

# 必知

## 关于本小册

两个星期前，我和我的好朋友决定做一系列的 Java 知识点常见重要问题的小册。小册的标准就一个，那就是：**取精华，取重点**。每一本小册，我们都会充分关注我们所总结的知识点是否达到这个标准。

推荐和我开源的 [《JavaGuide面试突击版》](https://snailclimb.gitee.io/javaguide-interview) 配套阅读，两者在很多知识点上内容都相同，不过小册的知识点涵盖更广一点。《JavaGuide面试突击版》在线阅读版本地址：<https://snailclimb.gitee.io/javaguide-interview>。

## 是否免费？

小册系列完全免费开源！地址：<https://github.com/Snailclimb/JavaGuide>。

## 小册系列会涵盖哪些知识点？

目前还不是特别确定，大概会涵盖如下一些知识点

1. Java：基础、集合、并发、JVM
2. 计算机基础：计算机网络、操作系统
3. 数据库：MySQL、Redis
4. 框架：Spring、Netty.....
5. 分布式

## 如何贡献

大家阅读过程中如果遇到错误的地方可以直接在 [JavaGuide](#) 提交 issue 或者 pr, 最欢迎 pr 的形式 ~ ~ ~

当然你也可以通过邮箱：[koushuangbwcx@163.com](mailto:koushuangbwcx@163.com) 与我交流。

希望大家给我提反馈的时候可以按照如下格式：

我觉得Java基础部分的1.2.1的内容的描述有问题，应该这样描述：~巴拉巴拉~ 会更好！具体可以参考 Oracle 官方文档，地址：~~~~。

为了提高准确性已经不必要的时间花费，希望大家尽量确保自己想法的准确性。

## 如何赞赏

如果觉得本文档对你有帮助的话，欢迎加入我的知识星球。创建星球的目的主要是为了提高知识沉淀，微信群的弊端相比大家都了解。星球没有免费的原因是设立了门槛，提高进入读者的质量。我会在星球回答大家的问题，更新更多的大厂面试干货！

知识星球



Guide哥

送你一张星球优惠券

¥ 23

立减

可用于

JavaGuide读者圈

2020/05/16 12:00 至 2020/07/16 12:00

前 500 名加入可用 ▶

长按二维码立抢优惠



点击关注[公众号](#)及时获取笔主最新文章，并可免费领取本文档配套的《Java 面试突击》以及 Java 工程师必备学习资源。

- 1. Java 基本功
  - 1.1. Java 入门（基础概念与常识）
    - 1.1.1. Java 语言有哪些特点?
    - 1.1.2. 关于 JVM JDK 和 JRE 最详细通俗的解答
      - 1.1.2.1. JVM
      - 1.1.2.2. JDK 和 JRE
    - 1.1.3. Oracle JDK 和 OpenJDK 的对比
    - 1.1.4. Java 和 C++的区别?
    - 1.1.5. 什么是 Java 程序的主类 应用程序和小程序的主类有何不同?
    - 1.1.6. Java 应用程序与小程序之间有哪些差别?
    - 1.1.7. import java 和 javax 有什么区别?
    - 1.1.8. 为什么说 Java 语言“编译与解释并存”?
  - 1.2. Java 语法
    - 1.2.1. 字符型常量和字符串常量的区别?
    - 1.2.2. 关于注释?
    - 1.2.3. 标识符和关键字的区别是什么?
    - 1.2.4. Java 中有哪些常见的关键字?
    - 1.2.5. 自增自减运算符
    - 1.2.6. continue、break、和return的区别是什么?
    - 1.2.7. Java泛型了解么? 什么是类型擦除? 介绍一下常用的通配符?
    - 1.2.8. ==和equals的区别
    - 1.2.9. hashCode()与 equals()
  - 1.3. 基本数据类型
    - 1.3.1. Java中的几种基本数据类型是什么? 对应的包装类型是什么? 各自占用多少字节呢?
    - 1.3.2. 自动装箱与拆箱
    - 1.3.3. 8种基本类型的包装类和常量池
  - 1.4. 方法（函数）
    - 1.4.1. 什么是方法的返回值?返回值在类的方法里的作用是什么?
    - 1.4.2. 为什么 Java 中只有值传递?
    - 1.4.3. 重载和重写的区别 - 1.4.3.1. 重载 - 1.4.3.2. 重写
    - 1.4.4. 深拷贝 vs 浅拷贝
    - 1.4.5. 方法的四种类型
- 2. Java 面向对象
  - 2.1. 类和对象
    - 2.1.1. 面向对象和面向过程的区别
    - 2.1.2. 构造器 Constructor 是否可被 override?
    - 2.1.3. 在 Java 中定义一个不做事且没有参数的构造方法的作用
    - 2.1.4. 成员变量与局部变量的区别有哪些?

- [2.1.5. 创建一个对象用什么运算符?对象实体与对象引用有何不同?](#)
  - [2.1.6. 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,该程序能正确执行吗?为什么?](#)
  - [2.1.7. 构造方法有哪些特性?](#)
  - [2.1.8. 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?](#)
  - [2.1.9. 对象的相等与指向他们的引用相等,两者有什么不同?](#)
- [2.2. 面向对象三大特征](#)
  - [2.2.1. 封装](#)
  - [2.2.2. 继承](#)
  - [2.2.3. 多态](#)
- [2.3. 修饰符](#)
  - [2.3.1. 在一个静态方法内调用一个非静态成员为什么是非法的?](#)
  - [2.3.2. 静态方法和实例方法有何不同](#)
  - [2.3.3. 常见关键字总结:static,final,this,super](#)
- [2.4. 接口和抽象类](#)
  - [2.4.1. 接口和抽象类的区别是什么?](#)
- [2.5. 其它重要知识点](#)
  - [2.5.1. String StringBuffer 和 StringBuilder 的区别是什么? String 为什么是不可变的?](#)
  - [2.5.2. Object 类的常见方法总结](#)
  - [2.5.3. == 与 equals\(重要\)](#)
  - [2.5.4. hashCode 与 equals \(重要\)](#)
    - [2.5.4.1. hashCode \(\) 介绍](#)
    - [2.5.4.2. 为什么要有 hashCode](#)
    - [2.5.4.3. hashCode \(\) 与 equals \(\) 的相关规定](#)
  - [2.5.5. Java 序列化中如果有些字段不想进行序列化,怎么办?](#)
  - [2.5.6. 获取用键盘输入常用的两种方法](#)
- [3. Java 核心技术](#)
  - [3.1. 集合](#)
    - [3.1.1. Collections 工具类和 Arrays 工具类常见方法总结](#)
  - [3.2. 异常](#)
    - [3.2.1. Java 异常类层次结构图](#)
    - [3.2.2. Throwable 类常用方法](#)
    - [3.2.3. try-catch-finally](#)
    - [3.2.4. 使用 try-with-resources 来代替try-catch-finally](#)
  - [3.3. 多线程](#)
    - [3.3.1. 简述线程、程序、进程的基本概念。以及他们之间关系是什么?](#)
    - [3.3.2. 线程有哪些基本状态?](#)
  - [3.4. 文件与 I\O 流](#)
    - [3.4.1. Java 中 IO 流分为几种?](#)

- [3.4.1.1. 既然有了字节流,为什么还要有字符流?](#)
- [3.4.1.2. BIO,NIO,AIO 有什么区别?](#)
- [4. 参考](#)
- [5. 公众号](#)

# 1. Java 基本功

## 1.1. Java 入门（基础概念与常识）

### 1.1.1. Java 语言有哪些特点?

1. 简单易学;
2. 面向对象（封装，继承，多态）；
3. 平台无关性（Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；
6. 支持多线程（C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；
7. 支持网络编程并且很方便（Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
8. 编译与解释并存；

修正（参见：[issue#544](#)）：C++11 开始（2011 年的时候），C++ 就引入了多线程库，在 windows、linux、macos 都可以使用 `std::thread` 和 `std::async` 来创建线程。参考链接：<http://www.cplusplus.com/reference/thread/thread/?kw=thread>

### 1.1.2. 关于 JVM JDK 和 JRE 最详细通俗的解答

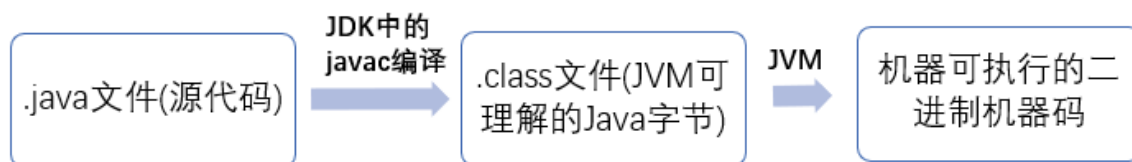
#### 1.1.2.1. JVM

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们都会给出相同的结果。

#### 什么是字节码?采用字节码的好处是什么?

在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 `.class` 的文件），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，在一定程度上解决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 程序运行时比较高效，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同操作系统的计算机上运行。

Java 程序从源代码到运行一般有下面 3 步：



我们需要格外注意的是 .class->机器码 这一步。在这一步 JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的(也就是所谓的热点代码)，所以后面引进了 JIT 编译器，而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 Java 是编译与解释共存的语言。

HotSpot 采用了惰性评估(Lazy Evaluation)的做法，根据二八定律，消耗大部分系统资源的只有那一小部分的代码（热点代码），而这也就是 JIT 所需要编译的部分。JVM 会根据代码每次被执行的情况收集信息并相应地做出一些优化，因此执行的次数越多，它的速度就越快。JDK 9 引入了一种新的编译模式 AOT(Ahead of Time Compilation)，它是直接将字节码编译成机器码，这样就避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。但是，AOT 编译器的编译质量是肯定比不上 JIT 编译器的。

## 总结：

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows, Linux, macOS），目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。

### 1.1.2.2. JDK 和 JRE

JDK 是 Java Development Kit，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet。

### 1.1.3. Oracle JDK 和 OpenJDK 的对比



可能在看这个问题之前很多人和我一样并没有接触和使用过 OpenJDK。那么 Oracle 和 OpenJDK 之间是否存在重大差异? 下面我通过收集到的一些资料, 为你解答这个被很多人忽视的问题。

对于 Java 7, 没什么关键的地方。OpenJDK 项目主要基于 Sun 捐赠的 HotSpot 源代码。此外, OpenJDK 被选为 Java 7 的参考实现, 由 Oracle 工程师维护。关于 JVM, JDK, JRE 和 OpenJDK 之间的区别, Oracle 博客帖子在 2012 年有一个更详细的答案:

问: OpenJDK 存储库中的源代码与用于构建 Oracle JDK 的代码之间有什么区别?

答: 非常接近 - 我们的 Oracle JDK 版本构建过程基于 OpenJDK 7 构建, 只添加了几个部分, 例如部署代码, 其中包括 Oracle 的 Java 插件和 Java WebStart 的实现, 以及一些封闭的源代码派对组件, 如图形光栅化器, 一些开源的第三方组件, 如 Rhino, 以及一些零碎的东西, 如附加文档或第三方字体。展望未来, 我们的目的是开源 Oracle JDK 的所有部分, 除了我们考虑商业功能的部分。

## 总结:

1. Oracle JDK 大概每 6 个月发一次主要版本, 而 OpenJDK 版本大概每三个月发布一次。但这不是固定的, 我觉得了解这个没啥用处。详情参见: <https://blogs.oracle.com/java-platform-group/update-and-faq-on-the-java-se-release-cadence>。
2. OpenJDK 是一个参考模型并且是完全开源的, 而 Oracle JDK 是 OpenJDK 的一个实现, 并不是完全开源的;
3. 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程序崩溃的问题, 但是, 只需切换到 Oracle JDK 就可以解决问题;
4. 在响应性和 JVM 性能方面, Oracle JDK 与 OpenJDK 相比提供了更好的性能;
5. 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本获得支持来获取最新版本;
6. Oracle JDK 根据二进制代码许可协议获得许可, 而 OpenJDK 根据 GPL v2 许可获得许可。

### 1.1.4. Java 和 C++的区别?

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! 没办法!!! 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, 但是接口可以多继承。
- Java 有自动内存管理机制, 不需要程序员手动释放无用内存
- 在 C 语言中, 字符串或字符数组最后都会有一个额外的字符 '\0' 来表示结束。但是, Java 语言中没有结束符这一概念。这是一个值得深度思考的问题, 具体原因推荐看这篇文章:

### 1.1.5. 什么是 Java 程序的主类 应用程序和小程序的主类有何不同?

一个程序中可以有多个类，但只能有一个类是主类。在 Java 应用程序中，这个主类是指包含 `main()` 方法的类。而在 Java 小程序中，这个主类是一个继承自系统类 `JApplet` 或 `Applet` 的子类。应用程序的主类不一定要是 `public` 类，但小程序的主类要求必须是 `public` 类。主类是 Java 程序执行的入口点。

### 1.1.6. Java 应用程序与小程序之间有哪些差别?

简单说应用程序是从主线程启动(也就是 `main()` 方法)。applet 小程序没有 `main()` 方法，主要是嵌在浏览器页面上运行(调用 `init()` 或者 `run()` 来启动)，嵌入浏览器这点跟 flash 的小游戏类似。

### 1.1.7. import java 和 javax 有什么区别?

刚开始的时候 JavaAPI 所必需的包是 `java` 开头的包，`javax` 当时只是扩展 API 包来使用。然而随着时间的推移，`javax` 逐渐地扩展成为 Java API 的组成部分。但是，将扩展从 `javax` 包移动到 `java` 包确实太麻烦了，最终会破坏一堆现有的代码。因此，最终决定 `javax` 包将成为标准 API 的一部分。

所以，实际上 `java` 和 `javax` 没有区别。这都是一个名字。

### 1.1.8. 为什么说 Java 语言“编译与解释并存”?

高级编程语言按照程序的执行方式分为编译型和解释型两种。简单来说，编译型语言是指编译器针对特定的操作系统将源代码一次性翻译成可被该平台执行的机器码；解释型语言是指解释器对源程序逐行解释成特定平台的机器码并立即执行。比如，你想阅读一本英文名著，你可以找一个英文翻译人员帮助你阅读，有两种选择方式，你可以先等翻译人员将全本的英文名著（也就是源码）都翻译成汉语，再去阅读，也可以让翻译人员翻译一段，你在旁边阅读一段，慢慢把书读完。

Java 语言既具有编译型语言的特征，也具有解释型语言的特征，因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码（\*.class 文件），这种字节码必须由 Java 解释器来解释执行。因此，我们可以认为 Java 语言编译与解释并存。

## 1.2. Java 语法

### 1.2.1. 字符型常量和字符串常量的区别?

1. 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
2. 含义上: 字符常量相当于一个整型值(ASCII 值), 可以参加表达式运算; 字符串常量代表一个地址值(该字符串在内存中存放位置)
3. 占内存大小 字符常量只占 2 个字节; 字符串常量占若干个字节 (注意: `char` 在 Java 中占两个字节)



java 编程思想第四版：2.2.2 节

Java要确定每种基本类型所占存储空间的大小。它们的大小并不像其他大多数语言那样随机器硬件架构的变化而变化。这种所占存储空间大小的不变性是Java程序比用其他大多数语言编写的程序更具可移植性的原因之一。

基本类型	大小	最小值	最大值	包装器类型
<b>boolean</b>	—	—	—	<b>Boolean</b>
<b>char</b>	16-bit	Unicode 0	Unicode 2 <sup>16</sup> -1	<b>Character</b>
<b>byte</b>	8 bits	-128	+127	<b>Byte</b>
<b>short</b>	16 bits	-2 <sup>15</sup>	+2 <sup>15</sup> -1	<b>Short</b>
<b>int</b>	32 bits	-2 <sup>31</sup>	+2 <sup>31</sup> -1	<b>Integer</b>
<b>long</b>	64 bits	-2 <sup>63</sup>	+2 <sup>63</sup> -1	<b>Long</b>
<b>float</b>	32 bits	IEEE754	IEEE754	<b>Float</b>
<b>double</b>	64 bits	IEEE754	IEEE754	<b>Double</b>
<b>void</b>	—	—	—	<b>Void</b>

### 1.2.2. 关于注释？

Java 中的注释有三种：

1. 单行注释
2. 多行注释
3. 文档注释。

在我们编写代码的时候，如果代码量比较少，我们自己或者团队其他成员还可以很轻易地看懂代码，但是当项目结构一旦复杂起来，我们就需要用到注释了。注释并不会执行，是我们程序员写给自己看的，注释是你的代码说明书，能够帮助看代码的人快速地理清代码之间的逻辑关系。因此，在写程序的时候随手加上注释是一个非常好的习惯。

《Clean Code》这本书明确指出：

*代码的注释不是越详细越好。实际上好的代码本身就是注释，我们要尽量规范和美化自己的代码来减少不必要的注释。*

*若编程语言足够有表达力，就不需要注释，尽量通过代码来阐述。*

举个例子：

去掉下面复杂的注释，只需要创建一个与注释所言同一事物的函数即可

```
// check to see if the employee is eligible for full benefits
if ((employee.falgs & HOURLY_FLAG) && (employee.age > 65))
```

应替换为

```
if (employee.isEligibleForFullBenefits())
```

### 1.2.3. 标识符和关键字的区别是什么？

在我们编写程序的时候，需要大量地为程序、类、变量、方法等取名字，于是就有了标识符，简单来说，标识符就是一个名字。但是有一些标识符，Java 语言已经赋予了其特殊的含义，只能用于特定的地方，这种特殊的标识符就是关键字。因此，关键字是被赋予特殊含义的标识符。比如，在我们的日常生活中，“警察局”这个名字已经被赋予了特殊的含义，所以如果你开一家店，店的名字不能叫“警察局”，“警察局”就是我们日常生活中的关键字。

### 1.2.4. Java中有哪些常见的关键字？

访问控制	private	protected	public				
类，方法和变量修饰符	abstract	class	extends	final	implements	interface	native
	new	static	strictfp	synchronized	transient	volatile	
程序控制	break	continue	return	do	while	if	else
	for	instanceof	switch	case	default		
错误处理	try	catch	throw	throws	finally		
包相关	import	package					
基本类型	boolean	byte	char	double	float	int	long
	short	null	true	false			
变量引用	super	this	void				
保留字	goto	const					

### 1.2.5. 自增自减运算符

在写代码的过程中，常见的一种情况是需要某个整数类型变量增加 1 或减少 1，Java 提供了一种特殊的运算符，用于这种表达式，叫做自增运算符（++）和自减运算符（--）。

++和--运算符可以放在操作数之前，也可以放在操作数之后，当运算符放在操作数之前时，先自增/减，再赋值；当运算符放在操作数之后时，先赋值，再自增/减。例如，当“b=++a”时，先自增（自己增加 1），再赋值（赋值给 b）；当“b=a++”时，先赋值（赋值给 b），再自增（自己增加 1）。也就是，++a 输出的是 a+1 的值，a++输出的是 a 值。用一句口诀就是：“符号在前就先加/减，符号在后就后加/减”。

### 1.2.6. continue、break、和return的区别是什么？

在循环结构中，当循环条件不满足或者循环次数达到要求时，循环会正常结束。但是，有时候可能需要在循环的过程中，当发生了某种条件之后，提前终止循环，这就需要用到下面几个关键词：

1. continue：指跳出当前的这一次循环，继续下一次循环。
2. break：指跳出整个循环体，继续执行循环下面的语句。

return 用于跳出所在方法，结束该方法的运行。return 一般有两种用法：

1. return; : 直接使用 return 结束方法执行，用于没有返回值函数的方法
2. return value; : return 一个特定值，用于有返回值函数的方法

### 1.2.7. Java泛型了解么？什么是类型擦除？介绍一下常用的通配符？

Java 泛型（generics）是 JDK 5 中引入的一个新特性，泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

**Java的泛型是伪泛型，这是因为Java在编译期间，所有的泛型信息都会被擦掉，这也就是通常所说类型擦除。** 更多关于类型擦除的问题，可以查看这篇文章：[《Java泛型类型擦除以及类型擦除带来的问题》](#)。

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

list.add(12);
//这里直接添加会报错
list.add("a");
Class<? extends List> clazz = list.getClass();
Method add = clazz.getDeclaredMethod("add", Object.class);
//但是通过反射添加，是可以的
add.invoke(list, "kl");

System.out.println(list)
```

泛型一般有三种使用方式:泛型类、泛型接口、泛型方法。

#### 1.泛型类：

```
//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型
//在实例化泛型类时，必须指定T的具体类型
public class Generic<T>{

    private T key;

    public Generic(T key) {
        this.key = key;
    }

    public T getKey(){
        return key;
    }
}
```

如何实例化泛型类：

```
Generic<Integer> genericInteger = new Generic<Integer>(123456);
```

## 2. 泛型接口：

```
public interface Generator<T> {
    public T method();
}
```

实现泛型接口，不指定类型：

```
class GeneratorImpl<T> implements Generator<T>{
    @Override
    public T method() {
        return null;
    }
}
```

实现泛型接口，指定类型：

```
class GeneratorImpl<T> implements Generator<String>{
    @Override
    public String method() {
        return "hello";
    }
}
```

## 3. 泛型方法：

```
public static < E > void printArray( E[] inputArray )
{
    for ( E element : inputArray ){
        System.out.printf( "%s ", element );
    }
    System.out.println();
}
```

使用：

```
// 创建不同类型数组： Integer, Double 和 Character
Integer[] intArray = { 1, 2, 3 };
String[] stringArray = { "Hello", "World" };
printArray( intArray );
printArray( stringArray );
```

常用的通配符为： T, E, K, V, ?

- ? 表示不确定的 java 类型

- T (type) 表示具体的一个java类型
- K V (key value) 分别代表java键值中的Key Value
- E (element) 代表Element

更多关于Java 泛型中的通配符可以查看这篇文章: [《聊一聊-JAVA 泛型中的通配符 T, E, K, V, ? 》](#)

### 1.2.8. ==和equals的区别

**==**: 它的作用是判断两个对象的地址是不是相等。即判断两个对象是不是同一个对象。  
(基本数据类型==比较的是值, 引用数据类型==比较的是内存地址)

因为Java 只有值传递, 所以, 对于== 来说, 不管是比较基本数据类型, 还是引用数据类型的变量, 其本质比较的都是值, 只是引用类型变量存的值是对象的地址。

**equals()**: 它的作用也是判断两个对象是否相等, 它不能用于比较基本数据类型的变量。equals()方法存在于Object类中, 而Object类是所有类的直接或间接父类。

Object类equals()方法:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

equals() 方法存在两种使用情况:

- 情况 1: 类没有覆盖 equals()方法。则通过equals()比较该类的两个对象时, 等价于通过“==”比较这两个对象。使用的默认是 Object类equals()方法。
- 情况 2: 类覆盖了 equals()方法。一般, 我们都覆盖 equals()方法来两个对象的内容相等; 若它们的内容相等, 则返回 true(即, 认为这两个对象相等)。

举个例子:

```
public class test1 {  
    public static void main(String[] args) {  
        String a = new String("ab"); // a 为一个引用  
        String b = new String("ab"); // b为另一个引用, 对象的内容一样  
        String aa = "ab"; // 放在常量池中  
        String bb = "ab"; // 从常量池中查找  
        if (aa == bb) // true  
            System.out.println("aa==bb");  
        if (a == b) // false, 非同一对象  
            System.out.println("a==b");  
        if (a.equals(b)) // true  
            System.out.println("aEqb");  
        if (42 == 42.0) { // true  
            System.out.println("true");  
        }  
    }  
}
```

```
}  
}
```

说明：

- String 中的 equals 方法是被重写过的，因为 Object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。
- 当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

String类equals()方法：

```
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;  
}
```

### 1.2.9. hashCode()与 equals()

面试官可能会问你：“你重写过 hashCode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

#### 1)hashCode()介绍：

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在 JDK 的 Object 类中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。另外需要注意的是：Object 的 hashCode 方法是本地方法，也就是用 C 语言或 C++ 实现的，该方法通常用来将对象的内存地址转换为整数之后返回。



```
public native int hashCode();
```

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

## 2)为什么要有 hashCode?

我们以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode?

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashCode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashCode 值作比较，如果没有相符的 hashCode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashCode 值的对象，这时会调用 equals () 方法来检查 hashCode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head fist java》第二版）。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

## 3)为什么重写 equals 时必须重写 hashCode 方法?

如果两个对象相等，则 hashCode 一定也是相同的。两个对象相等,对两个对象分别调用 equals 方法都返回 true。但是，两个对象有相同的 hashCode 值，它们也不一定是相等的。因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖。

*hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）*

## 4)为什么两个对象有相同的 hashCode 值，它们也不一定是相等的?

在这里解释一位小伙伴的问题。以下内容摘自《Head Fisrt Java》。

因为 hashCode() 所使用的杂凑算法也许刚好会让多个对象传回相同的杂凑值。越糟糕的杂凑算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode）。

我们刚刚也提到了 HashSet,如果 HashSet 在对比的时候，同样的 hashCode 有多个对象，它会使用 equals() 来判断是否真的相同。也就是说 hashCode 只是用来缩小查找成本。

更多关于 hashCode() 和 equals() 的内容可以查看：[Java hashCode\(\) 和 equals\(\) 的若干问题解答](#)

## 1.3. 基本数据类型

### 1.3.1. Java中的几种基本数据类型是什么？对应的包装类型是什么？各自占用多少字节呢？

Java中有8种基本数据类型，分别为：

1. 6种数字类型：byte、short、int、long、float、double
2. 1种字符类型：char
3. 1中布尔型：boolean。

这八种基本类型都有对应的包装类分别为：Byte、Short、Integer、Long、Float、Double、Character、Boolean

基本类型	位数	字节	默认值
int	32	4	0
short	16	2	0
long	64	8	0L
byte	8	1	0
char	16	2	'u0000'
float	32	4	of
double	64	8	od
boolean	1		false

对于boolean，官方文档未明确定义，它依赖于 JVM 厂商的具体实现。逻辑上理解是占用 1位，但是实际中会考虑计算机高效存储因素。

注意：

1. Java 里使用 long 类型的数据一定要在数值后面加上 **L**，否则将作为整型解析：
2. char a = 'h':char:单引号, String a = "hello":双引号

### 1.3.2. 自动装箱与拆箱

- **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；

更多内容见：[深入剖析Java中的装箱和拆箱](#)

### 1.3.3. 8种基本类型的包装类和常量池

Java 基本类型的包装类的大部分都实现了常量池技术，即 `Byte, Short, Integer, Long, Character, Boolean`；前面 4 种包装类默认创建了数值 `[-128, 127]` 的相应类型的缓存数据，`Character` 创建了数值在 `[0, 127]` 范围的缓存数据，`Boolean` 直接返回 `True Or False`。如果超出对应范围仍然会去创建新的对象。为啥把缓存设置为 `[-128, 127]` 区间？（参见[issue/461](#)）性能和资源之间的权衡。

```
public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

```
private static class CharacterCache {
    private CharacterCache() {}

    static final Character cache[] = new Character[127 + 1];
    static {
        for (int i = 0; i < cache.length; i++)
            cache[i] = new Character((char)i);
    }
}
```

两种浮点数类型的包装类 `Float, Double` 并没有实现常量池技术。 \*\*

```
Integer i1 = 33;
Integer i2 = 33;
System.out.println(i1 == i2); // 输出 true
Integer i11 = 333;
Integer i22 = 333;
System.out.println(i11 == i22); // 输出 false
Double i3 = 1.2;
Double i4 = 1.2;
System.out.println(i3 == i4); // 输出 false
```

**Integer 缓存源代码：**

```
/**
 *此方法将始终缓存-128 到 127（包括端点）范围内的值，并可以缓存此范围之外的其他值。
 */
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

**应用场景：**

1. `Integer i1=40`；Java 在编译的时候会直接将代码封装成 `Integer`

- `i1=Integer.valueOf(40);`，从而使用常量池中的对象。
2. `Integer i1 = new Integer(40);`这种情况下会创建新的对象。

```
Integer i1 = 40;
Integer i2 = new Integer(40);
System.out.println(i1==i2); //输出 false
```

**Integer 比较更丰富的一个例子：**

```
Integer i1 = 40;
Integer i2 = 40;
Integer i3 = 0;
Integer i4 = new Integer(40);
Integer i5 = new Integer(40);
Integer i6 = new Integer(0);

System.out.println("i1=i2    " + (i1 == i2));
System.out.println("i1=i2+i3  " + (i1 == i2 + i3));
System.out.println("i1=i4    " + (i1 == i4));
System.out.println("i4=i5    " + (i4 == i5));
System.out.println("i4=i5+i6  " + (i4 == i5 + i6));
System.out.println("40=i5+i6  " + (40 == i5 + i6));
```

结果：

```
i1=i2    true
i1=i2+i3  true
i1=i4    false
i4=i5    false
i4=i5+i6  true
40=i5+i6  true
```

解释：

语句 `i4 == i5 + i6`，因为`+`这个操作符不适用于 `Integer` 对象，首先 `i5` 和 `i6` 进行自动拆箱操作，进行数值相加，即 `i4 == 40`。然后 `Integer` 对象无法与数值进行直接比较，所以 `i4` 自动拆箱转为 `int` 值 `40`，最终这条语句转为 `40 == 40` 进行数值比较。

## 1.4. 方法（函数）

### 1.4.1. 什么是方法的返回值？返回值在类的方法里的作用是什么？

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

### 1.4.2. 为什么 Java 中只有值传递？

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。**按值调用(call by value)**表示方法接收的是调用者提供的值，而**按引用调用(call by reference)**表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。它用来描述各种程序设计语言（不只是Java)中方法参数传递方式。

**Java** 程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

下面通过 3 个例子来给大家说明

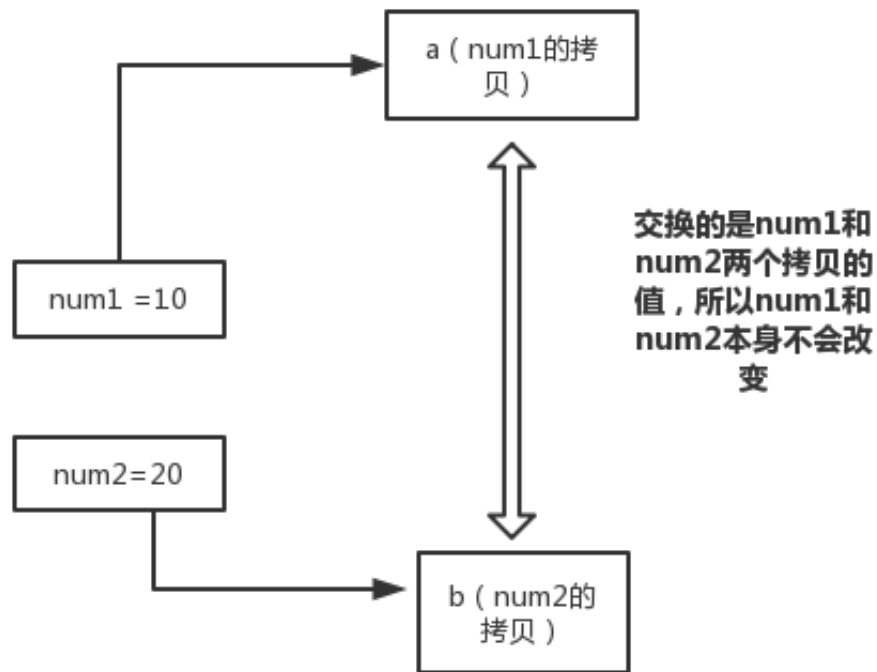
example 1

```
public static void main(String[] args) {  
    int num1 = 10;  
    int num2 = 20;  
  
    swap(num1, num2);  
  
    System.out.println("num1 = " + num1);  
    System.out.println("num2 = " + num2);  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
  
    System.out.println("a = " + a);  
    System.out.println("b = " + b);  
}
```

结果：

```
a = 20  
b = 10  
num1 = 10  
num2 = 20
```

解析：



在 swap 方法中，a、b 的值进行交换，并不会影响到 num1、num2。因为，a、b 中的值，只是从 num1、num2 的复制过来的。也就是说，a、b 相当于 num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而对对象引用作为参数就不一样，请看 **example2**。

#### example 2

```

public static void main(String[] args) {
    int[] arr = { 1, 2, 3, 4, 5 };
    System.out.println(arr[0]);
    change(arr);
    System.out.println(arr[0]);
}

public static void change(int[] array) {
    // 将数组的第一个元素变为0
    array[0] = 0;
}

```

结果：

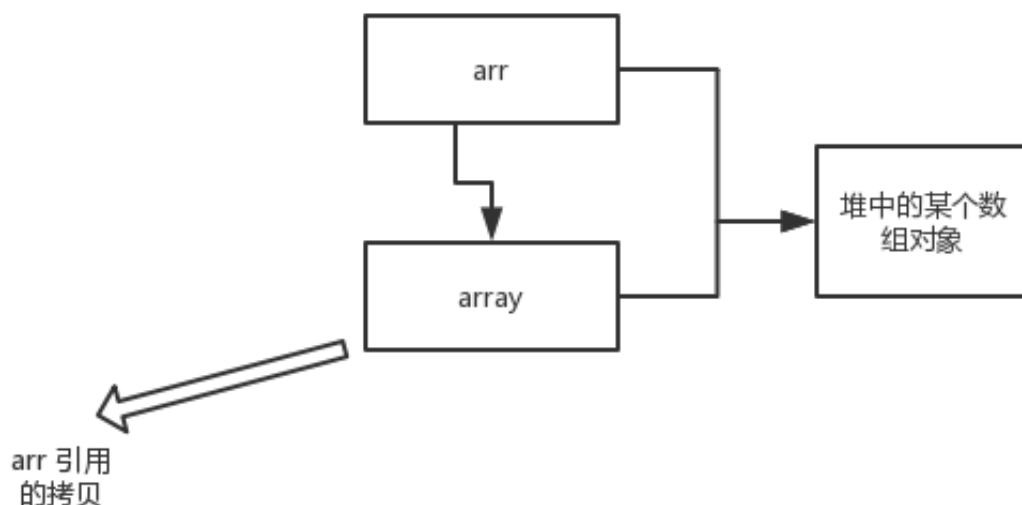
```

1
0

```



解析：



array 被初始化 arr 的拷贝也就是一个对象的引用，也就是说 array 和 arr 指向的是同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象上。

通过 **example2** 我们已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和 Pascal)提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为 Java 程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

### example 3

```

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student s1 = new Student("小张");
        Student s2 = new Student("小李");
        Test.swap(s1, s2);
        System.out.println("s1:" + s1.getName());
        System.out.println("s2:" + s2.getName());
    }

    public static void swap(Student x, Student y) {
        Student temp = x;
        x = y;
        y = temp;
    }
}
  
```

```

        System.out.println("x:" + x.getName());
        System.out.println("y:" + y.getName());
    }
}

```

结果：

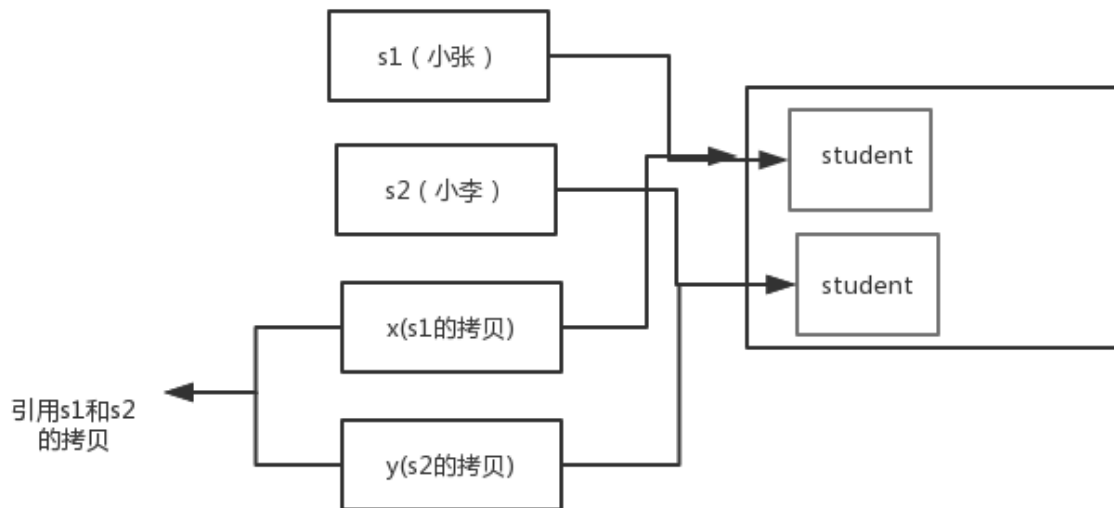
```

x:小李
y:小张
s1:小张
s2:小李

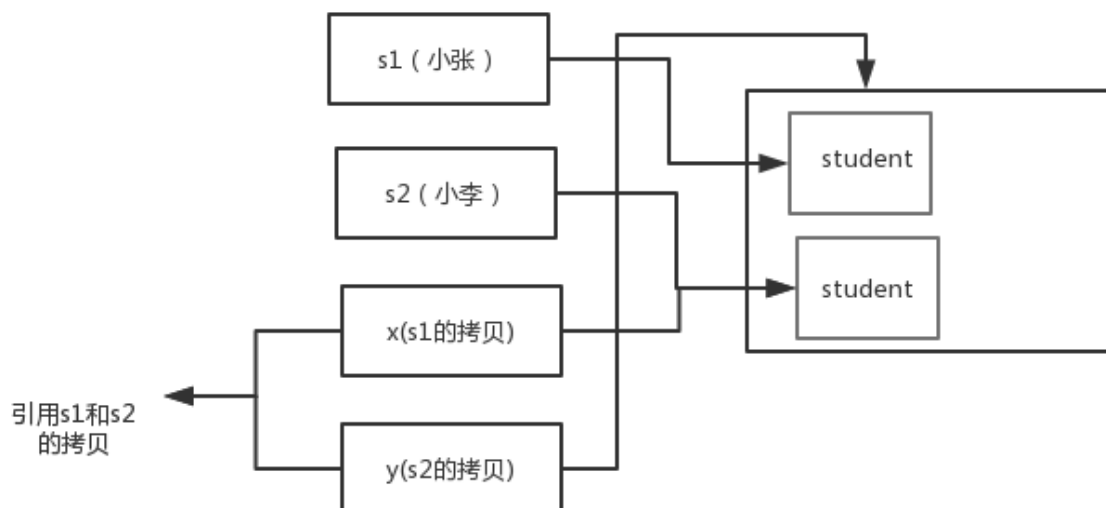
```

解析：

交换之前：



交换之后：



通过上面两张图可以很清晰的看出：**方法并没有改变存储在变量 `s1` 和 `s2` 中的对象引用**。**`swap` 方法的参数 `x` 和 `y` 被初始化为两个对象引用的拷贝**，这个方法交换的是这两个拷贝

## 总结

Java 程序设计语言对对象采用的不是引用调用，实际上，对象引用是按 值传递的。

下面再总结一下 Java 中方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型或布尔型）。
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能让对象参数引用一个新的对象。

参考：

《Java 核心技术卷 I》基础知识第十版第四章 4.5 小节

## 1.4.3. 重载和重写的区别

重载就是同样的一个方法能够根据输入数据的不同，做出不同的处理

重写就是当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法

### 1.4.3.1. 重载

发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。

下面是《Java 核心技术》对重载这个概念的介绍：

#### 4.6.1 重载


有些类有多个构造器。例如，可以如下构造一个空的 `StringBuilder` 对象：

```
StringBuilder messages = new StringBuilder();
```

或者，可以指定一个初始字符串：

```
StringBuilder todoList = new StringBuilder("To do:\n");
```

这种特征叫做**重载**（**overloading**）。如果多个方法（比如，`StringBuilder` 构造器方法）有相同的名字、不同的参数，便产生了**重载**。编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，就会产生编译时错误，因为根本不存在匹配，或者没有一个比其他的更好。（这个过程被称为**重载解析**（**overloading resolution**）。）

 **注释：**Java 允许**重载**任何方法，而不只是构造器方法。因此，要完整地描述一个方法，需要指出方法名以及参数类型。这叫做方法的签名（**signature**）。例如，`String` 类有 4 个称为 `indexOf` 的公有方法。它们的签名是

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

返回类型不是方法签名的一部分。也就是说，不能有两个名字相同、参数类型也相同却返回不同类型值的方法。

综上：重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。

#### 1.4.3.2. 重写

重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。

1. 返回值类型、方法名、参数列表必须相同，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。
2. 如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法，但是被 `static` 修饰的方法能够被再次声明。
3. 构造方法无法被重写

综上：重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变

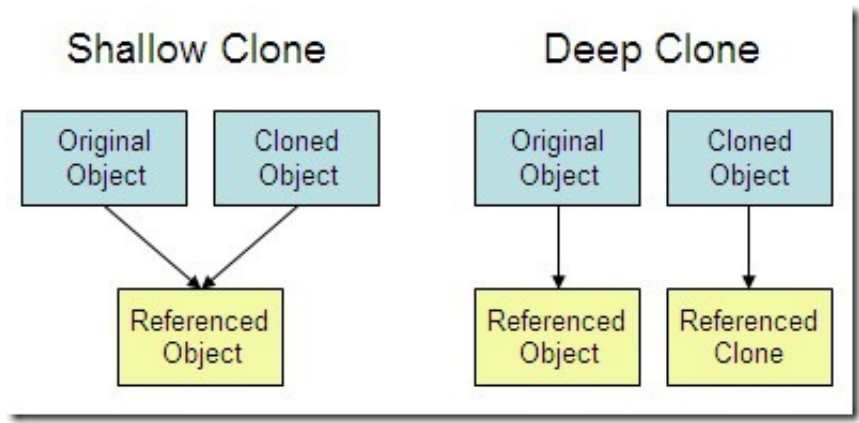
暖心的 Guide 哥最后再来个图标总结一下！

区别点	重载方法	重写方法
发生范围	同一个类	子类中
参数列表	必须修改	一定不能修改
返回类型	可修改	一定不能修改
异常	可修改	可以减少或删除，一定不能抛出新的或者更广的异常

访问修饰符	可修改	一定不能做更严格的限制（可以降低限制）
发生阶段	编译期	运行期

### 1.4.4. 深拷贝 vs 浅拷贝

1. **浅拷贝**：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝。
2. **深拷贝**：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝。



### 1.4.5. 方法的四种类型

#### 1、无参数无返回值的方法

```
// 无参数无返回值的方法 (如果方法没有返回值，不能不写，必须写void，表示没有返回值)
public void f1() {
    System.out.println("无参数无返回值的方法");
}
```

#### 2、有参数无返回值的方法

```
/**
 * 有参数无返回值的方法
 * 参数列表由零组到多组“参数类型+形参名”组合而成，多组参数之间以英文逗号（,）隔开，
 * 形参类型和形参名之间以英文空格隔开
 */
public void f2(int a, String b, int c) {
    System.out.println(a + "-->" + b + "-->" + c);
}
```

#### 3、有返回值无参数的方法

```
// 有返回值无参数的方法（返回值可以是任意的类型，在函数里面必须有return关键字返回对应的类型）
public int f3() {
    System.out.println("有返回值无参数的方法");
    return 2;
}
```

#### 4、有返回值有参数的方法

```
// 有返回值有参数的方法
public int f4(int a, int b) {
    return a * b;
}
```

#### 5、return 在无返回值方法的特殊使用

```
// return在无返回值方法的特殊使用
public void f5(int a) {
    if (a>10) {
        return; //表示结束所在方法（f5方法）的执行，下方的输出语句不会执行
    }

    System.out.println(a);
}
```

## 2. Java 面向对象

### 2.1. 类和对象

#### 2.1.1. 面向对象和面向过程的区别

- **面向过程**：面向过程性能比面向对象高。因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发。但是，**面向过程没有面向对象易维护、易复用、易扩展**。
- **面向对象**：面向对象易维护、易复用、易扩展。因为面向对象有封装、继承、多态性的特性，所以可以设计出低耦合的系统，使系统更加灵活、更加易于维护。但是，**面向对象性能比面向过程低**。

参见 issue：[面向过程：面向过程性能比面向对象高??](#)

这个并不是根本原因，面向过程也需要分配内存，计算内存偏移量，Java 性能差的主要原因并不是因为它是面向对象语言，而是 Java 是半编译语言，最终的执行代码并不是可以直接被 CPU 执行的二进制机械码。

而面向过程语言大多都是直接编译成机械码在电脑上执行，并且其它一些面向过程的脚本语言性能也并不一定比 Java 好。



### 2.1.2. 构造器 Constructor 是否可被 override?

Constructor 不能被 override (重写),但是可以 overload (重载),所以你可以看到一个类中有多个构造函数的情况。

### 2.1.3. 在 Java 中定义一个不做事且没有参数的构造方法的作用

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

### 2.1.4. 成员变量与局部变量的区别有哪些?

1. 从语法形式上看:成员变量是属于类的,而局部变量是在方法中定义的变量或是方法的参数;成员变量可以被 `public,private,static` 等修饰符所修饰,而局部变量不能被访问控制修饰符及 `static` 所修饰;但是,成员变量和局部变量都能被 `final` 所修饰。
2. 从变量在内存中的存储方式来看:如果成员变量是使用 `static` 修饰的,那么这个成员变量是属于类的,如果没有使用 `static` 修饰,这个成员变量是属于实例的。而对象存在于堆内存,局部变量则存在于栈内存。
3. 从变量在内存中的生存时间上看:成员变量是对象的一部分,它随着对象的创建而存在,而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值:则会自动以类型的默认值而赋值(一种情况例外:被 `final` 修饰的成员变量也必须显式地赋值),而局部变量则不会自动赋值。

### 2.1.5. 创建一个对象用什么运算符?对象实体与对象引用有何不同?

`new` 运算符, `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(对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,也可以系一个气球);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)。

### 2.1.6. 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, 该程序能正确执行吗? 为什么?

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法也会有默认的不带参数的构造方法。如果我们自己添加了类的构造方法(无论是否有参),Java 就不会再添加默认的无参数的构造方法了,这时候,就不能直接 `new` 一个对象而不传递参数了,所以我们一直在不知不觉地使用构造方法,这也是为什么我们在创建对象的时候后面要加一个括号(因为要调用无参的构造方法)。如果我们重载了有参的构造方法,记得都要把无参的构造方法也写出来(无论是否用到),因为这可以帮助我们在创建对象的时候少踩坑。

### 2.1.7. 构造方法有哪些特性?

1. 名字与类名相同。

2. 没有返回值，但不能用 `void` 声明构造函数。
3. 生成类的对象时自动执行，无需调用。

### 2.1.8. 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?

帮助子类做初始化工作。

### 2.1.9. 对象的相等与指向他们的引用相等,两者有什么不同?

对象的相等，比的是内存中存放的内容是否相等。而引用相等，比较的是他们指向的内存地址是否相等。

## 2.2. 面向对象三大特征

### 2.2.1. 封装

封装是指把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。就好像我们看不到挂在墙上的空调的内部的零件信息（也就是属性），但是可以通过遥控器（方法）来控制空调。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。但是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。就好像如果没有空调遥控器，那么我们就无法操控空调制冷，空调本身就没有意义了（当然现在还有很多其他方法，这里只是为了举例子）。

```
public class Student {  
    private int id;//id属性私有化  
    private String name;//name属性私有化  
  
    //获取id的方法  
    public int getId() {  
        return id;  
    }  
  
    //设置id的方法  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    //获取name的方法  
    public String getName() {  
        return name;  
    }  
  
    //设置name的方法  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

### 2.2.2. 继承

不同类型的对象，相互之间经常有一定数量的共同点。例如，小明同学、小红同学、小李同学，都共享学生的特性（班级、学号等）。同时，每一个对象还定义了额外的特性使得他们与众不同。例如小明的数学比较好，小红的性格惹人喜爱；小李的力气比较大。继承是使用已存在的类的定义作为基础建立新类的技术，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父类。通过使用继承，可以快速地创建新的类，可以提高代码的重用，程序的可维护性，节省大量创建新类的时间，提高我们的开发效率。

关于继承如下 3 点请记住：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，**只是拥有**。
2. 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。
3. 子类可以用自己的方式实现父类的方法。（以后介绍）。

### 2.2.3. 多态

多态，顾名思义，表示一个对象具有多种的状态。具体表现为父类的引用指向子类的实例。

多态的特点：

- 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
- 对象类型不可变，引用类型可变；
- 方法具有多态性，属性不具有多态性；
- 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
- 多态不能调用“只在子类存在但在父类不存在”的方法；
- 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。

## 2.3. 修饰符

### 2.3.1. 在一个静态方法内调用一个非静态成员为什么是非法的？

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量，也不可以访问非静态变量成员。

### 2.3.2. 静态方法和实例方法有何不同

1. 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
2. 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。

### 2.3.3. 常见关键字总结:static,final,this,super

详见笔主的这篇文章: <https://snailclimb.gitee.io/javaguide/#/docs/java/basic/final,static,this,super>

## 2.4. 接口和抽象类

### 2.4.1. 接口和抽象类的区别是什么?

1. 接口的方法默认是 `public`，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现)，而抽象类可以有非抽象的方法。
2. 接口中除了 `static`、`final` 变量，不能有其他变量，而抽象类中则不一定。
3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 `extends` 关键字扩展多个接口。
4. 接口方法默认修饰符是 `public`，抽象方法可以有 `public`、`protected` 和 `default` 这些修饰符（抽象方法就是为了被重写所以不能使用 `private` 关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注:

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，否则会报错。（详见 [issue:https://github.com/Snailclimb/JavaGuide/issues/146](https://github.com/Snailclimb/JavaGuide/issues/146)。
2. jdk9 的接口被允许定义私有方法。

总结一下 jdk7~jdk9 Java 中接口概念的变化（[相关阅读](#)）：

1. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。
2. jdk8 的时候接口可以有默认方法和静态方法功能。
3. Jdk 9 在接口中引入了私有方法和私有静态方法。

## 2.5. 其它重要知识点

### 2.5.1. String StringBuffer 和 StringBuilder 的区别是什么? String 为什么是不可变的?

简单的来说：String 类中使用 `final` 关键字修饰字符数组来保存字符串，`private final char value[]`，所以String对象是不可变的。

补充（来自[issue 675](#)）：在 Java 9 之后，String 类的实现改用 byte 数组存储字符串 `private final byte[] value;`

而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`但是没有用`final`关键字修饰，所以这两种对象都是可变的。

StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是 AbstractStringBuilder 实现的，大家可以自行查阅源码。

#### AbstractStringBuilder.java

```
abstract class AbstractStringBuilder implements Appendable,
CharSequence {
    /**
     * The value is used for character storage.
     */
    char[] value;

    /**
     * The count is the number of characters used.
     */
    int count;

    AbstractStringBuilder(int capacity) {
        value = new char[capacity];
    }
}
```

### 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步锁，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，然后将指针指向新的 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并改变对象引用。相同情况下使用 StringBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

#### 对于三者使用的总结：

1. 操作少量的数据: 适用 String
2. 单线程操作字符串缓冲区下操作大量数据: 适用 StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据: 适用 StringBuffer

### 2.5.2. Object 类的常见方法总结

Object 类是一个特殊的类，是所有类的父类。它主要提供了以下 11 个方法：

`public final native Class<?> getClass()`//native方法，用于返回当前运行时对象的Class对象，使用了final关键字修饰，故不允许子类重写。

`public native int hashCode()` //native方法，用于返回对象的哈希码，主要使用在哈希表中，比如JDK中的HashMap。

`public boolean equals(Object obj)`//用于比较2个对象的内存地址是否相等，String类对该方法进行了重写用户比较字符串的值是否相等。

`protected native Object clone() throws CloneNotSupportedException`//native方法，用于创建并返回当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 `x.clone() != x` 为true，`x.clone().getClass() == x.getClass()` 为true。Object本身没有实现Cloneable接口，所以不重写clone方法并且进行调用的话会发生CloneNotSupportedException异常。

`public String toString()`//返回类的名字@实例的哈希码的16进制的字符串。建议Object所有的子类都重写这个方法。

`public final native void notify()`//native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。

`public final native void notifyAll()`//native方法，并且不能重写。跟notify一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。

`public final native void wait(long timeout) throws InterruptedException`//native方法，并且不能重写。暂停线程的执行。注意：sleep方法没有释放锁，而wait方法释放了锁。timeout是等待时间。

`public final void wait(long timeout, int nanos) throws InterruptedException`//多了nanos参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。所以超时的时间还需要加上nanos毫秒。

`public final void wait() throws InterruptedException`//跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念

`protected void finalize() throws Throwable { }`//实例被垃圾回收器回收的时候触发的操作

### 2.5.3. == 与 equals(重要)

**==**：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象（基本数据类型==比较的是值，引用数据类型==比较的是内存地址）。

**equals()**：它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 情况 1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
- 情况 2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对



象的内容是否相等；若它们的内容相等，则返回 `true` (即，认为这两个对象相等)。

举个例子：

```
public class test1 {
    public static void main(String[] args) {
        String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b为另一个引用,对象的内容一样
        String aa = "ab"; // 放在常量池中
        String bb = "ab"; // 从常量池中查找
        if (aa == bb) // true
            System.out.println("aa==bb");
        if (a == b) // false, 非同一对象
            System.out.println("a==b");
        if (a.equals(b)) // true
            System.out.println("aEQb");
        if (42 == 42.0) { // true
            System.out.println("true");
        }
    }
}
```

说明：

- `String` 中的 `equals` 方法是被重写过的，因为 `object` 的 `equals` 方法是比较的对象的内存地址，而 `String` 的 `equals` 方法比较的是对象的值。
- 当创建 `String` 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 `String` 对象。

## 2.5.4. hashCode 与 equals (重要)

面试官可能会问你：“你重写过 `hashCode` 和 `equals` 么，为什么重写 `equals` 时必须重写 `hashCode` 方法？”

### 2.5.4.1. hashCode () 介绍

`hashCode()` 的作用是获取哈希码，也称为散列码；它实际上是返回一个 `int` 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。`hashCode()` 定义在 `JDK` 的 `Object.java` 中，这就意味着 `Java` 中的任何类都包含有 `hashCode()` 函数。

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

### 2.5.4.2. 为什么要有 hashCode

我们先以“**HashSet 如何检查重复**”为例子来说明为什么要有 **hashCode**：当你把对象加入 HashSet 时，HashSet 会先计算对象的 **hashCode** 值来判断对象加入的位置，同时也会与该位置其他已经加入的对象的 **hashCode** 值作比较，如果没有相符的 **hashCode**，HashSet 会假设对象没有重复出现。但是如果有相同 **hashCode** 值的对象，这时会调用 **equals()** 方法来检查 **hashCode** 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head first java》第二版）。这样我们就大大减少了 **equals** 的次數，相应就大大提高了执行速度。

通过我们可以看出：**hashCode()** 的作用就是**获取哈希码**，也称为散列码；它实际上是返回一个 **int** 整数。这个**哈希码**的作用是确定该对象在哈希表中的索引位置。**hashCode()**在散列表中才有用，在其它情况下没用。在散列表中 **hashCode()** 的作用是获取对象的散列码，进而确定该对象在散列表中的位置。

#### 2.5.4.3. hashCode () 与 equals () 的相关规定

1. 如果两个对象相等，则 **hashCode** 一定也是相同的
2. 两个对象相等,对两个对象分别调用 **equals** 方法都返回 **true**
3. 两个对象有相同的 **hashCode** 值，它们也不一定是相等的
4. 因此，**equals** 方法被覆盖过，则 **hashCode** 方法也必须被覆盖
5. **hashCode()** 的默认行为是对堆上的对象产生独特值。如果没有重写 **hashCode()**，则该 **class** 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

推荐阅读：[Java hashCode\(\) 和 equals\(\) 的若干问题解答](#)

#### 2.5.5. Java 序列化中如果有些字段不想进行序列化，怎么办？

对于不想进行序列化的变量，使用 **transient** 关键字修饰。

**transient** 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 **transient** 修饰的变量值不会被持久化和恢复。**transient** 只能修饰变量，不能修饰类和方法。

#### 2.5.6. 获取用键盘输入常用的两种方法

方法 1：通过 Scanner

```
Scanner input = new Scanner(System.in);
String s = input.nextLine();
input.close();
```

方法 2：通过 BufferedReader

```
BufferedReader input = new BufferedReader(new
InputStreamReader(System.in));
String s = input.readLine();
```

## 3. Java 核心技术

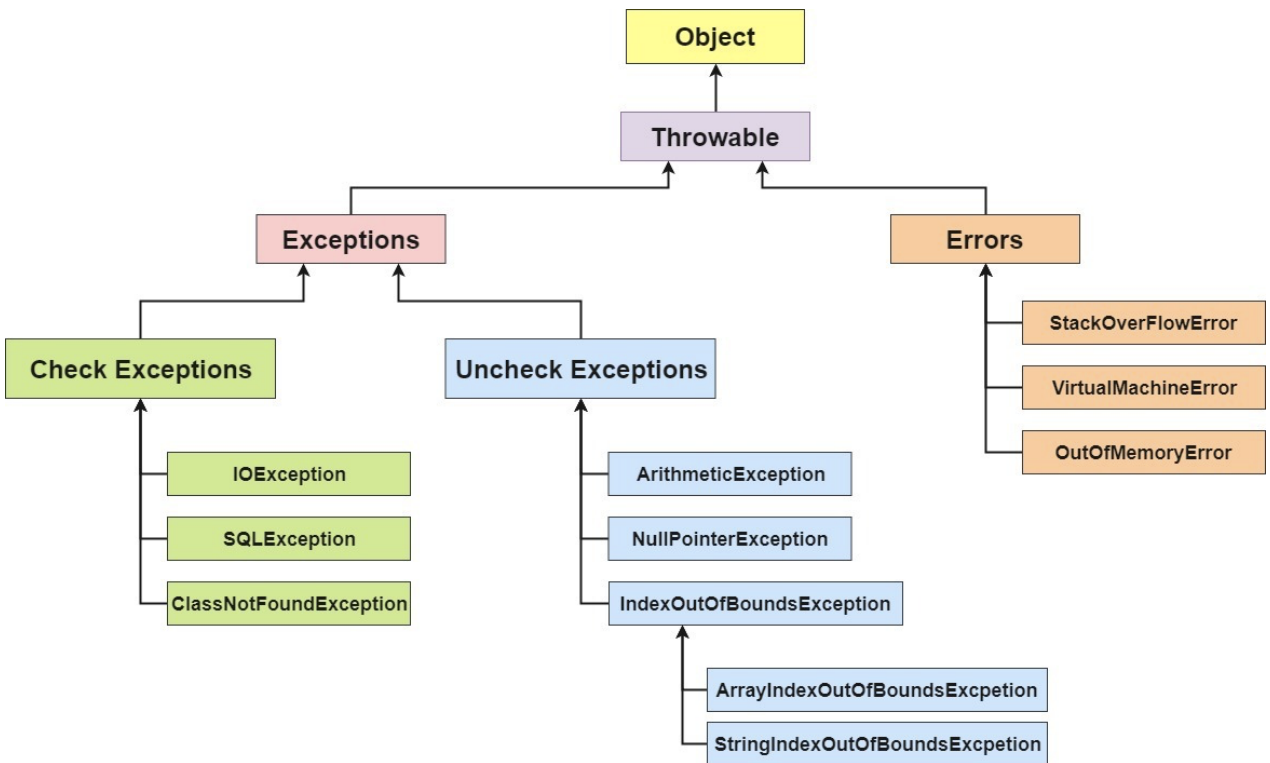
### 3.1. 集合

#### 3.1.1. Collections 工具类和 Arrays 工具类常见方法总结

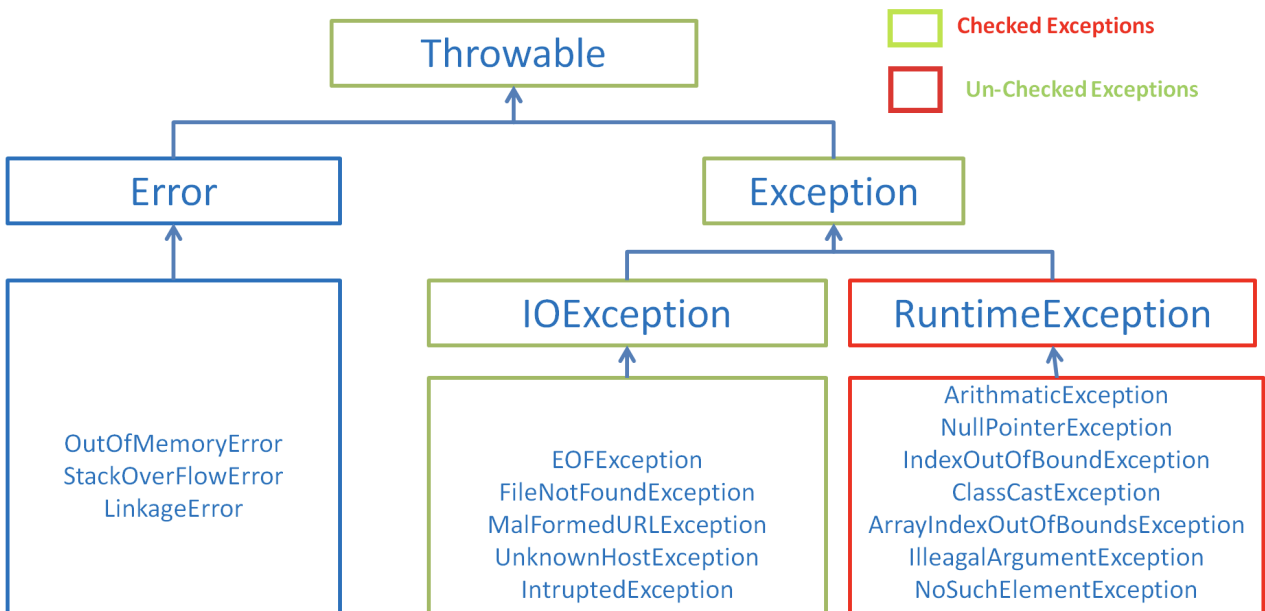
详见笔主的这篇文章: <https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/basic/Arrays,CollectionsCommonMethods.md>

### 3.2. 异常

#### 3.2.1. Java 异常类层次结构图



图片来自: <https://simplesnippets.tech/exception-handling-in-java-part-1/>



在 Java 中, 所有的异常都有一个共同的祖先 `java.lang` 包中的 **Throwable** 类。

**Throwable**: 有两个重要的子类: **Exception** (异常) 和 **Error** (错误), 二者都是 Java 异常处理的重要子类, 各自都包含大量子类。

**Error (错误)**: 是程序无法处理的错误, 表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关, 而表示代码运行时 JVM (Java 虚拟机) 出现的问题。例如, Java 虚拟机运行错误 (`Virtual MachineError`), 当 JVM 不再有继续执行操作所需的内存资源时, 将出现 `OutOfMemoryError`。这些异常发生时, Java 虚拟机 (JVM) 一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时, 如 Java 虚拟机运行错误 (`Virtual MachineError`)、类定义错误 (`NoClassDefFoundError`) 等。这些错误是不可查的, 因为它们在应用程序的控制和处理能力之外, 而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说, 即使确实发生了错误, 本质上也不应该试图去处理它所引起的异常状况。在 Java 中, 错误通过 **Error** 的子类描述。

**Exception (异常)**: 是程序本身可以处理的异常。Exception 类有一个重要的子类 **RuntimeException**。RuntimeException 异常由 Java 虚拟机抛出。**NullPointerException** (要访问的变量没有引用任何对象时, 抛出该异常)、**ArithmeticException** (算术运算异常, 一个整数除以 0 时, 抛出该异常) 和 **ArrayIndexOutOfBoundsException** (下标越界异常)。

注意: 异常和错误的区别: 异常能被程序本身处理, 错误是无法处理。

### 3.2.2. Throwable 类常用方法

- **public String getMessage()**: 返回异常发生时的简要描述
- **public String toString()**: 返回异常发生时的详细信息
- **public String getLocalizedMessage()**: 返回异常对象的本地化信息。使用 Throwable 的子类覆盖这个方法, 可以生成本地化信息。如果子类没有覆盖该方法, 则该方法返回的信息与 `getMessage()` 返回的结果相同
- **public void printStackTrace()**: 在控制台上打印 Throwable 对象封装的异常信息

### 3.2.3. try-catch-finally

- **try 块**: 用于捕获异常。其后可接零个或多个 catch 块, 如果没有 catch 块, 则必须跟一个 finally 块。
- **catch 块**: 用于处理 try 捕获到的异常。
- **finally 块**: 无论是否捕获或处理异常, finally 块里的语句都会被执行。当在 try 块或 catch 块中遇到 return 语句时, finally 语句块将在方法返回之前被执行。

在以下 4 种特殊情况下, finally 块不会被执行:

1. 在 finally 语句块第一行发生了异常。因为在其他行, finally 块还是会得到执行
2. 在前面的代码中用了 `System.exit(int)` 已退出程序。exit 是带参函数; 若该语句

- 在异常语句之后，`finally` 会执行
3. 程序所在的线程死亡。
  4. 关闭 CPU。

下面这部分内容来自 issue:<https://github.com/Snailclimb/JavaGuide/issues/190>。

**注意：** 当 `try` 语句和 `finally` 语句中都有 `return` 语句时，在方法返回之前，`finally` 语句的内容将被执行，并且 `finally` 语句的返回值将会覆盖原始的返回值。如下：

```
public class Test {
    public static int f(int value) {
        try {
            return value * value;
        } finally {
            if (value == 2) {
                return 0;
            }
        }
    }
}
```

如果调用 `f(2)`，返回值将是 0，因为 `finally` 语句的返回值覆盖了 `try` 语句块的返回值。

### 3.2.4. 使用 `try-with-resources` 来代替 `try-catch-finally`

《Effective Java》中明确指出：

面对必须要关闭的资源，我们总是应该优先使用 `try-with-resources` 而不是 `try-finally`。随之产生的代码更简短，更清晰，产生的异常对我们也更有用。`try-with-resources` 语句让我们更容易编写必须要关闭的资源的代码，若采用 `try-finally` 则几乎做不到这点。

Java 中类似于 `InputStream`、`OutputStream`、`Scanner`、`PrintWriter` 等的资源都需要我们调用 `close()` 方法来手动关闭，一般情况下我们都是通过 `try-catch-finally` 语句来实现这个需求，如下：

```
//读取文本文件的内容
Scanner scanner = null;
try {
    scanner = new Scanner(new File("D://read.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

```
    }  
}
```

使用Java 7之后的 `try-with-resources` 语句改造上面的代码:

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {  
    while (scanner.hasNext()) {  
        System.out.println(scanner.nextLine());  
    }  
} catch (FileNotFoundException fnfe) {  
    fnfe.printStackTrace();  
}
```

当然多个资源需要关闭的时候,使用 `try-with-resources` 实现起来也非常简单,如果你还是用`try-catch-finally`可能会带来很多问题。

通过使用分号分隔,可以在`try-with-resources`块中声明多个资源:

### 3.3. 多线程

#### 3.3.1. 简述线程、程序、进程的基本概念。以及他们之间关系是什么?

**线程**与进程相似,但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源,所以系统在产生一个线程,或是在各个线程之间作切换工作时,负担要比进程小得多,也正因为如此,线程也被称为轻量级进程。

**程序**是含有指令和数据的文件,被存储在磁盘或其他的数据存储设备中,也就是说程序是静态的代码。

**进程**是程序的一次执行过程,是系统运行程序的基本单位,因此进程是动态的。系统运行一个程序即是一个进程从创建,运行到消亡的过程。简单来说,一个进程就是一个执行中的程序,它在计算机中一个指令接着一个指令地执行着,同时,每个进程还占有某些系统资源如 CPU 时间,内存空间,文件,输入输出设备的使用权等等。换句话说,当程序在执行时,将会被操作系统载入内存中。线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的,而各线程则不一定,因为同一进程中的线程极有可能相互影响。从另一角度来说,进程属于操作系统的范畴,主要是同一段时间内,可以同时执行一个以上的程序,而线程则是在同一程序内几乎同时执行一个以上的程序段。

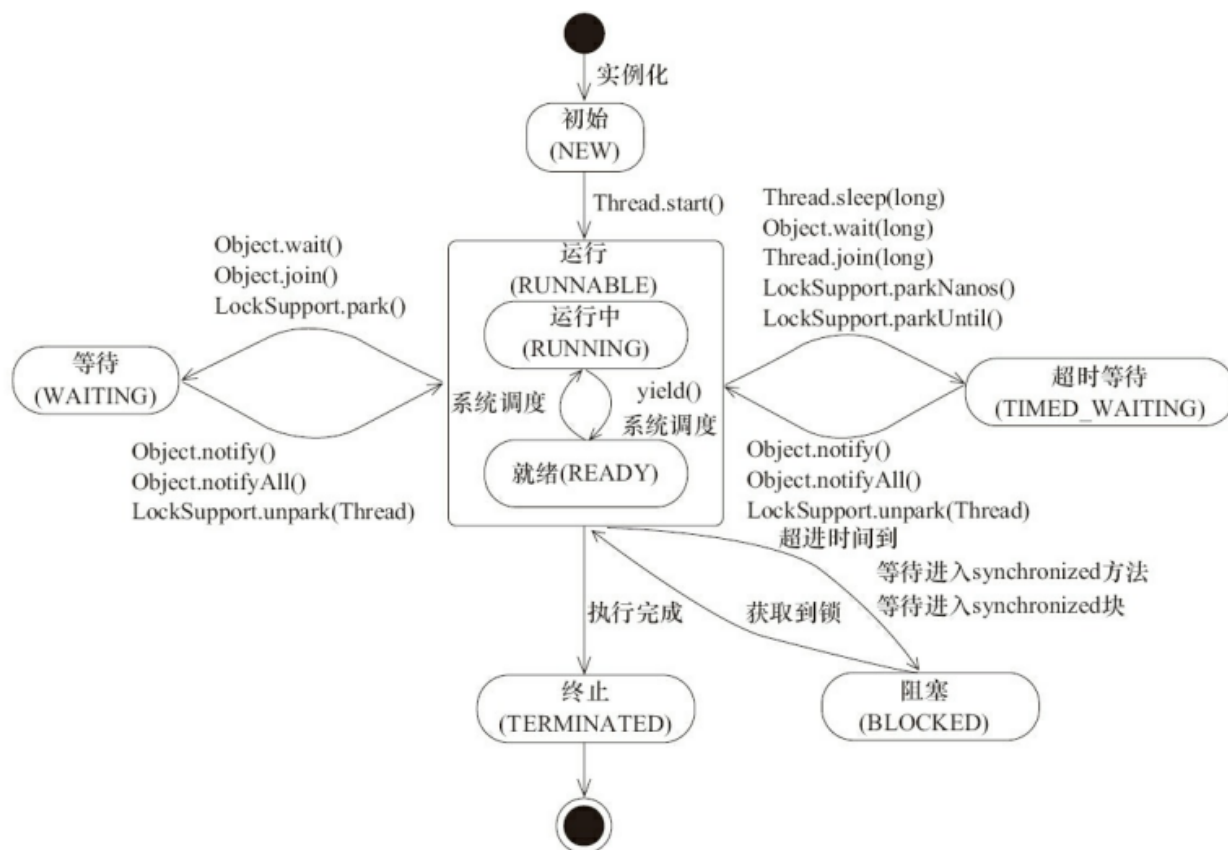
#### 3.3.2. 线程有哪些基本状态?

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态(图源《Java 并发编程艺术》4.1.4 节)。



状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：

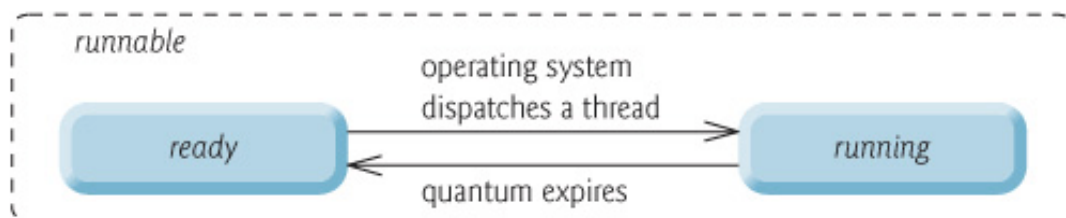


由上图可以看出：

线程创建之后它将处于 **NEW（新建）** 状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY（可运行）** 状态。可运行状态的线程获得了 cpu 时间片（timeslice）后就处于 **RUNNING（运行）** 状态。

操作系统隐藏 Java 虚拟机 (JVM) 中的 READY 和 RUNNING 状态，它只能看到 RUNNABLE 状态（图源：[HowToDoInJava: Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 RUNNABLE（运行中）状态。





当线程执行 `wait()` 方法之后，线程进入 **WAITING（等待）** 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 **TIME\_WAITING(超时等待)** 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 **TIMED WAITING** 状态。当超时时间到达后 Java 线程将会返回到 **RUNNABLE** 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 **BLOCKED（阻塞）** 状态。线程在执行 **Runnable** 的 `run()` 方法之后将会进入到 **TERMINATED（终止）** 状态。

## 3.4. 文件与 I/O 流

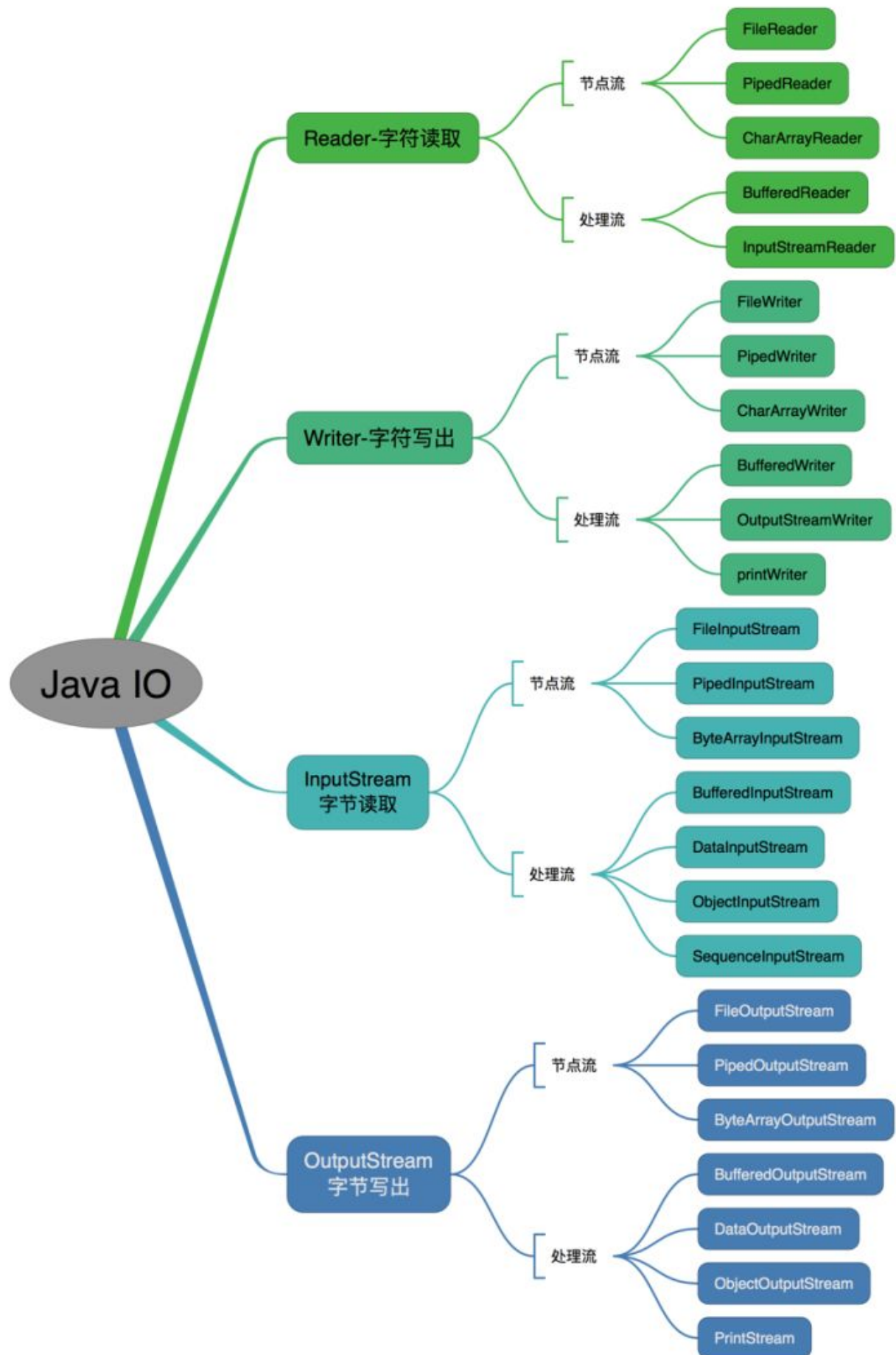
### 3.4.1. Java 中 IO 流分为几种？

- 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。

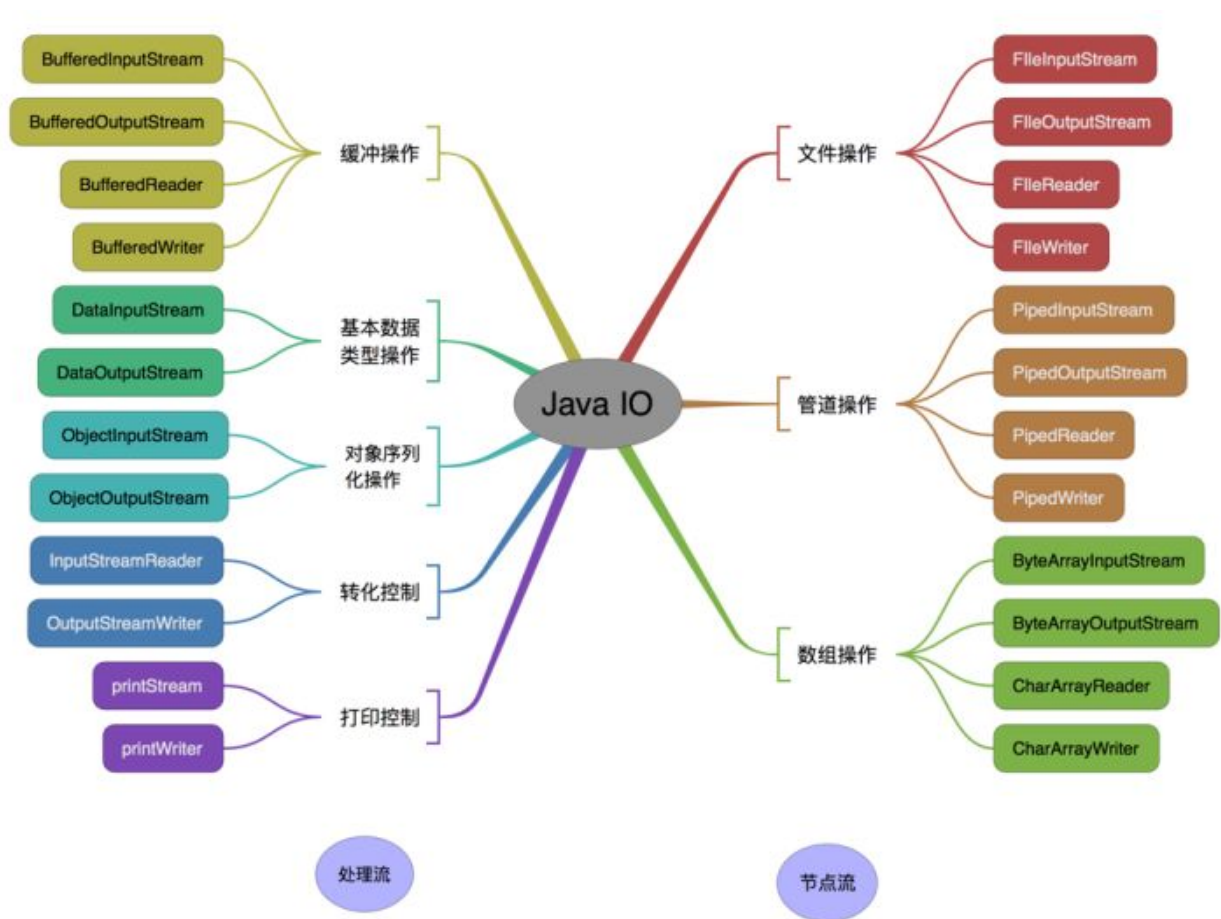
Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java Io 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- **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：



### 3.4.1.1. 既然有了字节流,为什么还要有字符流?

问题本质想问：不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？

回答：字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就容易出现乱码问题。所以，I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

### 3.4.1.2. BIO,NIO,AIO 有什么区别？

- BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- NIO (Non-blocking/New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 `java.nio` 包，提供了 `Channel`, `Selector`, `Buffer` 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 `Socket` 和 `ServerSocket` 相对应的 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相

反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

## 4. 参考

- <https://stackoverflow.com/questions/1906445/what-is-the-difference-between-jdk-and-jre>
- <https://www.educba.com/oracle-vs-openjdk/>
- <https://stackoverflow.com/questions/22358071/differences-between-oracle-jdk-and-openjdk?answertab=active#tab-top>

## 5. 公众号

如果大家想要实时关注我更新的文章以及分享的干货的话，可以关注我的公众号。

《Java 面试突击》：由本文档衍生的专为面试而生的《Java 面试突击》V2.0 PDF 版本公众号后台回复 "Java 面试突击" 即可免费领取！

**Java 工程师必备学习资源：**一些 Java 工程师常用学习资源公众号后台回复关键字“1”即可免费无套路获取。

