Java 编程四部曲之一：Java 基础

刘延栋

2024-09-11

目录

插图目录

列表目录

Java 编程四部曲

为什么我一下子有了四本书稿呢?

长期听我电台的人可能知道，大约 4 年前我就说，我正在写一本 “Java 编程” 的书。第一次写书，激情满满，很快就写完了 “初稿”，就跟当初录电台一样，一天录一期。但是 “初稿” 与 “出版” 之间的距离非常遥远——大约是大唐到灵山的距离吧，期间要经历九九八十一难，才有可能出版。我以前又没出过书，算是初生牛犊不怕虎，写就完了……

图 1: Java 编程四部曲封面

“写书”与“录电台”是容易形成“习惯”的，把书稿交给出版社后，就跟失恋了一样，心里空落落的，还是想每天写点东西。我有个“自以为是”的优点，我能把一件事情坚持下来，坚持很久。

> ❗ “坚持”并不一定是好事，要分情况
>
> 以前我“天真”的认为，坚持是一种好的品质，但是在遭遇很多磨难之后，我才意识到，如果“对错误的人与事，坚持不如早点放弃”。
> 对个人可以掌控的事情——这种事不多——坚持还是有用的。比如我坚持每个月录 3-4 期电台，坚持了 9 年，现在正在奔向 500 期电台的路上，我觉得这已经是我的生活休闲的方式。
>
> 比如我坚持背单词，当年听了新东方俞敏洪的忽悠，坚持背单词，后来我就成了学霸，英语大概都能考到 90% 的分数。
>
> 比如写书，我当初加入了豆瓣一个小组，叫每天写 1000 字……刚开始觉得有点难，后来，我每天写技术书，平均可以写 2500-3000 字，这就是这四本书的来历。
>
> 但是，有些事情是坚持不能改变的，就像你不能坚持每天用一杯 25 摄氏度的水，想用 4 天的时间凑出 100 摄氏度的水。如果路一开始就走错了，越坚持，错的就越离谱。

出版一本书，更多的是“修改”，而创作一本书，更多的是“炫耀”。我喜欢“炫耀”我懂的多，而不是“修改”细枝末节。

不经历过不知道，在漫长的等待过程中，我觉得我还是继续写点书吧，毕竟只写一本，好像“虚荣”的不够劲。就像我想抽大烟，结果只抽了一根香烟，不过瘾。索性又写了三本，当时的想法是出一本，然后再出一本，继续出一本，最后出一本，搞个“Java 编程四部曲”，过过瘾。

> 💡 著名的几部曲
>
> 比较出名的东西，都是三部曲，四部曲呢……
>
> 比如《家》《春》《秋》、《教父》、《黑客帝国》、《多情剑客无情剑》、《边城浪子》、《九月鹰飞》、《天涯 · 明月 · 刀》、《飞刀，又见飞刀》……
>
> 所以我也想试试。

九九八十一难，结果第一难都没熬过去。主要还是我个人生活上的失败，遇人不淑，

滥生无辜……人到中年，低头发现胸口插了一把刀子，还是从背后插的。

死倒是没死，但是痛是真痛。浑浑噩噩的开始收拾残局，该断的断，该舍的舍，该离的离。

其中就包括这几本书，电台的听众，总是时不时的问起我，我只能说，再等等看。一般来说，我都把初稿发给他们一部分，比如前 4 章。第一本书我写了 21 章，跟出版社搞了 3 年，才磨到第 5 章。如果按照这个进度，等着出版，要 10 年吧。再加上 Java 市场低迷，人工智能异军突起，传统的编程书就是鸡肋啦，我自己也衡量了一下，我的书，若是出版，极大的概率是最后按斤卖给废品站回收处理，对地球也是一种伤害。

不应该的坚持，对别人是一种伤害，不如放手。出版的梦也就惊醒了，结果，我现在手里有了 4 本书的书稿:)

这四本书要 100 万字，不想开源，想卖电子书

这四本书我就在网上卖电子版好了。这四本书总共 200 元。因为是电子书，按照互联网的规则，其实只能“卖”一本，然后其它就可以“免费”阅读了。这个我是知道的，所以我搞了一个社区叫学伴：https://xueban.app，如果你购买了这四本书，我会拉你到这个社区里。

这个社区的主要话题是：**学技术、学外语、搞事业**。在购买页面里，我会详细讲一下我做这个社区的想法。

如何购买呢？我专门写了一个购买的介绍，介绍了这四本书的购买。

链接如下：如何购买这四本电子书?

付款方式与学伴社区

付款方式

目前我没有好的方式，也没有第三方可以担保，以后我可能会开个淘宝店什么的，或者加个 stripe 收款，但是目前还没有。

如果我贸然放一个微信收款，很不合理，就跟加了个“美女”，上来第一句话就是今天她过生日，给她发个红包再聊一样荒唐。

我想把这事做成一个长久的“买卖”，总有一少部分人想学习编程，而我在编程这个领域工作了 20 年。这个买卖的本质是我以电子书和社区的方式售卖知识与服务。除了获得 4 本电子书，这 200 块的门槛足以阻拦白嫖党加入社区。

目前，我只服务加过微信的电台听众。

你发给我红包或者转账以后，一定要提供一个可用的电子邮件，我会把社区的邀请链接发给您。社区不支持公开注册。

学伴社区

电子书很容易就复制了，可能我也赚不到什么钱。这 200 块钱里，有一项服务是不可取代的，我搞了一个社区，叫学伴，这个名字不是山东大学启发我的，网址是https://xueban.app。如果你购买了这 4 本书，我会给你一个社区的邀请链接，这个社区的主要话题是：**学技术、学外语、搞事业**。

所谓的搞事业，有可能是学生想出国读书了，找个便宜点的国家；也有可能是找个远程工作，但是不知道怎么弄；也有可能是移民到国外，不要再这么卷了；如果你一直打工，到了 35 岁，HR 发个邮件通知你，明天你不用来了，在社区里，咱们谈谈如何尽量避免这种情况…… 我以后会在电台里多邀请一些在国外学习工作生活的人，采访一下他们，给大家，包括我，开开眼界。

> 如果没有审核资源，社区一定要有个付费墙
>
> 有本书叫《怪诞行为学》，作者在书里讲了一种现象，如果你把布朗尼——一种巧克力蛋糕——设置为免费，不管需要不需要，人们都会趋之若鹜，排起长队。只要你设置一个价格，哪怕只有 1 美分，也会让排队的人消失。他把这个现象起了个名字叫布朗尼效应。在中国，会更严重一点。如果你搞了个免费公开的社区，就会有人来请你
>
> 喝茶——实际上没有茶，也没有水，更没有布朗尼蛋糕，只有写保证书，并关掉社区这一条路。我试过，可以说是赔了金钱又剪了网线，不能再细讲了。

我卖电子书，也是践行**搞事业**的理念。打工了这么多年，我希望成为一个会做生意的人。目前，我理解的生意人，应该是提供相应的服务，收取相应的报酬。

这 4 本书，我都会提供前 4 章的预览，你可以先看看前 4 章，再决定要不要花这个钱。我给这几本书定了一个上线的日期。

书名	目前完成度	完整上线日期	提供的格式
《借助 AI 学习 Java 编程》	100%	2024.9.30 前	pdf
《借助 AI 用 Java 制作 2D 游戏》	100%	2024.12.31 前	pdf
《借助 AI 用 Java 制作 GUI 软件》	90%	2025.3.31 前	pdf
《Java Spring》	50%	2025.9.30 前	pdf

> 提示
>
> 我确实想提供 epub 的格式，但是…… 代码在 epub 里实在是太丑了，看不下去:(

从做电台来看，我还是比较能坚持的，**我想做一个坚持长期主义的生意人。写书，录电台，录中英文编程视频……** 我也要践行学技术、学英语、搞事业的理念。

既然结局已定，在死神来敲门之前，好好体验人生吧……我想成为做编程教学最好的电台主播，太虚的大话就不说了，免得像出书一样，被现实狠狠地打脸。

> 只要加足够的定语，就一定能成为最好的。
>
> “我想成为做编程教学最好的电台主播”，肯定有人不相信，是的，我知道至少有一个人不相信。那么，我再加几个定语吧：我想成为做 Java 编程教学最好的山东籍电台男
>
> 主播…… 只要定语足够多，那么，一定能成为最好的。

前言

说来惭愧，我本来想出版一本书的，由于诸多原因，书没法出版了。但是有一部分人——主要是电台的听众——帮我修定了书中很多错误，我本来说好会把他们的名字印在纸书上，但是，又食言了。把这些贵人的名字写在这里了，表示抱歉与感谢。下面是“原本的前言”。

> “原本的” 前言:)

本书旨在为对 Java 编程感兴趣的读者提供入门指南，无论您是希望开发安卓手机应用、网站、还是中间件，都能从中获取重要的基础知识。正如副标题“有趣有料的 Java 编程” 所言，本书不仅涵盖了编程语言和编程范式的基础知识，还穿插了大量引人入胜的编程故事，并深入挖掘其背后的历史真相。

例如，在介绍面向对象编程时，本书用了一定篇幅讲述了世界上首个面向对象编程语言 Simula，并梳理了面向对象编程的发展脉络。在数据类型方面，本书列举了一些因数据类型精度不足而导致的案例，如爱国者防御系统故障，导致一枚飞毛腿导弹命中士兵营房。而在介绍 Java 引入的函数式编程时，则从世界上第一门函数式语言 Lisp 讲起，带领读者认识 Java 函数式编程的优势与不足。

通过这种将历史与实践相结合的方式，本书不仅阐释了 Java 编程的核心概念，更揭示了编程语言发展背后的故事，希望能为读者带来更深层次的理解和启发。

我始终认为，学习编程语言的最佳途径之一，便是追溯其发展历史，了解其起源和演变。通过在历史中考证语言的进化轨迹，并在实践中学习编程技巧，我们能够更深入地理解语言发展的脉络，更真切地领会语言设计者的意图。

关于如何学习编程，我联想到清代文学家彭端淑所写的《为学》一文，这也是很多地方初中必学的古文。文中讲述了两个四川和尚前往南海的故事。富裕的和尚声称他准备了好几年，要买一艘船沿长江而下。而贫穷的和尚只带了一个水瓶和一个饭碗。一年后，穷和尚从南海归来，富和尚却仍在准备之中。这个故事告诉我们，行动胜于空谈。学习编程亦是如此，与其踟蹰不前，不如立即行动，在实践中不断摸索和学习。

关于本书

如果几年前，有人问我：“出版一本书” 是不是你人生的梦想之一？我会不假思索地回答他：“这是我来生的梦想。”

如今，我竟然写了一本书，我认为这得益于以下几个因素：首先，我活跃于网络（现在又有谁不是活跃在网络上呢？），自 2016 年起便开始录制名为 “软件那些事儿” 的 podcast，至今已录制了 400 多期节目。此外，我还经常录制编程视频并上传至 Bilibili 等视频网站。
其次，我拥有多年的软件开发经验，职业生涯始于 Sun 公司——正是这家公司创造了 Java。尽管我已经离开老东家多年，但心中仍存留着一份特殊的情感。这份情感也延伸至 Java，我对这门编程语言抱有许多好感。也正是因为这份情感，让我录第一期 podcast 的时候，就选择介绍 Java 语言的历史。现在写一本编程书，我首先想到的也是 Java 语言。

最后，也是我认为最重要的一点，我意识到，曾经在编程中所经历的许多纠结与折磨，本是可以避免的。从年轻时在 NetBeans 项目组摸爬滚打，边工作边学习，却苦于无法完全掌握 NetBeans 的运作原理的职场新手，到如今成长为能够较为镇定地面对大型项目的架构师，这个过程中，我遇到了无数挑战和困难，有些坑如果当时有人指点或许能够更早地爬出来。如今已是 2021 年，我发现仍有许多新人像当年的我一样，在同样的坑中踽踽独行。我希望通过本书，能够帮助他们尽快摆脱困境。

此外，由于种种历史原因，Java 的许多先进语言特性和开发模式在国内尚未得到广泛应用。我期望通过本书，倡导采用先进的语言特性，提升程序员的工作效率，使他们能够早日完成工作，提前下班。

要感谢的人

本书还邀请了众多我的电台听众担任早期读者，他们提供了宝贵的意见和建议。在此，我谨按反馈顺序向以下听众兼早期读者致以诚挚的谢意：

张敬畏、吉人、任占东、李振中、毕朋飞、张洁、胡海彦、郝金、王义祥、王恒、黄建晴、弥国伟、黄寒毅、戴峰、代亮、杨兆军、刘杰、唐青平、吴剑钢、王晨、常黎明、王卜锐、严济愈、范宗耀、刘强、陈少川、王之康、付志远、付小辉、邓俊德、赵庶剑、吴昱锡、金希朴、刘天尧、陈尧龙、刘凡恺、徐亚南、彭海璐、王琦、张训、李庆权、贾爱国、车恒彬、陈喆、刘冕、夏明飞、王超、乔鑫、杨碧军、李维洲、苏三、袁宏、徐玉峰、刘卫、周信、孙国庆、黄鑫源、王晋渤、黄国瑞、欧阳宁。

还有一些朋友不愿透露真实姓名，仅提供了网络 ID：傻牛、大成、浮游、素颜、touchmii、Coding L、推头的拓海、一贯可乐、腾、YuVenhol、wym、菜得抠脚、99、老法师、红薯、雯雯、城哥、丫头、葫芦娃、王女士，船哥、机器学习、秀儿、队长、黄盛、伴娘、Hmtsai、AUGUST、Andy、龙小龙、天涯明月刀、橙子和西瓜、TheWQ、Abel、肥仔强、alibos、NULL、hwf1324、阿强、Tr。

你们的真知灼见令本书增色不少，在此深表感谢！

再次对这些早期读者表示感谢，对没法把名字印在纸上再次表示抱歉。

为什么你读不完一本书?

这不怪你，因为任何知识，都有前置条件。所谓的前置条件，就是为了读懂第 1 章的概念，你需要读到第 9 章才会恍然大悟。

你可能会问，为什么不先写第 9 章呢？原因是相同的，如果你要读懂第 9 章的那个概念，要用到第 4 章里的概念。

越复杂越系统的专业或者技术书籍越是如此，从入门到精通是理想的状态。

且不说如何入门，其实，门在什么地方，都不一定能说得清楚。因为专业的知识，到处都是门，你只要从某个地方进入即可，只要深入研究这扇门，一定能将你传送到某个地方，最终会让你豁然开朗。

举个例子，无论你在大学里学哪个专业，这个专业的知识都被分解成了一门门的课程，这些课程之间互相联系，又分别独立。以计算机科学为例，《计算机组成原理》、《数据结构》、《Java 编程》、《网络与通信》、《操作系统原理》……这些课程之间绝对不是独立的，之间有千丝万缕的联系。

知识是网状的结构，而不是线性的。我们学的每一门知识，都像是一个迷宫。每个节点代表一个知识点，知识点越多，迷宫就越复杂。

下图是计算机网络知识的联系图，如果你要入门，那么，哪里才是真正的门?

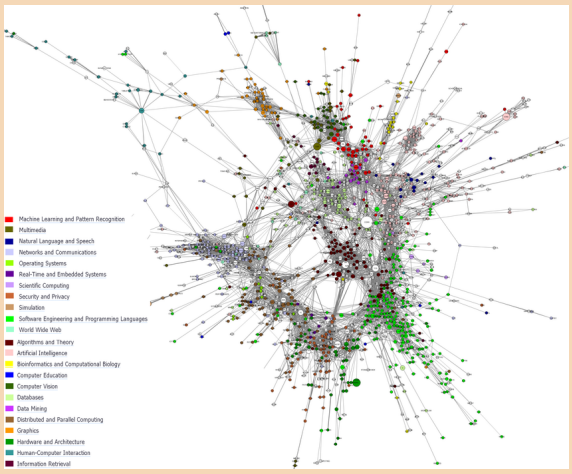

图 2: 计算机知识图谱太复杂

很难说清楚应该从哪个门进入，我们唯一能做的就是无论从哪个门进入，都要在这个迷宫一样的网络图里探索。网络图的特点是，每个节点都不是孤立的，你总可以从一个节点到另一个节点。

很多人读不完一本技术书籍的原因是：看到不懂的就放弃了。如何解决这个问题呢？也很简单：碰到不懂的不要放弃，而是先放一放，略过这个节点，到下一个节点。知识点不同于爱情，走过路过不要错过。知识点是只要你走过路过，就不会错过，总有那么一个时刻，你会突然间想到那个你当时不懂的点，然后豁然开朗。

因此，读书一定要养成的一个习惯是：**反复阅读教材 + 做题**。

反复阅读教材可能是最好的自学方法。以我为例吧，我读的高中是我们那个市最差的高中，学校坐落于农村，那个高中只招，也只能招周围农村的学生。别说城里的孩子了，就算农村里有点见识的家长，也不忍心孩子上那个高中。升学率有多低呢？每年可能招 700 个学生，能考上本科的人数，一般在 40-50 人左右。对，你没看错，不是 400-500 人，是 40-50 人，我高考的那年，好像是 39 人。我母校的命运也极为悲惨，由于名声在外，只要去读我那所高中，约等于高考失败，越来越多的学生和家长不再报考这所高中，生源就越来越差，最后实在是招不到学生了，就解散了，老师合并到了另一个学校。

我在这所高中里，考上了重点本科。给了我很大的信心，后来考研究生就不在话下了。虽然日后证明，光会考试的农村做题家其实没啥用（多么痛的领悟 !），但是，我的法宝就是反复阅读教材，随机的刷题。

> **i** 反复阅读、随机刷题有用么？
>
> 我觉得是有用的。有一本很出名的认知学的书叫《make it stick》，作者是 Peter C Brown，

> 我读这本书的时候，有一种似曾相识的感觉，这不就是我从高中以来一直使用的方法么？想要掌握知识，就要反复阅读加做题，如果只是重复的阅读，效果并不好。而是
>
> 要不停的反问自己，情景模拟，使大脑重建立记忆。这一点，在编程中的体验，我认为是读书，写代码，调试。调试的时候，就是长脑子的时候。

很多人读不完一本技术书籍的原因是因为太难了，但是太难了的原因是没有读完。实际上，如果能继续囫囵吞枣的读下去，再回来看前面，就能理解一些了。也就是说，书不是因为难才读不完，而是因为没读完才显得难。

我曾经给孩子报过美术课，因为家长总是觉得自己的孩子与众不同，万一我家孩子是个美术天才呢？学一段时间以后，才知道原来也是普通人，这样就安心了。孩子学了多少我不知道，但是我觉得那个叫韦唯的美术老师——之所以我能记住他的名字，是因为他的名字跟一个歌星一样——给我说的一句话，让我受用终生。我说，你看我家孩子画的，可真难看。这个老师说，画画都是这样，画到一半的时候，谁的画都不好看，只有完成了，画才好看。

我觉得这句话非常正确，软件也是如此，做软件，做到一半的时候，到处是 bug，运行起来颤颤悠悠的，但是慢慢的就修复了。很可惜的是，我们很难看到只画了一小部分的画，也不太容易见到 bug 到处都是的半成品。

读书也是如此，读到一半的时候，最难了，前面的搞不懂，后面的感觉更搞不懂。于是，大部分人读书就是读个前言，读个开头，就认为自己不是学这个的料。或者有些人一定要把所有的都搞懂，才肯继续下去，这样的结果要么是反复地困在第一第二第三章，要么彻底放弃了。

书，看不懂也要看完，反复的看，读书百遍，其义自见。

还有一点，你不用了解编程的所有知识，只需要了解你需要的那些就够了。就像你在一座城市里住了那么久，你知道的可能只是你周边的那一部分，至于其它地方，不知道也无所谓。如果要去，GPS 导航一下，也就到了。比如要做一个 2D 游戏，你暂时不用管什么 Java 网络编程，即使 Java 网络编程非常重要，但是，目前真用不上，你就先不用操心了。等到你熟悉了 Java 2D 游戏编程，会发现，原来 Java 网络编程也没那么难，你已经会了 90%，甚至更多。

这时候，就需要信心与耐心了。有些人有信心，但是总是三分钟热度，什么事情都想马上出成果，恨不能今天看了 Java 的编程书，明天做一个 Minecraft 游戏，卖给微软。还有一些人有耐心，但是没信心。不管你认为你能行还是不能行，你都是对的。如果你打心底就认为这玩意学不懂，太难了，那么，肯定是学不会的。

一不要急于求成，二不要妄自菲薄。静下心来，全神贯注的投入 200-300 个小时，一

定能搞清楚的。想一下，每天投入 3-4 个小时，2 个月就可以了。只要学会了一门编程语言，再学其它的，都会简单太多了。

每章内容概括

第 1 章初识 Java

> 提示
>
> 本章介绍了 Java 的历史，不是说“历史是一面镜子”么?

1.1 JAVA 简介
1.1.1 Java初次登场
1.1.2 指导Java发展的两份文档
1.1.3 Java面对的问题以及解决方案
1.2 JAVA 语言进化史
1.3 小朋友，你是不是有很多问号?
1.3.1 为什么James Gosling被称为终身仁慈独裁者?
1.3.2 Java的名字是谁起的?
1.3.3 James Gosling如何回答相比于C语言，Java不够自由?
1.3.4 Java的现状怎么样?
1.4 程序员故事

第 2 章第一行代码

> 提示
>
> 这一章的内容主要是把开发环境配置起来，运行第一个程序，在屏幕上输出两个单词：Hello World。

2.1 “传统”JAVA 版的 Hello World

第 3 章面向对象编程

> **提示**
>
> 这一章讲了面向对象编程的历史，并考证了面向对象编程这几大知识点的由来以及用途。

第 4 章变量

变量是构建程序最基本的单位，本章从变量的历史讲起，涵盖了变量命名、变量类型、变量作用域等内容。

第 5 章操作符

小学时候就学了算数操作符与赋值操作符，可能逻辑操作符有一点难度，至于自增、自减那些，一语也就点破了，总之，本章实在是没难度。

第 6 章结构化编程

人类开发软件之后，遇到了第一次软件危机，经过众多科学家的折腾，终于捯饬出来了结构化编程……简单来说，**所有的编程都可以用顺序、选择与循环来搞定**，我给出了他们当初的论文来证明这一点。这一章算是非常重要的。

第 7 章方法

英国科学家大卫·惠勒（David Wheeler）发明了方法，从此所有的编程语言都开始支持方法。可以说，没有方法，无法进行编程。同时，递归是一种比较特殊的方法，如果运用恰当，可以大大简化编程。

第 8 章数组

警告

数组是一种古老的数据结构，太过于“原始”，我觉得应该使用更先进的 ArrayList 等数据结构。但是，数组仍有大量的人在使用，所以，我们也要学习。

第 9 章类、对象与封装

如何把功能相关的代码组织到一起呢？答案是用“类”。

第 10 章包

如何把功能相关的类组织到一起呢？答案是用“包”。

第 11 章继承

权力可以继承，财富可以继承，代码就不可以继承么？当然能！继承最重要的功能是为了代码重用，减少代码的重复。官/富二代最重要的功能也是继承富一代的权与钱，减少从零奋斗的过程。

第 12 章抽象类

在《公孙龙子 · 白马论》中提出了“白马非马”的诡辩，如果第二个马是抽象类的话，那么“白马”就是“马”。“公孙龙”认为“马”不能实例化，但是“白马”可以实例化。我认为战国时期的公孙龙所持有的“每个事物只能是它自己，事物间不存在联系”的观点是错误的，至少在面向对象中是如此。不能实例化的“马”，在面向对象编程中叫“抽象类”。

第 13 章接口

在 11 章里讲过，Java 不能多重继承，爸当官，妈当官，难道孩子不能同时继承两者的权力与财富么？在现实中，肯定能！（不信你看哪个公子，哪条鲶鱼，都是多重继承的……）但是在 Java 中却不能，Java 不支持多重继承，如果你需要多重

继承怎么办？用接口。http://www.gotw.ca/publications/c_family_interview.htm 在这

篇访谈中，C++ 的作者与 Java 的作者，都谈到了编程语言的设计，其实 Java 的作者对接口与类的设计也不满意。

第 14 章多态

什么动物婴儿时四条腿，成年后两条腿，老年时三条腿？答案是：人。当我们调

用走路的时候，同样的人，但是有不同的走路形态，小时候爬，长大后两条腿，老年后拄着一根拐杖。这就是多态，用统一的方式（走路），却实现了不同的形

态（爬，走，拄拐）。本章介绍了如何用不同的方法实现多态。

第 15 章集合

第 8 章的时候，我“黑”了一下数组，数组作为一种数据结构，实在是太原始了。那用什么方法来代替数组呢？就是本章讲的集合。集合比数组好用太多了，不一个时代的产品。

第 16 章异常处理

嫦娥应悔偷灵药，碧海青天夜夜心。嫦娥犯的错，只能无限的懊悔，毕竟没有后悔药卖。但是 Java 中，为异常处理设置了非常多的防御措施，确保嫦娥偷不到灵药，防患于未然:)

第 17 章函数式编程

绝大部分 Java 编程书上，都不讲泛型，一是太难，二是觉得没什么用。但是，看看下面的语言：

Java之前不支持泛型，Java 5之后增加了泛型。
C#之前不支持泛型，C# 2.0之后增加了泛型。
Go语言之前不支持泛型，Go1.18之后增加了泛型。

难肯定是难的，有用肯定是有用的，花点时间是值得的。泛型解决的问题与继承、

多态解决的问题是一脉相承的：**代码复用**。比如容器中可以存整数，也可存小数，还可以存小猫小狗这样的对象……如果逻辑一样，一点小改动就要重新调整代码，自己也受不了…… 现实中也是如此，油罐车可以拉煤油，可以拉汽油，甚

至可以拉食用油……这当然可能会出现各种安全问题，所以需要很多的技术来确保安全。

第 18 章函数式编程

离婚的原因是什么？千千万万，但是最重要的一个是：心变了。bug 的原因是什么？千千万万，但是最重要的一个是：状态变了。什么叫状态变了呢？64 核的 CPU 中，有 1 个核修改了全局变量，但其它 63 个核不知道；有个文件被修改了，但其它进程不知道…… 状态改变，是 bug 的高发原因。如何让离婚率变为 0？方法是减少结婚，有感情的就同居得了。如何让 bug 减少？方法是减少命令式编程，增加函数式编程。于是，随着 CPU 核心的增多、分布式、协同式程序的增多，不改变程序状态的函数式编程“死灰复燃”了…… 本章从 Lisp 历史谈起，让大家清晰的了解什么叫函数式编程，为何 Java 的函数式编程不如 Lisp 那么优雅……

第 19 章枚举类型

在 Java 中，枚举已经强大到什么地步了呢？几乎所有类能做的事情，枚举都能做。枚举是一种特殊的类。枚举不止是简单的常量的集合，还可以拥有属性、方法和构造函数。还支持接口和实现。简单来说，**枚举是受限的类**。

第 20 章注解

注解是 Java 5 之后引入的。对司机而言，开车的时候，最重要的信息是路上的指示牌。比如看到“前方有急转弯”，你就会注意一点。同样，注解类似于这些路牌，当看到 @Override 的时候，编译器会去查一下父类中是否有这个方法，当看到 @Deprecated 的时候，会提示这个方法已经过时了…… 注解的本质是提供额外的元数据和指示，帮助编译器、工具和开发者更好地理解和处理代码。

第 21 章反射

据说，违反法律最刺激，要是违反了法律还不被制裁，就更是刺激了。比如古人有云：“妻不如...”，这句话你自己去搜搜吧。前面我们讲类的作用，又是封装，

又是隐藏。然后突然来一个机制，啥封装，啥隐藏？直接给你显微镜下看的明明白白的，并且能越过限制调用方法……是不是很刺激？反射就是这样的，使用反

射，咱们可以正大光明的走后门，所有的类定义的条条框框，都可以不用遵守。让你觉得像个特权阶级。

重要

这本书写了 21 章，而且肯定还有很多内容没有覆盖到，如果我说一些漂亮话，什么“书山有路勤为径，学海无涯苦作舟”，让大家再多学点什么的，一般人实际上内心是比较崩溃的。那学到什么时候是个头啊？！实际上，你根本不需要学这么多。我之所

以写这么多，是因为这是“写书”，其目的是“尽量覆盖”，而现实中做项目，其目的是“尽快做完下班”。如果你继续读我剩下的三本书，你会发现，每本书都只是用到这本

书里“部分内容”，比如做游戏，你就用不到反射，也用不到函数编程，因此会不会这些稀奇古怪的东西，对开发游戏完全没影响。

导读

这本书适合谁

《Java》是一本介绍 Java 编程的书。对 Java 编程感兴趣，并想使用 Java 语言开发软件，无论是给安卓手机开发软件，还是想用 Java 开发网站，甚至中间件的人，都可以从本书中获得最重要的基础知识。

本书的副标题是“有趣有料的 Java 编程”。如果你是一位对 Java 编程语言和计算机历史有强烈兴趣的爱好者，想必你一定也想了解编程语言和编程范式背后的故事，本书也能够充分的满足你的要求。副标题里的有趣和有料，对本书来说，意味着收集了很多有趣的编程故事，根据这些故事，再挖掘故事背后的历史真相。

事实上，依我所见，最好的学习方法是通过追溯编程语言的发展历史，了解语言从何产生，因何变化，一边在历史中考证语言的进化，一边在实践中学习编程的技巧，边考据，边学习。只有这样，才能更深入的理解语言进化的脉络，更真切的领会语言设计者的意图。

如果你认同以上的观点，那么这本书就适合你。

需要什么技术基础

从什么技术层面切入到 Java 编程领域？这是一个非常容易引起争论的话题。类似的问题还有学编程要求数学很好么？哪一门语言最适合初学者？要先学数据结构或者操作系统么？英语不好能学编程么？我认为学习编程和其它事情一样，比如学吉他、吹口琴、做蛋糕、追女孩…… 最大的障碍是：行动。

我们初中学过一篇清代文学家彭端淑写的古文，名字叫《为学》。在这篇文章中，讲了两个四川的和尚要去南海，一贫一富，富者说他准备了好几年，要买一艘船顺着长江去南海。穷和尚说他只有一个盛水的瓶一个盛饭的碗。过了一年，穷和尚从南海归来，而富和尚还在准备。

现在都已经是 AI 时代了，能搞明白个 20% 就开始动手吧。否则，光精神内耗，不停

的纠结要不要做，纠结十个月，你喜欢的女生都给别人生孩子了。先上车再补票，实在不行，再下车呗，反正也没买票，还多了一段乘车的体验。

善于下大棋，看二十步还没走一步的人，别说在 AI 时代了，就算在任何时代都活不过一集。比如，1807 发生了第二次哥本哈根战役，丹麦的海军被英国摧毁，哥本哈根这个城市还被炮击。丹麦投降，但是痛定思痛，觉得这样不行，要建立一支舰队再跟英国人搞一次，一血前耻！他们就种植了大量的橡树，无奈橡木生长缓慢，200 年后才长成。2007 年，丹麦的林业局通知丹麦海军，橡木终于长成了，可以去造船跟英国佬死磕了！只是，大家都不懂，橡木如何加工成航母呢?

对本书来说，如果你有编程的经验，相关的知识都储备的很好，那当然很好；如果没有编程的经验，也没有储备相关的知识，那也很好。千里之行，始于足下，本书对有无编程经验不做要求。人总要有第一次，理论重要，实战更重要，台上一分钟，台下十年功。

需要什么学习环境

小的学习环境是一台电脑，分硬件与软件两个环境。

硬件环境：一台可以运行的电脑是必不可少的，对硬件的要求不高，只要日常能上网的电脑，都可以用来学习 Java 编程。简单来说，能打主流游戏的电脑，肯定能学编程，能编程的电脑，可不一定能打游戏。但是，家长/女朋友/老婆并不知道这回事，你可以以学习的名义，配一块 4090 显卡。

软件环境：Java 是跨平台的，所以对电脑运行的操作系统没有要求，无论 Windows、Linux 还是苹果的 Mac OS 都可以。目前 Java 的稳定版是 21，也是本书使用的版本。

不过呢，当要体验新特征的时候，确实需要更新的版本，到时候再说。Java 现在进化很快——这也意味着曾经 Java 进化很慢——目前 Java 疯狂的从 Kotlin，Swift 等一众新生代语言中汲取力量，老黄瓜刷绿漆——装嫩。

大的环境就是教育了。大家可以想这样一个问题，中国教育与中国足球哪个更有希望呢?

我觉得，中国教育会更有希望，尤其是计算机专业。原因是，有大量的学生上大学后根本不上课，而是借助大学提供的巨大便利（主要是父母没法盯着了），通过网络自学，然后自谋职业。先用编程把自己的肚子填饱，如果还能学好英文，通过网络给外国的公司打工，如果干的好，就直接移民了，做世界公民不香么?

不要辜负这个时代啊！

中英对照的问题

计算机书有一个不能回避的问题是中英文对照，如何在中文和英文中保持平衡，对我来说是一大难题。在本书中，英文主要可以分为两大类，一类是涉及到的英文人名和英文公司名，别一类是涉及到的编程专业术语。

对第一类来说，我决定统一保持英文名，像 Java 之父 James Gosling，Sun 公司。对第二类来说，就比较不容易权衡了，像 Base Class 是翻译为“父类”好呢，还是翻译为“基类”好呢？还有 Derived Class 是翻译为“子类”好呢，还是翻译为“派生类”好呢？这都没有统一的叫法。

考虑到咱们国家的英语教育已经从小学开始了，能学编程的人，大部分都已经受过相当年限的英语教育，我决定在本书中，大部分类似面向对象，构造函数等不会产生歧义的用中文，少部分有歧义的，使用英语。像一些诸如 SDK（Software Development Kit）这样的简写，如果翻译为中文“软件开发工具包”，感觉比 SDK 更拗口，这时候也使用英文。

以讹传讹的大家心领神会就好，比如 NBA 有个球队叫费城 76 人队，其实人家跟 76 人没有任何关系，而是纪念 1776 年美国独立宣言。当然，这个说法也只是众多说法中的一个，但是我们的目的是知道怎么回事，而不是去死扣这些字眼。

编程的未来

现在 AI 横行，编程的未来是属于自然语言编程的。最佳的状态是，你跟 AI 对话，或者插个脑机接口，AI 就把你要做的事情做完了。根本不存在除 AI 之外的软件了，一切都是 AI。但是那一天，鬼知道什么时候能到来。

在 AI 完全搞定一切之前，编程语言与程序员尚有生存空间。即使在今天，AI 对程序员的帮助依旧巨大。我觉得，哪门编程语言容易重构，就应该学习哪门编程语言。因为，在 GitHub Copilot、OpenAI GPT……一众多模态大语言模型的帮助下，我们可以用“自然语言”来让 AI 帮我们“生成”代码，如果“生成的代码”不满意，就再让 AI 重新生成一次……

先学点，容易阅读，容易重构的编程语言，然后等着被自然语言取代吧。

声明

当我看到很多的明星、达人、大 V 以及显贵用蓝底白字的声明或者严正声明的时候，又是喊律师，又是删帖，又是封号的时候，他们实际上是要开始说谎了。语气越是咄咄逼人，内心越是脆弱不堪。

但是，没想到吧，我也要声明一下，用来甩锅。

能力不同，展示的水平就不同。以世界杯为例，一样的球，一样的球场，一样的草坪，对中国队而言，磕磕绊绊的打入 32 强，已经是莫大的胜利，但对巴西队而言，漂漂亮亮的夺冠才是他们的目标。

小时候，我有点喜欢足球，看的甲 A。真正让我爱上足球的是 1998 年的法国世界杯，那届世界杯，我认识了罗纳尔多，他展现出来的华丽技术让我惊叹，从此让我沉迷足球，他是我足球的 “初恋情人”。

在这本书里，我要向读者展示我的 Java 技术，对此我深感畏惧，诚惶诚恐。我尽最大的努力展示我最好的技术给各位。希望这本书让您喜欢上编程，喜欢上 Java。

虽然我希望能展示出巴西队 “华丽” 的桑巴技术，但是我也深知水平所限，可能展示成了中国队的 “磕磕绊绊”。如果看了这本书，您仍然不喜欢编程，那不是编程不好，也不是 Java 不好，是我展示得不好。

足球是世界第一大运动，并不因为中国队踢的稀烂而不成为第一大运动。Java 是房间里不可忽视的大象，并不因为有个叫栋哥的讲的稀烂而成为小众语言。

无论如何，愿您早日碰到自己编程的 “罗纳尔多”。

1 初识 Java

Contents

周公恐惧流言日，王莽谦恭下士时。
假使当年身便死，一生真伪有谁知！

说起这流言，哪个圣贤不挨骂？在编程界，哪个流行的语言不背负一身骂名？Java 是被人骂的最多的编程语言之一。毕竟，世界上只有两种编程语言，一种是饱受非议，另一种是无人问津。这一章介绍了 Java 的历史，从营销角度上来说，Java 是第一个被公司营销出来的编程语言。

Java 是一门蓝领语言。它不是博士论文材料，而是用于工作的语言。Java 对各行各业的程序员来说都似曾相识，因为我们更喜欢经过验证的东西。

——Java 之父 James Gosling

提示

原话是：Java is a blue collar language. It's not PhD thesis material but a language for a job. Java feels very familiar to many different programmers because we preferred tried-and-tested things。

这句话来自于 1997 年 James Gosling 写的一篇文章《The Feel of Java》。

1.1 Java 简介

Java 现在取得了巨大的成功，这往往被看作是必然的。Java 赶上互联网的浪潮，但人们往往会忽略这样一个事实：当时，没有多少人认为互联网能成为主流，更不要谈 Java 可以满足互联网编程的需求。

印度诗人泰戈尔曾经写过：“有时候爱情不是因为看到了才相信，而是因为相信才看得到。” 如果把其中的爱情，换成科技，我认为同样成立。

在 1995 年，Java 的创始人 James Gosling 写了两份文档，在文档中，他相信编程语言应该有的未来，于是创造出了 Java。本书，就从这两份文档开始吧。

1.1.1 Java 登场

1995 年 2 月，John Gage 受邀参加了一场会议，在会议上，他要发表一篇题为 “科技对教育的作用日益显著” 的演讲。在演讲的结尾，他想用一个演示软件来展示目前的高科技，为此，他已经花了一个多月来准备这个演示软件。

John Gage 演示的软件是用 Sun 公司尚未透露的编程语言写的，该语言的负责人是 James Gosling。因为事关重大，James Gosling 不放心让 John Gage 独自演示，于是，两人决定合作完成，演讲的部分由 John Gage 来做，展示的部分由 James Gosling 来做。

在演讲的结尾，James Gosling 出现在台上，他打开一个网页，上面有一个三维的红色分子模型，与当时人们常见的图片不同，这个分子模型可以随着 James Gosling 的鼠标放大缩小，旋转跳跃，这种新颖的操作方式，在 1995 年让在座的科技精英都赞叹不已。

演示所用的技术就是本书的主题：Java 编程语言。演示人 James Gosling 被人称为 Java 语言之父。

John Gage 简介

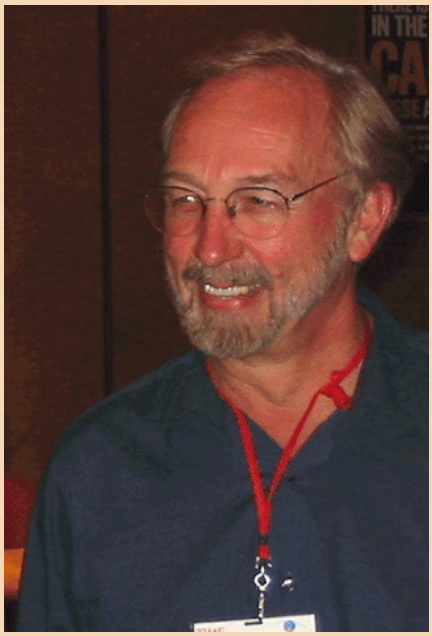

图 1.1: John Gage 在 2004 年 10 月的照片

出生于 1942 年，Sun 公司第 21 号员工，加州大学伯克利分校数学系肄业。在哈佛大学学习以后，开始对 Unix 感兴趣，后加入 Sun 公司，成为 Sun 公司的技术推广大使，营销界的精英人物。
关于 John Gage 的介绍，在 1996 年 wired 网站上有一篇名为《Power to the People》的文章专门介绍他，链接是：https://www.wired.com/1996/12/esgage

1.1.2 指导 Java 发展的两份文档

Java 之父 James Gosling 的故事想必大家已经耳熟能详，如果不太熟悉的话，推荐到国外计算机历史网站 Computer History Museum 上观看访谈 “James Gosling Oral History”，这个视频共分为为两部分，分别为 https://youtu.be/Yjq3hZXYp_k 和 https://www.youtu.be/LaJtYHvpa68。在这份长达近七小时的访谈中，James Gosling 谈了他本人和 Java 的方方面面。

1995 年 2 月，James Gosling 发布了一篇只有 9 页的文档来介绍 Java，这篇文档的名字叫《Java: an Overview》。这篇文档简明扼要的指出了当时计算机界存在的问题，并且给出了 Java 的解决方案。

随后在 1995 年 10 月，James Gosling 和 Henry McGilton 发布了 Java 第一版白皮书：《The Java Language Environment》。这篇 86 页的文档详细的介绍了 Java 的技术细节。

James Gosling 简介

图 1.2: 高司令的玉照

出生于 1955 年，加拿大人，Java 语言创始人。获美国卡内基梅隆大学计算机科学博士学位，1984 年加入 Sun 公司，在此，他研发了 Java 编程语言。

虽然日后 Java 发布过多份技术白皮书，但我认为这两份白皮书是 Java 所有白皮书中最重要的两份。在 Java 尚未被人所知的年代，James Gosling 写的这两份文档，如同明灯一样，指明了 Java 前进的方向。

现在 25 年过去了，在计算机领域，25 年让计算机软硬件有了长足的发展，但是这两份文档对 Java 学习仍然非常有指导意义。

这两份文档我读过很多次，坦白来讲，在技术领域，能指出问题的人不少，但是能做到抓住问题核心的同时又能提出有效解决方案的人则凤毛麟角。James Gosling 做到了，从 1995 年到今天，Java 的份额一路攀升到第一名，我想，在编程语言领域，没有比被程序员喜欢使用更好的赞美。

接下来，我们一起学习一下这两份文档。看看编程碰到了哪些难题，Java 又是如何解决的。

1.1.3 Java 面对的问题以及解决方案

起初，James Gosling 并没打算开发一门新的编程语言。

他参与了一个名为 Green 的项目，这个项目的目标是能让 “包括 VCR、电话、游戏机、手机、洗碗机等在内的” 消费电子产品能够 “透明地交互操作”。由于设备类型太多，

Gosling 的团队碰到了一个棘手的问题，用 C++ 语言支持如此众多的设备，起初可以通过修改 C++ 编译器来解决，随着设备越来越多，最后成了一个 “灾难”。于是，James Gosling 意识到，是时候做一个新的编程语言了。

和其它编程语言一样，Java 的发展史也是一个找到痛点并解决的历史。在软件发展史上，这种情况屡见不鲜，甚至有了专属自己词汇的描述：“dogfooding”。

> 什么叫 Dogfooding?
>
> Dogfooding 这个词语来自于 “Eating your own dog food”，可以翻译为 “吃自己的狗粮”。
>
> 在 IT 业界这句俚语可能最早是于 1988 年开始使用的。当时微软公司的高级主管保罗 · 马瑞兹曾写过一封题为 “Eating our own Dogfood”（吃我们自家的狗粮）的邮件，在邮件中他向微软局域网管理工具项目的测试主管布莱恩 · 瓦伦蒂尼提出 “提高内部使用自家产品比重” 的挑战。
>
> 从此以后，这一俚语就传播开来了。

在上世纪 90 年代，大部分程序员只能在 C 和 C++ 语言中选择。每家厂商都有自己独特的设备，有互不兼容的操作系统，再加上互联网的兴起，对设备的要求从单机进化到另一个维度：联网。

不止 C 和 C++ 语言，可供选择的语言如 Eiffel，SmallTalk，Objective C 都没有为互联网的爆发提供足够的支持。在非网络化的情况下，各种设备和程序相互独立。但是一旦接触到互联网，这些相互独立的设备就要面对未知的环境，原本无关紧要的程序漏洞，在互联网环境下可能会是一个灾难。

所有的一切，让编程越来越困难。

如何能在减轻程序员负担的同时，还能提高开发软件的质量，是 Java 自始至终一直在解决的问题。那 Java 是如何做到的呢?

1.1.4 采用熟悉的语法和熟悉的思想

Java 的设计者认为，要避免程序员花大量的时间来学习一门风格迥异的语言，最直接的方法莫过于借鉴 C 和 C++ 的语法，毕竟这是当时最流行的语言。这个策略被证明是非常有效的，大量的 C++ 程序员甚至不需要重新学习，就能用自己 C++ 的经验写 Java 程序。

Sun 公司的创始人 Bill Joy 对 Java 的支持功不可没。

Bill Joy 简介

图 1.3: Bill Joy 的照片

出生于 1954 年，加州大学伯克利分校硕士，在校期间和朋友一起创办了 Sun 公司，是公司的联合创始人。

在校期间，他是 BSD 系统的主要设计者，同时还是 vi 编辑器（本书完全使用该编辑器完成，所以，**我经常推荐别人用 Vim 编辑器**，一是为了装逼，二确实挺好用的:），他还是 C Shell 的作者。

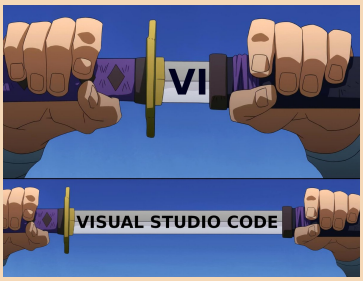

图 1.4: 那么，我现在推荐用哪个编辑器?

他是 Java 创始人 James Gosling 的上司兼好友，Java 项目几乎被取消，都是 Bill Joy 力排众议，持续支持该项目，最终获得成功。

Bill Joy 的故事可以参考《Unix Review》杂志在 1984 年对他的采访，名字是《Interview with Bill Joy》，在访谈中，有关于 vi 编辑器，Unix 等有趣的故事。在线网址是: https://web.archive.org/web/20120210184000/http://web.cecs.pdx.edu/~kirkenda/joy84.html

在语法上，Java 不仅继承了 C++，还简化了 C++，舍弃了一些诸如多重继承，操作符重载比较复杂的特性。同时为了简化编程，Java 还增加了诸如内存管理的功能，Java 语言可以自动的对内存进行垃圾回收。Michael Feldman 曾经这样评价 Java：“Java 从很多方面来说，就是简化版的 C++。”

> 提示
>
> 他评价的原文是：Java is, in many ways，C++−−。

如果你是有一定经验的程序员，同时写过 Java 与 C/C++，就会意识到，其实 Java 仅仅是在语法的层面上与 C/C++ 相似，实质上，Java 的实现更接近 Lisp 和 Smalltalk 那样的动态语言。

Java 集百家之长成一家之言，不仅借鉴了 C 和 C++ 语言，在文档中，Java 从不隐瞒这一点，宣称自己的独创性，反而对其它语言的优秀特性大加赞扬，会直言从哪种语言中获得灵感，比如在动态性方面，就从 Objective-C 借鉴了很多想法。在本书中，我会尽量对这些特征一一考证。

本书会有大量的章节讨论 Java 的语法和面向对象的问题，包括第 5 章和第 6 章的基本操作符和基本语法，第 3 章、第 9 章、第 10 章、第 11 章等章节都是讨论面向对象的问题。

1.1.4.1 改善可移植性

取得成功的编程语言，一定要兼顾这三个方面：运行速度、可移植性与安全性。同时满足运行速度与可移植性的语言就很少，更不要说还要兼顾安全性了。一般来说，运行速度快的语言，可移植性不高，可移植性高的语言，又难以做到兼容性。

用 C 语言写的 Unix 操作系统也被广泛的宣传具有强大的可移植性，但是这种 “可移植性” 是建立在对每一种机器要修改的基础上，比如有的机器 int 是 16 位的，有的机器是 32 位的，还是要根据不同的 CPU 进行代码的修改，然后才能做到可移植性。

> Unix 痛恨者手册
>
> 在网上有一本神书叫《Unix 痛恨者手册》，是一群 Unix 的讨厌者写的书。
>
> 书写完以后，请 C 语言与 Unix 系统的开发者丹尼斯 · 里奇写序言，丹尼斯 · 里奇也不客气，痛骂了本书，这本书把这个痛骂自己的话原封不动的当了序言。
>
> 网上有本书的中文翻译版，如果你找不到，可以到我的个人网站上去下载: https://liuyandong.com 或者到https://xueban.app这个社区下载。

Java 的出现，真正实现了可移植性。在 Java 里，所有的基本数据类型的大小都是规定好的，不会随着 CPU 的变化而变化。比如说，在 Java 中，所有的 int 意味 32 位，所有的 float 都是实现的 IEEE 754，这极大的方便了程序员的工作。

但是这并不是没有代价的，C 语言为了可移植性牺牲了语言的功能，Java 为了可移植性牺牲了部分运行速度。

在本书的第 4 章，我们会来详细讨论可移植性与基本数据类型这个问题。

1.1.5 采用解释执行

Java 语言是一种解释型语言，只要将 Java 源程序编译成字节码，这些字节码就可以直接在 Java 虚拟机上运行。相比于编译，链接是一个要轻量级的过程，因此开发的过程要更快一些。你可以想象一下现在为什么大家都喜欢用 “热更新” 的方式来开发 App，因为更轻量级，方便。

> **提示**
>
> 因为这份文件是历史文档，Java 1.0 的时候，只提供了一个纯解释的运行环境，现在已经有所改变。
>
> 等一会我们再来讨论 Java 到底是编译执行还是解释执行这件事。

毕竟 25 年过去了，Java 在这方面有了长足的进步，从某种意义上来说，现在的 Java 已经不完全是 James Gosling 在 1995 年的规划。尤其在 Java 是一个纯解释语言这一点上，发生了重大的变化。现在的 Java 早已经是兼具编译型和解释型语言的特点。

关于这方面内容，我会在第二章，和大家详细的研究 Java 虚拟机，Java 编译器各自的作用。在 1995 年的文档里，James Gosling 给出了一个数据，在一台 SS10 电脑上，用解释器运行 Java，每秒钟可以调用 30 万次函数，这个数据和 C/C++ 写的软件没有明显的差距。Java 虚拟机进化了多年，现在人们已经对 Java 的性能没有太多质疑，有很多的数据支持显示，Java 是运行最快的语言之一。

> **讽刺 Java 慢的段子**
>
> Java，C，C++，汇编……在一艘船上，船漏水了，为了保证其它人的安全，决定扔两个人下去。规则是讲笑话，只要有一个人不笑，就把讲笑话的人扔下去。
>
> 汇编是讲笑话的高手，他讲了一个笑话，把其它人都笑弯了腰，只有 Java 没笑，按规则，汇编被扔了下去。第二个轮到 C 语言讲了，C 语言还没开口，Java 就笑弯了腰，

> 众人不解，问 Java 笑什么？
>
> Java 回答：“刚才汇编的笑话太好笑了！”

目前来说，Java 的运行速度已经不再是问题，著名的游戏公司 ID Software 已经开源了其第一人称射击游戏 Quake2 的源代码，随后，有人将此引擎移植为 Java 项目，这个项目名为 Jake2，完整的代码可以在 github 网站上找到。具体的链接如下: https://github.com/demoth/jake2 。如果 Java 的性能足够满足第一人称射击游戏的要求，那么肯定可以满足商业程序的要求。

Java 代码的执行速度是不稳定的，高度依赖于运行平台与 Java 虚拟机。即使是同样一份代码，不做任何改进，在比较新的 Java 虚拟机上基本上运行速度都会有所提升。

最初的 Java 采用解释执行的原因是 Java 非常的务实，对性能的态度一贯是只要能提高开发效率，就可以牺牲原始性能。也正是因为这个原因，只有到了近些年，诸如 HotSpot 之类 Java 虚拟机日益成熟后，高性能计算领域才开始采用 Java 进行开发。

在设计语言和平台的时候，经常对需要的功能与运行效率之间进行平衡。如果要更“接近机器”，那么就有可能“降低开发效率”。C++ 的作者一直推崇他的“零成本原则”，这条原则的内容如下：C++ 遵从零成本原则：**不需要为用不到的功能付出代价**。再进一步，你要用的东西，性能已经达到极致了，就算再优化也不会有更好的性能了。

这个原则听起来是非常好的，但是对绝大多数人没什么用。因为对绝大多数来来说，自己没法掌控软件运行所依赖的底层机制，都是要借用别人写的操作系统来运行自己的代码。如果还要追求零成本的运行代价，开发人员就要付出巨大的认知代价，要搞清楚计算机是如何运行的，并不容易。

像 C/C++ 所采用的跨平台方法是预先编译（Ahead of Time, AOT），对不同的平台要分别编译，势必要详细了解那种平台的机器代码。这个原则是为像 C++ 作者这种级别的绝顶高手量身定做的，这里的零成本隐藏了太多东西。高手能实现的零成本解决方案，比如人家很容易实现一个解释器，对其他人的成本可能就太高了。

Java 不认可零成本原则，Java 做出的取舍是：借鉴 C/C++ 的预先编译，同时又不放弃解释执行。通过这种方法，可以降低写软件的难度。

1.2 Java 版本进化史

我写书的时候，写了很多这方面的内容，但是现在想起来，没什么用处。直接跳过吧。

有一点需要注意，Java 历史悠久，同时也意味着包袱沉重。Java 是极其成功的语言，打入了大量关系国计民生的行业，比如能源、医疗、金融…… 这些行业是不可能乱更新的，因此你工作之后，会出现这样的情况，这个软件可能跟你是同龄，或者比你小不了几岁。

一般来说，如果它是 Java 5 写的，你就不能够更新成 Java 21。你要知道，在一个能源部门，没有人可以承担这种责任，也没有人想承担这种责任。你大概只能按照那时候的 Java 版本来写软件，Java 5，Java 8……

旧版本不能用到新功能，如果你“不幸”工作中用到的是一个古老的 Java 版本，到时候再自己查书查资料吧。

1.3 小朋友，你是不是有很多问号?

1.3.1 为什么 James Golsling 被称为终身仁慈独裁者?

Java 的创始人 James Gosling 被称之为 BDFL。

BDFL 的意思是终身仁独裁者 BDFL（Benevolent Dictator For Life）是极少数软件开发者能拥有的头衔，一般是某个语言、某个项目的创始人。当软件社区出现争议并且无法和平解决的时候，要有一个最终的话事人来解决纷争。其它耳熟能详的 BDFL 有 Linux 的开发者 Linus、Python 的开发者吉多 · 范罗苏姆、Perl 的作者 Larry Wall、LLVM 的设计者 Chris Lattner 等等。

能够当 BDFL 的人，要有很强的领袖气质，能以德服人。而且，在编程领域中，独裁者是会“体面”的离开的。

不止 Java，像 Python，JavaScript 语言，都经历了开始主要由一个人主导，后来一个委员会主导，到最后，语言的创始人离开。2018 年 7 月 12 日，Python 创始人 Guido van Rossum 发邮件决定离开 Python 决策层，不再领导 Python 语言的开发。有兴趣的读者，可以找来那封信读一下。

最后，终身仁慈独裁者，跟传统政治上的独裁者是两码事，举目全球，传统政治行业上的独裁者几乎没有了。

1.3.2 Java 的名字是谁起的?

以目前能找到的资料，Java 项目最开始的时候，名字并不是 Java，而是 Oak。之所以叫 Oak，是因为在 James Gosling 的办公室外有一棵橡树。当时有一家已经存在的公司叫 Oak Technologies，只能改名。

> 为什么要叫 Java，是找算命先生算过么?
>
> Javaworld 曾经写了一篇文章叫《So why did they decide to call it Java?》，网址：https://www.javaworld.com/article/2077265/so-why-did-they-decide-to-call-it-java-.html
> 原来起名的时候有几个候选，分别是 Silk、Lyric、Pepper、Java。当时的产品经理 Kim Polese 提出了这个名字。

1.3.3 James Gosling 如何回答相比于 C 语言，Java 不够自由?

James Gosling 对此的解释是：“对 Java 来说，规矩有很多，一旦你适应了规矩，那么 Java 将是一种自由的语言。”他还以飞机为例打了个比方，在螺旋桨时代，飞机非常的自由，飞行员可以打开窗户，呼吸新鲜的空气，可以通过肉眼来观察方向。到了喷气式飞机时代，飞机的窗户是不能打开的，如果你打开，在 3 马赫的速度下，飞行员的脑袋将会被吹走。

最后，他总结说：“如果你想进一步让自己自由，就要放弃一些看起来是自由的东西。”

我不知道为什么，看到 James Golsing 讲这句话的时候，我脑海中想到的是阿尔贝 · 加缪写的《西西弗神话》，在一个看似没有意义的世界中，我们应该如何找到意义和应对编程、甚至生命的荒诞感。在加缪的哲学中，自由不仅仅是选择的能力，而是承担选择后果的责任。

这一点，倒是与 James Gosling 有相通之处。

1.3.4 Java 的现状怎么样?

无论是从找工作还是从学习编程的角度，Java 都是一门不可忽视的语言。

大家只要打开搜索引擎或者 GitHub 网站，查找一下语言的排名，或者在找工作的网站上看一下，就知道 Java 始终是最热门语言之一。写这本书的时候，我在找工作的网站 dice.com 上使用关键字 Java 时行了搜索，有 11,153 个工作，用 Python 当关键字，有 6,395 个工作，用 JavaScript，则有 8,634 个工作。这些结果应该相对直观的反应了市场需求。如果你的目标是学好英语，找外国的远程工作，Java 还是不错的。

从学术和技术角度来看，现在越来越多语言都是基于 Java 实现，运行在 Java 虚拟机之上。比如 Groovy、JRuby、Scala、Jython、Clojure 这些语言，都是如此，只要能运行于 Java 虚拟机，就是站在巨人的肩膀上，有了与生俱来的跨平台特征。从这个角度来看，学会 Java，能更深入的了解这些语言。

业界有一句名言：“没有人因为选择 Java 而被解雇。”这句话的潜台词是：Java 的背后有大公司的支持，这意味着可以得到某种保证，保证 Java 能得到积极维护和开发，具有繁荣的生态系统和社区，并已经 Java 已经在大型业务环境经受了考验。企业就喜欢这样的技术，可以减少技术选型的风险。这也是 Java 长期排名前茅的原因。

> **!** 工字不出头，出头入黄土
>
> 我还是想扯一点闲篇。我老家有句叫“工字不出头，出头入黄土”。这里的“工”有很多的解释，可能是“工人”，可能是“工资”，可能是“工程师”，可能是“打工”……因为

你在学习 Java，可能梦想是当个“软件工程师”，好吧，这也算“工”的一种。

亲爱的读者，靠这个“工”，是很难出头的，包括“打工”的“软件工程师”。本书的作者——也就是我，寒窗二十年，打工十余载，一事无成——已经证明了这一点，证明了我老家人的智慧，多么痛的领悟啊！不要试图靠“工”改命。

那靠什么？我还不知道，如果我知道了，我会在这里更新的。也希望你能告诉我，我的电子邮件是 liuyandong@gmail.com。感恩贵人的不吝赐教。

1.4 程序员故事

我很喜欢写小说，总是臭显摆一下。在一本编程书里写小说，纯粹是浪费纸张。但是在网上，不差这几 K 的流量，所以，请允许我，把以前写的小说放在这里。我尽量在每一章后面，写一则程序员的小故事，有些是真的，有些是杜撰的。

如果你不想看，直接略过就好。

2 第一行代码

Contents

编程难，纹身大哥胆寒！
枪林弹雨何所惧，更喜狱友菊花残。
垂死病中惊坐起，程序崩的没道理。

为什么会有这首打油诗呢? 因为我看到一个新闻，讲的是地球的另一边，美国，有 https://thelastmile.org 这样一个项目，如果你犯了事，被关进了监狱。监狱里就会培训你: 编程！

听说，这大大降低了犯罪率，因为很多惯犯出狱之后，一想到再次入狱，要去学编程，就放弃了作案的想法，那可太痛苦了，宁可死刑也不愿意去监狱学编程！

本章的主要目的是把开发环境给跑起来，用 Hello World 肯定学不到什么东西，但是至少能检测开发环境有没有跑起来。

> **Hello World 的由来**
>
> 1974 年，美国贝尔实验室，一位名叫 Brian Kernighan 的研究员，他写了一份名为《Programming in C – A Tutorial》的文档，在这个文档的第二段，他用 C 语言写了一个经典的程序，程序是输出 Hello World 这两个单词。
>
> 后来这个研究员又写了一本风靡全球的书叫《The C Programming Language》，在这本书里，他再次用了上面提到的程序，从此，几乎每一个编程语言的第一个程序都是输出 hello world 这两个单词。
>
> 写这个程序的目的是测试一下开发环境配置的是否正确。如果能输出这两个单词，至少说明“大概率”可用。本章的目的也是让开发环境跑起来。

2.1 “传统”Java 版的 Hello World

其源代码如下：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

这一章，我们要做的事情是搞明白：**用什么工具，把上面这段代码输入，输入之后，用什么工具把这些代码整成能运行的“软件”**。

2.1.1 用什么工具来输入代码?

简单来说，用 VSCode 或者 IntelliJ IDEA，前者是一个编辑器（虽说是编辑器，但是比十年前的 IDE，比如 Eclipse，NetBeans 的功能强太多了），后者是一个 IDE。说实在的，不管这两个你用哪个，都已经是非常强大了，强大到无论你用哪个，都可以通过菜单把代码运行起来。

Eclipse VS NetBeans 是当年争锋的两大 IDE

Eclipse 最初是由 IBM 开发的内部工具，在 2001 年发布为开源项目。Eclipse 的设计师是 Erich Gamma，同时他也是 JUnit 的作者。Eclipse 的精髓是插件，但是 Eclipse 的插件可以进入主进程，一些“水平很一般”的程序员，可能就把主进程给写坏了。

后来 VS Code 就吸取了经验，等等，为什么 VS Code 能吸取经验呢？因为 VS Code 是 Erich Gamma 跳槽到微软做出来的。

至于 NetBeans，也是一个 IDE。哎，本书的作者——就是我——曾经在 Sun 公司里去高校以及各种技术大会，比如 Intel 开发者大会，推广过 Solaris 和 NetBeans，推广真的好难，背着宣传单，背着光盘，还得送人家巧克力。人家还不一定用，只是把光盘上送的巧克力吃了，出门就把光盘扔了。

我就不多介绍什么是 VSCode 和 IntelliJ IDEA 了，问问 ChatGPT 吧。如果开发 Java 的话，项目一旦变大，IntelliJ IDEA 是更好的选择，这家捷克开发的软件，有收费版，也有免费的社区版，用免费的就足够了。

只需要点一下，代码就跑起来了，就跟自动档汽车一样，“傻瓜”都会开。

无论你选择哪个，我都建议你安装一些 AI 插件，这些 AI 插件可以自动帮你补全所有的代码。插件实在太多了，也没人给我广告费，所以我就不推荐了。我用的是 GitHub 的 Copilot，但是这个每个月要收 10 美元。如果没人给你这个钱，就用免费的吧，功能大差不差，尤其是对初学者，都是用最简单的功能，没差别。

推荐用哪个工具?

没有什么好推荐的，我做事的原则是：大事看原则，小事讲风格。

用哪个工具，缩进用空格还是用 tab，显然都是小事，你有自己的风格就好。又不是什么家国大事，婚姻大事，如果在这些大事上有冲突，赶紧跑路，最好打车跑路。

2.1.2 输入的时候需要注意什么?

编程语言是“语言”的一种，而且是“最死板”的那一类。错一点，也不给你运行。所以，对初学者，有以下需要注意的地方：

2.1.2.1 源代码的文件名要与类名严格一致

如果你是初学者，可能会好奇，什么叫类名？没关系，类名就是上面代码中 class 后面跟着的 HelloWorld。java 的源文件后缀名为.java。所以，源代码的文件名为 HelloWorld.java。

2.1.2.2 要注意全角与半角符号

在源代码中有分号、逗号、花括号……这些都应该用英文的半角字符，而不能用中文的全角字符。在英文中，这些符号都是半角——在排版时占据一个字符的宽度。而在中文中，这些符号都是全角——在排版时占据两个字符的宽度。

我当年初学的时候，在这上面可吃了大亏了。

2.1.2.3 可以用 AI 补全

要是搁在以前，我都是建议别人手打一遍代码，受受苦，但是现在，真是用不着了。AI 目前这么强大，现在你不让他干活，到时候它强大到足够奴役人类的时候，可不一定会让你清闲哦。早享受总是对的。

2.1.3 如何让源文件跑起来?

用的工具是 JDK，全称为 Java Development Kit（JDK），这是 Sun 针对 Java 开发人员发布的免费软件开发工具包（SDK，Software development kit）。自从 Java 推出以来，JDK 已经成为使用最广泛的 Java SDK，即使后来 Sun 公司被 Oracle 收购了，也叫 JDK，只是叫 Oracle JDK 了。

至于如何安装 JDK，我就不废话了，请打开 ChatGPT 这类大语言模型，输入类似“如何在 Windows 上配置 Java 开发环境？”大语言模型处理这种事情，特别拿手。

如果你使用的是 IntelliJ IDEA，可能你连装 JDK 的机会都没有，IntelliJ IDEA 当检测到你的电脑没有 JDK 的时候，会使用自己的 JetBrains Runtime，这是一个基于 OpenJDK 的修改版本，作为运行环境。

无论是借助大 AI 来帮助你完成的开发环境配置，还是 IntelliJ IDEA 自动安装的，最终，都会安装几个软件，最重要的有两个，一个叫 javac，另一个叫 java。

javac 的是 java 语言的编译器，其作用是将 java 源代码（就是前面那个 HelloWorld.java）编译成字节码，如果你用命令行的话，命令如下：

```
javac HelloWorld.java
```

这条命令执行之后，会生成字节码文件（你会发现执行完以上的命令之后，在存放 HelloWorld.java 的目录下，多了一个叫 HelloWorld.class 的文件）。然后，再使用 java 来运

行刚刚生成的字节码，命令如下：

```
java HelloWorld
```

这里需要注意的是，不需要 HelloWorld.class 这里的.class 后缀。

但是现在没人还这样手工来处理了，都是使用 VS Code 或者 IntelliJ IDEA，这些工具已经足够智能，完全自动的处理 javac 与 java 的执行过程。点个按钮，直接出结果。这当然简化了工作，但是也会让人搞不清楚到底发生了什么事。

各有利弊吧。希望你在 VS Code 上，或者 IntelliJ IDEA 上看到了 “Hello World” 这两个单词。如果没有，恭喜你，你碰到 bug 了。看下一小节。

2.1.4 碰到 bug 了怎么办?

如果你认为确实没有任何问题，但是就是不运行，哥们，第一件事情：重启（先重启开发环境，再重启电脑）。可能会有其它同事说你，你 TMD 就是会重启，这时候，如果你觉得有必要跟他争论，就以下面的理论为基础进行人身攻击。

图灵奖得主 Jim Gray 写的论文，名字叫《Why Do Computers Stop and What Can Be Done About It》，我把这篇论文放在我的社区 xueban.app 里。在这篇论文里，Jim Gray 认为，bug 为两种，一种是玻尔 bug，另一种是海森堡 bug。

按照 Jim Gray 的分类，玻尔 bug 是能通过测试，容易复现，也比较容易解决的 bug。海森堡 bug 是不确定的，不容易复现，不容易解决的 bug。正如两位科学家所研究的理论一样，一个如原子结构般确定，一个如电子行为般不确定。

如果重启能解决，那我们碰到的就是玻尔 bug。

如果重启还没有解决问题，一般来说，把报告的错误信息放到搜索引擎里，是个好办法。初学者，碰不到什么新鲜 bug，99.9999% 的情况下，咱们碰到的 bug 都是别人碰到，并且解决过的。

但是也不否认你确实碰到了别人没有碰到的 bug，虽然机率确实很低，而且是用 HelloWorld 这种代码碰到的，概率就更低了。

Jim Gray 简介

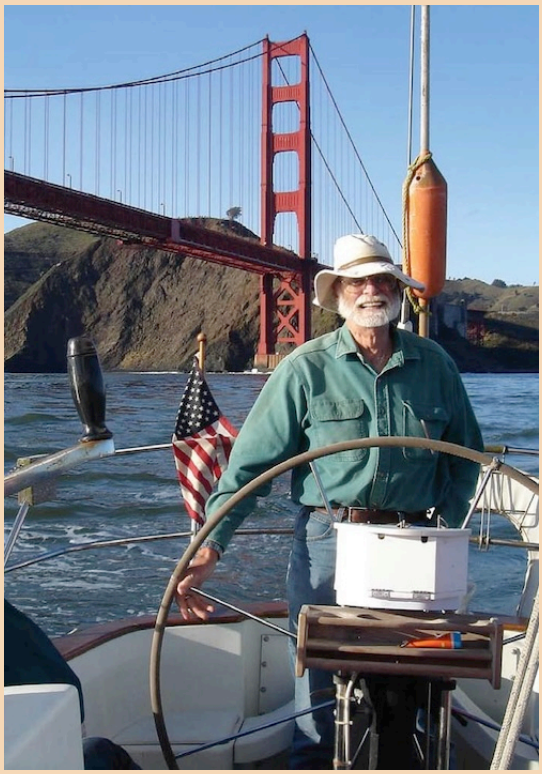

图 2.1: Jim Gray 的照片

Jim Gray 是第三位因推动数据库技术而获得图灵奖的人，在计算机方面有非常多贡献。他爱好冒险，在 2007 年 1 月 28 日，他驾驶着自己 40 英尺的船，消失在茫茫大海，至今杳无音讯。5 年后的 2012 年 1 月 28 日，Jim Gray 在法律上被宣告去世。

我平时还有个爱好是录 podcast，名字叫“软件那些事儿”，我录了快 500 期电台了。有一个系列叫图灵奖得主介绍，就介绍每一年图灵奖得主的信息，如果你有兴趣，可以搜来听听，或者到liuyandong.com上看文字版。

不过，凡事不能太绝对，比如，在 Java 的历史上，就有一个至今都没修复的 bug，这个 bug 是初学者就能碰到的。有兴趣的读者可以到 Java 的 bug 系统上去看看，网址是：https://bugs.java.com/bugdatabase/index.jsp 输入这个 bug 的编号：4252539.

按照 Java 语言的规则，main 函数必须被声明为 public，但是软件难免会出现 bug。在 Java 1.4 之前的一些版本里，public 不被声明为 public 也可以运行。这个 bug 从来没有被修复过，主要原因是如果修复了，可能会导致其它不可预知的结果。不少 bug 就是会被置之不理的。

苹果手机里也有“没人理”的 bug

如果大家用 iOS 1582 calendar 作为关键字来搜索，会找到这个 bug 的详细介绍。

简单来说，在 1582 年，人类的日历从儒略历换成了格里历。这就导致在 1582 年的 2 月份出现了 2 月 31 日。iOS 系统没有恰当的处理这个 bug，以后不知道会不会处理，但是谁会在意 1582 年的日历呢?

同样，**bug 足够出名，还会成为 feature**。（你足够出名，写的错别字就会成为通假字，人是一种贱嗖嗖的生物，欺软怕硬。鲁迅写就是通假字，你写就是错别字被都是扣 1 分）。

在软件史上，有很多类似的情况，bug 成了 feature，将错就错。比如 Unix 上的 creat，本来应该是 create，但是少写了一个 e，后来所有的 Unix 变种都将错就错的少写了一个 e。后来有人问到 Unix 的作者 Ken Thompson 如果你要重要设计 Unix，你会做哪些改变，他回答说把 creat 写成 create。这个故事来源：https://en.wikiquote.org/wiki/Ken_Thompson

2.2 八仙过海的 Java 虚拟机

这段是八卦，如果你没时间，不用看。如果用来八卦的话，还是稍微有点用处的。或者只需要看看最后我的感慨就可以了。

比如，现在大语言模型是热点，那么，所有有能力的公司都要搞自己的大模型出来。行不行是一回事，有没有是另一回事。如果不行，在牌桌上顶多是个送财童子；如果没有，那可就连牌桌都上不去了。

同样的道理，Java 在当年也是热点，而且还开源。只要你敢开源，洒家可就敢自主创新了。Java 的标准大家都要“尽量遵守”，也有故意掺沙子，甩石头，挖墙角的，比如微软当年搞的 J++ 语言，就是想把 Java 搞黄。但是 Java 虚拟机的强弱，就要各显神通了。

微软的 J++ 语言与 C# 语言

以前有个叫 Borland 的公司，其灵魂人物叫 Anders Hejlsberg，跟很多大佬一样，是个辍学生。Borland 公司跟微软有场官司，控告微软挖走了公司 30 多名重要雇员。这些雇员中，就有 Anders。最后 Borland 赢了官司，但是输了战争，从此销声匿迹。

Anders 去了微软以后，操刀负责 J++ 项目，这个项目从名字上看就是对标 Java。而且是 Java 的变种，其语法、关键字与 Java 完全相同，讲究一个微创新。Anders 的功力是不容小觑的，在他的操盘下，外加微软的无限资金与无限人才，J++ 很快就做到了当时 Java 所能做到的一切，并且在很多方面都超越了 Sun 的 Java。完爆这个词不好随便使用，但是说一句青出于蓝而胜于蓝，是不夸张的。

> 从技术上来说，我觉得微软胜；但是从道义上来讲，微软输。因为当时 Sun 给微软的协议，要求微软不能魔改 Java，要遵守标准。而且，微软那时没打算把这些技术用在 Windows 之外的平台上，那时跟 Linux 是水火不容，更不要说给 Linux 提供支持了。于是，法庭见，微软输了。
>
> 于是 J++ 项目停止，C# 项目出生。操刀人依旧是 Anders。于是纷争，就成了 Java 与 C# 的竞争，这么多年过去了，若论技术，我仍然觉得 C# 要比 Java 高明一点，但是微软多年来对开源社区的态度，以及经常跟合作伙伴上法庭，所以，C# 就是没法压 Java 一头。
>
> 后来，又有一家公司试图把 Java 给分裂掉，这家公司就是 Google，只是 Java 的东家换成了更善于打官司的 Oracle，那就是另一个故事了。

我们写的代码最终还是要跑在 Java 虚拟机上。现在业界有很多种 Java 虚拟机，在维基百科的页面上，活跃的 JVM 还有 9 种，不活跃的 JVM 有 16 种，我相信还有更多没有统计在这个页面上的 JVM。JVM 和 Java 语言是双子星，他们相伴相生，一损俱损，一荣俱荣。如果你对故事感兴趣，可以读一下 Java 虚拟机的故事。

2.2.1 Sun 的虚拟机

1996 年，Sun 发布 Java 语言的同时，还发布了世界上第一个 Java 虚拟机。这个虚拟机之所以重要，是因为是世界上第一个 JVM。缺点也很明显，这个 JVM 的执行效率非常差。

随后，Sun 公司开始对 Java 虚拟机进行改进。改进的成果是一款只可以运行在 Solaris 上的 JVM，名叫 Exact VM。如果大家用 Exact VM 在 Oracle 的官网上搜索，仍然可以找到一些信息。大部分信息都是介绍 Exact VM 和 HotSpot 在编译时如何设置参数。

但是 Sun 没有再继续推广这款名为 Exact VM 的虚拟机，原因是一个更为先进的虚拟机出现了，这个虚拟机也是目前最为流行的 Java 虚拟机之一，名字就叫 HotSpot。

HotSpot 并不是 Sun 公司做的，而是一家叫 Longview Technologies 的产品，这家公司的创始人，一个叫 Urs Hölzle，一个叫 Lars Bak，两人从 Sun 离职后创业。为了重新获得这两个员工和他们的产品，Sun 收购了这家公司。

该产品原本是针对 SmallTalk 语言做的虚拟机，被收购以后重新设计，转而支持 Java 语言，跟随 Java 1.2 的同时发布。

Java 的 1.2 版本，有 3 个 JVM，一个是 Sun 公司第一版的虚拟机，一个是 Solaris 上的 Exact VM，还有一个就是 HotSpot。默认的 JVM 是最慢的那一个，直到 1.3 版以后，才将默认的虚拟机换成 HotSpot。

在早期，只有 Sun 公司有 JVM，其它公司没有“版权”来染指 Java。直到 1998 年，Sun 公司成立了 JCP（Java Community Process）组织，这个组织希望让越来越多的公司参与进来，大家都分一杯羹。当然了，Sun 公司肯定是想做分羹的人，大家做 Java 可以，但是要通过 TCK（Technology Compatibility Kit），翻译成中文叫技术兼容性测试。

在 TCK 的约束下，不少公司通过了 TCK，也号称自己的虚拟机叫 JVM。在这个背景下，有很多公司推出了自己的 Java 虚拟机。最著名的厂商有 IBM、BEA、Microsoft 和 Apache。

> **IBM 的 J9 虚拟机**
>
> IBM 官网上介绍，这款 JVM 最早是由 IBM Ottawa 实验室一个 SmallTalk 的虚拟机扩展来的。
>
> 那时候，这个虚拟机有一个 bug 是因为 8k 值定义错误引起，工程师们花了很长时间终于发现并解决了这个错误，此后这个版本的虚拟机就被称为 K8 了。
>
> 后来出现的支持 Java 这个版本的虚拟机就被称为 J9 了。

与 Sun 公司收购 HotSpot 类似，IBM 的 J9 也是收购的。1996 年，IBM 收购了一家叫 OTI（Object Technology International）的公司，该公司有虚拟机产品。更巧合的是，这个产品最初也是为 SmallTalk 语言设计的虚拟机，后来在 Java 流行以后，才改为支持 Java 虚拟机。

2017 年，J9 变成了 IBM 主导的 Eclipse 组织的一个项目，名字已经改成了 Eclipse OpenJ9。现在建立在 Eclipse 开放运行时项目（OMR）之上，IBM 的专有项目大部分基于此，它完全兼容 Java 认证。

2.2.2 BEA 的 JRockit

BEA 公司目前已经被 Oracle 收购，所以，Oracle 拥有三个最主要的虚拟机中的两个。

BEA 公司是著名的 Java 中间件公司，曾经是 IBM 公司最重要的竞争对手，产品是与 IBM WebSphere 竞争的 WebLogic，IBM 有自己的 Java 虚拟机，BEA 也想有自己的 Java 虚拟机。

收购是最省时间的方式，BEA 就收购了 Appeal Virtual Machines 公司，这个公司的产品是 JRockit。

从名字可以猜一下，应该和火箭一样快吧。JRockit 的特点就是速度快，针对的市场是用专门硬件，专门服务器的商业用户，不针对消费者市场。这个公司宣传自己的产品是：“World’s Fastest JVM”，世界上最快的 Java 虚拟机。

BEA 被 Oracle 收购以后，就被 Oracle 暂停了，Oracle 没必要同时拥有两个 Java 虚拟机。在 2011 年，Oracle 宣布 JRockit 可以免费使用，但是由于多年没开发，JRockit 最高只能支持 Java 6。Oracle 承诺，会将 JRockit 的优秀特征在 OpenJDK 实现。

2.2.3 微软的 JVM

微软也出过 Java 虚拟机，并且性能还相当不错，在 1997 和 1998 年获得过《PC Magazine》杂志的编辑选择奖，在 1999 年宣称自己是 Windows 上最快的 Java 虚拟机。

微软为什么会花大力气帮助 Sun 来实现 Java 虚拟机呢？答案当然是想控制 Java 了。因为 Java，Sun 在 1997 年控告微软违反协议滥用 Java。直到 2001 年，微软败诉，赔偿 2000 万美元给 Sun 公司。

后来微软山寨了 Sun 的 Java，强推自己的 Visual J++，官司输了以后，又开发了 J# 和 C#，再推广 JVM 对微软已经没任何正面意义，所以，微软的 JVM 在 2003 年就停止开发，最晚支持到 2007 年。

有个大翻转比较有趣，Sun 在赢了官司以后，按照协议，Windows XP 不能预装 JVM。Sun 此时才回过味来，如果不预装 JVM，那么对于推广 Java 百害而无一利，于是又开始劝微软继续装 JVM。

那时 Sun 公司已经发布了 Java 1.4，微软只肯在 Windows XP Service Pack 1 中包含一个 1997 年的，基于 Java 1.1.4 版本的 JVM。

2.2.4 Apache 的 Harmony

Apache 也有 JVM，但是却不能称之为 JVM，前面讲过，想宣传自己为 JVM，要先得到 JCP 主导的 TCK 兼容性测试。Apache 得不到这个认证。

为什么会得不到这个认证呢？主要还是理念问题。

JCP 的执行委员 Doug Lea 如此评价 Oracle：“虽然 Sun Microsystems 已经制定了可以推动 JCP 创新的规则，但是 Oracle 并不理会这些规则，JCP 也许会成为任 Oracle 摆布的傀儡。”

Apache 组织是个非盈利组织，Oracle 是个以盈利为主要目的公司。理念谈不拢。Apache 希望 Java 能够不受任何公司的控制，让 Java 完全开源，做了名为 Harmony 的 Java 版本。

后来，Java 的创始人 James Gosling 也建议 Oracle 应该成立一个独立的 JCP 来控制 Java，但是 Oracle 不为所动。2010 年，JCP 开会讨论 Java 7 和 Java 8 的方向，这次会议双方的矛盾最终爆发，Apache 宣布退出 JCP，Oracle 乐见其成。

本来 IBM，Apache 和 Google 是推动 Harmony 的三巨头，但是 IBM 却发表声明说今后将退出 Harmony，以最大的努力推动 Oracle 的 OpenJDK 的发展。随后，IBM 辞去了

Harmony 项目主席的职位。

Apache 一方面无法得到 TCK 认证，另一方面，最坚定的支持者之一 IBM 也跑去了 Oracle 的阵营。在这种境地下，2011 年 12 月 16 日，Apache 宣布取消 Harmony 这个 JVM 的开发。

该项目包含一个名为动态运行时层虚拟机（Dynamic Runtime Layer Virtual Machine，DRLVM）的 Java 虚拟机实现，从官方文档介绍上说，该实现对 Java 6 类库的完成度超过了 97%，我在写书的时候，花了不少时间来研究这个 97% 是如何计算的，没找到更详细的信息。

Apache 有一个坚定的支持者，就是 Google。Google 的安卓系统是 Java 的最大用户之一，他们始终不愿意让 Oracle 掌握命脉，于是，他们坚定的走 Apache 的 Harmony 道路，开发自己的 JVM。

2.2.5 Google 的 Dalvik

Android 是 Google 最大的资产之一，凭借 Android，Google 掌握了手机市场。Sun 公司虽然一直想把 Java 推广到手机中，但是应该没有想到 Google 把这事做成了。

> **i** Dalvic 名字的起源
>
> Dalvik 由 Dan Bornstein 编写的，名字来源于他的祖先曾经居住过的小渔村达尔维克（Dalvik），位于冰岛埃亚峡湾。

在 Apache 宣布不再继续 Harmony 虚拟机以后，Google 从中获取了大量的代码添加到 Google Android SDK 中。在 2010 年 JCP 投票中，Google 和 Apache 坚定的站在一起反对 Oracle，Google 也就顺理成章的没有获得 TCK 认证。当然，Google 的 Dalvik 虚拟机压根就没打算取得 Oracle 的认证，再加上后来的 Android Runtime，Google 从不承认自己的产品是 JVM。

情况确实有点复杂。除了 Harmony 虚拟机，Android 还使用了一个交叉编译器来生成支持非 Java 虚拟机的不同的文件格式 dex。

从 2015 年开始，Google 已经不再采用 Apache Harmony 的类库，转而采用 OpenJDK。无论如何，我总觉得它是一种魔改版的 JVM。

介绍了前面几种 JVM，这只是其中一部分比较出名的。据我所知，有非常多的 Java 虚拟机，因为相对于庞大的 Java 类库，虚拟机的实现成本要小的多。相比于数量众多的 Java 虚拟机，历史上只有三个独立的 Java 类库，它们分别是：OpenJDK、GNU Classpath 和 Apache Harmony。目前仍然活跃的类库只有硕果仅存的一个：OpenJDK。

2.2.6 我的感慨

现在的 JDK 更多，如果你去搜的话，出名的至少数十个，不知名的就更不计其数了，每家稍微大点的公司都有自己家的 JDK。

为什么会这样呢？因为每家公司都不能依赖别人，是的，不止每家公司，每个人都不能依赖别人。中国有句古话叫靠山山倒，靠人人跑。

在不久以前，开源的 C++ 的编译器只有 gcc 这一家，苹果公司当时还没有现在这么有统治力，在苹果公司的开发工具 xcode 中，使用的是 gcc 这个编译器。那……会有问题么？

就像蛋蛋被别人捏住一样，虽然你的朋友可能不会捏，但是你总担心万一哪天，他一使劲，你不就完蛋了么？每当苹果公司需要 gcc 的某些功能的时候，gcc 总是爱理不理，钱留下，功能，慢慢排期吧。于是，Apple 也就不再纠结了，自己做自己的工具链。于是 Apple 开发了自己的 LLVM，Clang，不用再仰人鼻息。

大的公司不但要有自己的 JVM，还要有自己的编程语言，比如 Apple 家的 swift，Google 家的 Go，Microsoft 家有自己的 C#……

当你认为别人可能会攻击你的时候，那他一定会攻击你。这是我看《动物世界》最大的感慨，动物从来不攻击年富力强的动物，专门挑老弱病残来攻击，要么就攻击带娃的动物。只要你有弱点，那么这个弱点一定会成为攻击对象。中国有名言：麻绳专挑细处断。

而人，是动物的一种。

你的年龄会成为劣势，你的子女会成为你的软肋，如果你上有老下有小，年龄还偏大，并且还有贷款，好吧，你就是最弱的那个人。老板会欺负你，这几乎是肯定的。因为你没有能力反抗，你的身后什么都没有，搁在国外，加入个工会什么的，企业还是比较怕的。

希望你能理解，为什么这些公司，拼命的要把握住自己的命运。也希望你能把握住自己的命运。

2.3 “脱口秀”之表扬与自我表扬

前面讲了 Java 语言，javac 和 JVM，了解程序员，javac 和 JVM 之间的关系非常重要。下面我编了一个小故事，希望对理解这三者之间的关系有帮助。这个小故事的名字叫《表扬与自我表扬》，栋哥是程序员。

栋哥：有件事情我想和两位说一下，项目成功上线以后，咱们会有一个名额去东京旅游。我觉得咱们仨个这几年来一直在一起工作，我非常的感谢 javac 一直以来给我写的程序检查错误，给 JVM 输出最终执行的文件。当然了，javac 很棒，JVM 也非常棒，能一直稳定的运行咱们最终的项目。没有 javac 的检查，没有 JVM 的运行，我肯定没法完成任务，这些年来，我一直拿两位当兄弟看待！不过，我还是觉得我最有资格去东京，毕竟没有我，两位工作的再好，也是无本之木，无源之水。两位应该没什么意见吧？

javac：很好，一起工作有难同当的时候是兄弟，有福同享的时候就不是兄弟了，很好！不瞒两位说，如果不是我 javac 在两位之间做翻译，你们知道你们说的什么么？栋哥，不瞒你说，你写代码，如果有我输出的字节码质量的百分之一，我们还用天天加班么？你应该心里有数吧？从标点符号出错，到变量名出错，还有变量类型中的错误……这么说吧，如果不是我屏蔽了这些错误，你的代码如果直接给 JVM 去运行，他能天天死机。没有我，你们两个只能是最陌生的陌生人。

JVM：两位真是王婆卖瓜，自卖自夸啊，你们可真是厉害。我只想纠正两位一下，只要项目一上线，我就要 24 小时工作，你们工作再多，也不过是 996 么。两位既然自视甚高，我就不去旅游了，我还要工作呢。我只是提醒两位，没有人比我干活多。刚刚 javac 的发言，真是要笑死我了，你还当我们的翻译，对我来说，你就是个传话筒，栋哥你可以直接写字节码给我，咱们两个直接交流，还有 javac 说的那些错误处理，我都能处理。只要有错误，我就扔给你 ClassCastExceptions，咱们之间，不用有些人传话，有道是，传钱就怕传少了，传话就怕传多了啊。

javac：哈哈，JVM 你可真是可笑，你和栋哥直接交流，不是我说大话，栋哥用简单的编程语言写的代码都错误百出，你要是让他用字节码和你交流，他一天也写不出一行来。用字节码写程序，那就相当于搞活字印刷，先得让栋哥去学习如何用铅字排版，不，还没有铅字，得让栋哥学习如何将硫化铅提炼成铅。

栋哥：别吵了，没想到我在你们眼里是如此不堪。我并没有否认两位的功劳。我和两位不一样，你们俩只懂 Java 这一门语言，我是程序员，我还懂得 C 语言，Python 语言，PHP 语言……相比于两位来说，我可以说是站得高，看得远，用其它的语言中，其实也能实现两位的功能，比如说在 C 语言中……你们要干什么，打人是不对的，有话说话，不要动手啊，君子动口不……救命啊，打人了……

2.4 小朋友，你是不是有很多问号?

2.4.1 bytecode 是什么?

我在 IBM Developer 上曾经看过一篇文章，文章的名字叫 Java bytecode: Understanding bytecode makes you a better programmer 在这篇文章里，作者说：“对 Java 程序员来说，理解 bytecode 类似于让 C/C++ 程序员理解汇编语言”。

> **i** Kathleen Booth 与汇编语言
>
> Kathleen Booth 女士在 1947 年发明了汇编语言，并且设计了伦敦大学第一个汇编程序与自动解码。

> 汇编语言是一种非常低级的语言，一般不能在不同的平台之间移植。可以把汇编语言看作是机器语言的助记符，在不同的设备上有不同的机器语言指令集。

“bytecode” 里的 byte 有真实的含义，代表 byte，也就是 8 个 bit。8 bits 会产生 256 个组合。Java 虚拟机最多支持 256 个操作符。目前，只用了大概 80%。可以把 bytecode 理解为 “Java 虚拟机的机器码”。

javac 的工作就是把 Java 源代码转化成 bytecode。不管是在 Linux 上生成的 bytecode，还是在 Windows 上生成的 bytecode，都是一模一样的，这构成了 java 跨平台的基石。

2.4.2 javac 是编译器么?

肯定不是 gcc 那样的编译器，javac 不产生机器码。javac 产生的是 bytecode，如果大家了解 Windows 上的 dll 文件，或者 Linux 上的 so 文件，它们的作用非常相似。

在 Java 虚拟机中，负责产生机器码的是 JIT（Just-In-Time）运行时编译器。

在本书中提到 javac 是编译器，我的意思是编译成字节码。也可以认为 Java 上实际上有两种编译器，一个是 Java 字节码编译器，一个是 JIT 编译器。

JIT 编译器的工作原理是这样的：软件在运行的过程中，大部分时间用来运行少量的代码。当软件在解释模式下执行的时候，编译子系统会时刻监控软件的运行，并观察代码中执行最频繁的部分。在整个分析过程中，会捕获一些重要的信息，再根据这些信息进行优化，优化的原则是：把运行最频繁的部分，编译成机器码。这样一来，Java 代码就拥有了可以和 C/C++ 相媲美的性能。

多说一句，如果你多年未关注编程语言的飞速发展，可能错过了一些精彩的时刻。目前，主流的编程语言都已经支持 JIT 技术，像 PHP 8，JavaScript，Python 都已经开始引入 JIT 技术，这也是开源技术的魅力所在，只要一项技术被证实可靠，就会被其它语言采用。

对普通开发者来说，采用 HotSpot 的方法就很好。因为 HotSpot 采用的方法太复杂，普通开发者根本看不懂，看不懂就不会胡乱修改，反而不容易产生性能问题，只要跟着 Java 虚拟机升级就能获得益处。

2.4.3 Java 虚拟机只能运行 Java 语言么?

以前是，现在不是。

Java 虚拟机和 Java 语言现在已经渐行渐远，虽然 Java 编译的字节码只能跑在 Java 虚拟机上，但是 Java 虚拟机并不是只跑 Java 的字节码。目前来说，Java 语言和 Java 虚拟机在一定程度上是独立的，Java 虚拟机也许应该换个名字，比如叫 “万物虚拟机” 更贴切一些。

现在 Java 虚拟机可以执行任何语言生成的合法的文件，只要符合 Java 虚拟机的规范

就好。比如目前比较热门的 scala 语言用其编译器 scalac 生成的字节码，完全可以运行在 Java 虚拟机上。

Java 虚拟机如果要加载类，会先验证它们是不是符合规定的格式，如果符合，就允许其执行。本书不会对类文件的规范讲的太深入，只讲一个有点意思的事情吧。

在 Windows 上，可以用扩展名来识别文件类型，在 Unix 环境下，则要使用一些魔数（magic number）来识别。每个 Java 类文件都以魔数 0xCAFEBABE 开头的，这四个以十六进制表示的字符表示当前文件的类型。大家看到没有，最后四个字母是 BABE，当年 Java 可能没想到会这么火，也没想到这个单词放在今天的舆论下，会涉及性别歧视。现在有转机了，在 Java 9 中，为模块文件（JIMAGE）引入了新的魔数 0xCAFEDADA。也不知道“爸爸（DADA）”这个单词会不会在未来的日子里，也有歧视的意味。

比起社会的文化的变革速度，Java 的更新速度其实很慢的，所以我们要长期使用到 0xCAFEBABE 这个魔数，不过，如果你不用十六进制编辑器打开二进制文件，是不会看到这个涉嫌性别歧视的字串的。顺便提一句，github 这个网站，因为 master 这个词语涉嫌种族歧视，从 2020 年 10 月开始，新建的仓库的默认分支名由 master 改成 main。

> ⚠ 一些趣事
>
> 世界发展的非常螺旋，一方面，BABE 涉嫌对女性的歧视，Master 涉嫌对当年黑人的歧视，纷纷做出了修改。另一方面，歧视好像没有减少，越不让说，通过罚款，把人嘴堵住的方式，会不会让歧视从显性到隐性呢?
>
> 比如，当年有一个叫 Sarah Sharp 的女性开发者对 Linus 抱怨，指责 Linux 内核开发者邮件列表让她感到不舒服。原因是，内核开发者都是一些“生活在黑乎乎地下室里的，只有地震、伽马射线与妈妈”才能伤害他们的人。Linus 的邮件中，经常出现一些“不礼貌”的用词。后来失控了，Linus 也被迫向女性朋友们道歉。
>
> 这件事过去十多年了，根据我的观察，即使 Linus 不太敢大嘴巴喷人了，女性内核开发者也没增加。（至少出名的内核开发者，还是那几个“老人”）。

2.5 “未来版”的 Hello World

无可否认的是，相较于其他众多编程语言，Java 语法确实展现出了更为严谨且相对复杂的特性。然而，在当今时代背景下，特别是伴随大型语言模型的应用普及，实际编程过程中，越来越多的代码编写工作不再完全依赖程序员手动输入，而是更多地扮演代码设计与审核者的角色。在此转变中，Java 语言因其详尽且规范化的语法构造，恰恰有助于提升代码的可读性与一致性。从某种角度来看，这种所谓的“繁琐”特质实际上转化为了有利于团队协作和长期维护的优点。

尽管如此，Java 依旧在简化自己的代码，在 Java 21 中，引入了新的语言特征，该语言特征可以使 Java 的 Hello World 用下面的方式来书写：

```
void main() {
    System.out.println("Hello world!");
}
```

由于这是 Java 21 新引入的特征，所以在编译与运行的时候，跟传统的方式有些不同，要告诉编译器与虚拟机启用新的特征。所以编译的时候要使用下面的代码：

```
javac --release 21 --enable-preview NewHelloWorld.java
```

–release 21 选项指示编译器使用 Java 21 版本的语言和 API。–enable-preview 选项指示编译器启用 Java 21 中的预览功能。运行的时候，也要加上相应的参数，用如下的命令：

```
java  --enable-preview NewHelloWorld
```

在 Java 中，java –enable-preview NewHelloWorld 命令用于编译和运行使用预览功能的 Java 源代码。预览功能是指尚未最终确定并纳入 Java 标准的实验性功能。这些功能可能会在未来的 Java 版本中发生更改或删除。–enable-preview 选项指示 Java 编译器和运行时环境启用预览功能。如果不使用此选项，则无法编译或运行使用预览功能的代码。

2.6 如果你需要使用多个版本的 Java……

说实在的，只要你打工，什么屁事都能碰见。最大的可能是你公司里维护了 3 个版本的 Java，这非常常见，为了跟客户的运行环境相同，你需要安装 3 个 java，这时候推荐使用 SDKMAN 这个工具。

> 提示
>
> 我打工的时候，碰到过用 Java 1.2 版的，到现在我都记得要手动转换 int 与 Integer。考的是 C 证的本，结果人家整来一辆马车。

SDKMAN 是一个用于类 Unix 系统（包括但不限于 Linux、Mac OS X、Cygwin、Solaris 和 FreeBSD）的强大工具，它旨在简化软件开发工具包（SDKs）的安装和管理过程。对本书来说，用 SDKMAN 可以很方便的安装 JDK，其实，它还可以安装各种各样的流行框架与工具，在此不一一列举，安装以后，可以运行 sdk list 自行查看。

SDKMAN 的核心功能包括：一键安装、多版本管理与快速切换。以 Java 为例，我来演示以下以上的核心功能。

一键安装是通过提供的命令行接口，只需一条命令即可安装指定的 SDK 版本，例如安装特定版本的 Java JDK。比如，通过下面的命令可以安装 JDK 21:

```
sdk install java 21.0.2-tem
```

如果你不确定有哪些版本的 JDK 可供安装，可以通过如下的命令来查看：

```
sdk list java
```

这条命令运行之后，会给出非常多的 JDK 供选择。我可以安装任意数量版本的 JDK 在我的系统中，用下面的命令可以查看目前的系统中安装了哪些 JDK：

```
sdk list java | grep installed
```

如果要在不同的 JDK 中切换，可以使用如下的命令：

```
sdk default java <VERSION>
```

这里的 VERSION 只需要替换成你需要的并且已经下载到你电脑中的 Java 版本就可以了。

2.7 程序员故事

《潘新与潘闻两兄弟》

我的老板叫潘新，他有一个小他十二岁的弟弟叫潘闻。最近，潘闻来到他哥的公司实习，他也是学计算机的。老板让我带他弟弟熟悉一下公司和业务。

老板潘新的编程生涯是从汇编和 C 语言开始的，现在他已经不再写代码了，但是对编程有强烈的爱好。

他看了 Java 版的 HelloWorld，问了我一个问题，Java 的类名，函数名，变量名啊，能不能是中文的？

我觉得这是个好问题，正好可以给他弟弟找点事干，就扩展了这个问题：Java 语言支持的 Unicode 字符可以用在什么地方？变量名可以么？函数名可以么？类名可以么？

3 面向对象编程

Contents

难难难，
编程难，
面向对象谈一谈，
对了对象谈几句，
不对对象枉费口舌尖！

找对象难，找个对的对象更难，跟对象过日子是男上加男。但是凭良心说，在编程中，面向对象是真不算太难。

图 3.1: 男上加男

本章，且听我从头考证，细细说来。考证中，我发现面向对象的发展过程中，至少 6 个科学家因此获得图灵奖。更神奇的是，这些科学家对面向对象的看法并不统一。但是，都不同程度的受到了第一门面向对象 Simula 的影响。幸好，有很多的论文可以读，我发现，几乎所有的面向对象的概念，都已经在 Simula 中包含了，只是后来的语言，侧重点不同。Java 也只是其中之一。Simula 语言的故事，我觉得还挺有趣的，也是本章的重点。

但是写书，就要多写点，详细点。本章的计划是通过研究面向对象的历史来学习面向对象编程，具体的技术细节在第 9 章再讲，如果没有其它面向对象编程语言的经验，这一章中碰到不懂的问题直接跳过即可。

对 C++ 或者 Python 语言来说，可以使用面向对象编程，也可以不使用面向对象编程。但是对 Java 来说，没有面向对象就没法编程。从这个意义上来说，Java 是一种非常纯粹的面向对象编程。那面向对象有什么作用呢？James Gosling 在接受采访时说：“面向对象可以使你把一个系统分解开，系统的每个部分都能够被分解，这样对更新、调试等很多事情都有帮助。”

既然面向对象此的重要，我们不妨来研究一下面向对象的历史。

3.1 第一门面向对象编程语言——Simula

2001 年，美国计算机学会（ACM）将代表计算机界最高奖的图灵奖颁发给了两位挪威计算机科学家，一位是 Ole-Johan Dahl，另一位是 Kristen Nygaard，以表彰他们“通过设计编程语言 Simula 1 和 Simula 67，创造了面向对象编程的基本概念”这一伟大成就。

Kristen Nygaard 和 Ole-Johan Dahl

Kristen Nygaard 和 Ole-Johan Dahl 两人对面向对象的贡献太多了。

奥斯陆大学是挪威最好的大学，也是世界最好的大学之一，在这个大学里，有两栋隔路相望的楼，一个名为 Kristen Nygaards 楼，一个名为 Ole-Johan Dahl 楼。

> 这是纪念奥斯陆大学两位杰出的校友，这两个年轻人在编程进入危机的黑暗年代，用努力的工作和杰出的才能，让人们看到了一丝曙光，这丝曙光，开启了编程的新时代——面向对象编程的时代。
>
> 在共同获得图灵奖一年后，2002 年，两人先后离世。编程很难，感谢他们照亮了我们前进的路。

上世纪 60 年代，Simula 语言从诞生之初，就具备了现在大部分面向对象语言所具备的功能，包括但不限于类、继承、动态绑定、垃圾回收等功能。虽然那个时候还没有“面向对象”这个名称，但是在这个名称出现之前，面向对象的概念已经存在了。

让我们看看当年面向对象语言是如何发展而来的吧。

3.1.1 面向对象发明人

1952 年，二战结束不久，挪威政府决定成立一个叫 NCC（Norwegian Computing Centre）的组织，把全国零散的计算机资源整合起来。说起来容易，做起来难，因为这些计算机资源都分散在全国各个部门，比如有的在 NDRE（Norwegian Defence Research Establishment）这样的军方部门，有的在奥斯陆这样的大学，还有的在工业界。之前的挪威，还没有一个组织可以协调所有的部门，NCC 为此而成立。

在二战的时候，所有国家都知道了计算机的威力，挪威也不例外。毕竟挪威是在二战中与纳粹德国抗争时间坚持第二长的欧洲国家，仅次于俄国。战后，1954 年，他们研发了第一台计算机，名字叫 NUSSE（Numerical Universal Automatic Sequential Electronic Computer），这是一台真空管计算机。该真空管计算机制作完成以后，就被运到了上面所说的新成立的 NCC 里。

NCC 刚成立的几年，除了这台 NUSSE 真空管计算机，没有其它的计算机。拥有最多计算机资源的是挪威军方 NDRE，在当时战争的影响尚未完全消退的大环境下，任何国家都是如此，军方优先拥有最多的计算机资源。军方要制造自己的核反应堆，根本没有多余的计算机给别人。NCC 和 NDRE 这两个组织同时存在，计算机资源都很贫瘠，地主家也没有余粮，多一个组织，多一个僧，这两个组织反而加剧了僧多粥少的竞争关系。

军方要增加自己的计算机资源，就从英国的 Ferranti 公司买一台名为 Mercury 的电脑。该项目的负责人叫 Jan Garwick，之前在奥斯陆大学当教授，他招了两个人来当助手。

其中一个助手是 1952 年加入的新兵，他就是 Ole-Johan Dahl，他的任务是给 Mercury 那台电脑写一个编译器，这个编译器叫 MAC（Mercury Automatic Coding）。

另一个助手也是个士兵，是 1948 年入伍的 Kristen Nygaard，他被安排了另外一个项目，该项目是研究一个开放性的问题：如果有一天挪威要制造核武器了，应该如何提前模

拟核武器的爆炸威力?

Kristen Nygaard 以模拟核武器为题写了一篇名为《Theoretical Aspects of Monte Carlo Methods》的论文，随后在军方成了专职的核武器研究员。虽然他的专职工作是研究核武器，但是他兴趣广泛，他的研究范围从单纯的核武器扩大到世间万物，尤其是人力资源能不能也用计算机来建模呢？他对此很有兴趣。

他希望他的方法可以管理社会的组成单位：人。他想通过计算机模拟，使用统计学的方法来解决人工作效率不高的问题。通过计算机来模拟现实世界，这样想法听起来非常的诱人，尤其引起了苏联的注意，当时苏联是计划体制，如果人也可以被计算被管理，那就再好不过了。

这是一种将社会工程学和管理学相结合的产物，Kristen Nygaard 研究人的行为并加以预测和控制。苏联对此十分热衷，一直跟进他的研究，一旦有成果，苏联就把成果在自己的乌拉尔大型机上实现出来。

后来 Kristen Nygaard 与军方发生了一些摩擦，他于 1960 年离开军方并加入前文提到的 NCC。相比于军方，NCC 更多倾向于民用研究，在这里，他有更大的空间可以自由的发挥。在此，他思考这样一个问题，能不能将他在军用上做的工作转成民用呢?

一封当时保留下来的信如实记录了当时的情况，1962 年，Kristen Nygaard 给法国计算机科学家 Charles Salzmann 写了一封信，在信中他透露说他已经有了完整的模拟现实世界的概念，还没动手写语言的编译器，他想等语言先设计好了再动手做这个工作。他认识一位写程序的天才，两人都表示很乐观。在这封信里他提到的那位天才就是他未来的合作伙伴 Ole-Johan Dahl。

> 提示
>
> 这封信可以在这个网址查看：http://cs-exhibitions.uni-klu.ac.at/index.php?id=37

这两个年轻人在 1962 年终于开始共同工作了。

3.1.2 Simula 的研究成果

1963 年，他们两人开始研究并实现这个创意，两年后，于 1965 年完成第一阶段工作。这个阶段的成果在当时被称为 Simula，为了区分，后被约定俗成称之为 Simula I。

随后的两年，两人继续研究，于 1967 年发表了第二版本的语言，也就是后来的 Simula 67，Simula 67 已经有了面向对象的雏形，几个与面向对象相关的概念已经被提出。我们看看这几个重要的概念吧。

3.1.2.1 错误检查

由于 Simula 语言最初研究的是核爆炸，安全性显得特别重要，与其它同期的语言不同，Simula 格外重视安全，毕竟核爆炸可不是只让电脑死机那么简单了。Simula 在设计之初对错误十分重视，Simula 设计了两种错误检查，一种是编译时检查，一种是运行时检查。

现在我们把这种机制叫做类型安全（type safety）。

程序员在讲到类型安全的时候，不同语言的程序员，对这个概念的理解十分不同，讲出来的意义也就十分的宽泛。

在本书中，我所指的类型安全不仅是对一个 A 类型的变量赋值了一个 B 类型的值，[^2] 更多的是考虑到类的层面，比如每个对象创建后都要初始化，外部对类的访问要受相关的限制，对象抛出异常之前要先将自身重置到合法状态等等。类型安全的思想贯穿于整个编程中，而不仅仅在变量赋值这种技术细节。

不止 Java 借鉴了 Simula 的错误检查，目前像 Haskell 语言中的 inspect 方法或多或少都是从这里学来的。

3.1.2.2 继承

Ole-Johan Dahl 曾经这样写过：“增量的抽象，对已经抽象过可以加上一个前缀 C，只要有这个前缀 C，就可以使用其所有的属性。这就是继承的雏形，虽然还没有正式叫继承。”

其实对任何语言来说，都可以复用代码，至少可以通过简单的“复制粘贴”来完成代码的重复使用，但是这样复用的代码并不优美，而且还很难维护。如果能够直接使用别人已经完成的代码，或者自己先前抽象好的代码，而不是自己再重新开始，那么将会有效的降低工作量。Simula 语言在这个方面进行了探索。

Java 当然也借鉴了这个思路，Java 围绕类的概念做了很多工作，其中最重要的概念之一就是：“继承”。在以后的章节里，我会详细介绍继承的概念。

望文生义，继承的表面意思就是从“长辈”那里得来一些东西，在 Java 中，基本也是这样的，采用已有的类，无需改动这些类，就能获得相应的功能，这种方式在 Java 中也叫“继承”。

3.1.2.3 内存回收机制

Ole-Johan Dahl 和 C.A.R. Hoare 合作写过一篇论文《Hierarchical Program Structures》，在这篇论文中，他们介绍了当时的想法：“在实现 Simula 的时候，借鉴了 Algol 60 这个编程语言的实现方法，并且对 Algol 60 的 block 进行了必要的改进，将 block 视为数据，其它的程序可以使用这些 block，这后来演化成了类的使用方法。”

Simula 语言的两位作者写的《The Development of the Simula Language》里，也提到了这一点：“改进了存储机制，引入了内存垃圾回收机制。也许很多语言都意识到了 Algol block 的威力，但是第一个意识到并且做出垃圾回收的语言是 Simula，这在当时是一件了不起的创举。虽然内存垃圾回收很难，还有可能对语言的运行速度产生影响，但是为了以后程序员的方便，这点性能缺失不算什么。Simula 还独创了一种名为二维空闲列表的方式来负责内存垃圾回收。”

在计算机科学中，内存泄漏（Memory leak）是一种常见的 bug，由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

内存泄漏会因为减少可用内存的数量从而降低计算机的性能。最终，在最糟糕的情况下，过多的可用内存被泄露掉从而导致全部或部分设备停止正常工作，最终导致应用程序崩溃。

现在主流的编程语言如 Java 就提供了内存回收机制，这对保证软件的可靠性和安全性非常有好处。如果大家有 C/C++ 编程经验的话，很可能为了一个内存泄露花费数小时甚至数周来查找。有了内存回收机制以后，将节省大量的编程与调试时间。

3.1.2.4 动态绑定

Simula 开创性的使用了动态绑定技术，虽然当时的名字不叫 dynamic binding，而是叫 virtual。刚开始的时候，Simula 所有的属性都是静态的。Ole-Johan Dahl 是在最后一分钟才决定做成动态的，如果对属性定义为 virtual，那么就可以动态绑定了。后来的语言如 Smalltalk 更纯粹，直接把所有的属性和方法都定义为 virtual，在 C++ 中也用相同的关键字 virtual。

> **注释**
>
> 以上讲的这部分内容也在《The Development of the Simula Language》这篇文章中。大家可以自行搜索，或者到 xueban.app 上下载，如果能少刷两小时抖音看美女，看看这篇文章我觉得是值得的！

Java 语言同样借鉴了 Simula 语言，在默认情况下是可以动态绑定的，如果使用了 final 这个关键字，就是静态绑定从而阻止被覆盖。

3.1.2.5 小结

如果用 Peter Wegner 在 1987 年对对面向对象下的定义：object-oriented = objects + classes + inheritance 来衡量的话，至此，面向对象最重要的几个要素在 Simula 语言中都已经有了

雏形。

> **Peter Wegner 介绍**
>
> Peter Wegner 是一位出生于 1932 年的英国科学家，对面向对象编程有很大的贡献。
>
> 在 1999 年，他被奥地利授予奥地利科学与艺术荣誉奖，在去伦敦领奖的路上，出了车祸，昏迷了好久之后才苏醒，但是有了严重的后遗症。

如果只有技术，没有推广，Simula 可能仍然会像世界上绝大部分的编程语言一样无闻。著名的科技史作家、宾夕法尼亚大学历史学教授 Thomas Parker Hughes 在爱迪生的传记《Networks of Power: Electrification in Western Society》里这样评价：“像爱迪生这样伟大的发明家有这样的特征，为了达成目标，他们不仅有超越普通人和科学家的认知，还能够综合利用自己的社会关系，政治资源，商业手段。”

我觉得强有力的推广这种优秀的品质，在 Simula 两位创始人身上体现的也很明显，两位创始人不只是可以埋头搞科研的人，同样和爱迪生一样是政治和商业上的天才。他们不仅有雄心壮志，同样也有政治动员能力、商业运作水平，这两位创始人靠着自己无与伦比的谈判技巧和推广能力，把 Simula 语言从挪威推广到了整个英国、法国、美国、苏联，最后影响了全世界。

虽然这是一本编程的书，我还是希望大家能学到比编程更多的东西。要记住，酒香也怕巷子深。能够把一门编程语言推广起来，可不是一件容易的事情。

当你写出一款优秀的软件，或者创造了一个编程语言，只是成功走完了第一步。如何推广软件或者语言，让别人用你的软件或者语言，是更重要也是更困难的一步。接下来学习一下 Kristen Nygaard 和 Ole-Johan Dahl 是如何推广 Simula 的吧。

3.1.3 推广 Simula

挪威不是计算机强国，如果要推广自己的 Simula 语言，就要先在计算机上运行。在当时，计算机是非常昂贵的，NCC 有意从英国购买 KDF-9 这台大型计算机，但是这台计算机的价格实在太贵了，远远超出了 NCC 的预算。当时美国已经制造出了 UNIVAC 这样一台机器，NCC 想购买这台电脑。

> **提示**
>
> 再宣传一下我做的名字叫《软件那些事儿》的 podcast，在这个电台的 200 期到 203 期里，我讲过 UNIVAC 这台计算机。可以通过泛 podcast 客户端搜索收听。

Kristen Nygaard 表现出了天才般的谈判技巧，他找到了 UNIVAC 在欧洲的负责人 James W. Nickitas，经过了一次谈判，他说服了对方，对方同意把 UNIVAC 打 5 折。还敲定了下一

次会谈，要和 UNIVAC 软件的灵魂人物 Robert Bemer——前 IBM 计算机的核心之一——坐下来谈谈软件和编程语言的事情。

UNIVAC 的软件核心 Robert Bemer 在和 Kristen Nygaard 谈过以后，UNIVAC 不仅可以以半价卖给 NCC，而且还给 Simula 项目带来了一笔赞助，还有，Kristen Nygaard 成了北欧 UNIVAC 的销售代理。这样的谈判水平，已经可以说是高手中的高手了。

这种谈判并不是只发生了一次，在 Kristen Nygaard 推广 Simula 的时候，这种事情屡次上演，他总是能把最难谈的谈判像谈天一样搞定，很快 Simula 就可以在 UNIVAC，IBM360/370，CDC 6000，DEC System-10 等一系列当时最主流的机器上运行了。

Kristen Nygaard 不仅对编程和计算机有兴趣，他一生还投入了大量精力去做政治运动，始终站在劳动者一边，争取劳动者的权力。

Simula 影响了工业界和学术界，也培养出了一大批 Simula 的拥趸，比如 Smalltalk 的作者 Alan Kay，C++ 的作者 Bjarne Stroustrup 都曾声称自己的语言深受 Simula 的影响。

3.2 Java 中的一切皆对象

写这个小标题的时候我有点犹豫，因为很多语言都这样宣传自己。Java 是，Python 是，Ruby 也是。但是如果仔细的钻起牛角尖来，每门语言又都有一些不是“面向对象”的部分。对 Java 来说，如果不用面向对象的方法来写，根本就没法运行，所以我还是起了这个标题。

支持面向对象的语言有很多，这些语言都是按照设计者的意图来开发的，我们学 Java，那么 James Gosling 的意见是最值得参考的。幸好，James Gosling 为了推广 Java，给我们留下了大量的参考资料，他本人写过很多本书，比如《The Java Language Specification》、《The Java Language Environment》等。除了文本资料，还接受了大量的访谈。这些资料，可以解释 Java 中几乎所有的为什么？

前文讲过，作为先驱的 Simula 语言，有两个最重要的“学生”，一个是 SmallTalk，一个是 C++。这两个语言对 Simula 的继承与推广，影响了后来几乎所有的面向对象编程语言。这两个语言，像两座山峰，其它的语言，大部分都是在这两个山峰之间寻找自己的位置，要么离 SmallTalk 近一点，要么离 C++ 近一点，Java 也不例外。

前面讲了好多 Simula 语言的历史，但是“面向对象”这个词并不是 Simula 提出的，而是 SmallTalk 的作者 Alan Kay 提出的。对面向对象，Alan Kay 的理解是：“互相传递消息的程序设计就是面向对象。”对先驱 Simula 语言，Alan Kay 有自己的看法：“我不喜欢 Simula 中继承的做法，我并不是不喜欢类，但是我还没有见过不让我感到痛苦的包含类的编程语言。”

Alan Kay 介绍

Alan Kay 博士是计算机科学领域的重要人物，以其在面向对象编程和图形用户界面的开发方面的贡献而闻名。他于 1970 年代在施乐帕洛阿尔托研究中心（Xerox PARC）工作期间，提出了 SmallTalk 编程语言的概念，这是首批面向对象编程语言之一。Kay 还提出了 “Dynabook” 的构想，这是一种类似于今天平板电脑的个人计算设备，旨在通过计算技术来促进教育和创作。他被认为是计算机领域的先驱之一，并获得了多项重要奖项。

Alan Kay 博士在回答编程的问题时，对面向对象有自己的看法。并且是与 C++、Java 完全不同的看法。一般来说，咱们要同时在脑子里装上几种不同的看法，不要当成纸片人。关于面向对象，想看完整的问答，可以参考：http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

他还说过一句特别出名的话，我在短视频中听到过多次：预测未来最好的方法就是创造未来。（The best way to predict the future is to invent it.）

C++ 语言则是有不同的看法。在 C++ 作者 Bjarne Stroustrup 的著作《The Design and Evolution if C++》中，他认为 “Simula 的继承机制是解决问题的关键”，“类是一种创建用户自定义类型的功能”，“面向对象程序设计是使用了用户定义和继承的程序设计”。我觉得这可能与 C++ 作者的经历有关，他在发明 C++ 之前，一直用 C 和 Simula 语言，C 语言不支持类，Simula 又太慢。他最初的想法是实现一个速度很快的 Simula，最好像 C 语言一样快，C++ 最初的名字是 C with Class 语言。

Java 的位置处于两者之间。接下来，我们来看看 Java 是如何权衡利弊，被设计成这样的。

3.2.1 什么是对象

1995 年，James Gosling 写了第一份 Java 白皮书《The Java Language Environment》，在这本白皮书的第三章，详细的介绍了什么是对象 (object)。

当时是 1995 年，人们对 “面向对象” 有很多争论，现在 20 多年过去了，争论的声音越来越少。就像现在谈起 “面向过程” 编程，几乎没有什么争论一样，“面向对象” 也要经历同样的过程，尘埃会慢慢落定。

生活中充满了对象：车、咖啡机、鸭子、树都是对象。软件也由对象组成：按钮、菜单、图标也都是对象。无论是生活中，还是软件中的对象，都有自己的状态和行为。如何把现实中的 “对象” 建立在计算机中，是 “面向对象” 要解决的问题。

我们可以对一辆现实中的汽车进行建模，让其对应于计算机中。一辆汽车有其状态

（车速、油耗、颜色、手动档还是自动档等等）和行为（启动、转向、停车等）。

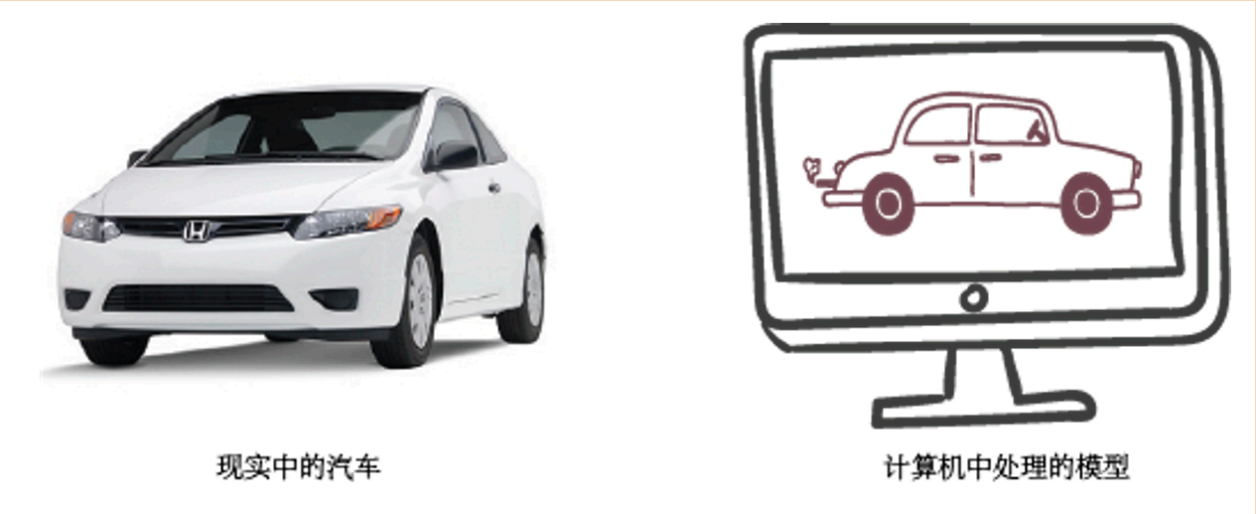

图 3.2: 现实中的对象与编程中的对象

当我们开着这辆车去上班后，可能会在办公室里查查自己买的股票，看看是不是又融断了。股票也是对象，也可以映射在电脑里，也有自己的状态（最高价、最低价、开盘价、收盘价等）和行为（股价波动、退市等等）。

看完十天融断四次的股票，你头晕眼花，想去冲一杯咖啡缓一缓。咖啡机也有自己的状态（水温、咖啡种类等等）和行为（加热、搅拌、流出一杯 Java 咖啡等等）。

所有这一切都可以映射为对象。

3.2.2 对象怎么工作

简单来说，人是对象的一种，人怎么工作，对象就是怎么工作。以牛马为例：牛马. 拉磨 ()，牛马. 吃 (苦)，牛马. 吃 (饭)，牛马. 喝 (水)。

这个 “.” 号就是 Java 中让对象工作的方法。

3.3 面向对象为什么这么难?

面向对象目前来说是最流行的软件开发技术，但是却比较难理解。要掌握这项技术，需要很多的锻炼，就算现在有了很多辅助工具，如果不能从内涵中理解面向对象编程，也无法熟练使用。在这一小节中，我想来讨论一下为什么面向对象这么难以理解，到底难在什么地方?

3.3.1 名词搅拌器

用 2000 多年前孔子的话来讲：“道不远人”。道并不远离人的日常生活，一个人修道的时候如果远离人的日常生活，那他修的就不是道了，用武侠小说中的话来说，可能已经走火入魔了。面向对象编程也是如此，不管这个技术多好，都不能远离程序员太多，如果这种技术已经让大部分程序员搞不懂了，那就不是好的编程技术。

如果面向对象编程让程序员有亲近感，首先在语言上要平易近人。显然，面向对象编程在这一点上做的不好，甚至可以说很差。面向对象有大量的名词，这些名词像把一本新华字典丢进了一个搅拌器，随机搅拌了一堆词语出来，比如下面的词汇：

泛化、特化、父类、多态、属性、委托、注入、构造函数、异常、框架、类库、组件、模式、用例、建模、重构、敏捷、重写、集合、关联……

这只是举了一部分中文的，还有大量英文缩写的没有列出来。这种名词搅拌器一样的编程语言，确实让人很头疼。如果是初学者，一看到这些名词，就已经吓的不敢深入学习了。

存在大量的术语有多方面的原因，如果了解了背后的原因，就不会如此纠结了。

一部分原因是广告的需要，公司推广一项技术，就要写的云山雾罩的，这样显得比较有技术含量。这个不仅是技术行业，在任何行业都是这样，以化妆品为例，我经常盯着老婆化妆品上那些诸如 “活泉精华”“抗氧精粹油” 发呆，每个字都认识，连起来就是不懂这是什么黑科技。

还有一部分原因是技术人员故意避开以前存在的名字，以便显示自己的独创性，其实没没什么独创性，人类总是重复发明旧的科技，然后新瓶装旧酒，用新的名字包装旧的技术。比如 Java 中的多态（polymorphism），这个技术无论是内涵与外延，都和动态绑定（dynamic binding）是一样的，后来又来了一个后期绑定（late binding）和一个运行时绑定（run-time binding）。同一个类似的技术，一下子有了四个名字。这种情况在面向对象的技术发展过程中实在是太普遍了，大家一定要一眼看穿这种鬼把戏。

3.3.2 滥用隐喻

在著名的《代码大全》这本书里，第二章讲了隐喻对理解软件开发的影响。如果选择错了隐喻，那么就会对软件开发有很大的误解。

同样，在面向对象中，如果乱用隐喻，也会让人对面向对象产生误解。

前文 3.2 小节举的例子是参考 James Gosling 写的 Java 白皮书《The Java Language Environment》的第三章。把现实中的对象映射为编程中的类，有助于理解面向对象编程，但是同时也有副作用，会让读者误以为现实事件的物体都可以映射为类。实际情况并不是这样的。

现实中的物体不是由类创建的，而且和面向对象编程中的类大相径庭。现实中的人是父母生的，不是类创建出来的实例。除此之外，现实中的人会根据场景的不同有多种角色，比如我在公司是“员工”，在家里是“父亲”和“丈夫”，对父母则是“儿子”。

但是在 Java 面向对象编程中，一旦根据类创建了实例，那么这个实例就只属于唯一的类，无论时间空间怎么改变，都无法改变这个实例的类型与行为。现实中的我，随着时间的变化，已经从“后浪”成了“前浪”，从“少年”变成了“大叔”。

软件只能涵盖人类工作的一小部分，并不因为引入了面向对象编程而让软件有质的变化。虽然在在推广和宣传面向对象编程的过程中，有意无意的夸大了该方法的优点，但是我们程序员切不可认为面向对象真能模拟现实世界。

在面向对象编程以后，还曾经兴起过一个叫“面向代理”的技术浪潮，但是这个技术很快销声匿迹了。“面向代理”的技术在宣传的过程中，有点类似“人工智能”，能够主动的创造软件，自主的响应人类的需求。很可惜，目前的技术实现不了所宣传的目标。

有了计算机，大量的工作仍然要用人来实现。目前的计算机架构决定了无法完全顶替人类来完成现实世界的工作。引用《Code: The Hidden Language of Computer Hardware and Software》的作者 Charles Petzold 的话来说：“人类有很多的交流形式不能用非此即彼的可能的选择来表示，但是这些交流形式对我们人类的生存又非常重要。这就是人类为何没有与计算机建立起浪漫关系的原因（无论如何，我们都不希望这种情况会发生）。如果你无法用图画或者声音来表达某种事物的时候，你就无法将这个信息用比特的形式来编码。”

是啊，像林俊杰的歌中唱的：“确认过眼神，我遇上对的人。”如果你相信面向对象可以对世间万物进行建模，那么，我们如何能用面向对象的方法对眼神进行建模呢?

隐喻能帮助我们理解面向对象编程，但是也会让干扰我们对面向对象编程的理解。这也是面向对象编程难以理解的又一个因素：滥用隐喻。

3.3.3 过度宣传

其实，比起现在的宣传，如果说编程的宣传有点捕风捉影，现在的宣传简直就是平地扣饼，编程已经是已经很内敛了。

> 面向对象编程是正确程序的替代品。（object oriented programs are offered as alternatives to correct ones.）—— Edsger W. Dijkstra

Dijkstra 的这句话提醒我们不要教条，过度宣传会给用户不切实际的幻想，经过 30 多年坚持不懈的宣传，让程序员误以为就没有面向对象解决不了的问题，我把这称之为“面向对象”综合症。

通过前面介绍 Simula 67 的历史，我们会知道 Simula 67 这个面向对象语言的先驱，当时并没有“发明”出面向对象这个概念，而是后来 Smalltalk 语言的作者提出的。在 Simula

67 中，引入 class 的作用仅仅是汇总子程序和变量的结构，到了 Smalltalk 中才开始使用继承结构来组织类库，所有的类都来自于 Object 类。

由于 Smalltalk 和 C++ 的流行，面向对象的概念已经不是某个人能左右的了。这种现象已经不能用技术来解释了，让我想起了郭德刚讲的一段相声，当一个人听到一个段子的时候，会添油加醋的渲染一番，然后再传给下一个人，下一个人也是如此这般，等传到十个人的时候，事情已经面目全非了。

以我工作的经历，那些完全不懂技术的领导也知道面向对象的好处，而且很有主见。领导或者项目经理在给程序员提意见的时候，经常说：“不要想得那么难，用面向对象的方法，对这些东西建个模，很容易的。” 然后，我还要装作恍然大悟的样子夸领导几句。

这都是过度宣传导致的结果，面向对象编程被包装成了无所不能的银弹。虽然现实世界是由一个又一个的对象组成的，但是想把现实世界映射在程序中，并不容易。

> **注释**
>
> IBM 大型机之父佛瑞德 · 布鲁克斯写过一篇论文《没有银弹：软件工程的本质性与附属性工作》。
>
> 在这篇论文中他强调强调由于软件的复杂性本质，而使真正的银弹并不存在；所谓的没有银弹是指没有任何一项技术或方法可使软件工程的生产力在十年内提高十倍。
>
> 其中银弹（Silver Bullet）的来历是：在欧洲民间传说及 19 世纪以来哥特小说风潮的影响下，银弹往往被描绘成具有驱魔功效的武器，是针对狼人等超自然怪物的特效武器。后来也被比喻为具有极端有效性的解决方法，作为杀手锏、最强杀招、王牌等的代称。

两界普利策得主约翰 · 卡雷鲁写过一本书叫《坏血：一个硅谷巨头的秘密与谎言》，在书里他这样写道：“雾件（Vaporware）反映了计算机行业的一种倾向，在涉及市场营销时，做法非常轻率散漫。微软、苹果和甲骨文都曾被谴责某些时候都有类似的做法，过度承诺是硅谷的标志性特征之一。” 雾件（Vaporware）这个词现在已经不太出现在媒体上了，但是这种过度吹嘘的行为至今仍然存在，甚至愈演愈烈。

以上就是面向对象为什么这么难的原因。

很多的程序员并不清楚为什么要用面向对象编程，也不了解面向对象编程的历史，反正大家都在用，有关面向对象编程的工具有很多，有好用的 IDE，有 UML 工具，那我也跟着用用就好了。在这种心态下，面向对象编程就成了一种很神秘的东西，“虽然不理解，但是大家都在用”。

本书要做的事情就是尽量把 Java 和面向对象的知识点讲清楚。为尽量避免上面讲到的三种缺点，本书会使用如下的原则：

1．为了避免名词搅拌器，尽量不过多引入术语。

2．会使用隐喻，但是不会到处使用隐喻，最终还是会落实到代码上。

3．不过度的宣传 Java 以及面向对象的强大，而是从为什么引入这种技术以及这些技术能解决什么问题的角度来讲解。

3.4 面向对象可以很简单

在大体了解了什么是对象，如何操控对象以后，我们再用例子讲解一个为什么面向对象编程有优点？

无论写什么软件，都要先进行设计，设计有简单有复杂，复杂的可能要采用很多文档图表来描述其设计过程，简单的可能只需要想一下，但是，都要有设计的过程。在设计的时候，第一原则是：要尽量让好事发生，让坏事不要发生。

设计阶段，人自然而然的是用面向对象的方法来思考问题。比如考虑一间房子时，不会去想门、窗户和玻璃的具体组成。如果房子很多，我们的视角会放大到小区。如果小区很多，我们的视角会再次发生改变，抽象到城市的层面。城市再多，会抽象成一个国家。

一层又一层的抽象之中，我们实际上是用面向对象的方法隐藏了很多细节信息，这主要是为了让我们的大脑有足够的处理空间。

这是有科学依据的，著名的计算机科学家 Dijkstra 曾经说过：“没有人的大脑可以容纳下一个现代的计算机软件”。目前的软件越来越庞大，数百万数千万行的软件早已经不是什么新鲜事，没有人可以了解一个大型软件的方方面面。我们如何确保在不知道其它功能的前提下，安心的做我们这一块功能？又如何确保我们做完了这个功能，能保证和其它部分和谐运行呢？面向对象就是来解决这个问题的。

> 软件危机的主要原因，很不客气地说：在没有机器的时候，编程根本不是问题；
> 当我们有了电脑，编程开始变成问题；而现在我们有巨大的电脑，编程就成为了
> 一个同样巨大的问题。
>
> – Dijkstra

目前，人们找到的一个解决方法是将整个系统分解为子系统，然后再将子系统分解为类，再将类分解为子程序，最后再把子程序变成可以运行的代码。

如何设计类是一门大学问，本书将会用一整本书来反复研究类和面向对象这件事。下面就先以一个足球游戏来简要的说明一下，先有个大体的概念。

3.4.1 FIFA 足球游戏

足球是世界第一大运动，上一届世界杯说有 22 亿人观看，这有可能是真的么？在游戏行业，有两个最著名的足球游戏，一个是 KONAMI 的《实况足球》，一个是 EA 出的《FIFA》。以 FIFA 19 来举例吧，假设 EA 把我们请去设计 FIFA 游戏，我们应该怎么做？

想象一下，如果我们要去设计 FIFA 这款足球游戏。这个游戏里，有数百支球队，数百位教练，数千名球员，看台边上举着相机的记者，看台上数万球迷……想想头就大了。幸好，我们可以用面向对象的方法把所涉及的对象抽象出来。

也许有些人没有玩过 FIFA 足球，为了讲编程概念比较容易，我先用文字描述一下吧，如果可能，自己找个视频看看，或者自己玩一下就更有直观的体验了。

大家在电视里看过足球吧，FIFA 游戏尽量模拟真实的足球，只是把控制权交给了玩家。玩家可以用手柄或者键盘控制场上踢球的球员。现实足球场上发生的一切，都尽量的在游戏中加以模拟。当然有所取舍，因为完全模拟是没办法做到的，如果仔细看得话，会看到场边有举着相机的记者，他们的动作是一模一样的，看台上的球迷也都雷同，场边的第四官员和教练动作也比较僵硬。原因就计算力有限，FIFA 使用的寒霜引擎非常耗费资源。好钢用在刀刃上，好运算力用在渲染球员上。

对这个游戏来说，最重要的部分是球员。让我们来看看如何抽象游戏中最重要的球员吧。

3.4.2 对球员进行抽象

下面是 FIFA 19 中的截图，在 FIFA 中，球员有很多的属性，我挑一小部分说一下。

图 3.3: FIFA 游戏中的截图

每个球员都有名字，图里面用的是 Messi，还有球员在场上的位置，Messi 在 FIFA 19 是右边锋 (RW)。再就是 94 这个能力值，这个能力值是 FIFA 游戏开发组给每个球员的综合评价，梅西和 C 罗是最高的 94 分。目前来说我知道是最低的能力值是天津权健的吴磊，能力值仅有 48 分，现在中超的射手王武磊能力值是 76 分。

还有一些属性是国家：阿根廷（显示的国旗），所属的球队（巴塞罗纳）。还有下面数字分别代表的单项能力值（满分是 100）：

PAC(速度)：87

SHO(射门)：92

PAS（过人）：92

DRI（盘带）：96

DEF（防守）：39

PHY（体能）：66

以上列出了球员的状态，除了属性，还要列出球员的行为。

当设计类的时候，主要考虑类的两个方面：

1. 这个类有什么属性？

2. 这个类有什么行为？

球员在球场的行为有很多种，像 FIFA 游戏，场上的每个球员可以响应 86 种动作[1]，目前只需要列出其中的几种：停球，传球，射门，防守。根据这些属性和行为，可以画出如下球员类所示：

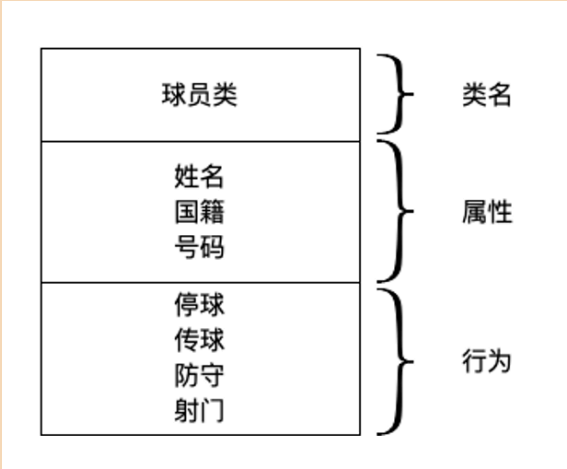

图 3.4: 球员的类图

经过这样的抽象，我们把球员的信息得以大幅简化。编程首要的任务是控制复杂性，用这种抽象的方法能有效的简化编程过程，这样，我们可以忽略掉次要的属性，而将主要的精力放在重要的属性上。在我们设计的球员类上，对外部程序来说，类名是可见的，但是类的内部信息比如如何实现停球，传球则是要隐藏的。学术一点叫数据封装与信息隐藏。

现代只要号称支持面向对象的语言都会提供数据封装与信息隐藏的机制，包括 Java。只是不同的编程语言支持的方式有些许不同，但是总体来说，都没有超过上一小节中讲的 Simula 语言所规定的范围。接下来，把数据封装和信息隐藏分别再讲一下。

3.4.2.1 数据封装

在讲数据封装以前，我们先考虑这样一件事，如果没有数据封装，我们应该如何写软件？比如游戏中的一个球员，在处理的时候，我们要分别处理每一项属性，比如身高、国籍、球衣号码还有各种球员的动作等等。

[1]如果大家对 FIFA 的操作有兴趣，可以到 xueban.app 这个网站中查看具体的动作，我当年闲的蛋疼，练了不少花哨动作。实际上，一般玩家常用的动作在 10 种以下。而且吧，如果你跟哥们踢球，玩的太花，容易被打。

数据封装的意思就是把这些乱七八糟的东西，封装好了，只传递一个对象过来。

目标是明确的，但是具体到实现上，就有各种各样的问题。不同的语言针对这些问题分别给出了自己的解决方案，本书在后面的章节中将会详细的介绍 Java 对这些不同问题所采用的取舍。

在 Java 中采用的是和 C++ 语言相似的方式来定义类，都是用的 class。但是在对象的分配与引用，是否回收上有些许的不同。

3.4.2.2 信息隐藏

当数据都被封装起来以后，为了提高可靠性，不管是有意还是无意，我们都不能让用户通过随意的方式修改里面的内容。如果要修改，只能通过我们允许的方式来修改，这就是信息隐藏。

黑箱操作在现实中一般不是好事，但是在面向对象编程中，把对象包装成黑箱是非常好的方法。这样的好处非常多，比如可以限制变动的影响范围，可以在黑箱内部修改数据结构或者方法，而无需修改调用程序本身。还可以“使用我们允许的方式”监视数据的使用，这样会促使我们思考类中的数据是否应该是全局的？在 Java 编程之中，我们会经常发现，“全局数据”其实是某个对象的数据。

具体到我们的球员类，我们不会想让外部的程序随意更改对象的名字，比如把 Messi 改成 Ronaldo。同时，我们又想让一些属性可以通过我们允许的方式被修改，比如更改场上的位置，有些球员可以踢前锋，也可以踢中场，这是可以修改的。基于这种要求，Java 也提供了相应的机制，字段是私有的（private）还是公有的。

3.4.3 继承

每个球队中都有一个场上队长，像梅西目前担任球队的队长，布教授是球队的副队长。从编程角度来看，队长和普通球员绝大部分属性是一样的，在球场上，队长会戴个队长的袖标。当把队长换下场的时候，有时候会出现一个过场动画，会有球员把队长袖标戴给副队长戴上，在玩游戏的时候，我会为这种细节感动。

队长类和球员类几乎是一样的，虽然直接把球员类拿过来是不行的，但是只要把球员类稍微的做一下修改，那就可以了。这时候又产生了一个问题，我们刚刚讲过，类要保证数据封装和数据隐藏，如果直接修改源码的话，修改人员要了解球员类的代码，才能做出修改，这很困难。为此，Java 引入了继承的机制来解决这个问题。

在编程中，如果新的类（比如队长类）可以继承已有的类（比如球员类）的数据和功能，而且新的类可以允许对原有的类增加或者修改一些内容，那么复用将会变得更加便利。用这种方法，程序员就可以从已经完成的类开始，修改得到其子类型，满足新的要求。这还能理顺类之间的关系，队长也是球员，两个类之间是父子关系，球员类是父类，队长类

是子类。

我们要给队长加个行为：挑边。挑边的意思是开场前，主裁判会扔个硬币，两队队长会根据硬币的正反来选择进攻的方向。按照前面的理论，只要有个机制确保能继承球员类即可。示意图如下：

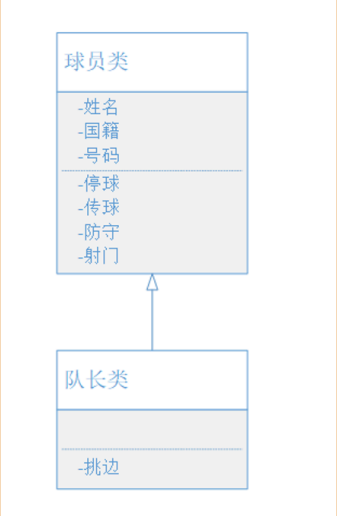

图 3.5: 球员继承

在这里，先有个概念即可，在实际的编程中，继承并不是这么单纯，实现继承的策略也不像这个例子中这么清晰。目前先不用担心这些细节，在第 9 章开始，我们再来一起学习 Java 提供的继承机制。

从理论上来说，继承可以从多个父类中继承，这种叫多继承；也可以从单个父类中继承，这种叫单继承。不同的语言在这个地方有所取舍，Java 采用了一种较为简单的方式，只支持单继承。这样虽然简单，同时也失去了多重继承的优点，为了弥补这种缺点，Java 语言实际上可以借助接口（Interface）来实现多重继承的功能，这个在以后的章节我们一起学习。

3.4.3.1 继承的缺点

继承的好处我们讲了好多了，再讲点继承的缺点。

前面讲了类的优点，可以把数据封装起来，还可以隐藏信息。但是过度的使用继承，会导致有多层的继承树，这样编程风格的代码，处理起来非常困难。假设我们有这样一个类，由多层继承而来，该类中有一个方法。如何找到这个方法最初的定义位置呢？没有好办法，只能先查父类，如果没有，再查父类的父类……就算查到了，也不敢贸然修改，因为不确定这个类影响的范围有多广，如果修改了这种类，一定要用一种叫回归测试的方法进行广泛测试。

> **回归测试是什么?**
>
> 回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。自动回归测试将大幅降低系统测试、维护升级等阶段的成本。

总之，凡事要有个取舍，我在工作中读过 10 来层的继承代码，理解起来非常难。尽量兼顾代码的复用度和代码的清晰度。

3.5 虚构的访谈

《FIFA 首席设计师》

以下故事纯属虚构。帕克休斯是一个虚构的记者，栋哥是一个虚构的 FIFA 首席设计师。

帕克休斯：你好，栋哥，我们都知道你已经成了 FIFA 的首席设计师，你能谈谈这次设计的目标么?

栋哥：这次设计我只有一个小目标，先赚他一个亿！

帕克休斯：呃……不是从经济收入上，能从程序设计的角度谈一下么，比如，听说你在设计这款足球游戏的时候用到了面向对象的编程方法，我对面向对象也有所了解，我在大学里的时候老师就讲过，面向对象就是类，包括封装，继承好像还有多态什么的。我有一个问题，听老师讲了这么多，还是有点不清楚，不知道这些具体怎么用，你能给我讲一下么?

栋哥：学校里主要是书本知识，书本知识是一种提炼与抽象，套用一句常用的话来说就是源于编程，又高于编程。在学校中，绝大部分学生最关注的是考试成绩，在现实的编程中，你认为编程最重要的是什么?

帕克休斯：是如期的发布产品?

栋哥：也可以这么说吧，如期发布高质量的产品是软件开发最困难的事情。像 FIFA 每年都是雷打不动的 9 月末发布，去年 9 月 28，今年还是 9 月 28。而每年的 FIFA 都要进行不少的改动，用软件开发大师爱德华 · 贝拉德话来说就是：“Walking on water and developing software from a specification are easy if both are frozen.” 翻译成中文是：“一边走在水上一边开发软件是非常容易的，只要两者都冻结了。” 但是现实社会不停的变化，软件需求也要不停的变化，比如在 2019 年 7 月份，我得知我们的竞争对手 KONAMI 在 7 月 3 日宣布和曼联成为合作伙伴，7 月 12 日宣布与拜仁慕尼黑达成协议，7 月 16 日拥有 C 罗的尤文图斯也加入了他们的阵营。为了应对这些变化，在 9 月末发布的《FIFA 20》中将不能采用这些球队的队名、队徽、球衣和球场，只能用虚拟的模型替换，侵犯了版权要赔钱的，还有两个月时间，对我们来说，变化是持续的。

但是，我们不能因为这种变化就延期游戏的发布。

帕克休斯：那么，你们是如何将这种影响减少到最低的呢？

栋哥：这其中最主要的关键是设计，一个优雅的软件，像 FIFA 这样的软件，是可以很容易应对改变的，也非常容易进行扩展和复用。这种设计可以称之为“面向对象设计”，优雅软件设计的关键之一。

帕克休斯：你说的“面向对象设计”和“面向对象编程”是一回事么？

栋哥：面向对象设计不仅仅从微观上着眼于编码，还要从宏观上着眼于产品。所以这两者不是一回事，但是又相互联系。面向对象设计需要面向对向编程，但是又不仅包含面向对象编程这一个概念，还要更多内涵，主要有这四个方面：

1. 面向对象编程的方法
2. 代码复用
3. 只改变极少量代码就可以改应对需求变化
4. 不改变任何代码就能扩展软件的功能

我分别来讲一下这四个方面，面向对象编程方法已经讲了太多了，这里就先不讲了。讲第二个代码复用，FIFA 已经做成了一年发一版的年货游戏，不可能每次都从头做起，代码的复用显得特别重要。实际上 FIFA 的引擎不仅是 FIFA 自己在用，EA 的很多游戏都在用，目前使用的是寒霜引擎，EA 公司的通用引擎，还用在《战地》、《极品飞车》等其它十二款游戏上。[2]

第三点是只改变极少量代码就可以应对需求的变化。像前面说的，7 月得知公司丢了几个球队的版权，9 月游戏发布，这种调整要非常迅速。

第四点是不改变任何代码就能扩展软件的功能。这一点，FIFA 做的也非常优秀，像 FIFA 19 刚发布的时候，并没有女足世界杯，但是在女足世界杯前夕，官方发布了补丁，该补丁升级了 22 支女足世界杯国家队，球衣队徽，官方比赛用球，真实的球场等等，这些扩展没有修改任何代码就完成了。

帕克休斯：你说的这些听起来不错，是你自己悟出来的么？

栋哥：当然不是我自己悟出来的。计算机行业中有很多人花费大量时间来研究“面向对象设计”，并且总结出了很多行之有效的原则，形成了不少设计模式供我们借鉴。像 SOLID 原则就是其中之一。

帕克休斯：SOLID 原则？

栋哥：是的，SOLID 是面向对象设计和面向对象编程中几个重要的原则，SOLID 是这几个原则的首字母的缩写，总共用有五个原则。分别是：

[2] 在 2010 年的时候，EA 通过不停的收购，拥有了很多世界知名的游戏品牌，但是也带来了一个巨大的麻烦，当时 EA 有十三款不同的游戏引擎，有虚幻引擎开发的《质量效应 2》，有变色龙引擎开发的《极品飞车》，这导致暴涨的开发费用。

1. 单一责任原则（The **S**ingle Responsibility Principle）
2. 开放封闭原则（The **O**pen Closed Principle）
3. 里氏替换原则（The **L**iskov Substitution Principle）
4. 接口分离原则（The **I**nterface Segregation Principle）
5. 依赖倒置原则（The **D**ependency Inversion Principle）

分别取上面各条原则黑体部分的首字母，就组成了 SOLID 这个单词。因为这几条原则太重要了，不光在面试时候经常被考官问，在实际的编程中，也要用到这五条原则。

帕克休斯：这样说太笼统了，你能详细讲一下么？

栋哥：好的，先来看第一个原则，单一责任原则。这个原则是说一个类只有一种类型责任。就拿球员类来说吧，十一名上场的球员有一名是守门员，守门员的职责和普通球员有所不同，所以，守门员和球员要分成两个类。

帕克休斯：也就是说在划分类的时候要根据类的责任了？

栋哥：可以这么说，像我前面所举的例子，普通球员和守门员大部分情况下都是一样的，甚至可以认为普通球员能做的，守门员都可以做。像德国的诺伊尔就经常跑到禁区外，简直就是个后卫，所以人们称之为门卫。在禁区之外，门将和普通球员一样，他们的区别在禁区之内，在禁区内，门将是可以用手的。所以，在设计的过程中可以这样来设计：

1. Player 类：普通球员类中只要做正常的设定即可。
2. Goalkeeper 类：门将类可以继承 Player 类，然后再添加在禁区内用手的操作。

帕克休斯：好吧，我就假装听懂了，能继续介绍一下这个开放封闭原则么？

栋哥：开放封闭原则也非常的重要，用一句话来解释可以称之为一个类要易于被扩展，但要难于被修改。通俗来说就是核心的类，如果要增加或者改变其功能，最好的方法是扩展这个类，除非万不得已，不要修改这个类，因为一旦修改，众多依赖于这个类的其它类都会受到影响。开放封闭原则的开放是对扩展开放，封闭则是对修改封闭。

帕克休斯：你这个原则让我想到了我们自己也是易于扩展而难于修改。如果我想社会一点，我可以贴个纹身贴，来个爬行动物贴在身上。如果我想显得潮一点，可以来个锡纸烫。这些都没有对我的身体造成影响，如果要动手术整容那可就比较麻烦了。所以，我觉得人类也是易于扩展，而难于修改的。

栋哥：你这个想法非常的好，我还没想到呢。对开放封闭原则来说，最重要的是抽象，把最重要的特征与功能抽象出来，如果抽象做的不好，这个类就设计的不够有扩展性。

帕克休斯：那你能给举个设计的比较有好的例子呢？

栋哥：这种扩展性设计的很好的软件有很多，比如浏览器上网，很多编辑器都有很好

的扩展性。拿浏览器来说，这几大主流的浏览器在写软件的时候，都是对一个抽象的服务器来写，至于这个服务器是 Apache 还是 NGINX，还是微软的 IIS，都没关系。所以，这样就减少了对具体服务器的依赖。对 NGINX 服务器来说，如果要扩展功能，是可以完全不用改变原有代码时行扩展的。所以，这种设计又开放又封闭。

帕克休斯：那还有第三个原则叫里氏替换原则，这是什么意思呢？里氏这个名字好奇怪啊。

栋哥：和牛顿定律是牛顿提出来的一样，里氏替换原则也是一位叫 Barbara Liskov 的女士提出的。Liskov 女士发表了一篇名为《数据的抽象与层次》的论文，在论文中她提出了这样一个观点："如果对每一个类型为 S 的对象 o1，都有类型为 T 的对象 o2，使得以 T 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 S 是类型 T 的子类型。"

> **注释**
>
> 这篇论文我放在 xueban.app 中，在论文《Data Abstraction and Hierarchy》中的原话是：If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.
> 确实比较抽象，我也看不出来这就是里氏置换原则，但是，别人都说是，所以，我也只能跟大流了……

但是这个表述实在是太科学严谨了，于是有人把这个描述通俗化了："派生类（子类）对象可以在程序中代替其基类（超类）对象。"

帕克休斯：这个说法还是太严谨了，能再举个通俗的例子么？这些原则是为了科研而提出的么？

栋哥：这些原则都是长期的经验总结，并不是为了科研而科研。我们在做设计的时候，最重要的目标是简化编程的难度，如果设计得当，可以让我们在写一个类的时候，安全的忽略其它的类。我们设计类的时候，也是冲着这个目标去的。

我们在使用类的继承时，不能给自己添麻烦。设想一下，如果我们继承的子类将父类已经实现的方法给重写了，会给我们写程序带来相当的惊喜。为了不让我们惊喜，使用继承，要遵守的原则是，只要出现父类的地方，都可以用子类代替。

比如说前面我举的例子，门将类继承自球员类，只要出现球员的地方，都可以用门将来代替。在现在中也是如此，只要教练愿意，是可以派上 11 个门将的，虽然没人这么做过。球员能拥有的属性和动作，因为门将是子类，所以门将都有，但是在程序中，反之并不如此，在程序中，门将比球员多了一些属性和动作，比如用手扑球，如果用父类来取代子类，那么这个门将就不会扑球了。

帕克休斯：我明白了，也就是说，在做测试的时候，所有针对父类的测试，在子类上都要通过，是么？

栋哥：是的，就是这个意思。所有在父类上执行的动作，在子类上都要执行的一模一样，这就是替换的原则，子类可以代替父类。

帕克休斯：那继续说一下第四个原则，接口分离原则吧。

栋哥：我举个例子吧，你应该去过火车站飞机场这种场合，那些地方有那种免费给手机充电的接口，长的样子就像下面这样。

现在每个人都有手机，你会用这样的手机充电线么？

帕克休斯：当然不会用了，我的手机只有一种接口，有那么多，根本没用啊。

栋哥：是的，在编程中也是如此。如果接口被设计的大而全，就像上面的充电器一样支持所有手机，会让程序显的非常杂乱，从而难以维护。最好的设计是只用自己有用的接口，将大而全的接口分离。当然了，在特殊的情况下，像机场，大而全的设计也有一定的存在价值，但是总体来说，在编程中，这样的设计基本上是没有好处的。如果有些类用不到大而全的接口，一定要记得分离。

帕克休斯：你这样说我就清楚了，我可不想拿着那么恐怖的充电器。那再讲一下最后一个设计原则吧，叫依赖倒置原则吧。

栋哥：先要理解什么叫依赖，在编程中的依赖和现实中的依赖有些不同。我用依赖造几个句子，比如“她不喜欢依赖别人，自己的事情总是自己做”。再比如“老婆和老公在生活中互相依赖”。在现实中，当我们说某个东西依赖另一个东西的时候，往往有些弱者依赖强者的味道在里面，或者互相依赖。在现实中，可以你依赖我，我依赖他，他依赖你，我们就像一个团结有爱的一家人。

但是，在软件中，你要是设计出了一个 A 类依赖 B 类，B 类依赖 C 类，C 类又依赖 A 类的死循环，那就有你受的了！

当我们说 A 类依赖 B 类的时候，是指 B 类是以局部变量的形式存在于 A 类之中。

帕克休斯：还是举个例子吧，这样说有点摸不着头脑。

栋哥：比如说我要去旅行，假设一个怀着“世界这么大，我想去看看”的旅行者类好了，这个旅行者有一个方法是旅出行，参数是某种交通工具。当我们出去的时候，是无法得知自己用哪种交通工具的，可能是走路，可能是坐三蹦子、出租车、高铁、飞机、轮船……

我来举个软件的例子，我开一下电脑，空口无凭，只要稍微看下代码就知道了。你现在还写代码么？

帕克休斯: 略懂一些，你讲吧。

栋哥：懂一点就好，看看下面的代码，这里面的 Traveller 旅行者类依赖巴士车类。

```
    class Bus {
        public String start(){
            return " 我是巴士车，来不急了，快上车！";
        }
}

    class Traveller {
        public void travel(Bus bus) {
            System.out.println(" 世界这么大，我想去看看");
            System.out.println(bus.start());
        }
    }

    public class TravellerTest{
        public static void main(String[] args){
            Traveller traveller = new Traveller();
            traveller.travel(new Bus());
        }
}
```

帕克休斯：这个我理解了，只要是巴士车，我们的旅行者就能远行。

栋哥：是的，但是还有个问题，如果出行的路上，没有巴士车呢?

帕克休斯：那可以坐其它交通工具啊，交通工具又不止一种。

栋哥：但是，我们看看上面的旅行者类，里面的 travel 有个参数是 Bus 类，如果不是 Bus 类，那就没办法了。只能一种交通工具准备一种调用方法。

帕克休斯：这也太不合理了吧！

栋哥：非常的不合理，仅仅是换一种交通工具，就要不停的修改旅行者类，这不是好的设计。原因是 Traveller 类和 Bus 类之间的耦合性太高了。必须降低他们之间的耦合度才行。

帕克休斯：那应该怎么做呢?

栋哥：可以引入一个抽象的接口 IStart。只要是交通工具，都可以跑起来。看看下面的代码：

```
 interface IStart{
     public String start();
 }

 class Bus implements IStart{
     public String start(){
         return " 我是巴士车，来不急了，快上车！";
     }
 }

class Tank implements IStart{
    public String start(){
        return " 我是坦克，上车吧！";
    }
}

class Traveller {
    public void travel(IStart vehicle) {
        System.out.println(" 世界这么大，我想去看看");
        System.out.println(vehicle.start());
    }
}

public class TravellerTest{
    public static void main(String[] args){
        Traveller traveller = new Traveller();
        traveller.travel(new Bus());
        traveller.travel(new Tank());
    }
}
```

帕克休斯：我知道了，这样无论怎么扩展，都不用再修改旅行者类了。

栋哥：是的，这就是依赖倒置原则。代表高层的旅行者类一般负责完成更主要的业务，一旦对它进行修改，引入错误的风险极大，使用依赖倒置原则就可以不对这个类进行修改。依赖倒置原则的核心就是要面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

帕克休斯：是不是只要掌握这几个设计原则就能设计出完美的类呢?

栋哥：当然不能这么说，只能说一般情况下设计类的时候都要考虑这几种原则，基于这几种原则，人们还总结了很多经验，这些经验有个名字叫设计模式。

帕克休斯：我知道设计模式，没想到设计模式是建立在这些原则之上呢。

栋哥：其实设计模式并不神秘，算是一些长期以来的经验总结吧，有了设计模式，能让我们少走不少弯路呢。

帕克休斯：今天的采访就到这里吧，下次有机会再来谈谈设计模式的话题。

栋哥：好的，再见。

3.6 程序员故事

《潘闻来到公司》

部署网络的工作又有趣又繁琐，有时候繁忙的工作会让我的睡眠严重不足，但是我喜欢这个工作，所以，我每天都会早早的赶到公司，去查一下今天又有什么工作要做。

早上我赶到公司的时候，老板已经在公司了，我照例去问他有什么安排。他说："今天你去接我弟弟吧，他大学二年级，要从外地赶来，他学的是计算机专业，这是他的电话，你给他打电话，问问什么情况。"

"没问题。"

"对了，他叫潘闻。"

我打了那个电话号码，在接通电话的那一刻，我以为把电话打给了老板，电话那头传来了一个和老板一模一样的声音。我赶忙仔细确认了一下电话号码，没打错。我问："你是潘闻么？你听起来和你哥一模一样。"

电话那头传来了爽朗的笑声："是吧，大家都这么说，我上午 11 点半就到了。我在火车站等你。"

我在车站接到他，简单的寒暄后，我认定潘闻是个阳光大男孩。他坐在副驾驶上，很自然的玩起了手机，一路上把他在手机上看到的新闻读给我听。东方的天空已经有些阴云，我问他："你查一下手机上今天会下雨么？"。

"今天会有雷阵雨。"

我没话找话的说了一句："你性格和你哥一点也不一样。"

潘闻沉默的片刻时间里，我意识到我说错了话，他悠然的回答："都说哥哥是上帝送给弟弟的天使，我哥是上帝送给我的考试。"我着急把话题引开，听到他说考试，就连忙问道："你现在刚刚考完期末考试吧？"

“是的，刚刚考完一些副科，还有两门主课《计算机组成原理》和《Java 语言程序设计》。”

“这两门课程都非常有用，你哥的公司里，最主要的软件就是 Java 写的，这下你有用武之地了。”一路上闲聊了很多，乌云越来越黑，虽然还是中午时分，路上的车已经开了大灯，一阵阵的大风吹着路边的尘土和塑料袋漫天飞扬。偶尔有两颗顽皮的雨滴从空中落下，落在挡风玻璃上，积攒好久，我才开一下雨刷把它们擦掉。

一个多小时后回到公司，潘新已经在公司等着我们吃午饭了。在见到潘新之后，潘闻像变了一个似的，在吃饭的时候，潘新只和我在说话，他的弟弟对他来说就像空气一样，视而不见。

这可能是我吃的最压抑的一顿饭了，期间我数次想把潘闻拉到对话中，潘闻仅回复一两个词，就又躲闪了出去。饭后，一起回到了公司。潘闻在门口远离他哥的地方找了一个位置坐下，变戏法似的拿出一本英语单词背了起来。

天越来越黑，风越来越大，大风吹着办公室的窗户在颤抖。

突然之间，潘新平静的问了一句：“你考试考的怎么样？”我看到潘闻的身体颤抖了一下，半个屁股悬空在椅子上，嘴半张着，似乎那句回答像航班因为天气晚点了一样，迟迟不肯出来。见到此景，我有些心疼潘闻，又想转移话题，就说了一句：“好像要下雨了。”

老天爷仿佛听见了这句话，以一声炸雷作为回应，天被这个炸雷炸出了一个大洞，水冲了出来。雨越下越大，像决堤的大坝，水已经分不出你我，挤着一起浇了下来。乌云中的墨汁似乎把雨水染黑，雨越下，天越黑。透过玻璃，只能看到红色的尾灯连成一条路的模样。

兄弟俩仿佛没有被这瓢泼大雨所打扰，迟到的回答还是潘闻的嘴里讲了出来：“没太考好。《Java 程序设计》考的马马虎虎，《计算机组成原理》没通过。”

相比于潘新的愤怒，窗外下的就像毛毛雨。潘新在咒骂着他的弟弟，数落着他弟弟的不是，我在那里感到手足无措，我被那一幕吓呆了，只听到潘新怒吼着让他的弟弟滚。潘闻默默的走了出去。我意识到，他没地方可去，就追了出去。

我说：“真遗憾事情变成这个样子。”

突然间，潘闻笑了，露出一口白牙：“你不必担心，我了解我哥，他已经不生气了，只要他骂了我，这些天就没事了。”

“真的么？”

“当然是真的，这么多年了，我早就摸清他的脾气了。他现在知道我的《计算机组成原理》没考过，过几天他肯定会让我写 Java 的项目，在路上你不是说公司的软件是用 Java 写的么？”

“是的，幸好你的 Java 通过了。”

“我说了我通过了么？我只是说我的 Java 考的马马虎虎，我可没说我通过了。”

“你的意思是……”

“是的，我的 Java 也没通过，两门主课都没通过，只有那些副课通过了。你一定得教教我，否则我下学期补考也不一定能通过。”

雷阵雨就是雷阵雨，不一会儿，天所若无其事的晴朗，幸好这钢筋水泥修成的城市排水系统出奇的不好，保留着下过雨的证据。

4 变量

Contents

> 人生若只如初见，何事秋风悲画扇。
>
> 等闲变却故人心，却道故人心易变。

在这个宇宙中，易变的可不止人心，而是一切。留不住时间，于是年龄会变。保不住新鲜，于是爱人会变。纹身店里，热恋时，胸前纹“一生最爱某某某”有多爱，分手后，用激光清理这些爱的“垃圾”，就有多恨。有办法么？有倒是有的，就是把“一生最爱”纹在身上，后面的“某某某”，用纹身贴纸即可。“一生最爱”是常量，而这个“某某某”就是本章要讲的变量。

提示

当我把纹身的故事讲给我的同事小潘的时候，我觉得这么有趣的故事，他应该会觉得我很风趣。

结果他说：“栋哥，你知道大力神杯么？”

我回复他说：“知道了，怎么了？”

他说：“每届世界杯冠军都会在上面刻上自己的名字，为什么要洗呢？别人在胸前纹身，你为什么不可以把你的名字纹在她背后呢？”

在编程中，变量有很多条条框框，比如，要有个名字，叫变量名。在软件开发领域很有影响力的 Martin Fowler，敏捷软件开发的大神，在 2009 年 7 月 14 日曾经发过这样一条推，内容如下：There are only two hard things in Computer Science: cache invalidation and naming thins —— Phil Karlton。这句话翻译成中文是：“在计算机科学只有两件最难的事，缓存失效和变量命名。– Phil Karlton”

起初我觉得 Phil Karlton 的这句话是玩笑，但是后来想了一下，并不是玩笑。在编程中我们可没有那么多玩笑，给变量命名真的很难。

在构建代码的过程中，最基本的构建活动是如何使用变量。要理解变量，一要像门卫一样问变量三个问题：“你是谁？”“从哪里来？”“到哪里去？”。

在《西游记》的第八十二回《姹女求阳元神护道》这一章里，有这样一段：

> 好呆子，把钉钯撒在腰里，下山凹，摇身一变，变做个黑胖和尚，摇摇摆摆走近怪前，深深唱个大喏道：“奶奶，贫僧稽首了。”那两个喜道：“这个和尚却好，会唱个喏儿，又会称道一声儿。”问道：“长老，那里来的？”八戒道：“那里来的。”又问：“那里去的？”又道：“那里去的。”又问：“你叫做甚么名字？”又答道：“我叫做甚么名字。”

这段描写里，八戒的回答相当于什么都没有回答，他既没有回答他从哪里来，又没有回答他向哪里去，又没有回答他是什么名字。仔细想来，我们很多时候都是八戒，好像是明白了，实际上什么都没说，什么也不知道。在学编程的时候，我们不能就这么糊弄过去，一定要搞清楚变量的来龙去脉。

4.1 变量的历史

在编程的最初，根本没有变量，也就不要说变量名了。最早期的电脑只解决特定的问题，不能编程。甚至，最早的电脑编程都不是我们现在见到的编程。

我在电台里讲过世界上最早的计算机之一 ENIAC，对这台电脑编程的方法是重新布线，每次编程就像重新装修房子改水电暖一样复杂。可能因为女人天生心灵手巧的原因吧，负责这项工作的人最初都是女程序员。有一篇名为《When Computers Were Women》文章介绍了当初给 ENIAC“编程” 的女程序员。对 ENIAC 贡献最多的六位女程序员，在 1997 年入选国际科技名人堂。在 CNN 拍的一个名为《Rediscovering WWII’s female ‘computers’》中也讲述了这些最初的女性程序员群体。

这篇《When Computers Were Women》文章的 pdf 文件我也放在xueban.app这个社区中了。

在当年的 ENIAC 上，“写程序” 意味着花好几天时间来手工更改线路。下图就是当年的工作场景。

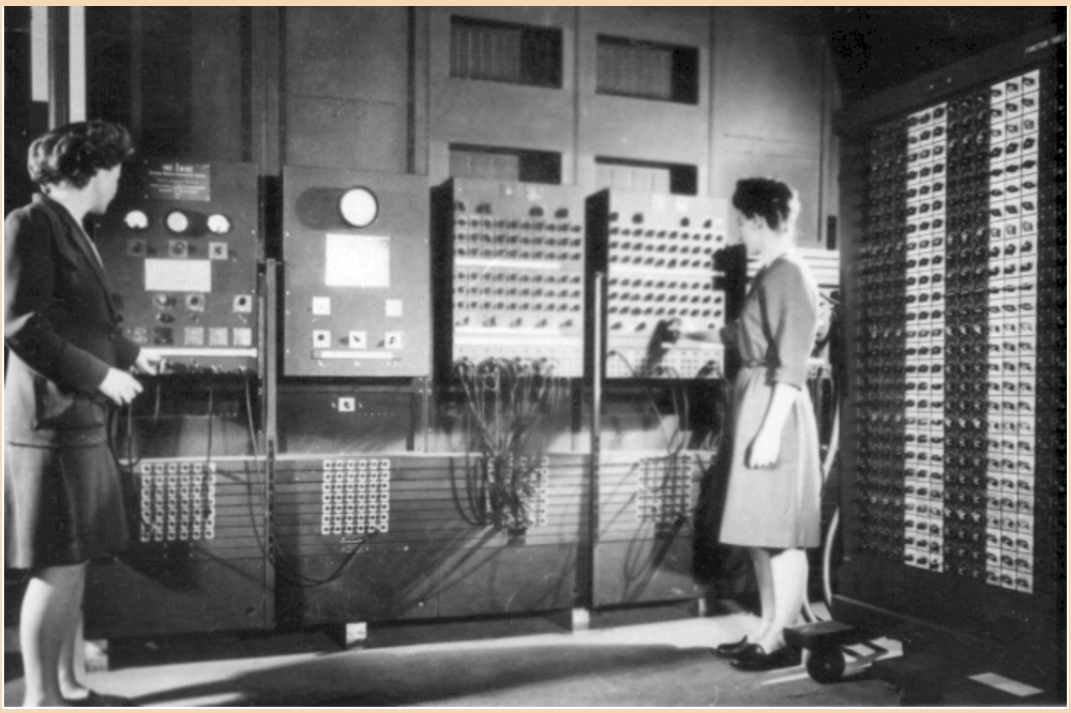

图 4.1: 两名程序员在操作 ENIAC 电脑

这样 “编程” 很麻烦，后来在冯 · 诺伊曼的指导下，Adele Goldstine 写了一段程序，让 ENIAC 拥有了存储程序的功能，让计算机改变功能从几天的时间缩短为几小时，付出的代价是运行的效率变慢。1949 年，冯 · 诺伊曼的建议和规划下，制造出了可以编程的计算机 EDSAC，输入程序和数据使用的媒介是纸带。当改变计算机的功能时，不再需要重新连接电缆，只要需要不断读取纸带上的数据就可以。这样一来，变更计算机上的程序变得更加简单方便了。

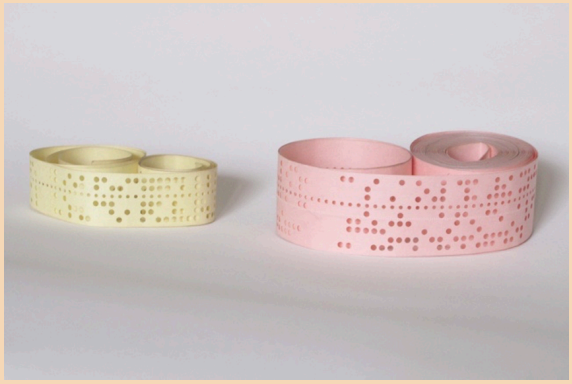

图 4.2: 当初编程用的 5 孔和 8 孔纸带

通过纸带输入程序看起来很神秘，实际上理解了汇编语言以后，并没有什么难理解的。

> 如果本书的读者对汇编语言不了解，这一段可以跳过。对 Java 编程来说，理解汇编语言并没有太大的帮助。如果你是编程的爱好者，推荐深入的学习。我特别想把事情的原理搞清楚。

国外有很多的爱好者，已经借助目前的硬件（比如树莓派），做出了当年的纸带仿制品。甚至当年的 ENIAC 电脑，也早已经用现代的技术做了复制。推荐大家在网上搜索 “ENIAC on a Chip”，可以找到不少 ENIAC 用芯片做的复制品。当年重达 27 吨的电脑，如今可以被一个学生完全复原在一个只有 8 毫米的芯片上。

再强调一次，如果没有汇编语言的经验，可以跳过这一节，或者就把这一节当成故事来看好了。早在 1949 年的 EDSAC 电脑上，已经有了变量的概念。EDSAC 电脑也有自己的模拟器，如果有人有兴趣，可以的网上搜索，或者在脚注里的链接去实际何体验一下这个与共和国同龄的电脑。在剑桥大学提供的一份名为《EDSAC Initial Orders and Squares Program》的文档，这这份文档中，已经出现了变量，那时候的变量，更像是帮助记忆的符号。

> **注释**
>
> 《EDSAC Initial Orders and Squares Program》这份文档我也放在xueban.app社区中了。

每款 CPU 都有自己的机器语言，只有机器语言才能在 CPU 上运行，但是机器语言对人类实在太不友好了，没人会用 0 和 1 来写软件。我以计算机系的流行的 X86 汇编举例，要执行这样一条命令：将一个 8 位的值移动到寄存器 AL 中，AL 寄存器的地址是 000，这个 8 位的值是十进制 97，用机器语言是这样写的：

```
10110000 01100001
```

其中后面的 01100001 是 97 这个数字，前面的那 8 位二进制要分成两部分，10110 在机器语言中代表“移动”，随后的三位 000 是 AL 寄存器的地址，所以 10110000 这一串的意思是“将一个值移动到 AL 寄存器中”。用这个太麻烦了，一不小心可能连 0 和 1 都写错了，所以，可以用十六进制稍微简化一下：

```
B0 61
```

这比用二进制好写一点，但是仍然没有太大改观。于是，人们想起了汇编语言，用 mov 来代替 10110，用 AL 寄存器的名字来代替这个寄存器的地址 000，至于 97 这个值，就用十六进制写吧，于是汇编就有了一点可读性：

```
MOV AL, 61h
```

用 AL 代替 000 这个地址，意义重大！套用第一个登上月球的阿姆斯的一句话：“这是汇编语言的一小步，却是编程历史上的一大步”，从此，有了变量以后，人们可以不用再记内存真实的物理地址了。

随后的编程语言，都采用了变量的形式。随着编程语言的进化，变量有了更多的意义。从机器语言到汇编语言，变量主要是用名字来代替数据的存储地址，使程序更容易编写，更容易阅读，更容易维护。后来像 Java，C/C++ 语言，变量已经不仅仅是存储地址的名字，变量要从更多的维度来考虑，**除了名字，还要从地址、数值、类型、作用域和生存周期这六个方面来考虑**。接下来，我就从这六个方面来研究 Java 的变量。

4.2 变量的命名

在 Java 编程语言中，正确地命名变量是至关重要的。一个好的变量名不仅能提高代码的清晰度，还能传达出有关变量的重要信息。例如，遵循命名规范的标识符能让读者一眼看出它是一个变量、常量、包，还是类。这样的命名实践对于代码的理解和后期维护都是非常有益的。Oracle 官方网站上就有一套推荐的命名规范，对于想要深入了解这一主题的开发者来说，搜索 “Naming Conventions” 以访问相关文档是一个很好的起点。

在 Java 编程语言中，除了字母（A~Z, a~z）、下划线（_）和美元符号（$）之外，任何 Unicode 字符都可以合法地用于变量命名，值得注意的是，数字不能作为变量名的首字符。这意味着在 Java 中，理论上可以使用各种非传统且独特的变量名。然而，官方还是推荐遵循以小写字母开头的变量命名习惯。这不仅符合大多数程序员的编码规范，而且有利于使用正则表达式对变量名进行统一处理和识别，从而提高代码的可读性和维护性。接下来，我们先来运行一段简单的代码。

```
public class VariableNamingExample {
    public static void main(String[] args) {
        // 分别声明两个整数类型的变量，名为 myNumber 和 mynumber
        int myNumber = 10;
        int mynumber = 11;

        // 声明一个字符串类型的变量，名为 myString
        String myString = "Hello, World!";

        // 声明一个布尔类型的变量，名为 isDone
        boolean isDone = true;

        //声明并初始化为 10 的整数类型变量 count
        int count = 10;

        System.out.println("The value of myNumber is: "
                                                + myNumber);
        System.out.println("The value of mynumber is: "
                                                + mynumber);
        System.out.println("The value of myString is: "
                                                + myString);
        System.out.println("The value of isDone is: "
                                                + isDone);
        System.out.println("The value of count is: "
                                                + count);
    }
}
```

上面的代码展示了变量名，该例子运行之后，得到的结果如下：

```
The value of myNumber is: 10
The value of mynumber is: 11
The value of myString is: Hello, World!
The value of isDone is: true
The value of count is: 10
```

在 Java 编程语言中，标识符是一种用于命名各种编程元素的字符序列，这些元素不仅涵盖了变量，还延伸至许多其他重要的结构。比如，标识符还包括以后要讲的类名、接口

名、方法名、包名及常量等众多编程标识。什么字符可以用作合法的标识符呢？这些字符包括字母（无论大小写）、数字、下划线（_）以及美元符号（$）。但是对变量来说，还是尽量遵守如下的约定：

1. 使用字母作为变量名的首字符更符合人们的阅读习惯，更容易理解和识别。使用下划线（_）作为首字符虽然合法但是不推荐。使用美元符号（$）作为首字符虽然也合法，但是容易跟一些框架冲突。
2. Java 编程语言具有大小写敏感特性，这意味着在变量名的识别与处理过程中，系统会严格区分大小写字母，即同一名称但大小写不同的变量会被视为两个独立的标识符实体。比如代码中的 myNumber 和 mynumber 就是两个不同的变量。
3. 采用具有明确含义的名字是编程中的良好实践，而驼峰命名法（camelCase）作为其中一种流行的标准格式，被广泛应用于 Java 开发中。在驼峰命名法下，单词首字母小写且后续每个单词首字母大写的书写规则，能够清晰地将多个词合成一个连贯且易于理解的标识符名称，从而有效增强代码的一致性和可辨识度。比如代码中的 isDone。
4. 在 Java 编程语言中，保留字（Reserved Words）是预先赋予特定含义的关键词，它们被系统内部使用来表示特殊的语法规则或执行特定的功能。由于这些保留字具有固定的用途和功能，因此不能将它们用作变量名、类名或其他标识符名称，否则会引发编译错误。例如，诸如 if、else、class、public 等均属于 Java 的保留字，不能作为自定义的编程元素标识符使用。

4.3 变量的数据类型

在 Java 编程语言中，必须为每个变量明确指定数据类型。这是由于 Java 属于静态类型语言，意味着所有变量的数据类型必须在编译时确定。这种设计使编译器能够提前识别类型错误，从而增强代码的安全性和稳定性。

在 Java 中，为变量指定类型是通过在声明变量时首先写出数据类型，紧接着是变量名，最后是赋值或初始化。Java 10 及更新版本引入了局部变量类型推断，这允许开发者在某些情况下使用 var 关键字声明变量，从而简化代码。但即使使用 var，编译器在背后仍然会确定变量的确切类型，所以这并不是真正的动态类型。以下展示了如何为变量指定类型。在这个例子中，height 被明确声明为 double 类型，而 score 则利用 var 关键字让编译器推断其类型。这种做法虽然提高了代码的简洁性，但应谨慎使用以保持代码的清晰度。

```
// 声明并初始化一个 double 类型的变量
double height = 1.0;
// Java 10 及以上版本支持使用 var 关键字进行局部变量类型推断
// 编译器会根据右侧的初始值自动推断出 score 为 int 类型
```

```
var score = 85;
```

在 Java 编程语言中，遵循前置类型标注的语法约定，即变量的类型声明必须在变量名之前。这种方式与 C 等较早的语言一致，它们都采用了类似的前置类型定义规则。与之相对，一些现代编程语言如 Swift 和 Kotlin 则更偏好后置类型标注，也就是先声明变量名，然后再指定其数据类型。两种类型标注方式的选择更多取决于习惯和个人偏好。有些人认为两者在代码可读性上存在差异，但实际上，现代集成开发环境 (IDE) 会通过颜色编码清晰地区分类型和变量名，因此在阅读代码时并不会造成实质性的影响。

接下来，我们将深入解析 Java 变量的数据类型体系。对于初学者而言，可能会感到困惑的是，Java 同时提供了基本数据类型 (primitive data type，也称为原始类型) 和引用数据类型 (reference data type)，这两套系统看似功能有所重叠。那么，为什么要引入这两套数据类型体系呢？官方解释指出，Java 之所以设计两套数据类型系统，是为了满足不同的编程需求。一方面，基本数据类型适用于简单的数值计算与存储，具有更高的性能效率；另一方面，引用数据类型则更侧重于提供面向对象编程的支持，从而实现更灵活、更强大的数据结构和操作。

在深入探讨具体细节之前，我想先分享一点我的个人看法。我认为，尽管 Java 提供了两套看似功能相似的数据类型体系，但其带来的优势可能并没有那么显著，而随之而来的问题却不容忽视。从性能角度来看，这样的设计在某些场景下提升了效率，但如果追求极致性能，我们或许更应该考虑使用 C 语言或借助编译器优化等手段。然而，在我看来，这一设计的主要不足在于它增加了学习的难度，容易使程序员在实际开发过程中混淆两种类型的使用，从而导致错误率上升。

为了解决这种类型混淆的问题并简化编程体验，Java 5 版本引入了一项关键特性：自动装箱（autoboxing）与拆箱（unboxing）机制。该机制允许程序在基本数据类型与对应的包装类之间自动进行转换，表面上减少了潜在的代码错误。然而，这也并非没有代价：一方面，自动装箱与拆箱操作本身并不高效；另一方面，频繁地进行自动装箱和拆箱操作会增加额外的对象创建和垃圾回收压力，特别是在循环或大量数据处理场景下，可能会对性能产生负面影响。

尽管存在这些问题，但对于 Java 的学习者而言，理解并掌握这些优缺点仍然至关重要。值得一提的是，后续出现的语言如 C# 和 Kotlin 在发展过程中也借鉴了 Java 的自动装箱与拆箱机制，并在此基础上进行了各自的设计改进和优化。因此，掌握 Java 的基础知识，对于学习 C# 和 Kotlin 这两门同样流行的编程语言也会有所帮助，让学习过程更加轻松。

基本数据类型直接存储固定的、不可分割的基本值，例如数字和字符。具体而言，Java 共有 8 种基本数据类型，包括 4 种整数类型 (int、byte、short、long)、2 种浮点数类型 (float、double)、1 种字符类型 (char) 以及布尔逻辑类型 (boolean)。这些类型的变量在内存中占用固定大小的空间。

与基本数据类型相比，引用数据类型更加复杂和灵活。它们并不直接存储具体的值，而是存储指向内存中其他位置的引用或地址，实际数据则存放在该地址处。这类数据类型涵盖了类 (class)、接口 (interface)、数组 (array) 以及其他复合结构。值得一提的是，有一种特殊的类被称为包装类，它们为每种基本数据类型提供了一个对应的引用数据类型，其对应关系如下表所示。

基本数据类型	包装类
boolean	Boolean
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Character

通过使用包装类，我们可以将基本数据类型的值转换为对象，从而在需要对象的地方使用它们，例如对象实例 (可以包含多个属性和方法的实体)、列表 (List)、集合 (Set)、映射 (Map) 以及更高级的数据结构如树 (Tree) 和图 (Graph) 等。这个转换过程由 Java 提供的自动装箱和拆箱机制自动完成，使得基本数据类型与它们的包装类之间能够无缝衔接。

尽管包装类为基本数据类型提供了对象化的表现形式和更丰富的功能扩展，但我们不能忽视基本数据类型作为 Java 语言基石的重要性。基本数据类型的变量存储效率高，占用内存少，这对于性能敏感的应用场景至关重要。此外，对基本数据类型的操作通常更为直接和高效，无需涉及装箱和拆箱过程带来的额外开销。因此，理解和掌握基本数据类型对于编写高效的 Java 代码至关重要。接下来，让我们深入学习基本数据类型。

4.3.1 基本数据类型

在 Java 编程语言中，基本数据类型是构建程序逻辑的基石。它们是最底层、最基础的数据结构，用于直接存储不可分割的原始值，而非引用其他对象或数据结构。

这些基本数据类型的变量在内存中占用固定大小的空间，并且在声明时就确定了其存储的内容类型和范围，访问速度快。因此，在处理大量简单数据或者对性能要求较高的场景中，选择合适的基本数据类型对于程序的效率和性能至关重要。

前文已经提到，Java 中共有 8 种基本数据类型，对这 8 种基本数据类型，可以分为以下 4 类来学习：整数类型、浮点数类型、字符类型以及布尔类型。

提醒一下大家，因为我是在写书，所以要稍微的整的精确一点。实际编程中，我自己也记不住我写的这些。尤其是某种类型的最大值与最小值，一般情况下，根本不必在意 int 类型的最大值最小值是多少。

有个传闻，传闻，传闻。有人问爱因斯坦声音的速度是多少？他拒绝回答。随后他说了一句著名的话：“**大学教育的价值不在于记住很多事实，而是训练大脑会思考。**”

写书的时候，我都是查过资料的，“显得”我记住了。写完之后，我就又忘记了。总之，你只要记住凡事有个界限就可以了。整数无脑用 int，浮点数无脑用 double :)

但是，研究一下处理这些字符的历史，是真的有趣。所以，我花了大量笔墨来讲这些历史。

4.3.1.1 整数类型

整数类型包括 byte、short、int、long，分别对应不同的字节长度和数值范围。

byte 占用 1 字节，取值范围为-128 至 127

short 占用 2 字节，取值范围为-32,768 至 32,767

int 占用 4 字节，取值范围为-2^{31} 至 $2^{31}-1$

long 占用 8 字节，取值范围为-2^{63} 至 $2^{63}-1$

在进行整数运算时，尤其需要警惕潜在的溢出风险。当计算结果超出目标整数类型的取值范围时，系统将无法正确存储该结果，进而引发溢出现象。需要注意的是，Java 语言并未内置自动检测和处理整数溢出的功能，这意味着程序员在实现涉及整数运算的代码段时，务必谨慎对待可能发生的溢出问题。

接下来，我们将通过如下代码示例展示 Java 整数的溢出风险以及溢出后 Java 所采用的处理方式。

```
public class OverflowExample {
    public static void main(String[] args) {
        // Java 中 int 类型的范围是-2^31 到 2^31-1
        // 最大整数值 2147483647
        int maxInt = Integer.MAX_VALUE;
        System.out.println(" 整数能表示的最大值: " + maxInt);

        // 当尝试加上一个正值时，会发生溢出
```

```
        // 输出结果将是 -2147483648
        // 因为发生溢出，数值从整数的最大值回到了最小值
        int overflowedValue = maxInt + 1;
        System.out.println("maxInt + 1:  "
                                    + overflowedValue);

        // 最小整数值 -2147483648
        int minInt = Integer.MIN_VALUE;
        System.out.println(" 整数能表示的最小值: " + minInt);
    }
}
```

上面的代码运行之后，输出结果如下：

```
整数能表示的最大值: 2147483647
maxInt + 1:  -2147483648
整数能表示的最小值: -2147483648
```

在这个示例程序中，我们首先获取 int 类型的上限值，并将其打印输出到终端。随后，我们尝试将这个已知的最大值加 1。由于这个操作会导致结果超出 int 类型的有效范围限制，因此系统会发生整数溢出。这意味着计算后的结果会循环回绕到 int 类型的下限，即最小可能值。

这种处理机制被称为回绕（wrap-around），指的是当数值超出某个数据类型的表示范围时，会回绕到该数据类型的最小值。例如，当一个 int 类型的变量超过其最大值时，它会“回绕”到 int 类型的最小值，就好像数字在一个圆环上循环一样。

在编程时，选择合适的整数类型对于提高代码效率和可读性至关重要。因此，我选择整数类型的策略是：在没有特殊需求或明确目的的情况下，优先选择 int 类型作为默认的整数数据类型。**接下来，我将深入剖析这一原则背后的原因。**

前文提到，Java 虚拟机 (JVM) 具备一套全面的指令集体系，涵盖了诸如算术运算、逻辑操作、比较判断、存储器管理以及控制流程等多方面的指令。这套指令集能够处理包括 int、long、float、double、boolean 和 char 在内的多种基本数据类型的运算。然而，值得注意的是，对于 byte 和 short 这样的较小整数类型，JVM 并没有特地设计单独的操作码来进行直接处理。

相反，为了简化指令集的设计结构，并提高代码执行效率，JVM 采用了隐式类型提升机制。当遇到 byte 或 short 类型的变量参与运算时，JVM 会先将这些值自动转换为 int 类型，然后再运用通用的 int 类型操作指令来执行实际操作。换句话说，虽然 byte 和 short 没

有专门的操作码，但它们在运算时会被“提升”为 int 类型，从而保证了指令集的简洁性和统一性，并在性能上实现了优化平衡。

既然在 Java 运算过程中，byte 和 short 类型的变量都会被自动提升至 int 类型进行操作，那么为什么不从一开始就只保留 int 类型呢？当然，在特定场景下，这种“一步到位”的做法并不总是最优解。例如，在对存储空间要求极为苛刻的情况下，选择 byte 或 short 能够显著地压缩数据尺寸，从而发挥关键作用。特别是在处理图像像素数据、网络传输等需要大量紧凑存储和高效传输的领域中，byte 类型和 short 类型的数据类型都有其独特的价值和必要性。

4.3.1.2 浮点数类型

在 Java 编程语言中，存在两种浮点数类型：

float：单精度浮点数类型，占用 4 字节（共计 32 位）内存，其数值范围大致从 3.4e-038 到 3.4e+038，能够提供大约 6 到 7 位的可靠有效数。由于其较小的内存占用，float 类型适用于对精度要求相对不高且需节省存储资源的应用场景。

> **注释**
>
> **3.4e-038** 表示的是 3.4 乘以 10 的负 38 次方，即小数点向左移动 38 位。这是 float 类型的最小正数值，接近于 0。3.4e+038 表示的是 3.4 乘以 10 的正 38 次方，即小数点向右移动 38 位。这是 float 类型的最大正数值。

double：双精度浮点数类型，占用 8 字节（共计 64 位）内存，数值范围大致为 1.7e-308 到 1.7e+308，并能确保大约 15 到 16 位的有效数字精度。由于更高的精度表现，double 通常被推荐为默认的浮点数类型，在不考虑特殊内存约束或精确度需求的情况下应优先使用。

> **注释**
>
> **1.7e-308** 表示的是 1.7 乘以 10 的负 308 次方，即小数点向左移动 308 位。这是 double 类型的最小正数值，接近于 0。1.7e+308 表示的是 1.7 乘以 10 的正 308 次方，即小数点向右移动 308 位。这是 double 类型的最大正数值。

这两种浮点数类型 (float 和 double) 均支持正负数表示，并且它们在实际存储时都遵循 IEEE 754 标准。这意味着它们无法准确表示所有实数，尤其是那些不能被精确转换成二进制分数的十进制数。

之所以会有精度损失的问题，是因为计算机内部采用二进制来表示数字。由于十进制小数并非都能精确转换为有限长度的二进制小数，这导致了部分十进制浮点数在计算机内只能近似表示。例如，十进制的 0.1 无法用二进制精确表示，因此在计算机中存储和计算时会产生微小的误差。

当涉及到比较两个浮点数是否相等时，由于上述精度损失的存在，直接使用等于运算符 “==” 来进行比较可能会得到意外的结果。例如，两个理论上应该相等但实际上由于浮点计算过程中产生的微小误差而变得不完全相等的浮点数，用 “==” 比较将返回 false。为了正确地比较两个浮点数是否足够接近以被认为是相等的，通常采用的方法是设定一个合理的误差范围（也称为 epsilon 值），并检查两者之差的绝对值是否小于这个误差阈值。

接下来，我们将通过示例演示浮点数存在的问题以及解决方法，代码如下所示：

```
import java.math.BigDecimal;

public class FloatingPointPrecisionDemo {
    public static void main(String[] args) {
        // 演示浮点数表示不精确
        float a = 0.0001f, b = 100000.0f;
        // 输出可能不是预期的 100000.0001
        System.out.println(" 直接加法结果： " + (a + b));

        // 使用误差范围（epsilon）进行浮点数比较
        float x = 0.1f, y = 0.10000001f;
        boolean areAlmostEqual = Math.abs(x - y) < 0.00001f;
        System.out.println(" 使用误差范围判断是否相等： "
                           + areAlmostEqual);

        // 在金融应用中建议使用 BigDecimal 类
        BigDecimal amount1 = new BigDecimal("1234.56");
        BigDecimal amount2 = new BigDecimal("789.01");

        BigDecimal sum = amount1.add(amount2);
        System.out.println(" 精确的货币计算结果： " + sum);
    }
}
```

运行之后，结果如下：

```
直接加法结果： 100000.0
使用误差范围判断是否相等： true
精确的货币计算结果： 2023.57
```

在计算机编程中，浮点数类型的累积误差是指在进行一系列浮点数运算过程中，由于

浮点数的表示方式和精度限制，导致最终结果与理论预期值之间产生的偏差。特别是对于那些不能精确表示为二进制分数的十进制数，在进行加减乘除等连续运算时，每一步计算都会引入一些微小的舍入误差。这些误差在单次运算中可能非常微小，几乎可以忽略不计，但随着计算步骤的增加，这些误差会逐渐累积起来，从而可能导致最终结果与实际期望值有较大的差异。这个问题在需要高精度计算的领域，如金融、科学计算等，尤为重要。

接下来，我将通过一个例子来解释浮点数类型的累积误差问题。

美国政府问责局（Government Accountability Office，GAO）曾发布过一份官方文件，其中披露了在 1991 年 2 月 25 日海湾战争期间发生的一起重大事件。这份报告指出，爱国者导弹防御系统遭遇了一次关键性失误，未能成功拦截一枚飞毛腿导弹，导致该导弹击中一处美军营房并造成 28 名士兵不幸丧生的悲剧。

> **注释**
>
> 这篇报告名为《Roundoff Error and the Patriot Missile》，我已经放到xueban.app上了，该报告详细讲了事故原因。

针对这一事件，专家 Robert Skeel 撰写了一份深度剖析报告。报告显示，爱国者导弹防御系统内部计时机制的设计缺陷是此次失效的核心原因。具体来说，由于计算机处理浮点数计算时无法避免微小的精度误差，该系统的计时精确度每经过 8 小时会产生约 0.0275 秒的累积偏差。截至事故发生之时，该系统已连续运行了整整 100 个小时，累计产生了大约 0.34375 秒的时间差错。

值得注意的是，飞毛腿导弹以高达 5 马赫的速度飞行，即每秒钟可以飞行约 1.7 公里。因此，这看似微不足道的 0.3 秒多的时间误差，在实际操作中却相当于让导弹飞行了约 600 米的距离，足以使其绕过原本应能拦截到的防线。正是这微小的精度损失，最终导致了这场重大灾难性后果。

爱国者导弹防御系统失误的案例，仅仅是计算机在执行浮点数运算时的常规“差错”的一个缩影。计算机在执行涉及浮点数运算的任务时，由于其二进制表示机制、精度限制以及标准化浮点数格式（如 IEEE 754）的设计特性，产生“小差错”是必然的结果。这些看似微不足道的“小差错”，提醒着我们计算机并非精确无误的工具，也激发着我们深入探究计算机如何处理数字的核心领域 —— 小数计算。

对计算机如何处理小数的历史进行深入探究，或许对实际编程没有直接的指导意义，但这段历史对于编程爱好者而言却充满了魅力。因此，我决定将关于计算机如何处理小数的历史附于下面，当作补充阅读材料，以飨那些对此怀揣好奇的读者。

4.3.2 计算机中小数处理的历史

当前，计算机科学中广泛采用的 IEEE 754 标准为大多数计算平台提供了表示带有小数部分的实数的方法。在 Java 编程语言中，我们所熟知的 float 数据类型名称源于“floating-point”这一术语，意指浮点数；而 double 类型的命名则因其精度大约是 float 类型两倍的特点而来。

提及浮点数，人们自然会联想到其对应的“定点数 (fixed-point)”。事实上，确实存在定点数这一概念，并且了解了定点数之后，对于浮点数的理解将更为轻松和直观。

接下来，我将详细介绍定点数是什么，以此作为理解浮点数的一个重要铺垫。

4.3.2.1 定点数

定点数（fixed-point numbers）是一种用于表示小数的方法，通过将数值分成整数部分和小数部分来实现。这种表示方式在嵌入式系统、DSP（数字信号处理）以及一些低功耗或性能受限的计算环境中尤为常见，因为这些环境可能不支持浮点运算，或者浮点运算的开销较高。类似地，在金融与商业领域，使用定点数可以有效的减少二进制与十进制转换的误差。

定点数通常采用固定长度的二进制位模式来编码数据，其中分配给整数和小数的位数均预先设定好。例如，在一个 16 位的定点数格式中，可以将其划分为 8 个整数位和 8 个小数位，小数点则始终固定于第 8 位（从低位数算起）。定点数的表示方法有多种，比如在数字信号处理领域中采用的 Q 格式表示法，在金融与商业领域，会用到 BCD 码的方式。接下来，我详细介绍一下 BCD（Binary-Coded Decimal，二进制编码的十进制）码的表示方法。

以下是 BCD 码的具体表示规则：它用四位二进制数精确对应十进制数的一个数字，两者之间有一一对应的转换关系，如下表所示:

十进制数字	二进制数字
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

由上面的对应表可知，每个十进制数对应 4 位二进制数字。如果要表示一位十进制数 8，则可以用 1000 来表示，如果要表示两位十进制数 10，则要分别表示 1 和 0，其中的 1 表示为 4 位二进制 0001，0 则表示为 4 位二进制 0000。可以把两个十进制数字放一个 8 位二进制中，正好一个字节中，这种把两个十进制放入一个字节的方法被称之为压缩 BCD（packed BCD）。那么十进制数字 10 用压缩 BCD 的方式表示为 00010000。

以日常生活中去超市为例，我们会经常看到商品的标价为 19.99 元、9.99 元这样的价格。在超市场景下，计价系统的小数位事先确定为两位小数，也就是精确度保留到分即可。假设我开了一家超市，要处理的账目是 ±1000 万，精度精确到分，这意味着系统可以表示如下范围的数字：-10,000,000.00 元到 +10,000,000.00 元。

当我要表示一笔 3,898,190.25 元的记录，可以从最后开始，以两位十进制数字为单位进行拆分，写成如下的方式：

3	89	81	90	25
00000011	10001001	11000001	10010000	00100101

定点数因其固有的特性，在特定应用场景下具有一些优势：其固定的数值表示形式使得存储空间需求更小，同时避免了复杂浮点运算的开销。然而，这种数据类型也存在局限性，即其数值范围和精度是预先设定且不可变的，无法根据实际需求进行灵活调整。若已知待处理的数据具有明确的数值边界，则采用定点格式的小数无疑是明智之举，它能够确保在预设范围内不会出现意外的精度误差。

在 Java 的标准库中，并未直接提供对定点数类型的内建支持，而是侧重于原生支持浮点数类型，如 float 和 double。尽管浮点数类型的计算速度可能相对较低，并且在某些情况下会出现一定的精度损失，但它们具备比定点数更高的精度水平以及更为宽广的数值表示能力。接下来，我们一同深入探讨 Java 中的浮点数类型的机制。

4.3.2.2 浮点数

在讲浮点数之前，先来讲一下如何用二进制表示小数。在计算机中，小数可以使用二进制表示法来存储和处理。二进制小数表示法基于科学计数法，将小数部分转换为二进制数，并使用科学计数法的形式来表示数值。

将十进制小数转化为二进制小数的步骤如下：

1. 将小数部分乘以 2，得到积和整数部分。
2. 将积的整数部分作为二进制数的一位，并将小数部分保留。
3. 将小数部分重复第 1 步和第 2 步，直到小数部分为 0 或达到所需的精度。

例如，将十进制小数 0.625 转化成二进制小数的过程如下：

0.625 × 2 = 1.25，整数部分为 1，小数部分为 0.25

0.25 × 2 = 0.5，整数部分为 0，小数部分为 0.5

0.5 × 2 = 1.0，整数部分为 1，小数部分为 0

因此，0.625 的二进制表示为 0.101。

那如何将二进制小数转换为十进制小数呢？其方法与将整数从十进制转换为二进制的方法类似，只不过需要将每一位的权值改为 2-n，其中 n 表示该位在二进制数中的位数。以下是将二进制小数转换为十进制小数的方法：

1. 将二进制小数的整数部分和小数部分分别转换为十进制数。
2. 对于小数部分，将每一位的权值改为 2-n，其中 n 表示该位在二进制数中的位数。
3. 将小数部分的每一位乘以对应的权值，并将所有结果相加，得到十进制小数的值。

举个例子，以 110.1101 这个带有小数点的二进制数为例，

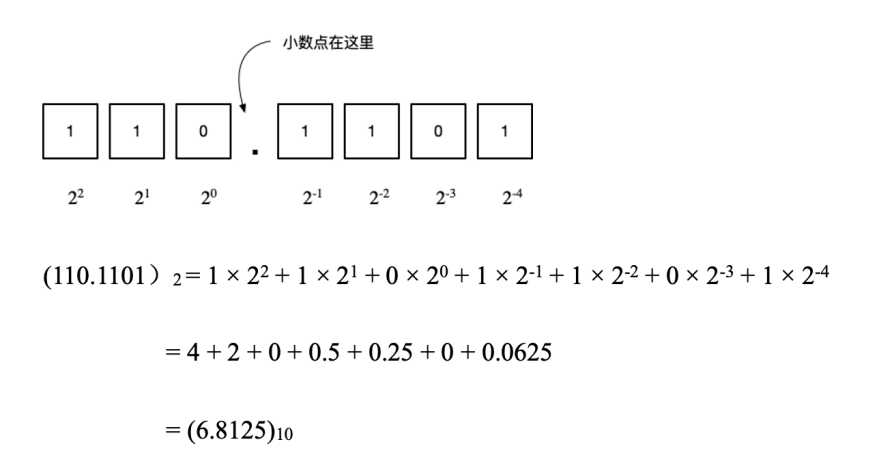

图 4.3: 二进制转为十进制

当我们在表示一个很大的数的时候，比如美国国债是 $25,000,000,000,000，数学上会采用科学计数法来计数，写成这样：2.5×10^{13}。在科学计数法中，我们称 2.5 为首数，10 的幂是 13，我们称 13 为指数。在计算机中也类似，只是给首数改了个名字叫有效数，指数没改名字，还是叫指数。

当我们看到 2.5 x 10^{13} 的时候，知道把小数点向右移动 13 位，当看到 4.5×10^{-3} 的时候，把小数点向左移动 3 位成为 0.00045。因此只要知道了指数，就知道怎么浮动小数点了，指数是正的就向右移，指数是负的就向左移。这就是浮点数的基本原理了，小数点浮动的方向和浮动多少，都由指数决定。

浮点数还有一个规定叫写法的规范化，什么叫规范化呢？还是以美国的国债为例，可以写成 2.5 x 10^{13}，虽然写成 25 x 10^{12}、250 x 10^{11} 都可以理解，但是后面这两种不规范。对

十进制来说，规范化是规定有效数只能介于 0 和 10 之间。对二进制浮点数来说，规范化的写法第一位只能是 1。以 110.1101 为例，规范化的二进制浮点数只能是 1.101101×2^2 这一种形式，其余的形式都是不规范的。

因为规范化的二进制浮点数第一位永远只能是 1，所以根本就没必要存储这一位。因此，对实数来说，IEEE 754 规定，只需要存储三个信息即可：代表正负的符号位、代表如何浮动小数点位置的指数和有效值。以 IEEE 定义的 4 个字节的 float 单精度格式为例。

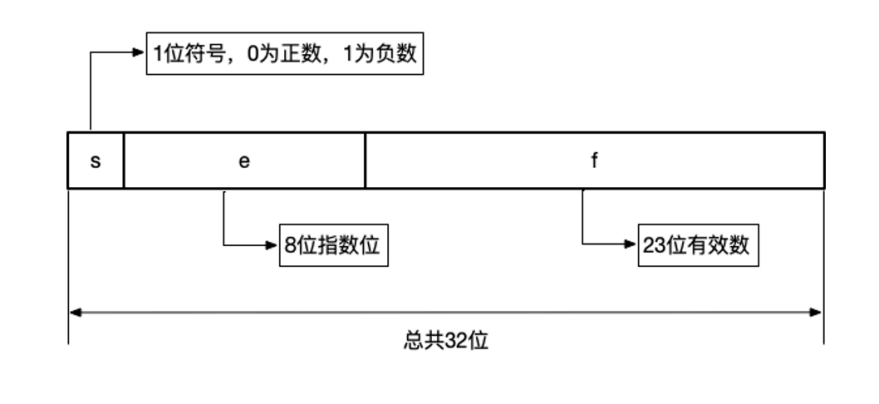

图 4.4: IEEE 754 规定的单精度浮点数 float

长度为 1 位的符号位 s 代表正数还是负数，0 表示正数，1 表示负数。

随后的 8 位是指数位 e，8 位可以代表 0 到 255 的数值。在此，我们取 1 到 254，0 和 255 留作他用。小数点有可能向左浮动，也有可能向右移动，我们规定这个数减去 127，正好可以左移或者右移，此处的 127 我们称之为偏移量。

接下来是 23 位有效数 f，前面已经讲过，有效数的第一位永远是 1，所以不用存这一位，只需要在计算的时候补上这个 1 即可。综上，这个特定的数可以用这个公式来计算：

$$(-1)^s \times 1.f \times 2^{e-127}$$

还有几种特殊情况要单独考虑，比如指数位都是 0 或者都是 1 的时候，这也是为什么指数位要保留了 0 和 255 两个数用作其它用途的原因，表示无穷大（Positive Infinity），无穷小（Negative Infinity）或者不合法 NaN（Not a Number），比如当 0/0 时或者对一个负数进行平方根操作会返回不合法。这三种情况在实际应用中并不多见，Java 为其提供了相应的常量分别是 Float.POSITIVE_INFINITY、Float.NEGATIVE_INFINITY 和 Float.NaN（相应的 double 类型也有这三个常量）。

接下来我们来看看单精度浮点的表示范围，理解了原理，套用上面的公 $(-1)^s \times 1.f \times 2^{e-127}$，我们知道绝对值最小的正负二进制是：

$$(-1)^s \times (1.00000000000000000000000)_2 \times 2^{-126}$$

指数位为 $(1)_{10}$，也就是让小数点移动到最左侧，有效值是一个 1，后面跟 23 个 0。这个数用十进制表示，近似的等于 ś1.17549435 $\times 10^{-38}$。最大表示的数是:

$$(-1)^s \times (1.11111111111111111111111)_2 \times 2^{-127}$$

指数位为 $(254)_{10}$，让小数点移动到最右边。有效值为 24 个 1。这个数如果用十进制表示的话，近似的等于 ±3.40282347 $\times 10^{38}$。所以，只要是用 IEEE 754 规定的浮点数，float 类型的取值范围是 ±1.17549435 $\times 10^{-38}$ 到 ±3.40282347 $\times 10^{38}$。

以上就是单精度浮点数的原理，双精度浮点数的原理跟单精度浮点数类似，不再赘述。

4.3.3 字符类型

在 Java 语言中，字符类型作为一种基础数据类型占据着重要地位，它专门用于处理单个字符信息。每个 char 变量占用两个字节（16 位）的空间，并采用 Unicode 字符集进行编码处理，确保能够表示全球范围内的多种语言和特殊符号。

接下来，我们将通过示例详细展示字符类型的常见用法，包括声明、初始化、比较、类型转换以及在字符串处理中的应用，

```
public class CharExample {
    public static void main(String[] args) {
        // 声明并初始化一个字符类型的变量
        char myChar = 'D';

        // 输出字符
        System.out.println("myChar 这个字符是: " + myChar);

        // 使用字符类型进行比较操作
        char anotherChar = 'B';
        if (myChar == anotherChar) {
            System.out.println(" 两个字符相同.");
        } else {
            System.out.println(" 两个字符不同");
        }

        // 将 ASCII 码转换为字符类型
        // ASCII 值对应字符'D'
        int asciiValue = 68;
        char charFromAscii = (char) asciiValue;
```

```
        System.out.println("ASCII 为 68 的字符为: "
                                + charFromAscii);

        // 使用字符类型进行字符串处理
        String name = "Alice";
        char firstLetter = name.charAt(0);
        System.out.println(" 字符串的第一个符为: "
                                        + firstLetter);
    }
}
```

这段代码展示了如何声明和初始化 char 变量，并进行简单的字符比较；同时演示了如何将 Unicode 码（这里是 ASCII 码）转换为字符类型，以及如何从字符串中提取首个字符。

在计算机科学领域，正确理解和处理字符是一项充满挑战却又饶有趣味的事情，在编程发展史上经历了相当长的一段时间。为了让有兴趣的读者深入了解这段历史脉络，我特地撰写了关于计算机中字符处理发展的历史。

计算机中字符处理的历史

在人类文明的长河中，信息传递与记录方式的发展始终伴随着科技进步的步伐。自古至今，从刻划在泥板上的楔形文字、镌刻在龟甲和兽骨上的甲骨文、书写在竹简上的汉字到印刷术的诞生，字符的记载形式不断演变。随着 20 世纪计算机技术的崛起，如何让冰冷的机器理解并处理人类语言中的字符，成为了一项至关重要的挑战。早期的计算机系统受限于硬件条件和编码体系的不统一，对字符的识别和处理存在着显著的局限性。

随着时间的推移，从博多码到 ASCII 码，再到 Unicode 的提出，每一步都是人类智慧在解决字符编码问题上的一次飞跃。这些编码方案不仅反映了计算机科学的进步，也揭示了全球化时代下文化交融与交流的需求。这段历史见证了字符编码从单一语言向多元文化的包容性转变，同时也为现代计算机系统提供了坚实的语言基础支持。在这段旅程中，我们将会探讨一系列里程碑式的字符编码标准及其背后的故事，从而深入理解计算机是如何逐步“学会”理解和处理各种字符的。

博多码

在看电影时，我们有时会目睹演员用手电筒发出闪烁的灯光以传达 SOS 求救信号的情节，这种编码方式是源自于约 1836 年由塞缪尔 · 摩尔斯发明的摩尔斯码。作为一种

历史悠久且曾占据主导地位的通信编码体系，摩尔斯码主要用于早期的电报传输，在二十世纪中叶之前扮演了至关重要的角色，直至随着技术进步和新型通信手段的涌现而逐渐淡出主流舞台。

博多码是在摩尔斯码之后出现的，是在 1870 年由法国工程师爱米尔 · 博多（Emile Baudot）发明的一种二进制编码体系。该编码系统巧妙地运用了 5 位二进制数来表示单个字符，从而理论上能够表达出 2 的 5 次方即 32 种不同的组合。为了扩展字符集，博多码引入了两种工作状态：字母状态和数字状态，使得在有限的 32 种组合内能够表示更多的字符。在字母模式下，这套编码被用来表示字母及特定符号；而在数字模式时，则用于呈现数字和其他相关符号。

博多码自发明后得到了广泛应用，尤其是在电报通信与早期电传打字机技术中发挥了重要作用。尽管随着时间的推移，它已被更先进的编码方式所取代，但博多码的一项关键设计遗产至今仍深深地影响着我们每一个人的日常生活——即键盘上用于切换字母与数字状态的 Shift 键。这一设计延续至今，已经成为了现代计算机输入设备不可或缺的一部分。

历经以电传打字机为代表的电传通信网络二十年的发展与积淀，历史迈入了全新的计算机时代。1949 年，具有里程碑意义的 EDSAC 计算机应运而生，这款计算机在字符编码上采用了名为 EDSAC 编码的方案。这一编码体系深受博多码的影响，同样采取 5 位二进制形式来表示一个字符，并沿用了博多码中用于切换字母与数字状态的关键设计——Shift 键。

EDSAC 计算机诞生后的十年间，计算机技术蓬勃发展，各类新型计算机如雨后春笋般涌现，然而这些计算机在字符编码方面并未形成统一标准，每种型号的计算机均有各自独特的编码体系。这一现象导致了不同型号计算机之间数据交流的严重障碍，它们无法直接识别和处理彼此的数据信息，必须通过繁琐且容易出错的编码转换过程才能实现有效的数据交互，极大地限制了当时计算机系统的互操作性和数据共享效率。

ASCII 码

鉴于当时计算机领域对于统一字符编码标准的迫切需求，IBM 的一位名叫 Bob Bemer 的员工于 1961 年向美国国家标准学会（ANSI）提出了 ASCII 编码方案。ASCII 全称为 American Standard Code for Information Interchange，即信息交换的美国标准代码，这一名称清晰地表达了其旨在消除不同计算机系统间信息交流障碍的初衷。

在提出后的两年内，ANSI 采纳并发布了首个 ASCII 编码标准版本。随着技术发展和应用需求的变化，ASCII 编码历经了多个版本的修订和完善。至 1972 年，ASCII 编码已有了一个较为成熟的版本，它为后续的信息科技领域奠定了基础性的文本编码规范，并对全球范围内的计算机通信产生了深远的影响。下图是 1972 年的 ASCII 版本。

USASCII code chart

Bits b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

观察上图所示的 ASCII 编码表不难发现，其主要涵盖了英文字符集以及一系列控制字符。ASCII 编码体系采用 7 位二进制数值对单个字符进行编码，基于这一特性，理论上最多能够表达 128 种不同的字符形态。这些字符囊括了从 0 到 9 的阿拉伯数字、全部 26 个英文字母（包括大写和小写）、各种标点符号以及其他一些具有特定功能的控制字符，共同构成了计算机处理文本信息的基础字符集合。

在 ASCII 编码体系中，每个字符均对应着一个独一无二的数字标识，这些编码可被计算机系统有效地识别和处理。例如，大写字母”A” 的 ASCII 编码为 65（其二进制表示形式为 01000001），小写字母”a” 的编码则为 97（对应的二进制码为 01100001），数字” 0” 的编码数值为 48（以二进制显示为 00110000），而标点符号”!“的编码是 33（其二进制表达为 00100001）。尽管 ASCII 编码所能涵盖的字符集相对有限，但其作为计算机编码的基础原理，在文本处理、网络通信等众多领域中依然发挥着不可或缺的作用。

随着计算机技术的发展以及全球范围内不同语言和符号需求的增长，单一的 ASCII 编码已无法满足全部需要。因此，人们扩展了 ASCII 编码体系，发展出了如 ASCII 扩展字符集 1、2 和 3 等多种扩充方案。同时，针对特定语言环境下的字符集，诸如简体中文采用了 GB2312 编码标准，繁体中文则依赖于 Big5 编码，俄语则采用 KOI8 编码机制等。此类编码方式各异，反映出各地区对字符处理的不同需求和复杂性。

Unicode 码

为了应对全球字符编码的挑战与需求，Unicode 方案应运而生，其别称包括统一码、万国码、国际码和单一码等，这些名称无一不揭示了 Unicode 所承载的伟大愿景——旨在创建一种能够涵盖全世界所有字符的统一编码体系。从这些多元化的翻译中，我们得以窥见 Unicode 背后宏大的理想与决心，即实现一个能全面囊括各地区语言文字及特殊符号的普适性编码标准。

1987 年，一个雄心勃勃的计划崭露头角，该计划由施乐公司（Xerox）的 Joe Becker、苹果公司（Apple）的 Lee Collins 以及 Mark Davis 共同发起。他们的愿景是创建一个能够全面覆盖全球主流字符集的统一编码体系。历经精心筹备后，于 1988 年这一划时代的 Unicode 标准应运而生，后被人称为 “Unicode 88”。

提示

1988 年的 Unicode 文档

1988 年发布的 Unicode 文档的 PDF 文件我已经放在 xueban.app 社区中，但是细节实在是太复杂，我只读了一点。之所以叫 Unicode，是因为想让这个编码 Unique，Unified 和 Universal。

在这部具有里程碑意义的 1988 年发布的文档中明确阐述了 Unicode 的核心目标：“旨在以一种切实可行且稳定可靠的方式满足全球范围内对于文本处理的需求。简而言之，Unicode 可以被视为 ASCII 码的宽广扩展版，其字符容量已从最初的范围拓展至 16 位，从而有能力容纳更丰富多元的语言和符号。”

让我们进行一个简单的计算，16 位二进制所能表示的最大字符数量为 $2^{16} = 65{,}536$ 个，其编码范围从 U+0000 一直到 U+FFFF。在这 65,536 个字符空间中，涵盖了全球绝大多数主流语言的字符集，诸如拉丁字母、希腊字母、西里尔字母、阿拉伯字母、希伯

来字母以及东方文字如汉字、日文平假名和片假名等。

在 Unicode 编码规范中，每一个字符都对应着一个独一无二的标识，这个标识被称为码点（Code Point）。而在 Java 编程语言中，通常采用"\u"前缀紧接着四位十六进制数的形式来表示 Unicode 码点，例如，"0041" 代表了拉丁字母"A" 的码点，而 "680B" 则表示的是汉字 "栋" 的码点。接下来，看一下下面代码的例子：

```
public class UnicodeExample1 {
    public static void main(String[] args) {

        // 输出拉丁字母"A"的 Unicode 编码形式
        char latinLetter = '\u0041';
        // 输出: 字母:  A
        System.out.println(" 字母:  " + latinLetter);

        // 输出汉字"栋"的 Unicode 编码形式
        char chineseCharacter = '\u680B';
        // 输出: 汉字: 栋
        System.out.println(" 汉字:  " + chineseCharacter);
    }
}
```

解释一下上面这段 Java 代码，这段代码主要用于演示如何在 Java 中使用 Unicode 编码表示和输出字符。

首先，代码声明了一个字符变量 latinLetter，并将其赋值为\u0041。在 Java 中，\u 开头的序列是一个 Unicode 转义序列，用于表示一个 Unicode 字符。\u0041 对应的正是 Unicode 编码中的拉丁字母"A"（对应 ASCII 编码中的 A）。

接下来，定义了另一个字符变量 chineseCharacter，并将其赋值为\u680B，这是汉字 "栋" 的 Unicode 编码。

总结来说，这段 Java 代码通过 Unicode 编码展示了如何在程序中表示和显示不同语言的字符，无论是拉丁字母还是汉字。通过 Unicode 编码，Java 能够支持全球各种语言和符号的统一表示和处理。

4.3.4 布尔类型

在 Java 编程语言中，布尔类型被设计用于表示逻辑状态，它具备两种明确且互斥的取值：true 和 false。使用关键字 boolean 声明布尔类型的变量，使得开发者能够在程序中定义和操作代表真假条件的逻辑值。

```
public class BooleanExample {
    public static void main(String[] args) {
        // 声明并初始化两个布尔变量，
        // 分别表示学生是否完成作业和是否通过考试

        boolean hasFinishedHomework = true;
        boolean passedTheExam = false;

        // 使用布尔值进行逻辑判断，并根据结果输出相应信息
        if (hasFinishedHomework && passedTheExam) {
            System.out.println(" 完成了作业并通过了考试");
        } else if (hasFinishedHomework) {
            System.out.println(" 完成了作业，但未通过考试");
        } else if (passedTheExam) {
            System.out.println(" 未完成作业，但已通过考试");
        } else {
            System.out.println(" 既未完成作业也未通过考试");
        }
    }
}
```

在这个例子中，我们声明了两个布尔变量 hasFinishedHomework 和 passedTheExam，分别代表学生是否完成了作业以及是否通过了考试。然后，我们使用选择语句 (if-else) 对这两个布尔值进行组合和单独判断，从而决定学生的不同状态及相应的反馈信息。这个简单的示例展示了如何运用布尔值来进行逻辑运算和决策控制，从而解决实际问题。

布尔类型作为一种相对简洁的基础数据类型，在计算机科学史上扮演着不可或缺的角色。为了让各位读者拓展阅读和深入了解这一重要概念的历史沿革，我将在下面讲一个布尔类型的插曲。

提示

布尔类型的由来

布尔数据类型的概念最早由 ALGOL 60 编程语言在 1960 年开创性地引入。自此以后，它成为了几乎所有后续设计的主流编程语言不可或缺的组成部分。

值得一提的是，C 语言在其最初的标准化版本 C89（亦称为 ANSI C）中，并未直接提供布尔类型的支持，而是采用了一种约定俗成的方式，即非零整数值代表逻辑真 (true)，而值为 0 则表示逻辑假 (false)。

直至 1999 年发布的 C 语言更新标准 C99 版本时，通过引入了头文件 stdbool.h，正式将布尔类型纳入规范。此举旨在为程序员提供一种更为直观和规范的方式来处理布尔逻辑表达式及其结果。

有很多语言借鉴了 C 语言的方法，比如 Python、JavaScript。幸好，Java 没有采用类似的方式。

4.3.5 类型转换

在 Java 编程中，数据类型转换是核心概念之一，它确保开发者能灵活处理八种基本数据类型的数值（如整数类型、浮点型、字符和布尔值），以适应复杂场景。不同类型间赋值时会发生转换：从较小到较大类型会自动进行（如 int 转 long），而从较大到较小类型则需显式指定，并确保不引发精度丢失或溢出问题。

掌握类型转换对编写高效且稳定的代码至关重要，可避免潜在错误并优化性能。类型转换可以分为两种情况：自动（隐式）转换和强制（显式）转换。接下来，我们将深入解析 Java 基本数据类型间的转换规则及应用示例。

自动（隐式）转换

在 Java 编程中，自动（隐式）类型转换是一种无需程序员显式指令即可完成的数据类型迁移过程。通常情况下，当两种数据类型之间存在层级关系时，即从数值范围较小的类型向数值范围较大的类型过渡，系统会自动执行此种转换。例如，可以将一个有效值域较窄的数据类型无缝转换为一个可容纳更大数值范围的数据类型，这种转换操作确保了数据精度得以完整保留。如下图所示，实线箭头所指示的方向代表着那些能够保持原始精度不损失的自动类型转换路径。

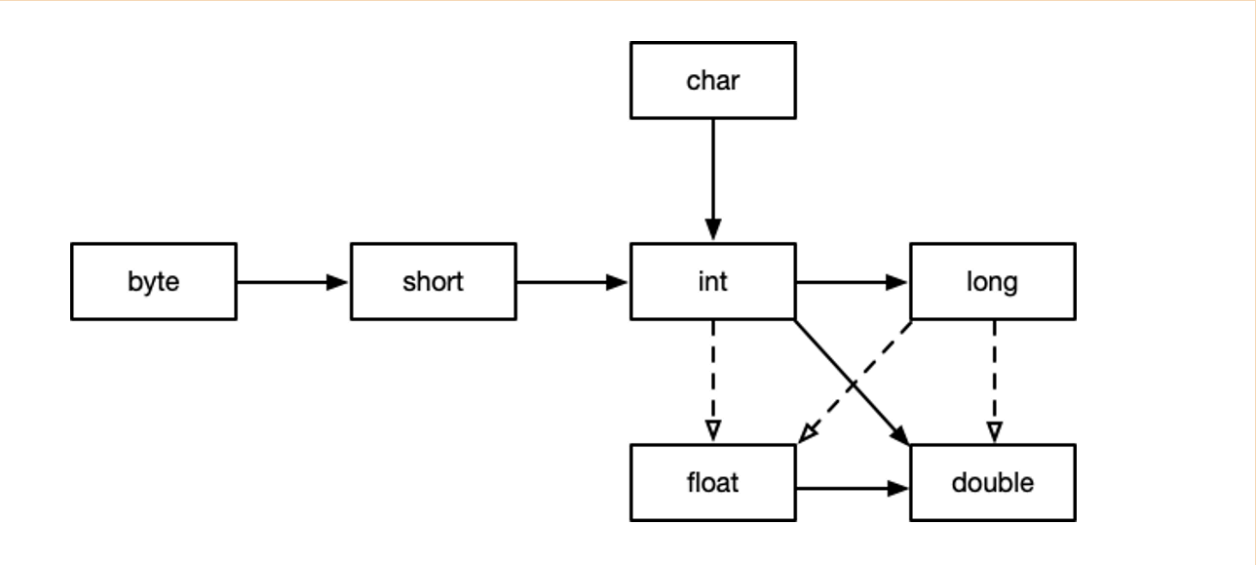

图 4.5: Java 基本数据类型之间的转换

自动类型转换依次遵从如下的规则：

规则 1：如果有一个操作数为 double 类型，那么先要将另一个操作数转换为 double 类型

规则 2：如果有一个操作数为 float 类型，那么先要将另一个操作数转换为 float 类型

规则 3：如果有一个操作数为 long 类型，那么将另一个操作数转换为 long 类型

规则 4：如果以上都不满足，那两个操作数都转换为 int 类型

了解了以上四条规则，我们通过几个示例来验证一下这些规则，代码如下所示。

```
public class TypeConversionExample {
    public static void main(String[] args) {
        // 规则 1: double 类型的优先级最高
        byte b = 5;
        double d = 10.5;
        System.out.println(b + d); // b 自动提升为 double 类型

        // 规则 2: 若有 float，其它转换为 float
        int i = 10;
        float f = 5.5f;
        System.out.println(i + f); // i 自动转换为 float 类型
```

```
        // 规则 3: 若有 long，其它转换为 long
        short s = 2;
        long l = 20L;
        System.out.println(s + l); // s 自动提升为 long 类型

        // 规则 4: 如果都是整数类型（非上述情况），则提升至 int
       byte b1 =120;
        byte b2 = 120;
        // 字符类型在运算前会先被转换为 int 类型
        System.out.println(b1 + b2);
    }
}
```

在实际编程中，当进行涉及 char 和 byte、short、int 之间的算术运算时，Java 会自动将其提升至 int 类型，然后再执行运算。此外，在表达式中涉及浮点数时，整数类型会根据规则提升至对应的浮点数类型。例如，如果一个表达式包含 int 和 double 类型的操作数，那么 int 类型的操作数会先被提升为 double 类型，然后再进行计算。

强制（显式）转换

在 Java 编程语言中，强制（显式）类型转换是指程序员明确地将一种数据类型转换为另一种数据类型的过程。例如，当试图将一个 double 类型的变量赋值给一个 int 类型的变量时，或者需要将一个 Object 类型的引用转换为 String 类型的引用时，就需要进行强制类型转换。

基本数据类型，例如整数类型（byte、short、int、long）、浮点数类型（float、double）或字符类型（char）之间互相转换时，如果目标类型无法自动容纳源类型的值，则必须使用强制类型转换。强制类型转换可能会导致精度丢失或溢出，因此程序员需要谨慎使用，并确保转换后的值在预期范围内。

为说明情况，我们通过如下代码来验证前面的说法。

```
public class TypeCastingExample {
    public static void main(String[] args) {
        // 示例 1: long 类型转 int 类型（可能丢失精度）
        // 下面的 largeValue 值大于 int 的最大值
        long largeValue = Integer.MAX_VALUE + 1;
        int intValue;
```

```
        // 强制类型转换
        // 由于 long 类型的 largeValue 超过了 int 的最大值，
        // 转换时会丢失精度，转换后的值不是 largeValue 的值
        intValue = (int) largeValue;
        System.out.println("long to int: " + intValue);

        // 示例 2: float 类型转 int 类型（同样可能丢失精度和尾数）
        float floatValue = 123.456f;
        // 强制转换为整数，会丢弃小数部分
        intValue = (int) floatValue;
        // 输出: 123，小数部分被舍去
        System.out.println("float to int: " + intValue);

        // 示例 3: 安全的类型转换（不丢失数据精度）
        // byte 的最大值是 127
        byte smallByte = 127;
        // 这种转换是安全的，因为 byte 的所有值都在 short 的范围内
        short smallShort = (short) smallByte;
        // 输出: 127
        System.out.println("byte to short: " + smallShort);
    }
}
```

在上面这个例子中，展示了几个不同场景下的强制类型转换：

1. 当尝试将一个超出目标类型表示范围的 long 值转换为 int 时，会发生精度丢失，因为 int 无法完整存储 long 的大数值。
2. 将 float 转换为 int 时，也会发生精度丢失，因为浮点数的小数部分会被截断，仅保留整数部分。
3. 在 byte 与 short 之间进行安全的转换，不会导致数据丢失，因为所有有效的 byte 值都能够在 short 类型中正确表示。

在实际编程中，应该尽量避免可能导致数据丢失的强制类型转换。为了保证数据的完整性和程序的正确性，在进行强制类型转换之前，务必充分理解目标类型的大小和取值范围。

4.4 引用数据类型

Java 中的引用数据类型变量与基本数据类型有着显著的不同。基本数据类型变量直接包含数据，而引用数据类型变量实际上并不直接存储对象本身，而是存储对对象的一个引用，即指向对象所在的内存地址。常见的引用数据类型有类（Class）、接口（Interface）、数组（Array）等。由于该类型的变量涉及内存管理和对象生命周期等更复杂的概念，为了确保讲解的连贯性和理解的深度，我决定将引用数据类型变量的详细讨论延后进行，待大家对类、接口等基础概念更为熟悉后再深入探究这一主题。

目前，只涉及一些表面的知识，比如，在 Java 中有如下的两条语句，它们分别用于定义两种不同类型的变量实例：

```
int age1 = 20;
Integer age2 = 20;
```

这两条声明语句揭示了 Java 中基本数据类型（如 int）与引用数据类型（如 Integer）之间的差异。

首先，int 是 Java 内建的基本整数类型，它直接存储数值，并在栈内存上分配空间，不涉及对象的概念。声明 int age1 = 20 时，变量 age1 将占用固定大小的空间来存储值 20，且由于其为基本数据类型，无法被赋予 null 值。

而 Integer 则是对基本数据类型 int 的封装类，属于引用数据类型。当声明 Integer age2 = 20 时，age2 变量实际存储的是一个指向 Integer 对象的引用，该对象在堆内存中创建并存储数值 20。由于 Integer 是一个对象，它可以具有额外的方法和属性，并且可以赋值为 null，这为其提供了更丰富的功能和灵活性，比如自动装箱与拆箱机制、与集合框架的良好兼容性等。

通过对比上述两条语句，我们可以初步理解 Java 类型系统中基本数据类型与引用数据类型的基本特征和使用场景，为进一步探究更深层次的编程技术打下坚实的基础。

4.5 变量的作用域

4.5.1 Java 变量的作用域

在 Java 中，变量的作用域指的是变量在程序中可以被访问的范围，如果在一个作用域内定义了一个变量，它会覆盖外层作用域定义的同名变量，这种情况称为变量遮蔽。无论对人名还是变量来说，朗朗上口的名字比较稀缺，非常容易重名。

重名问题普遍存在

《2019 年全国姓名报告》报告显示，全国户籍人口中，使用频率最高姓名“张伟”有近 30 万人。但是这 30 万“张伟”并没有对我们造成过多困扰。对我们每个人来说，“张伟”是有作用域的。

更普遍的例子，每个人都有“爸爸”，但是当喊“爸爸”的时候，并没有造成什么困扰，因为“爸爸”这个称呼在家庭这个作用域内是唯一的。当别人喊“爸爸”时，一定要注意作用域。

在 Java 中，变量的作用域取决于它被声明的位置。以下是一个简单的示例代码片段，通过下面代码中的例子可以清晰地展示类成员变量、方法局部变量以及嵌套块级作用域内变量的不同作用域：

```
public class ScopeExample {
    // 类作用域（类成员变量）：在整个类范围内有效
    public static int classVariable = 100;

    public void methodScope() {
        // 方法作用域（局部变量）：只在当前方法内部有效
        int methodVariable = 20;

        System.out.println("Class variable: "
                                + classVariable);
        System.out.println("Method variable: "
                                + methodVariable);

        // 块级作用域：只在当前大括号 {} 内有效
        {
            int blockVariable = 30;
            System.out.println("Block variable: "
                                + blockVariable);
        }

        // 尝试访问已超出作用域的 blockVariable 会编译错误
        // System.out.println(" 从 Bolck 外部访问"
                                + blockVariable);
```

```
    }

    public static void main(String[] args) {
        // 创建一个 ScopeExample 对象实例
        ScopeExample example = new ScopeExample();

        // 可以在 main 方法中直接访问类作用域的变量
        System.out.println(" 从 main 中访问类成员变量"
                           + example.classVariable);

        // 调用 methodScope 方法
        example.methodScope();

        // 在 main 方法中尝试访问 methodVariable
        // 或 blockVariable 会编译错误
        // 因为它们的作用域仅限于各自的方法和块内部
        // System.out.println(" 从 main 中访问"
                                   + methodVariable);
        // System.out.println(" 从 main 中访问"
                                   + blockVariable);
    }
}
```

在上述示例代码中，简要涉及了几种不同类型的作用域概念：类作用域（Class Scope）、、方法作用域（Method Scope）以及代码块作用域（Block Scope）。虽然目前尚未对这些作用域名进行详细的阐述，但它们正是后续章节要深入探讨的核心内容。通过逐步解析这些知识的不同特征和适用场景，读者将能够充分理解 Java 变量在其生命周期内如何受到作用域规则的制约与影响，并学会在实际编程过程中有效地管理和控制变量的范围。

在给出的代码示例中，我已通过详尽的注释对涉及不同作用域的变量进行了初步解读，并特意标识出那些超出作用域引用导致错误的部分，旨在直观展现变量的作用域这一核心概念。待你深入研读后续章节、系统性地学习之后，再次回顾此部分内容时，定会感到豁然开朗，原本看似复杂的知识点实则变得如此清晰易懂，恰似将厚重的知识体系“化繁为简”、“由厚读薄”的过程。

若能进一步探究历史上其他编程语言在处理作用域问题上的不同策略与机制，将极大地丰富我们对 Java 作用域特性的理解，并有助于从更广阔的视角去把握和应用这一关键概念。通过比较不同语言间的作用域设计理念与实践差异，不仅可以加深对 Java 作用域规则

的认识，还能促使我们更好地借鉴前人的智慧，在实际编程过程中更加得心应手地驾驭作用域这一核心要素。接下来，我讲一下作用域的历史。

4.5.2 作用域的历史

假如我们能回到编程语言发展的早期阶段，会发现彼时的语言在处理变量命名方面往往面临着严峻的挑战。由于早期编程中的变量几乎都是全局作用域，所有变量名在整个程序中均处于可见状态，这就如同在一个班级中，若存在多位同名的学生“张伟”，当教师在点名时，这些“张伟”无法确认老师喊的是谁。这种情况下，命名冲突的问题频频出现，严重影响了代码的可读性、可维护性和程序运行的稳定性。

假设一个班级中出现了多名“张伟”的情况，解决冲突的根本方式是要求其中一些学生改名。然而，将这种解决方案应用到编程领域，在面对庞大且复杂的代码体系时，单纯依靠避免变量名称重复的做法是不实际的（实际上，改名的方法在班级中也不现实）。因此，科学家提出了“作用域”这一概念，即为变量名定义特定的有效范围，确保在不同区域内的同名变量互不影响，从而彻底解决了程序中的命名冲突问题。

目前业界普遍采纳了两种作用域策略：一种是动态作用域，另一种则是静态作用域。Java 语言采用的是静态作用域机制，这意味着变量的作用域在编译阶段即可确定，并且其可见性和生命周期与程序的执行流程无关，仅依赖于其在源代码中的声明位置。通过这样的设计，Java 能够有效管理变量的使用和生命周期，保证了代码的准确性。

静态作用域在编程语言的编译阶段即被明确规定。它通过源代码中定义变量的位置，来界定变量的有效可见区域。在静态作用域机制下，一个函数或代码块内部声明的变量的作用域是由该函数或代码块本身的定义位置所决定的，并不受到调用位置的影响。当程序实际执行时，对变量的引用解析将严格遵循其在原始源代码中的声明顺序。

相反，动态作用域则是随着程序运行过程实时确定的。它以执行流程为基准，动态地界定变量的可见范围。在采用动态作用域的语言中，变量的有效性取决于当前正在执行的代码的上下文，而非其在源代码文本中的声明点。这意味着，查找并确定一个变量值的过程，会根据调用栈的具体状态以及代码的实际执行路径来进行。

当前，大多数编程语言普遍采纳的是静态作用域机制，这一做法在业界已深入人心，以至于习以为常。然而，作用域的概念及其发展历程远比当下所广泛采用的方案更为复杂多样。Perl 语言就是一个极富代表性的例子，它展示了作用域演进过程中的诸多挑战与变化。作为曾经的 Perl 程序员，我对 Perl 语言的作用域特性有着深刻的理解，可以说，Perl 语言几乎踩了变量作用域中所有的“坑”。通过了解 Perl 语言对作用域的处理方式，我们可以更全面地了解作用域概念的发展脉络。

故事的灵感来源于 Perl 语言的实际创始人 Larry Wall，一位毕业于加利福尼亚大学洛杉矶分校的杰出程序员。关于 Perl 语言，他公开过大量的书、博客与邮件。他曾任职于知名科技企业 Unisys 和希捷公司，身兼系统管理员和系统程序员双重角色，并在职业生涯的

某个阶段服务于美国国家安全局（NSA），参与了高度机密的项目研发。

Perl 语言诞生之初，正是为了应对从海量文本数据中高效提取关键信息这一迫切需求。接下来，我将通过一个虚构的故事来揭示 Perl 语言发展历程中的一个重要抉择：使用动态作用域，还是用静态作用域。希望这个故事可以帮助读者直观理解这两种作用域机制之间的本质区别。我将以第一人称的叙述方式，扮演故事中的主角进行虚构演绎。（在这本专注于 Java 语言的书中写 Perl 语言，希望能够为编程爱好者带来别样的乐趣与启示。）

4.5.3 虚构的故事：动态作用域和静态作用域

我是个程序员，我是计算机先驱，站在 1986 年人文和科技的十字路口上。

今天是 1986 年 6 月 22 日，在世界杯决赛的赛场上，马拉多纳带领阿根廷进了英格兰两个球。马拉多纳在球场上疯狂地挑衅，而英国人则绅士般站在球场，任由“潘帕斯草原上的雄鹰翱翔”。哦，忘记说一件事了，英国是个绅士国家，挑衅英国人只有一个结果，英国人会上诉到法院。

我不是阿根廷球迷，甚至，我不是足球球迷，但是今天我看到英格兰惨败，心里太高兴了，因为我的上司是个英国绅士，我的合作伙伴是英国公司，我已经快被工作压垮了，而上司却威胁我如果不能按期完成工作，那就法院见。

对了，我还没有说我的工作是什么呢。我的工作叫数据处理程序员，就是坐在计算机前面处理数据的那种人，现在是 1986 年，干我们这一行的还没多少人，以后也许会多吧，谁知道呢。我要给伦敦证券交易所的那帮混蛋处理数据，他们的数据太多，变化更多，不是每天都要数据，而是每秒都要数据，我真的快要被逼疯了。

我有一台最先进的计算机，但是只有几个蹩脚的工具，有两个处理文本数据的软件叫 sed 和 awk，还有一个不好用的编程语言叫 C 语言，还有个 shell，没有其它的了，这些工具像石器时代过来的，面对伦敦证券交易所每天产生的巨量数据，这些工具早晚会顶不住。

> 当年 Unix 上流行的 sed 和 AWK 是什么?
>
> AWK 是一种处理文本文件的语言，是一个强大的文本分析工具。特点是处理灵活，功能强大。可实现统计、制表以及其他功能。
>
> 之所以叫 AWK 是因为其取了三位创始人 Alfred Aho，Peter Weinberger, 和 Brian Kernighan 的 Family Name 的首字符。
>
> sed：Stream EDitor，流编辑器，它以行为单位处理字符流。

> 目前几乎没人使用这两个工具了，在中文世界中，由于编码的问题，不能处理中文，因此用的更少。

上帝保佑，如果我能有个更趁手的编程语言就好了，这个编程语言能够不像 C 语言一样每次使用前先编译，而是可以直接执行，这个语言还能和 sed 与 awk 完美结合。

我知道现在没有，求人不如求已，只要有空闲，我自己做一个这样的编程语言吧。

现在 Apple 公司很火，我也给我的编程语言起一个名字吧，就叫梨子语言，Pear 语言吧。这个语言是我自己用的，所以设计得没那么精密，只要能完成工作就好。

我有个优势，我有设计语言的条件，我妻子是个语言学家，她设计了一套语言，可以翻译多种版本的圣经，比我这个 Pear 语言可复杂多了。花了没多长时间，大概 3 个月吧，我这个语言就能运行了，至少再也不用写 C 语言了，伦敦证券交易所的那些数据每次都会有变动，再微小的修改，都要编译一次 C 语言，现在不用每次都编译了，我用得还是很开心的。

我把我这个语言告诉了我的同事，他们提了一些意见，总归不用 C 语言了，还有一些同事提出要把 sed 和 awk 也整合进来，我觉得挺好的。

又过了几个月，用我编程语言的人越来越多了，项目也快一年了，我打算把这个语言发到网上，如果有人喜欢，自己可以去编译一下，就发到 unix 的新闻组好了，就叫 Pear kit 1.0 吧。

我获得了很多赞扬，但是也有人提出了一些 bug，说这个语言不能很好地处理变量和作用域的问题。这个家伙发给我了一封邮件，我贴在这里，因为我的团队人不多，我还没遇到这个情况。

亲爱的 Pear 创作人，你好：

我非常喜欢你写的这个编程语言，每天都用得特别开心。我在工作中碰到了这样一个问题，因为我们是一个大型网站，有很多员工，Pear 语言在设计的时候对变量使用的是对照表，我理解你这样设计的优点是很容易实现，对个人来说已经够快了，但是我的同事太多了，经常会给某个变量起相同的名字，所以，每次都要处理名字相同的问题。

请问，有没有什么好办法或者好机制来处理名字相同的问题?

邮件就是这样，我读了好几遍。没想到竟然有商业公司用 Pear 语言写程序，如果人一多，确实会出现他说的那个问题。我最初设计的时候确实没考虑这么多，为了说明这个情况，我来解释一下，以便大家知道这到底是个什么问题，看看下面这段代码：

```
for( i = 0; i < 10; i++) {
    process();
```

```
    print i;
}
process() {
    i = 0;
}
```

由于我设计得不严谨，前面的循环里用到了一个变量 i，在循环里还调用了一个方法 process，结果这个 process 方法里也用到了变量 i，这样就把 i 的值改变了，导致上面的循环成了死循环。

我设计这个语言的时候，根本没有考虑这么多，我用了最简单的方法，用对照表，对照表的方法类似于身份证，出现一个变量就去一个列表中去查找，并且不能出现相同名字的变量，这个方法简单、粗暴、有效。

我暂时没想到好的解决方法，只好建议他尽量不要取相同的名字，或者如果有很多开发者的话，可以考虑每个开发者加上自己名字的前缀，比如 lyd_i 这样。

或者用下面这种方法，在方法入口处先保存原来的值，等到处理完以后，在方法出口的时候再把保存的值返回给变量。

```
process() {
  old_i = i;
  i = 0;
  i = old_i;
}
```

这是个权益之计，我知道我要想办法解决这个问题了。这样写太麻烦了，人为增加了难度，也会增加出错的几率。可以把这种写法加上一个关键字 local，有了这个关键字，可以让编程语言自己在程序入口处把值存起来，在方法出口处把值再复原回去。这样稍微减轻了一点工作量，上面这段代码可以这样写:

```
process() {
  local i;
  i = 0;
}
```

直到又有一个人提出这个方法仍然有 bug，这是我没想到的。我本以为是哪个人提出的无聊的问题，在我仔细看了邮件之后，才发现动态作用域的方法真的有严重的缺陷。

看看下面的代码:

```
x = "global";

print_x() {
    print x;
}

set_local() {
    local x = "local";
    print_x();
}
```

在这段代码中，当 set_local() 函数调用 print_x() 时，尽管 print_x() 并未接收任何参数，其内部试图输出的变量 x 的值却受到了 set_local() 函数内局部变量的影响。这是不合常理的，当调用一个函数的时候，如果没有给这个函数传递参数，那么这个函数的行为是不能改变的。

这个 bug 可以修吗？很难。如果要修，只能引入静态作用域，像 C 语言那样。但是，有这个必要吗？毕竟，我本来打算用动态作用域的……

4.6 程序员故事

《令人“恐怖”的 bug》

自从网络联通了每一个乡镇以后，我们这些以网络为生的人就有了一个符咒，每个人都痛恨去山区的机房或者基站里 debug，尤其是夜晚。

X 官庄村，在这里我之所以用 X 代替，是因为那一代的村子都叫 X 官庄，X 代表一个姓氏，大概有 10 来个小村子，蜿蜒散布于一条山沟之中。山里人的淳朴和执着，每到夜晚降临，每个村子会把一扇或铁或木做成的“门”关上，以防有人偷窃，在黎明时分再打开这扇“门”让道路恢复通畅。

由于人口不多，每个村子建一个机房成本太高了，因此，就在这几个村子大概中心的位置——一片荒山中略平的地方——建了一个共同的机房。机房里有监控设备，当有人进入的时候，会拍照，然后记录下来。最近出了一个诡异的情况，每到凌晨，设备会记录有人进入这个机房，但是查看监控的时候，又空无一人。甚至有时候会显示有多人进入，看监控的时候，连个鬼的影子都找不到。

设备厂家已经找过了，设备也换过了，始终有这个问题，公司让我和潘闻去看看怎么回事。潘闻很爽快的答应了，但是我知道，这条路是真的艰难，尤其是晚上去的时候，一路上挪开那几个村子放在路上的“门”，要挪七八次。

因为这个 “bug” 总是出现凌晨 1 点多，所以，那天晚上，我和潘闻 9 点就出发了。

夜已经很深了，远离了城市的灯光，抬头就可以看到银河挂在天空。我们的车就像一只飞在空中的萤火虫，顺着山路蔓延。很快到了第一个 “门”，这是一个很正常的 “铁门”，只是生锈了，我下车推开这个铁门，铁门发出巨大的 “吱吱嘎嘎” 的声音，铁门附近的房间里的灯马上就亮了，屋里传出一声怒吼 “谁？”，我回答我是来修机房的。房门被打开，一个亮度比照明弹还亮的手电射在我的脸上，我赶紧转头避开手电的强光。

盘问完毕，大爷放行的时候，我的眼睛才慢慢看清楚大爷的模样。已经秋天了，天气稍微有点冷，但是这个大爷只穿着拖鞋和一个裤衩，想必他应该是听到声音从床上一跃而起来抓 “贼” 的。

这样又经过了 5 个门，终于到达最后一个门了，最后一个门是最难缠的，因为这个门是专门挡机房这条路的，并没有村民每天来开关这扇门，所谓 “流水不腐，户枢不蠹”，这扇门好几个月才打开一次，户枢已经蠹的不像样了。

我让潘闻也下车，他表示不理解，我说这扇门一个人打不开。下车以后，我用手机的闪光灯检查了一下这扇门，它比上次我看到的它的时候更糟糕了。它本来有三个门枢，两个已经锈掉了，只剩下最下的一个门枢和一把 “锁” 来勉强维持平衡。这把 “锁” 其实是一根生锈的铁丝，我让潘闻一定要抓紧。然后，我小心翼翼的解开了那根生锈的铁丝，那道门像脱缰的野马，猛烈的跳动了一下，在潘闻还没有反应过来的时候，中间唯一管用的门枢断了，门重重的砸在潘闻的脚上。

在车灯的照射下，我看到他的脸先是变成红色，随后变成紫色，眼睛足足比平时大了一圈，额头上的青筋突然鼓得像网线似的。最后，喉咙里发出一声低沉的叫声，把门向前一推，跪在了地上。我吓的赶快过去问他伤的怎么样？他咒骂了足足一根烟的工夫才缓缓站起来，谢天谢地，他没有骨折，我和他把那扇门扔到路边，上了车，终于赶在 12 点前到达了这个荒野中的机房。

进了机房以后，我们查看了监控，今天好的不能再好了，好像故障神奇的修复了一样。为了模拟真实的情况，我们把机房的灯关了，只借着手机的背光东一句西一句的聊天。我心里想：“不会见鬼了吧，今天一点事情也没有。” 好像鬼读懂了我的心思一样，实然间，机房的灯打开了，监控开始工作，再查看监控的记录，显示进来了三个人。

我看看潘闻，潘闻看看我，在凌晨的荒山野岭里，竟然有三个人在机房里，这可比《张震讲故事》刺激多了。我提议把灯再关了，再试试能不能重复这个 bug。这次，我俩没什么好谈的了，小潘在手机里放了一首歌，我明白他的意思，走夜路吹口哨，能自己给自己壮胆。歌还没放完，灯又亮了，监控又开始工作，现在机房里已经有四个人了。

如此重复了几次，现在这个小小的机房里已经显示进来了十个人，挺拥挤的。我们俩个检查了电源，检查了设备，还重启了能重启的所有设备，都不管用。潘闻把机房门打开，把车发动起来，秋天的后半夜已经有点冷了，在山里，会更冷一些。我们在车里坐了一会，

车里开着暖气，车窗已经起了一层薄薄的水气。我们决定搞不定就离开了，突然，小潘盯着车窗，说：“会不会是潮湿导致的呢？”

我从工程车里拿出梯子，找到摄像头，摄像头在外面，这个摄像头没有做任何防潮的设计，再加上山里不同于城市，到夜里，潮湿的水气会凝结在摄像头上，所以会导致在夜间失灵。

我们提交了一个报告，公司把这个摄像头换成了有防潮设计的以后，机房里就再也没有发生过“灵异事件”了。在我的职业生涯中碰到过无数的 bug，有软件的 bug 有硬件的 bug，但是只有这个 bug 让我感到过毛骨悚然。

5 操作符

Contents

终日奔忙只为饥，才得有食又思衣。
置下绫罗身上穿，抬头又嫌房屋低。
盖下高楼并大厦，床前却少美貌妻。
娇妻美妾都娶下，又虑出门没马骑。
将钱买下高头马，马前马后少跟随。
家人招下数十个，有钱没势被人欺。
一铨铨到知县位，又说官小势位卑。
一攀攀到阁老位，每日思想要登基。
一日南面坐天下，又想神仙来下棋。
洞宾与他把棋下，又问哪是上天梯。
上天梯子未坐下，阎王发牌鬼来催。
若非此人大限到，上到天上还嫌低。

人性贪婪自不必多言，软件膨胀屡见不鲜。其实到了编程语言上，功能自然也是叠床架屋。就操作符来说，一个小小的“+”号，连小学生都知道的功能，最后却演化出千变万化的功能，着实是乱花渐欲迷人眼。

如果你问一个小学生说，1 + 1 等于几，他会说等于 2。如果你问 1 只猫加 1 只狗等于几？那就值得讨论一下了，如果回答 2，实际上是有问题的。猫跟狗是没法相加的。但是在编程语言中，尤其是高级一点的编程语言中，char 类型的“1” + 整数类型的 1，是可能相加的，而且得出的结果还不是 2。

不过呢，总体来说，Java 中的操作符，还是比较“符合直觉”的。如果你不想跳过这一章的话，可以继续读下去，如果你跳过这一章，我觉得问题也不大，不像 C++，可以把 + 重载运算符为 – 的操作。

读不读这一章，看你的心情了，我觉得跳过没关系。因为我要写书，必须得把内容写全。实际上，如果你会其它任何编程语言，这一章都可以跳过，到时候摊上事了，再回头来补票也不是不行。但是人这种动物，就是很奇怪，你不让看，他偏说：我倒要看看。

图 5.1: 我倒要看看

在 Java 编程语言中，操作符是一种特殊的符号或一组符号，它们被设计用来执行各种关键的计算、比较和逻辑操作。操作符可以细分为多个类别，包括但不限于算术操作符（如加减乘除）、比较操作符（用于判断两个值之间的大小关系）、逻辑操作符（处理布尔逻辑）以及位移操作符（对二进制数进行移位操作）。这些操作符的核心功能是根据预定义的规则对操作数进行运算，并由此生成一个新的结果值。

以表达式 a + b 为例，“+” 是一种算术操作符，负责执行加法运算；而 “a” 和 “b” 则是该操作符的操作数，它们是待相加的数值。通过操作符与操作数的配合，Java 能够实现复杂的数据处理和逻辑控制，并生成新的值。

在上一章节的讨论中，我们重点学习了变量这一核心概念，并深入讲解了变量类型的划分，主要将其分为两大类：基本数据类型和引用数据类型。当操作数为基本数据类型时，其与操作符之间的交互逻辑相对直观且易于掌握；然而，一旦操作数转变为引用数据类型，运算过程中可能会有一些“反直觉”的问题出现。

尤其值得注意的是，即使使用相同的操作符，对不同类型的变量进行操作，也可能导致完全不同的结果产生。因此，理解和掌握不同类型数据与操作符结合后的行为差异至关重要。举例来说：

```
public class ArithmeticOperatorExample {

    public static void main(String[] args) {
        // 基本数据类型（数值类型）的例子
        int num1 = 3, num2 = 4;
        int sum = num1 + num2;
        System.out.println(" 数值类型的加法结果: " + sum);

        // 引用数据类型（字符串）的例子
        String str3 = "Hello";
        String str4 = "World";
        String concat = str3 + " " + str4;
        System.out.println(" 字符串连接的结果: " + concat);
    }
}
```

这段代码很好地展示了“+”操作符在不同数据类型上的不同行为：

数值类型：当操作数为数值类型（如 int）时，“+”操作符执行加法运算，将两个数值相加并得到它们的和。

字符串类型：当操作数为字符串类型（如 String）时，“+”操作符执行字符串连接操作，将两个字符串首尾相连形成一个新的字符串。

这种现象用术语来说，“+”操作符在 Java 中具有重载的特性，它可以根据操作数的类型执行不同的操作。这体现了 Java 语言的灵活性，但也需要开发者注意操作数类型，避免产生预期之外的结果。Java 语言内建了超过 30 种操作符，它们可被归纳为以下几类：

1. **算术操作符**：这类操作符用于执行基本数学运算，如加（+）、减（-）、乘（*）、除（/）以及取模（%）等。
2. **赋值操作符**：这些操作符主要用于向变量赋值。除了简单的等于号（=）之外，还有复合赋值操作符，例如 +=、-=、*= 和 /=，它们分别实现将一个值与另一个值相加、减、乘或除后，再将结果赋给原变量。
3. **关系操作符**：这类操作符用来比较两个值的关系，返回布尔类型的真（true）或假（false）。关系操作符包括大于（>）、小于（<）、大于等于（>=）、小于等于（<=）以及等于（==）。

4. **逻辑操作符**：此类操作符用于组合布尔表达式以构建更为复杂的逻辑判断。其中包括逻辑与（&&）、逻辑或（||）、逻辑非（!），以及短路与（&）、短路或（|）、异或（^）等。
5. **自增、自减操作符**：这类操作符用于递增或递减变量的值。自增操作符（++）会使变量的值增加 1，它可以放在变量前面（前缀形式）或后面（后缀形式），两者的区别在于取值时机不同。而自减操作符（–）则会让变量的值减少 1，同样具有前缀形式和后缀形式。
6. **位操作符**：这类操作符在二进制位级别上对变量进行操作。位操作符包括按位与（&）、按位或（|）、按位非（~）、按位异或（^）、左移位（<<）、右移位（>>）和无符号右移位（>>>）
7. **三元条件操作符**：这是一种简化的选择语句形式（?:），根据一个布尔表达式的值来选择并计算两个可能的表达式中的一个。

以上各类操作符构成了 Java 编程语言的核心组成部分，程序员通过灵活运用这些操作符，能够编写出各种复杂度的算法和程序逻辑。因此，我将对上述几类 Java 操作符进行详细的介绍。首先介绍算术操作符。

5.1 算术操作符

Java 中的算术操作符主要用于进行数值计算，包括基本数据类型（如 int、double、float 等）以及某些支持算术运算的引用类型（如 String 类）。

```
public class ArithmeticOperatorsExample1 {

    public static void main(String[] args) {
        // 加法操作符
        int a = 5, b = 3;
        int sum = a + b;
        System.out.println(" 加法结果: " + sum);

        String str1 = "Hello";
        String str2 = "World";
        String combined = str1 + " " + str2;
        System.out.println(" 字符串连接结果: " + combined);

        // 减法操作符
        int c = 10, d = 4;
        int difference = c - d;
```

```
        System.out.println(" 减法结果: " + difference);

        // 乘法操作符
        int e = 3, f = 7;
        int product = e * f;
        System.out.println(" 乘法结果: " + product);

        // 除法操作符
        double g = 10.0, h = 3.0;
        double quotient = g / h;
        System.out.println(" 除法（浮点数）结果: " + quotient);

        // 取模/求余操作符
        int k = 10, l = 3;
        int remainder = k % l;
        System.out.println(" 取模结果: " + remainder);
    }
}
```

上面的示例运行以后得到：

```
加法结果: 8
字符串连接结果: Hello World
减法结果: 6
乘法结果: 21
除法（浮点数）结果: 3.3333333333333335
求余结果: 1
```

这里值得注意的是对于字符串的 “+” 运算符，它不是一个传统的算术操作符，在 Java 中，它还被设计用来拼接字符串。对基本数据类型来说，算术操作符遵循基本的数学运算规则，并且具有数学上的优先级和结合性，可以使用圆括号来改变操作符的优先级。

5.2 赋值操作符

在 Java 中，赋值操作符用于将一个值赋予变量。基本的赋值操作符是等号 = ，它用来将右边表达式的值复制给左边的变量。Java 中赋值操作符主要分为两种类型：一类是基本赋值操作符（=），另一类是复合赋值操作符。复合赋值操作符结合了算术或位运算，从而

简化了常见的操作步骤。值得注意的是：复合赋值操作符都会先执行相应的运算，然后将结果赋值给左侧变量，同时返回该结果作为新的值。在进行复合赋值时，注意数据类型的兼容性和可能发生的类型转换。接下来，用代码演示一下赋值操作。

```
public class AssignmentOperatorsExample {

    public static void main(String[] args) {
        // 基本赋值操作符
        int a;
        a = 5;
        System.out.println(" 基本赋值后的 a: " + a);

        // 复合赋值操作符示例
        int b = 3;
        b += 2; // 相当于 b = b + 2;
        System.out.println(" 加法赋值后的 b: " + b);

        int c = 7;
        c -= 4; // 相当于 c = c - 4;
        System.out.println(" 减法赋值后的 c: " + c);

        int d = 3;
        d *= 5; // 相当于 d = d * 5;
        System.out.println(" 乘法赋值后的 d: " + d);

        double e = 10.0;
        e /= 2.0; // 相当于 e = e / 2.0;
        System.out.println(" 除法赋值后的 e: " + e);

        int f = 13;
        f %= 5; // 相当于 f = f % 5;
        System.out.println(" 取模（求余数）赋值后的 f: " + f);

        int g = 4;
        g <<= 1; // 相当于 g = g << 1;
        System.out.println(" 左移赋值后的 g: " + g);
```

```
        int h = 8;
        h >>= 1; // 相当于 h = h >> 1;
        System.out.println(" 右移（有符号）赋值后的 h: " + h);

        int i = -8;
        i >>>= 1; // 相当于 i = i >>> 1;
        System.out.println(" 无符号右移赋值后的 i: " + i);

        int j = 0x0F;
        j &= 0x0B; // 相当于 j = j & 0x0B;
        System.out.println(" 按位与赋值后的 j: "
                + Integer.toHexString(j));

        int k = 0x0F;
        k |= 0x0A; // 相当于 k = k | 0x0A;
        System.out.println(" 按位或赋值后的 k: "
                + Integer.toHexString(k));

        int l = 0x0F;
        l ^= 0x0A; // 相当于 l = l ^ 0x0A;
        System.out.println(" 按位异或赋值后的 l: "
                + Integer.toHexString(l));
    }
}
```

以上代码运行之后的结果如下：

```
基本赋值后的 a: 5
加法赋值后的 b: 5
减法赋值后的 c: 3
乘法赋值后的 d: 15
除法赋值后的 e: 5.0
取模（求余数）赋值后的 f: 3
左移赋值后的 g: 8
右移（有符号）赋值后的 h: 4
无符号右移赋值后的 i: 2147483644
按位与赋值后的 j: b
```

```
按位或赋值后的 k: f
按位异或赋值后的 l: 5
```

5.3 关系操作符

在 Java 中，关系操作符用于比较两个值并返回一个布尔值（true 或 false）。当使用关系操作符时，要考虑值的类型。因此，我们将分别讨论基本数据类型和引用数据类型的比较。首先，让我们来看一下基本数据类型的比较。下面的代码是演示如何使用关系操作符的：

```
public class RelationalOperatorsExample {

    public static void main(String[] args) {
        // 基本类型比较
        int x = 5, y = 5;
        boolean isEqual = (x == y);
        System.out.println("x 是否等于 y: " + isEqual);

        int a = 10, b = 5;
        boolean isGreaterThan = (a > b);
        System.out.println("a 是否大于 b: " + isGreaterThan);

        boolean isLessThan = (b < a);
        System.out.println("b 是否小于 a: " + isLessThan);

        boolean isGreaterThanOrEqual = (a >= b);
        System.out.println("a 是否大于或等于 b: "
                                    + isGreaterThanOrEqual);

        boolean isLessThanOrEqual = (b <= a);
        System.out.println("b 是否小于或等于 a: "
                                    + isLessThanOrEqual);
    }
}
```

以上代码运行之后的结果如下：

```
x 是否等于 y: true
a 是否大于 b: true
b 是否小于 a: true
a 是否大于或等于 b: true
b 是否小于或等于 a: true
```

关系操作符通常不应用于布尔类型（boolean），因为布尔类型只有两种可能的值：true 和 false，其本身已经代表了逻辑上的真假，一般无需进行大小比较。

探讨了基本数据类型的比较之后，接下来我们将讨论关系运算符在引用数据类型中的应用。基本数据类型和引用数据类型在关系运算方面存在一些差异。为了更好地理解这些差异，让我们先看一些代码示例。

重要

下面代码的例子，我在 B 站和 YouTube 有一期视频专门讲这事，你如果是视频流选手，可以搜一下“软件那些事儿”这个账户中，有一期的标题是：Java 中 == 和 equals() 的区别。

总之，这里稍微有点重要，算是出问题比较多的地方。

```java
public class ReferenceTypeComparison {

    public static void main(String[] args) {
        // 基本类型的包装类 Integer 示例
        Integer a = new Integer(5);
        Integer b = new Integer(5);

        // 比较数值大小
        boolean equalByValue = (a == b);
        // 输出: true
        System.out.println("a 和 b 是否引用相等: "
                                        + equalByValue);

        // Integer 自动装箱优化示例
        Integer x = 5;
        Integer y = 5;
```

```
        boolean equalByValueExample = (x == y);
        // 输出: true
        System.out.println("x 和 y 是否引用相等: "
                           + equalByValueExample);

        // 使用 equals() 方法进行内容比较
        Integer m = new Integer(5);
        Integer n = new Integer(5);

        boolean equalByContentExample = m.equals(n);
        // 输出: true
        System.out.println("m 和 n 是否值相等: "
                           + equalByContentExample);

        // 使用 equals() 方法进行内容比较
        Integer p = 5;
        Integer q = 5;
        boolean equalByContentExample2 = p.equals(q);
        System.out.println("p 和 q 是否值相等: "
                           + equalByContentExample2);
    }
}
```

以上的代码运行以后，输出如下结果：

```
a 和 b 是否引用相等: false
x 和 y 是否引用相等: true
m 和 n 是否值相等: true
p 和 q 是否值相等: true
```

需要注意的是，在 Java 中，使用等于（==）运算符比较两个引用类型的变量时，实际上比较的是它们是否指向同一个对象，而不是它们的内容是否相等。在代码中，a 和 b 都是通过 new Integer(5) 创建的，这会在内存中的堆区域创建两个新的 Integer 对象，即使它们的内容都是 5，但它们是存储在不同内存地址的两个独立对象，所以 a == b 的结果是 false。

因此，如果想要比较两个 Integer 对象的值是否相等，应该使用 equals() 方法，如后来比较 m 与 n 两个变量时，使用的 m.equals(n) 方法。这个方法会“逐位比较”两个 Integer 对象的数值是否相等，这也是为何 m 与 n 相等的原因。当我说 equals() 方法使用“逐位比较”

时，指的是它会比较两个对象的每个组成部分是否完全相同。具体到 Integer 对象，“逐位比较” 意味着：

类型检查: 首先，equals() 方法会检查比较的对象是否都是 Integer 类型。如果不是，则直接返回 false，因为不同类型的对象不可能相等。

数值比较: 如果两个对象都是 Integer 类型，则 equals() 方法会提取它们所包含的整数值，并逐位进行比较，确保每个二进制位都完全相同。

例如，对于两个 Integer 对象 a 和 b ，如果它们的整数值都是 10，则它们的二进制表示都是 00001010 ，equals() 方法会逐位比较这两个二进制数，发现它们完全相同，因此返回 true。

需要注意的是，“逐位比较” 并不适用于所有类型的对象。对于自定义对象或其他引用类型，equals() 方法的行为可能会有所不同，具体取决于该类型的实现。通常，自定义对象会根据其自身属性的定义来判断是否相等。

再来看 x、y 这两个变量，这两个变量也是 Integer 类型的变量，但是与 a、b 不同的是，一个是使用 new Integer() 创建的，一个是使用自动装箱（即 Java 自动将基本类型转换为包装类型）创建的。这两种方式创建的对象存在一些差异，具体来说其区别如下：

相比之下，当使用 new Integer(value) 的方式创建一个 Integer 对象时，每次都会生成一个新的对象实例。这意味着对于相同的整数值，每次调用 new Integer() 都会在内存中分配新的空间来存储该值。而通过自动装箱（尤其是对于 -128 至 127 之间的整数）可能会复用已经存在的对象，这是因为 Java 内部维护了一个 Integer 对象池，用于缓存常用的 Integer 对象，从而节省内存并提高性能。

自动装箱的具体原理是，当一个基本类型的变量赋给 Integer 类型的变量时，编译器会自动插入 Integer.valueOf() 方法。该方法对 -128 到 127 之间的整数有一个缓存机制，这是为了提高性能并节省内存，因为这些数值在程序中使用非常频繁。因此，对于这个范围内的整数，多次自动装箱操作可能指向同一个 Integer 实例。

由于自动装箱的缓存机制，当使用比较（==）操作符比较 x、y 和使用 equals() 比较 p、q 的时候，返回的值都是 true，因为 equals() 方法会比较两个 Integer 对象的数值是否相等。为了更深入地理解自动装箱的缓存机制，让我们先来看下面的一段代码：

```
public class ReferenceTypeComparisonWithCache {

    public static void main(String[] args) {
        // Integer 自动装箱优化示例，-128 到 127 范围内的缓存区别
        Integer x = 127;
        Integer y = 127;
```

```
        boolean equalByValueWithinCacheRange = (x == y);
        // 输出: true
        System.out.println("x 和 y 是否引用相等（缓存范围内）: "
                + equalByValueWithinCacheRange);

        Integer m = 128;
        Integer n = 128;

        boolean equalByValueOutsideCacheRange = (m == n);
        // 输出: false
        System.out.println("m 和 n 是否引用相等（缓存范围外）: "
                + equalByValueOutsideCacheRange);

        // 使用 equals() 方法进行内容比较
        boolean equalByContent = m.equals(n);
        // 输出: true
        System.out.println("m 和 n 是否值相等: "
                + equalByContent);
    }
}
```

这段代码运行以后，其结果是：

```
x 和 y 是否引用相等（缓存范围内）: true
m 和 n 是否引用相等（缓存范围外）: false
m 和 n 是否值相等: true
```

通过上面的程序可以知道，当整数值在 -128 至 127 之间时，Integer 对象使用自动装箱和构造函数创建的对象具有相同的引用；而超出该范围时，则不再共享引用，这是因为 Java 认为超出此范围的数值使用频率较低，没有必要缓存。同时，equals() 方法始终可以确保基于内容的相等性检查，不受缓存范围的影响。

5.4 逻辑操作符

Java 中的逻辑操作符主要用于连接或反转布尔表达式，以组合或修改多个条件的逻辑结果。逻辑操作符主要是与或非三种，接下来，分别介绍这三种逻辑操作符。

5.4.1 逻辑“与”（AND）

逻辑“与”操作符在 Java 中使用符号 && 表示。如果两个操作数都为 true，则整个表达式的结果是 true；否则，结果就是 false。它遵循短路逻辑，这意味着如果左侧操作数为 false，右侧的表达式将不会被执行，因为无论右侧如何，整个表达式结果已经是 false。接下来，看一段演示代码：

```
public class LogicalAndExample {
    public static void main(String[] args) {
        boolean condition1 = true;
        boolean condition2 = false;

        // 使用逻辑与（AND）操作符
        if (condition1 && condition2) {
            System.out.println(" 两个条件均为真");
        } else {
            System.out.println(" 至少有一个条件为假");
        }

        // 短路逻辑的示例
        int value1 = 10;
        int value2 = 0;

        // 如果条件 1 为 false，则不会执行条件 2 的除法运算，
        // 因为无论其结果如何，整个表达式都将为 false
        if (value1 != 0 && value2 / value1 > 5) {
            System.out.println(" 如果 value1 不为零，
                                将会执行除法操作。");
        } else {
            System.out.println(" 基于短路，
                            不执行 if 条件中的除法操作。");
        }
    }
}
```

上面的代码运行之后，输出结果如下：

```
至少有一个条件为假
基于短路，不执行 if 条件中的除法操作。
```

这段代码展示了逻辑“与”操作符的使用和短路特性。首先，程序定义了两个布尔变量 condition1 和 condition2，并使用 && 操作符将它们连接起来，形成一个复合条件表达式。由于 condition2 的值为 false，整个表达式结果为 false，因此程序输出“至少有一个条件为假”。

接下来，程序定义了两个整型变量 value1 和 value2，并再次使用 && 操作符构建了一个复合条件表达式。由于逻辑“与”操作符的短路特性，当 value1 的值为 0 时，程序不会执行 value2 / value1 > 5 这部分代码，从而避免了除以零的错误。反之，如果 value1 的值非零，程序会继续判断 value2 / value1 是否大于 5，并根据结果决定执行 if 语句块还是 else 语句块中的代码。

5.4.2 逻辑“或”（OR）

逻辑或操作符在 Java 中使用符号 || 表示。它用于连接两个布尔表达式，只要其中一个表达式为 true，整个表达式的结果就为 true；只有当两个表达式都为 false 时，整个表达式的结果才是 false。

逻辑或操作符同样具有短路特性，这意味着如果左侧的表达式结果为 true，那么右侧的表达式将不会被执行，因为无论右侧表达式的值如何，整体逻辑或表达式的结果已经可以确定是 true。接下来，通过代码来验证前面所提到的问题：

```
public class LogicalOrExample {
    public static void main(String[] args) {
        boolean isMorning = false;
        boolean isAfternoon = true;

        // 使用逻辑或运算符 ||
        if (isMorning || isAfternoon) {
            System.out.println(" 现在是早上或下午");
        } else {
            System.out.println(" 现在既不是早上也不是下午");
        }

        // 短路逻辑演示
        boolean hasPermissionA = true;
        boolean hasPermissionB = false;
```

```
            // 因为 hasPermissionA 为 true,
            // 所以不需要检查 hasPermissionB 的值
            // 直接输出权限已授予
            if (hasPermissionA || hasPermissionB) {
                System.out.println(" 用户至少拥有权限 A 或 B,
                                    因此已授予访问权限");
            } else {
                System.out.println(" 用户没有权限 A 和 B, 无法访问");
            }
        }
}
```

以上代码运行之后，输出的结果是：

```
现在是早上或下午
用户至少拥有权限 A 或 B, 因此已授予访问权限
```

这段代码展示了逻辑或操作符的使用和短路特性。首先，程序定义了两个布尔变量 isMorning 和 isAfternoon，并使用 || 操作符将它们连接起来，形成一个复合条件表达式。由于 isAfternoon 的值为 true，整个表达式结果为 true，因此程序输出 “现在是早上或下午”。

接下来，程序定义了两个布尔变量 hasPermissionA 和 hasPermissionB，并再次使用 || 操作符构建了一个复合条件表达式。由于逻辑或操作符的短路特性，当 hasPermissionA 的值为 true 时，程序不会检查 hasPermissionB 的值，直接输出 “用户至少拥有权限 A 或 B，因此已授予访问权限”。这有效地提高了程序的效率，避免了不必要的计算。

5.4.3 逻辑 “非”（NOT）

逻辑非操作符在 Java 中使用符号! 表示。它用于对一个布尔表达式求反，将 true 变为 false，将 false 变为 true。下面的代码是使用逻辑非操作符的例子：

```
public class LogicalNotExample {
    public static void main(String[] args) {
        // 假设今天下雨了
        boolean isRaining = true;

        // 使用逻辑非运算符 !
        if (!isRaining) {
```

```
            System.out.println(" 今天没有下雨");
        } else {
            System.out.println(" 今天正在下雨");
        }
    }
}
```

上面的代码运行之后，输出结果如下：

```
今天正在下雨
```

这段代码展示了逻辑非操作符的使用。首先，程序定义了一个布尔变量 isRaining，并将其赋值为 true，表示今天正在下雨。然后，程序使用! 操作符对 isRaining 进行求反，得到的结果为 false。因此，if 语句的条件不成立，程序输出 “今天正在下雨”。

5.5 自增、自减操作符

在 Java 中，自增（++）和自减（–）操作符的主要用途在于方便地对变量进行加 1 或减 1 的操作，它们可以作为前缀形式或后缀形式出现。它们广泛应用于循环、计数器、数组遍历以及其他需要动态改变数值的场景。通过下面的代码来演示自增、自减的操作。

```
public class IncrementDecrementExample {
    public static void main(String[] args) {
        // 定义一个整数变量
        int counter = 0;

        // 前缀自增
        // 输出: 1，然后 counter 值变为 1
        System.out.println(" 前缀自增: " + (++counter));
        // 输出: 1
        System.out.println(" 当前 counter 值: " + counter);

        // 后缀自增
        // 输出: 1，然后 counter 值变为 2
        System.out.println(" 后缀自增: " + (counter++));
        // 输出: 2
        System.out.println(" 当前 counter 值: " + counter);
```

```
        // 前缀自减
        // 输出: 1，然后 counter 值变为 1
        System.out.println(" 前缀自减: " + (--counter));
        // 输出: 1
        System.out.println(" 当前 counter 值: " + counter);

        // 后缀自减
        // 输出: 1，然后 counter 值变为 0
        System.out.println(" 后缀自减: " + (counter--));
        // 输出: 0
        System.out.println(" 当前 counter 值: " + counter);
    }
}
```

以上的代码运行之后，其结果如下：

```
前缀自增: 1
当前 counter 值: 1
后缀自增: 1
当前 counter 值: 2
前缀自减: 1
当前 counter 值: 1
后缀自减: 1
当前 counter 值: 0
```

在 Java 中，自增（++）和自减（–）操作符用于将变量的值加 1 或减 1，它们可以作为前缀形式或后缀形式出现。前缀形式会先进行加/减操作，再返回变量的值；而后缀形式会先返回变量的值，再进行加/减操作。它们广泛应用于循环、计数器、数组遍历以及其他需要动态改变数值的场景。

传闻只是传闻

我还在上学的时候，老师会给我们讲 C 语言中的 a++ 与 a = a + 1 有什么区别。也不知道他是从哪里看来的，但是显然他深信不疑。他说 a++ 的效率要比 a = a + 1 要高，原因是巴拉巴拉巴拉巴拉……

我也就跟着相信了，后来，工作原因需要研究 gcc 编译后的汇编。我才发现，老师说的并不正确，gcc 会把 a++ 与 a = a + 1 编译成相同的汇编代码，两者是一模一样的。如果有下面两行代码：

```
int a = 2;
a = a +1;
a++;
```

被 gcc 编译以后，其汇编代码都是一样的，都是：

```
mov eax, 2    ; 把2放到 eax 寄存器
inc eax       ; 自增操作
```

我无意讲我当时的老师有多么差，毕竟我上大学时也没听过几次课，他的名字我已经忘记了。工作之后，有很多传闻，仅仅是传闻，经不起验证与推理的。这只是其中一个微不足道的小例子。

很多的最佳实践，只是在那个场景下适用。而且这种最佳实践，绝大部分不是设计出来的，而是形式所迫，逼出来的。

5.6 位操作符

Java 中的位操作符主要用于对整数的二进制位进行直接操作。这些操作符可以在底层对数据的比特位进行按位逻辑运算，例如按位与（&）、按位或（|）、按位异或（^）等，从而实现特定的算法和优化。它们常用于硬件控制、网络协议编码解码、图像处理等领域。接下来，通过下面的代码来演示位操作符的用法。

```
public class BitwiseOperationsExample {

    public static void main(String[] args) {
        // 示例整数
        int a = 6; // 0b0110
        int b = 3; // 0b0011
```

```
// 按位与 (&)
int bitwiseAndResult = a & b;
System.out.println(" 按位与结果: "
    + Integer.toBinaryString(bitwiseAndResult)
    + " (十进制: " + bitwiseAndResult + ")");

// 按位或 (|)
int bitwiseOrResult = a | b;
System.out.println(" 按位或结果: "
    + Integer.toBinaryString(bitwiseOrResult)
    + " (十进制: " + bitwiseOrResult + ")");

// 按位异或 (^)
int bitwiseXorResult = a ^ b;
System.out.println(" 按位异或结果: "
    + Integer.toBinaryString(bitwiseXorResult)
    + " (十进制: " + bitwiseXorResult + ")");

// 按位取反 (~)
int bitwiseNotA = ~a;
System.out.println(" 对 a 进行按位取反的结果: "
    + Integer.toBinaryString(bitwiseNotA)
    + " (十进制: " + bitwiseNotA + ")");

// 左移 (<<)
int leftShiftResult = a << 1;
System.out.println(" 左移一位的结果: "
    + Integer.toBinaryString(leftShiftResult)
    + " (十进制: " + leftShiftResult + ")");

// 有符号右移 (>>)
// 注意: 负数在右移时会根据符号位扩展
int signedRightShiftResult = -6 >> 1;
System.out.println(" 有符号右移一位的结果: "
    + signedRightShiftResult);
```

```
        // 无符号右移 (>>>)
        int unsignedRightShiftResult = -6 >>> 1;
        System.out.println(" 无符号右移一位的结果: "
            + unsignedRightShiftResult);
    }
}
```

以上代码运行之后输出的结果如下:

```
按位与结果: 10 (十进制: 2)
按位或结果: 111 (十进制: 7)
按位异或结果: 101 (十进制: 5)
对 a 进行按位取反的结果: 11111111111111111111111111111001
                                                (十进制: -7)
左移一位的结果: 1100 (十进制: 12)
有符号右移一位的结果: -3
无符号右移一位的结果: 2147483645
```

这段代码展示了 Java 中常见的位操作符，包括按位与（&）、按位或（|）、按位异或（^）、按位取反（~）、左移（«）、有符号右移（»）和无符号右移（»>）。程序首先定义了两个整型变量 a 和 b，并将其转换为二进制形式进行演示。然后，程序使用不同的位操作符对 a 和 b 进行运算，并将结果输出。例如，a & b 的结果是 2，这是因为只有 a 和 b 的二进制表示中都为 1 的位才会保留下来。

尽管在日常业务逻辑处理中，位操作符不如算术和逻辑运算符常见，但在特定场景下，它们的作用至关重要。例如，在网络编程、文件系统权限管理等领域，位掩码常用于存储一组标志或选项。在我的工作中，我经常使用位操作符进行数据编码和解码，以及处理网络协议。它们是实现高性能计算、低级别编程以及解决特定问题的有效工具。

5.7 三元操作符

Java 中的三元操作符（也称为条件运算符或三目运算符）是一种简洁的表达式，它允许在一行代码中根据一个布尔表达式的值来选择两个值中的一个。三元操作符的基本语法是:

```
condition ? valueIfTrue : valueIfFalse;
```

例如，x > y ? x : y 会返回 x 和 y 中较大的值。

这个表达式的执行流程如下：

1. 首先计算 condition 表达式，这是一个布尔表达式，如果为 true，则执行后续步骤。
2. 如果 condition 为 true，则返回 valueIfTrue 表达式的值。
3. 如果 condition 为 false，则返回 valueIfFalse 表达式的值。

接下来，看一下三元操作符的例子。

```
public class TernaryOperatorExample {
    public static void main(String[] args) {
        // 定义一个分数变量
        int score = 85;

        // 使用三元操作符计算等级
        String grade = (score >= 90)
            ? "A" : (score >= 80) ? "B" : "C";

        // 输出计算得到的等级
        System.out.println(" 分数: " + score + ", 等级: "
                                                + grade);
    }
}
```

上面的代码运行以后，输出结果如下：

```
分数: 85, 等级: B
```

当运行这个程序时，它会根据给定的分数（在这个例子中是 85）使用三元操作符计算并输出相应的等级。在本例中，由于 score 是 85，所以条件表达式 (score >= 80) 为 true，因此返回的 grade 值将是 “B”。

提示

最后，我想分享一个关于三元操作符的历史小知识。三元操作符最早起源于上世纪 60 年代初期由英国剑桥大学开发的 CPL（Combined Programming Language）语言。虽然 CPL 语言本身并未得到广泛应用，但它的简化版本 BCPL（Basic Combined Programming Language）却影响深远。BCPL 是世界上第一个使用大括号的语言，它影响了后来的 B 语言和 C 语言，并间接影响了包括 Java 在内的大量现代编程语言。

B 语言是 BCPL 的进一步简化，它被用于开发了第一个大型软件 UNIX。后来，B 语言

经过改进，最终演变为我们今天所熟知的 C 语言。C 语言将三元操作符发扬光大，并将其传承给了 Java 等后来的编程语言。

在研究编程语法的时候，我总是被这种薪火相传的事情感动。在短暂的编程历史中，无数的语言仅仅发出了微弱的光芒，随即便消失在了历史的长河中。正是这微小的光芒，汇集起来，形成了今天主流的编程语言，每一门语言都是站在前人的肩膀上发展起来的。

5.8 操作符的优先级

在 Java 中，操作符的优先级决定了不同操作符在表达式中的执行顺序。当一个表达式中包含多个操作符时，优先级决定了各个操作的执行顺序。例如，在 a + b * c 中，乘法具有高于加法的优先级，因此会先计算 b * c 再与 a 相加。可以使用括号来改变操作符的优先级。掌握 Java 操作符的优先级是编程技能的基础部分，它能帮助开发者编写更安全、高效和易于维护的代码，否则可能会导致代码行为与预期不符。

以下是一个按照优先级从高到低排列的 Java 操作符列表：

1. 圆括号 ()：最高优先级，用于改变运算顺序或调用方法、构造函数等。
2. 后缀自增/自减 (++ 和 –)：对于变量进行后缀递增或递减操作，如 x++ 或 y–，先使用当前值，然后增加或减少。
3. 一元正负号 (+ 和 -)、逻辑非 (!)、按位取反 (~)：对操作数进行加法相反数、逻辑否定或比特位翻转。
4. 乘法 (*)、除法 (/)、模 (%)：进行数值类型的乘法、除法和求余运算。
5. 加法 (+)、减法 (-)：进行数值类型的加法和减法运算。
6. 左移 («)、右移 (»)、无符号右移 (»>)：对整型数据进行位移操作。
7. 小于 (<)、大于 (>), 小于等于 (<=)、大于等于 (>=)：比较两个操作数的关系。
8. 等于 (==)、不等于 (!=)：检查两个操作数是否相等或不相等。
9. 按位与 (&)：对两个整数进行按位与运算。
10. 按位异或 (^)：对两个整数进行按位异或运算。
11. 按位或 (|)：对两个整数进行按位或运算。
12. 条件 AND (&&)：逻辑与运算，如果两边的操作数都为真，则结果为真；否则为假，并且遵循短路原则（左侧为假时，不会计算右侧）。
13. 条件 OR (||)：逻辑或运算，如果任意一边的操作数为真，则结果为真；否则为假，并且同样遵循短路原则（左侧为真时，不会计算右侧）。
14. 条件运算符（三目运算符）：condition ? valueIfTrue : valueIfFalse，根据条件决定返回哪个值。
15. 赋值运算符 (=、+=、-=、*=、/=、%=、«=、»=、»>=、&=、^=、|=)：赋值和复合赋值

操作。

了解这些优先级有助于编写更准确、可预测的代码，尤其是在没有使用括号明确指定执行顺序的情况下。当多个操作符在同一表达式中出现时，优先级高的会先执行。

5.9 本章附录

5.9.1 比较两个浮点数是否相同常用的几种方法是什么?

当我们在计算机中使用浮点数时，需要记住的一件事是，这些浮点数只是对实数的近似。这是因为计算机内部使用二进制来表示数字，而有些十进制数无法用有限的二进制位精确表示。浮点数的表示有固定的精度，这意味着有些数字不能精确地用浮点数表示。这就是为什么我们无法仅仅使用 == 运算符来比较两个浮点数是否相等。

我在本章的正文中已经举了一个使用 epsilon 来比较的例子，这是最简单也是最常用的浮点数比较方法。原则是计算两个浮点数之间的差的绝对值，如果这个绝对值小于某个非常小的数（epsilon），那么我们就认为这两个浮点数是相等的。epsilon 是一个非常小的正数，通常是一个很小的值，例如 1e-6 或 1e-8，epsilon 的值取决于具体的应用场景和所需的精度。

还有一种方法是通过相对误差来判断两个浮点数是否相等。相对误差指的是两个数之间差值与其中一个数的比值，而相对容差则是允许的相对误差的最大值。这种方法的原理是：创建一个方法来计算两个浮点数之间的差的绝对值，然后将其与它们的最大绝对值的一个小比例（相对容差）进行比较。相对容差的值通常是一个很小的值，例如 1e-6 或 1e-8，具体取决于所需的精度。如果差的绝对值小于或等于这个小比例，那么可以认为这两个浮点数是相等的。下面的代码是使用相对误差法判断两个浮点数是否相等的例子：

```
public class Main {
    public static void main(String[] args) {
        double num1 = 0.1 * 0.1;
        double num2 = 0.01;

        // 返回 true
        System.out.println(isClose(num1, num2, 1e-9));
    }

    public static boolean isClose(double a,
                                  double b, double relTol) {
        return Math.abs(a - b)
```

```
            <= relTol * Math.max(Math.abs(a), Math.abs(b));
    }
}
```

这段 Java 代码定义了一个叫做 isClose 的函数，用于比较两个浮点数是否相等，或者足够接近以至于可以被认为是相等的。该函数接受三个参数：两个要比较的浮点数 a 和 b，以及一个表示相对容差的浮点数 relTol。该函数首先计算两个浮点数之间的差的绝对值，然后将其与它们的最大绝对值乘以相对容差的结果进行比较。如果差的绝对值小于或等于这个结果，则认为这两个浮点数是相等的。

选择合适的 relTol 值至关重要，因为它会直接影响比较结果的准确性。你可能需要根据你的具体应用来选择一个合适的 relTol 值。如果你正在处理非常大或非常小的数，或者你需要非常高的精度，那么你可能需要选择一个更小的 relTol 值。

还有一种偶尔可见的方法，称为 ULP（Unit in the Last Place）方法。ULP 是指相邻浮点数之间的距离，也称为最小精度单位。这种方法的原理是：由于浮点数在计算机中的表示是离散的，相邻浮点数之间的差值是固定的。这个差值通常由浮点数的尾数位精度确定，并且在浮点数的有效范围内是一个固定的值，可以通过一些数学公式计算得出，也可以使用一些现成的库函数获取。

使用 ULP 方法可以更精确地判断两个浮点数之间的差异，但它只能比较相同类型的浮点数，不能混合比较 float 和 double 类型的浮点数。这是因为不同类型的浮点数具有不同的精度和表示方式，因此它们的 ULP 值也不同。相比于前面介绍的 epsilon 比较法和相对误差法，ULP 方法更难理解，需要对浮点数在计算机中的表示方式有一定的了解。幸好几乎所有的编程语言都内置了计算 ULP 的方法，Java 也不例外。下面的代码是使用 ULP 方法比较两个浮点数是否相等。

```
public class ULPComparisonExample {
    public static void main(String[] args) {
        double num1 = 0.1 * 0.1;
        double num2 = 0.01;

        double difference = Math.abs(num1 - num2);

        double ulp = Math.ulp(num1);

        if (difference <= ulp) {
            System.out.println(" 两个数相等");
        } else {
```

```
                System.out.println(" 两个数不等");
            }
        }
    }
```

这段代码使用 ULP 方法比较两个浮点数 num1 和 num2 是否实质上相等。“实质相等” 指的是两个数之间的差异是否小于或等于它们之间最小精度单位（ULP）的值。Math.ulp(num1) 返回 num1 的 ULP 值，即与 num1 最接近的浮点数之间的距离。

在实际应用中，ULP 方法常用于浮点数比较时避免由于精度损失导致的误判。需要注意的是，ULP 方法并不能保证两个浮点数在数学上完全相等，它只能保证它们之间的差异非常小，通常小于浮点数表示的精度。

以上代码运行以后输出的结果为：两个数相等

综上所述，比较两个浮点数是否相等并不容易。本章介绍了三种常用的方法：epsilon 比较法、相对误差法和 ULP 方法。大部分情况下，使用 epsilon 比较法应该能满足要求。应该根据具体的应用场景和所需的精度选择合适的方法。除了这里介绍的三种方法，一些第三方库也提供了相应的比较浮点数是否相等的方法，例如 Apache Commons Math 库中的 Precision.equals() 方法和 Google Guava 库中的 DoubleMath.fuzzyEquals() 方法。这些方法通常提供了更多的选项和更高的精度，可以根据具体的需求进行选择。

5.10 程序员故事

《老人与手机》

我看了一下今天的工作单：“乘园镇，中岗村，固定电话更换移动电话业务”。最近这几年，固定电话业务萎缩的非常厉害，公司大力推进固定电话改移动电话业务，人手不够，只要是有手有脚的人都要以此为主要工作。我是程序员，有手有脚，所以最近我做了很多这样的 “外单”。

相比于城里格子间里的工作，我很喜欢 “外单”。现在初春时节，城里虽然也有点点的花红柳绿，但这不过是虚虚的应个景儿。只有驱车到城外，满山的杜鹃轰轰烈烈的开，那灼眼的红色，一路摧枯拉朽的从路边烧到山顶。

车子在火海中穿行，经过的却是斑驳陆离的村庄，这种强烈的色彩对比给我强烈的视觉冲击。在这个装了橡胶轮子奔驰的世界里，城市化已经让村庄荒废了。我拨通了留下的电话，接电话的人告诉我他是帮另一位老大爷打的电话，并且告诉了我地址。我沿着村里仅有的一条小街道走到尽头，来到了那个看起来随时都要倒下的房子前。当我敲门的时候，门板剥落的油漆与灰尘不住的飘落。院里响起了一阵狗叫，随后一个满头白发的老人打开了门，一条狗安静的站在他身后，它静静的看着我，毫无敌意。

“我是来修电话的。”我说。那个老人笑了。

“好好好，进来吧，我不会用我的手机，我还是想用那个固定电话。”

他带我来到了一间卧室和客厅共用的狭小的房子里：“我现在一个人住，我那口子去年走了。我家里的电话只用来和我儿子联系，他在深圳打工。”他边说边指着桌子上的固定电话。

贫穷的证据随处可见。破旧的家具，一个暖水瓶。电器只有一个电灯和一部电话，老人的早饭还在桌上，一块馒头，几片咸菜。狗吃的早饭和老人是一样的，也是馒头，放在桌脚的碗里。

“我给我儿子打电话打不通了。”老人开口了。

“我知道，现在固定电话业务取消了，也就是说，这台电话不能用了。”当我说出这句话的时候，我看到老人有脸上流露出疑惑与失望的表情。我继续小心的说道：“你可以用手机。”

“我不会用手机。”老人回答。

“你可以到最近的营业厅里办一个每个月 3 元的套餐，营业厅里会送一台手机给你。”当我说这句话的时候，老人脸上的表情更疑惑了，他低声的问：“去哪里办理？”

“来吧，我开车带你去，你带上身份证。”当老人要上车的时候，他的狗一直跟着他，焦急的围着他转圈圈。我看出了它的焦虑，就让它和主人一起在后排，它安静的趴在老人的脚下。

到了营业厅，办了一个卡，我还要教老人如何使用手机。他不记得他儿子的电话号码，我又开车送到他家里，期间还折返了一次，买了一个插线板，老人家里没有给手机充电的插线板。

他从破旧的桌子里掏出一张纸，纸上记着他儿子的电话号码，我拨通了这个电话，跟他儿子说明了情况。当他跟他儿子通话的时候，我看到他的眼睛恢复了光亮。最后，我花了一些时间教他如何打电话，并且把我的电话号码也存在了那个手机上，告诉他，如果有问题，就随时联系我。

“我要付你多少钱？”

“啊，不用付钱。”我很快的说：“没什么，我只是路过这里，也不麻烦。”

老人有些吃惊：“但是你又买了这个。”他指了指那个插线板。

“没什么，这个不花什么钱的。”我说了再见，就走出了他的家门。老人和他的狗一直站在门口，我从后视镜里一直看到我的车子转弯，他们都没有离开。

一路上，我再也没有心思去看那鲜艳的杜鹃花，我的眼里只有那个破败的家，那个老人和那条狗。

后来，春节的一天，我接到一个电话，是他儿子打来的，老人让他儿子给我送一些家里种的花生。后来，我总会想到这个老人，不管我们认为多么简单易用的手机或者软件，对其他人可能真的难于上青天。做软件也好，做硬件也好，多考虑一下其他人。

6 结构化编程

Contents

> 软件危机的主要原因，很不客气地说：在没有机器的时候，编程根本不是问题；当我们有了计算机，编程开始变成问题；而现在我们有巨大的计算机，编程就成为了一个同样巨大的问题。
>
> —— Edsger Dijkstra 在图灵奖颁奖典礼上的演讲

在编程范式的发展过程中，面向过程编程（Procedural Programming）作为早期主流的编程思想，在软件开发中起到了奠基性的作用。例如 C 语言、Fortran 语言等。它强调程序设计以一系列可重复使用的函数或过程为核心，通过明确的步骤和指令序列来解决问题。面向过程编程注重的是数据处理的过程，开发者通常会将任务分解为一系列独立的功能单元，并按逻辑顺序调用这些功能单元来完成整体任务。面向过程编程的优点在于结构清晰、易于理解和调试，适合解决一些规模较小、逻辑简单的问题。

随着软件复杂度的增加和技术的进步，面向对象编程（Object-Oriented Programming, OOP）逐渐成为主导趋势。OOP 引入了类、对象、封装、继承以及多态等概念，使得代码组织更加模块化，更易于复用与维护，因为面向对象编程可以将数据和操作封装在一起，提高代码的复用性和可维护性。然而，即使在全面采用面向对象编程体系的项目中，面向过程编程的原则和实践依然具有不可忽视的价值。例如，在一些算法实现或底层功能模块中，仍然会使用面向过程的编程方式。

面向对象编程和面向过程编程并非相互排斥，而是可以相互融合。实际上，面向对象的设计中往往包含了面向过程的元素。例如，在类的方法内部实现特定功能时，仍然需要遵循清晰的过程逻辑；而在没有必要引入复杂类结构的情况下，简单的函数式或者过程式的解决方案可能更为高效和直观。此外，许多现代面向对象语言支持混合使用面向过程与面向对象的特性，例如，Python 语言既支持面向对象编程，也支持函数式编程，允许开发者根据实际需求灵活选择最适合的设计模式。

因此，在采用面向对象编程的时候，面向过程编程依然是十分重要的。它强调逻辑思维和问题分解能力，是编程思维的基础，对于理解问题并将其分解为可执行步骤至关重要。同时，面向过程编程的方法论在处理简单、直接的任务以及在面向对象模型内部的具体实现时，都将继续发挥着不可或缺的作用。在这些场景下，面向过程编程可以提供更简洁、更高效的解决方案。

面向过程编程强调的是结构化编程思想，例如 C 语言、Pascal 语言等。通过顺序结构、选择结构和循环结构将程序划分为一系列可重用的过程或函数，每个过程都负责完成一个具体任务。开发者通过调用这些过程并传递参数来构建更复杂的程序逻辑，从而实现问题的解决和功能的实现。结构化编程可以提高代码的可读性和可维护性，减少代码的复杂度。接下来，我们将一起学习结构化编程的思想。

6.1 结构化编程的由来

在软件工程的发展历程中，有两个公认的转折点被视作软件危机的重要阶段。首次软件危机发生在 20 世纪 60 至 70 年代期间，其中标志性事件之一是 IBM System/360 大型计算机系统的开发过程中所遭遇的困境。

由于当时缺乏有效的软件工程方法和管理工具，导致项目规模难以控制，代码质量难以保证，最终项目成本严重超出预算，并且交付时间一再延期。这次危机暴露了传统编程方法在处理复杂软件项目时的局限性，并促使人们寻求新的软件开发方法，例如结构化编程和面向对象编程。

面对这场首度爆发的软件危机，Edsger Dijkstra 等科学家提出了结构化编程的理念，这是一套以面向过程为核心的设计哲学。它通过采用 “自顶向下、逐步分解” 的原则，将复杂的问题分解为一系列简单、独立的子问题，然后逐个解决。从高层次抽象开始，逐步细化和分解问题，直到达到足够小的单元，然后再逐步实现这些单元。结构化编程强调使用顺

序结构、选择结构和循环结构来组织代码，避免使用 goto 语句，旨在将软件系统的复杂性控制在可管理的水平内，从而从整体上有效降低软件开发和维护的难度。

不得不提及的是 Dijkstra 的经典论文《GOTO 语句的危害》(“Go To Statement Considered Harmful”)。在这篇振聋发聩的论文中，Dijkstra 提出使用结构化编程取代 goto 语句的革命性想法。他认为，goto 语句会导致程序控制流混乱、难以理解和维护，从而增加程序的复杂性与错误风险。相反，结构化编程中的顺序、选择和循环结构，能够使程序逻辑清晰易懂，大大提升程序的可读性和可维护性。

Dijkstra 的这篇论文犹如一块巨石投入平静的湖面，在当时的计算机科学界掀起了巨大的波澜，成为了结构化编程兴起的关键契机。它促使人们重新思考编程的本质，推动了结构化编程范式的迅速发展和普及。结构化编程由此成为了一种标准的软件设计方法，并为后续的软件工程发展奠定了坚实的基础。当我们谈及结构化编程范式时，通常指的是一种以结构化的思维方式来组织和编写程序的方法论。其核心内容主要包括：

顺序结构 (Sequential Structure): 代码按照预定的顺序逐行执行，从程序的起点开始，直到终点结束。这种线性的执行路径清晰易懂，便于理解和调试。

选择结构 (Selection Structure): 根据不同的条件，选择性地执行相应的代码块。常见的形式包括 if 语句和 switch 语句，它们赋予程序分支选择的能力，使其能够根据不同的情况做出不同的响应。

循环结构 (Iteration Structure): 重复执行一段代码块，直到满足特定的终止条件。常见的形式包括 for 循环、while 循环和 do-while 循环。它们为程序提供了重复执行的能力，有效地处理需要迭代处理的任务。

这套革命性的方法论在 20 世纪 70 年代迅速席卷软件开发行业，成为主流趋势，并有效地化解了当时的软件危机。正因其在推进计算机科学与软件工程领域做出的卓越贡献，Edsger Dijkstra 于 1972 年被授予了计算机科学的最高荣誉——图灵奖。接下来，我们将深入探讨这三种控制结构的核心概念。

6.2 顺序结构

顺序结构是编程语言中最基本的控制结构之一，它体现了代码执行的线性顺序。每个语句都按照编写的先后顺序依次执行，如同一条笔直的河流，从源头流向终点。常见的顺序结构语句包括赋值语句、方法调用语句、输入输出语句等。由于其简单直观的特性，编程时主要考虑不同语句之间的依赖关系，确保执行顺序符合逻辑。接下来，下面的代码展示的就是顺序结构的示例。

```java
public class SequenceExample {
    public static void main(String[] args) {
        // 定义变量并初始化
        int a = 5;
        int b = 7;

        // 顺序执行多个操作
        int sum = a + b; // 第一个操作：计算和
        int product = a * b; // 第二个操作：计算积
        int difference = a - b; // 第三个操作：计算差

        // 输出结果
        System.out.println(" 两数之和为： " + sum);
        System.out.println(" 两数之积为： " + product);
        System.out.println(" 两数之差为： " + difference);
    }
}
```

我读初中的时候，山东省的语文教材里有一篇华罗庚先生写的《统筹方法》，那篇课文里，以烧开水泡茶为例，讲了如何把一项任务进行合理的安排。我们在写程序的时候，也要尽量考虑不同语句的依赖性与合理性。

我记得在那篇文章里，将烧开水泡茶分成了洗水壶，烧开水，洗茶壶，洗茶杯，拿茶叶等几个步骤，其中有几项的顺序是不能颠倒的，比如不能烧开水以后再洗水壶。这样的逻辑顺序在编程中同样适用，如果忽视这种顺序性，可能会导致程序错误或效率低下。

尽管顺序执行的代码结构相对直观且易于理解，但精心设计的代码执行顺序却能显著提升代码质量、可维护性和正确性。在实际编程中，代码块的执行顺序可以分为两类：第一类是前后顺序无关的代码，这些代码相互独立，其执行顺序不会影响程序结果；第二类是有明确顺序的代码，具有严格时间或逻辑依赖关系的部分，其执行顺序至关重要。理解并恰当处理这两类代码块之间的关系，是实现高效、稳健程序设计的关键所在。接下来，我们一起学习这两类代码：

6.2.1 前后顺序无关的代码

有些语句的执行顺序并不会影响程序的最终结果，它们在逻辑上与其他语句相互独立。编写这类代码时，应注重代码的可读性和流畅性，尽量将相关的语句组织在一起，以便于理解和维护。例如，假设我们正在开发一个超市管理系统，并需要生成报表，其中一些数据的计算顺序并不会影响最终的报表结果。

```
DrinksData drinksData; // 酒水数据
MeatData meatData; // 肉类数据
SportsData sportsData; //运动品数据

drinksData.computeMonthly(); //统计酒水月销量
meatData.computeMonthly(); //统计肉类月销量
sportsData.computeMonthly(); //统计运动品月销量

drinksData.computeAnnula(); //统计酒水年销量
meatData.computeAnnula(); //统计肉类年销量
sportsData.computeAnnula(); //统计运动品年销量

drinksData.print(); //输出酒水报表
meatData.print(); //输出肉类报表
sportsData.print(); //输出运动器报表
```

上述代码片段展示了生成三种商品报表的过程，分别针对酒水、肉类和运动品数据进行处理。尽管每种商品的报表生成过程没有严格的先后顺序，但初始代码将不同商品的数据处理过程交错在一起，导致代码的可读性和可维护性较差。例如，如果需要修改运动品报表的输出方式，则需要在整个代码中搜索相关的代码片段，十分不便。

为了提高代码的可读性和可维护性，我们可以将每种商品相关的代码组织在一起，形成清晰的代码块。优化后的代码如下所示：

```
DrinksData drinksData; // 酒水数据
drinksData.computeMonthly(); //统计酒水月销量
drinksData.computeAnnula(); //统计酒水年销量
drinksData.print(); //输出酒水报表

MeatData meatData; // 肉类数据
meatData.computeMonthly(); //统计肉类月销量
meatData.computeAnnula(); //统计肉类年销量
meatData.print(); //输出肉类报表

SportsData sportsData; //运动品数据
sportsData.computeMonthly(); //统计运动品月销量
sportsData.computeAnnula(); //统计运动品年销量
```

```
sportsData.print(); //输出运动器报表
```

优化后的代码将每种商品的数据处理过程集中在一起，形成独立的代码块，提高了代码的模块化程度。这种组织方式使得代码更容易理解和维护，也更容易进行后续的修改和扩展。此外，这种组织方式还暗示了代码可以进一步分解为三个独立的子程序，分别处理酒水、肉类和运动品数据，进一步提高代码的可重用性。

评估代码中顺序无关部分排列是否合理的一个有效方法是，观察阅读代码时的流畅度。当代码组织得井井有条时，程序员在理解某一逻辑片段时，视线能够在相关代码段附近自然流动，无需在大量代码间来回跳转寻找所需信息。反之，如果代码排列杂乱无章，则会迫使读者频繁地在不同的代码段之间跳转，才能理解代码的逻辑，降低阅读效率和理解难度。

在软件重构过程中，为了提升代码的可读性和可维护性，可以考虑将高度关联的语句封装成独立的子程序或函数。这种做法不仅能够强化代码的模块化结构，还有助于降低复杂度、消除冗余，并确保每个子程序专注于单一职责，从而提升整体代码质量和开发效率。

6.2.2 有明确顺序的代码

顺序结构中的代码按照编写时从上到下的自然顺序依次执行，如同一条流水线，每个语句的执行都依赖于前一个语句的完成。在这种结构中，程序员通过精心设计程序流程，确保了不同操作之间的逻辑依赖关系得以正确体现，即后一个操作通常需要前一个操作的结果作为输入或先决条件，从而保证程序的正确性和可靠性。

例如，在 Java 编程中，初始化变量、调用方法以及进行一系列计算等操作，如果存在明显的前后依赖关系，则必须严格遵循特定的执行顺序，否则程序将无法得到预期结果，甚至可能出现错误或异常。下面的代码演示的就是有明确顺序的结构。

```
public class SequenceExample1 {
    public static void main(String[] args) {
        // 1. 初始化变量

        int a = 5;
        int b = 3;

        // 2. 使用已初始化的变量进行计算（依赖于第 1 步）

        int sum = a + b;
        int product = a * b;
```

```
        // 3. 显示结果（依赖于第 2 步的计算）

        System.out.println(" 两数之和: " + sum);
        System.out.println(" 两数之积: " + product);

        // 4. 更改变量 a 的值
        // 不影响之前步骤的结果，但影响后续使用 a 的语句

        a = 7;

        // 5. 又一次使用更新后的变量 a 进行新的计算
        // 依赖于第 4 步的更改

        int newSum = a + b;
        System.out.println(" 变量值改变后两数之和: " + newSum);
    }
}
```

以上代码示例清晰地展示了顺序结构的特点：代码按照从上到下的顺序依次执行，每个步骤都依赖于前一个步骤的结果。变量的初始化为后续的计算提供了必要的输入，而计算结果又作为输出的基础。即使是变量值的修改，也影响了后续的计算结果。这种明确的执行顺序确保了程序逻辑的正确性和一致性。

6.3 选择结构

提示

老婆给当程序员的老公打电话："下班顺路买十个包子，如果看到卖西瓜的，买一个。"

当晚老公手捧一个包子进了家门，老婆怒道："你怎么只买一个包子？"

老公甚恐，喃喃道："因为我看到卖西瓜的了。"

在日常的表达中，选择结构的对话无处不在。比如两千多年前的孔子说过："富而可求也，虽执鞭之士，吾亦为之。如不可求，从吾所好。"这句话就是选择结构，孔子的意思是：如果富贵可求的话，那就算拿着鞭子给人驾车，我也干。如果富贵不可求，那还不如

干点自己喜欢的事情呢。这句话，我们可以用如下的流程图来表示：

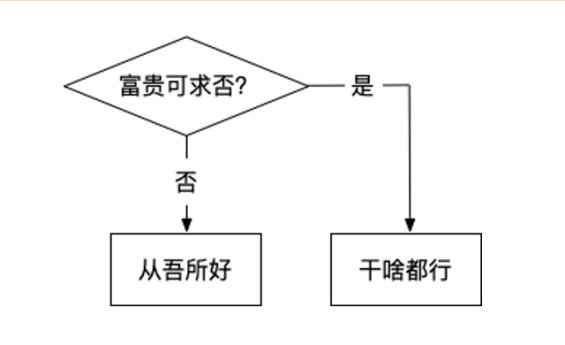

图 6.1: 日常表达中的选择结构

值得一提的是，上述代码片段的执行流程可以借助程序流程图 (Program Flowchart) 进行可视化展示。程序流程图是一种强大的图形化工具，它使用标准的图形符号表示程序中的不同操作，从而帮助程序员更好地理解和设计程序。程序流程图使用一系列标准的图形符号来表示不同的操作和控制流程，例如：

o 矩形框 (Rectangle): 表示程序的开始或结束步骤，以及普通的处理步骤。o 箭头 (Arrow): 表示程序的执行流程方向，指示下一步执行的步骤。o 菱形框 (Diamond): 表示判断或条件结构，通常包含一个条件表达式，根据条件的真假决定下一步执行的路径。o 椭圆框 (Oval): 表示输入或输出操作，例如从用户获取输入或将结果显示到屏幕上。

选择结构的历史可以追溯到早期计算机编程语言的萌芽阶段。在汇编语言和低级语言时代，程序员需要借助跳转指令来实现条件判断和分支操作，这种方式缺乏直观性和可读性，容易导致程序逻辑混乱和错误频发。随着高级编程语言的兴起和发展，选择语句逐渐成为编程语言的标配，为程序员提供了更加灵活和直观的条件判断和分支控制手段。

在 Java 中，选择结构主要通过 if 语句和 switch 语句来实现。它们为程序员提供了灵活的条件判断和分支控制能力，使程序能够根据不同的条件执行不同的代码块，从而实现更加丰富的逻辑功能。

如果你对选择结构的历史演变充满好奇，可以参阅下面关于选择结构发展历程的考证文章，了解更多精彩内容。

注释

选择结构的历史

选择结构在编程领域拥有悠久的历史。由于篇幅限制，我将简略介绍其发展历程，并提供相关资料供有兴趣深入了解的读者参考。

我曾读过一份名为《A Study of Inverse Interpolation of the Eniac》的文档，该文档写于 1946 年，现已在网络上公开。在文档的第 22 页，介绍了一种比较两个数大小的方法。如今，我们可能认为比较两个数大小是理所当然的事情，但对于早期的计算机而言，让它们理解 2 大于 1 并非易事。如果没有系统学习过计算机组成原理，要仅凭自身理解并实现这一功能需要非凡的智慧。

选择结构的运行依赖于比较操作，幸运的是，现代计算机已经在硬件层面解决了这一问题，为我们提供了便捷的比较指令。

在学习计算机组成原理时，我们了解到一种重要的寄存器——条件码寄存器。条件码寄存器的状态由处理器执行指令的结果决定，其各个标志位会根据算术或逻辑指令的结果进行设置或清除。通过读取这些标志位，我们可以获得计算后的结果状态，例如是否溢出、结果是否为零、是否为负数等。

条件码寄存器是编程语言执行选择结构（如 if 语句）和循环结构（如 while 循环）的硬件基础。如果没有这个基础，程序就无法根据计算结果进行判断，从而无法进行跳转或循环操作。

在硬件的支持下，随后的计算机编程语言几乎都引入了选择结构语句。然而，早期的选择结构语句与我们现在使用的形式存在着较大的差异。例如，在早期的 Fortran 语言中，选择结构是这样的：

```
IF(a) n1 n2 n3
```

在这段代码中，a 代表一个整数，其值可能为负数、零或正数，分别对应后面的 n1、n2 和 n3 语句块。这种选择结构与现代化的形式相差甚远。直到 1977 年，Fortran 语言才正式引入 IF—THEN—ELSE 的控制结构，为程序员提供了更清晰、更灵活的条件判断方式。

1958 年，计算机科学界意识到编程语言需要做到与机器无关，于是美国和德国的两个团队开始着手进行计算机语言的规范化工作。两个团队分别独立工作，德国团队提交了一份德文材料，而美国团队则提交了一份英文材料。

这时，语言的重要性就显现出来了。德文材料需要翻译成英文，而当时机器翻译技术尚未成熟，翻译版的英文材料可能存在错误。雪上加霜的是，原始的德文材料也遗失了，导致诸如 “else” 这样的关键字的提出者至今无法考证。

如果您想查看这份文档，可以搜索 “Proposal for a Universal Language for the Description of Computing Processes”。文档的开头就明确表示，由于时间紧迫，任务繁重，翻译可能存在错误或遗漏。这段历史也反映了早期计算机科学发展过程中语言交流和资料保存的重要性。

美国团队由 13 位杰出成员组成，他们共同打造的解决方案孕育了一门划时代的语言——ALGOL。更令人瞩目的是，该团队中有 3 位成员分别荣获了图灵奖，而获奖原因皆与 ALGOL 语言的贡献密不可分。ALGOL 的规范化进程对整个计算机编程语言领域产生了深远的影响。根据我的经验，追溯各种语言特征或关键字的起源，十之八九都与 ALGOL 有着千丝万缕的联系，因此 ALGOL 被誉为 “现代编程语言的鼻祖” 可谓实至名归。例如，本章将要探讨的选择结构和循环结构，以及前几章介绍的各类操作符、代码块、变量名和变量作用域等概念，均可追溯到 ALGOL 语言的演化历程。美国团队的珍贵文档保存完好，你只需搜索 “ACM ALGOL Proposal” 即可获取这份宝贵的资料。

或许有些读者会疑惑：阅读这些文档有何意义？若从纯粹的兴趣出发，最大的价值莫过于文档本身的趣味性。通过研读这些文档，你将体验到拨云见日般的豁然开朗，并惊叹于那些看似不起眼的功能背后，竟然蕴藏着如此深思熟虑的权衡与取舍。

从实用主义的角度来看，阅读这些文档能够让你深刻感受到，在先驱者们奠定的基础之上，后来的编程语言只能在有限的范围内进行 “锦上添花” 式的创新。例如，自从 ALGOL 定义了选择结构之后，后续的语言为了彰显自身特色，往往只能在关键字的选择上做文章。当别人使用 “else if” 时，它们便另辟蹊径，选择 “elsif” 或 “elif” 来体现差异。

6.3.1 if 语句

在 Java 语言中，if 语句作为选择结构的基石，它根据特定条件来控制程序的执行流程。else 子句的出现为 if 语句提供了更丰富的表达能力，使其能够处理更加复杂的条件分支。基于此，if 语句衍生出三种主要形式：

第一种是简洁的单分支选择语句，仅当给定布尔表达式的值为真时执行相应的代码块。

第二种是经典的双分支选择语句——if-else 结构，程序会根据布尔表达式的真假选择执行不同的代码路径。

最后一种是功能强大的多分支选择语句，它通过 if-else if 的形式，能够处理多个可能的条件分支，并按顺序逐一匹配，执行相应的代码段。

接下来，我们将深入剖析这三种不同形式的 if 选择语句的语法结构和应用场景。

6.3.1.1 单分支 if 语句

单分支 if 语句是 Java 编程语言中最基本的控制结构之一，它根据指定的布尔表达式结果来决定是否执行后续代码块。当布尔表达式的值为 true 时，执行紧随 if 语句后的代码块；反之，当布尔表达式的值为 false 时，则直接跳过该代码块，继续执行后续代码。下面演示代码中的分支结构为单分支 if 语句。

```
public class StudentGradeChecker {
    public static void main(String[] args) {
        // 假设学生的考试成绩
        int studentScore = 85;

        // 判断是否及格（假设 60 分为及格线）
        if (studentScore >= 60) {
            System.out.println(" 学生已达到及格标准！");
        }

    }
}
```

以上代码示例清晰地展示了单分支 if 语句的应用。程序首先定义了一个变量来存储学生的考试成绩，然后使用 if 语句判断该成绩是否达到及格标准。如果条件满足，则输出相应的提示信息；否则，程序不进行任何操作。单分支 if 语句简洁明了地实现了根据条件进行选择的逻辑。

6.3.1.2 双分支 if-else 语句

双分支 if-else 语句是编程语言中常用的条件控制结构，它允许程序根据某个布尔表达式的真假执行不同的代码块，从而实现更加灵活的逻辑判断。以下是一个示例，展示了如何使用双分支 if-else 语句检查用户年龄并输出不同的信息，我们将这个示例封装在一个名为 AgeChecker 的类中，如下代码所示：

```
public class AgeChecker {
    public static void main(String[] args) {
        int userAge = 21; // 假设用户的年龄

        if (userAge >= 18) {
            System.out.println(" 您已成年");
```

```
        } else {
            System.out.println(" 您未满 18 岁");
        }
    }
}
```

以上代码示例展示了双分支 if-else 语句的应用。程序首先定义一个变量存储用户的年龄，然后使用 if-else 语句判断用户是否成年。根据判断结果，程序会输出不同的提示信息。双分支 if-else 语句有效地处理了两种互斥的情况，确保程序能够根据不同的条件做出正确的响应。

6.3.1.3 多分支 if-else if 语句

Java 中的多分支 if-else if 语句能够处理多个条件判断的情况，允许程序根据不同的条件执行相应的代码块。这种结构支持任意数量的 else if 子句，程序会依次检查每个条件，直到找到一个满足条件的分支并执行其对应的代码块，或者在所有条件都不满足的情况下执行可选的 else 子句。

以下是一个示例，展示了如何使用多分支 if-else if 语句根据考试成绩划分等级，我们将这个示例封装在一个名为 GradeCalculator 的类中，完整的代码如下所示：

```
public class GradeCalculator {
    public static void main(String[] args) {
        int score = 88; // 假设学生的分数

        if (score >= 90) {
            System.out.println(" 您的成绩是 A 级");
        } else if (score >= 80 && score < 90) {
            System.out.println(" 您的成绩是 B 级");
        } else if (score >= 70 && score < 80) {
            System.out.println(" 您的成绩是 C 级");
        } else if (score >= 60 && score < 70) {
            System.out.println(" 您的成绩是 D 级");
        } else {
            System.out.println(" 您的成绩是 F 级");
        }
    }
}
```

以上代码示例展示了多分支 if-else if 语句的应用。程序首先定义一个变量存储学生的考试成绩，然后使用 if-else if 语句根据不同的分数区间将成绩划分为 A、B、C、D、F 五个等级。程序会依次检查每个条件，并输出满足条件的分数等级。多分支 if-else if 语句有效地处理了多个互斥条件的情况，使得程序能够根据不同的输入做出不同的响应。

以上我们探讨了 Java 中三种类型的 if 选择结构，它们为程序提供了根据条件执行不同代码路径的能力。通过灵活组合布尔表达式和逻辑运算符，我们可以构建出适应复杂业务场景的分支结构。然而，当需要处理多个可能的离散值时，if 语句可能会导致代码冗长且难以维护，降低代码的可读性和可扩展性。

为了优化这种场景下的决策流程，并提高代码的简洁性和可读性，Java 提供了一种名为 switch 的特殊选择结构。与 if 语句不同，switch 语句允许开发者针对单一变量或表达式的多种可能取值进行集中式的检查，并根据匹配项执行相应的代码块。这种结构特别适用于基于枚举类型、字符或整数值做出决策的情况，能够有效地避免代码的冗余和复杂性。

接下来，我将详细剖析 switch 语句的工作原理及其相关特性，包括 case 标签、default 子句以及从 Java 后来引入的 switch 表达式等新功能。通过学习并掌握 switch 语句的使用，您将能够更好地应对涉及多分支选择逻辑的问题，从而编写出更为高效且易于维护的 Java 代码。

6.3.2 switch 语句

在 Java 编程语言的演进过程中，switch 语句作为处理多重条件判断的核心机制之一，始终伴随着 Java 的发展不断革新与完善。这一历程可以划分为两个鲜明的阶段：从早期广泛应用的传统 switch 语句，到随着技术演进而诞生的现代 switch 语句，每个阶段都见证了 switch 语句在功能和易用性方面的显著提升。

传统 switch 语句自 Java 诞生之初便已存在，已经成为开发者进行多路分支选择的标准工具。然而，随着 Java 社区对开发者需求的深入理解和持续改进，switch 语句开始经历一场深刻的变革，旨在提升其灵活性、易用性和功能性，使其能够更好地适应现代软件开发的需求。

随着 Java 的不断发展，switch 语句被赋予了一系列崭新的特性和优化语法，极大地简化了代码编写，扩展了其功能边界，使程序员能够更加自如地应对复杂多元的条件场景。switch 表达式和模式匹配等新特性，不仅显著提升了代码的简洁性和可读性，更赋予了 switch 语句强大的适应能力，使其在现代软件开发中焕发出新的活力。

本书将深入剖析传统与现代 switch 语句的演变历程，对比它们的语法特点、应用场景及优劣之处，并通过丰富的实例和实践案例，帮助读者全面了解和熟练掌握 Java switch 语句的历史变迁及其强大功能的应用方法。接下来，我们将首先通过下面的演示代码展示传统 switch 语句与现代 switch 语句的区别，随后分别对两者展开详细的讨论和分析。

```
public class SwitchExample {
    public static void main(String[] args) {
        String dayOfWeek = "Monday";

        // 使用传统 switch 语句
        switch (dayOfWeek) {
            case "Monday":
                System.out.println(" 周一");
                break;
            case "Tuesday":
            case "Wednesday":
            case "Thursday":
            case "Friday":
                System.out.println(" 工作日");
                break;
            case "Saturday":
            case "Sunday":
                System.out.println(" 周末了");
                break;
            default:
                System.out.println(" 无效的日期输入！");
                break;
        }

        // 使用 Java 12+ 的 switch 表达式
        // 有时被称之为简化版 switch
        // 在 case 标签中，使用了箭头（->）来指定当条件匹配时
        // 应该返回的值，而不需要使用 break 语句来终止执行。

        String status = switch (dayOfWeek) {
            case "Monday" -> " 周一";
            case "Tuesday", "Wednesday",
              "Thursday", "Friday" -> " 工作日";
            case "Saturday", "Sunday" -> " 周末";
            default -> " 无效的日期输入！";
        };
```

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

以上代码的输出结果是相同的，都是输出“周一”。

代码示例对比了传统 switch 语句和现代 switch 表达式在实现相同功能时的区别。传统 switch 语句需要使用 break 语句来防止代码执行落入下一个 case，而 switch 表达式则通过箭头操作符简洁地将每个 case 与其返回值关联起来，避免了 break 语句的使用，使得代码更加简洁易读。

接下来，我们将深入探讨传统 switch 语句和现代 switch 表达式的语法、特性以及应用场景，帮助读者更好地理解和使用这两种重要的控制流程结构。

6.3.2.1 传统的 switch 语句

Java 中的传统 switch 语句通过匹配表达式的值来选择执行不同的代码块，它为多重条件判断和分支控制提供了一种清晰简洁的实现方式。以下是传统 switch 语句的基本语法：

```
switch (expression) {
    case value1:
        // 当表达式的值等于 value1 时执行的代码
        break;
    case value2:
        // 当表达式的值等于 value2 时执行的代码
        break;
    // 更多的 case 标签...
    default:
        // 当表达式的值与任何 case 都不匹配时执行的代码
        break;
}
```

Java 的传统 switch 语句结构通常包含四个关键组成部分：表达式 (expression)、case 子句、break 语句以及可选的 default 子句。下面我们将逐一详细解析这四种元素在 switch 语句中的功能和作用：

表达式（expression）

这是 switch 语句的起点，通常是一个变量或可以计算出确定值的表达式。它的类型必须为整型 (byte, short, char, int)、枚举类型 (Enum) 或字符串类型 (String)。switch 语句会根据 expression 的值来决定执行哪个 case 分支。

case 语句

紧跟在 switch 语句之后，每个 case 子句后面跟着一个常量或枚举成员。当 switch 表达式的值与某个 case 后面的常量匹配时，控制流将跳转到该 case 对应的代码块执行。每个 case 子句都是 switch 结构内部的一个分支路径。

break 语句

每个 case 分支后通常都跟有一个 break 语句。一旦执行了与表达相匹配的 case 内的代码，break 语句将确保控制流立即跳出整个 switch 结构。如果省略了 break，程序将会继续执行下一个 case 分支，这种现象被称为“穿透”（fall-through），需要谨慎使用以避免意料之外的行为。

default 语句

这个分支是可选的，用于定义当表达式的值与任何 case 都不匹配时应执行的操作。在实际编程中，建议总是包含一个 default 分支，以便处理所有可能的输入情况，提高代码的健壮性和容错性。

在上述提到的 switch 语句四大构成要素中，需要深入剖析的是 break 语句与 default 语句的作用及其使用场景。

break 语句在 switch 结构中起着至关重要的作用。当某个 case 分支的条件得到满足时，其关联的代码块会被执行。然而，若该分支缺少 break 语句，控制流将会“穿透”到下一个 case 分支，并继续执行其代码，直到遇到 break 或到达 switch 语句的末尾。这种现象称为“穿透”（fall-through）行为。

虽然在特定情况下，穿透行为可以简化代码逻辑，但通常会导致意想不到的结果，降低代码的可读性和可维护性。因此，为了确保 switch 语句按照预期执行，建议在每个 case 分支的末尾添加 break 语句，以便在执行完匹配项对应的代码后立即跳出 switch 结构。接下来，演示一下代码“穿透”的情况。

```
public class PaymentCalculator {

    public String calculateDiscount(int orderSize,
                                    boolean isMember) {
        String discountType = "";

        switch (orderSize) {
            case 1:
                discountType = " 无折扣";
                // 假设这是对订单量为 2 且是会员的情况
```

```
            case 2:
                if (isMember) {
                    discountType = " 会员专享 9 折";
                } else {
                    discountType = " 普通客户 9.5 折";
                }
            case 3:
            case 4:
                discountType += ", 3 或 4 件额外享受 8.5 折";
            case 5:
            case 6:
                discountType += ", 5 或 6 件额外享受 8 折";
            default:
                break;
        }

        return discountType;
    }

    public static void main(String[] args) {
        PaymentCalculator calculator =
                                new PaymentCalculator();
        System.out.println(" 购买 2 件商品时的折扣类型: "
            + calculator.calculateDiscount(2, true));
    }
}
```

在这个例子中，运行之后会输出：购买 2 件商品时的折扣类型：会员专享 9 折, 3 或 4 件额外享受 8.5 折, 5 或 6 件额外享受 8 折。

在这个例子中，当用户购买 2 件商品且是会员时，原本预期只应用 “会员专享 9 折” 的折扣。然而，由于 case 2 后缺少 break 语句，导致控制流穿透到后续的 case，错误地将所有满足条件的折扣信息也一并返回。这个示例突出了在 switch 语句中正确使用 break 语句的重要性，以避免 “穿透现象” 导致的逻辑错误和结果不准确。

为了避免这种错误，我们在每个 case 分支后添加了 break 语句，确保代码按照预期执行，即只应用匹配项对应的折扣信息。

注释

穿透现象造成的 bug

“穿透现象” 在编程实践中确实容易成为 bug 的潜在源头，并可能导致严重的后果。一个著名的例子发生在 1990 年 1 月 15 日，美国 AT&T 公司遭遇了一场长达 9 个小时的大规模电话网络瘫痪事件。此次故障的根源在于一段运行于 4ESS 交换机系统中的 C 语言代码。这段代码包含一个 switch 语句，程序员原本希望利用 break 跳出循环结构，但由于误用，实际上跳出了 switch 语句块，引发了连锁反应，最终导致 AT&T 全国范围内的长途电话服务陷入瘫痪。这一事件充分说明了 switch 语句中正确使用 break 语句的重要性，以及 “穿透现象” 可能带来的巨大风险。

若想深入了解这一具有里程碑意义的 AT&T 网络崩溃事件，您可以尝试搜索 “The Crash of the AT&T Network in 1990”，以获取详尽的故障分析报告。这份报告将揭示事件的来龙去脉，展示程序员对 switch 语句 “穿透现象” 的理解和使用不当如何引发了灾难性的后果。它为后世程序员敲响了警钟，提醒我们在编程实践中，准确理解和合理运用每一种语言特性至关重要，否则即使是微小的疏忽也可能造成巨大的损失。

default 语句作为 switch 语句的一个可选组成部分，扮演着重要的 “兜底” 角色。它通常位于所有 case 子句之后，用于处理那些未被任何 case 子句匹配的情况。通过提供 default 分支，程序员可以为 switch 表达式的任何可能取值设定默认行为，从而增强代码的健壮性和完整性。无论 switch 表达式的实际值如何变化，default 语句都确保至少存在一个明确的执行路径，这对于提升程序的容错能力和应对未知输入状况至关重要。接下来的代码演示的就是这种情况。

```
public class DayOfWeekChecker {

    public static void main(String[] args) {
        // 假设这个值是不合法的，不在 0 到 6（代表一周七天）之间
        int dayOfWeek = 8;

        printDayOfWeek(dayOfWeek);
    }

    public static void printDayOfWeek(int dayOfWeek) {
        switch (dayOfWeek) {
```

```
                case 0:
                    System.out.println(" 今天是星期日。");
                    break;
                case 1:
                    System.out.println(" 今天是星期一。");
                    break;
                case 2:
                    System.out.println(" 今天是星期二。");
                    break;
                case 3:
                    System.out.println(" 今天是星期三。");
                    break;
                case 4:
                    System.out.println(" 今天是星期四。");
                    break;
                case 5:
                    System.out.println(" 今天是星期五。");
                    break;
                case 6:
                    System.out.println(" 今天是星期六。");
                    break;
                default:
                    // 当 dayOfWeek 不在 0 至 6 范围内时
                    /// 执行此 default 分支
                    System.out.println(" 无效输入");
                    break;
            }
        }
}
```

在这段代码中，当 dayOfWeek 的值不在 0 到 6 的范围内时，没有任何 case 分支的条件得到满足。此时，default 语句就会执行，打印一条提示信息，告知用户输入无效。总之，default 语句为 switch 语句提供了处理未匹配情况的机制，增强了代码的鲁棒性（robustness，我真是服了这个翻译，竟然流行开了，作为一个山东人，我看到鲁棒性，第一想法是山东棒槌:）和容错能力，确保程序在遇到意外输入时也能正常运行。

6.3.2.2 现代的 switch 语句

在 Java 编程语言中，传统的 switch 语句是一种特殊的流程控制工具，主要用于根据变量或表达式的值来执行不同的代码块。这种结构化的决策机制为程序员提供了一种简洁有效的方式来处理多分支逻辑。传统 switch 语句的运作原理基于精确匹配原则：它会逐个检查 case 标签后的常量值是否与给定表达式的值相等。一旦找到匹配项，则执行相应 case 内部的代码块。

然而，Java 中的传统 switch 语句功能相对有限。它仅适用于基本数据类型（如 int、char 和枚举），以及从 Java 7 开始支持的 String 类型。此外，每个 case 后面必须紧跟一个 break 语句（除非有意利用 fall-through 特性），以确保在执行完相应的代码后跳出 switch 结构。若缺少 break 语句，程序将 “穿透” 到下一个 case 继续执行，直到遇到 break 或到达 switch 语句的末尾，这可能导致意想不到的结果。

尽管 switch 语句能有效地替代多个 if-else-if 条件判断结构，但传统 switch 语句的应用场景主要局限于单一变量值的简单比较。在现代 Java 版本出现之前，它不支持更复杂的模式匹配和返回值功能。随着 Java 语言的不断演进，switch 语句的功能得到了显著扩展和增强。例如，引入 switch 表达式和全面模式匹配等新特性，使得 switch 语句在处理复杂条件分支时更为灵活和强大。

接下来，我们将展示现代 switch 语法中的两个重要改进：箭头语法和返回值功能。首先，让我们来看下面的代码。

```
public class SwitchDemo {

    // 计算斐波那契数列的第 n 个数
    public static int fib(int n) {
        return switch (n) {
            case 0 -> 0;
            case 1 -> 1;
            default -> fib(n - 1) + fib(n - 2);
        };
    }

    // 检查字符串是否包含所有指定的字符
    public static boolean contains(String str,
                                   char... chars) {
        return switch (str) {
            case null -> false;
```

```
            case "" -> false;
            default -> {
                for (char c : chars) {
                    if (!str.contains(String.valueOf(c))) {
                        yield false;
                    }
                }
                yield true;
            }
        };
    }

    // 添加一个 main 方法来测试上面两个 switch
    public static void main(String[] args) {

        // 演示斐波那契数列
        SwitchDemo sd = new SwitchDemo();
        System.out.println(sd.fib(5));

        // 示例字符串与字符数组
        String testString = "Hello, World!";
        char[] testChars = { 'H', 'e', 'l', 'o',
                                        'W', 'r', 'd' };

        // 调用 contains 方法并打印结果
        boolean result = contains(testString, testChars);
        System.out.println(" 字符串 '" + testString
                + "' 是否包含所有指定字符: " + result);
    }
}
```

上面的代码运行以后，输出的结果如下：

```
5
字符串 'Hello, World!' 是否包含所有指定字符: true
```

在上面的示例代码中，我们看到了 switch 语句的一种新形式，即使用箭头 (->) 来代替传统的冒号 (:)。箭头符号的作用是将 switch 语句的分支与其结果关联起来。在这种新形

式中，每个 case 后面跟着一个箭头 (->)，然后是该 case 的结果表达式或语句块。由于每个 case 只有一个结果，执行完毕后会自动结束，因此这种形式的 switch 语句不再需要 break 语句，也避免了“穿透”（fall-through）现象的发生。

第二个改进是支持使用 yield 语句来返回结果。在 Java 12 之前，switch 语句不能像函数那样直接返回结果，如果需要从 switch 语句中返回结果，必须使用 break 语句并将其与返回值结合使用。而现在，我们可以使用 yield 语句来更简洁地返回结果。

需要注意的是，yield 并不是一个严格的关键字，而是一个“保留的类型名”。这意味着你不能用 yield 作为类名、接口名或枚举名，但可以用它作为方法名、变量名或包名。这是因为 Java 设计者希望尽可能减少新特性对现有代码的影响。如果 yield 是一个严格的关键字，那么所有已经使用 yield 作为标识符的代码都需要修改。通过将 yield 设为“保留的类型名”，Java 设计者避免了这个问题，保持了与现有代码的兼容性。

6.4 循环结构

循环作为一种核心的控制流程机制，赋予了程序重复执行特定代码段的能力，直至满足预设的终止条件。它可谓是编程语言的基石，极大地简化了处理大量数据和执行重复性任务的过程。在程序设计中，循环结构的重要性不言而喻。如果没有循环，程序员将不得不为每一个步骤手动编写重复的代码，这不仅会导致代码量剧增，而且会使得代码难以阅读、管理和维护，降低开发效率和代码质量。

在 Java 编程中，常用的循环结构主要包括 do-while 循环、while 循环、标准 for 循环以及增强型 for 循环（通常被称为 foreach 或 for-each 循环）。尽管这几种循环机制在很多场景下可以相互替代，实现相似的迭代逻辑，但它们各自具备独特的特性与适用情境。首先，我将简要概述各循环类型的特征，随后逐一详解它们在不同编程需求中的应用优势。

无限循环：一旦启动就不会停止的循环。这种循环通常用于需要持续运行的系统或设备，例如 1977 年 9 月 5 日发射的旅行者 1 号探测器，以及心脏起搏器等医疗设备。这些系统或设备需要无限循环来持续执行特定的任务，确保其正常运行。

计数循环：预先知道循环次数的循环。这种循环通常用于需要重复执行特定次数的操作，例如购买了一年的 QQ 会员，程序会循环 365 次提供相应的会员服务，当预先设定的次数完成后，循环即告结束。

连续求值循环：每次迭代都需要计算结束条件的循环。这种循环在预先不知道循环次数的情况下非常有用。例如，预付费电话卡在每次通话时都会检测话费余额，并根据余额是否充足来决定是否继续提供通话服务。这种循环的特点是每次迭代都需要对条件进行求值，直到条件不再满足为止。

迭代器循环：对集合内每个元素执行一次操作的循环。在编程中，集合是一种用于存储多个元素的对象，这些元素可以是基本数据类型，也可以是复杂的对象。例如，下一章

要介绍的数组就是一种常见的集合类型。迭代器循环允许我们遍历集合中的每个元素，并对其执行特定的操作，这在处理大量数据时非常方便。

接下来，将分别学习这四种循环。

6.4.1 do-while 循环

do-while 循环是 Java 和其他编程语言中的一种循环结构，它确保循环体至少被执行一次，然后再根据条件决定是否继续执行。do-while 循环的基本语法和工作原理如下：

```
do {
    // 循环体（代码块）
} while (布尔表达式); // 条件判断
```

在 do-while 循环中，首先会无条件地执行一次“循环体”内的代码，然后检查紧跟在 do 后面的 while 语句中的布尔表达式。如果该表达式的值为 true，则继续执行循环体；若为 false，则退出循环。接下来，演示一下 do-while 循环的用法。

```
public class DoWhileExample {

    public static void main(String[] args) {
        // 目标数值，这里为了演示用一个固定的值
        int targetNumber = 10;
        int sum = 0;

        // 使用 do-while 循环累加从 1 到目标数的和
        int i = 1;
        do {
            sum += i++;
        } while (i <= targetNumber);

        System.out.printf(" 从 1 累加到%d的和是: %d%n",
                                    targetNumber, sum);
    }
}
```

在这个例子中，我们直接定义了一个目标数值 targetNumber 为 10，然后通过 do-while 循环将从 1 开始的所有整数累加起来，直到达到 targetNumber 为止，最后输出累加的结果。在这个例子中，从 1 一直累加到 10，最终的结果是 55。

总结一下，在 Java 中，do-while 循环是一种后测试循环，即先执行循环体，然后检查条件。如果条件为真，再次执行循环体。这个过程会一直重复，直到条件为假。由于至少执行一次循环体，所以 do-while 循环特别适合以下几种应用场景：第一个是用户输入密码验证，只有当用户输入满足特定条件时，才会退出循环。

```
do {
    System.out.println(" 请输入密码: ");
    password = scanner.nextLine();
} while (!isValidPassword(password));
```

第二个应用场景是实现菜单驱动的程序，用户可以反复选择菜单项，直到选择退出，示例代码如下：

```
int choice;
do {
    System.out.println("1. 选项一");
    System.out.println("2. 选项二");
    System.out.println("3. 退出");
    choice = scanner.nextInt();
    // 处理用户的选择
} while (choice != 3);
```

第三个应用场景是用于实现游戏循环，只有当玩家选择退出游戏时，才会退出循环，示例如下：

```
do {
    // 游戏代码
    System.out.println(" 是否继续游戏? (yes/no)");
    continueGame = scanner.nextLine();
} while (continueGame.equalsIgnoreCase("yes"));
```

6.4.2 while 循环

Java 中的 while 循环是一种条件控制结构，允许程序在满足特定条件时重复执行一段代码。while 循环的基本语法是：

```
while (布尔表达式) {
    // 循环体（需要重复执行的代码）
```

```
}
```

其运行逻辑如下：首先，程序会检查括号内的布尔表达式的值。如果布尔表达式的值为 true，则执行循环体内的代码。执行完一次循环体后，程序返回到 while 语句重新检查布尔表达式的值。这个过程会不断重复，直到布尔表达式的值变为 false 为止，此时程序将跳出循环，继续执行 while 循环后的下一条语句。演示 while 循环如下：

```
public class WhileExample {

    public static void main(String[] args) {
        // 目标数值，这里为了演示用一个固定的值
        int targetNumber = 10;
        int sum = 0;

        // 使用 while 循环累加从 1 到目标数的和
        int i = 1;
        while (i <= targetNumber) {
            sum += i++;
        }

        System.out.printf(" 从 1 累加到%d的和是: %d%n",
            targetNumber, sum);
    }
}
```

上面的代码使用了一个 while 循环来实现累加操作。在 while 循环中，定义了一个计数器 i，并初始化为 1。只要 i 的值小于或等于 targetNumber，就会执行循环体。在循环体中，将 i 的值加到 sum 上，然后 i 自增 1。这个过程会一直重复，直到 i 的值大于 targetNumber。输出的结果跟前面的 do-while 循环是一样的。

while 循环的特点是，循环次数在进入循环前就已经确定。这使得 while 循环非常适合用于那些循环次数在运行时才能确定的情况。比如，当我们需要读取一个文件的内容时，可以使用 while 循环，其代码示例如下：

```
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
```

通过上面的示例，我们可以看到，while 循环和 do-while 循环都是用于重复执行一段代码，大部分情景下，两者可以相互替换。但它们在某些场景下的适用性有所不同：

while 循环：在每次循环开始之前检查条件。如果条件为假，那么循环体一次都不会执行。这使得 while 循环特别适合于循环次数在开始循环前就已知，或者需要在执行循环体之前验证条件的场景。例如，读取文件的每一行，直到文件结束；或者重复请求网络资源，直到接收到有效的响应。

do-while 循环：在每次循环结束之后检查条件。这就意味着，无论条件是否满足，循环体中的代码至少会被执行一次。这使得 do-while 循环特别适合于至少执行一次循环体，或者需要在执行循环体后验证条件的场景。例如，一个菜单驱动的程序，用户至少需要看到一次菜单；或者一个游戏，至少需要玩一轮。

总的来说，选择使用 while 循环还是 do-while 循环，主要取决于你是否需要在开始循环前检查条件，以及是否需要至少执行一次循环体。

在深入探讨了 while 和 do-while 循环结构的原理与应用之后，咱们再继续探索 Java 编程中的另一种强大且灵活的迭代工具——for 循环。for 循环以其简洁明了的语法形式，为程序员提供了高效处理序列数据、进行计数循环以及实现特定次数重复执行代码段的能力。从遍历数组元素到计算指定次数内的累加值，for 循环不仅能够简化代码编写，而且在优化执行效率方面具有显著的优势。接下来，我们将详细剖析 for 循环的内在机制及其多种应用场景，并通过实例展示如何有效地利用 for 循环来解决实际编程中的各类问题。

6.4.3 for 循环

Java 中的 for 循环是一种广泛使用的控制结构，它允许程序员在满足特定条件时重复执行一段代码块。Java 的 for 循环通常用于已知迭代次数的情况，或者遍历数组、集合等数据结构的所有元素。以下是 for 循环的基本语法：

```
for (初始化表达式; 布尔条件表达式; 更新表达式) {
    // 循环体（需要重复执行的代码）
}
```

接下来，用 for 循环实现的计算 1 到 10 之和：

```
public class SimpleCalculator {

    public static void main(String[] args) {
        // 使用 for 循环计算 1 到 10 的整数之和
        int sum = 0;
```

```
        for (int i = 1; i <= 10; i++) {
            sum += i;
        }

        System.out.println(" 从 1 到 10 的整数之和是: " + sum);
    }
}
```

结合上面 for 循环的基本语法和上面的示例，详细介绍每个部分的含义和作用如下：

初始化表达式：在循环开始之前执行一次，用于设置循环变量的初始值。在示例中是 i = 1，将循环变量的初始值设定为 1。

布尔条件表达式：每次循环开始前都会检查这个条件，如果为 true 则继续执行循环体内的代码；如果为 false 则退出循环。在示例中是 i <= 10，在每次循环的时候，都会检查 i 的值是否小于等于 10，如果为 true，则继续运行；如果为否，则退出循环。

更新表达式：在每次循环结束后执行，用于更新循环变量的值。在示例中 i++ 是自增运算符，在第 5 章中已经有介绍。它表示将变量 i 的当前值加 1，并返回操作前的原始值。

循环体：这是在每次循环中被执行的一段代码。在示例中是 sum += 1 这一句。在循环体内部使用这个表达式时，可以用来累计计数器 i 自增。原始的 sum 为 0，那么每次执行 sum += 1 后，sum 就会递增：0, 1, 2, 3… 这样依次增加。

总结一下，Java 传统的 for 循环结构对于已知次数的迭代以及对数组和集合元素的索引访问有着出色的表现。然而，在处理大量数据集合时，特别是当关注点在于直接操作每个元素而非其索引时，Java 提供了一种更为简洁高效的循环机制——增强型 for 循环（也称为 foreach 循环）。这种循环形式摒弃了显式地管理索引和边界检查，专注于元素本身的操作，可以提高代码的可读性和编写效率。接下来，我们将深入研究增强型 for 循环的具体语法和应用场景。

6.4.4 增强的 for 循环

Java 中的增强型 for 循环，也称为 foreach 循环，是一种简化了数组和集合遍历的循环结构。它从 Java 5 版本开始引入，用于更简洁、直观地访问数组或实现了 Iterable 接口的对象（如 List、Set 等集合）中的每一个元素，无需手动管理索引。其语法结构如下：

```
for (元素类型 变量名 : 集合或数组) {
    // 循环体内的代码，对变量名所代表的当前元素进行操作
}
```

以下的代码演示的是 Java 中使用增强型 for 循环遍历数组的例子：

```
public class EnhancedForLoopExample {

    public static void main(String[] args) {
        // 创建一个整数数组
        int[] numbers = { 1, 2, 3, 4, 5 };

        // 使用增强型 for 循环遍历数组并打印每个元素
        for (int number : numbers) {
            System.out.println(" 当前数字: " + number);
        }
    }
}
```

在上述示例中，增强型 for 循环自动遍历数组的每个元素，并将当前元素的值赋给指定类型的临时变量，在每次迭代时使用这个临时变量来访问和操作当前元素。这样可以避免编写复杂的索引计算和边界检查代码，使得代码更加简洁且易于理解。当运行这段代码时，它会输出：

```
当前数字: 1
当前数字: 2
当前数字: 3
当前数字: 4
当前数字: 5
```

前面讲过，增强型 for 循环的语法结构是专门为简化数组和集合类型的遍历而设计的。如果深入探究的话，对数组和对集合还是有区别的。对于例子中的数组而言，编译器会自动生成对应的索引访问代码，从而无需程序员显式地管理索引变量。增强型 for 循环实际上是对 for 循环的语法糖封装，底层仍然使用传统的 for 循环方式进行遍历。

而在遍历实现了 Iterable 接口的对象时，如各种集合类（List、Set 等），则是通过调用 iterator() 方法获取迭代器来进行遍历的。对集合来说，增强型 for 循环的底层实现依赖于迭代器模式，该模式将遍历操作与具体的数据结构分离，提供了一种统一的遍历方式。有关集合的内容，在后面章节里将详细讲解。

6.4.5 break 和 continue

在先前的讨论中，我们主要涉及了基于特定条件满足时自然终止的循环结构。然而，在实际编程实践中，存在一种更为灵活且实用的需求，即在遇到某种“非预期”或“提前”的退出条件时中断循环的执行过程。针对这一场景，Java 语言特地引入了 break 语句和 continue 语句以供开发者实现对循环流程的精细控制。

break 语句允许程序员在满足特定条件时立即跳出整个循环结构，无论该循环是否尚未完成预定的迭代次数。而 continue 语句则提供了另一种机制，它能够使当前循环迭代提前结束，并直接跳转至下一轮循环的判断阶段，从而避免执行剩余的本迭代内代码块。这两种语句共同丰富了循环控制的策略，使得程序可以根据实际情况动态地调整其执行路径。

在现实生活中，一个生动的场景可以类比于 Java 编程中的循环控制。比如，育儿过程中的一大挑战就是确保孩子完成一顿饭的进食。如果我们把这个过程比喻为一个循环操作，那么每一口饭的摄入就相当于一次循环迭代，而当碗里的食物全部吃完时，则标志着循环自然结束。

然而，在实际喂养过程中，孩子们可能会表现出不同的饮食偏好，就如同古代伯夷与叔齐义不食周粟，有些小孩则是义不食青椒，一旦遇到此类食物，他们会选择停止进食，这就类似于提前跳出吃饭这一“循环”。用代码来模拟这一过程为：

```
for (Food food : meal) {
    // 如果当前食物是青椒
    if (food.equals(" 青椒")) {
        // 孩子拒绝吃青椒，因此提前结束此次吃饭的“循环”
        break;
    }
    // 模拟吃饭的代码
    eat(food);
}
```

与 break 语句导致循环提前彻底终止不同，continue 语句的作用是仅跳过当前循环迭代中尚未执行的代码部分，并直接进入下一轮循环。换言之，在实际运用中，当遇到特定条件时，continue 可以让程序忽略当前循环体内的剩余操作，而非结束整个循环过程。

仍以喂孩子吃饭为例进行类比，假设有些小孩在面对不喜欢的食物（如青椒）时，并不会因此完全拒绝进食，而是会选择吐掉这一口含有青椒的食物，然后在家长的安抚下继续吃剩下的饭。在这种情境下，若使用编程语言模拟，将原先的 break 语句替换为 continue 语句就显得更为贴切：

```
for (Food food : meal) {
    // 如果当前食物是青椒
    if (food.equals(" 青椒")) {
        // 孩子不吃青椒，跳过本次循环内剩余的逻辑
        // 即不执行 eat(food)，进入下一口饭的检查
        continue;
    }
    // 模拟吃饭的代码
    eat(food);
}
```

为了更清晰地阐述 break 和 continue 在实际编程中的运用差异，接下来我将通过一个具体的 Java 代码实例来说明。该示例通过计算数组中非零元素的累计值以及遇到特定数值时的循环策略，展示 break 和 continue 两种控制语句的不同作用机制。以下代码是具体实现：

```
public class BreakContinueExample {
    public static void main(String[] args) {
        int[] numbers = { 0, 1, 2, 3, 4, 5, 6,
                                    7, 8, 9, 0, -1, 0 };

        // 示例 1: 使用 break 提前结束循环（找到第一个 0 时停止求和）
        int sumUntilZero = 0;
        for (int number : numbers) {
            if (number == 0) {
                break; // 遇到 0 就跳出循环，不再累加
            }
            sumUntilZero += number;
        }
        System.out.println(" 遇到第一个 0 之前的所有数字之和为: "
                                            + sumUntilZero);

        // 示例 2: 使用 continue 跳过特定值
        // 跳过所有 0 继续累加非零数
        int sumWithoutZero = 0;
        for (int number : numbers) {
            if (number == 0) {
```

```
                // 遇到 0 就跳过本次循环，但不终止整个循环
                continue;
            }
            sumWithoutZero += number;
        }
        System.out.println(" 所有非零数字之和为: "
                                    + sumWithoutZero);
    }
}
```

在上述的例子中，break 用于“求和直到遇到 0”的情况。当循环体内的条件满足（即当前元素等于 0）时，break 语句会使程序立即退出整个 for 循环，不再检查数组中的剩余元素。continue 则用于“计算数组中所有非零元素之和”的情况。当条件满足（即当前元素等于 0）时，continue 语句会让程序跳过当前迭代中剩余的代码（在这里是累加操作），直接进入下一次循环迭代，继续检查下一个元素是否为 0 或进行累加。因此，以上的代码运行之后，结果如下：

```
遇到第一个0之前的所有数字之和为: 0
所有非零数字之和为: 44
```

通过上述实例，我们不难发现 break 和 continue 语句的核心作用在于将程序的执行流程转移至特定位置，这种行为在某种程度上类似于无条件跳转指令 goto。回顾本章开篇时提及的关于业界对滥用 goto 语句引发的深度讨论与广泛争议，自然会引出一个问题：是否应当谨慎使用甚至避免使用 break 和 continue?

实际上，在编程实践中，尽管 Java 已废弃了 goto 关键字以避免其潜在的代码混乱风险，但 break 和 continue 仍然作为控制循环结构的重要手段存在。它们能够在遵循一定规范的前提下实现类似跳转的效果，并且在合理应用场景中发挥关键作用。因此，对于 break 和 continue 的使用，并非简单的提倡少用或不用，而是在理解和掌握其功能的基础上，确保在适宜场景下适度运用，以保持代码的清晰。

在 Java 语言中，尽管已将 goto 语句的实际语法弃用，但 goto 仍作为保留字存在于 Java 的词汇表内，至今未被赋予实际功能。从目前的语言发展趋势来看，我认为 goto 在未来被启用的可能性极小。然而，无论 goto 关键字是否会在未来激活，Java 通过其内置的“标签”机制，能够在特定限制下模拟 goto 语句的部分行为。

值得注意的是，标签这一特性在编程实践中的应用场景相对有限，主要集中在处理多重循环结构时，用于提供一种更为精确和可控的跳转逻辑。接下来，我们再来研究一下 Java 的标签。

6.4.6 标签

Java 中的标签（label）是一种为循环或 switch 语句提供唯一标识符的机制，允许程序员在复杂的嵌套循环结构中更精确地控制流程。通过在循环或 switch 前定义一个标识符后跟冒号（:），可以给特定的循环或 switch 块打上“标签”。

具体到 Java 编程中，标签常用于嵌套循环或 switch 语句中，允许程序员在满足一定条件时从深层嵌套结构中跳出至指定位置，这在实际开发中具有一定的实用性，尤其是在处理复杂的逻辑结构时。例如，在游戏编程中，可能需要根据不同状态从嵌套的游戏循环中提前退出；或者在解析复杂数据结构时，遇到错误就跳出整个解析过程等。下面的代码用于演示标签的用法。

```
public class LabelExample {
    public static void main(String[] args) {
        int outerLoopIndex = 0;

        // 对外层循环设置标签: loop1
        loop1: for (int i = 0; i < 5; i++) {
            System.out.println(" 外部循环迭代次数: "
                               + (outerLoopIndex + 1));

            for (int j = 0; j < 3; j++) {
                if (j == 1) { // 当内层循环达到某个条件时
                    System.out.println(" 从内层循环
                                        中断外部循环。");
                    // 使用标签跳转到外层循环结束的位置
                    break loop1;
                }
                System.out.println(" 内部循环迭代次数: " + j);
            }

            // 如果没有提前跳出，此处会更新外层循环索引
            outerLoopIndex++;
        }

        System.out.println(" 正常退出外部循环。");
    }
}
```

以上代码运行之后，输出的结果如下：

```
外部循环迭代次数：1
内部循环迭代次数：0
从内层循环中断外部循环。
正常退出外部循环。
```

标签是非常少用的一种语法，常见的用途是从多重嵌套中跳转出来。我想到了一部电影叫《盗梦空间》，在电影里，梦境会一层一层的嵌套。比如当有四重梦境的时候，必须依 4321 的顺序醒来才能回到现实。如果，导演克里斯托弗·诺兰学过 Java，知道有个东西叫标签，可以在每一层梦境里加上一个标签，就可以随意在不同的梦境中跳转了。

7 方法

Contents

> **!** 重要
>
> 在排列这一章的时候，本想在前面放一章数组，但是后来又觉得数组应该放在后面。总之呢，最初数组放前面，后来我又将数组放在了后面。
>
> 情况是这样，如果前面放数组一章，那涉及到方法的时候就没学。如果前面放方法一章，那涉及到数组的时候就没学。世间安得两全法，不负如来不负卿。
>
> 其实排列顺序的理由也不是很充分，我觉得还是放后面吧。数组的用处很广泛，但是我更推荐用集合，这些在后续章节会讲。

编程，本质上是一场将现实世界问题抽象化的旅程。在之前的章节中，我们逐步探索了变量、数据类型和表达式等基本概念。它们如同积木般，帮助我们将实际问题转化为计

算机可理解的抽象变量，并通过组合构建逻辑语句。更进一步，凭借控制结构的力量，我们得以精准掌控这些语句的执行顺序和条件，从而赋予程序计算的能力。

抽象，在编程世界里扮演着两重角色：数据抽象与过程抽象。数据抽象，旨在实现数据的封装，它是面向对象编程的基石，我们将在后续章节中深入探讨。而本章的主题是过程抽象。它强调将一系列操作或任务，巧妙地封装在一个方法之中。这种设计如同为复杂的瑞士手表设计了一个简洁的外壳，隐藏了内部的齿轮运作，只展现功能接口。开发者只需关注如何调用方法来完成任务，无需深究其内部的实现细节，从而简化了开发过程。

那方法是由谁发明的呢？你或许听过计算机科学界一句名言：“在计算机科学中，没有什么问题是不能通过增加一个中间层来解决的，但这通常会产生新的问题。”这句话的作者，David Wheeler 博士，正是方法的发明者。他参与了早期计算机 EDSAC 的研发，而 EDSAC 正是支持方法的先驱。为了纪念他的贡献，方法跳转的指令被命名为“惠勒跳转”，以示敬意。

在 Java 中，方法就像是代码积木，将特定功能的代码块封装起来，随时等待召唤。它们可以被类、对象，甚至是其他方法调用，协同完成复杂的任务。接下来，让我们一起来学习 Java 的方法。

7.1 方法的概述

在先前的章节中，我们的焦点实际上都在一个核心方法上，即 main 方法。然而，随着程序规模逐渐扩大，仅依赖单一的大块代码将难以实现高效编写与维护。当代码量尚处于较小阶段时，每一行代码的功能和意图较为清晰；但随着代码行数增长至数十乃至数百行，程序便会变得冗余复杂，让人望而生畏。为应对这一挑战，引入方法的概念成为至关重要的解决方案。方法堪称计算机科学领域的一项重大创新之一，在 Java 编程中发挥着举足轻重的作用。通过合理运用方法，可以有效的提高 Java 程序的可读性、可维护性和可扩展性。

在上一章节探讨数组应用时（整理电子书的时候，我把数组这一章放后面去了），曾利用数组来实现对本福特定律，如果回顾那时的代码，其结构显得较为复杂。在软件设计方法论中，遵循“单一职责原则”至关重要，这一原则强调每个方法应专注于一项具体任务，若一个方法因多种不同的需求变更而频繁调整，则表明它承载了过多的功能责任，这时就应当进行合理拆分以提高代码的清晰度和可维护性。

因此，在本章中，我们将进一步精炼代码结构，依据单一职责原则将计算本福特数字规律的方法独立抽象出来，使其功能更为纯粹且专注，从而提升整个程序的模块化程度与扩展性。精炼后的代码如下所示。

```
import java.util.Scanner;
import java.io.File;
```

```
public class Benford {

    // 返回首位数字
    public static int leadingDigit(int x) {
        while (x >= 10) {
            x = x / 10;
        }
        return x;
    }

    public static void main(String[] args)
                                    throws Exception {
        // 分别存放数字出现的次数
        int[] count = new int[10];
        // 统计总共输入了多少数字
        int n = 0;

        Scanner scanner = new Scanner(new File("data.txt"));

        while (scanner.hasNextInt()) {
            int x = scanner.nextInt();
            int digit = leadingDigit(x);
            count[digit]++;
            n++;
        }

        for (int i = 1; i < 10; i++) {
            // 输出结果
            System.out.printf("%d: %6.1f%%\n",
                                i, 100.0 * count[i] / n);
        }
    }
}
```

在上述代码中定义了一个名为 leadingDigit() 的方法，该方法专注于实现单一功能，即提取并返回一个整数的首位数字。此方法的返回类型为 int，并要求传入参数也为 int 类型。

在定义完成这个方法后，随后进行了调用。方法调用过程实质上是程序从 main 方法执行流程中转跳到 leadingDigit() 方法的执行环境，待 leadingDigit() 方法执行完毕后，控制权会重新交回至 main 方法继续执行。以下是对此方法结构的示意图表示：

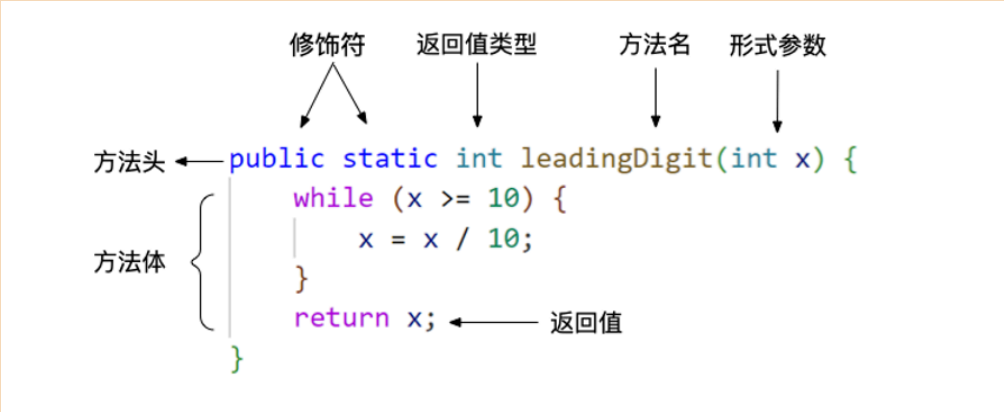

图 7.1: 方法的结构图

从上面的结构图可以看到，方法有方法头（method header，有时也被翻译为方法声明）和方法体两部分组成，每一部分又会涉及到几个小的知识点，接下来，我会详细介绍这些知识点。

7.2 方法的声明与调用

还是以本福特定律的代码为例，这个例子中定义了一个只有 3 行代码的 leadingDigit() 方法。现代的计算机程序通常都是由很多极短的方法外加很少较长的方法组成的。在代码数量少的时候，我们把其中的代码抽取出来，抽象成方法，就能有效的降低复杂度，以防代码无限扩张。所以，不要觉得代码太短而不抽象成方法。

重要

调用方法会造成多大的性能开销?

如果你是在性能稍弱的嵌入式上开发，说实在的，开销还是比较明显的。但是，相比于损失的这些性能，容易编写代码更为重要。

这也是一种趋势，功能分拆的越来越小，依赖越来越强。不管这是好事还是坏事，软件之间相互依赖越来越严重。

> 在 2016 年曾经发生过一件事，在 nodejs 社区，有个开发者 Azer Koçulu 撤回了他在 npm 上的 11 行代码，结果导致有大量用户的 npm 崩溃了。有兴趣的可以搜一下 “How one programmer broke the internet by deleting a tiny piece of code” 关注此事。这件事也从侧面说明了现在软件的趋势，依赖性越来越强，就算你写的项目，你都不知道你用了谁写的代码。

在 Java 中，一个典型的方法头基本包含如下几个部分：修饰符、返回值类型、方法名和形式参数（如果该方法有参数的话）和可选的抛出的异常。

一般来说，修饰符分为如下几类：访问修饰符（Access Modifier）、静态修饰符（static）、最终修饰符（final）和同步修饰符（synchronized）。访问修饰符用于指定方法的访问级别，可以是 public、protected、private 或默认无修饰符（package-private）。静态修饰符表明该方法是属于类的而不是类实例的，可以直接通过类名调用，无需创建对象。最终修饰符表示方法不能被重写。同步修饰符用于控制多线程环境下的并发访问。在这四种修饰符中，除了访问修饰符是必须的，其它三种都是可选的。

返回值类型定义了方法执行后返回的结果的数据类型。如果方法不返回任何值，则使用关键字 void。在本例中，返回值的类型是 int。

方法名称是方法的唯一标识符。方法名称应该以小写字母开头，并且尽量遵循驼峰命名规则，也就是说要遵循 Java 标识符的命名规则。

抛出的异常：这是一个可选的部分，定义了方法可能会抛出的异常。

再接下来，就是小括号里的参数了。Java 方法中的参数是指在方法定义中声明的输入变量，用于接收传递给方法的值或对象。在方法定义中，参数列表是放置在方法名后面的一对圆括号中，多个参数之间用逗号分隔。方法可以拥有零个或多个参数，具体的参数数量和类型取决于方法的需求。

还有一点需要 C/C++ 程序员注意，定义方法的位置是有要求么？在 Java 中，方法的声明位置对于其调用并没有硬性规定。你可以在方法被调用之前或之后声明方法，这是因为 Java 在编译时会处理所有的方法声明，并建立好方法的符号引用，所以在同一个类中，方法的调用顺序不受方法声明的顺序影响。在 C/C++ 中，函数必须在调用它之前被声明。这是因为 C/C++ 是按照源代码的顺序进行编译的，编译器在遇到函数调用时需要知道函数的声明信息（如返回类型、函数名、参数类型等）才能正确地生成代码。

7.3 方法的参数

方法作为一种可复用的代码块，在 Java 编程中具有接收外部输入参数并基于这些参数执行特定任务的能力。通过这种方式，方法能够灵活地处理各种不同的输入数据，并根据

需要产生相应的输出结果。一旦完成对传入数据的处理，方法可以将计算或处理后的值作为返回值传递回调用者。

接下来，我们将以一个具体的示例来展示如何在 Java 中定义一个接受参数并返回结果的方法。下面要写的程序是计算 BMI，要计算 BMI 需要提供身高和体重。计算的公式如下：$BMI =$ 假如一个人的身高是 1.75 米，体重是 72kg，那么 $BMI = 72/1.75^2 = 23.5$。当计算完成以后，可以把自己的身高与体重与下面的表格对比一下。

状态	身体质量指数 (BMI)
体重过轻	BMI < 18.5
正常范围	18.5 ≤ BMI < 24
体重过重	24 ≤ BMI < 27
轻度肥胖	27 ≤ BMI < 30
中度肥胖	30 ≤ BMI < 35
重度肥胖	BMI ≥ 35

计算 BMI 的代码如下所示：

```
import java.text.NumberFormat;
import java.util.Scanner;

public class CalculateBMI {

    public static double bmi(double weight,
                             double height) {
        double bmi = weight / (Math.pow(height, 2));
        return bmi;
    }

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.print(" 请输入体重 (千克): ");
        double weight = input.nextDouble();

        System.out.print(" 请输入身高 (米): ");
```

```
            double height = input.nextDouble();

            double bmi = bmi(weight, height);

            System.out.print(String.format
                                ("你的 BMI 是: %.1f", bmi));

            input.close();
        }

    }
```

在计算 BMI 指数的过程中，我们需向相关方法传递两个核心参数：体重和身高。尽管该计算逻辑相对简单，但在实际编程实践中，不慎颠倒参数的传递顺序是一个屡见不鲜的错误源头。《Software Errors and Complexity: An Empirical Investigation》这篇论文的研究揭示，大约 40% 的编程错误源自于调用方法时对参数顺序的混淆。为了解决因参数顺序不明导致的错误，科学家发明了命名参数。

在计算机编程技术中，命名参数是一种创新机制，它允许程序员在函数调用时明确指出每个参数值对应的参数名称，而非依赖传统的按位置传递方式——即必须严格按照函数定义中参数声明的先后顺序来传递值。

许多现代编程语言已内置支持命名参数特性，使得开发者能够清晰地指定参数名，而不仅限于依赖列表中的位置关系。然而，在目前版本的 Java（截至 2024 年）中，尚未提供原生的命名参数语法支持。因此，为了应对这一挑战，Java 开发者通常采用构造器模式或 Builder 设计模式，在创建类实例时清晰表达参数的意义；同时也可以借助第三方库如 lombok，通过注解模拟实现类似命名参数的效果，从而显著提升代码可读性并降低出错概率；最后，由于目前 IDE 开发环境中，都提供代码提示，以帮助程序员记住参数的顺序和类型。所以，命名参数显得并不是特别紧迫。

在实际编程中，有很多程序员会使用以上提到的方法模拟命名参数，所以，我觉得有必要讲一下这个 Java 中目前并不支持的命名参数。

7.3.1 命名参数

所谓命名参数就是允许你在方法调用时使用参数名称来指定参数值，而不仅仅按照位置顺序传递参数。使用命名参数可以提高代码的可读性和可维护性，特别是当方法有多个可选参数或参数顺序不重要时。由于目前 Java 尚不支持，我举个 Kotlin 的例子，让大家体验一下命名参数的优点。

即使你现在不懂 Kotlin 这门编程语言也没关系，如下的代码清单也是计算 BMI 的，代码如下：

```
fun calculateBMI(weight: Double, height: Double): Double {
    if (height <= 0) throw IllegalArgumentException(
                                                " 身高为正数")
    if (weight <= 0) throw IllegalArgumentException(
                                                " 体重为正数")
    return weight / (height * height)
}

fun main() {
    val bmi1 = calculateBMI(weight = 70.0, height = 1.75)
    println("BMI is $bmi1")
    val bmi2 = calculateBMI(height = 1.75, weight = 70.0)
    println("BMI is $bmi2")
}
```

通过上面的代码我们可以看到，使用命名参数以后，参数的顺序变得不重要。在调用 calculateBIM 方法的时候可以先输入身高，也可以先输入体重，由于明确指出 weight = 70.0，height = 1.75，代码的可读性也更强了。在 Java 中，目前可以用下面的代码来模拟实现命名参数：

```
public class BmiCalculator {
    private final double weight;
    private final double height;

    private BmiCalculator(BmiBuilder builder) {
        this.weight = builder.weight;
        this.height = builder.height;
    }

    public static class BmiBuilder {
        private double weight;
        private double height;

        public BmiBuilder setWeight(double weight) {
```

```
            if (weight <= 0)
                throw new IllegalArgumentException(
                                        " 体重必须为正");
            this.weight = weight;
            return this;
        }

        public BmiBuilder setHeight(double height) {
            if (height <= 0)
                throw new IllegalArgumentException(
                                        " 身高必须为正");
            this.height = height;
            return this;
        }

        public BmiCalculator build() {
            return new BmiCalculator(this);
        }

        public double calculateBmi() {
            return weight / (height * height);
        }
    }

    public static void main(String[] args) {
        double bmi = new BmiBuilder()
                .setWeight(70)
                .setHeight(1.75)
                .calculateBmi();
        System.out.println("BMI is " + bmi);
    }
}
```

在这个例子中，可以像使用命名参数一样使用 setWeight 和 setHeight 方法来设置参数。这两个方法返回 BmiBuilder 对象自身，可以链式地调用它们，从而使用任意顺序来调用这些方法，从而使得代码更容易理解。

! 重要

为什么 Java 一直不支持命名参数呢?

以下是我的个人观点，希望有人研究过源码之后来跟我讨论一下。我觉得我是对的，不过……我不是永远正确的人:）

因为字节码中一般不存储方法的参数名。根据前面的章节，我们已经知道 Java 的源代码要编译为后缀为 class 的字节码，为了减少字节码文件的大小，通常不在字节码文件中存储方法的参数名。

从 Java 8 开始，Java 编译器有一个选项 -parameters，可以将方法参数的名字包含在字节码中。这主要是为了支持一些新的语言特性和框架，例如 Spring 和 Java 8 的函数式编程接口。

需要注意的是，这个选项默认是关闭的，需要显式开启。虽然目前 Java 不支持命名参数，但是现代的 IDE 都会给出相应的提示，确保程序员不会混淆方法中的参数。

7.3.2 可变参数

在实际编程应用中，设想这样一个情景：我们需要为不同的班级计算平均成绩，而每个学生的考试成绩作为独立的输入参数。由于现实情况的不确定性，我们无法预先得知确切的成绩数量，因为这取决于具体参与考试的人数，不同班级可能人数不一，甚至存在个别学生缺考的情形。面对这种参数数量动态变化的问题，Java 语言提供了一种强有力的工具——可变参数（varargs）机制。

在 Java 编程语法中，可变参数是一种独特的特性，它允许我们在方法定义时声明一个可以接受零个或多个同类型参数的方法签名。具体来说，通过在参数类型后添加三个点符号（...）来标识该参数为可变参数，例如 “double... scores”。这意味着调用方法时，可以传递任意数量的相应类型数据作为参数，这些参数会被自动封装成数组供方法内部进行处理。

因此，在计算班级平均成绩等需要处理不定数量输入值的场景下，运用 Java 的可变参数功能能够极大地简化代码编写和调用过程，让方法更具灵活性与适应性，无需针对不同人数的情况编写多个重载方法。以下的代码是使用可变参数的例子。

```
public class ClassAverageCalculator {
    public static double calculateAverage(double... scores) {
        double sum = 0;
        for (double score : scores) {
            sum += score;
```

```
        }
        return sum / scores.length;
    }

    public static void main(String[] args) {
        double averageScore = calculateAverage(85.5,
                                92.0, 78.5, 89.5, 91.0);
        System.out.println("班级的平均成绩是: "
                                + averageScore);
    }
}
```

上面的代码运行以后的结果为：班级的平均成绩是: 87.3。

在以上示例中，我们定义了一个静态方法 calculateAverage，该方法接受可变参数 double... scores。在 calculateAverage 方法内部，我们使用一个循环遍历所有传递的成绩，并将它们相加以计算总和。最后，我们将总和除以成绩数量，得到平均成绩，并将其返回。

声明可变参数的语法是指定参数类型后跟着省略号（...），表示该参数可以接受任意数量（零个或多个）的值。可变参数必须是方法的最后一个参数。在方法内部，这些值将被视为一个数组，可以使用数组的相关语法来访问和处理这些参数。下面的是一个结合了固定参数与可变参数的例子，该例子定义了一个格式化消息的方法，其中固定参数用于指定消息的主题，而可变参数则用于传递任意数量的消息内容。

```
public class MessageFormatter {
    // 定义一个方法，第一个参数为主题，
    // 后面的参数为可变参数，用于存储消息的内容
    public static String formatMessage(String topic,
                                       Object... messageParts) {
        StringBuilder sb = new StringBuilder();

        // 固定参数：添加主题
        sb.append("【").append(topic).append("】\n");

        // 可变参数处理：遍历并添加每个消息内容部分
        for (Object part : messageParts) {
            sb.append(part).append("\n");
        }
```

```
        // 返回最终格式化的消息字符串
        return sb.toString().trim();
    }

    public static void main(String[] args) {
        // 调用示例，主题为" 每日新闻"，后面跟随多个消息内容
        String formattedMessage = formatMessage(" 每日新闻",
            "1. 本地新闻报道", "2. 国际事件分析", "3. 天气预报");
        System.out.println(formattedMessage);
    }
}
```

在这个例子中，formatMessage 方法首先接收一个固定的 String 类型参数 topic，然后通过可变参数 Object... messageParts 接收任意数量的对象作为消息内容。在方法内部，我们先添加固定的主题到 StringBuilder 对象中，然后循环遍历可变参数数组，并将所有消息内容逐条添加进去。这样，调用者可以根据需要传入不同数量的消息段落来构建一个完整的消息。

上面的程序运行之后的结果如下：

```
【每日新闻】
1. 本地新闻报道
2. 国际事件分析
3. 天气预报
```

7.3.3 形参与实参

在 Java 编程语言中，形参（形式参数）和实参（实际参数）是方法或函数定义与调用过程中的核心概念。

```
public class Example {
    // 定义一个方法，其中 x 和 y 是形参
    public static int add(int x, int y) {
        return x + y;
    }

    public static void main(String[] args) {
        // 在这里，3 和 5 是实参
```

```
        int result = add(3, 5);
        // 输出: 结果是: 8
        System.out.println(" 结果是: " + result);
    }
}
```

形参是在声明一个方法时定义的一组变量，它们如同容器或者占位符，用来接收并暂存调用该方法时传递进来的数据。例如，在声明一个计算两数之和的方法时，可以定义两个整型形参 int a 和 int b，表示这个方法需要接受两个整数值作为输入。形参的作用范围仅限于方法内部，一旦离开该方法的定义区域，它们就不再有效。

实参则是指在调用方法时，实际提供给方法的值或者表达式。当我们在程序中调用上述带有形参的方法时，传入的具体数值就是实参。例如，当我们调用 add(10, 20) 方法时，数字 10 和 20 就是传递给形参 a 和 b 的实参。实参的类型必须与对应的形参类型相匹配，且数量上也需一致，否则编译器将抛出错误。

简而言之，在 Java 中，形参代表了方法所需要的、待填充的数据位置，而实参则是在实际运行时向这些位置填充的具体数值。形参与实参之间的交互是方法调用机制的核心，使得我们可以通过定义通用的方法来处理不同情境下的具体问题。

7.4 方法的返回值

在 Java 中定义一个方法时，可以为其指定一个返回值类型，如果方法需要返回结果，则在方法声明时，在方法名前面使用指定类型的关键词（如 int、String、boolean 或任何自定义对象类型等）。如果方法不返回任何值，则返回类型应为 void。

在 Java 的方法中，使用 return 语句来返回方法的值。当一个方法被定义为具有非 void 类型的返回值时（如 int、String、Object 等），return 语句用于从方法中返回一个指定类型的值给调用者。例如，在一个计算两个整数之和的方法中：

```
    public int addNumbers(int a, int b) {
        return a + b; // 返回两个参数相加的结果
    }
```

上面这段代码中，当调用 addNumbers 方法并传入两个整数时，该方法执行完毕后将通过 return 语句返回它们的和。

值得注意的是，不论方法是否有返回值，只要遇到 return 语句，方法就会立即停止执行，并返回到调用它的位置。如果方法的返回类型是 void，则 return 后面不跟任何表达式。这时 return 只是为了结束方法的执行，没有具体的值返回给调用者。例如下面这个例子：

```
    public void printMessage(String message) {
        System.out.println(message);
        return; // 结束方法执行，但不返回任何值
    }
```

在一个方法中，可以有多个 return 语句，这通常用于根据不同的选择分支返回不同的结果或者在满足某个条件时提前结束方法。例如下面的代码，有多条 return 语句。

```
    public int findMax(int a, int b) {
        if (a > b) {
            return a; // 如果 a 大于 b，则直接返回 a 作为最大值
        } else {
            return b; // 否则返回 b 作为最大值
        }
    }
```

7.4.1 返回值的数量

Java 的方法在技术上每次只能返回一个值。不过，这个 “一个值” 可以是一个复合类型，例如数组、集合（如 ArrayList、HashMap 等）、类实例或者 Java 8 中的 Optional 类来封装多个结果。

早期的编程语言，如 C 和 Pascal，受限于当时的設計理念，仅支持函数返回单一值。这种设计理念旨在确保函数行为清晰且可预测，开发者始终明确函数返回值的类型，避免处理复杂的多值返回情况。

然而，编程语言不断演进，许多现代语言，例如 Python、Go 和 Lua，开始支持函数返回多个值。这种特性为处理需要生成多个结果的函数提供了更简洁、更直接的方式，提升了开发效率和代码可读性。

例如，在 Java 中，你可以返回一个对象或数组。接下来的代码演示了如何用数组返回多个值。

```
import java.util.Arrays;

public class MultipleValues {

    // 修改方法以接受一个整数数组，并返回包含最小值和最大值的新数组
    public static int[] returnMinMax(int[] numbers) {
```

```
        if (numbers == null || numbers.length == 0) {
            throw new IllegalArgumentException(" 数组为空");
        }

        int min = numbers[0];
        int max = numbers[0];

        for (int num : numbers) {
            if (num < min) {
                min = num;
            } else if (num > max) {
                max = num;
            }
        }

        // 返回一个新的包含最小值和最大值的数组
        return new int[]{min, max};
    }

    public static void main(String[] args) {
        int[] numbers = {10, 5, 20, 3, 15};
        int[] minMaxResult = returnMinMax(numbers);

        System.out.println(" 最小值: " + minMaxResult[0]);
        System.out.println(" 最大值: " + minMaxResult[1]);
    }
}
```

以上的程序运行之后结果如下:

```
最小值: 3
最大值: 20
```

通过这样的方式，就可以返回多个值了。像上面这个例子中，为了讲解方便，我用的数组 (如果想看数组，可以向后翻几章，写书很难写的非常线性)。虽然使用数组可以实现返回多个值的功能，但在实际项目中，建议考虑以下更优方案:

o **使用集合**: 可以使用集合类（如 List 或 Set）来存储多个返回值，提供更高的灵活性。

集合将在后续章节详细讲解。

o 使用 **Optional**: Java 8 引入了 Optional 类，可以更优雅地处理可能缺失的值，避免 NullPointerException 的风险。

o 创建自定义类: 最佳实践是创建一个专门的类来封装多个相关的数据项，提供更好的代码可读性和可维护性。如何创建自定义的类将会在本书后面详细讲解。

> **写难写的代码**
>
> 学习编程不仅仅是编写简单的代码，更重要的是挑战自我，接触并理解复杂代码。阅读和编写超出当前能力范围的代码，才能真正提升编程水平，获得更深入的理解和更大的收获。
>
> 直白点，要写点超越 “hello world” 级别的代码，兜兜转转的，只搞简单的，常在河边走，快点湿湿鞋。

接下来，我们将演示一段代码，其中使用了 Java 8 引入的 Optional 类，帮助我们更好地处理可能缺失的值。

在软件开发中，空指针异常一直是程序稳定性的头号威胁。想象一下，你构建了一个功能强大的数据分析模块，但如果用户输入了空值或数据缺失，而代码没有进行充分的防御性设计，那么整个程序就可能陷入崩溃的境地。

为了应对这一挑战，Java 8 引入了一项强大的工具 —— Optional 类。它提供了一种优雅而明确的方式来表达和处理可能不存在的值，有效增强了代码的健壮性和可读性。下面的演示代码，通过使用 Optional，我们可以减少因意外空值导致的错误，确保程序在面对各种情况时都能保持稳定运行。

```
import java.util.List;
import java.util.Optional;
import java.util.Collections;

public class NumberStatisticsCalculator {

    // 方法: 寻找并返回列表中的最小值和最大值（封装在 Optional 中）
    public Optional<int[]> findMinAndMax(
                            List<Integer> numbers) {
        if (numbers.isEmpty()) {
            return Optional.empty();
```

```
            } else {
                int min = Collections.min(numbers);
                int max = Collections.max(numbers);
                // 尽管这里用 Optional 包裹，
                // 但实质上还是返回一个包含两个值的对象
                return Optional.of(new int[]{min, max});
            }
        }

        // 示例主方法来调用 findMinAndMax 方法，并处理返回结果
        public static void main(String[] args) {
            NumberStatisticsCalculator calculator =
                        new NumberStatisticsCalculator();
            List<Integer> numbersList = List.of(
                                        10, 5, 20, 3, 15);

            Optional<int[]> minMaxResult =
                        calculator.findMinAndMax(numbersList);

            if (minMaxResult.isPresent()) {
                int[] minMaxValues = minMaxResult.get();
                System.out.println(" 最小值是: "
                                    + minMaxValues[0]);
                System.out.println(" 最大值是: "
                                    + minMaxValues[1]);
            } else {
                System.out.println("
                        列表是空的，无法计算最大值和最小值。");
            }
        }
    }
```

上面这段代码引入了 Optional，它能够很好地表达方法可能不返回任何有意义的结果这一情况，提高了代码的可读性和自解释性。阅读这段代码的人能立即明白这个方法可能会因为输入列表为空而无法计算出最小和最大值。以上的代码运行之后结果如下：

```
最小值: 3
最大值: 20
```

上述两种代码实现都能够返回多个值，但仍有提升空间。将最大值和最小值封装在数组中，在语义层面显得不够直观和明确。为此，值得探讨一种更为贴切的数据结构设计，它仅包含两个属性：最大值和最小值。

幸运的是，Java 提供了 Record 类型，完美地满足了封装多个返回值的需求。相比于数组，Record 拥有诸多优势：

- 语法简洁：Record 的声明更加简洁明了，减少了冗余代码。
- 类型安全：Record 提供了更强的类型安全性，避免了类型错误。
- 自动生成访问器：Record 自动生成访问器方法，方便获取字段值。
- 不可变性：Record 是不可变的，有助于提高程序性能和安全性。

因此，在实际开发中，推荐使用 Record 来替代简单的数组存储最大值和最小值。这不仅提升了代码的简洁性和可读性，还确保了类型安全性和潜在的性能优化。

接下来的代码将演示如何使用 Record 修改代码，实现更优雅的返回值封装。

```java
import java.util.Collections;
import java.util.Optional;
import java.util.List;
import java.lang.Record;

// 定义一个记录类来表示最小值和最大值
record MinMax(int min, int max) {}

public class NumberStatisticsCalculator2 {

    // 方法：寻找并返回列表中的最小值和最大值
    // 封装在 Optional 的 MinMax 记录中
    public Optional<MinMax> findMinAndMax(
                            List<Integer> numbers) {
        if (numbers.isEmpty()) {
            return Optional.empty();
        } else {
            int min = Collections.min(numbers);
```

```
            int max = Collections.max(numbers);
            return Optional.of(new MinMax(min, max));
        }
    }

    // 示例主方法来调用 findMinAndMax 方法，
    // 并处理返回结果
    public static void main(String[] args) {
        NumberStatisticsCalculator2 calculator =
                    new NumberStatisticsCalculator2();
        List<Integer> numbersList =
                    List.of(10, 5, 20, 3, 15);

        Optional<MinMax> minMaxResult =
                  calculator.findMinAndMax(numbersList);

        if (minMaxResult.isPresent()) {
            MinMax minMaxValues = minMaxResult.get();
            System.out.println(" 最小值: "
                    + minMaxValues.min());
            System.out.println(" 最大值: "
                    + minMaxValues.max());
        } else {
            System.out.println(" 列表为空，无法计算。");
        }
    }
}
```

通过以上三段代码的展示，我们不仅了解了在 Java 中返回多个值的方法，更看到了 Java 语言不断演进，拥抱现代化特性的历程。如果上面的代码暂时有不懂的地方，先不要着急，在本书后续章节会一一介绍。

尽管 Java 诞生已久，但 Java 社区始终积极接纳并引入新特性，保持语言的活力。例如，Record 类型最初作为 Java 14 的实验性特性引入，随后成为标准功能之一，为开发者提供了更优雅的数据封装方式。

Java 的持续演进，展示了它在保持稳定性和向后兼容性的同时，也积极拥抱现代化编程理念，为开发者提供更强大的工具和更便捷的开发体验。

对于熟悉其他编程语言的开发者而言，Java 的 Record 类型与 Swift 中的 Struct、Python 中的 tuple 以及 C# 中的 Record 都有着明显的相似之处。这种跨语言的共通性反映了编程语言设计理念的融合与演进，也体现了 Java 在借鉴其他语言优点方面的努力，致力于为开发者提供更直观、高效的数据结构表示方式。

在深入探讨了方法的声明、参数传递和调用机制后，我们将转向一个特殊而重要的场景：方法调用自身，即递归。递归是编程中一个既复杂又实用的概念，接下来，我们将深入剖析递归的原理和应用，全面理解这一计算机科学中不可或缺的概念。

7.5 递归

前面我们学习了方法之间相互调用的方式，例如 A 方法调用 B 方法。现在，我们将进入一个特殊的领域——方法调用自身，即递归。

当一个方法直接或间接地调用自身时，就形成了递归。这种拥有自我调用能力的方法被称为递归方法。递归是编程中一个充满魅力且强大的工具，它可以优雅地解决一些看似复杂的问题。

许多人认为递归难以理解，但实际上，递归现象在现实生活中随处可见，只是我们通常称之为“自相似性”。

例如，用显微镜观察雪花，我们会发现雪花的微观结构与其整体形状非常相似，呈现出精美的自相似性。类似地，蕨类植物的叶片也是自相似的，每一小块叶片都如同整个叶片的缩小版，这种自相似性可以无限延伸下去。这些自然界的例子展现了递归之美，也帮助我们更好地理解递归在编程中的应用。

数学领域同样存在着递归的概念，例如著名的科赫雪花。这是一种由瑞典科学家提出的分形图形，难以用传统几何学描述，但可以通过简单的递归过程生成。

在本章的最后，我将使用 Java 代码生成科赫雪花，展示递归的强大魅力。在此之前，让我们从简单的递归例子开始，逐步深入理解递归的奥秘。

7.5.1 递归步骤与终止条件

阶乘的定义是：一个正整数 n 与小于等于 n 的所有正整数的乘积。例如，计算 5 的阶乘，需要知道 4 的阶乘，以此类推，直到 1 的阶乘。

这就是递归的思想：将问题分解成更小的子问题，并通过递归调用解决这些子问题。这种分解成规模不断缩小的过程称为“递归步骤”。为了避免无限递归，必须定义一个终止条件，当满足该条件时，递归停止并返回结果。这个终止条件在计算机中称为“基本情况”。

在阶乘的例子中，当递归到 1 时，满足了停止条件，1 的阶乘可以称为阶乘的基本情况或终止条件。将 5 的阶乘分解为 4 的阶乘，进而分解成 3 的阶乘，最后分解成 2 的阶乘，

这个逐渐趋于基本情况的过程称为递归步骤。

接下来的代码演示的是如何使用递归计算阶乘。

```
public class Factorial {
    public static void main(String[] args) {
        int number = 5; // 要计算阶乘的数字
        long factorial = calculateFactorial(number);
        System.out.println(" 阶乘结果: " + factorial);
    }

    public static long calculateFactorial(int n) {
        if (n == 0) {
            return 1; // 基本情况: 0 的阶乘为 1
        } else {
             // 递归步骤: n 的阶乘等于 n 乘以 (n-1) 的阶乘
            return n * calculateFactorial(n - 1);
        }
    }
}
```

上面的代码运行以后，会输出如下的内容：阶乘结果：120

递归在树、图、分治和动态规划等领域有着广泛的应用。然而，这些经典的递归案例通常难以用图形直观地展示。相较于抽象的过程，我更倾向于通过图形来理解编程概念。

因此，我们将深入探讨本章开篇提到的科赫雪花。这个美丽的分形图形不仅可以帮助我们更直观地理解递归，还拥有迷人的特性：它的面积有限，但周长无限，同时我们还可以计算出每一层级的周长。

通过科赫雪花的案例，我们将进一步探索递归的奥秘，并领略其图形化之美。

7.5.2 递归实现科赫雪花

正如之前所述，编写递归方法的关键在于两点：

1. **找到基本情况**: 确定递归的终止条件，即何时停止递归调用。
2. **找到递归步骤**: 定义如何将问题分解成更小的子问题，并通过递归调用解决这些子问题。

接下来，我们将探讨如何在现实中构建科赫雪花，并分析其递归过程中的基本情况和递归步骤。

初始阶段：画一个等边三角形。

迭代过程：将三角形的每个边三等分，然后以此为底再构造一个以中间线段为底的等边三角形，再去掉这个等边三角形的底边。这样，原来的一条线段就变成了四条线段，形成一个新的图形。

无限迭代：重复第二步的过程，对每个新形成的线段执行同样的操作。下图就是迭代的前四次生成的图形示例。

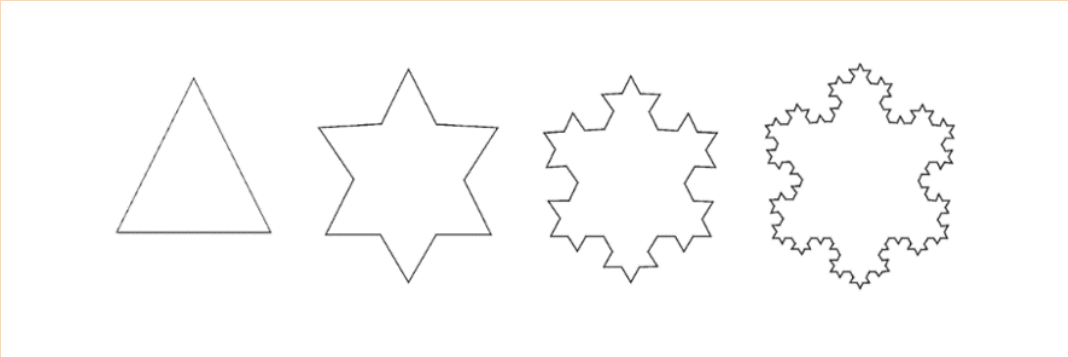

图 7.2: 科赫雪花的迭代过程

如果手工绘制科赫雪花，我们会从简单的三角形开始，逐步增加复杂度。然而，使用程序绘制时，过程则相反，需要从复杂情况逐步简化，最终达到基本情况。

在科赫雪花的递归绘制中，基本情况是深度为 0 时，程序只需绘制一条直线。其他情况则属于递归步骤，每一次递归都会使图形更接近基本情况。

绘制科赫雪花的 Java 代码如下：

```
import javax.swing.*;
import java.awt.*;

public class KochSnowflake extends JPanel {
    private int level;

    public KochSnowflake(int level) {
        this.level = level;
    }

    private void drawSnowflake(Graphics g, int level,
                               int x1, int y1, int x2, int y2) {
        int dx, dy, dx1, dy1, dx2, dy2;
```

```
        if (level == 0) {
            Graphics2D g2d = (Graphics2D) g;
            float thickness = 3;
            g2d.setStroke(new BasicStroke(thickness));
            g2d.drawLine(x1, y1, x2, y2);
            return;
        }

        dx = x2 - x1;
        dy = y2 - y1;

        dx1 = x1 + dx / 3;
        dy1 = y1 + dy / 3;

        int dx3 = x1 + dx * 2 / 3;
        int dy3 = y1 + dy * 2 / 3;

        dx2 = (int) (0.5 * (x1+x2) +
                          Math.sqrt(3) * (y1-y2) / 6);
        dy2 = (int) (0.5 * (y1+y2) +
                          Math.sqrt(3) * (x2-x1) / 6);

        drawSnowflake(g, level-1, x1, y1, dx1, dy1);
        drawSnowflake(g, level-1, dx1, dy1, dx2, dy2);
        drawSnowflake(g, level-1, dx2, dy2, dx3, dy3);
        drawSnowflake(g, level-1, dx3, dy3, x2, y2);
    }

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        drawSnowflake(g, level, 20, 280, 280, 280);
        drawSnowflake(g, level, 280, 280, 150, 20);
        drawSnowflake(g, level, 150, 20, 20, 280);
    }
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame("Koch Snowflake");
        frame.setDefaultCloseOperation(
                                JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(
                                new KochSnowflake(14));
        frame.pack();
        frame.setSize(new Dimension(640, 640));
        frame.setVisible(true);
    }
}
```

上述代码中包含了一些与图形相关的部分。由于目前企业很少使用 Java Swing 构建图形用户界面，因此本书不会深入讲解这方面的内容。代码中使用的图形组件非常基础，主要功能只是绘制直线。我会进行简单的介绍，确保不会对理解代码的核心逻辑造成太大影响。

KochSnowflake 类继承自 JPanel，用于展示科赫雪花。方法 KochSnowflake(int level) 接收一个参数 level，表示生成科赫雪花的层数。drawSnowflake 方法是一个递归函数，用于绘制科赫雪花的每一条边。当 level 为 0 时，它会直接在两个点之间绘制一条线；否则，它会将线段分成三段，形成一个等边三角形的中间部分，然后再对这四个新的线段进行相同的操作。paintComponent 方法是 JPanel 的一个重要方法，用于绘制面板的内容。在这个方法中，我们调用 drawSnowflake 方法三次，以绘制科赫雪花的三条边。

这个程序将会生成一个窗口，显示一个具有指定层数的科赫雪花。你可以通过修改 new KochSnowflake(10) 中的数字，来更改科赫雪花的层数。例如，new KochSnowflake(5) 将会生成一个具有 5 层的科赫雪花。

科赫雪花的复杂度随着层数的增加而呈指数级增长。一层雪花有 12 条线段，二层有 48 条，三层则达到 192 条。这种指数级增长的复杂性不仅赋予了科赫雪花视觉上的复杂和美感，也意味着计算机在渲染高层级雪花时需要消耗大量的计算和内存资源。

因此，层数过大可能导致程序运行缓慢或内存不足。在实际应用中，需要根据硬件性能选择合适的层数，通常绘制 5 到 10 层即可获得良好的视觉效果。在我的 32GB 内存电脑上，绘制 15 层雪花已经接近性能极限。

以下是程序生成的 5 层、10 层和 15 层科赫雪花，仅凭肉眼观察，已难以分辨其差异。

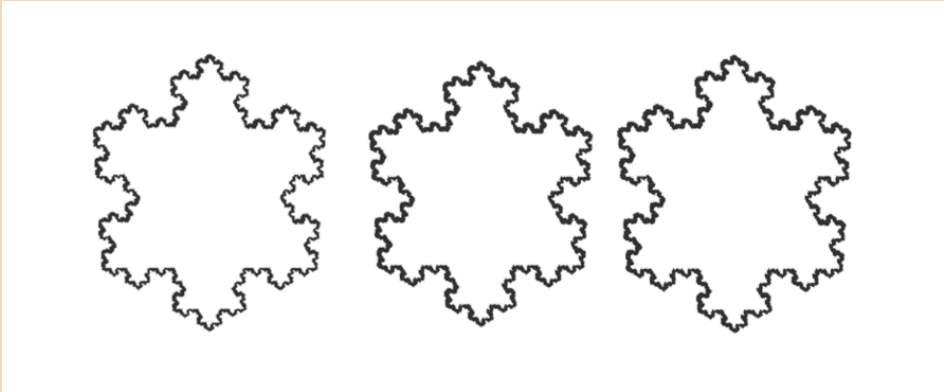

图 7.3: 5 层、10 层、15 层的科赫雪花

建议大家亲自运行代码，感受指数增长对程序性能的影响。从不到 1 秒生成 5 层图形，到十几分钟才能生成 15 层图形，这种巨大的差异体现了指数增长的威力。

这也是递归的缺点之一。递归调用涉及方法的入栈和出栈操作，这些操作会带来额外的开销。相比于使用迭代或循环的非递归解决方案，递归通常更消耗时间和资源。

7.6 程序员故事

《与我同龄的路由器》

“这台设备都要比我年龄大了。”当我坐在机房的地上，用螺丝刀用力拧已经生锈的螺丝，汗水不停的落在我的手背上时，我这么想。

为了找到这台机器，我已经在机房转了很多圈，每一个机架我都看过了，终于在一个没有标签的机架里找到了这台白色的机器，红色的旧金山金门大桥的标志印在白色的壳上，这是一台 1986 年思科公司生产的 Cisco Systems AGS 路由器。我想：“工作了快 20 年以后，它的寿命应该快结束了吧？”

在几个小时以前，厂区一半的计算机不能联网了，我被电话吵醒，凌晨 2 点从被窝里拖出来，在冰雪覆盖的路面上颠簸了 20 公里，来到这家发电厂查找故障。这次排除故障从头到尾都很糟糕。我一到这个电厂，两个人接待我，一胖一瘦，有点像金庸小说里的胖头陀与瘦头陀。虽然体型差异巨大，但是两个人都非常的沉默加忧郁，铁青的脸，好像随时在准备倒霉。我下车之后，想寒暄几句，握个手什么的，结果两人开门见山的说：“网络不通了，你快点弄好，要在早班上班前搞好。”这句话一出，让我刚伸出去的手触电一样缩了回来。

在去机房的路上，我问了几个问题，一问三不知，只知道网络不能用了。

这还不算最糟，我刚进机房，很意外的看到一个矮小的中年男人在抽烟，我心里嘀咕：“机房里还能抽烟，怪不得一问三不知。”哪想这个中年男人接我心里的话说道：“因为是夜

班我才抽烟的，要不然真熬不住。”怪不得心理学家说什么“思想是不出声的语言”，这个男人肯定是有什么招风耳，我想什么他都能听到。

我初步看了一下，大概有 20 多个机架，这些机器分属不同功能，有做办公自动化的，有做财务的，不一而足。最伤脑筋的事情是没人真正搞清楚这些线路的走向，毕竟 20 年了，网络在架床叠屋的环境下，能搞清楚的人越来越少。看完以后，我就知道，这次有不少苦头在等着呢。看着网线像意大利面一样盘在地上，我就知道今天可能要在这个机房里呆上一整天了。

“小伙子，你觉得怎么样？”抽烟男子问道，“网络不通了，是吧？没什么大事吧，以前也有个工程师，挺厉害的，他来搞几下就通了。书本上的东西是死的，经验是活的。”

当年我虽然从业不久，但是类似的话我听了不少，总有这种“以前的工程师”能神奇的把网络搞好。我只能报以礼貌的微笑。

“你参加工作多久了？”这个男人总有问不完的问题。

“半年多吧。”

“才半年？”他的脸上总算露出的笑容，仰头从口中吹出一根长长的烟柱，“嗯，是个新手啊，以前的那个工程师可是有 15 年工作经验呢，他可真厉害，我就没见过他搞不定的故障。他可真是个好手。”

说实在的，我在任何时候都不介意一个人对另一个人赞美，只是今天的这些话，让我实在是有些不舒服。我就沉默不语。

胖头陀和瘦头陀一直没开口，但是他们三人很自然的坐在一起，看着我一根一根的检查网线，在这个年久失修的机房里摸爬滚打。胖头陀和瘦头陀眯着眼睛看着，吸烟男则不停的评判着。在这个风雪交加的夜里，他一根接一根的吸烟，越吸越快，小眼睛兴奋的像烟头一样红，恐怕这么多年来都没有今天这么开心过吧。

愤怒像风暴一样慢慢的在我脑海中形成，我强忍着，还是一根一根的检查网线，试图找出网络的结构。一边寻找，一边用捆线器把那一堆像意大利面一样的网线整理的井井有条。功夫不负有心人，我终于找到了，那么多昂贵的设备传输的数据，都要通过这台 1986 年生产的机器路由传输出去去。现在这台设备已经不再运转了。

我总算可以继续开展我的工作了：“是这台路由器坏了，现在要更换一台路由器，这台路由器用的时间太久了，现在不工作了。”

“不可能吧，换新的没问题，如果换了新的还不工作怎么办？”吸烟男不死心的问。我没空再搭理他，就对胖头陀说：“这台机器是 1986 年生产的，可能刚建机房的时候，就已经买了这台机器。”胖头陀向瘦头陀使了个眼色，示意他去取一台新的路由器过来。

“不可能是 1986 年生产的，1986 年的设备还能运行？我在这里呆了快 10 年了，哪台设备是什么情况我门儿清，我可以跟你打赌，绝对不可能是 1986 年的设备。”吸烟男不依

不饶的说道。

一会儿，路由器拿过来了，我把旧的路由器拆下来，里面的背板上清晰的写着 1986 年制造，我递给吸烟男看。他狠狠的吸了一口烟，不敢相信自己的眼睛：“好家伙，真是 1986 年生产的啊，这么久的设备，竟然还能运行，真是见了鬼了！”吸烟男的活力好像一下子没了。

新的路由器被安装在机架上，我把网线接上，再接上我的计算机，把路由器配置好。一切都开始工作了，胖头陀打了个电话询问同事网络能不能用了？在得到了肯定的答复以后，我如释重负。我一面穿上自己的外套，觉得好像跑了个马拉松，混身的肌肉都在痛，嘴巴很干。我把东西收拾好，走出机房，外面的天已经有些亮了。雪中夹着小小的冰粒从天空飘落，把我的眼睛打的有点痛。

虽然我不同意吸烟男大部分的话，但是有一句话我还是挺认同的，他说“书本上的东西是死的，经验是活的。”

是啊，以从事信息工作的人来说，书上没讲的东西太多了。书上不会讲如何在雪夜里开车，书上不会讲如何面对客户的不友好，书上还不会讲如何处理绝望的情绪。我突然想起来当年的教科书上的一幅插画，在编程的时候，干净的房间里，客户在微笑，程序员在微笑，项目经理在微笑，甚至计算机屏幕上还画了一个微笑的表情符号，一幅和谐完美的画面，好像在聚餐一样。画里没有意大利面一样的网线，脸上没有汗水，来回的路上也没有雪。

这也是一本编程的书，我希望告诉大家这些事。在编程中，在项目实施中，总有无数的十字路口，大家要习惯，这些十字路口中没有红绿灯，过路口的时候一定要加倍小心，照顾好你自己。

8 数组

Contents

在之前的章节中，我们简要介绍了数组的基本概念，并展示了它在循环结构中的应用实例。然而，这只是冰山一角，数组的功能远不止于此。本章将深入探讨数组的概念、用途以及操作方法，为您揭开数组的神秘面纱，展现其强大的功能和广泛的应用场景。

8.1 数组的概述

要准确地考证数组这一数据结构的起源已非易事，但可以肯定的是，在计算机发展的早期阶段，数组就已经被广泛采纳并应用于各种计算场景之中。值得一提的是，1945 年约翰 · 冯 · 诺伊曼提出的归并排序算法就利用了数组这一结构进行高效的排序处理。总之，数组作为最基本的数据结构之一，其历史沿革与计算机科学的发展历程紧密交织，无疑是所有数据结构中最悠久且应用最为广泛的一种。

鉴于数组这一数据结构悠久的历史，Java 自 1.0 版本的标准库中就已内置了数组。同时，Java 标准库中还包含了 Vector 和 Hashtable 这两个关键的类。然而，随着技术发展，在 Java 1.2 版本引入更为先进且功能丰富的集合框架后，Vector 和 Hashtable 逐渐淡出，被 ArrayList、LinkedList 等集合类所取代。

尽管如此，数组在 Java 编程世界中的地位依然坚不可摧，应用场景十分广泛。在现代实际编程实践中，虽然 ArrayList 已经近乎完全承担起了原先由数组直接处理的诸多职责，但实质上，ArrayList 底层的核心实现机制依旧依赖于数组。接下来，我将深入剖析 Java 中数组的概念及其应用。

8.2 数组的声明

Java 数组是 Java 编程语言中用于存储固定数量、同类型数据的集合。在内存中，数组是一个连续的区域，每个元素可以通过其索引（从 0 开始）来访问和操作。Java 中的数组既可以存放基本数据类型，也可以存放引用数据类型，但是同一个数组中只能同时存同一种类型的元素。

在 Java 中使用数组之前，必须先声明数组。声明数组意味着定义一个数组类型的变量，但此时并未分配具体的内存空间来存储数组元素。声明数组的目的是告诉 Java 编译器你打算使用一个特定数据类型（如 int、String 等）的数组，并为此数组变量命名。声明数组的方法如下：

```
数据类型[] 数组名;
数据类型 数组名[];
```

大家可能会奇怪为何会有两种声明的方法，我也没有确切的答案。这两种声明方式在 Java 语言规范中是等效的，本质上并无区别。虽然在 Java 中这种写法同样有效，但在实际开发中第二种方法较少使用，因为根据 Java 的普遍编码规范（如 Oracle 官方推荐的 Java 编程规范），为了提高代码的可读性和一致性，建议采用第一种格式，即将方括号紧贴数据类型。我猜测，之所以支持第二种，可能是因为在 C/C++ 中，声明数组是用 int array[5] 这样的方式。

声明数组之后，还不能使用数组，因为声明只是创建了一个索引，并没有分配内存空间来存储元素。对数组进行元素分配并设置初始值之后才可以使用，而这个过程，就是数组的初始化。

8.3 数组的初始化

Java 中，初始化数组是指为数组分配内存空间，并为其元素赋予初始值的过程。初始化 Java 数组主要有两种方式：静态初始化和动态初始化。

8.3.1 静态初始化

静态初始化是指在创建数组的同时直接为其赋值。这种方式可以明确地指定数组中每个元素的具体值，并且编译器会根据初始值的数量自动计算数组的长度。

通过下面的演示代码，看看如何定义以及初始化数组。

```
class Example {
    // 静态初始化一个基本数据类型（整型）数组
    private static int[] integerArray = {
                                10, 20, 30, 40, 50 };

    // 静态初始化一个引用数据类型（字符串）数组
    private static String[] stringArray = { "Apple",
          "Banana", "Cherry", "Date", "Elderberry" };

    public static void printArrays() {
        System.out.println("Integer Array:");
        for (int value : integerArray) {
            System.out.println(value);
        }

        System.out.println("\nString Array:");
        for (String fruit : stringArray) {
            System.out.println(fruit);
        }
    }
}

// 使用该类打印数组内容
public class DataExample {
    public static void main(String[] args) {
        Example.printArrays();
    }
}
```

在上述例子中，首先创建了一个名为 DataExample 的类，其中包含了两种类型的静态初始化数组：一种是基本数据类型（整型）数组 integerArray，另一种是引用数据类型（字符串）数组 stringArray。通过 printArrays() 方法分别展示了如何遍历并打印这两种类型数组的内容。运行之后输出的结果如下：

```
Integer Array:
10
```

```
20
30
40
50

String Array:
Apple
Banana
Cherry
Date
Elderberry
```

8.3.2 动态初始化

在 Java 中，数组的动态初始化是指在声明数组时指定其长度，但不立即为数组元素赋值。具体来说，动态初始化的过程包含两个步骤：

第一步，声明数组变量：声明一个数组变量，但并不分配内存空间给数组。其语法如下：

```java
// 声明一个整数类型的数组
int[] dynamicArray;
// 声明一个字符串类型的数组
String[] stringArray;
```

第二步，分配内存并设置默认值：使用 new 关键字来创建数组对象，并为其分配指定长度的内存空间。此时，数组中的每个元素将被赋予该数据类型的默认初始值。例如，对于整型数组，所有元素都会被初始化为 0；对于引用类型数组，则会被初始化为 null。其语法如下：

```java
// 动态初始化一个长度为 5 的整型数组，所有元素默认为 0
dynamicArray = new int[5];
// 动态初始化一个长度为 10 的字符串数组，所有元素默认为 null
String[] stringArray = new String[10];
```

动态初始化后，程序员可以在后续代码中通过索引访问和修改这些已经分配了内存并具有默认值的数组元素。例如：

```
for (int i = 0; i < dynamicArray.length; i++) {
    dynamicArray[i] = i * 2; // 给数组元素逐个赋值
}
```

总结一下，相比于静态初始化，动态初始化提供了灵活性，允许开发者先声明数组长度而不必立即提供具体的元素值，随后再根据需求填充数组内容。接下来，我将前面静态初始化的例子，用动态初始化的方式重写一下，详见如下的代码：

```
class ArrayInitializer {

    // 动态初始化一个基本数据类型（整型）数组
    private static int[] integerArray;

    // 动态初始化一个引用数据类型（字符串）数组
    private static String[] stringArray;

    static {
        // 初始化整型数组并赋值
        integerArray = new int[5];
        for (int i = 0; i < integerArray.length; i++) {
            integerArray[i] = i * 10 + 10;
        }

        // 初始化字符串数组并赋值
        stringArray = new String[5];
        stringArray[0] = "Apple";
        stringArray[1] = "Banana";
        stringArray[2] = "Cherry";
        stringArray[3] = "Date";
        stringArray[4] = "Elderberry";
    }

    public static void printArrays() {
        System.out.println("Integer Array:");
        for (int value : integerArray) {
            System.out.println(value);
        }
```

```
        System.out.println("\nString Array:");
        for (String fruit : stringArray) {
            System.out.println(fruit);
        }
    }
}

// 使用该类打印数组内容
public class DataExample2 {
    public static void main(String[] args) {
        ArrayInitializer.printArrays();
    }
}
```

以上代码运行后的结果与静态初始化的示例相同，在此不再赘述。

静态初始化和动态初始化的主要区别在于：

1. 静态初始化在声明数组的同时就设置了初始值，而动态初始化需要在声明数组之后再设置初始值。
2. 静态初始化只能使用数组初始化器 {} 来设置初始值，而动态初始化可以使用任何方法来设置初始值。

如果在声明数组时就知道数组元素的初始值，建议使用静态初始化，这样代码更简洁。如果在声明数组时还不知道数组元素的初始值，则需要使用动态初始化。

8.4 多维数组

Java 中的多维数组是数组的嵌套结构，可以用来存储具有多个维度的数据。在二维数组中，每个元素本身又是一个一维数组；在三维数组中，每个元素则是一个二维数组，以此类推。这种数据结构非常适合于处理表格数据、图像像素矩阵或任何其他需要两个或更多索引来定位特定值的情况。

8.4.1 二维数组

Java 中的二维数组是一种多维数组，它可以在内存中表示一个表格或矩阵形式的数据结构。在二维数组中，每个元素都有两个索引，通常称为行索引和列索引。由于二维数组跟前面讲的一维数组差不多，所以下面就简略的介绍一下：

8.4.1.1 声明与初始化二维数组

跟一维数组一样，在 Java 中声明和初始化二维数组有两种方式，第一种是先声明数组变量再创建，代码如下：

```
int[][] matrix; // 声明一个未初始化的二维整数数组
matrix = new int[3][4]; // 创建一个 3 行 4 列的二维数组
```

第二种是直接声明并初始化，代码如下：

```
// 声明并创建一个 3 行 4 列的二维数组
int[][] matrix = new int[3][4];
```

或者使用初始值直接初始化，代码如下：

```
int[][] matrix = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

8.4.1.2 访问二维数组元素

访问二维数组中的元素需要提供两个索引，第一个索引代表行（row），第二个索引代表列（column）。方法如下：

```
int value = matrix[rowIndex][colIndex];
```

例如，获取上述示例中第 1 行第 2 列的元素：

```
// 此处将得到值为 2
int secondElementOfFirstRow = matrix[0][1];
```

8.4.1.3 遍历二维数组元素

可以通过嵌套循环来遍历二维数组的所有元素，代码如下：

```
// 遍历行
for (int i = 0; i < matrix.length; i++) {
    // 遍历列
```

```
        for (int j = 0; j < matrix[i].length; j++) {
            // 输出当前元素
            System.out.print(matrix[i][j] + " ");
        }
        // 换行以便区分不同的行
        System.out.println();
    }
```

最后，综合以上的用法，我们实际来运行一个例子，代码如下：

```
public class TwoDArray {

    public static void main(String[] args) {

        // 声明并初始化一个 2 行 3 列的整数型二维数组
        int[][] twoDArray = new int[2][3];

        // 动态初始化二维数组
        twoDArray[0] = new int[] { 1, 2, 3 };
        twoDArray[1] = new int[] { 4, 5, 6 };

        // 直接初始化二维数组
        int[][] anotherTwoDArray = {
                { 10, 20, 30 },
                { 40, 50, 60 }
        };

        // 访问二维数组中的元素
        // 输出: 20
        System.out.println(anotherTwoDArray[0][1]);

        // 遍历二维数组
        for (int i = 0; i < anotherTwoDArray.length; i++) {
            for (int j = 0; j < anotherTwoDArray[i].length;
                                                    j++) {
                System.out.print(
                                anotherTwoDArray[i][j] + " ");
```

```
            }
            System.out.println(); // 每行结束后换行
        }

    }
}
```

以上程序运行之后，结果如下：

```
20
10 20 30
40 50 60
```

8.4.2 三维数组

Java 中的三维数组是一种多维数组，它扩展了二维数组的概念，增加了第三个维度的索引，从而能够表示更复杂的数据结构，如立体空间中的数据点、图像的体积数据（例如 RGB 立方体）或其他任何需要三个独立索引来定位元素的情况。下面的演示了三维数组的用法。

```
public class ThreeDArray {

    public static void main(String[] args) {
        // 声明一个三维数组，名为 array，可以存储 1 行、2 列、3 个元素
        int[][][] array = new int[1][2][3];

        // 初始化数组
        for (int i = 0; i < array.length; i++) {
            for (int j = 0; j < array[i].length; j++) {
                for (int k = 0; k < array[i][j].length;
                                                    k++) {
                    array[i][j][k] = i + j + k;
                }
            }
        }

        // 遍历数组并输出元素值
```

```
            for (int i = 0; i < array.length; i++) {
                for (int j = 0; j < array[i].length; j++) {
                    for (int k = 0; k < array[i][j].length;
                                                    k++) {
                        System.out.println("array[" + i + "]
                                ["  + j + "]["  + k + "]
                                = " + array[i][j][k]);
                    }
                }
            }
        }
    }
```

这个例子展示了如何声明、初始化和遍历 Java 三维数组，解释一下上面的代码：

int[][][] array：声明一个三维数组，名为 array，可以存储 3 行、4 列、5 个元素。

array.length：获取数组的第一维长度，即行数。

array[i].length：获取数组的第二维长度，即列数。

array[i][j].length：获取数组的第三维长度，即元素个数。

for (int i = 0; i < array.length; i++)：使用嵌套循环遍历数组，并输出每个元素的值。

以上的代码运行之后，结果如下：

```
array[0][0][0] = 0
array[0][0][1] = 1
array[0][0][2] = 2
array[0][1][0] = 1
array[0][1][1] = 2
array[0][1][2] = 3
```

总结一下，一维数组是一种简单的数据结构，它可以存储一系列同类型的数据。一维数组的优点是简单易用，易于理解和实现。多维数组是一种复杂的数据结构，它可以存储多维数据。多维数组的优点是灵活，可以表示更复杂的数据结构。相比来说，一维数组的使用频率更高，因为它更简单易用。接下来，我用一维数组实现一个现实中的算法，实现本福特定律。

8.5 用数组实现本福特定律

为了更好地展现 Java 数组的实用价值，我们可以利用其特性来实现一个现实生活中的著名数学定律——本福特定律。在深入探讨如何运用 Java 数组对该定律进行编码之前，有必要先阐述一下本福特定律的基本内涵。该定律又称作首位数字法则，它揭示了一种普遍存在于各种自然及人为产生的数据集中的规律性现象：即无论数据集合多么庞大和复杂，其中各个数的首位非零数字分布呈现出一种独特的频率趋势。这个定律最早由 20 世纪 30 年代的美国物理学家弗兰克·本福特发现并以他的名字命名，他观察到，在大量的实际数据集中，较小的数字如 1 和 2 作为首位有效数字出现的概率相对较大，并且这一概率遵循着一定的数学模式。接下来，我们将借助 Java 数组来模拟和验证这一引人入胜的现象。

具体来说，本福特定律表明在许多自然和人工的数据集中，首位数字 d（在 1-9 之间）的频率符合以下公式：

$$P(d) = \log_{10}(d+1) - \log_{10}(d) = \log_{10}\left(1 + \frac{1}{d}\right)$$

这意味着首位数字为 1 的概率以 30.1% 的显著比例占据主导，而 2 紧随其后，作为首位数字出现的概率约为 17.6%，以此类推，直至 9，其作为首位数字的概率仅为 4.6%，这一系列分布规律出人意料地违背了直觉预期——原本我们认为所有数字作为首位的概率应大致相等。然而，正是这种独特的概率分布特点赋予了本福特定律强大的实用价值，它可以用来检验数据的真实性。接下来，我们将运用 Java 编程语言编写一个程序，通过实际计算来验证所给定的数据集是否遵循本福特定律的预测结果。

另外，由于数据需要的量比较大，我把数据存入一个文本文件中。程序中的 data.txt 文件存放在 chapter8/benford 文件夹里，这个文件包含了 30 多万个数据。详细的用数组实现本福特定律的代码如下：

```
import java.util.Scanner;
import java.io.File;

public class Benford {
    public static void main(String[] args)
                                throws Exception {
        // 分别存放数字出现的次数
        int [] count = new int[10];
        // 统计总共输入了多少数字
        int n = 0;

        Scanner scanner = new Scanner(new File("data.txt"));
```

```
            while (scanner.hasNextInt()){
                int x = scanner.nextInt();
                while ( x >= 10 ) {
                    x = x/10;
                }
                count[x]++;
                n++;
            }

            scanner.close();

            for(int i = 1; i < 10; i++) {
                // 输出结果
                System.out.printf("%d: %6.1f%%\n",
                                  i, 100.0 * count[i] / n);
            }
        }
    }
```

上面的代码运行以后，结果如下：

```
1:   30.8%
2:   19.3%
3:   13.0%
4:    9.9%
5:    7.4%
6:    5.9%
7:    5.2%
8:    4.4%
9:    4.1%
```

通过上面的结果可以得知，数字 1 确实在 30% 左右。使用本福特定律可以验证很多信息，比如会计有没有做假账，选举中有没有假选票，这个规律已经被应用于很多方面，在可靠性方面也得到了认可。如果你对这个定律有兴趣，请搜索 testingbenfordslaw 来查看更多与实际生活相关联的例子，也可以找到相关的数据，用上面的程序跑一下。

上述代码实例展示了 Java 中对数组进行定义、初始化、更新以及访问等基础操作。

8.6 程序员故事

《令人头痛的代码》

上次雪夜修那台与我同龄的路由器回来后，我想看一看公司的软件到底可以支持多少种硬件。如果现出现故障，至少可以显示一些有用的信息，比如通过 SNMP 标准能返回设备信息，而不用再两眼一摸黑搞得自己精疲力竭。

中午和潘新吃饭的时候，我把这个想法和他讲了一下。我说："我想研究一下咱们软件中处理不同设备的代码，看看有没有可能加上一些代码，如果设备出了故障，至少在软件中能个报错误出来。"

听到我讲出这个想法，潘新抬起头来看着我，半张着嘴，许久说不出话来。看那表情，仿佛有千言万语要说，这些话一股脑的涌上喉咙，像下班时间的马路，塞的结结实实的，一个字也没说出来。

过了许久，他才说："你不要内卷，如果你要去查这些分外之事的代码，那么我也要去做一些分外之事，这会让我的工作强度更高了。我可不想让自己累着，再说了，这些代码又不是不能跑，你碰它们干什么。"

没有死心，下班回去以后，我找到了相关代码，不看不知道，一看吓一跳。一个长达 15000 多行的源文件，里面有很多条判断语句，每条判断语句处理一种设备。当每加入一种新的设备，就要在这个文件中多增加一条判断语句。确实很让人头痛。

我想起了《代码大全》上给出的解决方法，如果判断语句过多，可以改用表格驱动法，我又拿起了这本重达 2 斤，近 10 厘米厚，一直垫在显示器下的书仔细研究了起来。

对程序员来说，从键盘上敲出的代码可以分成两种，一种代码是负责软件运行的逻辑，另一种代码是要处理的数据。相比于软件运行的逻辑，添加要处理的数据要简单的多。

我正在思考这件事的时候，敲门声响了，我才想到今天是周五了，楼下的退休老师又要来找我了。一个刚退休不久的高中数学老师的确要寻找新的生命活力，王老师喜欢上了买足球彩票，虽然王老师从来没看过足球，根本不知道足球队和球员的名字，但是他仍旧喜欢买足球彩票。

有次我偶尔也去买一次玩玩，在彩票点碰到了他，当他发现我竟然对于球队，球员和战绩还能侃侃而谈的时候，而且我还是个程序员，从他的面部表情可以看在出来，他对我越来越敬重。于是每到周五，他就借口给我送本书，或者修一下手机什么的，来探讨一下买足球彩票的问题。

我只中过一次二等奖，不到 300 块钱，又加上他认为程序员和计算机有某种神秘的联系——那可是计算机啊，他曾经这么说过——我就成了他心中掌握了某种魔法的超级足球预言家。我一直保留着这唯一的一次中奖的凭证，只要有机会我就拿出来给别人炫耀一下，

即使我早就赔进去了 1000 多块钱。我曾经给王老师看过我中奖的凭证，他赞叹不已，王老师一次也没中。

退休后的王老师没去跳广场舞，没每天早晨去公园用肚子顶树，而是研究起了足球彩票，在得知我是程序员以后，竟然也想学习编程了。他有一台女儿淘汰的计算机，在了解了他学编程的目的是为了辅助分析足彩以后，我推荐他可以试试.net 编程，这个可以很容易在他的计算机上做一个客户端，并且我把几本书送了他。

王老师就是这样一个人，对别人的事情不关心，别人也不能干预他的事情。我曾经亲眼看到他动手打了一个跟王老师的妻子推销理财产品的人，拿着扫把追着别人打，周围的人总称他为教书把脑子教坏了的“疯子”。

对我来说，他的脑子一点都没坏，我内心为他鸣不平。经过了短短几个月，期间虽然也问过我一些这样那样的问题，但是，他的彩票分析软件竟然真的搞起来了。这次，又周末了，估计又要问我点什么东西，每次他就来 20 分钟，还能吃一顿王太太做的美味食物。

我开门以后，还没坐定，王老师像往常一样开门见山的问道：“小刘啊，我问你个问题，我想统计一下所有球队的战绩，每打完一场，我就要更新一次代码，用你教我的那个判断语句，每次都要写一次，麻烦的很，有没有办法让我把每次比赛的战绩输入在一个文件里，就可以不去修改代码了呢？”

我突然意识到，王老师碰到的问题就是我要解决的问题：把数据与逻辑分离。

我大体解释了一下如何做，如何读取文件，如何处理以后，王老师很开心的离开了。我就打开那个长达 15000 行的代码，提取出了里面要支持的设备，总共有 1600 多种设备。与众多的设备相比，操作并不复杂，这 1600 多种设备总共支持 9 种操作，每种设备都从这 9 种操作中选出一种来。

这是非常清晰的逻辑，完全用不着 15000 多行代码做这件事情。我思考了一会儿，只要把这 1600 多种设备分成 9 个文件，每个文件对应一种操作就可以了。我又花了一些时间，把这 1600 多种设备分类，把原来的 9 种操作分别写了一个方法，原有的 15000 行代码被压缩成了 200 来行，毕竟那 15000 行代码，绝大部分都是重复的判断语句。

以后再添加设备，只需要添加到相应的 9 个文件之一，代码完全不需要改动。

周一上班以后，我把代码给潘新与潘闻看，他们俩个都表示我代码修改的非常好。我笑了，这一幕是我所最爱的，把一堆混乱的代码整理得井井有条，并且能得到同事们的认可，感觉这是平淡生活中少有的乐趣。

9 类、对象和封装

Contents

在本书第 3 章中，我们简单的介绍了面向对象。在那一章中，我曾提到设计一个类很简单，真正的难点在于如何妥善处理多个类之间的关系。这与现实生活颇为相似：一个人生活很容易，人和人之间的关系则很难。

从本章开始，我们将从技术细节上来学习面向对象编程。在 Java 面向对象编程中，有三个核心概念，它们被称为面向对象的三大特性，分别是：封装、继承和多态。

封装（Encapsulation）是指将数据（对象的状态）和操作数据的方法绑定在一起，作为一个独立的整体（即类）进行处理。通过访问修饰符如 private、protected 和 public 来控制对类内部成员变量和方法的访问权限，从而实现数据隐藏和外部对内部细节的隔离。继承（Inheritance）允许一个类（称为子类或派生类）从另一个类（称为父类或基类）继承已定义的状态和行为。这样子类就可以复用、扩展或修改父类的功能。在 Java 中，继承通过 extends 关键字实现。

多态（Polymorphism）是指同一个接口，不同对象可以有不同的实现方式，或者同一个方法名在不同的上下文中表现出不同的行为。多态有两种形式：静态多态（编译时多态，通过重载实现，即同一个类中有多个同名方法，参数列表不同）和动态多态（运行时多态，通过继承和接口实现，即子类重写父类方法后，在父类引用指向子类实例时调用方法时表现出子类的行为）。通过第 3 章的历史介绍，我们可以知道面向对象编程语言中的这三大特征并不是同时一次性引入的，而是随着面向对象编程思想的发展和完善逐步形成的。

封装的概念最早可以追溯到 Simula 67 语言，它是面向对象编程的先驱之一，其中包含了对实体和过程的封装。随后 Smalltalk 等语言进一步发展了封装的概念，并将其作为面向对象的核心原则之一。

继承的概念也在早期面向对象语言中得到体现，比如 Smalltalk 同样支持类的继承，使得子类能够扩展或修改父类的属性和行为。

至于多态，虽然在一些早期语言中存在类似机制，但现代面向对象编程中普遍接受的多态形式——尤其是在 Java 中所体现的运行时多态（通过虚函数或接口实现）是在 C++ 和后来的 Java 等语言中得到广泛应用的。

因此，可以说这些特征是在面向对象编程发展的不同阶段逐渐成熟并被各种编程语言采纳和完善的。在 Java 中，这三大特征都是语言设计的基本组成部分，开发者在设计类和系统时会同时运用它们来构建复杂且灵活的软件结构。在讲解的时候，本书本着从易到难的策略，先来介绍如何设计一个独立的类，也就是封装。

9.1 类与对象

在第 3 章的时候，我已经简略的讲了类与对象的关系。在此简要的回顾一下：类（Class）如同一种模板或蓝图，它定义了一类事物的共有属性（如姓名、年龄、能力值等）和行为（如射门、传球等）。以《FIFA》游戏为例，“球员类（Player）”就是这样一个模板，它预先规定了所有球员的基本信息和动作。

而对象（Object）则是类的实例化结果，是根据类的具体定义创建出的个体实体。例如，在游戏中，梅西和 C 罗就是依据“球员类（Player）”分别创建出的两个对象，他们各自拥有独特的属性值（如两人的年龄、所在球队等），同时都能执行类中定义的方法（如射门动作）。简而言之，类是抽象的定义，对象是类的具体体现。

Java 封装的原则体现了面向对象编程的核心思想，其核心概念是：隐藏内部实现细节，只通过公开接口进行交互。隐藏内部实现细节，是将对象的状态（属性或数据成员）隐藏起来，不让外部直接访问。通过公开接口进行访问则是通过定义清晰的公共接口（方法）来与外部交互，这样即便内部实现发生变化，只要接口不变，就不会影响到使用该类的其他部分。

从技术上来说，Java 使用 class 关键字来定义一个类，其框架如下：

```
class Player {

}
```

在这个简化的例子中，Player 是类的名字。在 Java 编程中，class 关键字是用来定义一个新的类的。类是面向对象编程的基本构造块，它可以包含属性（也称为字段或变量）和

方法（函数）。通常情况下，类会用来描述某种实体或概念的结构和行为。

接下来，我们将逐步将这个空 Player 类完善，对其添加相应的属性与方法、访问修饰符和构造函数等内容。

9.2 属性与方法

在 Java 编程语言中，类扮演着对象蓝图的角色，它详尽刻画了对象所拥有的属性特征和行为表现。类的构成要素主要包括两大部分——属性和方法。

属性（又称成员变量）：在球员类的场景中，属性用来界定球员的具体状态和特性。例如，一个球员类的属性可能涵盖姓名、年龄、身高、体重、球衣号码、擅长位置等基本信息，这些属性既可以是整型数值（如年龄、身高），也可以是字符串类型（如姓名）或其他复杂类型（如位置可以用一个枚举类型来表示）。

方法（又称成员函数）：类的方法则对应球员的各种行为和能力。在球员类中，方法可能包括奔跑、射门、传球、防守等一系列动作，这些方法内部包含了执行相应动作所需的指令序列。例如，一个球员类可以定义一个名为 shoot() 的方法来模拟射门动作，此方法不仅能够访问和操作球员的属性（比如力量值可能会影响射门力度），还可以与其他方法联动（如射门前先进行调整姿势）。

总之，在 Java 中，通过精心设计类的属性和方法，开发者能够精确模拟现实世界中各类实体（如球员）的内在特质与外在行为，进而构建出功能丰富、逻辑清晰的对象模型。

在本书的第 3 章中，我们分析了球员类，以及球员类拥有的属性与方法，并画出了球员类的 UML 图，为避免重复，请翻到第 3 章查看当时画的 UML 图。接下来，以下是实现的代码：

```
public class Player {
    // 定义属性
    String name;
    String position;
    int skillLevel;

    // 定义球员的行为方法
     void run() {
        System.out.println(name + " 正在奔跑...");
    }

     void shoot() {
```

```
        System.out.println(name + " 正在射门！");
    }

    void pass() {
        System.out.println(name + " 正在传球给队友...");
    }

    public static void main(String[] args) {
        // 创建一个 Player 对象
        Player player = new Player();
        // 为 Player 对象的属性赋值
        player.name = "C 罗";
        player.position = " 前锋";
        player.skillLevel = 95;
        // 调用 Player 对象的方法
        player.run();
        player.shoot();
        player.pass();
    }
}
```

上面的程序运行之后，结果如下：

```
C罗 正 在 奔 跑 ...
C罗 正 在 射 门 ！
C罗 正 在 传 球 给 队 友 ...
```

上面的代码虽然能够正常执行，但实际上存在显著的问题。主要体现在两个方面：首先，该类并未实现数据隐藏原则，在外部可以随意的修改对象的变量。其次，它未能提供公共接口以供外部访问其内部数据。为改进这些问题，我们可以对类的属性进行访问权限的控制，并提供相应的公开方法来进行数据访问和操作。接下来，看看 Java 都是提供了什么样的解决方案。

9.2.1 访问修饰符

在 Java 编程语言中，为了实现类的属性安全封装，即禁止外部直接访问和修改属性，特意引入了访问修饰符这一关键特性。访问修饰符主要用于管理和控制类内部属性与方法的可见范围和访问层级。Java 中共定义了四种访问修饰符，分别为：public（公共访问）、protected（受保护访问）、默认（无特定修饰符的包访问权限）以及 private（私有访问）。

针对封装这一面向对象设计原则，最为直接相关的便是 private 和 protected 修饰符。private 修饰符确保类的属性和方法仅在类的内部可见，严格限制了外部对类内部状态的直接访问。而 protected 修饰符虽然放宽了访问权限，允许同包下的其他类以及所有子类访问，即允许子类访问超类的部分内部实现，关于继承的内容，下一章会详细讲解。

至于 public 和无修饰符的默认访问权限，它们与封装的关联相对较弱，因此在本章里不做过多讲解。尽管如此，它们在类中仍发挥着重要作用，只是在涉及封装概念时，它们的重要性不如 private 和 protected 那样直接明显。待到涉及继承特性和类之间的协作时，它们的价值便会进一步体现出来。接下来，详细讲一下 private 这个修饰符。

private（私有）：使用 private 修饰的成员只对该类可见，其他任何类都无法访问。这种访问修饰符用于实现封装，确保类的内部实现细节对外部代码隐藏起来。

在前一小节所示的球员类（Player）代码示例中，如果我们对 Player 类中的各个属性——如姓名（name）、位置（position）和技能等级（skillLevel）——均采用 private 访问控制修饰符进行声明，那么这些属性将在类的外部将无法被访问。换言之，一旦将这些属性设定为私有，外部代码将不能直接读取或修改这些属性值的能力，从而确保了 Player 类内部数据的封装性与安全性。

但是这样，也引入了另一个问题，如果要访问或修改这些私有属性，应该怎么办呢？Java 也提供了相应的解决方法，那就是需要给球员类（Player）提供公共访问接口，如 getter 和 setter 方法，才能合法且安全地操作这些属性值。

9.2.2 getter 和 setter 方法

当我们将类的属性设置为 private 之后，确实需要提供相应的途径以便外部代码在必要时能够安全地访问或修改这些属性。Java 中常用的做法是为每个私有属性编写一对公共的 getter 和 setter 方法。

getter 方法（getXxx 形式）：Getter 方法用于返回私有属性的值。例如，球员类有一个私有属性 name，我们可以创建一个名为 getName 的方法来获取这个名字。getter 方法的目的是暴露私有属性的当前值，但不允许外部直接修改它。

```
public String getName() {
    return this.name;
}
```

setter 方法（setXxx 形式）：Setter 方法则是用来设置或修改私有属性的值。同样以 name 为例，我们会创建一个名为 setName 的方法，允许外部传入一个新的名字来更新属性。

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

通过这种方式，虽然实际的属性被封装起来，但外部代码依然可以通过调用这些公共方法来间接地访问和修改私有属性。同时，这种方法还为以后可能添加的数据验证、日志记录、事件触发等功能预留了空间，因为在 setter 方法内部可以很容易地添加额外的逻辑处理。这正是 Java 语言中封装原则的一种典型实现方式，既保证了数据安全，又实现了灵活的控制和扩展。

接下来，我们将运用上述所提及的访问修饰符以及 getter 与 setter 方法，对先前的球员类（Player）进行重构与优化。具体而言，我们将把球员类内部的属性，诸如姓名（name）、位置（position）和技能等级（skillLevel）等类属性，设置为 private 以实现封装，随后提供相应的公共 getter 和 setter 方法，以确保在维持数据安全性的同时，外界能够访问和修改这些私有属性。下面的代码是经过调整后的球员类代码：

```
public class Player {

    // 将属性设置为 private，实现封装
    private String name;
    private String position;
    private int skillLevel;

    // 提供 getter 方法以获取私有属性的值
    public String getName() {
        return this.name;
    }

    public String getPosition() {
    return this.position;
    }

    public int getSkillLevel() {
        return this.skillLevel;
    }

    // 提供 setter 方法以设置私有属性的值
    public void setName(String name) {
        this.name = name;
    }

    public void setPosition(String position) {
```

```
        this.position = position;
    }

    public void setSkillLevel(int skillLevel) {
        this.skillLevel = skillLevel;
    }

    // 定义球员的行为方法保持不变
    public void run() {
        System.out.println(getName() + " 正在奔跑...");
    }

    public void shoot() {
        System.out.println(getName() + " 正在射门！");
    }

    public void pass() {
        System.out.println(getName() + " 正在传球给队友...");
    }

    // 示例主方法，使用 getter 和 setter 为 Player 对象的属性赋值
    public static void main(String[] args) {
        // 创建一个 Player 对象
        Player player = new Player();
        // 使用 setter 方法为 Player 对象的属性赋值
        player.setName("C 罗");
        player.setPosition(" 前锋");
        player.setSkillLevel(95);
        // 调用 Player 对象的方法
        player.run();
        player.shoot();
        player.pass();
    }
}
```

现在，Player 类的属性已经被封装，外部代码无法直接访问和修改这些私有属性，只能通过提供的 getter 和 setter 方法进行操作。这样不仅增强了数据安全性，也为后续可能的

业务逻辑扩展提供了便利。这段代码运行的结果与前面那个版本相同，在此不再重复。

如果读者有诸如 swift 或者 kotlin 语言的经验，会觉得 Java 实在是太繁琐了，竟然需要自己写 getter 与 setter，别的语言都是根据情况自动完成的。在 Java 中可以么？简单来说，Java 不支持自动生成 getter 与 setter。但是，有变通的方法，而且是两个方法。一个是通过 IDE 来完成，另一个是通过第三方的包，下面我分别介绍一下。

9.2.2.1 通过 IDE 自动生成 setter 与 getter

如果你是使用的 IntelliJ IDEA，只需要在属性字段上右键，在弹出的菜单里选择“重构”，然后再在弹出的菜单里选择“封装字段”，会弹出如下的窗口。在这个窗口中选择想要添加的 setter 和 getter 即可。

图 9.1: 使用 IntelliJ IDE 自动生成 getter 和 setter

使用 IDE 自动生成 setter 与 getter 之后，代码会自动生成到相关的类文件中。但是相比于 swift 语言和 kotlin 语言，仍然不够整洁，可以采用第二种方法。

9.2.2.2 使用 lombok 开源库

我非常推荐这个开源库，它是一款非常受欢迎的 Java 开源库，其主要目的是通过注解处理器（Annotation Processor）机制，在编译阶段自动为 Java 类添加常见的样板代码，以

此减少手动编写重复且冗余的 Getter/Setter、构造函数、equals()、hashCode() 和 toString() 方法等工作，提高开发效率。

如果读者有兴趣的话，可以深入研究一下 Lombok 的原理。其原理也不复杂，当你在一个类中定义了私有属性，并使用了 Lombok 注解 @Data，编译时 Lombok 将自动生成所有必要的访问器（getter/setter）、equals()、hashCode() 以及 toString() 方法。不仅如此，Lombok 还提供了其他一系列注解如 @NonNull（空值检查）、@NoArgsConstructor, @AllArgsConstructor（无参构造器和全参构造器）、@Builder（创建建造者模式代码）等，进一步简化代码编写过程。

为了在集成开发环境中（如 Eclipse, IntelliJ IDEA）正常使用 Lombok，需要安装对应的 Lombok 插件，这样 IDE 就能够在编辑时就识别并应用 Lombok 注解，使得开发者在源代码级别就可以看到 Lombok “生成” 的效果，同时在编译时这些注解会被转换成实际的字节码。

总结来说，Project Lombok 能够帮助 Java 开发者保持代码简洁，集中精力于业务逻辑实现，而不是处理大量的模板代码。对 getter 和 setter 来说，仅需要在类中加入 @Getter 与 @Setter 即可。接下来，看看使用 lombok 之后的代码：

```
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Player {

    private String name;
    private String position;
    private int skillLevel;

    // 定义球员的行为方法保持不变
    public void run() {
        System.out.println(getName() + " 正在奔跑...");
    }

    public void shoot() {
        System.out.println(getName() + " 正在射门！");
    }
```

```
    public void pass() {
        System.out.println(getName() + " 正在传球给队友...");
    }

    public static void main(String[] args) {
        // 创建一个 Player 对象
        Player player = new Player();
        // 使用 setter 方法为 Player 对象的属性赋值
        // 这些方法由 Lombok 自动生成
        player.setName("C 罗");
        player.setPosition(" 前锋");
        player.setSkillLevel(95);
        // 调用 Player 对象的方法
        player.run();
        player.shoot();
        player.pass();
    }
}
```

再来回顾一下，看看上面的代码还有什么可以改进的地方？我们创建了一个 Player 对象，并通过一系列 setter 方法逐一为其属性（如姓名、位置和技能等级）进行了赋值。如果每次创建对象时都需要手动调用 setter 方法，这无疑增加了代码的冗余度和易错性。此外，若忘记为某些属性设置初始值，可能会导致对象处于不完整或无效的状态。

为此，Java 提供了一种更为优雅且高效的方式——构造函数（Constructor），用于在创建对象的同时初始化其属性。构造函数是一种特殊的、与类名相同的方法，它在对象实例化时自动执行，确保对象在诞生之初就能拥有完整的状态。

接下来，我们将探讨构造函数的概念，学习如何定义和使用构造函数，以便在创建对象时一次性初始化所有的属性，从而简化代码，提升程序质量和可读性。

9.2.3 构造函数

构造函数（Constructor）是 Java 中的一种特殊类型的方法，用于在创建对象时执行初始化操作。构造函数的名称必须与类名相同，并且没有返回类型（甚至不是 void）。当通过关键字 new 创建类的实例时，构造函数会自动调用，用于初始化对象的状态。

大家可以有个疑问，先前定义的球员类（Player）根本没有定义构造函数，那 Java 会怎么处理呢？如果在类中没有显式定义构造函数，Java 会自动提供一个默认的无参构造函数。默认构造函数会将所有成员变量初始化为默认值（如数值型变量初始化为 0，布尔型

变量初始化为 false，引用类型变量初始化为 null）。

接下来，我们从技术细节上来看看 Java 中类的构造函数。

9.2.3.1 构造函数的命名

构造函数的命名规则在 Java 语言中具有严格的规范，即其名称必须精确地匹配其所在类的名称，同时也要保持相同的大小写模式。例如，在名为 “Player” 的类中，相应的构造函数必须被命名为 “Player”，这样才能确保 Java 虚拟机正确识别并调用该构造函数以初始化新创建的对象实例。

9.2.3.2 构造函数没有返回类型

构造函数没有返回类型，包括 void 也不允许。构造函数与其他方法的一个显著区别在于它没有返回类型。这是因为构造函数的主要任务是在内存中为新对象分配空间，并初始化对象的状态，而不是像普通方法那样返回一个值。当创建类的新实例时，构造函数自动执行，最终的结果是生成并返回一个新对象的引用。因此，构造函数隐含地返回新创建的对象实例，而不是通过明确的返回类型表达。

9.2.3.3 构造函数可以有 0 个或者多个参数

当有 0 个参数的时候，被称之为无参构造函数，当不是 0 个参数的时候，被称为有参构造函数。我分别来讲一下这两种构造函数：前面提过，当一个类没有定义任何构造函数时，Java 编译器会默认提供一个无参数的构造函数，称为默认构造函数。

> **! 重要**
>
> 值得注意的是，在 Java 中，如果你没有为类定义任何构造函数，编译器会为你提供一个默认的无参数构造函数。然而，如果你为类定义了至少一个构造函数（无论这个构造函数是否带有参数），编译器就不会再提供默认的无参数构造函数。

这个构造函数通常什么都不做，或者只是初始化成员变量至它们的默认值。当然了，开发者也可以显式地为类定义一个无参构造函数，以便在创建对象时不传递任何初始值，或者实现自定义的初始化逻辑。例如下面的代码段中，开发者可以自定义初始化逻辑：

```
public class Player {
    public Player() { // 无参构造函数示例
        // 可能的初始化代码
    }
}
```

一般来说，开发者定义的是有参构造函数，可以具有任意数量和类型的参数，这些参数用于在创建对象时初始化对象的成员变量。每个不同的参数列表对应一个不同的构造函数，这种特性使得我们可以根据实际需求传递不同的参数来初始化对象的不同状态。接下为，我举一个实际的例子吧，详细代码如下所示。

```
public class FootballPlayer {
    private String name;
    private int age;
    private String nationalTeam; // 国家队名称

    // 添加 getter 方法以获取属性值
    public String getName() {
        return this.name;
    }

    public String getNationalTeam() {
        return this.nationalTeam;
    }

    // 无参构造函数，设置默认值
    public FootballPlayer() {
        this.name = " 未知球员";
        this.age = 0;
        this.nationalTeam = " 未加入任何国家队";
    }

    // 有参构造函数，初始化姓名和年龄
    public FootballPlayer(String playerName,
                          int playerAge) {
        this.name = playerName;
        this.age = playerAge;
        // 默认情况下，nationalTeam 留空或者设置为默认值
        this.nationalTeam = " 未加入任何国家队";
    }

    // 构造函数重载，同时初始化姓名、年龄和国家队名称
    public FootballPlayer(String playerName,
```

```
                int playerAge, String nationalTeamName) {
        this.name = playerName;
        this.age = playerAge;
        this.nationalTeam = nationalTeamName;
    }

    // 示例主方法，展示构造函数的使用
    public static void main(String[] args) {
        // 使用无参构造函数创建对象
        FootballPlayer unknownPlayer =
                            new FootballPlayer();
        System.out.println(unknownPlayer.getName());
        System.out.println(unknownPlayer.getNationalTeam());

        // 使用包含姓名和年龄参数的构造函数创建对象
        FootballPlayer player1 =
                        new FootballPlayer("C 罗", 36);
        System.out.println(player1.getName());
        System.out.println(player1.getNationalTeam());

        // 使用包含姓名、年龄和国家队名称参数的构造函数创建对象
        FootballPlayer player2 = new FootballPlayer(
                              " 梅西", 34, " 阿根廷国家队");
        System.out.println(player2.getName());
        System.out.println(player2.getNationalTeam());
    }
}
```

在这个例子中，FootballPlayer 类包含了三个构造函数，它们具有不同的参数列表。当创建对象时，可以根据实际需要传递不同的参数，Java 编译器会根据传递的参数类型和数量自动选择相应的构造函数来初始化对象，这就是构造函数重载在 Java 中的应用。

在这个类中，三个构造函数的区别是参数不同。其中两个有参构造函数的区别是有一个参数，代表球员是否被征召加入国家队，当被征召以后，调用三个参数的构造函数。当没有被征召，调用两个参数的构造函数。以上代码运行以后，结果如下：

```
未知球员
未加入任何国家队
```

C 罗
未 加 入 任 何 国 家 队
梅 西
阿 根 廷 国 家 队

在上面讲 setter 和 getter 以及构造函数的过程中，使用了一个至关重要的关键字——this。这个关键字在 Java 中代表着当前对象的引用，它在构造函数和其他方法中扮演着独特的角色。它不仅能够帮助我们清晰地标识和访问当前对象的成员变量，还能实现构造函数之间的互相调用。

因此，理解并熟练运用 this 关键字，不仅可以简化代码，提高程序的可读性，更能提升代码质量。接下来，我们将具体分析 this 在类中的具体用法。

9.2.4 如何使用 this 关键字

在 Java 中，this 关键字是一个非常重要的关键字，它代表了当前类的实例引用。在类的方法中，主要有以下三种用途：

第一，引用当前对象。在类的方法或构造函数内部，this 关键字代表当前对象的引用。这意味着可以通过 this 来访问当前对象的成员变量或调用成员方法。比如上述演示代码中如下的片段：

```
// 添加 getter 方法以获取属性值
    public String getName() {
        return this.name;
    }
```

在这段代码中，this 指代当前的对象实例，并且使用使用 this 来访问该实例的成员变量 name。

第二，避免与局部变量冲突。当方法参数或局部变量与成员变量同名时，使用 this 关键字可以明确地区分并引用成员变量。比如如下的代码片段：

```
public void setName(String name) {
        // 这里为了避免与参数 name 混淆，
        // 使用 this.name 来明确引用成员变量
        this.name = name;
}
```

在这段代码中，方法参数的名称是 name，同时，成员变量中亦有一个名为 name 的变量，为了能区分这两个变量，可以在成员变量名前加上 this 以示区别。

第三，调用当前类中其它的构造函数。可以通过 this(…) 的形式调用本类的其他构造函数，从而实现构造函数的复用。

```
public class MyClass {
    private String name;
    private int age;

    public MyClass(String name) {
        // 调用带两个参数的构造函数，并给 age 赋默认值
        this(name, 0);
    }

    public MyClass(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

在上述 Java 代码中，我们定义了一个名为 MyClass 的类，其中包含了两个私有成员变量 name 和 age。该类中编写了两个构造函数，分别为 MyClass(String name) 和 MyClass(String name, int age)。

首先，对于构造函数 MyClass(String name)，它接收一个字符串类型的参数 name，并在函数体内通过 this(name, 0) 调用了该类中的另一个构造函数 MyClass(String name, int age)。在这里，this 关键字用于在同一个类的不同构造函数之间进行相互调用，将当前构造函数接收到的 name 参数传递给另一个构造函数，并为 age 参数赋予一个默认值 0。

值得注意的是，在此类构造函数重载调用场景中，this 调用语句必须置于构造函数体的首行。这是因为在实例化对象时，构造函数的执行是从首行开始的，若 this 调用语句出现在非首行，可能会导致成员变量在被正确初始化之前就被访问或赋值，引发潜在的问题。通过在构造函数 MyClass(String name) 的第一行调用 MyClass(String name, int age)，确保了在初始化对象时遵循正确的初始化顺序，从而保证了成员变量的正确初始化和使用。

总的来说，this 关键字在 Java 中是一个非常重要的工具，它可以帮助我们在类的内部引用其自身的实例。通过 this，我们可以在构造函数、方法或者块中引用类的字段和方法，这对于区分参数和实例变量，以及在类的内部调用其他构造函数等场景非常有用。

9.2.5 静态成员

在本节中，我们将深入探讨 Java 编程语言中的静态成员——静态变量/常量和静态方法——的概念。静态变量与静态常量的关键差异在于它们的值是否可变：静态变量在程序执行期间其值是可以被修改的，而静态常量则在初始化之后始终保持不变，这一特点源于其声明时额外使用了 final 关键字加以限定。换言之，静态变量与静态常量的本质区别在于是否具备持久不变性，静态常量本质上是对某一固定值的持久引用。

首先，先讲静态变量。在 Java 中，静态变量归属于类层级，而非类的任何特定实例。这意味着，无论我们创建了多少个该类的实例，系统中仅存在一份静态变量的副本，所有实例均共享这份单一的静态变量资源。比如，我希望追踪某特定类的实例创建次数，由于每次实例化对象必然涉及调用构造函数，便可在构造函数内部利用静态变量实现计数功能。每当构造函数被执行一次，即意味着一个新对象被创建，此时静态变量随之累加。通过这种方法，便可借助于静态变量的特性，能够在不依赖于对象个体的前提下，有效地对类级别的全局状态进行管理和跟踪。

其次，再讲静态方法。静态方法也是类级别的成员。它们同样不与类的任何特定实例关联，程序员可以直接通过类名调用静态方法，无需事先实例化类。这一特性使得静态方法在执行与类本身相关联的操作时十分便利，且常常用于提供与类的状态无关的工具函数。比如，在先前所提及的场景中，通过定义一个静态方法，能够轻松实现直接基于类调用而非实例对象，实时输出已创建的对象个数。通过这种方式，即便不创建类的实例，也能方便地调用与类本身紧密相连的方法。

接下来，我将前面提到的例子实现出来，以展示如何运用静态变量统计类创建的对象个数。下面是展示静态变量与静态方法的代码示例：

```
public class Player2 {
    // 静态变量用于统计创建的对象个数
    private static int objectCount = 0;

    // 构造函数，每次调用时增加对象计数
    public Player2() {
        objectCount++;
    }

    // 静态方法，用于获取当前创建的对象个数
    public static int getObjectCount() {
        return objectCount;
    }
```

```
    // 示例：直接通过类名调用静态方法获取创建的对象个数
    public static void main(String[] args) {
        Player2 obj1 = new Player2();
        Player2 obj2 = new Player2();

        // 输出当前已创建的对象个数，无需通过实例调用
        // 输出：2
        System.out.println(Player2.getObjectCount());
    }
}
```

静态变量是类级别的变量，不属于任何一个实例，而是属于类本身。所有的实例都共享同一个静态变量。在这段代码中，objectCount 是一个静态变量，用于统计 Player2 类创建的实例个数。无论创建了多少个 Player2 的实例，objectCount 都是共享的，每创建一个实例，onjectCount 就增加 1。

静态方法也是类级别的，它可以直接通过类名调用，而不需要创建类的实例。在这段代码中，getObjectCount() 就是一个静态方法，它返回创建的 Player2 实例总数，即静态变量 objectCount 的值。因为是 getObjectCoun() 静态的，所以可以直接通过 Player2.getObjectCount() 来调用它，而不需要创建 Player2 的实例。

接下来，再关注静态常量这一特性。静态常量一旦初始化后，它的值将永久固定，不能再做修改。这一特性使静态常量成为理想的选择，用于表示那些在应用程序整个生命周期中始终保持稳定不变的数值，如圆周率 π 和自然对数的底数等数学元素。在软件开发中，当需要用到如绝对值计算、三角函数运算等不依赖于特定对象实例且不会随着运行环境改变的基础数学操作时，为每个对象实例单独复制这些固定功能实属冗余。因此，通过定义静态常量和静态方法，我们能够确保这些不变的操作和数值在整个程序中共享。

在编程实践中，直接声明和使用自定义静态变量的情况相对有限，但对 Java 预定义的静态常量的需求却相当普遍。尤其是像 Java 标准库 java.lang.Math 类所提供的那些基本数学常量，它们在各类数学计算和几何处理中扮演着不可或缺的角色。下面是从 JDK 源代码摘录的部分 Math 类的静态成员定义：

```
// 自然对数的底数 e
public static final double E = 2.7182818284590452354;
// 圆周率 π
public static final double PI = 3.14159265358979323846;
```

```
// 静态方法举例：计算绝对值
public static int abs(int a) {
    return (a < 0) ? -a : a;
}
```

在这段代码中，final 是一个关键字，用于声明一个常量，也就是说，一旦赋值后，就不能再改变它的值。这两行代码定义了两个静态常量 E 和 PI。这两个常量的值在初始化后就不能再改变。

final 关键字在 Java 编程语言中具有多重作用，不仅限于变量声明，还可以应用于类以及方法的定义。在先前提及的例子中，当用 final 来限定一个变量时，该变量即被赋予了不可变性属性，这意味着一旦该变量获得初始赋值，其后续便无法再进行重新赋值操作。

在本书后面讨论继承的时候，我们还会碰到 final 关键字在继承中的应用场景。其中，当 final 修饰一个类时，此类被视为终态类，即不允许任何其他类对其进行继承，从而确保了类设计上的封闭性。同样地，若 final 被用来修饰一个方法，则此方法具备不可覆盖（Overriding）的特性。也就是说，在子类中无法重写一个被声明为 final 的方法，这有助于保持方法行为的一致性和避免意料之外的多态性。不过，在当前阶段，对于 final 关键字在类和方法上应用的具体细节，我们暂时不做深入探讨。

接下来，看看如何使用定义好的静态常量跟静态方法吧，下面代码清单 9-6 这段代码展示了如何在 Java 中使用 java.lang.Math 类的静态常量和静态方法来进行数学计算。

```
import java.lang.Math;

public class MathExamples {
    public static void main(String[] args) {
        // 使用静态常量
        System.out.println(" 自然对数的底数 e: " + Math.E);
        System.out.println(" 圆周率 π: " + Math.PI);

        // 使用静态方法计算绝对值
        double number = -12.3;
        System.out.println(" 绝对值是: " + Math.abs(number));
    }
}
```

9.3 程序员故事

《潘闻要改进 Java 语言》

也许在编程史上最戏剧化的小插曲之一就是 windows 编程日益衰落了。短短的几年时间，所有的 windows 程序员都不见了。而我不知道该说有幸还是不幸见证了这个改变。

当我在上大学的时候，最让人感兴趣的是给微软的 windows 编程，《Programming Windows》的书有几厘米厚，然而几乎是一夜之间，大家都开始给 web 和手机开发软件了。现在，我才意识到，当年死记硬背那么多方法是没有任何必要的。

一天早上，我如常的上班，坐在那里查资料。潘闻坐在我旁边，看着非常的勤快，他的鼠标在快速的点击，显示器上也显示着一大堆英文的资料，目不转睛的看着显示器，头向前伸着。我的余光看到潘新也在偷看他，脸上带着温柔的笑容。

看到他们兄弟这样的亲近，我也不免高兴起来。每当他们两个在争吵，我就会很难过，看起来，今天会是个好日子。潘新走到潘闻的旁边，问他弟弟在看什么？潘闻回答道：“在看一些 Java 的设计规范，也许我能改进一些。”听到弟弟这么说，潘新的眼睛里发出阵阵光芒，语速越来越快，他一股脑的把他所知道的知识全都倒给他的弟弟，末了，转过头对我说：“我今天出去有点事，你和潘闻来聊聊，看看他想如何改进 Java。”说完，便一阵风似的出门了。

就在关门的同时，我看到潘闻的身子突然松驰了下来，开始端着茶杯一边喝茶一边上网。我试探的问潘闻：“你真打算改进 Java？”“看来你还是不了解情况。”潘闻说，他吹了吹茶杯的热气，笑得露出一口发亮的白牙，“我昨天在看 Java 的方法，感觉很简单，但是仔细一想又不简单，我说我想改进，只是为了让我哥高兴一下，不过我心里确实有几个疑问，我们来商量一下吧。”

潘闻说道：“你有没有觉得我们的代码写的太复杂了？”

“是啊，无论你做什么，用户的需求都会变的，所以软件自然就越做越复杂了。”我回答说。

“确实如此，前些天客户想找出所有带宽占用不超过 30% 的路由器。昨天又来问我能不能找出周末带宽占用不超过 30% 的路由器，这种需求真是让人头疼。有没有什么方法可以处理咱们这种情况呢？我倒是挺想改进点什么，比如，Java 语言什么的。”

我听了以后，微微一笑，每个程序员都会碰到这种情况，其实，每个程序员碰到的新问题早就被先前的程序员碰到过，并且提出了解决方案。我问小潘说：“你可曾听说过有个叫行为参数化的技术？”

“没有，你解释一下吧。”

“打个比方来说吧，我们上了这么多年学，已经习惯了三点一线，知道怎么去学校，去食

堂，回家。我们知道怎么去学校，怎么学英语，数学等。这相当于调用了一个 go_and_learn() 的方法。学校里的培养，总归是为了去社会上当个螺丝钉。于是，你毕业后，去公司打工，要做的事情其实和学校里没太大区别。老板让你去某个地方，拿些文件，学习这份文件。这还是相当于调用了 go_and_lean() 的方法。两者的区别可以通过不同的参数来体现，这里不同的参数是行为。”

“我有点明白你的意思了，我就是想要把用户的行为当作一个参数传进去，这样，我就不用再频繁的修改代码了。Java 有这样的机制？”

“当然有这样的机制了，而且还不止一种这样的机制。”

“那讲一讲吧。”

“在 Java 中，你可以使用匿名类的方式来把行为当作参数传递到方法里，也可以使用 Lambda 方法来实现更简洁的代码形式。”“我听说过 Lambda 方法，只是觉得它比较难，好像也没看到很多人用 Lambda 方法。是不是使用了 Lambda 方法，能让 Java 的功能变得更强？”

“Lambda 方法是 Java 8 引入的，现在很多年过去了，接受程度已经比较高了。原则上，Lambda 方法最主要的功能是让原来繁琐的代码变的简洁。原来做行为参数化的时候，一般是采用匿名类来表示多种行为，那样做的方式让人十分不满意：代码非常啰唆。啰嗦的代码会直接影响程序员在实践中使用行为参数化的积极性。”

小潘说道：“原来 Java 还有这么多没有大规模应用的特性啊？”

“是的，Java 一直在不停的进化，甚至进化的有点太快了，新的特性层出不穷，以至于根本没有推广，下一个新特性就出来了。”“推出这么多新的特性，如果没什么用，那干嘛还要不停的推出呢？”

我说道：“我推荐你读一篇论文吧，是 1966 年的一篇论文，作者是著名的计算机科学家 Peter Landin，他写了一篇名为《The Next 700 Programming Languages》的论文，预测了接下来的 700 种编程语言是什么样子，其中就有 Lambda。”

“竟然预测了 700 种编程语言，看来我想改进 Java 语言的想法还是太幼稚了。”

“当然不算幼稚了，以后还会出现越来越多的语言，哪个程序员不希望自己创造一种编程语言呢？编程语言就像生态一样，新的语言会出现，旧的语言会消亡。每个人都希望会出现一种语言能一统江湖，可是现实中不是这样。每种语言都有自己擅长的领域，C 与 C++ 在操作系统和嵌入式领域有很大的份额，因为它们运行的效率很高。Java 和 C# 则的企业领域站稳了脚跟。只要你有心去做，一切都是有可能的。”

“那好吧，我还是先去找你说的论文来读一下，看看 50 多年前的人是如何预测的吧。”

10 包

Contents

上一章，我们完成了对封装这一面向对象原则的学习。封装通过为类的属性设置访问限制，并通过 getter 和 setter 方法进行交互，实现了数据隐藏和对内部细节的隔离。在实际开发大型软件项目时，由于类的数量众多，仅仅依靠类的封装已经无法满足命名空间隔离的需求。随着项目的增长，类的数量会迅速增加，如何有效地组织这些类并防止命名冲突成为了关键问题。为此，Java 提供了一种强大的机制——包（package）。

Java 包是一种将相关类和接口组织在一起的方式，通过合理的结构划分，不仅可以降低类名冲突的风险，还可以提高代码的复用性和可维护性。包机制还允许我们实施访问权限控制，通过默认访问修饰符限定包内成员的可见范围。

接下来的内容，我们将步入 Java 包的世界，探讨其基本语法、用途以及最佳实践。通过本章的学习，您将会了解如何声明包、组织包结构、导入包中的类以及利用包来增强代码的分层与模块化设计。

10.1 包的简介

在 Java 中，包是一种用于组织类、接口、枚举和注释的机制。它类似于文件夹，可以将相关的类组织在一起，方便管理和使用。

在大型项目中，可能会有许多类和接口，如果不使用包进行组织，相同的类名可能导致混淆和编译错误。通过将相关的类和接口置于不同的包中，可以确保即使类名相同，只要包名不同，这些类也能在项目中和平共存。因此，Java 中使用包，主要有以下两个用途：

第一，包提供了对代码的逻辑分组和组织结构，使得开发者可以按照功能、模块或组件来划分代码。这种组织方式极大地提升了代码的可读性、可维护性和可重用性。

第二，包还提供了一种隐式的访问控制机制，即包访问权限。默认情况下，包内的类和接口只对同一包内的其他类可见，除非特别指定为 public。这增强了封装性，有助于隐藏实现细节，并允许开发者控制类的可见范围和使用方式。

这两个功能共同促进了 Java 应用程序的结构清晰、模块化和安全可控，是 Java 编程实践中至关重要的组成部分。

10.2 语法和用法

10.2.1 如何声明 package

在 Java 中声明包是为了组织和管理类，提供命名空间以避免类名冲突，并控制类的可见性。在 Java 源代码文件中，package 声明必须位于文件的顶部，作为非注释的首行代码。声明 package 的格式如下：

```
package [包名];
```

这里的 [包名] 是一个由点分隔的标识符序列，通常采用反向域名的方式命名，例如公司网站的域名反转加上项目或模块名称。

假设有一个名为 com.liuyandong.myapp 的包，其中包含了一个名为 MyClass.java 的类文件。在 MyClass.java 文件中，声明包的代码如下：

```
// liuyandong.com 项目的 MyClass 类
package com.liuyandong.myapp;

// 此处开始放置类的定义
public class MyClass {
    // 类的属性和方法...
}
```

在这个例子中，MyClass 类被声明在 com.liuyandong.myapp 这个包下。按照 Java 的规范，对应的.class 文件也应该保存在与包名匹配的目录结构下，即：

图 10.1: class 文件在电脑上的目录结构

这样，当其他 Java 类想要使用 MyClass 时，要么需要处于同一包下，要么需要通过 import 语句明确导入。接下来，我们讲一下如何导入 Java 包。

10.2.2 如何导入包

在 Java 中，导入（import）包是为了能够在当前类中使用到其他包中定义的类或接口。导入包是通过在 Java 源文件的顶部使用 import 关键字来完成的。导入包一般分为如下四种情况：

10.2.2.1 导入单个类或接口

如果你想导入某个特定包下的一个类或接口，可以直接写出类或接口的全限定名（fully qualified name）。在 Java 中，全限定名是指一个类或接口的完整名称，包括包名和类名或接口名，用点号分隔。

```
// 导入 java.util 包下的 ArrayList 类
import java.util.ArrayList;

public class Example {
    // 可以直接使用 ArrayList 类，无需完整路径
    ArrayList<String> list = new ArrayList<>();
}
```

10.2.2.2 导入整个包

如果需要导入一个包下的所有类，可以使用星号（*）通配符。但请注意，这种方式应谨慎使用，因为它可能会增加类加载时间和潜在的命名冲突风险。

```
// 导入 java.util 包下的所有类
import java.util.*;
```

```
public class Example {
    // 使用 List 接口和 ArrayList 类，因为已经导入了 java.util.*
    List<String> list = new ArrayList<>();
}
```

10.2.2.3 静态导入

除了常规导入外，还可以使用 import static 关键字来导入某个类中的静态成员（如静态方法或静态字段）。

```
// 导入 Math 类中的静态常量 PI
import static java.lang.Math.PI;

public class Example {
    public static void main(String[] args) {
        // 直接使用 PI，无需 Math.PI
        System.out.println(PI);
    }
}
```

上面的代码导入了 java.lang.Math 类中的静态常量 PI，并在 main() 方法中直接使用 PI。使用 import static 的好处有两点：一是简化代码，提高代码的可读性。二是避免重复写类名或接口名来限定静态成员。有利必有弊，如果导入的静态成员与其他类或接口中的成员同名，使用静态导入可能会造成命名冲突。

10.2.2.4 导入自定义包中的类

对于自定义的包，假设有一个名为 com.liuyandong.myapp 的包，里面有一个类 MyClass，导入它的方式如下：

```
// 导入自定义包中的 MyClass 类
import com.liuyandong.myapp.MyClass;

public class AnotherClass {
    // 可以直接使用 MyClass 类
    MyClass myObject = new MyClass();
}
```

10.2.3 如何使用 package 中的成员

在 Java 中，要使用 package 中的成员（类、接口、枚举、注解等），首先需要正确导入对应的 package 或类。然后就可以像使用当前 package 内的成员一样使用导入的 package 中的类或接口。下面是一个具体的例子：

假设我们在 com.liuyandong.math 包中有一个名为 Calculator 的公共类，该类中有一个计算两数之和的方法 add()。

首先，在 Calculator.java 中声明 package 并编写类：

```
// Calculator.java 文件
package com.liuyandong.math;

public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

然后，在另一个需要使用 Calculator 类的程序中，首先导入该类：

```
// 导入 Calculator 类所在的包
import com.liuyandong.math.Calculator;

public class MainProgram {
    public static void main(String[] args) {
        // 创建 Calculator 类的对象
        Calculator calc = new Calculator();
        // 调用 Calculator 类的 add() 方法
        int sum = calc.add(10, 20);
        System.out.println(" 两数之和为: " + sum);
    }
}
```

在上面的例子中：

我们首先在 MainProgram.java 中导入了 com.liuyandong.math 包中的 Calculator 类。然后在 main() 方法中，我们创建了 Calculator 类的一个实例，并通过这个实例调用了其公开方法 add() 来计算两个整数的和。最后，我们将计算的结果打印出来。

这样，我们就成功地在另一个包中的类中使用了 com.liuyandong.math 包中的成员。

10.3 最佳实践

10.3.1 如何命名包

在 Java 中，包的命名遵循一系列规范和最佳实践，以下是一些指导原则：

第一：全小写。包名应全部使用小写字母，不包含任何大写字母。即使包名由多个单词组成，也应当将所有单词拼接在一起，而不是使用下划线或中划线。

第二：域反转。包名通常以开发者组织或公司的 Internet 域名反转作为前缀，以保证全局唯一性。比如，如果您的公司网站是 www.example.com，对应的包名前缀将是 com.example。

第三：逻辑划分。包名的剩余部分应当反映软件的逻辑结构或者模块功能。例如，对于一个音频处理库，可能会有 com.example.audio.processing 这样的包名。

第四：避免关键字和保留字。包名中不应含有 Java 的关键字或保留字。

10.3.2 以包为结构组织项目代码

为了更好地组织项目代码，使其更易于理解、维护和扩展，Java 项目通常采用一些标准的组织结构。其中最常见的方法之一是将项目代码划分为不同的包，每个包包含一组相关的类。例如，一个名为 com.liuyandong.myapp 的包可以包含所有与应用程序核心功能相关的类。

以本书的代码为例，当我要讲解两个类之间的继承关系时，我建立一个名为 com.liuyandong.myapp.inheritance 的包，在这个包内，有两个类分别为：ParentClass.java 和 ChildClass.java。这两个类都放在反映它们包结构的文件夹路径下，因此在文件系统的组织结构如下图所示：

其中 src 通常是项目的源代码根目录，main 文件夹下存放程序的入口，随后，按照 Java 的包与文件系统路径的一一对应关系，类文件应放在与其包名对应的目录结构中。

当编译这些文件时，编译器会生成相应的 class 文件，这些 class 文件也会保存在类似的目录结构下，通常是在项目的 build/classes 或 target/classes 目录下（取决于使用的构建工具配置，Maven 或 Gradle 等）。这样组织的好处在于，当 Java 虚拟机（JVM）加载类时，它可以根据类的全限定名找到对应的 class 文件。

值得注意的是，以上所讲的只是约定成俗的规则，并非强制规定，在不同的项目中，也有些许区别，但是理念相通。比如，若是开发 Web 应用程序，则源码很可能放置在 WEB-INF/classes 下而不是 src 目录。如果是 JSP 项目，相关的 class 文件通常会被部署到服务器的某个特定目录下。

10.4 程序员故事

《测试员阿轴》

自从潘闻来到了公司，他哥哥把所有的杂事都推给了他，写代码，部署，客服，当然，还有软件的测试。这让潘闻十分的不满，但是又不能表现出来。

今天有点特别，哥哥潘新来到公司的时候，脸上挂着和蔼的笑容。我就知道，今天会是愉快的一天。他对我和他弟弟潘闻说："今天有个好消息告诉你们。"

"快说吧，别卖关子了。"我回答。

潘新笑着看了看了，又笑着看了看他弟弟，这可是铁树开花的事情，他竟然在对着他弟弟笑。他说："你们还记得上次修改代码，为了兼容不同的设备，把代码搞乱了的事情么？"

潘闻听到这句话，赶紧把头转向自己的计算机屏幕，手中鼠标的滚动速度增加了不少。但是潘新并没有责怪他，反而走过去用手拍了拍他弟弟的肩膀，说："你不要紧张，我们就要有一位测试员了。以后的测试工作，你就不用干了。"

潘闻抬头瞪着他，我也抬头瞪着他，潘新很得意的说："这个人是我亲自选的，她在日本一家公司里做测试，工作特别仔细，特别认真。"

"那她到底是个什么样的人呢？"潘闻问道。

潘新神秘的笑了一下，说："这个很难说，但是一定很符合我们公司的气质。我可不想像有些公司，请一些美女做公司的门面，整天化着浓妆什么事情也不干，还美其名曰什么程序员鼓励师。我们公司可不需要那样的花瓶。"

潘闻脸部表情越来越疑惑，插嘴问道："什么，我们公司不需要么？"

"当然不要！这样的人，每天在公司里走来走去，什么工作也不会做，还鼓励师，我看就是会打扰程序员的工作。还有一件事，公司有你在这里，我怎么会请一位鼓励师过来，你能放过她？"

"那你呢？"潘闻涨红了脸反问。

"好了，好了，现在说你。"潘新嚷道："下午一点钟，她会来公司，到时候大家就知道了。"

正好下午一点钟，一秒不多，一秒也不少，公司敲门进来了一位女士。潘新急忙跑过去，对我们说：“我来给大家介绍一下，这位是何小轴女士。这位叫刘栋梁，这位叫潘闻。”

她身材高挑，胸部臃肿，瓜子脸，披肩长发。我和潘闻对视了一眼，认为潘新完全不懂审美，这完全就是个“程序员鼓励师”啊。

“叫我阿轴吧，我已经工作了五年，都是做测试的工作，很高兴与大家见面，以后可能会挑大家的毛病哦。”

这时，潘闻已经抢先一步说要带阿轴熟悉一下公司的项目，潘新只好答应。

目前，阿轴已经在公司工作了一年多了。测试员和程序员之间发生的故事，阿轴与我们之间每天都在上演着。

与她的名字一样，阿轴真的是特别的轴。她纠结于每一个不完美的地方，每个行为不符合预期的功能，以至于我们都有点怕她了。

她有一个位置绝佳的办公桌，那个办公桌位于玻璃墙的一角。从那个位置，他向外能看到走廊和大厅，向内能看到老板的办公室。她工作细致入微，在短短半年内，就熟悉了公司的软件，还按照她在日本公司里的工作流程，做了详细的笔记和标签。只要有一个地方出现问题，她就能迅速的找到谁写的这段代码。

“潘闻，你过来一下，你的代码有点问题。”

这时候，潘闻生无可恋的从他的座位上站起来，拖着沉重的步伐走到阿轴的座位前。阿轴坐在她的计算机后面，一闪身，随手拉过一把椅子来让潘闻坐下。

“阿轴，有什么问题么？”

“当然有，要不我喊你过来做什么。”阿轴拿出做好的记录，开始一条一条的与潘闻确认。

起初，测试员阿轴刚来公司的时候，还是占据下风，毕竟我们早来一步。但是阿轴是一位勇敢且心细如麻的测试员，她有许多的技术来对软件进行残酷的测试，使你不得不佩服她的创意。每一次她喊我们的名字，总是会使人的血压突然升高。

今天也是如此，时间一分一秒的过去，我看着潘闻的脸越来越红，声音也越来越激动。但是阿轴依旧保持着相同的表情，相同的语速，一条一条的询问着。

最后，潘闻终于出来了，一句话也没说，自己坐在椅子上发呆。吃午饭的时候，阿轴又熟练的玩起了她那套打一巴掌给颗枣的把戏，给潘闻带回来一瓶可乐。潘闻对着可乐直摇头。

不可否认的是，我们的软件，真的是越来越稳定了。

老板都在考虑要不要在公司里推行 TDD（Test-Driven Development，测试驱动型开发）。如果一旦推行了 TDD，虽然不知道会发生什么事情，但是阿轴应该可以主导公司的软件开

发了。当老板问能不能用 TDD 开发模式的时候，我们陷入了沉思。

延申阅读：TDD 开法模式

TDD 开发模式是英文词组 Test-Driven Development 首字母的缩写，翻译为中文叫测试驱动型开发。

在这种方法之前，写代码的流程是先进行详细的设计，然后再进行编码，编码以后再进行测试。

TDD 方式把写代码的流程反了过来，先编写测试代码，然后写代码让测试用例通过。如果出现错误，就再编写代码，直到所有错误消失，使测试成功。最后再改善代码，使其变得整洁。

习惯了传统的编码方式的程序员，可能会觉得编写测试代码是一件又枯燥又麻烦的事情。实际上，使用 TDD 这种先写测试后写代码的方式，目前越来越流行，最大的优点是能得到及时的反馈。

如果用传统方式来写代码，有可能在设计阶段就搞错了，等到了测试阶段才能暴露这个错误（如果运气好的话）。那么期间要经历很长的时间，至少跨越详细设计和编码这两个阶段，做很多的无用功。如果使用 TDD 的方式，先有测试代码，那么在设计和编码阶段肯定会发现错误，如果发现不了，那测试代码无法顺利通过。这个巨大的优点，也使得 TDD 被引入了敏捷开发之中，得到了广泛的应用。

除了编写新的功能，修改软件功能也可以使用 TDD 开发模式。在修改软件功能的时候，先准备好测试代码，就可以随时检验代码是否可以正常运行。

还有一点，因为测试用例会有很多，为了能确保每个测试用例都能通过，我们会有意识的对所编写的代码进行功能上的分拆，使一个功能对应一个测试用例，这样，会使代码更加的模块化。

难点也有，就是测试代码一定要写的通俗易懂，最好是可以用以描述设计文档。这并不容易，需要很多锻炼。

可能大家会有疑问，这个 TDD 开发模式和 Java 有关系么?

实际上，关系并不是很大，但是又有千丝万缕的联系。TDD 开发严重依赖 xUnit 单元测试框架（其中的 x 在不同的语言中代表不同的字符或单词）。

推广 TDD、敏捷开发和 Java 的人很多都是相同的一群人和公司，比如 Rational Software 公司。后来这个公司被 IBM 收购，该公司提出了 UML、还对面向对象编程提出一些开发方法，同时，该公司有大量 Java 语言的先驱程序员。笔者所在的公司就在使用 IBM 的系列产品，从那些产品中，可以看到 Java 与各种新开发方法（敏捷开发、TDD、重构等）有极深的关系。

11 继承

Contents

在本书第 3 章中，我们已初步学习了 Java 面向对象编程中的关键特性——继承。这一机制为构建和组织复杂类层次结构提供了强大的工具，体现了类之间的“泛化”或“由一般到特殊”的内在逻辑联系，恰当地模拟了现实世界中各类事物间的层级体系和相互依存关系。

回顾先前内容，在第 3 章中我们定义了一个基础的球员类（Player），该类作为所有球员共性的抽象，包含了诸如姓名（name）、场上位置（position）和技能等级（skillLevel）等基本属性，以及诸如跑动（run）、射门（shoot）和传球（pass）等通用方法。接着，我们引入了门将类（Goalkeeper），它借助 Java 的继承机制，从球员类（Player）继承了全部的共有属性和方法。在此基础上，门将类又增添了反应速度（reactionSpeed）这一凸显守门员特质的属性，并实现了诸如 save() 这个特定于守门员角色的方法。

接下来，我们将从技术角度深入学习 Java 中是如何实现继承的。

11.1 继承的语法

在 Java 中，继承的语法如下：

```
public class 子类名 extends 父类名 {
    // 子类成员
}
```

其中，public 表示子类的访问权限，class 是关键字声明类，子类名是子类的名称，extends 关键字表示继承，父类名是父类的名称。举一个简单的例子如下：

```
public class Animal {
    public void eat() {
        System.out.println(" 动物吃饭");
    }
}

public class Dog extends Animal {
    public void bark() {
        System.out.println(" 狗叫");
    }
}
```

在这个实例中，Dog 类作为一个具体的子类，继承了 Animal 类的属性和行为。具体来说，Dog 类通过继承 Animal 类，使得 Dog 对象不仅具有自身的特性和方法，同时也能够调用 Animal 类中定义的通用 eat() 方法。这也是继承的用途——允许子类在保留并利用父类已有功能的基础上，进一步增添独有的属性和行为。

在 Java 中，子类继承父类的功能是面向对象编程的一个核心概念。然而，为了使子类能够继承父类的特性，父类必须具有一些可以被继承的特性。如我们在第 9 章中学习的，一个类主要包含属性、方法、构造函数以及静态成员。继承机制就是针对这些内容进行操作的。

这些内容中，属性、方法和构造函数属于类的实例，而静态成员则属于类本身。在继承过程中，这两种类型的成员有着显著的区别，所以我会分开讲解。在接下来的部分，将首先探讨如何继承属于类实例的属性、方法以及构造函数。

11.2 类实例的继承机制

11.2.1 子类可以继承父类的属性么?

在 Java 编程语言中，子类可以继承父类的属性，但这种 “继承” 是有一些特定条件的。能不能被继承，主要看父类属性的访问修饰符是什么，其原则如下：

第一，子类能够继承父类中声明为 public 或 protected 的属性。这意味着，这些属性可以在子类中直接被访问和使用。

第二，对于父类中声明为 default（也称为包私有）的属性，只有当子类和父类位于同一个包时，子类才能继承并访问这些属性。

第三，父类中的 private 属性不能被子类继承。这是因为 private 属性仅能在其所在的类内部被访问，因此子类无法访问或继承这些 private 属性。

接下来，我们将通过下面代码中具体的例子来进一步理解继承的概念。下面是拥有 public、protected 和 private 属性的父类代码：

```
package com.liuyandong.inheritance;

// 父类，包含不同访问权限的属性
public class ParentClass {
    // Public 属性，子类可以直接继承和访问
    public String publicVar = "Public from Parent";

    // Protected 属性，子类也可以直接继承和访问
    protected String protectedVar =
                            "Protected from Parent";

    // Default (package-private) 权限属性，
    // 由于子类和父类在同一包下，子类可以继承和访问
    String defaultVar = "Default
                        (package-private) from Parent";

    // Private 属性，子类不能继承和访问
    private String privateVar = "Private from Parent";
}
```

接下来，是子类的代码：

```
package com.liuyandong.inheritance;

// 子类，和父类位于同一包下
public class ChildClass extends ParentClass {
    public static void main(String[] args) {
```

```
        ChildClass child = new ChildClass();

        // 可以访问并打印继承来的 public 属性
        System.out.println("Public Var: "
                           + child.publicVar);

        // 可以访问并打印继承来的 protected 属性
        System.out.println("Protected Var: "
                           + child.protectedVar);

        // 可以访问并打印继承来的 default 权限属性
        System.out.println("Default Var: "
                           + child.defaultVar);

        // 尝试访问 private 属性会导致编译错误
        // 此处省略尝试访问的代码，因为 Java 不允许这样做
        // 下面这行代码将无法编译通过
        // System.out.println("Private Var: "
                           + child.privateVar);
    }
}
```

在 Java 中，如果在同一个包中，可以不使用 import 语句来引入同包下的类。这是因为 Java 中的包（package）提供了一种命名空间，允许在同一包中的类之间进行直接访问，而不需要显式地导入。当你在同一包中使用类时，编译器会自动将这些类视为可见的，因此不需要使用 import 语句。上面的代码运行之后，输出结果如下：

```
Public Var: Public from Parent
Protected Var: Protected from Parent
Default Var: Default (package-private) from Parent
```

总结起来，子类继承父类属性时，继承的属性会在子类对象中保持其原始值或行为，遵循如下规则：

1. 公共（public）和受保护（protected）属性可以被子类继承和访问。
2. 同包下的默认权限（package-private）属性也可以被同包下的子类继承和访问。
3. 私有（private）属性不会被子类继承。

11.2.2 子类可以继承父类的方法么?

跟前面讲的继承属性差不多，继承方法也遵循类似的访问权限规则。相比于属性继承，方法继承多了一个独特的特点：方法重写（Override）。这意味着子类能够定义一个与父类中的方法具有相同方法名、返回类型和参数列表的方法。这种机制让子类在继承父类方法的同时，实现改变行为的能力。

具体来说，当子类重写一父类的方法时，子类中的新方法将覆盖父类的原有的实现。当通过子类对象调用这个方法时，执行的将是子类提供的实现而非父类原来的实现。这一特性在面向对象编程中极为重要，因为它允许子类在维持继承方法一致性的同时，还可以根据具体场景进行差异化处理。接下来，我们来看下面代码子类对父类方法进行重写的例子。下面的父类拥有 public、protected 方法：

```
package com.liuyandong.overridedemo;

// 定义父类
public class ParentClass {
    // Public 方法，子类可以继承并重写
    public void publicDisplayMessage() {
        System.out.println(" 来自父类中的 public 方法");
    }

    // Protected 方法，子类可以继承并重写
    protected void protectedDisplayMessage() {
        System.out.println(" 来自父类中的 protected 方法");
    }
}
```

接下来，是子类代码：

```
package com.liuyandong.overridedemo;

// 定义子类，继承自 ParentClass
public class ChildClass extends ParentClass {
    // 子类重写了父类的 public 方法
    @Override
    public void publicDisplayMessage() {
        System.out.println(" 子类重写了父类
```

```
                                的 public 方法之后的行为");
    }

    // 子类重写了父类的 protected 方法
    @Override
    protected void protectedDisplayMessage() {
        System.out.println(" 子类重写了父类
                                的 protected 方法之后的行为");
    }

    public static void main(String[] args) {
        // 创建子类对象
        ChildClass child = new ChildClass();

        // 调用重写后的 public 方法
        child.publicDisplayMessage();

        // 调用重写后的 protected 方法
        // 注意: 虽然 protected 方法在子类外部不能直接调用
        // 但在子类内部是可以正常调用的
        child.protectedDisplayMessage();
    }
}
```

在这个例子中，ChildClass 继承了 ParentClass，并分别重写了 public 和 protected 两种访问权限的方法。protected 方法在子类内部是可以重写和访问的，但是在子类外部（例如在其他包中）不能直接访问。不过，当通过子类对象调用时，无论是 public 还是 protected 方法，都会执行子类中重写的方法版本。以上的代码运行之后，输出结果如下：

```
子类重写了父类的public方法之后的行为
子类重写了父类的protected方法之后的行为
```

在上面的例子中，出现了一个关键字 @Override，这个关键字是 Java 5 引入的。它告诉编译器下面的方法是重写父类中的方法。如果一个方法标记为 @Override，但是在父类中没有找到要重写的方法，那么编译器会报错。

在上述代码中，ChildClass 类继承了 ParentClass 类，并且重写了 publicDisplayMessage 和 protectedDisplayMessage 两个方法。这两个方法前面都标记了 @Override 注解，表示这两个方法是重写父类中的方法。

那如果不写 @Override 会怎么样呢？如果不写 @Override，Java 编译器不会对子类中方法进行特殊的处理。也就是说，如果子类的方法签名与父类中的方法签名相匹配，那么方法将会重写父类中的方法，即使没有使用 @Override 注解。

然而，如果子类的方法签名与父类中的任何方法都不匹配，那么该方法只是子类中的一个新方法，而不是重写的方法。如果原本打算重写父类中的方法，但是由于某种原因（例如，方法签名的错误）而没有做到，那么这可能会导致错误的行为。

使用 @Override 注解的主要好处是，它可以让编译器帮助程序员检查子类的代码。如果使用了 @Override 注解，但是并没有重写父类中的任何方法，那么编译器会生成一个错误消息。这可以帮助开发者避免因误解方法签名或误解所继承的类的行为而导致的错误。

综上所述，对公有方法（public）和保护方法（protected）来说，子类会继承父类的所有公有和保护方法。子类可以选择重写（Override）这些方法，也就是定义一个新的方法来替代父类中的方法。重写的方法必须和父类中的方法具有相同的方法名、返回类型和参数列表。

对默认方法（default）来说，如果子类和父类在同一个包中，子类会继承父类的默认方法。子类可以选择重写这些方法。如果子类和父类不在同一个包中，子类则不能继承父类的默认方法。

对私有方法（private）来说，子类不能继承父类的私有方法。即使子类中定义了一个与父类中的私有方法具有相同的方法名、返回类型和参数列表的方法，这个方法也不会重写父类中的方法，而只是子类中的一个新方法。

11.2.3 子类可以继承父类的构造函数么?

在 Java 中，子类不能继承父类的构造函数。每个类都有自己的构造函数，用于创建该类的对象。如果没有为类定义构造函数，Java 编译器会为你提供一个默认的无参数构造函数。然而，子类可以通过使用 super 关键字来调用父类的构造函数。这通常在子类的构造函数中完成，以确保父类的初始化工作被正确地完成。接下来，我们来学习一下子类如何调用父类的构造函数。下面的代码是拥有构造函数的父类：

```
package com.liuyandong.constructor;
public class ParentClass {
    protected String name;

    // 父类构造函数
    public ParentClass(String name) {
        this.name = name;
        System.out.println(" 父类的构造函数被调用，
```

```
                                其参数 name 为: " + name);
    }
}
```

下面是使用 super 关键字来调用父类构造函数的子类:

```
package com.liuyandong.constructor;
public class ChildClass extends ParentClass {
    private int age;

    // 子类构造函数，通过 super 关键字调用父类构造函数
    public ChildClass(String name, int age) {
        // 调用父类构造函数初始化 name 属性
        super(name);
        this.age = age;
        System.out.println(" 子类构造函数被调用，姓名为:
                        " + getName() + " 年龄为: " + age);
    }

    // 获取 name 属性的辅助方法
    public String getName() {
        return this.name;
    }
}
```

然后，在下面的测试类中调用子类的构造函数生成一个对象。

```
package com.liuyandong.constructor;

public class Main {
    public static void main(String[] args) {
        ChildClass child = new ChildClass("liuyandong", 38);
    }
}
```

解释一下以上的代码：这段代码定义了两个类：ParentClass 和 ChildClass。ChildClass 是 ParentClass 的子类。

ParentClass 有一个 protected 的属性 name，并且有一个构造函数，该构造函数接受一

个 name 参数，用于初始化 name 属性。

ChildClass 继承了 ParentClass，并添加了一个新的私有属性 ag。ChildClass 有一个构造函数，该构造函数接受两个参数：name 和 age。在 ChildClass 的构造函数中，首先通过 super(name) 调用父类的构造函数来初始化 name 属性，然后初始化 age 属性。

当你运行这段代码时，会首先调用 ParentClass 的构造函数，然后调用 ChildClass 的构造函数，最后在控制台上打印出一条消息，显示子类对象的 name 和 age 属性的值。运行之后的结果如下：

```
父类的构造函数被调用，其参数name为：liuyandong
子类构造函数被调用，姓名为：liuyandong 年龄为：38
```

还有一个知识点需要讲解，就是子类在调用父类的构造函数时，使用的关键字 super。

11.2.4 如何使用 super 关键字?

在 Java 中，super 关键字主要有两种用途，一种是调用父类的构造函数，另一种是访问父类的属性与方法。

除此之外，super 关键字还有一种较少见的用途，那就是在内部类中引用外部类的成员。如果一个内部类和它的外部类有同名的成员，那么在内部类中可以使用 super 关键字来引用外部类的成员。鉴于目前本书没有讲内部类，暂时不深入探讨这种情况。

使用 super 调用父类的构造函数

在使用 super 调用父类的构造函数时，要遵守以下三点规则：

第一，super 关键字只能在子类中使用，不能在没有继承关系的类中使用。

第二，super 关键字可以用来调用父类的构造函数，但是这种调用必须是子类构造函数的第一条语句。

第三，如果子类的构造函数没有显式地调用父类的构造函数，Java 编译器会自动插入一个调用父类无参数构造函数的 super() 语句。如果父类没有无参数的构造函数，编译器会报错。

前两条规则非常容易，无需多言，第三条规则需要说明一下。根据第 9 章的内容，在 Java 中，如果没有为类定义任何构造函数，编译器会为你提供一个默认的无参数构造函数。然而，如果你为类定义了至少一个构造函数（无论这个构造函数是否带有参数），编译器就不会再提供默认的无参数构造函数。比如，前面的父类中，我们已经实现了一构造函数如下：

```
    // 父类构造函数
    public ParentClass(String name) {
        this.name = name;
        System.out.println(" 父类的构造函数被调用，
                           其参数 name 为: " + name);
    }
```

这样，编译器将不会再自动提供一个默认的无参构造函数。在这种情况下，如果子类的构造函数没有显式地调用 ParentClass 的构造函数，编译器会尝试插入一个调用 ParentClass 无参数构造函数的 super() 语句，但是因为 ParentClass 没有无参数的构造函数，所以会导致编译错误。为了解决这个问题，子类的构造函数必须显式地调用 ParentClass 的构造函数。

使用 super 访问父类的属性与方法

在 Java 中，super 关键字可以用于在子类中访问父类的方法和属性。这在以下两种情况下特别有用：

第一种情况，当子类需要访问父类被子类重写的方法时，如果子类重写了父类的某个方法，那么在子类中直接调用该方法将会执行子类的版本。如果你想调用父类的版本，你可以使用 super 关键字。例如下面的代码：

```
public class ParentClass {
    public void print() {
        System.out.println(" 父类的 print 方法");
    }
}

public class ChildClass extends ParentClass {
    @Override
    public void print() {
        super.print();  // 调用父类的 print 方法
        System.out.println(" 子类的 print 方法");
    }
}
```

上面这段代码定义了两个类：ParentClass 和 ChildClass。ChildClass 是 ParentClass 的子类。ParentClass 有一个公有方法 print，该方法在控制台上打印一条消息。ChildClass 重写了 ParentClass 的 print 方法。在 ChildClass 的 print 方法中，首先通过 super.print() 调用父类的 print 方法，然后在控制台上打印一条新的消息。

当创建一个 ChildClass 的对象并调用其 print 方法时，将会首先打印出父类的消息，然后打印出子类的消息。这是因为 ChildClass 的 print 方法首先调用了 super.print()，然后打印出自己的消息。

通过这种方法，在子类中可以访问父类中被重写的方法。

第二种情况，当子类需要访问父类被子类隐藏的属性时，如果子类定义了一个与父类同名的属性，那么在子类中直接访问该属性将会访问到子类的属性，而不是父类的属性。如果想访问父类的属性，可以使用 super 关键字。

```
public class ParentClass {
    protected String name = " 父类";
}

public class ChildClass extends ParentClass {
    protected String name = " 子类";

    public void printName() {
        System.out.println(name);        // 打印 " 子类"
        System.out.println(super.name); // 打印 " 父类"
    }
}
```

前面这段代码定义了两个类：ParentClass 和 ChildClass。ChildClass 是 ParentClass 的子类。ParentClass 有一个 protected 的属性 name，其值被初始化为” 父类”。ChildClass 继承了 ParentClass，并且定义了一个同名的受保护属性 name，其值被初始化为” 子类”。这意味着 ChildClass 隐藏了父类的 name 属性。

ChildClass 还定义了一个公有的 printName 方法。在这个方法中，首先打印出子类的 name 属性，然后通过 super.name 打印出父类的 name 属性。使用 super.name 允许在子类中访问父类的被隐藏的属性。当创建一个 ChildClass 的对象并调用其 printName 方法时，将会首先打印出” 子类”，然后打印出” 父类”。

11.3 静态成员的继承机制

在 Java 中，静态成员（包括静态方法、静态变量和静态常量）可以被子类继承，但是它们的行为与上面所讲的实例方法、实例变量和构造函数有所不同。先来讲一下理论知识。

静态方法：静态方法可以被子类继承，但是不能被子类重写（Override）。如果子类中定义了一个与父类中的静态方法具有相同的方法名、返回类型和参数列表的方法，这个方

法不会重写父类中的方法，而只是隐藏了父类中的方法。也就是说，即使子类中有一个与父类中同名的静态方法，如果你通过一个父类引用来调用这个方法，实际上调用的还是父类中的方法。

静态变量和静态常量：静态变量和静态常量也可以被子类继承，但是它们在所有的实例中都是共享的，也就是说，所有的实例都访问同一块内存空间。如果子类中定义了一个与父类中的静态变量或静态常量同名的变量或常量，这个变量或常量不会重写父类中的变量或常量，而只是隐藏了父类中的变量或常量。这种情况被称为“隐藏”，而不是“覆盖”。这意味着，如果你通过子类名来访问这个静态成员，实际上访问的是子类中的静态成员；如果你通过父类名来访问这个静态成员，实际上访问的是父类中的静态成员。

讲完理论知识以后，我们再来用代码来验证这些概念。

拥有静态变量和静态方法的父类：

```
package com.liuyandong.staticInheritance;
// 定义父类
public class ParentClass {
    // 静态变量
    public static String staticField = " 父类中的静态变量";

    // 静态方法
    public static void staticMethod() {
        System.out.println(" 父类中的静态方法，该方法输出父类
                                的静态变量： " + staticField);
    }
}
```

子类继承父类并“覆盖”父类的静态变量：

```
package com.liuyandong.staticInheritance;

public class ChildClass extends ParentClass {

    public static void staticMethod() {
        System.out.println(" 子类中的静态方法，
            该方法输出子类的静态变量： " + staticField);
    }
}
```

实例化子类后做测试的代码:

```
package com.liuyandong.staticInheritance;

public class Main {
    // 子类可以直接访问父类的静态成员
    public static void main(String[] args) {
        // 访问和修改静态变量
        System.out.println(" 访问子类继承而来的静
            态变量为: "  + ChildClass.staticField);
        ChildClass.staticField = " 静态变量被子类修改";
        System.out.println(" 静态变量被子类修改以后,
            父类的静态变量为: "  + ParentClass.staticField);

        // 调用静态方法
        ParentClass.staticMethod();
        ChildClass.staticMethod();
    }
}
```

解释一下这段代码的原理。在 main 方法中，首先打印出子类继承自父类的静态变量 staticField 的值。然后，修改这个静态变量的值，并打印出修改后的值。需要注意的是，静态变量是属于类的，而不是属于某个具体的对象，所以当子类修改了静态变量的值，父类中的静态变量值也会被改变。

然后，调用了父类和子类的静态方法 staticMethod。这里需要注意的是，静态方法是不能被子类覆盖（override）的，所以即使子类中有同名的静态方法，调用的还是父类的静态方法。

总的来说，Java 中的静态变量是属于类的，而不是属于某个具体的对象。静态变量在所有的实例中都是共享的，也就是说，如果一个实例修改了静态变量的值，那么其他所有的实例看到的静态变量的值都会被改变。静态方法也是属于类的，而不是属于某个具体的对象，静态方法不能被子类覆盖。

输出的结果如下:

```
访问子类继承而来的静态变量为: 父类中的静态变量
静态变量被子类修改以后, 父类的静态变量为: 静态变量被子类修改
父类中的静态方法, 该方法输出父类的静态变量:  静态变量被子类修改
子类中的静态方法, 该方法输出子类的静态变量:  静态变量被子类修改
```

还有一点值得注意，在 Java 中，静态方法不能被覆盖（override），所以不能在子类中的静态方法上使用 @Override 注解。如果试图这样做，编译器会给出错误。

这是因为静态方法是属于类的，而不是属于某个具体的对象。当调用一个静态方法时，实际上是在调用类的静态方法，而不是在调用某个具体对象的静态方法。因此，静态方法不能被覆盖。

11.4 不可被继承的类和方法

在 Java 中，有些类在设计时就被设定为不可被继承，这通常是因为这些类已经被设计得非常完美，如果允许继承，可能会导致一些不良的结果。这些类通常具有很高的安全性和性能，如果允许继承，可能会破坏这些特性。

这些类是被声明为 final 的，这意味着它们不能被继承。例如，String 类、Integer 类和 Date 类都是 final 的，它们都不能被继承。

在第 9 章的时候，我们已经见过 final 这个关键字了，当时使用 final 这个关键字来创建常量。现在，我们再来学习一下 final 用在类与方法上有什么作用。

11.4.1 final 用于修饰类是阻止类继承

当一个类被声明为 final 时，它意味着该类是最终的，也就是说，它不能有任何子类。这意味着其他类无法继承并扩展这个 final 类的功能。这样做的主要原因是保护类的设计完整性，或者是由于该类的设计目的就是作为一个完整的、独立的实体，无需也不允许进一步的扩展。以 JDK 中 String 的源码为例，其源码如下：

```
public final class String {
}
```

在上述例子中，通过将 String 类声明为 final，Java 禁止了对该类的继承和方法重写，不需要也不期望其他开发人员通过继承来扩展或修改这些方法，从而确保了 API 的稳定性。

总之，使用 final 关键字修饰类是一种面向对象设计策略，它有助于提高代码的可靠性和安全性，特别是在设计那些不应被篡改的核心组件时。例如，Java 标准库中的 String 类就是一个著名的 final 类，确保了所有字符串操作的安全性和一致性。

11.4.2 final 用于修饰方法是阻止方法被覆盖

在 Java 编程语言中，final 关键字在类方法（也就是成员方法）中的应用是为了限制方法的可重写性。当一个方法被声明为 final 时，它具有了不可重写性的特征。如果父类中的一个方法被声明为 final，那么子类就不能覆盖（override）这个方法。这意味着即使子类继

承了父类，也无法提供自己的实现版本来替代父类中的 final 方法。比如，有如下的一个具有 final 方法的父类：

```
package com.liuyandong.finaldemo;

public class ParentClass {
    public final void displayMessage() {
        System.out.println(" 父类中的 final 方法");
    }
}
```

如果下面有子类试图覆盖父类中的 final 方法，会报错。

```
package com.liuyandong.finaldemo;

public class ChildClass extends ParentClass {
    // 试图重写父类的 final 方法会引发编译错误
    public void displayMessage() {
        System.out.println(" 子类中的 final 方法");
    }
}
```

使用 final 方法可以表达设计者的意图，即这个方法的行为应当在整个继承体系中保持固定，不希望也不允许子类对其进行更改。

从性能上来说，现代的 JVM 在方法调用时对虚方法表进行了优化，但在某些情况下，编译器知道一个方法不会被重写时，可以进行一些优化。对于 final 方法，编译器和 JVM 可以确保直接调用该方法，而不必经过动态绑定查找过程，这在理论上能带来一定的性能提升，尤其是在频繁调用方法的情况下。

还有一个问题，final 方法是属于子类还是属于父类呢？答案是 final 方法仍然属于父类的实例，因为 final 方法的定义在父类中，并且子类不能更改父类的定义。

接下来，用子类调用父类中 final 方法的示例：

```
class Animal {
    public final void speak() {
        System.out.println(" 动物在叫");
    }
}
```

```
class Dog extends Animal {

    public void eat() {
        System.out.println(" 狗在吃");
    }
}

public class FinalMethod {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.speak(); // 输出 " 动物在叫"

        Dog dog = new Dog();
        dog.speak();
        dog.eat(); // 输出 " 狗在吃"
    }
}
```

在这个例子中，Animal 类中的 speak() 方法是 final 方法，因此 Dog 类无法覆盖它。当调用 animal.speak() 或 dog.speak() 时，都会调用 Animal 类中的 speak() 方法，因为它是最终的实现。这一点跟调用 eat() 有所不同。

由此，我们可以证明，在 Java 中，如果父类中有 final 方法，子类就不可以覆盖这个方法。这个方法仍然属于父类的实例，不属于子类的实例。

总之，final 方法在类层级的设计中扮演着稳定性的角色，它有助于确保特定方法的实现不被子类随意改动，维持代码的一致性和可靠性。在实际项目中，尤其是在大型框架和 API 设计时，这种方法级别的不可变性常常作为设计原则的一部分来增强程序的健壮性。

11.5 程序员故事

《软件园冷暖》

从地铁站到软件园的这条路，是我每天上下班都要走的路。初冬季节，天亮的晚了，我比平日要晚去一点。

有人地面推广工作人员沿街让人扫二维码下载软件，穿着一件卡通唐老鸭的衣服，头大大的，胸前挂着二维码。因为全身大部分都包裹在那件衣服下面，看不出性别，更看不出年龄。唯一没有包裹住的地方就是一双鞋子，从鞋子的粉红颜色上来看，大概也许是个

年纪轻轻的姑娘。

她左手斜挎着一个印有公司 logo 的篮子，篮子里有一些类似钥匙扣、手环等一些小挂件。右手拿着一个捏一下，就会发出唐老鸭叫声的橡皮玩具。走一步，捏一下，响一声，像“秒针”一样记录着她的时间，仿佛那就是她生命的钟摆。行色匆匆的人们，从她身边闪过，甚至连头都没有转一下。在软件园这个寸土寸金的地段，我的脑海中闪现“一寸光阴一寸金，寸金难买寸光阴”的古老诗句。时间和土地，都标上了它的价格，有寸土寸金的地段，也有大片的荒芜。不要再说“寸金难买寸光阴”了，多少人为了一口饭吃想把自己的一生的光阴便宜的卖掉，还没人买呢。

唐老鸭对路过的行人点头鞠躬，希望能扫她胸前的二维码。在她鞠躬的那一刻，卡通头和卡通上衣之间的缝隙露了出来，一束黑色的长发铺在蓝色的外套上。看着她，也不知是黑色的头发，还是蓝色的忧郁，让我觉得很震动。仿佛一切成了慢动作，她鞠躬，她抬头，她又鞠躬，她又抬头，当然，并没有人停下脚步，她也目中无人似的，继续下一次鞠躬。再一想，总是这样跟着她后面看，万一她要我扫二维码了，这才三脚两步的走开了。

再走过去是一家店面，卖的是早餐，别家早餐店门口的音箱都是放着自家店的卖的吃食，“稀饭、包子、豆浆、油条、牛杂汤、牛肉面”，这家店的老板也许有些故事，他一年四季放着同一首英文歌曲《The Sound of Silence》，仿佛这高速的大城市，这缤纷的广告牌，这嘟嘟响的汽车喇叭，这吸溜吸溜吃牛肉面的食客，都无法耽误他听这首歌。我也喜欢听这首歌，耳朵如鱼得水，尤其是听到“The words of the prophets are written on the subway walls And tenement halls”心中总是泛起一些涟漪。听着这句音乐，转角就碰到了一堵铁栏杆做的墙，栏杆上还有微枯的藤曼叶，枯叶虚掩着巨大的一排字，是一个小学。

小学门口变得拥挤起来，家长开着私家车把小孩送到学校，车尚未停稳，车门便打开了，小孩从车上下来，随手把车门重重的关上，车便开走了。早餐店的英文歌还在唱着，可是歌词已经听不清了。小心躲避着急于奔向学校的学生，留意着店家橱窗又变了什么花样，店家的橱窗，和我们做的软件一样，不管代码写的如何的混乱，也要把最好的一面展示给用户，勾引起他们的购买欲。只有购买欲过度膨胀以后，为了买不必要的东西，便会想赚非分的钱，不惜说弥天的谎，甚至为非作歹。

快到软件园门口的时候，我掏出工牌，套在脖子上。有一次忘记戴工牌，被门口的保安拦在外面，工位就在几十米远的地方，却咫尺天涯，可望而不可及。我激动的和保安吵了两句，没想到惹怒了这位保安大人，他可能觉得只身一人和我争吵太过平淡，用手里的对讲机喊来了十来名保安来绷紧场面，目光四射，制服笔挺，非要吵到我司的领导亲自来领人才算结束。事后，还在门口贴了一张告示，凡不带工牌者，一律不得入内。自此，这个保安每次看到我，都要斜着眼看我的脖子上有没有套上工牌。最初，他并不斜着眼看我，但是他经不起我和他对视，只要他看我，我就微笑的从头到脚，从脚到头的打量他几次，这让他恼火，自此，他只斜着眼看我了。

进了软件园，一切所见的就都是我的日常工作了。在软件园中，无非就是三种人：老

板、项目经理和程序员。当然也有美术人员，测试人员，这些干活的，都可以归为程序员这一类。要区分这几种人一点也不能。前几天，我就看到两个人谈话，没人抬举程序员，一旦受抬举，连程序员本人都会窘迫。老板翘着二郎腿指手画脚地在说，另一个人则坐在对面，只有猴子没进化成人之前长尾巴的地方坐在椅子上，双手放在膝盖上端端正正的坐着。像刘姥姥进大观园，张开嘴笑着，笑得舌头都发了凉。

程序员之间的谈话，则要激烈的多，也真诚的多。有 bug 就有 bug，并没有程序员会可以隐瞒自己的 bug，这能有什么关系呢？打个补丁就可以了，这是中国，中国的天都是女娲打了补丁补过的呢。

12 抽象类

Contents

现实世界中充满了抽象概念，例如图形和交通工具。这些概念无法直接实例化，因为它们缺乏具体的形状、大小或其他特征。以图形为例，它是一个没有特定形状或大小的抽象概念，无法实例化为具体对象。然而，圆形、矩形和三角形等具体图形可以被实例化。

为了在编程中处理这种抽象概念，Java 引入了抽象类。抽象类是一种不能直接创建对象的类，它主要用于描述一组具有共同特征或行为但又不完全相同的类。通过使用 abstract 关键字声明抽象类，并可以在其中包含抽象方法。抽象方法是没有具体实现的方法，需要由子类提供具体的实现。

在本章中，我们将深入学习抽象类的概念、语法和用法。

12.1 抽象类概述

在 Java 中，抽象类是一种特殊的类，它不能被实例化。抽象类的主要目的是为子类定义通用行为和状态，并提供一个通用接口。

与普通类一样，抽象类也包含属性和方法。

属性：抽象类可以包含属性，这些属性可以在抽象类的方法和子类的方法中访问，也可以在子类中修改，以改变抽象类方法的行为。需要注意的是，Java 中没有“抽象属性”的概念，因为属性不能像方法那样被覆盖。当在子类中定义一个与父类同名的属性时，实际上是在子类中创建了一个新的属性，而不是覆盖父类的属性。

方法：抽象类的方法可以分为两种类型：

非抽象方法：与普通类中的方法相同，非抽象方法具有声明和实现，可以提供一些默认行为，子类可以直接使用或覆盖这些行为。

抽象方法：抽象方法只有声明，没有实现。它们使用 abstract 关键字标记，并以分号结束，而不是用花括号。抽象方法的实现由抽象类的子类提供。

抽象类通常用作基类，定义一些通用行为和状态，然后由子类实现或扩展这些行为和状态。这可以提高代码的复用性和灵活性，并利用多态性来实现更灵活的设计。

对抽象类来说，需要注意以下两点：

1. 如果一个类包含一个或多个抽象方法，那么它必须被声明为抽象类。
2. 抽象类可以不包含任何抽象方法，但这种情况并不常见。

关于第 2 点，我详细解释一下为什么抽象类可以不包含任何抽象方法，以及这种情况为什么不常见。

抽象类的主要目的是定义子类必须实现的行为。当一个类被声明为抽象类时，它就无法被实例化，这意味着它只能用作其他类的基类。通常，抽象类会包含一个或多个抽象方法，这些方法没有实现，需要由子类提供具体的实现。然而，在某些情况下，可能需要定义一个不能被实例化的类，但它又没有需要子类实现的抽象方法。例如：

工具类：有时，可能需要创建一个包含一些静态方法的类，这些方法可以被其他类使用，但不希望这个类被实例化。在这种情况下，可以将这个类声明为抽象类。

标记接口：标记接口是一种没有任何方法或属性的接口。它们用于标记类，以便其他代码可以识别它们。抽象类可以用于实现标记接口，因为抽象类也不能被实例化。

虽然抽象类可以不包含任何抽象方法，但这种情况并不常见，因为这违背了抽象类的主要目的。通常，如果您需要定义一个不能被实例化的类，最好使用接口或枚举。

接下来，我们将通过下面的一个示例来演示如何实现抽象类。

```
package com.liuyandong.abstractclass;
// 定义一个抽象图形类 Shape
public abstract class Shape {

    // 属性：颜色
    protected String color;

    // 构造器，初始化颜色
    public Shape(String color) {
        this.color = color;
    }
```

```
    // 非抽象方法：获取图形颜色
    public String getColor() {
        return color;
    }

    // 设置图形颜色
    public void setColor(String color) {
        this.color = color;
    }

    // 抽象方法：计算图形面积
    public abstract double getArea();

}
```

这段代码定义了一个名为 Shape 的抽象类，代表了一种抽象的图形。

Shape 类有一个 protected 属性 color，表示图形的颜色。protected 属性可以被同一个包中的其他类以及所有的子类访问。

Shape 类有一个构造器，接受一个 String 类型的参数 color，用于初始化图形的颜色。

Shape 类有两个非抽象方法：getColor 和 setColor。getColor 方法用于获取图形的颜色，setColor 方法用于设置图形的颜色。

Shape 类有一个抽象方法 getArea，用于计算图形的面积。这个方法没有实现，需要 Shape 类的子类来提供实现。

因为 Shape 类是抽象的，所以不能创建一个 Shape 对象。你需要定义一个 Shape 的子类，如 Circle 或 Rectangle，并在子类中提供 getArea 方法的实现，然后才可以创建一个子类的对象。

接下来，我们分别实现 Circle 和 Rectangle 这两个类。

```
package com.liuyandong.abstractclass;

// 继承 Shape 抽象类的子类：Circle（圆形）
public class Circle extends Shape {

    private double radius; // 圆形的半径属性
```

```
    public Circle(double radius, String color) {
        super(color);
        this.radius = radius;
    }

    // 实现抽象方法: 计算圆形面积
    @Override
    public double getArea() {
        return Math.PI * Math.pow(radius, 2);
    }
}
```

这段代码定义了一个名为 Circle 的类，它继承自 Shape 抽象类，代表了圆形。

Circle 类有一个私有属性 radius，表示圆的半径。私有属性只能被类自己的方法访问，不能被类的子类或其他类访问。

Circle 类有一个构造函数，接受一个 double 类型的参数 radius 和一个 String 类型的参数 color。radius 用于初始化圆的半径，color 通过 super(color) 传递给父类 Shape 的构造器，用于初始化图形的颜色。

Circle 类覆盖了父类 Shape 的抽象方法 getArea，提供了计算圆面积的实现。面积的计算公式是 π 乘以半径的平方，这个公式在 return Math.PI * Math.pow(radius, 2); 这行代码中实现。

因为 Circle 类提供了父类 Shape 的所有抽象方法的实现，所以 Circle 类不是抽象的，你可以创建一个 Circle 对象。例如，你可以这样创建一个半径为 5，颜色为”red” 的圆：

```
Circle circle = new Circle(5, "red");
```

接下来，继续实现 Rectangle 类:

```
package com.liuyandong.abstractclass;

// 继承 Shape 抽象类的另一个子类: Rectangle（矩形）
public class Rectangle extends Shape {

    private double width;
    private double height;
```

```
    public Rectangle(double width, double height,
                                    String color) {
        super(color);
        this.width = width;
        this.height = height;
    }

    // 实现抽象方法：计算矩形面积
    @Override
    public double getArea() {
        return width * height;
    }
}
```

这段代码定义了一个名为 Rectangle 的类，它继承自 Shape 抽象类，代表了一个矩形。

Rectangle 类有两个私有属性 width 和 height，分别表示矩形的宽度和高度。私有属性只能被类自己的方法访问，不能被类的子类或其他类访问。

Rectangle 类有一个构造函数，接受两个 double 类型的参数 width 和 height，以及一个 String 类型的参数 color。width 和 height 用于初始化矩形的宽度和高度，color 通过 super(color) 传递给父类 Shape 的构造器，用于初始化图形的颜色。

Rectangle 类覆盖了父类 Shape 的抽象方法 getArea，提供了计算矩形面积的实现。面积的计算公式是宽度乘以高度，这个公式在 return width * height; 这行代码中实现。

因为 Rectangle 类提供了父类 Shape 的所有抽象方法的实现，所以 Rectangle 类不是抽象的，可以创建一个 Rectangle 对象。例如，你可以这样创建一个宽度为 4，高度为 5，颜色为”blue” 的矩形：

```
Rectangle rectangle = new Rectangle(4, 5, "blue");
```

接下来，写一段代码来测试一下以上两个类:

```
package com.liuyandong.abstractclass;

public class Main {
    public static void main(String[] args) {
        // 创建一个矩形对象
        Rectangle rectangle = new Rectangle(2, 3, " 白色");
```

```
        // 调用子类的 getArea 方法
        System.out.println(" 矩形的面积为: " +
                                rectangle.getArea());
        // 调用父类的 getColor 方法
        System.out.println(" 矩形的颜色为: " +
                                rectangle.getColor());

        // 创建一个圆形对象
        Circle circle = new Circle(2, " 红色");
        // 调用子类的 getArea 方法
        System.out.println(" 圆形的面积为: " +
                                circle.getArea());
        // 调用父类的 getColor 方法
        System.out.println(" 圆形的颜色为: " +
                                circle.getColor());
    }
}
```

以上代码运行之后，结果如下：

```
矩形的面积为：6.0
矩形的颜色为：白色
圆形的面积为：12.566370614359172
圆形的颜色为：红色
```

12.2 程序员故事

《潘新“试车”》

离城市越来越远，路边开始没有护栏了。这条公路穿过城市的喧嚣，穿过拥挤的街道，穿过一片片树林，沿着河岸蜿蜒展开。我的车轮可以轻易的从柏油路滚到路边的草地上，我跟老板潘新说：“时间还早，要不要先下车看看风景。”

潘新简短的回答：“好。”

我把车子停在路边的野草上，脚下的绿草沿着河岸无际的展开。河里有几只不知名的鸟在游泳，岸边有放羊的牧人赶着吃草的羊，一切都像是我小时候在农村的样子，除了我能看到那个牧人在玩手机，还有一些远足的孩子在用四轴飞行器航拍河边的风景。

我靠在车旁，阵阵凉风吹过我的脸庞。我多么希望以后会住在这样的地方，品尝什么

叫恬静，体会什么叫和谐。

仿佛在一夜之间长大了一样，今天我要和潘新去一家高科技公司，做自动驾驶的。这家公司的试车场建在远离城市的地方，今天我们要去谈一些合作的项目。

几年前，我们还在为买车到底是买手动挡还是自动挡争论不休，现在却已经在讨论以后买车要不要自动驾驶了。这个世界上，没有什么永远不变的事情，不止汽车，手机这些高科技领域发生了巨大的变化，畜牧业又何尝没有惊人的进步？老旧的养猪方法已经被科学化的养猪方法逐步取代，老一辈养猪人赖以维生的技术已经被新一代的科技巨头所超越，听说，有些高科技公司已经实现了猪脸识别，进步和变迁正紧锣密鼓的渗透到每一个领域。

谁不进步就要被淘汰，在这样的恐惧下，我们公司也开始布局下一个产业，以防自己像恐龙一样被淘汰。这次要来考察的正是下一个热门的行业：自动驾驶。

我轻轻的叹了一口气，和潘新钻回车里。说实在的，行业的剧烈变化让我们都感到沮丧，只有当我看到小草没变，山川没变时，心里才稍微好受一点。这时候，我总在心里默念陶渊明的诗：“天地长不没，山川无改时。草木得常理，霜露荣悴之。”

车子到了试车场地，接待我们的是一个项目负责人，他简要的给我们介绍了自动驾驶的项目。

“这么多摄像头？”潘新围着实验车转了好几圈，又不停的将脑袋伸到车里观望，一会儿摸摸这个，一会儿动动那个。“车子倒是不错，到底能不能开的比人好啊？”

“潘先生，我们现在的技术，在大部分时候都比人开的要好要安全，虽然不能说是 100%，95% 的情况下都比人开的要好。”这个项目负责人夸下海口说道：“我们这个车已经开了 30 万公里，已经达到了 L4 级别，也就是说你上车以后，基本上可以睡觉了。”他讲完这句话以后，脸上浮现了项目经理特有的自信表情。

“这是试车场吧？”潘新问道。

“是的，这个场地全是我们的，跑下一圈下来有 10 公里，里面模拟了各种路况。”项目经理回答说。

“你不介意我坐上跑一圈吧？”潘新又问道。

“当然不介意。”项目经理笑着说，“你只要坐在这里，就可以了，开得非常稳。”潘新坐进了副驾驶的时候，项目经理也自信的坐在了后排。我楞住了，我在思考是坐在驾驶位，还是不坐了，毕竟从来没有坐过没有司机的车。

这时候，我的老板扭头对我喊到：“你赶紧上来，坐后面。”我只好无奈的坐在了后排。车子发动以后，我死死的盯着方向盘的位置，感到非常的诡异，像有一个透明人在开车一样，方向盘自动转来转去。幸好车子速度很慢，只有不到 50 公里/小时。

他们讲了一路的话，我一句也没听进去，就死死盯着方向盘，谢天谢地，终于开完了一圈。

下车以后，项目经理说：“潘先生，我们这个自动驾驶可以模仿车主的驾驶技术，如果你在开的话，自动驾驶就会进入学习模式，你开的越多，自动驾驶就会有你的驾驶风格。”

潘新的眼睛里闪出了一丝光芒，兴奋的问道：“这是真的么？”项目经理回答道：“当然是真的”。这句话还没有传到潘新的耳朵里，他已经坐在了主驾驶的位置上，迫不及待的让我们上车，他要教自动驾驶开车。

我刚坐好，赶紧拉上安全带。伴随项目经理关车门的声音，车子已经咆哮着冲上车道，我听见项目经理一声惨叫，他的头刚好碰在后排头枕上。

车子在赛道上左右旋转，车轮在赛道上摩擦出呜呜的声音，还间歇的抖了几抖。在一条直道上，我右手死死的拉着把手，项目经理在慌乱中竟然握紧了我的左手，把我抓的生疼。我看到潘新兴奋的耸着肩，引擎在他的地板油下吼叫。

可惜赛道太短了，一会儿工夫就要跑完了。潘新说：“我们来试试刹车吧。”他刚说完，就不留余地把刹车踩到底，那部车发出嘶鸣，我觉得车子像螃蟹一样横着走了十几米才终于停下。车窗外的风景才慢慢的显露出本来的面目。

项目经理的脸还是白的，潘新转过头，用手疑虑的托着下巴，眼神中充满疑惑地问道：“怎么样？我这样开车，自动驾驶软件能学会么？”

“你以前不会是开赛车的吧？”项目经理惊魂未定的问道。

“我以前是开卡丁车的，可惜没坚持来下。”潘新回答道，“如果我这样开，自动驾驶还会学习我的风格么？”

项目经理沉默不语，若有所思。

“要不，咱们再开几圈试试？”潘新问道。

这时候，项目经理突然活了过来，眼珠胀大了一圈，急忙回答说：“我一会还有点儿事，这样吧，车你可以随便开，我得先走了，让自动驾驶先学习着，我们还真没有请个专业赛车手来做测试呢……两位先看着，我先走了，一会儿让同事过来，咱们再来研究数据。”

在回去的路上，还是我在开车。“你觉得自动驾驶会取代我们么？”潘新突然问我。我回答说：“会取代我，但是不会取代你。你真的开过卡丁车？”

“开过几年，人总有实现不了的梦想，你说是么？”

一路上，他没有再讲话，我也没有。

13 接口

Contents

当项目变的庞大，不可避免的需要处理复杂的问题和系统。为了管理这种复杂性，需要一种方式来组织和构造的代码，使其易于理解、修改和扩展。Java 接口（interface）是实现这一目标的强大工具之一。

在 Java 中，讲到接口，往往有两种不同的含义。在概念性描述时，使用这个术语来描述类可以提供给其他代码的功能，比如某个类提供了什么公共接口。在本章中，我们将介绍该术语的更具体用法。在本章，接口的定义是一组协议，类可以实现（implement）接口以遵循这种协议。接口规定了类应该做什么，而不是怎么做。这种分离使得我们的代码更加灵活，因为我们可以改变类的内部实现，而不影响到使用这个类的代码。

在接下来的内容中，我们将深入探讨 Java 接口的各种特性和用法，以及如何利用接口来编写更优雅、更强大的代码。

13.1 接口小史

在编程领域中，Java 以其卓越的稳定性闻名遐迩，尤其体现在诸如封装与继承等核心特性上。相比于封装和继承，Java 接口经历了较显著的演进，但其始终坚守向下兼容的原

则，确保了现有代码库的平稳运作。对接口来说，Java 8 跟 Java 9 这两个版本连续对其进行了加强，所以非常有必要了解接口的历史，尤其是这两个版本中对 Java 接口的改进史。

接口的历史可以追溯到 Java 的最早版本。在最初的 Java 版本中，接口被引入作为一种定义行为的方式，而不包含任何实现。这使得 Java 可以实现类似于多重继承的功能，因为一个类可以实现多个接口。

在 Java 8 中，接口发生了重大的变化。Java 8 引入了默认方法和静态方法。默认方法允许接口包含方法的实现，这意味着接口可以提供方法的默认行为。静态方法则允许在接口上定义工具方法，而不需要创建一个工具类。

在 Java 9 中，接口又增加了一个新特性：私有方法。私有方法允许在接口中定义只能被接口的其他方法调用的方法，这使得接口的默认方法和静态方法可以共享代码。

在 Java 8 与 Java 9 对接口进行了大量改进之后，会发现接口跟上一章讲的抽象类太类似了。虽然还是有不少差别，但是相似点越来越多，功能越来越复杂，学习的难度也越来越高。这意味着开发者需要花更多的时间来理解和使用这些新特性。这是一个权衡的问题，不同的开发者可能会有不同的观点。如果觉得这些新特性增加了过多的复杂性，可以选择不使用它们，或者只在需要的时候使用它们。

以上就是 Java 接口的主要历史和发展。从最初的定义行为的工具，到现在可以包含实现、支持函数式编程的强大工具，Java 接口的发展反映了 Java 语言的发展和演变。

13.2 接口的定义与实现

在 Java 中，接口（Interface）用来定义一组方法签名，它们不提供实现，而是声明一组相关的行为。接口允许类通过实现接口来承诺将会提供接口中定义的方法的具体实现。下面的代码是一个简单的例子，我家里有一只猫一只狗，通过定义一个名为 Animal 的接口来展示如何定义行为：

```
package com.liuyandong.interfacedemo;

public interface Animal {
    // 声明一个所有动物都应具备的行为：发出声音
    void makeSound();
}
```

在 Java 中，接口中的方法默认是抽象的。这意味着这些方法只有声明，没有实现。在这个例子中，makeSound 方法是抽象的，它没有方法体，只有方法签名。然后，用代码清单 13-2 中的 Dog 类来实现 Animal 的接口：

```
package com.liuyandong.interfacedemo;

public class Dog implements Animal{
    // Dog 类实现了 Animal 接口，所以必须实现 makeSound 方法
    @Override
    public void makeSound() {
        System.out.println(" 汪汪汪");
    }
}
```

在 Java 中，当实现接口的类中重写接口的方法时，使用 @Override 注解是一个好的实践，但并不是必须的。在这里，@Override 注解有两个主要的作用：

一是提高代码的可读性：该注解表明该方法是重写了接口的方法，这对于阅读和理解代码很有帮助。二是编译器检查：如果使用了 @Override 注解，但实际上并没有重写接口的方法，编译器会报错。这可以帮助你发现可能的错误。所以，虽然使用 @Override 注解不是必须的，但是它可以帮助编写更清晰、更准确的代码。

接下来，再定义一个 Cat 类来实现 Animal 的接口：

```
package com.liuyandong.interfacedemo;

public class Cat implements Animal{
    // Cat 类实现了 Animal 接口，所以必须实现 makeSound 方法
    @Override
    public void makeSound() {
        System.out.println(" 喵喵喵");
    }
}
```

最后，看看有没有实现 makeSound() 方法，测试代码如下：

```
package com.liuyandong.interfacedemo;

public class Main {
    public static void main(String[] args) {
        Animal cat = new Cat();
        Animal dog = new Dog();
        cat.makeSound();
```

```
            dog.makeSound();
        }
    }
```

上面这些代码展示了 Java 中接口的使用，接下来，我来详细讲一下这些代码的用途。

首先，定义了一个名为 Animal 的接口，这个接口定义了一个方法 makeSound()。

然后，我们定义了两个类 Dog 和 Cat，它们都实现了 Animal 接口。这意味着它们都提供了 makeSound() 方法的实现。Dog 类的 makeSound() 方法输出 “汪汪汪”，而 Cat 类的 makeSound() 方法输出 “喵喵喵”。

在测试的 Main 类的 main 方法中，创建了一个 Dog 对象和一个 Cat 对象，但是我们将它们都视为 Animal 类型。这就是多态（有关多态的内容，在本书后面再讲）的一个例子：一个对象可以被视为它的自身类型、它的父类（基类或者超类）或它实现的接口。

然后，我们在这两个 Animal 对象上调用 makeSound() 方法。由于多态的存在，实际调用的方法取决于对象的实际类型。所以，myDog.makeSound() 调用的是 Dog 类的 makeSound() 方法，输出 “汪汪汪”，而 myCat.makeSound() 调用的是 Cat 类的 makeSound() 方法，输出 “喵喵喵”。

以上的内容就是接口最简单也是最广泛的应用，随后，Java 8 跟 Java 9 的出现，让接口变的功能强大的同时，也开始变的复杂。

13.3 接口与抽象类

13.3.1 接口的作用

在 Java 中，接口（Interface）是一种引用类型，在 Java 8 之前，它只是抽象方法的集合。虽然 Java 8 之后，接口既可以有属性，方法也不再限定为抽象了，但是其用途一直以来还保持不变。接口的主要用途有如下三种：

第一，定义行为规范：接口定义了一种行为的规范，实现接口的类需要遵循这种规范。接口中的所有方法都是抽象的，也就是说，它们只有声明没有实现。实现接口的类需要提供这些方法的具体实现。

第二，实现多重继承：Java 不支持多重继承，但是一个类可以实现多个接口，通过这种方式，Java 可以实现多重继承的效果。

第三，提高安全性：通过接口，我们可以隐藏类的实现细节，只向外界暴露我们想要暴露的方法。

看到这里的时候，你肯定会有这样一个疑问，接口跟抽象类有什么区别？实际上，两

者有非常多的相似点，比如两者都不能被实例化，只能被子类实现或继承；两者都可以包含抽象方法，即没有方法体的方法，需要在子类中实现；还有两者都用于实现多态性和代码复用。但是不可否认的是，目前两者仍然有一些差别。接下来，我们讲一下两者的差别。

13.3.2 接口与抽象类的区别

13.3.2.1 接口类的属性与抽象类中的属性不同

从 Java 8 开始，Java 中的接口可以拥有属性。但是接口中的属性与抽象类中的属性有以下区别：

第一，接口中的属性隐式为 static final，而抽象类中的属性跟普通类中是一样的。

第二，接口中的属性不能被子类覆盖，而抽象类中的属性可以被子类覆盖。

第三，接口中的属性只能通过接口的实现类进行访问，而抽象类中的属性可以直接通过子类进行访问。举一个例子来演示一下接口与抽象类中属性的区别：

```
package com.liuyandong.abstractclassvsinterface;

interface Animal {

    // 接口中的属性默认是 public static final
    String name = "Animal";

}

class AbstractAnimal implements Animal {
    String name = "Abstract Animal";
}

class Dog extends AbstractAnimal {
    String name = "Dog";
}

class Cat implements Animal {
    String name = "Cat";
}
public class Main {
    public static void main(String[] args) {
```

```
        // 输出接口的属性
        // 由于接口中的属性是 public static final,
        // 所以可以通过类名直接访问
        System.out.println(Animal.name);

        // 输出抽象类的属性
        // 由于抽象类中的属性是实例属性，所以不能通过类名直接访问
        // 但是可以通过实例访问
        // System.out.println(AbstractAnimal.name)
        AbstractAnimal abstractAnimal = new Dog();
        System.out.println(abstractAnimal.name);

        // 输出实现类的属性
        Dog dog = new Dog();
        Cat cat = new Cat();
        System.out.println(dog.name);
        System.out.println(cat.name);
    }
}
```

这段代码主要展示了 Java 中接口和抽象类中属性的不同特征。

首先，定义了一个接口 Animal，其中有一个默认为 public static final 的属性 name。在 Java 中，接口中的属性默认是 public static final 的，这是 Java 语言规定的。也就是说，这些修饰符是不能被修改的常量，一旦赋值就不能再改变。由于它们是静态的，所以可以直接通过接口名来访问，而不需要创建接口的实例。如果你需要一个可以修改的属性，或者一个非静态的属性，你可能需要使用类或者抽象类，而不是接口。

然后，定义了一个实现 Animal 接口的抽象类 AbstractAnimal，它也有一个属性 name。那抽象类 AbstractAnimal 中的 name 与 Animal 接口中的 name 有什么关系么？除了名字相同，没有任何关系。Animal 接口中的 name 属性是一个 public static final 的常量，而 AbstractAnimal 类中的 name 属性是一个实例属性，需要通过 AbstractAnimal 类的实例来访问。

当 AbstractAnimal 类实现 Animal 接口时，它并没有继承接口中的属性，而是需要实现接口中的方法（在这个例子中，为了讲解方便，Animal 接口没有定义任何方法，所以 AbstractAnimal 类没有需要实现的方法）。所以，AbstractAnimal 类中的 name 属性和 Animal 接口中的 name 属性是两个完全独立的属性。

接着，定义了一个继承自 AbstractAnimal 的类 Dog，它还有一个实例属性 name。在这

段代码中，Dog 类继承了 AbstractAnimal 类，并且两者都定义了名为 name 的属性。在 Java 中，这种情况被称为“字段隐藏”。

Dog 类中的 name 属性并没有覆盖 AbstractAnimal 类中的 name 属性，而是隐藏了它。这意味着，如果你有一个 Dog 对象并访问其 name 属性，你将看到的是 Dog 类中定义的 name，而不是 AbstractAnimal 类中的 name。如果你想访问 AbstractAnimal 类中的 name 属性，你需要将 Dog 对象向上转型为 AbstractAnimal 对象，然后访问 name 属性。例如：

```
Dog dog = new Dog();
// 输出 "Dog"
System.out.println(dog.name);
// 输出 "Abstract Animal"
System.out.println(((AbstractAnimal)dog).name);
```

之后，定义了一个实现 Animal 接口的类 Cat，它也有一个实例属性 name。

在 Main 类的 main 方法中，首先通过接口名直接访问了接口的属性 name，因为接口的属性默认是 public static final 的。

然后，通过 AbstractAnimal 的实例访问了抽象类的属性 name，因为抽象类的属性是实例属性，不能通过类名直接访问。

最后，通过 Dog 和 Cat 的实例访问了这两个类的属性 name。

以上代码运行之后的结果如下：

```
Animal
Abstract Animal
Dog
Cat
```

13.3.2.2 接口中的方法与抽象类中的方法也有些区别

在 Java 中，接口和抽象类中的方法主要有以下两点主要的区别：

默认实现：接口中的方法默认都是抽象的，不能有实现（除非是 Java 8 引入的默认方法和静态方法）。而抽象类中的方法可以是抽象的，也可以是非抽象的，即可以有具体的实现。

访问修饰符：起初接口中的方法默认都是 public 的，不能使用其他访问修饰符。从 Java 9 开始，接口可以有私有方法。而抽象类中的方法可以使用任何访问修饰符。接下来，我们用代码展示一下之间的区别，首先，定义一个有抽象方法，私有方法以及默认实现的方法的 Feedable 接口：

```
package com.liuyandong.methoddifference;

public interface Feedable {
    // 抽象方法，实现类必须提供实现
    void eat();

    // Java 8 引入的默认方法，接口内已经有实现
    default void defaultFeed() {
        System.out.println(" 接口中定义的默认方法.");
    }

    // Java 9 及以后版本，接口可以有私有方法
    private void privateHelperMethod() {
        // 私有方法仅能在接口内部使用
    }
}
```

上面这个接口中，提供了三种类型的方法：

第一种是 void eat()，这是一个抽象方法，这意味着任何实现了 Feedable 接口的类都必须提供该方法的具体实现。

第二种是 default void defaultFeed() {…}，这是 Java 8 引入的默认方法，它在接口中提供了方法体的实现。因此，实现类可以选择是否覆盖此默认实现。

第二种是 private void privateHelperMethod() {…}，这是 Java 9 及更高版本中接口支持的私有方法。注意，这个方法只能在接口内部被使用，实现类无法访问或覆盖它。

接下来，有两个类实现了这个接口如下：

```
// 展示接口中不同类型的方法在实现类中的区别
package com.liuyandong.methoddifference;

// Cat 和 Dog 类实现 Feedable 接口
class Cat implements Feedable {
    @Override
    public void eat() {
        System.out.println(" 猫吃东西");
    }
```

```
    // 可选地，可以覆盖默认方法
    @Override
    public void defaultFeed() {
        System.out.println(" 猫覆盖了接口中自定义的默认方法");
    }
}

class Dog implements Feedable {
    @Override
    public void eat() {
        System.out.println(" 狗吃东西");
    }
}
public class MethodsDifference {
    public static void main(String[] args) {
        Feedable myCat = new Cat();
        Feedable myDog = new Dog();

        // 输出 " 猫吃东西"
        myCat.eat();
        // 输出 " 猫覆盖了接口中自定义的默认方法"
        myCat.defaultFeed();

        // 输出 " 狗吃东西"
        myDog.eat();
        // 输出 " 接口中定义的默认方法."
        myDog.defaultFeed();

        // 注意：接口的私有方法不能在实现类中调用，下面这行代码会报错
        // myCat.privateHelperMethod();
    }
}
```

在主函数中，我们创建了 Cat 和 Dog 对象并将其引用类型声明为 Feedable，这是因为它们都实现了 Feedable 接口，所以可以调用接口中定义的所有公共方法。

当调用 myCat.eat() 和 myDog.eat() 时，分别输出对应的动物进食信息。

调用 myCat.defaultFeed() 时，由于 Cat 类覆盖了 defaultFeed() 方法，所以会输出覆盖后的内容。

调用 myDog.defaultFeed() 时，因为 Dog 类没有覆盖该方法，所以会执行接口中提供的默认实现。

最后关于注释部分，指出接口中的私有方法（如 privateHelperMethod()）在实现类中是不可见的，因此无法在实现类中直接调用。在接口中引入默认 default 方法是接口的一大重要改进，接下来，我们单独讲一下原因。

13.4 接口中引入 default 方法

在先前的示例中，我们借助 Animal 接口定义了一系列抽象方法，要求任何实现它的类必须提供相应的具体实现。随着需求演进，设想我们需要在 Animal 接口中新增一个名为 breathe() 的抽象方法。此时，已实现 Animal 接口的 Dog 类和 Cat 类将面临一个挑战：一旦我们在接口中增加新方法，这两个类均需各自独立地去实现 breathe() 方法。但考虑到现实中，狗和猫的呼吸方式并无差异，我们并不希望为了遵循接口而在每个类中重复相同的实现代码。

为解决这一问题，我们引入了 Java 8 中 default 关键字的功能特性。利用 default 关键字，我们能够在接口自身中提供一个 breathe() 方法的标准实现。这样一来，对于那些实现了 Animal 接口的类，如果它们没有自行提供 breathe() 方法的个性化实现，便会自动采纳接口中预设的默认实现。这种方式有效地消除了重复代码，从而提升了代码的复用性和可维护性。

接下来，看看加入了 default 方法的 Animal 接口：

```
package com.liuyandong.interfacedemo;

public interface Animal {
    // 声明一个所有动物都应具备的行为：发出声音
    void makeSound();
    default void breathe() {
        System.out.println(" 用肺呼吸");
    }
}
```

这样，Dog 类和 Cat 类不用做任何修改，便可以调用 breath() 方法。所以前面的测试代码运行之后输出结果如下：

喵 喵 喵

汪 汪 汪
用 肺 呼 吸
用 肺 呼 吸

然而，我们还需要解决一个问题。假设我们引入了一个新的 Fish 类，而鱼的呼吸方式并不是 Animal 接口默认提供的方式。在这种情况下，我们可以选择覆盖接口的默认方法。这样，Fish 类就可以提供适合自己的 breathe() 方法的实现，而不是使用 Animal 接口默认的实现。代码如下：

```
// 覆盖了接口中 default 方法的 Fish 类
package com.liuyandong.interfacedemo;

// Fish 类实现 Animal 接口并覆盖 breathe() 方法
public class Fish implements Animal {
    @Override
    public void makeSound() {
        System.out.println(" 啵啵啵");
    }

    // 覆盖默认的 breathe() 方法
    @Override
    public void breathe() {
        System.out.println(" 用鳃呼吸");
    }
}
```

上面的代码定义了一个名为 Fish 的类，它实现了 Animal 接口。这意味着 Fish 类提供了 makeSound() 方法的实现，输出” 啵啵啵”。同时，Fish 类也覆盖了 breathe() 方法，提供了自己的实现，输出” 用鳃呼吸”。

在这个例子中，Fish 类覆盖了接口的默认方法，提供了自己的实现。这是 Java 8 引入默认方法的一个重要特性，允许我们在不破坏实现接口的类的情况下，向接口添加新的方法。同时，实现接口的类也可以选择覆盖这些默认方法，提供自己的实现。

最后用下面的代码测试一下我们新定义的 Fish 类是否达到预期效果：

```
// 测试 Fish 类是否正确运行
package com.liuyandong.interfacedemo;
```

```
public class Main {
    public static void main(String[] args) {
        Animal fish = new Fish();
        fish.makeSound();
        fish.breathe();

    }
}
```

以上代码运行之后，输出结果如下：

```
啵 啵 啵
用 鳃 呼 吸
```

综合以上的例子，我们可以看到，在接口中添加新的方法通常会破坏实现该接口的所有类，因为它们需要实现新添加的方法。默认方法允许我们在不破坏现有代码的情况下，向接口添加新的方法。这是因为新添加的方法有一个默认的实现，所以实现接口的类不需要实现这个方法，除非它们需要覆盖这个默认实现。

13.5 接口中引入静态方法

自从 Java 8 版本起，接口的功能得到了显著增强，其中一项关键创新便是引入了静态方法。如同在类中引入静态方法的作用一样，接口内的静态方法也旨在封装那些与接口相关的独立于实例的方法。第 9 章曾探讨了类中静态方法的概念与应用，故在此不再深入展开。在此之前，Java 接口仅限于声明抽象方法，不允许拥有具体实现，这一局限性在一定程度上制约了接口的能力。

Java 8 的新特性允许接口直接定义静态方法，这就意味着开发者可以把与接口紧密相关的一些通用工具方法或辅助功能直接集成到接口之中，进而显著提升接口的便捷性。接下来，我们一起举一个例子，看看变化前后对编程有什么影响。

在 Java 8 之前，接口的设计原则主要是定义一组行为规范，而不能包含具体实现。这种情况下，如果存在一些与接口逻辑紧密相关的辅助方法，往往需要单独放置在一个工具类中供各个实现类使用，增加了类之间的耦合度和调用复杂度。假设有这样一个场景，我们有一个名为 Shape 的接口，用来表示各种形状，同时我们有一系列几何计算工具方法，比如计算面积、周长等。在 Java 8 之前，可能会有一个独立的 GeometryUtils 工具类来提供这些通用方法：

```
public class GeometryUtils {
    public static double calculateArea(Shape shape) {
        // 根据 shape 类型调用不同的计算方法
    }

    public static double calculatePerimeter(Shape shape) {
        // ...
    }
}

interface Shape {
    // 声明抽象方法，如 getBounds() 等
}
```

然而，在接口中引入了静态方法之后，我们可以直接在 Shape 接口中定义这些静态方法：

```
interface Shape {
    // 抽象方法依旧存在
    double getArea();

    double getPerimeter();

    // Java 8 新特性：静态方法
    static double calculateArea(Shape shape) {
        return shape.getArea();
    }

    static double calculatePerimeter(Shape shape) {
        return shape.getPerimeter();
    }
}
```

这样一来，与形状计算相关的静态方法直接嵌入到了 Shape 接口中，使得使用者无需额外引入工具类即可直接通过接口调用这些辅助方法，增强了代码的内聚性和可读性，同时也降低了对第三方类的依赖。这对于接口用户来说，无疑是一个更直观、更易于使用的改进。

```
package com.liuyandong.shapewithstaticmethod;

// 定义 Shape 接口，其中包含抽象方法和静态方法
interface Shape {
    // 抽象方法
    double getArea();

    double getPerimeter();

    // Java 8 新特性：静态方法
    static double calculateArea(Shape shape) {
        return shape.getArea();
    }

    static double calculatePerimeter(Shape shape) {
        return shape.getPerimeter();
    }
}

// 实现 Shape 接口的 Circle 类
class Circle implements Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override
    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }
```

```
}

// 实现 Shape 接口的 Rectangle 类
class Rectangle implements Shape {
    private final double width;
    private final double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }

    @Override
    public double getPerimeter() {
        return 2 * (width + height);
    }
}

public class ShapeWithStaticMethod {
    public static void main(String[] args) {
        // 创建 Circle 和 Rectangle 对象
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);

        // 直接通过接口的静态方法计算面积和周长
        System.out.println(" 圆的面积: " +
                           Shape.calculateArea(circle));
        System.out.println(" 圆的周长: " +
                           Shape.calculatePerimeter(circle));

        System.out.println(" 长方形的面积: " +
```

```
                Shape.calculateArea(rectangle));
        System.out.println(" 长方形的周长: " +
                Shape.calculatePerimeter(rectangle));
    }
}
```

在这个例子中，Shape 接口不仅定义了抽象方法，还定义了两个静态方法 calculateArea() 和 calculatePerimeter()，这些静态方法接收一个 Shape 类型的参数，可以直接调用传入对象的相应抽象方法获取面积和周长。在 main 方法中，可以通过接口的静态方法方便地计算不同形状的面积和周长，而无需显式地引用实现类。

13.6 接口的继承与多重继承

在本书前面章节讲了类继承的概念。在 Java 中，继承是一个面向对象编程的重要概念，它允许一个类（称为子类或派生类）继承另一个类（称为父类或基类）的属性和方法。Java 中的继承是单继承，也就是说，一个类只能直接继承一个单一的父类，但是通过继承体系可以实现多层继承的效果，即一个类继承自另一个类，而那个类又可以继承自另一个类，形成继承链。

而对于接口，Java 提供了更为灵活的继承机制。在 Java 中，接口可以继承其他接口，且可以继承多个接口。接口的继承使用关键字 extends，例如：

```
interface InterfaceA {
    void methodA();
}

interface InterfaceB {
    void methodB();
}

interface InterfaceC extends InterfaceA, InterfaceB {
    void methodC();
}
```

上述代码中，InterfaceC 继承了 InterfaceA 和 InterfaceB，这意味着任何实现 InterfaceC 的类都需要同时实现 methodA()、methodB() 和 methodC() 这三个方法。

由于接口可以继承多个接口，所以实际上 Java 通过接口实现了多重继承的效果。一个接口能够继承多个接口，从而合并多个接口中的方法声明。实现类通过实现这个接口，就

相当于间接实现了所有父接口的方法。接下来，通过例子来展示一下在 Java 中，如何通过接口来实现多重继承的。

```
package com.liuyandong.multipleinterfaces;

// 定义接口
interface InterfaceA {
    void methodA();
}

interface InterfaceB {
    void methodB();
}

interface InterfaceC extends InterfaceA, InterfaceB {
    void methodC();
}

// 创建一个实现 InterfaceC 的类，
// 该类需要实现 InterfaceA、InterfaceB 和 InterfaceC 的所有方法
class MyClass implements InterfaceC {
    @Override
    public void methodA() {
        System.out.println(" 实现了 InterfaceA 接口中
                                                的 methodA 方法");
    }

    @Override
    public void methodB() {
        System.out.println(" 实现了 InterfaceB 接口中
                                                的 methodB 方法");
    }

    @Override
    public void methodC() {
        System.out.println(" 实现了 InterfaceC 接口中
                                                的 methodC 方法");
```

```
    }
}

public class MultipleInterfaces {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // 通过接口引用调用方法
        InterfaceC myInterfaceObj = obj;

        // 输出 " 实现了 InterfaceA 接口中的 methodA 方法"
        myInterfaceObj.methodA();
        // 输出 " 实现了 InterfaceB 接口中的 methodB 方法"
        myInterfaceObj.methodB();
        // 输出 " 实现了 InterfaceC 接口中的 methodC 方法"
        myInterfaceObj.methodC();
    }
}
```

这种设计解决了 Java 类层次结构中可能出现的菱形继承问题，同时保证了接口的纯粹抽象性，因为接口本身不能有任何实现，只是定义了一组方法签名，确保了不会出现方法冲突的问题。既然讲到了菱形继承问题，我们来研究一下，为什么 Java 不允许类多重继承，而允许接口多重继承呢?

13.7 菱形继承问题

在 Java 中通常所说的“菱形继承”问题是指多继承的潜在问题，而不是 Java 实际遇到的情况，因为 Java 并不支持类之间的多继承。但在 C++ 中，多继承可能导致菱形继承问题，表现为同一个基类在继承体系中被多次引入，从而在子类中产生冗余的数据成员和方法调用的不确定性。

在 C++ 中，如果类 D 同时继承了类 B 和类 C，而类 B 和类 C 又都继承自同一个类 A，就会形成菱形继承结构:

在这种情况下，如果不采取特殊措施，类 D 中将会有两份来自类 A 的数据成员副本，造成数据冗余。另外，如果类 A 中有虚函数，并且类 B 和类 C 都覆盖了这个虚函数，那么在类 D 中调用这个虚函数时可能会出现二义性。

为了解决菱形继承带来的问题，C++ 引入了虚拟继承（Virtual Inheritance），通过关键

字 virtual 声明基类，使得在继承体系中只会生成一份基类 A 的实例，从而避免数据冗余和调用歧义。

而在 Java 中，由于 Java 只允许单继承（即一个类只能继承一个父类），并通过接口（Interface）来实现类似多继承的效果，所以不存在上述菱形继承问题。Java 接口之间可以多继承，但是接口不包含数据成员，只包含抽象方法（在 Java 8 及以后版本可以包含默认方法和静态方法），因此即便接口之间形成了菱形结构，也不会出现与 C++ 中相同的数据冗余和二义性问题。

在 Java 中讨论多重继承，并不是完全没有用处，尤其是接口越来越像类之后。Java 中所谓的 “菱形继承” 讨论，更多的是在探讨接口间的继承关系以及可能出现的多实现问题，尤其是在 Java 8 引入了 default 方法之后，当一个类实现多个接口，而这些接口中有同名的 default 方法时，Java 要求实现类明确指定调用哪一个接口的 default 方法，或者自己提供一个实现来消除歧义。但这并不是真正的菱形继承问题，而是接口多实现下的一种冲突解决策略。

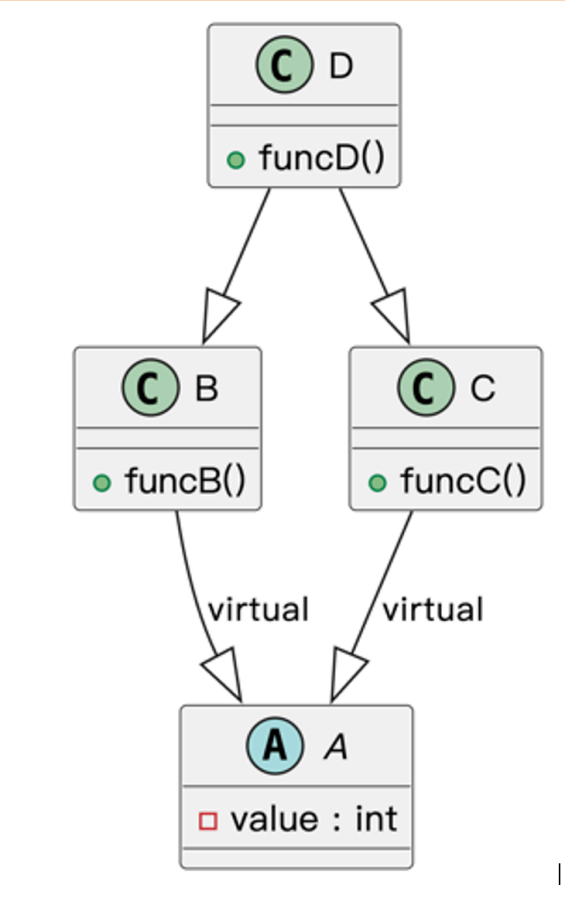

图 13.1: C++ 语言中 “菱形继承” 问题

13.8 程序员故事

“外向” 的程序员

我喜欢去乡村野外出差，公司里人人都知道了。因此，一旦有这样的机会，公司都愿意让我去，我也很乐意去远离城市的地方度过一段美好的时光。

得益于互联网，现在乡下和城市的关系在有些行业越来越紧密了。牛奶行业就是其中之一，城里的客户想了解奶牛厂的情况，他们想知道牛奶是不是干净无公害的，牛奶公司为了满足这样的需求，安装了监控，供广大的客户随时查看奶牛和牛奶厂的情况。农场建在远离城市的地方，想拥有速度很快的带宽并不是一件容易的事情。我要去的这个农场就是如此，在群山环绕中，有一大片广袤的草地，我沿着山路开车，沿途看过来，颇有一番“风吹草地现牛羊” 的风景。

来之前，我大体了解了这家农场的情况，农场有 800 多头牛，产奶的有 400 多头，一天可以产奶 6 吨多。已经在这里经营了 20 年，附近的城市都非常认可这个当地的品牌，农场除了养牛，还有其它类似农家乐这样的产业，让我们部署一个可以支持 5 万客户的监控。如果在城里，这没什么难度，但是这是在山上，奶牛又不上网，因此手机的信号非常弱，也没有光纤。

我到了农场以后，和工作人员说明了来意，结果所有人都很忙，分不出人手来与我接洽这个事情，让我先去屋里坐着等，于是我百无聊赖的等着。

“你是来部署光纤的么？请问尊姓大名？” 突然有个姑娘边笑边问边向我走来。

“啊……是的，我是来看看能不能部署光纤的，别人都喊我栋哥。我要先看一下具体情况才知道能不能部署光纤，毕竟拉根光纤到这里，并不容易。”

“好，我带你在这里看看吧。现在工人都很忙，现在正是挤奶的时候，他们都在挤奶。我有个很好记的名字，不许笑哦，我爸妈可能很喜欢用雅芳洗发水吧，他们给我起了个名字叫雅芳。”

雅芳领我到了屋外，笑着对我说：“你是城里来的吧，今天可要爬爬山了，就当锻炼了，这里总共有 8 个奶牛饲养场，这 8 个分布在不同的地方，你可以瞧见那几个。” 她边说边指了指山腰上几个白色的大房子。

我瞧着眼前这个叫雅芳的女孩说：“我当然不是城里人啊，我是农村长大的，农村做题家，现在在城里打工而已啦。爬这点山，对我一点难度也没有，要不选个最远的过去吧？”

“我们抄一条近路吧，就是有点难走，你还要带什么设备么？”

“好的，不用带设备，今天我只是来看看，大概估算一下距离，回去我写一份报告就可以了。”

她带着我穿过了一条深深的峡谷，这条峡谷一定经过了千百万年的冲刷，坚硬的石头

已经被溪水冲成了深深的沟壑。石头缝里，松树坚强的伸了出来，遮天蔽日的松叶让阳光无法直射下来。我们就这样行走在松软的土地上，比地毯还要舒服，多少年的松针一层一层的堆积在脚下。

一路上，我跟她讲我认识的每一种植物，还有哪些植物能吃，哪些植物不能吃，哪些植物好玩，哪些植物味道难闻。看到这些多年不见的植物，就像见到多年前的老朋友一样。我如数家珍的告诉她这种植物叫马齿苋，做熟了很好吃。那种植物叫灰菜，无论怎么洗，怎么做，吃到嘴里总有一种沙子的感觉，所以叫灰菜，洗不完的灰。这种植物叫苘麻，用来做绳子非常结实，种子可以做润滑油。那种植物叫曼陀罗，动物都不会吃这种植物，武侠小说里的蒙汗药就是用这种植物的种子做的，我们小时候不怕死，就用种子冲水喝，太难喝了，像喝油漆，喝完以后天旋地转的……我滔滔不绝的讲个不停，她只是偶尔回应一下。

不得不承认，这些年的城市生活，让我的体力有所下降。我渐渐跟不上她的脚步了。她依然健步如飞，我却只能硬撑着追赶。终于爬上了这个沟壑，来到了一个白色的奶牛棚，走进去一看，才知道与我想象的不同，现在的设备比我想象的要先进多了，挤奶，喂料，甚至奶牛在里面的行走路线都是全机械化来控制的。我高兴的问东问西，而且，不顾她的阻拦，我喝了一小杯没有经过巴氏消毒的牛奶。

很快，她带着我走完了几个牛棚，回来的路上，可能她已经察觉到气喘吁吁的我，就和我在树荫下休息。峡谷里的水声响在耳畔，柔和的风带着花香徐徐吹来，伴随着一些不知名的鸟叫。我陶醉在这优美的风景里，仿佛互联网，会议和程序员都已经离我远去。“你在这里工作可真是幸福，”我说，“每天都能看到这么好的风景。”

“我非常喜欢这里的生活，我也上过大学，而且，在大学里我也学的是计算机，应该跟你差不多的专业吧。”她停了一下，眺望着远方说：“看得出你确实是农村出来的，因为只有农村土地里长出来的孩子，才知道那么多植物。”

我不禁哈哈大笑：“是啊，做梦都想在农村呆着，光着脚丫在水库边上玩的日子，一去不复返了。”

接下来，我又跟她谈到我的工作，如何写代码，会碰到什么样的问题，然后一路谈到大学的生活，谈到小时候在农村生活的日子，谈到现在的同事，谈到潘新和潘闻，还有我那些一直没实现的梦想与抱负……

我从来不是一个外向的人，但是我今天竟然如此滔滔不绝的高谈阔论，甚至有时候我觉得她已经听烦了。但是她一直坐在那里，双臂抱住自己的膝盖在那里静静的听着，时不时的微笑。

“别人都说程序员都非常不善于交际，显然你有点例外啊，感觉你还是非常外向的。”她笑着说。

“不是有句话叫‘酒逢知己千杯少，话不投机半句多’么。其实，我也是不善于交际的，也许我这种性格属于依赖他人型外向吧，是因为你让我变得外向了。”

在返回的路上，我们又谈了一些，时间过得太快了。

“好了，我回去写一份可行性报告交上去，我会回来拉网线的。”我便侧身去开车门。

她微笑的回答：“谢谢你，最好能快点把网线拉好，让我们这个农村，也能早点用上高科技。”我把车子发动以后，放下车窗，和她挥手再见，她便转身回去了。

回去的路上，我一直在想怎么写这份报告，能让公司快点派我来拉光纤。

14 多态

Contents

为什么需要多态呢？假设你指挥一个交响乐队，里面有钢琴家、小提琴家、大提琴家等音乐家。当交响乐队演奏的时候，乐队的指挥会对这些不同各类的音乐家发出统一的“演奏”指令，然后这些不同类型的音乐家如何去弹奏不同的乐器，就不必管了。多态追求的目标就是“统一接口，不同实现”。

面向对象编程有三大基本原则: 封装、继承和多态。前面已经讲了封装与继承，现在讲最后一个原则：多态。多态是面向对象编程的一个核心概念, 它允许我们使用一个通用的接口来表示和操作不同的对象实例。

多态是面向对象编程中一个强大的工具，使代码更加灵活和可扩展。通过多态，代码可在运行时动态改变行为，根据实际情况决定调用哪个方法实现，而非在编码阶段就确定。这种动态灵活性，不仅便于在无需修改现有代码的情况下添加新行为，也使得代码可通过实现同一接口来扩展新类。

在 Java 中，主要使用继承、接口和泛型编程来实现多态。虽然这些概念很多在前面的学习中已经接触过，但可能还没有意识到它们实际上都是多态思想的实现基础。为了加深对多态概念的理解，让我们从另一个视角来审视之前学过的知识。

14.1 用继承来实现多态

在 Java 中通过继承来实现多态，需要依赖方法重写和向上转型这两项技术，缺一不可。接下来，我们来定义一下这两个概念：

方法重写 (Override): 子类重写父类的方法, 根据对象实际类型调用重写后的方法实现。这样同一个方法调用在不同对象实例上会表现出不同的行为，体现了多态性。

向上转型 (Upcasting): 将子类对象向上转型为父类对象的引用，在编译时加入多态性，使编译器只能调用父类中存在的方法，确保代码的统一性。在运行时，JVM 会根据实际对象类型去调用重写后的方法。

接下来，我们来看看这两个技术如何实现多态的，先从方法重写开始讲起：

14.1.1 方法（Override）重写为多态提供基础

在讲继承的时候，我们已经学过了方法的重写，当时并没有讲多态，这次咱们从多态的角度重新审视重写这个技术的作用。

方法的重写（Override）是指子类重新定义了一个与父类中的方法具有相同签名 (方法名、参数列表) 的方法。当子类继承父类后, 可以根据实际需求, 重新定义父类中已有的方法体内的代码逻辑。

方法重写在多态中扮演着至关重要的角色。方法重写为多态提供了可运行的基础，让多态行为得以真正实现。重写使得子类能够拥有自身独特的行为表现，而不再被统一约束于父类行为。这就好比一支交响乐队中的各种乐器，尽管它们有着继承的关联（比如都是乐器的子类），但也各自蕴含着独特的演奏方式。当指挥家发出 “演奏” 这一统一的指令时，每一种乐器都按照自身的演奏方法予以实施，钢琴奏响婉转动听的乐章，小提琴则高亢曼妙，定音鼓则砰砰直击人心，共同演绎出动人心弦的乐曲全景。正是由于这些不同乐器的对 “演奏” 方法重写，承载了各自独特的 “行为”，使整个乐队在一致的 “演奏指令” 下，演绎出了丰富多彩的音乐篇章。

接下来，还是以前面章节讲过的动物类之间的继承关系，来展示重写。

```
// 定义一个父类 Animal
class Animal {
    public void makeSound() {
        System.out.println(" 动物在叫");
    }
}
```

```
// 定义一个继承自 Animal 的子类 Dog
class Dog extends Animal {
    // 重写父类的 makeSound 方法
    @Override
    public void makeSound() {
        System.out.println(" 汪汪汪");
    }
}

// 测试类
public class PolymorphismDemo {
    public static void main(String[] args) {
        // 创建 Animal 对象和 Dog 对象
        Animal animal = new Animal();
        Dog dog = new Dog();

        // 调用 makeSound 方法
        animal.makeSound(); // 输出: 动物在叫
        dog.makeSound(); // 输出: 汪汪汪
    }
}
```

在这段代码中，Animal 是一个父类，Dog 是它的子类。Dog 类重写了 Animal 类的 makeSound 方法。在 PolymorphismDemo 类的 main 方法中，创建了两个 Animal 类型的引用，一个指向 Animal 对象，一个指向 Dog 对象。

当调用 animal.makeSound() 时，因为 animal 指向的是 Animal 对象，所以调用的是 Animal 类的 makeSound 方法，输出 “动物在叫”。

当调用 dog.makeSound() 时，因为 dog 是 Dog 类型的引用，因此 JVM 会调用 Dog 类中重写的 makeSound 方法，输出 “汪汪汪”。

值得注意的是，在 Java 中存在一个与方法重写 (method overriding) 相近但又有所区别的概念，那就是方法重载 (method overloading)。方法重载并不属于多态的范畴，它发生在编译期，而非运行时。方法重载允许在同一个类中定义多个方法名相同但参数列表不同 (参数类型、数量或顺序有所差异) 的方法。当调用重载方法时，编译器会根据传入的实参类型和数量，静态决定调用哪个重载版本，这是一种编译期的静态绑定行为。

重载的设计目的在于增强代码的可读性和使用便捷性，使程序员无需记住过多冗长的方法名称。但它并不涉及动态绑定，被调用的方法实现在编译期就已经确定。

相比之下，多态则是一种运行时的动态绑定行为。通过方法重写和对象实例的向上转型，同一个方法调用可以在运行时根据对象的实际类型，动态选择并执行对应的方法实现版本。

因此，尽管重写和重载都允许我们提供多种方法实现，但它们是两个不同的概念和机制。重载发生在编译期，属于静态绑定; 而多态则发生在运行期，是动态绑定的体现。所以，一般不会将方法重载视为多态的一种形式。

接下来，进入向上转型的学习。

14.1.2 向上转型（Upcasting）为多态提供统一调用方式

在 Java 中，我们可以将一个子类对象赋值给父类类型的变量，这种做法被称为向上转型 (upcasting)。向上转型是安全可靠的，因为子类对象包含了父类的全部特性和行为，它本质上就是一个父类对象。

通过这个指向子类实例的父类引用，我们可以调用在父类中定义的方法。如果该方法在子类中被重写过，JVM 就会动态绑定并执行子类重写的那个版本。这种在运行期根据对象实际类型去执行对应的方法行为，正是多态机制的体现。正是依赖这种向上转型后的动态绑定机制，我们才能在运行时自动调用重写的方法版本，发挥多态的强大魔力，使代码更加扩展性强、可维护性高。

```
// 使用向上转型和方法重写来实现多态的代码
// 定义一个父类 Animal
class Animal {
    public String name;

    public void makeSound() {
        System.out.println(" 动物在叫");
    }
}

// 定义一个继承自 Animal 的子类 Dog
class Dog extends Animal {
    public int age;

    // 重写父类的 makeSound 方法
    @Override
    public void makeSound() {
```

```
        System.out.println(" 汪汪汪");
    }
}

public class PolymorphismWithUpcastingDemo {
    public static void main(String[] args) {
        // 使用向上转型实现多态
        Animal dog = new Dog();
        dog.makeSound(); // 输出: 汪汪汪

        // 不使用向上转型
        Dog dog2 = new Dog();
        dog2.makeSound(); // 输出: 汪汪汪
        dog2.name = " 旺财";
        dog2.age = 3;

        // 使用向上转型
        Animal animal = dog2;
        System.out.println(animal.name); // 输出: 旺财
        // 编译错误: age 在 Animal 中不存在
        // System.out.println(animal.age);
        animal.makeSound(); // 输出: 汪汪汪

    }
}
```

14.1.3 继承实现多态的核心机制

要深入理解向上转型的原理，需要理解对象在内存中的具体组织方式，这有助于我们更深入地把握面向对象编程的本质。接下来，我要讲一个对象在内存中的组织方式:

在 Java 中，每个对象实例都会在内存中存储两种数据:

1. **对象的实际数据（属性）**。值得注意的是，如果类中没有显式定义任何属性，它也会自动获得一些来自 Object 类的属性，例如用于对象锁定的 monitor 字段等。但是这些字段对程序员来说是透明的，我们通常将它们忽略。
2. **指向该对象所属类的方法表（Method Table）的一个引用**。所谓方法表，是一种数据结构，存储着该类的所有方法的元数据信息，这些信息包括: 方法名、方法参数类型、

方法返回类型以及方法的内存入口地址 (执行代码的内存地址)。

每个类都在内存中有一张自己的方法表。当一个对象被创建时，它会获得一个指向其所属类的方法表的引用。这个引用占用的内存空间通常是一个指针的大小（在 64 位系统中是 8 个字节）。通过这个引用，对象就可以在运行时找到并调用其对应的方法实现。如果子类重写了父类的某个方法，子类对象的这个引用就会指向子类自身的方法表，从而调用重写后的方法版本，实现了多态行为。

用这些理论解释上面的代码如下，先来看对象实例在内存中的存储情况：

先来说 Animal 类的对象实例存两种数据：第一种是对象实际数据，也就是存储 name 字段的值。第二种是指向 Animal 类的方法表引用，这个方法表中保存的是 Animal 类中的 makeSound() 方法的元数据信息（方法名、方法参数类型、方法返回类型以及方法的内存入口地址）。

同样的，Dog 类的对象实例也存两种数据：第一种是存储从 Animal 类中继承而来的 name 字段的值，以及 Dog 类自身的 age 字段的值。第二种是指向 Dog 类的方法表引用，这个方法列表中保存的是 Dog 类中的 makeSound() 方法的元数据信息（方法名、方法参数类型、方法返回类型以及方法的内存入口地址）。

还有一种情况需要考虑，假设 Dog 类没有重写 Animal 类中的 makeSound() 方法，那么，Dog 类中的方法表中将包含一个指向 Animal 类的 makeSound() 方法的引用。这种设计的好处是节省内存空间, 避免在 Dog 类的方法表中重复存储 makeSound() 方法的元数据信息。因为 Dog 类并没有重写 makeSound() 方法, 所以它可以直接继承和复用父类 Animal 提供的 makeSound() 方法实现。

现在，当我们将一个子类对象赋值给一个父类类型的变量时，也就是向上转型（upcasting），比如示例代码中的 “Animal animal = dog2;” 这行代码实际上发生了以下过程:

o 子类对象的数据部分被当作父类类型看待，父类无法访问子类新增的字段。

这句话的意思是：对于 dog2 这个子类对象，它的数据部分有两个，一个是 name 属性，一个是 age 属性。这个 dog2 对象的数据部分，只能当作是 Animal 类型的数据部分，换句话说，只可访问 Animal 类中的 name 属性，而不可访问 Dog 类中的 age 属性。所以当访问 animal.age 时会报错。

o 子类对象的方法表引用被保留不变。

这句话的意思是，dog2 这个对象的方法表引用是指向 Dog 类的 makeSound() 方法。所以当访问 animal.makeSound() 时，使用的是 Dog 类的方法表引用。

这意味着，当通过父类类型的变量调用某个方法时:

1. 如果该方法在父类中没有被重写，则直接从父类的方法表中获取方法的入口地址并执行。

2. 如果该方法在子类中被重写了，则会从子类对象的方法表中获取重写后方法的入口地址并执行。

所以，多态是通过方法表的调用机制来实现的。JVM 能够在运行时动态决定该调用哪个方法实现版本，从而实现了同一操作符对应不同对象时产生不同行为的效果。

这种动态绑定的能力正是多态所体现的主要特性，它使得面向对象的程序设计更加灵活和扩展性更好。我们可以编写更加通用的代码，无需关心运行时对象的具体类型，增强了代码的可复用性。

14.2 用接口（Interface）实现多态

在 Java 中，一个类可以实现一个或多个接口，接口中定义的方法如果是抽象的，意味着它们没有默认的实现。一个类实现了一个接口，那么它就必须提供该接口所有方法的实现。然后，我们可以将实现了某个接口的类的对象赋值给该接口类型的变量，然后通过这个变量调用接口中的方法。在运行时，JVM 会调用实际对象的方法。这也是多态的一种非常重要的形式。

接下来的代码是定义 Animal 接口：

```
// 定义一个接口 Animal，它声明了一个抽象方法 makeSound
public interface Animal {
    void makeSound();
}
```

然后，有两个类，分别为 Dog 类与 Cat 类，实现了这个接口，其代码如下：

```
// 定义一个类 Dog，它实现了 Animal 接口
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println(" 汪汪汪");
    }
}

// 定义另一个类 Cat，它也实现了 Animal 接口
class Cat implements Animal {
    @Override
    public void makeSound() {
```

```
        System.out.println(" 喵喵喵");
    }
}

// 测试类，体现多态和接口应用
public class PolymorphismWithInterfaceDemo {
    public static void main(String[] args) {
        // 创建 Dog 和 Cat 对象
        Animal dog = new Dog();
        Animal cat = new Cat();

        // 通过 Animal 类型的引用调用 makeSound 方法
        dog.makeSound(); // 输出：汪汪汪
        cat.makeSound(); // 输出：喵喵喵

        // 尽管 dog 和 cat 变量都是 Animal 类型，
        // 但由于它们引用的实际对象类型不同，
        // JVM 在运行时会调用相应对象的 makeSound 方法，
        // 这就是多态的体现
    }
}
```

上面这段代码通过接口来实现多态。Animal 是一个接口，它定义了一个方法 makeSound。然后，Dog 和 Cat 类都实现了这个接口，也就是说，它们都提供了 makeSound 方法的实现。

在 main 方法中，创建了两个 Animal 类型的引用 dog 和 cat，分别指向 Dog 和 Cat 对象。然后，通过这两个引用调用 makeSound 方法。

虽然 dog 和 cat 都是 Animal 类型的引用，但是它们指向的实际对象类型是 Dog 和 Cat。因此，当调用 dog.makeSound() 和 cat.makeSound() 时，JVM 会在运行时根据实际的对象类型来决定调用哪个方法。如果 dog 指向的是 Dog 对象，就调用 Dog 类的 makeSound 方法；如果 cat 指向的是 Cat 对象，就调用 Cat 类的 makeSound 方法。

这就是多态的体现：同一个方法调用，在运行时可以有不同的行为，取决于对象的实际类型。

14.2.1 接口实现多态的核心机制

用接口实现多态背后的机制跟用继承实现多态背后的机制差不多。有关类的实例如何在内存中存储，存储了些什么东西，前文已有详细描述，在些不再赘述。在此，只讨论接口本身在内存中是如何存储的。具体来说，当程序开始运行，接口在内存中存如下几部分内容：

o **接口常量池**：接口中定义的所有常量（public static final 修饰的常量）都会存储在这里。在本例中没有常量，因此没有东西。

o **接口方法数据**：包括接口中定义的抽象方法、默认方法和静态方法的名称、返回类型、访问权限修饰符等元数据信息。在本例中是 makeSound() 这个方法。

o **接口继承的信息**：如果接口继承了其他接口，继承层次结构的相关信息也会存储在这里。在本例中的接口没有继承其它接口，因此没有。

o **接口注解信息**：被标注在接口上的注解内容会被存储。在本例中也没有。

大家可能发现了，与类在内存中的存储相比，接口本身并不存储任何实例字段和实例方法的实现代码，因为它是完全抽象的。当类实现某个接口时，该接口方法的实际实现代码会存储在实现类的元数据中。

在本例中，当我们将实现类对象（如 Dog 或 Cat 对象）赋值给接口类型（Animal）的变量时，发生了一次向上转型 (Upcasting) 的过程。前文已经讲过，这种向上转型是安全的，因为实现类对象本身就实现了接口定义的契约。

然后，当我们通过这个接口类型的变量来调用接口中声明的方法时，JVM 的动态绑定机制会介入，这时，要分为两个阶段来讨论：

第一个阶段为**编译期绑定**：在编译阶段，编译器会检查调用的方法是否在接口中存在，并对方法签名和参数类型进行检查。但此时编译器无法确定调用的具体实现版本。

第二个阶段为**运行期绑定**：当程序执行时，当调用接口方法时，JVM 会根据实际对象的类型（这里是 Dog 或 Cat），动态查找并执行该对象所属实现类中重写的方法版本。

也就是说，编译器只能检查被调用方法在接口中存在与否，而无法确定调用的实际实现。JVM 在运行时，通过动态绑定，根据实际对象的类型，自动调用该类中重写的方法实现。这种动态绑定机制使同一个方法调用可以根据对象的实际类型呈现出不同的行为，体现了多态的本质。

再回到例子中，当调用 dog.makeSound() 时，JVM 动态绑定了 Dog 类的 makeSound() 实现；而 cat.makeSound() 则绑定到了 Cat 类的 makeSound() 实现。

通过接口实现多态，我们只需关注抽象的接口契约，不必过多关心具体实现的细节，这提高了代码的可维护性和扩展性。同时，动态绑定使得新增不同实现类非常方便，提高了系统的灵活性。

总之，利用接口实现多态的关键就在于 Java 的动态绑定机制，使得相同的接口方法调用可以在运行时动态选择并执行不同对象的实现版本，从而实现了行为的多态性。

14.3 用继承跟用接口实现多态有什么不同?

使用继承中的重写与向上转型实现多态，和使用接口实现多态，虽然最终实现的多态效果是类似的，但在实现机制和使用场景上还是有一些区别:

第一，两者的实现机制不同

使用重写是基于继承关系，通过子类重写父类方法，配合向上转型，在运行时动态调用子类重写的方法版本。而使用接口实现多态, 是通过不同的实现类实现相同的接口方法，调用时基于接口类型的引用来动态绑定不同版本的方法。

第二，代码耦合度不同

继承会增加父子类之间的耦合度。而接口只定义了抽象方法，各实现类之间互相独立，耦合度更低。接口实现多态更利于代码解耦和可扩展性。

首先，使用继承实现时，子类与父类之间存在紧密的耦合关系。因为子类继承了父类的内部属性和实现代码，任何对父类的修改都可能直接影响到子类的行为。这种继承关系造成了代码的高度耦合，降低了灵活性和可维护性。

而接口则不同，它只定义了方法签名，没有具体的实现代码。不同的实现类在实现同一接口时，实现代码是相互独立的，没有直接的耦合关系。实现类只与接口发生了很小的依赖关系，这种依赖关系远比继承的耦合关系要小得多。

这种低耦合特性给予了接口更好的可扩展性。比如，新增一个接口实现类, 不会影响到其他实现类的代码。如果对接口进行修改, 只需要实现类重新实现即可。

因此, 从设计的角度来说，使用接口实现比继承实现更加灵活、可扩展和可维护。接口有效地约束了实现行为，同时还解耦了具体的实现代码。

所以在 Java 中，当需要代码解耦、可扩展时，我确实更推荐使用接口而不是继承。接口隔离了变化的部分（不同的实现）, 非常适合面向对象设计中的 “对扩展开放，对修改关闭” 的设计原则。

但继承和接口并非完全对立，在实际开发中需要组合使用。继承用于对象行为的扩展，接口用于行为的抽象和规范，两者相辅相成，共同构建健壮、可维护的面向对象系统。

综合以上内容，使用继承或者使用接口实现多态并非完全等价。在实际开发中, 我们可以根据设计的需求，组合使用这两种方式来分别实现代码的可扩展性和规范统一。

14.4 用泛型实现多态

实现多态的另一种常见方法是使用泛型方法。尽管 Java 的泛型引入相对较迟，但泛型方法已经成为实现多态的一种常用方法。泛型方法是指在方法定义中使用泛型参数的方法。泛型方法可以根据传入的参数类型来确定方法的行为，从而实现多态性。

由于目前还未学习泛型，所以我将泛型方法的相关内容放在泛型一章中详细讲解。如果您对泛型方法感兴趣，请随时查看泛型那一章。

15 集合

Contents

在 Java 中，集合（Collection）是构建稳健且高效应用程序的基石之一。它扮演着至关重要的角色，用于有效地组织、管理和操作数据。Java 集合框架作为一个精心设计的框架，提供了一系列接口与实现类，确保程序员能够针对不同场景和需求选择合适的工具。

本章将深入剖析 Java 集合框架的核心内容，学习集合的强大功能和灵活应用。本章将从以下几个方面展开学习：集合的基本概念、Java 集合框架的架构体系、常见集合接口与类的详解、迭代器机制的理解与应用以及集合的实际应用案例与最佳实践。

如果您想阅读 JDK 源码，我强烈推荐您从集合框架的源码开始。本书前面已经涵盖了理解集合框架源码所需的大部分知识。通过阅读集合框架的源码，您可以更好地理解 Java 中引入的各种技术的作用。

阅读 JDK 5 之前的集合框架源码尤其有益，因为那时的代码更简洁直接。在 JDK 5 之前，Java 语言缺少泛型、枚举、自动装箱和拆箱以及增强 for 循环等特性。虽然这些特性在现代 Java 编程中非常有用，但它们也使代码更加复杂。JDK 5 之前的源码没有这些特性，因此更容易理解。

举例来说，下面的代码是 JDK 5 之前 ArralyList 的源码，由于源码太长了，我做过一些简化并添加了注释：

```
// JDK 1.4 中 ArrayList 的源码片段示例
public class ArrayList {
    private Object[] elements;

    public ArrayList() {
        elements = new Object[10];
    }

    public void add(Object element) {
        // 检查数组是否已满
        if (size == elements.length) {
            // 扩展数组
            Object[] newElements = new Object[
                                        elements.length * 2];
            System.arraycopy(elements, 0, newElements, 0,
                                            elements.length);
            elements = newElements;
        }

        // 添加元素
        elements[size++] = element;
    }

    public Object get(int index) {
        return elements[index];
    }
}
```

作为对比，Java 5 由于添加了泛型，难度有些增加，其代码片段如下所示：

```
// JDK 5 之后 ArrayList 的源码片段示例

public class ArrayList<E> {
    private static final int DEFAULT_CAPACITY = 10;
```

```
    private E[] elements;
    private int size;

    public ArrayList() {
        this(DEFAULT_CAPACITY);
    }

    public ArrayList(int initialCapacity) {
        if (initialCapacity < 0) {
            throw new IllegalArgumentException(
                    "Illegal Capacity: " + initialCapacity);
        }
        elements = (E[]) new Object[initialCapacity];
    }

    // 添加元素
    public void add(E element) {
        ensureCapacity(size + 1);
        elements[size++] = element;
    }

    public E get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException(
                    "Index: " + index + ", Size: " + size);
        }
        return elements[index];
    }

    // 确保 ArrayList 有足够的容量来存储指定数量的元素。
    // 如果当前容量不足，则会扩展数组。
    private void ensureCapacity(int minCapacity) {
        if (minCapacity > elements.length) {
            int newCapacity = elements.length * 2 + 1;
            if (newCapacity < minCapacity) {
                newCapacity = minCapacity;
```

```
                }
                elements = Arrays.copyOf(elements, newCapacity);
            }
        }
    }
```

JDK 5 之后的 ArrayList 代码更加安全、高效和易于使用，因为它引入了泛型支持和更合理的错误处理机制。通过阅读源码，您可以体会到这些改进。

阅读源码也没什么技巧，也不用什么特殊的工具，大多数 IDE（集成开发环境）都提供内置的源码阅读功能。例如，在 IntelliJ IDEA 中，可以按住 Ctrl 键并单击类名或方法名，以跳转到其源码。IDE 通常还提供代码高亮、语法提示和代码导航等功能，使阅读源码更加容易。

此外，在阅读源码时，您可能会发现有些类看似简单，但实际上却非常复杂。例如，java.lang.String 类就属于这种情况。String 类表示不可变字符串，并提供了许多用于操作字符串的方法。由于需要处理字符串编码、比较、搜索和操作等各种复杂问题，String 类的源码非常复杂。

另一方面，有些类看似复杂，但实际上却非常简单。例如，本章介绍的集合类就相对容易理解。集合类主要用于存储和操作对象集合，它们的源码通常集中于数组操作，例如添加、删除、获取和遍历元素。

说实话，本章非常简单，就从基本概念讲起吧。

15.1 集合的基本概念

在 Java 1.0 中，只包含了少数几个集合类，例如 Vector、Hashtable 和 Stack。这些类都是线程安全的，但它们的功能有限，而且性能也不高。在 Java 1.2 中，引入了新的集合框架，新的集合框架更加强大、灵活和高效。

15.1.1 什么是集合

在 Java 中，当我们说集合的时候，一般情况下会有两重含义，这两种含义要根据上下文环境进行区分：

一重含义是指 Java 集合框架接口 Collection：java.util.Collection 是 Java 集合框架的一个顶层接口，它定义了所有集合（如列表 List、集 Set、队列 Queue 等）共有的行为特征。所有实现 Collection 接口的类都必须支持添加元素、删除元素、判断是否为空、获取大小等基本操作。例如，ArrayList、LinkedList、HashSet 和 TreeSet 都是 Collection 接口的实现类。

第二重含义是指集合：在一般意义上，当我们提到 “collections” 时，指的是任何由一组对象组成的集合，它可以是 Java 集合框架中任何类型的集合，包括但不限于 List、Set、Map 等。另外，Java 中还有一个工具类叫做 java.util.Collections，它提供了大量静态方法，用于对集合进行排序、查询、替换元素、同步控制等各种操作。这个类并不直接实现 Collection 接口，但它服务于集合操作。

总结一下，“collection” 作为 Java 集合框架中的接口，描述了所有集合对象应该具有的通用行为；而 “collections”（复数形式）有时泛指集合本身，有时特指 java.util.Collections 这个工具类，包含了对集合进行操作的各种实用方法。只要结合上下文语境，我想并不难对其进行区分。

15.1.2 集合和数组的区别

在本书前面章节中，我们学习了数组这一 Java 中基础且重要的数据结构，了解到数组是一种线性数据结构，它能够以固定顺序存储相同类型的一系列元素，并通过索引快速访问每个元素。数组在内存中是连续分配的空间，这赋予了它高效的随机访问能力，但同时也限制了其大小的固定性以及对不同类型数据存储的局限性。

然而，在复杂的应用场景下，仅仅依赖数组往往不足以应对数据管理和组织的需求。这就引入了 Java 集合框架——一个强大且灵活的数据结构集合，它们旨在解决数组所面临的局限性，并提供了一套更为全面且便捷的操作方法。集合框架的核心组件如 List、Set 和 Map 等，各自拥有独特的特性，以满足多样化的数据处理要求。

相比于数组，集合有如下的几个优点：

动态调节大小：集合的大小可以根据需要动态扩展或缩小，这意味着在运行时可以根据数据量的变化来调整集合的容量，这一点大大优于数组预先设定且不可更改的大小限制。

存储多样性：集合仅能存储对象引用，但这并不妨碍它处理基本类型，因为基本类型可通过其对应的包装类间接存储。同时，集合能够容纳不同类型的对象（如使用无限制通配符? 或具体泛型，泛型会在本书后面章节详细讲解），而数组则要求所有元素严格一致。

类型安全与泛型：虽然数组需要声明元素类型，但集合通过泛型提供了更强的类型检查保障，确保放入集合的元素符合预设类型，降低运行时因类型错误引发的问题。

操作丰富性：集合提供了丰富的内置操作，包括排序、查找、过滤、映射、归约等，这些操作通过各种接口和实现类得以实现，极大地简化了程序员处理集合数据的工作。

数据结构多样性：集合框架不仅仅局限于线性存储，还包括了如链表、哈希表、树等数据结构的实现，每种数据结构都有其特定的性能特征，适应不同的应用环境。

15.2 Java 集合框架的架构体系

作为 Java 集合框架（Java Collections Framework）的基础接口之一，Collection 接口代表一个可进行元素添加、删除、检查和迭代操作的容器。它定义了一组通用的操作，适用于所有集合类型。为了便于理解和分析，我将绘制一张简化版的集合常用类与接口关系图。简化后的图表虽然省略了一些细节，但并不影响对整体结构的理解。

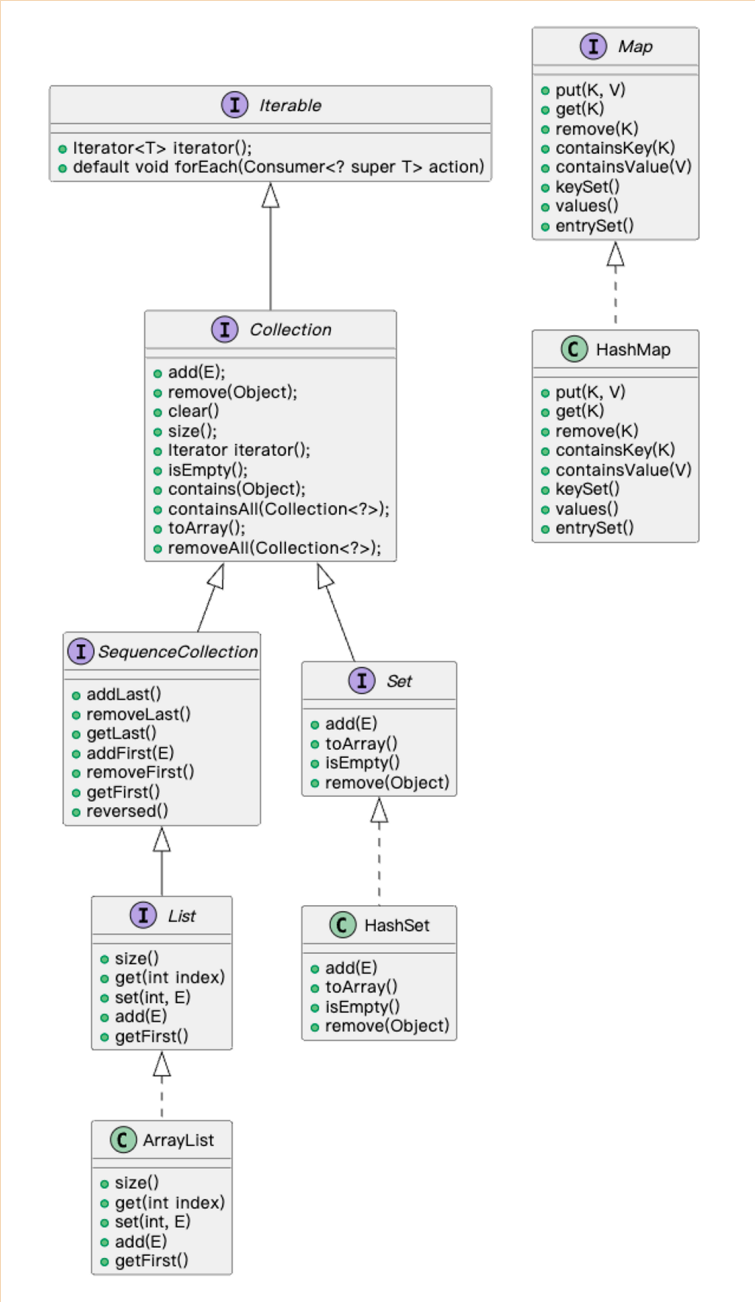

图 15.1: 集合中接口与类的继承关系图

上图是我根据 JDK 源码简化绘制的，省略了一些不常用的方法。细心的读者会发现，图中存在大量方法重复，例如 size() 方法在 ArrayList 类、List 接口和 Collection 接口中都存在。同一个方法为何要重复定义多次？我曾向许多人请教，也查阅过 StackOverflow 上的相关问题，发现不少人抱有同样的疑问，却始终没有得到明确的答案。

我尝试将接口中功能相同的方法名精简为一个，并编译了自定义的 JDK，程序运行正常。然而，我无法确定这种做法是否存在潜在的风险。若有读者对源码有更深入的理解，并能提供更合理的解释，我将洗耳恭听。这个问题困扰我多年，我逐渐倾向于 StackOverflow 上的一种解释：**这或许只是代码编写者当初设计疏忽所致**。

接下来，我们将深入探讨图中重要的类与接口。

15.3 核心接口与实现类

15.3.1 Iterable 接口

Iterable 接口的出现使得集合类的遍历操作更加简洁和统一。在 Java 5 之前，不同的集合类有不同的遍历方式，而 Iterable 接口的引入提供了一个通用的迭代机制。

先来讲一下 Iterable 这个接口，它在 Java 中扮演着重要的角色，表示一个可以被迭代的对象。Iterable 接口中有两个方法，一个是 forEach()，另一个是 iterator()。

o 实现了 iterator() 方法，意味着获取一个 Iterator 对象，该对象可以用来逐个访问集合中的元素，从而为对象提供迭代的能力。

o forEach() 方法是 Java 8 中引入的，它可以接受一个 Consumer 函数式接口作为参数，并对集合中的每个元素执行该函数，提供了一种更简洁的遍历方式。需要注意的是，forEach() 方法无法在遍历过程中修改集合元素。

许多常见的 Java 集合类，例如图中所示的 Collection 接口，都实现了 Iterable 接口。这意味着 Collection 接口的子类，例如 List、Set 和 Queue，都可以使用 for-each 循环或 forEach() 方法进行遍历。

15.3.2 Collection 接口

接下来，再来介绍 Java 集合的根接口之一：Collection。它定义了一组通用的方法，用于操作和管理一组对象，也称为元素。Collection 接口本身不提供任何具体的实现，而是由其他具体的集合类来实现。Collection 接口继承自 Iterable 接口，这意味着所有 Collection 的子类都可以使用 Iterable 接口定义的循环进行遍历。

Collection 接口的主要特点:

o 存储任意类型对象：Collection 可以存储任何类型的对象，包括自定义对象。

o 动态调节大小：Collection 的大小可以根据需要动态调整。

o 不保证顺序：Collection 接口本身不保证元素的存储顺序。如果需要保证顺序，可以使用其子接口 List。

o 允许重复元素: Collection 接口允许存储重复元素。如果需要确保元素唯一，可以使用其子接口 Set。

Collection 接口提供了一系列方法用于操作和管理集合元素，例如：

o add(E e)：将元素添加到集合中。

o remove(Object o)：从集合中移除元素。

o contains(Object o)：检查集合是否包含指定元素。

o size()：获取集合的大小。

o isEmpty()：检查集合是否为空。

15.3.3 List 接口以及实现类

List 接口是 Java 集合框架中 Collection 接口的子接口，它表示一个有序的集合，允许存储重复元素。List 接口扩展了 Collection 接口，添加了一些新的方法，例如：

o get(int index)：获取指定索引处的元素。

o set(int index, E element)：设置指定索引处的元素。

o add(int index, E element)：在指定索引处插入元素。

o remove(int index)：移除指定索引处的元素。

List 接口在 Java 中是一个非常重要的集合接口，它代表了一个有序的元素集合。与 Set 接口不同，List 接口允许存储重复的元素，并且可以通过元素的索引来访问元素。在 Java 中，List 接口有多个常见的实现类，其中最为常用的两个是 ArrayList 和 LinkedList。ArrayList 是 List 接口的一个动态数组实现。它的内部实现基于数组，但提供了动态调整大小的功能。当向 ArrayList 中添加元素时，如果当前数组空间不足，ArrayList 会自动创建一个新的更大的数组，并将原有数组的元素复制到新数组中。这种机制使得 ArrayList 在存储大量元素时仍然能够保持高效的性能。

ArrayList 的主要特点包括：

o 动态调整大小：可以根据需要自动增长和缩小容量。

o 元素访问速度快：由于基于数组实现，通过索引访问元素的速度非常快。

o 插入和删除操作相对较慢：在 ArrayList 的中间位置插入或删除元素时，需要移动大量元素，因此性能较差。

LinkedList 是 List 接口的另一个实现，它基于链表数据结构。与 ArrayList 不同，LinkedList 不需要在内存中连续存储元素，而是通过指针（或引用）将元素链接在一起。每个元素（通常称为节点）包含数据以及指向下一个和上一个元素的指针。

LinkedList 的主要特点包括：

o 插入和删除操作快：在 LinkedList 的中间位置插入或删除元素时，只需要修改相邻节点的指针，因此性能较好。o 元素访问速度相对较慢：由于元素在内存中不是连续存储的，通过索引访问元素时需要从头节点开始遍历链表，因此速度相对较慢。o 内存消耗较大：每个节点除了存储数据外，还需要存储指向前后节点的指针，因此相对于 ArrayList 来说，LinkedList 会占用更多的内存空间。

ArrayList 和 LinkedList 各有优缺点，适用于不同的场景。如果需要频繁地访问元素，并且元素数量较大，那么 ArrayList 可能是更好的选择。如果需要频繁地在列表中间插入或删除元素，那么 LinkedList 可能更合适。在实际开发中，应根据具体需求选择合适的 List 实现类。

理论讲完了，接下来，用实际的代码展示一下 List 的用法：

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        // 使用 ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("Apple");
        arrayList.add("Banana");
        arrayList.add("Cherry");
        System.out.println("ArrayList: " + arrayList);

        // 访问 ArrayList 中的元素
        System.out.println("Element at index 1: " +
                                        arrayList.get(1));

        // 使用 LinkedList
        List<String> linkedList = new LinkedList<>();
        linkedList.add("Orange");
```

```
        linkedList.addFirst("Lemon"); // 在链表开头插入元素
        linkedList.add("Pineapple");
        System.out.println("LinkedList: " + linkedList);

        // 访问 LinkedList 中的元素
        System.out.println("First element: " +
                                linkedList.getFirst());

        // 在 LinkedList 中间插入元素
        linkedList.add(2, "Grape");
        System.out.println("LinkedList after insertion: " +
                                            linkedList);

        // 删除 LinkedList 中的元素
        linkedList.remove("Lemon");
        System.out.println("LinkedList after deletion: " +
                                            linkedList);
    }
}
```

首先创建了一个 ArrayList 并向其中添加了几个元素。然后，打印出整个列表，并访问了索引为 1 的元素（即第二个元素，因为索引是从 0 开始的）。

接下来，创建了一个 LinkedList，并使用 addFirst 方法在链表的开头插入了一个元素。然后，打印出整个链表，并访问了链表中的第一个元素。

之后，在链表的第二个位置（索引为 2）插入了一个新元素，并打印出插入后的链表。

最后，从链表中删除了一个元素，并打印出删除后的链表。上面的程序运行之后，结果如下：

```
ArrayList: [Apple, Banana, Cherry]
Element at index 1: Banana
LinkedList: [Lemon, Orange, Pineapple]
First element: Lemon
LinkedList after insertion: [Lemon, Orange, Grape, Pineapple]
LinkedList after deletion: [Orange, Grape, Pineapple]
```

这个示例展示了 ArrayList 和 LinkedList 的一些基本用法，包括添加元素、访问元素、在特定位置插入元素以及删除元素。需要注意的是，在实际应用中，可能还需要处理其他

情况，例如检查列表是否为空、处理并发访问等。

15.3.4 Set 接口以及实现类

在 Java 中，Set 接口是 Java 集合框架的一部分，用于存储不重复的元素。它是 Collection 接口的一个子接口，因此它具有 Collection 接口的所有方法，并且增加了一些新的方法。

Set 的主要特性是它不包含重复的元素。更确切地说，如果试图向 Set 中添加一个已经存在的元素，那么 Set 不会改变，即新的 add 操作返回 false。

Set 接口有几个主要的实现类，包括 HashSet、LinkedHashSet、TreeSet 等。

HashSet 是 Set 接口的一个常用实现。它基于 HashMap 实现，因此它的元素存储不是有序的。HashSet 的添加、删除和查找操作的平均时间复杂度为 O(1)。然而，因为 HashMap 的工作原理是基于哈希表实现，它将键值对存储在一个数组中。当需要存储一个键值对时，HashMap 会根据键的哈希值计算出在数组中的位置，然后将该键值对存储在该位置上。当需要查询或删除一个键值对时，HashMap 也会根据键的哈希值找到数组中的相应位置，并在该位置上执行查找或删除操作。因此 HashSet 不能保证元素的迭代顺序与添加顺序相同。

LinkedHashSet 是 HashSet 的一个子类，它维护了一个运行于所有条目的双向链表。此链表定义了迭代顺序，即按照将元素插入到集合中的顺序（插入顺序）进行迭代，或者是按访问顺序进行迭代（当容量达到其初始容量并且尚未增加时）。注意，LinkedHashSet 的插入、删除和查找操作的平均时间复杂度仍然是 O(1)。

TreeSet 是 Set 接口的另一个实现，它使用树结构（通常是红黑树）来存储元素。这意味着 TreeSet 中的元素总是按照自然顺序或者创建 TreeSet 时提供的 Comparator 进行排序。因此，TreeSet 的添加、删除和查找操作的平均时间复杂度为 O(log n)。

总的来说，Set 接口和其实现类提供了强大的工具来存储和操作不重复的元素集合。选择哪种实现取决于你的具体需求，例如是否需要排序、是否需要保持元素的插入顺序等。

理论讲完了，接下来，用代码来验证一下：

```
// HashSet、LinkedHashSet 和 TreeSet 的演示代码
import java.util.*;

public class SetDemo {
    public static void main(String[] args) {
        // 创建 HashSet 实例
        Set<String> hashSet = new HashSet<>();
        hashSet.add("Apple");
        hashSet.add("Banana");
```

```
        hashSet.add("Cherry");
        // 尝试添加重复元素，不会添加成功
        hashSet.add("Apple");
        // HashSet 内部是通过哈希表实现的，
        // 因此添加元素的顺序是不固定的。
        System.out.println("HashSet: " + hashSet);

        // 创建 LinkedHashSet 实例
        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("Grape");
        linkedHashSet.add("Orange");
        linkedHashSet.add("Pear");
        // 尝试添加重复元素，不会添加成功
        linkedHashSet.add("Grape");
        // LinkedHashSet 集合像 HashSet 一样不允许存储重复元素，
        // 但它还额外通过链表结构来保存元素的插入顺序。
        // 因此，元素将按照它们被插入到集合中的先后顺序依次返回。
        System.out.println("LinkedHashSet（插入顺序）: "
                                        + linkedHashSet);

        // 创建 TreeSet 实例，
        // 并提供一个 Comparator 以按字母顺序排序

        Set<String> treeSet =
              new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
        treeSet.add("Mango");
        treeSet.add("Watermelon");
        treeSet.add("Pineapple");
        treeSet.add("Mango"); // 尝试添加重复元素，不会添加成功
        // 在 Java 中使用 TreeSet 时，
        // 集合内的元素会按照某种排序规则进行组织。
        // 由于 TreeSet 是基于红黑树（Red-Black Tree）实现的，
        // 它能够自动对添加的元素进行排序，要么依据元素自身的自然排序
        // 要么依据在创建 TreeSet 时提供的 Comparator 进行排序。

        System.out.println("TreeSet（按字母排序顺序）: "
```

```
                                                + treeSet);

        // 显示 LinkedHashSet 按照插入顺序遍历
        System.out.println("\nLinkedHashSet 按插入顺序遍历:");
        for (String fruit : linkedHashSet) {
            System.out.print(fruit + " ");
        }

        // 显示 TreeSet 按照排序顺序遍历
        System.out.println("\nTreeSet 按排序顺序遍历:");
        for (String fruit : treeSet) {
            System.out.print(fruit + " ");
        }
    }
}
```

在这个例子中，HashSet 不保证元素的顺序，添加和删除速度较快。

LinkedHashSet 保留了元素的插入顺序，所以遍历时输出的是按照插入顺序排列的元素。

TreeSet 则自动对元素进行了排序，这里的排序是通过传入的 String.CASE_INSENSITIVE_ORDER 比较器实现的，因此即便添加元素的顺序不同，最终遍历输出的结果是按字母顺序排列且不区分大小写的。

这些概念都属于数据结构的范畴，对于学习过数据结构的读者来说应该并不陌生。我在源码注释中详细解释了其运行原理，此处不再赘述。上面的代码运行之后，结果如下：

```
HashSet: [Apple, Cherry, Banana]
LinkedHashSet（插入顺序）: [Grape, Orange, Pear]
TreeSet（按字母排序顺序）: [Mango, Pineapple, Watermelon]

LinkedHashSet按插入顺序遍历:
Grape Orange Pear
TreeSet按排序顺序遍历:
Mango Pineapple Watermelon
```

15.3.5 Map 接口以及实现类

尽管从前面的类图中可以看出 Map 接口和 Collection 接口没有直接的继承关系，但它们在功能上却有着诸多相似之处，因此经常被相提并论。例如，两者都用于存储和管理数据集合，只是 Collection 接口存储单值元素，而 Map 接口存储键值对。在实际应用中，Map 和 Collection 经常协同使用。例如，可以使用 Map 来存储用户信息，其中键是用户名，值是用户信息对象。而用户信息对象本身可能包含多个属性，此时便可借助 Collection 来存储这些属性。

Map 是一个非常重要的接口，在 Map 中，我们不能存储重复的键（key），每个键只能映射一个值（value）。Map 接口的主要方法有：

o put(K key, V value): 将指定的值与此映射中的指定键关联（可选操作）。

o get(Object key): 返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。

o remove(Object key): 从此映射中移除一个键的映射关系（如果该键存在）（可选操作）。

o containsKey(Object key): 如果此映射包含指定键的映射关系，则返回 true。

o containsValue(Object value): 如果此映射将一个或多个键映射到指定值，则返回 true。

o size(): 返回此映射中的键-值映射关系数。

o isEmpty(): 如果此映射不包含键-值映射关系，则返回 true。

Map 接口有几个常用的实现类，包括 HashMap、LinkedHashMap、TreeMap 和 Hashtable 等。

HashMap：HashMap 是基于哈希表的 Map 接口实现。它允许使用 null 键和 null 值，并且不保证映射的顺序，特别是它不保证该顺序恒久不变。HashMap 是线程不安全的，如果需要线程安全，可以使用 Collections.synchronizedMap() 方法或者使用 ConcurrentHashMap。

LinkedHashMap：LinkedHashMap 是 HashMap 的一个子类，它维护了一个运行于所有条目的双向链表。此链表定义了迭代顺序，即按照将元素插入到映射中的顺序（插入顺序）或访问顺序对映射进行迭代。

TreeMap：TreeMap 是基于红黑树（一种自平衡的排序二叉树）的 NavigableMap 实现。TreeMap 中的元素按照键的自然顺序进行排序，或者根据创建 TreeMap 时提供的 Comparator 进行排序，具体取决于使用的构造方法。

Hashtable：Hashtable 是较旧的实现，它的功能和 HashMap 类似，但它是线程安全的。然而，由于 Hashtable 的同步性，它的性能相对较差。如果需要在多线程环境中使用 Map，建议使用 ConcurrentHashMap。

在选择使用哪个 Map 实现时，应根据你的具体需求来决定。例如，如果需要保持键值对的插入顺序，那么应该使用 LinkedHashMap；如果需要按照某种特定顺序对键值对进行排序，那么应该使用 TreeMap；如果只需要一个基本的键值对映射，并且不需要保持特定的顺序，那么 HashMap 可能是一个好的选择。同时，如果需要在多线程环境中使用 Map，那么应该选择线程安全的实现，如 ConcurrentHashMap。

接下来，演示一下 HashMap、LinkedHashMap 和 TreeMap 的用法

```
import java.util.*;

public class MapDemo {

    public static void main(String[] args) {
        // 使用 HashMap
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Apple", 1);
        hashMap.put("Banana", 2);
        hashMap.put("Cherry", 3);
        System.out.println("HashMap (unordered): "
                                            + hashMap);

        // 使用 TreeMap，键按照自然顺序排序
        Map<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Banana", 2);
        treeMap.put("Apple", 1);
        treeMap.put("Cherry", 3);
        System.out.println("TreeMap (sorted by key): "
                                            + treeMap);

        // 使用 LinkedHashMap，键值对按照插入顺序存储
        Map<String, Integer> linkedHashMap =
                                    new LinkedHashMap<>();
        linkedHashMap.put("Apple", 1);
        linkedHashMap.put("Banana", 2);
        linkedHashMap.put("Cherry", 3);
        System.out.println("LinkedHashMap
                    (insertion order): "  + linkedHashMap);
```

```
        // 更新和获取元素
        linkedHashMap.put("Apple", 4); // 更新已存在的键对应的值
        System.out.println("Value of 'Apple' after update: "
                           + linkedHashMap.get("Apple"));

        // 遍历 Map
        System.out.println("\nTraversal:");
        System.out.println("HashMap keys in iteration
                             order: " + hashMap.keySet());
        System.out.println("TreeMap keys in sorted
                             order: " + treeMap.keySet());
        System.out.println("LinkedHashMap keys in
              insertion order: " + linkedHashMap.keySet());

        // 删除元素
        hashMap.remove("Banana");
        System.out.println("After removing 'Banana': "
                                                + hashMap);
    }
}
```

在上面的代码中，HashMap 是一个基于哈希表实现的无序映射关系，插入和查找效率高，但不保证元素的顺序。

TreeMap 会根据键的自然顺序（对于字符串，就是字典顺序）或自定义的比较器对键进行排序，插入和查找的时间复杂度为 O(log n)。

LinkedHashMap 继承自 HashMap，除了具备 HashMap 的特性外，还维护了一个双链表，从而可以按照插入顺序遍历键值对，也可以选择 LRU（最近最少使用）或其他策略进行排序。

代码运行之后的结果如下：

```
HashMap (unordered): {Apple=1, Cherry=3, Banana=2}
TreeMap (sorted by key): {Apple=1, Banana=2, Cherry=3}
LinkedHashMap (insertion order): {Apple=1, Banana=2, Cherry=3}
Value of 'Apple' after update: 4

Traversal:
HashMap keys in iteration order: [Apple, Cherry, Banana]
```

```
TreeMap keys in sorted order: [Apple, Banana, Cherry]
LinkedHashMap keys in insertion order: [Apple, Banana, Cherry]
After removing 'Banana': {Apple=1, Cherry=3}
```

15.4 本章附录

15.4.1 ArrayList 的内部实现原理

ArrayList 使用的并非我们常见的普通数组，而是一种经过优化的特殊数组——动态数组。动态数组的魅力在于它能够在程序运行时灵活地调整容量，无论是扩容还是缩容，都能根据需求自动完成。这种特性使得动态数组无需在初始化时预先设定固定大小，而可以在运行时动态分配和管理内存空间，极大地提高了内存利用率和程序的灵活性。

虽然听起来似乎高深莫测，但实际上，深入探究 ArrayList 的源代码，会发现其背后的实现机制并不神秘。接下来，我们将详细剖析动态数组的工作原理，揭开它神秘的面纱。

“源码面前，了无秘密” 是许多程序员信奉的箴言。诚然，如果有充足的时间，这句话的确不假。然而，现代开源软件的代码量动辄百万行起步，例如 Java 的 JDK 15，仅类库和接口的源代码就超过 600 万行，更不用说 JVM 的源码。想要通读所有代码几乎是不可能的任务。

但如果我们只针对特定感兴趣的特性，查阅源代码仍然是快速解惑的最佳途径。鉴于前文已经阐述了数组的用法，阅读 ArrayList 的源代码应该不会存在太多障碍。

接下来，我们将一起探索 ArrayList 的源代码，揭示动态数组的奥秘。

简而言之，动态数组本质上就是一个普通数组，只是增加了一些智能化的机制，使其能够根据需要自动调整容量，从而更加灵活地管理内存空间。

当创建一个默认的 ArrayList 时，它会以一个空数组作为起点。首次添加元素时，ArrayList 会增长到一个初始容量（capacity），默认值为 10。需要注意的是，初始容量与 ArrayList 的大小（size）并非同一概念，如下图所示：

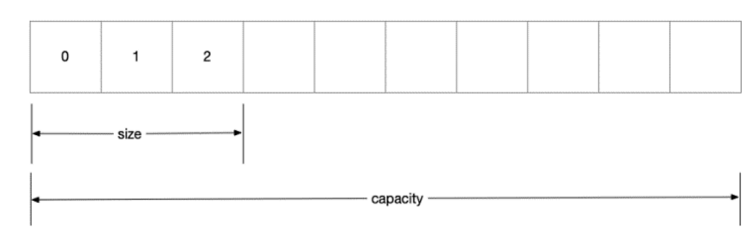

图 15.2: ArrayList 的大小与底层动态数组的容量的区别

在初始化 ArrayList 时，可以传入一个参数来指定其大小。若未指定大小，则会调用一

个名为 DEFAULT_CAPACITY 的整型变量，其值为 10，用于初始化数组。相关源代码如下：

```
/**
 * Default initial capacity.
 */
private static final int DEFAULT_CAPACITY = 10;
```

每当尝试添加元素时，若 ArrayList 已满，它会自动进行扩容，以确保有足够的空间容纳新元素。在 Oracle 的 Java 实现中，ArrayList 扩容时会生成一个新的数组，其容量为旧数组容量的 1.5 倍（取整）。扩容后的容量由扩容前的容量决定，相关源代码如下：

```
int oldCapacity = elementData.length;
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

上面第 2 行代码中的 » 操作符是位运算符，表示除以 2，在本例中，原先的数组容量是 10，扩容后的数组容量是 15。然后，它会将旧数组的所有元素复制到新数组中。其原理如下图所示：

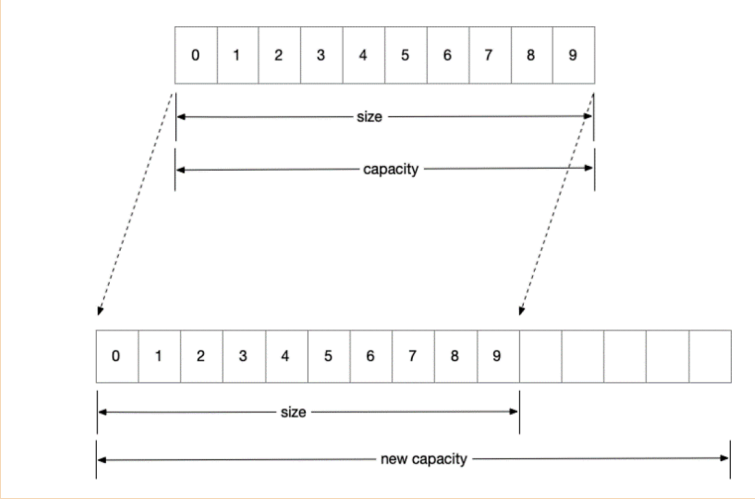

图 15.3: ArrayList 自动扩容的原理

这种自动扩容策略使得 ArrayList 能够在运行时动态调整容量，以适应程序的需求。然而，自动扩容也带来了一些性能损耗，包括创建新数组和复制旧数组元素的开销。因此，如果你预知将在 ArrayList 中存储大量元素，在创建时指定一个足够大的初始容量会更加高效。

了解了 ArrayList 自动扩容的原理，您可能会好奇：当 ArrayList 删除大量元素时，它会缩减容量吗？按常理推断，答案应该是肯定的。然而，ArrayList 并不主动缩减容量，这是因为缩减容量需要新建数组并进行元素复制，会对系统造成一定的压力。

不过，程序员可以手动调用 trimToSize() 方法来缩减 ArrayList 的容量。我认为 ArrayList 不主动缩减容量的设计是合理的，因为在现实世界中，无论是人口还是物品数量，总体趋势都是不断增长，即使存在波动，最终也会达到新的峰值。如果程序员确定 ArrayList 的元素数量已达极限，手动调用 trimToSize() 方法即可。

16 错误处理

Contents

在编程过程中，异常情况的发生是无法完全避免的。无论代码多么完备，都有可能遇到非预期的状态或不可控的外部因素，导致程序行为偏离正常执行路径。为了增强代码的健壮性和容错性，Java 为我们提供了异常处理机制。

异常处理是 Java 语言中非常重要的一个特性，它使得我们能够在运行时捕获和处理各种异常情况，避免程序异常终止，同时采取合理的恢复措施或返回友好的错误信息。通过合理使用异常处理，可以提高程序的可靠性、可维护性和用户体验。

异常本质上是一种对象，代表了某种特殊的、非正常的情况发生。Java 将异常划分为两大类：受检异常（checked exception）和非受检异常（unchecked exception），它们在抛出和捕获时有所不同。无论是受检还是非受检异常，Java 都提供了统一的处理方式，即 try-catch-finally 代码块。

通过本章的学习，你将全面了解 Java 异常处理机制的方方面面，包括：异常的基本概

念、异常捕获处理、异常抛出、自定义异常等。掌握了这些知识，你就能够自如地在代码中处理各种异常情况，编写出更加健壮、高质量的 Java 程序。

16.1 异常基础概念

16.1.1 什么是异常

在 Java 中，异常（Exception）代表程序执行期间遇到的非正常情况，它们是可以捕获和处理的事件。当异常发生时，Java 虚拟机（JVM）会自动生成一个异常对象并抛出。未经处理的异常将导致程序终止运行，因此合理的异常处理机制对于确保程序的稳定性和可靠性至关重要。

异常处理的作用和意义在于：它提升了程序的健壮性和容错能力，防止了因错误而导致的非法程序终止；它允许我们对不同的异常情况进行清晰的区分，从而便于进行系统性的处理和调试；通过将错误处理代码与业务逻辑代码分离，异常机制增强了代码的清晰度和可维护性；并且，异常能够在调用栈中传递，为错误的集中式处理提供了便利。

16.1.2 异常的类型和层次结构

在 Java 中，异常架构构建在 Throwable 基类之上，分为两个主要子类：Error 和 Exception。Error 类表示严重的系统级错误，例如 JVM 崩溃和资源耗尽，这些问题大多是不可恢复且不应被应用程序捕获的。而 Exception 类是程序中可处理事件的表示，分为受检异常（Checked Exception）和非受检异常（Unchecked Exception）。

受检异常是编译器强制要求程序员处理的异常。如果一个方法声明抛出受检异常，那么调用该方法的代码必须使用 try…catch 块捕获该异常，或者将该异常声明为抛出。例如，IOException 和 SQLException 是受检异常。受检异常的设计目的是鼓励程序员编写更健壮的代码。通过强制程序员处理受检异常，编译器可以帮助程序员避免一些常见的错误，例如忘记关闭文件或处理数据库连接失败。

非受检异常是编译器不会强制要求程序员处理的异常。如果一个方法声明抛出非受检异常，那么调用该方法的代码可以选择捕获该异常，也可以选择不捕获。例如，NullPointerException 和 ArrayIndexOutOfBoundsException 是非受检异常。非受检异常通常表示程序中的错误，例如空指针异常或数组越界异常。这些错误通常是由于程序员的疏忽造成的，编译器无法帮助程序员避免这些错误。

那么，何时使用受检异常和非受检异常呢？

一般来说，如果一个异常是程序员可以预见并处理的，那么应该使用受检异常。例如，如果一个方法打开一个文件，那么该方法应该声明抛出 FileNotFoundException，因为文件可能不存在。

如果一个异常是程序员无法预见或处理的，那么应该使用非受检异常。例如除以零时，Java 虚拟机会抛出此异常。因为它是 RuntimeException‘的子类，所以它是一个 unchecked exception。

接下来，画一张 Java 中的异常处理的类图。

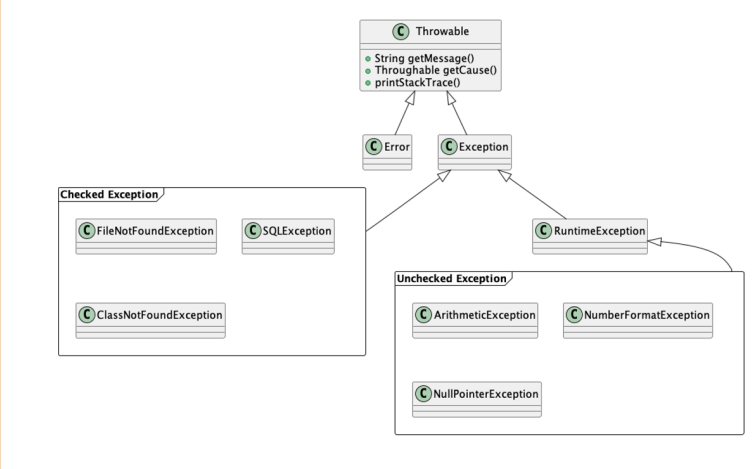

图 16.1: 异常的层次结构

上面是一张关于 Java 异常处理的类图。它展示了 Java 中不同类型的异常以及它们之间的继承关系。在 Java 编程语言中，所有异常都从 Throwable 类派生而来。Throwable 有两个主要子类：Error 和 Exception。Error 表示应用程序无法恢复的严重错误，例如系统崩溃或内存不足。Exception 则用于程序运行时发生的可恢复问题，如文件未找到、网络连接失败等。

Exception 又分为两种类型：Checked Exception 和 Unchecked Exception。Checked Exception 是指编译器强制检查的异常，即程序员必须显式地声明或者捕获这些异常。常见的 Checked Exception 包括 FileNotFoundException（文件未找到）和 SQLException（数据库操作异常）。Unchecked Exception 是在运行时抛出的异常，不需要在编译时进行显式声明。常见的 Unchecked Exception 有 ArithmeticException（算术运算异常）、NullPointerException（空指针异常）和 ArrayIndexOutOfBoundsException（数组下标越界异常）。

通过理解这张图，开发者可以更好地理解和处理 Java 中的各种异常情况。

16.1.3 Error 和 Exception 的区别

虽然 Error 和 Exception 都继承自 Throwable，但它们的用途和处理方式大不相同。Error 通常指示不可逆的系统级问题，诸如 OutOfMemoryError 或 StackOverflowError，而 Exception 则表示可捕获且应由开发者处理的条件。

Error 类本身拥有一些主要子类：VirtualMachineError 代表与虚拟机有关的问题，

AWTError 涉及到 Java 的高级窗口工具包图形界面错误，而 ThreadDeath 是线程结束时抛出的错误。与此相对的是 Exception，它有着更多细分的子类，用以区分和处理各种不同类型的可恢复异常。

一般来说，程序员不需要处理 Error 类。Error 类表示严重的错误，通常无法恢复，例如 OutOfMemoryError 或 StackOverflowError。

当 Error 发生时，通常意味着程序无法继续运行，因此捕获并处理 Error 通常没有意义。相反，程序员应该专注于防止 Error 的发生。例如，程序员可以通过优化代码来减少内存使用，从而防止 OutOfMemoryError 的发生。

当然，在某些情况下，程序员可能需要捕获并处理 Error。例如，如果程序员正在编写一个关键任务应用程序，那么他们可能需要捕获 Error 并记录错误信息，以便在程序崩溃后进行调试。

但是，在大多数情况下，程序员应该让 Error 传播并终止程序。

在编写应用程序时，开发者应该集中于捕获和处理 Exception 类及其子类。Exception 类表示可以恢复的异常，例如 IOException 或 SQLException。通过捕获和处理这些异常，开发者可以使程序更加健壮和可靠。接下来，我们一起学习如何捕获并处理异常。

16.2 异常捕获处理

在 Java 中，异常处理是保证程序健壮性和可靠性的重要机制。Java 提供了两种机制来处理异常：throws 语句和 try-catch-finally 块。

throws 语句用于声明一个方法可能抛出的异常。当一个方法调用另一个方法，而被调用的方法声明抛出受检异常时，调用方法必须使用 try…catch 块捕获异常，或者将异常声明为抛出。

try-catch-finally 块用于捕获和处理异常。try 块包含可能抛出异常的代码，catch 块捕获并处理异常，finally 块无论是否发生异常都会执行。

接下来，我们将分别介绍 throws 语句和 try-catch-finally 块。

16.2.1 throws 语句

在 Java 编程语言中，throws 是一个关键字，用于在方法签名中声明该方法可能会抛出的受检查异常。受检查异常是指那些继承自 java.lang.Exception 类但不包括 java.lang.RuntimeException 及其子类的异常类型。这类异常在编译时期要求程序员显式地处理，也就是说，如果一个方法可能会抛出受检查异常，除非在方法内部捕获它（使用 try-catch 块），否则必须在方法签名中使用 throws 关键字声明出来。

下面是一个 throws 关键字使用的示例：

```
public void readFromFile(String filePath)
                throws FileNotFoundException {
    File file = new File(filePath);
    FileReader fr = new FileReader(file);
    // ... 进行文件读取操作 ...
}
```

在这个例子中，readFromFile 方法在尝试读取指定路径下的文件时，如果文件不存在，则构造 FileReader 对象时会抛出 FileNotFoundException。由于这是个受检查异常，所以方法声明中包含 throws FileNotFoundException，告诉调用者这个方法在执行过程中可能会抛出这样的异常，并且调用者必须处理这个异常，要么在其调用的地方使用 try-catch 捕获，要么在调用者的签名中同样使用 throws 关键字声明并继续传递异常处理责任。

需要注意的是，throws 不是用于主动抛出异常的语句，而是用于声明方法与异常之间的关系。实际抛出异常的动作是由 throw 关键字完成的。同时，throws 也可以声明多个可能抛出的异常，只需用逗号分隔各个异常类型即可。而非受检查异常范畴覆盖的运行时异常（如 NullPointerException 或 IllegalArgumentException 等继承自 RuntimeException 的异常），则无需在方法签名中使用 throws 关键字声明。

throws 语句用于声明一个方法可能抛出的异常，但它并不能处理异常。为了捕获和处理异常，我们需要使用 try-catch-finally 语句。接下来，我们一起学习这部分内容。

16.2.2 try-catch-finally 语句

在 Java 中，可以使用 try-catch-finally 语句块来捕获异常。try 语句块用于指定要执行的代码，catch 语句块用于指定在发生异常时要执行的代码，finally 语句块中的代码“通常”会被执行，但“并非”总会被执行，它通常用于确保一些必要的清理工作总能完成，比如关闭数据库连接、网络连接或者释放系统资源。其语法块的结构如下：

```
try {
    // 可能会引发异常的代码块
} catch (ExceptionType1 e1) {
    // 处理 ExceptionType1 类型的异常
} catch (ExceptionType2 e2) {
    // 处理 ExceptionType2 类型的异常
} finally {
    // 可选的 finally 块，在无论是否捕获到异常，通常都会被执行
}
```

接下来，用上面的语法结构，写一个完整的 Java 程序示例，演示如何捕获和处理除数为零的异常，演示代码如下所示：

```
package com.liuyandong.excetion;

public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int a = 10;
            int b = 0;
            int c = a / b;
            System.out.println(c);
        } catch (ArithmeticException e) {
            System.out.println(" 除数不能为 0");
        } catch (Exception e) {
            System.out.println(" 出错了");
        } finally {
            System.out.println(" 不管如何，最后都会执行");
        }
    }
}
```

当运行上述程序时，由于尝试进行除以零的操作，因此会触发 ArithmeticException，并且控制权会转移到对应的 catch 块中，打印出相应的错误信息，然后执行 finally 块中的代码，最后继续执行主方法剩余的部分。运行之后输出的结果如下：

```
除数不能为0
不管如何，最后都会执行
```

前面的代码中，有两个 catch 语句，意义非常明确，就是用来捕获不同类型的异常。接下来，来讲一下多重 catch。

16.2.2.1 多重 catch

在 Java 中，多重 catch 指的是在一个 try-catch 结构中使用多个 catch 子句来捕获不同类型的异常。这样做的原因是，在一段代码块中可能会抛出多种不同的异常，而每种异常可能需要不同的处理方式。通过多重 catch，我们可以针对性地为每种异常类型编写不同的处理逻辑。

catch 语句块的顺序很重要，因为第一个匹配的 catch 语句块将被执行。因此，应该将

更具体的 catch 语句块放在前面，更通用的 catch 语句块放在后面。如果前面的语句写成了下面这种情况就是错误的：

```
// 下面这段代码是错误的示例
try {
    int a = 10;
    int b = 0;
    int c = a / b;
    System.out.println(c);
} catch (Exception e) {
    System.out.println(" 出错了");
} catch (ArithmeticException e) {
    System.out.println(" 除数不能为 0");
} finally {
    System.out.println(" 不管如何，最后都会执行");
}
```

上面的代码错误的原因如下：Exception 是 ArithmeticException 的父类，所以如果 catch 语句块的顺序是错误的，则即使发生了 ArithmeticException 异常，也会被 Exception 语句块捕获，从而导致无法输出更具体的错误信息。正确的做法是：应该将更具体的 catch 语句块放在前面，更通用的 catch 语句块放在后面。

Java 7 引入了一种更加简洁的语法，允许将多个异常类型放在一个 catch 块中。这种语法更清晰和紧凑。其语法如下：

```
try {
    // 可能会引发异常的代码块
} catch (ExceptionType1 | ExceptionType2
                        | ExceptionType3 e) {
    // 处理 ExceptionType1
    // 或 ExceptionType2
    // 或 ExceptionType3 类型的异常
}
```

接下来，演示 Java 7 引入的处理异常的新语法：

```
public class MultiCatch {
    public static void main(String[] args) {
```

```
        try {
            int result = divide(10, 0);
            System.out.println(" 运算结果: " + result);
        } catch (ArithmeticException
                | NullPointerException e) {
            System.out.println(" 出现错误: "
                               + e.getMessage());
        }
    }

    public static int divide(int a, int b) {
        return a/b;
    }
}
```

在上面的示例中，我们试图将 10 除以 0，这会引发一个 ArithmeticException 异常。我们还捕获了 NullPointerException 异常。在 catch 块中，我们使用 “|” 操作符将这两种异常类型组合在一起，并使用单个 catch 块来处理它们。

16.2.2.2 finally 块的使用场景

根据上面的内容，我们可以得出这样的结论，当程序运行的时候，只会有两种情况，运行成功或者运行失败。在运行成功的时候，也就是正常执行了 try 语句，此时 finally 语句肯定会执行。但是当程序运行失败的时候，就要考虑到底是哪种失败了。

一种是成功的失败，比如抛出异常并被 catch 语句块捕获，或者抛出异常但是没有被 catch 语句块捕获。在这两种成功的失败下，finally 语句也会执行。所以在这三种情况，一种 “成功的成功” 外加两种 “成功的失败” 下，finally 语句都会执行，其使用场景主要是以下几种情况：

资源管理：如打开文件、数据库连接或网络套接字等，不论是否发生异常，都需要确保在结束相关操作后能够及时关闭资源。比如，在打开文件后，要记得把文件关闭，代码如下：

```
try (BufferedReader reader = new BufferedReader(
                    new FileReader("file.txt"))) {
    // 使用 reader 读取文件
  } catch (IOException e) {
    // 处理 I/O 异常
```

```
    } finally {
      // 关闭 reader
      reader.close();
    }
  }
```

事务回滚：在数据库操作中，finally 块用于确保在任何情况下都能提交或回滚事务。演示代码如下：

```
      // 执行 SQL 语句
      try (Statement statement =
                    connection.createStatement()) {
          statement.executeUpdate("INSERT INTO users
                    (name, age) VALUES ('John Doe', 30)");
        } catch (SQLException e) {
          e.printStackTrace();
          // 回滚事务
          connection.rollback();
        } finally {
          // 关闭数据库连接
          connection.close();
        }
```

释放锁：在多线程环境下，finally 块用于确保线程安全操作结束后释放锁定资源。演示代码如下：

```
try {
    // 执行线程安全操作
    System.out.println(" 线程安全操作正在执行...");
  } finally {
    // 释放锁
    lock.unlock();
  }
```

以上是 finally 块常用的使用场景，finally 块通常会在任何情况下都会执行，但也有例外情况。在编写代码时，需要考虑这些例外情况，并采取相应的措施。最常见的情况是在 try 或者 catch 语句中调用了 System.exit()，此时 JVM 会立即停止运行程序，因此 finally 块将不会被执行。

16.3 Java 内置的常见异常类型

在 Java 中，异常是程序执行期间出现的问题，会打断正常的程序流程。Java 提供了多种内置的异常类型，以便更好地处理各种可能出现的问题。

当程序出现异常时，了解异常类型可以帮助我们更快地定位问题所在。通过查看异常堆栈跟踪信息，我们可以知道是哪个方法抛出了异常，以及异常发生的具体位置，从而快速地进行调试和修复。以下是一些 Java 中常见的异常类型：

16.3.1 非受检异常（Unckecked Exception）

这些异常通常在 Java 虚拟机正常运行期间抛出，表示 Java 虚拟机中出现了问题，从而可能会抛出一些异常。这些问题往往是程序错误的结果，比如逻辑错误或者不恰当的 API 使用。运行时异常通常不需要在代码中显式捕获，因为它们是编程错误，我们应该通过改进代码来避免这些错误。

o NullPointerException：当应用程序试图在需要对象的地方使用 null 时抛出。

o ArrayIndexOutOfBoundsException：当应用程序试图访问数组的非法索引时抛出。

o ClassCastException：当应用程序试图将一个对象强制转换为不是实例的子类时抛出。

o ArithmeticException：当出现异常的运算条件时抛出，例如除以零。

o IllegalArgumentException：当向方法传递非法或不适当参数时抛出。

o IllegalStateException：当在对象的不适当状态下调用方法时抛出。

16.3.2 受检异常（Checked Exceptions）

受检异常在编译时期就会被检查。如果代码中可能会抛出检查型异常，但没有使用 try-catch 块来捕获它，或者没有使用 throws 关键字来声明它，那么编译器会报错。检查型异常通常是由外部因素（如用户输入或文件 I/O）导致的问题，我们可以通过适当的错误处理来解决这些问题。

以下是一些常见的检查型异常：

o IOException：当发生输入/输出错误时抛出，例如文件读写错误或网络连接问题。

o SQLException：当使用 JDBC 进行数据库操作时，如果发生 SQL 错误或访问问题，将抛出此异常。

o ClassNotFoundException：当应用程序试图通过字符串名称加载类，但找不到具有指定名称的类定义时抛出。

o InterruptedException：当线程在等待、睡眠或占用时，另一个线程中断它时抛出。

16.3.3 错误（Error）

Error 类是 Java 所有错误类的超类。它们通常是 Java 虚拟机报告的问题，表示合理的应用程序不应该试图捕获的严重问题。大多数这样的错误都是异常条件，这是合理的应用程序不应该尝试处理的。例如，OutOfMemoryError 表示 JVM 没有足够的内存来继续执行程序。

处理异常是 Java 编程中非常重要的一部分，它可以帮助我们编写更健壮、更可靠的代码。通过合理地使用 try-catch 块和 throws 关键字，我们可以捕获并处理可能发生的异常，从而确保程序的稳定运行。

在 Java 编程中，除了内置的异常类型，我们还可以创建自定义的异常类型。自定义异常可以让我们更精确地表示程序中可能出现的特殊错误情况，使得代码更具可读性和可维护性。当内置的异常类型无法满足我们的需求时，或者我们想要提供更多的错误信息时，就可以考虑创建自定义异常。接下来，将介绍如何在 Java 中创建和使用自定义异常。

16.4 自定义异常

在 Java 中，自定义异常是指程序员根据需要创建的异常类，这些类通常继承自 Exception 类或其子类。自定义异常允许我们在遇到特定问题时抛出自定义的异常，从而提供更精确的错误信息，并使得错误处理更加灵活和有针对性。

要创建自定义异常，需要遵循以下步骤：

1. 继承自 Exception 或其子类：自定义异常类需要直接或间接地继承自 java.lang.Exception 或其子类，如 RuntimeException。
2. 提供构造方法：自定义异常类通常包含一个或多个构造方法，这些构造方法接受一个或多个字符串参数，用于设置异常消息。
3. 使用自定义异常：在代码中，当遇到特定错误情况时，可以使用 throw 关键字抛出自定义异常。

接下来，看看自定义的异常类：

```
package com.liuyandong.customexception;
public class MyCustomException extends Exception {
    // 默认的构造方法
    public MyCustomException() {
        super();
    }
```

```
    // 带有异常消息的构造方法
    public MyCustomException(String message) {
        super(message);
    }

    // 带有异常消息和原因的构造方法
    public MyCustomException(String message,
                             Throwable cause) {
        super(message, cause);
    }

    // 带有原因的构造方法
    public MyCustomException(Throwable cause) {
        super(cause);
    }
}
```

上面这段代码定义了一个名为 MyCustomException 的自定义异常类，它继承自 Java 的 Exception 类。这个自定义异常类提供了四个构造方法，分别用于不同的场景：

1. 默认的构造方法：当不需要提供任何额外信息时，可以使用这个构造方法创建异常对象。
2. 带有异常消息的构造方法：当需要提供关于异常的详细信息时，可以使用这个构造方法。这个信息可以通过 getMessage() 方法获取。
3. 带有异常消息和原因的构造方法：当需要提供关于异常的详细信息，并且这个异常是由另一个异常引起的，可以使用这个构造方法。这个信息可以通过 getMessage() 方法获取，原因可以通过 getCause() 方法获取。
4. 带有原因的构造方法：当需要提供这个异常的原因，但不需要提供详细信息时，可以使用这个构造方法。原因可以通过 getCause() 方法获取。

值得注意的是，自定义异常类通常需要提供一个带有字符串参数的构造方法，即带有异常消息的构造方法，因为这样可以在创建异常对象时提供关于异常的详细信息。同时，如果异常可能会由其他异常引起，那么也应该提供一个带有 Throwable 参数的构造方法，即带有原因的构造方法和带有异常消息和原因的构造方法，这样可以保存引发异常的原始异常信息。

另外，虽然自定义的异常类名没有规定，但是通常以 Exception 这个单词结尾。接下来，我们用下面的代码来测试以上自定义的异常。

```
package com.liuyandong.customexception;

public class CustomExceptionDemo {

    public static void main(String[] args) {
        try {
            methodThatThrowsCustomException("
                                    invalid input");
        } catch (MyCustomException e) {
            System.out.println(" 捕获了自定义的异常: "
                                        + e.getMessage());
            e.printStackTrace();
        }
    }

    // 此方法可能会抛出自定义异常
    public static void methodThatThrowsCustomException
              (String input) throws MyCustomException {
        if (!isValidInput(input)) {
            throw new MyCustomException(" 提供了非法的输入");
        }
        // 如果输入有效，则进行正常处理...
    }

    private static boolean isValidInput(String input) {
        // 在这里进行输入验证逻辑，假设输入长度小于 20 视为无效
        return input != null && input.length() >= 20;
    }
}
```

这段代码定义了一个名为 CustomExceptionDemo 的类，其中包含一个可能会抛出自定义异常 MyCustomException 的方法 methodThatThrowsCustomException。这个方法接受一个字符串输入，如果输入无效（在这里，假设输入长度小于 20 视为无效），则会抛出 MyCustomException。

在 main 方法中，我们调用了 methodThatThrowsCustomException 方法，并使用 try-catch 块捕获可能抛出的 MyCustomException。如果捕获到这个异常，打印出异常信息，并打印出异常的堆栈跟踪。

以上代码运行之后，结果如下：

```
捕获了自定义的异常： 提供了非法的输入
com.liuyandong.customexception.MyCustomException:
提供了非法的输入
    at com.liuyandong.customexception.CustomExceptionDemo.
    methodThatThrowsCustomException(
    CustomExceptionDemo.java:17)
    at com.liuyandong.customexception.CustomExceptionDemo
    .main(CustomExceptionDemo.java:7)
```

16.5 异常链

Java 中的异常链是一种处理异常的技术，它允许我们在捕获一个异常时，继续追踪引发该异常的根源。这种技术特别有用，因为它能帮助开发者更深入地理解程序运行中的错误，提供更多的上下文信息以便进行调试和错误处理。

具体来说，异常链是指一个异常对象中包含了另一个异常对象的情况。在 Java 中，我们通常通过在一个异常对象的构造函数中接受另一个异常对象作为参数来实现异常链。原异常被保存为新异常的一个属性（比如 cause），这样新异常就包含了原异常的信息。这种链式结构使得一个异常能够引发另一个异常，并将后者作为原因或根本原因。

例如，在一个复杂的程序中，一个方法可能在执行过程中遇到错误，如文件不存在或内存不足等。如果不使用异常链，当这些错误发生时，程序可能会崩溃，并丢失重要数据。然而，通过使用异常链，当捕获到一个异常时，我们可以继续向上追踪引发该异常的异常，直到找到引发异常的源头。这样，开发者就能更好地了解问题所在，修复代码，并确保程序的正常运行。

异常链在异常包装的场景下特别有用。有时候，需要在捕获一个异常后将其包装成另一个异常抛出。通过这种方式，可以将底层的异常信息传递给上层，以便在更高的抽象层次上处理异常，同时不丢失低层次的信息。这对于构建健壮、可维护的代码至关重要。接下来，用下面所示的代码来演示异常链的用法。

```
package com.liuyandong.customexception;

import java.io.File;
import java.io.IOException;

public class ExceptionChainExample {
```

```
public static void main(String[] args) {
    try {
        processSomething();
    } catch (MyBusinessException e) {
        e.printStackTrace();
    }
}

public static void processSomething()
            throws MyBusinessException {
    try {
        readFromFile("non_existent_file.txt");
    } catch (IOException ioException) {
        // 将 IO 异常封装到自定义业务异常中，形成异常链
        throw new MyBusinessException
          ("Failed to process due to an IO error",
                                        ioException);
    }
}

public static void readFromFile(String fileName)
                                throws IOException {
    File file = new File(fileName);
    if (!file.exists()) {
        throw new IOException("File not found: "
                                    + fileName);
    }
    // ... 正常的文件读取操作...
}

static class MyBusinessException extends Exception {
    public MyBusinessException(String message,
                                Throwable cause) {
        super(message, cause);
    }
}
```

```
}
```

这段代码定义了一个名为 ExceptionChainExample 的类，其中包含了两个方法：processSomething 和 readFromFile，以及一个自定义的检查型异常类 MyBusinessException。

readFromFile 方法接受一个文件名作为参数，然后尝试打开这个文件。如果文件不存在，它会抛出一个 IOException。

processSomething 方法调用了 readFromFile 方法，并使用 try-catch 块捕获可能抛出的 IOException。如果捕获到这个异常，它会创建一个新的 MyBusinessException，并将 IOException 作为原因传递给 MyBusinessException 的构造方法。这就形成了一个异常链：MyBusinessException 包含了引发它的 IOException。

在 main 方法中，我们调用了 processSomething 方法，并使用 try-catch 块捕获可能抛出的 MyBusinessException。如果捕获到这个异常，我们会打印出异常的堆栈跟踪。

这段代码体现了异常链的概念。当 readFromFile 方法抛出 IOException 时，processSomething 方法捕获这个异常，并将其封装到一个新的 MyBusinessException 中，然后抛出 MyBusinessException。这样，我们就可以在捕获 MyBusinessException 时，通过 getCause 方法获取到原始的 IOException，从而了解异常的详细上下文和原因。

16.6 异常与资源管理

Java 中的异常处理与资源管理紧密相关，尤其是在处理文件、数据库连接和网络套接字等需要在程序结束后释放的资源时。以打开文件为例，在 Linux 系统上，每个进程最多可以打开 1024 个文件。每打开一个文件都消耗一定的资源。如果程序打开文件后不关闭，就会导致资源泄漏。

Java 提供了异常处理机制（包括 try-catch-finally 结构）来帮助程序员优雅地管理和清理资源，即使在异常发生时也不例外。在执行 I/O 操作或其他涉及资源消耗的任务时，资源在创建后必须在不再需要时正确关闭。

传统上，资源是在 finally 块中关闭的，无论 try 块是否成功执行或是否抛出异常，finally 块中的代码始终会被执行，确保资源得到释放。例如，可以使用如下的代码来处理：

```
try {
    InputStream is = new FileInputStream("file.txt");
    // 对 is 进行读取操作...
} catch (IOException e) {
    // 处理异常，如记录错误日志
    e.printStackTrace();
```

```
} finally {
    // 无论如何，确保资源最终被关闭
    if (is != null) {
        try {
            is.close();
        } catch (IOException closeException) {
            // 如果关闭时又抛出异常，可以在此处理或忽略
            closeException.printStackTrace();
        }
    }
}
```

在这段代码中，无论 FileInputStream 对象在读取过程中是否抛出异常，finally 块都会被执行，从而确保 is.close() 方法被调用，关闭文件输入流。然而，这种方式存在一定的冗余和风险，特别是当 finally 块本身也抛出异常时，可能会隐藏原始异常。

幸运的是，自 Java 7 起，Java 引入了 try-with-resources 语句，它可以自动管理实现了 AutoCloseable 接口的资源。在 try-with-resources 结构结束时，Java 会自动调用 close() 方法来关闭资源，极大地简化了资源管理的代码。

例如，以下代码使用 try-with-resources 语句打开文件并读取数据：

```
try (BufferedReader reader = new BufferedReader(
                                new FileReader("file.txt"))) {
    // 执行文件读取操作
} catch (IOException e) {
    // 处理文件读取异常
} finally {
    // 在这里不需要显式关闭 reader，
    // 因为 Java 7 引入的 try-with-resources 会自动关闭它
}
```

在这个改进后的语法中，即使在读取文件的过程中出现了异常，BufferedReader 也会在离开 try 块时被自动关闭，无需在 finally 块中手动关闭。

Java 通过异常处理机制与资源管理相结合，既保障了程序在面对异常时的稳定性，也确保了系统资源的有效利用和释放，提高了代码的健壮性和安全性。

17 泛型

Contents

自 Java 1.0 问世以来，这门编程语言不断革新，陆续添加了许多重要的功能。若要评选后续版本中加入 Java 的最重要功能，我的排名将是：第一名，当属 Java 5 引入的泛型 (Generics)；第二名和第三名的排序略有争议，分别是 Java 5 推出的注解 (Annotations) 和 Java 8 带来的 Lambda 表达式。

泛型的地位毋庸置疑，它是 Java 发展历程中最为杰出的里程碑之一。泛型解决了 Java 在编译时无法进行参数化类型检查的根深蒂固的问题，极大地提升了代码的安全性和可读

性，开启了 Java 类型系统的新纪元。至于注解和 Lambda 表达式，两者孰先孰后、孰轻孰重，我认为存在一定争议。注解为 Java 代码添加了元数据支持，增强了语言的自描述能力和可扩展性，广泛应用于框架设计。而 Lambda 则为 Java 注入了函数式编程理念，提高了集合操作的效率，是 Java 拥抱现代编程范式的重要一步。

无论如何，这三大功能的加入，都是 Java 语言发展历程中最为重要的分水岭，它们共同推动了 Java 的巨大进步，让这门语言在安全性、灵活性和易维护性方面有了全新的提升，确保了 Java 在当代软件开发领域的领先地位。在本书中，我分别为这三大功能各写一章来介绍，本章介绍泛型。

17.1 泛型的概念与作用

泛型 (Generics) 的概念最早可以追溯到 20 世纪 70 年代，当时被应用于 ALGOL 68、CLU 等编程语言中。无论何种语言，引入泛型编程的根本目的，用一句话来概括就是: 通过参数化类型 (Parameterized Types)，实现代码最大程度的重用与类型安全。

泛型允许在定义类、接口或方法时使用类型参数，这些参数在使用时由编译器执行类型安全检查和绑定，从而保证了类型安全。同时，泛型也使代码更加通用，相同的类或方法只需编写一次，便可以应用于不同的数据类型，极大地提高了代码的重用性。

Java 语言在 2004 年发布的 Java 5 中才引入了泛型编程机制，解决了此前集合类型转换异常等问题。自那以后，Java 泛型不断得到完善和强化，成为 Java 中重要的编程特性之一，使 Java 程序在保证类型安全的前提下，也获得了更好的抽象能力和灵活性。

然而，自从 Java 5 引入泛型以来，其使用一直存在诸多争议和分歧。这些争论几乎涉及了 Java 泛型的方方面面，比如类型擦除机制、通配符的使用约束、类型转换问题、与可变参数的兼容性，以及与面向对象继承的契合度等。然而，这些争议与不足在某种程度上源自对 Java 语言发展历程的理解不够透彻。若能深入剖析 Java 设计者当时所面临的历史语境，或许就能体谅他们的权衡取舍，换作他人亦难以拿捏得当、将泛型特性实现得更加完美。

回溯 Java 泛型的发展历程或许能让我们更全面地认识其来龙去脉。接下来，我们不妨从这一视角出发，重新审视 Java 泛型特性的起源与变化。

17.1.1 Java 泛型的历史

前文已经指出，Java 泛型是从 Java 5 开始引入的，这表明 Java 的早期版本并不支持泛型。那么是谁引入了泛型呢？这位颇具盛名的人物就是 Martin Odersky，他以开发出 Scala 语言而广为人知。Scala 是一种多范式编程语言，融合了面向对象编程和函数式编程的特性，被认为是一种具有革命性的语言。Martin Odersky 因此而声名显赫。

在 Java 发布 Generic Java 的时候，主要的四位开发者 Martin Odersky、Philip Wadler、Gilad Bracha 和 David Stoutamire，他们穿着四种颜色的 T-Shirt，前面印着 GJ 两个字母，后面印着一语双关的宣传：Making Java easier to type and easier to type. 其中的一个 type 的意思是：类型化。翻译后的意思是” 使 Java 更容易使用，更容易类型化”，这指的是使 Java 中的类型系统更加清晰和易于理解，尤其是在使用泛型（generic）时。

化作英语老师班门弄斧

“Making Java easier to type and easier to type” 这句广告的妙处在于对 “type” 一词的双关用法。

Easier to type (编程更容易): 第一个 “type” 是指打字。Java 引入泛型（Generic Type）后，代码的表达变得更加简洁，减少了冗长的类型转换和重复代码。因此，开发者在编写代码时可以减少键入的内容，使得代码书写更加轻松快捷。

Easier to type (类型化更容易): 第二个 “type” 是指 “类型”（type）这一编程概念。泛型允许开发者在不明确指定特定类型的情况下使用更灵活和通用的类型。这样一来，类型的处理变得更加方便，代码的类型安全性也增强了，从而减少了运行时的错误。

这句广告巧妙地利用了 “type” 一词的双重含义，传达了泛型不仅能让代码书写变得简单，同时也让类型系统的使用更加灵活和安全。

图 17.1: 从左到右：Philip Wadler, Martin Odersky, Gilad Bracha, Dave Stoutamire

图 17.2: 背面的宣传

在 1995 年，Martin Odersky 已经获悉了 Java 这门新兴的编程语言。作为一名专业的语言设计者，他当时在德国的一所大学担任教授，并且正在设计一门新的编程语言，与 Java 有很多相似之处，但更加强大，其中包括泛型等特性。后来，他发布了这门编程语言，名为 Pizza。然而，尽管 Pizza 没有取得广泛的成功，但引起了 Java 团队的关注，特别是当时的 Java 主要负责人之一 Gilad Bracha。

我强烈建议大家搜索一下 Pizza 这门编程语言，就可以看到当年 Pizza 语言的语法，与 Java 8 以后的语法非常相似，从某种程度上可以说是超越时代的设计。为什么会有这样的效果呢？原因在于 Gilad Bracha 邀请 Martin Odersky 加入 Sun 公司，并且主导了 Java 泛型的引入。此外，他还发表了一篇非常著名的论文《Making the future safe for the past: Adding Genericity to the Java Programming Language》，该论文名翻译成中文非常有远见：“为过去铺平未来之路：将泛型性添加到 Java 编程语言中”。

在这篇论文中，作者强调了 Java 中缺乏泛型性导致的问题，如代码的冗余、不安全性和低效率。并认为引入泛型编程可以提高代码的抽象能力、类型安全性和性能。该论文提出了 GJ（Generic Java）的设计，一种通过编译期翻译的方式，将泛型语义引入 Java 语言的方案。GJ 克服了与遗留代码不兼容的障碍，并引入了原始类型和重构机制等创新概念。GJ 为 Java 泛型化铺平了道路，奠定了基础，最终实最终实现了 Java 5 及更高版本对泛型编程的原生支持。我认为，基本上前面提到的那些争议，都可以在这篇论文中找到答案。

其中最大的争议莫过于为了兼容 Java 5 之前的版本，采用了泛型擦除（Type Erasure）的技术。泛型擦除是 Java 语言实现泛型机制的一种策略，在 Java 中，尽管开发者在编写代码时可以使用泛型来创建具有参数化类型的类、接口和方法，但在编译后的字节码以及运行时的 Java 虚拟机（JVM）中，并不会保留这些具体的类型参数信息。这种设计导致了一些限制和所谓的 “历史遗留问题”，如无法在运行时通过反射获取确切的类型参数信息，以及不能创建泛型数组等。但是优点在于保持了 Java 一贯以来的 “兼容性”。在后续章节中，我们会逐一讨论这些问题。

17.1.2 泛型的用途与语法

接下来，我们将通过两个具有代表性的示例来直观展现泛型的实际运用和相关语法结构，旨在为后续章节对泛型更深层次的讲解奠定坚实的基础。首先，我们先看没有使用泛型的例子。

```
import java.util.ArrayList;

public class NonGenericExample {
    public static void main(String[] args) {
        // 创建一个非泛型 ArrayList，只能存储 Object 类型
        ArrayList list = new ArrayList();

        // 可以添加任意类型对象
        list.add("Hello");
        list.add(123);

        // 需要强制类型转换取出元素
        String str = (String) list.get(0);
        System.out.println(str);

        // 如果不进行正确的类型转换，
        // 可能会抛出 ClassCastException 异常
        try {
            Integer num = (Integer) list.get(1);
            System.out.println(num);
        } catch (ClassCastException e) {
            e.printStackTrace();
        }
    }
}
```

这段代码展示了一个没有使用泛型的示例。在这个示例中，使用了 ArrayList 而不是 ArrayList，这意味着它是一个非泛型的集合，只能存储 Object 类型的元素。因为没有指定具体的类型，所以可以向集合中添加任意类型的对象。

需要注意的是，由于没有使用泛型，所以在取出集合中的元素时需要进行强制类型转换。在代码中，通过 (String) 和 (Integer) 进行了类型转换来取出元素。如果不进行正确的类型转换，可能会在运行时抛出 ClassCastException 异常，因为在编译时无法保证取出的元素

类型与强制转换的类型一致。

尽管非泛型集合可以存储任意类型的对象，但在编程中推荐使用泛型来提高类型安全性和代码的清晰度，避免在运行时出现类型转换错误的问题。接下来，咱们再来使用泛型重写这段代码，完整的代码如下：

```
import java.util.ArrayList;

public class GenericExample {
    public static void main(String[] args) {
        // 创建一个泛型 ArrayList，指定只存储 String 类型
        ArrayList<String> list = new ArrayList<>();

        // 编译器会阻止添加非 String 类型
        list.add("Hello");
        // 这行代码会导致编译错误，
        // 因为不能向 String 集合中添加 Integer
        // list.add(123);

        // 不需要进行类型转换就能安全地获取元素
        String str = list.get(0);
        System.out.println(str);

        // 由于集合在编译时已经知道其元素类型，
        // 因此不会出现 ClassCastException 异常
    }
}
```

这段代码演示了泛型的用法。在这个例子中：

首先，通过 ArrayList list = new ArrayList<>(); 创建了一个泛型的 ArrayList，并指定该列表只能存储 String 类型的元素。使用 list.add(“Hello”); 添加了一个字符串元素到列表中。由于列表被声明为只能存储 String 类型，因此编译器会阻止向列表中添加非 String 类型的元素。

使用 String str = list.get(0); 从列表中获取元素，并将其存储在一个 String 类型的变量中。由于列表在编译时已经知道其元素类型为 String，因此不需要进行类型转换就能安全地获取元素。

在 Java 中，泛型的语法如下：

```
ArrayList<String> list = new ArrayList<>();
```

这条语法表示创建了一个泛型的 ArrayList 集合，并指定该集合只能存储 String 类型的元素。下面对这条语法进行详细解释：

ArrayList：这部分指定了要创建的 ArrayList 集合的类型。在尖括号 <> 中指定了泛型参数，即 String，表示该 ArrayList 只能存储 String 类型的元素。这样就限定了 ArrayList 只能存储特定类型的对象，提高了类型安全性。

list：这是创建的 ArrayList 对象的名称，可以根据需要自行命名。

new ArrayList<>()：这部分是用来实例化 ArrayList 对象的语法。在 <> 中没有指定具体的类型，而是留空。这是因为在 Java 7 及更高版本中引入了“钻石语法”，允许在实例化时省略泛型类型，编译器会根据上下文自动推断泛型类型。因此，new ArrayList<>() 会根据前面的 ArrayList 自动推断出要创建一个存储 String 类型的 ArrayList 对象。

总的来说，这段代码展示了如何使用泛型来在编译时就能够检查和限制集合中的元素类型，避免在运行时出现类型转换错误或 ClassCastException 异常。这提高了代码的安全性和可读性。

现在，让我们稍作停顿并深入思考一下，上述关于 ArrayList 的例子实际上只是泛型在 Java 中深远影响的初步展现。泛型不仅能够显著改变集合类的行为和使用方式，确保元素类型的严格一致性，而且其作用远不止于此。事实上，上述提及的 ArrayList 仅是一个泛型应用于集合框架以提升代码安全性的典型示例，而泛型在 Java 中的影响力还广泛渗透到类、接口、方法定义以及整个程序设计的诸多层面。

接下来，我们将继续从技术细节上研究 Java 泛型的各个核心组成部分，并通过实际代码示例来阐明它们的功能和优势。

17.2 定义泛型类

本小节的目标是定义泛型类以实现类型安全的数据容器。

在 Java 中，泛型类允许我们创建一个可重用的数据结构，该结构可以在不同场景下存储任何指定类型的对象。通过使用类型参数（如 T），可以确保在整个类中元素的类型一致性，并在编译时提供类型检查，从而避免运行时可能出现的 ClassCastException。接下来，演示定义泛型类的用法。

```
class GenericClass<T> {
    private T data;
```

```
    // 构造函数，接受一个 T 类型的参数
    public GenericClass(T data) {
        this.data = data;
    }

    // 获取数据的方法
    public T getData() {
        return data;
    }

    // 设置数据的方法
    public void setData(T data) {
        this.data = data;
    }

    public static void main(String[] args) {

        GenericClass<String> stringClass =
                                new GenericClass<>("Hello");
        GenericClass<Integer> integerClass =
                                new GenericClass<>(123);

        System.out.println(stringClass.getData());
        System.out.println(integerClass.getData());
    }
}
```

这段 Java 代码定义了一个名为 GenericClass 的泛型类，并在主方法中创建了两个实例，一个存储字符串类型数据，另一个存储整数类型数据。接下来，我详细解释这段代码：

class GenericClass：声明了一个泛型类 GenericClass，其中 T 是一个类型参数，代表任意类型，在实例化时可以替换为具体的数据类型。在 Java 泛型中，使用大写字母 T 作为类型参数的名称是一种常见的约定，但它并不是必须的。可以使用任何喜欢且不与 Java 关键字冲突的标识符来作为类型参数的名字。一般来说，有一些约定成俗的习惯：T 通常代表 “Type”，E 通常代表 “Element”（常用于集合），K 代表 “Key”（常用于映射或键值对结构），V 代表 “Value”（同样常用于映射或键值对结构）。使用这些的目的是提高代码可读性，并让其他开发者一看就知道这个类型参数大致代表什么含义。

private T data：类中定义了一个私有字段 data，其类型为 T。这意味着 data 可以存储任何指定类型（在实例化时确定）的对象。

构造函数接受一个类型为 T 的参数，并将其赋值给类的成员变量 data。这样，当创建 GenericClass 的实例时，可以传递一个与你指定类型相符的对象作为参数。

随后，实例化了两个 GenericClass 对象，分别为 stringClass 和 integerClass。前者存储字符串类型数据 “Hello”，后者存储整数类型数据 123。通过尖括号中的类型参数，编译器知道每个实例应该存储哪种类型的数据。

这段代码展示了如何使用 Java 泛型来创建一个通用且类型安全的数据容器类，可以根据需要存储不同类型的数据，这提高了类型系统的严谨性和代码质量，同时也降低了因类型转换带来的运行时错误风险。

17.3 定义泛型接口

本节的目标是设计和实现泛型接口以增强复用性，通过例子描述如何定义一个具有泛型参数的接口，例如 interface GenericInterface {...}，讨论接口中的泛型方法，并给出实现该接口的具体类如何指定泛型参数的具体类型。接下来，用如下的代码演示泛型接口的用法。

```
package com.liuyandong.genericinterface;

// 定义一个具有泛型参数 T 的接口
public interface GenericInterface<T> {
    // 泛型方法，声明一个接受 T 类型参数的方法
    void add(T item);

    // 泛型方法，返回一个 T 类型的元素
    T get(int index);
}
```

GenericInterface 是一个接口，其中是类型参数，代表任意非限定类型。在实际应用中，当实现这个接口时，可以指定 T 为任何具体的数据类型。

接口中的 add 和 get 方法都是泛型方法，它们的参数或返回值类型由类型参数 T 决定，这意味着当实现这些方法时，它们处理的数据类型将与实现类中指定的类型一致。接下来，让两个类实现泛型接口，这两个类的代码分别如下：

第一段代码演示的是实现接口的 StringImplementation 类：

```
package com.liuyandong.genericinterface;

import java.util.ArrayList;

public class StringImplementation implements
                            GenericInterface<String> {

    private final ArrayList<String> list =
                                new ArrayList<>();

    @Override
    public void add(String item) {
        list.add(item);
    }

    @Override
    public String get(int index) {
        if (index >= 0 && index < list.size()) {
            return list.get(index);
        }
        // 如果索引越界，返回 null
        return null;
    }
}
```

接下来是实现接口的 IntegerImplementation 类：

```
package com.liuyandong.genericinterface;

import java.util.ArrayList;

public class IntegerImplementation implements
                            GenericInterface<Integer> {

    private ArrayList<Integer> list = new ArrayList<>();
    @Override
    public void add(Integer item) {
```

```
            list.add(item);
        }

        @Override
        public Integer get(int index) {
            if (index >= 0 && index < list.size()) {
                return list.get(index);
            }
            // 如果索引越界，返回 null
            return null;
        }
    }
```

以上 StringImplementation 和 IntegerImplementation 分别实现了 GenericInterface 接口，并指定了 T 的具体类型——分别为 String 和 Integer。

在这两个实现类中，都定义了一个私有成员变量 ArrayList 来存储对应类型的数据，并实现了接口中定义的 add 和 get 方法，以操作存储的数据。

通过这种方式，泛型接口允许我们创建一个通用的契约，不同的实现类可以根据需要自由选择处理的数据类型，同时保持了类型安全性和代码复用性。接下来，我们用下面的代码测试一下这两个类。

```
package com.liuyandong.genericinterface;

public class GenericInterfaceDemo {

    public static void main(String[] args) {
        StringImplementation si =
                            new StringImplementation();
        si.add("Hello");
        System.out.println(si.get(0)); // 输出 Hello

        IntegerImplementation ii =
                            new IntegerImplementation();
        ii.add(123);
        System.out.println(ii.get(0)); // 输出 123
```

```
        }
    }
```

首先创建了一个 StringImplementation 对象 si，并添加了一个字符串”Hello”，然后通过 get(0) 获取并打印出第一个元素。接着创建了一个 IntegerImplementation 对象 ii，添加了一个整数 123，并同样获取并打印出第一个元素。这样展示了泛型接口在不同具体类型下的灵活性与复用性。

总结一下，相比于前面所学的普通接口，泛型接口有什么优点呢？主要有以下两个优点：

一是类型安全：泛型接口允许在定义接口时指定一个或多个类型参数（如 GenericInterface），这样实现该接口的类在使用时就必须指定具体的数据类型。这可以确保编译时进行类型检查，减少运行时由于类型转换引发的 ClassCastException 异常。

二是代码复用：泛型接口可以在多种数据类型上重用，无需为每种数据类型编写单独的接口。如例子所示，可以有一个 GenericListInterface 用于处理不同类型的列表，无论是 Integer、String 还是自定义类型，都可以通过继承这个接口并指定对应的类型参数来实现。

17.4 定义泛型方法

在 Java 中，泛型方法允许在方法声明中指定泛型类型参数，这些类型参数可以用于定义方法的返回类型、参数类型以及局部变量类型。泛型方法可以在普通类中、泛型类中或者接口中独立定义，它们的主要目的是提供一种强类型检查的机制，同时保持代码的复用性和灵活性。

定义泛型方法意味着在方法的定义中使用类型参数，使其能在多种数据类型之间灵活切换并在编译时进行类型检查。这样的方法允许调用者在调用方法时指定所需的具体类型，使得同一个方法签名可以根据不同的类型参数产生不同的行为，但执行的核心逻辑保持不变。

在讨论多态性的实现机制时，我提到了第三种实现方法，即通过泛型来实现多态性。通过定义泛型方法，我们可以实现在保持方法签名不变的前提下适应多种数据类型的操作需求。这意味着，同一个方法可以根据传入的不同参数类型执行相应的逻辑处理，从而充分展现了多态的特性，使得代码更具通用性和灵活性。

定义泛型方法的定义格式如下：

```
public <T> T genericMethod(T t) {
    // 方法体
}
```

- 泛型方法的类型参数部分由尖括号 <> 包围，位于方法返回类型之前。
- 类型参数可以有一个或多个，之间用逗号分隔。
- 类型参数可以在方法的返回类型、参数类型及方法体内使用。

举个例子，假设需要一个泛型方法来交换数组中两个元素的位置，其代码如下所示：

```
import java.util.Arrays;

public class GenericMethod {

    // 定义一个泛型方法 swap，
    // 它接受一个数组 arr 和两个索引 i、j，
    // 交换 arr[i] 和 arr[j] 的位置

    public static <T> void swap(T[] arr, int i, int j) {
        T temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        // 测试 swap 方法
        Integer[] intArray = {1, 2, 3, 4};
        System.out.println(" 交换之前: "
                           + Arrays.toString(intArray));
        swap(intArray, 0, 3);
        System.out.println(" 交换之后: "
                           + Arrays.toString(intArray));

        String[] stringArray = {"Hello", "World"};
        System.out.println(" 交换之前: "
                           + Arrays.toString(stringArray));
        swap(stringArray, 0, 1);
        System.out.println(" 交换之后: "
                           + Arrays.toString(stringArray));
    }
}
```

上面的代码定义了一个名为 GenericMethod 的类，其中包含一个泛型方法 swap。swap 方法接受一个类型为 T 的泛型数组 arr 和两个整数索引 i 和 j，然后交换数组中这两个索引位置的元素。main 方法展示了如何使用 swap 方法来交换一个 Integer 数组和一个 String 数组中元素的位置，并且在交换前后将数组内容打印出来。运行结果如下：

```
交换之前：[1, 2, 3, 4]
交换之后：[4, 2, 3, 1]
交换之前：[Hello, World]
交换之后：[World, Hello]
```

使用泛型方法 swap 的优点在于提供了更高的代码复用性和类型安全。该方法可以接受任何类型的数组，无需为不同类型编写多个重载方法，且在编译时会进行类型检查，避免了运行时类型转换错误。

17.4.1 泛型方法和多态

还记得在多态那一章里，我讲过泛型方法也算多态的一种实现方式。这里咱们再回头讲一下多态的问题。在给出的 swap 方法示例中，方法是泛型的，因为它可以接受任何类型的数组，并交换数组中两个元素的位置。这里的多态性体现在：

o 类型多态性：swap 方法能够操作任何类型的数组（如 Integer[]、String[] 等），这意味着它对类型是多态的。这种多态是在编译时决定的，编译器会为每种使用情形生成相应的类型检查和类型转换代码。

o 编译时安全：泛型方法在编译时就会进行类型检查，确保不会对类型进行错误的操作。例如，如果尝试将一个 Integer[] 数组传递给预期为 String[] 类型的方法，则会在编译时发现错误。

但是，相较于传统的基于继承或者基于接口的多态，用泛型还是有一些区别，主要的区别可以分为下面两点：

o 传统多态性依赖于类的继承关系和接口实现，它允许在运行时决定调用哪个类的哪个方法。

o 泛型方法的多态性不依赖于类的继承或接口实现，而是基于类型参数。它允许同一个方法根据传入的不同类型参数在编译时解析为对应的方法体。

总结，泛型方法通过允许在一个方法中操作多种类型的数据来实现一种类型的多态性，而这种多态性主要是在编译时而非运行时体现。这与基于继承的运行时多态性是不同的概念，但都是 Java 语言灵活性和强大类型安全性的体现。

17.4.2 泛型方法与泛型类、泛型接口的区别

泛型方法是在方法级别上应用泛型，其类型参数的作用域限于方法本身。这意味着无论类是否是泛型的，都可以定义泛型方法。

泛型类是在类级别上应用泛型，其类型参数可以在整个类中使用，包括其字段、方法返回类型、参数类型等。泛型类的一个实例化对象的类型参数在该对象中是固定的。

泛型接口与泛型类类似，但用于接口。实现泛型接口的类可以选择传递具体的类型参数给接口，或者也成为泛型类，将决定权交给实现它的类。

总结来说，泛型方法提供了一种在单个方法级别上使用类型参数的方式，而泛型类和接口则提供了一种在更广泛的范围内使用类型参数的机制。泛型方法可以单独存在，不依赖于泛型类或接口。

泛型方法实现的多态性与传统的多态性（基于继承体系的多态性）有所不同。在 Java 中，传统的多态性通常是指通过继承和接口实现的，它允许子类的对象在运行时被视为父类的实例，从而实现方法的动态绑定。

泛型方法实现的多态性是类型多态性，它基于泛型编程的概念。使用泛型方法时，方法可以接受不同类型的参数，并在编译时保证类型安全，而不是在运行时。这意味着你可以用同一个方法处理不同类型的数据，只要这些数据类型在方法调用时满足方法的类型约束即可。这种方式实现的多态性，更多地是在编译时期的灵活性和安全性，而不是运行时期对象的行为多态性。

17.5 类型通配符的概念与使用场景

Java 中的类型通配符，用? 表示，是泛型的一种特殊用法。它允许在定义泛型类、接口或方法时，使用一种更灵活的方式来指定类型参数。使用通配符可以编写更通用的代码，例如，可以编写一个方法来处理任何类型的 List，而无需为每种类型的 List 编写不同的方法。因此类型通配符可以帮助我们避免类型转换错误，并确保代码在编译时就能进行类型检查。

类型通配符可以分为三类，分别为：

o 无界通配符? ： 表示可以匹配任何类型。例如，List<?> 可以表示任何类型的 List。

o 上界通配符? extends T ： 表示可以匹配 T 类型或 T 的子类型。例如，List<? extends Number> 可以表示任何包含 Number 类型或其子类型元素的 List。

o 下界通配符? super T：表示可以匹配 T 类型或 T 的超类型。例如，List<? super Integer> 可以表示任何包含 Integer 类型或其超类型元素的 List。

接下来，我将分别介绍这三类通配符。

17.5.1 无界通配符 (?)

Java 泛型中的无界通配符是 <?>，它表示可以匹配任何类型。它通常用于以下情况：

o 泛型方法参数: 当方法需要操作泛型类型，但不需要对类型进行任何特定的限制时，可以使用无界通配符。例如：

```
public static <T> void printList(List<?> list) {
    for (Object element : list) {
      System.out.println(element);
    }
}
```

这个方法可以打印任何类型的列表，因为 <?> 可以匹配任何类型。

o 泛型类或接口: 当类或接口需要使用泛型类型，但不需要对类型进行任何特定的限制时，可以使用无界通配符。例如：

```
public interface Collection<?> {
    // ...
}
```

这个接口表示任何类型的集合。

需要注意的是，使用无界通配符时，编译器会将元素视为 Object 类型。这意味着你不能对元素进行任何类型特定的操作，例如调用特定方法或访问特定属性。例如，在上面的 printList 方法中，你不能将元素转换为特定类型，因为编译器不知道元素的实际类型。在使用泛型时，通常应该避免过度使用无界通配符 <?>，因为它会使代码变得不够明确，并且可能会导致编译时或运行时出现意外的行为。在大多数情况下，最好使用有界通配符（指马上要讲的上界通配符和下界通配符）或具体的泛型参数来提高代码的清晰性和类型安全性。

17.5.2 上界通配符（<? extends XXX> ）

Java 泛型中的上界通配符是 <? extends XXX>，它表示可以匹配 T 类型或 T 的子类型。它通常用于以下情况：

o 泛型方法参数: 当方法需要操作泛型类型，并且需要保证类型至少是某个特定类型或其子类型时，可以使用上界通配符。例如：

```
public static void copyList(List<? extends Number> source,
                            List<? super Number> destination) {
    for (Number number : source) {
      destination.add(number);
    }
}
```

这个方法可以将任何 Number 类型或其子类型的列表复制到另一个 Number 类型或其父类型的列表中。

o 泛型类或接口: 当类或接口需要使用泛型类型，并且需要保证类型至少是某个特定类型或其子类型时，可以使用上界通配符。例如:

```
public interface Comparable <T extends Comparable<T>> {
    int compareTo(T o);
}
```

这个接口表示任何可以与 T 类型或其子类型进行比较的类型。

使用上界通配符的好处是，它可以保证类型安全。例如，在上面的 copyList 方法中，由于源列表中的元素至少是 Number 类型，因此可以安全地将它们添加到目标列表中，因为目标列表可以接受 Number 类型或其父类型。

需要注意的是，使用上界通配符时，你只能读取元素，不能写入元素。这是因为编译器无法确定元素的具体类型，因此无法保证写入操作的类型安全。

17.5.3 下界通配符（<? super XXX> ）

Java 泛型中的下界通配符是 <? super XXX>，其中 T 是一个类型参数。它表示可以匹配 T 或 T 的任何父类型。它通常用于以下的情况:

o 泛型方法参数: 当方法需要操作泛型类型，并且需要保证类型至少是某个特定类型或其父类型时，可以使用下界通配符。例如:

```
public static <T> void addToList(List
                    <? super T>list, T element) {
    list.add(element);
}
```

这个方法可以将任何 T 类型或其子类型的元素添加到 T 类型或其父类型的列表中。

o 泛型类或接口: 当类或接口需要使用泛型类型，并且需要保证类型至少是某个特定类型或其父类型时，可以使用下界通配符。例如:

```
public interface Comparator <? super T> {
    int compare(T o1, T o2);
}
```

这个接口表示任何可以比较 T 类型或其父类型的比较器。

使用下界通配符的好处是，它可以保证写入操作的类型安全。例如，在上面的 addToList 方法中，由于元素的类型至少是 T，因此可以安全地将它添加到 T 类型或其父类型的列表中。

需要注意的是，使用下界通配符时，你只能写入元素，不能读取元素。这是因为编译器无法确定元素的具体类型，因此无法保证读取操作的类型安全。

总结一下:

上界通配符用于读取操作，保证类型至少是 T 或其子类型。下界通配符用于写入操作，保证类型至少是 T 或其父类型。其实记住以上的两句话就可以了，只要遵守这两条规则，通常就能避免泛型代码出现重大问题。然而，为了指导开发者进行更安全的泛型编程，《Effective Java》作者 Joshua Bloch 对此进行了更深入的总结，提出了 PECS 原则。接下来，咱们一起学习一下这个原则。

17.5.4 通配符的 PECS 原则 (Producer Extends,Consumer Super)

先来解释一下这两句英文是什么意思:

Producer Extends (生产者使用 extends): 如果你的泛型类或方法是生产者，即它只从泛型类型中读取数据，那么应该使用上界通配符 <? extends T>。

Consumer Super (消费者使用 super): 如果你的泛型类或方法是消费者，即它只向泛型类型中写入数据，那么应该使用下界通配符 <? super T>。

PECS 原则可以帮助我们编写更加安全和灵活的泛型代码。从安全性的角度来讲，使用 PECS 原则可以确保类型安全。例如，如果一个方法使用 <? extends Number> 作为参数，那么它可以接受任何 Number 的子类型，例如 Integer 或 Double。这样可以确保方法不会将错误类型的对象写入到泛型类型中。

从灵活性的角度来讲，使用 PECS 原则可以使代码更加灵活。例如，一个使用 <? super Integer> 作为参数的方法可以接受任何 Integer 的父类型，例如 Number 或 Object。这样可以使方法更加通用，可以处理更多类型的对象。

17.6 泛型的继承与子类型规则

Java 中的泛型系统设计得非常灵活，同时也相当复杂。它包括继承与子类型规则、类型的不变性以及通配符的协变与逆变等概念。让我们逐一详细解析这些概念。

17.6.1 泛型的不变性规则及其影响

在 Java 中，泛型类型的继承规则与普通类型有所不同。这主要是因为泛型的引入是为了在编译时提供更严格的类型检查，并且保持向后兼容性。

泛型类之间的继承：一个泛型类可以继承另一个泛型类，就像普通类一样。但是，即使类型参数相同，泛型类型也不会因为类型参数的关系自动继承或实现另一个泛型类型。

以下面的代码举例来说：

```
// 定义一个泛型类
class GenericClass<T> {
    private T value;

    public GenericClass(T value) {
      this.value = value;
    }

    public T getValue() {
      return value;
    }
  }

  // 定义两个普通类
  class A {}
  class B extends A {}

  // 现在创建两个泛型类的实例
  GenericClass<A> aGeneric = new GenericClass<>(new A());
  GenericClass<B> bGeneric = new GenericClass<>(new B());
```

在这个例子中，类 B 继承了类 A。但是，GenericClass 和 GenericClass 是两个独立的类，它们之间没有继承关系。

原因是，当编译器遇到泛型类时，会根据实际使用的类型参数进行替换。例如，

GenericClass 会被编译成一个使用 A 类型的类，而 GenericClass 会被编译成一个使用 B 类型的类。由于 A 和 B 是不同的类型，所以这两个类之间没有继承关系。

注意: 虽然泛型类之间没有直接的继承关系，但可以通过类型参数的限定来实现类似继承的效果。例如，可以声明一个泛型类 GenericClass，这样就限定了 T 必须是 A 的子类。

接下来再讲泛型类型的继承关系，泛型类型的继承关系确实有点绕，它基于原始类型，也就是忽略类型参数的类型。这意味着即使类型参数不同，泛型类型之间也能存在继承关系。我觉得用代码来演示一下会比较清晰：

```
// 定义一个泛型类
class GenericClass<T> {
    T value;

    public GenericClass(T value) {
      this.value = value;
    }
  }

  // 定义一个继承自 GenericClass 的子类，但使用不同的类型参数
  class SubGenericClass<U> extends GenericClass<String> {
    U otherValue;

    public SubGenericClass(String value, U otherValue) {
      super(value); // 调用父类的构造函数
      this.otherValue = otherValue;
    }
  }

  public class GenericInheritances {
    public static void main(String[] args) {
      // 创建 SubGenericClass 的实例，类型参数为 Integer
      SubGenericClass<Integer> subClass =
                      new SubGenericClass<>("Hello", 10);

      // 可以访问父类 GenericClass 的成员
      System.out.println(subClass.value); // 输出 "Hello"
      System.out.println(subClass.otherValue); // 输出 10
```

```
    }
}
```

在这个例子中，有以下几点值得注意：

1. GenericClass 是一个泛型类，它有一个类型参数 T。
2. SubGenericClass 继承自 GenericClass，但它使用了不同的类型参数 U。
3. 在 SubGenericClass 的构造函数中，调用了父类 GenericClass 的构造函数，并传入一个字符串类型的参数。
4. 在 main 方法中，创建了 SubGenericClass 的实例，并分别访问了父类和子类的成员。

需要注意的是，尽管 SubGenericClass 的类型参数是 Integer，但它仍然可以继承自 GenericClass，因为它们的原始类型都是 GenericClass。

在 Java 中，泛型是不变的（Invariant）。这意味着即使 TypeA 是 TypeB 的一个子类型，GenericType 也不是 GenericType 的子类型，反之亦然。换句话说，泛型类型之间的继承关系不会因为其类型参数之间的继承关系而改变，它们是完全独立的。

泛型不变性的影响主要体现在：

o 类型安全：不变性帮助保证了类型安全，因为你不能意外地将一个类型的对象赋值给另一个不兼容类型的变量。

o 限制：不变性也带来了一定的限制，比如在一些需要类型协变或逆变的场景中，直接使用泛型类型可能无法满足需求。

17.6.2 通配符的协变规则与逆变规则

为了提供更大的灵活性，Java 引入了通配符和泛型方法，允许在类型参数的使用上更加灵活，从而引出了协变与逆变的概念，协变可以使方法接受更广泛的类型，而逆变可以使方法更安全地修改泛型类型。接下来，我们来详细讲一下这两个概念。

17.6.2.1 协变（Covariance）

协变（Covariance）：在 Java 泛型中，协变指的是泛型类型可以随着其类型参数的变化而“向上”兼容，通过 <? extends Type> 来实现协变。如果 TypeA 是 TypeB 的子类型，则 GenericType<? extends TypeA> 可以看作是 GenericType<? extends TypeB> 的子类型。这允许你从泛型类型中读取数据时，确保数据至少是指定的类型或其子类型。比如，下面这个例子：

```
List<? extends Number> numbers = new ArrayList<Integer>();
```

在这个例子中，List<? extends Number> 表示一个可以包含任何 Number 子类型的列表。由于 Integer 是 Number 的子类型，因此可以将 ArrayList 赋值给 List<? extends Number> 类型的变量。在这种情况下，下面的代码是安全的。

```
Number number = numbers.get(0);
```

由于 numbers 列表中的元素至少是 Number 类型，因此可以安全地将 get(0) 返回的值赋值给 Number 类型的变量。

协变的好处：

o 提高代码的灵活性：允许使用更广泛的类型。

o 增强代码的可读性和可维护性：明确表明泛型类型可以接受哪些类型。

需要注意的是，协变只适用于读取数据。如果尝试将数据写入协变的泛型类型，则可能会导致类型安全问题。那写入数据应该用什么呢？就是接下来要讲的逆变。

17.6.2.2 逆变（Contravariance）

逆变是一种类型转换的方向，它与我们通常理解的类型转换方向相反。通常，子类型可以转换为父类型，这称为协变。而逆变则允许我们将父类型转换为子类型，但需要满足一些特定的条件。在 Java 中，使用 <? super Type> 语法来实现逆变。它表示该泛型类型接受 Type 或其任何父类型。这在处理需要写入数据到泛型类型时非常有用，因为它可以确保数据至少是指定的类型或其父类型，从而保证类型安全。

以下是一些解释逆变的关键点：

子类型化关系反转: 当 TypeA 是 TypeB 的子类型时，GenericType<? super TypeB> 可以被视为 GenericType<? super TypeA> 的子类型。这意味着你可以将一个 GenericType<? super TypeB> 的实例传递给需要 GenericType<? super TypeA> 的方法或变量。

写入数据: 逆变主要用于写入数据到泛型类型。由于 GenericType<? super Type> 接受 Type 或其任何父类型，你可以安全地将 Type 或其子类型的实例写入到该泛型类型中。

读取数据: 使用 <? super Type> 语法时，读取数据会受到限制。你只能读取 Object 类型的数据，因为编译器无法确定具体的类型。

为了更好地理解逆变，让我们看两个例子：

```
List<? super Number> list = new ArrayList<Number>();

list.add(10); // 可以添加 Integer，因为 Integer 是 Number 的子类
list.add(3.14); // 可以添加 Double，因为 Double 是 Number 的子类
```

```
Object obj = list.get(0); // 只能读取 Object 类型
```

在这个例子中，List<? super Number> 接受 Number 或其任何父类型。因此，我们可以添加 Integer 和 Double 到列表中。但是，读取数据时只能读取 Object 类型，因为编译器无法确定列表中元素的具体类型。

```
void copy(List<? super String> dest, List<String> src) {
    for (String str : src) {
      dest.add(str);
    }
  }

  List<Object> objList = new ArrayList<>();
  List<String> strList = new ArrayList<>();

  // 可以将 String 类型的列表复制到 Object 类型的列表
  copy(objList, strList);
```

在这个例子中，copy 方法接受一个 List<? super String> 类型的参数。这意味着我们可以将任何可以存储 String 或其父类型的列表传递给该方法。在这个例子中，我们将 String 类型的列表复制到 Object 类型的列表中，这是安全的，因为 Object 是 String 的父类。

总而言之，逆变是一种强大的工具，可以帮助我们编写更加灵活和安全的代码。使用 <? super Type> 语法可以确保写入到泛型类型的数据至少是指定的类型或其父类型，从而提高代码的类型安全性。

17.7 泛型中的类型擦除

17.7.1 什么是类型擦除

Java 泛型的引入是一个巨大的挑战，设计者们需要在保持向后兼容性的同时，引入这一全新的语言特性。由于早期 Java 版本中并未支持泛型，为了确保旧代码能够正常运行，Java 泛型采用了一种称为“类型擦除”（Type Erasure）的技术实现。这意味着，虽然在源代码层面可以定义和使用泛型类型，但在编译后的字节码中，所有泛型信息都会被擦除，泛型类型会被替换为其限定边界或 Object。这就导致了 Java 泛型实际上是”伪泛型”，它并不像 C++ 模板那样在运行时具备真正的类型信息。设计人员在引入泛型的同时，必须平衡向后兼容性与新特性的需求，最终选择了类型擦除这一实现方式，这使得 Java 泛型虽然增强了代码的类型安全性，但在运行时仍缺乏完整的类型信息支持。

17.7.2 JVM 在编译时如何处理泛型

在 Java 源码中使用泛型时，泛型信息只存在于编译阶段。一旦源码被编译成字节码，所有的泛型类型参数信息就会被擦除，JVM 执行的字节码中不包含泛型信息。这意味着：

o 类型擦除发生时，泛型类型参数被替换成了它们的边界类型或者 Object。

o 桥接方法（Bridge Methods）可能会被引入。为了保持多态性，当泛型类型被擦除时，编译器可能会生成桥接方法，以保持子类和父类之间的正确多态行为。

17.7.3 类型擦除对编程的影响

类型擦除确保了泛型的向后兼容性，但它也带来了一些影响和限制：

类型信息的丢失：在运行时，所有的泛型信息都被擦除，这意味着你无法在运行时查询一个泛型的具体类型参数。例如，不能直接判断一个 List 对象是 List 还是 List。

类型强转：因为泛型信息在运行时被擦除，所以编译器在必要时会自动插入类型强转代码。这些强转在源码中是隐式的，但有可能导致 ClassCastException。

泛型方法的重载限制：因为类型擦除，两个仅在泛型参数上有差异的方法会被视为相同的方法签名，因此不能重载。

无法使用泛型类型的实例化和类型比较：由于泛型信息在运行时不可用，所以你不能实例化泛型类型的数组，也不能在运行时进行类型比较（如 if (x instanceof List) 是非法的）。

虽然类型擦除带来了一些限制，但它是 Java 泛型向后兼容旧版本 Java 代码的一种必要手段。理解类型擦除及其对编程的影响，对于编写健壮且有效的 Java 泛型代码是非常重要的。开发者需要在利用泛型带来的编译时类型安全和泛型的灵活性之间，找到一个平衡点。

接下来，我们用实际的例子来看看由于泛型的类型擦除对编程的影响，以数组为例。

17.7.4 泛型与数组

泛型在 Java 中是一个非常强大的特性，它增加了程序的灵活性和类型安全性。然而，当涉及到数组时，泛型的使用受到了一些限制。让我们详细探讨这些限制及其原因。

不能实例化具体类型的泛型数组：Java 不允许直接创建泛型类型的数组，比如 List[]。这是因为泛型信息在运行时会被擦除，而数组需要在运行时知道其元素的具体类型来保证类型安全。这种限制防止了潜在的类型安全问题。

```
// 编译错误：不能创建具体泛型类型的数组
List<Integer>[] arrayOfLists = new List<Integer>[10];
```

一个常见的解决方案是创建一个未经检查的转换，可以使用 @SuppressWarnings(“unchecked”) 注解来抑制编译器警告，并将一个非泛型数组强制转换为泛型数组。但是，这样做需要谨慎，因为这可能会导致运行时类型错误：

```
@SuppressWarnings("unchecked")
List<Integer>[] arrayOfLists = (List<Integer>[])new List[10];
```

虽然不能实例化泛型类型的数组，但可以声明一个泛型数组的引用，这样做通常是为了与泛型代码的其他部分交互。例如，可以使用下面的代码：

```
List<Integer>[] arrayOfLists;
```

上面的代码声明了一个名为 arrayOfLists 的数组引用，该数组的元素类型是 List。这意味着 arrayOfLists 可以指向一个数组，该数组的每个元素都是一个 List 类型的对象。尽管不能直接创建具体泛型类型的数组，但是可以声明一个引用来引用它。

另外，还可以使用 List<List> 这样的集合类来代替泛型数组。这是因为集合类可以存储任何类型的对象，并且可以动态调整大小，更加灵活。因此，在大多数情况下，使用集合类来代替泛型数组是更好的选择，特别是在需要动态调整大小或者与泛型代码的其他部分交互时。

18 函数式编程

Contents

Java 8 引入函数式编程。在此之前，Java 主要是一种面向对象的编程语言，但随着软件开发的复杂性增加，以及并行和分布式计算的普及，Java 社区开始寻找新的编程范式来解决这些问题。

Java 8 在 2014 年发布，这是 Java 历史上的一个重要里程碑，因为它引入了函数式编程的概念和工具。Java 8 的主要目标之一就是使 Java 成为一种“混合”编程语言，既支持面向对象编程，也支持函数式编程。

Java 8 引入了 Lambda 表达式，这是实现函数式编程的关键。Lambda 表达式允许开发者将函数作为一等公民，可以将函数作为参数传递，也可以将函数作为返回值。此外，Java 8 还引入了一系列的函数式接口，如 Function、Predicate、Supplier 和 Consumer 等，以及流（Stream）API，这些都是实现函数式编程的重要工具。

在本章中，我们将逐步深入探讨上述的各个知识点，以便让大家对函数式编程有一个全面而深入的理解。

18.1 第一门函数式编程语言：Lisp 语言

Java 函数式编程的难点之一是它与 Java 的面向对象编程 (OOP) 范式有很大的不同。在 OOP 中，重点是对象和它们之间的交互。而在函数式编程中，重点是函数和数据。这需要开发人员改变思维方式，从面向对象的角度转向函数式角度。

尽管 Java 中的函数式编程颇具挑战性，但在 Lisp 等函数式编程语言中，它却显得轻而易举。学完本章后，可以对比 Java 和 Lisp 在函数式编程方面的差异，你会发现，Java 需要使用函数式接口等技术来实现 Lisp 中非常基础的功能。

Java 中的函数式编程与 Lisp 中的函数式编程相比有很大的差距。Lisp 是一种纯粹的函数式编程语言，而 Java 是一种混合了面向对象和函数式编程范式的语言。这意味着 Lisp 在函数式编程方面比 Java 更强大、更灵活。虽然 Java 的函数式编程功能远不及 Lisp，但是，Java 的函数式编程功能仍然可以帮助开发人员编写更简洁、更易读的代码。

Java 的函数式编程是 Lisp 的一个子集，接下来，我只介绍这部分子集，看看 Lisp 是如何的简单，而 Java 实现的是多么的复杂。当然了，这不止是 Java 的问题，在面向对象编程语言中，包括 Java、Python 都有各种各样的历史包袱，很难像 Lisp 那样轻装上阵。

18.1.1 Lisp 简介

Lisp 是一种历史悠久的编程语言，以其独特的语法和强大的功能而闻名。它是现存第二古老的高级编程语言，仅次于 Fortran。Lisp 最初是为人工智能 (AI) 应用程序开发的，但现在它被用于各种领域，包括 Web 开发、数据分析和游戏开发。

它的作者是约翰 · 麦卡锡（John McCarthy），他是一位美国计算机科学家和认知科学家，被誉为“人工智能之父”。麦卡锡于 1958 年发明了 Lisp，并于 1960 年发表了第一篇关于 Lisp 的论文。

由于 Lisp 的历史很长，在其发展过程中出现了很多方言，最著名的有 Scheme、Common Lisp 和 Racket。甚至有人把 Ruby 这门语言也称为 Lisp 的方言，我觉得有一定的道理。接下来，我用 Racket 这门 Lisp 的方言来简略的介绍一下 Lisp 的语法。

18.1.2 Racket 以及 Racket 的语法

Racket 是一种通用的编程语言，是 Lisp 家族中目前较为活跃的方言之一。其官方网站为 https://racket-lang.org，如果您有兴趣，可以下载官方提供的整套开发工具。如果您只是想了解 Java 实现了哪些函数式编程功能，则无需大费周章，因为 Java 只实现了极小的一部分。接下来，我将简要介绍 Racket 中与 Java 函数式编程相关的一些基本功能。

先从 Racket 最简单的语法开始讲起，先来看下面的代码：

```
;; 定义一个值为10的变量 x
(define x 10)
;; 输出变量 x 的值
(display x)

;; 定义一个名为 square 的函数，求参数的平方
(define (square x) (* x x))
;; 计算2的平方，输出4
(square 2)

;; 定义一个名为 add 的函数，求两个参数的和
(define (add x y) (+ x y))
(add 1 2)
```

我来解释一下，这段代码是用 Racket 语言编写的，Racket 是一种基于 Scheme 的函数式编程语言。下面是对这段代码的解释：

(define x 10)：这行代码定义了一个名为 x 的变量，并将其值设置为 10。

(display x)：这行代码输出变量 x 的值，即 10。

(define (square x) (* x x))：这行代码定义了一个名为 square 的函数，这个函数接受一个参数 x，并返回 x 的平方。

(square 2)：这行代码调用 square 函数，传入的参数为 2，所以它会输出 4（即 2 的平方）。

(define (add x y) (+ x y))：这行代码定义了一个名为 add 的函数，这个函数接受两个参数 x 和 y，并返回它们的和。

(add 1 2)：这行代码调用 add 函数，传入的参数为 1 和 2，所以它会输出 3（即 1+2 的结果）。

为了更好地理解 Racket 语法，我们将简要介绍其核心概念，并与 Java 的 Lambda 表达

式进行比较。需要注意的是，Java 的 Lambda 表达式仅实现了 Racket 语法中的一部分基本概念。

18.1.2.1 基本形式

Racket 使用前缀表示法，函数调用的形式是：(函数名参数 1 参数 2 ...)

例如：(+ 1 2) 计算 1+2 的结果，(+ 1 2 3) 则是计算 1+2+3 的结果。

18.1.2.2 定义变量与函数

在 Racket 中，使用 define 定义各种形式的值绑定，换句话说就是让值与名称的关联。下面是 define 的一些主要用法：

(define x 10) 定义一个值为 10 的变量 x。

(define lst '(1 2 3 4)) 使用' 语法定义一个列表 lst。

(define (square x) (* x x)) 定义了一个名为 square 的函数，参数为 x，返回 x 的平方。

(define double (lambda (x) (* 2 x))) 定义匿名函数 (Lambda)，这里定义了一个计算双倍的函数 double。

18.1.2.3 输出

在 Racket 中，display 是一个内置的过程，相当于 Java 中的 println()，用于将给定的值以外部可读的形式输出到当前输出端口。比如，(display x) 就是输出 x 的值。

Java 就实现了这些东西，Java 跟 Lisp 的设计目标不同，Lisp 最初设计的目标是作为一种符号计算语言，强调元编程能力和可扩展性。而 Java 的设计目标是创建一种简单、面向对象、健壮、跨平台的通用编程语言，更侧重可移植性和性能。因此，在权衡了易学性、性能、生态系统等多方面因素后，Java 设计者有意识地没有完全采用 Lisp 的语法和底层实现思路，而是选择了一种更实用主义的道路，以适应更广泛的受众群体和应用场景需求。这种权衡虽然带来了一定代价，但促进了 Java 的广泛流行。

18.2 Lambda 表达式

18.2.1 什么是 Lambda 表达式

在 Java 中，Lambda 表达式是一种匿名函数，可以将其视为没有名称的代码块。在 Lisp 中也有 Lambda 表达式，那 Java 中的 Lambda 表达式跟 Lisp 中的 Lambda 表达式有什么区别？

有许多区别，但是我认为最重要的区别是：在 Lisp 中，Lambda 表达式可以独立运行，而在 Java 中，Lambda 表达式要和函数式接口结合起来，才能运行。咱们先比较一下这两种语言中 Lambda 表达式的写法，等一会我再来解释为什么 Java 中的 Lambda 表达式要与函数式接口结合起来才可以运行。

在 Racket 中，Lambda 表达式用于创建匿名函数，其基本的语法如下：

```
(lambda (参数列表) 函数体)
```

参数列表是一个由 0 个或多个参数名组成的列表，函数体则描述了 Lambda 表达式的计算过程。

同样，在 Java 中，Lambda 表达式也是用于创建匿名函数，其基本语法如下：

```
(参数列表) -> { 函数体 }
```

这将创建一个匿名函数，该函数接受指定的参数，并计算并返回表达式的值。

这两个语言中 Lambda 表达式差不多，Java 中看起来更简洁一些，在一些 Lisp 方言中。下面我们来看看其具体的应用。

18.2.2 无参数的 Lambda 表达式

在 Racket 中，无参数的 Lambda 表达式的形式如下：

```
(lambda () 表达式)
```

比如，下面这个无参数的表达式总是返回 “hello world”：

```
((lambda () "hello world"))  ; 返回 "hello world"
```

在 Racket 中，表达式 (lambda () “hello world”) 创建了一个匿名函数，这个函数不接受任何参数，无论何时调用，都会返回字符串 “hello world”。

可以用上面代码里的方式调用这个函数：使用两对括号。第一对括号是用来调用函数的，第二对括号是 Lambda 表达式本身。因为 Lambda 表达式创建了一个函数，所以我们可以直接调用它。

或者，可以给这个匿名函数起一个名字叫 hello-world，用上一小节讲的 define 方法：

```
(define hello-world (lambda () "hello world"))
(hello-world)
```

第一行代码使用 define 语句定义了一个新的函数 hello-world。这个函数是由 Lambda 表达式创建的，这个 Lambda 表达式不接受任何参数（参数列表为空），并且它的函数体只是一个字符串 “hello world”，所以这个函数会无条件地返回 “hello world”。

第二行代码调用了刚刚定义的 hello-world 函数。因为 hello-world 函数不接受任何参数，所以我们可以直接调用它，不需要提供任何参数。这个函数调用会返回 “hello world”。

在 Java 中，也有无参数的 Lambda 表达式，其形式如下：

```
() -> 表达式 或者 () -> { 语句; }
```

比如，下面这个无参数表达式总是返回 “hello world”：

```
() -> "Hello, World!"
```

跟 Racket 类似，在 Java 中，用 () -> “hello, world!” 创建了匿名函数，这个函数不接受任何参数，无论何时调用，都会返回字符串 “hello world”。

在 Java 中，如果才能使用这个函数呢？这个与 Racket 有巨大的差别，如果你是初学者，先看看其调用方法，一会儿我们再一起研究为什么会这么复杂。

首先，要使用这个 Lambda 表达式来创建一个 Supplier 对象，Supplier 是一个函数式接口，它代表一个无参数的函数，返回一个结果。示例代码如下：

```
Supplier<String> helloWorld = () -> "Hello, World!";
// 输出 "Hello, World!"
System.out.println(helloWorld.get());
```

在这个例子中，helloWorld 是一个 Supplier 对象，当我们调用它的 get 方法时，它会执行 Lambda 表达式并返回结果。

再强调一下，如果目前看不懂 Java 如何使用 Lambda 表达式没关系，等一会才介绍函数接口。目前仅需要知道想调用没有参数的 Lambda 表达式，需要一个名为 Supplier 的接口。接下来，我们再来看有一个参数的 Lambda 表达式。

18.2.3 有一个参数的 Lambda 表达式

在 Racket 中，写有一个参数的 Lambda 表达式用下面的方式：

```
(lambda (x) 表达式)
```

例如，以下是一个接受一个参数并返回它的平方的 Lambda 表达式：

```
(lambda (x) (* x x))
```

可以使用 define 语句将这个匿名函数赋值给一个变量，然后通过这个变量来调用这个函数：

```
(define square (lambda (x) (* x x)))
(square 5)  ; 返回 25
```

在 Java 中，可以用以下三种方式来写有一个参数的 Lambda 表达式：

```
(x) -> { 表达式 } 或者 (x) -> 表达式  或者  x -> 表达式
```

这三种方式都可以，咱们选个简单的，用后者来实现一个接受一个参数并返回它的平方的 Lambda 表达式：

```
x -> x * x
```

有了前面 Java 调用无参数 Lambda 表达式的经验，想必你应该有心理预期，在 Java 中调用该表达式，肯定不像 Racket 那么简单。

```
Function<Integer, Integer> square = x -> x * x;
System.out.println(square.apply(5));  // 输出 25
```

上面的第一行代码使用 Java 的 Function 接口定义了一个新的函数 square。Function 接口是一个函数式接口，它代表一个接受一个参数并产生一个结果的函数。在这个例子中，square 函数接受一个 Integer 参数 x，并返回 x 的平方（x * x）。这个函数是由 Lambda 表达式 x -> x * x 创建的。

第二行代码调用了刚刚定义的 square 函数，并将结果打印到控制台。square.apply(5) 调用 square 函数并传递参数 5，然后 square 函数返回 5 的平方，即 25。然后，System.out.println 打印这个结果，所以控制台上会输出 25。

这里又引入了一个新的接口 Function<T, R>，先不用理会，再来看看两个参数的 Lambda 表达式应该如何写。

18.2.4 有两个参数的 Lambda 表达式

在 Racket 中，有两个参数的 Lambda 表达式这样写：

```
(lambda (x y) 表达式)
```

以下是一个接受两个参数并返回它们之和的 Lambda 表达式：

```
(lambda (x y) (+ x y))
```

可以使用 define 语句将这个匿名函数赋值给一个变量，然后通过这个变量来调用这个函数：

```
(define add (lambda (x y) (+ x y)))
(add 3 4)  ; 返回 7
```

在 Java 中，有两个参数的 Lambda 表达式语法如下：

```
(x, y) -> 表达式
```

例如，以下是一个接受两个参数并返回它们之和的 Lambda 表达式：

```
(x, y) -> x + y
```

有了前面两次的经验，你应该觉得，这时候，为了调用有两个参数的 Lambda 表达式，Java 肯定又得引入一个接口吧？是的，要创建一个 BiFunction 对象，对能使用有两个参数的接口：

```
BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;
System.out.println(add.apply(3, 4));  // 输出 7
```

在这个例子中，add 是一个 BiFunction 对象，当我们调用它的 apply 方法时，它会执行 Lambda 表达式并返回结果。

通过以上的三个例子，无参数的 Lambda 表达式、有一个参数的 Lambda 表达式和有两个参数的 Lambda 表达式，细心的可以总结出这样的规律：

函数式编程的一个关键概念是将函数（更准确地说，是 Lambda 表达式）赋值给变量。

在 Racket 语言中，函数可以像任何其他值一样被赋值给变量，这与对变量赋值的方式相同。然而，在 Java 中，由于函数不是一等公民，因此需要特殊的处理才能将函数赋值给变量。

接下来，讲一下将 Lambda 表达式赋值给变量这个关键概念。

18.2.5 将 Lambda 表达式赋值给变量

在函数式编程中，函数被视为“一等公民”，这意味着函数可以被像其他普通数据类型（如整数、字符串等）一样使用。具体来说，函数可以被赋值给变量，可以作为参数传递给

其他函数，也可以作为其他函数的返回值。

这种特性使得函数式编程语言非常强大和灵活。例如，你可以创建高阶函数（即接受其他函数作为参数或返回其他函数的函数），这使得你可以轻松地创建复杂的数据处理流程，而无需写大量的重复代码。

但是在 Java 中，所有的方法都必须属于一个类或接口。这是因为 Java 是一种面向对象的编程语言，它的设计理念是“一切皆对象”。在这种设计理念下，方法被视为对象的行为，因此它们必须属于某个类或接口。

这意味着你不能在 Java 中创建独立的函数或方法。即使是主方法（public static void main(String[] args)），它也必须属于一个类。

既然 Java 8 要开始支持函数式编程，那就必须按照函数式编程的规则来。于是，就出现了各种妥协，既要兼顾面向对象，又要兼顾函数式编程。最终的解决方案是：创建一个 Lambda 表达式，并将它赋值给一个函数式接口的变量。然后，可以像调用普通方法一样调用这个函数。但是，这仍然不是真正的“独立函数”，因为这个函数实际上是函数式接口的一个实例。

这个解决方案有两个要点：一是 Lambda 表达式，二是函数式接口。Lambda 表达式前面已经讲了，接下来，再来讲函数式接口。

18.3 函数式接口

18.3.1 为什么需要函数式接口

在 Racket 语言中，可以用如下的方式来给变量赋值：

```
(define x 10)
(define hello-world (lambda () "hello world"))
```

这是因为 Racket 是一种动态类型的编程语言，这意味着你不需要（也不能）在声明变量时指定它的类型。变量的类型是在运行时确定的，而不是在编译时。这使得 Racket 的语法比一些静态类型的语言（如 Java）更简洁，但也意味着失去了静态类型检查的一些优点，如早期错误检测和某些类型的自动补全。

第一行代码定义了一个新的变量 x，并将它初始化为 10。因为 Racket 是动态类型的，所以我们不需要（也不能）指定 x 的类型。

第二行代码定义了一个新的变量 hello-world，并将它初始化为一个 lambda 表达式。因为 Racket 是动态类型的，所以我们不需要（也不能）指定 hello-world 的类型。

Java 语言要求在定义变量时必须指定其类型，例如 int、String 或自定义类。然而，将

Lambda 表达式直接赋值给变量时会出现问题，因为 Java 中没有专门用于表示函数的类型。为了解决这个问题，Java 引入了函数式接口的概念。函数式接口是一种只有一个抽象方法的接口，专门用于为 Lambda 表达式提供类型。

接下来，我们分别讲函数式接口的类型以及为什么只能有一个抽象方法，先从函数式接口的类型讲起。

18.3.2 函数式接口的种类有多少

函数式编程就像是在做数学运算。在数学中，你有一个公式，给定一些输入，这个公式就会产生一个结果，而且这个结果只依赖于输入的值。

我们在学习数学的时候，会这样来分类函数：一元一次方程，二元一次方程，二元二次方程……是根据输入的参数有多少。比如现在的大语言模型，参数高达数十亿，数百亿，也许应该叫十亿元多少次方程吧。

Java 8 在 java.util.function 包中引入了一些常用的函数式接口，也用类似数学的方法来给函数分类，依据是输入参数与返回结果的个数，以下是一些主要的函数式接口：

o Function<T,R>：接受一个输入参数，返回一个结果。

o BiFunction<T,U,R>：接受两个输入参数，返回一个结果。

o Consumer：接受一个输入参数并执行一些操作，没有返回结果。

o BiConsumer<T,U>：接受两个输入参数并执行一些操作，没有返回结果。

o Predicate：接受一个输入参数，返回一个布尔值结果。

o BiPredicate<T,U>：接受两个输入参数，返回一个布尔值结果。

o Supplier：不接受输入参数，返回一个结果。

o UnaryOperator：接受一个参数为类型 T，返回值类型也为 T 的函数对象。

o BinaryOperator：接受两个参数为类型 T，返回值类型也为 T 的函数对象。

这些只是最常用的一些函数式接口，实际上 Java 还提供了其他一些函数式接口，如 IntFunction，LongFunction，DoubleFunction 等等，这些函数式接口用于处理特定类型的函数。

此外，还可以创建自己的函数式接口，只要确保它只定义了一个抽象方法即可。为什么只能有一个抽象方法呢？接下来，我们再来讲这个问题。

18.3.3 函数式接口为何只能有一个抽象方法?

在 Java 中，一个 Lambda 表达式只能表示一个单一的函数，因此，函数式接口只能有一个抽象方法。如果一个接口定义了多个抽象方法，那么它就不能被用作函数式接口。

如果尝试将一个 Lambda 表达式赋值给这样的接口，编译器会报错，因为它无法确定这个 Lambda 表达式应该对应哪个抽象方法。

注意，函数式接口可以有多个默认方法或静态方法，只要它们不是抽象的就可以。这是因为默认方法和静态方法都有实现，所以它们不会影响 Lambda 表达式或方法引用的对应关系。

在实现函数式接口的时候，通常使用 Lambda 表达式。同时，Java 引入了方法引用来作为 Lambda 表达式的有效补充，通过方法引用，程序员可以更直接地引用已有的方法，并将其无缝融入函数式编程上下文中，如同 Lambda 表达式一样传递和使用。接下来，我们再来学习一下方法引用。

18.3.4 方法引用

除了编写 Lambda 表达式之外，您还可以将方法引用传递给函数式接口。方法引用提供了一种更简洁的方式来引用现有方法，并将其作为 Lambda 表达式使用。简而言之，方法引用可以用来替代某些 lambda 表达式，使代码更简洁、更易读。

例如，以下代码使用 Lambda 表达式将 String::toUpperCase 方法引用传递给 Function 接口：

```
Function<String, String> toUpperCase = String::toUpperCase;
```

上面的代码跟下面的 Lambda 表达式等效：

```
Function<String, String> toUpperCase = str -> str.toUpperCase();
```

方法引用有以下三种类型：

o 静态方法引用：如果你需要引用一个静态方法，你可以使用类名:: 方法名的形式。例如，Math::sqrt 是 Math.sqrt(x) 的方法引用。

o 实例方法引用：如果你需要引用一个特定对象的实例方法，你可以使用实例名:: 方法名的形式。例如，假设你有一个 String 对象 str，str::length 是 str.length() 的方法引用。

o 构造方法引用：如果你需要引用一个类的构造方法，你可以使用类名::new 的形式。例如，ArrayList::new 是 ArrayList 的无参数构造方法的引用。

使用方法引用可以使代码更简洁、更易读，特别是当你的 lambda 表达式只是调用一个已经存在的方法时。

18.3.5 自定义函数式接口

在函数式编程范式中，函数被视为一等公民，可以像其他值一样被传递和操作。如果 Java 预定义的函数式接口无法满足您的需求，可以创建自定义的函数式接口。通过自定义函数式接口，可以增强函数的抽象性，使其更易于组合和操作，从而提高代码的可重用性和可读性。接下来，我将通过下面的代码来演示一下：

```
// 自定义函数式接口示例
@FunctionalInterface
public interface MyFunction {
    int apply(String input);
}
```

在上面这段代码中，@FunctionalInterface 是 Java 8 引入的一个注解，它用来指示一个接口应该被视为函数式接口。

@FunctionalInterface 注解不是必需的，可以不使用它，只要接口只有一个抽象方法，它就是一个函数式接口。然而，使用 @FunctionalInterface 注解有两个好处：

o 它可以清楚地表明意图，说明设计这个接口就是为了它能被用作函数式接口。

o 它可以启动编译器的检查，如果接口不符合函数式接口的要求（例如，它有多个抽象方法），编译器会报错。

在上面的代码中，MyFunction 接口被标记为 @FunctionalInterface，这意味着它应该只有一个抽象方法。实际上，它确实只有一个抽象方法 apply，所以它是一个有效的函数式接口。

总之，上面的代码片段定义了一个名为 MyFunction 的函数式接口。这个接口有一个名为 apply 的抽象方法，该方法接受一个 String 类型的参数并返回一个 int 类型的结果。接下来，代码清单 18-3 使用这个接口作为参数类型，并在一个方法中传递 Lambda 表达式来实现其功能。

```
public class FunctionExample {
    // 方法接受一个实现了 MyFunction 接口的 lambda 表达式
    public static void processString(MyFunction function,
                                     String text) {
        System.out.println("Processing: " + text + ",
                           result is: " + function.apply(text));
    }
```

```
    public static void main(String[] args) {
        // 使用自定义函数式接口的 lambda 表达式
        // 计算输入字符串的长度
        processString((input) -> input.length(),
                                    "Hello, World!");

        // 或者使用方法引用来实现
        // 同样计算字符串长度
        processString(String::length, "Java Rocks!");
    }
}
```

这段代码定义了一个名为 FunctionExample 的类，该类中包含两个静态方法：processString 和 main。

processString 方法接受两个参数：一个是实现了 MyFunction 接口的对象（在这里，我们期望传入一个 lambda 表达式），另一个是一个 String 对象。这个方法会调用 MyFunction 对象的 apply 方法处理字符串，然后打印出原始字符串和处理结果。

main 方法是程序的入口点。在这个方法中，我们调用了两次 processString 方法，每次都传入了一个 lambda 表达式（或方法引用）和一个字符串。

第一次调用 processString 方法时，我们传入了一个 lambda 表达式 (input) -> input.length() 和字符串 “Hello, World!”。这个 lambda 表达式实现了 MyFunction 接口，它接受一个字符串作为输入，返回这个字符串的长度。

第二次调用 processString 方法时，我们传入了一个方法引用 String::length 和字符串 “Java Rocks!”。这个方法引用也实现了 MyFunction 接口，它引用了 String 类的 length 方法，这个方法接受一个字符串作为输入，返回这个字符串的长度。

因此，当运行这段代码时，它会打印出以下内容：

```
Processing: Hello, World!, result is: 13
Processing: Java Rocks!, result is: 11
```

18.4 Stream API

18.4.1 什么是 Stream

Stream 是 Java 8 中引入的一种新的数据处理抽象。它的主要目标是提供一种高效且易于使用的工具，用于处理数据集合，特别是对于需要大量计算的数据集合。

Stream 的核心思想是，在数据集合上执行一系列的操作，每个操作都会生成一个新的 Stream，然后可以将这些操作链接在一起，形成一个操作管道。这种方式被称为“流式”编程或“管道”编程。仅从“Stream”这个英文单词的意义上来考虑，数据就像河流一样，我们可以在这条河上修建一个一个的拦水坝，每个拦水坝相当于一个操作，可以改变水流的方向或者对其进行过滤。

如果熟悉 Unix/Linux，应该对“管道”不陌生。Unix 的“管道”是将一个进程的输出直接作为另一个进程的输入，这种机制允许多个进程以管道的方式连接起来，形成一个数据处理链。用户可以将一个复杂的任务分解为多个简单的任务，然后将这些任务链接在一起。这种方式的优点是，每个任务都很简单，易于理解和测试。此外，由于每个任务都是一个独立的进程，所以可以利用多核处理器的能力来并行执行这些任务，从而提高性能。

Unix 的“管道”跟 Java 的 Stream 有异曲同工之妙。

例如，在 Java 中，可以创建一个 Stream，然后在这个 Stream 上执行 filter 操作来选出满足某个条件的元素，然后执行 map 操作来对每个元素进行某种转换，最后执行 reduce 操作来生成一个结果。

与 Unix 的“管道”不同，Java 中的 Stream 的操作是惰性的。这意味着在 Java 中，只有在需要结果时才 Stream 才会执行，而 Unix 中，只要创建了命令，就会开始执行。

18.4.2 创建 Stream 的各种方式

什么样的数据类型可以被创建为 Stream 呢？一句回答：任何类型。无论集合、数组、文件、网络数据还是自定义数据，都可以在 Java 中创建为 Stream。接下来，分别就常见的数据类型来演示一下如何创建 Stream。

18.4.2.1 从 Collection 中创建 Stream

Java 8 引入了新的方法到 java.util.Collection 接口中，这意味着任何实现了这个接口（如 ArrayList、LinkedList、HashSet 等）的对象都可以直接生成 Stream。示例代码如下：

```
// 假设有一个 List 实例
List<String> list = Arrays.asList("A", "B", "C", "D");

// 创建一个 Stream
Stream<String> stream = list.stream();

// 或者创建一个 parallelStream 并行流（适用于大量数据并行处理）
Stream<String> parallelStream = list.parallelStream();
```

18.4.2.2 从数组中创建 Stream

Java 8 提供了 java.util.Arrays 类的静态方法来将数组转换为 Stream。

```
// 假设有一个数组
String[] array = {"E", "F", "G", "H"};

// 使用 Arrays.stream() 方法将数组转换为 Stream
Stream<String> arrayStream = Arrays.stream(array);
```

18.4.2.3 从文件中创建 Stream

在 Java 中，可以使用 java.nio.file.Files 类提供的静态方法从文件中创建字节流或字符流。下面是如何从文件创建字符流，通过读取文件的每一行作为一个字符串元素的 Stream，示例代码如下：

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class FileToStreamExample {
    public static void main(String[] args)
                                    throws IOException {
        // 文件路径
        String filePath = "/path/to/your/file.txt";

        try (Stream<String> lines = Files.lines(
                                Paths.get(filePath))) {
            // 对文件中的每一行进行处理
            long lineCount = lines.count();
            System.out.println("Line count: " + lineCount);

            // 或者对每行应用其他操作，例如打印所有行
            lines.forEach(System.out::println);
        } catch (IOException e) {
            System.err.println("Error reading file: "
                                        + e.getMessage());
```

```
            }
        }
    }
```

Files.lines() 方法会返回一个包含文件所有行的 Stream，也就是说，当遍历这个 Stream 时，每次迭代获取到的是文件中的一行内容，而并非一个针对单个字符或子串的 Stream。该方法还会自动处理编码问题，使用系统默认的字符集（可以通过可选参数指定特定字符集）。当处理完流之后，由于使用了 try-with-resources 语句，底层资源会被自动关闭，确保了资源的有效管理。

以上是三种常见的创建 Stream 的方式，在创建 Stream 之后，就要对 Stream 进行操作，接下来，介绍 Stream 的常见操作。

18.4.3 Stream 操作的分类

Java Stream API 提供了一种高效且简洁的方法来处理集合数据。它由两部分操作组成：中间操作（Intermediate Operations）和终端操作（Terminal Operations）。中间操作是构建流的操作，它们的特点是惰性执行的，即不会立即执行，只有在遇到终端操作时才执行。中间操作的结果还是一个流，这允许多个中间操作可以连接起来，形成一个流水线。常见的中间操作包括 filter、map、sorted 等。

假设我们有一个 List，包含多个字符串，我们希望筛选出其中长度大于 3 的字符串，并将它们转换为大写。这个需求可以通过中间操作 filter 和 map 来实现。

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Java",
                "Stream","Filter", "Map", "Example");

        // 将集合转换成流
        List<String> result = words.stream()
                // 中间操作：筛选长度大于 3 的字符串
                .filter(word -> word.length() > 3)
                // 中间操作：将字符串转换为大写
                .map(String::toUpperCase)
```

```
                // 终端操作：收集结果到一个 List 中
                .collect(Collectors.toList());

        System.out.println(result); // 输出结果
    }
}
```

在这个例子中，filter(word -> word.length() > 3) 是一个中间操作，它通过传入的 Lambda 表达式筛选出长度大于 3 的字符串。

map(String::toUpperCase) 也是一个中间操作，它将流中的每个元素转换成大写。

collect(Collectors.toList()) 是一个终端操作，它触发了流的执行，并将结果收集到一个新的 List 中。运行之后，输出如下的结果：

```
[JAVA, STREAM, FILTER, EXAMPLE]
```

这个例子展示了如何通过流的中间操作来进行数据的处理和转换，而不需要显式地编写循环或条件判断，这种方式使代码更加简洁且易于理解。接下来，详细讲解一下什么是中间操作，什么是终端操作。

在 Java 的 Stream API 中，流操作分为两种基本类型：中间操作（Intermediate Operations）和终端操作（Terminal Operations）。这两类操作共同构成了流的处理流水线，但它们在功能和行为上有着根本的区别。

18.4.3.1 中间操作（Intermediate Operations）

中间操作用于转换流，它们接收一个流作为输入，然后返回一个新的流作为输出。这允许多个中间操作可以连接起来，形成一个查询的链条。中间操作是惰性的，意味着它们不会立即执行。只有当一个终端操作被触发时，中间操作才会执行。这种设计允许优化操作，例如延迟执行和短路。

常见的中间操作包括 filter（筛选）、map（映射）、sorted（排序）、distinct（去重）、limit（限制）、skip（跳过）等。

18.4.3.2 终端操作（Terminal Operations）

终端操作会从流的流水线生成结果。当一个终端操作执行时，它会触发前面的中间操作，并处理数据。一旦终端操作执行，流就被消费掉了，不能再被使用。

终端操作包括 forEach（遍历）、collect（收集到集合）、reduce（归纳）、findAny/findFirst（查找）、match（匹配）、count（计数）等。它们通常会返回一个非流的值，比如一个列表、一个整数，或者一个布尔值，有的甚至不返回任何内容（例如 forEach）。

18.4.3.3 中间操作与终端操作的区别

执行时机不同：中间操作是惰性的，仅仅定义了操作的内容，不会立即执行。只有当遇到终端操作时，中间操作才会被实际执行。而终端操作会立即触发流的处理。

返回类型不同：中间操作返回一个新的流，允许链式调用。终端操作则通常返回一个具体的值或者不返回值，并且它会结束流的链式调用。

使用次数不同：流一旦经过终端操作处理，就会被" 消费" 掉，不能再次使用。中间操作可以在流上多次应用，因为它们每次都会返回一个新的流。

执行目的不同：中间操作主要用于数据的转换和过滤，而终端操作用于产生一个结果，这个结果可能是一个值（比如总数、最小值）、一个容器（比如列表、映射）或者是一个动作（比如打印出每个元素）。

通过中间操作和终端操作的组合使用，Stream API 提供了一种高效、表达性强的方式来处理 Java 中的集合数据。接下来，我们再来学习一下两种操作中常见的操作类型。

18.4.4 常见的中间操作

Java 的 Stream API 中，中间操作可以大致分为以下几种类型：

18.4.4.1 筛选和切片（Filtering and Slicing）类型的操作

在 Java 的 Stream API 中，“筛选与切片” 是指一组中间操作，它们用于从流中选择某些元素，或减少流中元素的数量。这些操作对于处理集合数据时只关注部分元素非常有用。具体的筛选与切片操作包括：

o filter(Predicate)：这个操作会使用一个 Predictor（即一个返回布尔值的函数）来测试流中的每个元素。只有通过测试（即函数返回 true）的元素才会被包含在结果流中。这是一个非常常用的操作，用于从流中筛选出满足特定条件的元素。

o distinct()：这个操作会返回一个流，其中的元素是去除了重复元素后的集合。它依据元素的 equals() 方法来判断元素是否重复。distinct() 操作可以用于去除数据集中的重复项。

o limit(long maxSize)：这个操作会截取流使其最大长度不超过给定的数值。如果流的当前长度小于 maxSize，则其不会受到影响。这个操作常用于限制结果数量或者对流进行分页处理。

o skip(long n)：这个操作会丢弃流的前 n 个元素，如果流中的元素不足 n 个，则返回一个空流。skip() 操作在你需要跳过前面一些不需要的元素时非常有用。

以下的代码是一个使用 filter、distinct、limit 和 skip 操作的示例，展示了如何筛选出一组字符串中长度大于 3 的不重复字符串，并只获取结果的前两个。

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class FilteringAndSlicingExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList(
                "apple", "banana", "cherry",
                "apple", "date", "egg", "fig");

        List<String> filteredWords = words.stream()
                // 筛选长度大于 3 的字符串
                .filter(word -> word.length() > 3)
                .distinct() // 去重
                .limit(2) // 限制结果数量为 2
                // 收集到 List 中
                .collect(Collectors.toList());

        // 输出结果是 [apple, banana]
        System.out.println(filteredWords);
    }
}
```

这段 Java 代码是一个简单的示例，展示了如何使用 Java 8 的 Stream API 进行过滤和切片操作。

首先，创建了一个包含多个字符串的列表 words。然后，对这个列表调用.stream() 方法，将其转换为一个 Stream。接下来，在这个 Stream 上进行了一系列的中间操作：

o filter(word -> word.length() > 3)：这个操作会过滤掉长度不大于 3 的字符串。

o distinct()：这个操作会去除重复的元素。在这个例子中，它会去除重复的字符串。

o limit(2)：这个操作会限制结果的数量。在这个例子中，它会限制结果的数量为 2。

最后，使用 collect(Collectors.toList()) 将 Stream 转换回一个列表，并将结果赋值给 filteredWords。

18.4.4.2 映射（Mapping）类型的操作

在 Java 的 Stream API 中，映射是一类中间操作，用于将流中的元素转换或映射成另一种形式的元素。映射操作依赖于提供的函数，对流中的每个元素进行处理，然后将处理的结果组成一个新的流。这对于数据转换非常有用。主要的映射操作包括：

o map(Function<T, R>)：接收一个函数作为参数，这个函数会应用到每一个元素上，并将其映射成一个新的元素。这是最常用的映射操作，适用于转换元素类型或修改元素。

o flatMap(Function<T, Stream>)：与 map 操作类似，接收一个函数作为参数，这个函数应用到每一个元素上，不同的是，每个元素会被转换成一个流，然后再将所有创建的单个流合并成一个流。这个操作通常用于处理嵌套结构的流，如 Stream<List>。这个解释是学术上的解释，如果用通俗的解释，用一个成语就够了：百川会海。意思是条条江河都流归大海。

以下是 map 操作的演示代码，假设有一个字符串列表，现在想将列表中的每个字符串都转换为其对应的长度。接下来，演示映射类型 map 的操作：

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class MapExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Java", "Go",
                                           "C++", "Python");
        List<Integer> wordLengths = words.stream()
                .map(String::length) // 将每个字符串映射成其长度
                .collect(Collectors.toList());

        // 输出: [4, 2, 3, 6]
        System.out.println(wordLengths);
    }
}
```

上面的代码太简单了，注释已经给出了完整的解释，在此不再赘述。

接下来，再来展示 flatMap 的操作，考虑一个一个二维字符串列表，我们想要将这个嵌套列表“平坦化”成一个单一列表，使其包含所有的字符串，下面的代码用来展示映射类型 flatMap 的操作。

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {
        List<List<String>> listOfLists = Arrays.asList(
                Arrays.asList("Java", "Go"),
                Arrays.asList("Javascript", "C++")
        );
        List<String> flatList = listOfLists.stream()
                // 将每个内部列表转换成流，然后“平坦化”成一个流
                .flatMap(List::stream)
                .collect(Collectors.toList());

        // 输出: [Java, Go, Javascript, C++]
        System.out.println(flatList);
    }
}
```

具体来说，flatMap() 操作接受一个函数，这个函数应用于原始流中的每个元素，将每个元素转换成一个新的流。然后，flatMap() 操作将所有这些新生成的流中的元素合并到一个单一的流中，这样，原来分散在多个流中的元素就被汇集到了一个流里。

这个操作尤其有用于处理嵌套的集合结构，比如列表中的列表，或者是对流中的每个元素进行某种转换，而这种转换本身又产生了流（例如，字符串转换成字符流，列表转换成元素流等）。flatMap() 通过 “扁平化” 这些生成的流，简化了处理复杂数据结构的复杂度。

18.4.4.3 排序类型的操作

o sorted()：这个操作会按照元素的自然顺序进行排序。要求流中的元素类型实现了 Comparable 接口。如果元素本身没有实现 Comparable 接口，或者希望按照元素的自然顺序以外的其他方式进行排序，那么这个方法就不适用。

o sorted(Comparator<? super T> comparator)：这个操作允许提供一个自定义的 Comparator 比较器，根据比较器的逻辑对流中的元素进行排序。这提供了排序操作的极大灵活性，允许按照任意属性或条件对元素进行排序。接下来，分别演示一下这两个操作的用法，先演示 sorted 方法对字符串进行自然排序的例子：

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class SortingExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("banana",
                                "apple", "cherry", "date");
        List<String> sortedWords = words.stream()
                .sorted() // 自然排序
                .collect(Collectors.toList());

        // 输出: [apple, banana, cherry, date]
        System.out.println(sortedWords);
    }
}
```

在 Java 中，自然排序（Natural Ordering）是指按照对象的自然顺序进行排序。对于 Java 内置的类型，自然顺序如下：

o 对于 String 类型，自然顺序是按照字母表顺序（ASCII 码顺序）。

o 对于 Integer、Long、Short、Byte、Double、Float 等数值类型，自然顺序是按照数值大小。

o 对于 Character 类型，自然顺序是按照 Unicode 值。

o 对于 Boolean 类型，自然顺序是 false 在 true 之前。

对于自定义的类型，如果该类型实现了 Comparable 接口，那么 compareTo 方法定义的就是其自然顺序。例如，如果有一个 Person 类，它实现了 Comparable 接口，并在 compareTo 方法中按照人的年龄进行比较，那么对 Person 对象列表进行自然排序，就会按照人的年龄进行排序。

由于代码中是对英文 String 类型排序，所以按照字母表的表的顺序来排序。接下来，演示自定义比较器的方法，用字符串长度来对其进行排序。

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class CustomSortingExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList(
            "banana", "apple", "cherry", "watermelon");
        List<String> sortedByLengthWords = words.stream()
                // 根据字符串长度排序
                .sorted(Comparator.comparingInt(
                                    String::length))
                .collect(Collectors.toList());

        // 输出是: [apple, banana, cherry, watermelon]
        System.out.println(sortedByLengthWords);
    }
}
```

这两个例子展示了如何使用 Stream API 中的排序操作，对流中的元素进行自然排序或者根据自定义的比较器进行排序。这些操作使得对集合中的数据进行排序变得非常简单和灵活。

18.4.5 常见的终端操作

在 Java Stream API 中，终端操作是那些触发流处理的操作。与中间操作不同，终端操作会消耗流，并产生一个结果或副作用。一旦流进行了终端操作，它就不能再被使用了。终端操作主要分为几个类别，包括 “查找与匹配”、“聚合”、“遍历” 和 “收集” 等。接下来，分别介绍以下是一些常见的终端操作。

18.4.5.1 查找与匹配（Find and Match）

在 Java Stream 中，查找与匹配操作用于检查流中的元素是否符合某些条件。这类操作主要基于 Predicate 逻辑（即返回布尔值的函数）来执行，包括：

o anyMatch(Predicate<? super T> predicate)：检查流中是否至少有一个元素满足给定的条件。如果流是空的，则返回 false。

o allMatch(Predicate<? super T> predicate)：检查流中的所有元素是否都满足给定条件。如果流是空的，也返回 true，这是基于逻辑上的 “空集合普遍性质”（即空集合中的元素默认满足任何条件）。

o noneMatch(Predicate<? super T> predicate)：检查流中是否没有任何元素满足给定条件。如果流是空的，则返回 true。

这些操作都会返回一个 boolean 值，用于表示流中的元素是否符合特定的条件。它们都是” 短路” 操作，意味着在确定结果后，操作会立即结束，而不需要处理整个流。

假设我们有一个员工列表，我们想要检查是否满足以下条件：至少有一个员工年龄超过 30 岁；所有员工的年龄都超过 18 岁；没有员工的姓名以”J” 开头。代码如下所示：

```
import java.util.Arrays;
import java.util.List;

public class MatchExample {
    static class Employee {
        String name;
        int age;

        Employee(String name, int age) {
            this.name = name;
            this.age = age;
        }
    }

    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
                new Employee("Alice", 34),
                new Employee("Bob", 28),
                new Employee("Charlie", 24),
                new Employee("Diana", 42));

        boolean hasOver30 = employees.stream()
                .anyMatch(e -> e.age > 30);
        // 输出: true
        System.out.println(" 有超过 30 岁的员工么? "
                                            + hasOver30);

        boolean allOver18 = employees.stream()
                .allMatch(e -> e.age > 18);
        // 输出: true
        System.out.println(" 所有的员工都年满 18 岁了么? "
                                            + allOver18);
```

```
        boolean noneStartsWithJ = employees.stream()
                .noneMatch(e -> e.name.startsWith("J"));
        // 输出: true
        System.out.println(" 没名字以字母 J 开头的么? "
                                        + noneStartsWithJ);
    }
}
```

这个示例展示了如何使用查找和匹配操作来对集合进行逻辑检查，这些操作对于快速检查集合数据是否满足特定条件非常有用。

18.4.5.2 聚合操作（Aggregate Operations）

聚合操作是 Java Stream API 中的终端操作，用于将流中的元素组合成一个单一的结果。这些操作非常适合进行数值汇总、查找最大最小值或将所有元素合并成一个值。下面是聚合操作的一些详细介绍：

o count() 操作返回流中元素的数量。这是一个非常直接的操作，经常用于快速获取流中有多少个元素。

o max(Comparator) 和 min(Comparator) 操作分别用于找出流中的最大值和最小值。这两个操作都需要一个 Comparator 作为参数，以定义什么构成 “最大” 或 “最小”。

o reduce(BinaryOperator) 操作可以将流中所有元素重复结合起来，最终得到一个值。这个操作接受一个 BinaryOperator 作为参数，这是一种特殊的 BiFunction，接受两个同类型的参数，返回一个可能是相同类型的结果。reduce 操作可以用于求和、求乘积、找最大值等。

假设我们有一个整数列表，我们想要：计算列表中有多少个元素；找出最大的数字；找出最小的数字；计算所有数字的总和。接来下，演示聚合操作的例子：

```
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

public class AggregateOperationsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(
                    1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
        // 计算元素数量
        long count = numbers.stream().count();
        System.out.println("Count: " + count);

        // 找出最大值
        Optional<Integer> max = numbers.stream()
                                .max(Comparator.naturalOrder());
        max.ifPresent(m -> System.out.println("Max: " + m));

        // 找出最小值
        Optional<Integer> min = numbers.stream()
                                .min(Comparator.naturalOrder());
        min.ifPresent(m -> System.out.println("Min: " + m));

        // 计算总和
        int sum = numbers.stream().reduce(0, Integer::sum);
        System.out.println("Sum: " + sum);
    }
}
```

这个例子展示了如何使用 Java Stream API 的聚合操作来对一个整数列表进行分析，提取统计信息。这些操作是处理集合数据时的强大工具，简化了代码，提高了可读性。

18.4.5.3 遍历操作（Traversal Operations）

在 Java Stream API 中，遍历是一个常见的终端操作，用于对流中的每个元素执行某些操作。遍历流通常是为了产生副作用（比如打印输出、修改外部变量的值等），因为遍历本身不返回结果。最常用的遍历操作是 forEach。

forEach 操作接受一个 Consumer 函数式接口作为参数，这个 Consumer 对流中的每个元素执行操作。它是一个终端操作，意味着它会处理流并消耗掉流。

接下来，用下面的代码来展示，假设有一个员工列表，想要打印出每个员工的名字。

```
import java.util.Arrays;
import java.util.List;

public class StreamTraversalExample {
    static class Employee {
```

```
        String name;
        int age;

        Employee(String name, int age) {
            this.name = name;
            this.age = age;
        }
    }

    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
                new Employee("Alice", 30),
                new Employee("Bob", 35),
                new Employee("Charlie", 25)
        );

        // 使用 forEach 遍历流
        employees.stream()
                .forEach(employee ->
                    System.out.println(employee.name));
    }
}
```

在这个例子中，我们创建了一个 Employee 类的实例列表，并使用 stream().forEach() 对其进行遍历。对于列表中的每个 Employee 对象，我们使用 System.out.println 打印出其 name 属性。

forEach 是处理流时执行副作用操作的一个直接方式，但是要注意，由于流的并行操作，forEach 操作的执行顺序并不保证与流中元素的顺序相同。如果需要保持顺序，可以使用 forEachOrdered 方法。

18.4.5.4 收集操作（Collect Operations）

Java Stream API 的 collect 方法是一个终端操作，它可以把流中的元素累积成一个结果，通常是一个集合，如 List、Set 或 Map。它通过一个 Collector 接口实现来指定累积过程。Collector 接口包含了一系列方法来执行如下操作：建立新的结果容器（supplier 方法）、将元素添加到结果容器（accumulator 方法）、合并两个结果容器（combiner 方法），以及对结果容器应用最终转换（finisher 方法）。

Java 在 java.util.stream.Collectors 类中提供了一系列预定义的收集器，可以通过这些工厂方法来方便地创建常用的收集器。

下面的代码是一个使用 Stream 的 collect 方法的示例，它展示了如何将一个字符串列表中的所有字符串连接成一个单独的字符串，并同时演示了如何将字符串列表收集到一个 Set 中来去除重复项。

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamCollectExample {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("apple",
                        "banana", "cherry", "apple", "date");

        // 使用 joining 收集器连接字符串
        String concatenatedString = strings.stream()
                .collect(Collectors.joining(", "));
        System.out.println(" 连接字符串: "
                                + concatenatedString);

        // 使用 toSet 收集器去除重复项并收集到 Set
        Set<String> setOfStrings = strings.stream()
                .collect(Collectors.toSet());
        System.out.println(" 字符串集合 (Set): "
                                + setOfStrings);

        // 更复杂的操作，比如分组
        // 分组收集器 groupingBy，按照字符串长度分组
        var groupedByLength = strings.stream()
          .collect(Collectors.groupingBy(String::length));
        System.out.println(" 按长度分组: " + groupedByLength);
    }
}
```

在这个例子中，首先使用 Collectors.joining(“,”) 方法将字符串列表连接成一个由逗号

分隔的单一字符串。然后，使用 Collectors.toSet() 方法将字符串列表收集到一个 Set 中，自动去除了重复的元素（“apple”）。

最后，通过 Collectors.groupingBy(String::length) 方法按字符串长度进行分组，这展示了 collect 操作可以执行复杂的收集逻辑，如分组和分区。

这个例子展示了 collect 操作的强大功能，它是处理流中数据进行汇总、分组、连接等操作的关键。以上代码运行之后，结果如下：

```
连接字符串: apple, banana, cherry, apple, date
字符串集合(Set): [banana, date, apple, cherry]
按长度分组: {4=[date], 5=[apple, apple], 6=[banana, cherry]}
```

19 枚举类型

Contents

作为 Java 5 中引入的一项新特性，枚举类型（enum）填补了 Java 语言长期以来的空白。在此之前，C++ 和 Python 等语言已经支持枚举类型，这使得 Java 开发人员对枚举功能的呼声日益高涨。在 Java 5 之前，开发者通常使用一组常量来模拟枚举类型，但这种方法缺乏类型安全性和代码可读性，容易导致错误。

Java 5 引入的枚举类型为语言本身带来了强大的枚举支持。枚举类型提供了一种新的数据类型，可以为一组常量值定义一个新的类型，从而提高代码可读性和健壮性。Java 枚举在设计时特别注重类型安全和避免命名冲突，与 C++ 和 Python 相比，它提供了更强的类型约束和封装性。

Java 枚举类型是由 Joshua Bloch 设计的，在本书中，我已经多次提到他了，他是 Java 语言架构师之一，也是许多 Java 核心库的设计者，是被称为 “Java 之母” 的人。Bloch 在 2001 年向 Java 社区建议添加枚举类型，并最终在 Java 5 中实现了这一功能。枚举类型的引入极大地提高了 Java 代码的类型安全性和可读性。

接下来，让我们从易到难一起深入了解 Java 枚举类型。

19.1 枚举概述

19.1.1 什么是枚举

在 Java 中，枚举是一种特殊的数据类型，用于表示一组固定的、预定义的常量值。它可以用来替代传统的常量定义方式，例如使用 public static final 修饰的常量，并提供更强的类型安全性和代码可读性。

枚举类型可以看作是一个特殊的类，它包含了一组预定义的实例，每个实例都代表一个特定的常量值。这些实例被称为枚举常量，它们在枚举类型中是唯一的，并且不能被修改。比如，我们可以使用一个枚举类型来表示一周中的每一天。

```
public enum DayOfWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

在这个例子中，我们定义了一个名为 DayOfWeek 的枚举类型，它包含了七个枚举常量，分别代表星期的七天。

19.1.2 枚举和常量的区别

前文提过，Java 引入枚举之前，开发人员采用常量的方式来模拟枚举类型。以三元色为例，在引入枚举类型之前，开发人员可能采用如下的方式来表示三元色：

```
public static final String RED = "red";
public static final String GREEN = "green";
public static final String BLUE = "blue";
```

但是，如果使用枚举，就可以更加优雅和安全地表示三元色：

```
public enum Color {
    RED,
    GREEN,
    BLUE
```

```
}
```

使用枚举类型来表示三元色相比使用常量有以下几个优点：

类型安全性: 枚举类型可以确保变量只能保存预定义的枚举常量值，避免了使用常量时可能出现的类型错误。例如，使用常量时，可以将一个 String 类型的变量赋值给表示颜色的常量，这会导致潜在的错误。而使用枚举类型，编译器会强制要求变量的类型必须是 Color，从而避免了这种错误。

代码可读性: 枚举类型的名称可以清晰地表达常量的含义，使代码更易于理解和维护。例如，Color.RED 比 “RED” 更能清晰地表达颜色的含义。

避免命名冲突: 枚举类型可以避免不同常量之间的命名冲突，因为每个枚举常量都在枚举类型中是唯一的。例如，如果在不同的类中定义了名为 “RED” 的常量，就可能发生命名冲突。而使用枚举类型，每个枚举常量都是唯一的，不会发生命名冲突。

其他功能: 枚举类型还可以定义方法、属性和构造函数，使其更加灵活和强大。例如，可以为 Color 枚举类型定义一个方法，用于获取颜色的 RGB 值。

总而言之，与使用常量相比，枚举类型为三元色的表示提供了更优雅、安全和灵活的解决方案。现在，让我们深入探讨枚举类型的技术细节。

19.2 定义枚举

Java 中枚举的语法结构如下：

```
public enum 枚举类型名称 {
    枚举常量1,
    枚举常量2,
    ...
    枚举常量 n
}
```

o public 是可选的访问修饰符，可以省略。

o enum 是定义枚举类型的关键字。

o 枚举类型名称是枚举类型的名称，遵循 Java 标识符的命名规则。

o 枚举常量是枚举类型中定义的常量值，使用逗号分隔。

例如，上一小节定义的代码定义了一个名为 Color 的枚举类型，包含三个枚举常量：RED、GREEN 和 BLUE。实际上，枚举类型的功能远不止于此。枚举类型是一种特殊的类，

更准确地说，它是 java.lang.Enum 类的子类。由于继承了 Enum 类，枚举类型拥有类的一些特性，例如可以定义枚举构造函数和枚举方法。接下来的示例代码中，把前面的枚举类型的 Color 扩展一下，使其拥有枚举常量、枚举构造方法、枚举方法和枚举实例。

```
public enum Color {
    RED(255, 0, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255);

    private final int r;
    private final int g;
    private final int b;

    Color(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public int getR() {
        return r;
    }

    public int getG() {
        return g;
    }

    public int getB() {
        return b;
    }
}
```

这段代码定义了一个名为 Color 的枚举类型，它表示颜色。枚举类型是一种特殊的类，它有固定数量的实例，这些实例都是预先定义的。接下来详细解释一下其中的要点：

枚举常量

在这个 Color 枚举中，定义了三个枚举常量：RED、GREEN 和 BLUE。这些就是 Color 的实例，它们分别代表红色、绿色和蓝色。每个枚举常量后面的括号中的数字是这个颜色

在 RGB 颜色模型中的值。

枚举构造方法

Color 枚举中的构造方法是 Color(int r, int g, int b)。这个构造方法接受三个参数，分别代表红色、绿色和蓝色的值。注意，枚举的构造方法只能是私有的，不能从枚举类型外部调用。

枚举方法

Color 枚举中还定义了三个方法：getR()、getG() 和 getB()。这些方法分别返回枚举实例的红色、绿色和蓝色的值。

枚举实例

枚举实例指的是枚举类型中的一个具体的值。在给出的代码中，RED、GREEN 和 BLUE 就是 Color 枚举的三个实例。

每个枚举实例都是枚举类型的一个对象，它们都有枚举类型中定义的属性和方法。在这个例子中，每个 Color 实例都有 r、g 和 b 这三个属性，以及 getR()、getG() 和 getB() 这三个方法。

枚举实例是在枚举类型定义时就创建的，不能在其他地方创建枚举类型的新实例。这就意味着，一个枚举类型的实例数量是固定的，这是枚举类型的一个重要特性。

19.3 枚举的高级特性

19.3.1 枚举类型不能继承，但是可以实现接口

枚举类型不能被继承。在 Java 中，所有的枚举类型都隐式地继承自 java.lang.Enum 类，因此它们不能再继承其他的类。这是 Java 语言的一个设计决策，目的是保证枚举的安全性和一致性。

虽然枚举类型不能被继承，但是它们可以实现接口。这意味着，如果你想让你的枚举类型有一些共同的行为，你可以定义一个接口，然后让你的枚举类型实现这个接口。比如，我定义了如下的接口：

```
public interface Colorful {
    int getR();
    int getG();
    int getB();
}
```

然后，可以定义一个名为 Color 的枚举类，来实现这个接口。

```
public enum Color implements Colorful{
    RED(255, 0, 0),
    GREEN(0, 255, 0),
    BLUE(0, 0, 255);

    private final int r;
    private final int g;
    private final int b;

    Color(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    @Override
    public int getR() {
        return r;
    }

    @Override
    public int getG() {
        return g;
    }

    @Override
    public int getB() {
        return b;
    }
}
```

在这个例子中，Color 枚举实现了 Colorful 接口，这意味着它必须提供 getR()、getG() 和 getB() 这三个方法的实现。这样，我们就可以使用 Colorful 接口来引用任何实现了这个接口的枚举常量，这提供了更大的灵活性。

19.3.2 枚举无法被实例化

枚举类型的实例化是在枚举类型定义时自动完成的，每个枚举常量就是一个枚举类型的实例。不能使用 new 关键字来创建枚举类型的新实例。

在前面给出的代码中，RED、GREEN 和 BLUE 就是 Color 枚举的三个实例。这些实例在 Color 枚举定义时就被创建，因此不能再创建 Colo 的新实例。

这种设计使得枚举类型非常适合表示一组固定的值，例如颜色、星期、月份等。因为这些值的数量是固定的，所以使用枚举类型可以更好地表示这种情况。

19.3.3 枚举常量实例最好设置为 final

在 Java 中，枚举常量是不可变的，也就是说，一旦它们被创建，它们的值就不能改变。这就是为什么枚举常量是 final 的。

这种设计有几个好处：

一是安全性更高：由于枚举常量的值不能改变，所以你可以安全地将它们公开，而不用担心它们的值会被错误地修改。

二是性能更好：由于枚举常量的值是不变的，所以它们可以被安全地共享和重用，这可以提高性能。三是这样更简单：由于枚举常量的值是不变的，所以它们的行为更容易预测和理解。

在前面给出的代码中，RED、GREEN 和 BLUE 就是 Color 枚举的三个常量。它们的值在创建时就被设置，然后就不能再改变。这就是为什么它们的 r、g 和 b 字段都是 final 的。

19.3.4 不能依赖于枚举常量的序数

在 Java 中，枚举类型的编号默认是从 0 开始的。这个编号被称为枚举常量的序数（ordinal）。可以通过调用枚举常量的 ordinal() 方法来获取它的序数。

例如，对于前面给出的 Color 枚举，可以这样获取每个枚举常量的序数：

```
System.out.println(Color.RED.ordinal()); // 输出 int 类型的 0
System.out.println(Color.GREEN.ordinal());// 输出 int 类型的 1
System.out.println(Color.BLUE.ordinal());// 输出 int 类型的 2
```

需要注意的是，虽然可以使用 ordinal() 方法来获取枚举常量的序数，但是在实际编程中，通常不推荐依赖这个序数，因为它是由枚举常量的声明顺序决定的，如果改变了枚举常量的声明顺序，那么序数也会改变。

在 Java 中，枚举常量的序数是由它在枚举类型中的声明顺序决定的，第一个声明的枚

举常量的序数是 0，第二个声明的枚举常量的序数是 1，以此类推。可以通过调用枚举常量的 ordinal() 方法来获取它的序数，但是不能在运行中改变它的序数。

这种设计使得枚举类型的行为更加可预测和一致，因为可以确定每个枚举常量的序数总是固定的。

19.3.5 values() 和 valueOf() 方法

在 Java 的枚举类型中，values() 和 valueOf() 是两个内置的方法。

values() 方法：这个方法返回一个包含枚举类型中所有枚举常量的数组，这些枚举常量的顺序与它们在枚举类型中声明的顺序相同。这个方法通常用于遍历枚举类型中的所有枚举常量。

valueOf() 方法：这个方法接受一个字符串参数，返回与这个字符串相对应的枚举常量。如果枚举类型中没有与这个字符串相对应的枚举常量，这个方法会抛出 IllegalArgumentException。

接下来，展示一下这两个方法的用法：

```
// 这段代码会打印出 Color 枚举中的所有枚举常量。
for (Color color : Color.values()) {
    System.out.println(color);
}
// 这段代码会返回 Color 枚举中的 RED 枚举常量
Color color = Color.valueOf("RED");
System.out.println(color);
```

需要注意的是，valueOf() 方法对大小写敏感，所以传入的字符串必须与枚举常量的名称完全匹配，包括大小写。

19.3.6 EnumSet 和 EnumMap

EnumSet 和 EnumMap 是 Java 中的两种特殊的集合类型，它们专门用于处理枚举类型。

19.3.6.1 EnumSet

这是一个专门用于枚举类型的 Set 实现。它内部使用位向量来表示枚举常量集合，因此它非常紧凑和高效。EnumSet 支持所有的集合操作，并且这些操作都是以常数时间运行的，这使得它成为实现基于枚举类型的集合的最佳选择。这意味着它不包含重复项，并且支持集合的基本操作，如添加、删除、查询等。接下来，展示 EnumSet 的各种用法。

```
import java.util.EnumSet;
public class EnumSetExample {

    public static void main(String[] args) {
        // 创建一个包含所有枚举值的 EnumSet
        EnumSet<DayOfWeek> allDays =
                            EnumSet.allOf(DayOfWeek.class);
        System.out.println(" 一周包括: " + allDays);

        // 创建一个仅包含特定枚举值的 EnumSet
        EnumSet<DayOfWeek> workingDays =
                    EnumSet.of(DayOfWeek.MONDAY,
                    DayOfWeek.TUESDAY, DayOfWeek.WEDNESDAY,
                    DayOfWeek.THURSDAY, DayOfWeek.FRIDAY);
        System.out.println(" 工作日: " + workingDays);

        // 判断某个枚举值是否存在于集合中
        boolean hasSaturday =
                  workingDays.contains(DayOfWeek.SATURDAY);
        System.out.println(" 周六是工作日么?  " + hasSaturday);

        // 添加或移除枚举值
        workingDays.add(DayOfWeek.SATURDAY);
        System.out.println(" 把周六添加到工作日中: "
                                        + workingDays);

        workingDays.remove(DayOfWeek.SUNDAY);
        System.out.println(" 把周日移出工作日: "
                                        + workingDays);

        // 使用范围创建 EnumSet
        EnumSet<DayOfWeek> weekend =
                  EnumSet.range(DayOfWeek.SATURDAY,
                  DayOfWeek.SUNDAY);
        System.out.println(" 周末: " + weekend);
    }
```

```
}
```

为了节省篇幅，上面这段代码省略了 DayOfWeek 的代码，该代码可以在本章开头找到。这段代码是一个关于 Java 中 EnumSet 使用的示例。

首先，定义了一个名为 allDays 的 EnumSet，它包含了 DayOfWeek 枚举中的所有枚举常量，这是通过调用 EnumSet 的 allOf() 方法实现的。

然后，定义了一个名为 workingDays 的 EnumSet，它只包含了 DayOfWeek 枚举中的一部分枚举常量，即周一到周五。这是通过调用 EnumSet 的 of() 方法实现的。

接着，使用 contains() 方法检查 workingDays 是否包含 DayOfWeek.SATURDAY，并将结果打印出来。

然后，使用 add() 方法将 DayOfWeek.SATURDAY 添加到 workingDays 中，使用 remove() 方法将 DayOfWeek.SUNDAY 从 workingDays 中移除。

最后，使用 EnumSet 的 range() 方法创建了一个名为 weekend 的 EnumSet，它包含了 DayOfWeek 枚举中的一段范围的枚举常量，即周六和周日。

这段代码展示了 EnumSet 的主要用法，包括创建 EnumSet、添加和移除枚举常量、检查枚举常量是否存在于 EnumSet 中，以及使用范围创建 EnumSet。代码运行之后的结果如下：

```
一周包括: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                            FRIDAY, SATURDAY, SUNDAY]
工作日: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
周六是工作日么?  false
把周六添加到工作日中: [MONDAY, TUESDAY, WEDNESDAY,
                              THURSDAY, FRIDAY, SATURDAY]
把周日移出工作日: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                                        FRIDAY, SATURDAY]
周末: [SATURDAY, SUNDAY]
```

19.3.6.2 EnumMap

EnumMap 是 Java 集合框架中的一种特殊的 Map 实现，它的键（Key）必须来自单个枚举类型。由于枚举类型的值数量有限且固定，EnumMap 内部可以使用数组来存储数据，这使得 EnumMap 的存储和访问效率非常高。接下来，演示 EnumMap 的用法：

```
import java.util.EnumMap;
import java.util.Map;

public class EnumMapExample {
    public static void main(String[] args) {
        EnumMap<DayOfWeek, String> dayMap =
                    new EnumMap<>(DayOfWeek.class);
        dayMap.put(DayOfWeek.MONDAY, " 星期一");
        dayMap.put(DayOfWeek.TUESDAY, " 星期二");
        dayMap.put(DayOfWeek.WEDNESDAY, " 星期三");
        dayMap.put(DayOfWeek.THURSDAY, " 星期四");
        dayMap.put(DayOfWeek.FRIDAY, " 星期五");
        dayMap.put(DayOfWeek.SATURDAY, " 星期六");
        dayMap.put(DayOfWeek.SUNDAY, " 星期日");

        for (Map.Entry<DayOfWeek, String> entry :
                                dayMap.entrySet()) {
            System.out.println(entry.getKey() + ": "
                                + entry.getValue());
        }
    }
}
```

这段代码是一个关于 Java 中 EnumMap 使用的示例。EnumMap 是一个专门用于枚举类型的 Map 实现。

首先，创建了一个 EnumMap 实例 dayMap，它的键是 DayOfWeek 枚举的值，值是对应的中文名称。

然后，使用 put() 方法向 dayMap 中添加了七个键值对，每个键值对都由一个 DayOfWeek 枚举常量和一个字符串组成。

接着，使用 entrySet() 方法获取 dayMap 中的所有键值对，然后遍历这个键值对集合。在每次迭代中，都会打印出一个键值对，键和值之间用冒号和一个空格分隔。

这段代码展示了 EnumMap 的主要用法，包括创建 EnumMap、添加键值对，以及遍历键值对。上面的代码运行之后的结果如下：

```
MONDAY: 星期一
TUESDAY: 星期二
WEDNESDAY: 星期三
THURSDAY: 星期四
FRIDAY: 星期五
SATURDAY: 星期六
SUNDAY: 星期日
```

20 注解

Contents

在继承的例子中，我们使用了 @Override 这个注解（Annotation），这一章，就以它为契机，开启对注解的学习。注解是 Java 语言中的元数据，它可以为代码添加额外信息，用于描述、解释或修改代码的行为。注解本身并不影响代码的执行，但它可以被编译器、IDE 和其他工具读取和使用，从而提供更好的代码理解、分析和维护能力。

接下来，我们将深入探讨什么是注解，它的基本语法和使用方式，Java 中常见的预定义注解，以及如何创建和使用自定义注解。

20.1 什么是注解

Java 注解，也称为元数据（metadata），是一种代码级别的说明。它是 Java 5 引入的一个特性，与类、接口、枚举处于同一个层次。

注解的语法格式如下：

```
@<注解类型> ［参数列表］ 被注解的内容
```

其中：@ 符号表示这是一个注解，是注解的名称，[参数列表] 是可选的，用于为注解提供参数，被注解的内容可以是类、方法、字段、局部变量或方法参数。

Java 注解可以声明在以下元素前面：包、类、接口、枚举、方法、字段、局部变量、方法参数。下面的代码使用了四种注解，分别用于描述包、类、字段和方法。

```
// 包注解在这个例子中，我们使用 @Package 注解描述了包的名称
@Package(name = "com.liuyandong")
package com.liuyandong;

// 类注解
@Entity
public class Person {
  // 字段注解
  @Id
  private Long id;

  // 方法注解
  @Override
  public String toString() {
    return "Person{" +
      "id=" + id +
      '}';
  }
}
```

注解和被注解的内容可以写在一行中，也可以写在两行中。虽然注解的书写方式没有严格的规定，可以根据实际情况灵活选择。但是我建议为了可读性和一致性，写成两行。

Java 注解的作用主要有两个：一是为代码添加元数据，用于描述代码的额外信息，例如作者、版本、功能等。二是允许通过反射机制访问和处理元数据，从而实现代码的动态控制和扩展。

注解一般有如下的使用场景：一种是直接在代码中声明 Java 内置的注解。第二种是使用自定义注解扩展 Java 语言的功能。第三种是通过反射机制获取和处理注解信息。在本章中，我着重介绍前两种，第三种则放在反射机制那一章讲。

20.2 在代码中使用 Java 预定义的注解

20.2.1 注解与注释的区别

在 Java 中，注释和注解是两个密切相关但又有所区别的概念。

注释（Comment）是指代码中以 // 或 /* … */ 开头的文本，用于解释或说明代码。注释不会被编译器编译，仅供程序员阅读参考。

```
// 这是一条注释
public void myMethod() {
    // ...
}
```

// 这是一条注释是注释，用于解释 myMethod() 方法的功能。

注解（Annotation）是 Java 5.0 引入的一个特性，是一种代码级别的说明，可以为代码添加元数据（metadata）。注解会被编译器处理，并可以被反射机制访问和处理。

```
@Deprecated
public void oldMethod() {
    // ...
}
```

@Deprecated 是注解，用于表明 oldMethod() 方法已过时，不建议使用。

简单来说，注释是给程序员看的，而注解是给编译器和 JVM 看的。在实际开发中，注释和注解经常会一起使用。注释可以用来解释代码，而注解可以用来提供额外的元数据信息，提高代码的可读性和可维护性。

20.2.2 使用 Java 预定义的注解

在上一章，我们已经使用过 @Override 这个注解，该注解用于标识方法覆盖了父类中的同名方法。除此之外，Java 还内置了一些常用注解，我列举如下两个常用的注解：

@Deprecated：用于标识方法已过时，不建议使用。

@SuppressWarnings：用于抑制编译器警告。

@Deprecated 注解是 Java 中的一个标准注解，用于标记已过时的类、方法、属性等。该注解的作用如下：告知开发人员该类、方法、属性等已过时，不建议使用。在编译时，编译器会对使用已过时的类、方法、属性等发出警告。

在实际的编程中，可能会对现有的类、方法、属性等进行修改。在修改之前，可以使用 @Deprecated 注解来标记旧的类、方法、属性等已过时。以下的代码是使用 @Deprecated 注解的例子：

```java
public class DeprecatedDemo {
    /**
     * 这是一个已经过时的方法，将来可能会被移除，
     * 请改用新的替代方法 {@link #newMethod()}
     * @deprecated 从版本 2.0 开始不再推荐使用此方法
     */
    @Deprecated(since = "2.0", forRemoval = true)
    public void oldMethod() {
        System.out.println(" 这是旧方法，已被弃用！");
    }

    public void newMethod() {
        System.out.println(" 这是推荐的新方法！");
    }

    public static void main(String[] args) {
        DeprecatedDemo example = new DeprecatedDemo();
        // 调用旧方法时，IDE 会显示已过时的警告，编译器也会发出警告
        example.oldMethod();
        example.newMethod();
    }
}
```

在这段代码中，oldMethod() 方法被标记为 @Deprecated。这意味着，当其他代码尝试调用这个方法时，IDE 和编译器会发出警告，提示这个方法已经过时。

@Deprecated 注解有两个属性可以设置：

since：一个字符串，表示从哪个版本开始，这个程序元素被标记为过时。在这个例子中，oldMethod() 从版本”2.0” 开始就被标记为过时。

forRemoval：一个布尔值，如果为 true，表示这个程序元素在未来的版本中可能会被移除。在这个例子中，oldMethod() 被标记为可能会被移除。

这段代码中的 newMethod() 就是一个替代的方法，建议使用这个新方法来替代已经过时的 oldMethod()。

另一个在编程中经常使用的是 @SuppressWarnings 这个注解，用于抑制编译器警告。有时候，编译器会乱发警告，类似于微软的 word 一样，总是提醒你这里有问题，那里有问题。如果你不想看到编译器的警告，可以使用这个注解抑制它们。

```
public class SimpleSuppressionExample {

    public static void main(String[] args) {
        // 没有使用变量 i，通常编译器会给出"unused variable" 警告
        int i = 10;

        // 使用 @SupressWarnings 注解来抑制这个警告
        @SuppressWarnings("unused")
        int unusedVariable = 5;

        // 显然，这个注解也可以放在方法上，以抑制整个方法内的所有相关警告
        // 下面的方法调用了 Math.abs(Integer.MIN_VALUE)，
        // 这会导致"integer overflow" 警告，但我们选择忽略它
        @SuppressWarnings("all")
        int result = calculateSomething();

        System.out.println("This is a simple example.");
    }

    @SuppressWarnings({ "unused", "constant-conditions" })
    private static int calculateSomething() {
        int value = Math.abs(Integer.MIN_VALUE);
        return value;
    }
}
```

上面的代码中，首先声明了一个未使用的变量 i，这会触发编译器警告。然后，main 方法声明了一个使用 @SuppressWarnings 注解抑制警告的变量 unusedVariable。

最后，main 方法调用了 calculateSomething 方法，该方法会触发”integer overflow” 警告，但该警告也被 @SuppressWarnings 注解抑制了。

还有一点需要注意，这里是抑制编译器的警告，并不是抑制编辑器或者 IDE 的警告。由于各家推出的 IDE 不同，比如 Eclipse 和 IntelliJ IDEA，可能会在静态分析时产生不同的警告，并非编译错误，因此无法通过 @SuppressWarnings 注解抑制。

20.3 在代码中使用自定义的注解

在 Java 中使用自定义注解需要三个步骤：定义注解、使用注解和解释注解。

20.3.1 定义注解

使用 @interface 关键字定义注解类型。例如，定义一个名为 MyAnnotation 的注解：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String value() default "";
}
```

我来详细解释一下这段代码中的意思：

`@Retention(RetentionPolicy.RUNTIME)`：这是一个元注解，用于指定 MyAnnotation 注解自身的保留策略。RetentionPolicy.RUNTIME 表示此注解不仅在编译时期有效，而且在运行时也可以通过反射 API 获取到。这意味着当你的程序运行时，可以动态地检测到哪个方法使用了 MyAnnotation 注解以及其参数值。

`@Target(ElementType.METHOD)`：这也是个元注解，它指定了 MyAnnotation 可以应用到的目标元素类型。ElementType.METHOD 表示这个注解只能用于方法级别，即你只能将 MyAnnotation 注解放在类中的某个方法声明前。

`public \@interface MyAnnotation { ... }`：定义了一个新的注解类型 MyAnnotation。在 Java 中，注解是一种用于向编译器或 JVM 提供附加信息的元数据。

`String value() default "";`：在 MyAnnotation 注解内部定义了一个成员变量 value，类型为字符串（String），并且设置了一个默认值为空的字符串 ““。

当你在使用这个注解时，可以选择性地给 value 赋予一个值，例如 @MyAnnotation(“Hello”) 中的 “Hello” 就是赋给 value 的值。如果未明确赋予值，则使用默认值。

20.3.2 使用注解

将自定义注解应用于代码元素，例如方法：

```
public class MyClass {
    @MyAnnotation("Hello")
    public void myMethod() {
        // ...
```

```
    }
}
```

这段 Java 代码定义了一个名为 MyClass 的公共类，并在其中包含了一个被 @MyAnnotation(“Hello”) 注解修饰的公共方法 myMethod()。

@MyAnnotation(“Hello”)：这是一个自定义注解的应用实例，该注解的名称为 MyAnnotation，并且在此处传递了一个参数 “Hello” 给注解的 value 属性。由于我们之前定义的 MyAnnotation 注解包含一个 value 属性且有默认值，这里显式地设定了它的值为 “Hello”。

public void myMethod() {}：这是类中的一个公共方法，没有返回值，方法体中目前没有实现任何功能，用 // … 表示待补充的代码。

这段代码的意义在于，通过 @MyAnnotation(“Hello”) 标记了 myMethod() 方法，为了在运行时通过反射等手段读取和利用这个注解信息来进行某些特殊处理，如日志记录、AOP（面向切面编程）、框架功能增强等。具体的用途取决于你在其它地方如何解析和使用这个自定义注解。

20.3.3 解析注解

使用反射 API 获取注解信息并进行处理。例如，获取 myMethod 方法上的 MyAnnotation 注解：

```
MyClass myClass = new MyClass();
Method method = myClass.getClass().getMethod("myMethod");
MyAnnotation annotation =
                method.getAnnotation(MyAnnotation.class);
String value = annotation.value(); // 获取 value 元素的值
```

上面这段代码用了反射，而反射是下一章的内容。注解和反射往往一起配合使用，尤其是在一些高级框架中，如 Spring、Hibernate、MyBatis 等。注解可以作为一种元数据标记，然后通过反射机制在运行时扫描和解析这些注解，依据注解信息来动态地改变程序行为。

例如，在 Spring 框架中，通过注解如 @Component、@Service 等标记类作为 Bean，然后通过反射动态地初始化和管理这些 Bean。总结来说，注解主要用于提供额外的元数据信息，而反射则用于在运行时分析类结构和操作对象，两者相结合可实现强大的动态编程能力。

解析注解后，可以根据注解信息执行相应的操作。例如，根据 MyAnnotation 的 value 值进行不同的处理。

注意：自定义注解本身不会做任何事情，需要编写代码解析和处理注解信息才能发挥作用。注解可以与反射、AOP 等技术结合使用，实现更强大的功能。

21 反射

Contents

Java 反射机制虽然初看难懂深奥，但只要耐心学习并彻底理解，它实际上并非想象中那般艰涩。反射作为 Java 语言的重要特性，为我们提供了在程序运行期间检查和调整应用程序本身行为的强大能力。

反射允许我们在运行时检查类、接口、字段以及方法的元数据信息，并且还能动态地创建、检查和调用对象的属性和方法，无需提前了解它们的名称。这种动态获取程序元素信息和动态执行逻辑的能力，大大提高了 Java 程序的灵活性和可扩展性。

虽然反射的概念最初可能会显得有点抽象，但通过一步步掌握反射相关 API 的使用，你会发现它并非像想象中那么困难。一旦深入了解并熟练运用反射，就能体会到它在框架开发、单元测试、动态代理等多个领域的巨大应用价值。

反射虽不是日常开发中必须的技能，但作为一个专业的 Java 开发人员，了解和掌握反

射知识无疑是很有必要的。只要持之以恒学习和实践，反射定能为你的开发之路增添新的动力和乐趣。

Java 反射的应用非常广，我先举一个例子，加深对反射的印象。日常开发中，程序员们离不开 IDE 的自动补全功能，而这一智能化功能正与 Java 反射机制密切相关。自动补全的核心在于 IDE 对代码的语义理解和分析，而反射则为 IDE 提供了在运行时动态获取类结构信息 (如方法、属性等) 的大门。具体来说，IDE 是通过如下几步来实现自动补全的：

首先，当项目一旦打开，IDE 会将相关的 Java 源文件完整加载到内存中。随后，IDE 会基于这些代码生成抽象语法树 (Abstract Syntax Tree，AST)，以建模代码的逻辑结构。这个阶段，有时候会让我们觉得 IDE 有些卡顿，其原因是 IDE 创建索引，这需要消耗一定的 CPU 跟 IO 资源。

接下来，根据自动补全的上下文位置，IDE 会利用反射中的 Class.forName() 等方法，动态加载并解析相关类的元数据信息。通过反射获取到类的构造器、方法、属性等信息后，IDE 会与之前生成的 AST 结合分析，精准找出可以补全显示的代码元素。

最终，IDE 会以列表的形式，将这些可以补全的类名、方法名、变量名等悉数显示给开发者，供其选择和插入。

反射为 IDE 赋予了运行时动态解析代码的能力，使其不仅能分析当前的代码文本，更像解释器一样动态加载相关类的运行时信息，从而对代码有了更加准确的“理解”能力。这种智能化的代码语义分析，对自动补全提示等智能编程辅助功能的实现是至关重要的。希望通过这个例子，使大家直觉的感受到反射的作用。

接下来，我们将从易到难的顺序，逐步分析 Java 反射的用法与原理。

21.1 反射的历史与发展

Java 的反射现在用途如此之广，其实并不是 Java 反射的本意，更多的是无心插柳的结果，下面，我来讲一下反射背后的故事。

21.1.1 起初，反射仅仅是为了跨平台

在 Java 之前的一些面向对象语言中，由于语言本身与特定操作系统和硬件平台强绑定，程序在不同平台间移植时需要重写大量与平台相关的底层代码，造成了巨大的重复工作。举个例子，C++ 作为主要的面向对象语言，其编译器和运行库是针对特定操作系统和 CPU 架构编写的。例如 Windows 下的 C++ 程序使用了 Windows API，在 Linux 下就无法直接运行，因为 Linux 没有 Windows API 的实现。同样，在 x86 架构的 PC 上编译的 C++ 程序无法直接在 ARM 架构的手机上运行。要移植，就需要针对新平台重新编译或重写部分平台相关代码。

而 Java 被设计为一种可移植的语言，这意味着编写一次 Java 代码后，通过不同平台上的 Java 虚拟机 (JVM)，就可以在各种环境中运行，不需要为每个平台单独编写程序版本。但是，跨平台可移植带来的一个挑战是：如何针对不同的底层对象类型采取不同的处理逻辑？因为在不同平台上，同一种类型的对象在内存中的表示可能有所不同。

这就是反射在 Java 早期发挥重要作用的关键。通过反射，Java 程序可以在运行时动态获取一个对象的类型信息，检查它的内部结构，从而根据对象的实际类型采取相应的处理方式。来看一个具体的例子，说明如何利用反射检查对象类型并以统一方式处理 File 对象，从而实现可移植性。

众所周知，在 Windows 和 Linux 这两个不同的操作系统平台上，文件（File）对象的底层实现有所不同, 主要体现在以下几个方面:

在不同操作系统平台上，File 对象的底层实现可能有所不同，主要体现在以下几个方面：

文件路径分隔符不同：在 Windows 上使用\作为路径分隔符，而在 Linux/Unix 上使用/。

文件名大小写敏感：Windows 文件系统对文件名不区分大小写，而 Linux/Unix 是区分大小写的。

肯定还有我没想到的差异，这些差异肯定会影响到处理 File 对象的代码，如果硬编码针对某个特定平台，就失去了可移植性。所以 Java 提供了一个统一的对象来处理文件，具体来说，是能来 java.io.File 这个类来实现统一。换句话说，Java 语言本身已经为我们屏蔽了不同操作系统底层实现的差异，通过 JVM 为我们提供了一个统一的对象模型。我们在编码时只需要操作这些统一的 Java 对象，而不用关心具体平台的细节，从而实现了可移植性。

接下来的问题就变的有趣了，假设代码已经运行起来了，有一个方法 processFile(Object obj) 用来处理文件对象，如果传给这个方法的对象是合法的，能运行。如果传给这个方法的对象是不合法的，不就死机了么？如果在执行 processFile(Object obj) 的时候，我能对传给方法的参数 obj 问一句：“嘿，哥们，你是什么类啊？”然后 obj 这个对象回答，我再决定如何执行就可以了。

在 C++ 中，很难做到这一点。C++ 语言本身没有提供类似 Java 反射那样的内省机制，无法在运行时动态获取对象的类型信息。Java 的反射可以，它为我们提供了在运行时自动获取对象类型和结构信息的能力，这种语言层面的支持极大地简化了编写通用、可移植代码的难度。对 processFile(Object obj) 写一段伪代码如下（暂时看不懂没关系，学完本章以后，肯定能看懂）：

```
public void processFile(Object obj) throws Exception {
    // 获取对象类型
```

```
        Class<?> clazz = obj.getClass();

        // 检查是否为 File 类型或其子类
        if (java.io.File.class.isAssignableFrom(clazz)) {
            java.io.File file = (java.io.File)obj;

            // 统一处理文件
            System.out.println("Processing file: "
                                + file.getPath());
            // ... 其他文件操作
        } else {
            throw new IllegalArgumentException(
                                "Not a File object");
        }
    }
```

在上面这段代码中，使用了反射的主要有两处：

第一处是 Class<?> clazz = obj.getClass(); 这一行代码使用了反射机制中的 getClass() 方法，获取了 obj 对象所属的 Class 对象。Class 对象为我们提供了访问该类型信息的入口。这是反射的基础操作。

第二处是 java.io.File.class.isAssignableFrom(clazz)，这里使用了 isAssignableFrom() 方法，它是 Class 类中的一个反射方法，用于判断一个类型是否为另一个类型的子类或子接口。它内部实现需要检查两个类型之间的继承关系。

总结一下，这段代码利用了反射的 getClass() 和 isAssignableFrom() 两个核心方法，充分展现了反射在运行时获取和判断类型信息方面的能力，为实现可移植性奠定了基础。反射为 Java 程序带来了非常强大的动态编程和自省能力。

21.1.2 后来，反射的发展就出乎意料了

Java 语言引入反射机制的初始目的确实主要出于对可移植性的考量，但随着时间的推移，程序员们发现了反射更多的潜在用途，从而使得反射的应用场景日益扩大。

最初，Java 的设计者们意识到，为了实现“一次编写，到处运行”的可移植性目标，Java 必须有一种机制来动态获取运行时类型信息，并根据对象实际类型采取相应的处理逻辑。这正是反射被引入的根本原因。

但是，设计良好的语言特性往往会超出最初设计者的预期，发展出更广阔的应用前景。反射也不例外，随着 Java 生态的不断发展，程序员们逐步发现，反射除了保证可移植性之

外，还可以用于实现诸多强大的功能:

o 实现通用编程框架，如 IDE 自动补全、ORM 框架、依赖注入容器

o 构建动态代理，在运行时动态创建代理对象

o 支持动态执行字节码，实现动态脚本引擎

o 实现各种编程范式，如面向方面编程 (AOP)

o 构建通用日志、测试、监控等工具

o 支持 JavaBean 属性的自省和自动化处理

所以，可以说反射从最初仅仅是为了支持可移植性的底层机制，逐步发展成为 Java 语言不可或缺的动态编程利器，成为实现许多高级特性和框架的核心基础。

21.2 获取 Class 对象

在 Java 中，Class 对象用于表示类的元数据。每个类在内存中都有一个对应的 Class 对象，该对象包含了类的结构信息，例如类的名称、字段、方法、构造函数等。通过 Class 对象，我们可以在运行时动态获取和操作类的结构信息，进行反射操作。

Class 对象是 Java 反射机制的基础，可用于在运行时检查和操作类的属性和行为。只有获取到 Class 对象，才能进一步查看类的构造器、方法、字段等信息，并对它们进行操作。因此，Class 对象可视为反射 API 的入口和核心。

通过 Class 对象，我们可以动态创建实例、调用方法、访问字段等，实现了 Java 语言在运行时对代码的自省 (introspection) 和操作能力。Class 的引入奠定了反射在 Java 中广泛应用的基础，同时，要进行反射操作，必须先获取 Class 对象。

21.2.1 构建一个示例代码

在详细讲如何获取 Class 类之前，我们需要构建一个可以运行的例子，以方便我们对其进行操作。这个例子还是前面讲的球员类、门将类并且两者都实现了一个名为 CActor 的接口，实现该接口的球员可以拍摄广告。为演示如何使用反射，先构建一个例子来演示。

先是 CActor 接口的代码如下:

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

public interface CActor {

    void shootCommercial();
```

```
}
```

接下来是球员类的代码:

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

import com.l.r.CActor;

public class Player implements CActor{
    // 球员姓名，具有 getter 和 setter 方法
    private String name;
    static int playerCount = 0;

    // 球员年龄，只有 getter 方法
    private int age;

    // 构造函数
    public Player(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // name 的 getter 和 setter 方法
    public String getName() {
        return name;
    }

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

    // age 的 getter 方法
    public int getAge() {
        return age;
    }
```

```
    // 一个公开（public）的方法
    public void publicMethod() {
        System.out.println(" 这是一个 public 方法");
    }

    // 一个私有（private）的方法
    private void privateMethod() {
        System.out.println(" 这是一个 private 方法");
    }

    // 一个静态（static）方法
    public static void publicStaticMethod() {
        System.out.println("
                这是一个 public static void 的方法");
    }

    @Override
    public void shootCommercial() {
        System.out.println(" 这是实现接口中的方法");
    }
}
```

然后是门将类的代码：

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

public class Goalkeeper extends Player {
    private final int reactionSpeed;
    public Goalkeeper(String name, int age,
                      int reactionSpeed) {
        super(name, age);
        this.reactionSpeed = reactionSpeed;
    }
}
```

以上就是一个接口和两个类，非常的简单，下面是这三个元素之间的关系图：

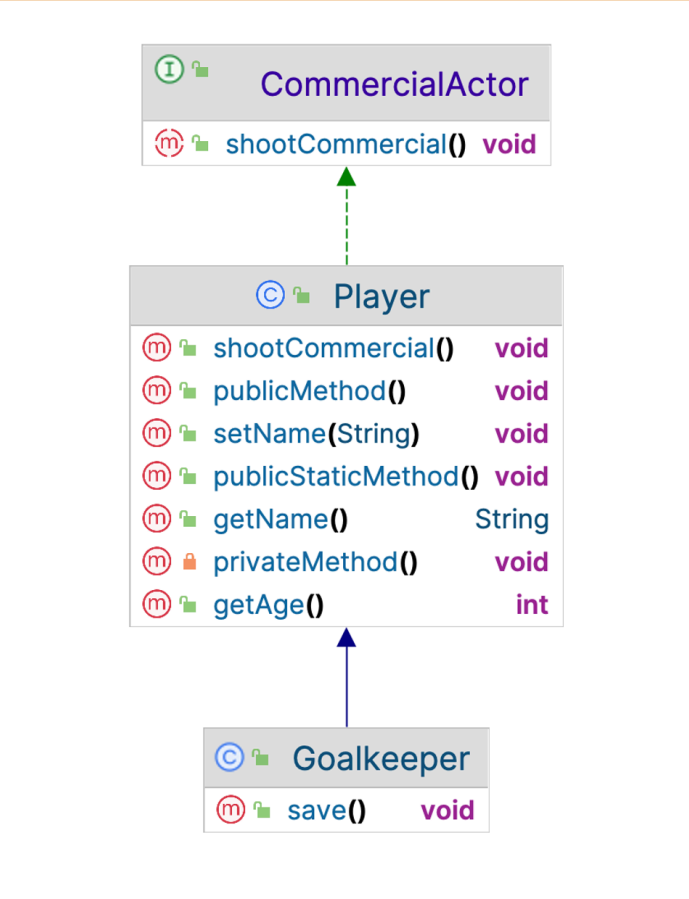

图 21.1: 接口与类之间的 UML 图

有了这些类，接下来, 我们将使用反射来展现这些元素之间的关系。

21.2.2 获取 Class 对象的三种方式

在 Java 的反射中，通常有三种方式来获取 Class 类，这三种方式分别是：

第一，通过对象实例的 getClass() 方法：如果对象实例存在，可以调用实例的 getClass() 方法来获取对象的运行时类。

第二种，通过类的 class 属性：当类的实例不存在时，可以使用类字面常量 class 来获取类的引用。

第三种，通过 Class.forName() 方法：可以通过类的全限定名（字符串形式）来获取 Class 对象。

接下来，展示如何用以上三种方法获取 Class 对象。

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

public class GetClassObject {
    public static void main(String[] args) {

        // 方法 1: 使用对象的 getClass 来获取 Class 类
        Player messi = new Player("Messi", 36);
        CommercialActor ronaldo =
                        new Player("Ronaldo", 38);
        Goalkeeper goalkeeper =
                    new Goalkeeper("Neuer", 36, 95);

        Class<?> playerClass1 = messi.getClass();
        Class<?> goalkeeperClass1 = goalkeeper.getClass();
        Class<?> commercialActorClass1 = ronaldo.getClass();

        System.out.println("messi 的 Class 对象是: "
                                    + playerClass1);
        System.out.println("goalkeeper 的 Class 对象是: "
                                + goalkeeperClass1);
        System.out.println("ronaldo 的 Class 对象是: "
                            + commercialActorClass1);

        // 方法 2: 使用 Class 的属性.class 来获取 Class 类
        Class<?> playerClass2 = Player.class;
        Class<?> goalkeeperClass2 = Goalkeeper.class;
        Class<?> commercialActorClass2 =
                                CommercialActor.class;

        System.out.println("Player 的 Class 对象是: "
                                        + playerClass2);
        System.out.println("Goalkeeper 的 Class 对象是: "
                                    + goalkeeperClass2);
        System.out.println("CommercialActor 的 Class 对象是: "
                                + commercialActorClass2);
```

```
        // 方法 3: 使用 Class.forName 来获取 Class 类
        try {
            Class<?> playerClass3 = Class.forName(
                    "com.liuyandong.reflectrion.Player");
            Class<?> goalkeeperClass3 =Class.forName(
                "com.liuyandong.reflectrion.Goalkeeper");
            Class<?> commercialActorClass3 = Class.forName("
              com.liuyandong.reflectrion.CommercialActor");

            System.out.println("Player 的 Class 对象是: "
                                                + playerClass3);
            System.out.println("Goalkeeper 的 Class 对象是: "
                                              + goalkeeperClass3);
            System.out.println("CommercialActor 的 Class
                            对象是: " + commercialActorClass3);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

    }
}
```

以上三种获取 Class 对象的结果都是相同的，值得一提的是，接口（Interface）在 Java 中也有对应的 Class 对象。每个接口在内存中也有一个对应的 Class 对象，用来表示接口的结构信息。通过 Class 对象，我们可以获取接口的名称、方法等信息，进行反射操作。接口的 Class 对象在 Java 反射中也扮演着重要的角色，可以用来检查和操作接口的属性和行为。

那这三种获取 Class 对象有什么区别么？还是有区别的，接下来，我来详细解释一下。

21.2.2.1 使用 getClass() 这种方式获取 Class 对象

这种方式的前提是已经有一个对象实例。每个对象实例在运行期都能够通过内部的方法获取到自身对应的 Class 对象。我们可以通过 “对象实例.getClass()” 语法来获取该对象对应类的 Class 对象。比如上面代码中的这一句：

```
Class<?> playerClass1 = messi.getClass();
```

你可能会好奇，这个方法是谁实现的呢？getClass() 方法在 Object 类中是一个 final 方法，意味着它不能被子类覆盖或重写。因此，无论是哪个类，包括用户自定义的类，都无

法覆盖 getClass() 方法。这是因为在 Java 中，Object 类中的 getClass() 方法已经提供了标准的实现方式，为了保证对象的类信息的一致性和准确性，不允许子类修改这个方法的行为。

而且 getClass() 方法在 Object 类中是一个 final 方法，意味着它不能被子类覆盖或重写。因此，无论是哪个类，包括用户自定义的类，都无法覆盖 getClass() 方法。这是因为在 Java 中，Object 类中的 getClass() 方法已经提供了标准的实现方式，为了保证对象的类信息的一致性和准确性，不允许子类修改这个方法的行为。所以，只要有对象实例，大家可以放心的使用 getClass() 方法。

21.2.2.2 使用类的 class 属性来获取 Class 对象

“类名.class” 这种获取 Class 对象的方式，适用于编译期间就已经能确定具体类型的情况，是最简单高效的方法。为什么说这种方式最高效呢？因为类名.class 是一个编译时常量，在编译时就已经确定了具体的类类型，所以它是一种简单高效的获取 Class 对象的方法。这种方式不需要创建类的实例，而且在编译时就已经确定了类型，所以具有较高的效率。

再深入一下，.class 属性是 Java 语言中的一种语法糖（syntactic sugar），它是一种方便的编码形式，由编译器在编译阶段进行转换处理。

具体来说，当我们编写上面例子中类似 Player.class 这样的代码时，编译器会在内部将其转换为调用 Class.forName(“com.l.r.Player”) 或者使用其他等价的字节码指令，从而获取对应的 Class 对象。这种转换过程对于开发者是透明的，我们可以直接使用.class 语法糖，而无需关心编译器内部的实现细节。这两种方法是获取 Class 对象最常见的情况，如果这两种情况都无法实现，则使用下面第三种方法。

21.2.2.3 通过 Class.forName() 方法来获取 Class 类

可能你会有疑惑，在上一条讲使用 “类名.class” 属性获取 Class 类的时候，我讲过，其内部可能是使用 Class.forName(“com.l.r.Player”) 来获取的 Class 对象，为何还需要显示调用呢？这是个好问题。

原因是，有时候，在编译期无法确定所需类的具体类型，需要根据运行时的条件动态加载相应的类。不直接使用 “类名.class” 的原因在于：“类名.class” 语法只能用于编译器在编译期就已经确定的类型。如果类型是未知的或者需要动态确定的，就无法使用这种方式。

一般来说，使用动态加载类这种方式的场景都比较高阶了，我举两个例子：

o 远程加载类：有些系统会通过网络从远程服务器动态加载类，这时需要使用类的全限定名从网络上获取类的字节码。比如 Tomcat 服务器就能在运行期动态加载更新后的类，而无需重启整个服务器。这种热部署能力对于需要经常更新发布的 Web 应用程序来说是非常有帮助的。

o 插件开发：使用动态加载实现插件热插拔功能，比如 IntelliJ IDEA 的插件开发，就用到了 OSGi 框架，这里就使用了动态加载类的技术。

在这些场景下，由于编译期无法获知具体的类名或类型信息，所以无法像 Player.class 这样直接使用类型标识符，必须使用 Class.forName() 这种动态加载类的方式。当然，这种方法也有一些缺点，比如可访问性检查会增加一些开销，因为加载的类有可能不存在，如果大家返回去看前面的代码，会发现我对这种情况做了一些简单的错误处理。

总之，第三种方法使用起来相比于前两种更加复杂，所以如果编译期就能确定类型，优先选择使用 “类名.class” 的方式获取 Class 对象。

这种动态加载类的情况，不止 Class.forName() 这一种，只是这种用的比较多。还有几种用的更少的，比如自定义 ClassLoader 的方法、反序列化方法、通过 Java 代理工具对 Class 进行操作加载以及使用 URLClassLoader 从指定的 URL 路径加载动态类。本书不再赘述。

在获取到目标的 Class 类之后，就可以利用反射的核心 API 对这个类进行动态操作了。接下来，开始介绍基础的反射 API。

21.3 基础的反射 API

反射让我们可以在运行期间检查或修改类、接口、字段、方法、构造方法等的元数据，并动态创建对象实例、操作字段值、调用任意方法等。通过反射机制，我们可以突破面向对象编程的封装性限制，使 Java 程序具有了极大的灵活性和可扩展性。但同时，反射操作也会带来一些性能和安全方面的开销，因此应该谨慎使用。

反射操作的核心在于使用 java.lang.reflect 包中提供的 API。这个包中包含了 Constructor、Field、Method 等几个核心类，分别对应着类的构造方法、字段和方法。通过这些类，我们可以执行各种反射操作。根据 Java 包里提供的 API，可以把反射的方法分为如下三类操作：构造类操作、属性类操作以及方法类操作。

下面，我将从根据以上三类操作，从最基本的操作开始，介绍如何使用反射 API 完成常见的任务，包括创建对象实例、操作字段值以及调用任意方法等。通过一步步实例演示，相信每个人都能掌握反射的基本用法，为将来编写更加灵活强大的 Java 代码打下坚实的基础。

21.3.1 构造类操作

Java 的 java.lang.reflect 包提供了反射功能，允许你在运行时检查和操作类、接口、字段和方法。使用反射，你可以在运行时动态地创建对象、调用方法、访问字段等。java.lang.reflect.Constructor 类提供了动态创建类实例的方法。流程大致有如下三个主要步骤：

1. 获取目标类的 Class 对象，可以使用 Class.forName("packageName.className") 或者 ClassName.class 来获取。

2. Class 类提供了 getConstructor() 和 getDeclaredConstructor() 方法来获取构造函数。这两个方法之间的区别在于它们对构造函数的访问权限要求不同。

   o getConstructor() 方法只能获取公共（public）的构造函数，即可以被外部访问的构造函数。

   o getDeclaredConstructor() 方法可以获取所有构造函数，包括私有（private）、受保护（protected）和默认（package-private）的构造函数。

   获取构造函数的方法如下：

```
// 获取公共构造函数
Constructor<MyClass> publicConstructor =
            MyClass.class.getConstructor();

// 获取私有构造函数
Constructor<MyClass> privateConstructor =
      MyClass.class.getDeclaredConstructor();
```

   值得注意得是：如果使用 getDeclaredConstructor() 方法获取私有构造函数，则需要在调用 newInstance() 方法创建实例之前，使用 setAccessible(true) 方法将其设置为可访问。

3. 通过调用 Constructor 对象的 newInstance() 方法，可以动态创建类的实例。这使得可以在运行时通过反射动态创建类的实例，而无需在编译时确定要创建的类的类型。

   理论讲完了，搞点实战：

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class ConstructorDemo {
    public static void main(String[] args) {
        try {
            // 1. 获取类对象
            Class<Player> clazz = Player.class;
```

```
            // 2. 获取构造方法对象
            Constructor<Player> constructor =
             clazz.getConstructor(String.class, int.class);

            // 3. 使用 newInstance() 方法创建对象
            Player player = constructor.newInstance(
                                        "Messi", 36);

            System.out.println(player.getName());
            System.out.println(player.getAge());
        } catch (NoSuchMethodException
            | InvocationTargetException
            | InstantiationException
            | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

21.3.2 属性类操作

Java 的 java.lang.reflect.Field 类允许你在运行时动态地访问和操作类的属性。以下是使用反射操作属性的一般流程：

1. 获取目标类的 Class 对象，可以使用 Class.forName("packageName.className") 或者 ClassName.class 来获取。

2. 使用 Class 对象的 getField() 或 getDeclaredField() 方法来获取目标属性。

   o getField() 方法用于获取公共（public）的属性。

   o getDeclaredField() 方法用于获取所有属性，包括公共的、受保护的、默认的（包级别的）和私有的属性。

3. 通过 Field 对象的 get() 和 set() 方法来获取和设置属性的值。

通过这些步骤，你可以在运行时通过反射动态访问和操作类的属性，而无需在编译时确定属性的名称和类型。接下来，演示一下反射的用法。

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

import java.lang.reflect.Field;

public class FieldReflectionExample {

    public static void main(String[] args)
                                    throws Exception {
        // 获取 Player 类的 Class 对象
        Class<Player> playerClass = Player.class;

        // 获取 name 属性
        Field nameField =
                playerClass.getDeclaredField("name");

        // 设置私有属性可访问
        nameField.setAccessible(true);
        System.out.println("name 属性类型: "
                        + nameField.getType());

        // 创建 Player 对象
        Player player = new Player(" 梅西", 36);

        // 获取 player 对象的 name 属性值
        String name = (String) nameField.get(player);
        System.out.println("player 的 name: " + name);

        // 设置 player 对象的 name 属性值
        nameField.set(player, "C 罗");
        System.out.println("player 的新 name: "
                                    + player.getName());

        // 获取 age 属性 (private)
        Field ageField =
                playerClass.getDeclaredField("age");
```

```
        // 设置私有属性可访问
        ageField.setAccessible(true);

        // 获取 player 对象的 age 属性值
        int age = (int) ageField.get(player);
        System.out.println("player 的 age: " + age);

        // 获取 Goalkeeper 类的 reactionSpeed 属性
        // (private final)

        Class<Goalkeeper> goalkeeperClass
                          = Goalkeeper.class;

        Field reactionSpeedField = goalkeeperClass
                  .getDeclaredField("reactionSpeed");
        reactionSpeedField.setAccessible(true);

        // 创建 Goalkeeper 对象
        Goalkeeper goalkeeper =
                  new Goalkeeper(" 诺伊尔", 37, 90);

        // 获取 goalkeeper 对象的 reactionSpeed 属性值
        int reactionSpeed =
            (int) reactionSpeedField.get(goalkeeper);
        System.out.println("goalkeeper 的 reactionSpeed: "
                                      + reactionSpeed);

        // 注意: 由于 reactionSpeed 是 final 属性, 无法修改其值
    }
}
```

这段 Java 代码演示了如何使用 Java 反射 API 操作类的成员变量，包括获取和设置私有字段的值。这里定义了一个抽象类 Player 以及它的子类 Goalkeeper，并对类中的字段进行了反射操作。

1. 首先，通过 Player.class 获取到 Player 类的 Class 对象，这是反射操作的基础。

2. 然后，调用 getDeclaredField(“name”) 方法获取 Player 类中的 name 字段，这是一个私

有（private）字段。接着调用 setAccessible(true) 方法允许访问私有 (private) 字段，这个步骤在这里也是必要的，因为反射默认不能修改 final 或 private 字段。

3. 创建一个 Player 对象，并通过反射获取和设置 name 字段的值。先调用 nameField.get(player) 获取球员的名字，然后调用 nameField.set(player, “C 罗”) 将其名字改为 “C 罗”。
4. 同样的方式，获取并输出 Player 类中私有的 age 字段的值。
5. 接下来，对 Goalkeeper 类做类似的操作，获取其私有的 reactionSpeed 字段的值。尽管 reactionSpeed 字段被声明为 private final，意味着它既不可外部访问也不可更改，但通过反射 API 依然可以读取其值。不过，由于 Java 语言规范的限制，final 字段是不能通过反射 API 进行修改的，所以代码中没有尝试去修改 reactionSpeed 的值。

总的来说，这段代码展示了如何利用 Java 反射机制打破封装性，获取和读取类中原本不受外部访问控制的字段（包括 private 和 final 字段）。值得注意地是：get() 和 set() 方法可能会抛出 IllegalAccessException 异常。如果要访问私有属性，需要使用 setAccessible(true) 方法来设置属性的可访问性。

21.3.3 方法类操作

Java 的 java.lang.reflect.Method 类允许在运行时动态地调用类的方法。以下是使用反射操作方法的一般流程：

1. 获取目标类的 Class 对象:

   可以使用 Class.forName(“packageName.className”) 或 ClassName.class 来获取。

2. 获取目标方法:

   使用 Class 对象的 getMethod() 或 getDeclaredMethod() 方法来获取目标方法。

   o getMethod() 方法用于获取公共 (public) 的方法。

   o getDeclaredMethod() 方法用于获取所有访问权限的方法，包括公共的、受保护的、默认的（包级别的）和私有的方法。

3. 创建类的实例 (如果需要):

   如果目标方法是非静态方法，需要先创建类的实例。

4. 调用方法:

   通过 Method 对象的 invoke() 方法来动态调用方法，并传递需要的参数。

通过这些步骤，您可以在运行时通过反射动态调用类的方法，而无需在编译时确定方法的名称和参数。接下来，结合前面的代码，演示如何使用反射动态访问和操作对象的方法：

```
package com.l.r; //为印刷考虑，缩短了包名，否则太长了

import java.lang.reflect.Method;

public class MethodReflectionExample {

    public static void main(String[] args)
                            throws Exception {
        // 1. 获取 Player 类的 Class 对象
        Class playerClass = Class.forName("com.l.r.Player");

        // 2. 创建 Player 类的实例（如果需要）
        Object playerInstance = playerClass
          .getDeclaredConstructor(String.class, int.class)
          .newInstance("Messi", 36);

        // 3. 获取目标方法
        // 示例 1: 调用公共方法
        Method publicMethod =
                playerClass.getMethod("publicMethod");
        // 打印 " 这是一个 public 方法"
        publicMethod.invoke(playerInstance);

        // 示例 2: 调用私有方法
        Method privateMethod =
            playerClass.getDeclaredMethod("privateMethod");
        privateMethod.setAccessible(true); // 设置私有方法可访问

        // 打印 " 这是一个 private 方法"
        privateMethod.invoke(playerInstance);

        // 示例 3: 调用静态方法
        Method publicStaticMethod =
            playerClass.getMethod("publicStaticMethod");

        // 打印 " 这是一个 public static void 的方法"
```

```
            publicStaticMethod.invoke(null);

            // 示例 4: 调用接口方法
            Method shootCommercialMethod =
                playerClass.getMethod("shootCommercial");

            // 打印 " 这是实现接口中的方法"
            shootCommercialMethod.invoke(playerInstance);
        }
    }
```

这段 Java 代码展示了如何使用 Java 反射 API 来动态调用类的方法，包括公共方法、私有方法、静态方法以及实现接口的方法。

1. 获取类对象：首先，通过 Class.forName(“com.l.r.Player”) 获取到了 Player 类的 Class 对象，这样我们就能对该类进行反射操作。
2. 创建类实例：接着，通过反射获取 Player 类的构造函数（这里假设有一个接受 String 和 int 作为参数的构造函数），并调用 newInstance() 方法创建一个 Player 类的实例，传入参数”Messi” 和 36，初始化对象。
3. 调用公共方法：
   - 使用 getMethod(“publicMethod”) 获取 Player 类的名为 publicMethod 的公共方法。
   - 调用 invoke() 方法执行该方法，传入实例对 playerInstance。假设 publicMethod 是一个无参的公共方法，它会在控制台打印字符串 “这是一个 public 方法”。
4. 调用私有方法：
   - 使用 getDeclaredMethod(“privateMethod”) 获取 Player 类的名为 privateMethod 的私有方法。
   - 由于私有方法默认不可见，我们需要调用 setAccessible(true) 使其可访问。
   - 然后同样使用 invoke() 方法执行该私有方法，也会在控制台打印相应字符串 “这是一个 private 方法”。
5. 调用静态方法：
   - 使用 getMethod(“publicStaticMethod”) 获取 Player 类的名为 publicStaticMethod 的公共静态方法。
   - 在调用 invoke() 方法时，由于静态方法不需要实例对象，所以我们传入 null。
   - 静态方法被执行并在控制台打印相应字符串 “这是一个 public static void 的方法”。

6. 调用接口方法：

   - 假定 Player 类实现了某个接口，并实现了接口中的 shootCommercial 方法。
   - 使用 getMethod(“shootCommercial”) 获取到这个接口方法。
   - 通过 invoke() 方法调用该接口方法，传入 playerInstance 作为执行对象，因为在接口方法的实现中，实际逻辑是由具体类（这里是 Player）完成的，所以需要传入其实例。
   - 控制台会打印出接口方法实现的字符串 “这是实现接口中的方法”。

通过以上步骤，代码演示了如何使用反射 API 在运行时根据方法名称动态地调用类的各种方法。这种机制打破了 Java 的正常访问权限规则，使程序在运行时具有高度的灵活性和动态性，但同时也需要谨慎使用，因为它可能导致安全问题和违反封装原则。

小结一下，基础反射 API 在 Java 中提供了在运行时访问和操作类信息的能力，包括获取类的结构信息、创建实例、调用方法和访问字段。核心类和方法包括 Class、Method、Field 和 Constructor。然而，反射操作相比直接调用方法更慢，可能绕过访问控制，且代码可读性较低，应谨慎使用以避免性能问题和安全漏洞。

技术发展和性能优化需求催生了更精炼高效的运行时操作机制：MethodHandle 和 VarHandle。接下来，我们进入更高阶的反射 API 的学习。

21.4 高阶的反射 API

java.lang.invoke.MethodHandle 是从 Java 8 开始引入的一个新的功能性特性，它是 JVM 层面的轻量级方法调用机制。MethodHandle 可以看作是对方法的高效、类型安全的引用，它可以用来直接调用实例方法、静态方法、构造方法以及其他 JVM 支持的操作（如数组读取和转换）。相比于传统的反射 API，MethodHandle 提供了更低的运行时开销和更高的灵活性。

通过 MethodHandle，开发者可以在运行时动态链接到特定的方法，并且能够调整方法调用的行为，例如改变调用者和接收者的上下文（即方法的 this 引用）、绑定方法参数等。由于它设计得更加底层和接近 JVM，因此性能损失比反射要小得多，特别适用于需要高性能动态调用场景。

java.lang.invoke.VarHandle 是在 Java 9 中进一步引入的一种类型安全的变量访问机制。类似于 MethodHandle 对方法的引用，VarHandle 提供了一种对变量的直接、低层次的访问方式，可以用于读取和修改对象字段、数组元素以及类似的数据结构。

VarHandle 旨在支持原子性和及各种内存模型下的同步操作，这使得它非常适合用于实现线程安全的数据结构和低级别并发控制。使用 VarHandle，开发者可以直接操作字段，执行无锁编程或者构建高级并发原语，其性能优于传统的反射或者其他基于 JNI 的方式

(JNI 方式是指 Java Native Interface 的方式，它是一种允许 Java 代码与本地代码（C/C++）相互调用的机制)。接下来，演示如何使用 VarHandle：

```
package com.l.r;

import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodType;
import java.lang.invoke.VarHandle;
import java.lang.reflect.Field;

public class HandleDemo {
    private int value = 123;

    public static void main(String[] args)
                                throws Throwable {
        // 使用 MethodHandle
        MethodHandleDemo();

        // 使用 VarHandle
        VarHandleDemo();
    }

    public void setValue(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }

    public static void MethodHandleDemo() throws Throwable {
        MethodHandles.Lookup lookup =
                    MethodHandles.lookup();

        MethodHandle setMH =
              lookup.findVirtual(HandleDemo.class,
```

```
                "setValue", MethodType.methodType(void.class,
                int.class));

        MethodHandle getMH =
                lookup.findVirtual(HandleDemo.class,
                "getValue", MethodType.methodType(int.class));

        HandleDemo obj = new HandleDemo();

        // 调用 setValue 将 value 设置为 999
        setMH.invoke(obj, 999);

        // 调用 getValue 获取 value 的值
        int value = (int) getMH.invoke(obj);

        System.out.println("MethodHandle: " + value);
    }

    public static void VarHandleDemo() throws Throwable {

        VarHandle valueHandle;
        Field valueField =
            HandleDemo.class.getDeclaredField("value");

        valueHandle = MethodHandles.lookup()
                        .unreflectVarHandle(valueField);

        HandleDemo obj = new HandleDemo();

        // 直接设置 value 值为 888
        valueHandle.set(obj, 888);

        // 直接获取 value 的值
        int value = (int) valueHandle.get(obj);
        System.out.println("VarHandle: " + value);
    }
```

```
}
```

这段 Java 代码演示了如何使用 Java 的 java.lang.invoke.MethodHandle 和 java.lang.invoke.VarHandle 来操作类的方法和字段。这两个 API 都是 Java 8 之后引入的 Java 虚拟机（JVM）层面的高级反射机制，旨在提供比常规反射 API 更高效、类型安全的方式来操作方法和字段。

先来说 MethodHandleDemo() 方法：

1. 使用 MethodHandles.lookup() 方法获取 Lookup 对象，它是查找和操作方法句柄的主要入口。
2. 调用 findVirtual() 方法获取到 HandleDemo 类中 setValue 和 getValue 方法对应的 MethodHandle 实例。
3. 创建 HandleDemo 对象，并通过 MethodHandle 实例 setMH 调用 setValue 方法，将 value 字段设置为 999。
4. 再通过 getMH 获取 getValue 方法的返回值（即 value 字段的当前值）并打印。

随后是 VarHandleDemo() 方法：

1. 获取 HandleDemo 类中 value 字段的 Field 对象。
2. 使用 MethodHandles.lookup().unreflectVarHandle() 方法从该字段创建一个 VarHandle 实例，可以直接访问和修改字段值。
3. 创建 HandleDemo 对象，并通过 VarHandle 实例 valueHandle 直接设置 value 字段的值为 888。
4. 再通过 valueHandle 获取 value 字段的当前值并打印。

与基础反射 API 的比较，MethodHandle 和 VarHandle 相比传统的 Java 反射 API 在某些特定场景下具有更高的性能和类型安全性，但它们也要求开发者对 JVM 级别操作有更深的理解，并且在一些简单场景下，基础反射 API 的使用可能更加直观易懂。

简而言之，MethodHandle 专注于方法调用，而 VarHandle 则专注于字段的访问和更新，两者都提供了比传统反射更高效、更灵活的运行时行为控制手段。同时，它们都支持可变参数类型和灵活的调用转换，增强了 Java 的运行时元编程能力。

21.5 反射的局限

Java 反射是一种强大的机制，允许程序在运行时查询和操作类、字段、方法、注解等。然而，反射并非没有缺点，它的使用确实应当谨慎，主要基于以下几个局限和考虑：

21.5.1 性能开销问题

在 Java 中，当我们直接调用一个方法时，这个方法的调用信息（如方法名、参数类型等）在编译时就已经确定了，JVM 可以直接通过这些信息找到并执行相应的方法，这个过程非常快。

然而，当我们使用反射来调用一个方法时，情况就完全不同了。首先，我们需要获取到这个方法的 Method 对象，这需要 JVM 在运行时去解析类的信息，找到这个方法，这个过程涉及到类型解析，需要一定的时间。然后，我们还需要通过 Method 对象的 invoke 方法来调用这个方法，这个过程涉及到访问检查，JVM 需要检查我们是否有权限调用这个方法，这也需要一定的时间。最后，invoke 方法的调用还涉及到一些额外的操作，如参数包装和解包装，这也会消耗一些时间。

因此，反射调用的性能成本相对于直接调用要高。这并不是说反射就一定不好，反射提供了很多直接调用无法实现的功能，如动态调用、修改私有字段等。但是，我们在使用反射时，需要意识到它的性能成本，尽量避免在性能敏感的地方使用反射。

21.5.2 破坏封装性

在 Java 中，封装是一种将数据（变量）和操作数据的方法结合在一起的机制，其中一些数据被隐藏，不允许外部直接访问，只能通过公开的方法（getter 和 setter）进行访问。这样做的目的是为了保护数据的完整性，防止被外部代码随意修改。

然而，反射可以突破这种封装性。通过反射，我们可以在运行时动态地访问、修改和调用类的私有字段和方法，甚至可以修改 final 字段的值。这就破坏了封装性，因为它允许我们在不应该访问或修改的地方访问或修改数据。

例如，假设我们有一个类，它有一个私有字段，我们不希望这个字段被外部代码访问或修改。但是，如果外部代码使用反射，它可以轻易地访问和修改这个字段，这可能会导致数据的不一致，引发错误。这就是为什么说反射破坏了封装性，使得代码变得难以管理。因为一旦我们开始使用反射，我们就失去了 Java 提供的封装性保护，我们需要自己来确保我们的代码不会误用反射带来的能力。

21.5.3 反射代码往往难以维护

反射代码通常比非反射代码更难以理解和维护，原因主要有以下几点：

首先，反射代码在运行时动态地获取类的信息和调用方法，这使得代码的行为在编译时无法确定，只能在运行时才能知道。这种动态性使得反射代码难以理解和预测。

其次，在反射代码中，很多操作（如获取字段、调用方法等）的参数都是字符串，这些字符串在编译时无法进行类型检查，只能在运行时才能知道是否正确。这使得反射代码容易出错，而且错误往往在运行时才能发现。

再次，反射代码可能会抛出很多类型的异常，如 NoSuchMethodException、IllegalAccessException 等。这些异常需要在代码中进行处理，这使得反射代码变得更加复杂。

最后，反射代码通常包含很多字符串和复杂的 API 调用，这使得代码的可读性较差，难以理解。

因此，虽然反射提供了很多强大的功能，但是我们在使用反射时，需要注意它的这些问题，尽量使反射代码保持简洁和清晰，以便于理解和维护。

后记

正当本书接近尾声之际，科技界发生了一件意义重大的事件：OpenAI 推出了大型语言模型。随之而来的一种观点认为，程序员将被人工智能取代，未来不再需要程序员。

果真如此吗？如果你使用过由 GPT 4 驱动的 GitHub Copilot 一段时间，就会发现目前的人工智能还远远达不到独立设计软件的水平。然而，人工智能可以帮助程序员提升效率，成为更优秀的程序员。

如果出现一项新技术，而你幻想它能解决你所有的问题，那么你注定会失望。因为世间万物皆有其规律，正如天下的盐都是咸的，所有的项目都是难的。

如果出现一项新技术，而你只是想用它解决一部分痛苦，比如 Java 相对冗长的语法，而人工智能可以协助你输入这些语句，那么你应该会感到很欣慰。

至于程序员失业的担忧，哪个行业又能保证永远不会被淘汰呢？况且，工作只是为了赚点钱，绝对不会多到让人无需工作。退一万步说，如果老板认为人工智能可以取代程序员，那么，不如咱们当老板吧，利用人工智能来帮咱们写软件！