

从语法、程序设计和架构、工具和框架、编码风格、编程思想
5个方面深入探讨编写高质量Java代码的技巧、禁忌和最佳实践



秦小波 著

Writing Solid Java Code: 151 Suggestions to Improve Your Java Programs

编写高质量代码 改善Java程序的151个建议

策划
PDG



机械工业出版社
China Machine Press



51CTO.COM
技术成就梦想

联合策划

是一本关于Java编码最佳实践的集大成之作，也是一本能指导Java程序员编写出高质量代码的指点迷津之作。全书从Java语法、程序的架构和设计、编码规范和编程习惯等方面为广大的Java程序员们总结出了151条极富借鉴意义的建议，这些建议都在实践中被证明是解决Java编码中疑难问题的最佳实践。如果能掌握本书中的内容，不仅能加深对Java语言的理解，还能提升程序架构和设计方面的能力，同时还能规范我们的开发行为和习惯，让我们成为优秀的程序员，编写出更高质量的代码。

—— 51CTO (www.51cto.com, 中国领先的IT技术网站)

Coding为我们创造了一个丰富多彩的虚拟世界，本书是作者秦小波多年工作经验的总结，也许能为所有从事Java软件开发的同行们提供有益的参考。本书除了与大家分享了151条编写高质量Java代码的宝贵建议之外，它还在试图告诉创造虚拟世界的程序员们，Coding不仅仅是使用智慧，更多的是对它的热爱，唯有热爱和用心才能编写出高质量的代码，才能开发出优秀应用。

—— 李海宁 交通银行软件开发中心总经理

本书有几个很突出的特点：第一，内容实用，它没有去讲解基本语法，那是大学教科书的内容，本书着重探讨了如何才能将Java代码编写得更高效、更优雅；第二，涉及面广，从语法到编程规范和编程思想，从JDK API到开源框架，都有所涉猎；第三，注重实战，书中的所有建议都是从实践中总结出来的，都是真实场景的重现，不是纸上谈兵。除此之外，本书内容精炼、语言幽默、通俗流畅，阅读体验十分好。对于正在进阶修炼途中的Java程序员来说，本书是提高开发技能和自身修养的好帮手。

—— 计文柯 资深Java技术专家，
著有畅销书《Spring技术内幕：深入解析Spring架构与设计原理》

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

實
戰



Writing Solid Java Code: 151 Suggestions to Improve Your Java Program

编写高质量代码 改善Java程序的151个建议

秦小波 著



机械工业出版社
China Machine Press

在通往“Java技术殿堂”的路上，本书将为你指点迷津！内容全部由Java编码的最佳实践组成，从语法、程序设计和架构、工具和框架、编码风格和编程思想等五大方面，对Java程序员遇到的各种棘手的疑难问题给出了经验性的解决方案，为Java程序员如何编写高质量的Java代码提出了151条极为宝贵的建议。对于每一个问题，不仅以建议的方式从正反两面给出了被实践证明为十分优秀的解决方案和非常糟糕的解决方案，而且还分析了问题产生的根源，犹如醍醐灌顶，让人豁然开朗。

全书一共12章，第1~3章针对Java语法本身提出了51条建议，例如覆写变长方法时应该注意哪些事项、final修饰的常量不要在运行期修改、匿名类的构造函数特殊在什么地方等；第4~9章重点针对JDK API的使用提出了80条建议，例如字符串的拼接方法该如何选择、枚举使用时有哪些注意事项、出现NullPointerException该如何处理、泛型的多重界限该如何使用、多线程编程如何预防死锁，等等；第10~12章针对程序性能、开源的工具和框架、编码风格和编程思想等方面提出了20条建议。

本书针对每个问题所设计的应用场景都非常典型，给出的建议也都与实践紧密结合。书中的每一条建议都可能在你的下一行代码、下一个应用或下一个项目中崭露头角，建议你将此书搁置在手边，随时查阅，一定能使你的学习和开发工作事半功倍。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

图书在版编目（CIP）数据

编写高质量代码：改善Java程序的151个建议 / 秦小波著. —北京：机械工业出版社，
2011.11

ISBN 978-7-111-36259-3

I. 编… II. 秦… III. JAVA 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2011）第 217329 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：杨绣国 陈佳媛

北京京北印刷有限公司印刷

2012 年 1 月第 1 版第 1 次印刷

186mm×240mm • 20 印张

标准书号：ISBN 978-7-111-36259-3

定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991; 88361066

购书热线：(010) 68326294; 88379649; 68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com



从决定撰写本书到完稿历时 9 个月，期间曾经遇到过种种困难和挫折，但这个过程让我明白了坚持的意义，明白了“行百里者半九十”的寓意——坚持下去，终于到了写前言的时刻。

为什么写这本书

从第一次敲出“Hello World”到现在已经有 15 年时间了，在这 15 年里，我当过程序员和架构师，也担任过项目经理和技术顾问——基本上与技术沾边的事情都做过。从第一次接触 Java 到现在，已经有 11 年 4 个月了，在这些年里，我对 Java 可谓是情有独钟，对其编程思想、开源产品、商业产品、趣闻轶事、风流人物等都有所了解和研究。对于 Java，我非常感激，从物质上来说，它给了我工作，帮助我养家糊口；从精神上来说，它带给我无数的喜悦、困惑、痛苦和无奈——一如我们的生活。

我不是技术高手，只是技术领域的一个拓荒者，我希望把自己的知识和经验贡献出来，以飨读者。在写作的过程中，我也反复地思考：我为谁而写这本书？为什么要写？

希望本书能帮您少走弯路

- 您是否曾经为了提供一个“One Line”的解决方案而彻夜地查看源代码？现在您不用了。
 - 您是否曾经为了理解某个算法而冥思苦想、阅览群书？现在您不用了。
 - 您是否曾经为了提升 0.1 秒的性能而对 N 种实现方案进行严格测试和对比？现在您不用了。
 - 您是否曾经为了避免多线程死锁问题而遍寻高手共同诊治？现在您不用了。
-

在学习和使用 Java 的过程中您是否在原本可以很快掌握或解决的问题上耗费了大量的时间和精力？也许您现在不用了，本书的很多内容都是我用曾经付出的代价换来的，希望它能帮助您少走弯路！

希望本书能帮您打牢基础

那些所谓的架构师、设计师、项目经理、分析师们，已经有多长时间没有写过代码了？代码是一切的基石，我不太信任连“Hello World”都没有写过的架构师。看看我们软件界的先辈们吧，Dennis M. Ritchie 决定创造一门“看上去很好”的语言时，如果只是站在高处呐喊，这门语言是划时代的，它有多么优秀，但不去实现，又有何用呢？没有 Dennis M. Ritchie 的亲自编码实现，C 语言不可能诞生，UNIX 操作系统也不可能诞生。Linux 在聚拢成千上万的开源狂热者对它进行开发和扩展之前，如果没有 Linus 的编码实现，仅凭他高声呐喊“我要创造一个划时代的操作系统”，有用吗？一切的一切都是以编码实现为前提的，代码是我们前进的基石。

这是一个英雄辈出的年代，我们每个人都希望自己被顶礼膜拜，可是这需要资本和实力，而我们的实力体现了我们处理技术问题的能力：

- 你能写出简单、清晰、高效的代码？——Show it!
- 你能架构一个稳定、健壮、快捷的系统？——Do it!
- 你能回答一个困扰 N 多人的问题？——Answer it!
- 你能修复一个系统 Bug？——Fix it!
- 你非常熟悉某个开源产品？——Broadcast it!
- 你能提升系统性能？——Tune it!

.....

但是，“工欲善其事，必先利其器”，在“善其事”之前，先看看我们的“器”是否已经磨得足够锋利了，是否能够在我们前进的路上披荆斩棘。无论您将来的职业发展方向是架构师、设计师、分析师、管理者，还是其他职位，只要您还与软件打交道，您就有必要打好技术基础。本书对核心的 Java 编程技术进行了凝练，如果能全部理解并付诸实践，您的基础一定会更加牢固。

希望本书能帮您打造一支技术战斗力强的团队

在您的团队中是否出现过以下现象：

- 没有人愿意听一场关于编码奥秘的讲座，他们觉得这是浪费时间；
- 没有人愿意去思考和探究一个算法，他们觉得这实在是多余，Google 完全可以解决；
- 没有人愿意主动重构一段代码，他们觉得新任务已经堆积成山了，“没有坏，就不要去修它”；

- 没有人愿意格式化一下代码，即便只需要按一下【Ctrl+Shift+F】快捷键，他们觉得代码写完就完了，何必再去温习；
- 没有人愿意花时间去深究一下开源框架，他们觉得够用就好；
.....

一支有实力的软件研发团队是建立在技术的基础之上的，团队成员之间需要经常地互相交流和切磋，尤其是基于可辨别、可理解的编码问题。不可否认，概念和思想也很重要，但我更看重基于代码的交流，因为代码不会说谎，比如 SOA，10 个人至少会有 5 个答案，但代码就不同了，同样的代码，结果只有一个，要么是错的，要么是对的，这才是一个技术团队应该有的氛围。本书中提出的这些问题绝大部分可能都是您的团队成员在日常的开发中会遇到的，我针对这些问题给出的建议不是唯一的解决方案，也许您的团队在讨论这一个个问题的时候能有更好的解决办法。通过对本书中的这些问题的争辩、讨论和实践能全面提升每一位团队成员的技术实力，从而增强整个团队的战斗力！

本书特色

深。本书不是一本语法书，它不会教您怎么编写 Java 代码，但是它会告诉您，为什么 StringBuilder 会比 String 类效率高，HashMap 的自增是如何实现的，为什么并行计算一般都是从 Executors 开始的……不仅仅告诉您 How（怎么做），而且还告诉您 Why（为什么要这样做）。

广。涉及面广，从编码规则到编程思想，从基本语法到系统框架，从 JDK API 到开源产品，全部都有涉猎，而且所有的建议都不是纸上谈兵，都与真实的场景相结合。

点。讲解一个知识点，而不是一个知识面，比如多线程，这里不提供多线程的解决方案，而是告诉您如何安全地停止一个线程，如何设置多线程关卡，什么时候该用 lock，什么时候该用 synchronize，等等。

精。简明扼要，直捣黄龙，一个建议就是对一个问题的解释和说明，以及提出相关的解决方案，不拖泥带水，只针对一个知识点进行讲解。

畅。本书延续了我一贯的写作风格，行云流水，娓娓道来，每次想好了一个主题后，都会先打一个腹稿，思考如何讲才能更流畅。本书不是一本很无趣的书，我一直想把它写得生动和优雅，但 Code 就是 Code，很多时候容不得深加工，最直接也就是最简洁的。

这是一本建议书，想想看，在您写代码的时候，有这样一本书籍在您的手边，告诉您如何才能编写出优雅而高效的代码，那将是一件多么惬意的事情啊！

本书面向的读者

- 寻找“One Line”（一行）解决方案的编码人员。

- 希望提升自己编码能力的程序员。
- 期望能够在开源世界仗剑而行的有志之士。
- 对编码痴情的人。

总之，只要还在 Java 圈子里混就有必要阅读本书，不管是程序员、测试人员、分析师、架构师，还是项目经理，都有必要。

如何阅读本书

首先声明，本书不是面向初级 Java 程序员的，在阅读本书之前至少要对基本的 Java 语法有初步了解，最好是参与过几个项目，写过一些代码，具备了这些条件，阅读本书才会有更大的收获，才会觉得是一种享受。

本书的各个章节和各个建议都是相对独立的，所以，您可以从任何章节的任何建议开始阅读。强烈建议您将它放在办公桌旁，遇到问题时随手翻阅。

本书附带有大量的源码（下载地址见华章网站 www.hzbook.com），建议大家在阅读本书时拷贝书中的示例代码，放到自己的收藏夹中，以备需要时使用。

勘误与支持

首先，我要为书中可能出现的错别字、多意句、歧义句、代码缺陷等错误向您真诚地道歉。虽然杨福川、杨绣国两位编辑和我都为此书付出了非常大的努力，但可能还是会有一些瑕疵，如果你在阅读本书时发现错误或有问题想一起讨论，请发邮件（cbf4life@126.com）给我，我会尽快给您回复。

本书的所有勘误，我都会发表在我的个人博客（<http://cbf4life.iteye.com/>）上。

致谢

首先，感谢杨福川和杨绣国两位编辑，在他们的编审下，本书才有了一个质的飞跃，没有他们的计划和安排，本书不可能出版。

其次，感谢家人的支持，为了写这本书，用尽了全部的休息时间，很少有时间陪伴父母和妻儿，甚至连吃一顿团圆饭都成了奢望，他们的大力支持让我信心满怀、干劲十足。儿子已经 6 岁了，明白骑在爸爸身上是对爸爸的折磨，也知道玩具是可以从网上买到的，“爸爸，给我买一个变形金刚……你在网上查呀……今天一定要买……”儿子在不知不觉中长大了。

再次，感谢交通银行“531”工程的所有领导和同事，是他们让我在这样超大规模的工程中学习和成长，使自己的技术和技能有了长足的进步；感谢我的领导李海宁总经理和周云

康高级经理，他们时时迸发出的闪光智慧让我受益匪浅；感谢软件开发中心所有同仁对我的帮助和鼓励！

最后，感谢我的朋友王骢，他无偿地把钥匙给我，让我有一个安静的地方思考和写作，有这样的朋友，人生无憾！

当然，还要感谢您，感谢您对本书的关注。

再次对本书中可能出现的错误表示歉意，真诚地接受大家的“轰炸”！

秦小波

2011年8月于上海



前 言

第 1 章 Java 开发中通用的方法和准则 /1

建议 1: 不要在常量和变量中出现易混淆的字母 /2

建议 2: 莫让常量蜕变成变量 /2

建议 3: 三元操作符的类型务必一致 /3

建议 4: 避免带有变长参数的方法重载 /4

建议 5: 别让 null 值和空值威胁到变长方法 /6

建议 6: 覆写变长方法也循规蹈矩 /7

建议 7: 警惕自增的陷阱 /8

建议 8: 不要让旧语法困扰你 /10

建议 9: 少用静态导入 /11

建议 10: 不要在本类中覆盖静态导入的变量和方法 /13

建议 11: 养成良好习惯, 显式声明 UID/14

建议 12: 避免用序列化类在构造函数中为不变量赋值 /17

建议 13: 避免为 final 变量复杂赋值 /19

建议 14: 使用序列化类的私有方法巧妙解决部分属性持久化问题 /20

建议 15: break 万万不可忘 /23

建议 16: 易变业务使用脚本语言编写 /25

- 建议 17: 慎用动态编译 /27
- 建议 18: 避免 instanceof 非预期结果 /29
- 建议 19: 断言绝对不是鸡肋 /31
- 建议 20: 不要只替换一个类 /33

第 2 章 基本类型 /35

- 建议 21: 用偶判断, 不用奇判断 /36
- 建议 22: 用整数类型处理货币 /37
- 建议 23: 不要让类型默默转换 /38
- 建议 24: 边界, 边界, 还是边界 /39
- 建议 25: 不要让四舍五入亏了一方 /41
- 建议 26: 提防包装类型的 null 值 /43
- 建议 27: 谨慎包装类型的大小比较 /45
- 建议 28: 优先使用整形池 /46
- 建议 29: 优先选择基本类型 /48
- 建议 30: 不要随便设置随机种子 /49

第 3 章 类、对象及方法 /52

- 建议 31: 在接口中不要存在实现代码 /53
- 建议 32: 静态变量一定要先声明后赋值 /54
- 建议 33: 不要覆写静态方法 /55
- 建议 34: 构造函数尽量简化 /57
- 建议 35: 避免在构造函数中初始化其他类 /58
- 建议 36: 使用构造代码块精炼程序 /60
- 建议 37: 构造代码块会想你所想 /61
- 建议 38: 使用静态内部类提高封装性 /63
- 建议 39: 使用匿名类的构造函数 /65
- 建议 40: 匿名类的构造函数很特殊 /66
- 建议 41: 让多重继承成为现实 /68
- 建议 42: 让工具类不可实例化 /70
- 建议 43: 避免对象的浅拷贝 /71
- 建议 44: 推荐使用序列化实现对象的拷贝 /73
- 建议 45: 覆写 equals 方法时不要识别不出自己 /74

- 建议 46: equals 应该考虑 null 值情景 /76
- 建议 47: 在 equals 中使用 getClass 进行类型判断 /77
- 建议 48: 覆写 equals 方法必须覆写 hashCode 方法 /78
- 建议 49: 推荐覆写 toString 方法 /80
- 建议 50: 使用 package-info 类为包服务 /81
- 建议 51: 不要主动进行垃圾回收 /82

第 4 章 字符串 /83

- 建议 52: 推荐使用 String 直接量赋值 /84
- 建议 53: 注意方法中传递的参数要求 /85
- 建议 54: 正确使用 String、StringBuffer、StringBuilder/86
- 建议 55: 注意字符串的位置 /87
- 建议 56: 自由选择字符串拼接方法 /88
- 建议 57: 推荐在复杂字符串操作中使用正则表达式 /90
- 建议 58: 强烈建议使用 UTF 编码 /92
- 建议 59: 对字符串排序持一种宽容的心态 /94

第 5 章 数组和集合 /97

- 建议 60: 性能考虑, 数组是首选 /98
- 建议 61: 若有必要, 使用变长数组 /99
- 建议 62: 警惕数组的浅拷贝 /100
- 建议 63: 在明确的场景下, 为集合指定初始容量 /101
- 建议 64: 多种最值算法, 适时选择 /104
- 建议 65: 避开基本类型数组转换列表陷阱 /105
- 建议 66: asList 方法产生的 List 对象不可更改 /107
- 建议 67: 不同的列表选择不同的遍历方法 /108
- 建议 68: 频繁插入和删除时使用 LinkedList/112
- 建议 69: 列表相等只需关心元素数据 /115
- 建议 70: 子列表只是原列表的一个视图 /117
- 建议 71: 推荐使用 subList 处理局部列表 /119
- 建议 72: 生成子列表后不要再操作原列表 /120
- 建议 73: 使用 Comparator 进行排序 /122
- 建议 74: 不推荐使用 binarySearch 对列表进行检索 /125

建议 75: 集合中的元素必须做到 compareTo 和 equals 同步 /127

建议 76: 集合运算时使用更优雅的方式 /129

建议 77: 使用 shuffle 打乱列表 /131

建议 78: 减少 HashMap 中元素的数量 /132

建议 79: 集合中的哈希码不要重复 /135

建议 80: 多线程使用 Vector 或 HashTable/139

建议 81: 非稳定排序推荐使用 List/141

建议 82: 由点及面, 一叶知秋——集合大家族 /143

第 6 章 枚举和注解 /145

建议 83: 推荐使用枚举定义常量 /146

建议 84: 使用构造函数协助描述枚举项 /149

建议 85: 小心 switch 带来的空值异常 /150

建议 86: 在 switch 的 default 代码块中增加 AssertionError 错误 /152

建议 87: 使用 valueOf 前必须进行校验 /152

建议 88: 用枚举实现工厂方法模式更简洁 /155

建议 89: 枚举项的数量限制在 64 个以内 /157

建议 90: 小心注解继承 /160

建议 91: 枚举和注解结合使用威力更大 /162

建议 92: 注意 @Override 不同版本的区别 /164

第 7 章 泛型和反射 /166

建议 93: Java 的泛型是类型擦除的 /167

建议 94: 不能初始化泛型参数和数组 /169

建议 95: 强制声明泛型的实际类型 /170

建议 96: 不同的场景使用不同的泛型通配符 /172

建议 97: 警惕泛型是不能协变和逆变的 /174

建议 98: 建议采用的顺序是 List<T>、List<?>、List<Object>/176

建议 99: 严格限定泛型类型采用多重界限 /177

建议 100: 数组的真实类型必须是泛型类型的子类型 /179

建议 101: 注意 Class 类的特殊性 /181

建议 102: 适时选择 getDeclared××× 和 get×××/181

建议 103: 反射访问属性或方法时将 Accessible 设置为 true /182

- 建议 104: 使用 `forName` 动态加载类文件 /184
- 建议 105: 动态加载不适合数组 /186
- 建议 106: 动态代理可以使代理模式更加灵活 /188
- 建议 107: 使用反射增加装饰模式的普适性 /190
- 建议 108: 反射让模板方法模式更强大 /192
- 建议 109: 不需要太多关注反射效率 /194

第 8 章 异常 /197

- 建议 110: 提倡异常封装 /198
- 建议 111: 采用异常链传递异常 /200
- 建议 112: 受检异常尽可能转化为非受检异常 /202
- 建议 113: 不要在 `finally` 块中处理返回值 /204
- 建议 114: 不要在构造函数中抛出异常 /207
- 建议 115: 使用 `Throwable` 获得栈信息 /210
- 建议 116: 异常只为异常服务 /212
- 建议 117: 多使用异常, 把性能问题放一边 /213

第 9 章 多线程和并发 /215

- 建议 118: 不推荐覆写 `start` 方法 /216
- 建议 119: 启动线程前 `stop` 方法是不可靠的 /218
- 建议 120: 不使用 `stop` 方法停止线程 /220
- 建议 121: 线程优先级只使用三个等级 /224
- 建议 122: 使用线程异常处理器提升系统可靠性 /226
- 建议 123: `volatile` 不能保证数据同步 /228
- 建议 124: 异步运算考虑使用 `Callable` 接口 /232
- 建议 125: 优先选择线程池 /233
- 建议 126: 适时选择不同的线程池来实现 /237
- 建议 127: `Lock` 与 `synchronized` 是不一样的 /240
- 建议 128: 预防线程死锁 /245
- 建议 129: 适当设置阻塞队列长度 /250
- 建议 130: 使用 `CountDownLatch` 协调子线程 /252
- 建议 131: `CyclicBarrier` 让多线程齐步走 /254

第 10 章 性能和效率 /256

- 建议 132: 提升 Java 性能的基本方法 /257
- 建议 133: 若非必要, 不要克隆对象 /259
- 建议 134: 推荐使用“望闻问切”的方式诊断性能 /261
- 建议 135: 必须定义性能衡量标准 /263
- 建议 136: 枪打出头鸟——解决首要系统性能问题 /264
- 建议 137: 调整 JVM 参数以提升性能 /266
- 建议 138: 性能是个大“咕咚” /268

第 11 章 开源世界 /271

- 建议 139: 大胆采用开源工具 /272
- 建议 140: 推荐使用 Guava 扩展工具包 /273
- 建议 141: Apache 扩展包 /276
- 建议 142: 推荐使用 Joda 日期时间扩展包 /280
- 建议 143: 可以选择多种 Collections 扩展 /282

第 12 章 思想为源 /285

- 建议 144: 提倡良好的代码风格 /286
- 建议 145: 不要完全依靠单元测试来发现问题 /287
- 建议 146: 让注释正确、清晰、简洁 /290
- 建议 147: 让接口的职责保持单一 /294
- 建议 148: 增强类的可替换性 /295
- 建议 149: 依赖抽象而不是实现 /298
- 建议 150: 抛弃 7 条不良的编码习惯 /299
- 建议 151: 以技术员自律而不是工人 /301



第1章

Java 开发中通用的方法和准则

The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself.

明白事理的人使自己适应世界；不明事理的人想让世界适应自己。

——萧伯纳

Java 的世界丰富又多彩，但同时也布满了荆棘陷阱，大家一不小心就可能跌入黑暗深渊，只有在了解了其通行规则后才能使自己在技术的海洋里遨游飞翔，恣意驰骋。

“千里之行始于足下”，本章主要讲述与 Java 语言基础有关的问题及建议的解决方案，例如常量和变量的注意事项、如何更安全地序列化、断言到底该如何使用等。

建议 1：不要在常量和变量中出现易混淆的字母

包名全小写，类名首字母全大写，常量全部大写并用下划线分隔，变量采用驼峰命名法（Camel Case）命名等，这些都是最基本的 Java 编码规范，是每个 Javaer 都应熟知的规则，但是在变量的声明中要注意不要引入容易混淆的字母。尝试阅读如下代码，思考一下打印出的 i 等于多少：

```
public class Client {
    public static void main(String[] args) {
        long i = 1l;
        System.out.println("i 的两倍是：" + (i+i));
    }
}
```

肯定有人会说：这么简单的例子还能出错？运行结果肯定是 22！实践是检验真理的唯一标准，将其拷贝到 Eclipse 中，然后 Run 一下看看，或许你会很奇怪，结果是 2，而不是 22，难道是 Eclipse 的显示有问题，少了个“2”？

因为赋给变量 i 的数字就是“1”，只是后面加了长整型变量的标志字母“l”而已。别说是挖坑让你跳，如果有类似程序出现在项目中，当你试图通过阅读代码来理解作者的思想时，此情此景就有可能会出现。所以，为了让您的程序更容易理解，字母“l”（还包括大写字母“O”）尽量不要和数字混用，以免使阅读者的理解与程序意图产生偏差。如果字母和数字必须混合使用，字母“l”务必大写，字母“O”则增加注释。

注意 字母“l”作为长整型标志时务必大写。

建议 2：莫让常量蜕变成变量

常量蜕变成变量？你胡扯吧，加了 final 和 static 的常量怎么可能会变呢？不可能二次赋值的呀。真的不可能吗？看我们神奇的魔术，代码如下：

```
public class Client {
    public static void main(String[] args) {
        System.out.println("常量会变哦：" + Const.RAND_CONST);
    }
}
```

```

}
/* 接口常量 */
interface Const{
    // 这还是常量吗?
    public static final int RAND_CONST = new Random().nextInt();
}

```

RAND_CONST 是常量吗？它的值会变吗？绝对会变！这种常量的定义方式是极不可取的，常量就是常量，在编译期就必须确定其值，不应该在运行期更改，否则程序的可读性会非常差，甚至连作者自己都不能确定在运行期发生了何种神奇的事情。

甭想着使用常量会变的这个功能来实现序列号算法、随机种子生成，除非这真的是项目中的唯一方案，否则就放弃吧，常量还是当常量使用。

注意 务必让常量的值在运行期保持不变。

建议 3：三元操作符的类型务必一致

三元操作符是 if-else 的简化写法，在项目中使用它的地方很多，也非常好用，但是好用又简单的东西并不表示就可以随便用，我们来看看下面这段代码：

```

public class Client {
    public static void main(String[] args) {
        int i = 80;
        String s = String.valueOf(i<100?90:100);
        String s1 = String.valueOf(i<100?90:100.0);
        System.out.println("两者是否相等 :" + s.equals(s1));
    }
}

```

分析一下这段程序：i 是 80，那它当然小于 100，两者的返回值肯定都是 90，再转成 String 类型，其值也绝对相等，毋庸置疑的。恩，分析得有点道理，但是变量 s 中三元操作符的第二个操作数是 100，而 s1 的第二个操作数是 100.0，难道没有影响吗？不可能有影响吧，三元操作符的条件都为真了，只返回第一个值嘛，与第二个值有一毛钱的关系吗？貌似有道理。

果真如此吗？我们通过结果来验证一下，运行结果是：“两者是否相等： false”，什么？不相等，Why？

问题就出在了 100 和 100.0 这两个数字上，在变量 s 中，三元操作符中的第一个操作数（90）和第二个操作数（100）都是 int 类型，类型相同，返回的结果也就是 int 类型的 90，而变量 s1 的情况就有点不同了，第一个操作数是 90（int 类型），第二个操作数却是 100.0，而这是个浮点数，也就是说两个操作数的类型不一致，可三元操作符必须要返回一个数据，

而且类型要确定，不可能条件为真时返回 int 类型，条件为假时返回 float 类型，编译器是不允许如此的，所以它就会进行类型转换了，int 型转换为浮点数 90.0，也就是说三元操作符的返回值是浮点数 90.0，那这当然与整型的 90 不相等了。这里可能有读者疑惑了：为什么是整型转为浮点，而不是浮点转为整型呢？这就涉及三元操作符类型的转换规则：

- 若两个操作数不可转换，则不做转换，返回值为 Object 类型。
- 若两个操作数是明确类型的表达式（比如变量），则按照正常的二进制数字来转换，int 类型转换为 long 类型，long 类型转换为 float 类型等。
- 若两个操作数中有一个是数字 S，另外一个是表达式，且其类型标示为 T，那么，若数字 S 在 T 的范围内，则转换为 T 类型；若 S 超出了 T 类型的范围，则 T 转换为 S 类型（可以参考“建议 22”，会对该问题进行展开描述）。
- 若两个操作数都是直接量数字（Literal）[⊖]，则返回值类型为范围较大者。

知道是什么原因了，相应的解决办法也就有了：保证三元操作符中的两个操作数类型一致，即可减少可能错误的发生。

建议 4：避免带有变长参数的方法重载

在项目和系统的开发中，为了提高方法的灵活度和可复用性，我们经常要传递不确定数量的参数到方法中，在 Java 5 之前常用的设计技巧就是把形参定义成 Collection 类型或其子类型，或者是数组类型，这种方法的缺点就是需要对空参数进行判断和筛选，比如实参为 null 值和长度为 0 的 Collection 或数组。而 Java 5 引入变长参数（varargs）就是为了更好地提高方法的复用性，让方法的调用者可以“随心所欲”地传递实参数量，当然变长参数也是要遵循一定规则的，比如变长参数必须是方法中的最后一个参数；一个方法不能定义多个变长参数等，这些基本规则需要牢记，但是即使记住了这些规则，仍然有可能出现错误，我们来看如下代码：

```
public class Client {
    // 简单折扣计算
    public void calPrice(int price,int discount){
        float knockdownPrice =price * discount / 100.0F;
        System.out.println("简单折扣后的价格是：" +formateCurrency(knockdownPrice));
    }
    // 复杂多折扣计算
    public void calPrice(int price,int... discounts){
        float knockdownPrice = price;
        for(int discount:discounts){
            knockdownPrice = knockdownPrice * discount / 100;
```

[⊖] “Literal” 也译作“字面量”。

```

    }
    System.out.println(" 复杂折扣后的价格是: " +formatCurrency(knockdownPrice));
}
// 格式化成本的货币形式
private String formatCurrency(float price) {
    return NumberFormat.getCurrencyInstance().format(price/100);
}

public static void main(String[] args) {
    Client client = new Client();
    //499 元的货物，打 75 折
    client.calPrice(49900, 75);
}
}
}

```

这是一个计算商品价格折扣的模拟类，带有两个参数的 calPrice 方法（该方法的业务逻辑是：提供商品的原价和折扣率，即可获得商品的折扣价）是一个简单的折扣计算方法，该方法在实际项目中经常会用到，这是单一的打折方法。而带有变长参数的 calPrice 方法则是较复杂的折扣计算方式，多种折扣的叠加运算（模拟类是一种比较简单的实现）在实际生活中也是经常见到的，比如在大甩卖期间对 VIP 会员再度进行打折；或者当天是你的生日，再给你打个 9 折，也就是俗话说的“折上折”。

业务逻辑清楚了，我们来仔细看看这两个方法，它们是重载吗？当然是了，重载的定义是“方法名相同，参数类型或数量不同”，很明显这两个方法是重载。但是再仔细瞧瞧，这个重载有点特殊：calPrice (int price,int... discounts) 的参数范畴覆盖了 calPrice (int price,int discount) 的参数范畴。那问题就出来了：对于 calPrice (49900,75) 这样的计算，到底该调用哪个方法来处理呢？

我们知道 Java 编译器是很聪明的，它在编译时会根据方法签名（Method Signature）来确定调用哪个方法，比如 calPrice (499900,75,95) 这个调用，很明显 75 和 95 会被转成一个包含两个元素的数组，并传递到 calPrice (int price,in.. discounts) 中，因为只有这一个方法签名符合该实参类型，这很容易理解。但是我们现在面对的是 calPrice (49900,75) 调用，这个“75”既可以被编译成 int 类型的“75”，也可以被编译成 int 数组“{75}”，即只包含一个元素的数组。那到底该调用哪一个方法呢？

我们先运行一下看看结果，运行结果是：

简单折扣后的价格是：¥374.25。

看来是调用了第一个方法，为什么会调用第一个方法，而不是第二个变长参数方法呢？因为 Java 在编译时，首先会根据实参的数量和类型（这里是 2 个实参，都为 int 类型，注意没有转成 int 数组）来进行处理，也就是查找到 calPrice(int price,int discount) 方法，而且确认它是否符合方法签名条件。现在的问题是编译器为什么会首先根据 2 个 int 类型的实参而不是 1 个 int 类型、1 个 int 数组类型的实参来查找方法呢？这是个好问题，也非常好回答：

因为 int 是一个原生数据类型，而数组本身是一个对象，编译器想要“偷懒”，于是它会从最简单的开始“猜想”，只要符合编译条件的即可通过，于是就出现了此问题。

问题是阐述清楚了，为了让我们的程序能被“人类”看懂，还是慎重考虑变长参数的方法重载吧，否则让人伤脑筋不说，说不定哪天就陷入这类小陷阱里了。

建议 5：别让 null 值和空值威胁到变长方法

上一建议讲解了变长参数的重载问题，本建议还会继续讨论变长参数的重载问题。上一建议的例子是变长参数的范围覆盖了非变长参数的范围，这次我们从两个都是变长参数的方法说起，代码如下：

```
public class Client {
    public void methodA(String str, Integer... is) {
    }

    public void methodA(String str, String... strs) {
        .
        .

        public static void main(String[] args) {
            Client client = new Client();
            client.methodA("China", 0);
            client.methodA("China", "People");
            client.methodA("China");
            client.methodA("China", null);
        }
    }
}
```

两个 methodA 都进行了重载，现在的问题是：上面的代码编译通不过，问题出在什么地方？看似很简单哦。

有两处编译通不过：client.methodA ("China") 和 client.methodA ("China",null)，估计你已经猜到了，两处的提示是相同的：方法模糊不清，编译器不知道调用哪一个方法，但这两处代码反映的代码味道可是不同的。

对于 methodA ("China") 方法，根据实参 “China” (String 类型)，两个方法都符合形参格式，编译器不知道该调用哪个方法，于是报错。我们来思考这个问题：Client 类是一个复杂的商业逻辑，提供了两个重载方法，从其他模块调用（系统内本地调用或系统外远程调用）时，调用者根据变长参数的规范调用，传入变长参数的实参数量可以是 N 个 ($N \geq 0$)，那当然可以写成 client.methodA ("china") 方法啊！完全符合规范，但是这却让编译器和调用者都很郁闷，程序符合规则却不能运行，如此问题，谁之责任呢？是 Client 类的设计者，他违反了 KISS 原则 (Keep It Simple, Stupid, 即懒人原则)，按照此规则设计的方法应该很容易调用，可是现在在遵循规范的情况下，程序竟然出错了，这对设计者和开发者而言都是

应该严禁出现的。

对于 client.methodA ("china",null) 方法，直接量 null 是没有类型的，虽然两个 methodA 方法都符合调用请求，但不知道调用哪一个，于是报错了。我们来体会一下它的坏味道：除了不符合上面的懒人原则外，这里还有一个非常不好的编码习惯，即调用者隐藏了实参类型，这是非常危险的，不仅仅调用者需要“猜测”该调用哪个方法，而且被调用者也可能产生内部逻辑混乱的情况。对于本例来说应该做如下修改：

```
public static void main(String[] args) {
    Client client = new Client();
    String[] strs = null;
    client.methodA("China",strs);
}
```

也就是说让编译器知道这个 null 值是 String 类型的，编译即可顺利通过，也就减少了错误的发生。

建议 6：覆写变长方法也循规蹈矩

在 Java 中，子类覆写父类中的方法很常见，这样做既可以修正 Bug 也可以提供扩展的业务功能支持，同时还符合开闭原则（Open-Closed Principle），我们来看一下覆写必须满足的条件：

- 重写方法不能缩小访问权限。
- **参数列表必须与被重写方法相同。**
- 返回类型必须与被重写方法的相同或是其子类。
- 重写方法不能抛出新的异常，或者超出父类范围的异常，但是可以抛出更少、更有限的异常，或者不抛出异常。

估计你已经猜测出下面要讲的内容了，为什么“参数列表必须与被重写方法的相同”采用不同的字体，这其中是不是有什么玄机？是的，还真有那么一点点小玄机。参数列表相同包括三层意思：参数数量相同、类型相同、顺序相同，看上去好像没什么问题，那我们来看一个例子，业务场景与上一个建议相同，商品打折，代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 向上转型
        Base base = new Sub();
        base.fun(100, 50);
        // 不转型
        Sub sub = new Sub();
        sub.fun(100, 50);
    }
}
```

```

}
// 基类
class Base{
    void fun(int price,int... discounts){
        System.out.println("Base.....fun");
    }
}

// 子类，覆写父类方法
class Sub extends Base{
    @Override
    void fun(int price,int[] discounts){
        System.out.println("Sub.....fun");
    }
}

```

请问：该程序有问题吗？——编译通不过。那问题出在什么地方呢？

@Override 注解吗？非也，覆写是正确的，因为父类的 calPrice 编译成字节码后的形参是一个 int 类型的形参加上一个 int 数组类型的形参，子类的参数列表也与此相同，那覆写是理所当然的了，所以加上 @Override 注解没有问题，只是 Eclipse 会提示这不是一种很好的编码风格。

难道是“sub.fun(100, 50)”这条语句？正解，确实是这条语句报错，提示找不到 fun(int,int) 方法。这太奇怪了：子类继承了父类的所有属性和方法，甭管是私有的还是公开的访问权限，同样的参数、同样的方法名，通过父类调用没有任何问题，通过子类调用却编译通不过，为啥？难道是没继承下来？或者子类缩小了父类方法的前置条件？那如果是这样，就不应该覆写，@Override 就应该报错，真是奇妙的事情！

事实上，base 对象是把子类对象 Sub 做了向上转型，形参列表是由父类决定的，由于是变长参数，在编译时，“base.fun(100, 50)”中的“50”这个实参会被编译器“猜测”而编译成“{50}”数组，再由子类 Sub 执行。我们再来看看直接调用子类的情况，这时编译器并不会把“50”做类型转换，因为数组本身也是一个对象，编译器还没有聪明到要在两个没有继承关系的类之间做转换，要知道 Java 是要求严格的类型匹配的，类型不匹配编译器自然就会拒绝执行，并给予错误提示。

这是个特例，覆写的方法参数列表竟然与父类不相同，这违背了覆写的定义，并且会引发莫名其妙的错误。所以读者在对变长参数进行覆写时，如果要使用此类似的方法，请找个黑屋仔细想想是不是一定要如此。

注意 覆写的方法参数与父类相同，不仅仅是类型、数量，还包括显示形式。

建议 7：警惕自增的陷阱

记得大学刚开始学 C 语言时，老师就说：自增有两种形式，分别是 i++ 和 ++i，i++ 表

示的是先赋值后加 1，`++i` 是先加 1 后赋值，这样理解了很多年也没出现问题，直到遇到如下代码，我才怀疑我的理解是不是错了：

```
public class Client {
    public static void main(String[] args) {
        int count = 0;
        for(int i=0;i<10;i++){
            count=count++;
        }
        System.out.println("count="+count);
    }
}
```

这个程序输出的 `count` 等于几？是 `count` 自加 10 次吗？答案等于 10？可以非常肯定地告诉你，答案错误！运行结果是 `count` 等于 0。为什么呢？

`count++` 是一个表达式，是有返回值的，它的返回值就是 `count` 自加前的值，Java 对自加是这样处理的：首先把 `count` 的值（注意是值，不是引用）拷贝到一个临时变量区，然后对 `count` 变量加 1，最后返回临时变量区的值。程序第一次循环时的详细处理步骤如下：

步骤 1 JVM 把 `count` 值（其值是 0）拷贝到临时变量区。

步骤 2 `count` 值加 1，这时候 `count` 的值是 1。

步骤 3 返回临时变量区的值，注意这个值是 0，没修改过。

步骤 4 返回值赋值给 `count`，此时 `count` 值被重置成 0。

“`count=count++`” 这条语句可以按照如下代码来理解：

```
public static int mockAdd(int count){
    // 先保存初始值
    int temp = count;
    // 做自增操作
    count = count+1;
    // 返回原始值
    return temp;
}
```

于是第一次循环后 `count` 的值还是 0，其他 9 次的循环也是一样的，最终你会发现 `count` 的值始终没有改变，仍然保持着最初的状态。

此例中代码作者的本意是希望 `count` 自增，所以想当然地认为赋值给自身就成了，不曾想掉到 Java 自增的陷阱中了。解决方法很简单，只要把 “`count=count++`” 修改为 “`count++`” 即可。该问题在不同的语言环境有不同的实现：C++ 中 “`count=count++`” 与 “`count++`” 是等效的，而在 PHP 中则保持着与 Java 相同的处理方式。每种语言对自增的实现方式各不同，读者有兴趣可以多找几种语言测试一下，思考一下原理。

下次如果看到某人 T 恤上印着 “`i=i++`”，千万不要鄙视他，记住，能够以不同的语言解释清楚这句话的人绝对不简单，应该表现出 “如滔滔江水” 般的敬仰，心理默念着 “高人，

绝世高人哪”。

建议 8：不要让旧语法困扰你

N 多年前接手了一个除了源码以外什么都没有的项目，没需求、没文档、没设计，原创者也已鸟兽散了，我们只能通过阅读源码来进行维护。期间，同事看到一段很“奇妙”的代码，让大家帮忙分析，代码片段如下：

```
public class Client {
    public static void main(String[] args) {
        // 数据定义及初始化
        int fee=200;
        // 其他业务处理
        saveDefault:save(fee);
        // 其他业务处理
    }

    static void saveDefault(){
    }

    static void save(int fee){
    }
}
```

该代码的业务含义是计算交易的手续费，最低手续费是 2 元，其业务逻辑大致看懂了，但是此代码非常神奇，“`saveDefault:save (fee)`”这句代码在此处出现后，后续就再也没有与此有关的代码了，这做何解释呢？更神奇的是，编译竟然还没有错，运行也很正常。Java 中竟然有冒号操作符，一般情况下，它除了在唯一一个三元操作符中存在外就没有其他地方可用了吧。当时连项目组里的高手也是一愣一愣的，翻语法书，也没有介绍冒号操作符的内容，而且，也不可能出现连括号都可以省掉的方法调用、方法级联啊！这也太牛了吧！

隔壁做 C 项目的同事过来串门，看我们在讨论这个问题，很惊奇地说“耶，Java 中还有标号呀，我以为 Java 这么高级的语言已经抛弃 goto 语句了……”，一语点醒梦中人：项目的原创者是 C 语言转过来的开发人员，所以他把 C 语言的 goto 习惯也带到项目中了，后来由于经过 N 手交接，重构了多次，到我们这里 goto 语句已经被重构掉了，但是跳转标号还保留着，估计上一届的重构者也是稀里糊涂的，不敢贸然修改，所以把这个重任留给了我们。

goto 语句中有着“double face”作用的关键字，它可以让程序从多层的循环中跳出，不用一层一层地退出，类似高楼着火了，来不及一楼一楼的下，goto 语句就可以让你“biu~”的一声从十层楼跳到地面上。这点确实很好，但同时也带来了代码结构混乱的问题，而且程序跳来跳去让人看着就头晕，还怎么调试？！这样做甚至会隐祸连连，比如标号前后对象构造或变量初始化，一旦跳到这个标号，程序就不可想象了，所以 Java 中抛弃了 goto 语法，

但还是保留了该关键字，只是不进行语义处理而已，与此类似的还有 const 关键字。

Java 中虽然没有了 goto 关键字，但是扩展了 break 和 continue 关键字，它们的后面都可以加上标号做跳转，完全实现了 goto 功能，同时也把 goto 的诟病带了进来，所以我们在阅读大牛的开源程序时，根本就看不到 break 或 continue 后跟标号的情况，甚至是 break 和 continue 都很少看到，这是提高代码可读性的一剂良药，旧语法就让它随风而去吧！

建议 9：少用静态导入

从 Java 5 开始引入了静态导入语法（import static），其目是为了减少字符输入量，提高代码的可阅读性，以便更好地理解程序。我们先来看一个不使用静态导入的例子，也就是一般导入：

```
public class MathUtils{
    // 计算圆面积
    public static double calCircleArea(double r) {
        return Math.PI * r * r;
    }
    // 计算球面积
    public static double calBallArea(double r) {
        return 4 * Math.PI * r * r;
    }
}
```

这是很简单的数学工具类，我们在这两个计算面积的方法中都引入了 java.lang.Math 类（该类是默认导入的）中的 PI（圆周率）常量，而 Math 这个类写在这里有点多余，特别是如果 MathUtils 中的方法比较多时，如果每次都要敲入 Math 这个类，繁琐且多余，静态导入可解决此类问题，使用静态导入后的程序如下：

```
import static java.lang.Math.PI;
public class MathUtils{
    // 计算圆面积
    public static double calCircleArea(double r) {
        return PI * r * r;
    }
    // 计算球面积
    public static double calBallArea(double r) {
        return 4 * PI * r * r;
    }
}
```

静态导入的作用是把 Math 类中的 PI 常量引入到本类中，这会使程序更简单，更容易阅读，只要看到 PI 就知道这是圆周率，不用每次都要把类名写全了。但是，滥用静态导入会使程序更难阅读，更难维护。静态导入后，代码中就不用再写类名了，但是我们知道类是“一

类事物的描述”，缺少了类名的修饰，静态属性和静态方法的表象意义可以被无限放大，这会让阅读者很难弄清楚其属性或方法代表何意，甚至是哪一个类的属性（方法）都要思考一番（当然，IDE 友好提示功能是另说），特别是在一个类中有多个静态导入语句时，若还使用了 *（星号）通配符，把一个类的所有静态元素都导入进来了，那简直就是噩梦。我们来看一段例子：

```
import static java.lang.Double.*;
import static java.lang.Math.*;
import static java.lang.Integer.*;
import static java.text.NumberFormat.*;

public class Client {
    // 输入半径和精度要求，计算面积
    public static void main(String[] args) {
        double s = PI * parseDouble(args[0]);
        NumberFormat nf = getInstance();
        nf.setMaximumFractionDigits(parseInt(args[1]));
        formatMessage(nf.format(s));
    }
    // 格式化消息输出
    public static void formatMessage(String s){
        System.out.println("圆面积是: "+s);
    }
}
```

就这么一段程序，看着就让人火大：常量 PI，这知道，是圆周率；parseDouble 方法可能是 Double 类的一个转换方法，这看名称也能猜测到。那紧接着的 getInstance 方法是哪个类的？是 Client 本地类？不对呀，没有这个方法，哦，原来是 NumberFormat 类的方法，这和 formatMessage 本地方法没有任何区别了——这代码也太难阅读了，非机器不可阅读。

所以，对于静态导入，一定要遵循两个规则：

- 不使用 *（星号）通配符，除非是导入静态常量类（只包含常量的类或接口）。
- 方法名是具有明确、清晰表象意义的工具类。

何为具有明确、清晰表象意义的工具类？我们来看看 JUnit 4 中使用的静态导入的例子，代码如下：

```
import static org.junit.Assert.*;
public class DaoTest {
    @Test
    public void testInsert(){
        // 断言
        assertEquals("foo", "foo");
        assertFalse(Boolean.FALSE);
    }
}
```

我们从程序中很容易判断出 `assertEquals` 方法是用来断言两个值是否相等的, `assertFalse` 方法则是断言表达式为假, 如此确实减少了代码量, 而且代码的可读性也提高了, 这也是静态导入用到正确地方所带来的好处。

建议 10：不要在本类中覆盖静态导入的变量和方法

如果一个类中的方法及属性与静态导入的方法及属性重名会出现什么问题呢? 我们先来看一个正常的静态导入, 代码如下:

```
import static java.lang.Math.PI;
import static java.lang.Math.abs;

public class Client {
    public static void main(String[] args) {
        System.out.println("PI=" + PI);
        System.out.println("abs(100)=" + abs(-100));
    }
}
```

很简单的例子, 打印出静态常量 `PI` 值, 计算 `-100` 的绝对值。现在的问题是: 如果我们在 `Client` 类中也定义了 `PI` 常量和 `abs` 方法, 会出现什么问题? 代码如下:

```
import static java.lang.Math.PI;
import static java.lang.Math.abs;

public class Client {
    // 常量名与静态导入的 PI 相同
    public final static String PI="祖冲之";
    // 方法名与静态导入的相同
    public static int abs(int abs){
        return 0;
    }

    public static void main(String[] args) {
        System.out.println("PI=" + PI);
        System.out.println("abs(100)=" + abs(-100));
    }
}
```

以上代码中, 定义了一个 `PI` 字符串类型的常量, 又定义了一个 `abs` 方法, 与静态导入的相同。首先说好消息: 编译器没有报错, 接下来是不好的消息了: 我们不知道哪个属性和哪个方法被调用了, 因为常量名和方法名相同, 到底调用了哪一个方法呢? 我们运行一下看看结果:

`PI=祖冲之`

```
abs(100)=0
```

很明显是本地的属性和方法被引用了，为什么不是 Math 类中的属性和方法呢？那是因为编译器有一个“最短路径”原则：如果能够在本类中查找到的变量、常量、方法，就不会到其他包或父类、接口中查找，以确保本类中的属性、方法优先。

因此，如果要变更一个被静态导入的方法，最好的办法是在原始类中重构，而不是在本类中覆盖。

建议 11：养成良好习惯，显式声明 UID

我们编写一个实现了 Serializable 接口（序列化标志接口）的类，Eclipse 马上就会给一个黄色警告：需要增加一个 Serial Version ID。为什么要增加？它是怎么计算出来的？有什么用？本章就来解释该问题。

类实现 Serializable 接口的目的是为了可持久化，比如网络传输或本地存储，为系统的分布和异构部署提供先决支持条件。若没有序列化，现在我们熟悉的远程调用、对象数据库都不可能存在，我们来看一个简单的序列化类：

```
public class Person implements Serializable{
    private String name;
    /*name 属性的getter/setter 方法省略 */
}
```

这是一个简单 JavaBean，实现了 Serializable 接口，可以在网络上传输，也可以本地存储然后读取。这里我们以 Java 消息服务（Java Message Service）方式传递该对象（即通过网络传递一个对象），定义在消息队列中的数据类型为 ObjectMessage，首先定义一个消息的生产者（Producer），代码如下：

```
public class Producer {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        person.setName("混世魔王");
        // 序列化，保存到磁盘上
        SerializationUtils.writeObject(person);
    }
}
```

这里引入了一个工具类 SerializationUtils，其作用是对一个类进行序列化和反序列化，并存储到硬盘上（模拟网络传输），其代码如下：

```
public class SerializationUtils {
    private static String FILE_NAME = "c:/obj.bin";
    // 序列化
    public static void writeObject(Serializable s) {
```

```

        try {
            ObjectOutputStream oos = new ObjectOutputStream(new
                FileOutputStream(FILE_NAME));
            oos.writeObject(s);
            oos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static Object readObject(){
        Object obj=null;
        // 反序列化
        try {
            ObjectInputStream input = new ObjectInputStream(new
                FileInputStream(FILE_NAME));
            obj = input.readObject();
            input.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return obj;
    }
}

```

通过对对象序列化过程，把一个对象从内存块转化为可传输的数据流，然后通过网络发送到消息消费者（Consumer）那里，并进行反序列化，生成实例对象，代码如下：

```

public class Consumer {
    public static void main(String[] args) throws Exception {
        // 反序列化
        Person p = (Person) SerializationUtils.readObject();
        System.out.println("name="+p.getName());
    }
}

```

这是一个反序列化过程，也就是对对象数据流转换为一个实例对象的过程，其运行后的输出结果为：混世魔王。这太 easy 了，是的，这就是序列化和反序列化典型的 demo。但此处隐藏着一个问题：如果消息的生产者和消息的消费者所参考的类（Person 类）有差异，会出现何种神奇事件？比如：消息生产者中的 Person 类增加了一个年龄属性，而消费者没有增加该属性。为啥没有增加？！因为这是个分布式部署的应用，你甚至都不知道这个应用部署在何处，特别是通过广播（broadcast）方式发送消息的情况，漏掉一两个订阅者也是很正常的。

在这种序列化和反序列化的类不一致的情形下，反序列化时会报一个 InvalidClassException 异常，原因是序列化和反序列化所对应的类版本发生了变化，JVM 不能把数据流转换为实例

对象。接着刨根问底：JVM 是根据什么来判断一个类版本的呢？

好问题，通过 `SerialVersionUID`，也叫做流标识符（Stream Unique Identifier），即类的版本定义的，它可以显式声明也可以隐式声明。显式声明格式如下：

```
private static final long serialVersionUID = XXXXXX;
```

而隐式声明则是我不声明，你编译器在编译的时候帮我生成。生成的依据是通过包名、类名、继承关系、非私有的方法和属性，以及参数、返回值等诸多因子计算得出的，极度复杂，基本上计算出来的这个值是唯一的。

`serialVersionUID` 如何生成已经说明了，我们再来看看 `serialVersionUID` 的作用。JVM 在反序列化时，会比较数据流中的 `serialVersionUID` 与类的 `serialVersionUID` 是否相同，如果相同，则认为类没有发生改变，可以把数据流 load 为实例对象；如果不相同，对不起，我 JVM 不干了，抛个异常 `InvalidClassException` 给你瞧瞧。这是一个非常好的校验机制，可以保证一个对象即使在网络或磁盘中“滚过”一次，仍能做到“出淤泥而不染”，完美地实现类的一致性。

但是，有时候我们需要一点特例场景，例如：我的类改变不大，JVM 是否可以把我以前的对象反序列化过来？就是依靠显式声明 `serialVersionUID`，向 JVM 撒谎说“我的类版本没有变更”，如此，我们编写的类就实现了向上兼容。我们修改一下上面的 `Person` 类，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 55799L;
    /* 其他保持不变 */
}
```

刚开始生产者和消费者持有的 `Person` 类版本一致，都是 V1.0，某天生产者的 `Person` 类版本变更了，增加了一个“年龄”属性，升级为 V2.0，而由于种种原因（比如程序员疏忽、升级时间窗口不同等）消费端的 `Person` 还保持为 V1.0 版本，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 5799L;
    private int age;
    /*age、name 的 getter/setter 方法省略 */
}
```

此时虽然生产者和消费者对应的类版本不同，但是显式声明的 `serialVersionUID` 相同，反序列化也是可以运行的，所带来的业务问题就是消费端不能读取到新增的业务属性（`age` 属性）而已。

通过此例，我们的反序列化实现了版本向上兼容的功能，使用 V1.0 版本的应用访问了一个 V2.0 版本的对象，这无疑提高了代码的健壮性。我们在编写序列化类代码时，随手加上 `serialVersionUID` 字段，也不会给我们带来太多的工作量，但它却可以在关键时候发挥异

乎寻常的作用。

注意 显式声明 serialVersionUID 可以避免对象不一致，但尽量不要以这种方式向 JVM “撒谎”。

建议 12：避免用序列化类在构造函数中为不变量赋值

我们知道带有 final 标识的属性是不变量，也就是说只能赋值一次，不能重复赋值，但是在序列化类中就有点复杂了，比如有这样一个类：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 71282334L;
    // 不变量
    public final String name="混世魔王";
}
```

这个 Person 类（此时 V1.0 版本）被序列化，然后存储在磁盘上，在反序列化时 name 属性会重新计算其值（这与 static 变量不同，static 变量压根就没有保存到数据流中），比如 name 属性修改成了“德天使”（版本升级为 V2.0），那么反序列化对象的 name 值就是“德天使”。保持新旧对象的 final 变量相同，有利于代码业务逻辑统一，这是序列化的基本规则之一，也就是说，如果 final 属性是一个直接量，在反序列化时就会重新计算。对这基本规则不多说，我们要说的是 final 变量另外一种赋值方式：通过构造函数赋值。代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 不变量初始不赋值
    public final String name;
    // 构造函数为不变量赋值
    public Person(){
        name="混世魔王";
    }
}
```

这也是我们常用的一种赋值方式，可以把这个 Person 类定义为版本 V1.0，然后进行序列化，看看有什么问题没有，序列化的代码如下所示：

```
public class Serialize {
    public static void main(String[] args) {
        // 序列化以持久保存
        SerializationUtils.writeObject(new Person());
    }
}
```

Person 的实例对象保存到了磁盘上，它是一个贫血对象（承载业务属性定义，但不包含其行为定义），我们做一个简单的模拟，修改一下 name 值代表变更，要注意的是

serialVersionUID 保持不变，修改后的代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 不变量初始不赋值
    public final String name;
    // 构造函数为不变量赋值
    public Person(){
        name="德天使";
    }
}
```

此时 Person 类的版本是 V2.0，但 serialVersionUID 没有改变，仍然可以反序列化，其代码如下：

```
public class Deserialize {
    public static void main(String[] args) {
        // 反序列化
        Person p = (Person)SerializationUtils.readObject();
        System.out.println(p.name);
    }
}
```

现在问题来了：打印的结果是什么？是混世魔王还是德天使？

答案即将揭晓，答案是：混世魔王。

final 类型的变量不是会重新计算吗？答案应该是“德天使”才对啊，为什么会是“混世魔王”？这是因为这里触及了反序列化的另一个规则：反序列化时构造函数不会执行。

反序列化的执行过程是这样的：JVM 从数据流中获取一个 Object 对象，然后根据数据流中的类文件描述信息（在序列化时，保存到磁盘的对象文件中包含了类描述信息，注意是类描述信息，不是类）查看，发现是 final 变量，需要重新计算，于是引用 Person 类中的 name 值，而此时 JVM 又发现 name 竟然没有赋值，不能引用，于是它很“聪明”地不再初始化，保持原值状态，所以结果就是“混世魔王”了。

读者不要以为这样的情况很少发生，如果使用 Java 开发过桌面应用，特别是参与过对性能要求较高的项目（比如交易类项目），那么很容易遇到这样的问题。比如一个 C/S 结构的在线外汇交易系统，要求提供 24 小时的联机服务，如果在升级的类中有一个 final 变量是构造函数赋值的，而且新旧版本还发生了变化，则在应用请求热切的过程中（非常短暂，可能只有 30 秒），很可能就会出现反序列化生成的 final 变量值与新产生的实例值不相同的情况，于是业务异常就产生了，情况严重的话甚至会影响交易数据，那可是天大的事故了。

注意 在序列化类中，不使用构造函数为 final 变量赋值。

建议 13：避免为 final 变量复杂赋值

为 final 变量赋值还有一种方式：通过方法赋值，即直接在声明时通过方法返回值赋值。还是以 Person 类为例来说明，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 通过方法返回值为 final 变量赋值
    public final String name=initName();
    // 初始化方法名
    public String initName(){
        return "混世魔王";
    }
}
```

name 属性是通过 initName 方法的返回值赋值的，这在复杂类中经常用到，这比使用构造函数赋值更简洁、易修改，那么如此用法在序列化时会不会有问题呢？我们一起来看看。Person 类写好了（定义为 V1.0 版本），先把它序列化，存储到本地文件，其代码与上一建议的 Serialize 类相同，不再赘述。

现在，Person 类的代码需要修改，initName 的返回值也改变了，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 通过方法返回值为 final 变量赋值
    public final String name=initName();
    // 初始化方法名
    public String initName(){
        return "德天使";
    }
}
```

上段代码仅仅修改了 initName 的返回值（Person 类为 V2.0 版本），也就是说通过 new 生成的 Person 对象的 final 变量值都是“德天使”。那么我们把之前存储在磁盘上的实例加载上来，name 值会是什么呢？

结果是：混世魔王。很诧异，上一建议说过 final 变量会被重新赋值，但是这个例子又没有重新赋值，为什么？

上个建议所说 final 会被重新赋值，其中的“值”指的是简单对象。简单对象包括：8 个基本类型，以及数组、字符串（字符串情况很复杂，不通过 new 关键字生成 String 对象的情况下，final 变量的赋值与基本类型相同），但是不能方法赋值。

其中的原理是这样的，保存到磁盘上（或网络传输）的对象文件包括两部分：

(1) 类描述信息

包括包路径、继承关系、访问权限、变量描述、变量访问权限、方法签名、返回值，以

及变量的关联类信息。要注意的一点是，它并不是 class 文件的翻版，它不记录方法、构造函数、static 变量等的具体实现。之所以类描述会被保存，很简单，是因为能去也能回嘛，这保证反序列化的健壮运行。

(2) 非瞬态 (transient 关键字) 和非静态 (static 关键字) 的实例变量值

注意，这里的值如果是一个基本类型，好说，就是一个简单值保存下来；如果是复杂对象，也简单，连该对象和关联类信息一起保存，并且持续递归下去（关联类也必须实现 Serializable 接口，否则会出现序列化异常），也就是说递归到最后，其实还是基本数据类型的保存。

正是因为这两点原因，一个持久化后的对象文件会比一个 class 类文件大很多，有兴趣的读者可以自己写个 Hello word 程序检验一下，其体积确实膨胀了不少。

总结一下，反序列化时 final 变量在以下情况下不会被重新赋值：

- 通过构造函数为 final 变量赋值。
- 通过方法返回值为 final 变量赋值。
- final 修饰的属性不是基本类型。

建议 14：使用序列化类的私有方法巧妙解决部分属性持久化问题

部分属性持久化问题看似很简单，只要把不需要持久化的属性加上瞬态关键字 (transient 关键字) 即可。这是一种解决方案，但有时候行不通。例如一个计税系统和人力资源系统 (HR 系统) 通过 RMI (Remote Method Invocation, 远程方法调用) 对接，计税系统需要从 HR 系统获得人员的姓名和基本工资，以作为纳税的依据，而 HR 系统的工资分为两部分：基本工资和绩效工资，基本工资没什么秘密，根据工作岗位和年限自己都可以计算出来，但绩效工资却是保密的，不能泄露到外系统，很明显这是两个相互关联的类。先来看薪水类 Salary 类的代码：

```
public class Salary implements Serializable{
    private static final long serialVersionUID = 44663L;
    // 基本工资
    private int basePay;
    // 绩效工资
    private int bonus;

    public Salary(int _basePay, int _bonus) {
        basePay = _basePay;
        bonus = _bonus;
    }
    /*getter/setter 方法省略*/
}
```

Person 类与 Salary 类是关联关系，代码如下：

```

public class Person implements Serializable{
    private static final long serialVersionUID = 60407L;
    // 姓名
    private String name;
    // 薪水
    private Salary salary;

    public Person(String _name, Salary _salary) {
        name = _name;
        salary = _salary;
    }
    /*getter/setter 方法省略 */
}

```

这是两个简单的 JavaBean，都实现了 Serializable 接口，都具备了持久化条件。首先计税系统请求 HR 系统对某一个 Person 对象进行序列化，把人员和工资信息传递到计税系统中，代码如下：

```

public class Serialize {
    public static void main(String[] args) {
        // 基本工资 1000 元，绩效工资 2500 元
        Salary salary = new Salary(1000, 2500);
        // 记录人员信息
        Person person = new Person("张三", salary);
        // HR 系统持久化，并传递到计税系统
        serializationUtils.writeObject(person);
    }
}

```

在通过网络传送到计税系统后，进行反序列化，代码如下：

```

public class Deserialize {
    public static void main(String[] args) {
        // 技术系统反序列化，并打印信息
        Person p = (Person) serializationUtils.readObject();
        StringBuffer sb = new StringBuffer();
        sb.append("姓名：" + p.getName());
        sb.append("\t基本工资：" + p.getSalary().getBasePay());
        sb.append("\t绩效工资：" + p.getSalary().getBonus());
        System.out.println(sb);
    }
}

```

打印出的结果很简单：

姓名：张三 基本工资：1000 绩效工资：2500。

但是这不符合需求，因为计税系统只能从 HR 系统中获得人员姓名和基本工资，而绩效工资是不能获得的，这是个保密数据，不允许发生泄露。怎么解决这个问题呢？你可能马上

会想到四种方案：

(1) 在 bonus 前加上 transient 关键字

这是一个方法，但不是一个好方法，加上 transient 关键字就标志着 Salary 类失去了分布式部署的功能，它可是 HR 系统最核心的类了，一旦遭遇性能瓶颈，想再实现分布式部署就不可能了，此方案否定。

(2) 新增业务对象

增加一个 Person4Tax 类，完全为计税系统服务，就是说它只有两个属性：姓名和基本工资。符合开闭原则，而且对原系统也没有侵入性，只是增加了工作量而已。这是个方法，但不是最优方法。

(3) 请求端过滤

在计税系统获得 Person 对象后，过滤掉 Salary 的 bonus 属性，方案可行但不合规矩，因为 HR 系统中的 Salary 类安全性竟然让外系统（计税系统）来承担，设计严重失职。

(4) 变更传输契约

例如改用 XML 传输，或者重建一个 Web Service 服务。可以做，但成本太高。

可能有读者会说了，你都在说别人的方案不好，你提供个优秀的方案看看！好的，这就展示一个优秀的方案。其中，实现了 Serializable 接口的类可以实现两个私有方法：writeObject 和 readObject，以影响和控制序列化和反序列化的过程。我们把 Person 类稍做修改，看看如何控制序列化和反序列化，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 60407L;
    // 姓名
    private String name;
    // 薪水
    private transient Salary salary;

    public Person(String _name, Salary _salary){
        name=_name;
        salary=_salary;
    }
    // 序列化委托方法
    private void writeObject(java.io.ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeInt(salary.getBasePay());
    }
    // 反序列化时委托方法
    private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        salary = new Salary(in.readInt(), 0);
    }
}
```

其他代码不做任何改动，我们先运行看看，结果为：

姓名：张三 基本工资：1000 绩效工资：0。

我们在 Person 类中增加了 writeObject 和 readObject 两个方法，并且访问权限都是私有级别，为什么这会改变程序的运行结果呢？其实这里使用了序列化独有的机制：序列化回调。Java 调用 ObjectOutputStream 类把一个对象转换成流数据时，会通过反射（Reflection）检查被序列化的类是否有 writeObject 方法，并且检查其是否符合私有、无返回值的特性。若有，则会委托该方法进行对象序列化，若没有，则由 ObjectOutputStream 按照默认规则继续序列化。同样，在从流数据恢复成实例对象时，也会检查是否有一个私有的 readObject 方法，如果有，则会通过该方法读取属性值。此处有几个关键点要说明：

(1) out.defaultWriteObject()

告知 JVM 按照默认的规则写入对象，惯例的写法是写在第一句话里。

(2) in.defaultReadObject()

告知 JVM 按照默认规则读入对象，惯例的写法也是写在第一句话里。

(3) out.writeXX 和 in.readXX

分别是写入和读出相应的值，类似一个队列，先进先出，如果此处有复杂的数据逻辑，建议按封装 Collection 对象处理。

可能有读者会提出，这似乎不是一种优雅的处理方案呀，为什么 JDK 没有对此提供一个更好的解决办法呢？比如访问者模式，或者设置钩子函数（Hook），完全可以更优雅地解决此类问题。我查阅了大量的文档，得出的结论是：无解，只能说这是一个可行的解决方案而已。

再回到我们的业务领域，通过上述方法重构后，其代码的修改量减少了许多，也优雅了许多。可能你又要反问了：如此一来，Person 类也失去了分布式部署的能力啊。确实是，但是 HR 系统的难点和重点是薪水计算，特别是绩效工资，它所依赖的参数很复杂（仅从数量上说就有上百甚至上千种），计算公式也不简单（一般是引入脚本语言，个性化公式定制），而相对来说 Person 类基本上都是“静态”属性，计算的可能性不大，所以即使为性能考虑，Person 类为分布式部署的意义也不大。

建议 15：break 万万不可忘

我们经常会写一些转换类，比如货币转换、日期转换、编码转换等，在金融领域里用到最多的要数中文数字转换了，比如把“1”转换为“壹”，不过，开源世界是不会提供此工具类的，因为它太贴合中国文化了，要转换还是得自己动手写，代码片段如下：

```

public class Client {
    public static void main(String[] args) {
        System.out.println("2 = "+toChineseNumberCase(2));
    }

    // 把阿拉伯数字翻译成中文大写数字
    public static String toChineseNumberCase(int n) {
        String chineseNumber = "";
        switch (n) {
            case 0:chineseNumber = "零";
            case 1:chineseNumber = "壹";
            case 2:chineseNumber = "贰";
            case 3:chineseNumber = "叁";
            case 4:chineseNumber = "肆";
            case 5:chineseNumber = "伍";
            case 6:chineseNumber = "陆";
            case 7:chineseNumber = "柒";
            case 8:chineseNumber = "捌";
            case 9:chineseNumber = "玖";
        }
        return chineseNumber;
    }
}

```

这是一个简单的转换类，并没有完整实现，只是一个金融项目片段。如此简单的代码应该不会有错吧，我们运行看看，结果是：2 = 玖。

恩？错了？回头再来看程序，马上领悟了：每个 case 语句后面少加了 break 关键字。程序从“case 2”后面的语句开始执行，直到找到最近的 break 语句结束，但可惜的是我们的程序中没有 break 语句，于是在程序执行的过程中，chineseNumber 的赋值语句会多次执行，会从等于“贰”、等于“叁”、等于“肆”，一直变换到等于“玖”，switch 语句执行结束了，于是结果也就如此了。

此类问题发生得非常频繁，但也很容易发现，只要做一下单元测试（Unit Test），问题立刻就会被发现并解决掉，但如果是在一堆的 case 语句中，其中某一条漏掉了 break 关键字，特别是在单元测试覆盖率不够高的时候（为什么不够高？在大点的项目中蹲过坑、打过仗的兄弟们可能都知道，项目质量是与项目工期息息相关的，而项目工期往往不是由项目人员决定的，所以如果一个项目的单元测试覆盖率能够达到 60%，你就可以笑了），也就是说分支条件可能覆盖不到的时候，那就会在生产中出现大事故了。

我曾遇到过一个类似的事，那是开发一个通过会员等级决定相关费率的系统，由于会员等级有 100 多个，所以测试时就采用了抽样测试的方法，测试时一切顺利，直到系统上线后，财务报表系统发现一个小概率的会员费率竟然出奇的低，于是就跟踪分析，发现是少了一个 break，此事不仅造成甲方经济上的损失，而且在外部也产生了不良的影响，最后该代码的作者被辞退了，测试人员、质量负责人、项目经理都做了相应的处罚。希望读者能引以

为戒，记住在 case 语句后面随手写上 break，养成良好的习惯。

对于此类问题，还有一个最简单的解决办法：修改 IDE 的警告级别，例如在 Eclipse 中，可以依次点击 Performaces → Java → Compiler → Errors/Warnings → Potential Programming problems，然后修改 ‘switch’ case fall-through 为 Errors 级别，如果你胆敢不在 case 语句中加入 break，那 Eclipse 直接就报个红叉给你看，这样就可以完全避免该问题的发生了。

建议 16：易变业务使用脚本语言编写

Java 世界一直在遭受着异种语言的入侵，比如 PHP、Ruby、Groovy、JavaScript 等，这些“入侵者”都有一个共同特征：全是同一类语言——脚本语言，它们都是在运行期解释执行的。为什么 Java 这种强编译型语言会需要这些脚本语言呢？那是因为脚本语言的三大特征，如下所示：

- 灵活。脚本语言一般都是动态类型，可以不用声明变量类型而直接使用，也可以在运行期改变类型。
- 便捷。脚本语言是一种解释型语言，不需要编译成二进制代码，也不需要像 Java 一样生成字节码。它的执行是依靠解释器解释的，因此在运行期变更代码非常容易，而且不用停止应用。
- 简单。只能说部分脚本语言简单，比如 Groovy，Java 程序员若转到 Groovy 程序语言上，只需要两个小时，看完语法说明，看完 Demo 即可使用了，没有太多的技术门槛。

脚本语言的这些特性是 Java 所缺少的，引入脚本语言可以使 Java 更强大，于是 Java 6 开始正式支持脚本语言。但是因为脚本语言比较多，Java 的开发者也很难确定该支持哪种语言，于是 JCP (Java Community Process) 很聪明地提出了 JSR223 规范，只要符合该规范的语言都可以在 Java 平台上运行（它对 JavaScript 是默认支持的），诸位读者有兴趣的话可以自己写个脚本语言，然后再实现 ScriptEngine，即可在 Java 平台上运行。

我们来分析一个案例，展现一下脚本语言是如何实现“拥抱变化”的。咱们编写一套模型计算公式，预测下一个工作日的股票走势（如果真有，那巴菲特就羞愧死了），即把国家政策、汇率、利率、地域系数等参数输入到公式中，然后计算出明天这支股票是涨还是跌，该公式是依靠历史数据推断而来的，会根据市场环境逐渐优化调整，也就是逐渐趋向“真理”的过程，在此过程中，公式经常需要修改（这里的修改不仅仅是参数修改，还涉及公式的算法修改），如果把这个公式写到一个类中（或者几个类中），就需要经常发布重启等操作（比如业务中断，需要冒烟测试（Smoke Testing）等），使用脚本语言则可以很好地简化这一过程，我们写一个简单公式来模拟一下，代码如下：

```
function formula(var1,var2){
    return var1 + var2 * factor;
}
```

这就是一个简单的脚本语言函数，可能你会很疑惑：factor（因子）这个变量是从哪儿来的？它是从上下文来的，类似于一个运行的环境变量。该 JavaScript 保存在 C:/model.js 中。下一步 Java 需要调用 JavaScript 公式，代码如下：

```
public static void main(String[] args) throws Exception {
    // 获得一个 JavaScript 的执行引擎
    ScriptEngine engine=new ScriptEngineManager().getEngineByName("javascript");
    // 建立上下文变量
    Bindings bind=engine.createBindings();
    bind.put("factor", 1);
    // 绑定上下文，作用域是当前引擎范围
    engine.setBindings(bind,ScriptContext.ENGINE_SCOPE);
    Scanner input = new Scanner(System.in);
    while(input.hasNextInt()){
        int first = input.nextInt();
        int sec = input.nextInt();
        System.out.println(" 输入参数是: "+first+", "+sec);
        // 执行 js 代码
        engine.eval(new FileReader("c:/model.js"));
        // 是否可调用方法
        if(engine instanceof Invocable){
            Invocable in=(Invocable)engine;
            // 执行 js 中的函数
            Double result = (Double)in.invokeFunction("formula",first,sec);
            System.out.println(" 运算结果: "+result.intValue());
        }
    }
}
```

上段代码使用 Scanner 类接受键盘输入的两个数字，然后调用 JavaScript 脚本的 formula 函数计算其结果，注意，除非输入了一个非 int 数字，否则当前 JVM 会一直运行，这也是模拟生产系统的在线变更状况。运行结果如下：

```
输入参数是: 1,2
运算结果: 3
```

此时，保持 JVM 的运行状态，我们修改一下 formula 函数，代码如下：

```
function formula(var1,var2){
    return var1 + var2 - factor;
}
```

其中，乘号变成了减号，计算公式发生了重大改变。回到 JVM 中继续输入，运行结果如下。

```
输入参数是: 1,2
运算结果: 2
```

修改 Java 代码，JVM 没有重启，输入参数也没有任何改变，仅仅改变脚本函数即可产生不同的结果。这就是脚本语言对系统设计最有利的地方：可以随时发布而不用重新部署；这也是我们 Javaer 最喜爱它的地方——即使进行变更，也能提供不间断的业务服务。

Java 6 不仅仅提供了代码级的脚本内置，还提供了一个 jrunscript 命令工具，它可以在批处理中发挥最大效能，而且不需要通过 JVM 解释脚本语言，可以直接通过该工具运行脚本。想想看，这是多么大的诱惑力呀！而且这个工具是可以跨操作系统的，脚本移植就更容易了。但是有一点需要注意：该工具是实验性的，在以后的 JDK 中会不会继续提供就很难说了。

建议 17：慎用动态编译

动态编译一直是 Java 的梦想，从 Java 6 版本它开始支持动态编译了，可以在运行期直接编译 .java 文件，执行 .class，并且能够获得相关的输入输出，甚至还能监听相关的事件。不过，我们最期望的还是给定一段代码，直接编译，然后运行，也就是空中编译执行（on-the-fly），来看如下代码：

```
public class Client {
    public static void main(String[] args) throws Exception {
        //Java 源代码
        String sourceStr = "public class Hello{public String sayHello (String name)
            {return \"Hello,\" + name + \"!\";}}";
        //类名及文件名
        String clsName = "Hello";
        //方法名
        String methodName = "sayHello";
        //当前编译器
        JavaCompiler cmp = ToolProvider.getSystemJavaCompiler();
        //Java 标准文件管理器
        StandardJavaFileManager fm = cmp.getStandardFileManager(null,null,null);
        //Java 文件对象
        JavaFileObject jfo = new StringJavaObject(clsName,sourceStr);
        //编译参数，类似于 javac <options> 中的 options
        List<String> optionsList = new ArrayList<String>();
        //编译文件的存放地方，注意：此处是为 Eclipse 工具特设的
        optionsList.addAll(Arrays.asList("-d","./bin"));
        //要编译的单元
        List<JavaFileObject> jfos = Arrays.asList(jfo);
        //设置编译环境
        JavaCompiler.CompilationTask task = cmp.getTask(null, fm, null,
            optionsList,null,jfos);
        //编译成功
        if(task.call()) {
```

```

        // 生成对象
        Object obj = Class.forName(clsName).newInstance();
        Class<? extends Object> cls = obj.getClass();
        // 调用 sayHello 方法
        Method m = cls.getMethod(methodName, String.class);
        String str = (String) m.invoke(obj, "Dynamic Compilation");
        System.out.println(str);
    }
}

// 文本中的 Java 对象
class StringJavaObject extends SimpleJavaFileObject{
    // 源代码
    private String content = "";
    // 遵循 Java 规范的类名及文件
    public StringJavaObject(String _javaFileName, String _content){
        super(_createStringJavaObjectUri(_javaFileName), Kind.SOURCE);
        content = _content;
    }
    // 产生一个 URL 资源路径
    private static URI _createStringJavaObjectUri(String name){
        // 注意此处没有设置包名
        return URI.create("String://" + name + Kind.SOURCE.extension);
    }
    // 文本文件代码
    @Override
    public CharSequence getCharContent(boolean ignoreEncodingErrors)
        throws IOException {
        return content;
    }
}

```

上面的代码较多，这是一个动态编译的模板程序，读者可以拷贝到项目中使用，代码中的中文注释也较多，相信读者看得懂，不多解释，读者只要明白一件事：只要是在本地静态编译能够实现的任务，比如编译参数、输入输出、错误监控等，动态编译就都能实现。

Java 的动态编译对源提供了多个渠道。比如，可以是字符串（例子中就是字符串），可以是文本文件，也可以是编译过的字节码文件 (.class 文件)，甚至可以是存放在数据库中的明文代码或是字节码。汇总成一句话，只要是符合 Java 规范的就都可以在运行期动态加载，其实现方式就是实现 JavaFileObject 接口，重写 getCharContent、openInputStream、openOutputStream，或者实现 JDK 已经提供的两个 SimpleJavaFileObject、ForwardingJavaFileObject，具体代码可以参考上个例子。

动态编译虽然是很好的工具，让我们可以更加自如地控制编译过程，但是在我目前所接触的项目中还是使用得较少。原因很简单，静态编译已经能够帮我们处理大部分的工作，甚至是全部的工作，即使真的需要动态编译，也有很好的替代方案，比如 JRuby、Groovy 等无缝的脚本语言。

另外，我们在使用动态编译时，需要注意以下几点：

(1) 在框架中谨慎使用

比如要在 Struts 中使用动态编译，动态实现一个类，它若继承自 ActionSupport 就希望它成为一个 Action。能做到，但是 debug 很困难；再比如在 Spring 中，写一个动态类，要让它动态注入到 Spring 容器中，这是需要花费老大功夫的。

(2) 不要在要求高性能的项目使用

动态编译毕竟需要一个编译过程，与静态编译相比多了一个执行环节，因此在高性能项目中不要使用动态编译。不过，如果是在工具类项目中它则可以很好地发挥其优越性，比如在 Eclipse 工具中写一个插件，就可以很好地使用动态编译，不用重启即可实现运行、调试功能，非常方便。

(3) 动态编译要考虑安全问题

如果你在 Web 界面上提供了一个功能，允许上传一个 Java 文件然后运行，那就等于说：“我的机器没有密码，大家都来看我的隐私吧”，这是非常典型的注入漏洞，只要上传一个恶意 Java 程序就可以让你所有的安全工作毁于一旦。

(4) 记录动态编译过程

建议记录源文件、目标文件、编译过程、执行过程等日志，不仅仅是为了诊断，还是为了安全和审计，对 Java 项目来说，空中编译和运行是很不让人放心的，留下这些依据可以更好地优化程序。

建议 18：避免 instanceof 非预期结果

instanceof 是一个简单的二元操作符，它是用来判断一个对象是否是一个类实例的，其操作类似于 >=、==，非常简单，我们来看段程序，代码如下：

```
public class Client {
    public static void main(String[] args) {
        //String 对象是否是 Object 的实例
        boolean b1 = "String" instanceof Object;
        //String 对象是否是 String 的实例
        boolean b2 = new String() instanceof String;
        //Object 对象是否是 String 的实例
        boolean b3 = new Object() instanceof String;
        //拆箱类型是否是装箱类型的实例
        boolean b4 = 'A' instanceof Character;
        //空对象是否是 String 的实例
        boolean b5 = null instanceof String;
        //类型转换后的空对象是否是 String 的实例
        boolean b6 = (String)null instanceof String;
        //Date 对象是否是 String 的实例
    }
}
```

```

        boolean b7 = new Date() instanceof String;
        // 在泛型类中判断 String 对象是否是 Date 的实例
        boolean b8 = new GenericClass<String>().isDateInstance("");
    }
}

class GenericClass<T>{
    // 判断是否是 Date 类型
    public boolean isDateInstance(T t) {
        return t instanceof Date;
    }
}

```

就这么一段程序，`instanceof`的所有应用场景都出现了，同时问题也产生了：这段程序中哪些语句会编译通不过？我们一个一个地来解说。

□ "String" instanceof Object

返回值是 `true`，这很正常，“String”是一个字符串，字符串又继承了 `Object`，那当然是返回 `true` 了。

□ `new String()` instanceof String

返回值是 `true`，没有任何问题，一个类的对象当然是它的实例了。

□ `new Object()` instanceof String

返回值是 `false`，`Object` 是父类，其对象当然不是 `String` 类的实例了。要注意的是，这句话其实完全可以编译通过，只要 `instanceof` 关键字的左右两个操作数有继承或实现关系，就可以编译通过。

□ 'A' instanceof Character

这句话可能有读者会猜错，事实上它编译不通过，为什么呢？因为 '`A`' 是一个 `char` 类型，也就是一个基本类型，不是一个对象，`instanceof` 只能用于对象的判断，不能用于基本类型的判断。

□ null instanceof String

返回值是 `false`，这是 `instanceof` 特有的规则：若左操作数是 `null`，结果就直接返回 `false`，不再运算右操作数是什么类。这对我们的程序非常有利，在使用 `instanceof` 操作符时，不用关心被判断的类（也就是左操作数）是否为 `null`，这与我们经常用到的 `equals`、`toString` 方法不同。

□ (String)null instanceof String

返回值是 `false`，不要看这里有个强制类型转换就认为结果是 `true`，不是的，`null` 是一个万用类型，也可以说它没类型，即使做类型转换还是个 `null`。

□ `new Date()` instanceof String

编译通不过，因为 `Date` 类和 `String` 没有继承或实现关系，所以在编译时直接就报错了；

`instanceof` 操作符的左右操作数必须有继承或实现关系，否则编译会失败。

□ `new GenericClass<String>().isDateInstance("")`

编译通不过？非也，编译通过了，返回值是 `false`，`T` 是个 `String` 类型，与 `Date` 之间没有继承或实现关系，为什么 "`t instanceof Date`" 会编译通过呢？那是因为 Java 的泛型是为编码服务的，在编译成字节码时，`T` 已经是 `Object` 类型了，传递的实参是 `String` 类型，也就是说 `T` 的表面类型是 `Object`，实际类型是 `String`，那 "`t instanceof Date`" 这句话就等价于 "`Object instance of Date`" 了，所以返回 `false` 就很正常了。

就这么一个简单的 `instanceof`，你答对几个？

建议 19：断言绝对不是鸡肋

在防御式编程中经常会用断言（Assertion）对参数和环境做出判断，避免程序因不当的输入或错误的环境而产生逻辑异常，断言在很多语言中都存在，C、C++、Python 都有不同的断言表示形式。在 Java 中的断言使用的是 `assert` 关键字，其基本的用法如下：

```
assert <布尔表达式>
assert <布尔表达式> : <错误信息>
```

在布尔表达式为假时，抛出 `AssertionError` 错误，并附带了错误信息。`assert` 的语法较简单，有以下两个特性：

(1) `assert` 默认是不启用的

我们知道断言是为调试程序服务的，目的是为了能够快速、方便地检查到程序异常，但 Java 在默认条件下是不启用的，要启用就需要在编译、运行时加上相关的关键字，这就不多说，有需要的话可以参考一下 Java 规范。

(2) `assert` 抛出的异常 `AssertionError` 是继承自 `Error` 的

断言失败后，JVM 会抛出一个 `AssertionError` 错误，它继承自 `Error`，注意，这是一个错误，是不可恢复的，也就表示这是一个严重问题，开发者必须予以关注并解决之。

`assert` 虽然是做断言的，但不能将其等价于 `if...else...` 这样的条件判断，它在以下两种情况不可使用：

(1) 在对外公开的方法中

我们知道防御式编程最核心的一点就是：所有的外部因素（输入参数、环境变量、上下文）都是“邪恶”的，都存在着企图摧毁程序的罪恶本源，为了抵制它，我们要在程序中处处检验，满地设卡，不满足条件就不再执行后续程序，以保护主程序的正确性，处处设卡没问题，但就是不能用断言做输入校验，特别是公开方法。我们来看一个例子：

```
public class Client {
```

```

public static void main(String[] args) {
    StringUtils.encode(null);
}
// 字符串处理工具类
class StringUtils{
    public static String encode(String str){
        assert str!=null:"加密的字符串为 null";
        /* 加密处理 */
    }
}

```

`encode` 方法对输入参数做了不为空的假设，如果为空，则抛出 `AssertionError` 错误，但这段程序存在一个严重的问题，`encode` 是一个 `public` 方法，这标志着是它对外公开的，任何一个类只要能够传递一个 `String` 类型的参数（遵守契约）就可以调用，但是 `Client` 类按照规范和契约调用 `enocde` 方法，却获得了一个 `AssertionError` 错误信息，是谁破坏了契约协定？——是 `encode` 方法自己。

(2) 在执行逻辑代码的情况下

`assert` 的支持是可选的，在开发时可以让它运行，但在生产系统中则不需要其运行了（以便提高性能），因此在 `assert` 的布尔表达式中不能执行逻辑代码，否则会因为环境不同而产生不同的逻辑，例如：

```

public void doSomething(List list, Object element){
    assert list.remove(element) :"删除元素 " + element + " 失败";
    /* 业务处理 */
}

```

这段代码在 `assert` 启用的环境下，没有任何问题，但是一旦投入到生产环境，就不会启用断言了，而这个方法也就彻底完蛋了，`list` 的删除动作永远都不会执行，所以也就永远不会有报错或异常，因为根本就没有执行嘛！

以上两种情况下不能使用 `assert`，那在什么情况下能够使用 `assert` 呢？一句话：按照正常执行逻辑不可能到达的代码区域可以放置 `assert`。具体分为三种情况：

(1) 在私有方法中放置 `assert` 作为输入参数的校验

在私有方法中可以放置 `assert` 校验输入参数，因为私有方法的使用者是作者自己，私有方法的调用者和被调用者之间是一种弱契约关系，或者说没有契约关系，其间的约束是依靠作者自己控制的，因此加上 `assert` 可以更好地预防自己犯错，或者无意的程序犯错。

(2) 流程控制中不可能达到的区域

这类似于 JUnit 的 `fail` 方法，其标志性的意义就是：程序执行到这里就是错误的，例如：

```

public void doSomething(){
    int i = 7;
    while(i >7){

```

```

    /* 业务处理 */
}
assert false:"到达这里就表示错误";
}

```

(3) 建立程序探针

我们可能会在一段程序中定义两个变量，分别代表两个不同的业务含义，但是两者有固定的关系，例如 var1=var2*2，那我们就可以在程序中到处设“桩”，断言这两者的关系，如果不满足即表明程序已经出现了异常，业务也就没有必要运行下去了。

建议 20：不要只替换一个类

我们经常在系统中定义一个常量接口（或常量类），以囊括系统中所涉及的常量，从而简化代码，方便开发，在很多的开源项目中已采用了类似的方法，比如在 Struts2 中，org.apache.struts2.StrutsConstants 就是一个常量类，它定义了 Struts 框架中与配置有关的常量，而 org.apache.struts2.StrutsStatics 则是一个常量接口，其中定义了 OGNL 访问的关键字。

关于常量接口（类）我们来看一个例子，首先定义一个常量类：

```

public class Constant {
    // 定义人类寿命极限
    public final static int MAX_AGE = 150;
}

```

这是一个非常简单的常量类，定义了人类的最大年龄，我们引用这个常量，代码如下：

```

public class Client {
    public static void main(String[] args) {
        System.out.println("人类寿命极限是：" + Constant.MAX_AGE);
    }
}

```

运行的结果非常简单（结果省略）。目前的代码编写都是在“智能型”IDE 工具中完成的，下面我们暂时回溯到原始时代，也就是回归到用记事本编写代码的年代，然后看看会发生什么奇妙事情（为什么要如此，稍后会给出答案）。

修改常量 Constant 类，人类的寿命增加了，最大能活到 180 岁，代码如下：

```

public class Constant {
    // 定义人类寿命极限
    public final static int MAX_AGE = 180;
}

```

然后重新编译：javac Constant，编译完成后执行：java Client，大家想看看输出的极限年龄是多少岁吗？

输出的结果是：“人类寿命极限是：150”，竟然没有改变为 180，太奇怪了，这是为何？

原因是：对于 final 修饰的基本类型和 String 类型，编译器会认为它是稳定态（Immutable Status），所以在编译时就直接把值编译到字节码中了，避免了在运行期引用（Run-time Reference），以提高代码的执行效率。针对我们的例子来说，Client 类在编译时，字节码中就写上了“150”这个常量，而不是一个地址引用，因此无论你后续怎么修改常量类，只要不重新编译 Client 类，输出还是照旧。

而对于 final 修饰的类（即非基本类型），编译器认为它是不稳定态（Mutable Status），在编译时建立的则是引用关系（该类型也叫做 Soft Final），如果 Client 类引入的常量是一个类或实例，即使不重新编译也会输出最新值。

千万不可小看了这点知识，细坑也能绊倒大象，比如在一个 Web 项目中，开发人员修改一个 final 类型的值（基本类型），考虑到重新发布风险较大，或者是时间较长，或者是审批流程过于繁琐，反正是为了偷懒，于是直接采用替换 class 类文件的方式发布。替换完毕后应用服务器自动重启，然后简单测试一下（比如本类引用 final 类型的常量），一切 OK。可运行几天后发现业务数据对不上，有的类（引用关系的类）使用了旧值，有的类（继承关系的类）使用的是新值，而且毫无头绪，让人一筹莫展，其实问题的根源就在于此。

恩，还有个小问题没有说明，我们的例子为什么不在 IDE 工具（比如 Eclipse）中运行呢？那是因为在 IDE 中不能重现该问题，若修改了 Constant 类，IDE 工具会自动编译所有的引用类，“智能”化屏蔽了该问题，但潜在的风险其实仍然存在。

注意 发布应用系统时禁止使用类文件替换方式，整体 WAR 包发布才是万全之策。



第2章

基本类型

不积跬步，无以至千里；
不积小流，无以成江海。

——荀子《劝学篇》

Java 中的基本数据类型（Primitive Data Types）有 8 个：byte、char、short、int、long、float、double、boolean，它们是 Java 最基本的单元，我们的每一段程序中都有它们的身影，但我们对如此熟悉的“伙伴”又了解多少呢？

积少成多，积土成山，本章我们就来一探这最基本的 8 个数据类型。

建议 21：用偶判断，不用奇判断

判断一个数是奇数还是偶数是小学里学的基本知识，能够被 2 整除的整数是偶数，不能被 2 整除的是奇数，这规则简单又明了，还有什么好考虑的？好，我们来看一个例子，代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 接收键盘输入参数
        Scanner input = new Scanner(System.in);
        System.out.print("请输入多个数字判断奇偶: ");
        while(input.hasNextInt()){
            int i = input.nextInt();
            String str = i + " -> " + (i%2 == 1?" 奇数 ":" 偶数 ");
            System.out.println(str);
        }
    }
}
```

输入多个数字，然后判断每个数字的奇偶性，不能被 2 整除就是奇数，其他的都是偶数，完全是根据奇偶数的定义编写的程序，我们来看看打印的结果：

```
请输入多个数字判断奇偶: 1 2 0 -1 -2
1-> 奇数
2-> 偶数
0-> 偶数
-1-> 偶数
-2-> 偶数
```

前三个还很靠谱，第四个参数 -1 怎么可能会是偶数呢，这 Java 也太差劲了，如此简单的计算也会错！别忙着下结论，我们先来了解一下 Java 中的取余（% 标示符）算法，模拟代码如下：

```
// 模拟取余计算，dividend 被除数，divisor 除数
public static int remainder(int dividend,int divisor){
    return dividend - dividend / divisor * divisor;
}
```

看到这段程序，相信大家都会心地笑了，原来 Java 是这么处理取余计算的呀。根据上面

的模拟取余可知，当输入 -1 的时候，运算结果是 -1，当然不等于 1 了，所以它就被判定为偶数了，也就是说是我们的判断失误了。问题明白了，修正也很简单，改为判断是否是偶数即可，代码如下：

```
i%2 ==0?"偶数 ":"奇数 "
```

注意 对于基础知识，我们应该“知其然，并知其所以然”。

建议 22：用整数类型处理货币

在日常生活中，最容易接触到的小数就是货币，比如你付给售货员 10 元钱购买一个 9.60 元的零食，售货员应该找你 0.4 元也就是 4 毛钱才对，我们来看下面的程序：

```
public class Client {
    public static void main(String[] args) {
        System.out.println(10.00-9.60);
    }
}
```

我们期望的结果是 0.4，也应该是这个数字，但是打印出来的却是 0.40000000000000036，这是为什么呢？

这是因为在计算机中浮点数有可能（注意是可能）是不准确的，它只能无限接近准确值，而不能完全精确。为什么会如此呢？这是由浮点数的存储规则所决定的，我们先来看 0.4 这个十进制小数如何转换成二进制小数，使用“乘 2 取整，顺序排列”法（不懂？这就没招了，太基础了），我们发现 0.4 不能使用二进制准确的表示，在二进制数世界里它是一个无限循环的小数，也就是说，“展示”都不能“展示”，更别说是内存中存储了（浮点数的存储包括三部分：符号位、指数位、尾数，具体不再介绍），可以这样理解，在十进制的世界里没有办法准确表示 1/3，那在二进制世界里当然也无法准确表示 1/5（如果二进制也有分数的话倒是可以表示），在二进制的世界里 1/5 是一个无限循环小数。

各位要说了，那我对结果取整不就对了吗？代码如下：

```
public class Client {
    public static void main(String[] args) {
        NumberFormat f = new DecimalFormat("#.##");
        System.out.println(f.format(10.00-9.60));
    }
}
```

打印出结果是 0.4，看似解决了，但是隐藏了一个很深的问题。我们来思考一下金融行业的计算方法，会计系统一般记录小数点后的 4 位小数，但是在汇总、展现、报表中，则只记录小数点后的 2 位小数，如果使用浮点数来计算货币，想想看，在大批量的加减乘除后结

果会有多大的差距（其中还涉及后面会讲到的四舍五入问题）！会计系统要的就是准确，但是却因为计算机的缘故不准确了，那真是罪过。要解决此问题有两种方法：

(1) 使用 BigDecimal

BigDecimal 是专门为弥补浮点数无法精确计算的缺憾而设计的类，并且它本身也提供了加减乘除的常用数学算法。特别是与数据库 Decimal 类型的字段映射时，BigDecimal 是最优的解决方案。

(2) 使用整型

把参与运算的值扩大 100 倍，并转变为整型，然后在展现时再缩小 100 倍，这样处理的好处是计算简单、准确，一般在非金融行业（如零售行业）应用较多。此方法还会用于某些零售 POS 机，它们的输入和输出全部是整数，那运算就更简单。

建议 23：不要让类型默默转换

我们出一个小学题给大家做做看，光速是每秒 30 万公里，根据光线旅行的时间，计算月亮与地球、太阳与地球之间的距离。代码如下：

```
public class Client {
    // 光速是 30 万公里 / 秒，常量
    public static final int LIGHT_SPEED = 30 * 10000 * 1000;
    public static void main(String[] args) {
        System.out.println("题目 1：月亮光照射到地球需要 1 秒，计算月亮和地球的距离。");
        long dis1 = LIGHT_SPEED * 1;
        System.out.println("月亮与地球的距离是：" + dis1 + " 米");
        System.out.println("-----");
        System.out.println("题目 2：太阳光照射到地球上需要 8 分钟，计算太阳到地球的距离。");
        // 可能要超出整数范围，使用 long 型
        long dis2 = LIGHT_SPEED * 60 * 8;
        System.out.println("太阳与地球的距离是：" + dis2 + " 米");
    }
}
```

估计你要鄙视了，这种小学生乘法计算有什么可做的。不错，确实就是一个乘法运算，我们运行一下看看结果：

题目 1：月亮光照射到地球需要 1 秒，计算月亮和地球的距离。

月亮与地球的距离是：300000000 米

题目 2：太阳光照射到地球上需要 8 分钟，计算太阳到地球的距离。

太阳与地球的距离是：-2028888064 米

太阳和地球的距离竟然是负的，诡异。dis2 不是已经考虑到 int 类型可能越界的问题，并使用了 long 型吗，为什么还会出现负值呢？

那是因为 Java 是先运算然后再进行类型转换的，具体地说就是因为 `disc2` 的三个运算参数都是 `int` 类型，三者相乘的结果虽然也是 `int` 类型，但是已经超过了 `int` 的最大值，所以其值就是负值了（为什么是负值？因为过界了就会从头开始），再转换成 `long` 型，结果还是负值。

问题知道了，解决起来也很简单，只要加个小小的“L”即可，代码如下：

```
long dis2 = LIGHT_SPEED * 60L * 8;
```

`60L` 是一个长整型，乘出来的结果也是一个长整型（此乃 Java 的基本转换规则，向数据范围大的方向转换，也就是加宽类型），在还没有超过 `int` 类型的范围时就已经转换为 `long` 型了，彻底解决了越界问题。在实际开发中，更通用的做法是主动声明式类型转化（注意不是强制类型转换），代码如下：

```
long dis2 = 1L * LIGHT_SPEED * 60 * 8;
```

既然期望的结果是 `long` 型，那就让第一个参与运算的参数也是 `long` 型（`1L`）吧，也就是说“嗨，我已经是长整型了，你们都跟着我一起转为长整型吧”。

注意 基本类型转换时，使用主动声明方式减少不必要的 Bug。

建议 24：边界，边界，还是边界

某商家生产的电子产品非常畅销，需要提前 30 天预订才能抢到手，同时它还规定了一个会员可拥有的最多产品数量，目的是防止囤积压货肆意加价。会员的预定过程是这样的：先登录官方网站，选择产品型号，然后设置需要预定的数量，提交，符合规则即提示下单成功，不符合规则提示下单失败。后台的处理逻辑模拟如下：

```
public class Client {
    // 一个会员拥有的最多数量
    public final static int LIMIT = 2000;
    public static void main(String[] args) {
        // 会员当前拥有的产品数量
        int cur = 1000;
        Scanner input = new Scanner(System.in);
        System.out.print("请输入需要预定的数量: ");
        while(input.hasNextInt()){
            int order = input.nextInt();
            // 当前拥有的与准备订购的产品数量之和
            if(order>0 && order+cur<=LIMIT){
                System.out.println("你已经成功预定的 "+order+" 个产品! ");
            }else{
        }}
```

```
        System.out.println("超过限额，预订失败！");  
    }  
}
```

这是一个简易的订单处理程序，其中 cur 代表的是会员已经拥有的产品数量，LIMIT 是一个会员最多拥有的产品数量（现实中这两个参数当然是从数据库中获得的，不过这里是一个模拟程序），如果当前预订数量与拥有数量之和超过了最大数量，则预订失败，否则下单成功。业务逻辑很简单，同时在 Web 界面上对订单数量做了严格的校验，比如不能是负值、不能超过最大数量等，但是人算不如天算，运行不到两小时数据库中就出现了异常数据：某会员拥有的产品的数量与预订数量之和远远大于限额。怎么会这样？程序逻辑上不可能有问题呀，这是如何产生的呢？我们来模拟一下，第一次输入：

请输入需要预定的数量：800
你已经成功预定的800个产品！

这完全满足条件，没有任何问题，继续输入：

请输入需要预定的数量: 2147483647
你已经成功预定的 2147483647 个产品

看到没，这个数字远远超过了 2000 的限额，但是竟然预订成功了，真是神奇！

看着 2147483647 这个数字很眼熟？那就对了，它是 int 类型的最大值，没错，有人输入了一个最大值，使校验条件失效了，Why？我们来看程序，order 的值是 2147483647，那再加上 1000 就超出 int 的范围了，其结果是 -2147482649，那当然是小于正数 2000 了！一句话可归结其原因：数字越界使检验条件失效。

在单元测试中，有一项测试叫做边界测试（也有叫做临界测试），如果一个方法接收的是 int 类型的参数，那以下三个值是必测的：0、正最大、负最小，其中正最大和负最小是边界值，如果这三个值都没有问题，方法才是比较安全可靠的。我们的例子就是因为缺少边界测试，致使生产系统产生了严重的偏差。

也许你要疑惑了，Web 界面既然已经做了严格的校验，为什么还能输入 2147483647 这么大的数字呢？是否说明 Web 校验不严格？错了，不是这样的，Web 校验都是在页面上通过 JavaScript 实现的，只能限制普通用户（这里的普通用户是指不懂 HTML、不懂 HTTP、不懂 Java 的简单使用者），而对于高手，这些校验基本上就是摆设，HTTP 是明文传输的，将其拦截几次，分析一下数据结构，然后再写一个模拟器，一切前端校验就都成了浮云！想往后台提交个什么数据那还不是信手拈来？！

建议 25：不要让四舍五入亏了一方

本建议还是来重温一个小学数学问题：四舍五入。四舍五入是一种近似精确的计算方法，在 Java 5 之前，我们一般是通过使用 `Math.round` 来获得指定精度的整数或小数的，这种方法使用非常广泛，代码如下：

```
public class Client {
    public static void main(String[] args) {
        System.out.println("10.5 近似值: " + Math.round(10.5));
        System.out.println("-10.5 近似值: " + Math.round(-10.5));
    }
}
```

输出结果为：

```
10.5 近似值: 11
-10.5 近似值: -10
```

这是四舍五入的经典案例，也是初级面试官很乐意选择的考题，绝对值相同的两个数字，近似值为什么就不同了呢？这是由 `Math.round` 采用的舍入规则所决定的（采用的是正无穷方向舍入规则，后面会讲解）。我们知道四舍五入是有误差的：其误差值是舍入位的一半。我们以舍入运用最频繁的银行利息计算为例来阐述该问题。

我们知道银行的盈利渠道主要是利息差，从储户手里收拢资金，然后放贷出去，其间的利息差额便是所获得的利润。对一个银行来说，对付给储户的利息的计算非常频繁，人民银行规定每个季度末月的 20 日为银行结息日，一年有 4 次的结息日。

场景介绍完毕，我们回过头来看四舍五入，小于 5 的数字被舍去，大于等于 5 的数字进位后舍去，由于所有位上的数字都是自然计算出来的，按照概率计算可知，被舍入的数字均匀分布在 0 到 9 之间，下面以 10 笔存款利息计算作为模型，以银行家的身份来思考这个算法：

□ 四舍。舍弃的数值：0.000、0.001、0.002、0.003、0.004，因为是舍弃的，对银行家来说，就不用付款给储户了，那每舍弃一个数字就会赚取相应的金额：0.000、0.001、0.002、0.003、0.004。

□ 五入。进位的数值：0.005、0.006、0.007、0.008、0.009，因为是进位，对银行家来说，每进一位就会多付款给储户，也就是亏损了，那亏损部分就是其对应的 10 进制补数：0.005、0.004、0.003、0.002、0.001。

因为舍弃和进位的数字是在 0 到 9 之间均匀分布的，所以对于银行家来说，每 10 笔存款的利息因采用四舍五入而获得的盈利是：

$0.000 + 0.001 + 0.002 + 0.003 + 0.004 - 0.005 - 0.004 - 0.003 - 0.002 - 0.001 = -0.005$

也就是说，每 10 笔的利息计算中就损失 0.005 元，即每笔利息计算损失 0.0005 元，这

对一家有 5 千万储户的银行来说（对国内的银行来说，5 千万是个很小的数字），每年仅仅因为四舍五入的误差而损失的金额是：

```
public class Client {
    public static void main(String[] args) {
        // 银行账户数量，5 千万
        int accountNum = 5000 * 10000;
        // 按照人行的规定，每个季度末月的 20 日为银行结息日
        double cost = 0.0005 * accountNum * 4;
        System.out.println("银行每年损失的金额：" + cost);
    }
}
```

输出的结果是：“银行每年损失的金额：100000.0”。即，每年因为一个算法误差就损失了 10 万元，事实上以上的假设条件都是非常保守的，实际情况可能损失得更多。那各位可能要说了，银行还要放贷呀，放出去这笔计算误差不就抵消掉了吗？不会抵消，银行的贷款数量是非常有限的，其数量级根本没有办法和存款相比。

这个算法误差是由美国银行家发现的（那可是私人银行，钱是自己的，白白损失了可不行），并且对此提出了一个修正算法，叫做银行家舍入（Banker's Round）的近似算法，其规则如下：

- 舍去位的数值小于 5 时，直接舍去；
- 舍去位的数值大于等于 6 时，进位后舍去；
- 当舍去位的数值等于 5 时，分两种情况：5 后面还有其他数字（非 0），则进位后舍去；若 5 后面是 0（即 5 是最后一个数字），则根据 5 前一位数的奇偶性来判断是否需要进位，奇数进位，偶数舍去。

以上规则汇总成一句话：四舍六入五考虑，五后非零就进一，五后为零看奇偶，五前为偶应舍去，五前为奇要进一。我们举例说明，取 2 位精度：

```
round(10.5551) = 10.56
round(10.555) = 10.56
round(10.545) = 10.54
```

要在 Java 5 以上的版本中使用银行家的舍入法则非常简单，直接使用 RoundingMode 类提供的 Round 模式即可，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 存款
        BigDecimal d = new BigDecimal(888888);
        // 月利率，乘 3 计算季利率
        BigDecimal r = new BigDecimal(0.001875 * 3);
        // 计算利息
        BigDecimal i = d.multiply(r).setScale(2, RoundingMode.HALF_EVEN);
```

```

        System.out.println("季利息是: "+i);
    }
}

```

在上面的例子中，我们使用了 BigDecimal 类，并且采用 setScale 方法设置了精度，同时传递了一个 RoundingMode.HALF_EVEN 参数表示使用银行家舍入法则进行近似计算，BigDecimal 和 RoundingMode 是一个绝配，想要采用什么舍入模式使用 RoundingMode 设置即可。目前 Java 支持以下七种舍入方式：

□ ROUND_UP：远离零方向舍入。

向远离 0 的方向舍入，也就是说，向绝对值最大的方向舍入，只要舍弃位非 0 即进位。

□ ROUND_DOWN：趋向零方向舍入。

向 0 方向靠拢，也就是说，向绝对值最小的方向输入，注意：所有的位都舍弃，不存在进位情况。

□ ROUND_CEILING：向正无穷方向舍入。

向正最大方向靠拢，如果是正数，舍入行为类似于 ROUND_UP；如果为负数，则舍入行为类似于 ROUND_DOWN。注意：Math.round 方法使用的即为此模式。

□ ROUND_FLOOR：向负无穷方向舍入。

向负无穷方向靠拢，如果是正数，则舍入行为类似于 ROUND_DOWN；如果是负数，则舍入行为类似于 ROUND_UP。

□ HALF_UP：最近数字舍入（5 进）。

这就是我们最最经典的四舍五入模式。

□ HALF_DOWN：最近数字舍入（5 舍）。

在四舍五入中，5 是进位的，而在 HALF_DOWN 中却是舍弃不进位。

□ HALF_EVEN：银行家算法。

在普通的项目中舍入模式不会有太多影响，可以直接使用 Math.round 方法，但在大量与货币数字交互的项目中，一定要选择好近似的计算模式，尽量减少因算法不同而造成的损失。

注意 根据不同的场景，慎重选择不同的舍入模式，以提高项目的精准度，减少算法损失。

建议 26：提防包装类型的 null 值

我们知道 Java 引入包装类型（Wrapper Types）是为了解决基本类型的实例化问题，以便让一个基本类型也能参与到面向对象的编程世界中。而在 Java 5 中泛型更是对基本类型说了“不”，如想把一个整型放到 List 中，就必须使用 Integer 包装类型。我们来看一段代码：

```
// 计算 list 中所有元素之和
public static int f(List<Integer> list) {
    int count = 0;
    for(int i:list) {
        count += i;
    }
    return count;
}
```

接收一个元素是整型的 List 参数，计算所有元素之和，这在统计、报表项目中很常见，我们来看看这段代码有没有问题。遍历一个列表，然后相加，应该没有问题。那我们再来写一个方法调用，代码如下：

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    list.add(null);
    System.out.println(f(list));
}
```

把 1、2 和空值都放到 List 中，然后调用方法计算，现在来思考一下会不会出错。应该不会出错吧，基本类型和包装类型都是可以通过自动装箱（Autoboxing）和自动拆箱（AutoUnboxing）自由转换的，null 应该可以转为 0 吧，真的是这样吗？我们运行一下看看结果：

```
Exception in thread "main" java.lang.NullPointerException
```

运行失败，报空指针异常，我们稍稍思考一下很快就知道原因了：在程序的 for 循环中，隐含了一个拆箱过程，在此过程中包装类型转换为了基本类型。我们知道拆箱过程是通过调用包装对象的 intValue 方法来实现的，由于包装对象是 null 值，访问其 intValue 方法报空指针异常也就在所难免了。问题清楚了，修改也很简单，加入 null 值检查即可，代码如下：

```
public static int f(List<Integer> list) {
    int count = 0;
    for (Integer i : list) {
        count += (i!=null)?i:0;
    }
    return count;
}
```

上面以 Integer 和 int 为例说明了拆箱问题，其他 7 个包装对象的拆箱过程也存在着同样的问题。包装对象和拆箱对象可以自由转换，这不假，但是要剔除 null 值，null 值并不能转化为基本类型。对于此类问题，我们谨记一点：包装类型参与运算时，要做 null 值校验。

建议 27：谨慎包装类型的大小比较

基本类型是可以比较大小的，其所对应的包装类型都实现了 Comparable 接口也说明了此问题，那我们来比较一下两个包装类型的大小，代码如下：

```
public class Client {
    public static void main(String[] args) {
        Integer i = new Integer(100);
        Integer j = new Integer(100);
        compare(i,j);
    }
    // 比较两个包装对象大小
    public static void compare(Integer i , Integer j) {
        System.out.println(i == j);
        System.out.println(i > j);
        System.out.println(i < j);
    }
}
```

代码很简单，产生了两个 Integer 对象，然后比较两者的关系，既然基本类型和包装类型是可以自由转换的，那上面的代码是不是就可打印出两个相等的值呢？让事实说话，运行结果如下：

```
false
false
false
```

竟然是 3 个 false，也就是说两个值之间不等，也没大小关系，这也太奇怪了吧。不奇怪，我们来一一解释。

□ $i == j$

在 Java 中 “==” 是用来判断两个操作数是否有相等关系的，如果是基本类型则判断值是否相等，如果是对象则判断是否是一个对象的两个引用，也就是地址是否相等，这里很明显是两个对象，两个地址，不可能相等。

□ $i > j$ 和 $i < j$

在 Java 中，“>” 和 “<” 用来判断两个数字类型的大小关系，注意只能是数字型的判断，对于 Integer 包装类型，是根据其 intValue() 方法的返回值（也就是其相应的基本类型）进行比较的（其他包装类型是根据相应的 value 值来比较的，如 doubleValue、floatValue 等），那很显然，两者不可能有大小关系的。

问题清楚了，修改总是比较容易的，直接使用 Integer 实例的 compareTo 方法即可。但是这类问题的产生更应该说是习惯问题，只要是两个对象之间的比较就应该采用相应的方法，而不是通过 Java 的默认机制来处理，除非你确定对此非常了解。

建议 28：优先使用整型池

上一建议我们解释了包装对象的比较问题，本建议将继续深入讨论相关问题，首先看如下代码：

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    while(input.hasNextInt()) {
        int ii = input.nextInt();
        System.out.println("\n====+" + ii + " 的相等判断 =====");
        // 两个通过 new 产生的 Integer 对象
        Integer i = new Integer(ii);
        Integer j = new Integer(ii);
        System.out.println("new 产生的对象: " + (i==j));

        // 基本类型转为包装类型后比较
        i=ii;
        j=ii;
        System.out.println("基本类型转换的对象: " + (i==j));

        // 通过静态方法生成一个实例
        i=Integer.valueOf(ii);
        j = Integer.valueOf(ii);
        System.out.println("valueOf 产生的对象: " + (i==j));
    }
}
```

输入多个数字，然后按照 3 种不同的方式产生 Integer 对象，判断其是否相等，注意这里使用了“==”，这说明判断的不是同一个对象。我们输入三个数字 127、128、555，结果如下：

```
=====127 的相等判断 =====
new 产生的对象: false
基本类型转换的对象: true
valueOf 产生的对象: true

=====128 的相等判断 =====
new 产生的对象: false
基本类型转换的对象: false
valueOf 产生的对象: false

=====555 的相等判断 =====
new 产生的对象: false
基本类型转换的对象: false
valueOf 产生的对象: false
```

很不可思议呀，数字 127 的比较结果竟然与其他两个数字不同，它的装箱动作所产生的对象竟然是同一个对象，valueOf 产生的也是同一个对象，但是大于 127 的数字 128 和 555

在比较过程中所产生的却不是同一个对象，这是为什么？我们一个一个来解释。

(1) new 产生的 Integer 对象

`new` 声明的就是要生成一个新的对象，没二话，这是两个对象，地址肯定不等，比较结果为 `false`。

(2) 装箱生成的对象

对于这一点，首先要说明的是装箱动作是通过 `valueOf` 方法实现的，也就是说后两个算法是相同的，那结果肯定也是一样的，现在的问题是：`valueOf` 是如何生成对象的呢？我们来阅读一下 `Integer.valueOf` 的实现代码：

```
public static Integer valueOf(int i) {
    final int offset = 128;
    if (i >= -128 && i <= 127) { // must cache
        return IntegerCache.cache[i + offset];
    }
    return new Integer(i);
}
```

这段代码的意思已经很明了了，如果是 `-128` 到 `127` 之间的 `int` 类型转换为 `Integer` 对象，则直接从 `cache` 数组中获得，那 `cache` 数组里是什么东西，代码如下：

```
static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static {
    for(int i = 0; i < cache.length; i++)
        cache[i] = new Integer(i - 128);
}
```

`cache` 是 `IntegerCache` 内部类的一个静态数组，容纳的是 `-128` 到 `127` 之间的 `Integer` 对象。通过 `valueOf` 产生包装对象时，如果 `int` 参数在 `-128` 和 `127` 之间，则直接从整型池中获得对象，不在该范围的 `int` 类型则通过 `new` 生成包装对象。

明白了这一点，要理解上面的输出结果就迎刃而解了，`127` 的包装对象是直接从整型池中获得的，不管你输入多少次 `127` 这个数字，获得的对象都是同一个，那地址当然都是相等的。而 `128`、`555` 超出了整型池范围，是通过 `new` 产生一个新的对象，地址不同，当然也就不再相等了。

以上的解释也是整型池的原理，整型池的存在不仅仅提高了系统性能，同时也节约了内存空间，这也是我们使用整型池的原因，也就是在声明包装对象的时候使用 `valueOf` 生成，而不是通过构造函数来生成的原因。顺便提醒大家，在判断对象是否相等的时候，最好是用 `equals` 方法，避免用 “`==`” 产生非预期结果。

注意 通过包装类的 `valueOf` 生成包装实例可以显著提高空间和时间性能。

建议 29：优先选择基本类型

包装类型是一个类，它提供了诸如构造方法、类型转换、比较等非常实用的功能，而且在 Java 5 之后又实现了与基本类型之间的自动转换，这使包装类型如虎添翼，更是应用广泛了，在开发中包装类型已经随处可见，但无论是从安全性、性能方面来说，还是从稳定性方面来说，基本类型都是首选方案。我们来看一段代码：

```
public class Client {
    public static void main(String[] args) {
        Client client = new Client();
        int i=140;
        // 分别传递 int 类型和 Integer 类型
        client.f(i);
        client.f(Integer.valueOf(i));
    }
    public void f(long a) {
        System.out.println(" 基本类型的方法被调用 ");
    }
    public void f(Long a) {
        System.out.println(" 包装类型的方法被调用 ");
    }
}
```

在上面的程序中首先声明了一个 int 变量 i，然后加宽转变成 long 型，再调用 f() 方法，分别传递 int 和 long 的基本类型和包装类型，诸位想想该程序是否能够编译？如果能编译输出结果又是什么呢？

首先，这段程序绝对是能够编译的。不过，说不能编译的同学还是很动了一番脑筋的，只是还缺点火候，你可能会猜测以下这些地方不能编译：

- f() 方法重载有问题。定义的两个 f() 方法实现了重载，一个形参是基本类型，一个形参是包装类型，这类重载很正常。虽然基本类型和包装类型有自动装箱、自动拆箱的功能，但并不影响它们的重载，自动拆箱（装箱）只有在赋值时才会发生，和重载没有关系。
- client.f(i) 报错。i 是 int 类型，传递到 fun(long l) 是没有任何问题的，编译器会自动把 i 的类型加宽，并将其转变为 long 型，这是基本类型的转换规则，也没有任何问题。
- client.f(Integer.valueOf(i)) 报错。代码中没有 f(Integer i) 方法，不可能接收一个 Integer 类型的参数，而且 Integer 和 Long 两个包装类型是兄弟关系，不是继承关系，那就是说肯定编译失败了？不，编译是成功的，稍后再解释为什么这里编译成功。

既然编译通过了，我们来看一下输出：

```
基本类型的方法被调用
基本类型的方法被调用
```

client.f(i) 的输出是正常的，我们已经解释过了。那第二个输出就让人很困惑了，为什么

会调用 f(long a) 方法呢？这是因为自动装箱有一个重要的原则：基本类型可以先加宽，再转变成宽类型的包装类型，但不能直接转变成宽类型的包装类型。这句话比较拗口，简单地说就是，int 可以加宽转变成 long，然后再转变成 Long 对象，但不能直接转变成包装类型，注意这里指的都是自动转换，不是通过构造函数生成。为了解释这个原则，我们再来看一个例子：

```
public class Client {
    public static void main(String[] args) {
        int i=100;
        f(i);
    }
    public static void f(Long l){
    }
}
```

这段程序编译是通不过的，因为 i 是一个 int 类型，不能自动转变为 Long 型。但是修改成以下代码就可以编译通过了：

```
public static void main(String[] args) {
    int i=100;
    long l = (long)i;
    f(l);
}
```

这就是 int 先加宽转变为 long 型，然后自动转换成 Long 型。规则说明白了，我们继续来看 f(Integer.valueOf(i)) 是如何调用的，Integer.valueOf(i) 返回的是一个 Integer 对象，这没错，但是 Integer 和 int 是可以互相转换的。没有 f(Integer i) 方法？没关系，编译器会尝试转换成 int 类型的实参调用，OK，这次成功了，与 f(i) 相同了，于是乎被加宽转变成 long 型——结果也很明显了。整个 f(Integer.valueOf(i)) 的执行过程是这样的：

- i 通过 valueOf 方法包装成一个 Integer 对象。
- 由于没有 f(Integer i) 方法，编译器“聪明”地把 Integer 对象转换成 int。
- int 自动拓宽为 long，编译结束。

使用包装类型确实有方便的地方，但是也会引起一些不必要的困惑，比如我们这个例子，如果 f() 的两个重载方法使用的是基本类型，而且实参也是基本类型，就不会产生以上问题，而且程序的可读性更强。自动装箱（拆箱）虽然很方便，但引起的问题也非常严重——我们甚至都不知道执行的是哪个方法。

注意 重申，基本类型优先考虑。

建议 30：不要随便设置随机种子

随机数在太多的地方使用了，比如加密、混淆数据等，我们使用随机数是期望获得一个

唯一的、不可仿造的数字，以避免产生相同的业务数据造成混乱。在 Java 项目中通常是通过 `Math.random` 方法和 `Random` 类来获得随机数的，我们来看一段代码：

```
public class Client {
    public static void main(String[] args) {
        Random r = new Random();
        for(int i=1;i<4;i++) {
            System.out.println("第 "+i+" 次: "+r.nextInt());
        }
    }
}
```

代码很简单，我们一般都是这样获得随机数的，运行此程序可知：三次打印的随机数都不相同，即使多次运行结果也不同，这也正是我们想要随机数的原因。我们再来看下面的程序：

```
public class Client {
    public static void main(String[] args) {
        Random r = new Random(1000);
        for(int i=1;i<4;i++) {
            System.out.println("第 "+i+" 次: "+r.nextInt());
        }
    }
}
```

上面使用了 `Random` 的有参构造，运行结果如下：

```
第 1 次: -498702880
第 2 次: -858606152
第 3 次: 1942818232
```

计算机不同输出的随机数也不同，但是有一点是相同的：在同一台机器上，甭管运行多少次，所打印的随机数都是相同的，也就是说第一次运行，会打印出这三个随机数，第二次运行还是打印出这三个随机数，只要是在同一台硬件机器上，就永远都会打印出相同的随机数，似乎随机数不随机了，问题何在？

那是因为产生随机数的种子被固定了，在 Java 中，随机数的产生取决于种子，随机数和种子之间的关系遵从以下两个规则：

- 种子不同，产生不同的随机数。
- 种子相同，即使实例不同也产生相同的随机数。

看完上面两个规则，我们再来看这个例子，会发现问题就出在有参构造上，`Random` 类的默认种子（无参构造）是 `System.nanoTime()` 的返回值（JDK 1.5 版本以前默认种子是 `System.currentTimeMillis()` 的返回值），注意这个值是距离某一个固定时间点的纳秒数，不同的操作系统和硬件有不同的固定时间点，也就是说不同的操作系统其纳秒值是不同的，而同一个操作系统纳秒值也会不同，随机数自然也就不同了。（顺便说下，`System.nanoTime` 不

能用于计算日期，那是因为“固定”的时间点是不确定的，纳秒值甚至可能是负值，这点与 System.currentTimeMillis 不同。）

new Random(1000) 显式地设置了随机种子为 1000，运行多次，虽然实例不同，但都会获得相同的三个随机数。所以，除非必要，否则不要设置随机种子。

顺便提一下，在 Java 中有两种方法可以获得不同的随机数：通过 java.util.Random 类获得随机数的原理和 Math.random 方法相同，Math.random() 方法也是通过生成一个 Random 类的实例，然后委托 nextDouble() 方法的，两者是殊途同归，没有差别。

注意 若非必要，不要设置随机数种子。



第3章

类、对象及方法

书读得多而不思考，你会觉得自己知道的很多。
书读得多而思考，你会觉得自己不懂的越来越多。

——伏尔泰

在面向对象编程（Object-Oriented Programming, OOP）的世界里，类和对象是真实世界的描述工具，方法是行为和动作的展示形式，封装、继承、多态则是其多姿多彩的主要实现方式，如此，OOP 才会像现在这样繁荣昌盛、欣欣向荣。

本章主要讲述关于 Java 类、对象、方法的种种规则、限制及建议，让读者在面向对象编程的世界中走得更远，飞得更高。

建议 31：在接口中不要存在实现代码

看到这样的标题读者可能会纳闷：接口中有实现代码？这怎么可能呢？确实，接口中可以声明常量，声明抽象方法，也可以继承父接口，但就是不能有具体实现，因为接口是一种契约（Contract），是一种框架性协议，这表明它的实现类都是同一种类型，或者是具备相似特征的一个集合体。对于一般程序，接口确实没有任何实现，但是在那些特殊的程序中就例外了，阅读如下代码：

```
public class Client {
    public static void main(String[] args) {
        // 调用接口的实现
        B.s.doSomething();
    }
}
// 在接口中存在实现代码
interface B{
    public static final S s = new S(){
        public void doSomething(){
            System.out.println("我在接口中实现了");
        }
    };
}
// 被实现的接口
interface S{
    public void doSomething();
}
```

仔细看 main 方法，注意那个 B 接口。它调用了接口常量，在没有任何显式实现类的情况下，它竟然打印出了结果，那 B 接口中的 s 常量（接口是 S）是在什么地方被实现的呢？答案是在 B 接口中。

在 B 接口中声明了一个静态常量 s，其值是一个匿名内部类（Anonymous Inner Class）的实例对象，就是该匿名内部类（当然，可以不用匿名，直接在接口中实现内部类也是允许的）实现了 S 接口。你看，在接口中存在着实现代码吧！

这确实很好，很强大，但是在一般的项目中，此类代码是严禁出现的，原因很简单：这是一种不好的编码习惯，接口是用来干什么的？接口是一个契约，不仅仅约束着实现者，同

时也是一个保证，保证提供的服务（常量、方法）是稳定、可靠的，如果把实现代码写到接口中，那接口就绑定了可能变化的因素，这就会导致实现不再稳定和可靠，是随时都可能被抛弃、被更改、被重构的。所以，接口中虽然可以有实现，但应避免使用。

注意 接口中不能存在实现代码。

建议 32：静态变量一定要先声明后赋值

这标题看着让人很纳闷，什么叫做变量一定要先声明后赋值？Java 中的变量不都是先声明后使用的吗？难道还能先使用后声明？能不能暂且不说，我们先来看一个例子，代码如下：

```
public class Client {
    public static int i=1;
    static{
        i=100;
    }
    public static void main(String[] args) {
        System.out.println(i);
    }
}
```

这段程序很简单，输出 100 嘛！对，确实是 100，我们再稍稍修改一下，代码如下：

```
public class Client {
    static{
        i=100;
    }
    public static int i=1;
    public static void main(String[] args) {
        System.out.println(i);
    }
}
```

注意，变量 i 的声明和赋值调换了位置，现在的问题是：这段程序能否编译？如果可以编译那输出是多少？还要注意：这个变量 i 可是先使用（也就是赋值）后声明的。

答案是：可以编译，没有任何问题，输出是 1。对，你没有看错，输出确实是 1，而不是 100。仅仅调换了一下位置，输出就变了，而且变量 i 还真是先使用后声明的，难道这世界真的颠倒了？

这要从静态变量的诞生说起了，静态变量是类加载时被分配到数据区（Data Area）的，它在内存中只有一个拷贝，不会被分配多次，其后的所有赋值操作都是值改变，地址则保持不变。我们知道 JVM 初始化变量是先声明空间，然后再赋值的，也就是说：

```
int i=100;
```

在 JVM 中是分开执行，等价于：

```
int i; // 分配地址空间
i=100; // 赋值
```

静态变量是在类初始化时首先被加载的，JVM 会去查找类中所有的静态声明，然后分配空间，注意这时候只是完成了地址空间的分配，还没有赋值，之后 JVM 会根据类中静态赋值（包括静态类赋值和静态块赋值）的先后顺序来执行。对于程序来说，就是先声明了 int 类型的地址空间，并把地址传递给了 i，然后按照类中的先后顺序执行赋值动作，首先执行静态块中 i=100，接着执行 i=1，那最后的结果就是 i=1 了。

哦，如此而已，那再问一个问题：如果有多个静态块对 i 继续赋值呢？i 当然还是等于 1 了，谁的位置最靠后谁有最终的决定权。

有些程序员喜欢把变量定义放到类的底部，如果这是实例变量还好说，没有任何问题，但如果是静态变量，而且还在静态块中进行了赋值，那这结果可就和你期望的不一样了，所以遵循 Java 通用的开发规范“变量先声明后使用”是一个良好的编码风格。

注意 再次重申变量要先声明后使用，这不是一句废话。

建议 33：不要覆写静态方法

我们知道在 Java 中可以通过覆写（Override）来增强或减弱父类的方法和行为，但覆写是针对非静态方法（也叫做实例方法，只有生成实例才能调用的方法）的，不能针对静态方法（static 修饰的方法，也叫做类方法），为什么呢？我们先看一个例子，代码如下：

```
public class Client {
    public static void main(String[] args) {
        Base base = new Sub();
        // 调用非静态方法
        base.doAnything();
        // 调用静态方法
        base.doSomething();
    }
}

class Base{
    // 父类静态方法
    public static void doSomething(){
        System.out.println("我是父类静态方法");
    }
    // 父类非静态方法
    public void doAnything(){
        System.out.println("我是父类非静态方法");
    }
}
```

```

    }
}

class Sub extends Base{
    // 子类同名、同参数的静态方法
    public static void doSomething(){
        System.out.println("我是子类静态方法");
    }
    // 覆写父类的非静态方法
    @Override
    public void doAnything(){
        System.out.println("我是子类非静态方法");
    }
}

```

注意看程序，子类的 doAnything 方法覆写了父类方法，这没有任何问题，那 doSomething 方法呢？它与父类的方法名相同，输入、输出也相同，按道理来说应该是覆写，不过到底是覆写呢？我们先看输出结果：

```

我是子类非静态方法
我是父类静态方法

```

这个结果很让人困惑，同样是调用子类方法，一个执行了子类方法，一个执行了父类方法，两者的差别仅仅是有没有 static 修饰，却得到不同的输出结果，原因何在呢？

我们知道一个实例对象有两个类型：表面类型（Apparent Type）和实际类型（Actual Type），表面类型是声明时的类型，实际类型是对象产生时的类型，比如我们例子，变量 base 的表面类型是 Base，实际类型是 Sub。对于非静态方法，它是根据对象的实际类型来执行的，也就是执行了 Sub 类中的 doAnything 方法。而对于静态方法来说就比较特殊了，首先静态方法不依赖实例对象，它是通过类名访问的；其次，可以通过对象访问静态方法，如果是通过对象调用静态方法，JVM 则会通过对对象的表面类型查找到静态方法的入口，继而执行之。因此上面的程序打印出“我是父类静态方法”，也就不足为奇了。

在子类中构建与父类相同的方法名、输入参数、输出参数、访问权限（权限可以扩大），并且父类、子类都是静态方法，此种行为叫做隐藏（Hide），它与覆写有两点不同：

- 表现形式不同。隐藏用于静态方法，覆写用于非静态方法。在代码上的表现是：@Override 注解可以用于覆写，不能用于隐藏。
- 职责不同。隐藏的目的是为了抛弃父类静态方法，重现子类方法，例如我们的例子，Sub.doSomething 的出现是为了遮盖父类的 Base.doSomething 方法，也就是期望父类的静态方法不要破坏子类的业务行为；而覆写则是将父类的行为增强或减弱，延续父类的职责。

解释了这么多，我们回头看一下本建议的标题：静态方法不能覆写，可以再续上一句话，虽然不能覆写，但是可以隐藏。顺便说一下，通过实例对象访问静态方法或静态属性不是好

习惯，它给代码带来了“坏味道”，建议读者阅之戒之。

建议 34：构造函数尽量简化

我们知道在通过 new 关键字生成对象时必然会调用构造函数，构造函数的简繁情况会直接影响实例对象的创建是否繁琐。在项目开发中，我们一般都会制订构造函数尽量简单，尽可能不抛异常，尽量不做复杂算法等规范，那如果一个构造函数确实复杂了会怎么样？我们来看一段代码：

```
public class Client {
    public static void main(String[] args) {
        Server s = new SimpleServer(1000);
    }
}
// 定义一个服务
abstract class Server{
    public final static int DEFAULT_PORT = 40000;
    public Server(){
        // 获得子类提供的端口号
        int port = getPort();
        System.out.println("端口号: " + port);
        /* 进行监听动作 */
    }
    // 由子类提供端口号，并做可用性检查
    protected abstract int getPort();
}

class SimpleServer extends Server{
    private int port=100;
    // 初始化传递一个端口号
    public SimpleServer(int _port){
        port = _port;
    }
    // 检查端口号是否有效，无效则使用默认端口，这里使用随机数模拟
    @Override
    protected int getPort() {
        return Math.random() > 0.5?port:DEFAULT_PORT;
    }
}
```

该代码是一个服务类的简单模拟程序，Server 类实现了服务器的创建逻辑，子类只要在生成实例对象时传递一个端口号即可创建一个监听该端口的服务，该代码的意图如下：

- 通过 SimpleServer 的构造函数接收端口参数。
- 子类的构造函数默认调用父类的构造函数。
- 父类构造函数调用子类的 getPort 方法获得端口号。

- 父类构造函数建立端口监听机制。
- 对象创建完毕，服务监听启动，正常运行。

貌似很合理，再仔细看看代码，确实也和我们的意图相吻合，那我们尝试多次运行看看，输出结果要么是“端口号：40000”，要么是“端口号：0”，永远不会出现“端口号：100”或是“端口号：1000”，这就奇怪了，40000 还好说，但那个 0 是怎么冒出来的呢？代码在什么地方出现问题了？

要解释这个问题，我们首先要说说子类是如何实例化的。子类实例化时，会首先初始化父类（注意这里是初始化，可不是生成父类对象），也就是初始化父类的变量，调用父类的构造函数，然后才会初始化子类的变量，调用子类自己的构造函数，最后生成一个实例对象。了解了相关知识，我们再来看上面的程序，其执行过程如下：

- 子类 SimpleServer 的构造函数接收 int 类型的参数：1000。
- 父类初始化常变量，也就是 DEFAULT_PORT 初始化，并设置为 40000。
- 执行父类无参构造函数，也就是子类的有参构造中默认包含了 super() 方法。
- 父类无参构造函数执行到“int port = getPort()”方法，调用子类的 getPort 方法实现。
- 子类的 getPort 方法返回 port 值（注意，此时 port 变量还没有赋值，是 0）或 DEFAULT_PORT（此时已经是 40000）了。
- 父类初始化完毕，开始初始化子类的实例变量，port 赋值 100。
- 执行子类构造函数，port 被重新赋值为 1000。
- 子类 SimpleServer 实例化结束，对象创建完毕。

终于清楚了，在类初始化时 getPort 方法返回的 port 值还没有赋值，port 只是获得了默认初始值（int 类的实例变量默认初始值是 0），因此 Server 永远监听的是 40000 端口了（0 端口是没有意义的）。这个问题的产生从浅处说是由于类元素初始化顺序导致的，从深处说是因为构造函数太复杂而引起的。构造函数用作初始化变量，声明实例的上下文，这都是简单的实现，没有任何问题，但我们的例子却实现了一个复杂的逻辑，而这放在构造函数里就不合适了。

问题知道了，修改也很简单，把父类的无参构造函数中的所有实现都移动到一个叫做 start 的方法中，将 SimpleServer 类初始化完毕，再调用其 start 方法即可实现服务器的启动工作，简洁而又直观，这也是大部分 JEE 服务器的实现方式。

注意 构造函数简化，再简化，应该达到“一眼洞穿”的境界。

建议 35：避免在构造函数中初始化其他类

构造函数是一个类初始化必须执行的代码，它决定着类的初始化效率，如果构造函数比

较复杂，而且还关联了其他类，则可能产生意想不到的问题，我们来看如下代码：

```

public class Client {
    public static void main(String[] args) {
        Son s = new Son();
        s.doSomething();
    }
}
// 父类
class Father{
    Father(){
        new Other();
    }
}// 子类
class Son extends Father{
    public void doSomething(){
        System.out.println("Hi,show me something");
    }
}
// 相关类
class Other{
    public Other(){
        new Son();
    }
}
}

```

这段代码并不复杂，只是在构造函数中初始化了其他类，想想看这段代码的运行结果是什么？是打印“Hi,show me something”吗？

答案是这段代码不能运行，报 StackOverflowError 异常，栈（Stack）内存溢出。这是因为声明 s 变量时，调用了 Son 的无参构造函数，JVM 又默认调用了父类 Father 的无参构造函数，接着 Father 类又初始化了 Other 类，而 Other 类又调用了 Son 类，于是一个死循环就诞生了，直到栈内存被消耗完毕为止。

可能有读者会觉得这样的场景不可能在开发中出现，那我们来思考这样的场景：Father 是由框架提供的，Son 类是我们自己编写的扩展代码，而 Other 类则是框架要求的拦截类（Interceptor 类或者 Handle 类或者 Hook 方法），再来看看该问题，这种场景不可能出现吗？

那有读者可能要说了，这种问题只要系统一运行就会发现，不可能对项目产生影响。

那是因为我们在这里展示的代码比较简单，很容易一眼洞穿，一个项目的构造函数可不止一两个，类之间的关系也不会这么简单的，要想瞥一眼就能明白是否有缺陷这对所有人员来说都是不可能完成的任务，解决此类问题的最好办法就是：不要在构造函数中声明初始化其他类，养成良好的习惯。

建议 36：使用构造代码块精炼程序

什么叫代码块（Code Block）？用大括号把多行代码封装在一起，形成一个独立的数据体，实现特定算法的代码集合即为代码块，一般来说代码块是不能单独运行的，必须要有运行主体。在 Java 中一共有四种类型的代码块：

(1) 普通代码块

就是在方法后面使用“{}”括起来的代码片段，它不能单独执行，必须通过方法名调用执行。

(2) 静态代码块

在类中使用 static 修饰，并使用“{}”括起来的代码片段，用于静态变量的初始化或对象创建前的环境初始化。

(3) 同步代码块

使用 synchronized 关键字修饰，并使用“{}”括起来的代码片段，它表示同一时间只能有一个线程进入到该方法块中，是一种多线程保护机制。

(4) 构造代码块

在类中没有任何的前缀或后缀，并使用“{}”括起来的代码片段。

我们知道，一个类至少有一个构造函数（如果没有，编译器会无私地为其创建一个无参构造函数），构造函数是在对象生成时调用的，那现在的问题来了：构造函数和构造代码块是什么关系？构造代码块是在什么时候执行的？在回答这个问题之前，我们先来看看编译器是如何处理构造代码块的，看如下代码：

```
public class Client {
{
    // 构造代码块
    System.out.println(" 执行构造代码块 ");
}

public Client(){
    System.out.println(" 执行无参构造 ");
}

public Client(String _str){
    System.out.println(" 执行有参构造 ");
}
}
```

这是一段非常简单的代码，它包含了构造代码块、无参构造、有参构造，我们知道代码块不具有独立执行的能力，那么编译器是如何处理构造代码块呢？很简单，编译器会把构造代码块插入到每个构造函数的最前端，上面的代码与如下代码等价：

```

public class Client {
    public Client(){
        System.out.println(" 执行构造代码块 ");
        System.out.println(" 执行无参构造 ");
    }

    public Client(String _str){
        System.out.println(" 执行构造代码块 ");
        System.out.println(" 执行有参构造 ");
    }
}

```

每个构造函数的最前端都被插入了构造代码块，很显然，在通过 new 关键字生成一个实例时会先执行构造代码块，然后再执行其他代码，也就是说：构造代码块会在每个构造函数内首先执行（需要注意的是：构造代码块不是在构造函数之前运行的，它依托于构造函数的执行），明白了这一点，我们就可以把构造代码块应用到如下场景中：

(1) 初始化实例变量 (Instance Variable)

如果每个构造函数都要初始化变量，可以通过构造代码块来实现。当然也可以通过定义一个方法，然后在每个构造函数中调用该方法来实现，没错，可以解决，但是要在每个构造函数中都调用该方法，而这就是其缺点，若采用构造代码块的方式则不用定义和调用，会直接由编译器写入到每个构造函数中，这才是解决此类问题的绝佳方式。

(2) 初始化实例环境

一个对象必须在适当的场景下才能存在，如果没有适当的场景，则就需要在创建对象时创建此场景，例如在 JEE 开发中，要产生 HTTP Request 必须首先建立 HTTP Session，在创建 HTTP Request 时就可以通过构造代码块来检查 HTTP Session 是否已经存在，不存在则创建之。

以上两个场景利用了构造代码块的两个特性：在每个构造函数中都运行和在构造函数中它会首先运行。很好地利用构造代码块的这两个特性不仅可以减少代码量，还可以让程序更容易阅读，特别是当所有的构造函数都要实现逻辑，而且这部分逻辑又很复杂时，这时就可以通过编写多个构造代码块来实现。每个代码块完成不同的业务逻辑（当然了，构造函数尽量简单，这是基本原则），按照业务顺序依次存放，这样在创建实例对象时 JVM 也就会按照顺序依次执行，实现复杂对象的模块化创建。

建议 37：构造代码块会想你所想

上一个建议中我们提议使用构造代码块来简化代码，并且也了解到编译器会自动把构造代码块插入到各个构造函数中，那我们接下来看看编译器是不是足够聪明，能够为我们解决真实的开发问题。有这样一个案例：统计一个类的实例数量。可能你要说了，这很简单，在

每个构造函数中加入一个对象计数器不就解决问题了吗？或者使用我们上一个建议介绍的，使用构造代码块也可以。确实如此，我们来看如下代码是否可行：

```
public class Client {
    public static void main(String[] args) {
        new Base();
        new Base("");
        new Base(0);
        System.out.println("实例对象数量: " + Base.getNumOfObjects());
    }
}

class Base{
    // 对象计数器
    private static int numOfObjects = 0;

    {
        // 构造代码块，计算产生对象数量
        numOfObjects++;
    }

    public Base(){
    }

    // 有参构造调用无参构造
    public Base(String _str){
        this();
    }

    // 有参构造不调用其他构造
    public Base(int _i){
    }

    // 返回在一个 JVM 中，创建了多少个实例对象
    public static int getNumOfObjects(){
        return numOfObjects;
    }
}
```

这段代码是可行的吗？能计算出实例对象的数量吗？哎，好像不对呀，如果编译器把构造代码块插入到各个构造函数中，那带有 String 形参的构造函数可就有问题，它会调用无参构造，那通过它生成 Base 对象时就会执行两次构造代码块：一次是由无参构造函数调用构造代码块，一次是执行自身的构造代码块，这样的话计算可就不准确了，main 函数实际在内存中产生了 3 个对象，但结果却会是 4。不过真是这样的吗？Are you sure？我们运行一下看看结果：

实例对象数量： 3

非常遗憾，你错了，实例对象的数量还是 3，程序没有任何问题。奇怪吗？不奇怪，上

一个建议是说编译器会把构造代码块插入到每一个构造函数中，但是有一个例外的情况没有说明：如果遇到 `this` 关键字（也就是构造函数调用自身其他的构造函数时）则不插入构造代码块，对于我们的例子来说，编译器在编译时发现 `String` 形参的构造函数调用了无参构造，于是放弃插入构造代码块，所以只执行了一次构造代码块——结果就是如此。

那 Java 编译器为什么会这么聪明呢？这还要从构造代码块的诞生说起，构造代码块是为了提取构造函数的共同量，减少各个构造函数的代码而产生的，因此，Java 就很聪明地认为把代码块插入到没有 `this` 方法的构造函数中即可，而调用其他构造函数的则不插入，确保每个构造函数只执行一次构造代码块。

还有一点需要说明，读者千万不要以为 `this` 是特殊情况，那 `super` 也会类似处理了。其实不会，在构造代码块的处理上，`super` 方法没有任何特殊的地方，编译器只是把构造代码块插入到 `super` 方法之后执行而已，仅此不同。

注意 放心地使用构造代码块吧，Java 已经想你所想了。

建议 38：使用静态内部类提高封装性

Java 中的嵌套类（Nested Class）分为两种：静态内部类（也叫静态嵌套类，Static Nested Class）和内部类（Inner Class）。内部类我们介绍过很多了，现在来看看静态内部类。什么是静态内部类呢？是内部类，并且是静态（`static` 修饰）的即为静态内部类。只有在是静态内部类的情况下才能把 `static` 修复符放在类前，其他任何时候 `static` 都是不能修饰类的。

静态内部类的形式很好理解，但是为什么需要静态内部类呢？那是因为静态内部类有两个优点：加强了类的封装性和提高了代码的可读性，我们通过一段代码来解释这两个优点，如下所示：

```
public class Person{
    //姓名
    private String name;
    //家庭
    private Home home;
    //构造函数设置属性值
    public Person(String _name){
        name = _name;
    }
    /* home、name 的 getter/setter 方法省略 */
}

public static class Home{
    //家庭地址
    private String address;
    //家庭电话
    private String tel;
```

```

        public Home(String _address, String _tel) {
            address = _address;
            tel = _tel;
        }
        /* address、tel 的 getter/setter 方法省略 */
    }
}

```

其中，Person 类中定义了一个静态内部类 Home，它表示的意思是“人的家庭信息”，由于 Home 类封装了家庭信息，不用在 Person 类中再定义 homeAddre、homeTel 等属性，这就使封装性提高了。同时我们仅仅通过代码就可以分析出 Person 和 Home 之间的强关联关系，也就是说语义增强了，可读性提高了。所以在使用时就会非常清楚它要表达的含义：

```

public static void main(String[] args) {
    // 定义张三这个人
    Person p = new Person("张三");
    // 设置张三的家庭信息
    p.setHome(new Person.Home("上海", "021"));
}

```

定义张三这个人，然后通过 Person.Home 类设置张三的家庭信息，这是不是就和我们真实世界的情形相同了？先登记人的主要信息，然后登记人员的分类信息。可能你又要问了，这和我们一般定义的类有什么区别呢？又有什么吸引人的地方呢？如下所示：

- 提高封装性。从代码位置上来讲，静态内部类放置在外部类内，其代码层意义就是：静态内部类是外部类的子行为或子属性，两者直接保持着一定的关系，比如在我们的例子中，看到 Home 类就知道它是 Person 的 Home 信息。
- 提高代码的可读性。相关联的代码放在一起，可读性当然提高了。
- 形似内部，神似外部。静态内部类虽然存在于外部类内，而且编译后的类文件名也包含外部类（格式是：外部类 +\$+ 内部类），但是它可以脱离外部类存在，也就是说我们仍然可以通过 new Home() 声明一个 Home 对象，只是需要导入 “Person.Home” 而已。

解释了这么多，读者可能会觉得外部类和静态内部类之间是组合关系（Composition）了，这是错误的，外部类和静态内部类之间有强关联关系，这仅仅表现在“字面”上，而深层次的抽象意义则依赖于类的设计。

那静态内部类与普通内部类有什么区别呢？问得好，区别如下：

(1) 静态内部类不持有外部类的引用

在普通内部类中，我们可以直接访问外部类的属性、方法，即使是 private 类型也可以访问，这是因为内部类持有一个外部类的引用，可以自由访问。而静态内部类，则只可以访问外部类的静态方法和静态属性（如果是 private 权限也能访问，这是由其代码位置所决定的），其他则不能访问。

(2) 静态内部类不依赖外部类

普通内部类与外部类之间是相互依赖的关系，内部类实例不能脱离外部类实例，也就是说它们会同生同死，一起声明，一起被垃圾回收器回收。而静态内部类是可以独立存在的，即使外部类消亡了，静态内部类还是可以存在的。

(3) 普通内部类不能声明 static 的方法和变量

普通内部类不能声明 static 的方法和变量，注意这里说的是变量，常量（也就是 final static 修饰的属性）还是可以的，而静态内部类形似外部类，没有任何限制。

建议 39：使用匿名类的构造函数

阅读如下代码，看看是否可以编译：

```
public static void main(String[] args) {
    List l1 = new ArrayList();
    List l2 = new ArrayList(){};
    List l3 = new ArrayList(){ {} };
    System.out.println(l1.getClass() == l2.getClass());
    System.out.println(l2.getClass() == l3.getClass());
    System.out.println(l1.getClass() == l3.getClass());
}
```

注意 ArrayList 后面的不同点：l1 变量后面什么都没有，l2 后面有一对 {}, l3 后面有 2 对嵌套的 {}，这段程序能不能编译呢？若能编译，那输出是多少呢？

答案是能编译，输出的是 3 个 false。l1 很容易解释，就是声明了 ArrayList 的实例对象，那 l2 和 l3 代表的是什么呢？

(1) l2=new ArrayList(){};

l2 代表的是一个匿名类的声明和赋值，它定义了一个继承于 ArrayList 的匿名类，只是没有任何的覆盖方法而已，其代码类似于：

```
// 定义一个继承 ArrayList 的内部类
class Sub extends ArrayList{
}
// 声明和赋值
List l2 = new Sub();
```

(2) l3=new ArrayList(){{}};

这个语句就有点怪了，还带了两对大括号，我们分开来解释就会明白了，这也是一个匿名类的定义，它的代码类似于：

```
// 定义一个继承 ArrayList 的内部类
class Sub extends ArrayList{
{}}
```

```

        // 初始化块
    }
}

// 声明和赋值
List l3 = new Sub();

```

看到了吧，就是多了一个初始化块而已，起到构造函数的功能。我们知道一个类肯定有一个构造函数，且构造函数的名称和类名相同，那问题来了：匿名类的构造函数是什么呢？它没有名字呀！很显然，初始化块就是它的构造函数。当然，一个类中的构造函数块可以是多个，也就是说可以出现如下代码：

```
List l3 = new ArrayList(){(){};(){};(){};};
```

上面的代码是正确无误，没有任何问题的。现在清楚了：匿名函数虽然没有名字，但也是可以有构造函数的，它用构造函数块来代替，那上面的 3 个输出就很清楚了：虽然父类相同，但是类还是不同的。

建议 40：匿名类的构造函数很特殊

在上一个建议中我们讲到匿名类虽然没有名字，但可以有一个初始化块来充当构造函数，那这个构造函数是否就和普通的构造函数完全一样呢？我们来看一个例子，设计一个计算器，进行加减乘除运算，代码如下：

```

// 定义一个枚举，限定操作符
enum Ops {ADD, SUB}
class Calculator {
    private int i, j, result;
    // 无参构造
    public Calculator() {}
    // 有参构造
    public Calculator(int _i, int _j) {
        i = _i;
        j = _j;
    }
    // 设置符号，是加法运算还是减法运算
    protected void setOperator(Ops _op) {
        result = _op.equals(Ops.ADD)?i+j:i-j;
    }
    // 取得运算结果
    public int getResult(){
        return result;
    }
}

```

代码的意图是，通过构造函数输入两个 int 类型的数字，然后根据设置的操作符（加法

还是减法) 进行计算, 编写一个客户端调用:

```
public static void main(String[] args) {
    Calculator c1 = new Calculator(1,2) {
        {
            •
            setOperator(Ops.ADD);
        }
    };
    System.out.println(c1.getResult());
}
```

这段匿名类的代码非常清晰: 接收两个参数 1 和 2, 然后设置一个操作符号, 计算其值, 结果是 3, 这毫无疑问, 但是这中间隐藏着一个问题: 带有参数的匿名类声明时到底是调用的哪一个构造函数呢? 我们把这段程序模拟一下:

```
// 加法计算
class Add extends Calculator {
{
    setOperator(Ops.ADD);
}
// 覆写父类的构造方法
public Add(int _i, int _j) {
}
}
```

匿名类和这个 Add 类是等价的吗? 可能有人会说: 上面只是把匿名类增加了一个名字, 其他的都没有改动, 那肯定是等价的啦! 毫无疑问! 那好, 你再写个客户端调用 Add 类的方法看看。是不是输出结果为 0 (为什么是 0? 这很容易, 有参构造没有赋值)。这说明两者不等价, 不过, 原因何在呢?

原来是因为匿名类的构造函数特殊处理机制, 一般类 (也就是具有显式名字的类) 的所有构造函数默认都是调用父类的无参构造的, 而匿名类因为没有名字, 只能由构造代码块代替, 也就无所谓的有参和无参构造函数了, 它在初始化时直接调用了父类的同参数构造, 然后再调用自己的构造代码块, 也就是说上面的匿名类与下面的代码是等价的:

```
// 加法计算
class Add extends Calculator {
{
    setOperator(Ops.ADD);
}
// 覆写父类的构造方法
public Add(int _i, int _j) {
    super(_i,_j);
}
}
```

它首先会调用父类有两个参数的构造函数, 而不是无参构造, 这是匿名类的构造函数与

普通类的差别，但是这一点也确实鲜有人细细琢磨，因为它的处理机制符合习惯呀，我传递两个参数，就是希望先调用父类有两个参数的构造，然后再执行我自己的构造函数，而 Java 的处理机制也正是如此处理的！

建议 41：让多重继承成为现实

在 Java 中一个类可以多重实现，但不能多重继承，也就是说一个类能够同时实现多个接口，但不能同时继承多个类。但有时候我们确实需要继承多个类，比如希望拥有两个类的行为功能，就很难使用单继承来解决问题了（当然，使用多层继承是可以解决的）。幸运的是 Java 中提供的内部类可以曲折地解决此问题，我们来看一个案例，定义一个父亲、母亲接口，描述父亲强壮、母亲温柔的理想情形，代码如下：

```
// 父亲
interface Father{
    public int strong();
}
// 母亲
interface Mother{
    public int kind();
}
```

其中 strong 和 kind 的返回值表示强壮和温柔的指数，指数越高强壮度和温柔度也就越高，这与在游戏中设置人物的属性值是一样的。我们继续来看父亲、母亲这两个实现：

```
class FatherImpl implements Father{
    // 父亲的强壮指数是 8
    public int strong(){
        return 8;
    }
}

class MotherImpl implements Mother{
    // 母亲的温柔指数是 8
    public int kind(){
        return 8;
    }
}
```

父亲强壮指数是 8，母亲温柔指数也是 8，门当户对，那他们生的儿子、女儿一定更优秀了，我们先来看儿子类，代码如下：

```
class Son extends FatherImpl implements Mother{
    @Override
    public int strong(){
```

```

    // 儿子比父亲强壮
    return super.strong() + 1;
}

@Override
public int kind(){
    return new MotherSpecial().kind();
}

private class MotherSpecial extends MotherImpl{
    public int kind(){
        // 儿子温柔指数降低了
        return super.kind() - 1;
    }
}
}

```

儿子继承自父亲，变得比父亲更强壮了（覆写父类 strong 方法），同时儿子也具有母亲的优点，只是温柔指数降低了。注意看，这里构造了 MotherSpecial 类继承母亲类，也就是获得了母亲类的行为方法，这也是内部类的一个重要特性：内部类可以继承一个与外部类无关的类，保证了内部类的独立性，正是基于这一点，多重继承才会成为可能。MotherSpecial 的这种内部类叫做成员内部类（也叫做实例内部类，Instance Inner Class）。我们再来看看女儿类，代码如下：

```

class Daughter extends MotherImpl implements Father{

    @Override
    public int strong() {
        return new FatherImpl(){
            @Override
            public int strong() {
                // 女儿的强壮指数降低了
                return super.strong() - 2 ;
            }
        }.strong();
    }
}

```

女儿继承了母亲的温柔指数，同时又覆写父类的强壮指数，不多解释。注意看覆写的 strong 方法，这里是创建了一个匿名内部类（Anonymous Inner Class）来覆写父类的方法，以完成继承父亲行为的功能。

多重继承指的是一个类可以同时从多于一个的父类那里继承行为与特征，按照这个定义来看，我们的儿子类、女儿类都实现了从父亲类、母亲类那里所继承的功能，应该属于多重继承。这要完全归功于内部类，诸位在需要用到多重继承时，可以思考一下内部类。

在现实生活中，也确实存在多重继承的问题，上面的例子是说后人即继承了父亲也继承

了母亲的行为和特征，再比如我国的特产动物“四不像”（学名麋鹿），其外形“似鹿非鹿，似马非马，似牛非牛，似驴非驴”，这你要是想用单继承表示就麻烦了，如果用多继承则可以很好地解决问题：定义鹿、马、牛、驴四个类，然后建立麋鹿类的多个内部类，继承它们即可。

建议 42：让工具类不可实例化

Java 项目中使用的工具类非常多，比如 JDK 自己的工具类 `java.lang.Math`、`java.util.Collections` 等都是我们经常用到的。工具类的方法和属性都是静态的，不需要生成实例即可访问，而且 JDK 也做了很好的处理，由于不希望被初始化，于是就设置构造函数为 `private` 访问权限，表示除了类本身外，谁都不能产生一个实例，我们来看一下 `java.lang.Math` 代码：

```
public final class Math {
    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}
}
```

之所以要将 “`Don't let anyone instantiate this class.`” 留下来，是因为 `Math` 的构造函数设置为 `private` 了：我就是一个工具类，我只想要其他类通过类名来访问，我不想你通过实例对象访问。这在平台型或框架型项目中已经足够了。但是如果已经告诉你不能这么做了，你还要生成一个 `Math` 实例来访问静态方法和属性（Java 的反射是如此的发达，修改个构造函数的访问权限易如反掌），那我就不保证正确性了，隐藏问题随时都有可能爆发！那我们在项目开发中有没有更好的限制办法呢？有，即不仅仅设置成 `private` 访问权限，还抛异常，代码如下：

```
public class UtilsClass {
    private UtilsClass(){
        throw new Error("不要实例化我！");
    }
}
```

如此做才能保证一个工具类不会实例化，并且保证所有的访问都是通过类名来进行的。需要注意一点的是，此工具类最好不要做继承的打算，因为如果子类可以实例化的话，那就要调用父类的构造函数，可是父类没有可以被访问的构造函数，于是问题就会出现。

注意 如果一个类不允许实例化，就要保证“平常”渠道都不能实例化它。

建议 43：避免对象的浅拷贝

我们知道一个类实现了 Cloneable 接口就表示它具备了被拷贝的能力，如果再覆写 clone() 方法就会完全具备拷贝能力。拷贝是在内存中进行的，所以在性能方面比直接通过 new 生成对象要快很多，特别是在大对象的生成上，这会使性能的提升非常显著。但是对象拷贝也有一个比较容易忽略的问题：浅拷贝（Shadow Clone，也叫做影子拷贝）存在对象属性拷贝不彻底的问题。我们来看这样一段代码：

```

public class Client {
    public static void main(String[] args) {
        // 定义父亲
        Person f = new Person("父亲");
        // 定义大儿子
        Person s1 = new Person("大儿子", f);
        // 小儿子的信息是通过大儿子拷贝过来的
        Person s2 = s1.clone();
        s2.setName("小儿子");
        System.out.println(s1.getName() + " 的父亲是 " + s1.getFather().getName());
        System.out.println(s2.getName() + " 的父亲是 " + s2.getFather().getName());
    }
}

class Person implements Cloneable{
    // 姓名
    private String name;
    // 父亲
    private Person father;

    public Person(String _name){
        name = _name;
    }
    public Person(String _name,Person _parent){
        name = _name;
        father = _parent;
    }
    /*name 和 parent 的 getter/setter 方法省略 */

    // 拷贝的实现
    @Override
    public Person clone(){
        Person p = null;
        try {
            p = (Person) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return p;
    }
}

```

程序中，我们描述了这样一个场景：一个父亲，有两个儿子，大小儿子同根同种，所以小儿子对象就通过拷贝大儿子对象来生成，运行输出的结果如下：

```
大儿子 的父亲是 父亲
小儿子 的父亲是 父亲
```

这很正确，没有问题。突然有一天，父亲心血来潮想让大儿子去认个干爹，也就是大儿子的父亲名称需要重新设置一下，代码如下：

```
public static void main(String[] args) {
    // 定义父亲
    Person f = new Person("父亲");
    // 定义大儿子
    Person s1 = new Person("大儿子", f);
    // 小儿子的信息是通过大儿子拷贝过来的
    Person s2 = s1.clone();
    s2.setName("小儿子");
    // 认干爹
    s1.getFather().setName("干爹");
    System.out.println(s1.getName() + " 的父亲是 " + s1.getFather().getName());
    System.out.println(s2.getName() + " 的父亲是 " + s2.getFather().getName());
}
```

上面仅仅修改了加粗字体部分，大儿子重新设置了父亲名称，我们期望的输出是：将大儿子父亲的名称修改为干爹，小儿子的父亲名称保持不变。下面来检查一下结果是否如此：

```
大儿子 的父亲是 干爹
小儿子 的父亲是 干爹
```

怎么回事，小儿子的父亲也成了“干爹”？两个儿子都没有，岂不是要气死“父亲”了！出现这个问题的原因就在于 `clone` 方法，我们知道所有类都继承自 `Object`，`Object` 提供了一个对象拷贝的默认方法，即上面代码中的 `super.clone` 方法，但是该方法是有缺陷的，它提供的是一种浅拷贝方式，也就是说它并不会把对象的所有属性全部拷贝一份，而是有选择性的拷贝，它的拷贝规则如下：

(1) 基本类型

如果变量是基本类型，则拷贝其值，比如 `int`、`float` 等。

(2) 对象

如果变量是一个实例对象，则拷贝地址引用，也就是说此时新拷贝出的对象与原有对象共享该实例变量，不受访问权限的限制。这在 Java 中是很疯狂的，因为它突破了访问权限的定义：一个 `private` 修饰的变量，竟然可以被两个不同的实例对象访问，这让 Java 的访问权限体系情何以堪！

(3) String 字符串

这个比较特殊，拷贝的也是一个地址，是个引用，但是在修改时，它会从字符串池

(String Pool) 中重新生成新的字符串，原有的字符串对象保持不变，在此处我们可以认为 String 是一个基本类型。(有关字符串的知识详见第 4 章。)

明白了这三个规则，上面的例子就很清晰了，小儿子对象是通过拷贝大儿子产生的，其父亲都是同一个人，也就是同一个对象，大儿子修改了父亲名称，小儿子也就跟着修改了——于是，父亲的两个儿子都没了！其实要更正也很简单，clone 方法的代码如下：

```
public Person clone() {
    Person p = null;
    try {
        p = (Person) super.clone();
        p.setFather(new Person(p.getFather().getName()));
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return p;
}
```

然后再运行，小儿子的父亲就不会是“干爹”了。如此就实现了对象的深拷贝 (Deep Clone)，保证拷贝出来的对象自成一体，不受“母体”的影响，和 new 生成的对象没有任何区别。

注意 浅拷贝只是 Java 提供的一种简单拷贝机制，不便于直接使用。

建议 44：推荐使用序列化实现对象的拷贝

上一个建议说了对象的浅拷贝问题，实现 Cloneable 接口就具备了拷贝能力，那我们来思考这样一个问题：如果一个项目中有大量的对象是通过拷贝生成的，那我们该如何处理？每个类都写一个 clone 方法，并且还要深拷贝？想想看这是何等巨大的工作量呀，是否有更好的方法呢？

其实，可以通过序列化方式来处理，在内存中通过字节流的拷贝来实现，也就是把母对象写到一个字节流中，再从字节流中将其读出来，这样就可以重建一个新对象了，该新对象与母对象之间不存在引用共享的问题，也就相当于深拷贝了一个新对象，代码如下：

```
public class CloneUtils {
    // 拷贝一个对象
    @SuppressWarnings("unchecked")
    public static <T extends Serializable> T clone(T obj) {
        // 拷贝产生的对象
        T clonedObj = null;
        try {
            // 读取对象字节数据
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(baos);
            oos.writeObject(obj);
            oos.close();
            ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(bais);
            clonedObj = (T) ois.readObject();
            ois.close();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```

        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(obj);
        oos.close();
        // 分配内存空间，写入原始对象，生成新对象
        ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bais);
        // 返回新对象，并做类型转换
        clonedObj = (T)ois.readObject();
        ois.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return clonedObj;
}
}

```

此工具类要求被拷贝的对象必须实现 `Serializable` 接口，否则是没办法拷贝的（当然，使用反射那是另外一种技巧），上一个建议中的例子只要稍微修改一下即可实现深拷贝，代码如下：

```

class Person implements Serializable{
    private static final long serialVersionUID = 1611293231L;
    /* 删除掉 clone 方法，其他代码保持不变 */
}

```

被拷贝的类只要实现 `Serializable` 这个标志性接口即可，不需要任何实现，当然 `serialVersionUID` 常量还是要加上去的，然后我们就可以通过 `CloneUtils` 工具进行对象的深拷贝了。用此方法进行对象拷贝时需要注意两点：

(1) 对象的内部属性都是可序列化的

如果有内部属性不可序列化，则会抛出序列化异常，这会让调试者很纳闷：生成一个对象怎么会出现序列化异常呢？从这一点来考虑，也需要把 `CloneUtils` 工具的异常进行细化处理。

(2) 注意方法和属性的特殊修饰符

比如 `final`、`static` 变量的序列化问题会被引入到对象拷贝中来（参考第 1 章），这点需要特别注意，同时 `transient` 变量（瞬态变量，不进行序列化的变量）也会影响到拷贝的效果。

当然，采用序列化方式拷贝时还有一个更简单的办法，即使用 Apache 下的 `commons` 工具包中的 `SerializationUtils` 类，直接使用更加简洁方便。

建议 45：覆写 `equals` 方法时不要识别不出自己

我们在写一个 `JavaBean` 时，经常会覆写 `equals` 方法，其目的是根据业务规则判断两个对象是否相等，比如我们写一个 `Person` 类，然后根据姓名判断两个实例对象是否相同，这在

DAO (Data Access Objects) 层是经常用到的。具体操作是先从数据库中获得两个 DTO (Data Transfer Object, 数据传输对象), 然后判断它们是否是相等的, 代码如下:

```
class Person{
    private String name;

    public Person(String _name){
        name = _name;
    }
    /*name 的 getter/setter 方法省略 */

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Person){
            Person p = (Person) obj;
            return name.equalsIgnoreCase(p.getName().trim());
        }
        return false;
    }
}
```

覆写的 equals 做了多个校验, 考虑到从 Web 上传递过来的对象有可能输入了前后空格, 所以用 trim 方法剪切一下, 看看代码有没有问题, 我们写一个 main:

```
public static void main(String[] args) {
    Person p1 = new Person("张三");
    Person p2 = new Person("张三 ");

    List<Person> l = new ArrayList<Person>();
    l.add(p1);
    l.add(p2);
    System.out.println("列表中是否包含张三: "+l.contains(p1));
    System.out.println("列表中是否包含张三 : "+l.contains(p2));
}
```

上面的代码产生了两个 Person 对象 (注意 p2 变量中的那个张三后面有一个空格), 然后放到 List 中, 最后判断 List 是否包含了这两个对象。看上去没有问题, 应该打印出两个 true 才是, 但是结果却是:

```
列表中是否包含张三: true
列表中是否包含张三 : false
```

刚刚放到 list 中的对象竟然说没有, 这太让人失望了, 原因何在呢? List 类检查是否包含元素时是通过调用对象的 equals 方法来判断的, 也就是说 contains(p2) 传递进去, 会依次执行 p2.equals(p1)、p2.equals(p2), 只要有一个返回 true, 结果就是 true, 可惜的是比较结果都是 false, 那问题就出来了: 难道 p2.equals(p2) 也为 false 不成?

还真说对了，`p2.equals(p2)` 确实是 `false`，看看我们的 `equals` 方法，它把第二个参数进行了剪切！也就是说比较的是如下等式：

```
"张三".equalsIgnoreCase("张三")
```

注意前面的“张三”是有空格的，那这个结果肯定是 `false` 了，错误也就此产生了。这是一个想做好事却办成了“坏事”的典型案例，它违背了 `equals` 方法的自反性原则：对于任何非空引用 `x`，`x.equals(x)` 应该返回 `true`。

问题知道了，解决也非常容易，只要把 `trim()` 去掉即可，注意解决的只是当前问题，该 `equals` 方法还存在其他问题。

建议 46：`equals` 应该考虑 null 值情景

继续上一建议的问题，我们解决了覆写 `equals` 的自反性问题，是不是就很完美了呢？再把 `main` 方法重构一下：

```
public static void main(String[] args) {
    Person p1 = new Person("张三");
    Person p2 = new Person(null);

    /* 其他部分没有任何修改，不再赘述 */
}
```

很小的改动，那运行结果是什么呢？是两个 `true` 吗？我们来看运行结果：

```
列表中是否包含张三: true
Exception in thread "main" java.lang.NullPointerException
```

竟然抛异常了！为什么 `p1` 就能在 `List` 中检查一遍，并且执行 `p1.equals` 方法，而到了 `p2` 就开始报错了呢？仔细分析一下程序，马上明白了：当执行到 `p2.equals(p1)` 时，由于 `p2` 的 `name` 是一个 `null` 值，所以调用 `name.equalsIgnoreCase` 方法时就会报空指针异常！出现这种情形是因为覆写 `equals` 没有遵循对称性原则：对于任何引用 `x` 和 `y` 的情形，如果 `x.equals(y)` 返回 `true`，那么 `y.equals(x)` 也应该返回 `true`。

问题知道了，解决也很简单，增加 `name` 是否为空进行判断即可，修改后的 `equals` 代码如下：

```
public boolean equals(Object obj) {
    if(obj instanceof Person){
        Person p = (Person) obj;
        if(p.getName() == null || name == null){
            return false;
        }else{
            return name.equalsIgnoreCase(p.getName());
        }
    }
}
```

```

        }
    }
    return false;
}
}

```

建议 47：在 equals 中使用 getClass 进行类型判断

本节我们继续讨论覆写 equals 的问题。这次我们编写一个员工 Employee 类继承 Person 类，这很正常，员工也是人嘛，而且在 JEE 中 JavaBean 有继承关系也很常见，代码如下：

```

class Employee extends Person{
    private int id;
    /*id 的getter/setter 方法省略*/
    public Employee(String _name,int _id) {
        super(_name);
        id = _id;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Employee){
            Employee e = (Employee) obj;
            return super.equals(obj)&& e.getId() == id;
        }
        return false;
    }
}

```

员工类增加了工号 ID 属性，同时也覆写了 equals 方法，只有在姓名和 ID 号都相同的情况下才表示是同一个员工，这是为了避免在一个公司中出现同名同姓员工的情况。看看上面的代码，这里校验条件已经相当完备了，应该不会再出错了，那我们编写一个 main 方法来看看，代码如下：

```

public static void main(String[] args) {
    Employee e1 = new Employee("张三",100);
    Employee e2 = new Employee("张三",1001);
    Person p1 = new Person("张三");
    System.out.println(p1.equals(e1));
    System.out.println(p1.equals(e2));
    System.out.println(e1.equals(e2));
}

```

上面定义了 2 个员工和 1 个社会闲杂人员，虽然他们同名同姓，但肯定不是同一个，输出应该都是 false，那我们看看运行结果：

```
true
```

```
true
false
```

很不给力嘛，`p1` 竟然等于 `e1`，也等于 `e2`，为什么不是同一个类的两个实例竟然也会相等呢？这很简单，因为 `p1.equals(e1)` 是调用父类 `Person` 的 `equals` 方法进行判断的，它使用 `instanceof` 关键字检查 `e1` 是否是 `Person` 的实例，由于两者存在继承关系，那结果当然是 `true` 了，相等也就没有任何问题了，但是反过来就不成立了，`e1` 或 `e2` 可不等于 `p1`，这也是违反对称性原则的一个典型案例。

更玄的是 `p1` 与 `e1`、`e2` 相等，但 `e1` 竟然与 `e2` 不相等，似乎一个简单的等号传递都不能实现。这才是我们要分析的真正重点：`e1.equals(e2)` 调用的是子类 `Employee` 的 `equals` 方法，不仅仅要判断姓名相同，还要判断工号是否相同，两者工号是不同的，不相等也是自然的了。等式不传递是因为违反了 `equals` 的传递性原则，传递性原则是指对于实例对象 `x`、`y`、`z` 来说，如果 `x.equals(y)` 返回 `true`，`y.equals(z)` 返回 `true`，那么 `x.equals(z)` 也应该返回 `true`。

这种情况发生的关键是父类使用了 `instanceof` 关键字，它是用来判断是否是一个类的实例对象的，这很容易让子类“钻空子”。想要解决也很简单，使用 `getClass` 来代替 `instanceof` 进行类型判断，`Person` 类的 `equals` 方法修改后如下所示：

```
public boolean equals(Object obj) {
    if(obj!=null && obj.getClass() == this.getClass()){
        Person p = (Person) obj;
        if(p.getName()==null || name==null){
            return false;
        }else{
            return name.equalsIgnoreCase(p.getName());
        }
    }
    return false;
}
```

当然，考虑到 `Employee` 也有可能被继承，也需要把它的 `instanceof` 修改为 `getClass`。总之，在覆写 `equals` 时建议使用 `getClass` 进行类型判断，而不要使用 `instanceof`。

建议 48：覆写 `equals` 方法必须覆写 `hashCode` 方法

覆写 `equals` 方法必须覆写 `hashCode` 方法，这条规则基本上每个 Javaer 都知道，这也是 JDK API 上反复说明的，不过为什么要这样做呢？这两个方法之间有什么关系呢？本建议就来解释该问题，我们先来看如下代码：

```
public static void main(String[] args) {
    // Person 类的实例作为 Map 的 key
    Map<Person, Object> map = new HashMap<Person, Object>()
```

```

    {
        put(new Person("张三"), new Object());
    }
};

// Person 类的实例作为 List 的元素
List<Person> list = new ArrayList<Person>() {
{
    add(new Person("张三"));
}
};

// 列表中是否包含
boolean b1 = list.contains(new Person("张三"));
// Map 中是否包含
boolean b2 = map.containsKey(new Person("张三"));
}
}

```

代码中的 Person 类与上一建议相同，equals 方法完美无缺。在这段代码中，我们在声明时直接调用方法赋值，这其实也是一个内部匿名类的操作（下一个建议会详细说明）。现在的问题是 b1 和 b2 这两个 boolean 值是否都为 true？

我们先来看 b1，Person 类的 equals 覆写了，不再判断两个地址是否相等，而是根据人员的姓名来判断两个对象是否相等，所以不管我们的 new Person (“张三”) 产生了多少个对象，它们都是相等的。把 “张三” 对象放入 List 中，再检查 List 中是否包含，那结果肯定是 true 了。

接着来看 b2，我们把张三这个对象作为了 Map 的键 (Key)，放进去的对象是张三，检查的对象还是张三，那应该和 List 的结果相同了，但是很遗憾，结果是 false。原因何在呢？

原因就是 HashMap 的底层处理机制是以数组的方式保存 Map 条目 (Map Entry) 的，这其中的关键是这个数组下标的处理机制：依据传入元素 hashCode 方法的返回值决定其数组的下标，如果该数组位置上已经有了 Map 条目，且与传入的键值相等则不处理，若不相等则覆盖；如果数组位置没有条目，则插入，并加入到 Map 条目的链表中。同理，检查键是否存在也是根据哈希码确定位置，然后遍历查找键值的。

接着深入探讨，那对象元素的 hashCode 方法返回的是什么值呢？它是一个对象的哈希码，是由 Object 类的本地方法生成的，确保每个对象有一个哈希码（这也是哈希算法的基本要求：任意输入 k，通过一定算法 f(k)，将其转换为非可逆的输出，对于两个输入 k1 和 k2，要求若 k1=k2，则必须 f(k1)=f(k2)，但也允许 k1 ≠ k2, f(k1)=f(k2) 的情况存在）。

那回到我们的例子上，由于我们没有重写 hashCode 方法，两个张三对象的 hashCode 方法返回值（也就是哈希码）肯定是不相同的了，在 HashMap 的数组中也就找不到对应的 Map 条目了，于是就返回了 false。

问题清楚了，修改也非常简单，重写一下 hashCode 方法即可，代码如下：

```
class Person {
```

```

/* 其他代码相同，不再赘述 */

@Override
public int hashCode() {
    return new HashCodeBuilder().append(name).toHashCode();
}
}

```

其中 `HashCodeBuilder` 是 `org.apache.commons.lang.builder` 包下的一个哈希码生成工具，使用起来非常方便，诸位可以直接在项目中集成。（为什么不直接写 `hashCode` 方法？因为哈希码的生成有很多种算法，自己写麻烦，事儿又多，所以采用拿来主义是最好的方法。）

建议 49：推荐覆写 `toString` 方法

为什么要覆写 `toString` 方法，这个问题很简单，因为 Java 提供的默认 `toString` 方法不友好，打印出来看不懂，不覆写不行，看这样一段代码：

```

public class Client {
    public static void main(String[] args) {
        System.out.println(new Person("张三"));
    }
}

class Person{
    private String name;

    public Person(String _name){
        name = _name;
    }
    /*name 的 getter/setter 方法省略 */
}

```

输出的结果是：`Person@1fc4bec`。如果机器不同，@ 后面的内容也会不同，但格式都是相同的：类名 + @ + `hashCode`，这玩意就是给机器看的，人哪能看得懂呀！这就是因为我们没有覆写 `Object` 类的 `toString` 方法的缘故，修改一下，代码如下所示：

```

public String toString(){
    return String.format("%s.name=%s", this.getClass(), name);
}

```

如此就可以在需要的时候输出可调试信息了，而且也非常友好，特别是在 Bean 流行的项目中（一般的 Web 项目就是这样），有了这样的输出才能更好的 debug，否则查找错误就如海底捞针呀！当然，当 Bean 的属性较多时，自己实现就不可取了，不过可以使用 apache 的 commons 工具包中的 `ToStringBuilder` 类，简洁、实用又方便。

可能有读者要说了，为什么通过 `println` 方法打印一个对象会调用 `toString` 方法？那是源

于 `println` 的实现机制：如果是一个原始类型就直接打印，如果是一个类类型，则打印出其 `toString` 方法的返回值，如此而已！

建议 50：使用 package-info 类为包服务

Java 中有一个特殊的类：`package-info` 类，它是专门为本包服务的，为什么说它特殊呢？主要体现在 3 个方面：

(1) 它不能随便被创建

在一般的 IDE 中，Eclipse、`package-info` 等文件是不能随便被创建的，会报“Type name is not valid”错误，类名无效。在 Java 变量定义规范中规定如下字符是允许的：字母、数字、下划线，以及那个不怎么常用的 \$ 符号，不过中划线可不在之列，那怎么创建这个文件呢？很简单，用记事本创建一个，然后拷贝进去再改一下就成了，更直接的办法就是从别的项目中拷贝过来。

(2) 它服务的对象很特殊

一个类是一类或一组事物的描述，比如 `Dog` 这个类，就是描述“旺财”的，那 `package-info` 这个类是描述什么的呢？它总要有一个被描述或被陈述的对象吧，它是描述和记录本包信息的。

(3) `package-info` 类不能有实现代码

`package-info` 类再怎么特殊也是一个类，也会被编译成 `package-info.class`，但是在 `package-info.java` 文件里不能声明 `package-info` 类。

`package-info` 类还有几个特殊的地方，比如不可以继承，没有接口，没有类间关系（关联、组合、聚合等）等，不再赘述，Java 中既然允许存在这么一个特殊的类，那肯定有其特殊的作用了，我们来看看它的作用，主要表现在以下三个方面：

(1) 声明友好类和包内访问常量

这个比较简单，而且很实用，比如一个包中有很多内部访问的类或常量，就可以统一放到 `package-info` 类中，这样很方便，而且便于集中管理，可以减少友好类到处游走的情况，代码如下：

```
// 这里是包类，声明一个包使用的公共类
class PkgClass{
    public void test(){}
}
// 包常量，只允许包内访问
class PkgConst{
    static final String PACKAGE_CONST="ABC";
}
```

注意以上代码是存放在 `package-info.java` 中的，虽然它没有编写 `package-info` 的实现，但是 `package-info.class` 类文件还是会生成。通过这样的定义，我们把一个包需要的类和常量都放置在本包下，在语义上和习惯上都能让程序员更适应。

(2) 为在包上标注注解提供便利

比如我们要写一个注解（Annotation），查看一个包下的所有对象，只要把注解标注到 package-info 文件中即可，而且在很多开源项目也采用了此方法，比如 Struts2 的 @namespace、Hibernate 的 @FilterDef 等。

(3) 提供包的整体注释说明

如果是分包开发，也就是说一个包实现了一个业务逻辑或功能点或模块或组件，则该包需要有一个很好的说明文档，说明这个包是做什么用的，版本变迁历史，与其他包的逻辑关系等，package-info 文件的作用在此就发挥出来了，这些都可以直接定义到此文件中，通过 javadoc 生成文档时，会把这些说明作为包文档的首页，让读者更容易对该包有一个整体的认识。当然在这点上它与 package.htm 的作用是相同的，不过 package-info 可以在代码中维护文档的完整性，并且可以实现代码与文档的同步更新。

解释了这么多，总结成一句话：在需要用到包的地方，就可以考虑一下 package-info 这个特殊类，也许能起到事半功倍的作用。

建议 51：不要主动进行垃圾回收

很久很久以前，在 Java 1.1 的年代里，我们经常会看到 System.gc 这样的调用——主动对垃圾进行回收。不过，在 Java 知识深入人心后，这样的代码就逐渐销声匿迹了——这是好现象，因为主动进行垃圾回收是一个非常危险的动作。

之所以危险，是因为 System.gc 要停止所有的响应（Stop the world），才能检查内存中是否有可回收的对象，这对一个应用系统来说风险极大，如果是一个 Web 应用，所有的请求都会暂停，等待垃圾回收器执行完毕，若此时堆内存（Heap）中的对象少的话则还可以接受，一旦对象较多（现在的 Web 项目是越做越大，框架、工具也越来越多，加载到内存中的对象当然也就更多了），那这个过程就非常耗时了，可能 0.01 秒，也可能是 1 秒，甚至是 20 秒，这就会严重影响到业务的正常运行。

例如，我们写这样一段代码：new String("abc")，该对象没有任何引用，对 JVM 来说就是个垃圾对象。JVM 的垃圾回收器线程第一次扫描（扫描时间不确定，在系统不繁忙的时候执行）时把它贴上一个标签，说“你是可以被回收的”，第二次扫描时才真正地回收该对象，并释放内存空间，如果我们直接调用 System.gc，则是在说“嗨，你，那个垃圾回收器过来检查一下有没有垃圾对象，回收一下”。瞧瞧看，程序主动招来了垃圾回收器，这意味着正在运行着的系统要让出资源，以供垃圾回收器执行，想想看吧，它会把所有的对象都检查一遍，然后处理掉那些垃圾对象。注意哦，是检查每个对象。

不要调用 System.gc，即使经常出现内存溢出也不要调用，内存溢出是可分析的，是可以查找出原因的，GC 可不是一个好招数！



Although the world is full of suffering, it is full also of the overcoming of it.

虽然世界充满了苦难，但总是能战胜的。

——Hellen Keller (海伦·凯勒，美国作家)

在 Class 的班级里，Object 班主任问道，“大家知道谁是我们班最受欢迎的同学吗？”

大家面面相觑，不解其意，既而交头接耳，窃窃私语，猛然间，所有的目光都投向了 String 同学，时光仿佛戛然而止，String 诧异地左顾右盼，然后羞红了脸，慢慢地低下了头，同时右手缓缓地举起来，直至胳膊完全伸直，形成一个大大“1”字。

建议 52：推荐使用 String 直接量赋值

一般对象都是通过 new 关键字生成的，但是 String 还有第二种生成方式，也就是我们经常使用的直接声明方式，比如 Str str = "a"，即是通过直接量“a”进行赋值的。对于 String 对象来说，这种方式是极力推荐的，但不建议使用 new String("a") 的方式赋值。为什么呢？我们来看一段程序：

```
public class Client {
    public static void main(String[] args) {
        String str1= "中国";
        String str2 = "中国";
        String str3 = new String("中国");
        String str4 = str3.intern();
        // 两个直接量是否相等
        boolean b1 = (str1==str2);
        // 直接量和对象是否相等
        boolean b2 = (str1 == str3);
        // 经过 intern 处理后的对象与直接量是否相等
        boolean b3 = (str1 == str4);
    }
}
```

注意看上面的程序，我们使用 “==” 判断的是两个对象的引用地址是否相同，也就是判断是否为同一个对象，打印的结果是 true, false, true。即有两个直接量是同一个对象（经过 intern 处理后的 String 与直接量是同一个对象），但直接通过 new 生成的对象却与之不相等，原因何在？

原因是 Java 为了避免在一个系统中大量产生 String 对象（为什么会大量产生？因为 String 字符串是程序中最经常使用的类型），于是就设计了一个字符串池（也有叫做字符串常量池，String Pool 或 String Constant Pool 或 String Literal Pool），在字符串池中所容纳的都是 String 字符串对象，它的创建机制是这样的：创建一个字符串时，首先检查池中是否有字面值相等的字符串，如果有，则不再创建，直接返回池中该对象的引用，若没有则创建之，然后放到池中，并返回新建对象的引用，这个池和我们平常所说的池概念非常相似。对于此例子来说，就是在创建第一个“中国”字符串时，先检查字符串池中有没有该对象，发现没有，于是就创建了“中国”这个字符串并放到池中，待再创建 str2 字符串时，由于池中已经有了该字符串，于是就直接返回了该对象的引用，此时，str1 和 str2 指向的是同一个地址，

所以使用“==”来判断那当然是相等的了。

那为什么使用 new String(“中国”)就不相等了呢？因为直接声明一个 String 对象是不检查字符串池的，也不会把对象放到池中，那当然“==”为 false 了。

那为什么使用 intern 方法处理后就又相等了呢？因为 intern 会检查当前的对象在对象池中是否有字面值相同的引用对象，如果有则返回池中对象，如果没有则放置到对象池中，并返回当前对象。

可能有读者要问了，对象放到池中会不会产生线程安全问题呀？好问题，不过 Java 已经考虑到了，String 类是一个不可变（Immutable）对象其实有两层意思：一是 String 类是 final 类，不可继承，不可能产生一个 String 的子类；二是在 String 类提供的所有方法中，如果有 String 返回值，就会新建一个 String 对象，不对原对象进行修改，这也就保证了原对象是不可改变的。

还有读者问了，放到池中，是不是要考虑垃圾回收问题呀？不用考虑了，虽然 Java 的每个对象都保存在堆内存中，但是字符串池非常特殊，它在编译期已经决定了其存在 JVM 的常量池（Constant Pool），垃圾回收器是不会对它进行回收的。

通过上面的介绍，我们发现 Java 在字符串的创建方面确实提供了非常好的机制，利用对象池不仅可以提高效率，同时也减少了内存空间的占用，建议大家在开发中使用直接量赋值方式，除非确有必要才新建立一个 String 对象。

建议 53：注意方法中传递的参数要求

有这样一个简单需求：写一个方法，实现从原始字符串中删除与之匹配的所有子字符串，比如在“蓝蓝的天，白云飘”中，删除“白云飘”，输出“蓝蓝的天，”，代码如下：

```
public class StringUtils {
    // 删除字符串
    public static String remove(String source, String sub) {
        return source.replaceAll(sub, "");
    }
}
```

StringUtils 工具类很简单，它采用了 String 的 replace 方法，该方法是做字符串替换的，我们来编写一个测试用例，检查 remove 方法是否正确，如下所示：

```
assertTrue(StringUtils.remove("好是好", "好").equals("是"));
```

测试的结果是绿条（Green Bar），正确无误，但是再看看如下的测试用例：

```
assertTrue(StringUtils.remove("$是$", "$").equals("是"));
```

上面只是把“好是好”中的两个“好”字替换成一个“\$”符号，猜猜结果会是什么，

应该也是绿条吧？但是非常遗憾，结果是红条，测试未通过。就这么简单一个的替换，为什么测试通不过呢？

问题就出在了 `replaceAll` 方法上，该方法确实需要传递两个 `String` 类型的参数，也确实进行了字符串替换，但是它要求第一个参数是一个正则表达式，符合正则表达式的字符串才会被替换。对上面的例子来说，第一个测试案例传递进来的是一个字符串“好”，这是一个全匹配查找替换，处理得非常正确，第二个测试案例传递进来的是“\$”符号，“\$”符号在正则表达式中表示的是字符串的结束位置，也就是说执行完 `repalceAll` 后，在字符串结尾的地方加上了空字符串，其结果还是“\$是\$”，所以测试失败也就在所难免了。问题清楚了，解决方案也就出来了：使用 `replace` 方法替代即可，它是 `repalceAll` 方法的简化版，可传递两个 `String` 参数继续替换，与我们的编码意图是相吻合的。

读者如果注意看 JDK 文档，会发现 `replace(CharSequence target,CharSequence replacement)` 方法是在 1.5 版本以后才开始提供的，在此之前如果要对一个字符串进行全替换，只能使用 `replaceAll` 方法，不过由于 `replaceAll` 方法的第二个参数使用了正则表达式，而且参数类型只要是 `CharSequence` 就可以（`String` 的父类），所以很容易让使用者误解，稍有不慎就会导致严重的替换错误。

注意 `replaceAll` 传递的第一个参数是正则表达式。

建议 54：正确使用 `String`、`StringBuffer`、`StringBuilder`

`CharSequence` 接口有三个实现类与字符串有关：`String`、`StringBuffer`、`StringBuilder`，虽然它们都与字符串有关，但是其处理机制是不同的。

`String` 类是不可改变的量，也就是创建后就不能再修改了，比如创建了一个“abc”这样的字符串对象，那么它在内存中永远都会是“abc”这样具有固定表面值的一个对象，不能被修改，即使想通过 `String` 提供的方法来尝试修改，也是要么创建一个新的字符串对象，要么返回自己，比如：

```
String str = "abc";
String str1 = str.substring(1);
```

其中，`str` 是一个字符串对象，其值是“abc”，通过 `substring` 方法又重新生成了一个字符串 `str1`，它的值是“bc”，也就是说 `str` 引用的对象一旦产生就永远不会改变。为什么上面还说有可能不创建对象而返回自己呢？那是因为采用 `str.substring(0)` 就不会创建新对象，JVM 会从字符串池中返回 `str` 的引用，也就是自身的引用。

`StringBuffer` 是一个可变字符序列，它与 `String` 一样，在内存中保存的都是一个有序的字符序列（`char` 类型的数组），不同点是 `StringBuffer` 对象的值是可改变的，例如：

```
StringBuffer sb = new StringBuffer("a");
sb.append("b");
```

从上面的代码可以看出 sb 的值在改变，初始化的时候是“a”，经过 append 方法后，其值变成了“ab”。可能有读者会问了，这与 String 类通过“+”连接有什么区别？例如：

```
String s = "a";
s = s + "b";
```

有区别，字符串变量 s 初始化时是“a”对象的引用，经过加号计算后，s 变量就修改为了“ab”的引用，但是初始化的“a”对象还是没有改变，只是变量 s 指向了新的引用地址。再看看 StringBuffer 的对象，它的引用地址虽不变，但值在改变。

StringBuilder 与 StringBuffer 基本相同，都是可变字符序列，不同点是：StringBuffer 是线程安全的，StringBuilder 是线程不安全的，翻翻两者的源代码，就会发现在 StringBuffer 的方法前都有 synchronized 关键字，这也是 StringBuffer 在性能上远低于 StringBuilder 的原因。

在性能方面，由于 String 类的操作都是产生新的 String 对象，而 StringBuilder 和 StringBuffer 只是一个字符数组的再扩容而已，所以 String 类的操作要远慢于 StringBuffer 和 StringBuilder。

弄清楚了三者的原理，我们就可以在不同的场景下使用不同的字符序列了：

(1) 使用 String 类的场景

在字符串不经常变化的场景中可以使用 String 类，例如常量的声明、少量的变量运算等。

(2) 使用 StringBuffer 类的场景

在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在多线程的环境中，则可以考虑使用 StringBuffer，例如 XML 解析、HTTP 参数解析和封装等。

(3) 使用 StringBuilder 类的场景

在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在单线程的环境中，则可以考虑使用 StringBuilder，如 SQL 语句的拼装、JSON 封装等。

注意 在适当的场景选用字符串类型。

建议 55：注意字符串的位置

看这样一段程序：

```
public static void main(String[] args) {
    String str1 = 1 + 2 + " apples";
    String str2 = "apples:" + 1 + 2;
}
```

想想看这两个字符串输出的苹果数量是否一致？如果一致，那是几个呢？

答案是不一致，str1 的值是“3 apples”，str2 的值是“apples:12”，这中间悬殊很大，只是把“apples”调换了一下位置，为何会发生如此大的变化呢？

这都源于 Java 对加号的处理机制：在使用加号进行计算的表达式中，只要遇到 String 字符串，则所有的数据都会转换为 String 类型进行拼接，如果是原始数据，则直接拼接，如果是对象，则调用 `toString` 方法的返回值然后拼接，如：

```
str = str + new ArrayList();
```

上面就是调用 `ArrayList` 对象的 `toString` 方法返回值进行拼接的。再回到前面的问题上，对于 str1 字符串，Java 的执行顺序是从左到右，先执行 $1+2$ ，也就是算数加法运算，结果等于 3，然后再与字符串进行拼接，结果就是“3 apples”，其形式类似于如下计算：

```
String str1 = (1 + 2) + " apples";
```

而对于 str2 字符串，由于第一个参与运算的是 String 类型，加上 1 后的结果是“apples:1”，这仍然是一个字符串，然后再与 2 相加，其结果还是一个字符串，也就是“apples:12”。这说明如果第一个参数是 String，则后续的所有计算都会转变成 String 类型，谁让字符串是老大呢！

注意 在“+”表达式中，String 字符串具有最高优先级。

建议 56：自由选择字符串拼接方法

对一个字符串进行拼接有三种方法：加号、`concat` 方法及 `StringBuilder`（或 `StringBuffer`，由于 `StringBuffer` 的方法与 `StringBuilder` 相同，文中不再赘述）的 `append` 方法，其中加号是最常用的，其他两种方式偶尔会出现在一些开源项目中，那这三者之间有什么区别吗？我们来看下面的例子：

```
// 加号拼接
str += "c";
//concat 方法连接
str = str.concat("c");
```

上面是两种不同的字符串拼接方式，循环 5 万次后再检查其执行的时间，加号方式的执行时间是 1438 毫秒，而 `concat` 方法的执行时间是 703 毫秒，时间相差 1 倍，如果使用 `StringBuilder` 方式，执行时间会更少，其代码如下：

```
public static void doWithStringBuffer() {
    StringBuilder sb = new StringBuilder("a");
    for(int i=0;i<50000;i++) {
```

```

        sb.append("c");
    }
    String str = sb.toString();
}
}

```

StringBuffer 的 append 方法的执行时间是 0 毫秒，说明时间非常非常短暂（毫秒不足以计时，读者可以使用纳秒进行计算）。这个实验也说明在字符串拼接方式中，append 方法最快，concat 方法次之，加号最慢，这是为何呢？

(1) “+”方法拼接字符串

虽然编译器对字符串的加号做了优化，它会使用 StringBuilder 的 append 方法进行追加，按道理来说，其执行时间也应该是 0 毫秒，不过它最终是通过 toString 方法转换成 String 字符串的，例子中“+”拼接的代码与如下代码相同：

```
str = new StringBuilder(str).append("c").toString();
```

注意看，它与纯粹使用 StringBuilder 的 append 方法是不同的：一是每次循环都会创建一个 StringBuilder 对象，二是每次执行完毕都要调用 toString 方法将其转换为字符串——它的执行时间就是耗费在这里了！

(2) concat 方法拼接字符串

我们从源码上看一下 concat 方法的实现，代码如下：

```

public String concat(String str) {
    int otherLen = str.length();
    // 如果追加的字符串长度为 0，则返回字符串本身
    if (otherLen == 0) {
        return this;
    }
    // 字符数组，容纳的是新字符串的字符
    char buf[] = new char[count + otherLen];
    // 取出原始字符串放到 buf 数组中
    getChars(0, count, buf, 0);
    // 追加的字符串转化成字符数组，添加到 buf 中
    str.getChars(0, otherLen, buf, count);
    // 复制字符数组，产生一个新的字符串
    return new String(0, count + otherLen, buf);
}

```

其整体看上去就是一个数组拷贝，虽然在内存中的处理都是原子性操作，速度非常快，不过，注意看最后的 return 语句，每次的 concat 操作都会新创建一个 String 对象，这就是 concat 速度慢下来的真正原因，它创建了 5 万个 String 对象呀！

(3) append 方法拼接字符串

StringBuilder 的 append 方法直接由父类 AbstractStringBuilder 实现，其代码如下：

```
public AbstractStringBuilder append(String str) {
```

```

// 如果是 null 值，则把 null 作为字符串处理
if (str == null) str = "null";
int len = str.length();
// 字符串长度为 0，则返回自身
if (len == 0) return this;
int newCount = count + len;
// 追加后的字符数组长度是否超过当前值
if (newCount > value.length)
    expandCapacity(newCount); // 加长，并做数组拷贝
// 字符串复制到目标数组
str.getChars(0, len, value, count);
count = newCount;
return this;
}

```

看到没，整个 append 方法都在做字符数组处理，加长，然后数组拷贝，这些都是基本的数据处理，没有新建任何对象，所以速度也就最快了！注意：例子中是在最后通过 StringBuffer 的 toString 返回了一个字符串，也就是说在 5 万次循环结束后才生成了一个 String 对象。

三者的实现方法不同，性能也就不同，但并不表示我们一定要使用 StringBuilder，这是因为“+”非常符合我们的编码习惯，适合人类阅读，两个字符串拼接，就用加号连一下，这很正常，也很友好，在大多数情况下我们都可以使用加号操作，只有在系统性能临界（如在性能“增之一分则太长”的情况下）的时候才可以考虑使用 concat 或 append 方法。而且，很多时候系统 80% 的性能是消耗在 20% 的代码上的，我们的精力应该更多的投入到算法和结构上。

注意 适当的场景使用适当的字符串拼接方式。

建议 57：推荐在复杂字符串操作中使用正则表达式

字符串的操作，诸如追加、合并、替换、倒序、分割等，都是在编码过程中经常用到的，而且 Java 也提供了 append、replace、reverse、split 等方法来完成这些操作，它们使用起来也确实方便，但是更多的时候，需要使用正则表达式来完成复杂的处理，我们来看一个例子：统计一篇文章中英文单词的数量，很简单吧？代码如下：

```

public static void main(String[] args) {
    // 接收键盘输入
    Scanner input = new Scanner(System.in);
    while(input.hasNext()){
        String str = input.nextLine();
        // 使用 split 方法分隔后统计
        int wordsCount = str.split(" ").length;
    }
}

```

```

        System.out.println(str + " 单词数: " + wordsCount);
    }
}
}

```

使用 split 方法根据空格来分割单词，然后计算分隔后的数组长度，这种方法可靠吗？可行吗？我们来看输出：

```

Today is Monday
Today is Monday 单词数: 3
Today is Monday
Today is Monday 单词数: 4
Today is Monday?No!
Today is Monday?No! 单词数: 3
I'm Ok.
I'm Ok. 单词数: 2

```

注意看输出，除了第一个输入“Todady is Monay”正确外，其他都是错误的！第二条输入中单词“Monday”前有 2 个连续的空格，第三条输入中“NO”单词的前后都没有空格，最后一个输入则没有把连写符号“!”考虑进去，这样统计出来的单词数量肯定错误一堆，那怎么做才合理呢？

如果考虑使用一个循环来处理这样的“异常”情况，会使程序的稳定性变差，而且要考虑太多太多的因素，这让程序的复杂性也大大提高了。那如何处理呢？可以考虑使用正则表达式，代码如下：

```

public static void main(String[] args) {
    // 接收键盘输入
    Scanner input = new Scanner(System.in);
    while (input.hasNext()) {
        String str = input.nextLine();
        // 正则表达式对象
        Pattern pattern = Pattern.compile("\b\w+\b");
        // 生成匹配器
        Matcher matcher = pattern.matcher(str);
        // 记录单词数量
        int wordsCount = 0;
        // 遍历查找匹配，统计单词数量
        while (matcher.find()) {
            wordsCount++;
        }
        System.out.println(str + " 单词数: " + wordsCount);
    }
}

```

准不准确，我们来看相同的输入所产生的结果：

```

Today is Monday
Today is Monday 单词数: 3

```

```

Today is Monday
Today is Monday 单词数: 3
Today is Monday?No!
Today is Monday?No! 单词数: 4
I'm Ok.
I'm Ok. 单词数: 3

```

每项的输出都是准确的，而且程序也不复杂，先生成一个正则表达式对象，然后使用匹配器进行匹配，之后通过一个 while 循环统计匹配的数量。需要说明的是，在 Java 的正则表达式中“\b”表示的是一个单词的边界，它是一个位置界定符，一边为字符或数字，另外一边则非字符或数字，例如“A”这样一个输入就有两个边界，即单词“A”的左右位置，这也就说明了为什么要加上“\w”（它表示的是字符或数字）。

正则表达式在字符串的查找、替换、剪切、复制、删除等方面有着非凡的作用，特别是面对大量的文本字符需要处理（如需要读取大量的 LOG 日志）时，使用正则表达式可以大幅地提高开发效率和系统性能，但是正则表达式是一个恶魔（Regular Expressions is evil），它会使程序难以读懂，想想看，写一个包含^、\$、\A、\s、\Q、+、?、()、[]、{}等符号的正则表达式，然后告诉你这是一个“这样，这样……”的字符串查找，你是不是要崩溃了？这代码只有上帝才能看懂了！

注意 正则表达式是恶魔，威力巨大，但难以控制。

建议 58：强烈建议使用 UTF 编码

Java 的乱码问题由来已久，有点经验的开发人员肯定都遇到过乱码问题，有时是从 Web 上接收的乱码，有时是从数据库中读取的乱码，有时是在外部接口中接收到的乱码文件，这些都让我们困惑不已，甚至是痛苦不堪，看如下代码：

```

public static void main(String[] args) throws Exception {
    String str = "汉字";
    // 读取字节
    byte[] b = str.getBytes("UTF-8");
    // 重新生成一个新的字符串
    System.out.println(new String(b));
}

```

Java 文件是通过 IDE 工具默认创建的，编码格式是 GBK，大家想想看上面的输出结果会是什么？可能是乱码吧？两个编码格式不相同。我们暂不公布结果，先解释一下 Java 中的编码规则。Java 程序涉及的编码包括两部分：

(1) Java 文件编码

如果我们使用记事本创建一个 .java 后缀的文件，则文件的编码格式就是操作系统默认

的格式。如果是使用 IDE 工具创建的，如 Eclipse，则依赖于 IDE 的设置，Eclipse 默认是操作系统编码（Windows 一般为 GBK）。

（2）Class 文件编码

通过 javac 命令生成的后缀名为 .class 的文件是 UTF-8 编码的 UNICODE 文件，这在任何操作系统上都是一样的，只要是 class 文件就会是 UNICODE 格式。需要说明的是，UTF 是 UNICODE 的存储和传输格式，它是为了解决 UNICODE 的高位占用冗余空间而产生的，使用 UTF 编码就标志着字符集使用的是 UNICODE。

再回到我们的例子上，getBytes 方法会根据指定的字符集提取出字节数组（这里按照 UNICODE 格式来提取），然后程序又通过 newString(byte[] bytes) 重新生成一个字符串。来看看 String 这个构造函数：通过操作系统默认的字符集解码指定的 byte 数组，构造一个新的 String。结果已经很清楚了，如果操作系统是 UTF-8 编码的话，输出就是正确的，如果不是，则会是乱码。由于这里使用的是默认编码 GBK，那么输出的结果也就是乱码了。我们再详细分解一下运行步骤：

步骤 1 创建 Client.java 文件。

该文件的默认编码 GBK（如果使用 Eclipse，则可以在属性查看到）。

步骤 2 编写代码（如上）。

步骤 3 保存，并使用 javac 编译。

注意我们没有使用“javac -encoding GBK Client.java”显式声明 Java 的编码格式，javac 会自动按照操作系统的编码（GBK）读取 Client.java 文件，然后将其编译成 .class 文件。

步骤 4 生成 .class 文件。

编译结束，生成 .class 文件，并保存到硬盘上。此时 .class 文件使用的是 UTF-8 格式编码的 UNICODE 字符集，可以通过 javap 命令阅读 class 文件。其中“汉字”变量也已经由 GBK 编码转变成 UNICODE 格式了。

步骤 5 运行 main 方法，提取“汉字”的字节数组。

“汉字”原本是按照 UTF-8 格式保存的，要再提取出来当然没有任何问题了。

步骤 6 重组字符串。

读取操作系统的编码格式（GBK），然后重新编码变量 b 的所有字节。问题就在这里产生了：因为 UNICODE 的存储格式是两个字节表示一个字符（注意这里是指 UCS-2 标准），虽然 GBK 也是 2 个字节表示一个字符，但两者之间没有影射关系，要想做转换只能读取映射表，不能实现自动转换——于是 JVM 就按照默认的编码格式（GBK）读取了 UNICODE 的两个字节。

步骤 7 输出乱码，程序运行结束。

问题清楚了，解决方案也随之产生，方案有两个。

步骤 8 修改代码。

明确指定编码即可，代码如下：

```
System.out.println(new String(b, "UTF-8"));
```

步骤 9 修改操作系统的编码方式。

各个操作系统的修改方式不同，不再赘述。

我们可以把从字符串读取字节的过程看作是数据传输的需要（比如网络、存储），而重组字符串则是业务逻辑的需求，这样就可使乱码现场重现：通过 JDBC 读取的字节数组是 GBK 的，而业务逻辑编码时采用的是 UTF-8，于是乱码产生了。对于此类问题，最好的解决办法就是使用统一的编码格式，要么都用 GBK；要么都用 UTF-8，各个组件、接口、逻辑层都用 UTF-8，拒绝独树一帜的情况。

问题解释清楚了，我们再来看以下代码：

```
public class Client {
    public static void main(String[] args) throws Exception {
        String str = "汉字";
        // 读取字节
        byte[] b = str.getBytes("GB2312");
        // 重新生成一个新的字符串
        System.out.println(new String(b));
    }
}
```

仅仅修改了读取字节的编码格式（修改成了 GB2312 格式的），结果会是怎样的呢？又或者将其修改成 GB18030，结果又是怎样的呢？结果都是“汉字”，不是乱码。哈哈，这是因为 GB2312 是中文字符集的 V1.0 版，GBK 是 V2.0 版本，GB18030 是 V3.0 版，版本是向下兼容的，只是它们包含的汉字数量不同而已，注意，UNICODE 可不在这个序列之内的。

注意 一个系统使用统一的编码。

建议 59：对字符串排序持一种宽容的心态

在 Java 中一涉及中文处理就会冒出很多问题来，其中排序也是一个让人头疼的课题，我们来看下面的代码：

```
public static void main(String[] args) {
    String[] strs = {"张三 (Z)", "李四 (L)", "王五 (W)"};
    // 排序，默认是升序
    Arrays.sort(strs);
    int i=0;
    for(String str:strs){
        System.out.println((++i) + "、" + str);
    }
}
```

上面的代码定义一个数组，然后进行升序排序，我们期望的结果是按照拼音升序排列，即为李四、王五、张三，但是结果却不是这样的：

- 1、张三 (Z)
- 2、李四 (L)
- 3、王五 (W)

这是按照什么排序的呀，非常混乱！我们知道 Arrays 工具类的默认排序是通过数组元素的 compareTo 方法来进行比较的，那我们来看 String 类的 compareTo 的主要实现：

```
while (k < lim) {
    // 原字符串的字符数组
    char c1 = v1[k];
    // 比较字符串的字符数组
    char c2 = v2[k];
    if (c1 != c2) {
        // 比较两者的 char 值大小
        return c1 - c2;
    }
    k++;
}
```

上面的代码先取得字符串的字符数组，然后一个一个地比较大小，注意这里是字符比较（减号操作符），也就是 UNICODE 码值的比较，查一下 UNICODE 代码表，“张”的码值是 5F20，而“李”是 674E，这样一看，“张”排在“李”的前面也就很正确了——但这明显与我们的意图冲突了。这一点在 JDK 文档中也有说明：对于非英文的 String 排序可能会出现不准确的情况。那该如何解决这个问题呢？Java 推荐使用 Collator 类进行排序，那好，我们把代码修改一下：

```
public static void main(String[] args) throws Exception {
    String[] strs = {"张三 (Z)", "李四 (L)", "王五 (W)"};
    // 定义一个中文排序器
    Comparator c = Collator.getInstance(Locale.CHINA);
    // 升序排列
    Arrays.sort(strs, c);
    int i=0;
    for(String str:strs){
        System.out.println((++i) + "、" + str);
    }
}
```

输出结果如下：

- 1、李四 (L)
- 2、王五 (W)
- 3、张三 (Z)

这确实是我们期望的结果，应该举杯庆贺了吧！但是且慢，中国的汉字博大精深，Java

是否都能精确的排序呢？最主要的一点是汉字中有象形文字，音形分离，是不是每个汉字都能按照拼音的顺序排列好呢？我们写一个复杂的汉字来看看：

```
public static void main(String[] args) throws Exception {
    String[] strs = {"犇 (B)", "鑫 (X)"};
    Arrays.sort(strs,Collator.getInstance(Locale.CHINA));
    int i=0;
    for(String str:strs){
        System.out.println((++i) + "、" + str);
    }
}
```

三个牛“犇”读 bēn，三个金“鑫”读 xīn，这两个字经常出现在饭店和商店的名称上，我们来看排序的输出结果：

- 1、鑫 (X)
- 2、犇 (B)

输出结果又乱了！不要责怪 Java，它已经尽量为我们考虑了，只是因为我们的汉字文化太博大精深了，要做好这个排序确实有点难为它。更深层次的原因是 Java 使用的是 UNICODE 编码，而中文 UNICODE 字符集是来源于 GB18030 的，GB18030 又是从 GB2312 发展起来，GB2312 是一个包含了 7000 多个字符的字符集，它是按照拼音排序，并且是连续的，之后的 GBK、GB18030 都是在其基础上扩充出来的，所以要让它们完整排序也就难上加难了。

如果是排序对象是经常使用的汉字，使用 Collator 类排序完全可以满足我们的要求，毕竟 GB2312 已经包含了大部分的汉字，如果需要严格排序，则要使用一些开源项目来自己实现了，比如 pinyin4j 可以把汉字转换为拼音，然后我们自己来实现排序算法，不过此时你也会发现要考虑诸如算法、同音字、多音字等众多问题。

注意 如果排序不是一个关键算法，使用 Collator 类即可。



第5章

数组和集合

噢，他明白了，河水既没有牛伯伯说的那么浅，也没有小松鼠说的那么深，只有自己亲自试过才知道。

——寓言故事《小马过河》

数据集处理是每种语言必备的功能，Java 更甚之，数据集可以允许重复，也可以不允许重复，可以允许 null 存在，也可以不允许 null 存在，可以自动排序，也可以不自动排序，可以是阻塞式的，也可以是非阻塞式的，可以是栈，也可以是队列……

本章将围绕我们使用最多的三个数据集合（数组、ArrayList 和 HashMap）来阐述在开发过程中要注意的事项，并由此延伸至 Set、Queue、Stack 等集合。

建议 60：性能考虑，数组是首选

数组在实际的系统开发中用得越来越少了，我们通常只有在阅读一些开源项目时才会看到它们的身影，在 Java 中它确实没有 List、Set、Map 这些集合类用起来方便，但是在基本类型处理方面，数组还是占优势的，而且集合类的底层也都是通过数组实现的，比如对一个数据集求和这样的计算：

```
// 对数组求和
public static int sum(int[] datas) {
    int sum = 0;
    for(int i=0;i<datas.length;i++) {
        sum +=datas[i];
    }
    return sum;
}
```

对一个 int 类型的数组求和，取出所有的数组元素并相加，此算法中如果是基本类型则使用数组效率是最高的，使用集合则效率次之。再看使用 List 求和：

```
// 对列表求和计算
public static int sum(List<Integer> datas) {
    int sum = 0;
    for(int i=0;i<datas.size();i++) {
        sum +=datas.get(i);
    }
    return sum;
}
```

注意看加粗字体，这里其实已经做了一个拆箱动作，Integer 对象通过 intValue 方法自动转换成了一个 int 基本类型，对于性能濒于临界的系统来说该方案是比较危险的，特别是大数量的时候，首先，在初始化 List 数组时要进行装箱动作，把一个 int 类型包装成一个 Integer 对象，虽然有整型池在，但不在整型池范围内的都会产生一个新的 Integer 对象，而且众所周知，基本类型是在栈内存中操作的，而对象则是在堆内存中操作的，栈内存的特点是速度快，容量小，堆内存的特点是速度慢，容量大（从性能上来讲，基本类型的处理占优势）。其次，在进行求和计算（或者其他遍历计算）时要做拆箱动作，因此无谓的性能消耗

也就产生了。

在实际测试中发现：对基本类型进行求和计算时，数组的效率是集合的 10 倍。

注意 性能要求较高的场景中使用数组替代集合。

建议 61：若有必要，使用变长数组

Java 中的数组是定长的，一旦经过初始化声明就不可改变长度，这在实际使用中非常不方便，比如要对班级学生的信息进行统计，因为我们不知道一个班级会有多少学生（随时都可能有学生入学、退学或转学），所以需要有一个足够大的数组来容纳所有的学生，但问题是多大才算足够大？10 年前一台台式机 64MB 的内存已经很牛了，现在要是没有 2GB 的内存你都不好意思跟别人交流计算机的配置，所以呀，这个足够大是相对于当时的场景而言的。随着环境的变化，“足够大”也可能会转变成“足够小”，然后就会出现超出数组最大容量的情况，那该如何解决呢？事实上，通过对数组扩容“婉转”地解决该问题，代码如下：

```
public static <T> T[] expandCapacity(T[] datas, int newLen) {
    // 不能是负值
    newLen = newLen<0?0:newLen;
    // 生成一个新数组，并拷贝原值
    return Arrays.copyOf(datas, newLen);
}
```

上述代码中采用的是 `Arrays` 数组工具类的 `copyOf` 方法，产生了一个 `newLen` 长度的新数组，并把原有的值拷贝了进去，之后就可以对超长的元素进行赋值了（依据类型的不同分别赋值为 0、`false` 或 `null`），使用方法如下：

```
public static void main(String[] args) {
    // 一个班级最多容量 60 个学生
    Stu[] classes = new Stu[60];
    /*classes 初始化 .....

    // 偶尔一个班级可以容纳 80 人，数组加长
    classes = expandCapacity(classes,80);
    /* 重新初始化超过限额的 20 人 .....
}
```

通过这样的处理方式，曲折地解决了数组的变长问题。其实，集合的长度自动维护功能的原理与此类似。在实际开发中，如果确实需要变长的数据集，数组也是在考虑范围之内的，不能因固定长度而将其否定之。

建议 62：警惕数组的浅拷贝

有这样一个例子，第一个箱子里有赤橙黄绿青蓝紫 7 色气球，现在希望在第二个箱子中也放入 7 个气球，其中最后一个气球改为蓝色，也就是赤橙黄绿青蓝蓝 7 个气球，那我们很容易就会想到第二个箱子中的气球可以通过拷贝第一个箱子中的气球来实现，毕竟有 6 个气球是一样的嘛，来看实现代码：

```
public class Client {
    public static void main(String[] args) {
        // 气球数量
        int balloonNum = 7;
        // 第一个箱子
        Balloon[] box1 = new Balloon[balloonNum];
        // 初始化第一个箱子中的气球
        for (int i = 0; i < balloonNum; i++) {
            box1[i] = new Balloon(Color.values()[i], i);
        }
        // 第二个箱子的气球是拷贝的第一个箱子里的
        Balloon[] box2 = Arrays.copyOf(box1, box1.length);
        // 修改最后一个气球颜色
        box2[6].setColor(Color.Blue);
        // 打印出第一个箱子中的气球颜色
        for (Balloon b:box1) {
            System.out.println(b);
        }
    }
    // 气球颜色
    enum Color {
        Red, Orange, Yellow, Green, Indigo, Blue, Violet;
    }
    // 气球
    class Balloon {
        // 编号
        private int id;
        // 颜色
        private Color color;

        public Balloon(Color _color, int _id) {
            color = _color;
            id = _id;
        }
        /*id、color 的 getter/setter 方法省略*/
        //apache-common 包下的 ToStringBuilder 重写 toString 方法
        public String toString() {
            return new ToStringBuilder(this)
                .append("编号", id)
                .append("颜色", color)
        }
    }
}
```

```

        .toString();
    }
}

```

第二个箱子里最后一个气球的颜色毫无疑问是被修改成蓝色了，不过我们是通过拷贝第一个箱子里的气球然后再修改的方式来实现的，那会对第一个箱子的气球颜色有影响吗？我们看输出：

```

Balloon@b2fd8f [编号 =0, 颜色 =Red]
Balloon@a20892 [编号 =1, 颜色 =Orange]
Balloon@158b649 [编号 =2, 颜色 =Yellow]
Balloon@1037c71 [编号 =3, 颜色 =Green]
Balloon@1546e25 [编号 =4, 颜色 =Indigo]
Balloon@8a0d5d [编号 =5, 颜色 =Blue]
Balloon@a470b8 [编号 =6, 颜色 =Blue]

```

最后一个气球颜色竟然也被修改了，我们只是希望修改第二个箱子的气球啊，这是为何？这是很典型的浅拷贝（Shallow Clone）问题，前面第1章的序列化中也介绍过，但是这里与之有一点不同：数组中的元素没有实现 Serializable 接口。

确实如此，通过 copyOf 方法产生的数组是一个浅拷贝，这与序列化的浅拷贝完全相同：基本类型是直接拷贝值，其他都是拷贝引用地址。需要说明的是，数组的 clone 方法也是与此相同的，同样是浅拷贝，而且集合的 clone 方法也都是浅拷贝，这就需要大家在拷贝时多留心了。

问题找到了，解决方案也很简单，遍历 box1 的每个元素，重新生成一个气球（Ballon）对象，并放置到 box2 数组中，代码较简单，不再赘述。

该方法用得最多的地方是在使用集合（如 List）进行业务处理时，比如发觉需要拷贝集合中的元素，可集合没有提供拷贝方法，如果自己写会很麻烦，所以干脆使用 List.toArray 方法转换成数组，然后通过 Arrays.copyOf 拷贝，再转换回集合，简单便捷！但是，非常遗憾的是，这里我们又撞到浅拷贝的枪口上了，虽然很多时候浅拷贝可以解决业务问题，但更多时候会留下隐患，需要我们提防又提防。

建议 63：在明确的场景下，为集合指定初始容量

我们经常使用 ArrayList、Vector、HashMap 等集合，一般都是直接用 new 跟上类名声明出一个集合来，然后使用 add、remove 等方法进行操作，而且因为它是自动管理长度的，所以不用我们特别费心超长的问题，这确实是一个非常好的优点，但也有我们必须要注意的事项。

下面以 ArrayList 为例深入了解一下 Java 是如何实现长度的动态管理的，先从 add 方法的阅读开始，代码如下。

```

public boolean add(E e) {
    // 扩展长度
    ensureCapacity(size + 1);
    // 追加元素
    elementData[size++] = e;
    return true;
}

```

我们知道 ArrayList 是一个大小可变的数组，但它在底层使用的是数组存储（也就是 elementData 变量），而且数组是定长的，要实现动态长度必然要进行长度的扩展，ensureCapacity 方法提供了此功能，代码如下：

```

public void ensureCapacity(int minCapacity) {
    // 修改计数器
    modCount++;
    // 上次（原始）定义的数组长度
    int oldCapacity = elementData.length;
    // 当前需要的长度超过了数组长度
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        // 计算新数组长度
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        // 数组拷贝，生成新数组
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

```

注意看新数组的长度计算方法，并不是增加一个元素，elementData 的长度就加 1，而是在达到 elementData 长度的临界点时，才将 elementData 扩容 1.5 倍，这样实现有什么好处呢？好处是避免了多次调用 copyOf 方法的性能开销，否则每加一个元素都要扩容一次，那性能岂不是非常糟糕？！

可能有读者要问了，这里为什么是 1.5 倍，而不是 2.5 倍、3.5 倍？这是一个好问题，原因是一次扩容太大（比如扩容 2.5 倍），占用的内存也就越大，浪费的内存也就越多（1.5 倍扩容，最多浪费 33% 的数组空间，而 2.5 倍则最多可能浪费 60% 的内存）；而一次扩容太小（比如每次扩容 1.1 倍），则需要多次对数组重新分配内存，性能消耗严重。经过测试验证，扩容 1.5 倍即满足了性能要求，也减少了内存消耗。

现在我们知道了 ArrayList 的扩容原则，那还有一个问题：elementData 的默认长度是多少呢？答案是 10，如果我们使用默认方式声明 ArrayList，如 new ArrayList()，则 elementData 的初始长度就是 10。我们来看 ArrayList 的无参构造：

```

// 无参构造，我们通常用得最多的就是这个
public ArrayList() {

```

```

// 默认是长度为 10 的数组
this(10);
}
// 指定数组长度的有参构造
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+initialCapacity);
    // 声明指定长度的数组，容纳 element
    this.elementData = new Object[initialCapacity];
}

```

默认初始化时声明了一个长度为 10 的数组，在通过 add 方法增加第 11 个元素时，ArrayList 类就自动扩展了，新的 elementData 数组长度是 $(10 \times 3)/2 + 1$ ，也就是 16，当增加到第 17 个元素时再次扩容为 $(16 \times 3)/2 + 1$ ，也就是 25，依此类推，实现了 ArrayList 的动态数组管理。

从这里我们可以看出，如果不设置初始容量，系统就按照 1.5 倍的规则扩容，每次扩容都是一次数组的拷贝，如果数据量很大，这样的拷贝会非常耗费资源，而且效率非常低下。如果我们已经知道一个 ArrayList 的可能长度，然后对 ArrayList 设置一个初始容量则可以显著提高系统性能。比如一个班级的学生，通常也就是 50 人左右，我们就声明 ArrayList 的默认容量为 50 的 1.5 倍（元素数量小，直接计算，避免数组拷贝），即 new ArrayList<Student>(75)，这样在使用 add 方法增加元素时，只要在 75 以内都不用做数组拷贝，超过了 75 才会按照默认规则扩容（也就是 1.5 倍扩容）。如此处理，对我们的开发逻辑并不会有任何影响，而且还可以提高运行效率（在大数据量下，是否指定容量会使性能相差 5 倍以上）。

弄明白了 ArrayList 的长度处理方式，那其他集合类型呢？我们先来看 Vector，它的处理方式与 ArrayList 相似，只是数组的长度计算方式不同而已，代码如下：

```

private void ensureCapacityHelper(int minCapacity) {
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object[] oldData = elementData;
        // 若有递增步长，则按步长增长；否则，扩容 2 倍
        int newCapacity = (capacityIncrement > 0) ? (oldCapacity + capacityIncrement)
            : (oldCapacity * 2);
        // 越界检查，否则超过 int 最大值
        if (newCapacity < minCapacity) {
            newCapacity = minCapacity;
        }
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

```

Vector 与 ArrayList 不同的地方是它提供了递增步长（capacityIncrement 变量），其值代

表的是每次数组拓长时要增加的长度，不设置此值则是容量翻倍（默认是不设置递增步长的，可以通过构造函数来设置递增步长）。其他集合类的扩容方式与此相似，如 HashMap 是按照倍数增加的，Stack 继承自 Vector，所采用的也是与其相同的扩容原则等，读者有兴趣可以自行研读一下 JDK 的源码。

注意 非常有必要在集合初始化时声明容量。

建议 64：多种最值算法，适时选择

对一批数据进行排序，然后找出其中的最大值或最小值，这是基本的数据结构知识。在 Java 中我们可以通过编写算法的方式，也可以通过数组先排序再取值的方式来实现。下面以求最大值为例，解释一下多种算法。

(1) 自行实现，快速查找最大值

先来看用快速查找法取最大值的算法，其代码如下：

```
public static int max(int[] data) {
    int max = data[0];
    for (int i: data) {
        max = max > i ? max : i;
    }
    return max;
}
```

这是我们经常使用的最大值算法，也是速度最快的算法。它不要求排序，只要遍历一遍数组即可找出最大值。

(2) 先排序，后取值

对于求最大值，也可以采用先排序后取值的方式，同样比较简单，代码如下：

```
public static int max(int[] data) {
    // 先排序
    Arrays.sort(data.clone());
    // 然后取值
    return data[data.length - 1];
}
```

从效率上来讲，当然是自己写快速查找法更快一些了，只用遍历一遍就可以计算出最大值。但在实际测试中我们发现，如果数组数量少于 1 万，两者基本上没有差别，在同一个毫秒级别里，此时就可以不用自己写算法了，直接使用数组先排序后取值的方式。

如果数组元素超过 1 万，就需要依据实际情况来考虑：自己实现，可以提升性能；先排序后取值，简单，通俗易懂。排除性能上的差异，两者都可以选择，甚至后者更方便一些，也更容易想到。

现在问题来了，在代码中为什么要先使用 `data.clone()` 拷贝再排序呢？那是因为数组也是一个对象，不拷贝不就改变了原有数组元素的顺序吗？除非数组元素的顺序无关紧要。

接着往下思考，如果要查找仅次于最大值的元素（也就是老二），该如何处理呢？要注意，数组的元素是可以重复的，最大值可能是多个，所以单单一个排序然后取倒数第二个元素是解决不了问题的。

此时，就需要一个特殊的排序算法了，先要剔除重复数据，然后再排序。当然，自己写算法也可以实现，但是集合类已经提供了非常好的方法，要是再使用数组自己写算法就显得有点过时了。数组不能剔除重复数据，但 `Set` 集合却是可以的，而且 `Set` 的子类 `TreeSet` 还能自动排序。代码如下：

```
public static int getSecond(Integer[] data) {
    // 转换为列表
    List<Integer> dataList = Arrays.asList(data);
    // 转换为 TreeSet，删除重复元素并升序排列
    TreeSet<Integer> ts = new TreeSet<Integer>(dataList);
    // 取得比最大值小的最大值，也就是老二了
    return ts.lower(ts.last());
}
```

剔除重复元素并升序排列，这都由 `TreeSet` 类实现的，然后可再使用 `lower` 方法寻找小于最大值的值。大家看，上面的程序非常简单吧？那如果是自己编写代码会怎么样？那至少要遍历数组两遍才能计算出老二的值，代码的复杂度将大大提升。

也许你会说，这个要求有点变态，怎么会有这样的需求？不，有这样的需求很正常，比如在学校按成绩排名时，如果一个年级有 1200 人，只要找出最高的三个分数（可不一定就是 3 个人，也可能更多人），是不是就是这种情况呢？因此在实际应用中求最值，包括最大值、最小值、第二大值、倒数第二小值等，使用集合是最简单的方式，当然若从性能方面来考虑，数组是最好的选择。

注意 最值计算时使用集合最简单，使用数组性能最优。

建议 65：避开基本类型数组转换列表陷阱

我们在开发过程中经常会使用 `Arrays` 和 `Collections` 这两个工具类在数组和列表之间转换，非常方便，但也有时候会出现一些奇怪的问题，来看如下代码：

```
public static void main(String[] args) {
    int[] data = {1,2,3,4,5};
    List list = Arrays.asList(data);
    System.out.println("列表中的元素数量是：" + list.size());
}
```

也许你会说，这很简单，list 变量的元素数量当然是 5 了。但是运行后打印出来的列表数量却是 1。

事实上 data 确实是一个有 5 个元素的 int 类型数组，只是通过 asList 转换成列表后就只有 1 个元素了，这是为什么呢？其他 4 个元素到什么地方去了呢？

我们仔细看一下 Arrays.asList 的方法说明：输入一个变长参数，返回一个固定长度的列表。注意这里是一个变长参数，看源代码：

```
public static <T> List<T> asList(T... a) {
    return new ArrayList<T>(a);
}
```

asList 方法输入的是一个泛型变长参数，我们知道基本类型是不能泛型化的，也就是说 8 个基本类型不能作为泛型参数，要想作为泛型参数就必须使用其所对应的包装类型。那前面的例子传递了一个 int 类型的数组，为什么程序没有报编译错呢？

在 Java 中，数组是一个对象，它是可以泛型化的，也就是说我们的例子是把一个 int 类型的数组作为 T 的类型，所以转换后在 List 中就只有一个类型为 int 数组的元素了，我们打印出来看看，代码如下：

```
public static void main(String[] args) {
    int[] data = {1,2,3,4,5};
    List list = Arrays.asList(data);
    System.out.println("元素类型：" + list.get(0).getClass());
    System.out.println("前后是否相等：" + data.equals(list.get(0)));
}
```

输出的结果是：

```
元素类型：class [I
前后是否相等：true
```

很明显，放在列表中的元素是一个 int 数组，可能有读者要问了，为什么“元素类型：”后的 class 是 “[I”？我们并没有指明是数组（Array）类型呀！这是因为 JVM 不可能输出 Array 类型，因为 Array 是属于 java.lang.reflect 包的，它是通过反射访问数组元素的工具类。在 Java 中任何一个数组的类都是 “[I”，究其原因就是 Java 并没有定义数组这一个类，它是在编译器编译的时候生成的，是一个特殊的类，在 JDK 的帮助中也没有任何数组类的信息。

弄清楚了问题，修改方案也就诞生了，直接使用包装类即可，代码如下：

```
public static void main(String[] args) {
    Integer[] data = {1,2,3,4,5};
    List list = Arrays.asList(data);
    System.out.println("列表中的元素数量是：" + list.size());
}
```

仅仅修改了加粗字体部分，把 int 替换为 Integer 即可让输出元素数量为 5。需要说明的

是，不仅仅是 int 类型的数组有这个问题，其他 7 个基本类型的数组也存在相似的问题，这就需要读者注意了，在把基本类型数组转换成列表时，要特别小心 asList 方法的陷阱，避免出现程序逻辑混乱的情况。

注意 原始类型数组不能作为 asList 的输入参数，否则会引起程序逻辑混乱。

建议 66：asList 方法产生的 List 对象不可更改

上一个建议指出了 asList 方法在转换基本类型数组时存在的问题，接着我们看一下 asList 方法返回的列表有何特殊的地方，代码如下所示：

```
enum Week{Sun,Mon, Tue, Wed,Thu,Fri,Sat}
public static void main(String[] args) {
    // 五天工作制
    Week[] workDays = {Week.Mon, Week.Tue, Week.Wed,Week.Thu,Week.Fri};
    // 转换为列表
    List<Week> list = Arrays.asList(workDays);
    // 增加周六也为工作日
    list.add(Week.Sat);
    /* 工作日开始干活了 */
}
```

很简单的程序呀，默认声明的工作日（workDays）是从周一到周五，偶尔周六也会算作工作日加入到工作日列表中。不过，这段程序执行时会不会有什么问题呢？

编译没有任何问题，但是一运行，却出现了如下结果：

```
Exception in thread "main" java.lang.UnsupportedOperationException
at java.util.AbstractList.add(AbstractList.java:131)
at java.util.AbstractList.add(AbstractList.java:91)
```

UnsupportedOperationException，不支持的操作？居然不支持 List 的 add 方法，这真是奇怪了！还是来追根寻源，看看 asList 方法的源代码：

```
public static <T> List<T> asList(T... a) {
    return new ArrayList<T>(a);
}
```

直接 new 了一个 ArrayList 对象返回，难道 ArrayList 不支持 add 方法？不可能呀！可能，问题就出在这个 ArrayList 类上，此 ArrayList 非 java.util.ArrayList，而是 Arrays 工具类的一个内置类，其构造函数如下所示：

```
// 这是一个静态私有内部类
private static class ArrayList<E> extends AbstractList<E>
implements RandomAccess, java.io.Serializable {
```

```
// 存储列表元素的数组
private final E[] a;
// 唯一的构造函数
ArrayList(E[] array) {
    if (array==null)
        throw new NullPointerException();
    a = array;
}
/* 其他方法省略 */
}
```

这里的 ArrayList 是一个静态私有内部类，除了 Arrays 能访问外，其他类都不能访问。仔细看这个类，它没有提供 add 方法，那肯定是父类 AbstractList 提供了，来看代码：

```
public boolean add(E e) {
    throw new UnsupportedOperationException();
}
```

父类确实提供了，但没有提供具体的实现（源代码上是通过 add 方法调用 add(int,E) 方法来实现的，为了便于讲解，此处缩减了代码），所以每个子类都需要自己覆写 add 方法，而 Arrays 的内部类 ArrayList 没有覆写，因此 add 一个元素就会报错了。

我们再深入地看看这个 ArrayList 静态内部类，它仅仅实现了 5 个方法：

- size：元素数量。
- toArray：转化为数组，实现了数组的浅拷贝。
- get：获得指定元素。
- set：重置某一元素值。
- contains：是否包含某元素。

对于我们经常使用的 List.add 和 List.remove 方法它都没有实现，也就是说 asList 返回的是一个长度不可变的列表，数组是多长，转换成的列表也就是多长，换句话说此处的列表只是数组的一个外壳，不再保持列表动态变长的特性，这才是我们要关注的重点（虽然此处 JDK 的设计有悖 OO 设计原则，但这不在我们讨论的范围内，而且我们也无力回天）。

有些开发者特别喜欢通过如下方式定义和初始化列表：

```
List<String> names = Arrays.asList("张三", "李四", "王五");
```

一句话完成了列表的定义和初始化，看似很便捷，却深藏着重大隐患——列表长度无法修改。想想看，如果这样一个 List 传递到一个允许 add 操作的方法中，那将会产生何种结果？如果读者有这种习惯，请慎之戒之，除非非常自信该 List 只用于读操作。

建议 67：不同的列表选择不同的遍历方法

我们来思考这样一个案例：统计一个省的各科高考平均值，比如数学平均分是多少，语

文平均分是多少等，这是每年招生办都会公布的数据，我们来想想看该算法应如何实现。当然使用数据库中的一个 SQL 语句就能求出平均值，不过这不再我们的考虑之列，这里还是使用纯 Java 的算法来解决之，看代码：

```

public static void main(String[] args) {
    // 学生数量，80万
    int stuNum = 80*10000;
    //List 集合，记录所有学生的分数
    List<Integer> scores = new ArrayList<Integer>(stuNum);
    // 写入分数
    for(int i=0;i<stuNum;i++){
        scores.add(new Random().nextInt(150));
    }
    // 记录开始计算时间
    long start = System.currentTimeMillis();
    System.out.println("平均分是：" + average(scores));
    System.out.println("执行时间：" + (System.currentTimeMillis() - start) + "ms");
}
// 计算平均数
public static int average(List<Integer> list){
    int sum = 0;
    // 遍历求和
    for(int i:list){
        sum +=i;
    }
    // 除以人数，计算平均值
    return sum/list.size();
}

```

把 80 万名学生的成绩放到一个 ArrayList 数组中，然后通过 foreach 方式遍历求和，再计算平均值，程序非常简单，输出的结果是：

```

平均分是：74
执行时间：47ms

```

仅仅求一个算术平均值就花费了 47 毫秒，不要说考虑其他诸如加权平均值、补充平均值等算法，那花的时间肯定更长。我们仔细分析一下 average 方法，加号操作是最基本操作，没有什么可以优化的，剩下的就是一个遍历了，问题是 List 的遍历可以优化吗？

我们可以尝试一下，List 的遍历还有另外一种方式，即通过下标方式来访问，代码如下：

```

// 计算平均数
public static int average(List<Integer> list) {
    int sum = 0;
    // 遍历求和
    for (int i = 0, size = list.size(); i < size; i++) {
        sum += list.get(i);
    }
    // 除以人数，计算平均值

```

```

    return sum / list.size();
}

```

不再使用 `foreach` 方式遍历列表，而是采用下标方式遍历，我们看看输出结果如何：

```

平均分是: 74
执行时间: 16ms

```

执行时间已经大幅度下降，性能提升了 65%，这是一个飞速提升！那为什么我们使用下标方式遍历数组会有这么高的性能提升呢？

这是因为 `ArrayList` 数组实现了 `RandomAccess` 接口（随机存取接口），这也就标志着 `ArrayList` 是一个可以随机存取的列表。在 Java 中，`RandomAccess` 和 `Cloneable`、`Serializable` 一样，都是标志性接口，不需要任何实现，只是用来表明其实现类具有某种特质的，实现了 `Cloneable` 表明可以被拷贝，实现了 `Serializable` 接口表明被序列化了，实现了 `RandomAccess` 则表明这个类可以随机存取，对我们的 `ArrayList` 来说也就标志着其数据元素之间没有关联，即两个位置相邻的元素之间没有相互依赖和索引关系，可以随机访问和存储。

我们知道，Java 中的 `foreach` 语法是 `iterator`（迭代器）的变形用法，也就是说上面的 `foreach` 与下面的代码等价：

```

for(Iterator<Integer> i=list.iterator(); i.hasNext(); ){
    sum +=i.next();
}

```

那我们再想想什么是迭代器，迭代器是 23 个设计模式中的一种，“提供一种方法访问一个容器对象中的各个元素，同时又无须暴露该对象的内部细节”，也就是说对于 `ArrayList`，需要先创建一个迭代器容器，然后屏蔽内部遍历细节，对外提供 `hasNext`、`next` 等方法。问题是 `ArrayList` 实现了 `RandomAccess` 接口，已表明元素之间本来没有关系，可是，为了使用迭代器就需要强制建立一种互相“知晓”的关系，比如上一个元素可以判断是否有下一个元素，以及下一个元素是什么等关系，这也就是通过 `foreach` 遍历耗时的原因。

Java 为 `ArrayList` 类加上了 `RandomAccess` 接口，就是在告诉我们，“嘿，`ArrayList` 是随机存取的，采用下标方式遍历列表速度会更快”，接着又有一个问题了：为什么不把 `RandomAccess` 加到所有的 `List` 实现类上呢？

那是因为有些 `List` 实现类不是随机存取的，而是有序存取的，比如 `LinkedList` 类，`LinkedList` 也是一个列表，但它实现了双向链表，每个数据结点中都有三个数据项：前节点的引用（`Previous Node`）、本节点元素（`Node Element`）、后继节点的引用（`Next Node`），这是数据结构的基本知识，不多讲了，也就是说在 `LinkedList` 中的两个元素本来就是有关联的，我知道你的存在，你也知道我的存在。那大家想想看，元素之间已经有关联关系了，使用 `foreach` 也就是迭代器方式是不是效率更高呢？我们修改一下例子，代码如下：

```

public static void main(String[] args) {
    // 学生数量，80万
    int stuNum = 80 * 10000;
    // List集合，记录所有学生的分数
    List<Integer> scores = new LinkedList<Integer>();
    /* 其他代码没有改变，不再赘述 */
}
public static int average(List<Integer> list) {
    int sum = 0;
    // foreach遍历求和
    for (int i : list) {
        sum += i;
    }
    // 除以人数，计算平均值
    return sum / list.size();
}

```

运行的结果如下：

```

平均分是：74
执行时间：16ms

```

确实如此，也是 16 毫秒，效率非常高。可能大家还想要测试一下下标方式（也就是采用 get 方法访问元素）遍历 LinkedList 元素的情况，其实不用测试，效率真的非常低，我们直接看源码：

```

public E get(int index) {
    return entry(index).element;
}

```

由 entry 方法查找指定下标的节点，然后返回其包含的元素，看 entry 方法：

```

private Entry<E> entry(int index) {
    /* 检查下标是否越界，代码不再拷贝 */
    Entry<E> e = header;
    if (index < (size >> 1)) {
        // 如果下标小于中间值，则从头节点开始搜索
        for (int i = 0; i <= index; i++)
            e = e.next;
    } else {
        // 如果下标大于等于中间值，则从尾节点反向遍历
        for (int i = size; i > index; i--)
            e = e.previous;
    }
    return e;
}

```

看懂了吗？程序会先判断输入的下标与中间值（size 右移一位，也就是除以 2 了）的关系，小于中间值则从头开始正向搜索，大于中间值则从尾节点反向搜索，想想看，每一次的

get 方法都是一个遍历，“性能”两字从何说起呢！

明白了随机存取列表和有序存取列表的区别，我们的 average 方法就必须重构了，以便实现不同的列表采用不同的遍历方式，代码如下：

```
public static int average(List<Integer> list) {
    int sum = 0;
    if (list instanceof RandomAccess) {
        // 可以随机存取，则使用下标遍历
        for (int i = 0, size = list.size(); i < size; i++) {
            sum += list.get(i);
        }
    } else {
        // 有序存取，使用 foreach 方式
        for (int i : list) {
            sum += i;
        }
    }
    // 除以人数，计算平均值
    return sum / list.size();
}
```

如此一来，列表的遍历就可以“以不变应万变”了，无论是随机存取列表还是有序列表，它都可以提供快速的遍历。

注意 列表遍历不是那么简单的，其中很有“学问”，适时选择最优的遍历方式，不要固化为一种。

建议 68：频繁插入和删除时使用 LinkedList

上一个建议介绍了列表的遍历方式，也就是“读”操作，本建议将介绍列表的“写”操作：即插入、删除、修改动作。

(1) 插入元素

列表中我们使用最多的是 ArrayList，下面来看看它的插入（add 方法）算法，源代码如下：

```
public void add(int index, E element) {
    /* 检查下标是否越界，代码不再拷贝 */
    // 若需要扩容，则增大底层数组的长度
    ensureCapacity(size+1);
    // 给 index 下标之后的元素（包括当前元素）的下标加 1，空出 index 位置
    System.arraycopy(elementData, index, elementData, index + 1,
size - index);
    // 赋值 index 位置元素
    elementData[index] = element;
    // 列表长度 +1
```

```

    size++;
}
}

```

注意看 `arraycopy` 方法，只要是插入一个元素，其后的元素就会向后移动一位，虽然 `arraycopy` 是一个本地方法，效率非常高，但频繁的插入，每次后面的元素都要拷贝一遍，效率就变低了，特别是在头位置插入元素时。现在的问题是，开发中确实会遇到要插入元素的情况，那有什么更好的方法解决此效率问题吗？

有，使用 `LinkedList` 类即可。我们知道 `LinkedList` 是一个双向链表，它的插入只是修改相邻元素的 `next` 和 `previous` 引用，其插入算法（`add` 方法）如下：

```

public void add(int index, E element) {
    addBefore(element, (index==size ? header : entry(index)));
}
}

```

这里调用了私有 `addBefore` 方法，该方法实现了在一个元素之前插入元素的算法，代码如下：

```

private Entry<E> addBefore(E e, Entry<E> entry) {
    // 组装一个新节点，previous 指向原节点的前节点，next 指向原节点
    Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);
    // 前节点的 next 指向自己
    newEntry.previous.next = newEntry;
    // 后节点的 previous 指向自己
    newEntry.next.previous = newEntry;
    // 长度 +1
    size++;
    // 修改计数器 +1
    modCount++;
    return newEntry;
}
}

```

这是一个典型的双向链表插入算法，把自己插入到链表，然后再把前节点的 `next` 和后节点的 `previous` 指向自己。想想看，这样一个插入元素（也就是 `Entry` 对象）的过程中，没有任何元素会有拷贝过程，只是引用地址改变了，那效率当然就高了。

经过实际测试得知，`LinkedList` 的插入效率比 `ArrayList` 快 50 倍以上。

(2) 删除元素

插入了解清楚了，我们再来看删除动作。`ArrayList` 提供了删除指定位置上的元素、删除指定值元素、删除一个下标范围内的元素集等删除动作，三者的实现原理基本相似，都是找到索引位置，然后删除。我们以最常用的删除指定下标的方法（`remove` 方法）为例来看看删除动作的性能到底如何，源码如下：

```

public E remove(int index) {
    // 下标校验
    RangeCheck(index);
}
}

```

```

// 修改计数器 +1
modCount++;
// 记录要删除的元素值
E oldValue = (E) elementData[index];
// 有多少个元素向前移动
int numMoved = size - index - 1;
if (numMoved > 0)
    // index 后的元素向前移动一位
    System.arraycopy(elementData, index+1, elementData, index, numMoved);
// 列表长度减 1，并且最后一位设为 null
elementData[--size] = null;
// 返回删除的值
return oldValue;
}

```

注意看，index 位置后的元素都向前移动了一位，最后一个位置空出来了，这又是一次数组拷贝，和插入一样，如果数据量大，删除动作必然会暴露出性能和效率方面的问题。ArrayList 其他的两个删除方法与此相似，不再赘述。

我们再来看看 LinkedList 的删除动作。LinkedList 提供了非常多的删除操作，比如删除指定位置元素、删除头元素等，与之相关的 poll 方法也会执行删除动作，下面来看最基本的删除指定位置元素的方法 remove，源代码如下：

```

private E remove(Entry<E> e) {
    // 取得原始值
    E result = e.element;
    // 前节点 next 指向当前节点的 next
    e.previous.next = e.next;
    // 后节点的 previous 指向当前节点的 previous
    e.next.previous = e.previous;
    // 置空当前节点的 next 和 previous
    e.next = e.previous = null;
    // 当前元素置空
    e.element = null;
    // 列表长度减 1
    size--;
    // 修改计数器 +1
    modCount++;
    return result;
}

```

这也是双向链表的标准删除算法，没有任何耗时的操作，全部是引用指针的变更，效率自然高了。

在实际测试中得知，处理大批量的删除动作，LinkedList 比 ArrayList 快 40 倍以上。

(3) 修改元素

写操作还有一个动作：修改元素值，在这一点上 LinkedList 输给了 ArrayList，这是因为 LinkedList 是顺序存取的，因此定位元素必然是一个遍历过程，效率大打折扣，我们来看 set

方法的代码：

```
public E set(int index, E element) {
    // 定位节点
    Entry<E> e = entry(index);
    E oldVal = e.element;
    // 节点的元素替换
    e.element = element;
    return oldVal;
}
```

看似很简洁，但是这里使用了 entry 方法定位元素，在上一个建议中我们已经说明了 LinkedList 这种顺序存取列表的元素定位方式会折半遍历，这是一个极耗时的操作。而 ArrayList 的修改动作则是数组元素的直接替换，简单高效。

在修改动作上，LinkedList 比 ArrayList 慢很多，特别是要进行大量的修改时，两者完全不在一个数量级上。

上面通过分析源码完成了 LinkedList 与 ArrayList 之间的 PK，其中 LinkedList 胜两局：删除和插入效率高；ArrayList 胜一局：修改元素效率高。总体上来说，在“写”方面，LinkedList 占优势，而且在实际使用中，修改是一个比较少的动作。因此，如果有大量的写操作（更多的是插入和删除动作），推荐使用 LinkedList。不过何为少量，何为大量呢？

这就要依赖诸位正在开发的系统了，一个实时交易的系统，即使写作操再少，使用 LinkedList 也比 ArrayList 合适，因为此类系统是争分夺秒的，多 N 个毫秒可能就会造成交易数据不准确；而对于一个批量系统来说，几十毫秒、几百毫秒，甚至是几千毫秒的差别意义都不大，这时是使用 LinkedList 还是 ArrayList 就看个人爱好了，当然，如果系统已经处于性能临界点了那就必须使用 LinkedList。

且慢，“写”操作还有一个增加（add 方法）操作，为什么这里没有 PK 呢？那是因为两者在增加元素时性能上基本没有什么差别，区别只是在增加时 LinkedList 生成了一个 Entry 元素，其 previous 指向倒数第二个 Entry，next 置空；而 ArrayList 则是把元素追加到了数组中而已，两者的性能差别非常微小，不再讨论。

建议 69：列表相等只需关心元素数据

我们来看一个判断列表相等的例子，代码如下：

```
public static void main(String[] args) {
    ArrayList<String> strs = new ArrayList<String>();
    strs.add("A");

    Vector<String> strs2 = new Vector<String>();
    strs2.add("A");
```

```

        System.out.println(strs.equals(strs2));
    }
}

```

两个类都不相同，一个是 ArrayList，一个是 Vector，那结果肯定不相等了！真是这样吗？其实结果是两者相等！

我们来详细分析一下为什么两者是相等的。两者都是列表（List），都实现了 List 接口，也都继承了 AbstractList 抽象类，其 equals 方法是在 AbstractList 中定义的，我们来看源代码：

```

public boolean equals(Object o) {
    if (o == this)
        return true;
    // 是否是 List 列表，注意这里：只要实现 list 接口即可
    if (!(o instanceof List))
        return false;
    // 通过迭代器访问 list 的所有元素
    ListIterator<E> e1 = listIterator();
    ListIterator e2 = ((List) o).listIterator();
    // 遍历两个 list 的元素
    while(e1.hasNext() && e2.hasNext()) {
        E o1 = e1.next();
        Object o2 = e2.next();
        // 只要存在着不相等就退出
        if (!(o1==null ? o2==null : o1.equals(o2)))
            return false;
    }
    // 长度是否也相等
    return !(e1.hasNext() || e2.hasNext());
}

```

看到没？这里只是要求实现了 List 接口就成，它不关心 List 的具体实现类。只要所有的元素相等，并且长度也相等就表明两个 List 是相等的，与具体的容量类型无关。也就是说，上面的例子中虽然一个是 ArrayList，一个是 Vector，只要里面的元素相等，那结果就是相等。

Java 如此处理也确实在为开发者考虑，列表只是一个容器，只要是同一种类型的容器（如 List），不用关心容器的细节差别（如 ArrayList 与 LinkedList），只要确定所有的元素数据相等，那这两个列表就是相等的。如此一来，我们在开发中就不用太关注容器细节了，可以把注意力更多地放在数据元素上，而且即使在中途重构容器类型，也不会对相等的判断产生太大的影响。

其他的集合类型，如 Set、Map 等与此相同，也是只关心集合元素，不用考虑集合类型。

注意 判断集合是否相等时只须关注元素是否相等即可。

建议 70：子列表只是原列表的一个视图

List 接口提供了 subList 方法，其作用是返回一个列表的子列表，这与 String 类的 subString 有点类似，但它们的功能是否相同呢？我们来看如下代码：

```
public static void main(String[] args) {
    // 定义一个包含两个字符串的列表
    List<String> c = new ArrayList<String>();
    c.add("A");
    c.add("B");
    // 构造一个包含 c 列表的字符串列表
    List<String> c1 = new ArrayList<String>(c);
    //subList 生成与 c 相同的列表
    List<String> c2 = c.subList(0, c.size());
    //c2 增加一个元素
    c2.add("C");
    System.out.println("c == c1? " + c.equals(c1));
    System.out.println("c == c2? " + c.equals(c2));
}
```

c1 是通过 ArrayList 的构造函数创建的，c2 是通过列表的 subList 方法创建的，然后 c2 又增加了一个元素 C，现在的问题是输出的结果是什么呢？列表 c 与 c1、c2 之间是什么关系呢？

别忙着回答这个问题，我们先来回想一下 String 类的 subString 方法，看看它是如何工作的，代码如下：

```
public static void main(String[] args) {
    String str = "AB";
    String str1 = new String(str);
    String str2 = str.substring(0) + "C";
    System.out.println("str == str1? " + str1.equals(str1));
    System.out.println("str == str2? " + str.equals(str2));
}
```

很明显，str 与 str1 是相等的（虽然不是同一个对象，但用 equals 方法判断是相等的），但它们与 str2 不相等，这毋庸置疑，因为 str2 在对象池中重新生成了一个新的对象，其表面值是 ABC，那当然与 str 和 str1 不相等了。

说完了 subString 的小插曲，现在回到 List 是否相等的判断上来。subList 与 subString 的输出结果是一样的吗？让事实说话，运行结果如下：

```
c == c1? false
c == c2? true
```

很遗憾，与 String 类刚好相反，同样是一个 sub 类型的操作，为什么会相反呢？仅仅回答“为什么”似不足以平复我们的惊讶，下面就从最底层的源代码来进行分析。

c2 是通过 subList 方法从 c 列表中生成的一个子列表，然后 c2 又增加了一个元素，可为

什么增加了一个元素还会相等呢？我们来看 subList 源码：

```
public List<E> subList(int fromIndex, int toIndex) {
    return (this instanceof RandomAccess ?
            new RandomAccessSubList<E>(this, fromIndex, toIndex) :
            new SubList<E>(this, fromIndex, toIndex));
}
```

subList 方法是由 AbstractList 实现的，它会根据是不是可以随机存取来提供不同的 SubList 实现方式，不过，随机存储的使用频率比较高，而且 RandomAccessSubList 也是 SubList 子类，所以所有的操作都是由 SubList 类实现的（除了自身的 SubList 方法外），那么，我们就直接来看 SubList 类的代码：

```
class SubList<E> extends AbstractList<E> {
    // 原始列表
    private AbstractList<E> l;
    // 偏移量
    private int offset;
    // 构造函数，注意 list 参数就是我们的原始列表
    SubList(AbstractList<E> list, int fromIndex, int toIndex) {
        /* 下标校验，省略 */
        // 传递原始列表
        l = list;
        offset = fromIndex;
        // 子列表的长度
        size = toIndex - fromIndex;
    }
    // 获得指定位置的元素
    public E get(int index) {
        /* 校验部分，省略 */
        // 从原始字符串中获得指定位置的元素
        return l.get(index+offset);
    }
    // 增加或插入
    public void add(int index, E element) {
        /* 校验部分，省略 */
        // 直接增加到原始字符串上
        l.add(index+offset, element);
        /* 处理长度和修改计数器 */
    }
    /* 其他方法省略 */
}
```

通过阅读这段代码，我们就非常清楚 subList 方法的实现原理了：它返回的 SubList 类也是 AbstractList 的子类，其所有的方法如 get、set、add、remove 等都是在原始列表上的操作，它自身并没有生成一个数组或是链表，也就是子列表只是原列表的一个视图（View），所有的修改动作都反映在了原列表上。

我们例子中的 c2 增加了一个元素 C，不过增加的元素 C 到了 c 列表上，两个变量的元素仍保持完全一致，相等也就很自然了。

解释完相等的问题，再回过头来看看为什么变量 c 与 c1 不相等。很简单，因为通过 ArrayList 构造函数创建的 List 对象 c1 实际上是新列表，它是通过数组的 copyOf 动作生成的，所生成的列表 c1 与原列表 c 之间没有任何关系（虽然是浅拷贝，但元素类型是 String，也就是说元素是深拷贝的），然后 c 又增加了元素，因为 c1 与 c 之间已经没有一毛钱的关系了，那自然是不相等了。

注意 subList 产生的列表只是一个视图，所有的修改动作直接作用于原列表。

建议 71：推荐使用 subList 处理局部列表

我们来看这样一个简单的需求：一个列表有 100 个元素，现在要删除索引位置为 20~30 的元素。这很简单，一个遍历很快就可以完成，代码如下：

```
public static void main(String[] args) {
    // 初始化一个固定长度，不可变列表
    List<Integer> initData= Collections.nCopies(100, 0);
    // 转换为可变列表
    List<Integer> list = new ArrayList<Integer>(initData);
    // 遍历，删除符合条件的元素
    for(int i=0,size=list.size();i<size;i++){
        if(i>=20 && i<30){
            list.remove(i);
        }
    }
}
```

或者

```
for (int i = 20; i < 30; i++) {
    if(i<list.size()){
        list.remove(i);
    }
}
```

相信首先出现在大家脑海中的实现就是此算法了，遍历一遍，符合条件的就删除，简单而又实用。不过，还有没有其他方式呢？有没有“one-lining”一行代码就解决问题的方式呢？

有，直接使用 ArrayList 的 removeRange 方法不就可以了么？等等，好像不可能呀，虽然 JDK 上有此方法，但是它有 protected 关键字修饰着，不能直接使用，那怎么办？看看如下代码。

```

public static void main(String[] args) {
    // 初始化一个固定长度，不可变列表
    List<Integer> initData = Collections.nCopies(100, 0);
    // 转换为可变列表
    ArrayList<Integer> list = new ArrayList<Integer>(initData);
    // 删除指定范围的元素
    list.subList(20, 30).clear();
}

```

上一个建议讲解了 `subList` 方法的具体实现方式，所有的操作都是在原始列表上进行的，那我们就用 `subList` 先取出一个子列表，然后清空。因为 `subList` 返回的 `List` 是原始列表的一个视图，删除这个视图中的所有元素，最终就会反映到原始字符串上，那么一行代码即解决问题了。

建议 72：生成子列表后不要再操作原列表

前面说了，`subList` 生成的子列表是原列表的一个视图，那在 `subList` 执行完后，如果修改了原列表的内容会怎样呢？视图是否会改变呢？如果是数据库视图，表数据变更了，视图当然会变了，至于 `subList` 生成的视图是否会改变，还是从源码上来看吧，代码如下：

```

public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add("A");
    list.add("B");
    list.add("C");

    List<String> subList = list.subList(0, 2);
    // 原字符串增加一个元素
    list.add("D");
    System.out.println("原列表长度: " + list.size());
    System.out.println("子列表长度: " + subList.size());
}

```

程序中有一个原始列表，生成了一个子列表，然后在原始列表中增加一个元素，最后打印出原始列表和子列表的长度，大家想一下，这段程序什么地方会出现错误呢？

`list.add("D")` 会报错吗？不会，`subList` 并没有锁定原列表，原列表当然可以继续修改。

难道有两个 `size` 方法？正确，确实是 `size` 方法出错了，输出结果如下：

```

原列表长度: 4
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.SubList.checkForComodification(AbstractList.java:752)
at java.util.SubList.size(AbstractList.java:625)

```

什么？居然是 `subList` 的 `size` 方法出现了异常，而且还是并发修改异常？这没道理呀，

这里根本就没有多线程操作，何来并发修改呢？这个问题很容易回答，那是因为 `subList` 取出的列表是原列表的一个视图，原数据集（代码中的 `list` 变量）修改了，但是 `subList` 取出的子列表不会重新生成一个新列表（这点与数据库视图是不相同的），后面在对子列表继续操作时，就会检测到修改计数器与预期的不相同，于是就抛出了并发修改异常。

出现这个问题的最终原因还是在子列表提供的 `size` 方法的检查上，还记得上面几个例子中经常提到的修改计数器吗？原因就在这里，我们来看看 `size` 的源代码：

```
public int size() {
    checkForComodification();
    return size;
}
```

其中的 `checkForComodification` 方法就是用于检测是否并发修改的，代码如下：

```
private void checkForComodification() {
    // 判断当前修改计数器是否与子列表生成时一致
    if (l.modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

`expectedModCount` 是从什么地方来的呢？它是在 `SubList` 子列表的构造函数中赋值的，其值等于生成子列表时的修改次数（`modCount` 变量）。因此在生成子列表后再修改原始列表，`l.modCount` 的值就必然比 `expectedModCount` 大 1，不再保持相等了，于是也就抛出了 `ConcurrentModificationException` 异常。

`subList` 的其他方法也会检测修改计数器，例如 `set`、`get`、`add` 等方法，若生成子列表后，再修改原列表，这些方法也会抛出 `ConcurrentModificationException` 异常。

对于子列表操作，因为视图是动态生成的，生成子列表后再操作原列表，必然会导致“视图”的不稳定，最有效的办法就是通过 `Collections.unmodifiableList` 方法设置列表为只读状态，代码如下：

```
public static void main(String[] args) {
    List<String> list = new ArrayList<String>();

    List<String> subList = list.subList(0, 2);
    // 设置列表为只读状态
    list = Collections.unmodifiableList(list);
    // 对 list 进行只读操作
    doReadSomething(list)
    // 对 subList 进行读写操作
    doReadAndWriteSomething(subList)
}
```

这在团队编码中特别有用，比如我生成了一个 `List`，需要调用其他同事写的共享方法，但是有一些元素是不能修改的，想想看，此时 `subList` 方法和 `unmodifiableList` 配合着使用是

不是就可以解决我们的问题了呢？防御式编程就是教我们如此做的。

这里还有一个问题，数据库的一张表可以有很多视图，我们的 List 也可以有多个视图，也就是可以有多个子列表，但问题是只要生成的子列表多于一个，则任何一个子列表就都不能修改了，否则就会抛出 ConcurrentModificationException 异常。

注意 subList 生成子列表后，保持原列表的只读状态。

建议 73：使用 Comparator 进行排序

在项目开发中，我们经常要对一组数据进行排序，或者升序或者降序，在 Java 中排序有多种方式，最土的方法就是自己写排序算法，比如冒泡排序、快速排序、二叉树排序等，但一般不要自己写，JDK 已经为我们提供了很多的排序算法，我们采用“拿来主义”就成了。

在 Java 中，要想给数据排序，有两种实现方式，一种是实现 Comparable 接口，一种是实现 Comparator 接口，这两者有什么区别呢？我们来看一个身边的例子，就比如给公司职员排序吧，最经常使用的是按照工号排序，先定义一个职员类代码，如下所示：

```
class Employee implements Comparable<Employee> {
    //id 是根据进入公司的先后顺序编码的
    private int id;
    // 姓名
    private String name;
    // 职位
    private Position position;

    public Employee(int _id, String _name, Position _position) {
        id = _id;
        name = _name;
        position = _position;
    }
    /*id、name、position 的 getter/setter 方法省略 */
    // 按照 id 号排序，也就是资历的深浅排序
    @Override
    public int compareTo(Employee o) {
        return new CompareToBuilder()
            .append(id, o.id).toComparison();
    }

    @Override
    public String toString(){
        return ToStringBuilder.reflectionToString(this);
    }
}
```

这是一个简单的 JavaBean，描述的是一个员工的基本信息，其中 id 号是员工编号，按

照进入公司的先后顺序编码，position 是岗位描述，表示是经理还是普通职员，这是一个枚举类型，代码如下：

```
enum Position {
    Boss, Manager, Staff
}
```

职位有三个级别：Boss（老板），Manager（经理），Staff（普通职员）。

注意 Employee 类中的 compareTo 方法，它是 Comparable 接口要求必须实现的方法，这里使用 apache 的工具类来实现，表明是按照 id 的自然序列排序的（也就是升序）。一切准备完毕，我们看看如何排序：

```
public static void main(String[] args) {
    List<Employee> list = new ArrayList<Employee>(5);
    // 一个老板
    list.add(new Employee(1001, "张三", Position.Boss));
    // 两个经理
    list.add(new Employee(1006, "赵七", Position.Manager));
    list.add(new Employee(1003, "王五", Position.Manager));
    // 两个职员
    list.add(new Employee(1002, "李四", Position.Staff));
    list.add(new Employee(1005, "马六", Position.Staff));
    // 按照 id 排序，也就是按照资历深浅排序
    Collections.sort(list);
    for(Employee e:list){
        System.out.println(e);
    }
}
```

在收集数据时按照职位高低来收集，这也是“为领导服务”理念的体现嘛，先登记领导，然后是小领导，最后是普通员工。排序后的输出如下：

```
Employee@1037c71[id=1001,name=张三,position=Boss]
Employee@b1c5fa[id=1002,name=李四,position=Staff]
Employee@f84386[id=1003,name=王五,position=Manager]
Employee@15d56d5[id=1005,name=马六,position=Staff]
Employee@efd552[id=1006,name=赵七,position=Manager]
```

是按照 id 号升序排列的，结果正确，但是，有时候我们希望按照职位来排序，那怎么做呢？此时，重构 Employee 类已经不合适了，Employee 已经是一个稳定类，为了一个排序功能修改它不是一个好办法，那有什么更好的解决办法吗？

有办法，看 Collections.sort 方法，它有一个重载方法 Collections.sort(List<T> list, Comparator<? super T> c)，可以接受一个 Comparator 实现类，这下就好办了，代码如下：

```
class PositionComparator implements Comparator<Employee>{
    @Override
```

```

public int compare(Employee o1, Employee o2) {
    // 按照职位降序排列
    return o1.getPosition().compareTo(o2.getPosition());
}
}

```

创建了一个职位排序法，依据职位的高低进行降序排列，然后只要把 Collections.sort(list) 修改为 Collections.sort(list, new PositionComparator()) 即可实现按职位排序的要求。

现在问题又来了：按职位临时倒序排列呢？注意只是临时的，是否要重写一个排序器？完全不用，有两个解决办法：

- 直接使用 Collections.reverse(List<?> list) 方法实现倒序排列。
- 通过 Collections.sort(list, Collections.reverseOrder(new PositionComparator())) 也可以实现倒序排列。

第二个问题：先按照职位排序，职位相同再按照工号排序，这如何处理？这可是我们经常遇到的实际问题。很好处理，在 compareTo 或 compare 方法中先判断职位是否相等，相等的话再根据工号排序，使用 apache 的工具类来简化处理，代码如下：

```

public int compareTo(Employee o) {
    return new CompareToBuilder()
        .append(position, o.position) // 职位排序
        .append(id, o.id).toComparison(); // 工号排序
}

```

在 JDK 中，对 Collections.sort 方法的解释是按照自然顺序进行升序排列，这种说法其实是不太准确的，sort 方法的排序方式并不是一成不变的升序，也可能是倒序，这依赖于 compareTo 的返回值，我们知道如果 compareTo 返回负数，表明当前值比对比值小，零表示相等，正数表明当前值比对比值大，比如我们修改一下 Employee 的 compareTo 方法，如下所示：

```

public int compareTo(Employee o) {
    return new CompareToBuilder()
        .append(o.id, id).toComparison();
}

```

两个参数调换了一下位置，也就是 compareTo 的返回值与之前正好相反，再使用 Collections.sort 进行排序，顺序也就相反了，这样就实现了倒序。

第三个问题：在 Java 中，为什么要有两个排序接口呢？

很多同学都提出了这个问题，其实也好回答，实现了 Comparable 接口的类表明自身是可比较的，有了比较才能进行排序；而 Comparator 接口是一个工具类接口，它的名字（比较器）也已经表明了它的作用：用作比较，它与原有类的逻辑没有关系，只是实现两个类的比较逻辑，从这方面来说，一个类可以有很多的比较器，只要有业务需求就可以产生比较器，

有比较器就可以产生 N 多种排序，而 Comparable 接口的排序只能说是实现类的默认排序算法，一个类稳定、成熟后其 compareTo 方法基本不会改变，也就是说一个类只能有一个固定的、由 compareTo 方法提供的默认排序算法。

注意 Comparable 接口可以作为实现类的默认排序法，Comparator 接口则是一个类的扩展排序工具。

建议 74：不推荐使用 binarySearch 对列表进行检索

对一个列表进行检索时，我们使用得最多的是 indexOf 方法，它简单、好用，而且也不会出错，虽然它只能检索到第一个符合条件的值，但是我们可以生成子列表后再检索，这样也就能够查找出所有符合条件的值了。

Collections 工具类也提供一个检索方法：binarySearch，这个是干什么的？该方法也是对一个列表进行检索的，可查找出指定值的索引值，但是在使用这个方法时就有一些注意事项了，我们看如下代码：

```
public static void main(String[] args) {
    List<String> cities = new ArrayList<String>();
    cities.add("上海");
    cities.add("广州");
    cities.add("广州");
    cities.add("北京");
    cities.add("天津");
    //indexOf 方法取得索引值
    int index1 = cities.indexOf("广州");
    //binarySearch 查找到索引值
    int index2 = Collections.binarySearch(cities, "广州");
    System.out.println("索引值 (indexOf): "+index1);
    System.out.println("索引值 (binarySearch): "+index2);
}
```

先不考虑运行结果，直接看 JDK 上对 binarySearch 的描述：使用二分搜索法搜索指定列表，以获得指定对象。其实现的功能与 indexOf 是相同的，只是使用的是二分法搜索列表，所以估计两种方法返回结果是一样的，看结果：

```
索引值 (indexOf): 1
索引值 (binarySearch): 2
```

结果不一样，虽然说我们有两个“广州”这样的元素，但是返回的结果都应该是 1 才对呀，为何 binarySearch 返回的结果是 2 呢？问题就出在二分法搜索上，二分法搜索就是“折半折半再折半”的搜索方法，简单，而且效率高。看看 JDK 是如何实现的。

```

public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key) {
    if (list instanceof RandomAccess || list.size() < 5000)
        // 随机存取列表或者元素数量少于 5000 的顺序存取列表
        return Collections.indexedBinarySearch(list, key);
    else
        // 元素数量大于 5000 的顺序存取列表
        return Collections.iteratorBinarySearch(list, key);
}

```

ArrayList 实现了 RandomAccess 接口，是一个顺序存取列表，使用了 indexedBinarySearch 方法，代码如下：

```

private static <T> int indexedBinarySearch(List<? extends Comparable<? super T>>
    list, T key) {
    // 默认上界
    int low = 0;
    // 默认下界
    int high = list.size() - 1;
    while (low <= high) {
        // 中间索引，无符号右移 1 位
        int mid = (low + high) >>> 1;
        // 中间值
        Comparable<? super T> midVal = list.get(mid);
        // 比较中间值
        int cmp = midVal.compareTo(key);
        // 重置上界和下界
        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            // 找到元素
            return mid;
    }
    // 没有找到，返回负值
    return -(low + 1);
}

```

这也没啥说的，就是二分法搜索的 Java 版实现。注意看加粗字体部分，首先是获得中间索引值，我们的例子中也就是 2，那索引值是 2 的元素值是多少呢？正好是“广州”，于是返回索引值 2，正确，没问题。那我们再看看 indexOf 的实现，代码如下：

```

public int indexOf(Object o) {
    if (o == null) {
        // null 元素查找
        for (int i = 0; i < size; i++)
            if (elementData[i] == null)
                return i;
    } else {

```

```

// 非 null 元素查找
for (int i = 0; i < size; i++)
    // 两个元素是否相等, 注意这里是 equals 方法
    if (o.equals(elementData[i]))
        return i;
}
// 没找到, 返回 -1
return -1;
}

```

`indexOf` 方法就是一个遍历，找到第一个元素值相等则返回，没什么玄机。回到我们的程序上来看，`for` 循环的第二遍即是我们要查找的“广州”，于是就返回索引值 1 了，也正确，没有任何问题。

两者的算法都没有问题，难道是我们用错了？这还真是我们的错，因为二分法查询的一个首要前提是：数据集已经实现升序排列，否则二分法查找的值是不准确的。不排序怎么确定是在小区（比中间值小的区域）中查找还是在大区（比中间值大的区域）中查找呢？二分法查找必须要先排序，这是二分法查找的首要条件。

问题清楚了，藐视解决方法也很简单，使用 `Collections.sort` 排下序即可解决。但这样真的可以解决吗？想想看，元素数据是从 Web 或数据库中传递进来的，原本是一个有规则的业务数据，我们为了查找一个元素对它进行了排序，也就是改变了元素在列表中的位置，那谁来保证业务规则此时的正确性呢？所以说，`binarySearch` 方法在此处受限了。当然，拷贝一个数组，然后再排序，再使用 `binarySearch` 查找指定值，也可以解决该问题。

使用 `binarySearch` 首要考虑排序问题，这是我们编码人员经常容易忘记的，而且在测试期间还没发现问题（因为测试时“制造”的数据都很有规律），等到投入生产系统后才发现查找的数据不准确，又是一个大 Bug 产生了，从这点来看，`indexOf` 要比 `binarySearch` 简单得多。

当然，使用 `binarySearch` 的二分法查找比 `indexOf` 的遍历算法性能上高很多，特别是在大数据集而且目标值又接近尾部时，`binarySearch` 方法与 `indexOf` 相比，性能上会提升几十倍，因此在从性能的角度考虑时可以选择 `binarySearch`。

注意 从性能方面考虑，`binarySearch` 是最好的选择。

建议 75：集合中的元素必须做到 `compareTo` 和 `equals` 同步

实现了 `Comparable` 接口的元素就可以排序，`compareTo` 方法是 `Comparable` 接口要求必须实现的，它与 `equals` 方法有关系吗？有关系，在 `compareTo` 的返回为 0 时，它表示的是进行比较的两个元素是相等的。`equals` 是不是也应该对此作出相应的动作呢？我们看如下代码。

```

class City implements Comparable<City> {
    // 城市编码
    private String code;
    // 城市名称
    private String name;

    public City(String _code, String _name) {
        code = _code;
        name = _name;
    }
    /*code、name 的 getter/setter 方法省略 */
    @Override
    public int compareTo(City o) {
        // 按照城市名称排序
        return new CompareToBuilder()
            .append(name, o.name)
            .toComparison();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (obj == this) {
            return true;
        }
        if (obj.getClass() != getClass()) {
            return false;
        }
        City city = (City) obj;
        // 根据 code 判断是否相等
        return new EqualsBuilder()
            .append(code, city.code)
            .isEquals();
    }
}

```

与上一个建议类似，把多个城市对象放在一个 List 中，然后使用不同的方法查找同一个城市，看看返回值有什么异常。代码如下：

```

public static void main(String[] args) {
    List<City> cities = new ArrayList<City>();
    cities.add(new City("021", "上海"));
    cities.add(new City("021", "沪"));
    // 排序
    Collections.sort(cities);
    // 查找对象
    City city = new City("021", "沪");
    // indexOf 方法取得索引值

```

```

int index1 = cities.indexOf(city);
// binarySearch 查找到索引值
int index2 = Collections.binarySearch(cities, city);
System.out.println("索引值 (indexOf): " + index1);
System.out.println("索引值 (binarySearch): " + index2);
}

```

输出的 index1 和 index2 应该一致吧，都是从一个列表中查找相同的元素，只是使用的算法不同嘛。但是很遗憾，结果不一致：

```

索引值 (indexOf): 0
索引值 (binarySearch): 1

```

indexOf 返回的是第一个元素，而 binarySearch 返回的是第二个元素（索引值是 1），这是怎么回事呢？

这是因为 indexOf 是通过 equals 方法判断的，equals 等于 true 就认为找到符合条件的元素了，而 binarySearch 查找的依据是 compareTo 方法的返回值，返回 0 即认为找到符合条件的元素。

仔细审查一下代码，我们覆写了 compareTo 和 equals 方法，但是两者并不一致。使用 indexOf 方法查找时，遍历每个元素，然后比较 equals 方法的返回值，因为 equals 方法是根据 code 判断的，因此当第一次循环时，equals 就返回了 true，indexOf 方法结束，查找到指定值。而使用 binarySearch 二分法查找时，依据的是每个元素的 compareTo 方法返回值，而 compareTo 方法又是依赖 name 属性的，name 相等就返回 0，binarySearch 就认为找到元素了。

问题明白了，修改也就很容易了，将 equals 方法修改成判断 name 是否相等即可，虽然可以解决问题，但这是一个很无奈的解决办法，而且还要依赖我们的系统是否支持此类修改，因为相等逻辑已经发生了很大的变化。

从这个例子中，我们可以理解两点：

- indexOf 依赖 equals 方法查找，binarySearch 则依赖 compareTo 方法查找。
- equals 是判断元素是否相等，compareTo 是判断元素在排序中的位置是否相同。

既然一个是决定排序位置，一个是决定相等，那我们就应该保证当排序位置相同时，其 equals 也相同，否则就会产生逻辑混乱。

注意 实现了 compareTo 方法，就应该覆写 equals 方法，确保两者同步。

建议 76：集合运算时使用更优雅的方式

在初中代数中，我们经常会求两个集合的并集、交集、差集等，在 Java 中也存在着此类运算，那如何实现呢？一提到此类集合操作，大部分的实现者都会说：对两个集合进行遍

历，即可求出结果。是的，遍历可以实现并集、交集、差集等运算，但这不是最优雅的处理方式。下面来看看如何进行更优雅、快速、方便的集合操作。

(1) 并集

也叫做合集，把两个集合加起来即可，这非常简单，代码如下：

```
public static void main(String[] args) {
    List<String> list1 = new ArrayList<String>();
    list1.add("A");
    list1.add("B");
    List<String> list2 = new ArrayList<String>();
    list2.add("C");
    list2.add("B");

    // 并集
    list1.addAll(list2);
}
```

此时，list1 中就是两个列表的并集元素了。

(2) 交集

计算两个集合的共有元素，也就是你有我也有的元素集合，代码如下：

```
list1.retainAll(list2);
```

其中的变量 list1 和 list2 是两个列表，仅此一句话，list1 中就只包含 list1、list2 中共有的元素了。注意 retainAll 方法会删除 list1 中没有出现在 list2 中的元素。

(3) 差集

由所有属于 A 但不属于 B 的元素组成的集合，叫做 A 与 B 的差集，也就是我有你没有的元素，代码如下：

```
list1.removeAll(list2);
```

也很简单，从 list1 中删除出现在 list2 的元素，即可得出 list1 与 list2 的差集部分。

(4) 无重复的并集

什么叫无重复的并集？并集是集合 A 加集合 B，那如果集合 A 和集合 B 有交集（也就是并集的元素数量大于 0），就需要确保并集的结果中只有一份交集，此为无重复的并集。此操作也比较简单，代码如下：

```
// 删除在 list1 中出现的元素
list2.removeAll(list1);
// 把剩余的 list2 元素加到 list1 中
list1.addAll(list2);
```

有读者可能说了，求出两个集合的并集，然后转变成 HashSet 剔除重复元素不就解决问题了吗？错了，这样解决是不行的，比如集合 A 有 10 个元素（其中有两个元素值是相同

的), 集合 B 有 8 个元素, 它们的交集有 2 个元素, 我们可以计算出它们的并集是 18 个元素, 而无重复的并集有 16 个元素, 但是如果使用 HashSet 算法, 算出来则只有 15 个元素, 因为你把集合 A 中原本就重复的元素也剔除掉了。

读者可能会很困惑, 为什么要介绍并集、交集、差集呢? 那是因为只要去检查一下代码, 就会发现, 很少有程序员使用 JDK 提供的方法来实现这些集合操作, 基本上都是采用的标准的嵌套 for 循环: 要并集就是加法, 要交集了就使用 contains 判断是否存在, 要差集了就使用 !contains (不包含), 有时候还要为这类操作提供一个单独方法, 看似很规范, 但已经脱离了优雅的味道。

集合的这些操作在持久层中使用得非常频繁, 从数据库中取出的就是多个数据集合, 之后我们就可以使用集合的各种方法构建我们需要的数据了, 需要两个集合的 and 结果, 那是交集, 需要两个集合的 or 结果, 那是并集, 需要两个集合的 not 结果, 那是差集。

建议 77: 使用 shuffle 打乱列表

在网站上我们经常会看到关键字云 (Word Cloud) 和标签云 (Tag Cloud), 用于表明这个关键字或标签是经常被查阅的, 而且还可以看到这些标签的动态运动, 每次刷新都会有不一样的关键字或标签, 让浏览器觉得这个网站的访问量非常大, 短短的几分钟就有这么多的搜索量。不过, 这在 Java 中该如何实现呢? 代码如下:

```
public static void main(String[] args) {
    int tagCloudNum = 10;
    List<String> tagClouds = new ArrayList<String>(tagCloudNum);
    // 初始化标签云, 一般是从数据库读入, 省略
    Random rand = new Random();
    for(int i=0;i<tagCloudNum;i++){
        // 取得随机位置
        int randomPosition = rand.nextInt(tagCloudNum);
        // 当前元素与随机元素交换
        String temp = tagClouds.get(i);
        tagClouds.set(i, tagClouds.get(randomPosition));
        tagClouds.set(randomPosition, temp);
    }
}
```

先从数据库中读出标签, 然后使用随机数打乱, 每次都产生不同的顺序, 恩, 确实能让浏览器感觉到我们的标签云顺序在变化——浏览器多嘛! 但是, 对于乱序处理我们可以有更好的实现方式, 先来修改第一版:

```
public static void main(String[] args) {
    int tagCloudNum = 10;
    List<String> tagClouds = new ArrayList<String>(tagCloudNum);
```

```

Random rand = new Random();
for(int i=0;i<tagCloudNum;i++){
    // 取得随机位置
    int randomPosition = rand.nextInt(tagCloudNum);
    // 当前元素与随机元素交换
    Collections.swap(tagClouds, i, randomPosition);
}
}

```

上面使用了 Collections 的 swap 方法，该方法会交换两个位置的元素值，不用我们自己写交换代码了。难道乱序到此就优化完了吗？没有，我们可以继续重构，第二版重构如下：

```

public static void main(String[] args) {
    int tagCloudNum = 10;
    List<String> tagClouds = new ArrayList<String>(tagCloudNum);
    // 打乱顺序
    Collections.shuffle(tagClouds);
}

```

这才是我们想要的结果，就这一句话，即可打乱一个列表的顺序，不用我们费尽心思的遍历、替换元素了。我们一般很少用到 shuffle 这个方法，那它可以用在什么地方呢？

□ 可以用在程序的“伪装”上。

比如我们例子中的标签云，或者是游戏中的打怪、修行、群殴时宝物的分配策略。

□ 可以用在抽奖程序中。

比如年会的抽奖程序，先使用 shuffle 把员工排序打乱，每个员工的中奖几率就是相等的了，然后就可以抽取第一名、第二名。

□ 可以用在安全传输方面。

比如发送端发送一组数据，先随机打乱顺序，然后加密发送，接收端解密，然后自行排序，即可实现即使是相同的数据源，也会产生不同密文的效果，加强了数据的安全性。

建议 78：减少 HashMap 中元素的数量

在系统开发中，我们经常会使用 HashMap 作为数据集容器，或者是用缓冲池来处理，一般很稳定，但偶尔也会出现内存溢出的问题（如 OutOfMemory 错误），而且这经常是与 HashMap 有关的，比如我们使用缓冲池操作数据时，大批量的增删查改操作就可能会让内存溢出，下面建立一段模拟程序，重现该问题，代码如下：

```

public static void main(String[] args) {
    Map<String, String> map = new HashMap<String, String>();
    final Runtime rt = Runtime.getRuntime();
    // JVM 终止前记录内存信息
    rt.addShutdownHook(new Thread() {

```

```

@Override
public void run() {
    StringBuffer sb = new StringBuffer();
    long heapMaxSize = rt.maxMemory() >> 20;
    sb.append("最大可用内存: " + heapMaxSize + "M\n");
    long total = rt.totalMemory() >> 20;
    sb.append("对内存大小: " + total + "M\n");
    long free = rt.freeMemory() >> 20;
    sb.append("空闲内存: " + free + "M");
    System.out.println(sb);
}
});
// 放入近 40 万键值对
for (int i = 0; i < 393217; i++) {
    map.put("key" + i, "vlaue" + i);
}
}
}

```

该程序只是向 Map 中放入了近 40 万个键值对元素（不是整 40 万个，而是 393217 个，为什么呢？请继续往后看），只是增加，没有任何其他操作。想想看，会出现什么问题？内存溢出？运行结果如下所示：

```

Exception in thread "main" 最大可用内存: 63M
java.lang.OutOfMemoryError: Java heap space
at java.util.HashMap.resize(HashMap.java:462)
at java.util.HashMap.addEntry(HashMap.java:755)
at java.util.HashMap.put(HashMap.java:385)
at Client.main(Client.java:24)
对内存大小: 63M
空闲内存: 7M

```

内存溢出了！可能会有读者说，这很好解决，在运行时增加“-Xmx”参数设置内存大小即可。这确实可以，不过浮于表面了，没有真正从溢出的最根本原因上来解决问题。

难道是 String 字符串太多了？不对呀，字符串对象加起来撑死也就 10MB，而且这里还空闲了 7MB 内存，不应该报内存溢出呀？

或者是 put 方法有缺陷，产生了内存泄露？不可能，这里还有 7MB 内存可用，应该要用尽了才会出现内存泄露啊。

为了更加清晰地理解该问题，我们与 ArrayList 做一个对比，把相同数据插入到 ArrayList 中看看会怎么样，代码如下：

```

public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    /*Runtime 增加的钩子函数相同，不再赘述*/
    // 放入 40 万同样字符串
    for (int i = 0; i < 400000; i++) {
        list.add("key" + i);
    }
}

```

```

        list.add("vlaue" + i);
    }
}

```

同样的程序，只是把 HashMap 修改成了 List，增加的字符串元素也相同（只是 HashMap 将其拆分成了两个字符串，一个是 key，一个是 value，此处则是把两个字符串放到 list 中），我们来看运行结果：

```

最大可用内存: 63M
对内存大小: 63M
空闲内存: 11M

```

ArrayList 运行很正常，没有出现内存溢出情况。两个容器，容纳的元素相同，数量相同，ArrayList 没有溢出，但 HashMap 却溢出了。很明显，这与 HashMap 内部的处理机制有极大的关系。

HashMap 在底层也是以数组方式保存元素的，其中每一个键值对就是一个元素，也就是说 HashMap 把键值对封装成了一个 Entry 对象，然后再把 Entry 放到了数组中，我们简单看一下 Entry 类：

```

static class Entry<K,V> implements Map.Entry<K,V> {
    // 键
    final K key;
    // 值
    V value;
    // 相同哈希码的下一个元素
    Entry<K,V> next;

    final int hash;

    /*key、value 的 getter/setter 方法，以及重写的 equals、hashCode、toString 方法 */
}
}

```

HashMap 底层的数组变量名叫 table，它是 Entry 类型的数组，保存的是一个一个的键值对（在我们的例子中 Entry 是由两个 String 类型组成的）。再回过头来想想，对我们的例子来说，HashMap 比 ArrayList 多了一次封装，把 String 类型的键值对转换成 Entry 对象后再放入数组，这就多了 40 万个对象，这应该是问题产生的第一个原因。

我们知道 HashMap 的长度也是可以动态增加的，它的扩容机制与 ArrayList 稍有不同，其代码如下：

```

if (size++ >= threshold)
    resize(2 * table.length);

```

在插入键值对时，会做长度校验，如果大于或等于阀值（threshold 变量），则数组长度增大一倍。不过，默认的阀值是多大的呢？默认是当前长度与加载因子的乘积。

```
threshold = (int)(newCapacity * loadFactor);
```

默认的加载因子（loadFactor 变量）是 0.75，也就是说只要 HashMap 的 size 大于数组长度的 0.75 倍时，就开始扩容，经过计算得知（怎么计算的？查找 2 的 N 次幂大于 40 万的最小值即为数组的最大长度，再乘以 0.75 就是最后一次扩容点，计算的结果是 N=19），在 Map 的 size 为 393216 时，符合了扩容条件，于是 393216 个元素准备开始大搬家，要扩容嘛，那首先要申请一个长度为 1048576（当前长度的两倍嘛，2 的 19 次方再乘以 2，即 2 的 20 次方）的数组，但问题是此时剩余的内存只有 7MB 了，不足以支撑此运算，于是就报内存溢出了！这是第二个原因，也是最根本的原因。

这也就解释了为什么还剩余着 7MB 内存就报内存溢出了。我们再来思考一下 ArrayList 的扩容策略，它是在小于数组长度的时候才会扩容 1.5 倍，经过计算得知，ArrayList 的 size 在超过 80 万后（一次加两个元素，40 万的两倍），最近的一次扩容会是在 size 为 1005308 时，也就是说，如果程序设置了增加元素的上限为 502655，同样会报内存溢出，因为它也要申请一个 1507963 长度的数组，如果没这么大的地方，就会报错了。

综合来说，HashMap 比 ArrayList 多了一个层 Entry 的底层对象封装，多占用了内存，并且它的扩容策略是 2 倍长度的递增，同时还会依据阀值判断规则进行判断，因此相对于 ArrayList 来说，它就会先出现内存溢出。

可能有读者在想，是不是可以在声明时指定 HashMap 的默认长度和加载因子来减少此问题的发生。是可以缓解此问题，可以不再频繁地进行数组扩容，但仍然避免不了内存溢出问题，因为键值对的封装对象 Entry 还是少不了的，内存依然增长较快。

注意 尽量让 HashMap 中的元素少量并简单。

建议 79：集合中的哈希码不要重复

在一个列表中查找某值是非常耗费资源的，随机存取的列表是遍历查找，顺序存储列表是链表查找，或者是 Collections 的二分法查找，但这都不够快，毕竟都是遍历嘛，最快的还要数以 Hash 开头的集合（如 HashMap、HashSet 等类）查找，我们以 HashMap 为例，看看它是如何查找 Key 值的，代码如下：

```
public static void main(String[] args) {
    int size = 10000;
    List<String> list = new ArrayList<String>(size);
    // 初始化
    for (int i = 0; i < size; i++) {
        list.add("value" + i);
    }
    // 记录开始时间，单位纳秒
```

```

long start = System.nanoTime();
// 开始查找
list.contains("value" + (size - 1));
// 记录结束时间，单位纳秒
long end = System.nanoTime();
System.out.println("list 执行时间: " + (end - start) + "ns");
//Map 的查找时间
Map<String, String> map = new HashMap<String, String>(size);
for (int i = 0; i < size; i++) {
    map.put("key" + i, "value" + i);
}
start = System.nanoTime();
map.containsKey("key" + (size - 1));
end = System.nanoTime();
System.out.println("map 执行时间: " + (end - start) + "ns");
}

```

两个不同的集合容器，一个是 ArrayList，一个是 HashMap，都是插入 10000 个元素，然后判断是否包含最后一个加入的元素。逻辑相同，但是执行的时间差别却非常大，结果如下：

```

list 执行时间: 982527ns
map 执行时间: 21231ns

```

HashMap 比 ArrayList 快了 40 多倍！两者的 contains 方法都是判断是否包含指定值，为何差距如此巨大呢？而且如果数据量增大，差距也会非线性地增大。

我们先来看 ArrayList，它的 contains 就是一个遍历对比，for 循环逐个进行遍历，判断 equals 的返回值是否为 true，为 true 即找到结果，不再遍历，这很简单，不再多说。

我们再来看看 HashMap 的 containsKey 方法是如何实现的，代码如下：

```

public boolean containsKey(Object key) {
    // 判断 getEntry 是否为空
    return getEntry(key) != null;
}

```

getEntry 方法会根据 key 值查找它的键值对（也就是 Entry 对象），如果没有找到，则返回 null。我们再来看看该方法又是如何实现的，代码如下：

```

final Entry<K,V> getEntry(Object key) {
    // 计算 key 的哈希码
    int hash = (key == null) ? 0 : key.hashCode();
    // 定位 Entry, indexFor 方法是根据 hash 定位数组的位置的
    for (Entry<K,V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {
        Object k;
        // 哈希码相同，并且键也相等才符合条件
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
            return e;
    }
}

```

```

    }
    return null;
}
}

```

注意看加粗字体部分，通过 `indexFor` 方法定位 `Entry` 在数组 `table` 中的位置，这是 `HashMap` 实现的一个关键点，怎么能根据 `hashCode` 定位它在数组中的位置呢？

要解释清楚这个问题，还需要从 `HashMap` 的 `table` 数组是如何存储元素的说起。首先要说明以下三点：

- `table` 数组的长度永远是 2 的 N 次幂。
- `table` 数组中的元素是 `Entry` 类型。
- `table` 数组中的元素位置是不连续的。

`table` 数组为何如此设计呢？下面逐步来说明。先来看 `HashMap` 是如何插入元素的，代码如下：

```

public V put(K key, V value) {
    //null 键处理
    if (key == null)  return putForNullKey(value);
    // 计算 hash 码，并定位元素
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        // 哈希码相同，并 key 相等，则覆盖
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    // 插入新元素，或者替换同哈希的旧元素并建立链表
    addEntry(hash, key, value, i);
    return null;
}

```

注意看，`HashMap` 每次增加元素时都会先计算其哈希码，然后使用 `hash` 方法再次对 `hashCode` 进行抽取和统计，同时兼顾哈希码的高位和低位信息产生一个唯一值，也就是说 `hashCode` 不同，`hash` 方法返回的值也不同。之后再通过 `indexFor` 方法与数组长度做一次与运算，即可计算出其在数组中的位置，简单地说，`hash` 方法和 `indexFor` 方法就是把哈希码转变成数组的下标，源代码如下：

```

static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
}

```

```

    return h ^ (h >>> 7) ^ (h >>> 4);
}

static int indexFor(int h, int length) {
    return h & (length-1);
}

```

这两个方法相当有深度，读者有兴趣可以深入研究一下，这已经超出了本书范畴，不再赘述。顺便说明一下，null 值也是可以作为 key 值的，它的位置永远是在 Entry 数组中的第一位。

现在有一个很重要的问题摆在面前了：哈希运算存在着哈希冲突问题，即对于一个固定的哈希算法 $f(k)$ ，允许出现 $f(k1)=f(k2)$ ，但 $k1 \neq k2$ 的情况，也就是说两个不同的 Entry，可能产生相同的哈希码，HashMap 是如何处理这种冲突问题的呢？答案是通过链表，每个键值对都是一个 Entry，其中每个 Entry 都有一个 next 变量，也就是说它会指向下一个键值对——很明显，这应该是一个单向链表，该链表是由 addEntry 方法完成的，其代码如下：

```

void addEntry(int hash, K key, V value, int bucketIndex) {
    // 取得当前位置元素
    Entry<K,V> e = table[bucketIndex];
    // 生成新的键值对，并进行替换，建立链表
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    // 判断是否需要扩容
    if (size++ >= threshold)
        resize(2 * table.length);
}

```

这段程序涵盖了两个业务逻辑：如果新加入的键值对的 hashCode 是唯一的，那直接插入的数组中，它 Entry 的 next 值则为 null；如果新加入的键值对的 hashCode 与其他元素冲突，则替换掉数组中的当前值，并把新加入的 Entry 的 next 变量指向被替换掉的元素——于是，一个链表就生成了，可以用图 5-1 来表示。

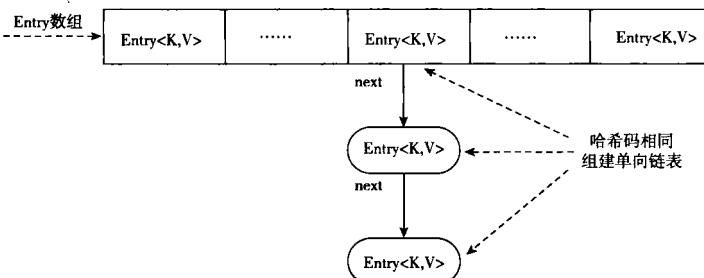


图 5-1 HashMap 存储结构图

HashMap 的存储主线还是数组，遇到哈希冲突的时候则使用链表解决。了解了 HashMap 是如何存储的，查找也就一目了然了：使用 hashCode 定位元素，若有哈希冲突，则遍历对比，换句话说在没有哈希冲突的情况下，HashMap 的查找则是依赖 hashCode 定位

的，因为是直接定位，那效率当然就高了！

知道 HashMap 的查找原理，我们就应该很清楚：如果哈希码相同，它的查找效率就与 ArrayList 没什么两样了，遍历对比，性能会大打折扣。特别是在那些进度紧张的项目中，虽重写了 hashCode 方法但返回值却是固定的，此时如果把这些对象插入到 HashMap 中，查找就相当耗时了。

注意 HashMap 中的 hashCode 应避免冲突。

建议 80：多线程使用 Vector 或 HashTable

Vector 是 ArrayList 的多线程版本，HashTable 是 HashMap 的多线程版本，这些概念我们都很清楚，也被前辈嘱咐过很多次，但我们经常会逃避使用 Vector 和 HashTable，因为用得少，不熟嘛！只有在真正需要的时候才会想要使用它们，但问题是什么时候算真正需要呢？我们来看一个例子，看看使用线程安全的 Vector 是否可以解决问题，代码如下：

```
public static void main(String[] args) {
    // 火车票列表
    final List<String> tickets = new ArrayList<String>();
    // 初始化票据池
    for (int i = 0; i < 100000; i++) {
        tickets.add("火车票 " + i);
    }
    // 退票
    Thread returnThread = new Thread() {
        public void run() {
            while (true) {
                tickets.add("车票 " + new Random().nextInt());
            }
        }
    };
    // 售票
    Thread saleThread = new Thread() {
        public void run() {
            for (String ticket : tickets) {
                tickets.remove(ticket);
            }
        }
    };
    // 启动退票线程
    returnThread.start();
    // 启动售票线程
    saleThread.start();
}
```

模拟火车站售票程序，先初始化一堆火车票，然后开始出售，同时也有退票产生，这段

程序有没有问题？可能会有读者看出了问题，`ArrayList` 是线程不安全的，两个线程访问同一个 `ArrayList` 数组肯定会有问题。

没错，确定有问题，运行结果如下：

```
Exception in thread "Thread-1" java.util.ConcurrentModificationException
  at java.util.AbstractList$Iter.checkForComodification(AbstractList.java:372)
  at java.util.AbstractList$Iter.next(AbstractList.java:343)
  at Client$2.run(Client.java:28)
```

运气好的话，该异常马上就会报出。也许有人会说这是一个典型错误，只须把 `ArrayList` 替换成 `Vector` 即可解决问题，真的是这样吗？我们把 `ArrayList` 替换成 `Vector` 后，结果照旧，仍然抛出相同的异常，`Vector` 已经是线程安全的，为什么还报这个错误呢？

这是因为他混淆了线程安全和同步修改异常，基本上所有的集合类都有一个叫做快速失败 (Fail-Fast) 的校验机制，当一个集合在被多个线程修改并访问时，就可能会出现 `ConcurrentModificationException` 异常，这是为了确保集合方法一致而设置的保护措施，它的实现原理就是我们经常提到的 `modCount` 修改计数器：如果在读列表时，`modCount` 发生变化（也就是有其他线程修改）则会抛出 `ConcurrentModificationException` 异常。这与线程同步是两码事，线程同步是为了保护集合中的数据不被脏读、脏写而设置的，我们来看线程安全到底用在什么地方，代码如下：

```
public static void main(String[] args) {
    // 火车票列表
    final List<String> tickets = new ArrayList<String>();
    // 初始化票据池
    for(int i=0;i<100000;i++){
        tickets.add("火车票 " + i);
    }
    //10 个窗口售票
    for(int i=0;i<10;i++){
        new Thread(){
            public void run() {
                while(true){
                    System.out.println(Thread.currentThread().getId()
                        +"--"+ tickets.remove(0));
                }
            };
        }.start();
    }
}
```

还是火车站售票程序，有 10 个窗口在卖火车票，程序打印出窗口号（也就是线程号）和车票编号，很快我们就会看到这样的输出：

```
13—火车票 96531
10—火车票 96531
```

```
9—火车票 96530
16—火车票 96530
```

注意看，上面有两个线程在卖同一张火车票，这才是线程不同步的问题，此时把 ArrayList 修改为 Vector 即可解决问题，因为 Vector 的每个方法前都加上了 synchronized 关键字，同时只会允许一个线程进入该方法，确保了程序的可靠性。

虽然在系统开发中我们一再说明，除非必要，否则不要使用 synchronized，这是从性能的角度考虑的，但是一旦涉及多线程时（注意这里说的是真正的多线程，不是并发修改的问题，比如一个线程增加，一个线程删除，这不属于多线程的范畴），Vector 会是最佳选择，当然自己在程序中加 synchronized 也是可行的方法。

HashMap 的线程安全类 HashTable 与此相同，不再赘述。

注意 多线程环境下考虑使用 Vector 或 HashTable。

建议 81：非稳定排序推荐使用 List

我们知道 Set 与 List 的最大区别就是 Set 中的元素不可以重复（这个重复指的 equals 方法的返回值相等），其他方面则没有太大的区别了，在 Set 的实现类中有一个比较常用的类需要了解一下：TreeSet，该类实现了类默认排序为升序的 Set 集合，如果插入一个元素，默认会按照升序排列（当然是根据 Comparable 接口的 compareTo 的返回值确定排序位置了），不过，这样的排序是不是在元素经常变化的场景中也适用呢？我们来看例子：

```
public static void main(String[] args) {
    SortedSet<Person> set = new TreeSet<Person>();
    // 身高 180CM
    set.add(new Person(180));
    // 身高 175CM
    set.add(new Person(175));

    for(Person p:set){
        System.out.println(" 身高 :" +p.getHeight());
    }
}

static class Person implements Comparable<Person>{
    // 身高
    private int height;

    public Person(int _age){
        height = _age;
    }
    /*height 的 getter/setter 方法省略 */
}
```

```
// 按照身高排序
public int compareTo(Person o) {
    return height - o.height;
}
}
```

这是 set 的简单用法，定义一个 Set 集合，之后放入两个元素，虽然 175 后放入，但是由于是按照升序排列的，所以输出的结果应该是 175 在前，180 在后，结果如下：

```
身高 :175
身高 :180
```

这没有问题，随着时间的推移，身高 175cm 的人长高了 10cm，而 180cm 却保持不变，那排序的位置应该改变一下吧，看代码：

```
public static void main(String[] args) {
    SortedSet<Person> set = new TreeSet<Person>();
    // 身高 180CM
    set.add(new Person(180));
    // 身高 175CM
    set.add(new Person(175));
    // 身高最矮的人大变身
    set.first().setHeight(185);
    for(Person p:set){
        System.out.println(" 身高 :" + p.getHeight());
    }
}
```

找出身高最矮的人，也就是排在第一个位的人，然后修改一下身高值，我们猜想一下输出结果是什么？重新排序了？看输出：

```
身高 :185
身高 :180
```

很可惜，竟然没有重新排序，偏离了我们的预期。这正是下面要说明的问题，SortedSet 接口（TreeSet 实现了该接口）只是定义了在给集合加入元素时将其进行排序，并不能保证元素修改后的排序结果，因此 TreeSet 适用于不变量的集合数据排序，比如 String、Integer 等类型，但不适用于可变量的排序，特别是不确定何时元素会发生变化的数据集合。

原因知道了，那如何解决此类重排序问题呢？有两种方式：

(1) Set 集合重排序

重新生成一个 Set 对象，也就是对原有的 Set 对象重排序，代码如下：

```
public static void main(String[] args) {
    SortedSet<Person> set = new TreeSet<Person>();
    // 身高 180CM
    set.add(new Person(180));
    // 身高 175CM
```

```

set.add(new Person(175));
// 身高最矮的人大变身
set.first().setHeight(185);
//set 重排序
set = new TreeSet<Person>(new ArrayList<Person>(set));
}

```

就这一句话即可重新排序。可能有读者会问，使用 `TreeSet(SortedSet<E> s)` 这个构造函数不是可以更好地解决问题吗？不行，该构造函数只是原 `Set` 的浅拷贝，如果里面有相同的元素，是不会重新排序的。

(2) 彻底重构掉 `TreeSet`，使用 `List` 解决问题

我们之所以使用 `TreeSet` 是希望实现自动排序，即使修改也能自动排序，既然它无法实现，那就用 `List` 来代替，然后再使用 `Collections.sort()` 方法对 `List` 排序，代码比较简单，不再赘述。

两种方法都可以解决我们的困境，到底哪一个是最优的呢？对于不变量的排序，例如直接量（也就是 8 个基本类型）、`String` 类型等，推荐使用 `TreeSet`，而对于可变量，例如我们自己写的类，可能会在逻辑处理中改变其排序关键值的，则建议使用 `List` 自行排序。

又有问题了，如果需要保证集合中元素的唯一性，又要保证元素值修改后排序正确，那该如何处理呢？`List` 不能保证集合中的元素唯一，它是可以重复的，而 `Set` 能保证元素唯一，不重复。如果采用 `List` 解决排序问题，就需要自行解决元素重复问题（若要剔除也很简单，转变为 `HashSet`，剔除后再转回来）。若采用 `TreeSet`，则需要解决元素修改后的排序问题，孰是孰非，就需要根据具体的开发场景来决定了。

注意 `SortedSet` 中的元素被修改后可能会影响其排序位置。

建议 82：由点及面，一叶知秋——集合大家族

Java 中的集合类实在是太丰富了，有常用的 `ArrayList`、`HashMap`，也有不常用的 `Stack`、`Queue`，有线程安全的 `Vector`、`HashTable`，也有线程不安全的 `LinkedList`、`TreeMap`，有阻塞式的 `ArrayBlockingQueue`，也有非阻塞式的 `PriorityQueue` 等，整个集合家族非常庞大，而且也是错综复杂，可以划分为以下几类：

(1) List

实现 `List` 接口的集合主要有：`ArrayList`、`LinkedList`、`Vector`、`Stack`，其中 `ArrayList` 是一个动态数组，`LinkedList` 是一个双向链表，`Vector` 是一个线程安全的动态数组，`Stack` 是一个对象栈，遵循先进后出的原则。

(2) Set

`Set` 是不包含重复元素的集合，其主要的实现类有：`EnumSet`、`HashSet`、`TreeSet`，其中

EnumSet 是枚举类型的专用 Set，所有元素都是枚举类型； HashSet 是以哈希码决定其元素位置的 Set，其原理与 HashMap 相似，它提供快速的插入和查找方法； TreeSet 是一个自动排序的 Set，它实现了 SortedSet 接口。

(3) Map

Map 是一个大家族，它可以分为排序 Map 和非排序 Map，排序 Map 主要是 TreeMap 类，它根据 Key 值进行自动排序；非排序 Map 主要包括：HashMap、HashTable、Properties、EnumMap 等，其中 Properties 是 HashTable 的子类，它的主要用途是从 Property 文件中加载数据，并提供方便的读写操作；EnumMap 则是要求其 Key 必须是某一个枚举类型。

Map 中还有一个 WeakHashMap 类需要说明，它是一个采用弱键方式实现的 Map 类，它的特点是：WeakHashMap 对象的存在并不会阻止垃圾回收器对键值对的回收，也就是说使用 WeakHashMap 装载数据不用担心内存溢出的问题，GC 会自动删除不用的键值对，这是好事。但也存在一个严重问题：GC 是静悄悄回收的（何时回收？God knows!），我们的程序无法知晓该动作，存在着重大的隐患。

(4) Queue

队列，它分为两类，一类是阻塞式队列，队列满了以后再插入元素则会抛出异常，主要包括：ArrayBlockingQueue、PriorityBlockingQueue、LinkedBlockingQueue，其中 ArrayBlockingQueue 是一个以数组方式实现的有界阻塞队列，PriorityBlockingQueue 是依照优先级组建的队列，LinkedBlockingQueue 是通过链表实现的阻塞队列；另一类是非阻塞队列，无边界的，只要内存允许，都可以持续追加元素，我们最经常使用的是 PriorityQueue 类。

还有一种队列，是双端队列，支持在头、尾两端插入和移除元素，它的主要实现类是：ArrayDeque、LinkedBlockingDeque、LinkedList。

(5) 数组

数组与集合的最大区别就是数组能够容纳基本类型，而集合就不行，更重要的一点就是所有的集合底层存储的都是数组。

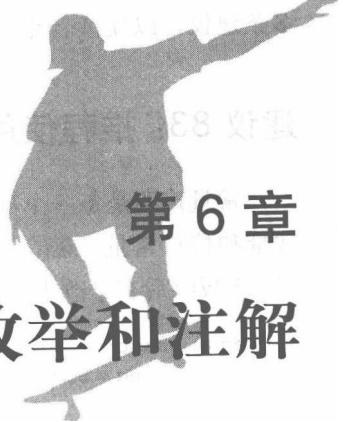
(6) 工具类

数组的工具类是 java.util.Arrays 和 java.lang.reflect.Array，集合的工具类是 java.util.Collections，有了这两个工具类，操作数组和集合会易如反掌，得心应手。

(7) 扩展类

集合类当然可以自行扩展了，想写一个自己的 List？没问题，但最好的办法还是“拿来主义”，可以使用 Apache 的 commons-collections 扩展包，也可以使用 Google 的 google-collections 扩展包，这些足以应对我们的开发需要。

注意 commons-collections、google-collections 是 JDK 之外的优秀数据集合工具包，使用拿来主义即可。



第6章

枚举和注解

日光之下，并无新事。

——《圣经》

枚举和注解都是在 Java 1.5 中引入的，虽然它们是后起之秀，但其功效不可小觑，枚举改变了常量的声明方式，注解耦合了数据和代码。本章就如何更好地使用注解和枚举提出了多条建议，以便读者能够在系统开发中更好地使用它们。

建议 83：推荐使用枚举定义常量

常量声明是每一个项目都不可或缺的，在 Java 1.5 之前，我们只有两种方式的声明：类常量和接口常量，若在项目中使用的是 Java 1.5 之前的版本基本上都是如此定义的。不过，在 1.5 版以后有了改进，即新增了一种常量声明方式：枚举声明常量，看如下代码：

```
enum Season {
    Spring, Summer, Autumn, Winter;
}
```

这是一个简单的枚举常量命名，清晰又简单。顺便提一句，JLS（Java Language Specification，Java 语言规范）提倡枚举项全部大写，字母之间用下划线分隔，这也是从常量的角度考虑的（当然，使用类似类名的命名方式也是比较友好的）。

可能有读者要问了：枚举常量与我们经常使用的类常量和静态常量相比有什么优势？问得好，枚举的优点主要表现在以下四个方面。

（1）枚举常量更简单

简不简单，我们来对比一下两者的定义和使用情况就知道了。先把 Season 枚举翻写成接口常量，代码如下：

```
interface Season {
    int Spring = 0;
    int Summer = 1;
    int Autumn = 2;
    int Winter = 3;
}
```

此处定义了春夏秋冬四个季节，类型都是 int，这与 Season 枚举的排序值是相同的。首先对比一下两者的定义，枚举常量只需要定义每个枚举项，不需要定义枚举值，而接口常量（或类常量）则必须定义值，否则编译通不过，即使我们不需要关注其值是多少也必须定义；其次，虽然两者被引用的方式相同（都是“类名.属性”，如 Season.Spring），但是枚举表示的是一个枚举项，字面含义是春天，而接口常量却是一个 int 类型，虽然其字面含义也是春天，但在运算中我们势必要关注其 int 值。

（2）枚举常量属于稳态型

例如，我们要给外星人描述一下地球上的春夏秋冬是什么样子的，使用接口常量应该是这样写。

```

public void describe(int s){
    //s 变量不能超越边界，校验条件
    if(s>=0 && s<4){
        switch (s) {
            case Season.Summer:
                System.out.println("Summer is very hot!");
                break;
            case Season.Winter:
                System.out.println("Winter is very cold!");
                break;
            .....
        }
    }
}

```

很简单，先使用 switch 语句判断是哪一个常量，然后输出。但问题是得对输入值进行检查，确定是否越界，如果常量非常庞大，校验输入就成了一件非常麻烦的事情，但这是一个不可逃避的过程，特别是如果我们的校验条件不严格，虽然编译照样可以通过，但是运行期就会产生无法预知的后果。

我们再来看看枚举常量是否能够避免校验问题，代码如下：

```

public void describe(Season s){
    switch (s) {
        case Summer:
            System.out.println(Season.Summer + " is very hot");
            break;
        case Winter:
            System.out.println(Season.Winter + "is very cold");
            break;
        .....
    }
}

```

不用校验，已经限定了是 Season 枚举，所以只能是 Season 类的四个实例，即春夏秋冬 4 个枚举项，想输入一个 int 类型或其他类型？门都没有！这也是我们最看重枚举的地方：在编译期间限定类型，不允许发生越界的情况。

(3) 枚举具有内置方法

有一个很简单的问题：如果要列出所有的季节常量，如何实现呢？接口常量或类常量可以通过反射来实现，这没错，只是虽然能实现，但会非常繁琐，读者有兴趣可以自己写一个反射类实现此功能（当然，一个一个地手动打印输出常量，也可以算是列出）。对于此类问题，使用枚举就可以非常简单地解决，代码如下：

```

public static void main(String[] args) {
    for(Season s:Season.values()){
}

```

```

        System.out.println(s);
    }
}

```

通过 values 方法获得所有的枚举项，然后打印出来即可。如此简单，得益于枚举内置的方法，每个枚举都是 java.lang.Enum 的子类，该基类提供了诸如获得排序值的 ordinal 方法、compareTo 比较方法等，大大简化了常量的访问。

(4) 枚举可以自定义方法

这一点似乎并不是枚举的优点，类常量也可以有自己的方法呀，但关键是枚举常量不仅可以定义静态方法，还可以定义非静态方法，而且还能能够从根本上杜绝常量类被实例化。比如我们要在常量定义中获得最舒服季节的方法，使用常量枚举的代码如下所示：

```

enum Season {
    Spring, Summer, Autumn, Winter;
    // 最舒服的季节
    public static Season getComfortableSeason() {
        return Spring;
    }
}

```

我们知道每个枚举项都是该枚举的一个实例，对于我们的例子来说，也就表示 Spring 其实是 Season 的一个实例，Summer 也是其中一个实例，那我们在枚举中定义的静态方法既可以在类（也就是枚举 Season）中引用，也可以在实例（也就是枚举项 Spring、Summer、Autumn、Winter）中引用，看如下代码：

```

public static void main(String[] args) {
    System.out.println("The most comfortable season is " + Season.getComfortableSeason());
}

```

那如果使用类常量要如何实现呢？代码如下：

```

class Season {
    public final static int Spring = 0;
    public final static int Summer = 1;
    public final static int Autumn = 2;
    public final static int Winter = 3;
    // 最舒服的季节
    public static int getComfortableSeason(){
        return Spring;
    }
}

```

想想看，我们要怎么才能打印出 “The most comfortable season is Spring” 这句话呢？除了使用 switch 判断外没有其他办法了。

虽然枚举常量在很多方面比接口常量和类常量好用，但是有一点它是比不上接口常量和

类常量的，那就是继承，枚举类型是不能有继承的，也就是说一个枚举常量定义完毕后，除非修改重构，否则无法做扩展，而接口常量和类常量则可以通过继承进行扩展。但是，一般常量在项目构建时就定义完毕了，很少会出现必须通过扩展才能实现业务逻辑的场景。

注意 在项目开发中，推荐使用枚举常量代替接口常量或类常量。

建议 84：使用构造函数协助描述枚举项

一般来说，我们经常使用的枚举项只有一个属性，即排序号，其默认值是从 0、1、2……这一点我们非常熟悉。但是除了排序号外，枚举还有一个（或多个）属性：枚举描述，它的含义是通过枚举的构造函数，声明每个枚举项（也就是枚举的实例）必须具有的属性和行为，这是对枚举项的描述或补充，目的是使枚举项表述的意义更加清晰准确。例如有这样的代码：

```
enum Season {
    Spring("春"), Summer("夏"), Autumn("秋"), Winter("冬");
    private String desc;
    Season(String _desc) {
        desc = _desc;
    }
    // 获得枚举描述
    public String getDesc() {
        return desc;
    }
}
```

其枚举项是英文的，描述是中文的，如此设计使其表述的意义更加精确，方便了多个协作者共同引用该常量。若不考虑描述的使用（即访问 getDesc 方法），它与如下接口定义的描述则很相似：

```
interface Season{
    //春
    int Spring = 0;
    //夏
    int Summer = 1;
    /*.....*/
}
```

比较两段代码，很容易看出使用枚举项描述是一个很好的解决方案，非常简单、清晰。因为是一个描述（Description），那我们在开发时就可以赋予更多的含义了，比如可以通过枚举构造函数声明业务值，定义可选项，添加属性等，看如下代码：

```
enum Role {
```

```

Admin("管理员",new Lifetime(), new Scope()),
User("普通用户",new Lifetime(), new Scope());
// 中文描述
private String name;
// 角色的生命期
private Lifetime lifeTime;
// 权限范围
private Scope scope;

Role(String _name,Lifetime _lt, Scope _scope) {
    name = _name;
    lifeTime = _lt;
    scope = _scope;
}
/*name、lifeTime、scope 的 get 方法较简单，不再赘述*/
}

```

这是一个角色定义类，描述了两个角色：管理员（Admin）和普通用户（User），同时它还通过构造函数对这两个角色进行了描述：

- name：表示的是该角色的中文名称。
- lifeTime：表示的是该角色的生命期，也就是多长时间该角色失效。
- scope：表示的是该角色的权限范围。

读者可以看出，这样一个描述可以使开发者对 Admin、User 两个常量有一个立体多维度的认知，有名称、生命期，还有范围，而且还可以在程序中方便地获得此类的属性。

推荐大家在枚举定义中为每个枚举项定义描述，特别是在大规模的项目开发中，大量的常量项定义使用枚举项描述比在接口常量或类常量中增加注释的方式友好得多，简洁得多。

建议 85：小心 switch 带来的空值异常

使用枚举定义常量时，会伴有大量的 switch 语句判断，目的是为每个枚举项解释其行为，例如这样一个方法：

```

public static void doSports(Season season) {
    switch (season) {
        case Spring:
            System.out.println("春天放风筝");
            break;
        case Summer:
            System.out.println("夏天游泳");
            break;
        case Autumn:
            System.out.println("秋天捉知了");
            break;
        case Winter:
    }
}

```

```

        System.out.println("冬天滑冰");
        break;
    default:
        System.out.println("输入错误！");
        break;
    }
}

```

上面的代码中输入了一个 Season 类型的枚举，然后使用 switch 进行匹配，目的是输出每个季节能进行的活动。现在的问题是：这段代码有没有问题？

我们先来看它是如何被调用的，因为要传递进来的是 Season 类型，也就是一个实例对象，那当然应该允许为空了，我们就传递一个 null 值进去看看有没有问题，代码如下：

```

public static void main(String[] args) {
    doSports(null);
}

```

似乎会打印出“输出错误！”，因为在 switch 中没有匹配到指定值，所以会打印出 default 的代码块，是这样的吗？不是，运行后的输出如下所示：

```

Exception in thread "main" java.lang.NullPointerException
at Client.doSports(Client.java:9)
at Client.main(Client.java:5)

```

竟然是空指针异常，第 9 行也就是 switch 那一行，怎么会有空指针呢？这就与枚举和 switch 的特性有关了，此问题也是在开发中经常发生的。我们知道，目前 Java 中的 switch 语句只能判断 byte、short、char、int 类型（JDK 7 已经允许使用 String 类型），这是 Java 编译器的限制。问题是为什么枚举类型也可以跟在 switch 后面呢？

很简单，因为编译时，编译器判断出 switch 语句后的参数是枚举类型，然后就会根据枚举的排序值继续匹配，也就是说上面的代码与以下代码相同：

```

public static void doSports(Season season) {
    switch (season.ordinal()) {
        case Season.Spring.ordinal():
            .....
        case Season.Summer.ordinal():
            .....
    }
}

```

看明白了吧，switch 语句是先计算 season 变量的排序值，然后与枚举常量的每个排序值进行对比的。在我们的例子中 season 变量是 null 值，无法执行 ordinal 方法，于是报空指针异常了。

问题清楚了，解决方法也很简单，在 doSports 方法中判断输入参数是否是 null 即可。

建议 86：在 switch 的 default 代码块中增加 AssertionError 错误

switch 后跟枚举类型，case 后列出所有的枚举项，这是一个使用枚举的主流写法，那留着 default 语句似乎没有任何作用，程序已经列举了所有的可能选项，肯定不会执行到 default 语句，看上去纯属多余嘛！错了，这个 default 还是很有用的。以我们定义的日志级别来举例说明，这是一个典型的枚举常量，如下所示：

```
enum LogLevel {
    DEBUG, INFO, WARN, ERROR;
}
```

一般在使用的时候，会通过 switch 语句来决定用户设置的级别，然后输出不同级别的日志，代码如下：

```
switch (logLevel) {
    case DEBUG:
        .....
    case INFO:
        .....
    case WARN:
        .....
    case ERROR:
        .....
}
```

由于把所有的枚举项都列举完了，不可能有其他值，所以就不需要 default 代码块了，这是普遍的认识，但问题是我们的 switch 代码与枚举之间没有强制约束关系，也就是说两者只是在语义上建立了联系，并没有一个强制约束，比如 LogLevel 枚举发生改变，增加了一个枚举项 FATAL，如果此时我们对 switch 语句不做任何修改，编译虽不会出现问题，但是运行期会发生非预期的错误：FATAL 类型的日志没有输出。

为了避免出现这类错误，建议在 default 后直接抛出一个 AssertionError 错误，其含义就是“不要跑到这里来，一跑到这里就会出问题”，这样可以保证在增加一个枚举项的情况下，若其他代码未修改，运行期马上就会报错，这样一来就很容易查找到错误，方便立刻排除。

当然也有其他方法解决此问题，比如修改 IDE 工具，以 Eclipse 为例，可以把 Java → Compiler → Errors/Warnings 中的“Enum type constant not covered on 'switch'" 设置为 Error 级别，如果不判断所有的枚举项就不能通过编译。

建议 87：使用 valueOf 前必须进行校验

我们知道每个枚举都是 java.lang.Enum 的子类，都可以访问 Enum 类提供的方法，比如 hashCode、name、valueOf 等，其中 valueOf 方法会把一个 String 类型的名称转变为枚举项，

也就是在枚举项中查找出字面值与该参数相等的枚举项。虽然这个方法很简单，但是JDK却做了一个对于开发人员来说并不简单的处理。我们来看代码：

```
public static void main(String[] args) {
    // 注意 summer 是小写
    List<String> params = Arrays.asList("Spring", "summer");
    for (String name : params) {
        // 查找字面值与 name 相同的枚举项
        Season s = Season.valueOf(name);
        if (s != null) {
            // 有该枚举项时
            System.out.println(s);
        } else {
            // 没有该枚举项
            System.out.println(" 无相关枚举项 ");
        }
    }
}
```

这段程序看似很完美了，其中考虑到从 String 转换为枚举类型可能存在着转换不成功的情况，比如没有匹配到指定值，此时 valueOf 的返回值应该为空，所以后面又紧跟着 if…else 判断输出。这段程序真的完美无缺了吗？那我们看看运行结果：

```
Spring
Exception in thread "main" java.lang.IllegalArgumentException: No enum const class
    Season.summer
    at java.lang.Enum.valueOf(Enum.java:196)
    at Season.valueOf(Client.java:1)
    at Client.main(Client.java:13)
```

报无效参数异常，也就是说我们的 summer（注意 s 小写）无法转换为 Season 枚举，无法转换就不转换嘛，那也别抛出非受检 IllegalArgumentException 异常啊，一旦抛出这个异常，后续的代码就不会运行了，这才是要命呀！这与我们的习惯用法非常不一致，例如我们从一个 List 中查找一个元素，即使不存在也不会报错，顶多 indexOf 方法返回 -1。

那么来深入分析一下该问题，valueOf 方法的源代码如下：

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType,
    String name) {
    // 通过反射，从常量列表中查找
    T result = enumType.enumConstantDirectory().get(name);
    if (result != null)
        return result;
    if (name == null)
        throw new NullPointerException("Name is null");
    // 最后排除无效参数异常
    throw new IllegalArgumentException("No enum const " + enumType + "." + name);
}
```

`valueOf` 方法先通过反射从枚举类的常量声明中查找，若找到就直接返回，若找不到则抛出无效参数异常。`valueOf` 本意是保护编码中的枚举安全性，使其不产生空枚举对象，简化枚举操作，但是却又引入了一个我们无法避免的 `IllegalArgumentException` 异常。

可能会有读者认为此处 `valueOf` 方法的源代码不对，这里要输入 2 个参数，而我们的 `Season.valueOf` 只传递一个 `String` 类型的参数。真的是这样吗？是的，因为 `valueOf (String name)` 方法是不可见的，是 JVM 内置的方法，我们只有通过阅读公开的 `valueOf` 方法来了解其运行原理了。

问题清楚了，有两个方法可以解决此问题：

(1) 使用 try...catch 捕捉异常

这是最直接也是最简单的方式，产生 `IllegalArgumentException` 即可确认为没有相同名称的枚举项，代码如下：

```
try {
    Season s = Season.valueOf(name);
    // 有该枚举项时的处理
    System.out.println(s);
} catch (Exception e) {
    System.out.println("无相关枚举项");
}
```

(2) 扩展枚举类

由于 `Enum` 类定义的方法基本上都是 `final` 类型的，所以不希望被覆盖，那我们可以学习 `String` 和 `List`，通过增加一个 `contains` 方法来判断是否包含指定的枚举项，然后再继续转换，代码如下：

```
enum Season {
    Spring, Summer, Autumn, Winter;
    // 是否包含指定名称的枚举项
    public static boolean contains(String name) {
        // 所有的枚举值
        Season[] season = values();
        // 遍历查找
        for (Season s : season) {
            if (s.name().equals(name)) {
                return true;
            }
        }
        return false;
    }
}
```

`Season` 枚举具备了静态方法 `contains` 后，就可以在 `valueOf` 前判断一下是否包含指定的枚举名称了，若包含则可以通过 `valueOf` 转换为 `Season` 枚举，若不包含则不转换。

建议 88：用枚举实现工厂方法模式更简洁

工厂方法模式（Factory Method Pattern）是“创建对象的接口，让子类决定实例化哪一个类，并使一个类的实例化延迟到其子类”。工厂方法模式在我们的开发工作中经常会用到。下面以汽车制造为例，看看一般的工厂方法模式是如何实现的，代码如下：

```
// 抽象产品
interface Car {
};

// 具体产品类
class FordCar implements Car {
};

// 具体产品类
class BuickCar implements Car {
};

// 工厂类
class CarFactory {
    // 生产汽车
    public static Car createCar(Class<? extends Car> c) {
        try {
            return (Car) c.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

这是最原始的工厂方法模式，有两个产品：福特汽车和别克汽车，然后通过工厂方法模式来生产。有了工厂方法模式，我们就不用关心一辆车具体是怎么生成的了，只要告诉工厂“给我生产一辆福特汽车”就可以了，下面是产出一辆福特汽车时客户端的代码：

```
public static void main(String[] args) {
    // 生产车辆
    Car car = CarFactory.createCar(FordCar.class);
}
```

这就是我们经常使用的工厂方法模式，但经常使用并不代表就是最优秀、最简洁的。此处再介绍一种通过枚举实现工厂方法模式的方案，谁优谁劣你自行评价。枚举实现工厂方法模式有两种方法：

（1）枚举非静态方法实现工厂方法模式

我们知道每个枚举项都是该枚举的实例对象，那是不是定义一个方法可以生成每个枚举项的对应产品来实现此模式呢？代码如下：

```
enum CarFactory {
```

```

// 定义工厂类能生产汽车的类型
FordCar, BuickCar;
// 生产汽车
public Car create() {
    switch (this) {
        case FordCar:
            return new FordCar();
        case BuickCar:
            return new BuickCar();
        default:
            throw new AssertionError("无效参数");
    }
}
}

```

`create` 是一个非静态方法，也就是只有通过 `FordCar`、`BuickCar` 枚举项才能访问。采用这种方式实现工厂方法模式时，客户端要生产一辆汽车就很简单了，代码如下：

```

public static void main(String[] args) {
    // 生产汽车
    Car car = CarFactory.BuickCar.create();
}

```

(2) 通过抽象方法生成产品

枚举类型虽然不能继承，但是可以用 `abstract` 修饰其方法，此时就表示该枚举是一个抽象枚举，需要每个枚举项自行实现该方法，也就是说枚举项的类型是该枚举的一个子类，我们来看代码：

```

enum CarFactory {
    FordCar {
        public Car create() {
            return new FordCar();
        }
    },
    BuickCar {
        public Car create() {
            return new BuickCar();
        }
    };
    // 抽象生产方法
    public abstract Car create();
}

```

首先定义一个抽象制造方法 `create`，然后每个枚举项自行实现。这种方式编译后会产生两个 `CarFactory` 的匿名子类，因为每个枚举项都要实现抽象 `create` 方法。客户端的调用与上一个方案相同，不再赘述。

读者可能会问：为什么要使用枚举类型的工厂方法模式呢？那是因为使用枚举类型的工

厂方法模式有以下三个优点：

(1) 避免错误调用的发生

一般工厂方法模式中的生产方法（也就是 `createCar` 方法）可以接收三种类型的参数：类型参数（如我们的例子）、`String` 参数（生产方法中判断 `String` 参数是需要生产什么产品）、`int` 参数（根据 `int` 值判断需要生产什么类型的产品），这三种参数都是宽泛的数据类型，很容易产生错误（比如边界问题、`null` 值问题），而且出现这类错误编译器还不会报警，例如：

```
public static void main(String[] args) {
    // 生产车辆
    Car car = CarFactory.createCar(Car.class);
}
```

`Car` 是一个接口，完全合乎 `createCar` 方法的要求，所以它在编译时不会报任何错误，但一运行起来就会报 `InstantiationException` 异常。而使用枚举类型的工厂方法模式就不存在该问题了，不需要传递任何参数，只需要选择好生产什么类型的产品即可。

(2) 性能好，使用便捷

枚举类型的计算是以 `int` 类型的计算为基础的，这是最基本的操作，性能当然会快，至于使用便捷，注意看客户端的调用，代码的字面意思就是“汽车工厂，我要一辆别克汽车，赶快生产”。

(3) 降低类间耦合

不管生产方法接收的是 `Class`、`String` 还是 `int` 的参数，都会成为客户端类的负担，这些类并不是客户端需要的，而是因为工厂方法的限制必须输入的，例如 `Class` 参数，对客户端 `main` 方法来说，它需要传递一个 `FordCar.class` 参数才能生产一辆福特汽车，除了在 `create` 方法中传递该参数外，业务类不需要改 `Car` 的实现类。这严重违背了迪米特原则（Law of Demeter，简称为 LoD），也就是最少知识原则：一个对象应该对其他对象有最少的了解。

而枚举类型的工厂方法就没有这种问题了，它只需要依赖工厂类就可以生产一辆符合接口的汽车，完全可以无视具体汽车类的存在。

注意 下一次，使用枚举来实现工厂方法模式。

建议 89：枚举项的数量限制在 64 个以内

为了更好地使用枚举，Java 提供了两个枚举集合：`EnumSet` 和 `EnumMap`，这两个集合的使用方法都比较简单，`EnumSet` 表示其元素必须是某一枚举的枚举项，`EnumMap` 表示 Key 值必须是某一枚举的枚举项，由于枚举类型的实例数量固定并且有限，相对来说 `EnumSet` 和 `EnumMap` 的效率会比其他 `Set` 和 `Map` 要高。

虽然 EnumSet 很好用，但是它有一个隐藏的特点，我们逐步分析。在项目中一般会把枚举用作常量定义，可能会定义非常多的枚举项，然后通过 EnumSet 访问、遍历，但它对不同的枚举数量有不同的处理方式。为了进行对比，我们定义两个枚举，一个数量等于 64，一个是 65（大于 64 即可，为什么是 64 而不是 128、512 呢？稍后解释），代码如下：

```
// 普通枚举项，数量等于 64
enum Const {
    A, B, C, ...., PC, QC, RC;
}
// 大枚举，数量超过 64
enum LargeConst {
    A, B, C, ...., KB, LB, MB;
}
```

Const 中的枚举项数量是 64，LargeConst 的数量是 65，其中的 号代表省略的枚举项（注意此处只是省略了，Java 不支持省略号）。接下来我们希望把这两个枚举转换为 EnumSet，然后判断一下它们的 class 类型是否相同，代码如下：

```
public static void main(String[] args) {
    // 创建包含所有枚举项的 EnumSet
    EnumSet<Const> cs = EnumSet.allOf(Const.class);
    EnumSet<LargeConst> lcs = EnumSet.allOf(LargeConst.class);
    // 打印出枚举项数量
    System.out.println("Const 枚举项数量: " + cs.size());
    System.out.println("LargeConst 枚举项数量: " + lcs.size());
    // 输出两个 EnumSet 的 class
    System.out.println(cs.getClass());
    System.out.println(lcs.getClass());
}
```

程序很简单，现在的问题是：cs 和 lcs 的 class 类型是否相同？应该相同吧，都是 EnumSet 类的工厂方法 allOf 生成的 EnumSet 类，而且 JDK API 也没有提示 EnumSet 有子类。我们来看输出结果：

```
Const 枚举项数量: 64
LargeConst 枚举项数量: 65
class java.util.RegularEnumSet
class java.util.JumboEnumSet
```

很遗憾，两者不相等。就差 1 个元素，两者就不相等了？确实如此，这也是我们要重点关注枚举项数量的原因。先来看看 Java 是如何处理的，首先跟踪 allOf 方法，其源代码如下：

```
public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) {
    // 生成一个空 EnumSet
    EnumSet<E> result = noneOf(elementType);
    // 加入所有的枚举项
```

```

        result.addAll();
        return result;
    }
}

```

allOf 通过 noneOf 方法首先生成一个 EnumSet 对象，然后把所有的枚举项都加进去，问题可能就出在 EnumSet 的生成上了，我们来看 noneOf 的代码：

```

public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {
    // 获得所有枚举项
    Enum[] universe = getUniverse(elementType);
    if (universe == null)
        throw new ClassCastException(elementType + " not an enum");

    if (universe.length <= 64)
        // 枚举数量小于等于 64
        return new RegularEnumSet<E>(elementType, universe);
    else
        // 枚举数量大于 64
        return new JumboEnumSet<E>(elementType, universe);
}

```

看到这里恍然大悟，Java 原来是如此处理的：当枚举项数量小于等于 64 时，创建一个 RegularEnumSet 实例对象，大于 64 时则创建一个 JumboEnumSet 实例对象。

紧接着的问题是：为什么要如此处理呢？这还要看看这两个类之间的差异，首先看 RegularEnumSet 类，代码如下：

```

class RegularEnumSet<E extends Enum<E>> extends EnumSet<E> {
    // 记录所有枚举排序号，注意是 long 型
    private long elements = 0L;
    // 构造函数
    RegularEnumSet(Class<E>elementType, Enum[] universe) {
        super(elementType, universe);
    }
    // 加入所有元素
    void addAll() {
        if (universe.length != 0)
            elements = -1L >>> -universe.length;
    }
}

```

我们知道枚举项的排序值 ordinal 是从 0、1、2……依次递增的，没有重号，没有跳号，RegularEnumSet 就是利用这一点把每个枚举项的 ordinal 映射到一个 long 类型的每个位上的，注意看 addAll 方法的 elements 元素，它使用了无符号右移操作，并且操作数是负值，位移也是负值，这表示是负数（符号位是 1）的“无符号左移”：符号位为 0，并补充低位，简单地说，Java 把一个不多于 64 个枚举项的枚举映射到了一个 long 类型变量上。这才是 EnumSet 处理的重点，其他的 size 方法、contains 方法等都是根据 elements 计算出来的。想

想看，一个 long 类型的数字包含了所有的枚举项，其效率和性能肯定是非常优秀的。

我们知道 long 类型是 64 位的，所以 RegularEnumSet 类型也就只能负责枚举项数量不大于 64 的枚举（这也是我们以 64 来举例，而不以 128 或 512 举例的原因），大于 64 则由 JumboEnumSet 处理，我们看它是怎么处理的：

```
class JumboEnumSet<E extends Enum<E>> extends EnumSet<E> {
    // 映射所有的枚举项
    private long elements[];
    // 构造函数
    JumboEnumSet(Class<E>elementType, Enum[] universe) {
        super(elementType, universe);
        // 默认长度是枚举项数量除以 64 再加 1
        elements = new long[(universe.length + 63) >>> 6];
    }

    void addAll() {
        //elements 中每个元素表示 64 个枚举项
        for (int i = 0; i < elements.length; i++)
            elements[i] = -1;
        elements[elements.length - 1] >>>= -universe.length;
        size = universe.length;
    }
}
```

JumboEnumSet 类把枚举项按照 64 个元素一组拆分成了多组，每组都映射到一个 long 类型的数字上，然后该数组再放置到 elements 数组中。简单来说 JumboEnumSet 类的原理与 RegularEnumSet 相似，只是 JumboEnumSet 使用了 long 数组容纳更多的枚举项。

不过，你会不会觉得这两段程序看着很让人郁闷呢？其实这是因为我们在开发中很少用到位移操作。读者可以这样理解，RegularEnumSet 是把每个枚举项编码映射到一个 long 类型数字的每个位上，JumboEnumSet 是先按照 64 个一组进行拆分，然后每个组再映射到一个 long 类型数字的每个位上，从这里我们也可以看出数字编码的奥秘！

从以上的分析可以看出，EnumSet 提供的两个实现都是基本的数字类型操作，其性能肯定比其他的 Set 类型要好很多，特别是 Enum 的数量少于 64 的时候，那简直就是飞一般的速度。

注意 枚举项数量不要超过 64，否则建议拆分。

建议 90：小心注解继承

Java 从 1.5 版开始引入了注解（Annotation），其目的是在不影响代码语义的情况下增强代码的可读性，并且不改变代码的执行逻辑，对于注解始终有两派争论，正方认为注解有益

于数据与代码的耦合，“在有代码的周边集合数据”；反方认为注解把代码和数据混淆在一起，增加了代码的易变性，削弱了程序的健壮性和稳定性。这些争论暂且不表，我们要说的是一个我们不常用的元注解（Meta-Annotation）：@Inherited，它表示一个注解是否可以自动被继承，我们来看它应如何使用。

思考一个例子，比如描述鸟类，它有颜色、体型、习性等属性，我们以颜色为例，定义一个注解来修饰一下，代码如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@interface Desc {
    enum Color {
        White, Grayish, Yellow;
    }
    // 默认颜色是白色的
    Color c() default Color.White;
}
```

该注解 Desc 前增加了三个元注解：Retention 表示的是该注解的保留级别，Target 表示的是该注解可以标注在什么地方，@ Inherited 表示该注解会被自动继承。注解定义完毕，我们把它标注在类上，代码如下：

```
@Desc(c = Color.White)
abstract class Bird {
    // 鸟的颜色
    public abstract Color getColor();
}
// 麻雀
class Sparrow extends Bird {
    private Color color;
    // 默认是浅灰色
    public Sparrow() {
        color = Color.Grayish;
    }
    // 构造函数定义鸟的颜色
    public Sparrow(Color _color) {
        color = _color;
    }
    @Override
    public Color getColor() {
        return color;
    }
}
// 鸟巢，工厂方法模式
enum BirdNest {
    Sparrow;
    // 鸟类繁殖
```

```

public Bird reproduce() {
    Desc bd = Sparrow.class.getAnnotation(Desc.class);
    return bd == null ? new Sparrow() : new Sparrow(bd.c());
}
}

```

程序比较简单，声明了一个 Bird 抽象类，并且标注了 Desc 注解，描述鸟类的颜色是白色的，然后编写了一个麻雀 Sparrow 类，它有两个构造函数，一个是默认的构造函数，也就是我们经常看到的麻雀是浅灰色的，另外一个构造函数是自定义麻雀的颜色，之后又定义了一个鸟巢（工厂方法模式），它是专门负责鸟类繁殖的，它的生产方法 reproduce 会根据实现类注解信息生成不同颜色的麻雀。我们编写一个客户端调用，代码如下：

```

public static void main(String[] args) {
    Bird bird = BirdNest.Sparrow.reproduce();
    Color color = bird.getColor();
    System.out.println("Bird's color is: " + color);
}

```

现在的问题是这段客户端程序会打印出什么来？因为采用了工厂方法模式，它最主要的问题就是 bird 变量到底采用了哪个构造函数来生成，是无参构造还是有参构造？如果我们单独看子类 Sparrow，它没有被添加任何注解，那工厂方法中的 bd 变量就应该是 null 了，应该调用的是无参构造。是不是如此呢？我们来看运行结果：

```
Bird's color is: White
```

白色？这是我们添加到父类 Bird 上的颜色，为什么？就是因为我们在注解上加了 @Inherited 注解，它表示的意思是我们只要把注解 @Desc 加到父类 Bird 上，它的所有子类都会自动从父类继承 @Desc 注解，不需要显式声明，这与 Java 类的继承有点不同，若 Sparrow 类继承了 Bird，则必须使用 extends 关键字声明，而 Bird 上的注解 @Desc 继承自 Bird 却不用显式声明，只要声明 @Desc 注解是可自动继承的即可。

采用 @Inherited 元注解有利有弊，利的地方是一个注解只要标注到父类，所有的子类都会自动具有与父类相同的注解，整齐、统一而且便于管理，弊的地方是单单阅读子类代码，我们无从知道为何逻辑会被改变，因为子类没有明显标注该注解。总体上来说，使用 @Inherited 元注解的弊大于利，特别是一个类的继承层次较深时，如果注解较多，则很难判断出是哪个注解对子类产生了逻辑劫持。

建议 91：枚举和注解结合使用威力更大

我们知道注解的写法和接口很类似，都采用了关键字 interface，而且都不能有实现代码，常量定义默认都是 public static final 类型的等，它们的主要不同点是：注解要在

interface 前加上 @ 字符，而且不能继承，不能实现，这经常会给我们的开发带来一些障碍。

我们来分析一个 ACL (Access Control List, 访问控制列表) 设计案例，看看如何避免这些障碍，ACL 有三个重要元素：

- 资源，有哪些信息是要被控制起来的。
- 权限级别，不同的访问者规划在不同的级别中。
- 控制器（也叫鉴权人），控制不同的级别访问不同的资源。

鉴权人是整个 ACL 的设计核心，我们从最主要的鉴权人开始，代码如下：

```
interface Identifier {
    // 无权访问时的礼貌语
    String REFUSE_WORD = "您无权访问";
    // 鉴权
    public boolean identify();
}
```

这是一个鉴权人接口，定义了一个常量和一个鉴权方法。接下来应该实现该鉴权方法，但问题是我们的权限级别和鉴权方法之间是紧耦合，若分拆成两个类显得有点啰嗦，怎么办？我们可以直接定义一个枚举来实现，代码如下：

```
enum CommonIdentifier implements Identifier {
    // 权限级别
    Reader, Author, Admin;
    // 实现鉴权
    public boolean identify() {
        return false;
    }
}
```

定义了一个通用鉴权者，使用的是枚举类型，并且实现了鉴权者接口。现在就剩下资源定义了，这很容易定义，资源就是我们写的类、方法等，之后再通过配置来决定哪些类、方法允许什么级别的访问，这里的问题是：怎么把资源和权限级别关联起来呢？使用 XML 配置文件？是个方法，但对我们的示例程序来说显得太繁重了，如果使用注解会更简洁些，不过这需要我们首先定义出权限级别的注解，代码如下：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Access {
    // 什么级别可以访问，默认是管理员
    CommonIdentifier level() default CommonIdentifier.Admin;
}
```

该注解是标注在类上面的，并且会保留到运行期。我们定义一个资源类，代码如下：

```
@Access(level = CommonIdentifier.Author)
```

```
class Foo {  
}
```

Foo 类只能是作者级别的人访问。场景都定义完毕了，那我们看看如何模拟 ACL 的实现，代码如下：

```
public static void main(String[] args) {  
    // 初始化商业逻辑  
    Foo b = new Foo();  
    // 获取注解  
    Access access = b.getClass().getAnnotation(Access.class);  
    // 没有 Access 注解或者鉴权失败  
    if (access == null || !access.level().identify()) {  
        // 没有 Access 注解或者鉴权失败  
        System.out.println(access.level().REFUSE_WORD);  
    }  
}
```

看看这段代码，简单、易读，而且如果我们是通过 ClassLoader 类来解释该注解的，那会使我们的开发更加简洁，所有的开发人员只要增加注解即可解决访问控制问题。注意看加粗代码，access 是一个注解类型，我们想使用 Identifier 接口的 identify 鉴权方法和 REFUSE_WORD 常量，但注解是不能继承的，那怎么办？此处，可通过枚举类型 CommonIdentifier 从中间做一个委派动作（Delegate），委派？没看到！你可以让 identify 返回一个对象，或者在 Identifier 上直接定义一个常量对象，那就是“赤裸裸”的委派了！

建议 92：注意 @Override 不同版本的区别

@Override 注解用于方法的覆写上，它在编译期有效，也就是 Java 编译器在编译时会根据该注解检查方法是否真的是覆写，如果不是就报错，拒绝编译。该注解可以很大程度地解决我们的误写问题，比如子类和父类的方法名少写了一个字符，或者是数字 0 和字母 O 未区分出来等，这基本上是每个程序员都曾经犯过的错误。在代码中加上 @Override 注解基本可以杜绝出现此类问题，但是 @Override 有个版本问题，我们来看如下代码：

```
interface Foo {  
    public void doSomething();  
}  
  
class Impl implements Foo {  
    @Override  
    public void doSomething() {  
    }  
}
```

这是一个简单的 @Override 示例，接口中定义了一个 doSomething 方法，实现类 Impl

实现此方法，并且在方法前加上了 @Override 注解。这段代码在 Java 1.6 版本上编译没有任何问题，虽然 doSomething 方法只是实现了接口定义，严格来说并不能算是覆写，但 @Override 出现在这里可以减少代码中可能出现的错误。

可如果在 Java 1.5 版本上编译此段代码，就会出现如下错误：

```
The method doSomething() of type Impl must override a superclass method Client.java
```

注意，这是个错误，不能继续编译。原因是 1.5 版中 @Override 是严格遵守覆写的定义：子类方法与父类方法必须具有相同的方法名、输入参数、输出参数（允许子类缩小）、访问权限（允许子类扩大），父类必须是一个类，不能是一个接口，否则不能算是覆写。而这在 Java 1.6 就开放了很多，实现接口的方法也可以加上 @Override 注解了，可以避免粗心大意导致方法名称与接口不一致的情况发生。

在多环境部署应用时，需要考虑 @Override 在不同版本下代表的意义，如果是 Java 1.6 版本的程序移植到 1.5 版本环境中，就需要删除实现接口方法上的 @Override 注解。

第7章

泛型和反射

Don't let complexity stop you. Be activists. Take on the big inequities. It will be one of the great experiences of your lives.

不要让这个世界的复杂性阻碍你的前进。要成为一个行动主义者，将解决人类的不平等视为己任。它将成为你生命中最重要的经历之一。

——比尔·盖茨在哈佛大学的演讲

泛型可以减少强制类型的转换，可以规范集合的元素类型，还可以提高代码的安全性和可读性，正是因为有这些优点，自从 Java 引入泛型后，项目的编码规则上便多了一条：优先使用泛型。

反射可以“看透”程序的运行情况，可以让我们在运行期知晓一个类或实例的运行状况，可以动态地加载和调用，虽然有一定的性能忧患，但它带给我们的便利远远大于其性能缺陷。

建议 93：Java 的泛型是类型擦除的

Java 泛型（Generic）的引入加强了参数类型的安全性，减少了类型的转换，它与 C++ 中的模板（Templates）比较类似，但是有一点不同的是：Java 的泛型在编译期有效，在运行期被删除，也就是说所有的泛型参数类型在编译后都会被清除掉，我们来看一个例子，代码如下：

```
public class Foo {
    //arrayMethod 接收数组参数，并进行重载
    public void arrayMethod(String[] strArray) {
    }
    public void arrayMethod(Integer[] intArray) {
    }
    //listMethod 接收泛型 List 参数，并进行重载
    public void listMethod(List<String> stringList) {
    }
    public void listMethod(List<Integer> intList) {
    }
}
```

程序很简单，编写了 4 个方法，arrayMethod 方法接收 String 数组和 Integer 数组，这是一个典型的重载，listMethod 接收元素类型为 String 和 Integer 的 List 变量。现在的问题是，这段程序是否能编译？如果不能，问题出在什么地方？

事实上，这段程序是无法编译的，编译时报错信息如下：

```
Method listMethod(List<Integer>) has the same erasure listMethod(List<E>) as
another method in type Foo
```

此错误的意思是说 listMethod(List<Integer>) 方法在编译时擦除类型后的方法是 listMethod(List<E>)，它与另外一个方法重复，通俗地说就是方法签名重复。这就是 Java 泛型擦除引起的问题：在编译后所有的泛型类型都会做相应的转化。转换规则如下：

- List<String>、List<Integer>、List<T> 擦除后的类型为 List。
- List<String>[] 擦除后的类型为 List[]。
- List<? extends E>、List<? super E> 擦除后的类型为 List<E>。

□ `List< T extends Serializable & Cloneable>` 擦除后为 `List< Serializable>`。

明白了这些擦除规则，再看如下代码：

```
public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add("abc");
    String str = list.get(0);
}
```

经过编译器的擦除处理后，上面的代码与下面的程序是一致的：

```
public static void main(String[] args) {
    List list = new ArrayList();
    list.add("abc");
    String str = (String)list.get(0);
}
```

Java 编译后的字节码中已经没有泛型的任何信息了，也就是说一个泛型类和一个普通类在经过编译后都指向了同一字节码，比如 `Foo<T>` 类，经过编译后将只有一份 `Foo.class` 类，不管是 `Foo<String>` 还是 `Foo<Integer>` 引用的都是同一字节码。Java 之所以如此处理，有两个原因：

- 避免 JVM 的大换血。C++ 的泛型生命周期延续到了运行期，而 Java 是在编译器擦除掉的，我们想想，如果 JVM 也把泛型类型延续到运行期，那么 JVM 就需要进行大量的重构工作了。
- 版本兼容。在编译期擦除可以更好地支持原生类型（Raw Type），在 Java 1.5 或 1.6 平台上，即使声明一个 `List` 这样的原生类型也是可以正常编译通过的，只是会产生警告信息而已。

明白了 Java 的泛型是类型擦除的，我们就可以解释类似如下的问题了：

(1) 泛型的 class 对象是相同的

每个类都有一个 `class` 属性，泛型化不会改变 `class` 属性的返回值，例如：

```
public static void main(String[] args) {
    List<String> ls = new ArrayList<String>();
    List<Integer> li = new ArrayList<Integer>();
    System.out.println(li.getClass() == li.getClass());
}
```

以上代码将返回为 `true`，原因很简单，`List<String>` 和 `List<Integer>` 擦除后的类型都是 `List`，没有任何区别。

(2) 泛型数组初始化时不能声明泛型类型

如下代码编译时通不过：

```
List<String>[] listArray = new List<String>[];
```

原因很简单，可以声明一个带有泛型参数的数组，但是不能初始化该数组，因为执行了类型擦除操作，List<Object>[] 与 List<String>[] 就是同一回事了，编译器拒绝如此声明。

(3) instanceof 不允许存在泛型参数

以下代码不能通过编译，原因一样，泛型类型被擦除了：

```
List<String> list = new ArrayList<String>();
System.out.println(list instanceof List<String>);
```

建议 94：不能初始化泛型参数和数组

泛型类型在编译期被擦除，我们在类初始化时将无法获得泛型的具体参数，比如这样的代码：

```
class Foo<T>{
    private T t = new T();
    private T[] tArray = new T[5];
    private List<T> list = new ArrayList<T>();
}
```

这段代码有什么问题呢？t、tArray、list 都是类变量，都是通过 new 声明了一个类型，看起来非常相似啊！但这段代码是编译通不过的，因为编译器在编译时需要获得 T 类型，但泛型在编译期类型已经被擦除了，所以 new T() 和 new T[5] 都会报错（可能有读者疑惑了：泛型类型可以擦除为顶级类 Object，那 T 类型擦除成 Object 不就可以编译了吗？这样也不行，泛型只是 Java 语言的一部分，Java 语言毕竟是一个强类型、编译型的安全语言，要确保运行期的稳定性和安全性就必须要求在编译器上严格检查）。可为什么 new ArrayList<T>() 却不会报错呢？

这是因为 ArrayList 表面是泛型，其实已经在编译期转型为 Object 了，我们来看一下 ArrayList 的源代码就清楚了，代码如下：

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    // 容纳元素的数组
    private transient Object[] elementData;
    // 构造函数
    public ArrayList() {
        this(10);
    }
    // 获得一个元素
    public E get(int index) {
        RangeCheck(index);
```

```

    // 返回前强制类型转换
    return (E) elementData[index];
}
}

```

注意看 `elementData` 的定义，它容纳了 `ArrayList` 的所有元素，其类型是 `Object` 数组，因为 `Object` 是所有类的父类，数组又允许协变（Covariant），因此 `elementData` 数组可以容纳所有的实例对象。元素加入时向上转型为 `Object` 类型（`E` 类型转为 `Object`），取出时向下转型为 `E` 类型（`Object` 转为 `E` 类型），如此处理而已。

在某些情况下，我们确实需要泛型数组，那该如何处理呢？代码如下：

```

class Foo<T>{
    // 不再初始化，由构造函数初始化
    private T t;
    private T[] tArray;
    private List<T> list= new ArrayList<T>();
    // 构造函数初始化
    public Foo(){
        try {
            Class<?> tType = Class.forName("");
            t = (T)tType.newInstance();
            tArray = (T[])Array.newInstance(tType,5);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

此时，运行就没有任何问题了。剩下的问题就是怎么在运行期获得 `T` 的类型，也就是 `tType` 参数，一般情况下泛型类型是无法获取的，不过，在客户端调用时多传输一个 `T` 类型的 `class` 就会解决问题。

类的成员变量是在类初始化前初始化的，所以要求在初始化前它必须具有明确的类型，否则就只能声明，不能初始化。

建议 95：强制声明泛型的实际类型

`Arrays` 工具类有一个方法 `asList` 可以把一个变长参数或数组转变为列表，但是它有一个缺点：它所生成的 `List` 长度是不可改变的，而这在我们的项目开发中有时会很不方便。如果你期望生成的列表长度是可变，那就需要自己来写一个数组的工具类了，代码如下：

```

class ArrayUtils{
    // 把一个变长参数转变为列表，并且长度可变
    public static <T> List<T> asList(T...t){
}

```

```

        List<T> list = new ArrayList<T>();
        Collections.addAll(list, t);
        return list;
    }
}

```

这很简单，与 `Arrays.asList` 的调用方式相同，我们传入一个泛型对象，然后返回相应的 `List`，代码如下：

```

public static void main(String[] args) {
    // 正常用法
    List<String> list1 = ArrayUtils.asList("A", "B");
    // 参数为空
    List list2 = ArrayUtils.asList();
    // 参数为数字和字符串的混合
    List list3 = ArrayUtils.asList(1, 2, 3.1);
}

```

这里有三个变量需要说明：

(1) 变量 list1

变量 `list1` 是一个常规用法，没有任何问题，泛型实际的参数类型是 `String`，返回的结果也就是一个容纳 `String` 元素的 `List` 对象。

(2) 变量 list2

变量 `list2` 中容纳的是什么元素呢？我们无法从代码中推断出 `list2` 列表到底容纳的是什么元素（因为它传递的参数是空，编译器也不知道泛型的实际参数类型是什么），不过，编译器会很“聪明”地推断出最顶层类 `Object` 就是其泛型类型，也就是说 `list2` 的完整定义如下：

```
List<Object> list2 = ArrayUtils.asList();
```

如此一来，编译时就不会给出“unchecked”警告了。现在新的问题出现了：如果期望 `list2` 是一个 `Integer` 类型的列表，而不是 `Object` 列表，因为后续的逻辑会把 `Integer` 类型加入到 `list2` 中，那该如何处理呢？

强制类型转化（把 `asList` 强制转换成 `List<Integer>`）？行不通，虽然 Java 的泛型是编译擦除式的，但是 `List<Object>` 和 `List<Integer>` 没有继承关系，不能进行强制转换。

重新声明一个 `List<Integer>`，然后读取 `List<Object>` 元素，一个一个地向下转型过去？麻烦，而且效率又低。

最好的解决方法是强制声明泛型类型，代码如下：

```
List<Integer> list2 = ArrayUtils.<Integer>asList();
```

就这么简单，`asList` 方法要求的是一个泛型参数，那我们就在输入前定义这是一个 `Integer` 类型的参数，当然，输出也是 `Integer` 类型的集合了。

(3) 变量 list3

变量 list3 有两种类型的元素：整数类型和浮点类型，那它生成的 List 泛型化参数应该是什么呢？是 Integer 和 Float 的父类 Number？你太高看编译器了，它不会如此推断的，当它发现多个元素的实际类型不一致时就会直接确认泛型类型是 Object，而不会去追索元素类的公共父类是什么，但是对于 list3，我们更期望它的泛型参数是 Number，都是数字嘛！参照 list2 变量，代码修改如下：

```
List<Number> list3 = ArrayUtils.<Number>asList(1, 2, 3.1);
```

Number 是 Integer 和 Float 的父类，先把三个输入参数向上转型为 Number，那么返回的结果也就是 List<Number> 类型了。

通过强制泛型参数类型，我们明确了泛型方法的输入、输出参数类型，问题是我们要在什么时候明确泛型类型呢？一句话：无法从代码中推断出泛型类型的情况下，即可强制声明泛型类型。

建议 96：不同的场景使用不同的泛型通配符

Java 泛型支持通配符（Wildcard），可以单独使用一个“？”表示任意类，也可以使用 extends 关键字表示某一个类（接口）的子类型，还可以使用 super 关键字表示某一个类（接口）的父类型，但问题是时候该用 extends，什么时候该用 super 呢？

(1) 泛型结构只参与“读”操作则限定上界（extends 关键字）

阅读如下代码，想想看我们的业务逻辑操作是否还能继续：

```
public static <E> void read(List<? super E> list) {
    for(Object obj:list){
        // 业务逻辑操作
    }
}
```

从 List 列表中读取元素的操作（比如一个数字列表中的求和计算），你觉得方法 read 能继续写下去吗？

答案是：不能，我们不知道 list 到底存放的是什么元素，只能推断出是 E 类型的父类（当然，也可以是 E 类型，下同，不再赘述），但问题是 E 类型的父类是什么呢？无法再推断，只有运行时才知道，那么编码期就完全无法操作了。当然，你可以把它当作是 Object 类来处理，需要时再转换成 E 类型——这完全违背了泛型的初衷。

在这种情况下，“读”操作如果期望从 List 集合中读取数据就需要使用 extends 关键字了，也就是要界定泛型的上界，代码如下：

```
public static <E> void read(List<? extends E> list){}
```

```

for(E e:list){
    // 业务逻辑处理
}
}

```

此时，已经推断出 List 集合中取出的是 E 类型的元素。具体是什么类型的元素就要等到运行时才能确定了，但它一定是一个确定的类型，比如 `read(Arrays.asList("A"))` 调用该方法时，可以推断出 List 中的元素类型是 String，之后就可以对 List 中的元素进行操作了，如加入到另外的 List<E> 集合中，或者作为 Map<E,V> 的键等。

(2) 泛型结构只参与“写”操作则限定下界（使用 super 关键字）

先看如下代码是否可以编译：

```

public static void write(List<? extends Number> list){
    // 加入一个元素
    list.add(123);
}

```

编译失败，失败的原因是 list 中的元素类型不确定，也就是编译器无法推断出泛型类型到底是什么，是 Integer 类型？是 Double？还是 Byte？这些都符合 extends 关键字的定义，由于无法确定实际的泛型类型，所以编译器拒绝了此类操作。

在此种情况下，只有一个元素是可以 add 进去的： null 值，这是因为 null 是一个万用类型，它可以是所有类的实例对象，所以可以加入到任何列表中。

Object 是否也可以？不可以，因为它不是 Number 子类，而且即使把 list 变量修改为 List<? extends Object> 类型也不能加入，原因很简单，编译器无法推断出泛型类型，加什么元素都是无效的。

在这种“写”操作的情况下，使用 super 关键字限定泛型类型的下界才是正道，代码如下：

```

public static void write(List<? super Number> list){
    list.add(123);
    list.add(3.14);
}

```

甭管它是 Integer 类型的 123，还是 Float 类型的 3.14，都可以加入到 list 列表中，因为它们都是 Number 类型，这就保证了泛型类的可靠性。

对于是要限定上界还是限定下界，JDK 的 Collections.copy 方法是一个非常好的例子，它实现了把源列表中的所有元素拷贝到目标列表中对应的索引位置上，代码如下：

```

public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    for (int i=0; i<srcSize; i++)
        dest.set(i, src.get(i));
}

```

源列表是用来提供数据的，所以 src 变量需要限定上界，带有 extends 关键字。目标列

表是用来写入数据的，所以 dest 变量需要界定上界，带有 super 关键字。

如果一个泛型结构即用作“读”操作又用作“写”操作，那该如何进行限定呢？不限定，使用确定的泛型类型即可，如 List<E>。

建议 97：警惕泛型是不能协变和逆变的

什么叫协变（covariance）和逆变（contravariance）？Wiki 上是这样定义的：

Within the type system of a programming language, covariance and contravariance refers to the ordering of types from narrower to wider and their interchangeability or equivalence in certain situations (such as parameters, generics, and return types).

在编程语言的类型框架中，协变和逆变是指宽类型和窄类型在某种情况下（如参数、泛型、返回值）替换或交换的特性，简单地说，协变是用一个窄类型替换宽类型，而逆变则是用宽类型覆盖窄类型。其实，在 Java 中协变和逆变我们已经用了很久了，只是我们没发觉而已，看如下代码：

```
class Base{
    public Number doStuff(){
        return 0;
    }
}

class Sub extends Base{
    @Override
    public Integer doStuff(){
        return 0;
    }
}
```

子类的 doStuff 方法返回值的类型比父类方法要窄，此时 doStuff 方法就是一个协变方法，同时根据 Java 的覆写定义来看，这又属于覆写。那逆变是怎么回事呢？代码如下：

```
class Base{
    public void doStuff(Integer i){
    }
}

class Sub extends Base{
    public void doStuff(Number n){
    }
}
```

子类的 doStuff 方法的参数类型比父类要宽，此时就是一个逆变方法，子类扩大了父类方法的输入参数，但根据覆写定义来看，doStuff 不属于覆写，只是重载而已。由于此时的

`doStuff` 方法已经与父类没有任何关系了，只是子类独立扩展出的一个行为，所以是否声明为 `doStuff` 方法名意义不大，逆变已经不具有特别的意义了，我们来重点关注一下协变，先看如下代码是否是协变：

```
public static void main(String[] args) {
    Base base = new Sub();
}
```

`base` 变量是否发生了协变？是的，发生了协变，`base` 变量是 `Base` 类型，它是父类，而其赋值却是子类实例，也就是用窄类型覆盖了宽类型。这也叫多态（Polymorphism），两者同含义，在 Java 世界里“重复发明”轮子的事情多了去了。

说了这么多，下面再来想想泛型是否也支持协变和逆变，答案是：泛型即不支持协变，也不支持逆变。很受伤是吧？为什么会不支持呢？

(1) 泛型不支持协变

数组和泛型很相似，一个是中括号，一个是尖括号，那我们就以数组为参照对象，看如下代码：

```
public static void main(String[] args) {
    // 数组支持协变
    Number[] n = new Integer[10];
    // 编译不通过，泛型不支持协变
    List<Number> ln = new ArrayList<Integer>();
}
```

`ArrayList` 是 `List` 的子类型，`Integer` 是 `Number` 的子类型，里氏替换原则（Liskov Substitution Principle）在此处行不通了，原因就是 Java 为了保证运行期的安全性，必须保证泛型参数类型是固定的，所以它不允许一个泛型参数可以同时包含两种类型，即使是父子类关系也不行。

泛型不支持协变，但可以使用通配符（Wildcard）模拟协变，代码如下所示：

```
//Number 的子类型（包括 Number 类型）都可以是泛型参数类型
List<? extends Number> ln = new ArrayList<Integer>();
```

“`? extends Number`” 表示的意思是，允许 `Number` 所有的子类（包括自身）作为泛型参数类型，但在运行期只能是一个具体类型，或者是 `Integer` 类型，或者是 `Double` 类型，或者是 `Number` 类型，也就是说通配符只是在编码期有效，运行期则必须是一个确定类型。

(2) 泛型不支持逆变

Java 虽然可以允许逆变存在，但在对类型赋值上是不允许逆变的，你不能把一个父类实例对象赋值给一个子类类型变量，泛型自然也不允许此种情况发生了，但是它可以使用 `super` 关键字来模拟实现，代码如下。

```
//Integer 的父类型（包括 Integer）都可以是泛型参数类型
List<? super Integer> li = new ArrayList<Number>();
```

“? super Integer”的意思是可以把所有 Integer 父类型（自身、父类或接口）作为泛型参数，这里看着就像是把一个 Number 类型的 ArrayList 赋值给了 Integer 类型的 List，其外观类似于使用一个宽类型覆盖一个窄类型，它模拟了逆变的实现。

泛型既不支持协变也不支持逆变，带有泛型参数的子类型定义与我们经常使用的类类型也不相同，其基本的类型关系如表 7-1 所示。

表 7-1 泛型通配符的 QA

问	答
Integer 是 Number 的子类型？	√
ArrayList<Integer> 是 List<Integer> 的子类型？	√
Integer[] 是 Number[] 的子类型？	√
List<Integer> 是 List<Number> 的子类型？	✗
List<Integer> 是 List<? extends Integer> 的子类型？	✗
List<Integer> 是 List<? super Integer> 的子类型？	✗

注意 Java 的泛型是不支持协变和逆变的，只是能够实现协变和逆变。

建议 98：建议采用的顺序是 List<T>、List<?>、List<Object>

List<T>、List<?>、List<Object> 这三者都可以容纳所有的对象，但使用的顺序应该是首选 List<T>，次之 List<?>，最后选择 List<Object>，原因如下：

(1) List<T> 是确定的某一个类型

List<T> 表示的是 List 集合中的元素都为 T 类型，具体类型在运行期决定；List<?> 表示的是任意类型，与 List<T> 类似，而 List<Object> 则表示 List 集合中的所有元素为 Object 类型，因为 Object 是所有类的父类，所以 List<Object> 也可以容纳所有的类类型，从这一字面上分析，List<T> 更符合习惯：编码者知道它是某一个类型，只是在运行期才确定而已。

(2) List<T> 可以进行读写操作

List<T> 可以进行诸如 add、remove 等操作，因为它的类型是固定的 T 类型，在编码期不需要进行任何的转型操作。

List<?> 是只读类型的，不能进行增加、修改操作，因为编译器不知道 List 中容纳的是什么类型的元素，也就无法校验类型是否安全了，而且 List<?> 读取出的元素都是 Object 类型的，需要主动转型，所以它经常用于泛型方法的返回值。注意，List<?> 虽然无法增加、修改元素，但是却可以删除元素，比如执行 remove、clear 等方法，那是因为它的删除动作与

泛型类型无关。

`List<Object>` 也可以读写操作，但是它执行写入操作时需要向上转型（Up cast），在读取数据后需要向下转型（Downcast），而此时已经失去了泛型存在的意义了。

打个比方，有一个篮子用来容纳物品，`List<T>` 的意思是说，“嘿，我这里有一个篮子，可以容纳固定类别的东西，比如西瓜、番茄等”。`List<?>` 的意思是说“嘿，我也有一个篮子，我可以容纳任何东西，只要是你想得到的”。而 `List<Object>` 就更有意思了，它说“嘿，我也有一个篮子，我可以容纳所有物质，只要你认为是物质的东西就都可以容纳进来”。

推而广之，`Dao<T>` 应该比 `Dao<?>`、`Dao<Object>` 更先采用，`Desc<Person>` 则比 `Desc<?>`、`Desc<Object>` 更优先采用。

建议 99：严格限定泛型类型采用多重界限

从哲学上来说，很难描述一个具体的人，你可以描述它的长相、性格、工作等，但是人都是有多重身份的，估计只有使用多个 And（与操作）将所有的描述串联起来才能描述一个完整的人，比如我，上班时我是一个职员，下班了坐公交车我是一个乘客，回家了我是父母的孩子，是儿子的父亲……角色时刻在变换。那如果我们要使用 Java 程序来对一类人进行管理，该如何做呢？比如在公交车费优惠系统中，对部分人员（如工资低于 2500 元的上班族并且是站立着的乘客）车费打 8 折，该如何实现呢？

注意这里的类型参数有两个限制条件：一为上班族；二为是乘客。具体到我们的程序中就应该是一个泛型参数具有两个上界（Upper Bound），首先定义两个接口及实现类，代码如下：

```
// 职员
interface Staff{
    // 工资
    public int getSalary();
}

// 乘客
interface Passenger{
    // 是否是站立状态
    public boolean isStanding();
}

// 定义“我”这个类型的人
class Me implements Staff,Passenger{
    public boolean isStanding(){
        return true;
    }
    public int getSalary() {
        return 2000;
    }
}
```

“Me”这种类型的人物有很多，比如做系统分析师也是一个职员，也坐公交车，但他的工资实现就和我不同，再比如 Boss 级的人物，偶尔也坐公交车，对大老板来说他也只是一个职员，他的实现类也不同，也就是说如果我们使用“T extends Me”是限定不了需求对象的，那该怎么办呢？可以考虑使用多重限定，代码如下：

```
// 工资低于 2500 元的上班族并且站立的乘客车票打 8 折
public static <T extends Staff & Passenger> void discount(T t) {
    if(t.getSalary()<2500 && t.isStanding()){
        System.out.println("恭喜你！您的车票打八折！");
    }
}

public static void main(String[] args) {
    discount(new Me());
}
```

使用“&”符号设定多重边界（Multi Bounds），指定泛型类型 T 必须是 Staff 和 Passenger 的共有子类型，此时变量 t 就具有了所有限定的方法和属性，要再进行判断就易如反掌了。

在 Java 的泛型中，可以使用“&”符号关联多个上界并实现多个边界限定，而且只有上界才有此限定，下界没有多重限定的情况。想想你就会明白：多个下界，编码者可自行推断出具体的类型，比如“? super Integer”和“? extends Double”，可以更细化为 Number 类型了，或者 Object 类型了，无须编译器推断了。

为什么要说明多重边界？是因为编码者太少使用它了，比如一个判断用户权限的方法，使用的是策略模式（Strategy Pattern），示意代码如下：

```
public class UserHandler<T extends User>{
    // 判断用户是否有权限执行操作
    public boolean permit(T user, List<Job> jobs) {
        List<Class<?>> iList = Arrays.asList(user.getClass().getInterfaces());
        // 判断是否是管理员
        if (iList.indexOf(Admin.class) > -1) {
            Admin admin = (Admin) user;
            /* 判断管理员是否有此权限 */
        } else {
            /* 判断普通用户是否有此权限 */
        }
        return false;
    }
}
```

此处进行了一次泛型参数类别判断，这里不仅仅违背了单一职责原则（Single Responsibility Principle），而且让“泛型”很汗颜：已经使用泛型限定参数的边界了，还要进行泛型类型判断。事实上，使用多重边界可以很方便地解决问题，而且非常优雅，建议读者在开发中考虑使用多重限定。

建议 100：数组的真实类型必须是泛型类型的子类型

List 接口的 toArray 方法可以把一个集合转化为数组，但是使用不方便， toArray() 方法返回的是一个 Object 数组，所以需要自行转变； toArray(T[] a) 虽然返回的是 T 类型的数组，但是还需要传入一个 T 类型的数组，这也挺麻烦的，我们期望输入的是一个泛型化的 List，这样就能转化为泛型数组了，来看看能不能实现，代码如下：

```
public static <T> T[] toArray(List<T> list) {
    T[] t = (T[]) new Object[list.size()];
    for(int i=0,n=list.size();i<n;i++) {
        t[i] = list.get(i);
    }
    return t;
}
```

上面把要输出的参数类型定义为 Object 数组，然后转型为 T 类型数组，之后遍历 List 赋值给数组的每个元素，这与 ArrayList 的 toArray 方法很类似（注意只是类似），客户端的调用如下：

```
public static void main(String[] args) {
    List<String> list = Arrays.asList("A", "B");
    for(String str:toArray(list)){
        System.out.println(str);
    }
}
```

编译没有任何问题，运行后出现如下异常：

```
Exception in thread "main" java.lang. ClassCastException: [Ljava.lang.Object;
cannot be cast to [Ljava.lang.String; at Client.main(Client.java:20)
```

类型转换异常，也就是说不能把一个 Object 数组转换为 String 数组，这段异常包含了两个问题：

□ 为什么 Object 数组不能向下转型为 String 数组？

数组是一个容器，只有确保容器内的所有元素类型与期望的类型有父子关系时才能转换， Object 数组只能保证数组内的元素是 Object 类型，却不能确保它们都是 String 的父类型或子类，所以类型转换失败。

□ 为什么是 main 方法抛出异常，而不是 toArray 方法？

其实，是在 toArray 方法中进行的类型向下转换，而不是 main 方法中。那为什么异常会在 main 方法中抛出，应该在 toArray 方法的 “T[] t = (T[])new Object[list.size()]” 这段代码才对呀？那是因为泛型是类型擦除的， toArray 方法经过编译后与如下代码相同：

```
public static Object[] toArray(List list){
```

```

// 此处的强制类型没必要存在，只是为了与源代码对比
Object[] t = (Object[])new Object[list.size()];
for(int i=0,n=list.size();i<n;i++){
    t[i] = list.get(i);
}
return t;
}

public static void main(String[] args) {
    List<String> list = Arrays.asList("A","B");
    for(String str:(String[])toArray(list)){
        System.out.println(str);
    }
}
}

```

阅读完此段代码就很清楚了：`toArray` 方法返回后会进行一次类型转换，`Object` 数组转换成了 `String` 数组，于是就报 `ClassCastException` 异常了。

`Object` 数组不能转为 `String` 数组，`T` 类型又无法在运行期获得，那该如何解决这个问题呢？其实，要想把一个 `Obejct` 数组转换为 `String` 数组，只要 `Object` 数组的实际类型（Actual Type）也是 `String` 就可以了，例如：

```

//objArray 的实际类型和表面类型都是 String 数组
Object[] objArray = {"A","B"};
// 抛出 ClassCastException
String[] strArray = (String[])objArray;

String[] ss = {"A","B"};
//objs 的真实类型是 String 数组，显示类型为 Object 数组
Object[] objs = ss;
// 顺利转换为 String 数组
String[] strs = (String[])objs;

```

明白了这个问题，我们就把泛型数组声明为泛型类的子类型吧！代码如下：

```

public static <T> T[] toArray(List<T> list, Class<T> tClass) {
    // 声明并初始化一个 T 类型的数组
    T[] t = (T[]) Array.newInstance(tClass, list.size());
    for(int i=0,n=list.size();i<n;i++){
        t[i] = list.get(i);
    }
    return t;
}

```

通过反射类 `Array` 声明了一个 `T` 类型的数组，由于我们无法在运行期获得泛型类型的参数，因此就需要调用者主动传入 `T` 参数类型。此时，客户端再调用就不会出现任何异常了。

在这里我们看到，当一个泛型类（特别是泛型集合）转变为泛型数组时，泛型数组的真实类型不能是泛型类型的父类型（比如顶层类 `Object`），只能是泛型类型的子类型（当然包

括自身类型), 否则就会出现类型转换异常。

建议 101：注意 Class 类的特殊性

Java 语言是先把 Java 源文件编译成后缀为 class 的字节码文件, 然后再通过 ClassLoader 机制把这些类文件加载到内存中, 最后生成实例执行的, 这是 Java 处理的基本机制, 但是加载到内存中的数据是如何描述一个类的呢? 比如在 Dog.class 文件中定义的是一个 Dog 类, 那它在内存中是如何展现的呢?

Java 使用一个元类 (MetaClass) 来描述加载到内存中的类数据, 这就是 Class 类, 它是一个描述类的类对象, 比如 Dog.class 文件加载到内存中后就会一个 Class 的实例对象描述之。因为 Class 类是“类中类”, 也就有预示着它有很多特殊的地方:

- 无构造函数。Java 中的类一般都有构造函数, 用于创建实例对象, 但是 Class 类却没有构造函数, 不能实例化, Class 对象是在加载类时由 Java 虚拟机通过调用类加载器中的 defineClass 方法自动构造的。
- 可以描述基本类型。虽然 8 个基本类型在 JVM 中并不是一个对象, 它们一般存在于栈内存中, 但是 Class 类仍然可以描述它们, 例如可以使用 int.class 表示 int 类型的类对象。
- 其对象都是单例模式。一个 Class 的实例对象描述一个类, 并且只描述一个类, 反过来也成立, 一个类只有一个 Class 实例对象, 如下代码返回的结果都为 true:

```
// 类的属性 class 所引用的对象与实例对象的 getClass 返回值相同
String.class.equals(new String().getClass())
"ABC".getClass().equals(String.class)
//class 实例对象不区分泛型
ArrayList.class.equals(new ArrayList<String>().getClass())
```

Class 类是 Java 的反射入口, 只有在获得了一个类的描述对象后才能动态地加载、调用, 一般获得一个 Class 对象有三种途径:

- 类属性方式, 如 String.class。
- 对象的 getClass 方法, 如 new String().getClass()。
- forName 方法加载, 如 Class.forName("java.lang.String")。

获得了 Class 对象后, 就可以通过 getAnnotations() 获得注解, 通过 getMethods() 获得方法, 通过 getConstructors() 获得构造函数等, 这为后续的反射代码铺平了道路。

建议 102：适时选择 getDeclared××× 和 get×××

Java 的 Class 类提供了很多的 getDeclared××× 方法和 get××× 方法, 例如 getDeclaredMethod 和 getMethod 成对出现, getDeclaredConstructors 和 getConstructors 也是成对出现, 那这

两者之间有什么差别呢？看如下代码：

```
public static void main(String[] args) throws Exception {
    // 方法名称
    String methodName = "doStuff";
    Method m1 = Foo.class.getDeclaredMethod(methodName);
    Method m2 = Foo.class.getMethod(methodName);
}
// 静态内部类
static class Foo {
    void doStuff() {}
}
```

此段代码运行后的输出如下：

```
Exception in thread "main" java.lang.NoSuchMethodException: Client$Foo.doStuff()
at java.lang.Class.getMethod(Class.java:1605)
at Client.main(Client.java:10)
```

该异常是说 m2 变量的 getDeclaredMethod 方法没有找到 doStuff 方法，明明有这个方法呀，为什么没有找到呢？这是因为 getMethod 方法获得的是所有 public 访问级别的方法，包括从父类继承的方法，而 getDeclaredMethod 获得是自身类的所有方法，包括公用（public）方法、私有（private）方法等，而且不受限于访问权限。

其他的 getDeclaredConstructors 和 getConstructors、getDeclaredFields 和 getFields 等与此相似。Java 之所以如此处理，是因为反射本意只是正常代码逻辑的一种补充，而不是让正常代码逻辑产生翻天覆地的变动，所以 public 的属性和方法最容易获取，私有属性和方法也可以获取，但要限定本类。

那现在问题来了：如果需要列出所有继承自父类的方法，该如何实现呢？简单，先获得父类，然后使用 getDeclaredMethods，之后持续递归即可。

建议 103：反射访问属性或方法时将 Accessible 设置为 true

Java 中通过反射执行一个方法的过程如下：获取一个方法对象，然后根据 isAccessible 返回值确定是否能够执行，如果返回值为 false 则需要调用 setAccessible (true)，最后再调用 invoke 执行方法，具体如下：

```
Method method = ....;
// 检查是否可以访问
if(!method.isAccessible()){
    method.setAccessible(true);
}
// 执行方法
method.invoke(obj, args);
```

此段代码已经成为了习惯用法：通过反射方式执行方法时，必须在 invoke 之前检查 Accessible 属性。这是一个好习惯，也确实应该如此，但方法对象的 Accessible 属性并不是用来决定是否可访问的，看如下代码：

```
public class Foo {
    public final void doStuff() {
        System.out.println("Do Stuff....");
    }
}
```

定义一个 public 类的 public 方法，这是一个没有任何限制的方法，按照我们对 Java 语言的理解，此时 doStuff 方法可以被任何一个类访问。我们编写一个客户端类来检查该方法是否可以反射执行：

```
public static void main(String[] args) throws Exception {
    // 反射获取方法
    Method m1 = Foo.class.getMethod("doStuff");
    // 打印出是否可访问
    System.out.println("Accessible = "+m1.isAccessible());
    // 执行方法
    m1.invoke(new Foo());
}
```

很简单的反射操作，获得一个方法，然后检查是否可以访问，最后执行方法输出。让我们来猜想一下结果：因为 Foo 类是 public 的，方法也是 public，全部都是最开放的访问权限，那么 Accessible 也应该等于 true。但是运行结果却是：

```
Accessible = false
Do Stuff....
```

为什么 Accessible 属性会等于 false？而且等于 false 了还能执行？这是因为 Accessible 的属性并不是我们语法层级理解的访问权限，而是指是否更容易获得，是否进行安全检查。

我们知道，动态修改一个类或方法或执行方法时都会受 Java 安全体系的制约，而安全的处理是非常消耗资源的（性能非常低），因此对于运行期要执行的方法或要修改的属性就提供了 Accessible 可选项：由开发者决定是否要逃避安全体系的检查。

阅读源代码是理解的最好方式，我们来看 AccessibleObject 类的源代码，它提供了取消默认访问控制检查的功能。首先查看 isAccessible 方法，代码如下：

```
public class AccessibleObject implements AnnotatedElement {
    // 定义反射的默认操作权限: suppressAccessChecks
    static final private java.security.Permission ACCESS_PERMISSION = new Reflect-
        Permission("suppressAccessChecks");
    // 是否重置了安全检查，默认是 false
    boolean override;
    // 默认构造函数
```

```

protected AccessibleObject() {}
// 是否可以快速获取，默认是不能
public boolean isAccessible() {
    return override;
}
}
}

```

AccessibleObject 是 Field、Method、Constructor 的父类，决定其是否可以快速访问而不进行访问控制检查，在 AccessibleObject 类中是以 override 变量保存该值的，但是具体是否快速执行是在 Method 类的 invoke 方法中决定的，代码如下：

```

public Object invoke(Object obj, Object... args) throws ... {
    // 检查是否可以快速获取，其值是父类 AccessibleObject 的 override 变量
    if (!override) {
        // 不能快速获取，要进行安全检查
        if (!Reflection.quickCheckMemberAccess(...)) {
            .....
            Reflection.ensureMemberAccess(...);
            .....
        }
    }
    // 直接执行方法
    return methodAccessor.invoke(obj, args);
}
}

```

看了这段代码，诸位就很清楚了：Accessible 属性只是用来判断是否需要进行安全检查的，如果不需要则直接执行，这就可以大幅度地提升系统性能（当然了，由于取消了安全检查，也可以运行 private 方法、访问 private 私有属性了）。经过测试，在大量的反射情况下，设置 Accessible 为 true 可以提升性能 20 倍以上。

AccessibleObject 的其他两个子类 Field 和 Constructor 与 Method 的情形相似：Accessible 属性决定 Field 和 Constructor 是否受访问控制检查。我们在设置 Field 或执行 Constructor 时，务必要设置 Accessible 为 true，这并不仅仅是因为操作习惯的问题，还是在为我们系统的性能考虑。

注意 对于我们已经“习惯”了的代码，多思考一下为什么。

建议 104：使用 `forName` 动态加载类文件

动态加载（Dynamic Loading）是指在程序运行时加载需要的类库文件，对 Java 程序来说，一般情况下，一个类文件在启动时或首次初始化时会被加载到内存中，而反射则可以在运行时再决定是否要加载一个类，比如从 Web 上接收一个 String 参数作为类名，然后在 JVM 中加载并初始化，这就是动态加载，此动态加载通常是通过 Class.forName(String) 实现

的，只是这个 `forName` 方法到底是什么意思呢？

我们知道一个类文件只有在被加载到内存中后才可能生成实例对象，也就是说一个对象的生成必然会经过以下两个步骤：

- 加载到内存中生成 Class 的实例对象。
- 通过 `new` 关键字生成实例对象。

如果我们使用的是 `import` 关键字产生的依赖包，JVM 在启动时会自动加载所有依赖包下的类文件，这没有什么问题，如果要动态加载类文件，就要使用 `forName` 方法了，但问题是为什么我们要使用 `forName` 方法动态加载一个类文件呢？那是因为我们不知道生成的实例对象是什么类型（如果知道就不用动态加载），而且方法和属性都不可访问呀。问题又来了：动态加载的意义在什么地方呢？

意义在于：加载一个类即表示要初始化该类的 `static` 变量，特别是 `static` 代码块，在这里我们可以做大量的工作，比如注册自己，初始化环境等，这才是我们要重点关注的逻辑，例如如下代码：

```
class Utils{
    // 静态代码块
    static{
        System.out.println("Do Something");
    }
}

public class Client {
    public static void main(String[] args) throws Exception{
        // 动态加载
        Class.forName("Utils");
    }
}
```

注意看 `Client` 类，我们并没有对 `Utils` 做任何初始化，只是通过 `forName` 方法加载了 `Utils` 类，但是却产生了一个“`Do Something`”的输出，这就是因为 `Utils` 类被加载后，JVM 会自动初始化其 `static` 变量和 `static` 代码块，这是类加载机制所决定的。

对于此种动态加载，最经典的应用就是数据库驱动程序的加载片段，代码如下：

```
// 加载驱动
Class.forName("com.mysql.jdbc.Driver");
String url="jdbc:mysql://localhost:3306/db?user=&password=";
Connection conn=DriverManager.getConnection(url);
Statement stmt=conn.createStatement();
.....
```

在没有 `Hibernate` 和 `Ibatis` 等 ORM 框架的情况下，基本上每个系统都会有这么一个 JDBC 连接类，然后提供诸如 `Query`、`Delete` 等的方法，大家有没有想过为什么要加上

`forName` 这句话呢？没有任何的输出呀，要它干什么用呢？事实上非常有用，我们看一下 `Driver` 类的源码：

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    // 静态代码块
    static {
        try {
            // 把自己注册到 DriverManager 中
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            // 异常处理
        }
    }
    // 构造函数
    public Driver() throws SQLException {
    }
}
```

该程序的逻辑是这样的：数据库的驱动程序已经由 `NonRegisteringDriver` 实现了，`Driver` 类只是负责把自己注册到 `DriverManager` 中。当程序动态加载该驱动时，也就是执行到 `Class.forName("com.mysql.jdbc.Driver")` 时，`Driver` 类会被加载到内存中，于是 `static` 代码块开始执行，也就是把自己注册到 `DriverManager` 中。

需要说明的是，`forName` 只是把一个类加载到内存中，并不保证由此产生一个实例对象，也不会执行任何方法，之所以会初始化 `static` 代码，那是由类加载机制所决定的，而不是 `forName` 方法决定的。也就是说，如果没有 `static` 属性或 `static` 代码块，`forName` 就只是加载类，没有任何的执行行为。

注意 `forName` 只是加载类，并不执行任何代码。

建议 105：动态加载不适合数组

上一个建议解释了为什么要使用 `forName`，本建议就来说说哪些地方不适合使用动态加载。如果 `forName` 要加载一个类，那它首先必须是一个类——8 个基本类型排除在外，它们不是一个具体的类；其次，它必须具有可追索的类路径，否则就会报 `ClassNotFoundException`。

在 Java 中，数组是一个非常特殊的类，虽然它是一个类，但没有定义类路径，例如这样的代码：

```
public static void main(String[] args) throws Exception{
    String[] strs = new String[10];
    Class.forName("java.lang.String[]");
}
```

`String[]` 是一个类型声明，它作为 `forName` 的参数应该也是可行的吧！但是非常遗憾，其运行结果如下：

```
Exception in thread "main" java.lang.ClassNotFoundException: java/lang/String[]
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Class.java:169)
at Client.main(Client.java:6)
```

产生 `ClassNotFoundException` 异常的原因是数组虽然是一个类，在声明时可以定义为 `String[]`，但编译器编译后会为不同的数组类型生成不同的类，具体如表 7-2 所示。

表 7-2 数组编译对应关系表

元素类型	编译后的类型
<code>byte[]</code>	[B]
<code>char[]</code>	[C]
<code>Double[]</code>	[D]
<code>Float[]</code>	[F]
<code>Int[]</code>	[I]
<code>Long[]</code>	[J]
<code>Short[]</code>	[S]
<code>Boolean</code>	[Z]
引用类型（如 <code>String[]</code> ）	[L 引用类型（如：[Ljava.lang.String;）]

在编码期，我们可以声明一个变量为 `String[]`，但是经过编译器编译后就成为了 `[Ljava.lang.String`。明白了这一点，再根据以上的表格可知，动态加载一个对象数组只要加载编译后的数组对象就可以了，代码如下：

```
// 加载一个 String 数组
Class.forName("[Ljava.lang.String;");
// 加载一个 long 数组
Class.forName("[J");
```

虽然以上代码可以动态加载一个数组类，但是这没有任何意义，因为它不能生成一个数组对象，也就是说以上代码只是把一个 `String` 类型的数组类和 `long` 类型的数组类加载到了内存中（如果内存中没有该类的话），并不能通过 `newInstance` 方法生成一个实例对象，因为它没有定义数组的长度，在 Java 中数组是定长的，没有长度的数组是不允许存在的。

既然反射不能定义一个数组，那问题就来了：如何动态加载一个数组呢？比如依据输入动态产生一个数组。其实可以使用 `Array` 数组反射类来动态加载，代码如下：

```
// 动态创建数组
String[] strs = (String[]) Array.newInstance(String.class, 8);
// 创建一个多维数组
int[][] ints = (int[][]) Array.newInstance(int.class, 2, 3);
```

因为数组比较特殊，要想动态创建和访问数组，基本的反射是无法实现的，“上帝对你关闭一扇门，同时会为你打开另外一扇窗”，于是 Java 就专门定义了一个 `Array` 数组反射工具类来实现动态探知数组的功能。

注意 通过反射操作数组使用 `Array` 类，不要采用通用的反射处理 API。

建议 106：动态代理可以使代理模式更加灵活

Java 的反射框架提供了动态代理（Dynamic Proxy）机制，允许在运行期对目标类生成代理，避免重复开发。我们知道一个静态代理是通过代理主题角色（Proxy）和具体主题角色（Real Subject）共同实现抽象主题角色（Subject）的逻辑的，只是代理主题角色把相关的执行逻辑委托给了具体主题角色而已，一个简单的静态代理如下所示：

```
// 抽象主题角色
interface Subject {
    // 定义一个方法
    public void request();
}

// 具体主题角色
class RealSubject implements Subject {
    // 实现方法
    public void request() {
        // 业务逻辑处理
    }
}

// 代理主题角色
class Proxy implements Subject {
    // 要代理哪个实现类
    private Subject subject = null;
    // 默认被代理者
    public Proxy(){
        subject = new RealSubject();
    }
    // 通过构造函数传递被代理者
    public Proxy(Subject _subject ){
        subject = _subject;
    }
    // 实现接口中定义的方法
    public void request() {
        before();
        subject.request();
        after();
    }
    // 预处理
    private void before(){
    }
}
```

```
        //do something  
    }  
    // 善后处理  
    private void after(){  
        //do something  
    }  
}
```

这是一个简单的静态代理。Java 还提供了 `java.lang.reflect.Proxy` 用于实现动态代理：只要提供一个抽象主题角色和具体主题角色，就可以动态实现其逻辑的，其示例代码如下：

```
// 抽象主题角色
interface Subject {
    // 定义一个方法
    public void request();
}

// 具体主题角色
class RealSubject implements Subject {
    // 实现方法
    public void request() {
        // 业务逻辑处理
    }
}

class SubjectHandler implements InvocationHandler {
    // 被代理的对象
    private Subject subject;

    public SubjectHandler(Subject _subject) {
        subject = _subject;
    }

    // 委托处理方法
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        // 预处理
        System.out.println(" 预处理 ");
        // 直接调用被代理类的方法
        Object obj = method.invoke(subject, args);
        // 后处理
        System.out.println(" 后处理 ");
        return obj;
    }
}
```

注意看，这里没有了代理主题角色，取而代之的是 `SubjectHandler` 作为主要的逻辑委托处理，其中 `invoke` 方法是接口 `InvocationHandler` 定义必须实现的，它完成了对真实方法的调用。

我们来详细了解一下 `InvocationHanlder` 接口，动态代理是根据被代理的接口生成所有方法的，也就是说给定一个（或多个）接口，动态代理会宣称“我已经实现该接口下的所有方法”。

法了”，那各位读者想想看，动态代理怎么才能实现代理接口中的方法呢？在默认情况下所有方法的返回值都是空的，是的，虽然代理已经实现了它，但是没有任何的逻辑含义，那怎么办？好办，通过 `InvocationHandler` 接口的实现类来实现，所有方法都是由该 Handler 进行处理的，即所有被代理的方法都由 `InvocationHandler` 接管实际的处理任务。

我们接着来看动态代理的场景类，代码如下：

```
public static void main(String[] args) {
    // 具体主题角色，也就是被代理类
    Subject subject = new RealSubject();
    // 代理实例的处理 Handler
    InvocationHandler handler = new SubjectHandler(subject);
    // 当前加载器
    ClassLoader cl = subject.getClass().getClassLoader();
    // 动态代理
    Subject proxy = (Subject) Proxy.newProxyInstance(cl, subject.getClass().
        getInterfaces(), handler);
    // 执行具体主题角色方法
    proxy.request();
}
```

此时就实现了不用显式创建代理类即实现代理的功能，例如可以在被代理角色执行前进行权限判断，或者执行后进行数据校验。

动态代理很容易实现通用的代理类，只要在 `InvocationHandler` 的 `invoke` 方法中读取持久化数据即可实现，而且还能实现动态切入的效果，这也是 AOP (Aspect Oriented Programming) 编程理念。

建议 107：使用反射增加装饰模式的普适性

装饰模式（Decorator Pattern）的定义是“动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比于生成子类更为灵活”，不过，使用 Java 的动态代理也可以实现装饰模式的效果，而且其灵活性、适应性都会更强。

我们以卡通片《猫和老鼠》（《Tom and Jerry》）为例，看看如何包装小 Jerry 让它更强大。首先定义 Jerry 的类：老鼠（Rat 类），代码如下：

```
interface Animal {
    public void doStuff();
}
// 老鼠是一种动物
class Rat implements Animal {
    public void doStuff() {
        System.out.println("Jerry will play with Tom.");
    }
}
```

接下来我们要给 Jerry 增加一些能力，比如飞行、钻地等能力，当然使用类继承也很容易实现，但我们这里只是临时地为 Rat 类增加这些能力，使用装饰模式更符合此处的场景。首先定义装饰类，代码如下：

```
// 定义某种能力
interface Feature {
    // 加载特性
    public void load();
}

// 飞行能力
class FlyFeature implements Feature {
    public void load() {
        System.out.println("增加一只翅膀.....");
    }
}

// 钻地能力
class DigFeature implements Feature {
    public void load() {
        System.out.println("增加钻地能力.....");
    }
}
```

此处定义了两种能力：一种是飞行，另一种是钻地，我们如果把这两种属性赋予到 Jerry 身上，那就需要一个包装动作类了，代码如下：

```
class DecorateAnimal implements Animal {
    // 被包装的动物
    private Animal animal;
    // 使用哪一个包装器
    private Class<? extends Feature> clz;
    public DecorateAnimal(Animal _animal, Class<? extends Feature> _clz) {
        animal = _animal;
        clz = _clz;
    }
    @Override
    public void doStuff() {
        InvocationHandler handler = new InvocationHandler(){
            // 具体包装行为
            public Object invoke(Object p, Method m, Object[] args) throws
                Throwable {
                Object obj = null;
                // 设置包装条件
                if(Modifier.isPublic(m.getModifiers())){
                    obj = m.invoke(clz.newInstance(), args);
                }
                animal.doStuff();
                return obj;
            }
        };
    }
}
```

```

        // 当前加载器
        ClassLoader cl = getClass().getClassLoader();
        // 动态代理，由 Handler 决定如何包装
        Feature proxy = (Feature) Proxy.newProxyInstance(cl, clz.
            getInterfaces(), handler);
        proxy.load();
    }
}

```

注意看 doStuff 方法，一个装饰类型必然是抽象构建（Component）的子类型，它必须要实现 doStuff，此处的 doStuff 方法委托给了动态代理执行，并且在动态代理的控制器 Handler 中还设置了决定装饰方式和行为的条件（即代码中 InvocationHandler 匿名类中的 if 判断语句），当然，此处也可以通过读取持久化数据的方式进行判断，这样就更加灵活了。

抽象构件有了，装饰类也有了，装饰动作类也完成了，那我们就可以编写客户端进行调用了，代码如下：

```

public static void main(String[] args) throws Exception {
    // 定义 Jerry 这只家喻户晓的老鼠
    Animal Jerry = new Rat();
    // 为 Jerry 增加飞行能力
    Jerry = new DecorateAnimal(Jerry, FlyFeature.class);
    // Jerry 增加挖掘能力
    Jerry = new DecorateAnimal(Jerry, DigFeature.class);
    // Jerry 开始耍猫了
    Jerry.doStuff();
}

```

此类代码是一个比较通用的装饰模式，只需要定义被装饰的类及装饰类即可，装饰行为由动态代理实现，实现了对装饰类和被装饰类的完全解耦，提供了系统的扩展性。

建议 108：反射让模板方法模式更强大

模板方法模式（Template Method Pattern）的定义是：定义一个操作中的算法骨架，将一些步骤延迟到子类中，使子类不改变一个算法的结构即可重定义该算法的某些特定步骤。简单地说，就是父类定义抽象模板作为骨架，其中包括基本方法（是由子类实现的方法，并且在模板方法被调用）和模板方法（实现对基本方法的调度，完成固定的逻辑），它使用了简单的继承和覆写机制，我们来看一个基本的例子。

我们经常会开发一些测试或演示程序，期望系统在启动时自行初始化，以方便测试或讲解，一般的做法是写一个 SQL 文件，在系统启动前手动导入，不过，这样不仅麻烦而且还容易出现错误，于是我们就自己动手写了一个初始化数据的框架：在系统（或容器）启动时自行初始化数据。但问题是每个应用程序要初始化的内容我们并不知道，只能由实现者自行编

写，那我们就必须给作者预留接口，此时就得考虑使用模板方法模式了，代码如下：

```
public abstract class AbsPopulator {
    // 模板方法
    public final void dataInitialing() throws Exception {
        // 调用基本方法
        doInit();
    }
    // 基本方法
    protected abstract void doInit();
}
```

这里定义了一个抽象模板类 AbsPopulator，它负责数据初始化，但是具体要初始化哪些数据则是由 doInit 方法决定的，这是一个抽象方法，子类必须实现，我们来看一个用户表数据的加载：

```
public class UserPopulator extends AbsPopulator {
    protected void doInit() {
        /* 初始化用户表，如创建、加载数据等 */
    }
}
```

该系统在启动时，查找所有的 AbsPopulator 实现类，然后 dataInitialing 实现数据的初始化。那读者可能要想了，怎么让容器知道这个 AbsPopulator 类呢？很简单，如果是使用 Spring 作为 IoC 容器的项目，直接在 dataInitialing 方法上加上 @PostConstruct 注解，Spring 容器启动完毕后会自动运行 dataInitialing 方法，由于这里的原理超出了本书的范畴，不再赘述。

现在的问题是：初始化一张 User 表需要非常多的操作，比如先建表，然后筛选数据，之后插入，最后校验，如果把这些都放到一个 doInit 方法里会非常庞大（即使提炼出多个方法承担不同的职责，代码的可读性依然很差），那该如何做呢？又或者 doInit 是没有任何的业务意义的，是否可以起一个优雅而又动听的名字呢？

答案是我们可以使用反射增强模板方法模式，使模板方法实现对一批固定规则的基本方法的调用。代码是最好的交流语言，我们看看怎么改造 AbsPopulator 类，代码如下：

```
public abstract class AbsPopulator {
    // 模板方法
    public final void dataInitialing() throws Exception {
        // 获得所有的 public 方法
        Method[] methods = getClass().getMethods();
        for (Method m : methods) {
            // 判断是否是数据初始化方法
            if (isInitDataMethod(m)) {
                m.invoke(this);
            }
        }
    }
}
```

```
// 判断是否是数据初始化方法，基本方法鉴别器
private boolean isInitDataMethod(Method m) {
    return m.getName().startsWith("init") // init 开始
    && Modifier.isPublic(m.getModifiers()) // 公开方法
    && m.getReturnType().equals(Void.TYPE) // 返回值是 void
    && !m.isVarArgs() // 输入参数为空
    && !Modifier.isAbstract(m.getModifiers()); // 不能是抽象方法
}
}
```

在一般的模板方法模式中，抽象模板（这里是 AbsPopulator 类）需要定义一系列的基本方法，一般都是 protected 访问级别的，并且是抽象方法，这标志着子类必须实现这些基本方法，这对子类来说既是一个约束也是一个负担。但是使用了反射后，不需要定义任何抽象方法，只需定义一个基本方法鉴别器（例子中 isInitDataMethod）即可加载符合规则的基本方法。鉴别器在此处的作用是鉴别子类方法中哪些是基本方法，模板方法（例子中的 dataInitializing）则根据基本方法鉴别器返回的结果通过反射执行相应的方法。

此时，如果需要进行大量的数据初始化工作，子类的实现就非常简单了，代码如下：

```
public class UserPopulator extends AbsPopulator {
    public void initUser() {
        /* 初始化用户表，如创建、加载数据等 */
    }
    public void initPassword(){
        /* 初始化密码 */
    }
    public void initJobs(){
        /* 初始化工作任务 */
    }
}
```

UserPopulator 类中的方法只要符合基本方法鉴别器条件即会被模板方法调用，方法的数据量也不再受父类的约束，实现了子类灵活定义基本方法、父类批量调用的功能，并且缩减了子类的代码量。

如果读者熟悉 JUnit 的话，就会看出此处的实现与 JUnit 非常类似，JUnit4 之前要求测试的方法名必须是以 test 开头的，并且无返回值、无参数，而且有 public 修饰，其实现的原理与此非常相似，读者有兴趣可以看看 JUnit 的源代码。

注意 决定使用模板方法模式时，请尝试使用反射方式实现，它会让你的程序更灵活、更强大。

建议 109：不需要太多关注反射效率

反射的效率是一个老生常谈的问题，有“经验”的开发人员经常使用这句话恐吓新

人：反射的效率是非常低的，不到万不得已就不要使用。事实上，这句话的前半句是对的，后半句是错的。

反射的效率相对于正常的代码执行确实低很多（经过测试，相差 15 倍左右），但是它是一个非常有效的运行期工具类，只要代码结构清晰、可读性好那就先开发起来，等到进行性能测试时证明此处性能确实有问题时再修改也不迟（一般情况下反射并不是性能的终极杀手，而代码结构混乱、可读性差则很可能会埋下性能隐患）。我们看这样一个例子：在运行期获得泛型类的泛型类型，代码如下：

```
class Utils {
    // 获得一个泛型类的实际泛型类型
    public static <T> Class<T> getGenricClassType(Class clz) {
        Type type = clz.getGenericSuperclass();
        if (type instanceof ParameterizedType) {
            ParameterizedType pt = (ParameterizedType) type;
            Type[] types = pt.getActualTypeArguments();
            if (types.length > 0 && types[0] instanceof Class) {
                // 若有多个泛型参数，依据位置索引返回
                return (Class) types[0];
            }
        }
        return (Class) Object.class;
    }
}
```

前面我们讲过，Java 的泛型类型只存在于编译期，那为什么这个工具类可以取得运行期的泛型类型呢？那是因为该工具只支持继承的泛型类，如果是在 Java 编译时已经确定了泛型类的类型参数，那当然可以通过泛型获得了。例如有这样一个泛型类：

```
abstract class BaseDao<T> {
    // 获得 T 的运行期类型
    private Class<T> clz = Utils.getGenricClassType(getClass());
    // 根据主键获得一条记录
    public void get(long id) {
        session.get(clz, id);
    }
}
// 操作 user 表
class UserDao extends BaseDao<String> {
```

对于 UserDao 类，编译器编译时已经明确了其参数类型是 String，因此可以通过反射的方式来获取其类型，这也是 getGenricClassType 方法使用的场景。

BaseDao 和 UserDao 是 ORM 中的常客，BaseDao 实现对数据库的基本操作，比如增删改查，而 UserDao 则是一个比较具体的数据库操作，其作用是对 User 表进行操作，如果 BaseDao 能够提供足够多的基本方法，比如单表的增删改查，那些与 UserDao 类似的

BaseDao 子类就可以省去大量的开发工作。但问题是持久层的 session 对象（这里模拟的是 Hibernate Session）需要明确一个具体的类型才能操作，比如 get 查询，需要获得两个参数：实体类类型（用于确定映射的数据表）和主键，主键好办，问题是实体类类型怎么获得呢？

子类自行传递？麻烦，而且也容易产生错误。

读取配置问题？可行，但效率不高。

最好的办法就是父类泛型化，子类明确泛型参数，然后通过反射读取相应的类型即可，于是就有了我们代码中的 `clz` 变量：通过反射获得泛型类型。如此实现后，`UserDao` 可以不用定义任何方法，继承过来的父类操作方法已经能满足基本需求了，这样代码结构清晰，可读性又好。我已将这种方式使用在多个项目中了，目前没有出现因为该反射引起的性能问题。

想想看，如果考虑反射效率问题，没有 `clz` 变量，不使用反射，每个 `BaseDao` 的子类都要实现一个查询操作，代码将会大量重复，违反了“Don't Repeat Yourself”这条最基本的编码规则，这会致使项目重构、优化难度加大，代码的复杂度也会提高很多。

对于反射效率问题，不要做任何的提前优化和预期，这基本上是杞人忧天，很少有项目是因为反射问题引起系统效率故障的（除非是拷贝工的垃圾代码，这不在我们的讨论范围之内），而且根据二八原则，80% 的性能消耗在 20% 的代码上，这 20% 的代码才是我们关注的重点，不要单单把反射作为重点关注对象。

注意 反射效率低是个真命题，但因为这一点而不使用它就是个假命题。



大成若缺，其用不弊。

大盈若冲，其用不穷。

——老子《道德经》

不管人类的思维有多么缜密，也存在“智者千虑必有一失”的缺憾。无论计算机技术怎么发展，也不可能穷尽所有的情景——这个世界是不完美的，是有缺陷的，完美的世界只存在于理想中。

对于软件帝国的缔造者来说，程序也是不完美的，异常情况随时都会出现，我们需要它为我们描述例外事件，需要它处理非预期的情景，需要它帮我们建立“完美世界”。

建议 110：提倡异常封装

Java 语言的异常处理机制可以确保程序的健壮性，提高系统的可用率，但是 Java API 提供的异常都是比较低级的（这里的低级是指“低级别”的异常），只有开发人员才能看得懂，才明白发生了什么问题。而对于终端用户来说，这些异常基本上就是天书，与业务无关，是纯计算机语言的描述，那该怎么办？这就需要我们对异常进行封装了。异常封装有三方面的优点：

（1）提高系统的友好性

例如，打开一个文件，如果文件不存在，则会报 FileNotFoundException 异常，如果该方法的编写者不做任何处理，直接抛到上层，则会降低系统的友好性，代码如下所示：

```
public static void doStuff() throws Exception {
    InputStream is = new FileInputStream("无效文件.txt");
    /* 文件操作 */
}
```

此时 doStuff 方法的友好性极差：出现异常时（比如文件不存在），该方法会直接把 FileNotFoundException 异常抛出到上层应用中（或者是最终用户），而上层应用（或用户）要么自己处理，要么接着抛，最终的结果就是让用户面对着“天书”式的文字发呆，用户不知道这是什么问题，只是知道系统告诉他“哦，我出错了，什么错误？你自己看着办吧”。

解决办法就封装异常，可以把异常的阅读者分为两类：开发人员和用户。开发人员查找问题，需要打印出堆栈信息，而用户则需要了解具体的业务原因，比如文件太大、不能同时编写文件等，代码如下：

```
public static void doStuff2() throws MyBussinessException{
    try {
        InputStream is = new FileInputStream("无效文件.txt");
    } catch (FileNotFoundException e) {
        // 为方便开发和维护人员而设置的异常信息
        e.printStackTrace();
        // 抛出业务异常
        throw new MyBussinessException(e);
    }
}
```

(2) 提高系统的可维护性

来看如下代码：

```
public void doStuff() {
    try{
        //do something
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

这是很多程序员容易犯的错误，抛出异常是吧？分类处理多麻烦，就写一个 catch 块来处理所有的异常吧，而且还信誓旦旦地说“JVM 会打印出栈中的出错信息”，虽然这没错，但是该信息只有开发人员自己才看得懂，维护人员看到这段异常时基本上无法处理，因为需要深入到代码逻辑中去分析问题。

正确的做法是对异常进行分类处理，并进行封装输出，代码如下：

```
public void doStuff(){
    try{
        //do something
    }catch(FileNotFoundException e){
        log.info("文件未找到，使用默认配置文件.....");
    }catch(SecurityException e){
        log.error("无权访问，可能原因是.....");
        e.printStackTrace();
    }
}
```

如此包装后，维护人员看到这样的异常就有了初步的判断，或者检查配置，或者初始化环境，不需要直接到代码层级去分析了。

(3) 解决 Java 异常机制自身的缺陷

Java 中的异常一次只能抛出一个，比如 doStuff 方法有两个逻辑代码片段，如果在第一个逻辑片段中抛出异常，则第二个逻辑片段就不再执行了，也就无法抛出第二个异常了，现在的问题是：如何才能一次抛出两个（或多个）异常呢？

其实，使用自行封装的异常可以解决该问题，代码如下：

```
class MyException extends Exception {
    // 容纳所有的异常
    private List<Throwable> causes = new ArrayList<Throwable>();
    // 构造函数，传递一个异常列表
    public MyException(List<? extends Throwable> _causes) {
        causes.addAll(_causes);
    }
    // 读取所有的异常
}
```

```

public List<Throwable> getExceptions() {
    return causes;
}
}
}

```

MyException 异常只是一个异常容器，可以容纳多个异常，但它本身并不代表任何异常含义，它所解决的是一次抛出多个异常的问题，具体调用如下：

```

public static void doStuff() throws MyException {
    List<Throwable> list = new ArrayList<Throwable>();
    // 第一个逻辑片段
    try {
        // Do Something
    } catch (Exception e) {
        list.add(e);
    }
    // 第二个逻辑片段
    try {
        // Do Something
    } catch (Exception e) {
        list.add(e);
    }
    // 检查是否有必要抛出异常
    if (list.size() > 0) {
        throw new MyException(list);
    }
}
}

```

这样一来，doStuff 方法的调用者就可以一次获得多个异常了，也能够为用户提供完整的例外情况说明。可能有读者会问：这种情况可能出现吗？怎么会要求一个方法抛出多个异常呢？

绝对可能出现，例如 Web 界面注册时，展现层依次把 User 对象传递到逻辑层，Register 方法需要对各个 Field 进行校验并注册，例如用户名不能重复，密码必须符合密码策略等，不要出现用户第一次提交时系统提示“用户名重复”，在用户修改用户名再次提交后，系统又提示“密码长度少于 6 位”的情况，这种操作模式下的用户体验非常糟糕，最好的解决办法就是封装异常，建立异常容器，一次性地对 User 对象进行校验，然后返回所有的异常。

建议 111：采用异常链传递异常

设计模式中有一个模式叫做责任链模式（Chain of Responsibility），它的目的是将多个对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止，异常的传递处理也应该采用责任链模式。

上一个建议中我们提出了异常需要封转，但仅仅封转还是不够的，还需要传递异常。我

们知道，一个系统友好性的标志是用户对该系统的“粘性”，粘性越高，系统越友好，粘性越低系统友好性越差，那问题是怎么提高系统的“粘性”呢？友好的界面和功能是一个方面，另外一方面就是系统出现非预期情况时的处理方式了。

比如，我们的 JEE 项目一般都有三层结构：持久层、逻辑层、展现层，持久层负责与数据库交互，逻辑层负责业务逻辑的实现，展现层负责 UI 数据的处理。有这样一个模块：用户第一次访问的时候，需要持久层从 user.xml 中读取信息，如果该文件不存在则提示用户创建之，那问题来了：如果我们直接把持久层的异常 FileNotFoundException 抛弃掉，逻辑层根本无从得知发生了何事，也就不能为展现层提供一个友好的处理结果了，最终倒霉的就是展现层：没有办法提供异常信息，只能告诉用户说“出错了，我也不知道出什么错了”——毫无友好性可言。

正确的做法是先封装，然后传递，过程如下：

- (1) 把 FileNotFoundException 封装为 MyException。
- (2) 抛出到逻辑层，逻辑层根据异常代码（或者自定义的异常类型）确定后续处理逻辑，然后抛出到展现层。
- (3) 展现层自行决定要展现什么，如果是管理员则可以展现低层级的异常，如果是普通用户则展示封装后的异常。

明白了异常为什么要传递，那接着的问题就是如何传递了。很简单，使用异常链进行异常的传递，我们以 IOException 为例来看看是如何传递的，代码如下：

```
public class IOException extends Exception {
    // 定义异常原因
    public IOException(String message) {
        super(message);
    }
    // 定义异常原因，并携带原始异常
    public IOException(String message, Throwable cause) {
        super(message, cause);
    }
    // 保留原始异常信息
    public IOException(Throwable cause) {
        super(cause);
    }
}
```

在 IOException 的构造函数中，上一个层级的异常可以通过异常链进行传递，链中传递异常的代码如下所示：

```
try{
    //Do Something
}catch(Exception e){
    throw new IOException(e);
}
```

捕捉到 Exception 异常，然后把它转化为 IOException 异常并抛出（此种方式也叫作异常转译），调用者获得该异常后再调用 getCause 方法即可获得 Exception 的异常信息，如此即可方便地查找到产生异常的根本信息，便于解决问题。

结合上一个建议来看，异常需要封装和传递，我们在进行系统开发时不要“吞噬”异常，也不要“赤裸裸”地抛出异常，封装后再抛出，或者通过异常链传递，可以达到系统更健壮、友好的目的。

建议 112：受检异常尽可能转化为非受检异常

为什么说是“尽可能”的转化呢？因为“把所有的受检异常（Checked Exception）都转化为非受检异常（Unchecked Exception）”这一想法是不现实的：受检异常是正常逻辑的一种补偿处理手段，特别是对可靠性要求比较高的系统来说，在某些条件下必须抛出受检异常以便由程序进行补偿处理，也就是说受检异常有合理的存在理由，那为什么要把受检异常转化为非受检异常呢？难道说受检异常有什么缺陷或不足吗？是的，受检异常确实有不足的地方：

（1）受检异常使接口声明脆弱

OOP（Object Oriented Programming，面向对象程序设计）要求我们尽量多地面向接口编程，可以提高代码的扩展性、稳定性等，但是一旦涉及异常问题就不一样了，例如系统初期是这样设计一个接口的：

```
interface User{
    // 修改用户名密码，抛出安全异常
    public void changePassword() throws MySecurityException;
}
```

随着系统的开发，User 接口有了多个实现者，比如普通的用户 UserImpl、模拟用户 MockUserImpl（用作测试或系统管理）、非实体用户 NonUserImpl（如自动执行机、逻辑处理器等），此时如果发现 changePassword 方法可能还需要抛出 RejectChangeException（拒绝修改异常，如自动执行机正在处理任务时不能修改其密码），那就需要修改 User 接口了：changePassword 方法增加抛出 RejectChangeException 异常，这会导致所有的 User 调用者都要追加对 RejectChangeException 异常问题的处理。

这里产生了两个问题：一是异常是主逻辑的补充逻辑，修改一个补充逻辑，就会导致主逻辑也被修改，也就是出现了实现类“逆影响”接口的情景，我们知道实现类是不稳定的，而接口是稳定的，一旦定义了异常，则增加了接口的不稳定性，这是对面向对象设计的严重亵渎；二是实现的类变更最终会影响到调用者，破坏了封装性，这也是迪米特法则所不能容忍的。

(2) 受检异常使代码的可读性降低

一个方法增加了受检异常，则必须有一个调用者对异常进行处理，比如无受检异常方法 doStuff 是这样调用的：

```
public static void main(String[] args) {
    doStuff();
}
```

doStuff 方法一旦增加受检异常就不一样了，代码如下：

```
public static void main(String[] args) {
    try {
        doStuff();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

doStuff 方法增加了 throws Exception，调用者就必须至少增加 4 条语句来处理该异常，代码膨胀许多，可读写性也降低了，特别是在多个异常需要捕捉的情况下，多个 catch 块多个异常处理，而且还可能在 catch 块中再次抛出异常，这大大降低了代码的可读性。

(3) 受检异常增加了开发工作量

我们知道，异常需要封装和传递，只有封装才能让异常更容易理解，上层模块才能更好的处理，可这也也会导致低层级的异常没玩没了的封装，无端加重了开发的工作量。比如 FileNotFoundException 在持久层抛出，就需要定义一个 MyException 进行封装，并抛出到上一个层级，于是增加了开发工作量。

受检异常有这么多的缺点，那有没有什么办法可以避免或减少这些缺点呢？有，很简单的一个规则：将受检异常转化为非受检异常即可，但是我们也不能把所有的受检异常转化为非受检异常，原因是在编码期上层模块不知道下层模块会抛出何种非受检异常，只有通过规则或文档来约束，可以这样说：

- 受检异常提出的是“法律下的自由”，必须遵守异常的约定才能自由编写代码。
- 非受检异常则是“协约性质的自由”，你必须告诉我你要抛出什么异常，否则不会处理。

以 User 接口为例，我们在声明接口时不再声明异常，而是在具体实现时根据不同的情况产生不同的非受检异常，这样持久层和逻辑层抛出的异常将会由展现层自行决定如何展示，不再受异常的规则约束了，大大简化开发工作，提高了代码的可读性。

那问题又来了：在开发和设计时什么样的受检异常有必要转化为非受检异常呢？“尽可能”是以什么作为判断依据呢？受检异常转换为非受检异常是需要根据项目的场景来决定的，例如同样是刷卡，员工拿着自己的工卡到考勤机上打考勤，此时如果附近有磁性物质干

扰，则考勤机可以把这种受检异常转化为非受检异常，黄灯闪烁后不做任何记录登记，因为考勤失败这种情景不是“致命”的业务逻辑，出错了，重新刷一下即可。但是到银行网点取钱就不一样了，拿着银行卡到银行取钱，同样有磁性物质干扰，刷不出来，那这种异常就必须登记处理，否则会成为威胁银行卡安全的事件。汇总成一句话：当受检异常威胁到了系统的安全性、稳定性、可靠性、正确性时，则必须处理，不能转化为非受检异常，其他情况则可以转换为非受检异常。

注意 受检异常威胁到系统的安全性、稳定性、可靠性、正确性时，不能转换为非受检异常。

建议 113：不要在 finally 块中处理返回值

在 finally 代码块中处理返回值，这是考试和面试中经常出现的题目。虽然可以以此来出考试题，但在项目中绝对不能在 finally 代码块中出现 return 语句，这是因为这种处理方式非常容易产生“误解”，会严重误导开发者。例如如下代码：

```
public static void main(String[] args) {
    try {
        doStuff(-1);
        doStuff(100);
    } catch (Exception e) {
        System.out.println("这里是永远都不会到达的");
    }
}
// 该方法抛出受检异常
public static int doStuff(int _p) throws Exception {
    try {
        if (_p < 0) {
            throw new DataFormatException("数据格式错误");
        } else {
            return _p;
        }
    } catch (Exception e) {
        // 异常处理
        throw e;
    } finally {
        return -1;
    }
}
```

对于这段代码，有两个问题：main 方法中的 doStuff 方法的返回值是什么？doStuff 方法永远都不会抛出异常吗？

答案是：doStuff(-1) 的值是 -1，doStuff(100) 的值也是 -1，调用 doStuff 方法永远都不会抛出异常，有这么神奇？原因就是我们在 finally 代码块中加入了 return 语句，而这会导致

出现以下两个问题：

(1) 覆盖了 try 代码块中的 return 返回值

当执行 doStuff(-1) 时，doStuff 方法产生了 DataFormatException 异常，catch 块在捕捉此异常后直接抛出，之后代码执行到 finally 代码块，就会重置返回值，结果就是 -1 了，也就是出现了先返回，再执行 finally，再重置返回值的情况。

有读者可能会扩展思考了：是不是可以定义一个变量，在 finally 中修改后再 return 呢？代码如下：

```
public static int doStuff() {
    int a = 1;
    try {
        return a;
    } catch (Exception e) {
    } finally {
        // 重新修改一下返回值
        a = -1;
    }
    return 0;
}
```

该方法的返回值永远是 1，而不会是 -1 或 0（为什么不会执行到“return 0”呢？原因是 finally 执行完毕后该方法已经有返回值了，后续代码就不会再执行了），这都是源于异常代码块的处理方式，在代码中加上 try 代码块就标志着运行时会有一个 Throwable 线程监视着该方法的运行，若出现异常，则交由异常逻辑处理。

我们知道方法是在栈内存中运行的，并且会按照“先进后出”的原则执行，main 方法调用了 doStuff 方法，则 main 方法在下层，doStuff 在上层，当 doStuff 方法执行完“return a”时，此方法的返回值已经确定是 in 类型 1 (a 变量的值，注意基本类型都是值拷贝，而不是引用)，此后 finally 代码块再修改 a 的值已经与 doStuff 返回者没有任何关系了，因此该方法永远都会返回 1。

继续追问：那是不是可以在 finally 代码块中修改引用类型的属性以达到修改返回值的效果呢？代码如下：

```
public static Person doStuff() {
    Person person = new Person();
    person.setName("张三");
    try {
        return person;
    } catch (Exception e) {
    } finally {
        // 重新修改一下返回值
        person.setName("李四");
    }
}
```

```

        person.setName("王五");
        return person;
    }

    class Person{
        private String name;
        /*name 的 getter/setter 方法省略 */
    }
}

```

此方法的返回值永远都是 name 为李四的 Person 对象，原因是 Person 是一个引用对象，在 try 代码块中的返回值是 Person 对象的地址，finally 中再修改那当然会是李四了。

(2) 屏蔽异常

为什么明明把异常 throw 出去了，但 main 方法却捕捉不到呢？这是因为异常线程在监视到有异常发生时，就会登记当前的异常类型为 DataFormatException，但是当执行器执行 finally 代码块时，则会重新为 doStuff 方法赋值，也就是告诉调用者“该方法执行正确，没有产生异常，返回值是 1”，于是乎，异常神奇的消失了，其简化代码如下所示：

```

public static void doSomething() {
    try {
        // 正常抛出异常
        throw new RuntimeException();
    } finally {
        // 告诉 JVM: 该方法正常返回
        return;
    }
}

public static void main(String[] args) {
    try {
        doSomething();
    } catch (RuntimeException e) {
        System.out.println("这里永远都不会到达！");
    }
}

```

上面 finally 代码块中的 return 已经告诉 JVM：doSomething 方法正常执行结束，没有异常，所以 main 方法就不可能获得任何异常信息了。这样的代码会使可读性大大降低，读者很难理解作者的意图，增加了修改的难度。

在 finally 中处理 return 返回值，代码看上去很完美，都符合逻辑，但是执行起来就会产生逻辑错误，最重要的一点是 finally 是用来做异常的收尾处理的，一旦加上了 return 语句就会让程序的复杂度徒然提升，而且会产生一些隐蔽性非常高的错误。

与 return 语句相似，System.exit(0) 或 Runtime.getRuntime().exit(0) 出现在异常代码块中也会产生非常多的错误假象，增加代码的复杂性，读者有兴趣可以自行研究一下。

注意 不要在 finally 代码块中出现 return 语句。

建议 114：不要在构造函数中抛出异常

Java 的异常机制有三种：

- Error 类及其子类表示的是错误，它是不需要程序员处理也不能处理的异常，比如 VirtualMachineError 虚拟机错误，ThreadDeath 线程僵死等。
- RuntimeException 类及其子类表示的是非受检异常，是系统可能会抛出的异常，程序员可以去处理，也可以不处理，最经典就是 NullPointerException 空指针异常和 IndexOutOfBoundsException 越界异常。
- Exception 类及其子类（不包含非受检异常）表示的是受检异常，这是程序员必须处理的异常，不处理则程序不能通过编译，比如 IOException 表示 I/O 异常，SQLException 表示数据库访问异常。

我们知道，一个对象的创建要经过内存分配、静态代码初始化、构造函数执行等过程，对象生成的关键步骤是构造函数，那是不是也允许在构造函数中抛出异常呢？从 Java 语法上来说，完全可以在构造函数中抛出异常，三类异常都可以，但是从系统设计和开发的角度来分析，则尽量不要在构造函数中抛出异常，我们以三种不同类型的异常来说明之。

（1）构造函数抛出错误是程序员无法处理的

在构造函数执行时，若发生了 VirtualMachineError 虚拟机错误，那就没招了，只能抛出，程序员不能预知此类错误的发生，也就不能捕捉处理。

（2）构造函数不应该抛出非受检异常

我们来看这样一个例子，代码如下：

```
class Person{
    public Person(int _age){
        // 不满 18 岁的用户对象不能建立
        if(_age<18){
            throw new RuntimeException(" 年龄必须大于 18 岁。 ");
        }
    }
    // 看限制级的电影
    public void seeMovie(){
        System.out.println(" 看限制级电影 ");
    }
}
```

这段代码的意图很明显，年龄不满 18 岁的用户根本不会生成一个 Person 实例对象，没有对象，类行为 seeMovie 方法就不可执行，想法很好，但这会导致不可预测的结果，比如我们这样引用 Person 类。

```

public static void main(String[] args) {
    Person p = new Person(17);
    p.seeMovie();
    /* 其他的逻辑处理 */
}

```

很显然，`p` 对象不能建立，因为是一个 `RuntimeException` 异常，开发人员可以捕捉也可以不捕捉，代码看上去逻辑很正确，没有任何瑕疵，但是事实上，这段程序会抛出异常，无法执行。这段代码给了我们两个警示：

□ 加重了上层代码编写者的负担

捕捉这个 `RuntimeException` 异常吧，那谁来告诉我有这个异常呢？只有通过文档来约束了，一旦 `Person` 类的构造函数经过重构后再抛出其他非受检异常，那 `main` 方法不用修改也是可以通过测试的，但是这里就可能会产生隐藏的缺陷，而且还是很难重现的缺陷。

不捕捉这个 `RuntimeException` 异常，这是我们通常的想法，既然已经写成了非受检异常，`main` 方法的编码者完全可以不处理这个异常嘛，大不了不执行 `Person` 的方法！这是非常危险的，一旦产生异常，整个线程都再继续执行，或者连接没有关闭，或者数据没有写入数据库，或者产生内存异常，这些都是会对整个系统产生影响。

□ 后续代码不会执行

`main` 方法的实现者原本只是想把 `p` 对象的建立作为其代码逻辑的一部分，执行完 `seeMovie` 方法后还需要完成其他逻辑，但是因为没有对非受检异常进行捕捉，异常最终会抛出到 JVM 中，这会导致整个线程执行结束后，后面所有的代码都不会继续执行了，这就对业务逻辑产生了致命的影响。

(3) 构造函数尽可能不要抛出受检异常

我们来看下面的例子，代码如下：

```

// 父类
class Base{
    // 父类抛出 IOException
    public Base() throws IOException{
        throw new IOException();
    }
}
// 子类
class Sub extends Base{
    // 子类抛出 Exception 异常
    public Sub() throws Exception {
    }
}

```

就这么一段简单的代码，展示了在构造函数中抛出受检异常的三个不利方面：

□ 导致子类代码膨胀

在我们的例子中子类的无参构造函数不能省略，原因是父类的无参构造函数抛出了

IOException 异常，子类的无参构造函数默认调用的是父类的构造函数，所以子类的无参构造也必须抛出 IOException 或其父类。

□ 违背了里氏替换原则

里氏替换原则是说“父类能出现的地方子类就可以出现，而且将父类替换为子类也不会产生任何异常”，那我们回过头来看看 Sub 类是否可以替换 Base 类，比如我们的上层代码是这样写的：

```
public static void main(String[] args) {
    try{
        Base base = new Base();
    }catch(IOException e){
        // 异常处理
    }
}
```

然后，我们期望把 new Base() 替换成 new Sub()，而且代码能够正常编译和运行。非常可惜，编译通不过，原因是 Sub 的构造函数抛出了 Exception 异常，它比父类的构造函数抛出的异常范围要宽，必须增加新的 catch 块才能解决。

可能有读者要问了，为什么 Java 的构造函数允许子类的构造函数抛出更广泛的异常类呢？这正好与类方法的异常机制相反，类方法的异常是这样要求的：

```
// 父类
class Base{
    // 父类方法抛出 Exception
    public void method() throws Exception{
    }
}
// 子类
class Sub extends Base{
    // 子类方法的异常类型必须是父类方法的子类型
    @Override
    public void method() throws IOException{
    }
}
```

子类的方法可以抛出多个异常，但都必须是被覆写方法的子类型，对我们的例子来说，Sub 类的 method 方法抛出的异常必须是 Exception 的子类或 Exception 类，这是 Java 覆写的要求。构造函数之所以与此相反，是因为构造函数没有覆写的概念，只是构造函数间的引用调用而已，所以在构造函数中抛出受检异常会违背里氏替换原则，使我们的程序缺乏灵活性。

□ 子类构造函数扩展受限

子类存在的原因就是期望实现并扩展父类的逻辑，但是父类构造函数抛出异常却会让子类构造函数的灵活性大大降低，例如我们期望这样的构造函数。

```

class Sub extends Base{
    public Sub() throws Exception {
        try{
            super();
        }catch(IOException e){
            // 异常处理后再抛出
            throw e;
        }finally{
            // 收尾处理
        }
    }
}

```

很不幸，这段代码编译通不过，原因是构造函数 Sub 中没有把 super() 放在第一句话中，想把父类的异常重新包装后再抛出是不可行的（当然，这里有很多种“曲线”的实现手段，比如重新定义一个方法，然后父子类的构造函数都调用该方法，那么子类构造函数就可以自由处理异常了），这是 Java 语法限制。

将以上三种异常类型汇总起来，对于构造函数，错误只能抛出，这是程序员无能为力的事情；非受检异常不要抛出，抛出了“对己对人”都是有害的；受检异常尽量不抛出，能用曲线的方式实现就用曲线方式实现，总之一句话：在构造函数中尽可能不出现异常。

注意 在构造函数中不要抛出异常，尽量曲线救国。

建议 115：使用 Throwable 获得栈信息

AOP 编程可以很轻松地控制一个方法调用哪些类，也能够控制哪些方法允许被调用，一般来说切面编程（比如 AspectJ）只能控制到方法级别，不能实现代码级别的植入（Weave），比如一个方法被类 A 的 m1 方法调用时返回 1，在类 B 的 m2 方法调用时返回 0（同参数情况下），这就要求被调用者具有识别调用者的能力。在这种情况下，可以使用 Throwable 获得栈信息，然后鉴别调用者并分别输出，代码如下：

```

class Foo{
    public static boolean m(){
        // 取得当前栈信息
        StackTraceElement[] sts = new Throwable().getStackTrace();
        // 检查是否是 m1 方法调用
        for(StackTraceElement st:sts){
            if(st.getMethodName().equals("m1")){
                return true;
            }
        }
        return false;
    }
}

```

```

    }
    // 调用者
    class Invoker{
        // 该方法打印出 true
        public static void m1(){
            System.out.println(Foo.m());
        }
        // 该方法打印出 false
        public static void m2(){
            System.out.println(Foo.m());
        }
    }
}

```

注意看 Invoker 类，两个方法 m1 和 m2 都调用了 Foo 的 m 方法，都是无参调用，返回值却不同，这是我们的 Throwable 类发挥效能了。JVM 在创建一个 Throwable 类及其子类时会把当前线程的栈信息记录下来，以便在输出异常时准确定位异常原因，我们来看 Throwable 源代码：

```

public class Throwable implements Serializable {
    // 出现异常的栈记录
    private StackTraceElement[] stackTrace;
    // 默认构造函数
    public Throwable() {
        // 记录栈帧
        fillInStackTrace();
    }
    // 本地方法，抓取执行时的栈信息
    public synchronized native Throwable fillInStackTrace();
}

```

在出现异常时（或主动声明一个 Throwable 对象时），JVM 会通过 fillInStackTrace 方法记录下栈帧信息，然后生成一个 Throwable 对象，这样我们就可以知道类间的调用顺序、方法名称及当前行号等了。

获得栈信息可以对调用者进行判断，然后决定不同的输出，比如我们的 m1 和 m2 方法，同样是输入参数，同样的调用方法，但是输出却不同，这看起来很像是一个 Bug：方法 m1 调用 m 方法是正常显示，而方法 m2 调用却会返回“错误”数据。因此我们虽然可以依据调用者不同产生不同的逻辑，但这仅局限在对此方法的广泛认知上。更多的时候我们使用 m 方法的变形体，代码如下：

```

class Foo {
    public static boolean m() {
        // 取得当前栈信息
        StackTraceElement[] sts = new Throwable().getStackTrace();
        // 检查是否是 m1 方法调用
        for (StackTraceElement st : sts) {
            if (st.getMethodName().equals("m1")) {

```

```
        return true;
    }
}
throw new RuntimeException("除 m1 方法外，该方法不允许其他方法调用 ");
}
```

只是把“`return false`”替换成一个运行期异常，除了`m1`方法外，其他方法调用都会产生异常，该方法常用作离线注册码校验，当破解者试图暴力破解时，由于主执行者不是期望的值，因此会返回一个经过包装和混淆的异常信息，大大增加了破解的难度。

建议 116：异常只为异常服务

异常只为异常服务，这是何解？难道异常还能为其他服务不成？确实能，异常原本是正常逻辑的一个补充，但是有时候会被当作主逻辑使用，看如下代码：

```
// 判断一个枚举是否包含 String 枚举项
public static <T extends Enum<T>> boolean Contain(Class<T> c, String name) {
    boolean result = false;
    try {
        Enum.valueOf(c, name);
        result = true;
    } catch (RuntimeException e) {
        // 只要是抛出异常，则认为是不包含
    }
    return result;
}
```

判断一个枚举是否包含指定的枚举项，这里会根据 `valueOf` 方法是否抛出异常来进行判断，如果抛出异常（一般是 `IllegalArgumentException` 异常），则认为是不包含，若不抛出异常则可以认为包含该枚举项，看上去这段代码很正常，但是其中却有三个错误：

- 异常判断降低了系统性能。
 - 降低了代码的可读性，只有详细了解 `valueOf` 方法的人才能读懂这样的代码，因为 `valueOf` 抛出的是一个非受检异常。
 - 隐藏了运行期可能产生的错误，`catch` 到异常，但没有做任何处理。

我们这段代码是用一段异常实现了一个正常的业务逻辑，这导致代码产生了坏味道。要解决此问题也很容易，即不在主逻辑中使用异常，代码如下：

```
// 判断一个枚举是否包含 String 枚举项
public static <T extends Enum<T>> boolean Contain(Class<T> c, String name) {
    // 遍历枚举项
    for(T t:c.getEnumConstants()){
        // 枚举项名称是否相等
    }
}
```

```

        if(t.name().equals(name)) {
            return true;
        }
    }
    return false;
}

```

异常只能用在非正常的情况下，不能成为正常情况的主逻辑，也就是说，异常只是主场景中的辅助场景，不能喧宾夺主。

而且，异常虽然是描述例外事件的，但能避免则避免之，除非是确实无法避免的异常，例如：

```

public static void main(String[] args) {
    File file = new File("文件.txt");
    try {
        FileInputStream fis = new FileInputStream(file);
        /* 其他业务逻辑处理 */
    } catch (FileNotFoundException e) {
        // 异常处理
    }
}

```

这样一段代码经常会在我们的项目中出现，但经常写并不代表不可优化，这里的异常类 `FileNotFoundException` 完全可以在它诞生前就消除掉：先判断文件是否存在，然后再生成 `FileInputStream` 对象，代码如下：

```

public static void main(String[] args) {
    File file = new File("文件.txt");
    // 经常出现的异常情况，可以先做判断
    if (file.exists() && !file.isDirectory()) {
        try {
        } catch () {
        }
    }
}

```

虽然增加了 `if` 判断语句，增加了代码量，但是却会减少 `FileNotFoundException` 异常出现的几率，提高了程序的性能和稳定性。

注意 异常只为确实异常的事件服务。

建议 117：多使用异常，把性能问题放一边

我们知道异常是主逻辑的例外逻辑，举个简单例子来说，比如我在马路上走（这是主

逻辑），突然开过一辆车，我要避让（这是受检异常，必须处理），继续走着，突然一架飞机从我头顶飞过（非受检异常），我可以选择继续行走（不捕捉），也可以选择指责其噪音污染（捕捉，主逻辑的补充处理），再继续走着，突然一颗流星砸下来，这没有选择，属于错误，不能做任何处理。这样具备完整例外情景的逻辑就具备了 OO 的味道，任何一个事物的处理都可能产生非预期结果，问题是需要以何种手段来处理，如果不使用异常就需要依靠返回值的不同来进行处理了，这严重失去了面向对象的风格。

我们在编写用例文档（Use Case Specification）时，其中有一项叫作“例外事件”，是用来描述主场景外的例外场景的，例如用户登录的用例，就会在“例外事件”中说明“连续 3 次登录失败即锁定用户账号”，这就是登录事件的一个异常处理，具体到我们的程序中就是：

```
public void login(){
    try{
        // 正常登录
    }catch(InvalidLoginException lie){
        // 用户名无效
    }catch(InvalidPsswordException pe){
        // 密码错误的异常
    }
    }catch(TooMuchLoginException tmle){
        // 多次登录失败的异常
    }
}
```

如此设计则可以让我们的 login 方法更符合实际的处理逻辑，同时使主逻辑（正常登录，try 代码块）更加清晰。当然了，使用异常还有很多优点，比如可让正常代码和异常代码分离、能快速查找问题（栈信息快照）等，但是异常有一个缺点：性能比较慢。

Java 的异常处理机制确实比较慢，这个“比较慢”是相对于诸如 String、Integer 等对象来说的，单单从对象的创建上来说，new 一个 IOException 会比 String 慢 5 倍，这从异常的处理机制上也可以解释：因为它要执行 fillInStackTrace 方法，要记录当前栈的快照，而 String 类则是直接申请一个内存创建对象，异常类慢一筹也就在所难免了。

而且，异常类是不能缓存的，期望预先建立大量的异常对象以提高异常性能也是不现实的。

难道异常的性能问题就没有任何可提高的办法了？确实没有，但是我们不能因为性能问题而放弃使用异常，而且经过测试，在 JDK 1.6 下，一个异常对象创建的时间只需要 1.4 毫秒左右（注意是毫秒，通常一个交易处理是在 100 毫秒左右），难道我们的系统连如此微小的性能消耗都不允许吗？

注意 性能问题不是拒绝异常的借口。



第9章

多线程和并发

We're here to put a dent in the universe.Otherwise why else even be here?

活着就是为了改变世界，难道还有其他原因吗？

——Steve Paul Jobs (史蒂夫·乔布斯)

多线程技术可以更好地利用系统资源，减少对用户的响应时间，提高系统的性能和效率，但同时也增加了系统的复杂性和运维难度，特别是在高并发、大压力、高可靠性的项目中，线程资源的同步、抢占、互斥等都需要慎重考虑，以避免产生性能损耗和线程死锁。

建议 118：不推荐覆写 start 方法

多线程比较简单的实现方式是继承 Thread 类，然后覆写 run 方法，在客户端程序中通过调用对象的 start 方法即可启动一个线程，这是多线程程序的标准写法。不知道读者是否还能回想起自己的第一个多线程 demo 呢？估计一般是这样写的：

```
class MultiThread extends Thread{
    @Override
    public void start(){
        // 调用线程体
        run();
    }

    @Override
    public void run(){
        //MultiThread do something.
    }
}
```

覆写 run 方法，这好办，写上自己的业务逻辑即可，但为什么要覆写 start 方法呢？最常见的理由是：要在客户端中调用 start 方法启动线程，不覆写 start 方法怎么启动 run 方法呢？于是乎就覆写了 start 方法，在方法内调用 run 方法。客户端代码是一个标准程序，代码如下：

```
public static void main(String[] args) {
    // 多线程对象
    MultiThread multiThread = new MultiThread();
    // 启动多线程
    multiThread.start();
}
```

相信读者都能看出这是一个错误的多线程应用，main 方法根本就没有启动一个子线程，整个应用程序中只有一个主线程在运行，并不会创建任何其他的线程。对此，有很简单的解决办法，只要删除 MultiThread 类中的 start 方法即可。

然后呢？就结束了吗？是的，很多时候确实到此结束了。找我解惑的同事或朋友中，很少有人会问为什么不必而且不能覆写 start 方法，仅仅就是因为“多线程应用就是这样写的”这个原因吗？

要说明这个问题，就需要看一下 Thread 类的源代码了。Thread 类的 start 方法的代码如下。

```

public synchronized void start() {
    // 判断线程状态，必须是未启动状态
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    // 加入线程组中
    group.add(this);
    // 分配栈内存，启动线程，运行 run 方法
    start0();
    // 在启动前设置了停止状态
    if (stopBeforeStart) {
        stop0(throwableFromStop);
    }
}
// 本地方法
private native void start0();

```

这里的关键是本地方法 start0，它实现了启动线程、申请栈内存、运行 run 方法、修改线程状态等职责，线程管理和栈内存管理都是由 JVM 负责的，如果覆盖了 start 方法，也就是撤消了线程管理和栈内存管理的能力，这样如何启动一个线程呢？事实上，不需要关注线程和栈内存的管理，只需要编码者实现多线程的逻辑即可（即 run 方法体），这也是 JVM 比较聪明的地方，简化多线程应用。

那可能有读者要问了：如果确实有必要覆写 start 方法，那该如何处理呢？这确实是一个罕见的要求，不过，要覆写也很容易，只要在 start 方法中加上 super.start 即可，代码如下：

```

class MultiThread extends Thread{
    @Override
    public void start(){
        /* 线程启动前的业务处理 */
        super.start();
        /* 线程启动后的业务处理 */
    }

    @Override
    public void run(){
        //MultiThread do something.
    }
}

```

注意看 start 方法，调用了父类的 start 方法，没有主动调用 run 方法，这是由 JVM 自行调用的，不用我们显式实现，而且是一定不能实现。此方式虽然解决了“覆写 start 方法”的问题，但是基本上无用武之地，到目前为止还没有发现一定要覆写 start 方法的多线程应用，所有要求覆写 start 的场景，都可以用其他的方式来实现，例如类变量、事件机制、监听等方式。

注意 继承自 Thread 类的多线程类不必覆写 start 方法。

建议 119：启动线程前 stop 方法是不可靠的

有这样一个案例，我们需要一个高效率的垃圾邮件制造机，也就是需要有尽可能多的线程来尽可能多地制造垃圾邮件，垃圾邮件需要的信息保存在数据库中，如收件地址、混淆后的标题、反垃圾处理后的内容等，垃圾制造机的作用就是从数据库中读取这些信息，判断是否符合条件（如收件地址必须包含 @ 符号、标题不能为空等），然后转换成一份真实的邮件发送出去。

整个应用逻辑很简单，这必然是一个多线程的应用，垃圾邮件制造机需要继承 Thread 类，代码如下：

```
// 垃圾邮件制造机
class SpamMachine extends Thread {
    @Override
    public void run() {
        // 制造垃圾邮件
        System.out.println("制造大量垃圾邮件 . . . . .");
    }
}
```

在客户端代码中需要发挥计算机的最大潜能来制造邮件，也就是说开尽量多的线程，这里我们使用一个 while 循环来处理，代码如下：

```
public static void main(String[] args) {
    // 不分昼夜地制造垃圾邮件
    while (true) {
        // 多线程多个垃圾邮件制造机
        SpamMachine sm = new SpamMachine();
        // 条件判断，不符合条件就设置该线程不可执行
        if (!false) {
            sm.stop();
        }
        // 如果线程是 stop 状态，则不会启动
        sm.start();
    }
}
```

在此段代码中，设置了一个极端条件：所有的线程在启动前都执行 stop 方法，虽然它是一个已过时 (Deprecated) 的方法，但它的运行逻辑还是正常的，况且 stop 方法在此处的目的并不是停止一个线程，而是设置线程为不可启用状态。想来这应该是没有问题的，但是运行结果却出现了奇怪的现象：部分线程还是启动了，也就是在某些线程（没有规律）中的 start 方法正常执行了。在不符合判断规则的情况下，不可启用状态的线程还是启用了。这是为什么呢？

这是线程启动 (start 方法) 的一个缺陷。Thread 类的 stop 方法会根据线程状态来判断

是终结线程还是设置线程为不可运行状态，对于未启动的线程（线程状态为 NEW）来说，会设置其标志位为不可启动，而其他的状态则是直接停止。stop 方法的源代码如下：

```

public final void stop() {
    if ((threadStatus != 0) && !isAlive()) {
        return;
    }
    stop1(new ThreadDeath());
}

private final synchronized void stop1(Throwable th) {
    /* 安全检查省略 */
    if (threadStatus != 0) {
        resume();
        stop0(th);
    } else {
        if (th == null) {
            throw new NullPointerException();
        }
        stopBeforeStart = true;
        throwableFromStop = th;
    }
}

```

这里设置了 stopBeforeStart 变量，标志着是在启动前设置了停止标志，在 start 方法中是这样校验的：

```

public synchronized void start() {
    // 分配栈内存，启动线程，运行 run 方法
    start0();
    // 在启动前设置了停止状态
    if (stopBeforeStart) {
        stop0(throwableFromStop);
    }
}
// 本地方法
private native void start0();

```

注意看 start0 方法 和 stop0 方法 的顺序，start0 方法 在 前，也 就 是 说 即使 stopBeforeStart 为 true (不可启动)，也会先启动一个线程，然后再 stop0 结束这个线程，而罪魁祸首就在这里！

明白了原因，我们的情景代码也就很容易修改了，代码如下：

```

public static void main(String[] args) {
    // 不分昼夜的制造垃圾邮件
    while (true) {
        // 条件判断，不符合条件就不创建线程
        if (!false) {
            // 多线程多个垃圾邮件制造机
        }
    }
}

```

```
        new SpamMachine().start();  
    }  
}
```

不再使用 `stop` 方法进行状态的设置，直接通过判断条件来决定线程是否可启用。对于 `start` 方法的该缺陷，一般不会引起太大的问题，只是增加了线程启动和停止的精度而已。

建议 120: 不使用 stop 方法停止线程

线程启动完毕后，在运行时可能需要终止，Java 提供的终止方法只有一个 stop，但是我不建议使用这个方法，因为它有以下三个问题：

(1) stop 方法是过时的

从 Java 编码规则来说，已经过时的方法不建议采用。

(2) stop 方法会导致代码逻辑不完整

`stop` 方法是一种“恶意”的中断，一旦执行 `stop` 方法，即终止当前正在运行的线程，不管线程逻辑是否完整，这是非常危险的。看如下的代码：

```
public static void main(String[] args) throws Exception {
    // 子线程
    Thread thread = new Thread() {
        @Override
        public void run() {
            try {
                // 子线程休眠 1 秒
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // 异常处理
            }
            System.out.println("此处代码不会执行");
        }
    };
    // 启动线程
    thread.start();
    // 主线程休眠 0.1 秒
    Thread.sleep(100);
    // 子线程停止
    thread.stop();
}
```

这段代码的逻辑是这样的：子线程是一个匿名内部类，它的 run 方法在执行时会休眠 1 秒钟，然后再执行后续的逻辑，而主线程则是休眠 0.1 秒后终止子线程的运行，也就是说，JVM 在执行 thread.stop() 时，子线程还在执行 sleep(1000)，此时 stop 方法会清除栈内信息，结束该线程，这也就导致了 run 方法的逻辑不完整，输出语句 println 代表的是一段逻辑，可

能非常重要，比如子线程的主逻辑、资源回收、情景初始化等，但是因为 stop 线程了，这些就都不再执行，于是就产生了业务逻辑不完整的情况。

这是极度危险的，因为我们不知道子线程会在什么时候被终止，stop 连基本的逻辑完整性都无法保证。而且此种操作也是非常隐蔽的，子线程执行到何处会被关闭很难定位，这为以后的维护带来了很多麻烦。

(3) stop 方法会破坏原子逻辑

多线程为了解决共享资源抢占的问题，使用了锁概念，避免资源不同步，但是正因此原因，stop 方法却会带来更大的麻烦：它会丢弃所有的锁，导致原子逻辑受损。例如有这样一段程序：

```
class MultiThread implements Runnable {
    int a = 0;

    @Override
    public void run() {
        // 同步代码块，保证原子操作
        synchronized ("") {
            // 自增
            a++;
            try {
                // 线程休眠 0.1 秒
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 自减
            a--;
            String tn = Thread.currentThread().getName();
            System.out.println(tn + ":a =" + a);
        }
    }
}
```

MultiThread 实现了 Runnable 接口，具备多线程能力，其中 run 方法中加上了 synchronized 代码块，表示内部是原子逻辑，它会先自增然后再自减少，按照 synchronized 同步代码块的规则来处理，此时无论启动多少个线程，打印出来的结果都应该是 a=0，但是如果有一个正在执行的线程被 stop，就会破坏这种原子逻辑，代码如下：

```
public static void main(String[] args) {
    MultiThread t = new MultiThread();
    Thread t1 = new Thread(t);
    // 启动 t1 线程
    t1.start();
    for (int i = 0; i < 5; i++) {
        new Thread(t).start();
    }
}
```

```

    }
    // 停止 t1 线程
    t1.stop();
}

```

首先要说明的是所有线程共享了一个 MultiThread 的实例变量 t，其次由于在 run 方法中加入了同步代码块，所以只能有一个线程进入到 synchronized 块中。此段代码的执行顺序如下：

- 1) 线程 t1 启动，并执行 run 方法，由于没有其他线程持同步代码块的锁，所以 t1 线程执行自加后执行到 sleep 方法即开始休眠，此时 a=1。
- 2) JVM 又启动了 5 个线程，也同时运行 run 方法，由于 synchronized 关键字的阻塞作用，这 5 个线程不能执行自增和自减操作，等待 t1 线程锁释放。
- 3) 主线程执行了 t1.stop 方法，终止了 t1 线程，注意，由于 a 变量是所有线程共享的，所以其他 5 个线程获得的 a 变量也是 1。
- 4) 其他 5 个线程依次获得 CPU 执行机会，打印出 a 值。

分析了这么多，相信读者也明白了输出的结果，结果如下：

```

Thread-5:a =1
Thread-4:a =1
Thread-3:a =1
Thread-2:a =1
Thread-1:a =1

```

原本期望 synchronized 同步代码块中的逻辑都是原子逻辑，不受外界线程的干扰，但是结果却出现原子逻辑被破坏的情况，这也是 stop 方法被废弃的一个重要原因：破坏了原子逻辑。

既然终止一个线程不能使用 stop 方法，那怎样才能终止一个正在运行的线程呢？答案也很简单，使用自定义的标志位决定线程的执行情况，代码如下：

```

class SafeStopThread extends Thread {
    // 此变量必须加上 volatile
    private volatile boolean stop = false;
    @Override
    public void run() {
        // 判断线程体是否运行
        while (!stop) {
            // Do Something
        }
    }
    // 线程终止
    public void terminate() {
        stop = true;
    }
}

```

这是很简单的办法，在线程体中判断是否需要停止运行，即可保证线程体的逻辑完整性，而且也不会破坏原子逻辑。可能有读者对 Java API 比较熟悉，于是提出疑问：Thread 不是还提供了 interrupt 中断线程的方法吗？这个方法可不是过时方法，那可以使用吗？它可以终止一个线程吗？

非常好的问题，interrupt，名字看上去很像是终止一个线程的方法，但是我可以很明确地告诉你，它不是，它不能终止一个正在执行着的线程，它只是修改中断标志而已，例如下面一段代码：

```
public static void main(String[] args) {
    Thread t1 = new Thread() {
        public void run() {
            // 线程一直运行
            while (true) {
                System.out.println("Running.....");
            }
        }
    };
    // 启动 t1 线程
    t1.start();
    // 中断 t1 线程
    t1.interrupt();
}
```

执行这段代码，你会发现一直有 Running 在输出，永远不会停止，似乎执行了 interrupt 没有任何变化，那是因为 interrupt 方法不能终止一个线程状态，它只会改变中断标志位（如果在 t1.interrupt() 前后输出 t1.isInterrupted() 则会发现分别输出了 false 和 true），如果需要终止该线程，还需要自行进行判断，例如我们可以使用 interrupt 编写出更加简洁、安全的终止线程代码：

```
class SafeStopThread extends Thread {
    @Override
    public void run() {
        // 判断线程体是否运行
        while (!isInterrupted()) {
            // Do Something
        }
    }
}
```

总之，如果期望终止一个正在运行的线程，则不能使用已经过时的 stop 方法，需要自行编码实现，如此即可保证原子逻辑不被破坏，代码逻辑不会出现异常。当然，如果我们使用的是线程池（比如 ThreadPoolExecutor 类），那么可以通过 shutdown 方法逐步关闭池中的线程，它采用的是比较温和、安全的关闭线程方法，完全不会产生类似 stop 方法的弊端。

建议 121：线程优先级只使用三个等级

线程的优先级（Priority）决定了线程获得 CPU 运行的机会，优先级越高获得的运行机会越大，优先级越低获得的机会越小。Java 的线程有 10 个级别（准确地说是 11 个级别，级别为 0 的线程是 JVM 的，应用程序不能设置该级别），那是不是说级别是 10 的线程肯定比级别为 9 的线程先运行呢？我们来看如下一个线程类：

```
class TestThread implements Runnable {
    // 启动线程
    public void start(int _priority) {
        Thread t = new Thread(this);
        // 设置线程优先级
        t.setPriority(_priority);
        t.start();
    }

    @Override
    public void run() {
        // 消耗 CPU 的计算，性能差的机器，请修改循环限制
        for (int i = 0; i < 100000; i++) {
            Math.hypot(Math.pow(924526789, i), Math.cos(i));
        }
        // 输出线程优先级
        System.out.println("Priority:" + Thread.currentThread().getPriority());
    }
}
```

该多线程类实现了 Runnable 接口，实现了 run 方法，注意在 run 方法中有一个比较占用 CPU 的计算，该计算毫无意义，只是为了保证一个线程尽可能多地消耗 CPU 资源，目的是为了观察 CPU 繁忙时不同优先级线程的执行顺序。需要说明的是，如果此处使用了 Thread.sleep() 方法，则不能体现出线程优先级的本质了，因为 CPU 并不繁忙，线程调度不会遵循优先级顺序来进行调度。

客户端的代码如下：

```
public static void main(String[] args) {
    // 启动 20 个不同优先级的线程
    for (int i = 0; i < 20; i++) {
        new TestThread().start(i % 10 + 1);
    }
}
```

这里创建了 20 个线程，每个线程在运行时都耗尽了 CPU 资源，因为优先级不同，线程调度应该最先处理优先级最高的，然后处理优先级最低的，也就是先执行 2 个优先级为 10 的线程，然后执行 2 个优先级为 9 的线程，2 个优先级为 8 的线程……但是结果却并不是这样的。

```
Priority:10
Priority:9
Priority:10
Priority:9
Priority:7
Priority:7
Priority:8
Priority:8
Priority:5
Priority:5
Priority:6
Priority:6
Priority:4
Priority:4
Priority:3
Priority:3
Priority:1
Priority:1
Priority:2
Priority:2
```

`println` 方法虽然有输出损耗，可能会影响到输出结果，但是不管运行多少次，你都会发现两个不争的事实：

(1) 并不是严格遵照线程优先级别来执行的

比如线程优先级为 9 的线程可能比优先级为 10 的线程先执行，优先级为 1 的线程可能比优先级为 2 的线程先执行，但很少会出现优先级为 2 的线程比优先级为 10 的线程先执行（这里用了一个词“很少”，是说确实有可能出现，只是几率非常低，因为优先级只是表示线程获得 CPU 运行的机会，并不代表强制的排序号）。

(2) 优先级差别越大，运行机会差别越明显

比如优先级为 10 的线程通常会比优先级为 2 的线程先执行，但是优先级为 6 的线程和优先级为 5 的线程差别就不太明显了，执行多次，你会发现有不同的顺序。

这两个现象是线程优先级的一个重要表现，之所以会出现这种情况，是因为线程运行是需要获得 CPU 资源的，那谁能决定哪个线程先获得哪个线程后获得呢？这是依照操作系统设定的线程优先级来分配的，也就是说，每个线程要运行，需要操作系统分配优先级和 CPU 资源，对于 Java 来说，JVM 调用操作系统的接口设置优先级，比如 Windows 操作系统是通过调用 `SetThreadPriority` 函数来设置的，问题来了：不同的操作系统线程优先级都相同吗？

这是个好问题。事实上，不同的操作系统线程优先级是不相同的，Windows 有 7 个优先级，Linux 有 140 个优先级，Freebsd 则有 255 个（此处指的是优先级总数，不同操作系统有不同的分类，如中断级线程、操作系统级等，各个操作系统具体用户可用的线程数量也不相同）。Java 是跨平台的系统，需要把这个 10 个优先级映射成不同操作系统的优先级，于是界定了 Java 的优先级只是代表抢占 CPU 的机会大小，优先级越高，抢占 CPU 的机会越大，被

优先执行的可能性越高，优先级相差不大，则抢占 CPU 的机会差别也不大，这就是导致了优先级为 9 的线程可能比优先级为 10 的线程先运行。

Java 的缔造者们也察觉到了线程优先问题，于是在 Thread 类中设置了三个优先级，此意就是告诉开发者，建议使用优先级常量，而不是 1 到 10 随机的数字。常量代码如下：

```
public class Thread implements Runnable {
    // 最低优先级
    public final static int MIN_PRIORITY = 1;
    // 普通优先级，默认值
    public final static int NORM_PRIORITY = 5;
    // 最高优先级
    public final static int MAX_PRIORITY = 10;
}
```

在编码时直接使用这些优先级常量，可以说在大部分情况下 MAX_PRIORITY 的线程会比 NORM_PRIORITY 的线程先运行，但是不能认为必然会先运行，不能把这个优先级做为核心业务的必然条件，Java 无法保证优先级高肯定会先执行，只能保证高优先级有更多的执行机会。因此，建议在开发时只使用此三类优先级，没有必要使用其他 7 个数字，这样也可以保证在不同的操作系统上优先级的表现基本相同。

明白了这个问题，那可能有读者要问了：如果优先级相同呢？这很好办，也是由操作系统决定的，基本上是按照 FIFO 原则（先入先出，First Input First Output），但也是不能完全保证。

注意 线程优先级推荐使用 MIN_PRIORITY、NORM_PRIORITY、MAX_PRIORITY 三个级别，不建议使用其他 7 个数字。

建议 122：使用线程异常处理器提升系统可靠性

我们要编写一个 Socket 应用，监听指定端口，实现数据包的接收和发送逻辑，这在早期系统间进行数据交互是经常使用的，这类接口通常需要考虑两个问题：一是避免线程阻塞，保证接收的数据尽快处理；二是接口的稳定性和可靠性问题，数据包很复杂，接口服务的系统也很多，一旦守候线程出现异常就会导致 Socket 停止响应，这是非常危险的，那我们有什么办法来避免吗？

Java 1.5 版本以后在 Thread 类中增加了 setUncaughtExceptionHandler 方法，实现了线程异常的捕捉和处理。可能大家会有一个疑问：如果 Socket 应用出现了不可预测的异常是否可以自动重启呢？其实使用线程异常处理器很容易解决，我们来看一个异常处理器应用的例子，代码如下：

```
class TcpServer implements Runnable {
    // 创建后即运行
```

```

public TcpServer() {
    Thread t = new Thread(this);
    t.setUncaughtExceptionHandler(new TcpServerExceptionHandler());
    t.start();
}

@Override
public void run() {
    // 正常业务运行，运行 3 秒
    for (int i = 0; i < 3; i++) {
        try {
            Thread.sleep(1000);
            System.out.println("系统正常运行：" + i);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // 抛出异常
    throw new RuntimeException();
}

// 异常处理器
private static class TcpServerExceptionHandler implements
    Thread.UncaughtExceptionHandler {
    @Override
    public void uncaughtException(Thread t, Throwable e) {
        // 记录线程异常信息
        System.out.println("线程 " + t.getName() + " 出现异常，自行重启，请分析原因。");
        e.printStackTrace();
        // 重启线程，保证业务不中断
        new TcpServer();
    }
}
}
}

```

这段代码的逻辑比较简单，在 TcpServer 类创建时即启动一个线程，提供 TCP 服务，例如接收和发送文件，具体逻辑在 run 方法中实现。同时，设置了该线程出现运行期异常（也就是 Uncaught Exception）时，由 TcpServerExceptionHandler 异常处理器来处理。那 TcpServerExceptionHandler 异常处理器做什么事呢？两件事：

- 记录异常信息，以便查找问题。
- 重新启动一个新线程，提供不间断的服务。

有了这两点，TcpServer 就可以稳定地运行了，即使出现异常也能自动重启。客户端代码比较简单，只需要 new TcpServer() 即可，运行结果如下：

```

系统正常运行：0
系统正常运行：1
系统正常运行：2

```

```

线程 Thread-0 出现异常，自行重启，请分析原因。
java.lang.RuntimeException
at TcpServer.run(Client.java:30)
at java.lang.Thread.run(Thread.java:619)
系统正常运行 :0
系统正常运行 :1
系统正常运行 :2
线程 Thread-1 出现异常，自行重启，请分析原因。
java.lang.RuntimeException
at TcpServer.run(Client.java:30)
at java.lang.Thread.run(Thread.java:619)

```

从运行结果上也可以看出，当 Thread-0 出现异常时，系统自动启动了 Thread-1 线程，继续提供服务，大大提高了系统的可靠性。

这段程序只是一个示例程序，若要在实际环境中应用，则需要注意以下三个方面：

(1) 共享资源锁定

如果线程异常产生的原因是资源被锁定，自动重启应用只会增加系统的负担，无法提供不间断服务。例如一个即时通信服务器（XMPP Server）出现信息不能写入的情况时，即使再怎么重启服务，也是无法解决问题的。在此情况下最好的办法是停止所有的线程，释放资源。

(2) 脏数据引起系统逻辑混乱

异常的产生中断了正在执行的业务逻辑，特别是如果正在执行一个原子操作（像即时通信服务器的用户验证和签到这两个事件应该在一个操作中处理，不允许出现验证成功但签到不成功的情况），但如果此时抛出了运行期异常就有可能会破坏正常的业务逻辑，例如出现用户认证通过了，但签到不成功的情况，在这种情景下重启应用服务器，虽然可以提供服务，但对部分用户则产生了逻辑异常。

(3) 内存溢出

线程异常了，但由该线程创建的对象并不会马上回收，如果再重新启动新线程，再创建一批新对象，特别是加入了场景接管，就非常危险了，例如即时通信服务，重新启动一个新线程必须保证原在线用户的透明性，即用户不会察觉服务重启，在此种情况下，就需要在线程初始化时加载大量对象以保证用户的状态信息，但是如果线程反复重启，很可能会引起 OutOfMemory 内存泄露问题。

建议 123: volatile 不能保证数据同步

`volatile` 关键字比较少用，原因无外乎两点，一是在 Java 1.5 之前该关键字在不同的操作系统上有不同的表现，所带来的问题就是移植性较差；二是比较难设计，而且误用较多，这也导致它的“名誉”受损。

我们知道，每个线程都运行在栈内存中，每个线程都有自己的工作内存（Working

Memory, 比如寄存器 Register、高速缓冲存储器 Cache 等), 线程的计算一般是通过工作内存进行交互的, 其示意图如图 9-1 所示。

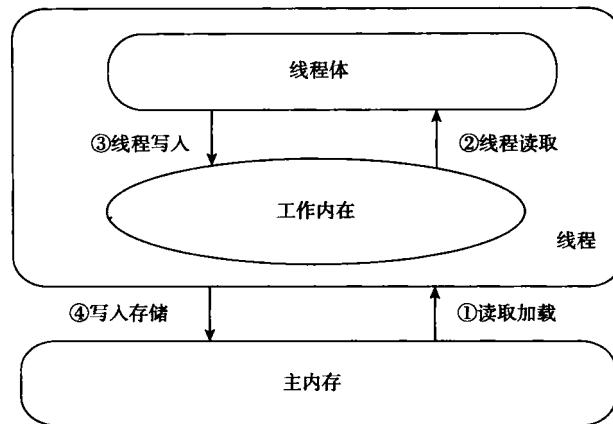


图 9-1 线程读取变量的示意图

从示意图上我们可以看到, 线程在初始化时从主内存中加载所需的变量值到工作内存中, 然后在线程运行时, 如果是读取, 则直接从工作内存中读取, 若是写入则先写到工作内存中, 之后再刷新到主存中, 这是 JVM 的一个简单的内存模型, 但是这样的结构在多线程的情况下有可能会出现问题, 比如: A 线程修改变量的值, 也刷新到了主存中, 但 B、C 线程在此时间内读取的还是本线程的工作内存, 也就是说它们读取的不是最“新鲜”的值, 此时就出现了不同线程持有的公共资源不同步的情况。

对于此类问题有很多解决办法, 比如使用 synchronized 同步代码块, 或者使用 Lock 锁来解决该问题, 不过, Java 可以使用 volatile 更简单地解决此类问题, 比如在一个变量前加上 volatile 关键字, 可以确保每个线程对本地变量的访问和修改都是直接与主内存交互的, 而不是与本线程的工作内存交互的, 保证每个线程都能获得最“新鲜”的变量值, 其示意图如图 9-2 所示。

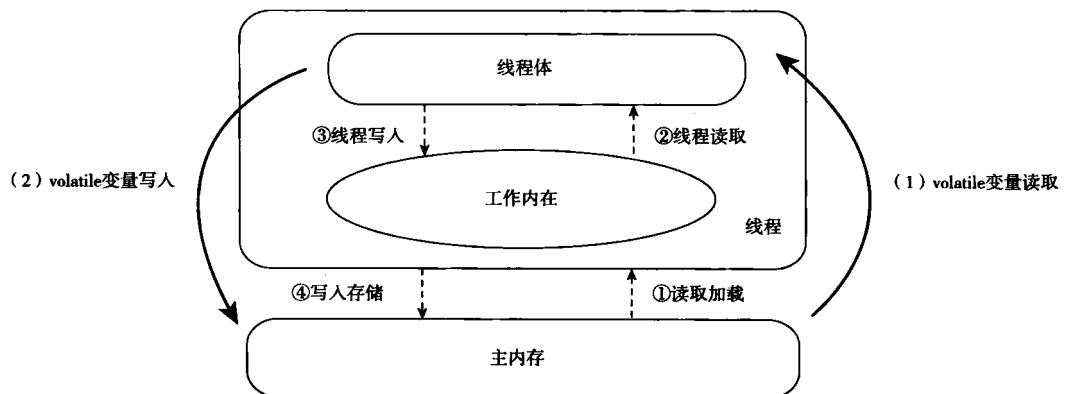


图 9-2 volatile 变量操作示意图

明白了 volatile 变量的原理，那我们思考一下：volatile 变量是否能够保证数据的同步性呢？两个线程同时修改一个 volatile 是否会产生脏数据呢？我们来看下面的代码：

```
class UnsafeThread implements Runnable {
    // 共享资源
    private volatile int count = 0;
    @Override
    public void run() {
        // 增加 CPU 的繁忙程度，不用关心其逻辑含义
        for (int i = 0; i < 1000; i++) {
            Math.hypot(Math.pow(92456789, i), Math.cos(i));
        }
        // 自增运算
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

上面的代码定义了一个多线程类，run 方法的主要逻辑是共享资源 count 的自加运算，而且我们还为 count 变量加上了 volatile 关键字，确保是从主内存中读取和写入的，如果有多个线程运行，也就是多个线程执行 count 变量的自加动作，count 变量会产生脏数据吗？想想看，我们已经为 count 加上了 volatile 关键字呀！模拟多线程的代码如下：

```
public static void main(String[] args) throws Exception {
    // 理想值，并作为最大循环次数
    int value = 1000;
    // 循环次数，防止出现无限循环造成死机情况
    int loops = 0;
    // 主线程组，用于估计活动线程数
    ThreadGroup tg = Thread.currentThread().getThreadGroup();
    while (loops++ < value) {
        // 共享资源清零
        UnsafeThread ut = new UnsafeThread();
        for (int i = 0; i < value; i++) {
            new Thread(ut).start();
        }
        // 先等 15 毫秒，等待活动线程数量成为 1
        do {
            Thread.sleep(15);
        } while (tg.activeCount() != 1);
        // 检查实际值与理论值是否一致
        if (ut.getCount() != value) {
            // 出现线程不安全的情况
            System.out.println("循环到第 " + loops + " 遍，出现线程不安全情况");
            System.out.println("此时，count=" + ut.getCount());
        }
    }
}
```

```
        System.exit(0);  
    }  
}
```

这是一段设计很巧妙的程序，要让 volatile 变量“出点丑”还是需要花点功夫的。此段程序的运行逻辑如下：

- 启动 100 个线程，修改共享资源 count 的值。
 - 暂停 15 毫秒，观察活动线程数是否为 1（即只剩下主线程在运行），若不为 1，则再等待 15 毫秒。
 - 判断共享资源是否是不安全的，即实际值与理想值是否相同，若不相同，则发现目标，此时 count 的值为脏数据。
 - 如果没有找到，继续循环，直到达到最大循环次数为止。

运行结果如下：

循环到第 247 遍，出现线程不安全情况
此时，`count=999`

这只是一个可能的结果，每次执行都有可能产生不同的结果。这也说明我们的 count 变量没有实现数据同步，在多个线程修改的情况下，count 的实际值与理论值产生了偏差，直接说明了 volatile 关键字并不能保证线程安全。

在解释原因之前，我们先说一下自加操作。`count++` 表示的是先取出 `count` 的值然后再加 1，也就是 `count = count+1`，所以，在某两个紧邻的时间片段内会发生如下神奇的事情：

(1) 第一个时间片段

A 线程获得执行机会，因为有关关键字 volatile 修饰，所以它从主内存中获得 count 的最新值 998，接下来的事情又分为两种类型：

- 如果是单 CPU，此时调度器暂停 A 线程执行，出让执行机会给 B 线程，于是 B 线程也获得了 count 的最新值 998。
 - 如果是多 CPU，此时线程 A 继续执行，而线程 B 也同时获得 count 的最新值 998。

(2) 第二个时间片段

- 如果是单 CPU, B 线程执行完加 1 动作 (这是一个原子处理), count 的值为 999, 由于 volatile 类型的变量, 所以直接写入主内存, 然后 A 线程继续执行, 计算的结果也是 999, 重新写入主内存中。
 - 如果是多 CPU, A 线程执行完加 1 动作后修改主内存的变量 count 为 999, 线程 B 执行完毕后也修改主内存中的变量为 999。

这两个时间片段执行完毕后，原本期望的结果为 1000，但运行后的值却为 999，这表示出现了线程不安全的情况。这也是我们要说明的：`volatile` 关键字并不能保证线程安全，它只

能保证当线程需要该变量的值时能够获得最新的值，而不能保证多个线程修改的安全性。

顺便说一下，在上面的代码中，UnsafeThread 类的消耗 CPU 计算是必须的，其目的是加重线程的负荷，以便出现单个线程抢占整个 CPU 资源的情景，否则很难模拟出 volatile 线程不安全的情况，读者可以自行模拟测试。

注意 volatile 不能保证数据是同步的，只能保证线程能够获得最新值。

建议 124：异步运算考虑使用 Callable 接口

多线程应用有两种实现方式，一种是实现 Runnable 接口，另一种是继承 Thread 类，这两个方式都有缺点：run 方法没有返回值，不能抛出异常（这两个缺点归根到底是 Runnable 接口的缺陷，Thread 也是实现了 Runnable 接口），如果需要知道一个线程的运行结果就需要用户自行设计，线程类自身也不能提供返回值和异常。但是从 Java 1.5 开始引入了一个新的接口 Callable，它类似于 Runnable 接口，实现它就可以实现多线程任务，Callable 的接口定义如下：

```
public interface Callable<V> {
    // 具有返回值，并可抛出异常
    V call() throws Exception;
}
```

实现 Callable 接口的类，只是表明它是一个可调用的任务，并不表示它具有多线程运算能力，还是需要执行器来执行的。我们先编写一个任务类，代码如下：

```
// 税款计算器
class TaxCalculator implements Callable<Integer> {
    // 本金
    private int seedMoney;
    // 接收主线程提供的参数
    public TaxCalculator(int _seedMoney) {
        seedMoney = _seedMoney;
    }
    @Override
    public Integer call() throws Exception {
        // 复杂计算，运行一次需要 10 秒
        TimeUnit.MILLISECONDS.sleep(10000);
        return seedMoney / 10;
    }
}
```

这里模拟了一个复杂运算：税款计算器，该运算可能要花费 10 秒钟的时间，此时不能让用户一直等着吧，需要给用户输出点什么，让用户知道系统还在运行，这也是系统友好性的体现：用户输入即有输出，若耗时较长，则显示运算进度。如果我们直接计算，就只有一个 main 线程，是不可能有友好提示的，如果税金不计算完毕，也不会执行后续动作，所以

此时最好的办法就是重启一个线程来运算，让 main 线程做进度提示，代码如下：

```
public static void main(String[] args) throws Exception {
    // 生成一个单线程的异步执行器
    ExecutorService es = Executors.newSingleThreadExecutor();
    // 线程执行后的期望值
    Future<Integer> future = es.submit(new TaxCalculator(100));
    while(!future.isDone()){
        // 还没有运算完成，等待 200 毫秒
        TimeUnit.MILLISECONDS.sleep(200);
        // 输出进度符号
        System.out.print("#");
    }
    System.out.println("\n计算完成，税金是：" + future.get() + " 元");
    es.shutdown();
}
```

在该段代码中，Executors 是一个静态工具类，提供了异步执行器的创建能力，如单线程执行器 newSingleThreadExecutor、固定线程数量的执行器 newFixedThreadPool 等，一般它是异步计算的入口类。Future 关注的是线程执行后的结果，比如有没有运行完毕，执行结果是多少等。此段代码的运行结果如下所示：

```
#####
计算完成，税金是：10 元
```

执行时，“#”会依次递增，表示系统正在运算，为用户提供了运算进度。此类异步计算的好处是：

- 尽可能多地占用系统资源，提供快速运算。
- 可以监控线程执行的情况，比如是否执行完毕、是否有返回值、是否有异常等。
- 可以为用户提供更好的支持，比如例子中的运算进度等。

建议 125：优先选择线程池

在 Java 1.5 之前，实现多线程编程比较麻烦，需要自己启动线程，并关注同步资源，防止出现线程死锁等问题，在 1.5 版之后引入了并行计算框架，大大简化了多线程开发。我们知道一个线程有五个状态：新建状态（New）、可运行状态（Runnable，也叫做运行状态）、阻塞状态（Blocked）、等待状态（Waiting）、结束状态（Terminated），线程的状态只能由新建转变为了运行态后才可能被阻塞或等待，最后终结，不可能产生本末倒置的情况，比如想把一个结束状态的线程转变为新建状态，则会出现异常，例如如下代码会抛出异常：

```
public static void main(String[] args) throws Exception {
    // 创建一个线程，新建状态
```

```

        Thread t = new Thread(new Runnable() {
            public void run() {
                System.out.println("线程在运行.....");
            }
        });
        // 运行状态
        t.start();
        // 是否是运行态，若不是则等待 10 毫秒
        while (!t.getState().equals(Thread.State.TERMINATED)) {
            TimeUnit.MILLISECONDS.sleep(10);
        }
        // 直接由结束态转变为运行态
        t.start();
    }
}

```

此段程序运行时会报 `IllegalThreadStateException` 异常，原因就是不能从结束状态直接转变为可运行状态，我们知道一个线程的运行时间分为三部分：T1 为线程启动时间，T2 为线程体的运行时间，T3 为线程销毁时间，如果一个线程不能被重复使用，每次创建一个线程都需要经过启动、运行、销毁这三个过程，那么这势必会增大系统的响应时间，有没有更好的办法降低线程的运行时间呢？

T2 是无法避免的，只有通过优化代码来实现降低运行时间。T1 和 T2 都可以通过线程池（Thread Pool）来缩减时间，比如在容器（或系统）启动时，创建足够多的线程，当容器（或系统）需要时直接从线程池中获得线程，运算出结果，再把线程返回到线程池中——`ExecutorService` 就是实现了线程池的执行器，我们来看一个示例代码：

```

public static void main(String[] args) {
    //2 个线程的线程池
    ExecutorService es = Executors.newFixedThreadPool(2);
    // 多次执行线程体
    for (int i = 0; i < 4; i++) {
        es.submit(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName());
            }
        });
    }
    // 关闭执行器
    es.shutdown();
}

```

此段代码首先创建了一个包含两个线程的线程池，然后在线程池中多次运行线程体，输出运行时的线程名称，结果如下：

```

pool-1-thread-1
pool-1-thread-2
pool-1-thread-1

```

pool-1-thread-2

本次代码执行了 4 遍线程体，按照我们之前阐述的“一个线程不可能从结束状态转变为可运行状态”，那为什么此处的 2 个线程可以反复使用呢？这就是我们要搞清楚的重点。

线程池的实现涉及以下三个名词：

(1) 工作线程 (Worker)

线程池中的线程，只有两个状态：可运行状态和等待状态，在没有任务时它们处于等待状态，运行时可以循环地执行任务。

(2) 任务接口 (Task)

这是每个任务必须实现的接口，以供工作线程调度器调度，它主要规定了任务的入口、任务执行完的场景处理、任务的执行状态等。这里有两种类型的任务：具有返回值（或异常）的 Callable 接口任务和无返回值并兼容旧版本的 Runnable 接口任务。

(3) 任务队列 (Work Queue)

也叫做工作队列，用于存放等待处理的任务，一般是 BlockingQueue 的实现类，用来实现任务的排队处理。

我们首先从线程池的创建说起，Executors.newFixedThreadPool(2) 表示创建一个具有 2 个线程的线程池，源代码如下：

```
public class Executors {
    public static ExecutorService newFixedThreadPool(int nThreads) {
        // 生成一个最大为 nThreads 的线程池执行器
        return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.
            MILLISCONDS,new LinkedBlockingQueue<Runnable>());
    }
}
```

这里使用了 LinkedBlockingQueue 作为任务队列管理器，所有等待处理的任务都会放在该队列中，需要注意的是，此队列是一个阻塞式的单端队列。线程池建立好了，那就需要线程在其中运行了，线程池中的线程是在 submit 第一次提交任务时建立的，代码如下：

```
public Future<?> submit(Runnable task) {
    // 检查任务是否为 null
    if (task == null) throw new NullPointerException();
    // 把 Runnable 任务包装成具有返回值的任务对象，不过此时并没有执行，只是包装
    RunnableFuture<Object> ftask = newTaskFor(task, null);
    // 执行此任务
    execute(ftask);
    // 返回任务预期执行结果
    return ftask;
}
```

此处的代码关键是 execute 方法，它实现了三个职责。

- 创建足够多的工作线程数，数量不超过最大线程数量，并保持线程处于运行或等待状态。
- 把等待处理的任务放到任务队列中。
- 从任务队列中取出任务来执行。

其中此处的关键是工作线程的创建，它也是通过 new Thread 方式创建的一个线程，只是它创建的并不是我们的任务线程（虽然我们的任务实现了 Runnable 接口，但它只是起一个标志性的作用），而是经过包装的 Worker 线程，代码如下：

```
private final class Worker implements Runnable {
    // 运行一次任务
    private void runTask(Runnable task) {
        // 这里的 task 才是我们自定义实现 Runnable 接口的任务
        task.run();
    }
    // 工作线程也是线程，必须实现的 run 方法
    public void run() {
        while (task != null || (task = getTask()) != null) {
            runTask(task);
            task = null;
        }
    }
    // 从任务队列中获得任务
    Runnable getTask() {
        for (;;) {
            return workQueue.take();
        }
    }
}
```

此处为示意代码，删除了大量的判断条件和锁资源。execute 方法是通过 Worker 类启动的一个工作线程，执行的是我们的第一个任务，然后该线程通过 getTask 方法从任务队列中获取任务，之后再继续执行，但问题是任务队列是一个 BlockingQueue，是阻塞式的，也就是说如果该队列元素为 0，则保持等待状态，直到有任务进入为止，我们来看 LinkedBlockingQueue 的 take 方法，代码如下：

```
public E take() throws InterruptedException {
    // 如果队列中元素数量为 0，则等待
    while (count.get() == 0)
        notEmpty.await();
    // 等待状态结束，弹出头元素
    x = extract();
    // 如果队列数量还多于 1 个，唤醒其他线程
    if (c > 1)
        notEmpty.signal();
    // 返回头元素
    return x;
}
```

分析到这里，我们就明白了线程池的创建过程：创建一个阻塞队列以容纳任务，在第一次执行任务时创建足够多的线程（不超过许可线程数），并处理任务，之后每个工作线程自行从任务队列中获得任务，直到任务队列中的任务数量为0为止，此时，线程将处于等待状态，一旦有任务再加入到队列中，即唤醒工作线程进行处理，实现线程的可复用性。

使用线程池减少的是线程的创建和销毁时间，这对于多线程应用来说非常有帮助，比如我们最常用的Servlet容器，每次请求处理的都是一个线程，如果不采用线程池技术，每次请求都会重新创建一个线程，这会导致系统的性能负荷加大，响应效率下降，降低了系统的友好性。

建议 126：适时选择不同的线程池来实现

Java的线程池实现从根本上来说只有两个：`ThreadPoolExecutor`类和`ScheduledThreadPoolExecutor`类，这两个类还是父子关系，但是Java为了简化并行计算，还提供了一个`Executors`的静态类，它可以直接生成多种不同的线程池执行器，比如单线程执行器、带缓冲功能的执行器等，但归根结底还是使`ThreadPoolExecutor`类或`ScheduledThreadPoolExecutor`类的封装类。

为了理解这些执行器，我们首先来`ThreadPoolExecutor`类，其中它复杂的构造函数可以很好解释该线程池的作用，代码如下：

```
public class ThreadPoolExecutor extends AbstractExecutorService{
    // 最完整的构造函数
    public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long
        keepAliveTime, TimeUnit unit, BlockingQueue<Runnable>
        workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler) {
        // 检验输入条件
        if (corePoolSize < 0 || maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize || keepAliveTime < 0)
            throw new IllegalArgumentException();
        // 检验运行环境
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
}
```

这是`ThreadPoolExecutor`最完整的构造函数，其他的构造函数都是引用该构造函数实现的，我们逐步来解释这些参数的含义。

□ **corePoolSize**: 最小线程数。

线程池启动后，在池中保持线程的最小数量。需要说明的是线程数量是逐步到达 corePoolSize 值的，例如 corePoolSize 被设置为 10，而任务数量只有 5，则线程池中最多会启动 5 个线程，而不是一次性地启动 10 个线程。

□ **maximumPoolSize**: 最大线程数量。

这是池中能够容纳的最大线程数量，如果超出，则使用 RejectedExecutionHandler 拒绝策略处理。

□ **keepAliveTime**: 线程最大生命期。

这里的生命周期有两个约束条件：一是该参数针对的是超过 corePoolSize 数量的线程；二是处于非运行状态的线程。这么说吧，如果 corePoolSize 为 10，maximumPoolSize 为 20，此时线程池中有 15 个线程在运行，一段时间后，其中有 3 个线程处于等待状态的时间超过了 keepAliveTime 指定的时间，则结束这 3 个线程，此时线程池中则还有 12 个线程正在运行。

□ **unit**: 时间单位。

这是 keepAliveTime 的时间单位，可以是纳秒、毫秒、秒、分钟等选项。

□ **workQueue**: 任务队列。

当线程池中的线程都处于运行状态，而此时任务数量继续增加，则需要有一个容器来容纳这些任务，这就是任务队列。

□ **threadFactory**: 线程工厂。

定义如何启动一个线程，可以设置线程名称，并且可以确认是否是后台线程等。

□ **handler**: 拒绝任务处理器。

由于超出线程数量和队列容量而对继续增加的任务进行处理的程序。

线程池的管理是这样一个过程：首先创建线程池，然后根据任务的数量逐步将线程增大到 corePoolSize 数量，如果此时仍有任务增加，则放置到 workQueue 中，直到 workQueue 爆满为止，然后继续增加池中的线程数量（增强处理能力），最终达到 maximumPoolSize，那如果此时还有任务要增加进来呢？这就需要 handler 来处理了，或者丢弃新任务，或者拒绝新任务，或者挤占已有任务等。

在任务队列和线程池都饱和的情况下，一旦有线程处于等待（任务处理完毕，没有新任务增加）状态的时间超过 keepAliveTime，则该线程终止，也就是说池中的线程数量会逐渐降低，直至为 corePoolSize 数量为止。

我们可以把线程池想象成这样一个场景：在一条生产线上，车间规定是可以有 corePoolSize 数量的工人，但是生产线刚建立时，工作不多，不需要那么多的人。随着工作数量的增加，工人数量也逐渐增加，直至增加到 corePoolSize 数量为止，此时任务还在增加，那怎么办呢？

好办，任务排队，corePoolSize 数量的工人不停歇地处理任务，新增加的任务按照一定

的规则存放在仓库中（也就是我们的 workQueue 中），一旦任务增加的速度超过了工人处理的能力，也就是说仓库爆满时，车间就会继续招聘工人（也就是扩大线程数），直至工人数达到 maximumPoolSize 为止，那如果所有的 maximumPoolSize 工人都在处理任务，而且仓库也是饱和状态，新增任务的该怎么处理呢？这就会扔给一个叫 handler 的专门机构去处理了，它要么丢弃这些新增的任务，要么无视，要么替换掉别的任务。

过了一段时间后，任务的数量逐渐减少了，导致有一部分工人处以待工状态，为了减少开支（Java 是为了减少系统资源消耗），于是开始辞退工人，直至保持为 corePoolSize 数量的工人为止，此时即使没有工作，也不再辞退工人（池中线程数量不再减少），这也是为了保证以后再有任务时能够快速的处理。

明白了线程池的概念，我们再来看 Executors 提供的几个创建线程池的便捷方法：

□ newSingleThreadExecutor：单线程池。

顾名思义就是一个池中只有一个线程在运行，该线程永不超时。而且由于是一个线程，当有多个任务需要处理时，会将它们放置到一个无界阻塞队列中逐个处理，它的实现代码如下：

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService(new ThreadPoolExecutor(1, 1, 0L,
        TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>()));
}
```

它的使用方法也非常简单，下面是简单的示例：

```
public static void main(String[] args) throws Exception {
    // 创建单线程执行器
    ExecutorService es = Executors.newSingleThreadExecutor();
    // 执行一个任务
    Future<String> future = es.submit(new Callable<String>() {
        public String call() throws Exception {
            return "";
        }
    });
    // 获得任务执行后的返回值
    System.out.println("返回值: " + future.get());
    // 关闭执行器
    es.shutdown();
}
```

□ newCachedThreadPool：缓冲功能的线程池。

建立了一个线程池，而且线程数量是没有限制的（当然，不能超过 Integer 的最大值），新增一个任务即有一个线程处理，或者复用之前空闲的线程，或者新启动一个线程，但是一旦一个线程在 60 秒内一直是出于等待状态时（也就是 1 分钟没工作可做），则会被终止，其源代码如下。

```

public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, new
        SynchronousQueue<Runnable>());
}

```

这里需要说明的是，任务队列使用了同步阻塞队列，这意味着向队列中加入一个元素，即可唤醒一个线程（新创建的线程或复用池中空闲线程）来处理，这种队列已经没有队列深度的概念了。

□ **newFixedThreadPool：**固定线程数量的线程池。

在初始化时已经决定了线程的最大数量，若任务添加的能力超出了线程处理能力，则建立阻塞队列容纳多余的任务，源代码如下：

```

public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS, new
        LinkedBlockingQueue<Runnable>());
}

```

上面返回的是一个 ThreadPoolExecutor，它的 corePoolSize 和 maximumPoolSize 是相等的，也就是说，最大线程数量也是 nThreads。如果任务增长速度非常快，超过了 LinkedBlockingQueue 的最大容量（Integer 的最大值），那此时会如何处理呢？会按照 ThreadPoolExecutor 默认的拒绝策略（默认是 DiscardPolicy，直接丢弃）来处理。

以上三种线程池执行器都是 ThreadPoolExecutor 的简化版，目的是帮助开发人员屏蔽过多的线程细节，简化多线程开发。当需要运行异步任务时，可以直接通过 Executors 获得一个线程池，然后运行任务，不需要关注 ThreadPoolExecutor 的一系列参数是什么含义。当然，有时候这三个线程池不能满足要求，此时则可以直接操作 ThreadPoolExecutor 来实现复杂的多线程运算。可以这样来比喻：newSingleThreadExecutor、newCachedThreadPool、newFixedThreadPool 是线程池的简化版，而 ThreadPoolExecutor 则是旗舰版——简化版更容易操作，需要了解的知识相对少些，方便实用，而且旗舰版功能齐全，适用面广，但难于驾驭。

建议 127：Lock 与 synchronized 是不一样的

很多编码者都会说，Lock 类和 synchronized 关键字用在代码块的并发性和内存上时语义是一样的，都是保持代码块同时只有一个线程具有执行权。这样的说法只对了一半，我们以一个任务提交给多个线程运行为例，来看看使用显式锁（Lock 类）和内部锁（synchronized 关键字）有什么不同。首先定义一个任务：

```

class Task {
    public void doSomething() {
        try {
            // 每个线程等待 2 秒钟，注意将此时的线程转为 WAITING 状态
        }
    }
}

```

```
        Thread.sleep(2000);
    } catch (Exception e) {
        // 异常处理
    }
    StringBuffer sb = new StringBuffer();
    // 线程名称
    sb.append("线程名称: " + Thread.currentThread().getName());
    // 运行的时间截
    sb.append(", 执行时间: " + Calendar.getInstance().get(13) + " s");
    System.out.println(sb);
}
```

该类模拟了一个执行时间比较长的计算，注意这里使用的是模拟方式，在使用 sleep 方法时线程的状态会从运行状态转变为等待状态。该任务要具备多线程能力时必须实现 Runnable 接口，我们分别建立两种不同的锁实现机制，首先看显式锁实现：

```
// 显式锁任务
class TaskWithLock extends Task implements Runnable {
    // 声明显式锁
    private final Lock lock = new ReentrantLock();
    @Override
    public void run() {
        try {
            // 开始锁定
            lock.lock();
            doSomething();
        } finally {
            // 释放锁
            lock.unlock();
        }
    }
}
```

这里有一点需要说明的是，显式锁的锁定和释放必须在一个 try……finally 块中，这是为了确保即使出现运行期异常也能正常释放锁，保证其他线程能够顺利执行。

内部锁的处理也非常简单，代码如下：

```
// 内部锁任务
class TaskWithSync extends Task implements Runnable {
    @Override
    public void run() {
        // 内部锁
        synchronized ("A") {
            doSomething();
        }
    }
}
```

这两个任务看着非常相似，应该能够产生相似的结果吧？我们建立一个模拟场景，保证同时有三个线程在运行，代码如下：

```
public static void runTasks(Class<? extends Runnable> clz) throws Exception {
    ExecutorService es = Executors.newCachedThreadPool();
    System.out.println("**** 开始执行 " + clz.getSimpleName() + " 任务 ****");
    // 启动三个线程
    for (int i = 0; i < 3; i++) {
        es.submit(clz.newInstance());
    }
    // 等待足够长的时间，然后关闭执行器
    TimeUnit.SECONDS.sleep(10);
    System.out.println("-----" + clz.getSimpleName() + " 任务执行完毕 -----");
    // 关闭执行器
    es.shutdown();
}

public static void main(String[] args) throws Exception {
    // 运行显式锁任务
    runTasks(TaskWithLock.class);
    // 运行内部锁任务
    runTasks(TaskWithSync.class);
}
```

按照一般的理解，Lock 和 synchronized 的处理方式是相同的，输出应该没有差别，但是很遗憾的是，输出差别其实很大。输出如下：

```
***** 开始执行 TaskWithLock 任务 *****
线程名称: pool-1-thread-1, 执行时间: 33 s
线程名称: pool-1-thread-2, 执行时间: 33 s
线程名称: pool-1-thread-3, 执行时间: 33 s
-----TaskWithLock 任务执行完毕 -----

***** 开始执行 TaskWithSync 任务 *****
线程名称: pool-2-thread-1, 执行时间: 43 s
线程名称: pool-2-thread-3, 执行时间: 45 s
线程名称: pool-2-thread-2, 执行时间: 47 s
-----TaskWithSync 任务执行完毕 -----
```

注意看运行的时间戳，显式锁是同时运行的，很显然在 pool-1-thread-1 线程执行到 sleep 时，其他两个线程也会运行到这里，一起等待，然后一起输出，这还具有线程互斥的概念吗？

而内部锁的输出则是我们的预期结果：pool-2-thread-1 线程在运行时其他线程处于等待状态，pool-2-thread-1 执行完毕后，JVM 从等待线程池中随机获得一个线程 pool-2-thread-3 执行，最后再执行 pool-2-thread-2，这正是我们希望的。

现在问题来了：Lock 锁为什么不出现互斥情况呢？

这是因为对于同步资源来说（示例中是代码块），显式锁是对象级别的锁，而内部锁是类级别的锁，也就是说 Lock 锁是跟随对象的，synchronized 锁是跟随类的，更简单地说把 Lock 定义为多线程类的私有属性是起不到资源互斥作用的，除非是把 Lock 定义为所有线程的共享变量。都说代码是最好的解释语言，我们来看一个 Lock 锁资源的代码：

```
public static void main(String[] args) {
    // 多个线程共享锁
    final Lock lock = new ReentrantLock();
    // 启动三个线程
    for (int i = 0; i < 3; i++) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    lock.lock();
                    // 休眠 2 秒钟
                    Thread.sleep(2000);

                    System.out.println(Thread.currentThread().getName());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        }).start();
    }
}
```

读者可以执行一下，会发现线程名称 Thread-0、Thread-1、Thread-2 会逐渐输出，也就是一个线程在执行时，其他线程就处于等待状态。注意，这里三个线程运行的实例对象是同一个类（都是 Client\$1 类的实例）。

那除了这一点不同之外，显式锁和内部锁还有什么不同呢？还有以下 4 点不同：

(1) Lock 支持更细粒度的锁控制

假设读写锁分离，写操作时不允许有读写操作存在，而读操作时读写可以并发执行，这一点内部锁就很难实现。显式锁的示例代码如下：

```
class Foo {
    // 可重入的读写锁
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    // 读锁
    private final Lock r = rwl.readLock();
    // 写锁
    private final Lock w = rwl.writeLock();

    // 多操作，可并发执行
    public void read() {
```

```

        try {
            r.lock();
            Thread.sleep(1000);
            System.out.println("read.....");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            r.unlock();
        }
    }
    // 写操作，同时只允许一个写操作
    public void write(Object _obj) {
        try {
            w.lock();
            Thread.sleep(1000);
            System.out.println("Writing.....");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            w.unlock();
        }
    }
}
}

```

可以编写一个 Runnable 的实现类，把 Foo 类作为资源进行调用（注意多线程是共享这个资源的），然后就会发现这样的现象：读写锁允许同时有多个读操作但只允许有一个写操作，也就是当有一个写线程在执行时，所有的读线程和写线程都会阻塞，直到写线程释放锁资源为止，而读锁则可以有多个线程同时执行。

(2) Lock 是无阻塞锁，synchronized 是阻塞锁

当线程 A 持有锁时，线程 B 也期望获得锁，此时，如果程序中使用的是显式锁，则 B 线程为等待状态（在通常的描述中，也认为此线程被阻塞了），若使用的是内部锁则为阻塞状态。

(3) Lock 可实现公平锁，synchronized 只能是非公平锁

什么叫非公平锁呢？当一个线程 A 持有锁，而线程 B、C 处于阻塞（或等待）状态时，若线程 A 释放锁，JVM 将从线程 B、C 中随机选择一个线程持有锁并使其获得执行权，这叫做非公平锁（因为它抛弃了先来后到的顺序）；若 JVM 选择了等待时间最长的一个线程持有锁，则为公平锁（保证每个线程的等待时间均衡）。需要注意的是，即使是公平锁，JVM 也无法准确做到“公平”，在程序中不能以此作为精确计算。

显式锁默认是非公平锁，但可以在构造函数中加入参数 true 来声明出公平锁，而 synchronized 实现的是非公平锁，它不能实现公平锁。

(4) Lock 是代码级的，synchronized 是 JVM 级的

Lock 是通过编码实现的，synchronized 是在运行期由 JVM 解释的，相对来说

`synchronized` 的优化可能性更高，毕竟是在最核心部分支持的，`Lock` 的优化则需要用户自行考虑。

显式锁和内部锁的功能各不相同，在性能上也稍有差别，但随着 JDK 的不断推进，相对来说，显式锁使用起来更加便利和强大，在实际开发中选择哪种类型的锁就需要根据实际情况考虑了：灵活、强大则选择 `Lock`，快捷、安全则选择 `synchronized`。

注意 两种不同的锁机制，根据不同的情况来选择。

建议 128：预防线程死锁

线程死锁（DeadLock）是多线程编码中最头疼的问题，也是最难重现的问题，因为 Java 是单进程多线程语言，一旦线程死锁，则很难通过外科手术式的方法使其起死回生，很多时候只有借助外部进程重启应用才能解决问题。我们看看下面的多线程代码是否会产生死锁：

```
class Foo implements Runnable{
    public void run(){
        // 执行递归函数
        fun(10);
    }
    // 递归函数
    public synchronized void fun(int i) {
        if (--i > 0) {
            for (int j = 0; j < i; j++) {
                System.out.print("*");
            }
            System.out.println(i);
            fun(i);
        }
    }
}
```

注意 `fun` 方法是一个递归函数，而且还加上了 `synchronized` 关键字，它保证同时只有一个线程能够执行，想想 `synchronized` 关键字的作用：当一个带有 `synchronized` 关键字的方法在执行时，其他 `synchronized` 方法会被阻塞，因为线程持有该对象的锁。比如有这样的代码：

```
static class Foo {
    public synchronized void m1() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // 异常处理
        }
    }
}
```

```

        System.out.println("m1 执行完毕");
    }
    public synchronized void m2() {
        System.out.println("m2 执行完毕");
    }
}

public static void main(String[] args) throws Exception {
    final Foo foo = new Foo();
    // 定义一个线程
    Thread t = new Thread(new Runnable() {
        public void run() {
            foo.m1();
        }
    });
    t.start();
    // 等待 t1 线程启动完毕
    Thread.sleep(10);
    // m2 方法需要等待 m1 执行完毕
    foo.m2();
}
}

```

相信读者明白会先输出“m1 执行完毕”，然后再输出“m2 执行完毕”，因为 m1 方法在执行时，线程 t 持有 foo 对象的锁，要想主线程获得 m2 方法的执行权限就必须等待 m1 方法执行完毕，也就是释放当前锁。明白了这个问题，我们思考一下上例中带有 synchronized 的递归函数是否能执行？会不会产生死锁？运行结果如下：

```

*****
*
*****
*
*****
*
*****
*
*****
*
*****
*
*****
*
```

一个倒三角形，没有产生死锁，正常执行，这是为何呢？很奇怪，是吗？那是因为在运行时当前线程（Thread-0）获得了 foo 对象的锁（synchronized 虽然是标注在方法上的，但实际作用的是整个对象），也就是该线程持有了 foo 对象的锁，所以它可以多次重入 fun 方法，也就是递归了。可以这样来思考该问题，一个宝箱有 N 把钥匙，分别由 N 个海盗持有（也就是我们 Java 中的线程了），但是同一时间只能由一把钥匙打开宝箱，获取宝物，只有在上一个海盗关闭了宝箱（释放锁）后，其他海盗才能继续打开锁获取宝物，这里还有一个规则：一旦一个海盗打开了宝箱，则该宝箱内的所有宝物对他来说都是开放的，即使是“宝箱中的宝箱”（即内箱）对他也是开放的。可以用以下代码来表述。

```

class Foo implements Runnable{
    public void run(){
        method1();
    }
    public synchronized void method1(){
        method2();
    }
    public synchronized void method2(){
        //Do Something
    }
}

```

方法 method1 是 synchronized 修饰的，方法 method2 也是 synchronized 修饰的，method1 调用 method2 是没有任何问题的，因为是同一个线程持有对象锁，在一个线程内多个 synchronized 方法重入完全是可行的，此种情况下不会产生死锁。

那什么情况下会产生死锁呢？看如下代码：

```

// 资源 A
static class A {
    public synchronized void a1(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " 进入 A.a1()");
        try {
            // 休眠 1 秒，仍然持有锁
            Thread.sleep(1000);
        } catch (Exception e) {
            // 异常处理
        }
        System.out.println(name + " 试图访问 B.b2()");
        b.b2();
    }
    public synchronized void a2() {
        System.out.println("进入 a.a2()");
    }
}
// 资源 B
static class B {
    public synchronized void b1(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " 进入 B.b1()");
        try {
            // 休眠 1 秒，仍然持有锁
            Thread.sleep(1000);
        } catch (Exception e) {
            // 异常处理
        }
        System.out.println(name + " 试图访问 A.a2()");
        a.a2();
    }
}

```

```

public synchronized void b2() {
    System.out.println("进入 B.b2()");
}
}

public static void main(String args[]) {
    final A a = new A();
    final B b = new B();
    // 线程 A
    new Thread(new Runnable() {
        public void run() {
            a.a1(b);
        }
    }, "线程 A").start();
    // 线程 B
    new Thread(new Runnable() {
        public void run() {
            b.b1(a);
        }
    }, "线程 B").start();
}
}

```

此段程序定义了两个资源 A 和 B，然后在两个线程 A、B 中使用了该资源，由于两个资源之间有交互操作，并且都是同步方法，因此在线程 A 休眠 1 秒钟后，它会试图访问资源 B 的 b2 方法，但是线程 B 持有该类的锁，并同时在等待 A 线程释放其锁资源，所以此时就出现了两个线程在互相等待释放资源的情况，也就是死锁了，运行结果如下：

```

线程 A 进入 A.a1()
线程 B 进入 B.b1()
线程 B 试图访问 A.a2()
线程 A 试图访问 B.b2()

```

此种情况下，线程 A 和线程 B 会一直互等下去，直到有外界干扰为止，比如终止一个线程，或者某一线程自行放弃资源的争抢，否则这两个线程就始终处于死锁状态了。我们知道要达到线程死锁需要四个条件：

- 互斥条件：一个资源每次只能被一个线程使用。
- 资源独占条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：线程已获得的资源在未使用完之前，不能强行剥夺。
- 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

只有满足了这些条件才可能产生线程死锁，这也同时告诫我们如果要解决线程死锁问题，就必须从这四个条件入手，一般情况下可以按照以下两种方式来解决：

(1) 避免或减少资源共享

一个资源被多个线程共享，若采用了同步机制，则产生的死锁可能性很大，特别是在项目比较庞大的情况下，很难杜绝死锁，对此最好的解决办法就是减少资源共享。

例如一个 B/S 结构的办公系统可以完全忽略资源共享，这是因为此类系统有三个特征：一是并发访问不会太高，二是读操作多于写操作，三是数据质量要求比较低，因此即使出现数据资源不同步的情况也不可能产生太大的影响，完全可以不使用同步技术。但是如果是一个支付清算系统就必须慎重考虑资源同步问题了，因为此类系统一是数据质量要求非常高（如果产生数据不同步的情况那可是重大生产事故），二是并发量大，不设置数据同步则会产生非常多的运算逻辑失效的情况，这会导致交易失败，产生大量的“脏”数据，系统可靠性将大大降低。

（2）使用自旋锁

回到前面的例子，线程 A 在等待线程 B 释放资源，而线程 B 又在等待线程 A 释放资源，僵持不下，那如果线程 B 设置了超时时间是不是就可以解决该死锁问题了呢？比如线程 B 在等待 2 秒后还是无法获得资源，则自行终结该任务，代码如下：

```
public void b2() {
    try {
        // 立刻获得锁，或者 2 秒等待锁资源
        if (lock.tryLock(2, TimeUnit.SECONDS)) {
            System.out.println("进入 B.b2()");
        }
    } catch (InterruptedException e) {
        // 异常处理
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

上面代码中使用 tryLock 实现了自旋锁（Spin Lock），它跟互斥锁一样，如果一个执行单元要想访问被自旋锁保护的共享资源，则必须先得到锁，在访问完共享资源后，也必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将“自旋”在那里，直到该自旋锁的保持者释放了锁为止。在我们的例子中就是线程 A 等待线程 B 释放锁，在 2 秒内不断尝试是否能够获得锁，达到 2 秒后还未获得锁资源，线程 A 则结束运行，线程 B 将获得资源继续执行，死锁解除。

对于死锁的描述最经典的案例是哲学家进餐（五位哲学家围坐在一张圆形餐桌旁，人手一根筷子，做以下两件事情：吃饭和思考。要求吃东西的时候停止思考，思考的时候停止吃东西，而且必须使用两根筷子才能吃东西），解决此问题的方法很多，比如引入服务生（资源调度）、资源分级等方法都可以很好地解决此类死锁问题。在我们 Java 多线程并发编程中，死锁很难避免，也不容易预防，对付它的最好办法是测试：提高测试覆盖率，建立有效的边界测试，加强资源监控，这些方法能使死锁无处遁形，即使发生了死锁现象也能迅速查找到

原因，提高系统的可靠性。

建议 129：适当设置阻塞队列长度

阻塞队列 BlockingQueue 扩展了 Queue、Collection 接口，对元素的插入和提取使用了“阻塞”处理，我们知道 Collection 下的实现类一般都采用了长度自行管理方式（也就是变长），比如这样的代码是可以正常运行的：

```
public static void main(String[] args) {
    // 定义初始长度为 5
    List<String> list = new ArrayList<String>(5);
    // 加入 10 个元素
    for(int i=0;i<10;i++){
        list.add("");
    }
}
```

上面的代码定义了列表的初始长度为 5，在实际使用时，当加入的元素超过初始容量时，ArrayList 会自行扩容，确保能够正常加入元素。那 BlockingQueue 也是集合，也实现了 Collection 接口，它的容量是否会自行管理呢？我们来看代码：

```
public static void main(String[] args) throws Exception {
    // 定义初始长度为 5
    BlockingQueue<String> bq = new ArrayBlockingQueue<String>(5);
    // 加入 10 个元素
    for (int i = 0; i < 10; i++) {
        bq.add("");
    }
}
```

BlockingQueue 能够自行扩容吗？答案是不能，运行结果如下：

```
Exception in thread "main" java.lang.IllegalStateException: Queue full
at java.util.AbstractQueue.add(AbstractQueue.java:71)
at java.util.concurrent.ArrayBlockingQueue.add(ArrayBlockingQueue.java:209)
at Client.main(Client.java:12)
```

没错，报队列已满异常，这是阻塞队列和非阻塞队列一个重要区别：阻塞队列的容量是固定的，非阻塞队列则是变长的。阻塞队列可以在声明时指定队列的容量，若指定的容量，则元素的数量不可超过该容量，若不指定，队列的容量为 Integer 的最大值。

阻塞队列和非阻塞队列有此区别的原因是阻塞队列是为了容纳（或排序）多线程任务而存在的，其服务的对象是多线程应用，而非阻塞队列容纳的则是普通的数据元素。我们来看一下 ArrayBlockingQueue 类最常用的 add 方法。

```

public class ArrayBlockingQueue<E> extends AbstractQueue<E>
implements BlockingQueue<E>, java.io.Serializable{
    // 容纳元素的数组
    private final E[] items;
    // 元素数量计数器
    private int count;
    public boolean add(E e) {
        // 调用 offer 方法尝试写入
        if (offer(e))
            return true;
        else
            // 写入失败，队列已满
            throw new IllegalStateException("Queue full");
    }

    public boolean offer(E e) {
        final ReentrantLock lock = this.lock;
        // 申请锁，只允许同时有一个线程操作
        lock.lock();
        try {
            // 元素计数器的计数与数组长度相同，表示队列已满
            if (count == items.length)
                return false;
            else { // 队列未满，插入元素
                insert(e);
                return true;
            }
        } finally {
            // 释放锁
            lock.unlock();
        }
    }
}
}

```

上面在加入元素时，如果判断出当前队列已满，则返回 false，表示插入失败，之后再包装成队列满异常。此处需要注意 offer 方法，如果我们直接调用 offer 方法插入元素，在超出容量的情况下，它除了返回 false 外，不会提供任何其他信息，如果我们的代码不做插入判断，那就会造成数据的“默默”丢失，这就是它与非阻塞队列的不同之处。

阻塞队列的这种机制对异步计算是非常有帮助的，例如我们定义深度为 100 的阻塞队列容纳 100 个任务，多个线程从该队列中获取任务并处理，当所有的线程都在繁忙，并且队列中任务数量已经为 100 时，也预示着系统运算压力非常巨大，而且处理结果的时间也会比较长，于是在第 101 个任务期望加入时，队列拒绝加入，而且返回异常，由系统自行处理，避免了异步运算的不可知性。但是如果应用期望无论等待多长时间都要运行该任务，不希望返回异常，那该怎么处理呢？

此时就需要用 BlockingQueue 接口定义的 put 方法了，它的作用也是把元素加入到队列

中，但它与 add、offer 方法不同，它会等待队列空出元素，再让自己加入进去，通俗地讲，put 方法提供的是一种“无赖”式的插入，无论等待多长时间都要把该元素插入到队列中，它的实现代码如下：

```
public void put(E e) throws InterruptedException {
    // 容纳元素的数组
    final E[] items = this.items;
    final ReentrantLock lock = this.lock;
    // 可中断锁
    lock.lockInterruptibly();
    try {
        try {
            // 队列满，等待其他线程移除元素
            while (count == items.length)
                notFull.await();
        } catch (InterruptedException ie) {
            // 被中断了，唤醒其他线程
            notFull.signal();
            throw ie;
        }
        // 插入元素
        insert(e);
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

put 方法的目的就是确保元素肯定会加入到队列中，问题是此种等待是一个循环，会不停地消耗系统资源，当等待加入的元素数量较多时势必会对系统性能产生影响，那该如何解决呢？JDK 已经想到了这个问题，它提供了带有超时时间的 offer 方法，其实现方法与 put 比较类似，只是使用 Condition 的 awaitNanos 方法来判断当前线程已经等待了多少纳秒，超时则返回 false。

与插入元素相对应，取出元素也有不同的实现，例如 remove、poll、take 等方法，对于此类方法的理解要建立在阻塞队列的长度固定的基础上，然后根据是否阻塞、阻塞是否超时等实际情况选用不同的插入和提取方法。

注意 阻塞队列的长度是固定的。

建议 130：使用 CountDownLatch 协调子线程

思考这样一个案例：百米赛跑，多个参加赛跑的人员在听到发令枪响后，开始跑步，到达终点后结束计时，然后统计平均成绩。这里有两点需要考虑：一是发令枪响，这是所有跑

步者（线程）接收到的出发信号，此处涉及裁判（主线程）如何通知跑步者（子线程）的问题；二是如何获知所有的跑步者完成了赛跑，也就是主线程如何知道子线程已经全部完成，这有很多种实现方式，此处我们使用 CountDownLatch 工具类来实现，代码如下：

```

static class Runner implements Callable<Integer> {
    // 开始信号
    private CountDownLatch begin;
    // 结束信号
    private CountDownLatch end;

    public Runner(CountDownLatch _begin, CountDownLatch _end) {
        begin = _begin;
        end = _end;
    }

    @Override
    public Integer call() throws Exception {
        // 跑步的成绩
        int score = new Random().nextInt(25);
        // 等待发令枪响起
        begin.await();
        // 跑步中 .....
        TimeUnit.MILLISECONDS.sleep(score);
        // 跑步者已经跑完全程
        end.countDown();
        return score;
    }
}

public static void main(String[] args) throws Exception {
    // 参加赛跑人数
    int num = 10;
    // 发令枪只响一次
    CountDownLatch begin = new CountDownLatch(1);
    // 参与跑步有多个
    CountDownLatch end = new CountDownLatch(num);
    // 每个跑步者一个跑道
    ExecutorService es = Executors.newFixedThreadPool(num);
    // 记录比赛成绩
    List<Future<Integer>> futures = new ArrayList<Future<Integer>>();
    // 跑步者就位，所有线程处于等待状态
    for (int i = 0; i < num; i++) {
        futures.add(es.submit(new Runner(begin, end)));
    }
    // 发令枪响，跑步者开始跑步
    begin.countDown();
    // 等待所有跑步者跑完全程
    end.await();
    int count = 0;
}

```

```

// 统计总分
for (Future<Integer> f : futures) {
    count += f.get();
}
System.out.println(" 平均分数为: " + count / num);
}

```

CountDownLatch 类是一个倒数的同步计数器，在程序中启动了两个计数器：一个是开始计数器 begin，表示的是发令枪；另外是结束计数器，一共有 10 个，表示的是每个线程的执行情况，也就是跑步者是否跑完比赛。程序执行逻辑如下：

- 1) 10 个线程都开始运行，执行到 begin.await 后线程阻塞，等待 begin 的计数变为 0。
- 2) 主线程调用 begin 的 countDown 方法，使 begin 的计数器为 0。
- 3) 10 个线程继续运行。
- 4) 主线程继续运行下一个语句，end 的计数器不为 0，主线程等待。
- 5) 每个线程运行结束时把 end 的计数器减 1，标志着本线程运行完毕。
- 6) 10 个线程全部结束，end 计数器为 0。
- 7) 主线程继续执行，打印出成绩平均值。

CountDownLatch 的作用是控制一个计数器，每个线程在运行完毕后会执行 countDown，表示自己运行结束，这对于多个子任务的计算特别有效，比如一个异步任务需要拆分成 10 个子任务执行，主任务必须要知道子任务是否完成，所有子任务完成后才能进行合并计算，从而保证了一个主任务的逻辑正确性。这和我们的实际工作非常类似，比如领导安排了一个大任务给我，我一个人不可能完成，于是我把该任务分解给 10 个人做，在 10 个人全部完成后，我把这 10 个结果组合起来返回给领导——这就是 CountDownLatch 的作用。

建议 131：CyclicBarrier 让多线程齐步走

思考这样一个案例：两个工人从两端挖掘隧道，各自独立奋战，中间不沟通，如果两人在汇合点处碰头了，则表明隧道已经挖通。这描绘的也是在多线程编程中，两个线程独立运行，在没有线程间通信的情况下，如何解决两个线程汇集在同一原点的问题。Java 提供了 CyclicBarrier（关卡，也有翻译为栅栏）工具类来实现，代码如下：

```

static class Worker implements Runnable {
    // 关卡
    private CyclicBarrier cb;
    public Worker(CyclicBarrier _cb) {
        cb = _cb;
    }
    public void run() {
        try {

```

```

        Thread.sleep(new Random().nextInt(1000));
System.out.println(Thread.currentThread().getName() + "- 到达汇合点");
        // 到达汇合点
        cb.await();
    } catch (Exception e) {
        // 异常处理
    }
}
}

public static void main(String[] args) throws Exception {
    // 设置汇集数量, 以及汇集完成后的任务
    CyclicBarrier cb = new CyclicBarrier(2, new Runnable() {
        public void run() {
            System.out.println("隧道已经打通! ");
        }
    });
    // 工人 1 挖隧道
    new Thread(new Worker(cb), "工人 1").start();
    // 工人 2 挖隧道
    new Thread(new Worker(cb), "工人 2").start();
}
}

```

在这段程序中, 定义了一个需要等待 2 个线程汇集的 CyclicBarrier 关卡, 并且定义了完成汇集后的任务 (输出“隧道已经打通!”), 然后启动了 2 个线程 (也就是 2 个工人) 开始执行任务。代码逻辑如下:

- 1) 2 个线程同时开始运行, 实现不同的任务, 执行时间不同。
- 2) “工人 1”线程首先到达汇合点 (也就是 cb.await 语句), 转变为等待状态。
- 3) “工人 2”线程到达汇合点, 满足预先的关卡条件 (2 个线程到达关卡), 继续执行。此时还会额外的执行两个动作: 执行关卡任务 (也就是 run 方法) 和唤醒“工人 1”线程。
- 4) “工人 1”线程继续执行。

CyclicBarrier 关卡可以让所有线程全部处于等待状态 (阻塞), 然后在满足条件的情况下继续执行, 这就好比是一条起跑线, 不管是如何到达起跑线的, 只要到达这条起跑线就必须等待其他人员, 待人员到齐后再各奔东西, CyclicBarrier 关注的是汇合点的信息, 而不在乎之前或之后做何处理。

CyclicBarrier 可以用在系统的性能测试中, 例如我们编写了一个核心算法, 但不能确定其可靠性和效率如何, 我们就可以让 N 个线程汇集到测试原点上, 然后“一声令下”, 所有的线程都引用该算法, 即可观察出算法是否有缺陷。



第 10 章

性能和效率

快，快点，再快点……大脑已经跟不上鼠标！

——佚名

在这个快餐时代，系统一直在提速，从未停步过，从每秒百万条指令的 CPU 到现在的每秒万亿条指令的多核 CPU，从最初发布一个帖子需要等待 N 小时才有回复到现在的微博，一个消息在几分钟内就可以传遍全球；从 N 天才能完成的一次转账交易，到现在的即时转账——我们进入了一个光速发展的时代，我们享受着，也在被追逐着——榨干硬件资源，加速所有能加速的，提升所有能提升的。

建议 132：提升 Java 性能的基本方法

Java 从诞生之日起就被质疑：字节码在 JVM 中运行是否会比机器码直接运行的效率会低很多？很多技术高手、权威网站都有类似的测试和争论，从而来表明 Java 比 C（或 C++）更快或效率相同。此类话题我们暂且不表（这类问题的争论没完没了，也许等到我们退休的时候，还想找个活动脑筋的方式，此类问题就会是最好的选择），我们先从如何提高 Java 的性能方面入手，看看怎么做才能让 Java 程序跑得更快，效率更高，吞吐量更大。

（1）不要在循环条件中计算

如果在循环（如 for 循环、while 循环）条件中计算，则每循环一遍就要计算一次，这会降低系统效率，就比如这样的代码：

```
// 每次循环都要计算 count*2
while(i<count*2){
    //Do Something
}
```

应该替换为：

```
// 只计算一遍
int total = count * 2;
while(i<total){
    //Do Something
}
```

（2）尽可能把变量、方法声明为 final static 类型

假设要将阿拉伯数字转换为中文数字，其定义如下：

```
public String toChineseNum(int num) {
    // 中文数字
    String[] cns = {"零", "壹", "贰", "叁", "肆", "伍", "陆", "柒", "捌", "玖"};
    return cns[num];
}
```

每次调用该方法时都会重新生成一个 cns 数组，注意该数组不会改变，属于不变数组，在这种情况下，把它声明为类变量，并且加上 final static 修饰会更合适，在类加载后就生成了该数组，每次方法调用则不再重新生成数组对象了，这有助于提高系统性能，代码如下。

```
// 声明为类变量
final static String[] cnz = {"零", "壹", "贰", "叁", "肆", "伍", "陆", "柒", "捌", "玖"};
public String toChineseNum(int num) {
    return cnz[num];
}
```

(3) 缩小变量的作用范围

关于变量，能定义在方法内的就定义在方法内，能定义在一个循环体内的就定义在循环体内，能放置在一个 try……catch 块内的就放置在该块内，其目的是加快 GC 的回收。

(4) 频繁字符串操作使用 StringBuilder 或 StringBuffer

虽然 String 的联接操作（“+”号）已经做了很多优化，但在大量的追加操作上 StringBuilder 或 StringBuffer 还是比“+”号的性能好很多，例如这样的代码：

```
String str = "Log file is ready.....";
for (int i = 0; i < max; i++) {
    // 此处生成三个对象
    str += "log " + i;
}
```

应该修改为：

```
StringBuilder sb = new StringBuilder(20000);
sb.append("Log file is ready.....");
for (int i = 0; i < max; i++) {
    sb.append("log " + i);
}
String log = sb.toString();
```

(5) 使用非线性检索

如果在 ArrayList 中存储了大量的数据，使用 indexOf 查找元素会比 java.util.Collections.binarySearch 的效率低很多，原因是 binarySearch 是二分搜索法，而 indexOf 使用的是逐个元素比对的方法。这里要注意：使用 binarySearch 搜索时，元素必须进行排序，否则准确性就不可靠了。

(6) 覆写 Exception 的 fillInStackTrace 方法

我们在第 8 章中提到 fillInStackTrace 方法是用来记录异常时的栈信息的，这是非常耗时的动作，如果我们在开发时不需要关注栈信息，则可以覆盖之，如下覆盖 fillInStackTrace 的自定义异常会使性能提升 10 倍以上：

```
class MyException extends Exception {
    public Throwable fillInStackTrace() {
        return this;
    }
}
```

(7) 不建立冗余对象

不需要建立的对象就不能建立，说起来很容易，要完全遵循此规则难度就很大了，我们经常就会无意地创建冗余对象，例如这样一段代码：

```
public void doSomething() {
    // 异常信息
    String exceptionMsg = "我出现异常了，快来救救我！";
    try {
        Thread.sleep(10);
    } catch (Exception e) {
        // 转换为自定义运行期异常
        throw new MyException(e, exceptionMsg);
    }
}
```

注意看变量 exceptionMsg，这个字符串变量在什么时候会被用到？只有在抛出异常时它才有用武之地，那它是什么时候创建的呢？只要该方法被调用就创建，不管会不会抛出异常。我们知道异常不是我们的主逻辑，不是我们代码必须或经常要到达的区域，那为了这个不经常出现的场景就每次都多定义一个字符串变量，合适吗？而且还要占用更多的内存！所以，在 catch 块中定义 exceptionMsg 方法才是正道：需要的时候才创建对象。

我们知道运行一段程序需要三种资源：CPU、内存、I/O，提升 CPU 的处理速度可以加快代码的执行速度，直接表现就是返回时间缩短了，效率提高了；内存是 Java 程序必须考虑的问题，在 32 位的机器上，一个 JVM 最多只能使用 2GB 的内存，而且程序占用的内存越大，寻址效率也就越低，这也是影响效率的一个因素。I/O 是程序展示和存储数据的主要通道，如果它很缓慢就会影响正常的显示效果。所以我们在编码时需要从这三个方面入手接口（当然了，任何程序优化都是从这三方面入手的）。

Java 的基本优化方法非常多，这里不再罗列，相信读者也有自己的小本本，上面所罗列的性能优化方法可能远比这里多，但是随着 Java 的不断升级，很多看似很正确的优化策略就逐渐过时了（或者说已经失效了），这一点还需要读者注意。最基本的优化方法就是自我验证，找出最佳的优化途径，提高系统性能，不可盲目信任。

建议 133：若非必要，不要克隆对象

通过 clone 方法生成一个对象时，就会不再执行构造函数了，只是在内存中进行数据块的拷贝，此方法看上去似乎应该比 new 方法的性能好很多，但是 Java 的缔造者们也认识到“二八原则”，80%（甚至更多）的对象是通过 new 关键字创建出来的，所以对 new 在生成对象（分配内存、初始化）时做了充分的性能优化，事实上，一般情况下 new 生成的对象比 clone 生成的性能方面要好很多，例如这样的代码。

```

private static class Apple implements Cloneable {
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new Error();
        }
    }
}

public static void main(String[] args) {
    // 循环 10 万次
    final int maxLoops = 10 * 10000;
    int loops = 0;
    // 开始时间
    long start = System.nanoTime();
    // "母" 对象
    Apple apple = new Apple();
    while (++loops < maxLoops) {
        apple.clone();
    }
    long mid = System.nanoTime();
    System.out.println("clone 方法生成对象耗时: " + (mid - start) + " ns");
    // new 生成对象
    while (--loops > 0) {
        new Apple();
    }
    long end = System.nanoTime();
    System.out.println("new 生成对象耗时: " + (end - mid) + " ns");
}
}

```

在上面的代码中，Apple 是一个简单的可拷贝类，用两种方式生成了 10 万个苹果：一种是通过克隆技术，一种是通过直接种植（也就是 new 关键字），按照我们的常识想当然地会认为克隆肯定比 new 要快，但是结果却是这样的：

```

clone 方法生成对象耗时: 18731431 ns
new 生成对象耗时: 2391924 ns

```

不用看具体的数字，数数位数就可以了：clone 方法花费的时间是 8 位数，而 new 方法是 7 位数，用 new 生成对象比 clone 方法快很多！原因是 Apple 的构造函数非常简单，而且 JVM 对 new 做了大量的性能优化，而 clone 方式只是一个冷僻的生成对象方式，并不是主流，它主要用于构造函数比较复杂，对象属性比较多，通过 new 关键字创建一个对象比较耗时间的时候。

注意 克隆对象并不比直接生成对象效率高。

建议 134：推荐使用“望闻问切”的方式诊断性能

“望闻问切”是中医诊断疾病的必经步骤，“望”是指观气色，“闻”是指听声息，“问”是指询问症状，“切”是指摸脉象，合称“四诊”，经过这四个步骤，大夫基本上就能确认病症所在，然后加以药物调理，或能还以病人健康身躯。

一个应用系统如果出现性能问题，不管是偶发性问题还是持久性问题，都是系统“生病”的表现，需要工程师去诊断，然后对症下药。我们可以把 Java 的性能诊断也分为此四个过程（把我们自己想象成医生吧，只是我们的英文名字不叫 Doctor，而是叫做 Trouble Shooter）：

(1) 望

观察性能问题的症状。有人投诉我们开发出的系统性能慢，如蜗牛爬行，执行一个操作，在等待它返回的过程中，用户已经完成了倒水、喝茶、抽烟等一系列消遣活动，但系统还是没返回结果！其实这是个好现象，至少我们能看到症状，从而可以对症下药。性能问题从表象上来看可以分为两类：

□ 不可（或很难）重现的偶发性问题

比如线程阻塞，在某种特殊条件下，多个线程访问共享资源时会被阻塞，但不会形成死锁，这种情况很难去重现，当用户打电话投诉时，我们自己赶到现场症状已经消失了，然后1个月内再也没有出现过，当我们都认为“磨合”期已过，系统已经正常运行的时候，又接到了类似的投诉，崩溃呀！对于这种情况，“望”已经不起作用了，不要为了看到症状而花费大量的时间和精力，可以采用后续提到的“闻问切”方式。

□ 可重现的性能问题

客户打电话给我们，反映系统性能缓慢，不需要我们赶到现场，自己观察一下生产机就可以发现部分交易缓慢，CPU 过高，可用内存较低等问题，在这种情况下我们至少要测试三个有性能问题的交易（或者三个与业务相关而技术无关的功能，或者与技术有关而业务无关的功能），为什么是三个呢？因为“永远不要带两块手表”，这会致使无法验证和校对。

比如三个不同的输入功能，都是用户输入信息，然后保存到数据库中，但是三个交易的性能都非常缓慢，通过初步的“望”我们就可以基本确认是与数据库或数据驱动相关的问题；若是只有一个交易缓慢，其他两个正常，那就可以大致定位到一个面：该交易的逻辑层出现问题。

(2) 闻

中医上的“闻”是大夫听（或嗅）患者不自觉发出的声音和气味，在性能优化上的“闻”则是关注项目被动产生的信息，其中包括：项目组的技术能力（主要取决于技术经理的技术能力）、文化氛围、群体的习惯和习性，以及他们专注和擅长的领域等，各位读者可能要疑惑了：中医上“闻”的对象是病人，而为什么这里“闻”的对象却是开发团队呢？

我们这样来思考该问题，如果是一个人（个体）生病了，找大夫如此处理是没有任何问题的，但是如果说是人类（群体）生病了，那如何追寻这个根源呢？假设人是上帝创造的，如果有一群外星生物说“人类都有自私的缺陷”，那是不是应该去观察一下上帝？了解这个缺陷是源于他的习惯性动作还是技能缺乏，或者是“文化传承”。对于一个 Java 应用来说，我们就是“上帝”，我们创造了他，给了他生命（能够运行），给了他尊严（用户需要它），给了他灵魂（解决了业务问题），那一旦他生病，是不是应该审视一下我们这些“上帝”呢？或者我们得自我反省一下呢？

如果项目组的技术能力很强，有资深的数据库专家，有顶尖的架构师，也有首席程序员，那性能问题产生的根源就应该定位在无意识的代码缺陷上。

如果项目组的文化氛围很糟糕，组员不交流，没有固定的代码规范，缺乏整体的架构等，那性能问题的根源就可能存在于某个配置上，或者相互的接口调用上。

如果项目组已经习惯了某一个框架，而且也习惯了框架的种种约束，那性能的根源就可能是有人越过了框架的协约。

需要注意的是，“闻”并不是主动地去了解，而是由技术（人或应用）自行挥发出的“味道”，需要我们要敏锐地抓住，这可能会对性能分析有非常大的帮助。

（3）问

“问”就是与技术人员（缔造者）和业务人员（使用者）一起探讨该问题，了解性能问题的历史状况，了解“慢”产生的前因后果，比如对于业务人员我们可以咨询：

性能是不是一直这样慢，从何时起慢到不能忍受？

哪一个操作或哪一类操作最慢，大概的等待时间是多长？

用户的操作习惯是什么，是喜欢快捷键还是喜欢用鼠标点击？

在什么时间段最慢，业务高峰期是否有滞顿现象，业务低谷是否也缓慢？

其他访问渠道，如移动设备是否也有效率问题？

业务品种和数量有没有激增，操作人员是否大规模增加？

是否在业务上发生过重大事项或重要变更，当时的性能如何？

用户的操作习惯有没有改变，或者用户是否自定义了某些功能？

而对于技术人员，我们就要从技术角度来询问性能问题了，而且由于技术人员对系统了如指掌，可能会“无意识”地回避问题，我们应该有技巧地处理这类问题，例如可以这样来询问技术人员：

系统日志是否记录了缓慢信息，是否可以回放缓慢交易？

缓慢时系统的 CPU、内存、I/O 如何？

高峰期和低谷时业务并发数量、并发交易种类、连接池的数量、数据的连接数量如何？

最早接到用户投诉是什么时候，是如何处理的，优化后如何？

数据量的增长幅度如何，是否有历史数据处理策略？

系统是否有不稳定的情况，是否出现过宕机，是否产生过 `javacore` 文件？
最后一次变更何时，变更的内容是哪些，变更后是否出现过性能问题？
操作系统、网络、存储、应用软件等环境是否发生过改变？
通过与技术人员和业务人员交流，我们可以对性能问题有一个整体认识，避免“管中窥豹，只见一斑”的偏见，更加有助于我们分析和定位问题。

(4) 切

“切”是“四诊”的最后一个环节，也是最重要的环节，这个环节结束我们就要给出定论：问题出在什么地方，该如何处理等。Java 的性能诊断也是类似的，“切”就要我们接触真实的系统数据，需要去看设计，看代码，看日志，看系统环境，然后是思考分析，最后给出结论。在这一环节中，需要注意两点：一是所有的非一手资料（如报告、非系统信息）都不是 100% 可信的，二是测试环境毕竟是测试环境，它只是证明假设的辅助工具，并不能证明方法或策略的正确性。

曾经遇到过这样一个案例，有一个 24 小时运行的高并发系统，从获得的资料上看，在出现偶发性的性能故障前系统没有做过任何变更，网络也没变更过，业务也没有过大的变动，业务人员的形容是“一夜之间系统就变慢了”，而且该问题在测试机上不能模拟重现。接到任务后，马上进行“望闻问”，都没有太大的收获。进入到“切”环节时，对大量的日志进行跟踪分析调试，最终锁定到了加密机上：加密机属于多个系统的共享资源，当排队加密数据时就有可能出现性能问题，最终的解决方案是增加一台加密机，于是系统性能恢复正常。

性能优化是一个漫长的工作，特别是对于偶发性的性能问题，不要期望找到“名医”立刻就能见效，这是不现实的，深入思考，寻根探源，最终必然能找到根源所在。中医上有一句话“病来如山倒，病去如抽丝”，系统诊断也应该这样一个过程，切忌急躁。

注意 性能诊断遵循“望闻问切”，不可过度急躁。

建议 135：必须定义性能衡量标准

出现性能问题不可怕，可怕的是没有目标，用户只是说“我希望它非常快”，或者说“和以前一样快”，在这种情况下，我们就需要把制定性能衡量标准放在首位了，原因有两个：

(1) 性能衡量标准是技术与业务之间的契约

“非常快”是一个直观性的描述，它不具有衡量的可能性，对技术人员来说，一个请求在 2 秒钟之内响应就可以认为是“非常快”了，但对业务人员来说，“非常快”指的是在 0.5 秒内看到结果——看，出现偏差了。如果不解决这种偏差，就有可能出现当技术人员认

为优化结束的时候，而业务人员还认为系统很慢，仍然需要提高继续性能，于是拒不签收验收文档，这就产生商务麻烦了。

(2) 性能衡量标志是技术优化的目标

性能优化是无底线的，性能优化得越厉害带来的副作用也就明显，例如代码的可读性差，可扩展性降低等，比如一个乘法计算，我们一般是这样写代码的：

```
int i = 100 * 16;
```

如果我们为了提升系统性能，使用左移的方式来计算，代码如下：

```
int i = 100 << 4;
```

性能确实提高了，但是也带来了副作用，比如代码的可读性降低了很多，要想让其他人员看明白这个左移是何意，就需要加上注释说“把 100 扩大 16 倍”，这在项目开发中是非常不合适的。因此为了让我们的代码保持优雅，减少“坏味道”的产生，就需要定义一个优化目标：优化到什么地步才算结束。

明白了性能标准的重要性，就需要在优化前就制定好它，一个好的性能衡量标准应该包括以下 KPI (Key Performance Indicators)：

- 核心业务的响应时间。一个新闻网站的核心业务就是新闻浏览，它的衡量标准就是打开一个新闻的时间；一个邮件系统的核心业务就是邮件发送和接收速度；一个管理型系统的核心就是流程提交，这也就是它的衡量标准。
- 重要业务的响应时间。重要业务是指在系统中占据前沿地位的业务，但是不会涉及业务数据的功能，例如一个业务系统需要登录后才能操作核心业务，这个登录交易就是它的重要交易，比如邮件系统的登录。

当然，性能衡量标准必须在一定的环境下，比如网络、操作系统、硬件设备等确定的情况下才会有意义，并且还需要限定并发数、资源数（如 10 万数据和 1000 万的数据响应时间肯定不同）等，当然很多时候我们并没有必要白纸黑字地签署一份协约，我们编写性能衡量标准更多地是为了确定一个目标，并尽快达到业务要求而已。

建议 136：枪打出头鸟——解决首要系统性能问题

在一个系统出现性能问题的时候，很少会出现只有一个功能有性能问题（一个功能出现性能问题的情况非常容易解决，基本上不会花费什么时间），系统一旦出现性能问题，也就意味着一批的功能都出现了问题，在这种情况下，我们要做的就是统计出业务人员认为重要而且缓慢的所有功能，然后按照重要优先级和响应时间进行排序，并找出前三名，而这就是我们要找的“准出头鸟”。

“准出头鸟”找到了，然后再对这三个功能进行综合分析，运行“望闻问切”策略，找到问题的可能根源，然后只修正第一个功能的性能缺陷，再来测试检查是否解决了这个问题，紧接着是第二个、第三个，循环之。可能读者会产生疑问：为什么这里只修正第一个缺陷，而不是三个一起全部修正？这是因为第一个性能缺陷才是我们真正的出头鸟，在我做过的性能优化项目中超过 80% 的只要修正了第一个缺陷，其他的性能问题就会自行解决或非常容易解决，已经不成为问题了。

比如 BBS 系统，从用户登录到用户浏览、发帖都非常缓慢，经过逐步筛选，确定登录就是“出头鸟”，需要着重解决，代码如下：

```
class Login extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse resp){
        // 从 req 中获得用户名和密码
        String userName = ....;
        String passwd = ....;
        // 由登录逻辑处理登录
        boolean loginSuccess = loginBiz.login(userName, passwd);
        if(loginSuccess){
            // 登录成功，签到
            Checker.sign(userName);
        }else{
            // 登录不成功，记录日志
            log.warn(userName + ", 登录失败!")
        }
    }
}
```

这是一个早期 Web 应用的典型验证代码，先验证用户是否登录成功，然后决定是否向 Session 中写入信息。在该 BBS 系统中，分析发现 login 和 sign 操作都非常耗时，那就首先跟踪 login，代码如下：

```
class LoginBiz{
    public void login(String _user, String _password){
        // 根据用户名获得用户对象
        User u = userDao.getUserByUser(_user);
        return u != null && u.getPasswd().equals(_password);
    }
}
```

追踪到这里，发现从数据中取出用户对象的效率很低，但是数据库的 CPU、内存、I/O 都没有问题，而且没有到达最大连接数。继续追踪下去，终于发现问题了：数据库的版本和 JDBC 的版本不一致，虽然在进行所有的连接、执行 SQL、断开等操作时都没有出现任何问题，但在多表的联合查询中速度非常慢。问题定位了，将其替换成数据库匹配的驱动程序，登录问题马上得到解决，并且其他所有性能慢的问题都解决了，归根结底其实都是数据库驱动问题引起的。

解决性能问题时，不要把所有的问题都摆在眼前，这只会“扰乱”你的思维，集中精力，找到那个“出头鸟”，解决它，在大部分情况下，一批性能问题都会迎刃而解，而且我们的用户关注最多的可能就是系统 20% 的功能，可能我们解决了这一部分，已经达到了用户的预期目标，也就标志着我们的优化工作可以结束了。

注意 解决性能优化要“单线程”小步前进，避免关注点过多而导致精力分散。

建议 137：调整 JVM 参数以提升性能

我们写的每一段 Java 程序都要在 JVM 中运行，如果程序已经优化到了极致，但还是觉得性能比较低，那 JVM 的优化就要提到日程上来了。不过，由于 JVM 又是系统运行的容器，所以稳定性也是必须考虑的，过度的优化可能就会导致系统故障频繁发生，致使系统质量大幅下降。下面提供了四个常用的 JVM 优化手段，供你在需要时参考。

(1) 调整堆内存大小

我们知道，在 JVM 中有两种内存：栈内存（Stack）和堆内存（Heap），栈内存的特点是空间比较小，速度快，用来存放对象的引用及程序中的基本类型；而堆内存的特点是空间比较大，速度慢，一般对象都会在这里生成、使用和消亡。

栈空间是由线程开辟，线程结束，栈空间由 JVM 回收，因此它的大小一般不会对性能有太大的影响，但是它会影响系统的稳定性，在超过栈内存的容量时，系统会报 StackOverflowError 错误。可以通过“`java -Xss <size>`”设置栈内存大小来解决此类问题。

堆内存的调整不能太随意，调整得太小，会导致 Full GC 频繁执行，轻则导致系统性能急速下降，重则导致系统根本无法使用；调整得太大，一则浪费资源（当然，若设置了最小堆内存则可以避免此问题），二则是产生系统不稳定的情况，例如在 32 位的机器上设置超过 1.8GB 的内存就有可能产生莫名其妙的错误。设置初始化堆内存为 1GB（也就是最小堆内存），最大堆内存为 1.5GB 可以用如下的参数：

```
java -Xmx1536m -Xms1024m
```

(2) 调整堆内存中各分区的比例

JVM 的堆内存包括三部分：新生区（Young Generation Space）、养老区（Tenure generation space）、永久存储区（Permanent Space），其中新生成的对象都在新生区，它又分为伊甸区（Eden Space）、幸存 0 区（Survivor 0 Space）和幸存 1 区（Survivor 1 Space），当在程序中使用了 new 关键字时，首先在伊甸区生成该对象，如果伊甸区满了，则用垃圾回收器先进行回收，然后把剩余的对象移动到幸存区（0 区或 1 区），可如果幸存区也满了呢？垃圾回收器会再回收一次，然后再把剩余的对象移动到养老区，那要是养老区也满了呢？此时

就会触发 Full GC（这是一个非常危险的动作，JVM 会停止所有的执行，所有系统资源都会让位给垃圾回收器），会对所有的对象过滤一遍，检查是否有可以回收的对象，如果还是没有的话，就抛出 OutOfMemoryError 错误，系统不干了！

清楚了这个原理（若还是不清楚，请看看《JVM Specification》），那我们就可以思考一下如何提升性能了：若扩大新生区，势必会减少养老区，这就可能产生不稳定的情况，一般情况下，新生区和养老区的比例为 1:3 左右，设置命令如下：

```
java -XX:NewSize=32m -XX:MaxNewSize=640m -XX:MaxPermSize=1280m -XX:NewRatio=5
```

该配置指定新生代初始化为 32MB（也就是新生区最小内存为 32M），最大不超过 640MB，养老区最大不超过 1280MB，新生区和养老区的比例为 1:5。

（3）变更 GC 的垃圾回收策略

Java 程序性能的最大障碍就是垃圾回收，我们不知道它何时会发生，也不知道它会执行多长时间，但是我们可以想办法改变它对系统的影响，比如启用并行垃圾回收、规定并行回收的线程数量等，命令格式如下：

```
java -XX:+UseParallelGC -XX:ParallelGCThreads=20
```

这里启用了并行垃圾收集机制，并且定义了 20 个收集线程（默认的收集线程等于 CPU 的数量），这对多 CPU 的系统是非常有帮助的，可以大大减少垃圾回收对系统的影响，提高系统性能。

当然，垃圾回收的策略还有很多属性可以修改，比如 UseSerialGC（启用串行 GC，默认值）、ScavengeBeforeFullGC（新生代 GC 优先于 Full GC 执行）、UseConcMarkSweepGC（对老生代采用并发标记交换算法进行 GC）等，这些参数需要在系统中逐步调试。

（4）更换 JVM

如果所有的 JVM 优化都不见效，那只有使用最后一招了：更换 JVM，目前市面上比较流行的 JVM 有三个产品：Java HotSpot VM、Oracle JRockit JVM、IBM JVM，其中 HotSpot 是我们经常使用的，稳定性、可靠性都不错；JRockit 则以效率著称，性能是它的优势，但在决定使用该 JVM 之前一定要做好全面的系统测试，它的某些行为可能会在 JRockit 上产生 Bug；IBM JVM 也比较稳定，而且它在 AIX 系统上的表现要远远好于其他操作系统。

JVM 的优化不能像程序优化一样，找到 Bug 就可以立刻解决，JVM 的优化一定是要循序渐进的，参数设置不可激进，特别是需要优化多个参数时，一定要逐步实施，确保每个优化步骤都达到了预期目标，否则会对整个系统的稳定性产生较大的风险。需要提醒的是以上带有“-XX”的 JVM 参数可能是不健壮的，SUN 也不推荐使用，可能后续会在没有通知的情况下就不再支持它了，但是它又非常好用，这需要在系统升级、迁移时慎重考虑。

建议 138：性能是个大“咕咚”

有一部动画片叫《咕咚来了》，其大致剧情是：三只小兔在湖边玩耍，忽然湖中传来“咕咚”一声，这奇怪的声音把小兔们吓了一大跳。小兔们刚想去看个究竟，又听到“咕咚”一声，这可把小兔们吓坏了，“快跑，咕咚来了，快逃呀！”小兔们转身就跑。

狐狸正同小鸟跳舞，与跑来的兔子碰了个满怀。狐狸一听“咕咚来了！”也紧张起来，跟着就跑。它们又惊醒了睡觉的小熊和树上的小猴，小熊和小猴也不问青红皂白，跟着它们跑起来，于是一路上跟着跑的动物越来越多，大象、河马、老虎、野猪……

岸上这阵骚乱，让湖中的青蛙感到十分惊讶，它拦住了这群吓蒙了的伙伴们，问出了什么事，大家七嘴八舌地形容“咕咚”是个多么可怕的怪物。青蛙问：“谁见到了？”大家互相推诿，谁也没有亲眼看见，于是决定回去看看明白。

回到湖边，又听见“咕咚”一声，仔细一看，原来是木瓜掉进水里发出的声音，众动物不禁大笑起来。

这寓言故事好笑吗？很好笑，但是要是发生在我们自己身上就不那么好笑了，比如说，某些 Javaer 一直在质疑 Java 系统的性能，于是我们自己也跟着怀疑 Java 的性能——这就是发生在我们身边的真实“咕咚”，Java 系统的性能问题本就是子虚乌有的事情，是我们自己吓唬自己，其实，我们可以从四个方面分析该问题：

(1) 没有慢的系统，只有不满足业务的系统

不管是使用 C 开发还是 Java 开发的项目，最终都会有一个产品诞生，或服务于大众（如网站），或服务于企业（企业级应用），谁来决定一个系统的快慢呢？不是计算机，它只会使用毫秒、纳秒去记录时间但不会做判断，它可以计算出一个交易执行了多长时间，但它不能决定这个时间是长还是短，那谁去判断呢？是人，准确地说是使用者，即使是开发人员自己根据日志记录的时间来判断系统是慢了还是快了，那也还是以使用者的身份来判断的，对一个系统毫无了解的人员是无法判断出一个系统的快慢的。

例如一个做统计的业务人员去看计费系统，即使响应需要 N 秒的时间，统计人员也会觉得非常快了，那是因为统计系统的结果经常是按照小时、天来计算的。再比如即时通信系统，有 1 秒内的延迟是可以接受的（发送者发出消息到接受者接收消息的时间间隔为 1 秒），但是语音通信系统若有 1 秒的延迟就是不可接受的了；发送邮件 N 分钟后才收到，这是可以容忍的，但是对于同城银行内转账来说，这个时间就是不可容忍的，必须在秒级完成。不同的系统所要求的性能不同，因此只要一个系统达到业务要求就可以认为它足够快，我们不要期望跨系统间的性能对比，这是毫无意义的。

如果有使用者告诉你，“这个系统太慢了”，也就是在间接地提醒您：系统没有满足业务需求，尚待继续努力。

(2) 没有慢的系统，只有架构不良的系统

在做系统架构设计时，架构师有没有考虑并行计算？有没有考虑云计算技术？有没有负载均衡？……这些都是解决我们性能问题的良方，只要架构设计得当，效率就不是问题。

即使是架构初期没有考虑扩展性，那我们也有一些手段可解决性能问题。比如有一个批处理系统，系统建设时的目标是：5小时内生成2000万条业务数据，可到第3年的时候，公司发生了大规模的变化（整合了其他同类公司），需要处理的数据更多了，在5小时内需要生成8000万业务。于是就得考虑架构的扩展了。有一个很简单的处理方案，即应用服务器水平扩展，增加业务数据源的纵向切割能力，均分数据压力，这样就可以很轻松地实现大数量的生成。

再比如，一个新闻网站，刚开始上线时访问的人员不多，响应都是在毫秒级别的，随着访问量的激增，响应时间呈阶梯型增加，资深会员流失率翻倍跳跃，如何解决该问题呢？解决方案有两个：一是增加IP层的负载均衡，或者硬件设备，或者软件架构，把访问者分配到多个不同的应用服务器上，降低单台应用服务器的性能压力；二是增强系统的处理能力，增大吞吐量，比如提升数据源的响应能力，划分数据的热度（如把数据划分为Hot、Warm、Cold等区域，分配不同的硬件资源和服务等级），很多时候这两个方案配合起来使用，会很快解决性能问题。

(3) 没有慢的系统，只有懒惰的技术人员

这里的技术人员涉及面很大，可以是开发人员，也可以是维护人员，甚至是应用软件的顾问人员（如数据库顾问、App Server的顾问）等。一个系统出现问题，或者是投产前后立刻出现的性能问题，或者是运行中突发的性能问题，或者是逐渐增长的数据（用户或业务数据）导致的性能问题，只要我们肯用心查找，并且拥有适当的资源（如源码和支持资源），一般都是可以解决的。最可怕的是我们的技术人员对性能问题漠不关心，对时间效率不够敏感，导致使用者怨声载道，三人成虎，最终致使此系统成为一个“慢得无法使用的系统”。

这也要求我们在开发初期就适当考虑一下性能问题，但不要把性能排为头号任务，它不是，它只是我们的一个关注点而已。

(4) 没有慢的系统，只有不愿意投入的系统

这里的投入指的是资源，包括软硬件资源、人员资源及资金资源等，这不是项目组能够单独解决的问题，但是它会严重影响系统的性能。曾经遇到一个运行超过8年的分析系统，从1年前开始只要是高峰期它的速度就会慢下来，分析下来，发现是因为并发用户超过了许可的数量，造成系统阻塞，性能缓慢，唯一解决的法就是购买更多的许可数量，但是8年了，一个系统的生命期还能有多少呢？——所以最后采用了自由放任的办法，让其自行走到寿命的终结点，然后建立新的分析系统。

当然，我们也会碰到查不出原因的性能问题，这不可否认，毕竟现在的系统越做越大，源代码动辄就十万、百万级别，让一个人或一个小团队将其彻头彻尾地查清楚也不现实，而

且性能问题涉及面非常广，如操作系统、数据库、网络、存储等，要想对这些技术都非常熟悉也很困难，但查不出问题并不代表我们解决不了，是的，这与治疗癌症相似，我们现在的科学还不知道它的发病机理，不知道为什么会产生癌细胞，但我们知道割除病变部位能够避免癌细胞扩散，性能问题也一样：我们可能不知道问题产生的原因，但我们可以有 N 种手段来解决它。能够解决的问题还算是问题吗？

而且，性能只是衡量系统的一个辅助指标，而不是主指标，如果您与业务人员交流，说“我们可以把系统的响应时间提升到 0.001 秒内，但前提是不实现您提出的需求”，您猜业务人员会同意吗？——不把我们这些“火星人”撵出门外已经算是客气的了！

注意 对现代化的系统建设来说，性能就是一个大“咕咚”——看清它的本质吧。



第 11 章

开 源 世 界

You deserve to be able to cooperate openly and freely with other people who use software. You deserve to be able to learn how the software works, and to teach your students with it. You deserve to be able to hire your favorite programmer to fix it when it breaks.

You deserve free software.

你可以公开、自由地与其他软件使用者合作，你有权了解软件的工作原理，并将其传授给你的学生，当软件发生问题时你完全可以雇用你所喜爱的程序员对它进行完善。

你理应得到自由的软件。

——Richard Matthew Stallman (理查·马修·斯托曼，自由软件运动的精神领袖)

很难想象一个项目不使用开源产品的情形，所有的框架都自己写，所有的工具类都自己堆砌，所有的运行容器都自己建立——这不是一个健康的项目，这个世界是分工合作的世界，有分享也有贡献，有索取也有回报，这才是 Java 人的理想世界，而且我们也正朝着这个方向前进。

不，我不想回到那个没有 Struts、Spring、Hibernate、Tomcat 的年代，绝对不想。

建议 139：大胆采用开源工具

我们经常会看到一个项目的 lib 中包含了大量的工具、框架包，要想看懂项目代码还应该对这些工具包有一个大致的了解，开源工具包确实会对我们的项目有非常大的帮助，比如提升代码质量，减少 Bug 产生，降低工作量等，但一旦项目中的工具杂乱无章时就会产生依赖的无序性，这会导致代码中隐藏着炸弹，不知何时就会突然引爆了。

而且，在 Java 世界中从来就不缺乏重复发明轮子的例子，MVC 框架有 Struts，也有 Spring MVC、WebWorker；IoC 容器有 Spring，也有 Google Guice；ORM 既有 Hibernate，也有 MyBatis；日志记录有经典的 log4j，也有崭新的 logback。可是选择多了，也会导致我们无从选择。因此，在选择开源工具和框架时要遵循一定的原则：

□ 普适性原则

选择一个工具或框架就必须考虑项目成员的整体技术水平，不能有太大的跨度或跳跃性，要确保大部分项目成员对工具都比较熟悉，若一个项目中的成员大部分是新员工，那么在持久层框架的选择上，使用 MyBatis 就比 Hibernate 要合适，因为 MyBatis 相对简单、方便；再比如在一个熟悉 SSH 开发的团队中，就不应该无故选择 Guice 作为 IoC 容器，除非是行政命令或为了尝鲜。

□ 唯一性原则

相同的工具只选择一个或一种，不要让多种相同或相似职能的工具共存。例如集合工具可以使用 Apache Commons 的 collections 包，当然也可以使用 Google Guava 的 Collections 工具包，但是在项目开发前就应该确认下来，不能让两者共存。

□ “大树纳凉”原则

在没有空调、电风扇的年代，最好的纳凉方式就是找一棵大树，躲在树荫下享受着习习凉风，惬意自在。我们在选择工具包时也应如此，得寻找比较有名的开源组织，比如 Apache、Spring、opensymphony（虽然已经关闭，但它曾经是那么耀眼、辉煌）、Google 等，这些开源组织一则具有固定的开发和运作风格，二则具有广阔的使用人群（很多情况下，我们不会是第一个发现 Bug 的人），在这样的大树下，我们才有时间和精力纳凉，而不会把大好的时间消耗在排查 Bug 上。

□ 精而专原则

在武术上，对一个顶级高手的描述是“精通十八般武器”，但对工具包来说这就不适合了，我们选择的工具包应该是精而专的，而不是广而多的，比如虽然 Spring 框架提供了 Utils 工具包，但在一般情况下不要使用它，因为它不专，Utils 工具包只是 Spring 框架中的一个附加功能而已，要用就用 Apache Commons 的 BeanUtils、Lang 等工具包。

□ 高热度原则

一个开源项目的热度越高，更新得就越频繁，使用的人群就越广，Bug 的曝光率就越快，修复效率也就越高，这对我们项目的稳定性来说是非常重要的。有很多开源项目可能已经很长时间没有更新了，或者是已经非常成熟了，或者是濒于关闭了，这我们不能要求太高，毕竟开源项目已经共享出了他人的精力和智力，我们在享受他人提供的成果的同时，也应该珍惜他人的劳动，最低的标准是不要诋毁开源项目。

对于开源工具，我们应该大胆采用，仔细筛选，如果确实所有的开源工具都无法满足我们的需求，那就自己开发一个开源项目，为千千万万的 Java 人服务，也为 Java 的生态系统贡献自己的力量。

建议 140：推荐使用 Guava 扩展工具包

说起 Guava（石榴），可能知道它的读者并不多，要是说起 Google-collections，相信大部分读者都有所耳闻。2008 年 Google 发布了 Google-collections 扩展工具包，主要是对 JDK 的 Collection 包进行了扩展，2010 年 Google 发布了 Guava 项目，其中包含了 collections、caching、primitives support、concurrency libraries、common annotations、I/O 等，这些都是项目编码中的基本工具包，我们大致浏览一下它的主要功能。

(1) Collections

com.google.common.collect 包中主要包括四部分：不可变集合（Immutable Collections）、多值 Map、Table 表和集合工具类。

□ 不可变集合

不可变集合包括 ImmutableList、ImmutableMap、ImmutableSet、ImmutableSortedMap、ImmutableSortedSet 等，它比不可修改集合（Unmodifiable Collections）更容易使用，效率更高，而且占用的内存更少。示例代码如下：

```
// 不可变列表
ImmutableList<String> list = ImmutableList.of("A", "B", "C");
// 不可变 Map
ImmutableMap<Integer, String> map = ImmutableMap.of(1, "壹", 2, "贰", 3, "叁");
```

其中的 of 方法有多个重载，其目的就是为了便于在初始化的时候直接生成一个不可变

集合。

□ 多值 Map

多值 Map 比较简单，在 JDK 中，Map 中的一个键对应一个值，在 put 一个键值对时，如果键重复了，则会覆盖原有的值，在大多数情况下这比较符合实际应用，但有的时候确实会存在一个键对应多个值的情况，比如我们的通讯录，一个人可能会对应两个或三个号码，此时使用 JDK 的 Map 就有点麻烦了。在这种情况下，使用 Guava 的 Multimap 可以很好地解决问题，代码如下：

```
// 多值 Map
Multimap<String, String> phoneBook = ArrayListMultimap.create();
phoneBook.put("张三", "110");
phoneBook.put("张三", "119");
System.out.println(phoneBook.get("张三"));
```

输出的结果是一个包含两个元素的 Collection，这是一种很巧妙的处理方式，可以方便地解决我们开发中的问题。

□ Table 表

在 GIS（Geographic Information System，地理信息系统）中，我们经常会把一个地点标注在一个坐标上，比如把上海人民广场标注在北纬 31.23、东经 121.48 的位置上，也就是说只要给出了准确的经度和纬度就可以进行精确的定位——两个键决定一个值，这在 Guava 中是使用 Table 来表示的，示例代码如下：

```
Table<Double, Double, String> g = HashBasedTable.create();
// 定义人民广场的经纬度坐标
g.put(31.23, 121.48, "人民广场");
// 输出坐标点的建筑物
g.get(31.23, 121.48);
```

其实 Guava 的 Table 类与我们经常接触的 DBRMS 表非常类似，可以认为它是一个没有 Schema 限定的数据表，比如：

```
//Table，完全类似于数据库表
Table<Integer, Integer, String> user = HashBasedTable.create();
// 第一行、第一列的值是张三
user.put(1, 1, "张三");
// 第一行、第二列的值是李四
user.put(1, 2, "李四");
// 第一行第一列是谁
user.get(1, 1);
```

□ 集合工具类

Guava 的集合工具类分得比较细，比如 Lists、Maps、Sets 分别对应的是 List、Map、Set 的工具类，它们的使用方法比较简单，比如 Map 的过滤，如下所示。

```
// 姓名、年龄键值对
Map<String, Integer> user = new HashMap<String, Integer>();
user.put("张三", 20);
user.put("李四", 22);
user.put("王五", 25);
// 所有年龄大于 20 岁的人员
Map<String, Integer> filteredMap = Maps.filterValues(user,
    new Predicate<Integer>() {
        public boolean apply(Integer _age) {
            return _age > 20;
        }
    });
}
```

(2) 字符串操作

Guava 提供了两个非常好用的字符串操作工具：Joiner 连接器和 Splitter 拆分器。当然，字符串的连接和拆分使用 JDK 的方法也可以实现，但是使用 Guava 更简单一些，比如字符串的连接，代码如下所示：

```
// 定义连接符号
Joiner joiner = Joiner.on(", ");
// 可以连接多个对象，不局限于 String；如果有 null，则跳过
String str = joiner.skipNulls().join("嘿", "Guava 很不错的。");
Map<String, String> map = new HashMap<String, String>();
map.put("张三", "普通员工");
map.put("李四", "领导");
// 键值之间以"是"连接，多个键值以空格分隔
System.out.println(joiner.on("\r\n").withKeyValueSeparator("是").join(map));
```

Joiner 不仅能够连接字符串，还能够把 Map 中的键值对串联起来，比直接输出 Map 优雅了许多。Splitter 是做字符拆分的，使用方法也比较简单，示例代码如下：

```
String str = "你好, Guava";
// 以", "中文逗号分隔
for (String s : Splitter.on(", ").split(str)) {
    System.out.println(s);
}
// 按照固定长度分隔
for (String s : Splitter.fixedLength(2).split(str)) {
    System.out.println(s);
}
```

注意 fixedLength 方法，它是按照给定长度进行拆分的，比如在进行格式化打印的时候，一行最大可以打印 120 个字符，此时使用该方法就非常简单了。

(3) 基本类型工具

基本类型工具在 primitives 包中，是以基本类型名 +s 的方式命名的，比如 Ints 是 int 的工具类，Doubles 是 double 的工具类，注意这些都是针对基本类型的，而不是针对包装类型的。如下代码所示。

```

int[] ints = { 10, 9, 20, 40, 80 };
// 从数组中取出最大值
System.out.println(Ints.max(ints));
List<Integer> integers = new ArrayList<Integer>();
// 把包装类型的集合转为基本类型数组
ints = Ints.toArray(integers);

```

Guava 还提供了其他操作（如 I/O 操作），相对来说功能不是非常强大，不再赘述，读者有兴趣可以自行下载源码研究一番。

建议 141：Apache 扩展包

Apache Commons 通用扩展包基本上是每个项目都会使用的，只是使用的多少不同而已，一般情况下 lang 包用作 JDK 的基础语言扩展，Collections 用作集合扩展，DBCP 用作数据库连接池等，考虑到 commons 的名气很响，下面将对它进行相应的介绍，以备在实际开发中使用。

(1) Lang

Apache 的 Lang 功能实在是太实用了，它的很多工具类都是我们在开发过程中经常会用到的，虽然采用 JDK 的原始类也可以实现，但会花费更多的精力，而且 Lang 的更新频度很高，用它时不用担心会有太多的 Bug。

□ 字符串操作工具类

JDK 提供了 String 类，也提供了一些基本的操作方法，但是要知道 String 类在项目中是应用最多的类，这也预示着 JDK 提供的 String 工具不足以满足开发需求，Lang 包弥补了这个缺陷，它提供了诸如 StringUtils（基本的 String 操作类）、StringEscapeUtils（String 的转义工具）、RandomStringUtils（随机字符串工具）等非常实用的工具，简单示例如下：

```

// 判断一个字符串是否为空, null 或 "" 都返回 true
StringUtils.isEmpty(str);
// 是否是数字
StringUtils.isNumeric(str);
// 最左边两个字符
StringUtils.left(str, 2);
// 统计子字符串出现的次数
StringUtils.countMatches(str, subString);
// 转义 XML 标示
StringEscapeUtils.escapeXml(str);
// 随机生成, 长度为 10 的仅字母的字符串
RandomStringUtils.randomAlphabetic(10);
// 随机生成, 长度为 10 的 ASCII 字符串
RandomStringUtils.randomAscii(10);
// 以一个单词为操作对象, 首字母大写, 输出结果为 :Abc Bcd
WordUtils.capitalize("abc bcd");

```

□ Object 工具类

每个类都有 equals、hashCode、toString 方法，如果我们自己编写的类需要覆写这些方法，就需要考虑很多的因素了，特别是 equals 方法，可以参考第 3 章有关 equals 的建议，如果我们使用 lang 包就会简单得多，示例代码如下：

```
class Person {
    private String name;
    private int age;
    /*getter/setter省略*/
    // 自定义输出格式
    public String toString() {
        return new ToStringBuilder(this)
            .append("姓名", name)
            .append("年龄", age)
            .toString();
    }
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (obj == this) {
            return true;
        }
        if (obj.getClass() != getClass()) {
            return false;
        }
        Person p = (Person) obj;
        // 只要姓名相同，就认为两个对象相等
        return new EqualsBuilder()
            .appendSuper(super.equals(obj))
            .append(name, p.name)
            .isEquals();
    }
    // 自定义 hashCode
    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this);
    }
}
```

□ 可变的基本类型

基本类型都有相应的包装类型，但是包装类型不能参与加、减、乘、除运算，要运算还得转化为基本类型，那如果希望使用包装类进行运算该怎么办呢？使用 Lang 包的示例如下：

```
// 声明一个可变的 int 类型
MutableInt mi = new MutableInt(10);
//mi 加 10，结果为 20
mi.add(10);
// 自加 1，结果为 21
mi.increment();
```

□ 其他 Utils 工具

Lang 包在日期处理方面主要提供了 DateUtils 和 DateFormatUtils 两个工具类，相比较而言它们没有 Joda 强大，而且方法也较简单，不再赘述。

Lang 包还提供了诸如 ArrayUtils、LocaleUtils、NumberUtils 等多个工具类，当项目中需要时可以查询一下 API，一般情况下都有相应的解决办法。

(2) BeanUtils

它是 JavaBean 的操作工具包，不仅可以实现属性的拷贝、转换等，还可以建立动态的 Bean，甚至建立一些自由度非常高的 Bean，我们简单地了解一下它的使用方法。

□ 属性拷贝

在分层开发时经常会遇到 PO (Persistence Object) 和 VO (Value Object) 之间的转换问题，不过，有多种方法可以解决之，比如自己写代码 PO.setXXX(VO.getXXX())，但是在属性较多的时候容易出错，最好的办法就是使用 BeanUtils 来操作，代码如下：

```
//PO 对象
User user = new User();
//VO 对象
Person person = new Person();
// 两个 Bean 属性拷贝
PropertyUtils.copyProperties(person, user);
// 把 Map 中的键值对拷贝到 Bean 上
Map<String, String> map = new HashMap<String, String>();
map.put("name", "张三");
PropertyUtils.copyProperties(person, map);
```

□ 动态 Bean 和自由 Bean

我们知道定义一个 Bean 必然会需要一个类，比如 User、Person 等，而且还必须在编译期定义完毕，生成 .class 文件，虽然 Bean 是一个有固定格式的数据载体，严格要求确实没错，但在某些时候这限制了 Bean 的灵活性，比如要在运行期生成一个动态 Bean，或者在需要生成无固定格式的 Bean 时，使用普通 Bean 就无法实现了。我们可以使用 BeanUtils 包解决该问题，示例代码如下：

```
// 动态 Bean，首先定义 Bean 类
DynaProperty[] props = new DynaProperty[] {
    new DynaProperty("name", String.class),
    new DynaProperty("age", int.class) };
BasicDynaClass dynaClass = new BasicDynaClass("people", null, props);
// 动态 Bean 对象
DynaBean people = dynaClass.newInstance();
/*people 的 get/set 操作 */
// 自由 Bean
DynaBean user = new LazyDynaBean();
// 直接定义属性和值
```

```

user.set("name", "张三");
// 定义属性名，限定属性类型为 Map
user.set("phoneNum", "tel", "021");
user.set("phoneNum", "mobile", "138");

// 属性类型为 ArrayList
user.set("address", 0, "上海");
user.set("address", 1, "北京");

```

□ 转换器

如果我们期望把一个 Bean 的所有 String 类型属性在输出之前都加上一个前缀，该如何做呢？一个一个进行属性过滤？或者使用反射来检查属性类型是否是 String，然后加上前缀？这样是可以解决，但不优雅，看 BeanUtils 如何解决：

```

// 一个简单的 Bean 对象
User user = new User("张三", 18);
// 转换工具
ConvertUtilsBean cub = new ConvertUtilsBean();
// 注册一个转换器
cub.register(new Converter() {
    public Object convert(Class type, Object value) {
        // 为每个 String 类型的属性加上前缀
        return "prefix-" + value;
    }
}, String.class);
// 建立一个依赖特定转换工具的 Bean 工具类
BeanUtilsBean beanUtils = new BeanUtilsBean(cub);
// 输出结果为：prefix- 张三
beanUtils.getProperty(user, "name");

```

(3) Collections

Collections 工具包提供了 ListUtils、MapUtils 等基本集合操作工具，比较常用而且较简单，这里就不再介绍了。需要重点说明的是 Collections 包中 3 个不太常用的集合对象，如下所示。

□ Bag

Bag 是 Collections 中的一种，它可以容纳重复元素，与 List 的最大不同点是它提供了重复元素的统计功能，比如一个盒子中有 100 个球，现在要计算出蓝色球的数量，使用 Bag 就很容易实现，代码如下：

```

// 一个盒子中装了 4 个球
Bag box = new HashBag(Arrays.asList("red", "blue", "black", "blue"));
// 又增加了 3 个蓝色球
box.add("blue", 3);
// 球的数量为 7
box.size();
// 蓝色球数量为 5
box.getCount("blue");

```

□ lazy 系列

有这样一句话“在我需要的时候，你再出现”，lazy 系列的集合就是起这样的作用的，在集合中的元素被访问时它才会生成，这也就涉及一个元素的生成问题了，可通过 Factory 的实现类来完成，示例代码如下：

```
// 把一个 List 包装成一个 lazy 类型
List<String> lazy = LazyList.decorate(new ArrayList(),
    new Factory() {
        public String create() {
            return "A";
        }
    });
// 访问了第 4 个元素，此时 0、1、2 元素为 null
String obj = lazy.get(3);
// 追加一个元素
lazy.add(" 第五个元素 ");
// 元素总数为 5 个
lazy.size();
```

□ 双向 Map

JDK 中的 Map 要求键必须唯一，而双向 Map (Bidirectional Map) 则要求键、值都必须唯一，也就是键值是一一对应的，此类 Map 的好处就是既可以根据键进行操作，也可以反向根据值进行操作，比如删除、查询等，示例代码如下：

```
// key、value 都不允许重复的 Map
BidiMap bidiMap = new TreeBidiMap();
bidiMap.put(1, "壹");
// 根据 key 获取 value
bidiMap.get(1);
// 根据 value 获取 key
bidiMap.getKey("壹");
// 根据 value 删除键值对
bidiMap.removeValue("壹");
```

Apache commons 项目还有很多非常好用的工具，如 DBCP、net、Math 等，但是这些包有个缺点，大部分更新比较缓慢，有些扩展类甚至可以说比较陈旧了，例如 Collections 中的大部分集合类不支持泛型，这让一些“泛型控”们很不舒服，总想自己再封装一下，提供一些泛型支持，这就需要读者在项目开发中自行考虑了。

建议 142：推荐使用 Joda 日期时间扩展包

开发一个项目必然要和日期时间打交道，特别是一些全球性的项目，必须要考虑语言和时区问题，但是在 JDK 中，日期时间的操作比较麻烦，例如 1000 小时后是星期几，伦敦时

间是几点等，这里介绍一下通过 Joda 开源包来操作时间的方法，非常简单方便。

(1) 本地格式的日期时间

依据操作系统或指定的区域输出日期或时间，例如：

```
// 当前时间截
DateTime dt = new DateTime();
// 输出英文星期
dt.dayOfWeek().getAsString(Locale.ENGLISH);
// 本地日期格式
dt.toLocalDate();
// 日期格式化
dt.toString(DateTimeFormat.forPattern("yyyy 年 M 月 d 日"));
```

(2) 日期计算

这是 Joda 最方便的地方，也是 JDK 最麻烦的地方，比如我们要计算 100 天后是星期几，直接使用 JDK 提供的日期类会非常麻烦，使用 Joda 就简单很多，例如：

```
// 当前时间截
DateTime dt = new DateTime();
// 加 100 小时是星期几
dt.plusHours(100).dayOfWeek();
// 100 天后的日期
dt.plusDays(100).toLocalDate();
// 10 年前的今天是星期几
dt.minusYears(10).dayOfWeek().getAsString();
// 离地球毁灭还有多少小时
Hours.hoursBetween(dt,new DateTime("2012-12-21")).getHours();
```

这里需要注意的是，`DateTime` 是一个不可变类型，与 `String` 非常类似，即使通过 `plusXXX`、`minusXX` 等方法进行操作，`dt` 对象仍然不会变，只是新生成一个 `DateTime` 对象而已。但是，Joda 也提供了一个可变类型的日期对象：`MutableDateTime` 类，这样，日期的加减操作就更加方便了，比如列出 10 年内的黑色星期五，实现代码如下（此实现若用 JDK 的类来计算会异常复杂，读者可以尝试一下）：

```
// 当前可变时间
MutableDateTime mdt = new MutableDateTime();
// 10 年后的日期
DateTime destDateTime= dt.plusYears(10);
while (mdt.isBefore(destDateTime)) {
    // 循环一次加 1 天
    mdt.addDays(1);
    // 是 13 号，并且是星期五
    if (mdt.getDayOfMonth() == 13 && mdt.getDayOfWeek() == 5) {
        // 打印出 10 年内所有的黑色星期五
        System.out.println(" 黑色星期五： " + mdt);
    }
}
```

(3) 时区时间

这个比较简单实用，给定一个时区或地区代码即可计算出该时区的时间，比如在一个全球系统中，数据库中全部是按照标准时间来记录的，但是在展示层要按照不同的用户、不同的时区展现，这就涉及时区计算了，代码如下：

```
// 当前时间截
DateTime dt = new DateTime();
// 此时伦敦市的时间
dt.withZone(DateTimeZone.forID("Europe/London"));
// 计算出标准时间
dt.withZone(DateTimeZone.UTC);
```

Joda 还有一个优点，它可以与 JDK 的日期库方便地进行转换，可以从 `java.util.Date` 类型转为 Joda 的 `DateTime` 类型，也可以从 Joda 的 `DateTime` 转为 `java.util.Date`，代码如下：

```
DateTime dt = new DateTime();
// Joda 的 DateTime 转为 JDK 的 Date
Date jdkDate = dt.toDate();
// JDK 的 Date 转为 Joda 的 DateTime
dt = new DateTime(jdkDate);
```

经过这样的转换，Joda 可以很好地与现有的日期类保持兼容，在需要复杂的日期计算时使用 Joda，在需要与其他系统通信或写到持久层中时则使用 JDK 的 `Date`。Joda 是一种令人惊奇的高效工具，无论是计算日期、打印日期，或是解析日期，Joda 都是首选，当然日期工具类也可以选择 `date4j`，它也是一个不错的开源工具，这里就不再赘述了。

建议 143：可以选择多种 Collections 扩展

为什么这么多的开源框架热衷于 `Collections` 的扩展呢？是因为我们程序（经典的定义：`程序 = 算法 + 数据结构`，想想看数据结构是为谁而服务的）主要处理的是一大批数据，而能容纳大量数据的也就是 `Collections` 类和数组了，但是数据的格式具有多样性，比如数据映射关系多样，数据类型多样等，下面我们再介绍三个比较有个性的 `Collections` 扩展工具包。

(1) fastutil

`fastutil`（按照 Java 的拼写规则应该为 `FastUtil`，但是官网就是这样命名的，我们尊重官方）是一个更新比较频繁的工具包，它的最新版本是 6.3，主要提供了两种功能：一种是限定键值类型（Type Specific）的 `Map`、`List`、`Set` 等，另一种是大容量的集合。我们先来看示例代码：

```
// 明确键类型的 Map
Int2ObjectMap<String> map = new Int2ObjectOpenHashMap<String>();
map.put(100, "A");
```

```
// 超大容量的 List，注意调整 JVM 的 Heap 内存
BigList<String> bigList = new ObjectBigArrayList< String>(
    1L + Integer.MAX_VALUE);
// 基本类型的集合，不再使用 Integer 包装类型
IntArrayList arrayList = new IntArrayList();
```

这里要特别注意的是大容量集合，什么叫大容量集合呢？我们知道一个 Collection 的最大容量是 Integer 的最大值 (2 147 483 647)，不能超过这个容量，一旦我们需要把一组超大的数据放到集合中，就必须要考虑对此进行拆分了，这会导致程序的复杂性提高，而 fastutil 则提供了 Big 系列的集合，它的最大容量是 Long 的最大值，这已经是一个非常庞大的数字了，超过这个容量基本上是不可能的。但在使用它的时候需要考虑内存溢出的问题，注意调节 Java 的 mx 参数配置。

(2) Trove

Trove 提供了一个快速、高效、低内存消耗的 Collection 集合，并且还提供了过滤和拦截的功能，同时还提供了基本类型的集合，示例代码如下：

```
// 基本类型的集合，不使用包装类型
TIntList intList = new TIntArrayList();
// 每个元素值乘以 2
intList.transformValues(new TIntFunction() {
    public int execute(int element) {
        return element * 2;
    }
});
// 过滤，把大于 200 的元素组成一个新的列表
TIntList t2 = intList.grep(new TIntProcedure() {
    public boolean execute(int _element) {
        return _element > 200;
    }
});
// 包装为 JDK 的 List
List<Integer> list = new TIntListDecorator(intList);
// 键类型确定 Map
TIntObjectMap<String> map = new TIntObjectHashMap<String>();
```

Trove 的最大优势是在高性能上，在进行一般的增加、修改、删除操作时，Trove 的响应时间比 JDK 的集合少一个数量级，比 fastutil 也会高很多，因此在高性能项目中要考虑使用 Trove。

(3) lambdaj

lambdaj 是一个纯净的集合操作工具，它不会提供任何的集合扩展，只会提供对集合的操作，比如查询、过滤、统一初始化等，特别是它的查询操作，非常类似于 DBRMS 上的 SQL 语句，而且也会提供诸如求和、求平均值等的方法，示例代码如下：

```
List<Integer> ints = new ArrayList<Integer>();
// 计算平均值
```

```
Lambda.avg(ints);
// 统计每个元素出现的次数，返回的是一个 Map
Lambda.count(ints);
// 按照年龄排序
List<Person> persons = new ArrayList<Person>();
Lambda.sort(persons, Lambda.on(Person.class).getAge());
// 串联所有元素的指定属性，输出为：张三，李四，王五
Lambda.joinFrom(persons).getName();
// 过滤出年龄大于 20 岁的所用元素，输出为一个子列表
Lambda.select(persons, new BaseMatcher<Person>() {
    @Override
    public boolean matches(Object _person) {
        Person p = (Person) _person;
        return p.getAge() > 20;
    }
    public void describeTo(Description desc) {
    }
});
// 查找出最大年龄
Lambda.maxFrom(persons).getAge();
// 抽取出所有姓名形成一个数组
Lambda.extract(persons, Lambda.on(Person.class).getName());
```

lambdaj 算是一个比较年轻的开源工具，但是它符合开发人员的习惯，对集合的操作提供了“One Line”式的解决方法，可以大大缩减代码的数量，而且也不会导致代码的可读性降低，读者可以在下一个项目中使用此类开源工具。



Java happens to be a really good language for a broad spectrum of topics.

Java 只是碰巧成为了一门用途广泛的优秀语言。

——James Gosling (詹姆斯·高斯林, Java 的创始人)

编码不仅仅是把代码写出来，还要求清晰地表达出编码者头脑中的逻辑，准确地传递到计算机中执行，同时也能够被其他编码者轻松阅读，而要实现这些目标，则要求代码有清晰、正确的思想，即编程思想。

编程思想是软件诞生的源泉，当它喷涌而发时，也是优秀软件诞生之时。

建议 144：提倡良好的代码风格

代码的版面和样式比较多，每个项目组基本上都有自己的编码规范，大家都希望形成良好的代码风格，以便提高代码的可读性，方便生成维护文档，减少缺陷出现的几率等，在 Java 的开发中一般都是按照《The Java Language Specification》（即《Java 编码规范》）来制定编码规范的，但是基本上每个项目组都会有一些自己的个性特征，我们不去评说哪一个代码风格优秀，哪一个是较差的，而是来分析一下优秀团队的编码风格应该具有哪些特征。

（1）整洁

不管代码风格的定义有多优秀，有多适合开发人员，如果代码结构混乱不堪，即使效率再高，也会使维护难以持续。我们知道，代码首先是给人看的，然后才是给机器执行的，对于机器来说，只要代码符合规范，不在乎其格式是否整洁、是否有缩进、是否有回车，只要代码正确就能正常运行，而人就不同了，没有缩进没有回车的代码基本上是不可阅读的。试想一下一个没有标点符号的文档，整篇就是一个段落，能读懂的基本上是天才（或许你会说古代的诗词就是没有标点符号的。确实，但它有韵律）。

（2）统一

从一个团队中诞生的代码应该具有一致的风格，要使用下挂式括号就全部使用下挂式括号，要使用 tab 缩进就全部使用 tab 缩进，要使用小驼峰方式命名就全部使用小驼峰方式命名，不要带有个人色彩的风格标识。这样可以让我们的代码看起来很职业，而不是一帮乌合之众产生的“稻草”式代码。

统一的代码风格还要求具有连贯性，我们应该在不同的模块、层级中使用相同的编码风格，而不能在展现层使用一种编码，在逻辑层又使用另外一种编码风格。一个项目的编码风格不应该因为所处的功能区不同而有所差异。当然，若使用多种异构语言开发项目，则可以考虑为不同的语言提供不同的规范。

（3）流行

一种潮流风行世界的时候必然有其诞生的原因（感冒也包括在内），一种编码格式的流行也必然有它存在的理由，我们完全可以借鉴流行的编码格式，没有必要对这种风格进行重塑，而且使用流行风格可以让新成员尽快融入项目，避免出现进入一个新环境而出现茫茫无助的状态。

不要让您的代码规范标新立异，独树一帜，跟随“风尚”也许是一种省事、省力、省心

最好的编码风格。

(4) 便捷

制定出来的编码规范必须有通用开发工具支撑，不能制定出只能由个别开发工具支持的规范，甚至是绑定在某一个IDE上。在小范围内独乐乐，可以提升代码的友好度，方便使用，但很难大范围内推而广之，特别是很难上升到工程级别。代码风格是为一个团队准备的，如果团队中就只有一个开发人员，基本上代码风格不会有太大差异，这是习惯和个性使然，但是如果团队中有多个成员，就需要防止给开发人员过度的自由了，不符合开发规范的代码要坚决予以重构，以使团队代码风格一致。

现在的项目中源代码逐渐增多，完全依靠人工来做代码走查很难查出问题，我们可以使用工具来统计代码，这里推荐使用 Checkstyle，它可以自定义代码模板，然后根据模板检查代码是否遵循规范，从而减少枯燥的代码走查。

建议145：不要完全依靠单元测试来发现问题

单元测试的目的是保证各个独立分割的程序单元的正确性，虽然它能够发现程序中存在的问题（或缺陷，或错误），但是单元测试只是排查程序错误的一种方式，不能保证代码中的所有错误都能被单元测试挖掘出来，原因有以下四点。

(1) 单元测试不可能测试所有的场景（路径）

单元测试必须测试的三种数据场景是：正常场景、边界场景、异常场景。一般情况下，如果这三种测试场景都能出现预期的结果，则认为代码正确，但问题是代码是人类思维的直观表达，要想完整地测试它就必须写出比生产代码多得多的测试代码，例如有这样一个类：

```
public class Foo {
    // 除法计算
    public int divid(int a,int b){
        return a / b;
    }
}
```

就这一个简单的除法计算，如果我们要进行完整的测试就必须建立三个不同的测试场景：正常数据场景，用来测试代码的主逻辑；边界数据场景，用来测试代码（或数据）在边界的情况下逻辑是否正确；异常数据场景，用来测试出现异常非故障时能否按照预期运行，测试类如下：

```
public class FooTest {
    // 构建测试对象
    private Foo foo = new Foo();

    // 正常测试场景
```

```

    @Test
    public void testDividNormal() {
        // 断言 100 除以 10 的结果为 10
        assertEquals(10, foo.divid(100, 10));
    }

    // 边界测试场景
    @Test
    public void testDividBroader() {
        // 断言最大值除以最小值结果为 0
        assertEquals(0, foo.divid(Integer.MAX_VALUE, Integer.MIN_VALUE));
        // 断言最小值除以最大值结果为 -1
        assertEquals(-1, foo.divid(Integer.MIN_VALUE, Integer.MAX_VALUE));
    }

    // 异常测试场景
    @Test(expected = ArithmeticException.class)
    public void testDividException() {
        // 断言除数为 0 时抛出 ArithmeticException
        foo.divid(100, 0);
        // 断言不会执行到这里
        fail();
    }
}
}

```

诸位可以看到这么简单的一个除法计算就需要如此多的测试代码，如果在生产代码中再加入就 if、switch 等判断语句，那它所需要的测试场景就会更加复杂了。只要有一个判断条件，就必须有两个测试场景（条件为真的场景和条件为假的场景），这也是在项目中的测试覆盖率不能达到 100% 的一个主要原因：单元测试的代码量远大于生产代码。通常在项目中，单元测试覆盖率很难达到 60%，因为不能 100% 覆盖，这就导致了代码测试的不完整性，隐藏的缺陷也就必然存在了。

(2) 代码整合错误是不可避免的

单元测试只是保证了分割的独立单元的正确性，它不能保证一个功能的完整性。单元测试首先会假设所有的依赖条件都满足，但真实情况并不是这样的，我们经常会发现虽然所有的单元测试都通过了，但在进行整合测试时仍然会产生大量的业务错误——很多情况下，此种错误是因为对代码的职责不清晰而引起的，这属于认知范畴，不能通过单元测试避免。

(3) 部分代码无法（或很难）测试

如果把如下代码放置在一个多线程的环境下，思考一下该如何测试呢？代码如下：

```

class Apple{
    // 苹果颜色
    private int color;
    public int getColor() {
        return color;
    }
}

```

```

public void setColor(int color) {
    this.color = color;
}
}

```

这是一个简单的 JavaBean，也是我们项目中经常出现的，对于此类 BO (Business Object)，通常情况下是不会进行单元测试的，想必你也会想这不用测试吧，很简单嘛，就一个 getter/setter 方法，出错的可能性不大。但这只是我们一厢情愿的想法，如果该 Apple 是在多线程环境下，你还认为不会出现线程不安全的情况吗？事实上，因为没有采用资源保护措施 (synchronized 或 Lock)，多个线程共同访问该对象时就会出现不安全的情况。现在问题来了：为什么在通常情况下不做此类对象的单元测试呢？

比如一个 JEE 应用，一般情况下都是多线程环境，但是我们很少对代码进行多线程测试，原因很简单，测试很复杂，很难进行全面的多线程测试。而且如果要保证在多线程下测试通过，就必须对代码增加大量的修饰，这必然会导致代码的可读性和可维护性降低，这也是我们一般都抛弃多线程测试的原因。

在 Spring 中，默认情况下每个注入的对象都是 Singleton 的，也就是单例的，每个类在内存中只有一个对象实例，这也是偶尔出现数据资源不一致现象的元凶：在多线程环境下数据未进行资源保护，特别是在系统压力较大、响应能力较低的情况下，数据资源出现不一致情况的可能性更大。

这只是一个单元测试很难覆盖的情景，还有一种情景是根本不能实施单元测试，比如不确定性算法 (Nondeterministic Algorithm)，什么叫不确定算法？像我们经常接触的函数 $f(x)$ ，给定一个确定的 x 值，就有确定的结果 $f(x)$ ，在任何时候输入 x ，都能获得固定的 $f(x)$ ，这就是确定性算法，也是我们经常接触的，但还有一种算法，比如要计算出明天通过某一个大桥的车辆数量，必须根据专家经验、天气、交通情况、是否是节庆日、是否有大型体育比赛、并行道路通行的情况等来进行计算，这些条件很多都是非确定的依据，所推导出的也是一个非确定结论的数据——明天通过大桥的车辆数量，想想看，这怎么进行单元测试，不确定算法只能无限接近而不能达到，单元测试只能对确定算法进行假设，不能对不确定算法进行验证。

(4) 单元测试验证的是编码人员的假设

我们都知道单元测试是白箱测试，一般情况下测试代码是编码人员自行编写的，我们可以这样理解，编码人员根据胸中的蓝图，迅速地实行了一个算法，然后通过断言确定算法是否与预期相匹配。简单地说，我们左手画了一个圆，右手拿着一个圆规进行测试，检验这是否是一个标准的圆，但问题是是谁要求我们画一个圆的呢？谁又能确定是一个直径为 2 厘米的圆而不是 2.1 厘米的圆？——代码的意图只是反映了编码者的理解，而单元测试只是验证了这种假设。想想看，如果编码者从一开始就误解了需求意图，此时的单元测试就充当了帮凶：验证了一个错误的假设。

指出单元测试的缺陷，并不是为了说明单元测试不必要，相反，单元测试在项目中是必须的，而且是非常重要的，特别敏捷开发中提倡的 TDD（Test-Driven Development）测试驱动开发：单元测试先行，而后才会编写生产代码，这可以大幅度地提升代码质量，加快项目开发的进度。

建议 146：让注释正确、清晰、简洁

从我们写第一个“Hello World”程序开始，就被谆谆教导“代码要有注释”，而且一加就是好多年，基本上是代码就有注释，而且有的还很多，甚至有的是长篇累牍的。我们先来看一些不好的注解习惯。

（1）废话式注释

比如这样的注释：

```
/*
 * 该算法不如某某算法优秀，可以优化，时间太紧，以后再说
 */
public void doSomethong() {
}
```

想说明什么？只是表明这个算法需要优化？还是说这是未完成的任务？那可以用 TODO 标记呀，注释是给人看的，此类代码提交上去后，基本上不会再修改它了，除非它出现 Bug，或者维护人员碰巧看到这个注释，然后选择了优化它——注释不是留给“撞大运”人员的。

（2）故事式注释

曾经检查过一段极品代码，注释写得非常全面，描述的是汉诺塔算法，从汉诺塔的故事（包括最原始的版本和多个变形版本）到算法分析，最后到算法比较和实际应用，写得那是栩栩如生，而且还不时加入了一些崭新的网络用语，幽默而又不失准确，可以说，看完这段注释基本上对汉诺塔的“前世今生”有了深刻的了解，但是我在检查后的改正意见是：把注释修改为“实现汉诺塔算法”即可。注释不是让你讲故事的地方，就这 7 个字，已经完全可以说明你的代码了！

我们的代码是给人看的，但不是给什么都不懂的外行看的，相信我们代码的阅读者一定是具有一定编码能力的，不是对代码过敏的“代码白痴”。

（3）不必要的注释

有些注释相对于代码来说完全没有必要，算不上是废话，只能说是多余的注解，看下面的例子。

```

class Foo{
    // 默认值为 0
    private int num;
    // 取值
    public int getNum() {
        return num;
    }
    // 输入 int 类型变量，无返回值
    public void setNum(int num) {
        this.num = num;
    }
    public void doSomething(){
        // 自增
        num++;
    }
}

```

以上四个注解是不必注释的典型代表（本书的代码上也有一些类似的注释，只是为了阐明代码片段，不用作生产代码）：

第一个默认值完全没有必要说明，相信代码的阅读者这点智商还是有的，他应该明白实例变量初始值为 0，即使加上个初始值也完全没有必要注释，除非有特殊含义。

第二个注释在 get 方法上，如此简单的代码，看代码比看注释所花费的时间长不了多少，不要低估了代码阅读者（很可能就是你，代码的编写者）的智商。

第三个注释描述了输入和输出参数类型，相信 IDE 吧，相信它会这么“智能化”的提示吧（基本上每个 IDE 都能实现输入补全和输入输出提示）！不需要我们手工撰写。这些注释难道是为那些用记事本编写代码的狂人准备的？可是在看到输入的 int 类型，输出的 void 后，难道他还不能明白吗？——注释完全多余了。

第四个注释也蔑视了代码阅读者的智商，这是编码的最基本算法，不用注释。

(4) 过时的注释

注释与代码的版本不一致，注释是 1.0 版本，而代码早已窜到了 5.0 版本，相信读者对此类注释深有感触：代码一直在升级，但注释永远保持不变，直到有一天，某一个“粗心”的家伙根据注释修正了一个代码缺陷，然后发现产生了连锁的缺陷反应才知道“代码世界”已经早已发生了变化，而此处的注释只是描述了最原始的信息。

这类注释不仅仅会出现在你我的代码中，同时也会出现在一些非常著名的开源系统中，毕竟注释不参与运行，只是给人看的，修改代码而不修改注释照样可以运行，添加注释只是“额外任务”而已。解决此类问题的最好办法就是保持注释与代码同步。

(5) 大块注释代码

可能是为了考虑代码的再次利用，有些大块注释掉的代码仍然保留在生产代码中，这不是一个好的习惯，大块注释代码不仅仅影响代码的阅读性，而且也隔断了代码的连贯性，特别是在代码中的间隔性注释，更增加了阅读的难度，会使 Bug 隐藏得更深。

此类注释代码完全可以使用版本管理来实现，而不是在生产代码中出现。这里要注意的是，如果代码临时不用（可能在下一版本中使用，或者在生产版本固化前可能会被使用），可以通过注释来解决，如果是废弃（在生产版本上肯定不用该代码），则应该完全删除掉。

(6) 流水账式的注释

比如有这样的注释：

```
/*
 * 2010-09-16 张三    创建该类，实现 XX 算法
 * 2010-10-28 李四    修正算法中的 XXXX 缺陷
 * 2010-11-30 李四    重构了 XXX 方法
 * 2011-02-06 王五    删除了 XXXX 无用方法
 * 2011-04-08 马六    优化了 XXX 的性能
 */
class Foo{}
```

相信读者看到过很多这样的注释，而且深以为这样的注释是一种好的方式，如果没有版本管理工具，这确实是一种非常好的注释，可以很清晰地看出该类的变化历程，但是有了版本管理工具，此类注释就不应该出现在这里了，应该出现在版本提交的注释上，版本管理可以更加清晰地浏览此类变更历程。

(7) 专为 JavaDoc 编写的注释

在注释中加入过多的 HTML 标签，可以使生成的 JavaDoc 文档格式整齐美观，这没错，但问题是代码中的注释是给人看的，如果只考虑 JavaDoc 的阅读者，那代码的阅读者（很可能就是一年后的你）就很难看懂代码上的注释了，能做的办法就是生成 JavaDoc 文档，然后文档和代码分开来阅读，这是一种让人迅速崩溃的“绝世良药”。

建议在注释中只保留 `<p>`、`<code>` 等几个常用的标签，不要增加 ``、`<table>`、`<div>` 等标签。

解释了这么多不好的注释，那好的注释应该是什么样子的呢？好的注释首先要求正确，注释与代码意图吻合；其次要求清晰，格式统一，文字准确；最后要求简洁，说明该说明的，惜字如金，拒绝不必要的注释，如下类型的注释就是好的注释：

(1) 法律版权信息

这是我们在阅读源代码时经常看到的，一般都是指向同一个法律版权声明的。

(2) 解释意图的注释

说明为什么要这样做，而不是怎么做的，比如解决了哪个 Bug，方法过时的原因是什么。像这样一个注释就是好的：

```
public class BasicDataSource implements DataSource {
    static {
        // Attempt to prevent deadlocks - see DBCP - 272
```

```

        DriverManager.getDrivers();
    }
}

```

(3) 警示性注释

这类注释是我们经常缺乏的，或者是经常忽视的（即使有了，也常常是与代码版本不匹配），比如可以在注释中提示此代码的缺陷，或者它在不同操作系统上的表现，或者警告后续人员不要随意修改，例如 tomcat 的源码 org.apache.tomcat.jni.Global 中有这样一段注释：

```

public class Global {
    /**
     * Note: it is important that the APR_STATUS_IS_EBUSY(s)
     * macro be used to determine
     * if the return value was APR_EBUSY, for portability reasons.
     * @param mutex the mutex on which to attempt the lock acquiring.
     */
    public static native int trylock(long mutex);
}

```

(4) TODO 注释

对于一些未完成的任务，则增加上 TODO 提示，并标明是什么事情没有做完，以方便下次看到这个 TODO 标记时还能记忆起要做什么事情，比如在 DBCP 源代码中有这样的 TODO 注释：

```

public class DelegatingStatement extends ... implements ... {
    /*
     * Note was protected prior to JDBC 4
     * TODO Consider adding build flags to make this protected
     * unless we are using JDBC 4.
     */
    public boolean isClosed() throws SQLException {
        return _closed;
    }
}

```

注释只是代码阅读的辅助信息，如果代码的表达能力足够清晰，根本就不需要注释，注释能够帮助我们更好地理解代码，但它所重视的是质量而不是数量。如果一段代码写得很糟糕，即使注解写得再漂亮，也不能解决腐烂代码带来的种种问题，记住，注释不是美化剂，不能美化你的代码，它只是一副催化剂，可以让优秀的代码更加优秀，让拙劣的代码更加腐朽。

注意 注释不是美化剂，而是催化剂，或为优秀加分，或为拙劣减分。

建议 147：让接口的职责保持单一

一个类所对应的需求功能越多，引起变化的可能性就越大，单一职责原则（Single Responsibility Principle，简称 SRP）就是要求我们的接口（或类）尽可能保持单一，它的定义是说“一个类有且仅有一个变化的原因（There should never be more than one reason for a class to change）”，那问题是什么是职责呢？

职责是一个接口（或类）要承担的业务含义，或是接口（或类）表现出的意图，例如一个 User 类可以包含写入用户信息到数据库、删除用户、修改用户密码等职责，而一个密码工具类则可以包含解密职责和加密职责。明白了什么是类的职责单一，再来看看它有什么好处。单一职责有以下三个优点：

（1）类的复杂性降低

职责单一，在实现什么职责时都有清晰明确的定义，那么接口（或类）的代码量就会减少，复杂度也就会减少。当然，接口（或类）的数量会增加上去，相互间的关系也会更复杂，这就需要适当把握了。

（2）可读性和可维护性提高

职责单一，会让类中的代码量减少，我们可以一眼看穿该类的实现方式，有助于提供代码的可读性，这也间接提升了代码的可维护性。

（3）降低变更风险

变更是必不可少的，如果接口（或类）的单一职责做得好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，那就会对系统的扩展性、维护性都有非常大的帮助。

既然单一职责有这么多的优点，那我们该如何应用到设计和编码中呢？下面以电话通信为例子来说明如何实施单一职责原则：

（1）分析职责

一次电话通信包含四个过程：拨号、通话、回应、挂机，我们来思考一下该如何划分职责，这四个过程包含了两个职责：一个是拨号和挂机表示的是通信协议的链接和关闭，另外一个是通话和回应所表示的数据交互。

问题是我们依靠什么来划分职责呢？依靠变化因素，我们可以这样考虑该问题：

通信协议的变化会引起数据交换的变化吗？会的！你能在 3G 网络视频聊天，但你很难在 2G 网络上实现。

数据交互的变化会引起通信协议的变化吗？会的！传输 2KB 的文件和 20GB 的文件需要的不可能是同一种网络，用 56KB 的“小猫”传输一个 20GB 的高清影视那是不可行的。

（2）设计接口

职责分析确定了两个职责，首先不要考虑实现类是如何设计的，我们首先应该通过两个接口来实现这两个职责。接口的定义如下。

```

// 通信协议
interface Connection {
    // 拨通电话
    public void dial();
    // 通话完毕，挂电话
    public void hangup();
}

// 数据传输
interface Transfer {
    // 通话
    public void chat();
}

```

(3) 合并实现

接口定义了两个职责，难道实现类就一定要分别实现这两个接口吗？这样做确实完全满足了单一职责原则的要求：每个接口和类职责分明，结构清晰，但是我相信读者在设计的时候肯定不会采用这种方式，因为一个电话类要把 ConnectionManager 和 DataTransfer 的实现类组合起来才能使用。组合是一种强耦合关系，你和我都有共同的生命期，这样的强耦合关系还不如使用接口实现的方式呢！而且这还增加了类的复杂性，多出了两个类。

通常的做法是一个实现类实现多个职责，也就是实现多个接口，代码如下：

```

// 电话
class Phone implements Connection, Transfer{
    // 实现各个接口
}

```

这样的设计才是完美的，一个类实现了两个接口，把两个职责融合在一个类中。你会觉得这个 Phone 有两个原因引起了变化，是的，但是别忘记了我们是面向接口编程的，我们对外公布的是接口而不是实现类。而且，如果真要实现类的单一职责，就必须使用上面的组合模式，这会引起类间的耦合过重、类的数量增加等问题，人为地增加了设计的复杂性。

对于单一职责原则，我建议接口一定要做到职责单一，类的设计尽量做到只有一个原因引起变化。

注意 接口职责一定要单一，实现类职责尽量单一。

建议 148：增强类的可替换性

Java 的三大特征：封装、继承、多态，这是每个初学者都会学习到的知识点，这里暂且不说封装和继承，单单说说多态。一个接口可以有多个实现方式，一个父类可以有多个子类，并且可以把不同的实现或子类赋给不同的接口或父类。多态的好处非常多，其中有一点就是增强了类的可替换性，但是单单一个多态特性，很难保证我们的类是完全可以替换的，

幸好还有一个里氏替换原则来约束。

里氏替换原则是说“所有引用基类的地方必须能透明地使用其子类的对象”，通俗点讲，只要父类型能出现的地方子类型就可以出现，而且将父类型替换为子类型还不会产生任何错误或异常，使用者可能根本就不需要知道是父类型还是子类型。但是，反过来就不行了，有子类型出现的地方，父类型未必就能适应。

为了增强类的可替换性，就要求我们在设计类的时候考虑以下三点：

(1) 子类型必须完全实现父类型的方法

子类型必须完全实现父类型的方法，难道还有不能实现父类型的方法？当然有，方法只是对象的行为，子类完全可以覆盖，正常情况下覆盖只会增强行为的能力，并不会“曲解”父类型的行为，一旦子类型的目的不是为了增强父类型行为，那替换的可能性就非常低了，比如这样的代码：

```
// 枪
interface Gun {
    // 枪用来干什么的？射击杀戮！
    public void shoot();
}

// 手枪
class Handgun implements Gun {
    @Override
    public void shoot() {
        System.out.println("手枪射击 ...");
    }
}

// 玩具枪
class ToyGun implements Gun {
    @Override
    public void shoot() {
        // 玩具枪不能射击，这个方法就不实现了
    }
}
```

上面定义了两种类型的枪支：手枪和玩具枪，手枪可以用来射击敌人（shoot 方法），但玩具枪就完全不同了，它不能用来射击，只是一个虚假的玩具而已，如果我们在要求使用枪支的地方传递了玩具枪会出现什么问题呢？代码如下：

```
public static void main(String[] args) {
    Gun gun = new Handgun();
    gun.shoot();
}
```

此处是一个手枪，用来射击，如果使用了子类型 ToyGun，士兵将会拿着玩具枪来杀人，可是射不出子弹呀！如果在 CS 游戏中有这种事情发生，那就等着被人爆头，然后看着自己凄凉的倒地。

此处不能替换的原因是子类型没有完全实现父类型的方法，而是丢弃了父类型的行为能力，导致子类型不具备父类型的部分功能了。

(2) 前置条件可以被放大

方法中的输入参数称为前置条件，它表达的含义是你要让我执行，就必须满足我的条件。前置条件是允许放大的，这样可以保证父类型行为逻辑的继承性，比如有这样的代码：

```
class Base {
    public void doStuff(HashMap map) {
    }
}

class Sub extends Base {
    public void doStuff(Map map) {
    }
}
```

这是 Java 的重载实现，子类型在实现父类型的同时也具备了自己的个性，可以处理比父类型更宽泛的任务，而且不会影响父类的任何行为，例如在如下代码中把父类型替换为子类型就不会有任何变化：

```
public static void main(String[] args) {
    Base b = new Base();
    b.doStuff(new HashMap());
}
```

此时，把 Base 全部替换为 Sub，所有的行为全部还是由父类型 Base 实现的，子类型的 doStuff 方法并没有调用，也就是说，子类型可以在扩展前置条件的情况下保持类的可替换性。

(3) 后置条件可以被缩小

父类型方法的返回值是类型 T，子类同名方法（重载或覆写）的返回值为 S，那么 S 可以是 T 的子集，这里又分为两种情况：

若是覆写，父类型和子类型的方法名名称就会相同，输入参数也相同（前置条件相同），只是返回值 S 是 T 类型的子集，子类型替换父类型完全没有问题。

若是重载，方法的输入参数类型或数量则不相同（前置条件不同），在使用子类型替换父类型的情况下，子类型的方法不会被调用到的，已经无关返回值类型了，此时子类依然具备可替换性。

增强类的可替换性，则增强了程序的健壮性，版本升级时也可以保持非常好的兼容性。即使增加子类，原有的子类还可以继续运行。在实际项目中，每个子类对应不同的业务含义，使用父类作为参数，传递不同的子类完成不同的业务逻辑，非常完美！

建议 149：依赖抽象而不是实现

在面向过程开发中，我们考虑的是如何实现，依赖的是每个具体实现，而在 OOP 中，则需要依赖每个接口，而不能依赖具体的实现，比如我们要到北京出差，应该依赖交通工具，而不是依赖的具体飞机或火车，也就是说我们依赖的是交通工具的运输能力，而不是具体的一架飞机或某一列火车。这样的依赖可以让我们实现解耦，保持代码间的松耦合，提高代码的复用率，这也是依赖倒置原则（Dependence Inversion Principle，简称 DIP）提出的要求。

依赖倒置原则的原始定义是：High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions。翻译过来，包含三层含义：

- 高层模块不应该依赖低层模块，两者都应该依赖其抽象。
- 抽象不应该依赖细节。
- 细节应该依赖抽象。

高层模块和低层模块容易理解，每一个逻辑的实现都是由原子逻辑组成的，不可分割的原子逻辑就是低层模块，原子逻辑的再组装就是高层模块。那什么是抽象，什么又是细节呢？在 Java 语言中，抽象就是指接口或抽象类，两者都是不能直接被实例化的；而细节就是实现类，实现接口或继承抽象类而产生的类就是细节，其特点是可以直接被实例化，也就是可以加上一个关键字 new 产生一个对象。依赖倒置原则在 Java 语言中的表现就是：

- 模块间的依赖是通过抽象发生的，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的。
- 接口或抽象类不依赖于实现类。
- 实现类依赖接口或抽象类。

更加精简的定义就是“面向接口编程”，它的本质就是通过抽象（接口或抽象类）使各个类或模块的实现彼此独立，互不影响，从而实现模块间的松耦合。那我们在项目中使用这个规则呢？只要遵循以下的几个规则就可以。

(1) 尽量抽象

每个类尽量都有接口或抽象类，或者抽象类和接口两者都具备。接口和抽象类都是属于抽象的，有了抽象才可能依赖倒置。

(2) 表面类型必须是抽象的

比如定义集合，尽量使用如下这种类型：

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

此时 list 的表面类型为 List，是一个列表的抽象，而实际类型为 ArrayList。在可能的情况下，尽量抽象表面类型，但也存在不必抽象的情况，比如工具类 xxxUtils，一般是不需要接口或是抽象类的。

(3) 任何类都不应该从具体类派生

如果一个项目处于开发状态，确实不应该有从具体类派生出子类的情况，但这也不是绝对的，因为人都是会犯错误的，有设计缺陷是在所难免的，因此只要是不超过两层的继承都是可以忍受的。特别是做项目维护时，基本上可以不考虑这个规则，为什么？维护工作基本上都是做扩展开发的，修复行为通过一个继承关系，覆写一个方法就可以修正一个很大的 Bug，何必再去继承最高的基类呢？（当然这种情况尽量发生在不甚了解父类或无法获得父类代码的情况下）。

(4) 尽量不要覆写基类的方法

如果基类是一个抽象类，而且这个方法已经实现了，那么子类尽量不要覆写。类间依赖的是抽象，覆写了抽象方法，对依赖的稳定性会产生一定的影响。

(5) 抽象不关注细节

接口负责定义 public 属性和方法，并且声明与其他对象的依赖关系，抽象类负责公共构造部分的实现，实现类准确地实现了业务逻辑，同时在适当的时候对父类进行了细化，各司其职才能保证抽象的依赖完美实现。

依赖抽象的优点在小型项目中很难体现出来，例如在小于 10 个人月的项目中，使用简单的 SSH 架构，基本上不用费太大力气就可以完成，是否采用抽象依赖原则影响不大。但是，在一个大中型项目中，采用抽象依赖可以带来非常多的优点，特别是可以规避一些非技术因素引起的问题。项目越大，需求变化的概率也就越大，通过采用依赖倒置原则设计的接口或抽象类对实现类进行约束，可以减少需求变化引起的工作量剧增的情况。人员的变动在大中型项目中也是时常存在的，如果项目设计优良、代码结构清晰，人员变化对项目的影响则基本为零。大中型项目的维护周期一般都很长，采用依赖倒置原则可以让维护人员轻松地扩展和维护。

建议 150：抛弃 7 条不良的编码习惯

很多人错误地认为编码只是熟练手的事情，其实要成为优秀的编码人员就必须进行自我剖析，抛弃不良的习惯，展示自己优秀的编码能力。通常不良的编码习惯有很多种，这里列出 7 条编码者经常会犯的错误，提醒大家注意。

(1) 自由格式的代码

如果我们不从一个团队角度出发，而是从程序员个体角度去看问题：A 项目的风格和 B 项目的风格迥异，甚至是在同一个项目的不同类中风格也不同，不用缩进，不添加注释，想加接口就加个接口，不想加就不加，我们要的是“自由，自由，还是自由”。——这不是一个成熟的职业者应该具有的，我们应该保持自己的代码风格，即使它是错的，也比没有风格要好。

(2) 不使用抽象的代码

这也算是 IDE 的便利性造成的一个习惯，一个业务类不进行抽象，直接进行编码，在需要修改时，要么依靠 IDE 批量重构，要么通过查找替换的方式重构，很方便，不是吗？而且，随着 SSH 的普及，“无抽象”编程进一步横行起来。要接口何用？想要修改的时候直接修改 XML 配置文件就可以了，要什么接口！系统间数据交互？序列化为 XML 传递，与接口何干？！

这是一种非常拙劣的习惯，抽象的意义不在于编码，而是在于对业务的理解，接口是对业务宏观的描述，而不是实现代码。

(3) 彰显个性的代码

技术人员追逐最新的技术，这本是无可厚非的，但是新技术只能作为技术的一个方向，不适合立刻投入生产中，要知道一个项目的运行质量是远远高于代码质量的，不要为了一个新颖的 API 就在生产中尝试使用，不要做小白鼠。

这里介绍一个“最小惊诧原则”（Principle Of Least Surprise 简称 POLS，或者 Principle Of Least Astonishment 简称 POLA），其意是说要使用最常见的，而不是最新颖的功能。在编码时，应寻找最常用的方法来实现，比如，同样有两个方法都能实现一个算法，选择那个最常用的，而不是那个别人一看就惊呼“哇哦，算法这么牛”的，让普通人都能看懂的代码才是最简洁的代码。

最小惊诧原则也同样适用于 UI 设计，当操作界面上两个元素冲突或重叠时，首选是那个让用户感到吃惊的元素。

(4) 死代码

可能是忘记删除的代码，也可能是故意保留的类似“到此一游”的签名代码，这些代码按照正常的执行逻辑是不可能被执行到，但是在某些未知情况下它就是执行了，此时就会产生严重的影响。

(5) 冗余代码

写了一个实现类，过了 N 天后又废弃了，之后这个类就永久地保留下去了，没人知道它为什么没被删除掉。甚至有时候竟然还能在生产机上寻找到测试程序的身影，它的生命力可谓顽强呀！估计如果有一天将生产机移植到月球上，这段代码可能还能存在。

曾经遇到过一个项目，项目中建立了单元测试机制，但在生产代码中还能看到 main 方法，谓之“测试方便”——删除它，它不应该在这里！

(6) 拒绝变化的代码

哲学上说任何事物都是在运动着的，但是我们有些代码却不遵循这个规律，一个在 JDK 1.1 中就过时的方法还还能在使用 JDK 1.6 项目中存在，谓之曰“没有坏，就不要去修它”——该重构它了，它没坏，但它赖以生存的环境已经变了！

我甚至还遇到过一个新项目还准备使用一个 5 年前的工具包（此工具包已经经历了 3 次

大的版本变更), 谓之曰“好用, 没有什么 Bug”, 但不要忘记了, 环境在前进, 我们不跟随就只能落单——不会有人陪着我们找 Bug, 不会有人去修正, 不会有人去做性能优化, 我们能做的就是孤军奋战了!

(7) 自以为是的代码

这是我们编码的最大忌讳, 认为自己无所不能, 编码不会出现任何错误, 于是不编写测试代码, 或者测试代码只是为了应付质量检查人员, 那等待我们的恶果就是系统上线后彻夜彻夜地修复 Bug——自己排除自己埋下的地雷。

自以为是还表现在对产品或工具的选型上, 相信自己编写的工具类, 而不是开源工具, 宁愿自己写序列化工具, 也不选择 kryo 或 protostuff; 宁愿自己写日期处理工具, 也不选择 Joda 或 date4j; 宁愿自己写批处理框架, 也不选择 Spring Batch, 这样是不行的! ——相信天外有天吧, 更多更好的工具等待着你去发掘。

建议 151: 以技术员自律而不是工人

技术人员和工人有什么不同呢? 这么来说吧, 在工厂的流水线上, 工人的任务是过来一个零件就把它安装在规定的位置上, 而技术人员的任务则是确定零件的尺寸、材料、安装位置等, 而且一旦出现问题, 技术人员还要能够查明出现问题的原因, 并且提出解决办法, 我们 Javaer 也应该是这样的, 这就需要我们逐步培养自己, 在提高自己技能的同时也提高自己的思维方式, 以下 20 条建议可以逐步把我们向技术人员方向培养。

(1) 熟悉工具

军人手中有枪, 农民手中有锄头, 而我们手里只有 Java, 这也是我们能够引以为豪的工具, 我们应该了解它的使用范围, 了解它的生态系统, 了解它的发展趋势——它也可能就是陪伴我们一生的那个工具, 也祝愿它是。

(2) 使用 IDE

在技术领域, 不要相信“无刀胜有刀”之类的鬼话——“高手都用记事本或 VI 开发”, 建议选择 Eclipse 或 NetBeans 作为开发工具, 而且坚持不移地使用它。

(3) 坚持编码

不要考虑自己的职位、岗位, 只要是 Java 圈子的生物都应该坚持编码, 没有编码, 就等于是无源之水, 无本之木, 何来灵感和灵性?

(4) 编码前思考

在坐下来开始编码之前, 必须已经完成设计, 最低要求是对开发中遇到的问题有清晰的认识, 不要在编码中解决问题。

(5) 坚持重构

不要相信一次就能写出优秀的代码, 这是不现实的, 任何优秀的代码、算法都是经过多

次重构磨练的，坚信自己的下一个版本或代码更优秀。

(6) 多写文档

写注释、写说明、写报告都是对代码或项目的回顾和总结，不仅仅是为了后续的参与人员，同时也是为了整理自己头脑中混乱的思维。

(7) 保持程序版本的简单性

一个项目不要保持多个版本，即使有分支也必须定义出项目合并的条件，或者时间约束，或者目标约束，不可任由版本扩散。

(8) 做好备份

世界上没有万无一失的事情，不做备份，一旦灾难发生就无挽救的余地了，经常把代码拷贝到不同的主机上备份是一个好习惯，如果能够自动备份那将是一个非常好的方式。

(9) 做单元测试

单元测试不仅能增强你的信心，也能给你带来好名声——后续者一看，“哇哦，单元测试写得这么完整，肯定是一个认真、负责的人”。

(10) 不要重复发明轮子

在项目中使用已经成熟的工具或框架，而不是自己编写。但是如果想共享一个新的 MVC 框架，那就尽管去重复发明轮子吧，它不是以交付为目的的，而是以技术研究为目标的。

(11) 不要拷贝

当您按下 Ctrl+C 的时候，问问自己“我在做什么？拷贝是否是唯一能做的？为什么不能重构一下呢”，不要让大段的代码散落在各处，不要做搬运工，不要做拷贝工，要做技术工。

(12) 让代码充满灵性

为变量、类、方法起个好听的名字是一个不错的主意，为代码增加必要的注释也是很好的办法，“One Line”能解决一个上百行代码的问题，也是一个优秀的实现。

(13) 测试自动化

不管是性能测试、单元测试，还是功能测试，想尽办法让它自动化，不要在测试之前手动配置或触发条件，这不够人性化，也同时让代码“汗颜”——本是用来自动执行的，但却被手动设置了条件。

(14) 做压力测试

不要相信业务人员“最多 200 个用户使用”之类的话，把业务人员制定的指标扩大 3 倍，然后再做压力测试。不要迷信自己的代码很健壮，在高并发时只有上帝知道发生了什么事，你又怎么能知道？

(15) “剽窃”不可耻

多看开源代码，学习一下人家是如何编码的，然后经常“剽窃”一下，这也是提高技能的最佳途径，我们不是孔乙己，“剽窃”不可耻。

(16) 坚持向敏捷学习

不管“敏捷”与“非敏捷”之间的争论有多激烈，敏捷中的一些思想是非常优秀的，例如 TDD 测试驱动开发、交流的重要性、循序渐进开发等。

(17) 重里更重面

UI (User Interface) 是“面”，Java 程序是“里”，客户首先感受到的是“面”，然后才是“里”，要想获得良好的第一印象，那就需要有一个简洁、清晰、便捷的 UI，即使“金玉其外败絮其中”，我们也可以继续重构。

(18) 分享

“独乐乐”不如“众乐乐”，把自己的代码分享出去收获的不仅仅是赞许，还有自己能力的提升——暴露出自己的 Bug，在众目睽睽之下修正之，知耻而后勇也。

(19) 刨根问底

有问题不可怕，可怕的是掩盖，或者虚假掩盖，“哦，这个问题呀，加上这个参数就可以解决了”——这不是解决问题的办法，在答案之后加上“是因为……”，这才是解决了问题。

(20) 横向扩展

Java 要运行在 JVM、操作系统上，同时还要与硬件、网络、存储交互，另外要遵循诸如 FTP、SMTP、HTTP 等协议，还要实现 Web Service、RMI、XML-RPC 等接口，所以我们必须熟悉相关的知识——扩展知识面，这些都是必须去学习的。

技术人员的武器就是技术，我们 Javaer 的武器就是 Java，如果我们能驰骋沙场，唯我独尊，而且屹立不倒，那就是我们成长为顶尖技术高手的时刻，朝着这一目标奋斗、努力吧，总有一天我们能够与 James Gosling (Java 的创始人之一)、Rod Johnson (Spring 项目的创始人)、Gavin King (Hibernate 的创始人) 坐而论道，煮酒论 Java 也！



深入理解Java虚拟机

JVM高级特性与最佳实践
Understanding the JVM
Advanced Features and Best Practices

周志明 著

■系统内存管理、执行子系统、程序编译与优化、高效并发等核心内容对JVM进行了全面而深入的分析，深刻揭示JVM的工作原理

注重实战，以解决实践中的疑难问题为首要目的，包含大量经典案例和最佳实践

作者：周志明 著 ISBN：978-7-111-34966-2 定价：69.00 元

Java程序是如何运行的？Java虚拟机在其中扮演了怎样的角色？如何让Java程序具有更高的并发性？许多Java程序员都会有诸如此类的疑问。无奈，国内在很长一段时间里都没有一本从实际应用的角度讲解Java虚拟机的著作，本书的出版可谓填补了这个空白。它从Java程序员的角度出发，系统地将Java程序运行过程中涉及的各种知识整合到了一起，并配以日常工作中可能会碰到的疑难案例，引领读者轻松踏上探索Java虚拟机的旅途，是广大对Java虚拟机感兴趣的读者的福音！

——莫枢 (RednaxelFX) 虚拟机和编程语言爱好者

在武侠的世界里，无论是至刚至强的《易筋经》，还是阴柔无比的《葵花宝典》，都离不开内功修炼。没有了内功心法，这些武术只是花拳绣腿的拙劣表演而已。软件业是武林江湖的一个翻版，也有着大量的模式、套路、规范等外功，但“外功修行，内功修神”，要想成为“扫地僧”一样的绝世高人，此书是必备的。

——秦小波 资深Java技术专家 / 著有畅销书《设计模式之禅》

对Java程序员来说，Java虚拟机可以说是既熟悉又神秘，极少有Java程序员能够抑制住自己探究它的冲动。可惜，分析JVM实现原理的书籍（特别是国内作者原创的）是少之又少。本书的出版可谓Java程序员的福音，作者将自己多年来在Java虚拟机领域的实践经验和研究心得汇集在了这本书中，不仅系统地讲解了Java虚拟机的工作机制和底层原理，而且更难能可贵的是与实践很好地结合了起来，具有非常强的实践指导意义，强烈推荐！

——计文柯 资深Java技术专家
著有畅销书《Spring技术内幕：深入解析Spring架构与设计实现原理》

构建安全Java应用的权威经典，5大社区一致鼎力推荐！

作者：梁栋 著 ISBN：978-7-111-29762-8 定价：69.00 元

如果你在思考下面这些问题，也许本书就是你想要的！

作为一名系统架构师，如何让你的系统不留有安全隐患？作为一名程序员，如何让你编写的代码没有安全漏洞？

为什么密码学是解决一切安全问题的银弹？密码学究竟是怎样一门学科？近千年 来，它经历了怎样的发展历程？它是如何延续至今并逐步发展壮大的？

博客、论坛、社区、网络聊天、企业级数据交互应用、网银平台等网络应用都无法逃避网络安全问题，如何在合适的环节选用合适的加密算法，从而提高系统的安全性？

Java 6支持哪些加密算法？如何扩充Java 6尚不支持的加密算法？如何增强系统的安全级别？

消息摘要算法和文件校验算法有什么关联？它与普通的循环冗余校验算法有何差别？如何使用消息摘要算法隐蔽敏感信息？

为何Base 64算法可以隐蔽敏感信息但却无法真正起到数据加密的作用，而对称加密算法却能轻而易举地起到数据加密的作用？Base 64算法与对称加密算法之间究竟有何关系？

Sun并没有提供官方的Base 64算法支持，我们又该如何构建该算法？针对Base 64算法，Apache Commons Codec和Bouncy Castle提供了怎样的支持？在其他加密算法中又起到了怎样的作用？

对称加密算法已经几乎能胜任所有的加密需求，为何要研制非对称加密算法？对称加密算法究竟有何弊端？非对称加密算法会是对称加密算法的替代者吗？

数字签名是手写签名的数字化产物，其算法与消息摘要算法有何关联？为什么这种算法在结合非对称加密算法密钥后就具备了认证身份的作用？

对称加密算法和非对称加密算法如何分发密钥，数字证书在其中充当了何种角色？数字证书又是如何发放的？

数字证书集多种加密算法于一身，它是如何传递密钥的，又是如何起身份认证作用的？在HTTPS协议中又是如何与SSL/TLS协议相结合构建安全平台的？

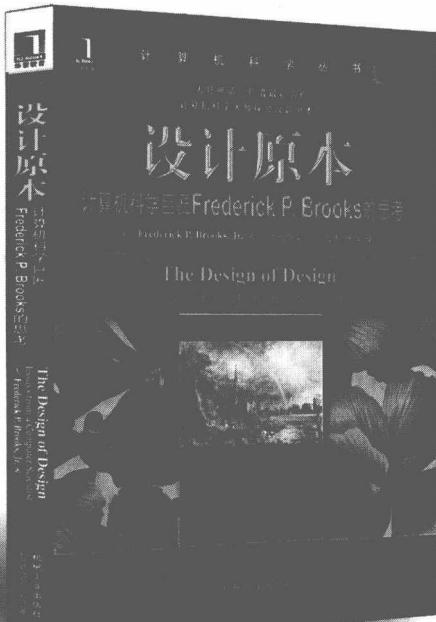
KeyTool和OpenSSL构建的数字证书究竟有何差别？如何在Java中使用这些工具构建的数字证书？

基于HTTPS协议的网银平台，堪称安全级别最高的网络应用，更是密码学应用领域最为成功的案例。Java 6提供了完备的HTTPS协议相关的API，如何使用这些API构建固若金汤的HTTPS平台？

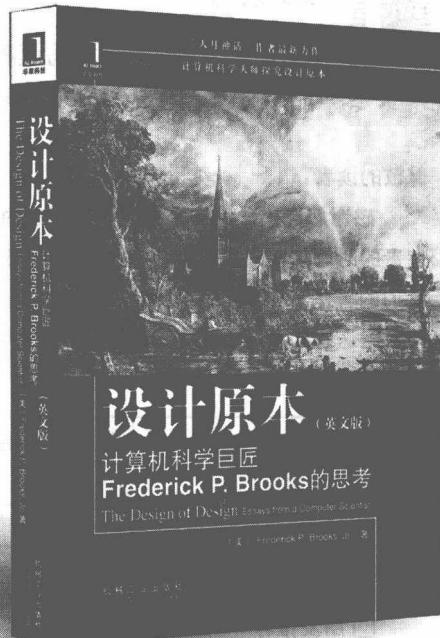




一本打开的书。
一扇开启的门。
通向科学圣殿的阶梯。
托起一流人才的基石。



ISBN: 978-7-111-32557-4
定价: 55.00

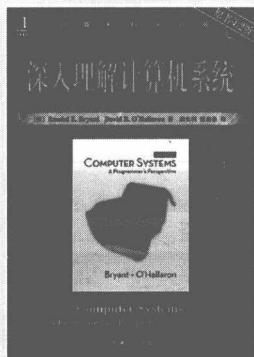


ISBN: 978-7-111-32503-1
定价: 69.00

《人月神话》作者最新力作 计算机科学大师探究设计原本

本书包含了多个行业设计者的特别领悟。Frederick P. Brooks, Jr.精确发现了所有设计项目中内在的不变因素，揭示了进行优秀设计的过程和模式。通过与几十位优秀设计者的对话，以及他自己在几个设计领域的经验，作者指出，大胆的设计决定会产生更好的结果。

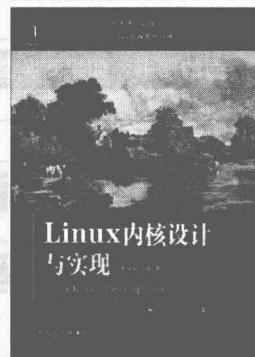
本书几乎涵盖所有有关设计的议题：从设计哲学谈到设计实践，从设计过程到设计灵感，既强调了设计思想的重要性，又对沟通中的种种细节都做了细致入微的描述，并且谈到了因地制宜做出妥协的具体准则。



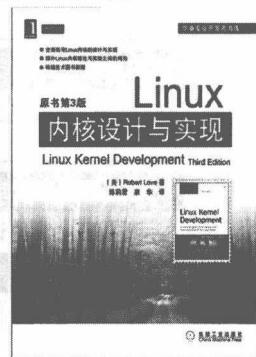
深入理解计算机系统（原书第2版）
ISBN: 978-7-111-32133-0
定价: 99.00



深入理解计算机系统（英文版·第2版）
ISBN: 978-7-111-32631-1
定价: 128.00



Linux内核设计与实现（英文版·第3版）
ISBN: 978-7-111-32792-9
定价: 69.00



Linux内核设计与实现（原书第3版）
ISBN: 978-7-111-33829-1
定价: 69.00元



专业成就人生
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会
获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：

书号：7-111-()

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

朋友推荐 书店 图书目录 杂志、报纸、网络等 其他

2. 您从哪里购买本书：

新华书店 计算机专业书店 网上书店 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

是，我的计划是_____ 否

7. 您希望获取图书信息的形式：

邮件 信函 短信 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收

邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzsj@hzbook.com