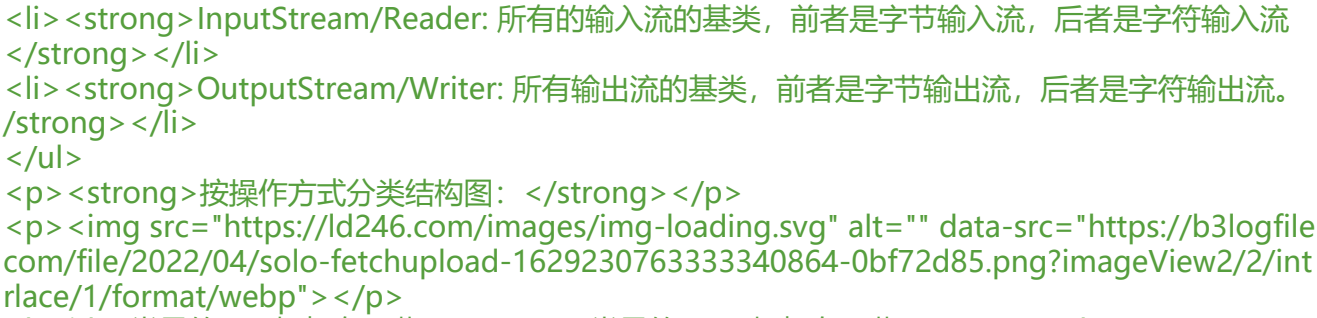
Files. exists()：检测文件路径是否存在。
Files. createFile()：创建文件。
Files. createDirectory()：创建文件夹。
Files. delete()：删除一个文件或目录。
Files. copy()：复制文件。
Files. move()：移动文件。
Files. size()：查看文件个数。
Files. read()：读取文件。
Files. write()：写入文件。

<h3 id="Java-中-IO-流分为几种--">Java 中 IO 流分为几种?</h3>

按照流的流向分，可以分为输入流和输出流；
按照操作单元划分，可以划分为字节流和字符流；
按照流的角色划分为节点流和处理流。

Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之存在非常紧密的联系， Java I0 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流
OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。/strong>

按操作方式分类结构图：

<h3 id="常见的NIO框架有哪些">常见的 NIO 框架有哪些</h3>
1、Mina** **
Mina 是 Apache 组织一个较新的项目，它为开发高性能和高可用性的网络应用程序供了非常便利的框架。当前发行的 Mina 版本 2.04 支持基于 JavaNIO 技术的 TCP/UDP 应用程序开、串口通讯程序，Mina 所支持的功能也在进一步的扩展中。
2、Netty** **
**Netty 是一款异步的事件驱动的网络应用框架和工具，用于快速开发可维护的高性能、高扩展协议服务器和客户端。也就是说，Netty 是一个 NIO 客户端/服务器框架，支持快速、简单地开发网应用，如协议服务器和客户端。它极大简化了网络编程，如 TCP 和 UDP 套接字服务器。 **
3、Grizzly** **
**Grizzly 是一种应用程序框架，专门解决编写成千上万用户访问服务器时候产生的各种问题。用 JAVANIO 作为基础，并隐藏其编程的复杂性。容易使用的高性能的 API。带来非阻塞 socketd 到议处理层。利用高性能的缓冲和缓冲管理使用高性能的线程池。 **
<h3 id="Java-IO-中的设计模式--重点-">Java IO 中的设计模式？（重点）</3>
重点是用到两个设计模式：装饰模式和适配器模式。
装饰模式：** 在由 InputStream、OutputStream、eader 和 Writer 代表的等级结构内部，有一些流处理器可以对另一些流处理器起到装饰作用，形成的、具有改善了的功能的流处理器。****
适配器模式： 在由 InputStream、utputStream、Reader 和 Writer 代表的等级结构内部，有一些流处理器是对其他类型的流处理器的配，这就是适配器的应用。**
<h3 id="在文件拷贝的时候-哪一种流可用于提升更多的性能--">在文件拷贝的时候，哪种流可用于提升更多的性能？</h3>

在字节流的时候，使用 BufferedInputStream 和 BufferedOutputStream。
在字符流的时候，使用 BufferedReader 和 BufferedWriter。

<h3 id="说说管道流-Piped-Stream-">说说管道流（Piped Stream）</h3>

有四种管道流：<code>PipedInputStream</code>、<code>PipedOutputStream</code>、<code>PipedReader</code> 和 <code>PipedWriter</code>。
在多个线程或进程中传递数据的时候管道流非常有用。

<h3 id="说说File类">说说 File 类</h3>

它不属于 IO 流，也不是用于文件操作的。
它主要是用于获取一个文件的属性、读写权限、大小等信息

```
</ul>
<h2 id="反射"><strong>反射</strong></h2>
<h3 id="什么是反射机制-"><strong>什么是反射机制？</strong></h3>
<ul>
<li><strong>JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和
法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对
的方法的功能称为 java 语言的反射机制。</strong></li>
<li><strong>静态编译和动态编译</strong>
<ul>
<li><strong>静态编译：在编译时确定类型，绑定对象</strong></li>
<li><strong>动态编译：运行时确定类型，绑定对象</strong></li>
</ul>
</li>
</ul>
<h3 id="反射机制优缺点"><strong>反射机制优缺点</strong></h3>
<ul>
<li><strong><strong>优点：</strong></strong>** 运行期类型的判断，动态加载类，提高代
灵活度。**</li>
<li><strong><strong>缺点：</strong></strong>** 性能瓶颈：反射相当于一系列解释操作，
知 JVM 要做的事情，性能比直接的 java 代码要慢很多。**</li>
</ul>
<h3 id="反射机制的应用场景有哪些-"><strong>反射机制的应用场景有哪些？</strong></h3>
<ul>
<li><strong>反射是框架设计的灵魂。</strong></li>
<li><strong>在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反
机制没有用，实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调用对应
字节码；动态代理设计模式也采用了反射机制，还有我们日常使用的 Spring / Hibernate 等框架也
量使用到了反射机制。</strong></li>
<li><strong>举例：① 我们在使用 JDBC 连接数据库时使用 Class.forName()通过反射加载数据库
驱动程序；②Spring 框架也用到很多反射机制，最经典的就是 xml 的配置模式。Spring 通过 XML
置模式装载 Bean 的过程：1) 将程序内所有 XML 或 Properties 配置文件加载入内存中; 2)Java 类里
解析 xml 或 properties 里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)使用反
机制，根据这个字符串获得某个类的 Class 实例; 4)动态配置实例的属性</strong></li>
</ul>
<h3 id="Java获取反射的三种方法"><strong>Java 获取反射的三种方法</strong></h3>
<p><strong>1.通过 new 对象实现反射机制 2.通过路径实现反射机制 3.通过类名实现反射机制</str
ng></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class Student {
</span></span><span class="highlight-line"><span class="highlight-cl">    private int id;
</span></span><span class="highlight-line"><span class="highlight-cl">    String name;
</span></span><span class="highlight-line"><span class="highlight-cl">    protected bool
an sex;
</span></span><span class="highlight-line"><span class="highlight-cl">    public float sco
e;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">public class Get {
</span></span><span class="highlight-line"><span class="highlight-cl">    //获取反射机制
种方式
</span></span><span class="highlight-line"><span class="highlight-cl">  public static void
main(String[] args) throws ClassNotFoundException {
</span></span><span class="highlight-line"><span class="highlight-cl">        //方式一(通
建立对象)
```

        Student stu = new Student();
        Class classob1 = stu.getClass();
        System.out.println(classobj1.getName());
        //方式二（所通过路径-相对路径）
    Class classobj2 = Class.forName("fanshe.Student");
        System.out.println(classobj2.getName());
        //方式三（通类名）
    Class classobj3 = Student.class;
        System.out.println(classobj3.getName());
    }
}
</code></pre>
<h2 id="字符串">字符串</h2>
<h3 id="字符型常量和字符串常量的区别">字符型常量和字符串常量的区别</3>

形式上: 字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符
含义上: 字符常量相当于一个整形值(ASCII 值),可以参加表达式运算 字符串常量代表一地址值(该字符串在内存中存放位置)
占内存大小 字符常量只占一个字节 字符串常量占若干个字节(至少一个字符结束标志)

<h3 id="什么是字符串常量池-">什么是字符串常量池？</h3>

字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避开辟多块空间存储相同的字符串，在创建字符串时 JVM 会首先检查字符串常量池，如果该字符串已存在池中，则返回它的引用，如果不存在，则实例化一个字符串放到池中，并返回其引用。

<h3 id="String-是最基本的数据类型吗">String 是最基本的数据类型吗</h3>

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（referencetype），Java 以后引入的枚举类型也算是一种比较特殊的引用类型。

<code>这是很基础的东西，但是很多初学者却容易忽视，Java 的 8 种基本数据类型中不包括 String，基本数据类型中用来描述文本数据的是 char，但是它只能表示单个字符，比如 ‘a’，‘好’ 之的，如果要描述一段文本，就需要用多个 char 类型的变量，也就是一个 char 类型数组，比如 “你好 就是长度为2的数组 char\[\] chars = { ‘你’，‘好’ };</code>
<code>但是使用数组过于麻烦，所以就有了 String，String 底层就是一个 char 类型的数组，是使用的时候开发者不需要直接操作底层数组，用更加简便的方式即可完成对字符串的使用。</code>
<h3 id="String有哪些特性">String 有哪些特性</h3>

不变性：String 是只读字符串，是一个典型的 immutable 对象，对它进行任何操作其实都是创建一个新的对象，再把引用指向该对象。不变模式的主要作用在于当一个对象需要被多线共享并频繁访问时，可以保证数据的一致性。
常量池优化：String 对象创建之后，会在字符串常量池中进行缓存，如果下次创建同的对象时，会直接返回缓存的引用。
final：使用 final 来定义 String 类，表示 String 类不能被继承，提高了系统的安全性

<h3 id="String为什么是不可变的吗-">String 为什么是不可变的吗？</h3>

简单来说就是 String 类利用了 final 修饰的 char 类型数组存储字符，源码如下图所以
/ The value is used for character storage. / private final char value[]*

<h3 id="String真的是不可变的吗-">String 真的是不可变的吗？</h3>

我觉得如果别人问这个问题的话，回答不可变就可以了。 下面只是给大家看两个有代性的例子：

1 String 不可变但不代表引用不可以变
<pre><code class="highlight-chroma">String str = "Hello";
str = str + " World ;
System.out.println "str=" + str);
</code></pre>

结果：
str=Hello World
解析：
实际上，原来 String 的内容是不变的，只是 str 由原来指向"Hello"的内存地址转为指"Hello World"的内存地址而已，也就是说多开辟了一块内存区域给"Hello World"字符串。

2.通过反射是可以修改所谓的“不可变”对象
<pre><code class="highlight-chroma">// 创建字符串"Hello World"， 并赋给引用s
String s = "Hello orld";

System.out.println "s = " + s); // Hello World

// 获取String类中value字段
Field valueFieldOf tring = String.class.getDeclaredField("value");

// 改变value属性访问权限
valueFieldOfString setAccessible(true);

</code></pre>

// 获取s对象上的value属性的值
char[] value = (char[]) valueFieldOfString.get(s);

// 改变value所引的数组中的第5个字符
value[5] = '_';

System.out.println "s = " + s); // Hello_World
</code></pre>

结果：
s = Hello World s = Hello_World
解析：
用反射可以访问私有成员， 然后反射出 String 对象中的 value 属性， 进而改变通过得的 value 引用改变数组的结构。但是一般我们不会这么做，这里只是简单提一下有这个东西。

<h3 id="是否可以继承-String-类">是否可以继承 String 类</h3>

String 类是 final 类，不可以被继承。

<h3 id="String-str--i-与-String-str-new-String--i--一样吗-">String str="i"与 String str=new String(“i”)一样吗？</h3>

不一样，因为内存的分配方式不一样。String str="i"的方式，java 虚拟机会将其分配常量池中；而 String str=new String(“i”) 则会被分到堆内存中。

<h3 id="String-s---new-String--xyz---创建了几个字符串对象">String s = new String(“xyz”);创建了几个字符串对象</h3>
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。

<h3 id="如何将字符串反转-">如何将字符串反转？</h3>
使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。
<h3 id="数组有没有-length--方法-String-有没有-length--方法">数组有没有 length()方法？String 有没有 length()方法</h3>

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。

<h3 id="String-类的常用方法都有那些-">String 类的常用方法都有那些？</h3>

indexOf()：返回指定字符的索引。
charAt()：返回指定索引处的字符。
replace()：字符串替换。
trim()：去除字符串两端空白。
split()：分割字符串，返回一个分割后的字符串数组。
getBytes()：返回字符串的 byte 类型数组。
length()：返回字符串长度。
toLowerCase()：将字符串转成小写字母。

toUpperCase()：将字符串转成大写字符。
substring()：截取字符串。
equals()：字符串比较。

<h3 id="在使用-HashMap-的时候-用-String-做-key-有什么好处-">在使用 HashMap 时候，用 String 做 key 有什么好处？</h3>

HashMap 内部实现是通过 key 的 hashcode 来确定 value 的存储位置，因为字符串不可变的，所以当创建字符串时，它的 hashcode 被缓存下来，不需要再次计算，所以相比于其他对更快。

<h3 id="String和StringBuffer-StringBuilder的区别是什么-String为什么是不可变的">Strng 和 StringBuffer、StringBuilder 的区别是什么？String 为什么是不可变的</h3>
可变性

String 类中使用字符数组保存字符串，private final char value[]，所以 string 对象不可变的。StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串，char[] value，这两种对象都是可变的。

线程安全性

String 中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity append、insert、indexOf 等公共方法。StringBuffer 对方法加了同步锁或者对调用的方法加了同锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象改变对象引用。相同情况下使用 StirngBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的能提升，但却要冒多线程不安全的风险。

对于三者使用的总结

如果要操作少量的数据用 = String
单线程操作字符串缓冲区 下操作大量数据 = StringBuilder
多线程操作字符串缓冲区 下操作大量数据 = StringBuffer
