



疯程

Java程序员的 基本修养

李刚 编著

疯程源自梦想

技术成就辉煌

疯程源自梦想
技术成就辉煌



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

编程

Java程序员的 基本修养

李刚 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书归纳了 Java 学习者、工作者在学习、工作过程中最欠缺的技术短板，本书把 Java 编程中的重点、要点、难点、常见陷阱收集在一起，旨在帮助读者重点突破这些看似“司空见惯”的基本功。

本书知识主要分为四个部分，第一部分主要介绍 Java 程序的内存管理，这部分是大多 Java 程序员最容易忽略的地方——因为 Java 不像 C，而且 Java 提供了垃圾回收机制，因此导致许多 Java 程序员对内存管理重视不够；第二部分主要介绍了 Java 编程过程中各种常见的陷阱，这些陷阱有些来自于李刚老师早年痛苦的经历，有些来自于他的众多学子的痛苦经历，都是 Java 程序员在编程过程中的“前车之鉴”，希望读者能引以为戒；第三部分主要介绍常用数据结构的 Java 实现，这部分内容也是大多 Java 程序员重视不够的地方——因为许多初级程序员往往会感觉：数据结构对实际开发帮助并不大，但实际上，我们每天开发都会使用数据结构，只是经常利用别人的实现而已；第四部分主要介绍 Java 程序开发的方法、经验等，它们是李刚老师多年的实际开发经验、培训经验的总结，更符合初学者的习惯，更能满足初学者的需要，因此掌握这些开发方法、经验可以更有效地进行开发。

本书提供了配套的网站：<http://www.crazyit.org>，读者在阅读该书过程中遇到任何技术问题都可登录该站点与李刚老师交流，也可与疯狂 Java 图书庞大的读者群交流。

本书不是一本包含所有技术细节的手册，而是承载了无数过来人的谆谆教导，书中内容为有一定的 Java 基础的读者而编写，尤其适合于有一到两年的 Java 学习经验的读者和参加工作不久的初级 Java 程序员，帮助他们突破技术基本功的瓶颈。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目 (CIP) 数据

疯狂 Java 程序员的基本修养 / 李刚编著. —北京：电子工业出版社，2013.1

ISBN 978-7-121-19232-6

I. ①疯… II. ①李… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2012) 第 297897 号

策划编辑：张月萍

责任编辑：葛 娜

印 刷：北京东光印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：30.25 字数：771 千字 彩插：1

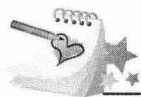
印 次：2013 年 1 月第 1 次印刷

印 数：4000 册 定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。



如何学习 Java

——谨以此文献给打算以编程为职业、并愿意为之疯狂的人

经常看到有些学生、求职者捧着一本类似 JBuilder 入门、Eclipse 指南之类的图书学习 Java，当他们学会了在这些工具中拖出窗体、安装按钮之后，就觉得自己掌握，甚至精通了 Java；又或是找来一本类似 JSP 动态网站编程之类的图书，学会使用 JSP 脚本编写一些页面后，就自我感觉掌握了 Java 开发。

还有一些学生、求职者听说 J2EE、Spring 或 EJB 很有前途，于是立即跑到书店或图书馆找来一本相关图书。希望立即学会它们，然后进入软件开发业、大显身手。

还有一些学生、求职者非常希望找到一本既速成、又大而全的图书，比如突击 J2EE 开发、一本书精通 J2EE 之类的图书（包括笔者曾出版的《轻量级 J2EE 企业应用实战》一书，据说销量不错），希望这样一本图书就可以打通自己的“任督二脉”，一跃成为 J2EE 开发高手。

也有些学生、求职者非常喜欢 J2EE 项目实战、项目大全之类的图书，他们的想法很单纯：我按照书上介绍，按图索骥、依葫芦画瓢，应该很快就可学会 J2EE，很快就能成为一个受人羡慕的 J2EE 程序员了。

.....

凡此种种，不一而足。但最后的结果往往是失败，因为这种学习没有积累、没有根基，学习过程中困难重重，每天都被一些相同、类似的问题所困扰，起初热情十足，经常上论坛询问，按别人的说法解决问题之后很高兴，既不知道为什么错？也不知道为什么对？只是盲目地抄袭别人的说法。最后的结果有两种：

① 久而久之，热情丧失，最后放弃学习。

② 大部分常见问题都问遍了，最后也可以从事一些重复性开发，但一旦遇到新问题，又将束手无策。

第二种情形在普通程序员中占了极大的比例，笔者多次听到、看到（在网络上）有些程序员抱怨：我做了 2 年多 Java 程序员了，工资还是 3000 多点。偶尔笔者会与他们聊聊工作相关内容，他们会告诉笔者：我也用 Spring 了啊，我也用 EJB 了啊……他们感到非常不平衡，为什么我的工资这么低？其实笔者很想告诉他们：你们太浮躁了！你们确实是用了 Spring、Hibernate 又或是 EJB，但你们未想过为什么要用这些技术？用这些技术有什么好处？如果不用这些技术行不行？

很多时候，我们的程序员把 Java 当成一种脚本，而不是一门面向对象的语言。他们习惯了在 JSP 脚本中使用 Java，但从不去想 JSP 如何运行，Web 服务器里的网络通信、多线程机制，为何一个 JSP 页面能同时向多个请求者提供服务？更不会想如何开发 Web 服务器；他们像代码机器一样编写 Spring Bean 代码，但从不去理解 Spring 容器的作用，更不会想如何开发 Spring 容器。

有时候，笔者的学生在编写五子棋、梭哈等作业感到困难时，会向他们的大学师兄、朋友求救，这些程序员告诉他：不用写了，网上有下载的！听到这样回答，笔者不禁感到哑然：网上还有 Windows 下载呢！网上下载和自己编写是两码事。偶尔，笔者会怀念以前黑色屏幕、

绿荧荧字符时代，那时候程序员很单纯：当我们想偷懒时，习惯思维是写一个小工具；现在程序员很聪明：当他们想偷懒时，习惯思维是从网上下一个小工具。但是，谁更幸福？

当笔者的学生把他们完成的小作业放上互联网之后，然后就有许多人称他们为“高手”！这个称呼却让他们万分惭愧；惭愧之余，他们也感到万分欣喜，非常有成就感，这就是编程的快乐。编程的过程，与寻宝的过程完全一样：历经辛苦，终于找到心中的梦想，这是何等的快乐？

如果真的打算将编程当成职业，那就不应该如此浮躁，而是应该扎扎实实先学好 Java 语言，然后按 Java 本身的学习规律，踏踏实实一步一个脚印地学习，把基本功练扎实了才可获得更大的成功。

实际情况是，有多少程序员真正掌握了 Java 的面向对象？真正掌握了 Java 的多线程、网络通信、反射等内容？有多少 Java 程序员真正理解了类初始化时内存运行过程？又有多少程序员理解 Java 对象从创建到消失的全部细节？有几个程序员真正独立地编写过五子棋、梭哈、桌面弹球这种小游戏？又有几个 Java 程序员敢说：我可以开发 Struts？我可以开发 Spring？我可以开发 Tomcat？很多人又会说：这些都是许多人开发出来的！实际情况是：许多开源框架的核心最初完全是由一个人开发的。现在这些优秀程序已经出来了！你，是否深入研究过它们，是否深入掌握了它们？

如果要真正掌握 Java，包括后期的 Java EE 相关技术（例如 Struts、Spring、Hibernate 和 EJB 等），一定要记住笔者的话：绝不要从 IDE（如 JBuilder、Eclipse 和 NetBeans）工具开始学习！IDE 工具的功能很强大，初学者学起来也很容易上手，但也非常危险：因为 IDE 工具已经为我们做了许多事情，而软件开发者要全部了解软件开发的全部步骤。



2012年12月1日



前言

Java 语言拥有的开发人群越来越大，大量程序员已经进入或正打算进入 Java 编程领域。这当然和 Java 语言本身的优秀不无关系，却也和 Java 编程入门简单有关。一个毫无编程基础的初学者，只要有点数据库和 SQL 基础，大概花不到一个月时间就可以学会编写 JSP 页面，说不定就可以找到一份 Java 编程的工作了。如果他肯再多下点功夫，学习一下编写 Struts Action 类、配置 Action，编写 Spring Bean 类、配置 Bean，他甚至可能自我感觉很不错了。

问题是：这种“快餐式”的程序员、“突击式”的程序员真的满足要求吗？如果仅仅满足于这些简单的、重复式开发，他们也许没有太多的问题，但他们可能很少有突破的机会。究其原因，很大程度上是因为他们的基本修养不够扎实。对他们而言，与其说 Java 是一种面向对象的语言，不如说更像一种脚本；他们从源代码层次来看程序运行（甚至只会从 Eclipse 等集成开发环境中看程序运行），完全无法从底层内存分配的角度来看程序运行；他们天天在用 Java 类库、用 Struts、用 Spring，但对这些东西的实现知之甚少——这又如何突破自己、获得更好的提高呢？

鉴于此种现状，“疯狂软件教育中心”训练过程中除了采用大量的实际项目来驱动教学之外，往往会花时间、精力来培养学员的基本修养。比如讲授 Spring 框架，如果只关注编写 Bean 类、配置 Bean，一天就足够了。而笔者往往会深入 Spring 框架的底层实现，带领学生从工厂模式、策略模式、门面模式、代理模式、命令模式的角度来深度分析 Spring 框架实现，然后进行对比，总结 Spring 框架的优势到底在哪里？不使用 Spring 框架是否有替换解决方案？进而感受设计模式对实际开发的帮助。

上面这些内容，看似“高深”，但其实质依然离不开 Java 编程的基本功。完全可以这样说：一旦读者真正把基本功打扎实了，将可以看得更高、更透彻。

在这样的背景下，笔者想将自己多年的一些经验、总结通过本书与大家分享，希望把自己多年积累的经验、心得表达出来；把自己走过的弯路“标”出来，让后来者尽量少走弯路。

本书内容



本书第一部分主要介绍 Java 内存管理相关方面的知识，内存管理既是 Java 程序员容易忽视的地方，又是 Java 编程的重点。实际上，许多有一定编程经验的 Java 开发者，自然而然就会关心垃圾回收、内存管理、性能优化相关内容。无论学习哪种语言，如果能真正从程序运行的底层机制、内存分配细节、内存回收细节把握程序执行过程，这样才能有豁然开朗的感觉，本书第一部分正是旨在帮助大家更好地掌握 Java 内存管理相关知识。

本书第三部分所介绍的常见数据结构、排序算法的 Java 实现，则是笔者一直想介绍的内容——也许你初涉编程时感受不到这些经典算法的用途，因为你可以直接利用别人的实现；但如果你希望突破自己，上升到另外一个高度时，你就不可避免地需要自己开发类库，而不是总使用别人的类库，那这些经典算法的作用就显现出来了。

本书第二部分和第四部分则主要来自于参加“疯狂软件教育中心”的学生，正如每个动手

编程的初学者，他们都曾经遭遇过各种各样的陷阱，笔者总是提醒他们应该将这些陷阱收集起来，以免再次陷进去。本书第二部分收集了 Java 编程中各种常见的陷阱；第四部分的内容则解决了他们进入实际开发之前的困扰，包括程序开发的基本方法，有效进行程序调试的方法，如何看待、使用 IDE 工具，软件测试等相关内容。

本书源代码的下载地址为：<http://www.broadview.com.cn/19232>。

本书写给谁看



如果你想从零开始学习 Java 编程，本书不适合你。如果你已经学会了 Java 基本语法，动手编程却感到困难重重，或者你已经是一个 Java 程序员了，实际开发中却感觉力不从心，本书将非常适合你。本书会帮助你找出自己的技术短板，提升 Java 编程的基本修养。

2012-12-1

目 录

CONTENTS

第 1 章 数组及其内存管理	1	3.2 Map 和 List	85
1.1 数组初始化	2	3.2.1 Map 的 values()方法	85
1.1.1 Java 数组是静态的	2	3.2.2 Map 和 List 的关系	91
1.1.2 数组一定要初始化吗	5	3.3 ArrayList 和 LinkedList	92
1.1.3 基本类型数组的初始化	7	3.3.1 Vector 和 ArrayList 的区别	94
1.1.4 引用类型数组的初始化	9	3.3.2 ArrayList 和 LinkedList 的 实现差异	97
1.2 使用数组	12	3.3.3 ArrayList 和 LinkedList 的 性能分析及适用场景	101
1.2.1 数组元素就是变量	12	3.4 Iterator 迭代器	101
1.2.2 没有多维数组	14	3.4.1 Iterator 实现类与迭代器模式	102
1.3 本章小结	20	3.4.2 迭代时删除指定元素	103
第 2 章 对象及其内存管理	21	3.5 本章小结	106
2.1 实例变量和类变量	22	第 4 章 Java 的内存回收	107
2.1.1 实例变量和类变量的属性	23	4.1 Java 引用的种类	108
2.1.2 实例变量的初始化时机	26	4.1.1 对象在内存中的状态	108
2.1.3 类变量的初始化时机	30	4.1.2 强引用	111
2.2 父类构造器	32	4.1.3 软引用	111
2.2.1 隐式调用和显式调用	32	4.1.4 弱引用	114
2.2.2 访问子类对象的实例变量	34	4.1.5 虚引用	118
2.2.3 调用被子类重写的方法	37	4.2 Java 的内存泄漏	119
2.3 父子实例的内存控制	39	4.3 垃圾回收机制	123
2.3.1 继承成员变量和继承 方法的区別	39	4.3.1 垃圾回收的基本算法	123
2.3.2 内存中子类实例	42	4.3.2 堆内存的分代回收	125
2.3.3 父、子类的类变量	47	4.3.3 与垃圾回收相关的附加选项	127
2.4 final 修饰符	48	4.3.4 常见的垃圾回收器	127
2.4.1 final 修饰的变量	48	4.4 内存管理小技巧	131
2.4.2 执行“宏替换”的变量	53	4.4.1 尽量使用直接量	132
2.4.3 final 方法不能被重写	57	4.4.2 使用 StringBuilder 和 StringBuffer 进行字符串连接	132
2.4.4 内部类中的局部变量	59	4.4.3 尽早释放无用对象的引用	132
2.5 本章小结	62	4.4.4 尽量少用静态变量	133
第 3 章 常见 Java 集合的实现细节	63	4.4.5 避免在经常调用的方法、 循环中创建 Java 对象	133
3.1 Set 和 Map	64	4.4.6 缓存经常使用的对象	134
3.1.1 Set 和 Map 的关系	64	4.4.7 尽量不要使用 finalize 方法	134
3.1.2 HashMap 和 HashSet	69	4.4.8 考虑使用 SoftReference	135
3.1.3 TreeMap 和 TreeSet	79		

4.5	本章小结	135	6.5	for 循环的陷阱	185
第 5 章	表达式中的陷阱	136	6.5.1	分号惹的祸	185
5.1	关于字符串的陷阱	137	6.5.2	小心循环计数器的值	188
5.1.1	JVM 对字符串的处理	137	6.5.3	浮点数作循环计数器	188
5.1.2	不可变的字符串	140	6.6	foreach 循环的循环计数器	190
5.1.3	字符串比较	142	6.7	本章小结	192
5.2	表达式类型的陷阱	144	第 7 章	面向对象的陷阱	193
5.2.1	表达式类型的自动提升	144	7.1	instanceof 运算符的陷阱	194
5.2.2	复合赋值运算符的陷阱	145	7.2	构造器的陷阱	198
5.2.3	Java 7 新增的二进制整数	147	7.2.1	构造器之前的 void	198
5.3	输入法导致的陷阱	148	7.2.2	构造器创建对象吗	199
5.4	注释字符必须合法	149	7.2.3	无限递归的构造器	203
5.5	转义字符的陷阱	149	7.3	持有当前类的实例	205
5.5.1	慎用字符的 Unicode 转义形式	149	7.4	到底调用哪个重载的方法	206
5.5.2	中止行注释的转义字符	150	7.5	方法重写的陷阱	209
5.6	泛型可能引起的错误	151	7.5.1	重写 private 方法	209
5.6.1	原始类型变量的赋值	151	7.5.2	重写其他访问权限的方法	210
5.6.2	原始类型带来的擦除	153	7.6	非静态内部类的陷阱	211
5.6.3	创建泛型数组的陷阱	155	7.6.1	非静态内部类的构造器	211
5.7	正则表达式的陷阱	157	7.6.2	非静态内部类不能 拥有静态成员	213
5.8	多线程的陷阱	158	7.6.3	非静态内部类的子类	214
5.8.1	不要调用 run 方法	158	7.7	static 关键字	215
5.8.2	静态的同步方法	160	7.7.1	静态方法属于类	215
5.8.3	静态初始化块启动新线程 执行初始化	162	7.7.2	静态内部类的限制	217
5.8.4	注意多线程执行环境	167	7.8	native 方法的陷阱	217
5.9	本章小结	171	7.9	本章小结	219
第 6 章	流程控制的陷阱	172	第 8 章	异常处理的陷阱	220
6.1	switch 语句陷阱	173	8.1	正确关闭资源的方式	221
6.1.1	default 分支永远会执行吗	173	8.1.1	传统关闭资源的方式	221
6.1.2	break 的重要性	174	8.1.2	使用 Java 7 增强的 try 语句关闭资源	224
6.1.3	Java 7 增强的 switch 表达式	176	8.2	finally 块的陷阱	226
6.2	标签引起的陷阱	177	8.2.1	finally 的执行规则	226
6.3	if 语句的陷阱	178	8.2.2	finally 块和方法返回值	227
6.3.1	else 隐含的条件	178	8.3	catch 块的用法	229
6.3.2	小心空语句	181	8.3.1	catch 块的顺序	229
6.4	循环体的花括号	182	8.3.2	不要用 catch 代替流程控制	231
6.4.1	什么时候可以省略花括号	182	8.3.3	只有 catch 可能抛出的异常	232
6.4.2	省略花括号的危险	183	8.3.4	做点实际的修复	235

8.4 继承得到的异常	237	11.2.1 二叉树的定义和基本概念	301
8.5 Java 7 增强的 throw 语句	238	11.2.2 二叉树的基本操作	302
8.6 本章小结	240	11.2.3 二叉树的顺序存储	303
第 9 章 线性表	241	11.2.4 二叉树的二叉链表存储	306
9.1 线性表概述	242	11.2.5 二叉树的三叉链表存储	310
9.1.1 线性表的定义及逻辑结构	242	11.3 遍历二叉树	313
9.1.2 线性表的基本操作	243	11.3.1 先序遍历	314
9.2 顺序存储结构	243	11.3.2 中序遍历	314
9.3 链式存储结构	248	11.3.3 后序遍历	315
9.3.1 单链表上的基本运算	249	11.3.4 广度优先(按层)遍历	316
9.3.2 循环链表	255	11.4 转换方法	316
9.3.3 双向链表	256	11.4.1 森林、树和二叉树的转换	317
9.4 线性表的分析	262	11.4.2 树的链表存储	318
9.4.1 线性表的实现分析	262	11.5 哈夫曼树	318
9.4.2 线性表的功能	263	11.5.1 哈夫曼树的定义和 基本概念	319
9.5 本章小结	264	11.5.2 创建哈夫曼树	319
第 10 章 栈和队列	265	11.5.3 哈夫曼编码	322
10.1 栈	266	11.6 排序二叉树	323
10.1.1 栈的基本定义	266	11.7 红黑树	331
10.1.2 栈的常用操作	267	11.7.1 插入操作	332
10.1.3 栈的顺序存储结构及实现	267	11.7.2 删除操作	335
10.1.4 栈的链式存储结构及实现	272	11.8 本章小结	344
10.1.5 Java 集合中的栈	275	第 12 章 常用的内部排序	345
10.2 队列	275	12.1 排序的基本概念	346
10.2.1 队列的基本定义	275	12.1.1 排序概述	346
10.2.2 队列的常用操作	276	12.1.2 内部排序的分类	347
10.2.3 队列的顺序存储结构及实现	276	12.2 选择排序法	347
10.2.4 循环队列	280	12.2.1 直接选择排序	347
10.2.5 队列的链式存储结构及实现	284	12.2.2 堆排序	351
10.2.6 Java 集合中的队列	287	12.3 交换排序	356
10.3 双端队列	288	12.3.1 冒泡排序	356
10.4 本章小结	289	12.3.2 快速排序	358
第 11 章 树和二叉树	290	12.4 插入排序	360
11.1 树的概述	291	12.4.1 直接插入排序	360
11.1.1 树的定义和基本术语	291	12.4.2 折半插入排序	362
11.1.2 树的基本操作	292	12.4.3 Shell 排序	364
11.1.3 父节点表示法	293	12.5 归并排序	367
11.1.4 子节点链表示法	296	12.6 桶式排序	370
11.2 二叉树	301	12.7 基数排序	372
		12.8 本章小结	375

第 13 章 程序开发经验谈.....	376	第 15 章 IDE 工具心法谈.....	421
13.1 扎实的基本功.....	377	15.1 何时开始利用 IDE 工具.....	422
13.1.1 快速的输入能力.....	377	15.2 IDE 工具概述.....	423
13.1.2 编程实现能力.....	379	15.2.1 IDE 工具的基本功能.....	424
13.1.3 快速排错.....	379	15.2.2 常见的 Java IDE 工具.....	425
13.2 程序开发之前.....	380	15.3 项目管理.....	428
13.2.1 分析软件的组件模型.....	380	15.3.1 建立项目.....	428
13.2.2 建立软件的数据模型.....	383	15.3.2 自动编译.....	434
13.3 理清程序的实现流程.....	384	15.3.3 自动部署、运行.....	435
13.3.1 各组件如何通信.....	384	15.4 代码管理.....	436
13.3.2 人机交互的实现.....	386	15.4.1 向导式的代码生成.....	436
13.3.3 复杂算法的分析.....	388	15.4.2 代码生成器.....	438
13.4 编写开发文档.....	391	15.4.3 代码提示.....	439
13.4.1 绘制建模图、流程图.....	391	15.4.4 自动代码补齐.....	441
13.4.2 提供简要说明.....	393	15.4.5 实时错误提示.....	441
13.4.3 编写伪码实现.....	393	15.5 项目调试.....	442
13.5 编码实现和开发心态.....	394	15.5.1 设置断点.....	442
13.5.1 开发是复杂的.....	394	15.5.2 单步调试.....	444
13.5.2 开发过程是漫长的.....	394	15.5.3 步入、步出.....	445
13.6 本章小结.....	395	15.6 团队协作功能.....	446
第 14 章 程序调试经验谈.....	396	15.7 本章小结.....	450
14.1 程序的可调试性.....	397	第 16 章 软件测试经验谈.....	451
14.1.1 增加注释.....	397	16.1 软件测试概述.....	452
14.1.2 使用 log.....	397	16.1.1 软件测试的概念和目的.....	452
14.2 程序调试的基本方法.....	405	16.1.2 软件测试的分类.....	454
14.2.1 借助编译器的代码审查.....	405	16.1.3 开发活动和测试活动.....	454
14.2.2 跟踪程序执行流程.....	408	16.1.4 常见的 Bug 管理工具.....	455
14.2.3 断点调试.....	409	16.2 单元测试.....	456
14.2.4 隔离调试.....	411	16.2.1 单元测试概述.....	456
14.2.5 错误重现.....	412	16.2.2 单元测试的逻辑覆盖.....	458
14.3 记录常见错误.....	414	16.2.3 JUnit 介绍.....	461
14.3.1 常见异常可能的错误原因.....	414	16.2.4 JUnit 的用法.....	461
14.3.2 常见运行时异常 可能的错误原因.....	416	16.3 系统测试和自动化测试.....	467
14.4 程序调试的整体思路.....	417	16.3.1 系统测试概述.....	467
14.4.1 分段调试.....	418	16.3.2 自动化测试.....	468
14.4.2 分模块调试.....	419	16.3.3 常见的自动化测试工具.....	469
14.5 调试心态.....	419	16.4 性能测试.....	470
14.5.1 谁都会出错.....	420	16.4.1 性能测试概述.....	470
14.5.2 调试比写程序更费时.....	420	16.4.2 性能测试的相关概念.....	471
14.6 本章小结.....	420	16.4.3 常见的性能测试工具.....	472
		16.5 本章小结.....	472

第 1 章

数组及其内存管理

引言

一家国际著名软件企业的面试。

“你的简历我看了，你会使用 Java？”面试官面无表情地问道。

“是的。”参加面试的人，成竹在胸地回答。

“那好，你给我叙述一下，在 Java 中，声明并创建数组的过程中，内存是如何分配的？”

“.....”

“Java 数组的初始化一共有哪几种方式，你能说一说吗？”

“.....”

“你知道基本类型数组和引用类型数组之间，在初始化时的内存分配机制有什么区别吗？”

“.....”

过了一会儿，房间的门打开了，可怜的面试者，狼狈地走了出来。

离开的时候，他喃喃自语：“原来，小小的数组，也有这么多的知识。”

本章要点

- ✎ Java 数组的基本语法
- ✎ Java 数组的内存分配机制
- ✎ 初始化基本类型数组的内存分配
- ✎ 数组引用变量和数组对象
- ✎ 数组元素等同于变量
- ✎ Java 数组的静态特性
- ✎ 初始化 Java 数组的两种方式
- ✎ 初始化应用类型数组的内存分配
- ✎ 何时是数组引用变量，何时是数组对象
- ✎ 多维数组的内存分配

Java 数组并不是很难的知识，如果单从用法角度来看，数组的用法并不难，只是很多程序员虽然一直使用 Java 数组，但他们往往对 Java 数组的内存分配把握并不准确。本章正是为了弥补程序员的这部分基本功而做的深入探讨。

本章将会深入探讨 Java 数组的静态特征。在使用 Java 数组之前必须先对数组对象进行初始化。当数组的所有元素都被分配了合适的内存空间，并指定了初始值时，数组初始化完成，程序以后将不能重新改变数组对象在内存中的位置和大小。从用法角度来看，数组元素相当于普通变量，程序既可把数组元素的值赋给普通变量，也可把普通变量的值赋给数组元素。

本章还将深入分析多维数组的实质，深入讲解多维数组和一维数组之间的关联，并通过程序示范如何将一维数组扩展成多维数组。

1.1 数组初始化

数组是大多数编程语言都提供的一种复合结构，如果程序需要多个类型相同的变量时，就可以考虑定义一个数组。Java 语言的数组变量是引用类型的变量，因此具有 Java 引用变量的特性。

▶▶ 1.1.1 Java 数组是静态的

Java 语言是典型的静态语言，因此 Java 数组是静态的，即当数组被初始化之后，该数组所占的内存空间、数组长度都是不可变的。Java 程序中的数组必须经过初始化才可使用。所谓初始化，即创建实际的数组对象，也就是在内存中为数组对象分配内存空间，并为每个数组元素指定初始值。

数组的初始化有以下两种方式。

- ▶ 静态初始化：初始化时由程序员显式指定每个数组元素的初始值，由系统决定数组长度。
- ▶ 动态初始化：初始化时程序员只指定数组长度，由系统为数组元素分配初始值。

不管采用哪种方式初始化 Java 数组，一旦初始化完成，该数组的长度就不可改变，Java 语言允许通过数组的 `length` 属性来访问数组的长度。示例如下。

程序清单：codes\01\1.1\ArrayTest.java

```
public class ArrayTest
{
    public static void main(String[] args)
    {
        // 采用静态初始化方式初始化第一个数组
        String[] books = new String[]
        {
            "疯狂 Java 讲义",
            "轻量级 Java EE 企业应用实战",
            "疯狂 Ajax 讲义",
            "疯狂 XML 讲义"
        };
        // 采用静态初始化的简化形式初始化第二个数组
        String[] names =
```

```

{
    "孙悟空",
    "猪八戒",
    "白骨精"
};
// 采用动态初始化的语法初始化第三个数组
String[] strArr = new String[5];
// 访问三个数组的长度
System.out.println("第一个数组的长度: " + books.length);
System.out.println("第二个数组的长度: " + names.length);
System.out.println("第三个数组的长度: " + strArr.length);
}
}

```

上面程序中的粗体字代码声明并初始化了三个数组。这三个数组的长度将会始终不变，程序输出三个数组的长度依次为 4、3、5。

前面已经指出，Java 语言的数组变量是引用类型的变量，books、names、strArr 这三个变量，以及各自引用的数组在内存中的分配示意图如图 1.1 所示。

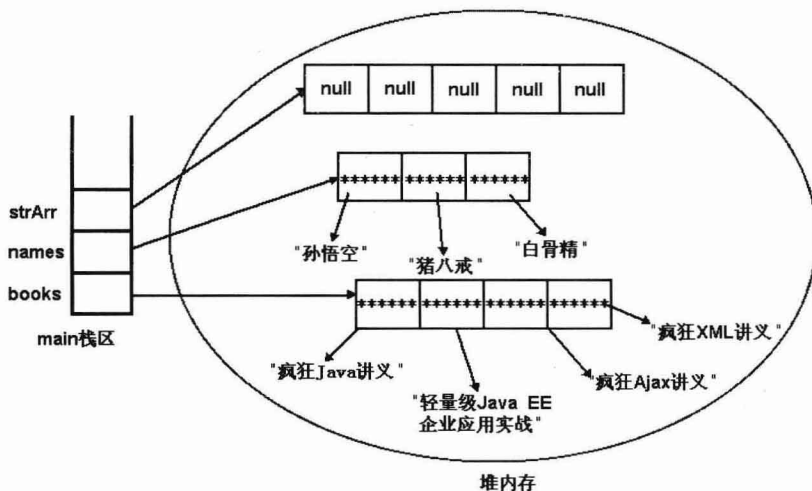


图 1.1 数组在内存中的分配示意图 1

从图 1.1 可以看出，对于静态初始化方式而言，程序员无须指定数组长度，指定该数组的数组元素，由系统来决定该数组的长度即可。例如 books 数组，为它指定了四个数组元素，它的长度就是 4；对于 names 数组，为它指定了三个元素，它的长度就是 3。

执行动态初始化时，程序员只需指定数组的长度，即为每个数组元素指定所需的内存空间，系统将负责为这些数组元素分配初始值。指定初始值时，系统将按如下规则分配初始值。

- 数组元素的类型是基本类型中的整数类型 (byte、short、int 和 long)，则数组元素的值是 0。
- 数组元素的类型是基本类型中的浮点类型 (float、double)，则数组元素的值是 0.0。
- 数组元素的类型是基本类型中的字符类型 (char)，则数组元素的值是 '\u0000'。
- 数组元素的类型是基本类型中的布尔类型 (boolean)，则数组元素的值是 false。
- 数组元素的类型是引用类型 (类、接口和数组)，则数组元素的值是 null。

注意：

不要同时使用静态初始化和动态初始化方式。也就是说，不要在进行数组初始化时，既指定数组的长度，也为每个数组元素分配初始值。



Java 数组是静态的，一旦数组初始化完成，数组元素的内存空间分配即结束，程序只能改变数组元素的值，而无法改变数组的长度。

需要指出的是，Java 的数组变量是一种引用类型的变量，数组变量并不是数组本身，它只是指向堆内存中的数组对象。因此，可以改变一个数组变量所引用的数组，这样可以造成数组长度可变的假象。假设，在上面程序的后面增加如下几行。

程序清单：codes\01\1.1\ArrayTest2.java

```
// 让 books 数组变量、strArr 数组变量指向 names 所引用的数组
books = names;
strArr = names;
System.out.println("-----");
System.out.println("books 数组的长度：" + books.length);
System.out.println("strArr 数组的长度：" + strArr.length);
// 改变 books 数组变量所引用的数组的第二个元素值
books[1] = "唐僧";
System.out.println("names 数组的第二个元素是：" + books[1]);
```

上面程序中粗体字代码将让 books 数组变量、strArr 数组变量都指向 names 数组变量所引用的数组，这样做的结果就是 books、strArr、names 这三个变量引用同一个数组对象。此时，三个引用变量和数组对象在内存中的分配示意图如图 1.2 所示。

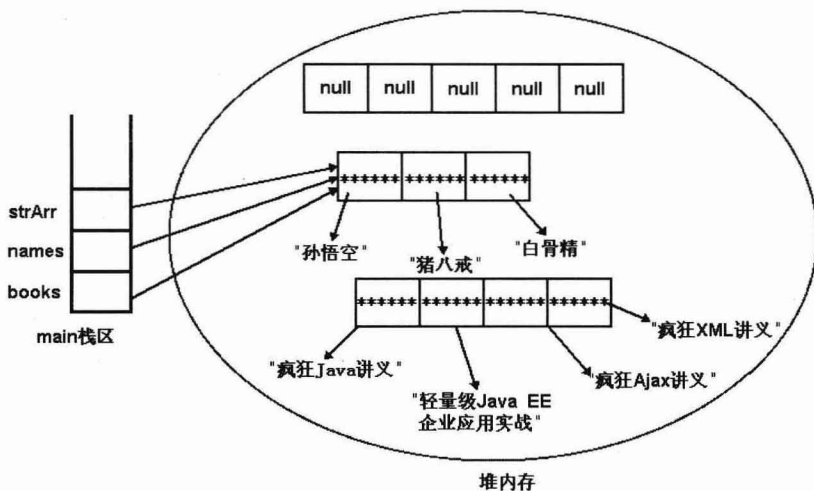


图 1.2 数组在内存中的分配示意图 2

从图 1.2 可以看出，此时 strArr、names 和 books 数组变量实际上引用了同一个数组对象。因此，当访问 books 数组、strArr 数组的长度时，将看到输出 3。这很容易造成一个假象：books 数组的长度从 4 变成了 3。实际上，数组对象本身的长度并没有发生改变，只是 books 数组变量发生了改变。books 数组变量原本指向图 1.2 下面的数组，当执行了 books = names; 语句之后，

books 数组将改为指向图 1.2 中间的数组，而原来 books 变量所引用的数组的长度依然是 4。

从图 1.2 还可以看出，原来 books 变量所引用的数组的长度依然是 4，但不再有任何引用变量引用该数组，因此它将会变成垃圾，等着垃圾回收机制来回收。此时，程序使用 books、names 和 strArr 这三个变量时，将会访问同一个数组对象，因此把 books 数组的第二个元素赋值为“唐僧”时，names 数组的第二个元素的值也会随之改变。

与 Java 这种静态语言不同的是，JavaScript 这种动态语言的数组长度是可以动态改变的，示例如下。

程序清单：codes\01\1.1\ArrTest.html

```
<script type="text/javascript">
    var arr = [];
    document.writeln("arr 的长度是: " + arr.length + "<br/>");
    // 为 arr 数组的两个数组元素赋值
    arr[2] = 6;
    arr[4] = "孙悟空";
    // 再次访问 arr 数组的长度
    document.writeln("arr 的长度是: " + arr.length + "<br/>");
</script>
```

上面是一个简单的 JavaScript 程序。它先定义了一个名为 arr 的空数组，因为它不包含任何数组元素，所以它的长度是 0。接着，为 arr 数组的第三个、第五个元素赋值，该数组的长度也自动变为 5。这就是 JavaScript 里动态数组和 Java 里静态数组的区别。

1.1.2 数组一定要初始化吗

阅读过疯狂 Java 体系的《疯狂 Java 讲义》的读者一定还记得：在使用 Java 数组之前必须先初始化数组（即在使用数组之前，必须先创建数组）。实际上，如果真正掌握了 Java 数组在内存中的分配机制，那么完全可以换一个方式来初始化数组。

始终记住：Java 的数组变量只是引用类型的变量，它并不是数组对象本身，只要让数组变量指向有效的数组对象，程序中即可使用该数组变量。示例如下。

程序清单：codes\01\1.1\ArrayTest3.java

```
public class ArrayTest3
{
    public static void main(String[] args)
    {
        // 定义并初始化 nums 数组
        int[] nums = new int[]{3, 5, 20, 12};
        // 定义一个 prices 数组变量
        int[] prices;
        // 让 prices 数组指向 nums 所引用的数组
        prices = nums;
        for (int i = 0; i < prices.length; i++ )
        {
            System.out.println(prices[i]);
        }
        // 将 prices 数组的第 3 个元素赋值为 34
        prices[2] = 34;
        // 访问 nums 数组的第 3 个元素，将看到输出 34
    }
}
```



```

System.out.println("nums 数组的第 3 个元素的值是: " + nums[2]);
    }
}
    
```

从上面粗体字代码可以看出，程序定义了 prices 数组之后，并未对 prices 数组进行初始化。当执行 int[] prices;之后，数组的内存分配示意图如图 1.3 所示。

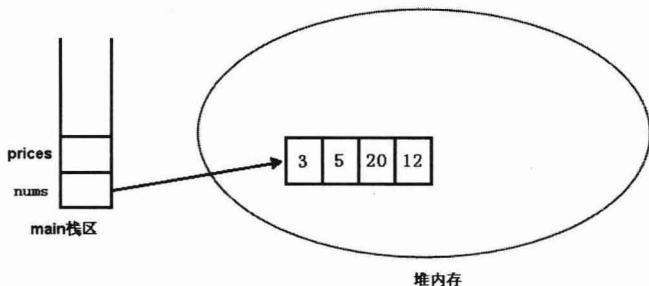


图 1.3 数组在内存中的分配示意图 3

从图 1.3 可以看出，此时的 prices 数组变量还未指向任何有效的内存，未指向任何数组对象，因此程序还不可使用 prices 数组变量。

当程序执行 prices = nums;之后，prices 变量将指向 nums 变量所引用的数组，此时 prices 变量和 nums 变量引用同一个数组对象。执行这条语句之后，prices 变量已经指向有效的内存及一个长度为 4 的数组对象，因此程序完全可以正常使用 prices 变量了。

注意：

在使用 Java 数组之前必须先进行初始化！可是现在 prices 变量却无须初始化，这不是互相矛盾吗？其实一点都不矛盾。关键是大部分时候，我们把数组变量和数组对象搞混了，数组变量只是一个引用变量（有点类似于 C 语言里的指针）；而数组对象就是保存在堆内存中的连续内存空间。对数组执行初始化，其实并不是对数组变量执行初始化，而是在堆内存中创建数组对象——也就是为该数组对象分配一块连续的内存空间，这块连续的内存空间的长度就是数组的长度。虽然上面程序中的 prices 变量看似没有经过初始化，但执行 prices = nums; 就会让 prices 变量直接指向一个已经存在的数组，因此 prices 变量即可使用。



对于数组变量来说，它并不需要进行所谓的初始化，只要让数组变量指向一个有效的数组对象，程序即可正常使用该数组变量。



提示：

Java 程序中的引用变量并不需要经过所谓的初始化操作，需要进行初始化的是引用变量所引用的对象。比如，数组变量不需要进行初始化操作，而数组对象本身需要进行初始化；对象的引用变量也不需要进行初始化，而对对象本身才需要进行初始化。需要指出的是，Java 的局部变量必须由程序员赋初始值，因此如果定义了局部变量的数组变量，程序必须对局部的数据变量进行赋值，即使将它赋为 null 也行。