MATLAB®&Simulink®
工程师系列丛书

MATLAB面向对象编程
——从入门到设计模式

徐 潇　李 远　编著

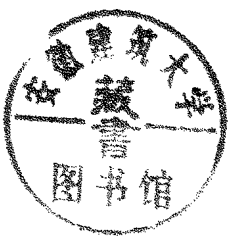
北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

MATLAB® examples

MATLAB®&Simulink®工程师系列丛书

MATLAB面向对象编程
——从入门到设计模式

徐 潇 李 远 编著



北京航空航天大学出版社

内 容 简 介

本书分为三部分。第一部分介绍 MATLAB 面向对象编程基础知识，包括什么是类，类之间的基本关系，以及 MATLAB 提供的面向对象编程语言的具体功能；第二部分是面向对象编程的进阶篇，对概念进行归类，方便读者在编程中遇到问题时查询和检索；第三部分把面向对象编程的方法应用到实际问题中，并且从实际问题中抽象出一般的解决方法，即设计模式。

本书可作为高等院校本科生、研究生 MATLAB 课程的辅助读物，也可作为从事科学计算、程序设计的科研人员的参考书。

图书在版编目(CIP)数据

MATLAB 面向对象编程：从入门到设计模式 / 徐潇，李远编著. -- 北京 ：北京航空航天大学出版社，2015.1

ISBN 978-7-5124-1609-3

Ⅰ. ①M… Ⅱ. ①徐… ②李… Ⅲ. ①Matlab 软件—程序设计 Ⅳ. ①TP317

中国版本图书馆 CIP 数据核字(2014)第 241311 号

MATLAB 面向对象编程：从入门到设计模式

徐 潇 李 远 编著

责任编辑 刘亚军 栾京辉 刘亚平

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191) http://www.buaapress.com.cn

发行部电话：(010)82317024 传真：(010)82328026

读者信箱：goodtextbook@126.com 邮购电话：(010)82316524

北京时代华都印刷有限公司印装 各地书店经销

*

开本：787×1 092 1/16 印张：22 字数：535 千字

2015 年 1 月第 1 版 2015 年 1 月第 1 次印刷 印数：4 000 册

ISBN 978-7-5124-1609-3 定价：46.00 元

MATLAB 中文论坛创始人 math：该书的第一位受益者（代序）

2009 年末，我应中国科学院南京土壤研究所（简称土壤所）的邀请，与他们的科研人员一起开发“土壤红外光谱信息系统”。这个系统非常复杂，它涉及中国海量土壤光谱数据的快速存储和读取，数据处理算法的开发、调试和验证，以及客户端多界面（GUI）的开发。其中，数据存取使用的是 MATLAB 数据库工具箱和 MySQL 数据库；数据处理算法（包括数据的滤波处理、降维、数据的匹配、预测等）使用的是 MATLAB 统计学工具箱、优化工具箱和神经网络工具箱；客户端的界面非常多，如数据库的可视化操作、算法参数的在线调试以及数据处理结果的展示等，所有的界面都是使用 MATLB GUIDE 完成的。从把系统的要求整理出来，到系统第一个版本的完成，用了将近 1 个月的时间。由于是密集型开发，所以在这段时间内我对整个系统的流程、架构非常熟悉，因此开发起来也不是特别困难。该系统在 2010 年获得了中国软件的著作权（编号：2010R11L027920）。

2013 年，土壤所再次邀请我。他们想对这个系统进行升级，并做成网络版——只要用户能连接网络并且使用 MATLAB，就能使用这个系统，使之不仅仅局限于在土壤所使用，更希望它能服务于所有的科研人员，同时给该系统增加多种算法。土壤所成立了专门的研究小组以开发和维护此系统。当我在思考如何指导该研究小组升级系统时，我面临一个比较棘手的问题：如果对原系统进行升级，需要改动的地方特别多。因为数据的读写、算法的运用以及界面展示这三者之间是高度耦合的，很多函数的实现都是在 MATLAB GUIDE 的回调函数里完成的。对于一个复杂的系统来说，一个地方的小小改动，通常需要测试整个系统架构和算法的稳定性，而且这也不利于系统更新。那么，科研人员有了新的数据匹配的算法，如何通过改变最少的代码来实现新的算法，同时又能保证系统的完整性和可靠性呢？

有一天我跟徐潇一起吃午饭，跟他分享了我遇到的问题。徐潇告诉我，软件设计中，解决这个问题的标准方法是使用面向对象编程和 MVC（Model-View-Controller，模型-视图-控制器）模式。虽然看起来这有点浪费以前的代码，但对于系统的长远稳定性和易维护性来说，这是大型系统的不二选择。而且他正在写一本关于 MATLAB 面向对象编程的书。他说，如果我感兴趣，他可以单独用一章专门来写如何基于 MATLAB 面向对象编程实现 MVC。我说好，你写好我第一个使用。两个星期后，徐潇发给我一个 PDF 文件，以非常通俗的例子诠释了如何实现 MVC 的过程，就是大家现在所看到的该书第 7 章：分离用户界面和模型。我在使用的过程中，充分地感受到了 MATLAB 面向对象编程的强大。我大概花了 10 小时的时间，就把 2009 年的系统架构改成了 MVC 的架构。2013 年 8 月，我把新的架构展示给了土壤所负责系统开发和维护的研究小组。该研究小组成员对 MATLAB 语言了解不是很多，但是这

并没有阻碍他们开发系统，因为我们已经完全把算法的模型（Model）、界面视图（View）以及如何实现用户输入的获取（如键盘、鼠标事件）这三者完全分开，放在了不同的类（Class）中。研究小组在一个星期之内就掌握了系统的架构，并且能独立地对系统进行开发和维护。

以上是我的亲身经历。我已把我的经历发表在我的个人博客里。如果对此文有任何疑问，可以在我的博客里给我留言。该文网址：http://www.ilovematlab.cn/blog-2-73.html。

math, 博士、教授、MATLAB 中文论坛独立创始人（www.ilovematlab.cn）

前　言

本书的编写从2011年4月开始，到2013年末结束，历时两年半。本书从理工科研究人员和学生的角度出发，通过三个进阶篇介绍MATLAB面向对象编程。

编写本书的难点是不仅需要介绍面向对象编程的思想和技巧，还要让非计算机专业的读者领会为什么需要面向对象编程，它对我们的科研工作将有什么样的帮助，并且怎样把面向对象的思想应用到科研程序中。

我们的写作理念是：技术实用，重点突出，代码简单易读，内容讲解图文并茂。

一本技术书籍，纯粹的文字叙述是必要的，因为文字叙述是最精确的；一本介绍编程的书，如果尽量提供例子代码，则能够帮助读者更深刻地理解文字概念；“一张图可胜过千言万语”，简洁明了的图表可以直观形象地表达文意。因此，我们不仅尽量使用最通俗的语言和最形象的图表阐述道理，以最典型且简洁易读的代码作为例程，全面讲解MATLAB面向对象编程从入门到设计模式，而且尽量让版式和代码的编排使读者看起来容易，以便带给读者最佳的阅读体验。除此之外，我们还加入了大量的面向对象编程的图形用户界面（UML），与我们所提供的代码相互对应，以反映代码中类、对象、属性、方法之间的关系。

为了平衡各专业的需求，书中所列举的例子大多是“通例”，而不是具体到某个专业领域的专题。但是本书作者也十分清楚，一本书要能够写好，需要的是能够“深入骨髓”到读者所遇到的最具体的专业问题，最好有对应范例供本专业的同仁参考。所以，在此也希望读者能够将“面向对象编程”的专业问题的程序以及产生的问题发布在MATLAB中文论坛本书的版块（http://www.ilovematlab.cn/forum-219-1.html）上。日积月累，论坛上一定会有更多的MATLAB面向对象编程范例可以参考，也会有更多的科研新人受益其中。

由于作者水平有限，书中存在的错误和疏漏之处，恳请广大读者和同行批评指正。本书勘误网址：http://www.ilovematlab.cn/thread-310165-1-1.html。

作　者

导　读

本书面向的读者既包括理工科研究人员和学生，也包括希望使用 MATLAB 构建高速、高效、高级的系统平台的用户。这些读者也许心中牢记着那句古老的工程谚语“如果程序没有坏，就不要动它”。有人也许会有这样的疑问：学习面向对象编程真的有必要吗？学习面向对象编程浪费时间吗？那么，这篇导读将回答这些疑问。

问：目前图书市场中有关 MATLAB 的书籍已经很多了，为什么还要写这本 MATLAB 面向对象编程的书？

答：区别于目前图书市场中其他的 MATLAB 语言编程和专业工具箱 MATLAB 编程的书籍，本书是第一本中文版 MATLAB 面向对象编程的书籍。我们更注重的是利用 MATLAB 提供的面向对象编程的语言来介绍 MATLAB 的编程思想，从而帮助读者提高对于 MATLAB 编程的运用深度。

问：我是理工科学生，MATLAB 对我来说很简单，为什么我还要学习 MATLAB 面向对象编程？

答：虽然 MATLAB 提供给用户的语法是简单的，使得用户上手快，但是这并不代表我们要解决的科研问题的方法是简单的。除了常用功能之外，MATLAB 还有很多强大的功能有待我们学习和运用，从而解决更复杂的问题。本书主要面向的读者群中包括理工专业的学生、学者，我们希望通过介绍 MATLAB 面向对象编程来帮助他们更好地解决科研中的问题。或许你曾有这种感觉：在科研和学习中，所写的程序一旦到达一定的规模，维护起来就会很困难，调试越来越慢。随着科研项目不断有新的要求，程序需要不断地修改和扩展；函数多达上百个；一旦有修改，则牵一发而动全身；有的时候，一个小的扩展甚至都需要做伤筋动骨的修改。MATLAB 面向对象编程和设计就是专门帮你解决这种问题的。本书的重点不是介绍某个函数或者技巧，而是介绍怎样从整体上去设计程序，小到一个家庭作业、一两个星期的项目，大到硕士或者博士的毕业设计、多人合作的项目。面向对象的思想会把你从繁重的程序维护中解脱出来，让你的注意力集中于真正需要解决的问题上，把需要解决的问题解决好。我们不是为了学习面向对象编程而学习面向对象编程，作为科研人员，我们都以高效务实为目标，如果一种技术能够让我们仅投入少量的时间去学习，并且学会了之后能让我们的科研工作如虎添翼，让我们有更多的时间去做其他事情，那么何乐而不为呢？

问：面向对象编程难道不是只有计算机专业的人才用的吗？

答：因为面向对象可以更好地解决软件设计问题，所以面向对象编程语言是计算机专业背景科研人员的一个自然选择。但是面向对象的方法并不是软件行业所独有的，任何学术背景的研究人员都可以使用面向对象编程，去解决各自行业的学术问题。目前主流的面向对象编程语言（如 C++ 和 Java）学习周期比较长，烦琐的语法将面向对象的方法和设计思想隐藏了起来，大多数非计算机专业背景的研究人员没有时间和精力先熟练掌握 C++ 和 Java 的语言，然后再学习面向对象的编程思想，进而用到实际的科研工作中来。其实在工程科学计算中，MATLAB 才是主流的语言。MATLAB 从 R2008a 之后开始提供新的面向对象的编程

方法，给用户提供了一个宝贵的机会，能够避开烦琐的语法，直接接触到核心的面向对象编程和设计模式的思想。所以，使用 MATLAB 语言，不具备计算机的专业知识也能学会面向对象编程和设计模式，而本书将成为你掌握它们的一座桥梁。

问：学习 MATLAB 面向对象编程需要有什么样的基础？

答：本书的第一部分就是要让具有初级的 MATLAB 语言基础的读者能够迅速且一步到位地把面向对象的思想渗透到自己的编程习惯中去。其实，只要懂得什么是变量，什么是函数，就完全能够开始学习 MATLAB OOP 了。对于有经验的读者，就是那些熟悉 MATLAB 语言和各种工具箱（Toolbox）的读者，本书的中级篇和设计模式篇能够使其更深入地了解 MATLAB 的体系，提高自己对程序的总体设计能力，做到事半功倍。

问：学习面向对象编程是否要花很多时间？我还有研究课题要做，没有那么多时间怎么办？

答：本书的作者都是理工科的背景，十分懂得如何用最少的时间学习最多的知识，也深知怎样有效地引导初学者成为精通者。我们期望的是让读者用最短的时间入门面向对象编程，以最小的成本学会面向对象编程的中级基础，并且能够顺利地进入到编程思想的学习当中去，越过面向对象编程语法上的障碍，真正地使用面向对象的编程方法。我们还尽量地让书中内容的编排便于查找，读者可以跳跃性地阅读自己所需要的内容。当工作变得复杂，需要更多 MATLAB 面向对象编程语言特性时再回过头来查找。

问：面向对象编程难学吗？我要学多久才能把它用到实际的编程中？

答：学习的难易与否主要看基于什么编程语言，目前主流的面向对象的编程语言，如 C++ 和 Java，语法和编译细节很烦琐，使得面向对象的思想被隐藏了起来。然而，MATLAB 的面向对象语言的支持提供了前所未有的机会，让我们能够迅速地越过这些障碍，真正学到编程的思想。众所周知，学习一门程序设计语言不但需要学习语言的语法，还需要不停地实践。本书将引导读者将这种编程思想融入到具体的程序书写中，并立刻将其应用到自己的编程中去，哪怕是一个简单的曲线拟合、图像生成和优化。另外，把已有的程序转化成面向对象的程序也不是一件麻烦的事情。我们在附录中将通过一个综合实例介绍如何把一个中型规模的 MATLAB 面向过程的程序转成面向对象的风格。总而言之，自己的科研课题就是实践编程思想的最好平台，好的编程思想可以让科研工作事半功倍。MATLAB 作为一种高级的工程科学计算语言，提供了在以往只有 CS 专业背景的人才能够具有的实现编程思想的机会。

问：采用面向对象的方法会不会降低我的编程速度？

答：良好的设计才是快速开发的根本。如果没有良好的设计，或许在一段时间之内，使用面向过程的方法编程进展很快，但是糟糕的设计会很快让速度慢下来。因为面向对象编程需要花大量的时间在调试程序上面，而无法添加新的功能，最终修改的时间将越来越长，最初的程序中被打上一个又一个的补丁，新的特性需要更多的代码才能实现。而面向对象的编程方法可以有助于提高程序设计的质量，从而加快开发速度。

问：MATLAB 的面向对象编程与 C++ 和 Java 的面向对象编程有什么不同吗？

答：MATLAB 是一款商业软件，提供面向对象编程的支持，这与 C++ 和 Java 有本质的不同；C++ 和 Java 给用户提供“基石”，用户需要花大力气去首先学习其语法，然后学用这

些“基石”的组合来解决复杂的问题，这需要深厚的基础知识和大量的时间，而大部分的科研工作者没有这样的时间和精力去专门学习一门语言来帮助他们解决问题。MATLAB 提供了这样一种渠道：把这些基石进行复杂的组合，然后当做语言的特殊功能提供给用户。用户只需要对这些特殊功能稍加了解，就可以很快掌握，并能在有限的时间内，以最高的效率完成任务。本书还会简单解释这些特殊的功能来自何方，大概是怎样实现的，目的是为了帮助读者更好地理解和使用这些功能。

问：面向对象编程和书中的设计模式是一回事吗？

答：面向对象是相对于面向过程的一种编程方式，是一种系统化编程的思路，教用户一开始就去系统化地设计程序。设计模式是建立在面向对象基础之上的针对一些常见的复杂问题的核心解决方法[①]。问题再复杂，都可以被分解成小的部分加以抽象，然后使用设计模式来高效地解决。有时解决问题的方法甚至可以是多种设计模式的结合，如果你能把这些套路使用在自己的科研工作编程中，你的科研工作必将如虎添翼。

问：MATLAB 面向对象编程不是有一本英文的用户手册吗，你们的这本书和这本英文手册比有什么优点？

答：本书部分参考了英文 MATLAB 面向对象编程用户手册中的内容，并且在此之上做了大量的改进，使其更适合理工科用户学习和阅读。具体说来：第一，这本用户手册有 600 多页英文，通读起来不是一件容易的事情；第二，它只介绍了 MATLAB 基本的面向对象技术，没有介绍设计模式，而设计模式才是真正利用 OOP 的试金石；第三，因为 OOP 和设计模式已经是很成熟的技术，我们在向读者介绍编程思想时，还参考了大量 C++ 和 Java 面向对象编程和设计模式的书籍。我们相信这本中文的 MATLAB 面向对象编程将比英文手册更加适合中文读者，而且学习和阅读的成本很低，我们的目的就是让读者花很少的时间和精力去学习并且掌握 MATLAB 面向对象编程。

最后，给出学习 MATLAB 面向对象编程的诀窍：

> 无论学习何种编程语言，最重要的都是实践。如果你能把本书中所有的例子都亲自通过键盘输入到计算机并运行一遍，那么你就一定能学会 MATLAB OOP 和设计模式了。

①Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice – Christopher Alexander(1977).

致《MATLAB 面向对象编程从入门到设计模式》试读版的读者

在过去的几个月内，即本书的修改过程中，我们收到了来自 ilovematlab 网站中本书试读版 25 名志愿读者的阅读体验和建议。其中包含了大到章节设置，小到标点符号、错别字的纠正等宝贵的建议和意见。在此，我们对所有提供阅读体验和建议的试读版志愿读者表示最诚挚的感谢。

作为感谢，我们也将在这里鸣谢大家的关心与支持。本书作者已经细心研读了汇总来的 25 名志愿读者的意见和建议，并依照这些意见和建议对书稿进行了修改，希望大家可以看到一本更有针对性、更贴近读者需求的《MATLAB 面向对象编程从入门到设计模式》。

最后，我们十分感谢 math 和他的网站的支持，十分感谢北京航空航天大学出版社允许我们开展小范围的试读工作。

在此，特别感谢 ilovematlab 网站中下列 ID 的读者（按字母排序）：

alazong，aqq，dabfxz，gnnaaayyyy，godspeed，hasen-chen，HEYsir，hkbl1988，jianfeidahai，lhwzz，liuyd07-20120625，流萧，matlab 情，南海小兵，paulke2011，qfwanglz，renxing1999，wangyude，wenquan，wjxchina，wukaiwukai，wusuo_2005，wyude，xudongmin99，ylyy779530170。

徐潇　李远

2013 年 1 月 3 日

目　录

第 1 部分　面向对象编程初级篇

第 2 部分 面向对象编程中级篇

附　录

第 1 部分

面向对象编程初级篇

第Ⅰ部分

面向对象编程概述

第 1 章　面向过程和面向对象程序设计

1.1　什么是面向过程的编程

“面向过程的程序设计”(Procedural Programming) 是一种以过程为核心的编程方法。使用该方法解决问题的关键是，先把问题的过程按照步骤分解出来，然后用函数 (Function) 的形式把这些步骤加以实现，并且依次调用它们。只要不是面向对象编程，一般来说都是面向过程的风格。面向过程编程方法的优点是简单快捷，缺点是面对复杂的程序难以修改和维护，下面举例说明。

假设要用 MATLAB 来模拟一个面馆的经营：要写一段程序，模拟顾客点菜和面馆做面条的过程（读者也可以把面条想象成数据，做面条好比是用函数对数据做一系列的处理）。先从简单的情况开始，假设做面条可以分解成如下步骤：和面，拉面，煮面，烧汤，等等。面向过程的方法做一碗汤面的 MATLAB 代码如下：

Script

```
1 dough         = prepareDough() ;              % 把面粉和成面团
2 noodle        = prepareNoodle(dough) ;        % 把面团擀成面条
3 boildedNoodle = boilNoodle(noodle) ;          % 把面条煮熟
4 soup          = prepareSoup() ;               % 准备面汤
5 noodlesoup    = mix(boildedNoodle,soup) ;     % 把面条倒入汤中
```

上述一系列函数调用对应做面条的各步骤，解释如下：

- 第 1 行调用准备面团函数 prepareDough() 的返回值是：面团 (dough) 变量。
- 第 2 行把刚得到的面团作为拉面函数 prepareNoodle() 的输入，而拉面函数的输出是擀好的面条 (noodle)。
- 第 3 行把面条 (noodle) 提供给煮面函数 boilNoodle()，得到煮熟的面条。
- 第 4 行做汤函数 prepareSoup() 返回做好的汤。
- 第 5 行调用 mix() 函数，负责把煮好的面条和汤混到一起。

把这个做面条的过程抽象出来，如图 1.1 所示：“面向过程的程序设计”以函数为中心，函数操纵数据，通过数据在多个过程直接传递共享来完成过程的模拟，函数和数据是分开的。这也是典型的工程科学计算中的模式，即从原始数据开始 (如信号、图像、矩阵)，然后用 MATLAB 函数对它们做处理和分析，函数调用结束得到计算结果。

图 1.1　面向过程以函数为中心，数据在函数中传递，数据和函数是分开的

MATLAB 提供了简单的语言和广泛的算法支持，用户可以用脚本、GUI 调用函数解决工程科学计算中的各种复杂问题。但是，随着科研问题的愈加复杂，程序不可避免地也会变得越来越复杂，修改起来也越来越困难。在后面，读者将会看到，面向过程的编程方式随着

频繁的程序修改和扩展，渐渐地显得捉襟见肘起来。

1.2 什么是面向对象的编程

和面向过程以函数为中心相比，面向对象编程 (Object Oriented Programming，OOP) 把任务分解成一个个相互独立的对象 (Object)，通过各对象之间的组合和通信来模拟实际问题。

1.2.1 什么是对象（Object）

OOP 中的对象指的是真实世界中具体的东西，比如一只狗、一辆汽车、一个坐标轴、坐标轴上的一个点、一条线等，即生活中一切有形或者无形，可以具体标识的事物，并且 OOP 中的对象以及真实世界中的事物，都有如下特点：

- 具体的事物都有各种属性，比如狗的名字，坐标轴上线段的长短，位置等。在 OOP 中，把它们定义成对象的属性（Property）。
- 具体的事物还具有相关的行为 (无论是主动的还是被动)，比如狗叫，汽车被驾驶，坐标轴上的线的颜色被改变。在 OOP 中，把这种行为定义成对象的方法 (Method)。

1.2.2 什么是类（Class）

简单来说，类（Class）就是对各个具体、相似对象的共性的抽象。比如，可以把人的共性抽象出来，用一个类来形容，如图1.2所示。

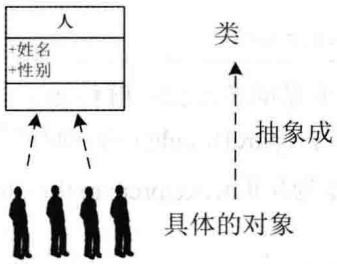


图 1.2 类是一种抽象：把个体的共性抽象出来

从这个角度来说，先有对象，再有类，类是对象共性的一种总结。

从另一个角度来说，也可以先有类，再有对象。构建新的具体对象，必须基于“模板”，这个“模板”就是“类”。比如，汽车设计师设计汽车蓝图，蓝图就是类，规定汽车高度、外观等，工厂工人根据蓝图，造出来的一辆辆具体的汽车就是对象，如图1.3所示。

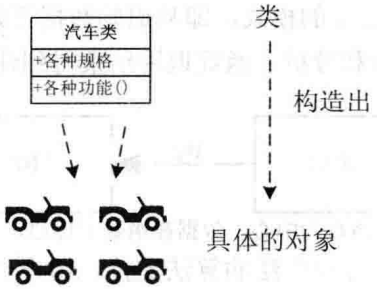


图 1.3 类是一种规范：个体根据规范被制造出来

总地来说，类用来规定一些相似的对象所具有的属性，以及它们的职责和行为，并且把它们封装；更重要的是，类提供了将数据和函数结合起来的方式，数据变成了类的属性，而函数变成了类的方法。

在 MATLAB 中，类和对象的概念无处不在，只不过大多数情况下，我们习惯了通过面向过程的方式使用它们。比如，用 figure 命令画一个图形窗口：

Script

```
>> f = figure ;
```

其中 figure 就是一个类的名称，而返回的 f 就是 figure 对象。

又比如，我们使用数据采集工具箱：

Script

```
s = daq.createSession('ni') ;
```

其中 createSession 方法返回的就是一个 session 对象，该对象介于 MATLAB 和数据采集卡之间，用来传输数据和对采集卡进行控制。

再比如，使用 MATLAB 定时器：

Script

```
>> t = timer ;
```

其中 timer 就是一个类，而调用 timer 命令返回的 t 是一个 timer 对象。

下面举一个抽象的例子：如何从具体的对象中抽象出共性并使其成为一个类。现在把二维坐标中的点都作为对象，为了创建二维点的类，首先要概括这些点的共同的、抽象的特征，比如每个点 (Point) 都有 x，y 坐标。除了特征，还要抽象出和这个点相关的行为，比如这个点的坐标，做归一化的操作 (Normalize)。通常用图1.4左来表示一个类，该图中有三个格子：

- 第一个格子指出了类的名字。
- 第二个格子指出了类所包含的属性：x 和 y 坐标，也可以叫做成员变量。
- 第三个格子指出了类所支持的行为 normalize()，也叫做成员方法。

用图1.4右来表示各个对象，即具体的点，各个对象之间是独立的。

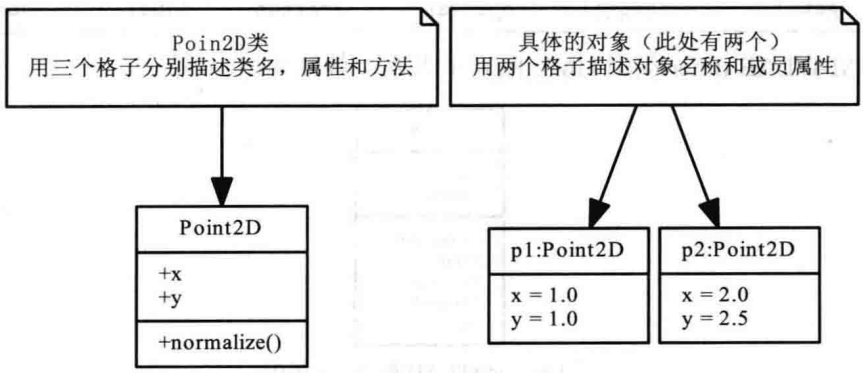


图 1.4 类和对象：Point2D 类和 Point2D 对象

类的概念可以广泛地应用到工程科学计算程序中。比如，可以把数据以及数据相关的算法封装在类中，可以把要控制的硬件驱动封装在类中，可以用类来架构复杂计算的流程，还可以用类来组织 MATLAB GUI 程序等。

1.2.3 什么是统一建模语言（UML）

如图1.4、1.5 所示的这种表示类和对象的图叫做 UML（Unified Modeling Language，统一建模语言）类图，它是一种对程序的图形表达方式。

在 UML 类图中，长方形表示类，长方形中分三个区域，从上到下分别是类名、成员变量和成员方法，属性和方法前面的加号表示它们是公有的[①]。另外，图1.4中右边的两个长方形用来表示具体的对象，在 UML 图的对象表示方法中，规定只写对象的名称和对象的属性，不写成员方法。

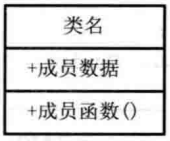


图 1.5 UML 中类的表示方法

给定一个对象，可以使用函数 properties 来查询对象所具有的属性；使用 methods 来查询类所支持的方法，比如下面检查 timer 类的对象 t 所具有的属性和支持的方法：

Script

```
>> t = timer ;          % 定义 t 为 timer 类的一个对象
>> properties(t)        % 查看这个对象 t 的属性
Properties for class timer:
    ud
    jobject
>>
>>
>> methods(t)
Methods for class timer:
ctranspose   end          horzcat     isvalid     set         stop        timercb
delete       eq           inspect     length      size        subsasgn    timerfind
disp         fieldnames   isempty     ne          start       subsref     timerfindall
display      get          isequal     openvar     startat     timer       vertcat
```

如果把 MATLAB timer 类用 UML 图表示出来，效果如图1.6所示。

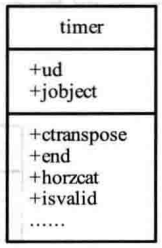


图 1.6 MATLAB 的 timer 类图

1. 如何把面条抽象成 Class

下面介绍如何定义一个 MATLAB 类。以面馆的面条为例，先要抽象出面条的特点，比

① “公有”的概念在第2.9节解释。

如种类（type）：有宽和细；状态（state），为熟和不熟，这些特点都是通过类的属性来表示的；还要抽象出面条相关的操作，通过方法来表示，比如煮面条 boil()。

下面给出一个面条类在 MATLAB 中简单的定义，具体语法可以参见第2.1节。

Noodle.m

```
classdef Noodle < handle
    properties
        type                    %  面条的种类
        state                   %  面条的状态
    end
    methods
        function boil(obj)      %  煮为成员方法
           obj.state = 'done' ;
        end
    end
end
```

目前，可以暂时忽略面向对象编程的语法，只需要知道：

- 上述定义中 Noodle 是类的名称;type 和 state 是类的 property，也可以叫做成员变量。
- MATLAB 要求每个类的定义保存为一个同名的文件，比如这段代码要保存为名字叫做 Noodle.m 的文件。

完成面条类的定义之后，下一步就是要使用这个“模板”，建立面条的实体对象 (object)。在 MATLAB 中，建立实体对象的方式是：调用类的构造函数 (Constructor)。在这里，暂时只需要知道构造函数是一种特殊的函数：

- 构造函数和类同名，比如类名叫做 Noodle，那么该类的构造函数也叫做 Noodle。
- 构造函数的返回值是构造出来的新的对象。

在脚本或者命令行环境中，调用构造函数产生对象的语句如下：

Script

```
noodle = Noodle();      %  调用构造函数 创建面条 obj
```

现在要煮面条，就调用和面条对象相关的操作boil()

Script

```
noodle.boil() ;         % 调用成员方法 boil,noodle 内部状态改变，面条被煮好了
```

注意，调用 boil 方法之后，并没有返回任何新的面条对象，面条被煮熟反映在面条对象内部状态的改变上。然后，还可以定义一个 Soup 类（这里从略），用来创建汤的实体对象，利用 Soup 类的 mix 成员方法，把煮好的面条放到汤中去，得到汤面。MATLAB 程序看上去是这样的：

Script

```
soup = Soup() ;         % 创建汤的 obj
soup.mix(noodle)        % 面条对象作为汤对象成员方法的输入
```

2. 文件类

再举一个对象和类的例子。我们知道，任何工程科学计算中都少不了数据的输入和输

出，其实这些数据文件也可以抽象成一个类。如果概括各种数据文件的基本特征，也就是文件类（FileClass）的属性，应该至少有文件名、文件格式、文件路径、文件句柄，当然最重要的还有数据内容。对文件的基本操作，或者说文件类的基本方法应该至少包括打开、阅读、写、关闭，用 UML 表示如图 1.7 所示。

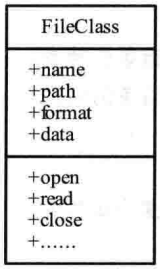


图 1.7 文件类 UML 图

该文件类在 MATLAB 中可以这样定义（具体内容之后再填充）

FileClass

```
classdef FileClass < handle
        properties                                          % 类的属性
          name
          path
          format
          data
       fID
    end
    methods
       function obj = FileClass(name,path)                  % 构造函数用来初始化文件类
         obj.name = name;
         obj.path =  path ;
         obj.open();
         obj.read();
       end
       function open(obj)                                   % open 成员方法负责打开文件
          fullpath=strcat(obj.path,filesep,obj.name) ;
          obj.fID = fopen(fullpath);
       end
       function read(obj)                                   % read 成员方法
          obj.data = textscan(obj.fID,'%s %s  %s');         % 假定数据文件中有两列数据
       end
       function delete(obj)
            fclose(obj.fID);
            disp('file closed');
        end
    end
end
```

相较于面向过程的编程方式，面向对象的编程方式把数据 (文件) 和对数据的操作 (打开、读取、关闭) 封装于一个类中。于是在命令行下，一个面向过程的数据读取变成了对文件对象 fileobj 的操作。在脚本中，可以声明文件类对象如下：

Script

```
>> fileobj = FileClass(filename,path);
```

上述定义中，open 和 read 函数会在类的构造函数中被调用，所以声明出文件对象之后，fileobj 中就自动地含有文件中的数据了。上述的定义中，所有的属性和操作都是 public 的，所以可以直接访问文件的数据属性，就像操作结构体一样：

Script

```
a  = fileobj.data ;   % 访问成员属性 data，并赋值给外部变量 a
```

关闭文件的操作放在一个叫做 delete 的函数中，其作用是当不再需要 fileobj 对象时，可以调用该函数，关闭文件句柄。

1.3 面向过程编程有哪些局限性

这节继续讨论面馆的例子。如果要解决的问题很简单，比如只是做一碗清汤面，面向过程和面向对象两种编程方式的优劣并不明显。但是，当要解决的问题变得逐渐复杂起来时，将会看到面向过程这种编程方式所暴露出来的问题逐渐增多。

1. 封装面条制作过程

现在需要继续完善面馆的运营。首先，老板把面馆分成前台和厨房。顾客只关心点菜吃面，不需要知晓面条的制作过程，所以只需要给前台提供一个函数，叫做 Order()，模拟服务员为顾客点菜，并且通知厨房做菜的过程，该函数的返回值是做好的汤面（noodlesoup）：

点菜函数：Order.m

```
function noodlesoup = Order()
    dough         = prepareDough() ;                    % 把面粉和成面团
    noodle        = prepareNoodle(dough) ;              % 把面团擀成面条
    boildedNoodle = boilNoodle(noodle) ;                % 把面条煮熟
    soup          = prepareSoup() ;                     % 准备面汤
    noodlesoup    = mix(boildedNoodle,soup) ;           % 把面条倒入汤中
end
```

2. 面馆开张了，目前只提供一种清汤面

面馆开张了，前台的 MATLAB 脚本可以写成：

Script

```
clear classes;                          % 清除工作空间中旧的类的定义
noodlesoup = Order();
```

如果面馆老板仅满足于开一个小店面，只提供一种面条，那么上述的程序足够了。但是，现实当然要比这个复杂得多，因此须一步一步地考虑更多的情况。

3. 顾客点了挂面和刀削面

现在来了两个顾客，一个要吃刀削面，一个要吃挂面。我们给刀削面起个名字叫做

chunk，给挂面起个名字叫做 regular，老板于是修改 Order 函数，增加一个参数，以反映顾客点了什么菜：

Script

```
clear classes;                          % 清除工作空间中旧的类的定义
noodlesoup1 = Order('chunk');           % 第一个顾客点刀削面
noodlesoup2 = Order('regular');         % 第二个顾客点挂面
```

不同的面条，做法当然也不同。于是用 switch 语句来处理不同的情况。还假设做刀削面和清汤面用的面团 dough 都是一样的，但是拉面和刀削面的制作方式是不一样的，于是再增加两个新的方法：

- prepareChunkNoodle 方法，返回的是削出来的面块。
- prepareRegularNoodle 方法，返回的是挂面。

因为煮不同的面条所用的时间也不一样：刀削面厚，要多煮一会儿，所以还需要给 boiledNoodle 增加一个新的参数，指定煮的时间长短：

新点菜函数：Order.m

```
function product = inStoreOrder(type)
    dough = prepareDough() ;                                    % 把面粉和成面团
    switch type                                                 % 刀削面和挂面做不同准备
        case 'chunk'
            noodle = prepareChunkNoodle(dough) ;                % 削面
            boilednoodle = boilNoodle(noodle,'longer') ;        % 煮的时间长一点
        case 'regular'
            noodle = prepareRegularNoodle(dough) ;              % 拉面
            boilednoodle = boilNoodle(noodle,'regular') ;       % 煮的时间短一点
    end
    soup = prepareSoup() ;                                      % 准备面汤
    product = mixSoupAndNoodle(boilednoodle,soup) ;             % 把面条倒入汤中
end
```

相应地，煮面的函数 boilNoodle 也需要修改，加入 switch 语句，处理煮面条时间的不同：

煮面函数：boilNoodle.m

```
function product = boilNoodle(doughnoodle,type)
    switch type
        case 'longer'       % 煮的时间长些
        case 'regular'      % 煮正常的时间
        end
end
```

4. 把炒面加到菜单上

老板忙得满头大汗，两碗面条终于做好了，这时又来了一个顾客，他点的既不是挂面，也不是刀削面，而是炒面，于是点菜函数 Order() 又要修改了。给 switch 再加一个选项，还要添加一个炒面条的函数 fryNoodle()。还要注意，炒面要先煮个半熟再下锅炒，还不用配汤。图 1.8 所示为做不同面条的步骤。

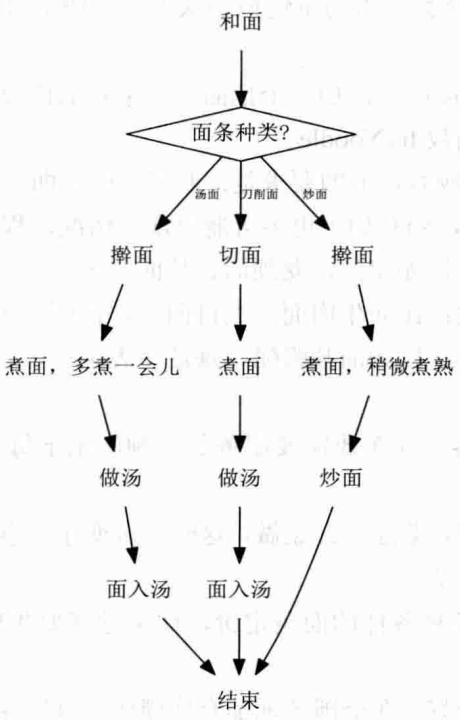


图 1.8 面向过程的做面条流程

根据图 1.8 所示的流程修改的程序代码，简单给 switch 语句中再添加一个 fry 的情况如下：

新的点菜函数：Order.m

```
function product = Order(type)
    dough = prepareDough() ;        % 和面
    switch type
      case 'chunk'  % 刀削面
          noodle = prepareChunkNoodle(dough) ;                    % 切面
          boilednoodle = boilNoodle(noodle,'longer') ;            % 多煮一会
          soup = prepareSoup() ;                                  % 做汤
          product = mixSoupAndNoodle(boilednoodle,soup) ;         % 把面条放入汤
      case 'regular' % 汤面
          noodle = prepareRegularNoodle(dough) ;                  % 擀挂面
          boilednoodle = boilNoodle(noodle,'regular') ;           % 煮的时间短一些
          soup = prepareSoup() ;                                  % 做汤
          product = mixSoupAndNoodle(boilednoodle,soup) ;         % 把面条放入汤
      case 'fry'      % 炒面
          noodle = prepareRegularNoodle(dough) ;                  % 擀挂面
          boilednoodle = boilNoodle(noodle,'short') ;             % 煮个半熟
          product = fryNoodle(boilednoodle);                      % 下锅炒
    end
end
```

现在看上去这个 Order 函数中做的事情似乎太多了，而这只是要修改的函数之一，老板还需要：

- 修改煮面函数 boilNoodle，添加一个把面煮得半熟的情况，用来做炒面。
- 还要添加新的炒面函数 fryNoodle。

不一会儿，又来了一个顾客，他的要求是：麻辣牛肉拉面。老板顿时忙不过来了。目前的程序离实际情况还差很远，面对如下更多可能出现的情况，程序又该如何修改和扩展？

- 面条的品种有很多，比如宽面、龙须面、拉面……
- 面条的主料数不胜数：比如牛肉面、大排面、炸酱面……
- 面条还要加各种佐料：比如油盐酱醋、辣油、香菜……

再考虑更复杂的情况:

- 如果同时有很多顾客，面馆还需要服务员按顺序记下每个顾客所点的品种，再去告诉厨房的师傅。
- 如果一个顾客点了菜，菜还没开始做，这时顾客变了主意，要修改点菜的内容，程序该如何应付这种情况？
- 该如何给菜单上的各种各样的面条定价，如果顾客要牛肉面再多加一份牛肉，该如何计算价格？
- 再假设面馆的生意很好，在全国各地都有连锁店，但是各地的风味和菜单都不同，如何修改程序让各地的连锁店都能自主经营等。

如果使用面向过程的方式，单单每增加一个品种，就要修改 Order 函数，还要添加其他新的函数，更复杂的情况会使程序迅速膨胀起来。各个函数之间重复的部分越来越多，虽然可以使用复制和粘贴。如果编写的是一个永远不需要修改的程序，或者只需剪剪贴贴还好。但更实际的情况是还没修改完毕，新的需求又来了，不得不再找到所有要修改的地方，并且把每个地方修改得一致。现在已经可以看出面向过程的困难：开一个面馆的因素很多、很难一开始就能把所有的可能性考虑到，把所有的因果关系分析清楚，更本质的因素是面馆的需求（Requirements）不是设计初期就固定的。实际过程中，面馆可以是不断扩张的。于是不得不在增加新函数和修改旧函数之间顾此失彼了，犯错误的几率也就越来越高。

从这里可以认识到面向过程的方式并不是解决这类复杂问题的最优方法。简而言之，如何让程序容易维护和扩张，并且新添加的代码不影响已经写好的程序，就是面向对象编程所能够帮助解决的问题。顺便指出，把已有程序修改成面向对象的风格，并不意味着要重写所有的代码。大多数情况下，如果已经有了面向过程的程序，可以用面向对象的思想去包装这些已有的程序，并且在此基础上继续维护和扩张已有的程序。

1.4 面向对象编程有哪些优点

1. OOP 把大问题分解成小的对象

面向对象的编程方法把一个复杂的大问题，比如经营一个面馆，分解成各个小问题 (模块)。读者可能会问，面向过程的方法也要把大问题化成小问题，那么两者到底有什么不同呢？可以这样理解，面向过程的小问题的单位是函数，数据在函数之间交互；而面向对象的

小问题的单位是模块，模块不但拥有数据还拥有方法，模块和模块之间通过组合和交互来解决问题，这样更贴近真实世界。比如经营一个面馆，用面向对象的方法，大致可分解成顾客模块、服务员模块、面条模块、主料模块、调料模块等，如图 1.9 所示。

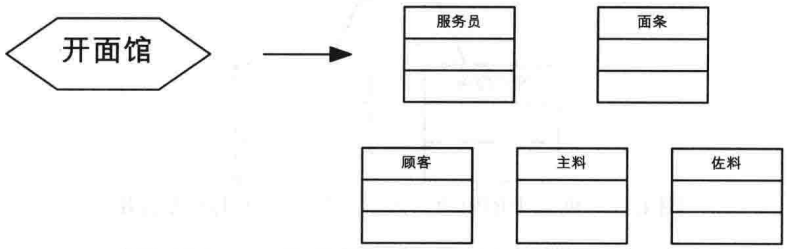


图 1.9 模块化的面馆

2. OOP 通过组合和信息传递完成任务

OOP 通过各个对象之间的组合和信息的传递完成任务。比如服务员接受顾客的点菜单，面条 + 主料 + 佐料，构成菜肴，然后由服务员端给顾客。顾客只需要和服务员交流，不需要知道厨房中是怎么生产面条的，主料和佐料是怎样加入汤面中去的。对象之间不仅可以相互交流，还可以组合形成新的对象。比如面条、主料和调料对象组合在一起，就可以变成一碗汤面，如图1.10所示。

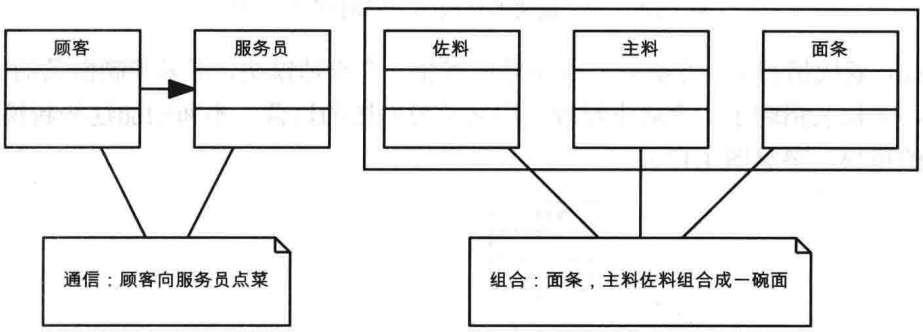


图 1.10 面馆中的对象通过通信和组合运营面馆

3. OOP 通过继承达到代码的复用

如果两个类的定义之间有明显的共同之处，面向对象可以通过继承来达到代码复用，相同的代码就不用写两次，并且修改时也只要修改一个地方即可。比如面馆的顾客来自五湖四海，有的喜欢吃宽边的挂面，有的喜欢吃窄边的挂面，无论宽边窄边，都是挂面，因此可以把挂面的基本特征抽象出来，定义成一个挂面基类，宽边和窄边模块可以复用这个挂面基类，它们就不需要在重复定义挂面的这些普遍特征了，如图1.11所示。

4. OOP 修改或者添加模块不会影响到其他模块

面向对象的编程方式对修改封闭，对扩展开放[①]。这就是说，一个好的面向对象的设计，添加新的模块和类不会影响到已经写好的程序。比如，老板决定增加菜单上的品种，修改主料类不会影响到面条和佐料类，如图1.12所示。

①Software entities should be open for extension, but closed for modification. Bertrand Meyer (1988). Object-Oriented Software Construction.

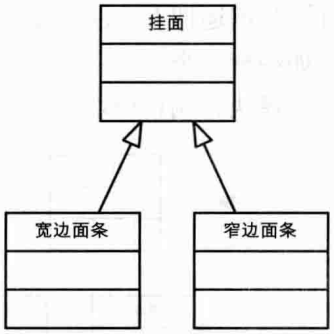


图 1.11 面馆中的模块通过继承达到代码的重复使用

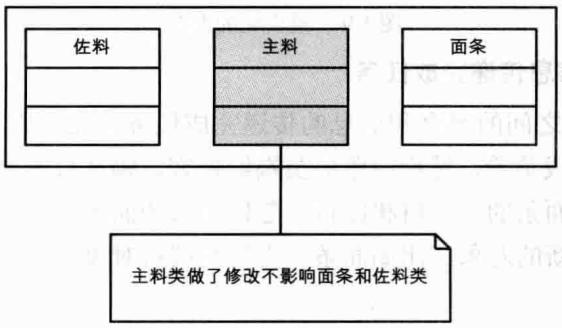


图 1.12 一个模块的修改不会影响到其他模块

再比如，老板招聘了一个厨师，程序里要增加一个厨师模块，用来控制面条的制作过程。过了几天，老板又招聘了一个店堂经理，用来统筹面馆的运营。不断增加这些新模块不会影响到已有的模块，参见图 1.13。

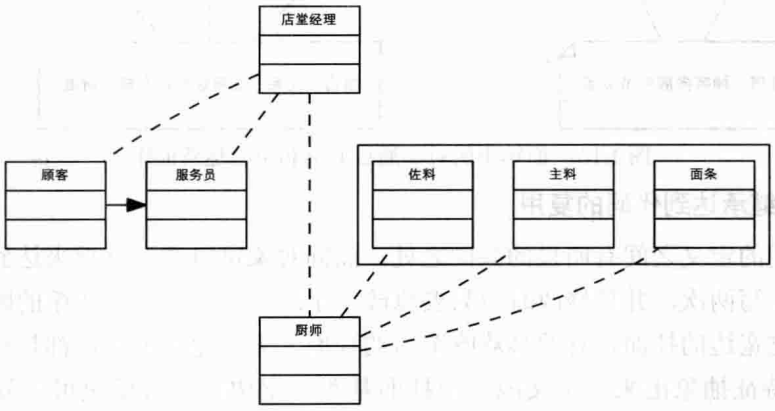


图 1.13 给面馆添加新的模块是简单的

第 2 章　MATLAB 面向对象程序入门

2.1　如何定义一个类

如前所述，MATLAB 中类和对象的概念无处不在，MATLAB 中的任何变量都属于一个类，就连最基本的数据类型 double，char 都不例外。在命令行下，可以通过简单的命令 whos 来检查变量所属的类：

Command Line

```
>> a = 7 ;
>> b = 'some string' ;
>> c = rand(4,4);
>> whos
  Name        Size              Bytes  Class     Attributes

  a           1x1                   8  double
  b           1x11                 22  char
  c           4x4                 128  double
```

从结果看，a 变量属于 double 类；c 变量是矩阵，也属于 double 类；b 是字符串，属于 char 类。

2008a 以后的 MATLAB 开始向用户提供新的面向对象的编程方法[①]，用户可以在 MATLAB 中定义自己的类。在入门篇中我们先给出最简单的定义类的语法：

ClassName.m

```
classdef  Point2D  < handle
    properties  % 属性 block 开始
        %......
    end         % 属性 block 结束

    methods     % 方法 block 开始
        %......
    end         % 方法 block 结束
end
```

说明：

- 任何 MATLAB Class 的定义都是由关键词 classdef 开始，end 结束。
- classdef 后面紧跟类的名字，在这里是 Point2D。
- 类名后面有一个 < handle，会在第3章中具体解释，现在先规定，所有的类的定义后面都要加上 < handle，并且本书中的绝大多数类都是这样定义的。
- 一个类定义中包含属性 block 和方法 block。

① “新”指的是支持用 classdef 来定义 MATLAB 类的方法，旧版本 MATLAB 支持其他定义类的方法，但是没有公开的文档记录。

下面是 Point2D 二维点类的定义，用来演示定义属性和方法的语法。

Point2D.m

```
classdef Point2D < handle
    properties
       x
       y
    end
    methods
        function obj = Point2D(x0,y0)     %  Point2D 类的构造函数
            obj.x = x0;
            obj.y = y0;
        end
        function  normalize(obj)          %  Point2D 坐标的归一化方法
            r = sqrt(obj.x^2 + obj.y^2);
            obj.x = obj.x/r;
            obj.y = obj.y/r;
        end
    end
end
```

- 因为这个类用来表示二维坐标轴上的点，所以 property block 中首先定义了该类的两个成员属性，分别是 x 和 y 的坐标。
- method block 中定义了两个方法：第一个是 Constructor(构造方法)，负责产生并且返回该 Point2D 的对象，这是由用户显式定义的类的构造函数；第二个方法叫做 normalize，负责把 x 和 y 的长度归一化。

上述成员方法 normalize 中的第一个参数是 obj，用来把对象当做参数传入 normalize 方法中，从接受参数的方法上来说，类的成员方法和普通的函数没有太大的不同。

问题： 学习本书时该使用哪个版本的 MATLAB

回答：用 classdef 的方式来定义类是 MATLAB 2008a 之后才支持的功能，如果读者使用的是早于 2008a 的版本，请至少升级 MATLAB 到 2008a 版本。虽然 2008a 中包括本书介绍的大部分面向对象的功能，但是还是建议读者尽量升级到最新的 MATLAB 版本。Mathworks 每半年发布一个新的 MATLAB 版本，几乎每个版本都会根据用户的反馈和新的需求，在保证兼容的前提下，引入更多新的面向对象的功能，更重要的是，新版本面向对象的性能总是不断地在加速和提高。

2.2 如何创建一个对象

method block 中有一个和 class 同名的方法，叫做 Constructor（构造函数，或者构造方法）。Constructor 是一个特殊的方法，它负责创建类的对象，通常它还可以用来初始化对象的属性，即给属性赋初值。

创建对象的方式是直接调用类的 Constructor。比如下面的 Script 创建出了两个对象 p1 和 p2，并且初始化了对象 p1 和 p2 的属性。

Script

```
p1 = Point2D(1.0,1.0) ;
p2 = Point2D(2.0,2.5) ;
```

乍看上去这和一般的函数调用相似，但这里的区别是：Point2D 不是一般的函数，而是一个类的 Constructor，并且返回值是一个对象。图2.1是 Point2D 类及其两个对象 p1 和 p2 的 UML 图，其中右上有折角的方框在 UML 中代表注释。

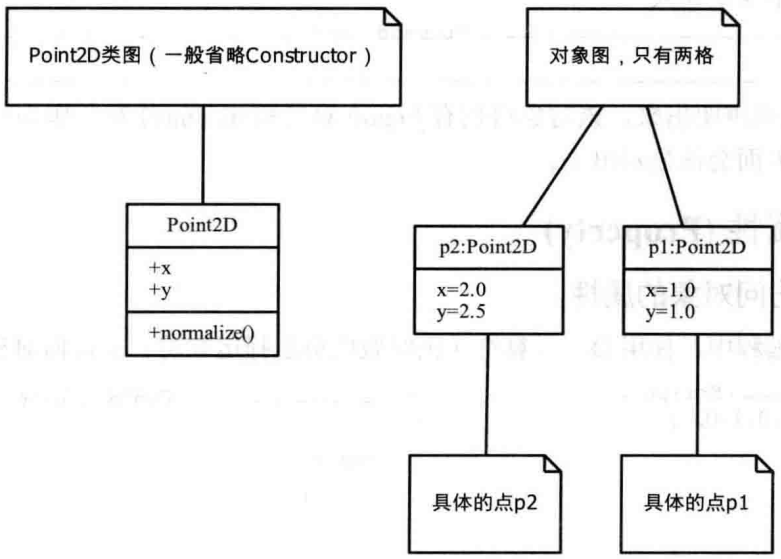


图 2.1 Point2D 类和对象的 UML

MATLAB 是弱类型检查语言，用上述方法定义的属性对类型没有限制，所以属性可以是 double 标量，也可以是 double 矩阵，甚至可以是 GUI 对象。比如下面我们定义一个如图2.2所示的简单的视图类 View，其作用是在 Figure 窗口中画一个文本编辑框，其草图如图2.3所示。

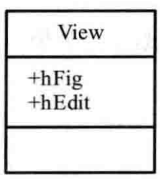


图 2.2 View 类 UML

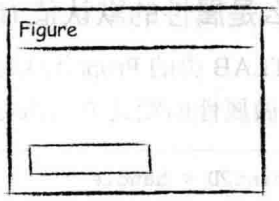


图 2.3 简单视图草图

该 View 类可以这样设计，类中一个属性是 Figure 对象，另一个属性是 uicontrol 的文本编辑框，在构造函数中，文本编辑框的 Parent 设置成 Figure 对象：

View.m

```
classdef View < handle
    properties
        hFig
        hEdit
    end
```

```
    methods
        function obj = View()
            % 两个属性都是对象
            obj.hFig  = figure();
            obj.hEdit = uicontrol('style','edit','parent',obj.hFig);
        end
    end
end
```

在命令行中如果输入

Command Line

```
>> obj= View() ;
```

View 类的对象被声明出来，该对象将拥有 Figure 对象和 uicontrol 对象作为其属性，一个带文本编辑框的界面会被显示出来。

2.3 类的属性 (Property)

2.3.1 如何访问对象的属性

面向对象编程中，使用 Dot 运算符（也叫做成员选择运算符）来访问对象的属性。

Script

```
p1 = Point2D(1.0,1.0) ;
p1.x

p1.x = 10 ;
p1.x
```

Command Line

```
ans =
     1.0

ans =
     10
```

这种普通属性的访问和赋值，其使用方法和结构体类似。在工程科学计算中，还可以声明一些具有特殊性质的属性，这就是下面几节要介绍的内容。

2.3.2 什么是属性的默认值 (Default Value)

在 MATLAB 类的 Property Block 定义中，可以为属性直接赋给一个值。通过这种方法提供的值，叫做属性的默认值（Default Value）：

Point2D.m

```
classdef Point2D < handle
    properties    % 提供属性 x 和 y 的默认值
       x = 0.0 ;
       y = 0.0 ;
    end
end
```

属性的默认值的设置还支持 MATLAB 表达式（Expression），MATLAB 会在构造对象时，自动计算这个表达式，并且给成员变量赋上对应的值，比如：

Point2D.m

```
classdef Point2D < handle
    properties
       x = cos(pi/12) ;
       y = sin(pi/12) ;
    end
end
```

注意：如果使用表达式，该表达式仅在类定义被 MATLAB 装载时执行一次。如果赋给属性的默认值是一个表达式，那么这个表达式计算的结果最好是固定的。比如在如下的 Record 类中，如果把 clock 当做默认值赋给属性 timeStamp，就很可能达不到想要的要求，因为 timeStamp 在类 Record 被装载时就设置好了，并且不再更新，而类的本意，则是要记录对象被创建的时间。类定义的装载时刻如图 2.4 所示。

Record

```
classdef Record < handle
    properties
          timeStamp = date
    end
end
```

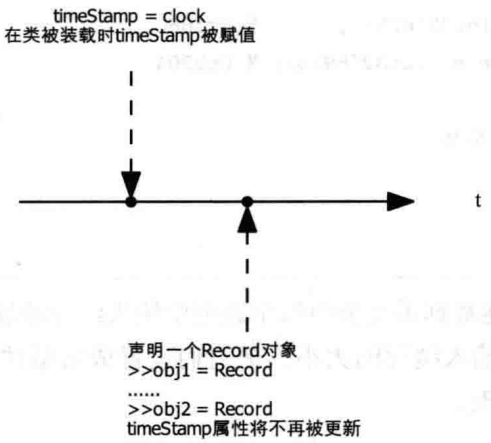


图 2.4 类定义的装载时刻在声明对象之前

除了可以在 Property Block 中给属性赋默认值，还可以在 Constructor 构造函数对成员变量做初始化，见第2.5.2节。

在定义类的 Property 时，还可以给这些 Property 指定一些特定的特性 (Attribute)，让它们具有特殊的性质。下面介绍几种常用的特性。

2.3.3 什么是常量 (Constant) 属性

常量 (Constant) 属性，就是在对象生存周期中值保持不变的属性。无论在类内部或者外部对该属性进行修改都将报错。定义 Constant property 需要使用 Constant 关键词。例如下面的代码中，第 3 行的 Constant 值必须要在类的定义体内指定：

```
classdef A < handle
     properties(Constant)
        R = pi/180
    end
end
```

如果不显式 (Explicitly) 地给定被声明成常量的属性一个特定的值，那么默认的 Constant 值是 empty double 。Constant property 的另一个用处是存储/封装一些常用的常量，以便在程序中可以不用创建出一个对象就直接使用类中的常量。比如，查询 A 类中常量成员 R：

Command Line

```
>> A.R    % 这里 A 是类名 而不是对象名
```

Constant 特性通常用来修饰那些工程计算中规定的常量、硬件指标。比如想用 MATLAB 来控制某个数据采集的硬件，并且该硬件厂商已经提供了 C 语言的 API，那么可以使用 MATLAB 的类来包装这个硬件的驱动，提供给其他 MATLAB 用户使用，而其中一些硬件指标，可以定义成 Constant 属性，用户通过该 MATLAB 类对象控制实际的数据采集硬件。

```
1  classdef DataAcqusitionHardware < handle
2      properties (Constant)
3          ......
4          Input_BufSize =  int32(6252);       % 0x186C
5          Input_OnbrdBufSize =  int32(8970); % 0x230A
6          ......
7          % 常量用来代表硬件指标
8       end
9   ......
10 end
```

其中第 4 行的属性描述数据采集卡中每个通道中输入缓冲能够放置的样本数量，第 5 行的属性描述采集卡上每个输入缓存的大小。把它们设置成常量的目的是表示在计算工作中（和硬件通信中）不允许修改。

2.3.4 什么是非独立 (Dependent) 属性

现实中对象存在这样的属性：其值依赖于其他的属性，一旦其他属性改变，该属性也做相应的变化，在概念上也可以理解成数学中的因变量。比如二维坐标中的点 $p(x,y)$ 到原点的距离 r 可以表示成

$$r = \sqrt{x^2 + y^2}$$

该 r 值因为依赖于 x 和 y，所以是非独立的 (Dependent)。给 r 赋值，最简单的方法，可以在构造函数中设定其初始值，比如：

Point2D.m

```
classdef Point2D < handle
    properties
```

```
        x
        y
        r
    end
    methods
        function obj = Point2D(x0,y0)
            obj.x = x0;
            obj.y = y0;
            obj.r = sqrt(obj.x^2 + obj.y^2);    %   在构造函数中赋值
        end
    end
end
```

但是这种做法的问题是：如果对象的 x 或者 y 值改变了，该属性 r 必须被重新计算，所以还要提供一个更新 r 值的方法。但是这样做是不方便的，因为每次检测到 x 和 y 值的改变，都要调用一次该更新方法。

解决方法是，在 MATLAB 类中，可以把这种 r 声明成 Dependent(依赖) 属性。

Dependent 属性的特点是：对象内部没有给该属性分配物理的存储空间，每次该属性被访问时，其值将被动态地计算出来。而计算该属性的方法由一个 get 方法提供，语法如下：

Point2D.m

```
1  classdef Point2D < handle
2      properties
3          x
4          y
5      end
6      properties(Dependent)
7          r
8      end
9      methods
10         function obj = Point2D(x0,y0)
11             obj.x = x0;
12             obj.y = y0;
13         end
14         function r = get.r(obj)          %  Dependent 属性的计算公式要放在 get 方法中
15             r = sqrt(obj.x^2 + obj.y^2);
16             disp('get.r called') ;
17         end
18     end
19 end
```

说明：

□ 第 7 行，成员变量 r 被放在了一个新的属性 block 当中，使用了关键词 Dependent。

□ 第 14 行，get 方法提供了动态计算出 obj.r 的公式，get 方法的使用将在第 2.8 节详细

讨论。

□ Dependent 属性在被需要时，可立即进行计算，在类内部没有占用实际的存储空间，当使用 save 命令时，Dependent 的属性不会被保存到 MAT 文件中去。

简而言之，可以随时访问对象 r 的最新值，比如：

Command Line

```
>> p1 = Point2D(1.0,1.0);
>> p1.r
get.r called
ans =
    1.4142
>> p1.x = 2.0;    % 此处修改 p1.x 的值
>> p1.r           % 检查 p1.r 的值
get.r called      % get.r 方法确实被调用 该方法中更新 r 值
ans =
    2.2361        % 发现 r 也自动作出了相应的改动
```

当在命令行中输入 p1.r 时，MATLAB 解释器 (Interpreter) 检查 classdef 中 r 的定义，发现其是 Dependent 属性，且 classdef 中提供了计算方法，于是解释器在内部调用了成员方法 get.r 即时计算 r 的值。方法 get.r 被调用，这可以从命令行中 get.r called 输出语句中看出。

读者也许会问，既然需要一个能够随时更新的 r 值，为什么不设计一个普通方法，比方叫做 calcR，来取代 get.r 方法，每次调用这个方法都可以得到最新的 r 值，比如：

CalcR.m

```
function r = calcR(obj)
   r = sqrt(obj.x^2 + obj.y^2);
end
```

原因是这样的，把 r 声明成 Dependent 属性还有一个好处：支持 dot 和向量化操作。如果 r 是一个矢量或者矩阵，在类的外部，可以对 r 直接进行矢量操作。比如：

```
>> obj.r(1:2)
```

如果 r 是一个结构体，在类的外部，可以使用 dot 继续访问 r 内部的其它 fields：

```
>> obj.r.otherfields
```

而普通方法 calcR 无法提供这样的便利。

下面我们再举一个 Dependent 属性的例子。接着前面 GUI 的例子，如果要在视图类代码中添加一个属性用来记录 GUI 在文本编辑框中的输入，该属性就可以设计成 Dependent 属性，因为它的值依赖于 uicontrol 的文本编辑控件中 string 的值。图2.5所示为 View 类中的 text 属性。

类的定义如下：

View.m

```
classdef View < handle
    properties
        hFig
```

```
        hEdit
    end
    properties(Dependent)
        text
    end
    methods
        function obj = View()
            obj.hFig = figure();
            obj.hEdit =  uicontrol('style','edit','parent',obj.hFig);
        end
        function str = get.text(obj)
            str = get(obj.hEdit,'String');
        end
    end
end
```

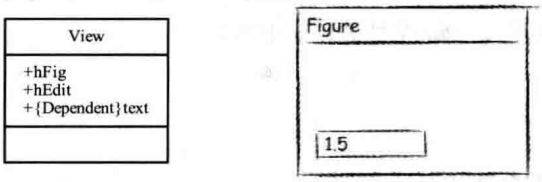


图 2.5 View 类中的 text 属性是一个典型的 Dependent 属性

这样每次对视图对象中的 text 属性进行查询时，该属性就会自动去查询文本编辑框中的用户输入了：

Command Line

```
>> obj = View();
>> obj.text
```

问题：什么是 MATLAB 的解释器 (Interpreter)

回答：MATLAB 解释器将 MATLAB 命令行、脚本和函数中的 MATLAB 的代码翻译成内部指令，并且执行。MATLAB 作为一种解释型语言，有很多好处，从写代码到执行代码的转换是立即完成的，并且源代码总是存在，所以一旦出现错误，MATLAB 解释器很容易就能指出错误的位置。它具有良好的交互性，适用于快速的程序开发。

2.3.5 什么是隐藏 (Hidden) 属性

隐藏 (Hidden) 的效果是在命令行中查看对象的信息时，该属性不会被显示出来。例如：

Command Line

```
>> obj = A()
obj =
  A with no properties.
  Methods
```

在类定义中可以使用关键词 Hidden 把成员属性定义成隐藏，例如：

A.m

```
classdef A < handle
    properties(Hidden)
        var
    end
end
```

MATLAB 中 Hidden 属性的默认值是 False，所以，如果不显式地把属性声明成 Hidden，该属性就是非隐藏的，并且 `properties(Hidden)` 和 `properties(Hidden=true)` 的声明效果是一样的，我们推荐第一种方式，因为这样代码更加简洁。

如果用户知道该属性的名字，仍然可以正常地访问该属性

Command Line

```
>> obj.var = 10
obj =
  A handle with no properties.
  Methods, Events, Superclasses
```

用户也可以通过这种方式隐藏成员方法，比如：

A.m

```
classdef A < handle
    methods(Hidden)
        function internalFunc(obj)
            disp('I am a hidden function');
        end
    end
end
```

Hidden 关键词的用处是隐藏类的内部细节。如果用户定义的类中有很多属性和方法，那么可以使用 Hidden 关键词指定那些不期望显示的属性和方法，如果在命令行中键入对象，则只显示最重要的内容。如果用户自己构造一个 MATLAB 类，并且提供给别的 MATLAB 用户使用，可以考虑使用 Hidden 关键词隐藏类内部的细节[①]。

2.4 类的方法 (Method)

2.4.1 如何定义类的方法

类的方法 (Method) 一般用来查询（Query）对象的状态，或向对象发出一个命令 (Command)，比如操作对象中的数据。在 MATLAB 面向对象编程中，类方法的定义要放在 method block 中，和一般函数定义类似，方法的定义由 function 关键词开始，end 关键词结束：

```
  ......
  methods
      function [returnValue] = functionName(arguments)
```

①即使不是出于保护代码的目的，隐藏类内部的细节也是一个好的习惯，提供给外部使用者的应该始终是一个固定的接口，内部的细节可以由类的作者随意地升级修改，从而做到向后兼容。

```
        ......
    end
end
......
```

使用 Point2D 类的例子，除了构造函数，我们再添加一个 normalize 方法，用来归一化成员属性 x 和 y：

Point2D.m

```
classdef Point2D < handle
    properties
        x
        y
    end
    methods
        function obj = Point2D(x0,y0)              % Point2D 的构造函数
            obj.x = x0;
            obj.y = y0;
        end
        function  normalize(obj)                   % 归一化成员方法 注意这里没有返回值
            mag   = sqrt(obj.x^2 + obj.y^2);       % 得到该点到原点 (0,0) 的距离
            obj.x = obj.x / mag ;                  % 归一化操作成员变量 x
            obj.y = obj.y / mag ;                  % 归一化操作成员变量 y
        end
    end
end
```

如果成员方法只有几行，可以把方法的实现 (Implementation) 放在类定义中，如果成员方法代码的行数比较多，还可以在类定义中仅给出该函数的声明，而把实现放到一个独立的文件中去。比如：

Point2D.m

```
classdef Point2D < handle
    properties
        x
        y
    end
    methods
        function obj = Point2D(x0,y0)
            obj.x = x0;
            obj.y = y0;
        end
        normalize(obj) ; 仅提供一个声明
    end
end
```

normalize.m

```
% 类方法作为一个独立的文件
function normalize(obj)
    mag = sqrt(obj.x^2 + obj.y^2);
    obj.x = obj.x /mag;
    obj.y = obj.y /mag;
end
```

说明：

- 关于如何组织类的定义、类的方法，具体可以参见第5章。
- 并不是所有的方法都可以在类定义中仅提供一个声明，而在另一个独立的方法文件中提供实现细节，比如类的 Constructor 和 Destructor(析构函数)，以及 static 方法(9.2节) 都必须在类的定义中实现。

2.4.2 如何调用类的方法

MATLAB 中，有两种调用对象成员方法的格式，并且这两个调用格式基本是等价的，如图2.6所示。

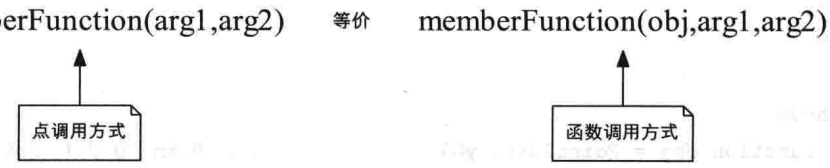


图 2.6 点调用方式和函数调用方式是等价的

注意，如果想要使用点调用方式，则 obj 必须作为成员方法的参数之一，并且通常作为第一变量。换句话说，在图2.6中的例子中，成员方法的定义要有 3 个输入变量（obj, arg1, arg2），只是第一个变量 obj 可以前置而已。

注意，在 MATLAB 面向对象编程中，obj 并不是一个 MATLAB 保留的关键词，使用 obj 只是为了说明传递给成员方法的该参数是一个对象，其他常用的表示对象的名字还包括 h,H,self，它们和 obj 一样，仅仅是一个代号，不是关键词。在这里，我们推荐尽量使用 obj 当做对象参数的名称，以便和 C++ 的 this 作出区分。[①]

1. 使用 OOP 的点 (Dot) 语法调用成员方法

大部分面向对象编程语言都使用“对象 +●+ 函数”的方式，即 obj.memberFunction() 来调用成员方法，例如：

Command Line

```
>> p1 = Point2D(1.0,1.0);          % 声明一个 Point2D 对象 p1
>> p1.normalize();                 % 调用成员方法 normalize
>> p1.x
ans =
    0.7071
>> p1.y
ans =
    0.7071
```

上述对象 p1 的 x 和 y 的初值都是 1，执行了归一化操作之后，x 和 y 的值都改变了。从语义上来说，还可以把调用成员方法，理解成编程者向对象发出了一条消息或者指令。面向对象编程，从这个角度来看，也可以被理解成一句话：向对象发送消息。比如上述的程序，我们向点对象 p1 发出了归一化的指令。如图2.7所示。

①区分的原因是 C++ 中的 this 参数是隐式 (Implicit) 的，而 MATLAB 面向对象编程中，该 obj 参数始终需要显式地包含在参数列表中。

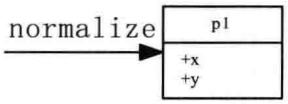


图 2.7 调用成员方法可以被理解成向对象发出指令

2. 使用传统的函数语法调用成员方法

MATLAB 还支持传统的函数调用方法，以保证其兼容性，对象被当做函数的一个参数传入：

Command Line

```
>> p1 = Point2D(1.0,1.0);
>> normalize(p1);                    % 对象当做函数的参数
```

2.4.3 点调用和函数式调用类方法的区别

虽然两种调用方式基本等价，但仍有细微的区别。

- 使用p1.normalize() 格式符合面向对象的风格，程序的可读性高，一目了然，这是施加在对象上的一种操作。使用normalize(obj) 方式满足传统的面向过程式的编程习惯，在某些情况下，兼容旧的 MATLAB 代码。
- 使用 Dot 语法可以清楚地告诉 MATLAB 要调用的是成员方法还是函数。作者建议 MATLAB 面向对象编程中，坚持使用 Dot 的方法调用对象的成员方法。并且

```
obj.memberFunction(arg1,arg2)
```

的调用方式，明确告诉 Dispatcher 要调用的是 obj 所属类的方法，如果是函数式调用，Dispatcher 还要做其他的工作才能判断到底要调用哪个方法。

- 如果使用 Dot 语法，MATLAB 的语法检查器 (Code Analyzer) 会及时帮用户检查语法错误，如果使用普通函数调用方法，Code Analyzer 没有办法区分用户到底是在调用函数还是在调用成员方法[①]，如果用户恰好重载了 built-in 函数，即使调用的语法正确，mlint 也会给出 warning，劝告用户不要重载内置函数。

2.4.4 什么是方法的签名

1. 为什么 obj 要作为方法的一个参数

如果回顾一下类和对象的 UML 表示方法，可以注意到在类的图表中有成员方法，而在对象的图中没有成员方法，其中的原因是：成员方法并不属于某一个特殊的对象，成员方法是被所有具体对象所共有的，如图 2.8 所示。

比如，Point2D 成员方法normalize()，被所有对象所共有：

Point2D.m

```
classdef Point2D < handle
......
    function   normalize(obj)
       mag = sqrt(obj.x^2 + obj.y^2);
       obj.x = obj.x / mag ;
```

①确定用户到底是在调用函数还是类方法，是 Dispatcher 的工作，并且唯有在执行时，Dispatcher 才会参与工作。

```
        obj.y = obj.y / mag ;
    end
......
end
```

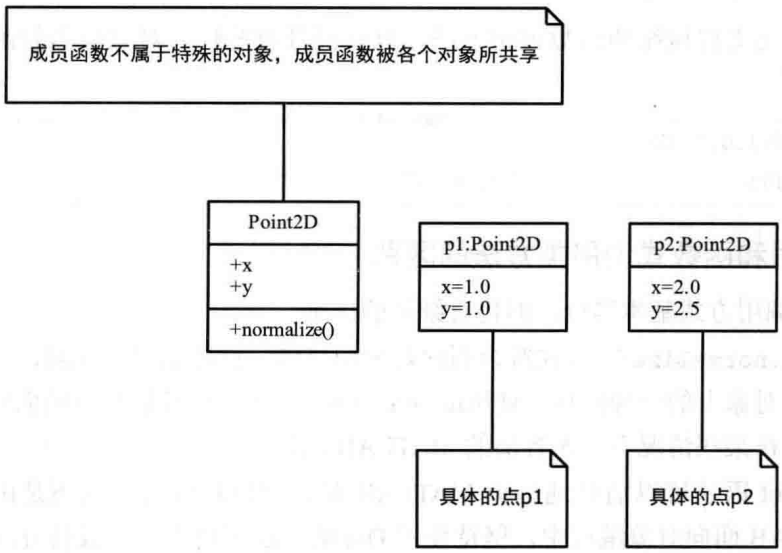


图 2.8 对象的 UML 框图中不包括成员方法

一个类可以有多个对象，每个对象都有自己的成员变量 x 和 y，彼此独立，比如：

Script

```
p1 = Point2D(1.0,1.0);
p2 = Point2D(1.5,1.0);
```

从实现的手段上来说，每个对象所能够调用的成员方法都是相同的，所以没有必要给每个对象都定义一个成员方法，相同类的对象是共享成员方法的。说得具体一点，即 p1 和 p2 调用的是类 Point2D 的成员方法，而不是对象 p1 或者 p2 的成员方法。之所以要把 obj 当做参数传给 normalize，就是因为要告诉成员方法 normalize，该归一化哪个对象的 x 和 y。

2. 什么是方法的签名 (Signature)

obj 必须作为方法的一个参数[①]还有一个原因，因为该对象连同方法的名称构成了该方法在 MATLAB 中独一无二的方法的 signature(签名)，如图 2.9 所示。MATLAB 的 Dispatcher 将利用这个签名去寻找匹配的函数，然后执行之。

再考虑这样一种情况，如果有两个对象 p1 和 p2，分别属于不同的类，并且每个类中都刚好有一个 normlize 方法，当不同的对象调用 normalize 方法时，MATALB 该如何找到匹配的 normalize 方法呢？再具体一点，我们定义了两个 Class，一个叫做 Point2D，一个叫做 Point3D。再声明一个 Point2D 的对象和一个 Point3D 的对象，都对它们调用 normalize 方法，MATLAB 该如何确定和 p1 和 p2 的匹配成员方法呢？比如：

①对象其实并不需要总是作为成员方法的第一个参数，但是从程序的可读性角度来说，最好还是把对象声明为成员方法的第一个参数。

Script

```
clear classes ;
p1 = Point2D(1.0,1.0)       ;
p2 = Point3D(1.0,1.0,1.0)   ;
normalize(p1)   ;
normalize(p2)   ;
```

Point2D.m

```
classdef Point2D < handle
    properties
       x
       y

    end
    methods
      function obj = Point2D(x0,y0)
        obj.x = x0;
        obj.y = y0;

      end
        %  做归一化的操作
        function   normalize(obj)
             disp('Point2D normalize');
             ......
        end
    end
end
```

Point3D.m

```
classdef Point3D < handle
    properties
       x
       y
       z
    end
    methods
     function obj = Point3D(x0,y0,z0)
           obj.x = x0;
           obj.y = y0;
           obj.z = z0;
       end
       %  做归一化的操作
       function   normalize(obj)
           disp('Point3D normalize');
           ......
       end
    end
end
```

MATLAB 是这样匹配方法的：在每次调用对象的方法时，MATLAB 的 Dispatcher 都会动态地判断该方法的签名 (signature)。所谓签名，就是每个类的方法在 MATLAB 内部的一个独一无二的名称。这种签名，通常由调用对象所属于的类的名字和方法的名称共同构成。所以，虽然从表面上看，两个属于不同类的对象都调用了 normalize 方法 (函数的名称相同)，但是 MATLAB 还是可以通过判断该函数的第一个实际参数的所属的类，即调用者所属于的类，来找到匹配该对象的方法。

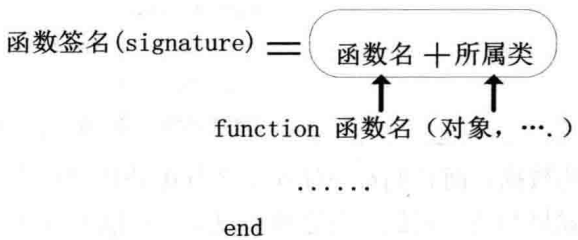


图 2.9 方法的 signature 由名称和对象的名称组成

输出的结果是：

Command Line

```
Point2D normalize called
Point3D normalize called
```

问题： **上述脚本中 clear classes 命令是必需的吗**

回答：是的，作为一个良好的编程习惯，在每个程序的开头使用 clear 清除残存的变量和旧的定义是必要的。特别是在类的定义被修改了之后。我们要清除内存中类的旧的定义，这样才能使新的修改生效。对 clear classes 的具体解释参见第2.10节。

2.4.5 类、对象、属性、方法之间的关系

OOP 介绍到这里，我们再解释一下类、对象、property 和 method 之间的从属关系。

首先，类是一个抽象的定义，类定义中含有 property 的定义，以及 method 的定义。类定义中既然包含这么多信息，类定义当然也是占用内存的[①]，内存中记录了比如 property 的名字、method 的名字等，其他和类有关的信息。

对象是一个物理实体，是根据 Class 模板创造出来的。在 MATLAB 中，对象也拥有物理内存，对象拥有属性，构成对象本身的也只有数据，任何成员方法不隶属于任何一个对象，成员方法和对象的关系就是绑定。

成员方法对应的是一种操作，类拥有该操作，对象可以调用所属类的成员方法，语法是`obj.method(arg)` 或者`method(obj,arg)`。该操作被该类所有对象所共享，所以同类对象具有一致的行为。

2.4.6 如何用 disp 方法定制对象的显示

我们知道，在 MATLAB 命令行下输入一个表达式，如果表达后不加；符号，MATLAB 将在命令行中显示这个表达式的值。对象的行为也是一样，如果在命令行中键入一个 MATLAB 对象，后面不加；符号，那么 MATLAB 会调用内置的 disp 函数来显示这个对象的基本信息，内置默认的 disp 函数对 MATLAB 对象的处理方法是：按照属性在类中的声明的顺序，按序把属性名称和其值打印出来，例如：

MyClass

```
classdef MyClass < handle

    properties
        prop1 = 'simple';
        prop2 = 'data';
    end

end
```

Command Line

```
>> obj = MyClass()
obj =
  MyClass handle

  Properties:
    prop1: 'simple'
    prop2: 'data'
  Methods,  Events,  Superclasses
```

如果类中包含复杂的数据，而我们希望能在命令行中获知对象内部的关键信息，使用这种按序打印属性值的方式显然是不足的。在这种情况下，可以重载类的 disp 方法来定制对象在命令行上的显示。比如在 MATLAB 数据采集工具箱中，声明一个 Session 对象，该对象可

①类的定义是如何存放在 MATLAB 内部的，请参考第 13 章。

用来控制 MATLAB 和硬件之间的通信[①]。

Command Line

```
>> mydaq = daq.createSession('ni');
```

然后添加两个 analog input 和两个 analog output 通道。

Command Line

```
>> mydaq.addAnalogOutputChannel('dev1', 'ao0', 'Voltage');
>> mydaq.addAnalogOutputChannel('dev1', 'ao1', 'Voltage');
>> mydaq.addAnalogInputChannel('Dev1', 'ai0', 'Voltage');
>> mydaq.addAnalogInputChannel('Dev1', 'ai1', 'Voltage');
```

这 4 个通道是该 Session 对象中最重要的内容，如果挨个的检查它们，在 MATLAB 的命令行下，得到的将是如下的输出：

Command Line

```
>> mydaq.Channels(1)
ans =
Data acquisition analog output voltage channel 'ao0' on device 'Dev1':
 TerminalConfig: SingleEnded
          Range: 0 to +5.0 Volts
           Name: ''
             ID: 'ao0'
         Device: [1x1 daq.ni.DeviceInfo]
MeasurementType: 'Voltage'
Properties, Methods, Events
>>
>> mydaq.Channels(3)
ans =
Data acquisition analog input voltage channel 'ai0' on device 'Dev1':
       Coupling: DC
 TerminalConfig: Differential
          Range: -20 to +20 Volts
           Name: ''
             ID: 'ai0'
         Device: [1x1 daq.ni.DeviceInfo]
MeasurementType: 'Voltage'
Properties, Methods, Events
```

该种标准 disp 行数太多，不够紧凑，所以数据采集工具箱中的该类重载了 disp 函数，使得用户只要在命令行键入对象名称，就可以用列表的形式显示 4 个通道的基本信息。

Command Line

```
>> mydaq
mydaq =
   Number of channels: 4
      index Type Device Channel   MeasurementType        Range       Name
      ----- ---- ------ ------- ------------------ ---------------- ----
```

①运行下面的代码需要 National Instruments 的数据采集卡，该例仅用来说明 disp 定制的用途。

```
      1     ao    Dev1    ao0     Voltage (SingleEnd) 0 to +5.0 Volts
      2     ao    Dev1    ao1     Voltage (SingleEnd) 0 to +5.0 Volts
      3     ai    Dev1    ai0     Voltage (Diff)      -20 to +20 Volts
      4     ai    Dev1    ai1     Voltage (Diff)      -20 to +20 Volts
Properties, Methods, Events
```

下面再举一个具体的例子来说明如何定制对象的输出 (display 方法)。比如下面的 MyStock 类，其中包含了一只股票符号、名称、收盘价等，如果使用内置的 disp 函数（这个例子中省去了从 Yahoo Finance 获取数据的过程）在命令行上得到的是一个简单的列表：

MyStock

```
classdef  MyStock < handle
    properties
        symbol
        name
        source
        exchange
        freq
        last_price
        last_date
        last_time
    end
    ...... % 其余略
end
```

Command Line

```
>> f = MyStock('f','5y','d') % 声明并显示对象
f =                                          <--
  MyStock handle                             <--
  Properties:                                <--
        symbol: 'F'
          name: 'Ford Motor Company'
        source: 'Yahoo Finance'
      exchange: 'NYSE'
          freq: 'd'
    last_price: 13.6800
     last_date: '1/25/2013'
     last_time: '4:03pm'
  Methods, Events, Superclasses              <--
```

同样，这个列表的缺点是：disp 函数默认打印出来的有些信息太基本，对用户来说不重要，比如被箭头标注的行：类的名称，类的 Header，最后一行的链接等。还有一些输出是对象的内部数据，不是什么关键信息，没有必要显示，比如资料的来源'Yahoo Finance'[①]。当然，还有一些重要的信息，比如时间、开盘价、收盘价，它们被分散在各自独立的行上，不够紧凑，我们希望最好能把这个重要信息放在一行，以表格的方式显示。

在 MATLAB 面向对象编程中，可以通过重新定义类的 disp 方法（也叫做覆盖 override）来定制对象在命令行上的输出内容，包含定制 disp 方法的类的对象，在命令行被用户查询时，MATLAB 会优先调用类中的 disp 方法。

下面我们重载这个 MyStock 类的 disp 方法，基本思路是先按照一定的格式构造要输出的字符串 (可以是包含换行符的多行字符串)，在该 disp 方法中，再调用内置的 disp 函数把这个字符串输出到命令行上去。

MyStock.m

```
......
  function disp(obj)
      s = sprintf('%-17s (%s:%s)\n',obj.name,obj.exchange,obj.symbol);
      s = [s,sprintf('-------------------------------------------\n')];
      s = [s,sprintf('Last Trade:          %6.2f',obj.last_price)];
      s = [s,sprintf('  (%s %s)\n',obj.last_time,obj.last_date)];
```

①可以使用第2.3.5节提到的方法，把这个属性设置成 Hidden。

```
      disp(s);
   end
......
```

效果如下（输出的结果变得更加紧凑了，该股票的收盘价、时间、日期都显示在了一行中）：

Command Line

```
>> f = MyStock('f','1y','d')
f =
Ford Motor Company (NYSE:F)
---------------------------------------------
Last Trade:          13.68  (4:03pm 1/25/2013)
```

2.5 类的构造函数 (Constructor)

2.5.1 什么是 Constructor

Constructor 是一种特殊的成员方法：

- Constructor 和类的名称相同，用来创造类的实例。
- MATLAB 类的定义中只能有一个 Constructor。
- Constructor 有且只能有一个返回值，且必须是新创建的对象，这是唯一创建一个新的对象的方式。

下面程序通过调用 Point2D 的 Constructor，并且提供其要求的参数，即 x 和 y 的初值，来获得一个二维点对象。

Point2D.m

```
classdef Point2D < handle
    properties
       x
       y
    end
    methods
        function obj = Point2D(x0,y0)     % 返回值必须是一个 obj
            obj.x = x0;
            obj.y = y0;
        end
    end
end
```

Command Line

```
>> clear classes;
>> p1 = Point2D(1.0,1.0);
```

2.5.2 如何在 Constructor 中给 property 赋值

在 Constructor 中也可以给对象的属性赋值。即使在 property block 中已经提供了默认值

的话，Constructor 中赋的新的值将取代 property block 中的默认值。

Point2D

```
classdef Point2D < handle
    properties
       x = cos(pi/12) ;      % Constructor 被调用之前,x 的默认值
       y = sin(pi/12) ;      % Constructor 被调用之前,y 的默认值
    end
    methods
        function obj = Point2D(x0,y0)
            obj.x = x0;      % 新的 x0,y0 将会取代 properties block 中的初值
            obj.y = y0;
        end
    end
end
```

MATLAB 在声明一个对象时，工作的顺序是：先装载类的定义，这时成员变量 x 和 y 的初始值将是 $\cos(\pi/12)$ 和 $\sin(\pi/12)$，然后再调用 Constructor，x 和 y 将在 Constructor 中被重新赋值。

2.5.3 如何让 Constructor 接受不同数目的参数

有些编程语言中，同一个函数可以有多种不同的定义和其对应的实现，调用函数时，编译器可以根据提供的参数的种类和个数，找到相匹配的函数。这叫做函数重载。MATLAB 是弱类型检查的解释性语言，不能通过参数数目的不同来决定调用哪个函数，类似的功能只能放到函数体中，通过判断参数的个数 (即 nargin) 来实现，根据 nargin 的不同，选择不同的代码。比如下面的 Point2D 类，Constructor 可以接受两个参数，也可以接受零个参数：

Point2D

```
1  classdef Point2D < handle
2      properties
3         x
4         y
5      end
6      methods
7          function obj = Point2D(x0,y0)
8              if  nargin == 0          % 如果没有提供参数
9                 obj.x = cos(pi/12) ;
10                obj.y = sin(pi/12) ;
11             elseif nargin == 2       % 如果提供了两个参数
12                 obj.x = x0;
13                 obj.y = y0;
14             end
15         end
16     end
17 end
```

我们可以不提供任何参数，调用 Constructor，得到一个对象。[①] 这种情况下，对象的属性 x 和 y 被初始化成cos(pi/12) 和sin(pi/12)。例如：

Script

```
p1 = Point2D()
```

Command Line

```
p1 =
  Point2D handle
  Properties:
    x: 0.965925826289068
    y: 0.258819045102521
  Methods, Events, Superclasses
```

也可以向 Constructor 提供两个初值，得到一个对象，其属性 x 和 y 的值是通过 Constructor 中第 13 行 elseif 的分支赋值的。

Script

```
p2 = Point2D(1,1)
```

Command Line

```
p2 =
  Point2D handle
  Properties:
    x: 1
    y: 1
  Methods, Events, Superclasses
```

2.5.4 什么是 Default Constructor

Default Constructor 一般性的定义是：不带任何参数的构造函数。由于 MATLAB 的函数可以设计为接受各种数目的参数，所以在 MATLAB 中，Default Constructor 可以定义成：即使不提供参数也可以产生对象的 Constructor。比如下面代码中的第 5~7 行：

Point2D.m

```
1  classdef Point2D < handle
2  ......
3      methods
4          function obj = Point2D(x0,y0)
5              if nargin == 0          %  这个部分其实就是 Default Constructor !
6                  obj.x = cos(pi/12) ;
7                  obj.y = sin(pi/12) ;
8              elseif nargin == 2      %  如果提供了两个参数
9                  obj.x = x0;
10                 obj.y = y0;
11             else                    %  对于其他提供参数的方式将报错
12                 error('wrong input arguments');
13             end
14         end
15     end
16 end
```

①在 MATLAB 中，如果不提供任何参数，对 Constructor 的调用除了可写成 p1 = Point2D()，还可以省略括号，写成 p1=Point2D。

上述的 Point2D Constructor 既可以接受两个参数，也可以接受零参数，都能够返回对象，所以也可以把 Default Constructor 理解成 Constructor 内部可以处理 nargin==0 的情况，并且返回新构建的对象的那部分代码。

2.5.5 用户一定要定义 Constructor 吗

对于这个问题，我们可以做一个简单的实验，下面是一个简单的 Simple 类的定义，只定义了一个属性，没有定义任何成员方法。

Simple.m

```
classdef Simple<handle
    properties
        x
    end
end
```

即使没有定义 Constructor，还是可以在命令行上使用 obj = Simple()。声明出一个对象来

Script

```
>> obj = Simple()  % 调用了 Constructor
```

Command Line

```
obj =                % 返回了一个 Simple 对象
  Simple handle
  Properties:
    x: []
  Methods, Events, Superclasses
```

没错，在命令行上的Simple() 命令，就是调用了 Simple 类的 Constructor，尽管用户没有显式地定义 Constructor。因为 MATLAB 规定，如果用户没有提供任何形式的 Constructor，MATLAB 会在内部给该类提供一个 Default Constructor。如果一定要把这个内部提供的 Default Constructor 在 MATLAB 中表示出来，那么它看上去将貌似是一个空函数。不过，其实 MATLAB 还是会在后台做一些基本的工作，比如给对象、属性分配内存空间等。

Simple.m

```
......
 function obj = Simple()
       % 内部什么内容都没有，但实际上 MATLAB 在后台给对象分配了内存空间并赋默认值
 end
......
```

上述的规定有两个含义：

第一，该自动提供的 Constructor 是 Default Constructor，不接受任何参数，如果尝试提供任何参数，MATLAB 将报错：参数过多。

Script

```
>> obj = Simple(1)
```

Command Line

```
Error using Simple
Too many input arguments.
```

第二，如果用户确实提供了 Consturctor，但是没有提供处理 nargin == 0 分支情况的代码，即该 Constructor 将不能接受零参数的情况，那么该用户定义的 Constructor 会抑制

MATLAB 在后台提供一个接受零参数的 Constructor。比如：

```
classdef Simple<handle
    properties
        x
        y
    end
    methods
        function obj = Simple(x0)  %  用户提供的 Constructor 接受一个参数
            obj.x = x0;
        end
    end
end
```

这时如果尝试使用obj = Simple()，错误将是可想而知的，一定是参数不够。

Script

```
>> obj = Simple()
```

Command Line

```
Error using Simple
Not enough input arguments.
```

也就是说，MATLAB 仅在特殊的情况下，即用户没有提供 Constructor 时，才会自动地提供这个 Default Constructor。

回到开始的问题上："用户一定要定义 Constructor 吗？"回答是：不一定，如果用户不定义，MATLAB 会帮用户提供一个最简单的"什么事情都不做的 Constructor"，如果这能够满足需要，那么用户可以不用定义 Constructor。

2.6 类的继承

2.6.1 什么是继承

继承是一种提供代码的重用方法，它是面向对象编程中最重要的概念之一。在入门篇中，我们旨在用短小的例子说明概念和语法，所以还是继续使用 Point 类的例子。假设已经有了一个二维点的类的定义，其中有一个print() 成员方法，把该类中的所有的属性输出到命令行。现在要定义一个三维的点类。当然，最直接的办法是另起炉灶，重头定义一个 Point3D 类，如图 2.10 所示。

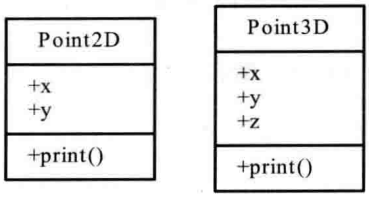


图 2.10 两个独立的类 Point2D 和 Point3D

UML 对应如下代码：

Point2D.m

```
classdef Point2D < handle
    properties
       x
       y

    end
    methods
        function obj = Point2D(x0,y0)
           obj.x = x0;
           obj.y = y0;

        end
        function print(obj)
           disp(['x =',num2str(obj.x)]);
           disp(['y =',num2str(obj.y)]);

        end
    end
end
```

Point3D.m

```
classdef Point3D < handle
    properties
       x
       y
       z
    end
    methods
        function obj = Point3D(x0,y0,z0)
           obj.x = x0 ;
           obj.y = y0 ;
           obj.z = z0 ;
        end
        function print(obj)
           disp(['x =',num2str(obj.x)]);
           disp(['y =',num2str(obj.y)]);
           disp(['z =',num2str(obj.z)]);
        end
    end
end
```

从代码角度来说，显然 Point3D 和 Point2D 类有许多相似之处：

- 成员变量相似。
- 构造函数相似。
- print 函数也相似。

从数学角度来说，三维空间中的点投影到二维空间，就是一个二维空间中的点，三维空间中的点只是多了一个 z 轴的坐标而已。

从功能角度来说，Point3D 类只是 Point2D 类的一个扩展。三维空间中的点是二维点的一种 (isa 的关系)。

面向对象中的“继承”提供这样一种机制，使得我们能够利用类和类之间“相似”的关系，利用已有的代码，在 Point2D 类的基础上定义出一个 Point3D 类。在 Point3D 类中，只需添加多出来的属性和方法就可以了。

Point3D.m

```
%   file name :
classdef Point3D < Point2D  % 用< 表示继承
    properties
       z
    end
    methods
        function obj = Point3D(x0,y0,z0)
           obj = obj@Point2D(x0,y0);
           obj.z = z0;
```

```
        end
        function print(obj)
            print@Point2D(obj);
            disp(['z =',num2str(obj.z)]);
        end
    end
end
```

在 MATLAB 中，可以这样让 Point3D 继承 Point2D 类：

继承关系也叫做泛化关系，被继承的类叫做父类或者基类 (Parent class 或 Base Class)；继承的类叫做子类或派生类（Child Class 或者 Derived Class）。[①]UML 图中规定，继承用空心箭头 △ 来表示，如图2.11所示。图中 Point3D 类继承自 Point2D 类。（代码中的 @ 表示调用父类，具体解释参考第2.6.2节）

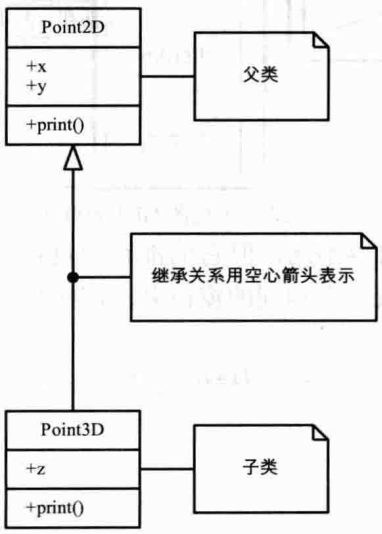


图 2.11 Point2D 作为 Point3D 的父类

在上述的例子中，使用“小于号”< 表示继承，即

```
classdef Point3D < Point2D
......
```

该行代码告诉 MATLAB 解释器：Point3D 类继承自 Point2D 类。

在 MATLAB 中，可以通过函数 isa 查询一个对象是否属于一个特定的类。比如：

Command Line

```
>> p2 = Point2D(1,1);
>> p3 = Point3D(1,1,1);
>> isa(p2,'Point2D')
ans =
     1
```

①Parent Child 关系不但常用来表示继承关系，还经常用来表示对象之间的所属关系，读者一定对 Handle Graphics 中的 Parent-Child 关系不陌生。

```
>> isa(p3,'Point2D')
ans =
     1
```

注意，因为三维点是二维点的一种，所以当查询isa(p3,'Point2D') 时，返回值也是true。

继承可以简单地用做对一类事物的总结。比如要编写一个 GUI 程序，该 GUI 包括多个视图，草图如图2.12所示。

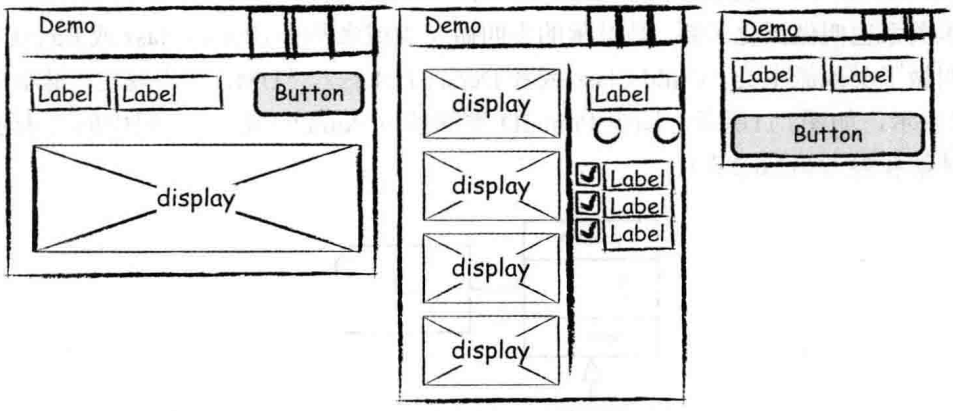


图 2.12 多视图 GUI 的草图

我们假设，View 的大小内容不同，但它们都有大致相同的外观。比如，不允许 resize，没有 menubar，没有 numbertitle，有相同的窗口名，这些共同的特征都可以总结到一个基类中去：

ViewBase.m

```
classdef ViewBase < handle
    properties
        hfig
        viewsize
        ID
    end
    methods
        function obj = ViewBase(viewsize,ID)
            obj.viewsize = viewsize;
            obj.ID = ID ;
            obj.hfig =  figure('pos',viewsize);
            set(obj.hfig,'resize','off',...     %  不许 resize
                'menubar','none',...            %  没有 menubar
                'numbertitle','off',...         %  title 中没有数字
                'name','Demo');                 %  窗口名相同
        end
    end
end
```

然后，各个子类 View 就可以继承这个 ViewBase，从而具有统一的风格。而视图上的控件对象，可以作为子类的属性。比如 ViewSmall 类，看上去可以是这样：

ViewBase.m

```
classdef ViewSmall < ViewBase
    properties
        ..... % 这里定义该视图上的控件
    end
    methods
        function obj = ViewSmall()
            % ViewSmall 类定制自己窗口的大小
            obj = obj@ViewBase([50,50,250,200],'Small');
        end
    end
end
```

2.6.2 为什么子类 Constructor 需要先调用父类 Constructor

为简单起见，我们先把 Point2D，Point3D 的代码简化如下：

Super.m

```
classdef Super      % 父类
    properties
        ......
    end
    methods
        function obj = Super()
            ......
            ......
        end
    end
end
```

Sub.m

```
classdef Sub < Super   % 子类
    properties
        ......
    end
    methods
        function obj = Sub()
            obj = obj@Super(); % 注意这里
            ......
        end
    end
end
```

我们注意到这样一行代码：`obj = obj@Super()` 。那么为什么子类的 Constructor 在做任何工作之前要先调用父类的 Constructor 呢？

这是因为在逻辑上，必须先有父才能有子，子类先继承了父类的属性和方法，然后才在父类的基础上增加了自己的属性和方法。所以在程序中，子类的 Constructor 先调用了父类的 Constructor。有时子类的属性的计算会依赖父类的属性，这也是父类 Constructor 要先被调用的原因之一。

代码中的@ 符号没有什么特别的含义（只是一个语法上的规定），告诉 MATLAB 解释器，在这里调用父类的 Constructor，返回一个对象。子类调用父类 Constructor 的示意图如图2.13所示。

从实用的角度来说，子类构造函数先调用父类的构造函数还可以理解成，在父类中先做一些基础的工作。比如，MATLAB 程序需要读入一些 XML 文件或文本文件作为计算的输入，则可以把读 XML 文件和文本文件的工作包装成类。总结两个类的共同点就是，不管要

读的文件的类型是什么，成员属性都要有文件名和文件的路径。所以，可以把这部分综合到一个基类 Reader 中去，如图2.14所示。

Reader.m

```
classdef Reader < handle
    properties
        filename
        path
    end
    methods
        function obj = Reader(filename,path)
            obj.filename = filename;
            obj.path = path;
        end
    end
end
```

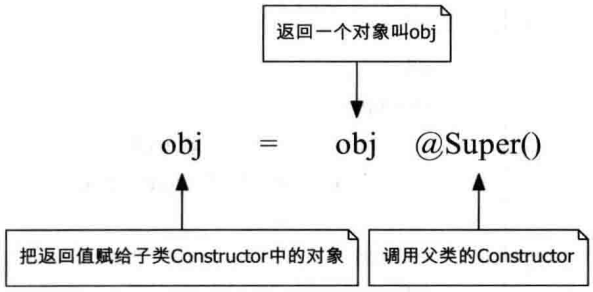


图 2.13 子类中调用父类 Constructor 的示意图

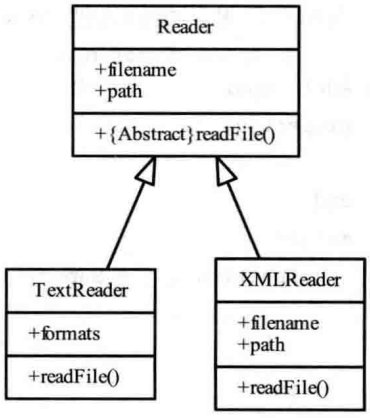


图 2.14 父类的作用是做一些基础工作

当然，两个子类都有一些属于自己的特殊的属性，比如，假设 TextReader 拥有一个属性叫做 formats，指明文本文件中的数据的存放方式，XMLReader 中有一个属性叫 tag，指明需要读的 XML 文件中的内容。声明一个子类的对象时，需要传递给构造函数三个参数：

Command Line

```
tReader = TextReader(name,path,formats);
xReader = XMLReader(name,path,tags);
```

在两个子类的构造函数的一开始，需要完成一些基础工作。这些基础工作就是，先调用父类 Reader 的构造函数，把 name 和 path 当做参数传递给父类的构造函数来初始化这两个属性。

TextReader

```
classdef TextReader < Reader
    properties
        formats
    end
```

```
    methods
        function obj = TextReader(filename,path,formats)
            obj = obj@Reader(filename,path) ; % 调用父类构造函数
            obj.formats = formats ;
        end
    end
end
```

XMLReader

```
classdef XMLReader < Reader
    properties
        tags
    end
    methods
        function obj = XMLReader(filename,path,tags)
            obj = obj@Reader(filename,path) ; % 调用父类构造函数
            obj.tags = tags ;
        end
    end
end
```

2.6.3 在子类方法中如何调用父类同名方法

子类和父类的成员方法可以有相同的名字，并且在子类的方法内部，可以调用父类的同名方法[①]。比如，在下面 Sub 类的 foo 方法中可以调用 Super 类的 foo 方法。这相当于在功能上，Sub 类的 foo 方法对 Super 类的 foo 方法进行了扩展。

Super.m

```
classdef Super
    properties
       ......
    end
    methods
        function foo(obj,argu)
            disp('Super foo called')
            ......
            ......
        end
    end
endc
```

Sub.m

```
classdef Sub < Super
    properties
        ......
    end
    methods
        function foo(obj,argu1,arug2)
           disp(' Sub foo called');
           foo@Super(obj,argu1);
           ......
        end
    end
end
```

图2.15示意在子类方法中调用父类同名方法的语法。@ 符号前面是父类方法的名称，后面是父类的类名。

①在子类中除同名方法外的其他地方不能调用父类的同名方法。

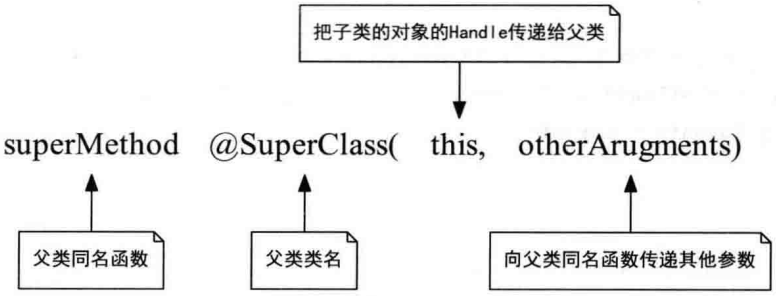


图 2.15 调用父类方法的语法

声明一个子类对象，并且调用 foo 方法，观察输出结果：

Command Line

```
>> aobj = Sub()
>> aobj.foo()
   Sub foo called
   Super foo called
```

子类的 foo 方法首先被调用，父类的 foo 方法接着被调用。

2.6.4 什么是多态

多态 (Polymorphism) 是建立在继承的基础之上的面向对象编程的精华之一。它指的是，同名的方法被不同的对象调用，能产生不同的行为 (形态)。沿用图2.11的例子，分别声明一个 Point2D 和 Point3D 对象。

Command Line

```
>> obj1 = Point2D()
>> obj1.print()
>>
>> obj2 = Point3D();
>> obj2.print()
```

```
x = 1
y = 1

x = 1
y = 1
z = 1
```

这里两个对象都调用了 print 方法，而输出的结果不同，此即调用同名方法而对象的行为不同。我们可以把上述的过程放到一个函数中去，如果 obj 函数的参数为：

someFunction.m

```
function someFunction(obj)
  if isa(obj,'Point2D')
        obj.print();
  end
end
```

该函数对参数的种类加以限制：要求是 Point2D 类，或者是其子类（Point3D），只要满足这个要求，print 就会被执行，但是语句obj.print() 具体调用哪一个 print 方法，将取决于运行时传入参数对象的种类。不同的 obj，对 print 指令作出的反应不同。

2.7 类之间的基本关系：继承、组合和聚集

面向对象的程序设计关键是对类的设计。设计一个单独的类是容易的，但是随着系统中类的数目的增多，设计的难点在于确定父类和子类，以及各个类之间的关系。本节先讨论类之间的几种基本关系：继承、组合和聚集。

2.7.1 如何判断 B 能否继承 A

□ 如果在逻辑上，B 是 A 的“一种”(a kind of，isa)，则允许 B 继承 A 的功能和属性。比如，Man 是 Human 的一种，Boy 是 Man 的一种，那么类 Man 可以从类 Human 中派生，类 Boy 可以从类 Man 中派生。

如下代码说明：A 是基类，B 继承了 A，那么 B 也就继承了 A 的成员变量和成员方法。

A.m

```
classdef A < handle
    properties
        a
    end
    methods
        function method1()
            ..
        end
        function method2()
            ..
        end
    end
end
```

B.m

```
classdef B < A
    properties
        b
    end
    methods
        function method3()
            ......
        end
    end
end
```

B 的对象将拥有属性 a，还可以调用 method1 和 method2。用来测试的脚本如下：

Script

```
clear ; clc ;
b = B();
b.method1();
b.method2();
b.method3();
```

这个简单的程序说明：使用“继承”可以提高程序的复用性。但是，如果反过来只为了增加程序的复用性而盲目使用继承，就会造成逻辑上的混乱和程序适用性降低。所以，我们要防止乱用“继承”。我们应当给“继承”立一些使用规则：

> 如果类A和类B不相关，不能为了使B的功能更多些而让B继承A的功能和属性。

2.7.2 企鹅和鸟之间是不是继承关系

公有继承和“是一个 (is a)”的等价关系，这个关系听起来简单，但在实际应用中“是一个”的关系不会总是那么显然。比如，有这样一个事实：企鹅是鸟；还有这样一个事实：鸟

会飞。如果想在 MATLAB 中表达这两个类直接的关系，我们也许会这样做：

Bird.m

```
classdef Bird < handle
    methods
        function fly(obj)      // 鸟会飞
            % ......
        end
    end
end
```

Penguin.m

```
classdef Penguin < Bird    // 企鹅是鸟

    ......



end
```

这时问题就出现了，因为这种继承关系意味着企鹅会飞，但我们知道这不是事实。造成这种情况的原因是语言不严密。说鸟会飞，并不是说所有的鸟会飞。通常，只有那些有飞行能力的鸟才会飞。如果更精确一点，我们都知道，实际上有多种不会飞的鸟，所以也许可以提供下面这样的层次结构作为解决方案 (这只是多种解决方法中的一种)，让企鹅的定义继承自不会飞的鸟类，如图2.16所示。

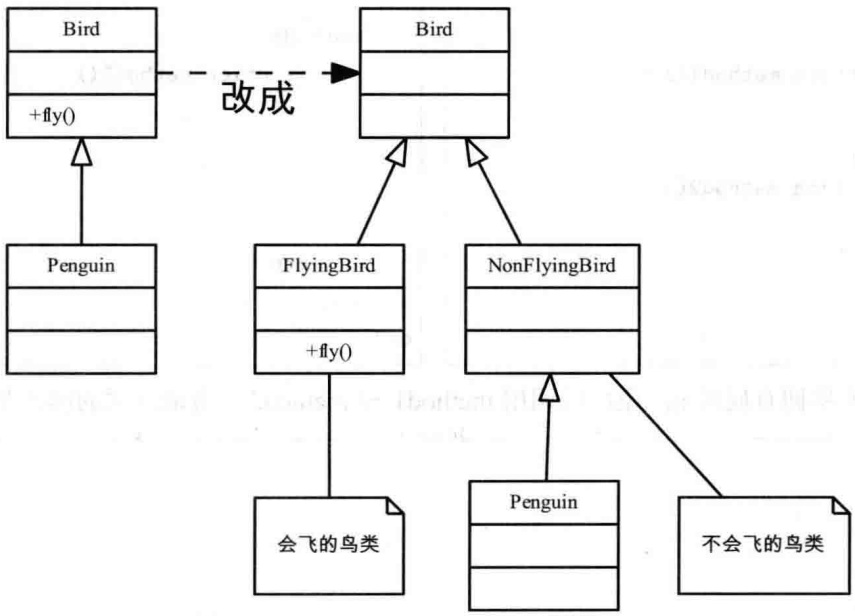


图 2.16 让企鹅类继承一个不会飞的鸟的类

企鹅和鸟这个例子是一个经典的例子，我们在第14.3节还要回顾这个例子，介绍其他更好的设计。

2.7.3 如何把类组合起来

如果在逻辑上 A 是 B 的“一部分”(a part of)，则不要为了让 B 得到 A 的功能去继承 A，而是要用 A 和其他东西组合出 B。比如眼 (Eye)、鼻 (Nose)、口 (Mouth)、耳 (Ear) 是头 (Head) 的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，而不是派生而成。

Nose

```
classdef Nose < handle
    % ......
end
```

Eye

```
classdef Eye < handle
    % ......
end
```

Mouth

```
classdef Mouth < handle
    % ......
end
```

Ear

```
classdef Ear  < handle
    % ......
end
```

注意，下面的 Head 的定义中，为简单起见，并没有对属性的类型加以限制，否则可以使用 isa 和 set 函数 (见第2.8节) 限制属性的类型。

Head.m

```
classdef Head < handle
    properties  % 头由眼鼻口耳组成
        eye     %   成员变量是对象
        nose
        mouth
        ear
    end

    methods
        function obj  = Head()
            obj.eye   = Eye()    ;
            obj.nose  = Nose()   ;
            obj.mouth = Mouth()  ;
            obj.ear   = Ear()    ;
        end
    end
end
```

相对而言，糟糕的设计方式是，不分青红皂白地用继承：让 Head 继承 Eye，Nose，Mouth 和 Ear。注意：下面的代码中，符号 & 表示多重继承[①]。

Head.m

```
classdef Head <  Eye & Nose & Mouth & Ear
     % ......
end
```

不使用继承的原因是：

- □ Head 和 Eye 之间，以及 Nose，Mouth，Ear 之间没有严格的 isa 关系，不能为让 Head 得到眼、鼻、口、耳的功能而使用继承。
- □ 组合比继承更适合这种类之间的关系。

①多重继承参见第8.2节。

在 UML 图上，我们规定用实心菱形箭头表示组合关系，如图2.17所示。

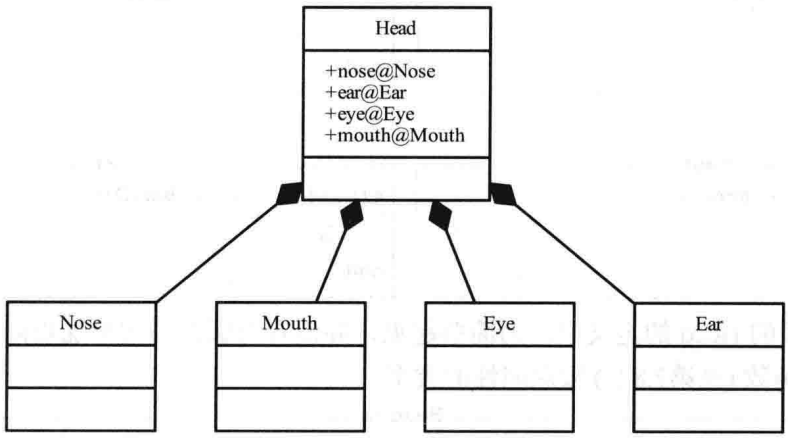


图 2.17 组合关系：Head 类由 Nose，Mouth，Eye 和 Ear 类组合完成

具体到 MATLAB 程序上，组合关系要求 Head 对象一定在内部拥有 Nose，Eye，Mouth，Ear 对象，这是通过 Head 对象的 Constructor 来保证的。

Head 类的 Constructor

```
function obj  = Head()
            obj.eye   = Eye()    ;
            obj.nose  = Nose()   ;
            obj.mouth = Mouth()  ;
            obj.ear   = Ear()    ;
end
```

这将保证在 Head 对象被创建时，其余 4 个对象也同时被创建。

在第2.2节提到的简单视图类中也是一个组合关系，视图类对象拥有 Figure 对象和 uicontrol 对象，为了保证 Figure 和 uicontrol 是视图类的必要部分，Figure 对象和编辑框对象的声明也放在了视图类的 Constructor 中：

View 类的 Constructor

```
function obj = View()
    obj.hFig  = figure();
    obj.hEdit = uicontrol('style','edit','parent',obj.hFig);
end
```

2.7.4 什么是组合聚集关系

1. 组合关系 Composition

先回顾一下组合关系：组合是一种强烈依赖的整体和部分关系。比如第 2.7.3 节的例子中，头一定由耳、鼻、口、耳组成。如图 2.18 所示，鸟不能没有翅膀、头和脚。实心的菱形箭头表示由翅膀、头和脚组合成一只健全的鸟。注意，线段上可以用数字表示数量对应关系。这里，一只鸟对应两支翅膀、两只脚。

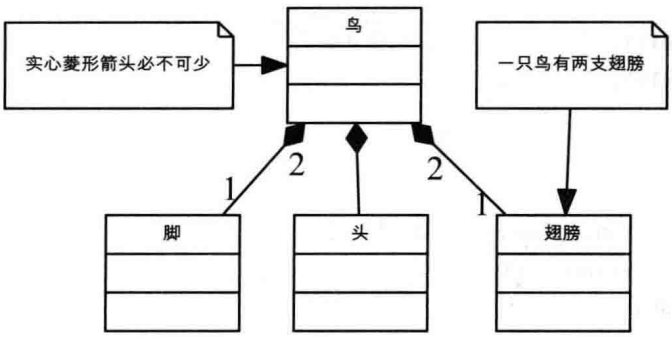


图 2.18 ♦表示必不可少：鸟 Class 必须要有翅膀、头和脚

2. 聚集关系 Aggregation

还有一种类似的关系叫做聚集。聚集是一种松散的整体和部分关系。如图2.19所示，自行车由架子、轮子和坐垫等部分组成。但是整体和部分并不是强烈的依赖，即使整体不存在了，比如自行车的架子不存在了，部分仍然可以存在。在 UML 中规定，聚集关系用空心菱形箭头符号 ◇ 表示。注意，表示聚集关系的线段上也可以加上数字，表示数量对应关系。这里，线段上的 1 和 2 表示一辆自行车有两个轮子。

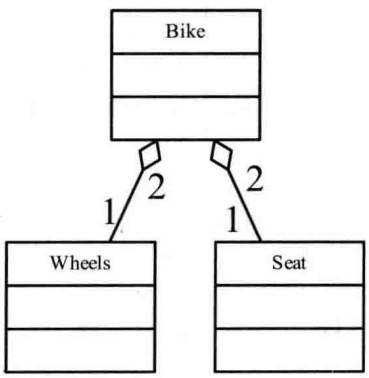


图 2.19 ◊表示松散的组合：坐垫、轮子不存在了，自行车还可以存在

聚集反映在程序上就是，Wheel 和 Seat 的对象可以独立创建（而不需要在 Bike 的 Constructor 中被创建）。创建完之后，再装到自行车架上去。

Bike.m

```
classdef Bike < handle
    properties
        frontWheel
        rearWheel
        seat
    end
end
```

例如：

Script

```
frontWheelObj = Wheel();              %  独立存在
rearwheelObj  = wheel();
seatObj  = seat();
......
bikeObj = Bike();
bikeObj.frontWheel = frontWheelObj;   % 聚集关系
bikeObj.rearWheel  = rareWheelObj ;
bikeObj.seat = seatObj;
```

这里省略了 Wheel 类和 Seat 类的定义，注意下段程序中，Wheel 和 Seat 类的对象可以单独被创建。

2.8 Handle 类的 set 和 get 方法

2.8.1 什么是 set 方法

set 方法为对象属性的赋值提供了一个中间层，例如下面的 A 代码部分。

A

```
classdef A < handle
    properties
        a
    end
    methods
      function  set.a(obj,val)
        if val >= 0
          obj.a = val ;
        else
          error('a must be positive');
        end
      end
    end
end
```

Command Line

```
>> obj = A();
>> obj.a = -10   % 试图给 a 赋负值
Error using A/set.a (line 10)
a must be positive
```

该类的定义中，提供了一个属性的 set 方法时，在 A 的外部，任何对属性 a 的赋值都将经过这个 set.a 的中间层（由 MATLAB 负责调用）。属性的 set 方法通常用来检测赋值是否符合要求。比如规定对象的某属性的值来自 GUI 界面上用户的输入，在把用户的输入直接赋给该属性之前，要检查该输入值（包括数据类型、阈值等）是否符合要求。那么这个检查输入的工作，可以交给属性的 set 方法。下面的例子中，A 类定义了一个属性 a 的 set 方法。在 set 方法中规定，属性 a 必须是正值，否则报错。

检查赋值是否符合要求还可以包括检查赋值的类型。沿用图 2.19 的例子，假设 wheels 是 Bike 类的一个属性，如果要确保给该属性赋的值是 Wheels 类对象，那么可以把 Bike 的 set 方法设计成如下形式：

```
classdef Bike < handle
    properties
        wheels
    end
    methods
        function set.wheels(obj,val)
            if( isa(val,'Wheels') ) % 检查输入 val 的类型
                obj.wheels = val;
            else
                error('input must be an instance of Wheels')
            end
        end
    end
end
```

set 方法还有一种常见的用法，就是当需要在设置该属性的值时，同时做其他一些工作。比如需要一个 LOG 来记录每次属性被赋的值，这样的工作就可以放到 set 方法中去。如果没有 set 方法，就要在每一个给该属性赋值的地方重复写记录的代码，这样的编程方式不利于程序的扩展和修改。

在下列情况下，set 方法不会被调用：

- 在 set 方法内部，对属性的赋值不会再次调用自身的 set 方法，比如上述的 set.a 方法中，`obj.a = 10` 不会再触发 set.a 方法。
- 在复制对象时，不会触发该属性的 set 方法。
- 在 property block 中给属性赋默认值时，不会触发该属性的 set 方法。但这不代表默认值可以无视 set 方法中的对数据有效性验证 (如果有的话) 。用户应该自行确保默认值能够通过该 set 方法验证，做到类定义的一致性。

但是，在 load 一个对象时，属性 set 方法会被 MATLAB 调用，如果这时对象的某些属性的值仍然是默认值时，这些默认值会经过 set 方法，被验证有效性。读者可以把 set 方法想象成数据有效性的最后一层保障。因为每个属性被 load 的顺序和在 MATLAB 中定义的顺序没有联系，所以在一个属性的 set 方法中，应该尽量不要访问类中的其他属性，因为这些属性可能还没有被 load，这叫做 Order Dependency。如果存在这样的属性，其值的设置依赖于其他属性的值，那么应该把这个属性声明成 Dependent[①]。Dependent 属性不会被保存到 MAT 文件中去，load 之后，需要使用该属性时，其值可以被即时计算出来。

2.8.2 什么是 get 方法

和 set 方法类似，get 方法提供对成员属性查询操作的一个中间层。如果定义了一个属性的 get 方法，则代码如下：

①见第2.3.4节。

A

```
classdef A < handle
    properties
        b = 10
    end
    methods
        function val =get.b(obj)
            val = obj.b;
            disp('getter called');
        end
    end
end
```

Command Line

```
>> obj = A();
>> obj.b
getter called

ans =

    10
```

在类的外部，对该属性的查询（query）都将经过这个中间方法。最常用的，给一个属性设置 get 方法的情况是，当这个属性是 Dependent 属性时，具体例子见第2.3.4节。这里再举一个 Dependent 属性和 get 方法一起使用的例子，针对规模较大的，需要长时间升级和维护的 MATLAB 程序。我们演示通过一个 get 的中间层，可以让程序变得向后兼容[①]。假设最初用户给 Record 类的定义为其中有一个属性 date 用来记录 Record 对象的时间，则在外部对类的使用为：

B

```
classdef Record < handle
    properties
        date
    end
end
```

Script

```
......
obj = B() ;
obj.date = date;
......
......
```

现在假设用户已经构造了大规模的面向对象的程序，而且这个 Record 类用处广泛，用户要修改升级 Record 类，比如要把类中的 date 的名字改成更有意义一点，比如叫做 timeStamp，最先想到的做法是，在所有的程序中搜索由 Record 定义出来的对象，找到之后把`obj.date`改成`obj.timeStamp`，但这样做是不可行的（读者可以想想为什么），而正确的做法是提供一个访问的中间层：

B 的新定义

```
classdef Record< handle
    properties(Dependent,Hidden)
        date
    end
    properties
        timeStamp                    % 新属性
    end
    methods
        function set.date(obj,val)
            obj.timeStamp = val;
```

①随着越来越多的用户互相共享 MATLAB 程序，每当用户已发布的程序需要更新时，向后兼容就显得尤为重要。

```
        end
        function val = get.date(obj)
            val = obj.timeStamp;        % 转而查询新属性
        end
    end
end
```

说明：

- 程序中的 set 和 get 方法给属性访问 date 提供了一个中间层，使得旧的程序能够不加修改，仍然可以使用新类的定义。set 和 get 与 Dependent 属性的组合用法给大型程序的升级之后的兼容提供了一个重要的手段。
- date 被设置成了 Dependent，这样不占内存；date 还被设置成了 Hidden，于是这个属性不会被命令行显示出来，新的类定义的使用者将觉察不到这个旧的属性。
- 在 set 和 get 中对 date 的查询和赋值请求，最后都转到了 timeStamp 上。

使用 set 和 get 方法是有时间成本的。在 MATLAB 中，调用成员方法的时间要大于直接访问属性的时间，并且 Set 和 Get 方法不会被 MATLAB 即时编译 (JIT) 加速。所以，应该仅在需要时，定义属性的 set 和 get 方法。还要避免如下没有任何附加价值的 set 和 get 方法。

没有任何价值的 set 和 get 方法

```
classdef A < handle
    properties
        var
    end
    methods
        function var = get.var(obj)
            var = obj.var;
        end
        function set.var(obj,var)
            obj.var = var;
        end
    end
end
```

2.9 如何设置属性和方法的访问权限

2.9.1 什么是 public，protected，private 权限

从面向过程到面向对象，最显著的一个区别就是，把数据和函数捆绑在了一起形成了类，数据变成了属性，函数 (Function) 变成了类的成员方法 (Method)。就数据而言，并不是所有被捆绑的数据都有必要提供给外部访问，有些数据可能是一些计算过程中的内部变量，外部程序并不需要知道这些细节，所以需要对访问的权限加以控制。从另一个角度就程序设计而言，为了尽量避免一个类中的某个行为干涉到同一系统中其他的类，应该让类仅公开必须让外界知道的内容，而隐藏其他一切不必要的内容，这也叫做封装。MATLAB 提供了关

键词Access = private,protected,public 来声明哪些属性和方法是可以公开访问的，受保护的，或者是不可以公开的，也就是私有的。例如：

SomeClass.m

```
classdef SomeClass < handle
    properties( Access = private )      % 类的私有属性
        prop_private
    end
    properties( Access = protected )    % 类的保护属性
        prop_protected
    end
    properties                          % 默认情况下是公有 (public) 的属性
        prop_public
    end

    methods( Access = 某种可访问性)     % 同上
        function result = someFunction(obj)
            ..... % 代码省略
        end
    end
end
```

下面以属性为例，比如定义一个 private 属性：

SomeClass.m

```
......
properties( Access = private )      % 类的私有属性
        prop_private
end
......
```

如果一个属性的特性被设置成Access = private，这表示只有该类的成员方法可以访问该数据，而子类和其他外部函数或者脚本无法访问到该成员变量。这也是类方法 (Method) 和普通函数 (Function) 的重要区别之一：类方法可以访问对象中的私有属性。[①]

SomeClass.m

```
......
 properties (Access = protected)
        prop_protected
 end
......
```

如果一个属性的特性被设置成Access = protected，这表示只有该类的成员方法，还有该类的子类可以访问该数据。

①函数 (Function) 和类方法 (Method) 的另一个重要区别是：类方法的输入参数中通常至少有一个是对象。

SomeClass.m

```
......
 properties
        prop_public
 end
......
```

成员变量的默认 Access 方式是 public，这表示不但在类的定义中，该类的成员方法以及该类的子类的成员方法，都可以访问到这个成员变量，而且在类之外的函数或者脚本也可以访问这个成员变量。

在 UML 中，表示成员属性和方法访问权限的方式是在属性和方法的名称前加一个修饰符号，如图2.20所示。

再以成员属性为例，通过示意图说明三种属性的被访问权限。

- □ + 表示 public，即该成员属性可以被其他对象、方法、函数所访问，如图2.21所示。public 应该尽量少用，共有意味着把类的成员变量和成员方法暴露给外部。这和面向对象的封装的原则是矛盾的。
- □ # 表示 protected，保护的数据和方法只对其内部方法和 subclass 可见，如图2.22所示。
- □ – 表示 private，私有的数据和方法只能被类内部的函数访问，如图2.23所示。但 Destructor 是个例外，见第8.1.5节。

图 2.20 成员属性和方法的访问权限

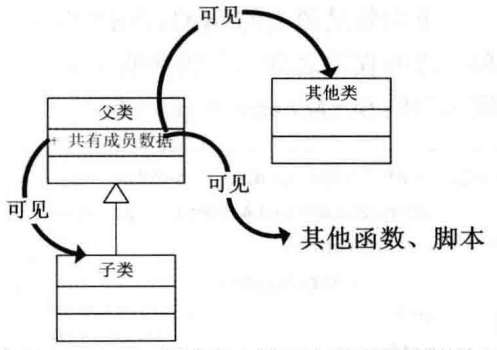


图 2.21 public 属性和方法可在任何地方被访问

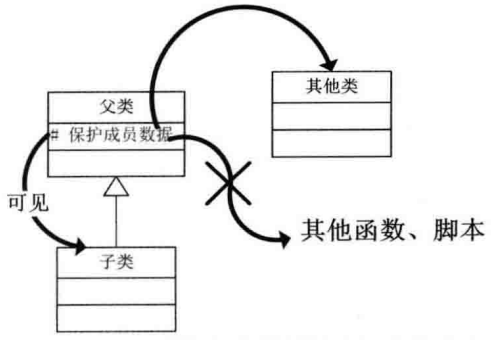


图 2.22 protected 属性和方法只能在子类中被访问到

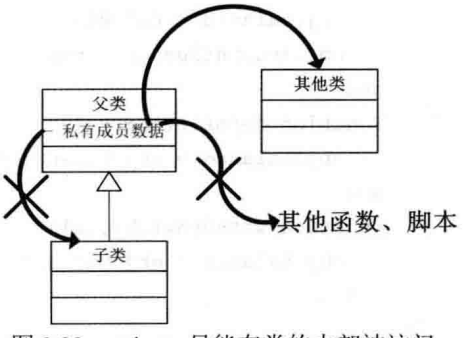


图 2.23 private 只能在类的内部被访问

对于属性的访问权限，还可以再细分成赋值 SetAccess 和查询 GetAccess 的权限，比如：

```
......
    properties (SetAccess = private, GetAccess = public)
            var
    end
......
```

说明该类属性可以被外界程序查询值，但不能被外部程序赋值，赋值只能在类的内部进行。

2.9.2 如何决定对类的属性和方法设置何种访问权限

下面举例说明如何决定对类的属性和方法，设置何种访问权限。假设要设计一个银行账户的类，该类的属性包括账户余额（balance）和账号（accountNumber），该类的方法包括取款（withdraw）和存款（deposit）。它们的访问权限可以设置如下：

- 账户余额应该设置成SetAccess = private，因为余额的变化只能通过存款和取款来实现，不能被外部随意赋值来改变，但是余额的值应该是可以被外部用户查询，所以GetAccess = public。
- 账号当然不能被外部随便改动，但是可以被外部查询，所以也是SetAccess = private和 GetAccess = public。
- 取款和存款两个类方法显然应该设置成 Access = public，因为这是该类提供给外部的两个功能，外部可以通过这两个方法，改变类的内部属性的值，即余额（balance）。

下面就是该类的 MATLAB 简略的实现。注意在类内部方法中，可以对属性进行任意访问，没有权限之分。在属性的定义中，因为 MATLAB 对所有访问权限的默认值都是 public，所以省略 GetAccess = public 。

BankAccount

```
classdef BankAccount < handle
    properties(SetAccess = private) % 默认 GetAccess = publicp
        balance
        accountNumber
    end
    methods
        function obj = BankAccount(balance,num)
           obj.balance = balance ;
           obj.accountNumber = num ;
        end
        function deposit(obj,val)
           obj.balance = obj.balance + val;
        end
        function withdraw(obj,val)
           obj.balance = obj.balance - val;
        end
    end
end
```

2.9.3 MATLAB 对属性访问的控制与 C++ 和 Java 有什么不同

MATLAB 对属性访问的控制和 C++ 与 Java 有些类似，但也有些独特之处。比如 public 属性，在 C++ 和 Java 中，如果把一个成员属性定义成 public，意味着该属性可以被外部直接访问和赋值，这相当于该属性直接暴露给了外部。MATLAB 不同的是，在外部直接访问和 public 属性之间，还有可能存在一个 set 和 get 的中间层，如果一个成员属性是 public 的，并且该属性定义了 set 和 get，那么对该属性的赋值则要经过 set 方法，对该属性的访问则要经过 get 方法，所以可以在这两个方法中，做一些必要的赋值的检查工作。所以说，即使是 public 属性，MATLAB 面向对象语言也提供了检查的措施。

再比如 private 属性，在 C++ 和 Java 中，如果把一个成员属性定义成 private，意味着该属性不可以被外部直接访问和赋值，但是习惯上，会提供一个 public 的 setter 和 getter 方法，将 C++ 或者 Java 的这个 setter 和 getter 用 MATLAB 的语言表示出来，例如下面的代码：

C++ 或 Java 风格的 setter getter

```
classdef A < handle
    properties(Access = private)
        a
    end
    methods
        function seta(obj,v)
            % 这里做一些输入的检查
            obj.a = v;
        end
        function v = geta(obj)
            v = obj.a
        end
    end
end
```

MATLAB 的 set 和 get 方法

```
classdef A < handle
    properties
        a
    end
    methods
        function set.a(obj,v)
            % 这里做一些输入的检查
            obj.a = v;
        end
        function v = get.a(obj)
            v = obj.a
        end
    end
end
```

两者是有区别的，左边的 seta 和 geta 是普通方法，不同于右边的 set.a 和 get.a。它们的访问和赋值方式不同：

Command Line

```
>> obj = A();
>> obj.seta(10);  % 显式地调用 seta 方法
>> obj.geta()
ans =
    10
```

Command Line

```
>> obj = A();
>> obj.a = 10;   % 隐式地调用 set.a 方法
>> obj.a
ans =
    10
```

使用 set 和 get 方法还便于程序的扩展，假设类最初的设计中没有定义 set 和 get 函数：

类最初的设计

```
classdef A < handle
    properties
        a
    end
end
```

类的外部使用

```
>> obj = A();
>> obj.a = 10;   % 直接访问属性
>> obj.a
ans =
    10
```

如果程序后期的开发过程中，意识到需要验证属性 a 的输入，只需要给类的定义中加上一个 set.a 方法即可，外部的使用不需要修改，而 C++ 或 Java 风格的 setter 和 getter 方法则把所有直接访问属性的程序修改成函数调用。

使用 set 和 get 方法还有一个优点，如果 a 是矢量或者矩阵，MATLAB 提供的 get 方法还支持 indexing 的方法，而作为普通方法的 geta 不支持 indexing 方式的访问。例如：

Command Line

```
>> obj = A();
>> obj.a = magic(2);
>> obj.a(1,:)
ans =
     1     3
```

使用 MATLAB 面向对象语言，要意识到其和传统面向对象语言的区别，我们建议对属性的访问和赋值应该使用 set 和 get 函数，而不要使用类似 C++ 和 Java 风格的 setter 和 getter。

2.10 Clear Classes 到底清除了什么

在 MATLAB 中，类的定义作为一种“信息”，也存在于内存中。[①]当类被首次使用时，MATLAB 会把该类定义一次性地装载到内存中，之后再声明类的对象时，就不需要再次去重新读入类的定义了。正是因为这个原因，在面向对象编程的过程中，如果定义了一个类，声明了一个对象，随后改变了该类的定义，再尝试声明一个对象时，会出现下面的 Warning Message：[②]

Command Line

```
Warning: The class file for '___' has been changed; but the change cannot be applied
 because objects based on the old class file still exist. If you use those objects,
 you might get unexpected results. You can use the 'clear' command to remove those
 objects. See 'help clear' for information on how to remove those objects.
```

这是 MATLAB 在提示用户，类的定义已经改变，但是工作空间中关于该类的定义还是旧的。不能刷新这个类的定义。其根本原因是：工作空间中还有该类的对象（instance）。该对象是用旧的定义声明出来的，如果刷新了，会造成同一个类的各个对象之间不一致。如果想使用新的类的定义，要先把工作空间中的和该类有关的，旧的对象先清除掉。这句话的言下之意是，只要有旧的 instance 存在，新的类定义就无法生效。这时通常有以下两种选择。

1. 使用 Clear obj 命令

如果工作空间中还有其他重要的变量存在，用户不希望全部清除它们，那么可以有选择地清除对象，比如工作空间中有类 A 的对象 obj1 和 obj2，并且我们修改了类 A 的定义，则只需要执行：

Script

```
>> clear obj1 obj2
```

MATLAB 接到这个指令后，清除了和 A 类相关的对象，下次再声明 A 的对象时，就可以使

①其本身也是一种对象，详见第13章。
②在 MATLAB R2014b 中引入了新的自动更新类定义的功能，将不会再出现该警告信息。

用新的定义了。

2. 使用 Clear Classes 命令

如果工作空间中的所有变量都不重要，或者如果一次性修改了几个类的定义，用户不想一个一个清除具体的对象，还可以使用`clear classes`命令，即

Script

```
>> clear classes
```

该命令将清除：

- 工作空间中的所有变量。
- 如果函数所持有的 persistent 变量是一个类的对象，也将被清除[①]。
- 所有之前被装载的类的定义。
- 类中的 Constant 属性。

①除非使用 mlock。

第 3 章　MATLAB 的句柄类和实体值类

3.1　引子：参数是如何传递到函数空间中去的

句柄（Handle）的概念，其实一直就存在于 MATLAB 语言中[①]。只有在 2008a 版本以后，MATLAB 开始向用户正式提供面向对象编程的功能，我们才终于有机会对 Handle 类有一个全面的了解。这一节我们从大家所熟悉的函数的参数传递机制出发，通过观察 MATLAB 的内存变化来初步认识 Handle 类。虽然这些知识点有些底层，但这却是正确使用 MATLAB Handle 类的基础。所谓 MATLAB 函数就是把一些指令集中在一个模块中，而使用函数的过程，就是外部程序调用该模块，传递给该函数的若干参数，然后函数做计算，并且把结果返回。比如在 MATLAB 中，一个简单的加法的函数可以定义如下：

myAdd.m

```
function result = myAdd(a,b)
    result = a + b;
end
```

该函数接收两个参数，做加法并把结果返回，在命令行中这样调用该函数：

Command Line

```
>> c = myAdd(1,2)
c =
     3
```

函数在运行时，MATLAB 会建立函数的工作区，工作区中存放函数内部变量，函数调用完毕，该工作区被销毁。我们多次提到，MATLAB 是弱检查类语言，因此在使用函数时，不检查参数的类型，所以这个 myAdd 函数的参数不但可以是标量，而且还可以是矢量或者矩阵[②]，比如：

Script

```
a =   eye(2,2)
b =   rand(2,2)
c =   myAdd(a,b)
```

Command Line

```
a =
     1     0
     0     1
b =
    0.3972    0.1277
    0.1983    0.7607
c =
    1.3972    0.1277
    0.1983    1.7607
```

我们再把参数的维数变得大一些，比如要做两个 10000×10000 的随机矩阵的加法[③]，当做 myAdd 的输入参数：

①比如 Handle Graphics 的函数返回的对象就是类似的 Handle 对象。
②这是 MATLAB 独一无二之处，即函数是支持向量化的。换句话说，这相当于 C++ 和 Java 的模板。
③为了说明问题，特意使用大型的矩阵，读者可以根据自己计算机的物理内存，选择合适的矩阵大小。

Script

```
N = 10000;
a = rand(N,N);
b = rand(N,N);
c = myAdd(a,b);
```

MATLAB 默认的数值数据类型是 double，每个 double 在 MATLAB 的工作空间中占 8 字节，所以 a 和 b 矩阵的大小大概每个是 760MB。这时，我们不由地要问一下，这么大的矩阵到底是怎么传到函数的工作空间中去的呢？如图3.1中问号所示，MATLAB 是不是真的在 myAdd 的函数空间中创建了 a 和 b 的拷贝，然后做相加得到 result，在把 result 拷贝回来给 c？直觉告诉我们，如果真的把两个 760MB 的矩阵拷贝到 myAdd 函数空间中去，那么这个函数做计算时，所需要的内存空间就是 760MB×3。这样做似乎效率不是很高！

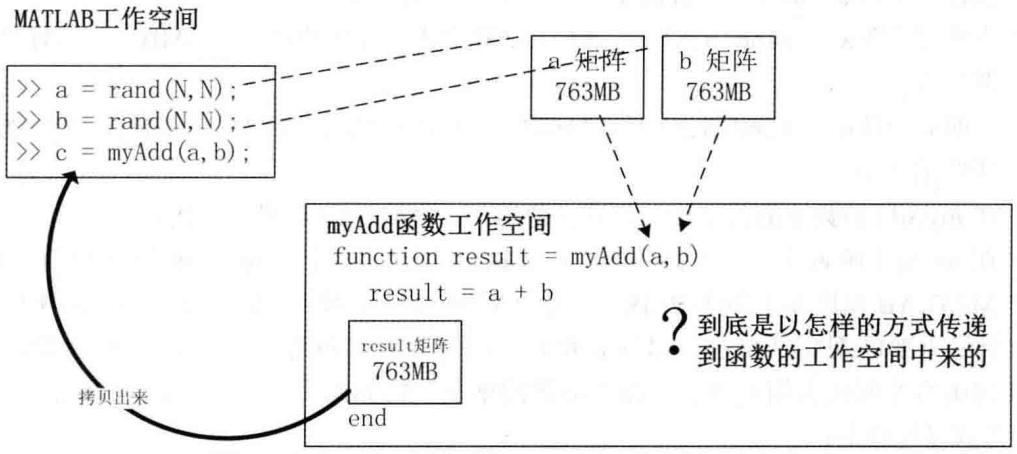


图 3.1 参数 a 和 b 到底是如何传递到函数空间当中去的

回答这个问题涉及 MATLAB 的函数参数传递机制。MATLAB 的函数参数传递机制到底是传值（函数空间的内部构造一个全同的拷贝），还是有什么其他更有效的方式？为了搞清楚这个问题，首先使用一下 memory 函数，这个函数可以帮助监视 MATLAB 内存的使用是如何随着函数调用变化的。

在命令行中输入 memory，MATLAB 将返回如下基本信息：

Command Line

```
>> memory
Maximum possible array:             45253 MB (4.745e+010 bytes)  % 机器能构造最大 Array
Memory available for all arrays:    45253 MB (4.745e+010 bytes)
Memory used by MATLAB:                535 MB (5.614e+008 bytes)  % 目前 MATLAB 已用内存
Physical Memory (RAM):              24567 MB (2.576e+010 bytes)  % 用户计算机的物理内存
```

事实上，memory 函数返回的是一个结构体，可以通过其中的一个 field 来提取目前 MATLAB 使用的内存，所以可以利用 memory 函数[①]，来纪录 Script 的每一行命令运行结束之后的内存使用情况。下面给出的是：左边为程序，右边为 MATLAB 内存的使用情况[②]。

①MAC 和 Linux 的 memory 命令不支持返回 MATLAB 可以使用的最大内存。
②返回结果将因个人而异，但是内存的相对增量是一样的。

Script

```
clear all ;
N = 10000;
a = rand(N,N);
b = rand(N,N);
c = myAdd(a,b);
```

Command Line

```
Line 2 MATLAB Memory : 622.543MB
Line 3 MATLAB Memory : 1385.4844MB
Line 4 MATLAB Memory : 2148.6641MB
Line 5 MATLAB Memory : 2911.8789MB
```

myAdd

```
function result = myAdd(a,b)
    result = a + b;
end
```

Command Line

```
Line 2 MATLAB Memory : 2148.6641MB
Line 3 MATLAB Memory : 2911.8789MB
```

可以发现：

- 执行到N =10000 时，MATLAB 使用了 622.54MB 的内存。
- 声明完矩阵a = rand(N,N) 后，MATLAB 的内存使用增加了 763MB，这恰好是 a 矩阵的大小。
- 声明完矩阵b = rand(N,N) 后，MATLAB 的内存使用又增加了 763MB，这恰好是 b 矩阵的大小。
- 在 myAdd 函数中的第 2 行，没有任何语句，MATLAB 没有增加内存。
- 在 myAdd 函数中，result = a + b 执行完后，一个新的 result 矩阵构建了出来，MATLAB 仅增加了 763MB 内存。这证明 myAdd 函数中并没有做 a 和 b 的拷贝。

一个最重要的观察结果是，尽管 a 和 b 矩阵被当做参数传递进了 myAdd 函数，但是 myAdd 的函数空间所占用的内存并没有显著的增加（763MB×2）。该例中，MATLAB 实际的传递方式是这样的：

- 如果参数的值在函数的内部没有被改变，函数的工作空间只复制了参数的必要数据，而实际的数据仍然在函数工作空间之外。
- 这些存在于函数工作空间中的必要数据通常只有一百多个字节，其中提供了访问实际数据的一种渠道（指针），如图3.2中的虚线所示。更具体一点，即提供了 a 矩阵和 b 矩阵在 Main 工作空间内存中的实际位置。

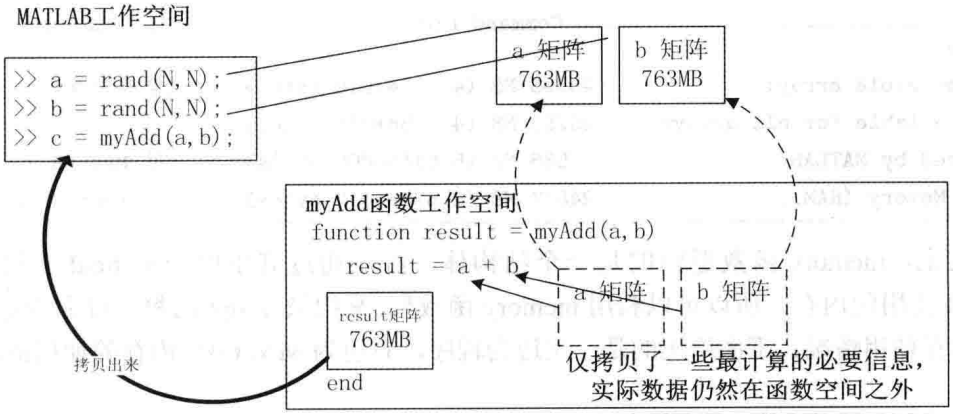


图 3.2 函数不修改参数时，参数中的必要信息被传进函数中，并不是全部

- 在函数内工作空间中要访问 a 和 b 矩阵的值时，就通过这些必要数据到 MATLAB 主工作空间中去访问，函数调用结束了，这必要数据也就跟随 Function 工作空间一起被销毁了。
- 这种做法也是容易理解的，如果在函数工作空间中的运算没有对参数做任何修改，就完全没有必要把参数全部拷贝到函数工作空间当中。

如果函数体内修改了参数的值，比如下面的 myAdd 函数，把传进来的 a 矩阵中的每个元素先做四舍五入，再做加法：

myAdd.m

```
function result = myAdd(a,b)
    a = round(a)  ;      % 对 a 先做四舍五入
    result = a + b;
end
```

那么在对 a 矩阵改变的前一刻，MATLAB 会在 myAdd 的函数工作空间中根据传入的必要数据，复制出一个局部的拷贝，以保证所有对 a 矩阵的修改都是局部的，即函数体内的对 a 的修改不会影响到 Main 工作空间中的 a 矩阵，这种技术通常叫做 Lazy Copy，就是不到万不得已时，不构造局部拷贝，如图3.3所示。

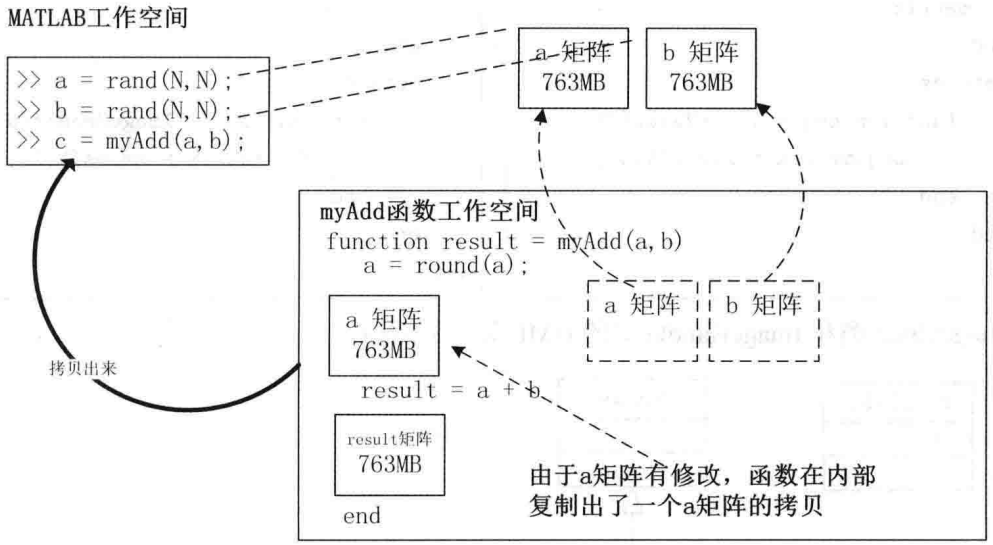


图 3.3 一旦函数要对参数作出修改，函数空间内会把要参数完全复制一份

严格说来，MATLAB 函数的参数传递机制是传值，而 Lazy Copy 可以被看做是在传值基础上的一个优化措施。如果 myAdd 函数的目的还包括修改传来的参数，那么还需要把修改过的 a 当做输出，修改才能生效。

myAdd.m

```
function [result a ] = myAdd(a,b)
    a      = round(a) ;
    result = a     + b;      % 对 a 先做四舍五入
end
```

如果 a 和 b 的数据体积很小，如何传递参数，对程序的性能影响没有太大的区别; 如果 a 和 b 的数据体积很大，比如是信号或者图像数据，做完全拷贝是低效的。MATLAB 面向对象提供了这样的技术，使得可以不用在函数体内构造局部拷贝也能修改传来参数的值。

3.2 MATLAB 的 Value Class 和 Handle Class

3.2.1 什么是 Value Class 和 Handle Class

MATLAB 面向对象的编程中有两种类，一种叫做 Value Class，可以翻译成数值类；另一种叫做 Handle Class，可以翻译为句柄类或者引用类。到目前为止，本书前面所举的 MATLAB 类的例子，大多都属于 Handle 类，如下列右面的代码所示。在这一节中，我们通过两种不同的方式来设计一个 Image 类，该类中包含一个属性叫做 matrix，用来详细的说明 Value 类和 Handle 类之间的区别[①]。要定义一个 Handle 类，用户只需要继承一个 Handle 基类即可，而 Value 类的定义如下列左面的代码所示。两个定义唯一的区别就是：用户定义的类是否继承了 MATLAB 内部提供的一个 Handle 基类。

ImageValue Value 类

```
classdef ImageValue
    properties
       matrix
    end
    methods
        function obj = ImageValue(N)
            obj.matrix = zeros(N,N);
        end
    end
end
```

ImageHandle Handle 类

```
classdef ImageHandle < handle  % 区别!!
    properties
        matrix
    end
    methods
        function obj = ImageHandle(N)
            obj.matrix = zeros(N,N);
        end
    end
end
```

ImageValue 类和 ImageHandle 类的 UML 如图3.4所示。

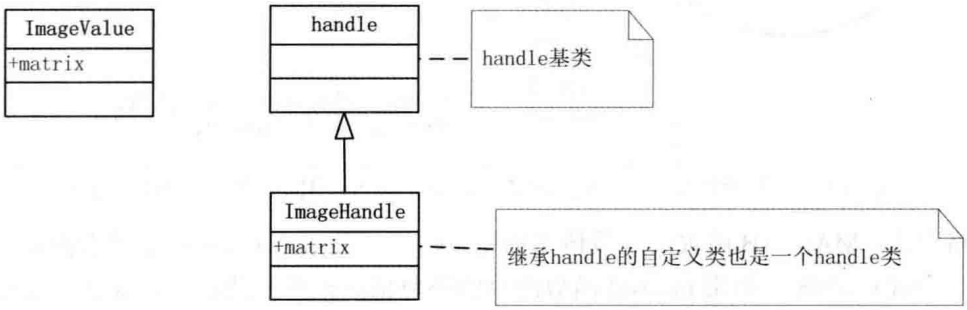


图 3.4 ImageValue 类和 ImageHandle 类的 UML

到目前为止，我们定义了两个 Image 类来包装矩阵数据，一个类定义成 Value Class，一个类定义成 Handle Class。现在来比较它们的区别。分别声明两个对象，并且检查 MATLAB 的内存使用情况：

①实际程序设计中，什么情况下使用 Value 类，什么情况下使用 Handle 类，请参见第3.2.5小节。

```
clear all ; clc ;
mValue  = ImageValue(10000)        ;
mHandle = ImageHandle(10000)       ;
```

Script

```
Line 1 MATLAB Memory : 610.4648MB
Line 2 MATLAB Memory : 1373.4063MB
Line 3 MATLAB Memory : 2136.3477MB
```

观察 MATLAB 内存的使用，首先观察到 MATLAB 依次在内存中对 mValue 和 mHandle 各分配了一个对象，每个对象的大小是 763MB。这是可以预料的。

但如果用 Whos 函数检查对象的大小，却发现这样一个事实：mValue 的大小是我们预计的 763MB，但是 mHandle 的大小却只有 112 字节[①]。实际情况是，MATLAB 确实为 Handle 类对象分配了内存，但是该对象中属性 matrix 的实际的大小却没有计入到 Handle 类对象所占用的内存中去。

Command Line

```
>> whos
  Name          Size                Bytes  Class          Attributes

  mHandle       1x1                   112  MatrixHandle
  mValue        1x1             800000104  MatrixValue
```

简单来说，原因是这样的：MATLAB 建立的这两个对象，在内存中建立如图3.5所示的两种不同的布局。

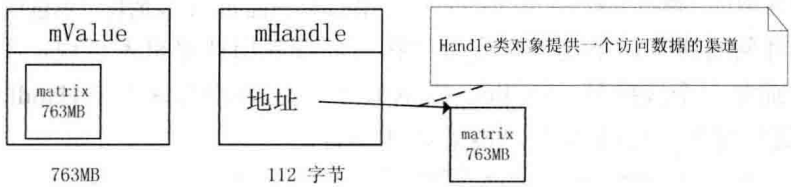


图 3.5 mHandle 对象中仅存放了数据访问的渠道

MATLAB 处理实体类对象的方式是：直接在内存中开辟一块区域，用以存放实体类的对象；MATLAB 处理句柄类对象的方式稍微多了一层，在内存中不但有一部分区域用来存放实际有用的数据，还有一个句柄对象，指向这块内存，如图3.5右所示。

“指向”在实用角度上可以被简单地理解成 Handle 类对象（即 mHandle）提供了对实际 matrix 的全权代理，用来提供对这片内存中数据的读和写，而对这个代理对象 mHandle 的使用，从外观上看和使用一般的 Value 类对象没有区别：

Command Line

```
>> mValue.matrix(1,1)
ans =
     0
>> mHandle.matrix(1,1)    % 使用上从外部看没有区别
ans =
     0
```

①使用不同的计算机和不同的版本，mHandle 的大小可能稍有差别，但是都远小于 763MB。

3.2.2 Value 类对象和 Handle 类对象拷贝有什么区别

Value 类和 Handle 类对象的另一大不同在拷贝时的行为，比如执行下面语句对 mValue 和 mHandle 做拷贝：

Script

```
nValue  = mValue  ;
nHandle = mHandle ;
```

MATLAB 对这两个看上去相似的命令的解释是完全不同的。在“概念上”，MATLAB 对于 Value 类对象，在内存中做了完全的拷贝，也叫深拷贝，如图3.6所示。

而对于 Handle 类对象，MATLAB 只拷贝 Handle 类对象本身，而没有拷贝句柄对象指向的实际数据，如图3.7所示。

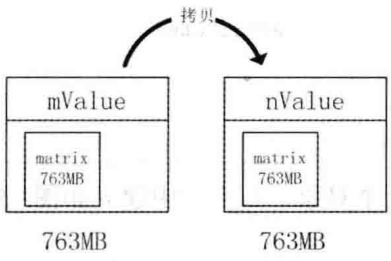


图 3.6 Value 对象的拷贝概念上是一个完全的复制

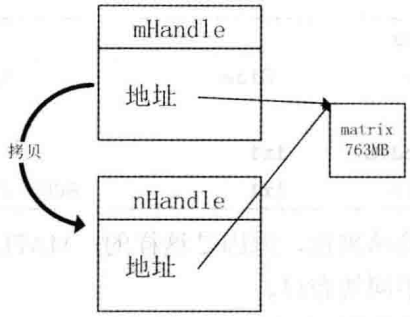


图 3.7 Handle 对象的拷贝不包括实际数据的复制

Handle 类对象的拷贝，不是一个完全的拷贝，或者用计算机术语说，不是一个深拷贝（Deep Copy），而是一个浅拷贝（Shallow Copy），因为它并没有深入到 Handle 类所指向的数据对实际数据进行拷贝。可以通过 whos 命令来验证。

Command Line

```
>> whos
  Name          Size                Bytes  Class          Attributes

  mHandle       1x1                   112  MatrixHandle
  nHandle       1x1                   112  MatrixHandle
  mValue        1x1             800000104  MatrixValue
  nValue        1x1             800000104  MatrixValue
```

浅拷贝最显著的一个特征是：如果通过 mHandle 来修改 matrix 属性，那么也会影响 hHandle 的 matrix 属性，因为这两个 Handle 对象共享了一份 matrix 数据。如何对 Handle 类的对象进行深拷贝是一个复杂的问题，我们留到第13.4节介绍。

3.2.3 Value 类对象和 Handle 类对象赋值有什么区别

Value 类对象和 Handle 类对象在拷贝上行为的不同，将带来赋值行为上的不同。3.2.2小节中，对于 Value 类对象的拷贝，MATLAB 从概念上在内存中做了完全拷贝，在命令行中使用 whos 可以验证 nValue 和 mValue 都是 763MB，但是如果检查 MATLAB 内存的使用却发现，对 mValue 做拷贝时，MATLAB 内存的使用并没有明显的增加。例如：

```
clear all;
mValue  = MatrixValue(10000);
mHandle = MatrixHandle(10000);
nValue = mValue ;   % 全拷贝
nHandle= mHandle;
```

```
Line 1 MATLAB Memory : 532.1641MB
Line 2 MATLAB Memory : 1295.1055MB
Line 3 MATLAB Memory : 2058.0469MB
Line 4 MATLAB Memory : 2058.0469MB  % 咦?
Line 5 MATLAB Memory : 2058.0469MB
```

这里还有另一个细节，是我们在第3.1节中提到的 Lazy Copy，即不到万不得已时，MATLAB 不会在内存中构造一个一模一样的拷贝。这个所谓的万不得已时，就是当对 nValue 中 matrix 的值作出改变时，当我们哪怕仅仅修改 nValue 矩阵中一个元素的值，MATLAB 在内存中马上就构造出一个实体的拷贝，然后再修改其中元素的值。

```
clear all;
mValue  = MatrixValue(10000);
mHandle = MatrixHandle(10000);
nValue = mValue ;
nHandle= mHandle;
nValue.matrix(1,1) = 10;  % 仅仅修改一个值
```

```
Line 1 MATLAB Memory : 532.1641MB
Line 2 MATLAB Memory : 1295.1055MB
Line 3 MATLAB Memory : 2058.0469MB
Line 4 MATLAB Memory : 2058.0469MB
Line 5 MATLAB Memory : 2058.0469MB
Line 6 MATLAB Memory : 2820.9883MB  % BOOM!
```

这时内存中 nValue 和 mValue 分别指向不同的数据，如图 3.8 所示。

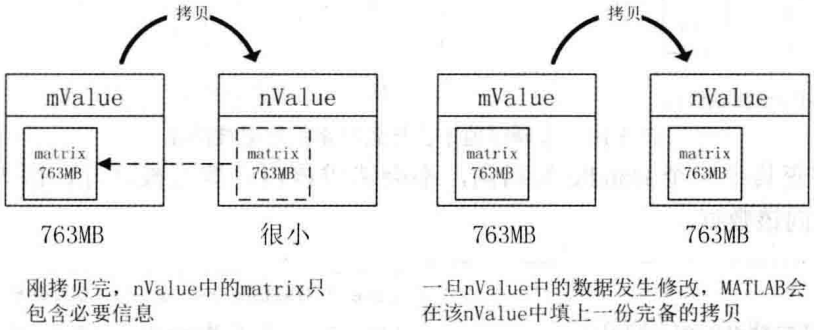


图 3.8 不到万不得已（被修改时），MATLAB 不会在内存中做完全的复制

对句柄对象的重新赋值要分情况讨论，比如下面对 mHandle 对象的重新赋值，如果内存数据只被一个 Handle 类的对象所指：

```
clear all;
mHandle  = MatrixHandle(10000);
nHandle =  MatrixHandle(10000);
mHandle = nHandle;
```

```
Line 1 MATLAB Memory : 607.7031MB
Line 2 MATLAB Memory : 1370.6445MB
Line 3 MATLAB Memory : 2133.5859MB
Line 4 MATLAB Memory : 1370.6445MB
```

重新给 mHandle 赋值，意味着原内存数据将不再被任何句柄所引用。这里重新赋值的结构是，原来的内存数据被清除，如图 3.9 所示。

从命令行的结果也可以看出，MATLAB 使用的内存减少了 763MB。如果有一个以上的 Handle 类对象指向同一块内存数据，如图3.10 所示，mHandle 和 lHandle 两个 Handle 对象就都指向了同一块数据。

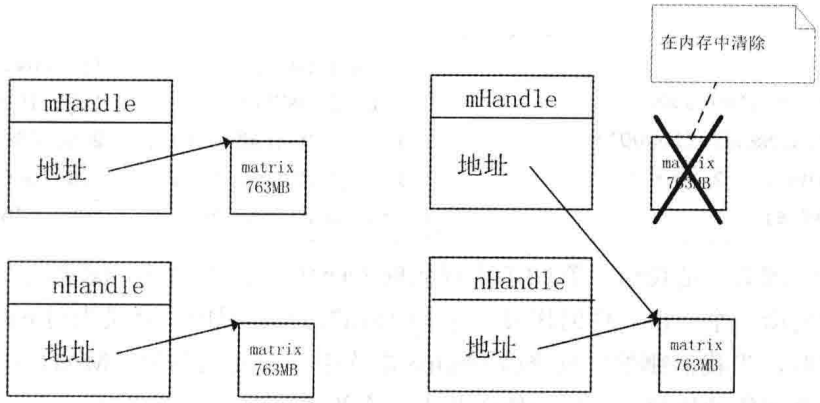


图 3.9 不再被引用的 matrix 将被清除掉

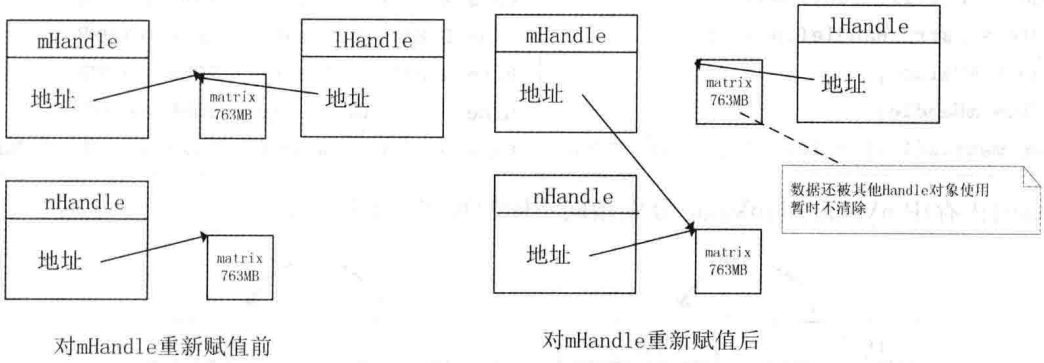


图 3.10 重新赋值不会导致原来的数据被清除

那么仅改变其中一个 Handle 的指向，不会造成该内存数据被清除，因为还剩下一个 Handle 对象指向该数据。

```
clear all;
mHandle  = MatrixHandle(10000);
lHandle = mHandle;
nHandle =  MatrixHandle(10000);
mHandle = nHandle;
```

```
Line 1 MATLAB Memory : 606.4961MB
Line 2 MATLAB Memory : 1369.4375MB

Line 4 MATLAB Memory : 2132.3789MB
Line 5 MATLAB Memory : 2132.3789MB
```

从命令行的结果可以看出，MATLAB 使用的内存没有变化。

判断内存数据是否被引用（指向）的技术叫做引用计数。也就是说，Handle 类对象指向内存数据，该内存数据还和一个计数器相联系，在 Handle 类被复制时，这个计数器其会自动 +1，记录指向该内存数据的 Handle 对象的数目。如果只有一个 Handle 对象指向该内存数据，当改变这个 Handle 对象的指向时，MATLAB 会自动销毁这个内存数据，如图3.11所示，因为没有 Handle 对象再引用这个内存数据了，不难理解，这个内存数据没有存在的必要了。

如果有多于一个的 Handle 对象指向内存数据，改变其中一个 Handle 对象的指向，不会造成内存数据被清除，但是引用计数被减 1，如图3.12所示。

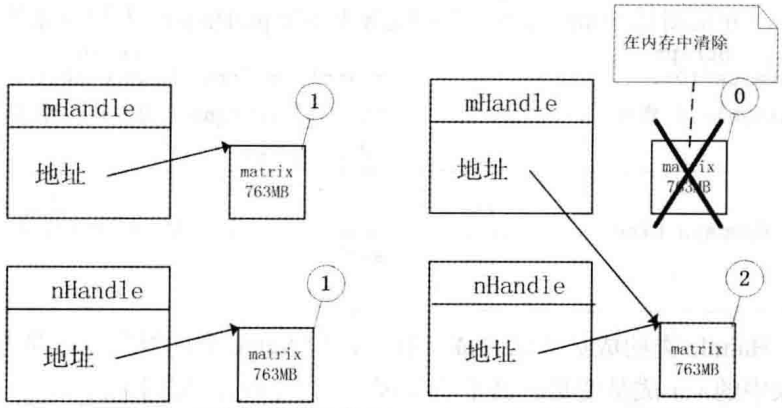


图 3.11 内部数据上的计数器如果为零，MATLAB 将清除之

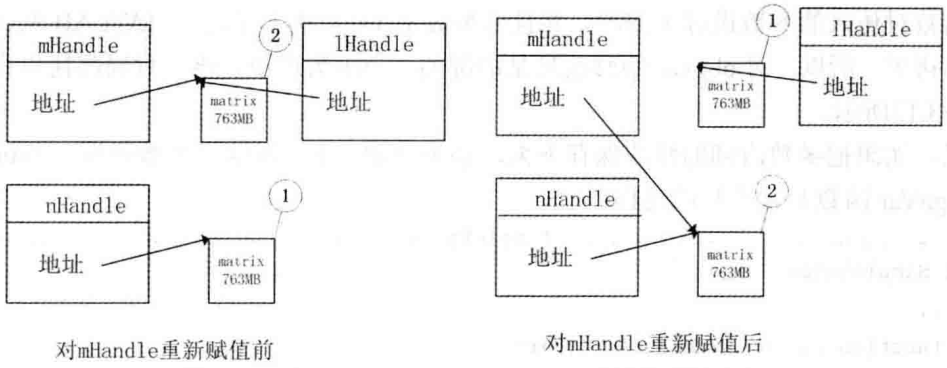


图 3.12 给 Handle 对象重新赋值会造成计数器的改变

3.2.4 Value 类对象和 Handle 类对象当做函数参数有什么区别

到目前为止，把 Handle 对象作为对数据访问的渠道，看上去像是多此一举。这一节把 Value 类对象和 Handle 类对象作为函数参数来解释它们的区别。下面左边的代码是一个 Value 类，右边的代码是一个 Handle 类的定义，它们各有一个成员属性叫做 var，并且都有一个成员方法，该成员方法企图修改成员属性 var 的值。

SimpleValue

```
classdef SimpleValue
    properties
        var
    end
    methods
        function obj = SimpleValue(var)
            obj.var = var;
        end
        function assignVar(obj,var)
            obj.var = var ;
        end
    end
end
```

SimpleHandle

```
classdef SimpleHandle < handle % 唯一区别
    properties
        var
    end
    methods
        function obj = SimpleHandle(var)
            obj.var = var;
        end
        function assignVar(obj,var)
            obj.var = var ;
        end
    end
end
```

在 Script 中，分别测试 SimpleValue 类的对象和 SimpleHandle 类的对象的成员方法：

Script
```
aValue = SimpleValue(10);
aValue.assignVar(20); % 修改
aValue.var
```

Script
```
bHandle = SimpleHandle(10);
bHandle.assignVar(20); % 修改
bHandle.var
```

得到输出如下：

Command Line
```
ans =
    10
```

Command Line
```
ans =
    20
```

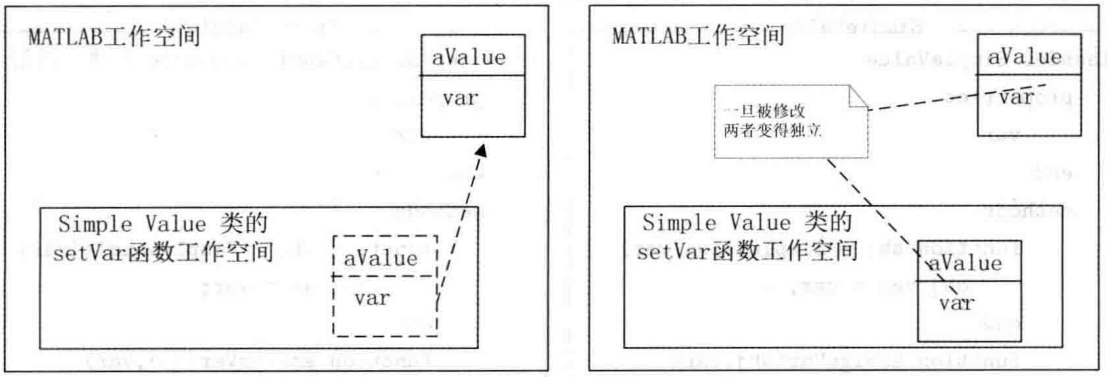
我们发现，Handle 类的成员方法完成了任务，但 Value 类的对象的成员方法没有按期望的那样修改对象中的 var 成员变量。其中的原因大家应该可以猜到了：

对于 Value 类对象，参数的传递方式是：拷贝必要的信息到函数工作空间中，当 MATLAB 发现该函数对传入的参数进行了修改，并且该参数是 Value 类对象时，MATLAB 就构造出了一个局部拷贝。所以，对 obj.var 的修改只是局部的，当函数退出，该临时局部拷贝也就消失了，如图3.13所示。

所以，如想把函数内部的修改保存下来，必须再把 obj 当做输出参数返回，SimpleValue 类的 assignVar 函数必须改写成这样：

SimpleValue
```
classdef SimpleValue
    ......
        function obj = assignVar(obj,var)
            obj.var = var ;
        end
    ......
end
```

说得更具体一点，必须再把 obj 当做输出参数返回，这句话隐含的意思是，Value 类通过方法操作对对象内部数值的改动，必须返回（即创造了）一个新的对象。



如果不改变aValue中的数据的值，Function Workspace中仅有aValue的必要信息

如果改变aValue中的数据的值，Function Workspace将构造出一个独立的Copy

图 3.13 一旦要被修改，函数空间构造出一个局部拷贝，修改是局部的

对于 Handle 类对象，参数的传递方式也是拷贝局部对象，但在函数内部，函数修改的不是针对 Handle 对象的而是 Handle 对象所指向的数据，如图3.14所示。所以，MATLAB 内

部并不需要做额外的处理，并且该函数成功的修改了 Handle 对象所指向的数据。

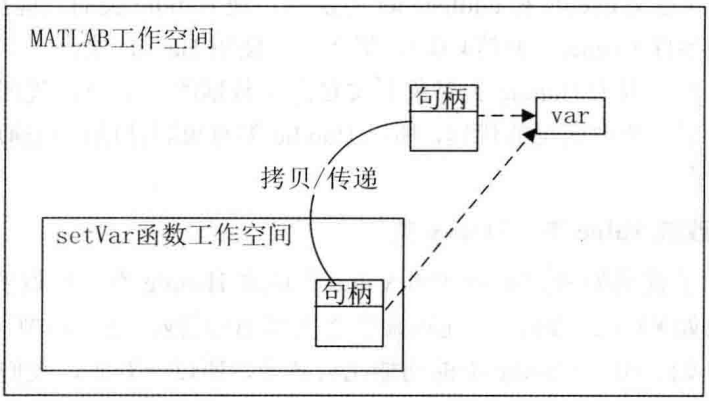


图 3.14 在函数空间中修改 Handle 对象即修改了实际数据

所以句柄类对象可以用来在函数中修改传来的参数。总地来说，MATLAB 对参数的处理方式依赖于用户的判断：如果用户希望在函数内修改参数，则用户需要传递 Handle 类对象，否则；就要使用 Value 类，并且把修改完的对象当做结果返回。

3.2.5 什么情况下使用 Value 类或 Handle 类

通过前几节的比较，可以总结出在何种情况下需要使用 Value 类，在何种情况下需要使用 Handle 类。

Value 类适用于比较简单的数据，其行为和 MATLAB 普通变量。matrix，struct，cell 基本一致。它们完全由其所含的数据值类定义。如果使用者并不在意副本的存在，并且希望每次执行对象的拷贝，都可以得到一个独立的副本，可以使用 Value 类。如果一个数据在其他多处有副本，并且我们希望修改其中的一个副本，其他所有的副本也可以受到影响，也被修改，可以使用 Handle 类。

Handle 类对象提供的是对实际数据的一个访问渠道，从实用计算的角度来说，如果我们的数据体积比较大，我们希望这些数据在各个方法、函数之间的传递迅捷，不需要被局部拷贝的话，则可以用 Handle 类来包装数据。MATLAB 的 Handle Graphics 是 MATLAB 中最复杂的一个 Handle 类，图形对象的体积比较大，把它们设计成句柄类，通过它们的句柄对它们进行控制。比如下面的代码中，h1 和 h2 实际指向的都是通过一个 line 对象。

Command Line

```
>> h1 = line ;
>> h2 = h1;
>> set(h2,'Color','red');
```

从物理的角度看，如果类的对象对应一个独一无二的物理对象，比如一个串口、一个打印机、一个窗口、对一个文件的访问接口，那么应该把它设计成 Handle 类，因为对这些对象的拷贝仅仅拷贝的是一个访问它们的渠道。如果把它们设计成 Value 类，当我们做拷贝时，将面临这样一个难题，无法从语义上解释一个独一无二的实体，在程序中有两个全同的拷贝。在第15.2节中，我们还将介绍如何控制类所能声明的对象的数量，对于这种独一无二的物理实体，使其只能有一个对象存在。

从功能的角度看，Value 类没有提供任何内置的方法，而 MATLAB 的 Handle 基类提供了 delete 方法，可以定义 events 和 addlistener 方法等，使 Handle 类的功能更加强大。如果想在定义的类中使用事件 Events（见第4 章），那么就要使用 Handle 类。

从性能的角度看，因为 Handle 类对象其实是内部数据的一个全权代理，对 Handle 类对象属性的访问经过了一个中间层的代理，所以 Handle 类对象属性的访问速度比 Value 类对象属性访问的速度稍慢。

1. 把 Handle 类改成 Value 类：Money 类

下面举一个例子说明如何判断该使用 Value 类还是 Handle 类。假设要用程序模拟和货币相关的计算。比如 ¥5 ×2 = ¥10，5 元钱乘以 2 变成 10 元钱。这个模拟可以从设计一个人民币（RMB）类开始。由于 Handle 类的功能比较齐全，所以一开始，我们先尝试着把 RMB 类设计成 Handle 类：

RMB

```
classdef RMB < handle
    properties(SetAccess = private)
        amount
    end
    methods
        function obj = RMB(val)
            obj.amount = val;
        end
    end
end
```

该类有一个 SetAccess 是私有的属性 amount，该属性用来保存票面的价值，其值在构造函数中初始化。为了模拟货币乘法的需要，下一步我们先添加一个乘法 times 方法：

RMB

```
......
    methods
        function times(obj,multiplier)
            obj.amount = obj.amount*multiplier
        end
    end
......
```

在命令行上，测试该类的使用如下：

Command Line

```
>> five = RMB(5)
five =
  RMB with properties:
    amount: 5
>>
>> five.times(2)
obj =
```

```
  RMB with properties:
    amount: 10
```

这似乎看上去一切正常，但仔细观察对象的输出，可以发现，有一些不合常理的地方。我们一开始把对象的名称设成了 five（5 元钱），但是经过一番操作之后，这张 5 元钱变成了 10 元钱，货币作为现实世界中一个实际存在的实体，没有与之对应的这样的操作（让 5 元钱值 10 元钱的操作）。再回头审视一下程序的本意，¥5 ×2 = ¥10，这其实要模拟的是：两张 5 元钱相加在变成 10 元钱的这样的一个过程，（读者可以想象一下去银行拿两张 5 元去换一张 10 元），这个 10 元钱应该是一个新的对象才对！所以 times 函数必须返回一个新的对象。而原来 5 元钱这个对象，将一直是面值 5 元，应该不允许任何操作将其改成 10 元！回想上节我们对 Value 类当函数参数定下的标准：即对内部数值的改动，必须返回（实际是创造了）一个新的对象。所以这个 RMB 类应该实际成 Value 类才对！于是，我们更改最初的设计，把 RMB 类设计成 Value 类：

RMB

```
classdef RMB                                        %  现在改成了 Value 类
    properties(SetAccess = private)
        amount
    end
    methods
        function obj = RMB(val)
            obj.amount = val;
        end
         function newobj = times(obj,multiplier)
            newobj = RMB(obj.amount*multiplier);    %  这里返回新的对象
        end
    end
end
```

在命令行上的运算，five 对象被乘以 2，返回一个新的 RMB 对象，命名为 ten，也就合乎常理了。

Command Line

```
>> five = RMB(5)
five =
  RMB with properties:
    amount: 5
>>
>> ten = five.times(2)
ten =
  RMB with properties:
    amount: 10
```

这样的设计还可以进一步扩展到加法上，添加一个加法方法：

RMB

```
......
    function newobj = plus(obj,obj2nd)
        newobj = RMB(obj.amount+obj2nd.amount);
    end
......
```

可以在命令行上这样做 RMB 对象的加法运算：

Command Line

```
>> five = RMB(5);
>> ten = five.plus(RMB(5))
ten =
  RMB with properties:
    amount: 10
```

在第12.5节中还会介绍重载+ 号运算符，使得在命令行上可以这样做对象之间的加法

Command Line

```
>> ten = RMB(5) + RMB(5)
ten =
  RMB with properties:
    amount: 10
```

Value 类的另个设计实例可以参见第12.5节的 String 类。

2. 把 Value 类改成 Handle 类：Customer 类

如果设计从简单的 Value 对象开始，在其中存放少量的数据。而后你可能会对这个数据中的内容加以修改，并且希望对任何一个对象的修改都能影响到所有使用此对象的地方，这时，需要把 Value 类改成 Handle 类。下面有一个 Order 类，其中有一个属性 ID 记录订单号，另个属性 customer 记录顾客的信息，该属性本身也是一个对象，属于 Customer 类，其中记录顾客的姓名和地址，在程序开发的初期，我们把该 Customer 类设计成 Value 类。最初的代码如下：

Order 类

```
classdef Order
  properties
    id
    customer
  end
  methods
    function obj = Order(id,customer)
      obj.id = id;
      obj.customer = customer;
    end
  end
end
```

Customer 类

```
classdef Customer
  properties
    name
    address
  end
  methods
    function obj = Customer(name,addr)
      obj.name = name;
      obj.address = addr;
    end
  end
end
```

使用方式为：先声明 Customer 对象来模拟用户注册，然后用户下单，生成三个 Order 对

象。使用两个 Value 类可以完成如下的简单工作。

Command Line

```
clear all;

Marc = Customer('Marc','Boston');
Steve = Customer('Steve','Cambridge');

o1 = Order('00001',Marc);
o2 = Order('00002',Marc);
o3 = Order('00003',Steve);
```

其中 Marc 是 Value 类对象，即使 o1 和 o2 两份订单都属于 Marc，但其中的 customer 属性仍然是各自独立的。如果 Marc 在下完订单之后，修改了自己的地址，我们会希望这样的修改能够影响所有 Marc 的订单，但是目前 o1 和 o2 的地址仍然是独立的，代码如下：

Command Line

```
Marc.address = 'Natick';    % 修改了地址
o1.customer
o2.customer
```

Command Line

```
ans =
  Customer with properties:
      name: 'Marc'
   address: 'Boston' % 地址仍然是 Boston
ans =
  Customer with properties:
      name: 'Marc'
   address: 'Boston' % 地址仍然是 Boston
```

这就意味着：要想仅仅在一处的修改能影响到所有使用该对象的地方，那么 o1 和 o2 订单就应该共享一个 Customer 对象。所以程序开发到这里，我们应该把 Customer 类从 Value 类变成 Handle 类。这样，对 Marc 对象地址的修改就可以影响到所有 Marc 所拥有的订单对象了。

新的 Customer

```
classdef Customer < handle
    properties
        name
        address
    end
    methods
        function obj = Customer(name,address)
            obj.name = name;
            obj.address = address;
        end
    end
end
```

使用如下：

CommandLine

```
Marc.address = 'Natick';    % 修改了地址
o1.customer
o2.customer
```

CommandLine

```
ans =
  Customer with properties:
       name: 'Marc'
    address: 'Natick'    % 地址修改了
ans =
  Customer with properties:
       name: 'Marc'
    address: 'Natick'    % 地址修改了
```

3.3 类的析构函数（Destructor）

3.3.1 什么是对象的生存周期

在 MATLAB 中，不论对象还是普通的变量（严格来说，普通变量也是对象）不论大小，都占用了内存资源，不需要时，就需要释放其所占用的资源。所谓对象的生存周期就是从对象产生到被释放的过程。

比如我们在命令行中，构造一个矩阵，做一些计算，最后再把这个 matrix 从 Main 工作空间中清除掉：

Command Line

```
>> matrix = rand(100,100);
>> whos
  Name          Size              Bytes  Class     Attributes
  matrix      100x100             80000  double
>>
>> clear matrix
```

matrix 变量的生存周期是在第 1 到 5 行之间。再比如，在函数 foo 中创建一个临时变量：

foo.m

```
function foo()
  ......
  matrix = rand(100,100);
  ......
end
```

该变量的生存周期是从第 3 行一直到 end 结束，对于局部变量，用户不需要显式地删除它，到函数结束时，MATLAB 会自动清除所有的局部变量。

3.3.2 什么是析构函数（Destructor）

在第2.5节曾提到类的 Constructor 负责产生对象以及初始化成员变量，打开文件句柄等，这些工作通常都占用一定的系统资源，比如内存。和类 Constructor 相反，析构（Destructor）函数在对象脱离其作用域或者被销毁时（比如对象所在的函数已经调用完毕）负责收尾工作，比如关闭文件句柄、释放数据所占的内存空间等，如图3.15所示。MATLAB 规定：对执行这种任务的类方法要命名成 delete。

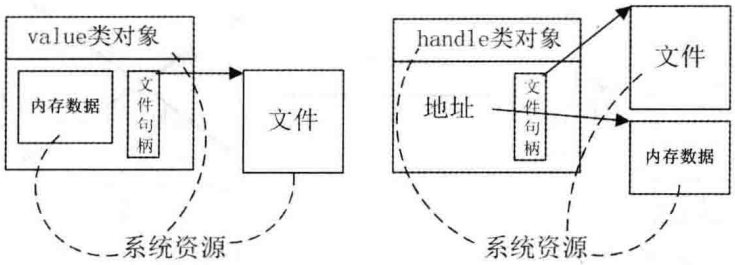


图 3.15 对象会占用系统资源，这些资源不需要时要及时的释放

无论是 Value 类还是 Handle 类，如果在对象的清除过程中涉及释放其所占用的系统资源，用户都需要定义自己的 delete 函数。由于 delete 的功能和 clear 命令类似，下面先从简单的 clear 命令谈起。

3.3.3 对 Object 使用 clear 会发生什么

1. 对一个 Value Object 使用 clear

对一个 Value Object 使用 clear，将直接把这个对象从工作空间中清除，用户如果没有显式地调用 clear 命令，当该对象离开其作用域时，MATLAB 也会自动地调用 clear 命令清除该对象，如图3.16所示。

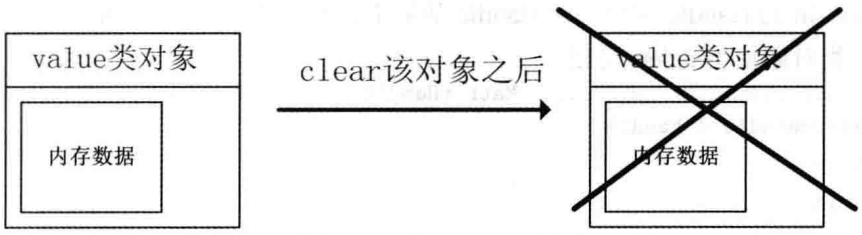


图 3.16 使用 clear 将把 Value 对象彻底清除

2. 对一个 Handle Object 使用 clear

对一个 Handle Object 直接使用 clear，将清除该 Handle 本身，该 Handle 指向的实际数据是否会被清除，则取决于该数据是否还被其他 Handle 所指向。如图3.17的情况，当只有一个 Handle 指向该 matrix，使用 clear 之后，该 Handle 被清除，matrix 上的引用计数变成零，并随着 Handle 对象的销毁而被释放。

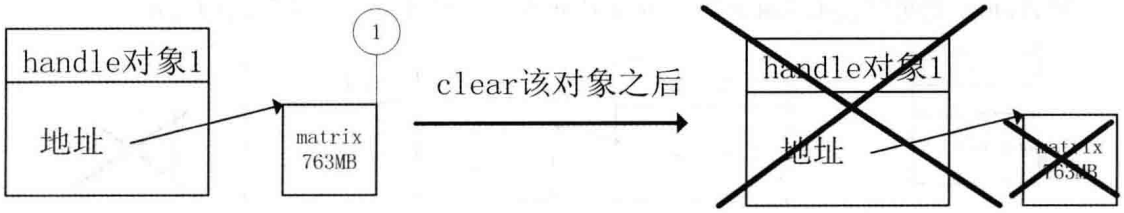


图 3.17 引用计数等于 1，clear 将把 Handle 对象彻底清除

如果 matrix 上的引用计数是 2，清除掉一个 Handle 对象将不会造成 matrix 所占用内存的释放，但是 matrix 上的引用计数将变成 1，如图3.18所示。

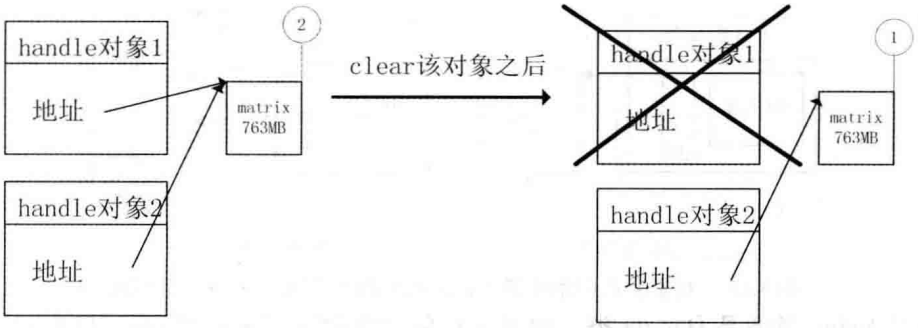


图 3.18 引用计数大于 1，clear 将清除一个 Handle 对象并把引用计数减 1

注意，这里的清除操作，将忽略对象中的属性的访问权限。也就是说，无论是对 Value 对象，还是 Handle 对象，即使是对象中有私有属性，在外部调用 clear 方法时，这些私有属性也将会被也会被清除。

3.3.4 对 Object 使用 delete 会发生什么

Value 类本身不存在默认的 delete 方法，所以如果用户不定义 delete 方法，对该对象使用 delete 方法将无从谈起。在后面的章节将讨论在什么情况下，最好要定义一个 delete 方法。这里集中讨论 Handle 类对象的 delete 方法。我们知道，用户定义 Handle 类需要继承自一个 MATLAB built-in 的 Handle 基类，该 Handle 基类中已经定义了一个 delete 方法，于是可以直接对 Handle 类对象使用 delete 方法。

MatrixHandle.m

```
classdef MatrixHandle < handle
    properties
        matrix
    end
    methods
        function obj = MatrixHandle(N)
            obj.matrix = zeros(N,N);
        end
    end
end
```

对 Handle 类对象使用 delete 将释放该 Handle 所指向的数据，如图3.19所示。

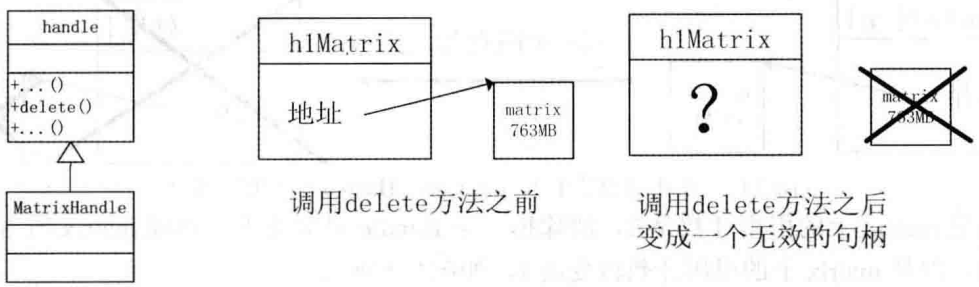


图 3.19 对 Handle 类对象使用 delete 将释放该 Handle 对象所指向的数据

可以通过检查 MATLAB 的内存消耗来验证 delete 的效果：

```
1 clear all ;  clc ;
2 N = 10000;
3 h1matrix = MatrixHandle(N);
4 h1matrix.delete() ;
```

```
Line 1 MATLAB Memory : 577.2695MB

Line 3 MATLAB Memory : 1340.2109MB
Line 4 MATLAB Memory : 577.2695MB
```

如果有两个 Handle 同时指向一个 matrix，对 Handle 类对象调用 delete 方法，MATLAB 不管内存数据上的引用计数是多少，如图3.20所示的情况，MATLAB 都将直接把内存数据清除，剩下两个失效的 Handle 类对象。

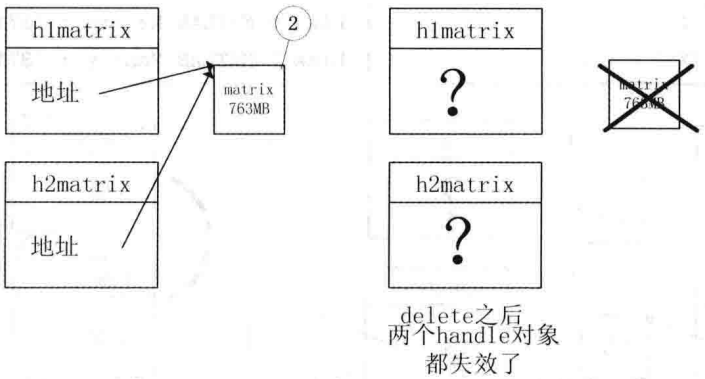


图 3.20 使用 delele 强行删除 Handle 对象所指向的内部数据

也可以通过检查 MATLAB 的内存消耗来验证：

script

```
clear all ; clc ;
N = 10000;
h1matrix = MatrixHandle(N);
h2matrix = h1matrix ;  %    h1matrix h2matrix 指向同一块内存数据
h1matrix.delete() ;
whos
```

MATLAB 还提供一个 isvalid 函数，用来检查 Handle 对象是否有指向任何实际的数据。该函数将对失效的 Handle 类对象返回 false，结果如下：

Command Line

```
>> isvalid(h1matrix)
ans =
     0
>> isvalid(h2matrix)
ans =
     0
```

删除所指向的内存数据之后，如果程序还尝试使用这个 Handle，MATLAB 将抛出一个错误，可以用 isvalid 函数实现判断一个句柄的有效性来避免这种错误。

Command Line

```
>> h1Matrix.matrix
??? Invalid or deleted object.
```

还可以重新让 Handle 类对象指向其他的内存数据，注意下面第 3 行删除了 h1Matrix Handle 对象所指向的内存数据，MATLAB 使用的内存减少了 673MB。紧接着又构造了一个 Handle 类对象，把 h1Matrix 也指向这个新的对象的内存数据，h1Matrix 重新也变得有效，如图3.21所示。

```
1 clear all ;  clc ;
2 N = 10000;
3 h1matrix = MatrixHandle(N);
4 h2matrix = MatrixHandle(N);
5
6 h1matrix.delete() ;
7 h1matrix = h2matrix ;
```

```
Line 1 MATLAB Memory : 608.6172MB

Line 3 MATLAB Memory : 1371.5586MB
Line 4 MATLAB Memory : 2134.5MB

Line 6 MATLAB Memory : 1371.5586MB
Line 7 MATLAB Memory : 1371.5586MB
```

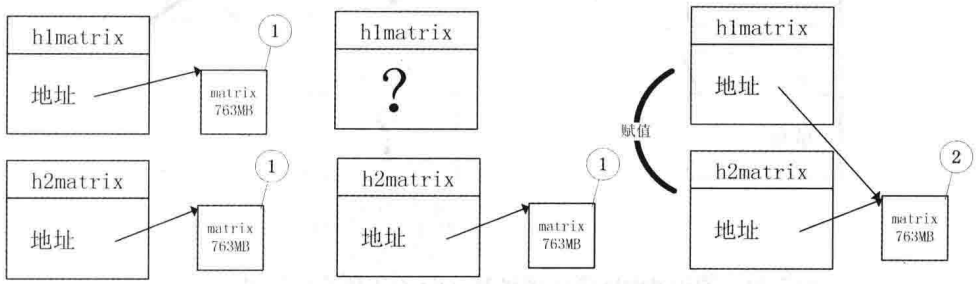


图 3.21 失效的 Handle 对象可以重新被赋值

3.3.5 什么情况下 delete 方法会被自动调用

MATLAB 作为一种科学计算语言，具有良好的内存管理机制，尤其是不需要用户去操心内存资源的释放，更是其主要特点之一[①]。在一些特殊时，MATLAB 会自动地调用 delete 方法，以达到对内存的自动管理。

- 对 Handle 对象的重新赋值会触发 MATLAB 调用 delete 方法，mHandle 在被重新赋值之前所指向的内存数据引用计数是 1，重新赋值之后，该块内存引用计数为零，delete 方法将被调用，该内存数据将被释放，如图3.22所示。

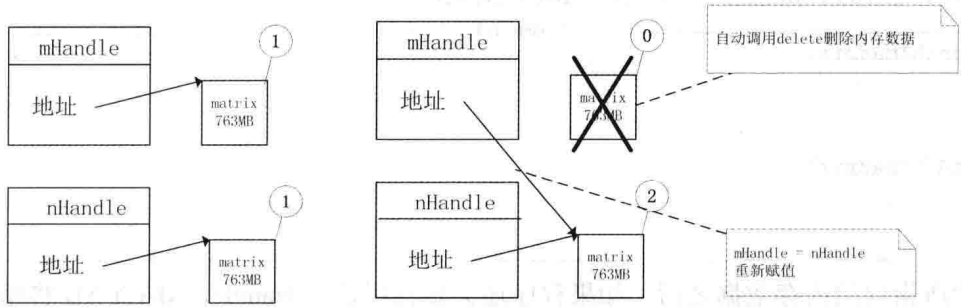


图 3.22 引用计数为零，delete 方法将自动被调用

- 当用户在工作空间中使用 clear 命令时，所有工作空间中的 Handle 对象的 delete 方法也会被调用。

①MATLAB 对资源的释放发生在确定的时刻，和 Java 的垃圾收集机制不同。

□ 当对象离开了作用域时，MATLAB 会自动调用其 delete 方法释放内存；如果 Handle 类对象是一个局部对象，离开作用域时，MATLAB 会自动清除 Handle 对象以及其所指向的所有内存数据。

```
clear all ;  clc ;
foo(10000);
```

```
Line 1 MATLAB Memory : 583.8398MB
Line 2 MATLAB Memory : 583.8398MB
```

函数工作空间以及内存使用如下：

```
1 function foo(N,tracker)
2     h1matrix = MatrixHandle(N);
3 end
```

```
Line 2 MATLAB Memory : 1346.7813MB
```

□ 如果内存数据上有两个或者两个以上的 Handle 类对象所指，其中一个是局部 Handle 类对象，则 MATLAB 在销毁局部对象时不会影响另一个 Handle 类对象，只是把引用计数减 1。

□ 当 A 和 B 都是 Handle 类，并且 B 对象隶属于 A 对象，即两个类是组合关系时[①]，A 对象的销毁，也将导致 B 对象的 delete 函数被调用，如果没有其他的 Handle 引用 B 对象，如图3.23所示。

如图3.23右所示，如果属性 B 上的引用计数大于 1，即还有其他的 Handle 对象指向该属性，则对 A 对象使用 Delete 方法，将把 B 对象上的引用计数减 1，B 对象的 delete 方法并不会被调用。

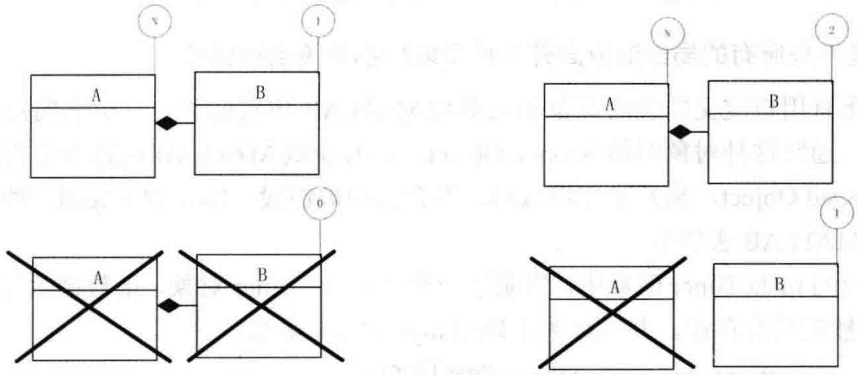


图 3.23 如果 A 和 B 对象是组合结构

同样的道理也适用于环状的情况。如图3.24左所示，三个类的对象相互包含形成环状，并且没有 B，C 对象上没有外部引用，那么对 A 对象的销毁会导致 B 对象的引用计数变为零，B 对象的 delete 方法被调用。同理，C 对象的 delete 方法也会被调用，结果如图3.24右所示。

如果 B 对象上恰好有一个外部引用，即程序还有其他地方要使用 B 的对象，这就是说，B 对象上的引用计数大于 1，则对 A 对象使用 delete 方法，将导致 B 对象的引用计数减 1，但 B 对象的 delete 方法不会被调用，C 的引用计数不变，结果如图3.25右所示。

①组合关系就是强烈的相互依赖，实例见第2.7.4小节。

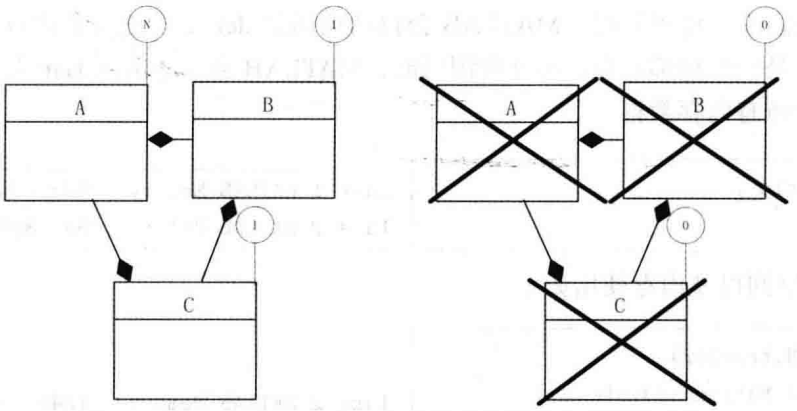


图 3.24 A、B 和 C 对象是组合结构并且没有外部的引用

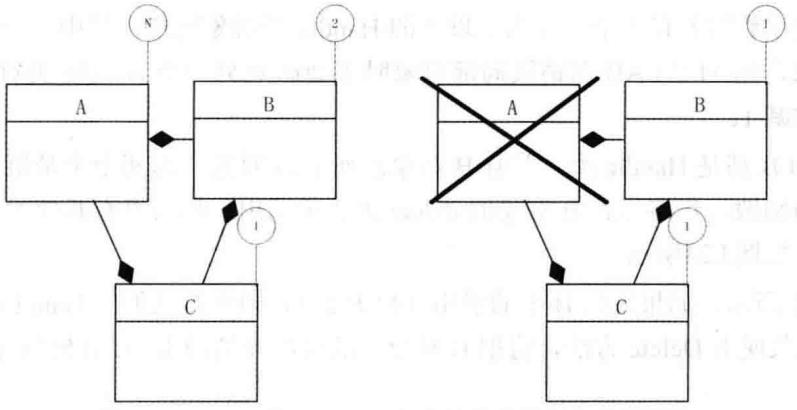


图 3.25 A、B 和 C 对象是组合结构并且存在外部引用

问题： **是不是所有的局部对象离开了作用域后都会被自动销毁**

回答：所有用户定义的类的对象和大多数 MATLAB 内置的类定义出来的对象，都具有这样的行为，通常这种对象叫做 Scoped Object。但有少数 MATLAB 内置类定义的对象，叫做 User-Managed Object，离开了作用域后，不会自动被销毁。User-Managed，顾名思义，需要用户指示 MATLAB 去销毁。

比如下面的 makeTimer 函数中，声明了一个“局部”timer 对象，函数调用完成之后，该 timer 对象仍然在后台存在，并以频率 1 调用 mycallback 函数：

makeTimer

```
function makeTimer()
  t = timer('TimerFcn',@mycallback, 'Period',
        1.0,'ExecutionMode','fixedSpacing');
  start(obj.t)
end
```

对于 User-Managed Object，需要知道两点：首先，因为这种对象有延长的生存期，程序中需要这种行为的对象，否则，每当使用 timer 时，就要把它们声明成全局对象，不方便；其次，MATLAB 提供了该种对象的全局查找和销毁的方法，比如下面的程序演示，在整个 MATLAB 中查找所有 timer 对象，停止计时，并且销毁它们。

```
timers = timerfind();
stop(timers);
delete(timers);
clear timers;
```

MATLAB 中，类似 Timer 这样的 User-managed Object 还有 analoginput 和 videoinput 等。更普遍的，如果 MATLAB 为该类提供了 XXXfind 方法（如 timerfine，daqfind），那么有可能该对象是 user-Managed 对象。

3.3.6 出现异常时 delete 函数如何被调用

面向对象的程序出现异常时，直到 MATLAB 捕获异常之前，所有在try catch 所处的函数的堆栈之上的，已经声明的对象，它们的析构函数都会自动调用。我们用下面的代码来具体说明。假设有 4 个相似的 Handle 类，我们在它们的构造函数和析构函数中用不同的 disp 语句来标记它们的调用时间。

A

```
classdef A < handle
    methods
        function obj = A()
            disp('A ctor called')
        end
        function delete(obj)
            disp('A destructor called');
        end
    end
end
```

B

```
classdef B < handle
    methods
        function obj = B()
            disp('B ctor called')
        end
        function delete(obj)
            disp('B destructor called');
        end
    end
end
```

C

```
classdef C < handle
    methods
        function obj = C()
            disp('C ctor called')
        end
        function delete(obj)
            disp('C destructor called');
        end
    end
end
```

D

```
classdef D < handle
    methods
        function obj = D()
            disp('D ctor called')
        end
        function delete(obj)
            disp('D destructor called');
        end
    end
end
```

我们在下段脚本中，放置了一段 try catch 代码。在 try 之前声明一个 A 类的对象，并且在 try 之后，funcTop 函数之前，声明了一个 B 对象：

Script

```
objA = A();
try
    objB =  B();
```

```
    funcTop();
catch expObj
    % some handling

end
```

在 funcTop 内部，先声明一个局部 C 类的对象 objC，然后调用了 funcBottom 函数，在最底层的 functionBottom 中，先声明了一个局部对象 objD，再故意抛出一个异常。

```
1 function funcTop()
2     objC = C();
3     funcBottom();
4     disp('will not reach here')
5
6 end
```

```
1 function funcBottom()
2     objD = D();
3     expObj = MException('id:Test','msg');
4     throw(expObj);
5     disp('will not reach here')
6 end
```

在命令上执行该脚本，输出如下：首先按先后顺序声明 A，B，C，D 对象，在 funcBottom 函数中的第 4 行抛出一个异常，此时 MATLAB 将不再执行 funcBottom 的第 5 行，向前回溯，发现 funcBottom 中声明了一个局部对象 objD，于是调用其析构函数，释放之，再继续向前回溯，但是在 functionBottom 中没有找到 catch block，于是退出 funcBottom 函数，向上一层，回到 funcTop 函数的第 4 行。同理，没有发现 try catch block，跳过第 4 行，调用 objC 的析构函数，继续向上回溯，退到 main script，继续向前回溯到 main 脚本，发现 try catch，进入 catch block，程序捕获异常。

Command Line

```
A ctor called
B ctor called
C ctor called
D ctor called
D destructor called
C destructor called
```

整个过程如图3.26所示，最后工作空间中还剩下的是 objA 和 objB。

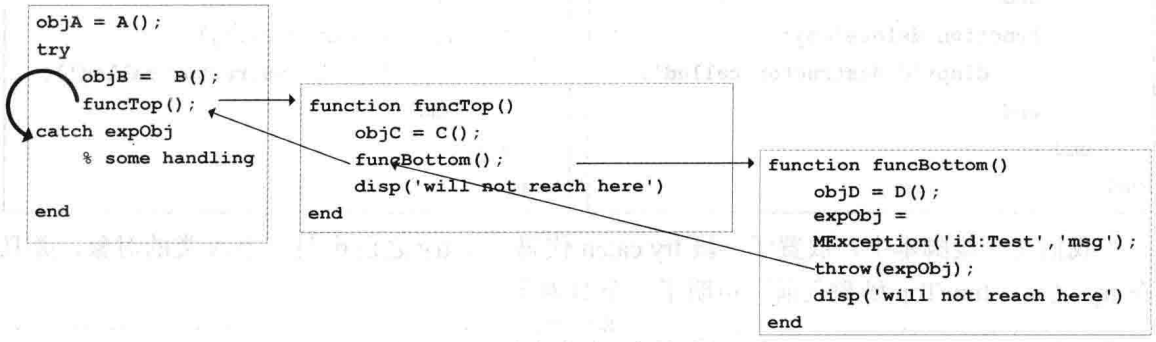


图 3.26 异常发生时对象的销毁顺序

3.3.7 何时用户需要自己定义一个 delete 方法

从用户的角度来说，当类的对象占用了一些系统资源，而无法自动释放时，需要自己重载一个 delete 方法来释放这些系统资源。比如下面的 Value 类和 Handle 类中，Constructor 中都打开了一个文件句柄，那么我们需要在 delete 方法中显式地关闭这个文件句柄。

AVal

```
classdef AVal
    properties
        fID = fopen('file.txt') ;
    end
    methods
        function delete(obj)
            fclose(obj.fID)
        end
    end
end
```

BHandle

```
classdef BHandle < handle
    properties
        fID = fopen('file.txt') ;
    end
    methods
        function delete(obj)
            fclose(obj.fID)
        end
    end
end
```

对于普通的属性，MATLAB 在销毁对象时会自动释放这些属性所占用的内存，没有必要专门定义个 delete 方法去删除对象中的某个属性。

1. Value 类没有析构函数

对于 Vaule 类而言，Value 类的 delete 方法只是用户自己定义的一个方法，不是 MATLAB 承认的合法的析构函数，所以该函数的名字其实是任意命名的，可以叫做 delete，也可以叫做 closeFileHandle。这个方法的存在，是提醒用户不要忘记释放资源。因为在 Value 类对象离开其作用域时，它的 delete 方法因为不是析构函数，MATLAB 不会自动调用它，所以用户要显式（Explicitly）地调用它。

Command Line

```
>> valObj = AVal();
>> ......
>> valObj.delete();    用户需要显式地调用该方法
>> ......
```

2. MATLAB 会自动调用 Handle 类对象的析构函数

对于 Handle 类而言：MATLAB Handle 基类已经提供了基本的 delete 方法的实现，如果用户有额外的需要，必须自己在子类中重载这个 delete 方法。而且如果有可能，还是应该主动地调用 delete 方法；否则 MATLAB 会在 Handle 类对象离开其作用域时，才会自动调用 clear 函数，从而触发自动调用 delete 方法。

这里，BHandle 类用户提供的 delete 方法，没有显式地调用父类 Handle 提供的 delete 方法。为了保证内存资源的释放，MATLAB 会在 fclose 代码执行完毕之后，在内部自动帮助用户去调用 Handle 父类的 delete 方法，释放对象所占用的资源。所以，在 MATLAB 内部，用户自定义的 delete 方法看上去是这样的。

BHandle.m

```
classdef BHandle < handle
```

```
    properties
        fID = fopen('file.txt') ;
    end
    methods
        function delete(obj)
            fclose(obj.fID)
            delete@handle();   % MATLAB 隐式地调用了父类 delete 方法
        end
    end
end
```

对象析构的顺序将在第8章中进行详细分析，这里只需要记住：MATLAB 通过在后台帮用户完成一系列的工作来释放内存资源。

3. 什么是 Handle 类的合法析构函数

MATLAB 自动调用用户重载的 delete 方法的前提条件是：用户定义了满足语法规则且合法的析构函数。这样，MATLAB 才能在众方法中找到它。

一个合法的析构函数，必须具有以下几点：

- 方法的名字叫做 delete。
- 方法没有返回值。
- 方法只接收一个参数（参数不能是 varargin），且该参数必须是 obj，即对象本身。
- 方法不允许是 Sealed，或者 Static，或者 Abstract 的，但 delete 方法可以是 private。这样仅限制不能在类的外部显式地直接调用 delete 方法。

如果用户定义的 delete 函数没有满足上述中的任意一点，用户仍然可以显式地调用该函数，只是 MATLAB 不会自动调用它而已。

4. MATLAB 类方法的种类

根据功能来分类，MATLAB 的类方法可以分成表3.1中所列的几种。到本章结束，我们已经介绍了其中的一部分，其余的方法将在以后的章节一一介绍。

表 3.1 MATLAB 类的方法

类 方 法	功 能	章 节
普通方法 Ordinary Method	作用在对象和对象数组上的一般操作	2.4
构造函数 Constructor	构造新的对象	2.5
析构函数 Destructor	对象销毁时的资源释放	3.3
属性方法 Setter Getter	提供属性访问的中间层	2.8
静态方法 Static Method	提供和类相关的操作，其调用不需要对象	9.2
抽象方法 Abstract Method	提供接口和规范	10.1
转换方法 Conversion Method	提供不同类对象之间的转换	11.5

第 4 章　事件和响应

4.1　事件（Event）

4.1.1　什么是事件

事件（Event）泛指对象内部状态的改变。MATLAB 中，GUI 编程经常使用事件机制。比如，GUI 中一个按钮被按下，就是一个事件，并且 Button 对象状态改变; 再比如，MATLAB 从硬件处采集数据，并且把采集来的数据存在对象的内部，每一次的采集，也是一个事件，并且内部数据的状态就发生一次改变。通常，事件的发生会触发一些响应。比如，在 MATLAB Graphics 中，选择 Zoom 功能进行缩放，x 和 y 坐标轴也会跟着做相应的变化，这些变化就是响应；再比如，要在 MATLAB 中展示实时采集来的数据，每次数据的变化，将触发图像的更新，这也是一种响应（GUI update），如图4.1所示。在事件发生和触发响应这样的模型中，通常把改变内部状态的对象叫做发布者，而把监听事件并做出响应的对象叫做观察者。利用 MATLAB OOP，用户可以自己定义类的事件。一个发布者可以拥有多个事件；一个观察者可以监听多个事件。

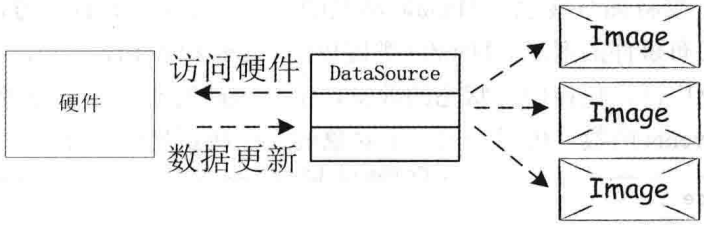


图 4.1　数据采集，数据改变（事件），数据重新显示（响应）

事件和响应一般用来在对象之间相互传递信息，因为其应用广泛，所以 MATLAB 在句柄基类内部就已经实现了这个功能。这就是说，任何用户定义的 Handle 类已经继承了 Handle 基类中与事件有关的功能。如果任意定义一个简单的 Handle 类 DataSource 如下，再查看该类所支持的方法，会发现其中有两个方法，分别是 addlistener 和 notify 方法，它们就是 Handle 基类中和事件机制相关的方法。

A.m

```
classdef DataSource < handle
    % 空类
end
```

Command Line

```
>> obj = DataSource()
obj =
  DataSource handle with no properties.
  Methods, Events, Superclasses
>>
>> methods(obj)
Methods for class DataSource:
```

```
DataSource   delete      findobj     ge          isvalid     lt          notify
addlistener  eq          findprop    gt          le          ne
```

4.1.2 如何定义事件和监听事件

MATLAB 规定，事件的定义要放在 event block 中。下面的代码给 DataSource 类定义了一个 event: dataChanged。

DataSource

```
classdef DataSource < handle
    ......
    events          %  Event Block 开始
        dataChanged
    end             %  Event Block 结束
    ......
    function internalDataChange(obj)
        obj.notify('dataChanged');  %  通知数据改变，  各个 GUI update
    end

end
```

于是该 DataSource 类将拥有继承自 Handle 基类的 notify 方法，并且该方法的作用是监视其数据变化的对象发布事件的消息。Handle 类提供的 event 和 notify 机制，能够处理的情况比 GUI 控件的事件更广泛，任何内部状态的改变，都可以触发事件，发布者只需要调用 notify 函数即可。使用 events 函数，可以列出一个对象的类中所定义的事件。

Command Line

```
>> obj = DataSource ;
>> events(obj)

Events for class DataSource:

    dataChanged
    ObjectBeingDestroyed
```

Handle 基类还提供另一个方法 addlistener。该方法用来在发布者处登记观察者，因为一个发布者可以拥有多个事件，所以登记监听者时，还要指定要监听的事件的名称。为了统一接口，MATLAB 统一地构造新的 listener 对象，在构造观察者时，用户只需要提供事件发生的响应函数即可。这就是说，如果一个观察者想监听事件，实际被登记的不是该观察者，而是该观察者的响应函数。这样一来，普通函数也可以在发布者处登记，这进一步提高了程序的灵活性。

方法 addlistener 用来构造监听者，登记的响应函数可以是普通函数：

```
lh = addlistener(eventObject,'EventName',@functionName)
```

也可以是类的成员方法：

```
lh = addlistener(eventObject,'EventName',@Obj.methodName)
```

还可以是一个类的静态方法[①]：

```
lh = addlistener(eventObject,'EventName',@ClassName.methodName)
```

注意：MATLAB 固定了第三个参数，即响应函数的接口，规定该函数局部所指向的函数必须至少接受两个输入，第一个参数是发布者对象（src）；第二个参数是事件的数据（eventdata），其本身也是一个对象，用户可以自己定义这个对象的类，以定制向监听者传递的数据，这将在第4.2节详述。所谓“如果一个观察者想监听事件，实际登记的不是该观察者，而是该观察者的响应函数”的意思其实是：MATLAB 使用 addlistener 方法在发布者和观察者之间建立一个中间层，发布者只接受由 addlistener 方法构造出来的 Listener 对象在其处注册，真正的观察者只需要把自己的响应函数提供给 addlistener 方法，addlistener 方法将把这个响应函数的句柄包装在其构造的对象的内部。这样就实现了用户定义的 Handle 类和具体的各种观察者之间的解耦合，可用大致的 UML 示意图表示（见图4.2），通过这种间接的方式，实现了 Publisher 类和 Observer 之间的解耦合。

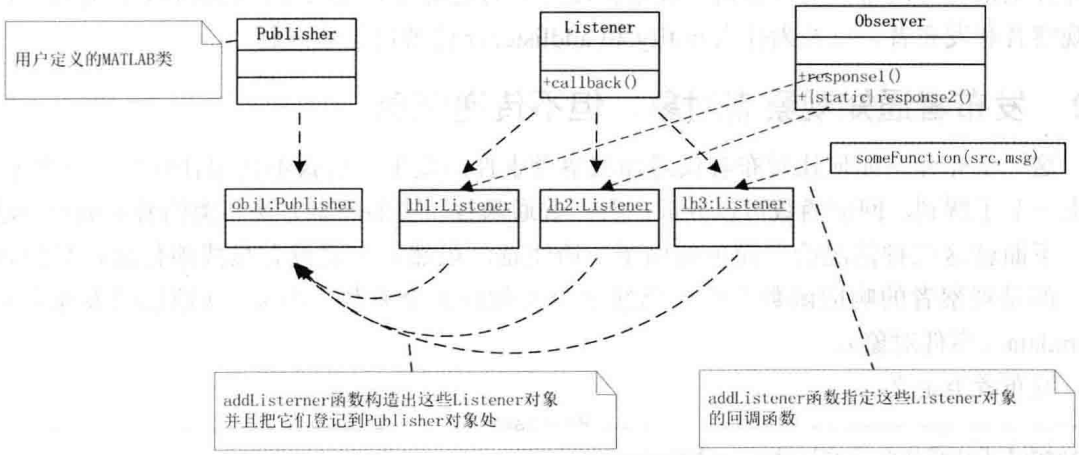


图 4.2 addlistener 统一构造监听者

4.1.3 为什么需要事件机制

如果假设 MATLAB 没有提供 event 和 notify 机制，并且希望上述例子中，DataSource 类的对象在数据改变之后可以通知其观察者对象，那么该类的设计至少需要做到以下几点：

- □ DataSource 要知道，哪些对象需要得到数据更新的通知。
- □ DataSource 还要知道，这些监听对象的响应函数名字叫什么，并且一一调用它们。

DataSource

```
classdef DataSource < handle
  ......
    function broadcastDataChanged(obj,observerObj1,observer2)
        % 数据发生变化
```

①参见第9.2节。

```
        someFunction();                 % 调用普通响应函数
        observerObj1.response();        % 调用观察者 1 的响应函数
        observerObj2.update();          % 调用观察者 2 的响应函数
    end
  ......
end
```

这种设计存在以下几个问题：

- □ DataSource 对象和观察者以及回调函数耦合得过于紧密，在 broadcastDataChanged 方法中，响应函数 response 和 update 是“hard code”，程序显得僵化。
- □ 如果将来要添加或者删除观察者，就要修改 DataSource 内部的实现，因此 DataSource 对修改不封闭，对扩展不开放，其主要原因还是这样做使 DataSource 和观察者之间的耦合过于紧密。
- □ 如果有 100 个观察者，那么在 broadcastDataChanged 中就要调用 100 个响应函数。显然，这样是不方便的。

这里遇到的问题正是 notify 和 addlistener 设计的出发点。在第17.1 节，我们会从这样的设计开始来思考发布者类该如何一般来管理内部的观察者对象以及它们的响应，并且如何解耦观察者和发布者，以及为什么 notify 和 addlistener 能解决上述问题。

4.2 发布者通知观察者对象，但不传递消息

这一节先介绍如何让发布者仅通知观察者事件的发生，而暂不传递任何数据给观察者。从上一节了解到，回调函数可以分成三类：普通函数、类的静态方法和类的普通成员方法。

下面就这三种情况给出简单的例子。请注意，虽然发布者没有显式地传递信息给观察者，但是观察者的响应函数声明中仍然至少要包括两个参数，即 src （消息的发布者）和 eventdata（事件对象）。

发布者类定义：

Publiser

```
classdef DataSourcePublisher < handle
    events
        dataChangedSimple                   % 定义一个简单事件
    end
end
```

如果观察者的响应函数是普通函数：

```
function updateViewSimpleFunc(scr,data)
    disp('updateViewSimpleFunc notified');      % 观察者的回调函数是普通函数
end
```

如果观察者的响应函数是静态方法或普通成员方法：

Observer

```
classdef Observer < handle
```

```
    methods(Static)
        function updateViewStatic(scr,data)      % 观察者的回调函数是静态方法
           disp('updateViewStatic notified') ;
        end
    end

    methods
        function updateView(obj,scr,data)        % 观察者的回调函数普通成员方法
             disp('updateView notified');
        end
    end
end
```

Script

```
p = DataSourcePublisher();
o = Observer();

p.addlistener('dataChangedSimple',@updateViewSimpleFunc);      % 提供回调函数构造 listener
p.addlistener('dataChangedSimple',@Observer.updateViewStatic);
p.addlistener('dataChangedSimple',@o.updateView);

..... % 经过一段时间数据发生变化
p.notify('dataChangedSimple');                                 % 发布
```

通过命令行结果证实，DataSource 对象确实通知了观察者，回调函数确实被调用：

Command Line

```
updateViewSimpleFunc notified
updateViewStatic notified
updateView notified
```

图4.3所示是注册监听者、观察者通知事件发生的顺序图。

MATLAB 面向对象系统对 event 的通知顺序并没有明确的规定。所以，为了保证程序能够在当前 MATLAB 版本下，以及将来的任何版本下运行都得到一致的结果，在使用 event-notify 机制时，应该尽量不要让程序对 Listerner 被通知的顺序有任何的依赖。

4.3 发布者通知观察者，并且传递消息

有时，发布者除了通知观察者，还需要向观察者发送一些数据，所以 MATLAB 还允许用户自定义一个消息类来定制要传递的信息，并且该消息类必须继承自 event.EventData 基类，如图 4.4 所示。比如下面这个自定义的 TimeStamp 类，负责构造一个时间戳对象，并把该对象传递给观察者，告知观察者数据改变的时间。

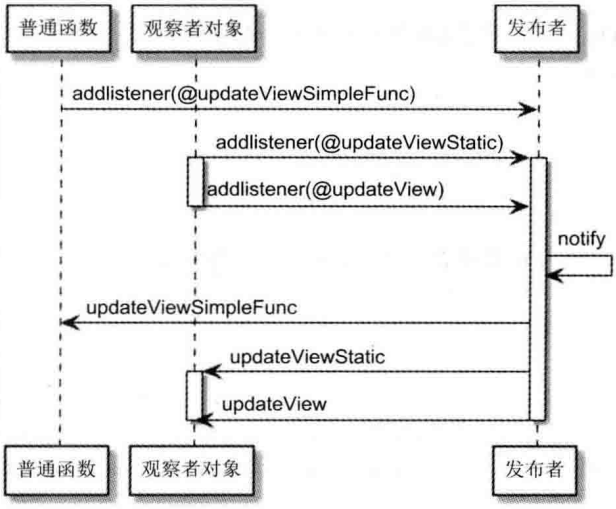


图 4.3 UML 序列图

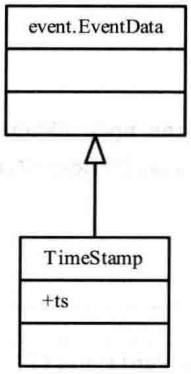


图 4.4 自定义事件类须继承自 event.EventData

该消息类把数据改变的时刻传递给观察者：

Message

```
classdef TimeStamp < event.EventData
    properties
        ts    % 时间戳对象内部封装当前时间信息
    end
    methods
        function obj = TimeStamp()
            tempTime=clock;
            obj.ts = ['(' num2str(tempTime(4),'%02.0f') ':' ...
                          num2str(tempTime(5),'%02.0f') ':' ...
                          num2str(tempTime(6),'%02.0f') ')'];
        end
    end
end
```

发布者发布数据改变的消息和一个消息对象：

Publisher

```
classdef DataSourcePublisher < handle
    events
        dataChanged
    end
    methods
    % 数据改变发生，发布者通知各个观察者，并且传递一个 TimeStamp 对象
        function queryData(obj)% query the hardware , data changed
            obj.notify('dataChanged',TimeStamp());
        end
    end
end
```

如果观察者的回调函数是普通函数，则该函数的定义中，第一个参数应该是消息源，第二个参数应该是源传来的消息：

foo

```
function updateViewSimpleFunc(scr,data)
    disp(['data changed at ',data.ts]);
end
```

如果观察者的回调函数是类的静态方法或普通成员方法，方法的定义如下：

Listener

```
classdef Observer < handle
   methods(Static)
       function updateViewStatic(scr,data)
           disp(['data changed at ',data.ts]);
       end
   end

   methods
       function updateView(obj,scr,data)
            disp(['data changed at ',data.ts]);
       end
   end
end
```

命令行下测试：

首先各个观察者在发布对象处进行注册：

Script

```
clear all ; clc;

p = DataSourcePublisher();
o = Observer();

p.addlistener('dataChanged',@updateViewSimpleFunc);
p.addlistener('dataChanged',@Observer.updateViewStatic);
p.addlistener('dataChanged',@o.updateView)

p.queryData();
```

通过命令行结果证实，观察者回调函数被执行，消息被正确传递：

Command Line

```
data changed at (11:17:38)
data changed at (11:17:38)
data changed at (11:17:38)
```

因为event.EventData 类是 Handle 类，所以任何用户定义的事件类本身也是 Handle 类，因此 event 还可以用来传递大型数据。举个例子，假设要传递一个 500×500 的矩阵作为消息，并且有 10 个观察者在发布者处注册，notify 所有观察者的成本仅是构造一个 Message 对

象，并且传递 10 次 Handle。因为 500 × 500 的矩阵只存在一个拷贝，且被 10 个观察者共享。

```
classdef Message < event.EventData
    properties
        matrix
    end
    methods
        function obj = Message(internalData)
           obj.matrix = internalData ;  % 假设 internalData 是一个 500X500 的矩阵
        end
    end
end
```

```
function queryData(obj)
    % 数据发生改变
    msgObj = Message(obj.internalData) ;
    obj.notify('dataChanged',msgObj);
end
```

不过，这样的设计还是涉及要构造一个内部数据对象的拷贝，把它放到消息子类的 matrix 属性中去。其他更节省空间的方法是：提供一个内部数据的公共接口给观察者，具体实现可参照第17.1节。

4.4 删除 listener

因为 listerner 对象本身是 Handle 对象，所以只要在创建 listener 时记得保存了其 Handle，在删除（注销）一个 listner 时，只需要调用 Handle 基类提供的 delete 方法即可。比如：

Command Line

```
>> lh = addlistener(eventObj,'EventName',@functionName) ;
>> delete(lh);
```

其中，lh 是 addlistener 返回的 Handle 对象。

第 5 章　MATLAB 类文件的组织结构

5.1　如何使用其他文件夹中的类的定义

到目前为止，我们只介绍了一种简单的定义类的方式，即把类的定义和方法的定义都包括在一个同类同名的.m 文件中。

Point.m

```
classdef Point < handle
   properties(Access = private)
       x
       y
   end
   methods
       function obj = Point(x,y)
           obj.x = x ;
           obj.y = y ;
       end
   end
end
```

在该类当前的目录下，该类的定义是可见的，可以直接声明该类的对象：

Script

```
>> p1 = Point(1.0 ,1.0 );
```

如果想在其他路径上使用该类，就要用 addpath 命令，把包含该类的文件夹加到当前 MATLAB 搜索路径中去，比如 Point.m 保存在Z:\folder1\folder2 目录中；而下面的 script 在其他目录中，如果要想声明 Point 类对象，就要使用 addpath：

Script

```
addpath('Z:\folder1\folder2');
p1 = Point(1.0 ,1.0 );
```

addpath 函数除了接受绝对路径，还接受相对路径作为参数，如图 5.1所示。

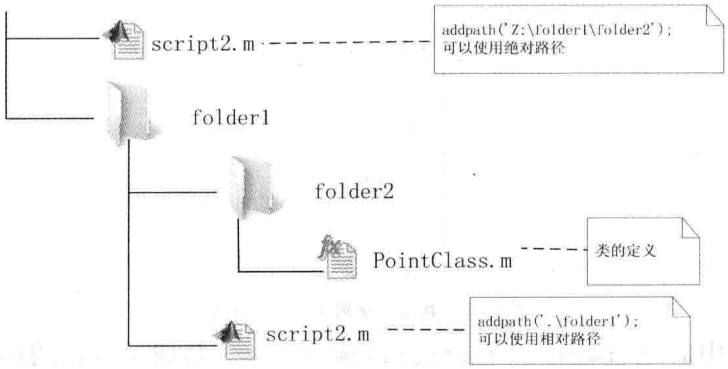


图 5.1　用文件夹和 addpath 来管理各个类

5.2 如何把类的定义和成员方法的定义分开

MATLAB 还支持另一种定义类的方法，适合类成员方法较多的情况，即在类的定义文件中仅仅提供方法的声明（Declaration），而不提供方法定义（Definition），即把方法的定义放到另一个独立的.m 文件中去。把类的定义和成员方法的定义分开，有利于开发更复杂的面向对象程序。这样，类的定义文件就不会扩大到难以管理的程度，如果一个项目是多人同时在开发和维护时，把成员方法的定义分散到单独的文件中，也有利于团队代码的版本管理。

为简单起见，我们还是用 Point2D 的例子。MATLAB 规定，如果要把方法的定义 normalize 和 disp 放在单独的文件中，那么类的定义 Point.m、normalized.m 和 display.m 必须都放在一个以 @ 开头的文件夹中，且该文件夹就必须命名为@Point。

Point

```
classdef Point < handle
   properties(Access = private)
       x
       y
   end
   methods
       function obj = Point(x,y)
           obj.x = x ;
           obj.y = y ;
       end
       % 类定义中必须包括方法的声明
       [ norm ] = normalize(obj);
       display(obj);

   end
end
```

normalize 方法的定义

```
function [ norm ] = normalize(obj)

    norm = sqrt(obj.x^2 + obj.y^2) ;
    obj.x = obj.x / norm ;
    obj.y = obj.y / norm ;



end
```

display 方法的定义

```
function display(obj)

    disp(['x = ',num2str(obj.x)]);
    disp(['y = ',num2str(obj.y)]);

end
```

类方法 display 和 normalize 的定义从外观上看和普通的函数很像，其实质区别在于这些类方法可以访问对象的私有数据，而普通的类方法不可以。图5.2所示为 Point 文件夹中的内容。

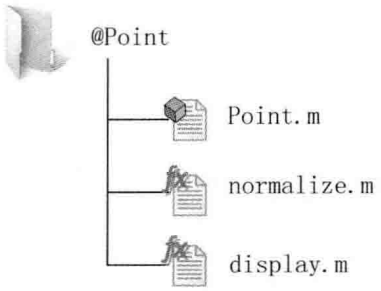


图 5.2 Point 文件夹中的内容

Point.m 文件中的 `[ norm ] = normalize(obj) ;` 是成员方法的声明。该声明告诉 MATLAB 解释器，这个方法的参数列表在哪里可以找到。该声明中指明了该方法有 1 个返

回值，该方法最多接收且只接收一个参数。MATLAB 规定，normalize 方法的声明仍要放在在 methods block 中，并且成员方法声明前面不需要 function 关键词。

问题： **哪些方法的定义一定要放在类定义中**

回答：并不是所有的成员方法都允许把定义和声明分开。以下方法的定义必须放在类的定义中：

- □ MATLAB 规定类的 Constructor 和 delete 方法的定义必须放在类定义中。
- □ MATLAB 规定任何属性的 set 和 get 方法的定义必须放在类定义中。
- □ MATLAB 规定类的 static 方法的定义必须放在类定义中。

一般来说，放在 @ 文件夹中的任何方法都被默认为类的成员方法，甚至不论该方法是否已在类定义中声明。换句话说，即使方法在类的定义中没有被声明，只要该方法被放置在 @ 文件夹中，就算得上是类的方法。当然，我们应该养成一个好的编程习惯，让方法的声明和定义做到一一对应。如果由于用户的疏忽，在类定义中确实没有某方法的声明，那么该方法的属性将使用方法属性的默认值（如非隐藏、public 的访问权限）。

问题： **如何使用 @Point 中类的定义**

回答：如果 main 程序需要使用 PointClass 的类定义，只需要放在和该文件夹同一目录下即可，使用起来和前面类定义中放置全部的方法定义没有任何区别。main 文件和 @PointClass 类文件夹的关系如图5.3所示。

main.m

```
% 在 main script 中使用 Point 类
clear all ; clc ;
p1 = Point(1.0 ,1.0 );
p1   % 这里将调用默认的 display 函数
```

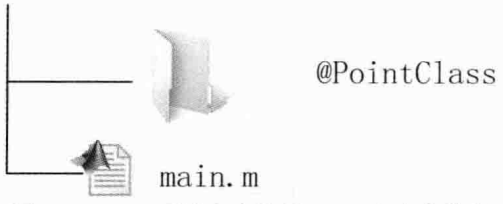


图 5.3 main 可以自由使用 Point 中定义的类

当然，更常见的做法还是用 addpath 函数把该@Point 文件夹添加到 MATLAB 的搜索路径上去，这样就可以在任何路径下使用 PointClass 的类定义了。

5.3 如何定义类的局部函数

在类的定义中，还可以定义局部函数（Local Function）。局部函数不是类的方法，在类的定义外部不可见，不能通过 obj.method 的方式从外部访问。局部函数仅对被类定义内部的方法可见。下面的例子中，给 Point 类定义了一个局部函数 localUtility。按照规定，局部函数的定义，即函数体，要放到 classdef 和 end 的后面。例如：

Point

```
classdef Point < handle
    properties(Access = private)
        x
```

```
        y
    end
    methods
        function obj = Point()
            [obj.x obj.y] = localUtility();  % 调用该文件中的局部函数
            ......
        end
        ......
        ......
    end
end   % classdef block 结束

%    Point 类定义中的局部函数
function  [x y] = localUtility()
......
end
```

从功能上来说，局部函数一般作为工具（Utility）函数存在，提供一些功能，但不足以特殊到要成为一个类的方法。从语法上来说，局部函数对函数参数的要求没有限制，不像类的实例方法那样，参数中一定要包含一个对象。当然，如果有需要，局部函数的参数中可以包括类的对象。局部函数就可以访问对象的私有属性了。

所有在同一文件中定义的类方法都能调用局部函数的方法，但那些只有声明没有定义的类方法除外，如图5.4所示。

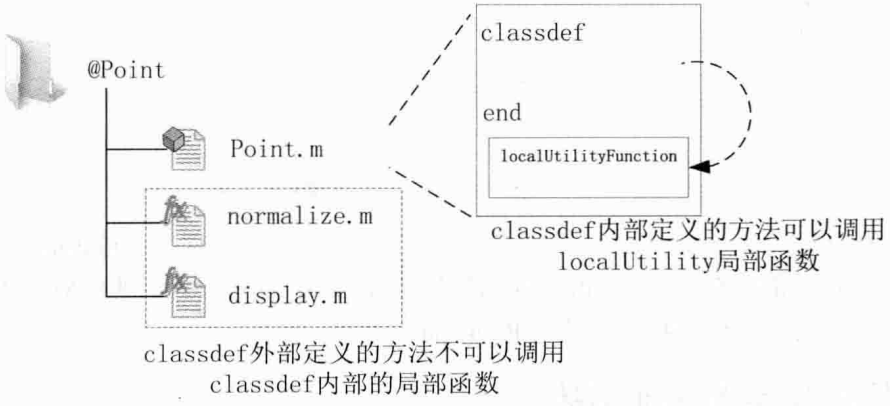


图 5.4 类局部函数的调用规则

在 classdef 外部定义的类的方法也可以拥有自己的局部函数，其调用规则与类的局部函数类似。

5.4 如何使用 Package 文件夹管理类

5.4.1 Package 中的类是如何组织的

如果程序的结构再复杂一些，还可以把各个类再进一步组合成 Package，即形如图5.5所

示的文件结构。

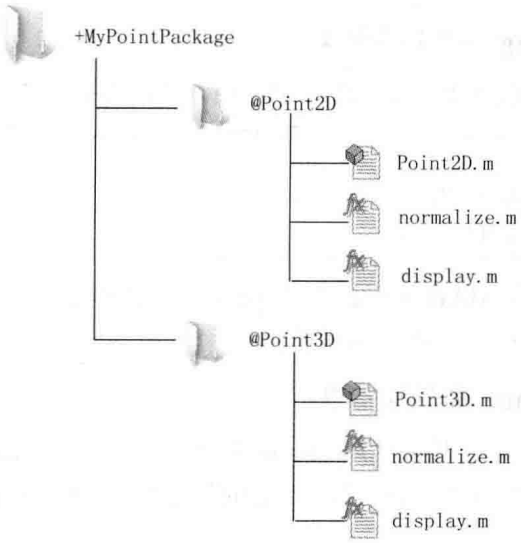


图 5.5 Package 中包含多个类

MATLAB 规定 Package 文件目录必须以加号 + 开头，Package 中还可以包括各个类的文件夹，各类之间还可以有继承关系。比如，其中的 Point3D 继承自 Point2D：

Point2D

```
classdef Point2D <handle
   properties(Access = private)
       x
       y
   end
   methods
       function obj = Point2D(x,y)
           obj.x = x ;
           obj.y = y ;
       end

       [ norm ] = normalize(obj);
       display(obj);
   end
end
```

Point3D

```
classdef Point3D <MyPointPackage.Point2D
  properties(Access = private)
       z
  end
  methods
   function obj = Point3D(x,y,z)
    obj=obj@MyPointPackage.Point2D(x,y);
    obj.z = z;
   end
   function display(obj)
    display@MyPointPackage.Point2D(obj);
     disp(['z = ',num2str(obj.z)]);
    end
   end
end
```

注意，如果要在 Point3D 中使用 Point2D 类，不要忘记在前面加上 Package 的名称。

```
classdef Point3D < MyPointPackage.Point2D        % 凡是使用到基类 要加上 Package 的名称
    ......
     obj = obj@MyPointPackage.Point2D(x,y);
    ......
     display@MyPointPackage.Point2D(obj);
    ......
```

```
end
```

5.4.2 如何使用 Package 中的某个类

如果一个类的定义是放在 Package 中，则使用该类时要在类名前加上 Package 的名称。

Script

```
clear all ; clc;
p1 = MyPointPackage.Point2D(1,1);
p2 = MyPointPackage.Point3D(1,1,1);
```

这种使用 Package 组织 MATLAB 定义的方法，可以让程序的层次更加清楚，在 MATLAB 的工具箱（toolbox）中可以看到许多这样的例子。

5.4.3 如何导入 Package 中的所有类

当然，也可以在程序的一开头就用 import 命令导入整个 Package。这样，调用 Package 中的类就不需要使用 Package 的名称了。下面的例子中， `MyPointPackage.*` 表示导入 Package 中所有的类。

Script

```
clear all ; clc;
import MyPointPackage.*;
p1 = Point2D(1,1);
p2 = Point3D(1,1,1);
p2.display();
```

5.5 函数和类方法重名到底调用谁

假设当前路径上有一个简单的 foo 函数，在`@AClass` 类定义中也有名为 foo 的类方法，如图5.6所示。

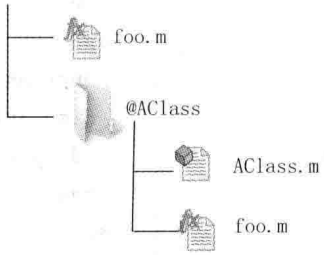


图 5.6 函数 foo 和类方法 foo 不会发生冲突

两个 function 同名，但是定义不同，一个是普通函数，一个是类 AClass 的成员方法。

foo.m 普通函数

```
function [ y ] = foo( x )
    y = x ;
end
```

AClass

```
classdef AClass < handle
    methods
        y = foo(obj,x) ; % 成员方法定义略
    end
   end
end
```

这种情况下，不会出现函数名称冲突的情况，调用哪个 function 将决定于用户程序的调用方法，不同的调用方法具有不同的 signature，MATLAB 将根据 signature 找到匹配的函数或方法，例如：

Script

```
foo(1)              % 调用当前目录上的函数 foo
obj = AClass();
foo(obj,1)          % 调用 AClass 类文件夹中的 foo 成员方法
```

5.6 Package 中的函数和当前路径上的同名函数谁有优先级

MATLAB 的 Package 中还可以放置普通的函数，如果两个普通的函数，有相同的函数名，一个在当前路径下，一个在 Package 中，且它们的函数签名是一样的，在 main 程序中，该函数被调用时，则 MATLAB 将调用路径上优先级最高的函数，如图5.7所示。

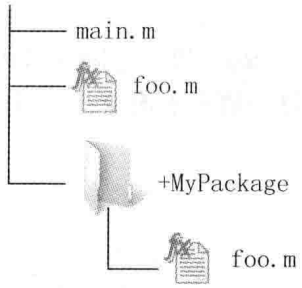


图 5.7 函数 foo 和 Package 函数 foo 谁被调用取决于优先级

默认情况下，最直接路径中的 foo 函数，具有最高的优先级。所谓最直接，可以理解成靠得最近的。也就是说，如果用户在 main 脚本中使用`foo(1)`，MATLAB 将调用当前路径下的 foo 函数，而不是 Package 中的 foo 函数，如果不确定优先级，可以在命令行下输入 which + 函数名来查询：

Script

```
>> which foo
Z:\foo.m
```

如果用户想要调用的是 Package 中的 foo 函数，可以先导入整个 Package。这样，MATLAB 会优先的在所导入的 Package 中寻找匹配的函数：

Script

```
>> import MyPackage.foo ;
>> which foo
Z:\+Package\foo.m  % static method or package function
```

结果显示，导入 Package 之后，MATLAB 路径将优先调用 Package 中的 foo 函数。

第 6 章　MATLAB 对象的保存和载入

6.1　save 和 load 命令

6.1.1　如何 save 和 load object

在 MATLAB 中，对普通变量的 save 和 load 操作，同样也适用于 MATLAB 对象，用户不需要额外操作。下面的命令把对象 obj 中的数据保存到名为 filename 的 MAT 文件中去。

```
>> save filename obj
```

下面的命令把 MAT 文件中的对象 obj 装载到工作空间中：

```
>> load matfilename obj
```

具体来说，save 命令把对象中的数据[①]以二进制的形式保存到 MAT 文件中，load 命令读取 MAT 文件中的数据，然后利用这些数据来构造并且初始化一个新的对象，再放到工作空间中去，如图6.1所示。

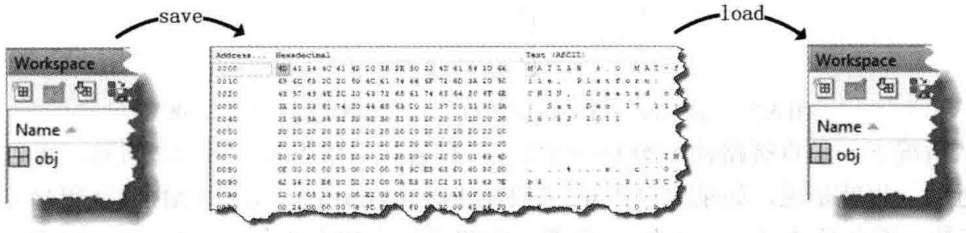


图 6.1　save 和 load 对象的示意图

MAT 文件的格式是公开的，对 MATLAB 和外部程序接口有兴趣的读者可以参考 MAT-File 技术手册，网址为：

http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf

6.1.2　MAT 文件中保存了 object 中的哪些内容

在 save 一个 MATLAB 对象时，MAT 文件中保存了如下内容：

- □ object 所属类的名称和 Package 名称。
- □ object 所属类的属性的默认值[②]。如果 MAT 文件中有多个同类对象，该默认值只要保存一份拷贝，其值将被各对象所共享。
- □ object 中普通属性的值。

1.　save 过程和属性的默认值（Default Value）

MATLAB 在 save 对象的属性的值时，会把该值和其默认值[③]（如果定义了）相比较，如

①其实是把对象转化为一个 MATLAB 的 struct。
②见第2.3.2小节。
③有时也叫做 Factory Default。

果是一样的，则为了节省 MAT 文件的空间，MATLAB 就不会保存该对象的该属性的值。在 load 时，直接把类的默认值赋给对象的属性。所以给类的属性定义默认值是一种节省 MAT 文件空间的方法。

除了节省 MAT 文件的空间，保存类的默认值的另一个作用是保持兼容性。如果 save 时类的某属性的默认值是某个值，而在 load 时该默认值的值发生了变化。为了保证装载对象时，对象能够恢复到其最初的状态，而不是新的状态，MATLAB 的 load 过程会沿用其旧的默认值，而不是使用新的默认值。

下面的例子中，首先使用左边的定义，声明一个对象，再 save；接着修改类定义中 address 的默认值，然后 load。例如：

旧的定义

```
classdef FaceBook < handle
    properties
        name
        address = 'Prime Parkway'
    end
end
```

新的定义

```
classdef FaceBook < handle
    properties
        name
        address = 'Apple Hill'
    end
end
```

Command Line

```
>> clear classes;
>> obj = FaceBook ;
>> obj.name = 'Xiao';
>> obj
obj =
  FaceBook with properties:
       name: 'Xiao'
    address: 'Prime Parkway'

>> save('XiaoFaceBook.mat','obj');
>>
>> clear classes
>> % 这里修改 FaceBook 中 address 的默认值
>>
>> load XiaoFaceBook
>> obj
obj =
  FaceBook with properties:
       name: 'Xiao'
    address: 'Prime Parkway'
```

通过观察结果发现，从 MAT 文件中 load 出的 obj 对象的 address 属性使用的仍然是旧的默认值“Prime Parkway”，这样的设计也是合理的，我们希望在 load 时，对象能够恢复保存时的原始状态。

2. MAT 文件中没有保存的内容

MAT 文件中没有保持类中的以下信息：

- □ 对象的 Transient、Constant 和 Dependent 类型的属性。
- □ 类的完整的定义。

3. 保存 Handle 类对象要注意检查 Handle 的有效性

保存 Handle 类对象时，要注意检查 Handle 的有效性。在第3.3.4节我们提到过，如果对 Handle 对象使用 delete 方法，该对象指向的数据将被释放，但是该 Handle 对象仍然存在，只不过它是一个无效的 Handle 对象。无效的 Handle 对象仍然是一个对象，它也可以被 save 和 load，只不过该对象内部再没有任何的有效数据，如图6.2所示。

ARef.m

```
classdef ARef < handle
    properties
        matrix
    end
end
```

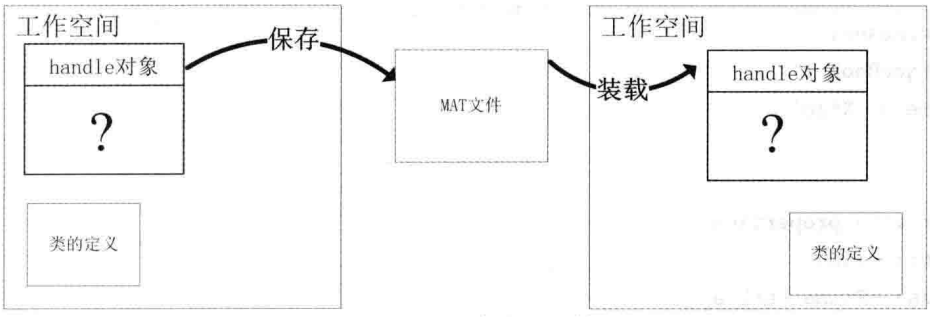


图 6.2 无效的 Handle 对象也可以被 save 和 load

比如：

Command Line

```
>> oRef = ARef                          % 声明一个 Handle 对象
oRef =
  ARef handle
  Properties:
    matrix: [4x4 double]               % Handle 对象中的数据
  Methods, Events, Superclasses
>> oRef.delete()                        % 调用 delete 方法
>> save
>> clear all
>> load
Loading from: matlab.mat
>> oRef
oRef =
  deleted ARef handle                  % 对象中没有任何数据
```

```
  Methods, Events, Superclasses
>> isvalid(oRef)                          %  load 到工作空间中的是一个无效的 Handle 对象
ans =
     0
```

因为 MAT 文件中不能保存类的定义，所以要求 MATLAB 在 load 对象时，必须能找到该对象的类的定义。所以，该类的定义必须在 MATLAB 的搜索路径上，并且该类的定义必须和 save 时的类的定义要保持“一致”，如图6.3所示。

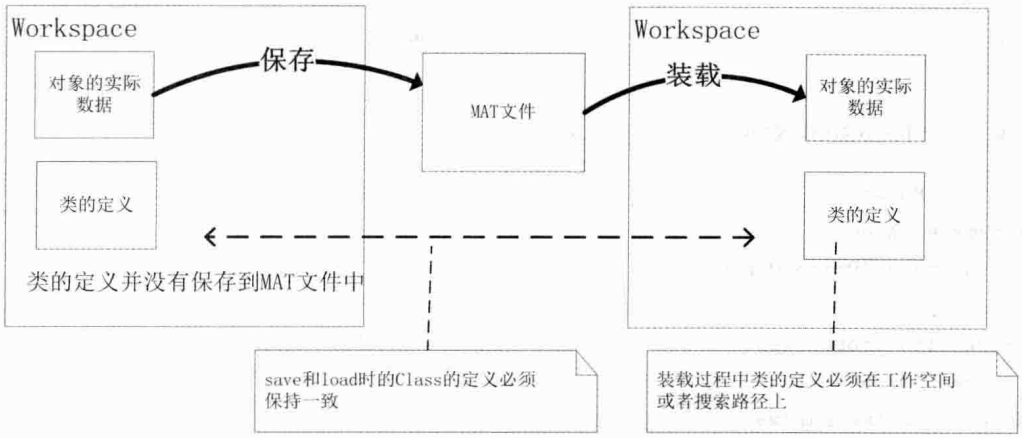


图 6.3 Load 对象时要保证对象的定义在当前工作空间中

在这里，读者可以把“一致”理解成“尽量一致，如果不一致，程序需要付出一定的代价来保证 load 的过程顺利”。这个问题，我们在后节会提到。

如果在 load object 时，MATLAB 找不到类的定义，那么 MATLAB 将不知道如何用 MAT 文件中的数据初始化对象。这时 MATLAB 对象系统将抛出一个异常，但是为了不影响 MAT 文件中其他变量的装载，MATLAB 对象系统会随即捕获该异常，抑制错误信息返回到命令行中，继续装载过程，接着装载 MAT 文件中其余的数据，其实装载对象的过程实际是失败的:

Command Line

```
>> save
>> clear classes          %    清除工作空间中的类的定义
>> load matlab.mat        %    假设类的定义，不在当前目录或者搜索路径中
Warning: Variable 'obj' originally saved as a A
cannot be instantiated as an object and will be read
in as a uint32.           %  实际上是失败的 load
```

上述的 obj 是保存在 matlab.mat 中的一个对象，假设它是 A 类的对象，因为 MATLAB 找不到 A 类的定义，所以 MATLAB 只能进行“简单的转化”。

6.1.3 如果类的定义在 save 之后发生了变化

实际计算和软件开发过程中，要做到 load 时类的定义和 save 时的完全一致是很困难的。因为我们一直都在提到，在开发的一开始，不可能把所有的需求和可能出现的情况都考虑到。这就是说，类的定义不能避免地是在不断变化中的。很有可能出现这种情况，在工程计算和开发过程中，我们积累了一些有用的数据，并且期望这些数据对不久的将来的工程有

用。但是，如果要求在 load 这批数据时，设计到的类的定义完全不变，是不太可能的。如果类的定义在 save 之后发生了变化，我们可以分以下几种情况：

1. 如果属性的名称变了

旧的定义

```
classdef FaceBook < handle
    properties
        name
        address
    end
end
```

新的定义

```
classdef FaceBook < handle
    properties
        Name
        address
    end
end
```

声明一个 FaceBook 对象，并进行简单的赋值：

Command Line

```
>> obj = FaceBook;
>> obj.name = 'Xiao' ;
>> obj.address = 'Prime Parkway'
obj =
  FaceBook with properties:
       name: 'Xiao'
    address: 'Prime Parkway'
>> save('XiaoFaceBook','obj');
```

保存之后，我们修改了 FaceBook 类的定义，把 name 属性的名字改成了首字母大写：

Command Line

```
>> clear classes
>> load XiaoFaceBook
>> obj
obj =
  FaceBook with properties:
       Name: []
    address: 'Prime Parkway'
```

当 load MAT 文件到工作空间时，虽然对象构造出来了，但是 obj 对象的属性 name 装载是失败的，MATLAB 跳过了 Name 属性，继续装载 address 属性。

2. 如果添加了新的属性

旧的定义

```
classdef FaceBook < handle
    properties
        name


    end
end
```

新的定义

```
classdef FaceBook < handle
    properties
        name
        address = 'Prime Parkway'
    end
end
```

声明一个 FaceBook 对象，并进行简单的赋值：

Command Line

```
>> obj = FaceBook;
>> obj.name = 'Xiao' ;
obj =
  FaceBook with properties:
       name: 'Xiao'
>> save('XiaoFaceBook','obj');
>>
>> clear classes
>> % 这里修改类的定义，添加一个 address 属性
>> load XiaoFaceBook
>> obj
obj =
  FaceBook with properties:
       name: 'Xiao'
    address: 'Prime Parkway'
```

当 load MAT 文件到工作空间时，对象被正常装载，并且其中多了一个 address 属性，且值取新定义中的默认值。

3. 如果属性被删除了

旧的定义

```
classdef FaceBook < handle
    properties
        name
        address
    end
end
```

新的定义

```
classdef FaceBook < handle
    properties
        name

    end
end
```

声明一个 FaceBook 对象，并进行简单的赋值：

Command Line

```
>> obj = FaceBook;
>> obj.name = 'Xiao' ;
>> obj.address = 'Prime Parkway'
obj =
  FaceBook with properties:
       name: 'Xiao'
    address: 'Prime Parkway'
>> save('XiaoFaceBook','obj');
>>
>> clear classes
>> % 这里修改类的定义，删除 address 属性
>> load XiaoFaceBook
>> obj
obj =
  FaceBook with properties:
```

```
    name: 'Xiao'
```

当 load MAT 文件到工作空间时，对象被正常装载，但是 address 属性将不再出现在对象中。

4. 如果属性的默认值变了

这种情况在介绍 save 和默认值时已经提及，不再赘述。

6.2 saveobj 和 loadobj 方法

6.2.1 如何定义 saveobj 方法

当用户需要扩展或者定制 save 的行为时，可以在类的定义中重新定义 saveobj 成员方法。一旦提供了该类的 saveobj 方法，那么当对该对象调用 save 命令时：

Command Line

```
>>save filename obj
```

MATLAB 就会调用该类自己的 saveobj 方法来保存对象。比如，下面的类中提供了一个简单的 saveobj 成员方法：

MyClass.m

```
classdef MyClass
    properties
        x
    end
    methods
        function s = saveobj(obj)
           s.x =  obj.x;  % s is a struct
        end
    end
    ..... % loadobj 留到下节定义
end
```

关于 saveobj 方法，有如下要点：

- 该方法的返回值是一个 struct，saveobj 方法把一个 object 转换成了 struct。
- 保存在 MAT 文件中的数据实际上就是 struct，并且该对象的类的名字也保存在 MAT 文件中。这个 struct 中包含了 load 时用来初始化新对象的必要数据，如图6.4所示。

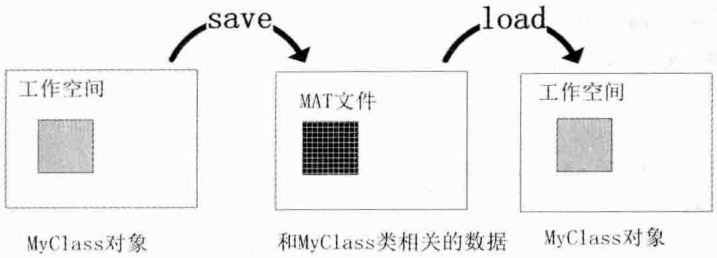


图 6.4 object 在 MAT 文件中保存成 struct

□ struct 中 field 的名称最好和对象中属性的名称保持一致，如上述 saveobj 方法所示，对象中的属性 x 对应 struct 中的 field x 。

6.2.2 如何定义 loadobj 方法

自定义 saveobj 方法的目的是告诉 MATLAB 如何保存对象中的数据。同理，为了保证在 load 过程中，MAT 文件中的数据能够被正确初始化新的对象，用户还需要提供一个配套的 loadobj 方法。该 loadobj 成员方法的参数是一个 struct，返回值必须是一个对象。针对前节的 saveobj 方法，可以这样定义匹配的 loadobj 方法：

```
......
    methods(Static)
        function obj = loadobj(obj)
            if isstruct(obj)                % 如果 obj 是一个 struct
                newobj = MyClass(obj.x);    % 利用 struct 中的信息重新构造一个对象
            end
            obj = newobj;                   % 返回这个新构造的 MyClass 类的对象
        end
    end
......
```

子定义类的 loadobj 方法，有如下要点：

□ loadobj 方法必须被声明成静态方法[①]，因为在调用 loadobj 时，类的对象还没有被建立起来，所以只能是静态的[②]。

□ 在 loadobj 方法内部的工作是提取 struct 中的信息，去构造新的对象。

□ loadobj 和 Constructor 类似，必须返回一个新构成的该类的对象。

当使用 load 命令时，即

Command Line

```
>> load filename obj
```

上述定义的 loadobj 成员方法会被自动调用，如果在 load 对象的过程中还有其他的工作要做，也可以一并放到 loadobj 方法中完成。

saveobj 和 loadobj 在工程科学计算中的一个用例是：首先可以用它们来保存中间计算结果。比如做自洽或者最优化计算，如果计算量大、时间长，只给出一次初始值，然后等很长时间期望其收敛，显然不是最佳的方法，如果计算到一半程序出错，还要从头再来更不方便。这种情况下，应该每隔一段时间（或一定步数的计算之后）对结果做一次 saveobj 保存，下次重新开始时，再使用 loadobj 利用上次的保存结果作为新的初始值，根据收敛的情况，适当调整自洽或者优化参数让计算继续，如图6.5所示。

再比如，我们要遍历大量的数据，单次计算的时间不长，但是计算的次数很多，总体计算时间仍然很长。这种情况下，我们也应该考虑使用 saveobj 把单次计算的结果保存下来，对大量的数据做分段的遍历，具体请参见本书附录B综合实例部分的图B.6。

①关于静态（Static）方法，见第9.2节。
②因为调用类的静态方法不需要对象事先存在。

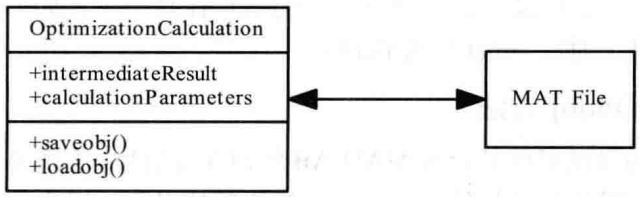


图 6.5 利用 saveobj 和 loadobj 记录计算的中间结果并且重新开始计算

6.3 继承情况下的 saveobj 和 loadobj 方法

6.3.1 存在继承时如何设计 saveobj 方法

当类存在继承结构时，如果要重新定义 saveobj，就需要给父类和子类各提供一个 saveobj 方法。当使用 save 命令时：

Command Line

```
>> save filename obj
```

MATLAB 把子类的 saveobj 方法作为入口，在子类 saveobj 方法体中，将首先调用父类的 saveobj 方法，把父类中的数据保存在结构体中，返回这个结构体，并且回到子类 saveobj 方法中，继续往该结构体中填放子类对象的数据。

MySuper

```
classdef MySuper
    properties
        X
        Y
    end
    methods
        function S = saveobj(obj)
            S.PointX = obj.X;
            S.PointY = obj.Y;
        end
    end
    ......
end
```

MySub

```
classdef MySub < MySuper
    properties
        Z

    end
    methods
        function S = saveobj(obj)
            S = saveobj@MySuper(obj);
            S.PointZ = obj.Z;
        end
    end
    ......
end
```

保存一个对象的过程可以用图6.6表示。

针对上面的设计，也许有的读者会问：把所有“转换 x，y，z 到 S 中”的任务都放到子类的 saveobj 方法中去，这样不是更省事吗？可以从以下三个方面来理解这样的设计：

- 从职责的角度来说：x 和 y 的属性属于 MySuper 的 Base 类，对它们的操作应该由 MySuper 类完成，而不是子类。
- 从类的设计角度来说，MySuper 和 MySub 中的数据以及 saveobj 方法应该独立于对方变化。例如，如果需要在 MySuper 中添加一个属性，只需要修改 MySuper 的 loadobj 即可，不会影响到子类的 saveobj；如果把所有属性的保存都集中到子类中完成，父类和子类就不相互独立了，这也叫做耦合。比如对父类属性的修改会影响到子类的

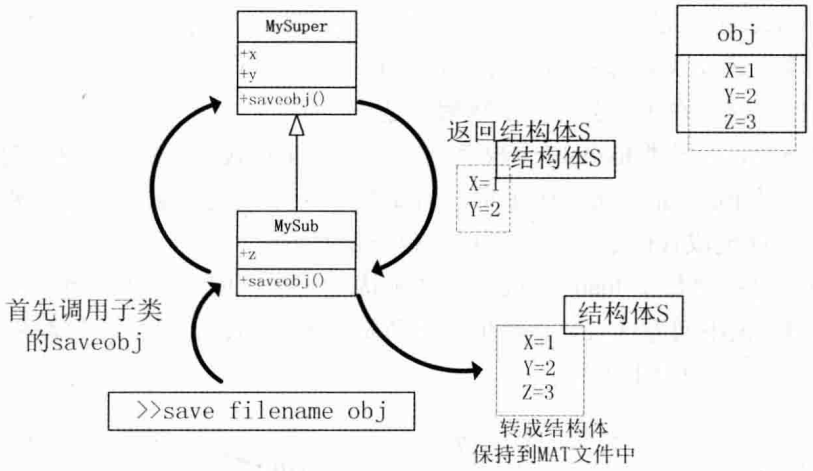


图 6.6 存在继承结构时 save 工作被分散到子类和父类中完成

saveobj 方法。

- 从父类的设计角度来说，提供一个父类的 saveobj 还可以方便父类对象的保存，因为父类也是一个完整的类。这样，父类的对象也可以使用 save 命令。

6.3.2 存在继承时如何设计 loadobj 方法

存在继承结构时，loadobj 方法稍微复杂一些，我们先按着前面 saveobj 的思路，试探性地设计一下子类的 loadobj 方法。该方法要满足下列基本要求：

- 子类的 loadobj 要负责构造一个子类对象。
- 从合理分工的角度来说，x 和 y 数据的装载应该由父类的 loadobj 来实现，z 数据的装载应该由子类的 loadobj 来实现。
- 父类的 loadobj 应该可以独立工作。

我们先试着仿照前面 saveobj 的步骤来建立子类和父类的 loadobj 方法，如图6.7所示。

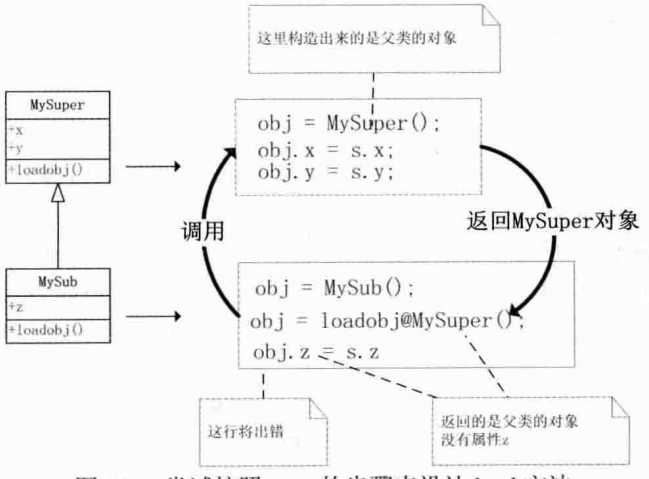


图 6.7 尝试按照 save 的步骤来设计 load 方法

完全模仿 saveobj 方法去设计 loadobj 方法的问题是：当子类中调用父类 loadobj 时，一个 MySuper 对象，即父类对象被创建出来了，并且返回给子类，但是因为这个父类的对象中没有 z 这个属性，这将在子类尝试给 z 赋值时造成出错。

另一个思路是，让子类 loadobj 全权负责对象的创建和数据的赋值，这当然是可以的，但这只能保证子类的 loadobj 方法工作正常，当要 load 一个父类对象时，还需要再定义一个父类的 loadobj，这就造成代码的重复，所以这也不是好方法。

标准的做法是：给每个 loadobj 成员方法都构造一个中间层，即在每个类中都添加一个中间方法 reload。该中间方法只负责赋值，不负责对象的创建。这样，父类和子类的 loadobj 就都可以正常工作了，如图6.8所示。

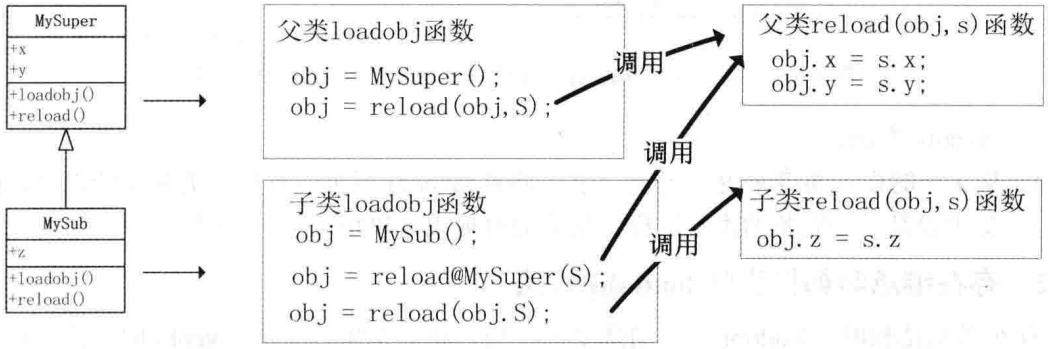


图 6.8 正确的 load 方法：添加一个中间层 reload 方法

按照上述的思路，程序如下：

MySuper

```
classdef MySuper
  ......
  methods
    function obj = reload(obj,S)
        obj.X = S.PointX;
        obj.Y = S.PointY;
    end
  end

  methods (Static)
   function obj = loadobj(S)
        obj = MySuper; % 创建父类对象
        obj = reload(obj,S);

   end
  end
end
```

MySub

```
classdef MySub < MySuper
   ......
   methods
     function obj = reload(obj,S)
       obj.Z = S.PointZ;

     end
    end

    methods (Static)
     function obj = loadobj(S)
        obj = MySub;  % 创建子类对象
        obj = reload@MySuper(obj,S);
        obj = reload(obj,S);
      end
    end
end
```

读者可以自行验证结果。

6.4 什么是瞬态（Transient）属性

我们给类提供 saveobj 方法的主要动机是，对 save 过程进行定制，并且可以控制 MAT 文件的大小。在 saveobj 方法中，可以指定哪些属性是重要结果，需要保存；哪些属性是计算的中间结果或者辅助变量，不需要保存。但通过 saveobj 这个手段，对 save 过程进行定制，要求在 saveobj 中具体写出该保存的那些属性，loadobj 也是同样的道理。容易想象，当类有很多属性时，比如一个类有 200 个属性，其中重要的属性有 100 个，只需要保存它们，在 saveobj 中遍历这 100 个属性就不方便了。即使要保存的属性数目不多，如果类在不断地被扩展的，每次增加一个新的需要保存的属性，都要去修改 saveobj 方法和 loadobj 方法，这样也是不方便的，而且难免在不断修改 saveobj 时出错。

MATLAB 的解决方法是：可以直接给属性添加一个特征修饰词，告诉 save 命令，哪些属性需要保存，哪些属性不需要保存。这个特性修饰词，就叫做瞬态（Transient）。因为凡是定义成了瞬态的属性都不会被 save 命令保存到 MAT 文件中去。有了 Transient 关键词，就可以不用再定义 saveobj 方法了。因此，一个包含有 200 个属性的类定义，可以写成如下的形式，以保证在 save 过程中只保存 100 个重要的属性：

```
classdef Calculation < handle
    properties(Transient)
      ......% 这里定义那 100 个计算中间变量
    end
    properties
      ......% 这个 block 中的属性默认是 Transient=false, 使用 save 时会被保存到 MAT 文件中
    end
    ......
end
```

在 load MAT 文件时，若没有对 Transient 属性做特殊处理，该属性 load 过之后，值为空：

Calculation

```
classdef Calculation < handle
    properties
        results
    end
    properties(Transient)
        intermediateVal
    end
    methods
        function obj = Calculation()
            disp('ctor called');
            obj.intermediateVal = 'disposable';
        end
   end
end
```

命令行测试如下（注意，构造函数仅被调用一次，装载之后 intermediateVal 属性为空）：

Command Line

```
>> obj = Calculation();
ctor called                                  % 构造函数只被调用了一次
>> obj.results='essential'                   % 假设这是要保存的属性的值
obj =
  Calculation handle
  Properties:
          results: 'essential'               % 保存之前两个属性都有具体的值
   intermediateVal: 'disposable'
  Methods, Events, Superclasses
>> save test.mat
>> load test.mat
>> obj
obj =
  Calculation handle
  Properties:
          results: 'essential'
  intermediateVal: []                        % Transient 属性没有被保存 load 为空
  Methods, Events, Superclasses
```

Transient 属性的性质介于 Dependet 属性和 Stored 属性之间，见表6.1。

表 6.1 Transient 属性的性质

属性的种类	是否分配内存	save 时是否被保存到 MAT 文件
Stored 属性	Y	Y
Transient 属性	Y	N
Dependent 属性	N	N

6.5 什么是装载时构造（ConstructOnLoad）

在上述的例子中，Constructor 只被调用了一次。也就是说，在装载 MAT 文件的过程中，默认情况下 MATLAB 不会去调用该类的构造函数。这也是为什么上述瞬态属性 intermediateVal 为空的原因。

调用 constructor 是给属性自动赋值的主要方法之一，如果希望在 load 过程中自动给某些属性赋值，可以使用一个类叫做 ConstructOnLoad 的关键词。MATLAB 规定，如果把一个类声明成ConstructOnLoad = true，那么在包含该类对象的 MAT 文件被加载时，MATLAB 会自动调用该类的缺省的构造函数（这也要求用户必须提供缺省的构造函数），示例如下：

```
classdef(ConstructOnLoad) Calculation < handle   % 该写法相当于 Constructor=true
    properties
        results
    end
    properties(Transient)
```

```
        intermediateVal
    end
    methods
        function obj = Calculation()
            disp('ctor called');
            obj.intermediateVal = 'initialValue';
        end
    end
end
```

命令行测试如下（注意，构造函数在装载时又被调用一次，装载之后 intermediateVal 属性值恢复成 initialValue）：

Command Line

```
>> obj = Calculation();
ctor called                          % 声明对象时 Constructor 被调用一次
>> obj.results = 'essential'
obj =
  Calculation handle
  Properties:
          results: 'essential'
    intermediateVal: 'initialValue'
  Methods, Events, Superclasses
>> save test.mat
>> load test.mat
ctor called                          % 装载对象时 缺省 Constructor 被调用
>> obj
obj =
  Calculation handle

  Properties:
          results: 'essential'
    intermediateVal: 'initialValue'   % intermediateVal 在调用 Constructor 过程中被恢复
  Methods, Events, Superclasses
```

总地来说，Transient 提供的功能好比是轻量级的 saveobj 方法，而 ConstructOnLoad 特性提供轻量级的 loadobj 过程中的控制。这两个特性能够满足大多数的定制 save 和 load 行为的需要，只有在 Transient 和 ConstructOnLoad 无法解决问题时，才需要考虑重载函数 saveobj 和 loadobj。

第 7 章　面向对象的 GUI 编程：分离用户界面和模型

7.1　如何使用 GUIDE 进行 GUI 编程

MATLAB GUI 编程和一般的 MATLAB 编程没有区别，所以也可以使用面向对象的思想。我们先从一个简单例子出发，回顾一下如何使用 GUIDE 进行 GUI 编程，再一步一步过渡到用面向对象来设计 GUI 程序。例子是这样的，假设需要设计一个提款和存款的界面如图7.1所示。

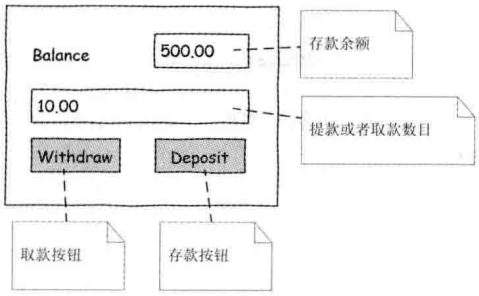


图 7.1　一个简单的银行存款、提款界面

比如，用户已有存款（Balance）500 元，如果在中间的文本框中输入 10 元，然后按下提款键（withdraw），存款将被减 10 元；如果按下存款键（Deposit），存款将增加 10 元。该 GUI 仅有 5 个控件对象，使用 MATLAB GUIDE 立即就可以画出这个界面的布局，如图7.2 所示[①]。

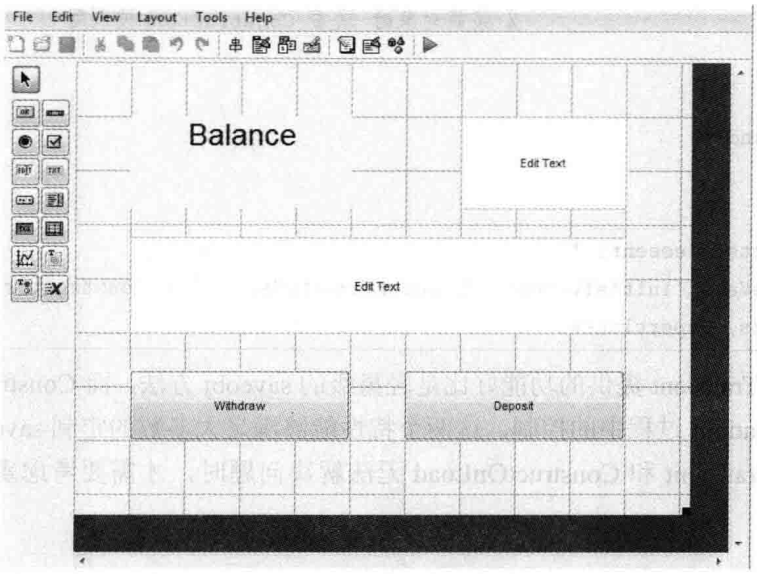


图 7.2　使用 GUIDE 设计的 MATLAB 界面

流程是这样的，GUIDE 会自动帮助用户生成一个主函数，默认的名字为`Guide_GUI`，该函数将包括若干子函数，如`Guide_GUI_OpeningFcn` 函数，在 GUI 变得可见之前该函数被执

①提醒：使用 GUIDE 设计 GUI，须要把设计保存成 FIG 格式的文件。

行，所以用户可以通过修改GUIDE_GUI_OpeningFcn，给 GUI 设置一些初始值。比如，设置两个文本框的初值，将全局数据放到 appdata 中去等。在这个例子中，我们把代表银行存款的 balance 变量放到 appdata 中。

Guide GUI OpeningFcn

```
function Guide_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
    handles.output = hObject;
    set(handles.inputbox,'string','0.00');        %  用户 input box 初值
    set(handles.balancebox,'string','500.00');    %  balance box 初值
    setappdata(handles.output,'balance',500);     %  把 balance 值放到 appdata 中
    guidata(hObject, handles);
```

在 GUIDE 构造出来的一系列子函数中，还包括两个按钮的回调函数，用户只需要在两个回调函数中填充需要执行的操作即可。在下面的 withdraw 按钮的回调函数withdraw_Callback 中做了如下工作：

(1) 得到 input box 中的用户输入，并且转成 double。

(2) 从 appdata 中得到 balance 的值，并且和 input 做减法。

(3) 更新 appdata 中的 balance 的值，最后更新界面上的 balance 的显示。

withdraw 按钮的回调函数

```
function withdraw_Callback(hObject, eventdata, handles)
    input = str2double(get(handles.inputbox,'string'));
    balance = getappdata(handles.output,'balance');
    setappdata(handles.output,'balance',balance - input);
    set(handles.balancebox,'string',num2str(balance - input));
```

存款按钮的回调函数deposit_Callback 和取款的类似：

Deposit 按钮的回调函数

```
function deposit_Callback(hObject, eventdata, handles)
    input = str2double(get(handles.input,'string'));
    balance = getappdata(handles.output,'balance');
    setappdata(handles.output,'balance',balance + input);
    set(handles.balancebox,'string',num2str(balance + input));
```

总地来说，这个 GUI 程序的结构较简单，即在一个 Figure 对象上安置了若干的控件对象。我们主要关心的是两个文本框 input box、balance box 和提款、取款两个按钮。程序中 balance 是一个重要的 GUI 的全局变量，所以把 balance 放到 Figure 对象的 appdata 中，以方便 balance 在各个函数之间的传递和更新。该程序的结构如图7.3和图7.4所示。

整个程序由一个主函数和一个 FIG 文件构成，主函数中的GUIDE_GUI_OpeningFcn 用来初始化控件对象的初值，以及 balance 变量。

主函数中有 GUIDE 自动生成的回调函数的框架，回调函数可以通过 handles 来访问 figure 上的各个控件对象，做必要的计算和更新。

GUIDE 的优点是迅速地构造简单界面；缺点也是明显的，当用户界面复杂到一定程度，尤其是需要多个界面，并且要经常修改时，使用主函数 + 若干子函数，并且用 GUIDE 界面

来布置各个控件对象位置的方法，就显得力不从心了。

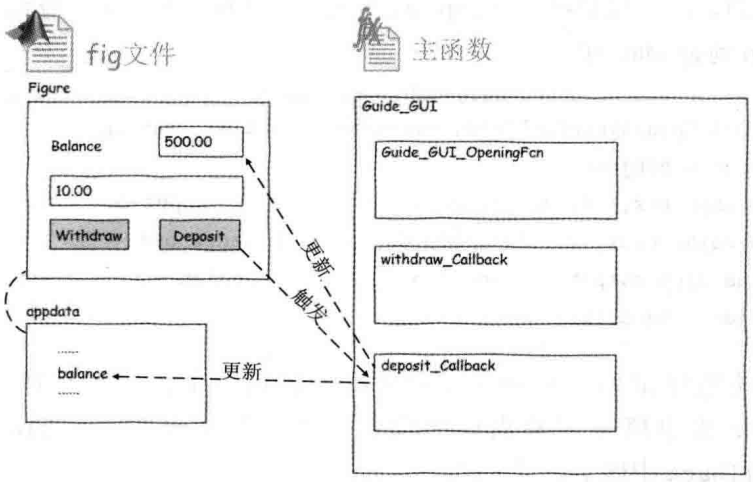


图 7.3　整个 GUI 由一个 FIG 文件和一个主函数构成，主函数中有若干子函数

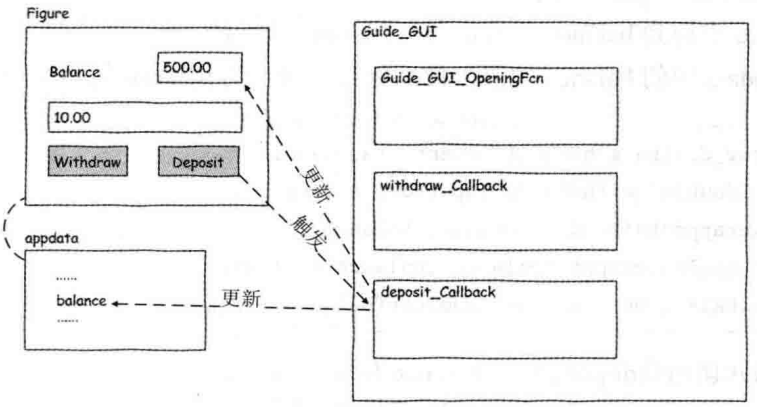


图 7.4　回调函数是子函数，由 GUIDE 自动生成，用户只需要填充内容

7.2　如何使用程序的方式（Programmatic）进行 GUI 编程

这一节我们绕过 GUIDE，介绍如何使用面向过程的方式进行 GUI 编程。这样的方式有更大的自由度，这也是 MATLAB 用户 GUI 编程的一种常见的方式。下面是提款、存款界面的 MATLAB 程序版本，为简单起见，所有代码都放在了一个脚本中。

第一步先构造初始数据：

main.m

```
% 构造初始数据
balance = 500 ;
input   = 0 ;
```

第二步是产生 figure 对象以及各个控件对象。为了能让各个控件在 Figure 上显示出来，各个控件对象的父对象（parent）要设置成 figure。

main.m

```
% 构造初始的 figure 对象和控件对象
hfig =  figure('pos',[100,100,300,300] );
```

```
withdrawButton = uicontrol('parent',hfig,'string','withdraw',...
                           'pos',[60 28 60 28]);
depositButton  = uicontrol('parent',hfig,'string','deposit',...
                           'pos',[180 28 60 28]);
inputBox    = uicontrol('parent',hfig,'style','edit','pos',[60 85 180 28],...
                        'string',num2str(input),'Tag','inputbox');
balanceBox = uicontrol('parent',hfig,'style','edit','pos',[180 142 60 28],...
                       'string',num2str(balance),'Tag','balancebox');
textBox = uicontrol('parent',hfig,'style','text','string','Balance',...
                    'pos',[60 142 60 28]);
```

注意，在产生控件对象时，我们明确指出了它们在 Figure 上的位置。

```
......
                           'pos',[60 28 60 28]);
......
```

也许有读者会说，这是用程序编写 GUI 的弱点，即要用数字精确地定位控件的位置，界面编程还是应该使用 GUIDE。对于这种看法，我们要认识到：首先，用程序的方式组织 GUI 和后面会提到的用面向对象的方式来组织较大规模的程序及其用户界面带来的编程的便利远远大于需要精确给定控件位置所带来的不便；其次，对控件在界面上的布局和控制本身也可以用面向对象的方式去解决，用户可以自己设计一个类，比如叫做 LayoutManager 来自动设置控件的位置。事实上，这个问题已经被 GUI Layout Toolbox 解决了[①]，将在后面的章节和附录中详细介绍这个工具箱。为了简单起见，在这里我们暂时先使用这种原始的给控件定位的方法。

目前，界面上的控件对象还没和回调函数连接起来，它们和 Figure 的关系是 Children 和 Parent 的关系，并且只有把控件的 Parent 设置成 Figure，它们才会被 MATLAB 画出来，如图7.5所示。

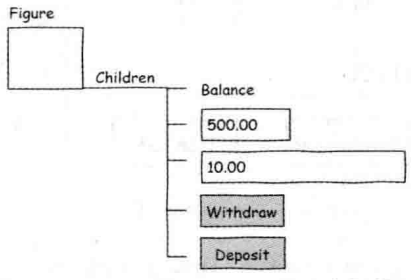


图 7.5 Figure 的 Children 是 5 个控件对象

整个界面上，需要对用户行为作出响应的控件对象是两个按钮。第三步就是在这两个按钮上注册回调函数（listeners）：

main.m

```
set(withdrawButton,'callback',@(o,e)withdraw_callback(o,e));
```

①http://www.mathworks.com/matlabcentral/fileexchange/27758-gui-layout-toolbox。

```
set(depositButton, 'callback',@(o,e)deposit_callback(o,e));
```

注意：根据 MATLAB 的规定，回调函数的第一个参数必须是发布消息的对象，即上面的参数 o，在这里，发布消息的是按钮本身；第二个参数是事件对象，即参数 e。

最后一步是设计两个回调函数，因为两个函数类似，这里只给出一个函数的定义：

withdraw callback

```
1 function withdraw_callback(o,e)
2     hfig = get(o,'Parent');
3     inputBox = findobj(hfig,'Tag','inputbox');
4     input = str2double(get(inputBox,'string'));          %  得到用户输入
5     balanceBox = findobj(hfig,'Tag','balancebox');       %  得到 balance 数值
6     balance = str2double(get(balanceBox,'string'));
7     balance = balance - input ;                          %  更新 balance
8     set(balanceBox,'string',num2str(balance));           %  更新 balance 显示
9 end
```

上述的回调函数做了如下的工作：

(1) 第 2 行，通过 get 函数得到按钮对象 o 的 Parent，即 figure 的 Handle。

(2) 第 3，4 行通过 figure 的 Handle 得到 inputbox 控件的 Handle，从而得到用户的输入。

(3) 第 5 行，通过 figure 的 Handle 得到 balancebox 控件的 Handle，从而得到目前余额。

(4) 第 7、8 行，把余额和用户输入做减法，并且更新 balancebox 中的余额数目。

使用程序方式构造 GUI 的过程如图7.6 所示。

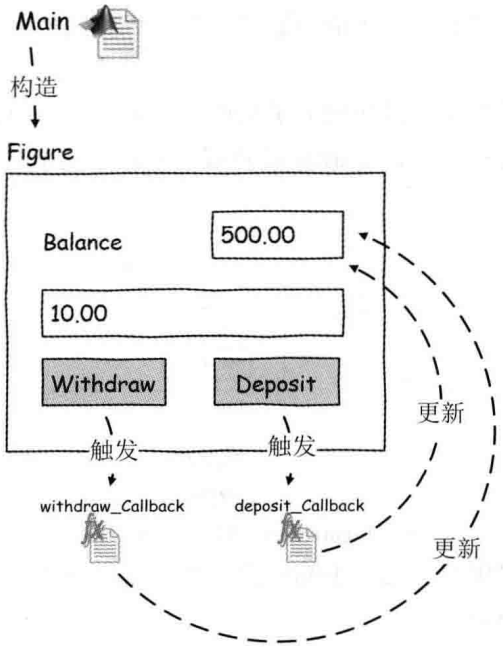


图 7.6　使用程序的方式构造 GUI

这个简单的通过程序的方式设计 GUI 的流程是这样的：

(1) 由脚本程序负责构造 Figure 对象以及 Figure 上的控件对象。

(2) 该脚本程序还负责给控件注册回调函数。

(3) 设计独立的 MATLAB 函数，作为回调函数。

通过这种完全用程序的方式来做 GUI 编程，结构简单，GUI 在脚本被执行时被动态生成，不需被保存成 FIG 文件。

7.3 如何用面向对象的方式进行 GUI 编程

既然 GUI 的设计可以通过面向过程的方式来设计，那么自然也可以把它用面向对象的思想来改造。这章介绍如何使用最常见的 GUI 面向对象编程模式，即模型–视图– 控制器（Model-View-Controller）模式进行 GUI 编程。仔细分析前面的 main 函数可以发现，程序可以被明显地分成两个部分。

程序的第一部分可以归结为模型（Model），反映程序的中心逻辑，在这里很简单，如果是存款，balance 的计算方法为

$$\text{balance} = \text{balance} + \text{input}$$

如果是提款，则是

$$\text{balance} = \text{balance} - \text{input}$$

反映到程序上，就是

模型部分

```
balance = balance - input ;
```

还有存款：

模型部分

```
balance = balance + input ;
```

程序的第二部分可以归结为视图（View），用来显示 User Interface 给用户。

(1) 它的职责包括产生 Figure 和控件对象，它们放在什么位置，以及设置默认值：

视图部分

```
% 构造初始的 figure 对象和控件对象
hfig =  figure('pos',[100,100,300,300] );
withdrawButton = uicontrol('parent',hfig,'string','withdraw',...
                           'pos',[60 28 60 28]);
depositButton  = uicontrol('parent',hfig,'string','deposit',...
                           'pos',[180 28 60 28]);
inputBox   = uicontrol('parent',hfig,'style','edit','pos',[60 85 180 28],...
                       'string',num2str(input),'Tag','inputbox');

balanceBox = uicontrol('parent',hfig,'style','edit','pos',[180 142 60 28],...
                      'string',num2str(balance),'Tag','balancebox');

textBox = uicontrol('parent',hfig,'style','text','string',...
                      'Balance','pos',[60 142 60 28]);
```

(2) 把控件和它们的回调函数联合起来：

设置两个按钮的回调函数

```
set(withdrawButton,'callback',@(o,e)withdraw_callback(o,e));
set(depositButton, 'callback',@(o,e)deposit_callback(o,e));
```

可以看出，实际编程中，模型反映的是程序的逻辑，是相对稳定的，而视图界面需要经常调整，而且界面的调整不应该影响到程序的模型。采用面向对象的思想，最显然的做法是把界面和模型封装到不同的类当中去，让各个类各司其职。这叫做把界面的变化和模型解耦。还有一些和界面模型无关的功能，比如处理用户的输入。我们把它们归到第三个类中去，叫做控制器（Controller）类。这样用三个基本的类来组织整个 GUI 程序就是我们要介绍的模型–视图–控制器（MVC）模式。总地来说，MVC 模式把 GUI 程序分解成三部分，如图7.7所示。

- □ 模型（Model）：负责程序的内在逻辑。
- □ 视图（View）：负责构造，展示用户界面。
- □ 控制器（Controller）：负责处理用户输入。

对于一个用户事件的响应，基本的流程如图7.7所示。回调函数在 Controller 中，首先由 Controller 中的回调函数作出第一轮的处理。如果该响应需要涉及程序的内在逻辑，由 Controller 负责调用 Model 中的相关函数；如果 Model 中的某些内在状态发生变化，还需要通知 View 对象；View 对象接到通知，查询 Model 的内在状态，并且在界面上作出更新，呈现给用户。

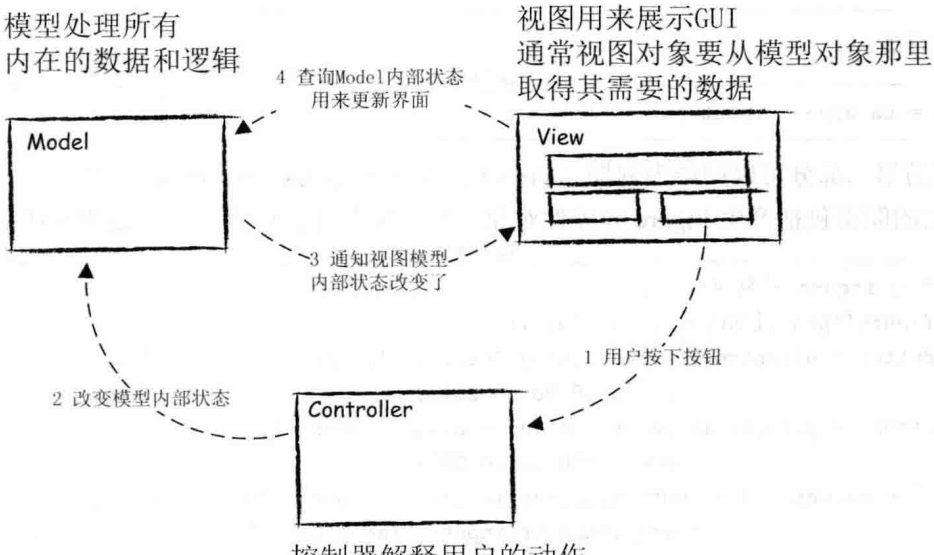


图 7.7　使用 MVC 方式分解 GUI

7.4　模型类中应该包括什么

在提款机模型中，最重要的一个数据是存款余额，所以把余额（balance）定义为 Model 类的一个属性。为了方便 View 类界面，即 View 类对象监听和更新 balance，我们还要定义

一个事件，叫做 balanceChanged。当 deposit 和 withdraw 函数被调用时，balance 将发生变化，model 类对象发出通知，给监听该事件的 listener。在 listener 处登记的 View 类中的回调函数将被调用，该函数查询 Model 中的 balance，并且更新显示，代码如下：

Model

```
classdef Model < handle
    properties
        balance             %  存款余额
    end
    events
        balanceChanged      %  余额发生变化事件
    end
    methods
        function obj = Model(balance)           %  Model 类的构造函数
           obj.balance = balance ;
        end
        function deposit(obj,val)               %    deposit 的最终处理函数
           obj.balance = obj.balance + val;
           obj.notify('balanceChanged');        %   通知
        end
        function withdraw(obj,val)              %    withdraw 的最终处理函数
           obj.balance = obj.balance - val;
           obj.notify('balanceChanged');        %  通知
        end
    end
end
```

如果要求存款不能为负，这样的逻辑判断也应该归属于 Model 类，可以添加到上述的 Model 中，这里从略。

Model 对象和 View 对象之间的关系是被监听和监听的关系（Observer Pattern）。如图7.8所示，Model 的余额的改变将触发 balancedChanged 事件，View 的响应式更新界面中的 balance 值。

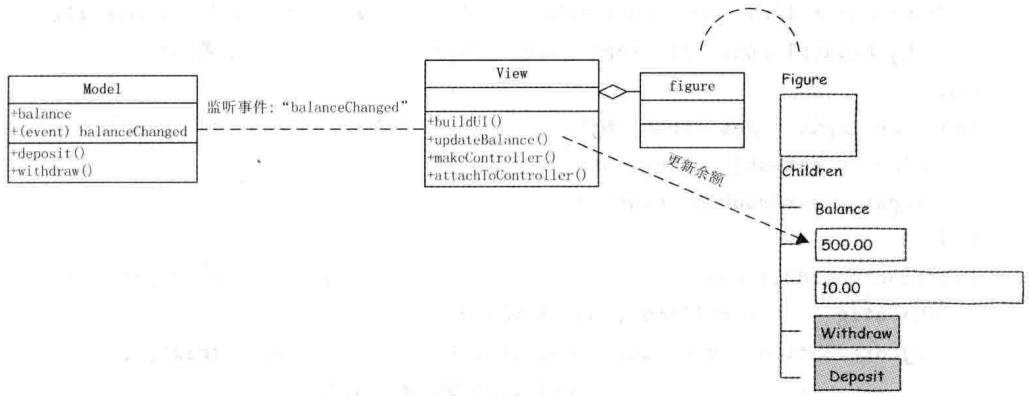


图 7.8 Model 类中还要定义个事件，让 View 类的对象去监听

7.5 视图类中应该包括什么

视图类（View）的职责包括：

- 向用户展示 GUI 界面。
- 在模型中注册 listener，监听模型内部状态的变化。
- 从模型中得到内部状态，并且显示到 GUI 上。

要理解 View 类，首先要懂得视图类和 figure 对象以及 figure 上的所有控件对象都是组合的关系，即视图对象拥有 figure 对象以及所有控件对象。其中，这些控件对象和 figure 对象之间的关系用 MATLAB Handle Graphics 的术语来说，属于 Parent 和 Children 的关系。

View

```
classdef View < handle
    properties
        viewSize       ;     % 用于控制视图的大小的数组
        hfig           ;     % 视图类对象拥有 figure 的 Handle
        drawButton     ;     % 控件对象 drawbutton 作为 View 类的属性
        depositButton  ;     % 控件对象 depositButton 作为 View 类的属性
        balanceBox     ;     % balanceBox 是用来显示余额的 uicontrol 对象
        numBox         ;     % numBox 是用户用来输入提款或者存款额度的 uicontrol 对象
        text           ;     % 视图上的静态字符
        modelObj       ;     % 视图类将拥有模型对象的 Handle
        controlObj     ;     % 视图类将拥有控制器对象的 Handle
    end
    properties(Dependent)
       input ;
    end
     methods
        function obj = View(modelObj)                          % View 类的构造函数
            obj.viewSize  =  [100,100,300,200];
            obj.modelObj = modelObj ;          % 初始化 View 对象中模型的 Handle
            obj.modelObj.addlistener('balanceChanged',@obj.updateBalance); % 注册
            obj.buildUI();                                                  % 构造显示
            obj.controlObj = obj.makeController();   % View 类负责产生 controller
            obj.attachToController(obj.controlObj); % 注册控件的回调函数
        end
        function input = get.input(obj)              % 该函数负责从界面上得到用户的输入
            input = get(obj.numBox,'string');
            input = str2double(input);
        end
        function buildUI(obj)                        % 构造界面并且展示给用户
           obj.hfig =  figure('pos',obj.viewSize);
           obj.drawButton = uicontrol('parent',obj.hfig,'string','draw',...
                                      'pos',[60 28 60 28]);
           obj.depositButton = uicontrol('parent',obj.hfig,'string','deposit',...
```

```
                                        'pos',[180 28 60 28]);
            obj.numBox = uicontrol('parent',obj.hfig,'style','edit',...
                                   'pos',[60 85 180 28],'tag','numBox');
            obj.text = uicontrol('parent',obj.hfig,'style','text','string',...
                                 'Balance','pos',[60 142 60 28]);
            obj.balanceBox = uicontrol('parent',obj.hfig,'style','edit',...
                               'pos',[180 142 60 28],'tag','balanceBox');
            obj.updateBalance();
        end
        function updateBalance(obj,scr,data)            % 更新界面上的 balance
            set(obj.balanceBox,'string',num2str(obj.modelObj.balance));
        end
        function controlObj = makeController(obj)       % View 负责产生自己的控制器
            controlObj = Controller(obj,obj.modelObj);
        end
        function attachToController(obj,controller)     % controller 构造完成之后
            funcH = @controller.callback_drawbutton;
            set(obj.drawButton,'callback',funcH);       % 给控件注册回调函数
            funcH = @controller.callback_depositbutton;
            set(obj.depositButton,'callback',funcH);
        end
    end
end
```

对 View 类设计的理解有如下要点：

- □ View 对象和 Controller 对象的关系：每个 View 对象必须至少拥有一个控制器。因为控制器负责处理用户的输入，所以必须根据视图的情况来设计，所以 View 类负责产生自己的 Controller。在上述代码中，对应的是 `obj.controlObj = obj.makeController()` 这一行。
- □ 因为视图类的职责仅仅是展示，不包括响应，所以 View 视图还要负责给自己拥有的控件注册回调函数。该回调函数来自于 Controller 类。
- □ View 类必须拥有 Model 对象的 Handle，这样才能在 updateBalance 函数中查询 Model 的内部状态，并且更新界面。

后面将看到，控制器对象也将拥有视图对象的 Handle，控制器和视图类关系紧密，是紧耦合。

在这个 View 类中，有两种注册“回调函数”的方式。一种是传统的方式，即使用 set 函数，对 MATLAB 的 GUI 控件注册回调函数：

给 GUI 控件注册回调函数

```
funcH = @controller.callback_drawbutton;
set(obj.drawButton,'callback',funcH);
```

另一种方式是如第4章提到的，给用户自定义类中的事件“注册回调函数”：

给 GUI 控件注册回调函数

```
obj.modelObj.addlistener('balanceChanged',@obj.updateBalance); % 注册
```

这两种方式大致是一致的，区别在于一个是内置的 UI 控件对象，一个是用户自己定义的对象。对于内置的 UI 控件，我们沿用传统的注册回调函数的方法；对于用户定义的类的时间，我们使用 addlistener 方式来注册监听者。视图和控制器之间的关系如图7.9所示。

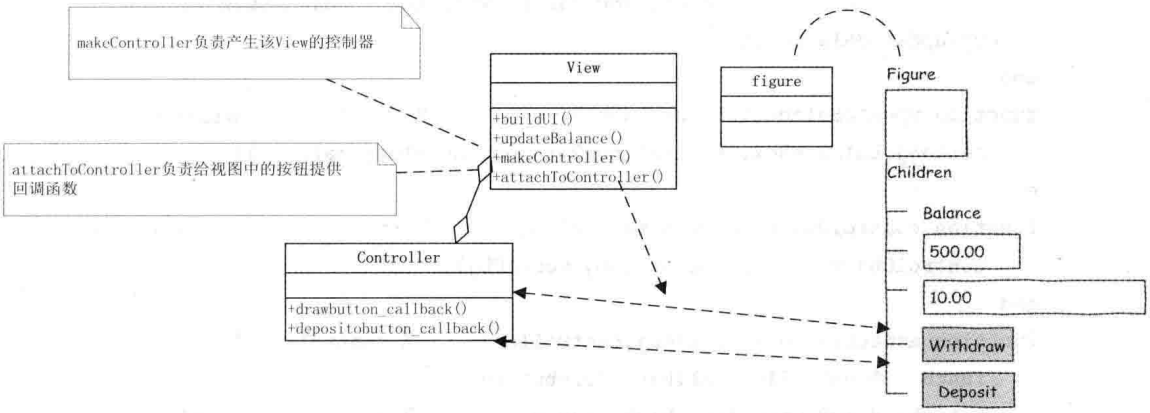


图 7.9 视图和控制器之间相互拥有对方的 Handle

7.6 控制器类中应该包括什么

控制器类的职责和内容是：让模型和视图解耦，处理来自用户的输入，解释用户和 GUI 的交互，改变视图类上控件的外观。比如打开保持文件，改变按钮的状态，擦除坐标轴上的图等。

要理解 Controller 类的设计，就必须要理解：为什么“处理用户输入”这样简单的功能不能放到 View 类中？答案是：如果程序简单，完全可以这样设计。但是我们需要认识到，这样会让 View 类的代码变得更复杂。视图将负责展示和处理用户输入两个功能，违背了单一职责的原则，并且还会造成视图和模型之间的强耦合。强耦合的缺点是：如果想更换视图背后的控制器（程序复杂到一定程度之后），程序对修改就不是封闭的。通过引入控制器类，让视图和模型之间做到松耦合，就可以让设计更加有弹性。设计本身没有对错之分，只是能否更好地符合需求。

该例子中，控制类相对简单，用户的单击按钮动作所导致的结果是模型中 balance 变量的改变。由于 balance 是 Model 类中的数据，所以实际的对 balance 的修改应该转移到 Model 类中进行，因此 Controller 类中按钮的回调函数仅仅是调用 Model 中的 withdraw 和 deposit 函数而已。

Controller

```
classdef Controller < handle
    properties
        viewObj   ;          % controller 对象必须拥有 view 对象的 Handle
        modelObj ;           % controller 对象必须拥有 model 对象的 Handle
    end
    methods
```

```
        function obj = Controller(viewObj,modelObj)  % controller 类构造函数
            % 初始化让控制器类对象拥有模型对象和视图对象的 Handle
            obj.viewObj   = viewObj;
            obj.modelObj = modelObj;
        end
        function callback_drawbutton(obj,src,event)     % draw 按钮的回调函数
            obj.modelObj.withdraw(obj.viewObj.input);  % 去调用 model 类的 withdraw 函数
        end
        function callback_depositbutton(obj,src,event) % deposit 按钮的回调函数
            obj.modelObj.deposit(obj.viewObj.input);   % 去调用 model 类的 deposit 函数
        end
    end
end
```

注意，Controller 类为了能够调用 Model 对象中的函数 withdraw 和 deposit，必须拥有 model 对象的 Handle。如图7.10所示，控制器必须拥有模型对象的 Handle。

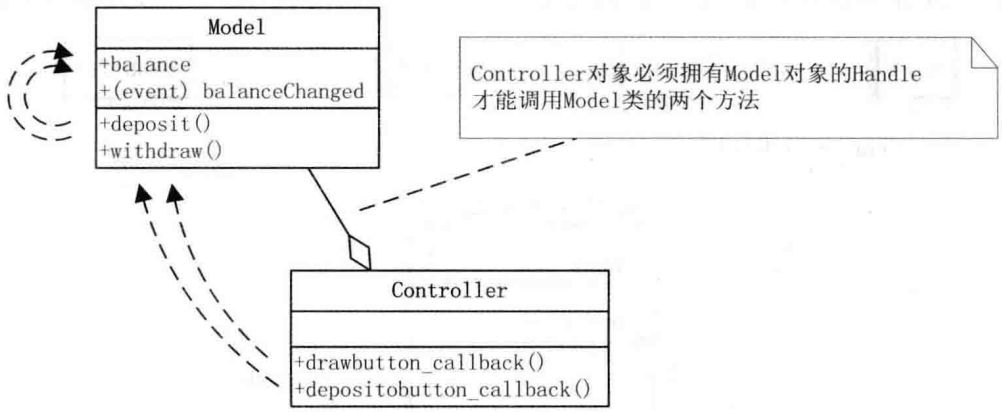


图 7.10 控制器必须拥有模型对象 Handle

7.7 如何把 Model、View 和 Controller 结合起来

前三节我们介绍了如何设计 MVC 的三个类，这节把这三个类放到一起去，下面就是主程序 main.m：

main.m

```
1 modelObj = Model(500);
2 viewObj = View(modelObj);
```

其中，第 1 行代码负责声明一个模型对象，并且初始化存款为 500 元；第 2 行声明一个视图对象，这个 View 类的构造函数的第一个参数是模型对象，所以 View 类内部拥有模型对象的 Handle。而 Controller 对象，将在 View 类的构造函数中被创建出来。从整体上来说，这个程序的架构如图7.11所示。

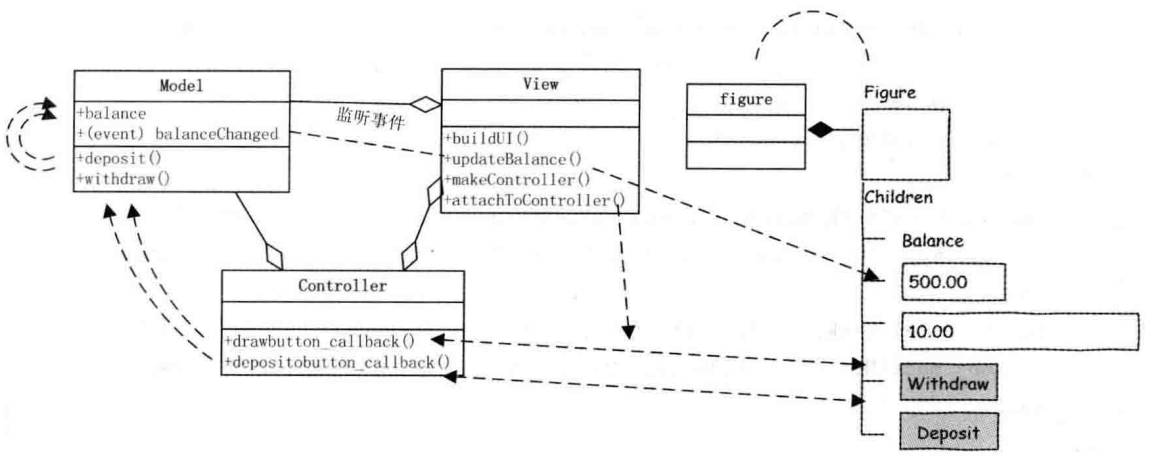


图 7.11 把 Model、View、Controller 放在一起

图7.12所示的序列图演示了 MVC 三个对象的产生顺序。序列图是按照时间顺序排列的对象之间的交互模式图。图中上方的方块表示对象，方块上延伸出来的线是对象的生命线，生命线之间的箭头代表对象之间消息的发送。对象通过互相发送消息来完成具体的任务。

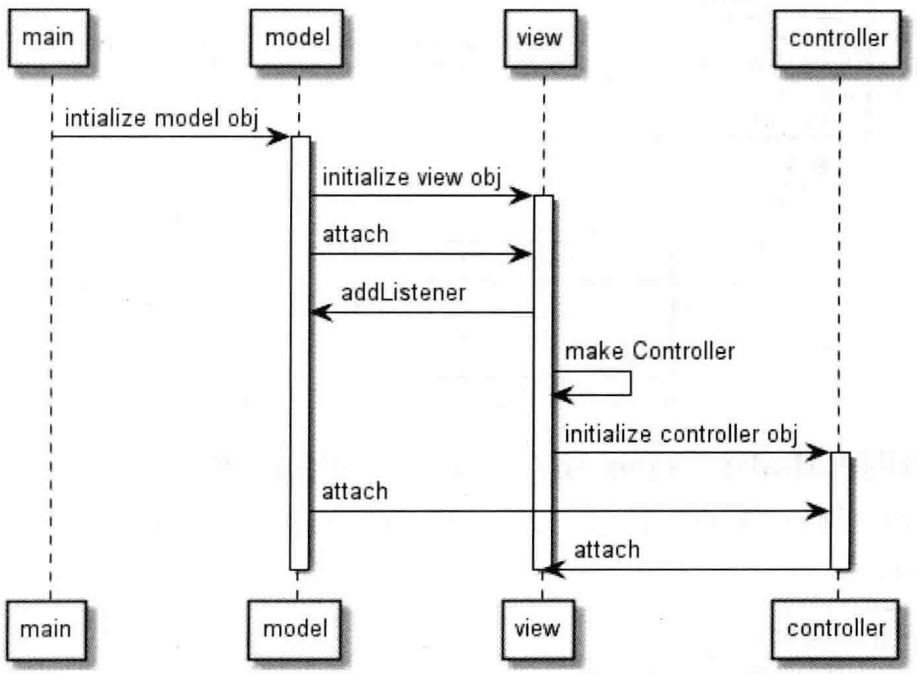


图 7.12 MVC 模型的对象产生的序列图

在命令行上键入：

Command Line

```
>> main
```

启动 GUI 程序，MATLAB 将构造一个 GUI 呈现给用户，并且等待用户的指令。如果用户在界面上输入数字，并且单击 withDraw 按钮，发生的事件可以用序列图7.13 表示。

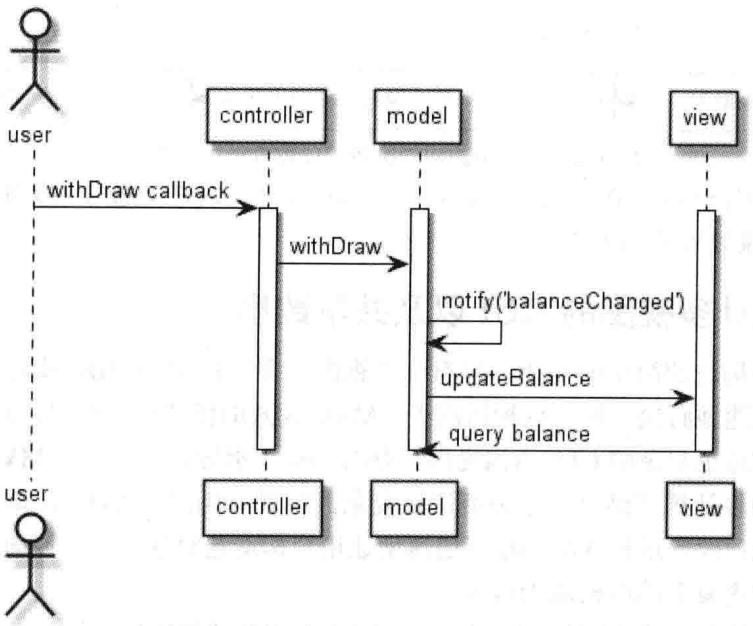


图 7.13 withDraw 按钮被按下之后发生的事件序列图

图7.13中，user 按下 withDraw 按钮，将触发一系列事件的发生。withDraw 按钮是 MATLAB 内置的控件对象，被监听的事件是“按钮被按下”，回调函数是 withdraw_callback。该函数的定义在 Controller 中，在视图类中的 attachToController 方法中被注册。withDraw 将造成存款余额的变化，在 Model 对象中，有一个自定义的事件，即“存款余额的变化” balanceChanged，有一个 listener 监听该事件，其内部的响应函数是视图类中的 updateBalance 方法。至此，用户按下 withDraw 按钮的事件处理完毕。

和前面的使用 GUIDE 的复杂程度相比，使用 OOP 看上去虽然是复杂了，但是该 MVC 的模型是构造更复杂的 GUI 程序的基础，在之后的程序修改和更新中，将能体现这种编程思路的简捷。

初学 MVC 模型，难点在于需要对 GUI 程序的各个部分进行职责和功能的划分，并且把它们分别放到 Model、View 或者 Controller 类中去。Model 类的职责和功能比较容易判断，View 类和 Controller 类中的内容要稍微仔细地加以区分。为了方便记忆，我们总结一下三个类中所通常包括的内容，见表7.1。

表 7.1 Model、View 和 Controller 中包括的内容

	View 类	Controller 类	Model 类
属性	控件对象 View 和 Model 对象的 Handle	View 和 Model 对象的 Handle	必要的业务逻辑
事件	用户和控件对象进行交互		内部状态的改变
方法	构造 UI 构造 Controller 对象 给自己的控件注册回调函数	各控件的回调函数 用户和 GUI 的之间互动的处理	必要的业务逻辑

注意，Controller 中控件的回调函数应仅涉及界面和用户交互的内容。如果 View 中控件触发的事件涉及业务逻辑，那么 Controller 中的回调函数其实可以被看做仅是一个中转函数，

如图7.14所示。

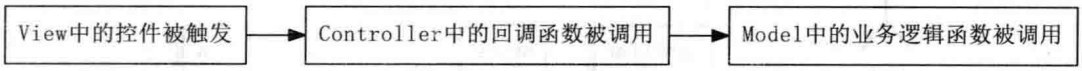


图 7.14 Controller 中的回调函数可以被看做是一个中转函数

最终的工作由 Model 中的方法来完成，在 Model 类可以定义通过相关事件，使用 notify 来通知 View 对象处理的结果。

7.8 如何设计多视图的 GUI 以及共享数据

如果用户所构造的简单的 GUI，只有一个视图，那么使用 GUIDE 或者面向过程式的设计方式就可以迅速地解决问题。这种情况下，MVC 模式的优势还不明显。MVC 的真正优势在于它为用户构造更复杂的 GUI 结构提供了基石和基本指导思想，而且 MVC 模式可以和其他的面向对象的设计模式结合起来去解决更复杂的问题。因此 MATLAB 语言可以满足用户不同层次的要求：面向过程式的编程可迅速地让用户构造起算法原型；而面向对象的编程让用户方便地搭建更复杂的结构成为可能。

假设要设计这样一个 MATLAB 程序：该程序和两个硬件通信，进行控制数据采集和数据分析，GUI 要求至少有三个视图，主视图用来可视化数据，主视图上有两个 Button，分别导向两个硬件的控制界面，当然程序的后台应该有专门的部分对数据进行分析，这里从略。如果严格按照 MVC 模式，不做任何简化，该 GUI 的设计草图如图7.15所示，根据标准的 MVC 模型，将有三个视图类、三个控制器类和三个模型类。

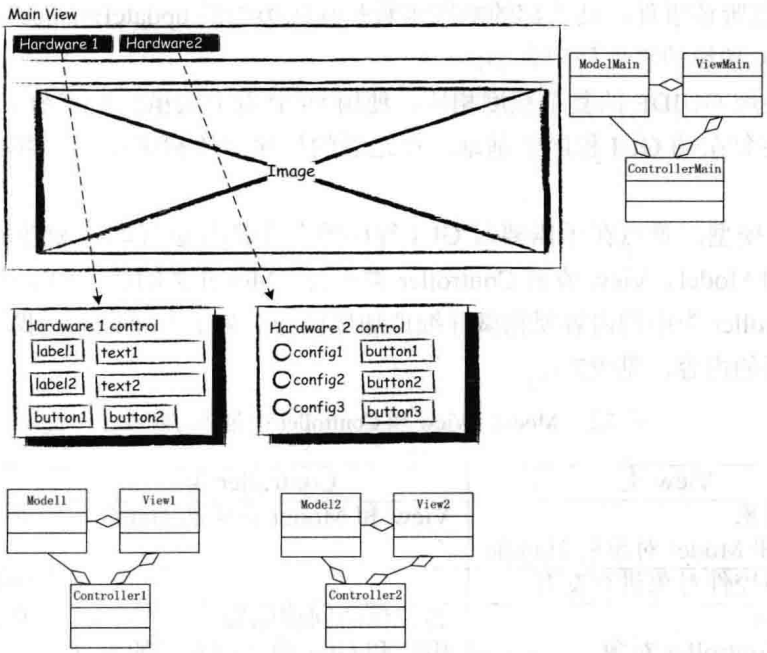


图 7.15 三个视图类、三个控控制器类和三个模型类

多视图 GUI 程序的一个常见问题就是 GUI 之间的相互“共享数据”，因为 MVC 模型把 GUI 的数据和界面分离，所以这个问题自然地就转换成模型对象与界面对象之间如何共享

数据的问题。其实，这个问题已经解决了[①]，即只要对象互相拥有彼此的 Handle 即可，就像标准的 MVC 模型中，Controller 和 View 类直接互相拥有对方的句柄来共享数据一样，如图7.16 所示。

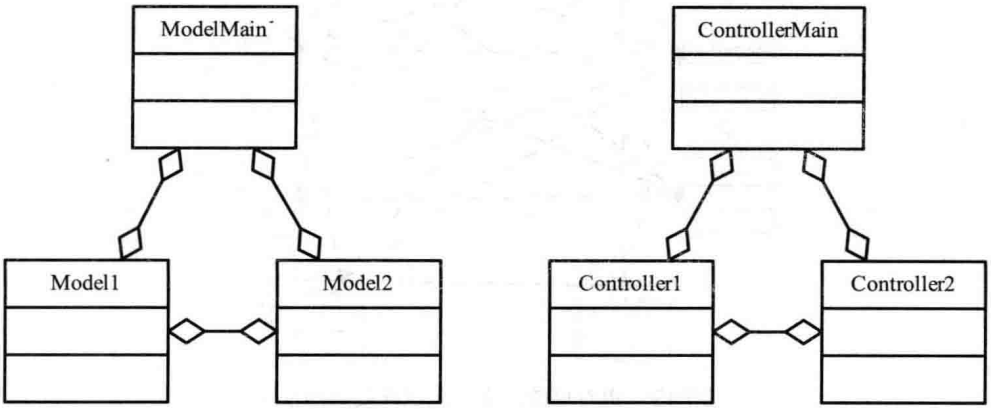


图 7.16 通过相互拥有 Handle 来共享数据

第一个硬件的 Model 类可以这样设计：

Model1

```
classdef Model1 < handle
    proprerties
        hModel2          % 该 Handel 的值可以在 constructor 或者 setter 中提供
        hModelMain
    end
    ......
    methods
        function accessData(obj)
            % 直接通过 hModel2 的 Handle 访问 hModel2 对象中的数据
            temp = hModel2.someProp ;
        end
    end
end
```

图7.16所示只是一个理想情况，更有可能的是各个视图、控制器、模型之间数据相互关联都需要得到彼此的数据，那么按照前面的思路，实际的 UML 就可能是像图7.17所示的这样错综复杂。

这个设计的缺点很明显：各个类之间的相互依赖耦合严重。解决方法之一是：设计一个 Context 类，如图 7.18 所示，其作用像一个枢纽[②]，要求其他的类的对象在创建时都在这个上下文类中注册，并且给每个对象一个独一无二的 ID，当一个对象需要取得其他类的对象的数据时，就可以通过 ID 到 Context 类对象处获取其他类的 Handle。

①使用 GUIDE 编程时，GUI 之间的交互数据的方法则不是那么一目了然。
②这属于一种行为模式，叫做 Mediator Pattern。

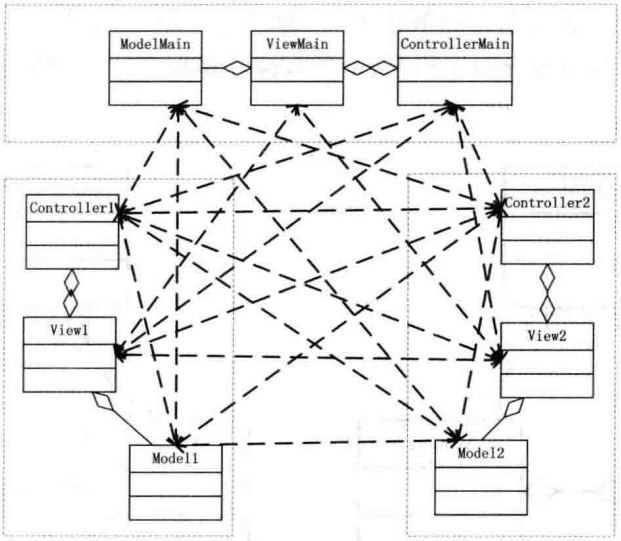


图 7.17 更有可能出现的是这样复杂的情况

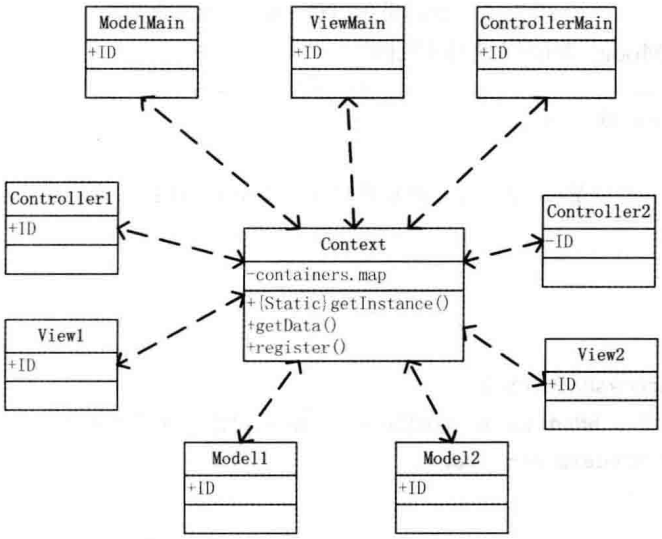


图 7.18 引入一个枢纽

因为我们希望在任何路径和函数中都可以随时通过 Context 对象得到所需要的对象的句柄，所以该对象应该是一个全局对象，而且一个程序只需要这么一个上下文类，这样就要限制该类所能产生的对象的数量，所以该类应该是一个 Singleton[①]，下面给出 Context 类的一个简单实现。

1. Context 类的设计

该上下文类中的核心数据结构是一个 MAP 容器（containers.Map）：其中，register 函数要求外部对象（client）提供 ID 作为 Key，并把外部对象的 Handle 作为 KeyValue，保存在 MAP 容器中；getData 方法通过 client 提供的 ID 在 MAP 容器中查询，并且返回注册的 Handle。

①详见第15.2 Singelton 节。

Context 类

```
classdef Context < handle
    properties
        dataDictionary ;
    end
    methods(Access = private)
        function obj = Context()
            obj.dataDictionary = containers.Map();
        end
    end
    methods(Static)
        function obj = getInstance()
            persistent localObj;
            if isempty(localObj) || ~isvalid(localObj)
                localObj = Context();
            end
            obj = localObj;
        end
    end
    methods
        register( obj,ID, data ) ;
        fdata = getData(obj,ID)  ;
    end
end
```

Context 类注册方法：

register 函数

```
function register( obj,ID, data )
    expr = sprintf(' obj.dataDictionary(\''%s\'') = data;',ID);
    eval(expr);
end
```

Context 类查询方法：

getData 函数

```
function data = getData(obj,ID)
 if isKey(obj.dataDictionary,ID)
     data = obj.dataDictionary(ID);
 else
     error('ID does not exist');
 end
end
```

2. 在 Context 对象处注册 ID

下面的脚本中假设，要控制的两个硬件对象分别是 Camera 和电源，于是我们声明出两个模型对象，并且每个对象都赋予一个在整个计算中独一无二的 ID。然后通过 getInstance 静态方法得到该上下文类的对象，再使用 register 方法注册。

Script

```
obj1 = ModelDevice('Camera');
obj2 = ModelDevice('PowerSource');
contextObj = Context.getInstance();              %  得到全局上下文对象
contextObj.register('Camera',obj1);              %  注册 Camera 对象
contextObj.register('PowerSource',obj2);         %  注册 PowerSource 对象
```

3. 从 Context 对象处查询 ID

下面的代码假设我们要在一个函数中获得模型对象 Camera 的句柄。只要 Context 类的定义在 MATLAB 的搜索路径上，我们就可以在任何环境中调用 getInstance 静态方法，Context 对象就好似一个被封装好的全局对象。getData 方法接受 ID 作为参数，返回和这个 ID 对应的句柄。注意，这里提供的只是最简化的 Context 类的实现，实用中，Context 类还要能够处理无效 ID 的情况。

function

```
function  someFunction()
    contextObj = Context.getInstance();
    hCamera = contextObj.getData('Camera');
end
```

如果需要，图7.18的设计还可进一步简化，要是各视图的 Controller 类都很简单，还可把三个 Controller 的职责集中到一个类中。这样，程序的设计可进一步简化成如图7.19所示。

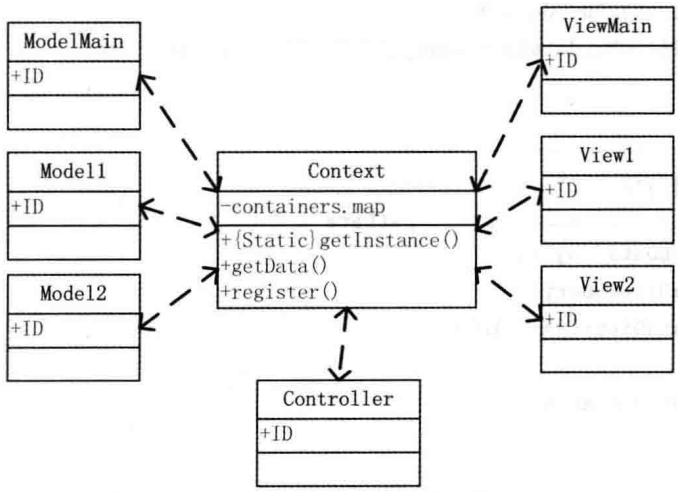


图 7.19 把三个 Controller 类合并到一起去

当然，上述设计仅仅是解决多视图 GUI 设计的一种方案之一。大多数情况下，在掌握的面向对象编程的基本原则之后，把设计方法用 UML 的方式表示出来，仔细分析问题的根源之后，解决方案其实可以是多种多样的。

7.9 如何设计 GUI 逻辑架构

从结构上来说，第7.8节的多视图 GUI 程序依然相对简单，而平时我们所见到的更多的是框架结构，即如图7.20所示的 GUI 程序。

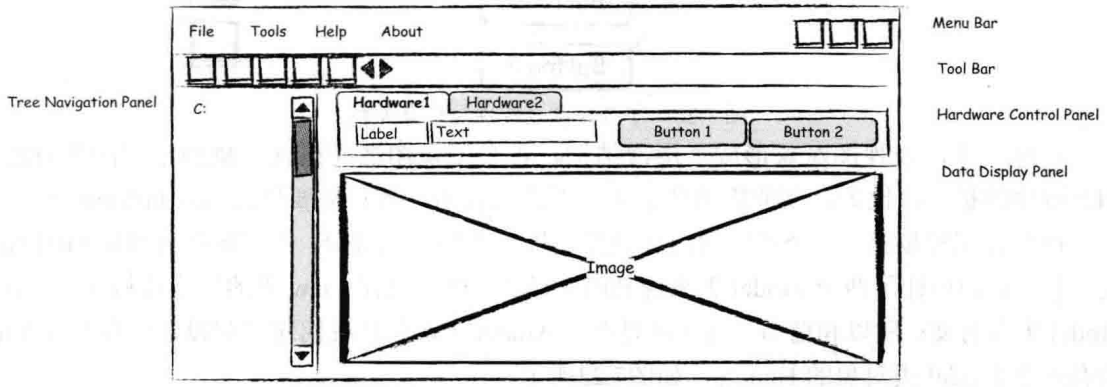


图 7.20 含有子视图的 GUI 草图

还是沿用第7.8节的例子，现在我们这样设计这个 GUI：该 GUI 有菜单栏、工具栏、控制面板选项、浏览栏，还有一个面板专门用来显示数据。在这节，我们先介绍如何在逻辑上设计这种 GUI 的架构，再在下一节中介绍如何对 GUI 上的控件进行自动布局。

针对这种含有 Menubar、Toolbar 和子视图的 GUI 结构，一种解决方法是：从基本的 MVC 模型出发，把各个视图类用 Composite（组合）关系组织起来（如图7.21所示），把各个控制器类用 Parent-Child 关系组织起来。

先介绍视图类。容易发现，较复杂的 GUI 可以分成各个子视图（它们本来在视觉上就是相互包含的关系），整个 GUI 的框架视图类叫做 MainView。该 MainView 可以作为一个最上层的视图对象容器，来包含菜单栏视图对象、工具栏视图对象、选项卡视图对象、浏览栏视图对象，还有数据显示视图对象。在初始化 MainView 对象时，也同时初始化各个子视图对象。

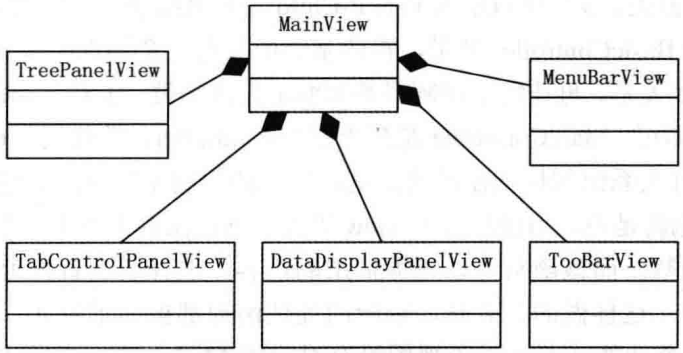


图 7.21 MainView 和其他子视图是组合关系

其实，子视图对象本身也是容器，大部分情况下，其进一步包含了面板和 uicontrol 的控件，比如 TabControlPanelView 包含两个 Tab 和若干 uicontrol 控件，如图7.22所示。在子视图的层次上，View 类的结构和前述的基本的 MVC 中 View 的结构没有什么不同。

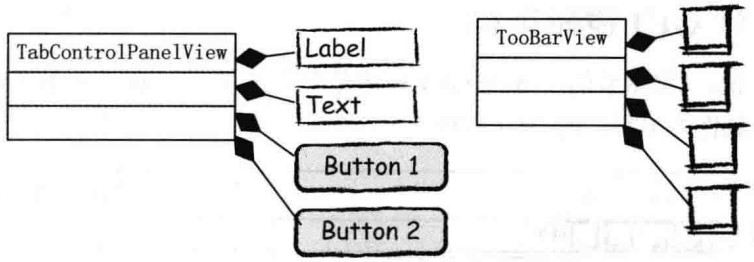


图 7.22 子视图和其控件是组合关系

这样一来，众视图对象形成了层次结构，各个子视图之间也就解耦和了。因此方便了 GUI 程序的扩张。比如，新的需求要给工具栏添加按钮，就只需要修改 TooBarView 类。

再来介绍模型类。一个程序的内部模型是程序中稳定的部分，不应该随外部的 GUI 而改变。上节我们设计了两个 Model 类来控制两个硬件，虽然现在 View 类的层次结构变了，但是 Model 类没有变，所以和前节一样，还是两个 Model 类，并且我们这里假设，只有 MainView 拥有两个 Model 类对象的 Handle，如图7.23所示。

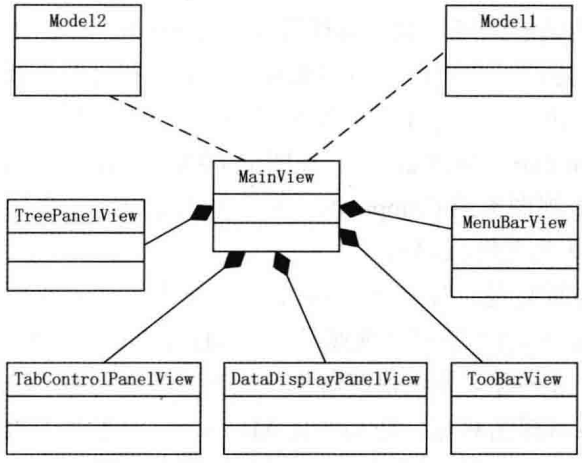


图 7.23 Model 是程序中稳定的部分

最后讨论控制器类。我们可以引入 Parent-Child 关系来组织它们的逻辑关系。如图7.24所示，首先定义一个 BasicController 基类，在该基类中，有一个属性叫做 parentController，用来定义 Parent-Child 关系。再让每个控制器类都继承自该基类，子 Controller 类只需要知道其 Parent 是哪个对象即可，MainController 是最上层的 Controller，是其他众 Controller 的 Parent。

用 Parent-Child 关系组织控制器的优点是：底层的控制器可以向高层的控制器转发其无法处理的请求[①]。也就是说，当用户与子 View 界面互动，比方说单击一个按钮，将触发底层控制器中的响应函数，而该函数可以根据请求的内容，选择要么直接处理这个请求。要么让其 Parent 去处理。这样做的好处是，响应的处理函数都被分了层次。如果响应涉及上层 Model 对象中的某个功能（而底层子视图没有对应的 Model 类），则可以沿着 Controller 的 Parent-Child 关系，把要求向上传递。这样，底层的控制器就无需事无巨细地处理所有请求了。

①这是行为模式中的 Chain of responsibility Pattern。

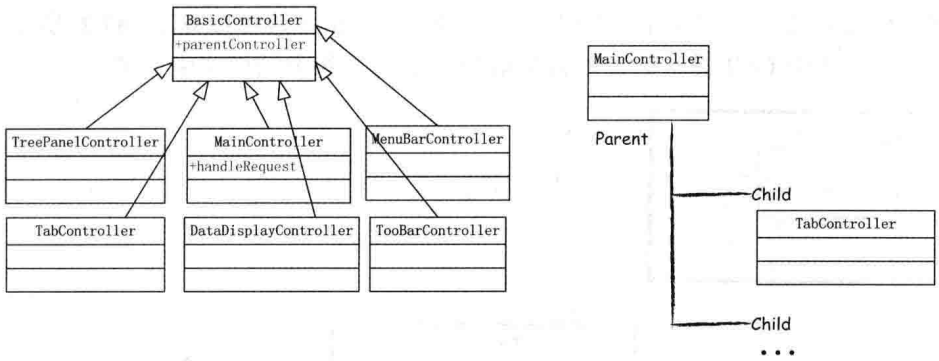


图 7.24 用 Child-Parent 关系来组织 Controller 类对象

一个底层控制器的响应函数可以用如下的方式，把请求转发给上层的控制器，其中 RequestID 用来标记请求的类型，提供给上层的控制器用以查找相应的响应函数。

```
function button1CallBack(obj)
   obj.parentController.handleRequest(RequestID)
end
```

图7.25中，由 Child Controller 提交来自 GUI 的请求。

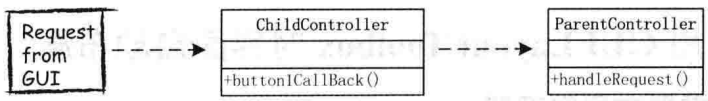


图 7.25 ChildController 向其 Parent Controller 提交请求

它并不确定最终处理请求的对象是谁，在复杂的 GUI 架构中，Controller 的结构可能是好几层，如果某个层次上的 Controller 无法处理请求，它可以把这个请求沿着 Parent-Child 链继续向上传递，直到有能够遇到可以处理该请求的控制器为止。

综合起来，这节介绍的设计 GUI 架构的模式叫做 HMVC（Hierarchical Model View Controller），如图7.26所示。其核心是通过 Controller 的 Parent-Child 结构，把各个子 MVC 模块连接起来。如果子模块 Controller 没法处理的响应，则通过 Parent-Child 链向上传递。

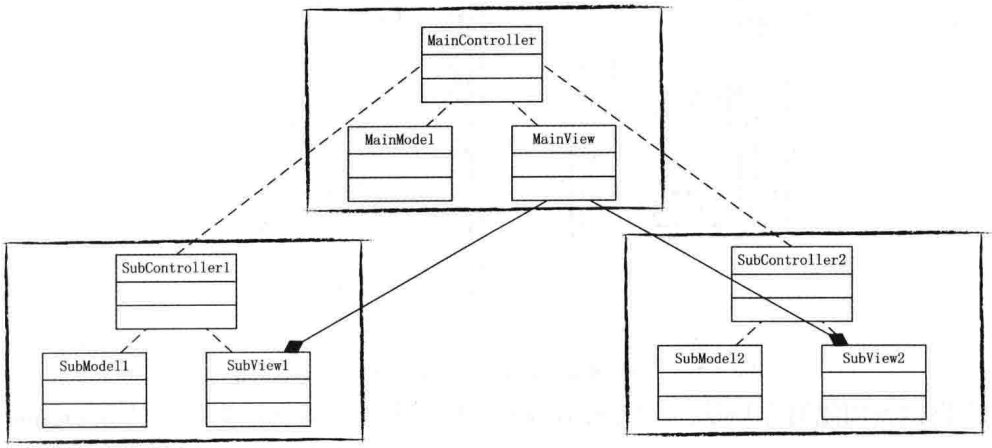


图 7.26 HMVC 模式

该种结构适合进一步地扩展 GUI 的架构，Parent-Child 链可以是如图7.27所示的多层形式。这样，一个大的 GUI 设计问题就被分解成各个层次的小的模块中去了。

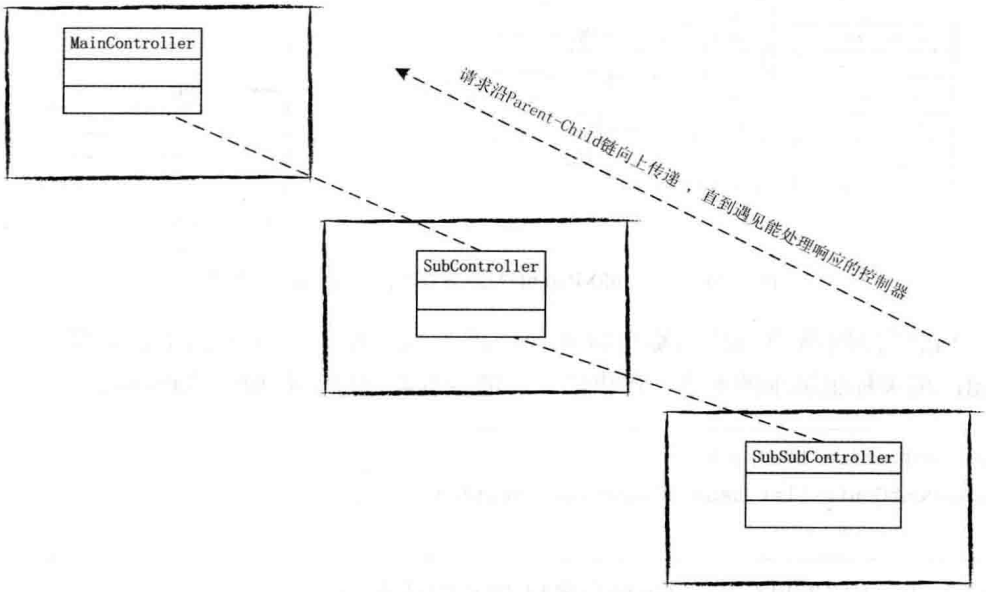


图 7.27 多层的 Controller 结构

7.10 如何使用 GUI Layout Toolbox 对界面自动布局

7.10.1 为什么需要布局管理器

我们先从设计简单的 GUI 开始，比如要用程序构造如图7.28左所示的 GUI，其中包含三个按钮。

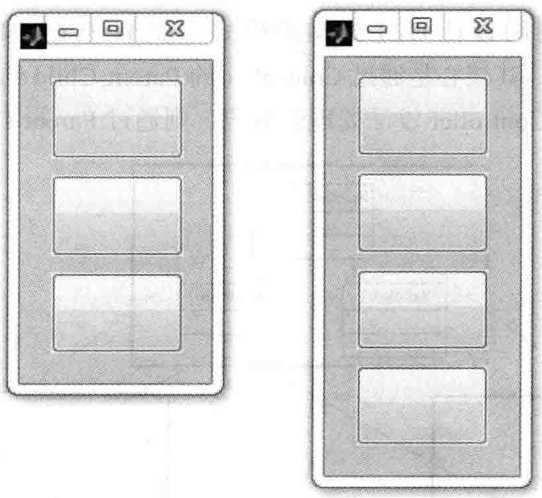


图 7.28 使用绝对布局法和相对布局法设计 GUI

首先尝试使用绝对布局法：声明 uicontrol 控件对象，并且直接指定控件在 Figure 上的位置：

绝对布局法

```
f = figure('Menubar','none','Toolbar','none','Position',[200 200 100 200]);
uicontrol('style','pushbutton','Position',[20 20 80 50], 'Parent',f)
uicontrol('style','pushbutton','Position',[20 80 80 50], 'Parent',f)
uicontrol('style','pushbutton','Position',[20 140 80 50],'Parent',f)
```

现在如果要扩展这个 GUI，再给它添加一个按钮，如图7.28右所示。于是，首先要增大 Figure 的尺寸，还要再计算出新的按钮的位置。这是绝对位置法的一个缺点：需要人工计算出控件的位置。

```
f = figure('Menubar','none','Toolbar','none','Position',[200 200 100 260]); % 增大
uicontrol('style','pushbutton','Position',[20 20 80 50], 'Parent',f)
uicontrol('style','pushbutton','Position',[20 80 80 50], 'Parent',f)
uicontrol('style','pushbutton','Position',[20 140 80 50],'Parent',f)
uicontrol('style','pushbutton','Position',[20 200 80 50],'Parent',f) % 人工计算位置
```

显然，该 GUI 是简化过的，而在实际 GUI 设计中，一个细微的设计的改动，可能要重新计算许多控件的位置，导致程序后期的维护修改成本变高。所以，绝对布局法仅适合简单的界面设计。尽管我们可以把 uicontrol 的单位设置成 normalized，把绝对布局变成相对布局来避免重新计算位置的问题。

相对布局法

```
f = figure('Menubar','none','Toolbar','none','Position',[200 200 100 200]);
uicontrol('style','pushbutton','Units','normalized',...
          'Position',[0.1 0.1  0.8 0.25],'Parent',f)
uicontrol('style','pushbutton','Units','normalized',...
          'Position',[0.1 0.4  0.8 0.25],'Parent',f)
uicontrol('style','pushbutton','Units','normalized',...
          'Position',[0.1 0.7  0.8 0.25],'Parent',f);
```

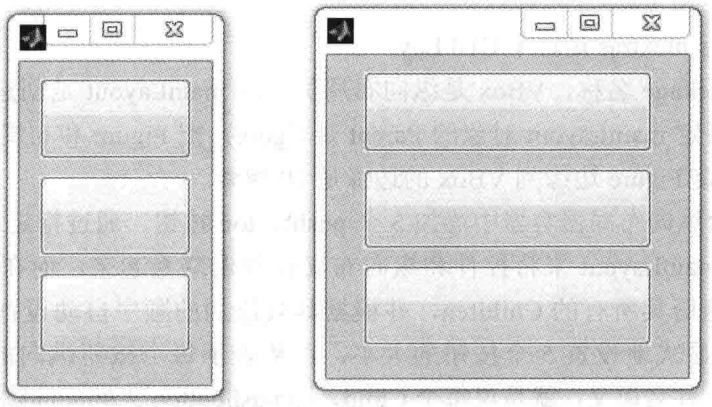


图 7.29 拖拽界面 pushbutton 的尺寸也跟着变化

上述的命令中，'Position',[0.1 0.1 0.8 0.25] 指定的是该按钮在 Figure 中的相对位置。但是这样又引入了一个新的问题，如果用鼠标拖拽这个界面的右下角，这三个按钮也会跟着变大，如图7.29右所示。这显然不是通常所见到的 GUI 界面的正常行为。如果想要按钮在放大 Figure 时保持原来的大小，就须要给控件提供一个 resize 函数，用来处理鼠标

拖拽的响应，而且 GUIDE 布局法也面临同样的问题。总地来说，无论是绝对位置布局方法、相对布局法，还是 GUIDE 布局法，扩展和修改界面所带来的附加工作都很多。

7.10.2　纵向布局类 VBox

在认识到了绝对和相对布局方法在 GUI 设计中的种种不方便之后，从这节开始介绍 GUI Layout Toolbox 工具箱[①]。该工具箱中提供了多种在 Figure 中摆放控件的布局类，如 uiextra.HBox，uiextras.VBox，uiextras.TabPanel。这些类简单易用，有了它们，利用程序的方式来设计 GUI 就变得方便多了。接着上节的三个按钮的例子，首先介绍纵向布局器 VBox 类，它用来对控件进行单列纵向布局，效果如图7.30所示。

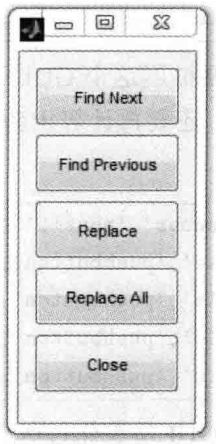


图 7.30　纵向布局器 VBox 效果图

实现图7.30的代码如下：

纵向布局器

```
1  f = figure('Menubar','none','Toolbar','none','Position',[500 500 80 250]);
2  mainLayout = uiextras.VBox( 'Parent', f ,'Padding',10);   % 纵向布局器对象
3
4  uicontrol('style','pushbutton','string','Find Next',     'Parent', mainLayout)
5  uicontrol('style','pushbutton','string','Find Previous', 'Parent', mainLayout)
6  uicontrol('style','pushbutton','string','Replace',       'Parent', mainLayout)
7  uicontrol('style','pushbutton','string','Replace All',   'Parent', mainLayout)
8  uicontrol('style','pushbutton','string','Close',         'Parent', mainLayout)
9
10 set( mainlayout, 'Sizes', [40 40 40 40 40], 'Spacing', 5 );
```

程序中：第 2 行 uiextras 是这个 GUI Layout 工具箱中的 Package 名称，VBox 是纵向布局类，而 mainLayout 是创建出来的纵向布局类的对象。通过指定 mainLayout 对象的 Parent 是 figure，把 Figure 和布局器对象联系起来，'Padding',10 指定 Figure 边缘到 VBox 的边缘是 10 像素。

第 4 至 8 行给纵向布局器容器中添加 5 个 pushbutton 按钮，通过指定这 5 个 pushbutton 控件的 Parent 是 mainLayout 来将控件和纵向布局管理器联系起来，而在 mainLayout 对象内部，会自动地遍历其所有的 Children，并根据其被添加的顺序自动设置它们在 Figure 上的位置。第 10 行显式地设置 5 个按钮的大小，正数表示每个按钮纵向的尺寸是 40 像素，而'Spacing', 5，顾名思义，就是说每个 Child，即 pushbutton 之间的间隔是 5 像素。Figure 对象和布局器对象之间的关系如图7.31所示。

①读者可以从 MATLAB Central 的 File Exchange 处下载该 GUI Layout Toolbox 工具箱。

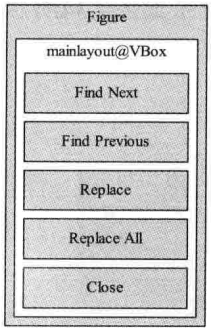


图 7.31 VBox 类对象管理 5 个 Pushbutton 的位置

图7.31中每个层次之间都是 Parent-Child 的关系，读者可以把 VBox 想象成一个没有边框，而且背景色和 Figure 的背景色相同的方框，这样它在视觉上就是隐形的；它存在的唯一目的就是作为一个容器，给自己内部的控件自动地设定位置。

7.10.3 横向布局类 HBox

这节我们使用布局管理工具箱中的 HBox 类来对控件进行单行横布局，界面要求如图7.32所示，并且要求在窗口被拉长时，效果如图7.33所示。

图 7.32 HBox 效果图

实现图7.32的代码如下：

横向布局器

```
1 f = figure('Menubar','none','Toolbar','none','Position',[500 500 200 45]);
2 mainLayout = uiextras.HBox( 'Parent', f ,'Padding',10);
3
4 uicontrol('style','text','string','Find what:', 'Parent', mainLayout,'FontSize',10)
5 uicontrol('style','popupmenu','string','prev1|prev2|prev3','background','white',
6           'Parent', mainLayout)
7 set( mainlayout, 'Sizes', [60 -1], 'Spacing', 10 );
```

其中，第 2 行声明了一个横向的布局器 mainLayout 对象，其 Parent 是 Figure。第 4 至 6 行声明两个 uicontrol 控件，作为被布局的对象，一个是标签，一个是下拉列表框。

第 7 行是控制两个控件相对大小的关键：'Sizes,[60 -1]'，其中正数 60 代表该控件的大小用像素来控制，而负数表示比例。这里-1 表示第二个控件将填满除去 60 像素后剩下的空间。同理，如果一个布局上有三个控件，并且尺寸关系是 'Sizes',[60 -1 -1]，那么表示第一个控件的横向尺寸是 60 像素，而剩下的两个控件大小是 1 比 1，填满剩余的空间。使用这种绝对像素和相对比例来控制控件大小的方法，可以有效地应对 Figure 尺寸变化控件大小变化的问题，比如拖拽 Figure 的右下角调整大小，第一个控件 text 对象将仍然保持其尺寸 60 像素，而下拉列表控件将随着 Figure 的增大而增大，如图7.33所示。

Figure 对象和布局器对象之间的关系如图7.34所示，图中的包含关系对应程序中对象之间的 Parent-Child 关系。

图 7.33 窗口被拉长后的效果

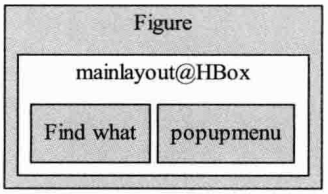


图 7.34 图像对象和布局器对象之间的关系

7.10.4 选项卡布局 TabPanel

选项卡布局也是一个常见的 GUI 布局方法。在 GUI Layout Toolbox 之前，在 MATLAB 中实现选项卡要费上一番周折，而现在可以使用 uiextras 包中的 TabPanel 类极简单地实现选项卡界面，效果如图7.35所示。

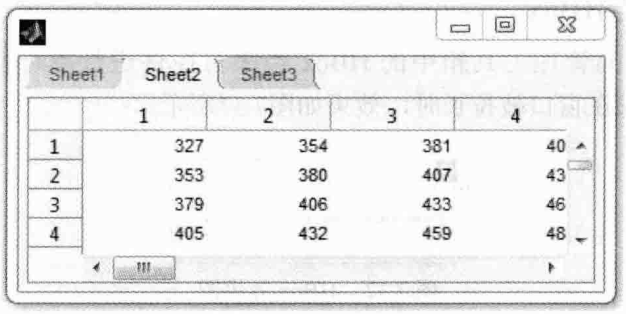


图 7.35 TabPanel 选项卡效果图

实现图7.35的代码如下：

选项卡布局

```
1 f = figure('Menubar','none','Toolbar','none');
2 mainLayout = uiextras.TabPanel( 'Parent', f, 'Padding', 5 );
3
4 uitable('Data',magic(25),'Parent', mainLayout );
5 uitable('Data',magic(25),'Parent', mainLayout );
6 uitable('Data',magic(25),'Parent', mainLayout );
7
8 p.TabNames = {'Sheet1', 'Sheet2', 'Sheet3'};
9 p.SelectedChild = 2;
```

其中，第 2 行声明一个选项卡布局器对象，是 Figure 仅有的直接 Child，该选项卡将布满整个 Figure。

第 4 至 6 行往 mainLayout 中添加 uitable 对象。

第 8 行通过选项卡对象的的 TabNames 属性给每个 Tab 的标题命名。

第 9 行的`p.SelectedChild = 2`表示默认情况下被选中的 Tab 是第二个。

Figure 和 TabPanel 对象以及控件之间的关系如图7.36所示，图中的包含关系对应程序中对象之间的 Parent-Child 关系。

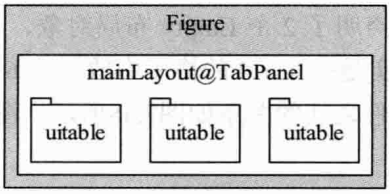


图 7.36 图像对象和布局器对象之间的关系

7.10.5 网格布局类 Grid

网格布局 Grid 类比 VBox 和 HBox 布局稍复杂，适用于多行多列的控件排列的情况。这节我们用网络布局类 Grid 来构造如图7.37所示的界面。

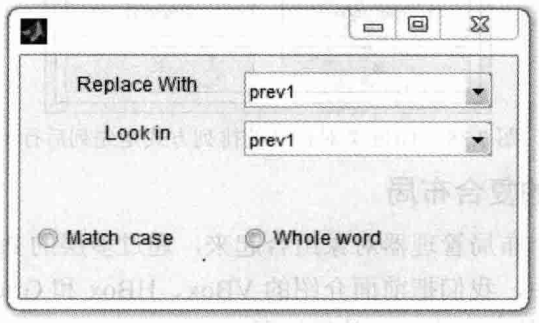


图 7.37 Grid 类效果图

Grid 布局

```
f = figure('Menubar','none','Toolbar','none','Position',[500 500 300 150]);
mainLayout = uiextras.Grid( 'Parent', f, 'Padding', 10 );
%   第一列
uicontrol('Style','text','String','Replace With','FontSize',9,'Parent', mainLayout);
uicontrol('Style','text','String','Look in','FontSize',9,'Parent', mainLayout);
uiextras.Empty('Parent', mainLayout);
uicontrol('Style','radio','String','Match  case','FontSize',9,'Parent', mainLayout);

% 第二列
uicontrol('style','popupmenu','string','prev1|prev2|prev3','background','w',...
          'Parent', mainLayout  );
uicontrol('style','popupmenu','string','prev1|prev2|prev3','background','w',...
          'Parent', mainLayout  );
uiextras.Empty('Parent', mainLayout);
uicontrol('Style','radio','String','Whole word','FontSize',9,'Parent', mainLayout);

set(mainLayout, 'RowSizes', [25 25 25 25] ,'ColumnSizes', [120 150], ,'Spacing',5);
```

其中，第 2 行声明了 Grid 类容器。

第 4 至 14 行一共声明了 8 个对象，添加到 mainLayout 容器中。

第 15 行中的 set 语句指定对该 8 个对象的排列方式是 4×2，并且 Grid 类对控件的排列方式是先列后行，所以第 4 至 7 行 4 个对象构造的是第一列，第 10 至 15 行构造第二列，如

图7.38 程序中的第 6，12 行还声明了 2 个 Empty 布局对象，该对象的作用是在 Grid 格局上空出一格来，所以上述程序其实是在 4 × 2 的格子中放置了 6 个 uicontrol 的控件的对象。

Figure 和 Grid 对象以及控件之间的关系如图7.38所示，图中的包含关系对应程序中对象之间的 Parent-Child 关系。

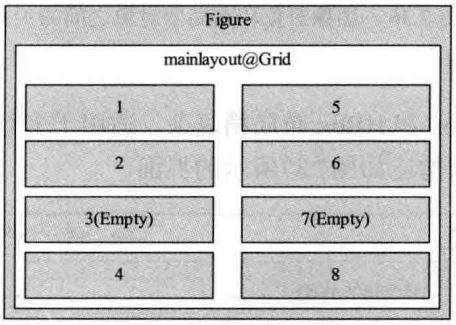


图 7.38 Grid 类对控件的排列方式是先列后行

7.10.6 GUI Layout 的复合布局

复合布局是指把各个布局管理器对象组合起来，通过多层的 Parent-Child 关系对界面进行更灵活的设计。这节中，我们把前面介绍的 VBox、HBox 和 Grid 类组合起来，放到一个界面上去模拟 MATLAB 的查询窗口，效果如图7.39所示。

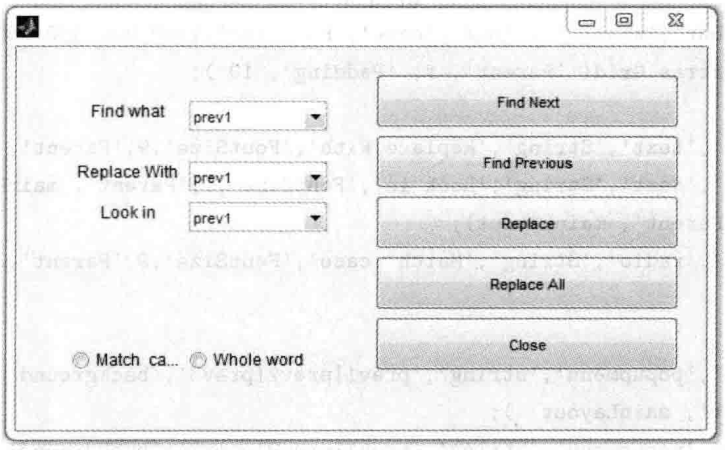


图 7.39 复合布局 HBox 中包含 VBox，HBox 和 Grid

实现图7.39的代码如下（仅包括所有的布局对象，具体控件对象省略）：

复合布局

```
f = figure('Menubar','none','Toolbar','none','pos',[200 200 500 230]);
mainLayout = uiextras.HBox('Paprent',f,'Spacing',10);
  leftLayout = uiextras.VBox('Parent',mainLayout,'Spacing',10,'Paddingp',5);
      lUpperLayout = uiextras.HBox('Parent',leftLayout);
      lLowerLayout = uiextras.Grid('Parent',leftLayout);
  rightLayout = uiextras.VBox('Parent',mainLayout,'Spacing',10);

..... % 其余略
```

图7.39可以先被横向地分成两栏，所以第 2 行先声明一个 HBox 对象来作为 mainLayout，作为 Figure 的直接 Children。在该 HBox 中，左右两栏显然都是纵向排列，所以 HBox 的两个之间的 Children 是 VBox。为了演示复合布局，在第 4，5 行中，我们特地把左栏继续细分成上半部分的 HBox 和下半部分的 Grid 两个 Layout。

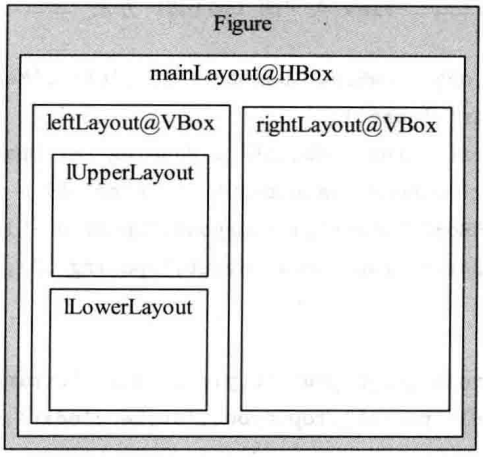


图 7.40 图像对象和布局器对象之间的关系

Figure 和各布局器对象之间的关系如图7.40所示，图中的包含关系对应程序中对象之间的 Parent-Child 关系。

7.10.7 把 GUI Layout Toolbox 和 MVC 模式结合起来

在实际应用中，GUI Layout Toolbox 和设计模式可以结合起来，可以使用 MVC 模式来管理程序模型和界面之间的逻辑关系，使用布局器类来管理界面上控件的几何关系。并且布局器对象存在的作用仅是自动指定界面上的控件的位置，对程序的逻辑没有影响。把已有的 GUI 修改成使用布局器对象的程序只需要简单的几行代码。这里我们以前面的提款机界面为例，说明如何在已有的 MVC 的继承的基础上使用布局管理器。其中提款机例子中 View 类最初的设计是这样的：

View 类中的 buildUI 方法

```
function buildUI(obj)        % 构造界面 并且展示给用户
  obj.hfig =  figure('pos',obj.viewSize,'NumberTitle','off','Menubar','none',...
                     'Toolbar','none');
  obj.drawButton = uicontrol('parent',obj.hfig,'string','draw',...
                             'pos',[60 28 60 28]);
  obj.depositButton = uicontrol('parent',obj.hfig,'string','deposit',...
                                'pos',[180 28 60 28]);
  obj.numBox = uicontrol('parent',obj.hfig,'style','edit',...
                         'pos',[60 85 180 28],'tag','numBox');
  obj.text = uicontrol('parent',obj.hfig,'style','text','string','Balance',...
                       'pos',[60 142 60 28]);
  obj.balanceBox = uicontrol('paprent',obj.hfig,'style','edit',...
                             'pos',[180 142 60 28],'tag','balanceBox');
```

```
  obj.updateBalance();
end
```

注意，上面使用了绝对布局法，其中每个按钮的摆放都要通过手工设计，很不方便。可以将其修改成如下使用布局器的形式：

View 类中的 buildUI 方法

```
function buildUI(obj)
 obj.hfig =  figure('pos',obj.viewSize,'NumberTitle','off','Menubar','none',...
                  'Toolbar','none');
 mainLayout = uiextras.VBox('Parent',obj.hfig,'Padding',5,'Spacing',10)
 topLayout = uiextras.HBox('Parent',mainLayout,'Spacing',5);
 middleLayout = uiextras.HBox('Parent',mainLayout,'Spacing',5);
 lowerLayout = uiextras.HBox('Parent',mainLayout,'Spacing',5);

 % 上层
 obj.text = uicontrol('parent',topLayout,'style','text','string','Balance');
 obj.balanceBox = uicontrol('parent',topLayout,'style','edit','background','w');
 % 中层
 obj.numBox = uicontrol('parent',middleLayout,'style','edit','background','w');
 % 下层
 obj.drawButton = uicontrol('parent',lowerLayout,'style','pushbutton',...
                          'string','draw');
 obj.depositButton = uicontrol('parent',lowerLayout,'style','pushbutton',...
                           'string','deposit');
 set(topLayout,'Sizes',[-1,-1]);
 set(lowerLayout,'Sizes',[-1,-1]);
 obj.updateBalance();
end
```

这里一共使用了 4 个布局器对象，效果如图7.41所示，布局器的嵌套关系如图7.42所示。

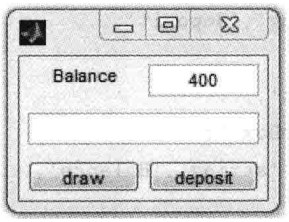


图 7.41 效果图

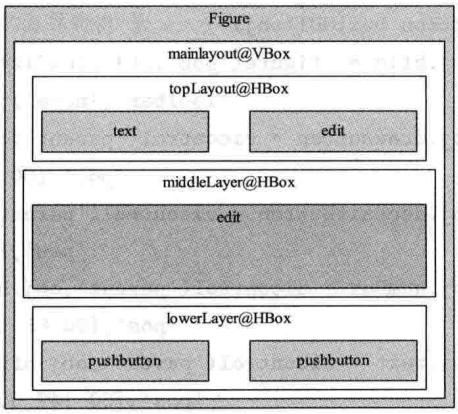


图 7.42 图像对象和布局器对象之间的关系

第 2 部分

面向对象编程中级篇

第3部分

面向对象编程与数据

第 8 章　类的继承进阶

8.1　继承情况下的 Constructor 和 Destructor

在基础篇中，我们初步介绍了 Constructor 和 Destructor 的用法。在中级篇中，我们将介绍在继承结构下，Constructor 和 Destructor 的使用规则。

8.1.1　什么情况需要手动调用基类的 Constructor

在初始化对象的过程中，如果有参数需要传递给基类 Constructor，则需要显式地（explicitly）在子类 Constructor 中调用基类的 Constructor。例如：

Base

```
classdef Base < handle
    properties
        a
    end
    methods
        function obj = Base(a)

            obj.a = a; % 初始化基类成员
        end
    end
end
```

Derived

```
classdef Derived < Base
    properties
        b
    end
    methods
        function obj = Derived(a,b)
          obj = obj@Base(a);
          obj.b = b;     % 初始化子类成员
        end
    end
end
```

如果声明一个对象：

Script

```
obj = Derived(1,2) ;
```

则子类对象的构造过程是这样的：Derived Constructor 先被调用，其中 Base Constructor 再被调用，a 数据被初始化；然后 Base Constructor 返回；最后完成 Derived Constructor 中的其余工作，b 数据被初始化。

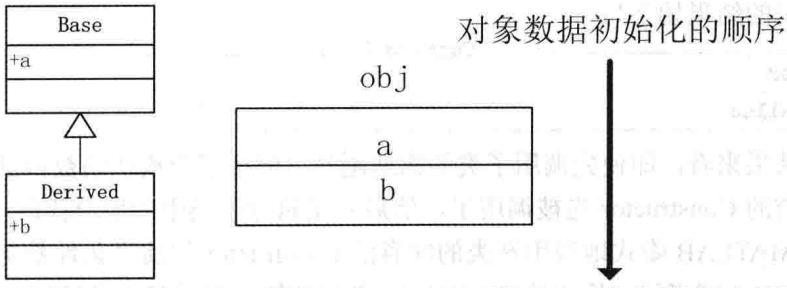


图 8.1　属性的初始化顺序：来自基类的属性先被初始化，其次才是子类的属性

虽然 Derived Constructor 是先被调用的，但是先被初始化的数据是 Base 类中的属性。如果用图8.1来表示一个对象的构造，则可以这样理解：Base 对象的属性是根基，Derived 的对象属性的初始化应该发生在 Base 属性初始化之后。

8.1.2 什么情况可以让 MATLAB 自动调用基类的 Constructor

还是沿用图8.1的继承结构，仅仅简化 Base 和 Derived 类中的成员，如果不需要向 Base 类的构造函数传递参数，就不需要在 Derived 的 Constructor 中显式地调用 parent 的 constructor。MATLAB 会隐式地自动帮助用户调用 Base 类的 Constructor。

Base

```
classdef Base<handle
    methods
        function obj = Base()


            disp('Base CTOR called');
        end
    end
end
```

Derived

```
classdef Derived< Base
    methods
        function obj = Derived()
            % MATLAB 在运行用户 disp 命令
            % 之前会先调用 Base 的构造函数
            disp('Derived CTOR called');
        end
    end
end
```

MATLAB 规定：如果在子类 Constructor 中，没有显式地调用基类的 Constructor，MATLAB 会在 Constructor 的一开始自动地调用基类的缺省的 Constructor，这相当于在 MATLAB 内部把 Derived 的类的定义修改成：

Derived

```
classdef Derived < Base
    methods
        function obj = Derived()
            obj = obj@Base() ;              % MATLAB 在内部做的工作相当于加上这行代码
            disp('Derived CTOR  called');
        end
    end
end
```

验证如下，声明一个 Child 对象：

Script

```
obj  = Sub();
```

命令行显示的结果如下[①]：

Command Line

```
Base CTOR called
Derived CTOR called
```

从命令行结果来看，即使先调用子类的构造函数，但在子类构造函数内部的任何代码被执行之前，基类的 Constructor 先被调用了，然后才是执行子类中的用户代码，如图8.2所示。

注意，让 MATLAB 隐式地调用基类的缺省的 Constructor 的前提条件是基类必须定义了缺省的 Constructor，或者说，基类的 Constructor 必须能够处理零参数的情况；否则 MATLAB 将报错。

如果子类存在多重继承结构，并且 Derived 中的 Constructor 没有显式地调用基 类的

①注意，该输出结果是 MATLAB R2011a 的结果。

Constructor，MATLAB 会根据继承的顺序调用基类的默认的 Constructor。

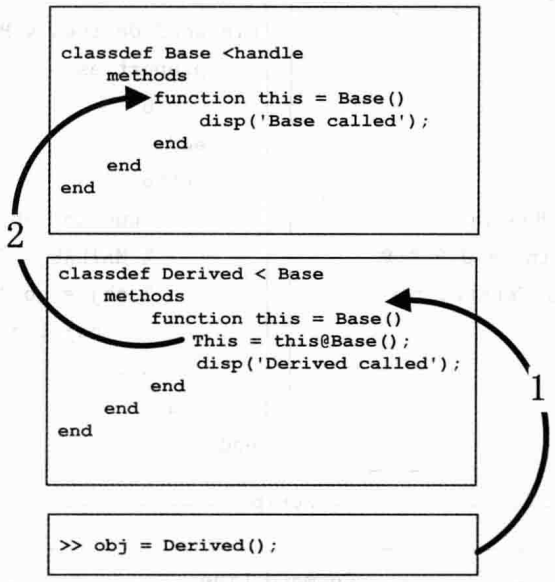


图 8.2 基类的构造函数在子类构造函数的一开始被调用

Derived

```
classdef Derived < Base1 & Base2
    methods
        function obj = Derived()



            disp('Derived CTOR called');
        end
    end
end
```

MATLAB 会在内部按照继承的顺序对 Derived 的 Constructor 进行扩充：

Derived

```
classdef Derived< Base1 & Base2
    methods
        function obj = Derived()
            obj = obj@Base1();  %  内部扩充
            obj = obj@Base2();  %  内部扩充
            disp('Derived CTOR called');
        end
    end
end
```

8.1.3 常见错误：没有提供缺省构造函数

上节我们提到，在不需要传递参数给基类的 Constructor 的情况下，MATLAB 会自动帮助用户调用基类的 Constructor。这里需要注意前提条件：用户必须提供一个缺省的构造函数；

否则，就会出现如下情况：

Base

```
classdef Base < handle
    properties
        a
    end
    methods
        function obj = Base(a)
        % 由于没有 nargin == 0 的判断
        % 这不是一个缺省 Constructor
            obj.a = a ;
        end
    end
end
```

Derived

```
classdef Derived < Base
    properties
        b
    end
    methods
        function obj = Derived(b)
        % MATLAB 将在这里尝试自动调用
        % obj = obj@Base(); 出错
            obj.b = b ;
        end
    end
end
```

Script

```
obj = Derived();
```

Command Line

```
??? Input argument "a" is undefined.

Error in ==> Base>Base.Base at 7
            obj.a = a ;

Error in ==> Derived>Derived.Derived at 6
        function obj = Derived(b)

Error in ==> main at 1
obj = Derived();
```

这里出错的原因是：因为 MATLAB 隐式地自动调用的是缺省的 Constructor，但是这里用户已经提供了一个自定义的 Constructor，并且没有提供 nargin==0 的情况的处理，所以 MATLAB 将显示无法找到所需要的构造函数的错误。

8.1.4 在 Constructor 中调用哪个成员方法

在第2.6.3小节中我们提到，子类中可以重新定义基类的方法（也叫做 override 基类的方法）。也就是说，如果有以下的代码结构，子类 Derived 的 foo 方法 override 了基类 Base 的 foo 方法。那么，如图8.3所示，当一个对象调用 foo 方法时，MATLAB 会动态地判断调用方法的对象实际上属于哪个类，如果属于子类，则调用子类中的 foo 方法；如果属于 Base 类，则调用 Base 类的 foo 方法。

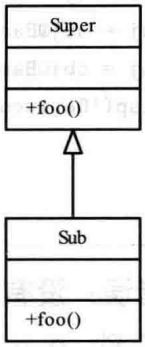


图 8.3 如果基类和子类中都定义了 foo 方法

Base

```
classdef Base < handle
    methods
        function obj = Base()
            disp('Base CTOR called');
            obj.foo();
        end
        function foo(obj)
            disp('Base foo called');
        end
    end
end
```

Derived

```
classdef Derived < Base
    methods
        function obj = Derived()

            disp('Derived CTOR called');
        end
        function foo(obj)
            disp('Derived foo called');
        end
    end
end
```

如果声明的是基类的对象，在基类的 Constructor 中调用的 foo 方法一定是来自于基类：

Script

```
oBase = Base();
```

Command Line

```
Base CTOR called
Base foo called
```

现在我们考虑这样的情况，当基类和子类都提供了 foo 方法，声明一个子类对象，初始化过程中基类的 Constructor 被调用，而基类 Constructor 中又调用了成员方法 foo，这时到底哪个 foo 方法被调用了？是子类的 foo，还是基类的 foo？我们可以先用代码验证一下：

Script

```
oDerived = Derived();
```

Command Line

```
Base CTOR called
Derived foo called
Derived CTOR called
```

从输出看出，如果声明的是子类对象，在基类的 Constructor 中，如果调用 foo 方法，MATLAB 将调用子类的 foo 方法。MATLAB 方法 Dispatch 的规则是：查找方法的 signature，而方法的 signature 由参数列表中的对象和方法名称决定。现在参数中的对象是 obj，它的类型是 Sub 类型，因此即使在父类的 Constructor 中，仍会调用子类的 foo 方法，如图 8.4所示。

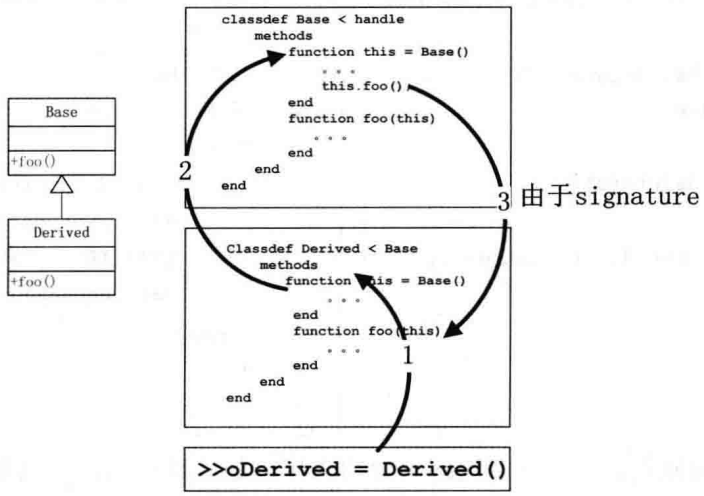


图 8.4 如果在基类的 Constructor 中调用 foo 方法：到底调用了哪个类的 foo 方法？

从语义上来说，当用户声明 DerivedObj = Derived() 时，其意图是声明一个子类对象，尽管构造这个子类对象，首先要调用基类的构成函数，但是从本质上来说，从构造的一开始，这个对象就是一个子类对象了。所以，即使是在基类的 Constructor 函数中调用了 foo 成员方法，MATLAB 也决定应该使用子类的 foo 方法。

8.1.5 析构函数被调用的顺序是什么

在继承体系下，一个子类对象的析构函数的调用恰好和构造函数的调用顺序相反，子类的析构函数先被调用，从上至下，基类的析构函数最后被调用：

Base

```
classdef Base < handle
    methods
        function delete(obj)
            disp('Base delete called');
        end
    end
end
```

Sub

```
classdef Sub < Base
    methods
        function delete(obj)
            disp('Sub delete called');
        end
    end
end
```

在脚本中首先声明一个对象，再显示地调用 delete 函数：

Script

```
obj = Sub();
obj.delete();
```

输出：

Command Line

```
Sub delete called
Base delete called
```

在继承结构下，用户只需要调用 Sub 类的 delete 函数，整个对象在 MATLAB 内所占的内存空间就会得到释放。MATLAB 会帮用户在子类的 delete 函数的末尾扩充，调用基类 delete 函数的命令，所以用户不需要在 Sub 的 delete 函数中显示地调用基类的 delete 函数。如下面的代码中，左边是用户定义的子类的 delete 函数，右边是这个 delete 函数在 MATLAB 内部中的等效的样子。

用户提供的 delete 函数

```
classdef Sub < Base
    methods
        function delete(obj)

            disp('Sub delete called');

        end
    end
end
```

MATLAB 内部的 delete 函数

```
classdef Sub < Base
    methods
        function delete(obj)
            disp('Sub delete called');
            %MATLAB 为用户补上这个命令
            delete@Base();
        end
    end
end
```

MATLAB 的规则是：在子类 delete 函数的末尾，隐式地自动调用其基类的 delete 函数。并且为了确保基类 destructor 能被调用，从而完全释放对象所占用的内存。MATLAB 在调用

基类的 delete 方法时，将忽略 delete 方法的访问权限。也就是说，即使 Base 类的 delete 方法被声明成了私有`Access = private`，在子类对象被 delete 时，该基类的 delete 函数也会被强制调用。

8.2 MATLAB 的多重继承

8.2.1 什么情况下需要多重继承

在了解什么是多重继承之前，我们先以一个动物园的动物为例来看一看使用多重继承的动机。用类来形容动物，用继承来形容动物的科属分类。比如大熊猫属于熊科，于是我们构造出动物园的动物基类 ZooAnimal，熊类 Bear 和熊猫类 Panda，Bear 继承自 ZooAnimal，Panda 继承自 Bear。再具体一些，ZooAnimal 中的属性可以是动物的具体的名字 name。ZooAnimal 中包含一些抽象的方法，比如演出 display 和喂食 feed。因为不同的动物演出和喂食是不同的，所以在 ZooAnimal 中这些方法都应该设置成 Abstract[①]，在具体的动物类中再实现这些方法。Bear 类中可以指定一些相对于 ZooAnimal 更细化的一些属性，比如，Bear 类的动物都需要一个具体的驯兽师 trainer 作为其属性。除了实际的动物，我们还可以构造一个辅助类 Endangered，它封装了抽象的濒临灭绝的动物，其中有一个抽象方法 protect，该方法表示稀有动物要受到保护。因为 Panda 即是 Bear 类也是濒临灭绝的动物，所以 Panda 还要继承 Endangered 类。Panda 类要具体实现 display 和 feed，还有 protect 方法，Panda 类还可以有自己的方法，比如熊猫很可爱，可以提供拥抱（Hug）方法。所以，这个例子的 UML 看上去是这样的：Panda 继承了一个以上的类，并且这种结构就是多重继承，如图 8.5所示。

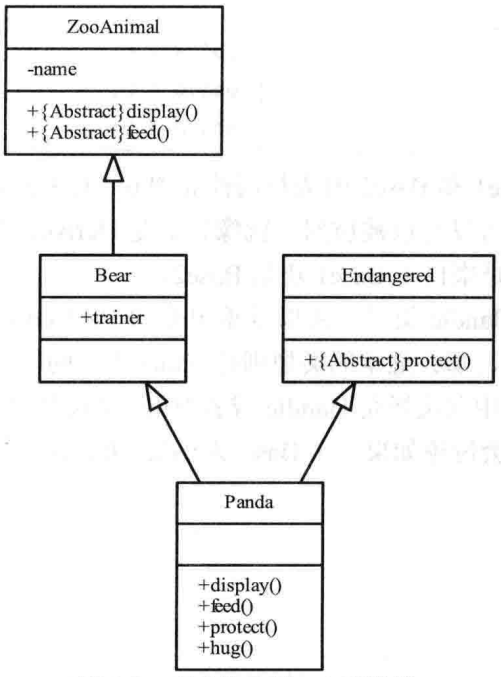


图 8.5 多重继承例：大熊猫类

①Abstract 类见第10章。

8.2.2 什么是多重继承

多重继承，顾名思义，就是包含一个以上的父类的继承，即如图8.6所示的这种结构。

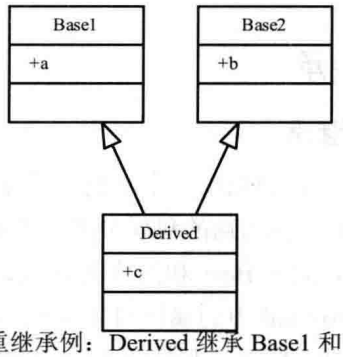


图 8.6 多重继承例：Derived 继承 Base1 和 Base2 基类

两个基类定义如下：

Base1

```
classdef Base1 < handle
    properties
        a
    end
end
```

Base2

```
classdef Base2 < handle
    properties
        b
    end
end
```

MATLAB 中规定，多重继承时要使用符号& 来串接各个父类。下面是 Derived 类的定义：

Derived

```
classdef Derived  <  Base1  & Base2
    properties
        c
    end
end
```

Script

```
obj = Derived();

obj.a = 0;          % 直接访问
obj.b = 0;          % 直接访问
obj.c = 0;
```

Derived 类继承了 Base1 和 Base2 的成员属性 a 和 b，对 Derived 类对象中 property 的访问和以往一样，即 a 和 b 可以直接被访问，就像它们是 Derived 中定义的 property 一样，不用区分该变量的声明到底是来自于 Base1 还是 Base2。

因为两个 Base 都是 Handle 类型，所以继承出来的 Derivedp 也是 Handle 类型的。如果两个 Base 类是 Value 类型，那么继承的类型则是 Value 类型的。

图8.7所示为多重继承中父类皆是 Handle 或者 Value 类的情况。

在第8.2.7小节我们将会讨论如果一个 Base 是 Value 类，另一个 Base 是 Handle 类的情况。

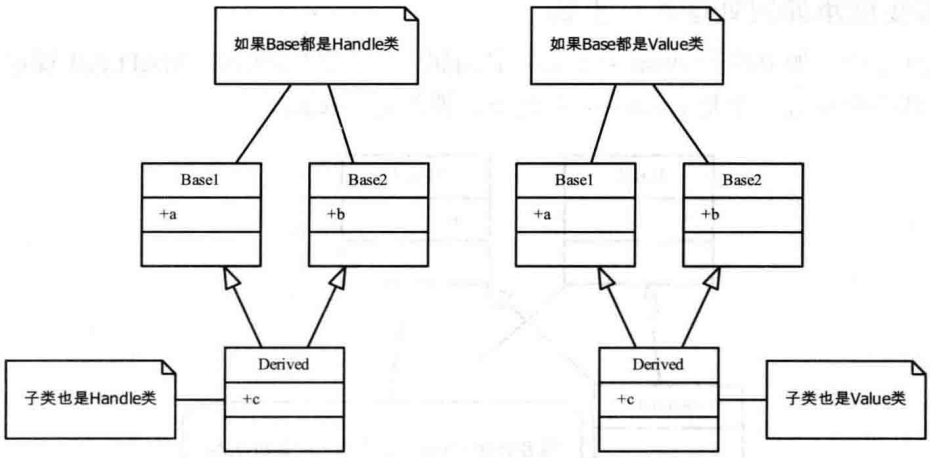


图 8.7 多重继承中父类皆是 Handle 类或者 Value 类的情况

8.2.3 构造函数被调用的顺序是什么

如果用户不明确指定父类 Constructor 的调用顺序，那么 MATLAB 将根据声明的先后顺序来调用父类的构造函数。同样，如果让 MATLAB 自动调用 Constructor，用户提供的 Constructor 必须包括零参数的情况，或者干脆不提供 Constructor。采用前节 ZooAnimal，Bear，Endangered，Panda 类的例子，把各个类简化得只剩下 Default Constructor：

ZooAnimal

```
classdef ZooAnimal < handle
    methods
        function obj = ZooAnimal()
         disp('ZooAnimal CTOR called');
        end
    end
end
```

Bear

```
classdef Bear < ZooAnimal
    methods
        function obj = Bear()
         disp('Bear CTOR called');
        end
    end
end
```

Endangered

```
classdef Endangered < handle
    methods
        function obj = Endangered()
         disp('Endangered CTOR called');
        end
    end
end
```

Panda

```
classdef Panda < Bear & Endangered
    methods
        function obj = Panda()
         disp('Panda CTOR called');
        end
    end
end
```

注意到 Panda 类的 Constructor 中并没有明确的声明 Constructor 的调用顺序，如果我们声明一个 Panda 对象，得到的输出和我们预计的一样。

Script

```
obj = Panda()
```

Command Line

```
ZooAnimal CTOR called
Bear CTOR called
Endangered CTOR called
Panda CTOR called
```

8.2.4 多重继承如何处理属性重名

多重继承中，如果两个 Base 类中有同名的属性，如图8.8所示，MATLAB 规定这个两个变量必须其中至少有一个是 private 或者两个属性都是 private。

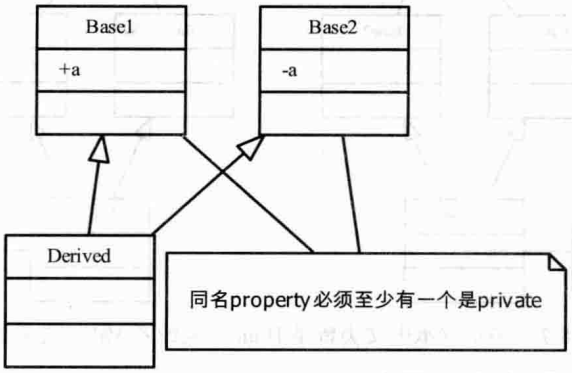


图 8.8 如果多重继承中属性重名：必须至少有一个是 private

Base1

```
classdef Base1  < handle
    properties
        a = 1
    end
end
```

Base2

```
classdef Base2  < handle
    properties(Access = private)
        a = 2;
    end
end
```

上述定义了两个基类，Base1 和 Base2。其中，Base1 的属性 a 是 public 的，并且默认值是 1；Base2 的属性 a 设置成了 private 的，默认值是 2，且 Derived 既继承 Base1 又继承 Base2：

Derived

```
classdef Derived  <  Base1  & Base2
end
```

如果声明一个 Derived 类的对象，其属性 a 的初始值来自 Base1：

Command Line

```
>> obj = Derived()
obj =
  Derived handle
  Properties:
    a: 1
  Methods, Events, Superclasses
```

MATLAB 禁止两个同名的属性都是 public（如图8.9所示），因为这将造成名称的模棱两可，将出现如下错误：

Command Line

```
Error using Derived
Property 'a' cannot be defined in both class 'Base2' and 'Base1'.
```

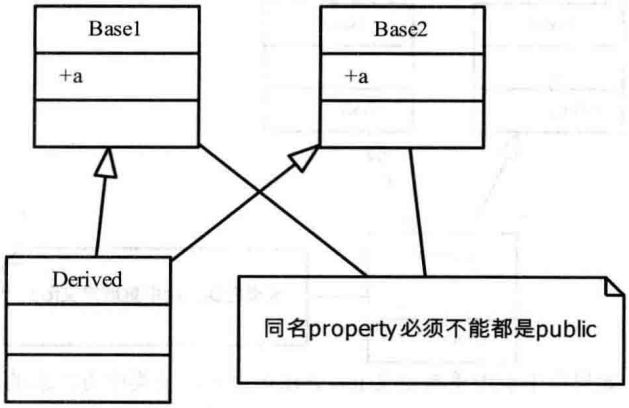


图 8.9 禁止同名属性都是 public

8.2.5 多重继承如何处理方法重名

多重继承中，除了属性重名，还可能存在方法重名的情况。MATLAB 允许各基类中存在同名方法，但必须是下列两种情况之一：

第一种情况：基类中至少有一个方法是 private 方法，如图8.10所示。

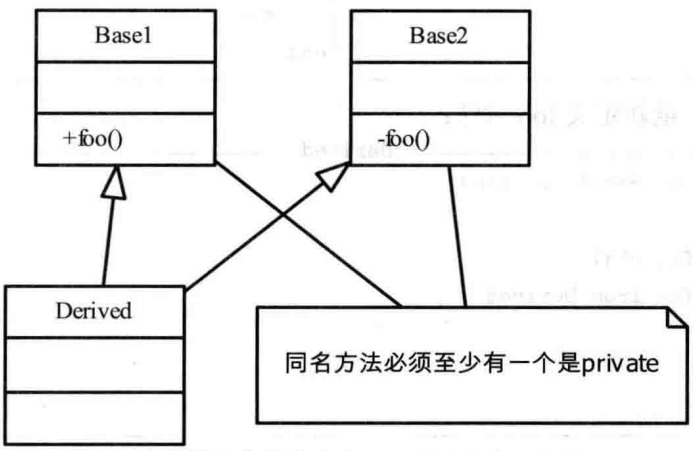


图 8.10 多重继承中方法重名：必须至少有一个是 private

当方法同名时，如果不添加以上限制，将导致下面的调用模棱两可：

Command Line

```
>> obj = Derived();
>> obj.foo() ;  % 错！  ??? 到底要调用哪个函数
```

这是因为两个函数的 Signature 相同，MATLAB 没办法判断用户到底是要调用 Base1 的方法，还是 Base2 的方法，冲突的本质是用户没有提供给同名方法足够的特征使得它们区分开来。

第二种情况：如果用户在 Derived 类中提供另一个同名方法 foo，这个方法将覆盖 Base1 和 Base2 中的 foo 方法，从而消除模棱两可的可能性， 如图8.11所示。

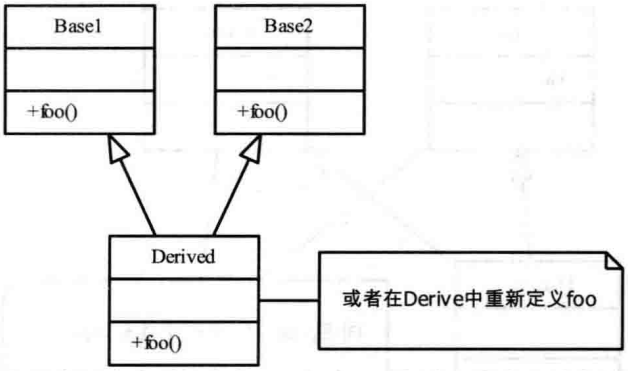


图 8.11 如果在子类中重新定义 foo 方法可以解决父类中方法名的冲突问题

在调用obj.foo() 时，MATLAB 将调用 Derived 类中定义的那个 foo 方法。

Base1

```
classdef Base1  < handle
    methods
        function foo(obj)
           disp('foo from Base1') ;
        end
    end
end
```

Base2

```
classdef Base2  < handle
    methods
        function foo(obj)
           disp('foo from Base2') ;
        end
    end
end
```

Derived 类中，重新定义 foo 方法：

Derived

```
classdef Derived  <  Base1  & Base2
    methods
        function foo(obj)
           disp('foo from Derived') ;
        end
    end
end
```

在命令行中调用的对象的 foo 方法来自 Derived 类：

Script

```
obj = Derived();
obj.foo();
```

Command Line

```
foo from Derived
```

8.2.6 什么是钻石型继承

钻石型继承指的是如图8.12所示的这种两级的继承结构。

简单地说，就是一个基类（在这里是 Base）在继承的层次中多次出现。首先举个例子来说明什么情况下会使用到这种继承。

沿用熊猫的例子，在动物学领域，人们对熊猫到底属于浣熊科（Racoon）还是熊科（Bear）一直都有争论，因为熊猫在外形、大小上像熊，而生态习性上像浣熊。从实用的编程的角度来看，我们可以认为 Panda 同时具备了 Racoon 和 Bear 的一些特征，实际的解决方案

是让 Panda 既继承 Racoon 类，又继承 Bear 类。所以，其 UML 看上去如图 8.13所示。

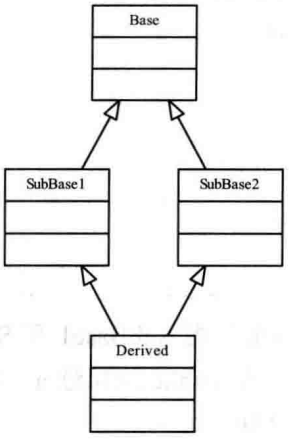


图 8.12 钻石继承的定义

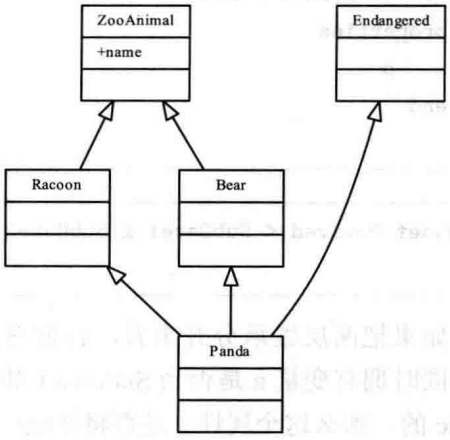


图 8.13 钻石继承一例：熊猫类

上述的 UML 中，ZooAnimal 基类在继承过程中多次出现。它既是 Bear 的基类，又是 Racoon 的基类，而且还是 Panda 的基类。这种结构就叫做钻石型继承。

问题： 钻石型继承是否有重名问题

首先把钻石型继承的问题简化一下，UML 如图8.14所示。

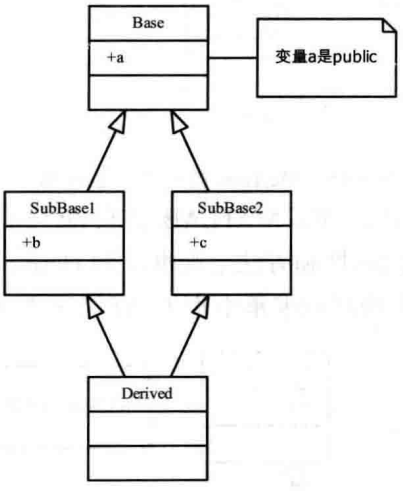


图 8.14 SubBase1 和 SubBase2 都继承了 Base 的 a 属性

程序的代码看上去将是这样的：Base 中的 a 是 public 属性。

Base

```
classdef Base < handle
    properties
        a
    end
end
```

SubBase1

```
classdef SubBase1 < Base
    properties
        b
    end
end
```

SubBase2

```
classdef SubBase2 < Base
    properties
        c
    end
end
```

Derived

```
classdef Derived < SubBase1 & SubBase2
end
```

如果把两层继承分开来看，也许有人会问，第一层继承使得 SubBase1 和 SubBase2 对象将同时拥有变量 a 是否当 SubBase1 和 SubBase2 都具有一个 public 的属性 a，且该属性是 public 的，那么这个属性 a 是否将造成一个二义性？如图8.15所示。

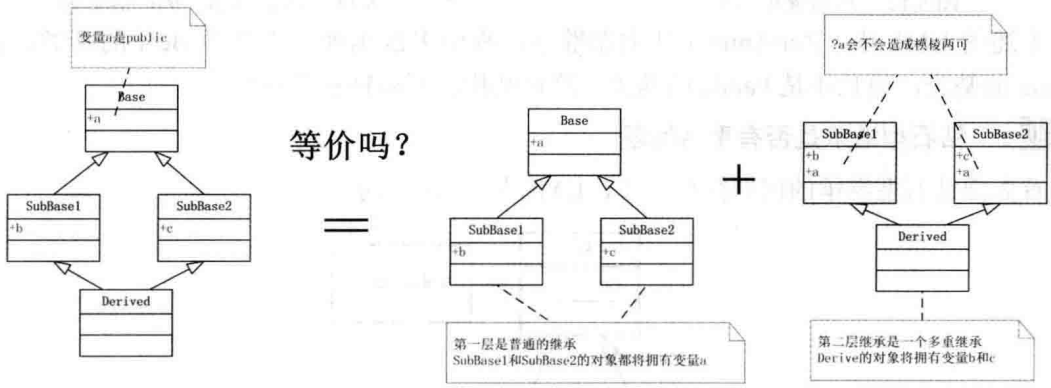


图 8.15 Derived 类存在重名问题吗？

答案是否定的。在这种情况下，MATLAB 会自动判断出 Base 在继承中多次出现，MATLAB 确保只有一个 Base 的属性和方法会被继承到 Derived 类中去。同理，若 Base 中有一个成员方法 foo，在这种钻石形状的继承中也不会存在模棱两可的情况，如图8.16所示。

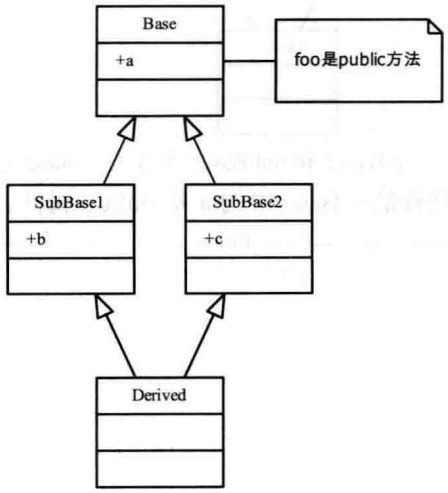


图 8.16 MATLAB 会确保不存在方法 Conflicts

8.2.7 如何同时继承 Value 类和 Handle 类

因为 Handle 类和 Value 类的行为有本质的不同，如果一个子类既继承 Handle 类，又继承 Value 类，那么 MATLAB 将报错。

```
classdef BaseV
end
```

```
classdef BaseH < handle
end
```

如果定义 Derived 同时继承 BaseV 和 BaseH，MATLAB 将会报错，如图8.17所示。

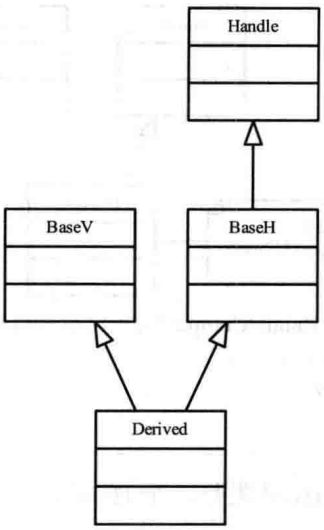


图 8.17 不允许既继承 Value 类又继承 Handle 类

```
classdef Derived < BaseH & BaseV

end
```

```
Command Line
??? Error using ==> Derived
If a class defines super-classes, all or
none must be handle classes.
```

但实际中，确实可能存在着这种情况，需要同时重用 Value 类和 Handle 类中的代码。针对这种需要，MATLAB 提供了一种关键词给 Value 类，叫做 HandleCompatible（适用于 R2011a 及其之后的版本）。只要给 Value 基类添上该关键词，该基类就可以和其他的 Handle 类一起出现在多重继承的上层结构中。例如：

```
classdef(HandleCompatible) BaseV
end
```

该关键词的使用有两个要点：

- 虽然使用了 HandleCompatible 关键词，但是该 BaseV 仍然是一个 Value 类，具有一切 Value 类的行为特点。
- 仅仅在需要既继承 Handle 类，又要继承 Value 类的情况下，才需要把普通的 Value 类用这个 HandleCompatible 关键词声明成这种特殊的 Value 类。

如图8.18所示，本节开始的定义修改成如下形式，就不会有错了。

```
classdef(HandleCompatible) BaseV
end
```

```
classdef BaseH < handle
end
```

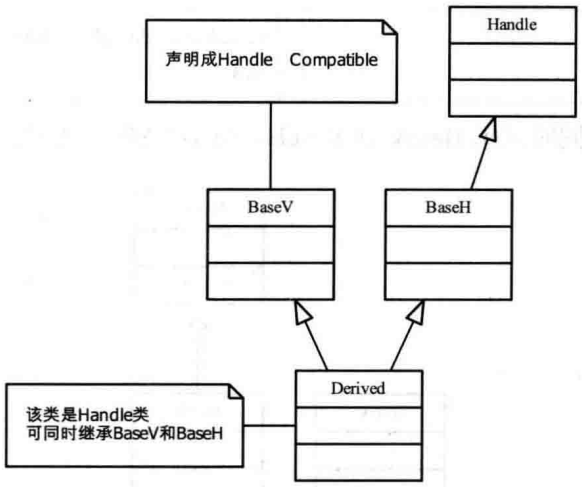


图 8.18 给 Value 类注明 HandleCompatible，它就可以和 Handle 类一起做父类了

```
classdef Derived < BaseH & BaseV
end
```

通过这种方法声明出来的 Derived 类是一个 Handle 类。

Script

```
>> obj = Derived()
```

Command Line

```
obj =
  Derived handle with no properties.
  Methods, Events, Superclasses
```

如图8.19所示，如果同时继承一个 Value 类和 Handle Compatible 的 Value 类，这也是允许的，这和同时继承两个 Value 类没有什么区别，即得到的子类仍然是一个 Value 类，但不是 HandleCompatible 的。

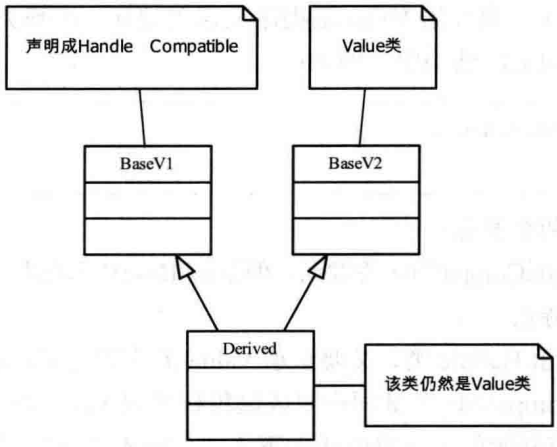


图 8.19 同时继承一个 Value 类和 HandleCompatible 的 Value 类仍是 Value 类

```
classdef(HandleCompatible) BaseV1
end
```

```
classdef BaseV2
end
```

```
classdef Derived < BaseV1 & BaseV2
end
```

Script

```
clear all; clear classes; clc;
obj = Derived()   % Still a Value class
```

Command Line

```
obj =
  Derived with no properties.
  Methods, Superclasses
```

8.3 如何禁止类被继承

定义一个类时，如果使用关键词 Sealed，该类将不能被其他类继承，Sealed 关键词使用的例子可参见第11.7.4小节以及第17.5节。

A.m

```
classdef (Sealed) A
    ......
end
```

或者

A.m

```
classdef (Sealed = true) A
    ......
end
```

如果另外一个类企图继承 A 类，例如：

```
classdef B < A
    ......
end
```

则在声明 B 的对象时，将出现如下错误：

Command Line

```
>> b = B();
??? Error using ==> B
Class 'A' is Sealed and may not be used as a super-class.
```

禁止类被继承的另一种方法是把构造函数声明成 private 的[①]。由于子类对象的建立必须要访问父类的构造函数，而 private 的构造函数将禁止子类的访问，所以该错误会出现在运行时，从而达到禁止继承的目的。当类的设计者不希望子类改变父类的行为，而希望锁定类的行为时，可以把类定义成 Sealed。

①private 的用法参见第2.9节。

第 9 章　类的成员方法进阶

9.1　Derived 类和 Base 类同名方法之间有哪几种关系

9.1.1　Derived 的方法覆盖 Base 的方法

所谓 Derived 类的成员方法覆盖 Base 同名成员方法的意思是：如果 Base 和 Derived 都定义了成员方法 foo（如图9.1所示），声明一个 Derived 类的对象，那么调用该对象的 foo 方法，实际被调用的是 Derived 类中的 foo 方法。Base 类中 foo 方法在外部看来，就像是被 Derived 的 foo 方法所覆盖了。

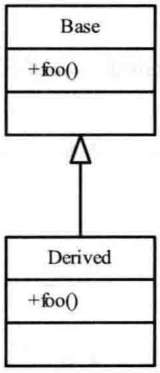


图 9.1　Derived 类中的 foo 定义将覆盖 Base 中的 foo 的定义

Base

```
classdef Base < handle
    methods
        function foo(obj)
            disp('from Base');
        end
    end
end
```

Derived

```
classdef Derived < Base
    methods
        function foo(obj)
            disp('from Derived');
        end
    end
end
```

Script

```
d = Derived();
d.foo()
```

Command Line

```
from Derived
```

9.1.2　Derived 的方法可以扩充 Base 的同名方法

从外部来看，虽然 Base 类中 foo 方法像是被覆盖了，但是在 Derived 类内部还是可以调用 Base 类中的 foo 方法的，如图9.2所示。从效果上来理解，也可以认为 Derived 类中的方法扩充了 Base 类的 foo 方法。

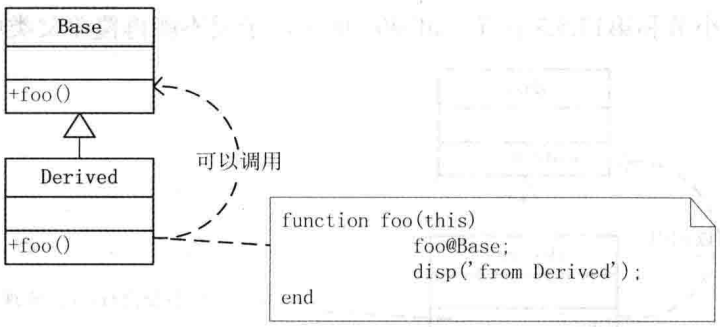


图 9.2 Derived 的 foo 方法内部可以调用 Base 的 foo 方法

Base

```
classdef Base < handle
    methods
        function foo(obj)

            disp('from Base');
        end
    end
end
```

Derived

```
classdef Derived < Base
    methods
        function foo(obj)
            foo@Base(obj); % 调用父类 foo 方法
            disp('from Derived');
        end
    end
end
```

Script

```
d = Derived();
d.foo();
```

Command Line

```
from Base
from Derived
```

注意，对于图9.3所示的双重继承结构，子类只能调用其直接的父类 SubBase 的方法，即 Derived 类不能直接调用 Base 类中的方法。

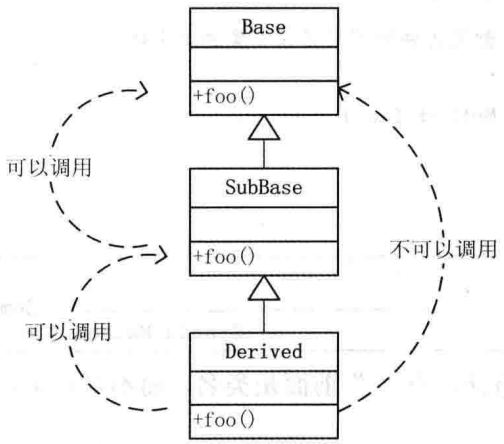


图 9.3 子类只能调用最直接的那个父类中的方法

9.1.3 Base 的方法可以禁止被 Derived 重写

当基类的作者要锁定该类中的某个方法的行为，即确保该方法不被（继承该基类的程序开发者）覆盖时，可以在基类方法中使用关键词 Sealed 加以限制。Sealed 关键词的具体用例

可以参见第11.7.5小节和第17.5.3小节。如图9.4所示，子类不能再覆盖父类中 Seal 的方法。

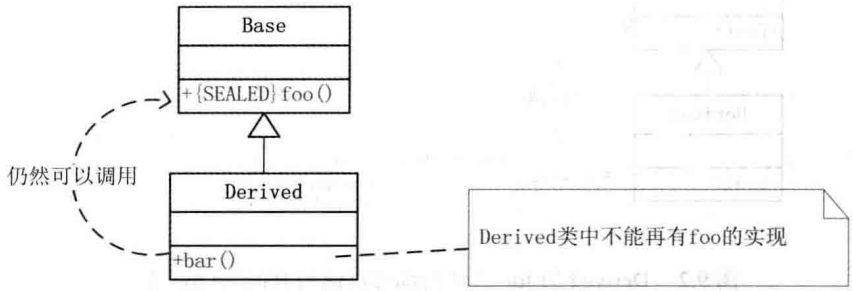


图 9.4 子类不能再覆盖 Base 中 Seal 的方法

Base

```
classdef Base < handle
    methods(Sealed)
        function foo(obj)
            % ......
        end
    end
end
```

Derived

```
classdef Derived < Base
    methods % 子类中不能再定义 foo 方法
        function bar(obj)
        % 可以调用 Base 的 foo 方法
        end
    end
end
```

Seal 关键词的另一用例可见第11.7节。

9.2 什么是静态（Static）方法

静态（Static）方法也叫做类方法，它为类服务，其最明显的特征是不需要对象[①]就能调用它。可以使用 Static 关键词来声明一个静态方法：

A

```
classdef A < handle
    methods(Static)    % 静态方法的定义必须在类的定义体内
        function foo()
            disp('Static Method foo')
        end
    end
end
```

Script

```
A.foo()
```

Command Line

```
Static Method foo
```

注意到上述调用 foo 方法，点“.”前面是类名，而不是对象名，且静态方法的定义必须放在类的定义体中间。

因为静态方法没有把对象当做参数，所以定义的静态方法既不能访问对象的一般属性，又不能调用类的一般方法，因为静态方法没有其他的途径得到 obj，而访问对象的属性和调用类的一般方法一定要指明对象。因为类的 Constant Property 同样也是为类服务的，而不属

①相较之下，需要通过对象调用的方法叫做实例方法。

于某个对象，所以静态方法可以访问类的 Constant Property，如图9.5所示。

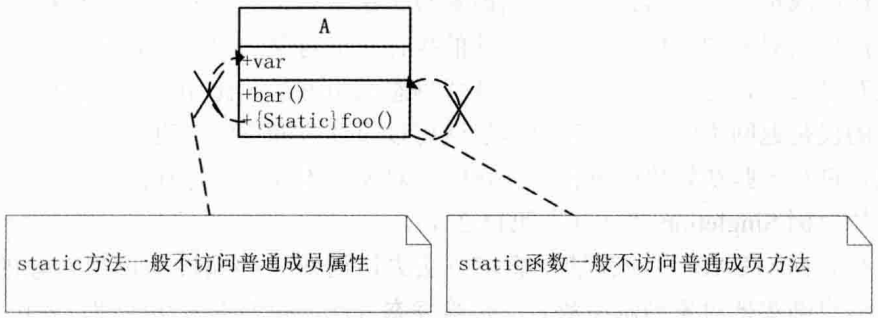


图 9.5 静态方法一般不访问成员属性和成员方法

A

```
classdef A < handle
    properties
        var
    end
    methods
        function bar(obj)
        end
    end
    methods(Static)
        function foo()
            % 由于该静态方法 foo 没有把对象作为参数
            % foo 方法体内既不能访问 obj.var, 也不能访问 obj.bar()
        end
    end
end
```

但是，类中的普通成员方法可以访问静态方法，调用规则和前面的例子一样，在方法前面加上类名即可。

A

```
classdef A < handle
    methods
        function bar(obj)
            A.foo();      % 成员方法内部调用类的静态方法
        end
    end
    methods(Static)
        function foo()
            disp('Static method foo')
        end
    end
end
```

Script

```
obj = A();
obj.bar()
```

Command Line

```
Static method foo
```

静态方法的设计思路是：它为整个类服务，而不是为某个特定的对象服务。如果一个成员方法需要访问成员属性，那么把对象当做参数传递给它是必要的，但有时有些成员方法确实不需要访问任何对象的属性，或者该方法的执行和类对象的内部状态无关，这时把对象作为参数就没有必要了，这些方法就可以设计成静态成员方法。比如，可以设想存在这样的方法，它的目的仅是返回类的名字，这个方法就可以设计成静态的。再比如，在产生一个对象之前，如果要进行一些初始化的操作，在这时，对象尚不存在，这样的方法也可以设计成静态函数，比如单例 Singletion 模式（见第15.2节）。

须要指出，MATLAB 并没有禁止静态方法去访问对象的属性和调用对象的实例方法。所以，如果用户确实把对象当做参数之一传给静态方法，在静态方法内部，就可以访问到对象的属性和成员方法。不过，这不是静态方法的常见用法，并且这样定义的静态方法没有太大的意义，所以这里不加介绍。

9.3 同一个类的各个对象如何共享变量

9.3.1 什么情况下各个对象需要共享变量

如果想让各个对象共享数据，最简单的方法就是声明 global 变量，但这样不方便管理，在规模稍大的程序中还会出现变量名称冲突。如果能够把一个数据“当做 global 变量去存储”，但又能被封装在类的内部，且和这个类的各个对象相联系，这样的处理方法是最佳的。

在 C++ 或者 Java 中，同一类的对象之间可以共享变量，这个是通过把成员变量声明成 static 来实现的。在 MATLAB 面向对象语法中提供 static 的支持不是什么难题，但是由于 MATLAB 多年来的语法习惯，提供 static 变量会造成旧有代码（MATLAB OOP 产生之前的代码）的兼容问题。为了避免这样的混乱，MATLAB 中并没有像 C++、Java 中对象的 static 成员变量，但是类似 C++ 和 Java 的 static 变量的功能可以通过两种方法来实现。

9.3.2 如何共享常量属性

如果在对象的生存期中，共享变量的值不变，那么就可以把该成员属性声明成 Constant。声明成 Constant 的属性被该类的所有的对象所共有。换一种说法，就是无论有多少个类的对象，内存中该 Constant 属性只有一个。

可以用一个简单的例子实验一下：下面代码的 ValueClassA 类中有一个1000 × 1000 的 double 矩阵声明成了 Constant。ValueClassB 类是同样的矩阵，但没使用 Constant 关键词。

ValueClassA

```
classdef ValueClassA
    properties(Constant)
        a = rand(1000,1000);
    end
    properties
        b = 0;
    end
end
```

ValueClassB

```
classdef ValueClassB
    properties
        a = rand(1000,1000);
    end
    properties
        b = 0;
    end
end
```

每个类声明两个对象，并且使用 whos 检查大小如下：

Script

```
clear
obj1 = ValueClassA();
obj2 = ValueClassA();
obj3 = ValueClassB();
obj4 = ValueClassB();
whos
```

Command Line

```
  Name  Size            Bytes  Class
  obj1  1x1                  64  ValueClassA
  obj2  1x1                  64  ValueClassA
  obj3  1x1             8000064  ValueClassB
  obj4  1x1             8000064  ValueClassB
```

我们看到，声明成 Constant 的矩阵没有计入 obj1 和 obj2 所占用的内存中，因为该 Constant 属性是被类的各个对象所共有的。

9.3.3 如何共享变量

如果要让类的各个对象共享变量，可以把该变量定义成静态成员方法中的 persistent 变量。如下面的例子，counter 是 static 方法 increase 中的 persistent 变量，用来记录该类一共创建了几个对象，其中 counter 就是被各个对象所共享的变量。

A

```
classdef A < handle
    methods(Static)
        function increase()
            persistent counter ;  % 各个对象共享的数据
            if isempty(counter)
                counter = 1;
            else
                counter = counter + 1;
            end
            disp(['objs =',num2str(counter)]);
        end
    end
    methods
        function obj = A()
           A.increase();
        end

    end
end
```

命令行测试如下：

Script

```
o1 = A();
o2 = A();
o3 = A();
o4 = A();
```

Command Line

```
objs =1
objs =2
objs =3
objs =3
```

问题：为什么 MATLAB 面向对象语言中不提供 static 变量

回答：熟悉 C++ 或者 Java 的读者可能会奇怪，让一个类的各个对象共享数据一般是通过 static 变量来实现的，但是 MATLAB 面向对象语言中并没有明确地提供 static 关键词用以修饰一个变量。

其原因是：MATLAB 长久以来的编程惯例是，在赋值时，变量比方法和类具有更高的优先级。以一个例子来说明这个问题：

```
A.C  = 10 ;
```

这样的语句，一直以来的习惯是，如果工作空间中没有 A 变量时，MATLAB 会构造出一个 struct 变量，且该 struct 中有一个 field 的名字叫做 C，被赋值为 10。为了保证向后的兼容性，为了让用户以前编写的代码不至于失效，MATLAB 语言必须始终支持这种赋值方法。

如果在 OOP 语言中引入 static 变量的支持，并且假设 A 是一个类的名字，而 C 是其中的 static 属性，则对该静态变量的赋值将不可避免地也写作：

```
A.C  = 10 ;
```

考虑到上述两种情况，可能的不兼容的情况也就出现了：如果用户的旧的代码`A.C = 10`的意图是建立一个 struct 变量，而这时，如果恰好 MATLAB 的搜索路径上存在一个叫做 A 的类，且其中恰好有一个 static 属性叫做 C，这将造成混乱。也就是说，用户以前编写的代码在新的 MATLAB 中不能被使用，这就是说，如果添加了这个新功能，将造成向后不兼容，所以 MATLAB 不支持 static 变量。

第10章 抽 象 类

10.1 什么是抽象类（Abstract）和抽象方法

当我们把类看做是数据和操作的集合时，通常会认为该类肯定要被实例化出至少一个对象，因为有了实体，才能对数据进行操作。其实在很多情况下，定义哪些不能被实例化出对象的类也是有价值的，这种类就叫做抽象（Abstract）类。抽象的反义词是具体，这种可以被实例化的类，通常也被叫做具体（Concrete）类。

- □ 从功能上说，抽象类是面向对象编程中的一种特殊的类，该类不能直接用来声明对象，其作用是为子类提供一个规范，比如规定一些子类必须实现的函数。
- □ 从语法上说，包含抽象方法的类叫做抽象类，可以使用关键词 Abstract 来定义抽象方法的方法。

下面的例子中 draw 方法被定义成了 Abstract 方法，所以 Shape 是一个抽象类。

Shape.m

```
classdef Shape < handle
    methods(Abstract)
        draw(obj)
    end
end
```

- □ 定义抽象方法，只需要一行声明，不需要具体的函数代码（不需要 function 和 end 的关键词），并且该类的子类包括有一个同名的、非抽象（具体）的方法。
- □ 包括抽象方法的类不可以使用 Sealed 关键词；否则，该类既不能被继承，又不能用来声明实例，定义这样的类没有任何用处。

MATLAB 中 Abstract 属性的默认值是 false。也就是说，如果不显示地使用关键词 Abstract，声明出来的方法默认是非 Abstract（即需要有具体的函数体）的方法，如下面代码的第一个区域的声明所示，第二个区域和第三个区域的格式都可以用来声明抽象方法，其效果是相同的，其中第二种的声明方式更加值得推荐，因为这样的代码更加简洁清楚。

```
......
    methods
        ......
    end
    methods(Abstract)       %   推荐的声明抽象方法的语法
        ......
    end
    methods(Abstract=true)
        ......
    end
......
```

子类在继承抽象类时，必须具体实现各个抽象函数，否则子类仍然是抽象类。比如 Circle 类继承 Shape 的基类，则 Circle 要实现 draw 方法：

Circle.m

```
classdef Circle < Shape
    methods
        function draw(obj)
            ......          % 具体的 draw 的函数实现
        end
    end
end
```

在 UML 中，一般用斜体来表示抽象方法，因为斜体往往难以辨认。本书中采用如下惯例，在方法的前面加上一个 Abstract 来明确提示读者这是一个抽象方法，如图10.1所示。

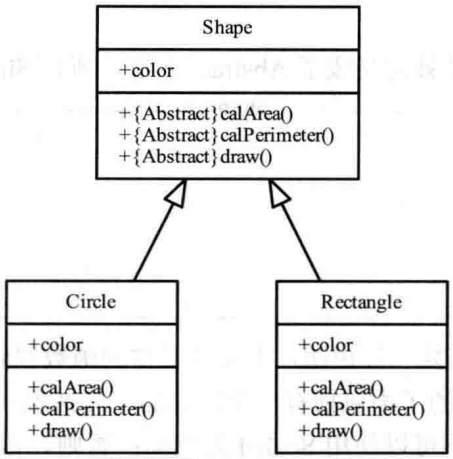


图 10.1 基类 Shape 中把 draw 声明成了 Abstract，子类 Circle 中一定要给出具体定义

10.2 为什么需要抽象类

10.1节介绍了什么是抽象类和抽象方法，这节用一个例子来说明抽象类的用处。要求设计一个类，用来描述一个几何形状，比如圆、长方形。这些几何形状有共同的属性，如颜色、集合、尺寸。这些形状也有着共同的方法和操作，比如计算面积、周长，以及作图。于是，可以设计一个叫做 Shape 的基类来概括这些具体形状的共性，比如，该 Shape 类中将包括计算面积的方法 calArea，计算周长的方法 calPerimeter 和作图指令 draw。虽然该 Shape 类包括了大家都需要的方法，但是这些方法，显然都无法在 Shape 类中被实现，因为还不知道具体的形状，所以就没有办法计算几何物体的面积、周长等，所以 Shape 类中的这些方法是必须抽象的。这里只声明，并不定义，把定义留到子类中完成，如图10.2所示。从类的角度，Shape 类必须是抽象的，不能声明出对象来，也很好理解，因为现实中，不存在一个几何物体（对象）没有形状。

注意，图10.2的例子，对在第2.6.4小节提到的多态（Polymorphism）的概念 是一个更形象的解释，在这里，多态指的就是多种形态，当我们给对象提供一条指令，如`obj.draw`，面

向对象的程序将根据 obj 的不同类型，画出不同形状的几何物体。

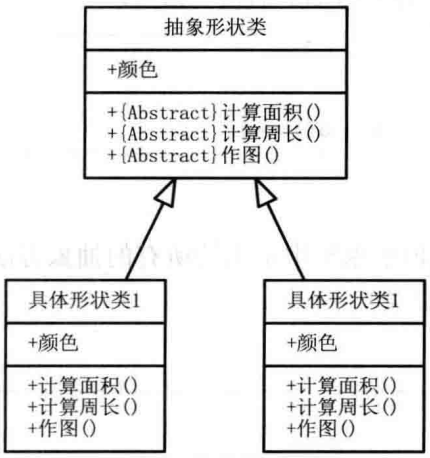


图 10.2 基类中的方法是抽象的，因为没办法实现

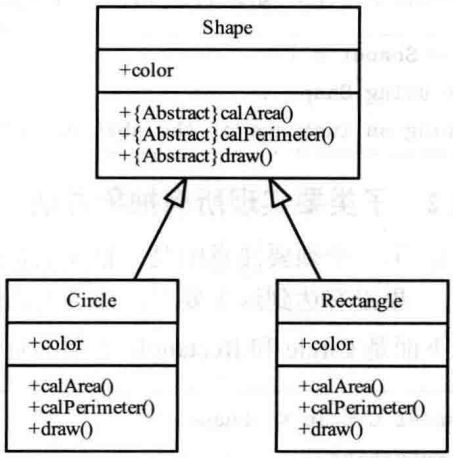


图 10.3 Shape 类中的三个方法必须声明成 Abstract

10.3 如何使用抽象类

10.3.1 抽象类不能直接用来声明对象

以10.2节定义的 Shape 类为基类，从 Shape 类继承出两个具体的子类，分别是圆和长方形。这两个类有自己特有的属性，如 Circle 类具有属性 radius，Rectangle 类具有属性 width 和 height。三个类的 UML 如图10.3所示。

Shape 基类的代码：

Shape.m

```
classdef Shape < handle
   properties
      color
   end
   methods
       function obj = Shape(color)
          obj.color = color;
       end
   end
   methods(Abstract)
      calArea(obj);
      calPerimeter(obj);
      draw(obj);
   end
end
```

- □ 抽象类可以有构造函数，只是不能利用这个构造函数声明出对象来，该构造函数，一般被子类构造函数所调用。
- □ 抽象方法声明中参数的个数并不是一个严格的限制。也就是说，子类中只需要实现

同名的方法即可，参数的数目不一定要和父类中的一致。

在命令行上，如果直接调用抽象类的构造函数，将给出如下的错误信息：

Command Line

```
>> s = Shape('b')
Error using Shape
Creating an instance of the Abstract class 'Shape' is not allowed.
```

10.3.2 子类要实现所有抽象方法

□ 第二个须要注意的是：抽象类的子类必须实现抽象类中定义的所有的抽象方法，如果没有达到这个要求，该子类仍然是一个抽象类。

下面是 Circle 和 Rectangle 子类的具体定义：

```
classdef Circle <  Shape
   properties
      radius
   end
   methods
       function obj = Circle(radius,color)
          obj = obj@Shape(color);
          obj.radius  = radius;
       end
       function area = calArea(obj)              % calArea 计算圆的面积
            area = pi*obj.radius^2;
       end
       function perimeter = calPerimeter(obj)  %  calPerimeter 计算圆的周长
           perimeter = 4*pi*obj.radius;
       end
       function draw(obj)                        %  draw 画圆
           [x,y] = pol2cart(linspace(0,2*pi,100),ones(1,100)*obj.radius);
           plot(x,y,obj.color);
           axis square;
       end
   end
end
```

□ 在 Circle 的 Constructor 中，可以调用父类的 Constructor 去初始化父类的成员变量。

Circle 类的使用范例如下，即先声明一个 Circle 对象 c，然后依次调用其成员方法：

Script

```
c = Circle(3,'b');
area = c.calArea()
perimeter = c.calPerimeter()
c.draw()
```

Command Line

```
area =
   28.2743
perimeter =
   37.6991
```

图10.4所示为圆形对象的 draw 方法的具体结果。

另一个子类 Rectangle 对于 calArea、calRectangle 和 draw 有着完全不同的实现。例如：

Rectangle

```
classdef Rectangle < Shape
    properties
        width
        height
    end
    methods
        function obj = Rectangle(width,height,color)
           obj = obj@Shape(color);
           obj.width = width ;
           obj.height = height;
        end
        function area = calArea(obj)                % calArea 计算长方形的面积
            area = obj.width*obj.height;
        end
        function perimeter = calPerimeter(obj)      %  calPerimeter 计算长方形的周长
           perimeter  = 2*(obj.width + obj.height);
        end
        function draw(obj)                          %  draw 画长方形
            rectangle('Position',[0,0,obj.width,obj.height]);
        end
    end
end
```

下面是 Circle 类的使用范例，即先声明一个 Rectangle 对象 r，然后依次调用其成员方法：

Script

```
r = Rectangle(3,4,'r');
area = r.calArea()
perimeter = r.calPerimeter()
r.draw()
```

Command Line

```
area =
    12
perimeter =
    14
```

图10.5所示为长方形对象的 draw 方法的具体结果。

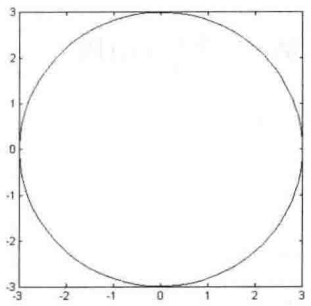


图 10.4 圆形对象的 draw 方法的具体结果

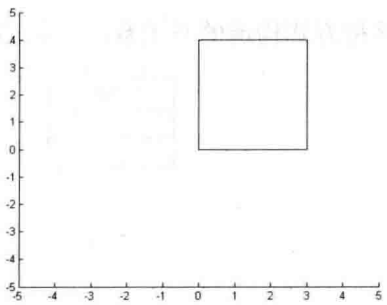


图 10.5 长方形对象的 draw 方法的具体结果

第 11 章　对象数组

11.1　如何把对象串接成数组

面向对象编程中最基本的变量是对象，当程序变得复杂需要使用大量的对象时，不可避免地，就需要把多个对象组合成一个集合。这一章专门介绍如何把多个对象组合起来。

现在假设有一个简单的 Square 类（方形），类中有一个成员变量 a，表示正方形的边长。注意，这个定义中没有显示地提供 Constructor，所以 MATLAB 将自动给这个类提供一个零参数的缺省 Constructor。

Square

```
classdef Square < handle
    properties
        a
    end
end
```

先回顾一下普通变量的串接。在 MATLAB 中，可以使用方括号[ ]把已有的变量组合起来，使其成为一个数组，比如下面的三个 double 的变量，组合在一起构成一个 double 数组 array：

Script

```
a1 = 1;
a2 = 2;
a3 = 3;
array=[a1,a2,a3] ;
```

这种方式叫做串接（Array Concatenation）。MATLAB 对象也可以使用 concatenation 操作。下面的代码将三个方型对象串接成了一个对象数组 objArray，其大小是 1 × 3：

Script

```
b1 = Square();
b2 = Square();
b3 = Square();
objArray = [b1,b2,b3];
```

用这种方式构造的对象数组，适用于对象数量较少的情况，如图11.1所示。

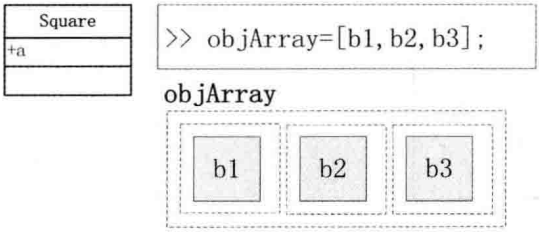


图 11.1　对象数组中放了三个方型对象

对 objArray 中对象元素的访问和对普通数组中元素的访问是一样的。比如objArray(1)将返回数组中的第一个对象，在此基础上，使用点 + 属性名称的语法，可以访问对象元素的属性，例如：

script

```
>> objArray(1).a = 10 ;      % 赋值
>> objArray(1).a             % 访问
ans =
    10
```

11.2 如何直接声明对象数组

如果对象的数目很多，那么象第11.1节那样，先构造出全部的对象，再把这些对象串接起来构造成对象数组，效率就很低了。这种情况下，可以直接声明一个对象数组。回顾声明一个普通的含有 10 个 double 的数组，可以直接给第 10 个元素赋一个初值，然后让 MATLAB 自动对数组的其余部分进行扩展（Expansion）。在下面的代码中，MATLAB 自动生成了一个 1×10 的数组，并把第 10 个元素置为 1，其余为没有被赋予初值的元素，用 0 作为初值[①]。

Script

```
>> array(1,10) = 1
array =
     0     0     0     0     0     0     0     0     0     1
```

对象数组的扩展也是一样的，只不过等式的右边赋的初始是一个对象。

Script

```
objArray(1,10)= Square();
```

上述命令中，MATLAB 生成了一个含有 10 个对象的数组，可以直接使用下标算符访问对象数组中的对象以及赋值。

Script

```
>> objArray(1)                    % 访问
ans =
  Square handle
  Properties:
    a: []
  Methods, Events, Superclasses
>> objArray(1).a = 10 ;           % 赋值
```

MATLAB 具体是如何扩展对象数组的呢？现在我们扩充一下 Square 类的定义，显示地定义一个 Constructor，该 Constructor 可以接收一个输入，并作为属性 a 的初值；也可以接收零个参数，这时 Constructor 把属性 a 初始化为 1。

Square.m

```
classdef Square <handle
    properties
        a
    end
```

①0 是 double 类对象的默认值（Default Value）。

```
    methods
        function obj = Square(val)
            if nargin == 1      % 如果给 Constructor 提供了参数
                obj.a = val;
            elseif nargin == 0  % 参数数目为零
                obj.a = 1 ;
                disp('default CTOR called'); % 该 disp 语句用来标记该部分被调几次
            end
        end
    end
end
```

下面清除工作空间中之前的类 Square 的定义重新声明一个 1×10 的对象数组：

Command Line

```
>> clear all
>> clear classes
>> objArray(1,10) = Square(5);
default CTOR called             % 命令行显示缺省构造函数只被调用了一次
```

可以使用objArray.a 的方法向量化地检查数组中每个对象的属性 a 的值

Command Line

```
>> objArray.a
ans =
     1                  % 调用了 Default CTOR Square()
ans =
     1
ans =
     1
ans =
     1
ans =
     1
ans =
     1
ans =
     1
ans =
     1
ans =
     1
ans =
     1
ans =
     5                  % 调用了 Square(5)
```

从结果中发现，MATLAB 仅对第 10 个元素的构造采用了Square(5)，而其余都使用了Square()。如图11.2所示，其原因是用户调用

Command Line

```
>> objArray(1,10) = Square(5);
```

则 MATLAB 解释器会把这个命令翻译成如下的指令：

□ 对第 10 个元素调用构造函数 Square(5)。

□ 对其余的 1 到 9 个元素调用缺省的构造函数。

事实上，因为第 1 到 9 个元素都会是一样的，所以调用 9 次 Constructor 是没有必要的。其实，缺省 Constructor 只被调用了一次，产生了一个对象，其余 8 个对象都是内部直接拷贝[①]。这也就是为什么命令行输出第 4 行“default CTOR called”的消息只出现了一次的原因。

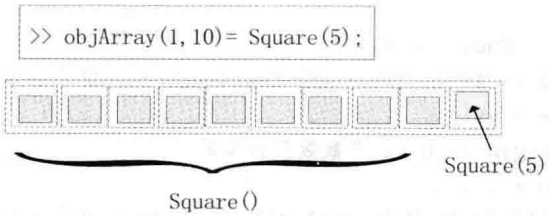


图 11.2 最后一个位置上对象的创建调用了 Sqaure(5)，其他位置只调用了一次 Default CTOR

如果 Square 类没有提供接收零参数的 Default Constructor，比如类的定义是：

Square

```
classdef Square <handle
    properties
        a
    end
    methods
        function obj = Square(val)
            obj.a = val;
        end
    end
end
```

声明一个同样的 1 × 10 数组将出现如下错误：

Command Line

```
>> clear classes
>> objArray(1,10) = Square(5);
??? Input argument "val" is undefined.
```

这是因为 MATLAB 在产生在 objArray 中第 1 到 9 个对象时，对 Square 的构造函数调用格式是Square()，只是没有提供 val，所以obj.a = val ；行会出错显示 val 没有定义[②]。

上面的例子中，扩展的对象数组是 Handle 类的对象数组，在数组扩展中，MATLAB 在内部直接拷贝的对象，尽管其内部数据的数值相同，但如果比较被直接拷贝的元素，它们的 Handle 仍然是不同的。

①调用函数的成本要高于直接拷贝的成本。

②如果希望给对象数组中所有的对象的 a 属性都赋初值 5，可以使用 for 循环。

Command Line

```
>> objArray(1,10) = Square(5);
>> objArray(1) == objArray(2)
ans =
     0
```

如果我们扩展的是 Value 类的对象数组，例如：

Square 是一个 Value 类

```
classdef Square
    properties
        a
    end
    methods
        function obj = Square(val)
            if nargin == 1      % 如果给 Constructor 提供了参数
                obj.a = val;
            elseif nargin == 0  % 参数数目为零是
                obj.a = 1 ;
                disp('default CTOR called'); % 该 disp 语句用来标记该部分被调几次
            end
        end
        function result = eq(input1,input2)
            result = (input1.a == input2.a) ;
        end
    end
end
```

注意，因为 Value 类对象没有定义 == 运算符，所以这里为了比较两个对象，还重载了 == 运算符，详见第12.7节。现在使用上述 Value 类的定义，我们发现，扩展出来的对象数组被直接拷贝的那些元素是相同的。

Command Line

```
>> objArray(1,10) = Square(5);
>> objArray(1) == objArray(2)
ans =
     1
```

11.3 如何使用 findobj 寻找特定的对象

当程序中对象的数量增多时，查找一个特定的对象就是一项常见的任务了。最简单的查找方法是，使用循环遍历集合中的每一个对象。除此之外，还可以使用内置的 findobj 函数来高效地查找集合中的某个对象。为了演示如何使用这个函数，首先构造一个 PhoneBook 类，类中的属性是姓名和电话号码：

PhoneBook.m

```
classdef PhoneBook < handle
    properties
        name
```

```
        number
    end
    methods
        function o = PhoneBook(n,p)
            o.name = n;
            o.number = p;
        end
    end
end
```

使用串接语法，构造一个简单的对象数组

```
PBook =[PhoneBook('Jack', 508000001),
        PhoneBook('Loren',508000002),
        PhoneBook('Doug', 508000002)];
```

首先演示使用 findobj 寻找属性name = Jack 的对象，语法如下，结果返回的是该对象数组中的第一个对象。

Script

```
o = findobj(PBook,'name','Jack')
```

Command Line

```
o =
  PhoneBook handle
  Properties:
      name: 'Jack'
    number: 508000001
  Methods, Events, Superclasses
```

再寻找电话号码等于 508000002 的对象，注意到 Loren 和 Doug 的号码都是 508000002，所以返回的结果是一个 1×2 的对象数组。

Script

```
o = findobj(PBook,'number',508000002)
```

Command Line

```
o =
  2x1 PhoneBook handle
  Properties:
    name
    number
  Methods, Events, Superclasses
```

这里返回了两个对象，可以利用逻辑'-and' 进一步给出更细致的查找条件。比如，需要查找电话号码等于 508000002 并且属性name = Loren 的对象，结果将返回的是数组中的第二个对象。

Script

```
o = findobj(PBook,'name','Loren',...
                '-and','number',508000002)
```

Command Line

```
o =
  PhoneBook handle
  Properties:
      name: 'Loren'
    number: 508000002
  Methods, Events, Superclasses
```

查找条件中，还可以用逻辑或 '-or' 关键词，下面的命令用于查找要么姓名等于 Jack，要么电话号码等于 508000002 的对象。结果数组中三个对象都满足这个条件，所以 findobj 的结果是一个 1 × 3 的对象数组。

Script

```
o = findobj(PBook,'name','Jack',...
                  '-or','number',508000002)
```

Command Line

```
o =
  3x1 PhoneBook handle
  Properties:
    name
    number
  Methods, Events, Superclasses
```

11.4 如何利用 Cell array 把不同类的对象组合到一起

假设有如下两个独立的类 Square 和 Circle：

Square

```
classdef Square < handle
    properties
        a    % 边长
    end
end
```

Circle

```
classdef Circle < handle
    properties
        r    % 半径
    end
end
```

当尝试使用前面介绍的方法，使用方括号 [ ] 把两个对象 Concatenate（串接）到一起时，将出现右边的错误：

Script

```
o1 = Square();
o2 = Circle();
oArray =[o1,o2];
```

Command Line

```
??? Error using ==> horzcat
The following error occurred converting
from Circle to Square:
Error using ==> Square
Too many input arguments.
Error in ==>
oArray =[o1,o2];
```

出现错误的原因是，MATLAB 规定对象数组中的元素的种类必须一致，如果不一致，当这里的 o1 和 o2 是不同种类的对象时，MATLAB 就会尝试把一个对象转换成另一个，如果找不到可以使用的对象转换函数，MATLAB 就会报错，如图11.3所示。

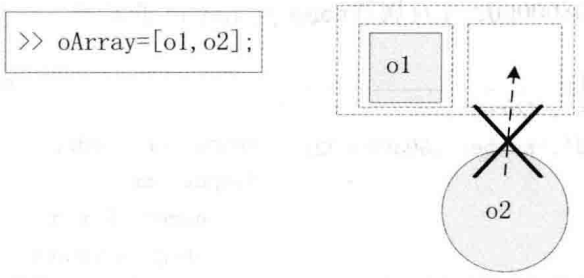


图 11.3 第一个位置放的是方对象，数组类型就是方对象，圆对象不经转换，不能放入该数组

最简单的解决方法是，使用元胞数组（Cell Array）。Cell Array 是 MATLAB 中专门用来存放不同种类数据的工具，它的功能当然也延续到了面向对象编程中。也可以很简单地使用花括号 { } 把两个对象串接（Concatenate）到一起组成 Cell array。

Script

```
o1 = Square();
o2 = Circle();
oCell = {o1,o2}
```

Command Line

```
oCell =
    [1x1 Square]    [1x1 Circle]
```

Cell 数组可以放置不同类型的对象，如图11.4所示。

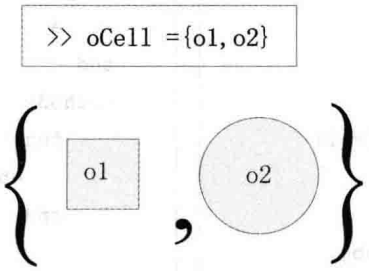


图 11.4 Cell 数组可以放置不同类型的对象

对于 Cell Array 中的单个对象的访问和普通 Cell Array 的访问是一样的，也是使用花括号 { } 加下标的形式。例如：

Command Line

```
>> oCell{1}.a = 10;   % 访问 Cell Array 中的一个元素
```

使用 Cell 来取代 array 盛放对象简单易用，但是有一定的缺陷。比如，如果恰好两个类中都有一个属性叫做 a：

A

```
classdef A < handle
    properties
        a
    end
end
```

B

```
classdef B < handle
    properties
        a
    end
end
```

则我们没有办法利用类似第11.2节中提到的以向量化地方式来集中访问数组中的对象内部的元素。

Script

```
o1 = A();
o2 = B();
oCell = {o1,o2}
oCell.a
```

Command Line

```
Attempt to reference field of non-structure
array Error
```

但是，对于很多用户来说，这种向量化地访问是一个重要的功能。我们在第11.7节介绍一种最佳的解决方案，在此之前，先介绍如何转换对象类型的办法。

11.5 什么是转换函数

转换函数就是一种类的方法，负责把该类的对象转换成其他类的对象。如果要把A对象转成B类对象，那么就在A类中定义一个方法叫做B，该方法内部将完成类型转换的实际工作，比如转换数据等，并返回一个新的B对象。

下面的代码中，在Circle类中定义了一个转换方法叫做Square。简单起见，该转换函数仅仅是把Circle的属性r赋值给新的Square对象的属性a：

Square

```
classdef Circle < handle
    properties
        r
    end
    methods
        function obj = Circle(val)
            obj.r = val;
        end
        function s0 = Square(obj)
            s0 = Square(obj.r);
            disp('Converter called');
        end
    end
end
```

Circle

```
classdef Square < handle
    properties
        a
    end
    methods
        function obj = Square(val)
            obj.a = val ;
        end


    end
end
```

转换函数可以显示地被调用。下面的代码是先声明一个Circle对象，再调用其转换函数得到一个Square对象。

```
>> c1 = Circle(10);
>> c2 = c1.Square()              % 调用 Circle 类的转换函数
Converter called                 % 输出表示转换函数确实被调用
c2 =
  Square with properties:        % 得到一个新的 Square 对象
    a: 10
```

如果我们回忆Constructor的定义（第2.5.1小节）：“Constructor和类的名称相同，有且只能有一个返回值，是唯一创建一个新的对象的方式”。从这个角度，其实可以将转换函数理解成对Constructor方法的重载（第12章），或许下面这种调用方式可以帮助读者更好地理解这句话：

```
>> c1 = Circle(10);
>> c2 = Square(c1)              % 这里到底调用了什么方法
Converter called                 % 输出表示转换函数确实被调用
c2 =
  Square with properties:        % 得到一个新的 Square 对象
    a: 10
```

其中第 2 行看似调用的是 Square 类的 Constructor，实际调用的是 Circle 类的 Square 转换函数。这是由 MATLAB 的 Dispatching 规则所决定的。

11.6 如何利用转换函数把不同类的对象组合到一起

转换函数更常见的用法是在隐式转换中。为了解释隐式转换，我们先仔细讨论 MATLAB 是如何处理下述命令的：

Script

```
o1 = Square();
o2 = Circle();
oArray =[o1,o2];
```

其中， `oArray =[o1,o2]`；其实被分成了两步执行：第一步是在该数组的第一个位置中填上 Square 对象，即

```
oArray(1) = Square() ;
```

并且从此规定这个数组中只能填放 Square 类的对象[①]；第二步给对象数组 oArray 的第二个位置赋值，并且等式的左边（RHS）是类型为 Circle 的对象。

```
oArray(2) = Circle();
```

对于数组中元素的赋值，MATLAB 会首先检查等式左边的值的类型和右边（LHS）的类型是否一致，在这里，LHS 的类型是 Square 类，而 RHS 的类型是 Circle 类。由于 Circle 和 Square 不是同一类的对象，按照规定，普通对象数组中只能存放相同类型的对象，所以这里需要调用 Circle 对象的转换函数。把 Circle 对象转换成 Square 对象。如果 Circle 类没有定义转换函数，MATLAB 会尝试把 RHS 直接作为参数提供给 Square 的构造函数，如果 Square 的构造函数无法处理这种参数，则 MATLAB 报错。转换函数可以把圆形对象转换成方形对象，如图11.5所示。

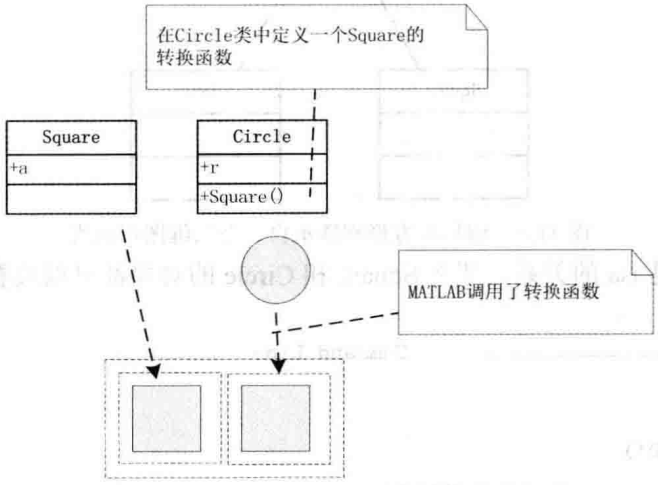


图 11.5 转换函数把圆形对象转换成了方形

①假设 Square 和 Circle 之间优先级没有区别。

定义了转换函数的 Circle 类的对象就可以和 Square 类的对象放到一起组成对象数组了。例如：

Script

```
o1 = Square(10);
o2 = Circle(10);
oCell  =[o1,o2];
```

Command Line

```
Converter called                    % 转换函数被调用
oCell =
  1x2 Square handle                 % Square 数组
  Properties:
    a
  Methods, Events, Superclasses
```

如果用户构造对象数组的本意就是让内部所有的元素都属于同一类，那么使用转换函数是正确的做法。如果用户只是想把各种对象组合到一起而没有要改变它们类型的意思，那么使用转换函数，就是画蛇添足了。下一节，将介绍如何不使用转换函数把不同类的对象组合到一起。

11.7 如何用非同类（Heterogeneous）数组盛放不同类对象

11.7.1 为什么需要 Heterogeneous 数组

通过前面的介绍，已经理解了 MATLAB 规定数组中必须放置同类对象的问题，因为从语义上来说，“圆形对象数组”中确实不应该允许存放“方形的对象”。但是，当继承存在的情况下，这个规定是否合理就需要再仔细思考了，比如图11.6所示的这种情况，即当 Square 和 Circle 都继承自一个基类 Shape2D 的情况。

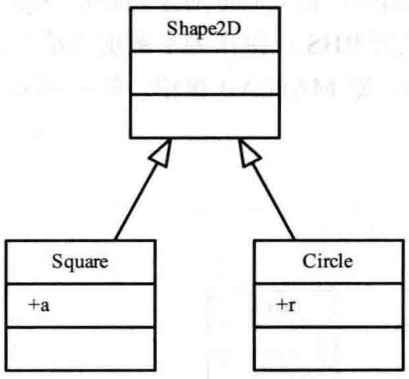


图 11.6 圆形和方形都继承自一个二维图形基类

首先根据继承是 isa 的关系，那么 Square 和 Circle 的对象都可以被看做是二维的形状对象：

Command Line

```
>> sObj = Square();
>> cObj = Circle();
>> isa(sObj,'Shape2D')
ans =
     1
>> isa(cObj,'Shape2D')
```

```
ans =
     1
```

再从语义的角度，我们可以说，“二维形状对象数组”中应该可以存放所有二维形状的对象。所以圆形对象和方形对象理应可以放到同一个对象数组中去。

针对这种情况，MATLAB 从 2011b 之后提出了一个新的解决方案，即规定：

(1) 只要两个类具有共同的父类；

(2) 并且该父类继承自一个叫做 Heterogeneous 的基类。

那么，从这两个类中声明出来的对象就可以放到同一个数组中去，且不需要任何的转换函数，如图11.7所示。

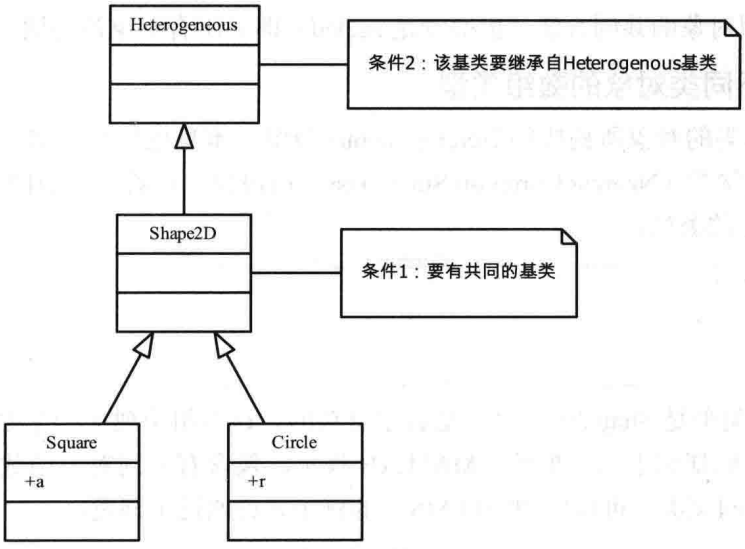


图 11.7 Square 和 Circle 的基类 Shape2D 继承自 Heterogeneous 类

注意：该功能仅存在于 2011b 及其之后的版本，如果想在旧版本中把属于不同类的对象组合到一起还得使用前几节介绍的方法。

具体基类是这样声明的：

Shape2D.m

```
classdef Shape2D < handle & matlab.mixin.Heterogeneous
end
```

其中，matlab 和 mixin 是 Package 的名字；Heterogenesou 是类的名字，该类中重新实现了数组串接和扩展的内置函数，使得其能够支持具有共同基类的对象的数组。

Circle 和 Square 的声明还和以前一样：

Square

```
classdef Square < Shape2D
    properties
        a
    end
end
```

Circle

```
classdef Circle < Shape2D
    properties
        r
    end
end
```

这样声明出来的 Square 对象和 Circle 对象，就可以放到一个数组中去了。

Script

```
o1 = Square();
o2 = Circle();
oArray = [o1,o2]
```

Command Line

```
oArray =

  1x2 heterogeneous Shape2D
```

使用 Heterogeneous 数组有以下优点：

- □ 不用定义 convert 函数，不存在对象之间的相互转换（有时用户也根本不希望进行这样的转换）。
- □ 构造出来的数组可以存放具有共同基类的对象，这也是符合我们的认知习惯的。
- □ 可以使用数组的下标语法，向量化地访问对象的共同特征。
- □ 可以使用对象的共同方法（但必须是 sealed）即对所有对象调用同一种方法。

11.7.2 含有不同类对象的数组类型

具有共同基类的对象所构成的 Heterogeneous 数组，本身也具有类型。该类型由数组中对象的共同最近父类（Nearest Common Superclass）所决定。接着上述的例子，使用 class 命令来查询 oArray 的类型：

Command Line

```
>> class(oArray)
ans =
Shape2D
```

显示该 Array 的类型是 Shape2D，这也是合乎意料的。该数组中放了一个 Circle 对象和一个 Square 对象，它们都属于二维形状。MATLAB 规定，包含有不同对象的数组的类型，总是取它们最近的共同父类，可以用如图11.8所示的例子来说明这个问题。

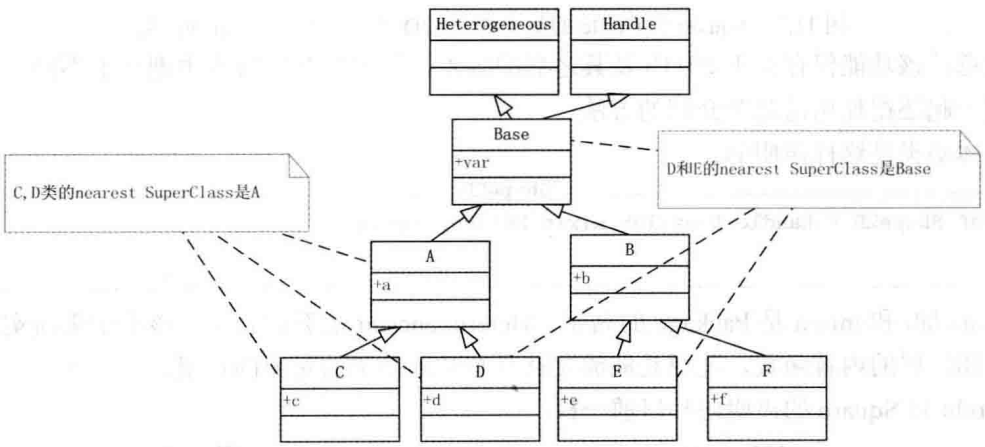


图 11.8 非同类对象数组的类型取决于最近共同父类

在图11.8中，因为总可以找到 A，B，C，D，E 类的组合之间的共同父类，比如 A 与 B 的共同父类是 Base，C 与 D 的共同父类是 A，并且它们都是继承自`matlab.mixin.Heterogeneous`基类，所以可以任意地把 A，B，C，D，E，F 的对象放入 array 中。简单起见，我们使用如下最简单的类的定义：

Base

```
classdef Base < handle & matlab.mixin.Heterogeneous
    properties
       var
    end
end
```

```
classdef A < Base
    properties
       a
    end
end
```

```
classdef B < Base
   properties
       b
   end
end
```

```
classdef C < A
   properties
       d
    end
end
```

```
classdef D <   A
   properties
       d
    end
end
```

```
classdef E < B
    properties
       e
    end
end
```

```
classdef F < B
   properties
       f
   end
end
```

在测试过程中，我们一边构造数组，一边观察该数组类型的变化：

Script

```
array = [C(),D()];class(array)
array = [E(),F()];class(array)
array = [A(),C()];class(array)
array = [A(),E()];class(array)
array = [A(),B()];class(array)
array = [Base(),E()];class(array)
```

Command Line

```
A
B
A
Base
Base
Base
```

这验证了之前的规定，数组的类型总是取对象们的最近父类。如果动态地扩展对象数组，数组的类型也会动态地发生变化，每次都是取共同的最近父类：

Script

```
array = [C(),D()];   class(array)
array = [array,E()]; class(array)
```

Command Line

```
A
Base
```

11.7.3 使用 Heterogeneous 要避免哪些情况

如图11.9所示，A 和 B 的对象不能放到同一个数组中去，因为它们没有除 Heterogenous 以外的共同的父类。

如果多重继承的情况中存在交叉继承，使得判断共同父类会出现模棱两可的情况，也是要避免的。比如图11.10所示的情况，C 和 D 的对象不能放到同一个对象数组中去。

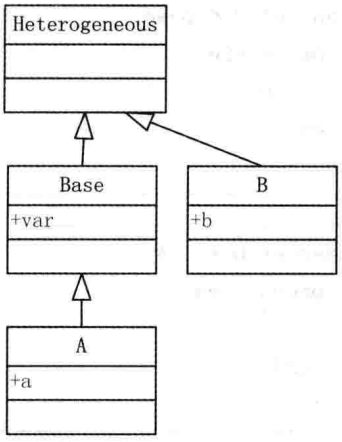


图 11.9 A 和 B 类的最近共同父类没有继承自 Heterogeneous 类

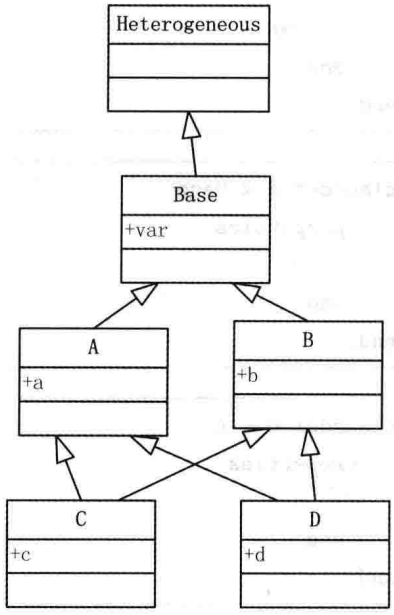


图 11.10 C 的 D 的共同父类不止一个

11.7.4 如何向量化遍历数组中对象的属性

举一个简单的例子，假设对象数组由一系列的 Circle 对象组成，每个 Circle 对象的半径 r 是不同的，如图11.11所示。

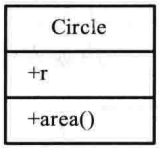


图 11.11 一个简单的 Circle 类：含有一个属性一个方法

构造出来的数组，如果用 ● 语法来访问数组中对象的共同属性，返回的结果将是一个用逗号分隔的 list。

Script

```
objArray =[Circle(1),Circle(2),...
           Circle(3),Circle(4)];
objArray.r
```

Command Line

```
ans =
     1
ans =
     2
ans =
     3
ans =
     4
```

可以使用方括弧 [ ] 来收集返回的结果，放到一个数组中去：

Script

```
r = [objArray.r]

size(r)
```

Command Line

```
r =
     1     2     3     4
ans =
     1     4
```

如果对象数组是一个 Heterogeneous 数组，MATLAB 仍支持使用● 语法来访问对象都共有的属性。比如图11.12所示的这种结构，如果 Heterogeneous 数组中存放的是 C 和 D 类的对象，则 MATLAB 将支持使用向量化地访问 C 和 D 对象的共同属性 a 和 b。

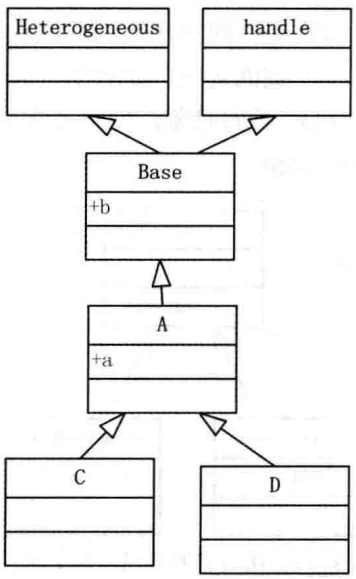


图 11.12 可以访问非同类对象数组中的对象的 a 和 b 属性

11.7.5 如何设计成员方法使其支持向量化遍历

沿用上一节的简单 Circle 例子，我们要求把成员方法设计成支持向量化地访问，以便可以不使用循环就能遍历数组中的对象，我们还期望能向量化地调用 area 方法，从而一次性地得到所有的 Circle 对象的面积。满足这些要求的 Circle 类的 area 成员方法设计如下：

Circle.m

```
classdef Circle < handle
    properties
        r
    end
    methods
        function obj = Circle(val)
            obj.r = val;
        end
        function s = area(obj)
           s = pi*[obj.r].^2;       % 设计 area 函数时 就考虑向量输入
        end
    end
end
```

这里要求，在设计类方法的一开始，就要考虑到输入参数 obj 有可能是一个矢量的情况。如果对整个对象数组使用这个成员方法，得到的结果将是一个 1 × 4 的双精度 array：

Script

```
objArray =[Circle(1),Circle(2),...
           Circle(3),Circle(4)];
s = objArray.area
```

Command Line

```
s =
    3.1416   12.5664   28.2743   50.2655
```

当在脚本中调用 `objArray.area` 时，整个对象数组 objArray 是被当做一个参数传入 area 方法中的。这也就是说，如果在 area 方法中检查 `size(obj)`，返回的结果将是1 × 4。

如果是 Heterogenous Array，想要支持向量化访问的成员方法，则必须将其声明成 Sealed。也就是说，该方法禁止子类中有不一致的定义。如图11.13所示，Heterogeneous array 中放置的是 ColoredCircle 和 ConcentricCircle 类的对象，如果要向量化地调用对象数组的 area 方法，则基类中的 area 方法必须声明成 Sealed。

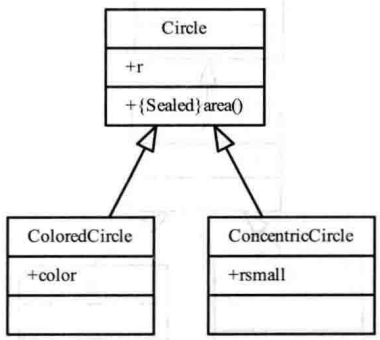


图 11.13 支持向量化访问的成员方法必须声明成 Sealed

问题：对象数组是否可以向量化调用 destructor

对象数组的各对象的 delete 方法是一个例外。首先，在继承结构中，delete 函数禁止定义成 Sealed，这很好理解。如果不这样，子类将无法定义自己的析构函数，因此导致子类的资源无法被释放；其次，尽管 delete 方法不是 Sealed，但是 MATLAB 对 delete 方法破例，仍旧支持使用`objArray.delete` 的格式来调用对象们的析构函数；最后，虽然可以使用 `objArray.delete`，但是 MATLAB 其实并没有向量化地调用对象们的析构函数，因为根本不存在这样一个共同的 delete 方法，对象数组中的对象具有不同的析构函数，所以 MATLAB 对`objArray.delete` 的解释将是：逐个调用每个对象的析构函数。

第 12 章　类的运算符重载

12.1　理解 MATLAB 的 subsref 和 subsasgn 函数

12.1.1　MATLAB 如何处理形如 a(1,:) 的表达式

MATLAB 的主要数据结构是矩阵、元胞数组和结构体。对于这些主要的数据结构，MATLAB 提供灵活的访问方式。比如，一个 3×3 的矩阵 A，可以使用圆括号 $A(a,b)$ 的形式来访问其中的某一个元素，还可以使用 $A(a,:), A(a:b,:)$，即 Slicing 的方式，来访问该矩阵的某一或某些行。例如：

Command Line

```
>> A = rand(3,3)
A =
    0.3922    0.7060    0.0462
    0.6555    0.0318    0.0971
    0.1712    0.2769    0.8235
>> A(1:2,:)
ans =
    0.3922    0.7060    0.0462
    0.6555    0.0318    0.0971
```

形如 `A(1:2,:)` 这样的表达式，会被 MATLAB 解释器（Interpreter）转换成一个 subsref 的函数调用，如图12.1所示。该 subsref 函数的第一个参数是要访问的数据 A；第二个参数是要访问元素所在的位置，并且该所在位置信息存放在一个结构体中。例如：

```
subsref(A,s);
```

其中 s 是一个结构体，用来存放`A(1:2,:)` 表达式中括号的类型和内容。这里，括号类型为`'()'`；内容分为两部分，一部分是`1:2`，另一部分是冒号。注意：该冒号作为一个字符，需要加上单引号。

```
s.type = '()';
s.subs={1:2,':'};
```

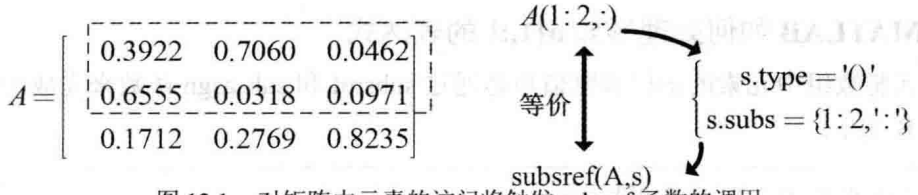


图 12.1　对矩阵中元素的访问将触发 subsref 函数的调用

用户也可以直接设置结构体 s 中的内容，然后调用 subsref 函数，对 A 中元素进行访问。比如要访问 A 矩阵中的 `A(1,1)` 元素：先给 struct s 的两个 field 赋值，然后把 s 当做参数，

再提供给 subsref 内置函数。例如：

Command Line

```
>> s.type = '()';
>> s.subs  = {1,':'};
>> subsref(A,s)
ans =
    0.3922    0.7060    0.0462
```

相似，对数组中元素的赋值也会被 Interpreter 转换成一个 subsasgn 的函数调用。例如：

```
A(1,1) = 0;
```

在内部将会被转换成

```
subsasgn(A,s,0);
```

其中，s 是一个结构体，其内容是：

```
s.type = '()';
s.subs={1,1};
```

可以显示地给 s 结构体赋值来达到和 A(1,1)=0 同样的效果。例如：

Command Line

```
>> s.type = '()';
>> s.subs  = {1,1};
>> subsasgn(A,s,0)      % 该函数返回被修改的值
ans =
    0
```

如图12.2所示，对矩阵元素的赋值将触发 subsasgn 的函数调用。

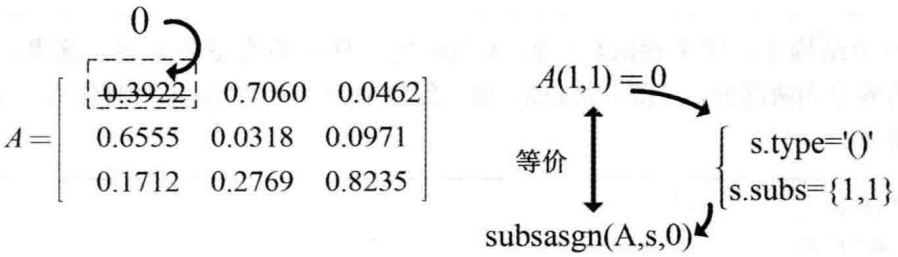


图 12.2 对矩阵中元素的赋值将触发 subsasgn 的函数调用

12.1.2 MATLAB 如何处理形如 a{1,:} 的表达式

对于元胞数组中元素的访问和赋值也是通过 subsref 和 subsasgn 函数来完成的。假设有元胞数组 B 如下：

```
B(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
B(1,2) = {'Anne Smith'};
B(2,1) = {3+7i};
```

```
B(2,2) = {-pi:pi/4:pi};
```

对元胞数组 B 第一列的访问：

```
B{:,1}
```

将会被 Interpreter 转换成如下的 subsref 函数调用，如图12.3所示。

```
s.type = '{}';
s.subs = {':',1};
subsref(B,s) ;
```

元胞数组和普通数组的区别在于，s.type 中的内容是花括号 { } 还是圆括号( )。

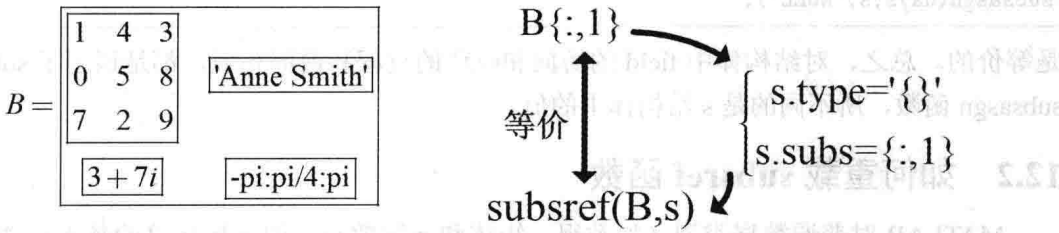


图 12.3 对元胞数组中元素的访问将触发 subsref 函数的调用

对 B 的第一个元胞的赋值：

```
B{1,1} = 0;
```

将会被 MATLAB 解释器转换成如下 subsasgn 的函数调用：

```
s.type = '{}';
s.subs = {1,1};
subsasgn(B,s,0);
```

12.1.3 MATLAB 如何处理形如 s.f 的表达式

对于 struct 的访问和赋值也可以通过直接调用内置函数来完成，只不过 s 的 type 栏位要换成“.”。如果有如下结构体：

```
days.f1 = 'Sunday' ;
days.f2 = 'Monday' ;
days.f3 = 'Tuesday';
```

对第一个栏位 f1 的访问

```
days.f1
```

和表达式

```
s.type = '.';
s.subs={'f1'};
subsref(days,s)
```

是等价的；而赋值语句

```
days.f3 ='NULL';
```

和表达式

```
s.type = '.';
s.subs={'f3'};
subsasgn(days,s,'NULL');
```

是等价的。总之，对结构体中 field 的访问和赋值的过程和前面相似，都是调用了 subsref 和 subsasgn 函数，所不同的是 s 结构体中的值。

12.2 如何重载 subsref 函数

MATLAB 对普通数据类型（如数组、矩阵和元胞数组）的下标运算的基本原理是把下面左边的表达转换成下面右边的等价的函数调用。

```
A(1,2)
```

```
subsref(A,s)
```

MATLAB 对象也同样支持类似的下标运算，即形如下面左边的访问，将触发对该对象所属类的 subsref 方法的调用。

```
obj(1,1)
```

```
subsref(obj,s)
```

具体说来，形如 subsref(obj,s) 的调用，将首先触发 MATLAB 检查该 obj 所属的类是否定义了名叫 subsref 的方法。如果用户定义了 subsref 方法，则优先调用该类的方法。通过这种方式，用户就可以给自己的类设计下标运算了。假设下面的一个类叫做 DataCollection，类中的成员数据分别是一个数组和一个元胞数组，则可以通过下面的方式重载 ( ) 和 { } 算符，使得对 obj 使用下标 (i,j) 时，可以直接访问到对象中的 matrix 变量；对 obj 使用下标 { } 时，可以直接访问到对象中的 cell 变量。例如：

DataCollection

```
classdef DataCollection < handle
    properties
        matrix
        cell
    end
    methods
        function sref = subsref(obj,s) % 该 subsref 相当于一个中转层
           switch s.type
```

```
            case '()'
                sref = builtin('subsref',obj.matrix,s);
            case '{}'
                sref = builtin('subsref',obj.cell,s);
            case '.'
                sref = builtin('subsref',obj,s);
        end
    end
  end
end
```

要注意两点，最终对属性 matrix 和 cell 的成员属性的访问，还是需要通过 MATLAB 内部的 subsref 函数来完成，用户通过重新定义一个 subsref 函数只是起到了一个中间层的作用。

另外，细心的读者可能已经注意到，switch 中有处理点的分支，而例子中只要求支持花括号和圆括号式访问。这是为什么呢？原因是一但重载了 subsref 函数，所有类似下面的表达

```
obj(1,1)
obj{1,1}
obj.matrix
obj.cell
```

MATLAB 都会调用 DataCollection 自己定义的 subsref 函数，由于重载的 subsref 必须还能够处理后两种情况，即处理 obj.matrix, obj.cell 的情况，所以 switch 中还要有点● 的情况，那个分支中，只是把请求原封不动地向内置的 subsref 函数传递。

12.3 如何重载 subsasgn 函数

同理，也可以重新定义 subsasgn 函数，以改变对象下标赋值的行为。例如：

DataCollection

```
classdef DataCollection < handle
    properties
        matrix
        cell
    end
    methods
         function  obj = subsasgn(obj,s,val)
           switch s.type
               case '()'
                  obj.matrix = builtin('subsasgn',obj.matrix,s,val);
                  % 注意这里返回的是 obj.matrix
               case '{}'
                  obj.cell = builtin('subsasgn',obj.cell,s,val);
                  % 注意这里返回的是 obj.cell
               case '.'
```

```
                builtin('subsasgn',obj,s,val);
            end
        end
    end
end
```

须要注意两点：首先，和重载 subsref 一样，当使用 (),{} 下标运算赋值时，用户定义的 subsref 将最先被访问，所以定义的 subsref 要保证三种情况都要考虑到，即要包括处理点运算符的分支；其次，上述的 Switch 中的前两个分支，返回的分别是 obj.matrix 和 obj.cell，这是因为该 subsasgn 的任务是修改对象的属性，而 obj.matrix 和 obj.cell 这两个属性不是 Handle 对象，所以在 subsasgn 函数内部的修改都是局部的修改，必须把它传回来。这相当于对 obj.matrix 和 obj.cell 做重新赋值。

重载 subsasgn 和 subsref 要小心，因为 subsasgn 和 subsref 是公共函数，且该公共函数提供了对内部属性（包括私有属性的接口）。读者可以试验一下，在上述的例子中，即使把 matrix 和 cell 定义成私有数据 (Access = private)，在类的外部，还是可以通过，obj(1,2) 和 obj(1,2) 对私有属性进行访问和赋值。换句话说，重载内置的 subsasgn 和 subsref 函数，将“重载”该类的属性的访问权限，使用时要慎重。

12.4 什么情况下重载下标运算符

在第12.8节中将给出例子，这里先罗列常见的重载下标运算符的情况：

- □ 当用户想完全禁止通过 Dot 来访问对象内部属性时，可以重载 subsref 函数[①]。
- □ 当用户想构造一个对象，使其行为看上去像函数一样，可以重载 subsref 函数。
- □ 当用户想在赋值对象数组时做更多的检查和限制时，可以重载 subsasgn 函数。
- □ 让一个标量对象的行为变得像矢量对象一样[②]。

需要注意，应该将使用自定义的下标运算符取代内置定义的运算符当做一个方便的编程手段，但不应该使用在对性能要求很高的程序当中。

12.5 如何重载 plus 函数

重载一个算符，就是重新设计这个算符的行为。比如在 MATLAB 中，尝试通过使用+号，把字符串串接起来，但是得到的结果，却是这些字符串的 ASCII 码值的和：

```
s1 = 'hello';
s2 = 'world';
s3 = s1 + s2;
```

Command Line

```
>> s3
s3 =
   223   212   222   216   211
```

①见12.8节例子。
②见12.2节，12.3节例子。

这是因为在 MATLAB 中，+ 的默认行为是做算术运算，所以在其内部，把 string 转换成数字，调用 plus 函数。其实，也可以像下面这样调用，得到的结果是一样的。

Command Line

```
>> plus('hello','world')
ans =
   223   212   222   216   211
```

还要注意，该 plus 函数要求输入的参数有相同的维数，否则出错：

Command Line

```
>> plus('goodbye','world')
Error using  +
Matrix dimensions must agree.
```

传统的和并两个字符串的操作是 strcat 函数：

Command Line

```
>> strcat(s1,s2)
ans =
helloworld
```

如果我们想要使用 + 号来达到 strcat 的效果，还有一个方法，就是重载这个 plus 运算符的行为。于是，可以这样设计一个 Value 类的 StringClass，其中的属性是要串接的字符串：

StringClass

```
classdef StringClass
        properties
            s
        end
        methods
            function obj = StringClass(sinput)
                obj.s = sinput;
            end

            function  newStrObj = plus(str1,str2)
                stemp = strcat(str1.s,str2.s);
                newStrObj = StringClass(stemp);      % 注意这个函数返回一个新的对象
            end
        end
  end
```

初始化两个对象 s1 和 s2：

```
s1 = StringClass('hello');
s2 = StringClass('world');
s3 = s1 + s2;
```

MATLAB 对其中第三行的解释会转换成函数调用 `plus(obj1,obj2)`。由于第一个参数是对象，MATLAB 会首先检查该对象的类中是否定义了 plus 函数。如果定义了，就优先调

用这个类方法，在该方法中，把两个对象中的 s 变量串接起来，返回一个新的字串，所以检查 s3 中的 s 属性正是我们期望的结果。

Command Line

```
>> s3.s
ans =
helloworld
```

12.6 MATLAB 的 Dispatching 规则是什么

接着第12.5节的例子，我们再仔细观察下面两段程序，最后一行都是 s1+s2，但是得到的结果却不一样。

```
1 s1 = 'hello';
2 s2 = 'world';
3 s3 = s1 + s2;
```

```
1 s1 = StringClass('hello');
2 s2 = StringClass('world');
3 s3 = s1 + s2;
```

这是为什么呢？这是因为 MATLAB 对两个 s1+s2 采取了不同的遣派（Dispatch）规则。在 MATLAB 的内部，s1+s2 首先被转换成了 plus(s1,s2) 形式的调用。然后 MATLAB 会检查参数的类型，以及该类型是否支持 plus 操作。左边代码中，参数的类型是内置的 string 类，该内置类中不支持加法操作，所以按照内部的隐式转换规则，把 string 转换成 Double 类型，做加法运算。右边的代码两个参数的类型都是我们自己定义的 StringClass，且该类中有 plus 方法的定义，于是调用 StringClass 中的 plus 方法。

第2.4.2小节中简单提到p1.normalize() 的命令和normalize(p1) 大致上是等价的。都是由 MATLAB 的 Dispatcher 做方法的分派，决定调用哪个 normalize 的方法。下面举一个更一般的调用类方法的例子：

```
obj.foo(arg1,arg2,arg3)
```

或者解释器解释过后的形式 foo(obj,arg1,arg2,arg3) 来讨论 Dispatcher 的规则：

- 对于型如obj.foo(arg1,arg2,arg3) 的调用，Dispatcher 会直接检查 obj 对象的类中是否定义了 foo 的方法。如果定义了，则调用之；如果没有定义，则报错。
- 对于型如foo(obj,arg1,arg2,arg3)Dispatcher 不会首先检查 obj 的类中是否定义了 foo 方法，而是先检查四个参数 obj，arg1，arg2，arg3，哪个参数所属的类是更高级别的。找到的该更高级别的类，叫做 dominant 类，然后查找该 dominant 类中是否定义了 foo 方法。如果定义了，则调用，如果没有定义，则报错。

类的级别是如何确定的呢。首先，MATLAB 规定，任何用户定义的类，其级别都高于 MATLAB 的内置类（double，single，char，logical，int64，uint64，int32，uint32，int16，uint16，int8，uint8，cell，struct，function，handle）。所以，在上面的 obj，arg1，arg2，arg3 例子中，如果 obj 是用户的自定义类，而 arg1，arg2，arg3 是 MATLAB 中的普通数据类型（当然不会恰好有这个 foo 方法的定义），那么即使对函数的调用写成

```
foo(arg1,arg2,obj,arg3)
```

Dispatcher 最终还是会把函数的调用分派到 obj 的类中的 foo 方法中去的。如果使用 dot 的调用形式`obj.foo(arg1,arg2,arg3)`，一律直接在 obj 的类定义中查找 foo 方法，Dispatching 规则和类的优先级无关。

MATLAB 还提供如下方法指定用户定义的类之间的级别：

```
classdef (InferiorClasses = {?A,?B}) C
    ......
end
```

通过这种方式定义出来的 C 的对象的级别比 A 和 B 的级别要高，如果这三个类出现在参数列表中，则 C 的对象是 dominant 对象。

12.7 如何判断两个对象是否相同

对于 Handle 类对象，== 运算是在 Base 类 Handle 中就定义好的算法（函数的名称叫做 eq），该运算符将检查 Handle 类对象所指向的实际数据是否是同一个。例如：

SomeHandleClass

```
classdef SomeHandleClass < handle
    properties
        s
    end
    methods
        function obj = SomeHandleClass()
            obj.s = rand(10,10);
        end
    end
end
```

Handle 类对象的拷贝是 shallow 拷贝，只复制了类中的地址，没有复制对象中实际所指向的数据的对象：

```
o2 = o1 ;
o3 = o2 ;
```

使用该== 算符，将得知两对象是相同的，因为它们指向的是相同的内部数据。

Command Line

```
>> o1 == o2
ans =
     1
>> o1== o3
ans =
     1
```

如果是 Value 类对象，并没有直接定义“==”运算符，需要用户自己指定行为。比如：

SomeValueClass

```
classdef SomeValueClass
    properties
       s = 'Hello';
    end
end
```

如果声明两个对象 v1 和 v2：

```
v1 = SomeValueClass();
v2 = v1;
```

如果尝试直接比较两个对象是否相等，MATLAB 会报错，指出 eq 函数没有定义。

Command Line

```
>> v1 == v2
Undefined function 'eq' for input arguments of type 'SomeValueClass'.
```

可以把 Value 对象的相同的判断标准，规定成两个对象中的属性 s 相同，那么该 Value 类的 eq 方法可以这样设计：

SomeValueClass

```
classdef SomeValueClass
    properties
       s = 'Hello';
    end
    methods
        function result = eq(input1,input2)
            result = strcmp(input1.s,input2.s) ;
        end
    end
end
```

这样就可以使用== 来比较两个对象了：

Command Line

```
>> v1 = SomeValueClass();
>> v2 = v1;
>> v2 == v1
ans =
1
```

12.8 如何让一个对象在行为上像一个函数

如果程序开发者把自己的 MATLAB 程序提供给一个没有面向对象经验的用户去使用，或者希望扩展一个普通的函数，使其具有类的功能（如 Handle 基类的功能），这种情况下，可以把自己的类的使用设计得和普通函数的调用一样。这种类的对象也叫做仿函数，它们实际上是对象，但是行为上模仿函数。

为了实现这种仿函类，首先我们来比较一下，函数调用和类方法调用的异同：

Command Line
```
>> myfunc(arg1,arg2);          % 普通函数调用
>> obj.mymethod(arg1,arg2);    % 类方法调用
```

调用类方法，需要提供该类的对象和方法名，这和myfunc(arg1,arg2)调用看上去没有什么共同的地方。但是，我们前面提到过，MATLAB 的类可以重载下标运算符 subsref，让类的对象能够处理类似如下的调用：

Command Line
```
>> obj(arg1,arg2)
```

这样看上去，就和函数调用相似多了。其实，obj(arg1,arg2) 和 myfunc(arg1,arg2) 在形式上，没有本质的区别，只是 obj 是对象名，myfunc 是函数名，如果把对象的名字声明成看上去像一个函数的名字，比如 myfunctor，对象的使用者是觉察不出区别来的。这样，一个对象就可以伪装成函数了。另外，为了让对象，完全看上去像一个函数，还要禁止使用 Dot 访问和 { } 形式的访问：

Command Line
```
>> myfunctor.propName      % 需要禁止，因为函数不支持这种访问
>> myfunctor{1,2}          % 需要禁止，因为函数不支持这种访问
```

在下面的一个例子中，我们设计一个用来遍历集合中元素的类 FunctorForEach，目的是要让它的对象表现得像一个函数（好比有些编程语言中常见的 for each 函数）一样。该“函数”接受两个 Iterator 作为开始和结束[①]，还接受一个函数句柄表示遍历时，对每个元素进行的操作。为简单起见，这里该操作仅是打印被遍历文件的名称。在脚本中，可以这样使用这个仿函数：

Script
```
agg = FileAggregator(pwd);
for_each = FunctorForEach();
for_each(agg.first(),agg.last(),@print);    % 完全是普通函数的调用方式
```

其中，print 函数很简单：

print
```
function print(input)
    disp(input);
end
```

下面的代码是 FunctorForEach 的类设计，该类重载了下标运算符 subsref，并且禁止了除 ( ) 以外的其他访问方式和赋值方式，其目的是要让该类的对象的行为表现得和普通函数一样。

FunctorForEach
```
classdef FunctorForEach
    methods
        function sref = subsref(obj,s)
          switch s.type
              case '()'
```

①Iterator 和 Aggregator 是设计模式的一种，详见第17.3节。

```
                iterBegin = s.subs{1};
                iterEnd = s.subs{2};
                funcH = s.subs{3};
                % le 运算符在 FileIterator 中重载
                while(iterBegin <= iterEnd)
                   % 遍历时对每个元素的操作
                   funcH(iterBegin.currentItem);
                   iterBegin.nextItem();
                end
            case '{}'
                error('not support');
            case '.'
                error('not support');
          end
        end
        function obj = subsasgn(obj,s,val)
          switch s.type
            case '()'
               error('not support');
            case '{}'
               error('not support');
            case '.'
               error('not support');
          end
        end
    end
end
```

FileAggregator 对象代表文件夹，其 first 和 last 方法分别用来提供一个指向文件夹首文件和末文件的 Iterator 对象：

FileAggregator

```
classdef FileAggregator < handle
    properties
        files
    end
    methods
        function obj = FileAggregator(path)
            tmp = dir(path);
            files = tmp(cell2mat({tmp(:).isdir})~=1);
            obj.files = {files.name};          % 文件名被存放在元胞数组中
        end
        function iter = first(obj)             % 返回一个 Iterator 对象
            iter = FileIterator(obj,1);
        end
```

```
        function iter = last(obj)              % 返回一个 Iterator 对象
             iter = FileIterator(obj,numel(obj.files));
        end
    end
end
```

在 FileIterator 类中，为了使用算术符号来比较两个 Iterator 的大小，还重载了<=(le) 运算符。

FileAggregator

```
classdef FileIterator < handle
    properties
        counter
        aggHandle
    end
    methods
        function obj = FileIterator(aggObj,index)
            obj.aggHandle = aggObj;
            obj.counter = index ;
        end

        function itemObj = currentItem(obj)
            obj.counter
            itemObj = obj.aggHandle.files(obj.counter);
        end
        function nextItem(obj)
            obj.counter = obj.counter +1;
        end
        function result = le(in1,in2)
            result = ((in1.counter  <= in2.counter)&&...
                      (in1.aggHandle == in2.aggHandle) );
        end
    end
end
```

12.9 MATLAB 中哪些算符允许重载

无论是什么样的算符，在 MATLAB 内部，解释器都会把对算符的使用转换成对应的函数调用，并且如果是二目运算，该函数调用的第一个参数是算符左边的对象，第二个参数是算符右边的对象。表12.1中所列是在 MATLAB 中允许重载的运算符。

表 12.1 MATLAB 允许重载的运算符

运算	实际调用的函数	算符的作用
a + b	plus(a,b)	Binary addition
a − b	minus(a,b)	Binary subtraction
−a	uminus(a)	Unary minus
+a	uplus(a)	Unary plus
a. ∗ b	times(a,b)	Element wise multiplication
a ∗ b	mtimes(a,b)	Matrix multiplication
a./b	rdivide(a,b)	Right element wise division
a.\b	ldivide(a,b)	Left element wise division
a/b	mrdivide(a,b)	Matrix right division
ap\b	mldivide(a,b)	Matrix left division
$a.^b$	power(a,b)	Element wise power
a^b	mpower(a,b)	Matrix power
a < b	lt(a,b)	Less than
a > b	gt(a,b)	Greater than
a <= b	le(a,b)	Less than or equal to
a >= b	ge(a,b)	Greater than or equal to
a ∼= b	ne(a,b)	Not equal to
a == b	eq(a,b)	Equality
a&b	and(a,b)	Logical AND
a|b	or(a,b)	Logical OR
a	not(a)	Logical NOT
a : d : b	colon(a,d,b)	Colon operator
a : b	colon(a,b)	Colon operator
a'	ctranspose(a)	Complex conjugate transpose
$a.'$	transpose(a)	Matrix transpose
[ab]	horzcat(a,b,...)	Horizontal concatenation
[a; b]	vertcat(a,b,...)	Vertical concatenation
a(s1, s2, ..., sn)	subsref(a,s)	Subscripted reference
a(s1, ..., sn) = b	subsasgn(a,s,b)	Subscripted assignment
b(a)	subsindex(a)	Subscript index

第 13 章　超　　类

13.1　什么是超类（Meta Class）

在第1.2.2小节引入类的概念时，我们提到对象是真实事件中的具体事物，为了有一个统一的描述它们的方式，我们把相似的事物的共性抽象出来构成类。如图13.1所示，各种车辆的共性可以用 Car 类来形容，公司的各个雇员可以用 Employee 类来形容。

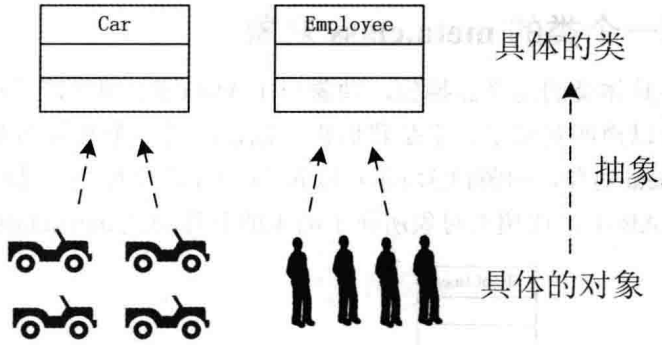


图 13.1　从不同的对象中抽象出共性构成类

下面我们把这个思路放宽一些，如图13.2所示，思考一下，当有很多种 Class 类的定义时，这些类的定义是否可以也被进一步地抽象地概括出来呢？

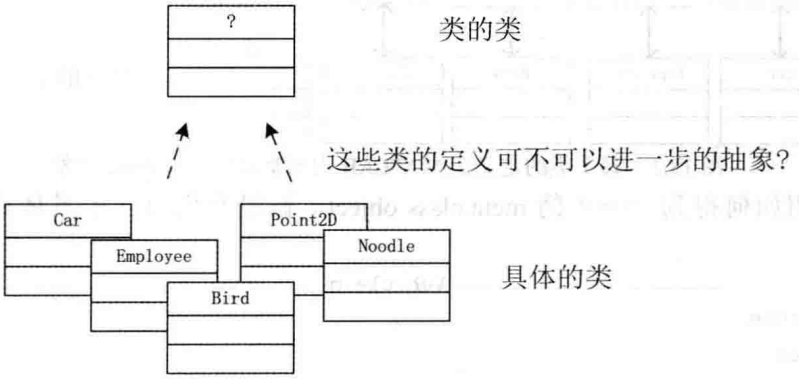


图 13.2　不同的类是否可以继续抽象出共性来？

所谓对类的“进一步抽象”就是通过观察 MATLAB 中各种类的定义，总结出这些类定义的一些共同特征。通过观察不难发现，这些类的定义的共同特征是各个类都是由一系列的属性、方法和事件组成的，并且每个属性和方法可以使用一些关键词来形容，于是可以总结出一种普遍的方式描述各种类。换句话说，可以用类的方式来描述各种类。这样的类叫做超类（Meta Class）。如果用 MATLAB 的语言把超类的定义写出来，看上去大致是这样的：

```
classdef MetaClass < handle
    properties
        Name            %  用户定义的类的名字
```

```
        PropertyList   %  用户定义的类中的成员属性名称的列表
        MethodList     %  用户定义的类中的成员方法名称的列表
        EventList
        ......
    end
    methods
        ......
    end
end
```

13.2 如何获得一个类的 meta.class 对象

上一节我们对各具体类的定义做抽象，抽象出了 Meta 类来形容这些具体类的定义，现在有了类定义，就可以声明对象了。于是我们也可以说，每一个具体的 MATLAB 类的定义都对应一个具体的 meta 对象，如图13.3所示。该 meta 对象用来描述具体的 MATLAB 类中的内容，并且在 MATLAB 中，该超类对象所属于的类的名称叫做 meta.class。

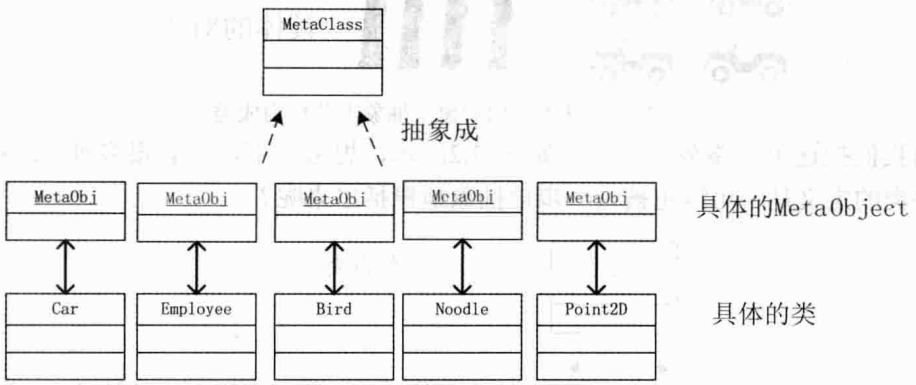


图 13.3 每个类的定义在 MATLAB 内部都对应一个 meta 对象

下面介绍如何得到一个类的 meta.class object，假设我们有一个具体的 Value 类叫做 Vehicle：

Vehicle.m

```
classdef Vehicle
    properties
      make
      year
      model
    end
end
```

第一种得到 meta 对象的方法是：如果已知类的名字，可以在类的前面加上一个问号来获得meta.class 对象，比如

Command Line

```
>> metaObj = ?Vehicle
```

得到的 Vehicle 类的 metaObj 如下：

Command Line

```
metaObj =
  meta.class handle
  Package: meta
  Properties:
                    Name: 'Vehicle'
             Description: ''
      DetailedDescription: ''
                  Hidden: 0
                  Sealed: 0
          ConstructOnLoad: 0
         HandleCompatible: 0
          InferiorClasses: {0x1 cell}
        ContainingPackage: []
             PropertyList: [3x1 meta.property]
               MethodList: [2x1 meta.method]
                EventList: [0x1 meta.event]
    EnumerationMemberList: [0x1 meta.EnumeratedValue]
           SuperclassList: [0x1 meta.class]
  Methods, Events, Superclasses
```

该对象中记录了 Vehicle 类定义中的具体信息。注意，虽然我们没有给 Vehicle 定义任何方法，但是该 metaObj 的 MethodList 中已经有两个方法了，这是 MATLAB 自动提供给该 Vehicle 类的。读者能猜到这是哪两个方法吗？

第二种方法是：如果已经有了一个类的对象，可以用meta.class 函数来获得meta.class 对象，比如：

```
>> metaObj = metaclass(obj) ;
```

第三种方法最灵活，如果类的名字是以字符串的形式存在的，那么可以利用 meta.class 类中的成员方法 fromName，该函数接受 string input，返回meta.class 对象，比如：

Command Line

```
>> name = 'Vehicle' ;
>> metaObj = meta.class.fromName(name) ;
```

13.3 meta.class 对象中有些什么内容

这一节中，我们以如下 Base 和 Derived 类为例[①]，分析meta.class 对象中的内容。

Base.m

```
classdef Base < handle
    properties
        aprop
    end
end
```

Derived.m

```
classdef Derived<Base
    properties
        b
    end
end
```

在命令行使用问号得到meta.class 对象：

①本节中的 MATLAB 输出来自使用 MATLAB2011b。

Command Line

```
>> obj = ?Derived
obj =
  meta.class handle
  Package: meta
  Properties:
                    Name: 'Derived'
             Description: ''
      DetailedDescription: ''
                  Hidden: 0              % 该类的 Hidden 属性为 false
                  Sealed: 0              % 该类的 Sealed 属性为 false
          ConstructOnLoad: 0
         HandleCompatible: 0
           InferiorClasses: {0x1 cell}
        ContainingPackage: []
             PropertyList: [2x1 meta.property]
              MethodList: [2x1 meta.method]
               EventList: [0x1 meta.event]
    EnumerationMemberList: [0x1 meta.EnumeratedValue]
          SuperclassList: [1x1 meta.class]
  Methods, Events, Superclasses
```

该 meta 对象中的属性、方法列表都是对象数组，如果在稍旧一点的 MATLAB 版本中meta.class 的对象中的属性和方法列表是用元胞数组组织起来的，所以访问其中 b 元素时，要根据数组的类型选择不同的方式。观察这个 meta 对象的属性之一 PropertyList：

```
......
PropertyList: [2x1 meta.property]
......
```

其中，meta 是 package 的名称，property 是这个 package 中关于属性的类，该 property 类的对象用来形容用户定义的类中属性的一些性质。这个 meta.package 中一共有如下几个类：

- □ meta.package
- □ meta.class
- □ meta.property
- □ meta.DynamicProperty
- □ meta.EnumeratedValue
- □ meta.method
- □ meta.event

并且 meta.class 类和其他的 meta 类之间的关系是组合关系。

问题： 如何系统地获得类中所有 property 的名字

这里举一个例子，说明如何提取 meta 对象中的信息。沿用上节 Derived 和 Base 类的定义，假如想通过程序获得 Derive 类定义中所有属性的名称，并且在程序中用 string 的形式记

录下来，可以这样做：

Command Line

```
>> metaobj =?Derived;
>> propNameList = {metaobj.PropertyList.Name}
propNameList =
    'b'    'aProp'
```

注意：

- metaobj.PropertyList 是一个对象数组，其中的内容是 meta.property 的对象。
- 使用 ● 语法，向量化地访问数组中对象的共同属性。
- metaobj.PropertyList.Name 返回的结果是 string 类型，由于 string 的长度不同，我们使用花括号{}，把返回的结果收集到元胞数组中去。

13.4 如何手动克隆一个对象

1. 回顾 Handle 类的拷贝

首先回顾一下 Handle 类对象的拷贝规则：

Ref

```
classdef Ref < handle
    properties
        a
    end
    methods
        function obj = Ref()
            obj.a = rand(1);
        end
    end
end
```

前面介绍过，通过如下方式拷贝得到的 obj1 和 obj2 其实指向的是内存中的同一个成员变量 a：

Script

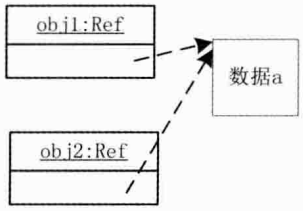
```
obj1 = Ref();
obj2 = obj1;
obj1.a
obj2.a
obj1.a = 10;  % 修改 obj1 的变量
obj2.a
```

Command Line

```
0.1576
0.1576
10        % obj2 的变量也被修改了
```

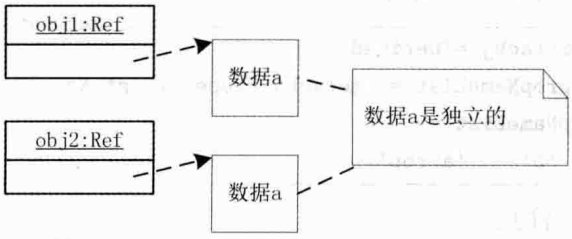
内存中的情况如图 13.4所示。

obj1 和 obj2 的属性 a 是相互关联的不独立，俗称浅拷贝。这节我们演示如何利用 meta.class 中的信息来实现深拷贝。如图13.5所示，具有这种功能的函数叫做克隆函数。



浅拷贝

图 13.4 浅拷贝



深拷贝

图 13.5 深拷贝

2. 简单克隆

先从最简单的克隆方法开始，我们可以构造一个方法，先在该方法中声明出一个新的对象，叫做 newObj，再把旧的对象的每一个属性的值都复制到新的对象 newObj 中，最后再将该 newObj 返回。

下面的例子中：

- Ref 是一个 Handle Class，Ref 类中有一个属性 a，它被初始化成一个随机数。
- Ref 的方法 simpleClone 首先构造一个新的对象 newObj，然后对 newObj 的成员 a 进行重新赋值，以达到克隆的目的。

Ref

```
classdef Ref < handle
    properties
        a
    end
    methods
        function obj = Ref()
            obj.a = rand(1);
        end
        function newobj = simpleClone(obj)  % 简单克隆
           newobj = Ref();
           newobj.a = obj.a;
        end
    end
end
```

下面的脚本测试这个 simpleClone 方法：

Script	Command Line	
`obj1 = Ref();`		
`obj1.a`	`0.3500`	`% a 的初值`
`obj2 = obj1.simpleClone();`		
`obj1.a = rand();  % 改变 a 的值`		
`obj1.a`	`0.2511`	
`obj2.a`	`0.3500`	`% obj2 的 a 值没变`

简单克隆方法的优点是实现简单；缺点是：如果要克隆的对象有很多的属性，那么一个一个地键入属性的名字就太麻烦了。如果修改 Ref 类时，又增加了新的属性，还要记得修改克隆函数，所以这个方法不灵活。下面我们接着修改这个方法。现在的目标是，自动地枚举类中的 property 的名字，其实这正是meta.class 对象中提供的信息。我们可以用meta.class 函数，先获得该类的 metaObject，然后取出其中 PropertyList 中属性的名字，然后遍历赋值即可。注意下面的例子中，我们特地把 property 的属性设置成了 private 来示例这种方法适合各种类型的属性。

Ref

```
classdef Ref < handle
    properties(SetAccess = private , GetAccess = private)
        a
        b
    end
    methods
        function obj = Ref()
            obj.a = rand(1);
            obj.b = rand(1);
        end
        function newobj = clone(obj)
           newobj = Ref();
           metaobj = metaclass(obj);                    % 得到 meta object
           props = {metaobj.PropertyList.Name};         % 得到 props 名字
           for j = 1: length(props)                     % 遍历
                 newobj.(props{j}) = obj.(props{j}) ;
           end
        end
    end
end
```

请读者自己验证 clone 的结果的确是深拷贝，并且两个对象的属性 a 和 b 是独立的变化。

3. 递归克隆

其实，仅利用 meta.PropertyList 中信息的克隆方法仍有局限性。考虑下面这种情况：

```
......
function newobj = clone(obj)
          newobj = Ref();
          metaobj = metaclass(obj);
          props = {metaobj.PropertyList.Name};
          for j = 1: length(props)
                newobj.(props{j}) = obj.(props{j}); % 如果 props 又是一个 Handle 对象呢
          end
       end
```

```
end
......
```

该方法中还有一个 = 符号，所以这里还有一个隐藏的前提，就是所有的 property 都是 Value 类型的对象，如果有一个 property，比如说 b，是 Handle 类型的，如图13.6所示，其中 property b 被初始化成一个 Handle 类的对象：

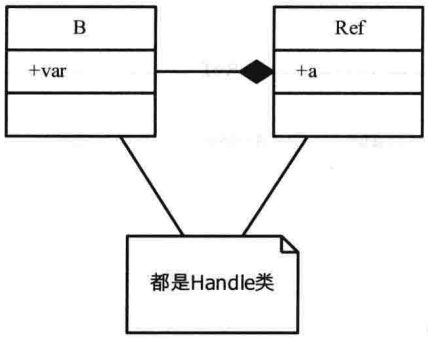


图 13.6 如果 Ref 类中的一个属性是 Handle 类的对象

如果使用 simpleClone 或者 Clone 方法，得到的 b 属性的拷贝仍然是浅拷贝，如图13.7右所示。

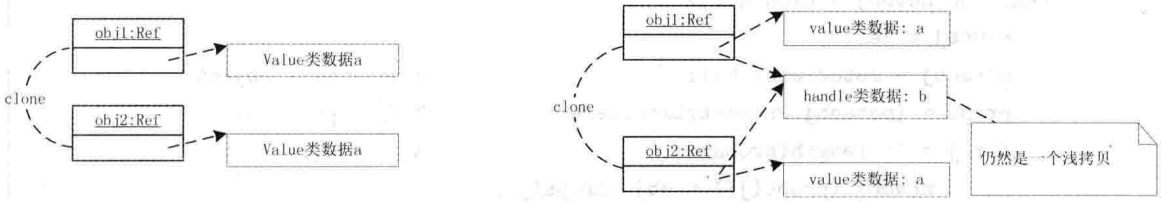


图 13.7 对象中的 b 属性还是浅拷贝

那么，如何对 Ref 类的变量进行深拷贝呢？这里提供一个大致的思路，就是检查每个属性。如果属性是 Handle 类对象，则递归地调用该对象的克隆函数。当然，这要求用户的设计 B 类中，也要定义 clone 方法。也就是说，用户需要提供两个类的 clone 函数，设计如下：

Ref

```
classdef Ref < handle
    properties
        a
        bobj
    end
    methods
        function obj = Ref()
            obj.a = rand(1);
            obj.bobj = BHandle();          % 其中一个属性被初始化成 Handle 类对象
        end

        function newobj = clone(obj)
           newobj = Ref();
           metaobj = metaclass(obj);
```

```
            props = {metaobj.PropertyList.Name};
            for j = 1: length(props)
                tmpProp = obj.(props{j}) ;
                if(isa(tmpProp,'handle'))  % 如果是 Handle 类对象，调用该类的 Clone 方法
                  newobj.(props{j}) = tmpProp.clone();
                else                       % 否则做直接赋值拷贝
                  newobj.(props{j}) = obj.(props{j}) ;
                end
            end
        end
    end
end
```

BHandle

```
classdef BHandle < handle
    properties
        var
    end
    methods
        function obj = BHandle()
            obj.var = rand(1);
        end

        function newobj = clone(obj)
            newobj = BHandle();
            metaobj = metaclass(obj);
            props = {metaobj.PropertyList.Name};
            for j = 1: length(props)
                tmpProp = obj.(props{j}) ;
                if(isa(tmpProp,'handle'))
                   % 当然程序不会运行到这，因为 var 不是 Handle 对象
                   newobj.(props{j}) = tmpProp.clone()
                else
                   newobj.(props{j}) = obj.(props{j}) ;
                end
            end
        end
    end
end
```

可以使用如下脚本验证 clone 完成深拷贝，即属性 bobj 的独立变化。

```
Script                                    Command Line
obj1 = Ref();
obj2 = obj1.clone();
obj1.bobj.var       % bobj.var 的初值          0.4468
obj2.bobj.var                                 0.4468
obj1.bobj.var = 10; % 改变 bobj.var 的值
obj1.bobj.var                                 10
obj2.bobj.var                                 0.4468
```

递归地调用 clone 方法将完成深拷贝，如图13.8所示。

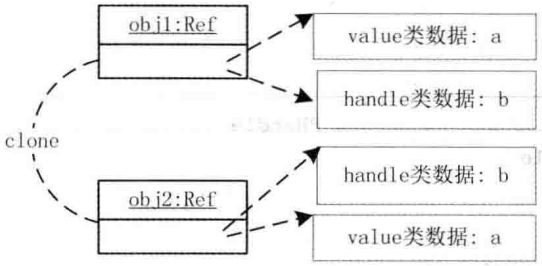


图 13.8 递归地调用 clone 方法将完成深拷贝

该方法仅对深拷贝的实现做一个简单的演示，当然还会存在更复杂的情况。比如，如果存在 A 对象和 B 对象互相包含的（有环）的情况，如何做深拷贝则是一个算法问题了，这里不再赘述。

13.5 如何使用 matlab.mixin.Copyable 自动克隆一个对象

从 R2011a 开始，MATLAB 提供了一个 mixin 类 Copyable，其中包括 copy 和 copyElement 两个方法来帮助用户自动完成基本的 Handle 类对象的深拷贝。其中 copy 方法是 Sealed，不允许子类重载，而 copyElement 是 protected 方法，允许子类重载。用户只需要让自定义的类继承自 matlab.mixin.Copyable 即可，如图13.9所示。

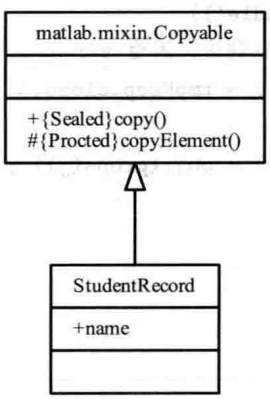


图 13.9 要想获得 copy 和 copyElement 方法，用户需继承自一个 Copyable 基类

这样定义出来的类仍然是一个 Handle 类，由于其继承了 matlab.mixin.Copyable，也就继承了基类中提供的 copy 和 copyElement 方法。比如图13.9中的 StudentRecord 类，对其使用 copy 方法，得到的新对象 record2，其是 record1 的深拷贝。在这个例子中，StudentRecord 类

需要具有深拷贝的原因可以这样理解，每个 StudentRecord 对象都代表一个学生，每个学生都应该有自己独立的 name，我们可以先拷贝一个对象，然后给该对象的属性 name 赋不同的值，从而得到的是两个不同的对象。

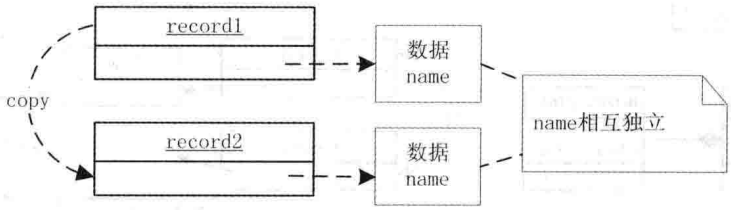


图 13.10 record2 对象是 record1 对象的深拷贝

深拷贝的方法和前面自动地遍历属性的方法相似，Copyable 类中的 copy 和 copyElement 方法组合在一起也是自动遍历的对象中的各个属性做拷贝。

StudentRecord 类很简单：

StudentRecord

```
classdef StudentRecord< matlab.mixin.Copyable
    properties
        name
    end
end
```

下面的脚本可以验证每个 record 对象的 name 属性都是独立的。

Script

```
record1 = StudentRecord();
record1.name = 'A';
record2 = copy(record1);
record2.name = 'B';
record1.name
record2.name
```

Command Line

```
A
B
```

Copyable 类中，默认的实现是对对象中的每个属性做简单的复制。比如 StudentRecord 中如果有一个属性 address 本身也是一个对象，并且这个对象恰好是 Value 类的，使用 copy 方法对 StudentRecord 的对象做拷贝时，该 HomePropInfo 的对象也将被深拷贝，如图13.11右所示。home 属性，或者说 HomeInfo 的对象应该具有自己的独立拷贝的原因很容易理解，每个学生都有各自独立的家和地址，修改一个学生的家庭住址不应该影响另一个对象学生的家庭住址。

Copyable 类中的方法默认的并不包括对属性做递归的深拷贝，如果 StudentRecord 中有一个属性 school 是 Handle 类的对象，使用 copy 方法对 StudentRecord 的对象做拷贝时，该对象将被浅拷贝，如图13.12右所示。这在什么情况下会被用到呢。举个例子，如果一个学校有 5000 个学生，每个学生都是独立的，所以对每个学生都要声明出一个 StudentRecord 对象，但是所有学生的 School 的信息都是一样的（至少我们可以假设是这样），所以没有必要构造出 5000 个完全相同的 schoolInfo 对象去当做 student 对象的属性。所以，如果这 5000 个学生可以共享一个 school 属性，或者说 SchoolInfo 对象，这也是完全合理的。如图13.13所示为

StudentRecord 类和 Homework 类都要继承自 Copyable 基类。

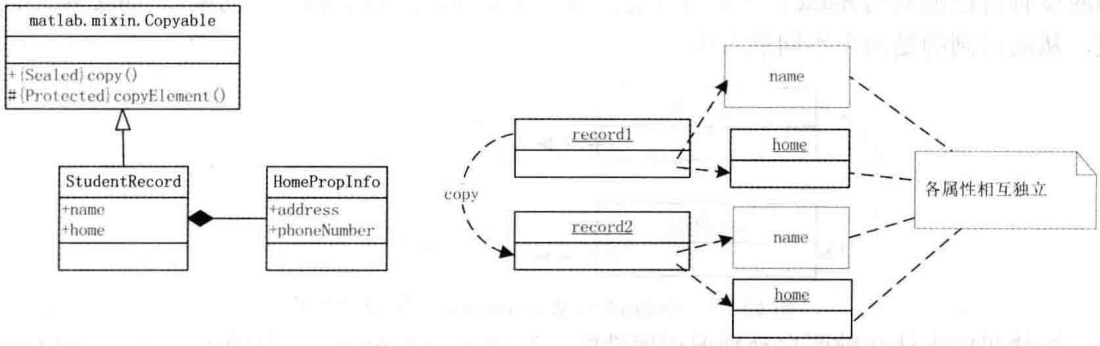


图 13.11 如果 StudentRecord 类中有一个属性是 Value 类对象

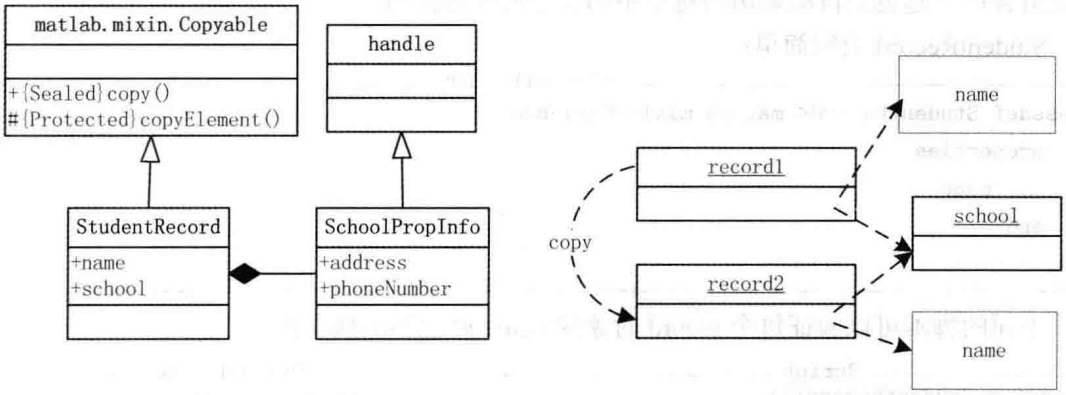


图 13.12 如果 StudentRecord 类中有一个属性是 Handle 类对象

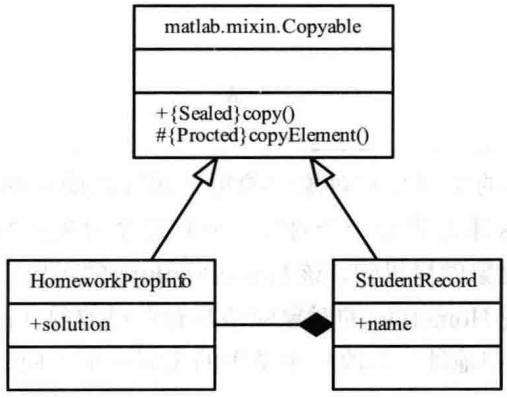
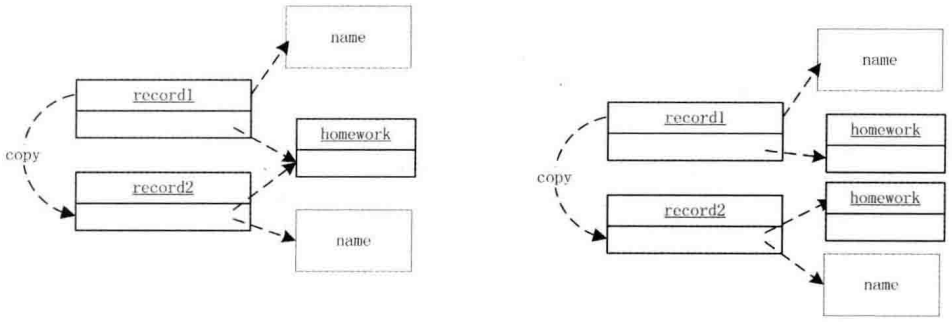


图 13.13 StudentRecord 类和 HomeworkPropInfo 类都要继承自 Copyable 基类

如果用户需要自己定制对象中每个属性的拷贝手段，可以重载 copyElement 方法。比如，再给 StudentRecord 添加一个属性叫做 homework，并且假设该 homework 是 Handle 类对象，copyElement 默认做浅拷贝。现在我们希望深拷贝该 homework 属性，可以先在 copyElement 中调用父类的 copyElement 方法，得到新的 StudentRecord 对象，这时其 homework 属性是被拷贝对象的 homework 属性的浅拷贝，如图13.14左所示。然后再对其中的 homework 属性做完全的复制，如图13.14右。因为 homework 属性本身是一个 Handle 类，对其做完全复制需要让其也继承自 matlab.mixin.Copyable 类。



先调用默认的copyElement得到的结果　　再对homework对象调用copy方法

图 13.14　StudentRecord 类 copyElement 方法中的两步

在子类中重载 copyElement 要注意，该方法也必须声明成 protected，因为声明成 protected 的方法只能被子类所调用。

StudentRecord

```
classdef StudentRecord< matlab.mixin.Copyable
    properties
        name
        homework
    end
    methods(Access = protected)
        function newobj = copyElement(obj)
            newobj = copyElement@matlab.mixin.Copyable(obj);
            newobj.homework = copy(obj.homework) ;
        end
    end
end
```

注意：该 HomeworkPropInfo 也继承自 Copyable 基类。

HomeworkPropInfo

```
classdef HomeworkPropInfo< matlab.mixin.Copyable
    properties
        solution
    end
end
```

下列脚本验证对 homework 属性的拷贝也是一个深拷贝：

Script

```
record1 = StudentRecord();
record1.homework = HomeworkPropInfo();
record1.homework.solution = 'cccc';
record2 = copy(record1);
record2.homework.solution = 'bcbc';
record1.homework.solution
record2.homework.solution
```

Command Line

```
                    % 初值
cccc   % 两个 homework 属性独立变化
bcbc
```

```
end
end
solution
  # private
classes homework_helper_algebra
```

```
end
end
  def solve_equation(equation)
    # Implement equation solving logic here
    # This is a placeholder implementation
    "Solution for #{equation}"
  end
end
```

```
# Example usage
helper = HomeworkHelper.new
puts helper.solve_equation("2x + 5 = 13")
```

第 3 部分

设计模式篇

第 14 章　面向对象程序设计的基本思想

通过前面两章的介绍，我们已经掌握了 MATLAB 面向对象编程的基本语法。从第 3 篇开始，我们将介绍如何使用 MATLAB OOP + 设计模式（Design Pattern）来解决工程科学计算中的实际问题。在讨论使用设计模式之前，我们先作个回顾；到目前为止，我们了解到面向对象设计思想中有两个“武器”，这两个“武器”用来描述类和类之间的关系，一个是继承，一个是组合，如图14.1所示。

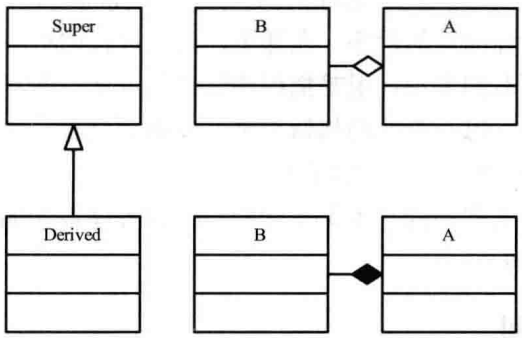


图 14.1　类之间的基本关系：继承和组合

关于继承的关系可以再细化。根据父类和子类的方法之间的关系，可以分为：子类继承父类的方法，子类重新实现父类的方法，子类必须（被强迫）实现父类的方法（父类中定义的是抽象方法），如图14.2所示。

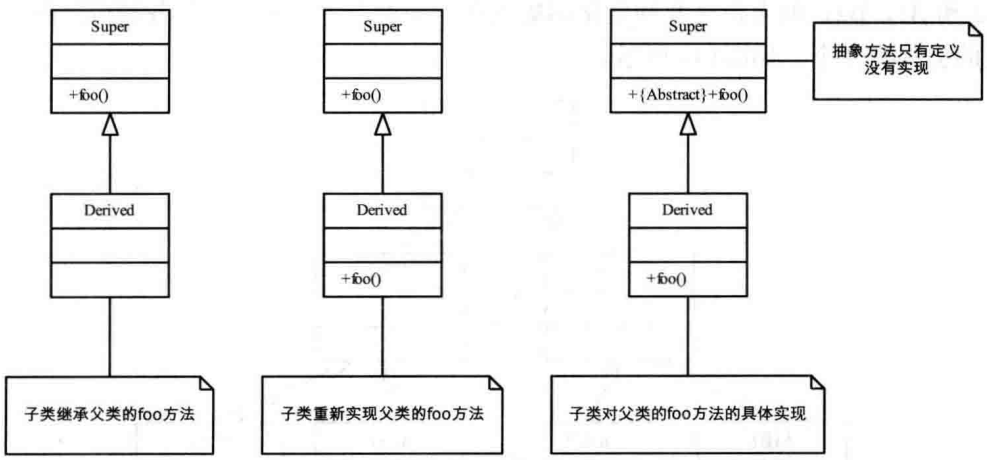


图 14.2　继承结构中父类和子类的方法之间的关系

类的组合关系分两种，◆ 实心菱形箭头表示非包括不可，◇ 空心菱形箭头表示松散的可有可无。这里我们不再讨论两者的区别，只需要记住组合是一种拥有关系。具体到 MATLAB 程序上，就是 A 类对象拥有一个 B 类的对象，B 类既可以是 Value 类，也可以是 Handle 类。这种拥有关系可以在 A 的构造函数中指定，比如把 A 的一个属性初始化成 B 类对象，也可以等到 A 类对象创建之后，通过专门的 set 方法来指定。组合关系在 UML 上还可以表示成如图14.3右边的 UML，两者是等价的。

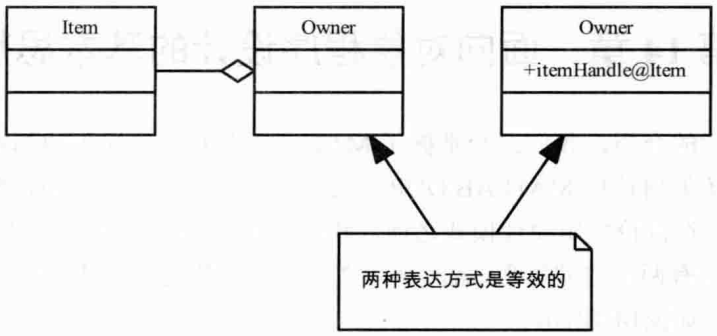


图 14.3 类的组合关系的两种表示方式

再回忆一下第 1 章设计面馆的程序。在那里，面向过程的程序设计的困难是：没办法在程序的一开始就考虑到所有的需求，很难把程序设计得灵活，面对不停增加的需求，程序扩张起来很困难，面对修改，每一个改动都似乎牵一发而动全身。这个困难其实来自于程序设计中一个永远不变的真理：“一直不变的是变化”。

回顾完了面向对象中类和类的基本关系和面向过程程序设计的困难，下面我们开始介绍几个 OOP 设计的基本原则。

14.1 单一职责原则

一个类最好只有一个引起它变化的因素。(Single Responsibility Principle)

单一原则可以用图14.4来解释。假设类中的变化因素是对两种方法的不同实现，分别是 A1，A2 和 B1，B2，如果把这两种变化都集成在一个类当中，并且穷尽各种可能的组合，则要定义的类共有 7 个，如图14.4所示。

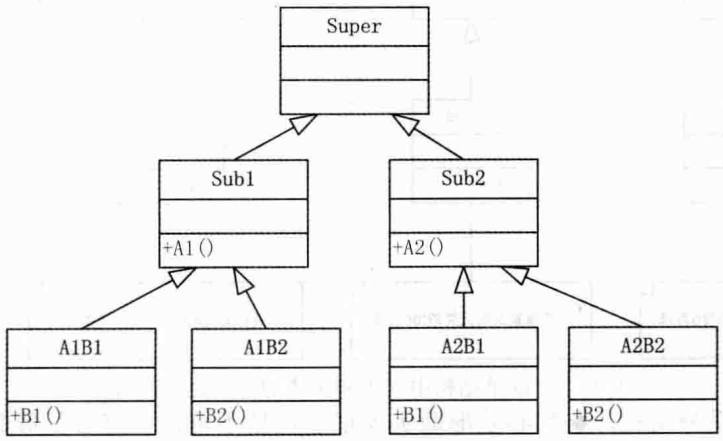


图 14.4 类中有太多的变化因素将导致类的数量增加

单一职责原则建议，最好一个类只承担一个变化。我们可以把一个类中变化的东西取出封装起来，让其他部分不受到影响，如图14.5所示。

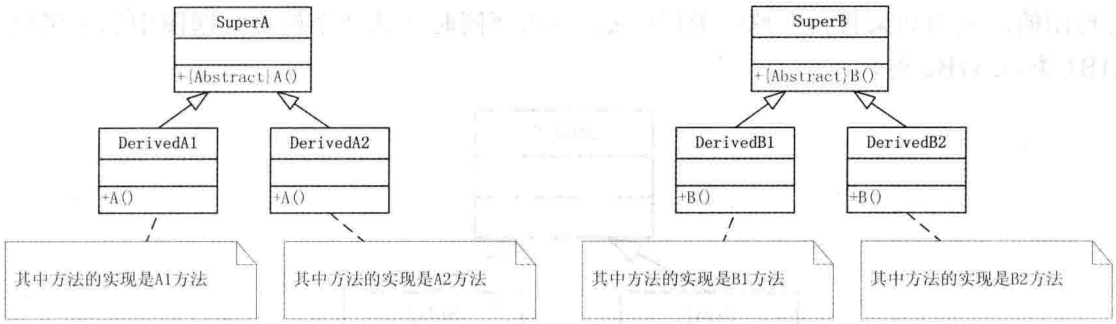


图 14.5 单一职责告诉我们把不同的变化封装到不同的类中去

这里暂时没有讨论涉及如何组织这两个类，第15到17章会详述。这种设计的好处是类的数量明显地减少了。

单一职责的原则很容易理解，通俗讲，就是把大的问题尽可能地分解成独立的小问题去解决。在程序设计中，我们一直都在自觉地使用，比如第 1 章经营面馆的例子（图1.13），经营面馆的职责被差分到店堂经理，服务员和厨师三个类当中去，其中任意一个类的变化都不会影响到其他类。

再比如第7.7节的图7.3，我们把 GUI 的职责分解成了三部分，分别是模型、视图和控制器，让它们各司其职。可以想象，如果我们把这三个类的功能硬塞到一个类当中去，这个类既要包含业务逻辑和内部数据，又要包含界面设计，还要给界面上的控件设置响应函数，如图14.6所示，这会是多么糟糕的一个设计[①]。

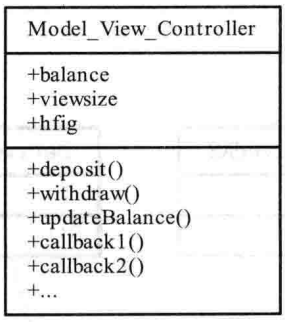


图 14.6 一个糟糕的无所不纳的 GUI 类

14.2 开放与封闭原则

> 程序的设计应该对修改是封闭的，对扩展是开放的。(Open-Closed Principles)

所谓对修改是封闭的，意思不是说程序一旦写好了就再也不用修改了，而是说，如果程序需要被修改，那么被修改的部分应该可以被隔离出来，并且对这部分的修改不会引起连锁反应，即影响到其他已有的模块。也就是说，不能出现“牵一发而动全身”的情况。在第1.3节中，使用的 switch 语句的 order 函数，就是面向过程编程中对修改不封闭的典型。在面向对象的设计中，如果不注意，也会违反开放封闭原则。如图14.7所示，该设计对修改不

①GUIDE 正是做了类似的工作。

是封闭的，因为如果我们要修改 B1 方法，将需要同时修改两个模块，即图中加了黑框的 A1B1 类和 A1B2 类。

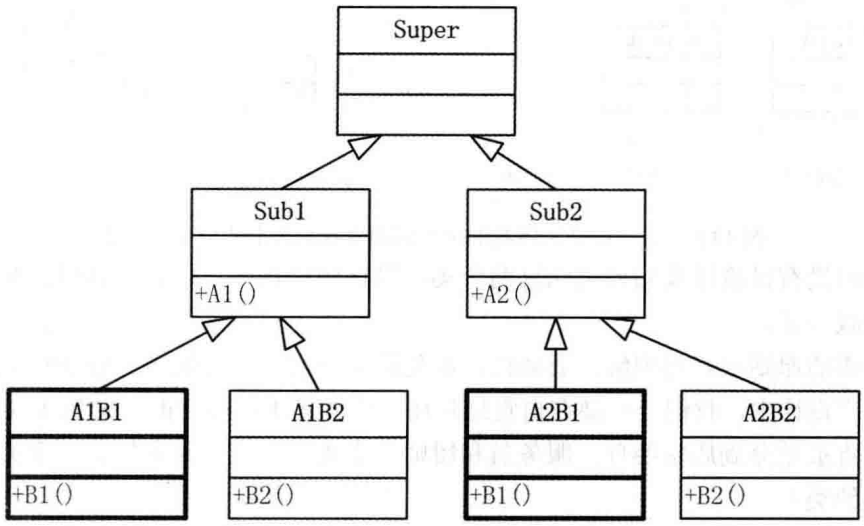


图 14.7 修改 B1 方法将需要修改两个类（A1B1，A2B1）

而图 14.8 所示的设计是对修改封闭的，因为对任意“DerivedA1，DerivedA2，DerivedB1，DerivedB2”的修改只会影响到一个封闭的模块。

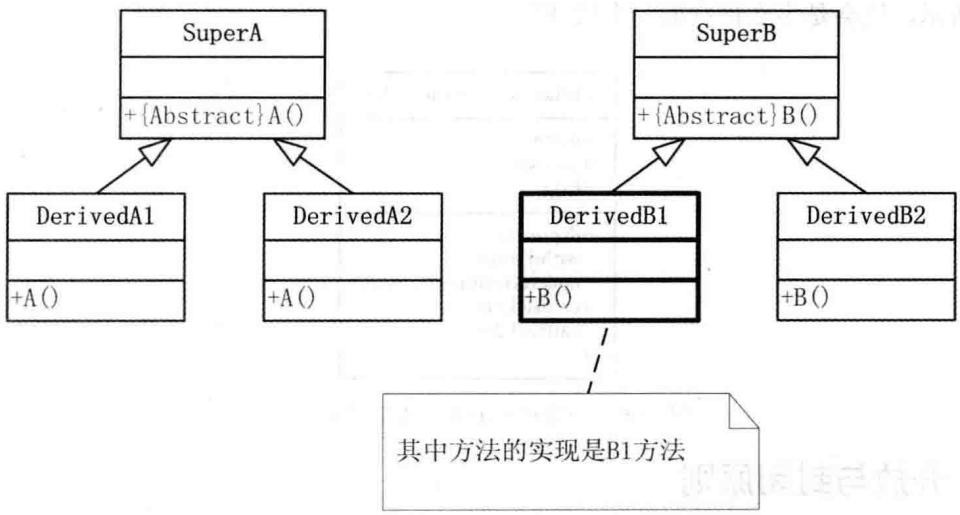


图 14.8 封闭的设计：一个方法的修改不会影响到其他模块

所谓对扩展开放，就是说，当新的需求到来时，添加新的模块不会影响已有模块。如图14.9中添加 DerivedA3 类，应该对已有的代码带来的改动是最小的。关于如何把程序设计得对扩展开放，会在后面的模式中详述。

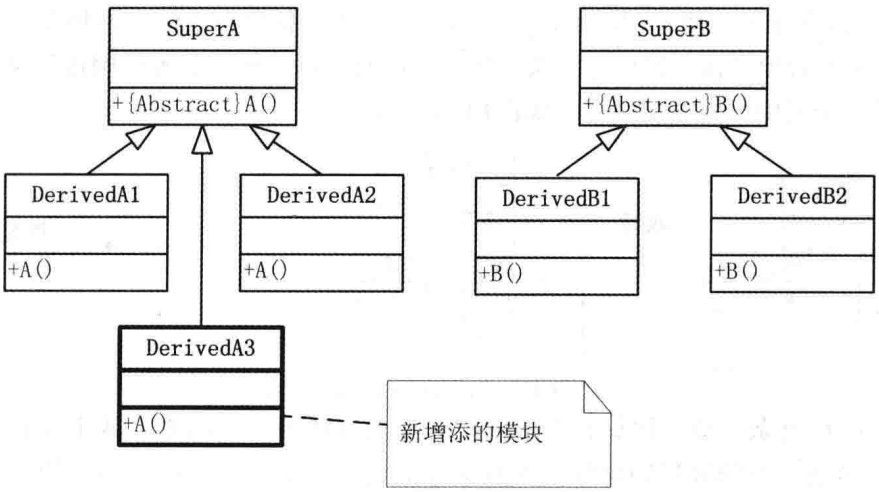


图 14.9 DerivedA3 类的添加不会影响到已有的类

14.3 多用组合少用继承

使用组合可以让系统有更大的弹性，不仅可以将算法族封装成类，还可以在运行时动态地改变对象的行为。

这里用图14.10来说明继承和组合之间的转换。假设 Super 类有两个方法 A 和 B。也可以理解成 Super 类的对象具有两种行为 A 和 B，可以像图14.10左一样把这两个方法封装在 Super 类里面，也可以换个角度，把这两个方法封装在两个方法类当中，然后让 Super 类拥有这个方法类的对象，如图14.10右所示。

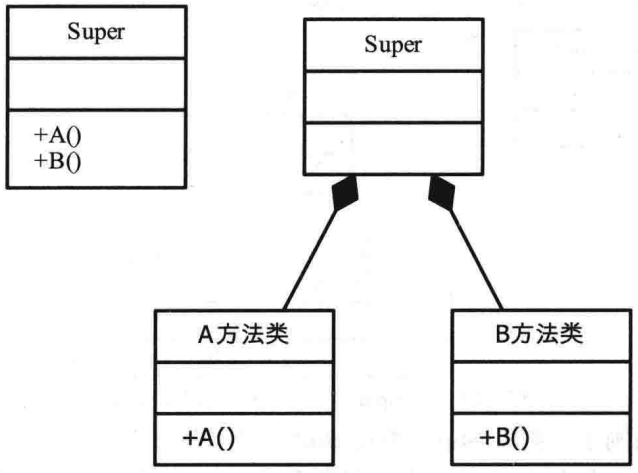


图 14.10 继承和组合之间的转换

如果程序仅仅是这么简单，我们还看不出组合的好处。现在假设 A 方法分化成 A1 方法和 A2 方法，它们相似但是不相同，使用继承的设计方案，UML 将变成图14.11左图。Derived1 和 Derived2 是 Super 的子类，具有不同的行为。更常见的做法是在 Super 类中声明一个抽象

方法 A[①]，并且在子类中用不同的方法去实现 A。由于含抽象方法的类是抽象类，不能直接声明对象，子类必须具体实现这个抽象方法才能声明出对象来，把 A 声明成抽象方法，可以保证（约束）子类中一定要实现之，如图14.11右所示。

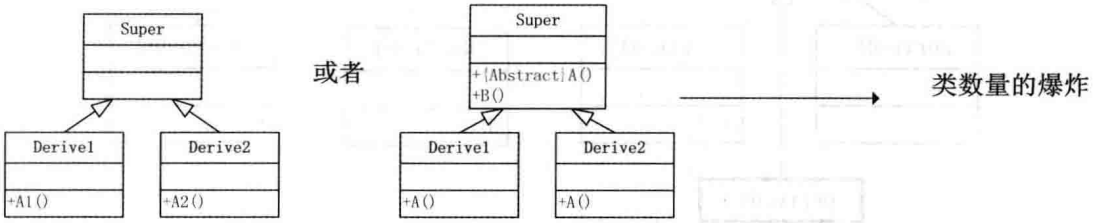


图 14.11 如果 A 方法出现分化

如果程序再复杂一点，比如把 Super 类中 B 的方法再分化出来，就出现我们在单一原则中出现的问题，类的数量的爆炸，并且这样的设计对修改不封闭。但是使用组合处理变化起来就要容易得多，如果要扩展方法，直接定义一个新的子类即可。从语言上，我们说 Super 类的对象拥有某种行为；从代码上，这句话的意思是，Super 类中拥有方法类的对象，如图14.12所示。

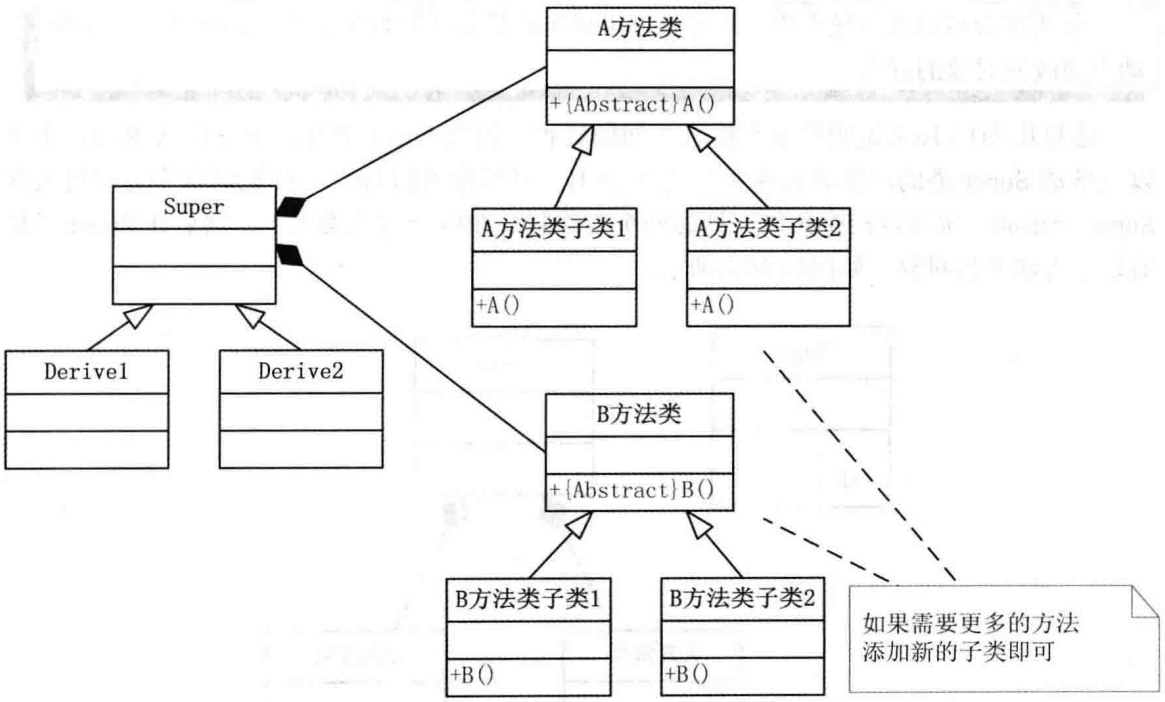


图 14.12 Super 类对象拥有方法类对象

再举一个经典的例子。假设 Super 类是 Bird，我们要讨论的行为是 fly，鸭子是鸟，鸭子会飞。企鹅也属于鸟，但是企鹅不会飞，在前面的章节中我们讨论过 fly 的行为必须针对不同的对象来定制，如果使用继承来处理这个问题，设计如图14.13所示。

这样的设计可以解决问题，但不是最佳的设计，使用我们刚学到的“多用组合少用继承”，把 fly 作为一种行为封装到方法类当中去，设计可以成为如图14.14所示的。

①原因参见第14.4节。

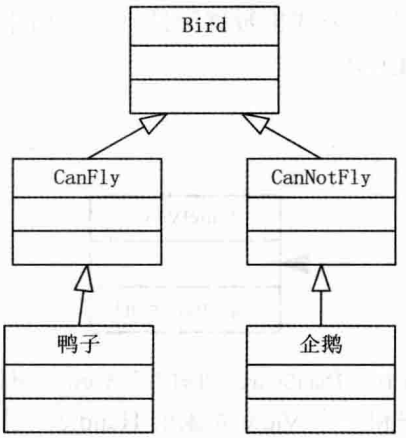


图 14.13 使用继承解决企鹅和鸟之间的关系

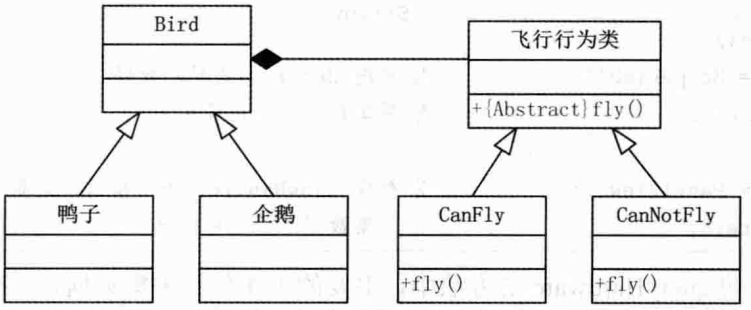


图 14.14 使用组合解决企鹅和鸟之间的关系

14.4 面向接口编程

我们先给接口一个形象的定义：假设程序中包含如图14.15所示的类，从 UML 图上来看，可以说接口（Interface）其实就是模块中的上层部分，即 Base1，Base2 基类。在好的面向对象程序设计中，上层模块 Base1 和 Base2 通常都是包含抽象方法的抽象类，而继承它们的子类要提供这些方法的实现。通常，我们也把这些子类叫做对接口（Interface）的实现（Implementation）。

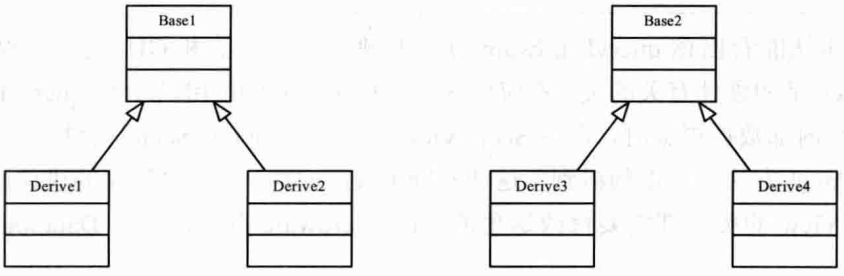


图 14.15 接口就是 UML 图中的上层结构

现在以一个例子来说明在面向对象程序设计中，接口是如何引入的，以及什么叫做面向接口的编程。假设有一个类叫做 DataSource，该类负责和硬件通信和采集硬件的数据，并且保存。每一次采集，数据发生变化，根据用户的选择，程序可以用图形的方式把数据可视化

(如用 ScopeView)，也可以数字的方式把数据简洁地表示出来（比如用 PanelView)。程序初步的设计包括三个类，如图14.16所示。

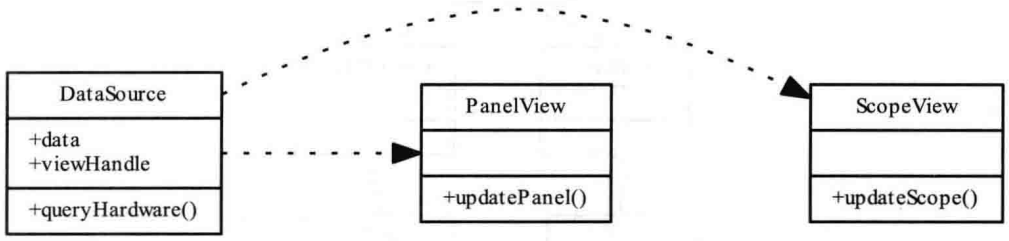


图 14.16 DataSource 通知两个 View 类更新显示

DataSource 拥有 data 数据和一个 View 对象的 Handle。根据该 viewHandle 的不同，采用不同的显示数据的方式，可以用如下脚本来测试。

Script

```
obj = DataSource()
obj. viewHandle= ScopeView();          % 采用 Scope 方式显示数据
obj.queryHardware();                   % 采集数据，并且可视化

obj. viewHandle= PanelView();          % 替换 viewHandle，采用精简方式显示数据
obj.queryHardware();                   % 采集数据，并且可视化
```

DataSource 的 queryHardware 的方法中，主要的工作包括采集数据，还有通知 GUI 刷新，向 GUI 传递数据。

queryHardware

```
function queryHardware(obj)
    %  程序查询硬件    内部数据更新
    if isa(obj.guiHandle,'ScopeView')
       obj.guiHandle.updateScope(data);     % 包含细节 updateScope
    elseif  isa(obj.guiHandle,'PanelView')
       obj.guiHandle.updatePanel(data);     % 包含细节 updatePanel
    end
end
```

我们立即就能看出该 queryHardware 方法的缺陷。该方法和 GUI 的细节结合得过于紧密，DataSouce 是和硬件有关的类，不应该包含任何关于 GUI 的细节，而 queryHardware 方法中，却要其在内部数据更新时，调用 ScopeView 类对象的 updateScope 方法，或者 PanelView 中的 updatePanel 方法，可以预料到，这些不同的 View 类，会随着程序的进化而变化，每次新添加一个 View 的类，都需要修改这里的 queryHardware 方法，这个 DataSource 类设计得很不方便。

解决方法是对 ScopeView 和 PanelView 加以抽象，抽象出一个 BasicView 基类出来，规定该 View 基类中含有一个抽象方法 update，各个 View 子类必须实现自己的 update 方法，如图14.17所示。

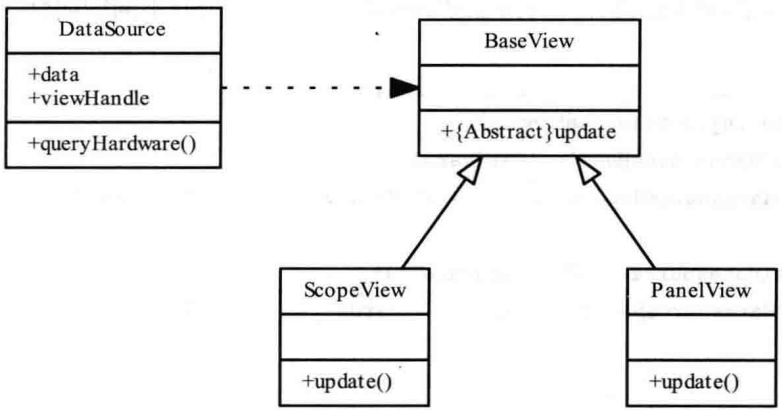


图 14.17 queryHardware 方法对 BaseView 接口编程

这样 queryHardware 类就可以简洁些了。

queryHardware 方法

```
function queryHardware(obj)
    % 程序查询硬件   内部数据更新
    if  isa(obj.guiHandle, 'BaseView')    % 使用接口类的名字
        obj.guiHandle.update(data);       % 使用接口类中规定的方法的名称
    end
end
```

我们曾把接口描述成是 UML 类图中的上层接口，现在我们观察一下 queryHardware 方法，发现其中出现的类的名称是上层类 BaseView，出现的方法的名称是上层类规定抽象方法 update 的名称，所以这个 queryHardware 方法给我们最直观的印象是：适用于更广泛的情况。用面向对象的术语讲，这就叫做面向接口（上层模块）的编程，细节依赖于抽象。

我们再扩充这个程序，进一步阐释面向接口的编程方式。假设有两个硬件，各自和 MATLAB 通信的方式不同，所以要设计两个 Hardware 类，分别叫 Hardware1 和 Hardware2，如图14.18所示。假设每次采集数据，数据量很大，把数据直接发送给 GUI 类不现实，所以需要把对象的 Handle 直接传给 View 对象，而不是发送数据，我们先提出一个初步设计，再一步一步地改进。还有，我们故意把 Hardware1 和 Hardware2 中的属性名称和方法名称设计得有一些不同，是为了体现抽象出一个基类的必要性。

其中 queryHardware1 方法需要把自己的 Handle 传给 View 对象。这样，View 对象可以直接访问 Hardware1 对象中的数据：

queryHardware1

```
function queryHardware1(obj)
    % 程序查询硬件   内部数据更新
    if  isa(obj.guiHandle, 'BaseView')
        obj.guiHandle.update(obj);        % 向 View 对象传递自身的 Handle
    end
end
```

而 update 方法，可以根据传来的 Handle 的类型的不同，访问不同的属性，比如 ScopeView 的 update 方法。

update

```
function update(obj,datasourceHandle)
        if isa(datasourceHandle,'Hardware1')
       % 查询 datasourceHandle.data1, 访问 Hardware1 对象的属性 data1

    elseif isa(datasourceHandle,'Hardware2')
       % 查询 datasourcehandle.data2, 访问 Hardware2 对象的属性 data2

    end
end
```

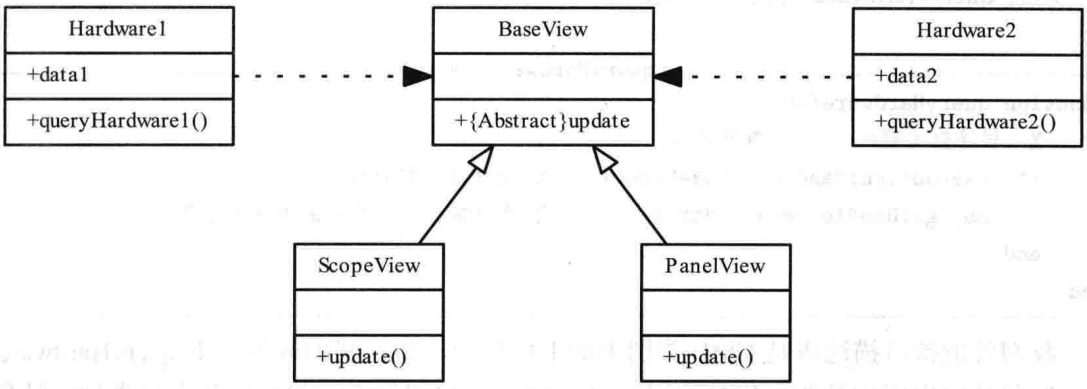


图 14.18 新增一个 Hardware 类

和第一个版本的 queryHardware 方法一样，我们看出了这个方法的设计缺陷：update 方法过分依赖于细节。因为两个 Hardware 类都很相似，现在我们抽象出一个 DataSource 基类来形容它们，如图14.19所示。

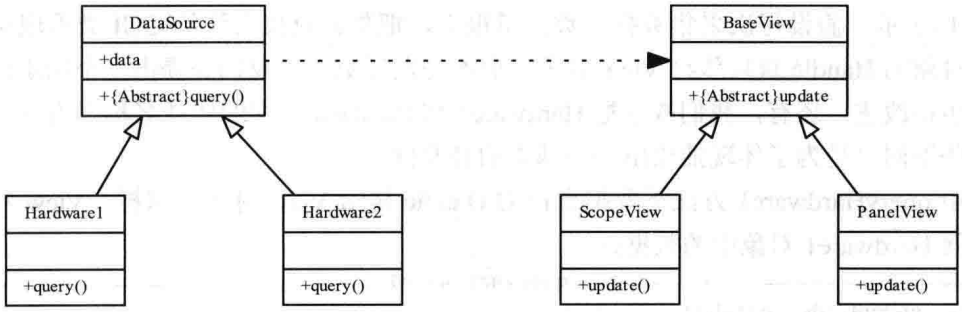


图 14.19 下层类面向上层接口编程

这样，我们的 View 子类中的 update 函数也是依赖抽象面向接口的了。其细节就是子类，就是 UML 中的下层结构，而抽象就是 UML 中的上层结构，即各个基类。

```
function update(obj,datasourceHandle)
        if isa(datasourceHandle,'DataSource')
```

```
                % query datasourceHandle.data, 访问 Hardware 对象的属性 data
    end
end
```

在这节的最后，我们再讨论一下，为什么要把子类中的方法名称抽象出来，放到基类中变成一个抽象方法。现在我们假设有一个 Client 类拥有或者使用 Derived1 或 Derived2 的对象，如果在 Derived1 和 Derived2 中相似的方法有着不同的名字（如图14.20所示），那么在 Client 处的代码势必将出现 obj.A1(),obj.A2() 类似的调用，这叫做高层模块依赖于底层模块，抽象依赖于细节，也叫做针对实现编程，是要避免的。因为 A1 和 A2 是具体的实现，万一要修改，会迫使高层 Client 模块也要做相应的修改。理想的情况是高层的代码应尽量地针对接口编程，即写得宽泛抽象，底层模块可以自由地修改、扩展，而不影响高层。正确的做法，即我们提到的更常见的做法是，在 Super 类中声明一个抽象方法 A，如图14.21所示。

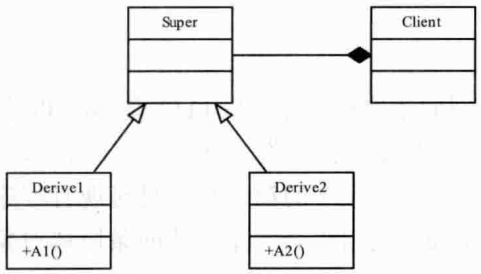


图 14.20 如果子类中相似的方法有不同的命名

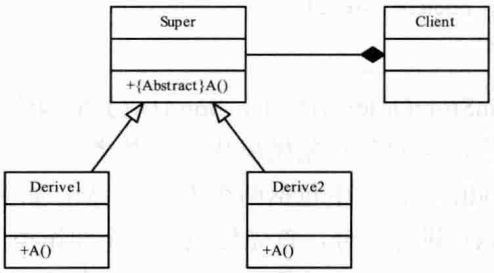


图 14.21 针对接口的设计

这样，在 Client 中的代码块就可以用 obj.A() 的调用了。Client 中的代码只和一般的 Super 类的对象做交互，并不需要知道该对象到底是 Derive1 类还是 Derive2 类，类似obj.A() 的调用，到底是 Derived1 对象的 A 方法的调用，还是 Derived2 对象的 A 方法的调用，Client 中的代码并不需要了解，这才是针对接口编程，也叫做依赖倒转原则：抽象不应该依赖于细节，细节应该依赖于抽象。

第 15 章　创建型模式

15.1　工厂模式：构造不同种类的面条

15.1.1　简单工厂模式

假设你是一个面馆的老板，在大学门口租了一个店面卖面条。现在要求用 MATLAB 程序来模拟面馆的运作过程。因为面馆初期很简单，所以程序也很短，只需要设计一个点菜函数就可以了。

inStoreOrder.m

```
function noodle = inStoreOrder()
    noodle = Noodle();
    noodle.boil();
    noodle.serve();
end
```

Command Linep

```
>> noodle = inStoreOrder();
```

在 inStoreOrder 函数中，`Noodle()` 是构造函数，返回面条原料对象；`boil()` 是 Noodle 类的方法，对面条对象做操作，将其煮熟；`serve()` 方法代表摆盘上菜。目前，程序只有一个 Noodle 类，其中包括两个方法，UML 如图15.1 所示。改进程序的第一步，是对现有程序根据职责进行拆分，首先构造出一个面馆类 NoodleHouse，使用组合关系，让面条原料对象作为 NoodleHouse 对象的属性，并且把点菜、煮面条和上菜方法也封装到 NoodleHouse 类中，如图15.2所示。

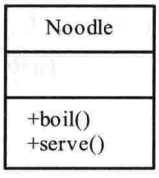


图 15.1　简单的面条类

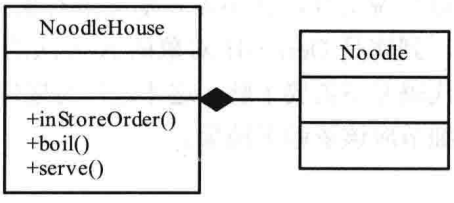


图 15.2　组合关系：面馆对象拥有面条对象

图15.2对应的 NoodleHouse 的代码如下：

NoodleHouse

```
classdef NoodleHouse < handle
    properties
        noodle % Noodle object
    end
    methods
        inStoreOrder(obj,orderType);
        boil(obj)  ;
        serve(obj);
    end
end
```

下一步是继续丰富面馆的菜单。小面馆刚刚开张，菜单暂时提供两种面条：风味可口的牛肉面和价廉物美的清汤面，NoodleHouse 对象负责判断产生牛肉面还是清汤面对象，如图15.3所示，

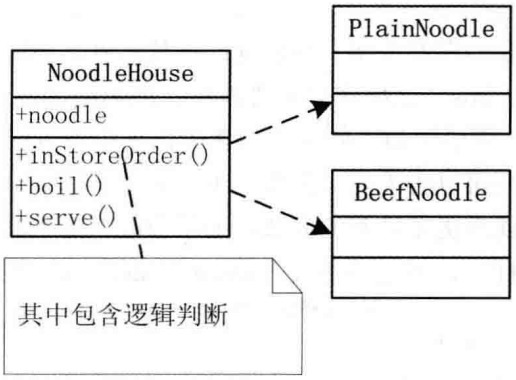


图 15.3 NoodleHouse 的 inStore 方法负责判断产生哪种面条对象

NoodleHouse 的 inStoreOrder 代码如下：

inStoreOrder

```
function  inStoreOrder(obj,orderType)
    switch lower(orderType)
        case 'beef'
            obj.noodle = BeefNoodle();
        case 'plain'
            obj.noodle = PlainNoodle();
        end
    obj.prepare();
    obj.boil();
    obj.serve();
end
```

其中包含 switch 判断语句，根据顾客的要求分别做不同的面条。

没过几个星期，面馆又发生了新的变化。老板招了一个四川厨师，做担担面很拿手；很多顾客反映牛肉面很好吃，但都不爱吃阳春面；还有的顾客要求菜单上增加炸酱面（起个名字叫做 Fried Sauce）。于是，我们赶紧修改 inStoreOrder 方法：

inStoreOrder

```
function inStoreOrder(obj,type)
    switch lower(type)
        case 'beef'
            obj.noodle = BeefNoodle();
        % case 'plain'                    % 阳春面卖得不好 决定从菜单上去掉
        %    noodle = PlainNoodle();
        case 'dandan'
            obj.noodle = DandanNoodle();
        case 'friedsauce'                 % 添加炸酱面
            obj.noodle = FriedSauceNoodle();
```

```
        end
    obj.boil();
    obj.serve();
end
```

程序改进到第三版，我们渐渐发现一个问题：每从菜单上增加或者去掉一个品种，不但需要增加一个面的种类的 Class（比如这里需要添加一个 FriedSauceNoodle 类），而且需要修改这个 inStoreOrder 方法。简单地说，就是添加新的需求很不方便。好在这个类的其余部分没有变化，程序侥幸似乎还可以再凑合一阵儿。可是又过了几个星期，面馆的生意越做越红火，周围居民希望菜单上能有更多种类的面条，如辣椒面、热干面、炒面、捞面等。我们的程序将不得不变成下面的样子，再这样下去，inStoreOrder 方法就要爆炸了！

inStoreOrder

```
function inStoreOrder(obj,type)
    switch lower(type)
        case '...'
             ...
        case '...'
             ...
        case '...'
             ...
        case '...'
            ...
        case '...'
            ...
        ......
    end
    obj.boil();
    obj.serve();
end
```

改进这个程序的第一步是：根据单一职责原则，把面条的生产和面馆的其他方法分开来，让它们各司其职。可以把产生面条原料的方法抽象出来成为一个类，该类有一个方法createNoodle()，负责制作各种不同的原料。假设辣椒面、热干面、炒面、捞面，不但烹制的方法不同，而且使用的面条的种类也不同。可在如图15.4所示的 UML 中，添加一个简单的面条工厂类，由工厂负责产生原料，再引入一个面条的基类，其中包括一些面条的共同的性质（这里从略）。因此，inStoreOrder 方法就不用再负责面条生产的细节了。

NoodleHouse

```
classdef NoodleHouse
    properties
        factory
        noodle
    end
    methods
```

```
        function obj = NoodleHouse(factory)
            obj.factory = factory ;   % 初始化面条工厂
        end
        function  noodle  = inStoreOrder(type)
            obj.noodle = obj.factory.createNoodle(type);  % 生产的工作由工厂完成
            obj.boil();
            obj.serve();
        end
    end
end
```

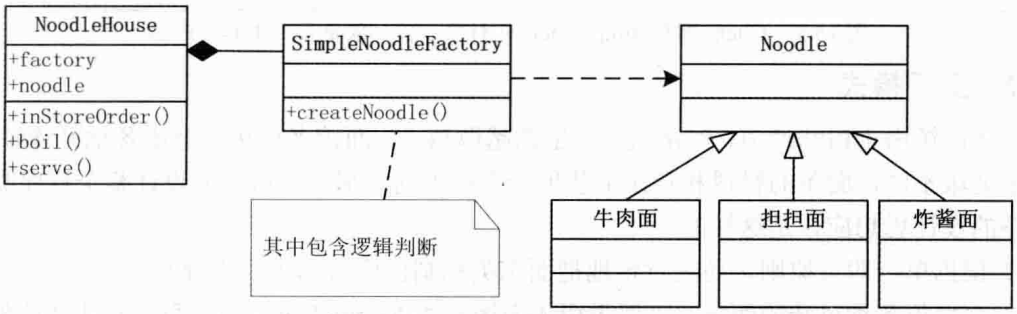


图 15.4 面馆有简单工厂的对象，由该对象负责生产面条

通过简单工厂，我们把生产面条的逻辑细节从上层模块的类方法 inStoreHouse 中，转移到了底层模块 SimpleNoodleFactory 类中，代码如下：

SimpleNoodleFactory

```
classdef SimpleNoodleFactory < handle
    methods
        Function noodle = createNoodle(orderType)
            switch lower(orderType)
               case 'beef'
                   noodle = BeefNoodle();
               case 'dandan'
                   noodle = DandanNoodle();
               case 'friedsauce'
                   noodle = FriedSauceNoodle();
             end
        end
    end
end
```

这种设计方法叫做简单工厂模式（Simple Factory Pattern）。其主要特点是：对象的产生细节由一个特定的类负责，并且该类中包含了必要的逻辑判断以产生不同类的对象。比如，createNoodle 方法根据 orderType 实例化出不同的对象。图 15.5是简单工厂的一般 UML 图。而该模式的优点是简单，NoodleHouse 类把负责生产面条的职责隐藏到了更底层的 SimpleFactory 模块中去，于是高层模块中的 inStoreOrder 方法不再跟随菜单的变化而变化；

该设计仍有不足之处，即简单工厂类这样的细节模块中仍然包含了各种具体产品的逻辑判断，但是需要认识到，这种逻辑是必需的，有时是无法避免的。好的设计中，这些细节存在于底层模块中，以达到高层模块和底层细节的解耦合。

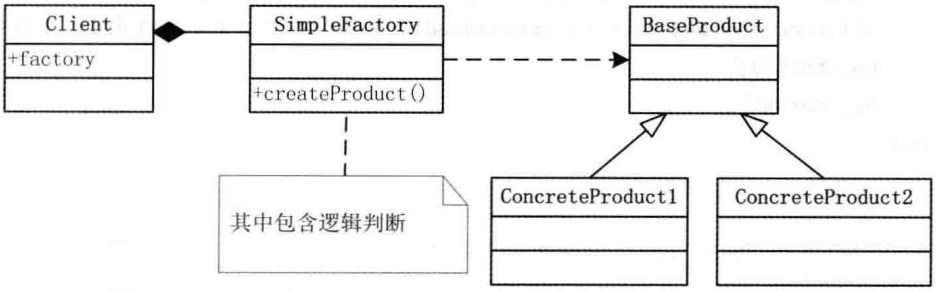


图 15.5 Client 拥有 SimpleFactory 对象，该对象负责产生具体产品

15.1.2 工厂模式

现在面馆由于口味地道，经济实惠，生意越做越火，面馆老板想在全国各地开连锁店了，但由于地域不同，面条的材料和制作工艺也会有所不同。那么，该如何设计整个程序呢？整个程序的设计思想应该是这样的：

- 根据单一职责原则，还是肯定地把面馆类和制作原料的工厂类分开。
- 要模拟各种风味的面馆，可以先引入面馆基类 NoodleHouse，把面馆的共性抽象出来以达到代码的复用；同理，还可以引入一个 Factory 类，把各个工厂的共性抽象出来。
- 具体的 NoodleHouse 的对象将拥有原料工厂的实例[①]，比如南方 NoodleHouse 对象，将拥有南方风味面条加工厂的实例。
- NoodleHouse 类中，应该有一个 createNoodle 方法，并且这个方法仅仅是一个包装方法（wrapper），该包装方法负责把构造面条的请求转发给具体的工厂对象。

这样，简单工厂模式就升级成了工厂模式，多了 Factory 基类和 Factory 具体类，如图15.6所示。

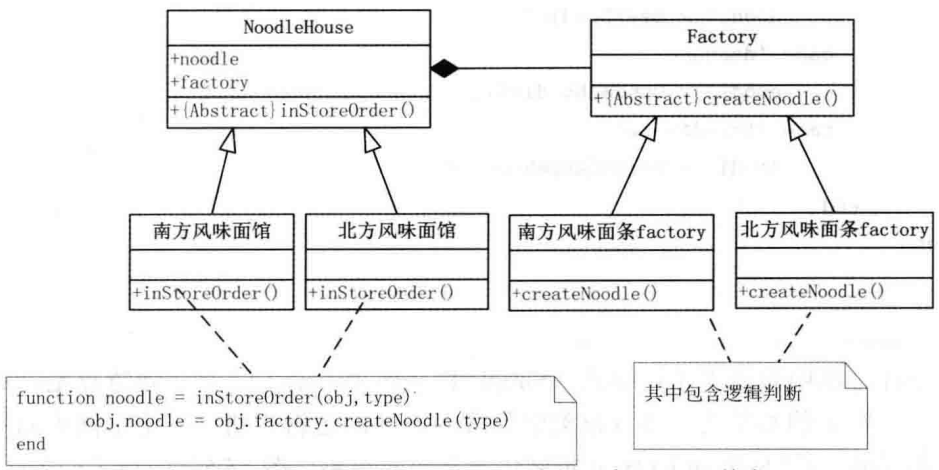


图 15.6 简单工厂升级成工厂，多了一个 Factory 基类

①OOP 中，实例（Instance）和对象（Object）是同义词。

不可避免，在具体的原料工厂类中仍将含有逻辑判断语句：根据具体（Concrete）的NoodleHouse传来的参数构造具体的面条对象，如图15.7所示。

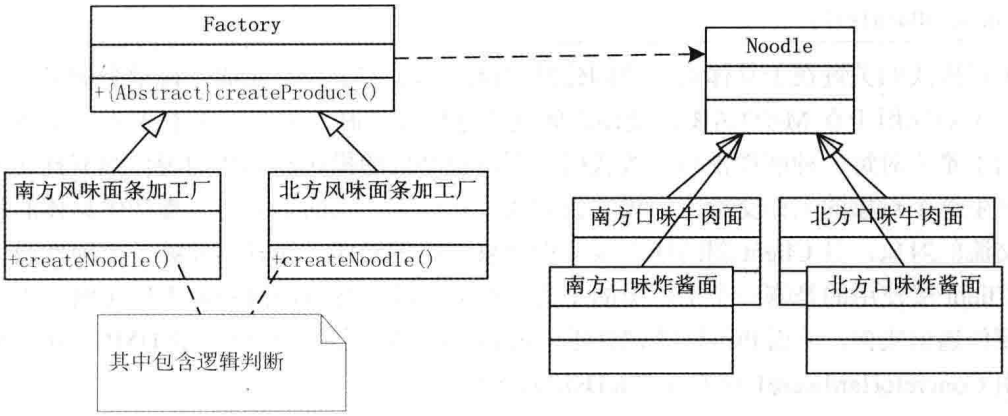


图 15.7 具体的工厂子类负责生产具体的产品

比如南方风味和北方风味的面条工厂，它们的 createNoodle 方法的定义是这样的：

```
function noodle = createNoodle(obj,type)
 switch type
  case 'beef'
    noodle = SouthBeefNoodle();
  case 'sauce'
    noodle = SouthSauceNoodle();
  end
end
```

```
function noodle = createNoodle(obj,type)
 switch type
  case 'beef'
    noodle = NorthBeefNoodle();
  case 'sauce'
    noodle = NorthSauceNoodle();
  end
end
```

整体设计的 UML 如图15.8所示。

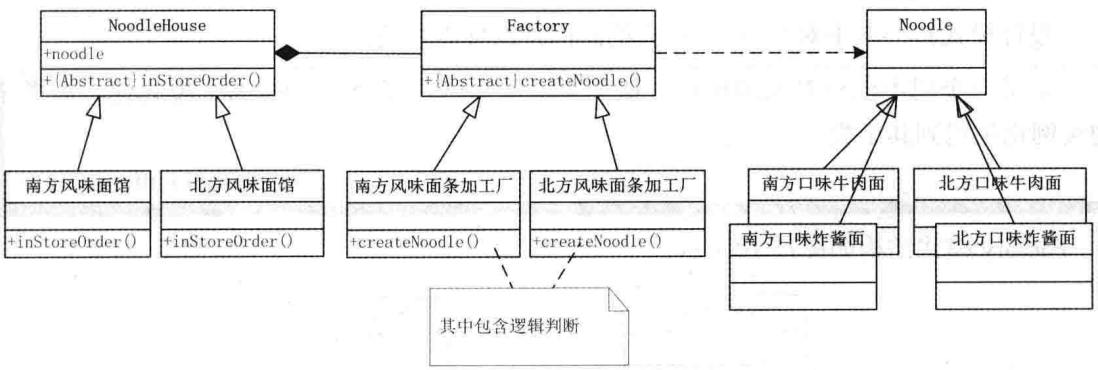


图 15.8 具体面馆对象将拥有具体的面条加工厂对象

该设计的要点是：

- NoodleHouse 对象拥有原料工厂的实例，原料工厂根据 NoodleHouse 的要求产生不同的面条原料。
- NoodleHouse 也可以更换原料工厂实例，从而产生不同的面条原料。

测试程序负责声明面条工厂和面馆对象，并且指定面馆使用的工厂对象。

```
Script
factory  = NorthNoodle();
house = NorthNoodleHouse(factory);   % 指定 NoodleHouse 使用的工厂
house.createNoodle();
```

工厂模式的关键在于具体对象的创建时机推迟到工厂子类中完成。再举个例子，比如要构造一个对象用于在 MATLAB 和硬件之间进行通信和控制，因为可能有不止一个硬件，而且我们不希望对每一种硬件都写一段代码，不希望高层的模块和具体的硬件细节打交道，所以把产生这个对象的工作交给工厂模式去完成。那么工厂类的子类将负责产生具体的和硬件直接交流的对象，且 Client 端的代码则负责抽象地控制硬件，而具体和哪一个硬件通信，取决于 Client 端使用的是哪一个工厂类的子类，在运行时，可以通过更换工厂实例，来产生不同的硬件通信实例，从而和不同的硬件进行通信。图15.9所示为该例子的 UML，通信交流的对象由 ConcreteHardware1 和 ConcreteHardware2 产生。

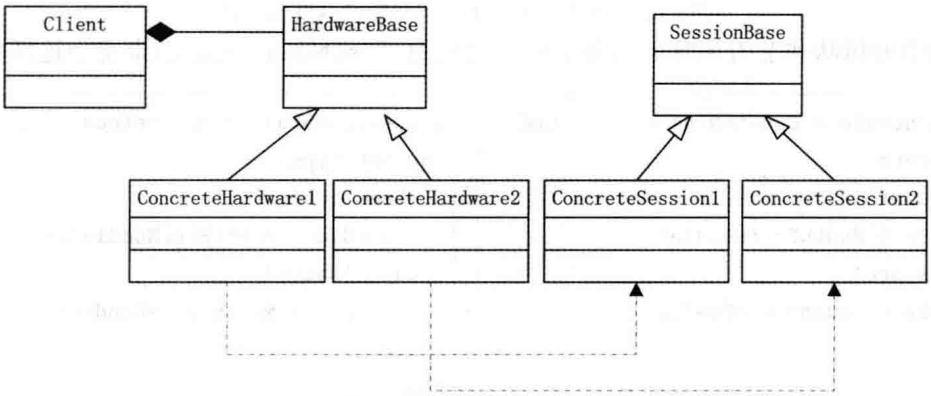


图 15.9 工厂模式实例

15.1.3 Factory 模式总结

《设计模式》一书中对 Factory 模式的意图是这样叙述的：

> 定义一个用于创建对象的接口，让子类决定实例化哪个类。Factory 模式使一个类的实例化延迟到其子类。
>
> Design Patterns GOF

Factory 模式结构如图15.10所示。

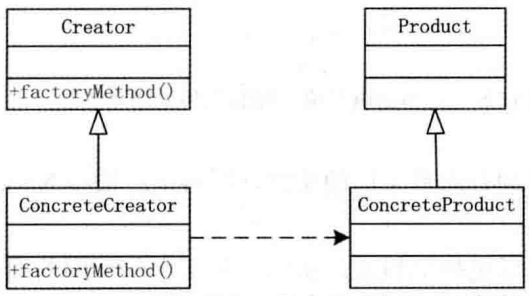


图 15.10 Creator 对象要创建的产品对象延迟到了其子类

Creator 依赖它的子类，其子类将通过调用具体的工厂对象得到一个适当的 Concrete Product 的实例。

15.1.4 如何进一步去掉 switch/if 语句

从图15.5的简单工厂模式到图15.8的工厂模式，生产对象的细节被分流到了底层的细节模块中。但在面条工厂的 createNoodle 方法中，仍然包含着 switch/if 判断语句，所以这还是没有完全做到对修改封闭。也就是说，该工厂的原料种类需要扩充时，对该方法的修改仍在所难免。究竟有没有可能把 switch/if 去掉呢？在 MATLAB 中这是可以做到的。为了演示这个技巧，我们先把面馆的例子简化一下，假设现在有两个类 Sub1 和 Sub2：

Sub1

```
classdef Sub1  < handle
    methods
        function obj = Sub1()
            disp('sub1 obj created');
        end
    end
end
```

Sub2

```
classdef Sub2 < handle
    methods
        function obj = Sub2()
            disp('sub2 obj created');
        end
    end
end
```

下面的 createObj 函数将根据输入的字符串来判断要构造 Sub1 还是 Sub2 的对象：

```
function obj = createObj(type)
    switch type
        case 'Sub1'
            obj = Sub1();
        case 'Sub2'
            obj = Sub2();
    end
end
```

初步可以这样测试这个函数：

Script

```
createObj('Sub1');
createObj('Sub2');
```

Command Line

```
sub1 obj created
sub2 obj created
```

createObj 函数中 switch 的使用，就是第15.1.2小节提到的“必要的逻辑判断”。也是希望能改进的地方。这也是对修改不封闭的部分。要达到这个目的，不论如何增加具体的 Sub 类的数量，createObj 函数都不受外部的影响（不需要被修改），在 MATLAB 中，可以用 eval 函数来实现这个要求。为简单起见，下面的代码舍去了对输入是否合法的验证[①]。新的 createObj 只有一行，不论传入的 classname 是什么样的字符串，eval 都将把这个字符串当做 MATLAB 命令去执行。当然，我们必须规定输入 classname 只能是类的名称的话，这个函数的作用其实就是动态地产生各种具体的对象。

①比如至少 MATLAB 要能找到该类的定义。

```
function obj = createObj(classname)
    obj = eval(classname);
end
```

测试还是和之前一样：

Script

```
createObj('Sub1');
createObj('Sub2');
```

Command Line

```
sub1 obj created
sub2 obj created
```

上述的例子要求 Sub1 和 Sub2 有缺省的 Constructor。如果恰好构造函数需要用户的输入，比如下面这个修改过的 Sub1 的定义：

Sub1.m

```
classdef Sub1 < handle
    properties
        a
    end
    methods
        function  obj = Sub1(var)
            obj.a = var;
        end
    end
end
```

用户可以使用 strcat 先构造要执行的命令的字符，然后再用 eval 函数来执行命令，得到产生的对象，比如：

Script

```
classname = 'Sub1';
cmd = strcat(classname,'(','10',')');
obj = eval(cmd);
```

或者可以使用 str2func 函数，从 classname 处获得类的构造函数的句柄，然后像正常使用构造函数那样那样调用该函数句柄。

Script

```
classname = 'Sub1';
ConstructorHandle = str2func(classname);
obj = ConstructorHandle(3);
```

15.1.5 抽象工厂

如果工厂生产的产品很复杂，如图15.11所示，牛肉面除了可以由不同种类的牛肉组成，比如红烧牛肉或牛腩，还可以由不同种类的面条组成；并且炸酱面里的酱也因为地域不同，可分成辣的和甜的；面条本身还可以分成宽边和细边的。这种情况下，还可以再进一步细化工厂类，如图15.12所示。

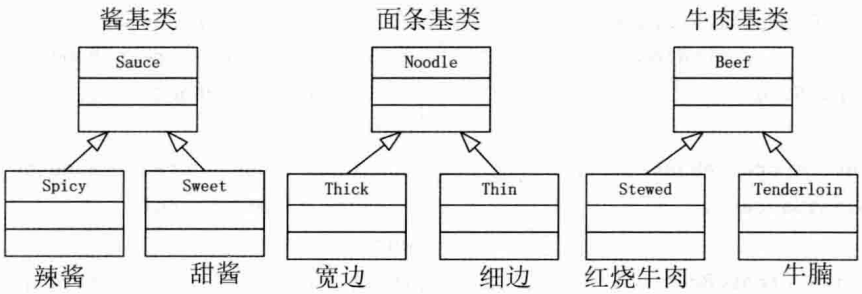


图 15.11 各种面食的组合结构

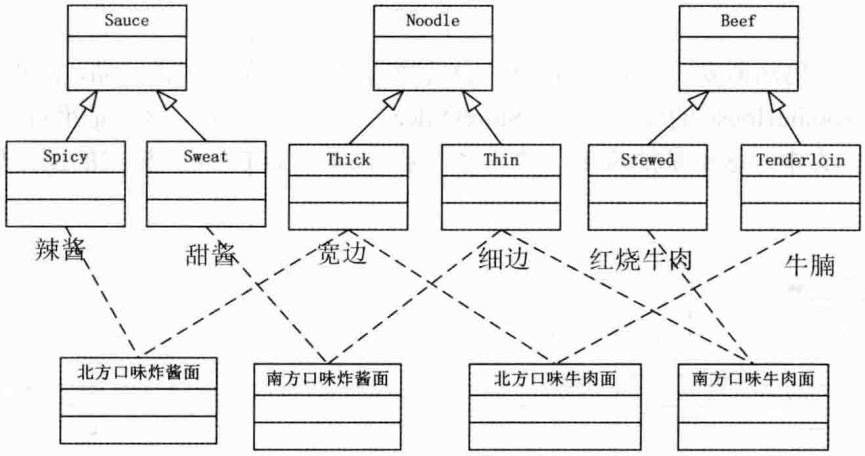


图 15.12 抽象工厂的进一步细化

而最后的产品是各种面条、酱类和牛肉对象的组合。比如，南方风味的牛肉面是干切牛肉加细边面条；北方风味的牛肉面则是牛腩加宽边面条；南方风味的炸酱面是甜酱加细边面条，北方风味的炸酱面是辣酱加宽边面条。

相应地，我们把 Factory 中的 createNoodle 细化成三个方法，如图15.13所示：由 createNoodle 来构造不同粗细的面条；由 createSauce 来构造不同的炸酱；由 createBeef 来构造牛肉辅料。

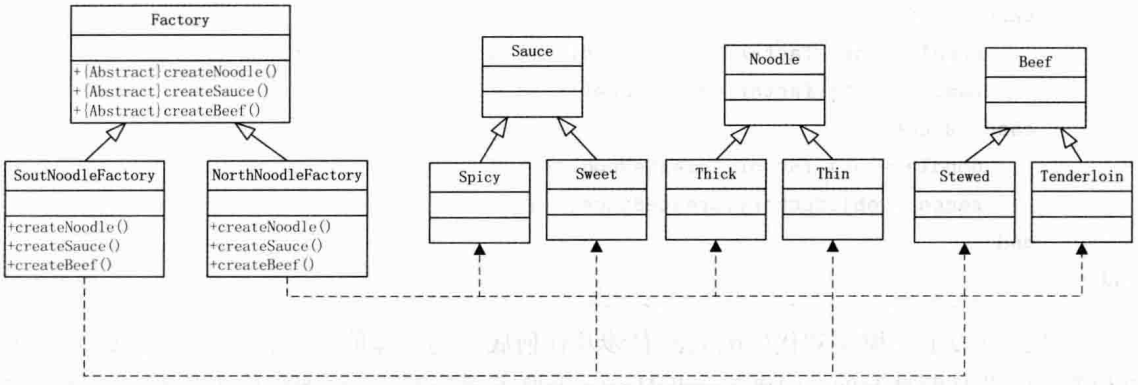


图 15.13 不同风格的面条工厂生产不同的面条、配料

具体工厂中的具体方法指定产生特定的产品，比如南方面厂的面条是细边，酱是甜的，而牛肉是红烧的。

SouthNoodleFactory

```
function noodle = createNoodle(obj)
  noodle = ThinNoodle();
end
function sauce = createSauce(obj)
  sauce = SweetSauce();
end
function beef = createBeef(obj)
  sauce = Stew();
end
```

SouthNoodleFactory

```
function noodle = createNoodle(obj)
  noodle = ThickNoodle();
end
function sauce = createSauce(obj)
  sauce = SpicySauce();
end
function beef = createBeef(obj)
  sauce = Tenderloin();
end
```

当然，我们仍然需要一个 switch/if 语句根据顾客所点的菜来提供产品，这次我们将该逻辑判断放到 NoodleHouse 的底层类的 inStoreOrder 方法中去。不过，这不是绝对的，也可以放到具体的工厂类中，这不是抽象工厂的关键。采用 Abstract Factory 模式的面馆如图15.14所示。

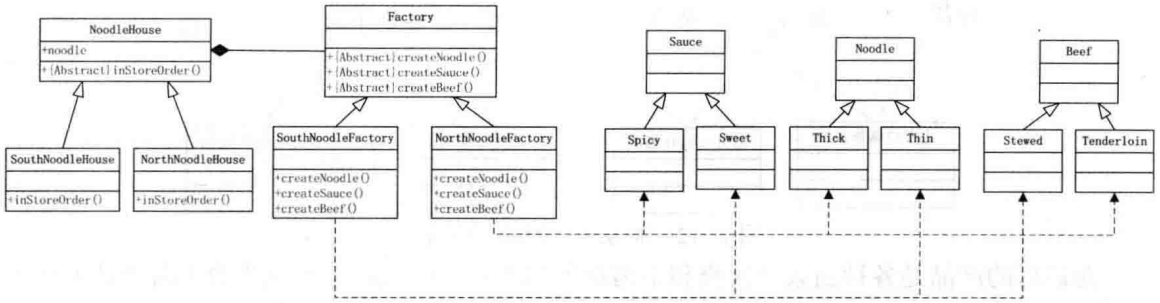


图 15.14 采用 Abstract Factory 模式的面馆

NoodleHouse 的具体类对象拥有具体的 Factory 的对象，所以构造何种菜肴全根据该具体的 Factory 的种类而定，inStoreOrder 方法可以写成这样：

inStoreOrder

```
function inStoreOrder(obj,type)
    switch type
        case 'beef'
            noodle = obj.factory.createNoodle(); % 不涉及何种具体的工厂
            beef =   obj.factory.createBeef();
        case 'sauce'
            noodle = obj.factory.createNoodle();
            sauce = obj.factory.createSauce();
        end
end
```

注意：这段上层模块的代码因为没有涉及任何底层的具体的工厂和产品，所以该上层模块和产品细节是解耦合的。如果 NoodleHouse 更换了面条工厂，这部分代码是不需要修改的。总地来说，NoodleHouse、Factory 和各种配料的关系构成了一个抽象工厂的模式。Abstract Factory Pattern 比 Factory Pattern 处理更加复杂的情况，具体的工厂的职责是构造多于一个的 Product。

15.1.6 Abstract Factory 模式总结

《设计模式》一书中对 Abstract Factory 模式的意图是这样叙述的：

> 提供一个创建一系列相关或者相互依赖的对象的接口，而无需指定它们具体的类。
>
> Design Patterns GOF

1. Abstract Factory 模式的结构

Abstract Factory 模式的结构如图15.15所示。

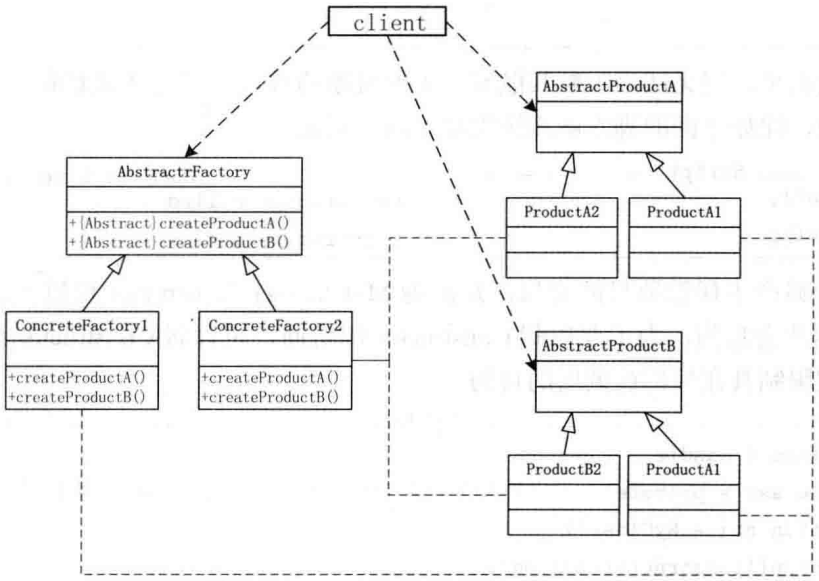


图 15.15 Abstract Factory 模式的结构

2. 类之间的协作

在运行时，Client 将负责创建一个 ConcreteFactory 类的实例，这个具体的工厂具有创建不同对象的代码。Client 可以更换具体的工厂以得到不同的具体的产品。

3. 何时使用 Abstract Factory 模式

Abstract Factory 模式适用于如下情况：

- □ 当一个系统要独立于它的产品创建、组合和表示时。
- □ 当一个系统由多个产品系列中的一个来配置时。
- □ 当需要强调一系列相关产品的设计，以便进行联合使用时。

15.2 单例模式：给工程计算添加一个 LOG 文件

15.2.1 如何控制对象的数量

用 MATLAB 进行工程科学计算时，常需要在过程中输出一些中间结果，用来调试程序。一般，我们把记录中间结果的文件叫做 LOG。对这个 LOG 文件的要求是，在整个程序的运行期间的任何地方，都能往 LOG 中写入数据，并且整个程序运行期间，有且只有一个 LOG。下面我们就用单例模式来解决这个问题。单例 Singleton 模式是设计模式中最简单的一种，也

是最常用的模式之一。该模式用来控制一个类所能产生的对象的数量。通常用来限制类只能产生一个对象。下面探讨其如何在 MATLAB 中实现。先观察如下一个简单的类 MyClass：

MyClass

```
classdef MyClass < handle
    methods
        function obj = MyClass()
            disp('Constructor called') % 打印文字表示 constructor 被调用
        end
    end
end
```

该类定义简单，但无法来限制其能够产生的对象的数量，因为外部 Client 可以调用构造函数任意次数，比如下面的脚本就连续生成了两个对象：

Script

```
obj1 = MyClass();
obj2 = MyClass();
```

Command Line

```
Constructor called
Constructor called
```

之所以能够产生任意数目的对象，是因为 MyClass 的 Constructor 可以不加限制地被调用，所以自然就会想到，为了要控制 Constructor 的访问，也许将 Constructor 声明成 private 方法可以达到限制其在外部被调用的目的。

MyClass

```
classdef MyClass < handle
    methods(Access = private)      % 如果构造函数被声明成了 private，外部将无法访问
        function obj = MyClass()
            disp('constructor called');
        end
    end
end
```

但是，我们很快就意识到，这样似乎根本没有机会构造出任何对象。所以这个问题到这里只解决了一半，剩下的一半是，我们还要提供一个中间层（方法）来间接地对 Constructor 进行访问。该方法除了必须是公共的方法之外，还必须满足如下条件：

- 该方法被调用时，如果 MyClass 还没有产生出一个对象，则该方法将产生一个对象。因为即使在 MyClass 没有被实例化之前，该方法也要允许被外部访问，所以这个方法必须是一个 Static 方法。
- 该方法内部要有一个标记，用来记录是不是已经产生过一个对象。
- 该方法被调用时，如果 MyClass 的对象已存在，则该方法返回上次产生的那个对象。

分析到这里，Singleton 的框架已经基本上清晰了。其 UML 图如15.16所示。

图 15.16 MyClass 类含有一个 private 的构造函数和一个 public 的静态方法

下面的代码是其经典的实现。

MyClass

```
1  classdef MyClass < handle
2      methods(Access = private)                          % 私有的构造函数
3          function obj = MyClass()
4              disp('construtor called');
5          end
6      end
7      methods(Static)
8          function obj = getInstance()                   % 静态的接口方法
9              persistent localObj;                       % Persistent local object
10             if isempty(localObj) || ~isvalid(localObj) % 如果 localObj 不存在创建
11                 localObj = MyClass();
12             end
13             obj = localObj;                            % 如果 localObj 已存在返回
14         end
15     end
16 end
```

说明：

- 因为 MyClass 的 Constructor 是 private 的，所以外部程序无法直接调用该 Constructor 进行实例化;getInstance 是一个静态方法，即使不存在类实例时，外部程序也可以调用该方法。
- 第 9 行，getInstance 静态方法内部有一个 persistent 变量，利用这个 persistent 变量来保存这个类的唯一的实例对象，所以每次该方法被调用时，都返回这个静态变量。
  - 如果 getInstance 第一次被调用，这时 localObj 还没有赋值，那么 getInstance 将产生一个对象，并且将它返回，这叫做实例化延迟，即仅在需要时再产生对象。
  - 如果 getInstance 再次被调用，这时 localObj 中已经存放了第一次被调用时产生的实例，getInstance 不是再创建新的对象，而是直接返回 localObj。
- Persistent 变量可以使用 clear all 来清除。一旦使用了 clear all 或者 clear classes，persistent 变量将不在存在，其中保持的数据也将消失。如果之后再次调用 getInstance 时，相当于第一次调用，将重新构造 MyClass 对象。

现在我们来验证这个设计。在下面的代码中 MyClass.getInstance 被调用了三次，从命令行的输出可以看出，Constructor 只被调用了一次。

Script

```
obj1 = MyClass.getInstance();
obj2 = MyClass.getInstance();
obj3 = MyClass.getInstance();
```

Command Line

```
construtor called
```

因为 MyClass 是 Handle 类，所以上述脚本中构造出来的 obj1，obj2，obj3 实际指向的都是同一个实例。

《设计模式》一书中对 Singleton 模式的意图是这样叙述的：

> 保证类仅有一个实例，并提供一个访问它的全局访问点。
>
> Design Patterns GOF

注意，Myclass 中的 constructor 是私有的，这将导致 MyClass 类不能被继承，因为其子类对象初始化时必须要能够访问基类 constructor。

15.2.2 应用：如何包装一个对象供全局使用

前面已经介绍了单例模式的基本实现，下面来讲解如何具体实现 LOG 类。如果使用面向过程的方法，需要在函数内部打开文件，写入数据，最后关闭文件，比如：

计算函数 1

```
1 function func1(filename)
2   fID = fopen(filename);  %  打开文件
3   fprintf(fID,'Hello from func 1\n')
4   fopen(fID);
5 end
```

计算函数 2

```
1 function func2(filename)
2   fID = fopen(filename);   %   再次打开
3   fprintf(fID,'Hello from func 2\n')
4   fopen(fID);
5 end
```

两个计算函数中第 2 行的代码重复了，而去除重复代码是改进程序最明显的一步。针对重复的 fopen，可以在这些函数的外部打开这个文件，然后把文件句柄传入，并且记得在程序结束处关闭这个文件句柄。

整个计算过程

```
......
fID = fopen(filenane);
......
func1(fID);
......
func2(fID);
......
fclose(fID);
......
```

如果需要记录许多函数的中间计算过程，给每个函数都添一个额外的参数显然是很不方便的。

接受句柄作为参数的计算函数 1

```
function func1(fID)
 fprintf(fID,'Hello from func 1\n')
end
```

接受句柄作为参数的计算函数 2

```
function func2(fID)
 fprintf(fID,'Hello from func 2\n')
end
```

如果这个文件句柄 fid 确实被频繁地使用，还有一些常见的方法，比如把这个句柄声明成全局变量，或者在主工作空间中声明这个变量，然后使用 assignin 在函数中获取主工作空间中的该变量。使用全局变量完成函数之间的数据共享是下策。下面我们介绍如何用面向对象的方法来对全局变量进行封装，以达到数据的共享的目的。

如果使用 Singleton 模式，可以把打开、关闭和写文件的操作封装到一个 LogClass 中去，只需要在各个函数内部调用 LogClass 的静态方法 getInstance，就可以得到统一的 Log 类对象，然后使用 print 完成输出。这样一来，就不需要再给每个函数添加参数，只要该类的定义在 MATLAB 的搜索路径上，Log 对象就可以在任何地方被使用。

使用 LogClass 的计算函数 1

```
function func1()
    log = LogClass.getInstance();
    log.print('Hello from func 1\n');
end
```

使用 LogClass 的计算函数 1

```
function func2()
    log = LogClass.getInstance();
    log.print('Hello from func 2\n');
end
```

LogClass 可以这样设计：

- 由私有构造函数负责打开 Log 文件；由 delete 成员方法负责关闭打开的文件。
- print 成员方法负责输出到外部。
- getInstance 是 static 方法，控制外部程序对该类的对象的创建和访问。

LogClass

```
classdef LogClass<handle
    properties
        fID
    end

    methods(Access = private)
        function obj = LogClass()    % Constructor 负责打开文件
            obj.fID = fopen('logfile.txt','a');
        end
    end

    methods
        function delete(obj)         % delete 方法负责关闭文件
            fclose(obj.fID);
        end
        function print(obj,string)   % print 方法封装 fprintf
           fprintf(obj.fID,string);
        end
    end

    methods(Static)
        function obj = getInstance() % static 方法控制外部的访问
           persistent localobj;
```

```
            if isempty(localobj) || ~isvalid(localobj)
                localobj = LogClass();
            end
            obj = localobj;
        end
    end
end
```

本书中 Singleton 的其他用例（Use Case）还有第7.8节的上下文类，第17.6节的 Caretaker 类。一般来说，Singleton 模式可以使用在那些计算中只需要一个对象的类上。在 GUI 编程中，Singleton 还可以用来控制一个类能够产生的视图（View）的数量。

15.3 建造者模式：如何用 MATLAB 构造一辆自行车

15.3.1 问题的提出

这节通过 MATLAB 模拟一个对象自行车的建造过程来介绍 Builder 模式。先对自行车做简化，假设自行车有三个主要部分：框架（Frame），轮子（Wheels）和坐垫（Seat）。然后把这三个部件进行抽象，引入一个部件基类，叫做 BikePart，自行车由 BikePark 的对象组合而成。如果用类图来表示自行车和部件的关系，如图15.17所示。

Bike 类和 BikePart 类定义如下，其中 Bike 的成员属性 parts 当做对象使用，用来盛放 BikeParts 对象；Bike 类中还有一个装配方法，叫做 addPart 方法，为简单起见，其功能仅仅是把各个配件添加到对象数组 parts 中去。

Bike

```
classdef Bike < handle
 properties
    parts
 end

 methods
  function addPart(obj,partObj)
    obj.parts = [partObj, obj.parts];
  end
 end
end
```

Part

```
classdef Part < handle
    properties
        name
    end

    methods
        function obj = Part(name)
            obj.name = name ;
        end
    end
end
```

有了 Part 对象，就可以开始装配自行车了。我们把装配过程放到一个叫做 constructBike 的方法中：

Script

```
function bikeobj = constructBike()
    bikeobj = Bike();
    bikeobj.addPart(Part('frame'));
    bikeobj.addPart(Part('wheels'));
    bikeobj.addPart(Part('seat'));
end
```

Command Line

```
>> constructBike()
ans =
  Bike handle
  Properties:
    parts: [1x3 Part]
  Methods, Events, Superclasses
```

该方法按照固定的顺序装配自行车：先添加框架，再轮子，最后添加坐垫。在命令行上调用 constructBike 方法，一辆普通的自行车就构造完毕了。

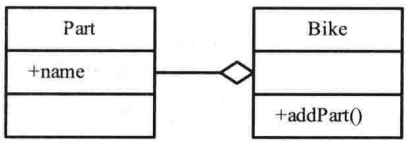


图 15.17 Bike 由 BikeParts 构成

下面我们继续添加新的需求。新的需求是：程序不但要能够制造普通的自行车，还要能构造山地自行车。这里假设：山地车配件的材料和普通自行车的材料不同，比如山地车的框架更坚硬，轮子更粗，坐垫更舒适，现在一个新的构造山地自行车的函数如下（它和 constructBike 没有太大的区别）：

```
function bikeobj = constructMountainBike()
    bikeobj = Bike();
    bikeobj.addPart(Part('sturdy frame'));
    bikeobj.addPart(Part('bigger wheels'));
    bikeobj.addPart(Part('comfy seat'));
end
```

如果我们再构造另一种新的自行车，比如公路自行车，只需要把 constructMountainBike 方法稍加修改即可。观察这些 Construct 方法容易发现自行车的构造顺序是相对稳定的，都是先框架，再轮子，再坐垫，只是建造时配件各自不同。Builder 模式正好适用于描述这样的过程。该模式用来构造一个复杂对象，在这里是构造一辆自行车。该模式把构成过程和构造的具体对象分离开来，让相同的构造过程创建出不同的产品。

Builder 模式中包含以下几个重要的类：

首先，需要有一个类来指导对象的构建过程，这个类通常叫做 Director（指导者），由它来控制构建顺序。在构造自行车的例子中，我们给这个类起名字叫做 BikeTechnician。其次，还需要有具体的 Builder（建筑者）类，用来制造不同的产品（Bike），这些类通常叫做 ConcreteBuilder。比如，构造山地自行车的类叫做 MountainBikerBuilder，构造公路自行车的类叫做 RoadBikeBuilder，这些类大致相似，但细节和具体的产品相关。而且我们还可以把它们的共性抽象出来，引入一个 BikeBuilder 的抽象类。综上，整体的 UML 如图15.18所示。

接着再讨论一下各个类的从属关系：BikeTechnician 类的对象将拥有 Builder 对象，也就是被其指挥的对象。Builder 所产生的产品是具体的自行车，Builder 类拥有产生的 Bike 对象。

该类的 MATLAB 代码简单实现如下（Bike 和 BikePart 仍是组合关系）：

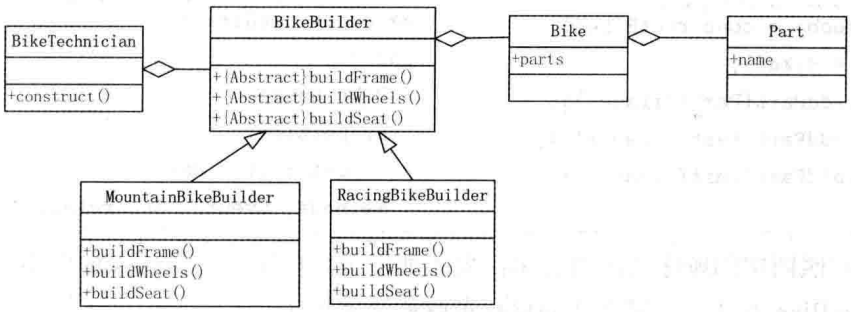


图 15.18 使用 Builder 模式构造自行车

Bike

```
classdef Bike < handle
   properties
        parts
   end
   methods
       function addPart(obj,part)
           obj.parts =[part,obj.parts];
       end
   end
end
```

BikePart

```
classdef Part
    properties
       name
    end
    methods
        function obj = Part(name)
            obj.name = name;
        end
    end
end
```

BikeTechnician 类负责指导自行车的构造。注意，该类对象拥有 Builder 对象。

BikeTechnician

```
classdef BikeTechnician < handle
   properties
      builder                              %  该 Builder 对象可以被更换
   end
   methods
       function set.builder(obj,builder)  %  该 setter 指定 builder 对象
            if(isa(builder,'BikeBuilder'))
               obj.builder = builder ;
            else
                error('input must be an instance of BikBuilder')
            end
       end
       function construct(obj)            %  固定了 Build 的顺序
           obj.builder.buildFrame();
           obj.builder.buildWheels();
           obj.builder.buildSeat();
       end
   end
end
```

BikeBuilder 是一个抽象类，其中包含有如下的抽象接口方法：buildFrame，buildWheels，buildSeats。这些方法的具体实现将留给 BikeBuilder 的子类去完成。BikeBuilder 类中还有一个 showBike 方法，该方法用来把 product 中对象的内容输出到命令行，用来简单地验证 product 中的内容。

BikeBuilder

```
classdef BikeBuilder < handle
    properties
       product = Bike()
    end
    methods(Abstract)
        buildFrame(obj);
        buildWheels(obj);
        buildSeat(obj);
    end
    methods
        function showBike(obj)
           disp(obj.product.parts.name)   % 输出名字到 command line
        end
    end
end
```

BikeBuilder 的子类分别是 MountainBikeBuilder 和 RoadBikeBuilder，它们实现各个具体配件的构造。为简单起见，仅用 name 这个属性的不同来区分各配件的不同。

MountainBikeBuilder

```
classdef MountainBikeBuilder< BikeBuilder
    methods
        function buildFrame(obj)
           obj.product.addPart(Part('sturdy frame'))
        end
        function buildWheels(obj)
           obj.product.addPart(Part('wide wheels'))
        end
        function buildSeat(obj)
           obj.product.addPart(Part('comfy seat'))
        end
    end
end
```

RoadBikeBuilder

```
classdef RoadBikeBuilder< BikeBuilder
    methods
        function buildFrame(obj)
           obj.product.addPart(Part('light frame'))
        end
```

```
        function buildWheels(obj)
            obj.product.addPart(Part('thin wheels'))
        end
        function buildSeat(obj)
            obj.product.addPart(Part('thin seat'))
        end
    end
end
```

下面进行测试：先声明一个 BikeTechnician 对象，再给其指定一个 Builder 是 Montain Bike Builder 对象，对 obj 发出 construct 的请求，一个 Bike 对象就被构造了出来。

用来测试的脚本以及命令行输出如下：

Script

```
obj = BikeTechnician();
obj.builder= MountainBikeBuilder();
obj.construct()
obj.builder.showBike()
```

Command Line

```
ans =          % 验证这些是山地车配件
comfy seat
ans =
wide wheels
ans =
sturdy frame
```

BikeTechnician 对象所拥有的 Builder 对象是可以替换的。下面把这个 Builder 替换成 RoadBikeBuilder，再次调用 construct，就可以生产公路车了。

Script

```
obj = BikeTechnician();
obj.builder = RoadBikeBuilder();
obj.construct()
obj.builder.showBike()
```

Command Line

```
ans =          % 验证这些是公路车配件
thin seat
ans =
thin wheels
ans =
thin frame
```

15.3.2 应用：Builder 模式为大规模计算做准备工作

任何复杂产品的构造，只要顺序固定，都可以考虑使用 Builder 模式。举一个实际计算的例子。比如，我们要对一个算法做一系列的大运算量的测试，首先要构造提供给算法的一系列的输入文件，如果计算量很大，还需要专门为这个系列的测试构造特殊的文件夹用来存放计算结果，因为通常这样的运算是在集群上完成的，所以创建临时文件夹也是必要的一部分。如果算法还依赖一些其他数据文件，程序还要负责往这些临时文件夹中拷贝计算所必需的文件，计算结束之后还要有一个程序负责到这些临时文件夹中收集计算结果，并且做必要的清理工作。上述这个计算过程，每步要做的工作都是固定的，也可以将其看做是产生一个复杂的产品。可以这样设计程序（如图15.19）来指导整个过程。

JobBuilderClass 中有三个抽象方法待具体的子类去实现，不同的子类实现代表对算法的一种大运算量的测试。ConcreteJob 子类中需要实现三个方法：GenDirectory 针对每次测试需要建立的临时文件夹，不同子类位置可能不同，数量也可能不同；CopyCommonFiles 针对

每次测试准备的不同的数据文件，也要放到具体的子类中；GenInput 针对每次测试使用的不同的运算参数。CreateJobs 方法则类似于之前的 construt 方法，按固定的顺序调用 GenDir，CopyFile 和 GenInput。

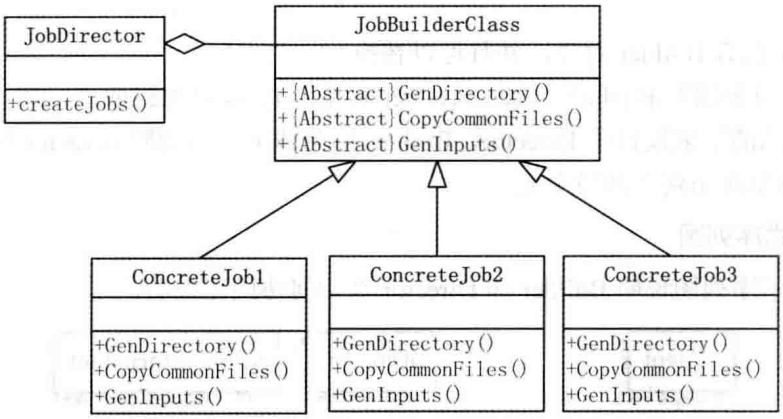


图 15.19 Builder 模式为大规模计算做准备工作

15.3.3 Builder 模式总结

《设计模式》一书中对 Builder 模式的意图是这样叙述的：

> 将一个复杂对象的构建与它的表示方法分离，使得同样的构建过程可以创建不同的表示。
>
> Design Patterns GOF

1. Builder 模式结构

Builder 模式的结构如图15.20所示，Director 拥有 Builder，Builder 拥有 Product，Product 由 Parts 组成。

在 Builder 模式中，Builder 接口提供给了 Director 一个构造产品的抽象接口。该接口使得 Director 可以隐藏具体产品的表示和内部构造，同时对 Client 也隐藏了该产品是如何装配的。如果要产生新的产品，或者改进已有的产品，只需要定义一个新的 ConcreteBuilder 即可。

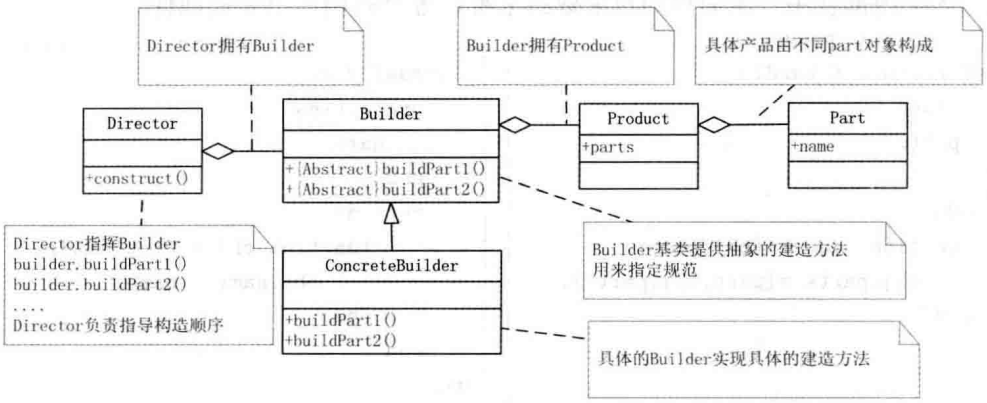


图 15.20 Builder 模式的结构

2. 类之间的协作

Builder 模式机构中各个类之间的关系如下：

- □ Client（外部程序）负责构造 Director 对象，并且设定该 Director 所要指导的具体的 Builder。
- □ Director 拥有 Builder 对象，并且可以替换。
- □ Builder 对象拥有 Product 对象，Product 对象由 Parts 对象组成。
- □ 构造产品的请求发自于 Director，Builder 接到请求之后按照 Director 所指导的顺序把 Parts 对象添加到产品当中去。

3. Builder 模式序列图

图15.21所示序列图说明 Builder 和 Director 是如何协同工作的。

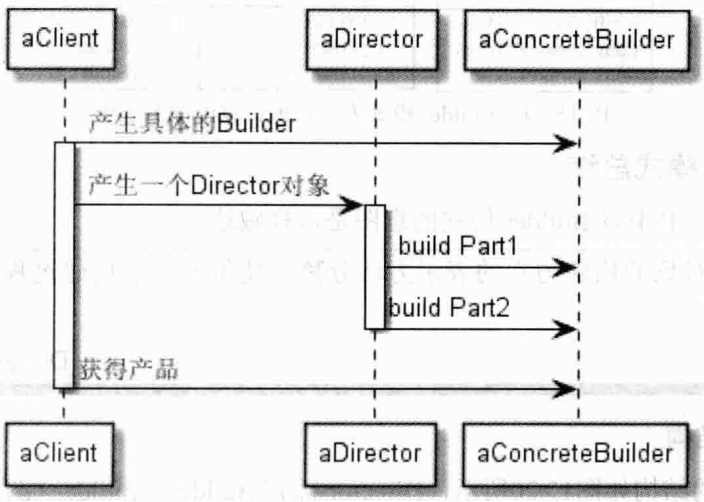


图 15.21 Client 控制 Director，Director 指导 Builder

4. Builder 模式框架的 MATLAB 实现

下面给出 Builder 模式框架的 MATLAB 实现。该实现以简洁为目的，仅用来说明类之间的重要的协作关系，并不一定是最完整的实现，其中 Product 泛指被构造的复杂对象。为简单起见，该类内部使用一个简单的对象数组来维护各个被构造出来的部件[①]。

Product

```
classdef Product < handle
   properties
        parts
   end
   methods
       function addPart(obj,part)
           obj.parts =[part,obj.parts];
       end
   end
end
```

Part

```
classdef Part
    properties
       name
    end
    methods
        function obj = Part(name)
            obj.name = name;
        end
    end
end
```

①更复杂的对象数组的使用请参见第11章。

创建的过程被固化在 Director 类的 construct 方法中，但是产生的具体产品对象的表示方法却可能不同。

Director

```
classdef Director < handle
   properties
      builder
   end
   methods
       function set.builder(obj,builder)
            if(isa(builder,...
                   'BikeBuilder'))
               obj.builder = builder ;
            else
                error('wrong input');
            end
       end
       function construct(obj)
           obj.builder.buildPart1();
           obj.builder.buildPart2();
       end
   end
end
```

Builder

```
classdef Builder < handle
    properties
       product = Product()
    end
    methods(Abstract)
        buildPart1(obj);
        buildPart2(obj);
    end
    methods
        function showProduct(obj)
           for iter = 1:length(obj)
            obj(iter).product.parts.name
           end
        end
    end



end
```

每个 ConcreteBuilder 内部都包含了创建一个特定产品的代码，不同的 Director 将调用这些代码来构造不同的 Product。

ConcreteBuilder

```
classdef ConcreteBuilder< Builder
    methods
        function buildPart1(obj)
           obj.product.addPart(Part('I am part 1'))
        end
        function buildPart2(obj)
           obj.product.addPart(Part('I am part 2'))
        end

    end
end
```

5. 何时使用 Builder 模式

以下情况可以考虑使用 Builder 模式：

- □ 当构造过程中，允许被构造的对象有不同的表示时。
- □ 当创建复杂对象的算法，要独立于该对象的装配方式时。

第 16 章　构造型模式

16.1　装饰者模式：动态地给对象添加额外的职责

16.1.1　装饰者模式的引入

在第15.1节 Factory 模式中，我们介绍了利用 Factory 和 AbstractFactory 模式解决各种面条和配料的生产问题。这一节接着扩展这个面馆的程序，设计一个定价系统，以计算面条和配料的定价。假设表16.1所列的菜单是面馆能够提供的各种配料及其价格，该计价程序要求根据顾客所点的内容，自动计算出价格。

表 16.1　配料及其价格表

面条种类	价格	肉类	价格	酱料	价格	其他配料	价格
挂面	2.00	牛腩	3.00	辣酱	0.50	煎蛋	1.00
拉面	3.00	猪排	2.00			香菇	1.00
炒面	2.00	鸡丁	2.50				
⋮							

最容易想到的做法是，给每一种可能搭配，声明一个与其一一对应的类，和第15.1节的处理方式相同，面条作为主食，顾客每点一份只能点挂面、拉面或者炒面中的一种，于是首先可以抽象出一个面条基类来，如图16.1所示。

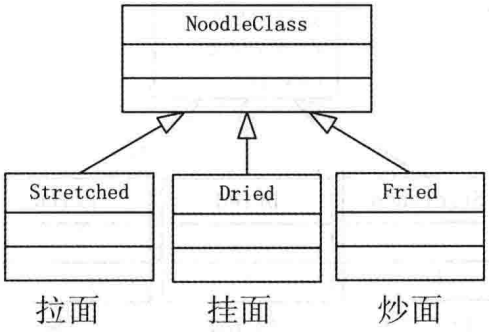


图 16.1　面条基类和子类

接下来要解决的问题是，把肉类酱料和配料也加到这个面向对象的设计中来。最直接的办法是有多少种搭配就定义多少种类，并且每个类都有一个 cost 方法，用来计算这碗面中所含原料的价格，如图 16.2所示。

这种设计又回到使用继承解决一切问题的老套路上了：随着菜单的丰富，类的数量也会增加得很快。这种设计方法不利于扩展：因为每增加一种新的配料，或者肉类，都会成倍地增加新的可能的搭配，所以成倍地增加新的类。而且这种设计方法还无法应对修改，因为如果有一种配料或者肉类的价格发生变化，就会有许多的类中的 cost 方法须要修改。

针对上述设计的缺点，可能的改进是：把肉类、酱料和配料都当做 Noodle 基类的属性，整合到 Noodle 基类中去，这样每种配料和酱料都对应 NoodleClass 中的一个属性。比如 beef

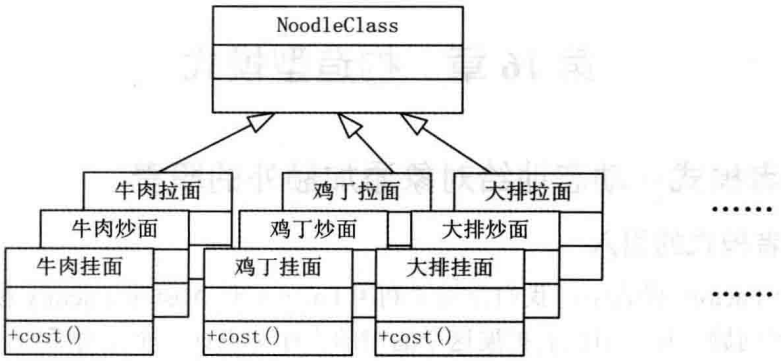


图 16.2 尝试设计 1：把每种可能的搭配都设计成一个类

属性，默认值为 false，如值为 true 就表示顾客点的菜中有牛肉；每个属性还有 set 和 get 方法，用来赋值和查询。基类中的 cost 方法用来计算所有配料、酱料加到一起的价格。具体面条类中也有一个 cost 方法，用来计算面条本身的价格，扩展了基类的方法。类的设计如图16.3所示。

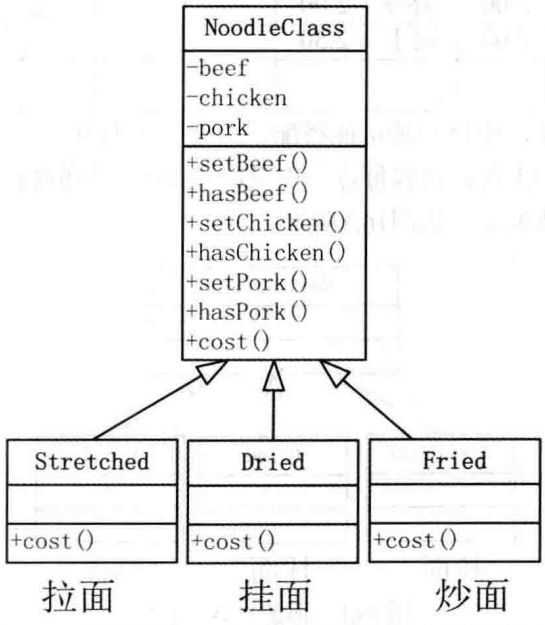


图 16.3 尝试设计 2：把所有可能的变化都放到基类中

这样的设计，类的总体数量确实是减少了，但仍然不够灵活：每增加一个新的配料或者肉类品种，都需要给 Noodle 基类添加一个新属性相应的 get 和 set 方法，还要修改 cost 方法。而且每当一种配料的价格发生变化时，基类的 cost 方法就需要加以修改。如果顾客点一份牛肉面，但要求再多加一份牛肉，还需要修改 Noodle 基类，使其能够记录牛肉的分量，并且在 cost 方法中用单价乘以数量。

我们期望的设计是让类容易扩展，在不修改现有代码的情况下，就可以搭配新的行为。如果能达到这样的目标，这样的设计就是有弹性的。这一节我们借这样一个计价例子来介绍一种设计模式：装饰者模式（Decorator Pattern）。基本的设计思想如图16.4和图16.5所示，我们以面条的种类为主体，然后运用调料或者配料来“装饰”（Decorate）面条。比如，客人如

果想要麻辣牛肉卤蛋拉面，那么我们要做的是：

- 先产生拉面对象。
- 用牛肉装饰拉面，成为牛肉拉面。
- 用卤蛋装饰牛肉拉面，成为牛肉卤蛋拉面。
- 用辣酱装饰牛肉卤蛋拉面，成为麻辣牛肉卤蛋拉面。

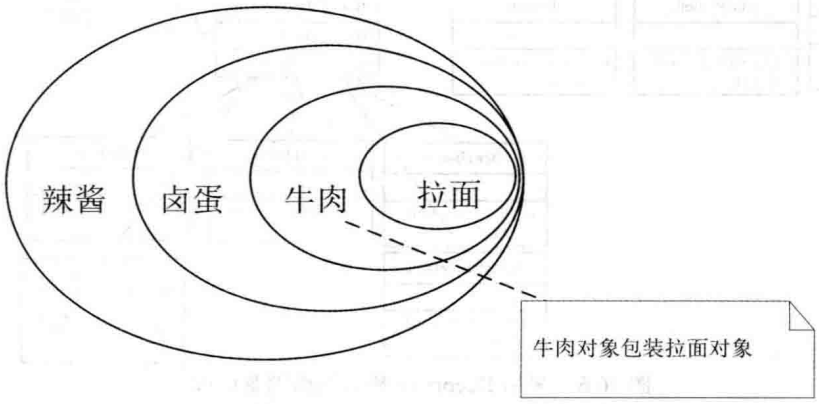


图 16.4 拉面对象被层层装饰

在计算价格时，通过调用最外层的装饰者的 cost 方法，触发一连串的对各个对象的 cost 方法的调用，最终返回总的价格，如图16.5所示。

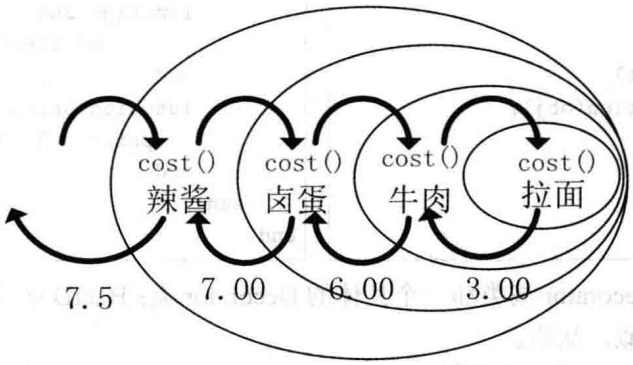


图 16.5 cost 方法将被层层调用

下面介绍如何具体实现这样的结构。

16.1.2 面馆菜单代码

我们先直接给出该菜单定价系统的 UML（见图16.6）。

下面是 Noodle 基类和一个具体的 Noodle 类 Stretched 的定义。注意：该 Stretched 类中包含了拉面本身的价格，其余的两个 Fried 和 Dried Noodle 类与 Stetched 类相似，这里从略。

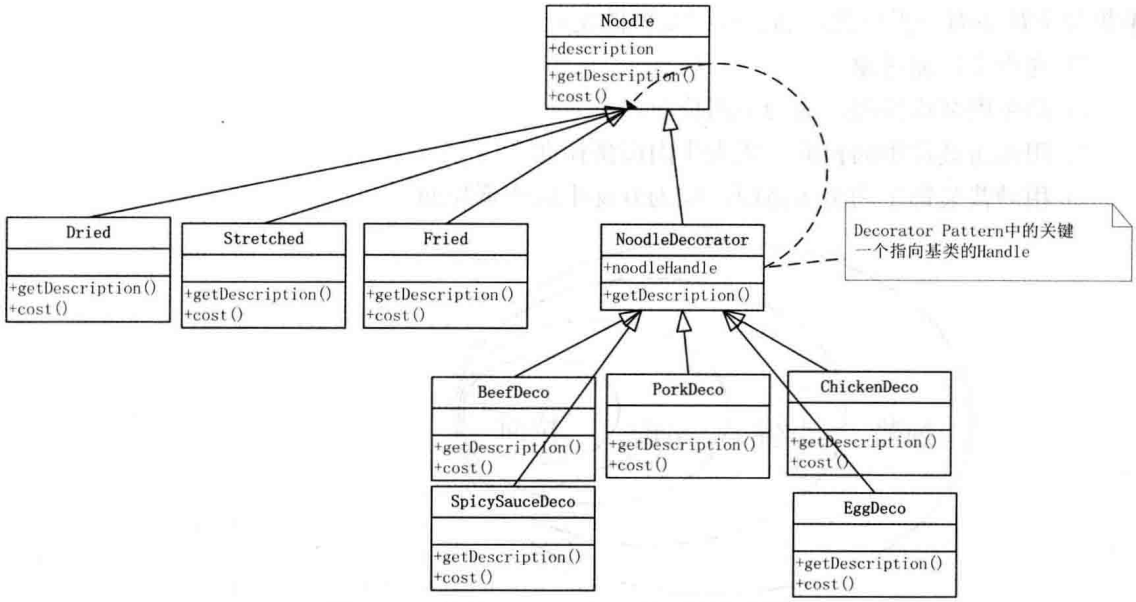


图 16.6 采用 Decorator 模式的面馆菜单程序

Noodle 面条基类

```
classdef Noodle < handle
    properties
        description
    end
    methods(Abstract)
        showDescription(obj);
        cost(obj);
    end
end
```

Stretched 拉面类

```
classdef Stretched < Noodle
    methods
        function showDescription(obj)
            disp('Stretched');
        end
        function price = cost(obj)
            price = 3.00 ;  % 拉面的价格
        end
    end
end
```

下面是 NoodleDecorator 基类和一个具体的 Decorator 类：BeefDeco 类，其他的 PorkDeco，ChickenDeco 等类相似，从略。

NoodleDecorator 装饰者基类

```
1  classdef NoodleDecorator < Noodle
2      properties
3          noodleHandle % 指向被装饰对象
4      end
5      methods
6       function obj = NoodleDecorator(hin)
7          obj.noodleHandle = hin;
8       end
9       function getDescription(obj)
10         % 基类 getDescription 中不需要内容
11      end
12     end
13 end
14
```

BeefDeco 装饰者具体类

```
1  classdef BeefDeco < NoodleDecorator
2  methods
3    function obj = BeefDeco(hin)
4     obj = obj@NoodleDecorator(hin);
5    end
6    function price = cost(obj)
7     price = obj.noodleHandle.cost()+3.00;
8    end
9    function getDescription(obj)
10     obj.noodleHandle.getDescription();
11     disp('Beef');
12    end
13 end
14 end
```

可以这样使用这段代码来计算最后的价格：

```
Script                                          Command Line
1 noodle  = Stretched();
2 noodle  = BeefDeco(noodle);        装饰
3 noodle  = EggDeco(noodle);         装饰
4 noodle  = SpicySauceDeco(noodle); 装饰
5 disp(noodle.cost())                              7.5000
6 noodle.getDescription()          Stretched Beef Egg SpicySauce
```

从对象的角度来看，每次装饰就是用新的装饰者指向被装饰的对象，如图16.7所示。

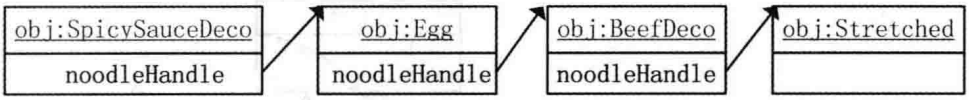


图 16.7 每个装饰者都有一个 Handle 指向被装饰的对象，这是一个对象链

说明：

- Script 的第 1 行，产生了一个具体的拉面对象。
- Script 的第 2 行，括号中的 noodle 是一个拉面对象，BeefDeco 是一个装饰者，每个装饰者对象中都有一个 noodleHandle 属性（继承自 NoodleDecorator 基类，见 NoodleDecorator 的第 3 行）。

  该 noodleHandle 将指向传进来的 noodle 对象（见 NoodleDecorator 的第 7 行），然后返回自己的 Handle。我们可以把这样的行为形象地理解成装饰或者包裹。
- Script 的第 3 行，这里括号中的 noodle 其实是 BeefDeco 对象，该 BeefDeco 对象已经装饰了 Stretched 对象，现在 BeefDeco 对象又被 EggDeco 对象所装饰。
- Script 的第 5 行，将首先触发 SpicySauceDeco 类的 cost 方法，该 cost 方法和 BeefDeco 的第 7 行类似，把自身的价格计入总价中，并且向下递归调用 cost 方法，调用被自己装饰的对象的 cost 方法，继续累计价格。
- Script 的第 6 行，getDescription 方法的功能是显示这碗面的类的所有内容（主食和装饰者），其工作原理和 cost 方法相似。

容易看出，这种设计对修改是封闭的，假如要修改牛肉的价格，只需要修改 BeefDeco 类即可，程序的其他部分不会受到影响。这种设计还支持多次装饰，比如顾客需要在面中加入两份牛肉，只需要这样：

```
Script
>> noodle = BeefDeco(noodle);
>> noodle = BeefDeco(noodle);
```

这种设计对扩展是开放的，若要添加新的配料，只需要添加新的 NoodleDecorator 子类即可。

16.1.3 装饰者模式总结

《设计模式》一书中对装饰者模式的意图是这样叙述的：

> 动态地给一个对象添加一些额外的职责，就增加功能来说，Decorator 模式相比生成子类更为灵活。
>
> Design Patterns GOF

Decorator 模式也叫做包装器（Wrapper）。

1. 装饰者模式结构

图16.8所示是装饰者模式的 Class Diagram 图。

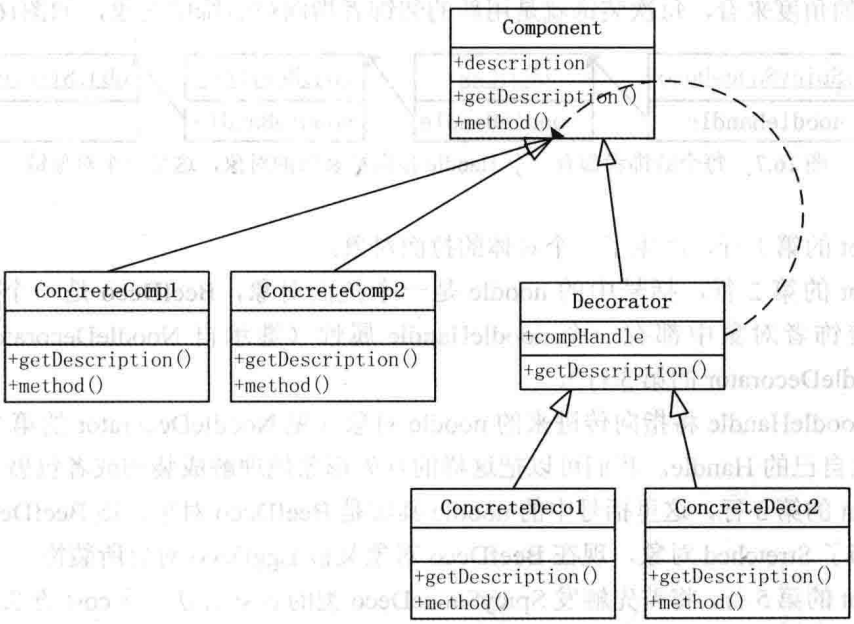


图 16.8 Decorate 维持一个指向 Component 对象的 Handle，并定义一个与 Component 一致的接口

2. 类之间的协作

装饰者模式的关键是：ConcreteComponent 和 ConcreteDeco 都有相同的基类，并且 Decorator 基类中的 compHandle 属性被用来指向 Component 对象，这使得一个 Componet 被装饰了之后，从外界看上去它仍然像是一个 Component 对象，并且可以继续被装饰下去。

3. 何时使用 Decorator 模式

《设计模式解析》一书中，对 Decorator 模式的适用场合是这样描述的：Decorator 模式可以避免通过创建子类来扩展类的功能，Decorator 是以动态的方式给单个对象添加新的功能。当想要扩展类，而又想避免子类数量爆炸时，可以考虑使用 Decorator 模式。

第17章 行为模式

17.1 观察者模式：用MATLAB实现观察者模式

17.1.1 发布和订阅的基本模型

在第4章中，我们介绍了用event（事件）机制和addlistener和notify两个类方法在对象之间传递信息。在这节中，我们抛开Handle基类中这些现成的定义和方法讨论如何用MATLAB来实现类似功能。具有这些基本功能的模式叫做ObserverPattern。这样的练习有助于更好地理解事件和响应机制。

假设我们要用MATLAB模拟这样一个问题：有一个网站，它的内容不定期地更新，浏览者希望每次网站更新时都会收到该网站的提醒——这意味着该网站要提供订阅功能；如果浏览者希望不再收到提醒，网站还要支持取消订阅。这个问题涉及网站和浏览者，在设计类的初步，通常的做法是首先把实体抽象成类。于是，我们先定义出两个类，分别是Website类和Subscriber类，如图17.1所示。

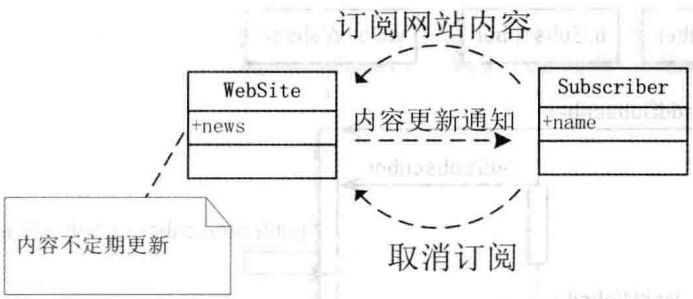


图17.1 观察者模式：网站发布和读者订阅的初步设计

图17.1中的虚线分别代表了网站的功能和订阅者的行为，反映到程序中，它们将对应的是Website类和Subscriber类的方法。现在讨论如何实现订阅、通知和取消订阅：

- Website类如果要支持Subscriber订阅和取消的行为，就要维护一个列表用来记录订阅网站内容的对象。具体到程序上，这个列表可以是一个对象数组或者对象元胞数组，或者是Heterogeneous对象数组[①]。
- 订阅和取消的功能对应的方法是addSubscriber和removeSubscriber。订阅者对象订阅网站，就是调用addSubscriber方法往对象数组中加一个对象；订阅者对象取消订阅，就是调用removeSubscriber方法在数组中找到该对象并且删除掉。
- 通知订阅者也对应一个Website类的方法，把这个方法命名为notifySubscriber：每当网站内容更新，就调用该方法通知订阅者。
- notifySubscriber方法必须能够触发订阅者对象的实际动作[②]，这里假设这个动作是登录网站并且浏览，给这个方法也起一个名字，叫做visitWebsite方法。

①对象数组的介绍见第11章。
②可以理解成回调函数。

综上，类中的属性和方法添加到 UML 中如图17.2所示。

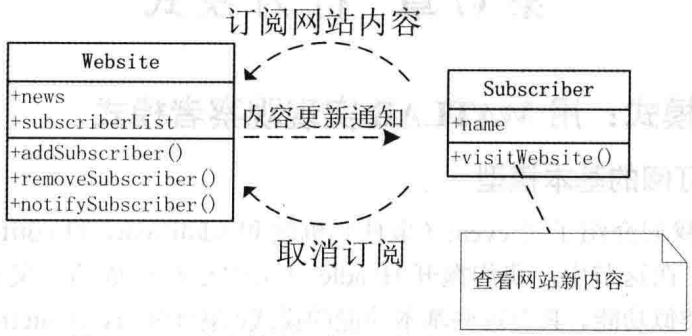


图 17.2 观察者模式：发布订阅模型 Class Diagram

还可以用序列图（Sequence Diagram）来描述订阅和通知的过程，如图17.3所示。该 Sequence Diagram 中有两个订阅者对象，分别是 a 和 b，它们先后通过调用 addSubscriber 方法，向 Website 对象提出订阅的请求（request），webSite 对象于是将 a 和 b 对象在内部登记下来，一段时间后，网站的内容发生更新，网站将调用 notifySubscriber 函数来通知内部登记的两个订阅者，随后将触发订阅者的行为，即 visitWebsite。

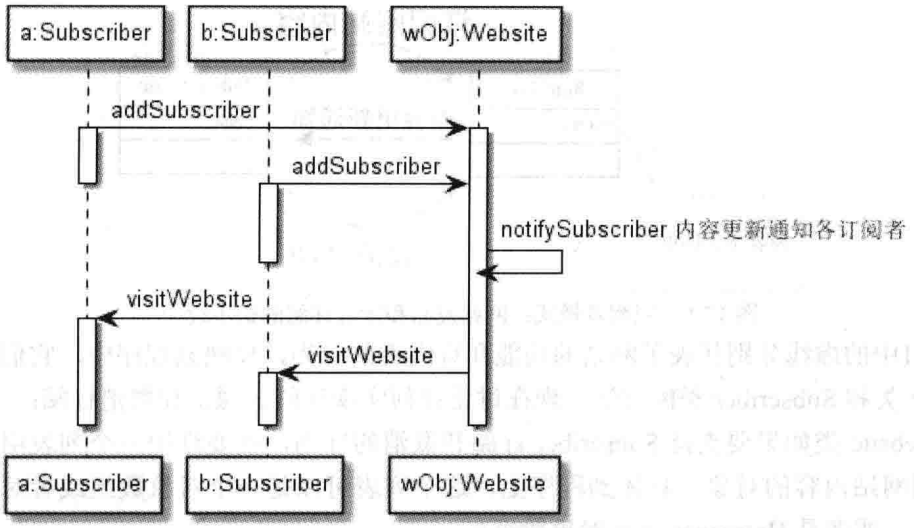


图 17.3 发布订阅模型的 Sequence Diagram

根据 UML（图17.2）和序列图（图17.3），网站和订阅者直接发布和订阅的初步模型可以实现为 Website 类和 Subscriber 类：

Website 类：

Website.m

```
classdef Website < handle
    properties
        subscriberList = [];                                % 订阅者列表
    end
    methods
        function addSubscriber(obj,sObj)                    % 添加订阅者
```

```
            obj.subscriberList = [obj.subscriberList,sObj];
        end

        function removeSubscriber(obj,name)                    % 删除订阅者
            index = ~strcmp(obj.subscriberList.name,name);
            obj.subscriberList = obj.subscriberList(index);
        end
        function notifySubscriber(obj)              % 触发订阅者的 visitWebsite
            for iter = 1:length(obj.subscriberList)            % 遍历订阅者
                obj.subscriberList(iter).visitWebsite();  % 调用回调函数
            end
        end
    end
end
```

Subscriber 类：

Subscriber.m

```
classdef Subscriber< handle
    properties
        name   % 假设 Subscriber 只有一个属性
    end
    methods
        function obj = Subscriber(name)
            obj.name = name;
        end
        function visitWebsite(obj)
            % 验证收到通知
           disp([obj.name,' notified,will visit the website']);
        end
    end
end
```

用来测试的脚本以及命令行输出如下：

Script

```
wObj = Website();
wObj.addSubscriber(Subscriber('a')); % 注册
wObj.addSubscriber(Subscriber('b'));
wObj.addSubscriber(Subscriber('c'));
wObj.notifySubscriber();                % 通知
wObj.removeSubscriber('a');             % 注销
wObj.removeSubscriber('c');
wObj.notifySubscriber();                % 通知
```

Command Line

```
a notified,will visit the website
b notified,will visit the website
c notified,will visit the website
b notified,will visit the website
```

17.1.2 订阅者查询发布者的状态

上节的实现中，visitWebsite仅是一个空的函数，现在新的要求是：允许订阅者查询网站的内容，比如访问网站的内部属性news，如图17.4所示。在订阅者被通知后，执行visitWebsite时，要能够访问网站对象，获得news属性。为了实现subscriber对象查询Website的内部状态的功能：

- 首先news应该是private属性，因为news是网站类中内部的数据，出于封装的目的，该属性不应该直接暴露给外部的类，所以Website类要提供一个getNews的public方法，用来提供访问news属性的公共渠道。
- 订阅者必须知道该去哪里访问news属性，所以订阅者必须有website对象的Handle。这就要求在网站类的notifySubscriber方法中，网站对象自己必须也作为一个参数传递给订阅对象。

经过改进的notifySubscriber和visitWebsite类的UML如图17.4所示。

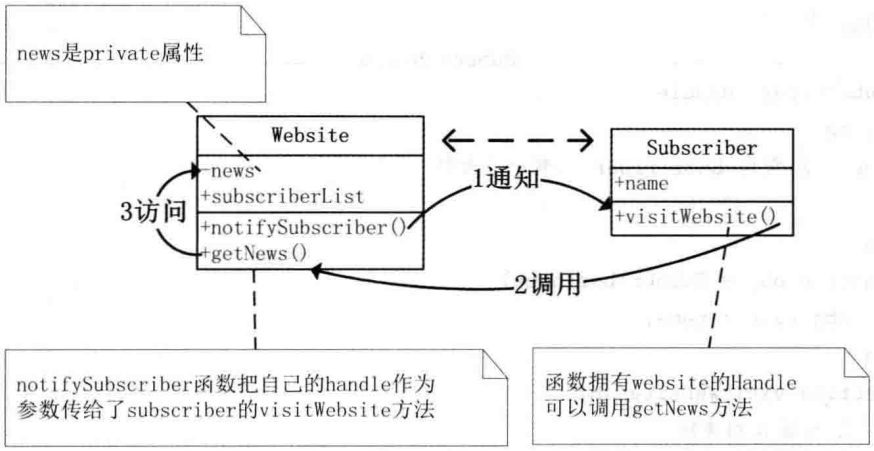


图 17.4 订阅发布模型：访问发布者中的数据

Subscriber订阅网站内容，网站通知内容更新，订阅者响应访问网站内容流程如图17.5所示。

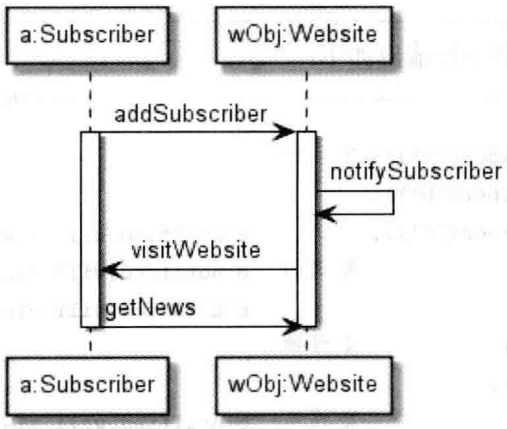


图 17.5 Subscribe者访问Website中的内容

以下是改进的Website和Subscriber类：

Website.m

```
classdef Website < handle
    properties(SetAccess = private )
        subscriberList = [];
        news = 'Headline ......';                          % 要访问的内容
    end
    methods
        ...... % 其他方法保持不变
        function notifySubscriber(obj)
            for iter = 1:length(obj.subscriberList)
                obj.subscriberList(iter).visitWebsite(obj);
            end
        end
        function news = get.news(obj)                      % PUBLIC 方法提供访问接口
            news = obj.news;
        end
    end
end
```

Subscriber.m

```
classdef Subscriber< handle
   %..... 其余保持不变
        function visitWebsite(objp,websiteRef)
           disp([obj.name,' notified,will visit the website']);
           disp(['news= ',websiteRef.getNews()]);          % 调用公用方法
        end
end
```

用来测试的脚本以及命令行输出如下：

Script

```
wObj = Website();
wObj.addSubscriber(Subscriber('a'));
wObj.notifySubscriber();
```

Command Line

```
a notified,will visit the website
news= Headline          % 得到内部状态
```

17.1.3 把发布者和订阅者抽象出来

可以预料，这种订阅和发布的关系应用范围很广，还可以用这种模式去形容其他的类之间的关系。比如，这种模式还可以用来形容猎头公司和求职者之间的关系。如图17.6所示。求职者在猎头公司注册，猎头公司有了新的职位消息，就通知登记的求职者，求职者收到通知，查询新职位信息，并且递简历。从代码复用的角度，应该把 addListener，removeListerner 和 notifySubscriber 这些基本的方法抽象出来，放到一个基类当中去。发布者的基类叫做 Publisher，订阅者实际上是监听的角色，基类叫做 Observer。所以，发布订阅模式也叫做观

察者模式（Observer Pattern）。

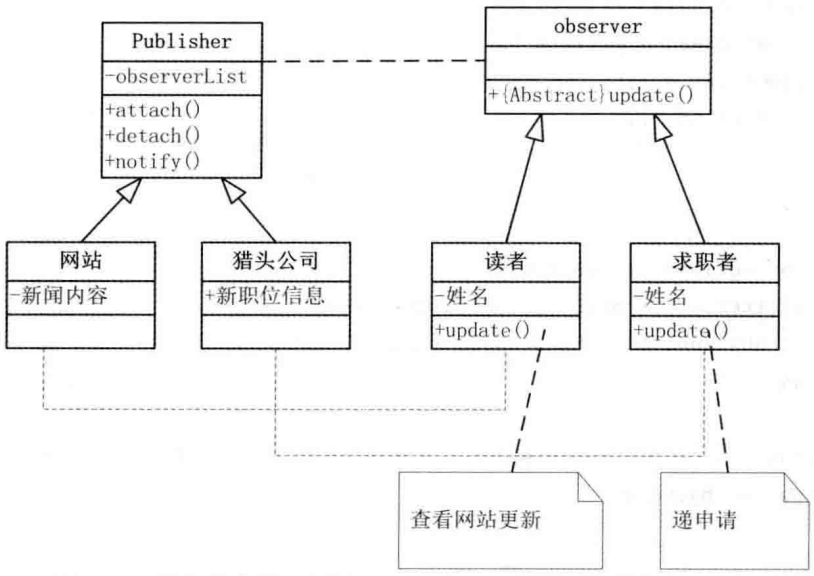


图 17.6 抽象出来的 Publisher 和 Observer 基类可以用于其他的类

不同的模型中，订阅者被通知之后，响应的行为显然是不同的。比如，读者响应的行为是：登录网站，查看更新。求职者响应的行为是：查看职位信息，递简历。为了统一接口，规定虽然这些响应的行为各有不同，但是它们的名字都必须叫做 update。该方法可以放到基类中去，并定义成 Abstract 方法，它用来提供一个规范，规定其子类必须提供具体的实现，这个方法可以有不同的实现（Implementation），这代表具体对象有具体的行为，但这个方法的名字一定要叫做 update。

17.1.4 Observer 模式总结

《设计模式》一书中对 Observer 模式的意图是这样叙述的：

> 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知，并且自动被更新。
>
> Design Patterns GOF

1. Observer 模式结构

Observer Pattern 的 UML 如图 17.7所示。

2. 类之间的协作

- 首先具体的观察者向 ConcreteSubject 对象发出订阅的请求，该 ConcreteSubject 对象把具体观察者加到其内部的一个列表中。
- 当 ConcreteSubject 内部发生改变，需要通知其观察者时，它将遍历其内部的观察者的列表，依次调用这些观察者的 update 方法。
- 观察者得到更新的通知，它可以向 ConcreteSubject 对象提出查询的要求。

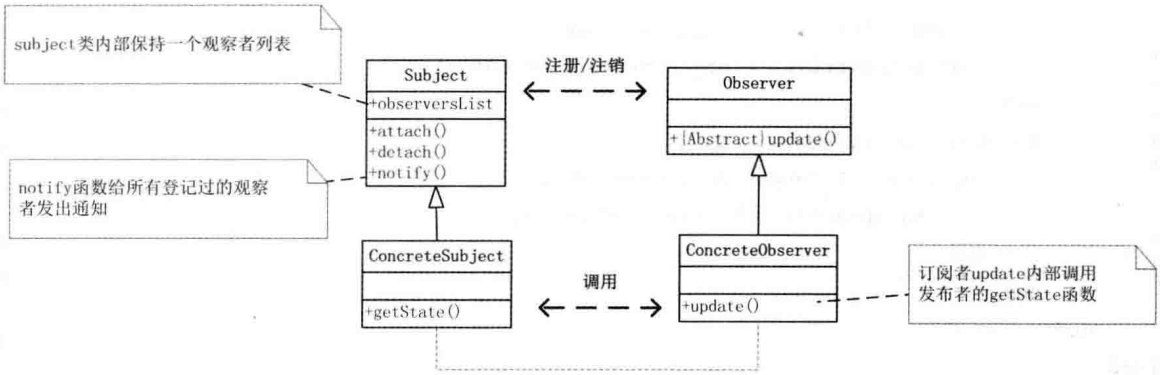


图 17.7 Observer Pattern UML

3. Observer 模式序列图

如图 17.8所示，发生的一系列动作说明了目标对象和观察者之间的协作。

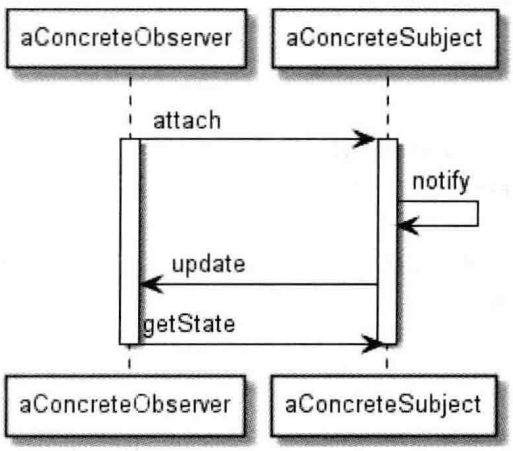


图 17.8 Sequence Diagram 说明了目标对象和观察者之间的协作

4. Observer Pattern 框架的 MATLAB 实现

下面给出 Observer Pattern 框架的 MATLAB 实现。该实现以简捷为目的，仅用来说明类之间的重要的协作关系，并不一定是最完整的实现。首先是发布者的基类，提供基本的添加、注销和通知功能：

Subject.m

```
classdef Subject < handle
    properties(Access = private )
        observerList = [];                                    % 内部登记表
    end
    methods
        function attach(obj,observer)
            obj.observerList = [obj.observerList,observer];   % 添加
        end

        function detach(obj,observer)
```

```
            index = [obj.observerList] ~= observer;          % 查找
            obj.observerList = obj.observerList(index);      % 删除
        end
        function notify(obj)
            for iter = 1:length(obj.observerList)
                obj.observerList(iter).update(obj);          % 通知
            end
        end
    end
end
```

下面是发布者的具体类，包含一个私有属性，以及公有的 set 和 get 的接口：

ConcreteSubject.m

```
classdef ConcreteSubject < Subject
    properties(Access = private)
        state                                                % 内部状态
    end
    methods
        function state = getState(obj)
            state = obj.state;
        end
        function setState(obj,val)                           % 公共接口
            obj.state = val ;
        end
    end
end
```

订阅者基类是一个抽象类，规定 update 方法的接口：

Observer.m

```
classdef Observer < handle
    methods(Abstract)
        update(obj,subject);
    end
end
```

订阅者的具体类，继承自抽象基类 Observer，其中包括 update 方法的具体实现：

ConcreteObserver.m

```
classdef ConcreteObserver < Observer
    properties
        name
    end
    methods
        function obj = ConcreteObserver(name)
            obj.name = name ;
        end
```

```
        function update(obj,subject)                    % 回调方法
            disp([obj.name,' notifyed! subject state = ',subject.getState]);
        end                                              % 访问 subject 内部状态
    end
end
```

用来测试的脚本以及命令行输出如下：

```
Script                                    Command Line
subObj = ConcreteSubject(); 构造发布者
subObj.setState('smoking'); 构造订阅者
a = ConcreteObserver('a');
b = ConcreteObserver('b');

subObj.attach(a);                  注册
subObj.attach(b);                  注册
subObj.notify()                    通知      a notifyed! subject state = smoking
                                             b notifyed! subject state = smoking
subObj.detach(a);                取消订阅
subObj.notify()                    通知      b notifyed! subject state = smoking
```

17.2 策略模式：分离图像数据和图像处理算法

17.2.1 问题的提出

假设小李是一个研究生，他的毕业设计项目是开发一个新的图像去模糊的算法。因为已有的图像去模糊的算法有很多，在研究最开始，小李需要把现有的各种图像去模糊的算法都试一下，再决定以哪一个具体的算法作为突破口加以改进。现在小李从最简单的脚本写起。

首先打开一个图像文件作为算法的输入：

```
I = imread('someImage.jpg');
```

比如有个算法叫做 deblurMethod，在脚本中调用这个函数，得到返回 J 结果，并且作图：

```
J = deblurMethod(I);
imshow(J);
```

接下来，小李再用同样的方式去调用其他的算法，依此类推。这个例子可以泛化到很一般的情况：给定数据，在不同的情况下使用不同的算法，把计算程序组织起来。

上述的例子是面向过程的，下面使用面向对象的方式对其开始进行改造。首先把数据（即图像）和对数据的处理方法（即图像处理方法）封装到一个类中（如下面左边的代码 ImageClass 所示），在右边的代码框中，原先函数对数据的操作变成了：先定义一个对象，再调用对象的成员方法 deblur。

ImageClass

```
classdef ImageClass < handle
  properties
        image
  end
  methods
   function obj = ImageClass(filename)
     obj.image = imread(filename) ;
   end
   img = deblur(obj); % 具体算法从略
  end
end
```

Command Line

```
obj = ImageClass('someImage.jpg');
img = obj.deblur();
imshow(j);
```

在上述的改造中，小李用类封装了数据和算法：deblur 函数变成了 ImageClass 中的类方法。现在导师布置新任务，要小李把文献上的几种方法都试一下。当然，我们假设这些程序是现成的，小李只需构造若干成员方法，再把这些算法包装起来即可。小李首先想到的是，直接把这些方法都放到 ImageClass 中去，如右边的代码框中，小李先声明一个对象，并且依次对其调用类方法。

ImageClass

```
classdef ImageClass < handle
    properties
        image
    end
    methods
        img = deblurWrapper1(obj);
        img = deblurWrapper2(obj);
        ..... % 具体算法从略
        .....
        img = deblurWrapper10(obj);
    end
end
```

Script

```
i = ImageClass('someImage.jpg');
j = i.deblurWrapper1();
figure; imshow(j);
j = i.deblurWrapper2();
figure; imshow(j);
j = i.deblurWrapper3();
figure; imshow(j);
j = i.deblurWrapper4();
figure; imshow(j);
.....
```

这种设计的缺点是，随着新算法的加入，类 ImageClass 的体积将不断增大，而且通常每种算法还会有一些对应参数，这些参数也不可避免地要存放在 ImageClass 中，所以最终 ImageClass 类会像下面的代码那样，类的定义将变得复杂且难以管理。

ImageClass

```
classdef ImageClass < handle
    properties
        image
        arg1
        arg2
        .....
        arg10
```

```
    end
    methods
        img = deblurWrapper1(obj,arg1);  % 算法 1
        img = deblurWrapper2(obj,arg2);  % 算法 2
        ......
        ......
        img = deblurWrapper10(obj,arg10);% 算法 10
    end
end
```

测试时算法还可能遇到这样的要求：图像的解析度大小内容或者格式不同，对不同的图像要求采取不同的算法。当算法简单时，可以通过 if 语句来选择算法；而当算法的选择变得很多时，if/switch 语句将造成程序僵化。而且算法并不是一成不变的，当要更换算法时，就要频繁地深入到这个底层的 if/switch 语句中修改细节。

程序片段

```
switch methods
    case 'method1'
        img = imgObj.deblurWrapper1();
    case 'method2'
        img = imgObj.deblurWrapper2();
    ......
end
```

更实际的图像处理问题是，一个数据（图像）要分好几步来处理，比如先预处理，再用算法找到要处理的区域，再分隔出要处理的区域，再做局部的图像处理，每一步的方法都有好几种可以选择，if/switch 语句就更是力不从心了。那么，该如何设计程序呢？

当存在一些通用算法，并且还有另一些算法要作用在不同的数据上时，直觉使我们会想到使用继承，如图 17.9所示。

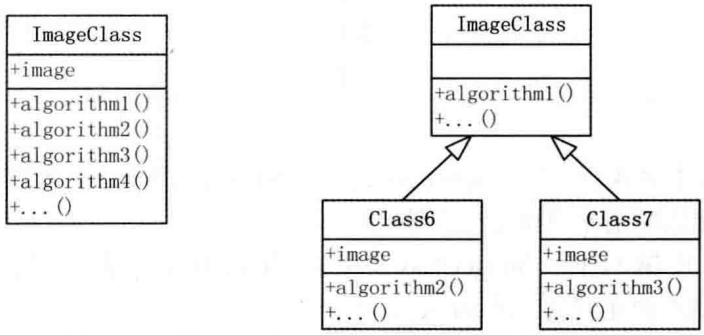


图 17.9 使用继承解决不同对象采用不同算法的问题

可是这样使用继承的设计依然存在问题，ImageClass 类还是很僵化，而且不能动态地改变应用在数据上的算法[①]。下面介绍使用 Strategy Pattern 来解决这种问题。

前面小李用的是成员方法包装数据和施加在数据上的算法。其实，类还可以用来只包装

①algorithm2，algorithm3 是固定在子类中的，同时固定在子类中还有数据。

算法，这就是策略模式。对于各种 deblur 的方法，可以抽象出一个基类来表达它们的共性，如图17.10所示。

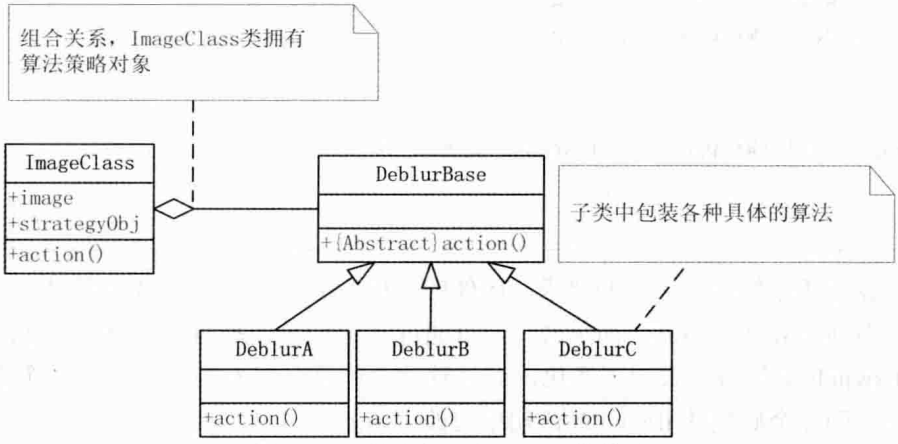


图 17.10 抽象出一个 Deblur 的基类

ImageClass 和 DeblurBase 基类的定义如下：

ImageClass

```
1  classdef ImageClass < handle
2   properties
3     image
4     strategyObj    %   拥有策略对象
5   end
6   methods
7     function obj = ImageClass(name)
8       obj.image = imread(name);
9     end
10    function img = action(obj)
11      img = obj.strategyObj.action(obj);
12    end
13  end
14 end
```

DeblurBase

```
1  classdef DeblurBase < handle
2      properties
3          arg
4      end
5      methods(Abstract)
6          img = action(imgObj);    % 抽象方法
7      end
8  end
9
10
11
12
13
14
```

说明：

- ImageClass 中第 4 行：属性 strategyObj 将指向具体的 Stategy 类的对象，可以通过对该属性的重新赋值来自由地更换算法。
- ImageClass 中第 11 行：ImageClass 类中的 action 方法只是一个包装方法，将对图像做处理的请求转化到具体的 strategy 对象中去了。
- 对图像数据到底采用何种算法取决于属性 strategyObj 到底指向了哪个 Strategy 对象。
- ImageClass 对象把自己当做参数传给 Strategy 类对象（第 11 行）。这样，Stragey 类中的具体的算法就可以直接访问 ImageClass 中的数据了。

具体的算法 DeblurA 和 DeblurB 看上去类似，都继承自抽象基类 DebluarBase，都要提供抽象方法的 action 的具体实现，这里省略。

DeblurA

```
classdef DeblurA < DeblurBase
   methods
       function img = action(obj,imgObj)
           % 具体操作略
       end
   end
end
```

DeblurB

```
classdef DeblurB < DeblurBase
   methods
       function img = action(obj,imgObj)
           % 具体操作略
       end
   end
end
```

在脚本中，我们这样使用策略模式：

Script

```
imgObj = ImageClass('someImage.jpg');
imaObj.strategy = DeblurA();      % 首先使用算法 A
img = imaObj.action();

imaObj.strategy = DeblurB();      % 更换算法 B
imgObj.action();                  % 仍然调用 action 方法
```

从表面上来看，并没有简化很多，但是整个程序把数据和处理数据的方法分散到了不同的类中，各个类各司其职。

17.2.2 应用：更复杂的分离数据和算法的例子

下面讨论一个更实际的图像处理的情况，比如人脸特征识别。通常，含有人脸的图像来自不同的数据库。不同的数据库光照不同，计算过程中的第一步通常是对图像做预处理。不同数据库中图像的类型不同，有的是人脸库，有的是半身照，有的是全身照，通常还要先用算法在图像中找到感兴趣的区域，即人脸。找到感兴趣的图像区域之后是用算法更精准地分割出该区域，最后一步才是对该区域做特征提取。以上每一个步骤都有不止一种算法，于是每一个步骤都可以使用一个 Strategy 基类，再把具体算法放到其子类中去。整个过程如图17.11所示。

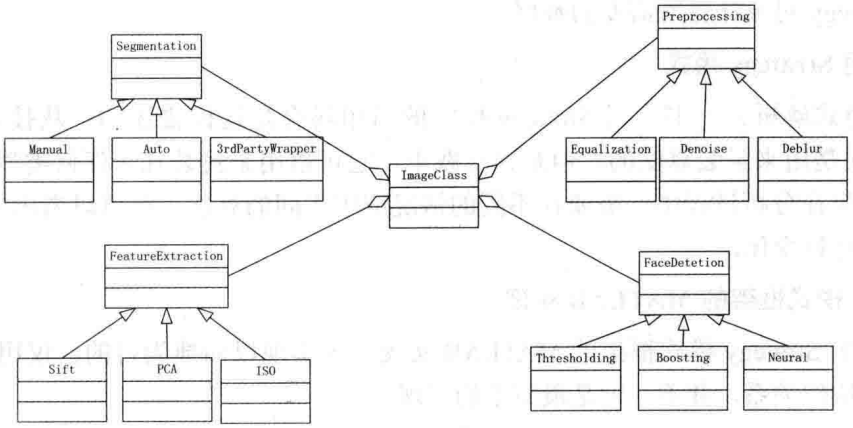


图 17.11 图像处理中的不同步骤都有不同的算法，使用多个 Strategy

有了这样一个框架，可以把施加在数据上的算法根据计算的不同阶段分得更细，并且自由的组合算法可达到最优的效果。比如从预处理基类策略中，可以实现平衡化算法子类、去

噪子类、去模糊子类。在特征提取策略中，可以在子类中实现各种流行的特征提取算法，这些算法可以是已经用 MATLAB 面向对象语言写成的算法，也可以是对第三方的图像处理库提供的算法的 MATLAB 包装类。在第17.5节中，将会介绍如何进一步抽象这个设计。

17.2.3 Strategy 模式总结

《设计模式》一书中对 Strategy 模式的意图是这样叙述的：

> 策略模式的意图是定义一系列的算法，把它们一个个封装起来，并且使它们可以互相替换。该模式使得算法可以独立于使用它的用户而变化。
>
> Design Patterns GOF

1. Strategy 模式结构

Strategy 模式的结构如图17.12所示。

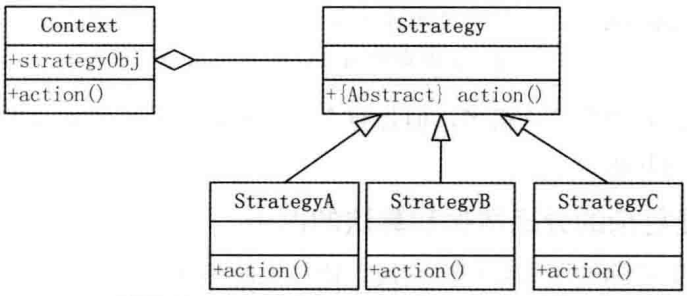


图 17.12 Context 类拥有 Strategy 对象，可以在 Context 对象上施加不同的策略

2. 类之间的协作

- □ 在 Context 对象中存放数据，而 Strategy 对象中存放算法，Context 对象可以选择所需要的算法。
- □ Context 对象把来自外界的计算请求转交给 Strategy 对象。
- □ 当转交计算请求时，Context 对象把自己作为一个参数传递给 Strategy 对象，以提供 Strategy 对象计算所需要的数据。

3. 何时使用 Strategy 模式

《设计模式解析》一书中对 Strategy 模式的适用场合是这样描述的：从技术角度而言，Strategy 模式是用来封装算法的，但是在实践中，它可以用来封装几乎任何类型的规则。一般来说，如果在分析过程中，需要在不同的情况应用不同的算法，就可以考虑使用 Strategy 模式来处理这种变化。

4. Strategy 模式框架的 MATLAB 实现

下面给出 Strategy 模式框架的 MATLAB 实现。该实现以简捷为目的，仅用来说明类之间的重要的协作关系，并不一定是最完整的实现。

Context

```
classdef Context < handle
    properties
```

```
        data
        strategyHandle
    end
    methods
        function action(obj)
            obj.strategyHandle(obj);  % 把来自外界的请求转交给 strategy 对象
        end
    end
end
```

算法基类：

Strategy

```
classdef Strategy < handle
    methods(Abstract)
        action(contextObj);
    end
end
```

算法具体类：

StraA

```
classdef StraA < Strategy
    methods
        function action(contextObj)
            ......
        end
    end
end
```

StraB

```
classdef StraB < Strategy
    methods
        function action(contextObj)
            ......
        end
    end
end
```

17.3 遍历者模式：工程科学计算中如何遍历大量数据

17.3.1 问题的提出

工程科学计算中，经常会碰到这样的问题：要遍历一个“集合”中的所有“元素”，这些元素可能是程序计算所需要的输入参数；也可能是要处理的数据、信号或图像，小到遍历一个数组，大到遍历一个目录中的所有文件（*.txt,*.jpeg,*.mat etc）。前面我们已经介绍过 Strategy Pattern，知道可以利用 Strategy Pattern 对数据使用不同的算法。这一节介绍有很多数据时，如何更好地遍历这些数据。从集合中提取元素的操作如图17.13所示。

从简单的面向过程的编程方式入手。如果所有的文件都是按序编号的，那么可以首先批量地产生文件名称，然后在一个循环中依次地打开它们，挨个对每个图像采用特定的算法。比如图像的命名规律是 a1.jpeg,a2.jpeg,...,a100.jpeg，我们可以把这些文件都放到一个文件夹中，然后在循环中生成文件名，打开，计算，然后保存结果，代码如下：

```
for i=1:100
    inputname = strcat('a',num2str(i),'.jpeg');
```

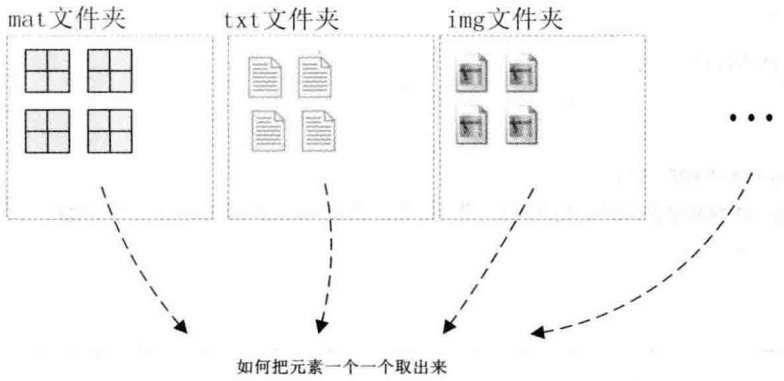


图 17.13 从集合中提取元素是一个常见的操作

```
    imgObj = Image(inputname) ;    % 假设已经有一个 Image 类，内部处理打开图像文件进行处理
    imaObj.doMethod();             % 处理数据
end
```

为简单起见，这里省略了给 Image 选择算法的部分，具体可以见第17.2节。其中，Image 是一个类，接受文件名称构造一个图像对象 imgObj。对于这个任务，MATLAB 仅用了 5 行代码就把问题解决了。

更实际的情况是，要处理的数据并不一定总是按规律命名的，假设有如下一批图像文件：

```
b1_Monday.jpeg,b2_Monday.jpeg,b3_Wednesday,...b20,c1.jpeg,.....c5.jpeg
```

因为文件名中间的数字不是连续的，是没有规律，所以动态地产生文件名的方式不是最有效的。这种情况下，我们想到可以利用 MATLAB 的一个内置函数 dir。该函数返回一个文件夹下所有文件的名称。其中因为 dir 函数除了返回目录中的文件，还会返回目录中子文件夹的名称，第 2 行取出了 dir 返回的结果中的文件夹的名称。

```
1 files = dir(c:\datafolder);
2 files = files(cell2mat({files(:).isdir})~=1)    % 去掉文件夹中的目录
3 for i = 1 : length(files)
4         inputname = files(i).name;
5         imgObj = Image(inputname);
6         imgObj.doMethod();
7 end
```

无论如何，MATLAB 还是相对简捷地解决了这个问题。采用 OOP 能比这段程序做得更好吗？我们再进一步给这段程序添加要求。比如，每次计算都产生了一些计算结果，其中一些结果是 LOG 形式的文本文件，一些结果是作为数据保存下来的，如 MAT 文件。在所有的计算完成之后，我们需要遍历这些文件以得到综合结果。那么，我们就要求写出至少三段类似的代码：

- 第一段代码遍历原始数据做计算，产生结果并且把结果保存下来。
- 第二段代码遍历 MAT 文件，作图。

□ 第三段代码遍历 LOG 文件，作图。

这样会产生很多重复的代码。再比如，如果数据的量很大，我们希望随机采样，而不是全部遍历，这就要在循环中加一个随机函数。如果要求提供更多遍历的方式，比如按照时间顺序从后往前遍历。这些要求不可避免地需要更多的集合的具体信息，因此信息难免暴露在外部程序中，使得程序过分依赖于具体实现，还会使程序变得不容易修改和扩展。容易看出，这些需求都关系到遍历元素集合的方式。在设计模式中，有现成的解决方案，这就是遍历者模式——Iterator Pattern。

17.3.2 聚集（Aggregator）和遍历者（Iterator）

要了解 Iterator Pattern，首先要了解 Aggregator。Aggregator 是集合的一种抽象，就是被 Iterator 遍历的对象。比如一个文件夹中的所有文件，就是一种 Aggregator，既然我们对遍历的方式可以用类抽象出来，那么被遍历的对象自然也可以用类抽象出来。下面直接给出遍历者模式的 UML（见图17.14）。

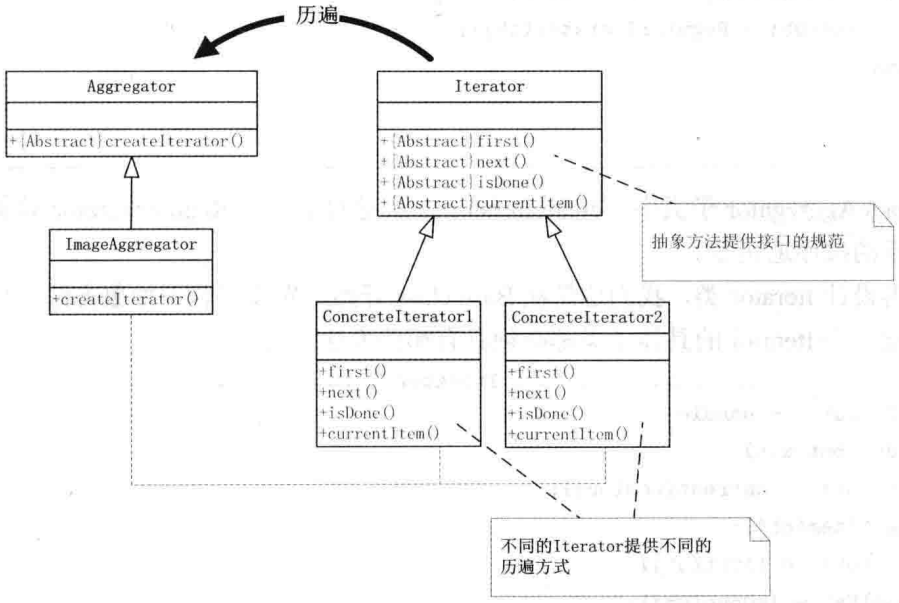


图 17.14 Aggregator 是集合的抽象，Iterator 遍历具体的 Aggregator 对象

下面是 Aggregator 的基类定义，其中定义了一个抽象方法 createIterator，所以是一个抽象类。在这节的例子中，具体的 Aggregator 是图像文件（名称）的集合[①]，具体的 Aggreator 在 createIterator 方法返回一个具体的遍历器对象，告诉外部，用什么方式来遍历自己，外部程序直接使用这个遍历器，就可以按照指定的方式访问 Aggregator 中的图像文件了。

Aggregator

```
classdef Aggregator < handle
    methods(Abstract)
        iterObj = createIterator(obj);
    end
```

①并不是一次性全部打开这些图像文件，而是遍历到哪个文件，就打开哪个文件。

```
end
```

ImageAggregator

```
classdef ImageAggregator < Aggregator
    properties
        files       % 保持所有聚集对象所有的文件
        totalNum;
    end
    methods
        function obj = ImageAggregator(path)
           tmp = dir(path);
           structs = tmp(cell2mat({tmp(:).isdir})~=1);
           obj.files = {structs.name};
           obj.totalNum = length(obj.files);
        end
        function iterObj = createIterator(obj)
            iterObj = RegularIterator(obj);
        end
    end
end
```

在 ImageAggregator 子类中，createIterator 方法返回了一个 RegularIterator 对象，表示这是一个正常的按序遍历器。

下面再设计 Iterator 类，我们还是从 Base class 开始，先设置若干抽象方法。这些抽象方法用来规定一个 Iterator 的具体子类都必须具有哪些方法。

Iterator

```
1 classdef Iterator < handle
2     methods(Abstract)
3         itemObj  = currentItem(obj);
4         nextItem(obj);
5         itemObj  = first(obj);
6         boolVal = isDone(obj);
7     end
8 end
```

说明：

- 第 3 行为 currentItem 用于返回其当前所指向的元素。如果遍历的是图像文件，则返回一个图像文件 ImageClass 的对象。
- 第 4 行的 nextItem 方法把内部指示器指向下一个元素。
- firstItem 返回聚集中的第一个图像对象；isDone 用来测试是否已经遍历所有的元素。

下面是一个具体的正常顺序遍历器的定义：

RegularIterator

```
classdef RegularIterator < Iterator
    properties
```

```
        aggHandle  % 遍历器拥有要遍历的聚集的 Handle
        counter    % counter 用来标记遍历器在集合中的位置
    end
    methods
        function obj = RegularIterator(agg)
            obj.aggHandle = agg ;
            obj.counter = 1 ;
        end
        function itemObj = first(obj)
           name = obj.aggHandle.files(1).name;
           itemObj = imread(name);  % 取出单个的 image
        end
        function itemObj = currentItem(obj)
           itemName =  obj.aggHandle.files(obj.counter);
           itemObj = imread(itemName);
        end
        function  nextItem(obj)
           obj.counter = obj.counter +1 ;  % 遍历器指向下一个元素
        end
        function boolVal = isDone(obj)
           boolVal = (obj.counter == obj.aggHandle.totalNum+1);
        end
    end
end
```

这样的设计对扩展是开放的，如果需要后向前遍历的遍历，只需要再添加一个类，叫做ReverseIterator，并且对四个方法做稍微的修改即可，反向遍历时，容器中的第一个元素是指向聚集中的最后的一个元素，next 方法相应的应该是每遍历一个元素减 1，isDone 其实就是判断 Iterator 内部的 counter 是否等于零，代码如下：

ReverseIterator

```
classdef ReverseIterator < Iterator
    properties
        aggHandle
        counter
    end
    methods
        function obj = RegularIterator(agg)
            obj.aggHandle = agg ;
            obj.counter = agg.totalNum ;
        end
        function itemObj = first(obj)
           name = obj.aggHandle.files(end).name;
           itemObj = imread(name);
        end
        function itemObj = currentItem(obj)
           itemName =  obj.aggHandle.files(obj.counter);
```

```
            itemObj = imread(itemName);
        end
        function  nextItem(obj)
            obj.counter = obj.counter  - 1 ;   % 遍历器指向下一个元素
        end
        function boolVal = isDone(obj)
            boolVal = (obj.counter == 0 );
        end
    end
end
```

上述的程序通过实现各种具体的遍历器来提供对一个复杂的 Aggregator 的多种方式的遍历的支持。Iterator 的存在简化了 Aggregator 的实现，Aggregator 内部不需要再提供遍历的机制，Aggregator 内部的细节也就不会暴露在外部。一个简单的使用遍历器的脚本看上去是这样的：

Script

```
agg = ImageAggregator(path);
iter = agg.createIterator();
while ~iter.isDone()
   image = iter.currentItem();          % 取出一个图像
   %  这里对 Image 对象做实际的计算工作
   iter.nextItem();                     % 遍历器移向下一个元素
end
```

回到前面的计算例子中，假设我们的工程计算需要遍历三种不同的数据库，而且每个数据库要求的遍历方式不同。我们可以给每个数据库设计一个 Aggregator 子类，由这些 Aggregator 子类负责产生具体的 Iterator 对象，Iterator 负责从数据库中取出数据，传递给 DataClass 做计算，计算后产生结果。这些结果本身也是大量元素的聚集，若有需要，也可以为这些结果设计具体 Aggregator 类用 Iterator 去遍历，整个程序的设计 UML 如图 17.15 所示。

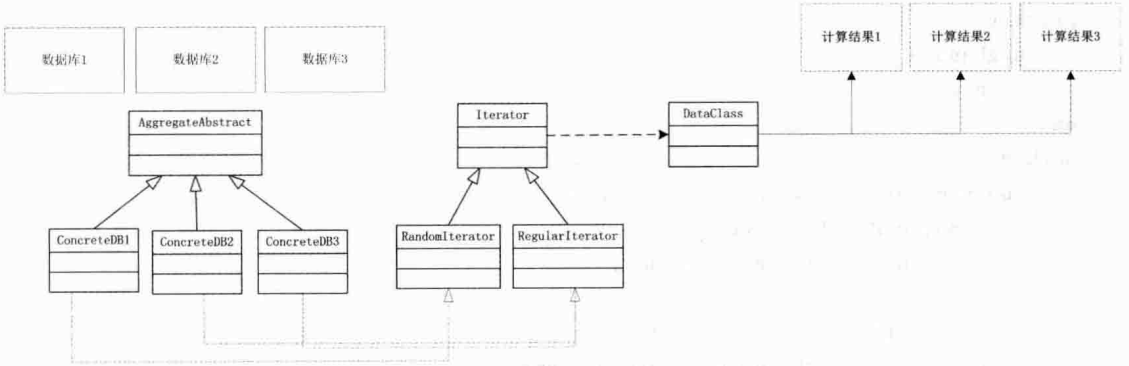


图 17.15 Aggregator 和 Iterator 实例

17.3.3 Iterator 模式总结

《设计模式》一书中对 Iterator 模式的意图是这样叙述的：

> Iterator（迭代器）的意图是用一种方法顺序去访问一个聚集对象中的各个元素，而又不用暴露对象的内部表示。
>
> Design Patterns GOF

1. Iterator 模式结构

Iterator 模式的结构如图17.16所示。

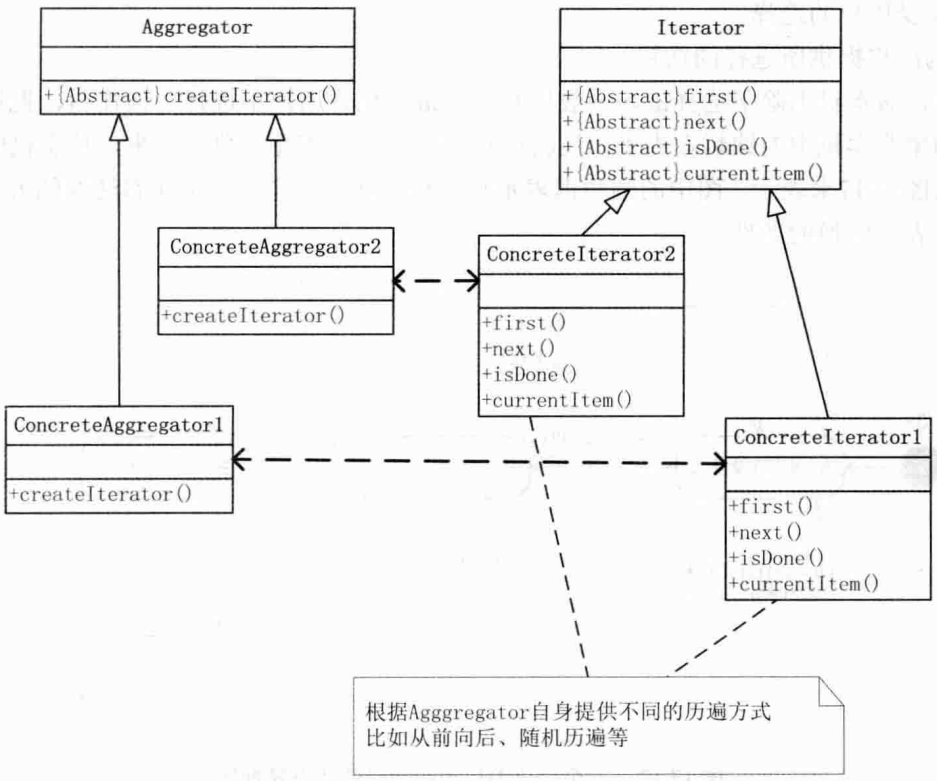


图 17.16 抽象的 Aggregator 和 Iterator 类提供接口，具体的 Iterator 类提供不同的遍历方式

2. 类之间的协作

- 因为遍历的方式和 Aggregator 自身的内部情况有关。只有 Aggregator 才有足够的信息来告诉外部类该如何遍历自己，所以具体的 Aggregator 负责产生具体的 Iterator。
- 具体的 Iterator 类将拥有 Aggregator 的 Handle 用来从集合中取出元素。
- 两个 Base 类用来提供接口，规定具体的 Aggregator 和具体的 Iterator 应该提供何种方法，以及方法的 signature 应该是什么样的。
- ConcreteIterator 中提供具体的不同的遍历方式，比如 ConcereteIterator1 可能提供的遍历方式是从前往后，而 ConcreteIterator2 提供的方式可以是随机遍历。

17.4 状态模式：用 MATLAB 模拟自动贩卖机

这一节以自动贩卖机为例，讲解如何用 MATLAB 模拟对象内部状态的转换。

17.4.1 使用 if 语句的自动贩卖机

大家对自动贩卖机应该都不陌生。其使用的过程是，用户塞入纸币，选择饮料（比如可乐、果汁或者矿泉水），然后贩卖机为用户提供所选择的饮料。上述过程中，贩卖机大致经历了以下几个状态：

- □ 没有接受任何用户的钱币。
- □ 接受了用户的钱币。
- □ 接受用户的选择。
- □ 为用户提供所选择的饮料。

还有，贩卖机上除了选择键，一般还有一个键，允许用户不选择任何饮料，把钱退回给用户，如果贩卖机中的饮料卖光了，还将提示用户饮料已售完。总结下来，贩卖机的状态变化可以用图17.17来表示。图中的圆角框表示贩卖机的状态，箭头表示状态转换的方向，箭头上的文字表示转换的条件。

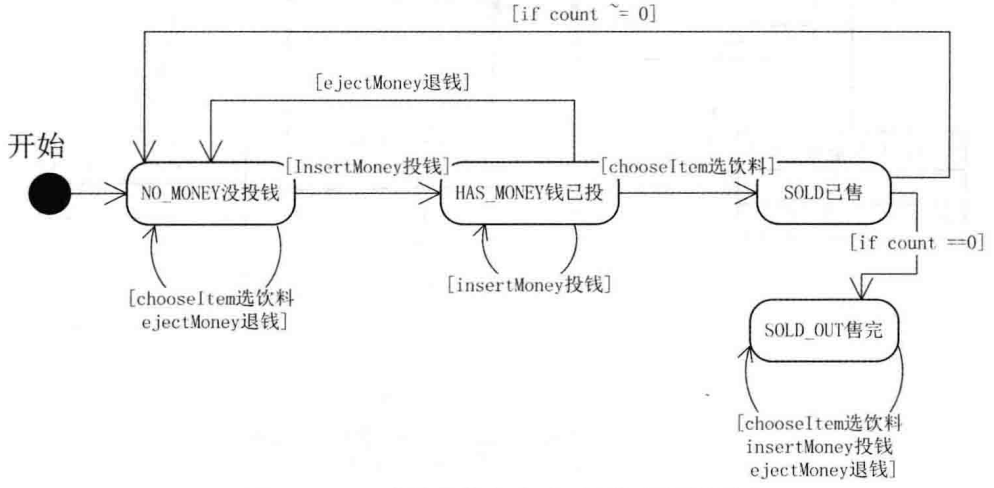


图 17.17 一个简化的自动贩卖机的状态转换图

观察其中的一个状态，如NO_MONEY 状态，其指向自身的弧线表示：在这种条件下，贩卖机的状态不变，如图17.18所示。用户如果没有付钱，贩卖机的状态将一直停留在NO_MONEY状态，无论是按选择饮料按钮，还是按退钱按钮，贩卖机都不会改变自身的状态。

下面用 MATLAB 来实现这个贩卖机的逻辑。首先尝试把贩卖机的所有状态都集中在一个类当中，为简单起见，该类的属性和方法都定义成 public：

VendingMachine

```
classdef VendingMachine < handle
   properties
       state       % 贩卖机内部的状态
       count       % 剩余商品的数量
   end
   properties(Constant)
       SOLD_OUT = 0;
       NO_MONEY = 1;
```

```
        HAS_MONEY =2;
        SOLD = 3;
    end
    methods
        function obj = VendingMachine(count)
            obj.count = count ;
            if count ==0
                obj.state = VendingMachine.SOLD_OUT;    % 商品数量为零显示没货
            else
                obj.state = VendingMachine.NO_MONEY;
            end
        end
        insertMoney(obj);
        ejectMoney(obj);
        chooseItem(obj);
        dispense(obj);
    end
end
```

图 17.18 指向自身的弧线表示：在该条件下，状态不改变

该类内部使用了四个 Constant 变量来表示贩卖机的四个状态：

- SOLD_OUT 表示售完。
- NO_MONEY 表示用户还没付钱。
- HAS_MONEY 表示用户已付钱。
- SOLD 表示商品已售。

该类的构造方法接受一个参数 count 用来初始化对象中商品的数量。如果 count 是零，那么贩卖机对象被实例化之后的状态是SOLD_OUT。该类还有四个成员方法，分别对应四个简单的操作：

- insertMoney 表示用户提供钱币。
- ejectMoney 表示用户还没有选择商品，按键要求贩卖机退还。
- chooseItem 表示用户按键选择商品。
- dispense 表示贩卖机出货。该 dispense 操作将在用户选完商品之后自动被触发。

其类图如17.19所示。

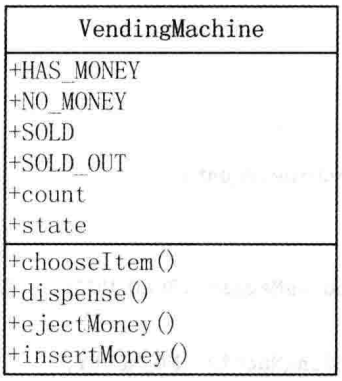


图 17.19 VendingMachine 类的初步设计：状态作为成员属性

下面我们使用 if statement 来实现这四个方法。首先是 insertMoney 成员方法，针对用户往贩卖机中付钱这个动作：

```
function insertMoney(obj)
    if obj.state == VendingMachine.HAS_MONEY          % 如果状态是“已经付过钱”
        disp('You have already paid');                % 提示用户钱已付
    elseif obj.state == VendingMachine.NO_MONEY       % 如果状态是“机器中没钱”
        obj.state = VendingMachine.HAS_MONEY ;        % 改变状态成为 HAS_MONEY
    elseif obj.state == VendingMachine.SOLD_OUT       % 如果状态是“机器中没有商品”
        disp('Machine is Sold out')                   % 提示没有商品 状态停留在 SOLD OUT
    elseif obj.state == VendingMachine.SOLD
        % 假设程序不会执行到这里
    end
end
```

insertMoney 方法中实现的逻辑反应到状态图中是图17.20中的虚线部分。

类似地，ejectMoney 成员方法针对用户要求机器退钱这个动作，可以这样实现：

```
function ejectMoney(obj)
    if obj.state == VendingMachine.HAS_MONEY          % 如果状态是“已经付过钱”
        disp('Returning money');                      % 提示 正在退钱
        obj.state = VendingMachine.NO_MONEY;          % 退钱后改变状态成为 NO_MONEY
    elseif obj.state == VendingMachine.NO_MONEY       % 如果状态是：机器中没钱
        disp('Show me the money');                    % 提示 你没塞过钱
    elseif obj.state == VendingMachine.SOLD_OUT       % 如果状态是“机器中没有商品”
        disp('Machine is Sold out')                   % 提示没有商品
    elseif obj.state == VendingMachine.SOLD           % 如果状态是“已售”
        disp('Too late');                             % 提示商品已售 不能退钱了
    end
end
```

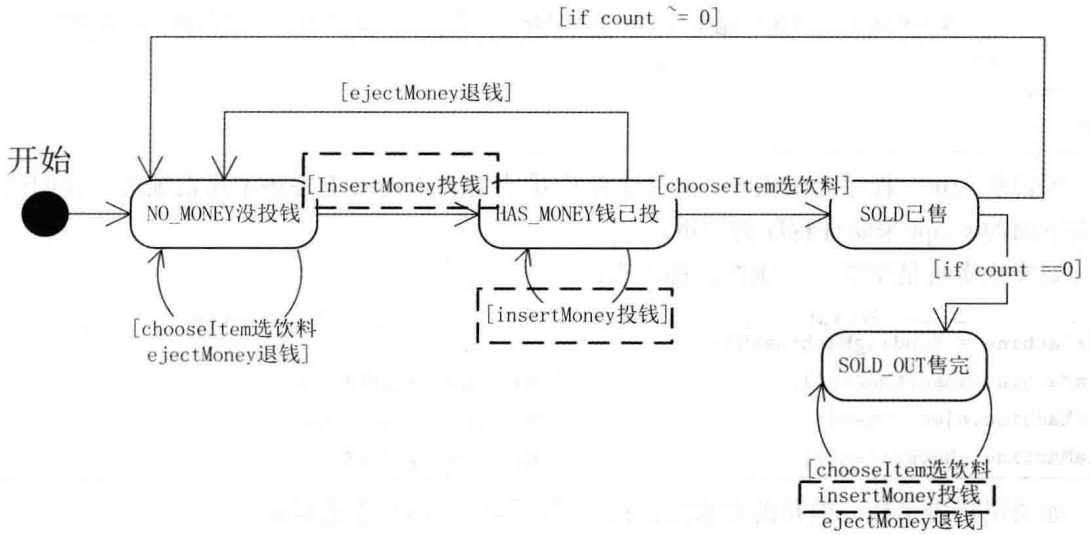


图 17.20 insertMoney 方法实现了状态图中的虚线部分

chooseItem 成员方法是用户按键选择商品的动作的响应：

```
function chooseItem(obj)
    if obj.state == VendingMachine.HAS_MONEY                % 如果状态是“已经付过钱”
        obj.state = VendingMachine.SOLD ;                   % 改变状态称为已售
        obj.dispense();                                     % 触发 dispense 方法
    elseif obj.state == VendingMachine.NO_MONEY
        disp('Show me the money');
    elseif obj.state == VendingMachine.SOLD_OUT
        disp('Machine is Sold out')
    elseif obj.state == VendingMachine.SOLD                 % 如果状态是“已售”
        disp('You have already push the button')            % 提示你已经选过了
    end
end
```

dispense 方法从贩卖机中取出饮料，饮料数量减 1，如果 count 变成零，把机器的状态置为SOLD_OUT：

```
function dispense(obj)
    if obj.state == VendingMachine.HAS_MONEY
       % do nothing
    elseif obj.state == VendingMachine.NO_MONEY
       % do nothing
    elseif obj.state == VendingMachine.SOLD_OUT
       % do nothing
    elseif obj.state == VendingMachine.SOLD
        disp('dispensing');
        obj.count = obj.count - 1;
        if obj.count ==0
            obj.state = VendingMachine.SOLD_OUT;  % 如果饮料售完，状态设成 SOLD_OUT
        else
```

```
            obj.state = VendingMachine.NO_MONEY;  % 如果还有饮料，状态设成 NO_MONEY
        end
    end
end
```

到目前为止，程序还不太长，各个方法中排比的if语句看上去还不是很烦琐。我们可以使用下面的script来验证程序的逻辑。

如果贩卖机是空的，一瓶饮料都没有：

Script	Command Line
colaMachine = VendingMachine(0);	
colaMachine.insertMoney();	Machine is Sold out
colaMachine.ejectMoney();	Machine is Sold out
colaMachine.chooseItem();	Machine is Sold out

如果用户投了钱，但是没有选择，按下了退钱键，再按下选择键：

Script	Command Line
colaMachine = VendingMachine(1);	
colaMachine.insertMoney();	
colaMachine.ejectMoney();	Returning money
colaMachine.chooseItem();	Show me the money

贩卖机中只有两瓶饮料，被两个用户买走了，之后的用户如果再投钱，则显示SOLD_OUT：

Script	Command Line
colaMachine = VendingMachine(2);	
colaMachine.insertMoney();	
colaMachine.chooseItem();	dispensing
colaMachine.insertMoney();	
colaMachine.chooseItem();	dispensing
colaMachine.insertMoney();	Machine is Sold out
colaMachine.insertMoney();	Machine is Sold out

下面继续扩充这个模型，假设贩卖机只接受一元钱纸币，但是一瓶可乐是两元钱，这里需要增加一个状态，就是当用户只塞了一元钱时，这时机器的状态是NOT_ENOUGH（钱还不够）。这个新的状态对整个状态图的影响如17.21图所示。

上面状态图中新增加的逻辑是：

- NO_MONEY时塞入一元钱，机器进入NOT_ENOUGH状态，再塞入一元钱进入HAS_MONEY状态。
- 在NOT_ENOUGH状态，如果用户可以要求退钱，状态返回到NO_MONEY。
- 在NOT_ENOUGH状态，如果用户按键选择饮料，则提示钱不够，并且贩卖机的状态仍然停留在NOT_ENOUGH状态。

为了增加这个新的状态，需要修改下列部分的代码（图17.21中粗体部分）：

- 在 VendingMachine 中增加一个 State。
- ejectMoney 方法。

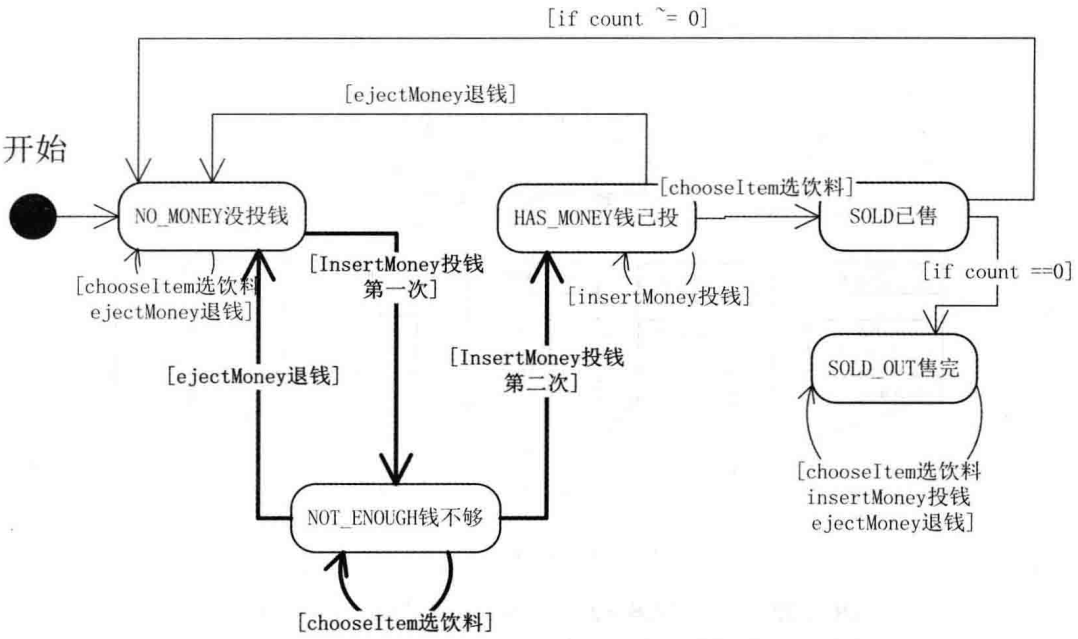


图 17.21 可乐改成两块钱一瓶，所以贩卖机要添加一个状态

- chooseItem 方法。
- InsertMoney 方法。

这些改动意味着：每个方法中都要增加额外的 if 语句。我们又面临这个老问题了，一个新的需求影响到很多部分，牵一发则动全身，要把所有需要修改的地方都找出来，并且修改得一致。这个设计额缺陷是，关于各个状态的具体逻辑散布在各个类当中，导致一旦需要对一个状态的逻辑加以修改，需要更新很多地方。而把一个状态的逻辑都集中在一个类当中的模式，就是我们要介绍的 State 模式！

17.4.2 使用 State Pattern 的自动贩卖机

为了使设计容易维护和扩展，我们要做的事情是：

- 首先定义一个 State 抽象类，在这个类中，每个贩卖机的动作都对应一个方法。
- 为机器的每个状态都写一个具体的类，在这些类中，实现对应状态下的行为。
- 摆脱旧的 if 语句，取而代之的是将动作委托到状态类当中去，使用多态来代替相关逻辑条件。

图17.22所示为使用状态模式的贩卖机，其中每个状态对应一个类。

下面先直接给出整个设计的 UML，并且解释什么叫做“把动作委托到状态类当中去，使用多态来代替逻辑条件”。图17.23所示为使用状态模式的贩卖机，Vending Machine 拥有的状态类对象。

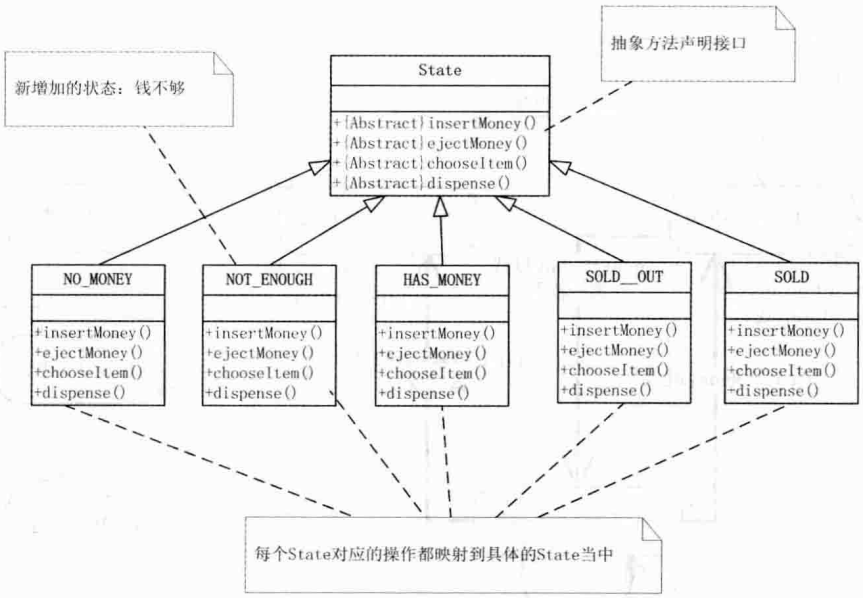


图 17.22 使用状态模式的贩卖机：每个状态都对应一个类

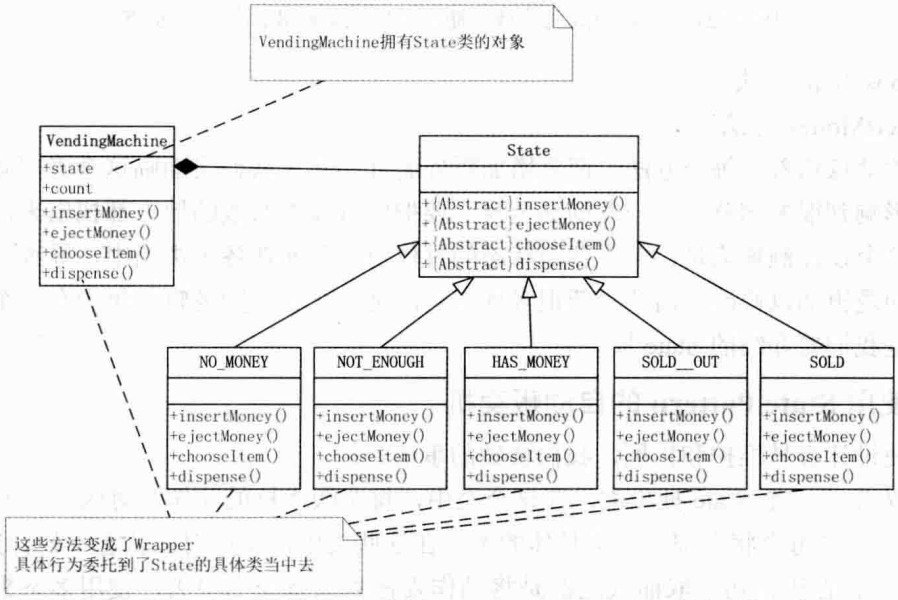


图 17.23 使用状态模式的贩卖机：VendingMachine 拥有状态类对象

从 VendingMachine 类开始讨论，贩卖机类中的一个属性是 State 类对象，而类中的方法仅仅是 State 类对象方法的包装方法（Wrapper），所以 VendingMachine 类的定义较之前简化了很多。

VendingMachine

```
classdef VendingMachine < handle
   properties
       state
       count
   end
    properties(Constant)
```

```
        SOLD_OUT = 0;
        NO_MONEY = 1;
        HAS_MONEY =2;
        SOLD = 3;
        NOT_ENOUGH =4;
    end
    methods
        function obj = VendingMachine(count)
            obj.count = count ;
            if count ==0
                obj.state = SoldOut();
            else
                obj.state = NoMoney();
            end
        end
        insertMoney(obj);          % Wrapper Method
        ejectMoney(obj);
        chooseItem(obj);
        dispense(obj);
    end
end
```

其中的四个 Method 仅仅是 Wrapper，如下所示，具体工作委托到具体 State 中去完成。

```
function insertMoney(obj)
    obj.state.insertMoney(obj);
end
```

```
function ejectMoney(obj)
    obj.state.ejectMoney(obj);
end
```

```
function chooseItem(obj)
    obj.state.chooseItem(obj);
end
```

```
function dispense(obj)
    obj.state.dispense(obj);
end
```

说明：

- 四个成员方法不知道obj.state是哪个具体的状态，这并不重要，因为请求被传递给具体的 State 的成员方法了，对象的行为将取决那个具体的状态。
- 在调用 State 的方法时，VendingMachine 还把自己的句柄传给了该 State 对象，即 obj 参数，这是因为 State 的具体动作将涉及改变 VendingMachine 自身的状态，所以 Vending Machine 必须提供一个渠道给具体的 State 对象，允许其帮自己来改变状态。

下面实现具体的状态类。为简单起见，所有的属性和方法都定义成 public 的，首先是 NoMoney 状态类，最明显的不同是，if 语句被分散到成员方法当中去了。

NoMoney

```
classdef NoMoney < State
    methods
```

```
        function insertMoney(obj,machineObj)
            machineObj.state = NotEnough();  % 改变 machine 对象内部的状态
        end
        function ejectMoney(obj,machineObj)
            disp('Show me the money');
        end
        function chooseItem(obj,machineObj)
            disp('Not enough Money');
        end
        function dispense(obj)
            % do nothing
        end
    end
end
```

NotEnough 类是一个新的状态类，在这个状态下，如果用户再次 insertMoney，贩卖机的状态将变成 HAS_MONEY，如果用户再选择 ejectMoney，则贩卖机的状态将退回到 NO_MONEY 状态。

NotEnough

```
classdef NotEnough < State
    methods
        function insertMoney(obj,machineObj)
            machineObj.state = HasMoney();      % 再投入一元钱转成 HAS_MONEY 状态
        end
        function ejectMoney(obj,machineObj)
            disp('returning the money');
            machineObj.state = NoMoney();
        end
        function chooseItem(obj,machineObj)
            disp('Not enough Money');
        end
        function dispense(obj)
            % do nothing
        end
    end
end
```

HasMoney 状态的逻辑和之前基本一样：

HasMoney

```
classdef HasMoney < State
    methods
        function insertMoney(obj,machineObj)
            disp('You have already paided');
        end
```

```
        function ejectMoney(obj,machineObj)
             disp('returning the money');
             machineObj.state = NoMoney();
        end
        function chooseItem(obj,machineObj)
            machineObj.state = Sold();
            machineObj.dispense()
        end
        function dispense(obj)
            % do nothing
        end
    end
end
```

SOLD 类的逻辑和之前版本基本一样：

Sold

```
classdef Sold < State
    methods
        function insertMoney(obj,machineObj)
             % WILL NEVER ENTER HERE
        end
        function ejectMoney(obj,machineObj)
             disp('Too late');
        end
        function chooseItem(obj,machineObj)
              disp('You have already push the button')
        end
        function dispense(obj,machineObj)
             disp('dispensing');
             machineObj.count = machineObj.count - 1;
             if machineObj.count ==0
                 machineObj.state = SoldOut();
             else
                 machineObj.state = NoMoney();
             end
        end
    end
end
```

SoldOut 类的逻辑和之前基本一样：

SoldOut

```
classdef SoldOut < State
    methods
        function insertMoney(obj,machineObj)
             disp('Machine is Sold out')
```

```
        end
        function ejectMoney(obj,machineObj)
              disp('Machine is Sold out')
        end
        function chooseItem(obj,machineObj)
              disp('Machine is Sold out')
        end
        function dispense(obj,machineObj)
              % WILL NEVER ENTER HERE
        end
    end
end
```

我们通过下面的 Script 来验证机器状态的变化，经过改进的贩卖机，可乐单价为 2 元钱，用户要调用 insertMoney 两次才能得到饮料：

Script	Command Line
colaMachine = VendingMachine(2);	
colaMachine.insertMoney();	
colaMachine.insertMoney();	
colaMachine.chooseItem();	dispensing
colaMachine.insertMoney();	
colaMachine.chooseItem();	Not enough Money
colaMachine.ejectMoney();	returning the money

17.4.3 State 模式总结

《设计模式》一书中对 State 模式的意图是这样叙述的：

> 状态模式：允许对象修改内部状态是改变它的行为，对象看起来好像是修改了它的类。
>
> Design Patterns GOF

1. State 模式的结构

State 模式的结构如图17.24所示。

2. 类之间的协作

- □ 对于来自外界的和自身状态有关的请求，Context 对象将这些请求委托给 ConcreteState 对象处理。
- □ ConcreteState 对象可以访问 Context 对象，也可以改变 Context 对象的内部状态。这也就要求：在 request 方法中，Context 对象必须把自己作为一个参数传递给 ConcreteState 对象。
- □ 使用 Context 类的 Client 不需要和 State 的对象打交道。

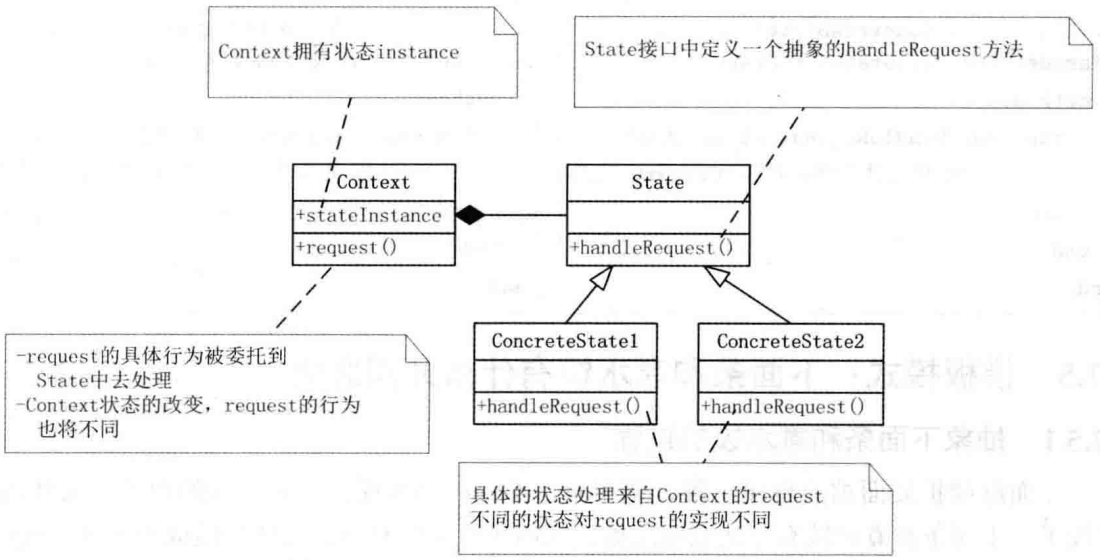


图 17.24 State 模式的结构：状态是具体的对象

3. 何时使用 State 模式

《设计模式》一书中对 State 模式的适用场合是这样描述的：

- 当对象的行为取决于它的状态，并且该对象在运行时会改变状态。也就是说，对象的行为会在运行时改变，可以考虑使用 State 模式。
- 当一个操作中含有庞大的多分支条件语句，而且这些分支语句依赖于对象的状态时，可以考虑使用 State 模式将每个条件分支放入一个独立的类之中，而对象的状态对应于拥有一个状态类对象。

4. State 模式框架的 MATLAB 实现

下面给出 State 模式框架的 MATLAB 实现。该实现以简捷为目的，仅用来说明类之间的重要的协作关系，并不一定是最完整的实现。接口抽象类 Context 和 State 的定义为：

Context

```
classdef Context < handle
    properties
        state
    end
    methods
      function request(obj,state)
          obj.state.handleRequest(obj);
      end
    end
end
```

State

```
classdef State < handle
    methods(Abstract)
        handleRequest(obj,context)
    end
end
```

两个 Concrete State 的简单实现：

ConcreteState1

```
classdef ConcreteState1 < State
  methods
    function handleRequest(obj,context)
            % 做具体工作，改变 Context 状态
    end
  end
end
```

ConcreteState2

```
classdef ConcreteState2 < State
  methods
    function handleRequest(obj,context)
            % 做具体工作，改变 Context 状态
    end
  end
end
```

17.5 模板模式：下面条和煮水饺有什么共同之处

17.5.1 抽象下面条和煮水饺的过程

下面继续扩展面馆的程序。假设现在面馆不仅向顾客提供各种风味的面条，又开始卖水饺了。下面条和煮水饺有什么共同之处，代码是否可以复用，这就是模板模式（Template Pattern）要解决的问题。首先，讨论下面条和煮水饺的过程。

简而言之，下面条需要如下步骤：

(1) 煮开水。

(2) 把面条放下去。

(3) 煮上 10 分钟。

(4) 把面条捞起来，加汤，加料。

煮饺子和下面条的过程类似：

(1) 煮开水。

(2) 把饺子放下去。

(3) 煮上 15 分钟。

(4) 把饺子捞起来，加汤。

先把上述每个步骤抽象成一个类的方法，并且把两个过程的类图表示出来，第一版的设计如图17.25所示。

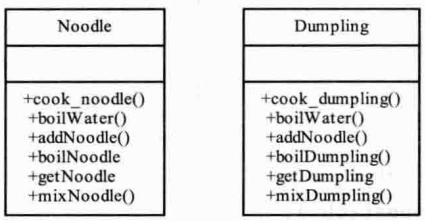


图 17.25 把下面条和煮水饺的过程用面向对象方法表示出来

另外，每个类还必须有一个方法负责按照正确的顺序调用 boilWater，add*，boil*，mix* 这些方法。我们给这个方法起名叫 cook。两个类的 cook 方法将是这样的：

```
classdef Noodle < handle
......
    methods
        function cook(obj)
            obj.boilWater();
            obj.addNoodle();
            obj.boilNoodle();
            obj.getNoodle();
            obj.mixNoodle();
        end
    end
......
end
```

```
classdef Dumpling < handle
......
    methods
        function cook(obj)
            obj.boilWater();
            obj.addDumpling();
            obj.boilDumpling();
            obj.getDumpling();
            obj.mixDumpling();
        end
    end
......
end
```

完成这两个任务的步骤十分相似，但在细节上又不完全相同。比如，煮饺子要比煮面条的时间长一些，面条煮好之后还要加主料和酱料，而饺子和着饺子汤即可……总之，我们发现了一些貌似重复的代码，这是好事情，表示需要考虑一下设计了。既然这两个类如此相似，似乎应该把共同的部分抽取出来，放进一个 Product 基类中去。图17.26是第二版的设计。

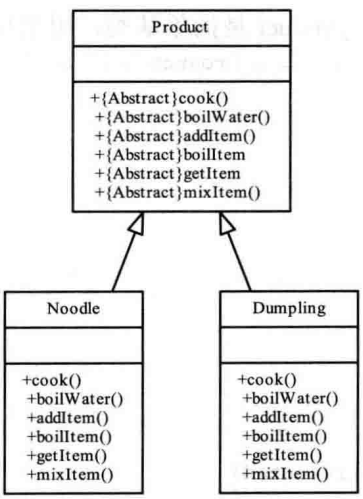


图 17.26 先把各个过程抽象到基类中

在这个设计中，把两个过程都需要有的方法综合出来，起统一的名字 addItem，boilItem，getItem，mixItem 作为基类的方法，因为各个方法的细节各有不同，所以在基类中声明这些方法成为 Abstract，要求在子类中有具体的实现。在第二版的基础上，我们再仔细观察一下 cook 方法，可以发现，因为下面条和煮水饺的步骤是一模一样的，所以 cook 方法可以作为一个非抽象方法放到基类中去。其实，这种把做事情的步骤抽象出来的模式，就是这节要介绍的 Template（模板）模式。图17.27是第三版的设计。

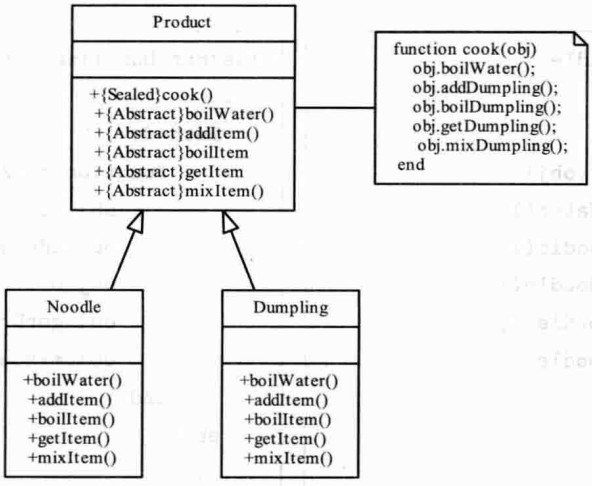


图 17.27 再把 cook 方法的过程抽象出来放到基类中去

和之前相比，我们还做了如下改进：

- 把 cook 方法设计成了 Seal 的方法。这样做的目的是固定烹调的顺序，禁止子类重新实现自己的 cook 方法，即禁止下面条和煮水饺子类任意调换烹饪的顺序。
- 把 boilWater，addItem，boilItem，getItem，mixItem 设计成了 protected 方法，目的是只允许类的内部方法（cook）调用它们。

下面是图17.27的实现代码，Product 是抽象基类，用来封装完成烹饪的步骤。

Product.m

```
classdef Product < handle
    methods(Sealed)
        function cook(obj)
            obj.boilWater();
            obj.addItem();
            obj.boilItem();
            obj.getItem();
            obj.mixItem();
        end
    end
    methods(Abstract,Access = protected)
        boilWater(obj);
        addItem(obj);
        boilItem(obj);
        getItem(obj);
        mixItem(obj);
    end
end
```

Noodle 类和 Dumpling 类继承自 Product 基类，为简单起见，在方法中，我们仅用 disp 语句的不同来表示方法在不同子类的细节上的不同。

```
classdef Noodle < Product
    methods(Access = protected)
        function  boilWater(obj)
            disp('boil water noodle');
        end
        function  addItem(obj)
            disp('add noodle');
        end
        function boilItem(obj)
           disp('boil noodle');
        end
        function getItem(obj)
           disp('get noodle');
        end
        function mixItem(obj)
           disp('mix noodle');
        end
    end
end
```

```
classdef Dumpling < Product
    methods(Access = protected)
        function  boilWater(obj)
            disp('boil water dumpling');
        end
        function  addItem(obj)
            disp('add dumpling');
        end
        function boilItem(obj)
           disp('boil dumpling');
        end
        function getItem(obj)
           disp('get dumpling');
        end
        function mixItem(obj)
           disp('mix dumpling');
        end
    end
end
```

用来测试的脚本以及命令行输出如下：

Script

```
clear classes ; clc ;
op1  = Noodle();
op1.cook();


op2 = Dumpling();
op2.cook();
```

Command Line

```
boil water noodle
add noodle

boil noodle
get noodle
mix noodle

boil water dumpling
add dumpling
boil dumpling
get dumpling
mix dumpling
```

上述的脚本中，先分别声明出两个不同的对象 op1 和 op2，代表下面条和煮水饺两个不同的过程，虽然其过程细节不同，但是它们都调用了相同的基类方法 cook。该 cook 方法再转而调用具体的 boilWater，addItem，…方法，但是到底调用哪个子类方法，则取决于对象到底属于哪个类。比如 op1 是做面条的类，虽然 cook 方法的实现在基类中，但是 op1.cook() 方法中调用的步骤是面条子类中的方法的具体实现。说得更具体一些，比如 cook 方法中的 obj.boilWater() 命令，将根据 obj 所属的类的不同调用不同的 boilWater 方法。

17.5.2 应用：把策略和模板模式结合起来

在第17.2.2小节中我们举过一个比较复杂的图像处理的例子，见图17.9。在这个例子中，我们假设对数据的处理有固定的一套步骤，但是每一套的步骤的具体算法都有多种选择，解决这个问题可以把策略模式和模板模式结合起来。这也是面向对象编程的优势所在。每个具体的设计模式都解决一个独立的小问题，在实际计算中，可以把各种模式结合起来使用，去解决更复杂的问题。

如果用模板模式来解决第17.2.2小节中人脸特征提取的工作，可以把固定的处理步骤封装在基类的 process 方法中，UML 如图17.28所示。

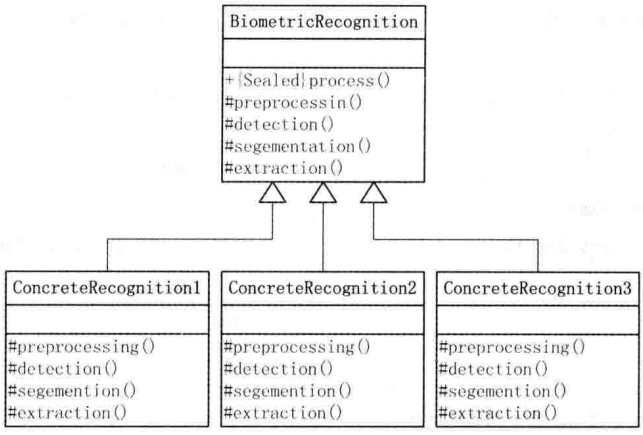


图 17.28 使用模板模式来解决人脸特征提取问题

如果想让 ConcreteRecognition 子类能够更加灵活地选择不同的算法，可以在该设计的基础上把具体的算法从子类中分离出来，利用组合，让 Recognition 基类拥有 Strategy 类的算法对象，如果以 preprocessing 算法为例，其 UML 如图17.29所示。

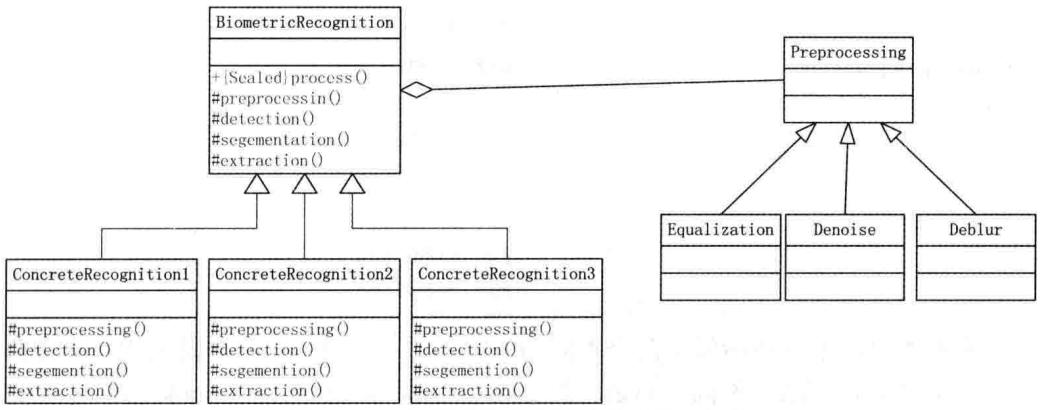


图 17.29 把模板模式和策略模式结合起来

当然，ConcreteRecognition 子类中的 preprocessing 方法只是一个包装方法，具体的实现被转移到 Processing 子类中去了。

17.5.3 Template 模式总结

《设计模式》一书中对 Template 模式的意图是这样叙述的：

> 模板方法模式：定义一个操作中的算法的骨架，将一些步骤的实现延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可以重新定义该算法的某些特殊的步骤。
>
> Design Patterns GOF

1. Template 模式的结构

Template 模式的结构如图17.30所示。

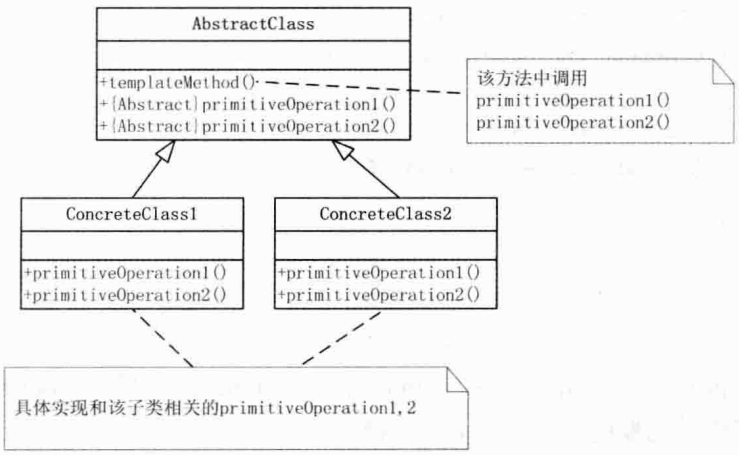


图 17.30 Template 模式中，父类调用子类方法，而不是子类调用父类方法

Template 模式中，含有反向控制的结构，因为通常都是子类调用父类的操作，而在这个模式里，却是父类调用子类的操作。这种结构也被叫做“好莱坞模式”，即“不要给我打电话，我们会通知你”[①]。

2. Template 模式框架的 MATLAB 实现

AbstractClass.m

```
classdef AbstractClass < handle
    methods(Sealed)
        function templateMethod(obj)              % 模板方法封装完成工作的步骤
           obj.primitiveOperation1();
           obj.primitiveOperation2();
        end
    end
    methods(Abstract,Access = protected)
       primitiveOperation1(obj);
       primitiveOperation2(obj);
    end
end
```

说明：

①Hollywood Principal: “don’t call us, we’ll call you.”

- □ 基类中的 templateMethod 要声明成 sealed，这样可以保证子类中不会有方法违反规定，重新定义 templateMethod 方法。因为按照约定，templateMethodtext 中的调用顺序是固定的，只是其中 primitiveOperation 的具体实现因子类而异。
- □ 两个 primitiveOperation 的方法要声明成 Abstract，这样就强制任何继承 AbstractClass 的子类都必须提供这两个方法的具体实现，并且这样还把接口固定了下来。

ConcreteClass.m

```
classdef ConcreteClass < AbstractClass
    methods(Access = protected)
        function primitiveOperation1(obj)
            disp('Concrete implementation of OP1');
        end
        function primitiveOperation2(obj)
            disp('Concrete implementation of OP2');
        end
    end
end
```

- □ 基本方法 primitiveOperation1 和 primitiveOperation2 要声明成 protected，这是为了保证这些方法只能被内部的（即 tempateMethod) 调用。

用来测试的脚本以及命令行输出如下：

Script

```
obj = ConcreteClass();
obj.templateMethod();
```

Command Line

```
Concrete implemenation of OP1
Concrete implemenation of OP2
```

17.6 备忘录模式：实现 GUI 的 UNDO 功能

17.6.1 如何记录对象的内部状态

有时程序在运行过程中有必要记录一个对象的内部状态，并且在某些情况下，把对象恢复到之前的状态，如图17.31所示。要实现这种机制，就必须事先把对象的状态信息保存起来。

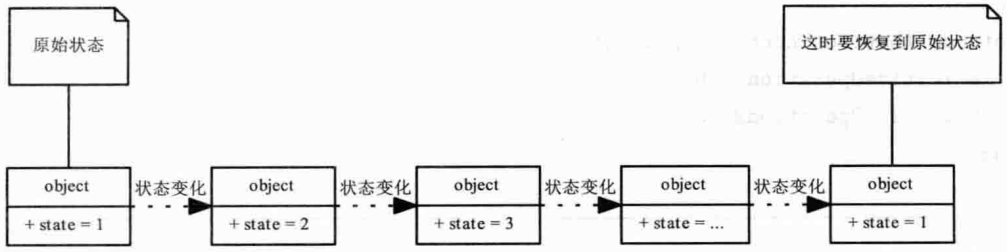


图 17.31 状态随时间变化，在某些情况下，需要将对象的状态恢复到从前

保存对象的状态信息不等同于直接保存对象本身，因为对象本身可能还包含了状态以外

的其他数据，程序也许不需要恢复那些数据，所以不能期望简单地仅使用 save 命令来完成任务。因此，有必要仅把对象中有用的信息单独提取出来加以保存，必要时加以恢复。在设计模式中，有现成的解决方案，叫做备忘录（Memento）模式。

下面给出备忘录模式的简单实现：

Model

```
classdef Model<handle
    properties
        state
    end
    methods
        function mObj = createMemento(obj,ID)  % 构造备忘录对象
            mObj = memento(obj.state,ID);
        end
        function setMemento(obj,mObj)          % 恢复状态
           obj.state =  mObj.state;
        end
        function show(obj)
            sprintf('internal state = %s',obj.state)
        end
    end
end
```

类之间的协作如图17.32所示。

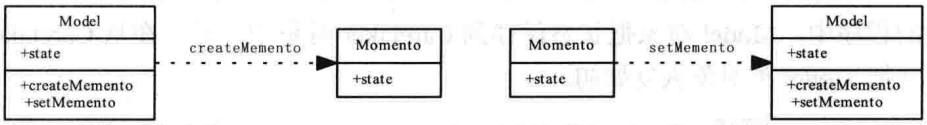


图 17.32 Model 负责构造 Memento 对象，还可以接受一个 Memento 对象来恢复自身的状态

- creatMemento 方法负责提取自身的状态，封装到一个 memento 对象中，并且返回新产生的对象。
- setMemento 方法接受一个 Memnto 对象，从中提取出状态信息，用来给自身赋值，以恢复到保存时的状态。

Memento

```
classdef memento < handle
    properties
        state
        ID
    end
    methods
        function obj = memento(state,ID)
            obj.state = state;
            obj.ID = ID ;
        end
```

```
    end
end
```

该 Momento 类用来封装 Model 中的状态信息，其中的 ID 用来标识 state，该 ID 由 createMemento 方法提供。Model 对象可以创建多个 Memento 对象，或者说 Model 对象可以保存多个自身的状态。

如图17.33所示，Caretaker 类是一个用来存储 Memento 的对象的容器，主要内部数据结构利用 MATLAB 容器 containers.Map 来实现，其 Key 是 ID，其 KeyValue 是 Memento 对象。

Caretaker

```
classdef Caretaker < handle
    properties
       states = containers.Map
    end
    methods
        function addState(obj,mObj)
            obj.states(mObj.ID) = mObj;
        end
        function mObj = getState(obj,ID)
            mObj = obj.states(ID);
        end
    end
end
```

下面的程序中，Model 对象把状态保存到 Caretaker 对象中，随后在从 Caretaker 中取出该状态，并把 Model 类对象恢复如初。

Script

```
1 modelObj = Model ;
2 modelObj.state = 'untouched';
3 modelObj.show()
4 holder = Caretaker();
5 mementoObj = modelObj.createMemento('original');
6 holder.addState(mementoObj);
7 modelObj.state = 'touched'; % 状态改变
8 modelObj.show()
9 modelObj.setMemento(holder.getState('original'))
10 modelObj.show()
```

Command Line

```
internal state = untouched

internal state = touched


internal state = untouched
```

- □ 脚本中第 2 行设置对象的初始状态是“untouched”。
- □ 第 6 行把这个状态记录到 Memento 对象中，并且设置状态 ID 为 orignal。
- □ 第 7 行把 Memento 保存到 Caretaker 对象中。
- □ 第 9 行改变 ModelObj 的状态。
- □ 第 12 行，在 Caretaker 中取出 ID 为“original”的 Memento，然后传回给 modelObj 对

象的 setMemento 方法，于是其状态恢复成“untouched”。

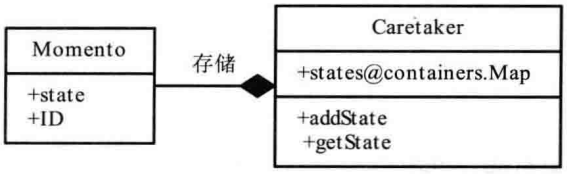


图 17.33 Caretaker 类是一个用来保存 Memento 的容器

17.6.2 应用：如何利用备忘录模式实现 GUI 的 do 和 undo 操作

这一节我们举一个具体的例子，使用前节的 Memento 框架，简单地演示如何通过保持 GUI 对象的内部状态来实现简单的 do 和 undo。GUI 设计如图17.34所示。

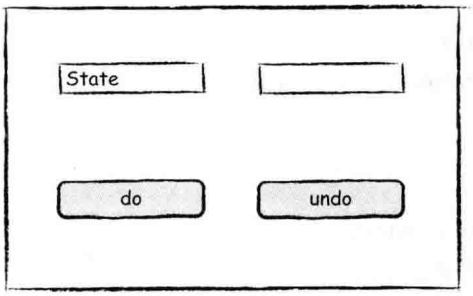


图 17.34 一个包含 do 和 undo 按钮的界面

该 GUI 中有一个文本框和两个按钮，用户可以在文本库中输入任意文本，表示一个状态，用户按下 do 键，GUI 将改变内部的状态值成为用户的输入，用户再按下 undo 键，则可以恢复到之前的状态。为简单起见，状态容器 CareHolder 仅存放一个状态。

1. Memento & Caretaker

Memento 的设计最简单，可略去 ID 仅仅封装一个变量：

```
classdef memento < handle
    properties
        state
    end
    methods
        function obj = memento(state)
            obj.state = state;
        end
    end
end
```

在实际 GUI 编程中，Caretaker 可能需要保存多个界面上用户的输入，所以 Caretaker 应该是一个全局类，并且会被各个 GUI 反复使用，这里把 Caretaker 设计成一个 Singleton 类，以保证其在整个程序中只存在一个对象。

```
classdef Caretaker < handle
    properties
       state
    end
    methods
        function addState(obj,mObj)
            obj.state = mObj;
        end
        function mObj = getState(obj)
            mObj = obj.state;
        end
    end
    methods(Access = private)
        function obj = Caretaker()
        end
    end
    methods(Static)
        function obj = getInstance()
           persistent localobj;
           if isempty(localobj) || ~isvalid(localobj)
               localobj = Caretaker();
           end
           obj = localobj;
        end
    end
end
```

2. Model

GUI 的设计沿用 MVC 模型，下面是 Model 类的代码：

```
classdef Model<handle
    properties
        state
    end
    events
        stateChanged
    end
    methods
        function obj = Model()
           obj.state = 'initial' ;
        end
        function mObj = createMemento(obj)
```

```
            mObj = memento(obj.state);
        end
        function setMemento(obj,mObj)
           obj.state =  mObj.state;
        end
        function show(obj)
            sprintf('internal state = %s',obj.state)
        end

        function do(obj,inputState)
           % save current state into memento
           holder = Caretaker.getInstance();
           holder.addState(obj.createMemento())
           obj.state = inputState ;
        end

        function undo(obj)
           % retrieve state info from holder
           holder = Caretaker.getInstance();
           obj.setMemento(holder.getState());
           obj.notify('stateChanged');
        end
    end
end
```

其中：

- do 方法是界面上 do 按钮的响应函数，其工作包括：
  - 使用 Caretaker 的静态方法得到单例 Caretaker 的全局对象句柄。
  - 把当前的状态保持到 Caretaker 对象中。
  - 更新自己的状态。
- undo 所做的方法与 do 恰好相反。
  - 使用 Caretaker 的静态方法得到单例 Caretaker 的全局对象句柄。
  - 从 Caretaker 内部得到上一个状态，使用 setMemento 恢复之前的状态。
  - 通知视图更新。

3. View

View 类的结构沿用 MVC 模式中的 View 类的基本结构：

```
classdef View < handle
    properties
        viewSize      ;
        hfig          ;
        doButton      ;
```

```
    undoButton    ;
    stateBox      ;
    text          ;
    modelObj      ;
    controlObj    ;
end
properties(Dependent)
   state
end
methods
    function obj = View(modelObj)
        obj.viewSize  =  [100,100,300,200];
        obj.modelObj = modelObj ;
        obj.modelObj.addlistener('stateChanged',@obj.updateState);
        obj.buildUI();
        obj.controlObj = obj.makeController();
        obj.attachToController(obj.controlObj);
    end
    function input = get.state(obj)
        input = get(obj.stateBox,'string');
        input = str2double(input);
    end
    function buildUI(obj)
       obj.hfig =  figure('pos',obj.viewSize);
       obj.doButton   = uicontrol('parent',obj.hfig,'string','do',
                                  'pos',[60 28 60 28]);
       obj.undoButton = uicontrol('parent',obj.hfig,'string','undo',
                                  'pos',[180 28 60 28]);
       obj.text       = uicontrol('parent',obj.hfig,'style','text','string',...
                                  'State','pos',[60 142 60 28]);
       obj.stateBox   = uicontrol('parent',obj.hfig,'style','edit',
                                  'pos',[180 142 60 28]);
       obj.updateState();
    end

    function updateState(obj,scr,data)
        set(obj.stateBox,'string',num2str(obj.modelObj.state));
    end

    function controlObj = makeController(obj)
        controlObj = Controller(obj,obj.modelObj);
    end
```

```
        function attachToController(obj,controller)
            funcH = @controller.callback_dobutton;
            set(obj.doButton,'callback',funcH);
            funcH = @controller.callback_undobutton;
            set(obj.undoButton,'callback',funcH);
        end
        function delete(obj)
           if ishandle(obj.hfig)
              close(obj.hfig);
           end
        end
    end
end
```

4. Controller

Controller 类沿用 MVC 模式中的 Controller 类的基本结构：

```
classdef Controller < handle
    properties
        viewObj  ;
        modelObj ;
    end
    methods
        function obj = Controller(viewObj,modelObj)
            obj.viewObj = viewObj;
            obj.modelObj = modelObj;
        end
        function callback_dobutton(obj,src,event)
            obj.modelObj.do(obj.viewObj.state);
        end
        function callback_undobutton(obj,src,event)
            obj.modelObj.undo();
        end
    end
end
```

5. main script

下面的脚本启动 GUI 窗口，读者可以自己验证 do 和 undo 按钮的效果。

```
close all ; clear all;
modelObj = Model();
viewObj = View(modelObj);
```

17.6.3 Memento 模式总结

《设计模式》一书中对 Memento 模式的意图是这样叙述的：

> 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样，以后可以将该对象复原到原先的状态。
>
> Design Patterns GOF

1. Memento 模式结构

Memento 模式的结构如图17.35所示。

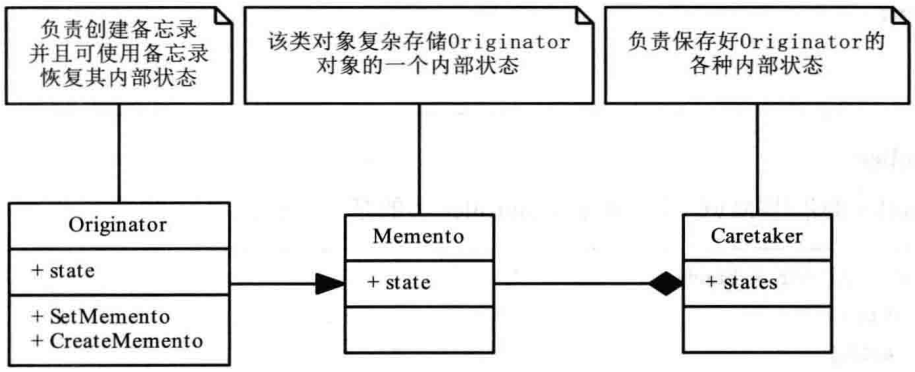


图 17.35 备忘录模式的 UML

2. 类之间的协作

- □ Originator 是拥有状态的对象，外部的命令向 Originator 发出一个请求，要求保存备忘录。
- □ Originator 把自身状态数据封装到一个 Memento 对象中，并且提交给 Caretaker 保存。
- □ 只有 Originator 知道该如何利用 Memento 对象中的数据；Caretaker 的工作仅仅是保存各个 Memento 对象，不能对备忘录的内容进行操作。

参考文献

[1] Erich Gamm, Richard Helm, Ralph Johnson, John Vlissides. 设计模式：可复用面向对象软件的基础 [M]. 北京：机械工业出版社, 2000.

[2] MathWorks. MATLAB® Object-Oriented Programming. MathWorks, 2011.

[3] 程杰. 大话设计模式 [M]. 北京：清华大学出版社, 2007.

[4] Alan Shalloway, James R Trott. 设计模式解析 [M]. 北京：人民邮电出版社, 2010.

[5] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First 设计模式 [M]. 北京：中国电力出版社, 2007.

[6] Bruce Eckel, Chuck Allison. C++ 编程思想：卷二实用编程技术 [M]. 北京：机械工业出版社, 2010.

[7] Stanley Lippman. 深度探索 C++ 对象模型 [M]. 武汉：华中科技大学出版社, 2001.

[8] Kent Beck. 以测试驱动的开发. Addison-Wesley, 2001.

附　　录

附录

附录 A　如何在 MATLAB IDE 中切换窗口

面向对象的编程方法提供了一种方式，把函数的处理和数据的存放分散到各个类当中去，随着程序变得越来越复杂，类的数量也将变得越来越多。因此，不可避免地，我们将要同时编辑多个文件、多个成员方法和多个类，在调试程序的同时，还要在不同的目录中切换，在不同的窗口中切换。一般情况下，可以使用鼠标进行操作，但是如果掌握了最常用的一些快捷键，将大大缩短这些操作的时间。这节将介绍最常用的几个快捷键。

1.　如何在 Editor 中的各个窗口之间切换

快捷键 `Ctrl+Pageup` 和 `Ctrl+Pagedown` 可以被用来在 Editor 内部各个打开的文件之间做切换，如图A.1所示。

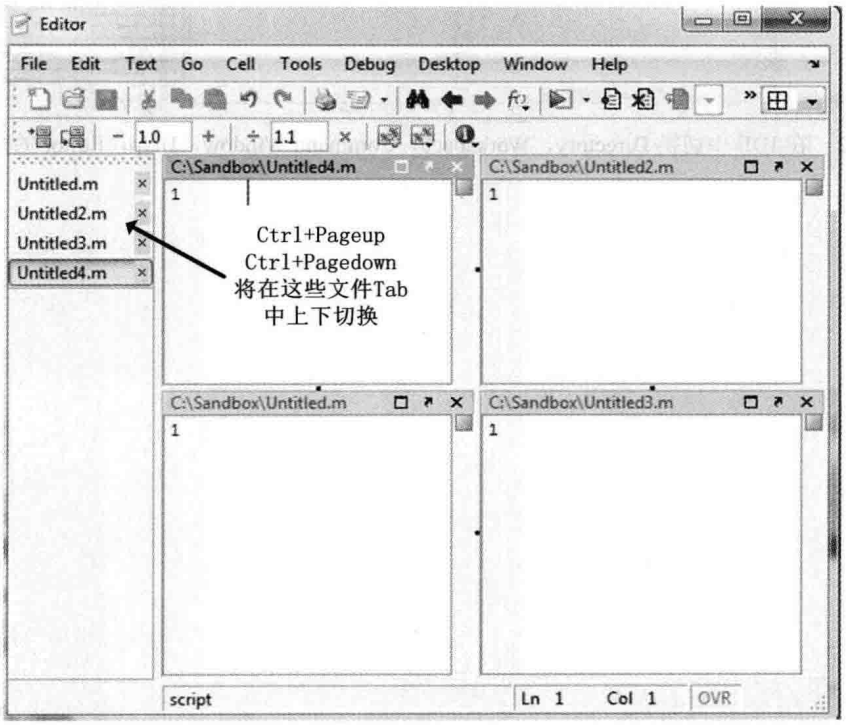


图 A.1　在 Editor 内部各个打开的文件之间进行切换的快捷键

2.　如何在 IDE 中的窗口之间切换

在 IDE 中切换 Directory，Workspace，Command Window，Help，Editor 的快捷键如图A.2所示。

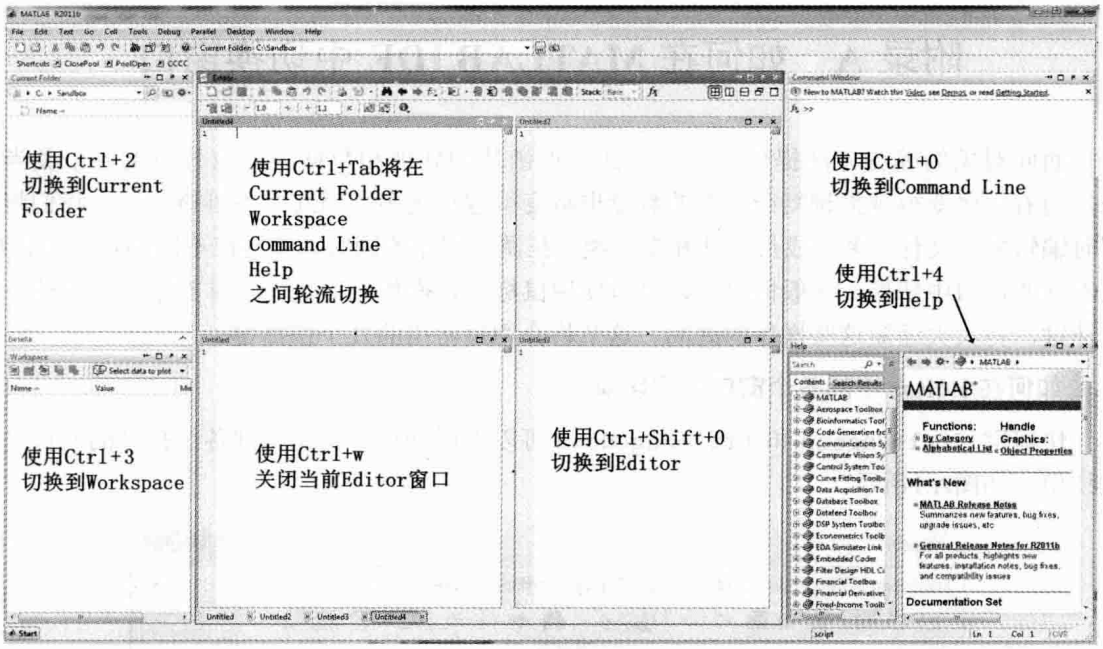


图 A.2 在 IDE 中切换 Directory，Workspace，Command Window，Help，Editor 的快捷键

附录 B　综合实例：如何把面向过程的程序转成面向对象的程序

使用面向对象编程的好处是，从长远的角度扩展和维护程序更方便，但是如何处理已经写好的面向过程的程序呢？这是一个常见的问题。这些已有的程序，可能是用户自己的，也可能是从网上下载的。如果要把所有的面向过程的程序重新写一遍，显然是不实际的，好在把面向过程的 MATLAB 程序改写成为面向对象的程序并不是一件麻烦的事情。这里，我们以一个实例介绍如何使一个面向过程的 MATLAB 程序通过简单修改变成面向对象的风格。我们假设，经过前面这么多章节的语法介绍，把一个函数和数据包装成一个类，对读者来说已经易如反掌，为了突出重点，我们仅使用 UML 来表示如何从大体上组织程序。

我们要改进的例子是一个图像处理程序，叫做 SIFT（图像局部特征描述算子），其大致算法是对输入的图像进行局部特征提取，然后对每一个局部特征进行编码，这些编码可以用来进行图像匹配，拼接和识别。SIFT 算法是一个流行的图像处理算法，在众多的 MATLAB 图像处理书籍中都被提到，也是比较典型的混合编程的例子。程序的作者用 MATLAB 作为主要的驱动，以及计算结果可视化的工具，遇到计算量大的图像运算，如果恰好 Base MATLAB[①]中没有提供内置的函数，就把该运算用 C 语言实现，然后编译成 .mex 文件，直接在 MATLAB 中调用。当然，读者并不需要太多图像处理的背景，只需要记住面向过程的基本特征，就像我们在第1.1节所说的那样，以函数为中心，用函数操纵数据，通过数据在多个过程直接传递共享来完成过程的模拟，函数和数据是分开的。而面向对象的特征是封装数据和算法，把任务分解成一个个相互独立的对象，通过各对象之间的组合和通信来完成任务。有兴趣的读者可以在如下地址下载到该程序：

```
http://www.vlfeat.org/~vedaldi/code/sift.html
```

我们集中讨论 demo 脚本文件和其主算法 sift 函数两个文件，以及如何把它们转换成面向对象的风格。当然，类的设计不是固定的，这里只提供一个大概的思路，希望能对读者有所启发。

1. demo 脚本

siftdemo.m

```
I1=imreadbw('data/img3.jpg') ;
I2=imreadbw('data/img5.jpg') ;

I1=I1-min(I1(:)) ;
I1=I1/max(I1(:)) ;
I2=I2-min(I2(:)) ;
I2=I2/max(I2(:)) ;
```

①MATLAB 各种专门的工具箱一般提供更广泛的内置函数。

我们从 demo 脚本文件开始，第 1 到 7 行是简单的读入两个需要匹配的图像文件，并且做归一化的处理。这里的图像文件即在函数之间传递的数据，所以首先设计一个 ImageClass 类，用来封装图像数据。该类还应该包括打开图像的功能。另外，第 4 到 7 行的归一化操作，可以放到一个 normalize 方法中，并且该方法可以在该图像的 Constructor 中被调用。综上，最初的 UML 如图B.1所示。

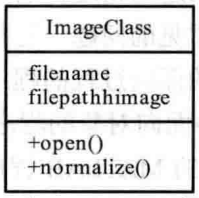


图 B.1 先用一个 ImageClass 来包装图像数据

可以预料，实际的图像匹配时，一定会遍历大量这样的图像文件。所以，可以采用第17.3节提到的 Aggregator 和遍历器的设计模式来对 sift 程序提供用于处理的图像文件，其 UML 如图B.2所示。

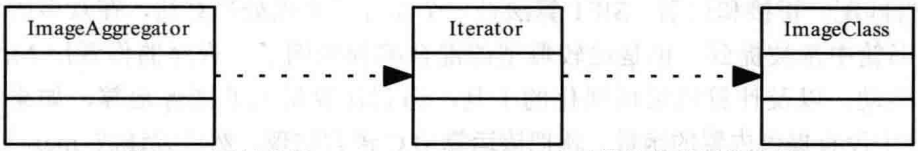


图 B.2 可以使用 Aggregator 和 Iterator 来对算法提供图像数据

下面继续分析 demo 文件：

siftdemo.m

```
9  fprintf('Computing frames and descriptors.\n') ;
10 [frames1,descr1,gss1,dogss1] = sift( I1, 'Verbosity', 1 ) ;
11 [frames2,descr2,gss2,dogss2] = sift( I2, 'Verbosity', 1 ) ;
12
13 figure(11) ; clf ; plotss(dogss1) ; colormap gray ;
14 figure(12) ; clf ; plotss(dogss2) ; colormap gray ;
15 drawnow ;
16
17 figure(2) ; clf ;
18 subplot(1,2,1) ; imagesc(I1) ; colormap gray ;
19 hold on ;
20 h=plotsiftframe( frames1 ) ; set(h,'LineWidth',2,'Color','g') ;
21 h=plotsiftframe( frames1 ) ; set(h,'LineWidth',1,'Color','k') ;
22
23 subplot(1,2,2) ; imagesc(I2) ; colormap gray ;
24 hold on ;
25 h=plotsiftframe( frames2 ) ; set(h,'LineWidth',2,'Color','g') ;
26 h=plotsiftframe( frames2 ) ; set(h,'LineWidth',1,'Color','k') ;
```

第 10，11 行调用的是该算法的核心函数 sift，但如何改成面向对象的风格，将在下节详述。这里，该函数接收图像矩阵 I 作为输入，如果改成面向对象的设计，该方法将接收

一个 ImageClass 的对象作为输入之一，函数的另一个参数是 Property-Value 对，该函数左边返回一系列计算结果 frames,descr,gss,dogss 根据第17.2节分离数据和算法的设计模式[①]，我们把图像数据和处理图像的算法分到两个类中去。而这些计算结果，显然和算法有更紧密的联系，它们应该都作为 SIFT 类中的属性（而不是 ImageClass 类中的属性）。对于形如 `'Verbosity' ,1` 的 Property-Value 对，我们将在后面讨论如何处理。初步的 UML 如图B.3所示。我们把和 sift 函数相对应的类方法叫做 action。

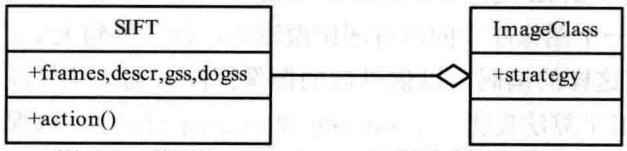


图 B.3　使用 Strategy Pattern 把算法和数据分离

如果将来还要添加其他（如图B.4）的特征匹配算法，我们可以抽象出一个基类来，而这个 action 方法在基类中就变成了 Abstract 方法。

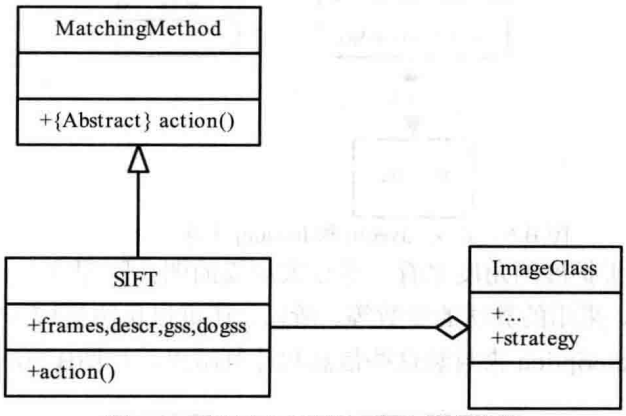


图 B.4　算法还可以进一步抽象出基类

在 demo 的第 13 到 26 行，用计算结果作图，显然，这些作图函数是和 SIFT 算法相关的，所以连同下面第 20 行的 plotsiftframe 函数，我们把它们封装到 SIFT 类去，如图B.5所示。

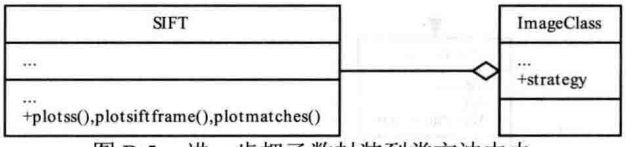


图 B.5　进一步把函数封装到类方法中去

siftdemo.m

```
28 fprintf('Computing matches.\n') ;
29 % By passing to integers we greatly enhance the matching speed (we use
30 % the scale factor 512 as Lowe's, but it could be greater without
31 % overflow)
32 descr1=uint8(512*descr1) ;
33 descr2=uint8(512*descr2) ;
34 tic ;
```

①这区别于简单的把数据和数据相关的算法封装到一个类中去的方法，原因见第17.2节。

```
35 matches=siftmatch( descr1, descr2 ) ;
36 fprintf('Matched in %.3f s\n', toc) ;
37
38 figure(3) ; clf ;
39 plotmatches(I1,I2,frames1(1:2,:),frames2(1:2,:),matches) ;
40 drawnow ;
```

第 32，33 行，其中 descr 是图像的局部特征编码，该编码是重要的计算结果。如前所述，可以预料，如果是对一个图像库中的所有图像做遍历，将会得到大量的这样的编码，应该有一个方法能我们保存这样的编码，以供以后的图像匹配之需（这样就不用每次做重复计算了）。所以，可以给 SIFT 算法提供一个 saveobj 和 loadobj 的方法，以保存计算结果，如图B.6所示。

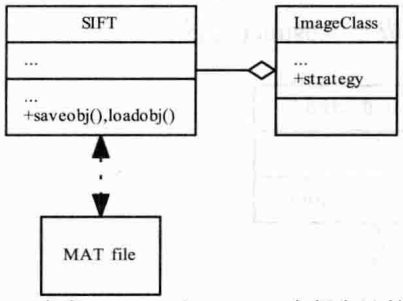


图 B.6 定义 saveobj 和 loadobj 来保存计算结果

从让程序可以便于扩展的角度来看，进行大规模的图像匹配肯定还需要比 descr 更多的信息，如图像的名称、采用的算法的参数等。所以，还可以仿照第17.6节图17.35所示的方法，构造一个特殊的类 Description 来封装这些信息和计算结果，如图B.7所示。

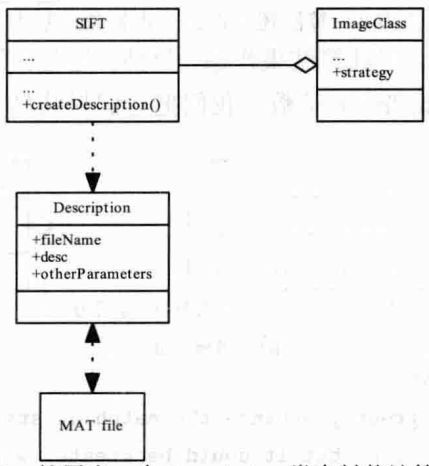


图 B.7 扩展出一个 Description 类来封装计算结果

第 35 行是对两个图像的特征码做匹配，第 39 行作图，siftmatch 做的是一个二元比较的操作，返回两者的比较结果 matches，可以把该函数用类方法进行包装，放到 SIFT 类中去。如果考虑特征提取是针对一个图像的操作，而特征匹配是针对两个 descr 的操作，还可以专门用类来包装这样的操作，设计一个类叫做 Compare，该类用来比较两个 Description 对

象，并且 siftmatch 是其中的一个方法。该方法返回结果 matches 可以作为该类的一个属性，plotmatches 也可以归到该类中，且 plotmatches 利用 descr 中的信息以及 matches 属性来作图，其 UML 如图B.8所示。

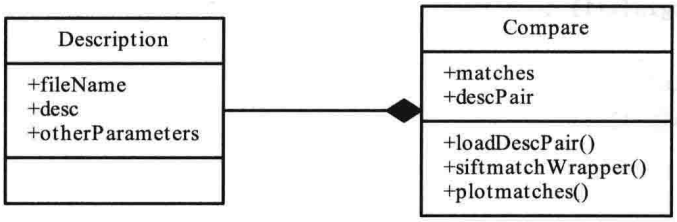


图 B.8　Compare 类用来封装二元操作

2. sift 函数

下面讨论核心算法 SIFT 函数，下面代码中的第 1 到 19 行是该函数的计算参数，这些参数可以作为 SIFT 类的属性，如图B.9，其中数值可以在 Property Block 中设置成 Default Value。

sift.m

```
1  function [frames,descriptors,gss,dogss]=sift(I,varargin)
2
3  if(nargin < 1)
4    error('At least one argument is required.') ;
5  end
6
7  [M,N,C] = size(I) ;
8
9  % Lowe's equivalents choices
10 S      =  3 ;
11 omin   = -1 ;
12 O      = floor(log2(min(M,N)))-omin-3 ; % up to 8x8 images
13 sigma0 = 1.6*2^(1/S) ;                  % smooth lev. -1 at 1.6
14 sigman = 0.5 ;
15 thresh = 0.04 / S / 2 ;
16 r      = 10 ;
17 NBP    = 4 ;
18 NBO    = 8 ;
19 magnif = 3.0 ;
20
21 % Parese input
22 compute_descriptor = 0 ;
23 discard_boundary_points = 1 ;
24 verb = 0 ;
25
26 for k=1:2:length(varargin)
27   switch lower(varargin{k})
28
```

```
    case 'numoctaves'
      O = varargin{k+1} ;

    case 'firstoctave'
      omin = varargin{k+1} ;

    case 'numlevels'
      S = varargin{k+1} ;

    case 'sigma0'
      sigma0 = varargin{k+1} ;

    case 'sigman'
      sigmaN = varargin{k+1} ;

    case 'threshold'
      thresh = varargin{k+1} ;

    case 'edgethreshold'
      r = varargin{k+1} ;

    case 'boundarypoint'
     discard_boundary_points = varargin{k+1} ;

    case 'numspatialbins'
      NBP = varargin{k+1} ;

    case 'numorientbins'
      NBO = varargin{k+1} ;

    case 'magnif'
      magnif = varargin{k+1} ;

    case 'verbosity'
     verb = varargin{k+1} ;

    otherwise
      error(['Unknown parameter ''' varargin{k} '''.']) ;
  end
end

% Arguments sanity check
if C > 1
```

```
72   error('I should be a grayscale image') ;
73 end
74
75 frames      = [] ;
76 descriptors = [] ;
```

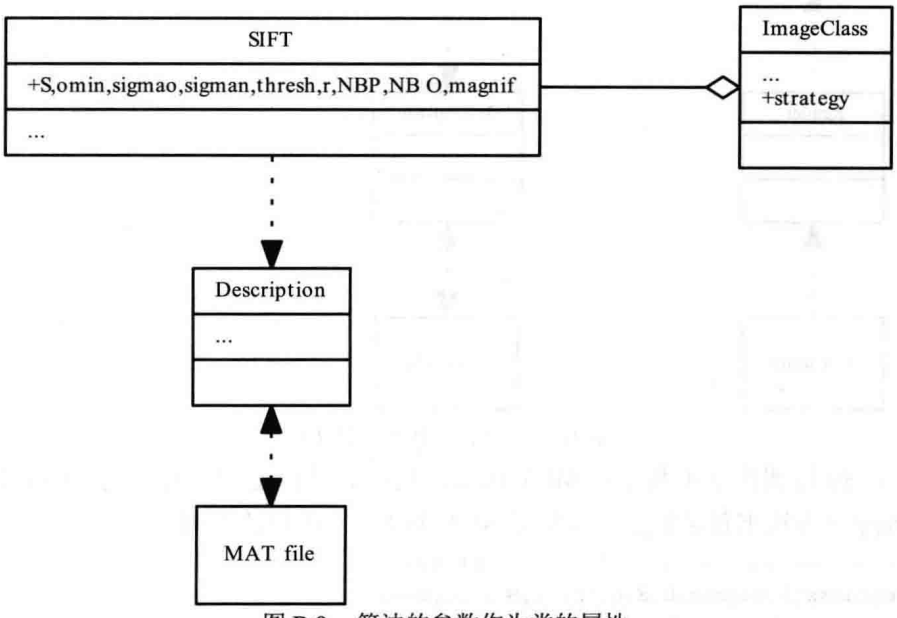


图 B.9　算法的参数作为类的属性

第 26 到 68 行用来接收外部提供的 PV 对作为算法的控制，可以预料的是，不同类型的图像，所用的算法参数不一样，实际工程计算中会有这样的需要，向程序提供各种不同的 PV 对来找到最优的参数，MATLAB 内部恰好提供了一个内置的 InputParser 类来负责参数的读入和验证，于是可以把参数的输入和验证从 SIFT 函数中分离出来。从扩展的角度，考虑到将来还可能对 SIFT 算法提出改进，我们可能会添加更多的计算参数，为了方便输入，可以统一地把参数用 .txt 文件或者 .xml 文件组织起来。这样一来，我们还要设计一个读文档的 reader 类，UML 如图B.10所示。

sift.m

```
78 % --------------------------------------------------------------------
79 %                                            SIFT Detector and Descriptor
80 % --------------------------------------------------------------------
81
82 % Compute scale spaces
83 if verb>0, fprintf('SIFT: computing scale space...') ; tic ; end
84
85 gss = gaussianss(I,sigman,0,S,omin,-1,S+1,sigma0) ;
86
87 if verb>0, fprintf('(%.3f s gss; ',toc) ; tic ; end
88
```

```
89 dogss = diffss(gss) ;
```

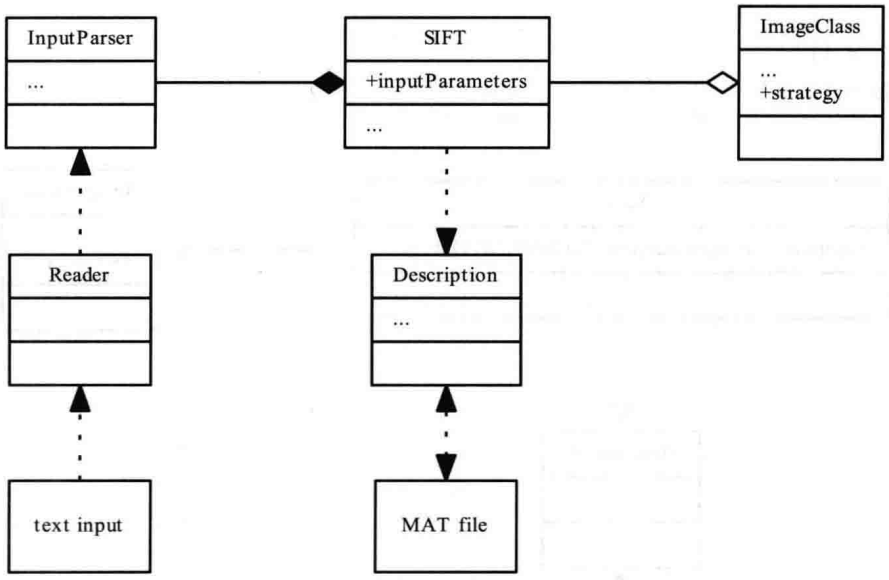


图 B.10 扩展程序的参数读入模块

第 85 和 89 行调用的函数都是 MEX 函数，其参数是固定的。对于这样的函数，可以设计一个 Wrapper 方法来包装它们，比如第 85 行对 MEX 函数的调用：

sift.m

```
1 gss = gaussianss(I,sigman,O,S,omin,-1,S+1,sigma0) ;
```

可以改写成对象方法的调用obj.gaussianssWrapper()：

sift.m

```
1 function  gaussianssWrapper(obj)   % 在类的方法内部调用 MEX 函数
2         obj.gss = gaussianss(obj.image,obj.sigman,obj.O,...
3               obj.S,obj.omin,-1,obj.S+1,obj.sigma0) ;
4 end
```

第 89 行对 MEX 函数的调用：

sift.m

```
1 dogss = diffss(gss) ;
```

可以改写成对象方法 diffsWrapper 的调用（其中 gss 是类中的属性）：

sift.m

```
1 function diffsWrapper(obj)
2         obj.dogss = diffs(obj.gss);
3 end
```

sift.m

```
91 if verb > 0, fprintf('%.3f s dogss) done\n',toc) ; end
92 if verb > 0
93   fprintf('SIFT scale space parameters [PropertyName in brackets]\n');
94   fprintf('  sigman [SigmaN]         : %f\n', sigman) ;
```

```
95   fprintf('  sigma0 [Sigma0]         : %f\n', dogss.sigma0) ;
96   fprintf('       O [NumOctaves]    : %d\n', dogss.O) ;
97   fprintf('       S [NumLevels]     : %d\n', dogss.S) ;
98   fprintf('    omin [FirstOctave]   : %d\n', dogss.omin) ;
99   fprintf('    smin                 : %d\n', dogss.smin) ;
100  fprintf('    smax                 : %d\n', dogss.smax) ;
101  fprintf('SIFT detector parameters\n')
102  fprintf('  thersh [Threshold]     : %e\n', thresh) ;
103  fprintf('       r [EdgeThreshold] : %.3f\n', r) ;
104  fprintf('SIFT descriptor parameters\n')
105  fprintf('  magnif [Magnif]        : %.3f\n', magnif) ;
106  fprintf('     NBP [NumSpatialBins]: %d\n', NBP) ;
107  fprintf('     NBO [NumOrientBins] : %d\n', NBO) ;
108 end
```

第 91 至 108 行是对计算中间结果的输出，可以设计一个 report 方法来向命令行输出这些结果。当然，如果是大批量的计算，我们希望这些结果可以被计算在 LOG 中，其设计模式可以参见第15.2节，UML 如图B.11所示。

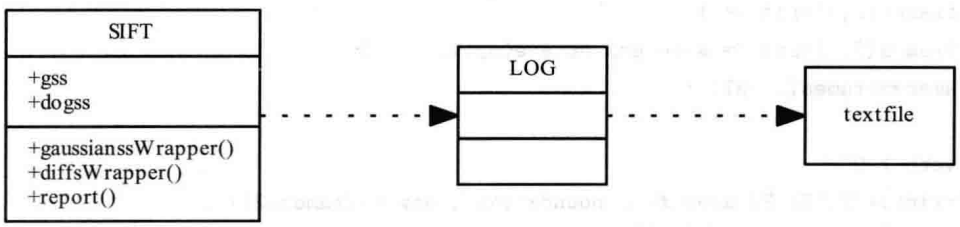


图 B.11　扩展出 LOG 类保存计算的中间输出

这样，SIFT 类中又增加了三个方法。

sift.m

```
110 for o=1:gss.O
111   if verb > 0
112     fprintf('SIFT: processing octave %d\n', o-1+omin) ;
113                 tic ;
114   end
115
116   % Local maxima of the DOG octave
117   % The 80% tricks discards early very weak points before refinement.
118   idx = siftlocalmax(  dogss.octave{o}, 0.8*thresh  ) ;
119   idx = [idx , siftlocalmax( - dogss.octave{o}, 0.8*thresh)] ;
```

第 110 至 119 还是调用 MEX 函数，即还是用 Wrapper 方法包装它们，UML 略。

sift.m

```
121   K=length(idx) ;
122   [i,j,s] = ind2sub( size( dogss.octave{o} ), idx ) ;
123   y=i-1 ;
```

```
124  x=j-1 ;
125  s=s-1+dogss.smin ;
126  oframes = [x(:)';y(:)';s(:)'] ;
127
128  if verb > 0
129    fprintf('SIFT: %d initial points (%.3f s)\n', ...
130      size(oframes, 2), toc) ;
131    tic ;
132  end
133
134  % Remove points too close to the boundary
135  if discard_boundary_points
136    % radius = maginf * sigma * NBP / 2
137    % sigma = sigma0 * 2^s/S
138
139    rad = magnif * gss.sigma0 * 2.^(oframes(3,:)/gss.S) * NBP / 2 ;
140    sel=find(...
141      oframes(1,:)-rad >= 1                       & ...
142      oframes(1,:)+rad <= size(gss.octave{o},2) & ...
143      oframes(2,:)-rad >= 1                       & ...
144      oframes(2,:)+rad <= size(gss.octave{o},1)     ) ;
145    oframes=oframes(:,sel) ;
146
147    if verb > 0
148      fprintf('SIFT: %d away from boundary\n', size(oframes,2)) ;
149      tic ;
150    end
151  end
152
153  % Refine the location, threshold strength and remove points on edges
154  oframes = siftrefinemx(...
155    oframes, ...
156    dogss.octave{o}, ...
157    dogss.smin, ...
158    thresh, ...
159    r) ;
160
161  if verb > 0
162    fprintf('SIFT: %d refined (%.3f s)\n', ...
163            size(oframes,2),toc) ;
164    tic ;
165  end
166
```

```
167   % Compute the oritentations
168   oframes = siftormx(...
169     oframes, ...
170     gss.octave{o}, ...
171     gss.S, ...
172     gss.smin, ...
173     gss.sigma0 ) ;
174
175   % Store frames
176   x     = 2^(o-1+gss.omin) * oframes(1,:) ;
177   y     = 2^(o-1+gss.omin) * oframes(2,:) ;
178   sigma = 2^(o-1+gss.omin) * gss.sigma0 * 2.^(oframes(3,:)/gss.S) ;
179   frames = [frames, [x(:)' ; y(:)' ; sigma(:)' ; oframes(4,:)] ] ;
180
181
182   % Descriptors
183   if nargout > 1
184     if verb > 0
185       fprintf('SIFT: computing descriptors...') ;
186       tic ;
187     end
188
189     sh = siftdescriptor(...
190       gss.octave{o}, ...
191       oframes, ...
192       gss.sigma0, ...
193       gss.S, ...
194       gss.smin, ...
195       'Magnif', magnif, ...
196       'NumSpatialBins', NBP, ...
197       'NumOrientBins', NBO) ;
198
199     descriptors = [descriptors, sh] ;
200
201     if verb > 0, fprintf('done (%.3f s)\n',toc) ; end
202   end
203 end
```

第 121 至 203 行是中间计算过程，类似地，我们对 MEX 函数使用 Wrapper 方法包装，输出使用 LOG 类把函数内部的重要数据定义成类的属性，以方便数据在函数之间的传递，不再赘述。

```
end
end
if perp > 0, fprintf( do
end
descriptors = [descriptors
'NumDistanceBins' NBD)
NumSpecialBins 'SB',...
'Magfit' magfit
Res auto,...
Se
n.s.sigma0,
sfreqs ...
Gee.orient...
sh = [sh;sin(pi*(
end
    tic;
    fprintf( str: commission measurement
if verb > 0
  if maxsc > 1
    Descr_Hour
    Images = [Images (ic) ( ) , algor
    Pyra = [Pyra.lum(j) * (e.sigma(i) *
    I    = [I;Pyra.sum * e.sigma(i)^2];
    r    = [r;Pyra.lum * e.sigma(i)];
  end
  % Se
    Pyra.lum(
    % Ge
    Tse
    Pyra.
    Pyra.lum(
    % Se
    r
    end
```

索　引

写在最后

本书的编写时间前后长达两年多。作者竭尽全力使它尽可能地通俗易懂又不失“深度”。在阅读完 MATLAB 中文论坛上的 25 位“试读版”志愿读者的反馈意见后，作者对本书又做了大量的修改。本书正式出版之后，也一定会有更多的读者提出宝贵的建议，我们将在之后的版本中加以完善。最后，再次感谢每一位对本书的编写和出版给予关心和帮助的人，同时也要感谢北京航空航天大学出版社的支持和编辑老师对作者的信任与指导。

特约策划：张延亮（math）
策划编辑：蔡　喆
封面设计：runsign

作者简介

徐　潇　软件工程师，物理博士，研究方向为电子结构计算、密度泛函算法开发；计算机硕士，研究方向为图像处理。在科研编程中经历了开发大型程序难以维护的困难，花了很多时间用于改进程序但总不尽如人意。从2009年接触并开始使用MATLAB面向对象编程，发觉工程进度被迅速加快，于是萌生了写一本介绍MATLAB面向对象编程的书的念头，希望把这个优秀的工具介绍给大家。

李　远　研究员，物理博士，研究方向为光电子器件；硕士就读于北京交通大学，后在Wake Forest University 取得物理博士学位，曾在华盛顿大学做博士后。希望这本《MATLAB面向对象编程——从入门到设计模式》能够给工程科学领域的工作者带来更多方便，尤其是在一些需要长期使用而且不断更新的程序上。

特别推荐

- MathWorks
- 迈斯沃克软件（北京）有限公司

特约技术支持

- MATLAB中文论坛（www.iLoveMatlab.cn）

上架建议：计算机软件

ISBN 978-7-5124-1609-3

9 787512 416093

定价：46.00元