Real World Instrumentation with Python

数据采集与控制系统自动化

真实世界的Python仪器监控

J.M.Hughes 著
OBP Group 译

O'REILLY®

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

O'REILLY®

真实世界的Python仪器监控

数据采集与控制系统自动化

Real World Instrumentation with Python

J. M. Hughes 著
OBP Group 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书主要探讨如何运用 Python 快速构建自动化仪器控制系统，帮助读者了解如何通过自行开发应用程序来监视或者控制仪器硬件。本书内容涵盖了从接线到建立接口，直到完成可用软件的整个过程。

本书适合需要进行仪表控制、机器人、数据采集、过程控制等相关工作的读者阅读参考。

版权贸易合同登记号　图字：01-2011-4233

图书在版编目（CIP）数据

真实世界的 Python 仪器监控：数据采集与控制系统自动化 /（美）休斯（Hughes,J.M.）著；OBP Group 译. —北京：电子工业出版社，2013.1

ISBN 978-7-121-18659-2

Ⅰ. ①真… Ⅱ. ①休… ②O… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字(2012)第 237794 号

策划编辑：张春雨
责任编辑：白　涛
封面设计：Karen Montgomery　张　健
印　　刷：北京丰源印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：37.5　字数：600 千字
印　　次：2013 年 1 月第 1 次印刷
定　　价：89.00 元

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

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

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”;创建第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了Make杂志,从而成为DIY革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖,共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版,在线服务或者面授课程,每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人,他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了:‘如果你在路上遇到岔路口,走小路(岔路)。’回顾过去Tim似乎每一次都选择了小路,而且有几次都是一闪即逝的机会,尽管大路也不错。”

——Linux Journal

译者序

Zoom.Quiet 代序

本书是“侠少”（张春雨编辑）推荐给我翻译的，只是因为之前组织过几本 Python 相关技术图书的翻译工程，就成为所谓资深人士，进而被编辑盯上了。可是拿到书一看，不是单纯讲 Python 开发技巧，而是如同 TBBT（《生活大爆炸》）片头曲背景 MV 那般，内容包罗万象：从电子到仪表到线缆再到高端的软件工程都有所包含，内容的推进也是高速但清晰的，而且完全是根据学习的自然路径组织的，一步一步，从简入繁，自然而然，明明白白，很有 Pythonic 的感觉！所以，俺就无耻地心动了。再说，华蟒用户组（CPyUG）订阅人数近一万，其中肯定有软硬兼修的高人，俺只要作好大妈的本职工作，就可以向中国 Python 社区贡献第一本硬件相关的好书了！于是，革命的乐观主义精神主导了俺的情绪。终于，在 2011 年春节前，俺接下了本书的翻译组织任务！

最终图书署名为 OBP Group（开放图书工作组），原因有三：

- 本书的翻译出版过程沿用了 OBP（Open Book Project，开放图书计划）的协同流程；
- 大家的主要协同场景是在 Google Group（邮件列表）中；
- 工作组专门为本书成立，翻译完成后就地解散，转为通过 Group 的形式存在，继续支持图书内容的讨论。

其中 OBP 的工作流程细节请参考：

http://code.google.com/p/openbookproject/wiki/HowToBuildBookOnline

目前，OBP 是作为一种开放的分布式协作流程而存在的。

- 任何人都可以使用 OBP 实践检验过的在线分布式团队的组织形式来完成任意作品。
- 其实就是将软件工程管理的思想及工具，组合应用在图书创译方面而已。
- 相比其他翻译团队，特殊在协同的工具链。

1. 在 Bitbucket.org 上使用 Mercurial 分布式仓库，或是在 Github/Gitcafe 中使用分布式仓库。
2. 基于 Sphinx 来组织新结构化文本（rST）。
3. 通过 readthedocs.org 随时编译为 html 格式的图书式网站。

果然，通过邮件列表，快速报名上来十多位有意向的译者。根据以往经验，三个和尚没水吃，人多时，大家都以为有人在翻译，自个儿就不动手了。于是，以 C 语言/Python 语言/硬件进行领域划分，俺单独负责所有 Python 及软件工程相关的内容，另外邀请四位译者分别承担其他两部分内容，再加上责任编辑，实际上形成了一个有专项目标的迷你社区。

那么什么是社区呢？

简单的说就是相对固定的成员在相对固定的场所对固定的目标进行固定的行动。所以，技术图书的翻译团队，如果是分布式的，其实就形成了一个确切的社区。只是，一般 OBP 专项社区随着图书的出版会快速转入静默，不是图书内容没有值得持续讨论的地方，而是大家不愿意回忆那些熬夜翻译赶进度的苦吧。

本书的专门列表也已经建立：rwipy-zh@googlegroups.com，欢迎大家订阅和加入讨论。

虽然原书有 600 多页，翻译为中文，也就 500 页的样子，其中还包含大段的代码，平均到每个人也就 100 多页，几万字而已。相比动辄几百万字的网络小说来，大家的工作量不算大。但是，技术文字的翻译，不是口水化的玄幻小说，至少要做到：

- 原文内容自个儿看得懂
- 翻译出来的中文自个儿看得明白
- 翻译出的文字任何人也看得明白

而且，近 20 个月，大家并不是全职翻译，都是从学习/工作/生活中挤出时间来义务翻译的。常常要花很大的精力，重新熟悉翻译过的内容再跟上作者思路，找到合适的表述方法来。这是个反复的，不断自我折腾的过程……所以，进度的控制就完全失控了，好在此失控是 K.K 所说的能自我调节的“失控”。有负责的催生婆——责任编辑一直在执行田间管理，译者们只要根据协同约定，调整好状态，按时完成章节的翻译，hg push 到封闭仓库中，所有相关成员就可以同步到变化继续了。

从意动到交付初版的这 20 个月里，个人来说发生了很多事儿，很 chaos。但是，能在社区还有编辑的支持下拿出成书来，真的是极大的安慰。只是，大家心里又随着不断的 review 中发现的各种小问题，而酝酿出怀疑是否将书中丰富的软硬件知识翻译到位了的惶恐。然而，只要有好书，值得翻译的书，硬着头皮也还是愿意继续惶恐的！

最后要强调的一点是，本书的 OBP Group 所有成员，从来没有坐在一起交流过，直到最后签约阶段，才在北京同其中一半的成员面对面吃了次准庆功宴。感谢网络以及开源工具的力量，更要感谢所有成员给力的大爱，使我们能又见到一本好书的面市！

以下是所有工作组成员的简述，再次感谢所有成员，如果内容方面有什么问题，责任肯定在我们这群自不量力的行者身上，欢迎大家通过任意方式告诉我们，我们会坚持修订书中的内容的！

Atommann <atommann@gmail.com>

开源电子硬件和自由软件爱好者。几年前得知外婆打电话不便，遂决定制作一部特殊的照片拨号电话机，经过不断自学和实践，最后做出 facephone 解决了问题，从此成为一名硬件 hacker。

浏览过图书的目录后，就发现这是一本难得的好书，它的内容和我的工作息息相关，于是决定加入翻译团队。我比较熟悉的是书中的电子学部分，于是一开始就着手相关的翻译，然而在翻译的过程中发现自己原本以为懂得的东西实际上很多都不懂，有时为了弄清楚一个知识点，竟然会费几天的时间去学习。这样做是因为我自己就是一个技术书籍的读者，如果翻译得不准确对读者是不负责的，对这本 O'Reilly 的图书更是不敢怠慢，我想要把自己翻译的部分翻译得尽量准确易懂。在边学边译的过程中，我重新学到很多知识，这是参与翻译过程的一个很大的收获。

由于是业余时间翻译，在时间上会很不够用。有时要等到老婆孩子睡觉之后才有时间坐在电脑前一边读一边译。有一段时间我感觉自己对翻译工作失去了信心，后来多亏“大妈”和张编辑的号召才又回到工作状态，完成了自己的翻译任务。书译完了，女儿也从几个月长到了两周岁。如果说要感谢，我要感谢我的女儿艾达和妻子小霞，很多本应要陪她们的周末和夜晚都被我用来翻译了。

Andy Shi <andy.shia@gmail.com>

我是一个码农，最早是在网上认识“侠少”的，后来在组织珠三角技术沙龙时和他有过一次合作。这次是作为“救火队员”被拉到这个团队的。

本书是关于使用 Python 开发现实世界中的仪器软件的，我对 Python 本身不算擅长，只是入门级别，好在自己翻译的这章涉及语言本身的内容很少，这也是我敢应承下来的原因之一。翻译所谓“信达雅”，达到任何一个都不容易，反复读，仔细揣摩作者的意思，然后翻成中文，又要反复读，反复改，让语言更加通顺，更加贴合作者的原意。此中辛苦，确实不经历过就没法体会。我是最后阶段才加入，所以对于团队感觉并不是很深切，

基本上也就和“大妈”、“侠少”有过交流。这种方式应该说有其优点，但是如何找到合适的人，如何协调好，还是件不容易的事情。

Grissiom <chaos.proton@gmail.com>

在SHLUG的邮件列表上看到“大妈”吼，然后发现这本书很有意思，并且自己也了解一部分相关知识，于是申请加入了翻译团队。

这本书的内容还是不错的，所以翻译的过程也是一个学习的过程，学习就有快乐。而且有时为了一句话反复思考半天，最后忽然想到“啊，原来可以这样翻译！”的时候更是快乐……其实个人一直很向往Github那种异步的工作方式，所以加入到OBP后还是很兴奋的。但是现实是残酷的，大家都分布在全国各地，时间和精力上并不完全一致，而且又都是利用空余时间做翻译，翻译又不是自己的专长。虽然大家都是怀了一颗奉献的心来的，但是有时毕竟会出现心有余而力不足的状况。所以整个翻译的进度就受到了影响。在此仅代表个人向编辑道个歉，感谢编辑长久以来的宽容和支持。OBP也可以总结这次合作的经验教训，为以后的项目积累经验。同时，在这本书的翻译，尤其是最后的审校过程中还是暴露出了现在传统出版行业和“互联网公司”型团队的衔接问题，包括版本控制流程、译文格式排版等。个人对出版行业了解不深，所以这个话题还是留给专业人士展开吧。

这次是我第一次参与翻译工作，出现各种疏漏在所难免，还请大家多多批评指正。

Kermit Mei<kermit.mei@gmail.com>
http://www.zeuux.com/home/kermit.mei

主要从事Linux下的嵌入式软件开发工作，4年工作经验。由于本人经验和时间所限，发布之际还是诚惶诚恐，担心工作中有不足和失误之处，万望大家发现问题多多批评，指正，谅解！

sa xiao <silasvenus@gmail.com>

远程松散的协作方式好处是很灵活，每个成员可以根据自己的情况自由安排工作时间，加上主导者“侠少”、“大妈”制订的比较完善的流程，使得这样的工作方式可行。但大家都是在有本职工作或者学业的情况下自觉自愿地工作，且没有硬性的外在约束，这也加大了协调的难度，延长了工期。一方面，工作规则和制度需要更完善；另一方面，随着更多项目的完成，成员的磨合度提升，效率自然会提高。

关于书的内容，切入点很好，翻译起来有一定的难度，其他的让市场说话吧。

孙伟 <sagasw@gmail.com>

学习使用 C 语言很多年以后，喜欢 Python 也有一两年了，有幸参与到本书的翻译当中，感觉是一个非常好的学习和锻炼。C 语言靠近底层系统，简洁直接，长久不衰；Python 语法清晰明快没什么花活：这两门语言都是我喜欢的。本书介绍用 Python 进行工业控制方面的编程，是 Python 书籍中比较少见的，感兴趣的朋友可以读读看。在我的博客 http://sunxiunan.com 上也有一些自己对编程方面的文章，欢迎大家阅读讨论。

第 4 章和第 5 章的翻译，断断续续花了我将近两个多月时间。虽然这两章不是很复杂，也没有什么生僻的内容，但是当自己把一句句英文翻译成中文的时候，我才发现这个事情不是想象中那么容易。有些句子要反复斟酌，有些术语英文好懂，翻译成中文后却有点怪怪的，需要多花时间找到恰当的表述方式，对我自己算是一个很好的锻炼。由于水平所限，想必还是有很多问题，欢迎大家指正。

xiaoma <cnxiaoma@gmail.com>:

开源软件爱好者和程序员。小学时在少科站老师的指导下组装过收音机，中学时开始接触 Apple II 电脑上的 Logo 编程，后逐渐走上了软件开发这条不归路。

这已经不是我第一次参与技术图书的翻译，虽然少了些许初夜般的兴奋，但仍然保持着诚惶诚恐的态度，唯恐糟蹋了这一本好书。由于本人才疏学浅，因此难免挂一漏万，不足之处请大家多多指正。想到本书的出版，能为读者提供学习和工作上的帮助，所有的吐糟都化为了青烟。在此要感谢“大妈”的奔走张罗和译者、编辑们的辛苦工作，使得本书的中译本能顺利出版。

Zoom.Quiet <zoom.quiet@gmail.com>:

Python 中文社区创始人 / 管理员之一，热心于 Python 社区的公益事业，大家熟知的社区“大妈”，OBP 及蟒营工程设计者 / 主持人，参与 / 组织 / 主持各种线上 / 线下活动，主持编撰了《可爱的 Python》，坚持用 Pythonic 感化国人进入 FLOSS 社区进行学习 / 分享 / 创造！

本书是从纯软件跨入软硬天师的最好引导，强烈建议准备或是已经在玩开源软件的行者置备！

最后，公开一下 OBP 本次项目的详细进展记录：

http://code.google.com/p/openbookproject/wiki/RwIwPyZhLog

前言

本书介绍自动化仪器及其自动化控制。我们将探讨如何运用 Python 语言快速轻巧地构建自动化仪器的控制系统。

从研究实验室到工业厂房，自动化仪器无所不在。一旦人们意识到收集随时间变迁的数据很有用，自然就需要某种手段来捕捉并完成数据记录。当然，人们可以取叠纸拿个时钟，盯着温度计、刻度盘或是其他仪表，定期记录数值，但是很快就会受不了这种乏味的工作。如果这一记录过程可以自动化，无疑将更加可靠和易行。幸运的是，技术的进步早已超越了手写日志及发条驱动的带状图记录的时代！

如今，人们可以购买各种便宜的物理仪器并使用计算机来获取数据。一旦计算机被连接到仪器，数据收集、分析和控制等等功能就可以自由扩展，唯一受限的只是实现者自身的创造力了。

本书的主要目的是向读者展示如何创建一个有能力同用户友好交互的仪器或控制应用程序软件，并使用最低成本运行起来。为此，我们仅基于最必需的步骤来创建程序，包括怎么使用不同类型的输入 / 输出硬件接入现实世界的底层接口。我们也将研究一些行之有效的方法，用以指导创建强大且可靠的程序。特别提醒，应该为数据处理所必需的算法支付设计费用。最终，我们将体验如何为用户设计命令的输入以及结果展示。如果读者能从本书中发现一些想法，并创造性地运用在各种仪器设备上，满足自己的需要，那么我的愿望也就达成了，善哉。

本书的目标读者

本书专为需要或是自制仪器控制器（也称为数据采集和控制系统）的人准备的。你可能是名研究员、软件开发者、学生、项目主管、工程师，或一个业余爱好者。想实现的应

用系统，可能只是在实验过程中需要的自动化电子测试系统，或是其他类型的自动化设备。

本书要完成的目标软件将是跨平台的。我假定你至少在 Windows 平台特别是 XP 平台玩得很顺。而我会使用 Ubuntu 发行版本的 Linux 系统，不过书中讨论的程序将在各种兼容发行版中良好运行，同时我也假定你知道如何使用 csh 或是 bash 命令行脚本。

由于本书是关于如何通过物理硬件同现实世界交互的，其中自然涉及了一系列相关电气产品。但是，并不要求读者是名有足够背景知识的电气工程师。在第 2 章，包含了基本电子理论知识的介绍，虽然事实上不必理解深层次的电子学知识也可以令计算机与现实世界交互，不过，知道多点相关领域知识绝对没坏处，万一首次遇到意外，我们可以从中获得思路。

不论读者的工作类型或场所怎样，最关键的，我假定你需要通过某些硬件接口捕获一些数据，或是产生控制信号。更加重要的是，需要轻便又精确且可靠地构建出这些仪器的控制软件来。

本书所用编程语言

我们将使用 Python 作为主要的编程语言，仅仅嵌入一点点的 C 程序。在本书中，我将假定你有一些编程经验，并对 Python 或 C（理想情况下，两者都）熟悉。如果不是这样，有 Perl 或 Tcl/Tk 或如 Matlab 或 IDL 分析工具的经验，也是一个合理的起点。

本书坚定地回避 Python 语言更深奥的知识，配合大量的实例代码、图表注释和截屏来引导理解。对 C 涉及得很少，只用来说明如何创建和使用 Python 应用的底层系统扩展。第 3 章覆盖了 Python 语言的基础介绍，第 4 章介绍了 C 语言的基础知识，对以上语言进一步的探究可通过阅读建议自行学习。

为什么选用 Python

Python 是 Guido van Rossum 在 80 年代末开发的解释型语言。因其是种即时编译的脚本语言，故而用户可以在 Python 命令行环境中直接创建并执行。语言本身很容易学习和理解，只要一开始别理会过多的高级功能（装饰器，自省，列表推导，等等）就行。因此，Python 提供了快速构建原型及易懂的双重好处，这反过来又有利于快速为不同的设备创建各种不同应用，没有开发者通常需要应对的学习曲线以及传统的编译语言依赖的特定供应商提供的编程环境。

Python 是高度可移植的，几乎运行在所有现代计算平台中。在项目中坚持只使用常用的

接口方法，应用程序就很可能在安装 Windows 的 PC 中编写，但是不用修改一行代码也可以在 Linux 操作系统中运行良好。甚至于可以在 Sun 的 Solaris 机器和 Apple 的 OS X 系统中运行，即使书中没有特意提及这一点。一旦 Python 必须配合特定平台的特定扩展或驱动程序，便失去了可移植性，所以在这些情况下，我将提供分别适用于 Windows 和 Linux 的替代品。

本书包括了完整的可用示例代码，并配合框图和流程图来说明关键点，操作一些现成的、低成本的接口硬件。

系统

我们将探究的是那种既可以用在实验室，也可以直接用于工业环境的仪器设备。比如说，应用于电子实验室、风洞中的设备，或是进行气象数据收集的设备。而所说的系统，可能只是一个简单的温度记录仪，也可能是个复杂的真空控制系统。

一般来说，本书描述的技术可以作用于任何可以连接到 PC 的硬件，当然总是有些设备是使用封闭标准的特殊硬件，但是我们不会处理这些，也不会深入到复杂的数据处理领域，比如说：炼油厂的自动工程方案，核电厂，或机器人飞船。系统在这些领域通常是配合同样精密和复杂的软件，并通过专用硬件来实现极其复杂的控制。我们只关注最通常的设备、驱动和接口，以及使用一些通用界面方法轻松构建出可用的系统。

方法论

我们通过现实世界的实例一步步地理解如何定义设备应用，选择合理的接口以及硬件，建立可能需要的底层驱动以便配合 Python 接口与完成硬件控制。我们还将探索 TkInter 和 wxPython 的图形界面，以及 curses 的图形化文字界面。

本书包含的主要内容如下。

- 如何封装一个硬件供应商的 DLL 驱动，以便 Python 扩展使用。
- 如何与以 USB 为基础的 I/O 设备通信。
- 如何使用类似 RS-232 和 RS-485 或是 GPIB 工业标准接口。
- 追加上一个什么样的硬件类型才可能发现并使用接口。

本书还提供了参考，可以索引到现成的开源工具和库，以便即使从零开始，也可以用最短的时间完成一个可用的硬件控制系统。

本书的内容组织

本书分为 14 章和 2 节附录。第 14 章将开头 12 章的所有实践集中应用为一系列现实世界的例子。第 1 ~ 6 章引入了基础概念，读者可以选择跳过。

每章重点内容主要如下。

第 1 章 仪器介绍

从整体上来说什么是仪器，如何控制系统的工作，以及这些概念如何在实际世界使用。涵盖的例子包含自动户外灯，电器仪表在工程中的测试，在实验室中控制化学过程和热度批处理。

第 2 章 基本电子学

作为手册书，必须对物理硬件接口以及如何完成一个自动化工程从整体上涉及的各个方面有描述。本章对电子以及电气产品进行了简介，然后探讨了内置功能模块以及数字控制接口、模拟接口、计数器和计时器。最后，介绍并评论了作为幕后技术的串行和并行接口。如果你已经熟悉电子电路原理和装置，可以跳过本章。不过，建议至少要关注一下，以便为今后可能的利用留个印象。

第 3 章 Python 编程语言

这本书不是 Python 教程，本章提供了 Python 的基本知识以及核心概念，以便读者快速上手将 Python 最常用的功能在本书讨论情景中应用起来。本章还提供了一系列工具的概述，以便协助大家更加轻松地进行编程。

第 4 章 C 语言

本章从较宏观的层次介绍了 C 语言知识。目标是提供足够的资料，以便理解书中实例代码，并不会引入 C 语言神秘的细节。幸运的是，C 其实是相对简单的语言，而在这一章的信息应足以让你开始创建自己的驱动扩展以供给 Python 脚本使用。

第 5 章 Python 扩展

本章介绍了 Python 扩展是如何创建的，以及通常有哪些种类的扩展。提供的例子，无论是在本章，或是在后面的章节，都可以作为你自己应用的模板起点。

第 6 章 硬件：工具与耗材

虽然读者可能从来没有触摸过仪器设备以及电烙铁，但是很有可能用过螺丝刀、钢丝钳和数字万用表（DMM），这也足够开始了。在这一章我提供了一个清单，来说明开发仪表系统需要什么样的基本工具，以便可以按图索骥填充到书架上一个小盒子里，为你将来可能真的动手创造自动系统时使用。在最后，充分讨论了两件可以帮助你免除猜测之苦快速定位问题的设备：示波器和逻辑分析仪。本章还就应该准

备哪些类型的工具提供了一些可行性建议，以便读者参考决定购置或是升级。

第 7 章 物理接口

列出了一系列用 Python 进行数据采集或是控制时最常见的物理接口。RS-232 和 RS-485，就仪器控制而言，这就是最常见的两个串行接口。本章还涵盖了一些讨论场景中会遇见的 USB 和 GPIB/IEEE-488 接口的基础知识。最后，我们将关注 PC 的可插入式 I/O 硬件，即通常的 PCI 型电路板，以及通常可从硬件供应商得到支持的 API 规范。

第 8 章 入门

本章描述了一个对软件开发行之有效的过程。将这一内容设置在这里是因为，不论用何种语言来开发自控系统，至关重要的是计划，明确核心功能，比照预期数据来观察测试结果，并持续改进这一过程。通过开发 Python 扩展，我们将击穿模糊和不确定之门，通过设备控制软件进入真实世界。

第 9 章 控制系统概念

要对现实世界进行数据采集和控制，必须至少掌握部分控制和系统论。本章扩展了在第 1 章中介绍的控制系统概念，详细介绍了检查与共同控制系统概念和模型，包括了如反馈、“Bang-Bang”控制器和比例积微分（PID）控制等议题。还提供了一个基本控制系统案例分析，并给出了如何选择合适模型的一些指导方针。最后，介绍如何应用数学控制系统转化为实际的 Python 代码。

第 10 章 建立和使用模拟器

建立和使用模拟器，可以帮助我们加快开发过程，通过提供一个可以安全试错的环境来检验思路，不仅针对控制软件，还可以针对可模拟的硬件提供一些宝贵的经验（非侵入式的）。无论是因为目标硬件暂时未能提供，或太昂贵无法承担损坏的风险，都可以通过模拟器简便地令软件可以运行，进行测试改进，获得足够的信心，令它正常在现实世界中工作。

第 11 章 设备 I/O

在这一章中，我们将看看如何使用在第 7 章中介绍的物理接口，在真实世界和你的应用间搬运数据。我们将从接口协议格式开始讨论一些基本概念，然后引入一些软件实例以介绍几个常用协议解析包：pySerial，pyParallel 和 PyVISA。最后，会展示一些技术来说明如何读取和写入数据到设备。我们将对比阻塞与非阻塞的 I/O 事件，以及如何应对潜在的数据 I/O 错误，以便使应用程序更加健壮。

第 12 章 文件读 / 写

本章谈及如何实施对设备文件的读 / 写以及审查，并对比了文件格式，从普通的

ASCII 和 CSV 文件到二进制文件数据库都进行了简述。我们还将研究 Python 对配置文件的处理能力，看看它通过库方法有多么容易存储和检索配置参数。

第 13 章 用户界面

除非应用程序是嵌入硬件或是专门设计作为后台运行过程的，否则可能都会需要某种用户界面。本章考察命令行界面，或是用 Python 的 curses 模块生成的文字控制界面，以及如何使用 ANSI 的终端仿真器程序显示数据，接受输入。本章还包含如何用 Python 内置的 TkInter 模块来生成图形界面，并另外阐述了 wxPython GUI 包。

第 14 章 真实实例

我们将考察几种不同类型的设备如何进行数据采集和控制。本章首先是一个捕捉连续的数据输出的数字万用表实例。然后，考察一个采用普通的串行接口进行命令和数据交换的数据采集装置。最后，我们将详细分析硬件供应商提供的一个 API 接口，通过 DLL 对 USB 设备进行数据 I/O 操作，借鉴较早章节的例子对常见设备完成几乎所有的关键操作，以展示如何将理论付诸实践。

最后是两个附录，包含了其他实用信息：

附录 A 自由及开源软件资源

附录 B 仪器设备资源

本书的格式

本书使用下列排版约定。

斜体 *Italic*

用来表示新术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体 `Constant width`

用来表示计算机代码片段，或者在文中引用 Python 模块，以及变量名、函式名、数据类型、语句、关键字等程序元素。

等宽加粗字体 **`Constant width bold`**

用来表示命令或者需要由用户输入的文本。

等宽斜体 *`Constant width italic`*

用来表示应该由用户提供或特定语境确定的值替换的文本。

这个图标表示提示、建议或一般说明。

这个图标表示警告。

中文版书中切口处的“▭”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引所列页码为原英文版页码。

代码示例的使用

本书的宗旨就是帮助你完成工作。一般而言，除非要原样引用大量代码，否则你可以在自己的程序和文档中随意使用书中的代码，而无须与我们联系以取得授权。例如，在编写程序时引用本书若干代码片段是无须授权的。然而销售或分发 O'Reilly 图书示例光盘则是需要授权的。引用本书内容以及代码来答疑解难是不需要授权的，但是将书中的代码大量引入到你的产品和文档中则需要授权。

如果你在引用书中内容时注明出处，我们将不胜感激，虽然这不是必需的。引用声明通常包含标题、作者、出版商和 ISBN 编码。例如：“*Real World Instrumentation with Python* by J.M. Hughes. Copyright 2011 John M. Hughes, 978-0-596-80956-0.”

如果你对书中代码的使用不在上述范围之列，敬请通过 *permissions@oreilly.com* 与我们联系。

Safari® Books Online

Safari Books Online 是一个按需出版数字图书馆，在这里可以轻松搜索到超过 7500 种技术及创意类参考书和视频，快速得到要找的答案。

通过订阅，可以从这一在线图书馆浏览任何网页，观看任何视频；可以在手机等移动设备上阅读书籍；可以在产品付印前获得新书信息，优先一睹创作中的手稿并给作者提交反馈；可以复制并粘贴代码示例，组织收藏夹，下载章节，对重点部分添加书签，创建笔记，打印页面，以及从许多其他的省时功能中受益。

O'Reilly Media 已经将本书英文版上传至 Safari Books Online 服务。欲获得本书（英文版）电子版，以及 O'Reilly 和其他出版商的类似主题电子书的完全访问权，请在 *http://my.safaribooksonline.com* 免费注册。

联系我们

对于本书的评论或问题请联系出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们为本书制作了一个 Web 页面，页面中包含了简介、样章，以及其他信息。可以从这里访问这个页面：

http://www.oreilly.com/catalog/9780596809560
http://www.oreilly.com.cn

如果要留言或者提交关于本书的技术问题的反馈，请发邮件至：

bookquestions@oreilly.com

本书的更多信息、资源、参考文献和新闻，请登录出版社官网：*http://www.oreilly.com* 或者 *http://www.oreilly.com.cn/*。

Facebook：*http://facebook.com/oreilly*
Twitter：*http://twitter.com/oreillymedia*
YouTube：*http://www.youtube.com/oreillymedia*

致谢

感谢那些在本书成书过程中给予我帮助的人们。我的妻子 Carol 和女儿 Seren，对于我常常一头扎在办公室，不能花时间陪伴她们，付出了很大的耐心和理解。我的朋友，同时也是同事，Michael North-Morris，总是那么乐观。本书责任编辑 Julie Steele，是她给了我为 O'Reilly 写书的机会。Rachel Head，一个勤奋的文字编辑，可以忍受我滥用英语。此外还有很多 O'Reilly 工作人员向我提供了友好帮助。

我还要感谢 LabJack 公司提供真实硬件供我使用，并且不惜花费时间提供支持，确保我能正常工作。也要感谢 Agilent 公司的 Janet Smith 向我提供其产品的优质照片。

—John Hughes

目录

第 1 章 仪器学概论 1

数据采集 2
控制输出 4
开环控制 5
闭环控制 6
顺序控制 8
应用概观 9
电子测试仪器 9
实验室仪器 11
过程控制 12
小结 14

第 2 章 基本电子学 15

电荷 15
电流 17
基础电路理论 18
电路原理图 20
直流电路特性 23
欧姆定律 24
电流吸入与电流输出 26
再谈电阻 27

交流电路 28
正弦波 29
电容器 30
电感器 34
其他波形：方波、斜波、三角波和脉冲 37
接口 38
离散数字 I/O 38
模拟 I/O 42
计数器与定时器 46
脉宽调制 48
串行 I/O 49
并行 I/O 51
小结 53
推荐阅读 54

第 3 章 Python 编程语言 55

安装 Python 56
Python 编程 57
Python 的命令行 57
命令行参数和环境 58
Python 中的对象 59
Python 中的数据类型 60
表达式 73
操作符 73
语句 79
字串 86
程序组织 91
模块导入 101
加载并运行 Python 程序 104
基础输入输出 106
提示和技巧 110
Python 开发工具 112
编辑器和 IDE 112

调试器 115
小结 115
推荐阅读 115

第 4 章 C 语言编程 117

安装 C 语言编程环境 117
使用 C 语言开发软件 118
一个简单的 C 程序 119
预处理指令 122
标准数据类型 126
用户定义类型 127
操作符 127
表达式 136
语句 136
数组和指针 143
结构 146
函式 150
标准库 151
编译 C 程序 152
C 语言综述 156
C 开发工具 156
小结 157
推荐阅读 157

第 5 章 Python 扩展 159

用 C 建立 Python 扩展 160
Python 的 C 扩展 API 161
扩展代码的模块组织 161
Python API 类型和函数 163
方法表 163
方法标记 164
传递数据 165
使用 Python 的 C 扩展 API 167

通用离散 I/O API 167
通用包装器示例 169
调用扩展 173
Python 的 ctypes 外部函数库 177
用 ctypes 载入外部 DLL 177
ctypes 中的基本数据类型 178
使用 ctypes 179
小结 179
推荐阅读 180

第 6 章 硬件：工具与耗材 181

必备工具 181
手工工具 182
数字万用表 184
焊接工具 187
最好能有的工具 189
高级工具 190
示波器 190
逻辑分析仪 192
测试设备注意事项 194
耗材 194
全新和二手 195
小结 196
推荐阅读 196

第 7 章 物理接口 197

连接器 197
DB 型连接器 198
USB 连接器 201
圆形连接器 202
接线端子 203
接线 205
连接器失效 207

串行接口 208
RS-232/EIA-232 209
RS-485/EIA-485 215
USB 220
Windows 虚拟串口 224
GPIB/IEEE-488 226
GPIB/IEEE-488 信号 226
GPIB 连接 228
GPIB 转接 USB 229
PC 总线接口设备 230
基于总线接口的优缺点 230
数据采集卡 232
GPIB 接口卡 232
旧并不代表差 233
小结 234
推荐阅读 234

第 8 章 开始干吧 235

项目定义 236
需求驱动的设计 236
从需求开始 237
工程目标 238
需求 239
为什么需要需求 240
良好的需求 241
全景 242
需求类型 242
用例 244
可追溯性 246
需求捕获 248
设计软件 248
软件设计说明 249
SDD 的图景 249

伪代码 253
分而治之 253
处理错误和故障 255
功能测试 256
为需求而测 257
测试用例 257
测试错误处理 260
回归测试 261
进展追踪 261
实施 262
代码风格 262
组织你的代码 264
代码复查 265
单元测试 268
连接到硬件 277
软件文档化 278
版本控制 281
缺陷跟踪 281
用户文档 282
小结 283
推荐阅读 283

第 9 章 控制系统概念 285

基础控制系统理论 286
线性控制系统 286
非线性控制系统 288
顺序控制系统 289
术语和符号 290
控制系统框图 292
传递函数 293
时间和频率 293
控制系统类型 298
开环控制 299

闭环控制 299
非线性控制：继电器控制器 306
顺序控制系统 308
比例、比例积分、比例积分微分控制 312
混合控制系统 317
用 Python 实现控制系统 318
线性比例控制器 318
开关式控制器 319
简单 PID 控制器 320
小结 324
推荐阅读 324

第 10 章 构建并使用仿真器 327

什么是仿真 328
低保真和高保真 329
模拟错误和故障 330
使用 Python 创建一个仿真器 333
程序包和模块的组织 334
数据输入 / 输出仿真器 334
交流电源控制器仿真 349
串行终端仿真器 358
使用终端仿真器脚本 359
显示仿真数据 361
gnuplot 361
使用 gnuplot 363
使用 gnuplot 将仿真器数据图表化 366
创建你自己的仿真器 369
确认仿真器的必要性 369
仿真的范围 370
时间和精力 371
小结 371
推荐阅读 371

第 11 章 仪器数据 I/O 373

数据 I/O：接口软件 373
接口格式与协议 374
Python 接口支持的工具包 383
Windows 平台上的替代品 389
在 Linux 下使用基于总线的硬件 I/O 设备 389
数据 I/O：数据采集与写入 391
基本数据 I/O 391
阻塞和非阻塞调用 398
数据 I/O 方法 399
数据 I/O 错误处理 402
处理不一致的数据 407
小结 411
推荐阅读 412

第 12 章 读写数据文件 413

ASCII 数据文件 414
原始的 ASCII 字符集 414
Python 的 ASCII 字符操作方法 416
读写 ASCII 平面文件 418
配置数据 425
AutoConvert.py 模块——自动转换字符串 427
FileUtils.py 模块——ASCII 数据文件 I/O 工具 430
二进制数据文件 440
平面二进制数据文件 440
用 Python 处理二进制数据 442
图像数据 453
小结 462
推荐阅读 462

第 13 章 用户界面 465

文本界面 465

控制台 465
ANSI 显示控制台技术 478
Python 和 curses 494
用不用 curse 是个问题吗 502
图形用户界面 502
图形用户界面的历史和概念 503
在 Python 中使用 GUI 504
TkInter 508
wxPython 514
小结 522
推荐阅读 523

第 14 章 实例 525

串行接口 525
简易 DMM 数据获取 526
串行接口的离散或模拟数据 I/O 设备 531
串行接口及对速度的考虑 535
USB 实例：LabJack U3 536
LabJack 连接 537
安装 LabJack 设备 538
LabJack 与 Python 539
小结 546
推荐阅读 547

附录 A 自由和开源软件资源 549

附录 B 仪器资源 553

索引 557

第 1 章

仪器学概论

无论现代科学和技术还有多少潜力没有发挥出来，至少它教会了人类一课：没有什么事情是不可能的。

—— Lewis Mumford，*Technics and Civilization*，1934

仪器学（Instrumentation）是个很宽泛的词，有着广泛和丰富的含义。如同大多数具有多种解释的词一样，其准确含义在很大程度上取决于具体使用环境以及使用者的身份。

仪器学可以被定义为仪器的应用，它们以系统或设备的形式完成某些测量或控制（或两者一起）中的特定目标。表 1-1 列出了一些在仪器系统里会用到的物理测量实例。

表1-1：物理测量实例

加速度	质量
电容	位置
化学性质	压力
电导率	辐射
电流	电阻
流量	温度
频率	速度
电感	黏度
光度	电压

人类的自然语言是一种不精确的交流媒介，易受语境影响，而且一句话可以几种不同的方式来解读，因此前面对仪器学的定义还是比较宽泛。例如，对于过程控制工程师而言，仪器学可能是指压力传感器、加热元件、电磁控制阀，以及传送带。研究型科学家可能会想到激光、光学功率传感器、伺服驱动的 X-Y 显微镜平台，以及事件计数器。电气工程师则会将仪器学定义为数字电压表、示波器、频率计、频谱分析仪，以及电源。

一般说来，能测量的都能被控制，其中有些东西比较容易控制，而另一些则较难（至少就我们现有的技术而言）。当某个测量输入信号被用来为某个系统(通常被称为被控对象)产生控制输出时，可能需要对输入信号进行一定的修正或变换，这样才能符合系统的工作参数。这些修正或变换包括放大、电流到电压的转换、时延、滤波，或者某些其他类型的变换。

在本书中，我们将研究如何利用容易买到的低成本设备配合 Python 编程语言（主要语言）来实现基于计算机的仪器设备，从而完成各种数据采集与控制任务。本章先从较高的视角对全书的一些基本概念做简要介绍，在剩余的篇幅中，我们将自始至终和这些概念打交道。本章也会展示一些简单的仪器项目实例。如果你对本章中所介绍的一些概念不太熟悉，不用过分担心，后面还会对它们做更详细的讨论。本章的主要目标是讲述一些基础知识，并介绍一些基本术语。

数据采集

从计算机的角度看，所有数据皆由数字值构成，而且所有的数字值在计算机内部电路中都是以电压或电流的形式表示的。在计算机外部，不能被直接表示为数字值的物理动作或物理现象必须被转换成电压或电流，进而转换成数字形式。现在，把真实世界的数据转换为数字形式的技术能力相较过去已经有了很大的进步。

在蒸汽时代，人们用机械仪表来监测锅炉或管道的压力。为了从仪表上捕获数据，需要有人在某个时间点把读数写入记录本或是单页纸上。在今天，我们会使用变送器把压力的物理值转换为电平信号，然后再用计算机数字化并采集。

前面已经暗示，有些输入数据已经是数字形式了，例如来自开关或其他的开关型传感器的输入，又或者是来自某种串口（如 RS-232 或 USB）的位串。在另一些情况下，输入将是连续可变的模拟信号（可能是电压，也可能是电流），这些信号被传感器测量并被转换成数字格式。

3

在说到数字数据时，我们指的是按位编码的、计算机可以直接处理的二进制值。二进制数据被看成是离散的，一个位只有两种可能的值：1 或 0，开或关，真或假。通常说数字数据有一个大小，它指的数据的位数，位是构成数据的基本单元。图 1-1 展示了从一个单独位到 16 位字的数字数据。数据以位为单位的大小决定了其能表示的最大值。例如，一个 8 位字节有 256 种可能的值（如果仅考虑正值）。

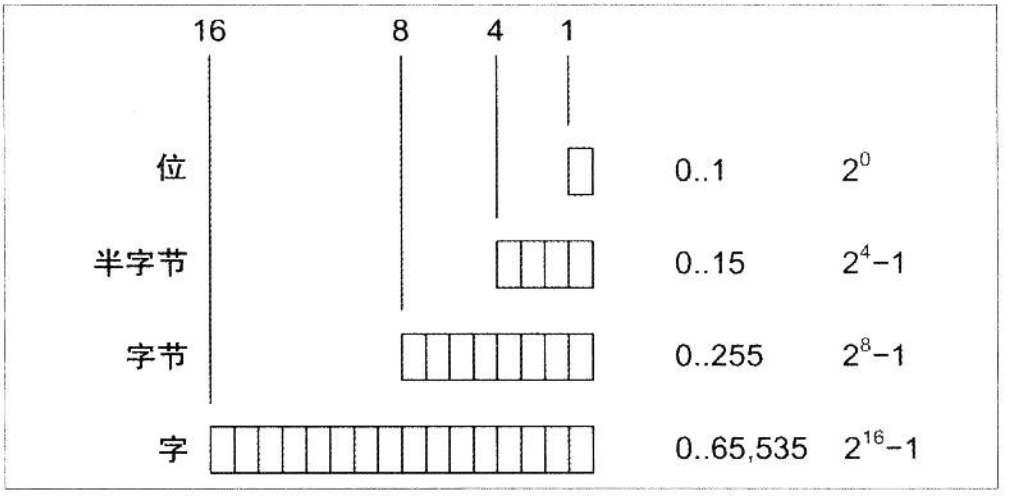


图1-1：二进制数据大小

对于来自诸如传感器开关的输入，数据大小可能只有一位。在一些别的情况下，例如测量压力或温度那样的模拟数据时，输入可能会被转换为 8,10,12,16 或更多位数的二进制数据。位数决定了所能表示的数值的范围。二进制除了能表示正数还能表示负数，而且有处理浮点数的标准格式，这些在图 1-1 中没有表现出来。

另一方面，模拟数据是连续变量，而且可取有效值范围内的任意值。例如，考虑 0 ~ 1 所有可能的浮点数的集合。我们可以看到像 0.01,0.834,0.59904041123 或 0.00000048 这样的数，以及在这个范围内的其他任意数字。模拟数据这个名字源自这样的一个事实：数据是连续可变的物理现象的模拟。

图 1-2 展示了以计算机为基础的数据采集系统的各种可能输入类型。开关等效于单个二进制数字（位）。串行通信接口可以是载有首尾相连的位串的一根导线，位串中每 8 位表示一个字母数字字符，或者表示一个二进制值。电压或电流形式的模拟输入信号通过采用叫做模数转换器（ADC）的设备转换成数字值。我们将在第 2 章中对这些设备和与之对应的数模转换器（DAC）做更详细的介绍。 4

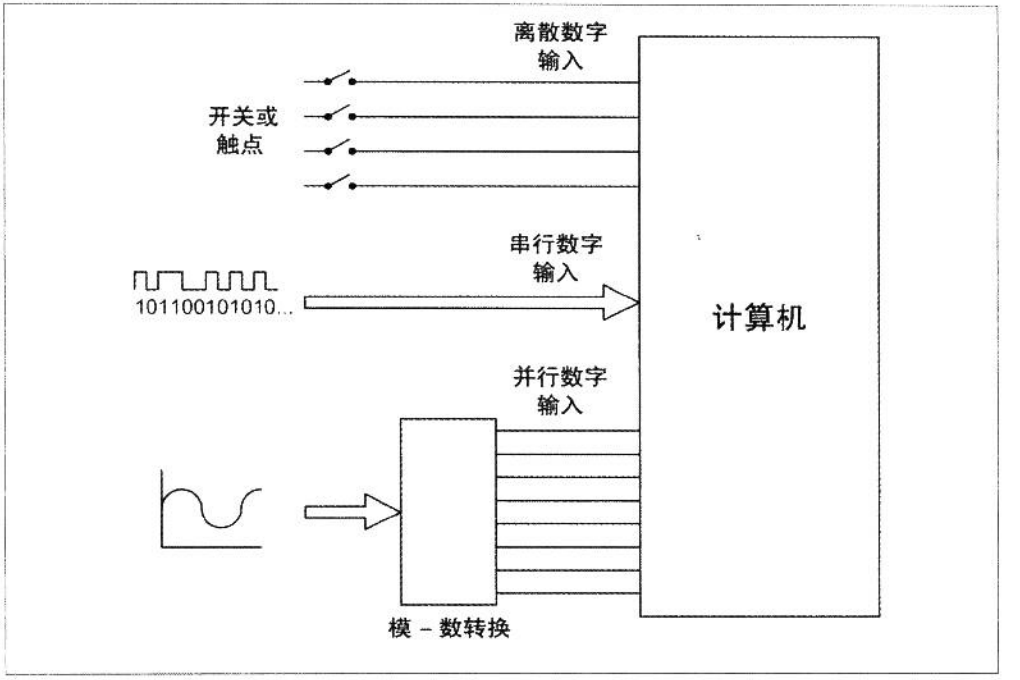


图1-2：数字与模拟数据输入

控制输出

仪器系统的数据采集部分负责感知物理世界并提供输入数据，而控制部分则使用这些数据在物理世界中施加变化。对物理设备施加控制涉及将某种命令或传感器输入转换成某种适当的形式，并以此使设备的活动发生变化。更准确地说，控制就是要产生数字或模拟信号（或两者皆有），然后用这些信号对设备或系统执行控制动作。根据是否使用反馈
5 的概念，线性控制系统大致可以分为开环和闭环这两种基本类型。

另一种常见的控制系统，即顺序控制系统（sequential control），把时间作为其主控制输入。在顺序控制系统中，事件相继发生在相对于初始事件的准确时间点上，而且各个事件通常是离散的。换言之，一个顺序事件不是开就是关，不是处于激活状态，就是处于非激活状态。从本质上讲，计算机就是一种顺序控制器，顺序控制器通常能用状态机来建立模型。我们将在第 8 章对状态图做一番考查。

所有这三种控制系统，在本书中都会遇到。第 9 章会对控制系统背后的理论做更详细的介绍，但在目前，一个高度的概览就足以应付稍后的讲解了。

开环控制

在开环结构中，系统的输出与控制输入之间不存在反馈。换言之，系统无法判断控制输出是否真正达到了想要的效果。即便如此，开环控制也还是有它的用处。开环控制系统的精度依赖于其各个组件的精度，同时取决于所控制的系统模型的好坏程度。图 1-3 显示了一个开环控制系统的简易框图。标为“受控设备”的部分可以是一个电动机、灯泡、风扇或者阀门。虽然乍一看开环控制中并没有什么东西，但事实上，某些开环控制具有极高的复杂度，而且这种控制还相当常见。

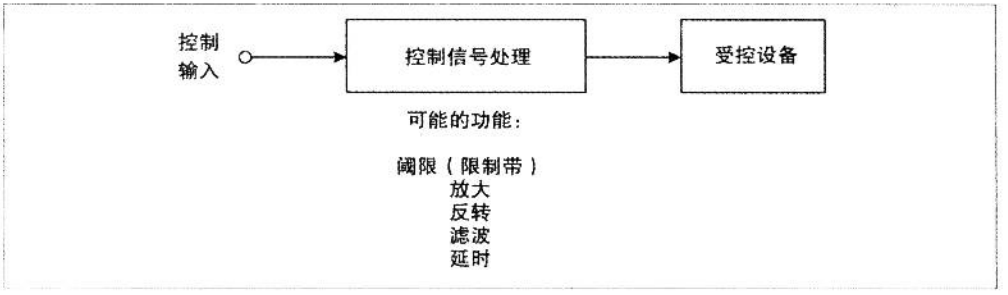


图1-3：开环控制

虽然在某种程度上我们可以认为开环控制系统是“瞎的”，但依然可以把时间参数集成到其设计之中。现实生活中的自动照明开关就是这样一个实例。图 1-4 显示了这种装置的一张大幅简化的框图。

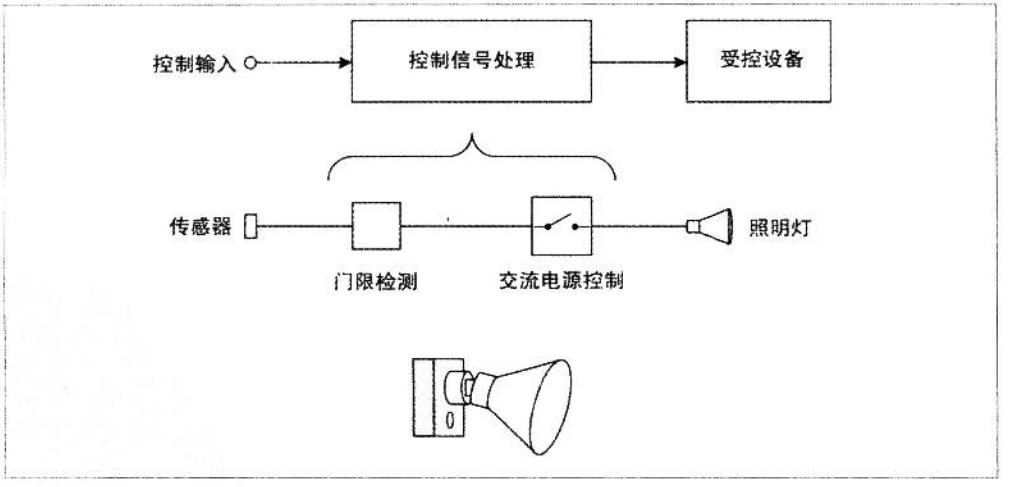


图1-4：开环控制实例

这些无处不在的装置有一个传感器（通常是红外），当有东西出现在传感器感应范围之内时，照明灯就会被激活点亮。在此装置中，没有用反馈信号来确保照明灯是否真正发光（至少普通的住宅用照明灯是这个样子），而且传感器根本无法区分出所感应到的到底是盗贼还是一只大家猫。

然而自动照明装置具有延时功能，可以让灯光在传感器的输入超过阈值后亮上一段时间。如果不这样做，则电灯先是被打开，然后随着传感器输入信号回落到阈值之下，电灯将被立即关闭。如图 1-5 所示。假如不采用时延来保持照明，那么只要这时来只大家猫在传感器前蹦上蹦下就会使电灯闪个不停。这可能会干扰到你的邻居（另外，自动照明延时过长也可能会让邻居受扰）。

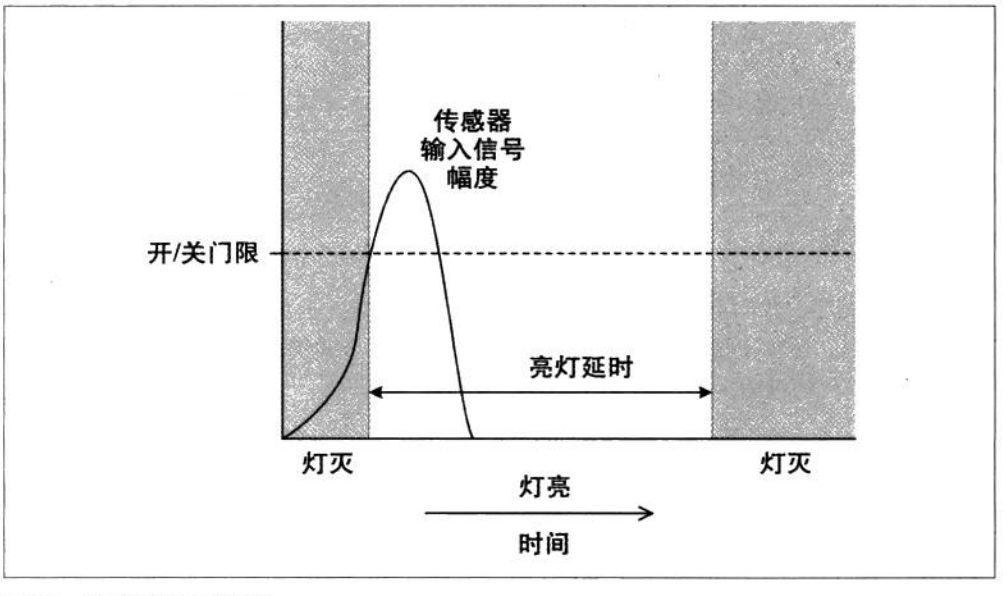


图1-5：带时延的开环控制

闭环控制

闭环控制利用取自受控设备或系统的数据，也就是反馈（feedback），来判断控制效果，然后根据一些内部算法（也叫“控制律”）对控制动作做出相应的调整。图 1-6 显示了一个基本闭环控制系统的框图。

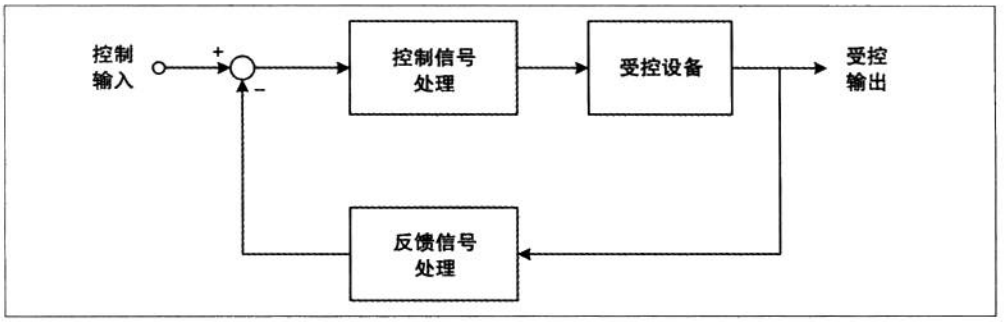


图1-6：闭环控制

在图 1-6 中，我们可以看到，控制输入和反馈信号在圆圈符号处使用相反的符号进行求和，这个圆圈符号被称为“求和点”或“总和节点”。相加的结果被称为控制误差。之所以这样做，是因为闭环控制的要点就是被控设备对控制信号的响应情况，控制信号由图中标识为“控制信号处理”的部分所生成。控制误差被输入到控制信号处理部分，然后系统会把控制输出输出到受控设备，竭尽所能使控制误差为零。那些熟悉运算放大器（op amp）电路的读者会立即发现：运算放大电路也是基于相同的工作原理。

有人可能会想到，图 1-6 中的系统框图并未表现出所有的细节。在控制和反馈处理部分的设计中都会包含一定的放大（增益）处理。还可能包含有衰减、滤波或阈限（limit threshold）功能。增益水平要根据实际应用进行选择，必要时甚至要采用非线性响应的放大。

我们来看一个有趣些的闭环控制案例。假设有一个容器，它里面的液体正以不断变化的速度向外排出，而我们要做的是把容器中的液位维持在一个恒定的位置。容器中液体的排出速度时快时慢，有时甚至根本就不排出。图 1-7 显示了该装置的结构和其上的控制回路。

传感器用于测量容器里的液位，如果液位低于指定值，系统就加大输入泵的流速，以使更多的液体流入容器。随着液位渐渐地接近目标设定值，泵的流速也开始变慢，而且一旦液位达到了目标值，则输入泵将完全停止工作。只要泵的输入能赶上并超过液体的排出速度，那么无论液体以多快的速度从容器中排出，利用这种方法都能对液位的变化进行自动补偿。

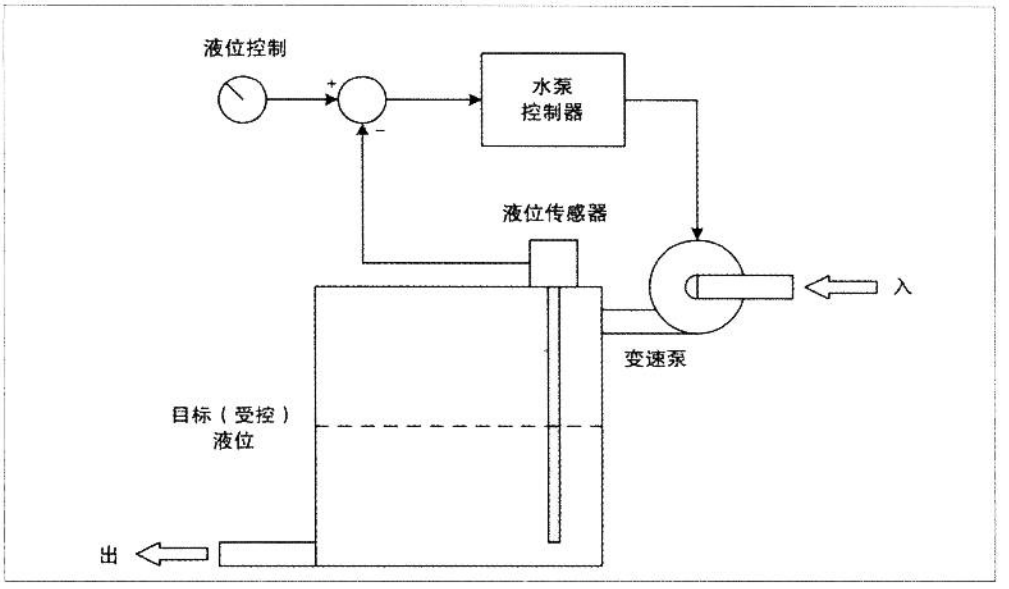


图1-7：闭环液位控制

9

顺序控制

顺序控制是控制系统中一种很常见的形式，而且易于实现。像麦片盒折盒机或动物饲料灌装机这类自动包装系统就是典型的时间顺序控制装置，它们采用电动或气动执行机构来执行一系列特定的控制动作。有些顺序控制系统会使用某些类型的传感器，以便在必要时改变控制顺序，或者感应故障条件并关闭系统。

图 1-8 显示了一个控制五个设备的顺序交流电源控制器的时序图。在这个例子中，在每个设备接通电源后加一个延时，可以让设备有一个稳定时间，并且能响应控制系统的查询以便确认设备是否运转正常。在这类系统中，每个设备通常都有三种可能状态：开启，关闭或发生故障。在定时程序中，控制器除了要控制设备的开或关外，还要检查每个设备以确认其是否正常上电。如果设备发生故障，那么控制器应该停止当前程序，或者启动一个自动关机过程，按照后开先停的原则终止已经开启的设备。

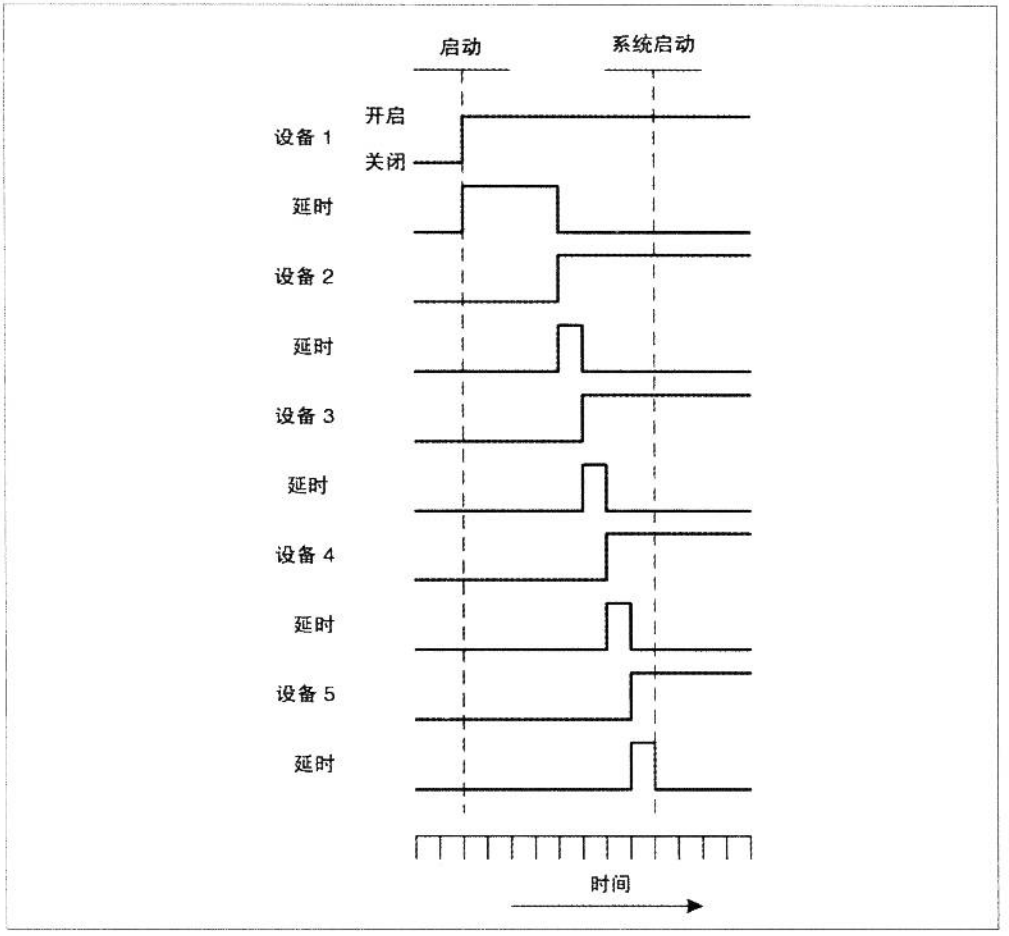


图1-8：顺序电源控制

应用概观

我们来快速浏览几个现实生活中基于计算机的仪器应用案例。请记住，这些例子的目标是展示自动化仪器都具有些什么样的用处，而不会详细地讲如何做一个特定的项目。在后面的章节中，我们将深入到接口、控制协议和软件算法的具体细节之中。

电子测试仪器

在电子实验室，甚至是装备良好的业余玩家的工作室里，多半会看到像示波器、逻辑分析仪、频率计、信号发生器，以及其他类似的设备。这些设备本身就很有用，但如果把它们集成起来组成自动化系统，那么它们能变得更为有用。

要想使用自动化装置中的某个测试设备，必须要有某种控制或采集接口。许多现代仪器都有 USB、以太网、GPIB、RS-232 接口，或者几种接口的组合（第 7 章和第 11 章将介绍这些接口）。有时，它们属于标准配置；而在另一些场合，在订购仪器时，这些功能要作为单独的选件购买。

图 1-9 显示了一个被测部件（Unit Under Test，UUT）的简单布置情况，它由某个信号驱动，其直流电源也是受控的，另外，它以逻辑分析仪记录和数字万用表（DMM）读数的形式采集测量数据。

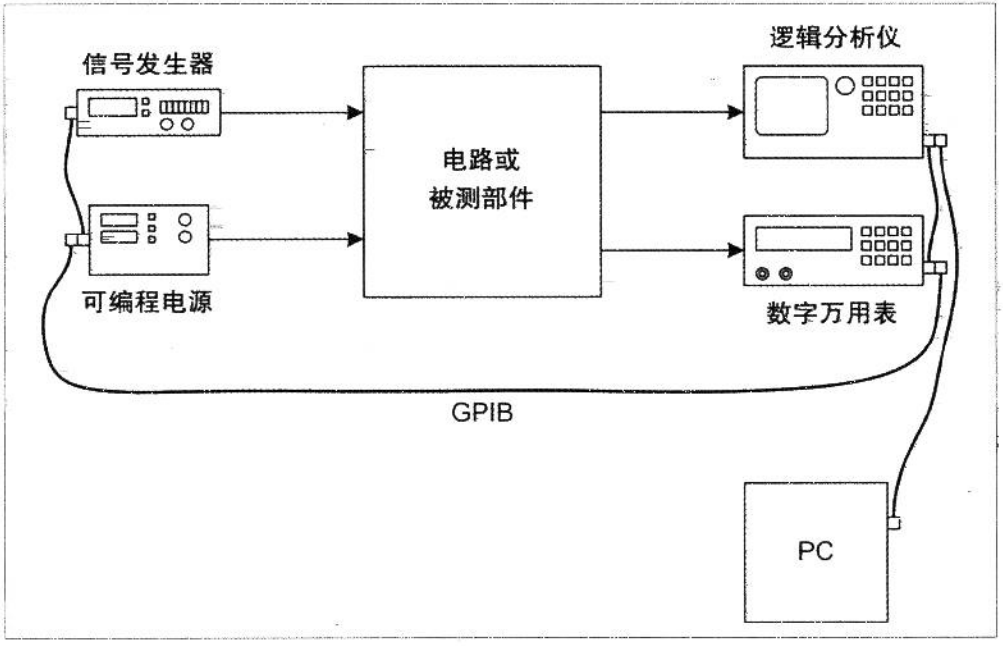


图1-9：测试仪器实例

10 图 1-9 所示的简易装置中有一个仪器，也就是信号发生器，它作为主激励输入连接到被测部件上。它产生的信号的形状（波形）和速率（频率）都是可编程的。信号电平（幅度）也能由 PC 控制。有两个仪器连接在被测部件的输出上，其中一个捕捉数字逻辑信号（即逻辑分析仪），另一个测量一路或多路电压值（即数字万用表）。还有一个可编程电源，它为被测部件提供可由计算机控制的电源。

在这个例子中，各种仪器通过 GPIB（General Purpose Interface Bus，通用接口总线，也叫 IEEE-488）连接到 PC 上。市面上有各种各样的 GPIB 接口器件，从插入式 PCI 卡到外置的 USB 转 GPIB 转接器都有。我们将在本书后面的章节中对其中一些进行考查，并

试图用各种方法为它们编写软件来控制仪器采集数据。

那这个装置有什么用处呢？图 1-9 所示可以看成是一个性能特性描述装置。如果被测部件会生成某种模式的数字信号来响应来自信号发生器的输入信号，则该测试方法就能将设备的这一工作行为捕捉下来。利用这一装置，还能获知当可编程电源的输出发生变化时，被测部件的行为将有怎样的变化，又或者当信号发生器的信号频率发生变化时，一些内部电压会有怎样的变化。前面提到的所有这些数据都能显示在 PC 显示器上，还能捕捉并存储到硬盘上，以供后面可能进行的分析使用。

实验室仪器

首先，一个研究实验室应该要有 pH 计、温度传感器、精密电炉、可调激光器和真空泵这些东西。图 1-10 显示了一个用于控制环境试验箱的仪器系统例子。

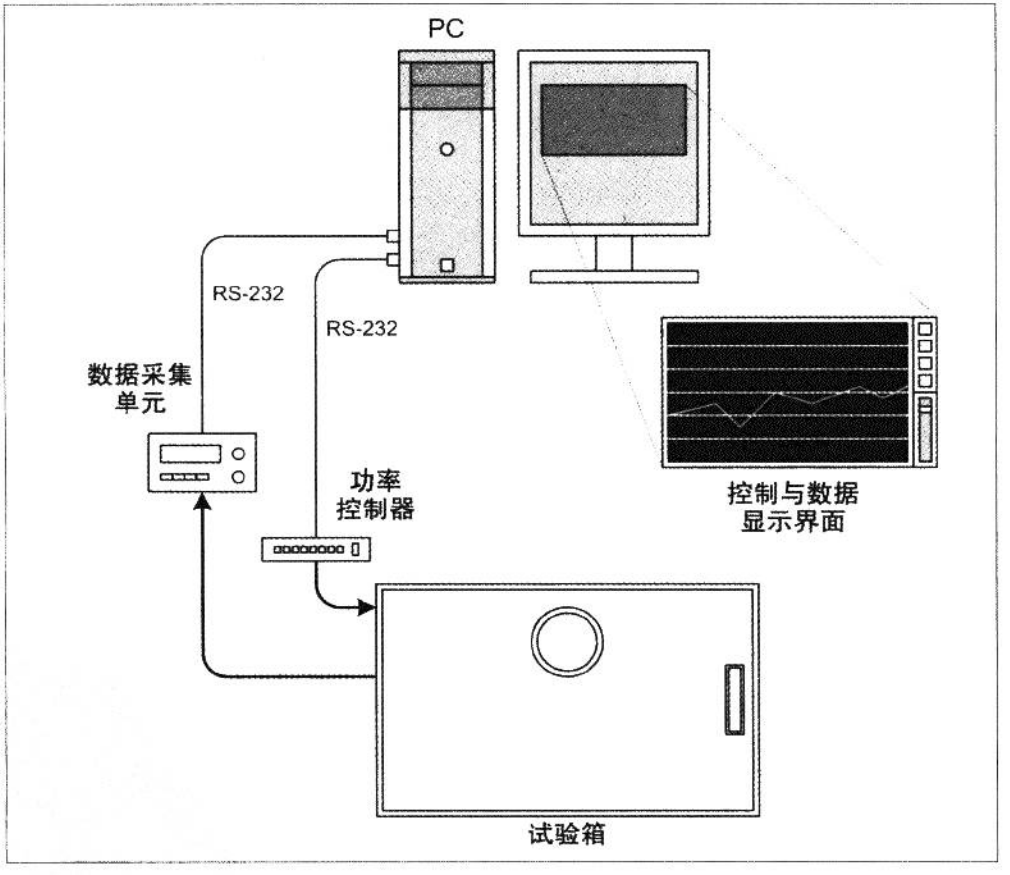


图1-10：实验仪器实例

在这里，试验箱用来做什么并不重要（它可以用于微生物培养，或许还能用于环氧树脂固化）。重要的是连接到其上的仪器，以及如何把它们一个一个地与计算机连接起来。在前面的例子里，采用了 GPIB 来实现仪器接口，在这里我们使用普通的老式 RS-232 串行连接。

12 图中的数据采集仪器负责对诸如温度、湿度之类的模拟信号进行检测和转换。它还能对任何连接到试验箱上的加热器或冷却器的电气状态进行监测。功率控制器则负责对试验箱里所有的加热器、冷却器、低温阀或别的受控功能进行控制。

这类装置的主要目标大概是在某个预定的范围内把温度一直维持在一个特定的值。针对装置的实际应用场合，有些试验箱还具有倾斜升温和降温功能。一般说来，这类系统中的任何实验都不是发生在短时间尺度上的；显著的变化基本上都要花费几分钟到数小时不等。

13 如果采用 Bang-Bang 控制器（一种开关式的非线性控制器，我们稍后会对其做详细介绍）实现方案，则根本无须对加载到加热器或冷却系统上的功率值做（连续）调节。它的工作方式和房间里的恒温器非常类似。由于控制器无须以很小的时间常量（即快采集速率）来运行，因此该仪器设备使用速度很慢的 RS-232 接口也没有问题。

过程控制

图 1-11 展现了一个简单的自动过程控制系统。该系统可用于生产人工枫树糖浆，也可看成是能生产某种产品的受控化学反应装置。请注意，这个框图画得并不十分标准，它的主要用意是阐明概念，而避免卷入标准化的过程控制符号体系的细节之中。

在图 1-11 中，我们还可以看到另一种接口类型——USB 接口模块。这些模块很常见，而且相对比较便宜。如果你愿意，甚至可以买个套件回来自己组装。这种模块大多数会提供一组离散输入和输出，一些带 10 位或 12 位转换的模拟输入，甚至还有一些模拟输出或者是一到两路脉宽调制（PWM）输出通道。

在图 1-11 中有四个阀门，标为 V1 ~ V4，其中每一个都被连接到 USB 接口模块的离散输出上。加热器也被连接到了离散输出上。请注意，虽然必须要有电路来把 USB 控制器的 5V 离散信号转换成足够驱动阀门或加热器的电流 / 电压，但图中并未画出这部分电路。我们将在第 2 章中考查怎样驱动使用大电流或高电压（或两者兼有）的外部设备。三个模拟输入被用于从传感器中获取液位值、温度值和压力数据。

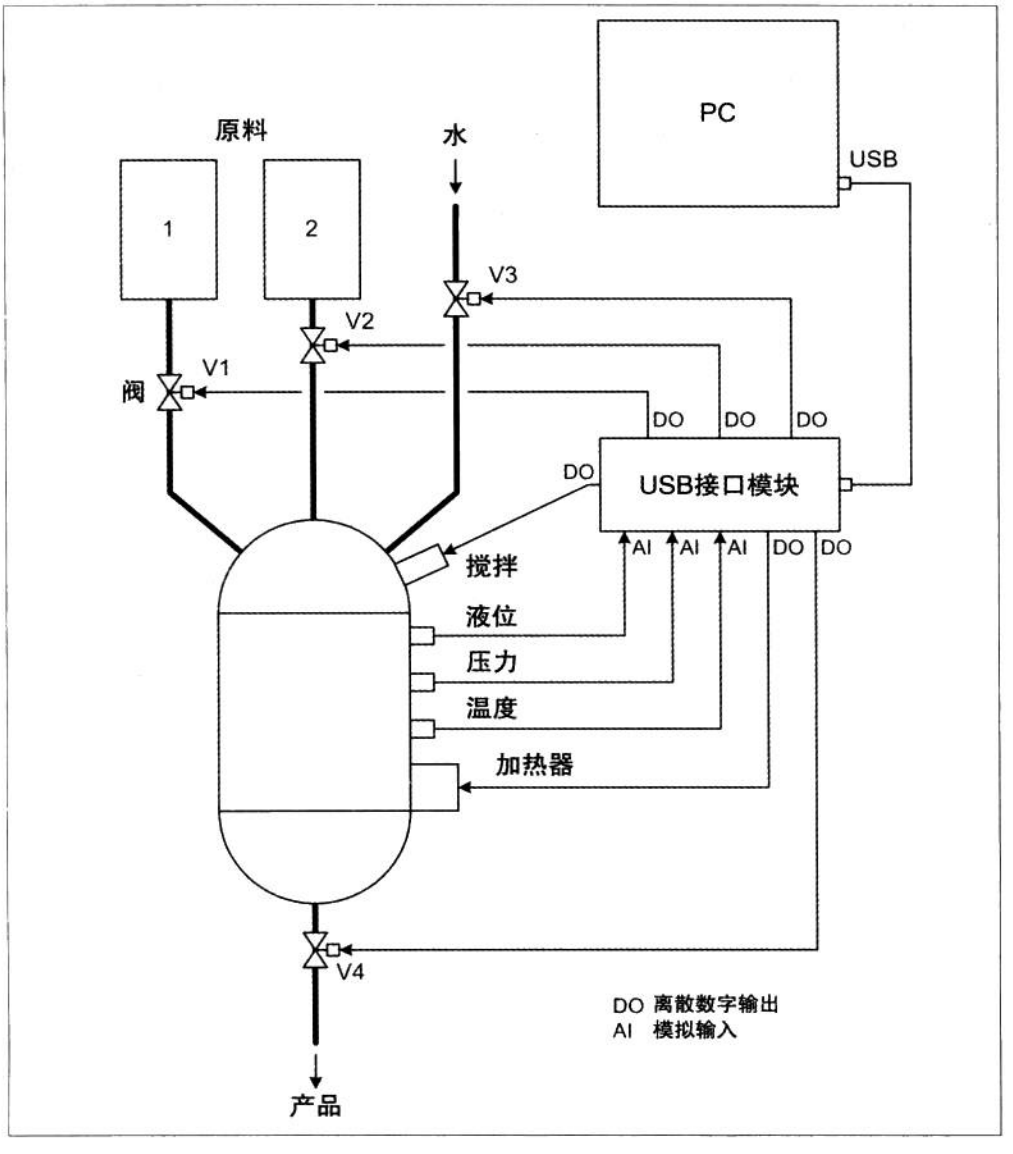


图1-11：一个简单的化学处理系统

和前面的例子一样，该系统也算不上是高速系统。只要每隔 1 ~ 5 秒读取一次传感器并且对控制进行更新（阀门和加热器），该系统就能良好工作。

小结

仪器学应用领域宽广而深厚，像这样一章，除了对仪器学是什么以及它能用来干什么做一个大致介绍之外，根本无法捕捉其全貌。一些术语和概念初看起来会显得比较陌生而且怪异，不过后面的章节会对它们做进一步的讲解。本章的主要目的是为你提出一些基本概念。随着讲解的推进，细节部分将会慢慢呈现出来。

第2章 基本电子学

事实上，电是由极其微小的被称为电子的东西构成的，肉眼看不到，除非你喝醉了。

——Dave Barry，美国幽默作家

尽管这是一本主要关于仪器学软件的书，但我们也必须要考虑在现实世界中软件与之相互作用的对象——换句话说，就是仪器学的硬件方面。本章将从仪器学的角度对电和基本的电子学做一个总体的高度概览，而不打算对背后的理论和物理学原理做深入的探究。

电子学是一个深厚而广阔的研究领域。为了不让本书成为一本这方面的参考读物，在此对一些主题只做了轻描淡写，有些甚至根本就没有提到。如果你对电子学的了解不只限于欧姆定律，那么可以直接跳过这章往下读，然而，如果你还不是很确定什么是欧姆定律，或者不能区分什么是电流输出（current source）和电流吸入（current sink），也不清楚“波形”一词是什么意思，又或者不知道数字和模拟输入/输出相互有什么区别，那本章就是为你而写。

我们先概括性地讲一下电荷和电流，然后介绍一些在电路原理图中使用的符号。接下来，我们会讲述非常基本的DC（直流）和AC（交流）电路，然后从电学角度对仪器系统里的各类输入和输出类型做一个探讨。在后面的章节中，对新概念会做必要的介绍和解释。本章最后提供了一份用于进一步学习电子学的参考书列表。

电荷

对于大多数人来说，“电”这个词通常指的是街边电线杆上悬拉着的电线里、墙上的插座里、计算机里或者是电池的端子上的东西。然而，准确地说，它究竟是什么呢？

一切物质都是由原子构成的。每个原子的核心部分是原子核，带有净正电荷，原子核的周围束缚着一个或一群电子，每个电子带有负电荷（虽然人们常听说电子绕着原子核作

轨道运动，然而这并非是像地球绕着太阳转那种经典的轨道；笔者建议你查阅一本现代化学或物理学课本以获得比此处更准确的定义）。原子核有一个或多个质子，每个质子都带正电荷。多数原子还有中子，中子只有质量而不带电荷（可以把它想成是原子核的“压舱物”）。一块平常普通物质里的原子其净电荷将为零（电中性），原因是电子的数量和质子的数量一样多，每个电子带单位负电荷，同时每个质子都带单位正电荷。图 2-1 展示了一个氢原子和一个铜原子的结构。

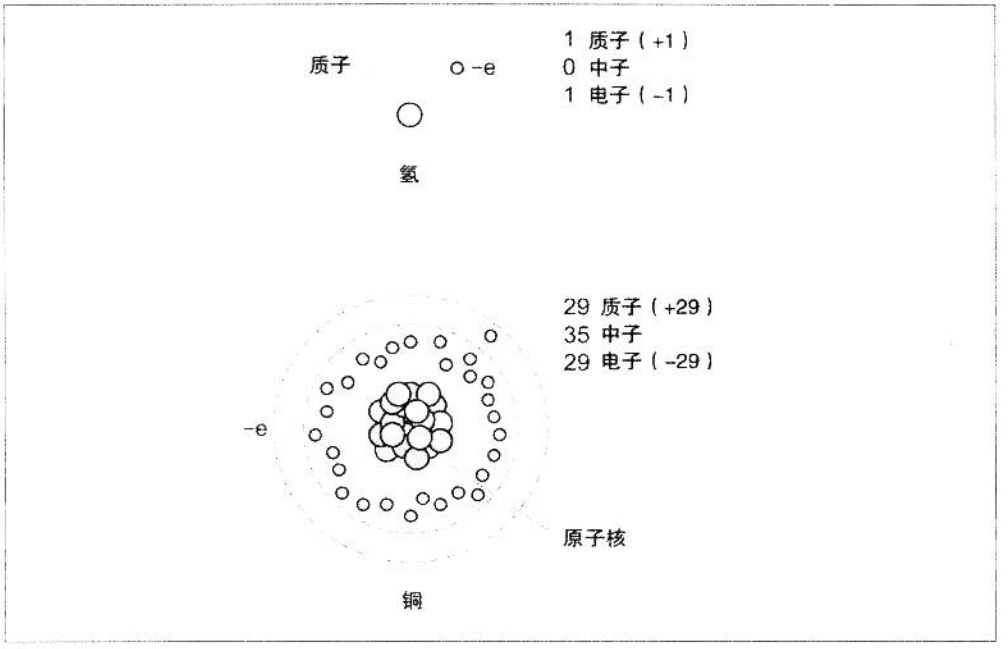


图2-1：原子结构

有些原子的电子被牢牢束缚住，而有些原子却能很轻易地失去或得到电子。这是为什么呢？下面做一个简要的描述。

在原子中，电子分布在被称为电子层的壳层上。最外的壳层叫作价电子层。有些元素的价电子层上只有一或两个电子，这样的壳层被认为是“非完整的”，这样的元素往往容易释放或捕获电子。注意图 2-1 中的铜原子，它有 29 个电子，其中有一个电子位于 28 个电子组成的主群体之外，它就是铜的价电子。由于该电子不是被牢固地束缚着，因此在束缚这个电子方面，铜原子并不花费过多的“精力”在上面。换句话说，铜是一种好的导体。另一方面，像硫这样的元素，不愿意放弃任何电子。硫被认为是导电性最差的元素之一，因此它是一种好的绝缘体。银居于导电性最好元素的榜首，这正是它在电子技术里被认为非常有用的原因。

17

前面讲述的模型已经可以满足要求，因此我们不打算进一步深入了解原子结构的内部秘密。在这里，我们真正感兴趣的是原子在传递电子时究竟发生了些什么，以及它们为什么会有这样的行为。

电流

电荷与电流是两个基本的电现象。电荷是物质的一种基本特性，是某个物体电子过多（负电荷）或过少（正电荷）时出现的结果，当物体带电荷时会试图达到电中性。

电荷的一个基本性质是同性电荷相互排斥，异性电荷相互吸引。正是这个原因让电子和质子在原子里束缚在一起，但它们也不能被直接吸引到一起去，这主要是受一些别的原子粒子的基本性质所影响。切记：负电荷排斥电子而正电荷吸引电子。

电荷，单就其本身来讲是有趣的，但从电子学的角度来看并不特别有用。联系到本书的目的，只有当电荷移动起来真正有意思的事才开始发生。电流就是流过电路的电子流。在干冷的天气走过地毯然后碰到门把手时，产生的静电被转移到门把手上，这也是电流。实际上，电流从电势高处（你）流向电势低处（门把手），这和水从瀑布流下的情形非常相似。毫无生气的静电立即变得有意思起来（它至少引起了你的注意）。

当构成导体的原子和电路元件使电子从一处转移到另一处时，电流就产生了。电子向正极方向移动，因此，如果拿两根导线把一只小灯泡接到一节电池上（和手电筒类似），电子会从电池负极出发，经过灯泡，然后回到正极。沿此路径，电子会使得灯丝达到白炽状态而发光。图 2-2 是对电流的可视化，从中可以看到导线里铜原子的简化图。当一个电子进入导线的一端时，它会使第一个原子带负电荷。这样这个原子就有了过多的电子。假设我们有一个连续的带净负电荷的电子源，新进的电子没有退路，不能从进入的地方跑回来，因此它们只能移动到下一个中性的原子上去。于是这个原子也带上了负电荷，有了多余的电子。为了再次变成电中性（原子的本性），它又把多余的电子传递到下一个（中性）原子上，如此重复一直到一个电子在导线的另一端出现。 18

图 2-3 是另一种认识电流的方法。在这里，想象一根从头到尾塞满了玻璃珠（电子）的管子（导体）。

如果从管子的一端推一颗玻璃珠进去，另一颗珠子将从管子另一端掉出来。管子里的珠子总数还是维持不变。注意，从导体一端进入的电子并不一定与从另一端出来的电子是同一个，如图 2-2 和图 2-3 所示。

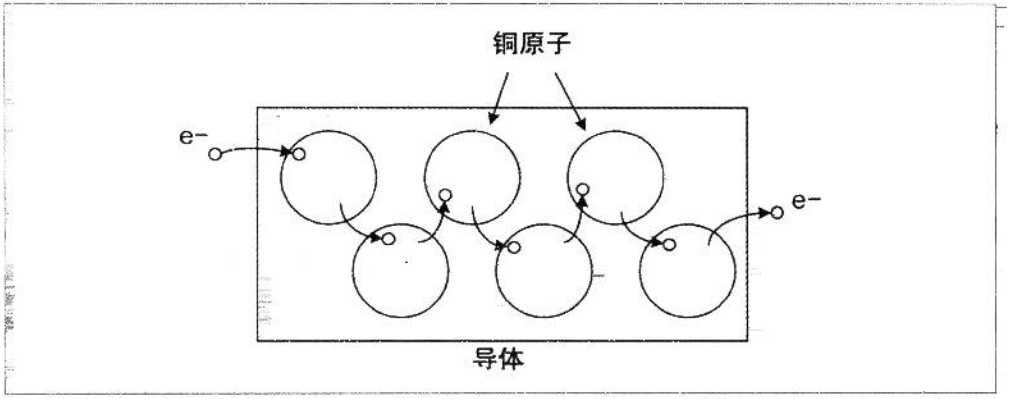


图2-2：铜线中电子的移动

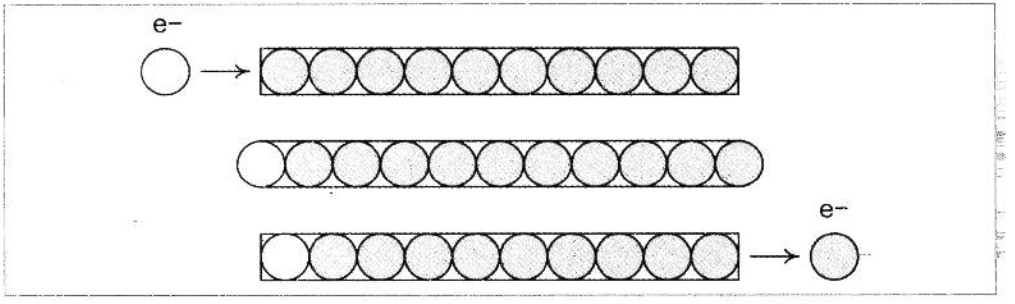


图2-3：电子移动的模拟

19

基础电路理论

当存在闭合回路和电势差时电就会流动，电子从电势高处流向电势低处。也就是说，要让电子流动必须得有电子源，还必须有电子的返回点。电流（此处指物理现象）由四个基本量表征：电压、电流、电阻和功率。我们将以图 2-4 中的简单电路为基础来进行后续讨论。

如图 2-4 所示，只朝一个方向流动的电流被称为直流电（DC），普通电池和计算机系统的 DC 电源就输出这种电流。周期性地改变流动方向的电流被称为交流电（AC），家用电源插座（如在美国）里出来的就是这种电。推动立体声音响喇叭的也是交流电。电流方向的变化速率被称为频率，以每秒多少个周期来衡量，单位是赫兹（Hz）。因此，一个 60Hz 的信号就是一个以每秒改变 60 次方向的电流所构成的。我们现在将只讲直流电路，稍后再讲交流电路。

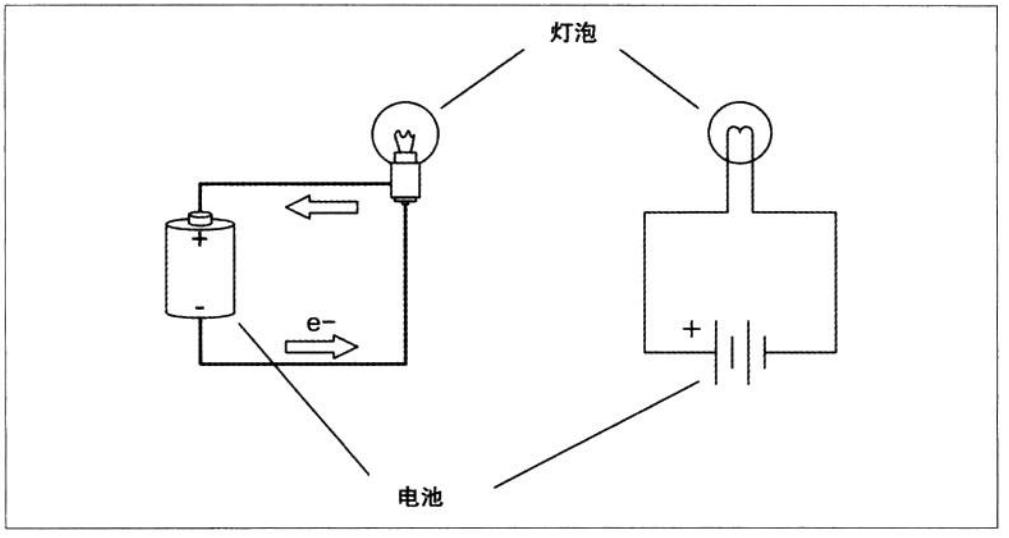


图2-4：简单电路图

按照约定，电流描述为从正极向地（负极）流动，事实上，电子是从电源的负极流向正极的。在图 2-4 中，箭头方向就是电子流动方向。尽管如此，从今往后，我们还是按正极流向负极这一传统约定来讨论电流。

电压是衡量有多少电荷或多大电势能往电路里推动电子的物理量，其单位是伏特（V）。 20
电荷会产生力，它会向别的带电物质施加力的作用。电荷越多，施加的力就越大。能量、力和势能这些经典力学里的概念同样适用于电荷。此处要重点记住高电压比低电压有更大的力。这就是为何你只能用图 2-4 中那种普通手电 1.5V 电池打出勉强可见的一点电火花，而闪电以其高达 10 000 000V 左右（以至更高）的电压能一直从云层到地面拉出一道明亮的光。闪电有更高的电压，因而其背后有更大的力，大到足以击穿作为绝缘层的空气。

电流是对流经电路的电子数量的度量。“电流”这个词的意思依赖于其所处上下文。在到目前为止的讨论中，我们讲的电流是指流过导体的电荷，这是一个物理现象。在电子学里，“电流”这个词通常是指在单位时间里流过导体特定位置的电子的数量。此时，它指的是一个物理量，其测量单位是安培（A 或安）。

电阻是对一个电路对于电流通过的阻碍能力的度量，其单位是欧姆。可以把电阻想成是机械摩擦（虽然这个类比并不完全准确）。电流在流过有电阻的物体时，会在物体两端产生一个电压降，因此电流的一部分能量（电压）以热的形式被损耗掉了。损耗的能量是关于通过电阻的电流和压降的函数。我们将很快看到一个著名的等式，该等式刻画了电压、电流和电阻之间的这一关系。

功率是对电流克服电路电阻或某种做功（如驱动电机）所消耗能量的度量，它是关于电压和电流的函数。常用测量单位是瓦特（W），偶尔也会看到以焦耳（J）等单位来表述。

电路原理图

在继续往下之前，我们需要引入一些符号以帮助描述讨论的内容。电路是用原理图来图形化表述的。各类电子元件都有与之对应的工业标准符号，这些符号在图上的连接关系描述了真实元件在实际电路里的连接关系。图 2-5 展示了常见的被称为“无源器件”的几个示例符号。

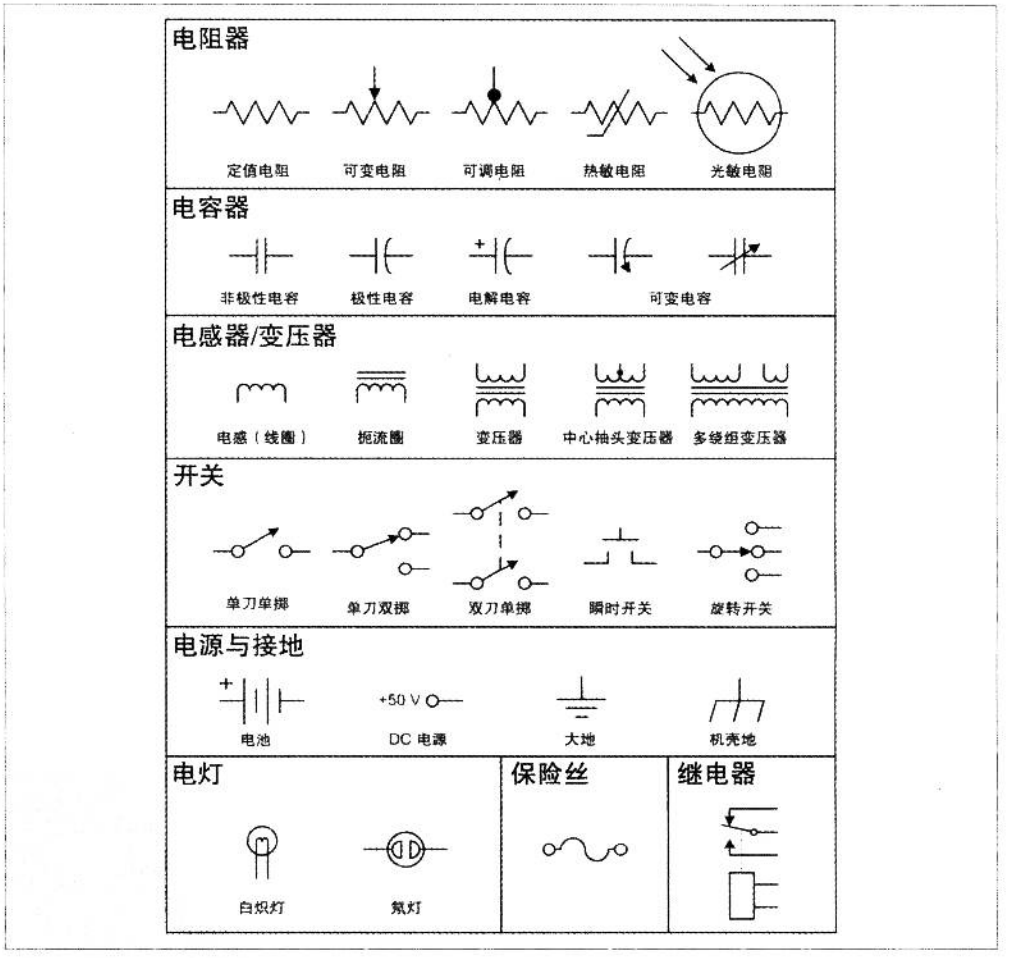


图2-5：常见电路原理图符号

图 2-6 是二极管（整流器）和各种晶体管的符号。这些器件被称为固态器件，由于能非线性地改变电流，因此也被认为是有源（主动）器件。

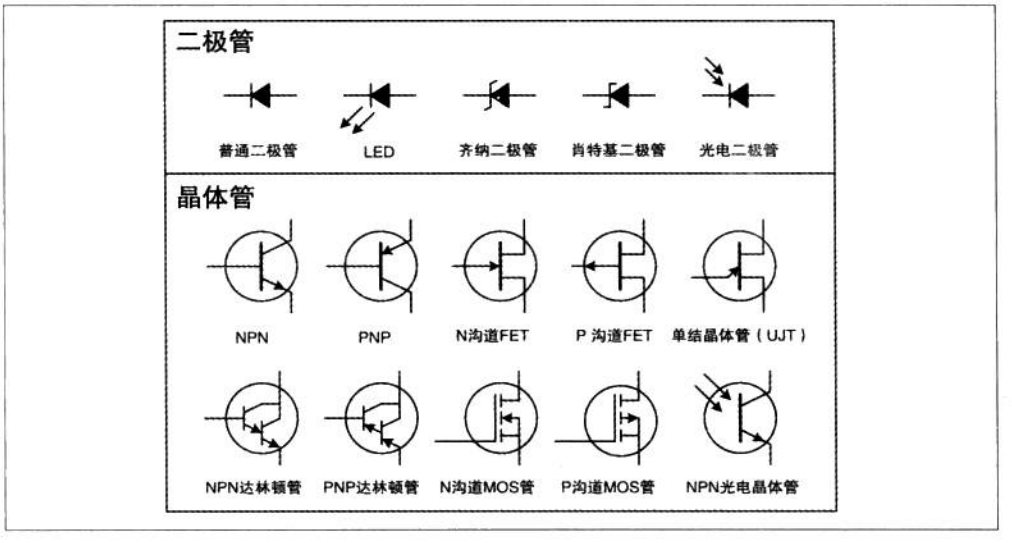


图2-6：固态器件

实际电路中的导线（或电路板上的布线）在原理图中用连线来表示。线与线交叉且相连时，在交叉处以一个实心圆点来表示；如果没有实心圆点，则表示它们只是简单的交叉。图 2-7 演示了这两种情况。一些旧式的原理图在一条线上画个小圆弧来表示不和与它相交的线连接，现在基本上已经不再用这种画法了。同样，为了避免画多条平行线来表示一组相关的线（例如数字数据总线），数据总线用一条粗重线条来表示，同时标示总线中独立导线的条数。

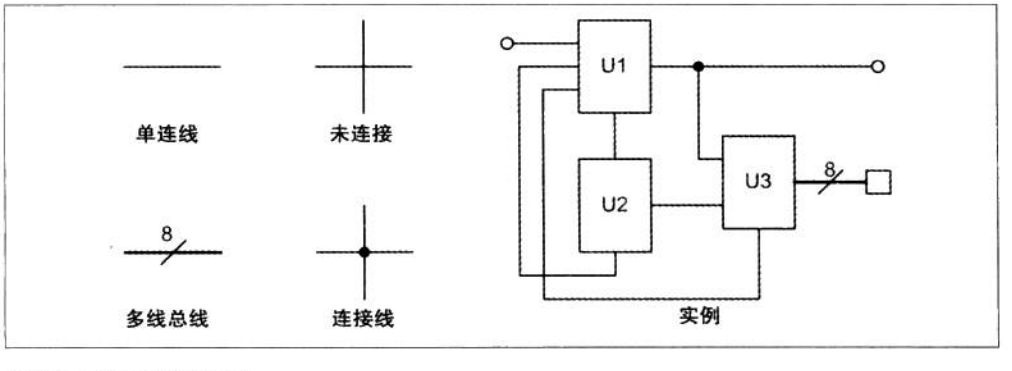


图2-7：原理图接线标记

23 数字逻辑电路有其自有的一套符号，同时集成电路（IC）里的功能模块，如触发器、计时器、寄存器、锁存器等等，通常都用矩形来表示，矩形上标示有输入和输出连接。图2-8展示了这些符号。

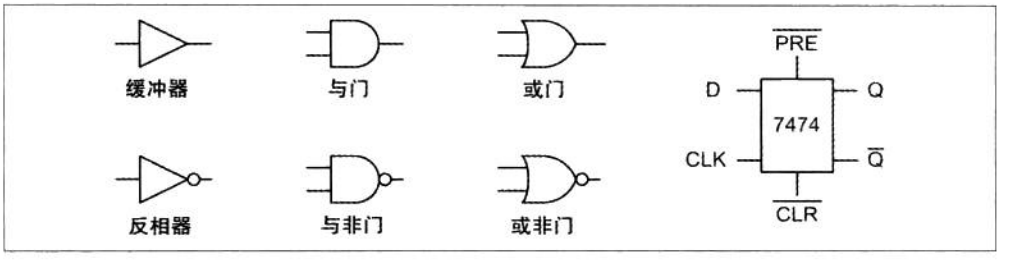


图2-8：数字逻辑符号

图 2-5~ 图 2-8 里的这些符号当然没有囊括所有的原理图符号。一整套当代使用的原理图符号比这里介绍的要多得多。事实上，这些原理图符号的一个小子集就足够本书使用了。

和其他专业一样，电子学充斥着它独有的一套奇怪的缩写，首字母缩略词和其他的行语。下面举几个例子：

DPDT

双刀双掷开关。

DPST

双刀单掷开关。

NPN

一种晶体管类型。该缩略语指的是构成工作结面（active junction）的 N 型 -P 型 -N 型掺杂组合，器件中经调制的电流就流经这些结面。

PNP

结构与 NPN 晶体管相反。它采用 P 型 -N 型 -P 型配置。

极性（*Polarity*）

端子或器件的正负状态。

有极性的（*Polarized*）

当一个器件在使用时总是要留意区分其中一个端子比另一个端子要“正”（或“负”）时，它就被称为是有极性的。无极性器件在连接上没有什么讲究，怎么连都行。

SPDT

单刀双掷开关。也指“机械中位断开”式开关。

SPST

单刀单掷开关。

数字逻辑符号上面的小圆圈（见图 2-28）表示取反。也就是如果逻辑真（1）遇到这个圆圈，它就会被取反而变成逻辑假（0），反之亦然。表 2-1 是与门的真值表（A 和 B 是输入）。 24

表2-1：与门真值表

A	B	输出
0	0	0
0	1	0
1	0	0
1	1	1

与非门在输出上有个小圆圈，表 2-2 是其真值表。

表2-2：与非门真值表

A	B	输出
0	0	1
0	1	1
1	0	1
1	1	0

千万别把逻辑取反和线路图里常用来标示接线端子的空心圆圈搞混淆了，那些圆圈不表示逻辑取反。只有当空心圆圈位于数字器件符号旁边时，它们才表示逻辑取反。

随着讲解的进一步深入，还会重新讲到原理图符号，因此现在并不需要急于把这些符号都牢记于心。当然，如果你的好奇心很强，笔者推荐你研究本章末尾的参考书，里面有关于原理图以及构成原理图的各种符号的详细资料。

直流电路特性

现在我们来仔细研究图 2-4 中的灯泡和电池组成的电路。这个电路看似简单，实则不然。

把灯泡连接到电池上，闭合电路，此时将有电流流过灯泡然后返回电池。如果是开路，则不会有电流，灯泡也不会发光（开关的作用就是用来断开或闭合电路）。要想在电路里产生电流，需要一个能量源，它能以某个足够推动电路工作的电压（电动势）产生一定数量的电子（电流）。

电流流过电路时，总会有一定的电阻；甚至导线也有一定的电阻。电阻低的电路与电阻高的电路相比更容易让电流通过，同时高电阻电路需要更高的电压才能达到和低电阻电 25

路同样的电流大小。例如，一个电路，其电压为 10 伏，电阻是 50 欧姆，电路里会产生 0.2A（也可说成 200mA，1mA = 1/1000A）的电流。如果把电阻增加到 100 欧姆，并且还要让电流维持在 200mA，则电压需要被提升到 20V。另外，在电流相同的情况下，电阻大的电路会比电阻小的电路有更大的功耗（表现为热量）。对此，我们稍后会详加解释。

欧姆定律

你可能已经猜到，在电压、电流和电阻之间有一个基本关系。它叫做欧姆定律，如下：

$$E=IR$$

其中 E 是电压，I 是电流，R 是电阻。这个简单的等式是电子学的基础，事实上，在实现多数仪器时，这是唯一不可或缺的等式。

在图 2-4 中，该电路只有两个元件：一节电池和一只灯泡。灯泡在该电路中作为“负载”。白炽灯的电阻随温度变化而变化，我们这里将假设灯泡发光时的电阻是 2 欧姆。电池电压是 1.5V，我们假设它能以额定电压持续一小时输出最大 500 毫安的电流。（这是电池的总容量，对于一节普通的 AA 型电池，容量约为 0.5 安时）。

根据欧姆定律，灯泡从电池吸收的电流大小由下式给出：

$$I=E/R$$

或

$$I=1.5/2$$

$$I=0.75\text{A}$$

式中，I 的值可以写为 750mA（毫安）。如果想要知道电池能坚持多久，只需用电池容量除以电流：

0.5/0.75=0.67 小时（大约）

这可以解释为什么在一些展销会上散发的那种小巧的，使用单节五号电池的手电用不了多久就要更换新电池。

在图 2-4 所示的简单电路中，流过灯丝的电子使灯丝被加热到能明亮发光的温度（大约 1600~2800℃）。灯丝之所以发热是因为它有电阻，因此比起电路中的导线，电流更不容易通过灯丝。强行使电流通过灯丝所耗费的能量以热的形式被表现出来。一个器件所转换及消耗的能量被定义为耗散功率，以瓦特（W）为单位。直流电路的功率计算方法是

用电压乘以电流，如下：

$P=EI$

如果想知道电路中的灯泡耗费多少电能，只需用灯泡两端的电压乘以电流：

$P=1.5\times0.75$

$P=1.125$ 瓦特

现在，我们来看一个稍复杂一点的电路，其中有几个新符号。注意图 2-9 中的简单 LED 电路，LED（发光二极管）上串联着一个电阻。电路标示的电源是 5V 直流，但是并未画出，还有一个接地的连接（它是电流返回电源的回路）。

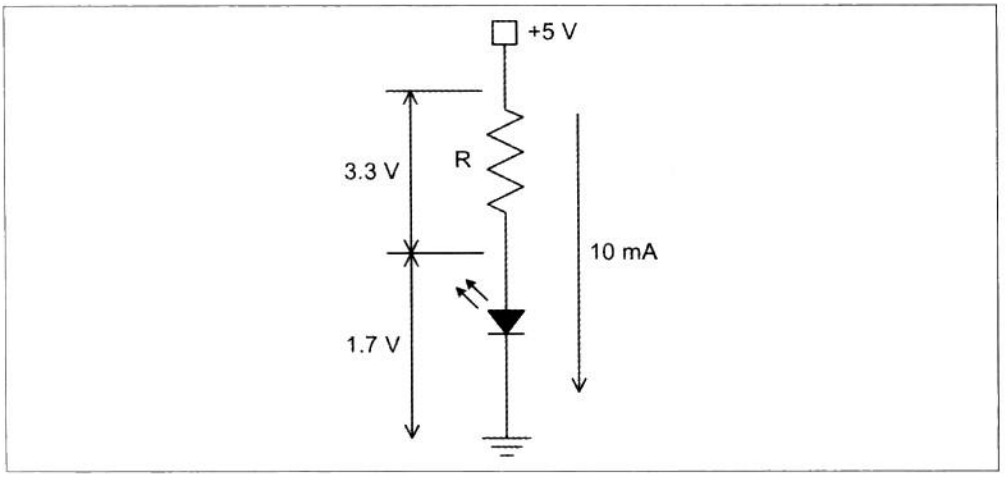


图2-9：简单LED电路图

一只典型的普通红色 LED 在其两极间会有约 1.7V 的压降。注意电阻上的压降和 LED 不一样，LED 上的压降是固态“结”（PN 结）的一个特性，这个固态结是 LED 的核心部分。如果 LED 的额定电流是 10 毫安（mA，或 0.01 安），电源为 5V，那么应该选用多大的电阻才能让 10mA 的电流通过 LED 呢？

由于假定了 LED 的压降为 1.7V，因此加到电阻上的压降就是 3.3V。如果用欧姆定律求解 R，将得到： 27

$R=E/I$

$R=3.3/0.01$

R 330欧姆

因此，用一只330欧姆的电阻就可以让LED得到10mA的电流并发光。这个简单的例子比你预想的要来得重要，到研究在数字离散I/O端口上连接LED时，它会再次出现。

电流吸入与电流输出

在查看某个接口器件的规格时，会碰到电流吸入（sink）和输出（source）额定容量的概念。这实际上是指，当在电源正极和器件之间连接一个负载时，器件能够“吸收”指定容量的电流而不会损坏。反之，当在器件和地之间连接一个负载时，器件能够安全地“供给”额定大小的电流。图2-10图示了这两个概念。

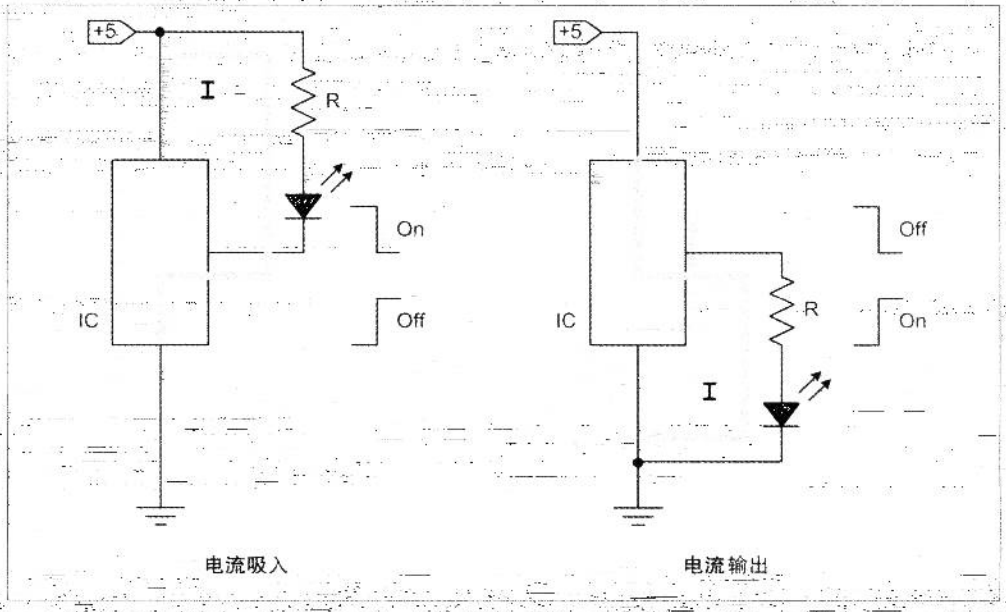


图2-10：电流吸入与电流输出

上升和下降符号指示了当器件输出高电平或低电平时LED的响应情况。当器件输出的电压低于+5V的电源电压时，电流吸收接法中的LED就会开始工作，而当器件输出的电压高于地时，电流供给接法中的LED将会开始工作。在图2-9中求R值的方法同样可用于此处。一定要注意不要让电流超过器件的最大额定吸入或输出容量。例如，一个器件的额定输出电流是20mA，这也许是指整个器件的容量，而不是指单独一个输出端口。

再谈电阻

电阻在电子电路里无处不在。电阻可以采用诸如弱导电性的碳材料，沉积在陶瓷芯上的碳膜，或绕在保护性封装里的一段阻性导线来制造。电阻除了有明确的阻值（单位欧姆），还有额定功率。常见的有 1/10,1/8,1/4,1/2,1,2 和 5 瓦，对特殊的应用还有更高的规格。额定功率是电阻在把自己烧毁前能安全地散发掉的最大功率。最后，电阻还有精度之分，精度范围从 20% 到小于 1% 不等，精度越高，电阻的价格也就越贵。最常见的精度是 5%。

图 2-11 所示是一只普通电阻。这种电阻用色环来标识其阻值，具体的色环代码可以在任何一本基础的电子学课本里找到，而且各个网站上也有相关的解释，此处不做介绍。

28

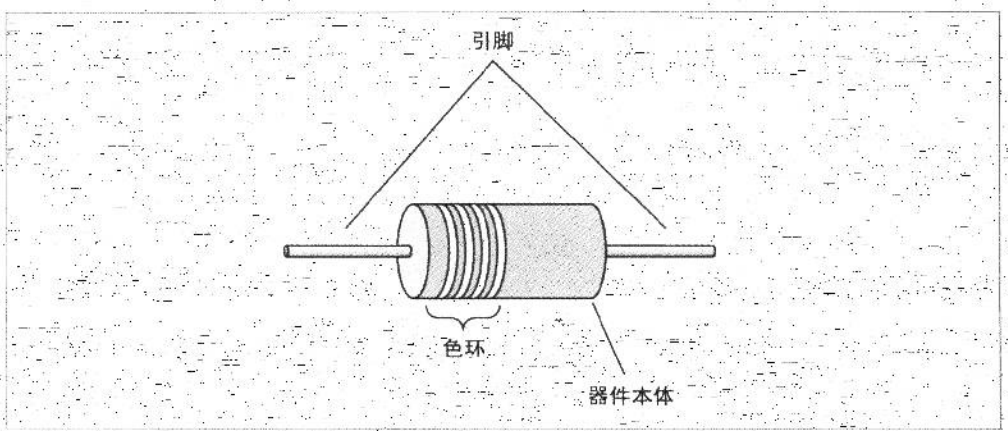


图2-11 普通碳质电阻

电阻可用来降低电压或限制电流。电阻作为限流器，常应用于类似图 2-9 那样的仪器应用电路中。电阻可以单独使用，也可以并联或串联使用。图 2-12 演示了电阻的串 / 并联以及等效电阻的计算方法。

现在，我们已经积累了足够多的直流电知识，是时候用电压、电流和电阻来做一些有趣的事情了。图 2-13 演示了一种简单地把两个电阻连接起来的接法，叫做分压电路。这种电路经常出现在电子电路里。

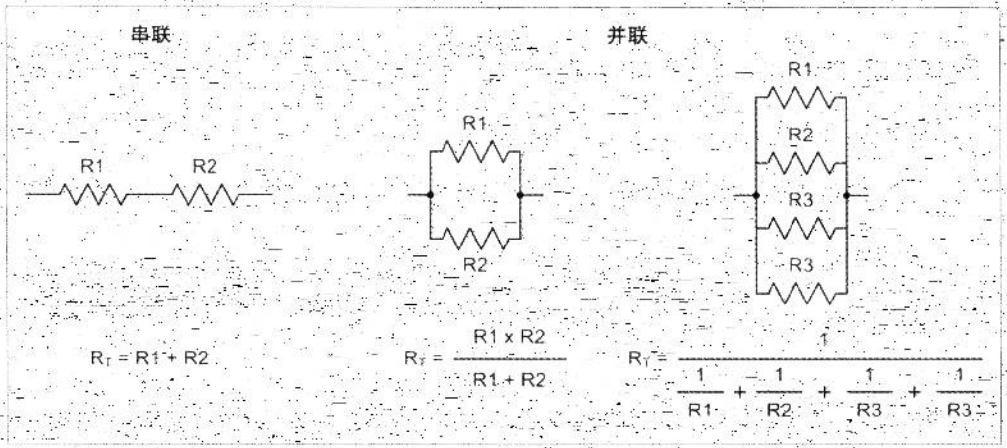


图2-12：串/并联电路

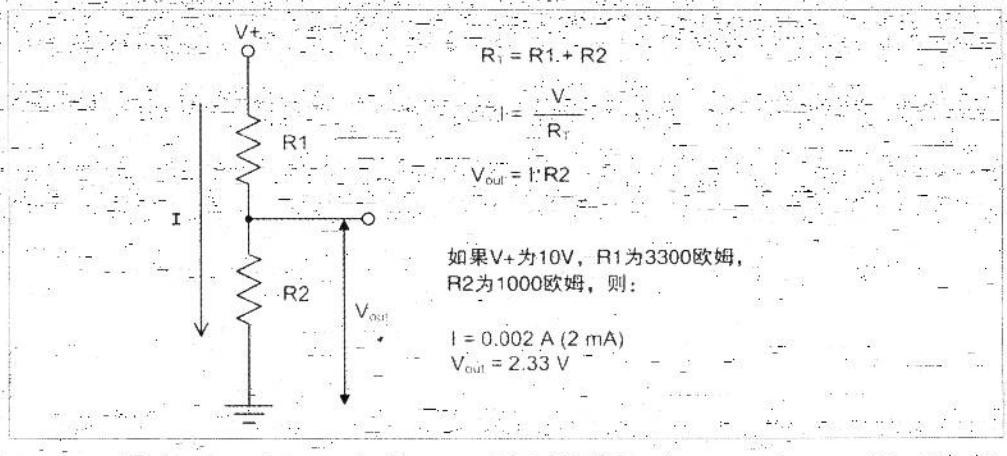


图2-13：分压电路

计算图 2-13 中的 *R2* 非常简单，只要知道总的阻值，就可以算出电流，由于经过两个电阻的电流一样大，因而只需简单应用 $E=IR$ 就可以求出 V_{out}。

交流电路

前面提到，电流方向随时间改变的电流称为交流电或 AC（Alternating Current）。相比直流电，交流电更复杂。除了和直流电一样有电压和电流的特性外，交流电还有频率和相位这两个特性。

在谈论住宅里的电源接线时，人们总会听到诸如“交流电”、“交流电压”或“交流电流”（电流在这里有点重复）这样的词。这些词通常分别指电源类型、电路的电压，以及电路中的电流。然而，当涉及仪器电路里的低电压、低电流交流电流时，通常只说“信号”或“交流信号”。

正弦波

交流信号可以多种波形出现，不过正弦波最为典型。由于正弦波是由单一频率所组成，因而是“纯净的”。别的波形能通过使用傅里叶分析技术分解为一组正弦波（在此不作深入介绍），而纯正弦波不能再进一步分解。

图 2-14 所示是一个标准的正弦波。由于在数学上正弦波是由下面的正弦函数所定义，因而得名。

$$V(t) = A\sin(2\pi ft + \theta)$$

其中，A 是振幅，f 是频率，t 是时间，θ 是相位。有时也会碰到下面这种形式。

$$V(t) = A\sin(\omega t + \theta)$$

式中的 ω 是角频率，实际上就是 $2\pi f$。

交流信号的一个基本特性是它的频率。频率是对信号在一秒钟内方向变化（也就是极性的变化）次数的度量，单位是赫兹（Hz）。信号频率 f 的倒数是其周期 t，它是重复波形之间的时间间隔：

$$f = 1/t$$

$$t = 1/f$$

例如，如果一台摄像机以 30ms/ 帧的速度生成画面，则其工作频率就是 33.33Hz。一个 60Hz 的信号的时间周期约为 16.67ms。一个频率为 10KHz（千赫兹）的信号，它的周期是 100 μs（微秒，通常写成 u 而不是 μ）。

31

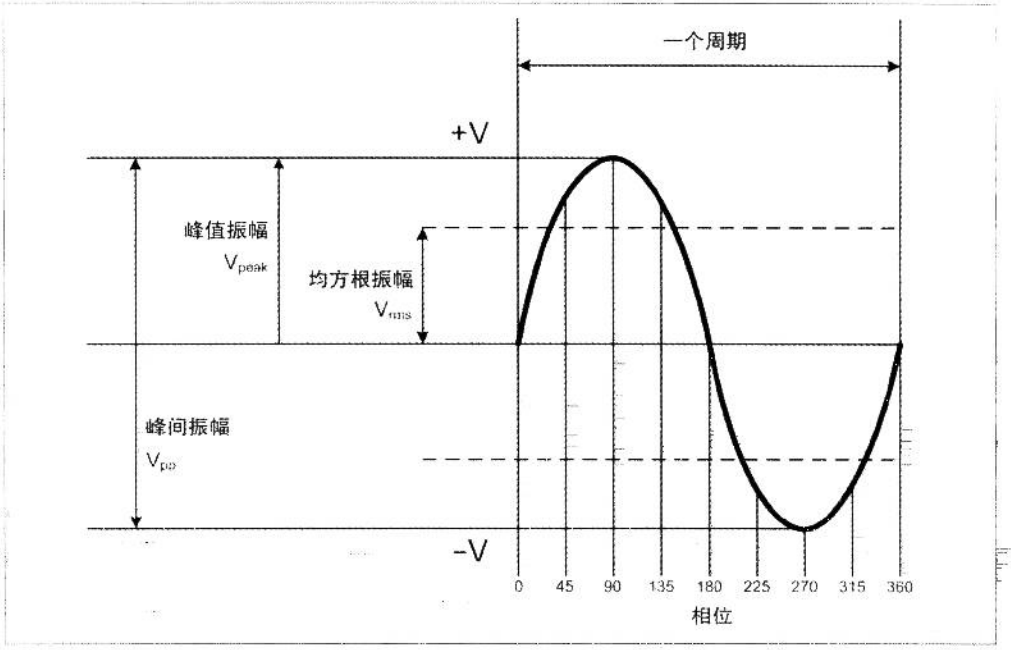


图2-14：正弦波

交流信号的另一个基本特性是其振幅。交流信号的振幅有三种描述方法：峰值振幅，峰间振幅，以及均方根（RMS，Root-Mean-Square）振幅。再次观察图 2-14，可以看到峰值（正弦波等式里的 A）指的是两侧相对于基准线的最大值。在谈到峰间振幅（写为 V_{pp}）时，指的是正峰与负峰之间的距离。最后，均方根振幅用来计算交流电路的功率（和直流电路一样，单位为瓦特）。对于正弦波，$V_{rms} = 0.707 \times V_{peak}$，对于别的波形将会是不同的值。

现在，我们来想一下：你屋里的交流电源比如说是 120VAC，这指的是它的均方根电压。V_{peak} 的值约为 165 伏，V_{pp} 的值约为 330 伏。你大可不必为 V_{pp} 的值而兴奋，但在选择用于交流电路的器件时知道 V_{peak} 的实际值为 165V 则会非常有用。记住，120VAC 的均方根电压主要用来计算功率。

32

电容器

电容器是一种无源器件，从本质上讲它是两块中间隔着小间隙的平行金属板。根据制造材料、容值、额定电压及尺寸的不同，电容器分很多种。有些电容器采用铝箔作为电极铝箔中间隔着同样薄的绝缘材料，另一些电容器采用浸在电解液里的多孔材料作为电极

间的分隔物；还有一些电容器以一片陶瓷材料加上两侧的金属化表面制成。图 2-15 所示是一只标准的电容器。

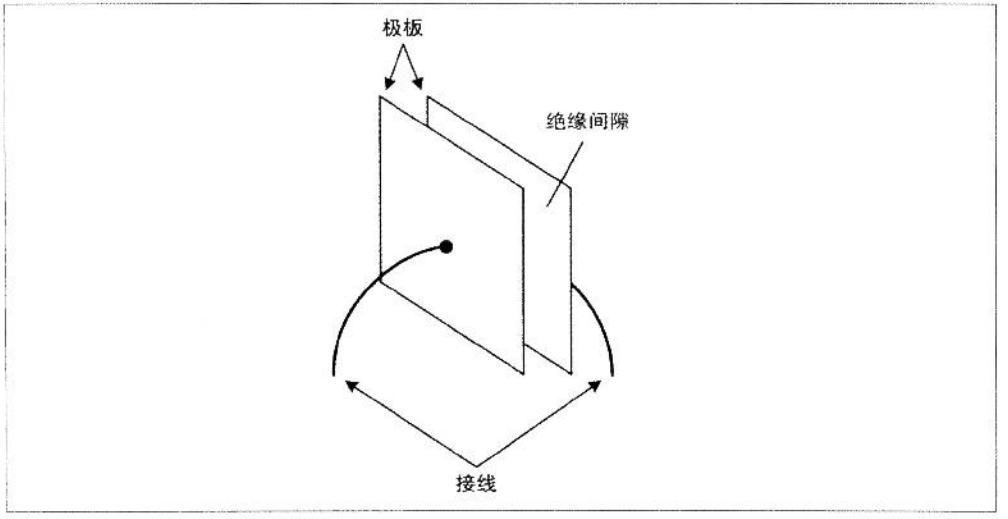


图2-15：电容器

电容的单位是法拉（F，以迈克尔·法拉第的名字命名）。在电子应用中会经常见到从数十皮法（pF，10^{-12}F）到几百微法（μF，10^{-6}F）的电容。有些特殊的应用还会用到几个法拉的电容。

电容器是一种存储电荷的装置，它和用于存储静电荷的莱顿瓶有几分相似之处（莱顿瓶于 1745 年左右在荷兰的莱顿大学被发明，因而得名）。电容器的容值取决于极板的面积，以及间隙的距离和类型。极板越大，可以存储越多电荷。极板间的距离决定了极板上电荷的相互作用强度。电容器只需用两块金属板，中间隔以空气即可制成；带有可移动金属板的这种电容器曾广泛应用于无线电设备的调节电路上，而且直到今天仍在应用。在电子电路中，多数小电容采用了一种可以让极板靠得非常近而又不会直接形成一条通路的介电质（绝缘）材料。

当给电容器加上电压时，其中一个极板会被充电带上一种电荷，同时另一个极板会带上相反极性的电荷。图 2-16 演示了这一情形。 33

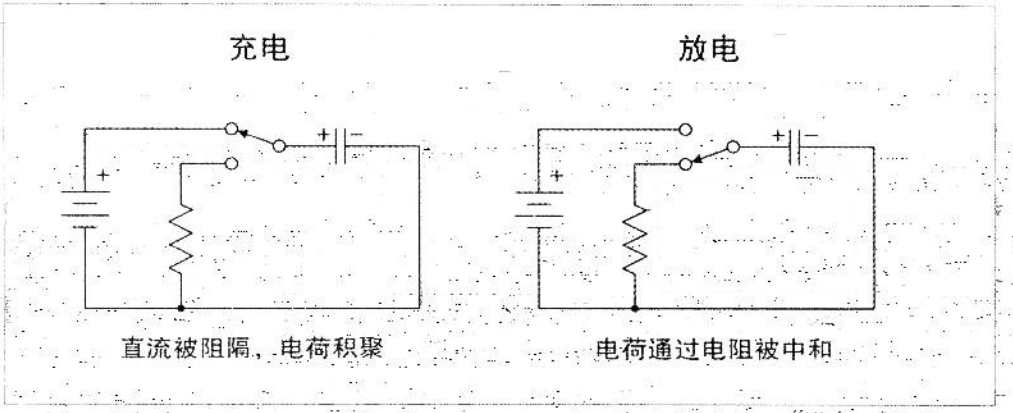


图2-16：直流电路中的电容

当图 2-16 中的开关被打到相反的位置时，电容器将会通过电阻释放掉先前蓄积的能量。你可能会想，可以把电容器看成是一种短寿命的电池。（事实上，“电池”这个词是由本杰明·富兰克林创造的，用来指前面提到的那种莱顿瓶阵列。）有趣的是，现在有容量为几个法拉的电容器，它们可以长时间存储大量电荷，在一些应用中确实替代了电池的角色。

电容器会阻止直流而让交流通过。交流信号通过电容器的的容易程度是关于容值、电路中相关部分的电阻和交流信号频率的函数。在图 2-17 所示电路中，交流电压源和直流电压源（如电池）共存于同一电路中。

从这张图里，我们马上就会看到一些有趣的事情。首先是交流和直流可以在同一条导线上同时存在（这实际上在电子电路里相当常见）。其次，交流信号会“搭乘”在直流电压上面，交流信号的过零点正好位于 V_{DC} 最大值水平上。根据具体环境，这通常被称为直流偏移或直流偏压。

注意在图 2-17 中，在 *R1* 上可以测量到 AC-DC 复合信号，但电容器 *C* 会阻止直流而只允许交流通过，因此在 *R2* 上测量将只能看到 AC 信号。事实上，我们忽略了电阻和电容之间的相互作用，它会影响电路将怎样响应不同的信号频率。换句话说，这是一种无源滤波器。

34 电容器和电阻器一样，也可以串联和并联，但是计算方法有些不同。图 2-18 显示了怎样计算串联或并联电容器的等效电容。

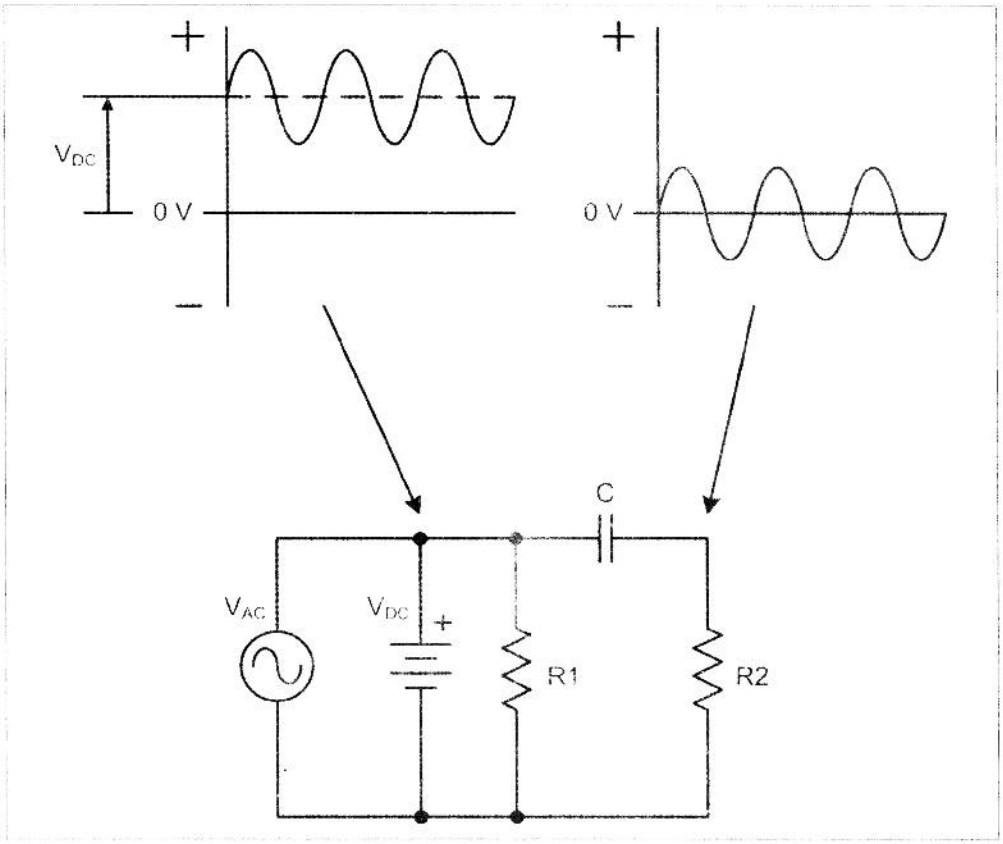


图2-17：直流阻隔

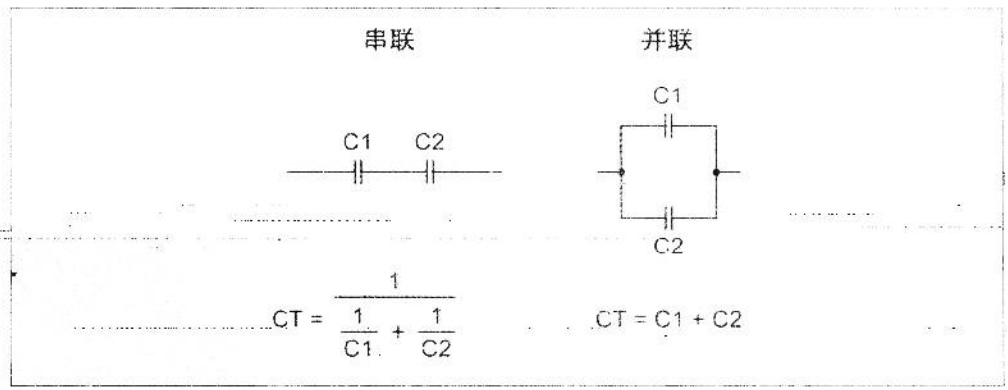


图2-18：电容的串联和并联

电容器阻止直流信号这一特性以各种有趣的方式被应用在仪器系统里。设想某人要用电磁转速计来测量一个旋转设备的每分钟转速（RPM）的场景。如果信号突然停止，不用去现场查看，怎样才能知道是转速计出了故障还是电路出了故障，或者是机械装置突然停止了呢？图 2-19 演示了一种解决方案。

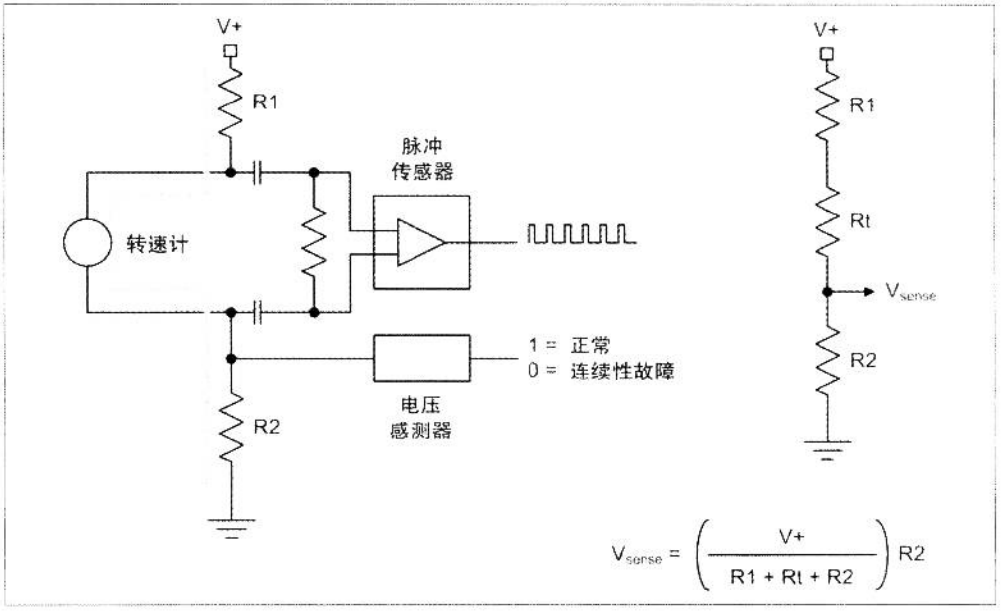


图2-19：转速表的连续性检测

在图 2-19 中，右侧的电路是等效直流电路，其中 *Rt* 是转速计自身的直流电阻，这个电阻大约只有几欧姆。电容器阻止直流，*R1*、*R2* 和 *Rt* 构成了一个分压电路。如果转速计出现故障并断开电路，则当电压检测器的输入电压为 0 时，它将监视到故障发生。反之，如果电压突然升高，则转速计可能发生了短路。最后，如果信号消失但 *R1*、*R2* 和 *Rt* 依然处于连通状态，那么不是脉冲传感器电路发生了故障就是出现了机械问题（也就是说某个东西被卡住了）。

36

电感器

电与磁密切相关（深层关系依然没有完全弄清楚）。流经导体的电流会产生一个磁场，同时变化的磁场会在导体中感生出电流。因此，当电流经过一根导体时，导体的周围就会相应产生一个磁场，如图 2-20 所示。如果是直流电，则磁场将一直维持不变，直到电流停止（此时磁场也将突然消失）。在磁场消失时，将产生一个与最初生成磁场的电流方

向想反的电流（消失中的磁场是变化的磁场）。要是把导线绕成一个线圈，则磁场将被集中在一起。使用铁芯会进一步增强此效应。

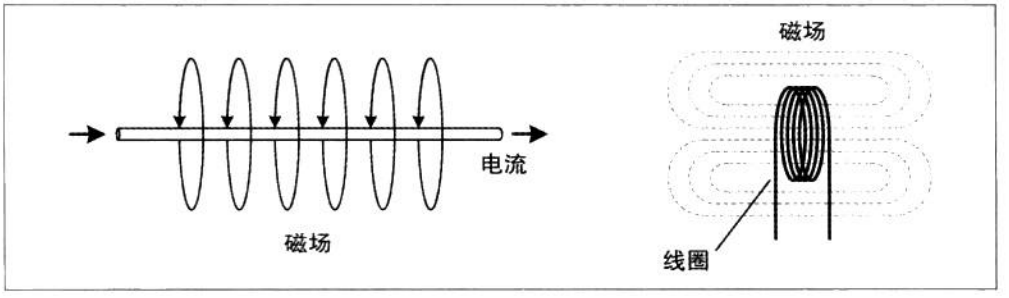


图2-20：电感器

给一个线圈通上直流电就成了一个电磁铁。另外，交流电磁铁也比较常见。在继电器、螺线管和电动机中都能找到电磁铁的身影。

当把线圈应用到电路中以改变电路工作行为时，对于交流信号，线圈会表现出阻抗随信号频率变化而变化的有趣效应。线圈对直流信号的唯一效应（磁场的产生和消失除外）是绕制线圈的导线的基本电阻。

线圈（也称为电感器）对交流信号的频率依赖响应被称为线圈的电感，其单位是亨利（H，因约瑟·亨利命名）。在电子电路中，最常见的单位是毫亨（mH）。

从前面的描述可以知道，单纯的一段导线也是一个电感器（尤其在高频时）。这种效应会使脉冲或方波那样的信号在长距离通信时衰减。

电感器在直流电路中基本上相当于一个短路（或者是一个非常非常低的电阻），此时它显得并无特别之处。然而，在交流电路中，电感器会在每次信号由峰值振幅（正或负）回到零时产生一个反向电流以阻碍交流电（阻抗一词因此而得名）。阻抗的大小是关于线圈的电感和交流信号频率的函数。 37

虽然感性电路和电路理论中的其他主题相比同样重要，但本书不打算对其细节做深入研究。通常情况下，对此不必过分担心，需要时我们会再来学习。有兴趣（或好奇心较强）的读者可以查阅章末的参考书了解更多的信息。

正如电流流动会产生一个磁场，变化的磁场也会产生电流。导体切割磁力线时，将会有电流出现。如图 2-21 中简化的交流发电机所示。

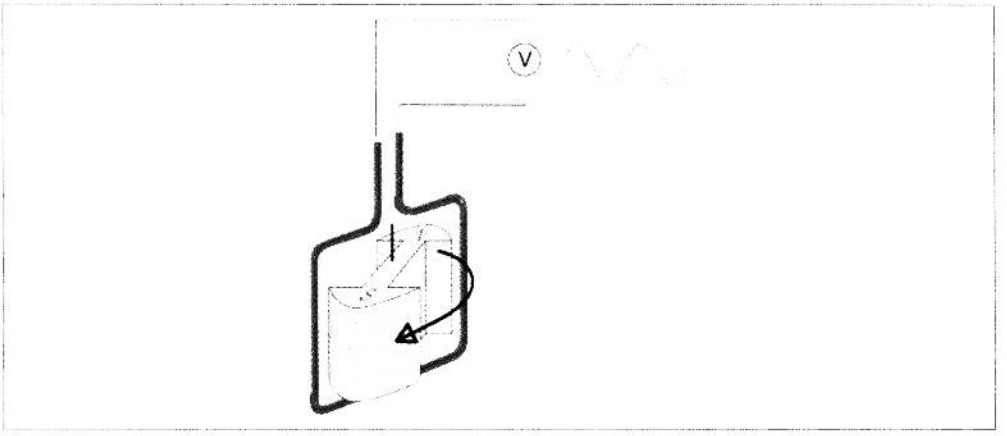


图2-21：交流发电机（Egmason作图，Wikipedia Commons，基于Creative Commons Attribution 3.0 Unported许可：http://commons.wikimedia.org/wiki/File:Alternator_1.svg）

在图 2-21 中，磁铁旋转时在导线中将感生出一个交流电压。这个装置还可以线圈绕着固定的磁场旋转的方式布置，最终的效果是一样的。这种效应在仪器里有某些有用的应用，尤其是当要测量某个机械装置的转速时。图 2-22 演示了电磁转速计背后的基本设想。

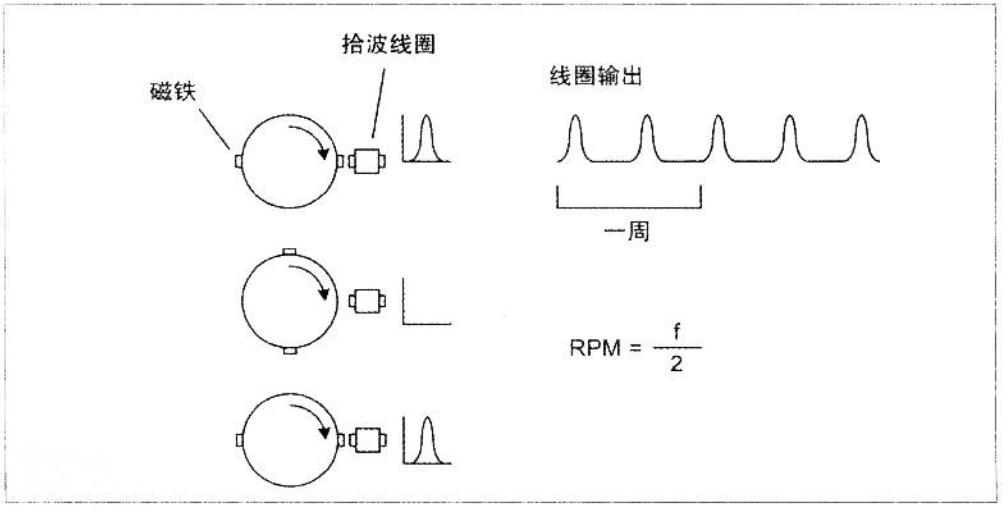


图2-22：电磁转速计

在图中，线圈的输出信号被理想化了，真实的信号看起来和这个有很大的区别。然而设想是一样的：当磁铁经过线圈时，在线圈上将产生一个电压脉冲或尖峰。图中用了两块磁铁，是因为通常要让旋转的设备或轴上的东西保持平衡，尤其当机械装置以高速旋转

时更是如此。由于有两块磁铁，因此每两个脉冲输出表示机构转了一圈。

其他波形：方波、斜波、三角波和脉冲

38

除了正弦波，在电子电路里还有一些别的常见波形。图 2-23 展示了一些基本的波形。

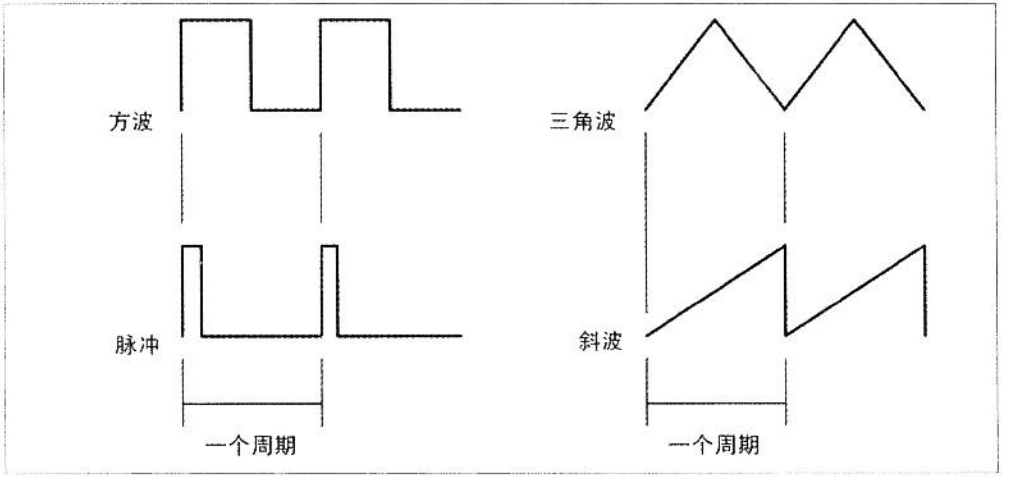


图2-23：常见信号波形

当然，还有一些非常复杂的波形，如音频信号。音频信号之所以复杂，是因为它包含有多种不同频率的信号。我们将主要关注在仪器电路里常见的那些波形。

39

最常见的（也最有用的）一种非正弦波形是方波及与它近似的脉冲波。虽然通常把方波画成瞬时接通和断开的样子，但其实方波要杂乱得多，如图 2-24 所示。

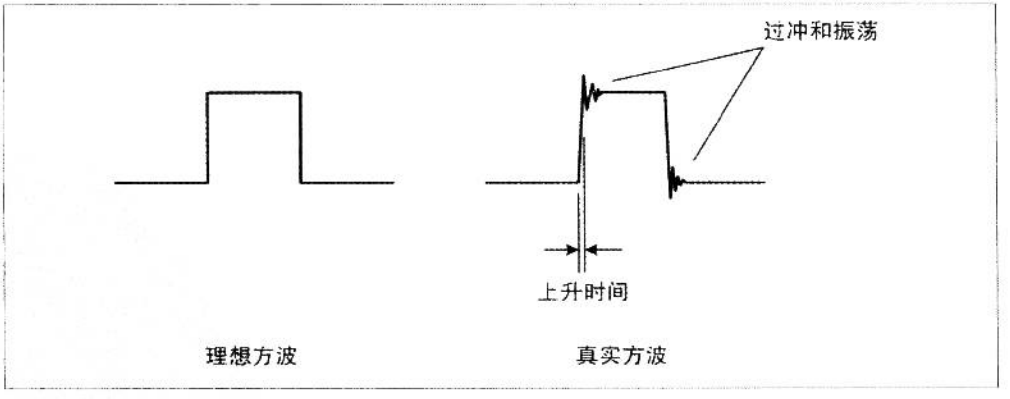


图2-24：理想与现实的方波

电路中的各种阻抗和电容效应导致了过冲和振荡的出现。由于导线具有内磁感应强度(如前所述)，通过一条超过几英尺的非屏蔽线传送脉冲或方波将导致在接收端出现一个退化信号。稍后将会看到，有办法可以应付或者至少减轻这一效应。

在与脉冲和方波打交道时，常常会提到波形的占空比。事实上，方波在本质上是一种占空比为 50% 的脉冲波，也就是说其一个周期的 1/2 是高电平。图 2-25 演示了几个不同占空比的脉冲波。

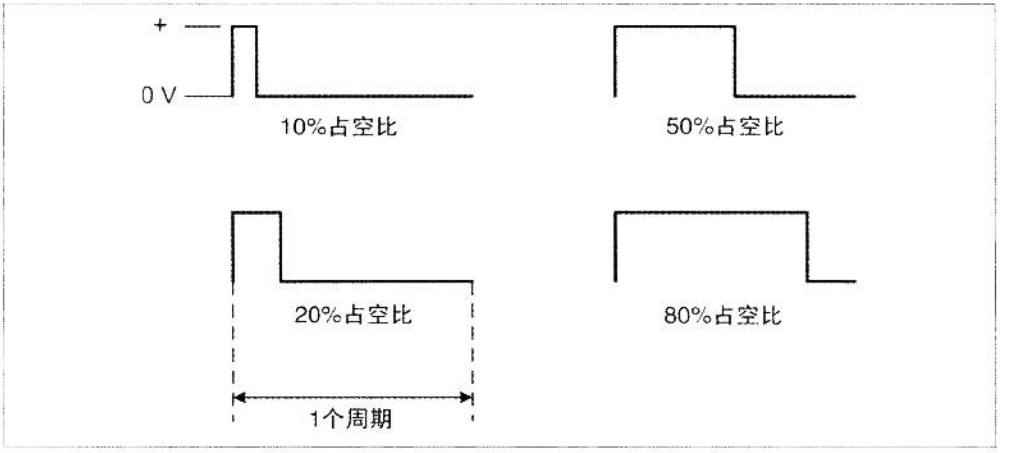


图2-25：占空比

接口

在仪器系统里，接口是软件与现实世界的交汇之处。接口可以是检测开关闭合状态的一个输入，来自温度传感器的电压值，或旋转轴上磁性传感器输出的脉冲；也可以是与某个独立仪器或某种类型的控制器相接的数据通信通道；甚至可以是另一个计算机系统，以太网接口就是一例。

40

离散数字 I/O

离散数字接口因输入 / 输出数据位（也称引线或引脚）的组织、读取和控制方式而得名。每当需要对离散输入状态进行检测或是要对具有离散运行状态的事物进行控制时就可使用离散数字 I/O。“离散”在这里指每条输入 / 输出线具有有限个唯一非连续状态。对于离散数字接口而言，这等同于两种状态：接通或断开，1 或 0。在软件中，离散 I/O 线既可被当成单个位值处理，也可被当成一组位的成员进行处理。在某些情况下，任意一条

线路都可被设置成输入或者输出。在其他情况下，接口会专门规定一组引线用于输出，而另一组则用于输入。就术语而言，常说“离散 I/O”或“离散线路”。

离散 I/O 线通常按 8 的倍数分组，8 是一个字节的大小，但是也可以构建任意条离散 I/O 线的接口（当然要在一定实际限制之内）。当离散接口按 8 位分组时，每 8 根线构成的一组被称为一个端口。

前面的说法存在大量不明之处，因为一个接口具体是怎样组织的完全取决于设计它的工程师。然而从微控制器的角度看，把 8 条离散线当成一个端口是微控制器内部架构的自然延伸。图 2-26 展示了怎样实现一个 8 位离散 I/O 端口。在廉价的微控制器和可编程逻辑器件出现之前，这是一种常见的接口实现方法，而且在今天依然在使用。

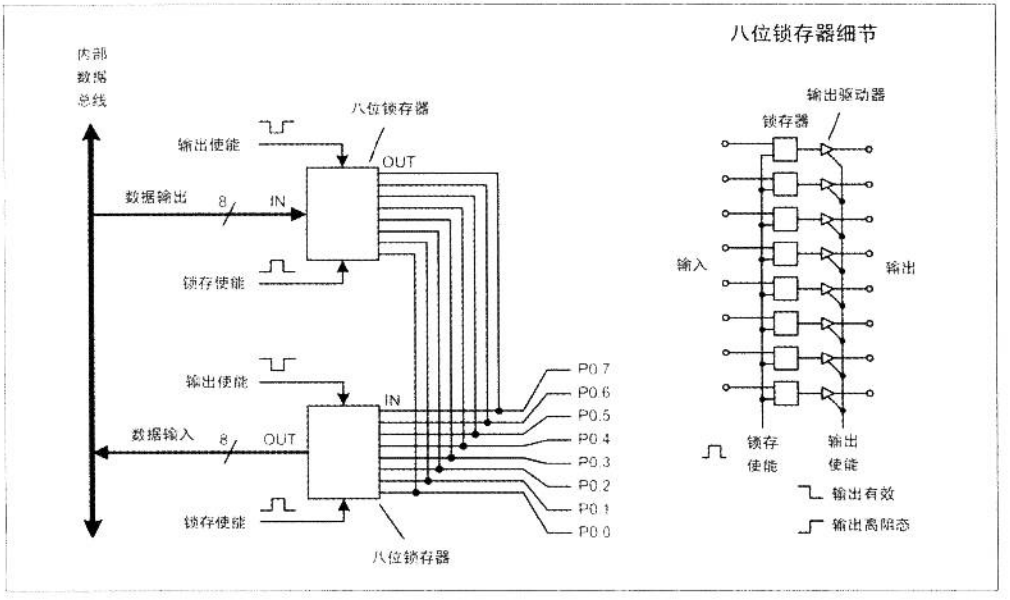


图2-26：离散I/O端口

在图 2-26 中，有两组 8 位信号（一个字节大小）分别连接到 8 位输出锁存器上，另一组 8 位信号从 8 位输入锁存器中出来。换句话说，该接口一次能读或写 8 位数据，或者说是 1 个字节。图中还有内部数据总线，它们会在某处被连接到主数据总线上。进入输出和输入锁存器的几个信号被用来控制锁存器何时可以对输入数据进行采样（锁存器使能信号），以及何时真正让数据出现在输出上（输出使能信号）。当锁存器的输出未使能时，它们将处于高阻态（Hi-Z），实际上相当于一个开路。这使得接口可以作为输入或输出口，但不能同时处于两种状态。设置输出端为高电平而试图读取可能处于低电平的外部设备

没有意义。即便没有东西过热，仍然没必要这样做。然而，在图 2-26 所示电路中，写数据到输出端然后读取回数据是可能的。有时这样可验证任何 I/O 端上都不存在对地的短接。

有些接口设备和一些微控制器，可将离散 I/O 单独设置成输入或者输出。此时，I/O 线可以在软件中独立寻址，而不是把 8 位当成一个群组一起使用。假设接口电路具备这种能力，就可以在软件中使用类似 P0.0、P0.1……一样的名字来指定特定的引脚（在这里指定的是端口 0 上的引脚）。

42 离散接口常采用标准的晶体管 - 晶体管逻辑（TTL）电平，当然有时也采用其他的电平。在 TTL 电路里，任何低于 0.8V 的电压都被认为是 0，介于 2.0V ~ V_{cc}（假设直流电源电压是 5 伏）的任何电压都是逻辑 1。0.8V ~ 2.0V 是无效的。为保证输入电平不会漂移到无效区域，常常用电阻把输入上拉或下拉。如图 2-27 所示。

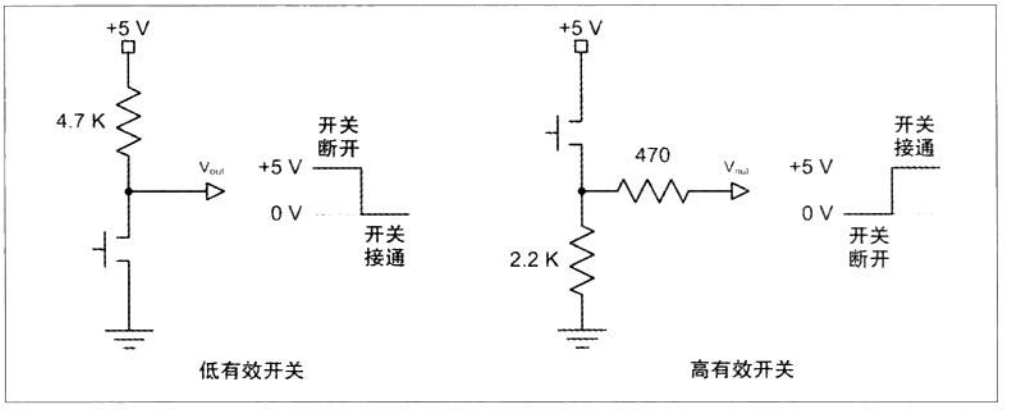


图2-27：上拉和下拉电阻

在图 2-27 中，当按钮开关闭合时将使 V_{out} 变为 0，在 4700 欧姆的电阻（图中标为 4.7K）中将会流过一个 1mA 多一点的电流，该电流在多数情况下都可忽略不计。更为重要的是，该电路会向输入数字口提供一个恒定的电压源，因此在开关闭合并将 V_{out} 拉低前（开关此时实际上相当于短路）输入将保持为逻辑 1 的状态。高有效开关电路的功能正好相反，虽然我们在此处试图在开关闭合前让 V_{out} 保持低状态。当开关断开时，数字输入端连接到 V_{out} 上，V_{out} 通过两个电阻被下拉到地。当开关闭合时，+5V 通过 470 欧姆的电阻被加载到输入上，输入端将检测到逻辑 1。470 欧电阻把流入到数字输入端的电流大小限制在约 11mA 左右。尽管这个电阻并不是绝对必要，但加上它有一定的妙处。如果没有这个电阻，电源的全部电流容量（可达数安培）就会暴露在连接到其上的电路的输入上。在开关闭合时，在 2200 欧姆的电阻上将流过约 2mA 电流。

在使用机械开关作为输入时，应时刻谨记开关常常会发生接触抖动或接触振动。换句话说，触点的闭合很少会干净利落地完成。与此相反，开关里的触点相互会在开关闭合时来回弹跳一会儿。这会导致在开关完全稳定在一个状态之前，会产生一系列短促的接通/断开事件。对于各种不同的开关，这个时间间隔通常在几毫秒到 100ms 左右不等。为避免响应接触抖动，要对输入进行“去抖动”处理，可以用逻辑方法，也可以在软件中去抖。

图 2-27 中的开关可以是一只简单的可移动部件上的限位开关，也可以是用于检测传送带上移动目标的光学传感器，甚至是大桶或烧杯里的浮球开关。上拉电阻在帮助减小长导线上的信号退化时也很有用，其工作方式是发送方工作时主动把信号拉低，否则数据线处于高电位，如图 2-28 所示。这种方法有助于防止引入噪音，也有助于让电压从高到低和从低到高的转变更准确无误（也就是说信号将不易在高电压和低电压之间浮动或漂移）。

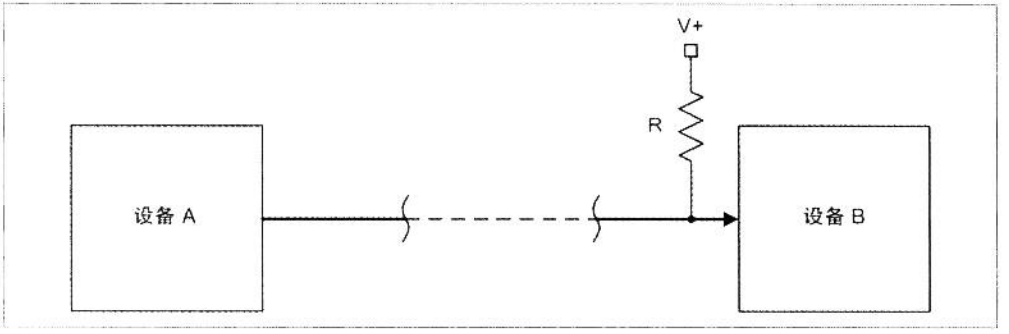


图2-28：信号上拉

电阻 R 的值在很大程度上依赖于设备 A 拥有的输出电路类型。某些逻辑器件有着被称为集电极开路的输出，它们被设计为和外部上拉电阻一起使用。对于别的没有集电极开路输出而同样可以使用外部上拉电阻的设备，要保证 R 的值不会导致该设备的灌入电流超过它的额定值。

我们已经见到如何用离散数字输出来直接控制 LED（回想图 2-9），但如果使用恰当的接口电路，数字离散输出同样可用于控制别的有趣的东西。其应用包括用来通断高电压、大电流的继电器，电磁阀，机械驱动器，或者电力制动器，等等。

图 2-29 展示了怎样使用离散输出来驱动继电器。离散输出通常只能输出大约 10 ~ 20mA 的小电流，但继电器需要更大的电流。如果将继电器直接连接到输出上，要么继电器不能工作，要么会导致离散 I/O 接口损坏，或者两种情况同时出现。

44

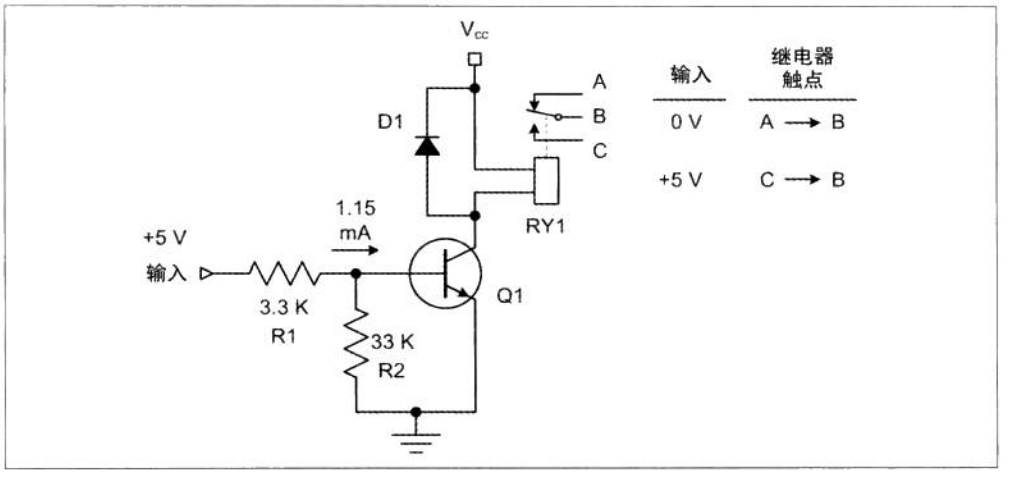


图2-29：继电器驱动电路

在图 2-29 中，使用了一个 NPN 晶体管作为电流开关。在这类电路中，选用哪种类型的 NPN 晶体管其实并不要紧，只要是被称为“小信号”的类型，并且足够应付通过继电器线圈的电流就行。对小型低功率继电器来说，选用 2N2222A 就足够了。当晶体管的基极（NPN 的 P 部分）为正时，晶体管就会导通，它能够控制比离散接口自身大得多的电流。*R1* 和 *R2* 构成了一个分压电路，当输入端的电压为 5V 时，在晶体管输入端上（基极）将有约 4.5V 的电压，电流约为 1.15mA。当没有输入电压时，*R2* 充当下拉电阻以确保晶体管处于截止状态。*R1* 是限流电阻，它提供足够驱动晶体管而又不至于让它损坏的电流。二极管 *D1* 用于阻止继电器线圈产生的反向电压尖峰回到晶体管，这个电压几乎肯定能够将其摧毁（回想我们前面讲到的当电感的电流被关断磁场突然消失时，将产生一个反向电流）。借助于继电器，现在我们已经掌握了能够控制大电流的电路，同样的电路也可用于驱动各种大电流负载。

模拟 I/O

用来将模拟数据转换为数字量的器件被称为模 - 数转换器（ADC）。反之，将数字量转换为模拟电压或电流的器件被称为数 - 模转换器（DAC）。这两种功能模块曾经非常昂贵，

45

而且必要的硬件要占据许多电路板和很大的机柜空间。如今许多廉价的微控制器自身已经内置了 ADC 和 DAC 功能。ADC 和 DAC 器件的电压范围和分辨率千差万别；对于某些应用，选用外部高分辨率且能以高转换速度工作的器件更为合理，然而即使是这些器件也不是特别贵。

模拟数据对于计算机来说是个疑难问题。因为计算机的离散数字本性，捕捉模拟数据并将其转换成数字形式，并且要达到 100% 忠实表现原始信号的精度水平是根本不可能的。

在将数字量转换为模拟信号时也是如此。这种效应被称为量化，并作为在离散时间点上获取或产生一个连续变化信号的一系列测量的结果出现。

模拟数据采集

当输入为模拟信号时，在采样事件之间的任何模拟信号上的变更将被彻底丢失，其结果对原始信号的忠实程度仅取决于每单位时间允许的采样个数。如图 2-30 所示，这幅图显示了对同一信号每两秒一次采样和每秒两次采样之间的差别。注意，更高采样率重建的结果更接近原始信号，但仍然不是 100% 的完美重建。

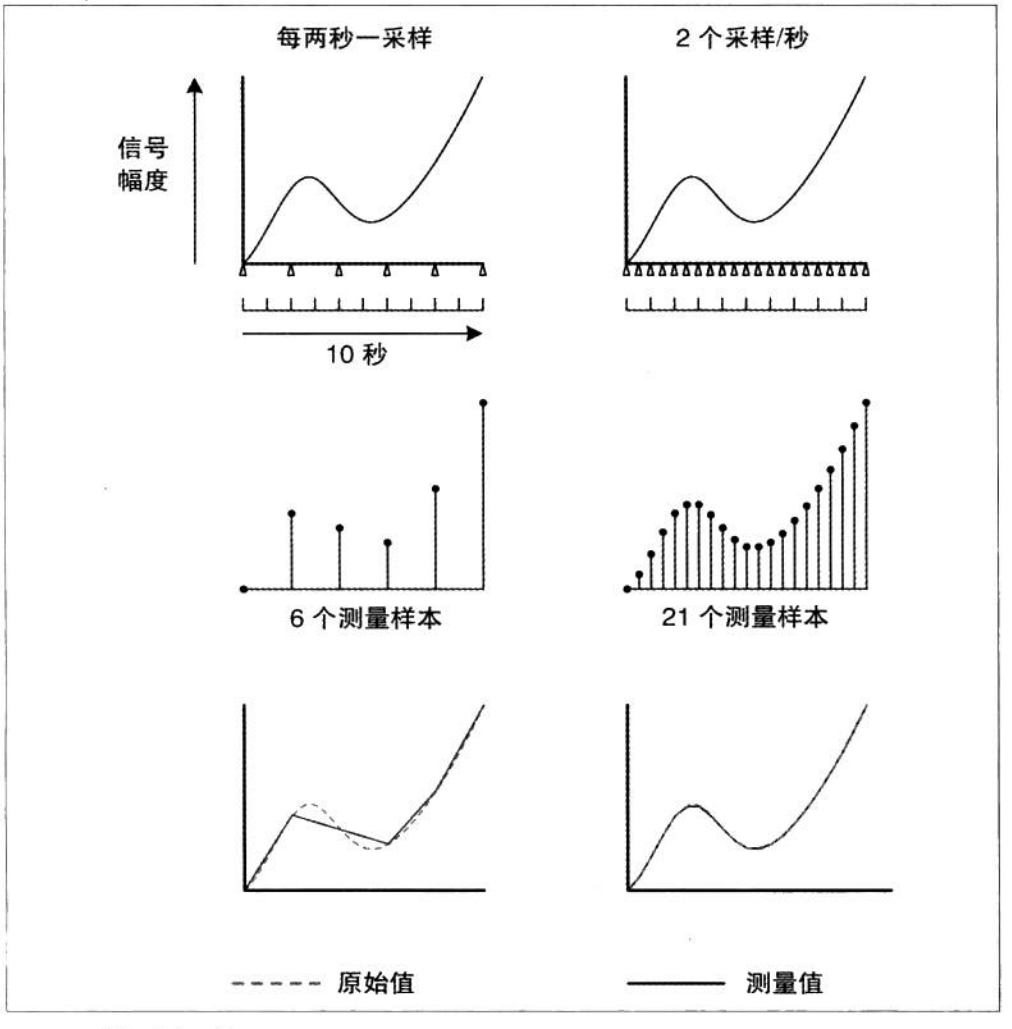


图2-30：模拟数据采样

当然，不是所有的应用都需要高水平的忠实信号来达到仪器的目标。在许多情况下，以几秒钟甚至是几分钟的时间间隔进行数据采样都是完全可以接受的。在那些测量输入变化不快的应用中尤其如此，例如电炉、空调系统或者农业孵化室。一些其他的应用，例如音频到数字信号的转换，需要非常高的采样率以便可以对感兴趣的最高频率进行准确的捕捉并对原始输入维持高保真的重现。

模拟数据通常以 8 ~ 24 位的分辨率或数据大小把每个采样转换成数字形式。小于 8 位或高于 24 位的分辨率不易得到，但也是可能的。分辨率为 8 位时，数据的取值范围是 0 ~ 255（或是 -128 ~ +127，如果用负值的话）。重复说一下，不是每个应用都总是需要很高的精度，有时不高的精度就绰绰有余了。

在把模拟量转换成数字量（或相反的操作）时，会在转换中遇到叫做量化误差的固有（不可避免）限制。简言之，它是转换中造成的原始信号和数字值（或编码）之间的误差。如图 2-31 所示，ADC 分辨率越低量化误差越显著。

根据图 2-31（仅为了讨论目的的一个近似），9 位的 ADC 有 512 个可能的值，它将产生
46 比有 256 个可能值的 8 位 ADC 更准确的转换结果。时间轴上的 T_{S0}、T_{S1}……是采样事件（采样率）。请注意，8 位转换器是否以较高速度采样并不要紧；这样虽然能够对在其分辨率之内的输入的快速变化进行探测和转换，但它不能达到比其基本的 8 位分辨率更好的效果。

在图 2-30 中，在较低采样率时精确性的丧失增加是因为缺少精确追踪模拟信号改变的采样，而不是转换分辨率不够高（实际上还未曾提到分辨率这个概念）。在图 2-31 中，由于量化误差而引起的精度的丧失是因为分辨率不够高。

47 采样率和采样分辨率一起决定了 ADC 可以怎样的精度把模拟信号转换成数字形式。采样分辨率能够以 V/ 步表示，或者换种说法，就是在转换器分辨率范围内每两个离散数字值之间的可测电压。如果我们有一个其最大满量程输入范围是 0 ~ 10V 的 8 位转换器，数字输出码的每个增量或步长将等效于 0.039V。这可以表示为

$$\text{分辨率} = \frac{V_{\max}}{2^n - 1}$$

48 因此，一个 $V_{\max}$ 为 10V 的 10 位转换器能够分辨 0.00978V/ 步，一个 12 位的器件能分辨 0.0024V/ 步，16 位 ADC 能够分辨 0.0001526V/ 步。

现在这些关于 ADC 的信息已够足以开展我们的工作，因此接下来我们将考查 ADC 的逆过程，也就是数字 - 模拟转换器。关于 ADC 装置的更多信息及其行为，请参考本章末尾列出的参考资料。

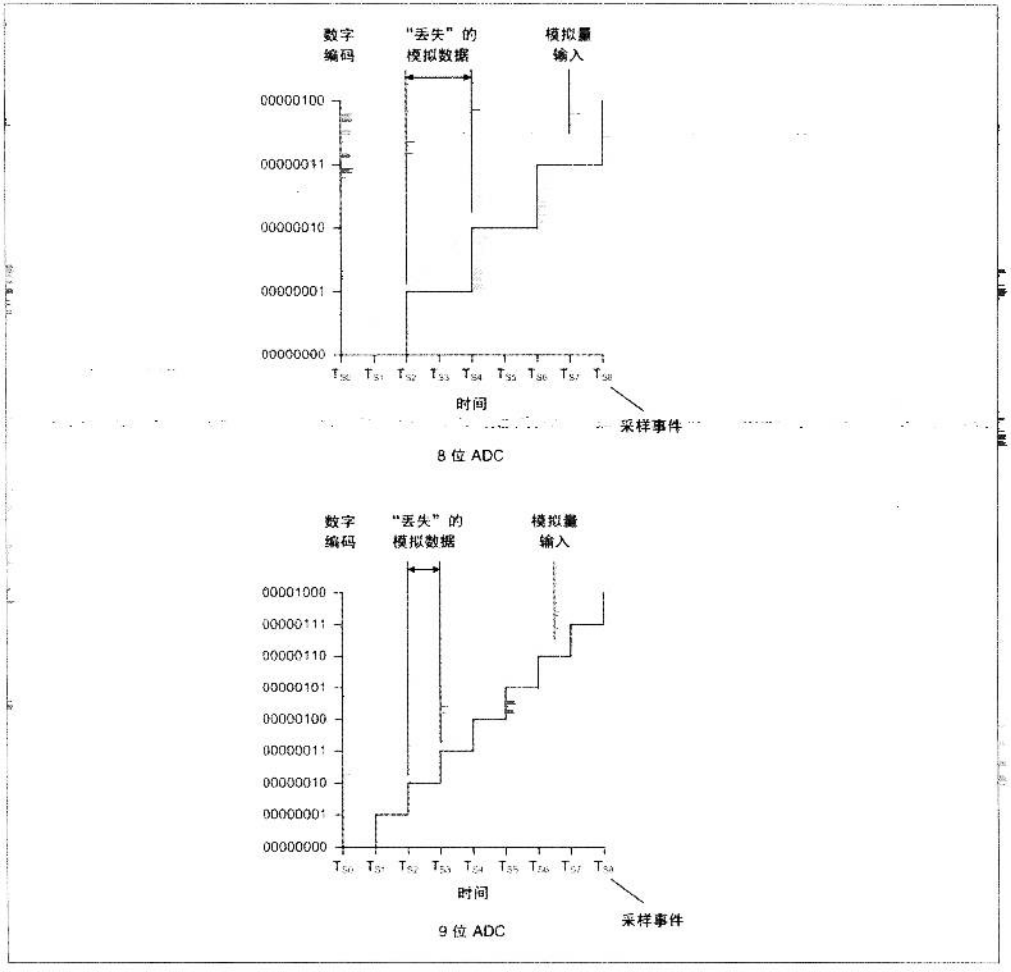


图2-31：ADC的量化误差

生成模拟量

和 ADC 相对应的是 DAC，也称为数 / 模转换器。这些器件会产生和数字输入值相对应的电压输出。和 ADC 类似，DAC 也有一些和分辨率相关的固有局限，这些器件同时也有量化问题。

图 2-32 展示了分辨率和输出更新率 / 采样率之间的关系。

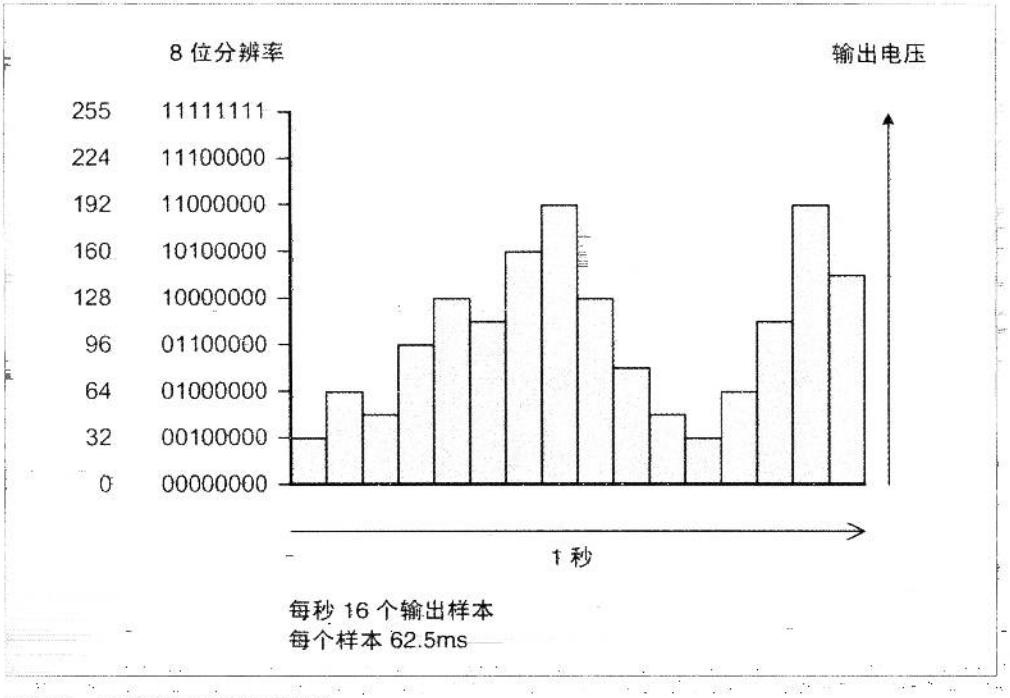


图2-32：DAC输出时序和分辨率

对于很多仪器和控制应用，输出采样率并不是一个紧要的参数，而且每秒一次或两次的样子就能够满足要求。当然，这假定了 DAC 要控制的对象不需要以比较高的速度变化。

对某些 DAC 装置而言，输出电压是使用一个外部参考电压来完成的。有些 DAC 器件把参考电压集成在自身内部。输出分辨率取决于用来生成输出值的位数，它等于输出电压范围除以可能的数字输入值的个数。输出的实际精度是关于设备线性度的函数。

计数器与定时器

计数器与定时器在很多数据采集和控制实现里属于基本组件。图 2-33 所示是一个带有运行 / 停止和清零控制输入的 16 位通用计数器。当运行 / 停止输入为高时（逻辑真），每检测到一个输入脉冲，计数器内部计数值都会相应加一。否则，脉冲输入将被忽略。清零输入可以让内部计数器被复位为 0。计数器并不要求脉冲输入必须连续，它同样可对随机事件进行计数。从图 2-33 可以看出，利用运行 / 停止信号在运行状态的时间长短可以算出输入脉冲的速率（也就是频率）。

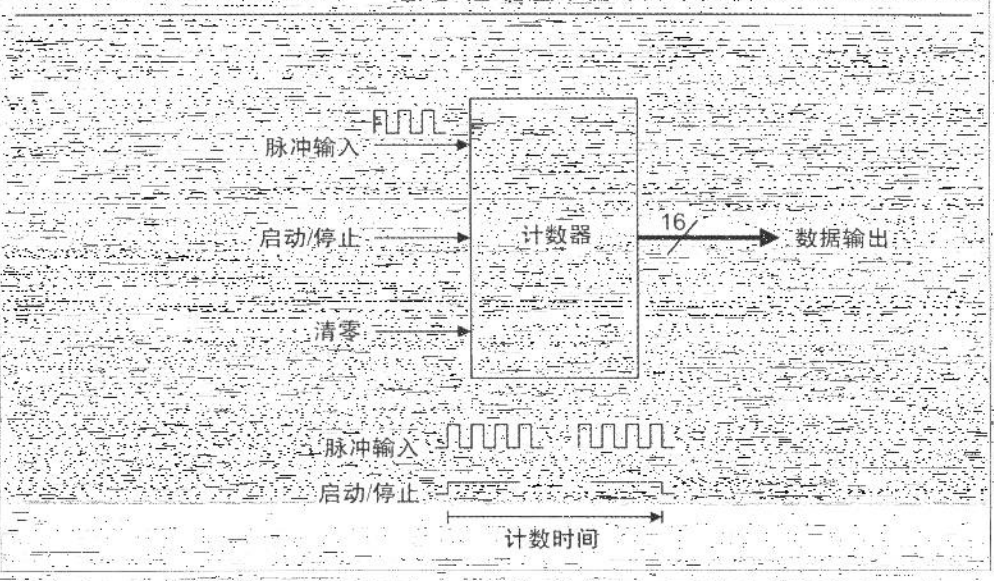


图2-33：计数器

定时器是一种内置计数器的装置，只是该计数器并不用来对外部事件进行计数，它用来正数或倒数，直到达到某一给定的数值。图 2-34 展示了一个用于单脉冲输出的普通定时器，以及通过将输出连接到复位输入的用于连续脉冲输出的版本。

50

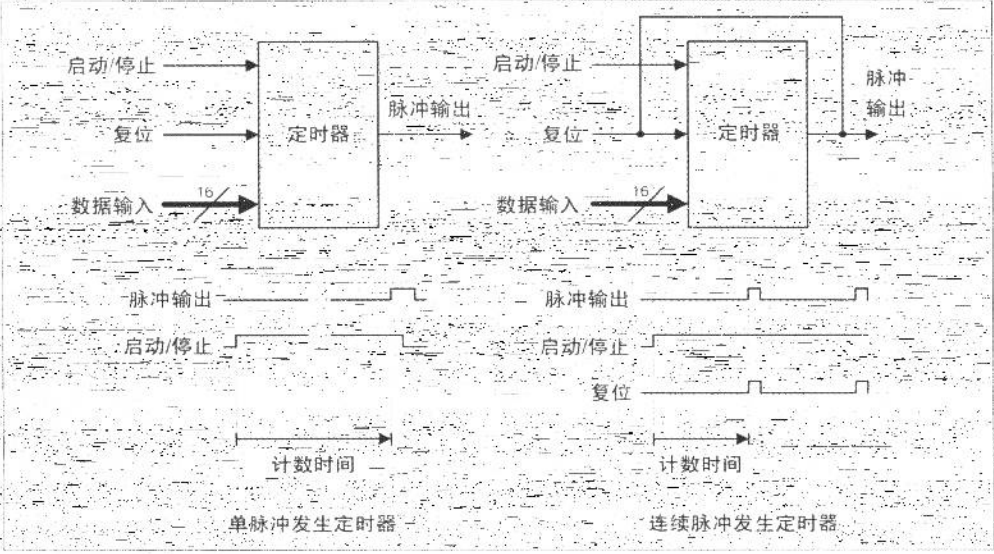


图2-34：定时器

16 位的计数值通过数据输入线被载入到定时器中。运行 / 停止输入控制计数动作，复位输入将计数器复位到 0（如果是正数）或者是预设值（如果是倒数）。某些定时器还允许从外部读取当前的计数值。

脉宽调制

脉宽调制（PWM）是一种常见的输出信号形式，常用于控制灯光、电机和其他设备。PWM 信号是一种可变占空比脉冲，如图 2-35 所示。当占空比增加时，功率的平均值也会相应地增加。

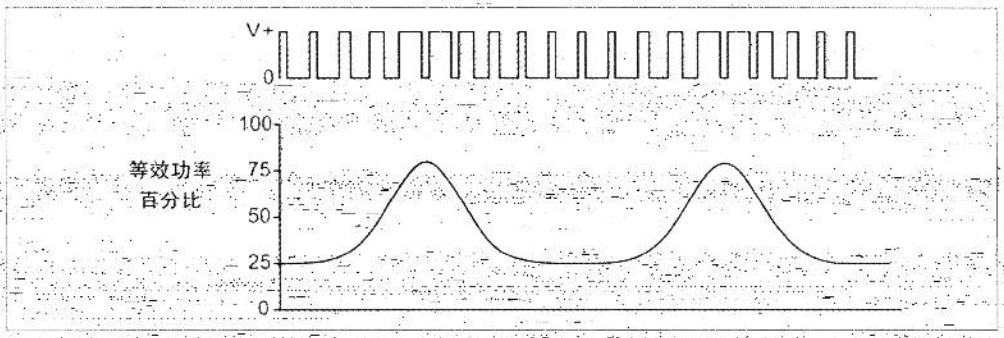


图2-35：脉宽调制

PWM 在直流电机的高效率控制中特别有效。直流电机的转速与提供给它的功率成正比。通过采用 PWM 而不是晶体管一类的有源线性控制器件，可以极高的效率调制平均功率。PWM 信号可以不借助 DAC 而直接由数字器件生成。图 2-36 是一个 PWM 直流电机控制框图。

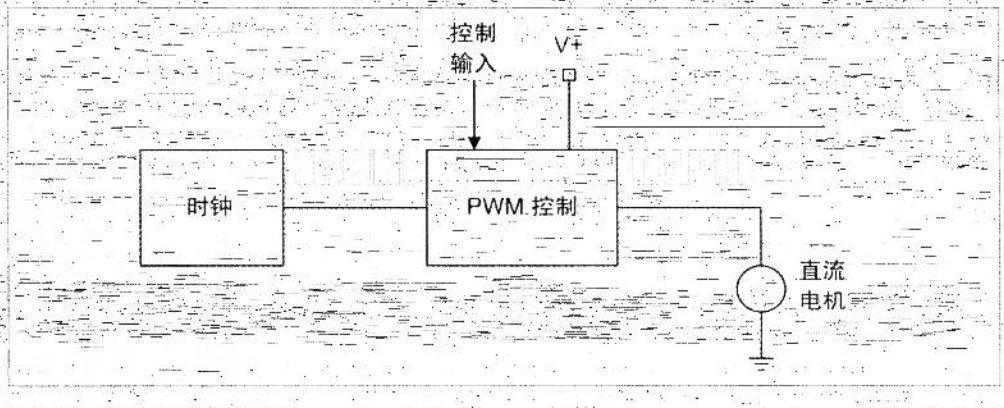


图2-36：PWM直流电机控制

串行 I/O

一个接口，如果其中的数据以一串数据位的形式通过一个单一的路径，那它就是串行接口，如图 2-37 所示。串行数据通常有两种实现方法：同步和异步。在同步串行配置中，有一条或二条线用于传送数据，一条（或两条）线用于时钟信号。当有效数据位出现在串行线路上时，时钟信号用来通知两端的电路。一些微控制器和 I/O 集成电路上的底层串行外围接口（SPI）就是同步串行接口的例子。

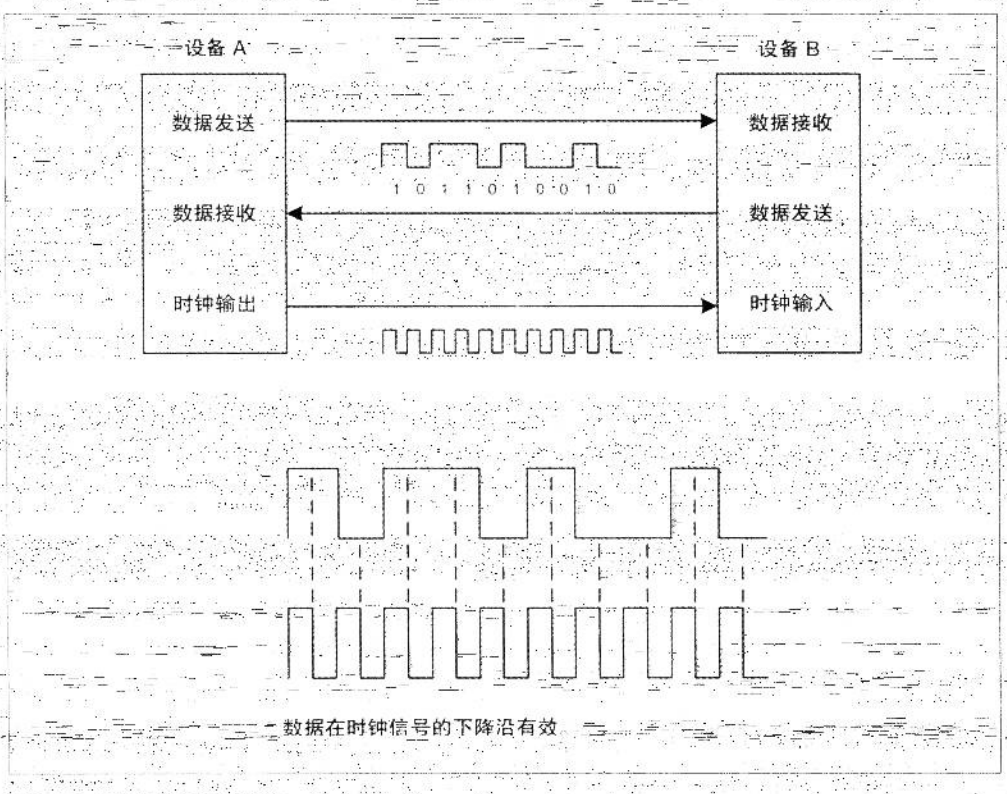


图2-37：同步串行数据通信

注意图 2-37，其中的设备 A 是时钟信号源，因此控制着接口数据交换的速率或速度。设备 B 只有当时钟有效并且被通过某个特殊的器件选定输入（图中未显示）选定时才发送和接收数据。图中显示了数据在时钟信号的下降沿有效的情况。同样请留意在图 2-37 中，时钟信号下降沿出现在数据位的中间位置。回想一下图 2-24 中的“真实”方波：可以看出在数据位串流中的方波的糟糕边沿之间对数据进行采样（或者说是“锁定”）降低了出错的风险。

在SPI接口中，通常在当主设备希望开始一个和从设备的通信时时钟信号才激活。别的类型的同步接口的时钟信号可能在整个过程中保持激活状态，还有一些接口会为通道两端都设立独立的时钟信号。

在把计算或仪器设备彼此相连时，一种更为常见的串行接口是异步串行接口。在这种通

信方案中，在通信设备之间不存在时钟信号，因此接收器电路必须自己同步数据。为此，串行位串为每个数据字符附加了额外的起始、停止和检验位。图2-38是一张大幅简化过的框图，展示了一个采用两个通用异步收发器（UART）设备构成的异步通信通道。曾经属于PC标配功能的RS-232接口就是一种异步串行接口，在早期版本的PC上使用Intel 8250 UART器件实现。这种器件依然能买到，但是通常是用更现代的逻辑兼容UART，如16550，它作为主板上芯片组的一部分被实现在一块自定义的逻辑门阵列器件中（现在你该知道PC中串口设置对话框中的“16550”是从哪里来的了吧，如果你以前不知道的话——它依然还在，只是已不再是一个分立零件）。这里有个有趣的历史趣闻，事实上在最初的RS-232规范中为时序预留了信号——它同样可被实现为同步接口。规范中依然有这样的定义，但如今在PC上已几乎不再使用。

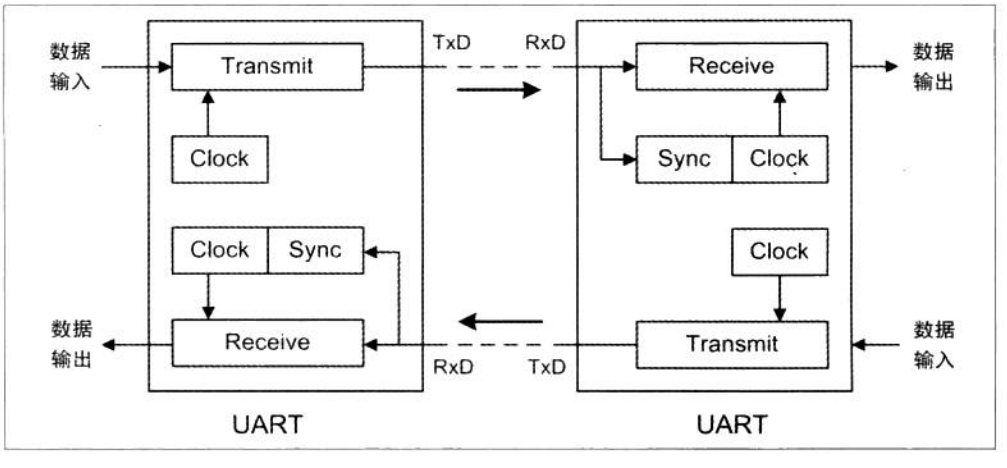


图2-38：异步串行接口

一款现代UART芯片包含了所有实现一个异步串行接口通道一端的必要电路。在图2-38中，标注为“Sync”的方框部分负责检测流入的数据流，以及根据数据速率调整接收端的时钟信号。

如果接口同时具备发送和接收数据的信号线，它就具有全双工操作的能力。术语“全双工”指的是接口能够同时发送和接收数据。只有一条数据通路的接口通常是半双工，它指的是连接到接口的设备在任何时刻只能作为发送器或接收器，接收和发送不能同时进行。

SPI 和 RS-232 只是串行接口中的两个例子。其他的还有 I^2C、USB、RS-422、RS-485、MIL-STD-1553、火线（FireWire）和以太网。它们之间的主要差别在于接口在电气工作方式上的不同，以及数据在通道间传输的速度。在第 7 章中，我们将更深入地考查串行接口，并且会对全双工和半双工操作之间的一些差别进行探究。

54

并行 I/O

并行 I/O 是指用一组离散 I/O 线在单一读或写操作情况下传输一组数据位的数据通信通道。个人计算机的打印机端口就是一个常见的并口实例，实际上，打印机端口除了引脚是预先专为实现打印机接口而安排外并无什么特别之处。图 2-39 显示了 PC 打印机端口的引脚分布和数据传输时序图。

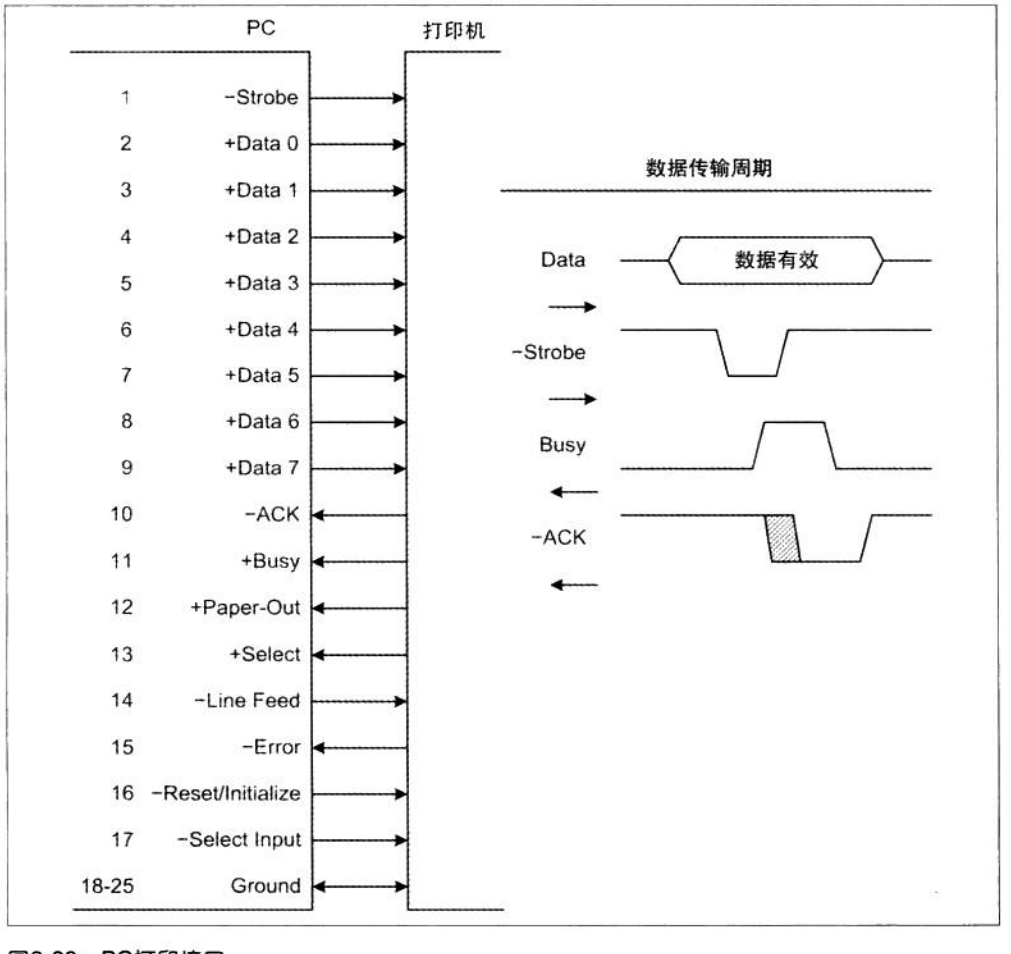


图2-39：PC打印接口

55 在图 2-39 中，时序图（我们在后面会碰到更多）显示了 Strobe、Busy 及 ACK 信号之间的一般关系。+ 号和 – 号分别指明了信号的高有效和低有效真值状态。另外请注意 ACK 波形中的阴影部分，它指明了 ACK 信号的开始可能出现变化，这是没问题的，因为发送端在 ACK 信号回到高状态（逻辑假）之前不会启动另一个传输过程。

并行 I/O 通道还可用通用离散 I/O 硬件实现，例如个人计算机上的 PCI 总线插卡。如图 2-40 所示。

56

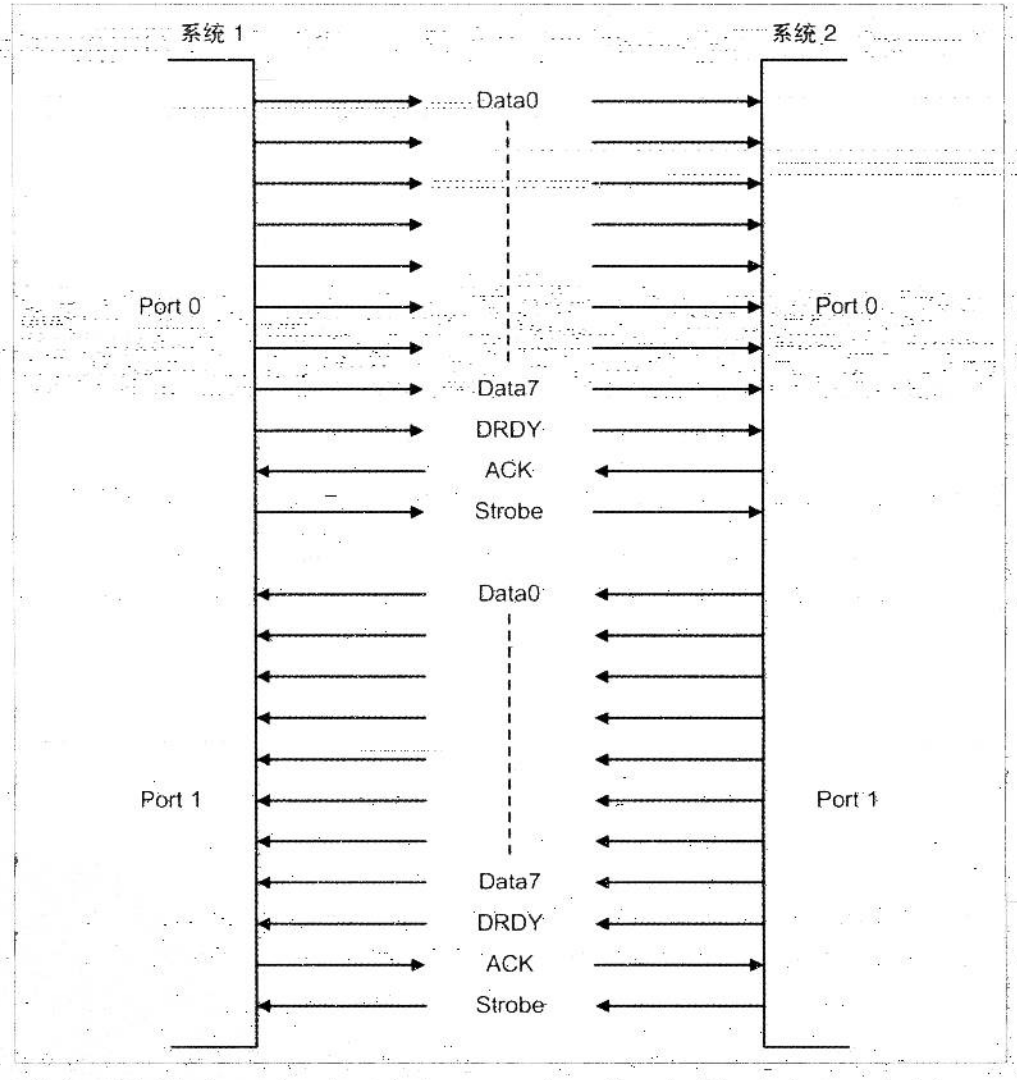


图2-40：双向并行接口

请留意这两种电路中都存在“握手”信号线。这些信号线被用来协调两个设备之间的数据交换，并且通常被实现为“低有效”逻辑（意即当信号线为低而不是高时表示逻辑真——这能够帮助降低由噪音或信号上的其他干扰所引起的错误信号）。在图 2-40 中，当一个设备有数据要发送时，它会将 DRDY（数据就绪）线拉低。接收端作为应答会将其 ACK（应答）线拉低并一直保持，直到数据传输完成。发送端将 Strobe 线拉低即开始传输过程，接收端会用该信号以某种方式把数据锁存到缓冲中。在接收端将 ACK 线释放后，发送端又可以重新开始一个全新的过程来发送另一组数据位。

对于每个传输周期而言，并行数据接口在一次转移多个数据位方面有着得天独厚的优势，而且一个完全以硬件方式实现的并行接口能以极高的速度传输数据。当所有这些功能以软件控制的方式来实现时，其运行速度将不如硬件实现的方式快，但依然要比 RS-232 接口要快得多。

图 2-41 是一个 90 年代后期实现的系统的框图，它使用了一台单板计算机（一块单一 PCB 上的工业 PC）和一台专用控制 PC 相通信，这台专用控制 PC 由于时序和资源限制方面的原因不能使用网络接口，但它支持专用的双向并行接口。

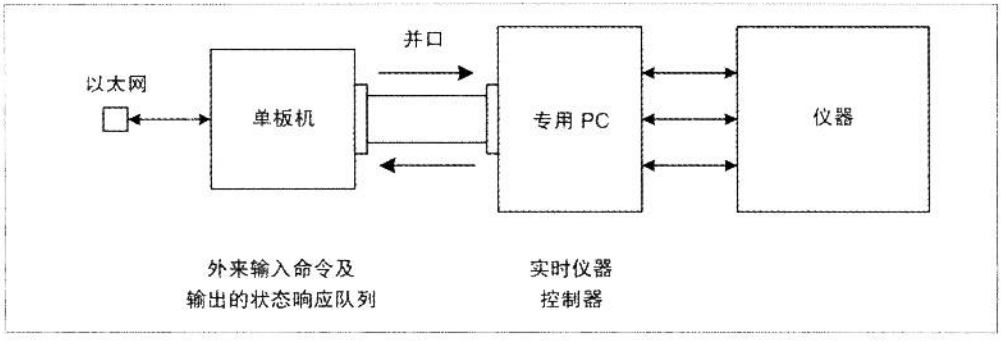


图2-41：PC到PC的并行接口

小结

在本章中，我们讲解了电荷、电流、电压和电阻的基本概念，还从一些实例中学习了怎样应用这些概念。我们也简单地学习了接口电子学，实际上我们几乎只学习了一些皮毛。尽管如此，你现在已经拥有了足够多的背景知识，在学习后面的数据采集和控制设备方面的内容时你会觉得更加轻松。

57 推荐阅读

电子学已经成为现代文明的基石之一，各类电子设备已经触及全球的每个角落。因此电子学领域的广阔和持续变化是毋庸置疑的。下面是一些笔者推荐的书籍，针对那些想要达到比本章内容更深入的读者。

The Art of Electronics, 2nd ed. Paul Horowitz and Winfield Hill, Cambridge University Press, 1989.

本书涵盖了从基本的电子器件到设计和制作高速低噪音的实验室级别的设备的一切知识。该书的写作风格轻松易读，使用的数学恰到好处而不纠缠于细节。书中包含了大量有趣的电子电路实例，也精选了一些需要避免的“烂”电路。

Electronics, 2nd ed. Allan Hambley, Prentice Hall, 1999.

本书适合作为大学电子学理论入门课程课本，本书的表述正规而且严格，但作者的精心布置让读者能够轻松进入主题，而不是丢下一堆公式让读者自己去理清头绪。本书对于理解电子元器件背后的理论基础及其应用是一份极具价值的资料。

Data Conversion Handbook. Analog Devices Inc., Newnes, 2004.

本书由 Analog Devices 公司的工程人员编写，书中阐述了工程人员对数据转换相关主题的权威处理方法，这些人设计出了目前一些最先进的数据转换设备。此书对诸如数据转换的历史、数据采样系统，以及数据转换器接口做了专题介绍。

第3章 Python编程语言

我认为就算完全禁止研发计算机语言，它们依然会各自拥有自个儿的开发语言，并立马悄悄地躲回各自角落中自娱自乐去。

——Dr. Steven D. Majewski[译注1]

自动化设备一个重要的需求是能够和计算机或是其他控制设备连接运行。说到这儿，术语“编程”会立即在一些读者头脑中浮现，实际上做到这一点有很多方法，有些甚至于不涉及编程语言（至少不是传统意义上的）。不过，本书中，我们选择了 Python 和少量的 C 来创造软件给自动设备用。

本章的目的是给出 Python 的基础介绍。在下一章将引入 C 语言，来为 Python 程序使用厂方提供的硬件驱动提供扩展，或是创造处理计算密集型任务的模块。本章不打算成为 Python 的深入教程或是手册；已经有太多优秀的图书可以承担这一角色了（具体参考本意的推荐阅读部分）。此外还可以在 Python 官网（*http://www.python.org*）找到从入门教材到高级话题的各种文档。

使用 Python 作为本书的主力开发语言，主要基于以下原因：足够简单易学；没有编译过程，可以直接加载执行（或是直接输入，如果够胆的话）；足够强大。同时，Python 也是种多范式语言，支持多种编程模型：面向过程的，面向对象的和函数式的。首先我们将进行通常的编程，然后将尝试追加图形界面（GUI）并用 C 来写扩展；进一步地，我们将遇到需要抛开通常的程序设计，完全使用我们自行创建的嵌入式对象。

而且，正如我们将见证的，Python 是天生的面向对象的，甚至于变量本质上也是对象；

译注1：早在 1992 年就为 Python 摇旗呐喊过的牛人：WHY PYTHON?（http://www.python.org/search/hypermail/python-1992/0274.html）作者，PyObjC 项目（http://sourceforge.net/projects/pyobjc/）主持人。

即使我们没有被迫进行 OOP 编程，实际上也每时每刻都在使用对象。如果还不清楚什么是“面向对象”，请阅读下面的文章。

面向过程与面向对象的编程

面向过程的编程被认为是一种命令式编程，核心概念是一组顺序排列的指令形成程序（类似食谱）。扩展程序包含在模块中，模块则由一组函式组成；每个函式执行特定活动（算法），并可以包含其私有数据；函式可以在其所在的模块中使用“全局”数据；函式可以调用其他函式；函式可以返回数据；模块可以从其他模块引用函式或是数据；这种设计可以令程序用模块的层次结构组织起来（即，结构化设计）。标准 C 程序就是一个通常的面向过程的例子。注意，这里我们使用“函式”作为“功能”的同义词，不过，在其他一些语言中，他们被区别对待了。在 Python 和 C 中，就只有函式。

面向对象的编程，用类（及其数据）来描述对象的特性概念以及可以在其上执行的操作（对象的方法）来扩展了同类事务的功能；一个对象，可以说是一类说明和模板，而这种描述本身是不可执行的。只有基于对象创建的实例才可执行；可以想象为，一个类就是个饼干压模，而用这压模作出的饼干有的可能含有坚果，有的又有巧克力，但是他们都称作“饼干”；必要的话，我们可以从指定的类创建很多可执行对象，每个对象除了从其父类继承的数据和方法之外，又都有各自特殊的；对象的数据，一般称为“属性”，而操作数据的类函式称为“方法”；对象，经常用来声明组件间的“有一个”以及“是一个”的关系。例如：一个摇椅“是一个”被称作椅子的类的实行；它“有一个”座位和两个摇杆（属性）；人们可以坐在它上面摇摆（方法）。

安装 Python

首先需要安装 Python。本书使用 2.6（不是 3.x）。对于 Windows 环境，ActiveState 发行版（http://www.activestate.com/activepython/）比较友好，当然从 python.org 获取的官方
61 版本也很好。这两种版本都包含有针对 Windows 定制的帮助和手册。如果运行在 Linux 上，可以直接使用软件包管理器（*synaptic*，*apt-get*，*rpm*，等等）安装即可。

如果想从源代码编译安装，请参考 *http://docs.python.org/using/unix.html#getting-and-installing-the-latest-version-of-python*。

Python 编程

现在至少已安装好了 Python，我们可以通过一个快速导览来体验这语言的主要特性。

Python 是种解释型语言。精确地说，是种编译为字节码的解释型语言。这意味着，Python 在执行脚本前会将脚本编译为一种二进制的可执行程序。其实这就是“解释型”的含义。Python 在其虚拟机中将形式文本解释为字节码来运行，虚拟机内部的指令集是经过优化的，虽然无法和通常的编译成机器码的编译型语言执行效率相比，但对于大多数应用程序来说，字节码的运行速度足够了。尤其是考虑到时下 PC 上的处理器（CPU 或叫中央处理器）都是在 1~3GHz。而以前 30MHz 已经算好的时候，程序本身的执行速度才是大事儿。

不论是否 Python 新手，笔者都强烈推荐阅读 Mark Lutz 写的 *Python Pocket Reference* (*http://oreilly.com/catalog/9780596158095/*)。这本书提供了一个简洁、组织良好、易读且体积小巧的 Python 语言核心功能 / 模块的速查手册，可以随身携带，随用随查。其他相关好书在推荐阅读部分分享。

Python 的命令行

如何以交互模式打开 Python 解释器，取决于所用操作系统。对 Windows 而言，通常是首先打开一个命令提示符窗口（有时被误称作“DOS 窗口”，但是 Windows 早已没有真正的 DOS 了）。在提示符（可能看起来有点不同）中输入以下命令：

```
C:\> python
```

62

应该会看到以下显示（前提是安装过 ActiveState 发行版，不过标准的 Python 发行版的提示也差不多完全一样）：

```
ActivePython 2.6.4.8 (ActiveState Software Inc.) based on
Python 2.6.4 (r264:75706, Nov 3 2009, 13:23:17) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 Linux（或是 BSD/Solaris）系统中也是类似的过程。打开一个 shell 窗口（无所谓哪种 shell 环境，*csh*、*ksh*、*bash* 或是其他），只要安装有 Python，键入 python 就可以看到交互环境的启动信息。

>>> 是 Python 的命令提示符，表示在等待输入点什么来运行。在 Windows 机器中使用 Ctrl-Z 来退出，在 Linux 系统中使用 Ctrl-D 来退出，键入 quit 不起作用。

Python 命令行是个尝试和体验的好地方。我们可以通过内建的帮助机制来探查所有函式。仅输入 help()，不带任何参数，将显示

```
>>> help()

Welcome to Python 2.6! This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics". Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

正如内置帮助所述，官方网站上的教程是可切实认识和使用 Python 的起点。本章从另一个方向来展示 Python。首先向读者介绍数据类型，接着是操作符和语句。我觉得面向对象的基础语言性质很重要，因为用 Python 来构建应用，总会遇到相同的情景，能使用内嵌的数据对象很便捷。

多年来，据我观察，如果教程试图忽略或淡化 Python 面向对象的根本性质，常常是一带而过，比如说："哦，对了……"或是"也类似这样，但这里不必担心……"我们将面对而不是避开这些话题。对内部机制的足够理解，有助于快速理解系统如何可以正常工作，以及为什么又不工作了。如果你是名 Python 新手，那么请通读本章和在线教程。

63

命令行参数和环境

Python 的联机帮助（手册页）内容非常丰富，但不幸的是，如果你只有 Windows 环境则很难体会到其中的方便。在 Linux 系统中，只需在 shell 提示符中键入 man python（实际上，如果安装正确，Python 就应该能在所有类 UNIX 系统中工作）。

在 Windows 下，可以通过在命令行中键入如下命令来获得简略的帮助。

```
C:\> python -h
```

倒回去看，显示大致如下：

```
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
```

```
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-Q arg : division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-t     : issue warnings about inconsistent tab usage (-tt: issue errors)
-u     : unbuffered binary stdout and stderr; also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-3     : warn about Python 3.x incompatibilities that 2to3 cannot trivially fix
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path. The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
```

64

这些命令行参数中的大多数通常可能都没有太多使用的必要，但偶尔也会派上用场（特别是其中的 -i，-tt 和 -v 开关）。环境变量，特别是 PYTHONHOME，很重要，应根据使用的 Python 版本在最初安装时所做的选择来设定。

Python 中的对象

一般来说，在 Python 中万物皆对象，包括数据的变量。一个赋值相当于创建一个新的对象，函式定义也一样。如果不熟悉面向对象的概念，不要太担心（可参考“面向过程与面向对象的编程”一文）。希望随着我们的叙述，它们会变得逐渐清晰。现在，只是展示可以在 Python 中期待找到什么类型的对象，后面会关注如何使用。

表 3-1 列出了 Python 中各种常见的对象类型。类型的类名是通过内置的 type() 方法所返

回的内容，如果涉及类型不匹配将有错误抛出。

表3-1：对象类型

对象类型	类型的类名	描述
Character	chr	单字节字符，在字串中使用
Integer	int	32 位整数
Float	float	双精度（64 位）浮点数
Long integer	long	任意大的整数
Complex	complex	复数
Character string	str	有序（数组）字符集
Dictionary	dict	键 / 值对字典
Tuple	tuple	不可变列表
Function	function	Python 函式对象
Object instance	instance	类实例对象
Object method	instancemethod	对象方法
Class object	classobj	类对象
File	file	磁盘文件对象

接下来将介绍以上这些对象：从数字数据开始，直至像列表、元组、字典等对象。

Python 中的数据类型

如果你已做过类似 Pascal 或 C 语言编程，那么你多半已熟悉变量的概念。它本质上是一个存储在特定内容位置的二进制值。但在 Python 中则完全不同，这也是比较有趣的地方。Python 提供了一般的数字数据类型，如整数、浮点数，等等。它也支持 complex 类型，这一类型封装了复数的实部和虚部。关键在于 Python 是如何实现变量的。

数字数据对象

在 Python 中，一个变量被赋值，实际上是创建了一个对象，值分配给这一对象（它成为对象的属性），然后“绑定”到一个名字。对象通常有一个特殊的构造方法，用来处理在内存中创建（实例化）一个新的对象并初始化的细节。对应地，一个对象可能有一个析构方法，当程序运行结束时将对象从内存中删除。在 Python 中，一个对象的清除，通常由垃圾收集来自动处理。

下面是个 Python 创建新的数据对象的例子。

```
>>> some_var = 5
```

首先生成属性值为 5 的 int 类对象，然后为其绑定名字 some_var（很快我们将看到名字

绑定是如何工作的)。也可以键入以下命令来得到相同的结果。

```
>>> some_var = int(5)
```

本例中，我们通过调用 int 类构造方法，并传入新对象建立时要分配的值，来明确地告诉 Python 所要的对象类型（整数）。重要的是要注意，这不是一个在 C 或 C++ 中的“转换”，而是 int 对象的实例化，这个对象封装了一个整型值 5。

这种形式一开始可能看起来有点怪异，不过很快就会变得习惯。另外，大多数时候你可以放心地忽略一个事实，即变量实际上都是对象，而把它们视作 C 或 C++ 中的变量。

```
>>> var_one = 5
>>> var_two = 10
>>> var_one + var_two
15
```

你可以随时查询一个对象的类型。

```
>>> type(some_var)
<type 'int'>
```

虽然刚刚说过 int() 不是类型转换，但是可以行使类似的职能，将一个数据对象转换成其他类型的。

66

```
>>> float_var = 5.5
>>> int_var = int(float_var)
>>> print int_var
5
```

请注意，float_var 的小数部分在转换后消失了。

八进制和十六进制整数表示法也是支持的，和 C 中的效果一样。

八进制整数
: 使用前导 0，如 0157。

十六进制整数
: 使用前导 0x，如 0x3FE。

八进制和十六进制值没有自己的类型类。这是因为当以这两种格式写入值分配给一个 Python 变量时，它会转换为对应的整数值。

```
>>> foo_hex = 0x2A7
>>> print foo_hex
679
```

等同于

```
>>> foo_hex = int("2A7",16)
>>> print foo_hex
679
```

那么究竟什么是“数据对象”？在 Python 中，变量名是由命名空间掌管的，而命名空间分很多层，从函式所在的局部命名空间到 Python 解释器执行环境所在的全局命名空间。当前，我们不用深究，仅使用本地命名空间的概念。

除了构成名称的字串，变量名没有任何价值。它们更像是句柄或标签，可附着于有价值的东西——即对象。图 3-1 显示了这一过程。

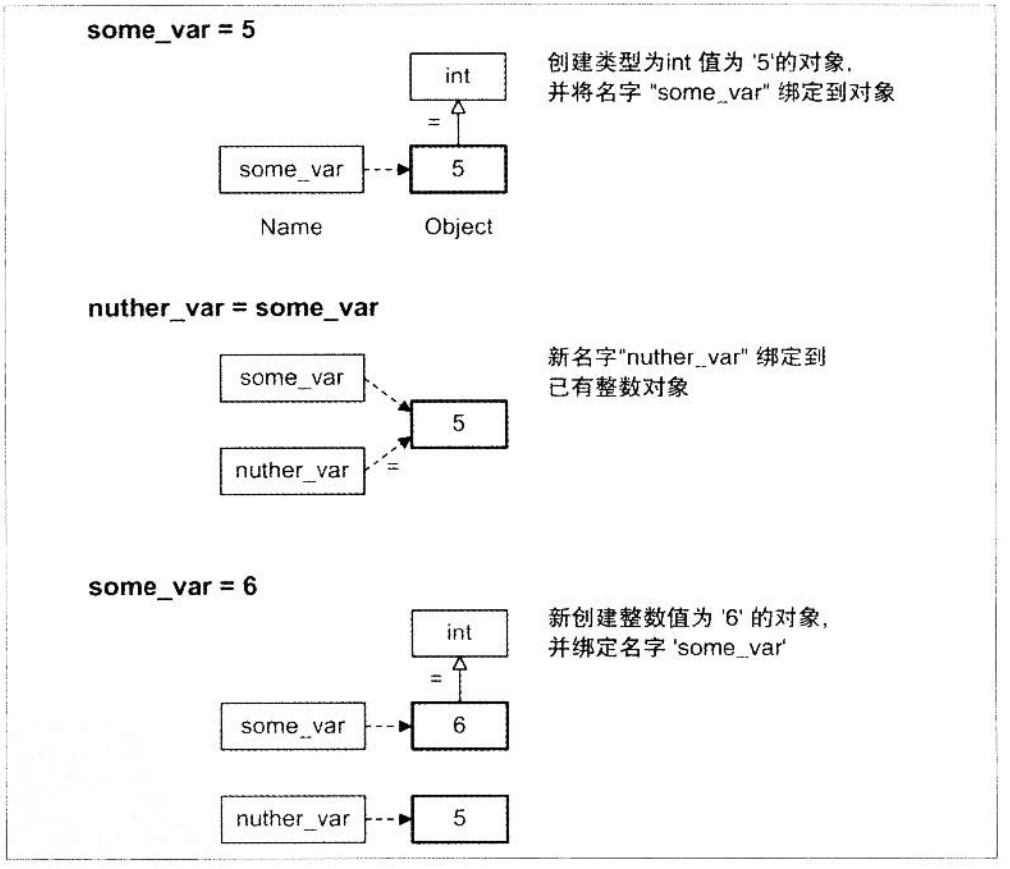


图3-1：数字数据类型

通常对象有内部函式，封装了对数据进行的操作。Python 的数据对象也不例外。如果创

建了一个整数对象，就可以像下面这样使用 help() 来查询 Python 可以进行哪些操作。

```
>>> int_var = 5
>>> help(5)
Help on int object:

class int(object)
 | int(x[, base]) -> integer
 |
 | Convert a string or number to an integer, if possible. A floating point
 | argument will be truncated towards zero (this does not include a string
 | representation of a floating point number!) When converting a string, use
 | the optional base. It is an error to supply a base when converting a
 | non-string. If base is zero, the proper base is guessed based on the
 | string content. If the argument is outside the integer range a
 | long object will be returned instead.
 |
 | Methods defined here:
 |
 | __abs__(...)
 |     x.__abs__() <==> abs(x)
 |
 | __add__(...)
 |     x.__add__(y) <==> x+y
 |
 | __and__(...)
 |     x.__and__(y) <==> x&y
 |
 | __cmp__(...)
 |     x.__cmp__(y) <==> cmp(x,y)
 |
 | __coerce__(...)
 |     x.__coerce__(y) <==> coerce(x, y)
 |
 | __div__(...)
 |    x.__div__(y) <==> x/y
 |
 | __divmod__(...)
 |    x.__divmod__(y) <==> divmod(x, y)
 |
 | __float__(...)
 |    x.__float__() <==> float(x)
 |
 | __floordiv__(...)
 |    x.__floordiv__(y) <==> x//y
 |
 | __format__(...)
 |
 | __getattribute__(...)
```

```
| x.__getattribute__('name') <==> x.name
|
| __getnewargs__(...)
|
-- More --
```

还有更多的内部方法，如果有兴趣，你可以仔细阅读它们（只需按空格键来读下一屏，按回车键下滚一行，或按 q 键返回到提示符），但主要的一点是，在 Python 中，数据对象“知道”如何用内置的特定类方法处理内部数据。换句话说，Python 解释器负责处理在内部将下面这样的语句

```
5 + 5
```

68 转换成如下等价的字节码的细节：

```
int(5).__add__(int(5))
```

然后执行之。

要适应 Python 中变量皆对象这一事实的确得花点儿时间。但这却正是 Python 强大特性之一，而且因为可以选择性地忽略这一特性，所以，在一切皆对象的 Python 中，依然可以创建出面向过程语言风格的程序。

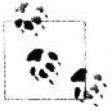
序列对象

Python 提供了三种有序集合数据对象类型：列表（数组）、字串和元组（列表样对象）。这些也被称为“序列对象”。“序列”的每一部分事实上都可以包含零个或由其他数据
69 组成的一个有序的对象序列。除了字串，其他序列对象都允许其成员要素是任何有效的 Python 对象。序列对象都有对应方法来操纵其数据，有些方法能作用于所有序列对象，有些针对特定的类型。表 3-2 列出了这三种序列类型及其部分属性。

表3-2：序列对象

对象类型	可变？	定界符
列表	是	[]
字串	否	'' 或 ""
元组	否	()

Python 的序列对象分可变和不变两类。例如，列表对象是可变的，因为其数据可以被修改。另一方面，字串是不能改变的。一个字串是无法取代、删除或直接插入字符的。所以字串对象是不可变的字符值的集合，被视为由字节数据组成的只读数组对象。



事实上，这种说法仅适用于 8 位的 UTF-8 字符编码，其他字符集（如 Unicode）的字串可能每个字节需要不止一个字符。不过在本书中，我们将只使用 UTF-8 字符编码（更多关于 ASCII 和 UTF-8 字符编码标准的内容请参阅第 12 章）。

要改变字串，就必须创建包含了变更的新字串。原始字串对象将保持不变，即使是给新的字串对象重复使用相同的变量名（这将“取消绑定”原始字串对象，取消绑定的对象往往会通过垃圾收集过程自动蒸发，但这是一个底层细节，我们并不需要操心）。

列表。列表是 Python 中最接近数组的对象，但是其中有一些技巧是在 C 或是 Pascal 的数组中无法实现的。列表是一个有序序列，其中的任何元素都可被替换成不同的对象。新元素添加到一个列表使用其 append 方法（还有一个 pop 方法，这意味着列表可以当成堆栈或是队列来用）实现，列表的内容可以就地排序。列表中的每个元素实际上是一个对象引用，就像一个数字数据变量名称是对一个数字数据对象的引用。事实上，列表可以包含对任何有效的 Python 对象的引用。考虑以下代码：

```
>>> import random
>>> alist = []
>>> alist.append(4)
>>> alist.append(55.89)
>>> alist.append('a short string')
>>> alist.append(random.random)
```

现在 alist 包含四个元素：一个整数，一个浮点数，一个字符串和一个 Python 的随机数模块的方法（我们将在后面详细讨论 import 语句）。现在可以检查 alist 的每个成员元素来验证这一点。 70

```
>>> alist[0]
4
>>> alist[1]
55.890000000000001
>>> alist[2]
'a short string'
>>> alist[3]
<built-in method random of Random object at 0x00A29D28>
```

如们想要一个随机数，仅需要调用 alist[3]()，即 random()。

```
>>> alist[3]()
0.87358651337544713
```

我们可以对 alist 中的元素进行简单的赋值来完成变更。

```
>>> alist[2]
'a short string'
```

```
>>> alist[2] = 'a better string'
>>> alist[2]
'a better string'
```

图 3-2 展示了在 alist 中发生了什么。

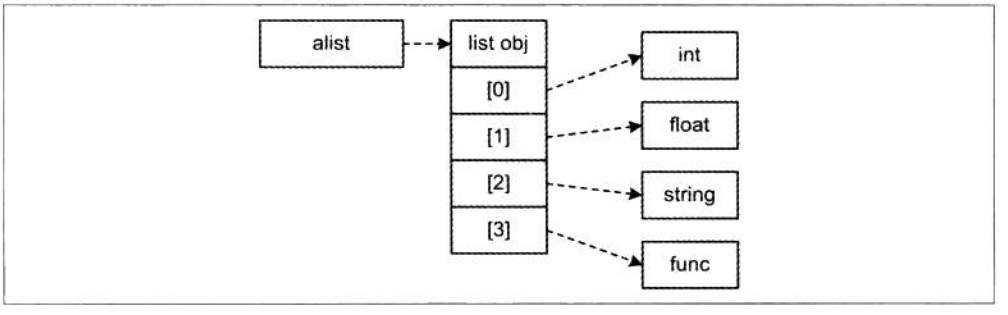


图3-2：列表对象的内部组织

我们可以用一个列表对象来体验 Python 的面向对象的基本性质，尝试在 Python 提示符中输入以下命令，并观察结果。

```
>>> list_name = []
>>> list_name.append(0)
>>> list_name.append(1)
>>> list_name
[0, 1]
>>> var_one = list_name
>>> var_two = list_name
>>> var_one
[0, 1]
>>> var_two
[0, 1]
>>> list_name[0] = 9
>>> var_one
[9, 1]
>>> var_two
[9, 1]
```

71

由于名称 var_one 和 var_two 都指向最初列表对象，即绑定为名称 list_name 的列表，因此在 list_name 改变时，其内容变化其他变量名都是能“看到”的。

正如其他 Python 中的对象，列表也有一系列方法。除前述的索引方法外还有很多。列表可以像下面一样依次头尾连接在一起。

```
>>> alist1 = [1,2,3,4,5]
>>> alist2 = [6,7,8,9,10]
```

```
>>> alist1 + alist2
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

查找列表中指定偏移量的项，可以用 index() 方法。

```
>>> alist2.index(8)
2
```

也可以反转列表。

```
>>> alist1.reverse()
>>> alist1
[5, 4, 3, 2, 1]
```

同样，也能对列表进行排序。

```
>>> slist = [8,22,0,5,16,99,14,-6,42,66]
>>> slist.sort()
>>> slist
[-6, 0, 5, 8, 14, 16, 22, 42, 66, 99]
```

以上两个例子的美妙之处在于是“就地”修订的。新的对象不是创建出来的，而是列表本身直接反转或排序了。即，列表是可变的。

字串。字串是有序的字节值的字符序列。字串是不可改变的（不像在 C 或 C++ 中），就是说无法像对待数组一样使用索引来修订内容。要修改一个字串，必须创建一个新的字串对象。不过，类似列表，字串的内容能够通过其索引引用。

下面是几个字串的例子。

```
>>> astr1 = 'This is a short string.'
>>> astr2 = "This is another short string."
>>> astr3 = "This string has 'embedded' single-quote chracters."
>>> astr4 = """This is an example
... of a multi-line
... string.
... """
>>>
```


虽然不能使用索引值改变一个字串的内容，但是可以使用索引来读取字串数据，Python 支持提取字串中的特定部分（或称为“切片”）。其结果是一个新的字串对象。如下代码将读取字符串变量 astr1 前四个字符，从第 0 位到第 4 位（不包含第 5 个字符）。

```
>>> print astr1[0:4]
This
```

我们也可以不写起始范围值 0，Python 会自动假定。

```
>>> print astr1[:4]
This
```

这种形式告诉了 Python 提取字串从开始到第 4 位的所有字符。我们也可以提取从第 4 位到行尾的所有字符。

```
>>> print astr1[4:]
 is a short string.
```

或是获取中间的部分。

```
>>> print astr1[10:15]
short
```

图 3-3 展示了 Python 中索引的工作方式。

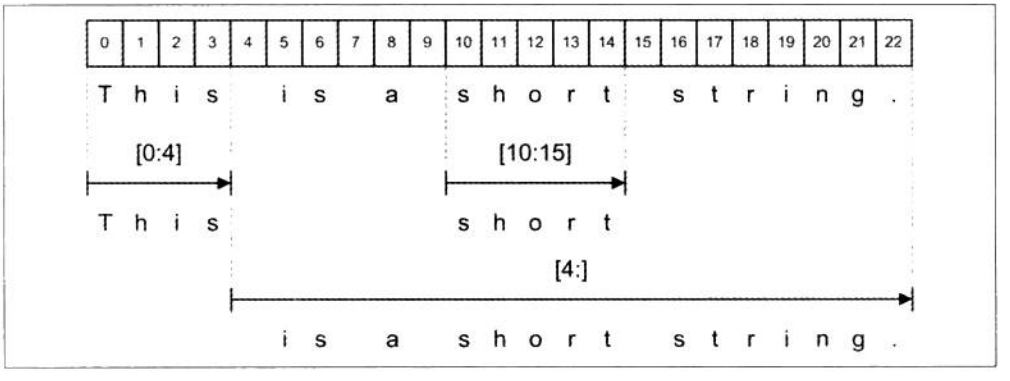


图3-3：字串索引

字串对象还包含了一系列方法，如首字大写、居中、统计字符出现次数等等，它们基本上都返回一个新的字串对象。

类似列表，字串使用 + 操作符来连接。

73

```
>>> str_cat = astr1 + " " + astr2
>>> print str_cat
This is a short string. This is another short string.
```

不出所料，其结果是一个新的字串对象。幸运的是，Python 采用了垃圾收集机制，当对象不再绑定到一个名字时，将悄悄地消失，原先所用内存将返回到共享池以待再用。这是一件好事，否则内存会很快被废弃数据对象填满。

```
>>> the_string = "This is the string."
>>> the_string = the_string[0:4]
>>> the_string
```

```
'This'
```

在上例中，最初的名称 the_string 绑定的对象，内容是“This is the string.“。当这个字串对象的初始部分被拉出时，一个新的对象被创建并分配到这个名字。原来的对象不再被绑定，于是得到回收。然而，如果一个对象有两个或多个名称共享，那么只要还有名称在绑定，就不会被回收。这样在程序生命周期内对象总是可以使用。

有一些字串方法可用以左 / 右对齐字串，替换其中的一个字，或转换字串中字符的大小写。下面是一些例子。

upper() 方法，转换所有字符为大写。

```
>>> print astr1.upper()
THIS IS A SHORT STRING.
```

find() 方法返回第一个匹配搜索模式的字串索引。

```
>>> print astr1.find('string')
16
```

replace() 方法根据搜索模式生成完成替换后的新字串。

```
>>> print astr1.replace('string', 'line')
This is a short line.
```

rjust() 方法（及与其对应的 ljust()）将一个字串重整为指定宽度的新字串。

```
>>> print astr1.rjust(30)
       This is a short string.
```

默认的填充字符是空格，也可以在第二参数中指定一个特殊字符作为填充字符。

```
>>> print astr1.rjust(30,'.')
.......This is a short string.
```

在 Python 提示符环境中，可以通过输入 help(str) 获知所有字串支持的操作。

元组 。元组是一种有趣的数据对象。类似列表，它是一个有序集，可以包含零个或多个项目，不过它是不可改变的。元组一旦创建就不能直接修改。通常提到元组时都会说明其所包含的项目数。例如，如你所料，二元组包含两个数据对象。一个对任何规模的元组都适用的速记法是“n 元组”。Python 中“0 元组”是允许的，虽然不怎么有趣和有用，但也许可以用作占位符。 74

Python 中列表用方括号做定界符，元组则使用圆括号。

```
>>> tuple2 = (1,2)
```

```
>>> tuple2
(1, 2)
```

元组的内容可以通过索引读取，这点与列表和字串相同。

```
>>> tuple4 = (9, 22.5, 0x16, 0)
>>> tuple4
(9, 22.5, 22, 0)
>>> tuple4[2]
22
>>> tuple4[0]
9
```

类似列表和字串，元组可以合并（生成新元组）。

```
>>> tuple2
(1, 2)
>>> tuple4
(9, 22.5, 22, 0)
>>> tuple6 = tuple2 + tuple4
>>> tuple6
(1, 2, 9, 22.5, 22, 0)
```

上例中，可以看到生成了一个新元组。

元组不能进行排序，但可以计数。想了解元组包含多少特定值对象，可以使用 count() 方法。

```
>>> tpl = (0, 0, 2, 2, 6, 0, 3, 2, 1, 0)
>>> tpl.count(0)
4
>>> tpl.count(2)
3
>>> tpl.count(6)
1
```

因为元组的每个内容项实际上是对象的引用，所以元组可以包含任意组合的有效 Python 对象，这点和列表相同。

75 字典

Python 的字典是独立的数据对象。字典中的数据集是种无序的键 / 值对形式，而不是一个数据元素的有序集。也就是说，每个数据元素有一个相关的键唯一标识它。这是 Python 中仅有的映射式数据对象。

同其他 Python 数据对象一样，字典可以作为参数传递给一个函数或作为返回值。也可以成为元组或列表数据元素，其值可以是任何有效的 Python 对象类型。不过，字典的键值必须是整数、字串和元组。换句话说，键必须是不可变的对象。

我们可以通过初始化键值对来完成字典的创建。

```
>>> dobj = {0:"zero", 1:"one", "food":"eat", "spam":42}
>>> dobj
{0: 'zero', 1: 'one', 'food': 'eat', 'spam': 42}
```

要获取特定的键对应的值，可以用类似索引的方式，不过其实不是索引。

```
>>> dobj[0]
'zero'
>>> dobj[1]
'one'
```

如果尝试访问字典中没有的键，Python 会抱怨。

```
>>> dobj[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

但只要给的是有效的键，就会得到一个有效的值。

```
>>> dobj["spam"]
42
```

字典有一整套方法来操纵纳入其中的数据。表 3-3 包含了可用的方法，接下来我们会详细考察其中的几个方法。

表3-3：字典方法

方法	描述
clear()	从字典中清除所有条目
copy()	字典的“浅”复制
get()	返回与键匹配的值，没有匹配时返回默认值
has_key()	查询字典中是否存在指定的键，若有返回 True，反之返回 False
items()	以二元组的形式返回字典中的所有键值对
iteritems()	以迭代形式返回字典中所有键值对
iterkeys()	以迭代形式返回字典中所有键
itervalues()	以迭代形式返回字典中所有值
keys()	以列表形式返回字典中所有键
pop()	通过键弹出一个特定条目，并从词典中删除之
popitem()	通过键值对弹出一个条目，并从词典中删除之
setdefault()	配置当 get() 匹配失败时返回的默认值
update()	从另一字典更新键匹配的值
values()	以列表形式返回字典中所有值

76

注意，字典没有列表中的 append() 方法。要添加一个新条目到字典中，只需简单分配一个值到一个新的键上。

```
>>> dobj[99] = "agent"
>>> dobj
{0: 'zero', 1: 'one', 99: 'agent', 'food': 'eat', 'spam': 42}
```

请注意，新键和对应的数据将在字典中插入到一个任意位置。字典不是序列对象，数据访问必须通过键查询，所以它确实不必关注键值对具体存放的位置。

这种技术还可以用于修改现有的键的值。

```
>>> dobj[1] = "the big one"
>>> dobj
{0: 'zero', 1: 'the big one', 99: 'agent', 'food': 'eat', 'spam': 42}
```

更安全的值获取方式是使用 get() 方法。

```
>>> dobj.get(99)
'agent'
```

如果试图获取一个不存在的键值，get() 方法默认会返回特殊值 None。在 Python 的命令行环境中将没有显示。

```
>>> dobj.get(256)
```

也可以选择一个默认值，来替代原先的 None。

```
>>> dobj.get(256,"Nope")
'Nope'
```

字典常用以保存全局数据（如参数），并能在外部没有提供可用值时，返回一个默认值以便程序使用预定义的参数值。

有时我们可能需要一个字典中所有内容的列表。items() 方法就是用以返回字典的键值对二元组对象的。

```
>>> dobj.items()
[(0, 'zero'), (1, 'the big one'), (99, 'agent'), ('food', 'eat'), ('spam',
42)]
```

77 如果要所有键的列表，则用 keys()。

```
>>> dobj.keys()
[0, 1, 99, 'food', 'spam']
```

最后，如果只对所有值有兴趣，可以用 values()。

```
>>> dobj.values()
['zero', 'the big one', 'agent', 'eat', 42]
```

以上这些对于使用字典足够了。后面还会看到其他一些有趣的字典用法和其他的 Python 数据类型。在此期间，随时在命令行交互环境中尝试各种新家伙。尝试体验是最好的学习方式之一。

表达式

本书中，我们将使用表达式的数学定义。也就是说，表达式是种良构的由变量和逻辑或数学符号组成的不包含等号（赋值）的序列，可以求得有效的逻辑或数字值。而语句（马上会看到）代表一项任务或是其他一些行动，可以包含表达式。

表达式可完成各种值运算，如加、减、比较等等。表达式可以很简单，如：

```
a + b
```

或是包含其他表达式，如：

```
((a + b) * c) ** z
```

括号用来标识运算的顺序。在上例中，乘法操作符（*）比加法（+）有较高的优先级，指数（**）则比乘法有更高的优先级，如果没有括号表达式会是这样理解：

```
a + b * c**z
```

如果我们用括号将隐含的优先级表示出来，看起来会更清晰：

```
a + (b * (c**z))
```

这当然不是表达式的原意。

表达式还可能包含其他算子。例如，假设有一个函式 epow() 返回 e 的高阶指数或其他表达式的结果。表达式可以包含对此函式的调用来创建新值。

```
n + epow(x - (2 * y))
```

这相当于在标准数学中写作 $n+e^{(x-2y)}$。

操作符

78

我们已经看过了 Python 支持的数据类型以及表达式，现在，来看看可以用操作符对它们做什么事儿。Python 提供了一套算术 / 逻辑和比较操作符，还有位操作、成员测试和身份测试的操作符，并提供了各种赋值操作符。

算术操作符

Python 提供了常用的四种基本算术操作符：加，减，乘和除。还有其他语言中没有的两种运算：指数和浮点除。表 3-4 列出了 Python 中的算术操作符。

表3-4：算术操作符

操作符	描述
+	加
-	减
*	乘
/	除
%	取模
**	指数
//	浮点除

在处理混合数字类型数据，Python 会自动地“提升”为其中最高级别的数字类型来操作。类型优先级如下：

```
complex
float
long
int
```

这意味着，如果一个表达式包含一个浮点值，但没有复数，结果将是一个浮点值。如果一个表达式包含一个长整数，但没有浮点数或是复数，其结果将是一个长整数。如果一个表达式包含一个复数，结果将是复数。所以，对于如下表达式：

```
5.0 * 5
```

结果将是一个浮点数：

```
25.0
```


如前所述，Python 还具有独特的除运算，称为“浮点除”。用于将一个浮点数的商值截断为最接近的一个整数值，但是结果返回为一个浮点数。在 Python 中，// 行为如下所示。

```
>>> 5/2
2
>>> 5//2
2
>>> 5.0/2
2.5
>>> 5.0//2
2.0
```

逻辑操作符

表 3-5 列出的 Python 逻辑操作符可作用于各种对象的真值。

表3-5：逻辑操作符

操作符	描述
and	逻辑与
or	逻辑或
not	逻辑非

Python 支持在逻辑表达式中使用关键字 True 和 False。请注意，以下任何一条都为 False。

- None 对象。
- 0（数字）。
- 空的序列对象（列表，元组，字串）。
- 空的字典。

其他值情况都被视作 True。另外也建议用 1 和 0 来对应 True 和 False。

比较操作符

比较操作符计算两个操作对象，并以相等、不等或大小形式描述其关系（见表 3-6）。

表3-6：比较操作符

操作符	描述
==	相等返回真，否则是假
!=	不等返回真，否则是假
<>	同 !=
>	大于返回真，否则是假
<	小于返回真，否则是假
>=	大于或等于返回真，否则是假
<=	小于或等于返回真，否则是假

使用比较操作符的 Python 表达式总是返回逻辑值真或假。

位操作符

Python 的与、或、异或操作符对操作数进行比特位的运算，不执行算术运算。具体参考表 3-7。

表3-7：位操作符

操作符	描述
&	二进制与
|	二进制或
^	二进制异或
~	二进制补
<<	二进制左移
>>	二进制右移

与操作将返回两个操作数那些同为 true（1）的数位，而或操作将“合并”两个操作数位，如图 3-4 所示。

图3-4：Python的位与和位或操作符

位操作对于设置特定的位（或）和测试某个位是否为 1（与）特别有用。异或操作返回两个操作数位元之间的差异，参见图 3-5 所示真值表。

81

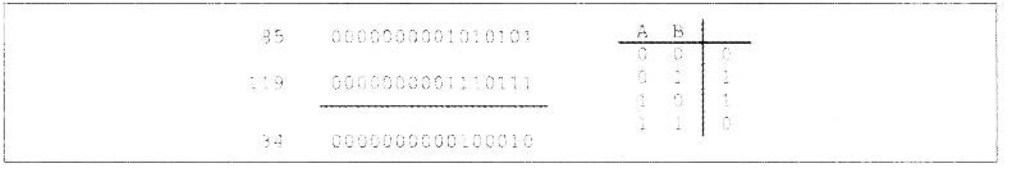


图3-5：Python的位异或操作符

求补运算将每个位值进行反转。也就是说，二进制值 00101100 变为 11010011。

二进制移位操作是将数据对象的内存数据向左或向右移动指定的数位。效果相当于乘 2^n（左移）或除 2^n（右移）（其中 n 是移动的位数）。例如：

```
>>> 2 << 1
4
>>> 2 << 2
8
```

```
>>> 2 << 3
16
>>> 16 >> 2
4
```

赋值操作符

我们已经看到，在 Python 中赋值涉及的不仅仅是一些数据填充到内存位置。赋值相当于实例化新的数据对象。Python 的赋值操作符列于表 3-8。

表3-8：赋值操作符

操作符	描述
=	简单赋值
+=	自增赋值（增量赋值）
-=	自减赋值（增量赋值）
*=	自乘赋值（增量赋值）
/=	自除赋值（增量赋值）
%=	自模赋值（增量赋值）
**=	自指赋值（增量赋值）
//=	浮点除再赋值（增量赋值）

除了简单的赋值操作符，Python 为每个算术操作符提供了一组自增操作。自增操作先执行对应操作，然后将结果赋给操作符左侧的对象。例如：

82

```
>>> a = 1
>>> a += 1
>>> a
2
```

成员操作符

成员操作符用以确定在一个序列或字典对象中是否存在（in）或不存在（not in）某个值或对象（见表 3-9）。请注意，字典对象只能对键进行测试，不能对值进行成员测试。

表3-9：成员操作符

操作符	描述
in	若包含查询对象返回 True，反之为 False
not in	若不含查询对象返回 True，反之为 False

可以像下面这么用 in 操作符：

```
if x in some_list:
    DoSomething(x, some_list)
```

该例中，函数 doSomething() 只有当 x 在 some_list 中时，才会被调用。也可以进行相反的测试，看看是否不在某个对象中。

```
if x not in some_dict:
    some_dict[x] = new_value
```

如果键 x 不在字典中，则会被追加并赋一个值。

身份操作符

Python 的身份操作符（见表 3-10）用于确定一个名字是否和另一个名字指向同一对象（is），或者不指向同一对象（is not）。

表3-10：身份操作符

操作符	描述
is	若为相同对象则为 True，否则为 False
is not	若不是相同对象则为 True，否则为 False

身份操作一般用于确定一个对象是否可用于某个特定的操作。如果操作符两边的变量名指向同一对象，则 is 表达式值为 True。如果操作符两边的变量名指向不同对象，则 is not 表达式值为 True。

83 这里给出一个（不可执行的）例子。

```
def GetFilePath(name):
    global pathParse

    if pathParse is None:
        pathParse = FileUtil.PathParse()

    file_path = pathParse(name)
    if len(file_path) > 1:
        return file_path
    else:
        return None
```

全局名称 pathParse（在模块开始处）被初始化为 None，在这个函式中将指向 FileUtil 模块类 pathParse 的对象。如果此处不进行判定，那么一旦被意外初始化成 None，接下来的程序将没有意义，会失败。

操作符优先级

前面已经看到了部分操作符的优先级特性，下面就来仔细看看。表 3-11 列出了 Python 所有操作符从低到高的优先顺序。

表3-11：操作符优先级

优先级	描述
最低	or
.	and
.	not x
.	in, not in, is, is not, <, <=, >, >=, <>, !=, ==
.	|
.	^
.	&
.	<<, >>
.	+, -
.	*, /, //, %
.	+x, -x, ~x
最高	**

前面提到过，可以使用括号来明确运算的顺序。如果不记得操作符的优先级，或是默认顺序不是你想要的，可以使用括号来获得所需的结果。使用括号来加以明确从来不是一件坏事。

语句

84

典型的程序由语句、注释和空白（空白行，空格，制表符等）组成。语句是由关键字和可选的表达式组成的一个确切的动作。一条语句可以是一个简单的任务。

```
>>> some_var = 5
```

也可以是控制语句的集合，比如 if-else 结构。

```
>>> if some_var < 10:
...     print "Yes"
...     print "Indeed"
... else:
...     print "Sorry"
...     print "Nope"
...
Yes
Indeed
```

Python 也因自己所没有的东西而变得有趣。比如，有其他语言开发经验的人可能注意到了，Python 没有 switch 和 case 语句。Python 中的 if-elif-else 结构通常足以满足同类需求。也没有像 C 语言中的结构体类型。在 Python 中，字典和列表类型可以模拟为一个结构来用，但通常没有这个必要。Python 也没有 do，比如 do-until 或是 do-while。Python 中

有 for 语句，但与 C 中的工作方式不同。

缩进

谈及程序结构时，人们常常提到语句块。块可以定义为一条或多条逻辑上相关联的语句。与 C 和其他语言不同，Python 没有使用特殊字符或保留字来声明语句块。它采用的是缩进！[译注 2] 例如，在 C 中，上面的 if-else 语句会这样写：

```
if (some_var < 10) {
    printf("Yes\n");
    printf("Indeed\n");
}
else {
    printf("Sorry\n");
    printf("Nope\n");
}
```

花括号告诉了 C 编译器如何对语句进行分组，实际上 C 不怎么在意每条语句有多少缩进，对于 C 而言这都是“空白”，编译器会忽略它们。但是在 Python 中，缩进至关重要，因为它会告诉解释器如何理解代码的结构，哪些语句有逻辑关联。具体的缩进量并不重要，
85 只要它是一致的。建议的缩进是每四个空格代表一个层次，不使用 TAB（一般认为 TAB 有点邪恶，因为它们并不总能在不同的编辑器之间优雅地迁移，有的编辑器可能将制表符解释为四个空格，而另一个可能将其转化为八个空格）。

有些人在争论 Python 用缩进表示代码块是否靠谱，对于那些有丰富的 C 或是 C++ 经验的程序员而言，这种形式很怪（虽然在计算机科学领域绝不算新的想法）。用缩进来声明程序层次的优点是可统一所有开发者的风格，并提高可读性。当然也有些人声称在大段代码中使用如 #endif、#endfor 和 #endwhile 等注释更加有利于代码的阅读，不过这里不做讨论。

注释

在 Python 中，注释用一个 # 字符表示（有时称为 # 号），可以出现在行中的任何地方。解释器会忽略 # 后面的所有内容。尽情使用注释来对程序做注解吧，只需注意要有价值。像下面这样的注释

```
a += 1    # increment by one
```

完全无用（可惜这种注释是最常见的），而如下注释

```
if (a + 1) > maxval:      # do not increment past limit
```

译注 2：这一设计恰恰是 Python 同时赢得拥趸以及招致反对的核心特性！喜欢的人爱得发疯，反感的人恨到发疯。笔者认为，这一设计虽然引发了一定编辑环境的配置争议，但是从根本上引入了代码形式和功能必须同时追求优美的开发态度，是一创举，应该得到推广。

就有助于消除理解障碍。

关键字

Python 仅仅包含 31 个保留关键字，如表 3-12 所示。

表3-12：Python的关键字

and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

本章我们将学习其中最为常用的关键字。其他的将在开始开发一些更加复杂的程序时适时引出。

简单语句

86

在 Python 中，一条简单的语句（见表 3-13）是指一行仅包含一个赋值或是关键字的代码。当然，语句也可以包含多条表达式。

表3-13：简单语句

关键字	描述
assert	assert *<expression>*; 如果 *<expression>* 不为 True，一个例外将被抛出
赋值 (=)	创建新数据对象并绑定到名字
增量赋值	参见表 3-8
pass	略过；执行时不作任何事儿
del	移除名称或是名称列表和对象间的绑定
print	发送到标准输出（stdout）
return	可选择的值或是表达式结果返回
yield	只用于生成器函式
raise	抛出例外
break	从 for 和 while 循环中跳出
continue	在 for 和 while 循环中强行忽略当前一层循环，跳到下一轮循环继续
import	将外部模块包含到当前名称空间中
global	指定一个名称列表作为当前模块的全局变量来处理
exec	支持 Python 代码的动态执行

以下部分刻意跳过了 del、exec 和 yield 语句，是因为本书中真的用不到它们。关于 import 语句的介绍则放在本章末尾的“模块导入”一节专门讨论。

assert。assert 语句通常用来判断某些条件是否已满足。如果没有，则引发异常。一般用于单元测试，有时也用于捕获标称条件（虽然还有其他方法可以做到这一点）。

赋值。赋值语句（=）大概是最基础的 Python 语句形式。正如我们已经看到的，赋值基本上相当于实例化某个类型的对象并将其绑定到一个名称。前面几节我们已经进行了广泛的讨论，这里不再深入。

增量赋值。增量赋值语句是非常有用的，在 Python 中也很常见。因为任何类型的赋值都将创建一个新的数据对象，所以，无法在表达式中包含增量赋值。换句话说，这行代码将无法工作：

87
```
if (a += 1) > maxval:
```

但是下面的代码可以。

```
if (a + 1) > maxval:
```

增量赋值中，先执行算术运算，其次才是赋值。对于 Python 的所有赋值操作符的简介，参见表 3-8。

pass。该语句是无操作语句，不执行任何操作。通常在语法上需要一条语句时用作占位符。往往在顶级类方法声明中使用，旨在通过子类方法进行覆盖。也可能出现在“回调”函式或方法中，以保持语法的完整，而并不真正需要做什么事情。

print。该语句会把一个或多个对象的值写到标准输出（stdout），除非标准输出已经重定向或打印输出本身就是重定向的。如果给定的对象不是一个字符串，print 会尝试将数据转换为字符串形式。默认情况下，print 将追加一个换行符（/n）到输出末尾，当然这是可禁用的。

return。该语句用于从函式或方法返回控制权给原调用方。return 语句可以选择将数据传递回调用方，这里的数据可以是任何有效的 Python 对象。如前所述，函式可能返回元组而非单一值，这使其可以同时返回全部状态和数据值（或更多）。虽然可以返回一个列表或字典，但这样庞大而复杂的数据对象对大型程序来说可能产生问题，因为这些数据类型有天生的不透明性。后面一节将进一步说明这些问题。

break。break 语句只可能发生在 for 或 while 循环中。它会终止当前的循环结构，并跳过可能的 else 语句。

continue。continue 语句只可能发生在 for 或 while 循环中。强制返回到循环开始的 for 或 while 语句继续。任何后续语句都将被跳过。如果 continue 语句之外有 try-finally 结构，那么 finally 将在下一轮循环前执行。我们将在稍后讨论 try-except 结构。

global。global 语句用于声明可由模块中的函式或方法修改的模块中的变量名。一般来说，这种全局变量在函式或是方法中出现之前，对于函式或是方法而言是只读的。

复合语句

复合语句是一组逻辑上相关的语句和控制其他语句的执行的语句构成。下面我们将探讨 if、while、for 和 try 语句，但会略过 with 语句，并把 def 和 class 语句留到下一节介绍。表 3-14 列出了 Python 的复合语句。

88

表3-14：复合语句

关键字	描述
if	可选或终结条件测试
while	初始条件为 True 执行循环
for	迭代一个可迭代对象（例如，列表、字串或元组）的元素
try	对一组语句声明异常处理
with	启用上下文管理
def	声明一个用户定义的函式或方法
class	声明一个用户定义的类

if 语句。Python 的 if 语句的表现和我们所期望的一样。跟随关键字 if 之后的表达式用以计算结果为 True 或 False。最简单的形式如下，只是一条 if 语句和包含一条或多条从属语句的语句块。

```
if <expression>:
    statement
    (more statements as necessary)
```

要指定替代的动作，使用 else 语句：

```
if <expression>:
    statement
    (more statements as necessary)
else:
    statement
    (and yet more statement if necessary)
```

用 elif 语句（是“else if”的缩写）可创建可能的结果选单。同 if 语句一样，需要后跟一个表达式，但它只能出现在 if 之后，不能单独使用。

```
if <expression>:
    statement
    (more statements as necessary)
elif <expression>:
    statement
    (more statements as necessary)
else:
    statement
    (and yet more statements if necessary)
```

while 语句。逻辑控制表达式是真，while 语句块就重复执行。

```
while <expression>:
    statement
    (more statements as necessary)
else:
    statement
    (and yet more statement if necessary)
```

89 如果循环正常结束，且没有遇到 break 语句，则执行 else 语句块（即控制表达式计算结果为 False）。下例中，循环控制使用了一个布尔变量，它被初始化为 True，然后从循环内部赋值为 False。

```
>>> loop_ok = True
>>> loop_cnt = 10
>>> while loop_ok:
...     print "%d Loop is OK" % loop_cnt
...     loop_cnt -= 1
...     if loop_cnt < 0:
...         loop_ok = False
... else:
...     print "%d Loop no longer OK" % loop_cnt
...
10 Loop is OK
9 Loop is OK
8 Loop is OK
7 Loop is OK
6 Loop is OK
5 Loop is OK
4 Loop is OK
3 Loop is OK
2 Loop is OK
1 Loop is OK
0 Loop is OK
-1 Loop no longer OK
```

else 语句是完全可选的。

continue 和 break 语句也可以用来控制 while 语句重新循环或是退出。如果用 break 语句

终止循环，else 语句也将略过，continue 语句则不会略过随后的所有语句。

for 语句。Python 中的 for 语句不同于其他语言。在 Python 中，for 语句用于遍历序列对象的值。for 语句还支持一个可选的 else 语句，且与在 while 中的效果一样。

```
for some_var in <sequence>:
    statement
    (more statements as necessary)
else:
    statement
    (and yet more statement if necessary)
```

有一种指定序列整数对象的方法是使用内建函式 range()。

```
>>> for i in range(0,5):
...     print i
...
0
1
2
3
4
```


for 也常用以对序列对象进行处理，如“列表”：

```
>>> alist = [1,2,3,4,5,6,7,8,9,10]
>>> for i in alist:
...     print i
...
1
2
3
4
5
6
7
8
9
10
```

for 处理的对象也不必是整数。也可以是元组中的一系列字串。

```
>>> stuple = ("this","is","a","4-tuple")
>>> for s in stuple:
...     print s
...
this
is
a
```

```
4-tuple
```

和 while 语句一样，for 语句也支持 continue 和 break 语句，且工作方式相同。

try 语句。try 语句用以捕获和处理异常，它类似于 C++ 或 Java 中的 try-catch 结构。这对创建强壮的 Python 应用程序非常有用，可供程序设计者实现一个在发生错误时包含默认替代方法的处理流程（通常是产生所谓的回溯消息，然后终止执行）。完整的 try-except 控制结构大致如下：

```
try:
    statement
    (more statements as necessary)
except <exception, err_info>:
    statement
    (more statements as necessary)
else:
    statement
    (more statements as necessary)
finally:
    statement
    (and yet more statements if necessary)
```


可指定一个特殊的异常类型（<*exception*>），如果没有给出，则将尝试截获语句块中的任何异常。可使用 Exception 基类，并指定一个变量记录异常信息，从而探查发生了什么异常。

```
try:
    f = open(fname, "r")
except Exception, err:
    print "File open failed: %s" % str(err)
```

上例中，如果文件打开失败，程序将不会终止。相反，一个消息将被打印到标准输出，说明相关的失败信息。

如果没有异常，else 语句将被执行，而如果在 try 语句块中遇到了 break 或是 continue，则 finally 块将执行。有关 try 语句和 Python 中的异常处理的更多介绍请参阅 Python 文档。

字串

Python 能轻松创建包含格式化数据的字串，这点在许多程序中被广泛使用，本书中遇到的程序也不例外。Python 的字串对象提供了一套丰富的方法，与格式化字串结合可以生成带格式的列，左 / 右对齐字段，输出各种指定的数据类型。字串真是非常重要，值得我们单独讨论。

字串引用

字串使用下列形式之一引用：

```
'A single-quote string.'
"A double-quote string."
'''This is a multiline string using triple single quotes.
It is a medium-length string. '''
"""This is a multiline string with triple double quotes containing many
characters along with some punctuation, and it is a very long string
indeed."""
```

多行字串可以跨越多行，\n（换行符）会自动插入，以确保输出保持原始格式。

字串方法

字串类型提供了许多方法，其中有些我们已经看过了。表 3-15 是从 Python 2.6 文档中摘取的完整的列表（不包括 Unicode 的方法）。

表3-15：字串方法

capitalize	lower
center	lstrip
count	partition
decode	replace
encode	rfind
endswith	rindex
expandtabs	rjust
find	rpartition
format	rsplit
index	rstrip
isalnum	split
isalpha	splitlines
isdigit	startswith
islower	strip
isspace	swapcase
istitle	title
isupper	translate
join	upper
ljust	zfill

92

其中一些相对更加常用，不过了解一下到底可以用些什么也挺好。对于这里没有讲到的方法，请参考官方文档。此外，要记住以下形式

```
new_sring = "string text".method()
```

和下面的形式一样正确。

```
new_string = string_var.method()
```

记住，对于调用方法的结果所创建的新字串需要有一个名字（字串不可变），否则，修改后的字串数据会消失。

表 3-16 列出了 14 种常用的字串处理方法。其他不常用的方法将在需要时进行介绍。下述说明中，使用 Python 文档中的格式来表示 (必需的) 和 [可选的] 参数

93 表3-16：常用字串方法

方法	描述
capitalize()	将字串首字母大写后返回
center(width[, fillchar])	返回在指定长度 width 中居中的字串文本。如果指定了 fillchar，则使用指定字符来填充两端的空白。默认 fillchar 是空格
count(sub[, start[, end]])	统计指定的子串出现的次数。start 和 end 参数用以指定原始字串的探查范围
find(sub[, start[, end]])	定位指定子串第一次出现在原字串的位置，并返回这一索引值。如果子串没有找到，则返回 -1
isalnum()	如果字串是纯粹由字母和数字组成的（0..9，A..Z，a..z），则返回 True，否则返回 False
isalpha()	如果字串是纯粹由字母组成的（A..Z，a..z），则返回 True，否则返回 False
isdigit()	如果字串是纯粹由数字组成的（0..9），则返回 True，否则返回 False
islower()	如果字串是纯粹由小写字母组成的（a..z），则返回 True，否则返回 False
isspace()	如果字串是纯粹由空白字符组成的（空格，制表符，换行等），则返回 True，否则返回 False
ljust(width[, fillchar])	返回在指定长度 width 内左对齐的字串。如果指定了 fillchar，则使用指定字符来填充右端空白。默认填充字符是空格。如果宽度比原字串还小，则原封不动地返回
lower()	将字串所有字母小写后返回
rjust(width[, fillchar])	返回在指定长度 width 内右对齐的字串。如果指定了 fillchar，则使用指定字符来填充左端空白。默认填充字符是空格。如果宽度比原字串还小，则原封不动地返回
split([sep[, maxsplit]])	返回一个由指定分隔符切分开的字串条目组成的列表。如果没有指定分隔符，默认使用空格（空白可以是任何长度，只要它是连续的）。如果指定了 maxsplit，则仅返回指定数目的条目
upper()	将字串所有字母大写后返回

格式化字串

基本上有两种方法可以对 Python 数据进行格式化。首先是前面看到的串接。再就是利用 Python 的字串格式化能力。哪种方法最适合，这取决于你的目标。虽然串接对简单字串容易操作，但却不提供对小数位数等的处理，而且当字串中包含 多需要嵌入的数据和字串时，使用串接就很烦琐了，试考虑下面的例子：


```
>>> data1 = 5.05567
>>> data2 = 34.678
>>> data3 = 0.00296705087
>>> data4 = 0
>>> runid = 1
>>> outstr1 = "Run "+str(runid)+": "+str(data1)+" "+str(data2)
>>> outstr2 = " "+str(data3)+" : "+str(data4)
>>> outstr = outstr1 + outstr2
>>> outstr
'Run 1: 5.05567 34.678 0.00296705087 : 0'
```

还有更简易的方法。类似C中的sprintf()函式，Python支持非常相似的字串格式化占位符。通过使用特殊的格式代码，可以指定各种数据如何以及插入到一个字串的哪里。这里使用占位符字串输出与上例相同的处理。

```
>>> outstr = "Run %d: %2.3f %2.3f %2.3f : %d" % (runid, data1, data2, data3, data4)
>>> outstr
'Run 1: 5.056 34.678 0.003 : 0'
```

注意，所有要输出到字串中的变量都应该包含在括号中，它是一个封闭的 N 元组。如果省略括号，将只有第一个变量名是有效的，并会引发错误。

```
>>> "%d %d %d" % 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

字串格式化占位符语法如下。

```
%[(name)][flags][width][.precision]type_code
```

每个占位符可在 % 后的第一项指定一个可选名称（并必须用括号）。接下来是可选标识，用于对位、前导空格，符号字符和 0 填充。然后是一个可选的宽度值，指定数据最小的占位宽度，如果数据包含小数部分，则该值为精度字段，指明输出时的小数数位。最后是类型码，指定转换成什么样的数据（字串，整数，长整，浮点等）。表 3-17 列出了所有可用的标识。表 3-18 总结了可用类型代码。

表3-17：字串格式化占位符

标志	含义
#	使用“替代形式”格式化（参见表 3-18 的说明）
0	用前导零来填补数值
-	左对齐（若都指定，则会覆盖 0 标识）
（空格）	在有效数字前用空格补齐
+	在数值前加符号字符（+ 或 -）。会覆盖“空格”标识

表3-18：字串格式化占位符类型代码

类型代码	含义	备注
d	有符号十进制整数	
i	有符号十进制整数	
o	有符号八进制数	这时用前导零 0 是不处理的
u	过时的类型	用 d 替代
x	有符号十六进制数（小写）	这时用前导零 0x 是不处理的
X	有符号十六进制数（大写）	这时用前导零 0X 是不处理的
e	指数式浮点数 （小写）	另一种形式是使用小数点，即使后面没跟数字
E	指数式浮点数 （大写）	替代形式同 e
f	十进制浮点数 （小写）	替代形式同 e
F	十进制浮点数 （大写）	替代形式同 e
g	小写指数式浮点数，如果指数小于 -4 或精度低于 4 位，同十进制格式	替代形式是包含一个小数点和尾随零
G	大写指数式浮点数，如果指数小于 -4 或精度低于 4 位，同十进制格式	同 g
c	单字符（接受整数或单字符的字串）	
r	字串（对任何 Python 对象使用 repr() 函数转换）	
s	字串（对任何 Python 对象使用 str() 函数转换）	
%	无参数转换	

字串方法能和字串格式化同时应用。这可能看起来有点古怪，但完全有效。

```
>>> "%d %d".ljust(20) % (2, 5)
'2 5 '
>>> "%d %d".rjust(20) % (2, 5)
'                2 5'
>>>
```

因为实际上并没有将字串对象赋值给新的名字，只是 Python 在打印时应用了格式。

最后，Python 支持字串中包含所谓的转义字符。这是种由一个反斜杠字符接一个特殊字符所组成的双字串码，如表 3-19 所示。

表3-19：字串格转义序列

96

转义序列	解释	ASCII
\'	单引号	'
\"	双引号	"
\\	单反斜线	\
\a	ASCII 响铃	BEL
\b	ASCII 退格	BS
\f	ASCII 换页	FF
\n	ASCII 换行	LF
\r	ASCII 回车	CR
\t	ASCII 水平制表符	TAB
\v	ASCII 垂直制表符	VT

如果反斜线字符（\）位于行末，之后紧跟换行符（LF 或 CRLF），则表示行延续。这将导致新行被解释器忽略，而将前后代码行当成一行来处理。

程序组织

到目前为止，我们一直在 Python 的命令提示符里做事。现在，我们来看如何通过函式、类和方法创建程序模块。

作用域

早先提及的定义不在作用域是什么意思？ 现在开始探讨这事儿。

之前提过，Python 是将对象的名称作为集合的概念绑定到命名空间的。其实，一个命名空间更像是一个字典对象，其中键是名称，值是关联的对象值。在 Python 中有三个层次的命名空间：本地，全局，内建。图 3-6 显示了 Python 模块的命名空间范围。

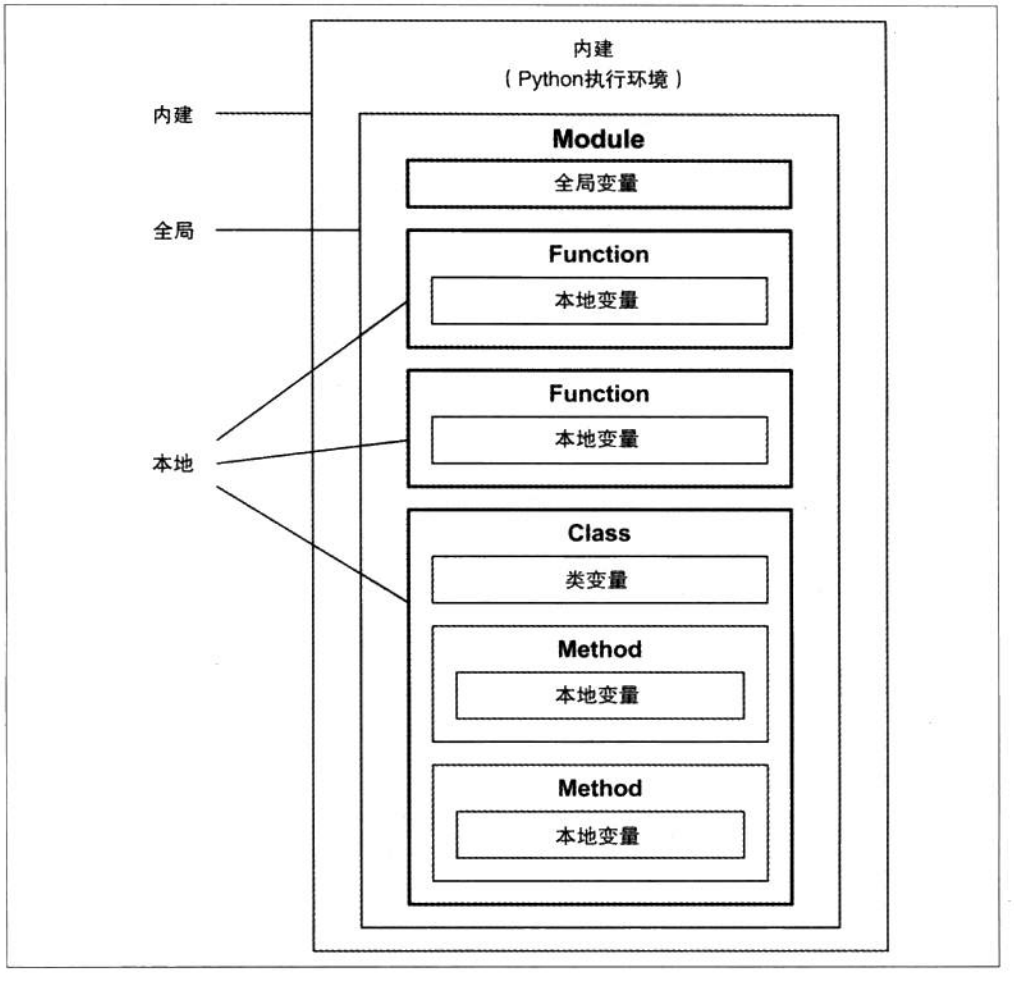


图3-6：Python的名称空间

如果名称是在函式或方法中使用，则首先尝试从一个本地名称空间搜索。其次，对全局名称空间搜索。最后，尝试内建名称空间。如果没有在任何空间发现该名称，则 Python 会抛出例外。图 3-7 展示了这一搜索原则。

97 本地作用域。本地作用域是指特定的函式、类或方法的名称空间。换言之，在一个函式中定义的任何变量都是局部的，在其外是不可见的。本地作用域还包括可能的嵌套函式。一会儿我们会研究这个情况。

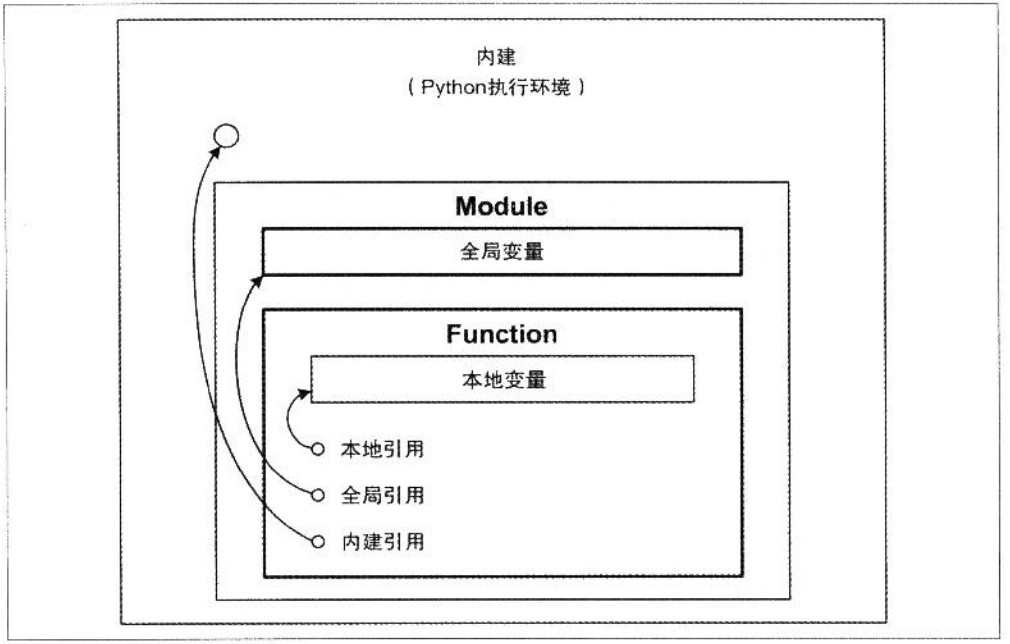


图3-7：名称空间的搜索层次

类对象对本地环境引入了另一个名称空间。在类对象中，任何在类的名称空间中定义的变量，对其内的任何方法而言都可以通过加前缀 self 的形式来访问。 98

```
self.some_var
```

一个类的对象实例的数据变量属性和方法可以通过使用“点号”形式从外部访问。

```
SomeObj = SomeClass()
SomeObj.var_name = value
```

这会将值分配给对象实例 SomeObj 的属性 var_name。如果 var_name 不存在，将立即创建。这使我们注意到这样一个有趣的现象：Python 的对象没有真正意义上的私有数据和方法。一切都可以方便地从外部读取，即使并不愿意。可以通过前缀下画线，来声明函式、类或变量，以防止由通配符被自动导入，但这也并不能真正隐藏之。甚至于用两个前导下画线字符来“糟蹋”对象名称，如果你知道方法，它仍然是可访问的。所以，这儿没什么真正的隐藏，责任赋予了程序员，要求我们要有礼貌，不看不该看的。

如果你并不确认这些究竟是什么意思，不用担心。我们将在之后开发用户界面应用时详细讨论。 99

全局作用域。全局作用域是指模块的名称空间。函式不能修改模块的全局变量，除非用 global 来声明。下面的示例脚本 globals.py 说明了这点：

```
# globals.py

var1 = 0
var2 = 1

def Function1():
    var1 = 1
    var2 = 2

    print var1, var2

def Function2():
    global var1, var2

    print var1, var2

    var1 = 3
    var2 = 4

    print var1, var2
```

为了测试，我们需要使用 import 语句加载它。这会让 Python 读模块和填充命令行的命名空间来发现有什么。

```
>>> import globals
```

一旦 globals.py 完成导入，我们就可以用 help() 函数来看里面是什么。

```
>>> help(globals)
Help on module globals:

NAME
    globals

FILE
    globals.py

FUNCTIONS
    Function1()

    Function2()

DATA
    var1 = 3
    var2 = 4
```

如果执行 Function1，就可以验证全局变量 var1 和 var2 未改变。

```
>>> globals.var1
0
>>> globals.var2
1
>>> globals.Function1()
1 2
>>> globals.var1
0
>>> globals.var2
1
```

不过，Function2 就能变更 var1 和 var2 的值。

```
>>> globals.Function2()
0 1
3 4
>>> globals.var1
3
>>> globals.var2
4
```

如果一个函式想修改的变量名和某全局变量相同，那么变量在函式内部使用时，就必须使用 global 加以区分。globals2.py 说明了这一情况。

```
# globals2.py

var1 = 0
var2 = 1

def Function1():
    print var1, var2

def Function2():
    var1 = 1
    var2 = 2

    print var1, var2

def Function3():
    print var1, var2

    var1 = 1
    var2 = 2

    print var1, var2
```

观察调用这三个函式时的反馈。

```
>>> import globals2
>>> globals2.Function1()
0 1
>>> globals2.Function2()
1 2
>>> globals2.Function3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "globals2.py", line 14, in Function3
    print var1, var2
UnboundLocalError: local variable 'var1' referenced before assignment
```


Function1() 成功了，因为没有局部变量和模块变量之间的冲突。Function1() 的 var1 和 var2 局部变量都在函式内部定义，是没有问题的。然而，Function3() 引发了错误。在这里，全局名称被阻止使用，因为相同名称已经放入函式局部空间，但第二次打印时，名称无法绑定到想打印的本地值对象上。因此，触发 UnboundLocalError 例外。如果在 print 语句之前声明 global，就不会发生错误。

内建作用域。内置的名称空间是指 Python 运行时环境。它包括类似 abs()，print 和各种异常的名称等等。如果你想得到一个内置名称的列表，只需在 Python 提示符输入 dir(__builtins__)。这里没有列出所有输出，是因为它相当大（至少有 144 个）。

模块和包

一个 Python 的源代码文件被称为一个模块。这是一个由变量定义语句组成语句集合，包含变量定义语句、import 语句、直接执行语句、函式定义语句和类定义语句。

模块包含于软件包中，而包实际上就是目录，其中包含一个或多个模块。包也可包含其他包。图 3-8 演示了这一图景。

模块是一个对象，因为我们已经看到，它有它自己的命名空间。一个模块也有类似其他 Python 对象的属性。模块的属性包括函式、类、方法，以及在其命名空间中定义的变量。

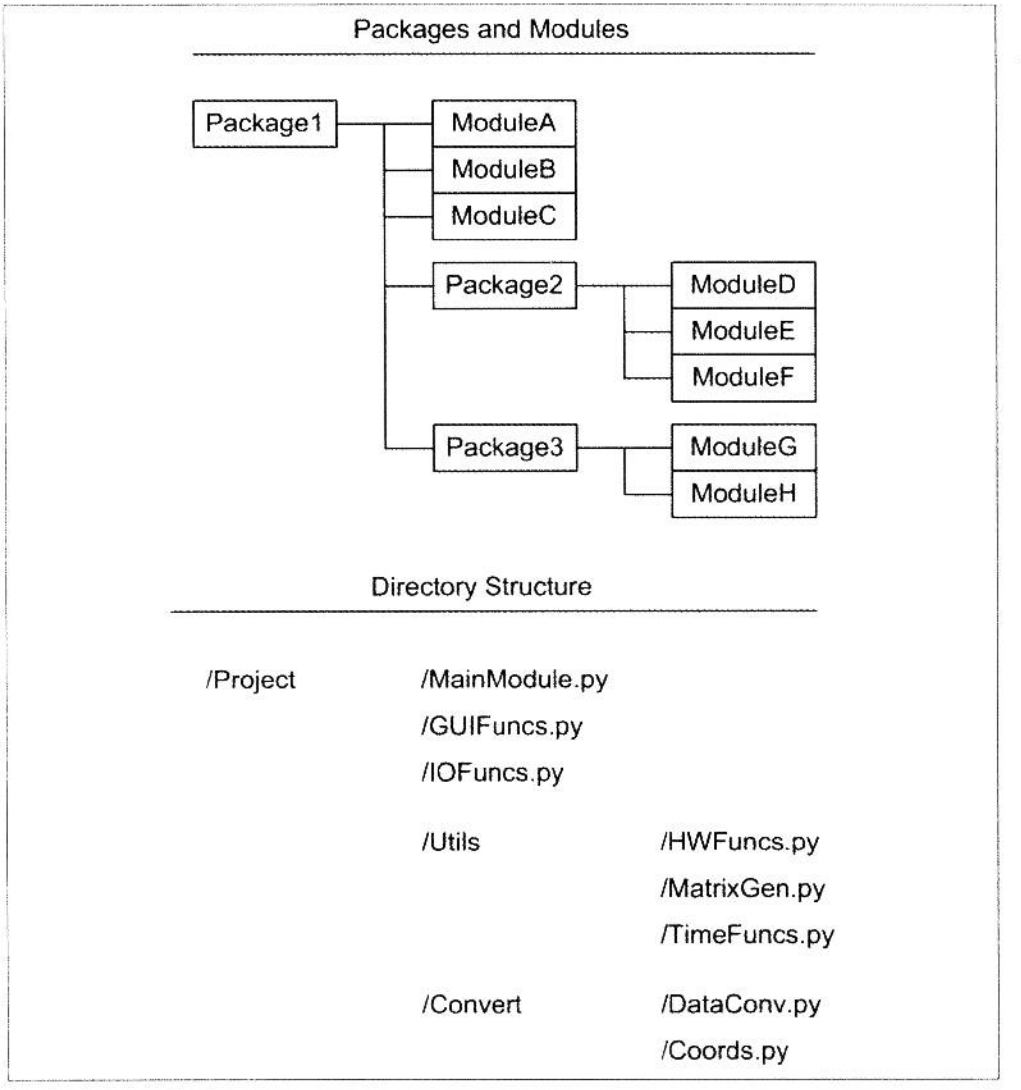


图3-8：包和模块

函式、类和方法

def 语句用于定义在模块中的函式和类中的方法。

```
def SomeName (parameters):
    """ docstring goes here.
    """
    local_var = value
```

```
    statement...
    statement...
    more statements...
```

102 def语句用于定义函式时，缩进和同级函式相同。当用于定义类方法时，相比类的声明，def语句要缩进一层。

103 函式和方法可以嵌套。这时，内部函式不能从外部访问。下面是一个嵌套函式调用的例子（subfuncs.py）。

```
#subfuncs.py

def MainFunc():
    def SubFunc1():
        print "SubFunc1"
    def SubFunc2():
        print "SubFunc2"
    def SubFunc3():
        def SubSubFunc1():
            print "SubSubFunc1"
        def SubSubFunc2():
            print "SubSubFunc2"
        SubSubFunc1()
        SubSubFunc2()
SubFunc1()
SubFunc2()
SubFunc3()
```

我们只能执行函式 MainFunc()；MainFunc() 名称空间外部是访问不到嵌套的函式的。如果导入 subfuncs，设法得到它的帮助，将看到：

```
>>> import subfuncs
>>> help(subfuncs)
Help on module subfuncs:
NAME
    subfuncs
FILE
    subfuncs.py
FUNCTIONS
    MainFunc()
```

但是，当执行 MainFunc() 时，可以看到所有子函式都执行了。

```
>>> import subfuncs
>>> subfuncs.MainFunc()
SubFunc1
SubFunc2
SubSubFunc1
```

```
SubSubFunc2
```

class 语句定义了一个类的对象，而这又是用来创建实例对象的。下面的类定义了一个定时器对象，可用于获取程序执行的运行时间。

```
import time

class TimeDelta:
    def __init__(self):
        self.tstart = 0
        self.tlast = 0
        self.tcurr = 0

        self.Reset()

    def GetDelta(self):
    """ Returns time since last call to GetDelta(). """
    self.tcurr = time.clock()
    delta = self.tcurr - self.tlast
    self.tlast = self.tcurr
    return delta

def GetTotal(self):
    """ Returns time since object created. """
    return time.clock() - self.tstart

def Reset(self):
    """ Initializes time attributes. """
    self.tstart = time.clock()
    self.tlast = self.tstart
```

104

此类的对象可在代码的任意位置实例化，并检查运行时间，而且可多次实例使用。如果 TimeDelta 是模块中的函式，将是相当尴尬的，但作为一个类，每个实例都可以保持自己的数据，以便启动时自行使用。

文档字串

文档字串是用来对模块、类、方法和函式提供文档的。在一个模块、函式、类或方法开始时的多行字串，将被 Python 认作文档字串，存储到对象的内部变量 __doc__，这就是在命令行环境中键入 help() 看到的文档。

下面的例子显示了如何使用文档字串。pass 语句使脚本可用，只要我们导入此代码，就可以使用 help() 来显示嵌入式文档。

```
#docstrings.py

""" Module level docstring.
```

```
    This describes the overall purpose and features of the module.
    It should not go into detail about each function or class as
    each of those objects has its own docstring.
"""

def Function1():
    """ A function docstring.

        Describes the purpose of the function, its inputs (if any)
        and what it will return (if anything).
    """
    pass

class Class1:
    """ Top-level class docstring.

        Like the module docstring, this is a general high-level
        description of the class. The methods and variable
        attributes are not described here.
    """

    def Method1():
        """ A method docstring.

            Similar to a function docstring.
        """
        pass

    def Method2():
        """ A method docstring.

            Similar to a function docstring.
        """
        pass
```

105

对该模块调用 help() 时，应该可以看到以下结果。

```
>>> import docstrings
>>> help(docstrings)
Help on module docstrings:

NAME
    docstrings - Module level docstring.

FILE
    docstrings.py

DESCRIPTION
```

```
    This describes the overall purpose and features of the module.
    It should not go into detail about each function or class as
    each of those objects has its own docstring.

CLASSES
    Class1

    class Class1
     | Top-level class docstring.
     |
     | Like the module docstring, this is a general high-level
     | description of the class. The methods and variable
     | attributes are not described here.
     |
     | Methods defined here:
     |
     | Method1()
     | A method docstring.
     |
     | Similar to a function docstring.
     |
     | Method2()
     | A method docstring.
     |
     | Similar to a function docstring.

FUNCTIONS
    Function1()
        A function docstring.

        Describes the purpose of the function, its inputs (if any)
        and what it will return (if anything).
```

106

模块导入

Python 的模块通过导入语句，即可使用其他模块的功能。一个模块导入 Python 时会首先检查是否已经导入过，若有，将会指向当前的命名空间中的同名对象，否则，会加载指定的模块，扫描它，并把所有名称添加到当前的命名空间。请注意，“当前的命名空间”可能指本地空间中的函式、类或方法，或者，也可能是全局命名空间中的一个模块。

在 Python 模块中的语句、函式或方法在加载前不会运行。这意味着模块的导入语句、赋值语句、函式和类定义语句，加载时会执行。但是函式或是方法内部的语句只有在调用时才会执行，当然，函式和类定义语句执行时，相应对象也将创建。

导入方法

导入语句有很多形式，下面这种是最常见的。

```
import module
```

模块中的对象添加到当前名称空间后，使用 module.function() 或 module.class() 来调用。要访问一个模块内的数据参数，使用 module.variable 形式。

别名引用是支持的。

```
import module as alias
```

这其实是进行了相同的模块导入，这样别名就可用来调用模块内部资源了。当一个模块有很长的名字时，这样导入使用起来很方便。例如：

```
import CommonReturnCodes as RetCodes
```

也可以从模块只导入一个指定的对象：

```
from module import somename
```

这种形式可从一个模块导入特定函式、类、模块或数据。这时可以使用不带模块的前缀形式来调用函数或参数 somename。

也可使用通配符来导入外部模块的一切，以添加到当前命名空间：

```
from module import *
```

107 一般认为，除非在某些特殊情况下才用通配符来导入，否则最好不用。尽量显式地指定导入对象，以免和现有名称空间中的对象名称冲突。因为除非有预防措施，通配符将导入模块的一切，如果和当前模块有的名称相同，目前同名对象将被覆盖。

另外，可以通过在属性名称前缀单或双下画线来控制是否暴露。如果有属性如下声明：

```
_some_name
```

在通配符导入中将不包括，但它仍然可以通过前缀模块名来引用到。如果用双下画线的形式来声明：

```
__some_name
```

这可能是 Python 中最常见的数据隐藏形式。而实际上，它仍然可以从父模块外部访问到，但其对外的名称是种“错位”，从而难以获得（但，不是不能获得）。

导入处理

因为 Python 一遇到 import 语句就会立即执行，并会追踪所有 import 语句，所以它会以深度优先的方式来逐一导入，直到所有涉及的模块都被处理。图 3-9 图形化显示了导入是如何工作的。

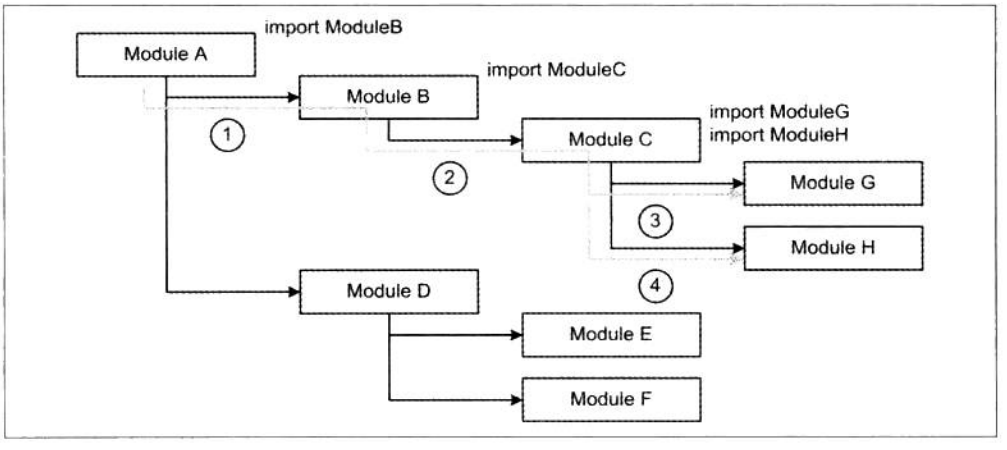


图3-9：模块导入序列

图中的模块导入序列使用数字表示在循环。模块 A 导入模块 B，B 又导入模块 C，从而导入模块 G 和 H；模块 D 将在模块 H 完成加载后才尝试导入。

108

循环导入

Python 的导入体制有个缺点，就是可能引发导入的“挂起”，即所谓的“循环导入”。考察一下图 3-10 的情况。

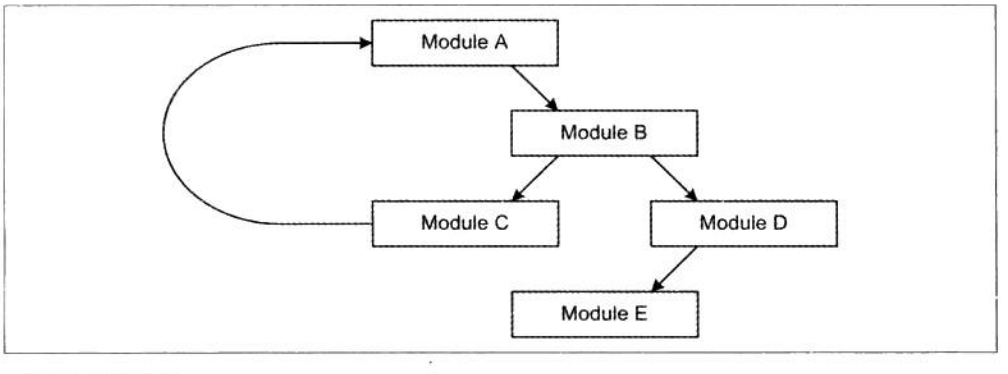


图3-10：循环导入

在这里，我们有个模块 A 将导入模块 B 的情况，当进一步导入模块 C 和 D 时，模块 C 又要求导入模块 A，这时 A 模块还在等待模块 B 完成对模块 C 的导入，所以，模块 B 无法继续完成模块 D 的导入，因为此时模块 C 并没有完成加载，于是整体过程死锁。

一个确定的避免循环导入的方式是坚持“决不向上导入，永远向下导入”的原则。这意味着模块应该分层导入，同时，模块实现时也没有必要从一个更高的层次导入。很多新人的典型错误是把一堆“假变量”（不期望有值改变的变量）设置在模块中给其他相关的模块功能使用，然后再导入整个模块仅为获得“假变量”对象。事实上各个模块中类似“假变量”的对象，是应该包含在自身中，这样就不必担心引发循环导入了。

加载并运行 Python 程序

下面的例子是一个完整的 Python 程序，其中不包含函式和类定义，也就是通常所说的“脚本”。它将产生一个包含随机数据的 PGM 格式的图像文件。结果看起来像旧式的电视调到一个空频道时满屏“雪花”的样子，这里的要点是来看一下，真实小 Python 程序的样子。任何图像浏览器都能够处理 PGM 文件并加载和显示出图像（ImageJ，一个免费的工具，可从 http://rsbweb.nih.gov/ij/ 下载，在这方面表现不错，可从 http://netpbm.sourceforge.net 这里获取关于 PGM 格式的详细信息）。

运行这一程序，不用启动 Python 交互环境，只需要在命令行中在脚本文件名前加上 python：

```
C:\samples\> python pgmrand.py
```

在 Linux 中类似：

```
/home/jmh/samples/% python pgmrand.py
```

在你的系统中，具体提示可能有所不同（除非你也将脚本保存在 samples 目录）。

若在使用 Linux，可能需要在脚本第一行加入以下声明：

```
#! /usr/bin/python
```

在其他系统，可能要修订路径以指向 Python 真实的安装目录，比如说 /usr/local/bin/python。

源代码如下。

```
""" Generates an 8 bpp "image" of random pixel values.

The sequence of operations used to create the PGM output file is as follows:
    1. Create the PGM header, consisting of:
```

```
        ID string (P5)
        Image width
        Image height
        Image data size (in bits/pixel)
    2. Generate height x width bytes of random values
    3. Write the header and data to an output file
"""
import random as rnd # use import alias for convenience

rnd.seed() # seed the random number generator

# image parameters are hardcoded in this example
width = 256
height = 256
pxsize = 255 # specify an 8 bpp image

# create the PGM header
hdrstr = "P5\n%d\n%d\n%d\n" % (width, height, pxsize)

# create a list of random values from 0 to 255
pixels = []
for i in range(0,width):
    for j in range(0,height):
        # generate random values of powers of 2
        pixval = 2**rnd.randint(0,8)
        # some values will be 256, so fix them
        if pixval > pxsize:
            pixval = pxsize
        pixels.append(pixval)

# convert array to character values
outpix = "".join(map(chr,pixels))❶

# append the "image" to the header
outstr = hdrstr + outpix

# and write it out to the disk
FILE = open("pgmtest.pgm","w")
FILE.write(outstr)
FILE.close()
```

110

❶ 字串的 join() 方法和 map() 函式在此用以创建输出字串并写入图片文件。

这是个非常值得回味的程序。其中唯一“棘手”的部分就是使用 join() 和 map() 方法来输出字串。因为 Python 没有天然的 byte 类型，只有一个 chr 类型来处理字串。所以，想要使用 byte 数组，就得自制一个字串扫描实现，针对每个字符进行转换，并串接到一个空字串中（"".join(map(chr, pixels)) 语句就是完成了这一转换）。注意，所有可以输出的参数是硬编码在脚本中的。

基础输入输出

程序必须使用一些手段来输入数据和输出结果以便完成通常的任务。Python 提供了几种方法来实现这两个目标：使用控制台命令行和文件对象。稍后，我们将研究类似串行端口、USB、网络插口和数据采集等硬件。但现在让我们来看看开箱状态下的 Python 可以做什么。

用户输入

从 stdin（标准输入）获取用户输入是最直截了当的。Python 提供了 raw_input() 来做这事儿。

111 模块 getInfo.py 包含了具体如何使用 raw_input() 的简单实例。

```
# getInfo.py

def ask():
    uname = raw_input("What is your name? ")
    utype = raw_input("What kind of being are you? ")
    uhome = raw_input("What planet are you from? ")
    print ""
    print "So, %s, you are a %s from %s." % (uname, utype, uhome)
    uack = raw_input("Is that correct? ")
    if uack[0] in ('y', 'Y'):
       print "Cool. Welcome."
    else:
       print "OK, whatever."
```

观察这个脚本如何工作，我们要导入 getInfo 模块，并调用 ask()。

```
>>> import getInfo
>>> getInfo.ask()
What is your name? zifnorg
What kind of being are you? Zeeble
What planet are you from? Arcturus III

So, zifnorg, you are a Zeeble from Arcturus III.
Is that correct? y
Cool. Welcome.
```

raw_input() 函数接受一个可选的提示字符串，并将自 stdin 获取的数据总是作为字串返回。如果程序期待数值，就需要转换。一种安全的方式是使用 try-except 来处理，以下是 getInfo2.py 的 try-except 修订。

```
def ask2():
    uname = raw_input("What is your name? ")
    utype = raw_input("What kind of being are you? ")
    uhome = raw_input("What planet are you from? ")
```

```
getgumps = True
while (getgumps):
    intmp = raw_input("How many mucklegumps do you own? ")
    try:
        ugumps = int(intmp)
    except:
        print "Sorry, you need to enter an integer number."
        continue
    else:
        getgumps = False
print ""
print "So, %s, you are a %s from %s, with %d mucklegumps."\
    % (uname, utype, uhome, ugumps)
uack = raw_input("Is that correct? ")
if uack[0] in ('y', 'Y'):
    print "Cool. Welcome."
else:
    print "OK, whatever."
```

在我们继续之前，对这个简单功能有几件事情要考虑。首先，它将只接受有关 112
mucklegumps 数目的 ，浮点数和字串都将被拒。其次，没办法在用户输入时优雅的警告并中止。这方面可以简单的通过特殊字符 (例如 a.) 来检查处理，或只检测空输入 (只按回车键而没有输入)。说到空的输入，如果用户在最后一个问题直接按回车键，Python 会抛出一个异常 ：

```
Is that correct? <enter>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "getInfo2.py", line 18, in ask2
    if uack[0] in ('y', 'Y'):
IndexError: string index out of range
```

在 if 表达式中，对 uack[0] 判定是否匹配 2 元组 ("y", "Y") 中任意一个，当直接回车时，返回零长度字串，将触发异常并得到捕获处理 ：

```
uack = raw_input("Is that correct? ")
try:
    if uack[0] in ('y', 'Y'):
        print "Cool. Welcome."
    else:
        print "OK, whatever."
except:
    print "Fine. Have a nice day."
```

在处理用户输入 (即，一个人在提示符状态下输入时)，必须始终意识到可能的输入错误或异常。人类总是会键入错误的超出限定的数据，意外的单词或短语，或是什么都没有。用户是不可预知的，所以，建设一个能捕获错误输入的程序永远是个好主意。

命令行参数

在命令行输入的程序参数由系统捕获并通过 Python 解析器变为列表。这一列表的第一个条目（索引值为 0）永远是程序本身。Python 的内建模块 sys 包含了这种数据处理方式。

以下程序 (argshow.py) 简单指出了如何打印出所有通过命令行获得的参数列表：

```
import sys

print "%d items in argument list\n" % len(sys.argv)

i = 1
for arg in sys.argv:
    print "%d: %s" % (i, arg)
    i += 1
```

113

这是应该的运行结果：

```
C:\samples> python argshow.py 1 2 3 4 -h -v
7 items in argument list

1: argshow.py
2: 1
3: 2
4: 3
5: 4
6: -h
7: -v
```

Python 还提供用于检测特定的参数，并提对应取值的模块，我们当前不讨论这方面的技巧。将在后面的章节来使用他们。

文件

Python 有一个内建的文件对象，提供了一些对磁盘文件的基础读写等各种处理方法。在 pgmrand.py 中我们已经看到了一点，现在来继续深入。

open() 方法用以创建一个文件对象实例：

```
>>> fname = "test1.txt"
>>> fmode = "w"
>>> f = open(fname, fmode)
```

当然，可以声明为可写：

```
f = open("test1.txt", "w")
```

一样获得了文件对象。

一但我们拥有了文件对象，就可以通过 write() 写入点东西：

```
>>> f.write("Test line 1\n")
>>> f.write("Test line 2\n")
>>> f.close()
```

文件现在应该包含两行文本：

```
Test line 1
Test line 2
```

注意，字串中写入了 \n（换行符的声明代码）。文件的 write() 方法不象 print 那样会在字串尾部自动追加换行符，所以必须被显式地包含在字符串中。

表 3-20 列出了常见的文件模式。

表3-20：文件I/O模式

114

模式	含义
r	读
rb	读二进制流
w	写
wb	写二进制流
a	追加
ab	二进制流追加

表 3-21 列出了一些文件对象常用的方法。其他更多可用方法，请参阅 Python 的文档。

表3-21：文件方法

方法	描述
close()	关闭一个文件
flush()	刷新内部缓冲区
read([size])	从文件中读取指定大小的字节
readline([size])	从文件中读取指定行数的内容
write(str)	将字串写入文件

控制台的打印输出

我们已经看到了 Python 的 print 函式的输出。其主要功能就是发送结果到当前定义为 STDOUT（标准输出）的通道中，print 函式可以透明的处理数字类型和字符串之间的转换，以便在控制台输出。先前有关章节已经讨论过字串格式化，配合 print 可以很好的完成格式化输出。

重定向打印

默认情况下，print 输出被发送到目前任何已定义的 STDOUT。通过使用“锯齿形”（>>）操作符，print 行为可以修订输出到任何提供了 write() 方法的对象中去。通常，这将是一个文件对象，如下所示：

```
>>> datastr = "This is a test."
>>> f = open("testfile.txt", "w")
>>> print >> f,datastr
>>> f.close()
```

115 提示和技巧

这里有个可能对你有用的意见收集。

模块全局变量

在模块文件头部先初始化一些全局变量，这通常是一个好主意。检查一个全局变量是否不存在将导致一个异常，使用前先确保存在，将避免后续的一些恶化。

潜在缺陷

将模块被导入时，其中的内部语句除了 def 语句其它并不被立即执行，bug 最可能潜伏其中，做到被真正调用，才显化。这时 try 语句是强大的盟友，但它并不是包治百病。良好的单元测试才是检测和消除这种缺陷的关键。

延期导入

有时候，你可能会遇到代码原作者试图用延期导入的方式解决循环导入问题，有问题的模块导入被放置在函式或是方法中，而不是在模块文件的顶部。虽然这在 Python 语法中是允许的，但被认为是很挫的形式，遭遇到它的明确表现就是有人坐在键盘前对这种脑残的设计骂个不停。然而，当处理遗留代码（或只是写得不好的代码）时，很可能无法避免使用这一招。强烈建议只有当你真正必须用并进行彻底测试后才谨慎使用。

字典作为函数参数

虽然 Python 允许任何数据对象可用作函数或方法的参数，但除非你有足以令人信服的理由，否则不要使用字典对象来作为参数使用。如果将字典对象用作参数，就必须详细记述含义，尽力避免结构变动，即便这样，依然会因为可动态改变结构以及各种字典对象的共享，导致无法理解和噩梦般调试。它甚至可视为是种无意被理解的混淆形式。基于同样的理由也不建议把列表作为参数使用。

函式返回值

元组是一个方便的形式，可以从函数返回多个值。例如，可用 2 元组的形式，一次性返回状态代码和数据值对。使用时，通过审查状态码确认函数是否成功，如果是 OK，余下就是得到的数据值。

模块作为对象

在 Python 中模块当然也是个对象（记得嘛，在 Python 世界一切都是对象），但是在 C 或是 C++ 倾向将模块视作源代码块，除了比较整洁的组织一些数据封装之外，并没有其它作用。下面是 wxPython GUI 的一个模块（进一步的事件 ID 含义将在以后章节讨论）：

```
# ResourceIDs.py

import wx

# File
idFileSave              = wx.NewId()
idFileSaveAs            = wx.NewId()
idFileNew               = wx.NewId()
idFileOpen              = wx.NewId()
idFileOpenGroup         = wx.NewId()
idFileClose             = wx.NewId()
idFileCloseAll          = wx.NewId()
idFilePrint             = wx.NewId()
idFilePrintPreview      = wx.NewId()
idFilePrintSetup        = wx.NewId()
```

wxPython 包含有 NewID() 的函数，用以在每次调用时自动分配新的 ID 号。当 ResourceIDs 完成导入时，相关语句和变量就完成对应数据对象的分配。使用其中输入的一个模块如下（也可使用一个别名）：

```
import ResourceIDs as rID

event_id = rID.idFileSave
```

这在大项目中很方便，尤其是那些 GUI 框架需要大量的事件 ID 时。从任意其它模块也可以安心导入纯数据模块，不会引发循环导入，因为其本身并没有导入其他任何模块（也许除了系统级模块）。如果纯数据模块中的属性名称是全程唯一的（每个名称使用一个特殊的前缀），那也可以被安全地用通配符风格进行导入。

使用文档化注释

曾几何时，有位物理学教授告诉我："如果明天你失忆了，那么文档就是你实验室中一切意义的所在。"可以肯定这是明智的建议。但很多人就是不愿意花费必要的时间在代

码中进行注释。这是愚蠢的，因为没有人可以预料什么东西重要，或是半年后什么东西将产生作用（还有人认为至多两个月一切都将改变）而且，注释应该包含作者对软件的感觉，以帮助后来者有效理解和修订代码。

117

代码风格

“PEP-8”文档（参阅 http://www.python.org）包含了一些编码风格的指导方针。你可以不同意这一切，但你至少应该查阅并熟悉它。那里包含了太多很好的经验。任何情况下，你都应该为您的代码保持某种类型的一贯风格，没有任何理由比维护代码的可读性，会令事情变得容易，特别是你必须重温旧代码时。

Python 开发工具

成功和之间的差异可能就是一个良好的发展环境！开发环境至少必须提供方式来支持，创建和编辑标准 ASCII 文本格式的 Python 源代码。其他配套工具，如调试器，自动文档生成和版本控制，都是很好的，不是绝对必要的。幸运的是，有太多可选择的，优秀的开放源码软件（自由和开放源码软件）或是廉价的很好的商业工具。

本节我们将以自由和开源软件为重点来简要介绍哪些是可用。真正的问题不是你用的什么工具，毕竟大多数人都（在过度的开发时间里）有自己的喜好和工作习惯。重要的是在多种选择中，决择简洁够用的工具组合来完成任务。

编辑器和 IDE

最低限度，需要一个文本编辑器或集成开发环境（IDE）来输入和编辑 Python 脚本。同时还需要编辑 C 代码来完成扩展（第 5 章我们将深入如何创建 Python 扩展），因此选择一种语言无关，或是能够识别不同语言并有语法高亮的环境。

编辑器和 IDE 之间的主要区别在于从工具本身可以完成多少操作。通常编辑器，只允许你做一件事：编辑。而一个 IDE，可以让你从编辑，到编译，调试，甚至度量和版本控制等各种事务。IDE 的目的是开发人员在整个开发过程中不必离开环境。

有些编辑器，也有从内启动另一个程序的能力，然后捕捉和显示程序的输出，但是这通常是一个附加的能力，本质上不是编辑工具的一部分，有些编辑器这种支持能力比别人更好很多（是的说的是谁，大家都知道！）一个全功能的 IDE 则是以某种形式集成所有这些功能，虽然有些 IDE 也需要来自外部工具的功能和应用。换句话说，能高度扩展功能的编辑器和 IDE 有时是很难区分的。

118

对于 Python 可能 IDE 不是必要的（虽然市面上有少数选择），因为它不是编译语言，大

部分使用 Python 的情况是在命令行或在包含 Python 应用的图形用户界面。

编辑器

如果你认为使用一个标准编辑器就足够了（这正是笔者使用的方式 ;-），有几个优秀的套件可供选择。表 3-22 列出了一些较受欢迎的推荐。

表3-22：简短的文本编辑器一览表

名称	操作系统	自由开源软件？	赞誉	吐糟
Emacs	Linux Windows 其他	是	支持先进的编辑功能，脚本，语法高亮，和多窗口显示	有一些陡峭的学习曲线，并使用很多不直观的必须记忆的多键组合命令
vi/vim	Linux Windows 其他	是	基本功能是简单易学的，在不同的Linux和类Unix平台中vi被非常广泛的支持。VIM除了传统的命令行操作还提供了一个GUI界面	学习更复杂和更高级的功能将是个艰难的过程，同样依赖不直观的组合键
nano	Linux	是	非常简单，支持部分语法高亮	基于pico编辑器及其控制键命令。能力有限
Slickedit	Linux Windows 其他	不是（很贵）	大量的功能，完整的GUI界面，可编程的宏，语法高亮。能仿效其他编辑的能力	对于大多数开发任务来说，需要学习过量的操作细节，而且价格超贵
UltraEdit	Linux Windows	不是	完整的GUI界面，非常简单易学，多个标签式的文本窗口，可编程的宏，和语法高亮	有很多一般开发者永远用不到的功能，需要努力才能弄清楚如何调整默认设置，来关闭一些不必要的默认值。需要投入金钱（而且不仅仅）

这仅仅是个部分编辑器的列表，其他还有大量可用编辑器，其中包括很好的开源软件。如果你还没找到一个非常喜欢的编辑器（或自己开发了一个），在你的开发平台中值得多尝试比较一下。但是注意：有些人似乎对特定的编辑器陷入了狂热。特别是 Emacs 和 vi 尤为明显，两者的持续纷争已远超过 20 年（参考 http://en.wikipedia.org/Editor_war）。其实，只要保持开放的心态，为工作选择正确的工具，至于神马 " 编辑器战争 " 就当免费娱乐！

119

IDE 工具

IDE 环境，试图整合一切程序员的需求到一个单一的工具中。第一种流行的为 PC 设计的低成本 IDE 是 Borland 的菲利普卡恩在 80 年代中期开发的 Turbo Pascal，大多数现代 IDE 都提供源代码文本编辑器，编译器或解释器的接口，自动编译工具，也许十包含一些版本控制的支持，以及某种形式的调试器。换句话说，它是个针对软件开发的一站式购物体验。我们这里列出的 IDE 并不是每一个都提供一切功能，但最起码，你应该期望一个文本编辑器和的运行外部工具和应用程序（诸如如编译器，解释器，和调试器）的能力。在这个意义上说，即使类似 UltraEdit 和 Emacs 的编辑器（表 3-22 列出过）也可用作集成开发环境（实际上通常就是这么用的）。表 3-23 列出了一些现成的适合用作 Python 开发的 IDE。

表3-23. 简短的IDE一览表

名称	操作系统	自由开源软件？	赞誉	吐槽
Boa	所有支持 Python 和 wxPython 的系统	是	用于创建和维护 wxPython 的 GUI 组件和应用程序的优秀工具；包括一个像样的编辑器和一个基本的 Python 调试器	仅为 wxPython GUI 包开发作了优化，无法作为通用全功能的开发环境
Idle	所有支持 Python 的系统	是	用 Python 本身支持对 Python 脚本的编程；提供多编辑窗口，函数/方法提示清单，Python shell 窗口，初级调试	Idle 的编辑窗口是自由浮动，有时要追踪个特定的窗口是很恼人的
Eclipse ()	Linux Windows 其他	是	由 Java 编写的异常灵活的多语言 IDE；通过插件模块提供各种开发支持，如 PyDev 对 Python 开发支持	项目/包的开发模型，需要应对相当陡峭的学习曲线，可能不是所有人都能适合
PythonWin	Windows	是	由 ActiveState 的 Python 发行提供，包含 Idle 的大部分相同功能	仅仅在 Windows 平台可用
WindIDE	Linux Windows 其他	不是	专门面向 Python 开发和调试提供很多功能	专门为 Python 开发服务的编辑器，当然，也可以用来写其它语言。界面可能过于纷乱，通常需要化时间来配置好

调试器

120

调试器允许一个软件开发人员在软件运行时深入其内部观察。尽管人们也许认为调试器是很少被使用的，其实在必要时，调试器可以迅速揭露程序中的严重问题，节省大量的调试时间。然而，正如其它成瘾物质，一个不小心调试器也将发展为一种严重的依赖症状！

那么，通过调试器我们究竟可以作什么？首先，调试器允许开发人员在源代码中选择一个特定的代码行，设置中"断点"，当程序执行到这一点，就可以制止程序运行并检查局部变量。调试器还支持对代码进行单步执行。如果调试器有支持"观察"，还可以选择特定的变量在"断点"处显示值，并逐句执行代码显示变量的变化。

目前没有"一刀切"式调试器，可以对任何语言进行调试，虽然也有一些"Shell"为多种开发语言提供了类似接口。

对于 Python，Boa，Idle，Eclipse，和 WingIDE 工具都包含了调试器。有种独立的 Python 调试器 Winpdb，也可用，以及 Python 本身还内置了一个命令行界面的调试器 PDB。

小结

对 Python 的简要游览就这样结束了。现你应该对 Python 看起来是什么样儿的有个一般性的感觉了，我故意掩盖了语言的许多方面，因为，毕竟这本书不是 Python 教程。正如一开始就说过的，有关 Python 语言本身有许多优秀的书籍，可以提供丰富的细节，而且在 Python 官方网站有关语言的一切，都有权威人士可供答疑。接下来，随着讨论的继续，我们会遇到 Python 的其他功能，若有必要，就地研究。

推荐阅读

如果你想进入更深的 Python 编程境界，从以下书籍开始吧：

Python in a Nutshell, 2nd ed. (http://oreilly.com/catalog/9780596100469/) Alex Martelli, O'Reilly Media, 2006.

绝对应该常备案头的简明参考。精心组织成方便查阅的形式，当你工作时，想快速寻找一个可用的思路时，这是个万用工具袋。

Programming Python, 3rd ed. (http://oreilly.com/catalog/9780596009250) Mark Lutz, O'Reilly Media, 2006.

1600 页的巨型手册，全面涵盖 Python 中从字符串到 GUI 的一切，是 Python 工作必备图书。

121

除了已在本章引用过的URL，网络中还有许多在线资源可用于Python：

http://diveintopython3.org

发布了Mark Pilgrim所撰图书：《深入Python》的完整文本还提供了PDF下载。书中提供了以练带学的方式，并使用无数的例子来说明关键概念和技巧。[译注3]

http://effbot.org

Fredrik Lundh的博客。在这里，可以找到数百篇有关Python的文章，下载和公开的图书，以及各种开发研究和尝试。行文有趣并有用，而且都是非常有见地并实用的。

译注3：也有人指出，这书研究Python的方式有些太学术化了，根本不实用。译者个人推荐包含在Python发行版文档中的教程，这是Python创始人Guido唯一发布的正式图书了。

第 4 章 123

C语言编程

底层编程对于程序员的灵魂是有益的。

——John Carmack, Id Software 联合创始人

C 语言是 Ken Thompson 等人在 1969 年到 1973 年期间，作为 UNIX 早期开发工作的一部分在贝尔实验室创建的。从那时起 C 语言就是 UNIX 不可分割的部分，UNIX 操作系统大部分就是使用 C 语言编写（一些必要的部分是使用汇编语言）。Linux、Solaris、BSD 及其他现代操作系统也都是用 C 语言编写。本书将使用 ANSI C 实现，ANSI C 在 20 世纪 80 年代中期标准化，在《C 编程语言（第 2 版）》（Brian Kernighan & Dennis Ritchie）中进行了解释。

本章不是一个 C 语言综合教程，只是想给读者提供足够知识，使得后续章节需要建立 Python 扩展时，能够有效地使用 C 语言。如果读者碰巧熟悉 C 语言的原生版本（也称为“K&R”），可能会注意到有些地方有所变化，但是掌握这些新特性也不难。

安装 C 语言编程环境

gcc 是一个被广泛使用的流行编译器。不幸的是，它不支持 Window（至少不是直接支持）。在 Linux 或者 Solaris 环境上，gcc 无须安装就能直接使用。即使没有安装，通过安装包管理器也很容易搞定安装问题。如果使用的是 Windows 操作系统，可能需要有一些额外工作，或者购买一个商业 C 编译器来安装运行。[译注 1]

一种解决方案是使用微软的 Visual Studio，但是不推荐大家使用。这不是说 Visual Studio 不好，其实它很不错。只是因为 Visual Studio 太依赖微软的 GUI 范式模式，即使建立一个简单 C 语言程序，也要点几下鼠标按几次按钮。如果读者想开发 Windows 下 124

译注 1：微软 Visual C++ Express 版本可以免费使用，在 MSDN 上搜索可以找到下载地址。

的大项目倒无所谓，但是我们不需要涉及这么复杂，只是想建立一个简单的 Python 扩展。所以不选择 Visual Studio 开发环境。

尽管微软的 C 编译器和链接器可以在命令行下使用，但是还是复杂得不适合我们柔弱的心。更为简便的选择是安装 MinGW（Minimaist GNU for Windows）编译器。MinGW 不难安装，如果有 gcc 经验，建立 Python 扩展也很容易。详细的安装和配置 MinGW 文档可以在 http://boodebr.org/main/python/build-windows-extensions 找到。

使用 C 语言开发软件

C 语言是纯面向过程的编译型语言。所有功能由函式包装，C 程序由文件构成，文件中包含函式的集合（总体上比较类似 Python 的模块机制）。每个文件中可能包含了一些全局变量和预处理指令。在 C 源文件中全局变量可以标记为 static，这样可以保证其他源文件访问不了这些全局变量。

C 源代码会编译成对象文件，接下来对象文件与其他对象文件（可能是当前程序的一部分，也可能是系统库的对象文件）链接在一起产生最终的可执行文件。应该着重说一下“对象文件”、“库对象”以及“可执行对象”与面向对象（object-oriented）编程没有任何关系。这些术语是从冰箱一样的大型机时代就已经有的历史术语。

在过程式编程范式中，程序中的函式是主要焦点，而不是函式操作的数据，而在面向对象编程中，数据和方法（method）是等同的，数据也封装在对象中。图 4-1 展示了不同之处。

不像 Python 程序，C 程序直接编译成二进制格式（机器语言），由 CPU 来运行。C 程序不转换成中间的 bytecode 形式，也没有虚拟处理器解释 bytecode。C 语言因此被认为“接近底层”，一般用于关注性能和紧凑性的应用程序中。例如操作系统就是主要范例。C 语言和它的面向对象兄弟 C++ 主要用于创建库，库包含其他程序也能重用的函式。实际上 C 语言如此接近底层，有时候被开玩笑称为披着华丽外衣的汇编语言，而且大多数 C 编译器都可以选择参数来输出对应源代码的汇编代码。

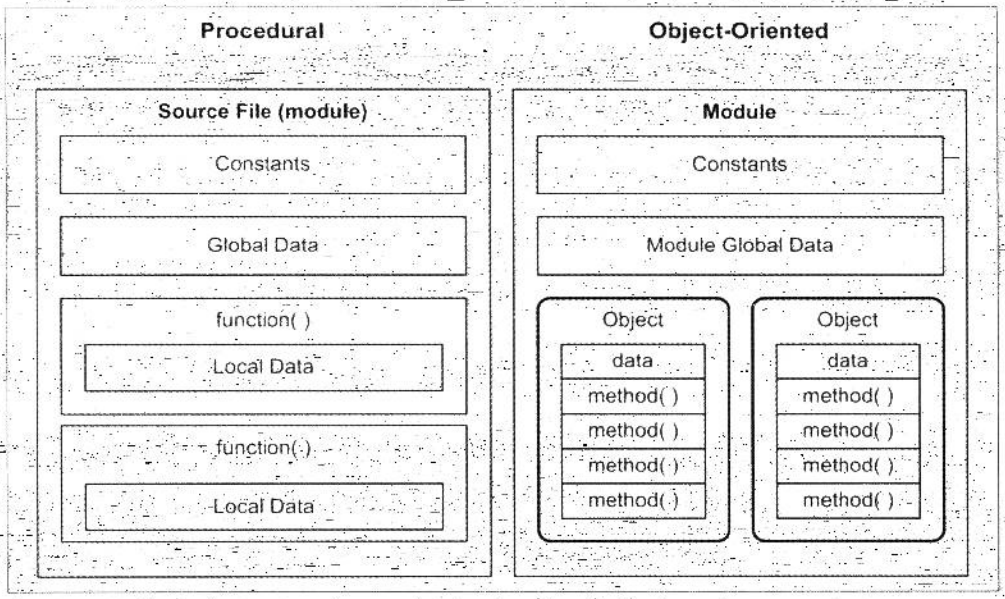


图4-1：过程式编程vs面向对象的函式组织

一个简单的C程序

大多数C语言教程都以“邪恶的”hello world作为开始示例。笔者倾向于使用更实际的例子，但仍然容易理解（希望如此）。这里选择一个简单的程序，生成正弦波形，用字符打印出来。如果你上过大学里的编程课，可能已经碰到过类似的问题了。

每个独立的C程序都有一个以main()命名的开始。一些C程序不是用作独立模块，而是作为其他模块使用的库函式（或者库函式集合）使用。在这些情况下就没有main()函式，而是由其他程序提供。我们看看下面的例子来快速浏览一个小型C程序：

```
/* sine_print.c
   Print a sideways sine wave pattern using ASCII characters

   Outputs an 80-byte array of ASCII characters (otherwise
   known as a string) 20 times, each time replacing elements
   in the array with an asterisk ('*') character to create a
   sideways plot of a sine function.
*/

#include <stdio.h>    /* for I/O functions          */
#include <math.h>     /* for the sine function     */
#include <string.h>   /* for the memset function   */
int main()
```

126

```
{
    /* local variable declarations */
    int i;                  /* loop index counter          */
    int offset;             /* offset into output string   */
    char sinstr[80];        /* data array                  */

    /* preload entire data array with spaces */
    memset(sinstr,0x20, 80);
    sinstr[79] = '\0';      /* append string terminator    */

    /* print 20 lines to cover one cycle */
    for(i = 0; i < 20; i++) {
        offset = 39 + (int)(39 * sin(M_PI * (float) i/10));

        sinstr[offset] = '*';
        printf("%s\n", sinstr);
        /* print done, clear the character */
        sinstr[offset] = ' ';
    }
}
```

如果读者已经安装了 C 编译器（如本章开始部分提到的），或者 C 编译器已经可以使用，就可以试着编译和运行这个程序。我们假定程序文件为 sine_print.c，在 Linux 系统下可以这样做：

```
% gcc sine_print.c -o sine_print
% chmod 775 sine_print
% ./sine_print
```

输出结果如图 4-2 所示。

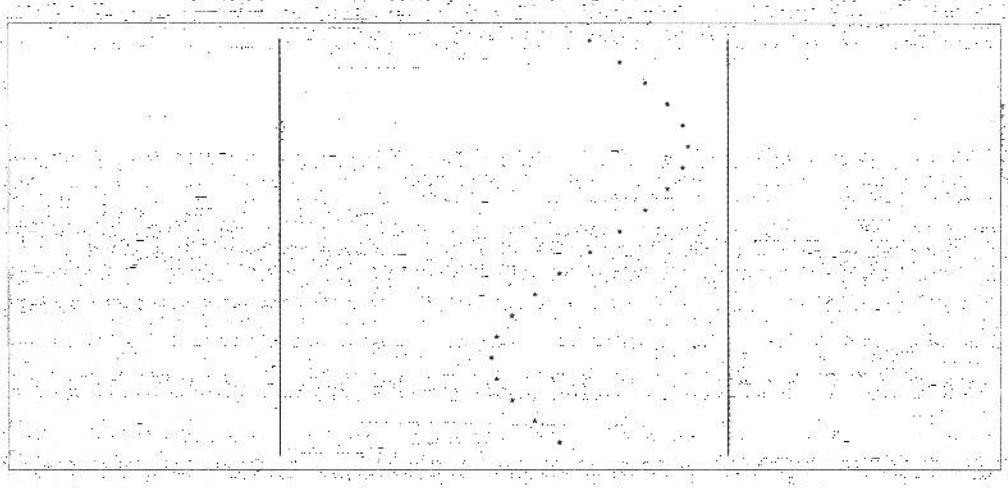


图4-2：sine_print 的输出

选择用这个例子而不是“hello world”作为开始，因为它展示了C语言几个关键特性。 127
我们将很快谈到这些特性，在后续章节也会更仔细的学习这些特性。

开始几行是程序注释。在C语言中注释以/*开始，用*/结束。一个注释块可能是多行的。

读者可能已经在ANSI C程序中看到过//形式的注释（我就看到过很多次），但这不是ANSI C标准的一部分[译注2]。//注释形式对于C++程序来说是合法的，大多数ANSI C/C++编译器都允许使用//。无论如何不推荐在纯粹C程序中使用//作为注释。

接下来我们能看到三个#include语句，它们指定了要包含在程序中的外部文件，也就是说其他文件会在编译器被调用之前读入，然后合并到现有代码中。其实#include语句不是C语言的一部分，而是预处理指令。预处理器可以识别一些保留字。实际上，我们可以认为预处理指令是一个小型编程语言。在源代码传递给编译器之前，预处理器也会从代码中移除所有注释。

然后示例中出现一个main()函式声明。在C语言中，一个函式的返回值是声明在函式名字前面，如果没有指定返回值那就是使用默认int类型。与Python一样，参数的使用是可选择的，所以可以看到一对空括号。

C语言使用大括号{}字符来标记代码块开始和结束。代码块可能包含多行语句，也可以没有内容。C语言的格式是自由的，只要语法正确，在名字之间有至少一个空格字符（或者标记）就行，编译器不关心有多少个多余空格字符存在。因此，我们可以建立一个绝对合法但对用户而言基本上不可解读的C程序。我们应该尽量避免写出这种代码。

sine_print程序声明了三个本地变量：i、offset和sinstr。变量i和offset是整型，sinstr是字符（bytes）数组。这些变量只有在sine_print存在的时候才存在，它们是有实际的内存位置，而不像是Python中的对象。

接下来程序调用了C标准库函式memset()，将sinstr所有元素设置为空格字符。注意这个函式有三个参数。第一个参数是目标内存地址，这里就是sinstr。第二个参数是我们想写入目标地址的字符值，这个字符值是ASCII空格字符（十六进制标记为0x20，十进制标记为32），这段内存是以数组sinstr第一个元素的内存地址为起始位置。最后一个参数指明sinstr有多少个元素（也就是说，sinstr的大小，在这个例子中是80）。注意，尽管sinstr是一个数组，我们不把它写成数组形式[译注3]。这是因为在C语言中数组与指针有着紧密的联系，我们后面会很快涉及这个话题。现在读者就可以假定使用没有索引形式 128

译注2：这不太准确，C99标准已经加入//形式的注释。
译注3：也就是说在memset调用时，不需要写sinstr[80]这种形式。

的数组名字，等同于数组索引值为 0 的元素内存地址。简单地说，memset() 会使用空格字符填充 80 个连续内存空间，这段内存空间的起始地址就是字符数组 sinstr 第 0 个元素的内存地址。

现在看接下来的赋值语句。在 C 语言中，一个字符串总是用所谓的“空字符”（string null）作为结束，也就是 8 位的 0 值。你可能还有印象，Python 不使用字符串结尾符号，因为 Python 字符串型对象在初始化的时候“知道”应该有多大。C 语言中字符串只是数组的另一种形式，如果没有结尾符号，对操作字符串的代码来说很可能就会“走过头”。

现在字符数组（也就是字符串）已经初始化，游戏可以开始了。for 循环修改 sinstr 字符串 20 次，把每次修改都打印到 stdout（标准输出）。在 for 循环中，第一条语句使用 C 语言数学计算库中的 sin() 函式来计算星号应该放在什么位置，把计算结果写到一个变量 offset 中。索引计数器 i 除以 10，来分解数学计算库中的 pi 值。在偏移值计算中，常数 39 决定了星号相对字符串的起始位置。

变量 offset 作为一个索引值确定 sinstr 中哪个位置需要将空格字符替换成星号。整个 sinstr 字符串包括最新插入的星号被打印出来。字符串中星号的位置被空格替换，然后整个过程再次重复。

最后，程序返回 0，结束整个程序运行。

预处理指令

C 语言提供预处理器支持，比如文件包含（#include）和命名字面值（#define）这两种最常用的功能。#define 预处理指令也被称为宏（macro）。预处理器也提供一些基本的条件测试和控制能力（#if,#elif,#ifdef,#ifndef,#else,#endif），以及“取消”宏定义（#undef）等等。注意预处理指令使用“#”符号开始，这与 Python 中的注释不是一个意思。

在一些 C 语言实现中，预处理器是一个单独程序，在编译过程一开始被调用，必要情况下也可以自己单独运行。预处理器扫描源代码，碰到预处理指令就开始工作，另外也移除代码注释。这也导致结果常常与原来代码有着非常大的不同。预处理器输出结果是一种编译器可以直接处理的格式：纯 C 语言标识（token），空格字符，再就没有其它内容了。
129 我们将要做的项目不需要什么高级的预处理器功能了。只要能包含文件到代码中以及定义一些常量，让编译器处理它们就行了。

#include

#include 指令如同名字所表示，用于把一个文件的内容引入到另一个文件中。比如一个常见例子是包含头文件的内容，从头文件这个名字可以看出，是被包含在源文件的开始部分。头文件一般包含比如其它文件或者库对象代码中的函式定义，以及这些函式相应

需要用到的宏定义。#include 指令也可以用于包含一个源代码文件到另一个源代码文件中，但是我们不太赞成使用这种用法。当我们讨论一个 C 程序结构的时候，会看看头文件是什么样以及如何使用它们。

在源代码文件中涉及到某些内容，那么 #include 指令必须出现在这些内容之前。把这些语句放在源文件开头是惯例（也是很好的主意）。在前面示例程序中（sine_print.c），有三个 #include 指令放在开头部分：

```
#include <stdio.h>      /* for I/O functions          */
#include <math.h>       /* for the sine function      */
#include <string.h>     /* for the memset function    */
```

这些语句告诉预处理器，要包含文件 stdio.h，math.h 以及 string.h 到程序中。这些文件都是 C 标准库的一部分，三个文件都包含 #define 语句，函式定义，以及更多的 #include 语句。为了能使用标准 I/O 函式，它们包含了所有需要的东西（printf()，memset() 和 pi 值）。另外基本数学计算中 sin() 函式也在标准库中。

#define

#define 宏指令将某个名字（一个字符串）与一个替换字符串关联起来。源代码中任何这个宏名出现的地方，都由替换字符串进行代替。#define 宏名能出现在任何可以直接敲入替换字符串的地方。考虑下面的例子：

```
#define UP 1
#define DOWN 0

if (avar < bvar) {
    return DOWN;
}
else {
    return UP;
}
```

经过预处理器处理以后，看上去是这个样子：

```
if (avar < bvar) {
    return 0;
}
else {
    return 1;
}
```

130

#define 宏通常用于定义常量，这个常量可能在单个或多个源模块中很多地方出现。这种方式比在代码中直接使用字面值，也就是所谓“裸数”（naked numbers）方式要好。主要原因是，假如程序在不同地方使用了同一个常量进行计算，在通用文件中使用宏定义

的方式修改起来要更简单，如果使用字面值，我们需要检查每个字面值出现的地方，决定是否替换；使用宏定义也减少了错误发生的几率。还有就是一些字面值是比较特殊的常量，如果没有注释，很难知道两个同样的值是否被用作不同目的。修改错误的字面值型常数可能会引发错误结果。

#define 也经常用于定义一个或者多个复杂语句，也可以接受变量。这也是“宏”（macro）这个名字的来源。这里有一个简单的例子：

```
#define MAX(a, b)      ((a) > (b)?(a) : (b))
```

三元条件运算符（?:）对 (a) > (b) 表达式进行短路径测试。如果这个表达式取值为 true，那么将返回 (a) 的值，否则，(b) 将作为结果返回。这是一种比较危险的宏，因为 a 可能以 ++a 的形式出现（我们将在后面讨论这种形式），这样 ++a 将计算两次，一次是在比较的时候，一次是在返回作为结果时。a 值将被增加 2，显然这不是我们想要的结果。

如果我们把示例中的两个 #define 语句放到一个单独的 updown.h 文件中，那这个文件就可以被需要这些定义的 C 源代码文件包含使用：

```
/* updown.h */

#define UP 1
#define DOWN 0
```

下面是一个使用的例子：

```
#include "updown.h"

if (avar < bvar) {
    return DOWN;
}
else {
    return UP;
}
```

当程序包含了 updown.h，结果与两个 #define 语句直接写在源代码中是一样的。

131 我们继续讨论之前，看看 sine_print.c 被 #define 宏定义替换字面值以后的样子（我把它命名为 sine_print2.c）：

```
/* sine_print2.c
   Print a sideways sine wave pattern using ASCII characters

   Outputs an 80-byte array of ASCII characters (otherwise
   known as a string) 20 times, each time replacing elements
   in the array with an asterisk ('*') character to create a
   sideways plot of a sine function.
```

```
   Incorporates #define macro for constants.
*/

#include <stdio.h> /* for I/O functions */
#include <math.h> /* for the sine function */
#include <string.h> /* for the memset function */

#define MAXLINES 20
#define MAXCHARS 80
#define MAXSTR (MAXCHARS-1)
#define MIDPNT ((MAXCHARS/2)-1)
#define SCALEDIV 10

int main()
{
   /* local variable declarations */
   int i; /* loop index counter */
   int offset; /* offset into output string */
   char sinstr[MAXCHARS]; /* data array */

   /* preload entire data array with spaces */
   memset(sinstr, 0x20, MAXCHARS);
   sinstr[MAXSTR] = '\0'; /* append string terminator */

   /* print MAXLINES lines to cover one cycle */
   for(i = 0; i < MAXLINES; i++) {
       offset = MIDPNT + (int)(MIDPNT * sin(M_PI * (float) i/SCALEDIV));

       sinstr[offset] = '*';
       printf("%s\n", sinstr);
       /* print done, clear the character */
       sinstr[offset] = ' ';
   }
}
```

这个版本把一些任务变得容易，比如将字符串一行字数从 80 变为 40 个字符。#define 定义了两个宏，一个 MAXSTR，一个 MIDPNT，它们都是使用了 MAXCHARS 这个宏进行计算，所以我们唯一需要做的就是把 MAXCHARS 改成 40。需要注意的是，当 MAXSTR 和 MIDPNT 进行代码中的宏替换以后，C 编译器从预处理器中得到的结果就像是：

```
sinstr[(80-1)] = '\0';
```

132

以及

```
sinstr[((80/2)-1) + (int)( ((80/2)-1) * sin(3.14159265358979323846 \
* (float) i/10))] = '*';
```

M_PI 宏定义在 math.h 头文件中，这里定义为

```
3.14159265358979323846
```

试着把 MAXCHARS 改成 40 或者 20，然后重新编译看看运行结果如何。

标准数据类型

C 语言只有四种基础数值型的数据类型。表 4-1 列出这四种类型。

表4-1：基本C数据类型

类型	描述
char	单一 byte 型（8bits）。如果没有指定无符号数，就是有符号类型的
int	一个整数，一般都是系统上的本地整数大小。一般都是 16 或者 32bit 大小。如果没有指定无符号的，就是有符号类型
float	单精度浮点值，一般表示为 32bits（4 个 bytes）。都是有符号的
double	双精度浮点值，一遍表示为 64bits（8 个 bytes）。都是有符号的

另外有四个修饰符可以与基本类型一起使用，指定存储数量大小以及期望的范围。它们是 short，long，signed，和 unsigned。

把它们放在一起就得到表 4-2 的类型定义。[译注 4]

表4-2：扩展C数据类型

类型	字节数	范围
unsigned char	1	0~255
signed char	1	-128~127
char（同 signed char）	1	-128~127
short（int）	2	-32 768~32 767
unsigned short（int）	2	0~65 535
unsigned int	4	0~4 294 967 295
signed int	4	-2 147 483 648~2 147 483 647
int（同 signed int）	4	-2 147 483 648~2 147 483 647
unsigned long	4	0~4 294 967 295
long（int）	4	-2 147 483 648~2 147 483 647
133 float	4	大约 +/-3.4E+/-38
double	8	大约 +/-1.798E+/-308
long double	12	注意，是 12 字节，不是 16

译注 4：char，signed char 并不一样，参考 http://en.wikipedia.org/wiki/C_variable_types_and_declarations

这些是典型的范围值。实际范围依赖机器架构和具体实现。具体细节参考具体的C编译器文档。

C语言中没有字符串类型；一个字符串就是以0作为结尾符的char类型数组。在C语言中，字符串与其它数组一样是可变的（或者借用Python的术语mutable，易变的）。

只单独使用short或者long，不与int类型名一起使用，这种例子也很常见。当我们指定了short或者long型的整数类型，int是隐含的，即使编译器能（至少说应该）接受int关键字，它也不是必需的。另外，如果没有指定，所有整数类型都是有符号的。

最后我们应该提到void关键字。这不是通常意义上的数据类型，而是作为占位符指明一个未定义类型。void通常用作指明函式没有返回值，或者作为占位符来说明一个指针指向任意有效内存地址（我们后面会提到指针）。

用户定义类型

C语言允许程序员定义一个标识符来代表一个已有数据类型或者已有类型组合起来的结构。例如：

```
typedef signed short int16_t;
typedef unsigned short uint16_t;
typedef float float32_t;
typedef double float64_t;
```

这些定义可以在/usr/include/sys/types.h（Linux、Mac、BSD操作系统）都可以找到。

如果有了这些定义就可以这样写：

```
int16_t var1, var2;
```

它和下面是一样的：

```
short var1, var2;
```

typedef也可以用于建立一个新类型定义，比如指针或者结构。

操作符

134

C语言提供算术、逻辑、比较等等操作符。也有一些特殊的一元操作符，比如前置加或者后置自加，前置减或者后置自减，逻辑非，一元算术正负操作符（算术非）等等。

一些操作符会在下面表格中出现多次，它们在不同环境下表现为不同操作。

算术操作符

C 语言中的算术操作符（见表 4-3）与其他编程语言中的算术操作符一样。

表4-3：算术操作符

操作符	描述
+	加法
-	减法
*	乘法
/	除法
%	取模

需要注意的是在不同类型的变量上进行算术操作。例如，当两个整数类型进行计算，数据类型大小中比较而言，小一点类型会被“提升”到与大一点的类型一样。比如说一个 short integer 与 long 做加法，short 类型变量会被“提升”为 long 类型。

一元操作符

C 语言中一元操作符对单个也只针对单一变量进行特定操作。一元操作符不能用于表达式，但是可以出现在表达式中。

表4-4：一元操作符

操作符	描述
+a	一元加法
-a	一元减法（算术负值）
++a	前置自增
a++	后继自增
--a	前置自减
a--	后继自减
!a	逻辑非（NOT）
135 ~a	位补码（取反）
*	指针提领（参考指针一节）
&	内存地址（参考指针一节）

这些操作符要解释一下。一元操作符中的正负操作符会修改变量符号。考虑下面这段代码：

```
#include <stdio.h>
void main() {
    int a, b;
```

```
    a = 5;
    printf("%d\n", a);
    b = -a;
    printf("%d\n", b);
}
```

当代码编译执行后，输出结果是这样的：

```
5
-5
```

自增和自减操作符会对变量值加一或者减一。自增和自减操作符的位置也很重要。如果 ++ 或 -- 操作符出现在变量之前，程序先对变量进行加减操作，然后变量被后续操作使用。如果 ++ 或 -- 出现在变量之后，后续操作先使用变量，然后变量再进行加减操作。下面代码对此进行展示：

```
#include <stdio.h>

void main()
{
    int a = 0;
    int b;

    b = ++a;
    printf("a: %d, b: %d\n", a, b);
    a = 0;
    b = a++;
    printf("a: %d, b: %d\n", a, b);
}
```

输出结果如下：

```
a: 1, b: 1
a: 1, b: 0
```

在第一个例子中，变量 a 值先增加，然后赋值给变量 b。第二个例子中对变量 b 的赋值发生在变量 a 自增操作之后（这是一个后置自增操作），所以变量 b 等于变量 a 自增之前的初值 0，而不是自增操作后的结果 1。

! 和 ~ 操作符像预期的一样。看看下面这个挺常见的例子：

```
a = !(b > c);
```

136

当变量 b 大于变量 c，结果将为假（false），尽管 b > c 这部分表达式为真（true）。! 是逻辑取反操作。二进制取反操作（~，1 的补码）会反转变量每一位，所以下面这个二进制值：

```
00100010
```

成为原来变量值的补码值，也就是：

```
11011101
```

我们将在后面指针部分讨论 * 和 & 操作符，现在先放在一边。

赋值与增量赋值

C 语言提供了一些有用的赋值操作符。请参考表 4-5。

表4-5：赋值操作符

操作符	描述
=	基本赋值
+=	相加并赋值
-=	相减并赋值
*=	相乘并赋值
/=	相除并赋值
%=	取模并赋值
<<=	位左移赋值
>>=	位右移赋值
&=	位与赋值
|=	位或赋值
^=	位异或赋值

C 语言中的赋值操作符拷贝值到一个内存位置。如果我们有两个变量，写成下面的语句：

```
x = y;
```

在变量 y 内存位置的值将被拷贝到变量 x 内存位置中。注意 x 和 y 在赋值操作发生之前就应该已经声明了，所以编译器可以定位这两个变量的内存位置。

137 像其它大多数现代编程语言一样，C 语言的增量操作符会对特定操作符两边的变量进行计算，然后把结果赋值给左手侧的变量，所以我们看看下面的表达式：

```
cnt += 10;
```

变量 cnt 的值与 10 相加，然后把结果赋值给 cnt。

比较操作符

C 语言的比较操作符和关系操作符(参考表 4-6)只能应用在数值上。它们不能用在字符串，结构以及数组上，尽管可能比较两个字符，比如 (“a” < “b “)。这个例子中结果将为真值，因为” a” 的 ASCII 数值要小于” b” 的 ASCII 数值。C 语言仅仅把它们看做 byte 值。

表4-6：C语言的比较操作符和关系操作符

操作符	描述
<	小于
<=	小于等于
>	大于
>=	大于等于
!=	不等于
==	等于

使用比较操作符的表达式一直返回逻辑真值 (true) 或者逻辑假值 (false)。在 C 语言中，逻辑假值被定义成 0，其他值都被认为是真值（尽管一般会使用 1 作为真值）。操作数可以是单一变量，也可以是其他表达式，这样可以允许复杂的复合表达式。

逻辑操作符

C 语言提供三个逻辑操作符，参考表 4-7。

表4-7：逻辑操作符

操作符	描述
!a	逻辑取反
&&	逻辑与
||	逻辑或

C 语言逻辑操作符对操作数的真值起作用。比如下面的语句：

138

```
tval = !x && (y < z);
```

真值表请参考表 4-8。

表4-8：tval真值表

tval	x	(y<z)
F	F	F
T	F	T
F	T	F
F	T	T

在这个例子中，只有当且仅当 x 为假以及表达式 (y < z) 为真的时候，tval 被赋值为真。

位操作符

C 语言长期作为系统级编程语言，留下不少历史遗产，比如说着丰富的位操作符。这些

位操作符在观念上认为“更接近硬件”，表 4-9 列出了 C 语言位操作符。相比 Python 这种高级编程语言，C 语言这种底层或者中级编程语言操纵硬件寄存器中的位（bit）更加容易。

表4-9：C语言的位操作符

操作符	描述
<<	位左移
>>	位右移
~a	位取反
&	位与
|	位或
^	位异或
<<=	位左移后赋值
>>=	位右移后赋值
&=	位与操作后赋值
|=	位或操作后赋值
^=	位异或后赋值

139 C 语言把增量操作符的概念延伸到位操作中。移位赋值操作符将左手侧操作数向左或者向右移动，移动位数由右手侧操作数指定，然后把结果再赋值给左手侧操作数。

如果有一个变量包含值 1，我们将其向左移动 2 位，结果将是 4，如图 4-3 所示。

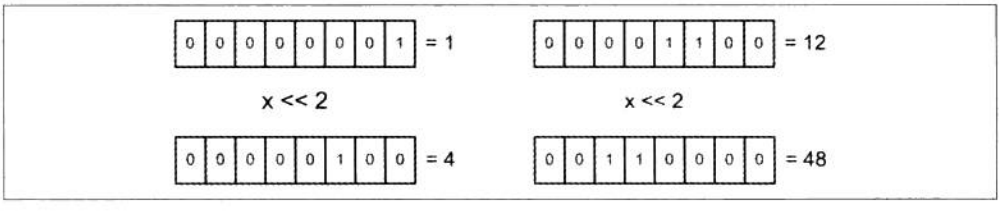


图4-3：位左移

在第一个例子中（左侧的），原始值 1 变成 4。在第二个例子中，原始值 12 向左移动 2 位变成 48。一个左移操作实际上就是原始变量值乘以 2 的阶乘，n 就是要移动的位数。相应的，右移操作相当于除法。没错，我们可以通过移位操作来加速算术，但仅限于操作 2 的阶乘。

AND、OR 以及 XOR 位操作符将操作符位与位一一对应。AND 和 OR 操作符大部分用于必须操作硬件寄存器单独某个位的情况，这种例子可以在接口电路中看到。AND 位操

作用于隔离整数中一个特定位，这个位可能是从硬件寄存器中读上来的。下面的代码显示了 AND 操作符是怎么使用的（我们假定变量都已经事先定义好了）：

```
regval = ReadReg(regaddr);
if (regval > 0) {
    if (regval & 0x08)
        SetDevice(devnum, SET_ON);
    else
        SetDevice(devnum, SET_OFF);

    regval &= 0xf7;
    WriteReg(regaddr, regval);
}
```

第一步先获取硬件寄存器当前内容，把数据写到变量 regval 中。然后 regval 检查是否有任何一bit 被设置（值将是否大于零）。如果是的话，那么位置在 2n3 会通过遮蔽其他位进行检查。使用 AND 操作符，如果 2n3 是 1，结果将不是零（实际上结果将为 0x08），那么会执行第二个 if 语句的真值部分。否则，第二个 if 的假值部分会被执行。最后，代码通过使用 0x08 的补码清除 2n3 位，并且把结果写回硬件寄存器。如果 regval 在 2n3 的位置上已经是 0，变量没有变化。这些代码会重置寄存器条件位或者终止外设硬件的一些行为。图 4-4 展示了 regval 位的操作。

140

操作符优先级

结束对 C 语言操作符概述前，我们看看操作符优先级这个话题。C 语言中每个操作符都有一个特定级别或者优先级，也就是表达式进行计算时的顺序。考虑下面两个表达式：

```
2 * 4 + 12
2 * (4 + 12)
```

它们看上去有些相似，但实际上有着不同的结果。第一个表达式结果为 20，第二个表达式结果为 32。原因是乘法操作比加法操作有更高的优先级。所以，如果没有告诉编译器括号的组合，乘法操作会先执行。

相关性也影响到表达式怎么计算的。在一般意义上，关联性定义了表达式中操作符相关操作数是如何计算的，以及同样计算顺序的操作符是以哪个方向进行计算。比如看我们看看下面的例子：

```
3 + 5 + 7    2 + 6
```

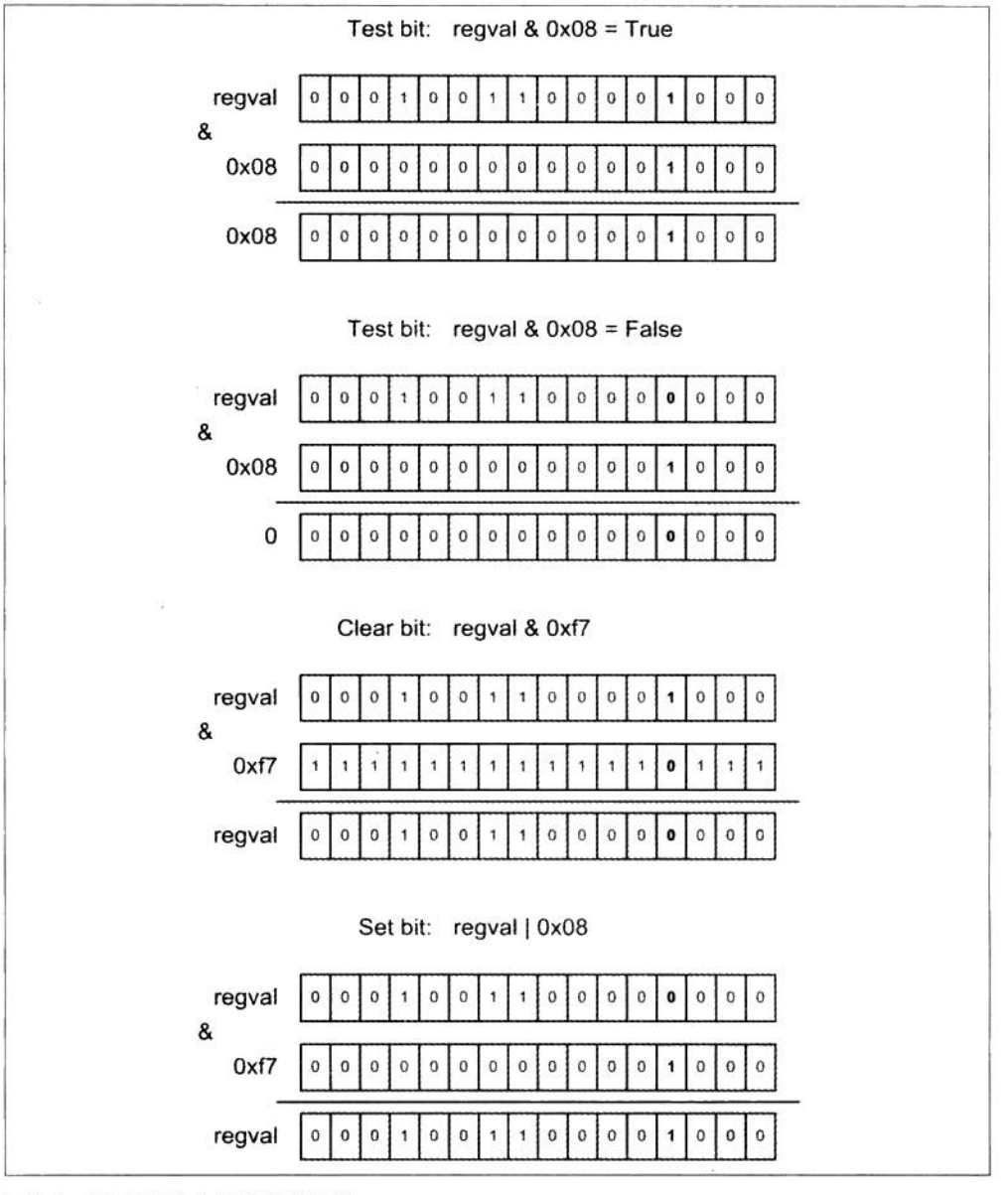


图4-4：使用C语言中的AND操作符

加法与减法的关联是从左到右。结果与下面表达式一样：

```
((((3 + 5) + 7)    2) + 6)
```

现在考虑下面的表达式：

```
3 + 2 * 6 + 8 / 4
```

乘法与出发的关联度也是从左到右，但是它们比加减法优先级更高，所以表达式被看成下面这样：

```
((3 + (2 * 6)) + (8 / 4))
```

如果我们期望的不是这样的，那就应该用括号来进行相应的分组。

一般来讲，使用括号来明晰编码的意图更好一些，而不是依赖语言优先级与关联规则。这将会更有可读性，会减少误解的可能。

表 4-10 按照优先级，以结合律从高到低顺序列出了 C 语言中的操作符。

142

表4-10：C操作符优先级

操作符	描述	结合关系
()	括号	从左到右
[]	方括号（数组标注）	
.	通过名字选择成员	
->	通过指针选择成员	
++ --	后置自增、自减	
++ --	前置自增、自减	从右到左
+ -	一元加、减	
! ~	逻辑取反、位补码	
(类型)	转型（修改类型）	
*	提领值	
&	取地址	
sizeof	判断大小（字节）	
* / %	乘、除、取模	从左到右
+ -	加法、减法	从左到右
<< >>	位左移、位右移	从左到右
< <=	小于、小于等于	从左到右
> >=	大于、大于等于	
== !=	等于、不等于	从左到右
&	位与	从右到左
^	位异或	从右到左
|	位或	从右到左
&&	逻辑与	从左到右
||	逻辑或	从左到右
?:	三元条件	从右到左
=	赋值	从右到左

续表

操作符	描述	结合关系
+= -=	加赋值、减赋值	
*= /=	乘赋值、除赋值	
%= &=	取模后赋值、位与赋值	
^= |=	位异或赋值、位或赋值	
<<= >>=	位左移赋值、位右移赋值	
,	逗号（分隔表达式）	从左到右

表达式

C 语言中表达式使用一个或多个操作符来定义一个计算动作。这可能是一个简单的比较，或者一个复杂多元布尔等式。当使用控制语句（我们后面会立刻提及）的时候，表达式一直以括号包围。对赋值语句中的表达式而言，括号是可选的，主要用于确定期望的操作被执行。括号包围的表达式有一个副作用（side effect），总是会返回一个值，这个返回值可能有用也可能没有用。

语句

C 程序由语句构成。语句与赋值语句、函式调用、控制语句或者它们的组合一样是可执行的。语句由记号 tokens（变量和函式名字）、表达式、或者是其他语句组成。C 是有着丰富控制逻辑的编程语言，允许程序员有很大的弹性来组织语句。

赋值语句会把赋值表达式右侧值拷贝到左侧记号中。左侧的记号必须是一个变量（可以是简单变量、也可以是数组中元素，或者结构体的某个成员），而左侧操作数可以是任意合法的变量名、常量、表达式或者可以返回值的语句。我们不能把值赋给函式或者常量，但是可以把函式返回值或者常量赋值给一个变量。

函式调用语句把程序运行控制转移到另一个函式。这个函式可能是与调用函式在同一个模块中，也可以是其它模块中。当外部函式结束返回，控制权返回给调用语句的下一个语句。一般能看到赋值语句与函式调用组合起来使用，就好像这样：

```
ret_val = ext_function();
```

C 程序控制语句控制程序的运行。有很多种控制语句可以操作分支、实现循环，或者直接转移控制权到程序另外部分。

语句的组合一般称为语句块（statement blocks），使用花括号 {} 包围起来。

if-else 语句

if 语句用于直接控制代码流程，或者运行的选路操作。测试表达式可以是 true（或任何非零值）或者 false（零值）。if 语句基本形式如下：

```
if (expression) {
    statement(s)
}
```

如果只有一个语句，花括号可以省略。

144

在简单的 if 语句，当 (expression) 这个表达式为 true，if 关联的语句会被执行。因为 C 语言不像 Python 严格要求代码格式，我们可以这样写一个简单的 if 语句：

```
if (expression) statement;
```

或者

```
if (expression) {statement; statement;}
```

在某些情况下，这种写法让代码更加容易读懂也比较简洁，但是大部分情况下应该少用这种不常用的形式。

else 语句可以处理当 (expression) 表达式为 false 的情况，也是很有用的：

```
if (expression1) {
    statements(s)
}
else {
    statements(s)
}
```

如果表达式 (expression) 为 true，程序执行第一块语句。如果表达式为 false，第二块语句被执行。

多个条件判断可以通过 else if 分组不同的条件来完成：

```
if (expression1) {
    /* block 1 */
    statements(s)
}
else if (expression2) {
    /* block 2 */
    statements(s)
}
else if (expression3) {
    /* block 3 */
    statements(s)
```

```
    }
    else {
        /* block 4 */
        statements(s)
    }
```

当表达式(expression1)为true，语句块block1将被运行。如果expression1不是true，(expression2)进行计算，如果它是true那么block2被执行。同样的规则应用到(expression3)，如果这些表达式都不是true，那么最终的else语句块block4被执行。注意最后一个else不是必需的。

145 switch语句

switch语句基于表达式的值来选择一个运行分支。表达式的值与常数进行比较，如果可以匹配，相关语句或者分支代码就会被执行：

```
switch (expression) {
    case constant:   statement;
    case constant:   statement;
    case constant:   statement;
    default:         statement;
}
```

常数可以是char、int、float、double类型，而且这些常数不能重复必须是唯一的。如果没有case语句匹配，那么可选的default相关语句会被执行。

如果case语句后面跟着多个语句，语句不需要用花括号括起来，尽管这在语法上是合法的：

```
switch (expression) {
    case constant:
        statement;
        statement;
    case constant: {
        statement;
        statement;
        statement;
        }
}
```

switch的一个陷阱或者说误解是穿越(fall-through)。考虑下面的示例代码，c_switch.c：

```
/* C switch example */
#include <stdio.h>

int main(void)
{
```

```
    int c;

    while ((c = getchar()) != EOF) {
        if (c == '.')
            break;

        switch(c) {
            case '0':
                printf("Numeral 0\n");
            case '1':
                printf("Numeral 1\n");
            case '2':
                printf("Numeral 2\n");
            case '3':
                printf("Numeral 3\n");
        }
    }
}
```


编译没有问题，当遇到一个句点符号（.）的时候 while 循环会通过 break 退出，但是输出结果像是下面这样：

```
0
Numeral 0
Numeral 1
Numeral 2
Numeral 3
1
Numeral 1
Numeral 2
Numeral 3
2
Numeral 2
Numeral 3
3
Numeral 3
.
```

当一个 case 语句匹配了变量 c 以后，不仅仅是它的代码被运行，但是运行“穿越”下去导致后面语句也被执行。这可能不是我们设想的（尽管有些情况下这种行为是程序员所期待的）。为了避免穿越的发生，可以使用 break 语句。这里是修改后加入 break 以后的 c_switch.c 代码：

```
/* C switch example */
#include <stdio.h>

int main(void)
{
```

```
    int c;

    while ((c = getchar()) != EOF) {
        if (c == '.')
            break;

        switch(c) {
            case '0':
                printf("Numeral 0\n"); break;
            case '1':
                printf("Numeral 1\n"); break;
            case '2':
                printf("Numeral 2\n"); break;
            case '3':
                printf("Numeral 3\n");
        }
    }
}
```

147 现在的输出结果就是我们期望的：

```
0
Numeral 0
1
Numeral 1
2
Numeral 2
3
Numeral 3
1
Numeral 1
3
Numeral 3
2
Numeral 2
.
```

应该在switch中一直使用break语句，除非有能说得通的理由才可以有例外。使用default也是一个好习惯。在某些编码规范中，比如在航天工业的程序编写，default语句是必需的，即使里面什么都不做。我们可以使用default配合switch来实现一个输入校验。设想一个函式需要特定的输入值，其它的情况都代表着错误应该被处理。这里有一个简单的代码片段：

```
switch(inval) {
    case 0:
    case 1:
    case 2:
    case 3:
```

```
            rc = OK;
            break;
        default:
            rc = BAD_VAL;
    }
```

在输入值在0~3，会设置变量rc为宏定义OK。如果是其它输入值，rc设置成为BAD_VAL。这里假设在后面代码中会检查rc的值来处理非法输入情况。即使我们可以用if和比较操作符写出等同的代码，好像这样：

```
if ((inval >= 0) && (inval <= 3)) {
    rc = OK;
else
    rc = BAD_VAL;
```

switch也处理不连续的值序列，不需要重新排序又长又复杂的逻辑和比较操作符。

while循环

当测试语句为true的时候，while循环会运行一个语句或者代码。

```
while (expression) {
    statement(s)
}
```

因为测试语句在循环体之前进行计算估值，有可能while代码块中的语句根本就不会执行。

do-while循环

do-while循环有些类似while，但是循环的测试表达式是在代码块最后部分进行计算：

```
do {
    statement(s)
} while (expression);
```

do-while循环体中的代码都至少运行一次，而在测试表达式为false的时候while循环中的代码根本就不执行。

for循环

for循环一般用在已经知道循环数量的情况，尽管它并不限制只能使用数值类型：

```
for (<initialization expression>, <test expression>, <iteration expression>) {
    statement(s)
}
```

for语句包含三个不同部分，每部分都是可选的。当省略了测试表达式（test expression）

的时候，逻辑值一直假定为 true。初始表达式（initialization expression）和迭代表达式（iteration expression）其中某一个或者两个都被省略的时候，就认为是没有操作。语法中的分号是必需的，用来标记一个或者多个被省略的表达式。

例如一个 for 循环省略了初始化和测试表达式：

```
int i = 0;

for (; ; i++) {
    if (i > 9)
        break;
}
```

想模拟一个无限循环，for 循环语句中的所有表达式都可以忽略：

```
int i = 0;

for (;;) {
    if (++i > 9)
        break;
}
```


这个例子中的无限循环仅意味着对于语句没有隐含的结束方式。如果内部测试一直不满足，那么循环将会一直运行下去。

C 语言 for 循环是非常灵活的，但一般都是用于可计数循环中（对比在 Python 中，for 语句设计成枚举对象某个范围的值）。初始化表达式建立循环计数器变量的初值，测试表达式控制循环的运行，迭代计数表达式在语句运行完以后执行。下面的代码展示了这个过程：

```
int i;

for (i = 0; i < 10; i++) {
    /* statements go here */
};
```

循环开始时将计数器变量 i 设置成 0。在循环开始时，先对测试表达式 i < 10 进行检查，然后运行循环体中的代码，接下来计数器表达式加一。再检查测试表达式，如果结果为 true，这个过程就会反复执行下去。

break 语句

break 语句用于结束或者中断一个循环（while，do，for）或 switch 语句块。当碰到 break 语句，会导致包含 break 语句最 近的代码块功能结束。循环中接下来的语句将不会执行。c_switch.c 示例演示了在循环中以及 switch 中使用 break。

continue 语句

continue 语句让代码运行控制立刻回到循环的顶层。它只能用在 while、do 和 for 循环语句中。continue 语句后面代码将不会被执行。

goto 语句

C 语言提供了一个有点危险有些邪恶也很容易被滥用和误用的关键字。你可能已经猜到了，就是 goto，可以让程序运行很直接的跳转到代码标记位置。像 Kernighan 和 Ritchie 所说的："严格说来，goto 是没用的。在实际工作中不用它也很容易写出代码。"大哉斯言。我们在本书中也不会用到 goto。

数组和指针

数组和指针在 C 语言里面是比较接近的概念；都指向内存中某块连续的存储区域。数组数组是一个类型和声名时指定的大小（即元素个数）组合起来的概念。指针也引用到某块指定了结构和大小的内存区域，尽管这和数组声明不太一样，我们在后面会谈到。因为这种紧密关系，可以用指针来引用一个数组的内容。相应的，数组索引可以认为是指向特定内存空间的指针。

在本章节，我们会讨论数组和指针以及它们的关联。特别会讨论指针是什么以及如何使用指针。

数组

数组是内存中一块连续的区域，存放有序的数据元素集合。当数组声明的时候与类型一起定义，数组中所有的数据元素都必须是这个类型。向我们已经看到的那样，C 语言中的字符串是 char 类型（byte）元素的数组。当然就像指针一样，可以有 int、float、double 类型的数组。

C 语言中数组的大小是固定的。一旦定义以后就不能改变了（实际上，如果用指针也不完全正确，我们后面会提到）。每个元素以及每个单元，在数组中都占据一定数量的内存空间来存放数组类型的变量。

这里有个比较丑陋的例子，把一个字符串数组的每个元素都打印出来，每个元素占据一行：

```
#include <stdio.h>

int i;
char cval = ' ';
```

```
int main(void)
{
    char str[16] = "A test string\0";

    for (i = 0; cval != '\0'; i++) {
        cval = str[i];
        if (cval != '\0') printf("%c\n", cval);
    }
}
```

要注意的是字符串 str 有一个 char 类型零作为结尾。如果没有它，for 循环会跑出数组进入到其它部分的内存中。也要注意到如果 cval 包含一个空字符，就不打印。

最后，有一个指针的数组，在特定机器（一般是 32 位）上使用无符号整型值来代表地址大小。

151 当数组通过索引值来存取，在数组中内存空间的实际位置要看数组的数据类型大小，就像图 4-5。

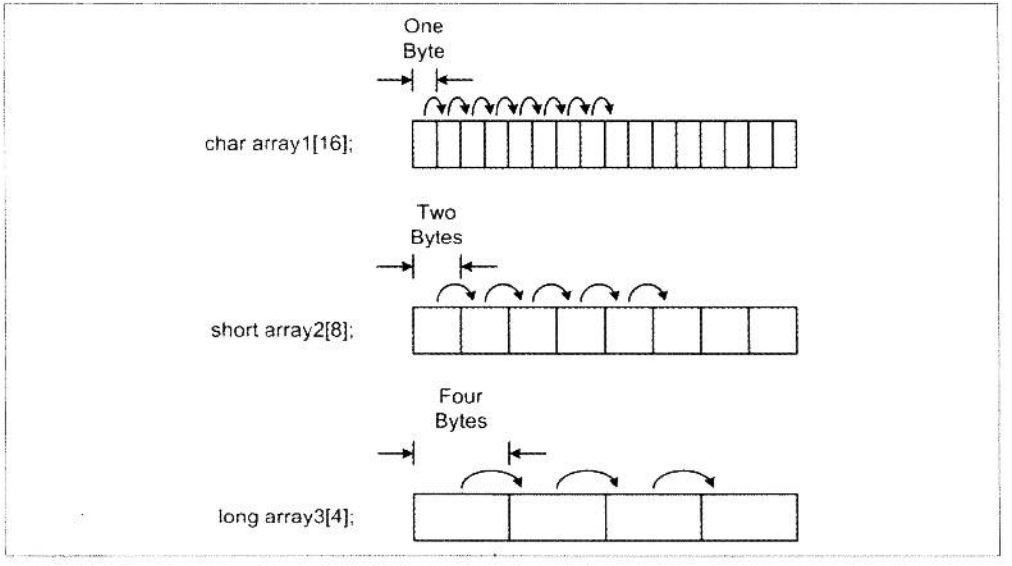
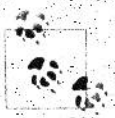


图4-5：数组索引

当 array1 的索引增加，内存位置每次增加一个字节（char）。当 array2 索引增加，内存位置每次增加两个字节。array3 则是每次增加 4 个字节。注意 C 语言中索引是从零起始。

指针

指针是一个包含了内存地址的变量。这个地址可能是另一个变量，也可能指向一块连续内存区域。也可能是I/O设备中某个控制寄存器的地址等等。指针是一个非常强大的概念，它使得某些问题更容易处理。因此C语言也有时被称为“指针的语言”。理解指针是什么以及如何有效使用，是写出简洁而强大的C程序的关键。常常能听到关于C语言指针的批评，没错，乱用指针会导致比较严重的问题，我们应该小心遵循某些规律来使用指针，这样就能像其它（没有指针概念的）编程语言一样写出安全可靠的代码。



有一个实际的例子C指针能用在太空飞船上管理图像数据处理的内存，参考Beautiful Data的第二章，Jeff Hammerbacher和Toby Segaran编辑。 152

想定义一个指针，可以用*操作符：

```
int *p;
```

这里定义了一个名字为p类型为int的指针，可以指向整型数据位置。想给指针赋值一个地址可以用&一元操作符：

```
int x = 5;
int y;
int *p;

p = &x;
```

指针变量现在包含了变量x的地址，我们可以用指针p来获取x的内容：

```
y = *p;
```

这会把x值赋给变量y。当*操作符这样使用的时候，我们称之为提领（dereferencing）或者间接指向（indirection）。

指针可以用做函式参数传递。这可以让C函式操作它范围外的数据，或者返回多个值。下面是一个通用的交换函式：

```
void swap (int *x, int *y)
{
    int tmp;
    tmp = *y;
    *y = *x;
    *x = tmp;
}
```

参数x和y由调用函式提供，像这样：

```
int a = 10
int b = 20;

swap(&a, &b);
```

注意当 swap() 调用，& 地址操作符用来传递两个变量 a 和 b 的地址。当 swap() 返回，变量 a 值为 20，变量 b 的值为 10.

前面提到数组和指针有关联，这里可以更好的看看两者关系。在 C 语言中数组第一个元素就是数组内存位置的基地址。我们可以把数组的基地址用 & 操作符赋值给指针，就像这样：

153

```
int anarray[10];
int *p;
p = &anarray[0];
```

因为 anarray 和 p 都定义为 int 类型，这两条语句是等同的：

```
anarray[5];
*(p+5);
```

这两个都存储着 anarray 内存中索引位置为 5 的值。我们用指针的形式，是说内存地址位置增加 5 个元素，而不是说指针指向的值增加 5。可以用这样的代码来把数组的基地址赋值给指针：

```
p = anarray
```

这种格式的隐含地址是 &anarray[0]，也是把数组基地址赋值给指针的常用形式。&anarray[0] 的形式不是必要的；如果不是把指针指向数组索引值不是 0 的元素，都可以直接使用数组名字这种形式。

指针可以用来指向其它比如简单变量或者数组。指针可以指向结构（在下一章会谈到），或者指向函式。如果以前没有学过用过，掌握 C 语言中的指针需要一点时间，但是学会指针概念是比较值得的，因为指针可以用比较紧凑有效的表达方式来完成一些比较困难甚至是不可能的任务。

结构

C 语言中的结构是不同类型变量的集合。一个结构有唯一的名字或者结构可以包含结构。

图 4-6 用图形表明了结构的语法。

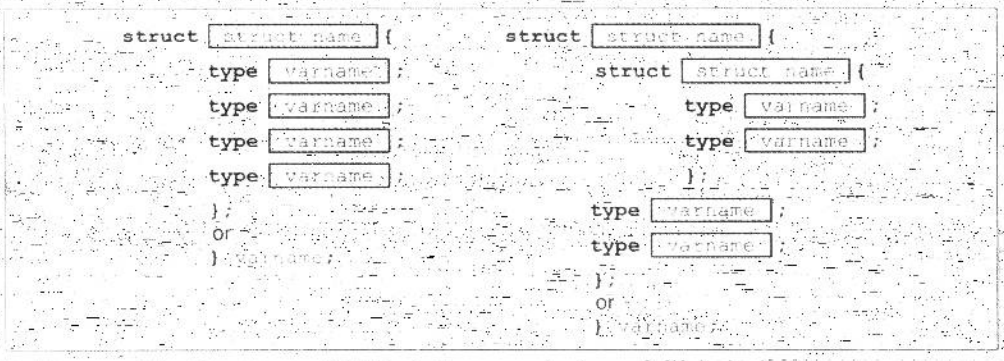


图4-6：C结构语法

我们举一个简单的例子，这是个简单的包含三个变量来保存一个简单输入测量值：

154

```
struct measdata {
    float meas_vpp;
    float meas_f;
    long curr_time;
};
```

这没有什么特定含义，我们可以这样来声明一些测量值：

```
struct measdata measurements[100];
```

注意：这个结构定义没有申请存储空间——只是定义而已。换句话说，这是一种新的由程序员定义的类型。我们可以通过加入一个或者多个变量名字来跳过声明结构类型的额外步骤：

```
struct measdata {
    float meas_vpp;
    float meas_f;
    long curr_time;
} measurements[100], single_meas;
```

可以用“点号语法”来使用一个结构成员。如果我们想使用第 47 个 measurement 的 meas_vpp 值，可以这样：

```
vpp_value = measurements[46].meas_vpp;
```

有一个更实际一点的例子，从输入来源获得 100 个 measurement：

```
#define MAXMEAS 100

struct measdata measurements[MAXMEAS];
int stop_meas = 0;
```

```
int i = 0;

while (!stop_meas) {
    measurements[i].meas_vpp = GetMeas(VPP);/* VPP defined elsewhere */
    measurements[i].meas_f = GetMeas(FREQ); /* FREQ defined elsewhere */
    measurements[i].meas_time = (long) time(); /* use standard library function */
    i++;
    if (i >= MAXMEAS) {
        stop_meas = 1;
    }

    /* maybe put some code here to check for an external stop condition */
}
```

我们可以用 typedef 关键字来建立一个新的结构类型：

```
typedef struct dpoint {
    int unit_id;
    int channel;
    float input_v;
} datapoint;
```


使用 typedef 的时候，结构名字是可选的。下面形式也可以：

```
typedef struct {
    int unit_id;
    int channel;
    float input_v;
} datapoint;
```

新类型可以这样使用来建立新变量：

```
datapoint input_data;
```

在继续学习之前，先看看指向结构的指针和如何通过指针获得结构内容。下面例子介绍了如何建立指向结构的指针数组，然后获取指向结构数据的内存，并引用结构内容：

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int unit_id;
    int channel;
    float input_v;
} datapoint;

int i;

datapoint *dpoint[10];
```

```
int main(void)
{
    for (i = 0; i < 10; i++) {
        dpoint[i] = (datapoint *) malloc(sizeof(datapoint));
        dpoint[i]->unit_id = i;
        dpoint[i]->channel = i + 1;
        dpoint[i]->input_v = i + 4.5;
    }

    for (i = 0; i < 10; i++) {
        printf("%d, %d: %f\n", dpoint[i]->unit_id,
                               dpoint[i]->channel,
                               dpoint[i]->input_v);
    }

    for (i = 9; i > 0; i--) {
        free(dpoint[i]);
    }
}
```

当我们运行代码，会生成下面的输出结果：

156

```
0, 1: 4.500000
1, 2: 5.500000
2, 3: 6.500000
3, 4: 7.500000
4, 5: 8.500000
5, 6: 9.500000
6, 7: 10.500000
7, 8: 11.500000
8, 9: 12.500000
9, 10: 13.500000
```

正好可以看看这里的一些新概念。首先建立了指针数组类型 datapoint 的变量 dpoint。当指针或者指针数组以这种方式建立的时候，指针值是非法的，因为内存地址还没赋值，它们可能指向任意可能不合法的内存。

在 main() 函式中，第一个 for 循环通过 malloc() 标准库函式，赋值 datapoint 大小的内存块给数组中每个元素。malloc() 函式从操作系统中申请一块大小指定的内存（在这个例子中，需要可容纳 datapoint 结构的实例）。第一个 for 循环也同时根据 datapoint 结构把申请的内存块初始化。

因为 dpoint 数据类型是 datapoint，malloc 提供的内存块将被视为 datapoint 的结构。请参考图 407`。

想访问每个结构中单独元素，可以使用箭头标记（arrow notation）。C 语言使用箭头标记

把指针关联到结构类型定义的元素。

第一个 for 循环读入结构的数据并且打印出来，最后一个 for 从数组从后向前释放每个元素的内存。

函式

在 C 语言函式中，所有可执行语句包含在函式中（预处理语句不在代码编译后的可执行部分里面）。C 语言不像 Python 模块可以直接执行。另外 C 语言中每个程序都必须有个 main()。这是程序的开始入口点。应用模块、支持模块或者库模块都将会编译链接到其它模块中，就不需要 main() 函式。

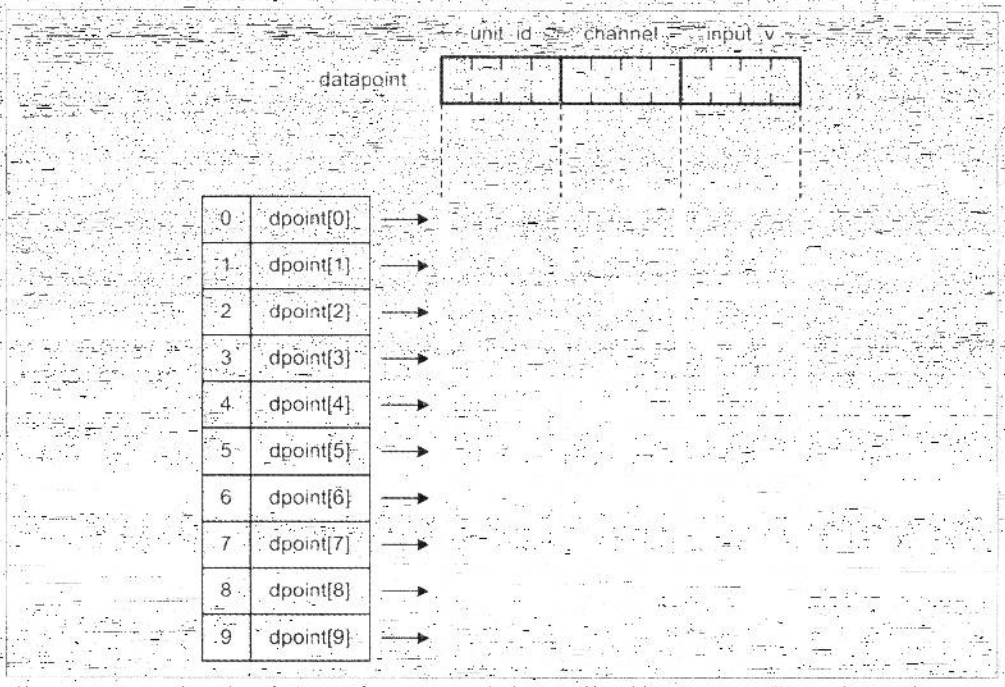


图4-7：结构映射到申请的内存

函式语法

C 函式基本语法像是这样：

```
[type] name (parameters)
{
```

```
    statements...
}
```

可选的 [type] 限定词指定函式返回值类型，如果没有，编译器一般会假定函式返回 integer 类型（某些编译器比如 gcc 可能产生类似缺少返回值的警告信息）。函式名由字母和数字字符构成，在实际编程中函式名是很大的风格问题。函式后面跟着括号括起来的零个或者多个参数。参数也有类型限定符，可能是值也可能是指针。在下一行，有一个左括号 {，跟着零到多个语句，最后结尾是一个右括号 }。

标准 C 语言函式是不能嵌套的，即使一些编译器允许函式嵌套作为非标准扩展（注意：不要这样做，否则你会后悔的）。每个函式都在它内部构成单独唯一的实体单元。

158

函式原型

函式原型（function prototype）定义了一个函式，可以帮助编译器解决提前引用或者外部引用问题。因为 C 语言需要变量、typedef 和函式在使用之前就定义好，把函式原型放在模块开头或者外部文件是一个很方便的做法。这让模块中的函式可以任何顺序出现。另外一种方式可以把函式定义放在调用之前，这对一些小的模块是可以接受的，但是对于包含很多函式的大模块可能是个问题。

函式原型就是一个分号结尾的函式定义语句。一个函式原型定义引用到当前源文件还没定义的函式，或者引用到另外独立对象单元中的函式。原型主要用来建立编译代码的占位符，然后链接器（linker）会填充。比如：

```
void bitstring(char *str, long dval);
```

函式原型指定函式返回类型（在这里是 void）、名字、参数个数以及类型。参数名字是可选的，这意味着下面的代码也可以工作的很好：

```
void bitstring(char *, long);
```

标准库

ANSI 兼容 C 编译器都带着一些头文件和库模块，提供诸如数学操作、字符串操作、内存管理、输入输出操作以及其它功能。表 4-11 列出简略的标准库列表。更多信息请参考所使用编译器相关文档。

表4-11：ANSI C标准库模块

文件名	描述
assert.h	定义 assert() 宏。用来帮助在程序的调试版中测试和检查错误。如果调试选项没有激活，assert() 宏就无效
ctype.h	声明用来测试区分字符的函式，比如 isalpha()、isdigit() 等等
errno.h	捕捉标准库其它函式产生的错误码
float.h	定义用来进行浮点数值计算的宏
limits.h	定义特定实现的常量宏，比如一个 byte 中位数，以及整数类型最大最小值等等
locale.h	定义比如各国货币表现形式以及浮点数格式等等。声明 setlocal() 函式
159 math.h	定义了算术函式比如 sin()、log()、pow() 以及 floor()。宏定义比如 M_PI、M_TWOPI、M_SQRT2 和 M_LOG2E 等等
setjmp.h	定义了宏 setjmp() 和函式 longjmp()，用来做非本地跳转
signal.h	提供了产生和处理系统级信号的方法，比如 SIGTERM 和 SIGSEGV
stdarg.h	定义了 va_start()、va_arg() 和 va_end() 宏来访问传入函式的可变参数
stddef.h	定义了一些在其他标准库头文件中会用到的类型和宏
stdio.h	提供输入和输出功能，包括 printf()，输出格式化，输入格式化，还有文件 I/O 操作
stdlib.h	包含了对各种工具函式的声明，包括 atoi()，rand()，malloc() 和 abs()。这里声明的函式和宏支持各种操作，包括转换、随机数、内存分配、进程控制、环境变量和排序等
string.h	定义了字串操作函式，比如 strcpy()，strcat() 和 strcmp()，以及内存操作函式，比如 memcpy()，memset()
time.h	提供了一些用于输入和转换当前日期及时间的函式。这里声明的函式的部分行为是由区域设置中的 LC_TIME 部分定义的

最常用的标准库模块是 math.h、stdio.h、stdlib.h 和 string.h。我建议读者看看这些模块，会有个大概认识。

编译 C 程序

典型的 C 程序常常是一些文件的集合。其中一些包含与程序有直接关系的源代码，而其它可能包含函式原型或者宏定义。也有一些文件直接包含在通用的库中。当程序编译的时候，所有这些都传递到编译器，转换成对象文件，然后链接成完整的可执行文件。图 4-8 表明了编译 C 源代码到可执行文件的步骤。

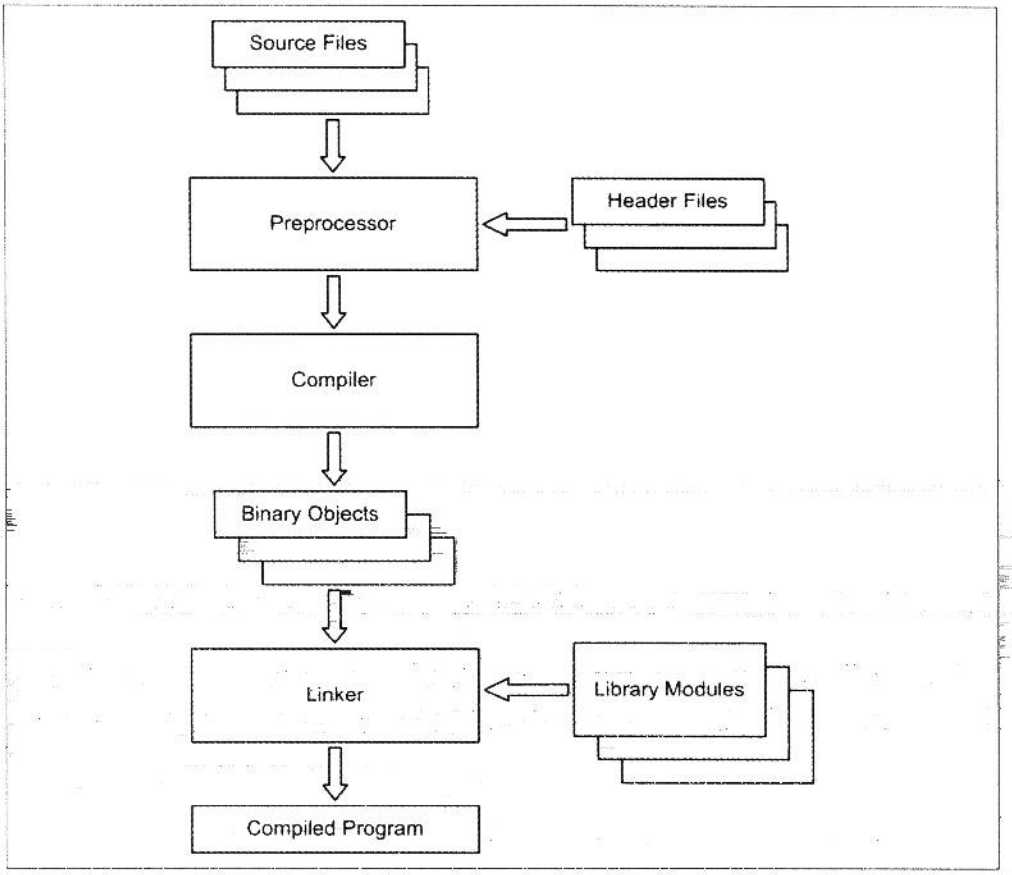


图4-8：C程序编译

头文件

在C程序中，一般有两种文件扩展名：.c和.h。.h文件扩展用于表示头文件，这么说是因为典型做法会把所有必要的.h文件放在代码开始部分，在所有实际代码之前。

头文件可以通过#include指令包含。它们一般包含函式原型、宏定义以及typedef语句，头文件也可以一常常也会发生一包含其它头文件。无论如何，把可执行函式代码放在头文件中被认为是比较差的形式和比较差的实践。即使在C语言中这对于编译执行安全可连接对象来说是件小事，也没有理由这么做。

目标文件

C编译器自己不产生可执行文件。它产生的是对象文件，也就是过去大型主机系统时期

160 的一个术语，指代由机器指令组成的二进制文件。实际上，一个对象文件是一个中间产物。为了执行，它仍然需要一些额外的代码，这由链接器来提供。

如果一个对象文件引用了外面其它的二进制物件，编译器插入一个占位符，建立一个这些物件文件的列表。链接器为了解决外部引用，接着读入这些数据。一些 C 编译器（最主要是微软 C 编译器）会自动混合一个运行期物件来访问 Windows 系统功能。

库文件

库文件是可以与其它程序交互的二进制对象（object）的集合。二进制库文件有特定的
161 格式，但实际上不是统一的格式。图 4-9 表示了一种常见库文件内部格式的例子。

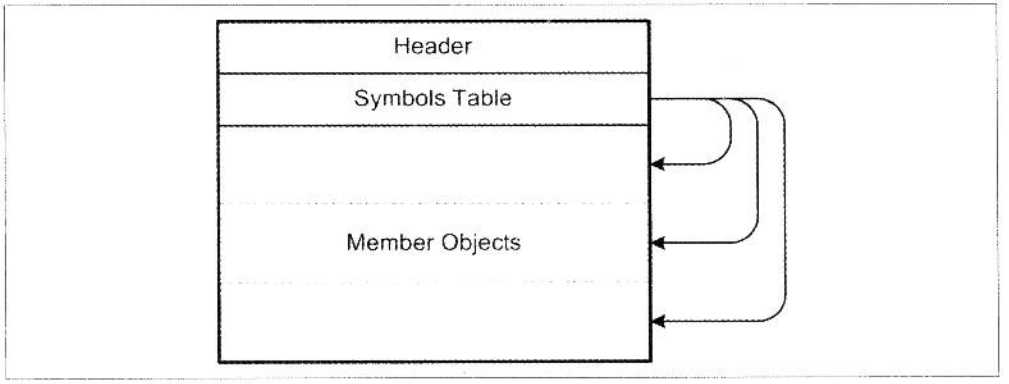


图4-9：常见库文件内部格式

库文件头部标识了类型，可以是静态的或者动态的。符号表包含了对象名字及指向每个对象的地址偏移值的列表。最后是物件对象本身。

链接（Linking）

链接器是一个独立的程序，可以把程序二进制对象与外部库模块链接到一起。它使用程序二进制对象中的外部引用占位符决定哪些外部对象需要被链接以及链接的地址。图 4-10 标识了链接过程怎么工作的。

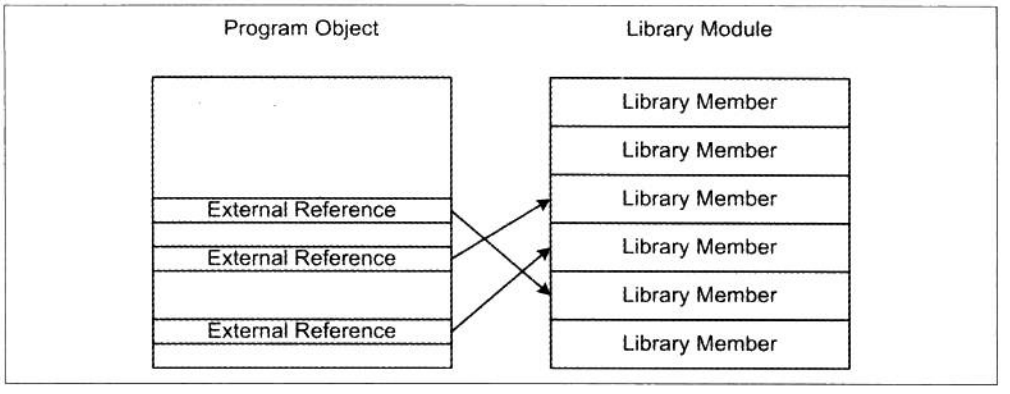


图4-10：程序二进制对象链接期

链接器一般是相对编程语言独立的。比如，我们可以将C生成的对象文件与FORTRAN产生的二进制对象链接在一起，只要编译器产生的两个二进制对象是链接器期望的同一种格式就可以。图4-11展示了一种常见的例子，一个程序二进制文件以及两个库模块、一个运行期对象链接到一起，产生特定操作系统上的可执行文件。不是所有的C语言实现都使用或者需要一个运行期对象模块，在某些实现中类似printf()这样的功能支持是内置的，即使程序中不需要这些功能。

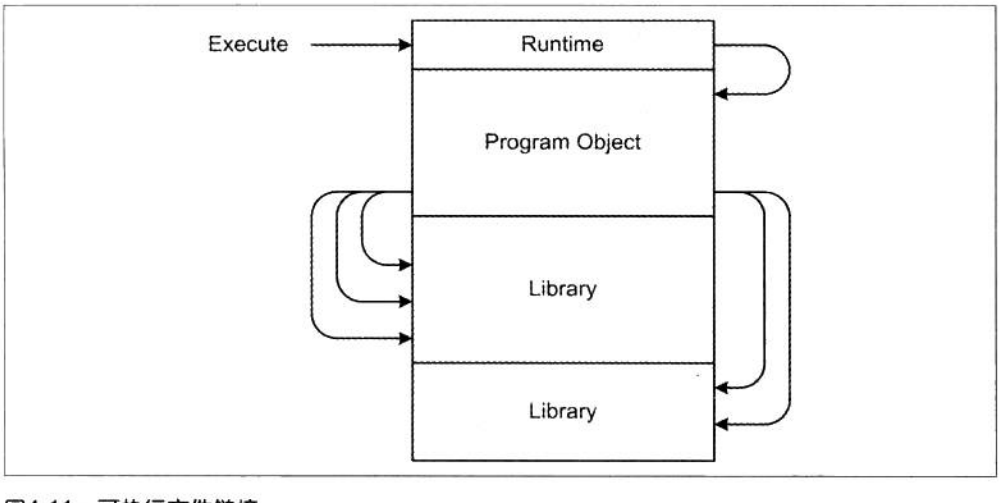


图4-11：可执行文件链接

程序一直从main()函式开始执行，这是程序载入时处理器默认指向的运行代码位置。

make

最后我们应该提一下 make。make 不是 C 编译器的一部分；它是一个单独的程序，尽管也包含在 C 编译器发布包中。make 是一个基于规则和动作的工具，用来维护一套代码文件。规则指定了如果文件缺失或者被修改以后需要做什么动作。当用来管理一个有着很多源代码和头文件的大型 C 程序，make 可以在需要时调用编译器或者链接器，来保证所有的二进制对象、库文件以及可执行文件都是最新的。几乎所有版本的 make 都有一套自己唯一的“小型语言”允许宏定义、名字替换、普通以及特定规则格式，以及能使用操作系统中其它应用的能力。我们这本书中不深入讨论 make，但是鼓励读者在线阅读一下 make 的文档或者看看操作系统附带编译器中的 make 文档。

C 语言综述

C 语言最初是一种用于系统级编程的过程式语言，后来演变成了一门通用语言，占据了一个在汇编语言之上，在高级语言比如 Python、Java、Lisp 之下的适当位置。因为 C 语言可以进行底层编程的特质，它很适合用于那些要与系统硬件和底层操作系统直接交互的任务。然而，这也是一把双刃剑，导致很容易写出一些 C 代码在主机系统上干一些非常令人不悦的事。要写出稳定、安全和可靠的 C 代码，一定要细心并遵循编码规范。

C 开发工具

如同 Python 一样，编程需要一个文本编辑器或者 IDE 来输入和编辑 C 源代码。

如同 Python 处理类似维护大量包和模块在运行时动态地更新到最新和提供类似内省作为内置能力，C 程序必须编译和链接在开发过程中必须作为独立步骤。

另外，C 语言也没有支持动态检视一个运行程序的能力，也不能输出什么模块被载入了（除非程序员自己把这些写在代码里了）。C 程序最终产品是一个二进制可执行文件对象，所以调试需要一个特定工具，能从二进制对象中读取各种头部信息和符号表，能在源代码中找到匹配。如果想让编译代码更具有可读性，必须在编译过程中包含调试信息。这产生一个更大的文件因为包含了各种符号表和其他调试器需要的数据。

可以参考第三章，有一个可用的编辑器列表。作为 IDE，Eclipse 有插件支持 C 编程，也支持 Python 开发。当然，微软 Visual Studio 支持 C 和 C++ 开发。

Linux 和 UNIX 类操作系统下，经典的 GNU 调试器 gdb 包含在大部分 gcc 发布包里面。如果你的系统有 gcc，gdb 很大可能也已经安装了。在 Windows 平台如果 Cygwin 或者 MinGW 之一安装了，gdb 也就安装好了。微软 Visual Studio 有一个非常棒的带有图形界面的符号化调试器。在 Linux 平台下，DDD 是一个很棒的为 gdb 提供图形界面包装的工

具。它也支持 Unix 系统上的其它 C 调试器。

小结

164

C 是一个令人着迷的编程语言，在底层上它去除了几乎所有对程序员的限制，把计算机系统内核整个开放给程序员进行检视和操作。同时，它也很适合作为一个中级过程语言。虽然仅仅接触到这门语言的一部分能力，但我们还是尽力覆盖了很多内容。利用本章学到的知识，读者将有能力编写自己的 Python 语言扩展来为其增加新功能及面向真实世界的新接口。

推荐阅读

C 语言从诞生到现在差不多有 40 年的历史了，成堆的书籍、论文、文章和报告加起来可称得上是不计其数。下面列出的只是其中冰山一角：

C Programming Languae, 2nd. Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1988.

由贝尔实验室这个发明 C 语言的地方的人编写和审阅完成，被认为是 C 语言最早的定义参考。已经更新过一版，现在的版本兼容 ANSI 标准。简洁扼要，容易使用，这本小书对一代一代的 C 程序员和软件工程师的帮助是抹不掉的。每个想使用 C 语言的开发者必须有一本。

The Standard C Library. P. J. Plauger, Prentice Hall, 1991.

一个详细便利的手册，涵盖了标准 C 库组件。这是一本参考书，值得买一本放在书架上。

Practical C Programming, 3rd ed. Steve Oualline, O'Reilly Media, 1997.:

一本直接深入，特别是在软件工程最佳实践方面很突出的书籍。示例展示了一些 C 语言编程中常见的陷阱，也有大量的为什么以及如何使用 C 编写程序的例子。

像大家期待的一样，互联网上也有大量的 C 编程资源。在 Google 搜索引擎输入“C programming language”就会返回巨量结果。这里是一些值得提及的：

http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

Dennis Ritchie 写的一个有趣的关于 C 语言的早期历史和开发的文字，概括了一些 1970 年左右加到 C 语言的想法。

http://cprogramminglanguage.net

165

包含了大量简单但是很有用的介绍材料，按照一本书的章节一样组织的很好。

http://www.gnu.org/s/libc/manual/html_node/index.html

GNU gcc 编译器套装提供的库文件的在线文档。同时还有一个很有用的“概念索引”（concept index），按照特定话题对相关主题进行了分组。

http://en.wikipedia.org/wiki/C_(programming_language)

在维基百科上关于 C 语言有个详细的条目，其中有很多指向其他信息源的链接。

http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

UIUC（伊利诺伊大学厄巴纳—香槟分校，也是 NCSA[译注5]的总部）ACM 学生分会有一个 Eric Huss 编写的在线 C 库参考指南，该文档有一个清晰的详细索引指向各个页面。

http://citeseerx.ist.psu.edu

一个极其有用的网站，有大量的研究论文和技术出版物。尽管这里面几乎没有什么教程级别的文章，但还是能找到大量有用的，引人入胜的材料。

译注 5：国家超级电脑应用中心（National Center for Supercomputing Applications，NCSA），世界上第一个图像化的网页浏览器 Mosaic 即诞生在这里。

第5章 167

Python扩展

简洁达到极致，是为优雅。

——Jon Franklin

这一章的主要目的是展示如何利用已有的二进制模块来扩展 Python。Python 提供了两种方式，我们来看看它们。

首先，Python 不能直接访问底层硬件，也不能与大多数硬件供应商提供的软件模块直接交互。Python 可以访问通过串口连接到电脑，或者是把 USB 作为虚拟串口（很多 USB 到 RS485 转换口使用这种技术）的硬件。对于这些应用，不需要使用扩展模块。

使用 C 语言建立扩展，可以让 Python 访问实际硬件，或者加入某些更快或者特定的功能。一般是使用动态链接库（DLL）来让 Python 获取内置库以外的能力。我们将主要用 Python 扩展来包装商业硬件和接口提供的 dll 文件，但也可以用来优化 Python 程序中的特定部分来提升处理速度。

另一个方法就是使用 Python 的 ctypes 库，可以直接访问外部 dll 提供的函数。ctypes 库可以直接在 Python 代码中使用，不需要写任何 C 语言代码，支持 Linux 和 Windows 平台。缺点是在你的程序和外部库之间需要额外的功能，这需要用 Python 来写，不像 C 语言作为包装器那种方式那么快。ctypes 在速度上问题，不怎么快。但如果你的程序不需要考虑运行速度，这可能是最容易的方法。

使用哪种方式取决于你想使用接口完成什么功能。如果外部 dll 已经实现你需要的所有 168
功能，就没有必要来使用 C 语言写你自己的扩展包装器。另一方面，如果你想加入一些额外的功能，比如错误检查，缓存处理或者基于外部 dll 的定制能力，应该考虑用 C 语言写一个包装器。

168 尽管这一章我使用 DLL 来指代 Windows 风格的 DLL 文件或者 Linux 下的 `.so`（shared object）动态库。Python 共享库对象一般有 `.pyd` 后缀。另外要注意，大部分关于供应商提供的用于数据获取和硬件控制的 DLL 模块默认情况下都指的是 Windows 环境。实际上 Linux 在这方面用的很少。电子工程业界是以 Windows 为中心的，比如我们将要处理的电脑插卡来数据获取和控制信号输出。

用 C 建立 Python 扩展

为了在自动化设备程序中使用 Python，我们不得不在硬件设备或者接口硬件与 Python 程序之间建立一个桥。某些情况下硬件厂商提供的应用程序接口（API）DLL 提供了所有必要功能，Python 程序只需要能使用即可。这时候可以考虑使用后面会提到的 ctypes 库。而也有可能硬件供应商的 DLL 文件没有提供必要功能，或者是，如果没有一些辅助功能就很难使用。这时候应考虑用 C 语言做扩展。图 5-1 展示了底层 API 库对象、扩展模块以及 Python 程序之间的关系。

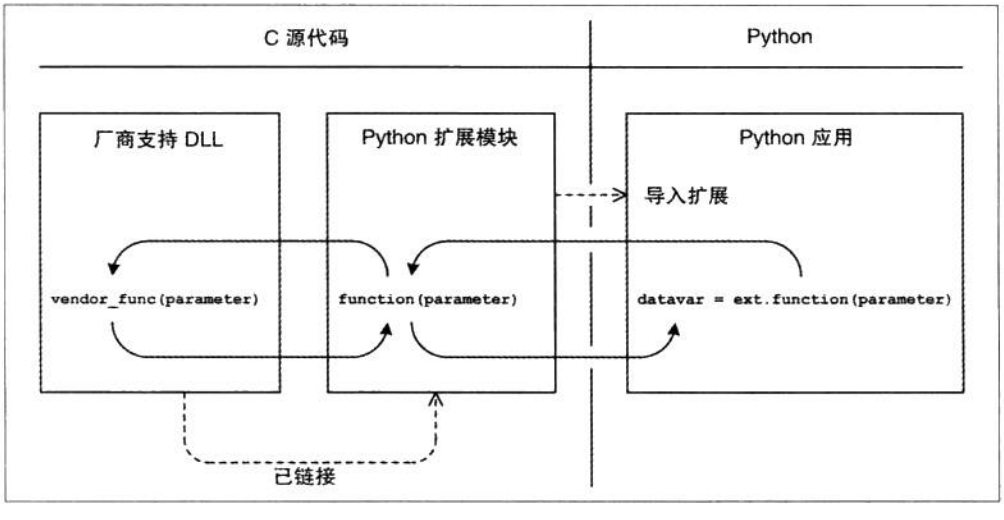


图5-1：Python扩展层次图

这一节是建立 Python 扩展模块中所涉及基本步骤的概述。也像是后续章节关于 Python 和 C 的概括总结，但是不要担心。我们这里只看看必要部分，而不会涉及太过细节的内容。后面当我们碰到某些特定应用我们会深入到细节部分进行讨论。这样可能会感觉把 Python 用的更舒服一些，没有太多的压力：这只不过是在学一个新的库 API，有一些需要注意的。

Python 的 C 扩展 API

169

在 Python 中加入一个新功能是简单直接没有痛苦的。Python 提供了 API 包含类型、宏、以及外部变量和函数，这些函数可以让 Python 引入 C 编写的扩展，从外面看上去这些扩展和 Python 自己写的扩展一样看待。使用 Python API，可引入 Python.h 头文件到 C 源代码，然后与 Python 库一起链接。

这一章目的是提供一个概括介绍，比如扩展是什么样子的，怎么搭建起来的，这本书里面我们将自己搭建大部分扩展。类似 SWIG 这样的工具（可以从 http://www.swig.org 得到）可以用来自动化建立 C 扩展模块的步骤，比如外部模块要用的头文件和大部分模块需要建立的扩展代码，这些大多编写中的常规行为，不是我们真正需要扩展做的。SWIG 也有一些有意思的用法，比如使用 Python 测试 C 代码。用法可以在这里看到：http://www.swig.org/papers/Py96/python96.html

扩展代码的模块组织

扩展源文件模块的内部布局一般是像图 5-2 所示。

170

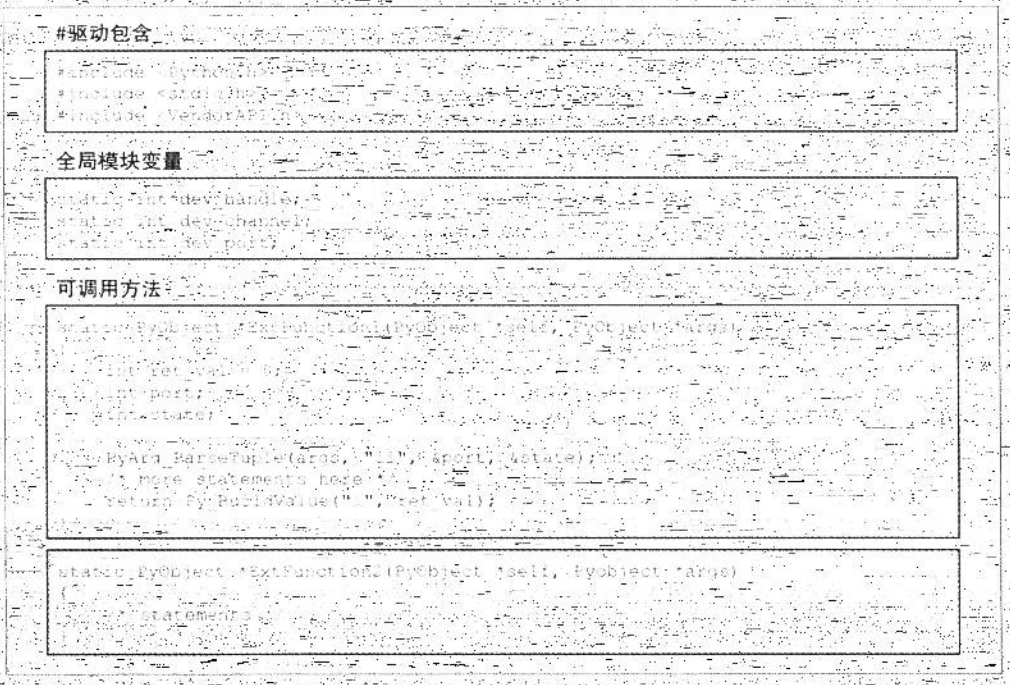


图5-2：Python扩展模块内部布局

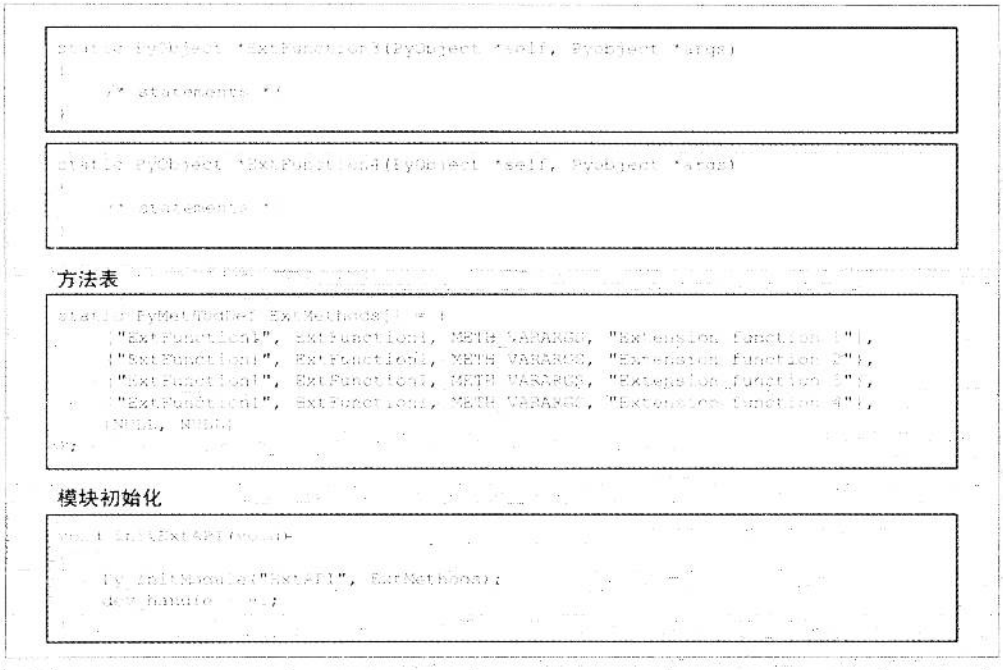


图5-2：Python 扩展模块内部布局（续图）

171 表 5-1 介绍了图 5-2 中展示的主要部分。

表5-1：Python 扩展模块

区域	描述
#include 指令	Python.h 必须是第一个被包含的头文件。后面是一些需要的 C 标准库头文件和硬件相关 API 头文件
全局模块变量	任何必要的全局模块变量都在这里声明，定义成静态的。实际上模块中每个变量和函数都定义成静态的，除了初始化函数是个例外
可调用函数（Callable methods）	一个扩展模块会有一定数量 Python 可以使用的方法。Python 把这些看作是方法，其实就是 C 源代码模块中函数。每个函数都定义 PyObject 作为返回值类型，函数参数也是 PyObject 指针形式
方法表（Method table）	方法表提供了一个编译模块和 Python 程序如何使用的动态关系
初始化（Initialization）	Python.h 必须是第一个被包含的头文件。后面是一些需要的 C 标准库头文件和硬件相关 API 头文件

Python API 类型和函数

图 5-2 的框架实例展示了 Python API 定义的一些类型：

PyObject:

代表一个任意的 Python 数据对象。

PyMethodDef:

用 NULL 结尾的数组来定义扩展模块中的方法。这个表由扩展模块的初始化函数传给 Python 解释器。

也有下面这些 API 函数：

PyArg_ParseTuple():

将参数列表以 `*arg` 的形式读入，处理为格式定义字串，并设置成对应的字典

Py_BuildValue():

根据格式定义好的字符串来构造 Python 返回值。

Py_InitModule():

传递方法表地址给解释器。

方法表

172

方法表定义了访问扩展中内部函数的入口点，后面会有一些更多的解释。表中一个条目的所有参数可以看下面：

```
{ 方法名， 方法函数， 方法标记， 方法文档 }
```

表 5-2 解释了这些参数：

表5-2：PyMethodDef 表字段

字段	解释
method_name	方法名，是一个 C 语言的 char* 类型，定义了扩展中函数可以让 Python 引入时候使用的名字。一般是用文字描述型的字符串
method_function	方法函数，是扩展模块中指向函数的一个指针，当通过扩展名字调用时，函数被调用。类型是 PyCFunction
method_flags	方法标记，是一个用来指明调用转换或者绑定转换的位域（bit field）。`METH_VARARGSMETH_VARARGS` 是最常用的，尽管某些情况下根本不需要什么标记。这个域在后面会做讨论
method_docstring	方法文档字符串，是可选的字符串用来作为函数的文档字符串。换句话说，这是会在引入扩展以后出现作为扩展的帮助部分

方法标记

方法标记是一个位域（bit field）。它的值要看标记怎么指定的，每个标记定义设置了整数变量中一个唯一的位。即使这样确保标记可以合并使用而不会彼此覆盖，在实际中只有调用转换标记 METH_VARARGS 和 METH_KEYWORDS 可以同时使用，尽管他们可以合并成一个绑定标记。关于方法标记的细节，可以参考 Python 文档（在线阅读 http://docs.python.org/c-api/structures.html）

我们只使用方法标记中的几个。其余的在处理复杂接口时更为有用，但是我们这里用不到。表 5-3 列出三个最常用的方法标记。在实际使用中，如果经常写你自己的扩展函数就会发现更多的内部标记也没有必要。

表5-3：通用方法标记

字段	解释
`METH_VARARGS`	这定义了典型的调用转换。这里，扩展函数是 PyCFunction，函数使用两个类型为 `PyObject*` 输入参数。第一个参数是 `PyObject* self (by convention)`，第二个是 `PyObject *args`（即使你可以命名为你希望的）。对 PyArg_ParseTuple（或者是相关函数）来从 args 来抽取参数值
`METH_KEYWORDS`	如果你想在函数中使用预定义关键值（比如 Python 风格的参数 varname=value），你需要制定 METH_KEYWORD 标记。在这里，函数接受三个参数：`PyObject *self`，`PyObject *args` 和 `PyObject *kwargs`，有一个字典包含了关键值和它们关联默认值
`METH_NOARGS`	函数不需要接受参数仍然需要两个输入参数，即使函数不需要检查参数。第一个参数 `PyObject* self` 是必需的，第二个参数可以设置为 NULL。

173

METH_VARARGS 示例

我们可以使用 PyArg_ParseTuple() 来抽取参数以及把数据写到函数中的变量里面：

```
static PyObject *ex_varargs(PyObject *self, PyObject *args)
{
    int param1;
    double param2;
    char param3[80];

    PyArg_ParseTuple(args, "ids", &dev_handle, &param2, param3);

    return PyString_FromFormat("Received %d, %f, %s" % (param1, param2, param3));
}
```

注意我们不需要对字符串使用地址操作符（尽管我们可以使用 &param3[0] 来达到同样

效果)。也要注意这个函数接收三个参数，它需要三个参数：不能多，也不能少。

METH_KEYWORDS 示例

函数使用 METH_KEYWORDS 调用转换，使用 Python C API 方法 PyArg_ParseTupleAndKeywords() 来抽取参数值和与名字相关的参数关键字。PyArg_ParseTupleAndKeywords() 语法定义为：

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                char *format, char **kwlist, ...);
```

参数 kwdict 是一个当函数调用时从 Python 运行时传递、指向字典对象的指针，kwlist 是一个 null 结尾的字符串列表，包含参数的名字。METH_KEYWORDS 你想要通过名字给指定参数赋值的时候非常有用。另一方面，METH_KEYWORDS 函数支持位置参数和命名参数。PyArg_ParseTupleAndKeywords() 函数可以这样用：

```
static PyObject *ex_keywords(PyObject *self, PyObject *args, PyObject *kw)
{
    int param1;
    double param2;
    char param3[80];
    static char *kwlist[] = {"param1", "param2", "param3", NULL}

    if (!PyArg_ParseTupleAndKeywords(*args, *kw, "ids", kwlist,
                                     &dev_handle, &param2, param3);
        return NULL;

    return PyString_FromFormat("Received %d, %f, %s" % (param1, param2, param3));
}
```

174

METH_NOARGS 示例

使用 METH_NOARGS 调用转换的函数只要忽略 args 参数：

```
static PyObject *ex_keywords(PyObject *self, PyObject *noargs)
{
    return PyString_FromFormat("No arguments received or expected");
}
```

传递数据

在应用程序和包装器之间传递整数没有问题，但是如果想传递字符串、浮点数、列表或者其他 Python 对象会怎样呢？ PyArg_ParseTuple()、PyArg_ParseTupleAndKeywords() 以及 Py_BuildValue() 方法可以处理通用的 Python 数据类型，也有其它方法可以使用，比如 PyArg_VaParse()、PyArg_Parse()、PyArg_UnpackTuple() 和 Py_VaBuildValue() 等等。

详细内容可以参考 Python C API 文档。

PyArg_ParseTuple() 和 PyArg_ParseTupleAndKeywords() 关键是理解用到的不同格式代码。表 5-4 列出一些比较常用的代码。

表5-4：PyArg_ParseTuple 类型代码

代码	Python 类型	C 类型	解释
b	integer	char	转换 Python 整数到一个小的整数（byte 大小）
c	长为一的字串	char	转换 Python 字符（一个长度为 1 的字符串）到 C 字符（char）
d	float	double	转换 Python 浮点数到 C 语言 double 类型
D	complex	Py_complex	转换 Python 复数到 Py_complex 结构
f	float	float	转换 Python 浮点数到 C 语言 float 类型
h	integer	short int	转换 Python 整数到 C 语言 short int 类型
i	integer	int	转换 Python 整数到 C 语言 int 类型
l	integer	long int	转换 Python 整数到 C 语言 long int 类型。
s	string	char *	建立一个到字符串的指针。字符串不能是 None 或者包含任何非 null 字节
s#	string	`char *,int`	产生两个值：第一个是指向字符串的指针，第二个是它的长度
z	string 或 None	`char *`	建立一个到字符串的指针，可以是 None（如果是 None，结果指针指向 NULL）
z#	string 或 None	`char *,int`	结果与 s# 一样，但是接收 None 或者有 null bytes 内容的字串

175

例如，如果我们有一个扩展函数，接受一个浮点值作为输入参数，比如说一个模拟量输出函数。我们可以写这样的代码：

```
static PyObject *WriteAnalog(PyObject *self, PyObject *args)
{
    int    dev_handle;
    int    out_port;
    float  out_value;

    PyArg_ParseTuple(args, "iif", &dev_handle, &out_port, &out_pin, &out_value);
    rc = AIOWriteData(dev_handle, out_port, out_pin, out_value);

    return Py_BuildValue("i", rc);
}
```

在这个例子中，函数 AIOWriteData() 是由硬件厂商的 DLL 提供。实际开发中，模拟 I/O 会比这个例子更复杂，因为常常会有类似输出范围的参数 (+/- V)，基准源，转换时钟等等。

使用 Python 的 C 扩展 API

我们这里可能不会涉及太多 Python API 中的函数和类型。但是，这里提到的函数和类型对于需要建立的扩展模块足够用了。

通用离散 I/O API

假设一个离散的数字 I/O 卡使用到某个 C 语言编写的扩展。首先，这个设备使用一个 DLL 通过 PCI 总线访问硬件，所以我们需要知道 DLL 暴露出来的 API 函数。它们定义在 DLL 一起提供的头文件里。这是一个非常简单的硬件，所以只需要八个基本函数。这里是一个假设要用到的头文件（PDev.h）：

```
/* PDev.h - API for a simple discrete digital I/O card */

typedef int dev_handle;
#define PDEV_OK 1
#define PDEV_ERR 0

/*
 * Open Device Channel
 *
 * Opens a channel to a particular I/O device. The device is specified
 * by passing its unit number, which is assigned to the device by a
 * setup utility. dev_handle is an int type.
 *
 * Returns: If dev_handle is > 0 then handle is valid
 *          If dev_handle is = 0 then an error has occurred
 */
dev_handle PDevOpen(int unit_num);

/* Close Device Channel
 * Closes a channel to a particular I/O device. Once a channel is
 * closed it must be explicitly re-opened by using the PDevOpen API
 * function. Closing a channel does not reset the DIO configuration.
 *
 * Returns: If return is 1 (true) then channel closed OK
 *          If return is = 0 then an error has occurred
 */
int PDevClose(dev_handle handle);

/*
 * Reset Device Configuration
 *
 * Forces the device to perform an internal reset. All
 * configuration is returned to the factory default
 * setting (All DIO is defined as inputs).
```

176

```
 *
 * Returns: 1 (true) if no error occurred
 *          0 (false) if error encountered
 */
int PDevCfgReset(dev_handle handle);

/*
 * Configure Device Discrete I/O
 *
 * Defines the pins to be assigned as outputs. By default all
 * pins are in the read mode initially, and they must be
 * specifically set to the write mode. All 16 I/O pins are
 * assigned at once. For any pin where the corresponding binary
 * value of the assignment parameter is 1, then the pin will
 * be an output.
 *
 * Returns: 1 (true) if no error occurred
 *          0 (false) if error encountered
 */
int PDevDIOCfg(dev_handle handle, int out_pins);

/*
 * Read Discrete Input Pin
 *
177 * Reads the data present on a particular pin. The pin may be
 * either an input or an output. If the pin is an output the
 * value returned will be the value last assigned to the pin.
 *
 * Returns: The input value for the specified pin, which
 *          will be either 0 or 1. An error is indicated
 *          by a return value of   1.
 */
int PDevDIOReadBit(dev_handle handle, int port, int pin);

/*
 * Read Discrete Input Port
 *
 * Reads the data present on an entire port and returns it as a
 * byte value. The individual pins may be either inputs or outputs.
 * If the pins have been defined as inputs then the value read
 * will be the value last assigned to the pins.
 *
 * Returns: An integer value representing the input states
 *          of all pins in the specified port. Only the
 *          least significant byte of the return value is
 *          valid. An error is indicated by a return value
 *          of   1.
 */
int PDevDIOReadPort(dev_handle handle, int port);
```

```
/*
 * Write Discrete Output Pin
 *
 * Sets a particular pin of a specified port to a value of either
 * 0 (off) or 1 (on, typically +5V). The pin must be defined as
 * an output or the call will return an error code.
 *
 * Returns: 1 (true) if no error occurred
 *          0 (false) if error encountered
 */
int PDevDIOWriteBit(dev_handle handle, int port, int pin, int value);

/*
 * Write Discrete Output Port
 *
 * Sets all of the pins of a specified port to the unsigned value
 * passed in port_value parameter. Only the lower eight bits of the
 * parameter are used. If any pin in the port is configured as an
 * input then an error code will be returned.
 *
 * Returns: 1 (true) if no error occurred
 * 0 (false) if error encountered
 */
int PDevDIOWritePort(dev_handle handle, int port, int value);
```

我们假想的设备有两个 8 位端口，每个端口的 pin（每位）可以设置成输入或者输出。pin 可以单独读写，也可以 8 位作为一组读写。

通用包装器示例

178

让我们建立一个设备 DLL 的简单包装器。我们不想要聪明，即使是我们想让包装器有扩展能力，有些东西是可以实现。我将在后面谈到这些。

第一步是决定我们要导入什么，决定要定义哪些包装器的全局变量：

```
#include     <Python.h>
#include     <stdio.h>
#include     <PDev.h>
```

注意在这个例子中，我们不需要任何静态全局变量。

所以现在我们已经有了文件开始部分了，让我们接着往下加。第二步是建立初始化函数，这是包装器 DLL（一个 .pyd 文件）载入时 Python 将要运行的，定义函数映射表如下：

```
static PyMethodDef PDevapiMethods[] = {
    {"OpenDevice",      OpenDev, METH_VARARGS,
                        "Open specific DIO device."},
```

```
    {"CloseDevice",     CloseDev, METH_VARARGS,
                        "Close an open DIO device."},
    {"ConfigOutputs",   ConfigDev, METH_VARARGS,
                        "Config DIO pins as outputs."},
    {"ConfigReset",     ConfigRst, METH_VARARGS,
                        "Reset all DIO pins to input mode."},
    {"ReadInputPin",    ReadPin, METH_VARARGS,
                        "Read a single specific pin."},
    {"ReadInputPort",   ReadPort, METH_VARARGS,
                        "Read an entire 8-bit port."},
    {"WriteOutputPin",  WritePin, METH_VARARGS,
                        "Write to a specific pin."},
    {"WriteOutputPort", WritePort, METH_VARARGS,
                        "Write to an entire port."},
    {NULL, NULL}
};

/*****************************************************************************/
/* initPDevapi
 *
 * Initialize this module when loaded by Python. Instantiates the methods table.
 *
 * No input parameters.
 *
 * Returns nothing.
 *****************************************************************************/
void initPDevAPI(void)
{
    Py_InitModule("PDevapi", PDevapiMethods);

    dev_handle =   1;
}
```

179 这里没什么神奇的；我们已经把 API 中函数和包装函数一一对应起来。这意味着 Python 调用模块中功能至少需要保存设备 IO 和设备 handle。

最后我们定义接口函数。这里是整个扩展，我将其命名为 PDevAPI.c：

```
#include   <Python.h>
#include   <stdio.h>
#include   <PDev.h>

static PyObject *OpenDev(PyObject *self, PyObject *args)
{
    int dev_num;
    int dev_handle;

    PyArg_ParseTuple(args, "i", &dev_num);
    dev_handle = PDevOpen(dev_num);
```

```
    return Py_BuildValue("i", dev_handle);
}

static PyObject *CloseDev(PyObject *self, PyObject *args)
{
    int dev_handle;
    int rc;

    PyArg_ParseTuple(args, "i", &dev_handle);
    rc = PDevClose(dev_handle);

    return Py_BuildValue("i", rc);
}

static PyObject *ConfigDev(PyObject *self, PyObject *args)
{
    int    dev_handle;
    char   cfg_str[32];
    int    rc;

    memset((char *) cfg_str, '\0', 32); /* clear config string */

    PyArg_ParseTuple(args, "is", &dev_handle, cfg_str);
    rc = PDevDIOCfg(dev_handle, cfg_str);

    return Py_BuildValue("i", rc);
}

static PyObject *ConfigRst(PyObject *self, PyObject *args)
{
    int    dev_handle;
    int    rc;

    PyArg_ParseTuple(args, "i", &dev_handle);
    rc = PDevCfgReset(dev_handle);
    return Py_BuildValue("i", rc);
}

static PyObject *ReadPin(PyObject *self, PyObject *args)
{
    int dev_handle;
    int in_port;
    int in_pin;
    int in_value;

    PyArg_ParseTuple(args, "iii", &dev_handle, &in_port, &in_pin);
    in_value = PDevDIOReadBit(dev_handle, in_port, in_pin);

    return Py_BuildValue("i", in_value);
```

180

```
}

static PyObject *ReadPort(PyObject *self, PyObject *args)
{
    int dev_handle;
    int in_port;
    int in_value;

    PyArg_ParseTuple(args, "iii", &dev_handle, &in_port);
    in_value = PDevDIOReadPort(dev_handle, in_port);

    return Py_BuildValue("i", in_value);
}

static PyObject *WritePin(PyObject *self, PyObject *args)
{
    int dev_handle;
    int out_port;
    int out_pin;
    int out_value;

    PyArg_ParseTuple(args, "iii", &dev_handle, &out_port, &out_pin, &out_value);
    rc = PDevDIOWriteBit(dev_handle, out_port, out_pin, out_value);

    return Py_BuildValue("i", rc);
}

static PyObject *WritePort(PyObject *self, PyObject *args)
    int dev_handle;
    int out_port;
    int out_value;

    PyArg_ParseTuple(args, "iii", &dev_handle, &out_port, &out_value);
    rc = PDevDIOWritePort(dev_handle, out_port, out_value);

    return Py_BuildValue("i", rc);
}

static PyMethodDef PDevapiMethods[] = {
    {"OpenDevice", OpenDev, METH_VARARGS,
181  "Open specific DIO device."},
    {"CloseDevice", CloseDev, METH_VARARGS,
     "Close an open DIO device."},
    {"ConfigOutputs", ConfigDev, METH_VARARGS,
     "Config DIO pins as outputs."},
    {"ConfigReset", ConfigRst, METH_VARARGS,
     "Reset all DIO pins to input mode."},
    {"ReadInputPin", ReadPin, METH_VARARGS,
     "Read a single specific pin."},
    {"ReadInputPort", ReadPort, METH_VARARGS,
```

```
        "Read an entire 8-bit port."},
    {"WriteOutputPin", WritePin, METH_VARARGS,
     "Write to a specific pin."},
    {"WriteOutputPort", WritePort, METH_VARARGS,
     "Write to an entire port."},
    {NULL, NULL}
};

/*****************************************************************************/
/* initPDevapi
 *
 * Initialize this module when loaded by Python. Instantiates the methods table.
 *
 * No input parameters.
 *
 * Returns nothing.
 *****************************************************************************/
void initPDevAPI(void)
{
    Py_InitModule("PDevapi", PDevapiMethods);
```

应该指出这些 API 仅使用整数和字符串作为参数值，函数也仅仅返回整数。扩展代码仅仅是在设备 API 和 Python 之间翻译包装而已。在大部分情况下，这也就是一个包装器需要做到的。

调用扩展

在实际应用中，我们可能需要建立 Python 模块来操作设备通道，以及保存 API 返回的 handle，写到指定位，然后校验修改的位是我们期望的，然后设置在其它事情上的输出模式回放能力。

让我们假设我们想得到特定输入 pin 的值，以及写一个 8bit 值到一个端口，基于特定输入 pin 的 true/false 状态来设置一些外部硬件状态。这里是一个简单的例子：

```
import PDevAPI

PDdevID = 1
PDev = 0

# open device, check for error return
  PDev_handle = PDevAPI.OpenDevice(PDevID)

if PDev_handle:
    # define port 0 as inputs, port 1 as outputs
    PDevAPI.ConfigOutputs(PDev_handle, 0xFF00)

    pinval = PDevAPI.ReadInputPin(PDev_handle, 0, 2)
```

182

```
        if pinval:
            portval = 49
        else:
            portval = 0
        PDevAPI.WriteOutputPort(PDev_handle, 1, 42)
        PDevAPI.CloseDevice(PDev_handle)
        return 1
    else:
        print "Could not open device: %d" % PdevID
        return 0
```

例子开始试着打开硬件的接口。如果成功，handle 值会返回正的整数，在 Python 相当于 True。如果失败了返回 0，在 Python 认为是 False。

接下来，我们定义输入和输出。对 PDev.ConfigOutputs 的调用就是这个功能。0xFF00 这个值说明硬件中 16 位中的高 8 位作为输出：

```
0xFF0016 = 11111111000000002
```

现在我们有一个有效可用的设备 handle，以及一个配置好的 I/O，我们可以读一个特定位，基于返回值来做一些动作。在这里如果输入位是 true，就把十进制值 42 写到输出端口；否则我们往输出端口都写 0。

数字 42 本身就很有意思（译者注：也许这里是指《银河系漫游指南》中生命、宇宙以及一切的答案 42），但是这个例子中二进制等值就是 00101010。这可以是加热器单元的控制信号，或者是比如灯的控制信号，或者是其它可以通过开关量控制的任何东西。

最后，让我们看看没有放在代码中的内容。

首先，ConfigOutputs()、ReadInputPin()、WriteOutputPort() 和 CloseDevice() 调用没有错误返回值检查。没有加入这些是为了让程序更简单，但实际中程序员应该检查返回值，特别是如果接口控制能做一些事导致损失的情况下。实际上非常常见的情况是，在高可靠性软件中负责错误检查以及故障处理比直接处理控制逻辑的代码要多得多。很容易设置一个输出值，但是值设置以后如果没有正常工作，检测问题所在就需要花一些功夫了。
183
软件必须做一些动作来从故障中恢复，至少能把系统转到一个安全状态，把可能的损害最小化。在后面章节我会展示如何做简单的故障分析，让读者能对问题有些了解，以及该如何做能让软件处理问题。

示例代码也没有提供直接处理位操作的支持。我们可以很容易写一些辅助程序来让一些指定位使能或者失效，而不需要干扰组成端口的 byte 数据中其它位。一个比较时髦的方法是”读 - 修改 - 写“(read-modify-write)。这里是如何设置一个特定位：

```
def SetPortBit(PDev_handle, portnum, bitpos):
    # read the current port state
    portval = PDevAPI.ReadInputPort(PDev_handle, portnum)

    if portval >= 0:
        insval = 1 << bitpos
        # change the designated bit
        newval = portval | insval
        # write it back out
        rc = PDevAPI.WriteOutputPort(PDev_handle, portnum, newval)
    else:
        rc = 0
return rc
```

移位操作产生一个 2 的冥值。所以一个位在 2(0) 位置的值是 1,2(4) 是 16，诸如此类。结果值，或者说蒙版（mask）会和从端口读入的值（大部分离散数字型 I/O 设备允许你从任意端口或者 pin 读取，即使是被设计为输出的）进行“或”操作，蒙版值起作用。无论指定位上是什么值，都会被设置成 1，蒙版其它位都是 0，所以原来 portval 中原始数据是 1 的仍然是 1,0 仍然是 0。（参考第三章关于 Python 位操作符如何工作的概观）

清除某个位（设置为 0）可以建立一个蒙版，除了我们想要操作的位，其它都设置为 1，然后使用“与”操作（而不是“或”操作）。这里是一个例子：

```
def ClearPortBit(PDev_handle, bitpos):
    # read the current port state
    portval = PDevAPI.ReadInputPort(PDev_handle, portnum)

    if portval >= 0:
        insval = 1 << bitpos
        newval = ~insval & portval
        rc = PDevAPI.WriteOutputPort(PDev_handle, portnum, newval)
    else:
        rc = 0
return rc
```

函数 ClearPortBit() 读取当前输出端口状态，使用左移操作建立一个位值，然后使用这个值的补码和当前端口值进行 Python 的位 AND 操作。如果 bitpos 是 2，结果数据 insval 是 4，在位置 2(2) 位是 1，或者二进制 00000100。反转结果就是 11111011，或者 251。当这个值和端口值 portval 进行 AND 与操作，原先端口位值是 1 的仍然是 1，0 仍然是 0，我们想清除的 2(2) 位置就一直是 0。

184

你可能关注输出端口脉冲干扰，或者当一个新值写回时输出一个短峰值，但是不要担心。一个正确设计过的离散数字 I/O 端口不会这样，所以你能安全地假定硬件一直能像期望一样工作。如果你碰到一些硬件发生了这些情况，你可能想考虑一些别的。你不需要担

心的原因是几乎所有的离散数字 I/O 设备都有锁存器制。数据载入到锁存器，然后锁存器驱动输出 pin。结果是任何闭锁输出已经是 1 的将保持为 1。（参考第二章关于离散数字 I/O 端口锁存器的原理图）

我们当然能用 C 语言编写处理位操作的辅助程序，放到扩展中，而不是像我们前面演示的那样在 Python 中处理。结果代码会比原生 Python 更快，但是可能不值得这么麻烦。更容易的是让核心底层 I/O 函数放在包装模块中，然后建立一个模块与底层包装器交互。Python 接口模块也能够实现错误检查以及参数校验，提供额外的不要求运行速度的功能。实际上包装器有两部分：一部分用 C 语言编写与硬件交互，另一部分用 Python 编写提供应用能使用的扩展能力。图 5-3 用块状图展示了这个概念。

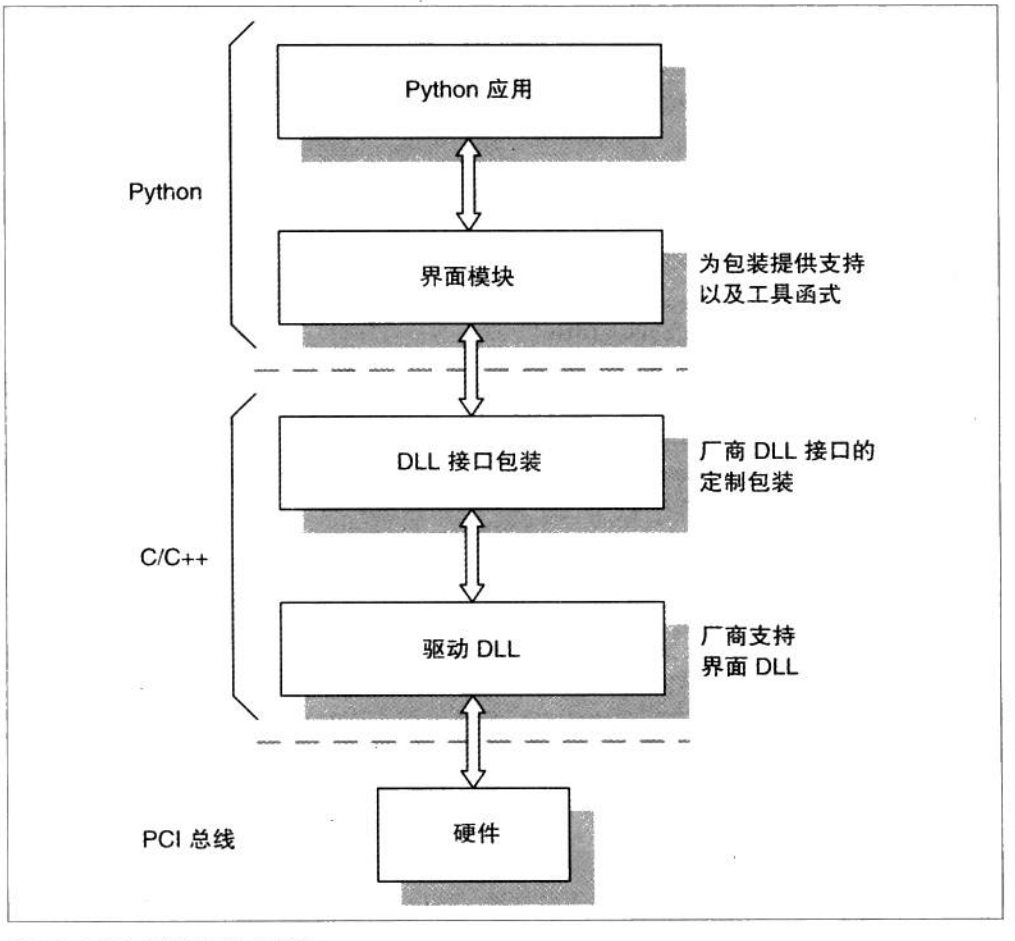


图5-3：DLL 包装扩展支持模块

Python 的 ctypes 外部函数库

这是另外一种使用 DLL 暴露出函数的方式：ctypes，Python 访问外部 DLL 对象的库。这一节介绍 ctypes 的概况。

ctypes 库给 Python 程序一种可以直接访问外部 DLL 函数的能力，不需要用 C 来编写接口代码。ctypes 从 2.5 以及后续版本以后，就是标准发布包的一部分，支持 Linux 和 Windows 操作系统，我们也需要对两个不同系统的区别有基本了解。

用 ctypes 载入外部 DLL

ctypes 输出三个主要接口类：cdll，windll 和 oledll。cdll 可以被 Linux 和 Windows 使用，支持使用 cdecl 调用转换方式。windll 类支持 Windows 库使用 stdcall 调用转换。oledll 也是用 stdcall 调用转换，但是它假定库函数返回一个 HRESULT 错误码。cdecl 是 C 程序默认调用转换方式，大部分 Linux 上的库对象都使用这种方式。在 Windows 系统你应该检查技术文档查看不同的库对象使用什么调用转换。

使用 ctypes 是向前兼容的。这里是 Windows 的 C 运行时库 msvcrt（Microsoft Visual C Run-Time）使用 ctypes 访问的例子：

185

```
>>> from ctypes import *
>>> msvcrt = cdll.msvcrt
>>> msvcrt
<CDLL 'msvcrt', handle 78000000 at 97b0f0>
>>> msvcrt.printf("Testing...\n")
Testing...
11
>>> rc = msvcrt.printf("Testing...\n")
Testing...
```

这里调用可靠的 printf() 函数，如同你期望的那样运行。注意调用返回传递给 printf() 的字符串的字符个数。如果你看看 printf() 的文档就会发现这是 printf() 要做的，尽管绝大多数情况下返回值都被忽略了。为了捕捉返回值，防止它显示到 console，我使用了变量 rc。

186

在 Linux 操作系统下需要指定共享库文件的全名：

```
libc = cdll.LoadLibrary("libc.so.6")
```

也可以将库名直接传递给类构造函数，好像这样：

```
libc = CDLL("libc.so.6")
```

这种方式在 Windows DLL 下也可以使用：

```
libc = CDLL("msvcrt.dll")
```

注意在 Windows 下 .dll 扩展是自动的，但是 Linux 下必须使用全名。

一旦 DLL 通过一个 ctypes 类被载入，它的内部函数都可以通过 ctypes 结果对象中的方法来访问。

ctypes 中的基本数据类型

当调用外部 DLL 中的函数或者获取返回值，我们需要知道 ctypes 如何在 Python 内置数据类型和 C 数据类型之间转换的。表 5-5 显示了一个 ctypes 和 C 语言以及 Python 数据类型的比较。

表5-5：ctypes,C,Python类型对比

ctypes 类型	C 类型	Python 类型
c_char	char	长 1 的 string
c_wchar	wchar_t	长 1 的 unicade string
c_ubyte	unsigned char	int/long
c_ushot	unsigned short	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 或是 long long	int/long
c_ulonglong	unsigned __int64 或是 long long	int/long
c_float	float	float
187 c_double	double	float
c_longdouble	long double	float
c_char_p	`char *(NULL-terminated)`	string 或 None
c_wchar_p	`wchar_t *(NULL-terminated)`	unicode 或 None
c_void_p	void *	int/long 或 None

理解 ctypes 如何在 Python 和 C 之间转换是必需的。例如，如果我们想从 Python 传递一个浮点数到 DLL 函数，就必须有正确的类型转换。只处理 Python 的 float 类型是不行的。

假设一个 DLL 已经载入，我们现在有一个可用的对象叫 someDLL。如果它包含一个我们要用的函数叫 someFunc()，这个函数接收一个整型参数和一个浮点类型参数，我们可能认为下面的代码可以工作：

```
int_var = 5
float_var = 22.47
someDLL.someFunc(int_var, float_var)
```

不幸的是，这其实会比较悲剧，Python 将产生一个异常回溯。原因是所有 Python 除了整型和字符串都必须用相应的 ctypes 类型包装起来，就像这样：

```
someDLL.someFunc(int_var, c_double(float_var))
```

这样就会像预期一样的运行了。

内置 Python 类型 None, integer, long 和 string 都是可以通过 ctypes 直接和 DLL 中函数直接使用的。一个 None 对象将作为 NULL 指针传递。一个 string 对象将作为一个指针传递，指针指向内存中数据申请的位置。整型数据（integer 和 long）使用平台默认的 C 语言 int 类型。注意 long 数据对象将转换成默认的 C 语言 int 类型。

使用 ctypes

即使 ctypes 比较简单，如果你想传递指针，定义回调函数或者得到一个 DLL 暴露出来变量的值，可能会变得复杂。在本书中我们会使用简单的功能，但是鼓励你看看 ctypes 文档（*http://docs.python.org/library/ctypes.html*）。另外 ctypes 中，如果不仅仅要访问外部 DLL，那么各种各样的数据类型就很有用。在 12 章我会演示如何使用 ctypes 在 Python 程序中直接处理二进制数据和 C 语言 structure，在第 14 章我们将演示如何使用 ctypes 操作一个 USB 接口的商业 API。 188

小结

本章中我们看到如何建立一个 Python 程序和外部设备供应商提供 DLL 之间的接口。幸运的是，给 Python 加功能不是一件痛苦的事，只需要有一些预先的设计。这意味着在设计和实现 Python 扩展过程中，阅读现有的文档是必需的，这也是为什么本章充斥着相关资源的网址。

阅读和理解外部 DLL 的 API 也是很有必要。当建立一个已有 DLL 的包装式扩展。偶尔也需要直接联络供应商来寻求技术支持。需要事先说明，很多供应商可能对 Python 一无所知，也不想对 Python 有什么了解，所以你可能花费一些时间来找到建立可用接口的最佳方法。

在我们已经提到的内容里，读者应该已经知道自己大体了解到什么程度，以及想达到什么高度，对如何提高自己的开发水平有一个大概了解。在第 14 章我们将演示一个使用 ctypes 的真实硬件的可用包装器，我们将在开发过程中覆盖更多细节。

推荐阅读

有意思的是，好像现在还没有一本书是只关注建立 Python 扩展这个主题。网上有不少资源，最主要的还是 Python 官方文档：

http://docs.python.org/release/2.6.5/

这个官方文档是针对 2.6.5 版本，包含了几个章节是使用 C 或者 C++ 扩展 Python。这应该是查找额外信息的第一站。

http://docs.python.org/release/2.6.5/c-api/index.html:

Python 这两章提供了详细的扩展模块的讨论和建立它们用到的功能。

http://docs.python.org/library/ctypes.html:

想使用 ctypes 这是必须阅读的。

第6章 189

硬件：工具与耗材

Byrne 定律：在任何电路中，电器装置和线路都会烧掉自身来保护熔断器。

——Robert Byrne

虽然有人能做到仅凭一把螺丝刀就能实现一个复杂的仪器系统，但在某些情况下，电烙铁、剪线钳和数字万用表（DMM）极有可能会派上用场——尤其是当某些东西从一开始就工作得不是很正常，又或者是需要考虑到某些东西会偶然性地损坏时。

在本章中，我们将看看一个用来从事仪器工作的基本工具箱可能需要哪些工具。这些工具并不算多，它们可以轻易全部装进一个小箱子里，然后搁到架子上的某个角落处。另外，总会有某些时候你极想看看在系统内部的工作情况，为此，我们加上了对示波器和逻辑分析仪两种测试设备的简短讨论，它们可以帮你消除猜测，并找到接口或控制问题的根源。

必备工具

首先，你需要一些像样的手工工具。这些工具可以从当地五金店按需求自由挑选组合或买那种用一个漂亮的拉链便携包装起来的套装。即便是一套普通的工具，干起活来也会高效而迅速得多，否则不得不沮丧地下到大厅或穿越小镇去找恰当的工具。即使这些工具在一个项目中只用过一次，损失也不大，而且它们用过之后都还在，将来也许能再次派上用场。

在处理仪器接口时，一块数字万用表是必不可少的。用它可以方便地测量输入 / 输出端子上是否有电压存在，直流电源是否已经工作（以及电压是否正确），某些型号的万用表还内置有接口，能使仪表被当做单通道数据采集设备来使用。 190

手工工具

在你的工具箱中应包含有一套十字螺丝刀和一字螺丝刀，基本上较小的型号就行。有很多五金商店和家居装修大卖场出售此类工具。一个基本的工具箱还应该有一套像宝石匠使用的那种迷你螺丝刀，在处理多芯连接器和很多接口模块上的小螺钉时，它们是必备工具。当要和电线打交道时，一副尖嘴钳和斜口钳也是必不可少的。再加上一把 6 英寸的可调扳手，以及一个组合式剥线钳和一把压线钳，这样这套基本工具就更完满了。这些工具通常被预打包成一整套出售，还有就是电子生产级工具，它们可以从多个在线供应商那里订购。

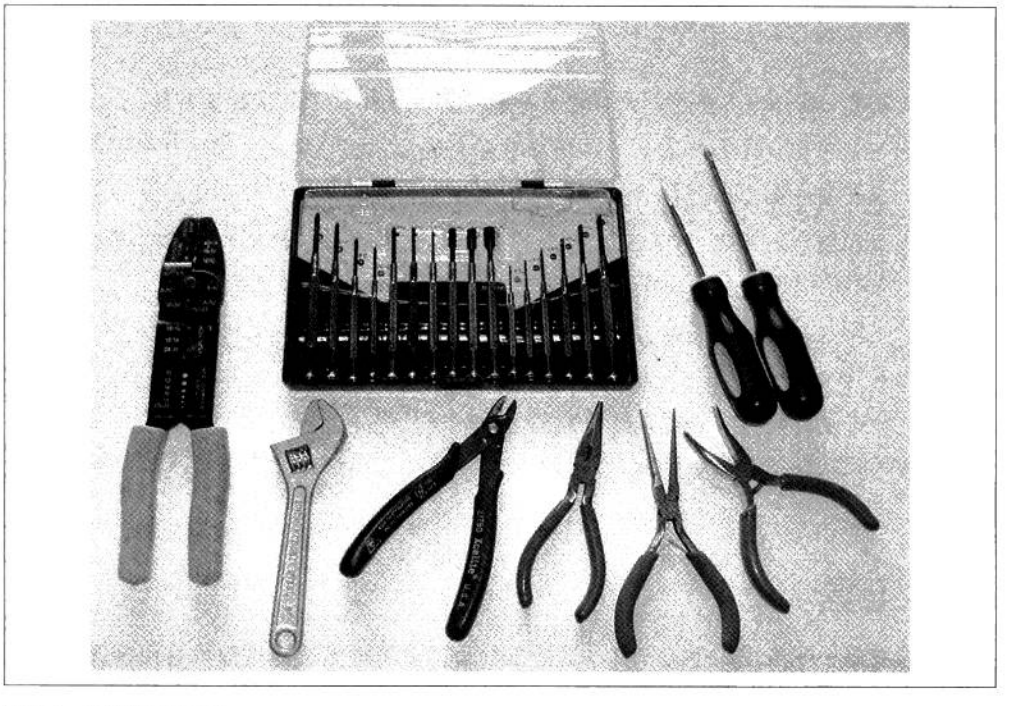


图6-1：基本手工工具

当然，把工具全都买回来组合成一个完整的储备工具箱也未尝不可。图 6-2 展示了一套装备过度的工具。对一次性工作来说，这绝对是有点过头，但如果你一直需要用到工具和怪异的小部件，一套像这样的工具也是完全必要的。

图6-2：装备过度的工具箱

善待工具

除非万不得已，否则千万别用钳子去拧螺母之类的东西。钳子通常会打滑并把螺母边缘磨圆，而这些边缘恰好界定螺母的六边形轮廓。若用钳子去拧螺母，只需试上几个轮回就有可能导致使用扳手或者套筒那样的正确工具都拧不动螺母了。

另外，不要用图 6-1 中黑色手柄那种剪线钳去剪较粗的电线。剪线钳薄而锋利的刀口很容易被弄缺并使其报废。还有就是最好不要把剪线钳当成剥线钳来用，虽然经过一定的练习后也能这样干，但很难控制力度以避免剪到或剪断内部的引线，受损的电线在有缺口的地方极易断开。因此，请使用剥线工具来剥除电线的绝缘材料。图 6-1 中上面印字的大号工具是一种剥线钳、端子压线钳及螺钉剪钳的一个组合工具。对于细线（范围在 18 ～ 32 号之间），要考虑买一把带可调节大小功能的专用剥线工具。这些工具的价格范

围从几美元浮动到 100 美元以上，生产级工具还要更贵一些，价格取决于工具质量和所具备的功能。

192

购买工具

除了本地五金店外，还有很多地方可以买到工具。其中一些零售商专做亚洲进口的折扣工具，这些东西在一些家居装修店的“特价商品处理区”也可以买到。专业级工具在电子供应公司和各个经销商那里均有出售。特价商品处理区的工具一般没什么问题，但要注意检查一下工具操作起来是否顺畅，刀口是否锋利，均匀的机械加工表面，对于套筒内侧和六角扳手外部，则要求棱角分明，不要有圆角。

表 6-1 列出的只是许多工具可能的货源中的一些典型代表，这里绝对没有为某个特定品牌或经销商做广告的意思。

表6-1：电子工具货源

货源	描述	网址
Allied Electronics	在线电子元件与工具经销商	*http://www.alliedelec.com*
Digi-Key 公司	在线电子元件与工具经销商	*http://www.digikey.com*
Electronic Toolbox	提供各种廉价工具和常用物品	*http://www.electronictoolbox.com*
MCM Electronics	出售一系列零件、常用物品、套件及工具	*http://www.mcmelectronics.com*
RadioShack	出售精选工具并在线供应商品；各家分店卖的东西依店而定	*http://www.radioshack.com*
Stanley Supply and Services	出售一系列电子方面的工具品牌	*http://www.stanleysupplyservices.com*
Techni-Tool	工具供应商并向电子工业供货	*http://www.techni-tool.com*

获取有关工具信息的最佳方式之一就是向那些已经在电子工业行业里有工作经验的人士请教。多数技术人员和工程师都有各自的偏好，你也会听到一些他们对不喜欢的工具的某些看法（及其原因）。花点时间逛逛各个网站也是一个了解都有哪些工具以及它们的价格范围的不错的办法。

数字万用表

现代化的数字万用表（DMM）经过多年的持续发展，已经从最初的粗陋形式演进成一种复杂而多功能的仪器。各种万用表的价格千差万别，依赖于种种因素，例如，精度、耐用性以及所具备的功能。一块简单可用的手持式数字万用表能以低于 20 美元的价格买到，而一些专业的高精度型号的售价则会超过 500 美元。图 6-3 同时展示了一款便宜的型号

和一款较贵的型号，后者具备用于数据采集的输出端口。

图中这两款万用表都能测量 DC（直流）电压、AC（交流）电压、DC 电流、AC 电流和电阻。图 6-3 左边的型号有测量小晶体管的插座，而右侧的发烧级仪器能测量频率和电容，它还能捕捉并将测量结果保持。此外，还有台式数字万用表，如图 6-4 所示。

193

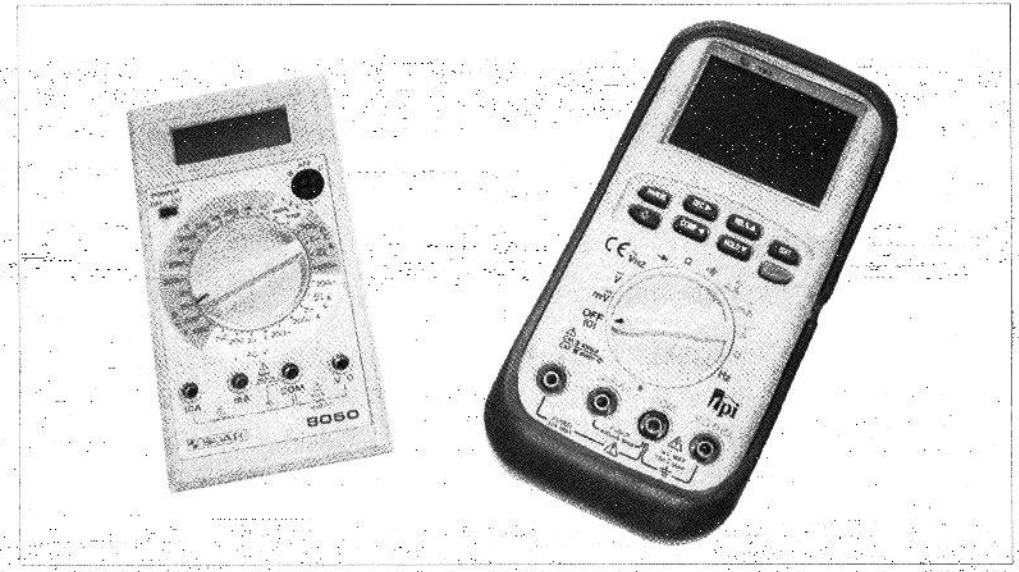


图6-3：数字万用表

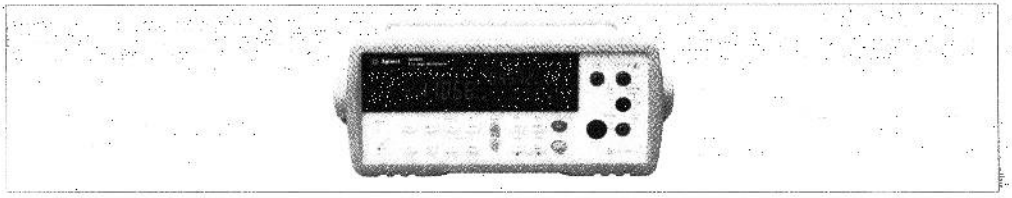


图6-4：精密台式数字万用表（Agilent 34405A）

高端万用表在价格上会高达数千美元，但它们具备工作精度好、可靠性高的特点，而且还有数据和控制接口。当我们在第 11 章中介绍带串行和 GPIB 接口的仪表时，我们会再次提到图 6-4 所示的这种仪表。

使用高端数字万用表时，你还能获得一份校准证书，而且大多数专业级仪器会定期使用商业或内部校准服务进行校准。除非你要做的工作对高精度和可复验性测量有所要求，

否则这些高端仪表完全是不必要的。高端仪表还有一个妙用，就是偶尔可以在同一电压源和参考电阻下用一块仪表对另一块仪表进行交叉检查，即便如此，高端万用表也不是很有必要。

194

数字万用表的分辨率

数字万用表的分辨率各不相同。这里说的分辨率是指万用表能准确显示的有效数字的位数。一般而言，分辨率越高，则仪表的价格也就越贵。

数字万用表常见的分辨率有 3½、4½ 和 5½ 位。其中，½ 指的是最高位的数字，它只能显示 0 或 1。数字万用表的基本分辨率是由仪表内部的模 - 数转换器（ADC）的分辨率位数所决定的。

数字万用表使用提示

下面是使用数字万用表时要时刻谨记的一些注意事项：

- 在测量以地为参考的电压时，应始终先将万用表负极表笔接地。如果有条件，用某种鳄鱼夹来连接，这样它就不易脱落，然后只用正极表笔来测量。请谨记，如果接地的表笔脱落，万用表（以及未连接的接地表笔）上的电位会变得和正极表笔所接触的电位相同。这在你测量低压直流电时问题可能不大，但当测量的是高压电时，这绝对是极其危险的。
- 要警惕万用表笔尖在测量端子上有可能打滑。如果在一块通电电路板上出现这种情况，并且表笔尖正好将连接器或芯片的引脚短路，那可能会出现灾难性的后果。表笔打滑也可能导致你的双手接触到潜在的致命电压，这是另一个将接地表笔夹住，然后只用正极表笔去测量的正当理由。在测量像墙壁交流电或高压直流电路那样会导致伤残或致命的电压时，遵循古老的“放一只手在身后”这一法则是不会错的。
- 绝对不要在任何通电电路中测量阻值。当万用表被设定到电阻测量模式时，它通常不能承受稍长时间的电压输入，时间一长就会导致仪表损坏。
- 绝对不要将万用表接到高于其最大额定值的电压源上。但这通常并不是个问题，因为目前多数数字万用表能够承受高达 600V 的输入电压。如果你要测量高于仪表最大量程的电压，应该考虑购买一个特殊的高压探头。另外，请记住交流电压通常给出的是 RMS 值（有效值），而不是峰值。
- 在试图做测量之前，请务必检查仪表的设置。虽然几乎所有的数字万用表在输入上都有内部熔断器（俗称保险丝），但用万用表的电流挡去测量电压依然是个非常糟糕的主意。这样做实际上在正极和负极之间形成了一个短路（中间串联了一个低阻值的分流电阻），这时如果一个表笔是接地的，那么仪表的熔断器将被烧断，内部电路将被损坏，被测电路甚至也会被烧毁。

- 万用表的表笔引线存在电阻。这个值通常很低，约为0.5Ω，但如果你试图测量某些像低阻值的功率绕线电阻时（例如，0.1Ω），万用表引线所贡献的读数会超过被测器件所贡献的读数值。有些数字万用表有引线阻值补偿功能，但对于那些没有这种功能的万用表，你应该先测量引线的阻值，然后从仪表指示的读数上将其减去。像电流分流器这类装置，它们的电阻通常只有几毫欧，这时通常要使用惠斯通电桥或凯里-福斯特电桥（Carey Foster bridge）来测量极低的电阻。作为历史的见证，图6-5展示了一台老式的电桥测试仪。 195

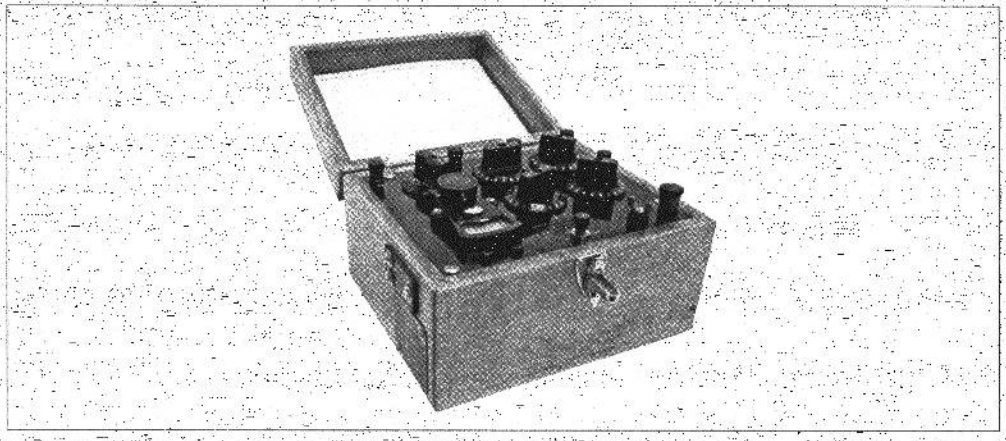


图6-5：老式的Leeds & Northrup电阻电桥测试仪

焊接工具

利用熔化的低熔点合金把两条导线结合起来，或者把器件的引脚与印制电路板上的铜走线相连接，这就是焊接（它是低熔点合金众多应用中的一种）。锡铅合金（通常是60%的锡和40%的铅，当然，还有其他成分）曾经是最常用的焊接材料，然而随着近年对要求禁用含铅材料的推动，新型合金也开始不断涌现。然而，当要购买焊锡以备不时之需时，最方便的类型还是锡-铅合金。锡-铅合金相对而言还是比较安全的（在合金中，铅被束缚到锡上），而且只要使用者遵循适当的预防措施，含铅焊锡不会导致明显的健康风险。

电子应用所使用的焊锡通常是一种软而粗的银色金属丝，它的芯是中空的，里面填充有松香助焊剂材料。助焊剂在焊锡熔化之前就会熔化并流动，这可以帮助去除焊锡和被焊接表面的氧化物。有人喜欢用膏状助焊剂，它是一种浓浓的、黏黏的棕色膏状物，通常装在小罐子里出售。其用法是在实际焊接开始前把它涂到焊接处，助焊膏有时也与带松香的焊锡丝一起使用，以达到在电路板上形成稳固的连接。锡铅比为60/40的焊锡熔点约为188℃。 196

电烙铁的尺寸大小、工作温度，以及价格不尽相同。本地电子 / 五金店里通常有便宜的电烙铁出售，功率约为 15W，头部是铅笔尖形状。这种烙铁能应付大多数小线径和线路板类的工作，在任何地方，它们的价格都在 10 ～ 25 美元之间。不过这种烙铁没有温度控制功能，因此，烙铁尖的温度有时会变得很高，从而导致不良焊点或者会烧坏电路板。还有就是在焊接稍大的东西或较粗的电线时，这种烙铁的温度会急剧下降。如果你打算以后要经常焊东西（而不是偶尔为之），那完全有必要花点钱买把带温度控制而且可以更换烙铁尖的电烙铁。这些烙铁的价格介于 50 ～ 300 美元之间，由其功率大小和其所具备的功能（比如温度用户可控、数字温度显示、内部接地，等等）而定。图 6-6 是一款廉价的 15W 铅笔型电烙铁。你也许已经注意到图 6-2 中工具箱的顶部托盘里也有一把类似的烙铁。

图6-6：铅笔型电烙铁

除了烙铁，你还要有清洁烙铁尖的东西以及在烙铁不使用时放置它们的方法（直接摆在工作台或桌面上可不行）。市面上有那种带托盘的烙铁架，托盘中放置湿润的海绵用来清洁烙铁咀上的锡渣和氧化物。还有一种无须用水的烙铁洁咀器，它的结构是一个小罐子里塞着一团蓬松的、弹簧状的金属细丝。此外，有一种膏状物质也能用来清洁烙铁尖并让其变得光亮（也装在小罐子里出售）。后面提到的这两种东西在电子制造业里都常被用到，然而它们在普通用户的工具箱里并不十分流行。

在网上有许许多多讨论焊锡、电烙铁以及如何焊接的资料。在你开始真正的项目之前，最好找些废旧的导线或者报废的电路板做大量练习。焊接这门技术可能听起来容易，看上去也简单，但是要想焊得好，则需要一定的技术级别，而这只有不断地练习才能办到。

焊枪与电烙铁

提醒一句：除非真的需要，否则请不要购买焊枪。焊枪能产生大量的热量，它们适用于焊接像粗导线或铜管之类厚重的材料。它们不适用于细电线、连接器或电路板，有时即便是熟练的操作者，在使用焊枪时也会因为疏忽而造成严重的损坏。

最好能有的工具

除了我们已经讨论过的必备工具外，还有其他一些工具，它们在某些特定的工作中极其有用。它们使用的频率不如一般的手工工具那样高，但是真正要用到的时候，其他任何工具都无法替代它们。考虑采购下面这些工具：

老虎钳

这种沉重、方形下巴的钳子被用来折弯、扭曲以及剪切粗电线。也可以用它们折弯薄的钣金条或者干净利落地折断一片塑料。老虎钳在备货充足的五金店里都能找到，偶尔也能在特价商品处理区看到便宜的进口货。

六角扳手组

六角扳手组有时也称为六角匙或 *Allen* 扳手，它是一种带有六边形横截面的简单工具，用来拧头部是内六角的螺栓和螺钉。这些紧固件在电子设备和实验室设备上很常见，除了六角扳手，其他工具根本奈何不了它们。六角扳手也开始变得相对便宜，在尺寸方面，SAE[译注1]和公制的都有。

螺母起子

从它的名字就可以看出来，这些工具是被设计来拧螺母的。它们通常看起来像螺丝刀的样子，只不过头部是套筒，而不是螺丝刀的尖头，还有一些具有 T 形手柄，以施加更多的力矩到螺母上。尺寸为 SAE 和公制的都有，可以整套购买，一套装有 4 ~ 8 个或更多各种大小的工具。

微型套筒组

它们类似于螺母起子，套筒被设计用来拧六角螺母和六角螺栓，不同之处在于它们使用的是呈直角的棘轮手柄，这样可以比螺母起子施加更大的力矩（有时相差非常大）。一套小巧紧凑的微型套筒组包含一组套筒、一个转换延长杆以及一个棘轮手柄。在尺寸方面，SAE 和公制的都有（一般组合在同一套中）。这种工具一般用于稍大的螺母和螺栓，典型的 SAE 尺寸范围是 3/16 ~ 9/16 英寸（1 英寸 =0.0254 米），公制尺寸范围是 4 ~ 14mm。

译注 1：SAE 即 Society of Automotive Engineers（美国汽车工程师学会）的简称。

198

高级工具

用数字万用表这样的工具可以完成很多工作，但有些事情是它所不能办到的。一个数字万用表无法应付的应用就是对电子波形的直接测量。另一个是对多条并行线路上的逻辑信号进行可视化显示。对这两种情况，示波器和逻辑分析仪这样的工具可以澄清事实和猜测。

示波器

示波器是一种非常古老的仪器，它以各种形式存在至少 80 年之久（也许还要更久，这取决于人们如何定义示波器）。示波器成功的原因在于用它来查看随时间变化的电子信号非常有用。早期的电子示波器使用真空管来放大信号，并用一种专门的叫做静电阴极射线管（CRT）的真空管来作为主显示器。示波器 CRT 在工作原理上类似于老式计算机显示器和电视机上所用的那种 CRT，不同之处在于示波器的 CRT 通常用电场，而不是偏转磁场来偏转射向银光屏的电子束。

现代示波器已演变为数字化仪器，使用平板显示器中的那种 LCD 作为显示器——真空管放大器的日子已经一去不返了，而且 CRT 显示器也在快速被淘汰。图 6-7 是一个经过简化了的双踪数字示波器的前面板。

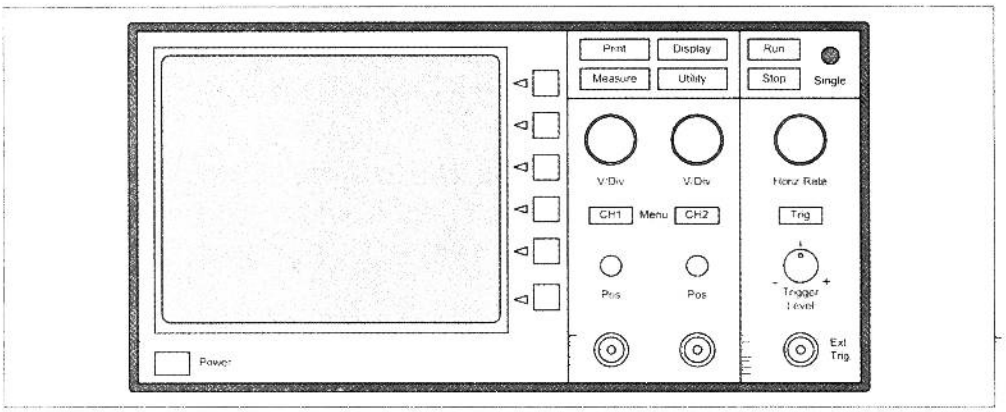


图6-7：普通数字示波器

199

虽然有单踪示波器，但能同时显示两路信号的双踪示波器却最为常见。此外，还有一些示波器能显示 4 路或更多的通道。最新的高端示波器备有彩色显示器，具有高级内置信号处理和测量功能，并带有网络接口。

示波器的垂直输入通道将输入信号进行放大，并使显示点在相对于被定义为 0 或地的水平线

上做上下移动。仪器水平控制部分驱动显示点从左向右穿过显示屏，这被称为水平扫描。一个触发电路对水平扫描和垂直通道中的一个输入信号进行同步，以使显示的波形不会在屏幕上来回游走。

示波器的一个基本指标是其带宽大小。低端仪器的带宽通常低于 25MHz，这是指它们将衰减 25 MHz以上的信号。当信号中有高于其基频的谐波存在时，这个问题开始显现出来。例如，如果用一台 25MHz 的示波器测量一个 25MHz 的信号，它极有可能显示基本的波形，但是它不能准确显示高于这一频率的谐波，这样信号的一些关键特征将无法显示出来。这在检查方波时就是一个问题，方波可能包含导致故障原因的高次谐波成分，但这些成分在低速示波器上根本无法显示，如图 6-8 所示。

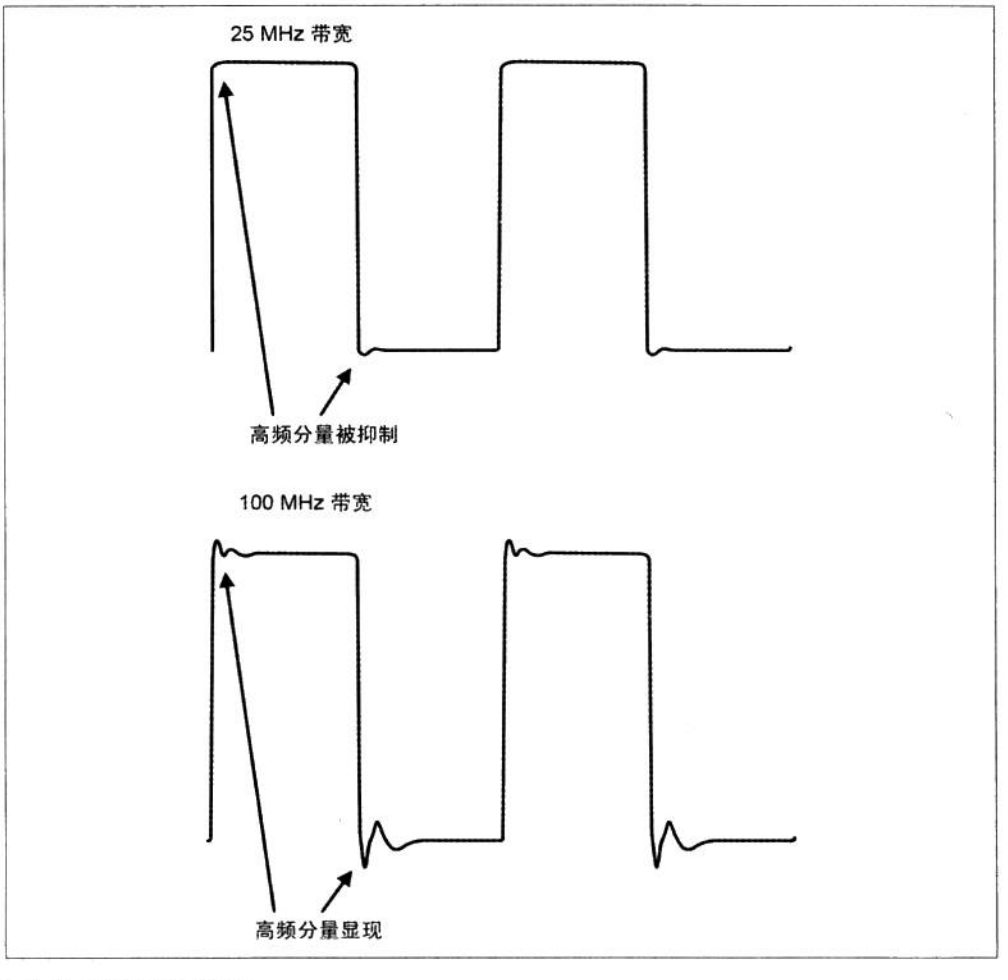


图6-8：示波器带宽效应

一个经验法则是示波器的带宽至少是欲测信号基频的 4 倍。如果手边只有一台示波器，并且其带宽接近于最快的测量信号频率，那么你可能认为信号中存在并未显示的成分。

图 6-9 显示了一台数字示波器的屏幕画面。从图中可以看到两个通道，其中一个通道的范围是 0 ~ 3V，另一个通道的显示范围是 0 ~ 12V。测量光标（*x* 和 *o* 符号）用来测量光标之间的时间间隔，在这里是 36.3s。这里用的是一台使用 CRT 显示器的老型号仪器。

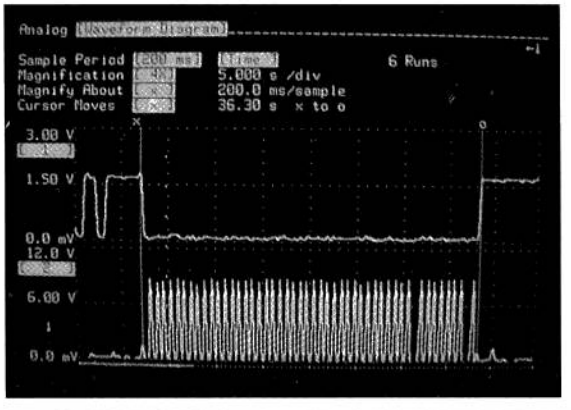


图6-9：数字示波器显示画面

如果你对示波器不太熟悉，那值得花点时间仔细研究一下它们的工作原理以及如何使用。在网上可以找到很多非常好的相关资料。要记住示波器（以及多数其他的测试仪器）只能显示其测量能力范围之内的数据，并总是试图对丢失的数据进行插值处理，要是假设所显示的数据是完全精确无误，则会得到错误的结果。

逻辑分析仪

逻辑分析仪是一种相当有用的仪器，用于测量和监视数字电路内部的工作情况。逻辑分析仪同时对一组数字输入进行捕捉，并将二进制值存储在短期踪迹存储器中。踪迹存储器中的内容稍后被读出并以时序图的形式显示出来。在第 2 章，中我们已经见过这些时序图，后面我们会见到更多这样的图。

200 图 6-10 是一台简易逻辑分析仪的框图。这个框图描绘了一台功能完备的仪器，但是还有一些别的方式可以达到同样的效果。例如，如果信号以相对较慢的速度变化，可以把 PC 上的并行打印机端口当做一个简易的 4 通道逻辑分析仪使用（PC 并口上有 4 条输入线）。

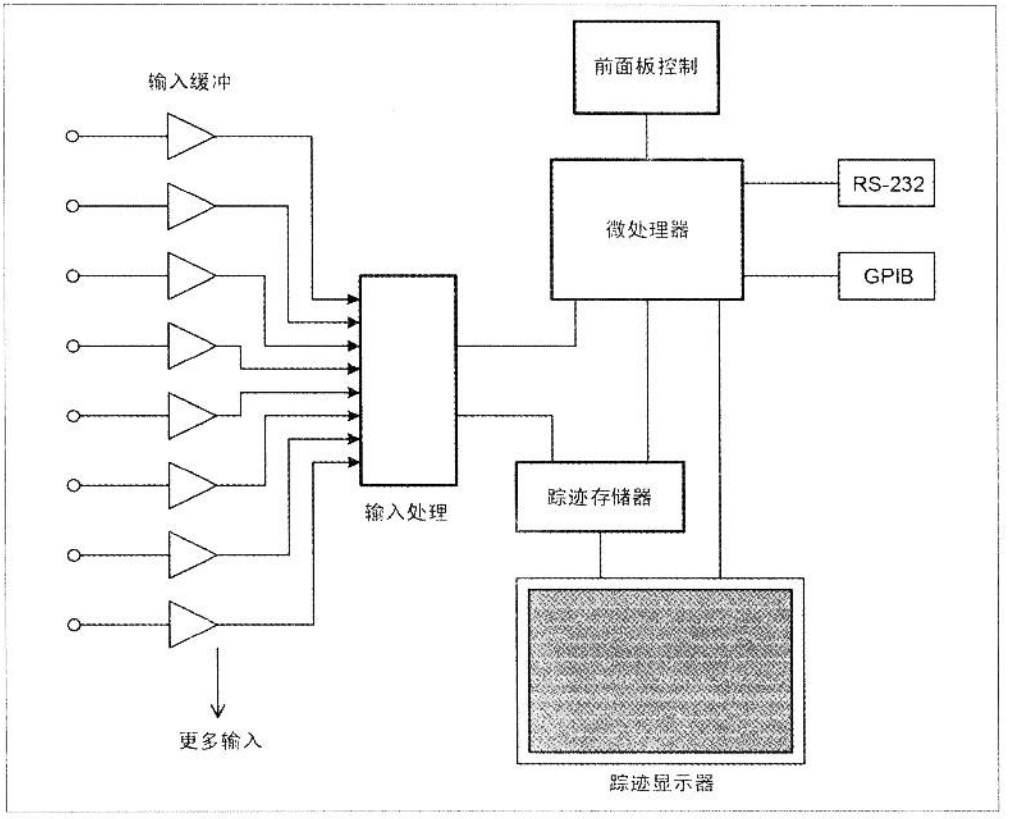


图6-10：逻辑分析仪的内部组织结构

此外，还可以买到便宜的USB逻辑分析仪模块，这种模块使用PC显示器来显示，而不是自带一个显示器。图6-11就是一个例子。

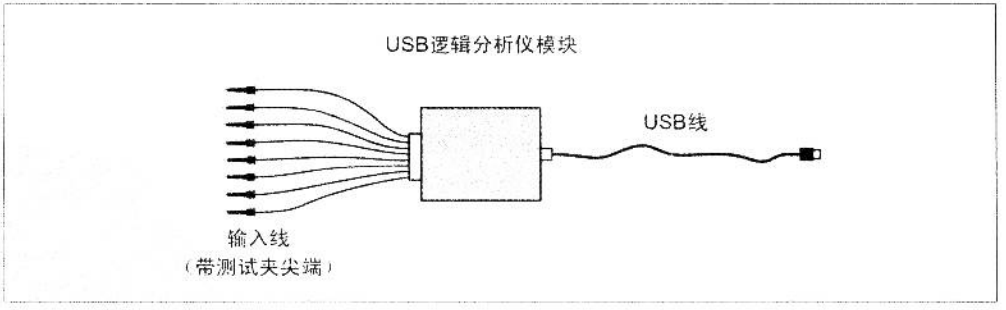


图6-11：USB逻辑分析仪模块

USB 逻辑分析仪模块的价格和性能千差万别，价格从大约 150 美元到 6000 多美元不等。由于基本上不需要捕捉并显示 50MHz 以上的信号，因此，对大多数仪器化应用而言，一台低价位的逻辑分析仪就能完成所有的任务。此外，某些逻辑分析仪还具有诸如串行协议分析仪之类的功能，这在检查串行接口的数据流时非常方便。

测试设备注意事项

下面是一些在使用电子测试设备时需牢记的要点：

- 作为一般规则，除非仪器被明确设计为所谓的“浮地”操作模式，否则千万不要把任何用交流（AC）供电的示波器、逻辑分析仪或其他仪器的探头接地线连接到电路的 DC 电源上。在进行测量时，除非仪表被专门设计为允许接地点“浮动”，否则探头接地线仅供接地使用，并且要始终保持连接到接地点上。如果你只想确认两点之间是否有电压，而且这两点都不是接地点，那么请使用数字万用表或者其他电池供电的手持示波器进行测量。
- 请始终注意仪器的最大电压输入量程。有些仪器的上限电压为 25V DC，而有些仪器配合适当的探头，可以测量高达数百伏的电压。
- 在使用数字示波器或逻辑分析仪时，应知晓仪器的输入频率范围是受限的。如果输入频率超出最大限制，则显示结果将会“走样”，并显示出一个失真的信号，该信号的频率低于实际被测量信号的频率。制造商在说明书上通常会标明仪器的最高可测量频率。

耗材

如果手边储备有一些像电线、接线帽和电工胶带之类的常用物品，而且在需要时可以迅速取用，则干起活来会顺心很多。不尽如此，它们还能保证最终结果既稳定、可靠，又有专业的外观。表 6-2 列出了一些可备不时之需的推荐耗材。

表6-2：推荐耗材

物品	描述
绝缘电子连接线	#22[译注2]绞合线，各种颜色
	#20 绞合线，各种颜色
	#18 绞合线，各种颜色
	#16 绞合线，各种颜色
屏蔽同轴电缆	RG-58，特性阻抗 50Ω
接线帽（Wire nuts）	小号（用于 #18 ~ 14 导线）

译注 2：#22 是美国线规（American Wire Gauge，AWG）。

续表

物品	描述
电工胶带	黑色
聚酰亚胺（Kapton）高温胶带	1/4 英寸宽
热缩管	各种直径（从 3/32 英寸到 1/2 英寸）
电阻	1/4W;100Ω、330Ω、1kΩ、2.2kΩ、4.7kΩ 和 10kΩ
线耳（Wire lugs）	适用于 #18 ~ #16 线径

电线和电缆通常绕在线轴上出售，长度从 10 英尺到 1000 多英尺不等。对于电子连接线，买 25 英尺一卷的那种就行，在色彩上则选择一些标准颜色（如红色、黑色、白色、蓝色以及绿色），通过颜色，人们能够对电源线（红色）、地线（黑色）和信号线进行快速识别。有些电线，尤其是小线径电线，它们的芯是单股铜丝，而不是多股绞合线。单芯线在过度折弯或者在剥线被切伤时有断裂风险，因此，除非有什么特别的理由，否则不推荐使用。相比之下，绞合线要更柔韧，而且更耐用。

表 6-2 中里提到了 Kapton 高温胶带，很多航空航天工业和电子制造业之外的人也许都没听说过这种东西，因此，值得对其进行简要说明。Kapton 胶带从本质上说是一种聚酰亚胺（polyimide）薄膜，它具有极好的耐温性能（–273℃ ~ +400℃）。除了其他一些应用外，Kapton 被用于制造柔性电路板，被用做接线间的绝缘材料，在宇宙飞船上被用做电缆束的保护包覆材料，太空服有一个外层也用的是这种材料。Kapton 胶带在撕下时几乎不会留下任何残渣，它在很多应用中都要好过电工胶带。Kapton 在那些需要考虑低释气（outgassing）因素的环境中（如真空室）也有很多应用。最后一点，Kapton 胶带呈透明的金黄色，将其用来给成品线缆做标记会相当不错。

除了 Kapton 高温胶带偶尔不太好买之外，表 6-2 中列出的大部分东西都能从表 6-1 所列 204
的供应商处买到。小批量的 Kapton 高温胶带可以从 *http://www.kaptontape.com* 订购。

全新和二手

大多数仪器项目对测试设备没有很高的要求。由于数据采样和控制更新时间间隔相对较长（100ms 到数分钟），因此，完全没必要采用最新、最贵的高速设备。也就是说，那些放在库房架子上掉灰的旧示波器应付当前的工作可能都还绰绰有余（当然，前提是还能正常工作）。

既然如此，那么购买二手测试和仪器设备也是完全可行的。eBay 上就有二手设备出售，但你要做好适当的心理准备，因为当你收到包裹并打开时，可能会发现物品已经摔坏或者根本就不能工作。这种情况相对比较少见，但也不是完全没有可能。

如果你买的东西在运输中被快递公司又扔又踢导致损坏，又或者发现实际物品和卖家描述的根本名不副实，此时可以使用 PayPal 的买方保护方案（PayPal 是 eBay 的首选支付系统），这样钱倒是可以追回，但所浪费的时间却一去不返了。

本地电子商店也可能会打折出售一些精选的二手或多余设备。选择旧设备的秘诀是你得知道要用它来测量什么东西。比如，在选示波器时，你可能会要求示波器的最低带宽为 100MHz。数字万用表至少要有足够的位数，这样当你测量某系统的电压时，才能满足对测量精度的要求。在大多数情况下，3½ 位的万用表一般就能满足要求，如果要检查 ADC 或 DAC 的基本工作情况，则有必要使用 4½ 位的万用表。

小结

现在，对于到底需要买些什么样的基本工具，你心里应该有个数了，也大概知道要到哪里才能找到这些工具。此外，我们还对一些可能会用到的基本电子测试设备进行了介绍，并且还对购买全新和二手工具设备的优缺点进行了简要的讨论。

除了基本手工工具和一块合适的数字万用表外，你基本上不需要其他工具就可以开始干活了（可能偶尔还需要几样专用工具来打理连接器）。

205

推荐阅读

如果你想对测试设备和测量方面的内容进行更深入的探索，下面这本书可以作为一个不错的入门起点：

Electronic Test Instruments: Analog and Digital Measurements, 2nd ed. Robert A. Witte, Prentice Hall, 2002.

该书对基础测量理论作了深入的介绍，接着对现代电子测试仪器及其应用作了一个可读性极强而且信息丰富的概述。每个主题都以作者自己经历的实际案例为支撑。

你还可以在电子电路理论书籍的附录中找到对诸如数字万用表和示波器的基本理论和应用的描述。

最后，很多测试设备制造商为自己的产品提供了极佳的在线教程。因此，建议在搜索网页前先到制造商网站上看看，以获取更多的信息。

第 7 章 207

物理接口

> 通常，连接器导致的故障要比其他类型的组件更多。这些故障大多数没有被报告，是因为重新连接一下就可以“修复”它们。
>
> ——W.Ireson、Clyde Coombs 和 Richard Moss, *Handbook of Reliability Engineering and managment*（第 2 版），1995

本章将介绍接口的类型，包括物理的和软件的，在我们试图通过 Python 进行数据采集或控制仪器时，这些接口是最有可能用到的。虽然我们在第 2 章里曾经看到过一些简单的电子产品，但是在这里我们将开始深入了解它们。

首先看一下物理连接器，特别是常用的基于 PC 的仪器所带的接口。在本章结束时，你将能确定哪类接口是你希望从给定连接器类型中找到的，或者至少能轻而易举地识别出连接器。你也应该记住并不需要总是遵循连接器的使用惯例，所以不需要为发现一个 DB-9 接头被用于电动机控制信号，或者一个圆形航空接头被用于以太网连接而感到吃惊。

然后，我们将注意力转向串行接口，即 RS-232 和 RS-485，这是两个最常见的串行接口类型。接着我们将学到 USB 和 GPIB/IEEE-488 接口的基础知识，并且讨论可能会在哪里碰到它们。

最后，我们研究将 I/O 硬件设计成可以插拔到 PC 总线——通常为 PCI 类型的电路板——以期得到硬件厂商在软件方面的支持。

208

连接器

物理接口都配有某种类型的连接器。可能是常见的 9 针或者 25 针的 DB 型和 PC 上的打印机接口中的一个，也有可能是 4 针的 USB 接口。还可能是常见于工业和航空设备的一个圆形连接器，从 2 针到 100 针以上。连接器的大小和形状还要同印制电路板（PCB）

相吻合，甚至有些类型的连接器就直接做在 PCB 的边上，如 PC 主板上的插卡槽。接线端子用于连接导线的一头，并且针对不同的设备有多种类型。通过快速浏览电子产品分销商商品目录中“连接器”章节，你可以了解到哪些是可用的。

图 7-1 显示了同时使用圆形连接器和接线端子的面板配件。我们将简短介绍这些连接器。

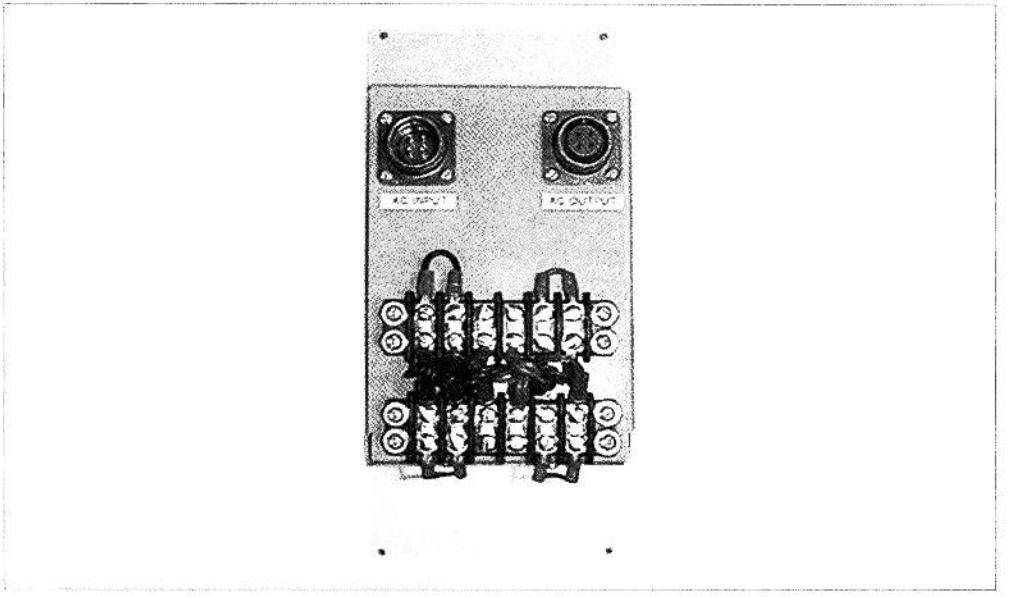


图7-1：带有不同类型连接器的配件

DB 型连接器

图 7-2 显示了一个 DB-9 连接器的母头，图 7-3 显示了其另一头。这些通常用于 RS-232 接口，但它们也可用于直流电源连接器和 RS-485 连接器，甚至是特定仪器的接口信号连接器。

209

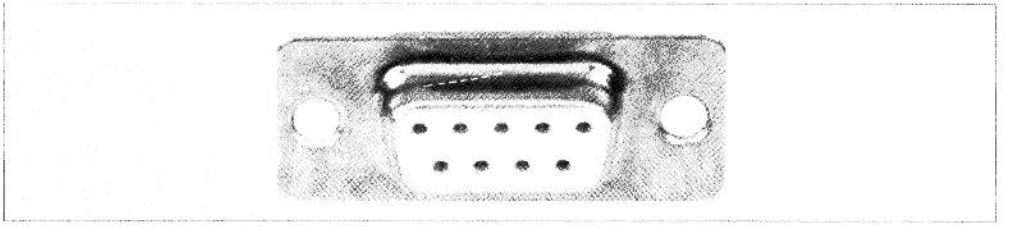


图7-2：DB-9母头连接器

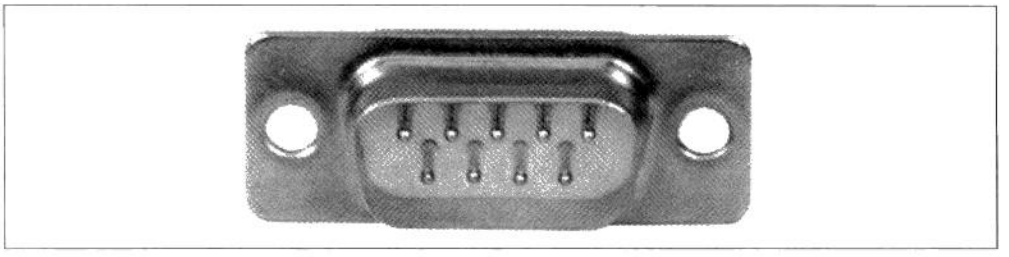


图7-3：DB-9公头连接器

另一种常见的连接器是 BD-25。它和 DB-9 的外形相同，其主要的区别在于，它为了适用更广泛，采用了 25 针，而不是 9 针。DB-25 连接器最常用于 RS-232 标准的完全实现。它也常用于桌面 PC（和一些较老的笔记本电脑）上的并口打印机输出端口。由于 DB-9 更小，并且不需要所有的原始 RS-232 信号就能实现一个串行接口，因此，DB-9 已经普遍替换掉 DB-25。图 7-4 显示了一个具有“锡孔”的“DB-25 母头”连接器的接线背面（显示得更多一点）。

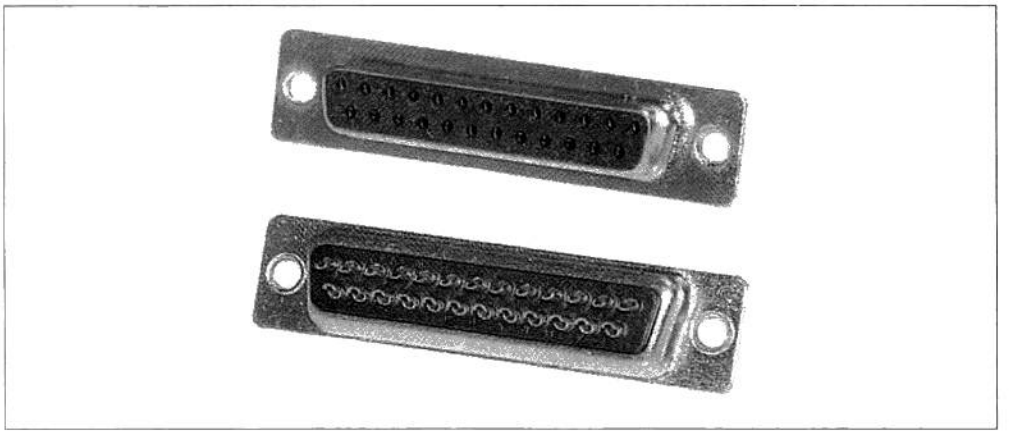


图7-4：DB-25的前后形状

请注意，到目前为止，所见到的 DB 类型的连接器都是面板或者外壳形式。换句话说，它们被设计成既可以作为金属或者塑料面板通过螺栓孔连接到机箱一侧，也可以用所谓的“后壳”方式安装，如图 7-5 所示。

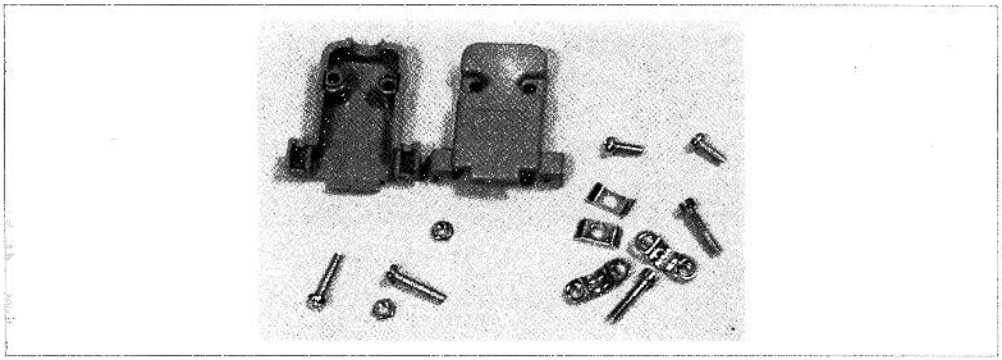


图7-5：DB-9的后壳部分

完全组装起来的连接器看起来就如图 7-6 所示的那样。

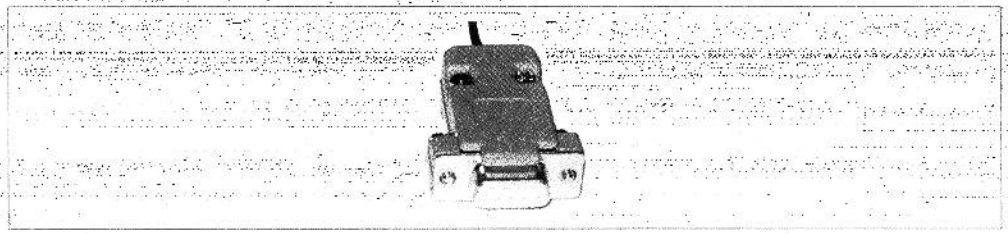


图7-6：后壳组装好的DB-9连接器

无论是 DB 类型连接器的公头还是母头，都很适合 PCB 安装。PC 主板上的 DB-9 和 DB-25 连接器与主板上的电路走线直接连接。至于针数，DB 类型连接器也可以是 15 针、37 针和 50 针。最后，高密度的 D 型连接器提供了多种类型，如用于视频显示器的连接。这些连接器使用了三排或更多排的针脚。

DB-9 连接器的针脚编号如图 7-7 所示。要注意的是，从连接器的正面看过去，公头和母头的编号是相反的。

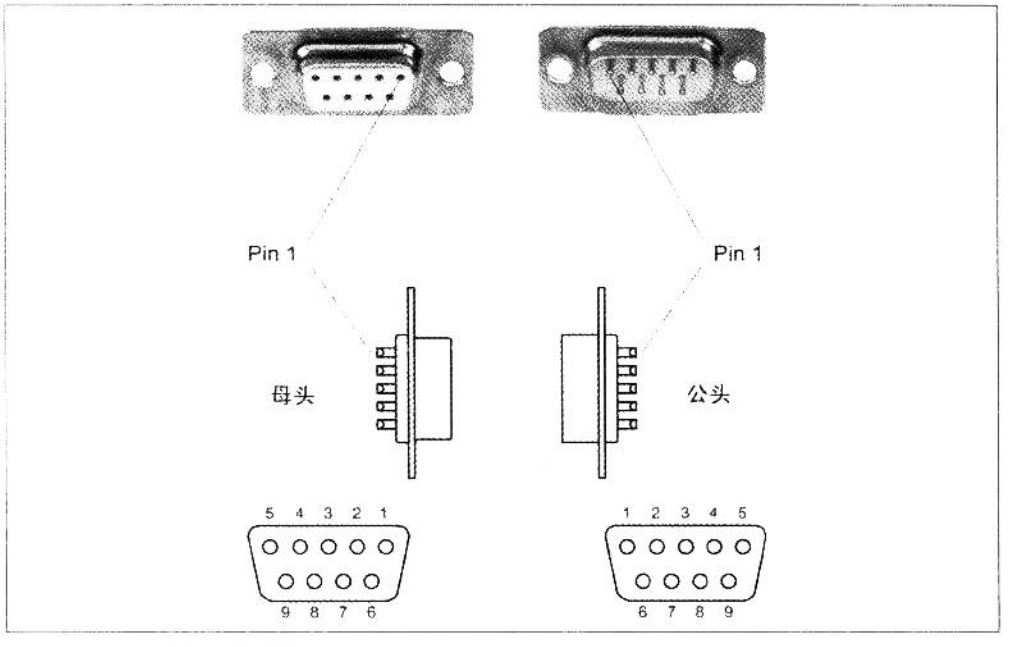


图7-7：DB-9的针和孔的编号

USB 连接器

210

USB 标准定义了通用串行总线（Universal Serial Bus，简称 USB）连接器的形状和大小。它们总共有四根线：两根是数据信号线（D+ 和 D−），另外两根为电源线（+5V）和地线。通常，这些接头都是预先压制好的，所以（幸运的是）你不必去组装它们。你所需要知道的是 USB 标准定义了不同类型的连接器，表 7-1 列出了四种基本的 USB 连接器类型。

表7-1：基本的USB连接器类型

211

类型		描述
类型 A		主要用在主机或控制器的 USB 连接器端口。在 PC 上，用于连接键盘、鼠标、绘图板、数码相机、打印机和集线器等设备
类型 B		一个方形的连接器，其中一边的角上有用于定位到插座的斜边。这类连接器通常用于诸如集线器和打印机这样的外部设备中

续表

类型		描述
迷你 USB		这种类型常见于消费类数码相机和其他移动设备中
微型 USB		与迷你 USB 很相似，但它更细。微型 USB 连接器设计得比迷你 USB 更耐磨损

212 我们所遇到的仪器上通常都是 A 型和 B 型的 USB 接口。迷你 USB 和微型 USB 连接器在设备中并不常见。在标准之外，消费电子产品的制造商已经知道如何适应他们特有产品的小型化和微型化。即使在今天，许多流行的 MP3 播放器和移动电话上仍使用古怪的接口，而不用标准的 USB 线。

圆形连接器

圆形连接器在我们周围到处都有，尽管它并不总是以这个名字出现。电视机或者立体声音响背面的“phono”型连接器就是单回路圆形连接器的例子。用在屏蔽电缆上的 BNC 连接器是另一种经常出现在测试设备和无线电装置上的圆形连接器。在专业的音频设备中，我们会发现用于麦克风输入、跳线和低压电源连接器的 1、2 或 4 路（引脚）圆形连接器。耳机或者耳塞上的 2.5mm 和 3.5mm 立体声插头也是圆形连接器，但是这些通常被称为插入到“插孔”的“插头”。

在本书中，术语“圆形连接器”是指特定类型的连接器，如图 7-8 所示。

图7-8：MIL-型圆形连接器

如图 7-8 所示的连接器被称为“MIL- 型”[译注 1] 连接器。这是由于这些类型的连接器最初用于军事和航空航天应用。因此，它们都非常牢固，并可以忍受恶劣的环境。有些类型完

译注 1：美国军用规格的英文缩写是 MIL。

全由塑料制成，其价格便宜，但仍相对牢固。还有其他类型的连接器则相当昂贵，它们是按订单生产的厂商用航空级材料定制组装的（超过 200 美元的军用标准连接器也不少见）。这两种类型的引脚从两个到上百个不等。

图 7-9 展示了 AN/ARC-220。这是美国军方使用的移动无线电系统，该系统通过各种圆形连接器接入各类航空电子及电力系统。 213

虽然你可能从未在野外遇到过 AN/ARC-220，但你可能遇到的那些看起来坚固耐用的仪器都是类似这种的。

图7-9：AN/ARC-220 电台（图片来源：美国军方）

制作 MIL 型圆形连接器，涉及引脚或插座的锡杯或焊接线，一般要使用特殊的（通常比较昂贵）引脚和插座的两端卷边专用工具。一些低成本的多芯圆形连接器使用冲压形成的针脚和插座，而不紧固件，尽管它们不是很牢固可靠，但是相当便宜。

接线端子

接线端子很古老（从技术时间意义上说），它们出现在 20 世纪早期，随后开始普遍使用，并取代了 19 世纪晚期的螺纹接线柱和夹子连接器。第二次世界大战提供了开发更强大和可靠的连接器的主要诱因，然而创新的步伐并没有自那时起放慢。

“接线端子”指的是各种不同样式的终端设备。如图 7-10 所示的一个栅栏式接线端子，使用了螺钉压住一根线或者卷紧凸耳式连接器。

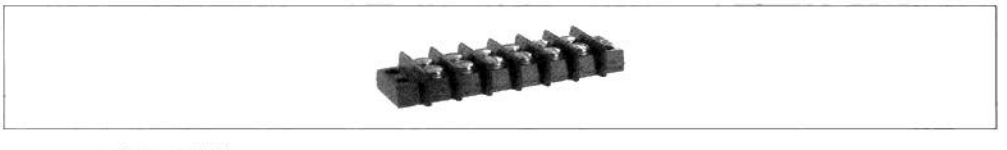


图7-10：栅栏式接线端子

尽管我们可以简单地将裸线绕在螺钉头上，但这不是一个最好的方法。该线也可能滑出去，特别是在多股的情况下。当使用裸线时，单导体铜线效果最好的是栅栏式接线端子。然而，由于螺钉头导致一个固定的压力点集中到弯曲的线上，这导致了线有可能断掉。片式连接器是将线连接到栅栏式接线端子的最佳方式。一个典型的适用于栅栏式接线端子的片式连接器如图 7-11 所示。你也可以在图 7-1 所示的配件中发现它们。

这种类型的连接器确实需要使用一个压接工具，但这些常见的工具很容易在任何货品丰富的五金商店或汽车零部件供应商处获得。

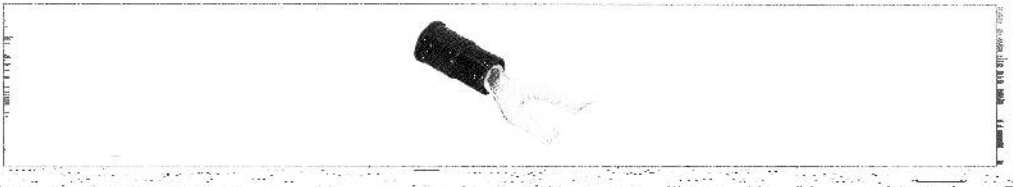


图7-11：压接式铲片

PCB 板载型是一种较为普遍用于仪器设备上的紧凑型接线端子，如图 7-12 所示。这些都设计成可以直接焊接到电路板上，并且提供了方便和可靠的接口连接电路。

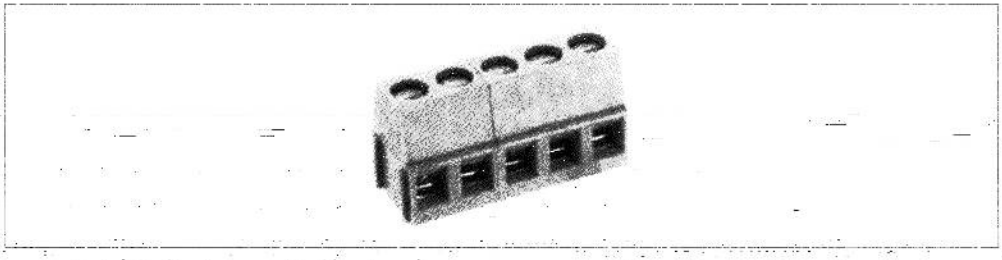


图7-12：典型的PCB板载接线端子

这类紧凑型连接器在有意用在 PC 上的仪器设备中很常见，如稍后我们会讲到的基于 USB 接口的设备。另一种更常见的是在 PC 上插一块接口卡，通过一条特殊的多芯电缆线连接到带有紧凑型接线端子的模块，如图 7-13 所示。

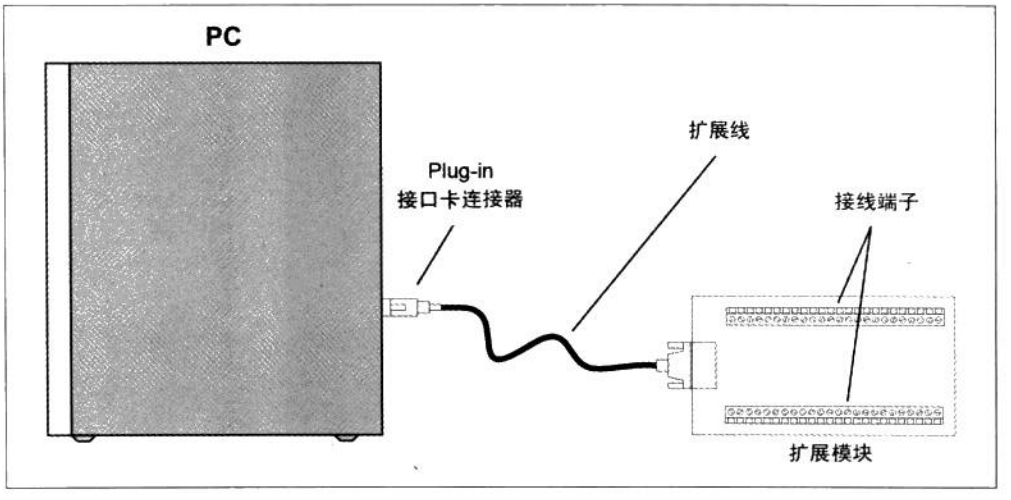


图7-13：PC 接口卡扩展模块实例

接线到一个紧凑型接线端子只需要将导线的顶端剥掉 1/4 ～ 3/16 英寸的绝缘层，将裸线插入到螺丝孔下面的洞中，然后拧紧螺钉，直到它牢牢地夹住导线，不需要焊接或者压接。其缺点是接线端子要比其他一些类型的连接器多占用空间，并且它们并不适用于那些首要考虑耐用性的场合。

接线

在第一次就正确地接线到连接器可以节省大量的时间，并且继续做下去可能还会节省金钱。正如到现在你可能了解到的那样，连接导线到除了接线端子以外的连接器的两个主要手段是焊接和压接，对于接线端子，既可以将裸线插入到端子位置，也可以将压接式铲片固定在终端螺钉下。无论哪种类型的连接器，这些都是正确的（和不正确的）将导线连接到连接器的方式，让我们在这里看看这几种方式。

焊接

如果你是一个焊接新手，在实际处理一个连接器之前做一些练习可能是个好主意。花一点时间用在废电线和旧的电路板上会在将来得到好处。美国政府出版物是焊接技术和标准的一个极好的信息来源。例如，美国航空航天局的标准 8739.3 介绍了焊料连接，并且包含了丰富的有用的信息。

专为焊接而设计的连接器有一个特殊的功能，称为“焊杯”，它在连接导线的针脚的一端加工成型。图 7-14 显示了美国航空航天局（NASA）如何定义一个良好的焊杯连接。

216

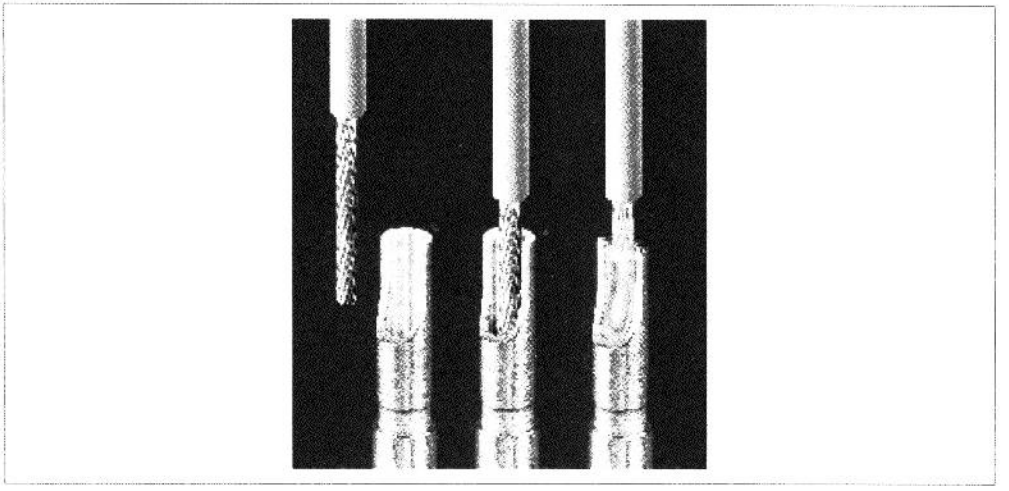


图7-14：焊杯连接（图片来源：NASA）

压接

压接型连接器要比焊杯连接更容易组装，并且为一些应用提供了一个更牢固的连接，但只有根据制造商的规范使用正确的工具才行。图 7-15 显示了一个为圆形连接器加工的压针。一旦导线从背面用特殊工具压入插到连接器上的针脚，它就和连接器主体锁定到位。可以使用另外的特殊工具移除。当然，每个针脚在母头处有一个与之匹配的插座。而每个压接针也需要插入连接器。压接针脚和插座需要为特定型号和尺寸的连接器定制独特的工具。MIL 型连接器的压接工具的成本高达 500 美元(或更多)。其他类型的压接工具，如那些用于 DB 型连接器或低成本的矩形连接器，由 Molex 或 Hirose 生产，费用通常并不昂贵，但也不便宜。

217

图7-15：MIL型压接针

压接针脚和插座的压接部分都有内在的应力消除设计。一个压接式的针脚或插座不仅“咬”到裸线，而且还有第二个区域，靠近针脚或者插座的开口端，紧紧夹住导线的绝缘层。当正确组装，并且和相应的后壳接合在一起时，压接式圆形或 DB 型连接器是非常坚固的——这也就是为什么你会在飞机、航天器、军用装置和恶劣工业环境下大量发现它们的原因。

接线注意事项

在接线到连接器时，必须注意以下几点：

- 作为一个普遍原则，当使用螺钉型接线端子时，不要在绞线的终点上锡。这似乎是个好主意，但是它导致了金属疲劳断裂的情况发生。让导线裸露，把它扭紧，并且让接线端子的螺钉去固定住。
- 你一定要避免焊接一根导线到原本并不需要焊接的连接器。换句话说，如果连接器使用的是插入针脚和为压接工具设计的插座，企图焊接导线或插座往往会导致连接发生机械脆弱。也可能变成这样的情况，尽管焊接针脚或者插座可以插入到一个连接器中，但是任何焊接的毛刺部分可能会挂在两边卡住的位置，那样接下去后，就不可再移除它，只能作废。
- 在工作中始终使用正确的导线。不要使用粗细不合格的导线，也不要尝试将超出规格的导线硬塞进一个接线端子，或者使用较小规格的压片。同样的道理，如果在连接器上设计了锡杯，比如 18 ~ 22 号线，那么试图用 14 号线焊接，就可能导致导线没有完全进入锡杯内，并且这样的连接可能会导致相邻的针脚短路。
- 保持东西整洁。尽可能用尼龙绳或塑料线将相关信号的导线捆扎在一起。如果有大量的线缆束，给它们贴上标签可以节省时间，而且当将来你沿着电缆弄清楚每根导线应该连接到哪里时，能很快知道。
- 请务必在电缆末端使用连接器外壳。这同样适用于圆形和 DB 型连接器。后壳不仅可以保护接线不受损害，它也提供了必要的应变消除。在实验室或商店里经常可以看到没有外壳的 DB 型连接器。虽然不组装外壳可以省去一些力气，但是在频繁使用的情况下，连接器的接线很可能持续不了多久。 218

连接器失效

一个连接器可能会失效的原因有很多，它们都是这样或那样的机械故障。不良的焊接连接、折断或弯曲的针脚、接线端子的螺钉松动，以及导线断裂只是一些可能会导致失效的途径。

正如在本章开始处所说的那样，松动或连接不正确的连接器是问题的主要来源。如果一个连接器设计成使用螺钉固定，那么就拧紧它。如果一个连接器设计成使用螺纹或卡口式外壳锁定到位，那么就用一切办法锁住它。有些连接器，如 RCA 音频插头或 USB 接口，设计成没有任何附加硬件在物理上确保它们插牢，但只是插进去就表示插牢，这是一个增加神秘间歇性失效体验的很好方式。

如果一个连接器传输的是低电平信号，尤其是在高阻抗电路中，腐蚀或污垢会导致连接信号受到噪声干扰，或导致故障间歇性神秘出现和消失。腐蚀可能是来源于手指尖上的油性或酸性物质。使用非抗恶劣环境的连接器也可能导致这个问题。

最后，你应该知道，连接器通常都有一定数量的“插拔次数”，超过这个次数后，它就会变得破损和连接变得不可靠，以至于无法忍受。每次连接都是“配对”到其匹配的部分，这计为一次插拔次数。一些廉价的 DB 型连接器可能只有 50 次左右的插拔次数（如果真是很便宜的连接器，可能还更少）。其他类型，如 USB 连接器，在明显的磨损问题出现之前，它可能有超过 5000 次的插拔。

串行接口

在我们接触到的仪器设备接口中，有三种最常见的串行接口，它们分别是 RS-232、RS-485 和 USB。RS-232 和 RS-485 标准最初是由电子工业联盟（Electronic Industries Alliance，简称 EIA）指定的推荐标准（Recommended Standard，简称 RS）。当维护组织将其名称前缀改变时，先改成了 EIA-232，最后改成了 FIA-232。该标准目前由电信行业协会（Telecommunications Industry Association）修改和维护。但是由于 RS 术语在电子工程和电信的历史中是如此的根深蒂固，因此，一直沿用了下来。本书将使用 RS-232 和 RS-485 来称呼这些标准。

219 过去的十年里，USB 接口变得十分普遍，几乎在每台 PC 上，USB 已经从很大程度上取代了 RS-232 接口。只有桌面或者机架式 PC 仍然保留着串口（甚至有些薄的机架式 PC 上也取消了串口）。最新型号的笔记本电脑和所谓的上网本现在都只有 USB 接口。

在特定的设备上有各种不同类型的特殊用途和高可靠性的串行接口。其中包括 CAN（Controller Area Network）、FieldBus 和 Profibus。另外，在军用设备和一些航空电子设备上可以找到 MIL-STD-1553 串行总线。这些专门的接口在有限的领域之外都是不常见的，因此，我们将不去讨论它。但是，串行通信背后的基本概念适用于所有的串行接口，RS-485 接口是一些工业级接口所基于的底层框架。

半双工和全双工

术语“半双工”和“全双工”是指一对设备（或人）通过某种通道通信时的数据传输模式。在半双工系统中，任何一方设备轮流作为发送方或者接收方，双方不能同时通信。一个常见的半双工的例子就是双向无线电设备（又称对讲机），任何一人在开始通话前都要等另外一人结束通话。在这里，术语“10-4”、“over”、“roger”和“over and out”发挥了作用：它们表明位于通信通道两端的人知道何时轮到他们开始说话。否则，他们互相抢着说话，没有人能够明白他们说些什么。在串行接口中，半双工通道有一组电缆线。两端设备之间的数据流方向是由设备采用的通信协议所确定的。

在全双工系统中有两个通道，一旦准备好发送数据，每个通道的发送端都可以发送。例如，一个RS-232连接只有一条电缆，但是其中有两条导线，每条有一个数据发送方向（事实上，有两条以上的导线，但我们将在稍后介绍）。每条导线是其自己的通信通道。以太网是另外一种全双工的接口类型，但是它也可以同样工作在半双工模式。另外，RS-485往往作为半双工接口实现。一个基本的RS-485接口只有一对导线支撑单个接口通道，在数据交换期间，两端的设备必须轮流发送和接收。

RS-232/EIA-232

RS-232是一种基于电压的接口，也就是说，逻辑0和逻辑1之间的区别是由当前信号线的电压值所决定的。该标准不指定数据的编码，只定义了硬件接口，但它和ASCII字符编码标准（在第12章中详细讨论）密切相关，以至于这两个标准已经密不可分。要注意，RS-232具有一定的局限性。例如，它（通常）不能在PC或其他系统的同一个“端口”上连接多个设备。换句话说，它是一个端到端的接口，如图7-16所示。由于使用电压的波动表示信号，以及RS-232往往容易受到周围环境和噪声的干扰，因此，它还具有线缆长度和传输速度的限制。

220

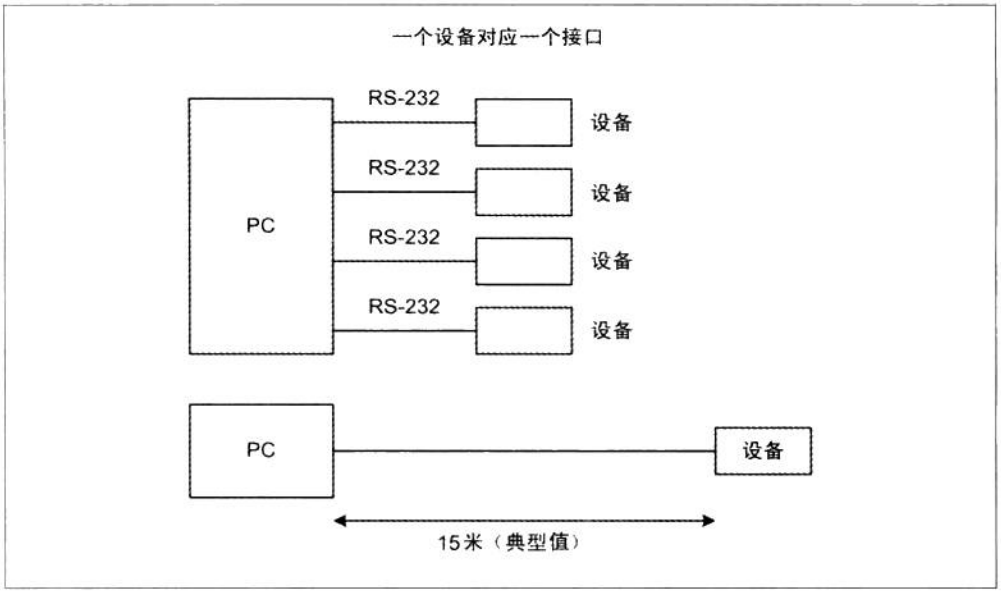


图7-16：RS-232设备连接

RS-232 数据格式

正如我们刚才所提到的，RS-232 是基于电压的接口。它使用一个电压值表示逻辑真（1），用另一个代表逻辑假（0）。图 7-17 显示了它是如何工作的。

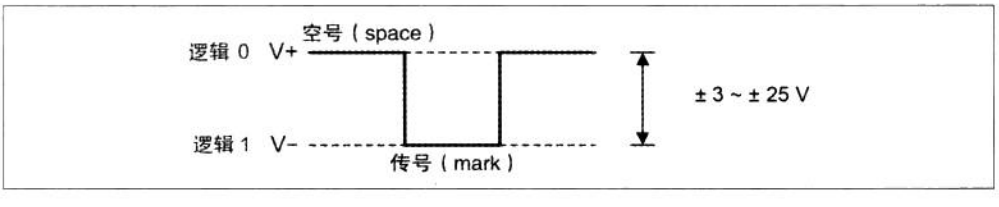


图7-17：RS-232信号电平

请注意，RS-232 数据信号采用负逻辑。也就是说，一个逻辑真（1）是一个负电平，而逻辑假（0）是一个正电平。

221 当使用 ASCII 编码时，一个 RS-232 字符由一个起始位（传号）紧跟 5 ~ 9 位数据位、一个可选的校验位，以及一个或两个停止位（空号）组成。图 7-18 分别显示了 8 位数据位（无奇偶校验位）和 1 位停止位，以及 7 位数据位（纯 ASCII）、1 位偶校验位和 1 位停止位的数据格式。在这两种情况下，实际接收或者发送的字符或字节是 10 位。每个数据单位从起始位开始算起，到结束位（如果有的话），被称为“帧”。

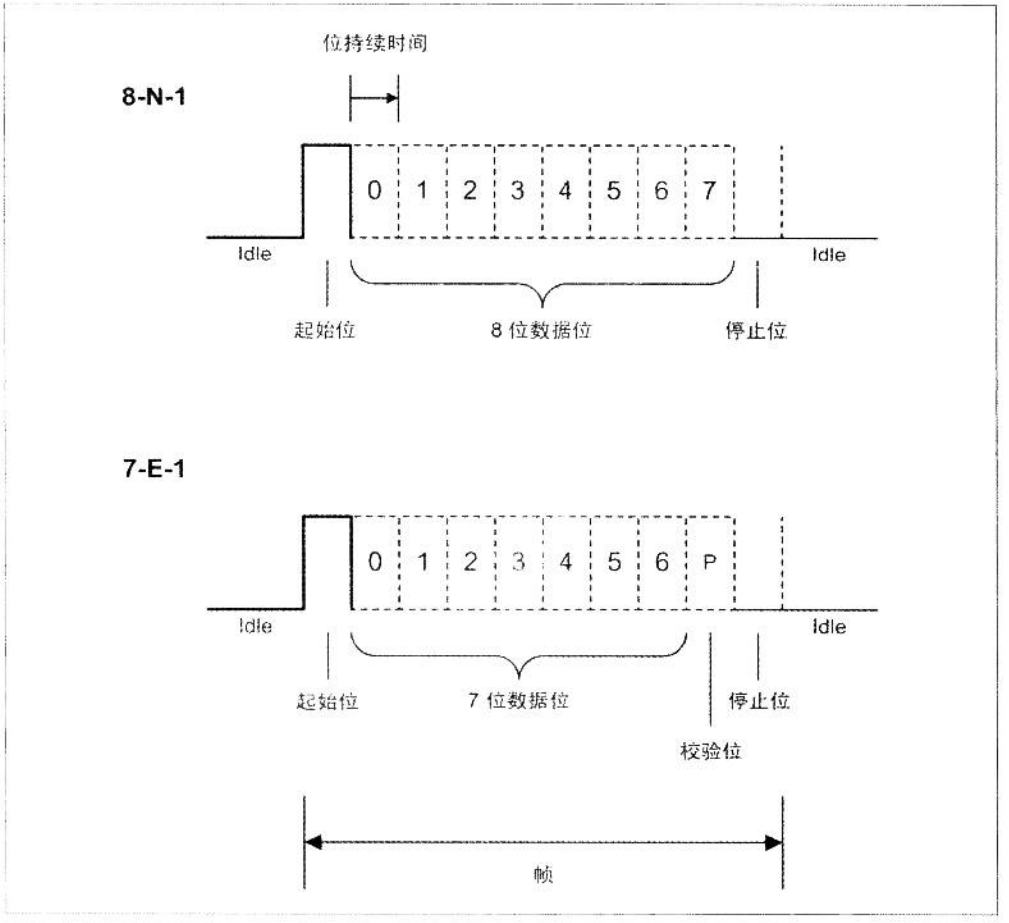


图7-18：RS-232数据格式

为了使一个特定的格式能正常工作，通信通道的两端必须一开始就配置正确。试图将一个配置为 8-N-1 的设备连接到一个配置为 7-E-1 的设备将不能正常工作，即使是两端发送和接收的每帧数据都是 10 位。如果那样，结果就是在 8 位端收到很多错误的数据，而在 7 位端产生大量的奇偶校验错误。

每秒发送的比特数称为波特（baud），用于衡量一个 RS-232 接口的速度或者数据传输速率。由于一帧中的 10 个位中至少有 2 位表示开始和结束位，这意味着在 9600 波特速度运行下的串行接口每秒发送和接收的字节或者字符不会超过 1200 个(9600/8)。也就是说，帧中只有 80% 包含的是真正的数据。在这种情况下，真正的最高速率正好为每秒 960 个字符。通常情况下，如果你知道每个数据帧的位数，用这个数去除波特率，就能得到每秒字符的有效速率（CPS）。

222

RS-232 信号

RS-232 接口用一组信号在两个设备之间传输数据和握手信号。这些信号大多最初定义的目标应用是将一个终端或者主机连接到调制解调器。外部的调制解调器日渐稀少，但是大多数的 RS-232 接口仍然保持着各种信号线，如图 7-19 所示。注意，这是 DB-9 型连接器，当使用 DB-25 型连接器时，有可能需要实现其他几根信号线，尽管它们很少用到。

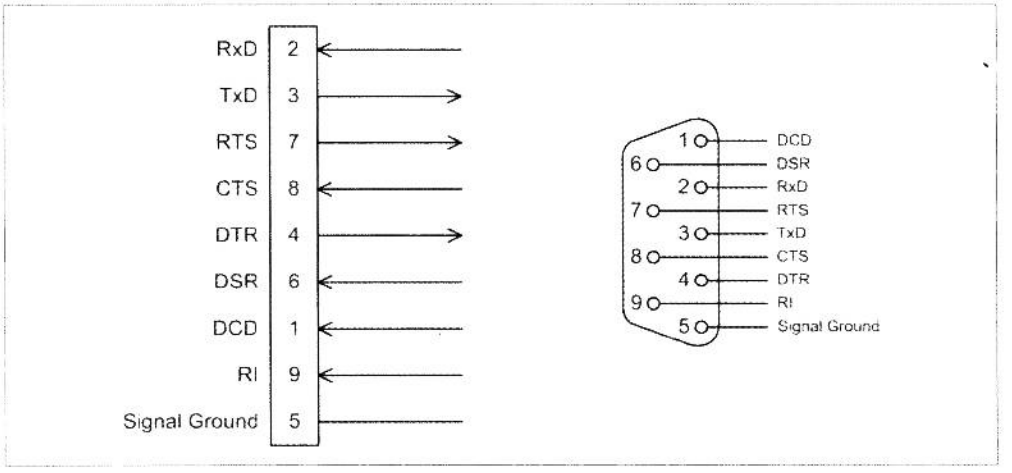


图7-19：RS-232信号

图 7-19 显示的基础 RS-232 信号是实现带有硬件握手的完整 RS-232 接口所必需的全部。表 7-2 列出了这些信号的定义。

223

表7-2：RS-232 DB-9引脚定义

信号	定义
RxD	接收数据
TxD	发送数据
RTS	请求发送
CTS	允许发送
DTR	数据终端准备完成
DSR	数据准备完成
DCD	载波检测
RI	振铃提示

RS-232 是全双工接口，这意味着数据可以同时双向传输。它同时也是异步的，所有数据的时钟同步是来自传入的数据流本身，而不是来自接口中的一条额外时钟信号线（当然，这也有一个例外，RS-232 可以作为一个同步接口实现，但这很少见）。RS-232 也可以工

作在半双工模式下。更多的信息请参阅“半双工和全双工”。

在许多情况下，只有 RxD（接收）和 TxD（发送）数据线是真正需要的，并且一些仪器确实是这样连线的。大多数带有 RS-232 接口的仪器通过正确的电缆连接 PC 都能很好地工作（特别是仪器附带的电缆），没有修改过的接口连接线应该是必需的。

DTE 和 DCE

在使用 RS-232 接口时，无疑会遇到首字母缩略词 DTE（Data Terminal Equipment，数据终端设备）和 DCE（Data Communications Equipment，数据通信设备）。这些术语来自大型主机和声耦合调制解调器时代，用来定义串行通信的端点和连接设备。该术语最初是由 IBM 引入的，用于描述其主机产品的通信设备和协议。

在 RS-232 的场合中，DTE 是一个串行数据通信信道的终端设备。DTE 中的“终端”并非是指带有卷子的设备和键盘（电传打字机终端或 TTY），或者是一个 CRT 显示器和键盘（老式计算机终端或曾经的“玻璃 TTY”），它的字面意思是“终点”。图 7-20 图形化显示了这个设置。

从另一个角度看，可分为“数据接收器”和“数据发送源”。数据接收器接收数据源发来的数据。信道的两端（两个 DTE 设备）都可以作为接收器或者发送源。DCE 设备通过某些类型的通信介质为两端提供信道。对于一个使用调制解调器的系统来说，这通常就是一条电话线。 224

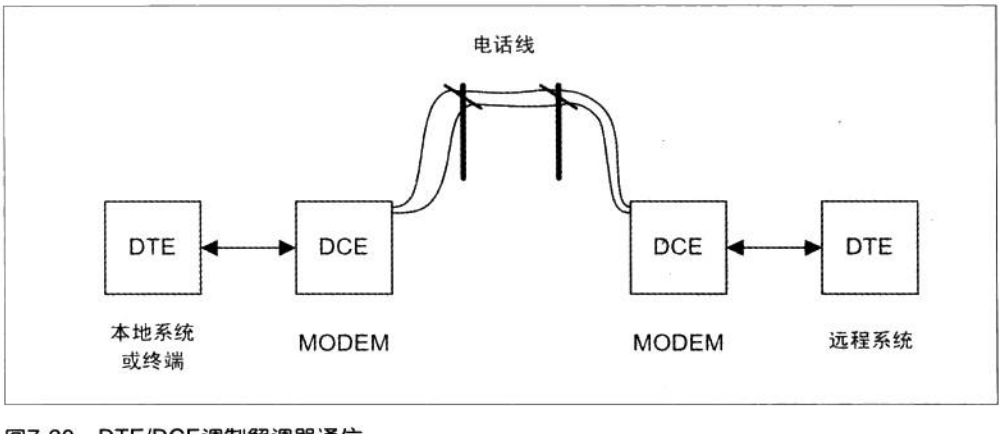


图7-20：DTE/DCE调制解调器通信

如今，调制解调器正在成为一个濒危物种，尽管它们仍然在美国的一些偏远地区和世界各地缺乏高速互联网服务的地方用于数据通信。然而连线使用 RS-232 电缆和连接器仍

然保留了这一传统，这也就是为什么要了解它以便能正确使用 RS-232 连接设备的重要原因。

上一节中所描述的信号线是参照 DTE 命名的。换句话说，在一个 DTE 设备上，TxD 是数据源或输出。而在一个 DCE 设备上，它就是接收器或从 DTE 的 TxD 输入。这同样也适用于 RxD 线。实际上，DCE 的数据源和接收器在功能上与 DTE 的 TxD 和 RxD 线正好相反，即使它们都有相同的名称。这看上去有些困惑，但其结果就是，当一个 DTE 和 DCE 连接时，其接口之间的连线是针对针的（1 针对 1 针、2 针对 2 针，其他以此类推）。

如果你需要连接的两个设备碰巧都是 DTE，你需要使用所谓的交叉线，或者如果你不需要握手信号线，那么只要一条零调制解调器电缆即可，图 7-21 显示了在 DTE 到 DTE 接口中 TxD 和 RxD 线是如何交叉连接的。它没有显示握手信号线是如何连接的。

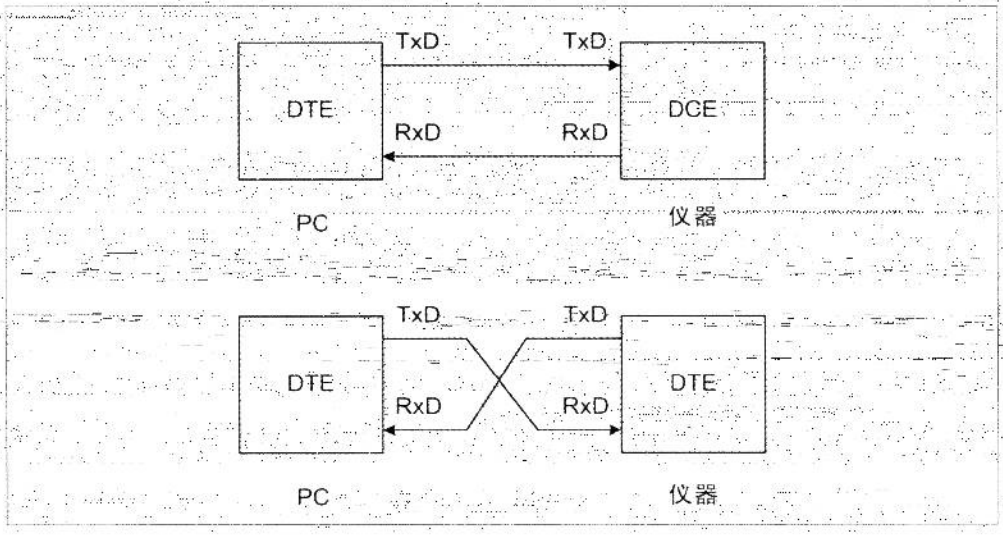


图7-21：交叉或零调制解调器接口

大多数带有一个串口的设备都是作为 DTE 设备连接的，但有些可以配置成 DCE。这可以使用跳线、PCB 上的微型开关、前置面板控制器，甚至可以通过软件配置。PC 内置的串口一般是作为一个 DTE 实现的。

225 与低速接口设备通信，或者通信协议有严格的命令和相应的格式时，你可能并不需要 RS-232 的握手信号线。在这种情况下，你可以使用一条现成的零调制解调器电缆或一个零调制解调器适配器。图 7-22 显示了这样的适配器。

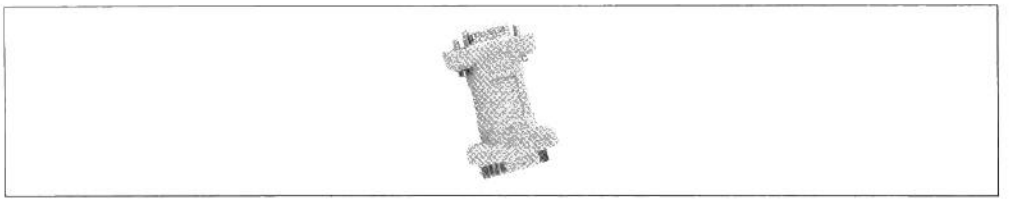


图7-22：DB-9零调制解调器适配器

RS-485/EIA-485

RS-485 常见于仪器控制接口和工业环境中。像它的前身 RS-422 一样，它具有高水平的抗噪声能力，并且在某些情况下可以将电缆长度延伸到 1200m。RS-485 也比 RS-232 更快。它在 10m 电缆连接时可以支持高达 35Mbit/s 的速率，在 1200m 时也可以达到 100kbit/s。

RS-485 信号

226

RS-485 拥有使用差分信号的能力。通过两条配对的电线在两个方向上传输数据，而不是使用一条专用线在某一特定的方向传输，它不能同时在两个方向上传输。差分接口的两条线的极性总是相反的，两条线之间相对的变化状态表示一个逻辑值从“1”到“0”，反之亦然。图 7-23 显示了一个典型的包含一个起始位和停止位的异步串行数据。作为对比，它也显示了 RS-485 信号对应的数字 TTL 输入。

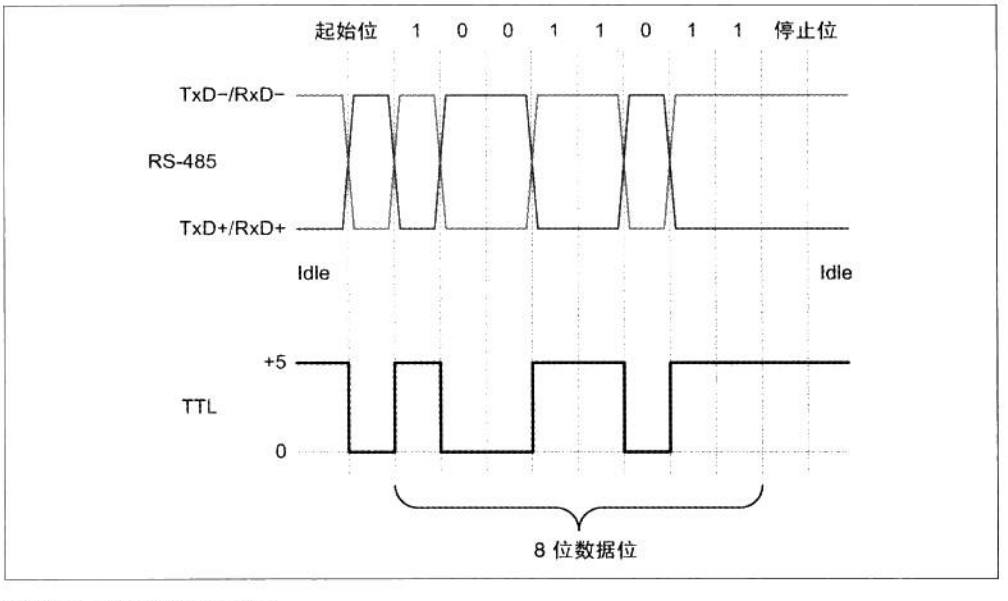


图7-23：RS-485信号电平

请注意，在完成一字节的数据传输时，+和－信号总是返回到初始状态。

线路驱动器和接收器

在RS-485接口中，每个连接点使用一对设备，包含一个差分发送器和差分接收器，如图7-24所示。

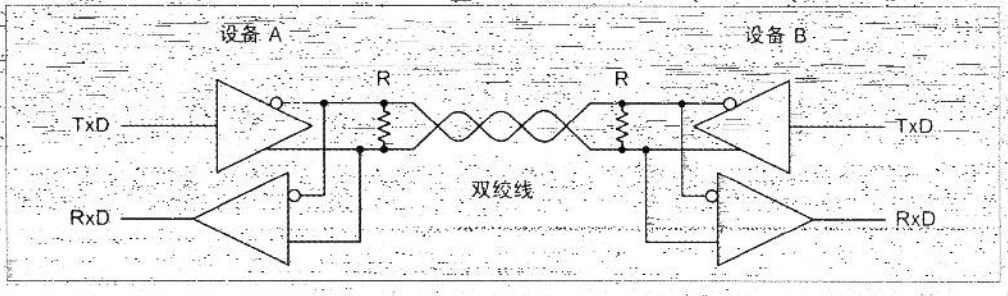


图7-24：两线模式下的RS-485接口驱动器

RS-485可以作为两线半双工接口或双向四线全双工接口实现，但对于许多应用来说，不需要全双工操作。一个四线的设置如图7-25所示。

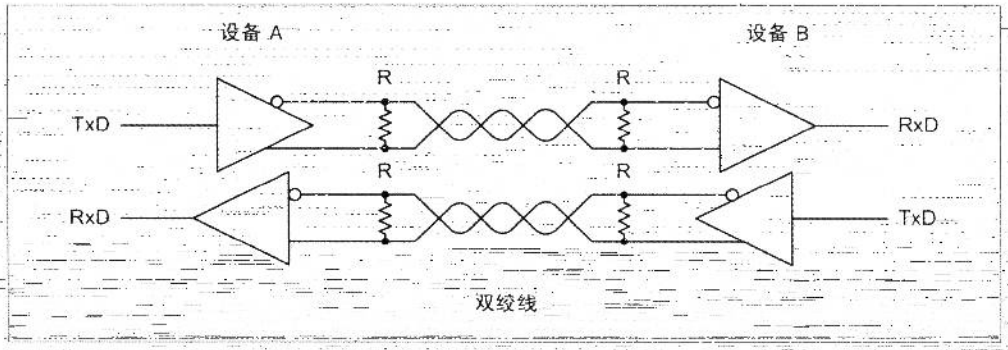


图7-25：四线模式下的RS-485接口驱动器

RS-485多分支

RS-485还允许多台设备或者节点连接到串行“总线”上，即所谓的多分支配置。这就是如图7-26所示的那样。要实现这个，RS-485驱动器的发送器（输出）部分必须能够置于Hi-Z或高阻态模式。当RS-485在两线模式下连接时，这也是必不可少的。

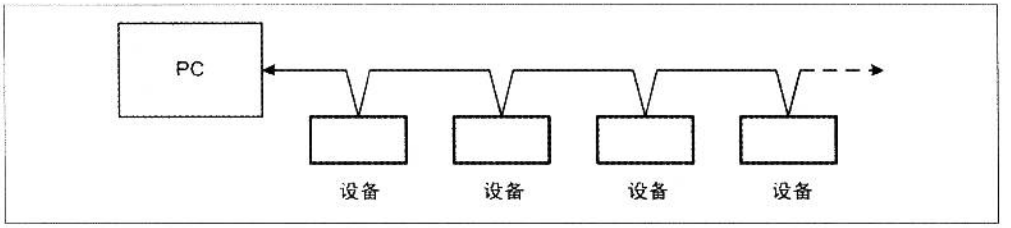


图7-26：RS-485多分支

这样做是由于如果发送器总是有效地连接到接口的话，那么它可能和线上某处的其他发送器产生冲突。在图 7-27 中，你可以看到驱动器是如何轮流成为“发言者”或数据源的，当线路处在半双工模式下时，取决于数据在哪个方向上移动。实际上没必要断开接收器，因此，在任何时候，它们都能收到接口上的数据。 228

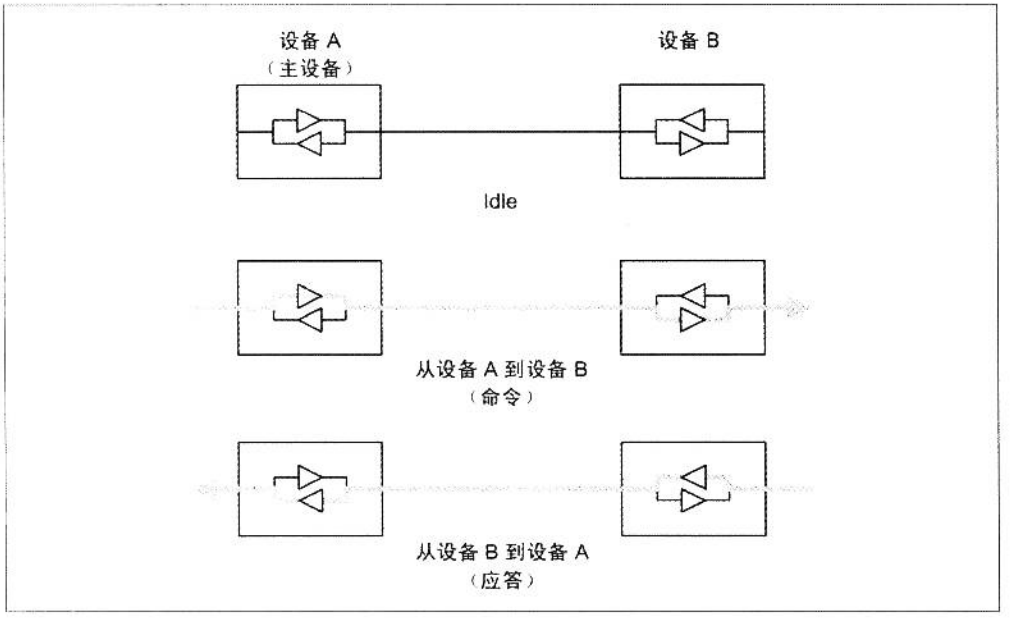


图7-27：RS-485半双工数据流

尽管在一个 RS-485 网络中可以有多台控制器，但是在一个典型的多分支配置中，只有一台设备被指定为 RS-485 的总线控制器，而其他所有的设备都从属于它。默认的模式是控制器发送，从属设备接收。当控制器特别要求从属设备发送数据时，它们的位置交换。当这种情况发生时，称其为“反转”。图 7-28 显示了数据在两个 RS-485 双线配置的设备之间如何传输。

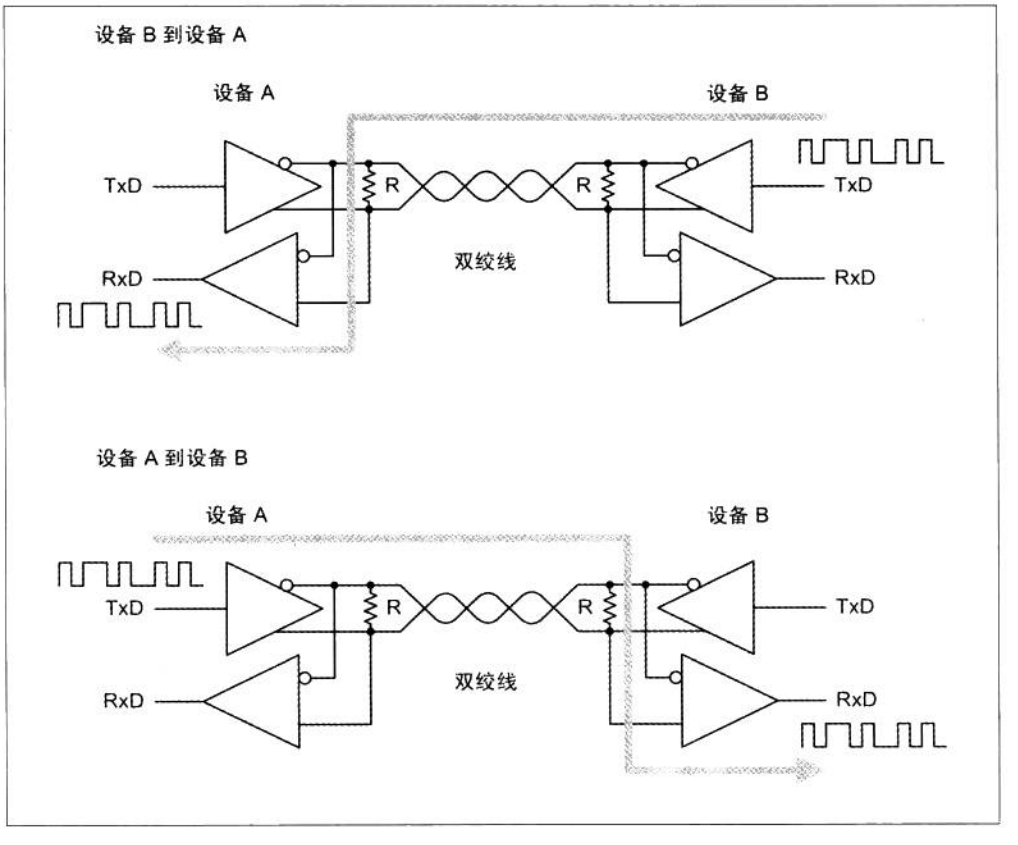


图7-28：RS-485半双工操作

在使用一个半双工 RS-485 接口时，必须考虑接口执行一个反转所需要的时间。即使一个接口可以检测到反转并且自动从发送状态转成接收状态，但是仍然需要少量的时间。有些 RS-232 到 RS-485 的转换器能够通过 RS-232 接口的 RTS 线很好地执行反转。图 7-29 说明了这是如何工作的。

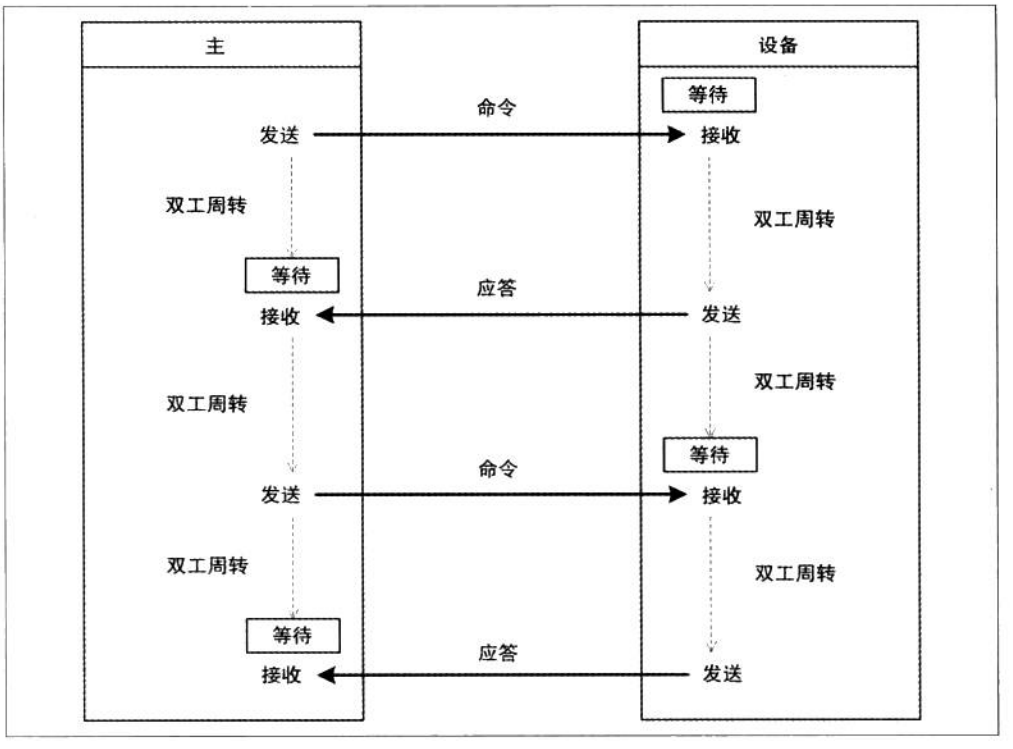


图7-29：RS-485命令-应答顺序

请注意，图 7-29 中任何设备处于接收模式时，当该设备收听从信道的另外一端发来的应答时，都有一个确定的等待时间。

RS-232 和 RS-485

表 7-3 包含了 RS-232 和 RS-485 的一些电气特性对比。

表7-3：RS-232和RS-485

特性	RS-232	RS-485
差分	否	是
最大设备数量	1	32
最大接收器数量	1	32
运行模式	半双工和全双工	半双工或全双工
网络拓扑	点对点	多分支
最长距离	15m	1200m
12m 时最高速率	20kbit/s	35Mbit/s
1200m 时最高速率	n/a	100kbit/s

230 通常来说，RS-232 适用于速率不是很高，且电缆长度在 5m 左右的设备上。许多外部仪器设备使用很短（2 ～ 10 个字符）的命令，返回同样短的应答，有时候执行命令的时间大大超过命令从主机系统发送到仪器所需要的时间。对于这些类型的设备，一个工作在 9600 波特率的 RS-232 接口是完全可以接收的。在其他情况下，例如，有很多传感器或控制器分布在一个大系统的单条通信总线上的时候，RS-485 可以很好地工作。很多制造商为这类应用出售这样的设备，我们将在第 14 章真实的例子中做详细研究。

231 USB

通用串行总线(USB)是另一种形式的半双工异步串行接口。它在某些方面和RS-485类似，数据作为USB电缆中的一对数据线上的差分信号进行传输，其电缆还包含电源线和地线。另外，一个USB网络只有一个接口能作为控制器（或主机），而其他的都作为从属设备。

图 7-30 显示了一个 USB 网络，其包含 1 个带内部集线器的主机系统、2 个外部集线器和 8 个 USB 设备。

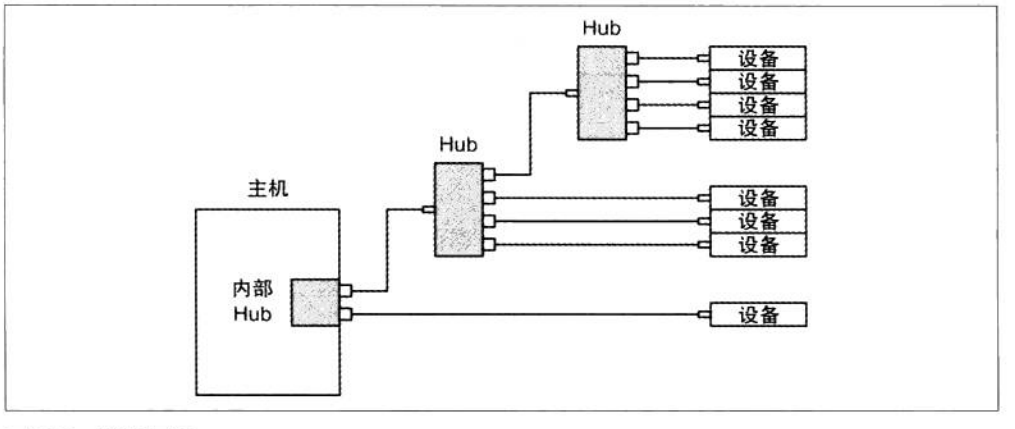


图7-30：USB网络

在大多数情况下，我们并不需要去关心 USB 接口本身的底层细节。Windows 和 Linux 操作系统都包含了底层的驱动程序，以处理主机的琐碎任务，同时提供软件处理与 USB 设备的底层通信。在某些情况下，厂商也会提供一个库模块，用于支持用户编写的应用程序访问。尽管 Windows 比 Linux 更常见，但是一些 Linux 接口驱动也能工作得很好。

USB 类型

USB 标准定义了使用 USB 接口设备的不同类型。表 7-4 中列出了你可能会遇到的一些常见类型。

232

表7-4：常见的USB设备类型

USB 分类	实例
通信	以太网适配器、调制解调器
HID（人机接口设备）	键盘、鼠标等
影像	摄像头、扫描仪
IrDA	红外数据传送器 / 控制器
大容量存储	硬盘、固态硬盘、U 盘
PID（物理接口设备）	力反馈操纵杆
打印机	激光打印机等
智能卡	智能卡阅读器
测试和测量（USBTMC）	测试和测量设备
视频	摄像头

可能你已经很熟悉 HID 和大容量存储类型，这两类设备包括如键盘、鼠标、简单的操纵杆、移动硬盘和 U 盘（也叫闪盘）。HID 类型是比较容易实现的，大多数操作系统都自带通用的 HID 类型驱动程序，因此，使用 HID 类型实现的设备看上去并不像鼠标或者键盘那样常见。

如果 USB 设备使用了一个独特的接口（还是有一个对应的 USB 类型），它由供应商提供必要的接口程序，包括任何操作系统所需的底层驱动。在这种情况下，你通常需要在安装附加设备前先安装驱动程序软件，使得当新设备被检测到时，操作系统能识别出来。

由于我们不会去写任何底层的 USB 驱动软件，因此，我们不会去深入了解这些 USB 类型。这个工作的前提条件是所有需要的驱动软件都已经准备好。

USB 数据传输速率

当 USB 设备工作时，了解接口性能方面的期望值是有好处的。USB 接口设备的一个潜在缺点是速度：与使用基于 PCI 总线接口或专用的独立数据采集系统相比，多数不是特别快。

USB 接口的最大数据传输速率范围从 1.5Mbit/s 到 4Gbit/s，该速率视接口遵循的标准等级而定。USB 标准的版本号定义了 USB 的数据传输速率。也就是说，与 USB 1.1 兼容的设备，在全速模式下，理论上可以达到最高 12Mbit/s 的数据传输速率，而 USB 3.0 兼
233 容设备则有一个最高为 4Gbit/s 的数据传输速率。表 7-5 列出了规格等级和相关的最大数据传输速率。

表7-5：USB版本

版本号	发布时间	最大速率	速率名称	备注 / 附加特性
1.0	1996	1.5Mbit/s	低速	非常有限的行业应用
		12Mbit/s	全速	
1.1	1998	1.5Mbit/s	低速	最早广泛采用的版本
		12Mbit/s	全速	
2.0	2000	480Mbit/s	高速	迷你和微型连接器、电源管理
3.0	2007	4Gbit/s	超速	改良的连接器，向下兼容

USB 3.0 还是相当新的，大多数我们所碰到的 USB 设备都是 1.1 或 2.0 兼容的。你应该知道，即使是所谓的 USB 2.0 高速设备，其达到 480 Mbit/s 速率的可能性也微乎其微。USB 中微控制器接收命令、解码、执行请求的动作和返回应答到主机所花费的时间，比人们期望的数据传输速度要慢得多。另外，主控制器管理通信的能力也比理论上的最大数据传输速率慢。如果主机忙于处理其他任务，它可能无法足够快地为 USB 通道服务，以维持一个很高的数据吞吐量。

USB 设备和集线器的连接方式对通信的响应也起到很大的作用。一个使用 USB 1.1 集线器和设备的 USB 1.1 网络，其最高速率等于网络中最慢的设备速率。USB 2.0 集线器能从高速数据中分离出低速和全速流量。当购买新的 USB 组件时，你应该避免 USB 1.1 集线器，并且坚持要 USB 2.0 的。这样，你就可以避免一个高速的 2.0 设备被 USB 网络中的 1.1 设备拖慢（假设主控制器本身是 USB 2.0 高速兼容的）。

但是速度并非一切，而且速度在许多仪表系统中并不重要，只要能满足基本的响应时间要求即可。由于低成本和易用性的吸引，在下面一节中，我们将介绍各种低成本的仪器设备。

USB 仪器

一个我们希望碰到的 USB 仪器设备的通用版如图 7-31 所示，这个和几个现成的商业设备类似，是为了说明我们期望发现的输入和输出类型。

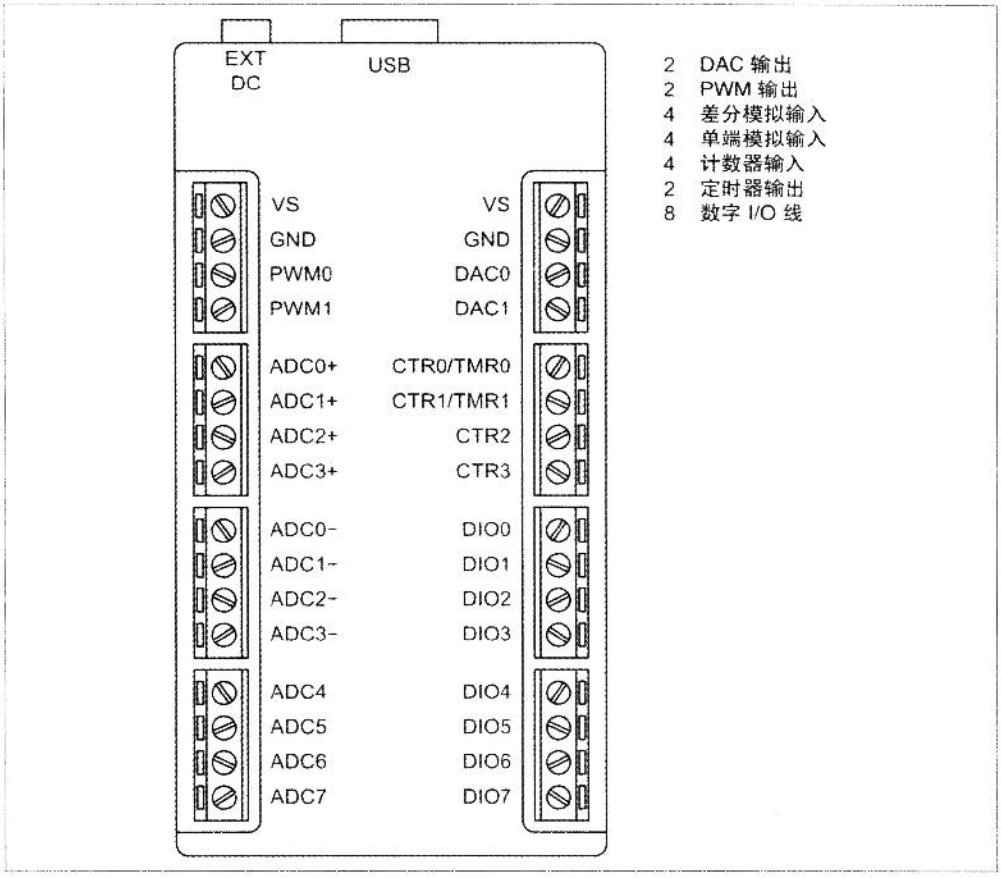


图7-31：通用USB仪器I/O设备

很多USB接口设备还包括一个高级的内部功能和配置选项。例如，在图7-31所示的假想设备中，分离的I/O口可以单独配置为输入或者输出、模拟输入和输出，以及一些计数器和计时器端口。有很多种这类典型的设备。PWM输出应该能接受控制参数，以确定基本的时钟频率和占空比。有些设备甚至可能具有直接通过一个模拟输入控制占空比的能力。

234

现在让我们来看看真正的USB接口设备。图7-32所示的是一种在售的USB仪器设备——LabJack U3。这是一种低成本（约110美元）的部件，其具有可配置的分离I/O、模拟输入和输出通道计时器、计数器，支持SPI、I^2C和异步串行协议。U3采用了USB全速（12Mbit/s）接口。

235

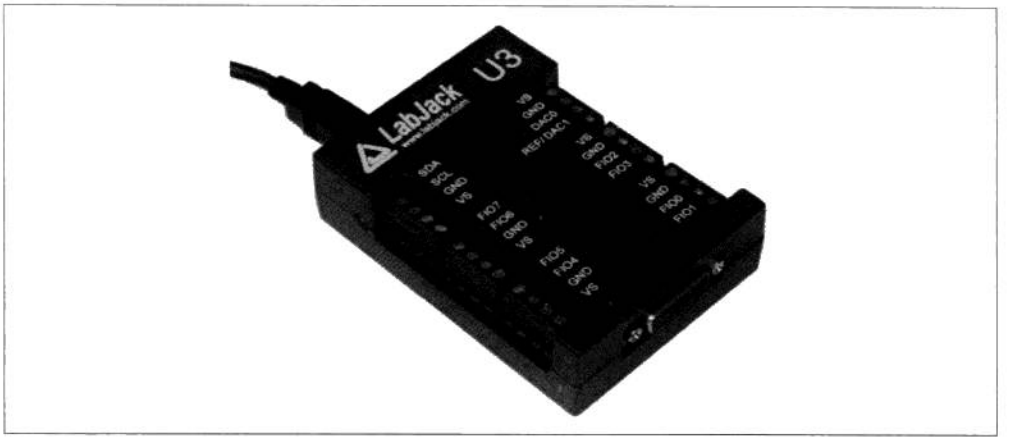


图7-32：LabJack USB接口设备

Windows 虚拟串口

在 Windows 系统下，我们可以实现所谓的虚拟串口（virtual serial port，简称 VSP）或虚拟 COM 口（virtual COM port，简称 VCP）。作为“端口重定向”的一个方面，VSP 已经在相当长的时间里成为 Windows 的一部分。虚拟串口支持其他任何 Windows 串口界面的大多数或全部常见的功能。你可以设置波特率和数据位数，并且读取和写入数据，就像你在使用一个“真正”的串口一样。虚拟串口还可以模拟物理串口所使用的控制线，如 RTS、CTS 等。

虚拟串口的一个主要应用是为物理 USB 端口提供通用和方便的界面，并且让设备能连上去。图 7-33 显示了一个 VSP 到 USB 接口的简图。

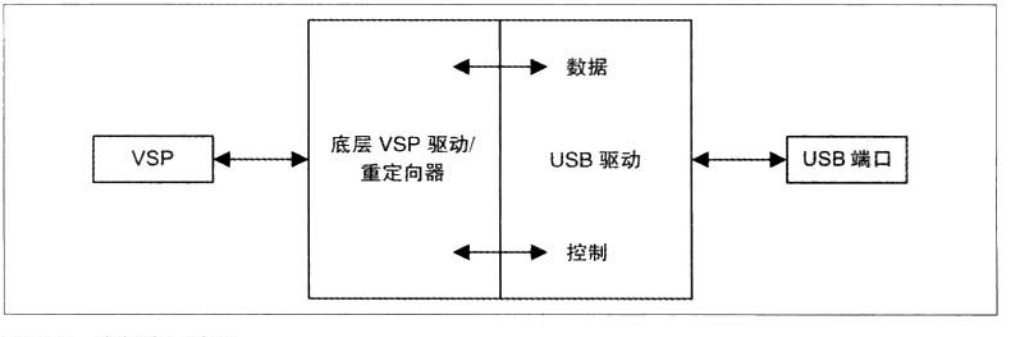


图7-33：虚拟串行接口

236 在某些情况下，虚拟串口作为一个 USB 到串口的转换器使用。USB 到 RS-232 和

RS-485 接口的转换器是现成的，图 7-34 显示的是这样一个将串口仪器链接到 PC 上 USB 端口的设备通用图。FTDI 公司和 Silicon Labs 公司这样的厂商提供的 IC 芯片实现了 USB 转串行接口所必需的全部功能。随着最新的计算机（尤其是笔记本和上网本）都取消了 RS-232 串行接口，很多厂商的产品都将转向此类接口，以代替使用 RS-232 或 RS-485 接口作为主控制器或数据传输接口。在其他情况下，它可以作为一个主连接仪器或设备，在设计中有且只有一个 USB 接口，包含了一个驱动程序，用于设备的接口协议和串行数据之间的转换。我们在后面看到的 USB 到 GPIB 接口就是一个这样的设备。

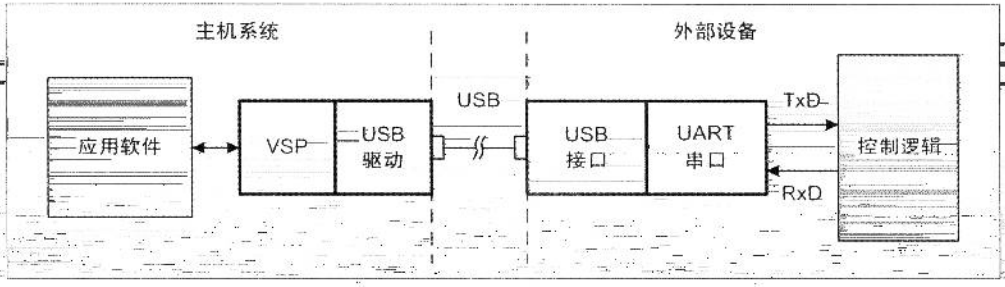


图7-34：USB到串行接口

USB 重定向只是一个虚拟串口和端口重定向的应用程序。一个 Windows 虚拟串口可以像一个普通串行接口重定向 I/O 到另一个端口，甚至一个网络接口那样使用。稍后我们看到的来自 Vyacheslav Frolov 的 *com0com* 包是一个开源工具。*com0com* 使用端口重定向来实现一对虚拟串口的内部连接。默认配置为零调制解调器（但这可以在安装驱动时修改）。换句话说，往一个端口写数据会在另外一个端口上接收到。图 7-35 显示了 *com0com* 如何工作。

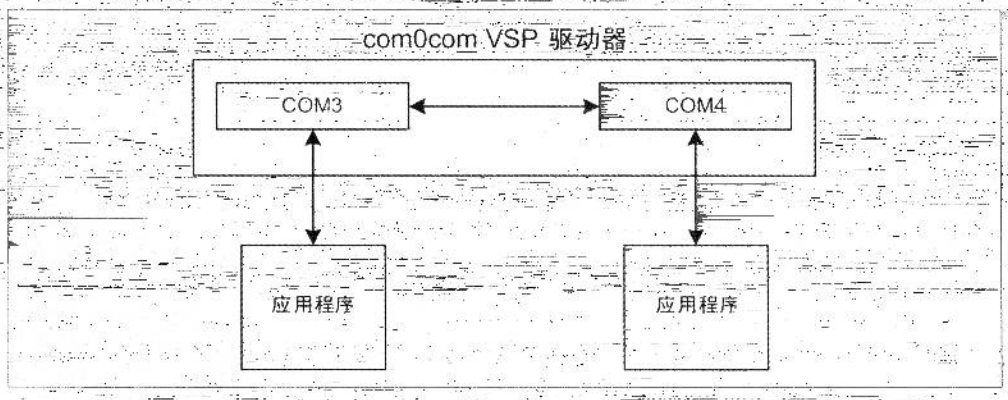


图7-35：com0com VSP工具

使用一个名为 *com2tcp*（包含 *com0com*）的实用工具，可以通过网络连接两台不同计算机上的两对虚拟串行端口，如图 7-36 所示。

一个像 *com0com* 这样的实用工具可以非常方便地实现实时监测和跟踪工具或命令行风格的用户界面，或使用串行接口的模拟仪器。在使用仪器设备时，串行接口重定向是一个强大的功能。

237

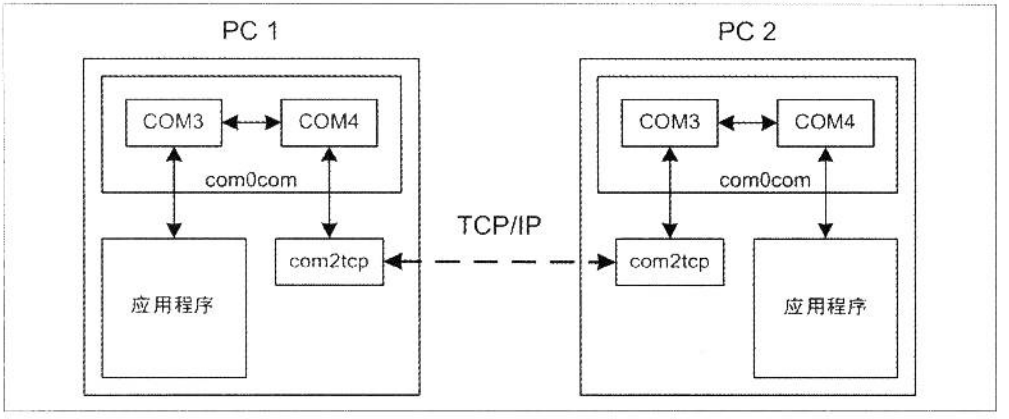


图7-36：TCP/IP上的com0com

238

GPIB/IEEE-488

为了将测试仪器与其他测试设备、HP 打印机、HP 的小型机和海量数据存储设备连接起来，惠普公司在 20 世纪 60 年代后期开发了通用接口总线（The General Purpose Interface Bus，简称 GPIB）。它最早被称为 HP 接口总线（HPIB），1975 年成为 IEEE 标准。

尽管 GPIB（当时称为 HPIB）在 20 世纪 70 年代末的时候只用在一些 HP 计算机设备同大型外部磁盘驱动器和行式打印机的接口上，它也从来没有真正成为数字外设的标准接口。但是，目前在测试和数据采集设备上仍然很常见到。

GPIB/IEEE-488 信号

GPIB 是一种并行接口类型，在一根电缆中包含了数据、命令和接口信号线。GPIB 的寻址允许多达 15 个设备共享一个 8 位并行数据总线。最新版本的标准允许最高到 8MB/s 的数据传输速率。

在内部，GPIB 将 12 条线用于不同的信号，然后将这 12 条线屏蔽并与地线连接。表 7-6 列出了 GPIB 的信号，图 7-37 显示了一个 GPIB 连接器的引脚输出。

表7-6：GPIB信号

信号	引脚	功能
DIO1	1	数据 / 命令
DIO2	2	数据 / 命令
DIO3	3	数据 / 命令
DIO4	4	数据 / 命令
EOI	5	结束或识别
DAV	6	有效数据
NRFD	7	没有读到数据
NDAC	8	没有收到数据
IFC	9	接口清除
SRQ	10	请求服务
ATN	11	注意
Shield	12	线缆屏蔽
DIO5	13	数据 / 命令
DIO6	14	数据 / 命令
DIO7	15	数据 / 命令
DIO8	16	数据 / 命令
REN	17	远端许可
Ground	18	地线
Ground	19	地线
Ground	20	地线
Ground	21	地线
Ground	22	地线
Ground	23	地线
Ground	24	地线

239

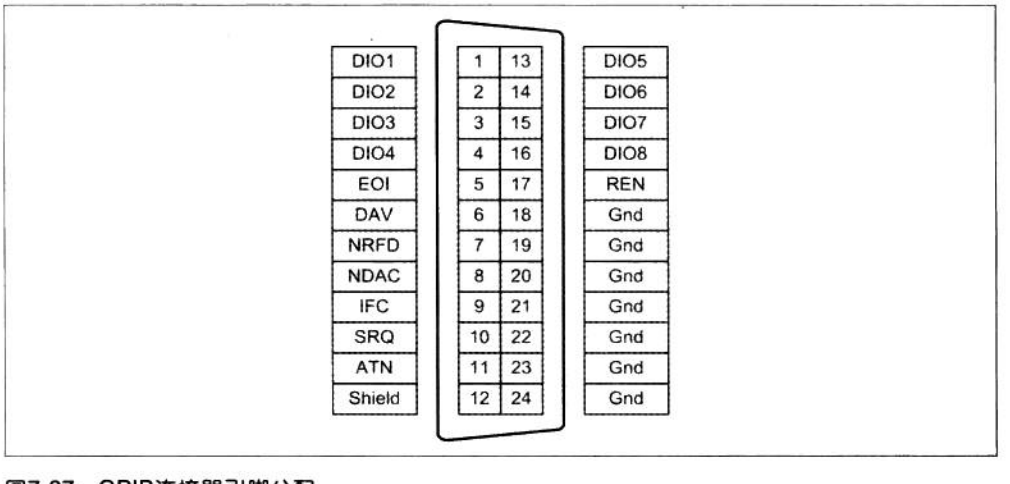


图7-37：GPIB连接器引脚分配

GPIB 连接

GPIB 允许使用一个特制的 24 针连接器连接到堆叠。总线上最大的数据传输率取决于其中最慢的设备。图 7-38 显示了一个 GPIB/IEEE-488 连接器以及如何依次将两个或者多个设备以堆叠方式连接在一起。

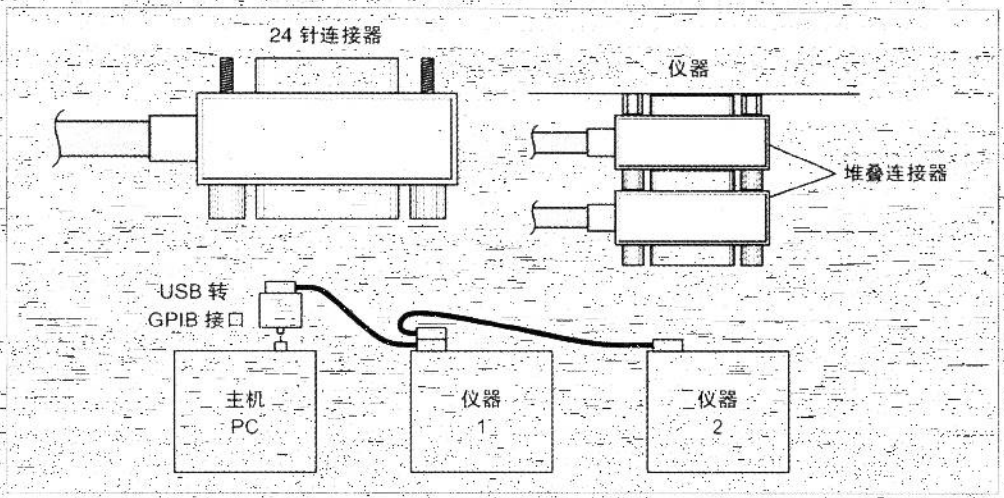


图7-38：GPIB仪器接口

GPIB 设备也支持以星形方式连接，如图 7-39 所示。

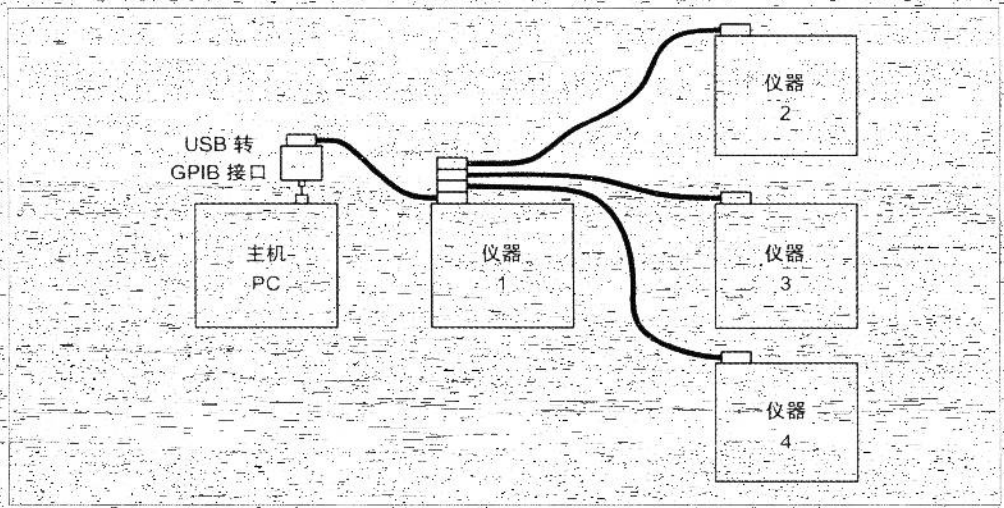


图7-39：GPIB星形连接配置

GPIB 转接 USB

虽然可以在一个全尺寸的计算机中安装一块 GPIB 的 PCI 接口卡，但是现在也可以使用价格便宜的 USB 转 GPIB 接口模块，其中一种如图 7-40 所示。这些在笔记本类电脑上工作得很好，而且大多数都为应用程序提供了接口软件，同时还为定制软件提供了带有应用程序接口（API）的驱动程序。

图7-40：USB转GPIB接口

Prologix 接口如图 7-40 所示，使用的是 FTDI 公司的 FT245R USB 接口芯片，并且提供了两个接口模式：Windows 虚拟串口 /Linux 串口驱动程序、通过库模块直接访问。FTDI 公司同时提供 Windows 和 Linux 的驱动程序。注意到 Prologix 设备可以直接连接到仪器的 GPIB 端口，因此，当只有一个 GPIB 设备时，能节省下够买 GPIB 线缆的费用。也可以为多台仪器中的每一台配置一个自己的转换器，并且通过 USB 集线器连接，而不是在这些设备中通过 GPIB 菊花链线缆连接。

在内部，一个 USB 转 GPIB 接口通常有两个主要组成部分：USB 接口硬件和 GPIB 接口处理器，如图 7-41 所示。 240

241

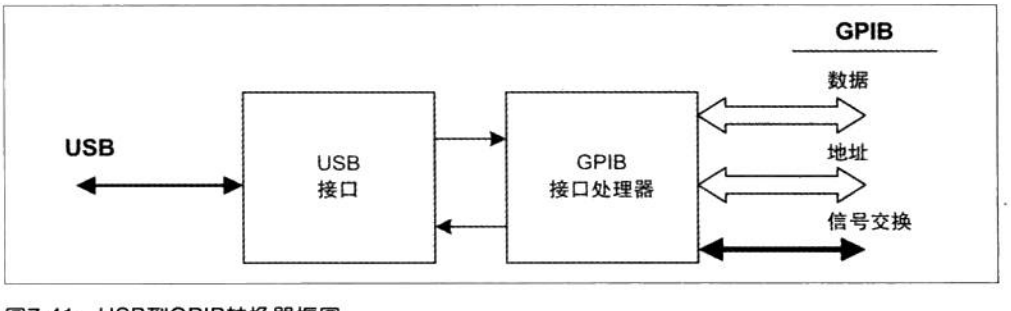


图7-41：USB到GPIB转换器框图

通过 USB 到 GPIB 转换器，你基本上都在使用一个串行端口（USB）与并行仪器总线（GPIB）通信。转换器内部逻辑处理 GPIB 握手和数据传送操作。如果该转换器使用的是一个虚拟串口，那么所有适用于串行编程的在这里也适用。

PC 总线接口设备

直接连到计算机内部总线的设备通常不会同连接到串口或 GPIB 型设备一样具有命令 - 响应协议。相反，它们的接口协议以驱动软件提供的可以调用的函式形式体现。由于更高一层软件接口的复杂性，它们通常难以使用。

你可以买到各种能够实时捕获高频信号的 PC 接口卡，然而却找不到一个 RS-232 或 RS-485
242
接口的这种设备，只有少数的带有高速 USB 接口（并且 USB 设备往往采用某种类型的缓存或临时存储，以帮助设备和主机系统之间的数据传输）。

基于总线接口卡最常见的类型是工业标准外围组件互连（Peripheral Component Interconnect，简称 PCI）总线及其最新变种——PCI Express（PCIe）总线。一个典型的 PCI 多功能数据采集（DAQ）卡如图 7-42 所示。另外，也有一些工业 VME 总线的附加卡和 PXI 总线的仪器，但是我们在本书不会介绍，大多数我们要实现的用 PCI 或 PCIe 型卡都能实现。

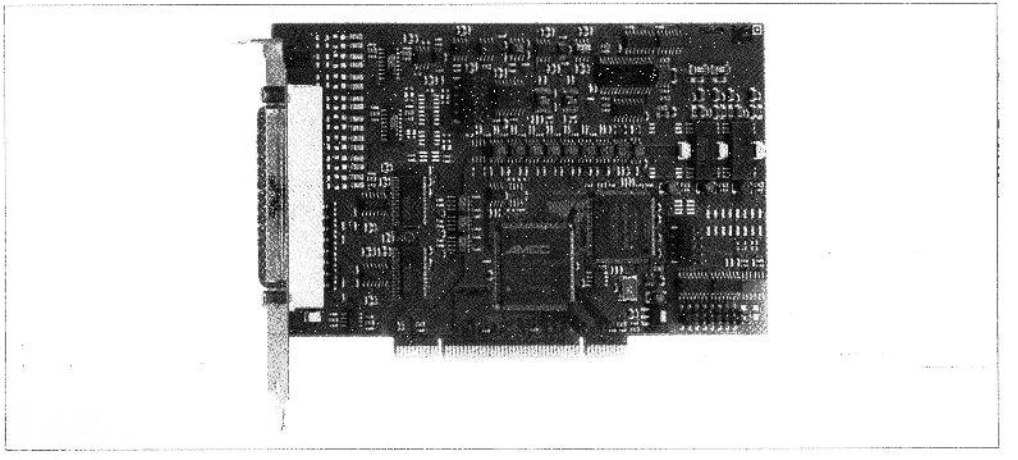


图7-42：PCI接口卡（ADDI-DATA APCI-3001）

基于总线接口的优缺点

基于总线的接口与串行接口设备相比，有以下两个优势：

- 由于基于总线的接口直接和计算机内部的数据总线连接，因此，速度总是快过串行接口。数据可以在设备间全速（或接近全速）双向传输，从而避免了串行接口的瓶颈。
- 设备的内部设置可以直接访问，并且设备可以生成主机系统的内部中断信号，这使得它可以将功能整合在一起，而不是创建一个使用串行接口连接的外部设备。

图 7-43 显示了不同类型的 PCI 接口卡和主机系统其他部分的关系。

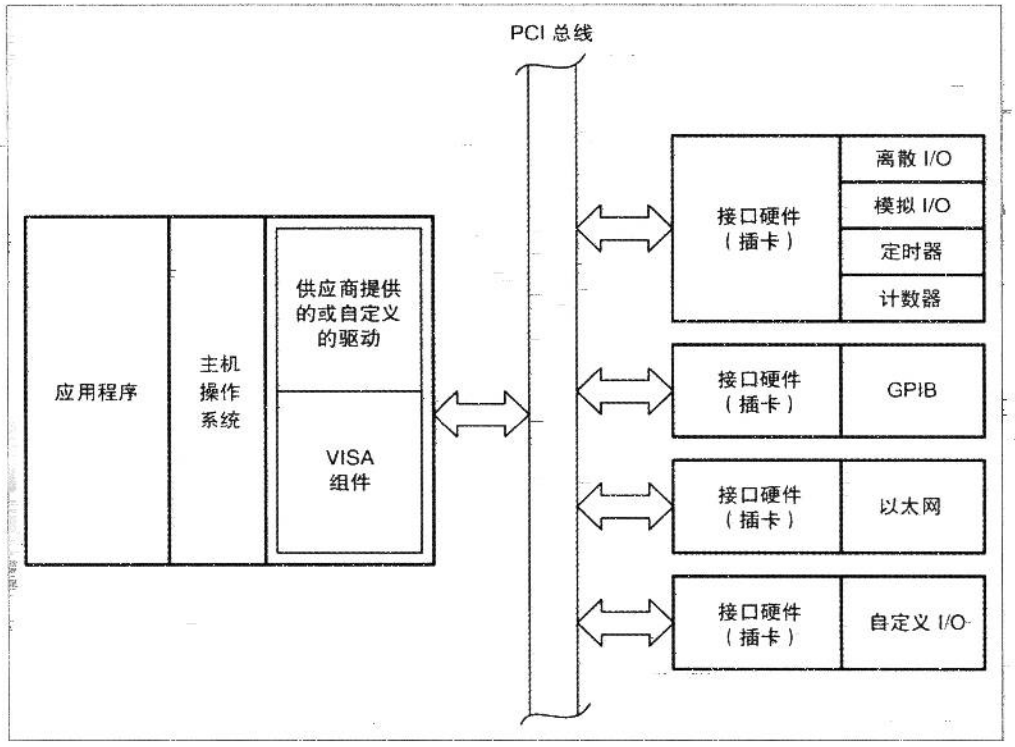


图7-43：PCI接口卡

在图 7-43 中要注意的是一个 PCI 卡总是需要额外的驱动软件作为操作系统、应用程序和接口卡本身硬件之间的接口。驱动软件提供了一组 API，其定义了用于访问硬件接口的不同操作，例如，读、写、设置参数、返回状态值等。我们将在第 11 章中讨论 VISA 驱动程序。

对于插入接口硬件来说，其缺点是需要驱动程序。我们不能仅仅通过一个串行端口来回传输 ASCII 字符串，必须创建软件与驱动程序的 API 交互，或者使用接口卡厂商提供的应用软件。另一个缺点是 PC 的背面将很快变得拥挤不堪，一些卡因使用了高密度连接

器而变得脆弱，必须使用抗疲劳支架以防止它们被拉扯或扭曲（这将导致损坏连接器、卡，或者都损坏）。此外，在 PC 上安装接口卡也不是很方便，而使用串口、USB 或 GPIB 接口（通常情况下）插拔起来就不用那么麻烦。

数据采集卡

用于采集数据的 PCI 卡种类繁多。通常，这些都被称为 DAQ(Data Acquisition，数据采集）卡。它们的范围从简单的离散 I/O 接口到复杂的多功能设备，都已经接近计算机了。

PCI 数据采集和控制卡的功能有四种基本类别，这些都列在表 7-7 中。

表7-7：基本PCI数据采集和控制卡的功能

功能	描述
模拟输出	多路模拟输出，通常具有 12 位或 16 位分辨率。有些型号还提供了离散数字 I/O 功能
模拟输入	多路单端或双端（差分）模拟输入，为 12 位或 16 位分辨率。通常有 8 个输入通道。有些型号还提供了离散数字 I/O 功能
离散数字 I/O	提供 24 ~ 96 通道模块，通常为 TTL 兼容
计数器 / 定时器	提供多个计数器 / 定时器功能（有些型号最多到 20 个）。有些型号还提供了离散数字 I/O 功能

还有第五种，被称为多功能 DAQ 卡，将以上四种的大部分功能或全部功能整合到一张卡上。这种类型的 PCI 卡可能带有模拟输入（通常为 12 或 16 位分辨率）、两个或多个模拟输出、离散 I/O 和计数器 / 定时器功能，所有的这些逻辑控制嵌入在卡上。如图 7-42 所示的 PCI 卡就是一个这样的多功能 DAQ 卡。

GPIB 接口卡

一些制造商生产 PCI GPIB 接口卡，这些卡通过访问由制造商提供的驱动软件，及其提供的编程接口的公开函式来配置、寻址和数据交换。

鉴于 GPIB 具有最高 8MB/s 或 64Mb/s 的速率，GPIB 接口理论上的最大运行速度比全速 USB 快五倍（不是速度高达 480Mbit/s 的高速 USB 接口）。因此，尽管一个全速的 USB 转 GPI 接口使用起来简单方便，但是对于需要高速数据传输的 GPIB 接口来说毫无用处。虽然也有几种高速的 USB 转 GPIB 接口可以使用，但是其成本要比全速设备高出好几倍。

安捷伦的 82350B 是一种插入到 PCI 总线中的 GPIB 接口卡，如图 7-44 所示。

PCI 接口卡通常会提供诸如高速数据传输和全套的 IVI 标准接口功能这样的特性。

245

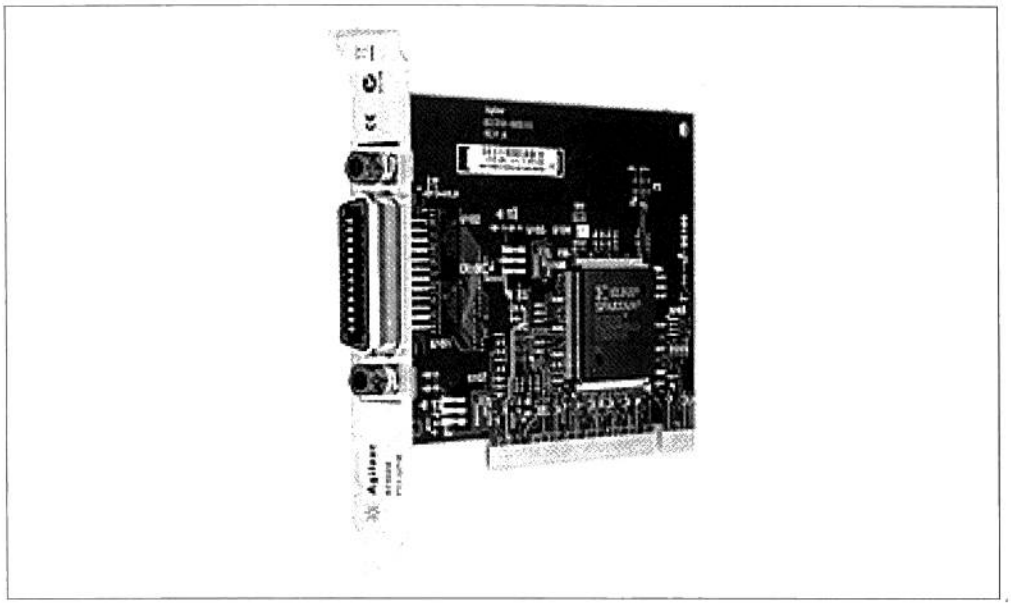


图7-44：安捷伦的82350B GPIB PCI接口卡

旧并不代表差

对于使用过的测试设备，有大量用过的仪器和接口设备可用。它们大多数仍然可以使用——只是不如新设备那样快，或者没有新的那些花里胡哨的功能而已。实际上，当你工作的系统对于表现出的变化和反应的时间在几秒到几分钟（甚至几个小时）的时候，这些都不是问题。问题是这些旧设备是否容易连接到一台 PC 上，并且需要什么样的接口软件。

旧设备的一个缺点是它们通常不会有任何 USB 支持。但是，这在许多情况下还真算不上是个大问题。除非设备使用了一个特别定制的接口，否则它也应该有一个可用的串口或 GPIB 接口。

采用了自定义接口的旧插卡设备，如果使用的是 ISA 型总线连接，那就无法连接到现代的计算机上。除了某些型号的工业机架式计算机，根本不可能买到具有 ISA 型总线插槽的计算机了。虽然可能会找到一台带有合适 CPU 的旧 PC 能够将 Python 运行起来，但是装配系统和为其编写接口软件所付出的努力通常是不值得的，买一块新型的带有现代接口的新设备更有意义。

仍然有大量的旧设备提供串口或 GPIB 接口，有些是值得被系统采用的，而不是将它们

246

回收或当垃圾填埋掉。正如我们将在第 11 章中看到的那样，可以实现一个合适的接口与外部仪器通信，而且编写这样的软件并不困难。

小结

本章我们讨论了很多底层的知识，现在我们应该真正开始思考如何在仪器系统中选择和使用物理接口。我们已经了解如何在不同的接口类型中使用不同的连接器，也学会了组装连接器的基本知识。接着，我们从硬件角度预览了稍后将处理的各类接口。当我们在后面章节中开始讨论真正的仪器和系统时，这方面的知识会很有用。

推荐阅读

尽管有些参考书籍涵盖了关于我们在本章中的主题，但是你能在网上找到你所需要了解的一切。网上有大量关于 RS-232、RS-485、USB 和 GPIB 的信息。例如：

http://www.omega.com/techref/pdf/rs-232.pdf

Omega Engineering 提供了一个 RS-232 标准的单页摘要。你可以放在工作台或者贴在实验室里作为一个快速参考。

此外，正如你所期望的，维基百科上有关串行通信的所有条目。

寻找连接器及其专用工具的技术信息的最好地方是分销商的网站。以下列出几个网址：

http://www.DigiKey.com
http://www.alliedelec.com
http://www.mouser.com

另一个很好的资源是国家和军用标准。这些文档中很多都深入有关组装和接口技术的细节，如果你不需要所有的信息，他们通常也会提供一些高层次的材料，有的甚至只能作为教程。以下这份美国航空航天局（NASA）的文档就是其中一个标准，其他还有很多。

247 *NASA STD 8739.3,“ Soldered Electrical Connections.” NASA Technical Standards Program, NASA, 1997.*

在这个技术标准计划中，大量的技术标准可以从 NASA 那里免费得到。其中一个是高可靠性的焊接技术简明指南，可以通过 *http://www.hq.nasa.gov/office/codeq/doctree/87393.htm* 得到。

美国航空航天局（NASA）所提供的技术标准目录在 *http://standards.nasa.gov/documents/nasa* 中可以找到。

第 8 章 248

开始干吧

领先的秘密是开始干，开始干的秘密在于将复杂的设计转变成一系列可控的小任务，并按次序就地开工！

——Mark Twain

在本章中，我们将继续遭遇那些在各种规模的工程中被反复证明过的东西。对，没错，这里说的就是计划，这包含需求分析和对应的设计说明、文档化对可预见的各种问题的测试规划，然后用这些测试来验证是否吻合设计要求。要知道，缺乏或不明确的需求设计一向是软件项目失败的主要原因，另一大因素就是测试不足。

对于小项目，如果没按期交付成果当然是恼人的，但可能不算是个大灾难。对于大项目，其结果可能就是灾难性的。对于逻辑复杂，而且健壮性又有要求的程序，在开始之前进行需求分析，并构筑良好的测试环境是必需的。另外，还要注意对于仪器软件，必须面对现实世界中可能非常严重的不确定、模糊、蠕变和其他各种意外，这时，有一个明确的技术路径和良好定义的目标就非常关键了。

本章的目的很简单：我们想让你成功。在此假设你已可以很好地完成一些小软件，尽管它们本身可能很简单，但却能成为其他工程的关键组件。它或许是实验室仪器的控制器，或许是研究过程中的数据采集系统，也可能是生产环境中的自动化测试系统。又或许是世界级高尔夫球场的洒水控制器系统。不管是什么，如果它不能工作，或者不能可靠地工作，可能就意味着时间、收入的损失，或是无用数据，甚至更糟。

多年的经历使我们逐渐确信需求分析是十分必要的，我们不认为任何人坐在键盘旁边猛敲一会就能写出没有 bug 且功能完善的软件来。其实，我们遇到过很多人真信那是可能的（他们当中的许多人真的是心里有数的），更糟的是，有的人不明白学校编程课里的那点小练习不经过大改是没法成为真正的应用程序的。也有一些人没法明白为什么他们 249

的软件不按照他们（或者客户）预期的那样运行，不明白它为什么充满了bug。其实那完全可以避免的。

我们希望本章能向你展示为软件制定一些基本的需求和计划是多么的简单。在这一阶段所花费的时间和精力会在项目的后期给你带来丰厚的回报。除了不用解决bug或者对错误的数据感到奇怪外，你还可以自豪地展示你的作品，因为你知道你已经采取了正确的措施来保证软件按照预期工作。

项目定义

你究竟想创造什么？这是最重要的问题，但是多数人在这上面花的精力并不比在会议中有意识地随手涂鸦更多。缺少真实的需求分析和将需求转化为能用的代码的计划是大部分软件工程失败的原因（当前，所有的重大项目工程中50%～70%是失败的——这取决于你如何定义“失败”）。试试搜索“需求不当导致项目失败”（poor requirements—software failures），就知道有多少文章在写这些事儿了。

所有的项目都是由某种需求陈述开始的。它可以是一个非正式的想法，或者是电子邮件里的一个请求，当然也可以是正式的文档。从这一刻开始，项目的目标就以工作说明书（statement of work，SOW）的形式开始发展了。工作说明书有需求陈述所不具备的细节。接着各种具体要求就聚集过来，最后才能得到软件设计指标（design specification）。图8-1展示了从需求陈述到设计指标过程中的每一步变迁。

需求在每一阶段都得到进一步细化。细节最后在软件设计文档（software design description，SDD）中得到最终陈述。（理想中的）说明书包含足够的信息来促成满足要求的软件的诞生。在接下来的内容中，我们将逐一体会迈向设计说明书的每一步。

需求驱动的设计

如果你学习过相关课程，就应该反应过来了，寻求项目定义的过程正好和所谓的“瀑布”（waterfall）式软件开发模式相吻合。如果你不熟悉这个软件工程的术语“瀑布”（waterfall），也不用理会它，这只是一种软件构思和开发活动的组织方式。瀑布（waterfall）模型是描述软件开发的生命周期（software development life cycle，SDLC）的方法之一。图8-1基本上是瀑布模型的前半部分，后面将会看到更多。当然，在不同的问题领域可能有更优越的模型，不过，我们并不迷信哪种模式，也不相信对需求的剪裁能更好地与真实世界同步。对需求的修订，最大的问题是如何使变化尽可能地减少对已完成工作的影响，变更会引发“涟漪效应”，如果涟漪过大，将变成真正的巨浪。在大型项目中，这将变成重大问题，不过，同样的问题在小项目中，现场也就解决了，没什么大不了的。

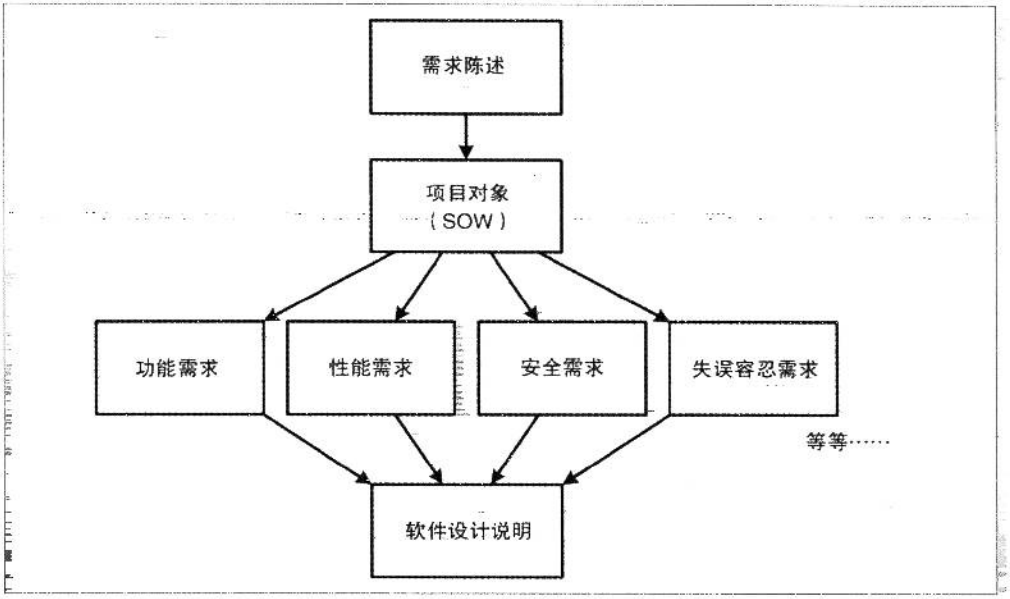


图8-1：需求的演化

本书一直运用所谓的“需求驱动设计（RDD）”模式。当需求可以尽早确立并不会经常变动时，RDD 工作良好。有些项目，特别是那些和人进行交流的项目（例如，银行、医疗记录和其他“流通性”领域），可能需要具有比 RDD 更多的灵活性。在我们的场景中，则多数是类似“黑盒子”的商用接口和控制协议，都是专门用来完成反复进行的简单任务，除非硬件磨损或电源关闭，否则不会出现变更，主要的挑战在于如何确保所有的部件可靠地结合在一起，并明确收集的数据是什么。

251

软件开发模式就像工具箱中的工具：你总是想（我们假设）要使用适当的工具完成手头的工作，螺丝刀无法和螺钉配合起来，锤子对于螺钉是无用的。在我们的例子中，我们相信该工具恰好是 RDD 范式。同样的道理也适用于编程语言、编辑器和开发平台。20 世纪一位著名的社会学家马斯洛曾经说过：“如果你所拥有的唯一工具是一把锤子，你往往会看到每一个钉子的问题。”如果你想要了解更多其他的软件开发模式，我们会鼓励你自主探索已有的大量信息。事实上，线上和线下会议中，的确存在各种开发模式的激烈讨论。

从需求开始

无论采用哪种开发模式，建立需求陈述（statement of need）是一切的开端。它可以用一句简单的话概括：“我们需要能根据预设值来监视和控制试验室中的温度和照明水平。”

或是更复杂一些，如："在进行压力测试时，我们希望能够对多达 20 个固态激光器的功率和操作状态同时进行调节和监控。"注意，需求陈述中不应该包含任何实现细节，只是单纯的概念，阐述需要什么。当我们讨论需求时，是无法从模糊的输入建立什么东西而最后又能完成预期目标的，因为，在这一阶段根本无法明确什么是实际的期望。当你在给人们创建东西时（如客户），一般而言，他们不会清楚自己真正期许的是什么。

图 8-2 显示了一个虚拟交流电源控制器项目的需求陈述。这份说明没有包括任何细节，但它确实有很多含义。虽然还有很多悬而未决的问题，但足够帮助我们推断系统必须由计算机控制，它必须是无人值守的。再分析几分钟还应推导出如果它是 7×24 小时自动运行的，应该能够感知到是否发生错误并及时采取适当的行动，并以某种方式提醒操作员。

252 最重要的一点是，需求陈述（statement of need）和需求（requirement）是两回事。在某些圈子里，它有时被风趣地称为"性欲"。它只有基础大纲，不包含整个结构，但会仔细考虑并指出是什么，通过一些问题指出了前进的方向，以获取必要的细节。

需求陈述

试验组需要一种可以自动呼叫的交流电源测试系统，以便监测各测试单元所在的燃烧室，目前在预先约定的固定时刻进行的手工控制方法既烦琐且易出错，涉及 24 小时（或更长）的测试周期时，就意味着要有运营组在整个测试期间进行值守。自动化系统将允许测试期间无人值守。

图8-2：需求陈述

工程目标

工程目标是下一级详细的设计阶段，也被称为工作说明书。这些仍然不是严格意义上的需求，但已经相当明确地定义了如何达成需求的具体目标。例如，我们假设的交流控制系统的工作说明书可能如图 8-3 所示的形式。

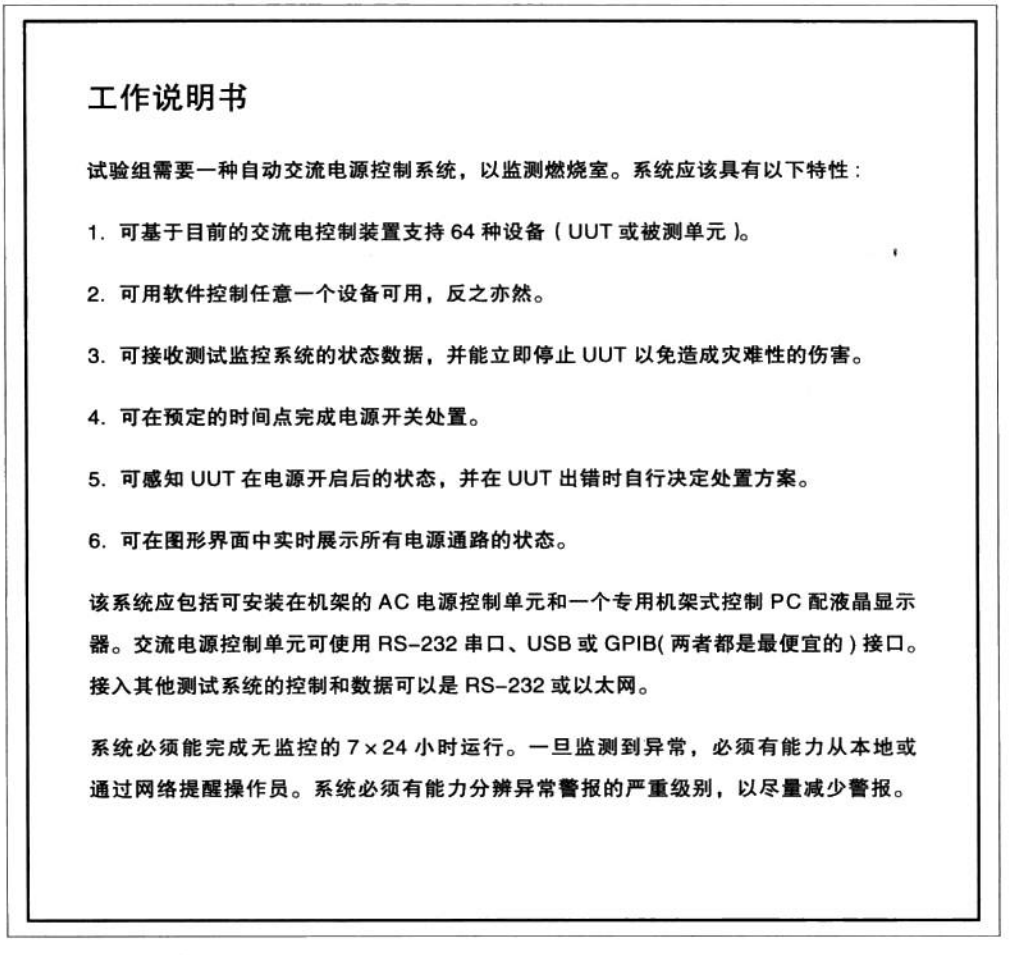


图8-3：工作说明书

要注意的是，这里的目标仍然不包含任何明确定义的参数，例如，工期、特例或数据通信协议，只包含最终系统高层次的功能特性（即一般行为），以及这些功能需求从哪儿衍生出来。工作说明书（SOW）定义了从最终用户的视角来判定工程是否成败的基础标准（或从你的角度来看，如果这是为你自己的工程在写的 SOW）。

需求

需求来自很多层级的细节。SOW 之后的层级可能是高阶工程目标、科学需求、操作需求，等等，通过逐层递进的细节积累，最终将软件的主要需求包含在 SDD（软件设计文档）中。

253 较低层次的需求都来源于他们的要求，在自上而下的方式（瀑布模型）中，各层需求分解自高层需求，每层都需要进行需求分析，以得出足够的细节来指导实施。究竟这种分解活动要用多少时间，依工程类型而定，有些足够简单的只要一个可测试的功能需求就
254 好了，而其他的（像喷气式飞机、航天器上的指标和推进控制系统）可能就需要进行大量需求分析，以确保所有特殊的状况都已经包含。所有底层实现的界定和审查都有对应系统的必要指标和方案，这种工程的需求文档能轻易超过数百页。需求分析的难点之一，就是对所有部分的了解必然保持更新，不断增补细节。

为什么需要需求

没有需求或没有工作说明书，就无法对最终的结果形成清晰的思路，也无法明确如何行事。当然，有时系统应该做什么只是一些模糊的概念，模糊的概念终究是模糊的，肯定是不和其他人的模糊概念相同的。午餐期间，餐巾纸上的涂鸦不是需求；会议期间，草草记在白板上的也不是。类似的东西都不是明确的需求，不仅是因为它们不符合可核查的标准，而且还因为它们只是些浪花儿，而实际上其下面可能是非常深的海沟。

即使是最小的需求集合，也足以引导我们定义出明确和具体的实现要求。那种可以令所有的人认同的就是好需求，同时，好需求还可以对成品的成败给出明确的测量。

一则真实的需求故事

很久以前，我们负责为一个大型的射电望远镜系统的滤波器阵列建立实时数据采集系统。该滤波器阵列是旧设备，其存储单元甚至要人工拖出来，而且有 640 个独立通道，各自对应到一个特定的频率。我们的工作是将高速模拟数字转换器和旧的滤波器阵列系统整合起来，通过实时操作系统获得每个通道的数据，射频工程师（RF engineer）也时刻准备好了，一切都很靠谱，直到我们向滤波器阵列负责人咨询有什么特殊需要时，开启了一段奇幻之旅……

我们问："每个通道的转换时间最大可容忍多少？"回答："嗯嗯嗯，尽可能快"，这明显是不靠谱的，所以我们尝试换种方式再问，仍然收到模糊的回答。一来一去，就这样至少 30 分钟，我们越来越沮丧，而其他人越来越恼火，认为我们只是企图制造一些数字。最终我们总算挤出了一点儿基本的数值来开展工作，但仍然不得不用乱猜的方式来弥补一些"失踪"的数值。

最终，系统满足，甚至超过了预期的时间和稳定性要求。事实上，它工作得太好了一点。我们第一次尝试让它在全速度下运行，对望远镜所有 640 个通道的控制和数据流进行处理，于是崩溃了，结果整个价值数百万美元的射电望远镜也死了，用了两个小时才重新启动整个系统。幸运的是，这发生在预定的工程时间之内，否则我们可能不得不面对一群愤怒的科学家。这个故事的寓意是：永远不要假设用户能够提供需求中正确的基准值（并愿意承认他的确不知道）。你得做好准备自行填空，并仔细检查你的假设。这其实是个好主意，让你需要的答案从必须监控的报警中自己跳出来。这样，至少他们无法指责你总是要拷问他们。

良好的需求

需求的描述必须是清晰、准确和毫不含糊的，所有的特性必须指出具备什么行为来满足一个或多个项目目标，每条需求的文本总会包含“应……”的句式，这里有个例句：

> 对任何被待测部件（UUT），要求系统应能在 AC 电源被接通后 100ms 内检测到未响应的 UUT。

靠谱的需求具有一个非常明显的特性：它是可验证的。换句话说，它定义了可以用测试来验证特性能力的具体数值、限制和例外（如果有的话）。前例中的功能要求就是可测试的，因为提供了精确的事件和时间要求，但是，即使包含“应……”的句式，人们也很容易写出无法测试的需求描述，例如：

> 该系统应能够及时发现并妥善处理无效数据。

就行文本身而言，这不是个真正的需求，只算个要求。例如，到底意味着什么？什么是“无效数据”？什么是“处理”？怎么算是否合适？什么都没有定义，因此，无法进行有效的测试。需求的底线是这个“需求”能否被验证！

基本上有五个基本点可以促成靠谱的需求，表 8-1 中列出了这些需求。

表8-1：需求的必要元素

元素	描述
必需的	需求只应该提供足以达成在 SOW 中定义目标的基础功能和特性就好。换言之，不得有“花里胡哨”的想象
明确的	每条需要只能有唯一的一种阐述。对需求多种可能的解释将导致系统无法满足 SOW 中定义的目标，需求必须简洁明确
一致的	需求绝不能引入相互冲突的目标、参数或功能。项目范围内的需求必须能互相支撑

续表

元素	描述
256 可追踪的	每条需求必须有一个可追溯的成立理由，这确立了需求或是 SOW 中定义目标间的关联并支撑起高级别的需求项。换言之，需求能够向下传递并向上追踪
可验证的	每条需求必须能够进行测试、分析、检查或（在某些情况下）演示来验证是否成功。一个无法验证的需求是无法实现的，因为无法明确该软件是否实现承诺的

当然，还有其他标准，但是表 8-1 中的五个标准通常被认为是最重要的。

全景

探讨了需求的基本要求之后，我们才能更好地理解什么是合适的软件开发和核查。图 8-4 显示了如何从 SOW（项目描述和目标）到达可用的软件，以及如何依据需求设立测试。

需求类型

我们已经明确了 SOW（项目描述和目标），在讨论图 8-4 之前，根据流程，接下来我们会在需求集合中遇到三个不同类型的要求：一体化、性能和功能。这些都是可能出现在相同文档中密切相关的要求，也有基于以上被称为“派生需求”的。让我们仔细看看这些不同类型的需求。

功能需求

功能需求以及任何相关的性能和集成需求，通常指对外部设备和系统的行为。换句话说，它们响应电源设置为“开”或是接受某种输入，系统应该做什么，功能需求可以定义为将系统或是设备视为“黑盒子”时应有的行为。也即，对于任何给定的输入，必须生成相应的输出。至于如何实现，这不在功能范围内，那是由 SDD（软件设计文档）实现的。

性能需求

性能需求描述系统必须有多出色。通常涉及类似如速度（单位时间内完成传输的数据）、容量（可存储多少数据），以及实时显示更新率（视频直播、实时数据图形显示等）
257 相关的指标。性能需求一般是对功能需求的补充，所以，两者有时会合并成同一需求文档。

集成需求

集成需求定义了系统如何同其他系统交互。关注类似通信协议、命令、数据格式以及物理接口。有时这类需求也附有性能需求。另外，集成需求可能包含在单独的接口控制文件（Interface Control Document，ICD）中。

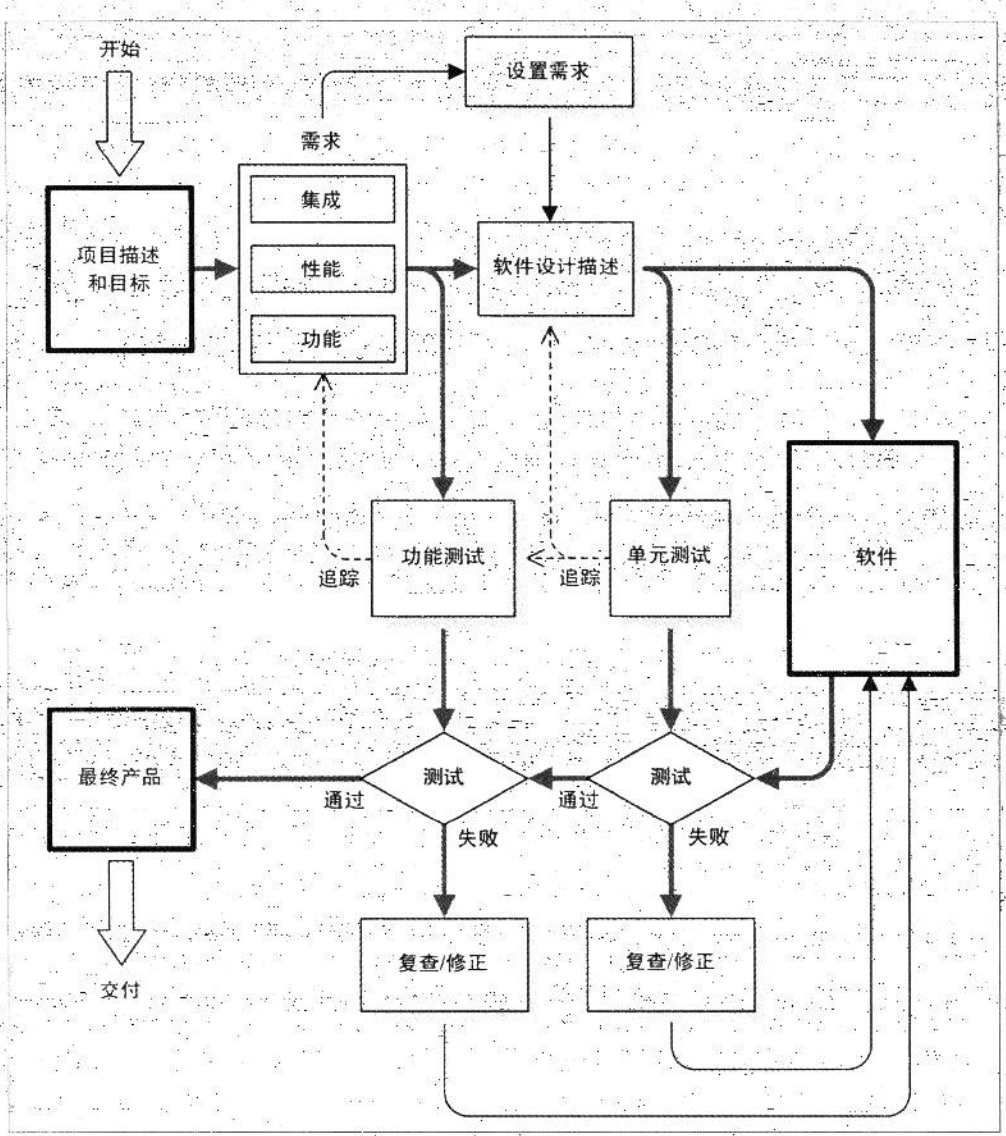


图8-4：开发流程中的软件需求

派生需求

派生需求是在规定的功能中没有被明确要求但是包含在需求分析和设计文档中的性能或集成性需求。例如，系统可能需要有通信接口错误检测报告功能，虽然没有在其他地方明确指出，但是考虑到底层开发的便利，应该包含进来。可以这么说，派生需求就是“查漏补缺”，为其他高层需求提供可追溯的设计元素。

多数情况下，需求是要求着眼于我们将做什么，这有时被称为合同形式的需求。即每个需求必须是包含“应……”句式的书面语句，形成明确的可核查的有组织、有次序的僵硬的正式文档。直至今日，特别是在航空航天和国防行业，这都是标准的需求捕捉和记录格式。然而，目前商业软件的需求工程中，已经有趋势在淘汰这一模式，用例或其他技术正在兴起。图 8-5 显示了新风格需求记录的例子。在下一节我们就来看看用例。

1.1 数据采集子系统软件应能够以每秒 1000 个样本的速率，接收并转换模拟信号为 16 位有符号的数据样本。

1.2 数据采集子系统软件应产生 -32768 ~ 32768 之间的值，输入电压范围为 -10 ~ +10 V，精度为 ±1 位。

1.3 数据采集子系统软件应能对超出硬件检测范畴的正负电压设置一个软件的状态位，以特定的软件状态为条件，根据约定的范围条件返回最大值或最小值。

1.4 数据采集子系统软件应通过配置参数文件，调节输入可接收的上下限数值。

1.5 在实际输入超出配置限定的范围时，数据采集子系统软件应设置一个条件状态位，并返回转换后的实际数值。

1.6 数据采集子系统软件应能检测到数据转换硬件的失败，用一个条件状态位来标识，并返回一个错误代码值代替有效的数据。

图8-5：合同式需求

对于小需求集（20 ~ 50 条之间的），其实用什么形式都可以，但是当需求集变得非常大时，形式就成为问题了。因为，对于确定的事物，合同式需求比较抽象，即会脱离 SOW 和其他可能丢失的原始信息，这样就可能难以察觉需求间的相互关联。

用例

合同式需求不是唯一的需求捕获方法。此外，已经提及对于图形用户界面或是其他人机交互的需求细节，合同式需求就变得很笨拙。这时，用例式需求可能就更加合适。

258

类似算法和数学方面的需求就不适合用例式的需求捕获，其他非功能性特征如性能、定时，或是安全要求也同样不适用。

伊瓦·雅各布森（Ivar Jacobson）在1986年发明了用例，它作为一种模型，描述系统和一个或是多个用户系统（可以是真实的人或其他系统）之间为完成特定任务的相互作用。用例将系统视为一个“黑盒子”，只关注重点输入/输出操作。从这个角度看，用例是一种功能需求。所以，用例可能不适合作为底层执行的情况说明方式。尽管有些人也试图在SDD中强行尝试用例形式。在20世纪90年代末，用例方法被纳入了统一建模语言（UML）。

259

用例最简单的形式中，有四个主要的元素：角色、系统、目标和实现目标需要的必要步骤。用例可以通过分层的组织形式来定义其他用例，一般从高层次的用例开始，并逐级引入细节。也可以通过关联同级的其他用例来一起描述为完成共同目标的必要步骤。

图8-6展示一个假想的温室控制系统的使用情况（诚然有点做作）。在这个例子中使用了6个元素，如表8-2所示。

表8-2：用例基础元素

260

元素	描述
身份标识	一个用例的唯一标识。在项目的整个生命周期中，用于明确地指代特定的用例
标题	用例标题。和身份标识绑定在一起，在整个项目周期中用于简要说明用例
描述	也被称为用例的“目标”。在较高层次定义用例的企图
角色	在描述中指出需要完成的目标或目标的代理（人或其他对象）
系统	响应角色来实现目标的事物
假设	为了目标的可达成，而设定的必需条件
步骤	为了完成任务所必须完成的事件序列（可能只有一个）

如果只接触用例，你可能认为它们只是被称为“用例图”的漫画而已。其实不是，用例图的目的不是为了定义功能要求，而是提供一种视图，可以从高层次表述系统将由谁怎么使用。图8-6展示了一个典型的正式用例，有时也叫传统用例或者具体用例。

另一个需要注意的是，良好的需求标准也同时适用于用例，毕竟到最后都是为了需求描述而服务的。事实上，一个正式的用例图就是一个或一组合同式需求对单一目标的可视化描述。

用例的生成和组织的细节是此书范围以外的内容，如果你有进一步了解的兴趣，我们建议参考本章结束时的“推荐阅读”一节，或者也可以钻研互联网上的资料。维基百科有一个很好的介绍，网址为 *http://en.wikipedia.org/wiki/Use_case*。

我们建议使用ASCII纯文本文件来组织用例，因为你可以对文本使用类似CVS或Subversion的版本工具进行控制。CVS和Subversion有能力自动追踪文件的修改到特殊标记的行、最后的作者、版本号、上次检查的日期等。

身份标识：UC-GH-021

标题：	定期获得温室控制系统的温度数据
描述：	监控系统（演员）定期查询控制系统，并获得 GUI 显示的温度数据。控制系统和监控系统之间的交易是通过网络连接使用一个自定义的通信协议完成的。
演员：	演员组成一个外置的计算机系统，用于监视主控制系统和显示数据，如温度随着时间的推移和错误通知使用 GUI。
系统：	控制温室内的温度、湿度、浇水和照明的专用控制系统。
假设：	通信协议用于交换命令和数据之间的演员和系统的定义在该项目 ICD 中，监测和控制系统的实施。 控制系统供电和运行正常。 监控系统通电和运行正常。 以太网连接之间已建立的控制和监视系统。
步骤：	1. 演员向系统发送查询请求目前的温室温度。 2. 控制系统响应与 ASCII 字符的字符串定义在 ICD。 3. 演员分析系统的响应，要么给出一个新的数据点，要么显示适当的错误消息。 4. 如果控制系统没有响应，则在显示器上显示相应的错误。

图8-6：正式使用用例的范例

可追溯性

可追溯性是指需求和其他关联事务间存在的先决条件问题。无论是合同式还是用例形式的需求，都必须说明需求存在的理由。

图 8-7 展示了可追溯性的图景，从 SOW 开始，然后派生出功能和需求，再到 SDD，最后是软件本身。这一流程在图 8-4 中也用虚线标示出了这一“追溯”关系。

262

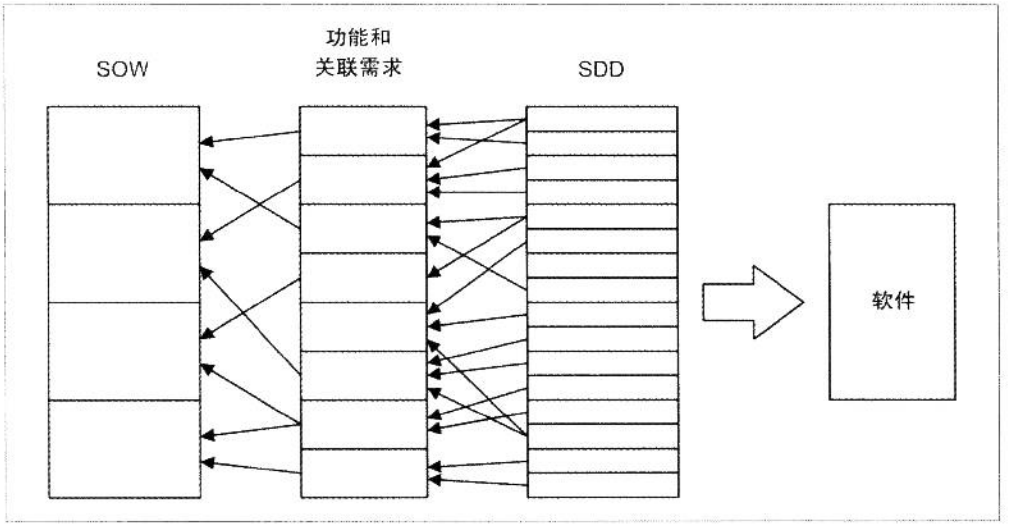


图8-7：需求的可追溯性

在每个级别中，可追溯性对于“需求范围”的核实至关重要。换句话说，功能和派生需求是否已经涵盖所有 SOW 要求的事情？SDD 是否已经涵盖所有的功能和派生需求？没有可追溯性，可能无法明确以上的基本问题，尤其是在需求复杂的案例中。

一个需求追踪矩阵如表 8-3 所示，它展现了各基本需求是如何被映射到每个功能的。类似的有所谓的“验证矩阵”，用于将测试映射到对应的需求。

表8-3：需求追踪矩阵

功能编号	需求	SDD 定义
1.1	数据采集子系统软件应能够以每秒 1000 个样本的速率，接收并转换模拟信号为 16 位有符号的数据样本	3.2.1
1.2	数据采集子系统软件应产生 -32768 ~ +32767 之间的值，输入精度为 -10 ~ +10 V 的输入电压对应 ±1 位。	3.2.2
1.3	数据采集子系统软件应能对超出硬件检测范畴的正负电压设置一个软件的状态位，以特定的软件状态位为条件，根据约定的范围条件返回最大或最小值	3.1.1 3.7.3
1.4	数据采集子系统软件应通过配置参数文件，调节输入可接收的上下限数值	3.3.2
1.5	在实际输入超出配置限定的范围时，数据采集子系统软件应设置一个条件状态位，并返回转换后的实际数值	3.4.1 4.2.4
1.6	数据采集子系统软件应能够检测到数据转换硬件的失败，用一个条件状态位来标识，并返回一个错误代码值代替有效的数据	3.4.2 4.2.5

263

表 8-3 中，条目使用了需求的原始文本，其实只用需求标题的也足够了。而且读者也明白，这里只是将 SDD 部分的引用目标填到表格里。

需求捕获

捕获正确的需求可能是个艰巨的任务，特别是如果用户自身并不知道答案时。在研究环境中，这实际上是常态，因为主要用户（首席研究员、部门主管等）对项目可产生的数据远比明确系统如何工作要更感兴趣，他们真的不在乎所有设备的详细信息，能做什么，是否可靠和准确，数据转换率、电压限额等具体问题。所以，从他们心中捕获需求往往难以高效达成。

类似的情况也存在于商业环境中，其软件产品的需求是由营销部门主导的。营销部门对包含什么功能才能形成有竞争力的产品更加有兴趣，对于具体底层的细节，咨询他们通常是没有什么结果。

当然，在航空航天和国防工业，以及其他一些工业领域，有些特定需求在高层级就已能得出，例如，性能、错误处理，等等。但在这些领域外，这类需求更多地被归类成异常规则。即使已有详细规定，这类需求也不应盲目接受，必须有人将审查必要的一致性和可行性，而且他们的努力将成为控制项目成本的主要动力。

综上所述，多数情况下，如果你想根据扎实的需求来展开工作（也就是说，你真的渴望要这些东西），你就得去帮助用户创建需求。为此，你可能需要采访所有相关的人士，明确他们对项目的真实想法。如果需求文档不存在，就应立即建立一条需求，并获得所有相关人的同意（在商务用语中，你需要得到“买进”的许可）。一旦关键需求部分到位，接下来就是按照层次先后进行细节的推敲，直到足够指引设计、创建、测试所需的系统。

264

若试图捕捉到底层细节，起草问题清单是个好方法，清单中要包含你去采访的人、要获取的答案，以及从回答中推断出的信息，最好也包含你排除的需求、假设和相应的提供者。

对于需求捕获和文档，虽然有专门的商业工具，但是，一个字处理甚至电子表格软件通常足以满足大多数小到中等规模项目的需求管理。如果你选择使用其他方法，如用例，也有可用的开放源码工具，但一般文字处理软件真的足够用了。

设计软件

现在，你有了一些功能需求（可以是任何格式记述的）后，是时候开始思考设计软件来满足这些要求了。设计有多少自由度，取决于功能需求对目标的描述有多详细。如果每条需求都非常详细，即使面对一个松散的功能需求集，你也不会有多少回旋余地。

软件设计说明

图 8-4 的开发流程中指出，获得最终软件的前一件事情就是软件设计说明（SDD）。实际上，SDD 是底层的实现要求，通常情况下，SDD 描述了比功能需求更多的细节，包含返回值、流程图、框图、继承图、消息序列图，等等。SDD 是用于构建目标系统的系统性理论读本，应该包含足够多的细节，可以支持程序员直接转化为工作代码。SDD 也遵循经典的软件工程路线来完成。换句话说，可能会有专门的章节来描述数据采集、数据处理，或者描述用户界面等。一个完备的 SDD 可能包含一个或多个概要介绍部分，来说明在 SOW（工作说明书）中指出的开发任务的技术实现细节。

对于小型项目的 SDD，也许没有必要，但除非整个工程只有你一个人完成，否则，为了确保每个参与者都明确什么是必要的需求，以及软件如何来满足这些需求，SDD 是最合适的载体，它可以同时用来驱动软件和单元测试的开发。

面对 SDD 要保持头脑清醒的事情是，SDD 仅能由对应的需求驱动产生，每个条目必须可追溯对应到一个或多个功能需求。

在需求驱动设计（Requirements-Driven Design，RDD）开发模式中，如果有什么东西不能追溯到更高级别的需求，那它不用设计。如果事情真的需要设计，但没有对应的需求，那么，必须为之建立派生需求。 265

无论如何，SDD 的存在表明了人们想把某些想法加到软件中去，即它无法从含混不清的事物中抽出来，并假设成完整和准确的描述。

SDD 的图景

一份 SDD 可以简单到只有几页文字，但如果有图形，可以帮助读者（这可能就是日后的你自己）更好地理解软件中一些体系结构的关键环节，图形表示也可以让你和设计师从新的角度来理解软件。原先的方法无法工作时，使用其他类型的图表理解后就成功的情况并不少见。图形不仅仅用来表示模式，帮助设计识别问题所在，还可以提高沟通效率，得出更好的做事方式。

本节绝不是任何图形表示或建模技术的教程，只是相关概览。我们期望读者可以根据实际情况自主使用书中所提及的工具。另外，这里介绍的各种术语和概念在网络中都免费提供有大量的信息可供参考。

框图

框图一般用以表述系统中不同功能区中软件和硬件及其相互的关系。事实上，本书已经展示了很多框图，例如图 8-4 就是框图（实际上，它是一种混合图，既是框图，又是一

个流程表）。如果设计得当，框图能以非常紧凑的形式传达大量信息。

为了编写 SDD，描述系统的概览框图是一个很好的开端。事实上，一个良好的框图配合着一两页文本或是网页，就可以是一个简单项目的整个 SDD。

流程图

流程图概念的历史比计算机本身的年龄都大，至少比编程语言问世要早 30 年。最初的流程图用于表示工业生产中的活动流程和模型。1947 年左右，在 IBM 由赫尔曼（Herman Goldstine）和冯·诺伊曼（John von Neumann）创造了今天我们熟知的软件流程图，ISO
266
标准 5807:1985 包含最新的流程图定义。不幸的是，必须上缴 100 美元才能看到原文，不过，有篇从 IBM 流传出来的老文件（GC20-815-1，“流程图技术”，1969 年）可以在网上找到并免费下载。

图 8-8 展示了一个简单控制器的流程图，该流程图不仅显示了该算法的基本功能，还显示了哪里可改进。例如，没有规定处理传感器 1 的输出错误，也没有约定如何检查以确定输出（任何可能）是否为响应。

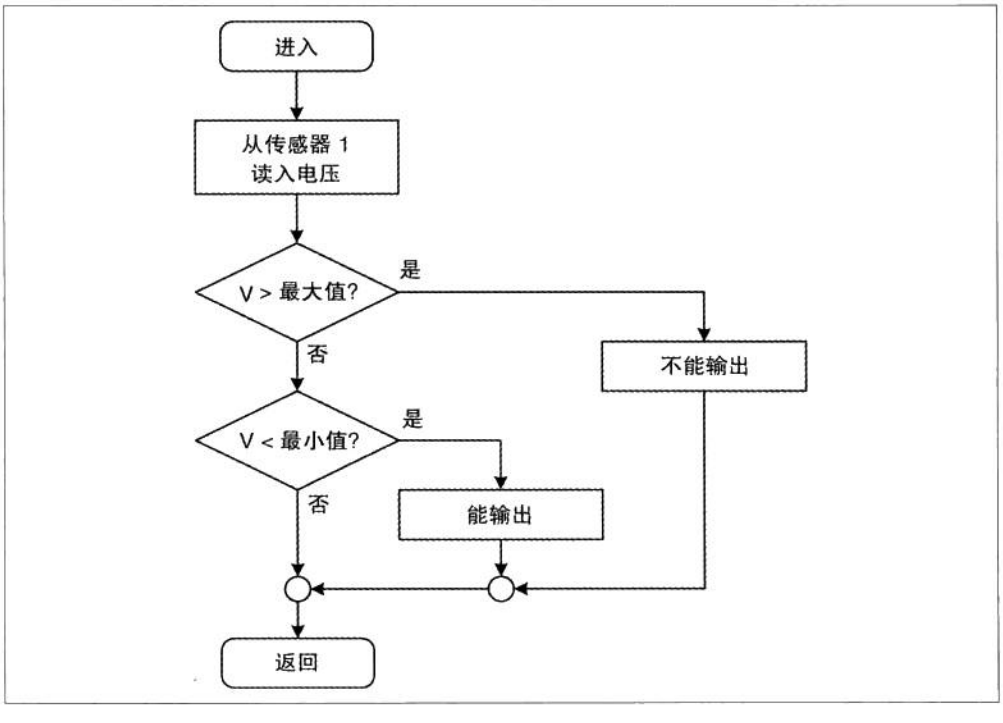


图8-8：流程图示例

有些人可能会对流程图嗤之以鼻，并声称它们已经过时，但现实中流程图的运用是非常活跃和良好的。在 UML 中，流程图相当于“活动图”，流程图是程序逻辑路径有效的可视化形式，在复杂控制流（如交换机或循环）的表述中，虽然流程图有些力不从心，尽管如此，流程图作为程序段逻辑的直观说明方式在 SDD 中就不应该被忽视。

状态图

状态图是一种对状态变化进行建模的强大工具。事实上，状态图已发展到通过专用工具可以将它直接转化成程序代码进行编译的地步。此外，这类工具通常可以创建基于状态图的仿真器并能执行，以便在开发过程中观察其行为。

图 8-9 展示了一个简单的开关式控制的状态图，图中的通道可以是交流电力控制，也可以是继电器。从图中我们可以看到，控制器接受三种类型的输入：ON 命令、OFF 命令和 RESET 命令。你可能还注意到，如果因某部件工作失常而导致出现错误时，逻辑会忽略除了 RESET 命令之外的任何输入。

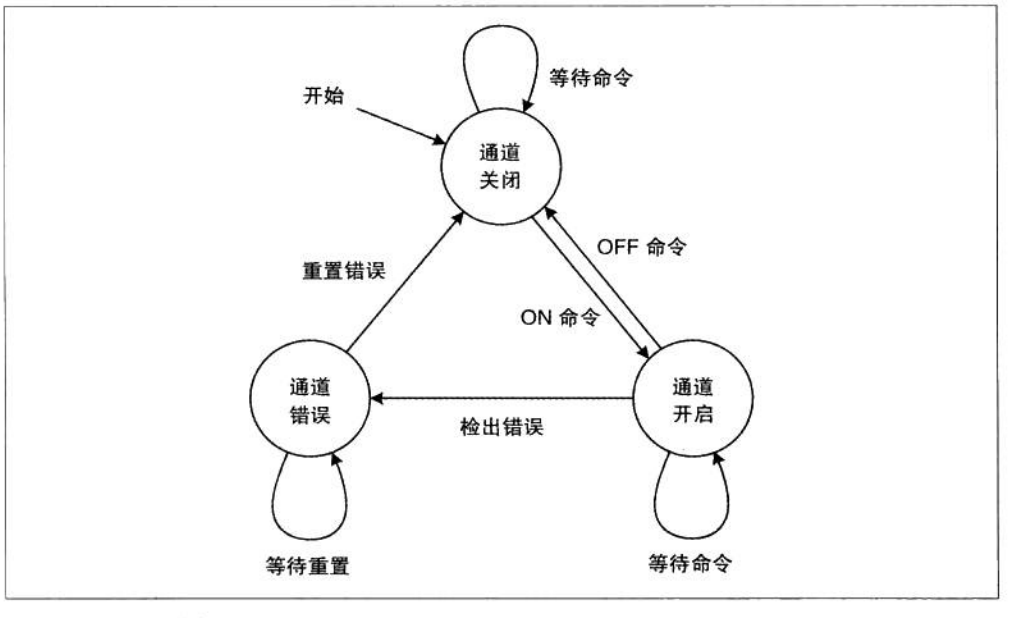


图8-9：状态图示例

消息序列图

消息序列图（The message sequence chart，MSC）由国际电信联盟（International Telecommunication Unio，ITU）维护的 Z.120 文件定义。以目前的形式，消息序列图（MSC）是对多个实体间指令 - 响应式交互进行建模描述的强大工具。UML 中的序列图相当于消息序列图，图 8-10 展示了一个简单的消息序列图。

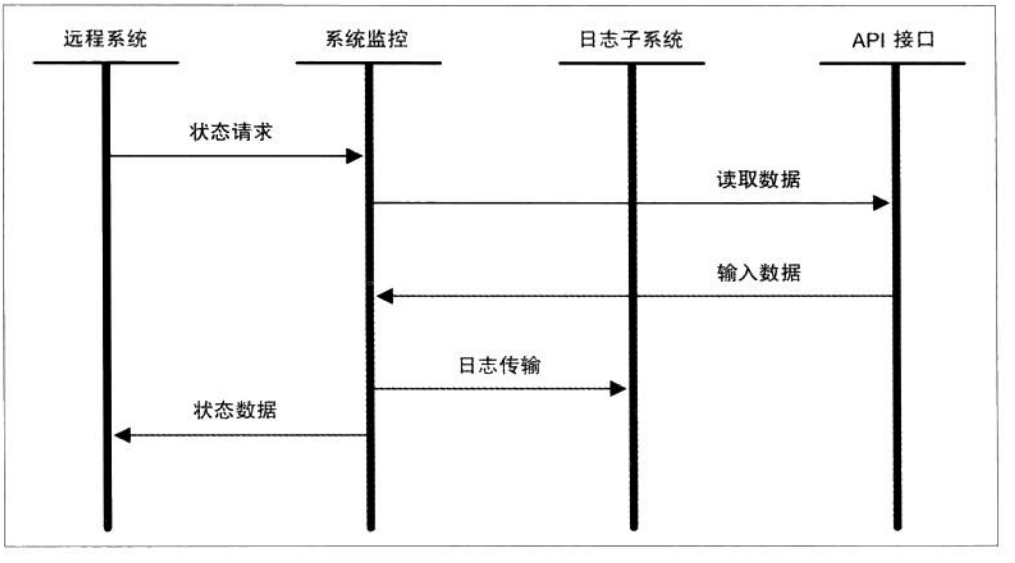


图8-10：消息序列图（MSC）示例

MSC 模型中的实体是指可能产生交互的实体，如用户、流程或服务。交互通常涉及各种形式的交流，如请求、指令或数据。在图 8-10 所示的简单消息序列图中有四个流程：远程系统、系统监控、日志子系统、API 接口。当远程系统要获取数据时，它向系统监视器发出一个请求，由命令接口 API 解析返回系统指令，然后与硬件进行交互，最终返回请求的数据，系统监视器在整个周期内记录所有 API 接口的响应和远程系统的原始请求。

即便像图 8-10 所示那样简单的消息序列图（MSC），依然可以令你快速注意到一些东西。首先，一个状态请求需要一个有限的时间来处理，消息序列图并没有显示有限的时间究竟是多长。在更详细的消息序列图中，可以表示成可接收的响应时间上限。其次，简单的消息序列图中不显示在 API 接口级别出错时的备用响应方案。当然，具有完整语法的消息序列图是可以有效地表述出来的。

Z.120 的最新修订版包含大量的消息序列图语法上的细节，如果对此有兴趣，绝对值得研究通过使用这些图示可以表达什么。当然，有各种商业或是开源工具可以辅助创建消息序列图。

伪代码

也许你对图形没感觉，宁愿处理文本。那么，有一种建模称为伪代码（Pseudocode）。

伪代码是用自然语言书写而成的，基于某种样式将程序或算法描述成人们可直接读的文本（不论哪种语言，只要支持必要的技术词汇就可以，但似乎多数情况都是用英语）。伪代码通常使用编程语言的结构约定，如缩进、if 语句、等等，但它并不打算成为真正的编程语言。

目前还没有标准的伪代码格式，但一般是借用现有的编程语言，C 语言成为通用的语法选择，有些语法元素一般会省略，比如变量声明、函式调用或循环结构内的代码块会被替换为一个或两个自然语言的描述语句。伪代码的主要目的是记录算法的功能或程序，而不是低层如何实施的细节，虽然在某些情况下包含一些细节可能更有用。

在实践中，伪代码随着作者和环境的不同可能有风格的很大差异。它看起来可以极像一个真正的编程语言，也可以近似一篇散文。理想情况下，作者应该努力在两种风格间取得平衡。

图 8-11 显示了一个伪代码示例：一个从底层功能读取模拟数据的函式。正如你所见，ReadAnalog 的主要意图是包装底层 ReadAnalogChan 的参数检查，并取代从驱动 API 返回的代码，将其转化成高级别的程序可以理解的事物（实际上，它简化了返回代码，因为某些 API 调用可能会返回 20 或 30 种不同的数值，都是些只有硬件工程师感兴趣的各种硬件错误警告）。

伪代码应足够详细，以记录程序或算法的基本功能，同时还要确保对编程不熟悉的人也容易阅读。这使依法办事的代码具备可审查性，以确保最终的方案将满足设计规范，没有潜伏缺陷，没有忽略原作者的意图（因为人们往往忽略了自己写作中的错误）。

分而治之

我们已考察过一些创建 SDD 可用的工具，现在让我们来看看如何真正组织一个 SDD。图 8-12 展示了如何根据高层次系统框图创建 SDD 的大纲。

270

```
功能 读取模拟数据
输入：设备，通道
输出：退出码，数据值

开始
    设置 退出码 为 NO_ERR
    设置 数据值 为 zero

    如果 通道 <0 或 通道 > 通道 所允许最大值，那么
        设置 退出码 为 ERR_BAD_PARAM
    结束如果

    如果 设备 <0 或 设备 > 设备 所允许最大值，那么
        设置 退出码 为 ERR_BAD_PARAM
    结束如果

    如果 退出码 是 NO_ERR，那么
        调用 日志读取，返回状态码 及 数据值
    如果 返回状态码错误，那么
        设置 数据值 为 zero
        设置 退出码 为 DEV_FAULT
      否则
        设置 退出码 为 状态码
      结束如果
    结束如果

    返回 退出码 以及 数据值
结束
```

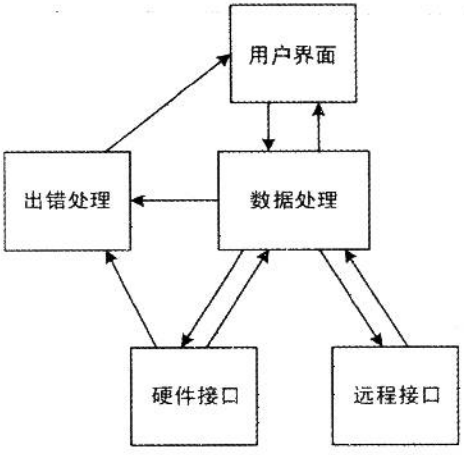
图8-11：伪代码示例

这里的思想很简单：只为系统框图中的主要功能块撰写章节。需要增补的可能是两个图示，或是描述如何将功能模块融入整个系统。详细描述可以包含在适当的章节中，或是每个模块有对应的章节，这些都由你定。

正如之前所述，SDD 不必包含所有细节的大部头文章，但应该包含足够的细节，以便读者可以了解如何实现软件，另外，也展示了软件的开发计划，而不仅仅是不管不顾地开发（hacked）出来。顺便说一下，开发计划绝对不应是“粉饰太平”之类的表面文章，如果不按照计划进行，软件很可能在几个月或是几年后出问题，通过 SDD 的帮助，我们可以事先明确为什么发生问题，并指定数个方案应对。如果软件是产生用以研究的数据，

271

SDD 也可以在数据的准确性不断出现问题时，事先给出方法来处理。SDD 超值发挥之时，就是你在捍卫由软件生成的数据之时。



1.0	系统概述
1.1	系统说明
1.2	系统框图
2.0	硬件接口
2.1	API 扩展
2.2	串口
3.0	数据处理
3.1	控制算法
3.2	趋势监控
3.3	参数读/写
4.0	用户界面
4.1	图形架构
4.2	菜单项
4.3	数据显示
4.4	状态展示
5.0	远程接口
5.1	TCP/IP 套接服务
5.1.1	命令协议
5.1.2	数据格式
5.2	串口
5.2.1	命令协议
5.2.2	数据格式
6.0	错误响应
6.1	失败分析
6.2	出错处理程序
7.0	验证矩阵

图8-12：SDD组成

处理错误和故障

不幸的是，我们没有生活在完美的世界里，我们也没有遇到一个完美的软件开发者，所以，设计文档必须考虑到意外，并说明错误或故障一旦发生，软件将如何处置，这是个有关安全的赌注，即使在软件中对明显的错误都进行了处理，例如，发现和处理超标输入、硬件错误或计划外的事件序列，到最后，仍然会有意外发生。

272

确定潜在的故障

找出潜在故障的方法之一，就是做基本的故障分析。故障分析是一种简单而有效的方式，即先列出所有的系统可能会失败的途径，然后识别软件会如何处理。从本质上说，它是一套“这是什么？”的问题，你最好尝试对每个问题给出一个合理的答案。

表 8-4 显示了一个简单的故障分析表。

表8-4：简单故障分析

故障	原因	对策
硬件输入错误	无效的输入通道	通知用户并记录，系统不会停止
	无效的输入端口	通知用户并记录，系统不会停止
	硬件无响应	通知用户，系统终止
硬件输出错误	无效的输出通道	通知用户并记录，系统不会停止
	无效的输出端口	通知用户并记录，系统不会停止
	硬件无响应	通知用户，系统终止
无效的用户输入	用户提供的参数或命令代码是无效的	通知用户并记录，系统不会停止

这种类型的分析使用电子表格可以很容易进行。随着一步步的增补（系统开始真正工作），表 8-4 有可能被扩大，以涵盖更多可能的失败的案例，但这里的思路是，只要每种错误都能列出来，那么响应策略就可以被定义。审查潜在的故障时，你可能会说：“不，这种情况不可能出现！”正是这种假设，导致多年来各种壮观的软件故障出现。

故障响应

失败的条件一般可以分为三组：致命的、非致命性和微不足道的。重要的是要规划出系统应该如何处理每一类故障。例如，当出现致命的错误时，系统是优雅地尝试自行关闭，还是直接死掉（即崩溃）？如果出现非致命的错误，是个别功能还是所有的功能应该被禁用？系统仍然是可用的吗？怎么算是细小的错误？沉默地记录事件以后，系统应该尝试发送哪种形式的通知？或者只是完全忽略它？等等，所有这些事情，在制定对策时，都应该考虑到。

功能测试

现在我们的设计文件中包含了一些要求和希望，但在写第一行代码之前，我们还需要认真思考有关测试，这里指的是功能测试，而不是单元测试，这两种测试的目标和方法不

273

能混为一谈。单元测试在之后的章节将提及，现在让我们来看看功能测试，看看为什么说它是检验需求的基本方法（可以说是唯一的）。

为需求而测

功能测试，顾名思义，摆在首位的是对系统能否达成需求的功能测试。建立功能测试用例的要求其实很简单，良好的需求设计已经包含了所有必要的测试案例的信息。例如，假设我们有这样一个需求：

> 数据采集子系统软件应能够获取和转换以每秒 1000 个样本为速率的 16 位有符号整数的数据样本。

由此可以得出，我们需要一个输入，以便触发系统生成 -32768 ~ 32767 之间的值，我们还需要能够对一秒内生成的样本进行计数。如果软件有能力来实时确定其实际采样率，并显示它（或至少记录），我们基本上完成了。如果它无法做到，也许这就应该收录为一条派生需求，然后进入 SDD（这也是需求如何被发现的例子）。

但是，即使没有新的派生需求，仍然是可以确定采样率的。也许系统可以伴有时间戳来保存数据样本到一个缓冲的文件，也许不用保存每个样品，但也许会保存每 100 个样本数据。如果日志记录功能纳入软件中，就可能获得必要的核查数据的方式。最后，它可能会连接仪器的硬件，结合数据采集的时间，换算成运行时的转换速率。

顺便说一下，这也就是为什么在开始编码前要进设计功能测试。在测试用例的设计过程中，原有的缺陷就可能忽然明显起来，那么这时修订，远比代码全部完成了再改要轻松得多。在写代码之前编写测试案例，可以照亮思维的暗角，刺激思维方式，终将在后期的开发过程中节省时间（也可能是钱）。

测试用例

功能测试用例描述必要的场景，以便测试系统对特定的要求在特定的环境中如何响应。需求驱动的测试用例却是反过来的，通过实际的测试执行来定义或生成功能。功能测试的情况也可以是用例的一部分，但在本书中，我们将以两个独立的实体来讨论用例和功能测试。

图 8-13 用框图形式展示了需求驱动的测试用例之间的关系，随着程序的执行，最后根据测试数据的通过 / 失败状态来最终确定需求是否达成。

274

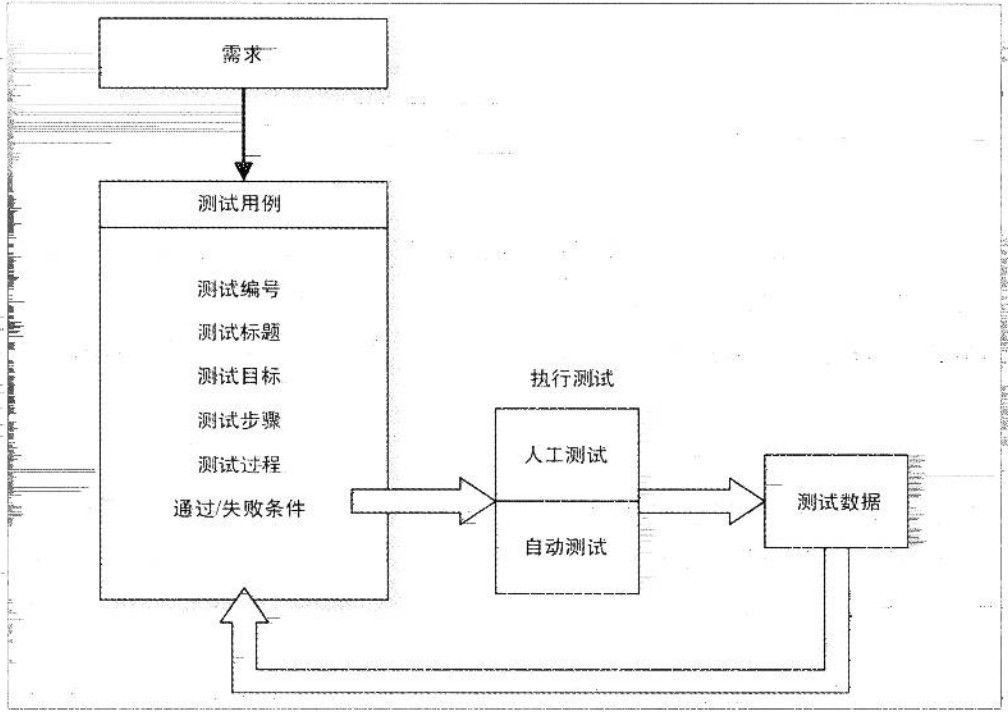


图8-13：需要和测试用例的关系

在某些场景中，你可能会听到“测试程序”或者“测试程序文档”，这就是“脚本”，或者说为了测试要完成的一系列行动。实际上也像演员（测试仪）要遵循的脚本（例如，第1步、第2步、3步等），或者说某种类型方案的测试程序（例如，一个Python脚本，或者一个通信模拟器的控制脚本）。这些构成图8-13中指出的“自动测试”。

将测试分为两个独立但相关的文件——用例和程序，多数情况并不是真的有必要，然而，随着项目复杂程度的增补，将测试分成可执行的（测试程序）以及不可执行的（测试用例）两个部分，就很有必要。因为抽象出来的测试案例以及程序都可以独立编写，互不干扰，面对成批的相关测试时，这么安排绝对有必要。当然，这么做也相应地增加了精力和成本，毕竟现在有两个文件要维护。然而，不论怎么组织，测试都是来“买通”需求的，即根据通过或失败的测试状态来独立判定成果。

275

图8-14展示了测试用例的模板，类似图8-13中所示，测试用例中的测试过程就是案例的一部分。

测试 _id 功能测试标题

测试类型： 人工测试、脚本或扩展驱动？
模拟用值： 是否在虚拟模式中测试？
硬件用值： 是否在真实硬件中运行？
脚本带宽： 如果是自动化测试，哪些脚本要使用？
C 扩展 / 源码：如果测试有支持命令 / 控制资源，它们在哪里？

目标

此处叙述要进行此项测试的理由，以及测试人员须知。不用定义如何算通过 / 失败。

预测步骤

定义测试准备工作，说明所有在测试开始前应该准备好的各方面。

A. 步骤
B. 步骤
C. 步骤
（追加必要条目）

测试过程

此处详细说明测试环节。每一环节可以包含多个步骤，每一步骤都应该是可检验的行动（比如："将开关 s1 置为开启，并确认通路 1 的灯点亮"或是"启动测试脚本，并确认在脚本运行期间没有错误信息抛出"）。

A. 步骤描述
B. 步骤描述
C. 步骤描述
D. 步骤描述
（追加必要步骤）

通过 / 失败

这里定义测试通过 / 失败标准。通过的测试标准表现出来是不超过定义的指标范围或限制，或测试执行期间没有生成任何错误。失败的测试标准可能是函式执行的输出值过高或过低，也可能是在测试过程中发生任何错误，从而抛出错误消息，这都是明显的失败表现。

图8-14：通用测试用例模板

276 同时注意，图 8-14 的模板头部附加了一些文字。我们喜欢在每一个测试案例头部追加一个概要，以说明是什么类型的测试，需要什么方式的编程或脚本支持（即自动化）。摘要还指出测试是否需要在模拟模式下运行软件，或者必须以实际的硬件来执行。请注意，如果“模拟器检验”为“是”，“硬件检验”为“否”，这时意味着该软件内部包含仿真功能，有些功能可能无法和运行态系统一样执行（例如，使用故障注入来测试错误处理）。其次，如果有配套的自动化脚本，将在此进行命名。最后，如果有外部信号源命令或控制信号，也在这里指出。

测试用例可以作为大文档中的章节内容，也可以收集在单独的文件中，这两种方法各有利弊，大文件方便统一查阅。单独的文本允许测试案例一起置于版本管理中进行追踪和修订。

你将需要一些方法来记录每个测试用例的通过 / 失败状态，并标上最后的执行日期。电子表格可以很好地管理这些，在打印时，记得在用例 ID 号前留点空间，以便记录执行结果。

测试错误处理

测试中常常被忽视的方面是故障检测和错误处理。之前我们讨论故障分析时简要谈及了一点，现在，我们将论述为什么不仅要对功能的正确响应进行测试，更要测试系统未按期望执行时将如何应对。

软件错误处理的测试往往比测试成功更加困难，例如，如何在通信通道中策动故障？如何在模拟器中引发模 - 数转换器的故障？能够预想出多少种可能的类型错误，就能在释放到真实世界前对其进行测试，从而保证所有人的利益。

测试错误处理中常用的方法被称为“故障注入”。顾名思义，这涉及在代码中纳入某些功能，以允许测试可达。简单地说，就是设置会引起故障的内部变量。做到这点的方法之一就是通过网络套接字连接，在连接前，必须通过配置参数才可以启用这种专用通道。串口连接也可担任这个角色。

如果测试涉及外部设备，只需拔下它就等于注入了故障，但只能引发一种类型的故障，这种做法将不会注入因数据损坏而造成的错误，它只会模拟通信的缺失。

277 重要的是牢记测试系统错误的响应目的，不是来模拟错误条件自身，而是激发对应处理错误的代码。我们发现创建全局状态变量集很有用，这个集合可以内置在记录软件执行错误的条件模块，例如，底层硬件数据读取模块中，在其成功或失败时可记录对应的全局状态变量，它不应该覆盖现有的错误指示，然后可以检查代码的错误状态，以便进一

步从硬件获取数据，如果出现错误，则触发相应的行动。在这种方案中就可以简单地通过修改全局变量的设置来引发对应的错误条件。底层代码不会屏蔽它，而上级代码将看到并在发生了错误时回应。只有最底层的代码部分无法通过这种方法进行测试。但它们可以被分开处理，使用其他测试技术进行测试。

错误处理和错误注入就介绍这么多，稍后在有关章节再探讨。当然，要注意：总会有不不止一种方法去做，这里描述的只是其中一个办法。

回归测试

回归测试背后的想法很简单：关注的问题是“近期的变化是否破坏了什么？”任何时候，你对代码进行了更改，无论是为了修复一个现有问题，还是实现新的功能，你都应该做回归测试。为了执行回归测试，通常对修改后的软件运行部分或全部现有的功能测试。局部回归测试将侧重于变化中的特定软件功能，一个完整的回归测试将执行所有可用的功能测试。

对软件进行更改必然会频繁地引入缺陷，这是既定事实。可能是激发了先前处于休眠状态的错误，或是由于创建了新的执行路径，从而包含了以前未知的缺陷。“修复”行为本身总是不尽如人意，感觉软件发“脆”，那是因为软件总是有没有考虑到的超标、不正常、非期待的输入。回归测试可以帮助发现这类问题，以便在它们变得更加混乱或是发布去前及时被修订。

回归测试不仅只能同功能测试结合，它也适用于单元测试。回归测试在多数情况下是可以自动运行的，这意味着回归测试可以作为完整的测试套件反复运行。比如，在每晚的自动构建中，这样，每日清晨的第一件事就是查阅测试简报，明确当天要修订什么。

进展追踪

278

如果在整个项目的生命周期中，对功能测试结果进行追踪，你可能获得类似图 8-15 所示的图形。这是相当典型的依赖功能测试验证成效的项目，开始时软件有点粗糙，然后随着项目的进展不断改善。如果该项目的开发活动还包括广泛的单元测试和代码复查，这种情况可以得到改善，功能测试整体失败率水平应比这里显示的要呈现更快的下降趋势。但是，在任何情况下，最终目标都是追求在项目结束时，达到测试的零失败和 100%的需求覆盖。

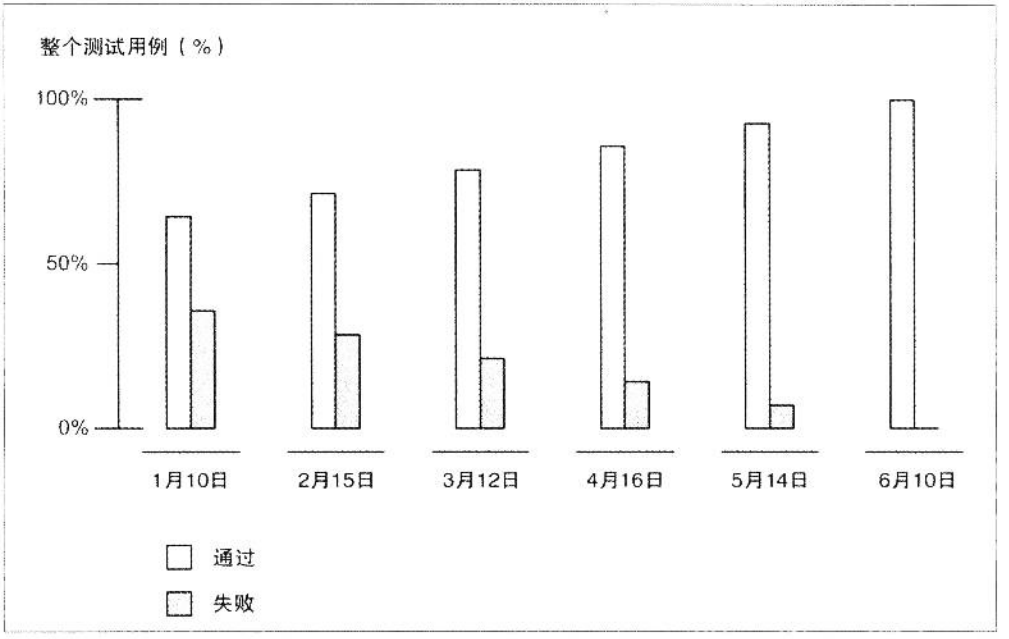


图8-15：功能测试失败与通过率的变化

实施

现在，终于来到了最有趣的部分：编写软件。我们已经讨论了需求、设计和功能测试，现在我们总算可以把注意力转向代码的编写活动了。

代码应该做什么，已经在 SDD 中被详细定义了。本节讨论如何对代码进行组织、格式化、审查，并使用单元测试方法进行测试。这些细节有时会作为鸡毛蒜皮的小事被忽视，但这是错误的，很可能埋下问题，直到开发完成几个月甚至几年后爆发，漫不经心的编程造就了那些悲惨生活，即使软件已经发布，但是，一旦有需要，你还是得独自回来重新理解自己写的代码。

279

代码风格

几乎每种编程语言都有自己的代码风格，有时同一编程语言可以有不同的风格。甚至不同的公司、政府机构、研究团体、开源项目和个别开发商也都有自己的代码风格。在某些情况下，代码风格可能是正式的文件，而在多数情况下，代码风格只是随着时间的推移和发展被反复使用并最终成为习俗的东西。

在某些情况下，编码风格可能被当成一组可核查的编码要求来执行，这在高可靠性或安全性软件的生产过程中是典型要求。已有的各种工具可以根据一组编码规则分析源代码，并在代码偏离规则时生成消息。对于本书中涉及项目的严谨程度而言，我们不认为有必要提倡使用工具。但是，我们认为定义或通过一个良好的编码风格，建立一些基本规则，然后坚持贯彻很重要。

为什么编码风格是重要的

始终遵循一个特定编码风格的一个好处是，它会使软件代码拥有清晰和易于理解的格式，形成一致的风格。这通常并不只是为了美观，它使完成的代码除原作者以外的人也可以轻易读懂和降低审查的成本。

另一个好处是，它定义的编程风格可以避免语言潜在的危险的功能。例如，在C语言中可以使用指针参数直接指向已分配给该程序的内存，这对很多应用软件而言是好技巧，但当软件运行在内存受限的嵌入式系统时，编码规则通常会禁止使用动态内存分配，以避免致命的指针错误和内存“泄漏”。在嵌入式系统里运行，周围不会总有人来按复位按钮，所以，这些情况可以引发的问题绝不仅仅是程序崩溃了。另一个例子是，Python语言允许程序员动态创建字典对象，并通过参数和返回值，随时动态地修改它们。结果在字典对象经过多个函式或方法调整后，字典实际已经很难确认它是从哪里来的，以及它在做什么。

采纳现有的编码风格准则

280

Python.org的爱好者已经发布了在线文档，并定义了一套基本的编码风格准则。主文档为PEP-8。“Python代码风格指南”；PEP-257，“Docstring公约”也包含有用的信息。

风格主要是有关布局、名称和代码组织的一致性约定。引用PEP-8：“风格指南事关一致性。一致性对于风格指南是重要的，在一个项目内保持一致性更加重要，在模块或功能内部保持一致性则是最重要的事。”

如果你是Python新手或普通的程序员，我们建议你尽可能地坚守PEP-8的约定，明确什么有用，什么没用，这是最好的办法。当你获得或是拥有了足够的经验时，你可能会使用自己定制的合理风格准则了。

虽然总是有一些C编程风格准则不断漂出来，而且看起来都有很大变化，但是，有些程序员总是使用古老的K & R的“ANSI C”参考书作为自己的风格指南。在其他情况下，风格指南可能已经存在，或可能有明确的编码风格的要求。那么，请确保你选择的风格与现行的保持一致。

组织你的代码

如果你有一个良好的 SDD 作为开始，那么代码显而易见的应该根据 SDD 来组织。当然，也有一些基本的方针可以运用，使代码保持整齐、整洁，便于阅读。

首先，要避免将一切塞入一个文件的诱惑。划分成多个只包含相关功能或类的模块不会被惩罚，其实这样组织是好事，因为有助于理解软件，而且也令改变更加容易，因为在修订相关模块时，并不会影响其他部分的代码。所以，即使 SDD 暗示 20 种不同的功能要安置在同一个模块，你可能也要考虑应该分解成两个、三个或是四个模块，每个模块只包含紧密关联的功能。

Python 内置有包和模块的组织方式，有助于确保代码的整齐，如果你不熟悉相关概念，应该花一点时间阅读相关文档。C 语言没有相应的源代码组织方式，但也能很方便地将代码模块化到头文件、源文件或是库模块中。

图 8-16 用框图展示了仪器应用系统的代码组织方式。这只是其中一种方法，其他方法也是可能的。其中的要点是，代码依据水平切分的功能来组织，图中的每个块仅涉及指定的群体或是某阶段的功能。

281

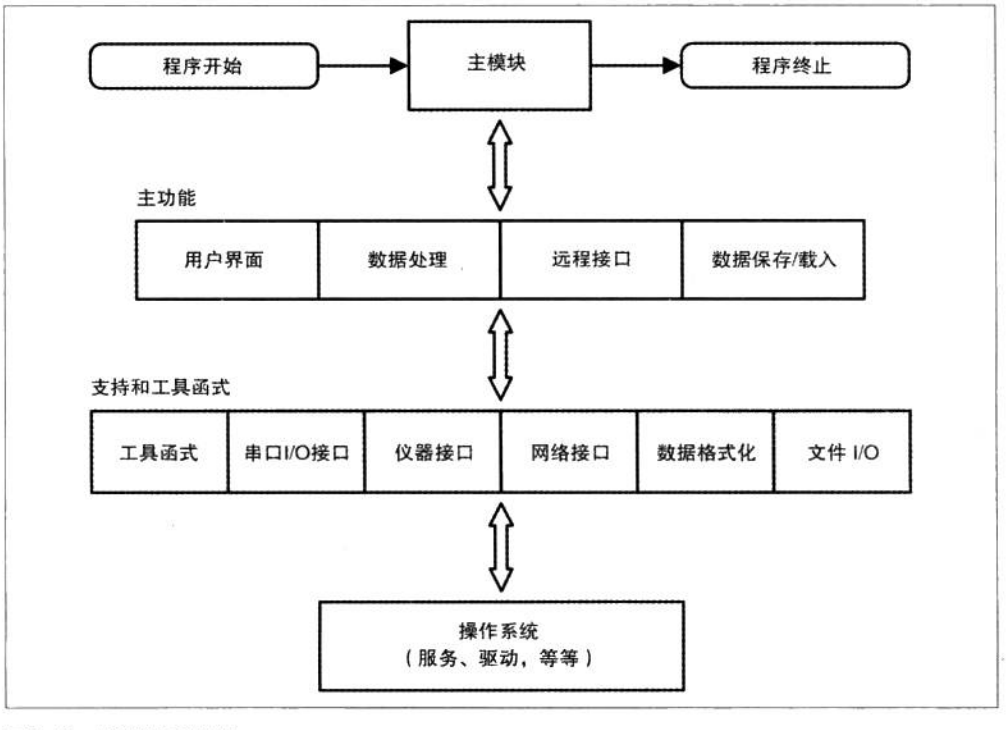


图8-16：代码组成案例

代码复查

代码复查非常有用，尤其是在帮助团队维持代码和文档的一致性，以及帮助发现原作者一直忽视的或未意识到的错误时。但是，代码复查不能替代测试，这两项活动好像枚硬币的两面，缺一不可。

如果处理不当，代码复查也可能是一项艰巨的苦差事。有一次，我们参加了一系列的代码复查，竟然历时近八天。为什么呢？因为在参加复查前，其他人并没有事先查阅对应的 SDD，所以他们不知道结构，也不知道功能的意图，所有的小细节都要逐一向他们解释，就好像重新进行详细设计。好在毕竟都是我们必须回顾的代码，总体而言也不算浪费时间。 282

当仅有两个人写代码时，复查可以很简单，也可以成为正式的会议，包含主持人、一名秘书和一些评论，加上软件的作者来推进。这一切都取决于软件的复杂度、重要程度，以及是否一开始就有 SDD。对于本书中的大部分项目，其实只要有人认真看一下代码，就足够了。即使你已离开，其他人（可能都不是一名程序员）也可以轻易理解你可能忽略了什么事情。

下面是我们认为进行代码复查时要注意的几条非常有用的规则：

- 每位参与者都应该事先熟悉代码。如果有人没有做到，则取消复查资格。
- 每位参与者都应该事先查阅代码对的 SDD 章节。
- 关注适合进行代码复查的方面：风格一致性、明显的逻辑错误（例如，if-elif-else 结构问题）、未初始化的变量、不匹配的 malloc 和 free 调用（C 代码），或放置不当的类模块的范围变量（反之亦然）等，这只列举几个例子，具体的应该在对应的项目中事先约定。
- 使用一个清单并坚持记录。代码复查清单应包含所有的关键信息，在复查期间，尽量不要偏离检查列表太远，因为这会浪费时间。

最后一点尤为重要。一个基本的清单通过帮助审查期间集中注意力，可以节省时间和精力。它提供了一个方便的方法来保持跟踪审查结果。一个最小的清单看起来可能像图 8-17 那样。

1. 是否按照 SDD 设计进行了正确实施？
2. 是否有任何明显的拼写错误和错别字？
3. 软件是否遵守项目的编码准则和公约？
4. 各种注释有意义吗？都有必要吗？注释足够或太多？
5. 数据对象（例如，结构、字典、数组等）被充分定义了吗？
6. 如果变量根据字面含义来使用，是否正确？
7. 在多个地方被使用的同一常数，是否将它们统一在共同的位置定义？
8. 是否有代码重复明显的部分？是否可以封装为函式或方法？
9. 是否有代码永远不会被执行？
10. 所有定义过的变量和常量都真正使用了吗？

图8-17：最小的复查检查列表

让我们进一步理解图 8-17 中条目的含义：

1. 是否按照 SDD 设计进行了正确实施？

   代码必须尽最大可能遵循 SDD 的设计（当然，这取决于 SDD 本身有多靠谱）。假设我们的目的是符合需求（初始或派生），代码不遵循 SDD 是最有可能造成需求不被吻合的原因。不论怎么花里胡哨或是要酷，只要没有吻合用户的需求，最好的结局也是浪费程序员时间开发出无用的软件。在最坏的情况下，无文档、不符合需求的特性，可能会向代码引入各种不良的缺陷，即使用单元测试也无法察觉。

283 2. 是否有任何明显的拼写错误和错别字？

   虽然单元测试可能发现一些拼写错误的变量名，但是无法捕获所有类似的错误，特别是像 Python[译注1] 这种允许代码来动态创建变量的语言。一个变量可能已在另一个模块、函式或方法中被定义，即使拼写错误的代码仍可能通过单元测试，比如，在模块或是类中有变量只用于追踪内部状态，而且只能通过类或是模块之外的代码进

译注 1：除了项目足够简单之外，Python 代码行文如此像自然英语也是有这种效果的重要保证。另外，Python 使用缩进，而不是其他语言选择特殊的字符来划分语法结构，也使代码更加干净、有层次感，更有助于理解。当然，这种设计也引来了大量的非议，不过，仁者见仁，智者见智，译者本身的感受是非常舒服的。

行读取和设置，如果拼写错误，那么它可能永远不会被改变，然而外部调用者也永远意识不到其中的问题。

3. 软件是否遵守项目的编码准则和公约？
   这是相当明显的，并且在审查期间会更加明显。如果团队选择了一个编码风格，那么它应该始终用于保持一致性。
4. 各种注释有意义吗？都有必要吗？注释足够好还是太多好？
   注释有时被用来掩盖糟糕的编码（或许还没有理解代码的含义，或代码写得有点夸大），注释也常常被滥用（例如，注释 a +=1。实际上这种情况很常见）。注释只应出现在需要解释的地方，包括功能或方法应该做什么。 284
5. 数据对象（例如，结构、字典、数组等）被充分定义了吗？
   所有的数据对象都应该有一个注释，除非它们的使用是显而易见的，尤其是对于 Python 中字典或复合结构的对象。用在循环中作为简单索引计数器的变量可能并不需要一个注释。
6. 如果变量根据字面含义来使用，是否正确？
   文本常量的值应该被“一成不变”地使用。所以需要从一开始就正确，错误的输入可能导致软件发生严重的意外行为。
7. 在多个地方被使用的同一常数，是否将它们统一在共同的位置定义？
   如果同一常量在两个或两个以上的地方被使用，就应该转换成名字具有全局唯一性标识的变量，而不是“赤裸裸的数据”。这使得开发者能够在一个地方进行修订，所有使用的地方都得到统一变更。另一种方式是尝试人工追踪常量可能出现的每一个地方，随着数量的增加，总是会有错过的现象，从而引发无数悲痛。
8. 是否有代码重复明显的部分？是否可以封装为函式或方法？
   如果代码中在多个地点重复包含本质上相同的功能代码，那么，该功能应该是一个可以随时调用的函式或方法。
9. 是否有代码永远不会被执行？
   所谓的“死码”正是要避免的东西。很难确定它真正应该做什么，有可能后来有人无意间激活了这段代码，即使完全不理解它为什么存在，从而混乱了软件，或是为了一个只有在非常独特的情况下才可能发生的，设置了这一未经考验的执行路径。如果不准备使用它，就清除它，要真正确保所有的代码执行，需要使用代码覆盖工具，不过在复查时人工可以注意到这种“死码”，是一个良好的开端。
10. 所有定义过的变量和常量都真正使用了吗？
    如果一个模块、函式、方法或类中定义了变量，它就应该被使用。如果从始至终都没有使用过，那就像死代码段一样，应该被清除。

当然，你可以使用比这个清单更多的检查条目，但我们觉得这是一个足够好的开端。它

足够短，但已经可以切实检查自己的代码（如果没有其他人来做复查）。当然，有第二双眼睛（或更多）总是更好的。

285

单元测试

单元测试的目的是为通过检验软件的每一小功能的质量，从而确保整体的单元可以是一个函式、一个类或在类中的单个方法，这通常是可执行代码的最小逻辑单元。例如，有一个函式，它接受一个从外部仪器采集到的原始二进制数据，执行一些扩展和范围检查，并由返回代码返回一个二进制值，这就可以作为一个单元，用合理和超标的输入值来完成测试。当单元被测试时，代码覆盖率也就可以确定了。这在 C 语言中较困难，特别是当代码作为低层扩展模块时，但它是可行的，不过，在本书中，我们将坚持用 Python 进行单元测试。

定义单元测试

单元测试并不一定直接映射到一个需求，但它是对 SDD 的直接映射。假设 SDD 定义了一个数据衡量函式，则代码类似：

```
def ScaledInput(data):
    rc = NO_ERR

    scaled_data = data

    if data >= DATA_MIN and data <= DATA_MAX:
        scaled_data = (data * data_scale) + data_offset
        if scaled_data > SCALE_MAX:
            scaled_data = SCALE_MAX
            rc = ERR_MAXSCALE
        elif scaled_data < SCALE_MIN:
            scaled_data = SCALE_MIN
            rc = ERR_MINSCALE
    else:
        rc = ERR_OVER

    return (rc, scaled_data)
```

对我们而言，可以假设 data_scale 和 data_offset 是在其他地方设置的（也许是软件初始化时）。DATA_MIN、DATA_MAX、SCALE_MIN、SCALE_MAX、NO_ERR、ERR_MAXSCALE，以及 ERR_MINSCALE 是“常量”，都是硬编译的数值，并在执行过程中不发生改变，当然，也可能在其模块中会有变动。

最后，请注意，返回的数据值是由原始输入的值初始化而成的。如果发生值范畴的错误，该函式将依然有数据返回，但它不会对数据进行缩放或以其他方式调整。如果发生缩放

错误，返回的数据将是最大或最小允许值。这也是为什么返回的是二元组，包含用以进行功能检查的返回码（RC）来识别发生了什么。

单元测试构造所有可能的输入用于强制遍历所有可能的执行路径。实现这一目标的方法之一是创建一个测试数据表。在 ScaledInput() 的情况下，我们可以看到，有三个明显的输入案例：太低、太高和范围之内。表 8-5 显示了此函式对应的三个测试用例。 286

表8-5：ScaledInput()测试用例

测试	输入	RC（返回码）	输出
1	data < DATA_MIN	ERR_OVER	原始输入数据
2	data > DATA_MAX	ERR_OVER	原始输入数据
3	min <= data <= max	NO_ERR	处理后的数据

这是一个很好的开始，但它不完整：没有测试 scaled_data 可能超过比例变换的限制，所以，原有的三个测试用例无法覆盖到所有的执行路径。表 8-6 显示了这个看似简单的功能所需要的完整的测试用例。

表8-6：ScaledInput() 测试用例（完整）

测试	输入	数据缩放	数据偏移	RC（返回码）	输出
1	data < DATA_MIN	1.0	0.0	ERR_OVER	原始输入数据
2	data > DATA_MIN	1.0	0.0	ERR_OVER	原始输入数据
3	min <= data <= max	2.0	0.0	NO_ERR	处理后的数据
4	min <= data <= max	10.0	0.0	ERR_MAXSCALE	SCALE_MAX
5	min <= data <= max	-10.0	0.0	ERR_MINSCALE	SCALE_MIN
6	min <= data <= max	1.0	1000.0	ERR_MAXSCALE	SCALE_MAX
7	min <= data <= max	1.0	-1000.0	ERR_MINSCALE	SCALE_MIN

data_scale 和 data_offset 的实际值当然取决于 SCALE_MAX 和 SCALE_MIN 的值，但对我们来说，可以假设它们都应匹配表中定义的测试用例结果。这点也适用于 DATA_MIN 和 DATA_MAX。

最后两个案例虽然看起来真的没必要，它们只是补全了所有可能的值来覆盖函式功能，我们想指出的是，表格也描述了函式预期的行为，其中包括 RC（返回码）不是 NO_ERR 的情况。一个单元测试的返回值只要不吻合表中的期待值，就算是失败，不仅仅是 RC 有返回错误码，每个单元测试用例文档都应该包含类似表 8-6 中所有测试成功的必要条件的组合。

无正当理由，纯粹浪费时间的创意测试

单元测试即使没有同功能测试一样直接映射到对应的需求，仍需要有某种文件来形容它们的目的是展示和如何达成。无论是单元测试还是功能测试，没有对应的存在理由和通过/失败标准描述，通常只是浪费时间，即使只是最简单的测试。如果你不知道一个测试应该证明什么，那你怎么能判定运行是否成功？

这里有个值得重视的箴言：包含无文档的测试代码比根本不包含测试代码的代码要更加烂。为什么？未经测试的代码，你可以相当肯定，它可能含有错误，而代码包含无意义的测试可能会麻痹你，即使代码含有严重缺陷的可能性仍然很高，你也有种虚假的安全感。总之（其他许多人多次重复说过类似的）：为测试而测试比没有测试还要糟。

执行单元测试

有了测试用例表，现在我们可以编写本单元的实际测试了。幸运的是，Python 包括一个内置的单元测试模块，甚至包含代码覆盖能力（正如 Python 的用户常说的：电池内置）。下面是一个称为 InputUtils.py 的模块，它只包含一个函式 ScaledInput()。我们没有理由不包含更多的实用函式来进行更好的输入验证和处理功能，但现在，先这样就好。

ScaledInput() 使用的“常量”在同一模块中定义，虽然在现实中，你可能希望封装到一个单纯的定义模块，使它们可以轻易地在其他应用程序中导入。虽然我们知道，通配符导入一般是要避免使用的，但是，在这里，我们没有对所有的变量名使用下画线前缀，在实践中这样做是一个好主意，因为你不知道后来的人会怎么使用你的代码。

先来看一下模块代码：

```
""" InputUtils.py

    收集各种输入验证和处理函式。

    目前仅有 ScaledInput()
"""
NO_ERR = 0
ERR_OVER = -1
ERR_MAXSCALE = -2
ERR_MINSCALE = -3

DATA_MIN = -10.0
DATA_MAX = 10.0
```

```
SCALE_MIN = -50.0
SCALE_MAX = 50.0

data_scale = 1.0
data_offset = 0.0

def ScaledInput(data):
    rc = NO_ERR
    scaled_data = data

    if data >= DATA_MIN and data <= DATA_MAX:
        scaled_data = (data * data_scale) + data_offset
        if scaled_data > SCALE_MAX:
            scaled_data = SCALE_MAX
            rc = ERR_MAXSCALE
        elif scaled_data < SCALE_MIN:
            scaled_data = SCALE_MIN
            rc = ERR_MINSCALE
    else:
        rc = ERR_OVER

    return (rc, scaled_data)
```

现在我们只需以某种方式运用表 8-6 中列出的所有的测试参数来执行 `ScaledInput()`。Python 的单元测试库（`Unittest`）是首选工具。

在 Python 中包含了所有相关的库，甚至有几种方法进行单元测试，这里只展示其中一种。我们比较喜欢这种方法，因为它简单、直白（至少对我们来说）。让我们来看看代码，它位于 `test_001.py` 模块中，然后运行它，看看能做什么：

```
import unittest
import InputUtils as UUT

class test_001_UT(unittest.TestCase):

    def test_001_UT_01(self):
        UUT.data_scale = 1.0
        UUT.data_offset = 0.0
        data = -11
        rc, sdata = UUT.ScaledInput(data)
        assert rc == UUT.ERR_OVER
        assert sdata == data

    def test_001_UT_02(self):
        UUT.data_scale = 1.0
        UUT.data_offset = 0.0
        data = 11
        rc, sdata = UUT.ScaledInput(data)
```

```
            assert rc == UUT.ERR_OVER
            assert sdata == data

        def test_001_UT_03(self):
            UUT.data_scale = 2.0
            UUT.data_offset = 0.0
289 >       data = 5
            rc, sdata = UUT.ScaledInput(data)
            assert rc == UUT.NO_ERR
            assert sdata == data * UUT.data_scale

        def test_001_UT_04(self):
            UUT.data_scale = 10.0
            UUT.data_offset = 0.0
            data = 10
            rc, sdata = UUT.ScaledInput(data)
            assert rc == UUT.ERR_MAXSCALE
            assert sdata == UUT.SCALE_MAX

        def test_001_UT_05(self):
            UUT.data_scale = -10.0
            UUT.data_offset = 0.0
            data = 10
            rc, sdata = UUT.ScaledInput(data)
            assert rc == UUT.ERR_MINSCALE
            assert sdata == UUT.SCALE_MIN

        def test_001_UT_06(self):
            UUT.data_scale = 1.0
            UUT.data_offset = 1000.0
            data = 10
            rc, sdata = UUT.ScaledInput(data)
            assert rc == UUT.ERR_MAXSCALE
            assert sdata == UUT.SCALE_MAX

        def test_001_UT_07(self):
            UUT.data_scale = 1.0
            UUT.data_offset = -1000.0
            data = 10
            rc, sdata = UUT.ScaledInput(data)
            assert rc == UUT.ERR_MINSCALE
            assert sdata == UUT.SCALE_MIN


    suite = unittest.TestLoader().loadTestsFromTestCase(test_001_UT)
    unittest.TextTestRunner(verbosity=3).run(suite)
```

首先要注意的是，它仅导入了两个模块：Python 的单元测试库（unittest）和名为 UUT 的目标模块 InputUtils，就能进行测试。然后是声明继承自通用测试的类作为测试用例

模板。在这个新的类中，我们创建了七个成员方法，每一个方法对应表 8-6 中设定的测试案例。

每个测试案例方法接收类似的一组参数：数据、缩放、偏移、量参数。变量 data_scale 和 data_offset 驻留在 UUT 模块中，而输入的数据在每个测试案例方法中都直接传递到 ScaledInput()。

测试用例的方法在初始条件设置之后，调用 ScaledInput()。返回代码和处理后的数据，捕获为 RC 和 SDATA。

使用 Python 的断言（assert）语句返回测试结果，如果表达式为 True，则断言通过；只有案例中所有方法的断言都为 True 的情况下，测试才算通过。输出结果看起来像下面这样：

290

```
>>> import test_001
test_001_UT_01 (test_001.test_001_UT) ... ok
test_001_UT_02 (test_001.test_001_UT) ... ok
test_001_UT_03 (test_001.test_001_UT) ... ok
test_001_UT_04 (test_001.test_001_UT) ... ok
test_001_UT_05 (test_001.test_001_UT) ... ok
test_001_UT_06 (test_001.test_001_UT) ... ok
test_001_UT_07 (test_001.test_001_UT) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK
```

断言与 assertEqual() 及其朋友们

单元测试库（unittest）专门针对单元测试使用其 TestCase 类中的断言（assert）类型方法来收集测试结果。两个最常用的方法是 assertEqual() 和 assertNotEqual()。作为示例，我们可以用这些方法来重写 test_001_UT_07()：

```
def test_001_UT_07(self):
    UUT.data_scale = 1.0
    UUT.data_offset = -1000.0
    data = 10
    rc, sdata = UUT.ScaledInput(data)
    self.assertEqual(rc, UUT.ERR_MINSCALE)
    self.assertEqual(sdata, UUT.SCALE_MIN)
```

为什么用 assertEqual() 而不是单纯的 assert？有两个可能的原因：第一，如果代码运行时启用优化（-O 命令行开关），assert 语句会被忽略；第二，assertEqual() 可能会产生一些 assert 不会反馈的额外信息（至少，没有额外的一些工作，我们将演示这种情

况)。让我们来观察测试用例 07，使用普通的 assert 语句，遇到一个错误导致最终的断言失败时，运行后输出的结果会是下面这样的：

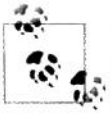
```
>>> import test_001
test_001_UT_01 (test_001.test_001_UT) ... ok
test_001_UT_02 (test_001.test_001_UT) ... ok
test_001_UT_03 (test_001.test_001_UT) ... ok
test_001_UT_04 (test_001.test_001_UT) ... ok
test_001_UT_05 (test_001.test_001_UT) ... ok
test_001_UT_06 (test_001.test_001_UT) ... ok
test_001_UT_07 (test_001.test_001_UT) ... FAIL

======================================================================
FAIL: test_001_UT_07 (test_001.test_001_UT)
----------------------------------------------------------------------
Traceback (most recent call last):

291>    File "test_001.py", line 61, in test_001_UT_07
          assert sdata == data
AssertionError

----------------------------------------------------------------------
Ran 7 tests in 0.050s

FAILED (failures=1)
```

虽然这表明测试案例 07 失败了，但它不会告诉我们为什么。如果我们使用 assertEqual() 断言语句，并再次运行测试，它仍然会失败，但 assertEqual() 方法会显示失败时两个变量的值以协助检查：

```
>>> import test_001
test_001_UT_01 (test_001.test_001_UT) ... ok
test_001_UT_02 (test_001.test_001_UT) ... ok
test_001_UT_03 (test_001.test_001_UT) ... ok
test_001_UT_04 (test_001.test_001_UT) ... ok
test_001_UT_05 (test_001.test_001_UT) ... ok
test_001_UT_06 (test_001.test_001_UT) ... ok
test_001_UT_07 (test_001.test_001_UT) ... FAIL

======================================================================
FAIL: test_001_UT_07 (test_001.test_001_UT)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_001.py", line 61, in test_001_UT_07
    self.assertEqual(sdata, data)
AssertionError: -50.0 != 10

----------------------------------------------------------------------
```

```
Ran 7 tests in 0.000s

FAILED (failures=1)
```

现在我们可以看到具体哪儿出了问题，并确切到是哪一行的代码。



为公平起见，我们得指明，创建这个测试案例，是为了说明在 ScaledInput() 函式处理的数据值和原数据输入参数间进行比较会引发错误，而不会进入 UUT.ERR_MINSCALE。

虽然目前已指出 assertEqual() 代替原始断言（assert）的好处，但我们认为，归结到最终还是要看你需要怎么完成你的单元测试。如果你是从未准备根据单元测试来优化你的代码，或对于你来说看到每次失败的具体参数并不是令人信服的需求，那么，继续使用原始的 assert 语句吧。虽说只是一种观察失败的形式，只需要多输入一些文字，就可以获得更多，何乐而不为？

正式的 assert 语法如下（在 Python 2.6.5 文档的 6.3 节所述）：

292

```
assert_stmt ::= "assert" expression ["," expression]
```

第二个表达式可用于消息追加到异常的输出，如：

```
>>> var1 = 1
>>> var2 = 2
>>> assert var1 == var2, "%s != %s, at least not in this universe!" % (var1, var2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: 1 != 2, at least not in this universe!
```

以上只算是一次简单的鸟瞰，展示单元测试（unittest）可以做什么。当然，Python 文档中还有更多关于单元测试和它的方法的信息。

希望大家明白，单元测试是作为正常的软件开发应做的事。每当完成了新功能或方法后，你都应该立即为之创建对应的单元测试，并经常运行。在某些环境中可能会设专人维护功能测试和测试环境，但代码编写者是唯一有资格创建单元测试的人。这比其他人来揣摩代码之后，再为之创建测试案例要靠谱得多。

代码覆盖率

虽然单元测试是用于执行基本的底层测试，它也可以同时达到另一个同样重要的目的：代码覆盖分析。

如果你不熟悉代码覆盖率分析，也不用担心，它其实很简单，就是要求每行程序代码必须是可执行的。换句话说，不希望在软件代码中存在任何将来会出现的问题——“死码”。

代码覆盖分析还能告诉你，单元测试是否是完整的，不完整的单元测试可能在你的软件中遗漏包含不良信息的暗角，所以测试时，你就得想办法确保触摸到一切。

图 8-18 展示了 ScaledInput() 函式的流程图。如果和表 8-6 列出的测试情况对比，你应该能够很容易地识别每个测试用例的执行路径（只要根据返回代码）。为了更好看，我们改变了一些变量名。

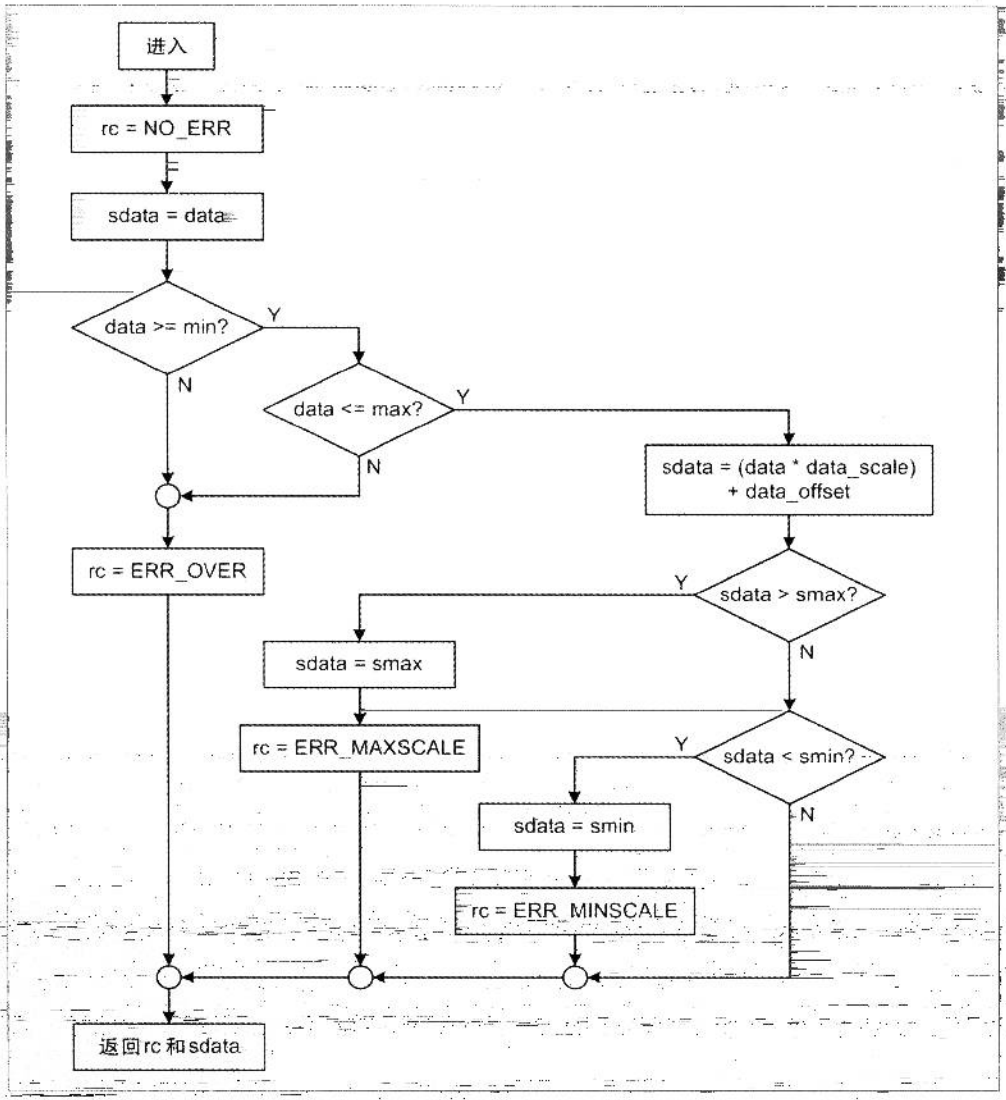


图8-18：ScaledInput()函式流程图

正如我们前面讨论过的，发现死码的方式之一是通过代码复查。另一个方法是使用工具监控软件的执行，并跟踪报表记录哪些已执行，哪些还没有。我们推荐使用 Ned Batchelder 发布的代码覆盖测试工具（*http://nedbatchelder.com/code/coverage/*）。

一旦安装完成，只需在命令提示符下输入 `coverage` 参数并指定源文件名（注意，不是 python coverage，而是 coverage 命令），就能运行它。此命令将执行指定的程序模块和输出覆盖率统计数据。例如，使用我们的示例单元测试，输入：

293

```
coverage -x test_001.py
```

294

`-x` 命令行开关告诉 `coverage` 执行程序并收集覆盖数据，也可用新的 `run` 参数。在命令行输入 `coverage help`，可以查阅新风格参数的形式，`coverage help classic` 则是查阅旧式的命令行参数，我们比较喜欢旧风格的参数。输出结果应完全像我们以前使用 `unittest` 模块运行时看到的那样（直接输出到标准输出），但是 *coverage* 能为我们生成外部文件（覆盖度量数据文件）来记录汇报。

使用 *coverage* 的 `report` 功能，就可以看到一个很不错的代码覆盖率数据表。`-m` 参数开关控制是否输出那些没有被执行的代码行数统计信息：

```
coverage report -m
```

输出如下：

```
Name          Stmts  Exec  Cover  Missing
-------------------------------------------
inpututils       23    23   100%
test_001         54    54   100%
-------------------------------------------
TOTAL            77    77   100%
```

这表明，对于 `ScaledInput()` 在图 8-18 中每一个可能的路径都覆盖的情况下，所有的代码都能得到测试。所以，这里没有死码！ *coverage* 甚至还追踪了测试包本身（`test_001`），当然你可以在分析中忽略这部分，具体的参考 *coverage* 的文档。

连接到硬件

要想协同硬件良好地运行，软件本身靠谱是整个工程中必不可少的一个点。希望这一点越早考虑到越好，因为你肯定不愿意到最后才发现硬件并没有按照你的设想工作，或是你的代码总是有这样或那样的问题，直到硬件接口终于激活（我们曾经一直面对这类问题，说实话，它们一点也不好玩），这些 bug 都是些类似数据表述、位顺序、时间值或是其他琐碎的问题，你绝对不想花 11 个小时将手机硬件调配好后，再不得不等一个月，才能真正使用它。

从体积小易管理的底层模块开始创建。换句话说，若应用通过串行端口与外部设备进行通信，那么应该第一时间测试硬件接口（或模拟器的串行接口模块，我们将在第 10 章讨论）。确认正常工作之后，再添加更多的功能。人们最常犯的最大错误就是在开发新
295
系统时，试图一次性将所有的功能都做出来。如果你非常幸运，可能会运行起来，但是绝大多数情况是相反的，这种全好或是全坏的开发方式真正的问题在于，你可能做错了不止一件事，但是错误可能相互掩盖，导致你无法发现，更糟的是，个别单元独立运行时没有问题，但是当你尝试整合其他单元时，立即引发严重的问题。

下面是一份推荐的开发顺序，在具体的工程中可能略有不同，但是宏观上建议这么来推进：

1. 底层扩展（驱动器之类）。
2. 通信（例如，串行接口处理和远程通信支持）。
3. 实用工具类（如我们前面看到的 `InputUtils.py` 模块）。
4. 数据处理和错误响应。

换句话说，软件集成以自下而上的顺序，从硬件到本身来实现，这样实施的好处如下：

- 先对底层的配套功能进行测试和验证，为以后增加复杂的功能提供了坚实的基础。
- 如果代码是正确的并充分模块化，底层模块（甚至一些中层模块）可归到“软件零件盒”在未来的项目中复用。
- 当软件很小时，错误能轻易被识别和解决，从而使异常在扩散到整个软件前，得到有效的管理。

一旦所有的基本功能都完成，最后一个步骤才是配合测试和验证模块整合所有的上层功能，例如，数据处理和用户界面。

软件文档化

虽然 SDD 可作为软件的开发计划，但不是代码级文档必须存在的。就 SDD 的详细水平而言，至多笼统地描述应用程序的架构，不会涉及函式或方法参数、全局变量等具体细节。事实上，不在 SDD 投入过多的细节是有道理的，因为根据单元测试和集成活动的结果，架构总是在频繁地变动。那么，我们把类似功能参数的描述放在哪里合适呢？当然是在代码中。

Python 包括嵌入式文档的概念，被称为文档字串（docstring）。对于自动提取注释并生
296
成漂亮的含有索引的 HTML 格式文档，我们推荐出自爱德华·洛珀（Edward Loper）的 Epydoc 工具，可以从 *http://epydoc.sourceforge.net* 下载。

为了说明它是如何工作的，再次引用 InputUtils 模块，只是这一次我们添加了文档字串，并把它更名为 InputUtils2.py：

```
""" InputUtils.py

    输入检验及处理函式。

    开始使用 (ScaledInput)。
"""
NO_ERR          = 0         #: No error
ERR_OVER        = -1        #: Overrange error code (+ or -)
ERR_MAXSCALE    = -2        #: Maximum + output value exceeded
ERR_MINSCALE    = -3        #: Maximum - output value exceeded

DATA_MIN        = -10.0     #: Maximum - input value
DATA_MAX        = 10.0      #: Maximum + input value
SCALE_MIN       = -50.0     #: Maximum - output value
SCALE_MAX       = 50.0      #: Maximum + output value

data_scale      = 1.0       #: scaling coefficient
data_offset     = 0.0       #: offset coefficient

def ScaledInput(data):
    """ 输入数据检查以及放大。

        使用全局系数来放大数值并应用到偏移中。

        效果同公式 y = mx+b，其中 m 是放大系数，b 是偏移，x 则是输入数值，m 和 b 的值通过
        全局变量调节。

        输入数据根据预定义的范围进行检验。处理结果一样进行检查，以确保不输出限制范围。

        如果输入数据超过限定范围，则返回 ERR_OVER，如果输出超过限定范围，则返回
        最大值 (+ 或是 -)。

        返回的是双值字典，同时包含返回码以及结果值。

        @param data: 输入数值

        @return:    如果没有错误，就以双值字典的形式返回返回码和修订后的数值，
                    来替代未处理的输入数值。
    """
    rc = NO_ERR
    scaled_data = data

    if data >= DATA_MIN and data <= DATA_MAX:
        scaled_data = (data * data_scale) + data_offset
```

297

```
        if scaled_data > SCALE_MAX:
            scaled_data = SCALE_MAX
            rc = ERR_MAXSCALE
        elif scaled_data < SCALE_MIN:
            scaled_data = SCALE_MIN
            rc = ERR_MINSCALE
    else:
        rc = ERR_OVER

    return (rc, scaled_data)
```

Epydoc 将默认生成一组 HTML 文件。如果打开顶层的 *index.html*，看起来像图 8-19 一样。请注意，模块的全局变量也进行了文档化。

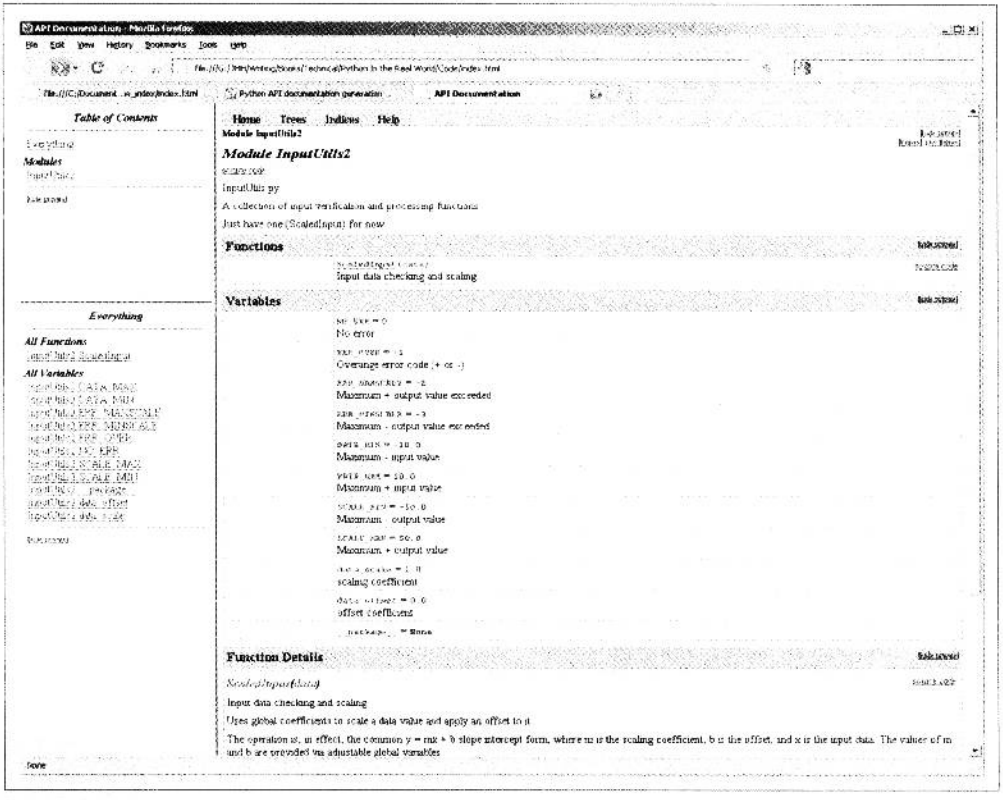


图8-19：Epydoc输出示例

Epydoc 是个非常有用且强大的工具。对于 C 或 C++ 语言，有个同类工具——Doxygen[译注2]，由迪米特·冯·黑森（Dimitri van Heesch）开发，同样根据注释中的约定标注生成 HTML 文档，可从 SourceForge 中访问：*http://sourceforge.net/projects/doxygen/*。

版本控制

版本控制是任何设计和开发活动的重要组成部分，一个良好的版本控制工具可以让你跟踪特定文件的更改和维护历史。如果发现有不如意的，版本控制系统也提供了一种方法可以方便地回退到指定的较早版本。即使你是唯一的软件开发人员，是否使用版本控制工具，就如同一旦发生意外，是只能哀号着重新键入，还是随时能安心地说："哦，没问题，我退回以前的版本就好。"这两种生存状态间的差异。

我们选择的版本控制工具是 CVS。虽然 CVS 已经很有年头了，但是依然在被广泛地使用，新工具 Subversion 也出现了，但我们感觉还是 CVS 使用起来最舒服（顺便说一下，CVS 其实算是 RCS 的后裔）。我们主张尽量使用纯文本文件，比如，源代码、用例或 HTML 文件，尽量不和其他二进制文件混合使用，比如，Word 文档，虽然有可用的开源工具，但支持 Word/Excel 类型文档的版本历史处理。

有关版本控制[译注3]的知识太多，一整本书都不够容纳的。本书的重点不是这个，但我们会鼓励你继续探索，由珍妮弗·威斯波曼（Jennifer Vesperman）撰写的"*Essential CVS*"（O'Reilly 出版）一书开始就非常好，当然维基百科上也有记载（*http://en.wikipedia.org/wiki/Concurrent_Versions_System*）。

最后，我们发现，CVS 在大多数 Linux 安装的联机帮助页中有大量组织良好的信息。事实上，我们最喜欢的 CVS"手册"是打印输出的 CVS 的联机帮助页和一些额外的提示页面。

缺陷跟踪

Bug 跟踪工具又称为缺陷跟踪者，它在团队环境中是必不可少的工具，当某人（也许是测试工程师）发现可能的缺陷时，通常由另一个人（问题代码的编写者）来试图解决问题。当然对于单人的项目，可能没必要配备完整的缺陷跟踪系统，只要善用版本控制工具，确保用笔记或是其他方式来跟踪"待办"任务就足够了。由于本书的重点是小项目，在一到两个人的小团队中追踪问题不会花费太多时间。不过，我们会尽量提供足够的信息（包括网址和软件包），以便你判定是否有更好的解决方案。

译注2：从实现上看，Doxygen 还可以输出 pdf、rdf、chm 等格式文档，而且 Doxygen 支持多种注释形式，也可以兼容 Python 脚本等。

译注3：从实现上看，对于小型团队，特别是分布式合作的团队，当前各种分布式版本控制系统比 CVS 这种集中式的要好，比如 Python 工程自身使用的 Mercurial(Hg)，以及 Linux 内核团队使用的 git。

缺陷跟踪系统背后的想法很简单。首先，发现缺陷并录入到系统，接着将对应产生的“传票”分配给某人。然后缺陷条目就可以通过设定好的流程进行处置，比如：非读、修订中、测试中、已核实和已解决。条目也可能在测试和修订环节中往复几次，才能最终标定为已解决。

299 大多数开源缺陷跟踪系统利用 Web 界面，有的需要专门的 Web 服务器。我们偏爱 Roundup 工具，它简单易用，而且可以部署在 SourceForge 上，并不需要 Web 服务器（如果你愿意，也可以部署到专用主机中）。当然，它是完全用 Python 实现的。

如果你想进一步研究 Roundup，请访问主页 *http://roundup.sourceforge.net*。

最后，推荐史蒂夫·麦康奈尔（Steve McConnell）发布的一篇有趣的文章，阐述了缺陷跟踪如何协助软件发布，其访问地址是 *http://www.stevemcconnell.com/ieeesoftware/bp09.htm*。

用户文档

在某个时间点，你的项目将完成（严正期望）。所有的需求已被定义，SDD 已完成，代码编写和测试也完成了，而且一切看起来工作良好。如果一切都顺利按计划一丝不苟地达成了，可运行的软件就是你兢兢业业辛勤工作的成果。但是，还必须完成真正的最后一步：用户文档。

即使你已经创建了主要供自己使用的文档，但是，还应该有一些介绍如何使用你的软件的文档。对于范畴有限的项目，我们常在三孔活页夹中使用横格纸将主要程序段、参数表、错误码和其他必要的信息打印成单页，追加到活页夹中。从本质上讲，这就是完整的用户手册的一个高度浓缩的版本。这种方法的优点是可以随时创建每个页面，根据需要放入活页夹中，一旦事情发生变化（几乎总是会变化），替换它们也非常容易。这样活页夹就生活在实验室中了，任何人（包括我自己）都需要这样一个活页夹，以便随时更新并查阅系统相关信息。

如果系统涉及范畴足够复杂，你就要考虑创建实际的用户手册[译注4]。这听起来令人生畏，但参考其他系统的用户手册是一种靠谱的开始方法。你喜欢（或不喜欢）哪种手册？为什么？选择一两个你认为写得好的，参照着来就成。

译注4：从效果上看，对于小型团队，特别是分布式协作团队，使用维基系统，跟随开发进度，随时增补，完成关键信息的组织，本身就是一个可维护和可交付的用户手册。当然，最好事先根据用户最应该掌握的用户手册内容，规划好章节结构和层次。*http://www.wikimatrix.org/* 列出了几乎所有的维基引擎系统，译者推荐用纯 Python 实现的 MoinMoin，因为它不依赖数据库，可独立运行。

小结

本章论述了软件需求、设计、实施、测试和文件。需求在软件工程中就像舵手与船，虽然没有最基础的需求，也能很轻易地创造出点东西，但是并不会按照预期来工作，甚至根本不能正常工作。在完成需求的功能后，通过测试来检验，不仅有助于找出代码中潜在的缺陷，而且还能确保代码在吻合需求的路线上稳固发展。如果你诚心渴望获得认同，就得形成文件，以便他人容易理解和欣赏你的成果（可能无法达到你理解的深度）。然而，本章的核心要点并不是要求严格遵守特定的过程或是生命周期的模式，也不是如何创造包含大量技术细节的文档，而是大致指出软件开发的过程，以便读者明确当前处于什么位置，还需要做什么来达成目标，以及给出整体图景，分享文档的组织结构，以便读者可以参考。

推荐阅读

涵盖本章主题内容的好书数以百计，我在此只能推荐特别喜欢的：

Code Complete, Second Edition. Steve McConnell, Microsoft Press, 2004.

我有这本书的第一版和第二版，我认为它们都是很优秀的。McConnell出色地指出了成功界定或实施高质量软件的必要步骤，我只能鼓励更多的人来阅读和应用他的著作中提到的各种资料。

Software Requirements: Objects, Functions and States, 2nd ed. Alan Davis, Prentice Hall, 1993.

这是有关软件需求本质的书籍中我最喜欢的一本。我总是带着这本书参加会议并反复引用关键语句。该书作者以清晰的风格，明确讨论了分析和捕获需求的各种方法，还包括了各种方法的实用案例，书中的引用列表提供了丰富的路径进一步探索和发现。

Software Testing. James McCaffrey, Booksurge, 2009.

作者通过亲身访谈一些测试经理，针对软件测试的技能和技巧整理出一个很好的概览，章节内容精巧，对软件测试新人而言是很好的开始。该书还可以当成经验丰富的从业者随时查询。

Writing Effective Use Cases. Alistair Cockburn, Addison-Wesley, 2001.

本书是介绍如何编写好测试用例的易用指南，它介绍了良好的用例由什么构成，指出在制作好案例的过程中可能遇到什么陷井。虽然我并不热衷于用户案例（我更喜欢需求分层），如果你在规划用户案例，应该先读一下这本书。

301 *Essential CVS, Second Edition. (http://oreilly.com/catalog/9780596527037) Jennifer Vesperman. O'Reilly, 2006.*

这本书介绍CVS相关的基本概念，并从那里开始讨论标记、分支、合并和日志等其他议题。有助于填补正规的CVS在线参考文档的空白，并澄清一些尖端的CVS的特殊功能。

谈及网上资源，在Google中输入“软件需求”会返回数百万点击率之类的信息。下面是我从自己的书签里列出的几个值得注意的网址：

http://software.gsfc.nasa.gov

在戈达德太空飞行中心的美国航空航天局，其过程资产库发布到了网络中。虽然相对典型的小项目来说，其过程过于复杂，但是整体观念和方法是普遍适用的。这值得一试，看看这里是否有东西用于你自己的开发活动。

http://www.techwr-l.com/techwhirl/magazine/writing/softwarerequirementspecs.html

TECHWR-L是面向技术交流的网站，包含了各种讨论如何编写需求规模说明等相关的文章。

http://www.aspera-3.org/idfs/APAF_SRS_V1.0.pdf

这是来自瑞典空间物理研究所的Aspera-3项目的SRS[译注5]（“火星快车”任务的一部分）。真正的SRS并不是经常都能看到（特别是美国航空航天局最近的项目就没有，多亏了ITAR[译注6]），但在这里你可以悠闲地查阅。

译注5：SRS（Software Requirements Specification，软件需求规格）

译注6：ITAR（International Traffic in Arms Regulations，国际武器贸易条例）

第 9 章 303

控制系统概念

如果所有的事情看起来都在掌控之中，则说明你还不够快。

——Mario Andretti

如果一本关于在现实中测量数据和控制系统的书不讨论控制系统基础和它相关的概念，它将是不完整的。尽管本章并不是要深入严谨地讲解控制系统，但如果你需要，它却可以让你自己组装一个可用的控制系统。

以第 1 章中介绍的内容为基础，本章会用正式一点的定义更深入地介绍普通控制系统的概念和其他一些基本的细节，还会介绍一些基本的控制系统分析和提供一些选择合适模型的指南。

本章将把精力集中在由软件和机电原件组成的简单的控制系统上。这些系统可以由现有的仪器和控制器件组成，比如，数字万用表、数据采集器、电动机控制器、电源、功率控制模块等。你不用设计和制作任何电路板（当然除非你愿意去做），也不用去管那些神奇的器件和接口——你所需要的东西应该在货架上都有了。实际上，它或许正在货架的某个地方慢慢地吃灰。

我们会以对线性、非线性和顺序控制系统的概述作为开始，然后介绍控制系统设计中的一些术语和符号的定义。接着，我们会浏览系统框图，并介绍它们在控制系统设计中是如何使用的。我们会对时域和频域的区别做一个快速的了解，并且介绍这些概念是如何应用在控制系统理论中的。本书不会涉及诸如拉普拉斯变换方程之类的知识，而只是介绍概念。这主要是因为我们所涉及的控制系统类型可以很容易用普通数学建模和实现。

下一节会涉及一些具有代表性的控制系统，同时也会介绍它们所应用的术语和理论。我们会对开环、闭环、顺序、PID（比例 - 积分 - 微分）、非线性开关和混合控制系统进行描述和举例。 304

本章最后，我们将会考虑在用 Python 设计和实现一个控制系统时会遇到什么问题。我们会看到比例控制、非线性开关控制和一个简单的 PID 控制的例子。

基础控制系统理论

我们被控制系统所包围着，尽管我们有生物特征，但我们本身就是某种形式的控制系统。一个控制系统可以像电灯开关一样极其简单，也可以极其复杂，比如，飞机里的自动导航设备或者炼油厂里的控制系统。

从广义上讲，一个控制系统是生物的、机械的、气动的、电子的或者其他类型部件的组合。它可以用输入的某种形式来控制或者调整输出动作。有能力监视和调整自己行为的控制系统使用了一种叫反馈的技术，这种反馈技术通过比较输入和输出来产生一个偏差值，并以此为基础对输出进行必要的修正。

一个控制系统并不总是简简单单的一个东西，它可能是由许多子系统组成的，每个子系统又可能会使用不同的控制形式。当把它们组合在一起时，各个子系统形成有良好定义行为（当然是在理想情况下）的内聚的个体。从规模和复杂度来说，控制系统总体的大小是它能力的函数。在这个基础上，我们甚至可以说地球中的大气是一个巨大的自调节气候控制和水力分配系统。而它本身又是整个星球系统的子系统。从小范围来看，一艘大船（比如一艘货船）是一个运载货物的系统，它也是由从发动机到方向舵控制系统等许多子系统组成的。

如果你看看周边的控制系统，你可能会发现它们非常简单或者是由简单的子系统协同工作来达到一个特定的（或许也是复杂的）结果的。

在这一章里，我们将主要涉及三种控制系统：线性的、非线性的和顺序的。一个线性的控制系统使输出变量是输入变量的连续函数。而非线性控制对一个线性输入的响应并不是连续的，就像名字里所表达的一样。一个顺序系统会在一系列状态之间切换，其中每一个状态会产生一个特定的输出或者一个后续的内部状态。我们不会研究模糊逻辑、自
305 适应控制，或者多输入 - 多输出控制系统。它们是现代控制系统研究中深奥的话题，而且与本书的范围相差甚远。

线性控制系统

就像我们在本节前言里所说的那样，一个线性控制系统所生成的输出是其输入的连续线性函数。例如，考虑下面这个函数：

$$y = mx + b$$

这就是广为人知的斜截式方程。它确定了 x 和 y 之间的线性比例关系：y 等于 x 乘以 m，b 是附加在输出上的一个可选的偏移量。这个可以用在控制系统中吗？当然可以。实际上，它本身就可以作为一种形式的比例控制。方程 9-1 显示了同样的方程用控制系统符号重写之后的结果：

$$u = K_p \times e + P \qquad \text{（方程 9-1）}$$

对于离散系统，方程 9-1 可以写成方程 9-2 的形式。

$$u(t) = K_p \times e(t) + P \qquad \text{（方程 9-2）}$$

在方程 9-2 中，$u(t)$ 是输出；$e(t)$ 是系统偏差；K_p 是对系统偏差的比例增益，P 是稳态偏差（一般是 0）。符号 t 代表了瞬态时间，也就是“时刻”。它除了表示 u 和 e 都是在时刻 t 的值以外，并不在方程里起什么别的作用。

不用担心什么是偏差以及这里的各个符号都是什么意思。我们马上就会讨论它们。重要的是，注意这只是披上华丽外衣的斜截式方程。

控制系统一般分为两类：闭环控制系统和开环控制系统。它们之间主要的区别在于是否有能力感知控制器对被控对象所产生的影响，并据此调整它的输出。闭环控制系统也叫做反馈控制系统。

图 9-1 显示了从方程 9-1 中得到的结果。这里用偏差值作为输入，但是它其实是参考值（或者是控制输入）与从被控设备得到的反馈的差值（我们马上就会说到这个）。主要的意思是在理想情况下，它是线性的。 306

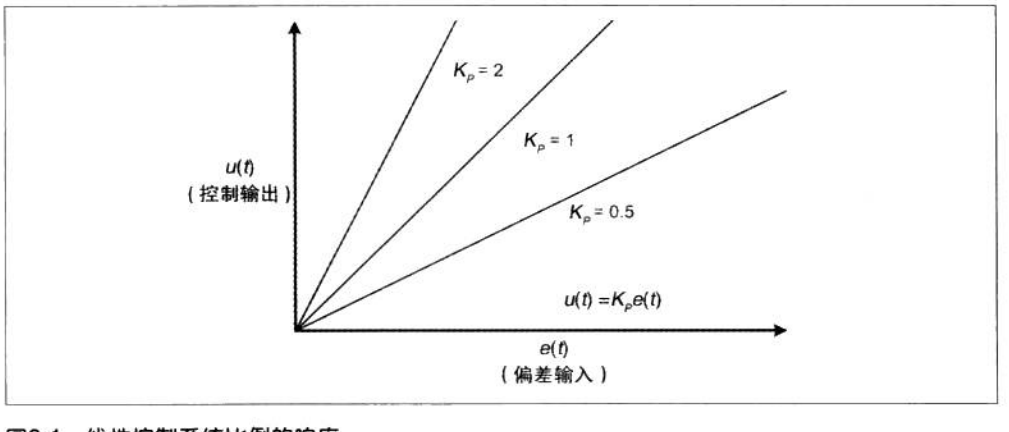


图9-1：线性控制系统比例的响应

应该指出的是，尽管线性模型被广泛地应用于控制系统分析和建模中，在现实里其实没有真正的线性系统。因为各种原因，每个系统都会在特定情况下或多或少地表现出非线性。而且，在一个闭环控制系统中，控制器的输出响应与反馈有关。后面我们将会看到，从质量、惯性、滞后环节中所产生的反馈可以并且经常使系统输出一条不是很好的直线。

非线性控制系统

一个非线性控制系统的输入和输出并不是线性连续的关系。例如，就像图 9-2 所示的，输出可以在两个或多个状态之间切换。

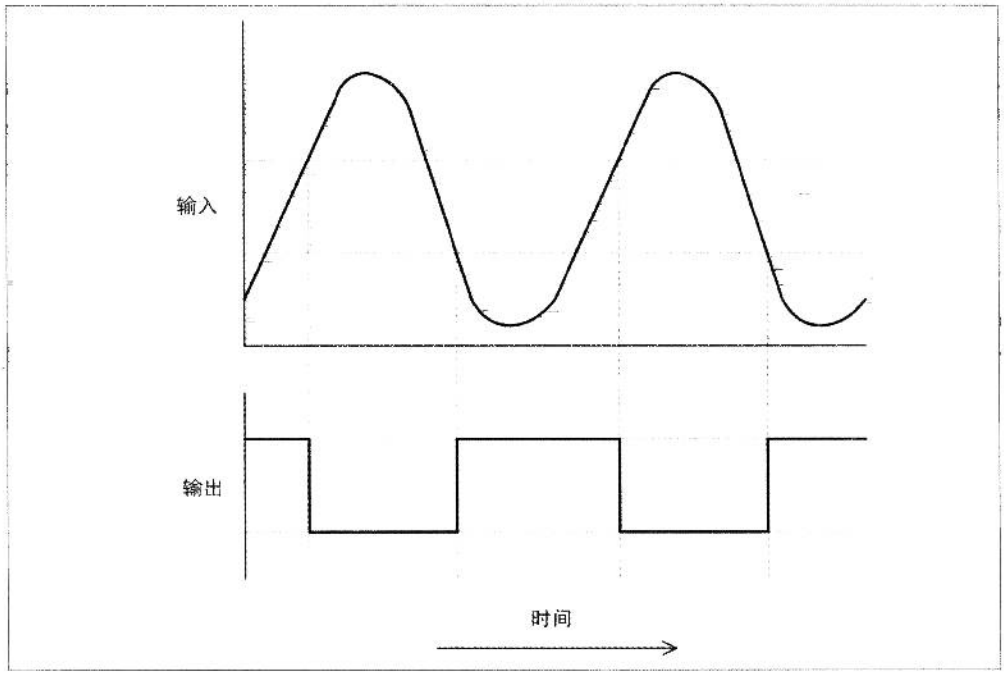


图9-2：非线性控制系统的响应

在数学中，图 9-2 中的行为可以表示为类似于方程 9-3 的分段函数。

$$f(V_{in}, S_{low}, S_{high}, H) = \begin{cases} 1, & V_{in} \leqslant S_{low} - H \\ 0, & V_{in} \geqslant S_{low} + H \end{cases} \qquad \text{（方程 9-3）}$$

当 $S_{low} - H < V_{in} < S_{high} + H$ 时，输出保持不变。

这是所谓继电器式或者开关式控制器的典型行为。如果输入超过了一个阈值，输出就会发生变化。否则，它就保持不变。我们很快就会对这种控制器进行深入了解，并且看看它们会用在实际生活中的什么地方。 307

现在我们来考虑一种像短脉冲一样只会在特定的一段时间里起作用的控制系统。为了实现一个连续的控制功能，脉冲频率和持续时间都必须在控制之中。图 9-3 展示了这种控制器。

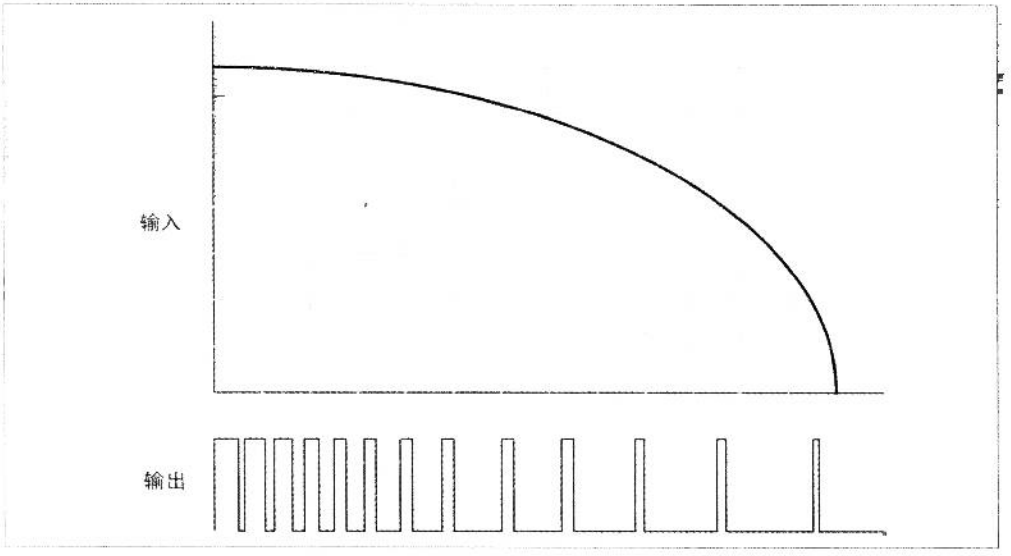


图9-3：非线性脉冲控制

这种控制器可以在很多地方找到，比如，新型汽车中的防抱死刹车系统（即 ABS）和星际着陆器上变占空比火箭发动机的控制系统里。在这种情况下，发动机不是开就是关，而且因为需要冷却，发动机不能打开很长时间（否则，它们就会因为过热而烧毁）。

注意，线性和非线性系统都能实现对变量的连续控制。它们的区别在于为了达到控制目的，输入是怎样作用于输出的。稍后我们将会看到，尽管控制器本身的输出肯定不是平滑和线性的，非线性控制器也能产生平滑的结果。

顺序控制系统 308

一个顺序控制器有许多独立的、伴随着离散输入和输出信号的状态。顺序控制器可以打开或者关闭具有离散状态的器件或者线性（或非线性）的子系统。要点就是在预定的阶段内，被控器件或者处于激活状态，或者处于未激活状态，或者是开，或者是关。

顺序控制系统经常可以在需要以一定顺序和一定时间段执行特定动作的应用场合中看到。例如，自动洒水系统。在内部，洒水器是根据洒水区域的模型来控制的。它一般会根据喷头所在的院子（当然，也可以类似于高尔夫球场或公园）来划分成组。

图 9-4 显示了一个五区自动喷水灭火系统的时序图。该系统通过编程控制每个区在一周指定的日子里打开指定的时长。

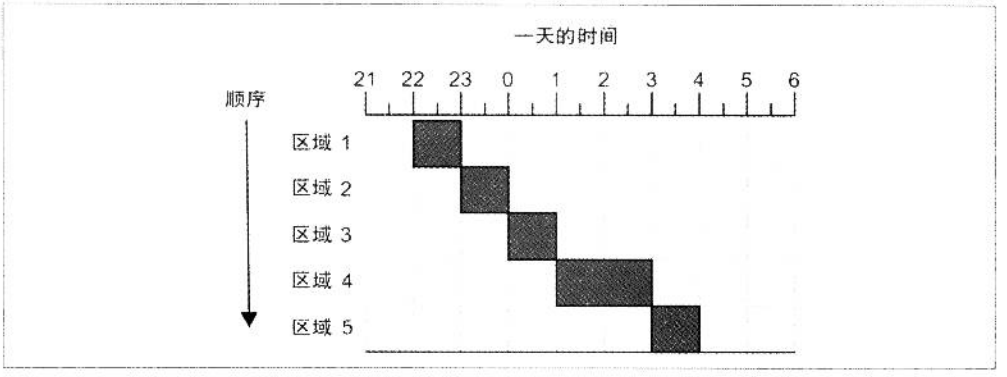


图9-4：洒水器顺序控制系统

在这个例子中，控制系统在晚上 22:00 启动，早上 4:00 停止。典型的自动洒水控制系统是一种只和当前时间有关的顺序控制系统。它没有其他的输入来调整它的动作，它不关心地皮是否已经湿了，也不关心是否正在下着倾盆大雨，它总是会在指定的时间给草地浇水。

309

术语和符号

在更深入地介绍控制系统之前，我们有必要先了解一些遇到的术语和符号。像其他先进的学科一样，控制系统工程有它自己的术语和符号。表 9-1 列出了一些在涉及控制系统时会经常遇到的基本术语。

表9-1：控制系统中的术语

术语	描述
闭环控制	一个闭环控制系统包含从被控过程或被控对象获取的反馈，这样它就可以自动调整控制输出，从而达到抵消扰动，保持被控对象输出的作用
连续系统	一个在任何可能的连续时间内都有确定行为的控制系统
控制信号	也被叫做“输出”。这是为了使被控对象按照需要的方式动作而给它的输入信号
控制系统	为了调整自己或其他系统的行为，可以接受输入和产生输出的系统。参见“系统”。

续表

术语	描述
受控输出	由系统根据输入产生的响应。如果是闭环控制系统，则输入中也包含反馈。在框图中，以 c [译注1] 表示
离散系统	一个只在特定的离散时间点上有确定行为的控制系统
偏差	参考输入和反馈值的差
反馈	一种从被控对象引入控制系统的输入，并且它和参考输入一起生成偏差值。在框图中用 b 表示[译注2]
增益	控制系统中用来改变控制信号或反馈信号的乘数。尽管增益是可变的，但是在传统的控制系统中，它并不随时间变化。在某些自适应或非线性控制系统中，增益是时间的函数
输入	当把控制器看成是一个整体时，也叫做“参考输入”。其他情况下，指控制器中任何功能模块的输入
线性控制	输出是输入的线性连续函数的控制
非线性控制	输出不是输入的线性连续函数的控制。它的输出可能是输入的离散状态非连续的函数
输出	一般指控制器或者其中间功能模块的输出，而不是被控对象的输出（参见“受控输出”）。在框图中用 u 表示
开环控制	没有用反馈来检测控制效果的控制系统。一个开环控制系统主要依靠本身的精度和标定来保证控制的准确
被控对象	被控制的过程或系统。根据被控制的器件和系统的不同，也被称为“被控过程”或“被控系统”
参考输入	对控制系统的激励。参考输入代表所需要的被控对象的输出或者行为。在框图中用 r 表示。参考“输入”
采样数据	以一定间隔获得的数据。每个数据代表在离散时间点上的特定信号或系统的状态
加和节点	参考输入减去[译注3]反馈的那一点。也可以指两个或更多信号做数学加和的节点。
系统	由相互联系的功能模块组成，像一个整体一样运转的组合

310

表 9-2 列出了一些在控制系统框图中会普遍遇到的字母符号。

表9-2：通用控制系统框图符号

符号	含义
b	反馈变量（或信号）
c	系统的被控输出
e	由 $r-b$ 产生的偏差值
K	增益值
r	系统的参考输入
u	被控变量（比如给被控对象的输入）

译注1：原文中把“受控输出”和“输出”的数学符号弄反了，翻译时更正。
译注2：原文中把“受控输出”和“输出”的数学符号弄反了，翻译时更正。
译注3：“减”也是一种“加”。

控制系统框图

在控制系统设计和分析中会大量用到框图，在此会介绍一些在后面章节中将会用到的基本概念。图 9-5 用典型的表示法展示了开环和闭环的控制系统。

311

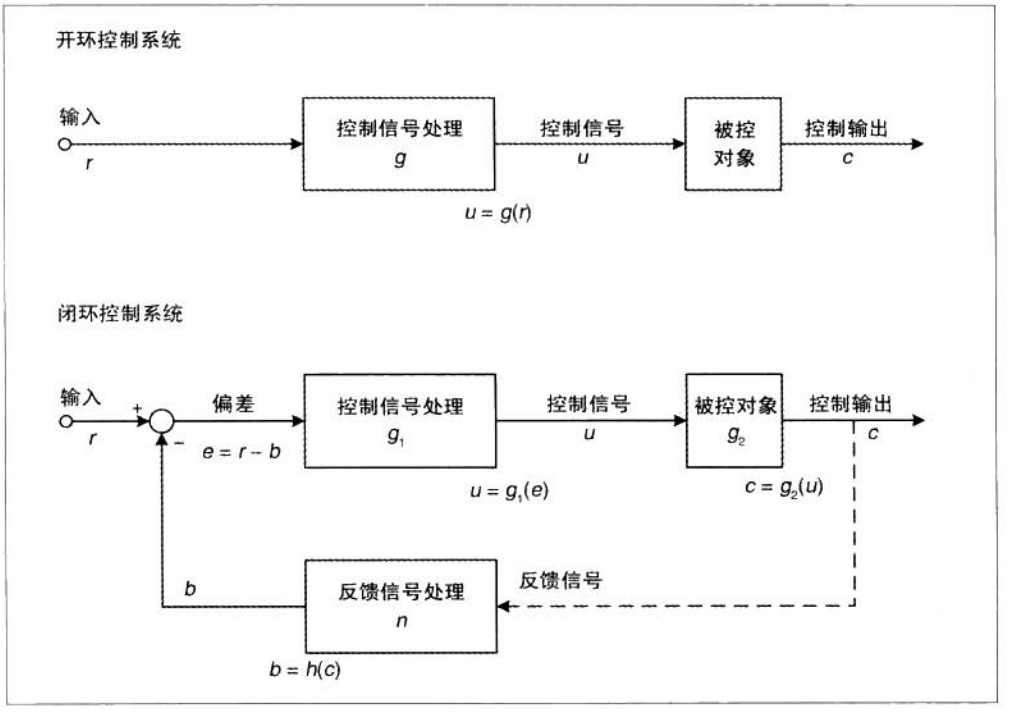


图9-5：控制系统框图

每一个控制系统至少有一个通常叫做参考输入的输入，以及至少一个通常叫做控制信号或者控制变量的输出。被控对象最后的输出叫做受控输出。在闭环控制系统里，这就是那个被测量并且用在反馈回路中的变量。系统输入和受控输出之间的关系定义了系统的行为。在图 9-5 中，参考输入用 r 来表示，控制变量是 u，受控输出是 c。这些符号是历史遗留下的产物，并且仍然被普遍使用着。所以，我们在这里也用它们来表示。

输入 / 输出关系

框图中的每一个方框都有输入和输出，代表了过程函数。它还可以有像偏差或者外界扰动一样的辅助性输入。

输出可以和所加的输入相等，也可以不等。实际上，就像在图 9-5 中用“控制信号处理”

所示的，在输出和输入之间有某种数学处理是非常普遍的事。注意这个框里有个符号表示它内部的函式。这里是 g（或者闭环系统中的 g_1）。这可以像相乘（增益）或者相加（偏移）一样简单。它也可以涉及积分、微分或者其他操作。应用什么样的函数完全取决于 312 控制信号需要如何响应输入。

反馈

在一个闭环控制系统中，控制信号是根据偏差得来的，偏差是输入和反馈值的差。反馈的符号是 b，它等于 $h(c)$。其中，c 是从受控输出中得到的反馈信号。那个圆形的符号叫做加和节点。在图 9-5 中，加和节点的输入 b 有一个负号。它意味着这是一个负反馈系统。但是，正反馈系统也是可能的。在这种情况下，那个符号就会变成正号。

传递函数

系统中输入和输出之间的关系通常用传递函数来描述。一般来讲，控制系统框图中的每个环节都可以用某种传递函数来描述。而整个系统的输入 / 输出关系可以用系统内部的传递函数推导出来。设计和分析控制系统的活动之一可能就是把内部的各个传递函数化简成一个描述整体输入 / 输出关系的传递函数。

在数学上，一个传递函数是非时变系统输入和输出的描述。这是什么意思？我们很快将会介绍非时变系统。现在可以理解为传递函数只是应用于频域里，并且这个函式关系并不依赖于时间，只和频率有关系。

在控制论里，传递函数是用类似于傅里叶变换的一种积分变换——拉普拉斯变换——推导出来的。两者主要的区别是傅里叶变换把信号或函数分解成不同的频率，而拉普拉斯变换把它们分解成不同的“时刻 (moments)”。在控制论中，拉普拉斯变换经常被用来把信号从时域转换到频域。当处理复杂的或者是对频率敏感的系统时，拉普拉斯变换得来的传递函数一般会比较好处理，也会简化系统的模型。

尽管拉普拉斯变换被广泛使用，但是它并不是必需的。本章不会用拉普拉斯变换，主要是因为我们将要处理的问题非常简单，也是因为我们只会涉及那些速度慢到对频率响应要求不大的系统，并且在时域系统里分析它们。根据我们的需要，基础的代数和计算就已经够了。

时间和频率

313

时间和频率在控制系统设计中是关键部分。活动会在一段时间内发生；事件可能以某一固定或者变化的频率发生；交流信号会有一个特定的频率（如果信号复杂，也会有几个频率）。在数字控制系统中的处理也需要定量的时间。这都是系统设计中的重要组成部分。

时域和频域

当讨论关于时间的数学函数或者电子信号时，我们就会用时域来解决问题。反之，如果我们主要在频率方面分析和处理交流信号，我们就会用频域来处理。这两个术语指的是在对系统进行分析和建模时用的数学方法。哪种方法合适取决于我们所需要的分析结果。

这些区别可用于整个系统，但通常用于特定的子功能系统中。例如，一个时钟或定时子系统在时域中操作，而滤波器或移相子系统则是在频域中操作。

区分时域和频域的一个方法是看图中横坐标的单位。图 9-6 展示了在示波器（在第 6 章介绍过）和频谱分析仪（FSA）中可能会看到的图像。

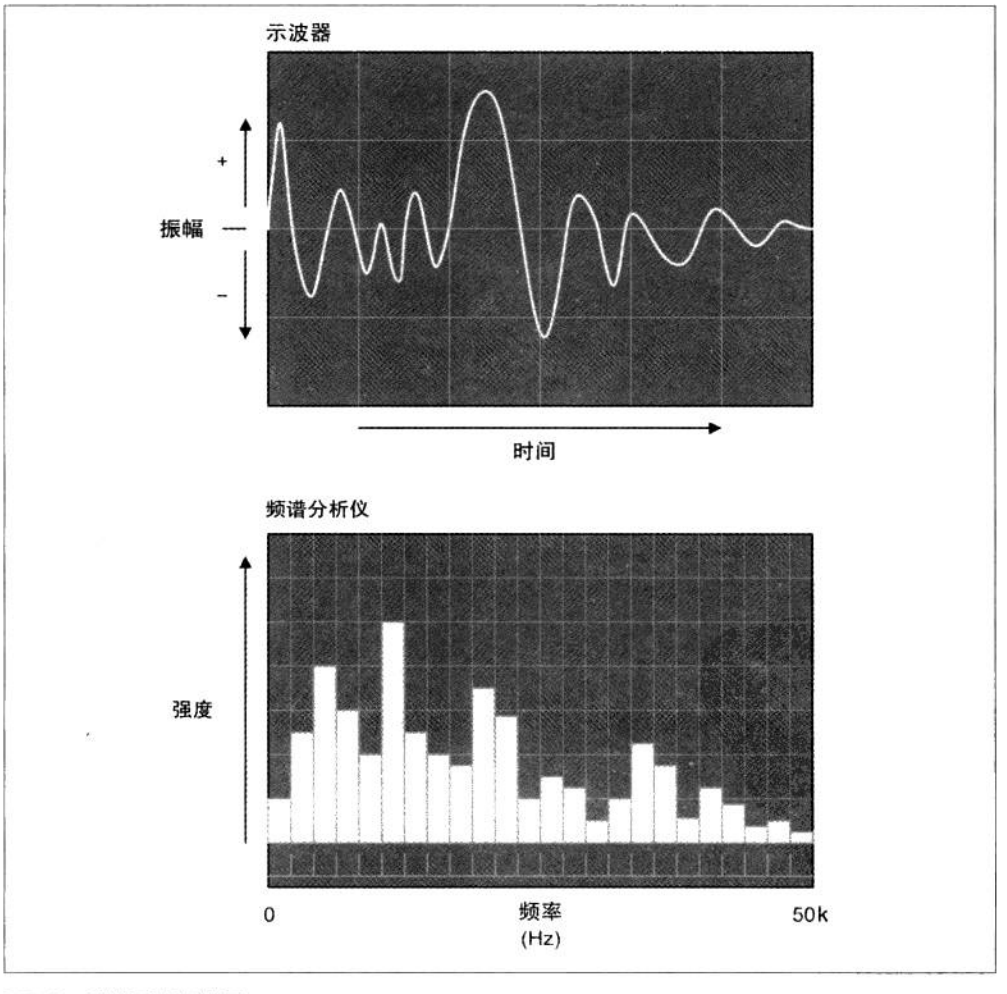


图9-6：时域图和频域图

这里的要点是示波器在时域中工作，而频谱分析仪在频域中工作。关键是X轴的变量不同。在示波器里，X轴是时间，Y轴是信号在对应时间上的强度。我们可以直接在示波器上测量出一个波形的时间间隔和振幅，但是它并不能直接告诉你信号里的频谱分布。你需要一个频谱分析仪才能得到它。

频谱分析仪的X轴显示的是频率，在图9-6中是从0到50kHz。尽管这里是用柱状图画的，但也可以用其他方法画。频谱分析仪工作时会从信号中提取频谱（可能是用一组离散滤波器，或者用傅里叶变换处理信号），其结果就是一组数据代表各个频率部分及其所对应的幅值。

我们也可以根据需要在两个“域”之间切换，因为时间和频率互为倒数。它只和你看问 314
题的角度有关。所以，如果你有两个间隔20ms的波形，它的频率就是20ms的倒数，也就是50Hz。换句话说：

$$f = 1/t$$

和

$$t = 1/f$$ 315

另一个例子是系统中每隔500μs会传来某种传感器的信号。如果对500μs求倒数，我们会得到2000Hz（或2kHz）。如果我们需要对频率上限是1kHz的信号进行采样[译注4]，这个数据就非常重要了。

时间与控制系统行为

有的控制系统行为受时间影响。我们可以用术语“时变”和“非时变”来形容系统对时间的敏感程度。

一个非时变系统的输出与时间无关。t_0时成立的关系式$y_{t0}=f(x_{t0})$会和$t1$时的关系式$y_{t1}=f(x_{t1})$产生的值相同。换句话说，无论时间是多少，系统对于相同的x值，总会输出相同的y值。

对于非时变系统，特别是线性非时变系统（linear time-invariant systems，LTI），主要在频域里运转。它们每一个都有由输入信号的傅里叶变换和系统传递函数所确定的输入/输出关系。放大器就是一种线性非时变系统。它与信号到达的时间没有关系，它只根据信号的频率和内部的传递函数来处理信号[译注5]。一个理想的放大器对于相同的输入总是会给出相同的输出。

译注4：参见“采样定理”，*http://zh.wikipedia.org/zh-cn/* 采样处理。
译注5：实际上，一个放大器并不关心信号的频率。作者在这里说的会根据信号的频率来选择不同放大倍数的器件应该是“均衡器”。

一个时变系统的输出和时间有关系。时间是速度的一部分（速度 = 距离 / 时间），所以，一个运动中的系统是时变的。比如，飞机的自动导航系统必须要把时间和空速考虑在内，这样它才能够根据飞机的速度和朝向来估计位置。所有的这些因素都和时间有关。

离散时间控制系统

最后，我们介绍离散时间控制系统（discrete-time control system）。离散时间控制系统又叫做离散控制系统。这类控制系统可以是线性的，也可以是非线性的，当我们不用顺序控制系统时，我们就会直接用这种控制系统。实际上，所有用计算机和软件进行信号处理和控制的系统都是离散系统。

在传统的模拟控制系统中，输入和输出之间的关系是立即和连续的——一个在输入或者反馈上的变化会立即反应在输出上。对于离散线性控制系统来说，不管是在计算机里还是在数字控制电路里，输入数据采集（包括参考输入和偏差）、控制过程、输出过程都
316 是在一个时钟的指挥下在离散的时间上一步一步进行的（即连续的）。图 9-7 展示了一个离散闭环控制系统的框图。

在图 9-7 中，被标记为“时钟（Clock）”的框驱动着系统的控制逻辑。它可以是一个微处理器或者是一个软件。就像标记 t_1、t_2 和 t_3 所示的，它又会相应地开启输入、处理和输出功能。不过需要知道离散时间系统并不意味着时变系统。在这个例子中，在任意的时间 t 上，对于给定的 r 和 b，系统都会输出相同的 u 和 c。这是一个离散线性非时变系统。

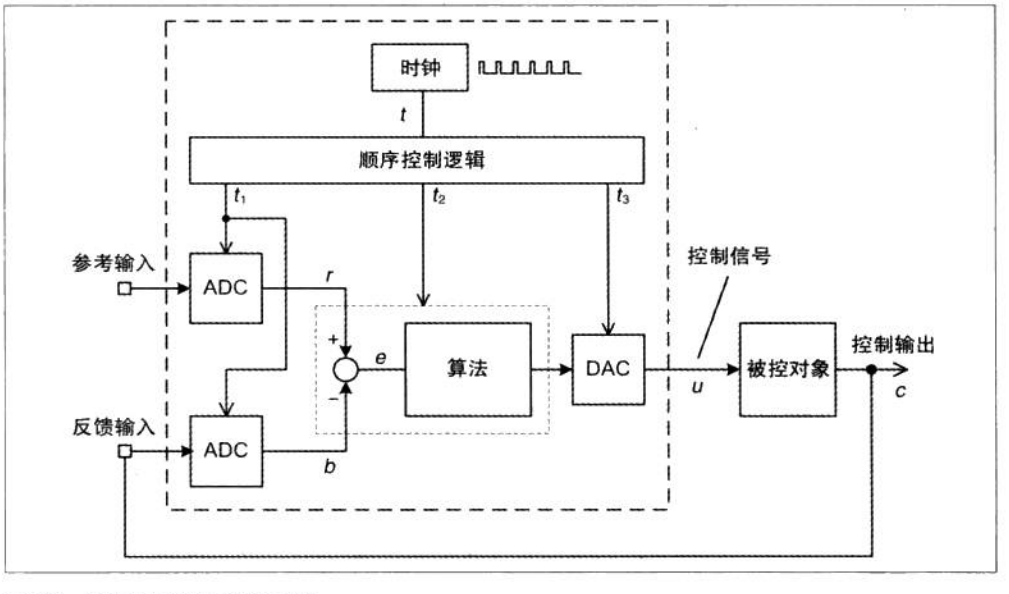
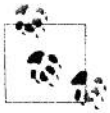


图9-7：离散时间闭环控制系统



有一种叫做 Z 变换的技术可以用来对离散时间系统进行建模和分析。这和拉普拉斯变换在连续系统中的应用类似。如果系统中有连续环节，也有离散环节，那么为了方便分析，从一个变换到另一个变换是很常见的。本书不会涉及 Z 变换，但是我们希望你知道它。我们关心的是时域上的离散系统和它对控制系统的影响。

在离散时间控制系统中，完成整个输入、处理和输出所需要的总时间决定了什么时候获取数据，什么时间输出控制。

被控系统有一个叫时间常数的常量。系统的时间常数是系统输出变化一定量所需的时间。 317
换句话说，小的变化（比如噪声）或许可以忽略，但是大的变化必须要被检测和控制。控制系统完成一个控制循环所需要的时间是系统时间常数的函数。

图 9-8 显示了一个计算机控制系统的主要结构。它由图 9-7 中的三个主要部分组成：ADC 数据采集（采集框）、数据处理算法（处理框）和 DAC 控制输出（控制框）。在离散系统中，这些步骤都是按照一定的顺序执行的，每一步都需要一定时间（t_a、t_p 和 t_o）。

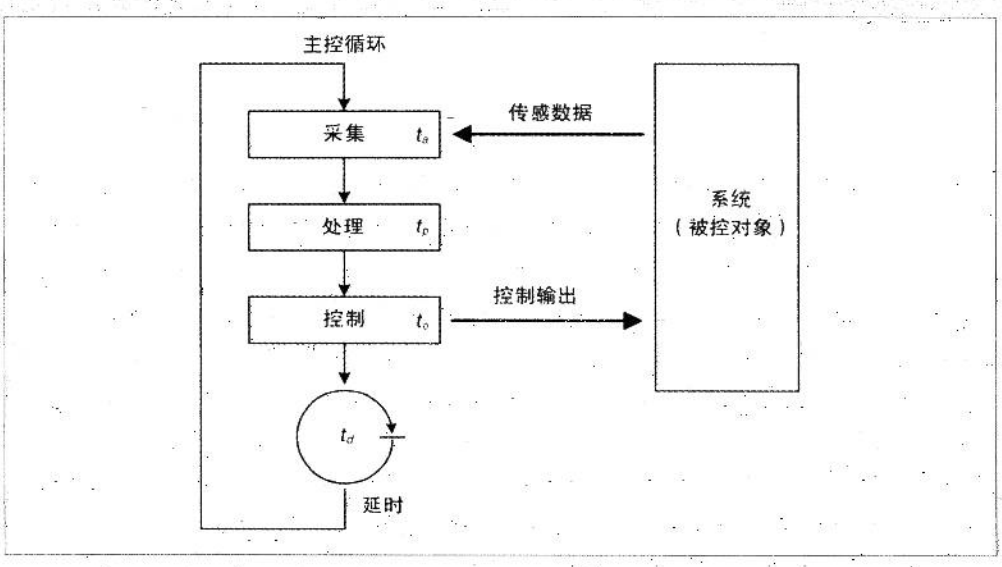


图9-8：离散控制系统软件流程

图中还有一个圆形符号代表了延时 t_d。在输出生成之后，延时可以让外部系统在流程被重复之前有所响应和生成新的输出。一般来说，这个延时会根据系统的时间常数有所调整。

还需要注意到这个图是一个死循环，这是测试系统的典型做法。一旦确定并实现了应用

所需要的步骤，控制程序就会以循环的形式来运行。不断地重复采集、处理、控制这个循环，直到程序被终止。

318 图 9-9 显示了图 9-8 的一个时序图。从这里我们可以看出每个步骤都会耗费一定的时间。线变高代表这个环节被激活，线变低代表这个环节被关闭。

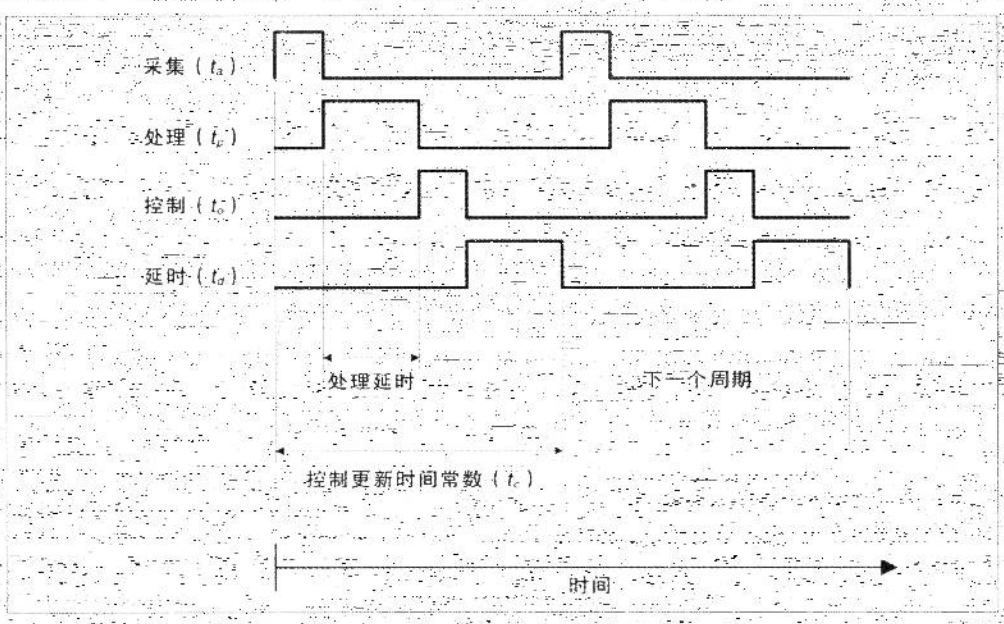


图9-9：控制系统软件时序

图 9-9 中控制系统的控制循环时间等于两个采集开始之间的时间。从图 9-9 中可以看出：

$$t_c = t_a + t_p + t_c + t_d$$

这或许是一个简单的等式，但是我们马上就会看到，那些变量的取值会对被控系统有深远的影响。在许多情况下，延时占了大部分循环时间，但是如果外部设备响应比较慢，采样时间也会较长。

控制系统类型

到现在为止，我们只介绍了所谓控制系统的一个基本的概述。本节将会看到一些具体的例子，并且应用我们所介绍过的一些概念和术语。我们首先看看开环控制系统；然后是闭环控制、顺序控制和非线性控制。最后，我们将结束于比例积分微分（PID）控制。它是当今应用最广泛的控制形式。

开环控制

319

在第 1 章中，我们把一个自动户外灯当做非线性开环系统的一个例子。线性和非线性开环控制系统是以输入 / 输出关系为标准来进行区分的，就像我们已经看到的，一个开环控制器对系统的状态没有“感知”。精确度和可重现性完全取决于内部的精确度和对系统各个环节的标定。

燃气灶就是开环系统中输入 / 输出关系的一个熟悉的例子。对煎锅的加热量受前面板上气门的控制（控制输入），气门上的气压受燃气管道上的压力调节器（通常是屋外的燃气表）控制。一旦点着了火，它就会产生强度正比于气门开度的火焰。但是它无法知道锅是不是达到了指定的温度，也无法根据它来调节火焰的强度。一些老的燃气灶甚至不能确定是不是点着火了。这样，天然气就会被释放到厨房里（这也是故意在天然气中加入一些臭气的原因），它们或多或少是线性关系，但完全是开环关系。如果操作者（也就是厨师）把气门开得过大，炒鸡蛋没准就火大了，意大利面没准也会炒糊了。而燃气灶仍然像没事人一样地继续烧。

在输入 / 输出关系明确和对输出要求不高的条件下，开环控制也是很有用的。但是，开环控制系统不能对系统时刻都存在的变化做出反应，也不能处理瞬态的扰动或者偏差。在条件变化时，必须要有人工的干预。

闭环控制

闭环系统用反馈来实现动态的控制系统。闭环控制系统又叫做反馈控制系统。它可以是线性的、非线性的，甚至是顺序的。

位置控制——反馈控制基础

在闭环反馈控制系统中，一个或多个传感器会对输出进行监视，并且把结果返回到系统中用于调整对被控系统或被控对象 (plant) 的操作。在一个负反馈系统中，反馈的目的是使从加和节点生成的偏差减小（回忆图 9-5）。

比如，图 9-10 展示了一个第 1 章中的水箱控制系统。

这个系统和时间是无关的，在大部分情况下也是线性的，所以它是线性非时变系统。和本章其他例子一样，我们假设系统的变化比较缓慢，从而可以不考虑系统的频率响应。水泵控制器只接受水位设置和反馈的输入。

320

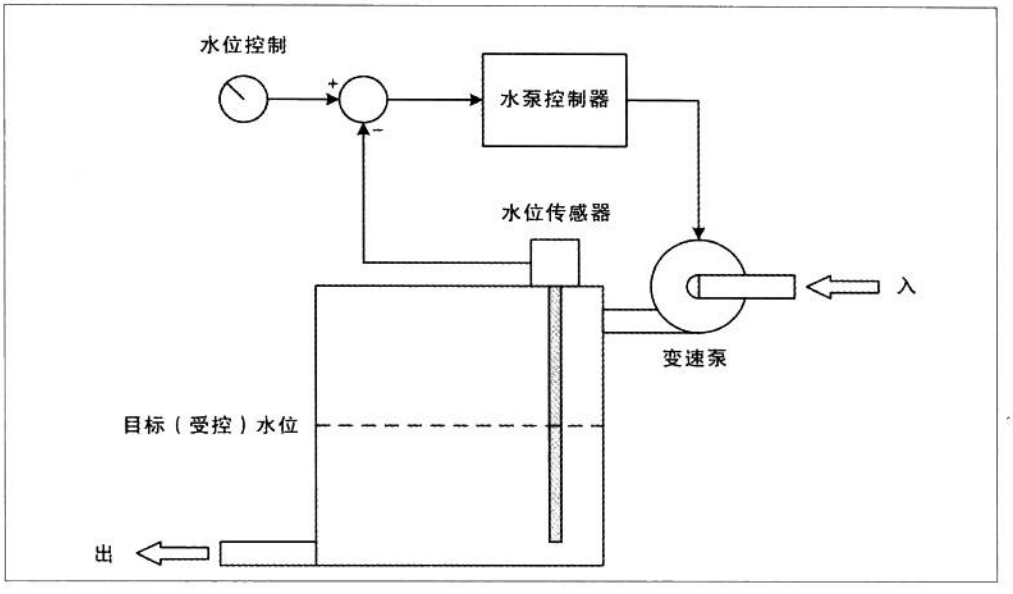


图9-10：闭环水箱水位控制

让我们再来看看这个看似简单的线性系统。图 9-11 显示了系统框图，图 9-12 显示了当水位变化时水泵的响应。

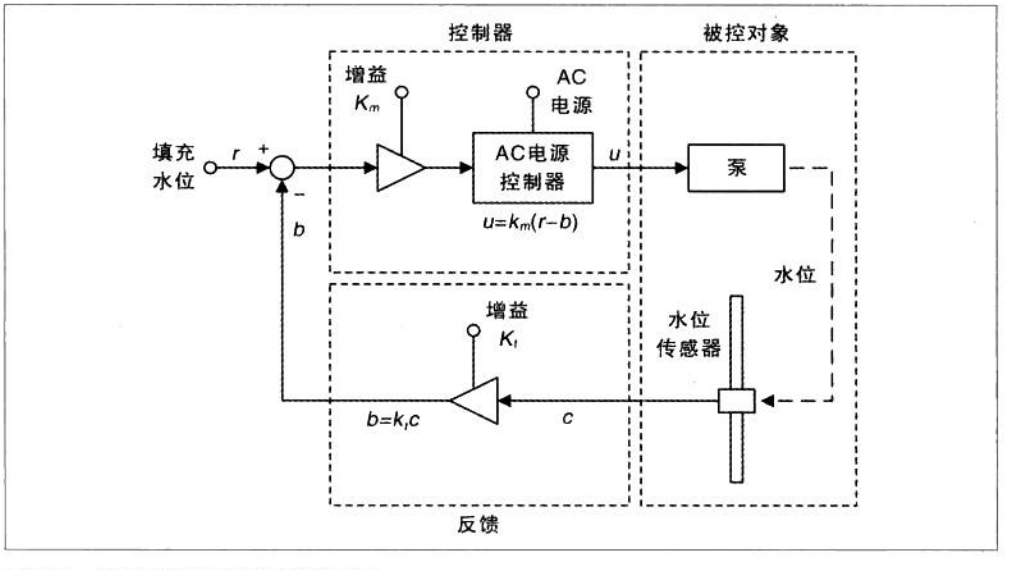


图9-11：闭环水箱水位控制系统框图

系统框图还包括诸如增益可调的放大器和电动机的交流电源控制器等细节。因为这是一个比例控制系统，增益的大小决定了系统的响应能力（回想图 9-1）。增益 K_m 和 K_f 对系统有累积效应。如果增益太小，那么系统就不会让水泵快速地运转，以填补水箱中流失的水。在这个系统里，K_m 是主增益，K_f 是反馈增益。

图 9-12 中显示了随着水从水箱中流出水位的变化情况。图 9-12 中下面的图描述了水泵电动机对水位的响应情况。它们是反向的比例关系。

321

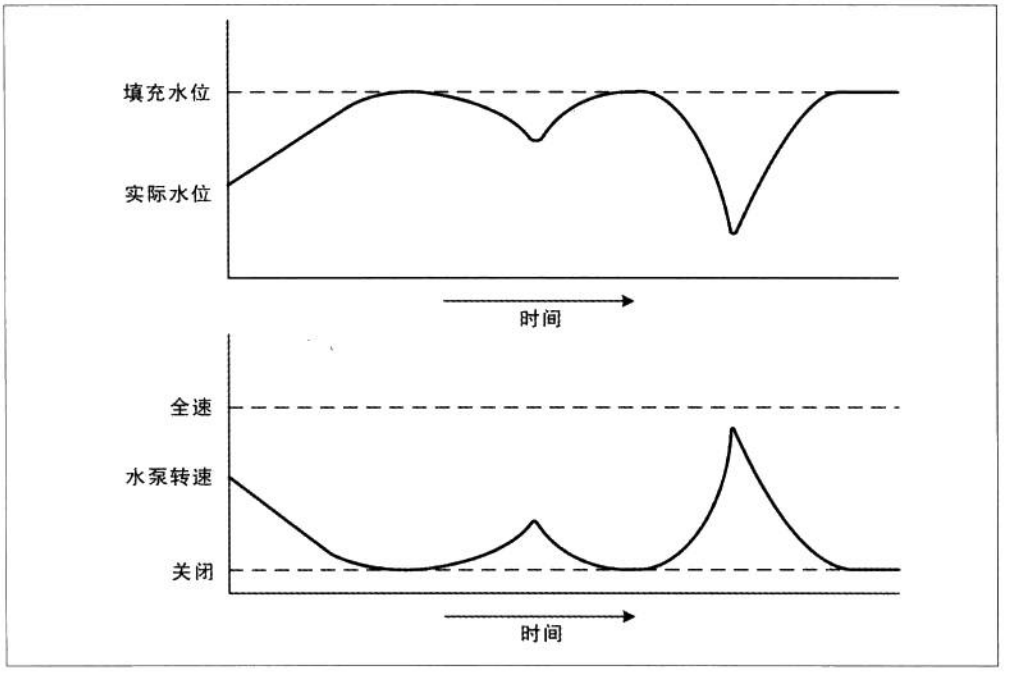


图9-12：水箱控制系统响应图

这是一个位置控制系统——这里的位置指的是水箱中的水位。当水箱水位达到或者超过规定值时，水泵被关闭。当水位不足时，水泵被开启。换句话说，整个系统就是为了控制浮动的水位。水泵正好可以用来达到改变水位的目的。

322

当然，这些图只是一个近似表示。在真实的系统里，你可能会看到电动机动作和偏差值减小之间的滞后，还可能有水位的超调。如果水箱有小但是持续的水流流出，那么水位很可能会在设定值上下振荡。这些和其他问题可以通过在控制器中加入死区、时延和信号滤波等措施来处理。增益值也可以扮演一个重要的角色。调整增益值以优化性能被叫做整定。尽管对于这样简单的系统来说或许不是很难，但对于其他类型的系统来说，可

能是一个挑战。

速度控制——前馈控制和 PWM 控制器

如果不进行修改，简单的闭环控制对于涉及速度的系统就有点力不从心了。原因就在于类似于水箱的系统（见图 9-11）的控制目标是通过控制浮子的位置来控制水位的。一旦浮子的位置达到了要求，偏差就是 0，控制动作也就可以停止了（在理想情况下）。

如果我们想控制一个设备的速度，比如直流电动机，那么控制系统必须通过控制电动机的输入电压来保持传动轴的转速，甚至是在负载变化的情况下。很显然，必须要有一定形式的反馈才能达到这种效果。但是，先考虑图 9-13，它展示了一个简单的开环直流电动机控制器。

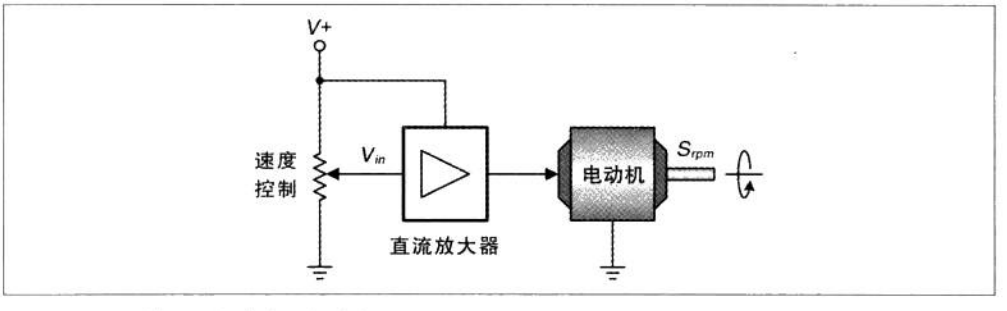


图9-13：简单的开环直流电动机控制

323 这是一个线性控制，也就是加在电动机上的电压是直流放大器的线性函数（V_{in}），传动轴速度（S_{rpm}）又和电动机的输入电压成正比。放大器是关键器件，因为直流电动机会吸收电流。如果需要的话，放大器也可以提供一些增益。

以最简单的形式，图 9-13 的方程是：

$$S_{rpm} = M_r G V_{in} \qquad \text{（方程 9-3）}$$

式中：

S_{rpm} 是电动机输出转速（转 / 分钟）；

M_r 是电动机的响应系数；

G 是放大器的增益；

V_{in} 是放大器的输入。

在电动机开环控制方程中，系数 M_r 代表电动机本身的一些因素。它包含与负载和转速有关的电动机的电磁特征。但是在这里我们可以把它们统统放到 M_r 中。

如果我们想设定电动机的速度并且保持这个速度，那么就需要某种类型的反馈。假设我们在电动机轴上有速度计，我们就可以用它作为反馈环的输入。

像图 9-11 那样的闭环控制系统也是可以用的，不过需要调整 r、b 和 e 的增益值，稳定性也是一个问题。任何负载的变化都有可能引起传动轴速度的振荡。根据各种增益值的大小，振荡需要一定的时间才能衰减。

另一个方案是在系统中加入一个能够检测系统负载变化的前馈通道。在前馈控制器中，控制值直接被送到被控对象上，然后它再产生预期的响应。听起来很熟悉，对吧？应该是的。前馈控制实质上就是图 9-13 那样的开环控制。实际上，前馈的另一个名字就是开环。

为了实现一个定速度控制系统，我们可以加入一个或多个反馈通道来抵消负载的影响，增强速度的稳定性。这些通道的作用合起来对电动机进行控制。整个系统的框图如图 9-14。这里，来自直流转速计（其实就是一个直流发电动机）的反馈用来提供稳定性。电流探测器用来检测加在电动机上负载的变化。前馈输入、速度偏差和来自电流控制器的反馈一起控制电动机。[译注 6]

324

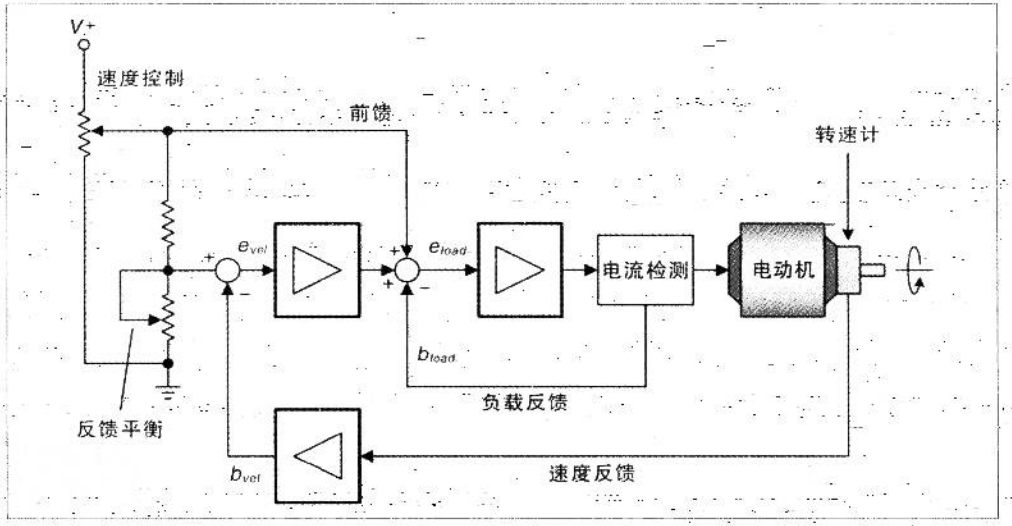


图9-14：前馈直流电动机速度控制器

当电动机的输出转速达到参考输入时，速度偏差 b_{vel} 会变为 0。注意，参考输入和速度

译注 6：作者在这里用的不只是前馈控制，它实际上是一个带前馈通道的串级控制系统。

输入是呈正比的。如果电动机上的负载发生变化，可以用 b_{load} 来调整电动机的输入电压，以达到抵消负载的变化（在转速一定时，低负载时的电流比高负载时要小很多）。图 9-15 显示了偏差值 e_{vel} 是如何提高系统稳定性的。

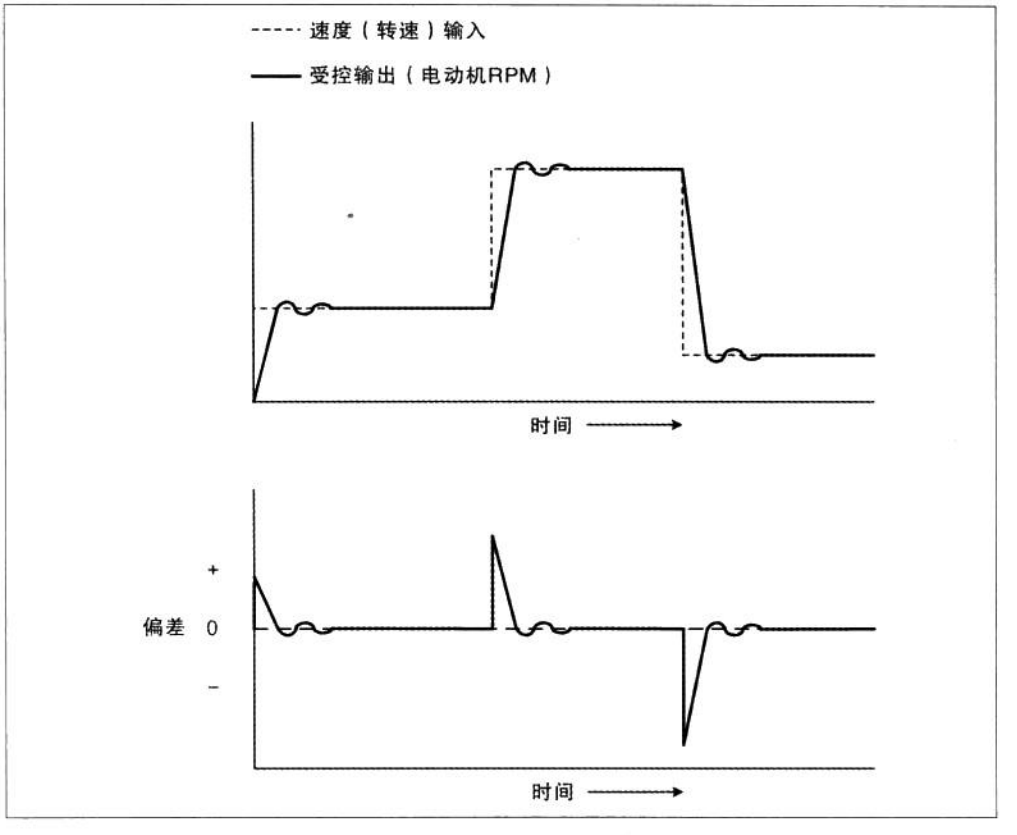


图9-15：设定值变化时偏差值的变化

另一个控制电动机转速的方法是用脉冲来控制电动机和反馈速度，如图 9-16 所示。对于电动机速度控制，脉宽调制（pulse width modulation，PWM）比变压控制有更好的电效率（electrical efficiency）。PWM 控制器不用 DAC 器件产生控制信号，更容易实现。脉冲型的编码器可以在电动机轴转一圈时产生一个或者两个，甚至是四个脉冲。所以，我们在一段时间内记录脉冲数，就可以计算出电动机的转速。因为是 PWM 输出，所以电气上是简单的，但需要注意的是，这种连续可变控制的控制器却是一种非线性控制。它
325 严格依赖于编码器输入、所需转速的设定和调制 PWM 输出，最终的结果类似于图 9-3。

这里已经展示了一个基本的电动机控制器所需要的器件，你应该也对各个部分有了了解。

这些控制器一般由电子电路或者微控制器模块的构成。如果你需要控制电动机的速度，在此建议你购买商用电动机控制单元。图 9-17 显示了一个应用了电动机控制单元的系统框图。

在图 9-17 中，输入电动机控制器的指令需要符合控制器制造厂商所设计的标准。一般来讲，接收 ASCII 字符串的控制器会用一个字符作为指令（设定方向、速度、时间等），然后每个指令对应合适的数据参数。用 RS-232 或者 RS-485 进行通信是比较普遍的，但也有一些电动机控制器以总线的形式进行通信。

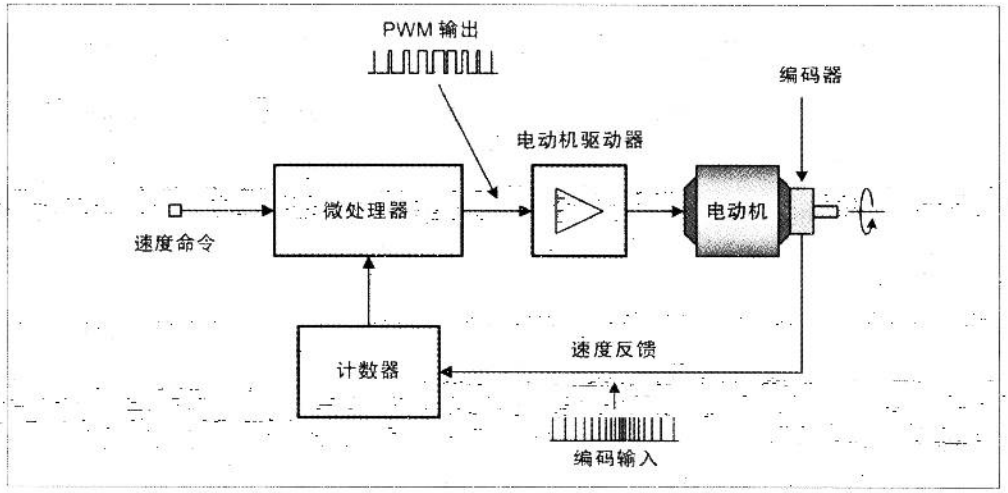


图9-16：PWM电动机速度控制

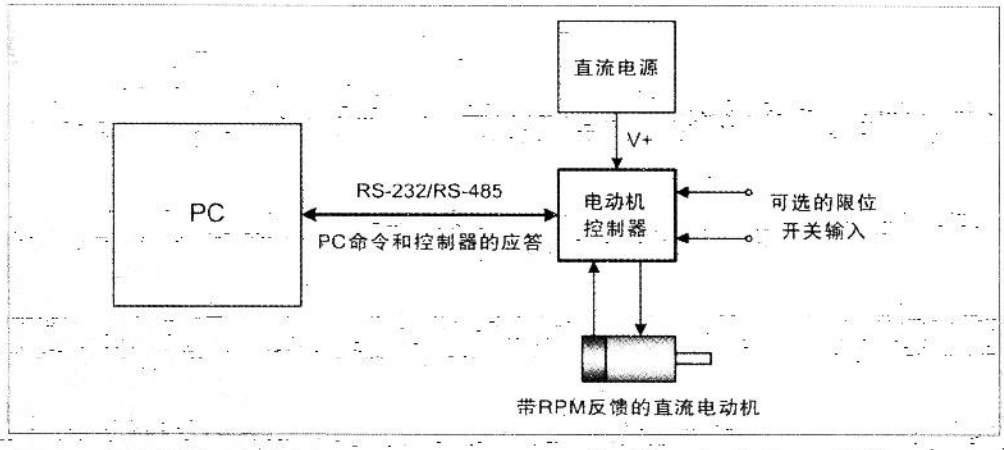


图9-17：商用直流电动机控制器

非线性控制：继电器控制器

继电器控制器也称开关控制器，是一种简单的非线性控制系统。之所以叫做继电器控制器，是因为它对于线性输入的响应像继电器一样，不是开就是关，要么全有，要么没有。在电器时代之前，它们都是由在两个触点之间移动的控制杆制作的。每当控制杆从一个
327 位置移动到另一个位置时，就会发出“嘣”的一声[译注7]。一般家用的热水器和空调都是闭环的继电器控制系统。我们在第 1 章中看到的自动脚灯控制就是开环的继电器控制系统的一个实例。

非线性控制系统经常会有滞回现象。也就是系统响应对于控制输出有一个滞后。打开和关闭都可以有这个滞后。用机械术语来说，我们可以把它想成一种“快速动作（snap action）”。一个常见的机械滞后的例子就是三环笔记本（three-ring notebook）。当用一定的力度打开时，三个环会突然张开。闭合时也是一样的。如果那些环在受力时简单地张开或者闭合，那它们就没有那么好用了。

在继电器控制器里，可以用滞回来调整控制动作。图 9-18 显示了空调中恒温器里的滞回现象，它也展示了如果没有滞回作用，恒温器会发生什么现象：频繁地切换开关会迅速用坏空调。

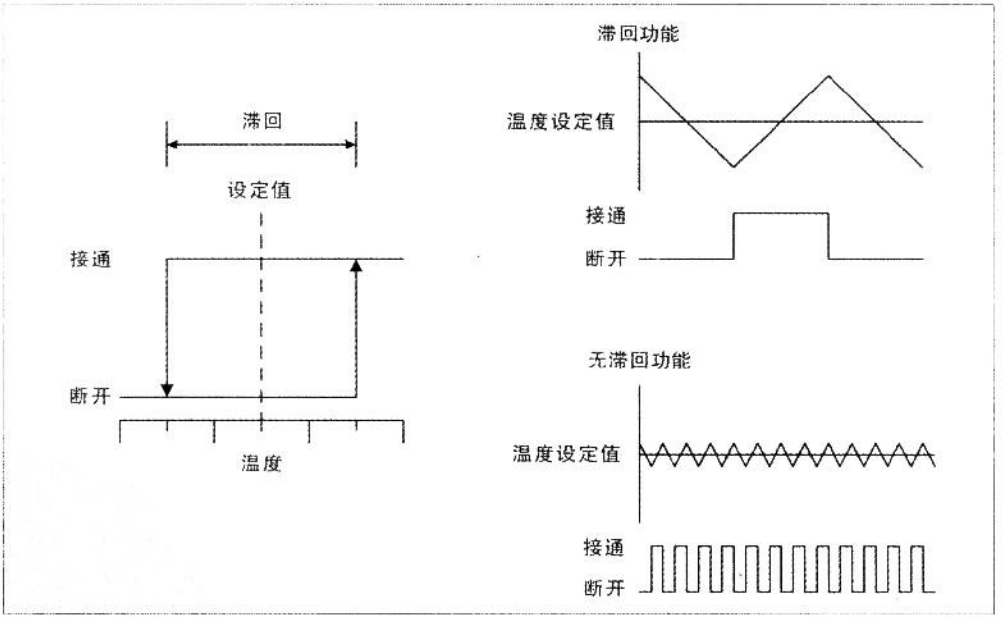


图9-18：滞回作用

译注 7：继电器控制器的英文是 bang-bang controller，这一段是对英语原词的解释。

因为在图 9-18 中显示了滞回作用，空调在温度稍微高于设定的温度时才会被开启，直到温度略低于设定的温度时才会停止工作。这不仅意味着温度会在设定值附近摇摆，而且意味着空调不会持续地、频繁地被开启和关闭。如果没有滞回作用，恒温器会努力把温 328
度保持在设定值上，这就意味着频繁地切换空调的电源。在一般情况下，这对于制冷系统的压缩机来说可不是一件好事情。

机电的继电器控制器是简单、鲁棒的器件。它们主要依靠滞回作用来达到一个合适的响应能力。但是，如果继电器控制器是一个由软件组成的离散控制系统，那么系统中各种时间常数就会对系统响应有决定性的影响。

图 9-19 是一个空调系统简易的控制器流程图。在这里，滞回是由加在设定值 s 上的偏移量 H 决定的。如果探测到的温度 t 在设定值加上偏移量之上，空调就会被开启。反之亦然。

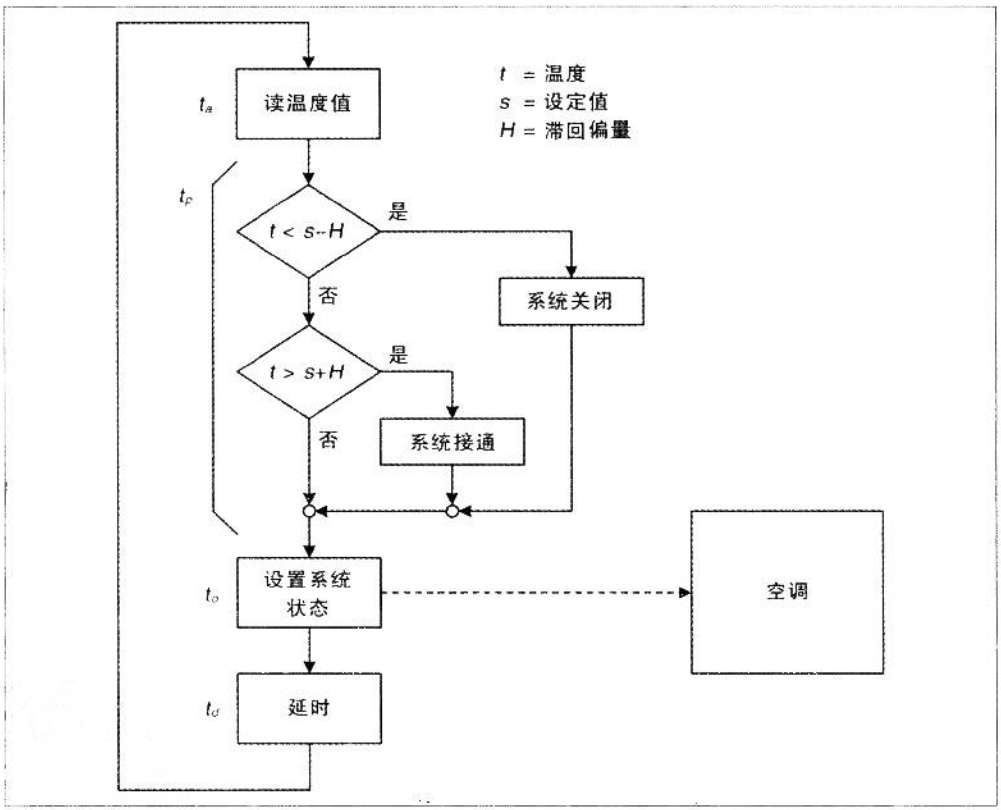


图9-19：软件继电器控制

如果回头看看图 9-9，然后再看图 9-19，我们可以发现，t_a、t_p 和 t_o 所占用的时间可以忽 329

略不计。在控制器循环周期里占用时间最多的是延时时间 t_d。

控制循环所占的时间越短越好（所谓短，是和被控系统相对应的，可能几个毫秒的量级也是可以的）。这是离散时间系统，所以就不会有像机电系统或者模拟控制器那样的连续输入响应。采样频率需要足够高才能避免过多地超调或欠调（undershoot）。在高采样频率下，滞后作用对控制输出起了决定性作用。

确定 t_c 最佳值时必须要考虑受控系统的响应特性。图 9-20 显示了一个理想的继电器控制器在相对高的采样频率下的响应。我们假设 t_a、t_p 和 t_o 占 t_c 的时间很少，所以就没有画出。

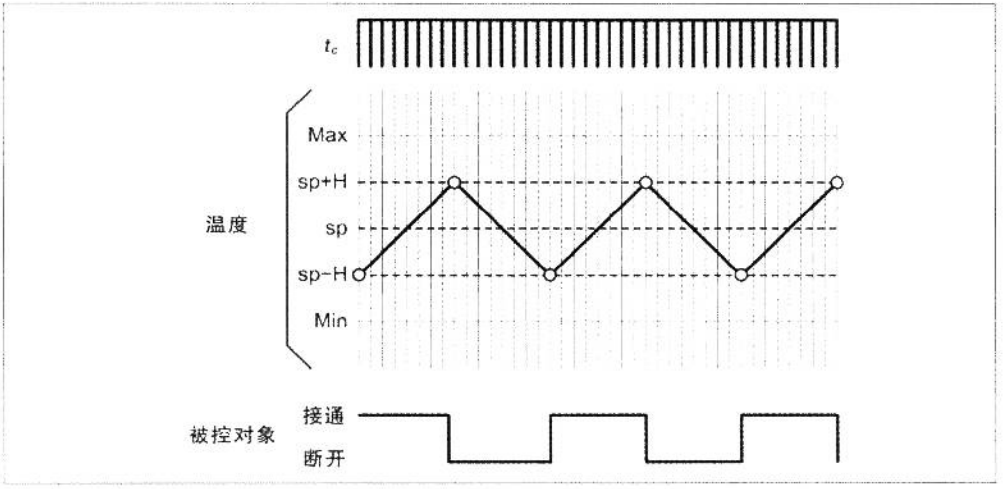


图9-20：继电器控制响应

控制器的循环周期和滞后作用一起决定了离散继电器控制器的总体响应能力。在实际中，因为控制器跟不上变化，所以继电器控制不能在频繁变化的场合应用。如果 t_a 和 t_o 之间的时间相对于变化速度来说过长，控制输出会在那段时间里发生明显的变化。这会导致超调或者欠调，甚至超过所允许的范围。

330

顺序控制系统

一般来说，顺序控制系统实现起来比较直接，从非常简单到极其复杂都有。它们经常用在需要一系列动作才能完成工作的场合里。之前，我们把自动洒水系统作为顺序控制的一个例子展示给了大家，下面会介绍一个略微复杂和有趣的例子。

图 9-21 展示了一个顺序控制的机器人。它的机械结构由一个水平安装的导轨、一个定速的电动机、一对极限传感器和一个工具（tool head）组成。该系统可用于生物样品的转移，

或是在框架之间穿线，也可以在同步另一个机器人在其他位置上的所有动作。

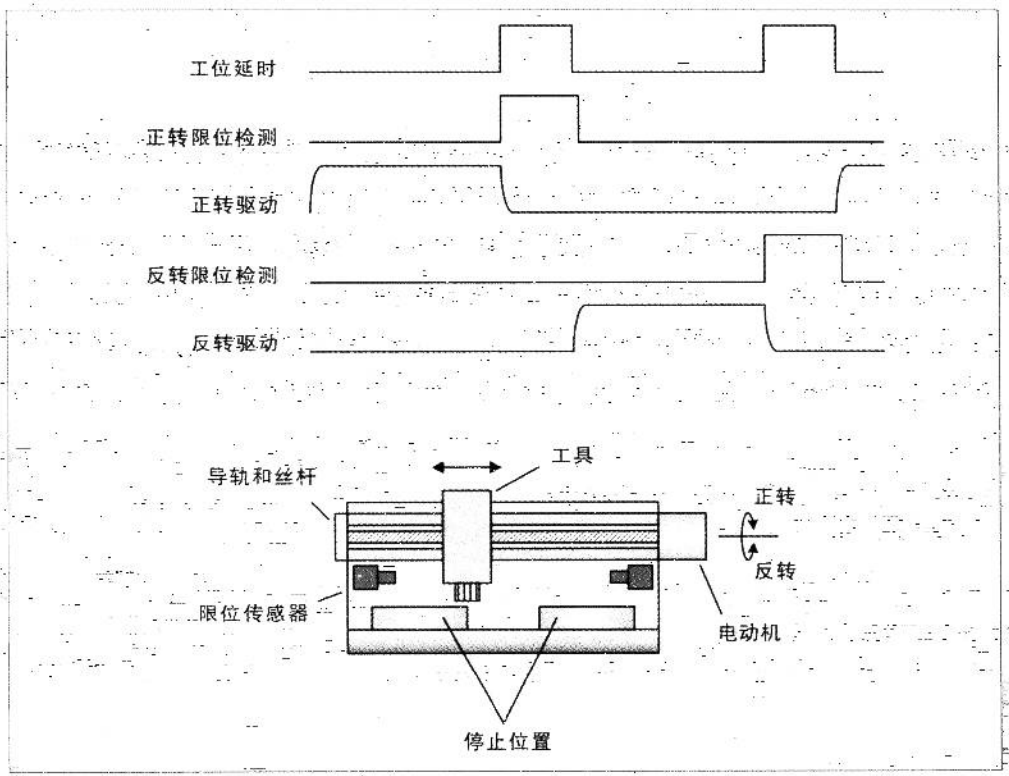


图9-21：顺序控制的机器人系统

这个机器只有一个自由度（即在一个方向上运动），不是左就是右。在图 9-21 中用顺时针和逆时针来表示电动机转动丝杠的方向。它并不在工具移动的过程中追踪工具的位置，它只在工具走到停止位的时候才会有感应。极限传感器的位置决定了工具的停止位置。 <331>

在图 9-21 的顺时针和逆时针时序图中，电动机的运动没有突变，这是因为电动机是有惯性的，它需要一定的时间才能达到全速，断电之后也需要一定的时间才能完全停下来。注意，极限传感器的状态在工具反向移动到极限位置时会立刻发生变化，而在电动机正向移开极限位置时不会立刻变化。这是因为工具需要一定的时间才能从极限传感器下移开。最后，因为工具已经在最边上并且正在运动，所以我们可以假定当电动机顺时针旋转时不需要检查逆时针方向的极限传感器（它也应该是处于激活状态）。对于逆时针旋转也是一样。

我们之前说过，可以把顺序控制系统看做是状态机，这在图 9-22 中可以看出。当工具到

达极限传感器时，电动机停止转动，工具的运动也就停止了。在工具返回另一边之前，它会等待（延时）一段时间。整个循环只有在被外部中断时才会停止。

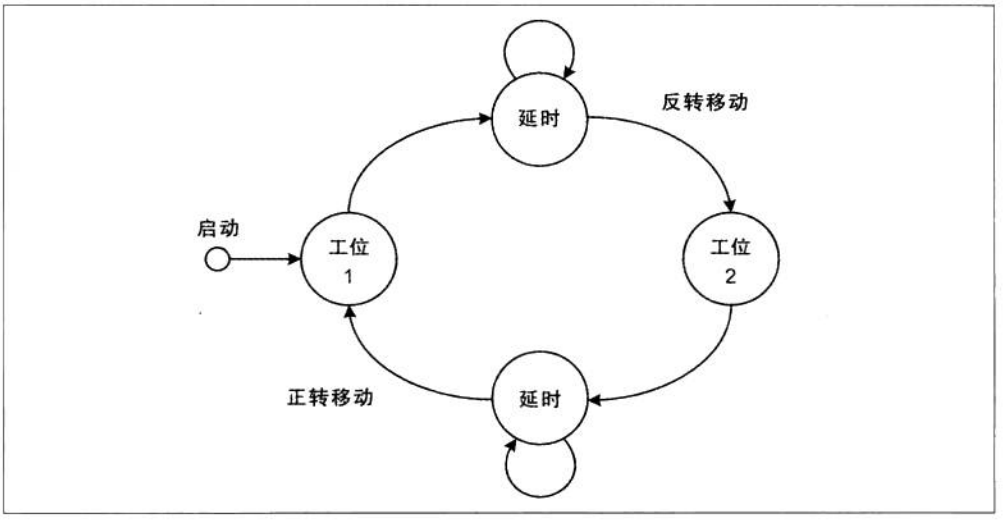


图9-22：顺序控制系统状态

在真实的系统中，你可能还会用到一些错误检查，比如，用超时来检查电动机或者工具是否被卡在两个极限传感器中间了。如果用一个定速电动机，那么工具从一个位置移动到另一个位置的时间应该是一个常数，比如几十毫秒。如果移动的时间过长，就需要注意了。

332

顺序控制系统在工业生产环境中使用非常普遍。它往往采用可编程逻辑控制器（PLC）设备来实现。标准框图是一种常见的顺序控制系统的描述方式。PLC 技术也有其自己的框图类型，被称为梯形图和顺序功能图（SFC）。流程图可以用来描述连续的系统，但实际上除了简单的设计外，它太冗长了。

如果要实现顺序控制器，那么值得探讨都有哪些图形化的手段可以使用。就个人而言，我们恰巧喜欢 SFC 型图，因为它提供的抽象层次比梯形图的更高，结构比流程图更紧凑。图 9-23 展示显示了 IEC 61131-3 型的 SFC 图。

在图 9-23 中，跨线的加重线表示一个条件，当它们为真时，才能继续执行到下一个路径。在每个步骤中可以描述采取的行动，为随后的条件测试需要提供什么输入，或需要执行什么系统函数。

图 9-23 描述出了一个重要的事情，就是顺序控制器不仅可以执行固定的一系列步骤，它

也可以有分支。换句话说，一个顺序控制系统不仅可以结合 if-then 类型的决策点和有条件的循环，也可以结合闭环式的反馈。

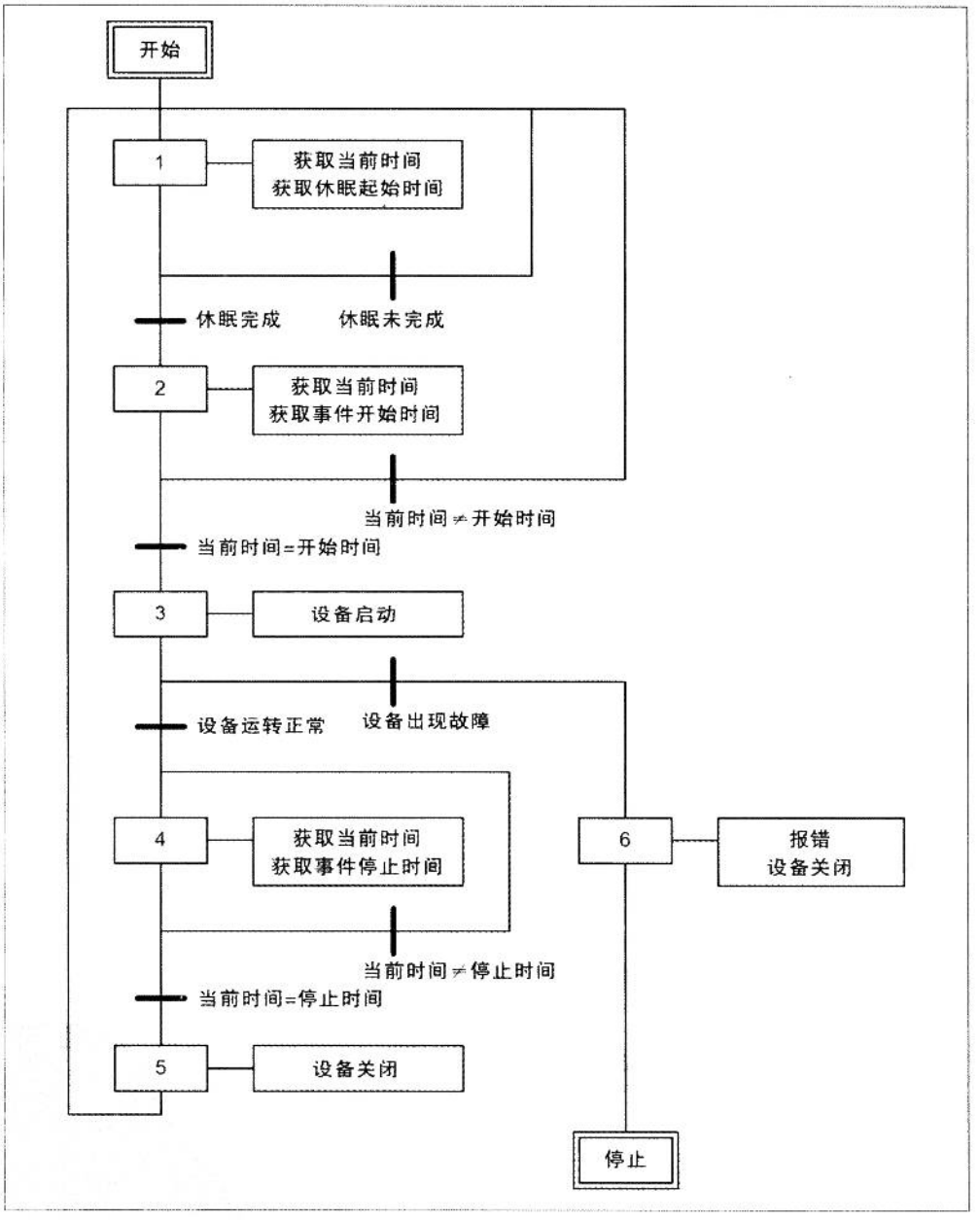


图9-23：SFC图示例

比例、比例积分、比例积分微分控制

比例控制是线性反馈控制系统的关键。比例控制器比继电器控制器要复杂一点，但是因为它可以根据变化的环境来改变输出，有一个平滑、连续的线性控制性能，所以它具有明显的优势。但是单纯的比例控制也有缺点，比如“静差（Droop）”和对突变的响应不好。

我们已经见到了一些基本的比例控制的例子（比如，图 9-11），但是现在我们要更深入地了解它们。我们还会看到可以通过加入积分和微分的作用，使系统变为 PI（proportional-integral，比例积分）和 PID（proportional-integral-derivative，比例积分微分）来消除纯比例控制的缺点。

334

买现成的还是自己写

市场上有许多针对不同场合的商用 PID 控制器[译注 8]。比如，有些是专门针对温度控制的，有些是针对压力控制的，以及其他针对运动控制设计的，等等。其价格也不同，从 100 美元入门级的 PID 伺服控制器到接近 400 美元的工业级模块化的温度控制器，以及几千美元高可靠性的工业控制单元都有。

所以，是应该买一个 PID 控制器，还是自己用软件写一个呢？答案取决于你对硬件的预算是多少、你手上设备的数据 I/O 方式，以及你愿意花费多少时间来实现一个自己的 PID 算法并且让它正确地运行。

如果你只需要一个简单的控制系统，对系统响应的要求也不是那么高，你或许可以用一个类似于我们将要展示的 Python 的程序来实现。另一方面，如果你的应用需要高精度、高实时性的响应，你就应该去买一个商用的控制器。还需要指出的是，如果你并不是真的需要一个 PI 或者 PID 控制器，你或许也应该省下力气来，干脆就不用。

一个真实的 PID 故事

很久以前，我们为一个大型数控立式铣床做伺服控制。铣床是用来把铝或者钢切削成不同形状的，它们一般都非常大，而且非常重。

系统里的一个 PID 伺服控制器对设定点的跟踪有问题。它会慢慢地在设定点上来回移动，刚好能够切掉一些金属，使得你能够在切削面上看见小波纹。因为这个机器是用来生产用于科学研究的高精度设备部件的，所以，这种情况是不被允许的。

译注 8：作者在这里针对的是美国市场。但是对国内市场也有一定的参考价值。

我们的任务就是弄明白导致问题的原因是系统里的噪声还是一个松动的部件或一个坏掉的控制器，进而排除它们。

当我们在机器边上的控制室里工作的时候，我意外发现了一个问题：在问题伺服控制器上的一个虚焊。不幸的是，这正好在反馈回路上。也就是说，那个控制单元没有接受反馈的调节，我们当时正在一个运转的系统上工作。当仔细检查问题区域的时候，我们听到系统中一个运动部件的伺服电动机达到了全速。它已经失去了反馈，并且正竭尽全力地向正无穷奔去。正当我们跳起来按下急停按钮的时候，整个平台——总共有 600 磅——砰地一下撞到了轨道的尽头。平台的惯性大到它通过极限开关的时候都没停。

突然的撞击造成了巨大且刺耳的金属撞击声，两吨重的轧机甚至都轻微地跳了起来。声音的一部分是由平台的丝杠和驱动螺母达到极限位置造成的。它是非常昂贵的滚珠丝杠，但螺母还是被撞碎了。于是我们被由几百颗滚珠形成的滚珠雨浇了一身，335破碎的螺母散落了一地，整个车间的机械工都来了，只看到这一团乱麻的场景。我们坐在地上看着在头顶晃来晃去的钢架，正琢磨发生了什么事情。后来，在我们换上直径一英寸的不锈钢丝杠（旧的那个弯掉了）、驱动螺母、限位开关、极限位置和伺服控制器之后，机器正常工作了。对我们来说，这是一个珍贵（也是代价高昂的）的经历。它教给我们怎样去设计一个危险的系统和在处于运行状态的系统上工作时需要注意什么。

PID 概览

工业控制应用领域中，大多数控制器都是 PID 控制器，无论是线性的还是非线性的。线性 PID 控制器已经以各种形式存在了 100 多年。最早，它们是基于控制杆、齿轮、阀门、活塞和风箱的机械、液压、气动装置。因为技术的发展，DC 伺服、电子管和晶体管开始应用了。所有这些都是在连续的时域里工作的。随着计算机控制系统的出现，PID 控制的分析和实现开始转移到非线性和离散时域中，并且诸如采样时间和采样精度之类的课题变得越来越重要。我们会以 PID 在连续时域中的基本理论入手，然后看看怎么样把它转化到离散时域中，并用软件实现 PID 控制。

一个完整的 PID 控制器由三个部分（或者说是项）组成。它们在图 9-24 中被展示出来了。

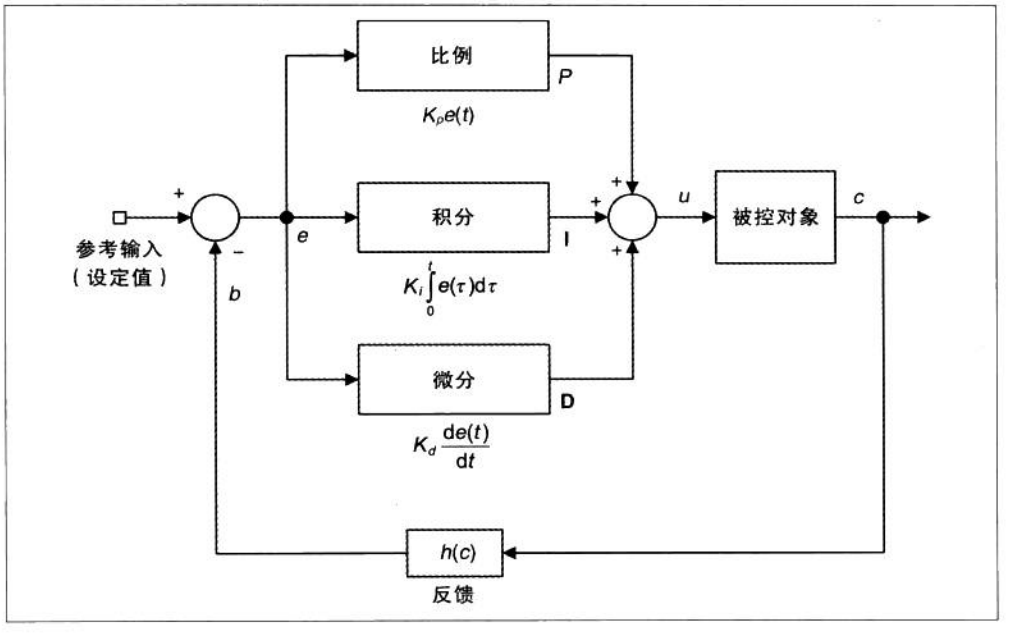


图9-24：PID控制器框图

PID 控制器的输出是由三个项的和组成的：

$$u = P + I + D$$

其中的每一项在决定系统稳定性和响应特性中都起着特殊的角色。

从数学上讲，理想的 PID 控制器可以写成方程 9-4 的形式。

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \qquad \text{（方程 9-4）}$$

其中：

e 代表系统偏差 $r-b$；

336 K_p 代表比例增益；

K_i 代表积分增益；

K_d 代表微分增益；

t 代表时刻；

u 代表控制输出；

τ 表示积分时间（不一定与 t 一样）。

实际上，更常见的 PID 方程形式如方程 9-5 所示。

$$u(t)=K_c e(t)+\frac{K_c}{T_i}\int_0^t e(\tau)\mathrm{d}\tau+K_c T_d\frac{\mathrm{d}e(t)}{\mathrm{d}t}+u_o \qquad \text{（方程 9-5）}$$

方程 9-4 和方程 9-5 都可以用于在连续时域里描述 PID 控制。它们主要的差别在于在方程 9-5 中，增益 K_c 应用到了所有的项上。I 和 D 自己的表现是由 T_i 和 T_d 决定的，它们分别被叫做积分时间和微分时间。因为你可能会遇到这种只用一个增益描述的 PID“标准形式”（方程 9-6），所以这里提及了它。 337

$$u(t)=K_c\left[e(t)+\frac{1}{T_i}\int_0^t e(\tau)\mathrm{d}\tau+T_d\frac{\mathrm{d}e(t)}{\mathrm{d}t}\right] \qquad \text{（方程 9-6）}$$

在 PID 控制器中，比例项是主要项，积分和微分项起的作用很小（有时是小很多）。实际上，一个 PI 控制器正好是 D 项为零的情况。你还可以做一个表现得完全像比例控制器的 PID 控制器，只需把 I 和 D 设置成零就可以了。

比例控制项

比例控制是一种线性反馈控制，我们已经在本章“线性控制系统”部分看到了一些例子。这里要介绍的是比例控制的一些缺点，以此作为介绍积分项和微分项的前奏。

当输入信号缓慢改变和受控对象上的反馈没有突变时，比例控制还是工作得不错的。但是，比例控制不能很好地处理突变和瞬态过程，也很可能产生超调和欠调。如果变化太快，系统或许会跟不上设定值（参考输入）。图 9-25 给出了对于不同的 K_p 值，比例控制系统对阶跃输入的响应。

阶跃输入在系统的操作过程中虽然永远不会用到，但它对于分析控制系统响应的特性非常有用。在图 9-25 中主要可以看出 K_p 是怎么影响系统对输入值的快速变化的。高增益的系统会使系统的响应更快，但是它可能会产生超调，然后在设定点附近振动一段时间。如果增益足够高，系统可能永远不会稳定下来，而且如果增益过高了，系统还可能产生振荡。相反，如果增益太低了，系统在可以测量的时间内就无法做出响应，并且有很大的静差。

338

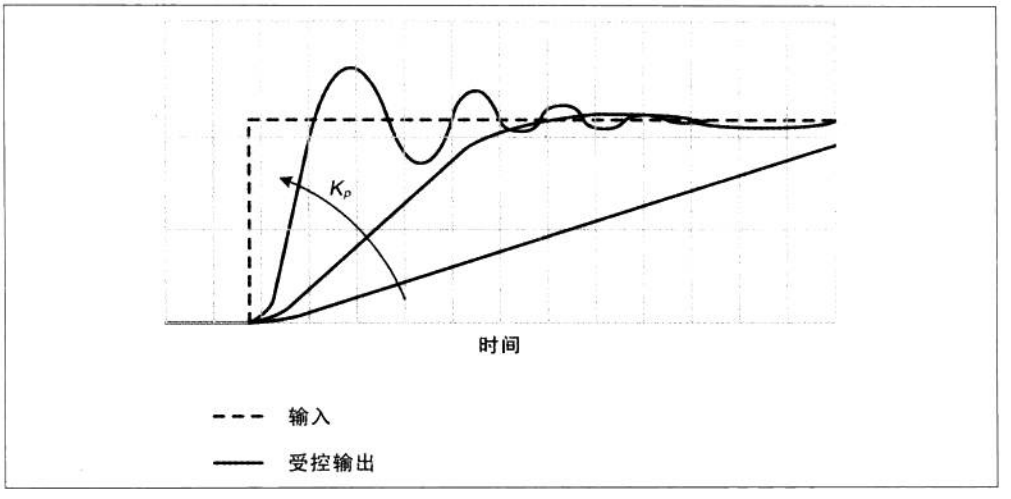


图9-25：比例项控制响应

“静差（Droop）”是比例控制系统中一个具有代表性的问题，它表现为永远不会达到设定值，相反，它会有一个低于设定值的稳态误差。这主要归结于控制器的增益或被控对象的增益不同。静差可以由在输出上加入一个偏移量来缓解，或者用积分项来解决，就像 PI、PID 型的控制器一样。图 9-26 展示了静差的效应。

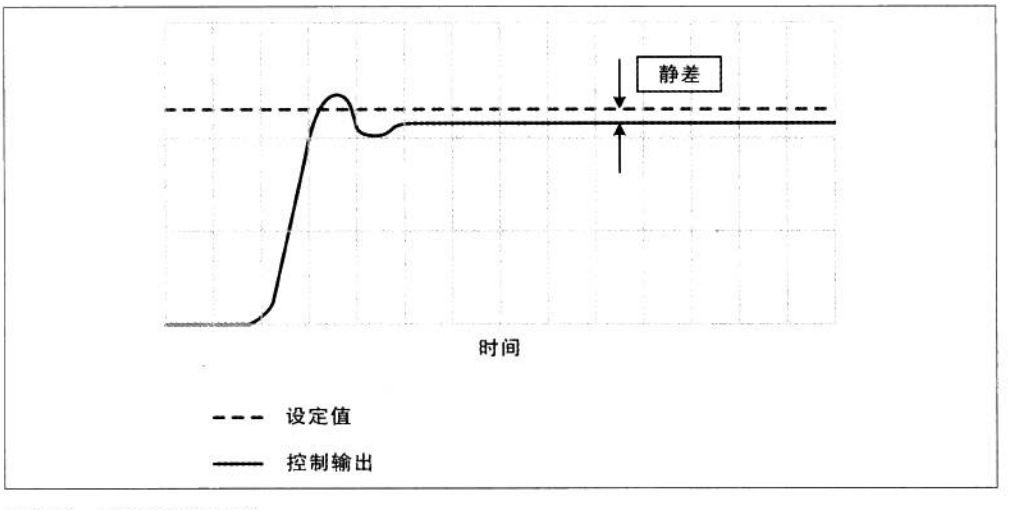


图9-26：比例控制的静差

339 还会有外部因素影响比例控制对控制输入的响应。它们包括被控对象的响应、延时和瞬态输入。

PI 和 PID 控制

积分项也被称为归零项（reset），用于消除静差。在输出中加入积分项的目的就是积累输出和设定值之间的偏差，从而使输出更快地向设定值靠拢。因此，比例增益 K_p 就需要根据 I 项的输出而相应地调小。

如果我们还记得偏差的结果是 $r-b$，那么积分项的值显然在偏差大的时候增长快，而输出越接近参考输入，积分项增长得越慢。所以，对比单纯的 P 项来说，有了积分项之后，就会驱动系统向设定值更快地移动。但是，如果 K_i 太大，系统就会出现超调，还可能变得不稳定。一般都会根据特定的应用来调节积分项的大小。

在一个完整的 PID 控制器中，微分项是用来减慢控制器输出变化率的。它的作用是指示系统接近设定值的速度，所以，微分项的唯一作用是限制或防止超调。但是，微分项也会放大噪声。如果 K_d 太大，瞬态和噪声可能会使得系统变得不稳定。

当所有的三项都起作用并且被正确地整定后，就会有如图 9-27 所示的响应。

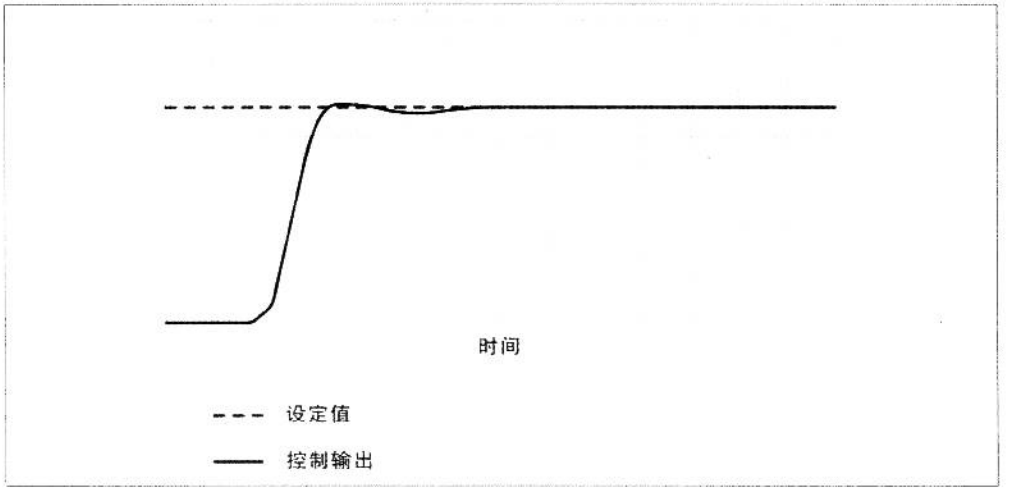


图9-27：整定后PID控制器的响应

混合控制系统

340

顺序控制系统和线性控制系统的区分并不总是那么明显。在控制系统中不难发现混合有各种控制形式，因为它总是会包含各种类型的子系统。

考虑一个在啤酒厂里灌装啤酒的控制系统，这个系统可能是由许多子系统组成的。一个子系统控制瓶子输送机，它的功能应该是保证空瓶子在特定的时间出现在注射喷嘴下面。

为了实现这个功能，它必须考虑到不同瓶子的重量来精确控制输送机的速度。另一个子系统可以控制灌装操作。它需要感知瓶子出现在喷嘴下面的时间，然后注射一定数量的啤酒。注射啤酒的量是时间的函式(阀门会打开几秒)。你还可以想想更多的因素。很快，啤酒灌装机就会变成一个非常复杂的系统。它本身是由许多相互关联的子系统构成的(有些是顺序控制，有些是线性控制，甚至还可能有非线性控制)。

用 Python 实现控制系统

我们首先以一个简单的线性闭环比例控制函式开始介绍本节。或许它看起来并不多，但是它却有一个基本比例控制所需的所有环节。接下来是一个以简单开关控制器形式呈现的非线性控制。它对于一个只制冷不制热的空调系统来说足够了。但是添加制热的功能和制冷一样直接，而且不会有任何困难的挑战（它与制冷相反）。

最后，我们会实现一个基本的线性 PID 控制器，并且把方程 9-4 写成离散形式，从而可以方便地用 Python 实现。

在第 10 章会展示一个可以用来获得真实数据并生成响应曲线的模拟器。

线性比例控制器

一个线性比例控制器非常简单。回忆一下本章开始时看到的公式：

$$u(t) = K_p e(t) + P$$

如方程 9-7 所示，我们还可以用输入 r 和 b 来展开这个公式：

341

$$u(t) = K_p(r(t) - b(t)) + P \qquad \text{(方程 9-7)}$$

下面是实现方程 9-7 的代码：

```
"""简单比例控制。

获得参考输入数据和反馈数据，并且根据下述公式生成一个比例控制值：

    u = Kp(r - b) + P

b 等于 c*Kb，c 是受控对象或系统的输出。Kb 是增益，它可以使
b 被缩放到和 r(参考)相同的量级。

Acquire 是一个占位函式，请用真正采集变量的函式替换它。作为传递给 Acquire 的参数，
rinput 和 cinput 也应针对实际的系统进行修改。

Kp 和 Kb 对于特定的应用有特定的值。参数 P 是加在输出上的偏移。
```

```
"""
# 本地全局变量。可以用 module.varname 来从外部获取它们。
Kp = 1.0
Kb = 1.0
P  = 0

# 根据实际情况修改
rinput = 0
cinput = 1

def Acquire(port):
    return 1

def PControl():
    rval = Acquire(rinput)
    bval = Acquire(cinput) * Kb
    eval = rval - bval
    return (Kp * eval) + P
```

在这个例子中，函式 AnalogIn() 只是占位函式[译注9]，需要根据具体的应用来替换它们。同样，变量 rinput 和 cinput 也应与具体的意义关联起来。

开关式控制器

我们以前说过，开关式控制器是一种输出非线性但输入线性的控制器。在本例中，我们假定非线性控制器的响应由一高一低两个阈值决定。看下列代码的时候，你或许需要参考方程 9-3。

```
import time      # 延时需要

# 假常量
OFF = 0
ON  = 1

H = 2.0              # 滞后范围
delay_time = 0.1     # 循环延时

# 需要根据真实的输入和输出修改
temp_sense = 0
device     = 0

def BangBang():
    do_loop = True
    sys_state = OFF

    while (do_loop):
```

342

译注9：原文是 dummy-placeholder，意为假的占位符。但它又是函式，所以暂译为“占位函式”，下同。

```
if not do_loop:
    break

curr_temp = AnalogIn(temp_sense)      # 占位函式

if curr_temp <= set_temp - H:
    sys_state = OFF

if curr_temp >= set_temp + H:
    sys_state = ON

# 这里假设设置相同的值不会引起问题，并且输出只会在设置变化时才变化
SetPort(device, sys_state)            # 占位函式

time.sleep(delay_time)
```

这个函式直接与方程 9-3 相对应，它与我们第一次介绍非线性控制时的行为是一样的。与本节中的其他例子一样，它没有错误处理。它也可以通过扩展从而同时支持制冷和制热。而且，AnalogIn() 和 SetPort() 只是占位函式，变量 temp_sense 和 device 也需要根据实际情况修改。

简单 PID 控制器

用软件实现 PID 控制器算法的第一步就是把形如方程 9-4 中连续的 PID 控制函数离散化。

首先，我们有积分项的一个离散近似（方程 9-8）：

$$\int_0^t e(\tau)\mathrm{d}\tau \approx T_s \sum_{i=0}^{t} e(i) \qquad \text{（方程 9-8）}$$

343 式中：

$e(i)$ 表示第 i 步的偏差值；

i 表示积分步骤；

T_s 表示积分步长（Δt）。

然后是微分项[译注 10]（方程 9-9）。

$$\frac{\mathrm{d}e(t)}{\mathrm{d}t} \approx \frac{e(t) - e(t-1)}{T_s} \qquad \text{（方程 9-9）}$$

译注 10：原文为 derivative form，但是根据公式的内容疑为微分形式。

式中：

t 表示时刻；

$e(t-1)$ 表示前一个 e 值。与 $e(t)$ 之间的间隔是 T_s。

注意，我们现在是在用离散时间，所以 t 只是为了在离散和连续形式间建立联系。

我们不用去管方程 9-4 中的比例项，它已经简单到可以直接翻译成代码了。

现在我们就可以把离散近似代入方程 9-4，得到方程 9-10。

$$u(t) = K_p e(t) + K_i \left(\frac{T_s}{T_i} \sum_{i=0}^{t} e(i) \right) + K_d \frac{T_d \left(e(t) - e(t-1) \right)}{T_s} \qquad \text{（方程 9-10）}$$

在方程 9-10 中：

T_i 表示积分的时间步长。

T_d 表示微分的时间步长。

T_s 表示积分步长（Δt）。

虽然这种形式考虑到了时间因素，能产生更好的积分和微分项值的近似值，并允许细粒度的控制，但往往没有必要为了积分和微分将整个 T_s 划分成更小的周期。如果我们有一个相对较快的控制回路，并假设在一个单位的时间间隔内，方程 9-10 就可以简化为方程 9-11。 344

$$u(t) = K_p e(t) + K_i \sum_{t=0}^{t} e(t) + K_d \frac{e(t) - e(t-1)}{t - (t-1)} \qquad \text{（方程 9-11）}$$

下面的代码实现了方程 9-11 所表达的 PID 控制器：

```
class PID:
    """ 简单 PID 控制。

        这个类实现了一个简单的 PID 控制算法。实例化时所有的增益都是零。所
        以，GenOut 只会输出零。
    """
    def __init__(self):
        # 初始化增益
        self.Kp = 0
        self.Kd = 0
        self.Ki = 0
```

```
        self.Initialize()

    def SetKp(self, invar):
        """ 设置比例增益 """
        self.Kp = invar

    def SetKi(self, invar):
        """ 设置积分增益 """
        self.Ki = invar

    def SetKd(self, invar):
        """ 设置微分增益 """
        self.Kd = invar

    def SetPrevErr(self, preverr):
        """ 设置上一个偏差值 """
        self.prev_err = preverr

    def Initialize(self):
        # 设置时间变量
        self.currtm = time.time()
        self.prevtm = self.currtm

        self.prev_err = 0

        # 各项结果
        self.Cp = 0
        self.Ci = 0
        self.Cd = 0

345     def GenOut(self, error):
        """ 进行 PID 运算并且根据时间的增量 (dt) 和偏差值 (参数 error) 生成控
            制变量。
        """
        self.currtm = time.time()                    # 获取时间
        dt = self.currtm - self.prevtm               # 计算时间增量 t
        de = error - self.prev_err                   # 计算偏差的增量

        self.Cp = self.Kp * error                    # 比例项
        self.Ci += error * dt                        # 积分项

        self.Cd = 0
        if dt > 0:                                   # 不要除零
            self.Cd = de/dt                          # 微分项

        self.prevtm = self.currtm                    # 为下一步保存时间 t
        self.prev_err = error                        # 保存偏差值 t-1
```

```
        # 对各项求和，返回结果
        return self.Cp + (self.Ki * self.Ci) + (self.Kd * self.Cd)
```

通过整定 K_p、K_i 和 K_d 来调整 PID 控制器被当做是一种魔术（black art）。有几种方法可以整定 PID 参数，它包括 Ziegler-Nichols 法、基于软件的自动整定和试错法。

下面是一些在调整 PID 参数时常用的规律：

- K_p 和上升时间有关。增大 K_p 会减小上升时间，同时可能引起更多的超调和更长的调节时间。减小 K_p 可以增大上升时间，同时可以使超调减小（或者消失）。K_p 项本身不能消除稳态偏差。
- K_i 项可以消除稳态偏差。但是，如果 K_i 过大，就会引起超调并且增大调节时间。K_i 和 K_p 平衡时才能获得较短的上升时间和较小的超调量。
- K_d 项可以减小超调和调节时间。但是过大的 K_d 会引起系统不稳定和振荡，适当的 K_d 值可以改善系统的稳定性。

若想用上面所示的 PID 控制器，你需要先创建一个控制器实例，然后设置 K_p、K_i 和 K_d 参数[译注11]：

```
pid = PID()
pid.SetKp(Kp)
pid.SetKi(Ki)
pid.SetKd(Kd)
```

可以用一个循环来读取反馈值，调用 PID 控制器的 GenOut() 方法，并且把控制值传给受控设备：

```
fb = 0
outv = 0

PID_loop = True

while PID_loop:
    err = sp - fb     # 假设 sp 在其他地方设置了

    outv = pid.GenOut(err)
    AnalogOut(outv)

    time.sleep(.05)

    fb = AnalogIn(fb_input)
```

346

译注 11：K_d、K_p 或 K_i 过大都有可能引起系统不稳定，不过这需要具体问题具体分析。

注意，设定值 sp 必须在循环开始之前就设置好，或者，你也可以在大的控制循环中调用 PID 控制器，这样就可以在运行时改变 sp 了：

```
def GetPID():
    global fb

    err = sp - fb

    outv = pid.GenOut(err)
    AnalogOut(outv)

    time.sleep(.05)

    fb = AnalogIn(fb_input)
```

系统循环还可以做其他事情，比如更新用户界面、处理得到的数据、错误检测等：

```
while sys_active:
    # 循环体
    UpdateUI()    # 从用户那里得到设定值 sp
    GetPID()
    # 其他事情
```

如果 sp 是一个全局变量，那么 GUI 可以让用户去修改它。GetPID() 就可以根据它生成新的控制值了。

小结

本章只是对控制系统理论和应用的一个轻量级的简介，它并未面面俱到，我们只是浏览了控制论和控制应用的大概。这个学科是由无数工程师和研究员花了很长时间建立起来的。

347 我们希望你现在对能够构建什么样的控制系统、它们怎样工作、怎样为了你的应用选型有一个大体的概念。

推荐阅读

如果你想更深入地学习控制系统及其应用，这里推荐以下书籍：

Advanced PID Control . Karl Åström and Tore Hägglund, ISA—The Instrumentation, Systems, and Automation Society, 2005.

如果你想更深入地了解控制论，特别是 PID 控制，这本书可以是一个很好的开始。它几乎囊括了 PID 控制器的所有方面，并且有真实的例子和 PID 控制技术相关的数

学基础。它还包含讨论模型处理、控制设计和预测控制的章节。它对于每一个工作在控制领域的人都有参考价值。

Introduction to Control System Technology, 7th ed. Robert Bateson, Prentice Hall, 2001.

对于想学习控制系统和应用的人来说，在此推荐这本入门书籍。其中的数学知识不会超出第一年微积分的水平，而且作者也列举了大量实例来帮助理解各种概念，它有一章专门介绍基础电工学。每一章都会以词汇表的形式列出所有的定义。

Computer-Controlled Systems, 3rd ed. Karl Åström and Bjorn Wittenmark, Prentice Hall, 1996.

尽管这本书写得很好，并且有 Matlab® 和 Simulink® 的例子，但我们不会推荐它作为控制系统的入门书籍。但是一旦你开始做，需要针对特定的问题有所领悟的话，它是一个不错的参考。如果你经常做控制系统（或者计划去做），我们觉得这本书是一个随身的参考书或者更深入学习的资料。

Real Time Programming: Neglected Topics, 4th ed. Caxton C. Foster, Addison-Wesley, 1982.

Foster 的书是一本简洁明了的关于实时数据采集和控制系统中各种主题的小书。作者的写作风格轻松风趣，书中只用了最少的数学，在几个短小的章节中讲解了基本的控制论。它还涉及基本的数字滤波器、信号处理和通信技术等。它的出版时间很早了，但是或许还能找到。

对于控制系统，网上还有很多出色的资源。甚至还有一些免费软件。你可以从下面的资源开始：

348

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.1850&rep=rep1&type=pdf

PDF 版的 *Feedback Control Theory* 由 John Doyle、Bruce Francis 和 Allen Tannenbaum 编写（Macmillan，1990）。你也可以购买纸质版的形式。它以经典控制论和传递函数分析为导向，介绍了一些先进的概念。它特别强调鲁棒性。

http://aer.ual.es/modelling/

花一些时间阅读上面的书之后，你可以看看这个。它是西班牙 Almeria 大学做的一些交互学习资料的主页。那些交互性的程序对于动态地展示一些概念是非常有用的。其中的 PID 控制程序部分能与 *Advanced PID Control* 一书配套使用。注意，瑞士公司 Calerga Sarl 的商业网站（*http://www.calerga.com/contrib/index.html*）中也有一些学习软件。

http://www.cds.caltech.edu/~murray/amwiki/index.php/Main_Page

Karl Åström's 和 Richard Murray 的维基。这里有 *Feedback Systems: An Introduction for Scientists and Engineers*（Princeton University Press）的全文以及一些例子和练习。

它是反馈控制系统的一个很好的介绍，并且并不回避数学。如果你对微积分记得不是很清楚，你还可以从这里学到很多东西。

http://www.me.cmu.edu/ctms/controls/ctms/pid/pid.htm

用 Matlab 和 Simulink 写的控制系统教程集。

当然，维基百科也有相当多的条目是与控制系统相关的。

第 10 章 构建并使用仿真器

是你玩的某种形式的游戏吗?

—C. A. Chung, *Simulation Modeling Handbook: A practical Approach*

到目前为止，在本书中，我们已经讲解了 Python 语言编程基础，复习了电子学的一些要点，并探索了控制系统理论中的冰山一角。我们确实讲了很多内容，但是我们主要的挑战实际上是将计算机连接到仪器或控制系统中，在进入该领域之前，我们先介绍一个重要的主题：仿真。

在工程上，仿真可以用在很多事物上，从简单设备到一个完整的复杂系统。电子工程里，在 IC 和其他器件焊接之前，电路仿真可以用于研究和分析模拟与数字线路设计。系统工程师创建复杂的工业仿真系统，在铺设管道和安装传送带之前，就可以评估各种控制策略和工作流模型。军队和航空公司利用逼真的飞行仿真器来训练飞行员，利用它，飞行员们可以学习操作流程和技术，并进行练习。这样就可以避免飞机和飞机里的人（还有地面上的人）可能遇到的危险。

本章的主要目标是为你配备可扩展的仿真工具集，以便可以在后面其他项目中重复使用，同时让你理解，仿真在什么时候和什么场合是有用的，什么场合它不适用。为此，我们将测试几个完全用 Python 编写的完整的模拟器。最后，我们将看几种方法，它们通过修改常用的（同时也是免费的）软件工具来创建其他仿真器。

在第一个例子中，我们将考虑仿真一个通用的多功能设备，它拥有模拟 I/O 和开关量 I/O。第二个仿真器例子是一个 8 通道的交流电源控制器。虽然本章介绍的仿真器会涉及数据输入 / 输出、数据采集和用户接口，但是我们会将这些主题放到以后的章节再做深入讨论。第 11 章将会介绍数据 I/O 的细节，第 12 章将介绍使用文件来加载和保存数据的一些方法，在第 13 章中，我们将探究用户接口的更多细节，其中包括 TkInter 和

 wxPython 这两种图形用户接口。在学习到这些内容时，我们希望你会主动回头研究本章讲的这些仿真器，并用后面获得的知识来扩展它们的功能。

什么是仿真

如果你玩过视频游戏，那么你就用过仿真器。最早取得商业成功的一个视频游戏 Atari's *Pong*（雅达利的乒乓球），就是一个乒乓球游戏的仿真（尽管用今天的标准来看它很简陋）。事实上，所有的视频游戏都是在仿真某些东西，即使它们仿真的东西有可能并不存在于现实世界中，但它们也仍然是仿真。同样的道理，一个仿真的控制系统在现实中并不存在，它允许我们尝试不同的新奇想法，虚拟最坏情况的场景来评估系统反应，并探索各种行为模式，而这所有的一切都不会有硬件损伤或者人身安全的风险。

仿真的关键概念是所有的仿真器都是基于几种模式中的一种。模式可以很简单，也可以很复杂。模式可以是基于事件或时间序列的（比如大型机场用的自动行李处理设备），基于纯粹数学的（比如透镜和反射镜的光学性能），或者基于两者和其他一些因素的组合。作为理解仿真核心模式的一种方式，你可以将其想象成一个动态虚拟系统。举个例子，假如你有一个高精度的仿真器，它专门用于一些类型的化学品处理系统。本质上，在它的仿真软件中拥有一个虚拟化学加工系统，只要仿真的精度允许，它将展现尽可能多的反应和实际物品的特性。一个实际的高精度仿真甚至可以生产模拟的化学产品。

图 10-1 显示了一个仿真器如何与它在现实世界中的对应物一致。仪表测量系统（典型的如本书中开发和测试的）使用一个仿真接口来和模型交互。该模型允许我们观察和分析测量软件在连接到一个系统（在该情况下是一个虚拟系统）时如何工作。我们可以实现一个拥有基本性能的简略仿真模型，也可以创建一些拥有高精度的仿真模型，还可以放入模拟缺陷到模型里，来检验仪表测量系统的反应。

在实现数据采集和控制系统时，仿真一个连接到控制 PC 的设备，可用来加速开发过程，并提供一个安全的环境来彻底测试你的设想。仿真有时候是无价的，没有它就无法完成一些任务，它可以让你深刻洞察测量程序和被仿真的设备或者系统。进行仿真不管是因为测试硬件还不可用，还是因为目标系统硬件太贵重不能冒被损坏的风险，它都是一个好的方法，因为它让软件可以运行和测试，从而获得一个高度的自信，确认它将会在现实世界中正确工作。

351

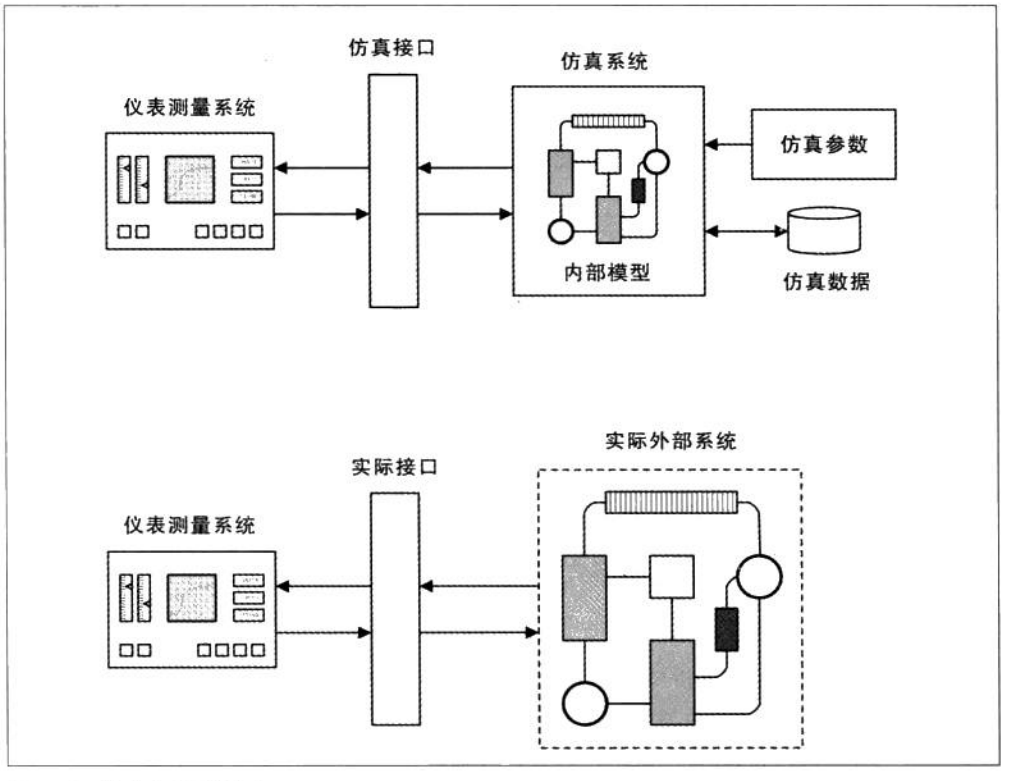


图10-1：仿真与现实世界

低保真和高保真

当我们谈论仿真时，首先要考虑的是保真度问题。仿真的逼真度决定了它能多精确地模拟真实系统。每提高一级保真度，花费在实现该仿真上的金钱和精力都会大幅增加，所以，你必须决定到什么程度就足够好，并不断抗拒将其逼真度提高的诱惑。

在第一次尝试写一个仿真器时，一个常见的错误是把所有的东西都扔进去。即使是一个有权使用来自真实系统的大量真实数据的经验丰富的专家，通常也不会这么做。在开始，有太多的未知数。微妙的行为交互有可能令人惊讶，也可能是永远不会在实际系统中看到的，一个系统如果设定太多不透明的部分，就会包含各种陷阱。

这就是仿真都是递增地创建起来的原因。首先，定义软件接口或者 API。一个普通的接口可能使用和 API 函式不一样的名字，但是它接收以及返回的数据应该尽可能和实际 API 使用的数据相似。换句话说，仿真应该在接口间传递数据，并支持你创建的控制系统或者仪器系统的基础算法功能。一旦接口仿真达到可以支持初期的测试和开发，就可

352

以按必要的值来提高仿真的保真度。

不得不说，有时仅仅由一个API组成的低保真仿真器也可以工作得很好。例如，如果你所需要的只是检查一个线性控制功能是否正常工作，那么一个用来驱动仿真的简单数据源与一些捕获并保存输出的方法可能就是所有必需的东西。

模拟错误和故障

除了模拟一个工作系统的功能外，仿真还需要有能力模拟错误。换句话说，它必须能够模拟在某些预设情况下被中断。

如果你做过故障分析，就像在第8章讲过的那样，应该会清楚可能发生哪些类型的错误以及这些错误如何使系统失效。模拟这些错误的能力允许你去弄明白你的软件该如何处理它们。

假如你想将故障注入能力包含到仿真中，初步的故障分析是基本的第一步。

一般来说，有两种主要级别的故障（不包括仪器程序代码本身所隐含的bug）发挥作用：接口故障和系统故障。这两种故障之间的分界线有时不那么明显，不过在一个仿真环境中，还是有办法把它们区分开来的。这样，按顺序可以很容易明确区分因和果，从而避免处理真实的物理部件带来的复杂和混乱。

接口故障

接口故障导致一个错误，顾名思义，这种错误出现在正在开发或者测试的系统和它背后的系统仿真之间的API层。该类型的故障可能表现为通信错误，比如，数据损坏或者无响应。换句话说，接口故障发生在设备软件和仿真系统之间。在真实的系统中，接口错误可能是由断线、被侵蚀的连接、有缺陷的电子器件或者假性噪声等导致的。

模拟接口故障一般包括使通信通道失效，或者往通道注入随机数据这样的方法。你也可以模拟一个总线接口硬件里（如可插拔的I/O卡）的故障，不过这种属于我们早前提到过的灰色地带，可能当做系统故障来处理更好一些。

353

图10-2展示了一个拥有基本的故障注入能力的接口仿真。注意，TxD（发送）从仪器软件到仪器仿真器的输出没有停用的功能。这不是一个疏忽，因为如果仪器采用一个命令响应类型的控制界面停用TxD和测试设备软件之间的连接，是没有太多意义的。仪器（或者模拟的仪器）只是停下等待一个命令，它没有任何方法来侦测连接是否断掉。

从仪器到仪器软件之间的路径是另外一回事。在这个过程中，当接收不到仪器的响应时，仪器软件应该能够侦测到，然后执行一个预定义的策略。

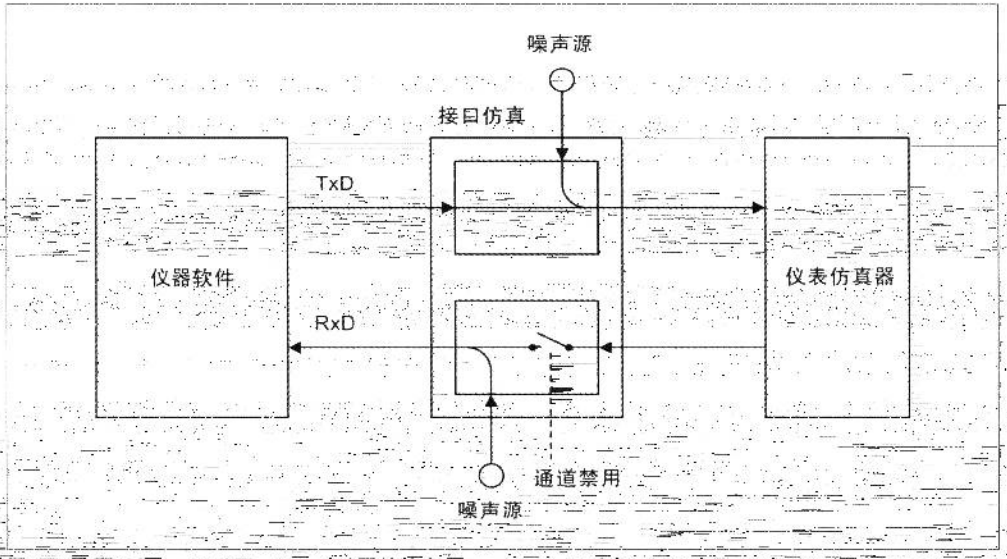


图10-2：接口故障注入

在这里，一个合理的疑问会是：为什么不模拟在仪器仿真器里停用通信，而使用接口仿真？理由是仪器或者控制系统可能会与系统中其他的一些部件交互，这些部件即使在仪器的主要连接不可用的情况下，仪器软件依然可以检测到。换句话说，“聪明”的设备软件有时可以从其他来源获得确认（例如，感知电源状态的改变）表明一个正确的命令动作发生，以此判定接口是导致问题的可能原因。

图10-2还表明了在TxD和RxD（接收）通道上都有噪声注入能力。在通信信道处理的是ASCII字符流的情况下，可以通过注入随机字符来模拟线路噪声。任何人只要曾经使用过调制解调器来与远程计算机通信，就很可能在调制解调器连接到远程主机或者从远程主机断开时，发现过这种线路噪声的例子。 354

能够处理损坏数据的仪器软件很可能拥有重复一个操作的能力，或者至少可以发送一个查询，来判定一个命令是否被正确接收。它甚至可能通过检查来自仪器的响应来判断哪里发生了噪声。如果设备抱怨它收到一个无效的命令，那么噪声发生在TxD通道。如果来自仪器的响应是无用信息，那么噪声很可能发生在RxD通道。

系统故障

系统故障有时是由系统内部引起的，有时是由它的子系统（依赖于系统的复杂程度）引起的。模拟各种系统故障的能力提高了仿真的价值，允许你测试和验证仪器软件故障响应。

系统故障可以有许多种情况，不过在仿真器里归结为表格中的一个值或者函式的一个返回码。换句话说，在正常情况下，如果仿真器对特定输入的响应是返回一个期望的数值，那么模拟一个故障简单到只需要包含返回一个无效数值或者一个错误提示的逻辑。

一种方法是直接在仿真器的函式或者方法返回语句前注入故障，如图 10-3 所示。

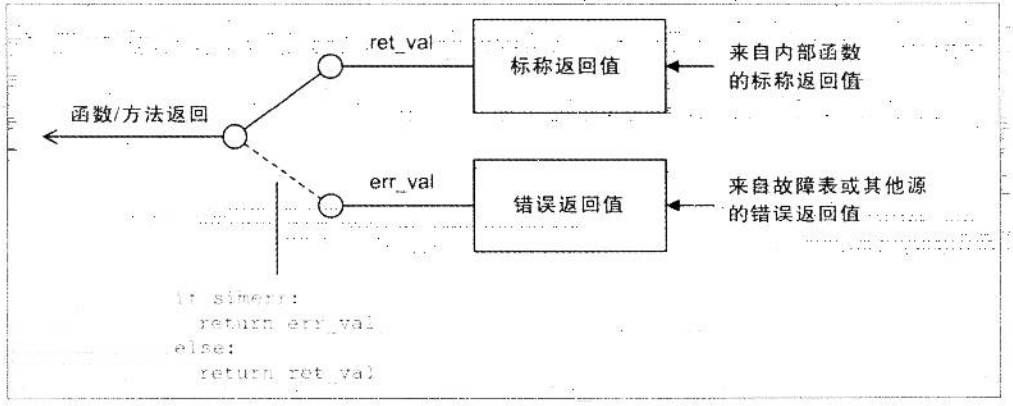


图10-3：系统级故障注入

假定图 10-3 中的 ret_val 保存函式或者方法正常要返回的标称值，设置 simerr 值为 True 将导致错误返回值(err_val)被回传给调用者。错误值通过一个存取器方法来设置，
355 可以从一个错误表中加载，也可以在某个用户界面中手工输入。

另一种方法是使用一个组件的全局变量或者一个对象变量来创建一个故障状态，如图 10-4 所示。

在图 10-4 中，SimObject 类的定义经过设计明显暴露了在类的方法中使用的关键变量。在一个特别简单的场景里，这些将是只读变量，在类实例化时被赋值一次，后面不会被类的方法改变值。实现加锁能力来阻止方法修改共享的对象变量是可行的，但是那样做通常要增加复杂性，有时也没有必要。

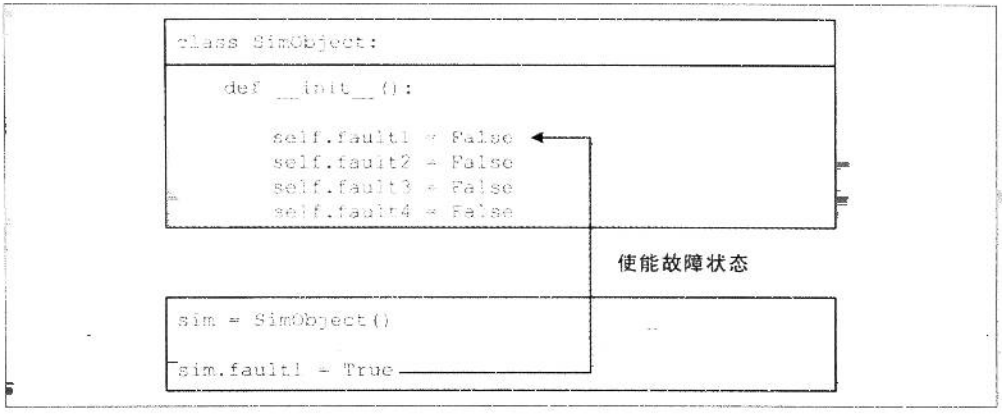


图10-4：使用对象变量模拟故障注入

如果你对单元测试技术很有经验，可能会觉得所有的这些听起来很熟悉，这种感觉可能是对的（我们在第 8 章也做过相关讨论）。在单元测试里，一个主要的目标是执行程序所有可能的路径，来达到完全的代码覆盖。如果被测试单元只有在错误发生的情况下才可能被执行，那么单元测试环境必须能够设置变量或者指定输入来模拟适当的错误响应。一个拥有故障注入能力的仿真会做几乎相同的事情，虽然是因为不同的原因。

被作为一个测试环境时，一个仿真器可以被用来创建故障场景，这些场景若不使用仿真器，将很难（甚至不可能）在真实硬件上再现，至少不冒严重的（以及代价昂贵的）物理损坏的风险的话不可能实现。拥有故障注入能力的仿真允许你观察自己创建的控制系统或者数据采集系统的行为，观察它如何响应各种出错状态，如果产品需求定义了错误响应，你还可以判别它是否与这些需求相符。

使用 Python 创建一个仿真器

356

在本节，我们将查看两个完整的用 Python 语言编写的仿真器：DevSim 和简单的电源控制器（SPC）。随后，我们将看到一些其他的使用免费开源软件实现仿真的方法。这些仿真器的源代码放在本书的资源库里，你可以从本书的主页去访问它。虽然这些是功能完整的仿真器，但它们也仅仅是例子，而不是产品级的工具。我们希望你可以使用它们来启发你的灵感，或者作为起点，去创建自己的仿真工具满足你特有的需求。

Python 十分适合用来创建仿真器。它易用，灵活性高，允许你很容易地实现一些东西（例如，插件模块）。用它来开发仿真器，常常可以很快地开始并完成，而且一旦完成，在需要的时候，它们也很容易扩展和改进。另外，当安装了 SciPy 或者 NumPy 扩展库时，Python 能够进行一些令人印象深刻的数学运算，我们马上还会看到，产生图形化的输出

也并不困难。

另一方面，用 Python 编写的代码没有比用 C 或者 C++ 编写的代码运行快。C 或者 C++ 是自然语言，同时也是 Python 的底层解释器。如果需要高速的数据生成或者快速响应，你应考虑其他办法。幸运的是，大多数仪器程序有相当长的固定时间来开始，所以速度通常不是问题。

程序包和模块的组织

我们将在本节查看的仿真器是 DevSim 和 SPC。它们每一个都可以放在自己的包（子目录）里。我们就是这样安排它们的，其中 SPC 放在 ACSim 目录里。DevSim 导入 FileUtils 模块，该模块将在第 12 章详细介绍，它用来读写 ASCII 数据文件。另外，它还导入了一个叫做 RetCodes 的模块，该模块包含一组伪常数，用来返回码值。它被特意设为只读的共享文件。典型的，FleUtils 和 RetCodes 模块，以及其他你所有的仿真器和实用工具需要共享的模块，应该放置在单独命名为 SimLib 的包里，如图 10-5 所示。

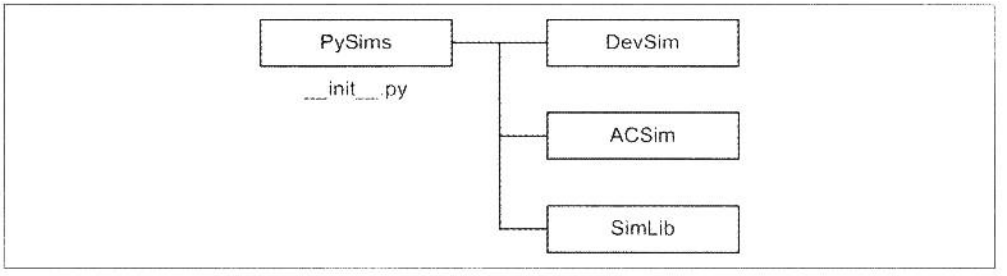


图10-5：仿真器包结构

357 如果你还没有做，现在是个合适的时间暂停一下，去从本书的主页上下载源代码。在那里，你能找到比这里看到的更多的东西，并且已经被整理得很整洁，还附带了安装说明书。

请注意顶层的包 PySims 有一个名为 _init_.py 的文件。这个文件确立了 PySims 作为包层级的“锚点”，它可以包含文档字符串、包的初始化代码，或者什么都没有。后面研究 *gnuplot* 时，会看到开发者使用 _init_.py 来处理包初始化的推荐方法。

数据输入 / 输出仿真器

首先要看的是一个相当重要的仿真器，它被用来替代一个基于总线的多功能 I/O 卡。它可以直接与第 9 章看到的 PID 控制代码一起使用（我们会在稍后向你展示如何做到）。它也有能力将用户定义功能整合进去，成为响应数据处理的一部分，这对模拟机械的惰

性或者使用滤波函式这样的事项很有用。

DevSim 内部构件

尽管 DevSim 的 API 看起来可能令人生畏，实际上，它主要的只是一些存取器方法，用来设置仿真参数，以及返回当前的值。它的基本形式里，实际上没有模拟任何特定的外部设备或系统。那些功能是你自己添加进去的，以用户定义函式或者附加软件的形式来满足项目特定的需求。

我们不会将所有的代码列在这里（大约有 1000 行代码），而是描述它的内部架构，列出各种可用的方法，高亮显示其中一些，并讨论仿真器如何使用。如果可以，你应该将源代码放在手边作为参考。

从内部来看，DevSim 是目前我们看到过的最复杂的东西，但是它的主要功能只是数据路由选择。图 10-6 是一个针对 DevSim 的 IC 数据表风格的逻辑框图。

当数据从 DevSim 通过时被缓冲，每个活动都被同步，以便以锁步样式出现。DevSim 也是多线程的：4 个线程处理循环功能（波形产生）和 4 个管理文件输入，第 9 个线程在主循环里处理仿真器基本的排序。

DevSim 类的 _init_() 方法预先设置了内部参数的默认值，然后调用内部方法 _run() 作为它最后的行为。_run() 的代码如下： 358

```
#--------------------------------------------------------------------
# Simulator launch
#--------------------------------------------------------------------
def __run(self):
    """ 仿真开始。

        实例化的主循环线程、周期及文件 I/O 线程，然后等待开始标志变为 True，当启动标志为
        True 时，主循环线程启动。
    """
    # 输入线程对象列表                                          359
    cycThread = [None, None, None, None]
    fileThread = [None, None, None, None]

    simloop = threading.Thread(target=self.__simLoop)

    # 创建 4 个周期和文件输入线程句柄
    for inch in range(0,4):
        cycThread[inch] = \
            threading.Thread(target=self.__cyclic, args=[inch])
        fileThread[inch] = \
            threading.Thread(target=self.__fileData, args=[inch])
    # 启动刚创建的循环和文件 I/O 线程
```

```
        cycThread[inch].start()
        fileThread[inch].start()

    # 主线程等待启动信号
    wait_start = True
    while wait_start:
        if self.startSim == True:
            wait_start = False
        else:
            time.sleep(0.1) # 等待 100ms
    simloop.start() # 启动
```

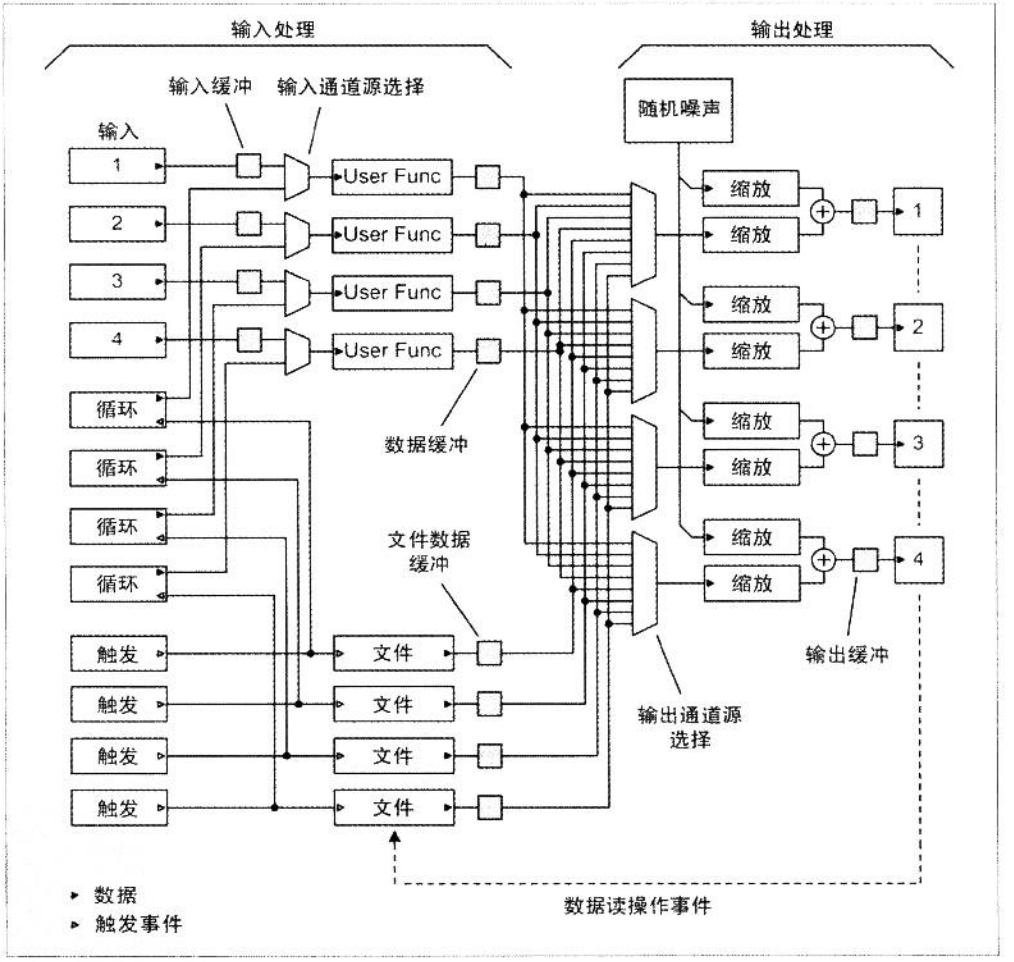


图10-6：DevSim内部逻辑

首先，主线程 simloop() 被创建，接着，8 个输入线程在一组列表中被实例化。因为每个线程都被创建，所以它们依次开始运行。然后 _run() 方法等待一个开始信号，最后启动 simloop() 主线程。

该仿真器设计为当一个布尔变量（startSim）被设置为真（True）时开始运行。你可以在仿真器被使能前设置各种参数，也可以在它开始运行后动态地设置。通过设置布尔类型的对象变量 stopSim 为 True，可以停止仿真器的运行。

在 4 个循环线程和 4 个文件输入线程中，每个都在“事件”发生时加载数据到缓存中。就 DevSim 来说，一个事件是一个简单的标志变量，通过设置和复位它来控制活动；从底层消息机制来说，它不是那种在一个 GUI 中随处可见的真的事件。图 10-7 展示了线程是如何获取输入数据的，然后把它放入缓存中用于最后的选路输出。注意，_cyclic() 线程每次可以从 5 个可能的输入中选择一个：DC（恒定值）、正弦波、脉冲（方波）、斜波以及锯齿波。循环数据以固定的速率被更新，这个速率由参数访问器 setCyclicRate() 来设置，每一个循环的输入可以拥有一个唯一的速率值。循环输入对产生一个固定的类似 DC 那样的固定值，或者能够被连接到仿真器的仪器软件读取和处理的周期性波形很有用。

使用循环输入线程允许它们运行在和仿真器的主线程不同的速率上，这种轮转允许它们 360
以特定的频率创建波形数据。如果不使用线程来实现，模拟的波形将按照仿真器主循环的速率产生变化，这在很多情形下是不合适的。允许循环数据生成器异步运行是一个时序解耦的范例。若要在异步数据产生时捕获它，缓存是必要的。

361

多线程执行

现代的操作系统都支持多进程，通过在进程间快速地切换 CPU，在同一时间运行或多或少的进程。一个进程（在这个上下文里）指的是一个程序和它的内存空间，它的系统环境变量和在创建时分配给它的各种可能的 I/O 资源。这些组成了进程的运行环境，对于进程可能有很多过时的观念，需要转变。

另一方面，线程有时也被称为轻量级进程，因为线程运行在父进程的上下文里。换句话说，进程内的所有线程分享父进程的内存空间和运行环境。可以把这种类型的线程想象成能够独立运行的函式。因为它们运行在同一个共享的环境中，所以一个线程有可能会干扰同一父进程中的另一个线程所使用的全局变量。这种类型的数据耦合可能导致烦人的数据损坏和死锁状态，所以，对什么时候以及如何使用线程要仔细考虑。

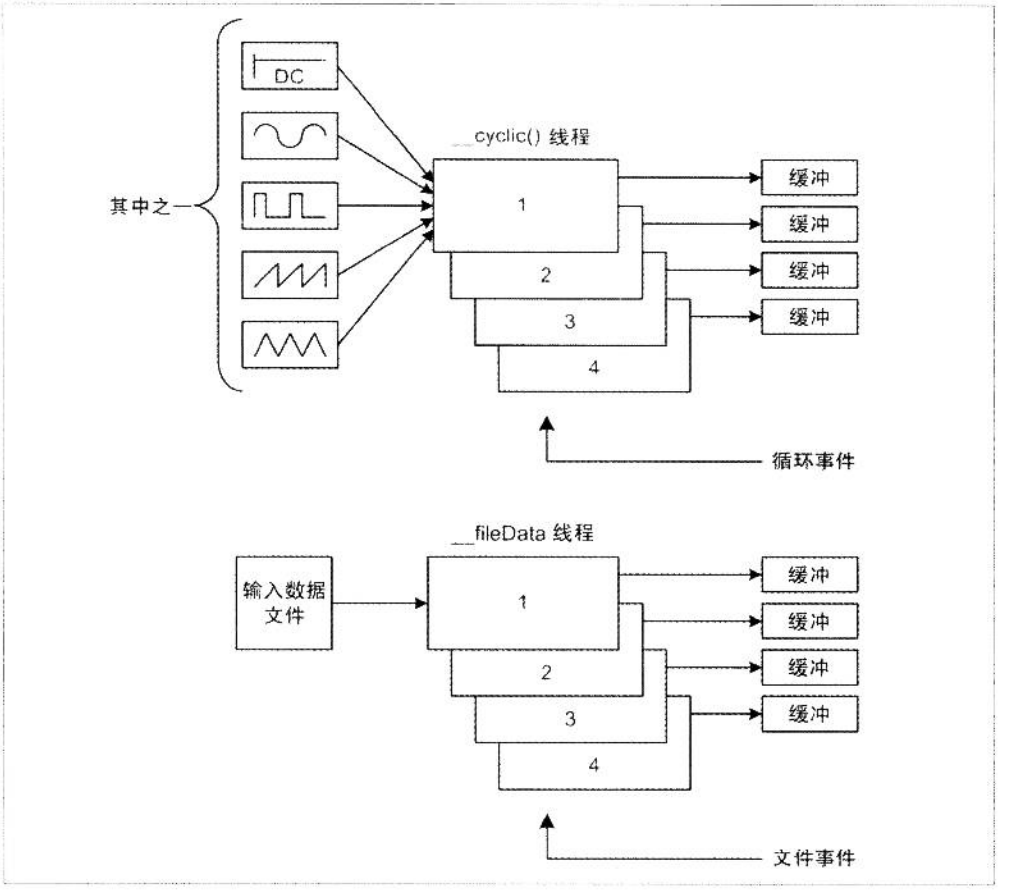


图10-7：循环线程和文件输入线程

线程 _fileData() 用来从文件中获取数据，然后将它们传送到实验阶段的仪器软件。该特性允许你反复、轮流获取一组输入数据。这对评估测试状态下的仪器软件很有用，可以观察它在一组已知的输入情况下的行为。

主循环线程代码如下，其中所有的数据被收集起来，并选路发送到输出缓存：

```
def __simLoop(self):
    """ 仿真主循环。

        在主循环中不断检查每个输入通道的输入数据（无论是来自外部呼叫还是来自周期事件源），
        如果数据可用，则启动处理链，这将最终导致数据出现在输出缓冲器中。

        作为一个线程的主循环，其将永远运行下去。所有的外部数据将被缓冲，所有的输出数据也
        被缓冲。在周期模式下，输出缓冲器中的新数据可用时，将覆盖原先的数据。
```

```
        """
        while self.stopSim != True:
            # 扫描所有的4个通道，获取有用的数据
            for ichan in range(0, 4):
                if self.in_src[ichan] == DS.EXT_IN:
                    indata = self.inbuffer[ichan]
                else:
                    indata = __getCycData(ichan)

                # 如果用户函式有定义，则将数据应用于其中
                self.databuffer[ichan] = self.__doUserFunc(ichan, indata)

                # 获取文件数据（如果已经配置好）
                self.__getFileData(ichan)

                # 逐一通过输出通道，移动数据
                for ochan in range(0,4):
                    if (self.out_src[ochan] >= DS.INCHAN1) and \
                        (self.out_src[ochan] <= DS.INCHAN4):
                        outdata = self.databuffer[ochan]
                    else:
                        outdata = self.filebuffer[ochan]

                    randdata = self.__getRandom()

                    oscaled = self.__scaleData(outdata, self.outscale[ochan])
                    rscaled = self.__scaleData(randdata, self.randscale[ochan])

                    self.outbuffer[ochan] = oscaled + rscaled
                    self.outavail[ochan] = True

                time.sleep(self.simtime)
```

362

通读 _simloop() 方法的代码，就相当于跟着图 10-6 所示的数据流从左到右。第一步是获取外部提供的（通过 sendData() 方法，该方法将在下一节详述）或者从某个循环数据源来的数据。接下来，数据被传送到用户定义的函式（如果有的话）。然后数据被写入缓存（databuffer）。

假如数据输入是一个文件，数据被从文件中读出来并保存在另一个缓冲区。要注意，一个方法被调用并不意味着事情在实际发生。一个输入方法可能是不活动的，从而在内部被跳过，这依赖于仿真器是如何配置的。

一旦输入数据存在于两个输入缓冲中的其中一个，它就被读出并传送到输出。在传送过程中获得随机数据（为了模拟噪声），然后数据被随意缩放，再和缩放后的随机数据叠加。最后，这些数据到达终点，即每个通道的输出缓冲区。

DevSim 方法

DevSim 类包含了 26 个公共方法和 15 个内部方法。大多数公共方法被用于设置或者检索内部参数的值。这些公共方法将在下面的列表中描述。

实际使用时，参数 inchan 和 outchan 可以是从 0 到 3 中的任何一个通道数。

363

参数访问器方法如下：

getCyclicLevel(*inchan*):

返回一个二元组，正常则返回 NO_ERR 和特定通道的当前循环源的值，假如 inchan 超出范围，则返回 BAD_PARAM 和 None。

setCyclicLevel(*inchan*,*level*):

为指定输入通道，对处于 CYCNONE 模式的循环源设置输出值。如果成功，则返回 NO_ERR，如果 inchan 超出范围，则返回 BAD_PARM。

getCyclicOffset():

返回当前的循环偏移值。

setCyclicOffset(*offset*):

为所有的循环数据设置当前的循环偏移量。偏移量是峰到峰的范围内的输出相对于零的转换值，或者被称为偏离率。

getCyclicRate(*inchan*):

返回一个二元组，或者是 NO_ERR 和指定通道当前的循环速率，或者是 BAD_PARAM 和 None。

setCyclicRate(*inchan*,*rate*):

参数 rate 定义了指定输入通道的循环速率，以 1/10 秒为单位。注意，这是循环数据的周期，不是频率。频率是 rate 的倒数。如果成功，则返回 NO_ERR，如果 *inchan* 超出范围，则返回 BAD_PARAM。

getCyclicType(*inchan*):

返回一个二元组，或者是 NO_ERR 和指定通道当前的循环波形，或者 BAD_PARAM 和 None。

setCyclicType(*inchan*, *cytype*):

定义循环数据源的输出波形。可能的数据波形有：正弦波、脉冲波和锯齿波。一个循环源也可以被设置为产生一个常量输出值，并且该值可以在模拟器运行的情况下随时改变。这实际上是模拟了一个可变电压的电源。表 10-1 列出了 5 种有效的循环数据类型。

表10-1：仿真器循环数据类型

循环类型	数值	说明
CYCNONE	0	常量输出
CYCSINE	1	正弦波
CYCPULSE	2	50%占空比的脉冲（也就是方波）
CYCRAMP	3	斜波
CYCSAW	4	上升/下降沿对称的锯齿波

调用成功，则返回 NO_ERR，如果 *cyctype* 是无效的或者 *inchan* 超出范围，则返回 BAD_PARAM。

setDataFile(*infile*, *path*, *filename*, *recycle=True*)：

364

定义并打开一个数据文件作为输入，然后由 *infile* 索引指定为数据源。参数 *path* 如果未指定（空字符串或者为None），那么默认路径被假定为当前工作目录。输入文件必须包含指定格式的数据，该格式必须是 ASCIIDataRead 类所支持的 4 种格式中的一种，这个类在 FileUtils 模块中定义。

如果参数 recycle 为 True，当遇到 EOF（文件结束符）时，文件指针将被重置到开始，并从头开始读，默认的行为是回收数据文件。如果一个数据源文件已经被打开，用于一个输出通道，这时候调用该方法将导致打开的文件被关闭，然后一个新的文件将被打开。

如果文件打开失败，则返回 OPEN_ERR，如果 *infile* 无效，返回 BAD_PARAM；其他情况则返回 NO_ERR。

getDataScale(*outchan*)：

返回一个二元组，或者是 NO_ERR 和指定通道的当前数据缩放比例，或者是 BAD_PARAM 和 None。

setDataScale(*outchan*, *scale*)：

设置输出通道的输入数据比例因子。每一个输出通道有一个可选的唯一乘法比例因子应用可用。成功时返回 NO_ERR，当 *inchan* 超出范围时，返回 BAD_PARAM。

getFunction(*inchan*)：

返回一个二元组，或者是 NO_ERR 和指定通道的当前函式字符串，或者是 BAD_PARAM 和 None。

setFunction(*inchan*, *funcstr*='')：

对指定输入数据通道的数据流使用用户提供的函式表达式进行处理，表达式可以使用两个预先定义的变量：

x0：

输入数据

x1:

上一个数据（1/Z）

函式是一个字符串。它可以引用 x0 和 x1 的值，但是不能包含等号（赋值）。函式的结果被作为输出通道的数据输入。传递 None 或者一个空字符串将不对数据进行操作。如果成功，则返回 NO_ERR，如果 *inchan* 超出范围，则返回 BAD_PARAM。

getInputSrc(*inchan*):

返回一个二元组，或者是 NO_ERR 和指定通道的输入源，或者是 BAD_PARAM 和 None。

setInputSrc(*inchan*,*source*):

为一个输入通道选择数据源。*inchan* 可以是从 0 到 3 中任意有效的输入通道数。*source* 参数可以是 EXT_IN（默认）或者 CYCLIC。输入源可以随时在运行中更换。运行成功则返回 NO_ERR，如果参数 *inchan* 超出范围或者 *source* 无效，则返回 BAD_PARAM。

365

getOutputDest(*outchan*):

返回一个二元组，或者是 NO_ERR 和当前输出通道的数据源，或者返回 BAD_PARAM 和 None。

setOutputDest(*outchan*,*source*):

为一个输出通道选择数据源。*Outchan* 可以是任何有效的输出通道数（数字 0 ~ 3）。*Source* 参数可以是 INCHAN1、INCHAN2、INCHAN3、INCHAN4、SRCFILE1、SRCFILE2、SRCFILE3 以及 SRCFILE4 中的一个。如果成功，则返回 NO_ERR，如果 *inchar* 超出范围或者 *source* 是无效值，则返回 BAD_PARAM。

getRandScale(*outchan*):

返回一个二元组，或者是 NO_ERR 和当前随机数据放大系数，或者是 BAD_PARAM 和 None。

setRandScale(*outchan*,*scale*):

设置输出通道随机数据比例因子。每一个输出通道都可以有一个任意的乘法比例因子应用于随机数据。如果比例被设置为 0，就没有随机数值被加到数据里。如果成功，则返回 NO_ERR，如果 *inchan* 超出范围，则返回 BAD_PARAM。

getSimeTime():

返回当前仿真器的循环时间。

setSimTime(*time*):

设置仿真的全部循环时间。这实际上是主循环要在每个循环迭代之间暂停的总时间。该时间的单位是 1/10 秒。没有返回值。

getTrigMode(*inchan*):

返回一个二元组，或者是 NO_ERR 和特定通道的当前触发模式，或者是 BAD_PARAM 和 None。

setTriggerMode(*inchan*,*mode*):

为指定的通道设置触发模式。触发模式可以是 NO_TRIG(0)、EXT_TRIG(1)、或者 INT_TRIG(2) 中的一个。

在 NO_TRIG 模式下，所有的循环源依照用 setCyclicClock() 方法设置的时钟频率连续不断地运行，数据源文件在一个输出通道被访问之前不会被读取。

在 EXT_TRIG 模式下，循环源执行一个单一操作，并且文件源在每次触发时都被读一次。在 INT_TRIG 模式下，循环源执行一次循环，并且数据源文件在每次输出通道被访问时读取一次。

如果成功，则返回 NO_ERR，如果 *inchan* 参数值超出范围或者 *mode* 参数是无效值，则返回 BAD_PARAM。

仿真器控制及 I/O 方法如下：

genTrigger(*inchan*):

366

产生一个触发事件。依赖于触发模式，一个触发事件将导致一个循环数据源的迭代，或导致从数据输入文件读出一条记录。调用成功则返回 NO_ERR，如果 *inchan* 超出范围，则返回 BAD_PARAM。

输出 / 输出方法如下：

readData(*outchan*,*block=True*,*timeout=1.0*):

从输出缓冲区里返回可用于指定输出通道的数据。如果阻塞特性被使能，该方法将阻塞，且不会马上返回到调用者，直到数据变为可用或者指定的超时周期用完。返回一个二元组，该二元组由返回码和来自输出通道的数据组成。如果成功，则返回 NO_ERR，如果 *outchan* 超出范围，则返回 BAD_PARAM。如果返回码是除了 NO_ERR 以外的其他任何值，则数据值将为 0。

sendData(*inchan*,*dativa*l):

将调用者提供的数据写入指定的通道。输入缓冲区中的数据在仿真器的每一个循环中被读取。如果成功，则返回 NO_ERR；如果 *incha* 超出范围，则返回 BAD_PARAM。

表面上看起来虽然很复杂，但该仿真器实际上可以归结为两种主要方法：readData() 和 sendData()。其他所有的方法都是为输入和输出之间将要发生的动作做准备的。

一些简单例子

接下来的代码示例演示了在仿真器中从一个输入到一个输出的数据流。它没有使用可选的用户函式，也没有对数据进行放大或者加入噪声。

```
#! /bin/python
# TestDevSim1.py
#
# 回应数据写入到模拟器的输出。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import  time
from    DevSim  import  DevSim
import  SimLib.RetCodes as RC
import  DevSim.DevSimDefs as DS


def testDevSim1():
    print "Init DevSim"
    simIO = DevSim.DevSim()

    # 设置仿真设备
    simIO.setInputSrc(DS.INCHAN1, DS.EXT_IN)
    simIO.setOutputDest(DS.OUTCHAN1, DS.INCHAN1)

    simIO.startSim = True
    time.sleep(1)

    loopcount = 0
    while loopcount < 10:
        simIO.sendData(DS.INCHAN1, (5.0 + loopcount))
        print simIO.readData(DS.OUTCHAN1)
        loopcount += 1

    simIO.stopSim = True    # 设置停止标志
    print "DevSim terminated"
```

367

参阅这段代码和图 10-6，第一个有意思的点是 setInputSrc() 和 setOutputSrc() 语句。在这两个方法的调用中，第一个参数确定是要使用输入通道还是输出通道，第二个参数指定数据从哪里来。第一个方法调用声明给输入通道 1 的数据将从外部输入通道 1 获得，这在方法 sendData() 被调用时发生。换句话说，它控制“输入通道源的选择”（见图 10-6 中两个小的梯形输入符号）。第二个方法调用（setOutputSrc()）决定出现在输出通道 1 上的数据要从哪里获取。在这种情况下，可选参数掌管八输入源选择器的行为，并指定输入通道 1。注意，输入通道 1 可以是一个来自外部的直接输入，也可以是内部循环数据源之一。如果跟随图 10-6 中的数据路径，我们看到它通过输入端的用户定义函式块，然后通过输出端的一个放大块和随机数据（噪声）求和点。这个例子没有使用这其中的任何特性，它实际上是输入和输出的一个直接连接。

第二个例子使用了一个数据文件作为仿真器的输入：

```
#! /bin/python
# TestDevSim2.py
#
# 从输入文件中读取数据，并将数据传递到输出。必须将包含10个ASCII格式的值的文件存放在当前目录。
#
# Source code from the book "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import  time
from    DevSim  import  DevSim
import  SimLib.RetCodes as RC
import  DevSim.DevSimDefs as DS


def testDevSim2():
    print "Init DevSim"
    simIO = DevSim.DevSim()

    # 定义欲用的源文件
    rc = simIO.setDataFile(DS.SRCFILE1, None, "indata1.dat")
    if rc != RC.NO_ERR:
        print "Error opening input data file"
        print "DevSim start aborted"
    else:
        # 设置仿真设备
        simIO.setOutputDest(DS.OUTCHAN1, DS.SRCFILE1)

        simIO.startSim = True
        time.sleep(1)

        loopcount = 0
        while loopcount < 20:
            # 读取并打印来自文件的数据
            rc, dval = simIO.readData(DS.OUTCHAN1, block=True)
            if rc == RC.NO_ERR:
                print "%2d - %4.3f" % ((loopcount + 1), float(dval))
            else:
                print "%2d - error: %d" % rc
            loopcount += 1

        # 完成。关闭所有的文件并退出
        simIO.stopSim = True
        print "DevSim terminated"
```

368

方法 setDataFile() 的调用定义了将和索引 SRCFILE1 相关联的数据文件。注意，setDataFile() 还打开了文件，如果文件打开失败，将返回一个错误代码 OPEN_ERR。在这个简单的例子中，我们没有检查错误码的值，只确定了它不是 NO_ERR。对

setOutputSrc()的调用将数据文件和OUTCHAN1关联起来。每当readData()方法被调用时，文件被访问，一个单条记录被读取。然后数据通过八输入源选择器（见图10-6）到达输出放大和随机数据求和段，最后出现在输出端。对readData()的调用实际上是对内部数据文件读操作的触发事件。

因为此处没有放大操作和随机数据“噪声”应用于来自输入文件的值，所以，基本上它就是一个打开并读取数据文件的方法。然而，它也是一个创建更复杂的仿真机构的有用部件，比如，图10-8所示的仿真。其中，DevSim用来从一个输入文件里读取数据，采用用户定义函式和放大处理数据，然后根据结果运行一些测试代码。这只是DevSim既用于提供增强的测试，又用于仿真功能的一种方法。

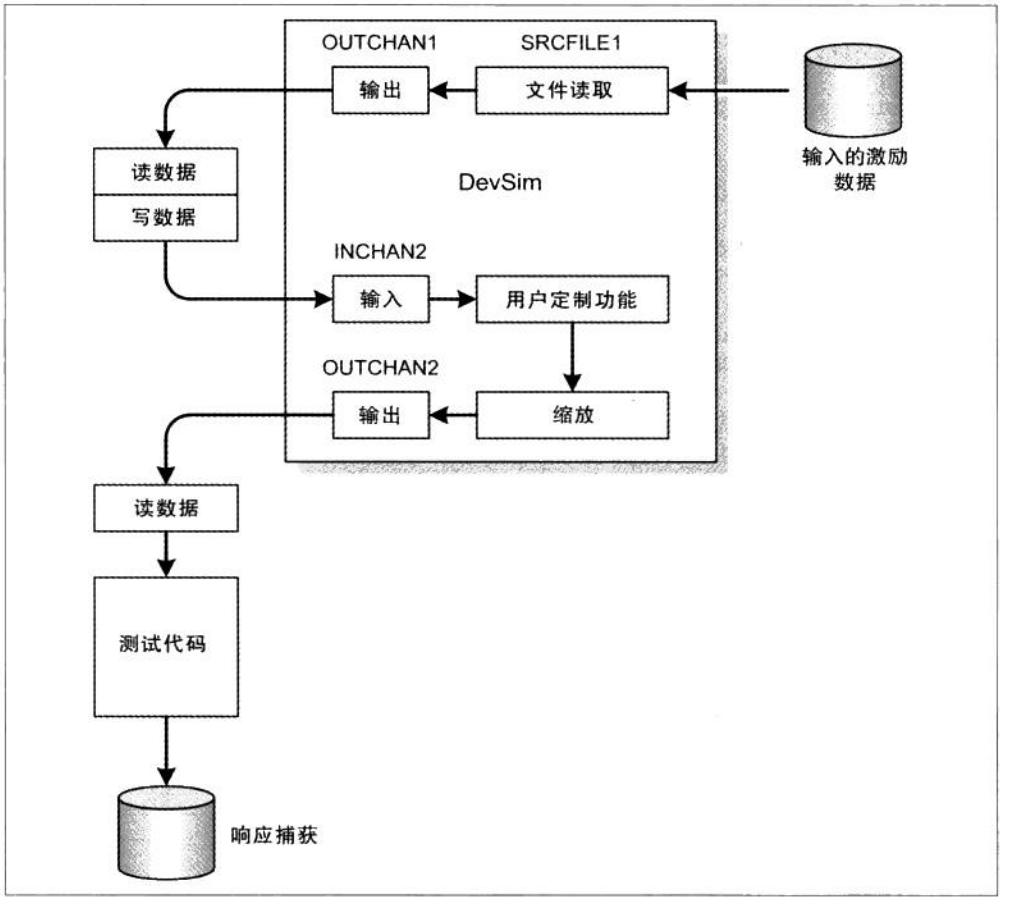


图10-8：DevSim使用示例

用户定义函式

DevSim 有能力在输入源选择器之后，往输入数据流里插入一个简单的用户定义函式。一个用户自定义函式是一个包含有效 Python 语句的字符串。它使用 Python 的内建方法 `eval()` 来做内部处理，并且提供了两个预定义的变量 `x0` 和 `x1`。`x0` 是当前输入数据值，`x1` 是上一个当前数据值(1/z 单位的延时)。实际上，使用这两个变量可以做相当多的事情。该语句求值的结果成为到八通道源选择器的输出。如果函式字符串是空的，那么就会被忽略。

例如，语句 `x0*2` 会将输入数据乘以 2，它就是一个简单的放大功能。不过，语句 `(x0/(x0**2))+x1` 比较起来更有意思一些。它是一个指数表达式，给定一个线性系列的输入值（例如，0 ~ 10），它产生的输出将如图 10-9 所示的那样。

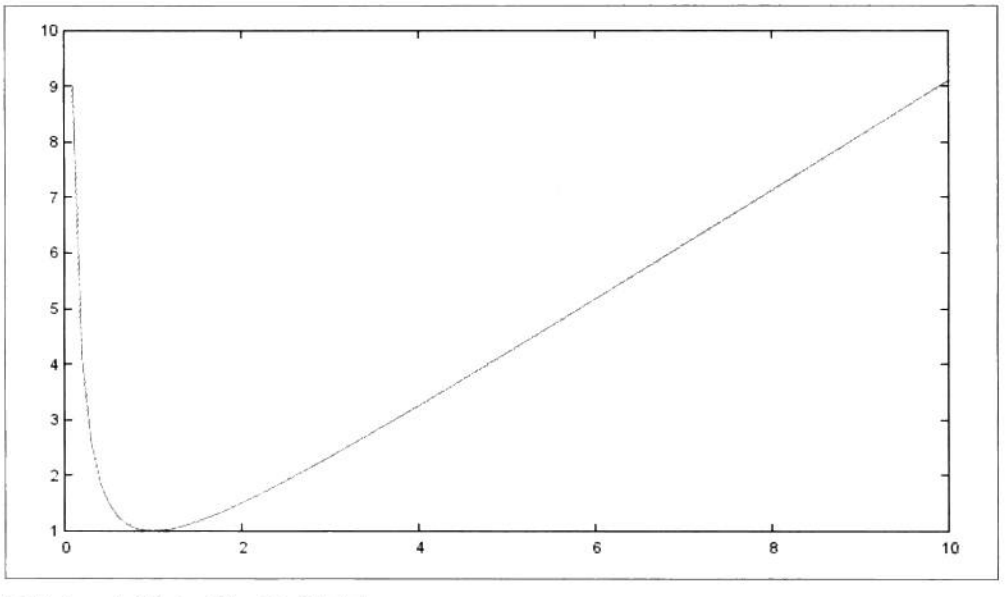


图10-9：（x0/(x0**2))+x1）的图示

使用用户定义函式有以下主要限制：

1. 用户定义的函式不能使用除 `x0` 和 `x1` 之外的任何变量。
2. 变量 `x0` 和 `x1` 是只读的（也就是只能是输入）。
3. 一个用户定义的函式不能包含赋值。
4. 用户定义的函式必须是一条单语句；多条语句和条件语句是不允许的。

369

即使有这些限制，使用用户定义功能还是可以完成相当多的任务，包括模拟一些物理特性，

比如惯性延迟或简单的数字滤波器。

下面的例子演示了一个用户定义功能是如何被使用的：

```
#! /bin/python
# TestDevSim3.py
#
# 演示使用自定义的函式字符串。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import  time
from    DevSim  import  DevSim
import  SimLib.RetCodes as RC
import  DevSim.DevSimDefs as DS


370     def testDevSim3():
            print "Init DevSim"
            simIO = DevSim.DevSim()

            # 设置仿真设备
            simIO.setInputSrc(DS.INCHAN1, DS.EXT_IN)
            simIO.setOutputDest(DS.OUTCHAN1, DS.INCHAN1)
            simIO.setFunction(DS.INCHAN1, "x0 * 2")

            simIO.startSim = True
            time.sleep(1)

            loopcount = 0
            while loopcount < 10:
                # 发送数据到模拟器
                simIO.sendData(DS.INCHAN1, loopcount)

                # 调用函数从仿真器采集数据
                rc, simdata = simIO.readData(DS.OUTCHAN1)
                if rc != RC.NO_ERR:
                    print "DevSim returned: %d on read" % rc
                    break

                print "%d %f" % (loopcount, simdata)

                loopcount += 1
            # 返回循环继续处理

            simIO.stopSim = True
            print "DevSim terminated"
```

这个例子仅仅是前面提到的两倍放大功能。

循环功能

371

仿真器的循环功能提供了一个方便的数据源，它是可预见的循环数据。对于数据的波形和时序，你有完全的控制权。举个例子，你能够建立一个仿真，它使用脉冲模式来评估一个控制算法的响应。其他循环功能可以被用来模拟机械装置的速度反馈，用来模拟温度随时间的变化，或者作为数据源去检查极限传感器。

因为循环功能拥有运行在异步模式下的能力，所以你需要确认不会在循环数据失真的情况下终止，之所以出现失真，是因为数据读取速率太慢，如图 10-10 所示。在这种情况下，或者提高采样率，或者减低循环速率，以保证样本数据读取速度不少于循环速率的 4 倍。

噪声

最后，在输出之前，有随机数据注入功能。不管是数据还是随机“噪声”的相对范围，或者级别，都是可控的。这允许你建立它们两者之间的平衡。注意，随机数据是被加到数据流上，不是简单的注入。它不考虑模拟离散的瞬变事件，只是模拟调制噪声，这种噪声可以在噪声电压源中被发现（或者一个被腐蚀的连接器）。

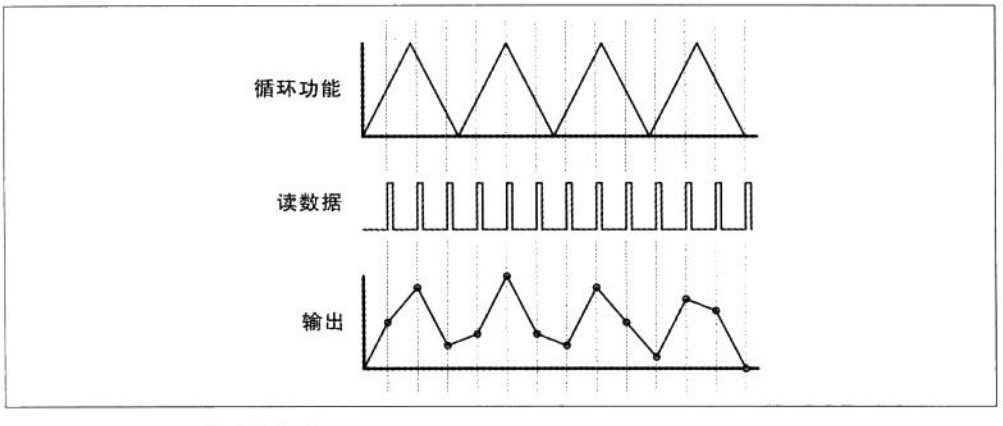


图10-10：循环数据的读出失真

交流电源控制器仿真

接下来我们要看的例子是一个八通道交流电源控制装置的仿真，这种类型的装置经常在大型服务器安装场所、实验室和工业设施中被发现。这个例子将展示一个简单的命令响应类型的仪器如何运转，同时也提供一些对采用了串行接口仪器内部的观察。这里介绍的内容可以在其他设备中看到，比如，激光控制器、电子测试装备、温度控制器，以及运动控制装备中。我们也会看一下在不使用第二台计算机以及物理电缆的情况下，如何

372

同一个串行接口仿真器通信。

简单电源控制器模型

这个仿真示范了一个假想的叫做简单电源控制器（Simple Power Controller，SPC）的设备。它没有任何在真实装备上常见的附加功能，比如，密码保护、控制器单元 ID 分配等，主要是因为这些特性对控制电源来说不是真的有必要。不过，它模拟了每个 AC 通道上的电子断路器所包含的东西。这个和基于航天应用中的电源控制设备相似。它们保护每个通道防止过流的情况，并在必要时能够远程重置。

假想的简单电源控制器(SPC)的示意图如图 10-11 所示。正如你所看到的，它相当简单(事实上，大部分设备都像这样)。装备的“智能”全部包含在它的微控制器内部。

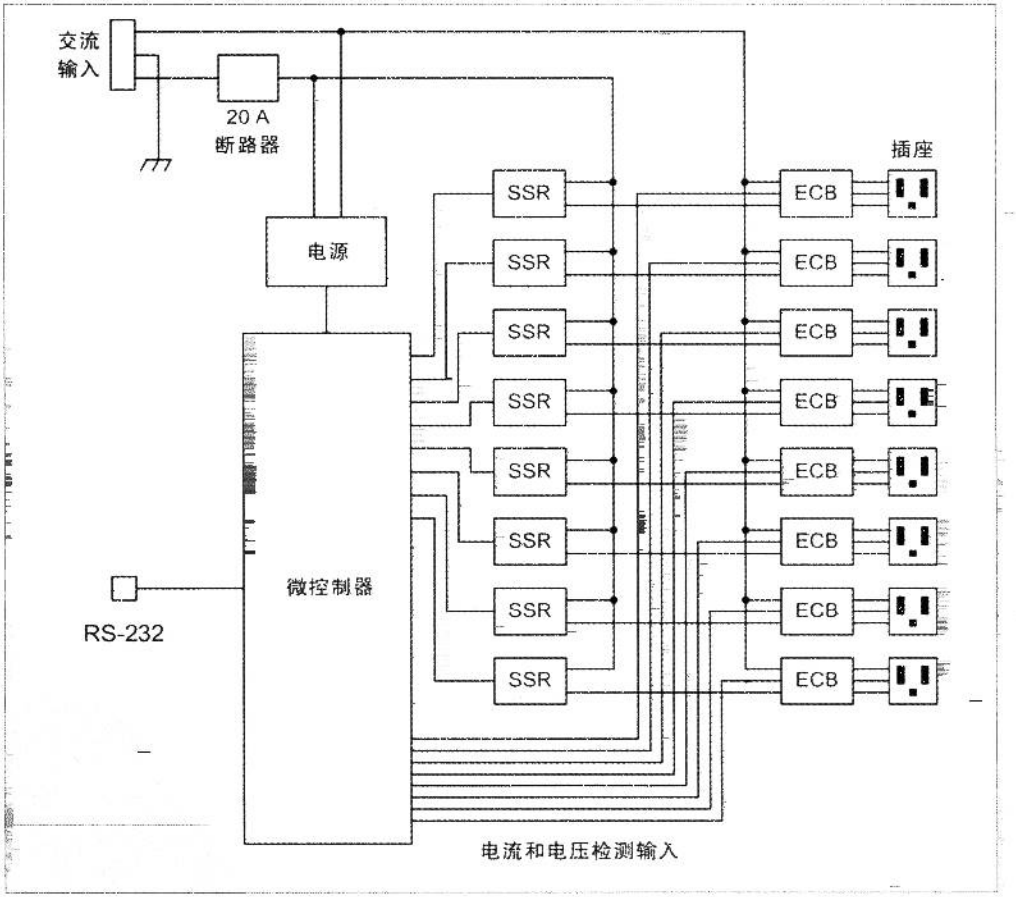


图10-11：交流仿真功能块示意图

在图10-11中，标记为SSR的块是固态继电器，而微控制器可以是任何合适的设备。注意，该图没有显示任何前面板的控制，是因为仿真器不会涉及这些。该仿真器的目标是依照远程控制接口去模拟微控制器的功能。当然，如果你愿意，也可加入一个漂亮的GUI（我们将在第13章讨论用户界面）。

SPC的串行接口和虚拟串行端口

与DevSim仿真器和它的API不同，SPC仿真器使用一个串行接口，它是一个简单的波特率为9800的N-8-1类型的接口，它使用*pySerial*库来实现，我们将在第11章讨论这个库的细节。目前，我们可以假定*pySerial*提供了所有必需的功能来打开串口、设置串口参数，以及读写数据。

在使用串行接口的模拟器及与它接口的对象之间需要端口连接。一个方法是使用两台计算机，在它们之间用对绞线连接（像第7章介绍的那样），但是第三台计算机并不总是可行的选项，何况一些计算机甚至都没有串口可用（比如笔记本和上网本）。

解决方法是使用被称为“虚拟端口”的东西在两个使用串行接口的应用之间创建连接。一个Windows环境下非常实用工具是Vyacheslav Frolov的*com0com*包，你可以从*http://com0com.sourceforge.net*下载它。

*com0com*通过创建一对配置为使用双绞线连接的虚拟串口来工作。当这些端口是为了那些只能处理像COM1、COM2、COM3等这样名字的程序所创建的时，你可以为它们分配这样的标准名称，即从一个程序观点去看，这些端口的表现完全像是真实的物理端口。你能够设置波特率、查询状态，总之，可以做正常串口上做的所有事情。图10-12显示了仪器软件是如何使用两个虚拟串口来和一个串行接口类型的仿真器通信的。

373

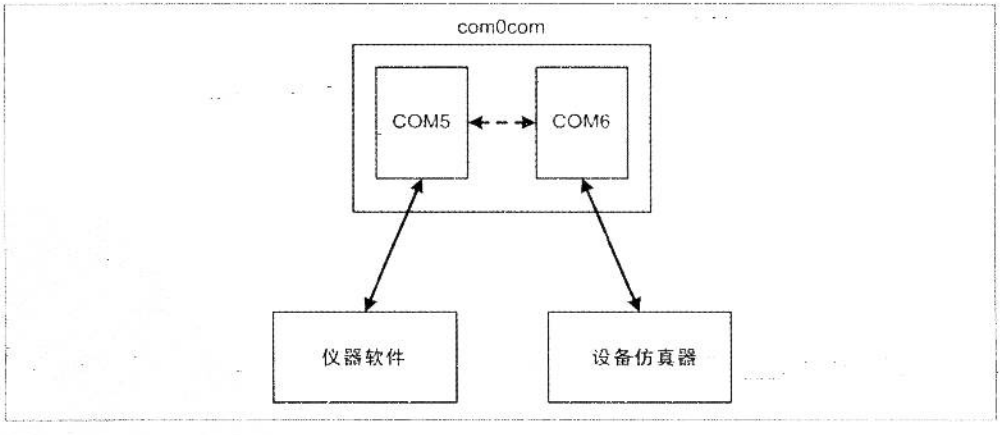


图10-12：使用*com0com*

Linux 用户可以改为使用 *tty0tty*，它可以从 *http://tty0tty.sourceforge.net* 处获得。它相当小，文档也较简单，不过它的用户级版本对我们来说，在 Ubuntu10.04 上工作得很好。

374

与 SPC 通信

主要有两种方法与 SPC 仿真交互：直接通过终端仿真器，或者经由仪器软件。如果你正好想动手练习这个仿真器，你就需要一些方法来发送命令并观察响应。这里选定的工具是终端仿真器。实际上，你使用什么终端仿真器没有关系，只要你能配置它来使用 *com0com*（或者 *tty0tty*）创建的虚拟串口即可。

在 Windows 2000 和 Windows XP 系统上，你可以使用古老的超级终端，或者希望查看一下 Tera Term（我们稍后将研究它），在 Linux 系统上，有 minicom 仿真器可用。如果你使用的是 Windows Vista 或者 Windows 7。你需要寻找一个终端仿真器，有很多可用的。使用终端仿真器和虚拟串口时，其设置基本上和图 10-12 一样，不过“仪器软件”被“终端仿真器”所替代。

SPC 命令集

SPC 使用了 10 个简单的命令来控制输出、检查状态，以及设置配置参数。所有的命令都是 3 个字符的长度，并且所有的命令至少都带有一个参数。命令列表见表 10-2。

表10-2：SPC命令

命令	参数	说明
ALL	state	马上使能或者禁用所有的 8 个交流通道
POW	ch,state	设置指定通道的状态为打开或者关闭
SEQ	state	开始上电或者断电流程
375 STM	time	设置两个顺序步骤间的延时，以毫秒为单位
SOR	ch,ch,ch,ch,ch,ch,ch,ch	定义上电或者断电的顺序
SEM	mode	设置序列的错误处理模式
CHK	ch	返回指定通道或者所有通道的 on/off 状态
ECB	ch	返回指定电子断路器（ECB）或所有 ECB 的正确 / 错误状态
LIM	ch, amps	设定指定通道 ECB 的电流限定值
RST	ch	重置指定通道的 ECB

命令说明

下面是几个 SPC 命令的通用说明：

- 所有的命令至少需要一个参数。没有 0 参数的命令。

- 通道使用 ASCII 码的数字 1 ~ 8 来标记（包括 1 和 8）。数值 0 在 CHK 命令里用来表示所有的通道。
- ASCII 码的数字 1 和 0 分别被用来在 ALL 和 POW 命令里表示 On 和 Off，在 SEQ 命令里表示启动和关机。
- 序列模式命令 SEM 用一个单独的 ASCII 码数字（0、1 或者 2）来表示模式选择项。
- 在 STM 和 LIM 命令中使用的时间和电流限定值是纯粹的 ASCII 格式的整数值，并且电流（单位为安培）最大为 2 位数（0 ~ 99），时间（单位为秒[译注 1]）最大为 3 位数（0 ~ 999）。
- SOR 命令使用 1 ~ 8 通道数的列表作为参数，通道数之间用逗号分隔。
- 所有的命令返回一个响应；没有哑命令。命令的响应或者是 1，或者是 0。1 表示 On 或者 OK，0 表示 Off 或者错误。

现在，让我们进一步看一下这些命令：

ALL *state*:

按顺序依次使能或者停用所有的 8 个 AC 通道，没有中间延时。*state* 参数可以是 1（On）或者 0（off）。

如果成功，则响应 1(OK)；如果任意已上电通道的 ECB 被触发，则返回 0。立即返回（不等待命令执行完）。

POW *ch, state*:

设置一个通道的状态为 On 或者 Off。

Ch 是通道号，*state* 可以是 1（On）或者 0(Off)。

如果成功，则响应 1；如果 ECB 在上电状态被触发或者其他错误发生，则返回 0。在返回前，等待命令操作完成。

SEQ *state*:

376

命令控制器开始上电或者断电流程。如果事先没有使用 SOR 命令指定顺序，上电的顺序将是从低到高，断电的顺序则相反。

参数 *state* 可以是 1（上电）或者 0（断电）。如果成功，则响应 1；如果任何一个通道的 ECB 在有电状态被触发，则响应 0。马上返回（不等待命令完成）。

STM *time*

设置在上电或者断电序列中每步之间暂停的总时间。默认的暂停时间是 1 秒，time 参数限定为一个整数值。如果成功，则响应 1；如果时间值无效，则响应 0。

SOR *ch,ch,ch,ch,ch,ch,ch,ch*

定义启动和关机的序列顺序。关机的顺序和启动顺序正好相反。这个列表可以包含从 1 到 8 的通道 ID 条目。任何不在该列表中的通道将被排除在序列之外，未使用的

译注 1：原文如此，应当是作者的笔误，根据前述，应该是毫秒。

列表位置用 0 占位。

如果成功，则响应 1；如果序列参数无效，则响应 0。

SEM *mode*:

设置上电序列的错误处理方式，*mode* 定义如下：

0:

标准（默认）操作。如果任何通道的 ECB 出错，控制器将按反序停止所有活动通道的电源。

1:

错误保持状态。如果在启动流程中任何通道的 ECB 出错，控制器将继续按序列启动下一个通道。

2:

错误继续模式。在启动期间，如果任何通道 ECB 出错，控制器将继续该序列列表中的下一个通道。

如果成功，则响应 1，如果 *mode* 参数值无效，则响应 0。

CHK *ch*|0:

返回 *ch* 通道 ECB 的状态：1（OK）或者 0（error）。如果 *ch* 设为 0，则返回所有 8 个通道 ECB 的状态列表，每个状态值之间用逗号来分隔。如果该通道 ECB 跳闸，也返回一个 0 字符通道。使用 ERR 命令检查 ECB 状态。

377 LIM *ch*|0,*amps*:

设置 *ch* 通道 ECB 的电流限定值。如果给定通道 ID 为 0，所有通道的限定值将被指定为 *amps*。

如果成功，则响应为 1；如果通道 ID 或者电流限定值无效，则响应为 0。

RST *ch*:

尝试复位通道 *ch* 的 ECB。

如果成功，则响应 1；如果 ECB 无法复位，则响应 0。

SPC 仿真器内部

SPC 是一个简单的命令 - 响应类型的设备。它永远不需要对自己和主机系统之间的通信进行初始化。这意味着，使用一个简单的命令识别程序就可以有效地模拟它。同时，因为 SPC 是离散的基于状态的仿真器，所以，它需要一组数据去定义每个电源控制通道的状态。一个通道何时以及如何从一个状态转换到另一个状态，取决于前面章节描述的那些命令。图 10-13 显示了 SPC 仿真器为了模仿一个物理系统所需要的内部数据。

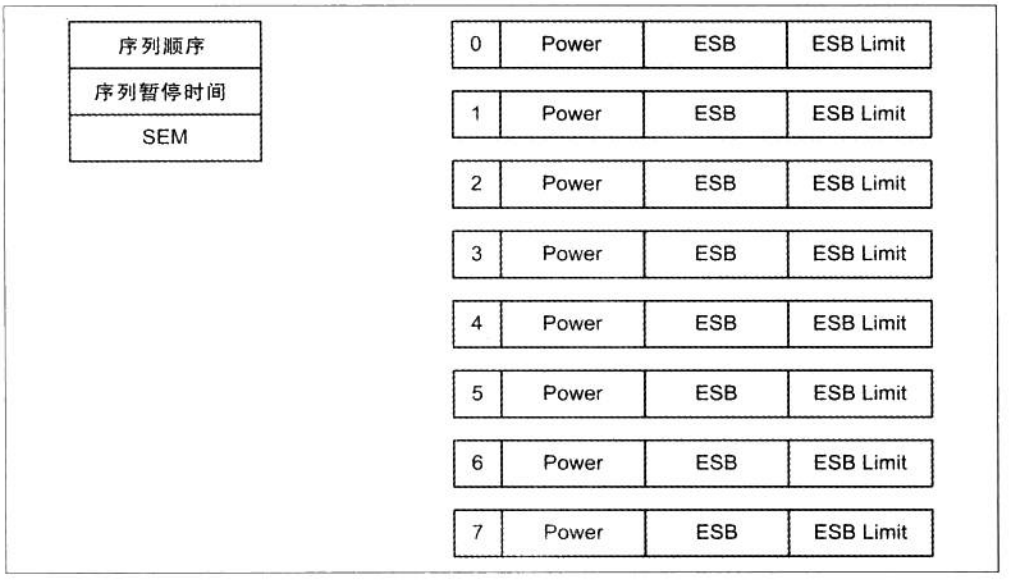


图10-13：SPC内部数据

其中的顺序控制数据对象应用于所有的通道，每个通道有 3 个属性：电源状态（1 或者 0）、ECB 状态（1 或者 0），以及该通道的 ECB 电流限定值。

接下来是命令识别程序，很简单：

378

```
def Dispatch(self, instr):
    cmdstrs = instr.split()

    if len(cmdstrs) >= 2:
        if len(cmdstrs[0]) == 3:
            if cmdstrs[0].upper() == "ALL":
                self.SetAll(cmdstrs)
            elif cmdstrs[0].upper() == "POW":
                self.SetPower(cmdstrs)
            elif cmdstrs[0].upper() == "SEQ":
                self.SetSeq(cmdstrs)
            elif cmdstrs[0].upper() == "STM":
                self.SetSTM(cmdstrs)
            elif cmdstrs[0].upper() == "SOR":
                self.SetOrder(cmdstrs)
            elif cmdstrs[0].upper() == "SEM":
                self.SetSEM(cmdstrs)
            elif cmdstrs[0].upper() == "CHK":
                self.ChkChan(cmdstrs)
            elif cmdstrs[0].upper() == "ECB":
                self.ChkECB(cmdstrs)
```

```
            elif cmdstrs[0].upper() == "LIM":
                self.SetLimit(cmdstrs)
            elif cmdstrs[0].upper() == "RST":
                self.RstChan(cmdstrs)
            else:
                SendResp("ER")
        else:
            SendResp("ER")
    else:
        SendResp("ER")
```

接收到命令以后，Dispatch() 方法被调用。从主机系统进来的字符串被分割到一个列表里，该列表将包含两个或更多的元素。第一个元素将是命令关键字。关键字后面的参数数量根据命令的不同而有所不同，但是没有任何一个命令是零参数的。

当命令被解码后，10 个工具方法中的一个被调用来写入数据到内部数据表，从内部数据表里获取数据，或者调用通道控制来完成命令动作（如果不仅仅是一个状态查询）。注意，每个命令都有一个命令工具方法。

图 10-14 显示了 SPC 仿真器中典型的命令 - 响应式交互的消息序列图（MSC）。

根据命令的不同，从 SPC 命令处理程序回到主机（终端模拟器或者仪器软件）有两种可能的返回路径。一些命令提供了即时响应，不等待通道控制逻辑去完成动作。另一些命令将会等待，然后返回成功或失败的指示。

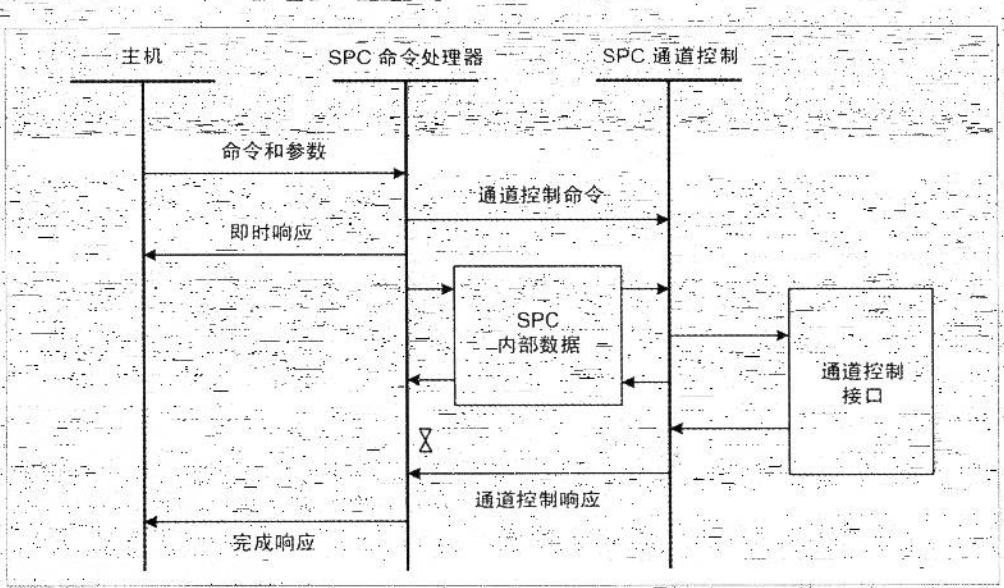


图10-14：SPC命令-响应消息序列图

配置 SPC

SPC 仿真器使用了一个配置文件，也就是常见的“INI”文件，用于保存各种配置参数，并在启动时将它们读出来。*spc.ini* 文件可能会像下面这样：

```
[SPC]
SPORT=COM4
SBAUD=9600
SDATA=8
SPAR=N
SSTOP=1
ECB1=2.5
ECB2=2.5
ECB3=5.0
SOR=[3,2,1,4,5,8,7,8,6]
STM=2.0
SEN=0
```

INI 文件中的值会被加载到内部数据表中；不过它们可能会被通过调用 SPC 命令而重写为新的数值。在 INI 文件中未定义的参数将会采用它们的默认值。

与 SPC 仿真器的交互

当 SPC 仿真器被启动时，它将首先尝试打开 INI 文件中定义的串口。如果成功，它将开始它的主循环并等待进来的命令。当 SPC 接收到一个回车键而没有其他输入时，它将返回提示字符（>）。下面是一个示例对话，它使能电源通道 1：

380

```
> CHK 0
[0,0,0,0,0,0,0,0]
> POW 1,1
1
> CHK 0
[1,0,0,0,0,0,0,0]
```

为了使用 SPC 模拟器和仪器软件，第一步是引起 SPC 的注意。一些类似下面这样的代码片段能够满足要求（我们假设一个串口早已被打开，并且被 sport 对象应用）：

```
gotprompt = False
last_time = time.time()
MAXWAIT = 5.0

while True:
    sport.write("\r")
    instr = sport.read(2)
    if instr==">":
        gotprompt = True
        break
    if time.time() - last_time >MAXWAIT:
```

```
            break
        time.sleep(0.5)
```

这段代码每 500ms 会发送一个回车字符，直到 SPC 发送应答或者超过 5s 没应答。当 SPC 应答一个提示符时，它退出循环，系统准备通信。

使用 SPC 作为一个框架，你应该可以创建任何简单的使用串行控制接口的设备或者仪器的仿真器。SPC 仿真器也展示了一个仿真如何被用来评估一个在现实世界中（还）不存在的设备或者仪器。使用真实系统来工作是无可替代的，不管是真实的还是仿真的，去感受一下什么它可以做，什么它不能做，是可行的。

串行终端仿真器

当使用串行接口的仪器或者子系统来工作时，改进一些通用的工具来创建一个拥有完美可用性的仿真器有时是可行的。

在 Windows 系统下，有一个叫做 Tera Term 的工具是本节将集中关注的工具。它最初由T.Teranashi 在 20 世纪 90 年代中期完成（最后一次更新是 1999 年，那时候 2.3 版发布），Tera Term 不但支持串行 I/O，还支持 Telnet 登录，不过 2.3 的原始发布版本不支持 SSH。

虽然串行终端仿真器不再是一个大的需求，而且 Tera Term 也变得有些过时，不过它有一些东西特别受人关注：一个功能强大的脚本语言。搭配一个像 com0com 这样的工具，使用 Tera Term 去创建一个专业的串行 I/O 仪器的仿真是可行的，在开发和测试时，你
381
可以使用它来和你的仪器软件通信。Tera Term 在通信连接的另一端同样工作得很好，我们曾经使用它作为一个其他应用中的功能测试驱动器，用于一个嵌入式成像系统和一个激光干扰仪系统。当然，我们也很多次仅仅把它当做一个终端仿真器来使用。

你可以从 *http://hp.vector.co.jp/authors/VA002416/teraterm.html* 处下载 Tera Term，并获得更多关于它的信息。它的源代码可以自由获取，并且还可以获取一些插件。详细内容请查看网页。

安装 Tera Term 很容易。下载完归档文件后，把它解压到一个临时位置。然后找到文件 *setup.exe* 并运行它。这样，Tera Term 会被安装在 *c:\Program Files\TTERMPRO* 目录下（除非你指定了不同的位置）。安装完成后，你可以删除临时目录中的内容。

安装完成后，将在 Windows“开始”菜单中的 Programs 里创建一个子项，名称为“Tera Term Pro”。你可以使用鼠标右键拖动标记有“Tera Term Pro”的程序图标到桌面上创建一个图标。据我们所知，Tera Term 工作在 Windows 2000 和 Windows XP 下。

虽然所有的 Linux 安装版都预装了叫做 *minicom* 的终端仿真器工具，不过还有其他的仿真器同样可用。有一些像 *minicom* 这样的，对脚本语言支持相当有限，反之，也有一些功能更加完整。我们倾向于将 Tera Term 看做一个免费的开源串行终端仿真器应该拥有些什么功能的典范，在剩下的讨论里，我们将坚持这一点。如果你正在使用一个 Linux 系统，一定要探索对你可用的所有选项。考虑到 Linux 下有庞大数量的软件可用，我们确信总有一些可以满足你的需求。无论如何，在看过了 Tera Term 可以做什么以后，你应该对找什么样的软件就有了一些主意。

使用终端仿真器脚本

终端仿真器脚本主要的焦点是条件测试。换句话说：*if*（如果）一些东西是这样，*then*（那么）做这些，*else*（否则）做其他事。虽然一个终端仿真器可以有很多附加功能，但基本的拨号并通知一个连接的脚本总会归结为如下形式：

```
:dial
send string "55-1212"
if busy then goto dial
wait connected
if connected print "CONNECTED"
if not connected print "ERROR - COULD NOT CONNECT"
exit
```

如果我们承认 wait 语句实际上是一种 IF-ELSE 语句形式，那我们就可以把这个简单的脚本看做是一个条件测试序列。

这里强调这一点，是因为一旦你理解了终端仿真器中使用的脚本语言背后的范式，把这 382
些工具用于预期外的任务就会变得很容易。

Tera Term 脚本语言（Tera Term 语言，或者像作者那样叫 TTL）是一个全功能的语言，它不仅提供了基本的命令来处理一个通信上下文中的 IF-THEN 判定，而且包含了产生对话窗口、向文件中写数据、从文件中读、字符串转换，以及执行外部程序的命令。该语言提供了流控制语句，比如 IF-THEN（加上 ELSEIF 和 ELSE）、FOR，以及 WHILE。它没有一套完整的数学函数，而且它只支持两种数据类型：整数和字符串。但是，考虑到它最初想要做的，这已经让人感觉很完美了，而且那些也不是一个重要的障碍。你可以选择 Tera Term 程序的 Help 主菜单项来浏览在线文档（没有用户手册）。注意，主帮助文件里将脚本功能称为“宏（MACRO）”。

下面是对 SPC 的一个连接测试，我们前面看到过 Python 版本，现在将它转换为 TTL：

```
waitcnt = 0
```

```
:connect
; 检查最大值
If wantcnt > 9 then
    goto noconnect
endif

send " "
recvln
;recvln 将其返回输入到 "inputstr"
strcompare inputstr ">"
; 根据 "result" 的比较设定 strcompare
if result = 0 then
    goto start
else
    ; 使用数字暂停
    pause 1
    ; 跳回重试
    goto connect
endif

:start
; 略过错误显示
goto endconnect

:noconnect
messagebox "SPC not responding" "Error"

:endconnect
; 此时，用户可开始输入命令
```

383 如果你曾经使用过 BASIC，或者 MS-DOS 和 Windows 系统下所谓的批处理文件，这些代码看起来应该很熟悉。Tera Term 的 TTL 支持子程序（CALL-RETURN），而且它有可用性极好的 WHILE 语句，不过在这个例子中不使用它们。

和大部分终端仿真器一样，Tera Term 在同一时间只处理一个外部连接，所以使用 Tera Term 作为仪器系统的控制逻辑不是很合理（至少不容易，可以用外部程序和数据文件来完成）。Tera Term 最有用的是创建一个仪器或者设备的仿真，来与 Python 写的仪器应用程序通信。实际上，可以在 TTL 中使用 Tera Term 完整地实现 SPC 仿真器。几乎所有带有串行接口的简单仪器都可以考虑使用 Tera Term 来仿真。

Tera Term 对驱动其他系统的重复测试也是很有用的。事实上，在为航天探测器开发图像获取和处理软件时，Tera Term 被用来推送成千上万的测试图片到图像压缩软件，并将每个测试的结果记入日志文件。它工作得很完美，产生了大量数据进行筛选。

显示仿真数据

一个仿真能产生许多有用的数据，但是仅仅看文件里一列列的数据，不像看数据的图表那么直观。本节将向你展示如何使用仿真器产生的数据来创建有趣并且有用的数据图形式的图形化输出。

我们将焦点集中在 *gnuplot*（*http://www.gnuplot.info*）上，它是一个古老的工具，可用于 UNIX 和 Linux 系统，同时也有 Windows 版本可用，所有的版本都可以使用 Python 语言来显示动态产生的数据。第 13 章将讨论用户界面和产生图形输出更加精巧的方法，不过这里是开始的好地方。

gnuplot

gnuplot 是一个强大且完善的图形绘图工具，它能够产生从简单的线状图表到复杂的数据可视化的图形输出。虽然它最初是为 UNIX 开发的，但是同样也有 Windows 版本可用。gnuplot 有一个可供使用的内建命令行形式的用户界面，拥有加载绘图命令和数据文件的能力。它还可以使用所谓的管道来进行命令输入，这允许其他程序驱动绘图显示。本节将简单描述两个方法，它们允许 Python 程序发送数据和命令给 *gnuplot* 以便显示。第一个是一个简单的 Python 的 popen() 方法的示例。虽然该方法实现起来很直接和简单，但它没有做什么事情来帮助你使用 *gnuplot*，它只是发送命令。因此，程序员需要对 *gnuplot* 程序以及它的各种命令和配置参数有很好的理解。第二个方法使用了 Michael Haggerty 的 gnuplot.py 包，该包实现了一个 *gnuplot* 的包装对象，用来让程序员处理命令接口的一些细节。*gnuplot* 的文档包含一系列 HTML 页面，它包含在发行包里，还可以使用在线文档（*http://gnuplot-py.sourceforge.net/doc/Gnuplot/index.html*）。

384

如果你的系统上没有安装 *gnuplot*，第一步就是安装它。如果你运行的是 Linux，则可以直接使用它。如果没有，一个使用包管理（*apt-get*、*rpm*、*synaptic* 等）的快速会话可以用来完成安装。如果你运行的是 Windows，你需要从 SourceForge 下载 *gnuplot* 安装包。如果你乐意，也可以下载源代码，然后从头创建它（不推荐它，除非你确实知道你在做什么，而且你的系统没有包管理工具可用）。*gnuplot* 的当前版本是 4.4.2，你可以从 *http://gnuplot.sourceforge.net* 处找到它。

在 Windows 上安装 gnuplot

这里假设 Linux 用户会使用包管理器安装 *gnuplot*，所以这里不会讲述这个过程。下面的步骤仅适用于 Windows 用户。

如果第一次安装支持包，请按以下顺序执行安装：

1. 解压归档文件 *gp440win32.zip* 到你的 C 盘，或者其他你喜欢的地方。你甚至可以把它放在 C 盘的“Program Files”目录下。当归档文件解压完成后，将会创建一个叫做 *gnuplot* 的目录。
2. 通过 Settings → Control Panel → System → Advanced → Environment Variables 打开环境变量对话框（在 Windows 2000 和 XP 下），或者你也可以用鼠标右键单击桌面上“我的电脑”图标，然后选择“属性”。设置 GNUPLOT 环境变量（设为系统变量，而不是用户变量）指向 *gnuplot* 目录下的 *binary* 子目录。查看 *gnuplot* 目录下的 *INSTALL* 文件，可以获取更多关于可以用到的各种各样环境变量的信息。
   使用环境变量对话框将 *gnuplot\binary* 目录放入 Windows 的搜索路径（PATH 环境变量）。例如，如果你把 *gnuplot* 放到 C 盘根目录下，你应该将 *C:\gnuplot\binary* 加入到搜索路径字符串。注意，路径字符串中条目之间使用分号来分隔。
3. 作为可选项，你可以在桌面上创建一个图标用文件 *wgnuplot.exe* 来启动 *gnuplot*。确认“Start in”参数指向 *gnuplot\binary* 目录（用鼠标右键单击图标，然后从弹出菜单中选择属性）。
4. 完整浏览 *gnuplot\docs* 目录下的文档，尤其是 *gnuplot-4.4.0.pdf*。也阅读 *README.windows*，它位于 *gnuplot* 的根目录下。

385

Windows 版的 *gnuplot* 包包含了以下可执行文件：

wgnuplot.exe

一个 Windows GUI 版本的 *gnuplot*。和非 GUI 版本一样，提供了同样的命令行控制台接口，但是为命令行使用了一个文本编辑器形式的显示，并且包含一个菜单栏和一些可以通过单击来完成常用操作的按钮。

wgnuplot_pipes.exe

和 *wgnuplot* 一样，但是有支持内部管道的优势，形式如：

```
Plot'<awk -f change.awk data.dat'
```

gnuplot.exe

传统的文本（控制台）接口版本的 *gnuplot* 可执行文件支持所有相关的管道功能。这意味着该程序也可以接受 *stdin*（标准输入）上的命令，并在 *stdout*（标准输出）上打印信息。这是在整合 *gnuplot* 和其他程序（例如 Python 程序）时优先选择的执行文件。

pgnuplot.exe

一个“助手”程序，它会接受 *stdin*（标准输入）上的命令，然后通过管道把它们发

给一个活动的（或者新建一个）*wgnuplot.exe* 控制台窗口。命令行选项都被传送给 *wgnuplot*。

你应该为 *wgnuplot* 创建一个快捷方式图标放到桌面上，当 Python 程序创建一个管道时，就要打开 *gnuplot*。

使用 gnuplot

正如前面介绍的，我们将考察从 Python 程序接入 *gnuplot* 的两种方法。第一种很简单，但是需要对 *gnuplot* 命令组有深入的领会。第二种方法需要你自己处理大量的细节，不过它也隐藏了其中一些命令的细节，它是按照其他一些人关于一个 Python-*gnuplot* 接口应该是什么样的观点实现的。你可以选择最容易的方法去完成你的目标，不过在你决定哪种方法对你的应用最有意义之前，至少应该浏览过 *gnuplot* 和 `gnuplot.py` 可用的文档。

方法 1：使用 Python 的 popen() 方法

386

如果你希望能够使用管道来发送命令给运行在 Windows 下的 *gnuplot*，你必须使用 *gnuplot* 版本，而不能用 *wgnuplot*。这是因为 Windows GUI 应用程序（比如 *wgnuplot*）不接收从 *stdin* 来的输入。当你使用 `popen()` 创建管道时，你可以告诉 Python 使用 *gnulot*。作为一种选择，你可以使用 *pgnuplot* 来达到使用 *wgnuplot* 同样的结果。在 Linux 系统上，这不是一个问题（那里没有 *wgnuplot* 或者 *pgnuplot* 二进制文件）。

下面的例子 `gptest.py` 将启动 *gnuplot*，并显示一系列图表：

```
#! /usr/bin/python
# gptest.py

import os
import time

f=os.popen('gnuplot','W')

print >> f, 'set title "Simple plot demo" 1, 1 font "arial,11"'
print >> f, 'set key font "arial, 9"'
print >> f, 'set tics font "arial, 8"'

print >> f, "set yrange[-20:+20]"
print >> f, "set xrange[-10:+10]"
print >> f, 'set xlabel "Input" font "arial,11"'
print >> f, 'set ylabel "Output" font "arial,11"'

for n in range(100)
    #plot sine output with zeo line (the 0 term)
    Print >> f, 'plot sin(X * %i) * 10, 0' % (n)
```

```
        time.sleep(0.1)

    f.flush()

    #退出前停止
    time.sleep(2)
```

为了运行这个例子，保存这些代码到一个文件（例如：gptest.py），或者从本书的源代码中载入该文件。在 Windows 机器上，输入以下内容到命令提示：

```
python gptest.py
```

在 Linux 下，你可以仅仅输入脚本名字，假定文件被标记为可执行，并且 Python 解释器位于 */usr/bin*：

```
% gptest.py
```

当它运行时，你应该看到一个扩展又收缩几次的正弦波。实际上发生的事情是，每次 plot 命令被调用时，*gnuplot* 生成一个 *x* 范围从 −10 到 +10 的图表。结果看上去好像一个动画图片，但这实际上是一系列的图表接连不断地呈现出来的结果。注意，*gnuplot* 窗口在脚本一完成就关闭了，同时 Python 终止。

387 下面两行来自方法 1 示例的代码，有可能被选择包含到 *gnuplot.ini* 文件中，该文件和 *gnuplot* 一样位于 binary 目录下：

```
set key font "arial, 9"
set tics font "arial, 8"
```

请参考 *gnuplot* 文档来获取更多关于 *gnuplot.ini* 文件以及它的使用信息。

方法 2：gnuplot.py

第二种方法是使用 Michael Haggerty 的 gnuplot.py 包（1.8 版），该程序可以从 *http://gnuplot-py.sourceforge.net* 获取。你还需要 *Numpy* 包（1.4.1 版本），可以从 *http://numpy.scipy.org* 获取。对于 Windows，你需要下载和安装 *numpy-1.4.1-win32-superpack-python26.exe*（只需要运行该文件来开始安装）。

要安装 gnuplot.py，依照下列步骤：

1. 将 *gnuplot-py-1.8.zip* 解压到一个临时目录。它将创建一个名为 *gnuplot-py-1.8* 的文件夹。
2. 打开命令窗口，在命令窗口里进入 *gnuplot-py-1.8* 临时目录，你会看到一个名为 *setup.py* 的文件。

3. 在命令提示符后输入：

```
python setup.py install
```

在执行安装脚本期间，你将看到很多行输出出现在屏幕上。如果遇到一个错误，安装脚本将停止；否则，你现在已经准备好继续往下进行了。

测试 gnuplot.py

在 *<python>\Lib\site-pckages\Gnuplot* 目录下，运行 *demo.py* 文件，命令如下：

```
python demo.py
```

<python> 是你安装 Python 2.6 的地址，在 Windows 系统下，它可能是 *:C:\Python2.6*。

如果一切顺利，你将看到一个图表显示出来。在命令窗口中按下 Enter 键，将会使一系列图形显示出来。

使用 gnuplot.py

如果按照它被导入到你的应用中的方式来说，gnuplot.py 多少有一些不寻常。如果你看一下 *demo.py*，你将看到它导入了 *Gnuplot* 和 *Gnuplot.funcutils*。但是在 *Gnuplot* 目录下没有“Gnuplot.py”。这里发生的事情是包初始化器 _init_.py 被导入。它按顺序导入其余必要的模块。_init_.py 模块也包含顶层的含有许多信息的文档字符串。如果你检查 _init_.py，你可能也会注意到在文件的底部包含以下代码： 388

```
if _name_ == '_main_':
    import demo
    demo.demo()
```

这意味着你可以从 *Gnuplot* 目录输入：

```
python _init_.py
```

然后，演示程序将会运行。

这种方法很有意思的是，当 gnuplot.py 包被导入时，_init_.py 模块会马上被评估，必要的导入将会放在适当的地方，并且在你的程序之前就可用。

若要导入 gnuplot.py 到你的程序中，你至少必须导入主模块：

```
import Gnuplot
```

你可以像下面这样使用点号导入附加模块：

```
import Gnuplot.funcutils
```

使用 gnuplot 将仿真器数据图表化

接下来的几个例子会把一个包含一组单字段记录的数据文件内容图表化（我们将在第 12 章看到 ASCII 数据文件的更多细节）。

我们将使用第 9 章介绍的 PID 类创建一个数据文件，用于图形化控制器的脉冲响应。

首先，是如何使用管道方法来完成：

```
#! /bin/python
# PIDPlot.py
#
# 使用 gnuplot 将 PID 函式的输出生成为图形
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import time
import os
import PID

def PIDPlot(Kp=1, Ki=0, Kd=0):
    pid = PID.PID()

    pid.SetKp(Kp)
    pid.SetKi(Ki)
    pid.SetKd(Kd)

    time.sleep(.1)
389 f = open('pidplot.dat','w')

    sp = 0
    fb = 0
    outv = 0

    print "Kp: %2.3f Ki: %2.3f Kd: %2.3f" %\
          (pid.Kp, pid.Ki, pid.Kd)

    for i in range(1,51):
        # 求和节点
        err = sp - fb

        # PID 块
        outv = pid.GenOut(err)

        # 控制反馈
        if sp > 0:
            fb += (outv - (1/i))
```

```
            # 从 sp = 0 开始，在 t(10) 模拟一个阶跃输入
            if i > 9:
                sp = 1

            print >> f,"%d % 2.3f % 2.3f % 2.3f % 2.3f" %\
                      (i, sp, fb, err, outv)
            time.sleep(.05)

        f.close()

    gp=os.popen('gnuplot', 'w')
    print >> gp, "set yrange[-1:2]"

    for i in range(0, 10):
        kpval = 0.9 + (i * .1)
        PIDPlot(kpval)
        print >> gp, "plot 'pidplot.dat' using 1:2 with lines, \
                          'pidplot.dat' using 1:3 with lines"

    raw_input('Press return to exit...\n')
```

这将产生一系列 K_p 值范围从 0.9 到 1.8 的图表。最后的图表会一直被显示，直到用户按下 Enter 键为止，这时程序终止，*gnuplot* 被关闭。图表最后的输出（K_p=1.8）如图 10-15 所示。

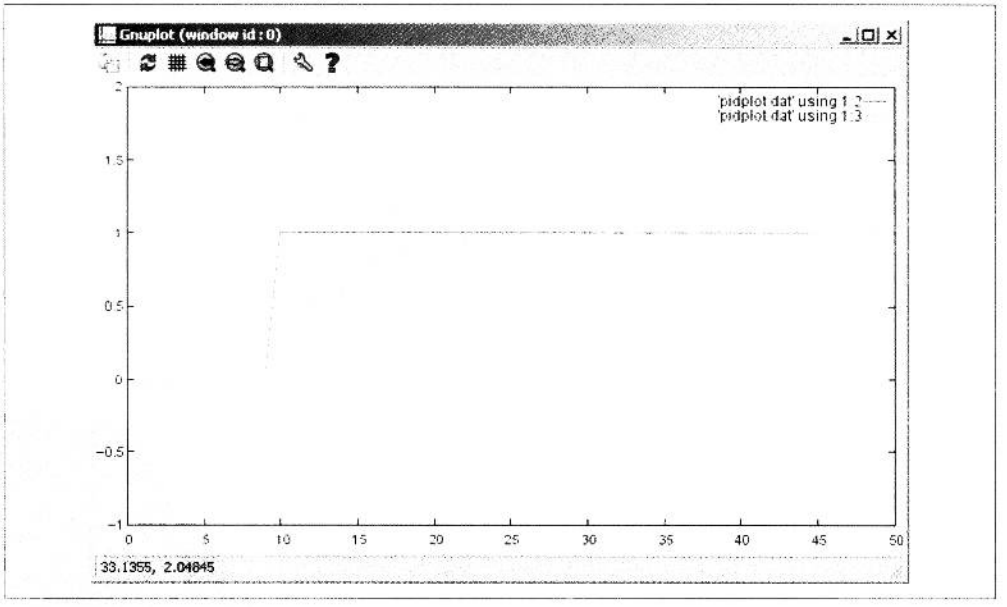


图10-15：gnuplot PID图表

你可能记得第 9 章中，如果比例放大率太高，系统在响应突然的输入改变时，将显示出不稳定性，这在图 10-15 中可以看到。

接下来，是用 gnuplot.py 图表化数据的方法：

```
# PIDPlot.py
#
# 使用 gnuplot 将 PID 函式的输出生成为图形
#
# 源代码出自 "Real World Instrumentation with Python"
# By J.M.Hughes, published by O'Reilly.

390  import Gnuplot, Gnuplot.funcutils

import time
import os
import PID

def PIDPlot(Kp=1, Ki=0, Kd=0):
    pid = PID.PID()

    pid.SetKp(Kp)
    pid.SetKi(Ki)
    pid.SetKd(Kd)

    time.sleep(.1)
    f = open('pidplot.dat', 'W')

    sp = 0
    fb = 0
    outv = 0

    print "Kp: %2.3f Ki: %2.3f Kd: %2.3f" %\
          (pid.Kp, pid.Ki, pid.Kd)

    for i in range(1,51):
        # 求和节点
        err = sp - fb

        #PID 块
        outv = pid.GenOut(err)

        # 控制反馈
        if sp > 0:
            fb += (outv - (1/i))

        # 从 sp = 0 开始，在 t(10) 模拟一个阶跃输入
        if i > 9:
            sp =1
```

```
            print >> f, "%d  % 2.3f  % 2.3f  %2.3f  % 2.3f" %\
                        (i, sp, fb, err, outv)
            time.sleep(.05)

        f.close()

    gp = Gnuplot.Gnuplot()
    gp.clear()
    gp.title('PID Response')
    gp.set_range('yrange', (-1,2))

    for i in range(0, 10):
        kpval = 0.9 + (i * .1)
        PIDPlot(kpval)
        gp.plot(Gnuplot.File('pidplot.dat', using=(1,2), with_='lines'),
                Gnuplot.File('pidplot.dat', using=(1,3), with_='lines'))

    raw_input('Press return to exit…\n')
```

391

输出结果看起来和前面的一样，只是现在在图表上面多了一个标题。关于 `gnuplot.py` 可以做的其他事情的一些例子，请查看它包含的 `demo.py` 和 `test.py` 文件，它们位于 *gnuplot* 目录下，也就是包安装的地方（通常是 *python26/Lib/site-packages*）。

创建你自己的仿真器

到目前为止，我们已经介绍了什么是仿真器，并且也介绍了使用它们的一些方法，现在是时间考虑一下创建这样一个仿真器需要做些什么了。我们是仿真器的一个忠实粉丝，不过我们也在努力用一些现实主义来综合自己的热情。一个随便摆弄一下仿真的项目将很容易耗掉大部分时间和预算。所以，在开始创建自己的仿真器之前，有以下 3 个关键问题应该问问自己：

1. 为什么你想使用仿真器？
2. 你想仿真什么？
3. 你有多少时间和精力用来创建一个仿真器？

392

你如何回答这些问题将有助于你避免在一些实际上不需要的事情上花费时间（即使创建和摆弄它很有趣）。

确认仿真器的必要性

首先要明确是否真的需要仿真？如果没有仿真，就无法完成某个仪器控制软件，这才算是充分理由。正如本章开始介绍的，需要仿真器的情况只可能存在于编写软件时，如果

硬件在某个约定的日期无法完成时，与其冒着进度延误的风险等待硬件，不如使用仿真，这至少可以开始创建仪器软件并测试整个框架。

另一个例子是，硬件有时是唯一且特殊的，软件让它做的一些事情有明确风险，可能损坏它。一些运动控制系统就属于这种（回忆一下第 9 章关于超出控制范围的 PID 伺服控制器的内容），比如一些可能会陷入高温或者高压系统。

仿真的范围

当你考虑仿真器时，写任何代码之前先定义它的仿真范围是必不可少的。换句话说，它将仿真什么，以及为了让它可以使用需要达到什么样的精度。

图 10-16 显示了一种形象化的方式展现仿真可能涉及的细节和复杂度的级别。起点是最下面的 I/O 仿真级，当附件层被加入时，仿真的保真度会提高。然而，它花费的时间和精力也会增加。

在很多情况下，仅仅有 I/O 和命令处理级别可能就足够了。我们在本章看到的仿真器实际上不比这里多多少，只是增加了一点点处理设置和恢复状态及参数的功能。图 10-16 中方块的面积越大，意味着你加入的功能越多，仿真器就变得越复杂和昂贵。

虽然每一个你能考虑的仿真器必须基于项目的特定需求来做评估，但这句话通常是适用的：如果你能够精确地仿真接口和命令处理过程，它可能已经足够好，可以让你开始进行工作。如果后面证明一些细节真的是必需的，那么你总能够在实现时添加它们。

393

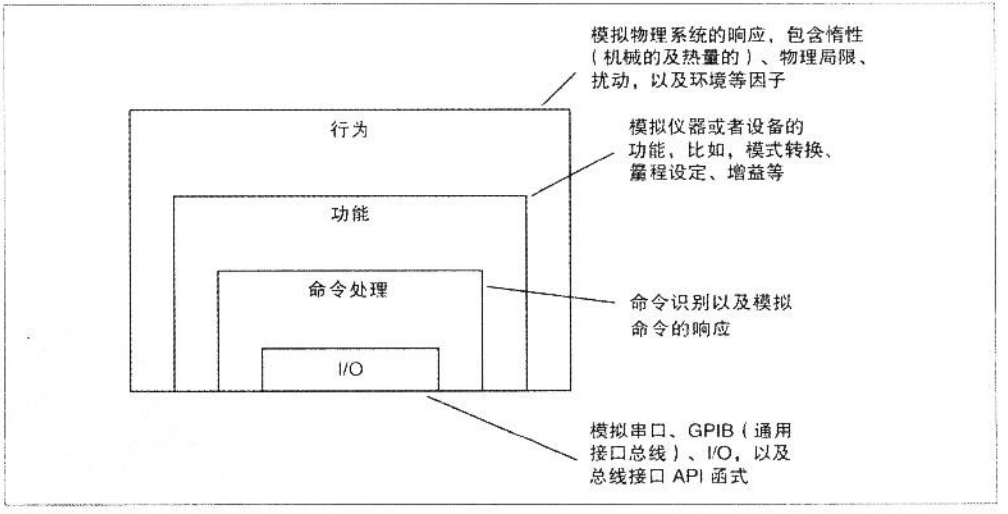


图10-16：仿真级别

时间和精力

写一个仿真器要花费时间和精力。有时候，这种花费是需要引起注意的，有一种情况是不寻常的，那就是创建和验证仿真器花费的精力与你实现所开发的系统花费的精力一样多（甚至更多）。如果系统非常关键（比如实验性飞行器的一部分），或者仿真器通用性很高，可以被其他项目使用，又或者它能够在稍后的生产环境中被用于做产品的测试，那么这种情况可能是合理的。如果它只是为一个简单系统做的一次性的事情，就不应该比它的实际需要更复杂。甚至可以说：如果有一个好的调试器可用，那么仿真器压根就不是真的需要，这种情况下，你会很乐意使用一个简单文件来捕获数据。

小结

如果使用恰当，仿真将是一个强大的工具，可以帮助你节省精力，并避免后期硬件和软件都准备好后才发现的代价巨大的错误和因它导致的项目延误。它同时也是一个潜在的时间吞噬机和开发延误的可能原因，所以，决定什么时间和如何使用仿真是很重要的。

推荐阅读

394

如果你有兴趣更深入了解仿真，下面的书籍是一个好的开始：

Simulation Modeling Handbook: A Practical Approach. Christopher Chung(ed.), CRC Press,2003.

一本仿真主题的论文和随笔集，它集中了实用性的应用程序，而不是理论问题。该书提供了详细的规程，覆盖了问题分析、模型开发，以及数据分析。

Software Fault Injection: Inoculating Programs Against Errors. J.Voas and G.McGraw, John Wiley & Sons , 1998.

一本最早深入讲解故障注入主题的书籍之一。虽然是面向高端的关注安全的容错性系统，不过这本书的技术和内容可以被用于任何软件开发项目。

Gnuplot in Action. Philipp K.Janert, Manning Publications, 2009.

以如何使用 *gnuplot* 创建有用和有趣的数据可视化显示为线索，对 *gnuplot* 做了一次深入探寻。

虽然可用的入门级材料看起来非常少，但还有一些通用资源可以在 Web 上找到。下面是几个你可能感兴趣的链接：

http://www4.ncsu.edu/~hp/simulation.pdf

这个链接指向 PDF 版本的 *Computer Simulation Techniques: The Definitive*

Introduction!, 作者是 Harry Perros。它覆盖的内容其实不像名字所暗示的那么广泛，这本书介绍了传统的运筹学（OR）类型的仿真，而且包括大量的关于随机性、采样理论以及估算技术的内容。

http://sip.clarku.edu/index.html

书籍 *Introduction to Computer Simulation Methods* 的配套网站。该书的作者是 Harvey Gould、Jan Tobochnik 和 Woflgang Christion（Addison-Wesley 出版社）。虽然书籍本身不能从这里获取，但是有大量有用的注释、教程以及一些 PDF 格式的样章。这本书主要介绍计算物理学以及物理系统的仿真。我们在这里推荐它，主要是因为它提供了使用数学技术处理各种仿真问题的方法。

第 11 章 395

仪器数据I/O

不要在取得数据之前就开始推理。

——Sir Arthur Conan Doyle

第 7 章中我们已经见到过各种物理接口和信号协议，这些东西你可能已经在某些仪器系统中遇到过。现在，我们将学习如何使用这些接口在现实世界和我们的应用之间传递数据。

仪器系统收集或者产生的数据有多种格式，用于满足各方面的需求。我们先讨论一些接口的格式和协议，并定义一些后继的示例中将会用到的基本概念，以此作为本章的开始。然后我们快速地浏览一些 Python 所支持的作为可用接口的工具包：即 *pySerial*、*pyParallel* 和 *PyVISA* 包。

最后，我会向你展示一些读/写工控数据的技术。我们将看到阻塞和非阻塞 I/O 的区别，异步输入和输出事件，以及如何管理潜在的数据 I/O 错误以使应用更为健壮。

数据 I/O：接口软件

多年来，计算机接口硬件已经从简单的、使用串行通信设备和映射到计算机内存地址空间的 I/O 寄存器，发展成为拥有自己的内置处理器、板载逻辑、先进的协议和大量 API 的复杂的子系统。随着复杂性的增加，独特的接口和协议的数量也开始增多。正如你可能想到的那样，如果一个系统必须支持多个特定接口，且每个接口都具有自己独特的处理方式，这可能会导致一些很麻烦的问题。

人们很早就意识到，每个设备都拥有一套自定义接口没多大意义，更何况是很多设备都共享相同的内部函式，甚至具有相似的函式。为了遏制即将产生的混乱并在不同的应用 396

领域间建立一致的接口，各种行业标准化组织纷纷成立。这些组织开始为接口和可能使用这些接口的软件定义了一系列的原则和规定。这些标准后来作为普遍性的解决方案被各种不同的设备所接受。电子工业协会（EIA）在1962年公布了最初的RS-232规范，这一规范经过几次修订后，一直沿用到今天。各种通用标准也被其他组织开发出来，如美国国家标准协会（ANSI）、电气电子工程师学会（IEEE），以及可互换虚拟仪器（IVI）基金会。

话虽如此，但偶尔也会有例外。尽管有大量的通用标准用于通信和仪器接口，但不是每个制造商都遵循它们，有时一个设备很难适用一个已有框架。如果你想在你的系统中使用一个以其特定方式来工作的设备，你可能就需要想方设法兼容此设备。特别是当你要使用一个先于标准而存在的旧仪器或设备时，更容易遇到这种情况。

接口格式与协议

不用去关心用于特定接口的连接器是什么类型，甚至连数据在接口之间的传输方式也不用去管，最主要的事情是有数据在主机（或者叫主控，如果你愿意的话）和连接到它的设备之间传输。

当然，在进行数据采集和仪表控制时，数据传输的方式有很多种。一种方法是使用为了诸如串口或USB的通用接口编写的定制软件。另一种方法是利用行业标准的驱动程序，以及在不同物理接口之间提供了统一API的协议，这些物理接口包括串口、USB、GPIB和基于总线的硬件。基于IVI标准的接口驱动就是这样的例子。在本节中，我们将简要地介绍一下如何实现各种命令和数据协议，和一些用于实现它们的通用标准和原则，以及一些工业设备上常用的物理接口。

计算机和现实世界交互的最简单的办法就是通过一个串行或者并行的端口。简单的结果之一是物理上的简单：计算机（通常）已经具有某种类型的串行或并行接口，因此物理上的连接通常只需一根电缆。但从软件的角度来看，它可以是任何简单的东西，特别是使用USB或者GPIB时。我们将会简单地提到这些。

另外一种方法我们已经在第7章见过，就是使用直接连到计算机内部数据和地址总线上的插入式电路板。电路板会以内存地址空间或I/O地址空间中的一段地址的方式呈现给
397 CPU（中央处理单元）。通用CPU上的工作方式如图11-1所示，一类CPU没有为外设提供独立的I/O总线（如Motorola 68000家族），而另一类CPU则专门为I/O功能提供了一条总线（如大多数当代PC上的Intel处理器）。

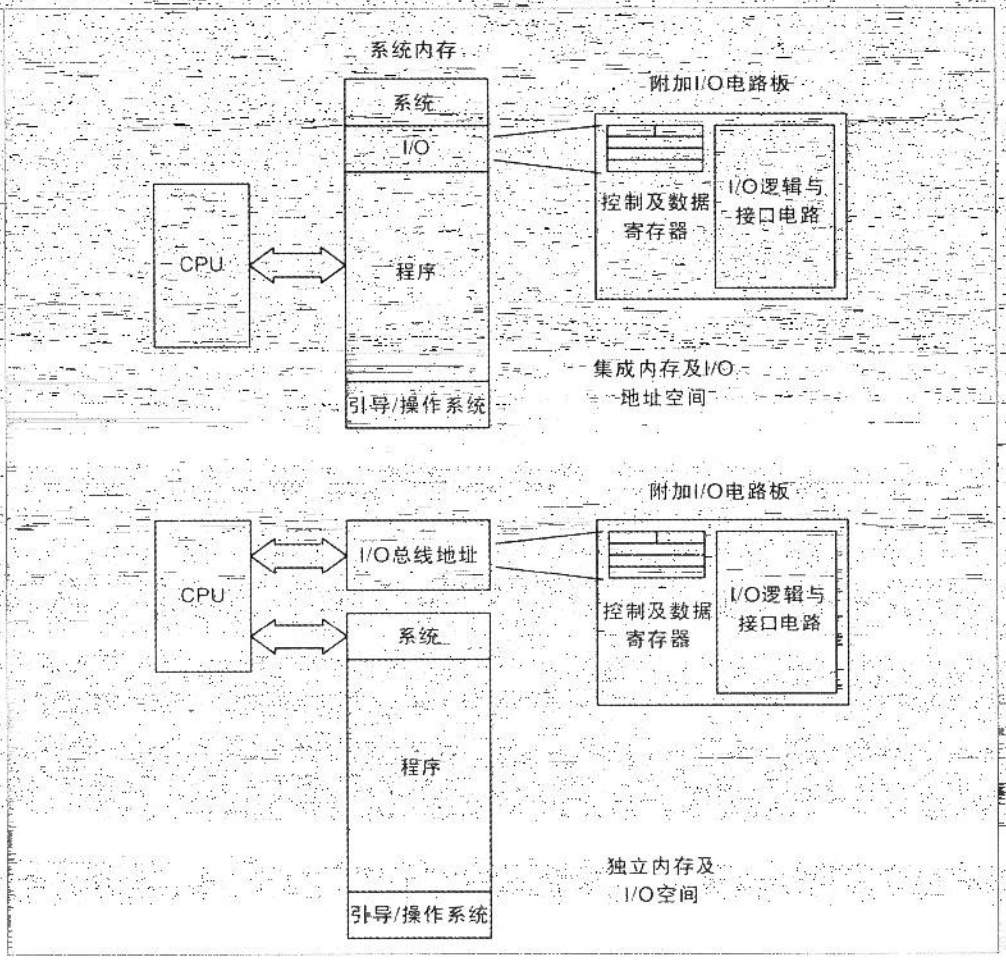


图11-1：CPU的I/O地址方式

第一代 PC 中的电路板使用工业标准结构（ISA）总线。最早的ISA总线，也就是所谓的AT总线（出现在 IBM 的AT 系列PC 中），充分地利用了 Intel CPU内置 I/O 总线和直接将各种扩充板以寄存器的方式暴露给 CPU 的优势。后来的总线机制，如VESA 本地总线、EISA 和 PCI，都使用特定电路（称为“芯片集”）作为调节器工作于 CPU 和 I/O 设备之间。这使得寻址更加灵活，对直接内存访问的支持更好，且速度更快。但不管总线是什么类型，插件电路板都使用寄存器在板上电路和 CPU 之间传递命令和响应数据。

使用寄存器的 I/O 硬件在工作时，有一层称为驱动的软件用来操作底层接口的细节。驱 398
动为使用硬件的程序提供了一组接口，它通常或多或少地以透明的方式工作，如处理中断或大批量数据传输。从应用软件的角度看，驱动就是一组可以调用的函式。从驱动的

角度来看，硬件被看做一些内存中或者 I/O 地址空间中的存储单元。驱动的另一个特性是它可以被整合进操作系统从而扩展其基本功能。在现代操作系统中，出于各种安全和系统可靠性方面的考虑，运行在用户级（user level）的程序通常不能直接访问底层的硬件。操作系统要能够协调对系统中硬件的访问，并防止各种冲突和潜在的系统错误。

驱动还可能被用来访问标准串行或并行端口，而且它们经常使用 USB 或者 GPIB 类接口。在某些情况下，如标准串行接口，操作系统中已有的驱动可能已经足够。但有时，也需要一个特殊的驱动来操作这些接口。还有一些软件通过标准接口和一个外部设备进行通信，此类软件被称做 I/O 处理程序（I/O handler），而不是设备驱动。你可以把一个 I/O 处理程序想象成一个类似于转换器的东西。

如此一来，无论什么形式的 I/O，都需要一个驱动或 I/O 处理程序作为协调器作用于硬件和应用软件之间。另外，在硬件或外设中，有一些用于控制物理接口与设备的硬件逻辑和控制电路通信的功能。图 11-2 展示了较为常见的功能组件，当与连接到主控 PC 的外部仪器或基于总线的设备进行交互时，就可能遇到它们。

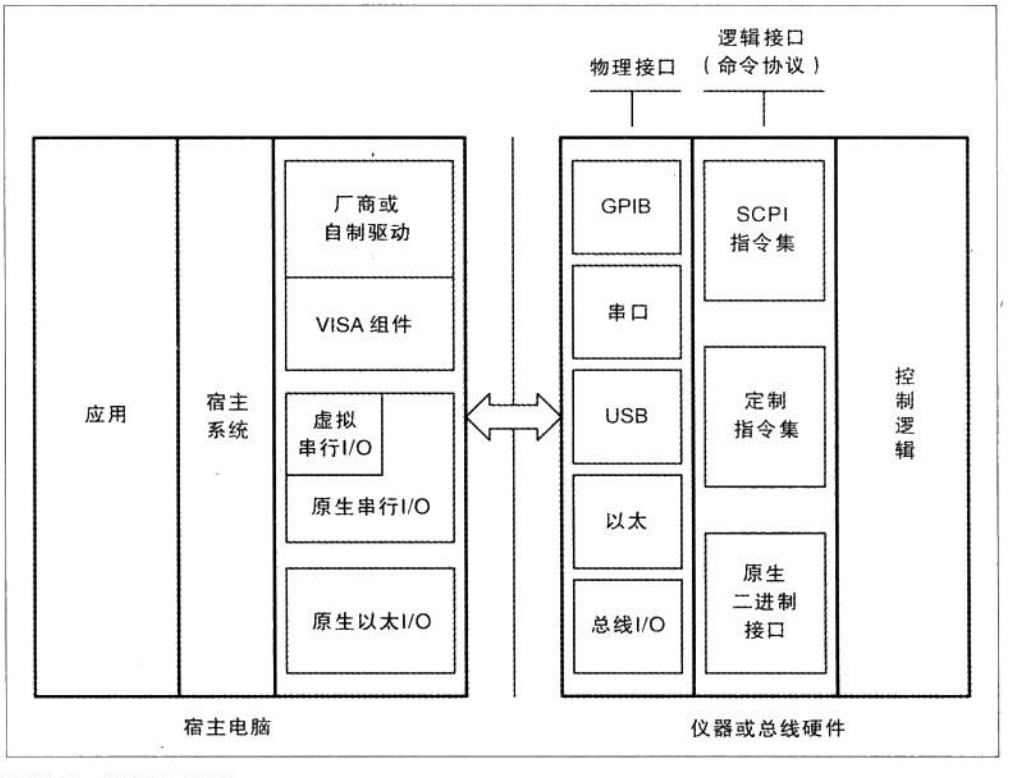


图11-2：仪器接口组件

在接下来的几节中，我们将会定义各种首字母组合词，并更详细地查看每种组件。现在，将图 11-2 作为一个索引思考一下。当我们在后面几个小节中具体地研究各功能组件时，还将回顾它。

IVI——可互换虚拟仪器

在仪器仪表行业中，标准的 IVI 套件通常来自 Windows 平台，而且很多仪器制造商现在都提供了 IVI 兼容的驱动。IVI 主要针对仪器仪表应用定义了一套标准仪器接口和命令。在标准 IVI 套件创建之前，有多个标准被同时使用，其中最值得一提的就是可编程仪器的标准命令（SCPI，有时读做“skippy”）和后来的虚拟仪器软件架构（VISA）标准。每个制造商都能够在他们各自的实现中做一点不合标准的事情，而且他们经常这么干。

SCPI 标准为仪表定义了一组标准命令，而 VISA 定义了一组在不同 I/O（如 GPIB 和 VXI）接口上通用的 API。SCPI 和 VISA 现在都已成为 IVI 套件的一部分。这些标准最主要的重点在于定义通用的接口，以帮助减轻或者消除接口间的编程差异。注意，尽管 SCPI 和 VISA 已是 IVI 套件整体中的一部分，但它们仍是不同的。 <399

如果你打算在你的 GPIB 接口的仪表中使用诸如数字万用表（DMM）和计数器等仪器，你最好熟悉 SCPI。如果你打算充分利用厂商的 VISA 驱动，你也需要了解这些。很快，我们将看到如何通过 Python 分别在 Windows 和 Linux 平台上使用 VISA 驱动。

当然，在某些情况下诸如 SCPI 或 VISA 等标准无法直接使用。在这种情况下可能别无选择，只能尝试使用由制造商提供的接口软件，如果没有就只能自己写一个了。但我要强调一点，写一个设备驱动或者 I/O 处理程序往往不是一个普通的任务，你应该尽可能避免这么做。

IVI-兼容的驱动。根据项目的复杂度和主机上的操作系统，采用类似 IVI 的驱动可能会更有意义，而不是去试着“玩你自己的”API。IVI 基金会标准为很多仪器和接口硬件定义了驱动程序体系。IVI 的设计采用了共享通用功能软件组件的思想，因此其针对不同设备的 API 看起来是十分相似的。IVI 基于 VISA I/O 标准（我们马上会看到），同时也兼用了 SCPI 协议标准。图 11-3 展示了 IVI 体系的整体概貌。 <400

IVI 兼容的软件能提供一些显著的优势，包括状态缓存、多线程驱动、模拟功能、仪器可互换性。IVI 要求使用其标准化的接口来处理不同类型仪器之间的细节，这样就能使系统实现者把重点放在数据处理和显示软件上，而不必再去处理系统中每一种仪器特有的接口代码。在大多数情况下都没问题，但仅限于 IVI 兼容的商用现成技术（COTS）软件。如果你需要用一种未被支持的语言（如 Python）来访问一个仪器，或者使用没有内置 IVI 接口功能的数据采集和分析工具，就需要自己动手使他们能够很好地工作在一起。 <401

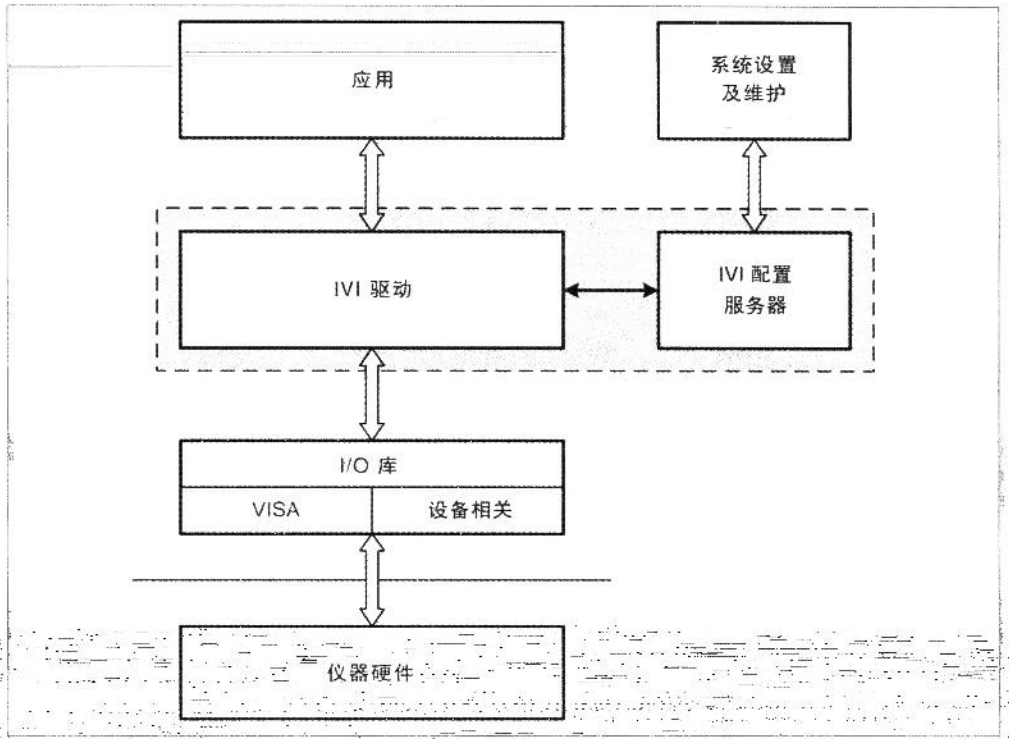


图11-3 IVI架构概览

IVI的一个潜在的缺点是，一个完全IVI兼容（通过IVI认证）的驱动只能在Microsoft Windows平台上使用。这在IVI基金会公布IVI规格说明书的时候就说得很清楚。尽管一些仪器制造商为其能够在Linux上工作的产品制作了“IVI风格”的驱动，但如果你正在各种制造商之间寻找真正的跨平台兼容产品，你可能得考虑到这一点。

VISA——虚拟仪器软件架构

VISA是一种被广泛使用的接口I/O API技术规范，用来和通过GPIB、VXIbus、串口、以太网或者USB类接口连接到PC的仪器进行通信。VISA标准也是IVI套件的一个核心组件。

VISA库定义了一个使用Windows DLL模块的标准化API，通常是*visa32.dll*。VISA同时支持微软组件对象模型（COM）技术。如果应用程序遵循VISA标准，那么来自不同厂商之间的VISA驱动实现通常是可以互换的。

并非所有的仪器都自带 VISA 驱动，一些 VISA 支持可能是购买产品时的额外选项。GPIB 接口的产品通常都会自带 VISA 驱动，而且还有 Linux 版本的，如国家仪器（NI）出售的插卡。安捷伦（Agilent）也出售自带 VISA 组件的 GPIB 接口，而且安捷伦还推荐一个来自第三方资源，在 Linux 下可用的 VISA 接口。

SCPI——可编程仪器的标准命令

SCPI 标准定义了使用可编程仪器时的语法、命令结构和数据格式。SCPI 没有定义确切的物理接口（GPIB、RS-232、USB 等），这意味着它是一个接口无关的标准。SCPI 在 IEEE-488.2 标准之前，两者有些相似，但后者有更多的领域限制。

SCPI 命令是 ASCII 字符串。响应也可能是 ASCII 字符串，尽管在很多情况下它们都是二进制数据（如传输大批量数据时）。SCPI 命令组织在含有一套基本命令集的仪器类中。支持 SCPI 协议的仪器不需要底层的 VISA I/O 功能，只要主系统能够和设备通信就可以（SCPI 是一个命令协议，所以它是接口无关的）。图 11-4 展示了工作于 SCPI 标准的设备模型。SCPI 功能驻留在设备接口的逻辑接口层，正如先前图 11-2 所示。

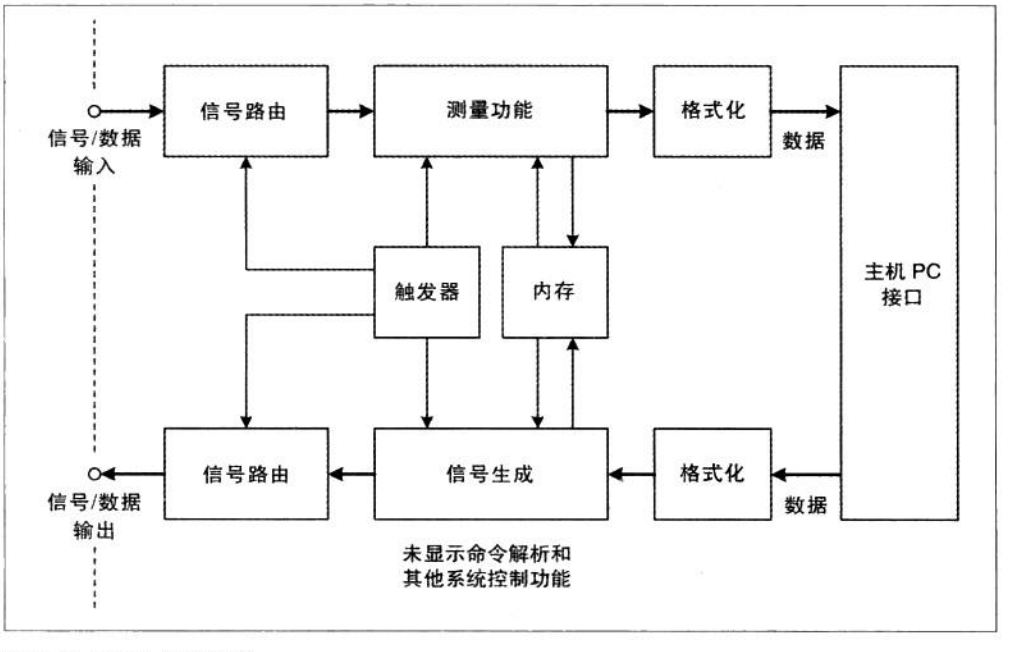


图11-4：SCPI 仪器模型

402 不是所有的设备都会同时用到图 11-4 所示的所有功能。某些类型的仪器可能只是输入型设备，如温度传感器或数字万用表。对于这些仪器而言，可能不会有信号生成部分。还有其他某些类型的光谱分析仪，可能兼有输入和输出功能，因此它们应该包含 SCPI 模型中的所有模块（或至少也包括其中很大一部分）。

指定仪器上的可用命令基于仪器的类型。SCPI 1999 标准定义了 8 个仪器类，其中每个都使用一个特定的 SCPI 命令子集：

- 底盘测功机（Chassis Dynamometers）
- 数字仪表（Digital Meters）
- 数字转换器（Digitizers）
- 发射器（Emissions Benches）
- 发射测试单元（Emission Test Cell）
- 电源（Power Supplies）
- 射频与微波源（RF & Microwave Sources）
- 信号转换器（Signal Switchers）

403 上述术语中有些可能不是特别直观。当说到 SCPI 时，数字转换器是一个用来测量电压波形的设备——换言之，就是一个示波器或逻辑分析仪。信号转换器是一个用来控制信号在某种路由或者交换网络中路径的仪器。这种仪器简单的只有一个开关，而复杂的，可能包含有一个多路输入输出的选择矩阵。一些仪器制造商生产的设备，兼有信号交换功能和可选的数据采集或控制功能。安捷伦 34970A 数据采集交换单元（Agilent 34970A Data Acquisition Switch Unit）和吉时利 3706 系统开关 / 万用表（Keithley 3706 System Switch/Multimeter）就是这类仪器。

SCPI 被组织成相互关联的指令组。一个组由根和命令组成，而且每个根命令都有一些可选的参数。命令组的一个例子是 `MEASure` 命令。（SCPI 允许用大写字母表示命令缩写，如 `MEASure` 可以被写做 `MEAS`。）

图 11-5 显示了简化的 `MEASure` 命令树图，这个命令可能在数字万用表中被用到。要创建一个命令字符串，由 `MEASure` 开始从树的左边向右边移动，挑选必要的参数关键字。

下面是一个 SCPI 命令的例子，数字万用表中会用到这样的命令，如带有 GPIB 接口的 Agilent 34405A DMM（见图 6-4）：

```
MEASure:VOLTage:DC?
```

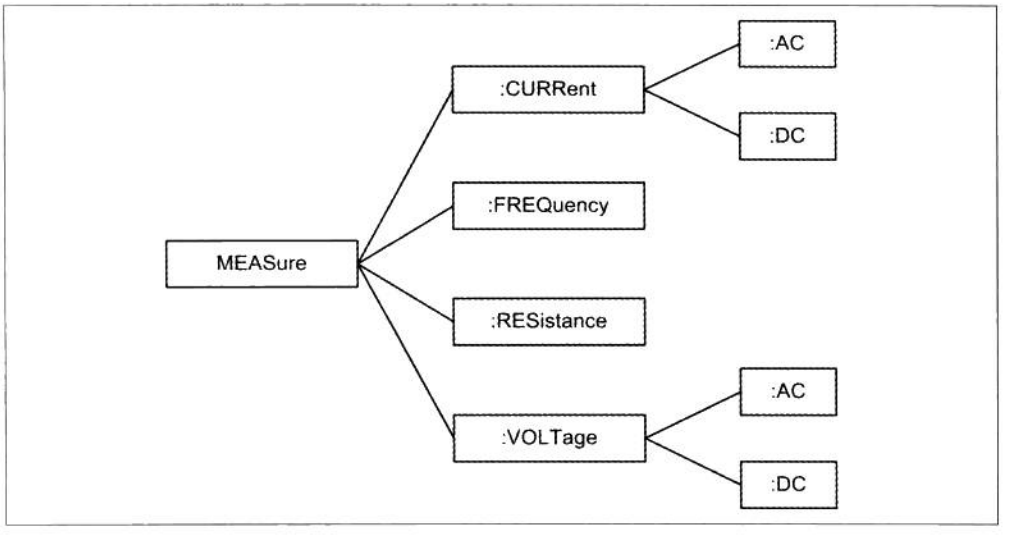


图11-5：SCPI MEASure 命令树实例

假设到 GPIB 端口或接口设备的访问已经被建立，且仪器已被正确初始化，这条指令会命令一个直流测量仪器使用任何适当的自动调节，并返回结果（隐含在命令字符串的问号后）。 404

下面是一个在 Agilent 34405A DMM 上的备选方案，用命令序列来设置 DC 输入范围并进行测量：

```
CONFigure:VOLTage:DC 1, 0.0001
TRIGger:SOURce IMMediate
INITiate
FETCh?
```

这个命令序列配置一个 DC，使之输入范围为 1V 且分辨率是 0.1mV，并将测量触发模式设置为即时模式（immediate）。INITiate 命令让仪器处于即时模式下的“等待触发”的状态，仪器开始连续读数。FETCh 命令返回最新读取到的电压值。

像先前提到的那样，SCPI 命令可以使用缩写。前面的命令序列可以缩写成下面的方式：

```
MEAS:VOLT:DC?
```

或者：

```
CONF:VOLT:DC 1, 0.0001
TRIG:SOUR IMM
```

```
INIT
FETC?
```

对整个 SCPI 规格说明书的描述超出了本书的范围。要获得更多的信息，可参考本章最后的“推荐阅读”一节。此外，你还需参考一下每台仪器附带的编程文档，以明确它是如何实现 SCPI 的。尽管很多仪器制造商都遵循 SCPI 标准，但他们也可能因某些特性而进行一些调整。

特定协议

一些设备（如一些低成本的数字万用表）使用在个别模式下特定的命令和数据协议。例如，tpi 183 在仪表侧面有一个 3.5mm 的插孔，并连续输出 1200 波特的 RS-232 数据流。数据类型由仪表手动控制设置决定（一个大型旋钮开关）——从串口没有办法去设置它。输出是一个 ASCII 字符串，格式被定义成 FAR DDDDDDT 的样子。F 位置的 ASCII 字符是功能码（0 到 B），A 是手动或自动范围模式（0 或 1），而 R 是范围码（0 到 5）。其次是一个空格字符和 6 个用来表示浮点数据的 ASCII 字符（即字符串格式的 DDDDDD 部分）。字符 T 表示输出字符串的结束。

405 虽然此接口可能只是针对 tpi 183 的，但这绝不是一个唯一的例子。很多设备——特别是老式设备，都使用独特的接口协议。甚至一些当代的 USB 类型的设备都有它们自己的特定命令和数据接口协议。

另一个例子是连接到 RS-485 总线上的设备所使用的命令和相应协议。指定一个设备作为主设备（一般是 PC 主机）是一种常见的情况，而其他设备只在接收到指明发送给它们的命令时才会做出响应。在这种情况下，设备标示符必须被包含在 RS-485 总线上的每个命令中，以提醒相关设备某个命令是发送给它的。其他设备也会接收到此命令，但它们不会做出回应。图 11-6 所示为一个命令和响应消息格式的例子。

406 图 11-6 中的响应总是以 /0 开始，因为在这样一套设计中，主控的设备 ID 几乎总是 0。既然只有一位 ASCII 数字可以用来表示设备 ID 号，这样的协议也就仅能支持最多 15 个独立的设备，地址从 1 到 F。从图 11-6 中我们还能推算出最多有 256 种可能的命令或者响应码（假设以十六进制数表示，即从 00 到 FF）。发送的命令和参数数据，以及返回的响应数据长度都是可变的，由命令的类型定义。命令消息由一个 ! 字符结束，而每个响应则由一个 # 字符结束。要注意，这只是一个可能的协议，尽管它出自于市场上已有的产品。命令和响应消息最终是什么样子，还是取决于设计此产品的工程师。

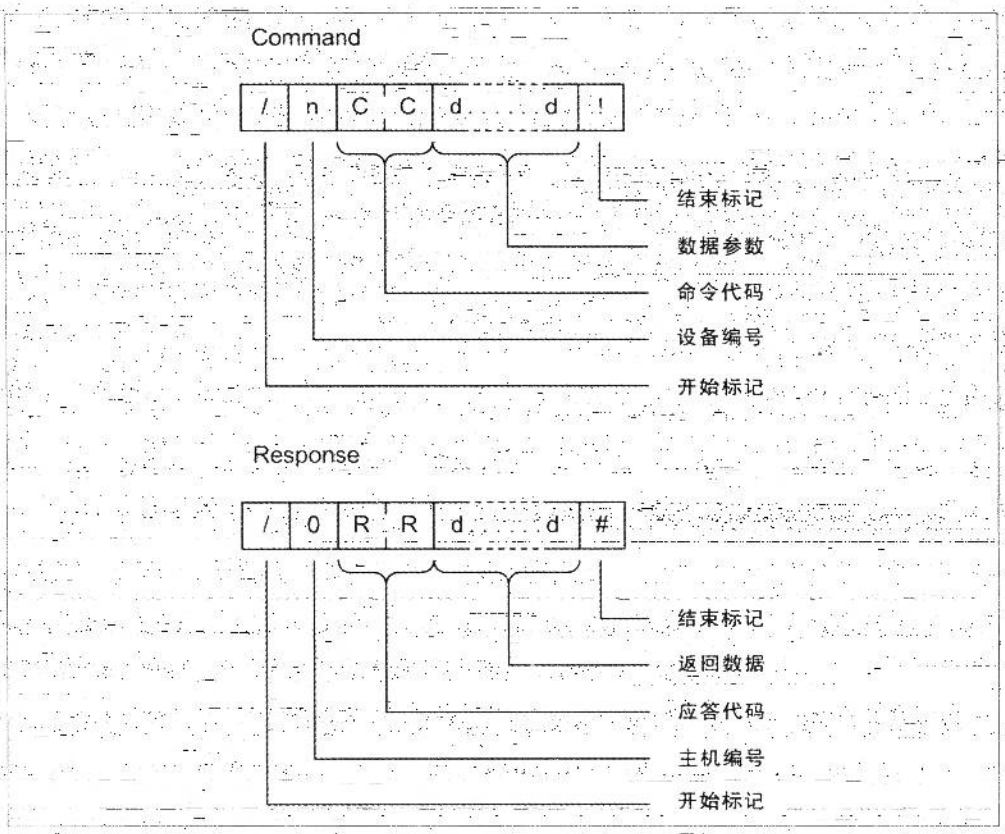


图11-6：命令及回应格式

Python 接口支持的工具包

Python 中有一些 I/O 工具包能够用来在 Python 应用中实现各种类型的接口。这些工具有串口、并行端口 I/O、USB，以及包括 GPIB 的 VISA 类型的仪器接口。在这一节中，我们将快速浏览这三种不同的工具包，所有这些工具包都能够很容易地移植到不同平台上（主要是 Windows 和 Linux）。

pySerial

pySerial 工具包由 Chris Liechti 开发并维护，封装了一个 Python 程序通过串口进行通信时所需的必要功能。你可以从 *http://pyserial.sourceforge.net* 下载。*pySerial* 会根据主机所用的操作系统，自动选择合适的后端（物理接口和系统驱动），而且它支持 Windows、Linux、BSD、Jython，以及 IronPython 环境。

Serial 类在不同平台上提供了同一套必要功能的方法，安装后使用起来很直观。假设某设备通过串口连接，并能显示从 Python 中写入的数据，那么向其发送一个字符串就像下面这么简单：

```
>>> import serial
>>> sport = serial.Serial(0)          # 开启串行端口
>>> print sport.portstr               # 打印端口字串
>>> sport.write("Port opend\r\n")     # 写入包含回车以及换行符的字串
```

当这些代码从 Python 提示符中执行后，会有如下响应：

```
COM1
```

（当然得假设使用了 COM1），而在这个连接的另一端你将会看到：

```
Port Opened
```

407

当用完串口后，我们可以优雅地将之关闭（需要时还可以被重新打开）：

```
>>> sport.close()                     # 关闭端口
```

pySerial 还支持各种端口配置参数和超时设置，而且提供了诸如 `read()`、`wirte()` 和 `readline()` 等方法。

如果没有可用的串口，如笔记本电脑或者上网本电脑，你将会看到像这样一个错误回溯：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c:\APython26\lib\site-packages\serial\serialwin32.py", line 30,
    in __init__
    SerialBase.__init__(self, *args, **kwargs)
  File "c:\APython26\lib\site-packages\serial\serialutil.py", line 201,
    in __init__
    self.open()
s File "c:\APython26\lib\site-packages\serial\serialwin32.py", line 56,
    in open
    raise SerialException("could not open port %s: %s" % (self.portstr,
    ctypes.WinError()))
serial.serialutil.SerialException: could not open port COM1: [Error 2]
The system cannot find the file specified.
```

通过不向 Serial 对象的初始化方法中传入参数，你还可以很容易地创建一个没有被打开的串口实例：

```
>>> import serial
>>> sport = serial.Serial()
>>> sport.baudrate = 19200
>>> sport.port = 0
```

由于串口对象已被实例化，还定义了一些基本的参数，因此它能够按需要被操作或者关闭。isOpen() 方法用来检查串口状态：

```
>>> sport.open()
>>> sport.isOpen()
True
>>> sport.close()
>>> sport.isOpen()
False
```

表 11-1 到表 11-6 提供了 *pySerial* 中一些方法的摘要，按功能分类组织。在绝大多数情况下，你的需求不会超出这些内容，但 *pySerial* 还是为处理串行通信提供了一套相当全面的方法。甚至连 RS-232 中的一些非常晦涩难懂的功能都支持。

408

表11-1：pySerial端口打开/关闭方法

方法	描述
open()	使用当前的设置打开（或者重新打开）端口
close()	关闭端口但不销毁端口对象。它可以被重新打开
isOpen()	如果端口已经打开就返回 True，否则返回 False

表11-2：pySerial端口读/写方法

方法	描述
read(*size=1*)	从串口读取 size 个字节。如果设置了超时，它可能返回少于所请求的字节数。如果没有超时处理，这个方法会阻塞直到读取到所请求的字节数
readline(*size=None, eol='n'*)	读一个字符串直到接收到一个行尾（eol）符号（默认为一个 \n）或读超时发生
readlines(*sizehint=None, eol='n'*)	读取多行直到读超时发生。参数 sizehint 被忽略
write(*data*)	通过串口输出给定字符串
writelines(*lines*)	向串口写入多行字符串

表11-3：pySerial数据缓冲管理方法

方法	描述
flushInput()	清空输入缓冲上的数据，丢弃当前缓冲区中所有的数据
flushOutput()	清空输出缓冲上的数据，终止当前输出并丢弃当前缓冲区中所保留的数据
inWaiting()	返回当前输入缓冲区中的字符个数
outWaiting()	返回当前输出缓冲区中的字符个数

表11-4：pySerial端口参数方法

方法	描述
getBaudrate()	返回当前设置的波特率
setBaudrate(*baudrate*)	设置端口波特率。这个方法不能在端口已经打开后使用
getByteSize()	返回当前设置的字节大小
setByteSize(*bytesize*)	设置数据字节位长
getDsrDtr()	返回当前的 DSR/DTR 流控制设置
setDsrDtr(*dsrdtr=None*)	设置 DSR/DTR 流控制行为
getParity()	返回当前的奇偶性设置
setParity(*parity*)	设置端口的奇偶性
getPort()	返回当前端口设置
setPort(*port*)	设置端口号或者端口名称
getRtsCts()	返回当前 RTS/CTS 流控制设置
setRtsCts(*rtscts*)	设置 RTS/CTS 流控制行为
getStopbits()	返回当前设置的停止位
setStopbits(*stopbits*)	设置停止位使用的位数
getTimeout()	返回当前超时设置
setTimeout(*timeout*)	设置读超时时间
getWriteTimeout()	返回当前写超时设置
setWriteTimeout(*timeout*)	设置写超时时间
getXonXoff()	返回当前 XON/XOFF 设置
setXonXoff(*xonxoff*)	设置 XON/XOFF 流控制行为

409

表11-5：pySerial端口性能方法

方法	描述
getSupportedBaudrates()	返回串口所支持的波特率列表
getSupportedByteSizes()	返回串口所支持的字节大小列表
getSupportedParities()	返回串口所支持的奇偶位设置列表
getSupportedStopbits()	返回串口所支持的停止位列表

表11-6：pySerial硬件握手状态方法

方法	描述
getCD()	返回载波检测（Carrier Detect）状态
getCTS()	返回清除发送（Clear To Send）状态
getDSR()	返回数据装置就绪（Data Set Ready）状态
getRI()	返回振铃指示（Ring Indicator）状态
setDTR(*level=1*)	设置数据终端就绪（Data Terminal Ready）到指定的状态
setRTS(*level=1*)	设置请求发送（Request To Send）到指定的状态

pySerial 不直接支持 RS-485 接口，但它在具有半双工自动转换功能的 RS-232-to-RS-485 转换器上能够正常工作。而且，通过虚拟串口（Windows 平台）或者一个 */dev* 目录下 *tty* 类型的设备实体（Linux 平台），它还能和 USB-to-RS-485 转换器一起工作。*pySerial* 包还提供了一个处于试验阶段的 RFC 2217 服务器的实现。

对于安装和额外的使用说明，请参阅 *pySerial* 网站。

pyParallel

410

pyParallel（*http://pyserial.sourceforge.net/pyparallel.html*）是 *pySerial* 的姊妹项目，由同一个开发者创建。*pyParallel* 的目的是封装平台无关的 Python 程序对并行端口的访问（可回顾第 2 章关于并行端口的讨论）。目前，它仅支持 Windows 和 Linux。

与 *pySerial* 不同的是，*pyParallel* 没有打开（open）和关闭（close）方法。如果实例化时没有指定端口参数，*pyParallel* 将尝试使用第一个可用的并行端口。除此之外，还可以以字符串的形式指定一个特定端口的名字。

下面是一个简单的例子：

```
>>> import parallel
>>> pport = parallel.Parallel()      # 开启第一个可用的并口
>>> pport.setData(0x55)
>>> pport.setData(0xAA)
```

这个例子会向并行端口中写入值 0x55，然后紧接着再写入一个 0xAA。表 11-7 罗列了此可用的方法。

表11-7：pyParallel 方法

方法	描述
setData(*value*)	向并行端口的数据引脚写入一个字节的值
setDataStrobe(*level*)	设置数据选通（Data Strobe）为 *level*（0 或 1）
setAutoFeed(*level*)	设置自动换行（Auto Feed）为 *level*（0 或 1）
setInitOut(*level*)	设置初始化为 *level*（0 或 1）
getInSelected()	读取选择（Select）的状态
getInPaperOut()	读取缺纸（Paper Out）的状态
getInAcknowledge()	读取确认（Acknowledge）的状态

注意，*pyParallel* 并没有提供读取繁忙或者输入错误的功能，但 *pyParallel* 却允许你直接操作端口上的握手输出引脚（handshaking output lines）。

在 Windows 机器上，*pyParallel* 需要直接访问物理端口的硬件。它不能和 USB 并行端口适配器一起使用，所以在仅有 USB 端口的笔记本或上网本上无法工作。此外，它也不支

持扩展并行端口（Extended Parallel Port，EEP）的功能。

通过 *pyParallel* 向一个外设发送数据时，必须在软件控制下每次输出一个字节。这听起来可能显得笨拙，但除非使用内置流量控制管理和内部缓冲功能的智能硬件，否则确实没有其他办法了。程序与打印机通信时，必须至少设置和清除数据选通（Data Strobe）引脚，并检查从打印机返回的应答（Acknowledge）引脚。

411 PC 上的并行端口并不仅限于向打印机发送数据。下面是一个有趣的例子，展示了如何从并行端口驱动一个 LCD 显示：*http://pyserial.svn.sourceforge.net/viewvc/pyserial/trunk/pyparallel/examples/*。

还有其他一些有趣的应用，如并行端口控制 DAC 设备，感应离散的数字输入，以及使用 8 位输出控制继电器或其他设备。不足之处是，端口电路的设计不能承受很大的电流，所以通常还会需要一些外部接口电路。

PyVISA

PyVISA 工具包为 Windows 上的 IVI 标准的 VISA 驱动和 Linux 系统上的 IVI 兼容 VISA 驱动提供了一个 Python 的 API，它使用一个驱动 DLL 或者一个设备供应商提供的库文件。在 Windows 机器上，这个工具包通常会在 *C:\WINNT\system32* 目录下寻找名为 *visa32.dll* 的一个动态链接库（DLL）。而在 Linux 系统上，美国国家仪器公司（National Instruments，NI）会以一个名为 *libvisa.so.7* 的共享库（在 Linux 上等价于 Windows 系统的 DLL）的方式提供一个 IVI 兼容的 VISA 驱动。这个文件通常会被放在 */usr/local/vxipnp/linux/bin* 目录下。

NI Linux 版本的 *visa32* 对下列发行版均支持良好：

- Red Hat Enterprise Linux Workstation 4
- Red Hat Enterprise Linux Desktop + Workstation 5
- SUSE Linux 10.1
- openSUSE 10.2
- Mandriva Linux 2006
- Mandriva Linux 2007

请参考本章最后“推荐阅读”一节以便了解更多关于 NI VISA 驱动的信息。

在 Windows 上，不需要花太多时间去寻找你所需要的资源。现在具有 IVI 兼容接口功能的仪器基本上都自带了 Windows 平台的 VISA 驱动，所以你应该先找一下随设备一起提供的原始 CD，或者从一个设备供应商的主页上下载相应的 VISA 组件。

VISA 通过扩展的 *PyVISA* 可以支持串口、GPIB、GPIB-VXI、VXI、TCP/IP，以及 USB 接口。我们先使用 VISA 与带有 GPIB 的设备进行交互。下面是一个简单的例子，用于展示 *PyVISA* 如何使用：

```
>>> import visa
>>> dmm = visa.instrument("GPIB::2")
>>> print dmm.ask("*IDN?")
```

这段代码告诉 VISA 驱动我们想要通过对象 dmm 使用仪器的 GPIB 地址 2。dmm.ask 方法发送指定的字符串（这里是 "IDN?"），它会返回仪器响应的一个该设备内部的标识字符串。 412

你可以从 *PyVISA* 项目的主页上获取更多信息：*http://pyvisa.sourceforge.net*。

VISA 提供了太多不同的功能，我们无法在这里全都提到。要了解更多 VISA 自身的细节，可以参考来自 IVI 基金会的 VPP-4.3 VISA 库文档。

Windows 平台上的替代品

有一个叫做 *PyUniversalLibrary* 的开源（OSS）项目正在对 Measurement Computing 公司的通用库 API（Universal Library API）进行封装。据其主页所示，它还没有全部完成，但是已经有足够的功能可用。你可以在这里获取更多信息：*https://code.astraw.com/projects/PyUniversalLibrary*。

UNC Python 工具包中还包含了一个对美国国家仪器公司的老式 NI-DAQ 驱动的封装，详见：*http://sourceforge.net/projects/uncpythontools/*。

在 Linux 下使用基于总线的硬件 I/O 设备

将一个设备接入 Windows 机器通常比较容易，供应商一般都会提供相应产品的接口驱动。对于 Linux，使用串口、GPIB 或者 USB 端口通信的设备一般不会有什么大问题。但是，如果有插卡连接到 PC 内部的 PCI 总线，情况就比较复杂了。我们已经在第 5 章中看到过，可以通过扩展的方式，允许它作为一个 Windows 下的 DLL 封装来提供服务，并访问一个连接到内部总线上的设备。在 Linux 中，每个 I/O 设备都需要一个专门为 Linux 环境编写的驱动，以及一些用于配置设备内部参数的工具。有不少仪器制造商都不直接支持 Linux。这通常不是有意为之，只是（当时）在仪器应用领域没有多少使用 Linux 系统的，这就无法证明为同时支持两个版本的接口软件而付出的努力和开销是否值得。不过，有一个叫做 Comedi 的项目提供了一些方法，能够将仪器接口硬件连接到 Linux 主机上。

Comedi 项目

Comedi 项目由 David Schleef 于 1996 年发起，是一个允许 Linux 系统和与各种数据采集或数字接口硬件进行通信的底层驱动的集合。它是一个开源项目，其主页是 *http://www.comedi.org*。

413 *comedi* 工具包是三个互补的软件组件的组合。第一个组件是一个通用的设备无关的 API，它能够与一系列内核模块进行交互，这些模块为通用 API 提供支持（第二个组件），最后，还有一个为配置各种卡提供接口的函式库（第三个组件）。Comedi 团队（尽可能地）和硬件供应商协同工作来搜集一些信息，获得硬件用来测试和校验，并开发驱动。从某种意义上讲，Comedi 可以说是 Linux 下 IVI 套件的驱动。

如果你想自己下载并构建 Comedi，要确保 *comedi* 和 *comedilib* 两个工具包都被下载。此外，你可能还会需要 *comedi_examples* 文件。

Comedi 硬件支持

Comedi 大概支持下面的接口硬件厂商：

- ADLink
- Advantech
- Amplicon
- Analog Devices
- ComputerBoards
- Contec
- Data Translation
- Fastwel
- General Standards Corporation
- ICP
- Inova
- Intelligent Instrumentation
- IOTech
- ITL
- JR3
- Keithley Metrabyte
- Kolter Electronic
- Measurement Computing
- Mechatronic Systems, Inc.

- Meilhaus
- Micro/sys
- Motorola
- National Instruments
- Quanser Consulting
- Quatech
- Real Time Devices
- Sensoray
- SSV Embedded Systems
- Winsystems

并不是来自所有公司的所有设备都能够被 Comedi 所支持，但 Comedi 涵盖面还是很广的，已经支持超过 400 种设备（而且还在增加）。读者可以从 Comedi 的网站找到完整的支持列表。

通过 Python 使用 Comedi

comedi 可以通过 Simplified Wrapper and Interface Generator（SWIG）产生一个 *comedilib* 的封装。读者可以从 *http://www.swig.org* 获取更多有关 SWIG 的文档。在 Google 上还有一个讨论组，如果遇到 *comedi* 方面的问题可以去那儿寻求帮助。

数据 I/O：数据采集与写入

现在，我们已经初步了解了当需要和仪器硬件进行交互时都需要哪些软件，接下来让我们掀开它们神秘的面纱，看看它们是如何为我们工作的。

基本数据 I/O

就数据采集而言，基本有两种数据源：外部仪器和计算机自身内置的数据采集硬件。这两种数据源都会与你的应用软件进行数据传输。有时候传输是直接的，如从设备底层代码直接访问设备的硬件寄存器。这种风格的接口编程现在已经很少见了，因为底层操作系统趋于禁止使用用户级代码直接访问硬件。在大多数情况下，会涉及一个中间件，如一个驱动或者一个厂商定义的 API（回顾第 5 章），或者一个接口库（如 *pySerial*）。

从一个外设采集数据，或发送（如发送一个命令）数据到另一个设备，有多种实现方法。如果你想发送数据，首选的最直观的方法就是将数据写入端口或者设备，让它发送。如果想读取数据，最直接的方法是简单地读取需要的数据。

这两种方法都假定当设备发送一个命令或请求一个数据时，它会自动并实时地执行一些
415 必要的硬件功能，将数据转换成一个内部寄存器地址、内部命令代码或者返回值。在大多数情况下，这一假定是有效的。但是还有一些情况它不会按照预想的方式工作。例如，进行一个写操作时可能会发生一个错误，或者设备驱动 API 函式可能阻塞一段时间才返回，或根本就不返回。

读取数据

当从一个总线设备读取数据时，设备接口通常会返回一个能够被立即使用的二进制数。你只需调用一个函式，而不用发送命令。此外，命令和数据通常都使用一个 ASCII 字符串的命令 - 应答格式，ASCII 到二进制的转换在 Python 中很容易处理。

使用 SCPI 的仪器总会返回一些字符串，其中包含一个或多个用逗号分隔的数值。幸运的是，Python 能够很容易地处理字符串格式的数字。假如我们有一个仪器，在进行尺度测量时，会返回 "+4.85510000E-01" 这样的数据，并假定下列代码段中的函式 getDataResponse() 会返回一个仪器响应的字符串或者 None。那么，就可以使用 Python 的浮点类型对象的构造函式做必要的转换：

```
raw_data = getDataResponse(instID)
if raw_data:
    data_val = float(raw_data)
```

如果 raw_data 不是 None，data_val 的值会像我们所期望的那样是 0.48851。

下面让我们来看另一个例子，这次测量命令包含了两个以上的返回值。假设一个仪器在查询时返回了 4 个值，放到了字符串 "+5.50500000E00,-2.66000000E-01,+8.24000000E01,-6.34370000E00" 中。在 Python 中，可以很容易地将一个用逗号分隔的字符串转换成一个字符串列表。下面的代码段可以用来处理这种情况：

```
data_val = []
data_str = getDataSet(instID)
raw_vals = data_str.split(",")
for raw_data in raw_vals:
    data_val.append(float(raw_data))
```

执行过这段代码之后，列表变量 data_val 会包含 4 个浮点数值：

```
[5.5049999999999999, -0.26600000000000001, 82.400000000000006,
-6.3437000000000001]
```

由于使用了 Python 的字符串 - 浮点转换方法，这些数字看上去有点怪，但其本质上与原字符串中的数据是一致的。结果看上去奇怪是因为浮点值由 CPU 控制（Python 默认情况下不会使之更优雅，除非你指定它这么做）。

如果你从某仪器中采集到的 ASCII 字符串数据包含数字之外的其他字符，那你可能需要手动地做一些解析，以提取字符串中你所关心的特定区段。这些 ASCII 数据还需要被转换成某种二进制格式。我们之前看到的 RS-485 接口就属于这种情形。 416

某些情况下，可能会返回一个混有数字和非数字字符且无固定格式的字符串。我们之前介绍的 tpi 183 DMM 生成固定格式的数据。这样很好处理，你只需从字符串中提取字符串切片（可参见第 3 章了解更多有关 Python 切片的知识）。然而，并非所有的情况都如此。有时，返回的字符串数据的长度，甚至主要的头部字节都会变化。

如果你正在处理的仪器或设备采用有固定起始位置和结束位置的数据格式，你可以通过切片来提取你需要的数据。如果开始位置不固定，那你就需要在使用切片提取数据之前遍历字符串，找出数据的起始位置。

写入数据

之前提到过，要访问一个总线设备，通常只需调用此设备的 API。它本身不需要什么命令，但通常会将参数值写入设备，或者调用一个函式来启动或停止一些行为（如定时器或者时钟功能）。

向一个利用 SCPI 的外部设备写入 ASCII 数据（如命令和参数值）或一个特定命令格式时，需要创建一个字符串命令，对设备进行写入操作，并且等待设备响应。在这种命令 - 应答（command-response）情景中，设备只有在收到请求时才会返回数据，它不会自己主动发送数据。有时甚至会不响应返回请求。

举例来说，假定我们有一个 GPIB 设备，如一个可编程电源。这个例子基于 Agilent E364xA 系列，该系列包含一些非 SCPI 命令，我们在此不做讨论。现在，我们只使用下面的命令：

```
OUTPut
    [:STATE] {ON|OFF}
    [:STATE]?

[SOURce:]
    CURRent
    CURRent?
    VOLTage
    VOLTage?

MEASure
    :CURRent?
    [:VOLTage]?
```

ON|OFF 参数外面的大括号表示待选择。同时，还有一些项目被置于中括号中，包括一个 417

关键命令 SOURce。这表示这些内容是可选的，中括号里的参数是默认值。因此，对于一个 MEASure 命令，如果命令像这样给出：

```
MEAS?
```

它将会返回电源的输出电压。要获得当前值，必须显式地指定：

```
MEAS:CURR?
```

由于 SOURce 命令是可选的，下面一组命令会将输出电压设置成 5.1 伏直流电（DC），并把当前电流限制在 1.0A：

```
VOLT 5.1
CURR 1.0
```

如果愿意，还可以使用 OUTPut 命令控制电源输出，如：

```
OUTP:OFF
SOUR:VOLT 5.1
SOUR:CURR 1.0
OUTP:ON
```

在修改 V 和 I 参数之前，这个操作会中断输出。

要取回当前设置，可以使用 SOURce:VOLTage 或 SOURce:CURRent 命令，并使用一个问号来表示查询：

```
VOLT?
```

它会返回一个类似这样的响应结果：

```
5.00000
```

要在输出终端上监视当前正在进行的工作，可以使用 MEASure 命令，如下：

```
MEAS:CURR?
```

它会返回（示例）：

```
0.20000
```

这个命令会读取输出的电压：

```
MEAS?
```

并返回：

```
5.00000
```

最后，我们可以通过查询格式的 OUTPut 命令来检查电源输出是否可用：

```
OUTP?
```

如果电源可用，它会返回一个 ASCII 的“1”，否则就返回“0”。

418

我没有说明这些命令是如何发送以及响应是如何返回的，因为这些操作有多种实现方式，如串口、USB 或者 GPIB。但我们可以设想有一个名为 sendCommand() 的函式，它会帮我们处理好这一切。在这个例子中，我们将实现这个 setPowerSupply() 函式，它接收两个参数，volts 和 current，并将其发送至指定设备：

```
def setPowerSupply(volts, current):
    rc = OK
    volts_str = "%2.2f" % float(volts)
    current_str = "2.2f" % float(current)

    cmd_str = "VOLT " + volts_str
    rc = sendCommand(instID, cmd_str)
    if rc == OK:
        cmd_str = "CURR ", current_str
        rc = sendCommand(instID, cmd_str)

return rc
```

这看起来更直接一些，但是其中也有一些东西可能并不是特别容易看懂。

rc（返回码）被预置为 OK（乐观一些），输入的参数 volts 和 current 被转换为字符串表示。注意，此处的格式被指定为 %2.2f。这样创建出来的字符串更容易被设备控制。同时，输入的参数被用来创建浮点类型的变量对象，并嵌入到字符串变量里。如果传入的参数是浮点型的，就不会改变什么，但如果参数是整型的，它们会被转换。而且，这个函式的两个参数也能够接收整型或者浮点型数据的字符串表示。

顺便说一句，这是一个非常便利和强大的技巧。它几乎可以处理你所能传入的任何可用格式的数值，而且优雅地将其转换为一个浮点类型。当传入的参数是一个非数字的字符串、一个字符串格式的十六进制的数值、一个 *n* 元组（*n*-tuple）或者一个字典（dictionary）时，它会执行失败并抛出异常，但它很容易被 try-except 捕获和处理。

接下来，sendCommand() 函式被调用了。它可能使用 GPIB，或者访问一个串口 I/O。只要仪器能接收到指令，数据到底是如何发送的，还真就不是多么重要了。

现在，我们已经知道如何发送指令，那么，怎么判断该仪器是否已经收到指令并按指令进行工作呢？对于电源的情形，我们主要关心其输出的状态和输出的实际值。检测输出

状态（On 或 Off）非常直接，正如我们所见到的那样，但要测定出输出级别是否等于或者接近指定的数值，就多少有些挑战性了。

419 主要原因是我们要从数字转到模拟，而模拟世界充满了微妙的差异。依赖于仪器的精度，命令设备产生 5.000V DC（直流）输出，输出端的实际电压可能只有 4.999V，或是误差范围内的任意值。下面的代码片段展示了一种通过返回值来检测误差上下界的实现方法：

```
def testDelta(testval, targval, tolerance=0.001):
    testval_float = float(testval)
    targmax = float(targval) * (1 + tolerance)
    targmin = float(targval) * (1 - tolerance)
    if (testval_float >= targmax) or (testval_float <= targmin):
        return False
    else:
        return True
```

与之前的例子一样，浮点对象转换在此处也被用到。这可以保证其内部的所有变量都是浮点型的。

如果传入 `testDelta()` 的值在某个目标值（`targval`）的 +/− 范围内，这个函式会返回 `True`，否则返回 `False`。误差范围在目标值两侧是对称的，如图 11-7 所示。

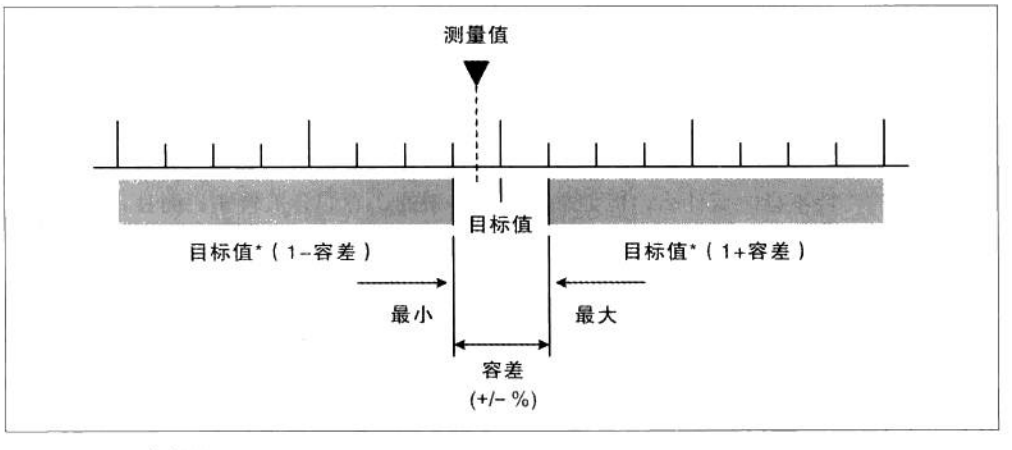


图11-7：数据容差检测

如果我们想获取一个不对称的误差范围，可以对目标值指定一个偏移量。偏移量可以移动到原误差范围内的任何位置，从而获得所需的误差范围。

图 11-8 展示了设置并读取仪器（如电源或其他类型的模拟输出型设备）输出，然后将返
420 回值与初始命令进行对比的功能。就系统本身而言，只要检测实际输出值，然后在错误

发生时报错就足够了，但是，我们还会做一点额外的工作，如尝试重发命令，或者进行系统初始化。

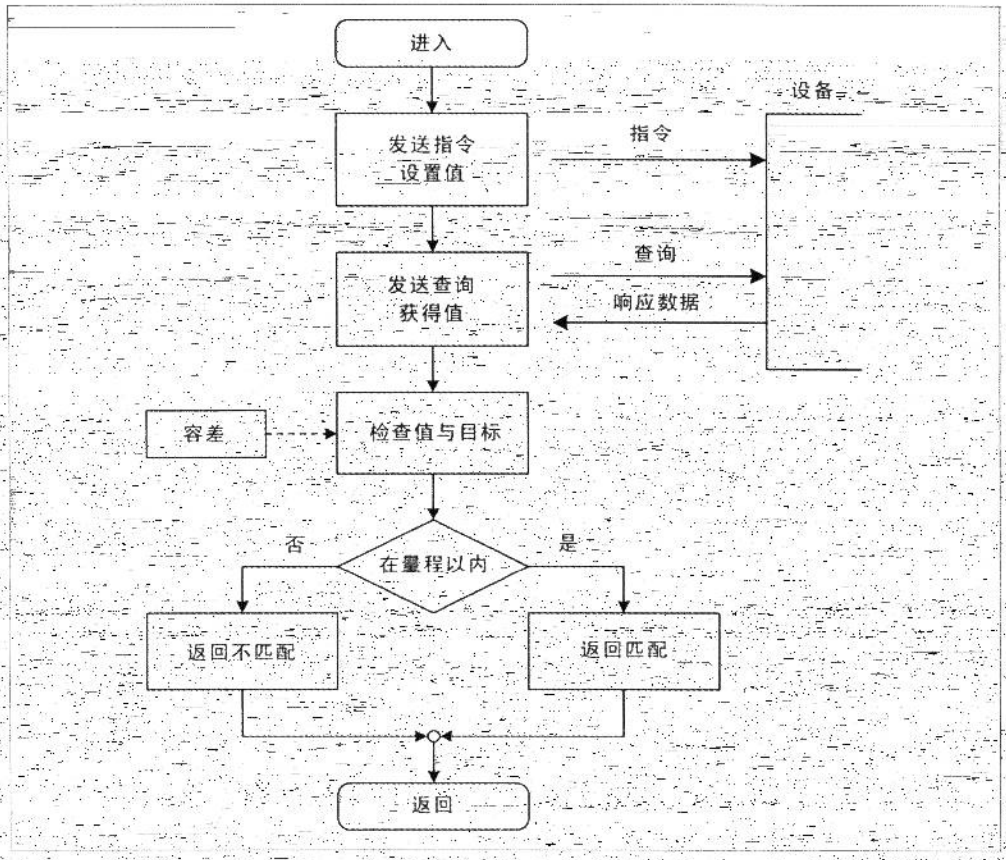


图11-8：配置并检查仪器输出

我们可能对此会感到惊奇："为什么这么麻烦？"好问题！下面这个例子就回答你这个问题。

你可以设置一个可编程电源的最大电流和输出电压。如果电流超过可编程限制的范围会怎样呢？一些电源会被配置成"恒流（constant current）"模式。此时，电源输出的电流会尽量和预设的上限保持一致，甚至当负载接近短路时，电压开始下降至零。如果设置输出电压超过上限，而电流还在负载允许的范围内也是一样的。

例如，如果限定1.0A电流和5.0V直流输出电压，且有一10Ω电阻加载到电源上，那么实际的电流会限制在500mA。电流达到500mA时，电压会保持在5.0V，它同样也满足

421

1A 的限制。如果将接入的电阻减到 4Ω，电源不能输出超过 1A 的电流（之前设定的上限），所以，它最多提供 4V 直流电压，即电流到达 1A 时的电压。这是根据第 2 章所讲到的欧姆定律简单地算出来的。因此，如果设置电流达到或接近最大值，而且你可能希望电源能够有机会在系统中达到这个值，可以通过测量电压来检测类似问题。在这种情况下，控制软件可能会在设置检测电压出错时立即发送一个 OUTP OFF 命令。

阻塞和非阻塞调用

现在来介绍一些概念，它们可以帮助你构建健壮且可靠的应用。我们会先讨论一些阻塞和非阻塞的函式调用，然后再看一些基本的出错控制技术。

有一种根据函式被调用后返回的速度来描述函式的方法或功能。一些函式只有当得到某些结果时才会返回，另一些则立即返回，而不会去等待任何特殊响应。也就是说，函式可能是阻塞式（调用的代码会等待一个响应），或者非阻塞式（调用立即返回，通常会有一个表示成功或失败的返回值）的。

事实上，所有软件的函式（还有方法）都能够按阻塞式或非阻塞式分类，而且一般应用中的大多数函式都是阻塞式的——即它们直到特定行为结束或出错时才会返回。你可以从图 11-9 中的消息序列图（MSC）中看到这一点。我们使用 Function1() 调用 Function2()，Function2() 又依次调用了 Function3() 和 Function4()。Function1() 从 Function2() 获取结果的时间，依赖于 Function 2,3,4 完成它们各自工作所需的时间。整个过程中，Function1() 会被阻塞。（在一个 MSC 图中，函式或过程中的事件按照从上到下的顺序发生，函式或过程之间的处理用水平线表示。）

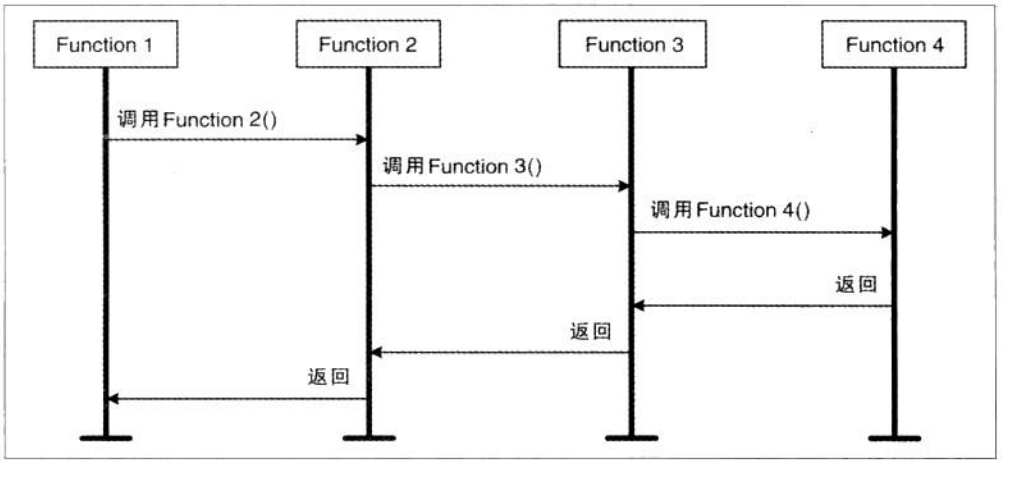


图11-9：功能模块

阻塞允许函式保持同步，并按照代码中预期的流程执行。在调用发生时，它所请求的其他过程或数据未必是有效的，因此，为了响应这些处理，在返回到该调用之前，函式会阻塞等待其他过程结束。由此所产生的一个副作用是，你的应用会被强行挂起，直到函式返回。

我们最感兴趣的一种阻塞是，一个应用必须等待某个接口函式，而该接口函式又等待一个硬件设备的响应，如图 11-10 所示。请注意，图中有一个计时器符号。它表示如果硬件在一个预定的时间内没有响应，那么该接口函式将会终止，并返回一个错误。 422

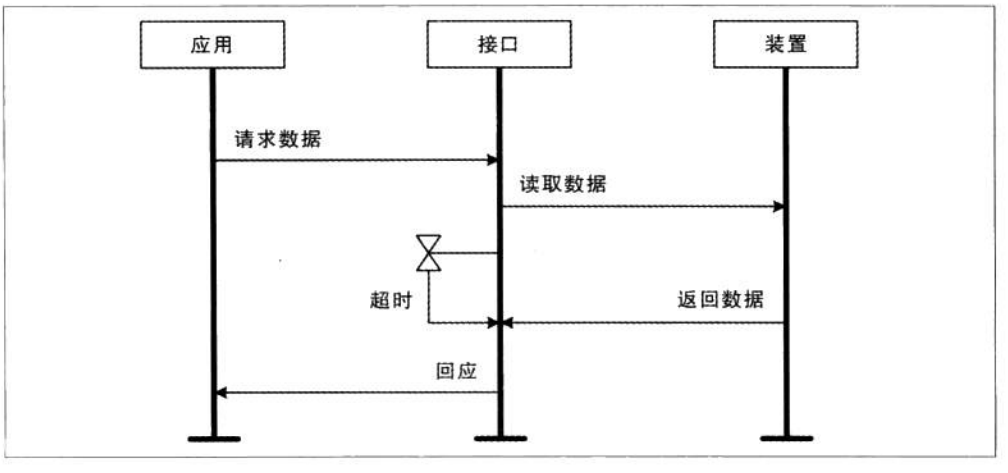


图11-10：I/O 处理

在很多情况下，让一个阻塞调用稍加等待再返回可能并无大碍，而且也更方便实现轮询和重发功能。但是，这里会有一个警告：一个工作于 I/O 设备的阻塞调用，如果没有某种超时处理，就可能会永久挂起。这通常是件很糟糕的事情，而且遇到这种情况一般只能通过关闭 Python 并重启应用来解决。如果你的代码在一个“前不着村，后不着店”的无人值守的机器上运行，那么产生一个挂起阻塞调用的故障可就真的非常糟糕了。 423

使用非阻塞函式调用可以解决此类问题。这样做需要一些额外的代码，但是它在处理网络通信和数据请求时非常有用。我们接下来很快就会看到如何使用这种方式。

数据 I/O 方法

我们现在已经知道阻塞和非阻塞功能都需要什么了，接下来看看这些概念是如何在 I/O 接口的各种操作模式中使用的。我们先从最简单的按需处理的 I/O(on-demand I/O)开始，然后再介绍轮询 I/O，最后了解一下多线程 I/O。

按需处理的数据 I/O

之前提到过，将数据从应用程序中传入或传出的两个最直接的方法就是对端口或设备进行读或者写操作。要向一个串口（RS-232 或 RS-485）或 GPIB 型接口发送数据，通常不需要担心阻塞调用。对于不使用硬件握手（hardware handshaking）的 RS-232 接口，数据通过硬件端口即时发出。具有唯一的主控和多个接收者（listeners）的 RS-485 接口，不会阻塞在主控设备上，但是接收者可能会在一段时间内无响应。GPIB 也会遇到接收者不响应发送者的情况，但是大多数 GPIB 接口 API 和相关硬件能够对此进行检测，并返回一个错误码。向类似 PCI 接口卡的设备的硬件接口 API 写数据时，阻塞通常都不会有问题，如果发生错误，调用同样会返回一个错误码。

如果你的软件使用按需处理的调用来读取数据，它们通常会是阻塞调用，你应该总是在软件中检测返回值。如果一个阻塞函式有超时参数，就会被明确地使用，但不是所有阻塞调用的 API 都有超时（或许它会认为超时不会发生）。对于这些情况，你应该在你的软件中使用非阻塞版本的 API 函式，并采取其他方法来实现超时机制。

轮询数据 I/O

非阻塞调用会立即返回，并通过返回值通知调用者操作是否成功。非阻塞调用可以用来防止 I/O 挂起，但是需要额外的代码来支持它。例如，假设我们用来从设备读取数据的
424 API 兼有阻塞和非阻塞版本的 I/O 函式，或该 I/O 函式可以设置一个参数来控制阻塞。这样的话，你就可以将一个非阻塞调用放到一个能够检测超时的循环中，如：

```
def GetData(port_num, tmax=5.0):
    checking = True
    tstart = time.time()
    while checking:
        rc, data = ReadNonBlocking()
        if rc == ERR:
            break
        if time.time() - tstart > tmax:
            checking = False
            rc = TIMEOUT
        else:
            time.sleep(0.05)     # 每次检测后等待 50ms

    return rc, data
```

这是一个轮询（Polling）的例子：该函式会尝试通过轮询端口（调用 ReadNonBlocking()）从指定设备请求数据，直到可用数据出现。在每次请求之间会睡眠 50ms。这个延时主要是为了照顾所读设备，因为很多设备都无法容忍被一直催着要数据。

要想获得一个实际的轮询功能，并同时不会使应用的其他部分暂停，你就得使用线程。

使用线程请求数据

迄今为止，我们已经学习了按需处理数据 I/O 和轮询数据 I/O。现在，我们很快地看一下，当系统整体不陷入轮询循环时，是如何检测就位数据的。

在下面的代码框架中，有两个我们之前没见过的 API 函式：SendTrigger() 和 GetData()。我们假定它们是获取数据的硬件 API 的一部分，其功能如其名称所示。请求的数据类型没有明确指定，因为这些对于该示例无关紧要。只要能得到指定数目的采样且没有错误发生，它可以是任何内容。

```
class AcqData:
    def __init__(self, port_num, timeout):
        self.timeout = timeout
        self.dataport = port_num
        self.dvals = []          # 用于获取数据的列表
        self.dsamps = 0          # 实际读取的数据量
        self.get_rc = 0          # 0 表示 OK，负值表示错误
        self.get_done = False    # 如果线程结束则为 True


def Trigger(self):
    SendTrigger(self.dataport)                                    425


def _get_data(self, numsamples):
    cnt = 0
    acqfail = False

    while not acqfail:
        self.get_rc, dataval = GetData(self.dataport, self.timeout)
        if self.get_rc == OK:
            self.datasamps = cnt + 1
            self.dvals.append(datavalue)

            cnt += 1
            if cnt > numsamples:
                break
        else:
            acqfail = True
    self.get_done = True


def StartDataSamples(self, samplecnt):
    try:
        acq_thread = threading.Thread(target=self._get_data, args=(samplecnt))
        acq_thread.start()

        self.Trigger()          # 开始获取数据
```

```
        except Exception, e:
            print "Acquire fault: %s" % str(e)

    def GetDataSamples(self):
        if self.get_done == True:
            return (get_rc, self.dsmaps, self.dvals)
        else:
            return (NOT_DONE, 0, 0)
```

这段代码以函式 get_data() 的形式，使用了一个线程，它不断地读取外部设备来获取不定长度的数据。注意，我们假定了 API 函式 GetData() 支持超时参数，并会在超时发生时返回一个错误码。

这个例子的重点是线程是如何被创建的，以及我们如何检测数据获取是否完成。Python 的线程库包含了一个叫做 join() 的线程对象方法，它接受一个可选的 timeout 参数，当某线程需要等待另一个线程结束时，通常用它来阻塞该线程的执行。上面的例子中我们没有用到 join()，因此该线程可以自主运行。存取器(accessor)函式 GetDataSamples() 检测变量 self.get_done 来确保线程是否已经结束。如果结束，GetDataSamples() 将会返回采集到的数据。如果线程仍在执行，它会返回一个 3 元组(3-tuple)，其中第一项被设置为 NOT_DONE。最后，由调用者通过返回值来判断采样总数是否与所请求的数据量相匹配。

426

这只是完成该工作的方法之一，但它展示了一个使用线程工作时经常遇到的基本问题，即程序什么时候停下来等待某些条件，或什么时候结束自己的工作。对于一个持续运行的程序，可以通过将 GetDataSamples() 放到一个单独的主循环中来进行处理。通过在循环中不断检测读回的结果，来查看所请求的数据是否可用。否则，程序就只能检测最后得到的结果。

数据I/O错误处理

无论概率有多小，错误总是会发生的，在处理对真实世界的接口时更是如此。串行接口的杂散噪声、模拟输入电压越界，或者外部仪器故障都有可能导致错误的发生。软件如何检测并处理错误直接关系到其健壮性。换句话说，健壮的软件往往具有高度的容错性。

故障、错误和失败

讨论容错系统时经常会遇到几个通用术语。虽然这些经常交替使用的术语是通用的，但它们在容错系统上下文环境中有着专门的特定含义。

故障（*Fault*）

系统内存在代码、电子或机械方面的缺陷。一个故障只有在偶然出现或者错误发生时才会被发现。那些没有被发现的故障称做潜在故障（*latent fault*）。

错误（*Error*）

当系统遇到故障时会产生某些异常行为。这种异常行为可能是一个错误的运算结果，也可能是通信信道上多余或缺失的数据，或者是一个无用的（或潜在的灾难性）物理行为。根据是否影响程序继续运行，错误通常被会分为致命错误和非致命错误。

失败（*Failure*）

不履行预期行为的错误，通常会被视做系统规范中所定义的必需的行为的偏差。

总之，故障导致错误，错误导致失败。众所周知，要发现一个复杂系统中的所有可能存在的故障几乎是不可能的。即便是看似简单的系统也会藏匿一些潜在的故障。容错设计的目标就是要创建这样一种系统，哪怕有故障出现，也能够在一定程度上使系统继续运转。

一个系统（包括软件、硬件或者软硬件复合系统）能够被称为可容错的（fault-tolerant），就意味着它能够检测出错的情况，并采取相应处理或者忽略错误，以使系统继续运转（可能会在功能上做一定程度的缩减），而不是崩溃或者突然停止。这种在错误发生时对某功能进行一定程度上的缩减以保证系统继续运转的能力，称为优雅的降级（graceful degradation）。当然，如果错误持续上演，系统最终会在某些时候终止，但在此之前，它会给出足够的通知，而且不会以灾难性的方式来终止系统。 427

实际上，大多数故障都会最终导致程序终止或者崩溃。如何对最可能出现的故障以及由此所产生的错误进行规划，主要取决于该故障可能引发的麻烦有多大。它可能是无关紧要（只需忽略或者移除即可）的，也可能是一个真正的大麻烦（导致爆炸、起火或其他灾难的发生）。如果你做了前期规划，如第 8 章中所讨论的那样，那么你就应该能够识别那些最险恶的情景，并知道故障发生时应如何应对。

错误类型

错误可以被泛泛地归为两类：非致命的和致命的。非致命错误（nonfatal error）是类似通信信道间断之类的事情，可能会导致媒体中的噪声或者其他扰动，或者某人不小心踢到桌子下的接口。系统可能会继续运转而没有任何副作用，直到信道被重新连接，这取

决于系统的速度以及错误发生的时期等因素。另一个例子可能是一个仪器出于某种原因偶尔无法及时响应。如果命令或查询能够成功地被重发，且不产生任何副作用，那么这个错误可被视为非致命的。（注意，不致命并不等于不烦人！）

致命错误意味着如果要让系统继续工作，就要进行一些有效的干预。否则，就需要全部关闭。实验中所使用的直流电源失去控制就是致命错误的一个例子。除非有一个可用的备用电源能够自动接入，否则在问题解决之前系统都会被关闭。再例如，有一套控制系统用来为真空环境的抽气泵提供液态氮，控制接口电路或传输命令的通信信道如果发生故障，都可能导致该控制器出问题。这两种情况中的任何一种都将导致系统开始失去真空环境，并可能损坏离子测量仪或溅射仪之类的东西。至少，在问题解决之前它会停止当前的工作。

错误重试与系统终止

当检测到错误时，重试某个操作有时是有意义的，或许还会先调整某些参数来修复错误。这听上去挺智能的（而且确实可以做到），但在实现之前必须认真地考虑错误发生的上
428 下文、原因以及可能导致的后果。贸然尝试重试一个失败的操作可能引发严重的后果。

提供的错误检测和自我修复功能越多，系统就会变得越复杂。这是显而易见的，而这种复杂性如何体现，以及随后产生的影响是很难预测的，这些影响可能不仅仅局限于对一个特定的子系统，还可能牵扯到系统整体。随着复杂性的增加，更多新的缺陷可能会被引入，而且还可能会增加一些意想不到的执行分支。

图 11-11 展示了一个容错版本的数据 I/O 错误控制方案。该方案可能不适用于所有应用，但它说明了为什么实现一个健壮的或含有容错机制的软件，要比那些进行 I/O 操作时只返回成功或失败的软件难一个数量级（或者更多）。对容错性进行测试时更是如此。在图 11-11 中，当错误发生时有三个分支可能被执行。除了 I/O 操作自身以外，每一个分支都必须通过模拟 I/O 及其错误上下文来进行测试。这种严格的测试需要投入更多工作，但如果你确实需要这种级别的健壮性，那也就只能如此了。

注意，图 11-11 展示了所有的代码。数据 I/O 操作及其返回值（如成功或失败）简单明了，可能最多一两行代码就能实现。而含有错误控制的设计，其代码量可能会增长 10 到 100 倍。尤其是带有容错机制的软件。其中错误检测及其处理占据了很大一部分，而真正的 I/O 处理其实只是其中很小一部分。另外请注意，最后的判断块“备份已被使用？”表示，如果备份已被使用（即测试为真），除了失败就没有别的选择了。

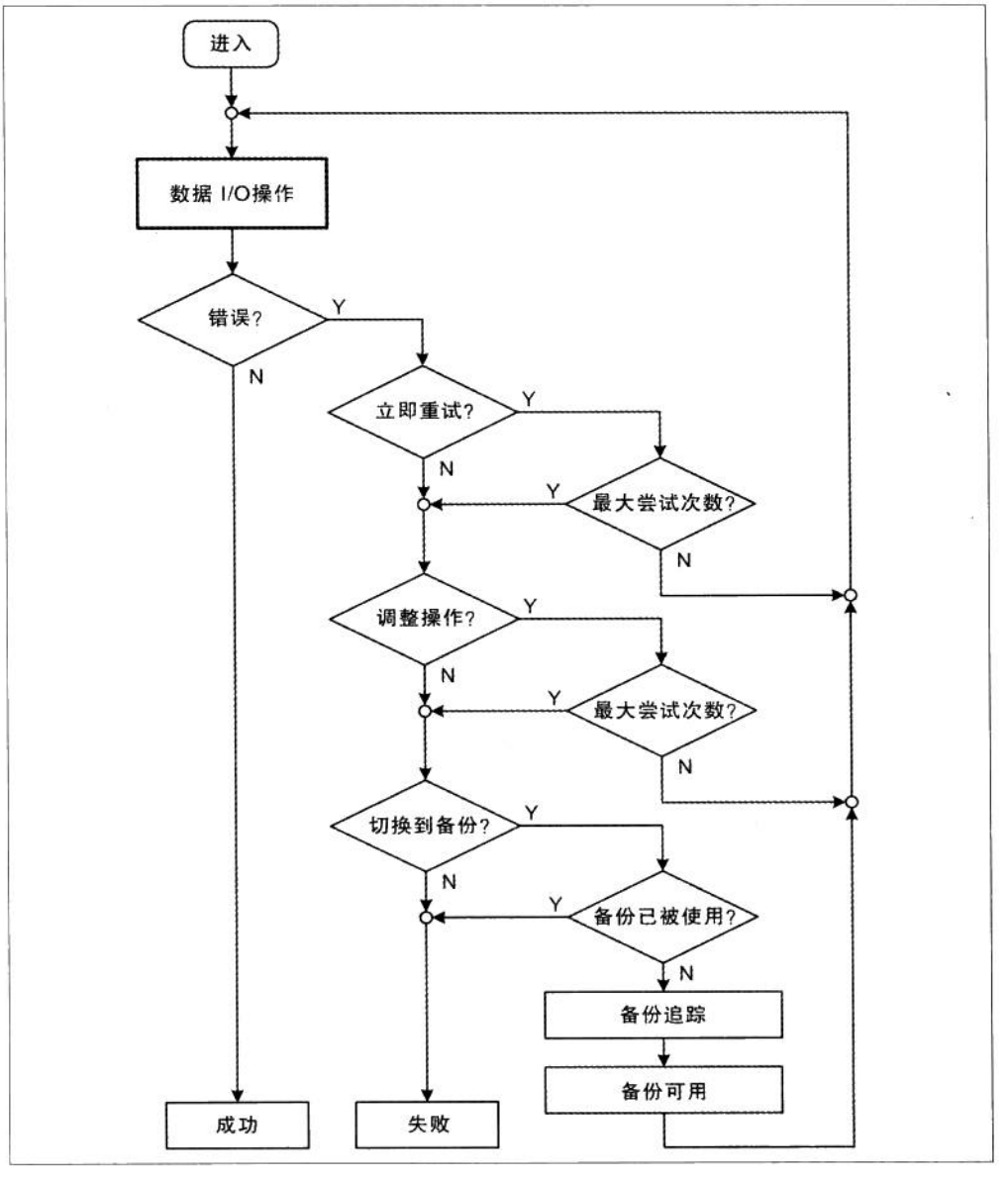


图11-11：防止失效的数据 I/O 错误处理

当检测到一个错误，并尝试对其进行处理时，系统必须做出决定，到底是尝试对错误进行修复（还要考虑如何修复），还是直接正常地关闭程序。处理逻辑必须包括描述错误发生时的上下文以及系统当前状态的数据，并可能还需要定义一些不允许使用的操作。

例如，对于一个控制容器压力的系统，如果不进行某种检查以确定是否释放压力控制系统，就简单地放弃控制，可能就不是一个好的做法。如果压力持续增加，即使在水泵和加热器已被关闭的情况下（这是可能发生的），容器都有可能发生爆炸的危险，特别是当压力开始过载有错误发生的情况。在控制系统彻底结束之前，正常的关机可能需要考虑进行某种排气的工作。

429 类似的，如果一个错误发生在一个某种大型起重系统中，能直接终止系统吗？如果是某种举起或移动大型物体工作中用到的马达的供电装置或随动装置，那么直接切断电源可不是一个好主意。

430 系统可能需要某些类型的制动或锁定机制，使大物体降低到一个安全的位置再关机通常会很有意义。

上述顾虑同样适用于对失败操作进行重试的情况。重试可能不太适用于某些失败，如缺少直接位置的反馈信息，或者温度传感器发生故障。其他失败因素可能只是短暂存在的，在彻底放弃之前，可以进行若干次重试操作。

考虑以下这种情况，从动设备的位置依赖于主设备的位置，而且两者在一个时间段内，都以较慢且相对的速度移动。两者之间由一个通信信道联系，该信道偶尔会由于系统负载或其他因素而中断。这时，随主设备而运行的从动设备或许可以在一小段时间内预测其将要到达的位置。这就能使在主设备未刷新数据时，从动设备可以持续运行。如果与主设备之间的通信在一定时间内无法重新建立，从动设备将进入一个出错状态。如果在超时前能和主设备恢复通信，它就能刷新自己的位置，并视情况重置超时计时器。

故障情况分析，我们在第 8 章的“处理错误和故障”一节中也简要地讲过，它在做出上述决定时发挥着重要作用。如果做得好，它可以在需要决定到底是突然终止，还是正常终止，或是试图恢复时，为你提供指导。缺乏故障分析时，最好的选择往往只是正常终止，并提供足够的信息（通常是在一个崩溃日志中或类似的东西），以让其他人回过头来定位造成问题的原因。

错误 / 警告消息单发（single-shot）逻辑

系统有时会产生一个非致命的错误或一条警告信息，你可能很想知道发生了什么错误，但却不希望看到不断重复的警告信息。

这方面的例子可能是一个高速数据采集设备，受限于某些情况，可能每隔一段时间就会错过一个采集数据的操作。API 库的作者可能已经意识到这样很不好，但你可能并没有，特别是如果你的软件足够智能，以至于可以抛出一个损坏的采样并进行简单的重试时（正如我们之前所讨论的那样）。只要你的软件在数据上使用一个时间戳，而且对数据的实

时性没有特别具体的要求（其实使用一个更加精确的时间戳也可以处理这种情况），你通常可以简单地忽略该错误，并尝试重新获取采样。

下面就是一种处理方法：

431

```
# 在模块的全局命名空间中的某个地方，我们定义了一些
# 控制变量并为其赋初始值（也可以是一些对象的变量）：

msglock = False
errcnt  = 0
errcntmax = 9    # 发生 10 次错误后才开始锁定

# 以下是实际进行数据捕获的函式 / 方法，以及出错后的消息：

def grabData():
    global msglock, errcnt

    rc = Acquire()

    if rc != OK:
        if msglock == False:
            errcnt += 1
            if errcnt > errcntmax:
                print "ERROR: Data acquisition failed %d times" % (errcntmax + 1)
                msglock = True
    else:
        msglock = False
        errcnt = 0

    return rc
```

此处所采用的主要思想是在错误积累到一定数量之后，才发出错误消息。当错误接连发生时，变量 errcnt 将会递增。当它积累到一个阈值时，就会打印一个错误消息，但只打印一次。从 Acquire() 返回的第一个正确的结果将会重置错误计数器和屏蔽变量 msglock。

错误计数器和锁定逻辑也可以放到一个独立的函式中，但函式或方法调用也会占用一定时间，如果你有速度方面的需求，那么不要再用函式进行封装可能更好，或者使用内联代码。

处理不一致的数据

当试图获取数据时，你可能偶尔会遇到数据质量不稳定的情况。在那些上电后需要一段时间的稳定才能进行正常读取的设备上时有发生。在其他情况下，被测量的（误差）值是非常小的，只有系统自身的噪声才会将严重的错误引入数据。

等待稳定

某些仪器或者外设有时会需要一个稳定时期，才能返回正确的数据。如果你预先知道这段时间，那通常只需在读取数据之前等待即可。然而，如果这个时间段是可变的（比如受操作环境温度影响），那么一个确定的超时时间就无法被完全信任了。

图 11-12 展示了如何通过一串连续的测量结果来确定输入稳定的时刻。

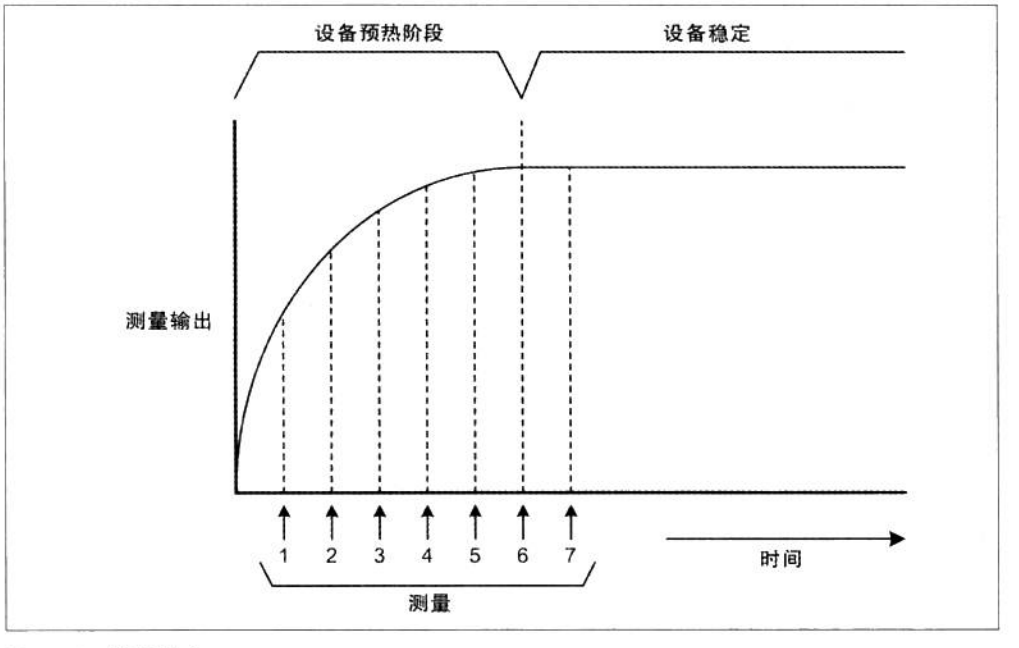


图11-12：等待稳定

视仪器自身情况而定，各测量值之间的差值（delta）可能会随着仪器的预热逐渐减小，直到趋近于零。如图 11-12 所示，1 和 2 之间的差值会比较大，但 6 和 7 之间的差值就会变小。因为有噪声和转换错误，差值肯定不会精确地变为零。

对于其他情况，数据一开始可能变化很大，然后开始趋于一个固定的输出值，正如图 11-13 所示。精密的固态激光控制器在试图测量输出光束的波长时，有时会表现出此类
433 行为。直到控制器和激光头都达到一个最佳的作业温度，否则激光的波长和功率可能波动很大，有时会相当厉害。

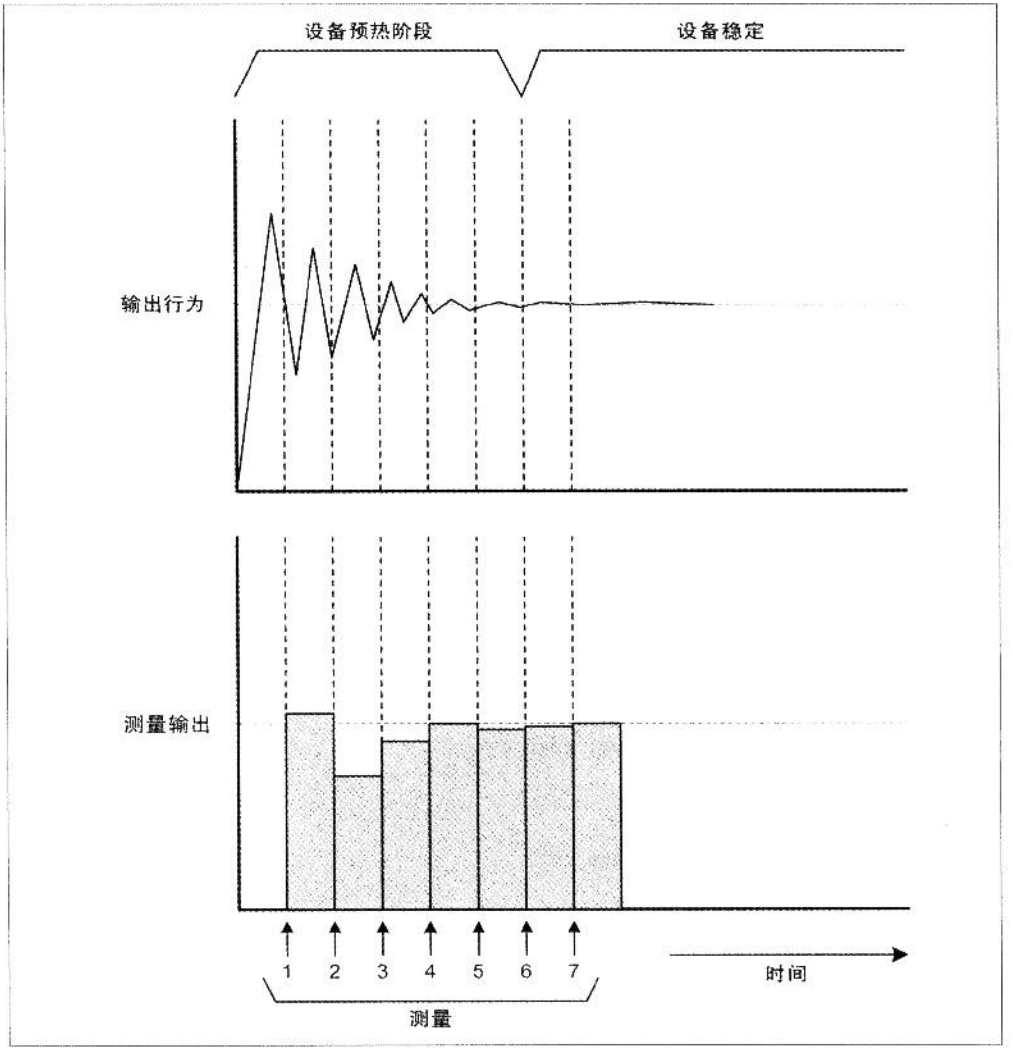


图11-13：温度上升收敛

之前看到的差量测试函式（见本章前面“写入数据”一节）就是一个检测预热阶段稳定性的很好的备选方案。使用一点点额外的逻辑就可以设置能接受的偏差量的误差范围。

在预热阶段，通过简单的定时等待来进行读取和测试的方法，在故障检测方面有着显著的优势。通过采集一系列的测试数据并研究它们之间的差量，就有可能检测出数据源是否已经趋于稳定，并能够知道在一个合理的时间段内读取到稳定输出是否存在问题。 434

处理噪声：求平均值

现在，考虑一下图 11-14 所示的情况。图中给出了一系列的测量值，它看上去是以随机方式浮动的。根据选择的测量尺度的不同，所见可能与其实际情况会有些出入，但读取操作确实是不稳定的。在处理模拟数据，特别是当你想获得一个偏差较小的精确结果时，这种情形会经常出现，加之系统本身还有噪声。

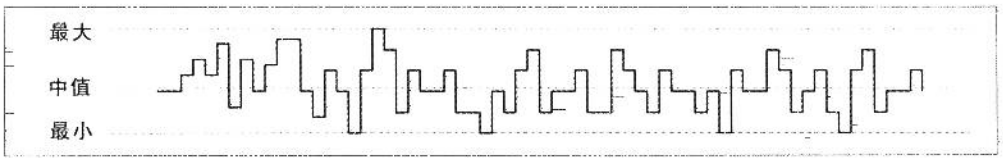


图11-14：数据噪声

一种处理的办法是计算一段时间内输入数据的平均值。下面是一个简单的函式，用来计算传入数据流的平均值：

```
data_sum = 0
data_avg = 0
samp_cnt = 0

def sampAvg(data_val):
    global data_sum, data_avg, samp_cnt

    samp_cnt += 1
    data_sum += data_val
    data_avg = data_sum / samp_cnt

    return data_avg
```

在调用该函式之前，全局变量 data_sum、data_avg 和 samp_cnt 必须被设为 0。现在，假定有一个叫做 dataset 的列表，获取到的数据被求平均值后，交由该列表维护。每一个这样的调用都会获得一个采样，将其与已取得的样品进行平均值计算，并将结果追加到列表中：

```
dataset.append(sampAvg(readInputData()))
```

对于那些每个采样都有所变化的数据，计算平均值非常有用。图 11-15 展示了计算平均值如何使一个波动很大的曲线变得平滑。

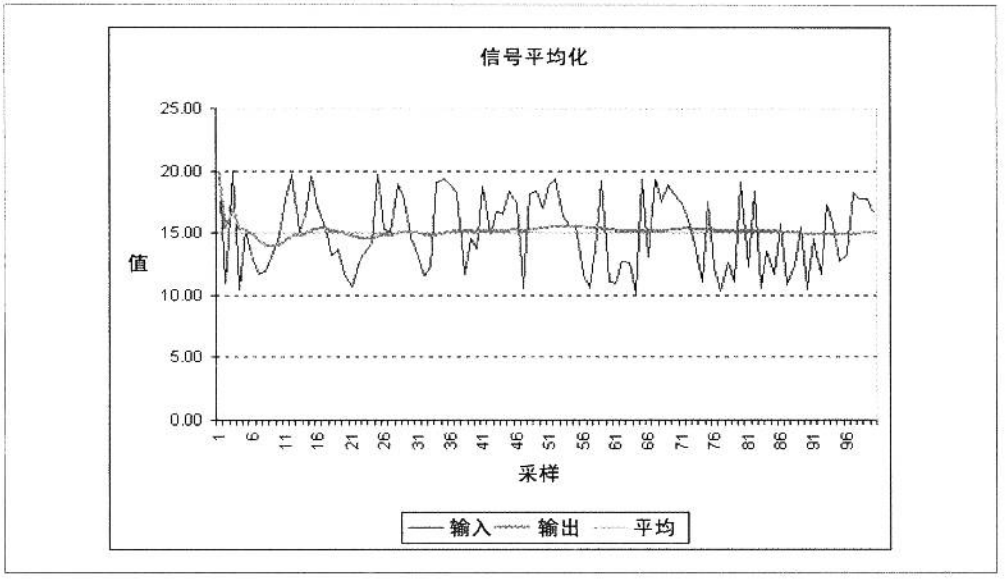


图11-15：快速变化数据的平均化

当然，这不是真实的数据（要是产生出这种凌乱的数据，很可能是某处出现严重的问题了），但它展示了如何通过计算平均值来处理这种输入。 435

顺便说一下，在这个例子中使用的求平均值的函式不是最优的，它只是为了说明这种方法。如果函式能够持续计算一个动态平均值将会更好，这样就不必担心变量的总和最后变成一个巨大的数字。

当数据在小范围内围绕一个稳定或者变化缓慢的值波动时，计算平均值的办法通常都适用，但还是应该谨慎一些。一个趋于正向或负向的被平均了的信号会牵带着该平均值，但如有一个变动迅速发生——比如说，一个返回至原始水平的短暂差异——那么这个变动将会被平均掉。[译注1]

小结

现在，通过了解本章所涵盖的内容，你应该已经对使用 Python 操作外设所涉及的内容及其相关工具有了一定的了解。我们已经看到串口、并口、USB、GPIB、SCPI 和 VISA 接口，以及如何实现一个健壮且具有容错能力的接口。当然，有很多更深层次的知识并未涉及，但凭借目前所学的内容，我们已经可以在一些真正的硬件上工作了。

译注1：此处大意为计算平均值所得到的结果不利于考察短期的变化。

推荐阅读

在书籍方面，USB 可能是最热门的话题。要再仔细找找，你可能才能找到一两本 RS-232 或者 RS-485 方面的书。时下有关 GPIB（或 IEEE-488）方面的书还真不多，但可以通过互联网来获取更多有用的信息。在 Google 中输入“GPIB”或“IEEE-488”将找到大量信息，其中有很多资料都相当不错。

大多数 Python 方面的书籍都介绍了 Python 的 C/C++ 扩展，可以查阅特定硬件的 API 文档以便有效地使用这些接口。

作为一个起点，下面这些资料可能会对你有所帮助。

Real Time Programming: Neglected Topics, 4th ed. Caxton C. Foster, Addison-Wesley, 1982.

此书包含一个简洁而实用的 GPIB 接口及其硬件握手方面的概述。这本书早已绝版，但仍可找到可用的副本（第 9 章也曾引用）。

USB Complete: The Developer's Guide, 4th ed. Jan Axelson, Lakeview Research, 2009.

如果你想了解 USB 方面的更多内容，可以看看这本书。Axelson 很好地解释了 USB 接口的底层细节，特别是在人机接口设备（HID）实现方面。如果你想实现自己的 USB 接口，或者学习更多 USB 方面的知识，这本书将是一个好的开始。

http://www.ivifoundation.org/docs/SCPI-99.PDF

这个链接所指向的文档包含了一份完整的 SCPI 规范。可以通过 IVI（可交互虚拟设备）基金会免费获取。

http://www.ivifoundation.org/specifications/default.aspx

IVI 基金会还提供免费的 VPP-4.3 VISA 库的参考文档。如果你打算使用一个 VISA 兼容接口，你手头上应该有一份该文档的副本。

http://joule.ni.com/nidu/cds/view/p/id/852/lang/en

可以通过此链接下载到 NI-VISA 4.2 驱动套件的 ISO 镜像文件，并直接刻录到 CD。

下面几个半导体公司提供免费的应用文档，其中涵盖多种仪器系统实现方面的很多有趣话题：

- Analog Devices, Inc. (*http://www.analog.com*)
- Maxim/Dallas Semiconductor (*http://www.maxim-ic.com/appnotes10.cfm*)
- National Semiconductor (*http://www.national.com/apnotes/*)
- Texas Instruments (*http://www.ti.com*)

第 12 章 437

读写数据文件

两个人正在研究他们部门的新计算机的输出。分析了一个小时左右，其中一人说："你知道吗，至少得 400 人用上 250 年的时间才会测出这么大的错误？"

—Anonymous

采集到数据后，最好能将其存入文件用以分析和归档。简单的应用，如电子控温器之类简单的应用，可能不太需要保存数据，但若从长远考虑，即使是最简单的应用，保存采集到的数据都可能会提供有价值的信息。在第 10 章中，我们介绍过将结果保存到一个 ASCII 文件中的模拟器，这些结果可用于日后的审查和分析。在本章中，我们将深入了解这些工作是如何完成的，同时还会学习其他一些用于保存数据的方法。

例如，可以想象一下，你家中有一台恒温器，可以记录周围内外环境的温度、恒温点、热水器和空调的活动，并支持某些信息设置。有了这些数据，你就可以看到热水器或空调的工作情况、工作周期，以及是否有人定期打开风扇等。这些数据所能告诉你的东西可能超乎你的想象，特别是如果你连续收集了一到两年的数据以后。

数据可以多种格式来表示。有时它们只是一串 ASCII 字符串格式的数字。某些类型的数据可能把诸如时间和日期信息和测量值组合在一起，还有些数据可能会包含多组测量值和相关的参数，并以结构化的二进制格式存储。无论使用何种格式，保存数据的主要目的都是为了以后能方便复查，或希望用这些数据做一些有用的工作。

"有用"是一个宽泛的术语，但我们这里所说的有用是指数据能够在各种上下文中被方便地访问。换言之，就是这些数据具有一致性的逻辑结构，而且包含了足够的辅助性信息，可以和特定资源、时间或者一个位置联系起来。能够使用文本来表示数值的 ASCII 数据，非常适用于这些场景。 438

Python 在 ASCII 数据处理方面表现得很好，这将是本章重点要阐述的内容。一般而言，

对于那些不需要处理图片、大型数组，或者复杂的复合型数据对象等数据的系统，可以直接使用 ASCII 文件。但并不是所有的数据都能通过 ASCII 来管理，尤其是当需要与其他应用进行交互或者通过网络进行二进制数据传输时。本章的另一个重点就是要讲述如何在只使用 Python 的情况下，创建、保存并读取二进制数据对象。

本章我们将看到很多代码，包括几个功能完整的函式。到本章结束时，我们就将创建起一套可重用的函式和类的集合，当要创建仪表和控制软件时，可以将这些工具与第 10 章所讲的各种仿真器整合到一起。

ASCII 数据文件

经过编码后用来表示字母字符的数据是最佳的计算机交互语言，这种用法已有超过 40 年的历史了。通常所说的文本数据或文本文件，是指这些数据可以直接发送到打印机、显示在终端上，或者可以使用主机系统自带的功能进行编辑、排序和查找。为了实现这一层的跨平台兼容性，ASCII 应运而生，并随后纳入到 UTF 的标准中。

ASCII 编码标准已有至少 50 年的历史。ASCII 是美国信息交换标准码（American Standard Code for Information Interchange）的缩写，它自带了 33 个非打印控制字符和 95 个可打印字符，共计 128 个字符编码。你可能已经猜到，它是一个 7 位的编码方案（2^7=128）。最初的 ASCII 被设计为 8 位字符，但考虑到数据传输的代价以及奇偶位的问题，它最终被驳回了。另一个因素是，需要使用它的设备，如电传打字机和纸带阅读机 / 打孔设备，通常只支持 7 位数据。如果想从历史的角度了解 ASCII 和早期的网络通信之间的关系，你可以看看 Vint Cerf 于 1969 年所写的“RFC20——ASCII 用于网络交换的数据格式”，参见本章最后的“推荐阅读”一节。

现在，最通用的字符编码方式是 UTF-8。该方案保留了原有的 128 个 ASCII 字符，并增加了可再容纳 128 个新字符的空间，扩展了数据的字节长度（具体实现依赖于所使用的编码实现）。增加的数据位和字节用于编码特殊的符号字符，而且英文字母以外的字符集还需要两个或两个以上的字节来编码。我们将主要处理 UTF-8 中原始的 7 位 ASCII 的部分，尽管研究完整的 UTF-8 规范也是一个有趣的话题。

439

原始的 ASCII 字符集

ASCII 设计之初要兼容一些今天已经不复存在的设备和应用，因此它包含了一些奇怪的内容，如其中的很多控制字符，但你可以安全地忽略它们。图 12-1 所示的 ASCII 控制字符罗列了 ASCII 中的非打印控制字符。BEL、BS、LF、CR 和 FF 编码在今天依旧被广泛地使用着。其余的虽然已经很少被用到，但如果确实需要，你仍可正常地使用它们。

Dec	Hex	Symbol	Control Character Definition
0	0	NUL	Null
1	1	SOH	Start of Heading
2	2	STX	Start of Text
3	3	ETX	End of Text
4	4	EOT	End of Transmission
5	5	ENQ	Enquiry
6	6	ACK	Acknowledge
7	7	BEL	Bell (audible or attention signal)
8	8	BS	Backspace
9	9	TAB	Horizontal Tabulation
10	A	LF	Line Feed
11	B	VT	Vertical Tabulation
12	C	FF	Form Feed
13	D	CR	Carriage Return
14	E	SO	Shift Out
15	F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	End of Transmission Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator
127	7F	Del	Delete

图12-1：ASCII控制字符

大多数控制字符看起来已经过时了，但在某些情况下它们还是可以发挥作用的。如果你的项目中用到了老的设备，你可能会发现，ACK、NAK 或者也可能是 EOT 和 ETB 这些字符可用于同步两台设备之间的通信。如果你正在设计一个自己的系统，你也可能会用到这些字符。但是，需要提出一个警告：如果你使用了除 CR 和 LF 之外的非打印字符，那么要想在一个人机交互的终端模拟器上使用你的接口可能就不那么容易了（例如，Tera Term 通信应用程序和第 10 章提到的终端模拟器）。

还要注意在这些不同的操作上，文本行的结束符是不同的。MS-DOS 和 Windows 使用一

对字符 CR 和 LF 作为 EOL（end-of-line）结束符。UNIX、Linux、AIX 以及其他 UNIX 类系统使用一个 LF 字符作为结束符。老的苹果产品使用一个单独的 CR（基于 BSD 的 OS X 使用一个 LF 字符）。

可打印字符中定义了标点符号、数字、美国英文字母表中的大小写字母。图 12-2 所示的 ASCII 可打印字符罗列了这些可打印的 ASCII 字符。

Dec	Hex	Symbol	Dec	Hex	Symbol	Dec	Hex	Symbol
32	20	(space)	65	41	A	98	62	b
33	21	!	66	42	B	99	63	c
34	22	"	67	43	C	100	64	d
35	23	#	68	44	D	101	65	e
36	24	$	69	45	E	102	66	f
37	25	%	70	46	F	103	67	g
38	26	&	71	47	G	104	68	h
39	27	'	72	48	H	105	69	i
40	28	(	73	49	I	106	6A	j
41	29	)	74	4A	J	107	6B	k
42	2A	*	75	4B	K	108	6C	l
43	2B	+	76	4C	L	109	6D	m
44	2C	,	77	4D	M	110	6E	n
45	2D	-	78	4E	N	111	6F	o
46	2E	.	79	4F	O	112	70	p
47	2F	/	80	50	P	113	71	q
48	30	0	81	51	Q	114	72	r
49	31	1	82	52	R	115	73	s
50	32	2	83	53	S	116	74	t
51	33	3	84	54	T	117	75	u
52	34	4	85	55	U	118	76	v
53	35	5	86	56	V	119	77	w
54	36	6	87	57	W	120	78	x
55	37	7	88	58	X	121	79	y
56	38	8	89	59	Y	122	7A	z
57	39	9	90	5A	Z	123	7B	{
58	3A	:	91	5B	[	124	7C	|
59	3B	;	92	5C	\	125	7D	}
60	3C	<	93	5D	]	126	7E	~
61	3D	=	94	5E	^			
62	3E	>	95	5F	_			
63	3F	?	96	60	`			
64	40	@	97	61	a			

图12-2：ASCII可打印字符

Python 的 ASCII 字符操作方法

你可能已经注意到图 12-2 所示的 ASCII 可打印字符中相应的大小写字母字符的值相差 32（十六进制为 0x20）。因此，如果你想将一个小写字母转换为大写字母，只需检测该字符的 ASCII 值是否大于等于 0x61 且小于等于 0x7A，然后减去 0x20 就可以了。

虽然类似实现在 C 代码和汇编语言中经常用到，但在 Python 中大可不必如此。内置的

一些函式已经包含诸如字符和序数（整型）之间相互转换的功能，而且 Python 提供了各种字符串方法来进行像大小写转换这样的操作。

可以使用 chr() 和 ord() 函式来处理 ASCII 字符对应的数值。下面是 Python 文档中关于这几个函式的介绍（引自 Python 标准库文档，第二节“内置函式”）： <440

chr(*i*)

返回一个 ASCII 编码值为整数 *i* 的字符串。例如，chr(97) 返回字符串 'a'。该函式的作用和 ord() 正好相反。这个参数必须在范围 [0..255] 内，如果参数越界就会产生一个 ValueError 异常。可再参考 unichr()。

ord(*c*)

传入长度为 1 的字符串，如果参数是一个 unicode 对象，就返回其 Unicode 编码的整型值，否则，如果是一个 8 位字符串，就返回该字节的值。例如，ord('a') 返回整数 97，ord(u'\u2020') 返回 8224。这个函式与处理 8 位字符串的 chr()，以及处理 unicode 对象的 unichr() 正好相反。如果传入 unicode 参数且 Python 中构建了 UCS2 Unicode 的支持，那么字符的编码必须在 [0..65535] 范围内；否则如果字符长度为 2，就会产生一个 TypeError 异常。 <441

事实上，这些函式都是字节到整数的转换。因为和 C 语言不同，Python 没有 char 类型，你不能从字符串中只摘取一个字符或希望直接得到一个 8 位的值。例如：

```
>>> foo = "ABCD"
>>> foo[0]
'A'
>>> int(foo[0])
Traceback(most recent call last):
  File "<stdin>", ine 1, in <module>
ValueError: invalid literal for int() with base 10: 'A'
```

这样做不能工作，但是如果你使用 ord() 函式，就能得到一个 ASCII 字符的整型值： <442

```
>>> ord(foo[0])
65
```

如果想要反向转换，可以使用 chr() 函式：

```
>>> chr(65)
'A'
```

或者，传入十六进制数的格式：

```
>>> chr(0x41)
```

```
'A'
```

chr() 返回一个单字符的字符串对象。

字符串操作方法还有很多种。我们最早曾在第 3 章中见到过这些方法，在本章中将进一步学习如何使用它们，如进行大小写转换、对齐和解析等操作。

读写 ASCII 平面文件

一个 ASCII 平面文件（flat file）是指一个文件由一行或者多行称为记录（records）的 ASCII 文本组成。每条记录可能包含一个或多个数据元素或字段（fields），之间用空格、逗号、斜杠，或者其他字符分隔。每条记录都是一个完整的数据实体。术语“平面（flat）”是指单一文件中的数据存放在一个二维表格的行和列中，且同一文件中每条数据之间没有直接关系。平面文件数据库的历史和计算机一样久，而且它们至今仍在各种应用中扮演着重要角色。

记录

一条记录是一行以 EOL（end-of-line）字符（在 Windows 上是一个字符串）终止的字符。每条记录字段之间都用分隔符分开。有时，可以通过计算某些字段是否总是具有相同大小来确定分隔符的位置，但我不推荐这种做法，因为它会降低数据的可读性。分隔字符允许数据字段宽度可变，只要一条记录中字段的总数和预期相匹配就可以了。固定宽度的办法是可行的，但我相信采用分隔符也很不错。

这里有一条记录的例子，它可能会在数据采集时被用到：

```
0021 080728 101829 02  4.99
```

443 这是一个包含了 5 个固定宽度字段的柱状记录，每个字段之间采用空格字符作为分隔符。这些字段（从左到右）依次为序号，YYMMDD 格式的日期数据，HHMMSS 格式的 24 小时制时间，输入端口号，以及从端口读取到的电压值。虽然此处使用了空格字符作为分隔符，但任何不会出现在数据字段中的字符都是可以用做分隔符的（逗号、冒号、竖线标志、等号，甚至美元符号等）。电压值前面多出来的空格是留给可能会用到的负号的。

以 YYMMDD 格式来写日期可以方便值的排序和提取。使用更一般的格式 DDMMYY（或 DD MMM YY 风格，MMM 表示月份前三个字母的缩写）可能更易于阅读，但在排序的时候比较麻烦，可能要写更多额外的代码。

微软的 Excel 表格能够处理使用固定字段宽度或自定义分隔符的 ASCII 数据平面文件（它自己知道怎样处理一个我们即将提到的 CSV 文件，不需要用户进行干预）。下面是从一个理想的仪器系统中采集的一段数据：

```
0000 80728 101808 02  4.78
0001 80728 101809 02  4.82
0002 80728 101810 02  4.80
0003 80728 101811 02  4.84
0004 80728 101812 02  4.86
0005 80728 101813 02  4.83
0006 80728 101814 02  4.86
0007 80728 101815 02  4.87
0008 80728 101816 02  4.85
0009 80728 101817 02  4.88
0010 80728 101818 02  4.89
0011 80728 101819 02  4.90
0012 80728 101820 02  4.91
0013 80728 101821 02  4.90
0014 80728 101822 02  4.91
0015 80728 101823 02  4.92
0016 80728 101824 02  4.94
0017 80728 101825 02  4.95
0018 80728 101826 02  4.97
0019 80728 101827 02  4.96
0020 80728 101828 02  4.98
0021 80728 101829 02  4.99
0022 80728 101830 02  4.98
0023 80728 101831 02  5.00
0024 80728 101832 02  5.01
0025 80728 101833 02  5.03
0026 80728 101834 02  5.04
0027 80728 101835 02  5.02
0028 80728 101836 02  5.05
0029 80728 101837 02  5.06
0030 80728 101838 02  5.08
```

从这些数据中可看出，输入从 2008 年 7 月 28 日 10:18:08 开始活跃，整个数据涵盖了 30 个测量点。图 12-3 所示的简单折线图由输入的电压数据产生。该图通过向 Excel 导入数据，然后使用简单的折线图绘制。 444

向 ASCII 数据文件写入数据

在 Python 中保存 ASCII 数据很简单，只需将数据按一定格式保存到一个字符串中，再将该字符串写入文件即可。首先，我们打开一个文件来接收数据：

```
fout = open("datafile.txt", "a")
```

有两点值得注意。首先，文件名在这里是直接写在代码里的，而你可能并不想一直如此。或许使用一个字符串变量会更好，就像第 10 章介绍 I/O 设备仿真器时所做的那样。其次，此处没有检测打开文件是否成功。

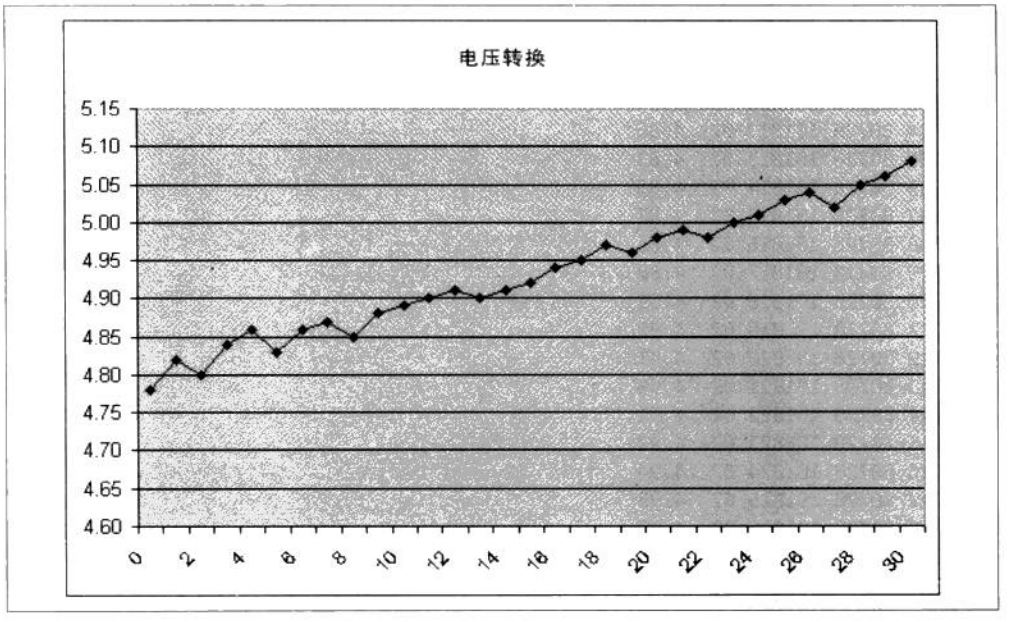


图12-3：简单输入数据图

可以像下面这样加上这些被遗漏的特性：

```
dataname = "datafile.txt"
datamode = "a"

try:
    fout = open(dataname, datamode)
except Exception, e:
    print "Output file open error: %s" % str(e)
```

445 当然，还有其他实现方法，但这种已经能够达到我们的目的了。

文件被打开后，就可以往里面写数据了。假设 Python 的 time 和 datetime 库模块都已被导出，所以我们可以产生时间戳数据：

```
# 从系统中获得当前时间和日期
t = datetime.datetime.now()
currdatetime = t.timetuple()
currutime = time.mktime(t.timetuple())
yr = str(currdatetime[0])
curr_date = "%02d"%int(yr[2:]) + "%02d"%currdatetime[1] + "%02d"%currdatetime[2]
curr_time = "%02d:"%currdatetime[3] + "%02d:"%currdatetime[4] +
"%02d"%currdatetime[5]
tstamp = curr_date + " " + curr_time
```

tstamp 是一个字符串，其中包含 YYMMDD 格式表示的年份，和 HHMMSS 格式表示的当前时间，之间用一个空格字符分隔。最后，再将数据格式转化成之前示例数据的那个样子，然后将该行数据写入输出文件：

```
outstr = "%d %s %d %4.2f" % (seq_num, tstamp, port_num, dataval)
fout.write(outstr+"\n")
```

此处变量 seq_num、port_num 以及 dataval 应该是程序其他部分提供的。

注意，一个文件对象的 write() 方法不会自动添加 EOL，因此必须显式地增加一个 \n 结束字符。

还有一种写数据的方法（Python 中几乎总有多种方法），那就是使用 print 语句，而且不需要 EOL 字符（print 自己就支持它，如下）：

```
print outstr >> fout
```

还可以将生成时间戳和输出数据的代码放到一个函式或方法中，这样就可以在任何时候重用它们。

从 ASCII 数据文件读取数据

从 ASCII 数据文件读取数据与向其写入数据一样简单。Python 提供了一些非常有用的方法来处理字符串，包括按照指定的界定符（delimiter）将一个字符串中各个组成部分解析出来。

假定我们想要读取的一个 ASCII 数据文件内容如下：

```
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
1.000   0.000   0.000   0.000
1.000   1.085   1.000   1.250
1.000   0.839  -0.085  -0.106
1.000   0.908   0.161   0.201
1.000   0.900   0.092   0.116
1.000   0.909   0.100   0.125
1.000   0.914   0.091   0.113
1.000   0.919   0.086   0.108
1.000   0.923   0.081   0.101
1.000   0.927   0.077   0.096
1.000   0.930   0.073   0.091
1.000   0.934   0.070   0.087
1.000   0.936   0.066   0.083
1.000   0.939   0.064   0.079
1.000   0.941   0.061   0.076
```

446

```
1.000   0.944   0.059   0.073
1.000   0.946   0.056   0.070
1.000   0.948   0.054   0.068
1.000   0.950   0.052   0.065
```

可以像下面这样读取每一行纵向格式（columnar-format）的ASCII数据文件，并打印每个字段。它不像看起来那样特别有用，但却有助于展示一些关键点：

```
fin = open('testdata.dat', 'r')
for line in fin:
    lineparts = line.split()
    for i in range(0, len(lineparts)-1):
        print lineparts[i],
        print " ",
    print lineparts[i + 1]
```

这一小段代码中有两点值得注意。首先，它不关心文件中有多少列，而只要有每个字段之间一个或多个空白字符就可以。其次，各列数据是按行来打印的，为了防止换行发生，在for循环中除了最后一个print，其他在print语句后面加了个逗号。这是为什么每个for循环会在每行的最后一个字段之前终止。下面是其输出结果：

```
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
1.000   0.000   0.000   0.000
1.000   1.085   1.000   1.250
1.000   0.839   -0.085  -0.106
1.000   0.908   0.161   0.201
1.000   0.900   0.092   0.116
1.000   0.909   0.100   0.125
1.000   0.914   0.091   0.113
1.000   0.919   0.086   0.108
1.000   0.923   0.081   0.101
1.000   0.927   0.077   0.096
1.000   0.930   0.073   0.091
1.000   0.934   0.070   0.087
1.000   0.936   0.066   0.083
1.000   0.939   0.064   0.079
1.000   0.941   0.061   0.076
1.000   0.944   0.059   0.073
447 ▷ 1.000   0.946   0.056   0.070
1.000   0.948   0.054   0.068
1.000   0.950   0.052   0.065
```

基于之前所看到的，我们可以实现一个通用的ASCII纵向数据文件阅读器来提取特定列的数据值。有了这样的工具，我们就可以通过管道很容易地将其输出导入到另一个工具（如绘图），或者保存到一个临时文件中留以备用。在Linux系统上，如果你的Python解

释器不在 */usr/bin* 目录下，你可能需要改变第一行的指向：

```
#! /usr/bin/python
#=================================================================
# readascii
#
# 一个从包含列数据的 ASCII 文件中提取数据的简单工具
#
# 包括能够提取指定的列数据或一个范围的列数据
# 也可以跳过指定数目的行
# 打印输出到标准输出
#=================================================================
import sys
import getopt

startcol = 0  # 默认从 0 列开始
colspan  = -1 # 默认取所有列
hdrskip  = 0  # 默认跳过 0 行标题行（无标题行）
fname    = '' # 默认输入文件名为空

def usage():
    print "Usage: readascii [options] file_name"
    print " Options:"
    print " -c Start column (default is zero)"
    print " -s Column span (default is all columns"
    print " -h # of header lines to skip (default is zero)"
    sys.exit(1)

# 获取命令行参数
if len(sys.argv) > 1:
    try:
        clopts, clargs = getopt.getopt(sys.argv[1:], ':c:s:h:')
    except getopt.GetoptError, err:
        print str(err)
       sys.exit(2)
    #endtry

    for opt, arg in clopts:
        if opt == "-c":
            startcol = int(arg)
        elif opt == "-s":
            colspan = int(arg)
        elif opt == "-h":
            hdrskip = int(arg)
        else:
            print "Unrecognized option"
            usage()
    if len(clargs) > 0:
        fname = clargs[0]
```

448

```
else:
    usage()

# 打开输入文件
try:
    fin = open(fname, 'r')
except Exception, err:
    print "Error: %s" % str(err)
    sys.exit(2)

# 检验是否需要跳过标题行
if hdrskip > 0:
    for i in range(0, hdrskip):
        fin.readline()

# 从输入文件中读取、解析并输出选定的字段
for line in fin:
    lineparts = line.split()
    if colspan == -1:
        colspan = len(lineparts)
    for i in range(startcol, (startcol + colspan)):
        print lineparts[i],
        print " ",
    print ""
```

要在 Windows 上使用这个工具，可以直接在命令行中这样调用：

```
python readascii.py datafile.dat -c1 -s1 -h3
```

在 Linux 中，只需输入脚本名字（脚本文件需要可执行权限）。这两种方法都会启动 Python，然后加载并执行 readascii.py，并使用输入文件 *datafile.dat*。这些选项指定该工具应该只提取第二列（列的编号从零开始开始）并跳过输入文件最上面的三行。

CSV 文件

一个 CSV（comma-separated values，逗号分隔值）文件是一种平面文件，它几乎已成为各种应用程序之间交换数据的标准方法。CSV 已经存在很长时间了，甚至在个人电脑出现前就已经有了。大多数表格应用都支持 CSV，Python 的标准库中甚至提供了一套专门处理 CSV 数据的方法，即 csv 库模块。

不幸的是，CSV 文件并没有专门的“标准”。CSV 有多种风格，就像“方言（dialects）”一样。微软的 Excel 就是一种独特的 CSV 方言。RFC 4810，“一般格式和 MIME 类型的逗号分隔值（CSV）文件”（参见本章最后的“推荐阅读”一节），定义了一个 CSV 文件的推荐
449
标准。PEP 305 为 Python 标准 CSV 模块定义了一套 API，它能够支持各种不同的 CSV。更多关于 CSV 格式的信息可以在维基百科（Wikipedia）上找到。

如果你只是想要一个简单的CSV，它不需要什么字段之类的头部信息，只要能将数据导入到Excel之类的程序中就行，那么你可以简单地创建一个字符串格式的数据，就像本章已经测试过的那些。然而，如果你想要体验更好一些，可能要研究一下Python的CSV功能了。

Python的csv库有两个预先注册好的方言：*excel-tab*和*excel*。你可以用csv.register_dialect()方法定义并注册一个新的方言。csv模块支持带引号的字符串和空列，它主要的优点是能够帮助你处理更多细节问题。如果你不需要这一级别的控制，也可以使用自己的方法。

配置数据

配置数据文件是一个通用且方便的方法，用来对应用程序的控制参数进行集中式管理。在Windows平台上，它们通常被称为“INI”文件，并被各种应用广泛地使用着，尽管微软已经明确建议开发者放弃这种方法，转而使用Windows注册（Windows Registry）机制。配置数据文件在Windows系统上通常有一个*.ini*的扩展名（因此称为“INI”）。在UNIX类系统上，配置数据文件通常指“config”文件，而且可以有任意的扩展名，不过按照惯例会使用**.conf*和**.rc*。我们从现在开始将统一使用“配置文件（config file）”这一术语来指代这两种类型。

一个配置文件是一个ASCII平面文件数据库，其中每条记录只有两个字段，称做键/值对（KVP，key/value pair）。在Windows系统上分隔符通常是等号（=）。在Linux系统上，很多应用可能会使用一个冒号代之。

尽管参数配置很多时候都只是在功能完成后才想到随手指定的，但一个应用的配置参数在其测试、调整和维护过程中充当着重要的角色。配置文件冗余而过大，会导致程序崩溃或者产生其他奇怪问题。事先花点时间规划一下如何利用你的配置文件：对大小、背景和内容类型设置一些初始化值，然后在使用过程中保持一致性。你以后会得益于此。

基本的配置文件结构

简单地说，一个配置文件包含一个或多个键/值对，每个键/值对都是一行记录，键/值对之间通常用一个等号或者冒号隔开。下面是一个*C:\Windows*目录下的INI文件的示例：

```
[Window]
Xpos=0
Xright=640
Ypos=0
Ybottom=1024
[Font]
```

```
Height=10
Weight=400
Italic=0
CharSet=0
Pitch=49
Name=Times New Roman
```

下面这几行代码是 Linux 机器上的一个 *.*conf* 文件，它使用冒号做分隔符：

```
wordlist_extend: true
minimum_word_length: 1
maximum_word_length: 25
wordlist_cache_size: 10485760
wordlist_page_size: 32768
wordlist_compress: 0
wordlist_wordrecord_description: NONE
```

配置文件也可以包含其他特性，如首行信息、注释等。注意，不是所有平台和库都支持所有可能存在的变种。要了解 Python 对配置文件提供了多少支持，请参阅这篇文档：*http://docs.python.org/library/configparser.html*。维基百科对此也有一篇质量很高的文章，可以在 *http://en.wikipedia.org/wiki/INI_file* 找到。

使用配置文件

长期以来，在使用配置文件方面，好的或不好的方法我都见过。这里有一些从我日常工作中整理出来的经验，可能对你也会有用。这些经验以 Python 为主（我最近一直在用的），但它们同样适用于其他语言。

1. 多个小配置文件要比一个大文件好。
   小文件在代码中更容易管理。改变一个特定子系统的行为只需要改变它自己的配置（越小越好），而不需要去修改一个巨大的全局配置文件。
   小文件更容易阅读和理解，这可以方便用户在必要时手动编辑它们。
   在代码中对小文件进行修改更安全。小文件能够隔离配置文件中潜在的错误，并防止灾难性的错误在众多参数之间蔓延。例如，当修改配置文件中的值时，可能因为一个系统故障而导致之前所有的配置信息损坏。
2. 尽可能地避免使用 Windows 注册表。
   只要你的程序有需要被移植到其他平台的可能性，或者将来某一天会希望用户手动修改配置文件，那就尽可能不要使用 Windows 注册表来保存你的配置数据。甚至只要有一丝的可能，就应使用基于纯文本的 INI 型文件代之。这既能确保可移植性，也能方便用户使用。
3. 在键和值中使用简单的字符串。
   如果一个值无法使用一个简单的字符串来描述，那它很可能就不应该被放入配置文

451

件中。换言之，如果代码不能检测出 KVP 中值（value）的数据类型（如整型、浮点型、长整型、字符串类型；或者是 Python 中的元组、列表或字典对象类型），它就可能是一个自定义格式的数据文件。Python 针对序列化对象提供了提取机制，还有一些针对二进制数据的选项，我们将在本章稍后讲述。

4. 成对的数据元素。

   当一个参数需要额外的信息时，如一个测量单位，使用 Python 的元组或者将单位值放在一个独立的括号中存储。例如，可以当做一个 Python 元组：

```
output_lambda:        (687, nm)
```

   或者两个独立的参数：

```
output_lambda:        687
output_lambda_units:  nm
```

   不要像这样存储：

```
output_lambda:        687 nm
```

   诸如自动转换器（下一节中会讲到）这样的 Python 工具能够非常容易地处理所有标准的 Python 类型，但它无法控制像“687 nm”这样的值。要解决这样的问题，你就得自己实现一个自定义的解析器，不过我宁愿花这时间去做别的事情。采用独立的参数来存储值和单位可以使之在任何语言中都更易于处理。

5. 不要滥用配置文件。

   对于数据存储，二进制格式更为适合。将一个 2D 图像（哪怕是很小的）转换为一个 ASCII 的 2D 数组存放到配置文件中并不是一个好主意。先别笑，我还真见过有人这么干。

AutoConvert.py 模块——自动转换字符串

下面的工具函式会尝试将一个合法的 Python 字符串类型数据转换成其实际对应的数据对象。其创建的最初目的是操作一个配置文件中的值串（即 KVP 中的 V），但它同时也可以处理你可能关心的任何数据（也有一些例外——见代码注释）。如果输入已经是一个非字符串类型的数据，它会被简单地忽略。

这是 AutoConvert.py 的源代码：

452

```
def AutoConvert(input):
    """ 试图识别任何类型的输入并转换成标准的 Python 数据类型。
```

```
    用于处理的配置数据文件（即"INI"文件）中存储的KV值对，
    总会是字符串格式。

    返回转换后的值（如果没有转换发生则同输入值）同数据类型合为2元组数据。
    总是返回有效的数据，即使返回的仅仅是原始输入的一个副本，
    永远不返回 None。

    需要注意的是，布尔值字符（"t"、"T"、"TRUE"等）会转换成 0 或 1。
    不返回真正 Python 的 True 或 False 的布尔值。
    如果你真的想要输出布尔值，这点可以很简单地加以修订。
"""
if type(input) == str:
    if input.isalpha():
        # 将T/F字符转换为1/0
        if len(input) == 1 and input.upper() == 'T':
            ret_val = 1
        elif len(input) == 1 and input.upper() == 'F':
            ret_val = 0
        else:
            if input.upper() == "TRUE":
                ret_val = 1
            elif input.upper() == "FALSE":
                ret_val = 0
            else:
                ret_val = input
            #endif
        #endif
    elif input.isdigit():
        # 字符串中的整数，直接转换
        ret_val = int(input)
    elif input.isalnum():
        # 混合字符串，然后继续
        ret_val = input
    else:
        # 检查是否是浮点数，或是其他的
        try:
            ret_val = float(input)
        except:
            # 检查是否是元组、列表或是字典
            # 以下代码将误作为参数的内部功能或内置方法的调用尝试
            try:
                ret_val = eval(input, {"__builtins__":{}}, {})
            except:
                # eval检查后，继续
                ret_val = input
    #endif
else:
    # 我们将假设输入是一个有效的类型，却没能在字符串中被捕获，
    # 这可能是有问题的，因为从传统的*.ini文件中提取的值应该是一个字符串。
    ret_val = input
```

```
    #endif

    ret_type = type(ret_val)

    return ret_val, ret_type


if __name__ == "__main__":
    print "%s  %s" % AutoConvert("T")
    print "%s  %s" % AutoConvert("F")
    print "%s  %s" % AutoConvert("t")
    print "%s  %s" % AutoConvert("f")
    print "%s  %s" % AutoConvert("[0, 4, 1, 8]")
    print "%s  %s" % AutoConvert("5.5")
    print "%s  %s" % AutoConvert("-2")
    print "%s  %s" % AutoConvert("42")
    print "%s  %s" % AutoConvert("Spam, spam, spam")
    print "%s  %s" % AutoConvert("(1, 2, 3)")
    print "%s  %s" % AutoConvert("{1: 'fee', 2: 'fie', 3: 'foe'}")
    print "%s  %s" % AutoConvert(1)
    print "%s  %s" % AutoConvert(99.98)
    print "%s  %s" % AutoConvert([1, 2,])
    print "%s  %s" % AutoConvert((9, 8, 7))
```

当你按照如下方式运行 AutoConvert.py 时：

```
python AutoConvert.py
```

你会得到这样的输出：

```
1   <type 'int'>
0   <type 'int'>
1   <type 'int'>
0   <type 'int'>
[0, 4, 1, 8] <type 'list'>
5.5   <type 'float'>
-2.0    <type 'float'>
42    <type 'int'>
Spam, spam, spam <type 'str'>
(1, 2, 3)    <type 'tuple'>
{1: 'fee', 2: 'fie', 3: 'foe'}  <type 'dict'>
1   <type 'int'>
99.98     <type 'float'>
[1, 2]    <type 'list'>
(9, 8, 8) <type 'tuple'>
```

在大多数情况下，AutoConvert.py 都会准确地猜出输入的是什么数据，并正确地处理它们。

454 我需要说明一下在 AutoConvert.py 中用到的 eval() 这个方法：

```
ret_val = eval(input, {"__builtins__":{}}, {})
```

使用 eval() 的时候不可掉以轻心。它被认为是一个安全风险，因为它会执行任何有效的 Python 语句。为了解决这个问题，我用这种方式调用它，这样它不会执行内置函式或方法，且无法访问本地的名字。eval() 是一个强大的功能，但要小心使用。

FileUtils.py 模块——ASCII 数据文件 I/O 工具

接下来，我们来看一个以固定格式读 / 写 ASCII 数据文件的工具模块。为了读 / 写 ASCII 数据记录，FileUtils.py 库模块定义了两个类来读写 ASCII 数据记录，并用到了两个文本文件：ASCIIDataWrite 和 ASCIIDataRead。在 FileUtils.py 中，我们可以看到一个和所讨论主题相关的例子是如何工作的。它也可以作为一个框架，你可以基于此扩展出自己所需要的功能。

我使用类而不是函式来实现这个工具，主要是因为对象可以维护其自身的内部变量（亦称为属性），而且每个对象的实例都会关联到一个指定文件。从本质上讲，对象封装了一组函式来进行格式处理，错误检查，当然，还有文件读 / 写。这允许每个类都能根据不同的目的创建多个独立的对象实体，而不必担心共享的全局变量或文件访问会发生冲突。

每个类都提供了打开、关闭，以及读 / 写 ASCII 数据的方法：

```
class ASCIIDataWrite()
    openOutput(self, path, file_name, reset_file=False)
    closeOutput(self)
    writeData(self, dataval, use_sn=False, use_ts=False)

class ASCIIDataRead()
    openInput(self, path, file_name)
    closeInput(self)
    readDataRecord(self)
    readDataFields(self)
```

该模块还提供了一个共享的函式来关闭一个文件对象：

```
closeFile(file_id)
```

单行记录中的数据是表 12-1 中的四种指定格式之一。

表12-1：ASCII 数据格式

方法	格式
4	[*sequence number*] [*date*] [*time*] [*data*]
3	[*date*] [*time*] [*data*]
2	[*sequence number*] [*data*]
1	[*data*]

采用这种方式，就可以通过简单地计算一条记录中字段的个数，来获知该记录都包含什么字段。其顺序总是与表 12-1 所示一致。（注意，时间戳实际上是两个字段：*[date]* 和 *[time]*。）例如，有两个或四个字段的记录会包含一个序号。只有三个字段的记录会包含时间戳的字段，但没有序号。所有的记录格式都包含一个 *[data]* 字段，它总是以浮点数字符串的形式存入文件。换言之，写入整型时也会保留被设为零的小数部分。

ASCIIDataRead 类方法在表 12-2 中有所描述，表 12-3 展示了 ASCIIDataWrite 类的方法。

表12-2：ASCIIDataRead 类的方法

方法	描述
closeInput(*self*)	关闭一个被打开的输入文件。如果 *file* 没有被打开，就返回一个错误，否则就调用模块函式 closeFile() 来关闭实际的文件
openInput(*self*,*path*, *file_name*)	为输入 ASCII 数据而打开一个文件。如果 *path* 没有指定（传入的 path 是一个空字符串），该函式会尝试在当前目录打开指定名称的文件
readDataFields(*self*)	读取 ASCII 数据文件中记录的各个字段，每次读取一条记录，并返回一个列表对象，每个元素表示一个字段。遇到 EOF 就返回一个空列表。每个字段中的数据会根据记录中字段的总数从字符串类型转换成相应的数据类型。返回一个由返回码和字段列表对象组成的二元组
readDataRecord(*self*)	读取完整的记录字符串，每次读取一条记录，并原样返回文件中的记录字符串，遇到空记录时会返回 EOF。返回值是一个由返回码和记录字符串组成的二元组

表12-3：ASCIIDataWrite 类的方法

方法	描述
closeOutput(*self*)	关闭一个已经打开的输出文件。如果 *file* 没有被打开，就返回一个错误
openOutput(*self*, *path*, *file_name*, *reset_file=False*)	为输出 ASCII 数据而打开一个文件。如果 *path* 没有指定（path 是一个空字符串），就在当前目录打开该文件。如果 *reset_file* 参数是 False，该文件会以追加（append）模式打开。如果为 True，该文件会被以写（write）模式打开，如果该文件已经存在，其中的任何已存在的数据都会被删除

续表

方法	描述
writeData(*self*, *dataval*, *use_sn=False*, *use_ts=False*)	生成一个包含 ASCII 数据值的字符串。如果 *use_sn* 为 False，就不会往记录中插入序号。否则，该对象会维护一个序号计数器，并添加到该记录开头。如果 *use_ts* 是 Flase，就不会用到时间戳。否则，会获取一个时间戳，并用到输出字符串中

456 图 12-4 所示的是 ASCIIDataRead 和 ASCIIDataWrite 定义的对象是如何访问 ASCII 文件内容的。图中所示的是这样一种情况，输入文件（文件 1）包含着一些作用于系统的数据——或者说，这些值会被转换成模拟电压。响应数据被捕获并被 ASCIIDataWrite 对象存储在文件 2 和文件 3 中。

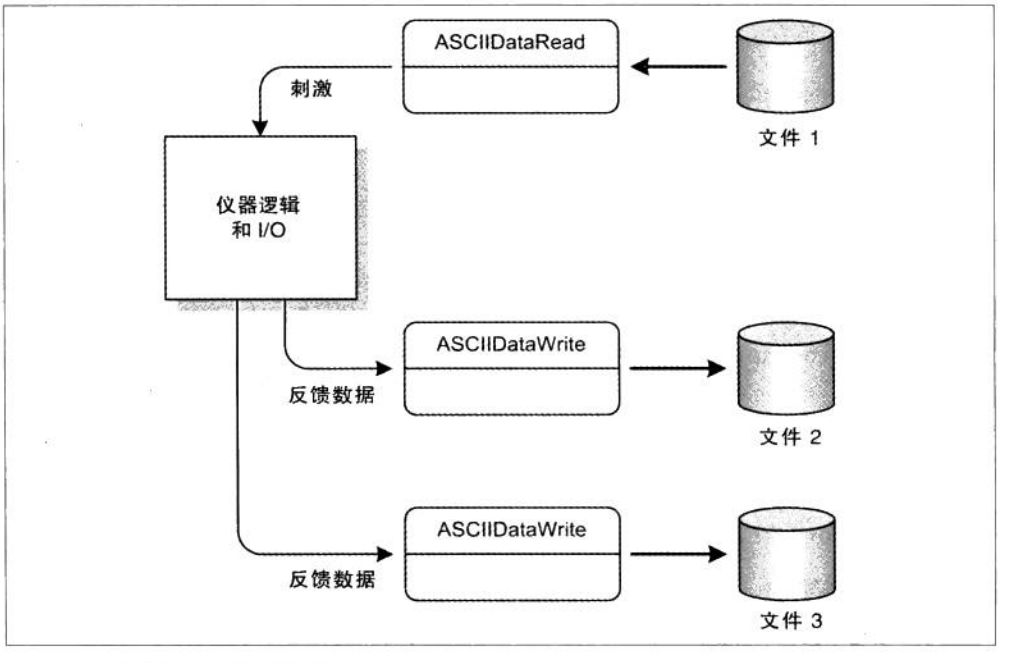


图12-4：使用ASCIIData类接收数据

下面是 FileUtils.py 的源代码：

```
#! /usr/bin/python

""" ASCII 数据文件 R/W 实用类

定义了两个类，使用文本文件来读取和写入 ASCII 数据记录。
```

模块中支持的方法有打开、关闭、读取和写入 ASCII 数据形式的单行记录。
ASCIIDataWrite 类处理数据记录写入，
ASCIIDataRead 类处理读取事务。
实例化对象维护自己的文件对象的引用，同时可以有多个实例被激活。

有 4 个可用的记录格式，如下所示：

4 个字段：[序号][日期][时间][数据]
3 个字段：[日期][时间][数据]
2 个字段：[序号][数据]
1 个字段：[数据]

457

要注意，时间戳实际上是两个字段：[数据][时间]。
2 个或 4 个字段的记录将包含序列号。
只有 3 个字段的记录将包含时间戳字段，但没有序列号。
所有的记录格式都必须包含 [数据] 字段，并且所有字段都作为字符串来被写入。

[数据] 字段总是以表示浮点值的字符串形式写入文件。
也就是说，写入整数时，小数点部分被设置为零。
"""

```
import  os

import  TimeUtils            # 时间及日期工具箱
import  RetCodes    as RC    # 共享出的返回码定义


class ASCIIDataWrite:
    """ 用以将 ASCII 格式的数据记录写入文件的方法。

        定义对象来将 ASCII 格式数据记录写入文本文件，
        每一个对象都是唯一的，允许同时使用多个对象
    """
    def __init__(self):
        self.seq_num  = 0
        self.file_ref = None


    def openOutput(self, path, file_name, reset_file=False):
        """ 打开用以输出 ASCII 数据的文件。

            如果没有指定路径（空字串作为路径）将尝试从当前目录打开文件。

            如果参数 reset_file 为 False，文件使用追加模式打开。
            为 True 将以写入模式打开，如果文件已经存在，则原先的数据将被清空。
        """
        rc = RC.NO_ERR

        if len(file_name) > 0:
            # 生成全路径名
```

```
            file_path = os.path.join(path, file_name)

            if reset_file:
                fmode = "w"
            else:
458             fmode = "a"

            try:
                self.file_ref = open(file_path, fmode)
            except Exception, e:
                rc = RC.OPEN_ERR
                print "%s" % str(e)
        else:
            rc = RC.NO_NAME

        return rc


    def closeOutput(self):
        """ 关闭已经打开的输出文件。

            如果文件没有打开，将返回错误。
        """
        rc = RC.NO_ERR

        if self.file_ref and self.file_ref != None:
            rc = closeFile(self.file_ref)
        else:
            rc = RC.NO_FILE

        return rc


    def writeData(self, dataval, use_sn=False, use_ts=False):
        """ 以 ASCII 格式生成包含数据的字串。

            如果 use_ts 为 False，则不追加时间戳，
            否则生成时间戳并附加到输出字串中。
        """
        rc = RC.NO_ERR

        if use_sn:
            # 需要初始化序列数？
            if self.seq_num == 0:
                self.seq_num = 1
            sn = "%02d " % self.seq_num
            self.seq_num += 1
        else:
            sn = ""
```

```
        if use_ts:
            ts = TimeUtils.getTS() + " "
        else:
            ts = ""

        hdr = sn + ts

        # 如果 self.file_ref 为 None，则此实例中的文件还未打开。
        if self.file_ref == None:
            rc = RC.NO_FILE

        # 如果出错了，不用继续处理。
        if rc == RC.NO_ERR:
            try:
                dstr = " %f" % float(dataval)
            except Exception, e:
                rc = RC.INV_DATA
                print "%s" % str(e)

            if rc == RC.NO_ERR:
                outstr = hdr + dstr + "\n"

                try:
                    self.file_ref.write(outstr)
                except Exception, e:
                    rc = RC.WRITE_ERR
                    print "%s" % str(e)

        return rc


class ASCIIDataRead:
    """ 定义对象，用于从标准的文本文件读取 ASCII 数据记录。
        每个对象都是唯一的，且任何时候允许多个对象同时使用。
    """
    def __init__(self):
        self.file_ref  = None


    def openInput(self, path, file_name):
        """ 打开文件用以 ASCII 数据输入。

            如果没有指定路径（空字串作为路径）将尝试从当前目录打开文件。
        """
        rc = RC.NO_ERR

        # 没指定路径时，从本地目录尝试。
        if path == None:
            path = './'
```

459

```
        # 生成全路径名
        file_path = os.path.join(path, file_name)

        try:
            self.file_ref = open(file_path, "r")
        except Exception, e:
            rc = RC.OPEN_ERR
            self.file_ref = None

        return rc


    def closeInput(self):
        """ 关闭打开的输入文件。

            如果文件没有打开，将返回一个错误。

            调用模块的 closeFile() 函式进行实际的关闭。
        """
        rc = RC.NO_ERR

        if self.file_ref and self.file_ref != None:
            rc = closeFile(self.file_ref)
        else:
            rc = RC.NO_FILE

        return rc


    def readDataRecord(self):
        """ 读取记录字符串并返回应该的格式。

            从文件一次读取一条记录，并返回整个记录字符串。
            空的记录字符串则返回 EOF。

            返回值是包含返回代码和记录的字符串组成的 2 元组。
        """
        rc = RC.NO_ERR

        # 检查文件中的读取
        if self.file_ref != None:
            # 从文件中提取一行
            try:
                record = self.file_ref.readline()
            except Exception, e:
                record = ""
                rc = RC.READ_ERR
        else:
            record = ""
```

460

```
            rc = RC.NO_FILE

        return rc, record


    def readDataFields(self):
        """ 从 ASCII 格式文件读取记录字段。

            从文件一次读取一条记录，并以列表形式返回每一个字段。
            记录列表为空，则返回 EOF。

            每个字段的数据根据记录的数字，从字符串转换为适当的数据类型。
                                                                    <461>
            返回值是包含返回代码和记录的字符串组成的 2 元组。
        """
        rc = RC.NO_ERR
        readflds = []
        retflds  = []

        # 从文件提取记录字串
        rc, recstr = self.readDataRecord()

        # 拆分为组件元素
        if rc == RC.NO_ERR:
            if len(recstr) > 0:
                readflds = recstr.split()

                # 使用 try-catch 事件来进行类型检验，
                # 并将整数转换为浮点数
                try:
                    if len(readflds) == 4:
                        retflds.append(int(readflds[0]))
                        retflds.append(int(readflds[1]))
                        retflds.append(readflds[2])
                        retflds.append(float(readflds[3]))
                    elif len(readflds) == 3:
                        retflds.append(int(readflds[0]))
                        retflds.append(readflds[1])
                        retflds.append(float(readflds[2]))
                    elif len(readflds) == 2:
                        retflds.append(int(readflds[0]))
                        retflds.append(float(readflds[1]))
                    elif len(readflds) == 1:
                        retflds.append(float(readflds[0]))
                    else:
                        rc = RC.INV_FORMAT
                except Exception, e:
                    print str(e)
                    retflds = []
                    rc = RC.INV_DATA
```

```
            else:
                rc = RC.NO_DATA

        return rc, retflds


    def getData(self):
        """ 只返回记录的数据部分。

            返回值是包含返回代码和记录的字符串组成的 2 元组。

            一条记录总是包含数据字段。
            此方法仅仅返回浮点数据格式的数据字段，
462         如果数据字段不存在，或是检索记录时发生错误，返回 None。
        """
        retdata = None

        rc, infields = self.readDataFields()
        if rc == RC.NO_ERR:
            # 此时 readDataFields() 正当地处理完毕，
            # 就知道有多少有效字段可以来干活了。
            if len(infields) == 4:
                retdata = float(infields[3])
            elif len(infields) == 3:
                retdata = float(infields[2])
            elif len(infields) == 2:
                retdata = float(infields[1])
            elif len(infields) == 1:
                retdata = float(infields[0])
        return rc, retdata


# 模块函式

def closeFile(file_id):
    """ 关闭打开的输入或输出文件。

        file_id 是 Python 文件对象的引用。
    """
    rc = RC.NO_ERR

    try:
        file_id.close()
    except Exception, e:
        rc = RC.INV_FILE
        print "%s" % str(e)

    return rc
```

```
if __name__ == "__main__":
    fout = ASCIIDataWrite()

    fin  = ASCIIDataRead()

    fout.openOutput("./", "futest.dat")

    fout.writeData(2.5, use_ts=True)
    fout.writeData(2.6, use_ts=True)
    fout.writeData(2.7, use_ts=True)
    fout.writeData(2.8, use_ts=True)
    fout.writeData(2.9, use_ts=True)
    fout.writeData(3.0, use_ts=True)

    fout.closeOutput()

    fin.openInput("./","futest.dat")

    print "Read Records"
    print "%d %s" % fin.readDataRecord(),
    print "%d %s" % fin.readDataRecord(),
    print "%d %s" % fin.readDataRecord(),
    print "%d %s" % fin.readDataRecord(),
    print "%d %s" % fin.readDataRecord(),
    print "%d %s" % fin.readDataRecord(),

    fin.closeInput()

    fin.openInput("./","futest.dat")

    print "Read Fields"
    print "%d %s" % fin.readDataFields()
    print "%d %s" % fin.readDataFields()
    print "%d %s" % fin.readDataFields()
    print "%d %s" % fin.readDataFields()
    print "%d %s" % fin.readDataFields()
    print "%d %s" % fin.readDataFields()

    fin.closeInput()
```

463

在 shell 命令行执行 FileUtils.py 后，你会看到下面的输出：

```
$ python Fileutils.py
Read Records
0 120117 16:42:20  2.500000
0 120117 16:42:20  2.600000
0 120117 16:42:20  2.700000
0 120117 16:42:20  2.800000
0 120117 16:42:20  2.900000
0 120117 16:42:20  3.000000
```

```
Read Fields
0 [120117, '16:42:20', 2.5]
0 [120117, '16:42:20', 2.6]
0 [120117, '16:42:20', 2.7]
0 [120117, '16:42:20', 2.8]
0 [120117, '16:42:20', 2.9]
0 [120117, '16:42:20', 3.0]
```

为了访问文件开始的第一条记录，对输入文件进行“倒带（rewind）”而不是关闭后重新打开，这应该说是一个不错的功能（但并非是必需的）。通过位置（第 *n* 条记录）序号或者时间日期范围等定位出某条特定记录，应该也是一个方便的功能。好了，如果你觉得需要，就去实现这些有趣的东西吧。

二进制数据文件

二进制文件通常比 ASCII 文件更适合用于数据存储。原因很简单，一个像 48373 这样用 16 位就可以表示的数值在 ASCII 中要占用 5 个字节，但换成二进制就只需要两个字节，
464
即十六进制数 BC F5。一个包含 16K（2^{14} 项）个不重复的值的文件，用 ASCII 表示，文件大小估计有 81920（82K）字节（假定每个值是 5 个字符），但是换成二进制格式只需要 32K（每个值占两个字节）。

二进制文件的缺点是它必须使用特定的软件来读取数据，虽然二进制文件可以使用二进制文件编辑器打开，但是编辑过程很麻烦，而且容易出错。所以最好使用“理解”文件内部结构且能高效访问数据的软件来进行操作。

对于图像数据、大型数字数组和混有二进制与 ASCII 值的结构化数据，二进制数据文件是首选存储格式。结构化二进制数据的压缩文件就是一种二进制文件。

平面二进制数据文件

和平面 ASCII 文件（flat ASCII file）一样，一个平面二进制文件由一条或多条记录组成。每条记录可以是固定长度的也可以是不固定长度的，不过后者往往需要做一些额外的工作来支持。

例如，你希望使用二进制文件将数据传送给另一个程序，而且这个程序能够将文件内结构化的数据直接解析为 C 或 C++ 结构体数据类型。为使这些二进制格式可传输，你需要将这些值从 Python 程序中取出来，加载到一个或多个结构体中，然后将结构体写入一个文件。

下面是一个简单的 C 结构体：

```
typedef struct {
    int     seq_num;
    int     chan;
    int     mode;
    double  data_val;
    char    stat_codes[3];
    } input_data;
```

在 C 程序中，将数据以二进制格式存储最直接的方法就是把这样的结构体直接写入文件，每个结构体都对应一条特定的记录。文件中通常还需要某些头部数据，其中至少要包含当前结构体的总数。图 12-5 所示的二进制文件结构示例展示了我们定义的这个结构体的数据在文件中可能的组织方式。

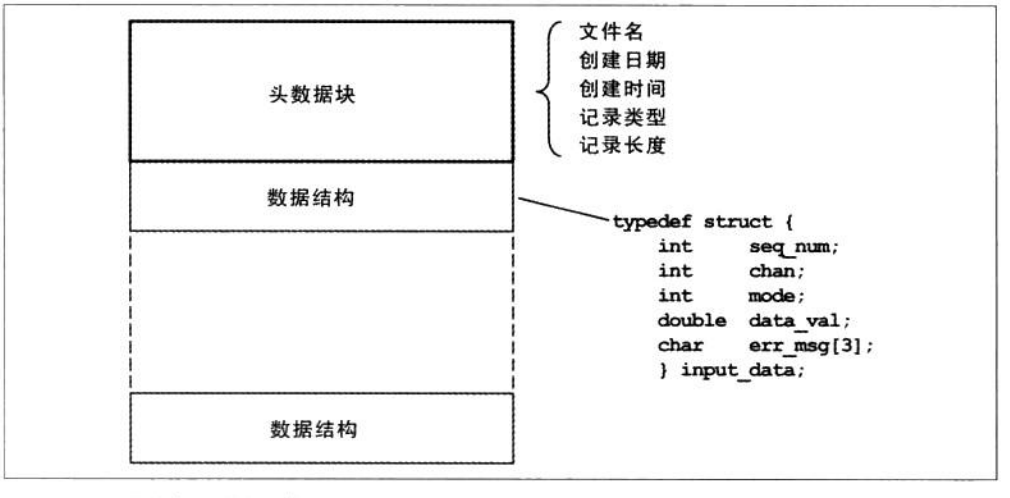


图12-5：二进制文件结构示例

不必被头部过多的信息吓到，这里只要知道有一个头部区域（有时称做块），而且其中包含了文件中结构体的总数就可以了。头部后面的每个子块都包含一个完整的二进制格式的结构体数据。

这为我们引入了关于填充的问题。计算机根据其寄存器寻址能力，有一个最佳的内存数据大小。例如，一个 32 位 CPU 最佳的内存数据大小可能是 32 位的，也就是说，在 CPU 和内存之间交换数据时，如果每块数据是 32 位的，运行效率最佳。很多 32 位 CPU 能够同样高效地处理 16 位的数据，但处理 8 位数据时需要做些额外的工作，效率上会有一定的损失。出于效率方面的考虑，定义结构体时最好能针对其大小进行 32 位或者 16 位对齐。 465

图 12-6 所示的结果为我们展示了图 12-5 所示的二进制文件结构示例中的结构体映射到

内存中的样子。留意一下其中的阴影部分，第一个阴影部分将16位整型变量对齐到32位。虽然这可能不是必需的，但它能够防止8个字节的双精度值超出32位。

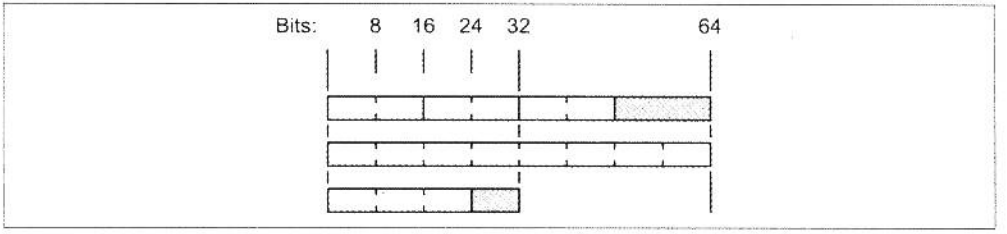


图12-6：结构填充

466

另外，图12-6所示的结果映射中，3个字节的stat_codes数组（在Python中是3个字符的字符串）也有一个扩展字节，使之按32位对齐。幸运的是，大多数编译器都能自动完成这些工作。Python中处理二进制数据的方法也内置了对齐处理的功能。

最后，你需要容忍的是，使用结构体记录的二进制数据文件的灵活性可能不高。也就是说，其中所存储的什么数据，以及数据是如何组织的，都是由相关记录的结构体明确定义的，尤其是用处理C或C++编写的应用程序。而通过一个ASCII数据文件，只要读取数据的程序足够智能，你就能存储任何能够使用ASCII表示的数据（这是使用CSV进行数据交换的一个优点）。

用Python处理二进制数据

Python库包括一些模块能很方便地处理C结构体格式的二进制数据。一个是ctypes，另一个是struct模块。我们先来看看ctypes是如何工作的，然后再重点研究一下struct库模块。

使用ctypes模块操作结构化二进制数据

如我们在第5章中所看到的那样，ctypes库模块通常被用来访问外部模块，而无须再使用C或者SWIG（见第14章的LabJack数据采集设备）之类的工具重新开发一个扩展模块。它也可以被用来在Python程序中创建并操作二进制数据对象。

下面这个例子ctypes_struct.py展示了如何使用ctypes模块中所定义的各种类型来创建一个二进制结构体对象，并为之赋值：

```
# ctypes_struct.py

import ctypes
```

```
class DataRecord(ctypes.Structure):
     _fields_ = [ ('seq_num', ctypes.c_short),
                  ('chan', ctypes.c_short),
                  ('mode', ctypes.c_short),
                  ('data_val', ctypes.c_double),
                  ('err_msg', ctypes.c_char * 3) ]

drec = DataRecord()

drec.seq_num = 1
drec.chan = 4
drec.mode = 0
drec.data_val = 2.355
drec.err_msg = '030'

print "seq_num : %d" % drec.seq_num
print "chan    : %d" % drec.chan
print "mode    : %d" % drec.mode
print "data_val: %f" % drec.data_val
print "err_msg : %s" % drec.err_msg
```

467

DataRecord继承自ctypes的Structure类。Structure类型的对象（还有联合类Union）必须定义一个名为_fields的属性，它是一个二元组的列表，每项中都包含一个字段名字和一个字段类型。对于err_msg字段，我用“* 3”来表示，用以存放字符串的3个连续的字符类型实例（我们很快会看到其重要性）。另外注意，在drec对象被实例化时并没有对其进行初始化。各字段的值随后用字段名显式指定。然而，也可以在对象实例化时通过向构造函式传入数据值参数来进行初始化，如：

```
# ctypes_struct2.py

import ctypes

class DataRecord(ctypes.Structure):
     _fields_ = [ ('seq_num', ctypes.c_short),
                  ('chan', ctypes.c_short),
                  ('mode', ctypes.c_short),
                  ('data_val', ctypes.c_double),
                  ('err_msg', ctypes.c_char * 3) ]

drec = DataRecord(1, 4, 0, 2.355, '030')

print "seq_num : %d" % drec.seq_num
print "chan    : %d" % drec.chan
print "mode    : %d" % drec.mode
print "data_val: %f" % drec.data_val
print "err_msg : %s" % drec.err_msg
```

这两种方式的输出是一致的。我更倾向于能自主指定字段的这个版本，因为它可以更有

效地防止给结构体元素分配一个错误的值，而且可读性更强。如果你选择使用构造函式初始化的方法，要保证传入参数的顺序和各字段的顺序保持一致。

接下来，我们来看看它是如何工作的。你可以通过下面的命令来执行 ctypes_struct.py 或者 ctypes_struct2.py：

```
python ctypes_struct.py
```

或：

```
python ctypes_struct2.py
```

其输出应该如下所示：

```
seq_num : 1
chan    : 4
mode    : 0
data_val: 2.355000
err_msg : 000
```

468

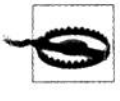
Python 中没有原生（native）的结构体类型。通过 ctypes Structure 类，你能够像在 C/C++ 里那样，在 Python 中创建并操控结构体对象。为什么要这样做呢？如果你将 Python 作为一个模板语言，最终要将程序转换为一个 C/C++ 代码的产品，那么，从一开始就使用结构体类对象将会让你的转换工作更加容易。此外，如果你想创建非常健壮的代码，那就应该避免使用字典（dictionary）对象（它会在不经意处发生改变），多考虑使用结构体对象（它一旦定义好就会固定下来）。最后，我们可以创建一个结构体对象，并将其写入一个二进制文件让其他应用读取，还可以通过网络套接字连接来发送结构体对象。同样，我们当然也可以从其他二进制文件或者网络套接字中提取其中的数据元素。

接下来要看到的示例脚本 ctypes_struct_file.py 会将二进制结构体中的内容写入一个文件，为了证明其确实起作用了，它还会将这些内容再从文件中重新读出来。

```
# ctypes_stuct_file.py

import ctypes

class DataRecord(ctypes.Structure):
    _fields_ = [ ('seq_num', ctypes.c_short),
                 ('chan', ctypes.c_short),
                 ('mode', ctypes.c_short),
                 ('data_val', ctypes.c_double),
                 ('err_msg', ctypes.c_char * 3) ]

drec = DataRecord()
```

```
drec.seq_num = 1
drec.chan = 4
drec.mode = 0
drec.data_val = 2.355
drec.err_msg = '030'

print "Written to structure:"
print "seq_num : %d" % drec.seq_num
print "chan    : %d" % drec.chan
print "mode    : %d" % drec.mode
print "data_val: %f" % drec.data_val
print "err_msg : %s" % drec.err_msg
print "\n"

# 输出二进制数据
fout = open('bindata.dat', 'wb')
fout.write(drec)
fout.close()

# 现在将其读回到一个新的 DataRecord 实例中
fin = open('bindata.dat', 'rb')

drec2 = DataRecord()
fin.readinto(drec2)
fin.close()

print "Read from structure:"
print "seq_num : %d" % drec2.seq_num
print "chan    : %d" % drec2.chan
print "mode    : %d" % drec2.mode
print "data_val: %f" % drec2.data_val
print "err_msg : %s" % drec2.err_msg
```

469



注意，该例中 3 个字符的字符串 err_msg 的表示方法定义在这些例子中。ctypes 有一个字符（字节）类型和一个字符串指针类型（ctypes.c_char_p）。如果你使用指针类型，就会有一个隐含的缓冲区用于存放字符串数据，结构体会包含一个地址，而不是一串字符值。当数据存在文件中时，这种方式是不适用的。

ctypes_struct_file.py 使用了一个过去在 Python 社区中有争议的方法，即 readinto() 文件对象方法。如果你向 Python 查询关于它的帮助文档，会得到下面的信息（针对 Python version 2.6.5）：

```
>>> help(file.readinto)
Help on method_descriptor:

readinto(...)
    readinto() -> Undocumented. Don't use this; it may go away.
```

然而，在版本 2.7 的文档中你会看到：

readinto(b)：

读取长度为 len(b) 字节到二进制数组 b，并返回读取的长度。

同 read()，多次读取时，可能对底层的原始数据流造成影响，除非后者是‘交互的’。

当底层数据流不是块模式时，会抛出 BlockingIOError 异常，此时不会有任何有效数据。

（基于 Python 标准库文档，基础 I/O 类）

同样的描述还出现在 Python 3.x 版本中，因此我相信它已经被接受而且短期内不会被移除。任何时候，如果现有的 Python 版本已经足以完成你的工作，你就不应该仅仅为了升级而升级。依赖于已经改变了的东西，就会引入突然崩溃的风险。就像那句老话：“如果它没有坏掉，就不要修正它。”

现在，我们来看看如何将一串结构体中的数据存入文件，再将它们读入到一个数组中。示例脚本 ctypes_struct_file2.py 展示了这个过程：

```
470  # ctypes_struct_file2.py

     import ctypes

     class DataRecord(ctypes.Structure):
         fields = [ ('seq_num', ctypes.c_short),
                    ('chan', ctypes.c_short),
                    ('mode', ctypes.c_short),
                    ('data_val', ctypes.c_double),
                    ('err_msg', ctypes.c_char * 3) ]

     drec = DataRecord()

     fout = open('bindata.dat', 'wb')

     # 向文件中写入 10 组 drec 结构对象
     for i in range(0, 10):
         drec.seq_num = i
         drec.chan = (i + 2)
         drec.mode = 0
         drec.data_val = (2.0 + (i/10.0))
         drec.err_msg = '030'

         # 输出二进制数据
         fout.write(drec)
```

```
fout.close()


print "Read from file:"

# 创建结构数组
# q 作为变量是一个虚拟的计数器变量
drec2 = [DataRecord() for q in range(0,10)]

# 现在将其读回到一个新的 DataRecord 实例中
fin = open('bindata.dat', 'rb')

for i in range(0, 10):
    try:
        rc = fin.readinto(drec2[i])
    except:
        pass
    else:
        if rc > 0:
            print "rec num : %d" % i
            print "rec size: %d" % rc
            print "seq_num : %d" % drec2[i].seq_num
            print "chan    : %d" % drec2[i].chan
            print "mode    : %d" % drec2[i].mode
            print "data_val: %f" % drec2[i].data_val
            print "err_msg : %s" % drec2[i].err_msg

fin.close()
```

脚本的第一部分生成了 10 个结构体对象 drec 的实例，并将其写入一个文件。接下来，关闭该文件，然后重新打开读取数据。用一个循环通过 readinto() 方法导出每个结构体实例，再将数据加载到一个结构体数组中。计数器到达限制的次数时，读数据的循环会结束，并产生一个异常或返回零。异常会被默认方法处理，虽然 readinto() 方法也有一些与异常相关的方法（可以从 Python 库文档了解到更多细节）。

471

使用 struct 来操作结构化二进制数据

Python 中的 struct 模块的功能与我们所见到的 ctypes 类似。

和 ctypes 一样，struct 用于方便地处理二进制数据结构体，而无须引入其他额外的定制扩展工作。你可以在 Python 标准库文档的第 7 章（String Services）中找到更具体的介绍。和 ctypes 不同的是，struct 将二进制数据作为二进制值的字符串来处理（这可能也是为什么标准库要将其放到 String Services 部分来介绍的原因）。

struct 模块提供了 5 个函式，见表 12-4。

表12-4：struct函式

函式	描述
pack(fmt, v1, v2, ...)	返回一个指定格式（fmt 参数）的二进制值的字符串
pack_into(fmt, buffer, offset, v1, v2, ...)	按指定格式对数据打包，然后将打包后的数据从指定的 offset 位置写入 buffer
unpack(fmt, string)	按照指定格式，从一个打包的字符串中解压数据，并返回一个包含解压后数据的 *n* 元组（*n*-tuple）
unpack_form(fmt, buffer[, offset=0])	按照指定格式，由偏移位置 offset 开始，从 buffer 中解压数据
calcsize(fmt)	返回指定格式结构体的大小（例如，打包的二进制字符串）。或者说，calcsize() 计算指定格式结构体的长度

除了表 12-4 中所列出的函式之外，还有一个提供了一组等价方法的 Struct 类，它包含两个附加的对象属性：format 和 size。Struct 类的方法没有格式参数，而是在对象实例化时设置该其 format 属性。size 属性表示 format 所指定格式打包数据的大小。在 Python 提示符中请求 Struct 对象帮助，会得到下面的输出结果（为适应页边距对文字排版略作调整）：

472

```
class Struct(object)
| Compiled struct object
|
| Methods defined here:
|
| __delattr__(...)
|     x.__delattr__('name') <==> del x.name
|
| __getattribute__(...)
|     x.__getattribute__('name') <==> x.name
|
| __init__(...)
|     x.__init__(...) initializes x; see x.__class__.__doc__
|     for signature
|
| __setattr__(...)
|     x.__setattr__('name', value) <==> x.name = value
|
| pack(...)
|     S.pack(v1, v2, ...) -> string
|
|     Return a string containing values v1, v2, ... packed
|     according to this Struct's format. See struct.__doc__
|     for more on format strings.
|
| pack_into(...)
|     S.pack_into(buffer, offset, v1, v2, ...)
```

```
|
|     Pack the values v1, v2, ... according to this
|     Struct's format, write the packed bytes into the
|     writable buffer buf starting at offset. Note that
|     the offset is not an optional argument.
|     See struct.__doc__ for more on format strings.
|
| unpack(...)
|     S.unpack(str) -> (v1, v2, ...)
|
|     Return tuple containing values unpacked according to
|     this Struct's format. Requires len(str) == self.size.
|     See struct.__doc__ for more on format strings.
|
| unpack_from(...)
|     S.unpack_from(buffer[, offset]) -> (v1, v2, ...)
|
|     Return tuple containing values unpacked according to
|     this Struct's format. Unlike unpack, unpack_from can
|     unpack values from any object supporting the buffer
|     API, not just str.
|     Requires len(buffer[offset:]) >= self.size.
|     See struct.__doc__ for more on format strings.
|
| ----------------------------------------------------------
| Data descriptors defined here:
|
| format
|     struct format string
|
| size                                                        <473>
|     struct size in bytes
|
| ----------------------------------------------------------
| Data and other attributes defined here:
|
| __new__ = <built-in method __new__ of type object>
|     T.__new__(S, ...) -> a new object with type S, a subtype of T
```

ctypes模块定义了数据对象类型，而struct模块则使用格式编码。这些格式编码列在表12-5中。

表12-5：struct数据类型格式编码

格式编码	C类型	Python类型	打包后的大小
x	pad byte	no value	1
c	char	长度为1的string	1
b	signed char	integer	1

续表

格式编码	C 类型	Python 类型	打包后的大小
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8
s	char[]	string	Variable
p	char[]	string	Variable
P	void*	integer	OS dependent

要了解更多关于格式编码的细节内容请参阅 Python 标准库文档。

现在，我们来创建一个打包的二进制字符串，其对应的结构体定义和我们之前在 ctypes 的例子中见到的相同：

```
# pack_struct.py

import struct
import binascii
import ctypes

474
# 原生数据结构定义
# _fields_ = [ ('seq_num', ctypes.c_short),
#              ('chan', ctypes.c_short),
#              ('mode', ctypes.c_short),
#              ('data_val', ctypes.c_double),
#              ('err_msg', ctypes.c_char * 3) ]
#
# 等价的结构格式字符串：
# 'hhhd3s'

seq_num = 1
chan = 4
mode = 0
data_val = 2.355
err_msg = '030'
```

```
srec = struct.pack('hhhd3s', seq_num, chan, mode, data_val, err_msg)

# 使用 binascii.hexlify 可以看到二进制字符串中有什么
print binascii.hexlify(srec)
```

运行 pack_struct.py，你应当能看到下面的结果：

python pack_struct.py

```
0100040000000000d7a3703d0ad70240303330
```

大多数 struct 格式编码都会有一个计数或者大小的前缀。在 pack_struct.py 中，字符串格式的 3s 表示含有 3 个元素的字符串。非字符串格式编码，像 3h 和 'hhh' 意思相同。虽然复合格式字符串也能够识别这种表达，但将格式字符串表示为 '3hd3s' 而不是 'hhhd3s' 更益于提高可读性。

如果你更喜欢用 Struct 类，那么就要注意，在对象实例化时，Struct 的构造函式只接受一个参数，就是格式字符串。随后格式字符串被记录在 format 属性中。脚本 pack_struct_obj.py 使用 Struct 类来实现和 pack_struct.py 相同的结果：

```
# pack_struct_obj.py

import struct
import binascii

# 原生数据结构定义
# _fields_ = [ ('seq_num', ctypes.c_short),
#              ('chan', ctypes.c_short),
#              ('mode', ctypes.c_short),
#              ('data_val', ctypes.c_double),
#              ('err_msg', ctypes.c_char * 3) ]
#
# 等价的结构格式字符串：
# 'hhhd3s'

seq_num = 1
chan = 4
mode = 0
data_val = 2.355
err_msg = '030'

datavals = (seq_num, chan, mode, data_val, err_msg)
sobj = struct.Struct('hhhd3s')
srec = sobj.pack(*datavals)

print binascii.hexlify(srec)
```

475

在继续之前，再来看最后一个例子，通过 pack_struct_file.py 这个例子，我们可以看到 struct 和 ctypes 也可以同时工作：

```
# pack_struct_file.py

import struct
import binascii
import ctypes

# 原生数据结构定义
# _fields_ = [ ('seq_num', ctypes.c_short),
#              ('chan', ctypes.c_short),
#              ('mode', ctypes.c_short),
#              ('data_val', ctypes.c_double),
#              ('err_msg', ctypes.c_char * 3) ]
#
# 等价的结构格式字符串：
# 'hhhd3s'

seq_num = 1
chan = 4
mode = 0
data_val = 2.355
err_msg = '030'

srec = struct.pack('hhhd3s', seq_num, chan, mode, data_val, err_msg)

print binascii.hexlify(srec)

fout = open('bindata.dat','wb')
fout.write(srec)
fout.close()

# 现在将其读回到一个新的 DataRecord 实例中
class DataRecord(ctypes.Structure):
    _fields_ = [('seq_num', ctypes.c_short),
                ('chan', ctypes.c_short),
                ('mode', ctypes.c_short),
                ('data_val', ctypes.c_double),
                ('err_msg', ctypes.c_char * 3)]

fin = open('bindata.dat', 'rb')

476

drec = DataRecord()
fin.readinto(drec)
fin.close()

print "Read from structure:"
print "seq_num : %d" % drec.seq_num
```

```
print "chan    : %d" % drec.chan
print "mode    : %d" % drec.mode
print "data_val: %f" % drec.data_val
print "err_msg : %s" % drec.err_msg
```

这个例子的前半部分，我用到了 struct 的 pack 函式，用来创建一个打包的二进制字符串并将其写入一个文件。后半部分重新打开该文件，然后将其中的数据读取到一个 ctypes 结构体中。运行 pack_struct_file.py 会看到下面的结果：

```
python pack_struct_file.py
0100040000000000d7a3703d0ad70240303330
Read from structure
seq_num : 1
chan    : 4
mode    : 0
data_val: 2.355000
err_msg : 030
```

好了，这很有趣，而且现在你的 Python 工具箱里又多了两个处理二进制数据的工具。选用哪种技术取决于你所要完成的工作，以及你的编程风格。

图像数据

现在我打算研究另一种经常碰到的二进制数据格式：图像。一个包含代表多少光能穿越的二维数组通常被称为“亮度图”（luminance map），例如光能穿越 CCD（电荷耦合器件）传感器（可在普通的 CCD 照相机中找到）。你也可能遇到术语“强度图”（intensity map），它涉及图片数据，并且是一个描述某些东西穿越数据阵列的密度比较通用的术语。“强度图”广泛应用于数据描述应用中，比如，一个城市烟雾的密度图，或者印制板上点阵格子的温度，或者医院里 X 光机探测器阵列的强度水平。这些通常都不是图片，但这些密度可以被转化成灰度级图片格式（或者甚至颜色，具体依赖于应用和数据类型），并被人们查看和解读。

下面来看看图片的几何学。图片是一个由像素定义长度和宽度的二维数组，整体尺寸是用长度和宽度的乘积描述的。也存在第三维度，是用比特来描述的数目尺寸，用来描述数组中每一个 x 和 y 像素点的亮度。一些灰度图中每个像素用 8 位来描述数据，有些用 16 位。

通常你会看到用 8bpp 和 16bpp 来表述像素数据尺寸。这些只是表述每个像素 n 比特的速记方法。n 可以是 8、16 或者其他（一些彩色图片格式每个像素点使用 24bpp 或者 32bpp 来描述）。长度、宽度和像素尺寸参数构成了一个图片的几何描述。 477

在仪器和控制系统中，通常没有许多图片数据，但如果把数据处理成强度图的形式，往往会有意想不到的效果。例如，如果应用中包含堆叠式蜂房的温度数据，比如长度和宽

度都是 5，数据可以被实时映射成“强度图”来描述最热和最冷的蜂房盒子，或者也可以指示堆叠蜂房的温度变化。蜜蜂产生热量，天热时你会看到它们站在蜂房门口煽动它们的翅膀来把凉爽空气引入蜂房内部。每一个蜂房或许可以用一个 50 × 50 的像素块来描述，然后给每一个蜂房指定一个亮度密度比。结果将是一个 250 × 250 像素点的图片，如图 12-7 所示。

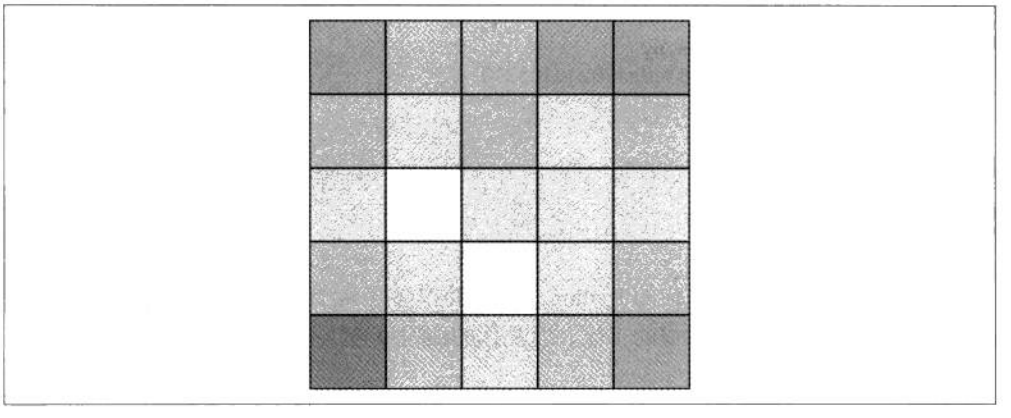


图12-7：数据显示为图像

如果一个应用产生了一系列的这类图片，例如，一天 24 张，那么这些图片可以被串联成电影来表述温度的变化。用静态数据或者数字表格则很难达到如此直观的效果。

PGM

应用中有很多种图片格式，有些格式在我们日常浏览网页时会很常见（PNG、JPEG 和 GIF 是网页中不可或缺的一部分）。其他格式可能不那么常见，但是它们应用于特殊应用中，Netpbm 图片格式就是其中之一。

478 PGM 是 Netbpm 组件中的一种格式，PGM 是一种最简单的图片数据格式。PGM 格式文档中这样描述：“PGM 格式是一种最简单的灰度级文件格式。它的设计理念是方便学习和编程。（它如此简单以至于很多人可以轻松逆向工程分析它的格式，因为这要比读它的规格文档还要容易）。”你可以在如下网址找到 PGM 格式的规格说明书：*http://netpbm.sourceforge.net/doc/pgm.html*。

PGM 主要的优点是它的简单。它可以用来存储灰度级（luminance map）图片数据。但是它也可以被用来存储其他类型的 8 位或者 16 位数据。数据也不局限于二维数组。同时，PGM 格式规格书指出，它可以用非图像数据来创建“合成图像”。Netpbm 工具可以转换数据为可视化的东西，例如 JPEG 图片。

虽然 PGM 格式最早设计成用 8 位来描述数据，但是最近版本的 PGM 格式可支持的最大数据尺寸是 65535，这允许你使用 16 位来描述每个数据元素或者像素点。对一般图片应用来说已经足够，但是对随机数据存储来说可能有些局限性。

PGM 文件结构

PGM 格式极其简单，很容易创建 PGM 文件并且非常容易阅读。一个 PGM 文件包含头部分，头部分可以携带注释，也可以不携带，紧跟头部分的是数据部分。图 12-8 描述了一个 PGM 文件的内部格式。

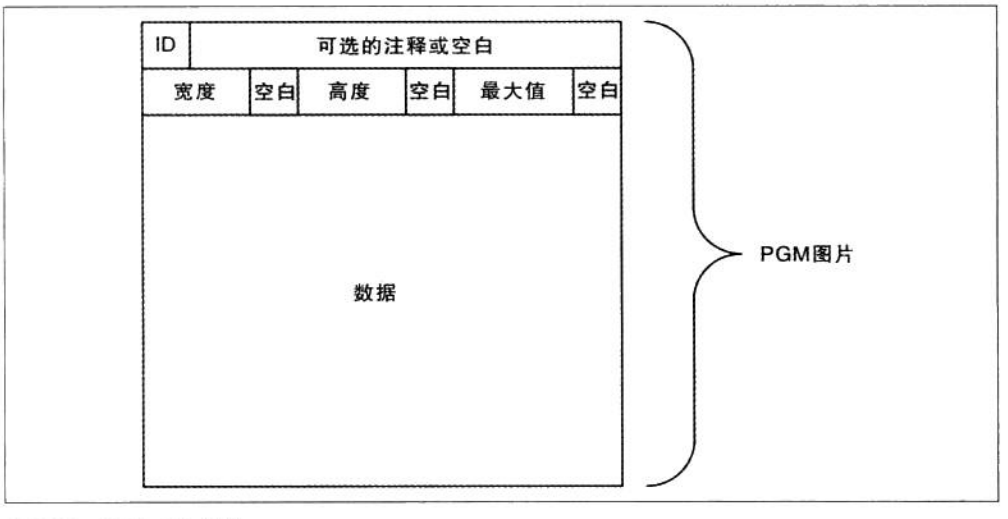


图12-8：PGM文件结构

我喜欢 PGM 格式是因为文件头部分的所有字符都是可以打印的 ASCII 字符。在我过去的一个项目中曾使用 PGM 文件格式来存储远端图像数据。当数据被写到文件时（一个图片一个文件），自动生成注释，内容包括时间戳、图片全名、设备名称以及各种设置（曝光时间、使用的滤镜、摄像头的 X 和 Y 指向位置等）的数值。这些让我们可以扫描图片目录然后抓取所有带有 # 注释符号的行。一个简单的脚本被用来创建美好格式的图片日志。

479

pgmtest.py 是一个简单的脚本例子，它可以创建一个包含随机数据的 256 像素 ×256 像素的 PGM 图片。

```
# pgmtst.py
# 生成 8bpp 图片的随机像素值
```

```
import random as rnd

# print ID string (P5)
# print comments (if any)
# print width
# print height
# print size
# print data

#rnd = random
rnd.seed()

width  = 256
height = 256
pxsize = 255

# 创建 PGM 头部
hdrstr = "P5\n%d\n%d\n%d\n" % (width, height, pxsize)

pixels = []
for i in range(0,width):
    for j in range(0,height):
        # 生成随机数
        pixval = int(255 * rnd.random())
        # 修订是 256 的值
        if pixval > pxsize:
            pixval = pxsize
        #endif
        pixels.append(pixval)
    #endfor
#endfor

# 转换列表为字符串值
outpix = "".join(map(chr,pixels))

# 将“图片”追加到头部之后
outstr = hdrstr + outpix

480 # 输出到硬盘
fimg = open("pgmtest.pgm","w")
fimg.write(outstr)
fimg.close()
```

当脚本被执行时，结果将保存在 *pgmtest.pgm* 文件中。你可以使用 ImageJ（可以从 *http://rsbweb.nih.gov/ij/* 下载）工具来查看此文件。当打开文件时，你可以看到如图 12-9 所示的图片。

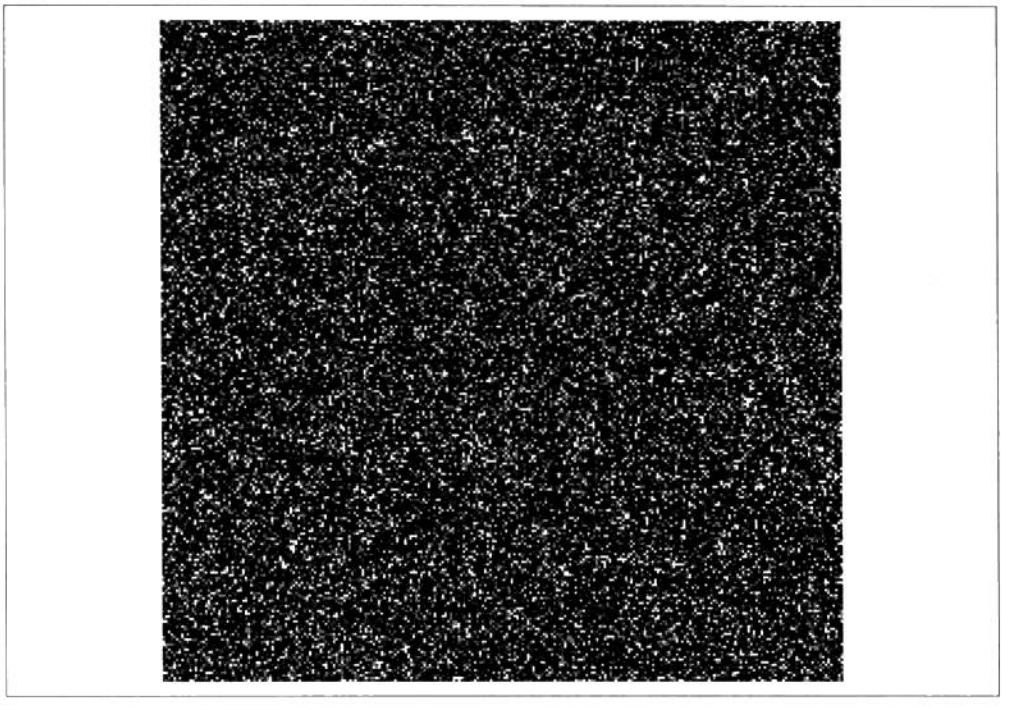


图12-9：8bpp样例PGM图像

我必须承认，图 12-9 所示的图像不是很令人兴奋（除非你喜欢看老式模拟信号电视上的雪花）。但是对代码稍作修改，就会使图片变得有趣了。这是为什么我喜欢用 PGM 格式来完成那些需要快速但不需要太清晰的显示工作——创建一个 PGM 图片文件不需要太细致的工作。

pgmtest.py 中的嵌套循环生成了真正的图片。一旦二维矩阵被创建，它就会被 join() 和 map() 方法添加到图片数据列表对象中：

```
outpix = "".join(map(chr,pixels))
```

这段代码迭代数组中的所有像素，映射每一个整型数据到一个字符型数据，然后把结果 481
中一个字符的字符串加入到 outpix 链表里。Python 文档描述内嵌 map() 方法如下（Phtyon 标准库文档，第二部分的“内嵌函式”）：

map(*function*, *iterable*, ...):

将 iterable 中的元素传入 function 中调用该函式，并将所有结果放入一个列表中返回。如果额外的 iterable 参数被传入，function 必须接收这么多参数，然后并

行使用所有 iterables 中的参数。如果一个 iterable 比另一个短，那就用 None 元素扩展它。如果 *function* 是 None，就假定传入了一个设返回参数自身的函式（identity function）；如果有多个参数，map() 就返回一个由元组组成的列表，每个元组中包含了各 iterables 中对应的元素（一种转置操作）。iterable 参数可以是一个序列或者任何可迭代的对象，返回值总是一个列表。

你可能注意到这种方法有一个缺陷。map(chr, pixels) 的结果是一个单字节字符串，它只能处理 0 到 255 之间的 8 位值。那么如何处理 16 位数据呢？

解决方式是使用 struct 来创建 16 位数据。下面的例子是一个完整的 PGM 图片输出函式。它使用一个 8 位或 16 位的数据元素列表，然后创建一个简单的 PGM 图片。

```
# PGMWrite image generator Ntility function
import struct

def PGMWrite(imgsrc, imgname, filename, width, height, bitdepth=8):
    """ 从捕获数据中生成 8bpp 或是 16bpp 的 PGM 图片。

        参数：

        imgsrc:     图片源数据（整数列表）
        imgname:    图片名，用以写入文件头部
        filename:    输出的图片文件名
        width:      图片宽度
        height:     图片高度
    """

    # 检验源数据格式
    if type(imgsrc) != list:
        print "Input data must be a list of integer values"
        return

    # 检验像素深度
    sizemult = 0
    if bitdepth == 8:
        sizemult  = 1
        img_depth = 255
    elif bitdepth >= 9:
        sizemult  = 2
        img_depth = 65535
    else:
        print "Invalid pixel depth"
        return

482 # 生成图片参数
    img_height = height
```

```
img_width  = width
img_size   = img_height * img_width
data_size  = img_size * sizemult

# 初始化图片输出列表
pixels = []

# 从输入数据生成图片列表
i = 0   # 输入索引
# 从图片列表加载数据
for y in range(0, img_height):
    for x in range(0, img_width):
        # 如果 mod 为 0，从数据源提取下一数值
        inval = imgsrc[i]

        if (bitdepth == 8) and (inval > 255):
            pixval = 255
        elif inval > 65535:
            pixval = 65535
        else:
            pixval = inval

        i += 1
        pixels.append(pixval)

if bitdepth == 8:
    pix_data = "".join(map(chr,pixels))
else:
    pix_data = pixels

# 加载头数据值，并获得字符串长度
img_type_str    = "P5\n"
img_name_str    = "#%s\n" % imgname      # 标准图片头部
img_width_str   = "%d\n"  % img_height
img_height_str  = "%d\n"  % img_width
img_depth_str   = "%d\n"  % img_depth

img_name_len    = len(img_name_str)
img_width_len   = len(img_width_str)
img_height_len  = len(img_height_str)
img_depth_len   = len(img_depth_str)

# 创建图片头结构
hdrvals  = (img_type_str, img_name_str, img_width_str,
            img_height_str, img_depth_str)

hdrobj = struct.Struct('3s %ds %ds %ds %ds'% (img_name_len,
                                              img_width_len,
                                              img_height_len,
```

```
                                                    img_depth_len))
        # 创建像素数据
        if sizemult == 1:
            pixobj = struct.Struct('%dc'% (img_size))
483     else:
            pixobj = struct.Struct('%dH'% (img_size))

        # 将数据推入结构
        img_hdr_data = hdrobj.pack(*hdrvals)
        img_pix_data = pixobj.pack(*pix_data)
        img_data = img_hdr_data + img_pix_data

        # 现在将所有东西写入文件
        fimg = open(filename,"wb")
        fimg.write(img_data)
        fimg.close()


if __name__ == "__main__":
    # 生成 8bpp 图片
    datavals = []
    for i in range(0, 65536):
        datavals.append(i/256)

    PGMWrite(datavals, "incshade8", "incshade8.pgm", 256, 256, bitdepth=8)

    # 生成 16bpp 图片
    datavals = []
    for i in range(0, 65536):
        datavals.append(i/256)

    PGMWrite(datavals, "incshade16", "incshade16.pgm", 256, 256, bitdepth=16)
```

图 12-10 展示了以 256 的公约数作为输入的 16bpp 输出（我们稍后会讲到）。

有很多函式值得讨论。我先说一下模块底部的 PGMWrite() 代码调用。

生成的两幅图像，像素大小分别是 8 位和 16 位的。两种情况都创建了一个包含一串递增值的数组（列表）。注意，从 for 循环中得到的值被存入数组前，都被 256 整除过。这个操作能将数据“缩放”到图片尺寸内。你可以试着将值改为 128 或 16 的值，看看会出现什么结果。其结果可能出乎你所料，花一点时间来理解这些行为是值得的。你还可以使用其他类型的陈列数组作为输入，来生成网络图案、目标图像、压缩性能网格，甚至一个房间中的温度或湿度在一年中的变化，等等。

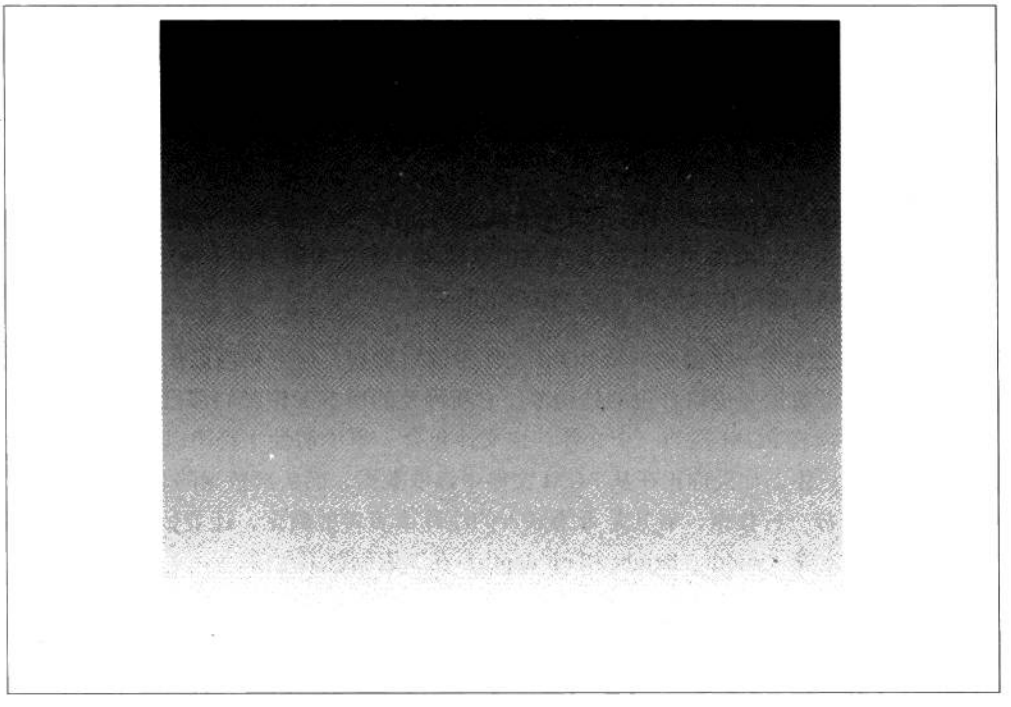


图12-10：16bpp样例PGM图像

现在让我们看看 PGMWrite() 本身。该函式一开始先检查输入数组的类型，并确保位宽参数大于等于 8。在一个产品应用中，这个函式可能被用来操作从相机或其他成像数据源得到的以 0 填充的 10 位或 12 位数据，所以将任何大于 8 位的数据当做 16 位数据更为方便。 484

接下来是从输入数组中读出并存入输出数组的图片参数。输出数组通过两个嵌套的 for 循环创建。这么做并不是必须的，但方便修改，如需要支持重新抽样、按比例对比等功能时。

在内部循环中（x 轴的那个），有一个用以确保像素值不会超出所使用图片类型所允许范围的检测。任何值只要超过了这个范围，就会被设置为该范围内的一个最大值。也就是说，如果你传入的数组中包含大于最大值的值，这些像素会直接在图片上显示成白色。基本上，它算是一种防御性编程技术。

图像数据生成后，该函式会创建其头部的字符串信息并计算其长度。当头部数据结构对象的格式字符串被创建后，会用这些值对相应区域的大小进行裁剪。 485

接下来就是图像数据本身了，它是纯二进制数据。其头部只有一个结构定义，但针对两种图像有不同的定义：一个是 8bpp 图像，另一个是 16bpp 图像。

头部数据的结构字符串再将图像数据的字符串连接在一起。它之所以能够工作，是因为这些数据是封装成字符串的，Python 能很好地将其合并到单个字符串对象中。

最后，`PGMWrite()` 打开一个指定的文件，并写入该结构体。结果是一个完整的 PGM 图像文件，如图 12-8 所示。

虽然你可能永远不需要处理图片，但这个函式还是提供了一些很不错的选择。PGM 格式能够可以被用来存储除图片之外的其他二进制数据，而且无须采用 *x* 轴、*y* 轴图像数组的加载方案，你只需简单地使用一个一维数组即可。

我希望读者自己实现一个函式，从 PGM 文件中读取数据，这个相当简单。我已经展示过如何处理头部数据并提取打包的结构体。它所涉及的所有工作我们都已经提到过。你的读取函式只需将 PGM 视为一个文件，该文件包含一串说明性字符串以及紧随其后的二进制数据。记住，如果你正在从 PGM 文件中读取数据，而头部数据的总数无法得知，你就需要检查每一行数据，看其是否为嵌入的注释或者参数数据，还要留意 PGM 头部有三个精确的字段（width、height 和 bit depth）在二进制数组之前。

小结

本章中我们查看了 ASCII 和二进制数据，以及一些相关应用。我们也了解到 Python 的内建函式和库函式如何转换数据，以及在同一脚本内如何混合两种数据类型来创建复杂的数据对象，比如图片。最后，我们检查了一个产生二进制图片数据的函式例子，并看到如何应用之前介绍的知识来创建相应函式。

推荐阅读

如果你想了解更多关于读写数据文件信息的话，下面的文章和书籍都是不错的参考资料。

RFC20, " ASCII Format for Network Interchange." Vint Cerf, 1969.

早期 RFC 文档位于 *http://www.faqs.org/rfcs/rfc20.html*。

486

RFC 4810, " Common Format and MIME Type for Comma-Separated Values (CSV) Files." Y. Shafranovich, Network Working Group, The Internet Society, 2005.

RFC 4810 定义了 CSV 文件的“text/csv” MIME 类型所用的格式，见 *http://tools.ietf.org/html/rfc4180*。

PEP 305, " CSV File API." K. Altis et al., python.org, 2008.

PEP 305 定义了 Python 的标准 CSV 模块的 API。作为 Python 标准库文档的补充资料，它相当有用，见 *http://www.python.org/dev/peps/pep-0305/*。

Beautiful Data: The Stories Behind Elegant Data Solutions. T. Segaran and J. Hammerbacher (eds.), O'Reilly Media, 2009.

搜集了一些有趣的文章和科技论文，内容涵盖了数据采集、处理、可视化领域等方面的话题。当你需要寻找一种呈现复杂格式数据的方式时，可以从这本书中获取一些思路。

Python Essential Reference, 4th ed. David M. Beazley, Addison-Wesley, 2009.

尽管这本书不适合一个 Python 新手，但是这本书是一个关于 Python 语言和库以及特性的很好的参考书。例如 generators、coroutines、closures、metaclasses，以及 decorators。此书也涵盖了 Python 文档不曾完全描述的底层方法和操作。

最后，维基百科有一篇关于配置文件的文章 *http://en.wikipedia.org/wiki/Configuration_file*（包含很多有趣的历史轶事）。还有一篇位于 *http://en.wikipedia.org/wiki/INI_file*。

第13章 用户界面

487

最小惊讶原则：尽可能使用户界面具有一致性和可预测性。

—Anonymous

除非一个应用是非常底层的，或是被专门设计成一个后台运行的程序，否则，它就可能需要某些类型的用户界面。在本章中，我们将对各种用户交互方法进行探究。一开始，我们会看看只用命令行能完成什么样的工作。接着研究如何使用一个符合ANSI标准的终端模拟器程序来显示数据、接收输入，然后研究Python的curses屏幕控制软件包。随后，我们将进入色彩明亮、图形精美的图像和对话框的世界，研究标准Python提供的TkInter GUI（图形用户界面）工具箱。最后，我们还将简要介绍wxPython GUI软件包。

文本界面

对所有带显示器的计算机而言，文本界面是最基础的用户界面。我之所以强调带有显示器，是因为从技术角度来看，最早的界面是布满了指示灯和开关的控制板，随后出现了打印机终端，但直到CRT出现之后，人机界面（HMI）设备才开始盛行起来。

控制台

与Python等程序进行交互，最直接的方式就是控制台。在Windows下有所谓的“DOS工具箱”，即大家所熟悉的“命令行提示符”（Windows命令行处理程序是*cmd.exe*）。在UNIX或Linux系统上，控制台即shell命令行，如*sh*、*bash*、*ksh*、*tcsh*或者其他你正在使用的shell界面。如果没有启用窗口管理器，整个屏幕就是一个控制台，但它只在一个窗口形状的区域工作。以字符串的形式向shell发送数据很简单，只需调用print语句就行。但在Python中获取用户输入却不像想象中那么简单，主要是因为Python没有一

488

个跨平台的 getch() 或 getche() 方法（分别是取字符和取字符并回显）。不过，其内置方法 raw_input() 可满足大多数情况下处理一般用户输入的需求，并且能够运行于所有支持 Python 的平台。我接下来还会介绍如何使用 *ctypes* 以及同时兼容 Linux 和 Windows 平台的 C 库函数，但在此之前，我们先来看看控制台都能完成哪些工作。

控制台显示示例

尽管你随时可以用临时性的 print 语句在控制台上输出数据，但当数据在屏幕上滚过时，用户却很难搞清楚屏幕上到底在输出些什么东西。建立一个用于数据显示的模板并使用固定的格式，可以让数据更易读，并能极大地减少用户受挫的可能（你自己也可能是用户）。

如果想在控制台上使用固定格式的数据，那你至少要做两件事：清空屏幕和使用一个模板来包含你的数据。我们先来看看模板。

如果不用 ANSI 终端控制字符串（见下一节），程序没有任何办法在控制台窗口中对光标进行定位。每行显示都需要考虑其自身的定位和数据显示格式。如下面的 console1.py 所示：

```
#! /usr/bin/python
# console1.py
#
# 使用内建的 print 方法演示控制台输出。
#
# 源代码出自 "Real World Instrumentation with Python"
# by J. M. Hughes, published by O'Reilly.

import time

ainstat   = ['ACTIVE','OFF','ACTIVE','OFF']
aindata   = [0.0, 0.0, 0.0, 0.0]
discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
discin    = [0,1,0,0,0,0,0,0]

print '\n' * 50
print ""
print "System Status"
print ""
print "Analog Input 0  : %10s  %f" % (ainstat[0], aindata[0])
print "Analog Input 1  : %10s  %f" % (ainstat[1], aindata[1])
print "Analog Input 2  : %10s  %f" % (ainstat[2], aindata[2])
print "Analog Input 3  : %10s  %f" % (ainstat[3], aindata[3])
print ""
print "Discrete Input 0: %3s  %d" % (discstate[0], discin[0])
print "Discrete Input 1: %3s  %d" % (discstate[1], discin[1])
```

489

```
print "Discrete Input 2: %3s  %d" % (discstate[2], discin[2])
print "Discrete Input 3: %3s  %d" % (discstate[3], discin[3])
print "Discrete Input 4: %3s  %d" % (discstate[4], discin[4])
print "Discrete Input 5: %3s  %d" % (discstate[5], discin[5])
print "Discrete Input 6: %3s  %d" % (discstate[6], discin[6])
print "Discrete Input 7: %3s  %d" % (discstate[7], discin[7])
print ""

time.sleep(2)
```

竖直方向上的位置由每行数据打印到控制台上的顺序决定，而水平方向的定位由每一行的格式来决定。紧接在 4 个变量初始化之后的代码行，`print "n" * 50`，通过打印 50 个空行来模拟一次“清屏”。我使用 50 行是因为我有一个较大的命令行窗口，不过即便打印的空行多于窗口所支持的总行数也不会有任何影响。上面程序的运行结果应该如下所示：

```
System Status

Analog Input 0  :     ACTIVE 0.000000
Analog Input 1  :        OFF 0.000000
Analog Input 2  :     ACTIVE 0.000000
Analog Input 3  :        OFF 0.000000

Discrete Input 0: OFF 0
Discrete Input 1: OFF 1
Discrete Input 2: OFF 0
Discrete Input 3: OFF 0
Discrete Input 4: OFF 0
Discrete Input 5: OFF 0
Discrete Input 6: OFF 0
Discrete Input 7: OFF 0
```

因为原始的控制台 I/O 没有一个通用的清屏功能，打印一连串空字符串的 print 语句能够被用来使当前所有显示的数据滚动出屏幕，以为新数据腾出空间。注意，如果窗口比模板中的行数大，模板中的文本会在窗口下端结束（最后一行打印在提示符上方）。

不过，还有另一种方法。我们可以找出代码运行在什么操作系统上，然后执行适当的清屏命令，如下：

```
def clrDisp(numlines=50):
    if os.name == "posix":
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        os.system('CLS')
    else:
        print '\n' * numlines
```

下面的例子即采用此技术：

```
#! /usr/bin/python
# console2.py
#
# 使用内建的 print 方法和与特定操作系统相关的清屏技术演示控制台输出。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import time
import os

ainstat   = ['OFF','OFF','OFF','OFF']
aindata   = [0.0, 0.0, 0.0, 0.0]
discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
discin    = [0,0,0,0,0,0,0,0]

def clrDisp(numlines=50):
    if os.name == "posix":
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        os.system('CLS')
    else:
        print '\n' * numlines

def drawData():
    print ""
    print "System Status"
    print ""
    print "Analog Input 0  : %10s  %f" % (ainstat[0], aindata[0])
    print "Analog Input 1  : %10s  %f" % (ainstat[1], aindata[1])
    print "Analog Input 2  : %10s  %f" % (ainstat[2], aindata[2])
    print "Analog Input 3  : %10s  %f" % (ainstat[3], aindata[3])
    print ""
    print "Discrete Input 0: %3s  %d" % (discstate[0], discin[0])
    print "Discrete Input 1: %3s  %d" % (discstate[1], discin[1])
    print "Discrete Input 2: %3s  %d" % (discstate[2], discin[2])
    print "Discrete Input 3: %3s  %d" % (discstate[3], discin[3])
    print "Discrete Input 4: %3s  %d" % (discstate[4], discin[4])
    print "Discrete Input 5: %3s  %d" % (discstate[5], discin[5])
    print "Discrete Input 6: %3s  %d" % (discstate[6], discin[6])
    print "Discrete Input 7: %3s  %d" % (discstate[7], discin[7])
    print ""

ainstat[1] = "ACTIVE"
ainstat[3] = "ACTIVE"
aindata[1] = 5.2
aindata[3] = 8.9
```

```
discstate[0] = "ON"
discin[0] = 1
clrDisp()
drawData()
time.sleep(1.0)

ainstat[1] = "OFF"
ainstat[3] = "OFF"
aindata[1] = 0.0
aindata[3] = 0.0
ainstat[0] = "ACTIVE"
ainstat[2] = "ACTIVE"
aindata[0] = 5.2
aindata[2] = 8.9
discstate[0] = "OFF"
discin[0] = 0
discstate[1] = "ON"
discin[1] = 1
clrDisp()
drawData()
time.sleep(1.0)
```

491

现在我们已经搭建了一个基本可用的文本显示界面。要更新显示的数据，首先用适当的命令清屏，然后使用 print 方法输出模板中的行。如果无法确定操作系统，这个方法比使用空行清屏的方法具有更快的屏幕刷新速度。图13-1 表示了我们迄今所见到的文本控制台数据显示的流程。

读取用户输入

现在我们已经能够显示数据了，但还缺少一件重要的事情：用户输入。幸运的是，Python 已经提供了一个放诸四海皆准的输入方法来从控制台读取输入。raw_input() 方法每次从控制台读取一行数据，并将其作为一个字符串返回。这里所说的一行，是指以 EOL 结尾的零个或者多个字符。EOL 不会被追加在 raw_input() 返回的字符串结尾。

raw_input() 方法也能显示一个字符串提示符，如果安装了 readline 模块，它还能使用该模块来支持行编辑并记录输入历史。下面是控制台示例的第三个版本，这次在代码中加入了 raw_input()：

```
#! /usr/bin/python
# console3.py
#
# 使用内建的 print 方法和 raw_input() 方法演示控制台输出。
#
```

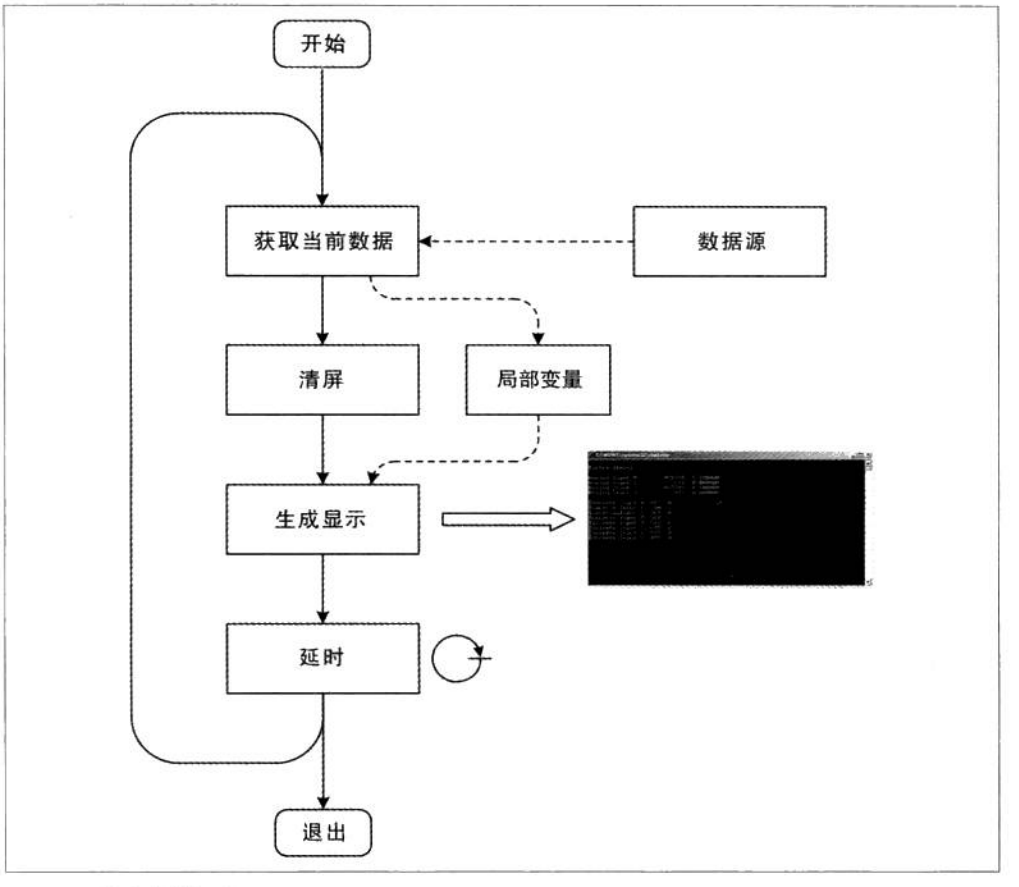


图13-1：文本式数据显示

```
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import time
import os

ainstat   = ['OFF','OFF','OFF','OFF']
aindata   = [0.0, 0.0, 0.0, 0.0]
discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
discin    = [0,0,0,0,0,0,0,0]
x = 0

def initData():
    global ainstat, aindata, discstate, discin

    ainstat   = ['OFF','OFF','OFF','OFF']
```

492

```
    aindata   = [0.0, 0.0, 0.0, 0.0]
    discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
    discin    = [0,0,0,0,0,0,0,0]

def readData():
    global ainstat, aindata, discstate, discin, x

    initData()
    discstate[x] = 'ON'
    discin[x] = 1
    x += 1
    if x > (len(discin) - 1):
        x = 0

def clrDisp(numlines=50):
    if os.name == "posix":
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        os.system('CLS')
    else:
        print '\n' * numlines

def drawData():
    print ""
    print "System Status"
    print ""
    print "Analog Input 0  : %10s  %f" % (ainstat[0], aindata[0])
    print "Analog Input 1  : %10s  %f" % (ainstat[1], aindata[1])
    print "Analog Input 2  : %10s  %f" % (ainstat[2], aindata[2])
    print "Analog Input 3  : %10s  %f" % (ainstat[3], aindata[3])
    print ""
    print "Discrete Input 0: %3s  %d" % (discstate[0], discin[0])
    print "Discrete Input 1: %3s  %d" % (discstate[1], discin[1])
    print "Discrete Input 2: %3s  %d" % (discstate[2], discin[2])
    print "Discrete Input 3: %3s  %d" % (discstate[3], discin[3])
    print "Discrete Input 4: %3s  %d" % (discstate[4], discin[4])
    print "Discrete Input 5: %3s  %d" % (discstate[5], discin[5])
    print "Discrete Input 6: %3s  %d" % (discstate[6], discin[6])
    print "Discrete Input 7: %3s  %d" % (discstate[7], discin[7])
    print ""

while True:
    readData()
    clrDisp()
    drawData()
    instr = raw_input("Command: ")
    if instr.upper() == 'X':
        break
```

每一次按Enter键，console3.py将按每次一行的方式依次改变离散输入行。如果输入x(或

X）字符，它就会终止。

493 然而，这个方法有一个缺点。raw_input()方法仅在阻塞模式下工作。换言之，它会一直阻塞，直至用户输入某些数据。这就意味着，只要raw_input()在一个主显示循环中，就不能用一个定时器来自动刷新屏幕。只要用户按X键之外的任意键并回车，本程序就不会读取新数据，也不会对显示进行更新。

不过，还是有一个解决方案，那就是把它置于一个线程中。我们可以创建一个线程从用户那里获取数据，用一个标志来通知主循环它什么时候该结束。代码如console4.py所示：

```
#! /usr/bin/python
# console4.py
#
# 使用内建的 print 方法和独立线程中的 raw_input() 方法演示控制台输出。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import time
import os
import threading

ainstat   = ['OFF','OFF','OFF','OFF']
aindata   = [0.0, 0.0, 0.0, 0.0]
discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
discin    = [0,0,0,0,0,0,0,0]

running = True
x = 0

def initData():
    global ainstat, aindata, discstate, discin

    ainstat   = ['OFF','OFF','OFF','OFF']
    aindata   = [0.0, 0.0, 0.0, 0.0]
    discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
    discin    = [0,0,0,0,0,0,0,0]

def readData():
    global ainstat, aindata, discstate, discin, x

    initData()
    discstate[x] = 'ON'
    discin[x] = 1
    x += 1
    if x > (len(discin) - 1):
        x = 0
```

```
def clrDisp(numlines=50):
    if os.name == "posix":
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        os.system('CLS')
    else:
        print '\n' * numlines

def drawData():
    print ""
    print "System Status"
    print ""
    print "Analog Input 0  : %10s  %f" % (ainstat[0], aindata[0])
    print "Analog Input 1  : %10s  %f" % (ainstat[1], aindata[1])
    print "Analog Input 2  : %10s  %f" % (ainstat[2], aindata[2])
    print "Analog Input 3  : %10s  %f" % (ainstat[3], aindata[3])
    print ""
    print "Discrete Input 0: %3s  %d" % (discstate[0], discin[0])
    print "Discrete Input 1: %3s  %d" % (discstate[1], discin[1])
    print "Discrete Input 2: %3s  %d" % (discstate[2], discin[2])
    print "Discrete Input 3: %3s  %d" % (discstate[3], discin[3])
    print "Discrete Input 4: %3s  %d" % (discstate[4], discin[4])
    print "Discrete Input 5: %3s  %d" % (discstate[5], discin[5])
    print "Discrete Input 6: %3s  %d" % (discstate[6], discin[6])
    print "Discrete Input 7: %3s  %d" % (discstate[7], discin[7])
    print ""
    print "Enter X to terminate"

def getCommand():
    global running

    while True:
        instr = raw_input()
        if instr.upper() == 'X':
            running = False
            break

#---------------------------------------------------------------------
# 启动 raw_input() 处理程序线程
getinput = threading.Thread(target=getCommand)
getinput.start()

while running:
    readData()
    clrDisp()
    drawData()
    time.sleep(1.0)
```

495

现在我们已经有了一个能用来创建控制台数据显示的通用框架，它能够接收基本命令，而且只用到了 print 和 raw_input() 内置方法。

这时，或许你会认为基于控制台的字符显示不是特别有用，图 13-2 所示的是一个屏幕快照，显示了一组用来测试一些空间任务仪器的字符界面窗口。

图13-2：基于文本的测试系统数据显示

这 6 个窗口每个都仅使用了 ASCII 字符，没有 ANSI 定位或者任何其他更炫的效果（我们很快就会提到 ANSI 显示控制）。屏幕显示命令行接口给航天飞机的处理器，响应和信道处理函式状态通过无线信道发回。还有一个窗口用来输入命令，尽管这个系统由其他工具（没显示出来）驱动，这些工具会发送飞行器软件所用格式的命令行串。

与操作系统相关的控制台 I/O

使用我们所看到的这种基于字符显示的好处之一（可能也是最大的好处）就是它们大多都是平台无关的。print 和 raw_input() 方法不依赖具体平台，只有清屏方法需要用到一些底层操作系统的知识。然而，使用 print 和 raw_input() 还是有一些限制的，当你需要提供给用户显示和 I/O 之外的更多控制的时候，这些情况就更为突出。

Linux 和 Windows 都提供了一组基本函式来处理底层控制台 I/O。在 Linux 下，控制台 I/O 功能函式如 getch() 和 ungetch() 在 curses 库中，它们不是标准 I/O 函式库的一部分。Windows 将类似函式作为 msvcrt DLL 模块的一部分提供给用户。Python 通过包含库模块 msvcrt 来访问 Microsoft msvcrt DLL 中的函式，在 UNIX/Linux 环境下则包含 curses 库模块。在下一节，我们将看到如何使用这些和其他底层控制台 I/O 函式，并使用 ANSI 终端控制序列和 curses 屏幕显示软件包，来创建结构化的的文本界面。不过，在深入研究 ANSI 和 curses 之前，让我们先来看看这些基本函式是如何被用在字符界面显示上的。

496

Linux 和 Windows 中文本界面的差异

现在，我们已经走到 Linux 和 Windows 开始分道扬镳的岔路口了。一旦离开简单的 print 语句和 raw_input() 的领域，我们就会开始遇到这两个操作系统之间的一些显著差异，这是由它们的底层设计哲学造成的。

使用 ANSI 来控制终端显示的功能很早就已成为 UNIX 和后来的 Linux 的一部分。UNIX 在小型计算机和主机上发展成一个带有终端用户界面的多任务分时系统，而 Linux 也继承了这种方式。而 Windows 则是从 DOS 发展而来的，DOS 上支持多任务，而且是单用户单屏幕。直到 Microsoft 发布了 ANSI.sys 驱动模块，PC 上才能解释 ANSI 序列来控制光标定位，并直接操作基于文本的界面。现在的 Windows 几乎都不支持 ANSI 屏幕控制。因为它更注重 GUI 用户交互，Windows 似乎对这种旧的做事方式不再有什么兴趣。

497

然而，缺少多任务和多线程（至少在最开始的 DOS 上如此）迫使了一些新方法的实现，最后，一个有用的函式 kbhit() 出现在 DOS 和 Windows 环境中。这个方法很有用，事实上它已经是 Microsoft Visual C 运行库（msvcrt DLL 模块）标准的一部分，而且很久以前就如此了。在 Linux 上没有直接的替代品。

kbhit() 是一个非阻塞函式，如果一个字符在控制台输入缓冲中等待，它就会返回 True。这样就允许一个单任务、没有线程的运行环境中运行的程序，在主循环中不阻塞地检测输入（回顾一下之前的 console3.py 中的阻塞行为）。在示例 console4.py 中，用了一个线程用来实现类似 kbhit() 的功能。当然，在 Linux 下有其他的办法实现这样的功能，那就是调用 select() 函式，不过我不想在这里引入它。但是我鼓励你自己去研究它，那是非常有趣的探索。

从用户那里获取单个字符时，Windows 下的 getche() 函式和 Python 的 raw_input() 方法在本质上是一样的。要读取一个字符串，一个方法是将 getche() 放到一个循环中去执行，直到读取到 EOF。这个方法能工作，但不是十分优雅。与之相比，Python 的 raw_input() 方法要强大得多，而且易于使用。

使用 Python 的 msvcrt 库模块

对于 Windows 环境而言，Python 包含一个便利的模块，支持程序访问 Microsoft 运行时库 msvcrt 所提供的一些底层函式。注意，这并不适用于 Linux 平台。下一节我们讲 curses 和 ANSI 控制序列的时候会看到在 Linux 下如何完成这些工作。表 13-1 罗列了 Python 的 msvcrt 库模块可用的函式。

表13-1：Python的msvcrt模块函式

msvcrt 函式	描述
kbhit()	如果按键事件发生且输入缓冲区中有等待读取的字符，则返回真
getch()	调用时阻塞，直到按键事件发生并返回输入缓冲中的字符。不在终端回显输入的字符，并且不会等待 EOL 才返回
getwch()	“宽字符”版本的 getch()；返回一个 Unicode 值
getche()	等价于 getch()，但会在终端回显可打印的字符
getwche()	“宽字符”版本的 getche()；返回一个 Unicode 值
putch(*char*)	不经过缓冲直接打印一个字符到终端
putwch(*unicode_char*)	“宽字符”版本的 putch()，接收一个 Unicode 字符值
ungetch(*char*)	实现了一个将字符压入控制台缓冲的“push-back”，让被压入的字符成为 getch() 或者 getche() 下一次要读取的字符
ungetch(*unicode_char*)	宽字符版本的 ungetch()，接收一个 Unicode 字符值后入栈

下面是示例程序 console5.py 的程序清单，从中可以看到如何用 kbhit() 代替在 console4.py 中所使用的线程和 raw_input()。

```
#! /usr/bin/python-
# console5.py
#
# 使用内建的 print 方法演示控制台输出，并演示如何使用 msvcrt 的 kbhit() 函式。
#
# 只能在 Windows 系统中使用，不能用于 Linux 操作系统。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import time
import os
import msvcrt

ainstat   = ['OFF','OFF','OFF','OFF']
aindata   = [0.0, 0.0, 0.0, 0.0]
discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
discin    = [0,0,0,0,0,0,0,0]

running = True
```

```
x = 0

def initData():
    global ainstat, aindata, discstate, discin

    ainstat   = ['OFF','OFF','OFF','OFF']
    aindata   = [0.0, 0.0, 0.0, 0.0]
    discstate = ['OFF','OFF','OFF','OFF','OFF','OFF','OFF','OFF']
    discin    = [0,0,0,0,0,0,0,0]

def readData():
    global ainstat, aindata, discstate, discin, x

    initData()
    discstate[x] = 'ON'
    discin[x] = 1
    x += 1
    if x > (len(discin) - 1):
        x = 0

def clrDisp(numlines=50):
    if os.name == "posix":
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        os.system('CLS')
    else:
        print '\n' * numlines

def drawData():
    print ""
    print "System Status"
    print ""
    print "Analog Input 0  : %10s  %f" % (ainstat[0], aindata[0])
    print "Analog Input 1  : %10s  %f" % (ainstat[1], aindata[1])
    print "Analog Input 2  : %10s  %f" % (ainstat[2], aindata[2])
    print "Analog Input 3  : %10s  %f" % (ainstat[3], aindata[3])
    print ""
    print "Discrete Input 0: %3s  %d" % (discstate[0], discin[0])
    print "Discrete Input 1: %3s  %d" % (discstate[1], discin[1])
    print "Discrete Input 2: %3s  %d" % (discstate[2], discin[2])
    print "Discrete Input 3: %3s  %d" % (discstate[3], discin[3])
    print "Discrete Input 4: %3s  %d" % (discstate[4], discin[4])
    print "Discrete Input 5: %3s  %d" % (discstate[5], discin[5])
    print "Discrete Input 6: %3s  %d" % (discstate[6], discin[6])
    print "Discrete Input 7: %3s  %d" % (discstate[7], discin[7])
    print ""
    print "Enter X to terminate"

def getCommand():
    global running
```

499

```
    if msvcrt.kbhit():
        inchar = msvcrt.getch()
        if inchar.upper() == 'X':
            running = False

#---------------------------------------------------------------------

while running:
    readData()
    clrDisp()
    drawData()
    getCommand()
    if running == True:
        time.sleep(1.0)
```

console4.py 和 console5.py 的一个主要不同之处是 getCommand() 函式的实现。在 console5.py 中不再使用一个线程，此外，raw_input() 也被替换成了 kbhit()。现在只需按一下 X 键就能退出，而在 console4.py 中则要按完 X 键后再按 Enter 键。不过，

console4.py 和 console5.py 最大的区别是，console5.py 不能在 Linux 上运行，尽管两者行为极其相似。使用 msvcrt 模块让 console5.py 变得和特定操作系统相关，它只能在 Windows 环境中运行。

ANSI 显示控制台技术

20 世纪 70 年代中期，视频显示终端（VDT）作为当时流行的打孔机和电传打字机的接任者出现。VDT 之所以成为继任者，在一定程度上归因于它和计算机系统的接口方式，还有它提供了更加灵活的动态显示。对于一个同时支持多用户的分时系统，一个简单的串行接口（如 RS-232）就能满足一个终端连接到主机的所有工作。之所以能这样，是因为屏幕管理和键盘输入的琐碎工作都交给了终端去完成，主机只需发送数据和命令到终端，并获得用户的输入即可。从某种意义上讲，这就是早期的分布式应用。

事情发展得很快，不久就出现了可视化的，并能在屏幕上任何位置定位的光标技术，从而可以将数据置于显示模板，或者表框中。可定位光标的技术对于实现全屏文本编辑器而言是必需的，如 *vi*（UNIX）和 *edt*（用于 DEC 的 VMS OS）。同时，人们还发现屏幕控制对游戏也很有用，如果你玩过 Rogue、Nethack 或者 Empire，你就知道这些早期游戏是什么样子了。其中一些给人留下了深刻的印象，要知道它们仅使用了 ASCII 字符集。

最初，终端制造商习惯性地想设计他们自己特有的屏幕控制方案。这很快就产生了一个问题，因为如果你想为一个系统添加更多终端，新的终端可能跟已经存在的软件不兼容。在大多数方案中，通过发送一个不可打印的字符（或者字符序列）来控制终端显示和处理过程。只要 X 厂商的终端和 Y 厂商的终端能识别相同字符序列，则一切都安然无事。

但是情况并非总是如此。UNIX 系统通过“termcap”——终端兼容性翻译方案来解决这一问题，这种机制能够从给定的函式中为终端选择合适的控制序列。在 Linux 系统上，你还能在 */etc* 目录下找到 *termcap* 的数据文件库文件，尽管它已经被 *terminfo* 所取代。

1976 年，ANSI 转义序列有了一个最初的标准，即 ECMA-48。它后来发展成为 ISO/IEC 6429。这个 ANSI 标准在 1981 年左右被采纳，成为正式的 ANSI X3.64，而且这仍然是引述标准化终端控制序列的最常用方式。在后面的篇幅中，当提及“ANSI”时，你就可以认为我指的是 ECMA-48/ANSI X3.64。

虽然 VDT 已经过时，但 ANSI 控制序列依旧非常有用，而且今后可能只要字符型显示存在它就依然会有用武之地。例如，Linux（和 UNIX）上的 Xterm 工具就支持 DEC VT100 VDT 系列控制序列，而且它无疑是 ANSI 标准最普遍的方式。Xterm 还支持不同颜色，甚至有一个 Tektronix 4014 仿真模式可以用来模拟向量图形。更多有关 Xterm 的 ANSI 功能的内容可参见本章最后“推荐阅读”一节中的 URL。 501

ANSI 和 Windows

Windows 本身并不直接支持 ANSI 控制台屏幕控制，而且我不推荐使用 16 位的 *ANSI.sys* 驱动。Jason Hood 的开源软件包 *ansicon* 是 *ANSI.sys* 的替代方案，它包含了对 32 位和 64 位 Windows 的 ANSI 驱动。安装了 *ANSI32.dll* 或者 *ANSI64.dll* 以后，*cmd.exe* 命令窗口就能处理一部分有用的 ANSI 控制序列，但它还不能处理所有 ANSI 序列。你需要明确决定自己需要发送什么样的序列。该软件包的源代码可以从这个网址下载：*http://adoxa.110mb.com/ansicon/index.html*。

Windows 上有多种能够识别 ANSI 序列的控制台 shell 替代品和终端模拟器。其中，有些好，有些差；有些是免费的，还有一些需要付费。我们已经在第 7 章和第 10 章见到过 Tera Term 终端模拟器与 com0com 虚拟串口工具的一些应用。这里还有一个有点儿技巧的小程序。使用 com0com 来写一个 Python 应用，它的 I/O 通过串口连接到一个 com0com 的虚拟端口，然后使用 Tera Term 的 VT100 终端模拟器连接到另外一个端口，所有这些事情都在同一台机器上完成。

基本 ANSI 控制序列

现在我们来看看 ANSI 控制序列。表 13-2 是 ANSI 控制序列集合中的一部分，包含了我认为实现我们的目的最为需要的序列，也就是光标定位和一些显示管理序列。我舍弃了一些图像显示属性控制（如高亮显示、低亮显示、颜色等）之类的功能，因为在这里它们只会使简单的主题复杂化。然而，如果你确实需要在橙色的区域上显示明亮的蓝色的闪烁字符，则尽管自由地使用这些功能。

502 表13-2：基本的ANSI控制序列

序列	功能	描述
<ESC>[*row;col* H	光标移动或返	设置光标位置到下一个文本的开始处。若未指定参数行 / 列
<ESC>[H	回起始位置	（如，<ESC>[H），则将光标移动到屏幕左上角的起始位置
<ESC>[*count* A	光标向上移动	光标上移 *count* 行；*count* 的默认值为 1
<ESC>[A		
<ESC>[*count* B	光标向下移动	光标下移 *count* 行；*count* 的默认值为 1
<ESC>[B		
<ESC>[*count* C	光标向前移动	光标向前移 *count* 列；*count* 的默认值为 1
<ESC>[C		
<ESC>[*count* D	光标向后移动	光标向后移 *count* 列；*count* 的默认值为 1
<ESC>[D		
<ESC>E	下一行	光标移动至下一行开始处
<ESC>[s	保存光标位置	保存光标当前位置
<ESC>[u	恢复光标位置	恢复到一个存储过的光标位置
<ESC>D	垂直下移	光标移动到下一行的相同列上
<ESC>M	垂直上移	光标移动至上一行的相同列上
<ESC>[0K	删除至行尾	删除从当前位置到行尾的内容
<ESC>[2K	删除一行	删除当前一行内容
<ESC>[2J	清屏	用背景颜色清屏，并将光标移动至起始位置

注意，<ESC>表示ASCII码的“escape”字符，在Python字符串中可以写作\x1b。斜体字变量名表示可更改的十进制参数，例如，row可以被替换成一个行号。对某些序列，表中显示了两种命令形式。第二个形式是默认形式，在该形式中，如果默认行为可以满足要求，可以省略数值参数。最后，一些序列的开始处只有一个<ESC>字符，而另一些则用<ESC>[这两个字符（即escape字符\x1b后面跟一个左方括符[）。

下面这个例子，bgansi.py是一个条状图显示的例子，使用了表13-2中的控制序列：

```
#!/usr/bin/python
# bgansi.py
#
# 演示基本的ANSI屏幕控制，还演示了如何使用sys库模块进行stdout(标准输出)。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import random
import time
from sys import stdout

MAXEXT  = 30
```

```
ROWSTRT = 7      # 第 1 行使用屏幕的第 8 行
COLSTRT = 9      # 列开始偏移
VALPOS  = 45     # 用于显示列的值

ran = random.random
output  = stdout.write
outflush = stdout.flush

def generateBars():
    # 清屏
    output("\x1b[2J")

    # 把 8 个 ID 字符串和标记放置在屏幕最左边
    # 列起始位置在从顶部开始的第 8 行（行 7）
    i = 1
    for row in range(ROWSTRT,15):
        # 设置光标位置
        output("\x1b[%d;%dH" % (row, 0))
        # 写标记字符
        output("Chan %d |" % i)
        i += 1


# 边条按从 0 到 7 编号
def updateBar(barnum, extent):
    # 调整到实际位置
    row = barnum + ROWSTRT

    # 限定长度以防止触碰到显示的右边沿
    if extent > MAXEXT:
        extent = MAXEXT
    # 确保总是有打印
    if extent < 1:
        extent = 1

    # 先清除行（使图形收缩）
    output("\x1b[%d;%dH" % (row, COLSTRT))
    # 删至行尾
    output("\x1b[0K")

    # 遍历所有位置，直至 extent
    for col in range(0, extent):
        # 设置位置
        output("\x1b[%d;%dH" % (row, COLSTRT+col))
        # 使用等号填充边条
        output("=")
        # 写入实际使用的值
        output("\x1b[%d;%dH" % (row, VALPOS))
        output("%d" % extent)
        outflush()
```

503

```
def runTest():
    generateBars()

    for x in range(0, 100):
        for barnum in range(0, 8):
            # 随机数函式返回 0 ~ 1 之间的浮点数，用来缩放 MAXEXT
            val = int(ran() * MAXEXT)
            updateBar(barnum, val)
504 >       # 停顿一下
            time.sleep(0.1)

    output("\x1b[%d;%dH" % (20, 0))
    outflush()
    print ""

if __name__ == '__main__':
    runTest()
```

bgansi.py 的输出如图 13-3 所示。这只是一个屏幕快照，在运行时它是活动的。

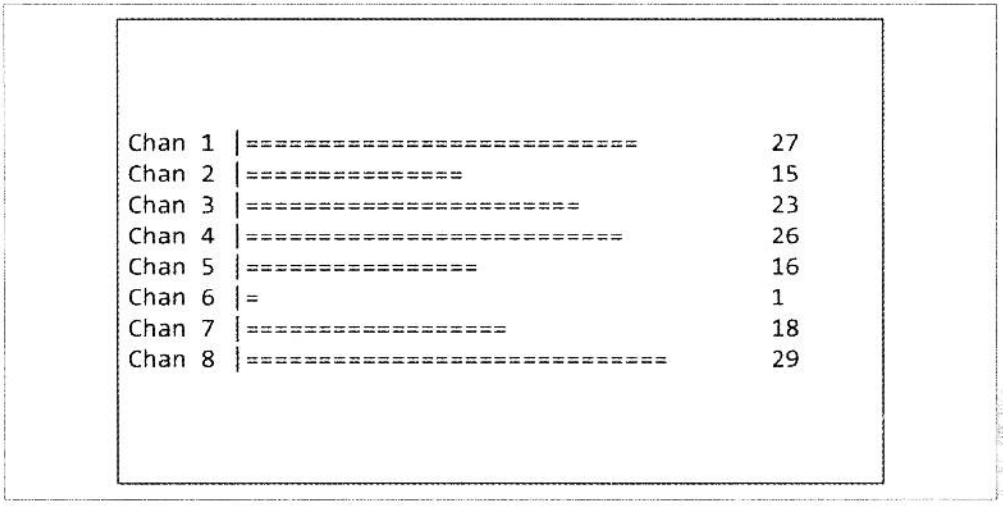


图13-3：ANSI 条状图输出示例

bgansi.py 中还有几点需要关注一下。对初学者而言，要注意我并没有使用 Python 内建的 print 函式。在这个例子中，我们只是想直接输出到 *stdout*，无须进行任何修改，所以这个例子直接调用了 stdout 对象的 write() 方法。而且，因为 *stdout* 是内置缓冲的，flush() 方法用来保证所有的输出在想要它们输出时都真正地输出。

在 bgansi.py 中一个值得注意的地方是，代码里有大量的 \x1b[序列，而且使用它的 ANSI 控制字符串不是很直观。在代码中以这种风格来使用 ANSI 控制代码完全就是自找苦吃。一个隐藏细节的办法是使用伪宏（pseudo-macro）定义别名，如下：

```
CSI = "\x1b["
CLR = "2J"
```

但这还不是最好的解决方案。更好的方案是使用一个方法库，而不仅仅是把细节隐藏起来。在下一节我们将看到这样一个库，利用它，不仅不用查看序列细节，而且还能重定向 I/O。在这之后，我们将研究 Python 针对 curses 库的接口。 505

SimpleANSI 库

不是每个应用都需要用到全部 ANSI 控制序列集。有时候只要能切换光标并在指定位置接收用户输入就足够了。但事实上，这种序列确实趋于在做一种笨拙的工作而且很难阅读，因此，一个针对某些类型的、简洁而优雅的 API 将对我们有很大帮助。

现在假设我们有一个屏幕布局如图 13-4 所示。

```
Input Data Monitor
Current Date: 101002
Current Time: 18:27:08

Chan    Last Value
  1        6.49
  2        7.70
  3        2.37
  4        7.89
  5        8.73
  6        6.23
  7        2.60
  8        6.22

Enter X to exit.
```

图13-4：ANSI数据显示

SimpleANSI 是一个通过 ANSITerm 类提供了基本屏幕控制功能的库模块，现在我们来了解一下它如何被用在图 13-4 中。ANSITerm 类支持表 13-2 中所列的控制序列，而且包括两个修改，允许选择性地控制“删除至行尾”和“删除一行”序列的初始光标位置（分

别是 ANSITerm 的 clrEOL() 和 clrLine() 方法)。

ANSITerm类为ANSI屏幕控制提供了17个公有方法,见表13-3中。此外,它还处理套接字、串口和控制台I/O模式。

506 表13-3: ANSITerm类的方法

方法	描述
clrEOL(*row,col*)	清除行内从 *row*、*col* 到 EOL 部分
clrLine(*row*)	清除第 *row* 行整行
clrScreen()	清空屏幕
indexDown()	向下移动 / 滚动一行
indexUp()	向上移动 / 滚动一行
moveBack(*count*)	光标左移 *count* 列
moveDown(*count*)	光标下移 *count* 行
moveForward(*count*)	光标右移 *count* 列
moveHome()	光标移至左上角
moveNextline()	移动光标到下一行行首
movePos(*row,col*)	移动光标到屏幕上的 (*row,col*) 位置
moveUp(*count*)	光标向上移动 *count* 行
readInput(*reset=False*)	获得用户输入并回显。如果 *reset=True*，光标会在输入结束后返回到初始位置
resetDev()	重置终端到初始状态
restorePos()	还原光标到最后存储的位置上
savePos()	保存光标当前位置
writeOutput(*outstr*)	将指定的字符串写入当前光标所在位置

下面列出的即是 SimpleANSI 模块的源代码。注意，I/O 按所选定的类型被映射到一对对象上。默认情况是使用控制台，但是类的初始化也接受可用的套接字或者串口 I/O 对象。

```
#! /usr/bin/python
# SimpleANSI.py
#
# 简单地针对 ANSI 屏幕控制的函式集
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

""" 用于 Python 的简单 VT100/xterm ANSI 终端功能函式。

    此模块是基于 C 代码写成的，而 C 代码最初是为嵌入式控制器上的 VxWorks 编写的。
    它用于控制 ANSI 兼容型终端或终端模拟器的显示。它能和 UNIX/Linux 系统上
    的 Xterm，Windows 下的 CygWin 以及 Tera Term 一起工作。
```

```
    ANSITerm 类支持通过 ANSI 的控制台，或是串行连接以及网络套接字进行 I/O。

    ANSITerm 从来不打算替代 curses，
    它仅仅是一个把格式化的数据快捷显示出来的方法，仅此而已。
    在诊断、状态显示，以及简单的命令接口方面它很有用。

    伪宏 CSI 是 ANSI 的“命令序列引导码”。

    注：此代码没有在所有可能的环境中，为所有可能的或可行的用例进行测试。
    它可能包含错误、遗漏，或者其他一些令人不悦之处。
"""

from sys import stdout
import time

ESC = "\x1b"
CSI = ESC+"["

CON = 0
SKT = 1
SIO = 2

class ANSITerm:
    """ 简单 ANSI 终端控制。

        支持通过 ANSI 的控制台，或是串行连接以及网络套接字进行 I/O。

        经由网络套接字通信时，它被假定物理端口是一个 socket 的发送和接收方法，
        并已经在别处打开。

        如果使用串行端口，该端口必须已经打开。
        在这种情况下，ioport 必须引用一个有效的 pySerial 的对象。

        默认的 I/O 模式是使用控制台，它必须支持 ANSI 控制序列，
        否则将只是打印 ANSI 序列，不会解释执行。
        另外请注意，所有的用户通过控制台输入时，按下回车键即表示输入完成。
        这是由 Python 的 raw_input() 函式支持的，所以没有原生 getch() 样的函式。

          屏幕坐标按 (row, col) 方式指定，换句话说，就是 (y, x)，这与 curses 的
          处理方式相同。
    """
    def __init__(self, ioport=None, porttype=CON):
        """ 初始化 ANSIterm 对象。

            如果 porttype 是 CON 以外的，ioport 必须引用一个有效的 I/O 端口对象。

            如果 porttype 是 CON，则 self.port 会分配为 None。

            默认的 I/O 方式是控制台。
        """
```

```
        self.pktsize = 1024      # SKT 模式的默认值
        self.port    = ioport    # SKT 及 SIO 端口对象
        self.portOK  = False     # 端口是否是有效的标识

        # 映射到适当的 I/O 处理程序
        if porttype == SKT:
            if self.port:
                self.portOK   = True
                self.outfunc  = self.__sktOutput
                self.inpfunc  = self.__sktInput
        elif porttype == SIO:
            if self.port:
                self.portOK   = True
                self.outfunc  = self.__sioOutput
                self.inpfunc  = self.__sioInput
        else:
            self.port     = None
            self.portOK   = True
            self.outfunc  = self.__conOutput
            self.outflush  = self.__conFlush
            self.inpfunc  = self.__conInput

    #-----------------------------------------------------------------
    # I/O 处理程序
    #-----------------------------------------------------------------
    # 虽然可以使用单行函式形式来完成定义，
    # 但是目前的形式方便未来随时加入新的错误处理。
    # 例如，能捕捉到错误并进行日志的数据 I/O，
    #
    # 需要注意的是，套接字和串行 I/O 的方法假设，
    # 我们使用的是标准 Python 的 socket 对象或 pySerial 端口对象。
    # 其他类型的 I/O 对象可能需要不同的读 / 写方法。
    #-----------------------------------------------------------------
    def __sktOutput(self, outstr):
        self.port.send(outstr)

    def __sioOutput(self, outstr):
        self.port.write(outstr)

    def __conOutput(self, outstr):
        stdout.write(outstr)

    def __conFlush(self):
        stdout.flush()

    def __sktInput(self):
        return self.port.recv(self.pktsize)

    def __sioInput(self):
```

```
        return self.port.readline()

    def __conInput(self):
        return raw_input()      # no prompt is specified for raw_input

    #-------------------------------------------------------------------
    # 光标定位
    #-------------------------------------------------------------------
    def moveHome(self):
        """ 移动光标到左上角
        """
        if self.portOK:
            self.outfunc("%sH" % CSI)

    def moveNextline(self):
        """ 移动光标到下一行开始
        """
        if self.portOK:
            self.outfunc("%sE" % ESC)

    def movePos(self, row, col):
        """ 将光标移动到屏幕的 row, rol 位置
        """
        if self.portOK:
            self.outfunc("%s%d;%dH" % (CSI, row, col))

    def moveUp(self, count):
        """ 光标向上移动 count 行
        """
        if self.portOK:
            self.outfunc("%s%dA" % (CSI, count))

    def moveDown(self, count):
        """ 光标向下移动 count 行
        """
        if self.portOK:
            self.outfunc("%s%dB" % (CSI, count))

    def moveFoward(self, count):
        """ 光标向右移动 count 列
        """
        if self.portOK:
            self.outfunc("%s%dC" % (CSI, count))

    def moveBack(self, count):
        """ 光标向左移动 count 列
        """
        if self.portOK:
            self.outfunc("%s%dD" % (CSI, count))
```

509

```
    def indexUp(self):
        """ 向上移动 / 滚动一行
        """
        if self.portOK:
            self.outfunc("%D" % ESC)

510 >   def indexDown(self):
        """ 向下移动 / 滚动一行
        """
        if self.portOK:
            self.outfunc("%M" % ESC)

    def savePos(self):
        """ 保存光标当前位置
        """
        if self.portOK:
            self.outfunc("%ss" % CSI);

    def restorePos(self):
        """ 还原光标到最后存储的位置
        """
        if self.portOK:
            self.outfunc("%su" % CSI);

    #-----------------------------------------------------------------
    # 显示控制
    #-----------------------------------------------------------------
    def clrScreen(self):
        if self.portOK:
            self.outfunc("%s2J" % CSI);

    def clrEOL(self, row=None, col=None):
        """ 从指定或是当前位置清行。

            如果 row 及 col 为空，则使用当前位置。
        """
        if self.portOK:
            if row and col:
                self.movePos(row,col);
            self.outfunc("%s0K" % CSI)

    def clrLine(self, row=None):
        """ 清除第 row 行整行，

            如果 row 为空，使用当前位置。
        """
        if self.portOK:
            if row:
                self.movePos(row, 1)
            self.outfunc("%s2K" % CSI)
```

```
    def resetDev(self):
        """ 重置终端到初始状态
        """
        if self.portOK:
            self.outfunc("%sc" % ESC)

    #-----------------------------------------------------------------
    # 输入
    #-----------------------------------------------------------------
    def readInput(self, reset=False):
        """ 获取用户输入并回显。

            如果需要一个提示，它必须在方法调用前先生成到合适的位置。

            如果 reset 为 True，那么当输入处理程序返回时，
            光标将被重新定位到用户输入的起始位置之前。
            提供这种能力主要是为了补偿使用 Python 的 raw_input() 交互时，
            用户在控制台需要按 Enter 键完成输入。
        """
        instr = ""
        if self.portOK:
            if reset: self.savePos()
            instr = self.inpfunc()
            if reset: self.restorePos()
            return instr

    #-----------------------------------------------------------------
    # 输出
    #-----------------------------------------------------------------
    def writeOutput(self, outstr):
        """ 写任意字符串到当前光标位置
        """
        if self.portOK:
            self.outfunc(outstr)
            self.outflush()


# 自测
if __name__ == "__main__":
    term = ANSITerm(None, 0)

    term.clrScreen()

    # 将显示的行按 1~24 编号
    for i in range(1,21):
        term.movePos(i,0)
        term.writeOutput("%0d" % i)

    # 写一些内容到屏幕
```

511

```
term.movePos(14,4)
term.writeOutput("* This is line 14, column 4")
time.sleep(1)
term.movePos(15,4)
term.writeOutput("* This is line 15, column 4")
time.sleep(1)

# 使用一系列字符形成对角线
for i in range(2,12):
    term.movePos(i,i+4)
    term.writeOutput("X")
    time.sleep(0.1)

for i in range(2,12):
    term.movePos(i,i+4)
    term.writeOutput(" ")
    time.sleep(0.1)

for i in range(2,12):
    term.movePos(i,i+4)
    term.writeOutput("X")
    time.sleep(0.1)

# 用笨方法实现闪烁
for i in range(0,10):
    term.movePos(17,10)
    term.writeOutput("blick blink")
    time.sleep(0.5)
    term.clrEOL(17,10)
    time.sleep(0.5)

term.movePos(18,4)
term.writeOutput("Did it blink? (y/n): ")
instr = term.readInput()
term.movePos(19,4)
term.writeOutput("You answered %s, thank you for playing." % instr)

term.movePos(22,1)
# and that's it
```

512

使用 SimpleANSI

回顾图 13-4 所示的 ANSI 数据显示，我们首先会注意到它是固定格式的显示。创建此类显示的一个方法是使用预定义的模板来产生基本的屏幕显示，再使用新数据不断地重新绘制。示例 console4.py 就是这么做的。这虽然可以工作，但如果能够绘制一次静态部分显示，然后只需改变必要的数据区域，将会更加高效。示例 console6.py 展示了使用 ANSITerm 类来控制可变数据并接收用户输入。它只产生一次主显示画面，后面都只对主

显示区的特定位置进行修改。console6.py 代码如下：

```
#!/usr/bin/python
# console6.py
#
# 演示 SimpleANSI 模块的用法。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import random
import time
import datetime
import threading
import SimpleANSI

data_vals = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
currdate = ""
currtime = ""
updt_cnt1 = 0
updt_cnt2 = 0

ran = random.random

def getDate():
    global currdate

    t = datetime.datetime.now()
    currdatetime = t.timetuple()

    yr = str(currdatetime[0])
    currdate = "%02d"%int(yr[2:]) + "%02d"%currdatetime[1] +\
       "%02d"%currdatetime[2]

def getTime():
    global currtime

    t = datetime.datetime.now()
    currdatetime = t.timetuple()

    currtime = "%02d:"%currdatetime[3] + "%02d:"%currdatetime[4] +\
        "%02d"%currdatetime[5]

# 将数据和时间写入屏幕
def writeDateTime():
    getDate()
    getTime()

    term.clrEOL(2,15)
```

513

```
        term.movePos(2,15)
        term.writeOutput("%s" % currdate)
        term.clrEOL(3,15)
        term.movePos(3,15)
        term.writeOutput("%s" % currtime)

    # 获得仿真的数据输入值
    def getDataVals():
        global data_vals, updt_cnt1, updt_cnt2

        data_vals[0] = ran() * 10.0
        data_vals[1] = ran() * 10.0

        if updt_cnt1 >= 4:
            for i in range(2,5):
                data_vals[i] = ran() * 10.0
            updt_cnt1 = 0
        else:
            updt_cnt1 += 1

        if updt_cnt2 >= 10:
            for i in range(4,8):
                data_vals[i] = ran() * 10.0
            updt_cnt2 = 0
        else:
            updt_cnt2 += 1

    # 写通道数据值
    def writeDataVals():
        idx = 0
        for i in range(6,14):
            term.movePos(i,10)
            term.writeOutput("%6.2f" % data_vals[idx])
            idx += 1
        # 更新数据后将光标放到文字后
        term.movePos(16,1)

    # 生成主显示
    def mainScreen():
        term.clrScreen()

        term.movePos(1,1)
        term.writeOutput("Input Data Monitor")

        term.movePos(2,1)
        term.writeOutput("Current Date:")
        term.movePos(3,1)
        term.writeOutput("Current Time:")
        writeDateTime()
```

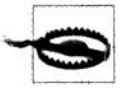
514

```
    term.movePos(5,1)
    term.writeOutput("Chan    Last Value")

    rownum = 1
    for row in range(6,14):
        term.movePos(row,1)
        term.writeOutput("  %d" % rownum)
        rownum += 1

    writeDataVals()

    term.movePos(15,1)
    term.writeOutput("Enter X to exit.")

# raw input() 处理程序线程
def getCommand():
    global exit_loop

    while True:
        instr = raw_input()
        if instr.upper() == 'X':
            exit_loop = True
            break
        time.sleep(0.1)

#--------------------------------------------------------------------
# 主循环
#--------------------------------------------------------------------
term = SimpleANSI.ANSITerm(None, 0)

exit_loop = False

# 启动 raw_input() 处理程序线程
getinput = threading.Thread(target=getCommand)
getinput.start()

mainScreen()

while exit_loop == False:
    writeDateTime()
    getDataVals()
    writeDataVals()
    time.sleep(0.2)
```

515

console6.py 实际上是示例 console4.py 的精简版。我再次使用了线程技巧来防止 Python 的 raw_input() 方法等待用户输入时阻塞主循环。同时也用到了 Python 的日期和时间方法。此处的主要目的是，展示如何通过 SimpleANSI 模块来使用 ANSI 控制序列构建基于一个静态模板的动态 ASCII 显示。

Python 和 curses



这里讨论的 curses 和 Python 的 curses 的实现，主要是针对 UNIX/Linux 系统上的 Python。Python 不会移植 curses 库模块到 Windows 系统上（Python 在 Windows 系统上不能导入 curses 模块）。尽管有一些替代方案，如 Windows 应用程序可以使用 *ANSI32.dll* 驱动提供的 SimpleANSI 模块。

虽然我并不怀疑软件开发已经导致大量富有创意的咒骂，但这一小节不是讨论编程的挫折的。本节内容是关于使用 ANSI 控制序列在终端显示上控制字符的 curses 库的。

我们已经看到使用几个简单的 ANSI 控制序列所能做的工作，而且如果你使用过 *vi*、*Emacs* 或者 *edt* 编辑器，你就已经看到了使用所有 ANSI 控制序列所能完成的工作的例子。尽管我们要认识迄今已经用到的 ANSI 控制序列的子集还需一个很长的过程，但认识所有 ANSI 序列的能力可以使用一些特定的程序。幸运的是，curses 库已经为我们准备好了这些。

curses 库提供了在离开其他区域时控制指定区域显示（也就是说，curses 提供了窗口）、改变文字风格和颜色、滚动文字等诸如此类的工作。你还能够用它创建下拉菜单、复选框和其他有用的用户输入工具。

516

curses 的历史

curses 源自 20 世纪 80 年代早期的 BSD UNIX 系统，后来被 AT&T 的 System V Release 4.0（SVr4）所采纳。在 Linux 系统上，ANSI 屏幕控制是通过 ncurses 库实现的（ncurses 意指“new curses”，一个开源的 SVr4 curses 库替代方案）。从现在开始，当我提及 curses 时，你也可以认为是 ncurses。

必须注意到 curses 是一个 UNIX/Linux 下的工具，而且在 Windows 上没有任何可用的 curses 库的本地移植。在运行于 Windows 系统上的 Python 程序中导出 curses 会出现一个错误，因为 Windows 版本的 Python 不支持 curses 库模块。虽然有过一些创建兼容 Windows 的 curses 方面的尝试，但我还没有用过它们，所以不便评论。

Python 的 curses 库模块

Python 的 curses 库模块是一个围绕一些标准 ncurses 库函式的简单的封装。它不是一个完整的封装，但它实现了相当一部分最常用的函式。我不打算在此涵盖整个 Python curses 库模块的内容，建议读者通过阅读 Python 文档来获取更多细节。我们将讨论 curses 背后的基本概念，并学习如何使用它在一个 Xterm 窗口上管理 ASCII 屏幕显示。

任何使用 curses 的 Python 程序必须首先初始化 curses 库，并实例化一个 curses 窗

口对象来管理整个显示区域。要结束一个 curses 会话并释放对屏幕的控制，可以调用 endwin() 函式。下面的例子演示了这个过程，启动一个 curses，什么都不干然后再随即关闭它：

```
import curses
stdscr = curses.initscr()

# 此处写 curses 语句

curses.endwin()
```

在 curses 中，主要的逻辑对象是窗口，显示中的各个窗口按照一定的层级关系进行排列，并将第一个窗口（stdscr，整个显示区域）作为父窗口。图 13-5 所示的是这种排列方式的伪 3D 示意图。

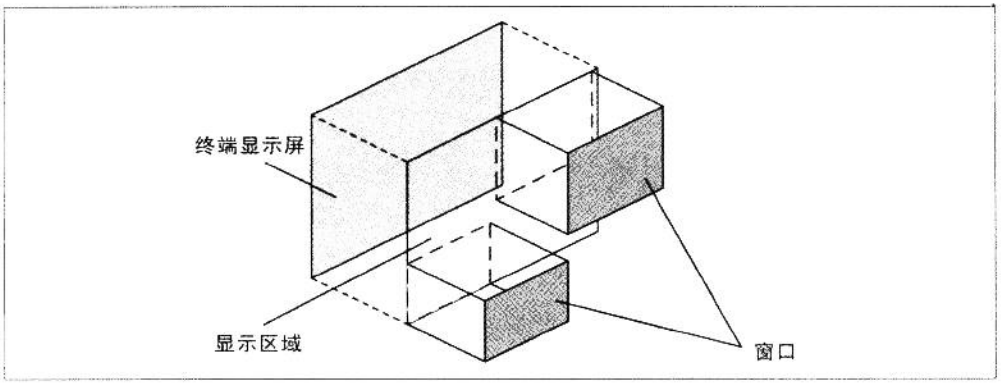


图13-5：curses窗口等级

写一个基于 curses 的应用基本上就是定义窗口，实现所有可能用到的动态控制（下拉菜单需要出现在适当的位置，光标可能需要响应 Tab 键等），然后将必要的文本和数据输入区域置于其中。curses 在其内部一直追踪各个物件与每个窗口之间的相对位置关系，如光标位置。听上去很简单吧，那就让我们一起来试试。

用 curses 做一个简单的数据显示

517

第一步，我们先把示例 console6.py 转换到 curses 实现。这能使我们看到之前用过的基本 ANSI 控制序列如何适用于 curses 方案中。注意，尽管输出相同，但它不能再在 MS Windows 系统上运行了，原因之前已经解释过。

curses 函式和方法有两种形式：从属于 curses 整体模块的模块级函式，以及在窗口对象上操作的窗口相关方法。这个项目中，我们不需要 curses 的所有功能，只会用到其中的一个子集。表 13-4 罗列了我们最开始将使用的 curses 库函式的定义，表 13-5 定义了

我们需要的窗口对象方法。方括号表示可选参数。

表13-4：curses库函式精选

函式	描述
cbreak()	打开 cbreak 模式。在这种模式下，普通的行缓冲无效，而且输入的字符可能从标准输入中被顺序地读取。特殊键字符，如 Ctrl+C 依然保持其原有功能
nocbreak()	退出 cbreak 模式，并返回普通的缓冲输入操作模式
echo()	打开回显模式，输入的每个字符会回显
noecho()	退出回显模式，输入字符不再回显
initscr()	初始化 curses 模块。返回的对象（一个窗口对象）表示整个显示屏幕。该窗口是顶级窗口
endwin()	关闭 curses 并返回到终端的正常状态
curs_set (*visibility*)	控制光标的显示。*visibility* 参数的值可以是 0、1 或者 2，作用分别是隐藏、强调或者完全可见。“完全可见”的确切意思依赖于所使用的终端（或者终端模拟器）
ungetch(*ch*)	将 ch 压入输入流的栈上，以使之成为 getch() 返回的下一个字符

表13-5：curses窗口方法精选

方法	描述
win.addstr([*y*,*x*], str[,*attr*])	如果指定了 [*y*,*x*]，则将 *str* 写入到此处，否则写到当前位置。如果指定了，则使用显示属性 *attr*。任何已存在于屏幕上的字符都会被重写
win.clrtoeol()	擦除从当前位置到行尾的所有字符
win.erase()	清空整个窗口
win.getch([*y*,*x*])	如果指定了 [*y*,*x*]，就从从终端的 [*y*,*x*] 处获取一个字符。返回值不一定是一个有效的 ASCII 字符，但应该是一些大于 255 的功能键之类的。在不延迟模式（非阻塞，使用 nodelay() 方法设定）下，如果没有可用的输入就会返回 -1；否则 getch() 将会阻塞等待直到一个键被按下。在其他方式出现前，假设返回值是一个整数
win.getstr([*y*,*x*])	如果指定 [*y*,*x*]，则从指定位置读取一个用户输入的字符串。提供有限的行编辑功能
win.move(*new_y*,*new_x*)	将光标移动到窗口的（*new_y*,*new_x*）位置
win.nodelay(*yes*)	控制 getch() 的阻塞行为。如果 *yes* 是 1，getch() 将不会阻塞
win.subwin([*nlines*, *ncols*],*begin_y*, *begin_x*)	返回一个子窗口对象。（*begin_y*,*begin_x*）定义了左上角位置，*ncols* 和 *nlines* 分别定义了宽度和高度。如果省略了 *ncols* 和 *nlines*，新的窗口将会延伸到显示区域的右下角

有趣的是，ANSITerm 实际上提供了比 curses 更底层的功能。换言之，curses 库在其方法中封装了很多底层的 ANSI 控制序列，而 ANSITerm 则将它们暴露出来。这并不出人意

料，因为 ANSITerm 实际上不过是一个对 ANSI 序列的漂亮封装，而且它并没有提供窗口管理或者其他的高级功能。

第一个例子 curses1.py 没什么新奇之处。它仅仅创建一个像 console6.py 那样不断更新的显示：

```
# !/usr/bin/python
# curses1.py
#
# 演示使用 Python 的 curses 模块，基于示例 console6.py 修改而来。
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import random
import time
import datetime
import curses
import traceback

data_vals = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
currdate = ""
currtime = ""
updt_cnt1 = 0
updt_cnt2 = 0

ran = random.random

def getDate():
    global currdate

    t = datetime.datetime.now()
    currdatetime = t.timetuple()
    yr = str(currdatetime[0])
    currdate = "%02d"%int(yr[2:]) + "%02d"%currdatetime[1] +\
        "%02d"%currdatetime[2]

def getTime():
    global currtime

    t = datetime.datetime.now()
    currdatetime = t.timetuple()
    currtime = "%02d:"%currdatetime[3] + "%02d:"%currdatetime[4] +\
        "%02d"%currdatetime[5]

# 将数据及时间写入屏幕
def writeDateTime(win):
    getDate()
```

519

```
        getTime()

        win.move(2,15)
        win.clrtoeol()
        win.addstr("%s" % currdate)
        win.move(3,15)
        win.clrtoeol()
        win.addstr("%s" % currtime)
        win.refresh()

    # 获取仿真的数据输入值
    def getDataVals():
        global data_vals, updt_cnt1, updt_cnt2

        data_vals[0] = ran() * 10.0
        data_vals[1] = ran() * 10.0

        if updt_cnt1 >= 4:
            for i in range(2,5):
                data_vals[i] = ran() * 10.0
            updt_cnt1 = 0
        else:
            updt_cnt1 += 1

        if updt_cnt2 >= 10:
            for i in range(4,8):
                data_vals[i] = ran() * 10.0
            updt_cnt2 = 0
        else:
            updt_cnt2 += 1

    # 写通道数据值
    def writeDataVals(win):
        idx = 0
        for i in range(6,14):
            win.move(i,10)
            win.clrtoeol()
            win.addstr("%6.2f" % data_vals[idx])
            idx += 1

        win.refresh()
        # 当更新完毕时将光标放在显示文字之后
        win.move(16,1)

    # 生成主显示
    def mainScreen(win):
        win.erase()

        win.move(1,1)
        win.addstr("Input Data Monitor")
```

```
    win.refresh()

    win.move(2,1)
    win.addstr("Current Date:")
    win.move(3,1)
    win.addstr("Current Time:")
    win.refresh()

    writeDateTime(win)

    win.move(5,1)
    win.addstr("Chan    Last Value")
    win.refresh()

    rownum = 1
    for row in range(6,14):
        win.move(row, 1)
        win.addstr("  %d" % rownum)
        rownum += 1
    win.refresh()

    writeDataVals(win)

    win.move(15,1)
    win.addstr("Enter X to exit.")
    win.refresh()

def mainloop(win):                                          521
    win.nodelay(1)  # 禁止 getch() 阻塞
    # 绘制主显示模板
    mainScreen(win)

    # 直到用户想退出才停止运行
    while 1:
        # 检查键盘输入
        inch = win.getch()
        # 如果没有合法字符则 getch() 返回 -1
        if inch != -1:
            # 检查 inch 是否是真正的退出字符
            instr = hr(inch))
            if instr.upper() == 'X':
                break
        writeDateTime(win)
        getDataVals()
        writeDataVals(win)
        time.sleep(0.2)

def startup():
    # 借用 David Mertz（很早就在 IBM 连载可爱的 Python 系列文章的 Python 达人）的主意
    # 使用 try-except 结构包装初始化过程
```

```
    try:
        # 初始化 curses
        stdscr = curses.initscr()

        # 关闭键回显，进入 cbreak 模式，此时对键盘输入不使用缓冲功能
        curses.noecho()
        curses.cbreak()

        mainloop(stdscr)                    # 进入主循环

        # 全部返回正常模式
        curses.echo()
        curses.nocbreak()

        curses.endwin()                     # 结束 curses
    except:
        # 一旦出现错误，重置终端
        curses.echo()
        curses.nocbreak()
        curses.endwin()
        traceback.print_exc()               # 打印异常

if __name__=='__main__':
    startup()
```

在 curses 中，屏幕坐标用（y, x）顺序指定——等同于（*row*, *column*）——ANSITerm 类也是这么做的。主窗口对象（stdscr）被创建后，调用 noecho() 和 cbreak() 关闭本地回显和输入缓冲。最后，注意 mainloop() 使用 stdscr 作为它的唯一参数被调用。其后使用的所有窗口方法都是该对象的方法。

curses 内部的窗口对象管理方法不支持自动发送输出内容到屏幕上，因此，你必须通知它何时去刷新屏幕。造成这个问题的原因之一，可能是因为使用了一个非常复杂的
522 curses 驱动的显示。与数据更新时屏幕上闪烁着的独立的行不同，refresh() 方法能够被用来刷新整个屏幕，从而改变一组显示项或者独立的窗口，甚至一次刷新全部控件。使用类似 refresh() 这样的方法，另一方面的原因是当一个像 addstr() 这样的方法被调用时，它改变了在特定窗口内部用于显示的数据，但没有自动传递这个改变到显示器。你必须通知它什么时候去做这些事。

增加一个子窗口

假设你想要一个弹出的子窗口来响应用户输入。首先，要定义一个窗口。在 curses2.py 中，我将子窗口对象封装到了 openSubWindow() 函式中，这段代码被放到 mainloop() 之前。

```
def openSubWindow(win):
    # 创建子窗口并保持打开，直到用户按下 X 键。
```

```
    subwin = win.subwin(10, 30, 10, 10)
    subwin.nodelay(1)       # 禁止 getch() 阻塞
    subwin.erase()
    subwin.bkgdset(' ')
    subwin.refresh()
    subwin.addstr(3, 0, "Enter X to exit subwindow")
    subwin.refresh()
    while 1:
        inch = subwin.getch()
        if inch != -1:
            instr = chr(inch)
            if instr.upper() == 'X':
                break
        time.sleep(0.2)
```

这个新函式在 mainloop() 函式中使用一个新增的 W 命令来调用，如下所示。

```
def mainloop(win):
    win.nodelay(1)          # 禁止 getch() 阻塞
    # 绘制主显示模板
    mainScreen(win)

    # 直到用户想退出才停止运行
    while 1:
        # 检查键盘输入
        inch = win.getch()
        # 如果没有合法字符则 getch() 返回 -1
        if inch != -1:
            # 检查 inch 是否是真正的退出字符
            instr = chr(inch)
            if instr.upper() == 'X':
                break
            if instr.upper() == 'W':
                openSubWindow(win)
                # 恢复主屏幕
                mainScreen(win)
        writeDateTime(win)
        getDataVals()
        writeDataVals(win)
        time.sleep(0.2)
```

523

当用户按下 w（或 W）键时，就会看到一个 10×30 字符的显示区域，上面写着一行“Enter X to exit subwindow.”当子窗口函式退出时，mainloop() 函式重绘显示区域，以覆盖子窗口所留下的空白区域。

当然，还有一些更优雅的方法来处理子窗口的创建和删除，只是采用这种方法看上去更简单。示例 curses2.py 没有为子窗口绘制边框，也没有使用阴影或者其他颜色来填充背景。curses 库中这些特性的实际行为在极大程度上依赖于显示将如何处理 ANSI 命令序列，而这

又依赖于系统的 termifo 的定义。如果你打算在你的应用中使用 curses，最好能花一些时间来学习 Python 的 curses 模块和 terminfo 的帮助手册。另外，还得看一下本章最后“推荐阅读”这一节。

用不用 curse 是个问题吗

如果你不需要 curses 中所提供的高级功能，而且希望你的应用不依赖一个完整的 GUI 库就可以移植，你可能会希望使用一些像 SimpleANSI 中所用的简单的方法来完成它。大多数仪器应用实际上并不需要花哨的控制界面，而且我们已经看到过如何创建具有实时更新功能的数据显示画面。

然而，curses 提供了一些很难用简单的 ANSI 库重现的功能，至少不能脱离一些重要的代码。而且，由于 curses 已经被写好了，并且很成熟，因此，如果你需要在你的应用中创建子窗口、菜单、鼠标控制和对话框，并且不需要在 Windows 上运行这样的程序，使用 curses 会更有意义。

虽然 curses 不是一个跨平台的解决方案，但它却是显示接口兼容的。任何能够显示 ASCII 字符并能解释 ANSI 控制序列的系统都可能支持 curses 驱动的显示。包括运行在 Windows 上带有 VT100 仿真能力的终端仿真器，Linux 系统上的 Xterm 窗口，甚至一个像 VT100 这样的哑终端。在一个仪器系统中，使用一个 ANSI/ASCII 终端或者终端仿真器，通过串口或者网络远程连接到一个远程主机系统上，听起来也不是那么疯狂。安装了许多程序的服务器经常使用 curses 来显示系统状态或者作为控制接口，甚至在没有主控制台（也没有 GUI）时，允许操作员直接访问系统。

524 在一个仪器系统中，你可能想使用一种简单的通信协议对一个远程系统进行监控，而且在主控 PC 和各种传感器及执行机构之间通过一条细小的数据线，而非一条粗笨的电缆连在一起。或者这些设备由于某些原因（有爆炸危险、噪声、辐射、散热以及低温限制等）不太适合与主机放在一起。在这些情况下，使用 ANSI 控制序列来产生一个易读、优雅和高效的数据控制显示是一个敏捷且相对简单的方法。

（向莎士比亚道歉，原文：My apologies to W.Shakespeare，指标题使用了莎翁那句著名台词的结构。）

图形用户界面

所有通用的现代操作系统都会在其设计中采纳某种图形用户界面（GUI）。只有那些专为低层嵌入式应用而设计的操作系统才没有自带 GUI，因为它们根本不需要（事实上，它们通常没有任何用户显示接口）。

GUI 是操作系统核心最上层的功能，甚至在很多情况下都是可选的。在 UNIX/Linux 系统中，操作系统以一系列分散的启动步骤被加载后 GUI 才会启动，在 Linux 上，没有 GUI 系统也可以运行。在 Windows 上，GUI 被整合到系统实现中，虽然从技术上讲还是独立的一层功能，但它被设计成不容易取消的。

这一节我们将学习如何使用 Python 创建一个 GUI。我不会陷入细节的代码示例中，这主要是因为示例如果超过简单的“Hello World”的 GUI，往往就会变得很复杂。我们接下来会关心 GUI 的概念是如何产生的，为什么你要使用它，而不是用我们之前介绍过的简单的命令行或者 ANSI 方案。我们将通过一些简单的 GUI 数据显示示例，来展示如何创建一个 GUI。

图形用户界面的历史和概念

在介绍一组 Python 的 GUI 工具之前，我们先来回顾一下现在所熟悉的 GUI 的历史。很多人依然认为是苹果发明了 GUI，但事实并非如此。第一个真正可用的 GUI 是 20 世纪 70 年代在施乐公司的 Palo Alto 研究中心（PARC）发明的，运行于一台施乐公司的计算机系统上。史蒂夫·乔布斯在 1979 年的一次 PARC 设备展览会上看到了它，由此 Apple Lisa 和 Mac 便由此接踵而至。当微软看清楚了苹果的发展方向后，就决定放弃 DOS 命令行，转而开发 Windows。最初运行于 UNIX 上，后来又被移植到 Linux 上的 X Window 系统，也是为了向用户提供图形界面体验。而施乐公司自己最终却未能及时地 <525> 市场化一个通用系统来从他们自己的发明中获取收益。

现代计算机系统的图形界面包含了多层抽象。GUI 肩负诸多责任，如管理应用驱动的窗口在主显示设备上布局，跟踪所谓 Z-order（窗口之间如何叠加）的信息，保证窗口在需要重绘自身某部分显示区域时收到通知。每个应用窗口都有自己的机制来管理用户输入、数据输出、重绘或重获焦点。在某些类型的窗口显示中，如多文档界面（MDI），应用的窗口是一个“父”窗口，可以包含一个或者多个“子”窗口，每个窗口都可以包含一个菜单栏、多个按钮、对话框、图片显示和其他组件（即所谓的控件）。每一层都有其自己的显示管理功能，并且能够在父窗口的上下文中与其他窗口或者系统中所有上层 GUI 进行通信。

图形用户界面并不仅比命令行或者其他基于文本的界面更美观，它也更直观和易于理解。然而，你可能已经根据前面的章节意识到，这种易用性并不是没有代价的。

GUI 比一个提供相同功能的文本界面更为复杂。GUI 并不是某个应用功能的一小部分，在大多数情况下，它扮演系统整体应用的角色。换言之，如果你想为某个应用添加一个 ASCII 文本类型的界面，基本上只需关心如何使用 ANSI 终端控制序列通过传统的 I/O 调用来操作显示。除此之外再也没什么特别的了。而对于一个 GUI，窗口管理器不仅仅

管理当前显示的界面元素，还要将应用的功能封装到其框架之中。如果你想在 GUI 中提供一个“外部（outboard）”功能，你可能还得考虑使用外部进程、命名管道、套接字和其他形式的进程间通信（IPC）来实现这种机制。如此一来，使用 GUI 很快就会变得越来越复杂。

那么，为什么还要用 GUI 呢？因为 GUI 更形象，且功能更为强大。GUI 可以比单纯的 ASCII 更准确而生动地展现数据，曲线表和图表就是最好的例子。而且 GUI 还能够显示图像，并通过用户与图像进行交互的方式来处理用户的输入。不少高端的工控系统都使用了这样的用户界面，例如，根据实时数据显示一个特定对象的示意图，用户可以通过点击这些对象所对应的图形来操纵该对象。

使用 GUI 还能表现出 3D 的效果，甚至一个可以让用户通过变换角度、平移、放大或缩小来观测的固体模型，整个过程会连续性地刷新显示。某些激光干涉仪系统使用这样的显示方式，通过连续地获取新的测量结果来分析一个视觉平面的细微特征。

526 虽然有这么多动人的特性，我还是打算只介绍一些易于实现的 GUI。如果你想了解更多内容，可以参看本章最后“推荐阅读”一节。还可以获得一些你喜欢的 GUI 应用程序的源代码（如果它们是开源的），从而学习它们是如何实现的。

在 Python 中使用 GUI

这一小节我将介绍两个 Python 中通用的 GUI 工具包：TkInter 和 wxPython。尽管两者最终效果在本质上是相同的，但当你需要选择使用哪一个时，要知道它们在使用以及能完成的工作等方面的基本差异。此外，我们还会学习一些设计和布局界面的工具。

在正式开始之前，我们需要规定一些术语。但首先，我要指出的是，这些术语是可互换的，因此有时候并不准确。我会尽量在这一章中保持一致，希望能减少潜在的冲突。

GUI 中主要有两种类型的对象：容器和控件。一个容器用来组合其他容器或者控制器，并在其内部维护这些 GUI 对象之间的功能上的关联。容器包括窗口、框架、对话框和模态窗口。窗口类对象的一个显著特性是它能够在一个层次结构中，作为一个顶层对象存在于另一个窗口类对象内部。

控件是指那些提供用户交互功能的 GUI 对象。控件可以是按钮、滚动条或用于绘图的画布；也可指由文本区域或者其他简单控件合成的复杂控件。在某些工具中，还有一些特殊的容器控件，它们没有真实的窗口，但能够用来在逻辑上组织一系列控件相关的功能，如面板控件。控件会作为成员对象存在于一个窗口对象的上下文中。

当然了，事情总是不会被定义得那么简单明了，还是会有一些例外情况。例如，一些窗

口对象，如框架可以同时是一个顶层对象和另一个窗口的下级对象。还有，一个控件可以像容器一样工作，尽管它也必须有一个父窗口。理解 Python 中 GUI 元件都是对象这一本质很重要，而且通常都是通过继承一个已经存在的类来创建新对象的。只要你理解了 GUI 如何生成及其如何工作，这段灰色地带就不再那么麻烦。

眼下，你可以认为一个窗口对象就是控件或其他窗口所在的那块区域，而且那些窗口不会接受输入，除非它们内部还有其他控件。

考察一个 GUI 有三种基本方法：一是子类继承自父类定义的面向对象（OO-based）的软件；二是按层次结构组织各对象的结构体，这些对象共享显示区域，但从面向对象（OO）的角度来看，它们之间又没有必然的联系；三是从功能上看，它是一个由事件驱动的应用软件。我们下面开始从 OO 的角度来学习它们。

527

GUI 对象

在采用了面向对象方法的工具包中，每个窗口或者控件都在一个类中定义，每个类提供了一套方法和属性。wxPython 就是其中之一，而且其中的 wxWidgets 是非常 OO 的设计。Tk 包起初还不是一个真正的 OO 实现（tcl 不是一种 OO 的语言），但是通过 TkInter，它也有了封装的类、对象和方法。采用 OO 方法也意味着你可以通过继承一个已经存在的类来创建一个新的控件类，可以重写一些已经存在的方法或者添加新的方法。

在大多数情况下，一个控件会继承自一个已有的窗口或者控件，而这些窗口或者控件也同样可能继承自其他父类。也就是说，一个框架对象可以继承自一个窗口对象，而一个按钮控件可以继承自一个控件对象，如图 13-6 所示。

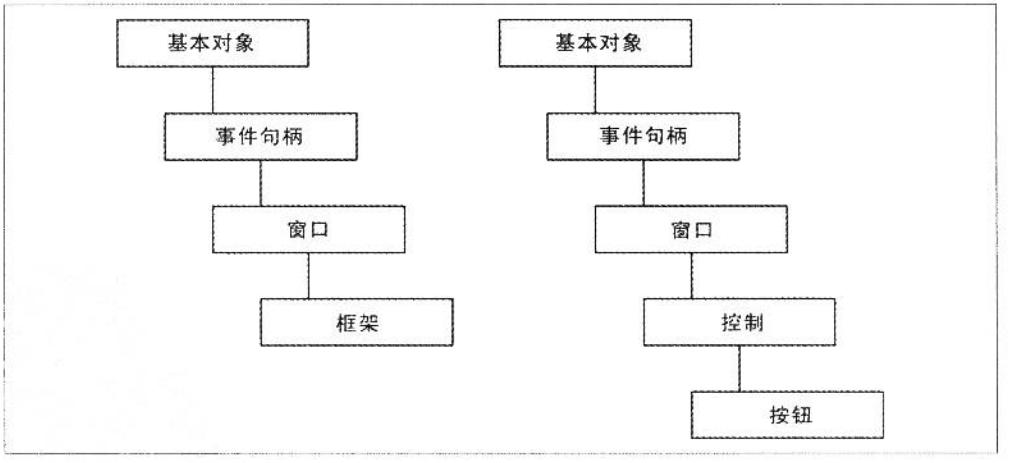


图13-6：GUI对象继承

也不是所有的 GUI 实现都遵循 wxPython 中的这一机制，但是窗口对象和控件对象之间的主要差异，在后面将要用到的两个工具包中都是普遍存在的。

基本的 GUI 显示结构

GUI 通常被组织为层次结构的图形元件，其中包含有一个父对象，或者叫基础对象，通常是一个框架类的控件(参见图 13-6)。这个对象用来充当锚位或者 GUI 子对象的父对象，这些子对象自己也可能是其他对象的父对象，以此类推，如图 13-7。这不是一个继承的情况，但却相当于功能关系的分层。

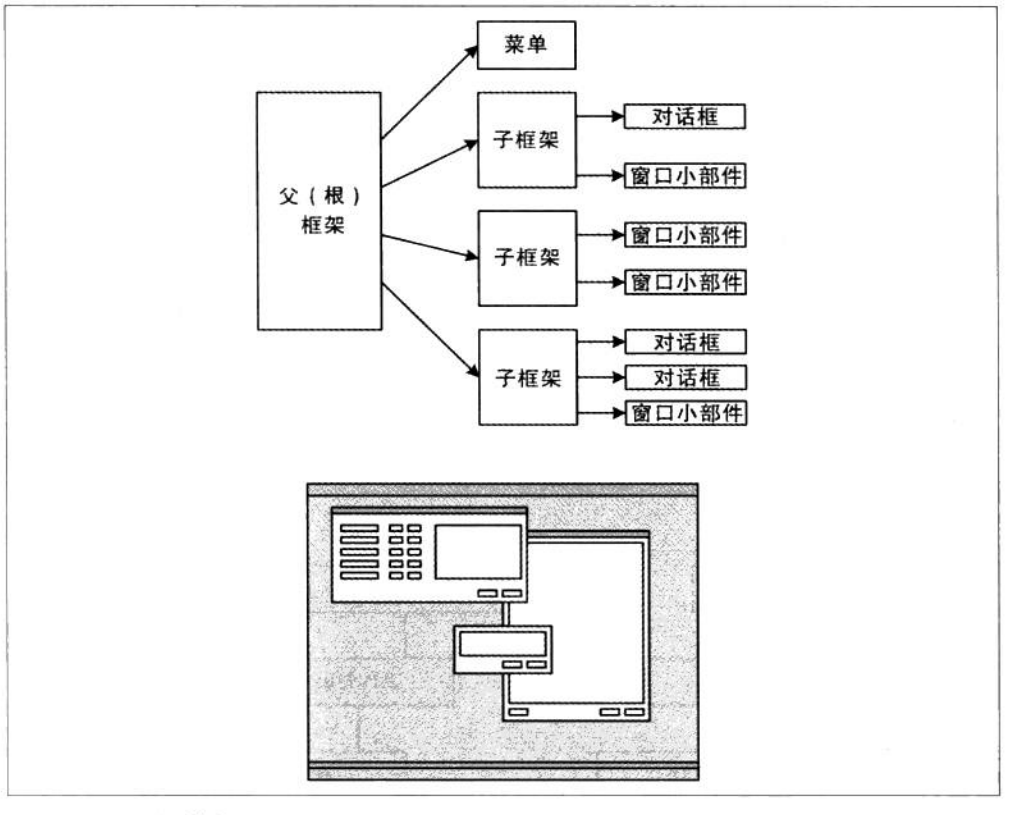


图13-7：GUI对象等级

528 图 13-7 所示的树状示意图中，每个子对象都有一个父对象和一些属于自己的子对象（也可能没有）。当控件绑定到一个类似于框架的窗口对象上时，就可以通过父对象管理这一组对象。

GUI 功能

从功能上讲，GUI 基于一个刺激和响应模型。刺激是指一个事件，这个事件可能来自用户、内部进程或者软件之外的其他地方。当用户输入一个命令，可能是使用鼠标单击一个按钮或者按下一个按键，一个事件就会产生，而且 GUI 会做出某种方式的响应。在等待命令期间，GUI 或许会接受外部事件并刷新数据显示，甚至通过定时器等方法产生内部事件。这就是所谓的事件驱动程序设计，它是当前 GUI 的主要模型。

事件由分配给事件源的事件处理程序（handler）处理。这个过程称为绑定，一个事件处理程序被认为绑定到一个指定的事件。事件处理程序可以是自带的，也可以调用其他函式或者方法来实现某些特定处理。某些事件处理程序集成在 GUI 框架中，其中包括通知 529 显示更新，例如，用户将一个窗口移到另一个窗口之上，或者所有打开的窗口突然全部关闭。

但我们最关心的还是 Python GUI 中的各种控件所产生的事件。每个按钮、文本框、滚动条等，必须有一个与之关联的事件处理程序。这些在 TkInter 中通常称之为“回调”，而 wxPython 则是一个“事件处理程序（event handlers）”，但其最终作用相同。

GUI 是事件驱动的，因此显示通常会被挂起，直到它接收到某种事件。如每隔 1 秒左右刷新一次来自某仪器的数据。除非有某种方式产生一个事件，调用某个函式来查询设备的新数据并刷新屏幕，否则屏幕会一直僵在那里等着用户单击“刷新”按钮来手动请求新数据。这听上去很简单，但是很多 GUI 开发新手都曾坐在那里凝视着他们的杰作，思考如何才能不必重复单击一个按钮就能看到界面的变化。解决办法就是使用某种内部定时器，它能够产生一个事件来从诸如某种仪器这样的外部资源中获得新数据。

wxPython 在其设计框架中包含了一些定时器函式，而且用起来相对直观。TkInter 采用了不同的方法，在控件内部产生一个事件，并能够在一定时间之后调用一个回调函式。这个虽然不是很直观，但是如果使用得当，它们也能够很好地工作。

GUI 主循环

GUI 在一个主循环中执行，如图 13-8 所示，和我们在 ANSI 字符界面中所见到的情况相似，主循环的主要作用是启动界面、处理事件和消息，并在退出时关闭所有内容等。

`mainloop()` 方法在 TkInter 和 wxPython GUI 应用中都是最基本和必需的部分。这两个工具包在主循环实现的细节上稍有不同，但最终的目的一样。主循环开始后，程序不会返回，除非 GUI 关闭或者程序退出。

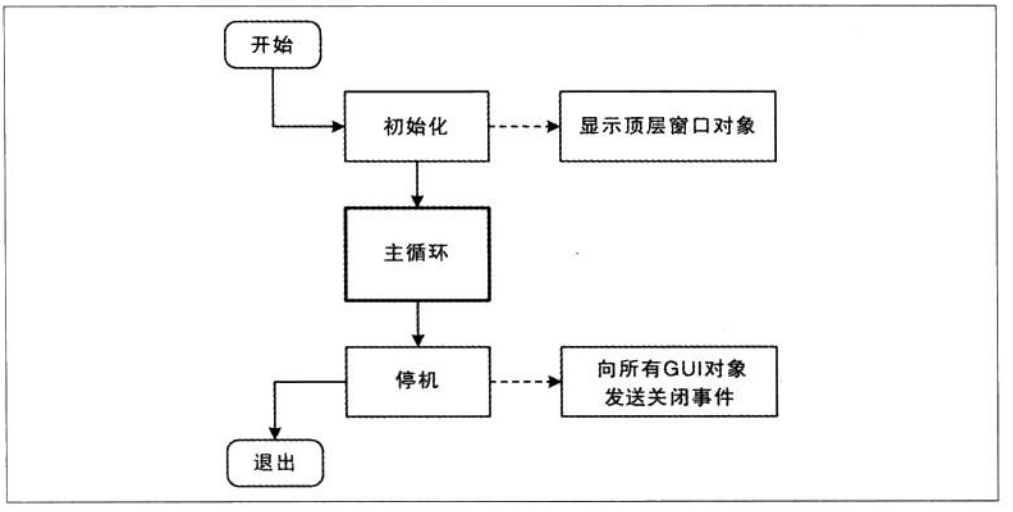


图13-8：GUI主循环

TkInter

Python 将 TkInter GUI 工具包作为其内置标准发行的一部分。TkInter 的意思是 tcl/Tk 语言的 Tk 部分（tcl 和 Tk 通常被当做一个东西）。Tk 控件工具用起来相当简单，而且它非常成熟，它大约已经存在近 20 年了。

530

设计你的 GUI

在动手实现一个 GUI 项目之前，首先要确定界面的外观。实现一个 GUI 需要考虑很多问题，如使用哪些控件，如何管理控件的布局，需要实现哪些功能（当用户和界面交互时要做什么），必要时甚至需要考虑其内部事件生成器。即便是那些不是十分复杂的 GUI，也不能指望把它们放在一起闭上眼睛就能完成任务。在开始前，需要花费一些时间对此进行思考。

在设计一个 GUI 布局时，我喜欢先将我所设想的 GUI 画出来，然后以此为模板进行下一步工作。你可能有一些不同的方法，但至少要先在一张纸上画出你的 GUI 布局，这在开始定义控件或者写代码时，你会发现它非常有帮助。

几何管理

TkInter 提供了三种形式进行控件布局，又称做几何管理（geometry management），分别是打包（pack）、网格（grid）和手动定位（placement）技术。其中，打包方法是最简单、最常用的方法。通过打包，控件从外部边缘到中心处被添加到窗口，效果上使之尽可能

接近窗口中心。

当一个控件由打包管理时，它在屏幕上的位置就可以紧贴着窗口边缘或者另一个控件，新加入的控件也会紧挨着它。打包几何管理器会尝试以占去窗口最小控件的方式来排列控件。打包最大的缺点是没有提供一些直接控制控件的结束位置的方法，尽管你可以通过添加一个“填充控件”（padding widgets）来实现这一功能（相当于一个空的控制面板之类的控件）。如果顶层的框架大小被改变，控件的位置会被打乱。 531

几何定位管理器使用窗口中 x 和 y 像素数值的绝对位置来定位控件。通过定位，我们就可以创建各种可视化组件，除非窗口的确很小也很简单，否则建议使用排版工具来辅助定位而不是手工设定绝对位置。

我倾向于使用 TkInter GUI 提供的网格（grid）布局方法。网格几何管理器将窗口划分成多个网格。列和行的宽度和高度由控件大小决定。

简单的 TkInter 示例

下面是一个简单的 TkInterGUI 的代码，tkdemo1.py。它使用了网格几何管理器将 6 个控件放到两列中：

```
#! /usr/bin/python
# tkdemo1.py
#
# 演示 TkInter 和网络安置方法
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

from Tkinter import *

class demoGUI(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.createWidgets()

    def createWidgets(self):
        # 获得顶级帧定义
        top=self.winfo_toplevel()
        # 在窗口管理器中设置起始位置
        top.wm_geometry('+50+100')

        # 设置窗口标题
        self.master.title("Demo 1")

        # 配置全局网格行为
```

```
        self.master.rowconfigure( 0, weight = 1 )
        self.master.columnconfigure( 0, weight = 1 )
        self.grid(sticky = W+E+N+S)

        # 使用标签控件所使用的字符串对象
        self.var1 = StringVar()
        self.var1.set("")
        self.var2 = StringVar()
        self.var2.set("")

        # 输出状态切换标志
        self.toggle1 = 0
        self.toggle2 = 0

        # 创建三个按钮和三个标签控件，其中之一作为占位符（现在）。

        # 绑定按钮 1 和 2 的事件处理程序。

        # 两个激活的标签部件上的文字会显示为黑色底绿字。

        self.button1 = Button(self, text="Button 1", width=10)
        self.button1.grid(row=0, column=0)
        self.button1.bind("<Button-1>", self.button1_Click)

        self.text1 = Label(self, text="", width=10, relief=SUNKEN,
                           bg="black", fg="green",
                           textvariable=self.var1)
        self.text1.grid(row=0, column=10)

        self.button2 = Button(self, text="Button 2", width=10)
        self.button2.grid(row=1, column=0)
        self.button2.bind("<Button-1>", self.button2_Click)

        self.text2 = Label(self, text="", width=10, relief=SUNKEN,
                           bg="black", fg="green",
                           textvariable=self.var2)
        self.text2.grid(row=1, column=10)

        self.button3 = Button(self, text="Quit", width=10,
                              command=self.quit)
        self.button3.grid(row=2, column=0)

        # 虚拟的空间填充物
        # 你可以修改显示的内容
        self.text3 = Label(self, text="", width=10)
        self.text3.grid(row=2, column=10)

    def button1_Click(self, event):
        if self.toggle1 == 0:
            self.var1.set("0000")
```

```
            self.toggle1 = 1
        else:
            self.var1.set("1111")
            self.toggle1 = 0

        print "Button 1"

    def button2_Click(self, event):
        if self.toggle2 == 0:
            self.var2.set("0000")
            self.toggle2 = 1
        else:
            self.var2.set("1111")
            self.toggle2 = 0

        print "Button 2"

app = demoGUI()
app.mainloop()
```

533

在 Windows 系统中，示例 tkdemo1.py 会创建一个如图 13-9 所示的对话框。在 Linux 上运行的结果几乎与此一样，只是不同的窗口管理器在风格上稍有差异。

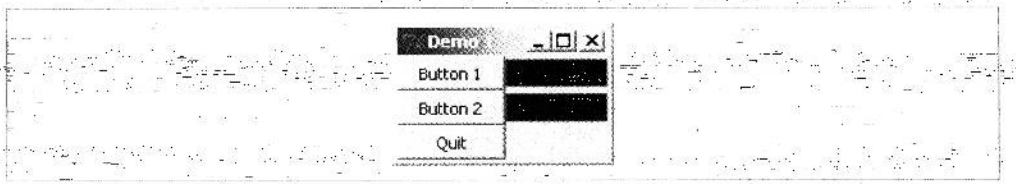


图13-9：TkInter图形界面示例

在下面的例子 tkdemo2.py 中，我使用 label 控件实现文本重复输出，因为这样比较简单。坦白地讲，Tk 的文本控件用起来很痛苦，所以我尽量不用它。下面是 tkdemo2.py 的源代码：

```
#!/usr/bin/python
# tkdemo2.py
#
# 第二版 TkInter以及网格安置方法演示
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.
from Tkinter import *
import time

class demoGUI(Frame):
```

```
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.createWidgets()

    def createWidgets(self):
        # 获得顶级帧定义
        top=self.winfo_toplevel()
        # 在窗口管理器中设置起始位置
        top.wm_geometry('+50+100')

        # 设置窗口标题
        self.master.title("Demo 2")

        # 创建标签部件将使用的字符串对象
        self.var1 = StringVar()
        self.var1.set("")
        self.var2 = StringVar()
        self.var2.set("")

        self.master.rowconfigure(0, weight = 1)
534     self.master.columnconfigure(0, weight = 1)
        self.grid(sticky = W+E+N+S)

        self.text1 = Label(self, text="", width = 15, height = 4,
                           relief=RAISED, bg="white", fg="black",
                           textvariable=self.var1)
        self.text1.grid(rowspan = 2, sticky = W+E+N+S)

        self.button1 = Button(self, text = "RUN", width = 10, height = 2)
        self.button1.grid(row = 0, column = 1, sticky = W+E+N+S)
        self.button1.bind("<Button-1>", self.button1_Click)

        self.button2 = Button(self, text = "STOP", width = 10, height = 2)
        self.button2.grid(row = 0, column = 2, sticky = W+E+N+S)
        self.button2.bind("<Button-1>", self.button2_Click)

        self.button3 = Button(self, text = "Test", width = 10, height = 2)
        self.button3.grid(row = 1, column = 1,sticky = W+E+N+S)
        self.button3.bind("<Button-1>", self.button3_Click)

        self.button4 = Button(self, text = "Reset", width = 10, height = 2)
        self.button4.grid(row = 1, column = 2, sticky = W+E+N+S)
        self.button4.bind("<Button-1>", self.button4_Click)

        self.entry = Entry(self, relief=RAISED)
        self.entry.grid(row = 2, columnspan = 2, sticky = W+E+N+S)
        self.entry.insert(INSERT, "Command")

        self.text2 = Label(self, text="Stopped", width = 2, height = 2,
                           relief=RAISED, bg="white", fg="black",
```

```
                                textvariable=self.var2)
        self.text2.grid(row = 2, column = 2, sticky = W+E+N+S)

        self.rowconfigure(1, weight = 1)
        self.columnconfigure(1, weight = 1)

    def button1_Click(self, event):
        self.var1.set("")
        self.var2.set("Running")

    def button2_Click(self, event):
        self.var1.set("")
        self.var2.set("Stopped")

    def button3_Click(self, event):
        time.sleep(1)
        self.var1.set("Test OK")
        self.var2.set("Stopped")

    def button4_Click(self, event):
        time.sleep(1)
        self.var1.set("Reset OK")
        self.var2.set("Stopped")

app = demoGUI()
app.mainloop()
```

535

tkdemo2.py 的输出如图 13-10 所示。

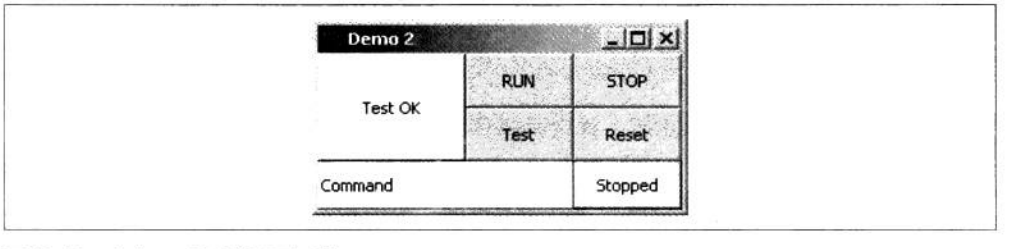


图13-10：tkdemo2图形界面示例

tkdemo2.py 中有一些地方需要注意。首先怎样使用是网格几何管理器用来组织不同大小的控件，其次是按钮如何使一个事件作用于多个 label 控件上。最后，你可能已经注意到，当单击 Test 或者 Reset 按钮时，GUI 会有短暂的冻结。这是因为这些按钮的事件处理程序中调用了 time.sleep(1)。这个例子也说明了为什么响应一个 GUI 事件所需的时间，应该小于处理一个输入事件所需的时间来完成响应，不能比这更慢了。

TkInter 的工具和资源

TkInter 提供了几个很不错的布局工具。当然，Google 是个好的开始。SourceForge（*http://www.sourceforge.net*）也不错。我个人则比较倾向于手动创建 GUI。尽管如此，还是应该在这里介绍一些或许你会觉得有用的 TkInter GUI 构造工具。

PAGE（*http://page.sourceforge.net*）

PAGE 工具用来创建单窗口，但并不能用来构造完整的应用。它是一个可靠的工具，用来减少创建一个含有多控件的复杂窗口时所需的枯燥的工作。在使用前一定要阅读介绍文档，以确保你完全理解 PAGE 的目标与限制。

SpecTcl（*http://spectcl.sourceforge.net*）

这个工具有一段时间没有更新过了，而且可能会有一些古怪的行为，但是我最后一次使用它时还是觉得很不错。如果你有兴趣，或许值得一试。

要阅读和查找 TkInter 的资料，可以参见本章最后的“推荐阅读”一节。

wxPython

wxPython 在 Python GUI 领域还比较年轻，但它的底层库已经存在有一段时间了。它实际上是 C++ 工具包 wxWidgets 的封装。wxWidgets 的一个主要目标是实现移植过程中保持主操作系统的外观体验。

536

我倾向于用 TkInter 通过网格几何管理器手动地构建一个简单的 GUI，而我更喜欢使用 wxPython 和一些辅助工具以像素为单位精确地指定控件的像素坐标位置。为了达到这个目的，我采用了 Boa Constructor wxPython 图形用户界面设计器，它集成了很有用的调试器和文本编辑器。你可以从 *http://boa-constructor.sourceforge.net* 下载 Boa。

wxPython 有一些很有用的功能，如定时事件生成器，通过子类化 GUI 对象来创建复杂的功能，线程安全操作，易于与其他库整合，如 NumPy 和 PIL（the Python Imaging Library）。但是，TkInter 也能做上述的很多事情，而且它还有出色的画布控件，所以我不提倡贬低它们当中的任何一个来抬高另一个。但是，正如其他一切事物那样，完成一个工作总有相对简单的方法。这一切都归结为要为工作选择正确的工具。

设计一个 wxPython GUI

就像我之前使用 TkInter 那样，为 GUI 做一下设计是个好主意。尽管像 Boa 这种可视化工具使得尝试不同的控件排列很简单，但它还是容易使你陷身于一些细节当中。还是使事情简单一些好。

决定了你的 GUI 应用外观之后，就启动 Boa（假设你已经安装了它）。在它初始化之后，

你就能看到如图 13-11 所示的三个窗口。

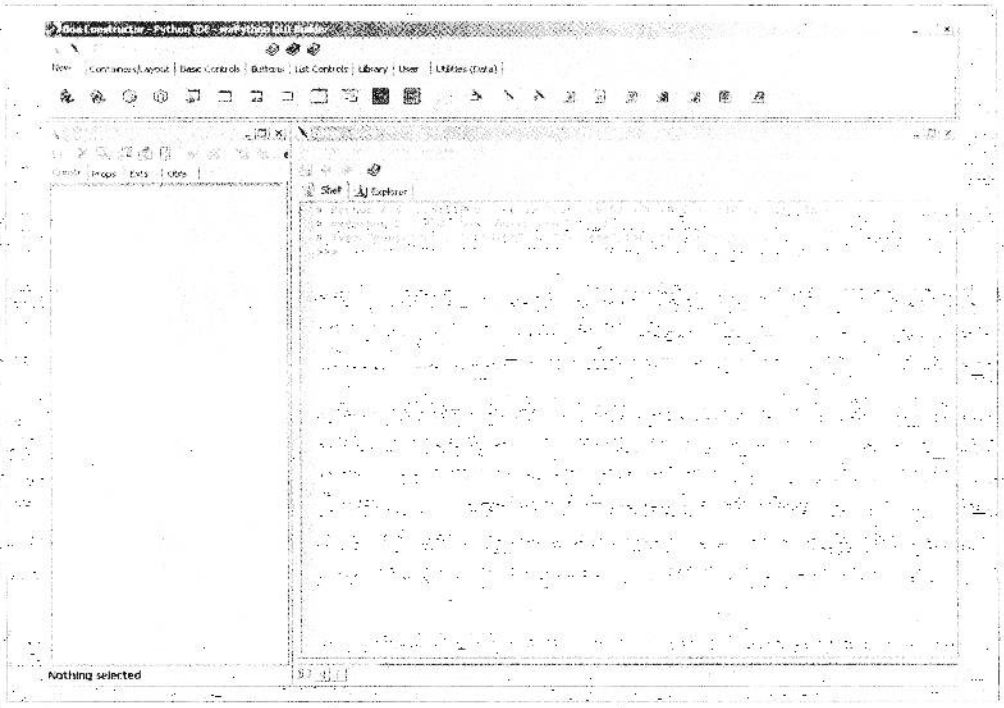


图13-11：Boa的设计界面

创建一个简单的 wxPython GUI

Boa 可以用来创建完整的 GUI，从最上层的 wx.app 对象到它所需要的各种框架和对话框。然而如何整合这些到一个能够工作的应用中还是要靠你自己。我们将再次看到一个简单的单框架对话框类型的 GUI，和之前使用 TkInter 所创建的有点相似。

这个 GUI 将有一些数据显示区域、一些按钮，以及一些位图状态指示器。它事实上是一个我们本章开头所见到的 ANSI 数据显示的 GUI 变种。

Boa 运行后，首先要创建一个新的设计。在顶部窗口，选择“New”选项卡（它应该已处于选中状态）。然后选择第 6 个按钮，wx.frame。现在，主窗口会出现一些 Python 代码，这是我们要创建的应用的主干部分。

接下来进入一个有意思的环节。选择代码编辑器显示区域上方工具栏中有蓝色箭头的图标，你会看到一个如图 13-12 所示的新窗口。

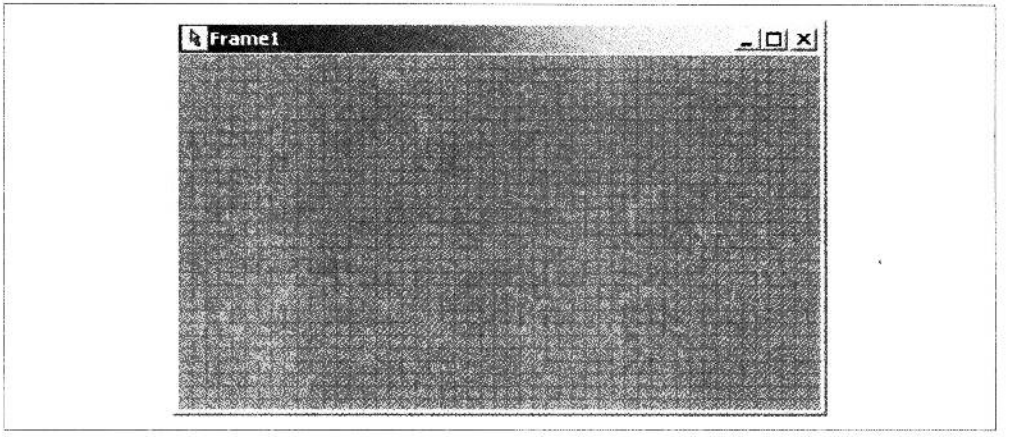


图13-12：Boa的窗口排版界面

538 我打算快进一下，因为我打算向你展示 Boa 的用途，但不想陷入细节步骤（那又需要一本书来介绍了）。我已经从窗口最上面的工具条选好各种控件，并将其丢入布局好的区域。而且我已经用窗口最左面的控制面板编写了好多个数据项。这允许你做很多事情，诸如设置标题栏、改变字体和颜色、定义大小，或是设置按钮或者文本控件中使用的各种文本字符串。

图 13-13 所示的是这几分钟有意义的工作后的最终的结果。

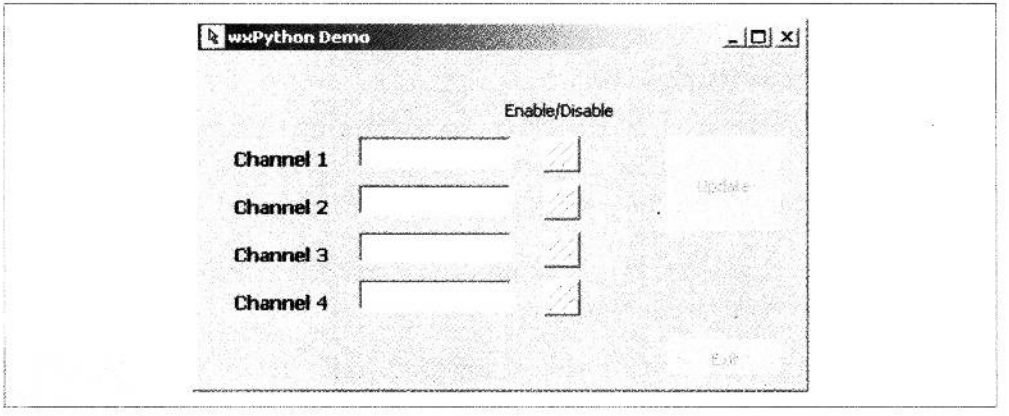


图13-13：排版成果

要创建这样一个框架，我们首先调整背景颜色，然后放置各种控件。然后，再给 6 个按钮选择恰当的名字，并为其生成事件处理程序框架。你可能会对那 4 个使用了对角线位

图（bitmap）的按钮感到好奇，其实它们可以动态地使用一张图片作为背景，接下来我就要实现这些功能。当我单击编辑窗口顶部的蓝色对号时，Boa会将设计转换为如下代码：

```
#Boa:Frame:Frame1

import wx
import wx.lib.buttons

def create(parent):
    return Frame1(parent)

[wxID_FRAME1, wxID_FRAME1BITMAPBUTTON1, wxID_FRAME1BITMAPBUTTON2,
 wxID_FRAME1BITMAPBUTTON3, wxID_FRAME1BITMAPBUTTON4, wxID_FRAME1EXITBUTTON,
 wxID_FRAME1STATICTEXT1, wxID_FRAME1STATICTEXT2, wxID_FRAME1STATICTEXT3,
 wxID_FRAME1STATICTEXT4, wxID_FRAME1STATICTEXT5, wxID_FRAME1TEXTCTRL1,
 wxID_FRAME1TEXTCTRL2, wxID_FRAME1TEXTCTRL3, wxID_FRAME1TEXTCTRL4,
 wxID_FRAME1UPDTBUTTON,
] = [wx.NewId() for _init_ctrls in range(16)]

class Frame1(wx.Frame):
    def _init_ctrls(self, prnt):
        # 生成方法，不要编辑
        wx.Frame.__init__(self, id=wxID_FRAME1, name='', parent=prnt,
              pos=wx.Point(331, 258), size=wx.Size(400, 250),
              style=wx.DEFAULT_FRAME_STYLE, title='wxPython Demo')
        self.SetClientSize(wx.Size(392, 223))
        self.SetBackgroundColour(wx.Colour(187, 187, 187))
        self.SetToolTipString('')

        self.staticText1 = wx.StaticText(id=wxID_FRAME1STATICTEXT1,
              label='Channel 1', name='staticText1', parent=self,
              pos=wx.Point(24, 64), size=wx.Size(61, 14), style=0)
        self.staticText1.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, False,
              'Tahoma'))

        self.staticText2 = wx.StaticText(id=wxID_FRAME1STATICTEXT2,
              label='Channel 2', name='staticText2', parent=self,
              pos=wx.Point(24, 96), size=wx.Size(61, 14), style=0)
        self.staticText2.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, False,
              'Tahoma'))

        self.staticText3 = wx.StaticText(id=wxID_FRAME1STATICTEXT3,
              label='Channel 3', name='staticText3', parent=self,
              pos=wx.Point(24, 128), size=wx.Size(61, 14), style=0)
        self.staticText3.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, False,
              'Tahoma'))

        self.staticText4 = wx.StaticText(id=wxID_FRAME1STATICTEXT4,
              label='Channel 4', name='staticText4', parent=self,
```

539

```
            pos=wx.Point(24, 160), size=wx.Size(61, 14), style=0)
        self.staticText4.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, False,
            'Tahoma'))

        self.textCtrl1 = wx.TextCtrl(id=wxID_FRAME1TEXTCTRL1, name='textCtrl1',
            parent=self, pos=wx.Point(104, 56), size=wx.Size(100, 21),
            style=0, value='')
        self.textCtrl1.SetEditable(False)
        self.textCtrl1.SetToolTipString('')

        self.textCtrl2 = wx.TextCtrl(id=wxID_FRAME1TEXTCTRL2, name='textCtrl2',
            parent=self, pos=wx.Point(104, 88), size=wx.Size(100, 21),
            style=0, value='')
        self.textCtrl2.SetEditable(False)
        self.textCtrl2.SetToolTipString('')

        self.textCtrl3 = wx.TextCtrl(id=wxID_FRAME1TEXTCTRL3, name='textCtrl3',
            parent=self, pos=wx.Point(104, 120), size=wx.Size(100, 21),
            style=0, value='')
        self.textCtrl3.SetEditable(False)
        self.textCtrl3.SetToolTipString('')

        self.textCtrl4 = wx.TextCtrl(id=wxID_FRAME1TEXTCTRL4, name='textCtrl4',
            parent=self, pos=wx.Point(104, 152), size=wx.Size(100, 21),
            style=0, value='')
        self.textCtrl4.SetEditable(False)
        self.textCtrl4.SetToolTipString('')

        self.bitmapButton1 = wx.BitmapButton(bitmap=wx.NullBitmap,
            id=wxID_FRAME1BITMAPBUTTON1, name='bitmapButton1', parent=self,
            pos=wx.Point(224, 56), size=wx.Size(24, 24),
            style=wx.BU_AUTODRAW)
        self.bitmapButton1.SetToolTipString('')
        self.bitmapButton1.Bind(wx.EVT_BUTTON, self.OnBitmapButton1Button,
            id=wxID_FRAME1BITMAPBUTTON1)

        self.bitmapButton2 = wx.BitmapButton(bitmap=wx.NullBitmap,
            id=wxID_FRAME1BITMAPBUTTON2, name='bitmapButton2', parent=self,
            pos=wx.Point(224, 88), size=wx.Size(24, 24),
            style=wx.BU_AUTODRAW)
        self.bitmapButton2.SetToolTipString('')
        self.bitmapButton2.Bind(wx.EVT_BUTTON, self.OnBitmapButton2Button,
            id=wxID_FRAME1BITMAPBUTTON2)

        self.bitmapButton3 = wx.BitmapButton(bitmap=wx.NullBitmap,
            id=wxID_FRAME1BITMAPBUTTON3, name='bitmapButton3', parent=self,
            pos=wx.Point(224, 120), size=wx.Size(24, 24),
            style=wx.BU_AUTODRAW)
        self.bitmapButton3.SetToolTipString('')
        self.bitmapButton3.Bind(wx.EVT_BUTTON, self.OnBitmapButton3Button,
```

```
            id=wxID_FRAME1BITMAPBUTTON3)

        self.bitmapButton4 = wx.BitmapButton(bitmap=wx.NullBitmap,
            id=wxID_FRAME1BITMAPBUTTON4, name='bitmapButton4', parent=self,
            pos=wx.Point(224, 152), size=wx.Size(24, 24),
            style=wx.BU_AUTODRAW)
        self.bitmapButton4.SetToolTipString('')
        self.bitmapButton4.Bind(wx.EVT_BUTTON, self.OnBitmapButton4Button,
            id=wxID_FRAME1BITMAPBUTTON4)

        self.updtButton = wx.lib.buttons.GenButton(id=wxID_FRAME1UPDTBUTTON,
            label='Update', name='updtButton', parent=self, pos=wx.Point(304,
            56), size=wx.Size(76, 65), style=0)
        self.updtButton.SetToolTipString('Fetch fresh data')
        self.updtButton.Bind(wx.EVT_BUTTON, self.OnUpdtButtonButton,
            id=wxID_FRAME1UPDTBUTTON)

        self.exitButton = wx.lib.buttons.GenButton(id=wxID_FRAME1EXITBUTTON,
            label='Exit', name='exitButton', parent=self, pos=wx.Point(304,
            192), size=wx.Size(76, 25), style=0)
        self.exitButton.Bind(wx.EVT_BUTTON, self.OnExitButtonButton,
            id=wxID_FRAME1EXITBUTTON)

        self.staticText5 = wx.StaticText(id=wxID_FRAME1STATICTEXT5,
            label='Enable/Disable', name='staticText5', parent=self,
            pos=wx.Point(200, 32), size=wx.Size(70, 13), style=0)

    def __init__(self, parent):
        self._init_ctrls(parent)

    def OnBitmapButton1Button(self, event):
        event.Skip()

    def OnBitmapButton2Button(self, event):
        event.Skip()

    def OnBitmapButton3Button(self, event):
        event.Skip()

    def OnBitmapButton4Button(self, event):
        event.Skip()

    def OnUpdtButtonButton(self, event):
        event.Skip()

    def OnExitButtonButton(self, event):
        event.Skip()
```

541

尽管只是一个框架，但其中还是包含了很多代码。所幸，我们不必担心 __init__() 以上

部分的代码，只需关心这 6 个按钮的事件处理程序就可以了。

因为 Exit 按钮还没有做任何事情，我们需要加入下面的代码让它能够真正实现退出：

```
def OnExitButtonButton(self, event):
    self.Destroy()
```

如果你的应用需要在关闭前做一些内部处理，可以放到这里来做。在我们的代码里只需在程度代码底部调用 Destroy() 方法，其他的事情就丢给底层来做了。

接下来是 4 个 Enable/Disable 按钮。我打算用较小的位图资源来表示开 / 关状态。这些图像通过事件处理程序进行替换。

initButtons() 这个新方法会被用来初始化 4 个跟踪变量的状态，并为每个按钮分配一个红色的位图。

```
def initButtons(self):
    self.cnt = 1

    self.btn1State = Flase
    self.btn2State = Flase
    self.btn3State = Flase
    self.btn4State = Flase

    self.bitmapButton1.SetBitmapLabel(bitmap=
        wx.Bitmap(u'red24off.bmp',wx.BITMAP_TYPE_BMP))
    self.bitmapButton2.SetBitmapLabel(bitmap=
        wx.Bitmap(u'red24off.bmp',wx.BITMAP_TYPE_BMP))
    self.bitmapButton3.SetBitmapLabel(bitmap=
        wx.Bitmap(u'red24off.bmp',wx.BITMAP_TYPE_BMP))
    self.bitmapButton4.SetBitmapLabel(bitmap=
        wx.Bitmap(u'red24off.bmp',wx.BITMAP_TYPE_BMP))
```

542

变量 self.cnt 是供 Update 按钮使用的。在 __init__() 中，initButtons() 会在控件初始化完成后被调用：

```
def __init__(self,parent):
    self._init_ctrls(parent)
    self.initButtons()
```

下面是第一个按钮的事件处理程序：

```
def OnBitmapButton1Button(self, event):
    if self.btn1State == False:
        self.btn1State = True
        self.bitmapButton1.SetBitmapLabel(bitmap=
                wx.Bitmap(u'green24on.bmp', wx.BITMAP_TYPE_BMP))
    else:
```

```
            self.btn1State = False
            self.bitmapButton1.SetBitmapLabel(bitmap=
                    wx.Bitmap(u'red24off.bmp', wx.BITMAP_TYPE_BMP))
```

其他三个按钮与之等价，只是改了个名字。最后，当 Update 按钮被单击时，其事件处理程序每次都会将一个字符串推送到每个文本控件中。

```
def OnUpdtButtonButton(self, event):
    self.textCtrl1.SetValue(str(self.cnt))
    self.textCtrl2.SetValue(str(self.cnt))
    self.textCtrl3.SetValue(str(self.cnt))
    self.textCtrl4.SetValue(str(self.cnt))
    self.cnt += 1
```

要让这个程序运行，我们只需再往文件最后添加下面一小段代码：

```
if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = create(None)
    frame.Show(True)
    app.MainLoop()
```

这里使用了 wxPython 的 `wx.PySimpleApp()` 类创建一个 app 对象来提供一个主循环。wxexample.py 的运行结果如图 13-14 所示。

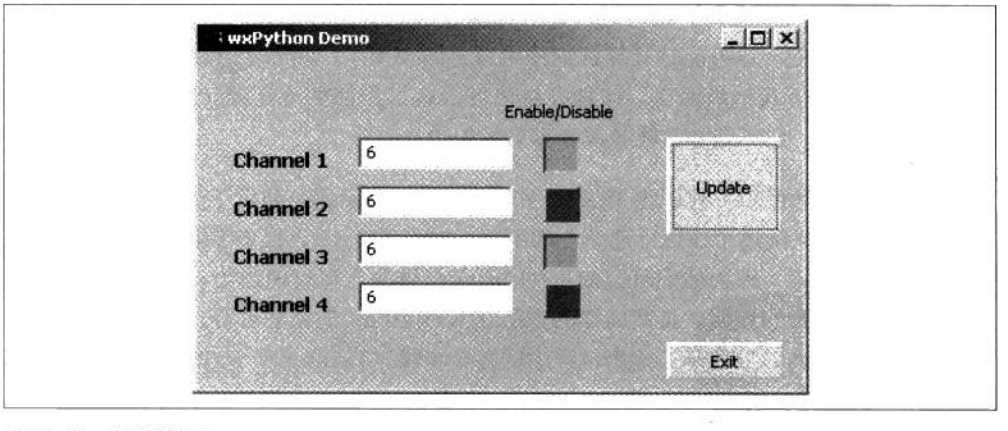


图13-14：运行结果

创建 wxexample.py 的过程中还有诸多细节我没有提到。如果你想了解更多关于 wxPython 或者 Boa 构造器的内容，我建议你自己动手再实现一次这个例子。如果你想看到这些控件的设置，只需加载源代码到 Boa 中，然后查看变量设置面板。

wxPython 的工具和资源

我们已经看到如何使用 Boa 构造器这样的工具来创建一个 wxPython 应用，除此之外，还有其他一些有用的工具。

543 *wxGlade (http://wxglade.sourceforge.net)*

wxGlade 不像 Boa 那样用于创建完整的应用，它仿照 GTK+/GNOME 环境下的 GUI 设计工具 Glade，可以用来构建独立的窗口和对话框，就像 TkInter 的 PAGE 一样。

PythonCard (http://pythoncard.sourceforge.net)

作为一种创建简单 GUI 应用的工具，PythonCard 使用了模板和一些基础控件。它针对新手来说是个很好的开始。不过，其源代码自 2007 年以后就再也没有更新过，而 Python 和 wxPython 在这段时间内却有很大的发展。其中的一些特性可能已经无效了。

在网上有很多可用的信息，本章最后的“推荐阅读”一节罗列了一些重要的资源。

小结

本章我们分别介绍了文本界面和图形界面。我们已经看到，虽然文本界面对很多应用来说表现不错，但很多时候应用需要一个逼真的界面或界面接口，单纯的字符无法支持到这一级别。

有一点你可能已经注意到，就是这两种界面编程都需要某种形式的主循环来处理输入、输出和刷新的功能。这可能是计算机编程中最早的模型了，相对于尝试扫描屏幕来控制功能，然后再分析其结果而言，这种办法使得很多工作都更为直观（我见过那种程序，你应该已经猜到，它不是非常健壮或者容易维护的）。

544 我分别通过手动和使用工具创建 TkInter 和 wxPython 的示例，演示了这两种方法，用一种相对轻松的方式阐述了在设计理念上的不同。对于 TkInter，很多专业的开发人员不会被一个工具所困扰，因为他们可以使用 TkInter 自带的几何管理能力，快速地创建功能强大而且外观优雅的界面。虽然同样的方法在 wxPython 上完全可行，但我还是选择使用 Boa 构造器工具，并展示了这个过程中的各个步骤。TkInter 和 wxPython（甚至其他 GUI 库）中的大多数 GUI 工具的工作方式是相似的，所以这个例子对于其他工具或者类似情况应该也会有所帮助。

我希望你在此所了解到的东西能为你自己的应用带来一些新想法。正如我在本章多次所提到的，如果打算使用类似 curses 或者 GUI 去完成一些重要的工作，你应该自己去查阅相关书籍和后面“推荐阅读”一节中的链接。

推荐阅读

下面是我使用 curses、wxPython 和 TkInter GUI 工具的过程中所发现的一些有用的书，还有一些和本章相关的有趣的资源链接。

Programmer's Guide to ncurses. Daan Gookin, John Wiley & Sons, 2007.

Dan Gookin 的书涵盖了 ncurses 库 5.5 版本，而且涉及 UNIX、Linux 和 Mac OSX。尽管它只讲了 C API，但有助于搞清楚 Python 的 curses 库能做什么，以及为什么这么做，而且其中很多例子可以很容易地转换成 Python 代码。该书的索引部分也非常有用，为查找那些复杂或是被遗忘的函式提供了一个快捷的方法。

wxPython in Action. Noel Rappin and Roibin Dunn, Manning Publications, 2006.

此书涵盖了 Python API 的 wxWidgets 库。其中包括很多例子，因为涉及领域太多，所以一些主题比其他同类图书更深入一些。要想进一步挖掘相关内容，只能查阅 wxPython 源代码，或是 wxWidgets 的 C++ 源代码了。在任何情况下，如果想参与 wxPython 相关的开发，手边就应该有这本书。

Cross-Platform GUI Programming with WxWidgets. Julian Smart, Kevin Hock, and Stefan Csomer, Pearson Education/Prentice Hall, 2005.

书名已经表明这本书是为 wxWidgets 准备的。这本书很厚，包含很多细节，而且当 wxPython 不容易理解时，你可以转向这里找到答案。要强调的是，你需要能够读懂最基本的 C++，才能从此书中获取更多东西。

Python and Tkinter. John E. Crayson, Manning Publications, 2000.

545

这本书详细地讲述了 TkInter GUI 工具，书中还有大量的示例和一些可用控件的说明。为了简洁，书中忽略了一些细节，而且关于 Python Megawidgets（pmw）附加库的部分也已经过时了（考虑到此书已有十多年的历史，这也不足为奇了）。尤其是 pmw 使用的 BLT 模块，看上去它已经非常过时了，我看到的最后一次发行和 Python 2.6 是同一时期。但如果你只想使用 TkInter，而且不打算使用 BLT 中的功能，那就没问题了。

http://arstechnica.com/old/content/2005/05/gui.ars/4

Jerremy Reimer 的文章“A History of the GUI”，该文带领读者回顾了 GUI 从始至今这一路上那些重要的里程碑。虽然没什么技术细节，但它却很好地强调了我们每天都与之交互的计算机系统中的那些基本原理和耳熟能详的概念。

http://articles.sitepoint.com/print/real-history-gui

Mike Tuck 写的一篇很有趣的文章，“The Real History of the GUI”。这篇文章讲述了一些我们今天所熟悉的 GUI 框架的主要开发过程，同时还给出了一些有意思的相关链接。

http://invisible-island.net/xterm/ctlseqs/ctlseqs.html

“Xterm Control Sequences”是一个 Xterm 所能识别的命令列表。最初是由美国加州

大学伯克利分校的 Edward May 编辑的，内容简明扼要。如果你打算在 Linux 环境下使用 Xterm 窗口来工作，你应该把这份文档放在手边。

http://www.ecma-international.org/publications/standards/Ecma-048.htm

ECMA-48 规格说明书可以从 ECMA 国际的这个 URL 下载到 PDF 文档。它的定位不仅仅是 ANSI 类型的控制序列，还包括 ASCII 字符集。100 余页内容读起来并不轻松，但其中包含不少专业细节和对其他标准的引用，绝对值得下载。

http://www.catb.org/~esr/writings/taouu/taouu.html

Eric Raymond 的 *The Art of Unix Usability* 全文的在线内容都在这个 URL 中。不幸的是，我还不能在本地下载一个 PDF 版本，我更喜欢纸质阅读而不是一直盯着屏幕。Raymond 先生提供了一些在不同的公司和操作系统之间偶尔穿插的一些深刻见解，并从早期的打孔设备开始，探索用户界面背后的历史和原因。“Rules of Usabilit”（可用性规则）一节中包含了一些宝贵的经验，它们常在用户界面的设计中被忽视。本章开头所引用的话就是其中之一，尽管我不知道这是否是 Raymond 先生的首创。

http://www.joelonsoftware.com/articles/Biculturalism.html

Joel Spolsky 在其博客 *Joel on Software* 上针对 Eric Raymond 的 *The Art of UNIX Programming*（*The Art of Unix Usability* 的姐妹篇）提供了一篇智慧而富有洞察力的评论。虽然是软件开发方面的著作，但评论中提出了一些关于 UNIX 和 Windows 之间文化差异的有趣见解。虽然 Python 已经发展成一门跨平台的、独立的语言，而且在用户体验上消除了平台之间的很多差异，但仍然值得花些时间来理解为何 UNIX（还有 Linux）如同航天飞机的驾驶舱，而 Windows 却像舒适的四开门轿车，带有电动门窗，自动换挡，而且音响上的旋钮和按钮比仪表盘上的还多。即便你不打算一生都从事软件开发工作，你也可以通过这篇文章理解为什么当代系统及其用户界面会是现在这个样子。

546

http://www.pythonware.com/library/tkinter

Fredrik Lundh 的这个包含索引的页面文档对 TkInter 做了充分的介绍。虽然有可用的 PDF 文档，不过是看起来比较别扭的双联印刷版式。很不幸，这导致它基本上无法被打印和装订。但除了格式问题外，其中的例子选得很好，而且容易理解。如果你拥有此文档，以及下面列出的 NMT 的引用和 Python 官方文档，那么即便没有更多其他教材，你应该也可以比较容易地创建一些可用的 TkInter GUI 了。

http://infohost.nmt.edu/tcc/help/pubs/tkinter/

来自新墨西哥理工大学的在线文档“Tkinter 8.4 Reference: A GUI for Python”。这个链接还提供了可用的 PDF 格式文档。尽管不是非常详尽，但这个重要的参考资料（120 页的 PDF 文档）中涵盖了如布局方法、属性以及各种通用的 Tk 控件等多个主题，其中包括 Tk 特有的画布控件。

第 14 章 547

实例

小心带着螺丝刀的程序员。

——Leonard Brandwein

我们希望本章能够成为一个对本书之前展示内容的总结，所以不会包含一些深入的讨论。与其写成一个手把手的分析和指南，我们更想让它成为你日后解决问题的启发。对于本章例子中所涉及的问题，应该都可以从本书之前的内容里找到答案。如果你需要更多的细节，可以参考各章或附录中提供的出处和链接。

此刻你可能已经想到，仪器接口大致可以归为两类：一种需要给计算机加入某种附加硬件，另一种只需要某种电缆。本章我们会通过细致地分析那些只需电缆的例子作为我们学习之旅的收尾。从目前知道的来看，我们会看到如何在现实生活中利用 RS-232 串口和 USB 接口采集数据和控制设备。

我们将从一个 DMM 数据捕捉软件开始，该 DMM 具有串行输出功能（和我们在第 11 章中所略有涉及的是同一块仪表），然后会关注其他一些使用更为传统的命令 - 应答协议的串口 I/O 设备。接下来是一个 USB 数据采集和控制设备，即 LabJack U3。由于其复杂性，也由于它是此类设备的典范，我们会多花一些时间对 U3 进行讲解。

串行接口

一提到串行接口，很多人会想到 RS-232 或者 RS-485。但是在现实中，串行接口可以是多种多样的。比如，我们使用一个设备将 USB 转换为 GPIB，又通过虚拟串口读回数据，尽管 GPIB 是一个并行的数据总线，我们实际上也是使用串行接口。适配器及其配套的 548
USB 驱动所做的工作被称为数据流的“串行化”，所以从软件的角度控制和监控仪器，这是一个传统的普通串口。

起码在数据和命令的发送和接收上，基于 RS-232 的串行接口在先天上更加容易使用。另外一点就是，几乎所有使用串行接口的仪器和设备都使用命令 - 应答协议。不用命令的情况下发送数据的仪器是很少见的，但是它们的确存在。我们接下来讲到的 tpi 183 数字万用表就是一个例外。

简易 DMM 数据获取

Tpi 183 是一款便宜的数字万用表型号（很多批发商的报价都在 130 美金左右），它拥有多种功能和不错的精确度。同时还提供了一个带 3.5mm 插孔的串行端口，且能以 1200 波特的速度持续输出 ASCII 数据流。只要把仪表附带的可选的接口线连接到 PC 上，你就能捕获仪表的数据并显示出来，或将其存入日志文件。图 14-1 展示了 183，RS-232 接口位于其外壳的右侧。

图14-1：Tpi 183数字万用表

549

183A 是 183 的一个版本，但使用了不同的接口。这里的讨论只针对原版的 183，而不是 183A。

你可以扭转旋钮打开万用表同时按住“COMP”按钮来启动 RS-232 输出。在串行输出被激活的情况下，仪表的自动关机功能会被禁用，这样会很快耗尽电池的电量。但由于不能接外部电源，如果想让它长时间地运行，此方法就不是一个好的选择了。

183 的串行接口工具包里有软件示例和特制的 RS-232 接口线。这个接口是光电隔离的，因此连接线通过从 PC 串口的一条 RS-232 控制线中取电给仪表内部的光电隔离器提供直流电源。由于 183 数据线里有些额外的内部部件，所以一般的 RS-232 线不能用于 183 仪表。如果你的笔记本电脑或上网本没有真正的 RS-232 串行接口，可以使用 RS-232-to-USB 适配器来通过一个虚拟的串行接口进行传输，只要该 USB 适配器能够提供必要的 RS-232 控制线电压即可。

183 会根据图 14-2 中显示的格式将仪表功能、量程模式、量程及数据值编码成 ASCII 字符串。

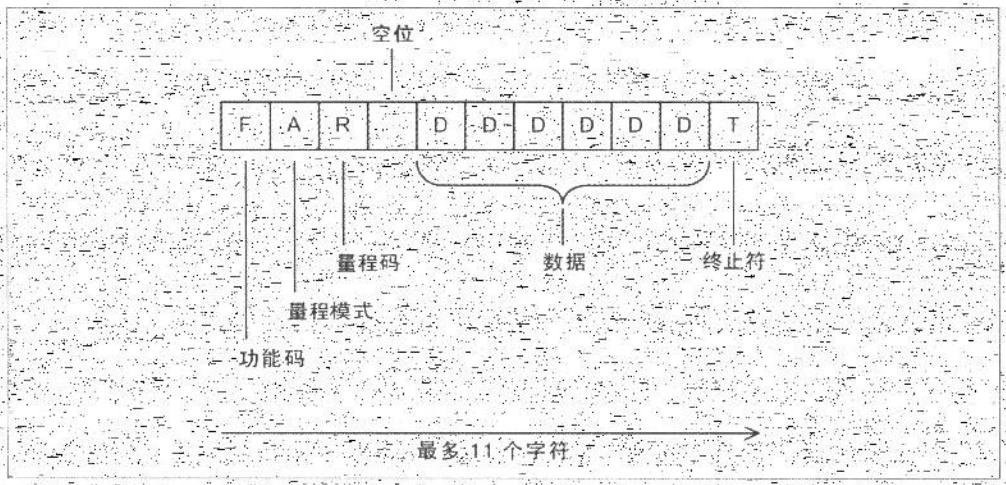


图14-2：tpi 183的数据输出格式

有 4 种不同的数据项或字段，还有，功能码字段实际上是十六进制的。功能码有 11 种可能值和一个不使用的编码，共计 12 个值，从而形成从 0 到 B 的序列。

表 14-1 显示了各个功能码的含义，表 14-2 展示了在量程码中使用的编码。 550

表14-1：183 DMM功能码

编码	功能
0	AC 伏
1	欧姆
2	DC 伏
3	DC 毫伏
4	AC 安培
5	DC 安培

续表

编码	功能
6	二极管测量
7	DC 毫安
8	未分配
9	AC 毫安
A	电容
B	频率

表14-2：183 DMM范围码

	量程					
功能码	0	1	2	3	4	5
0,2	4.00V	40.00V	400.0V	1,000V		
7,9	40.00MA	400.0MA				
4,5	4.000A	10.00A				
1	400.0	4.000k	40.00k	400.0k	4.000M	40.00M
B	200.00Hz	2.0000kHz	20.00kHz	200.00kHz		
A	400.0nF	4.000μF	40.00μF			
3	400 mV					
6	4.000V					

由于我们知道会得到什么，下面的 Python 代码片断展示了获取 4 个字段的数据是多么容易（instr 是来自 DMM 的数据）。

```
fcode = instr[0]
mcode = instr[1]
rcode = instr[2]
data = instr[4:len(instr)-1]
```

551 前三个字段的长度始终都是一个字符。最后一个字段指所有位于空格（instr[3]）和字符串末尾减 1 之间的部分（我们不关心终止符）。字符串本身必须长过 9 个字符。只要我们得到的仪表字符串小于或等于 9 个字符，它都应被丢弃。

要编写一个从该仪表捕捉数据的软件，我们有多种方法可供选择，而立刻会想到的方法有两种。一是在循环中等待仪表，并在读取数据的间隔处理诸如写入日志文件或更新显示这样的事务。另一个方法是利用一个线程获取输入数据，并将其推入队列，一个主循环稍后从中读取数据，并写入到某类显示上（或存入文件）。第一种方法实现起来更直接了当。第二种则更灵活，同时实现起来也会稍微困难一些。

一定要注意，仪表输出的字符串不包含 EOL 字符（文件结束符）。当一个字符串结束，

另一个会立刻被传输。在这种情况下获取数据最安全的办法是使用 *pySerial* 的 read() 方法，并在一次获取一个字符的同时寻找终止符。这样，我前面提到的字符串长度限制就有用武之地了：如果该捕捉软件正好从字符串中间位置开始读取字符，它仍然能够识别终止符并试图从不完整的字符串中提取数据。当你的软件开始监听时，是无法知道仪表的输出会是什么样子的，因此它需要和仪表同步。确保至少 10 个字符被读取和终止符出现将极大地降低得到无效数据的概率。

read183dmm.py 里展示了一段从 183 DMM 读取数据流的简单脚本代码。

```
#!/usr/bin/python
# 183 DMM data capture example
#
# 简单演示从 DMM 获得数据
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import serial

sport = serial.Serial()
sport.baudrate = 1200
sport.port = "com17"
sport.setTimeout(2) # give up after 5 seconds
sport.open()

instr = ""
fetch_data = True
short_count = 0
timeout_cnt = 0
maxtries = 5    # timeout = read timeout * maxtries

while fetch_data:
  getstr = True
  # 输入字符串读循环
  # 每次从 DMM 获取一个字符以构建成输入字符串

  while getstr:
      inchar = sport.read(1)
      if inchar == "":
          # 什么也读不到表示出现超时
          print "%s timeout, count: %d" % (inchar, timeout_cnt)
          timeout_cnt += 1
          if timeout_cnt > maxtries:
             getstr = False
      else:
          # 检测终端字符是否读入，如果已读入
          # 则认为该输入字符串已经结束
```

552

```
            if inchar == '&':
                getstr = False
            instr += inchar
            timeout_cnt = 0      # 重置超时计数器

    # 如果出现超时即停止
    if timeout_cnt == 0:
        if len(instr) > 9:
            # 如果字符串可用，就放入数据中
            # pull out the data
            fcode = instr[0]
            mcode = instr[1]
            rcode = instr[2]
            data = instr[4:len(instr)-1]

            # 在此进行实际的显示或记入日志，此处的 print 语句只是起占位作用。
            print "%1s %1s %1s ->  %s" % (fcode, mcode, rcode, data)

            # 重置非完整读取计数器
            short_count = 0
        else:
            # 如果重复得到连续的短字符串，就表明有问题
            short_count += 1
            # 如果接连出现 5 次，终止循环，退出脚本
            if short_count > 5:
                fetch_data = False
        # 不论发生什么情况，都清空输入字符串
        instr = ""
    else:
        # 如果在输入字符串处理过程中超时，就结束主循环
        fetch_data = False

print "Data acquistion terminated"
```

read183dmm.py 没有提供用户输入。它会在连续 5 次读取超时之后终止，因此要退出的
553 话，你只要关掉 DMM 或将接口线从插口拔掉即可。它也会在连续接收 5 次短字符串 (len(instr) < 10,) 之后终止，这意味着存在通信问题。

你很可能想要将字段提取语句后面的 print 替换成更有用的东西，比如将数据存入文件或获取用户输入命令的能力。对于用户输入，我们在第 13 章看到的线程技术在此处会很管用。同样，如我们在第 13 章中所见，要扩展 read183dmm.py 使其具备美观的显示功能将是一件很容易的事。只要你真的想做，为该程序建立图形化用户界面也并非什么难事，尽管如此，对于本程序，我还是建议采用 ANSI 来做显示。

串行接口的离散或模拟数据 I/O 设备

如果使用命令 - 应答协议，为串口设备编写数据捕捉和控制应用程序常常是很简单的。前面的 tpi 183 DMM 就是一个例子，尽管它有点名不符实（它只发送数据，不接收命令）。Omega Engineering 的 D1000 系列数据变送器是另一个更为普遍的例子。这些设备是非常典型的数据采集模块，它们利用普通的串行接口从主控制器 PC 接收命令并返回应答。我之所以在本节选择它们，并非因为我喜欢它们胜过其他同类设备，而是因为它们拥有一个丰富的命令集并且以很小的体积集成了大量的功能。它们在现有的串行数据采集模块中并非是最便宜的产品，但它们和大多数 Omega 产品一样都很坚固易用，至今我还没有遇到过失灵的情况。

图 14-3 展示了 Omega D1000 数字数据变送器系列中的一员，D1112。D1000 家族中具有模拟输入功能的成员能从某种传感器返回数据值，比较典型的情况是温度传感器。它们还具有创建报警点（alarm set-point）和数据范围缩放的能力，有些型号还具有扩展离散数字 I/O 的功能。

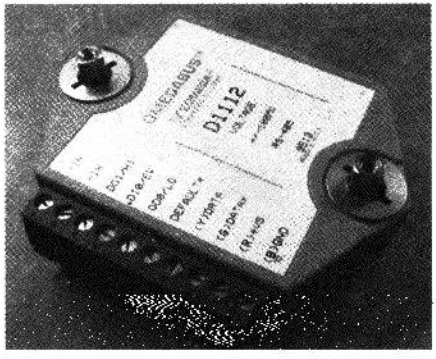


图14-3：Omega D1112 RS-485数据变送器

D1000 在其命令协议中使用了一种设备寻址方案，命令都会以一个设备的 ID 号作为前缀，如下：

```
$1RD
```

设备会以下面这样的内容作为应答（具体内容取决于输入和缩放比例）：

```
\*+00097.40
```

设备 ID 前缀允许多个 D1000 装置以菊花链的形式共享单个 RS-232 或 RS-485 接口。这和第 11 章中介绍的 RS-485 电动机控制器方案有相似之处。图 14-4 展示了如何使用多个 D1000 型设备来对热处理室中的温度进行监控。

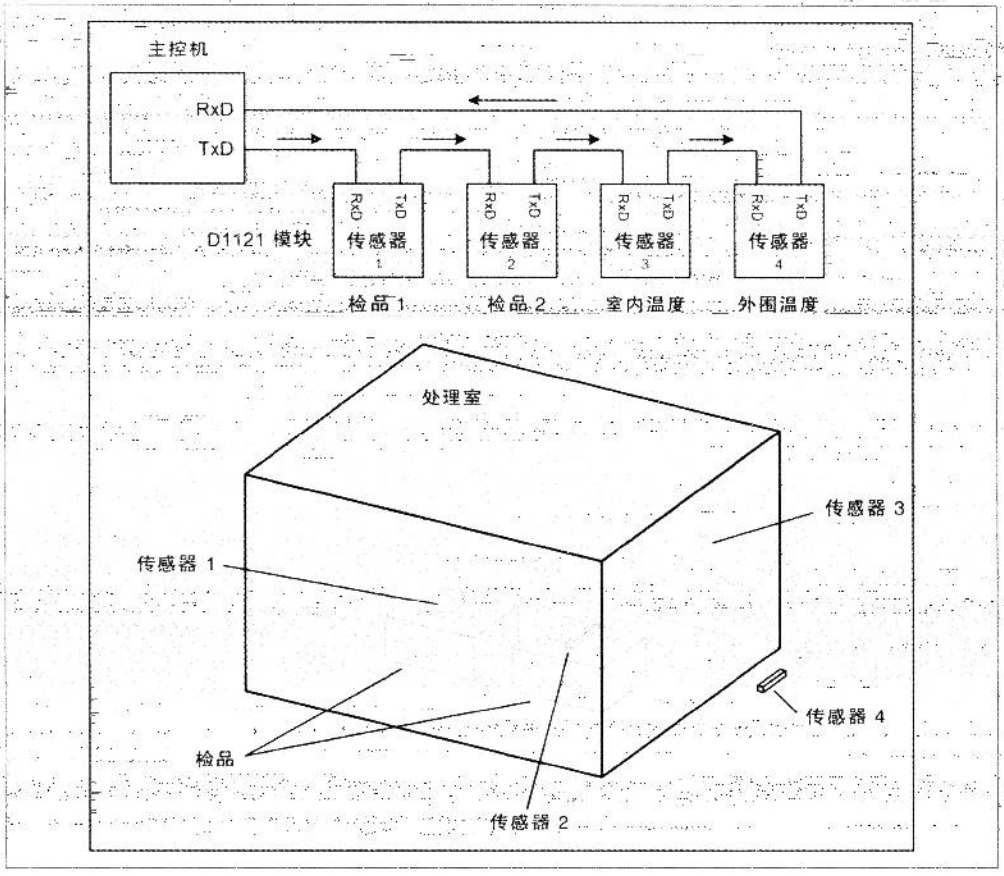


图14-4：D1121 热处理室应用

554 图 14-4 中的装置使用了 4 个传感器：两个用于处理室中的检品以防止热损坏，一个用于监测处理室温度，一个用于监测外部环境温度。传感器的类型可以是热电偶、热敏电阻或 RTD 型传感器，只要对 1121 的输入不超过 1V 就行（其他型号对应不同的输入电压范围）。标为传感器 1 和传感器 2 的装置不用来作为调节处理室温度的控制输入，而是用来作为当检品超出某预设温度限制时的关机输入。传感器 3 是温度控制器的主输入。控制器可以使用继电器式控制，也可以使用 PID 算法，这要根据所需的温度调节严格度以及处理室使用的加热 / 冷却系统类型来决定。传感器 4 只是一个外围温度传感器。尽管它在处理室的运转中并不起什么作用，但从该传感器收集的数据可以被用来判断在不同外围温度情况下处理室能多好地维持特定温度。换句话说，如果处理室隔热不好且处于加热状态，当外围温度较低时加热单元将有更大的工作负荷，因为它需要尽力克服处理室的热损失。通过将外围传感器的数据与内部温度以及加热单元的工作情况进行比较，我们就可以估算出处理室的隔热效率。

D1000 模块体积虽小，但它们集成了相当数量的功能。该设备可以被命令以两种不同的 555
字符串格式来进行应答，Omega 称之为长形（long-form）或短形（short-form）。还可为每个模块指派一个标识字符串。一个模块可以是“HEATER”，另一个也许是“COOLER”，等等。

所有与标准 D1000 变送器的通信都是以命令 - 应答形式进行的。和 tpi 183 不一样，标准 556
D1000 设备从不自己发起一次会话（尽管有可能在订购时指明需要有这样的功能）。只要 D1000 还未向主控系统发回应答，则一次通信处理就被视为未完成。

图 14-5 显示了一个 D1000 命令格式。命令提示字符对 D1000 发出一个命令即将抵达的信号。D1000 模块能识别 4 种不同的提示字符，如表 14-3 所示。美元符号（$）是最常用的提示字符，它让 D1000 返回一个短形应答。井号（#）让 D1000 产生长形应答。[和] 提示被用来和所谓的扩展寻址模式一起用，这里不进行深入介绍。

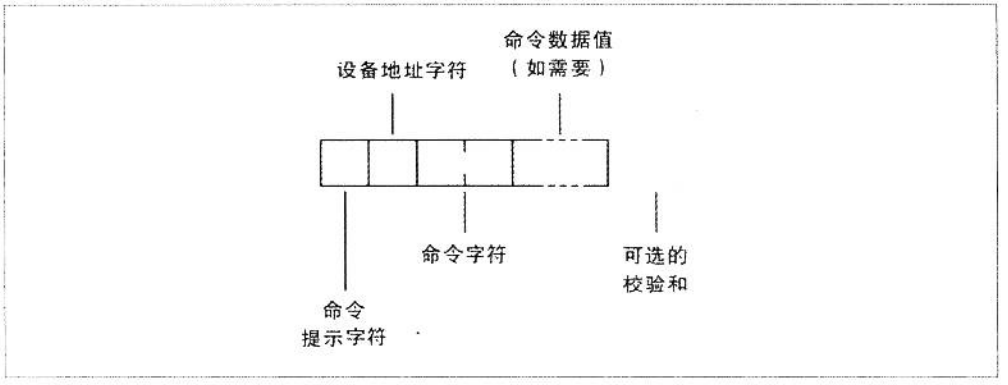


图14-5：D1000命令格式

表14-3：D1000命令提示字符

提示字符	定义
$	设备返回短形应答
#	设备返回长形应答
[	用于扩展寻址模式
]	用于扩展寻址模式

在提示字符后接下来是一个单一的设备地址字符。D1000 设备的地址由用户指派。任何可打印 ASCII 字符都能用做地址字符，但为了便于管理，最好把选择范围限制在 0~9 或 0~F 之间。使用不易从数字排列转换的地址字符会让使用循环去遍历一组菊花链式的设备变得难以操作。

地址字符之后则是两个或三个设备命令字符，接着是命令所需要的任何参数数据。有些
557 命令，如查询命令，不需要任何额外数据。最后，命令可使用可选的两字符检验和，具体见 D1000 用户手册。表 14-4 是一个 D1000 命令集摘要。更多的细节，包括对每个命令及其应答的详述，请参见用户手册。

表14-4：D1000命令组

助记符	定义	短形返回
DI	读取报警或数字输入	*<data>
DO	设置数字输出	*
ND	新数据	*<float value>
RD	读取数据	*<float value>
RE	读取事件计数	*<event counter value>
REA	读取扩展地址	*<ext address>
RH	读取高报警值	*<high alarm value><mode char>
RID	读取标识符	*<ID string>
RL	读取低报警值	*<low alarm value><mode char>
RPT	读取脉冲过渡	*<transition chars>
RS	读取设置	*<setup values>
RZ	读零	*<float value>
WE	写使能	*
CA	清除报警	*
CE	清除事件	*
CZ	清零	*
DA	解除报警	
EA	启动报警	*
EC	事件读取 & 清除	*<event counter value>
HI	设置高报警上限	*
ID	设置模块标识	*
LO	设置低报警下限	*
PT	脉冲过渡	*
RR	远程复位	*
SU	设置模块	*
SP	设置设定值	*
TS	满度修整	*
TZ	零点修整	*
WEA	写扩展地址	*

558 D1000 对命令的响应从一个前缀符号开始，跟着是数据、错误消息，或者什么也没有。有两种应答字符串的前缀：星号（*）和问号（?）。响应情景的第三种类型是当 D1000

什么都不返回并出现超时时。星号指示返回无错误，问号则指示出错。出现错误时，问号后紧跟模块 ID，然后是错误信息。这里是一个带额外字符的命令有可能出现的错误应答。

```
$1RDX
?1 SYNTAX ERROR
```

检查响应包含查看应答字符串的第一个字符。如果是星号，那一切正常；否则就有错误，而且一定要进行处理。模拟数据是以 9 个字符组成的字符串返回，其中包括 1 个符号字符，5 个数字，1 个小数点和 2 位小数。例如，RD 命令会导致设备有如下应答：

```
*+00220.40
```

很方便的是，Python 会照原样整个接收字符串中的数据值部分，然后无须任何额外步骤就能将其转换成浮点值。

一些命令不会返回短形模式的数据字符串，但所有无错信息都会至少返回一个星号。每种命令的应答对该命令都是独特的（没有一种普适的命令应答），所以软件需要知道能从设备那里得到什么样的应答。

当与像 D1000 这样的设备打交道，尤其是如果你发觉自己经常使用它们时，也许值得为此创建一个包含一组能处理各种功能的类。这比在代码中到处零散地敲打命令要清楚和简便许多，这也使得可以在一个地方做修改（类的定义），同时在所有使用了类的地方都生效。

Omega D1000 系列设备只是现有针对带有串行接口的数据 I/O 产品种类中的一个例子。针对热电偶输入、RTD 传感器、离散数字 I/O，以及模拟与数字混合 I/O 的设备有很多。在 Google 中输入“digital I/O RS-232”或“digital I/O RS-485”会搜索到针对工业应用的很多厂商，从卖配件到高端设备，无所不包。我建议当选购串行接口 I/O 设备时，你务必确认你订购的不是工业上那种 DIN 导轨安装设备。虽然这些设备没有什么问题，但其物理安装要求会证实这比配有简单安装孔的东西会导致更多的麻烦，而且它们更贵。

串行接口及对速度的考虑

559

基于 RS-232 的串行接口速度很慢，因为 RS-232 的上限速度约在 20kb/s，而且此值会随着接口线的长度增加而下降。RS-485 会快很多，最大速度可达 35Mb/s，而且用 USB 2.0 的理论最大数据传输速度可达 480Mb/s。但接口速度并不是故事的全部，控制接口的软件能多快地传输和接收数据也是一个问题。

对本书中描述的一些种类的应用来说，10ms（100Hz）的周期性采集时间就足以认为是

较快了，而 100 Hz 也正好在运行于现代 PC 上的 Python 的能力范围之内。但是，20Hz 的更新率对于很多普通数据采集和控制应用来说常常已经绰绰有余。同时请记住，随着采集速度的增加，硬件成本和软件复杂度也会相应地增加。

这里的基本意思是说，对预定的应用只要速度足够快，则设备使用低速的 RS-232 串口来采集数据或进行控制就绝对没什么问题。下面我们介绍如何计算一个串行接口是否足够快。

假设我们有一个 9600 波特率（baud）RS-232 接口的设备。每个命令长度是 8 字节，最长应答是 10 字节。在命令与应答之间，在采集设备上有一个最大为 1ms 的延迟（该延迟一部分是由设备内部的 ADC 转换时间，一部分则是由采集事件之间的命令或数据处理所需的时间决定的）。

当速度为 9600 波特率时，从主机发送命令到设备大约需要 8.3ms，等待应答返回大约需要 10.4ms，Python 程序处理应答数据也需要一些时间。如果把所有的时间相加，则会得到一次完整的命令 - 转换 - 应答 - 处理事务周期，大概会有 21ms。这就得出约为 48Hz 的结论。从这里我们可以看出，串口的传输时间决定了设备的采集速率，使用低速串口的设备都有这样的共性。随着接口速度的提升，命令 - 应答事务所需的时间就不是那么限制性的因素了。

尽管 48Hz 对于热控、环境监测或恒星测光之类的应用来说通常已经绰绰有余，但这个速度却不足以对像脉冲或其他出现在数据采集事件之间的瞬变现象这类的事件进行采集。由于控制系统的总循环时间大部分都用在与外部设备或系统通信上，因此可以通过增加波特率或转用 RS-485 这两种方式来加快速度并缩短总循环时间。

然而，还有另一种方法，那就是 USB。在下一节我们将看到如何使用 USB 数据采集和控制设备，尽管软件的复杂度会稍微上升，但这将带来更高的速度和多功能操作。

560

USB 实例：LabJack U3

USB 设备的技术提升持续驱动着带有 USB 接口的 I/O 设备的成本下降。带有这类新式 I/O 模块的设备在五年以前还需要花费上百上千美元，如今的价格在 100 美元左右，这取决于速度、转换分辨率、I/O 通道的类型和数量。

其中一个低成本设备就是图 14-6 中显示的 LabJack U3。该数据采集设备提供了离散数字 I/O、模拟 I/O 和计数 / 定时 I/O。它所有的命令和数据应答事务都通过一个 USB 接口来完成。

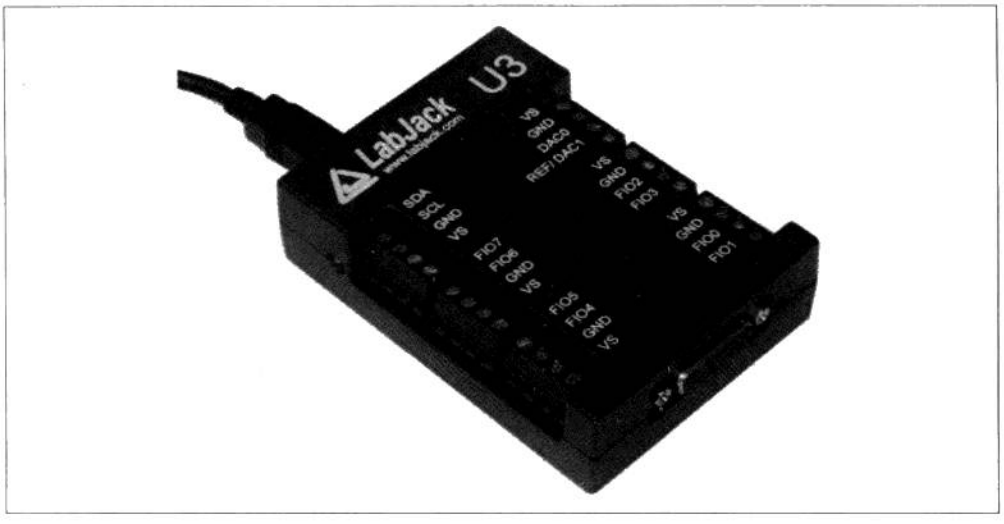


图14-6：LabJack U3 USB DAQ设备

U3 由 USB 接口供电，没有提供这类设备普遍具备的外部供电。因此，即使在一些端子位置上有 +5V 的直流电，也应该注意不要取过多的电流而导致 USB 连接中断。

LabJack 连接

图 14-7 展示了一张 U3 连接图。U3 上很多接线端子可以被配置成不同的工作模式，而且配置参数可以保存，以确保设备上电时处于正确的状态。除了端子排连接，U3 还有一个引出了 12 条额外信号线的 DB-15 连接器。

你可以通过 USB 集线器一次将多个 LabJack 连接到 PC，但是注意数据通信量的上升会导致所有连接设备的反馈变慢。

LabJack U3 在其接线端子排上提供了 8 个 FIO（灵活 I/O）端口，在 DB-15 连接器上 561
还有 8 个 EIO（扩展 I/O）端口和 4 个 CIO（控制 I/O）端口。此外，还有两个 DAC 输出，6 个 VS（+5VDC）接线端，5 个公共接地端和两个保护接地端。注意，标识为 AIN0 到 AIN3 的接线端其实是编号从 FIO0 到 FIO3 的 FIO 端口。这几个端口被默认设置为模拟输入，但你可以通过修改 LabJack 的配置来改变其功能。可以用 Windows 下的 LJControlPanel 工具（每个 LabJack 都有附带）来修改配置，或者也可以使用接口驱动命令来进行设置。

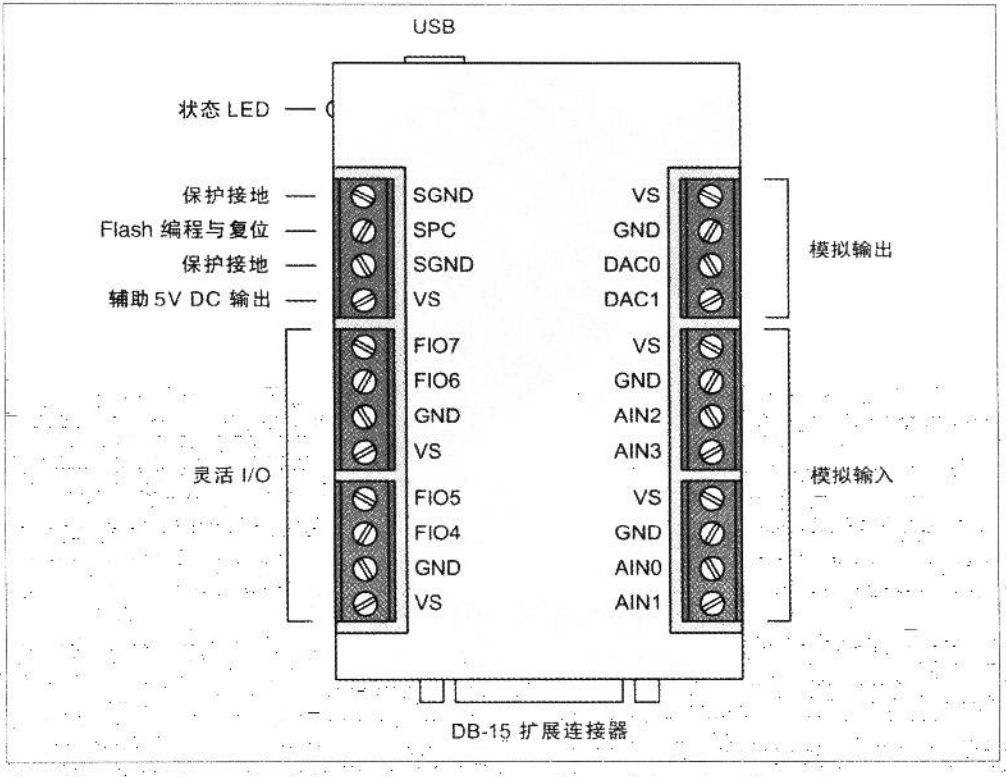


图14-7：LabJack U3-HV连接

除了 U3-HV 型号，每个 FIO 和 EIO 端口都能被设置成数字输入 / 输出或模拟输入。在 U3 的 HV 版本中，FIO0 至 FIO3 线皆被设为模拟输入而且不能被重新配置。DB-15 连接器中的 4 条 CIO 线是专门的数字线。两个 DAC 输出可以工作在 8-bit 模式或 16-bit 模式，范围是 0.04V 至 4.95V。使用 8-bit 模式会得到更稳定的输出和更少的杂波。

562 U3 还有两个定时器和两个时钟。任何时候，只要一个定时器或时钟被使用，它就要占用一个 FIO 通道。以 U3-HV 为例，FIO4 会第一个被占用，接着是 FIO5，依此类推。标准 U3 则从 FIO0 开始进行计数器 / 定时器端口分配。

安装 LabJack 设备

LabJack 为 Windows 和 Linux 都提供了驱动，还有完整的 Python 接口包。首先我们会概览一下如何在 Windows 或 Linux 下安装和运行 LabJack——实际上这非常简单。然后下一小节我们将深入了解 LabJack 的 Python 接口。

Windows 下的 LabJack

在 Windows 下使用 LabJack 设备，需要安装 LabJack 配套 CD 中提供的 UD 驱动。在将 LabJack 连接至你的电脑前应先安装该驱动。每个 LabJack DAQ 设备都配有一张单页的快速上手指南，可使整个上手过程迅速而简易。

Linux 下的 LabJack

在 Linux 下使用 LabJack 设备，需要安装 LabJack 配套的 Exodriver 驱动包。Exodriver 是以源代码形式提供的，唯一的依赖条件是 *libUSB* 库。该库文件在大多数现代 Linux 系统中一般都已自带，即使没有也能通过包管理器（package manager）很容易地安装。此外，你还需要一个 C 编译器（因为大多数的 Linux 发行版都自带了 *gcc*，这应该不成问题）。

编译驱动程序非常简单，只需键入 `make` 命令即可，编译成功后，执行 `make install`。执行安装步骤时，你需要 *root* 权限（或使用 *sudo* 命令）。LabJack 官方网站上有详细的安装指南，详见 *http://labjack.com/support/linux-and-mac-os-x-drivers*。

LabJack 与 Python

LabJack 设备的 Python 接口，被称为 LabJackPython，是对 UD 和 Exodriver 底层接口的封装。它允许使用该驱动的功能，并同时支持了 Modbus 基于寄存器式命令以及 LabJack 驱动中的底层功能。

安装和测试 LabJackPython

一旦安装好 LabJack 设备并成功跑起来之后，接下来就可以下载并安装 Python 封装器。如果想了解更多有关 Python 绑定的信息，请参考 *http://labjack.com/support/labjackpython*，该网页同时也包含了安装指南。

总而言之，所有你真正需要做的就是下载最新的 LabJack Python 文件（在写本书时是 *LabJackPython-7-20-2010.zip*），解压缩，切换到 src 目录，然后输入： 563

```
python setup.py install
```

就这么简单。然后你就能启动 Python 并导入 LabJack 模块了，如下（如果你有 U6，改为导入 U6）：

```
>>> import u3
```

接着我们需要创建一个 U3 类的对象，后面的设备操作命令将使用该对象：

```
>>> u3d = u3.U3()
```

如果 LabJack 已连接并准备就绪，你不会看到该命令返回任何内容，但这已经为输入指

令读取数据做好准备了。

LabJack 驱动结构

现在我们离开串行接口设备的领域，深入 API 的领地，这意味着我们需要理解 LabJack 驱动 API 是如何构造的以便更有效地使用它。和 LabJack 这样的设备打交道很像是作业于总线式设备和它的底层驱动。有着很多的功能，但并不像我们之前看到的串行设备那样容易使用。

在我们深入话题之前，你应至少浏览一下 LabJack 的用户指南。也应将 LabJackPython.py 和 u3.py（也可能是 u6.py）这些源文件准备在手，因为你将不时地参阅它们。

LabJack 的驱动 API 是由一系列的层级组织起来的，它从 Exodriver 或 UD 的驱动开始，最终以为特定的 LabJack 设备型号裁剪的功能作为结束。如图 14-8 所示，其中也展示了 u12 接口，不使用 LabJackPython 接口模块而使用其自己的 liblabjackusb.so 硬件驱动的独立 API。将其包含在图 14-8 中只是为了完整性考虑，本小节我们将主要关注 u3。

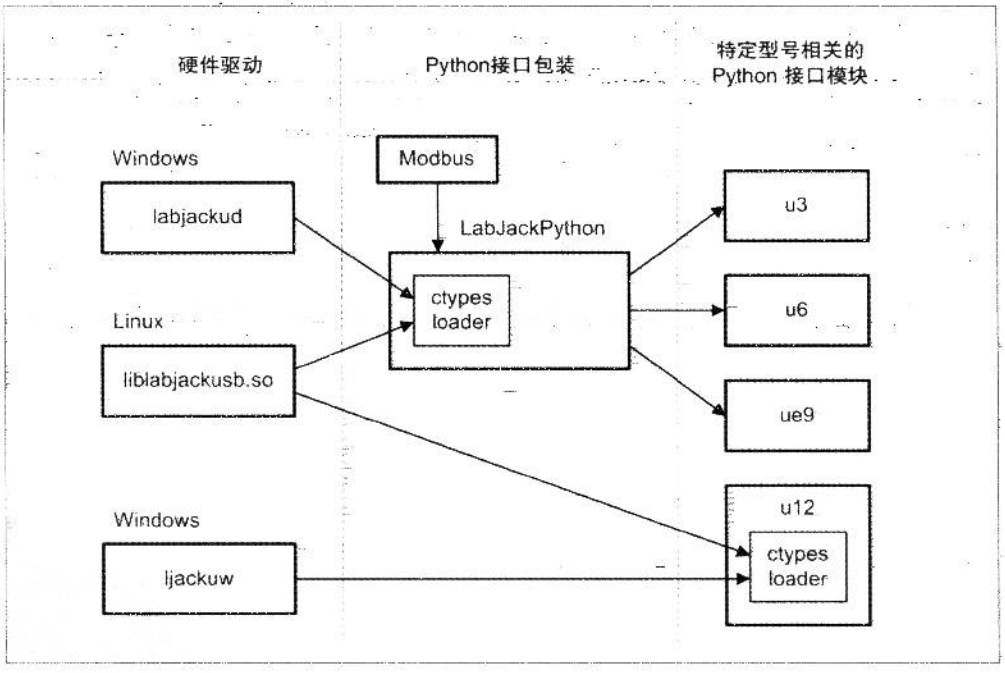


图14-8：LabJack软件接口组件

我们首要关注的是 LabJackPython.py 模块和 u3.py 模块。u3 模块中包括了 U3 类的定义，

它是基于 LabJackPython 中 Device 类的一个派生类。图 14-9 展示了 Device 和 U3 类的继承关系。

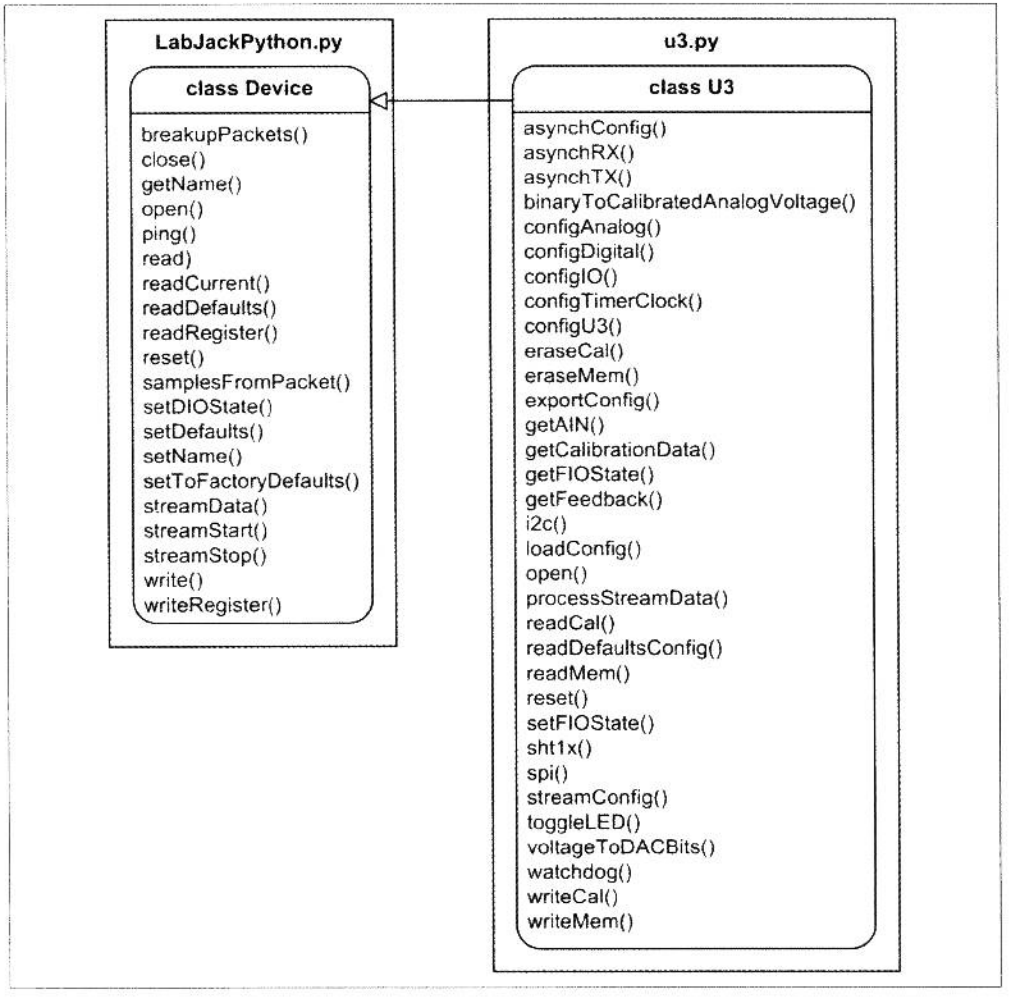


图14-9：U3接口类的继承

除 Device 和 U3 类以外，LabJackPython 和 u3 模块中还有很多非常有用的方法，但需要确认其中的方法不是特定于操作系统的。例如，LabJackPython.py 中的 ePut() 和 eGet() 方法只针对 Windows。如果你选择使用这些方法，请记住你的程序将不再是可移植的。

除 U3 类外，LabJackPython 模块中还有一些方法和类是不用 U3 类就能访问的。例如， 564

得到已连接系统的 U3 型设备列表，你可以这样使用 listAll() 方法：

```
>>> import u3
>>> u3.listAll(3)
```

参数值 3 代表你想要 U3 设备的列表，如果是 U6 设备的话你就用 6。我在这个例子中得到一个字典对象，其中只有一个 U3：

```
{320037071: {'devType': 3,
'serialNumber': 320037071,
'ipAddress': '0.0.0.0',
'localId': 1}}
```

如果你安装了 Epydoc（参见第 8 章），我建议使用它来创建一系列 HTML 页面作为不同 Python 组件的在线文档。你将得到比使用 Python 的 help() 命令或看源代码容易阅读得多的格式美观的文档。当在 LabJackPython.py、Modbus.py 和 u3.py 模块上使用时，Epydoc 的输出就像图 14-10 所示的截屏一样。

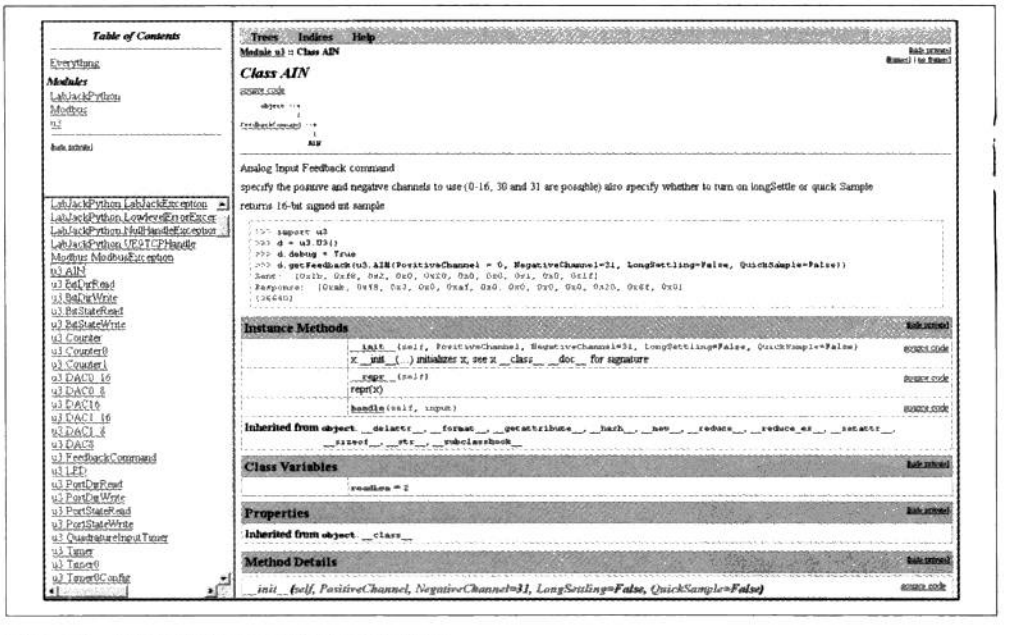


图14-10：Epydoc对LabJackPython的输出

565

U3 配置

管理 U3 设备的配置是一件重要的事，这使得设备启动时端口定义能如预期那样作用于特定的应用。U3 类为这一目的提供了 configIO()、configAnalog()、configDigital()、

configTimerClock() 和 configU3() 方法。LabJack 文档对各个配置管理方法进行了详细的说明。

这里展示一下如何使用 configU3()：

566

```
>>> import u3
>>> u3d = u3.U3()
>>> print u3d.configU3()
```

这将返回一个字典对象。在我的系统上，其返回值如下：

```
{'TimerClockConfig': 2, 'TimerClockDivisor': 256, 'LocalID': 1,
'SerialNumber': 320037071, 'CIOState': 0, 'TimerCounterMask': 64,
'DAC1Enable': 1, 'EIODirection': 0, 'DeviceName': 'U3-HV',
'FIODirection': 48, 'FirmwareVersion': '1.24', 'CIODirection': 0,
'DAC0': 0, 'DAC1': 0, 'EIOAnalog': 0, 'CompatibilityOptions': 0,
'EIOState': 0, 'HardwareVersion': '1.30', 'FIOAnalog': 15,
'VersionInfo': 18, 'FIOState': 0, 'BootloaderVersion': '0.27',
'ProductID': 3}
```

如果你能使用 Windows 电脑，或者计划在 Windows 下使用 LabJack，我建议使用 LJControlPanel 工具，至少到你熟悉了配置管理方法和接口模块中的方法为止。

配合 getFeedback() 方法使用 Python 接口

如果你大概看过 u3.py 模块，你也许会注意到，在 U3 类以外还有一些小类的定义，它们都继承了 FeedbackCommand() 类。（如果你还没有看过，现在最好看一看。）。在 U3 类的 getFeedback() 方法中，它们被用来执行各种操作。

下面是 U3 类中 getFeedback() 方法的定义：

567

getFeedback(*self*, **commandlist*)

commandlist 参数是一个 FeedbackCommand 型对象的列表。getFeedback() 方法将该列表打包传给 U3。返回一个含有零或多个条目的列表，在输入列表中每个条目对应一个对象。

这是一个控制 U3 侧面的状态 LED 的简单例子：

```
>>> import u3
>>> u3d = u3.U3()
>>> u3d.getFeedback(u3.LED(0))
[None]
>>> u3d.getFeedback(u3.LED(1))
[None]
```

你可以如下设置 DAC 输出电压：

```
>>> u3d.getFeedback(u3.DAC16(Dac=0, Value = 0x7fff))
[None]
>>> u3d.getFeedback(u3.DAC16(Dac=1, Value = 0xffff))
[None]
>>> u3d.getFeedback(u3.DAC16(Dac=0, Value = 0x0))
[None]
>>> u3d.getFeedback(u3.DAC16(Dac=1, Value = 0x0))
[None]
```

输入前两个 DAC 命令后，你应能在 DAC0 输出端子上测量到一个 2.5V 左右的电压，在 DAC1 上则大约有 4.95V 的输出电压（0x7fff 是 0xffff 的一半）。最后两个命令将 DAC 输出设置回 0。getFeedback() 方法调用了 u3.DAC16() 方法去给 DAC 传值。注意这里返回值是 None。

下面是 u3.py 模块中 FeedbackCommand 继承类命令的列表。为了更易读，我把类当成一个方法或函式列出了每个类和传给其 __init__() 函式的参数：

- AIN(*PositiveChannel, NegativeChannel=31, LongSettling=False, QuickSample=False*)
- WaitShort(*Time*)
- WaitLong(*Time*)
- LED(*State*)
- BitStateRead(*IONumber*)
- BitStateWrite(*IONumber, State*)
- BitDirRead(*IONumber*)
- BitDirWrite(*IONumber, Direction*)
- PortStateRead()
- PortStateWrite(*State, WriteMask=[0xff, 0xff, 0xff]*)
- PortDirRead()
- PortDirWrite(*Direction, WriteMask=[0xff, 0xff, 0xff]*)
- DAC8(*Dac, Value*)
- DAC0_8(*Value*)
- DAC1_8(*Value*)
- DAC16(*Dac, Value*)
- DAC0_16(*Value*)
- DAC1_16(*Value*)
- Timer(*timer, UpdateReset=False, Value=0, Mode=None*)
- Timer0(*UpdateReset=False, Value=0, Mode=None*)

568

- Timer1(*UpdateReset=False, Value=0, Mode=None*)
- QuadratureInputTimer(*UpdateReset=False, Value=0*)
- TimerStopInput1(*UpdateReset=False, Value=0*)
- TimerConfig(*timer, TimerMode, Value=0*)
- Timer0Config(*TimerMode, Value=0*)
- Timer1Config(*TimerMode, Value=0*)
- Counter(*counter, Reset=False*)
- Counter0(*Reset=False*)
- Counter1(*Reset=False*)

虽然这看起来像是代码中使用的方法，但它们实际上是实例化对象的类的定义。这些对象被 u3d.getFeedback() 方法调用来执行一个特定动作或一组动作。更多描述参见 u3.py 的源代码（或者 Epydoc 的输出，如果你已经生成了的话）。还有，在导入 u3 模块后你便能像这样使用 Python 内建的 help() 方法了：

```
>>> import u3
>>> help(u3.WaitLong)
```

现在我们已经知道 getFeedback() 方法能做什么了，现在让它工作起来。u3ledblink.py 是一个简单的脚本，能让状态 LED 闪烁，直到在 AIN0 端子的输入电压超过某预设的限制：

```
#!/usr/bin/python
# LabJack demonstration
#
# 闪烁 U3 的状态 LED，直到 AIN0 上的输入超过限制值
#
# 源代码出自 "Real World Instrumentation with Python"
# By J. M. Hughes, published by O'Reilly.

import u3
import time

u3d = u3.U3()

LEDoff  = u3.LED(0)
LEDon   = u3.LED(1)
AINcmd  = u3.AIN(0, 31, False, False)

toggle = 0

while True:
    # 通过循环完成闪动
    if toggle == 0:
```

569

```
            u3d.getFeedback(LEDon)
            toggle = 1
        else:
            u3d.getFeedback(LEDoff)
            toggle = 0

        # getFeedback 返回一个只带一个元素的列表
        inval = u3d.getFeedback(AINcmd)[0]
        print inval
        if inval > 40000:
            break
        time.sleep(1)

    u3d.getFeedback(LEDon)
    print "Done."
```

任何在 0 到 5V DC 之间的电压源都能作为 AIN0 的输入。用一条短接线触碰一下 AIN0 和 VS 端子也能达到让 LED 停止闪烁的目的。

getFeedback() 并不限于一次一个命令，你可以叠加多个命令。这里是一个 FIO4（在我的设置里被配置成了数字输出）迅速开关的例子。它速度很快，你需要一个示波器才能看清其活动：

```
>>> biton = u3.BitStateWrite(4,1)
>>> bitoff = u3.BitStateWrite(4,0)
>>> u3d.getFeedback(bitoff, biton, bitoff, biton, bitoff)
```

使用 getFeedback() 时应注意在默认的超时设置下有些命令的响应会很花时间，你会得到驱动抛出的异常。WaitLong() 类的命令也许看起来很有吸引力，但使用 Python 的 sleep() 方法而不是在一个命令对象列表中嵌入延迟会更保险。总之，记得让事情保持简短，并且不要将太多的命令堆在 getFeedback() 里就行了。

现在你已经得到足够的信息来让 LabJack 做有用的事情了。但这只是冰山一角，该设备还能做很多事情。用户指南和 LabJackPython 源代码中包含了使用更多高级功能所需的信息。

570

小结

本章的例子只是各种可能仪器硬件的一些小样本，从中也可以看到用 Python 采集数据和控制外部设备能做些什么。我们没有讨论 GPIB 或总线式 I/O 的具体例子，尽管这类设备在仪器系统中被广泛地应用。但如果你回顾第 2 章、第 7 章和第 11 章，会发现这些主题在那些章节里有详细的讲解。

本章讨论的设备是特地选择来阐述仪器硬件中一些普遍概念的，比如应用广泛的命令-应答范式。从这个角度讲，串行数据采集设备在很多方面都与 GPIB 设备有相似之处，并且一些实验仪器拥有在使用 SCPI 模型的同时通过串口通信的能力。LabJack 数据采集仪器利用了一个驱动 API 和 Python 接口层，你会发现这和 PC 总线多功能 I/O 卡上的东西非常相似。一旦你理解了各种接口背后的基本概念，在与任何仪器设备打交道时你都将变得应付自如。

只要稍微花点钱你就能将几乎任何 PC 转变成数据采集和控制系统，使用 Python，你能编写程序让它做任何你想让它做的事，同时利用到强大且具有扩展性的 Python 开发环境。你的程序能根据需要变得简单或者复杂，而且因为这是你的软件，你能在需要的时候修改它以跟上需求和仪器环境的变化。

如果你对电子学熟悉的话，你甚至可以创造自己的 I/O 硬件，比如我们在第 11 章讨论过的并行端口接口。你也可以通过并入网络和分布式控制来扩展系统的能力。

对于可能做到的，我们只是接触了皮毛。随着你对仪器设备和系统了解的加深，越来越多的可能性会开始变得明显。值得一提的是，某种程度上的自动化的例子在我们的实验室和工厂无处不在，在生产楼里、在我们的车辆中、在我们的办公室和家里，通过一些实践、一点耐心再加上你的创造力，你便能应用本书中的信息去设计和创造稳定而优雅的数据采集和控制系统去满足你的特定需求。

推荐阅读

本书的主题为你提供了使用本章涉及设备所需的信息。当然，当要安装和编程像数据变送器和 DAQ 单元这类设备时，主要信息源应该是制造商提供的手册。对新设备你应仔细阅读文档，因为并非所有设备都很直观易用（实际上，这类设备非常之少）。为此，我 571
把本章所涉及设备的一些线上文档的链接以及其他信息资源列在下面：

http://www.omega.com/DAS/pdf/D1000.pdf

Omega 的 D1000 数据变送器产品系列概述。

http://www.omega.com/Manuals/manualpdf/M0662.pdf

Omega D1000 系列的用户手册。

http://labjack.com

关于 U3 和 U6 LabJack 设备还有 Python 接口的资料。这个网站包含有应用笔记、博客和其他很多对于 LabJack 用户有用的条目。

附录 A 573

自由和开源软件资源

本附录仅包含书中提及的各种工具和库以及模块，算不上详尽的清单。有许多优秀的开源软件包可工作在 Linux 和 Windows 的 Python 上。我会鼓励读者亲自去探索，如 SourceForge 和 Berlios 网站。Python 官网 *python.org* 也链接有其他 Python 特定领域的软件包。

Python 2.6.5 资源

http://www.python.org

Python 官方发行版网站。

http://www.activestate.com

ActiveState 网站，一种 Windows 为中心的 Python 发行版，包含很多适用工具。

IDE（综合开发环境，Integrated development environments）

Boa Constructor

Python 基础 IDE，包含完备的 GUI 设计器（wxPython）。

http://boa-constructor.sourceforge.net

Eclipse

强大的基于 Java 的 IDE 通用插件（*PyDev*），支持 Python。

http://www.eclipse.org

Idle

内置在 Python 发行版中的多窗口 IDE（基于 Tk 框架）。

PythonWin 574

ActiveState 发行版中内置的仅 Windows 平台可用 IDE，也从 SourceForge 独立发布 :*http://sourceforge.net/projects/pywin32/*

pySerial

跨平台串口界面库，支持 RS-232 标准串口。*pySerial* 支持类文件操作接口，使用类似 `read()` 和 `write()` 方法来进行二进制数据传输（非 EOL 传输或是比特拆封）。

http://pyserial.sourceforge.net

pyParallel

支持访问 Windows 和 Linux 平台中的标准并行端口的并口库。

http://pyserial.sourceforge.net/pyparallel.html

pyUSB

通过底层驱动和 Python 的 `ctypes` 库，支持访问 Windows 和 Linux 上的 USB 设备。

http://sourceforge.net/apps/mediawiki/pyusb/index.php?title=Main_Page

PyVISA

一个跨平台的 VISA API，支持商业 *visa32.dll* 及各个厂商的同类 DLL 模块，在 Linux 中，VISA 由 National Instruments 的 `libvisa.so.7` 库提供支持。

http://pyvisa.sourceforge.net

NumPy

Python 的数学包，支持 *n* 维数组、线性代数、傅里叶变换和随机数生成等功能。

http://numpy.scipy.org

SciPy

为数学、科学、工程计算提供了丰富工具和功能的库。

http://www.scipy.org

PyUniversalLibrary

Measurement Computing 公司的通用驱动和接口套件的接口库，仅用于 Windows 系统。

https://code.astraw.com/projects/PyUniversalLibrary/

575 GUI 库和开发工具

Boa 构建器（wxPython）

包含编辑和调试器的 wxPython GUI 设计器。

http://boa-constructor.sourceforge.net

PAGE（Tkinter）

tcl/Tk 写成的拖曳式 GUI 构建器，可生成使用 TkInter 库的模块。

http://page.sourceforge.net

PythonCard (wxPython)

基于模板的 wxPython GUI 设计器。

http://pythoncard.sourceforge.net

pmw (Tkinter)

Python 的 TkInter megawidgets 增补，包含各种 TkInter 发行版中没有的组件和功能。

http://pmw.sourceforge.net

SpecTcl (Tkinter)

另一个 TkInter 图形窗口设计工具，是 Sun 公司原创的，已在 Sun 许可协议下开源。

http://spectcl.sourceforge.net

wxGlade (wxPython)

在 Glade 设计工具之上的 wxPython GUI 设计器。

http://wxglade.sourceforge.net

wxPython

Python 的 wxPython 库，是对 wxWidgets 库的包装，wxWidgets 本身并不依赖 Python。

http://wxpython.org

绘图工具

gnuplot

在 Linux 和 Windows 中都能工作的强大绘图包，支持 2D/3D 绘图，并有丰富的内置数学函式。

http://gnuplot.sourceforge.net

gnuplot.py

一个方便 *gnuplot* 与 Python 协同工作的接口层和命令转换辅助工具。

http://gnuplot-py.sourceforge.net

系统工具和实用程序 576

ansicon

Windows *ANSI.dll* 驱动的替代品，支持全新 32 位和 64 位的 Windows。

http://adoxa.110mb.com/ansicon/index.html

com0com

Windows 内核模式驱动，用于创建通过零调制解调器连接的虚拟串行端口。

http://com0com.sourceforge.net

Cygwin and Xcygwin

在 Windows 下提供 Linux 模拟环境。Xcygwin 包包含了 X server 以及各种 X 应

用程序 (Xterm、Xclock、Xfig，等等)。

http://www.cygwin.com

Tera Term

古老但依然好用的终端模拟，包含一种不错的脚本语言。

http://hp.vector.co.jp/authors/VA002416/teraterm.html

tty0tty

Linux 下的 com0com，允许两个 tty 设备以零调制解调器的方式建立交叉连接。

http://tty0tty.sourceforge.net

图像数据工具和库

ImageJ

国立卫生研究院发布的一个灵活且高度可配置的图像浏览器和图像处理工具。

http://rsbweb.nih.gov/ij/

Netpbm

Netpbm 库是图像转换和处理工具。

http://netpbm.sourceforge.net/doc/pgm.html

577 数据采集设备的 Linux 驱动程序包

Comedi

低层驱动的集合，允许 Linux 系统与各类数据采集和数字接口卡通信。包括 SWIG 脚本来创建驱动的 Python 包装。

http://www.comedi.org

Measurement Computing Linux Drivers

来自北卡罗来纳州立大学的 Warren Jasper 写的整套 Measurement Computing 公司各种产品的 Linux 驱动。注意，只是 Linux 的驱动，没有提供 Python 的 API。需要准备自己的 Python 包装。

ftp://lx10.tx.ncsu.edu/pub/Linux/drivers/

很有可能这里漏掉了一个或两个（甚至更多）与本书有关的软件包，但不是故意的。从 Google 或 SourceForge 上搜索“Python control systems”或“Python data acquisition”能返回一些更有趣的包。

Python 软件包列表也发布在 *http://pypi.python.org/pypi/*，这张列表相当长。另外，如果你使用 Linux，一定要使用包管理器来检查，看看有什么包可用于 Python，这比手动安装容易很多（而且解决了软件包的依赖关系）。

最后我想指出，你应当正确理解，如果正在使用开源软件包，它可能不如一些商业产品，它可能包含一些错误或不完整。但也牢记，对你而言成本为零，而它们的作者只因热爱开发以及渴望分享他们的工作，而在用自己的时间来工作，请维护这些软件！

附录 B 578

仪器资源

本附录包含了测试设备和数据采集仪器制造商以及二手设备网点。当然，这只是一个参考，你完全可以使用列表之外的任意设备。

记住，仪器的功能和价格的变化范围非常大，而且这两者之间不总是呈对应关系。在支付数百（甚至数千）美元买一块硬件之前，值得花时间做些研究，以判断该产品是否符合你的要求。

本附录的另一个目的是帮助你找到旧仪器的技术文档。有许多上古时代的数字万用表、控制器、电闸和数据采集单元，仍然具备完美的功能，只等你拿着它们的文档用起。记住，在实验室架子上或是储藏室中，总是有一些老伙计“含情脉脉”地等待着你。

厂商

以下是仪器和测试设备制造商的简要名单。为每个厂家只罗列了具有代表性的产品类型，而且，在某些情况下列出的只是他们生产的很少一部分。

Agilent

示波器、逻辑分析仪、数字仪表、信号发生器、频谱分析仪、电源开关、数据采集单元、等等。

http://www.agilent.com

AMETEK

电源和电子负载。

http://www.programmablepower.com

Anritsu 579

射频、微波、光学以及数据通信测试设备。

http://www.us.anritsu.com

Bitscope

USB 示波器、信号发生器和逻辑分析仪。

http://www.bitscope.com

B&K Precision

示波器、数字仪表、信号发生器、频谱分析仪、电源等。

http://www.bkprecision.com

Elan Digital Systems

微型 USB 示波器、函式发生器、脉冲发生器和计数器。

http://www.elandigitalsystems.com/measurement/index.php

Fluke

手持和台式仪表、信号源、分析仪和示波器。

http://www.fluke.com

Keithley

数据采集系统、函式发生器、数字仪表和电源。

http://www.keithley.com

LeCroy

示波器、逻辑分析仪和协议分析仪。

http://www.lecroy.com

Saleae

低成本 USB 逻辑分析仪。

http://www.saleae.com

Stanford Research Systems

科学数据采集和控制仪器，再加上测试和测量设备。

http://www.thinksrs.com

Tektronix

示波器、逻辑分析仪、数字仪表、信号发生器、频谱分析仪、电源、数据采集单位等。

http://www.tek.com

580

二手测试设备资源

下面列出几个提供二手和翻新测试设备的公司。虽然从 eBay 或本地二手市场总能淘到很多好东西，但当你从一家专业二手测试设备公司采购时，他们的设备通常是经过校准的（或者，如果你要求的话，他们就会校准），而且通常带有 90 天的保修期。通常情况下，你也将得到所有必要的探头、电缆和表笔，这样就少了和 eBay 上那些商家打交道的烦恼，他们会事先把这些配件拆下来并把它们单独出售。有时，还附带有免费的用户手册。

AccuSource Electronics

http://www.accusrc.com

Metric Test

http://www.metrictest.com

Test Equipment Depot

http://www.testequipmentdepot.com

Test Equity

http://www.testequity.com

Tucker Electronics

http://www.tucker.com

手册

有时很难找到一块旧设备的使用手册，要么是因为制造商已不存在，要么是因为不再提供下载。此时，你可以看看下面这些网站。多数情况下，他们所保存的是原始的手册，而不是 PDF 文件，这些手册的价格大约在 20 ~ 50 美元之间，而且如果手册很大或是很罕见，价格还会更高一些。有时，电子版文档可以免费下载。

eServiceInfo

http://www.eserviceinfo.com

Manuals Plus

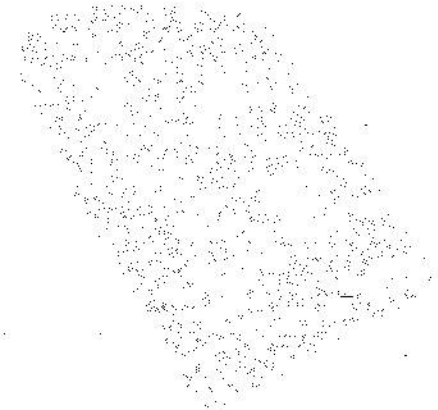
http://www.manualsplus.com

Technical Specialists

http://schematics4you.com

Your Manual Source

http://www.yourmanualsource.com



索引†

Symbols

16bpp image, 483
8bpp PGM image, 480
>> (chevron operator), 114
>>> (Python command-line prompt), 62
{ } (curly brackets), 127
(hash character)
  preprocessor directives and, 128
  Python, 85
+ (plus) operator (Python), 72
?: (ternary conditional), 130

A

AC (alternating current), 19, 30
AC circuits, 30–39
AC power controller simulator, 371–380
  (see also SPC)
  simulator block diagram, 373
activity diagrams, 267
ADCs (analog-to-digital converters), 4, 44, 45
aliased cyclic function readout, 371
ALL command (SPC), 375
Allen wrenches, 197
alternating current (AC), 19, 30
amperes (A or amp), 20
amplitude, 31
AN/ARC-220 mobile radio, 213
analog data, 2, 3
analog data sampling, 45
analog I/O, 44
analog-to-digital converters (ADCs) (see ADCs)
AND gate logic, 24
AND operator (C language), 139
ANSI bar graph example output, 504
ANSI escape sequence standards, 500
ANSI X3.64, 500
ansicon package, 501
ANSITerm class, 505
arithmetic operators (C language), 134
arithmetic operators (Python), 78
arrays (C language), 150
ASCII character encoding standard, 439
ASCII data files, 438–463
  configuration files, 449–451
  flat files, 442
  I/O utilities for, 454–463
  Microsoft Excel and, 443
  Python handling of ASCII characters, 439
  reading, 445
  string conversion with AutoConvert.py, 451
  writing, 444
ASCIIData formats, 454
ASCIIDataWrite and ASCIIDataRead classes (Python), 454–463
assert statement (Python), 86
assertEqual( ) and assertNotEqual( ) methods, 291
assignment operators (C language), 136
assignment operators (Python), 81
assignment statement (Python), 86
asynchronous serial data communication, 52
AT bus, 397
atom organization, 16
attributes, 60
augmented assignment operators (C language), 137
augmented assignment statements (Python), 86

†： 索引所列页码为本书英文版页码，请参照用“▭”表示的原书页码。

AutoConvert.py module, 451
automated instrumentation, xiii

B

backshell assemblies, 209
bang-bang controllers, 326–329
  Python, implementation in, 341
barrier terminal block, 213
basic ANSI control sequences, 501
basic USB connector types, 211
basic use case elements, 260
Batchelder, Ned, 293
baud, 222
bench model digital multimeters, 193
bgansi.py, 502
binary data files, 463–466
  flat files, 464
  image data (see image data)
  Python, handling with, 466
    ctypes module, 466–471
    struct module, 471–476
binary shift operators (Python), 81
binding, 528
bitwise operators (C language), 138
bitwise operators (Python), 80
block diagrams, 266, 310
blocking function calls, 421
Boa Constructor wxPython GUI builder, 536
Boa's widget layout window, 536
break statement (C language), 149
break statement (Python), 87
"bubble" symbols, 24
bug tracking, 299
built-in scope (Python), 101
bus-based hardware I/O devices and Linux, 412
bus-based interfaces, 241
  advantages and disadvantages, 242
  DAQ cards, 244
  GPIB cards, 244
  PCI bus, 242
bytecode, 61

C

C programming language, 123
  ANSI C standard library, 158
  building C programs, 159–162
  code blocks, 127
  development tools, 163
  installing, 123–
  libraries, 125
  online resources, 164
  operators, 134–140
  Python extensions (see Python extensions)
  return types of functions, 127
  simple program example, 125
  software development using, 124–163
  standard data types, 132
  user-defined types, 133
cable, 203
callable methods, 171
callbacks, 529
capacitors, 32
case conversion of ASCII characters, 439
cdll interface class, 484
checklists, 283
chipsets, 397
CHK command (SPC), 376
chr( ) function (Python), 440
circuit schematics, 20
circuit theory, 19
circular connectors, 212
class statement (Python), 103
classes, 60
ClearPortBit( ) function, 183
closed-loop control systems, 6, 305
closed-loop water tank level control, 319–
cmd.exe, 487
code coverage analysis, 293
code examples, xx
coding, 279–300–
  code reviews, 282
    checklists, 283–285
  coding styles, 280
    guidelines for Python, 117
  connecting to hardware, 295
  defect tracking, 299
  documentation, 296
  organizing of code, 281
  unit testing, 286–295
  user documentation, 300
  version control, 299
com0com package, 236, 372, 501
com2tcp, 236
Comedi project and package, 412
  Python, using with, 414
  supported hardware, 413

command line (Python), 61
  command-line options, 63
comments (C language), 127
comments (Python), 85, 116
common electronic schematic symbols, 20
common method flags (Python extensions), 172
comparison operators (C language), 137
comparison operators (Python), 79
compiled programs, 124
complex data type, 65
compound statements (Python), 87
config files, 449
connector backshells, 209
connectors, 208–218
  connector failures, 218
  crimp-type connectors, 216
  crimp-type spade lugs, 214
  proper wiring of, 217
console interface, 487–500
console1.py, 488
console6.py, 512
constructor methods (Python), 65
contact bounce or contact chatter, 42
containers, GUI, 526
continue statement (C language), 149
continue statement (Python), 87
contract-style requirements, 258
control error, 7
control output, 4
control systems, 304
  bang-bang controllers, 326–329
  closed-loop controls, 319–325
  control system types, 318–340
  examples, 9–13
  hybrid control systems, 340
  open-loop controls, 319
  proportional, PI, and PID controllers, 332
  Python, implementation in, 340–346
    bang-bang controller, 341
    linear proportional controller, 340
    simple PID controller, 342–346
  sequential control systems, 330–332
control systems theory, 304
  block diagrams, 310
  control signal, 311
  control system diagram symbols, 310
  control system notation, 305
  controlled output, 311
  controlled variable, 311
  discrete-time control systems, 315
  feedback, 312
  input-output relationships, 311
  linear control systems, 305
  nonlinear control systems, 306
  sequential control systems, 308
  software flow, 317
  software timing, 318
  terminology, 309
  time and control system behavior, 315
  time and frequency, 313
  transfer functions, 312
controls (GUIs), 526
counters, digital, 49
coverage package, 293
CPS (characters per second), 222
crossover cable, 224
CSV (comma-separated values) files, 448
csv library (Python), 449
ctypes library (Python), 167, 466
  basic data types, 186
  ctypes_struct.py, 466
  foreign function library, 184–187
    loading external DLLs, 184
  using, 187
ctypes, C, and Python types, comparison of, 186
curly brackets ({ }), 127
current, 20
current flow, 17
current sink and current source, 27
curses library (Python on Unix/Linux), 515–524
  advantages and disadvantages, 523
  curses history, 516
  selected functions and windows methods, 517
  simple data display using, 517–522
    subwindows, adding, 522
curses window hierarchy, 516
curses1.py, 518
CVS, 299
cyclic functions, 371

D

D1000 series data transmitters, 553
  D1000 commands, 556
  D1121 thermal chamber application, 553

DACs (digital-to-analog converters), 4, 44, 48
DAQ (Data Acquisition) cards, 244
(see also bus-based interfaces)
data acquisition, 2
data acquisition and control systems, xiv
data displayed as an image, 477
data files
ASCII data files (see ASCII data files)
data I/O, 414–421
handling data I/O errors, 426–431
handling inconsistent data, 431–435
noise and averaging, 434
waiting for stability, 432
reading data, 415–416
writing data, 416–421
data I/O interface software, 395
data I/O methods, 423–426
acquiring data using threads, 424
on-demand data I/O, 423
polled data I/O, 423
data sink and data source, 224
data types (C language), 132
data types (Python), 65
DB-9 null-modem adapter, 225
DB-9 pin and socket numbering, 209
DB-type connectors, 208
DC (direct current), 19
DC blocking, 33
DC circuits, 24–30
DC motor control, 51
DC offset or bias, 33
DCE (Data Communications Equipment), 223
debuggers (C language), 163
debuggers (Python), 120
DEC VT100 VDT control sequences set, 501
def statement (Python), 101
defect tracking, 299
defects, testing for (Python), 115
deferred imports (Python), 115
#define macro directive, 129
delimiters, sequence objects (Python), 69
delta, 432
derived requirements, 258
destructor methods (Python), 65
development environments (Python), 117–120
DevSim
cyclic and file input threads, 359
cyclic functions, 371
DevSim API, 357–362
DevSim internal logic, 357
DevSim methods, 362–366
examples, 366–368
noise, 371
parameter accessor methods, 363
simulator control and I
O methods, 365
usage example, 368
user-defined functions, 368–
dictionaries (Python), 75
risks of using as function parameters, 115
dictionary methods (Python), 75
digital data, 3
digital logic symbols, 23
digital multimeters (DMMs), 189, 192–195
usage tips, 194
digital oscilloscope display, 199
digital-to-analog converters (DACs) (see DACs)
digitizers, 403
direct current (DC), 19
discrete digital I/O, 40
discrete-time closed-loop control system, 316
discrete-time control systems, 315
DLL wrapper extension and support module, 184
DLLs (dynamically linked libraries), 167
DMM data capture, 548
DMM function and DMM range codes, 550
do-while loop (C language), 148
docstrings (Python), 104, 116, 296
drivers, 398
droop, 338
DTE (Data Terminal Equipment), 223
DTE/DCE MODEM communications, 223
dual-trace oscilloscopes, 198
duty cycles, 39

E

ECB command (SPC), 376
ECMA-48, 500
EIA-232 interface (see RS-232 interface)
EIA-485 interface, 225
electric charge, 17
electromagnetic tachometer, 37
Electronic Industries Alliance (EIA), 218
electronic tool sources, 192
electronics, 15–55
AC circuits, 30–39
circuit schematics, 20

circuit theory, 19
DC circuit characteristics, 24–30
electric current, 17
electrical charge, 15
interfaces, 39–55
electronics supplies, 203
electronics test instrumentation, 9
electrostatic cathode ray tubes (CRTs), 198
else block (Python), 89
endwin( ) function, 516
Epydoc tool, 297
output example, 298
error handling, testing, 277
event handlers and event sources, 528
event-driven programming, 528
exception trapping and handling (Python), 90
execution environment, 361
exiting the Python command line, 62
exponent operator (Python), 78
expressions (C language), 143
expressions (Python), 77
extended C data types, 132
extensions, 167

F

failure analysis, 273
failure responses, 273
farads (F), 32
faults, 352
feedback, 6, 304, 312
feedback control systems, 319–325
feedback controllers, 305
FeedbackCommand class commands, 567
FETCh command, 404
fields, 442
file I/O modes, 113
file methods, 114
FileUtils module, 356
FileUtils.py module, 454–463
find( ) method (Python), 73
flat binary files, 464
flat files, 442
floor division operator (Python), 78
flowcharts, 266
for loop (C language), 148
for statement (Python), 89
formal use case, 261
frames, 221
frequency, 19, 30
frequency domain, 313
frequency spectrum analyzer (FSA), 313
Frolov, Vyacheslav, 236
FSA (frequency spectrum analyzer), 313
full-duplex, 53, 219
function blocking, 421
function return values and tuples (Python), 115
functional requirements, 257
functional testing, 274
pass versus fail results over time, 279
regression testing, 278
test cases, 274
testing error handling, 277
tracking progress, 279
functions (C language), 156
functions, defining (Python), 101

G

garbage collection (Python), 65, 73
gating conditions, 332
gcc compiler, 123
generic digital oscilloscope, 198
generic test case template, 276
generic wrappers (Python extensions), 178
geometry management, 530
geometry of images, 476
get( ) method (Python), 76
GetData( ) function, 424
getFeedback( ) method, 566
global module variables, 171
global scope (Python), 99
global statement (Python), 87
global variables (Python), 115
gnuplot, 383–391
gnuplot.ini file, 387
Microsoft Windows, installation on, 384
simulator data, plotting with, 388–391
using, 385
via gnuplot.py package, 387
via popen( ) method (Python), 386
goto statement (C language), 149
GPIB (General Purpose Interface Bus), 10, 237
connections, 239
PCI GPIB cards, 244
signals, 238
via USB, 239
gptest.py, 386
grid method, 531

GUI object inheritance, 527
GUIs (Graphical User Interfaces), 524–529
  display structure, 527
  functionality, 528
  history and concepts, 524
  main loop, 529
  object-oriented character, 527
  Python-based GUIs, 526–529
  TkInter implementation, 529–535
    geometry management, 530
    planning, 530
    TkInter example, 531–535
    tools and resources, 535
  wxPython implementation, 535–543
    building a GUI, 536
    designing a GUI, 536
    tools and resources, 542

H

half-duplex, 53, 219
hand tools, 190
handshaking, 55
header files (C language), 159
help( ) command (Python), 62
help( ) function (Python), 66
help(str) command (Python), 73
henries (H), 36
Hertz (Hz), 19
hex wrenches or keys, 197
hexadecimal notation, 66
Hood, Jason, 501
horizontal sweep, 199
hybrid control systems, 340
hysteresis, 327

I

I/O handler, 398
I/O transaction, 422
identity operators (Python), 82
IDEs (integrated development environments), 119
  for Python, 117
IEEE-488 (see GPIB)
if statement (Python), 88
if-else statement (C language), 143
image data, 476–485
  image formats, 477
ImageJ tool, 480
import statement (Python)
  cyclic imports, 108
  methods, 106
  processing, 107
#include directive, 129, 171
#include statements, 127
indentation and Python program structure, 84
index( ) method (Python), 71
inductors, 36
INI files, 449
initialization, 171
INITiate command, 404
input-output relationships, 311
InputUtils.py, 288
InputUtils2.py, 297
instrument interface components, 398
instrumentation, 1
instrumentation systems, xv
integer (Python object type), 187
integration requirements, 258
intensity maps, 476
interface faults, 352
interface formats and protocols, 396–406
  IVI standards, 398
  SCPI standard, 401
  unique protocols, 404
    command and response formats, 405
  VISA standard, 401
interface support packages (Python), 406–412
interfaces, 39–55
interpreted programming languages, 61
ISA bus, 397
items( ) method (Python), 76
IVI (interchangeable virtual instrument), 398
  architecture overview, 400
  IVI-compliant drivers, 400

J

Jacobson, Ivar, 260
join( ) thread object method, 425
joules (J), 20

K

Kapton tape, 203
key/value pair (KVP) organization, 449
keys( ) method (Python), 77
keywords (Python), 85

L

LabJack U3 device, 560
  connections, 560
  driver structure, 563
  installing LabJackPython, 562
  LabJackPython.py module, 563
  Linux installation, 562
  Python interface, usage with the getFeedback( ) method, 566
  U3 class (Python), 565
  U3 configuration, 565
  U3 interface class derivation, 563
  u3.py module, 563
  u3ledblink.py, 568
  Windows installation, 562
laboratory instrumentation, 11
ladder diagrams, 332
Laplace transform, 312
Leyden jar, 32
libraries (C language), 125, 161
libvisa.so.7, 411
Liechti, Chris, 406
lightweight process, 361
LIM command (SPC), 377
line termination in different operating systems, 439
linear control systems, 305
linear proportional controller, Python implementation, 340
lineman's pliers, 197
lines, 40
linker (C language), 161
Linux
  bus-based hardware I/O devices, using with, 412
  comedi drivers, 412
  console interfaces, 487
lists (Python), 68, 69
ljust( ) method (Python), 73
local scope (Python), 97
local variables (C language), 127
logic analyzers, 199
logical operators (C language), 137
logical operators (Python), 79
long (Python object type), 187
Loper, Edward, 297
luminance maps, 476

M

macros, 128
main( ) function (C language), 125
mainloop( ) method (Python), 529
make utility (C language), 162
map( ) method (Python), 481
mapped objects (Python), 75
mask, 183
MDI (multiple document interface), 525
MEASure command, 403
membership operators (Python), 82
memset( ) library function (C language), 127
message sequence charts (MSCs), 268
method flags (Python extensions), 172
method table (Python extensions), 171, 172
methods, 60
methods, defining (Python), 101
METH_KEYWORDS, 173
METH_NOARGS, 174
METH_VARARGS, 173
Microsoft Visual Studio, 123
Microsoft Windows
  console interfaces, 487
  curses library module, absence on, 515
  electronics hardware and, 168
  Tera Term, 380
  VSPs (virtual serial ports), 235
MIL-type circular connector, 212
MIL-type crimp pin, 216
millihenries (mH), 36
MinGW (Minimalist GNU for Windows), 124
miniature socket sets, 197
minicom, 381
models, 350
module global variables (Python), 115
modules (Python), 60, 101
  considered as objects, 116
  creating, 96–105
  importing, 106–108
MSCs (message sequence charts), 268
multidrop configuration, 227
multifunction DAQ cards, 244

N

namespaces (Python), 66
  scope and, 96–101
NAND (Not-AND) gate logic, 24
Netpbm image format family, 477

NewID( ) function (Python), 116
noise (in simulations), 371
noisy data, 434
nonblocking function calls, 421
None (Python object type), 187
nonlinear control systems, 306
nonlinear pulse control, 307
nonprintable characters, risks of using, 439
null-modem cable, 224
numeric data objects (Python), 66
nut drivers, 197

O

object files (C language), 160
object types (Python), 64
object-oriented programming, 60
octal notation, 66
ohms, 20
Ohm's law, 25
oledll interface class, 184
Omega D1112 RS-485 data transmitter, 553
Omega Engineering, 553
on-off controllers, 326
open( ) method, 113
open-collector output, 43
open-loop control systems, 5, 319
operators (C language), 134–140
  operator precedence, 140–142
operators (Python), 78–83
  operator precedence, 83
OR operator (C language), 139
orbital shells, 16
ord( ) function (Python), 440
oscilloscopes, 198
  bandwidth, 199
overstuffed toolkit, 190

P

packages (Python), 101
pack_struct_file.py, 475
padding, 465
panel-mount connectors, 209
parallel interfaces, 54
pass statement (Python), 87
PC bus interface hardware, 241–244
PCB-mount terminal block, 214
PCBs (printed circuit boards), 208
PCI (Peripheral Component Interconnect) bus, 242
PCI DAQ (Data Acquisition) cards, 244
PCI Express bus, 242
PCI GPIB interface cards, 244
PDevAPI.c, 179
PEP-257, Docstring Conventions, 281
PEP-8 coding style, 117
PEP-8, Style Guide for Python Code, 281
performance requirements, 257
PGM (Portable Grayscale Map) image format, 478
  file structure, 478
pgmtest.pgm, 480
pgmtest.py, 479
PGMWrite( ) function (Python), 484
physical interfaces, 207
  older interfaces, 245
PID (proportional-integral-derivative) controllers, 334–339
  Python, implementation in, 342–346
PID control block diagram, 335
pins, 40
pipes, 383
pixel data size, 477
place geometry manager, 531
planning, 249
  project objectives, 253
  projects, defining, 250
  requirements, 253–265
    capturing requirements, 264
    characteristics of well-formed requirements, 256
    requirement types, 257
    software requirements in development flow, 257
    traceability, 261
    use cases, 258
  requirements-driven design, 251
  statement of need, 252
plant, 2, 319
PLC (programmable logic controller), 332
plug-in circuit boards, 396
pointers (C language), 151–153
popen( ) method (Python), 386
ports, 40
POW command (SPC), 375
power, 20
preprocessor directives, 127, 128–132

print statement (Python), 87, 114, 445
procedural programming, 60
process, 361
process control, 13
programmable logic controller (PLC), 332
proportional control term, 337
proportional controller droop, 338
proportional term control response, 337
pseudocode, 270
pull-up and pull-down resistors, 43
pulse-width modulation (PWM), 50
PWM (pulse-width modulation), 50
PyArg_Parse( ) method, 174
PyArg_ParseTuple type codes, 174
PyArg_ParseTuple( ) method, 173, 174
PyArg_ParseTupleAndKeywords( ) method, 173, 174
PyArg_UnpackTuple( ) method, 174
PyArg_VaParse( ) method, 174
.pyd extension, 168
pyParallel package, 410
pySerial package, 406–409
PySims package, 357
Python extensions, 167
  (see also C programming language)
  C extension API, 169–184
    API types and functions, 171
    extension source module organization, 169
    generic discrete I/O API, 175–177
    generic wrapper example, 178–181
    method flags, 172–174
    method table, 172
    passing data, 174
  calling extensions, 181–184
    error checking, 182
  creating, 168
  ctypes library (see ctypes library)
  hierarchy, 168
Python manpage, 63
Python programming language, xv, 59
  command line, 61
  curses library (see curses library)
  data types, 65
    ctypes data types, compared to, 186
  input and output, 110–114
    command-line parameters, 112
    console output using print, 114
    files, 113
    redirecting print output, 114
  installing, 60
  interface support packages, 406–412
  loading and running programs, 108
  objects, 64
  Python extensions (see Python extensions)
  simulators, creating in (see Python simulators)
  text editors and IDEs, 117
  unittest facility, 289
Python simulators, 356–380
  AC power controller simulator, 371–380
  data I/O simulator, 357–371
  packages and modules, 356
PythonCard, 543
PyUniversalLibrary, 412
PyVISA package, 411
Py_BuildValue( ) method, 174

Q

quantization and quantization error, 45
quoting of string literals (Python), 91

R

range( ) function (Python), 89
rapidly changing average data, 434
raw_input( ) function (Python), 110
RDD (requirements-driven design), 251
read-modify-write, 183
read183dmm.py, 551
readData( ) method, 368
readinto( ) file object method, 469
records, 442
reference input, 311
regression testing, 278
relay driver circuit, 43
replace( ) method (Python), 73
requirement and test case relationship, 274
requirements traceability matrix, 263
requirements-driven design (RDD), 251
reserved keywords in Python, 85
reset, 339
resistance, 20, 25
resistors, 27
resolution of data, 45
RetCodes module, 356
return statement (Python), 87
rjust( ) method (Python), 73

Roundup tool, 300
RS-232 and RS-485 interfaces, 218
RS-232 DB-9 pin definitions, 222
RS-232 interface, 53, 219
  data formats, 220
  signals, 222
  versus RS-485, 229
RS-485, 405
RS-485 command-response sequence, 228
RS-485 half-duplex data flow and operation, 228
RS-485 interface, 225
RS-485 interface drivers in two-wire and four-wire modes, 226
RS-485 signals, 226
RST command (SPC), 377

S

ScaledInput( ) test cases, 287
schematic wiring notation, 22
schematics, 20
Schleef, David, 412
scope (Python), 96–101
SCPI (Standard Commands for Programmable Instruments), 401
SDD (software design description), 250, 265–269
  block diagrams, 266
  failure analysis, 273
  failure responses, 273
  flowcharts, 266
  functional testing (see functional testing)
  graphics, usage in, 266
  handling errors and faults, 272
  message sequence charts, 268
  organization, 270
  pseudocode, 270
  state diagrams, 268
  unit testing, mapping to, 286
SDLC (software development life cycle), 251
SEM command (SPC), 376
SendTrigger( ) function, 424
SEQ command (SPC), 376
sequence objects (Python), 68
sequential control systems, 5, 9, 308, 330–332
  control system states, 331
  SFCs, 332
serial interfaces, 51, 207, 218, 547–553
  discrete and analog data I/O devices, 553–558
  speed considerations, 559
  tpi model 183 DMM, data capture from, 548–553
serial terminal emulators, 380–383
  terminal emulator scripting, 381
series and parallel resistance, 29
set and check instrument output, 420
setDataFile( ) method, 368
setInputSrc( ) and setOutputSrc( ) methods, 367
SFCs (sequential function charts), 332
signal switchers, 403
SimLib package, 356
simple LED circuit, 26
Simple Power Controller (see SPC)
SimpleANSI library, 505–515
  using, 512
SimpleANSI library module, 505
simulation, 349–350
  fidelity, 351
  Python simulators, creating (see Python simulators)
  simulating errors and faults, 352
  versus the real world, 350
simulation package structure, 356
simulator cyclic data types, 363
simulator fault injection using object variables, 355
simulators, 349
  creating, 391
    evaluating needs, 392
    scope of simulation, 392
    time and effort, 393
  data I/O simulator (see DevSim)
  serial terminal emulators, 380
  simulation data, displaying, 383–391
    gnuplot, using (see gnuplot)
sine waves, 30
sine_print.c program, 126
sine_print2.c, 131
single-trace oscilloscopes, 198
size of digital data, 3
slope intercept equation, 305
software bang-bang controller, 328
software design description (SDD) (see SDD)
software design process, 265–273
  software design description (see SDD)

software development
coding (see coding)
software development life cycle (SDLC), 251
solder cup connection, 215
soldering, 195, 215
soldering guns versus soldering irons, 197
solid-state components, 21
SOR command (SPC), 376
SOW (statement of work), 250
SPC (Simple Power Controller), 372
command set, 374–377
communication with, 374
configuring, 379
interacting with, 379
serial interface and virtual serial ports, 372
simulator internals, 377
SPC command-response MSC, 378
special-purpose interfaces, 219
sprinkler system sequential control, 308
square wave, 39
standard library (C language), 158
state diagrams, 268
statement of need, 250
statement of work (SOW), 250
statements (C language), 143–149
statements (Python), 84–91
blocks and indentation, 84
compound statements, 87
simple statements, 86
static global variables (C language), 124
stdin, 110
stdout, 114
step input, 337
STM command (SPC), 376
stranded versus single-conductor wire, 203
string (Python object type), 187
string indexing (Python), 72
string nulls (C language), 128
strings (Python), 68, 71, 91–96
commonly used string methods, 92
string escape sequences, 95
string format placeholder flags, 94
string formatting, 93
string methods, 91
string quotes, 91
Struct class (Python), 471
struct data type format codes, 473
struct functions, 471
struct module (Python)
16-bit image data, building with, 481
Structure class (ctypes), 467
structure padding, 465
structured programming, 60
structures (C language), 153–156
SubVersion, 299
summing junction or node, 7
summing node or summing junction, 312
supplies, 203
SWIG (Simplified Wrapper and Interface Generator), 414
SWIG tool, 169
switch statement (C language), 145–147
symbols in circuit schematics, 20
synchronous serial data communication, 51
system faults, 352, 354
system time constant, 317
system-level fault injection, 354

T

tachometer continuity sense, 34
Telecommunications Industry Association (TIA), 218
Tera Term, 380
TTL (Tera Term Language) scripting language, 382
Teranashi, T., 380
termcap (Unix), 500
terminal blocks, 213
terminfo (Linux), 500
terms, 335
ternary conditional (?:), 130
test cases, 274
test procedures, 275
text data or text files, 438
text editors, 117
text-based interfaces, 487–524
ANSI display control techniques, 500
basic ANSI control sequences, 501
SimpleANSI library, 505–515
Windows and, 501
consoles, 487–500
curses, 515
textual data display, 491
thread, 361
time and frequency domain graphs, 313
time domain, 313
time-invariant and time-variant systems, 315
timers, 49

tkdemo1.py, 531
tkdemo2.py, 533
tokens, 127
tools, 189–202
  assessing quality, 192
  electronic tool sources, 192
  logic analyzers, 199
  new versus used tools, 204
  proper use, 191
  proper use of test equipment, 202
  soldering tools, 195
tpi 183, 404
tpi model 183 data output format, 549
tpi model 183 DMM, 548
traceability, 261
transfer functions, 312
transistor-transistor logic (TTL) voltage, 42
try statement (Python), 90
tstamp, 445
ttyOtty package, 373
tuples (Python), 68, 74
  returning function values with, 115
typedef (C language), 133

U

UART (Universal Asynchronous Receive-Transmitter) devices, 53
unary operators (C language), 134
unbounded loops, 148
UNC Python Tools package, 412
unit testing, 286–295
  code coverage analysis, 293
  definition, 286
  implementing, 288–291
  mapping to the SDD, 286
  Python unittest facility, 289
  Python unittest library
    assert-type methods, 291
unittest library (Python), 289
upper( ) method (Python), 73
USB (Universal Serial Bus) interface, 218, 231–234
  LabJack U3, 560–569
    installation, 562
    LabJack connections, 560
    LabJackPython interface, 562
USB connectors, 210
USB interface modules, 13
USB logic analyzer modules, 200
USB-to-GPIB interface, 239
USB-to-serial converters, 236
use cases, 258
user input (Python), 110
user interfaces, 487
  text-based interfaces (see text-based interfaces)
user level, 398
user-defined functions, 368
UTF-8 character encoding, 438
UUT (unit under test), 9

V

valence shell, 16
value tolerance checking, 419
values( ) method (Python), 77
variables (Python), 65
VDT (Video Display Terminal), 500
version control, 299
vintage Leeds and Northrup resistance bridge test set, 195
virtual ports, 372
VISA (Virtual Instrument Software Architecture), 401
visa32.dll, 401, 411
voltage and volts (V), 20
voltage dividers, 29
VSPs (virtual serial ports), 235

W

warmup convergence, 433
waterfall model of software life-cycle, 251
watt (W), 20
waveforms, 38
  sine waves, 30
wgnuplot, limitations of, 386
while loop (C language), 148
while statement (Python), 88
widgets, 525, 526
windll interface class, 184
Windows VSPs (virtual serial ports), 235
wire, 203
wire cutters and strippers, 191
wiring, 215–218
  correct technique, 217
wrappers, 167
write( ) method (Python), 113, 445
wxexample.py, 542

wxGlade, 543

X

XOR operator (C language), 139

Z

z-transforms, 316

关于作者

John M.Hughes 是一位拥有超过 30 年工作经验的嵌入式系统工程师，工作领域涉及电气、嵌入式系统和软件、航空航天系统，以及科学应用编程。他曾经负责设计凤凰号火星着陆器的表面成像软件，商用和军用飞机的数字引擎控制，自动测试系统，射电望远镜的数据采集和实时控制等工作。Hughes 已在各种应用中使用 Python 很多年了，包括詹姆斯·韦伯太空望远镜中的多波长激光干涉仪系统的对准检验软件。目前，他正在亚利桑那大学，在成像系统的仿真和分析方面继续使用 Python。

封面介绍

本书封面上的动物是冠鸦（冠小嘴乌鸦），也被称为苏格兰乌鸦、丹麦乌鸦、灰鸦或渡鸦，广泛分布在欧洲和中东。因为冠鸦同常见的吃腐肉的乌鸦如此相近，以前常被认为是同一品种。然而，截至 2002 年，冠鸦已成为完全的种，并有 4 个公认的亚种。

冠鸦的羽毛主要是灰白色的，它的翅膀、尾巴，尤其是在它的头部和喉咙外罩着光泽的黑色羽毛，从而被命名为冠鸦。冠鸦成年时，平均翼展为 98 厘米，身长从 48 至 52 厘米不等。冠鸦同相近的乌鸦类似，也喜欢吃腐肉，是一种杂食性的清道夫。据了解，冠鸦会从其他鸟类的巢中偷蛋，在沿海地区，冠鸦懂得从高处丢下软体动物或是螃蟹，破碎后得以食用。

戴着头巾的乌鸦，这种形象，在传统的凯尔特民间传说中有着特殊的意义，同时也流传于苏格兰高地和爱尔兰地区。在 18 世纪，苏格兰牧羊人供养冠鸦，以防止他们攻击自己的羊群。而在丹麦的法罗群岛，圣烛节的早上，少女们会观察冠鸦的飞行轨迹，以明确未来丈夫来自何方。

封面图片取自约翰逊的 Natural History。