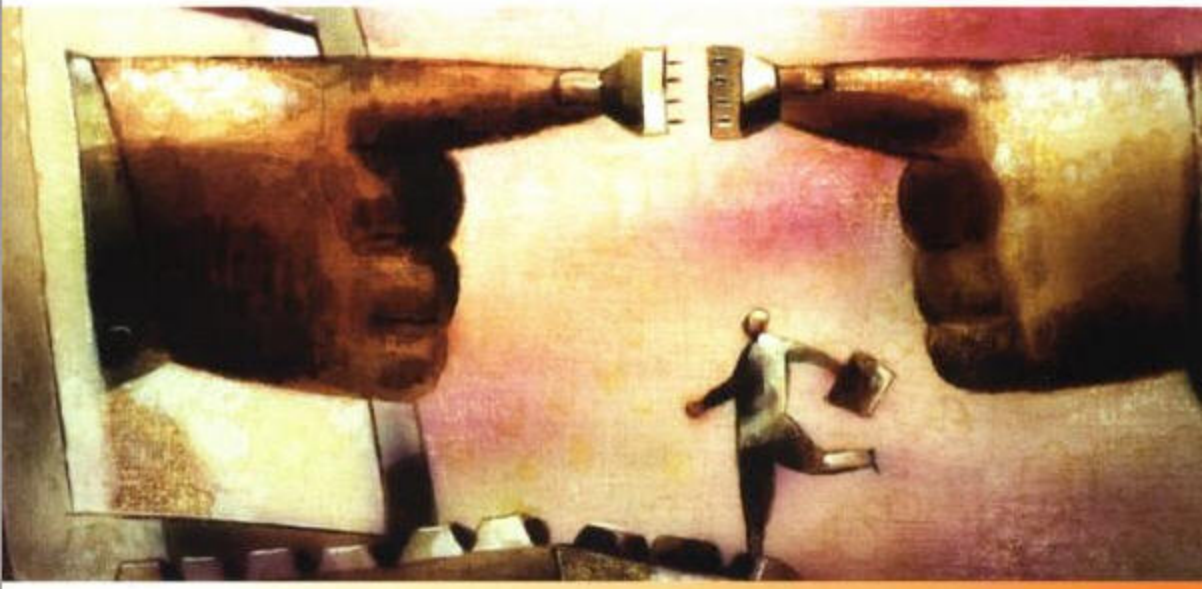
"*About Face 2.0* could completely redefine how software applications are created!"
— Pete McBreen, author, *Software Craftmanship*

ABOUT FACE 2.0

THE ESSENTIALS OF INTERACTION DESIGN



ALAN COOPER
& ROBERT REIMANN

International Bestseller
Completely revised and updated

Copyrighted Material

内 容 简 介

本书是一本设计数字化产品的启蒙书，它在帮助您设计更有吸引力、更有效的对话框的同时，还将帮助您理解用户如何了解、学习您的软件，以及与之交互的方式。本书着重讲述了有关交互设计的原理和方法：第一篇强调设计过程，以及对用户的系统理解；第二篇提供了策略原理和工具；第三篇更深地钻研了战术性的问题。

本书探索了一个独特的设计领域，即复杂系统行为的设计。本书论述一种具有革命意义的设计观念——目标导向设计过程。其作者 Alan Cooper 是一位在交互设计前沿有着 10 年设计咨询经验及 25 年计算机工业界经验的卓越权威。

本书是一本难得的大师经典之作，是一本数字产品规划师、项目经理、设计师、可用性从业人员，以及程序员都想得到的书——这是一本使得我们的软件和我们的世界变得更美好的书！

Copyright © 2003 Alan Cooper.

All rights reserved. Authorized translation from the English language edition published by John Wiley & Sons, Inc.

本书简体中文专有翻译出版权由 John Wiley & Sons Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2003-1539

图书在版编目（CIP）数据

软件观念革命：交互设计精髓 /（美）库珀（Cooper, A.）等著；詹剑锋等译. —北京：电子工业出版社，2005.6

书名原文：ABOUT FACE 2.0：The Essentials of Interaction Design

ISBN 7-121-01180-8

Ⅰ. 软…　Ⅱ. ①库…②詹…　Ⅲ. 软件设计　Ⅳ. TP311.5

中国版本图书馆 CIP 数据核字（2005）第 042940 号

责任编辑：孙学瑛　朱沭红

印　　刷：北京智力达印刷有限公司

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

经　　销：各地新华书店

开　　本：787×980　1/16　印张：37.25　字数：416 千字

印　　次：2005 年 6 月第 1 次印刷

印　　数：5000 册　　定价：89.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。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。

献给 Sue，22 年，你始终如一，是我最好的朋友。

献给 Julie，因为你的爱、仁慈，以及长久的鼓励。

献给 Cooper 公司所有的职员，无论是过去的，还是现在的，抑或是将来的。

献给那些有太多梦想的设计从业人员，你们一起努力创造了这个设计行业。

对本书的溢美之词

本书是构筑在 Alan Cooper 那令人崇敬的一流能力之上的。它聚集了当今最先进的交互设计观念，内容涉猎广泛，信息量充足，并富有洞察力。

Jesse James Garret, “The Elements of User Experience”一书的作者

通过对外部设计进行严格的规范，“目标驱动设计”让程序员把精力集中在他们最擅长的方面——开发能够实现设计规范的软件实体。目标驱动设计允许用户体验设计师创建程序员能够实际使用的外部设计规范。

Pete McBreen, “software craftsmanship”一书的作者

Alan Cooper 和 Robert Reimann 具有非常独特的能力，阐明了一些一般难以掌握的概念。本书充满着这样的高见。第二版的出版再次让读者惊叹：“是的，就是它!”

James Fawcette, Visual Studio Magazine 和 VSLive! Conferences 的创建者

在 Alan Cooper 的畅销书“The Inmates are Running the Asylum”中，他将利箭射向了软件行业。他谴责在当今软件的工作方式和设计中存在一些不可容忍的缺陷。在本书中，Alan Cooper 和 Robert Reimann 阐述了应该如何将软件做得更好。他们对设计方法和交互原则（interaction principle）进行了透彻而系统的讨论，为设计者提供了最佳指导。借此，设计者可以获得真正为用户工作的数字产品。

Terry Winograd, 斯坦福大学计算机科学教授，“Bringing Design to Software”的编辑

对那些有兴趣创建和使用人物角色的设计者来说，本书尤其及时。人物角色是我所见到的最引人注目的设计工具，它将“用户”作为一个抽象的概念，并将其转换为设计和业务交流的具体工具。在本书中，这一强大实践的开发者会告诉您，它是如何做到的。

Harley Manning, Forrester 研究院研究主管

作 者 介 绍

Alan Cooper 是软件开发者、程序员、程序设计师和理论家的先驱。他，众所周知，Visual Basic 之父，被认为可能是第一个为微型计算机编写了严格意义上的商业软件的人。近十年来，他的软件设计咨询公司——Cooper，已经帮助众多的公司开发出新的软件并且改善他们的技术。在 Cooper 公司，Alan 领导制定了一种新的成功开发软件的方法学——目标驱动软件过程（goal-directed process），另外，他还创造了人物角色（persona）技术，自从他 1998 年在他的第二本书“The Inmates Are Running the Asylum”中公开这种技术之后，已经广被采用。Cooper 也是非常有名的人性技术方面（humanizing technology）的作家、讲演者和布道者。

Robert Reimann 在过去 15 年一直作为设计师、作家、演讲者和咨询师来推动数字化产品的发展。他在电子商务、门户站点、桌面产品、创作环境、医疗和科技设备、无线和手持设备等领域领导了很多个交互设计项目，主要面向新兴企业和福布斯 500 强这样的客户。在 1996 年加入 Cooper 公司后，Reimann 领导了本书中描述的很多目标驱动设计方法的开发和优化工作。他在多所专业院校中，以及向国际上的同行们宣讲这些方法。他也是 UC Berkeley 设计学院顾问委员会的成员。

致　谢

作者谨向以下个人致以衷心的感谢，没有他们就不可能有新版的《软件观念革命》：Sue Cooper，Cooper的执行总裁，他为本书花费了大量时间；Wiley出版公司的Chris Webb，他参与了本书的制作；Pat Fleck，是他为本项目设法融资；Sara Shlaer，我们的项目编辑；Mary Lagu，Wiley出版公司的稿件编辑——他们自始至终和我们一起愉快地工作，在思想表达和提炼语言方面给我们以很大的帮助。

我们还想感谢以下同事和Cooper的设计师们，对他们为本书所做的贡献致以诚挚的谢意：Kim Goodwin在过去5年一直与作者合作，一起完善并提炼出本书第一部分叙述的许多概念和过程；Hugh Dubberly，在他的帮助下完善了第7章末尾提到的原则，也是在他的帮助下用几张漂亮的图表清楚地阐明了第1章中的目标导向过程；Gretchen Anderson, Elaine Brechin和Doug LeMoine，感谢他们对第4章中用户和市场调查做出的卓越贡献；SAP公司的Ernest Kinsolving和Joerg Beringer，感谢他们对第8章的Web门户姿态方面做出的卓越贡献；Wayne greenwood，感谢他对第11章中的控件映射方面做出的卓越贡献；Nate Fortin，感谢他在第19章中的视觉品牌和视觉交互设计方面做出的卓越贡献；Jonathan Korman，在跋中有一些思想来源于他即将出版的著作“Best to Market”；Dave Cronin，感谢他在公用信息亭和Web设计方面的许多绝好想法；以及Chris Weeldreyer，感谢他在嵌入式系统设计方面的许多真知灼见。我们还想感谢Elizabeth Bacon, Steve Calde, John Dunning, Kim Goodwin, Wayne Greenwood, Lane Halley, Berm Lee, Ryan Olshavsky, Angela Quail和Chris Weeldreyer，感谢他们在本书中使用的Cooper设计和示例中所做的贡献。

我们还要感谢客户共享健康系统的David West和Fujitsu Softek公司的Mike Kay和Bill Chang。感谢他们允许我们在本书中使用来自Cooper设计项目的例子。我们还要感谢那些有远见和魄力与我们合作，并支持我们的所有客户。

最后，我们还要感谢以下作者和同行，多年来在他们的影响下才形成了我们的思想：Christopher Alexander, Edward Tufte, Kevin Mullet, Victor Papanek, Donald Norman, Larry Constantine, Challis Hodge, Shelley Evenson, Clifford Nass, Byron Reeves, Stephen Pinker和Terry Swack。

内容快览

第一篇　了解你的用户

第一部分　弥合差距

1　目标导向设计……5
2　实现模型和心智模型……24
3　新手、专家和中间用户……35
4　理解用户：定性研究……42
5　用户建模：人物角色和目标……59
6　脚本提纲：将目标转换为设计……81
7　综合好的设计：原则和模式……98

第二篇　设计行为与形式

第二部分　去除障碍，达到目标

8　软件姿态……111
9　和谐与流……129
10　消除附加工作……146
11　导航和调整……155
12　理解撤销……171
13　重新思考“Files”和“Save”……182

第三部分　提供高效能和愉悦

14　设计体贴的软件……199
15　设计智能的软件……209
16　改进数据检索……218
17　改进数据输入……227

18 为不同的需要进行设计 …… 234

第四部分 应用视觉设计原则

19 外观设计 …… 245
20 隐喻、习惯用法和启示 …… 267

第三篇 交互细节

第五部分 鼠标和操作

21 直接操作和定点设备 …… 285
22 选择 …… 305
23 拖放 …… 313
24 操作控件、对象和连接 …… 326

第六部分 控件及其行为

25 窗口行为 …… 345
26 使用控件 …… 361
27 菜单：教学向量 …… 389
28 使用菜单 …… 397
29 使用工具条和工具提示 …… 408
30 使用对话框 …… 420
31 对话框礼节 …… 435
32 创建更好的控件 …… 450

第七部分 与用户的交流

33 消除错误 …… 461
34 通知和确认 …… 471
35 与用户的其他交流方式 …… 482

36　安装过程 …… 491

第八部分　超越桌面的设计

37　Web 设计 …… 503
38　嵌入式系统的设计 …… 514
附录 A　本书公理集 …… 525
附录 B　本书设计技巧集 …… 529
跋：给同行的话 …… 533

导　读

叶展，清华大学自动化系本科毕业，后赴美留学，先后取得伊州理工学院（Illinois Institute of Technology）的计算机硕士学位和卡耐基·美隆大学（Carnegie Mellon University）的人机交互（Human-Computer Interaction）硕士学位，现在美国 BCS 管理和 IT 咨询顾问公司任职。叶展目前主要的研究和工作领域是人机交互理论在游戏设计中的应用，人机界面设计与评测，以及软件开发流程设计和管理，他是这些领域有一定影响的专家，并应邀在包括 CHI 等一系列重要国际会议上发表了论文和演讲。

读经典著作如同饮醇酒，回味隽永。而给经典著作写序，则如推销醇酒与人，在别人的沉醉中分享快乐。

摆在我面前的就是一本经典著作，一本计算机领域的经典著作。众所周知，计算机领域多的是应景之作，比如某某软件版本 X.0 的使用指南之类，而少能经得起时间考验的经典著作。其原因一方面是计算机领域发展迅猛，知识更新代谢极快；另一方面，则是计算机领域应用重于理论，所以有思想深度的著作比较少。

一本书要成为经典，起码要两个条件。其一是著者拥有深邃的思想，且文笔流畅。大师级人物，往往单从文字上的修养就看得出。比如经济界的 George Stigler（诺贝尔奖获得者，被誉为经济学界一支笔），以及软件工程方面的 Frederick Brooks（《神秘的人月》著者），他们的著作拿起来阅读几段，你马上就可以在行文中感到那种大家的雍容风度。而更重要的是，大师们在书中所讲述的往往不是细枝末节的技巧和技术，而是一种深邃的思想方法，可以给人以深层次的启发。那种风度和深度，是难于模仿的。

要成为经典的第二个条件是，作者的思想要经得起时间的考验。对计算机书籍来说，起码要能经历 10 年的考验。这个标准比之其他领域已经是很宽松的了，但在这个标准下大部分计算机书籍会落马。

以笔者来看，本书基本上符合前面的两个条件，是一本计算机领域的经典之作。

本书的作者 Alan Cooper，是计算机业界一位成名高手。除了早期在 Visual Basic 方面的工作外（他被誉为 Visual Basic 之父），更重要的是他曾站到了一个新领域的前沿，参与并影响了软件开发领域一次深刻的变革。而这个新领域，就是人机交互

（Human-Computer Interaction）。这个变革，是软件开发领域的第三次革命。

在软件开发领域出现过三次革命——20 世纪 50 年代高级语言的出现，使得软件开发从机器硬件（机器语言）的束缚中解脱出来，程序员能够从（抽象＋结构）层次来进行思考。此为第一次革命。20 世纪 70 年代软件工程兴起，使得软件开发的注意力由语言和编译器技术拓展到软件开发的过程（software process）。人们意识到：要提高一个软件产品最终的静态质量，必须提高这个产品产生过程的动态质量。此为第二次革命。

而 20 世纪 90 年代以来，随着计算机软件和商业行为的联系越来越紧密，特别是因特网的兴起，人们进一步认识到：软件不是孤立的，软件的质量并不是仅由其本身就能决定的，而是由（软件＋用户）这个大系统来决定的。软件的成功，在于是否它能够成功嵌入到用户的商业活动中。对人的因素的重视，使得一个新的领域崛起。这就是人机交互。经过十几年发展，人机交互理论已经全面改观了一般商用软件设计开发的流程和方式，成为业界的标准。此为第三次革命。

每次革命或变革，都会有豪杰之士涌现，为改变旧的思想和宣传一种新的思想而摇旗呐喊，成为领导变革的预言家和代言人。在 20 世纪 90 年代，一批人物涌现，一批著作发表，为人机交互理论在业界的应用打开了局面。Donald Norman 在 1990 年出版了“The Design of Everyday Things”，Jakob Nielsen 在 1994 年出版了“Usability Engineering”，本书作者在 1995 年出版了本书的第一版。这些人的著作，都经受了十年的考验，现在都成了经典。他们当时的思想，现在已经成为业界的主流。他们也自然而然地成为了人机交互领域举足轻重的领导者。

阅读本书，最重要的是了解作者所阐述的关于软件设计开发的高层次理念和指导思想。因为作者是最新一次软件开发思想变革的积极参与者，他亲自现身说法写的书理当是记录这个思想变革的宝贵的第一手资料。正因为如此，笔者窃以为本书的第一部分最为重要，乃为全书的灵魂。这部分从了解用户，了解用户需求讲起，到构建用户模型，到设计场景来描述软件系统现在和未来的行为模式，到如何把对用户的理解和行为模式转换为设计方案。作者不仅把软件设计的整个过程流畅清晰地描述出来，而且真知灼见不断涌现于其中。下面随便列举一二。

- **软件的设计和开发，不要囫囵吞在一块，最好要分成两个单独的过程——设计过程和开发过程**：传统的软件工程理论，是对整个软件设计开发的过程化研究，而更侧重编程测试和项目规划部分，并且把设计和开发混在一起。而现在人机

交互理论，实际上是把软件设计这部分提出来，是对软件设计的过程化分析，还借用了认知心理学和其他领域的成果。目前业界普遍认为：对商用软件来说，这两个阶段分开，有助于软件质量的提高。

- **应当以用户为中心去设计软件，而不是以某项新技术或者技术人员为中心去开发软件**：这一点是软件走出象牙塔，渗入人类生活和商业领域的必然后果。作者虽然是程序员出身，但对以程序员（技术的代表）为核心的软件开发的局限性有清醒的认识，并指出这种方法再也不能适应开发软件产品的需要。基于用户的设计（User-Centered Design）是20世纪90年代以后被“炒”得最火的一个词。它实际上是说在软件设计过程中要围绕用户和商业活动来进行，是不是围绕技术和程序员来运行。
- 决定软件成功与否的不是一个软件有多少个功能，而是这些功能是否有用和好用。
- **设计软件，重要的是设计用户行为**：作者所极力鼓吹的一个新的名词——交互设计（Interaction Design）的含义就是软件设计师设计的不是死的软件，不是静止的界面，而是活的行为，是用户和软件硬件环境之间的动态交互，并寻求动态的最优。

需要指出的是：以上思想，在1995年本书初版时乃为革命，如今则为业界常识——起码是美国商用软件开发领域人所共奉的常识。现今但凡大一点的与软件开发有关的公司，其软件设计开发过程都是按本书作者所提出的思路改进过的。从另一个角度讲，这更体现了20世纪90年代这场变革的影响之深远。

在本书的第二部分，Alan Cooper介绍了一大把新概念和新名词（众所周知，Alan Cooper在业界有卖弄新名词的“不良”嗜好。业内人最爱开的玩笑之一就是传说在某某会议上Alan Cooper又发明了一个新的英文词。当然能够在计算机业内成为这种玩笑的对象本身就说明这个人很有影响力）。这些新名词，由于是独此一家，别无分号，读者读来需要一定的辨别力。需要记住的是：虽然这些名词比较新奇，但其含义和基本思路应该是容易接受的。

与前两部分比，本书的第三部分就完全是实用性的了，有诸多实际的设计案例和讨论，而且主要是基于现有图形用户界面的格局。

由于本书的这种结构——由抽象理念入手，到具体的设计方法和案例，使得它适合各类读者阅读。软件公司的领导者可以通过前两部分了解软件行业的最新思潮，并以此为指导思想来改进自己公司内部的软件开发流程。软件开发人员，可以学习书中介绍的具体方法，也更可以从更实际的案例讨论中获得启发。对学生来说，除了学习编程等“硬”技术外，通过读书了解一下软件行业“软”的思想，拓展眼界，受益将会不浅。

国内软件行业经过多年发展已经初具规模，当然在发展的过程中也遇到诸多问题。目前的一个共识是中国软件业和外国比，最大的劣势并不是在具体某项技术或者编程方面。中国的勤奋而又有天分的程序员，可以获得美国业界的编程大奖。由此可见单打独斗中国人是可以的。但项目一大起来，中国软件业的固有劣势就显现出来了。中国的传统弱项主要是在软件工程和软件过程等方面。而现在西方软件行业又进了一步，在软件工程的基础上搞出了人机交互理论，又引发了一次革命。

我们目前对这场革命的了解还是很肤浅，人机交互领域在国内的科研、教学和应用都还在起步阶段。这就很有些旧的差距没有弥补上，而新的差距又产生了的危险！这本书现在被介绍到国内来，将有助于我们填补这方面的差距。此其时也！

叶　展

人机交互分析师

2005 年 3 月于美国芝加哥

作者专门为本书中文版所做的序

PREFACE TO THE CHINESE LANGUAGE EDITION

When we were contacted to provide a new preface to the Chinese language edition of About Face 2.0, we were thrilled at the enhance of reaching a vast new audience of students, designers, developers, and human factors specialists.

In recent years, China has become not only a major manufacturer of software-enabled digital products, but has also become increasingly significant consumer of these products, from computers to cellular telephones to home and personal entertainment systems. This means that millions of new users in China are being exposed to software applications and digital products, many perhaps for the first time. There is thus a unique opportunity for user interface and interaction designers to dramatically improve the quality and desirability of a whole new generation of software-enabled products that will affect the lives of hundreds of millions of people.

As in the West, it has taken some time for Chinese developers and manufacturers to understand the critical importance of user interface and user interaction to making digital products useful and successful. Beyond improving the quality of the lives of their users, products that are easier to use and understand and which better meet user needs, have an additional potential for increasing profits and market share for the company which creates them. We hope that this book will lead the way to significantly better digital products in the marketplace, and also to increased opportunities for those students, designers, programmers, and manufacturers who heed its words.

Interaction Design is a new field of design in which behavior, not form, is the most critical element. Form must support behavior, but when users interact with a complex digital product,

it is the behavior of the product that dictates the perceived quality and user experience. products which are not software-enabled don't have complex behaviors: a hammer as a single, simple behavior, and requires no design beyond that of its form. However, a cellular phone, PDA, or digital camera has many complex behaviors, and these require careful and specialized design methods. About Face 2.0 seeks to describe the fundamentals of this new and exciting area of behavioral interaction design, which we predict will become a primary field of design in the 21st century.

Most of the examples in this book come from desktop computer applications or from the world-wide web. However, almost everything described in this book is also applicable to digital devices.

About Face 2.0 is not a collection of "cookbook" design solutions, nor does it contain a style guide of user interface standards. Rather, it provides a unique process and framework within which to design products and product behaviors that truly address the innermost needs and desires of users. Section I of the book describes this systematic process-- we call it Goal-Directed Design-- which operates on the premise that if you have a deep understanding of a product's users, as well as an understanding of their motivations for using the product (their goals), you can develop an interface that directly addresses the most important user needs. Sections II and III of the book provide both high-level and detailed design principles to help guide designers in choosing product behaviors that both fulfill user needs and remove roadblocks to user pleasure and productivity. We have used these methods worldwide over the last 13 years to create designs for hundreds of digital devices, software products, and web-based services or start-up companies and multi-billion dollar corporations alike. Since its publication in the US in 2003, **About Face 2.0 has also been adopted as a textbook in Computer Science and Design departments at dozens of universities across the US and in Europe.** The use of Personas (a powerful tool for characterizing users described in Section I), in particular, has become an almost universally-adopted design and human factors best-practice.

The future of Interaction Design is bright both in China and in the West. We hope that this book will inspire outstanding design in a new generation of Chinese digital products and services!

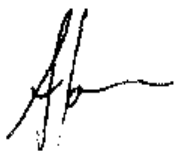
Robert Reimann

Brookline, Massachusetts, USA

Alan Cooper

Menlo Park, California, USA

January 2005



作者专门为本书中文版所做的序（译文）

为中文版所做的序

当我们被联系为该写中文版的序言时，我们非常兴奋，因为我们有机会接触一批新的而为数众多的读者，包括学生、设计人员、开发人员及人性因素专家等。

在最近几年，中国不仅成为基于软件的数字产品——从计算机、蜂窝电话到家庭和个人娱乐系统——的主要制造者，也成为这些产品的日益重要的消费者。这就意味着在中国数以百万的新用户正在使用软件和数字产品，其中不乏初学者。由此可见，对于用户界面口和交互设计师来说，这是一个独一无二的机会，他们的工作可以极大地改进这些数字产品的质量和合意性，并最终影响数以百万计的人们的生活。

与西方一样，中国的开发者和制造商经过了一段时间才理解：用户界面和用户交互对于使得数字产品更有用和更成方面所具有的极端重要性。除了改善用户的生活质量以外，容易使用和理解，并且能够更好地满足用户需求的产品有额外的潜力，为生产商增加利润和市场份额。我们希望本书将会为市场引领一条通向更好的数字产品的道路，也可以为本书的读者，不论学生、设计人员、开发人员，还是制造商，带来更多的机遇。

交互设计是设计中的一个新领域，在这里是行为，而非形式成为最为关键的因素。形式必须支持行为，但是当用户和一个复杂的数字产品交互时，用户所能感受到的产品质量和从中获得的亲身体验均来自于产品的行为。不是基于软件的产品不会有复杂的行为：锤子只有单一而简单的行为，除了样式以外不需要别的设计。然而，蜂窝电话、掌上电脑或者数码相机有很多复杂的行为，而这些行为需要仔细而专门的设计方法。本书试图描述行为和交互式设计这一崭新而又令人兴奋的领域的基本原则，我们预测这一领域将会成为21世纪设计学中的一个主要领域。

本书的大部分例子来源于桌面计算机应用和 WWW。尽管如此，书中几乎所有的内容也适用于别的数字设备。

本书并不致力于阐述所有可能的设计方法，它也不包括用户界面标准的风格指南。事实上，本书提供了一个独特的过程和框架，借助它可以设计产品和产品的行为，而这

些行为真正地解决了用户最核心的需求和意愿。

书的第一篇描述了这一系统的过程——我们称之为目标导向设计——这一部分有一个前提，就是：如果你对产品的用户有深入了解，也了解他们使用该产品的动机（他们的目的），那么你可以为最重要的用户需求开发界面。

本书第二篇和第三篇提供了高层次和细节的设计原则，主要涉及如何选择产品行为，既可以满足用户需求，又可以为用户消除障碍，这无疑会提高用户的满意度和生产率。在过去的 13 年里，我们在全世界使用了上面所说的方法为小到刚起步，大到有上百亿美金的公司的数百种数字设备，软件产品及基于 Web 的服务做了设计。

自从该书 2003 年在美国出版以来，在美国和欧洲本书已被许多大学的计算机科学和设计专业选为教材。尤其是人物角色的使用（在第一篇所描述的刻画用户的一个强有力的工具），几乎已经成为普遍采纳的设计和人性因素的最优方法。

无论在中国还是西方，交互式设计的未来都是光明的。我们希望本书能激发出中国新一代数字产品和服务中的杰出设计！

Robert Reimann

美国马萨诸塞州的布鲁克林

Alan Cooper

美国加利福尼亚州的门洛帕克

2005.1

译 者 序

今天，人类不仅在认识着世界，也在创造着新世界。软件作为人类所创造的最复杂的人工制品（artifact）之一，已不仅仅是人类智慧和工具的延伸，而在某种程度上作为虚拟世界新法则的执行者和实施者统治着我们。诺贝尔物理学奖获得者费曼曾经以这种方式描述过人类创造新事物的过程——我们创造新事物，而被创造的新事物按照某种规则又创造新的事物，突然某一瞬间，不同于人类灵魂的事物出现了：它与人类灵魂迥然不同，或许还有着恶意，威胁着人类。一个智者以这种玄想的方式展现了对人类创造物的恐惧。

今天的软件人工制品会以这种方式工作吗？是否会威胁到我们的人类？作为软件业的一名从业人员，译者深知以 0 或者 1 为工作基础的计算机所有的智慧来自于设计师和程序员的智慧，本身不具有恶意。然而，现实的情况是“受不正确的设计观念影响开发的软件已经开始威胁到大众用户”，技术派论者甚至以“计算机盲”通常称这些和计算机工作者一样富有智慧的人们。

请尊重你的用户！Alan Cooper，这位在图形用户界面领域驰骋数十年的大师给出了如此的忠告。大师的忠告是中肯而辛辣的，技术不能高高在上，而应该植根于土壤，软件工人们不能脱离为人民服务的宗旨，否则就要被革命了。新技术经济的沉沦也许指示着新的机遇：为大众用户服务，采取全新的目标导向设计方法。

这种方法关注用户的目标；认真地研究实际用户和潜在用户，定义具体的原型用户(人物角色，persona)；使用人物角色作为脚本提纲(scenarios)的主要人物；人物角色作为定义交互产品功能、行为和形式的主要工具；遵循行为设计的原理。在系统模型方面，作者精彩地辨析了程序员的实现模型(implementation model)和用户的心智模型(mental model)之间的差异，指出程序员通常为了容易实现的私利牺牲用户利益，用实现模型取代用户的心智模型，从而产生了认知方面的鸿沟，因此在用户界面领域有必要区分设计和编程的责任。在用户分析方面，将用户分为新手用户(beginner)、中间用户(intermediate)和专家用户(expert)三类，提出了没有用户愿意永远做新手用户，只有少数用户才会成为专家用户。因此，大多数用户都是永久的中间用户(perpetual intermediate user)，设计应该为中间用户优化的精辟论解。

在行为和形式设计方面，作者深刻地揭示了一些现象背后隐藏的本质。首先从行为立场出发，定义了软件姿态(software posture)的概念，将软件分为独占(sovereign)、暂态(transient)、精灵(daemonic)和辅助(auxiliary)四种姿态。不同姿态的软件对应不同的用户类型，如独占姿态应用的用户是永久的中间用户，应该在使用整个屏幕的情况下优化独占姿态的应用；而暂态姿态应用应该简单、清晰而切中要点，保持在一个窗口和视图内。在用户心理层次，作者深刻地揭示了流(flow)状态："深深的完全沉思状态"，经常产生"轻微的欢娱"，能够让你忘记时间的流逝。因此，软件交互应该促进和加强流状态，而不是打断或者干扰流状态。为了创建高效的软件，作者提出了附加工作(excise)的概念，分析了附加工作产生的原因，指出只有消除附加工作，用户才会效率更高。在这种背景下，作者也详细地分析了图形用户界面的导航(navigation)问题，以及如何消除导航中出现的附加工作。另外，作者指出要想开发效能高并使用户愉悦的软件，软件必须能够体贴人(considerate)和足够聪明（smart）。也正是从这个角度出发，作者详细地讨论了如何改善数据检索(data retrieval)和数据输入(data entry)，使其体贴人和足够聪明。

在讨论如何为不同用户设计时，作者指出有两个概念在根据不同的经验水平将用户的需求进行分类方面特别有用：命令向量(command vector)和工作集(working set)。命令向量是允许用户向程序发起指令的特殊技术。下拉菜单、弹出菜单和工具条控件都是命令向量的例子。好的用户界面提供多种命令向量，这种冗余性让不同技能水平的用户和不同偏好的用户根据自己的意愿和能力向程序发起命令。因为每个用户都在无意识地记忆经常使用的命令，持久的中间用户记住了命令和功能的适度子集：工作集。虽然严格地说，没有一个标准的工作集可以覆盖所有用户的需求，但是用户和使用模式的研究和建模可以产生一个较小的功能子集。这个最小工作集可以通过目标导向的设计方法确定：利用场景来发现你的人物角色所需求的功能。这些需求直接转化成最小工作集的内容。

作者也详细地阐述了视觉界面设计的一些基本原理，并在具体的背景下讨论了这些视觉界面设计原理的应用。如视觉界面必须：

- 避免视觉噪声(visual noise)和杂乱(clutter)；
- 使用对比(contrast)，相似性（similarity）与分层(layering)来区分和组织元素；
- 在每一个组织层次提供视觉结构和流；
- 使用紧凑、一致而背景合适的图像；
- 全面而有目的地结合风格和功能。

在对主流的三类界面范例：以实现为中心（implementation-centric）、隐喻(metaphoric)和习惯用法(idiomatic)分析的基础上，作者独特而精辟地指出隐喻的限制，强调了习惯用法的力量，深刻地指出“所有的习惯用法都需要学习，而好的习惯用法只需要学一次”。

另外，作者凭借渊博的知识和丰富的产业界经验，深刻全面地揭示和阐述了各种交互细节的本质、演化和蕴含的发展机遇，如直接操作(direct manipulation)、选择、拖放(drag and drop)。在图形用户界面发展的历史背景下，作者高屋建瓴地详细讨论了各种控件和它们的行为。

在译者学习和翻译这本书的过程中，深深地感受和体会到作者的大师风范以及卓尔不群的见解和深刻的思想，并且在具体的科研实践中受益。相信任何一个读者只要用心读这本书，都会有同样的感受，并有丰富的收获。

本书的出版凝聚了很多人的努力，在此我要感谢 De Dream[1]的蒋芳女士为本书倾情审校，电子工业出版社的参与此书出版的所有工作人员所付出的辛勤劳动，其中特别是张姝、方舟两位同志为此书所做的先期努力。

詹剑锋

[1] De Dream’是国内第一家专门研究软产品（包括软件、网站、消费类电子产品中的人机交互部分、客户服务）交互设计技术的组织，并拥有自己的网站。

前　言

Hugh Dubberly 是 Dubberly 设计办公室的首席设计师，他关注于通信系统、交互设计及信息设计。20 世纪 80 年代中期和 20 世纪 90 年代后期，Dubberly 在苹果电脑公司管理多学科设计团队，接着为整个公司管理创造性的服务和公司标志。在 Apple 期间，他服务于 Pasadena 的 Art Center College of Design，成为计算机图形部的第一个创建主席。后来他去了 Netscape，并且成为设计副总裁。Hugh 也在 San Jose 州立大学，图形设计系，IIT 的设计学院，以及 Stanford University 计算机科学系教授课程。

在这本书里，作者 Alan Cooper 和 Robert Reimann 探索了被传统产品设计师和可用性从业人员所忽略的设计领域：复杂系统行为的设计——特别是逐渐无处不在，有时几乎看不见的基于软件的技术。Cooper 和 Reimann 坚信除了对技术本身的理解，对人类自身的理解是使这些交互系统不仅功能强大，而且使用起来令人愉悦的关键。

Alan Cooper 不是个普通的设计师——他是个程序员、发明家、战略家，是美国图形艺术学会（AIGA）和 ACM SIGCHI 的正式会员。他在这些领域都卓有成效。他有很多对程序员、设计师、产品计划人员和可用性从业人员说非常重要的观念。

Cooper 自 25 年前 PC 诞生以来一直从事软件设计。很少有人对软件界面设计做了如此长久而深刻的思考。他的结论是如今的多数软件根本不适合人类使用——“无能而又暴虐”是数字产品的两个显著特征。

问题出在哪呢？

问题在于：软件没有通过外部形式展示自己——机械设备常常是这样做的。软件增加一项新的功能几乎不需要成本，而机械设备增加功能则几乎总要增加成本。软件缺乏足够的负反馈来控制它的复杂度。结果导致纯粹的小题大做：数字产品特性成堆。麻烦在于每增加一个特性，产品就更难使用。这就使产品越来越难于使用——在我们试图使用它们的时候也就越来越烦闷。

随着微处理器功能进一步增加，计算机的价格越来越低，问题也增多了。结果是计算机集成进越来越多的产品。有计算机的地方就必须有软件，用户交互也很常见。现在已经很难找到哪种新轿车、家电或者消费电子设备不需要用户与软件进行交互。我们慢

慢地被软件所窒息——最好的情况是软件没有满足我们的需要，最糟糕的情况则是软件可能造成对人类的威胁。

本书作者指责问题出在软件开发过程的技术驱动特性。在该书中 Cooper 和 Reimann 提倡一种截然相反的方法：人类驱动过程。在这个过程的开始、结尾和中间都是为满足用户的目标，即目标导向设计过程。

在传统的软件开发过程中，一个公司内部的许多人——客户也经常是这样——要求新的特性。在很多公司里，最后的功能列表成为事实上的产品计划。程序员通过选择和商议功能列表，而使得这个方法更加糟糕，通常要在开发时间和功能特征之间作出权衡。在这种过程中，很难确定产品什么时候完成，更不用说好的产品。

Cooper 和 Reimann 做出结论：问题的核心在于负责开发软件的人们不知道什么才构成好的产品，甚至是合适的产品。同样，他们也不知道什么样的过程产生成功的产品。简而言之，他们通过试错法开发，使得产品的顾客满意度不会比碰运气更好。

在本书详细论述的目标导向的设计中，Cooper 和 Reimann 鼓吹了 6 种对常规软件开发方法的显著改善：

- 先设计，后编程（旧的方式是：尽可能早地编程——而在结束时做一些设计；或者在渐进的环境中，设计和编程同时进行）。
- 区分设计责任和编程责任（旧的方式是：程序员做出显著的有关用户如何与产品交互的决定——通常是在编程进行期间，这种做法明显地存在利益冲突）。
- 关注满足用户目标（旧的方式是：分析用户的任务，而不关注他们的目标，以及为什么他们完成任务）。
- 基于认真地研究实际和潜在的用户定义具体的原型用户（**人物角色**）（旧的方式是：经理人员和程序员讨论不具体的“终端用户”，允许术语“用户”变形适应任何情况）。
- 使用人物角色作为**脚本提纲**的主要人物——人物角色作为定义交互产品中功能、行为和形式的主要工具（旧的方式是：使用市场清单列表来定义产品的功能，而让程序员确定他们认为哪些应该创建）。
- 遵循为了行为做设计的原则（旧的方式是：单独依赖于形式原理，猜测其他的部分，接着通过用户的测试来反复进行糟糕的交互，直到大多数糟糕的问题通过打补丁的方式解决）。

这本书是一本设计数字化产品行为的启蒙书，它由一个在交互设计前沿有着10年设计咨询经验以及25年计算机工业界经验的卓越的权威撰写。这也是一本没有哪一个产品规划师、界面设计师及可用性从业人员或者程序员不想得到的书——一本使得我们的软件和我们的世界变得更美好的书。我们应该留意作者的忠告。

Hugh Dubberly

第二版导言

本书旨在为您提供一些有效而实用的工具来设计用户界面。这些工具很明显分为截然不同的两类：战术性工具和策略性工具。战术性工具是关于使用和创建用户界面的习惯用法（如对话框和按钮）的一些提示和技巧。而策略性工具是思考用户界面习惯用法的方式，换言之，即用户与用户界面习惯用法的交互方式。

虽然已经有了一些介绍策略性或者战术性准则的书，我们的目标在于写一本能将两者融为一体的书。在帮助您设计更有吸引力、更有效的对话框的同时，本书还将帮助您理解用户如何了解您的软件，以及与之交互的方式。

设计有效的用户交互和界面之关键在于将策略性和战术性的方法合二为一。例如，客观上不存在好的对话框——对话框的品质取决于具体的应用情形：用户是谁，他们的背景和目标是什么。仅仅应用一系列的战术性说明，会使创建用户界面变得更容易，但这并不能使最终结果更好。同样，对于用户应该如何与您的系统交互的深层次思考也不能改善软件本身。真正奏效的是：在策略上对用户与特定软件的交互方式保持敏感的同时，拥有一个可以在任意情况下应用的在你掌握之中的战术工具盒。本书既会加深您对用户的理解，又将教您如何把这些理解转变为设计理念。

谁应该读这本书

在 1995 年 8 月，软件观念革命这本书第一次出版时，**界面设计**还是个未开垦的新领域。少数人在软件工程的影子下，勇敢地以用户界面设计师的头衔工作，正如机敏的小哺乳动物在粗暴的巨龙阴影下爬行。正如在软件观念革命第一版中所指出的，软件设计被人们错误地理解和评价。过去是怎么做的，程序员通常就怎么做。很多处境不佳的文档工程师、培训者、技术支持人员，以及处于增长趋势的**可用性从业人员**都意识到：某些事情应该改变。

Web 令人吃惊的、似乎是一夜之间的发展和流行，驱动了这种改变。突然间，易用性（ease Of use)成了挂在每个人嘴边的术语。在 20 世纪 90 年代初期，即多媒体短暂流行的期间，涉足数字产品设计的传统设计师纷纷转向 Web。表面上，新的设计师头衔像杂草一样涌现：信息设计师(information designer)、信息架构师(information architect)、用

户体验策略师(user experience strategist)，以及交互设计师(interaction designer)。公司的首级职务（首席用户体验官，user experience officer）一开始就存在，他的工作核心是创建以用户为中心的产品。很多重点大学都争先恐后地开展这些理论的培训。与此同时，可用性和人性因素（human factor）从业人员的地位也在提升，现在被承认是推动更好产品设计的领导者。

虽然 Web 使得界面技术倒退了不止十年，但它无可争议地将用户需求永久地置于公司的关注内。作者坚信：.COM 的衰败只能使得用户及其需求的可见性，以及对它们的关注在将来变得更加明显。人们一般对新技术感到厌倦。消费者传达了明确的信息，他们所需要的好技术是容易使用的，并且能满足他们需求的技术。

因此，作者很高兴地说，新版本的读者群会大大地增大：任何对用户与数字产品的交互感兴趣的人都会在读这本书的过程中获得独特的能力洞察力。程序员、与数字产品相关的设计者、可用性从业人员、项目经理都会从此书中受益匪浅。读过《软件观念革命》第一版或《软件创新之路》第一版的读者会在此发现更新颖且更详细的有关设计方法和原则的详细信息。

为什么要做交互设计

《软件观念革命》的第一版描述了一门被称为软件设计的学科，同时也可称为用户界面设计。在这两个术语中，用户界面设计有更强的生命力。在本书里，我们仍会使用它，而且大多数是合适的。

然而，笔者很清楚，本书所讨论的内容要远比用户界面设计的范围广。界面这个词本意是指表面，本书所阐述的大多数设计问题要远远比 CRT 屏幕的表面问题深奥，它直接触及了“数字产品是什么”，以及“数字产品要做什么”等核心问题。

近些年来，对于这类设计，人们已经提出了许许多多的术语。在 2000 年左右，公司对 Web 的兴趣达到顶点时，被称为信息架构（information architecture，IA）的学科似乎最终包含了此处这里讨论的这类设计。但是正如 Web 在经济方面的前景已经暗淡一样，IA 基本上也保留了它以 Web 为中心的狭隘视图：如何组织和浏览页面上的内容。随着新经济的明显下滑，IA 产业的好运也逐渐消失。

另一个近年来流行的术语是**体验设计**。美国图形艺术研究所(American Institute of Graphic Artists，AIGA)特别提倡使用这个术语来概括用于开发数字产品和系统的不同设

计与可用性学科。这个想法很有吸引力，但它仍然回避了一个问题——什么样的设计才是交互式系统设计的真正核心，交互式系统设计是一种明显不同于已有设计的崭新设计。

体验设计这种想法也有一定的问题。在笔者看来，体验是人与人工制品（或者其他生物）交互的结果。体验出现在一定的上下文中，进一步由内部、心理的个人环境所调节，这种个人环境由动机、过去经验、气质和多种认知因素形成。

作为设计者，我们不能声称能够设计一种人工制品或者系统的用户体验，但我们能够设计与人工制品交互的机制，以改善用户体验。因为，我们相信体验发生在人和人工制品交互的过程中，我们已经选择了“交互设计”这个术语来表示本书描述的这类设计，该术语由 Bill Moggridge 和 Bill Verplank 在 20 世纪 80 年代首创。你不能设计体验本身，但你能设计调节和引导体验的交互行为。

交互设计的定义

简单地说，交互设计是**人工制品**、环境和系统的行为，以及传达这种行为的**外形元素**的设计与定义。不像传统的设计学科主要关注**形式**，最近则是关注**内容和内涵**，而交互设计首先旨在规划和描述事物的行为方式，然后描述传达这种行为的最有效形式（参见图 1）。

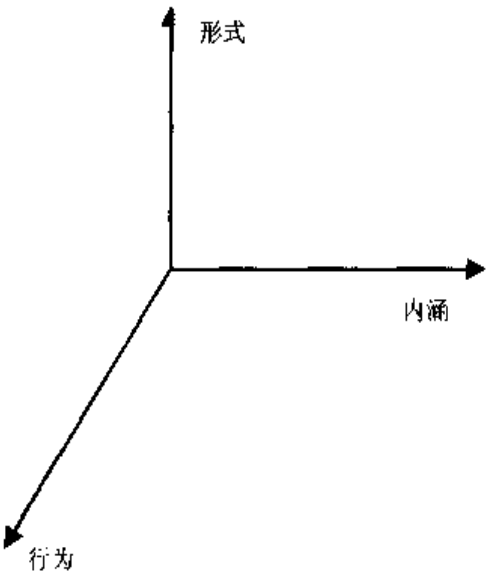
交互设计借鉴了传统设计、可用性及工程学科的理论和技术。它是一个具有独特方法和实践的综合体，而不只是部分的叠加。它也是一门工程学科，具有不同于其他科学和工程学科的方法。

交互设计是一门特别关注以下内容的学科：

- 定义与产品的行为和使用密切相关的产品形式。
- 预测产品的使用如何影响产品与用户的关系，以及用户对产品的理解。
- 探索产品、人和**上下文**（物质、文化和历史）之间的对话(Riemann 和 Forlizzi,2001)。

交互设计从“目标导向”的角度解决产品设计：

- 要形成对人们希望的产品使用方式，以及人们为什么想用这种产品等问题的见解。
- 尊重用户及其目标。
- 对于产品特征与使用属性，要有一个完全的形态，而不能太简单。
- 展望未来，要看到产品可能的样子，它们并不必然就像当前这样。



设计的三维

交互设计着重于传统设计较少探索的领域：行为设计。

所有的设计影响人类行为：结构关于人们使用空间的方式，与形式和光线有关。如果没人对张贴画所表达的信息有所反应，那张贴画又有什么意义呢？

引入交互技术——计算机的礼节——来设计人工制品的行为，以及这种行为如何影响和支持人的目标和期望？已经成为一门值得关注的学科。

理解交互设计和传统设计关注点的不同方法之一是借助历史透镜。在 20 世纪的上半叶，设计者主要关注形式。后来设计者逐渐关注内涵，例如，产品设计师和结构师在 20 世纪 70 年代引入了本土和怀旧形式。直到今天，这种趋势仍在继续，诸如 PT Cruiser 的怀旧风格的汽车。今天，信息设计师继续关注内涵，包括可用内容的设计。

在最近 15 年以来，越来越多的设计师开始谈论行为：基于软件(software-enabled)的产品（或复杂的机械）直接与用户交互的动态方式。

这些关注（形式、内涵及行为）并不是相互排斥的。交互产品必须多少包含各部分：软件应用关注更多的是行为和形式，而对内容的需求较少；Web 站点和公用信息亭关注更多的是内容和形式，而较少关注复杂的行为。

图 1　设计的三维元素：设计在传统上关注形式，最近关注更多的是内容和内涵。最新的设计元素是行为：复杂系统与人交互的方式

因为，复杂系统的行为通常不是一个审美学的问题，而是认知因素和逻辑过程的结合，交互设计应该采用系统化方法，且能从这种方法中受益匪浅。

交互设计师应该，也是首先要做的，理解使用他们设计的人们的目标、动机和期望（即心智模型[2]，mental model）。这些最好能被理解为“叙述”[3](narrative)——一时间轴上的

[2] **译者注**　作者定义的术语，见第 2 章。

[3] **译者注**　作者定义的术语，见第 6 章。

逻辑（或者情感）进展。

与这些“叙述”相适应，所设计的人工制品必须具有它们自己的行为叙述，且这些行为必须成功地与用户的期望吻合。不像大多数机械制品，只有简单的行为，观察之下一切都一目了然，软件和其他数字产品因为其行为潜在的复杂性，所以它们需要交互设计。软件对于观察者是不透明的，然而它所表现出来的可能行为几乎是无限的。

一些设计者，以设计的传统形式，如视觉、声音、触觉、模式、风格及习惯用法为论据，认为交互元素应被视为随着时间变化的感觉数据流，类似于动画，因此完全可以通过传统设计方法来描述。然而，这种论点有严重的缺陷：尽管交互设计面向形式的方面非常重要，但是，除非它们是通过有效和合适的行为组织的，否则几乎没用。如果没有一个逻辑结构或流来帮助解决用户的实际问题，那么面向形式的交互设计本身只是隔靴搔痒，价值值得怀疑。

换句话说，如果没有条理化的“叙述”，感觉数据本身毫无意义。电影不能光有特效，叙述也必非常不可少。这一点，对与数字产品的交互而言，更有效。因为对话不在第三方可观察到的虚幻境界中发生。相反，它是在人和设计的人工制品之间的交流，Bill Buxton（1990）称之为“非文字的自然语言”。这种对话（也就是行为）的期望和设计是交互设计的实质。

本书的内涵和外延

本书是一本有关交互设计（交互系统复杂且以用户为中心的行为设计）的原理和方法的参考书。本书的第一篇强调**设计过程**，以及对用户的系统理解；第二篇提供了策略原理和工具；第三篇更深地钻研战术性的问题。

本书不打算以**指南**的姿态出现，或者提供一些界面标准。实际上，你会在第19章了解到为什么那些工具的使用是有限制的，而仅仅与特定环境相关。也就是说，在本书中描述的过程和原理是与你选择的风格指南相兼容的，它也是解决任意这类问题的一本极好的配套书。指南擅长于回答做什么，但无法回答为什么这样做。本书打算解决交互系统设计中未解决的问题。

设计交互系统有4个步骤：研究问题域，理解用户及其需求；定义解决问题的框架；完善设计细节。

很多业内人士会加上第5个步骤：确认——让用户测试方案的有效性。他们这么做，

不会错。最后一步是众所周知称为可用性学科的一部分。

可用性方面的重要文献在持续增长，但有关交互设计的资料却相对很少。本书专门关注交互设计的过程和原理，设计方案的测试方法则留给出版的相关学术著作。本书可以与可用性工程方法和实践的文献配套使用。通过和谐地结合这两个学科，你会获得最好的设计成果。

交互设计的工作语言

物理学家们开发并应用物理学的专门术语。这些术语不仅阐明了他们身边的特殊过程和物体，而且也影响了他们的思考方式。一个科学家如何向其他科学家表述一个问题、想法或者发现呢？技术工作语言是每一门学科的基础，从蜘蛛研究到印刷机的工作方式。

计算机产业也不例外。我们有丰富而复杂的语言来描述一些细微差别，诸如，并发、递归及编译器等词汇。然而，那些只是工程术语。在交互设计领域我们还没有类似的丰富语言，这种现状需要改变。

我们已经听说过有关描述数字产品和界面的有效性或用户友好性的讨论。当某人提到有效的用户界面时，他是指代码？控件的数目？容易编程？容易学习？还是容易使用呢？当然，这些词语在聪明的技术人员心中会像魔法一样召唤一些图像，但要成功而系统地设计人与复杂的数字系统交互，仅有模糊的图像是不够的。

在交互设计领域，缺少一致而专门的术语使设计者的努力遭受挫折。没有精确的术语，我们被迫用手比划。没有清晰而可细致区分的术语，我们偶尔会将事物错误定位，忽略有意义的重要事实，或不经意间将坏的当成好的。

为了设计有效的数字产品，我们必须拥有能够精确描述我们所寻求的目标的词汇，以及用于实现这些目标的工具。除非我们能够创造出自己的技术工作语言，否则交互设计不会成为一门真正的科学、艺术或者工艺；除非我们发展出精确并可分析的方法来思考和讨论我们所做的事情，否则不会有成功的实践；除非我们在用来描述“我们所做的，所关心的，以及如何判断是否成功地实现目标”这些术语方面达成一致，否则我们不仅不能有效地工作，而且我们在外部世界尤其是软件工程和商业世界的可信度也会受到威胁。

与第一版的不同

自 1995 年 About Face 的第一版首次出版以来，界面设计领域的很多内容已经发生改变。然而，也有很多内容保持不变。本书保留了仍然适用的内容，更新了已经改变的内容，提供了一些新的材料。这些材料不仅反映了近 7 年的发展，也包括作者近年来在实践中发展的一些新概念。

下面是您会在本书中发现的最显著改变：

- 正如作者所期望的，本书完全重新组织了内容表达结构，这种方式使读者更容易阅读或参考。本书分为三篇：第一篇讨论有关用户和设计的高层次思想和设计过程；第二篇讨论高层次的交互设计原理；第三篇讨论低层次的界面设计原理。
- 在 Alan Cooper 的著作“The Inmates Are Running the Asylum”(SAM, S, 1999) 中首先引入概念的基础上，又增加了好几个新的章节，详细地描述了目标导向设计的过程，包括研究技术，人物角色[4]的创建，以及如何使用人物角色和脚本提纲来综合交互设计方案。
- 本书做了很多努力，以专门解决与非桌面平台相关的问题。第 37 章和 38 章各自解决了 Web 及设备平台的交互和界面设计。
- 本书已经尽可能更新了术语和例子，以反应产业界的当前状况。为了改善清晰性和可读性，正文整体上已经重新编辑。
- 当作者在撰写这些材料时，希望读者将发现这些增加和改变是值得的。

本书中使用的惯例

在本书中，我们使用了很多惯例，希望能够借此阐述清楚一些重要的观点，并且为读者提供丰富和有用的体验。

平台

本书讨论数字化交互产品的设计。现在大部分计算机运行的是 Windows 操作系统，这就是为什么需要更加理解如何创建有效和目标导向的用户界面。

---

[4] **译者注** 作者将在第 1 章详细论述人物角色和脚本提纲等术语。

前面已经说过，本书中大多数材料是与平台无关的。它同样适用于所有的桌面平台，如 Mac OS, Motif 及其他。大多数甚至适用于差异很大的平台，如公用信息亭、手持设备、嵌入式系统及其他的平台。

例子

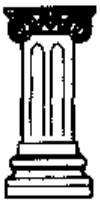
本书的大多数例子选自微软的办公套件：Word，Excel，Powerpoint，Outlook，以及 IE 浏览器，还有 Adobe Photoshop，Illustrator，以及其他的一些例子。我们之所以坚持选择使用主流程序有两个原因：首先，读者至少熟悉这些例子；其次，重要的是想表明即使是最好的产品也能够通过目标导向的方法来改善。在他们特别具有说服力的时候，我们也选择了一些 Mac OS X 的例子。

新版本的一些例子来自已经废弃的软件或者操作系统版本。这些例子能够说明设计观点，作者感觉到非常有用，有必要保留。而大多数例子来自正在流行的软件和操作系统。

设计格言和术语

在交互设计这个新的领域缺少的是一些术语和原理集。借助它们，设计师、程序员和可用性从业人员能够在一起交流行为设计方面的问题。我们希望在本书中能够提供起点，在此基础上，从业人员在合适的时候能够应用和扩充这些不断演变的术语和原理列表。

原则，或者你更偏向于说格言，继承了从很多设计实践中获得的智慧。它们可以分为两类，每一类都用不同的图标表示，如下所示。



公理：低价买入，高价卖出。

在本书中，公理指用途广泛的交互设计原则。当你发现自己被困难的概念问题缠住时，可以求助这些公理。所有公理都收录在附录 A 里。

设计技巧：让你的火药保持干燥。

有些设计格言使用范围并不广，但它们非常有用。当你遇到本书中描述的特定设计子问题时，设计技巧能够帮助你，它们收录在附录 B 里。

对于交互设计从业人员来说特定含义的术语，当第一次提到时用粗体字突出显示。有些术语是作者独创，但很多是其他人所创造的或者已经得到普遍使用。

让我们开始设计

我们希望本书能为您提供信息，并激发您的兴趣，尤其希望您能够以新的方式思考数字产品的设计。交互设计的实践仍在不断地演化，它新颖而丰富多彩，足以在同一个题目上产生各种各样的观点。如果您有有趣的观点，或者仅仅想和我们交流，我们很希望能在 acooper@cooper.com 和 rmreimann@aol.com 中收到您的电子邮件。

目　录

第一篇　了解你的用户

第一部分　弥合差距

1　目标导向设计……5
数字产品需要更好的设计方法……5
数字产品设计的现状……6
我们对用户一无所知……9
我们面对利益冲突……9
我们缺少软件开发过程……9
制造业设计的演变……10
行为规划和设计……12
识别用户目标……12
目标 VS 任务……13
设计要满足上下文中的目标……14
目标导向的设计过程……15
弥合差距……16
过程概况……18
产品成功的关键是目标，而不是特性……22
2　实现模型和心智模型……24
实现模型……24
用户心智模型……25
表现模型……25
大多数软件遵从实现模型……28
工程师设计的界面遵从实现模型……28
数学思维方式产生实现模型的界面……29
机械时代的表现模型与信息时代的表现模型……30
机械时代的表现方式……31
新技术需要新的表现方式……31
机械时代的表现方式有损用户交互……32

对机械时代表现方式的改善：一个例子 ····· 33
3 新手、专家和中间用户 ····· 35
永久的中间用户 ····· 35
为中间用户优化 ····· 37
新手需要什么 ····· 38
让新手开始 ····· 39
专家需要什么 ····· 39
永久的中间用户需要什么 ····· 40
4 理解用户：定性研究 ····· 42
定性研究与定量研究 ····· 42
定性研究的价值 ····· 43
定性研究的类型 ····· 44
人种学调查：用户访谈和用户观察 ····· 47
上下文调查 ····· 48
上下文调查的改进 ····· 48
为人种学调查做准备 ····· 49
进行人种学调查 ····· 52
其他类型的研究 ····· 56
焦点小组 ····· 56
市场统计和市场划分 ····· 57
可用性和用户测试 ····· 57
5 用户建模：人物角色和目标 ····· 59
为什么要用模型 ····· 60
人物角色 ····· 60
作为设计工具的人物角色的力度 ····· 61
人物角色产生于研究 ····· 63
作为个体代表的人物角色 ····· 63
人物角色代表着场景中不同的用户类 ····· 64
发掘了行为深度的人物角色 ····· 65
人物角色必须有动机 ····· 66
人物角色与用户角色 ····· 66
人物角色与用户简要 ····· 67

人物角色与市场划分……67
用户人物角色与非用户人物角色……68
目标……69
目标激发用户模式……69
目标必须来自定性数据……69
目标类型……69
成功的产品必须首先满足用户目标……72
构造人物角色……73
再次讨论人物角色假设……74
将访谈主体映射为行为变量……74
识别显著的行为模式……74
综合特征和相关目标……75
检查完整性和独特性……76
展开叙述……77
指定人物角色类型……78
其他模型……80
6 脚本提纲：将目标转换为设计……81
作为设计工具的叙述……81
设计中的脚本提纲……82
在脚本提纲中使用人物角色……83
基于人物角色的脚本提纲与用例……83
用基于人物角色的设计来构思设计方案……84
定义需求……84
定义交互框架……90
优化形式和行为……95
7 综合好的设计：原则和模式……98
交互设计原则……98
原则减少工作量……99
在不同细节层次运作的原则……99
原则与风格指南……100
交互设计模式……100
交互和建筑模式……100

交互设计模式的类型 101
结构模式、模式嵌套和预定制的设计 102
交互设计规则 102
伦理的交互设计 103
注重实效的交互设计 104
优雅的交互设计 105

第二篇　设计行为与形式

第二部分　去除障碍，达到目标

8　软件姿态 111
桌面的姿态 112
独占姿态 112
暂时姿态 117
精灵姿态 120
辅助姿态 122
Web 的姿态 123
面向信息的站点 123
事务网站和 Web 应用 124
Web 门户 125
其他平台的姿态 126
公用信息亭 126
手持设备 127
电器 127
9　和谐与流 129
流与透明性 129
遵循心智模型 130
引导，但不要讨论 131
将工具放在顺手的地方 132
无模态反馈 132
和谐 133

增加的技巧：更少，更好 …… 134
区分可能性和概率 …… 135
提供比较 …… 137
使用图形化输入 …… 139
反映程序状态 …… 139
避免不必要的程序状态报告 …… 140
避免白板 …… 141
命令调用与配置 …… 141
提问与提供选择 …… 142
隐藏弹射座椅控制杆 …… 144
10 消除附加工作 …… 146
什么是附加工作 …… 146
GUI 附加工作 …… 147
附加工作和专家用户 …… 148
训练工具 …… 148
“纯粹”的附加工作 …… 149
视觉附加工作 …… 149
确定什么是附加工作 …… 150
停止进度 …… 150
错误、通知和确认信息 …… 151
让用户申请许可 …… 152
不必要的保护措施 …… 153
不响应 …… 154
常见的附加工作陷阱 …… 154
11 导航和调整 …… 155
导航是附加工作 …… 155
导航类型 …… 156
在多个窗口或者页面之间导航 …… 156
在窗格之间导航 …… 156
在工具和菜单之间导航 …… 158
信息导航 …… 159
改善导航 …… 160

减少目的地数目 …… 160
提供导航标志 …… 161
提供总体视图 …… 163
提供合适的控件——功能映射 …… 165
调整界面以适应用户需要 …… 167
避免层次关系 …… 169
12 理解撤销 …… 171
用户和撤销 …… 171
用户有关错误的心智模型 …… 172
撤销支持用户探索 …… 172
设计撤销功能 …… 173
撤销的类型和变体 …… 173
渐增动作和过程动作 …… 174
隐蔽撤销和解释性撤销 …… 174
单次撤销和多次撤销 …… 174
恢复 …… 176
分组多次撤销 …… 177
类似撤销行为的其他模型 …… 178
比较：这看起来怎么样 …… 178
分类撤销 …… 178
被删除的数据缓冲区 …… 180
里程碑和复原 …… 180
冻结 …… 181
防撤销操作 …… 181
13 重新思考“Files”和“Save” …… 182
保存文件改变怎么了 …… 182
实现模型存在的问题 …… 184
关闭和不想要的改变 …… 184
另存为 …… 184
归档 …… 186
实现模型与心智模型 …… 187

隐藏文件系统的实现模型 …… 187
设计统一的文件表现模型 …… 188
统一文档管理 …… 189
自动保存文档 …… 189
创建文档的拷贝 …… 190
命名和重命名文档 …… 191
存放和移动文档 …… 191
指定文档的存储格式 …… 191
对改变进行复原 …… 192
放弃所有的改变 …… 192
创建文档的里程碑拷贝 …… 192
新的文件菜单 …… 192
文件菜单的新名字 …… 193
磁盘和文件系统是一个特性 …… 194
改变的时机 …… 195

第三部分　提供高效能和愉悦

14　设计体贴的软件 …… 199
设计体贴的软件 …… 199
什么使软件更体贴 …… 200
体贴的软件对用户感兴趣 …… 201
体贴的软件是恭顺的 …… 201
体贴的软件是乐于助人的 …… 201
体贴的软件具有常识 …… 202
体贴的软件预见需求 …… 202
体贴的软件是尽责的 …… 202
体贴的软件不会因为自己的问题而增加你的负担 …… 203
体贴的软件及时通知我们 …… 204
体贴的软件有理解能力 …… 204
体贴的软件是自信的 …… 204
体贴的软件不问许多问题 …… 205

体贴的软件失败也不失风度 …… 205
体贴的软件知道什么时候调整规则 …… 206
体贴的软件承担责任 …… 207
体贴的软件是可能的 …… 208
15 设计智能的软件 …… 209
利用计算机的空闲周期 …… 210
浪费的时钟周期 …… 210
更好地利用时钟周期 …… 211
给软件赋予记忆 …… 211
任务一致性 …… 212
记住选择和默认 …… 212
记住模式 …… 213
要记住的动作 …… 213
文件位置 …… 214
可推论的信息 …… 214
多会话撤销 …… 214
过去的数据输入 …… 214
程序文件的外部活动 …… 215
赋予应用程序记忆 …… 215
缩小决策集合 …… 216
偏好阈值 …… 216
多数情况下，多数是对的 …… 217
记忆带来不同 …… 217
16 改进数据检索 …… 218
存储和检索系统 …… 218
物理世界的存储和检索 …… 219
一切都各就其位：通过位置存储和检索 …… 219
基于索引的检索 …… 219
数字世界的存储和检索 …… 220
检索方法 …… 221
基于属性的检索系统 …… 221

关系数据库与数字汤 …… 223
组织非结构化的事物 …… 223
数据库存在的问题 …… 223
基于属性的替代方案 …… 224
自然语言输出：一种基于属性检索系统的理想界面 …… 225
17 改进数据输入 …… 227
数据完整性与数据免疫 …… 227
数据免疫 …… 229
丢失数据怎么办 …… 229
数据输入和规避能力 …… 231
审核与校正 …… 231
18 为不同的需要进行设计 …… 234
命令向量和工作集 …… 234
直接向量和教学向量 …… 235
工作集和人物角色 …… 235
让新手用户变为中间用户 …… 236
世界向量和头脑向量 …… 236
记忆向量 …… 237
个性化和配置 …… 238
特殊的模态行为 …… 239
本土化和全球化 …… 240
库和模板 …… 241

第四部分　应用视觉设计原则

19 外观设计 …… 245
视觉艺术和视觉设计 …… 245
图形设计和视觉界面设计 …… 246
图形设计和用户界面 …… 246
视觉界面设计和视觉信息设计 …… 246
工业设计 …… 247

可视界面设计原则……247
避免视觉噪音和杂乱……248
使用对比和分层来区分和组织元素……249
在每个组织层次提供视觉结构和流……251
使用紧凑、一致和上下文适宜的图像……254
全面而有目的地结合风格和功能……256
可视信息设计的原则……257
加强视觉对比……258
显示因果关系……259
显示多个变量……259
在一帧显示中结合文本、图形和数据……260
确保内容的品质、相关性和完整性……260
在相邻空间上显示事物，而不是按时间堆积……260
可量化的数据就要量化……261
在可视界面中使用文本和颜色……261
使用文本……261
使用颜色……262
颜色的滥用……262
一致性和标准化……263
界面标准化的益处……263
界面标准化的风险……264
标准、指导准则和经验法则……264
什么时候打破规则……265
应用程序之间的一致性和标准……265
20 隐喻、习惯用法和启示……267
界面范例……267
实现为中心的界面……268
隐喻界面……268
习惯用法界面……270
隐喻的其他局限性……272
找到好的隐喻……272
使用全局隐喻所带来的问题……273

Mac 和隐喻：修正主义观点 …… 274
创建习惯用法 …… 275
手动启示 …… 277
手动启示的语义学 …… 278
实现用户期望的启示 …… 278

第三篇　交互细节

第五部分　鼠标和操作

21　直接操作和定点设备 …… 285
直接操作 …… 285
程序操作与内容操作 …… 287
直接操作过程的三个阶段 …… 287
直接操作习惯用法的视觉反馈 …… 288
直接操作是有效交互的关键 …… 288
定点设备 …… 289
光笔和阴极射线管（CRT） …… 289
鼠标和间接操作 …… 290
其他定点设备 …… 291
笔的回归 …… 292
鼠标的使用 …… 293
鼠标按钮知多少 …… 295
鼠标左键 …… 295
鼠标右键 …… 296
鼠标中间键 …… 296
用鼠标指向和单击 …… 296
鼠标释放和按下事件 …… 299
光标 …… 300
受范性和暗示 …… 300
输入焦点 …… 303

元键 …… 304
22 选择 …… 305
对象—动词的次序和选择 …… 305
离散选择和连续选择 …… 306
互斥 …… 307
添加选择 …… 308
插入和替换 …… 309
成组选择 …… 310
选择的视觉提示 …… 310
反色 …… 311
选择彩色对象 …… 312
23 拖放 …… 313
拖放的定义 …… 313
内部拖放和外部拖放 …… 314
源—目标范例 …… 314
拖动数据对象到目标功能 …… 315
拖动功能对象到目标数据 …… 316
源—目标交互 …… 316
拖动时的视觉反馈 …… 317
完成拖放操作 …… 319
插入目标 …… 319
完成时的视觉反馈 …… 320
其他拖放交互问题 …… 320
自动滚屏 …… 320
避免拖放抖动 …… 322
鼠标游标 …… 324
24 操作控件、对象和连接 …… 326
控件操作 …… 326
单击—拖动 …… 326
从捕获中逃逸 …… 327
受范性反应 …… 329

单击—拖动控件 …… 330
调色板工具的行为 …… 330
对象操作 …… 332
调整位置 …… 333
调整大小和调整形状 …… 333
调整大小和调整形状的元键变体 …… 335
三维对象的操作 …… 336
对象连接 …… 340

第六部分　控件及其行为

25　窗口行为 …… 345
PARC 和 Alto …… 345
PARC 原则 …… 346
视觉隐喻 …… 347
避免模态 …… 347
层叠窗口 …… 348
微软与平铺窗口 …… 349
全屏应用程序 …… 349
多窗格应用程序 …… 350
选择你的窗口 …… 351
不必要的空间 …… 351
必要的房间 …… 352
窗口污染 …… 354
窗口状态 …… 355
为什么最小化 …… 356
为何要多元状态 …… 358
MDI 与 SDI …… 359
26　使用控件 …… 361
避免布满控件的对话框 …… 361
命令控件 …… 362
按钮 …… 362

图标按钮……363
选择控件……365
复选框……365
触发按钮：一种应该避免的选择习惯用法……366
单选按钮……367
组合图标按钮……369
列表控件……370
树形控件……376
输入控件……376
有界输入控件和无界输入控件……377
微调控制项……378
无界输入：文本编辑控件……379
显示控件……385
文本控件……385
可恨的滚动条……385
滑动块和刻度盘……387
拇指轮……387
分隔器……387
抽屉和拉动杆……387
27 菜单：教学向量……389
命令行界面……389
顺序层次关系菜单……390
Lotus 1-2-3 界面……391
下拉菜单和弹出菜单……393
今天的菜单：教学向量……394
28 使用菜单……397
标准菜单……397
文件菜单……398
编辑菜单……399
窗口菜单……399
帮助菜单……399

可选菜单……400
视图菜单……400
插入菜单……400
设置菜单……400
格式菜单……400
工具菜单……401
一些有问题的菜单习惯用法……401
级联菜单……401
扩展菜单……402
突然弹出式菜单……403
菜单项惯例及变体……404
禁止菜单项……404
使菜单项具有校验标记……404
触发菜单项……404
菜单图标……405
快捷键……406
助记符……407
Windows 系统菜单……407
29 使用工具条和工具提示……408
工具条：可见的立即功能按钮……408
工具条不是菜单……409
工具条使菜单更限于教学目的……409
工具条和工具条控件……409
工具条上的图标与文本……410
标签化图标按钮存在的问题……411
解释工具条控件……411
气球帮助：第一次尝试……411
工具提示……412
禁用工具条控件……413
工具条的演化……414
状态指示工具条控件……414
工具条上的菜单……415

可移动工具条 …… 415
可定制工具条 …… 416
Windows 任务栏：特殊目的的工具条 …… 417
开始菜单 …… 417
快速启动工具条 …… 418
窗口按钮 …… 418
状态区 …… 419
任务栏上应该有更多的工具条吗 …… 419
30 使用对话框 …… 420
对话框暂停了正常交互 …… 420
对话框基础 …… 422
模态对话框 …… 422
非模态对话框 …… 423
非模态对话框存在的问题 …… 424
改进非模态对话框的两个方法 …… 424
目标导向对话框 …… 429
属性对话框 …… 429
功能对话框 …… 430
进度对话框 …… 431
消除进度对话框 …… 433
公告对话框 …… 433
31 对话框礼节 …… 435
礼貌是对话框的美德 …… 435
标题栏 …… 436
暂时姿态 …… 437
减少附加工作 …… 438
了解需要减少附加工作的位置 …… 439
了解是否需要减少附加工作 …… 439
模态对话框的终止命令 …… 440
关闭框 …… 441
帮助按钮 …… 442

键盘快捷方式 …… 443
标签对话框 …… 443
广度与深度 …… 444
堆叠标签：滥用对话框 …… 445
扩展对话框 …… 446
级联对话框 …… 447
动态对话框 …… 448
32 创建更好的控件 …… 450
直接操作控件 …… 450
例 1：可拖动的阴影 …… 451
例 2：指定网格 …… 451
摘录控件 …… 452
视觉控件 …… 455

第七部分 与用户的交流

33 消除错误 …… 461
错误对话框被滥用了 …… 461
为什么我们有这么多的错误消息 …… 462
错误消息怎么啦 …… 462
人们讨厌错误消息 …… 462
到底是谁的错 …… 463
错误消息不起作用 …… 465
消除错误消息 …… 465
使错误不可能 …… 466
正面反馈 …… 467
有特例吗 …… 468
改进错误消息：最后一招 …… 469
错误的结束 …… 470
34 通知和确认 …… 471
提示和确认 …… 471

提示：宣布显而易见的内容 …… 471
确认 …… 473
取代对话框：丰富的非模态反馈 …… 477
丰富的视觉非模态反馈 …… 477
听觉反馈 …… 479
35 与用户的其他交流方式 …… 482
桌面上的标志 …… 482
程序名称 …… 482
程序图标 …… 483
辅助应用程序窗口 …… 484
“关于”对话框 …… 484
闪屏 …… 486
共享软件闪屏 …… 487
在线帮助 …… 487
索引 …… 487
快捷方式和总览视图 …… 488
不是为新手用户准备的 …… 488
非模态和交互式帮助 …… 488
向导 …… 489
“智能”代理 …… 490
36 安装过程 …… 491
最好的安装就是无须安装 …… 491
接二连三的厄运 …… 492
在不通知后果的情况下，要求用户做出响应 …… 493
不告诉用户动作的影响范围 …… 494
提出用户不可能知道答案的问题 …… 495
向用户询问计算机自身能够回答的问题 …… 496
不做准备 …… 496
不提供卸载方法 …… 497
忽略先前活动 …… 498
滥用系统范围内的文件 …… 498
将文件放在不妥当的地方 …… 498

覆盖共享文件 …… 499
不向用户提供有关程序的任何信息 …… 500
混淆配置和安装 …… 500
要求用户主动参与 …… 500

第八部分　超越桌面的设计

37　Web 设计 …… 503
好消息和坏消息 …… 504
Web 设计的普遍神话 …… 505
Web 站点与 Web 应用 …… 507
Web 站点 …… 507
Web 应用 …… 507
38　嵌入式系统的设计 …… 514
一般设计原则 …… 514
不要把你的产品当成计算机 …… 515
结合硬件和软件设计 …… 515
上下文驱动设计 …… 516
明智地使用模态 …… 517
限制范围 …… 518
平衡导航与显示密度 …… 518
将输入复杂性最小化 …… 519
为你的平台定制 …… 519
手持设备的设计 …… 520
设计公用信息亭 …… 521
公用信息亭的姿态和导航 …… 522
事务与探索 …… 522
公共环境下的交互 …… 523
管理输入 …… 523
听觉界面的设计 …… 524
附录 A　本书公理集 …… 525
附录 B　本书设计技巧集 …… 529
跋：给同行的话 …… 533

第一篇

了解你的用户

交互设计不是附属于我们的系统、应用和硬件的皮毛。可以想像，在我们创建好产品的内部功能后，再想为产品创建一个好的用户体验，就像是在说好的涂料层能将洞穴变成大厦一样。即使技术能够让我们自由地创造，与此同时它也将我们束缚在与人类行为的自然表达恰恰相反的思维方式上。几乎所有与数字产品设计相关的问题都是因为有良好意识、聪明、能干的设计师关注于错误的事情。我们必须关注于用户极力获取的目标，以及他们为那么做的动机，而不是技术和任务——即使用户不能清晰地表达。本篇详细地概括了弥合在用户目标和产品设计之间存在的差距的方法。

第一部分

弥合差距

1 目标导向设计

2 实现模型和心智模型

3 新手、专家和中间用户

4 理解用户：定性研究

5 用户建模：人物角色和目标

6 脚本提纲：将目标转换为设计

7 综合好的设计：原则和模式

1

目标导向设计

本书有一个简单的前提：如果实现用户的目标是我们设计过程的基础，用户就会满意、开心。如果用户开心，他就会乐意给我们付钱（并且向其他人推荐），然后我们就会取得商业上的成功。

表面上，这个前提看起来既直观又明显：让用户开心，你的产品便会获得成功。那为什么又有那么多数字产品使用起来既困难又不令人愉快呢？为什么我们不能快乐或者成功——抑或两者兼得呢？

数字产品需要更好的设计方法

现在大多数的数字产品从开发过程中出现的情形，就像怪物从冒着水泡的池子里面出来一样。开发者不是认真与用户一起规划和执行，而是一直创建技术解决方案，直到最终对它们失去控制为止。就像疯狂的科学家一样，他们失败，是因为他们没有将人性融入他们的创作之中。

根据工业设计师 Victor Papanek 的定义，设计（design）是为构建有意义的秩序而付出的有意识的直觉上的努力。本书的作者提出了在某种程度上更详细的定义：

- 理解用户的期望、需要、动机和上下文。
- 理解业务、技术和行业上的需求及限制。
- 将这些所知道的东西转化为对产品的规划（或者产品本身），使得产品的形式、内容和行为变得有用、能用，令人向往，并且在经济和技术上可行。

这个定义可以适用于设计的所有领域，尽管不同领域的关注点从形式、内容到行为上均有所不同。

如果采用适当的方法，设计可以为技术产品提供通常所缺少的人性关系。但是，很明显，当前数字产品的设计方法并不是很效，或者不像它们所宣扬的那样有效。

数字产品设计的现状

大部分数字产品的生产都是从工程的角度开发的。诚然，市场部门有时候能够提供需求，但是他们考虑得更多的是竞争对手，为 IT 部门提供一些功能清单，或者基于对用户的调查或消费者愿望列表做一些猜测，而对有关用户实际上需要什么，他们知之甚少。上面这些方法都没有系统地考虑用户的目标。我们很快就会看到为什么目标那么重要。

开发者有着与用户完全不同的职责，他们的关注点在于技术和工程方法学。与此同时，市场部门则关注什么能吸引媒体注意力，关注功能清单和用户的购买愿望。而当用户被直接问到他们所使用的产品时，他们则倾向于关注底层的任务——与你的猜测相反，很少有用户能够意识到或者清晰地阐述他们的目标。

遗憾的是，使用这些设计方法的后果是：软件令人恼怒，不能满足用户需要，生产率也降低了。图 1-1 显示了软件开发过程的演变，以及设计所在的位置，如果还有设计的话。大多数数字产品的开发局限在这个演变过程中的第 1 步、第 2 步或者第 3 步。在这些开发过程中，设计或者不起什么实际作用，或者成为糟糕交互的表面补救措施，正如作者的一个客户所言，就像“猪脸上的口红”。我们马上将要讨论到，为了确保产品真正满足用户需要，设计过程需要位于编码和测试之前。

下面的一些例子很好地解释了为什么只关注技术、市场或者任务（而不是为用户及其目标进行设计）所产生的软件正逐渐被我们所抛弃。

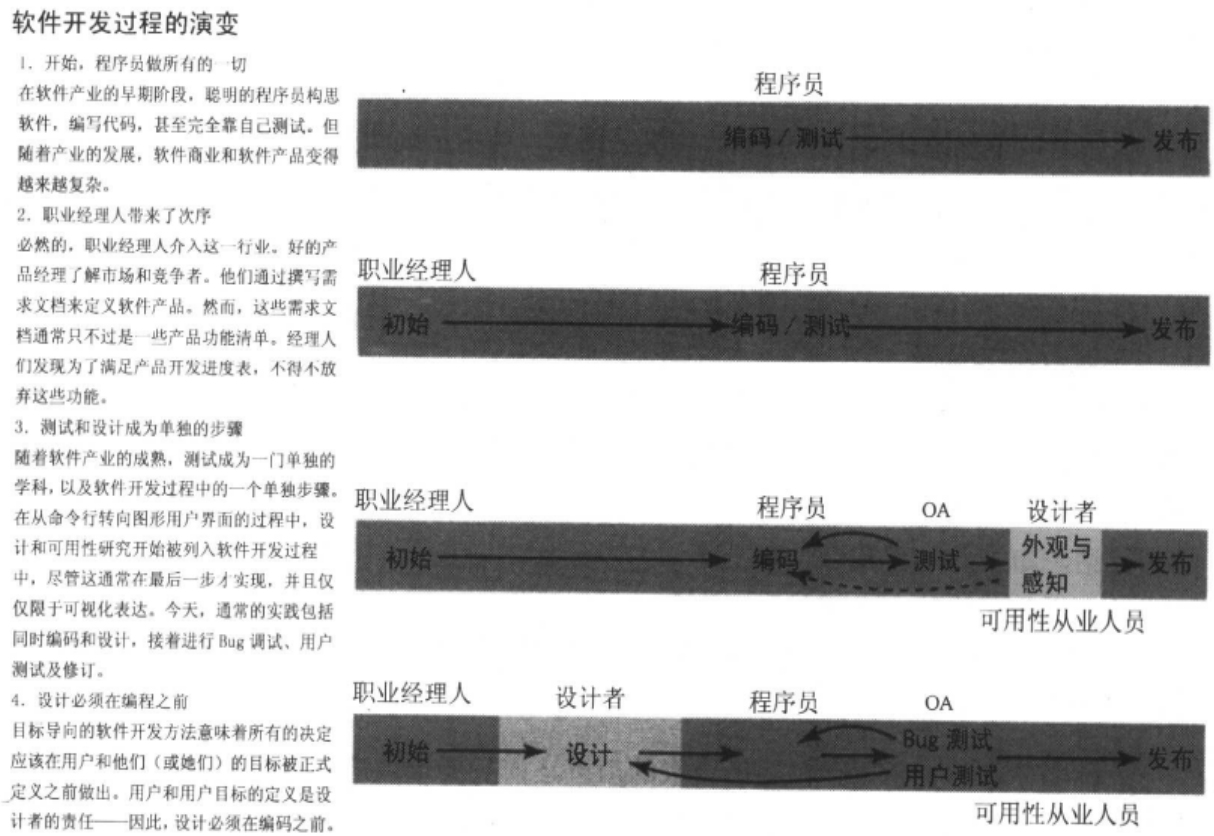


图 1-1 软件开发过程的演变。当前，设计通常只是事后的想法。反过来，它应该位于任何编码和测试之前。

【软件不友好】

软件通常对用户很粗鲁。它们指责用户犯错误，哪怕根本不是或者不应该是用户的问题。如图 1-2 所示，错误消息框像野草一样弹出来宣告用户又失败了。这些错误消息还要求用户用同意的方式承认自己的错误：单击“OK”按钮。

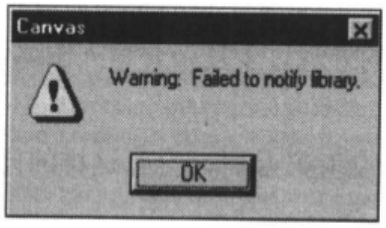


图 1-2 感谢分享。为什么程序没有通知库？它想通知库什么？为什么它要告诉我们？为什么我们要在意呢？总之，我们在 OK 什么？程序失败了，根本就不 OK。

软件经常审问用户，以一串干巴巴的，用户不愿意或者不准备回答的问题来刺激用户，例如“你把那个文件藏在哪里？”那些貌似友好的问题，例如“你确定吗？”及“你真的想删除那个文件，还是因为其他原因按下删除键？”也同样令人感到愤怒，

尊严被贬！

【软件主观臆测】

软件经常主观地认为用户了解计算机。例如，当用户编辑文件完毕时，他关闭文件，然后程序问他是否想要保存。这个问题后面隐藏的技术并不简单，它与 CPU 直接访问随机存储在磁盘介质上的信息的能力有关 一但是，初次使用计算机的人又如何知道这些呢？

【软件很晦涩】

软件经常很晦涩，让用户无法了解它的内容、意图和行为。程序经常使用一些一般用户无法理解的行话来表达自己，例如“有多少停止位”，有时甚至是一些只有专家才使用的行话，如“请指定 IRQ”等。

功能隐藏在菜单、对话框和窗口之后。如果用户找不到帮助系统，他又如何知道答案在帮助系统中呢？即使用户找到了正确的对话框，而他看到的可能只是干巴巴的缩写，晦涩的命令，以及难以理解的图标。

软件经常在不提示用户问题所带来的后果之前，要求用户回答困难的问题，其频繁程度超乎我们的想像。例如，如果用户没有被告知不同的安装在功能和所用磁盘空间上的差异，他们如何能够决定应该选择完全安装、定制安装，还是便携式安装呢？

【软件行为不当】

如果一个 10 岁小孩像这类软件一样做事的话，他可能会被打发进自己的屋子，没有晚餐吃。“程序”像一个 10 岁的小孩那样忘记关门，将鞋子丢在地板中间，不记得五分钟之前告诉过他的事。例如，如果你将一份 Microsoft Word 文档保存，再打印，接着关闭它，此时程序还会再次问你是否想保存。显然，程序以为打印会改变文档，即使实际情况并不是这样。“很抱歉，妈妈，我没听到你说的话”，程序会像一个 10 岁的小孩那样解释？

程序经常要我们退出主要任务流，去处理应急的事情。可能带来不良后果的命令也经常呈现在前端，使得用户很容易不小心触发它。很多程序的外观过于复杂并令人困惑，从而使得导航和理解变得困难。

那么，真正的问题是什么呢？为什么技术产业界在设计交互产品时，会有这样的问题呢？

我们对用户一无所知

这是一个悲哀的事实，数字化技术产业界没有很好地理解什么能让用户感到快乐。实际上，我们在开发大多数技术产品时未能很好地了解用户。我们可能知道我们的用户属于哪一个市场划分：他们能赚多少钱；他们愿意在周末花多少钱；他们愿意买什么类型的车。甚至我们可能模糊地知道他们有什么样的职业，他们经常要完成哪些主要任务。但这些是否告诉了我们如何才能使他们快乐呢？这些是否表明他们实际上会如何使用我们正在开发的产品呢？这些又是否告诉我们他们为什么在做那些可能需要我们的产品提供功能的事呢？为什么他们可能会选择我们的产品，而不是我们竞争对手的产品呢？或者我们又如何能确信他们会这样做呢？不幸的是，这些都做不到。

我们很快就会看到怎样表述理解用户及其产品使用行为的问题。

我们面对利益冲突

这是妨碍开发商和制造者让用户感到快乐的第二个原因。在数字产品开发世界里，存在着严重的利益冲突。开发产品的人——程序员——通常也是设计产品的人，通常需要在容易编码和容易使用这两者之间做出选择。因为程序员的能力通常由能否有效编码，以及能否在紧张而难以置信的最后期限交付产品来判断，因此不难想像大多数基于软件的产品会采取什么样的策略。正如在法律案件中，我们不允许检察官判决案件一样，我们也应该确保产品的设计者不能是产品的开发者。希望程序员对用户、商业和技术一视同仁，是不可能的。

我们缺少软件开发过程

数字化技术产业界不能制造成功产品的第三个原因是它没有可靠的开发过程。或者，更精确地说，它缺乏进行产品开发的完整过程。工程部门遵循或者应该遵循严格的工程方法来确保技术可行性和质量。相似地，市场、销售和其他业务部门遵循他们的方法来确保新产品在商业上的生存能力。这里所缺少的是一个可重复的分析过程，该过程能够将对用户的理解转变为能同时满足用户需求和激发他们想像力的产品。

提到复杂的机械设备时，我们想当然地认为，除了工程化以外，他们已经经过认真设计，以便使用。大多数机械产品很简单，即使是最复杂的机械产品与大多数软件和基

于软件的产品相比也很简单，后者常超过一百万行代码（将它与极其复杂的机械产品，如航天飞机相比，它包括 25 万个部件，而其中又只有一小部分是可以移动的）。然而，大多数软件从来没有采用以用户为中心的严格设计过程。

当前，确定软件的功能以及与用户交互方式的过程完全与软件开发交织在一起。程序员沉醉于对算法和代码的思考，而在设计用户界面时，就像矿工“设计”深坑和残料堆的地形一样。软件界面设计过程要么即兴为之，要么就根本不存在。

今天大多数程序员都信奉这样的想法，经常性地将用户直接吸收到软件开发过程中，如每周，有时候甚至是每天，能够解决设计问题。虽然和用户一起分享设计的责任能起到有益的作用，但它忽略了一个严重的方法学缺陷：混淆领域知识和设计知识。用户虽然可能能够清晰地理解交互问题，但通常不能清晰地提出解决问题的方法。设计是专门的技术，就像编程一样。程序员可能永远不会要求用户帮他编码，设计问题也应该同等对待。

理解如何创建可行的过程，把以用户为中心的设计引入到软件中来，我们需要多了解一些制造业设计的历史，以及交互产品的挑战如何充分地改变了对设计的要求。

制造业设计的演变

在工业制造的早期阶段，工程和市场过程就足以产生令人期望的产品：生产人们愿意购买的锤子、柴油机或者牙膏，好的工艺与合理的价格就足够了。随着时间的推进，消费产品的制造者意识到他们需要在功能相同的产品上与竞争者有所区分。这样，设计作为一种增加用户对产品期望的手段被引入。图形设计师开始介入来创建更有效的包装和广告，工业设计师则创建更舒适、有用和令人兴奋的产品形式。

有意识地引入设计体现了 Larry Keeley 所标志的产品开发需要关注的三个现代要素：可行性、生存能力和期望性（参见图 1-3）。如果在这三个要素中有一个较弱，则产品不可能经受时间的考验。

那么现在考虑计算机，人类所创造的第一个几乎有无限行为能力的机器（只要把这些行为适当编成软件）。这个复杂行为（或者说交互）的有趣特性在于它完全改变了它所触及产品的性质。对于人类来说，交互引人注目，使得交互产品的其他特性变得边缘化而不重要。谁会注意桌子底下的黑箱子呢？交互式的屏幕、鼠标和键盘吸引了人们的注意力。然而，软件和其他数字产品的交互行为，在设计上本应受到最多的关注，却经常被完全忽略了。

开发成功的产品

目标导向的设计方法的前提就是要平衡商业与满足用户需求的制作工程。

你开始这样提问，“人们期望什么？”，接着问到，“在人们的期望当中，哪些能保证商业上的生存能力”。最后，你问到“对应于能够保证商业上生存能力的人们的期望，我们能开发什么产品呢？”常见的问题是人们主要关注于技术，而忽略了商业上的生存能力和期望性。

仅仅理解每个元素的重要性只是刚开始。理解必须付诸行动。我们熟悉商业和技术过程：你创建一个商业模型，接着开发一个商业计划；同样在技术方面，你创建工程模型，接着是工程规范。

目标导向的设计过程类似于规划过程。它产生可靠的用户模型，以及系统的用户交互规划。

用户规划确定顾客购买产品的可能性。业务规划确定业务开始到发展的可能性——从那时以后，销售会支持这种成长。技术规划确定了成功制造产品和成功交付的可能性。

这三种因素的综合作用确定了产品能够成功的总体可能性。

Larry Keeley建议了最初的模型（见上），在这个模型的基础上创建了右边的图。Keeley的模型描述了高技术商业中存在的三个主要品质。

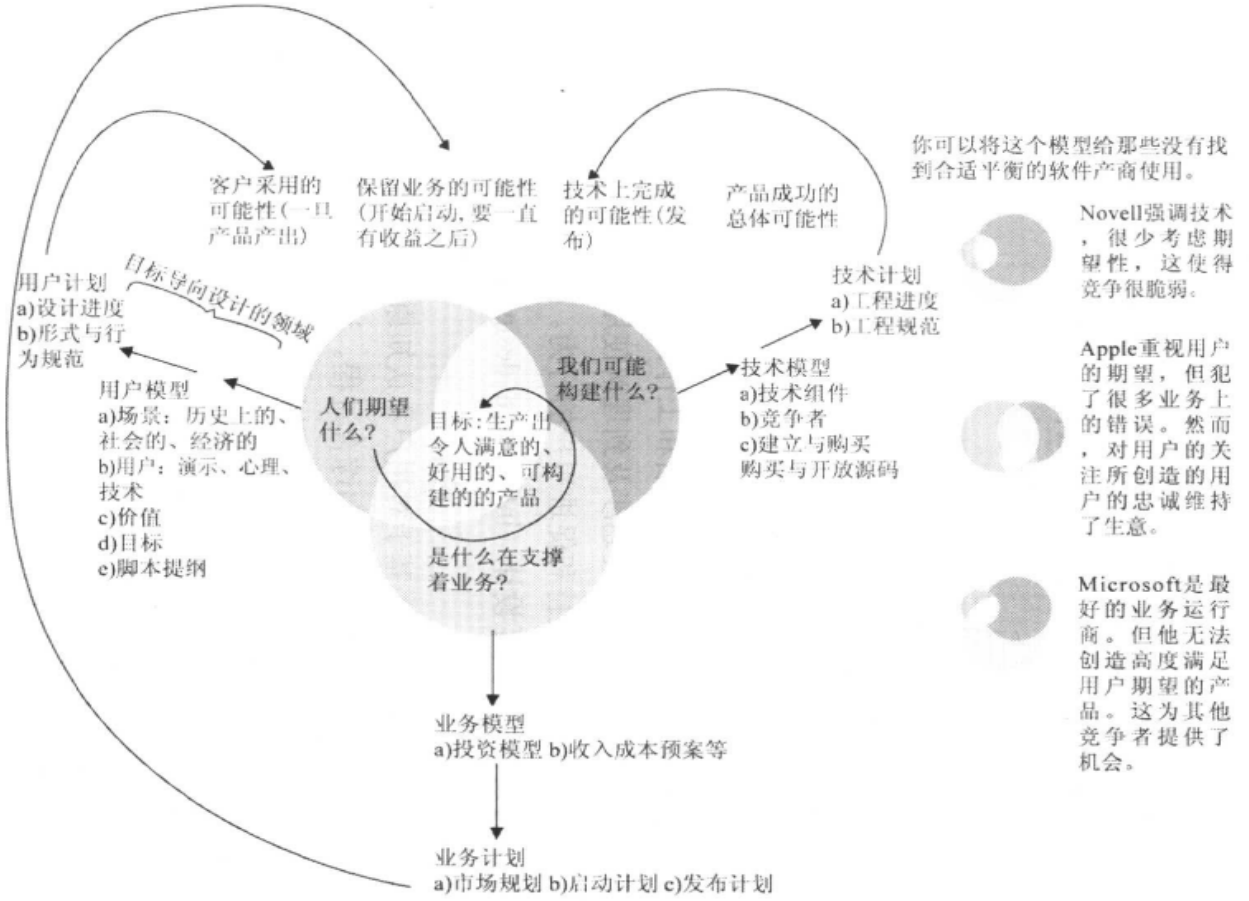


图 1-3　创建成功的数字产品。中间的图是在左边 Keeley 的三角（见左图）基础上扩充的，它显示了开发成功的技术产品需要连续遵从的三个主要过程。本书解决了首要和最重要的问题：如何创建人们期望的产品。

过去，设计传统是企业用来提供产品合意性的重要支柱，但是在交互世界里它并不能提供太多的指导。行为设计是一个不同的问题，需要更多的上下文知识，而不仅仅是视觉组合和品牌的问题。行为设计需要了解从购买到使用的整个过程中用户与产品的关系。最重要的，是了解用户希望如何使用产品，以什么样的方式使用产品，以及在使用产品中希望达到的目的。

行为规划和设计

规划复杂的数字产品，尤其是直接与人交互的数字产品，需要专业设计师先期的大量努力，就像规划与人交互的复杂物理结构需要职业建筑师付出巨大努力一样。对于建筑师而言，规划涉及对于占据这些空间的人如何生活和工作的理解，从而设计出支持和协助这些行为的空间布局。对于数字产品，规划涉及到理解使用这些产品的人如何生活和工作，使设计产品的行为和形式能够支持和促进用户的行为。建筑是一个传统而成熟的行业。产品和系统行为的设计——**交互设计**（interaction design）——相当新，只是近些年来才开始以一个学科出现。

交互设计不是审美学的选择问题，而是基于对用户和认知原理的理解。这是一个好消息，因为它使得行为的设计服从于反复的分析和综合过程。这并不意味着行为的设计可以自动化，也不意味着任意超出形式和内容的设计可以自动化，但它确实意味着系统方法是可能的。形式规则和审美学当然不能放弃，相反，它必须与对其他更重要的、通过合适地设计行为和实现用户目标的关注融为一体。

面向行为的设计表述用户的目标（Rudolf,1998），本书提出了一系列的方法来阐述对这种新的面向行为设计的需要：**目标导向设计（goal-directed design）**。为了理解目标导向设计，我们首先需要更好地理解用户的目标，以及这些目标如何提供了开启设计合适的交互行为之门的钥匙。

识别用户目标

那么什么是用户目标呢？我们如何识别这些目标呢？我们如何知道它们是真实的呢？它们是否对所有用户都一样呢？它们是否随着时间发生改变呢？我们将试着在本章中余下的小节回答这些问题。

用户目标常常与我们猜测的不同。例如，我们可能会猜测一个会计的目标是高效处理发票。这可能不是事实，有效的发票处理更可能是会计雇主的目标。会计的个人目标更可

能是关心自己能否显得胜任本职工作，以及在完成例行的重复任务同时保持这份工作。

不管我们所从事的工作和我们必须完成的任务是什么，我们中的大多数人都有一些简单而共同的个人目标，即使我们有更高的渴望，它们与工作相关的事情相比也显得更加个人化：如获得提升，了解更多的领域知识，或者成为他人的榜样。

设计开发的产品如果只实现商业目标，它们会失败，成功的产品需要满足用户的个人目标。当设计能够满足用户的个人目标时，对商业目标的实现会更加有效，其原因我们会在后面的章节进行更详细的探讨。

如果你仔细研究当前大多数的商业软件、Web 站点及数字产品，你会发现它们的用户界面经常无法满足用户目标并且不停地发出警告。他们反复地：

- 让用户觉得自己很愚蠢。
- 导致用户犯更大的错误。
- 降低用户速度，以至于用户不能完成足够的任务。
- 导航[1]带来的不便让用户无法获得乐趣，且（或）让用户感到厌烦。

大多数这样的软件也同样无法满足业务方面的目标。发票无法得到高效处理，顾客无法准时得到服务，决策无法得到充分的支持。这并不是巧合。

开发这些产品的公司没有正确的优先级。大多数眼光狭隘，只关注实现问题，这转移了他们对用户需要的关注。

即使行业开始关注用户，他们也无法改变他们的产品，因为常规的软件开发过程假定用户界面应该在编码开始之后开发——有时候甚至是在编码结束之后。但正如在开工之后无法设计一幢大楼一样，编码开始以后，很难实现让程序为用户服务的目标（在编码完成之后，更加做不到）。

最后，当公司确实关注用户时，他们却过多地关注用户要完成的任务，而没有充分关注他们完成任务时的目标。在技术上很成功，并能很好地完成每一个任务的软件仍然可能在商业上遭遇失败。虽然我们不能忽视技术或任务，但是在以满足用户目标为目的的设计蓝图中，技术和任务只是其中的一部分。

目标 VS 任务

目标不等于任务。目标是终结条件，相反，任务只是有助于达到目标的中间步骤。目标激发人们去执行任务。不混淆任务和目标非常重要，但它们很容易搅在一起。

---

[1] **译者注** 作者定义的术语，相关内容见第 11 章。

幸运的是，有一个方法可以非常容易区分任务和目标。目标受人的动机驱使，很难随时间的推移而改变，甚至根本不变。任务是短暂的，几乎完全基于身边的技术。例如，当从圣路易斯到旧金山旅行时，一个人的目标可能是快速、舒适和安全。而在1850年，一个移民会选择大篷车旅行，并且，出于安全考虑，他会一直带着他信赖的来复枪。今天，从圣路易斯到旧金山旅行，一个商人会选择喷气式飞机，并且从安全的角度，要求他将防身武器留在家里。目标没有改变，但因为在某些方面完全相反的技术，任务发生了改变。

借助目标，允许你利用技术来消除不相关的任务。例如，如果你要在早晨上班，你的目标是尽可能快而安全地到达目的地。使用今天的技术，要完成的任务意味着，上车，勇敢地开着上路，或者先等火车，然后走一段路。在星际迷航中那样的未来，你可能插上一个开关，通过运输光束物化到办公室里就行了[2]。关注目标同样能帮助设计者消除那些技术上对人来说不重要的任务。

很多开发者和可用性专家通过询问“有哪些任务”来进行界面设计。虽然这样可能能完成工作，但不会产生最好的解决方法，并且经常不能让用户满意。在设计界面时，任务分析很有用，但前提条件是之前已经分析过用户的目标。完全基于任务会让设计陷入由过时技术产生的模型，或者所使用的模型只能满足公司的目标而不能满足用户的目标。通过询问“用户的目标是什么”，我们可以辨明困惑，并且创造更合适和令人满意的设计。

设计要满足上下文中的目标

很多设计师认为，将界面设计得容易学习应该始终是设计的一个目标。容易学习是一个重要的指导原则，但在现实中，正如Brenda Laurel（1990）所提到的，设计目标实际上依赖于具体上下文——谁是用户，他们在做什么，以及他们的目标是什么。你无法通过遵循与你产品用户的目标和需求没有联系的规则来创建好的设计。

让我们考察一下自动呼叫分发系统。该产品的用户基于他们处理了多少呼叫来收费。他们最关注的不是产品容易学习上手，而是分发用户呼叫的效率，以及呼叫完成的速度。容易使用也很重要，因为它会影响用户的心情及最终职员的转换率，所以在设计中应该同时考虑易用性和吞吐率。但毋庸置疑，吞吐率是用户对系统的决定性需求，如果必要，容易使用应该在第二步考虑。在用户学会使用以后，如果程序还让用户一步一步执行，

[2] 译者注　意思是将自己的身体分解为可高速传输的物质，通过光束传播。

只会让用户感到沮丧。

另外一个方面，如果正在设计的产品是公司大厅里帮助来访者寻找路线的公用信息亭，那么，明显地，目标是便于新用户的使用。

好的设计让用户变得更有效率，这是交互设计的通用指导准则，它尤其适用于那些关注生产率的工具。这个指导准则考虑了：用户的通用目标——不要显得愚笨，更具体的商业目标和容易使用等。

设计师应负责如何让用户高效地使用产品。仅能帮助用户完成任务但不能满足目标的软件很少能够让用户高效地工作。如果任务是往数据库里输入 5 000 个名字和地址，仅能顺利完成任务的数据库应用就没有办法像自动从账目系统中提取名字的自动化系统那样满足用户目标。

虽然关注任务是用户的工作，但设计师的工作是超越这些任务，以识别谁是最重要的用户，并且接着确定什么是他们可能的目标，以及为什么是这样。我们将在本章接下来的小节中继续描述整个设计过程，并在第一部分的余下章中进行详细的阐述。这个设计过程提供了回答设计师所关注问题的结构，通过这个结构，能够系统地获得基于这些信息的解决方案。

目标导向的设计过程

对于大多数重技术的公司，即使他们有一个开发过程，也不会有一个充分地以用户为中心的设计过程，就是那些更先进，自认为有着良好过程的组织也会遭遇到一些棘手的问题，这些问题来自传统的研究和设计方法。

近些年来，企业界已经意识到对创建好的产品来说，用户研究非常必要，但很多组织并没有真正了解这类研究的本质特征。定量市场分析和市场划分对于销售产品非常有用，但无法提供用户实际上如何使用产品的关键信息——尤其是具有复杂行为的产品（将在第 5 章更深入地讨论这个主题）。第二个问题出现在分析结果之后：大多数传统方法无法提供将研究成果变为设计方法的手段。几百页的用户调查数据不能轻易地转变为产品需求，传统方法甚至很少提到应该怎样以逻辑的、合适的界面结构来表达需求。设计仍然是一个黑盒问题："奇迹将在此发生"。无法将用户和最终产品联系在一起的设计过程，就导致了研究结果和最终设计方案之间存在差距。我们很快就会看到如何用目标导向方法来处理这类问题。

弥合差距

正如我们已经简略讨论过的，设计在软件开发过程中扮演的角色需要改变。我们需要以新的方式思考设计，做出产品决策。

【作为产品定义的设计】

不幸的是，设计在技术工业领域已经成为一个受限制的术语。对很多开发者和管理者来说，这个单词已经意味着在图 1-1 显示的第三个过程中所发生的事情：**实现模型**（implementation model）中的视觉翻新（见第 2 章）。但是设计在进行恰当应用时（如图 1-1 中所示的第 4 种进程），不仅可以明晰用户需求，而且能给出关于产品行为和外观的详细方案。换句话说，设计基于用户目标、业务需要和技术限制，提供了真正的产品定义（product definition）。

【作为调研者的设计师】

设计一旦能提供产品定义，则设计师需要担任比传统设计更广泛的责任，尤其当设计的对象是复杂的交互系统时。

当前开发过程的主要问题之一是其中角色分配死板：调研者调研，设计师设计（见图 1-4）。可用性和市场调研者分析用户和市场调研的结果，接着又把它们转交给设计师或者程序员。模型中缺少的是能将调研转换和综合为设计方案的系统方法。解决这个问题的一个方法就是让设计师学着成为调研者。

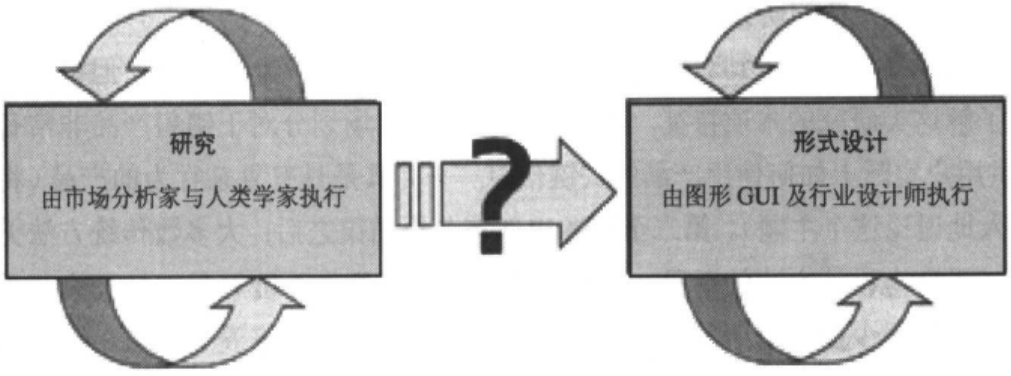


图 1-4　有问题的设计过程。传统上，调研和设计分开了，两种活动由不同的专业人员处理。直到最近为止，调研还主要指市场调研，设计则太局限于视觉设计或表面层次上的工业设计。更近以来，用户调研范围扩大到包括定性的人种学调查数据。然而，如果没有让设计师加入到调研过程中来，调研数据和设计方案之间最多只能保持脆弱的联系。

设计师加入调研过程有一个非常有说服力的理由：设计师最强大的设计工具是同理心（empathy）——感知用户感觉的能力。合适的用户调研需要设计师直接、广泛地深入了解用户，并沉浸在用户的世界中，在提出方案之前长时间地为用户着想。在产品开发中最危险的实践是将设计师与用户隔离，因为这样做会消除设计师的同理心。

另外，让一个纯调研者从设计的角度去了解什么样的用户信息对设计真正重要，非常困难，直接让设计师加入调研就解决了这两个问题。

在作者的实践过程中，设计师接受了第 4 章中所谈到的技术培训，然后，在没有后续支持和协作的情况下进行调研。如果你的团队有时间和资源对设计师进行这类技术的培训，则这将是一个满意的解决方案。如果没有条件，交叉学科的设计师团队和专门的用户调研人员也是适合的。

虽然，设计师从事的调研让我们了解了目标导向设计方法的部分内容，但在调研结果和设计细节之间仍然存在着转化上的差距。正如我们在下面所要讨论的，困惑在于中间还有一些缺失的环节。

【在调研和设计之间：模型、需求和框架】

在目前通用的设计方法中，很少有系统的、行之有效的方法，能够将调研阶段收集的数据转换为详细的设计规范。前面已经解释了这种情形的部分原因，在历史上，设计师不参与调研过程，并且不得不依赖于第三方对用户行为和期望的描述。

另一个原因在于：几乎没有方法能以合适的、指导产品定义的方式捕获用户行为。大多数方法提供的是任务层次的信息，而不是关于用户目标的信息。这类与任务有关的信息在定义布局（layout）、工作流（workflow），以及将功能变换为界面控件这些方面非常有用，但在定义产品的基本框架，它做什么，以及它应该如何满足用户广泛的需求这些方面的作用是有限的。我们需要的是一个清晰的、系统的过程来定义用户模型，确定设计需求，以及将这些需要转换为高层次的交互框架（见图 1-5）。

关于用户的知识必须合成到一个模型中，把用户数据变成设计工具。其他模型，如工作流和环境上下文非常重要，也非常有用，但是合适的用户模型异乎寻常的关键。在创建了用户模型之后，使用模式、心智模型以及通过它们捕获的用户目标，能够系统地映射到交互框架中，而交互框架也表述了在其他模型中捕获的业务和技术规则。

通过将新技术和已有的方法更有效地组织在一起，目标导向设计力求弥合当前数字产品开发过程中的用户调研和设计之间的差距。

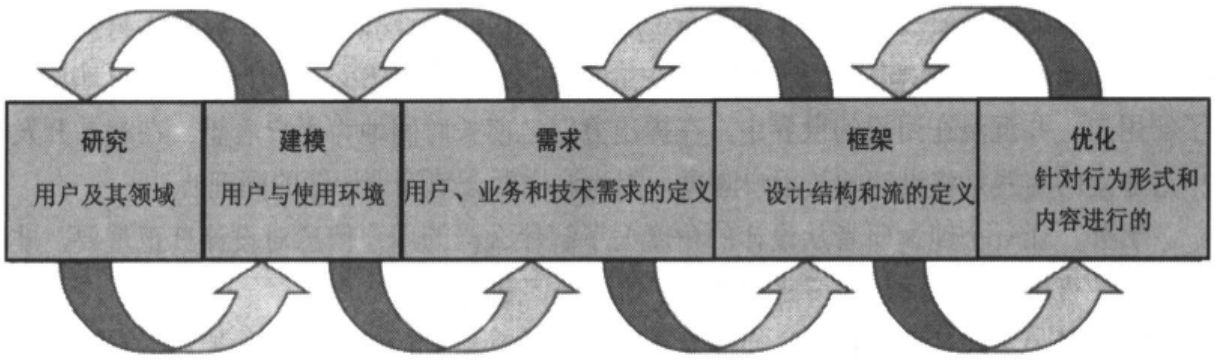


图 1-5 弥合调研和设计之间的差距。差距的弥合主要由三个活动完成。把调研结果综合为设计工具的建模过程，在模型的基础上综合和定义需求的过程，以及将这些模型和需求中捕获的知识转换为设计框架的过程。这些设计框架反映了用户的目标和需要，同时也包括了业务和技术上的规则。

过程概况

目标导向设计组合了多种技术，人类学调查、涉众访谈（stakeholder interview）、市场调研、产品/文献调研、详细的用户模型、基于脚本提纲[3]的设计，以及一些核心的交互模式和原则。它提供了满足用户需求和目标的解决方案，与此同时也表述了业务/组织和技术上的规则（参见图 1-6）。这个过程基本上可以分为五个阶段：调研、建模、需求定义、框架定义和优化。这些阶段遵循了 Gillian Cramption Smith 和 Philip Tabor（1996）指出的交互设计的五个子活动——理解、抽象、结构化、描述和优化——其中强调了为用户行为建模和对系统行为的定义。

本章中以下各小节介绍了目标导向设计五个阶段的高层次视图，具体情况如图 1-6 所示。第 4，5 和 6 章从更为面向过程的角度论述了各个阶段中涉及到的方法。

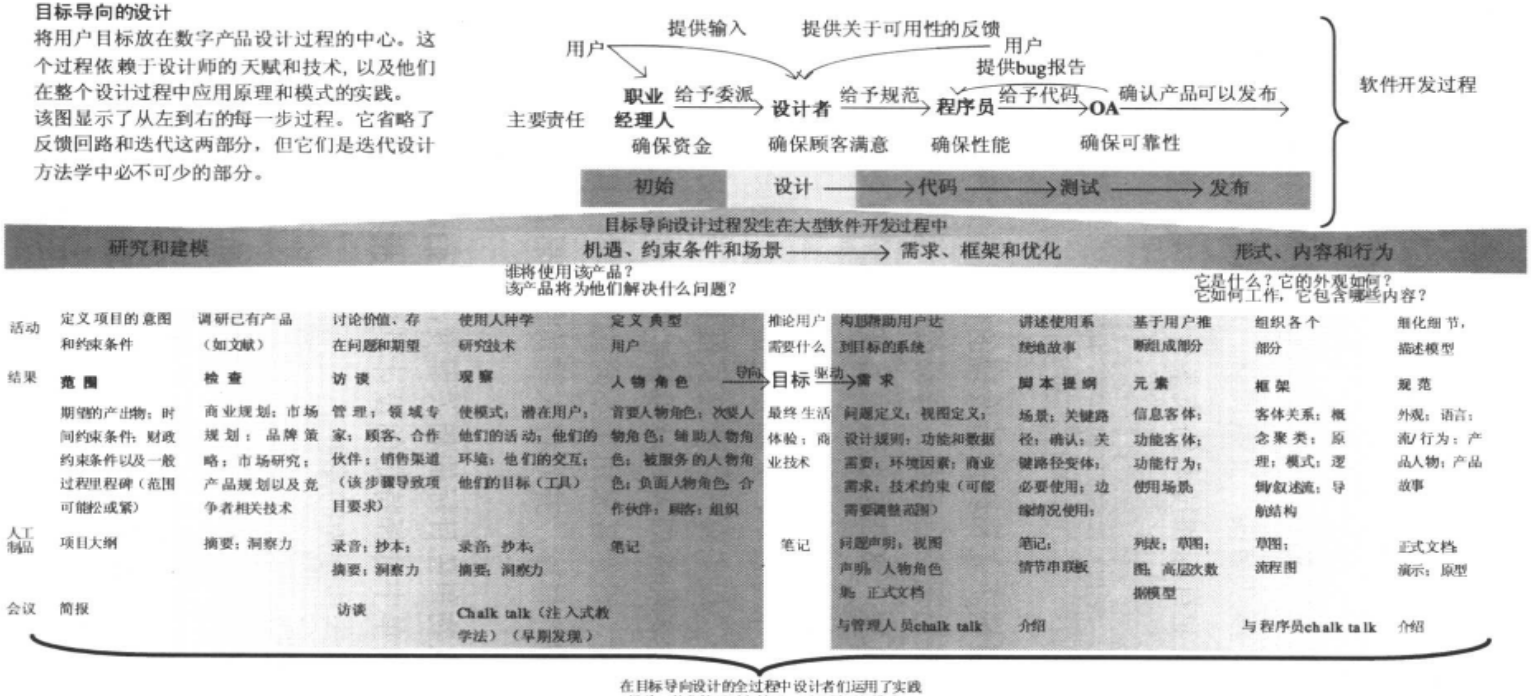
【调研】

调研阶段使用了人种学现场研究（ethnographic field study）技术，观察与场景相关的访谈提供产品潜在用户和实际用户的定性数据。它也包括对竞争产品的审核、市场研究和技术白皮书的回顾，以及与涉众、开发者、学科专家（Subject Matter Experts, SME），以及特定领域技术专家的一对一访谈。

[3] **译者注** 脚本提纲：作者专门定义的“目标导向设计”术语。

目标导向的设计

将用户目标放在数字产品设计过程的中心。这个过程依赖于设计师的天赋和技术，以及他们在整个设计过程中应用原理和模式的实践。该图显示了从左到右的每一步过程。它省略了反馈回路和迭代这两部分，但它们是迭代设计方法学中必不可少的部分。



设计团队实践

设计组织创建和运做的方式。提到和未提到影响工作的规则。关键实践包括：

团队结构和协作；
“忘我”的设计；
合适分配任务；
软件过程的演变和加强；
评估和自我评估

设计师的天赋和技巧

设计师天生的能力和背景也影响着工作。对交互设计师来说关键的技巧包括：

想像力；概念技巧；可视化技巧；写作技巧；口头表达技巧；移情作用能力；人际关系能力；脑力风暴技巧；分析技巧；咨询技巧；

设计原则

原则指导设计师在创造时做出选择。原则适用于从概念沉思到小的细节不同层次的设计

不要伤害（希波克拉底）；改善人类状况；帮助实现用户目标；适应用户能力；提供最简单的完整方案；提供一致的解决方案；合适地刺激认知和感情；提供具有商业生存能力和可行的解决方案；

设计模式

设计模式是设计师在分析阶段，尤其是综合阶段能够识别或者应用地反复出现的形式和结构。Christopher Alexander《模式语言》一书为建筑结构提供了模式范例。软件交互的模式也必须被收集和分类。例如，将窗口分为相邻地两个垂直窗格的通用模式：左边较小的窗格提供工具和环境。而右边较大的窗格提供工作空间或者细节

图 1-6　目标导向的设计过程

现场观察和用户访谈的主要成果之一是产生一系列的使用模式——这些可识别的行为有助于对潜在的和已有产品的使用方式进行分类。这些模式暗示着目标和动机（使用产品时所期望的结果，包括特定的和一般性的结果）。在商业和技术领域，这些行为模式倾向于映射为职业角色。对于消费产品，它们更可能对应于生活方式的选择。使用模式和相关的目标促使了建模阶段**人物角色**[4]（**Personas**）的创建。市场研究帮助选择和过滤适合公司商业模型的有效人物角色。涉众访谈、文献调研、产品审核加深了设计师对领域的了解，阐明了设计必须支持的商业目标和技术约束条件。第 4 章更详细地讨论了目标导向的研究技术。

【建模】

在建模阶段，我们把通过分析现场研究和访谈所发现的使用模式和工作流模式综合到领域模型和用户模型中。领域模型包括信息流和工作流程图。用户模型，或者说人物角色，是非常详细的、合成的用户原型，这些原型代表着在研究阶段观察和识别到的不同行为模式、目标和动机。

在叙述性的、基于脚本提纲的设计途径中，人物角色是主角。这一途径在框架定义阶段迭代地产生设计理念，在优化阶段提供有关设计连贯性和恰当性方面的反馈，并作为一个强有力的交流工具来帮助开发者和管理者理解设计动机，以及基于用户需求对产品功能进行优先级排序。在建模阶段，设计者采用了不同的方法学工具来对人物角色进行综合、区分和排序，挖掘不同类型的目标，并且将人物角色映射到不同的行为范畴，以确保没有空白或重复。

基于每个人物角色目标和其他人物角色目标的相关程度，我们通过比较其目标并且赋予它们不同的优先级从人物角色清单中选择特定的设计目标。指定人物角色类型的过程确定了每个人物角色对最终设计形式和行为的影响。

可能的用户人物角色类型指派包括：

- 首要人物角色——人物角色的需求非常独特，需要特别的界面形式和行为。
- 次要人物角色——主要界面通过轻微修改或补充以满足人物角色需求。
- 补充人物角色——该人物角色的需求完全可以由主要界面满足。
- 所服务的人物角色——人物角色不是产品的实际用户，但受产品及其使用的间接影响。
- 负面人物角色——创建的人物角色作为具有修饰色彩的显式样例，表明产品不为

[4] **译者注** 人物角色：作者定义的专门术语。

谁设计。

有关人物角色和目标开发的详细讨论见第 5 章。

【需求定义】

在需求定义阶段，团队所采用的设计方法提供了用户与其他模型之间紧密的联系和设计的框架。这个阶段采用基于脚本提纲的设计方法（Carroll, 1995），其重要突破在于脚本提纲不是关注抽象的用户任务，它首要关注的是满足具体用户的目标和需求。从人物角色中可以得到哪些任务是真正重要的及原因，这样能得到的界面同时具有最小化的必需任务（努力）和最大化收益。人物角色成为这些脚本提纲中的主要角色，设计师通过人物角色的形式探索设计空间。

对每一个界面 / 首要人物角色来说，在需求定义阶段中的设计过程中要分析人物角色数据和功能性需求（用术语“对象”、“行为”和“上下文”来表达）。在不同的上下文中，基于人物角色目标、行为及与其他人物角色的交互来对这些数据和需求进行优先级排序。

这种分析通常是一种迭代地优化**上下文脚本提纲**（context scenario）的过程，从人物角色使用产品的日常生活开始，描述高层次的产品接触点，到持续定义不断深化的细节。随着这种迭代的发展，业务目标和技术限制也要同时考虑进来，并且要与人物角色的目标和需求平衡。这个过程的产物就是**需求定义**（requirements definition），它平衡着设计时需要遵从的用户需求、业务需求和技术需求。

【框架定义】

在框架定义阶段，除了场景脚本提纲以外，团队还使用两个与它有关的重要系统方法来合成一个**交互框架**（interaction framework）。第一个是一套通用的**交互设计原则**（interaction design principle），如视觉设计中的有关原则（参见第 19 章），提供了在多种场景中定义合适系统行为的指导原则。第 2 和第 3 章及整个第二部分专门用于描述适用于框架定义阶段的高层次交互设计原则。

第二个重要的方法学工具是一套**交互设计模式**（interaction design pattern），对前面已经分析过的不同问题，它给出了通用解决方案（基于场景的不同而有所变化）。这些模式类似于 Christpher Alexander（1977）发展的建筑结构设计模式。交互设计模式是按照层次组织的，并且随着新场景的出现而不断发展。它们用已经证明的设计知识来解决问题，经常为设计带来便利，而并没有限制设计师的创造力。

在这个高度上描述了数据和功能性需求之后，按照交互原则，把它们转变为设计元素，然后再按照模式和原理，又将它们组织为设计草图和行为描述。这个过程的产物是

交互框架的定义（interaction framework definition）——一些稳定的设计概念——它们为后续设计细节提供了逻辑上的和总体的形式结构。接下来的持续迭代关注范围更窄的场景，并在优化阶段提供这些细节。这种方法在自上而下（面向模式）和自下而上（面向原则）的设计之间有一个折中。

【精化】

精化阶段类似于框架定义阶段，但更多地关注任务一致性。这一阶段使用**关键路径**（**key path**）（预排）和确认脚本提纲[5]，通过更详细的界面关注情节串连图板。精化阶段的最终成果是详细的设计文档、**形式和行为规范**（**form and behavior specification**），这一文档是按照上下文要求，以书面文字或者交互媒体方式来记录的。第 6 章更加详细地讨论了需求定义、框架定义和精化这三个阶段中的人物角色、脚本提纲，以及原则和模式的使用。

产品成功的关键是目标，而不是特性

程序员和工程师——对技术感兴趣的一类人——有着从功能和特性的角度考虑产品的强烈倾向。这很自然——因为这正是程序员创建软件的方式：一个功能接着一个功能。问题在于这不是用户想使用产品的方式。

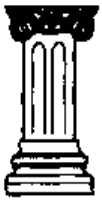
确定某个功能是否应该包括在产品中不应该取决于它的实现技术。背后的决定因素永远不能只是我们拥有完成这项功能的技术能力。决定性因素应该是这一功能是否直接或者间接地帮助实现用户目标，与此同时仍然满足业务需要。

即使在产品开发周期中的压力和混乱下，成功的交互设计师也应该对用户目标高度敏感。虽然我们在本书中讨论了其他很多技术和工具，但始终会回到用户目标。它是交互设计实践的基础。

目标导向的过程，还有它在设计决策时清晰的理论基础，能更容易地说服工程师，能让市场和管理部门参与进来，能确保正在考虑的设计不是凭空猜测，或者仅仅是团队成员个人偏好的反映。

---

[5] **译者注** 脚本提纲的一种，见第 6 章。



公理 交互设计不是凭空猜测。

目标导向设计也是回答在定义和设计数字产品过程中出现的最重要问题的强有力工具：

- 如何发现用户？
- 如何知道哪些是用户试图完成的事？
- 产品应该如何工作？
- 产品应该采取什么形式？
- 用户将如何与产品交互？
- 如何能最有效地组织产品的功能？
- 产品以何种方式面向首次使用的用户？
- 产品如何体现在技术上易理解和易控制？
- 产品如何处理用户问题？
- 产品如何帮助比较生疏的用户变得更熟练？
- 产品如何为专家级用户提供足够的深度？

在本书的余下部分，我们将帮助您回答这些问题。您会得到您所需要的工具来识别产品的关键用户，理解他们及其目标，以及通过使用目标导向设计将这些理解转变为成功的设计方案。

2

实现模型和心智模型

计算机界经常使用**计算机文化**（computer literacy）这个术语。权威人士经常谈到一些人具备这种文化，而一些人则没有；具有这种文化的人又如何会在信息经济中取得成功，而缺乏计算机文化的人则不可避免地落入社会经济学的断层中。然而所谓的计算机文化只不过是一种委婉的说法——让用户极力去理解计算机奇怪的逻辑，而不是让基于软件的产品灵活地适应用户的思考方式。

在本章中，我们将讨论不了解用户及其使用数字产品的方式如何加速了计算机文化鸿沟的产生，而符合用户思考和工作方式的软件又将如何帮助解决这些问题。

实现模型

任何机器都有实现其目标的机制。例如，电影放映机使用复杂的移动图片序列来创建这种动态的感觉。它在一个瞬间让明亮的光线透过半透明的微缩图像，然后在它移向另外一副微缩图像的瞬间挡住光线，接着在下一个瞬间再次投射光线。电影放映机每秒24次放映新的图像，重复这个过程。基于软件的产品并没有这样的机制，取而代之，它们使用算法和相互通信的代码模块。这种有关机器和程序如何实际工作的表达被 Donald

Norman（1989）和其他人称为**系统模型**（System model）。作者更愿意使用**实现模型**（implementation model）这个术语，因为它描述了程序在代码中实现的细节。

用户心智模型

从电影观众的角度来说，在观看一部引人入胜的电影时，很容易忘记影片齿孔间的细微差别和瞬间的光线打断。很多看电影的人，实际上根本不知道放映机如何工作，或是它工作的方式与电视有什么不同。在观众的想像里，放映机只不过是发射出在大屏幕上移动的图片而已。这就称为用户的**心智模型**，或者**概念模型**。

人们并不需要知道复杂产品的实际工作细节来掌握它的使用方法，为了便于使用，人们在认知上创建了一种简捷的解释方式，这种方式对他们与产品的交互来说已经足够了，但并不一定能够反映产品实际的内部工作机制。比如，很多人想像，当他们将真空吸尘器和搅拌机接到墙上的电源插座时，电流会像水一样通过黑色的小电线管从墙里流向电器。这种心智模型很适合于家用电器。但实际上家用电器的实现模型并不涉及到什么液体类的东西在电线中流动，而是电压每秒钟反转 120 次。尽管电力公司需要知道这些细节，但这些细节与用户无关。

然而，在数字世界里，用户的心智模型和实现模型通常差别很大。我们经常忽略像“手机的工作方式不同于固定电话”这样的细节。相反，它实际上是一个在两分钟通话的过程中，要在半打不同的基站天线中交换连接的无线电接收器。了解这些细节并不会有助于我们对它的使用。

对于软件应用来说，实现模型和心智模型之间的差异非常明显。在这种情况下，复杂的实现使得用户不可能看到用户行为和程序反应之间存在的机械关系。当用户使用计算机编辑数字声音或者创造视频特效，如变形时，无法在机械世界中找到类似的例子，所以我们的心智模型有必要和实现模型不同。即使这种关系有可能是可见的，但它们可能对大多数人来说还是无法理解的。

表现模型

软件显示给外界的行为外表是由程序员或者设计师创造的。这种表达并不一定精确地描述了软件实际上在计算机里如何工作，虽然很不幸的是它经常如此。与其他媒介相比，这种独立于计算机的真实行为来表达其功能的能力，在软件世界里更加明显——它允许聪明的设计师隐藏关于软件如何真正完成工作的细节。在数字化世界里，“实现了什

么”和“提供了什么”之间缺乏的联系带来了数字化世界里的第三种模型，设计师的**表现模型**（represented model）——设计师选择如何将程序的功能展现给用户的方式。Donald Norman（1989）简单地称之为**设计师模型**（designer's model）。

在软件世界里，程序的表现模型可以（也常常应该）与程序的工作结构迥然不同。例如，操作系统能够使网络文件服务器看起来像本地磁盘一样。这种模型并没有表现出物理磁盘可能在数里之遥的事实。在机械世界里，表现模型的概念并不存在类似事物。这三个模型的关系见图 2-1。

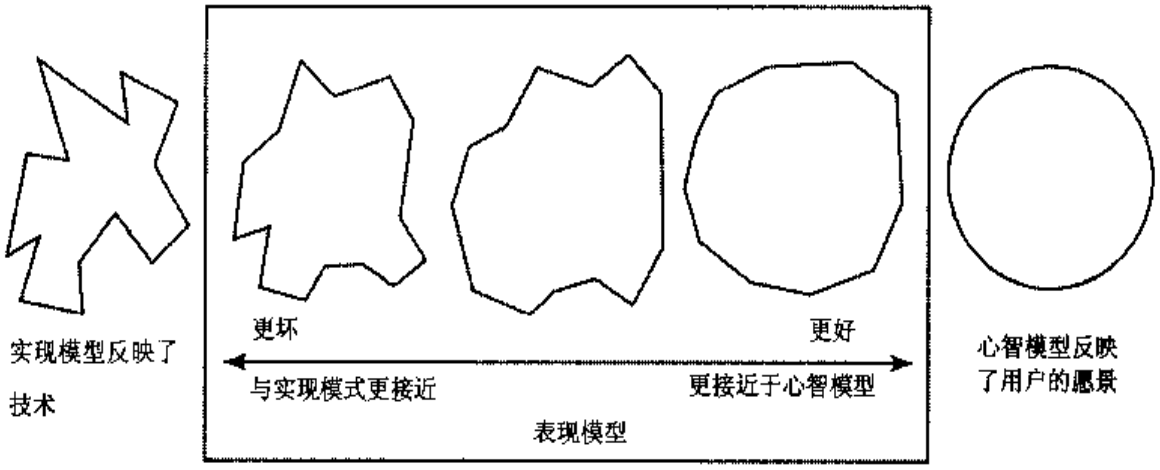


图 2-1　工程师开发软件的方式通常是给定的，并且常常受制于技术和业务上的限制。有关软件实际上如何工作的模型称为实现模型。用户与软件交互的心智模型是用户理解他们所需要进行的工作，以及程序如何帮助他们完成工作的过程。这种模型基于他们自己如何完成任务，以及计算机如何工作的想法。设计师选择如何将软件工作机制表现给用户的方式称为表现模型。表现模型和其他两个模型不同，它是设计师能够极强地对软件进行控制的体现。设计师一个很重要的目标是使表现模型和用户的心智模型尽可能相互匹配，因此设计师能否详细地理解目标用户所想到的软件使用方式，非常关键。

表现模型离用户心智模型越近，用户就会发现程序越容易使用和理解。一般来说，假如（通常也如此）用户有关任务的心智模型不同于软件的实现模型，向用户提供过分接近实现模型的表现模型会严重地降低用户学习和使用程序的能力。

我们倾向于形成比现实更简单的心智模型。如果创造了比实际实现模型更简单的表现模型，我们就能帮助用户更好地理解。踩下汽车的刹车踏板，可能让你想到按下控制杆，摩擦车轮来降低车速。而实际的机制包括水压柱、管道和金属垫板挤压一个穿孔的圆板。为了简化事实，我们创造了更有效但不精确的心智模型。对于软件来说，我们想像：当单击滚动条时，电子表单软件将新的单元格显现在视图里。实际上没有单元表格，

只不过存在一个紧凑的数据结构，以及它们之间的指针，在这个基础上程序实时合成一个新的图像来显示而已。

理解软件如何实际工作常常有助于人们使用它，但这种理解需要很大的代价。表现模型允许软件创造者通过简化软件透明的工作方式来解决问题。这种成本完全是内部的，用户永远不必知道。抛弃了实现模型而更接近用户心智模型的用户界面更好。

公理　用户界面应该避免实现模型，而支持用户心智模型。

在 Adobe Photoshop 里，用户使用称为变化（variation）的功能来调整图像的色彩平衡和亮度。在变化功能的界面中，不是提供输入颜色值（在实现模型中所需要的）的数值框，而是显示一系列代表不同色彩平衡（color balance）的微缩图像（如图 2-2 所示）。用户通过单击这些图像来选择他们所想要的色彩设置。这种界面更好地符合用户的心智模型，因为用户——可能是一个图形艺术家——会以他们的图像看起来像什么来思考，而不是抽象的数字。

如果软件的表现模型非常符合用户的心智模型，通过提供能使用户很容易理解他们的目标和需要如何得到满足的认知框架，消除用户界面中一些不必要的复杂性。

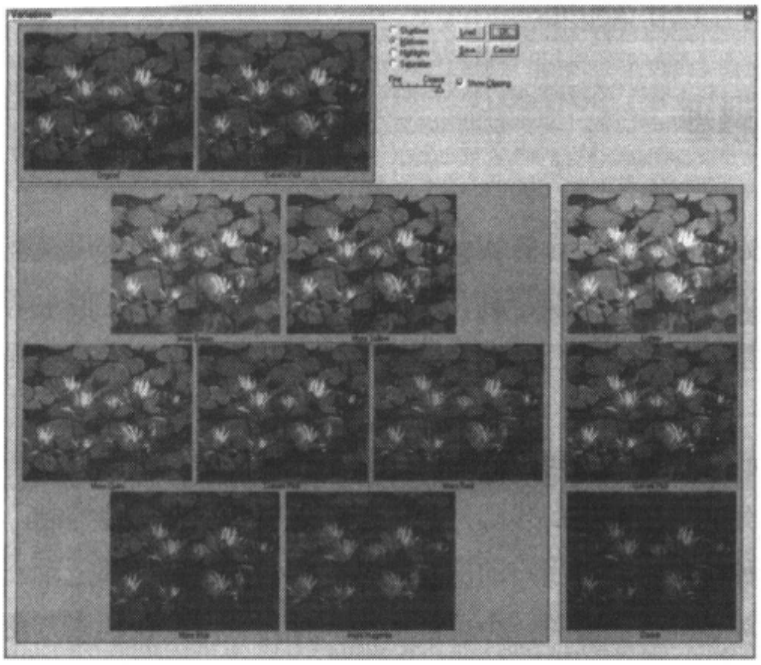
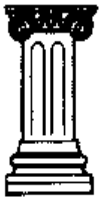


图 2-2　Adobe Photoshop 有大量的匹配用户心智模型的软件设计的例子。调整功能的界面显示了一组微缩的图像，通过可调整的增量来变化色彩平衡和亮度。用户能够单击最接近他们期望的颜色设置的图像。这个图像成为更多变化的微缩图像的新的默认值。界面遵循追逐特定效果的图形艺术家的心智模型，而不是使用一些抽象的数字。



公理：目标导向的交互反映了用户的心智模型。

用户的心智模型并不一定要真实或者准确，但它应该能够让用户更有效地工作。例如，大多数非技术类的用户想像计算机屏幕是它的心脏。这很自然，因为计算机屏幕是他们一直盯着，并且能够看到计算机在做什么的地方。如果你向用户指出，计算机的心脏实质上是他们桌子下面黑箱子里面很小的硅芯片，他可能会耸耸肩，并且忽略这些对他们来说没有意义的信息。CPU 不同于视频显示器的事实并不能帮助他们思考与计算机交互的方式，即使这在技术上这是更准确的概念。

大多数软件遵从实现模型

很容易设计反映实现模型的软件。一个按钮代表一个功能，一个域代表一个数据输入，一个页面代表一个事务步骤，一个对话框代表一段代码模块，这在软件世界里屡见不鲜。尽管这通常反映了工程实践的基础，但并不能提供对用户目标及用户要达到这些目标需要完成的任务的一致反映。这样的界面对用户的影响就像 Terry Gilliam 在 1984 年左右拍摄的电影 Brazil（充满着半开玩笑的悲惨画面的电影）中无处不在的外部管道系统一样，除了疏远和困惑用户之外，没什么用处。

工程师设计的界面遵从实现模型

工程师了解软件精确的工作方式，他们设计的软件界面，经常产生与程序实现模型一致的表现模型。对于工程师来说，这些模型非常具有逻辑性、真实并且精确；但不幸的是，对于用户来说这些界面不是很有效，并且没有多大帮助。大多数用户并不在意程序实际上是如何实现的。

在遵从实现模型而不是用户心智模型设计用户界面时，基于软件的传真产品就是一个非常好的例子。发传真的过程分成很多细碎的步骤，每一步都需要用户费劲地控制，而从用户的角度来说根本没有这种必要。它与用户的交互就是完全以与软件内部逻辑一致的方式呈现的。用很多单独的对话框代表着每一个可能的用户行为。用户在程序方便接收信息的时候收到提示——而不是在很自然由用户提供的时候。

如果想发送传真，我们需要将它发送到先前已经在程序里输入姓名的人，或者新的还未在程序中输入的人。但问题是程序遵从实现模型：完成每一个功能的代码散落在不同的模块中，所以程序在不同的对话框里显示这些功能。为了选择或输入姓名，我们必须将主程序划分为两种方式，尽管选择或者输入姓名是程序的主要功能之一。设计师没有设想用户可能采取的传真创建和发送步骤，而是在完成程序不得不做的事。

解决这种问题有一种更好的方式：无论什么时候，当我们输入新的姓名和传真号码时，程序自动记录它们。程序的主窗口显示先前已经输入过的收发传真人列表，允许我们从中快速选择，如果我们想这样做的话。

即使是 Windows 界面有时候也偏向实现模型[1]。浏览器试图以统一的模式显示整个系统的存储设备，而且要成功地与用户交流，它们的行为也必须统一。但实际上，它们的行为依赖于特定存储设备的物理性质。如果你在同一个硬盘的不同目录之间拖放文件，程序将这解释为移动，这意味着从老的目录中删除文件，并将文件添加到新的目录，和心智模型一致。然而，如果你把文件从硬盘 C 拖到硬盘 D，这种行为解释为拷贝，意味着该文件将添加到新的目录，但不会从旧的目录中删除。

这与实现模型（底层文件系统实际工作的方式）是一致的。当操作系统将文件从一个目录移动到同一硬盘的另一个目录时，它只不过重新定位了磁盘内容表中的文件项，实际上绝不会删除和重写文件。但当操作系统将一个文件移动到另外的硬盘时，它必须物理地拷贝该文件到新的硬盘上。要符合用户的心智模型，它应该删除原始内容，即使这与实现模型不一致。

因为将文件从一个硬盘拖放到另外一个硬盘解释为拷贝可能是大家期望的行为，尤其当从一个硬盘拷贝到另外的可移动媒体诸如 ZIP 硬盘时，所以很多人没有意识到这是与实现模型不一致带来的副作用。随着计算机的成熟，当逻辑卷代表的不仅仅是物理硬盘时，这种副作用也会不再有用了，相反还会令人恼怒，因为我们不得不记住每个卷类型的特殊行为。

数学思维方式产生实现模型的界面

交互设计师需要向用户屏蔽实现模型。仅仅因为技术适合解决软件构造中的问题，并不意味着它适合成为用户的心智模型。仅仅因为你的车子由焊接的金属块构成，并不意味着你必须要学会焊接点火的方式才能开车。

---

[1] **译者注** 以微软 Windows 浏览器的实现来讨论完全遵从用户心智模型的重要性。

在处理和表达软件中的信息时，所用的大多数数据结构和算法是基于数学算法的逻辑工具。所有的程序员都熟悉这些算法，包括递归、层次数据结构和多线程。当用户界面试图精确地表达递归、层次数据或者多线程时，问题就出现了。

数学的思维方式使程序员特别容易掉入实现模型的陷阱。程序员通过数学思考来解决编程问题，所以他们自然将这些数学模型当做构造用户界面的合适术语。这与事实相差甚远。

设计提示：用户不理解布尔逻辑。

例如，程序员的工具箱里最耐用和最有用的工具是布尔（Boolean）代数。它是一个非常紧凑的数学系统，通常用于描述数字计算机中严格 ON/OFF 的行为。它只有两个主要的操作：AND 和 OR。问题在于，英语中恰好也有 AND 和 OR，而它们的意思在普通人看来——非程序员的那些人——又正好与布尔代数中的相反。如果程序用布尔符号来表达，可以想像用户一定会误解。

例如，在查询数据库时，这一问题频频出现。如果想从文件中提取居住在 Arizona 和 Texas 的员工，我们会用语言表达为："寻找居住在 Arizona 和 Texas 的职员"，而根据数据库的布尔代数术语，我们会说："寻找居住在 Arizona OR Texas 的职员"。没有人同时生活在两个地方，所以"寻找居住在 Arizona AND Texas 的职员"是没有意义的。在布尔代数里，这只会产生空的答案。

因此，数据库查询程序或者其他用布尔函数与用户交互的程序注定会遇到严重的用户界面问题。期望用户来解决这些混乱是不合理的。既然能熟练使用自然语言，为什么一定还要他们用另一种不熟悉的语言来表达并重新定义一些关键词呢？

机械时代的表现模型与信息时代的表现模型

从工业化的机械产品到数字化信息对象，我们正经历着一场令人难以置信的转变。这种转变刚刚开始，并且步伐正在迅速加快。与信息时代相比，工业化产生的社会巨变可能会相形见绌。

机械时代的表现方式

对于我们来说，很自然地将我们所习惯的先前时代的语言和比喻尽量带入不确定的新时代，正如工业革命的历史所显示的，新技术的成果刚开始经常只能用更早的技术语言来表达。例如，我们曾把火车机车称为铁马，把汽车称为无马的马车。不幸的是，这种比喻和语言严重歪曲了我们的思维。

在新的环境里，我们自然倾向于采用旧的表达，有时候，这种做法是允许的，因为功能还是相同的，即使底层技术不同了。例如，当我们不再使用打字机打字，而改用计算机上的字处理软件时，我们采用的是相同任务的机械时代表现方式。打字机使用小小的金属制表键，在有多处空格时快速跳转直至达到特定的栏。这种过程，随着技术的自然发展，称为 tabbing 或者设置 tab。字处理软件也有 tab 键，因为它们的功能相同。无论卷在滚筒上的纸，还是在视频的图像上工作，你都需要快速地跳转到特定的边距偏移位置。

然而，有时候，机械时代的表现方式不应该完全转换到数字时代的对应物上。我们不使用缰绳或者舵柄来控制汽车，虽然这两者都在汽车时代的早期曾经使用过。而我们花了很多年才得到适合汽车的操纵用语。在字处理软件中，在填满一页后，我们不需要加载新的空白页面，而是让文档持续地滚动，并在两页之间标以明显的记号。

新技术需要新的表现方式

有时候，仅仅因为新技术使得某些任务、过程或者概念甚至目标的第一次出现成为可能。因为它们刚刚发明，所以人们没有预先想像过。例如，当电话机刚刚发明出来时，和其他很多东西一起，被吹捧为对音乐和新闻进行广播的手段，尽管它最后成为最流行和广泛发展的个人通信手段。那时候的人们没有想到电话会成为无处不在的个人工具，人们将它放在口袋和钱包里，甚至观看在戏剧演出时，它也发出烦人的声音。

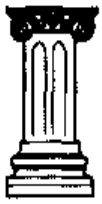
以我们机械时代的观念体系，一开始我们需要经历一个艰难的时期才能找到合适的信息时代表现方式。直到有合适的用户群，才能看到我们创造的软件产品的实际用处。例如，电子邮件的实际好处并不在于它比邮件快——机械时代的观点，它真正的好处在于促进了现代商业组织的扁平化及民主化过程——信息时代的观点。Web 的实际好处并不在于便宜或者更有效的通信和发布——机械时代的观点，它的实际好处在于创造了许多虚拟社群——信息时代的观点，这是它在我们的观念里充分物化以后才显现出来的。因为要经历一个艰难的时期才能看到人们如何使用数字产品，我们过分地倾向于依赖过

去在机械时代的表现方式。

机械时代的表现方式有损用户交互

当我们将一些熟悉的机械时代的表现方式用于计算机时，会遇到一些问题。简而言之，机械时代的过程和表现方式会有损信息时代产品的用户交互。例如，使用计算机在信笺上打印地址与用笔和墨水在信笺上写明地址相比，要花费更多的成本（虽然，前者看起来更整洁）。仅仅当这个过程大批量自动化后，情况才能得到改善——如你需要在500个信封上注明地址。

另外一个例子是计算机上的通讯录。如果它在计算机中显示为一个便笺，与现实中的地址簿相比，它会更加复杂，不方便，而且难以使用。比如说，实际的通讯录以姓的第一个字母的顺序来存放名字。但如果你想使用名字的第一个字母来查找姓名呢？机械时代的产品无法帮助你——你不得不逐页地进行人工扫描。同样对于完全复制而成的数字化版本来说，你也不能用名字的第一个字母进行搜索。而且在计算机屏幕上，你不可能得到书本方式那种精美的视觉暗示（折起来的页角，铅笔做的注释等）。与此同时，滚动条和对话框更难以使用和浏览，与简单的页面相比，也更难以理解。



公理：不要只在界面上复制机械时代的产品，而不进行信息时代的加强。

现实世界的机械系统，比如钢笔和纸，有着它们各自的优点和缺点。软件则有着完全不同的优缺点。当机械产品在没有改善的情况下复制到信息化时代时，它们就结合了新旧两种事物的弱点。在通讯录的例子中，计算机能够很容易地根据名字搜索。但是，如果完全采用与机械产品相同的方式存储名字，就没法用新的方式搜索。这样我们既局限于信息媒介的能力，又没有从原始的机械方式中获得任何好处。

当设计师靠机械时代的表现方式指导他们时，虽然计算机提供了更强大的信息管理能力，他们却对其视而不见。

对机械时代表现方式的改善：一个例子

虽然新的技术会带来一些全新的概念，但也能在旧概念的基础上进行扩展，允许设计师利用新技术，为用户对界面表现进行修正。

例如，拿日历来说，在非数字化世界里，日历由纸张制作，并且通常以每月一页的格式分割。这是基于纸张、文件夹、公文包和抽屉的大小而做出的合理折中。

带可视化日历的程序非常普遍，而且它们几乎都是一次显示一个月份。即使可以像 Outlook 那样一次不止显示一个月份页，它们也以离散的每月一块的方式来显示日期，为什么呢？

纸质日历每次显示一个月的日期，因为它们受限于纸张的大小，并且一个月是方便的分隔点。计算机屏幕并不受限制，但大多数设计师完全拷贝机械产品（见图 2-3）。在计算机上，日历很容易成为连续的日期、星期或者月份的滚动序列，如图 2-4 所显示的那样。如果星期是连续的，而不按月份打断，则从 8 月 28 号到 9 月 4 号的时间安排会更加容易。

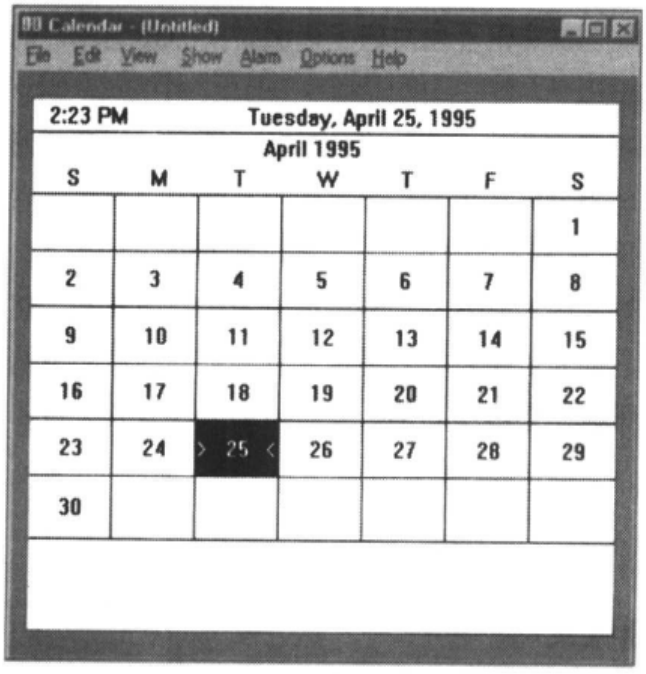


图 2-3 无处不在的日历是那样熟悉，以至于我们设计屏幕上的日历时，很少应用信息时代的敏感特性。最初设计的日历适应堆积的纸张，而不是交互式的数字化显示。你如何重新设计它呢？日历有哪些特性是老式的机械时代平台产物呢？

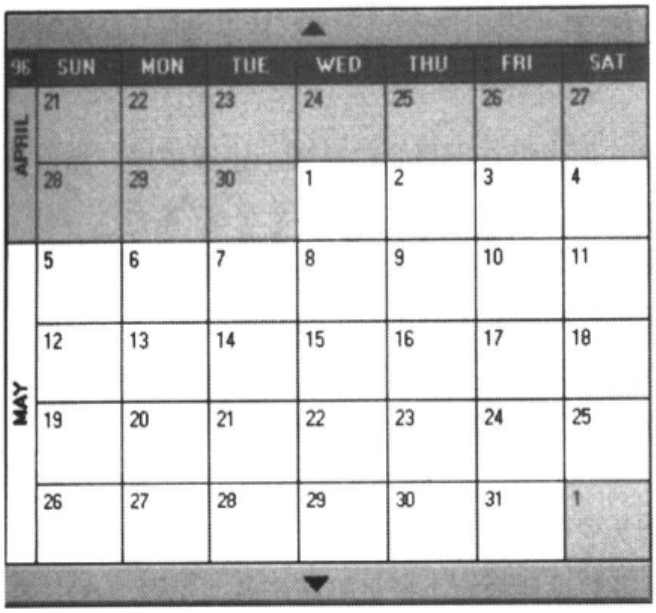
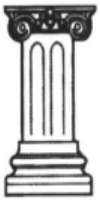


图 2-4 对计算机用户来说，滚动是非常熟悉的常用用法。为了优化，为什么在日历中不用滚动的表达来取代面向页面的表达呢？它也解决了跨越月的边界进行时间安排的机械表达问题。不要根据习惯将旧的限制带入新的平台。你能想到别的改进吗？

相似地，数字化日历的网格始终有着固定的大小。为什么日期栏的宽度或者星期栏的高度不能像数据表那样可调整呢？你肯定想调整周末栏的大小，以反映它相对工作日的重要性。如果你是一名商业人士，你的工作周会比假期要求更多的日历空间。这种像电子表单一样的常见用法众所周知——也就是说非常普遍，但是机械时代的表现被人们那么坚定地重复着——很少有软件商想偏离它。

图 2-3 所示软件的设计者可能会认为日历应作为一个标准的对象，不应该改变其熟悉的方式。令人惊讶的是，大多数时间管理的软件在内部处理为一个连续的统一体（它的实现模型），而仅仅在用户界面上将它们显示为离散的月份（它的表现模型）。

一些人可能会认为每月一页的日历更好，因为它容易识别，并且为用户所熟悉。然而，新的模型和旧的模型区别不大，除了它能允许用户更容易地做过去不容易做的事情——跨越时间边界安排时间。对于适应一种熟悉系统更新的、更有用的表现方式来说，人们并不会觉得有多困难。



公理：重要的改变必须比原来更好。

个人信息管理器（PIM）和日程安排工具中的纸型日历是语言影响设计的无声证词。如果依赖于机械时代的语言，我们会创建机械时代的软件。更好的软件应该基于信息时代的思维方式。

3

新手、专家和中间用户

大部分计算机用户都很清楚：打开新软件产品的塑料包装，意味着在学习新界面的那几天会受到挫折和失望；另一方面，很多有经验的用户也可能发现自己会长时间地感到沮丧，因为程序始终把他当成**新手**（beginner）。看起来很难找到一个合适的平衡点，能够同时满足新手和**专家**（expert）的要求。

在交互和界面设计中，如何用同一个界面满足新手用户和专家用户的需求是长久以来存在的难题之一。一些程序员和设计师完全放弃这个想法，而选择分别提供新手模式和专家模式，新手模式通常是专家模式过分简化、功能减弱的子集。当然，没有人想长期在新手模式下使用软件，但从新手模式跳跃到专家模式的难度就像从陡峭的悬崖跳到大量鲨鱼出没的壕沟里一样，这个壕沟就是设计的实现模型。那么，解决方案是什么呢？解决这个困难的方法在于对用户掌握新概念和任务方式的不同理解。

永久的中间用户

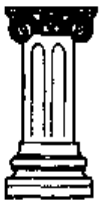
大多数用户既不是新手，也不是专家，相反，他们是**中间用户**（Intermediate）。

进行某种活动的不同经验层次人数分布，就像大多数人口分布一样，遵循着经典的

统计正态分布曲线。对于几乎所有需要知识和技巧的活动来说，如果我们针对不同的熟练程度画出人数曲线，位于曲线左边的新手和位于曲线右边的专家人数都是相对而言较少的，大多数都是位于曲线中间的中间用户。

然而，统计数据并不能说明所有问题。正态分布曲线只不过是瞬间快照，虽然大多数中间用户倾向于保留在这一类型中，而新手不会永远是新手。要维持高水平的技术程度很困难，因此专家们也在快速变化。新手和专家随着时间变化都会倾向于变成中间用户。

虽然每个人都会在一段时间内以新手的形式存在，但没有人会长期停留在这个状态。人们不喜欢显得不称职。而就定义来说，新手意味着不称职。相反，学习和提高是令人高兴的，所以新手努力尽快成为中间用户——或者，他们干脆放弃。例如，所有滑雪的人，都会在新手层次停留一段时间，但那些不能很快取得进步，也就是摔跤过多的人会很快放弃这种运动。剩下的人则会从初学者变为普通的运动者。只有少数人会成为滑双黑钻雪道的高手。



公理：没有人愿意停留在新手级别。

位于曲线左端的新手或者会迁移到中间用户的突出部分，或者会在曲线上消失，去发现一些新的产品或者活动——在其中，他们能够成为中间用户。大多数用户为熟练使用软件而努力。而他们的熟练程度取决于使用软件的频繁程度，像潮水一样有涨有落。Larry Constantine 最早揭示了为中间用户设计的重要性，在他的书“Software for Use”（1999）中，他将那些用户称为**不断提高的中间用户（improving intermediates）**。作者更愿意使用**永久的中间用户（perpetual intermediates）这个术语**，因为新手虽然能很快进步成为中间用户，但他们很少能够继续成为专家。

好的滑雪胜地都有适合学习的平缓坡道，以及能够对真正的滑雪者构成挑战的专家级滑道。但如果滑雪胜地要想在商业上保持成功，它需要迎合永久的中间层滑雪者，并且不吓跑新手，或者无视专家的存在。新手必须发现自己很容易进入中间用户，而那些为不知所措的永久中间层所提供的帮助，则不应该成为专家级滑雪者垂直滑道上的障碍。

可以采用相同的方法平衡用户界面。它既不迎合新手，也不迎合专家，相反，界面的大多数努力用于满足永久的中间用户。与此同时，避免冒犯新手或者专家，因为我们承认他们同样很重要。

处于中间状态的大多数用户都很愿意进一步学习，但通常没有时间。偶尔，也会出现一些机会。有时候，那些中间用户为了完成一个大的项目，持续几个星期大量使用产品。在这段时间内，他们学到一些新的东西，知识也增长了。

然而，有时候，他们又会几个月都不使用该软件，忘掉了大量重要内容。当他们重新使用软件时，他们不是新手，但需要一些提示，回到以前的状态。

如果一个用户发现自己几个小时之后仍然不能取得进步，超越新手阶段，他们会完全放弃，然后换另外一个。没有人愿意时间长了还对某个任务不称职。

为中间用户优化

现在，让我们对比一下中间用户在正态分布曲线上的位置，以及为他们开发软件的方式。对于程序员来说，他们完全有资格成为所编软件的专家，因为他们必须考虑每一种可能的使用情况，无论这些情况多么模糊和不可能，他们都需要创建程序代码来处理。他们自然的倾向是设计实现模型方式的软件，并且在交互中对每一种可能的情况都给予同等重视，因为作为专家他们不存在理解的问题。

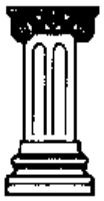
与此同时，销售人员、市场人员和管理人员——他们没有一个可能是专家用户，甚至是中间用户——由于经常要向对软件不熟悉的顾客、记者、合作伙伴及投资商演示产品。因为经常和新手打交道，这些以业人员通常对用户群有一种强烈的偏见。因此，销售和市场人员经常劝说让界面为新手服务就不足为奇。他们要求培训与产品捆绑在一起以帮助苦苦奋斗的新手用户。

程序员只创造适合专家的界面，而市场人员要求只适合新手的交互，但正如我们已经看到的，数目最多、最稳定和最重要的用户群是中间用户。

想到大多数实际用户通常被忽略是令人吃惊的，但情况经常就是这样。你可以在很多企业和基于软件的商业产品中看到这种情况。整体设计偏向于专家用户，而与此同时，一些令人厌烦的工具，如向导（wizard）和 Clippy[1] 捆绑在一起来满足市场部门对新用户的理解。专家级用户很少使用它们，而新手则希望尽快摆脱这些令人不安的有关他们无知的提示。但是，永久的中间用户，则需要永久地面对它们。

---

[1] **译者注** 一种充满视觉隐喻的图形用户界面，作者将它作为例子来引用。



公理：为中间用户优化。

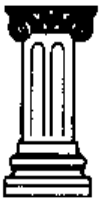
我们的目标既不是吸引新手，也不是将中间用户推向专家层。我们的目标有三个方面：首先让新手快速和无痛苦地成为中间用户；其次避免在那些想成为专家的人面前设置障碍；最后，最为重要的是，让永久的中间用户感到愉快，因为在技术层次上他们稳定地待在中间层。

我们需要多花一些时间，让程序对永久的中间用户来说功能强大而容易使用。同样，我们也必须包容新手和专家，但不能让所占比例最大的一部分用户感到不舒服。本章接下来描述实现这个目标的一些基本策略。

新手需要什么

不可否认，新手是敏感的，而且很容易在开始有挫折感，但我们必须记住，不可将新手状态视为目标。没有人希望永远是新手。它只不过是每个人必须经历的一段过程。好的软件缩短这一过程，并且不将注意力集中在这一过程上。

作为一个交互设计师，最好能想像一下用户——尤其是新手——非常聪明并且非常忙。他们需要一些指示，但不是很多，学习过程应该快速而且富有针对性。如果一个滑雪教练一开始就讲授高山生态学和气象学，他会失去学生，不管他们滑雪的天分如何。用户想要学习如何操作程序，并不意味着他需要或者想要学习里面的工作机制。



公理：将用户想像成非常聪明但非常忙的人。

另外一方面，聪明的人在理解原因和效果后，会学得更好，所以你必须让他们理解为什么软件会像那样工作。我们使用心智模型来弥合这种矛盾。如果界面的表现模型紧密地符合用户的心智模型（正如第2章讨论的），它可以在不强迫用户了解实现模型的情况下，为用户提供所需要的理解。

让新手开始

一个新手必须迅速掌握程序的概念和范围，不然他会彻底放弃。所以设计师第一等的大事就是确保程序充分反映了用户关于任务的心智模型。他可能想不起到底是使用哪个命令来执行特定对象，但是会确切地想起对象和动作之间的关系——一些重要的概念——如果界面的概念结构与他的心智模型一致。

让新手转变为中间用户需要程序提供特别的帮助，而一旦他成为中间用户后，这种帮助反过来会妨碍用户。这意味着无论你提供什么样的帮助，它都不应该在界面里固定下来。当不再需要这种服务时，这种帮助应该消失。

在向新手提供这样的帮助时，标准的在线帮助是一个很糟糕的工具。我们在第 35 章会谈到更多的帮助问题，帮助的主要功能是为用户提供参考。新手不需要参考信息，他们需要概括性的信息，比如说一份操作指南。

单独的指南工具——显示在对话框里——是交流大体情况、范围和目标的好工具。当用户开始使用这些工具时，对话框显示程序的基本目标和工具，告诉用户基本的功能。只要这种引导持续集中在新手所关注的问题，诸如范围和目标，而避免那些只有中间用户和专家才关心的问题（我们将在下面讨论），对于帮助新手来说，它应该就足够了。

新手也依赖于一些菜单学习和执行命令（详细的有关讨论见第 27 章）。菜单可能执行起来很慢而且沉闷，但它们彻底而详细，让人放心。菜单项发起的对话框（如果它们那么做）应该是解释性的（简捷和精炼），并且有方便的“取消”按钮。

专家需要什么

专家也是非常重要的人群，因为他们对缺少经验的用户来说有着异乎寻常的影响。当一个新用户考虑产品时，他会更加信赖专家而不是中间用户的看法。如果专家说：“这个产品不好”，可能意指“这个产品对于专家来说不好”。但是，新手不知道这些，他会考虑专家的建议，即使这些建议并不适用。

专家可能会不时寻找深奥的功能，并且会经常使用其中一些。可以肯定的是，对经常使用的工具集，他们要求能快速访问，这个工具集可能非常大。换句话说，专家需要所有的快捷方式。

任何一天花几个小时使用某个数字产品的人都会快速记住界面的细微差别。他们也不想将所有经常使用的命令记在脑子里，尽管这经常不可避免。频繁的使用要求记忆，并且记忆也是合理的。

专家用户将持续而积极地学习更多的内容，并且将了解到更多他们自身行动和程序的行为，以及表达之间的关系。专家欣赏更新的、更强大功能。对程序的精通使他们不会受到复杂性增加的干扰。

永久的中间用户需要什么

永久的中间用户需要能够访问工具。因为他们已经掌握了这些程序的意图和范围，不再需要解释，对于中间用户来说，工具提示（Tooltip）（参见第 29 章）是最好的中间用户习语。工具提示没有谈到范围、意图和内容，它只是用最简单的常用用户语言来告诉你程序的功能，而且它使用的视觉空间也最少。

永久的中间用户知道如何使用参考资料，只要不是必须一次解决所有问题，他们就有深入学习和研究的动机。这意味着在线帮助是永久中间用户的极佳工具。他们通过索引使用帮助，因此索引部分必须非常全面。

永久的中间用户会确定他们经常使用和很少使用的功能。用户可能会遇到一些模糊的特性，但他会很快地识别出自己经常使用的功能（可能是下意识地）。中间用户通常要求这些常用功能里面的工具放在用户界面的前端和中心位置，容易寻找和记忆。

永久的中间用户通常知道高级功能的存在，即使他们可能不需要也不知道如何使用那些功能。但是软件具有这些高级特性的事实让永久的中间用户放心，让他们确信，投资购买这个程序是正确的选择。如果普通的滑雪者知道在那片树林之后有一个真正高难度的黑钻石专家滑道，他们会感到很放心，即使从来不曾想过使用那个跑道，这也会让她感到充满了梦想和向往。如图 3-1 所示。

你的程序代码必须同时解决业余爱好者和专家可能会遇到的各种情况。但不要让这样的技术需求影响你的设计理念。是的，你必须为专家用户提供那些功能，你也必须为新手提供支持。但更重要的是，必须将你大部分的才智、时间和资源为大部分代表用户——永久的中间用户而设计，为他们提供最好的交互。

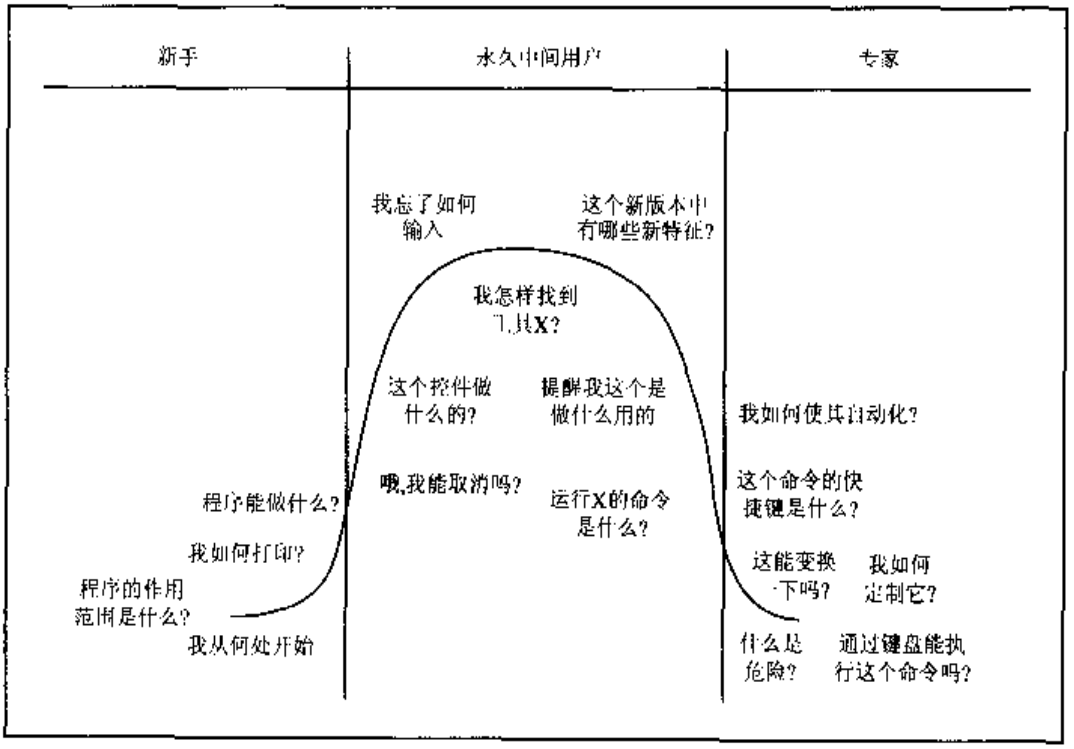


图 3-1 用户对软件的需求随经验而明显不同，因此呈现给用户的工具应该反映这种不同。如果程序的最佳品质在于让第一次使用的用户觉得非常容易，那么它并不会被大家所欣赏，因为大多数用户很快就会成为永久的中间用户。相似地，如果是职业人员和全职专家才会使用的产品，界面则需要迎合他们独特的需要。

4

理解用户：定性研究

任何设计努力的结果，都必须由它最终在多大程度上满足了用户，或委托开发组织的需求来判断。无论设计师多么熟练和富有创造力，如果没有清晰而详细的关于目标用户的知识，问题的约束条件，设计希望获取什么样的业务目标和组织目标，则她成功的机会将会很小。

像“是什么和怎么样（what and how）”这样的问题最好进行定性研究，而不是进行度量或人口统计（虽然它们也有它们的用途）。正如我们将在本章中看到的，定性的研究方法有很多种，每一种在完成整个产品的设计全景方面都起着非常重要的作用。

定性研究与定量研究

大多数人把研究这个词与科学和客观联系起来。这种联系并没错，但是它让很多人产生偏见：只有产生所谓的客观事实（定量数据）的研究才是有效的研究。“数字不会撒谎”这一观念在商业和工程领域非常流行，虽然我们非常理性地知道数字——特别是那些属于人类活动的数字——至少能够像单词一样进行戏剧性的处理或者重新解释。

自然科学——如物理学——所收集的数据和从人类活动中收集的数据不同：电子们

不会每时每刻改变心情，物理学家对实验的严格控制可以将所观察的行为从外部世界孤立开，而这在社会科学中是不可能的。任何企图将人类行为简化为统计的努力都会忽略一些重要的细微差别，尽管这些差别可能不会直接影响商业计划，但确实会显著地影响产品设计。定量分析只能回答那些诸如沿某个轴减少多少的问题，而定性分析能够以丰富的、多元的形式详细回答“什么是”，“怎么样”，及“为什么”等问题。

长久以来，社会科学家已经意识到人类的行为太复杂，受制于太多的变量，无法仅仅靠定量数据来理解。可用性从业人员们从人种学和其他学科借用了一些技术，发展了很多定性的方法，来收集有关用户行为的有用数据，用于一个更注重实效的目标：帮助建立能够更好地为用户服务的产品。

本章余下的章节会关注定性研究技术，这些技术将用来支持后续章节描述的各种方法。在本章的最后，我们会简单地讨论定性研究如何能，以及不能，支持这样的目标。

定性研究的价值

与定量研究相比，定性研究以另一种更有用的方式帮助我们理解产品的问题域、上下文和约束条件。它也能更快和更容易地帮助我们识别产品用户和潜在用户的行为模式。特别是，定性研究能帮助我们理解：

- 已有的产品及其使用方式。
- 新产品或已有产品的潜在用户，以及当前这些用户如何面对新产品设计希望处理的活动和问题。
- 将要设计产品中所含的技术、业务和上下文——**问题域**（domain）。
- 问题域中的词汇和其他社会方面。

定性研究也通过以下方式帮助解决设计项目的进展问题：

- 为设计团队提供可信性和权威性方面的依据，因为设计决定可以追溯到研究结果。
- 统一团队对问题域和用户所关心问题的理解。
- 使管理者能够在产品设计问题上做出更综合的决定，否则这种决定只能基于猜测和个人偏好。

我们在实践中的经验显示：除了上面提到的好处以外与定量方法相比，定性研究方法更快，成本更低，更灵活，以及更有可能为这些能带来更优设计的重要问题提供有用的答案：

- 用户采用当前的方法完成产品预计完成的功能时，会遇到一些什么样的问题？

- 在人们生活中，产品适合哪种更普遍的使用场景？可以以何种方式？
- 用户使用该产品时的基本目标是什么，哪些基本任务能帮助人们达成这些目标？

定性研究的类型

在社会科学和可用性教程中有很多定性研究的方法和技术，我们鼓励读者去阅读这些文献。在本章，作者主要关注那些在近十几年的实践中已经证明很有效的技术，偶尔也关注在可用性工程领域广泛实践的一些相似技术。我们注重从实际应用的角度，而不是从理论的角度讲述这些技术。我们讨论的定性研究技术包括：

- 涉众访谈。
- 主题专家（Subject Matter Expert，SME）访谈。
- 用户和顾客访谈。
- 用户观察/人种学现场研究。
- 文献调研。
- 产品/原型和竞争审查。

【涉众访谈】

尽管新产品设计的研究必须以理解用户为最终目的，但我们应该首先理解建立产品的业务和技术上的上下文。这不仅对确保得到切实可行的最终结果非常必要，而且对为设计团队、管理层和工程团队提供共同语言和理解也非常必要。

涉众（stakeholders）是设计委托组织的所有关键成员，通常包括管理人员以及工程、销售、产品营销、市场交流、顾客支持和可用性各方面的主要成员。它可能也包括如委托组织的业务伙伴等其他组织的类似人员和主管人员。涉众访谈应该在其他用户研究之前开展。

通常，与每个涉众一对一的交流是最有效的，不要在大的跨部门团队中进行交流。一对一的安排能促使一些涉众畅所欲言，单独的访谈也能确保独立的看法不会在人群中淹没。访谈不要超过一个小时，如果确认某个涉众可以提供特别有价值的信息，还可以召开后续会议来继续沟通。

需要从涉众那里收集的重要信息类型包括：

- **在每个涉众看来，产品最初的愿景是什么？**和盲人摸象的故事一样，你可能会发现，对于将要设计的产品，每个业务部门的考虑都有略微不同和不完整的角度。因此，设计工作的一部分就是对各个用户和顾客的看法进行协调。

- **预算和进度是什么样的？**对这个问题的回答提供了对设计工作范围的现实检查，如果用户研究显示需要更大（或更小）的调研范围，它也为管理提供了决策点。
- **有什么技术约束条件？**另一个重要的设计范围决定因素是在给定财政预算、时间和技术约束条件下，对技术可行性的坚定理解。
- **业务上的驱动力是什么？**让设计团队理解商业目标非常重要。这又产生了一个决策点，用户研究应该指出业务需求和用户需求之间的冲突。设计必须尽可能建立一种对用户、顾客和产品提供者来说共赢的局面。
- **涉众对用户的看法如何？**与用户有关系的涉众（诸如顾客支持代表）可能有一些重要的用户观点，帮助你形成用户研究计划。你也可能会发现一些涉众对用户的理解与你在研究中发现的有很大的差别。这些信息会成为这个过程后续管理的重要决策点。

理解这些问题及其对设计方案的影响，有助于设计师更好地为产品的用户和顾客服务。达成内部的一致会帮助你把业务上可能没有意识到的问题连贯起来；达成内部一致对于设计过程后续的决策非常关键，并且能够为你的团队建立信誉。

【主题专家（SME）访谈】

一些涉众可能也是**主题专家**：即你所设计的产品所在使用领域的专家。更多的主题专家曾经是产品或是上一代产品的用户，现在则可能是培训、管理或咨询人员。他们通常是涉众聘请的专家，而他们本身却不是涉众。和涉众一样，SME 对产品和用户能提供一些有价值的看法，但设计师应该小心，主题专家们代表了某种有所歪曲的看法。借助主题专家时应该考虑以下几点：

- SME 是专家用户。产品或领域的长期经验意味着他们可能已经习惯了当前的交互。他们可能倾向于专家级的控制，而不是为永久的中间用户设计的交互。SME 通常不是产品的当前用户，他们更倾向于从管理角度出发看问题。
- SME 知识渊博，但他们不是设计师。他们可能对改进产品有很多想法。其中有些会行之有效而且富有价值，但那些建议中最重要的信息是，什么原因让 SME 产生了改进的想法。
- 对于医疗、科技或者财政等复杂或高度专业化领域来说，SME 非常必要。如果为技术或其他专业领域设计，你可能需要从 SME 那里获取指导，除非你本身就是 SME。借助 SME 能够获取有关复杂规则和最好工业实践的信息。在复杂领域中，SME 关于用户角色和特点的知识对规划用户研究非常关键。
- 你想在整个设计过程中接触主题专家。如果你的产品领域需要借助 SME，你应

该能够在不同设计阶段借助他们完成设计细节的现实检查。确保你在早期访谈中得到这样的机会。

【用户、顾客访谈】

用户和顾客很容易混淆。对于消费类产品来说，顾客通常就是用户，但在公司或技术领域，用户和顾客通常指不同的人。显然，应该对这两组人都进行访谈，因为每组都有自己的产品观点，需要用不同的方式包括进来，形成最终设计。

产品的**顾客（customer）**通常指那些做出购买决定的人。对于消费类产品，顾客通常就是产品用户。但是对面向儿童和十几岁青少年的产品来说，顾客是他们的父母或监护人。对于大多数企业或者技术产品来说，顾客和用户经常不是同个人——经常是一名IT 经理——有着截然不同的目标和需要。为了让产品具有生存能力，理解顾客及其目标非常重要。同样，要意识到这些顾客实际上很少使用产品，当他们使用的时候，也和用户的使用方式不同。

与顾客访谈时，你想了解：

- 他们购买产品的目的。
- 他们在当前解决方案下碰到的问题。
- 他们在购买你所设计的这类产品时的决策过程。
- 他们在安装、维护和管理产品中的角色。
- 与领域相关的问题和词汇。

像 SME 一样，顾客可能也有很多如何改进产品设计的看法。同样，分析这些建议来确定背后隐藏的问题非常重要，因为更好和更完整的解决方案可能会在设计过程的后期浮现出来。

产品的**用户（user）**应该是设计的主要关注点。他们是那些亲自使用产品来达到某些目标的人（而不是他们的经理或者产品支持团队）。潜在用户是那些当前没有使用，但将来会使用这些产品的候选人。用户访谈主要包括当前用户（如果产品已经存在，并且正在修订），以及潜在的用户（包括竞争产品用户和非自动化系统的用户，如果合适的话）。我们有兴趣从用户那里了解到的信息包括：

- 使用产品时的问题和挫折（对于潜在用户，指类似的系统）。
- 产品如何适应用户生活和工作的上下文——什么时候、为什么，以及如何使用产品，这就是产品的**用户行为模式**（patterns of user behavior）。
- 用户角度的领域知识——用户完成工作需要知道的信息。
- 对用户当前任务的基本理解——包括产品需要的和不支持的。
- 对用户目标的清楚理解——他们使用产品的动机和期望。

【用户观察】

大多数用户不能准确地评估他们自己的行为（Pinker, 1999），尤其在超出了他们活动环境的情况下。因此，如果在设计师希望记录的场景之外进行访谈，将会产生不完整和不精确的数据。基本上，你可以与用户讨论他们对自身行为的看法，或者可以直接观察。后面的这种方式能提供更好的结果。

很多可用性专家利用技术辅助手段，诸如视频和音频记录器来获取用户谈到的和所做的一切。一定要小心不要让这些技术显得太突出，否则用户会分心，并且表现和平时不同。

或许，收集定性用户数据最有效的技术是把访谈和观察结合起来，允许设计师提问，并且将问题引导到他们实时观察的用户行为和场所上。

【文献调研】

在涉众访谈的同时，设计团队也应该研究与产品或产品所在领域相关的任何文献。这可以（也应该）包括产品市场规划、市场研究、技术规范和白皮书、本领域业务和技术期刊文献、竞争性研究、相关/竞争产品新闻的 Web 搜索、可用性研究结果和度量数据，以及顾客支持数据，如呼叫中心统计数据。

设计团队应该收集这些文献，以它们为基础来设计对涉众和主题专家（SME）提出的问题。之后也可以把它们作为对其他领域知识和词汇表的补充，并且用它们来检查所收集到用户的数据。

【产品和竞争性审核】

除了涉众和 SME 访谈，研究已有的产品版本或者原型，以及主要的竞争产品，对于设计团队来说也很有帮助。这么做让设计团队能够判断技术发展水平，并且在访谈期间为提问提供动力。理想的情况下，设计团队应该参加当前产品和竞争产品界面的非正式**启发式评估**或**专家评估**（**heuristic or expert review**），并且基于交互和视觉设计原则进行比较（这些内容可以在本书的后面章节中找到）。这个过程让设计团队在熟悉当前用户能够获得的能力和限制的同时，也能得到产品当前功能范围的总体概念。

人种学调查：用户访谈和用户观察

基于多年的设计研究实践，作者相信，在设计师的所有方法中，结合一对一的访谈和对工作/生活方式的观察，是收集用户和他们目标的定性数据时最有效的工具。**人种学调查**（ethnographic interviews）（Wood, 1996）是一种揉和了浸入式观察和引导式访谈的

技术。

Hugh Beyer 和 Karen Holtzblatt 已经率先使用了他们称为**上下文调查**（contextual inquiry）的人种学调查技术。这种方法，已经在工业界获得了好的发展，并且为定性的用户研究提供了合理的基础。对这一技术的详细描述可参见他们的书《上下文设计》（Contextual design, 1998）的前四章。上下文调查方法和下面将要描述的方法大致类似，但也有一些细微之处显著不同。

上下文调查

根据 Beyer 和 Holtzblatt 的观点，**上下文调查**基于师傅-学徒式的学习模型：观察和提出与用户相关的问题，仿佛用户是师傅工匠，而访谈者是新的学徒。Bayer 和 Holtzblatt 也列举了从事人种学调查的四个基本原理。

- **上下文环境**：不是在一个整洁的会客室里对用户进行访谈，而重要的是在正常的工作环境里，或者是其他对产品合适的物理场景中观察用户并且与他们交流。在用户工作时观察用户，并且在堆满了他们每天使用的产品的环境里向用户提问，能够发现他们行为的所有重要细节。
- **协作**：访谈和观察用户应该采取协作的方式探索用户，对工作的观察和对工作结构和细节的讨论可以交替进行。
- **解释**：设计师的大部分工作就是研究以用户行为、环境、言论等搜集到的事实，设计师应该将这些事实作为整体考虑和分析，来发现一些设计含义。当然，调查者必须小心避免基于自己对事实的解释做出假设，而不经过用户的验证。
- **焦点**：设计师不是带着固定的调查表或者让访谈漫无边际地进行，而是需要巧妙地引导访谈，来获取与设计问题相关的数据。

上下文调查的改进

上下文调查形成了定性研究的理论基础，但作为一个具体的方法，至少在作者看来，它存在一些限制而且效率不高。就我们的体会，下面的过程改进措施能得到一个更合理的研究阶段，从而为一份成功的设计创造更好的条件。

- **缩短访谈过程**：上下文调查假定用户访谈需要一整天的时间。作者发现把访谈缩短到一个小时来收集必要的用户数据已经足够了，只要已经安排了充足的访谈数量（对于每一个假定的人物角色或者类型，大约需要六个精心选定的用户）。

与找一些同意花一整天时间的用户相比，找到愿意花一小时的一组用户会更容易，也更有效。

- **缩减设计团队规模**：场景调查假定有一个大的设计团队，同时进行多场访谈，接着是团队所有成员都听取报告会。作者发现由同一组设计师依次进行每一场访谈更有效。这样，设计团队可以很小（两个或者三个设计师），而更为重要的是，这意味着整个团队与所有访谈的用户直接交互，允许成员更有效地分析和综合用户数据。
- **首先识别目标**：Beyer 和 Holtzblatt 所描述的上下文调查支持的设计过程完全以任务为中心。我们建议在确定与目标相关的任务之前，采用人种学调查首先识别用户目标，并对用户目标进行优先级排序。
- **超越商业上下文**：上下文调查主要假定了一个商业产品和一种企业上下文环境。但在消费者领域，进行人种学调查也是可能的，虽然提问的重点会稍微有些不同，我们在本章后面部分会提到这一点。

下面是一些准备和引导人种学调查时的一般方法和技巧。

为人种学调查做准备

人种学是从人类学中借用的一个术语，意味着对人类文化进行系统的浸入式研究。在人类学中，人种学研究人员在他们所研究和记录的文化中花数年的时间进行浸入式的生活。人种学调查借用了这种研究类型的精髓，并将它应用在一个微观的层次，目标在于理解人们与个体产品交互时的行为和习惯，而不是去理解整个文化的行为和社会礼仪。

【识别候选人】

因为设计者必须捕捉与某个产品相关的所有用户行为，在规划一系列的访谈时，确定合适的不同用户样本和用户类型非常重要。借助从涉众、主题专家访谈和文献调研收集的信息，设计师需要建立一个假设作为起点，以确定对什么样的用户和潜在用户进行访谈。

Kim Goodwin（2002a）已经创建了**人物角色假设**（persona hypothesis）这个术语，它是标志和综合人物角色，也就是下一章我们会详细讨论的**用户原型**（user archetype）的第一步。人物角色假设基于可能的行为差异，而不是**人口统计学**（demographics），同时也考虑到了已经得到的目标市场和人口统计学因素。产品领域的性质决定了人物角色假设创建的方式。商业用户在行为模式和动机方面通常和消费者用户大不相同，创建人物角色假设的技术在这两种情况下也不相同。

【人物角色假设】

人物角色假设是为特定领域的产品定义不同用户种类（有时候是顾客）的第一步。这一假设是确定一系列初始访谈的基础，随着访谈的进展，如果数据显示存在一开始没有识别的用户类型，可能需要安排新的访谈。

人物角色假设试图在较高的层次上解决以下问题。

- 哪些不同类别的人可能会使用这些产品？
- 他们的需求和行为可能会怎样变化？
- 需要研究哪些行为范围和环境类型？

【业务和消费领域的角色】

需求和行为的模式，乃至用户类型在不同的业务、技术和消费类产品之间差别很明显。对于商业产品来说，角色——不同用户种类所需要的信息和通用任务集合——提供了重要的初始组织原则。例如，在企业门户中，我们可能会发现这些大致的角色：

- 在门户网站上搜索内容的人。
- 在门户网站上上载和更新内容的人。
- 技术上对门户网站进行管理的人。

在业务和技术上下文里，角色通常大致映射到职位描述，所以通过理解系统用户（或者潜在用户）所在的职位类别，获得一些较合理的初始访谈用户类型相对来说比较容易。

不像业务用户，消费者没有具体的职位描述，并且他们使用产品时通常跨越多个场景。他们的角色映射更接近生活方式的选择，因此从这个意义上说，即使对单个产品，也很容易为消费者用户确定多个不同角色。对于消费者来说，角色通常用行为变量来表示更合适。

【行为和人口统计变量】

除了角色，**人物角色假设**（persona hypothesis）[1]还努力识别可能基于用户需要和行为来区分他们的变量。最有用，但要进行研究才有望得到的是行为变量：在整个范围内变化的行为类型。例如，对于电子商务应用，我们可能会找到几个与购物相关的行为范围：

- 购物频率（经常——不经常）。
- 购物的爱好程度（喜欢购物——厌恶购物）。
- 购物动机（买便宜货——只买对的）。

---

[1] **译者注** 此处对角色（role）和人物角色（persona）在概念的内涵上进行了区分，后者包括前者。

不但消费者用户类型的定义通常可以大致通过组合对应的行为变量来完成，行为变量对于找到业务和技术用户类型也非常重要。在同一业务角色定义中的人可能有不同的存在动机和对将来打算的不同渴望。行为变量可以捕捉到这些，虽然通常必须先收集用户数据。

考虑到收据用户数据之前精确预测行为变量十分困难，另外一个建立人物角色假设时的有用方法是使用人口统计学变量。制订访谈计划时，你可以利用市场研究来标志产品目标人群的年龄、位置、性别和收入。被访者应该分布在这些人口统计学范围内。

【领域专长与技术专长】

要提到的一个重要行为类型区别是技术专长（数字技术知识）和领域专长（与产品相关的特殊主题领域知识）之间的不同。不同的用户有不同程度的技术专长；相似，产品的一些用户可能对产品的领域知识不太擅长（例如，通用的分类账户应用中的会计知识）。因此，根据产品的设计目标，领域支持与技术上容易使用一样，也可能是产品设计的必要部分。如果没有界面提供的支持，领域经验较少的用户只会使用领域相关产品的很小一部分功能。如果缺乏经验的用户也是领域相关产品目标市场的一部分，必须考虑如何支持领域经验缺乏的行为。

【环境因素】

最后要考虑的一点，尤其在商业产品中，是用户所在组织之间的文化差异。例如，较小的公司倾向于工作人员之间有更多的直接接触，而较大的公司则有层层的组织机构。这些环境变量分为几类：

- 公司规模（小公司——跨国公司）。
- IT 程度（特定——严格）。
- 安全级别（松——紧）。

像行为变量一样，如果没有领域方面的研究支持，这些环境变量很难识别出来，因为这些模式确实因行业和地理区域的差异而显著变化。

【组织计划】

在创建好人物角色假设后，连同潜在角色、行为、人口统计学和环境变量，必须制订一个访谈计划来与负责提供用户接触机会的人进行沟通。

每一个在人物角色假设中标志的角色、行为变量、人口统计变量和环境变量都应该在 4 到 6 次访谈中进行探究（有时候如果领域特别复杂，将需要更多访谈）。然而，这些访谈可以重叠，与一个在二十多岁喜欢购物的女性访谈完全可以接受，这可以看做 3 种不同变量的访谈：性别、年龄组和渴望购物程度。通过聪明地将变量映射到具有不同概

况的访谈对象，完全可以将访谈数目控制在可管理的程度之内。

进行人种学调查

在构造了人物角色假设，并且从中推导出访谈计划之后，你将开始访谈了——假定你能接触到访谈对象！在形成访谈计划期间，设计师应该与可以接触到用户的项目涉众一起工作。项目涉众参与通常是让访谈发生的最好方式，特别是对商业和技术产品来说。

如果项目涉众不能帮助你接触到用户，可以与专门从事市场和可用性研究的公司联系。通常在寻找各种人口分布的消费者方面，这些公司非常有用。但这种方法，困难在于有时候不容易找到允许你在他们家里或者工作地点进行访谈的人。

作为消费类产品的最后一种可选情况，设计师可以招募自己的朋友和亲戚。这样的话，很容易在一个自然的环境里观察这些访谈对象，但考虑到人口统计和行为变量的多样性，则又有很大限制。

【访谈团队和时刻】

作者喜欢每次访谈由两个设计师组成的团队：一个引导访谈并适当记笔记；而另外一个人则详细地记笔记（在访谈期间角色可以切换）。与用户进行访谈，一个小时就足够了，除了在家里与消费者见面，或者在用户与产品交互时和他们一起探索这些特殊情况。团队应该将访谈限制在每天 6 次，这样设计师有充足的时间在访谈之间做总结和调整策略，并且不会太累。

【人种学调查阶段】

一个项目的完整人种学调查可以分为三个按时间顺序排列的阶段。不同阶段的调查所采用的方法都会和上一阶段有所不同，这反映出，来自每次访谈的用户行为知识在不断增长。在开始时关注点比较广泛，针对的是总体结构和与目标相关的问题。在这个周期的最后阶段，范围就比较狭窄，关注点变成了特定的功能及与任务相关的问题。

- **早期阶段的访谈**具有探索性质，重点在于从用户角度收集领域知识。广泛而开放地讨论一般问题，而较少深究细节。
- **在中间阶段的访谈**中，设计者开始寻找使用模式，并且提出一些开放式的和澄清性的问题来连接各个要点。一般来说问题更关注领域细节，现在设计师已经掌握了基本规则、结构和领域词汇表。
- **后期阶段的访谈**用来确定先前观察到的模式，进一步澄清用户角色和行为，并且对任务和信息需要的假设进行细微调整。封闭型问题大量增加，对数据进行

收尾。

在你对实际访谈对象有了一些概念之后，如果有机会，与项目涉众一起为访谈周期的每一个阶段安排最合适的人会非常有帮助。有时候，在访谈周期的早期阶段和结束阶段，你都有可能希望回过头去对某个知识丰富、表达清楚的访谈对象进行重新访谈。

【基本方法】

人种学调查的基本方法简单直观，并且不需要什么高科技。虽然掌握访谈对象的细微差别需要一段时间，但是如果任何实践的人遵循了下面的建议，他们会收获大量有用的定性数据。

- 在交互发生的地方进行访谈。
- 避免一组固定的问题。
- 首先关注目标，任务其次。
- 避免让用户成为设计师。
- 避免讨论技术。
- 鼓励讲故事。
- 请求演示和讲解。
- 避免诱导性的问题。

在下面的小节里我们将更详细地描述每一种方法。

【在交互发生的地方进行访谈】

遵循上下文调查的首要原则，进行访谈的地方位于用户实际使用产品的地方非常重要，这不仅让访谈者有机会观察到正在使用的产品，也让访谈团队有机会了解交互发生的环境，从而可以深入地洞察产品的约束条件及用户的需求和目标。

近距离地观察环境有可能发现访谈对象没有提到的任务线索。例如，注意一下他们需要的信息类型（桌面上的纸张或者屏幕边缘贴的便条）；不充分的系统（手头参考资料和用户手册）；任务的频率和优先级（收件箱和发件箱）；他们遵循的工作流类型（备忘录、图表、日历）。未经许可不要偷窥，但如果你看到有趣的事情，可以请访谈对象进行讨论。

【避免一组固定的问题】

如果用一组固定的问题来进行人种学调查，你不仅冒着疏远访谈对象的风险，而且也可能会错失丰富而有价值的用户数据。人种学调查的整个前提条件（和上下文调查）是，作为调查者，我们无法充分了解领域去预先假定我们要问的问题；我们必须从访谈对象那里了解到哪些是重要的。这意味着在心中知道某些问题类型非常有用。

这里有一些**面向目标的问题**，可以考虑。

- 机会：当前有什么活动在浪费你的时间？
- 目标：什么事会使你一天过得很愉快？或是很糟糕？
- 优先级：哪些是最重要的？
- 信息：什么帮助你做决定？

另一类有用的问题是**面向系统的问题**。

- 功能：你使用产品做得最多的是什么事？
- 频率：产品的哪个部分你用得最多？
- 偏好：你最喜欢产品的哪些方面，什么是你的最爱？
- 失败：你如何解决遇到的问题？
- 经验：你使用什么样的快捷键？

对于商业产品，**面向工作流的问题**很有帮助。

- 过程：你今天一早过来，做的第一件事是什么？之后呢？
- 发生的事情和循环：这件事你多久做一次？什么事是每月或每周要做，但不是每天都做的？
- 特殊情况：典型的一天是如何度过的，什么会是不寻常的事件？

为了更好地理解用户的动机，你可以使用一些**面向态度的问题**。

- 期望：怎么看待你自己五年后的前景？
- 避免：你不愿意做什么？哪些事你在拖延？
- 动机：关于你的工作（或者生活方式）你最满意的是什么？哪些问题是你常常会首先解决的？

【首先关注目标，任务其次】

不像上下文调查和其他大多数定性研究方法，人种学调查首先要理解用户行为的原因——是什么激发了不同角色个体的行为，以及他们希望最终如何达到目标——而不是他们执行的任务是哪些。理解任务是重要的，并且任务必须仔细记录。但这些任务最终会进行调整，来在最后设计中更好地配合用户的目标。

【避免让用户成为设计师】

引导访谈对象进行问题研究，但要避免寻求解决方案。大多数时候，这些解决方案对于他自己来说可能比较好，但这些方案要不没有经过深思熟虑，要不就是一些人物角色等用户建模工具想尽量避免的特殊方法。也就是说，如果用户不假思索地说出一个有趣的想法，无论如何先记录下来以备参考，谢谢用户提供的信息，然后继续。

【避免讨论技术】

就像你不想将用户当做设计师一样，你也不想将他们视为程序员或者工程师。对于技术产品或者科学产品来说，技术始终是一个问题，必须区分领域相关的技术与产品相关的技术，并且避免后者。

【鼓励讲故事】

相对于请用户给出设计建议，更有效的是，鼓励他们讲一些使用产品（不管是你正在重新设计产品的早先版本，还是一个类似的产品）的体验故事：他们如何使用它；对它怎么看；使用产品时，还与其他什么人交互；在哪些场合使用它；等等。这些故事是理解用户如何与产品交互和联系的最好方式之一。试着同样鼓励讲述在典型情况下和非典型情况下的故事。

【请求演示和讲解】

在了解了用户活动和交互的流程和结构，并且完成了其他的问题之后，请求访谈对象进行演示和讲解，或者展示与设计问题相关的人工制品非常有用。可以是领域相关的人工制品、软件界面、纸张系统、工作环境参观，最好能包括以上所有内容。不仅要认真记录人工制品本身（在这个阶段，数码摄像机或者摄像机非常方便），也要注意访谈对象如何描述它们。同时也要问足够多的问题进行澄清。

【避免诱导性的问题】

在访谈中需要避免使用对答案有诱导性的问题，这点很重要。就像在法庭上一样，律师能够凭借他们的权威性，通过向证人建议答案来使他们产生偏见。这里设计师也能够通过隐含（或者显式）地建议有关行为的解决方法和观点，无意识地使访谈对象产生偏见。引导性问题的例子有：

- 是否 X 特性会对你有帮助？
- 你喜欢 X，是吗？
- 如果能够获得 X 特性，你认为自己会使用它们吗？

【访谈之后】

在每次访谈之后，团队对笔记进行比较，并且讨论任何观察到的有趣倾向或者在最近访谈中出现的一些细节。如果有时间，也应该回头看一看旧的笔记中是否还有在其他访谈中尚未回答的问题，并研究那些已经适当回答了的问题。这些信息应该用来为后续访谈中将采用的方法制订策略。

在访谈过程完成之后，再重新过一遍所有的笔记，标记或突出显示数据中的倾向和

模式，对于下一步——从累积的研究中创建人物角色——来说非常有用。如果有帮助的话，团队可以把笔记装订成一本，对任何录像带进行回顾，打印出图像将它放在笔记文件夹或者公共区域，比如墙上，这样大家同时都能看到。这会对后面的设计阶段非常有用。

其他类型的研究

本章集中关注定性研究，目标是要收集后面将用到的用户数据（在下一章中将描述），以构造健壮的用户和领域模型，在此基础上，形成目标导向设计合成中的关键工具。设计师可能熟悉，也可能会遇到其他类型的研究，其中大多数作为设计工具并不合适，但在开发阶段中的其他时候或许有用。在本节中，我们来讨论其中几种比较突出的研究方法，以及看它们如何切入到整体开发活动中去。

焦点小组

市场部门特别喜欢通过焦点小组（focus groups）[2]收集用户数据。用户代表的选择一般是参照先前已经标志的人群统计数据。在焦点小组中，用户代表们聚在一个屋子里，有人询问一组结构化的问题，和/或提供一组结构化的选择答案。通常这种会议会以视频或者音频形式记录下来，供以后查阅。焦点小组是传统产品市场研究的标准技术。他们对产品形式、视觉外观或者工业设计的初始反应测定非常有用，也能够用于收集用户长时间使用某产品的反应和情况。

虽然焦点小组看起来能够提供必需的用户接触，但这种方法在很多方面不适合作为设计工具。焦点小组擅长得出人们拥有的，或者愿意（不愿意）购买的产品信息，但是它不适合收集人们实际上使用产品做什么以及如何使用产品的数据。正如已经提到的，人们在精确评价自身做什么，以及如何做等方面存在困难。他们的行为需要观察才能充分捕获。最后，焦点小组，因为他们是一个团队活动，倾向于达成一致意见：大多数人的或声音最大的意见最后将成为小组观点。这对设计过程来说很可怕，因为设计师必须了解产品必须表达的所有不同行为模式。焦点小组倾向于抑制观念的多样性，而这些正是设计师所需要理解的。

---

[2] **译者注** 焦点小组，如通过公开讨论从较多的人或样品中选出的一个小组，了解其成员对特殊主题或地区的意见或情绪反映，尤其是用在市场调研或政治分析中。

市场统计和市场划分

销售专业经常以猜测的方式确定用户的购买动机。一个最有力的工具就是市场划分，它将具有不同需求的人们进行分组，以确定什么类型的消费者最容易接受某种特定的产品或者市场消息。

市场人员根据一组统计学和地理变量，诸如年龄、种族、教育、收入或位置来划分消费者，这些原始数据的收集通常通过结合市场调研和焦点小组的方式进行。更复杂的消费者数据包括心理描绘和行为变量，也包括态度、生活方式、价值、意识形态、排斥风险以及决策制订模式。分类系统，诸如 SRI 的 VALS 划分（SRI, 2002）和 Jonathan Robbin 的地理人口统计（geodemographic）[3]PRIZM 聚集（Weiss, 2000），通过预测消费者的购买能力、动机、自我取向和资源，能够使数据更加清晰。这些市场建模技术不仅能够预测产品和服务的市场可接受度，而且也是非常强大的说服管理人员开发某个产品的工具。毕竟，如果你知道 *X* 个人可能愿意以 *Y* 价格购买某项产品或者服务，那么很容易评估潜在的投资回报。

然而，理解某些人为什么想买某些东西和实际定义产品——它是什么，它如何工作，以及它会将如何使用——并不是一回事。市场划分是一个非常有用的市场定义工具，但在产品定义方面它的效率很低。

可以看出，通过市场研究收集的数据和通过定性用户研究收集的数据之间可以互补。正如已经讨论的，人种学调查者应该利用市场研究来帮助他们选择访谈目标。我们在第 5 章会更加详细地讨论市场划分模型和用户模型。

可用性和用户测试

可用性，或者用户测试，集中关注用户与产品交互中可测量的特性。对产品可用性的评估重点在于标准化测试，以产生可以计量的数据。在可用性测试中，结果经常显示出产品的问题，以及产品成功之处。

可用性测试需要在设计的人工制品上进行。这使可用性测试放在设计周期的较后阶段，即在有了一个连贯的概念和充分的细节以产生文档或原型之后。用户测试也应该作为测试周期的一部分在产品本身上进行，测试结果同时提供给设计师和开发者。

---

[3] **译者注** 原词为 geodeomographic，译者认为是 geographic 和的 demograpchic 的结合，所以翻译为地理人口统计。

因为用户测试结果基本上是可测量和定量的，可用性研究如果用来比较具体设计变量，并从中选择最有效的方案方面，就会非常有用。当你需要验证或者优化交互机制，或者具体设计元素的形式和表达方式时，从可用性测试收集的顾客反馈最有用了。

可用性在以下方面的测试特别有效。

- 命名：分节/按钮的标签有意义吗？某些词语是否比其他的要好？
- 组织方式：相对于提供服务的产品，这对提供信息的产品特别有用。信息分类是否合适？信息项是否都在顾客可能寻找的地方？
- 第一次使用和可发现性：新用户是否容易找到常用的项目？指令是否很清晰？指令是否必要？
- 有效性：顾客能否有效地完成特定任务？是否有错误发生？在哪里？发生频率如何？

当进行用户测试时，确定你在进行测试的目标是实际可测量的，并且测试结果将有助于校正设计问题。Jakob Nielsen 的《可用性工程》（Usability Engineering, 1993）是一本经典的可用性著作，能够在这个主题上提供最好的指导。

花些时间来规划用户研究，将合适的技术用于开发周期的合适阶段，将设计出好的产品，同时也能避免时间和资源上的浪费。在实验室中测试产品是否成功可能会提供大量的数据，但它们不一定很有价值。在开发过程的早期使用人种学调查允许你作为设计师真正理解你的用户——他们的需求和动机。基于定性数据研究和研究支持的模型，一旦有了稳定的设计想法，要判断你所做出的设计选择是否有效，可用性测试会是一个很有效的工具。而定性研究能帮助你在设计这一艰难的过程中有一个好的开始。

5

用户建模：人物角色和目标

最强大的工具在概念上非常简单，但在应用时比较复杂，必须经过深思熟虑。作者使用的最强大的交互设计工具在表面上比较简单：精确的用户描述模型：他期望达到什么目的，以及为什么要达到目的。这种深思熟虑在构建和使用这种模型方面非常明显。

这些我们称之为**人物角色**（personas）的用户模型，并不是真实的人群，但他们基于人们真实的行为和动机，并且在整个设计过程中代表着真实的人群。他们是在人种学调查收集到的实际用户的行为数据的基础上形成的**综合原型**（composite archetype）。我们在研究阶段发现人物角色，然后在建模阶段定型。通过理解我们的人物角色，我们理解在特定场景下我们用户的目标——特定的用户场景是将用户数据变换为设计框架的关键工具。

有很多有用的工具可以作为交互设计师的工具，但作者感觉到人物角色在这些工具中是最强大的。本章关注于人物角色和目标，以及创建这些人物角色的过程，其他模型在本章的最后有简略的讨论。

为什么要用模型

模型在设计、开发和科学中广泛使用。为了更好地理解或者形象化地表达复杂的结构和关系，需要以一种方式表达它们，而模型正是强有力的表达工具。没有这些模型，我们无法借助全景视图或者组织原则来理解没有结构化的原始数据。好的设计模型强调突出的结构特性或者关系，而不会强调不显著的细节。

因为我们是为用户设计的，所以非常重要的是，必须理解和可视化用户之间，用户、社会和物理环境之间，以及用户和我们希望设计的产品之间的一些显著特性。

就像物理学家在原始的观察数据及在这些数据中发现的综合模式的基础上创建原子模型一样，设计师也在原始行为、在数据中观察到的行为及直觉的综合模式基础上创建用户模型。只有当我们将这些模式定型后，我们才可以系统地构造能够顺利地匹配用户的行为、心智模型和用户目标的交互模式。人物角色提供了形式化手段。

人物角色

创建一个能满足广大用户群的产品时，逻辑上是使产品的功能尽可能广泛，以容纳最多的用户。然而，这种逻辑是错误的。成功容纳大量用户的最好方式是为具有特定需要的个体类型设计。

当你任意广泛地扩展产品的功能时，你会增加所有用户的认知负担及导航[1]成本。能够愉悦某些用户的功能可能会降低其他用户的满意程度（见图 5-1）。

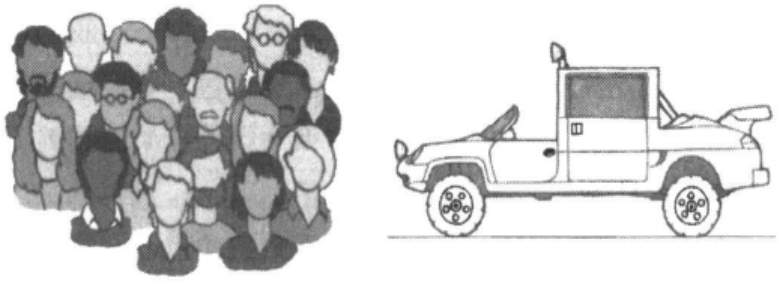


图 5-1　一个非常简单的说明人物角色用途的例子。如果你想设计一辆能够让所有可能的司机开心的汽车，最后的结果是这辆汽车拥有所有的功能，但没有一个人喜欢。今天的软件设计也是如此，企图满足过多的用户，导致用户满意度下降。图 5-2 提供了另外一种可选择的方法。

[1] **译者注**　导航的含义见第 11 章。

关键在于选择为之设计的合适个体，这些个体的需求代表着很大一部分关键成员的需求（见图 5-2），也在于了解如何对设计元素进行优先级排序，以解决最重要用户的需求，而尽量不为其他用户带来不便利。人物角色作为一个强大的工具能够理解用户需求，区分不同的用户类型，并且对用户进行优先级排序，以便在功能和行为设计中确定最重要的目标。

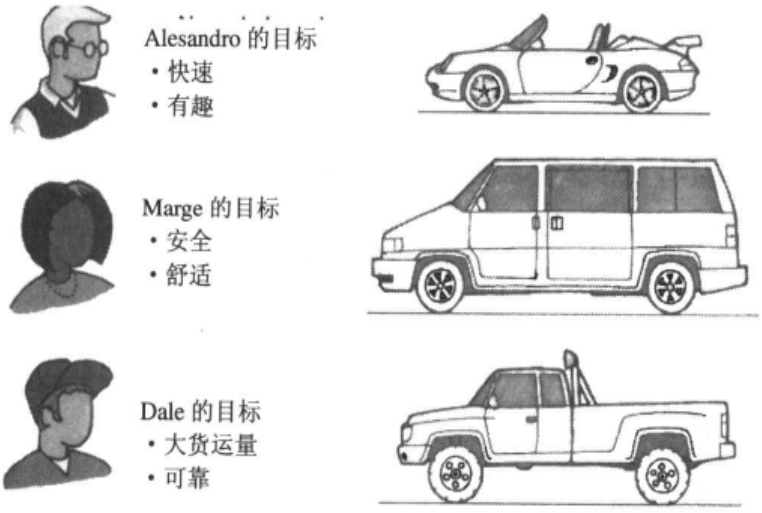


图 5-2 人物角色的实用性的简单示例。通过为有不同具体目标的用户设计不同的车子，我们同时能使与我们的目标用户有相似需求的司机感到满意。这种情况同样对于数字产品和软件设计来说也成立。

自从人物角色在《软件创新之路》[2]中作为用户建模的工具被介绍以来，人物角色在可用性领域获得广泛的关注，但在某些方面，他们经常也被误解。本节尝试深入地澄清和解释人物角色隐含的一些概念和动机。

作为设计工具的人物角色的力度

人物角色是非常强大的、多目标的设计工具，能够帮助克服当前数字产品开发中的很多问题。人物角色在以下这些方面帮助设计师。

- 确定产品应该做什么，以及产品应具有的行为。人物角色目标和任务提供了设计的基础。
- 与涉众、开发者和其他设计师交流。人物角色提供讨论设计决策的通用语言，并且帮助约束设计过程的每一个阶段均以用户为中心。
- 为设计创建一致和约定。通用的语言带来了通用的理解。人物角色减少了对详细的图形模型的需求，因为，正如作者已经发现的，通过人物角色使用的叙述

[2] **译者注** 该书由电子工业出版社出版。

结构，能够更容易理解用户行为的很多细微差别。

- 测度设计的效率。设计选择能够在人物角色上进行测试，就像在正式的过程中显示给实际的用户一样。虽然这不能替代在实际用户上测试的需要，但它为设计师提供了强大的现实检查工具以解决设计问题。这允许设计在**白板**（white board）上反复、快速而低成本地进行，并且在进行实际的用户测试时，产生更强壮的**设计基线**（design baseline）。
- 帮助实现其他与产品相关的努力，诸如市场和销售规划。作者已经看到使用这种方法的客户重新调整他们组织内的人物角色，为销售活动、组织结构及其他策略规划活动提供情况。产品开发以外的商业组织期望产品用户的完善性知识，他们通常对人物角色非常感兴趣。

人物角色也解决了在产品开发过程中出现的三个以用户为中心的设计问题：

- 弹性用户
- 自引用设计
- 边缘情况设计

在下面的小节中，我们将简略地讨论每一种情况。

【弹性用户】

虽然满足用户是我们的目标，当用户这个术语应用于特定的设计问题和场景时，很容易产生问题——产品团队的每一个人都有着对用户和用户需要的理解。当到了做产品设计决定的时候，该“用户”变为了弹性的，变形和调整来适应团队中强势者的观点和假设。

当程序员发现用户能够轻松地从复杂的文件系统（嵌套的、层次的）中获取所需信息时，她们将这一弹性用户定义为通晓计算机文化的超级用户。而当程序员发现用户借助自动工具（wizard）才能够克服最困难的过程时，他们又将弹性用户定义为不成熟的新手。“为弹性用户设计”赋予了开发者根据自己的意愿编码，而仍然能够为“用户”服务的许可。然而，我们的目标是设计合适地能满足实际用户需要的软件。实际用户——以及代表他们的人物角色——并不是弹性的，相反他们有着基于目标、能力和场景的具体需求。

【自参考设计】

自参考设计是指设计师或者程序员将他们自己的目标、动机、技巧及心智模型投射到产品的设计中，大多数很“酷”的产品设计归于此类。用户不会超越像设计师这样的专门人员，这种设计方式对于很狭窄的产品范围合适，但完全不适用于大多数产品。同

样，当程序员创建基于实现模型的产品时，程序员应用的是自参考设计。对于这类产品，程序员很满意，因为他们能完美地理解产品的工作方式。但在非程序员中，很少会有同样的感觉。

【边缘情况设计】

另外一种人物角色能够预防的问题是为边缘情况设计——这种情况经常会发生，但通常不会发生在目标人物角色上。程序员必须考虑边缘情况，但它们又不应该成为设计的关注点。人物角色提供了设计的现实检查。我们能够问："Julie 会经常进行这种操作吗？她曾经会吗？"了解了这些，我们能够清晰地对这些功能进行优先级排序。

人物角色产生于研究

人物角色必须像其他模型一样，基于实际世界的观察。正如在前面的章节讨论的一样，用于综合人物角色的主要数据源来自人种学调查，场景调查，或者其他类似的与实际用户和潜在用户的对话和观察。根据（第 4 章中所描述的）过程搜集的数据的质量直接影响着用于澄清和引导综合设计方案的人物角色的功效性。以功效性粗略排序，能够支持和补充人物角色创建的其他数据包括：

- 在使用场景以外与用户的访谈。
- 涉众和主题专家提供的有关用户的信息。
- 市场研究数据，诸如焦点小组和调研。
- 市场划分模型。
- 从文献调研和前面的研究中收集的数据。

然而，这些补充数据中没有一种能够代替在用户的自然环境中直接与用户交互，以及对他们的观察。在创建得非常好的人物角色的描述中，每一个单词都能够追溯到对用户所说的话的引用，或者观察到的用户行为。

作为个体代表的人物角色

人物角色作为用户模型能够代表具体的个体。他们并不是实际的人，但他们直接从对实际的人观察过程中综合而来。人物角色能够成为成功的用户模型的关键元素之一在于他们是**化身**（personification）（Constantine 和 Lockwood,2002）。他们被描述为具体的个体。这种方式是合适和有效的，因为人物角色作为用户模型有着独特特性：开发团队对

于设计的目标有着同理心[3]。对于设计师来说，同理心非常关键，他们会基于人物角色的认知和情感元素来做出设计框架和细节的详细决定，一个典型的例子是设定人物角色的目标。（我们将在本章的末尾讨论目标、行为和人物角色的重要关联。）然而，同理心的力量不应该因为设计团队中其他成员的介入而被打折扣。作者已经观察到：从设计角度看人物角色不仅使得设计过程能够更容易集成具体的设计元素，而且从涉众采纳的角度看更具有竞争力。当人物角色已经被认真和合适地设计时，涉众和程序员开始考虑问题，仿佛他们是实际的用户。

Grudin 和 Pruitt（2002）已经谈到了电视节目中虚构的人物吸引观众的魔力。他们也注意到，体验派表演方法（method acting）作为演员用于理解和塑造实际的人物的工具的效能。实际上，正如我们将在第 6 章看到的，从用户观察中创建人物角色的过程，以及从这些人物角色的角度进行角色扮演的脚本提纲在很多方面类似于体验派表演方法。有位设计师将作者描述的人物角色目标导向的用法比做交互设计中的 Stanislavsky[4]方法。

人物角色代表着场景中不同的用户类

虽然人物角色代表着具体的个体，与此同时，他们也代表着具体交互产品的一个用户类或类型。具体来说，一个人物角色将关于某个具体产品使用（如果产品还不存在，则是领域内的相似活动）的使用模式和行为模式封装为一个独特的集合。这些模式主要通过分析人种学调查来标志，在必要并且合适时，补充数据也起一定支持作用。这些模式以及与工作或者生活风格相关的角色将人物角色定义为**用户原型**（user archetype）[5]（Mikkelson N. 和 Lee, W.O., 2000）。作者将人物角色称为**综合的用户原型**（composite user archetypes），因为从某种意义上来说，人物角色是在研究阶段，通过在相似的角色中观察不同的个体，聚集相关的用户模式来综合的。

---

[3] **译者注** 同理心见第 6 章，表示认同和理解别人的处境、感情和动机。

[4] **译者注** 斯坦尼斯拉夫斯基，康斯坦丁，1863-1938，俄国演员和导演。他创办了莫斯科艺术剧院，编制了许多契诃夫的戏剧，并且发展了一种强调演员心理动因的有创意的演戏方法。

[5] **译者注** 需要注意 role（角色）、persona（人物角色）和 user archetype（用户原型）之间的含义的细微差别。

【人物角色和重用】

拥有不止一个产品的组织通常想重用相同的人物角色。然而，为了有效，人物角色必须是与场景对应的——他们应该聚焦在特定领域的某个具体产品行为和目标上。因为人物角色是根据对特定场景中与特定产品交互的用户的观察来创建的，他们不能轻易地在不同产品之间被复用（Grudin 和 Pruitt,2002），即使那些产品形成一个紧密相连的套件。即使那样，行为的关注点也可能在一个产品上不同于另外一个产品，所以研究者必须进行补充的用户研究。作者相信，在大多数情况下，人物角色必须为不同的产品单独研究和开发。

【原型与固定形式】

不要将人物角色原型与固定形式混淆。固定形式在大多数方面，是创建得很好的人物角色的对立面。固定形式代表着设计师或者研究者的偏见和假设，而不是实际的数据。基于不完整的研究创建的人物角色（或者在综合时，对被访者的同理心和敏感度不够），则有可能将人物角色降级为固定形式。创建人物角色时，对他们代表的人必须表示尊重和尊敬。如果设计师不尊重他创建的人物角色，其他人也不会尊重他们。人物角色也会带来社会和政治意识的前沿问题（Grudin 和 Pruitt，2002）。因为人物角色提供了精确的设计目标，并且作为设计团队的交流工具，设计师必须认真地选择特定的人群统计特征。理想地，人物角色人群统计应该综合反映研究者在访谈人群中所观察到的，并且根据广泛的市场研究调节。人物角色应该是典型的和可信赖的，但不是一成不变的。如果数据不是结论性的，或者特征对于设计或者可接受性来说不重要，作者宁愿选择不同于性别、人种、年龄和地理分布的另外数据。

发掘了行为深度的人物角色

产品的目标市场描述了人群统计和生活风格，有时候也涉及了职业角色。它不能描述的是目标市场成员展示的有关产品本身和与产品相关的场景下的行为范畴。范畴不同于均值：人物角色不是要创建一般性用户，而是去标志行为范畴内的可被模仿的行为类型。

人物角色满足了理解在给定的产品领域内用户如何行为的需要——他们如何理解产品，用产品做什么，以及在可能会影响产品的范围和定义的其他场景下如何行为。因为人物角色必须描述行为的范围以捕获各种可能的与产品交互的方式，设计师必须标志与任一给定产品相关的人物角色集合。多个人物角色将连续的行为范围变为离散的行为聚

集。不同的人物角色代表不同的相关行为组。通过研究，这些相关性应该非常明显，并且在逻辑上建立关系。行为聚集的过程在本章的后面将更加详细地讨论。

人物角色必须有动机

所有人的行为背后都隐含着动机；一些很明显，而一些则比较微妙。人物角色以目标的方式捕获这些动机非常关键。我们为人物角色枚举的目标（我们将在本章的末尾详细地讨论这些）是动机的简称，动机不仅指出了特定的使用模式，也提供了这些行为为什么存在的理由。理解用户执行任务的原因让设计师能够改善或者甚至是消除某些任务，同时仍然能够完成目标。

人物角色与用户角色

用户角色和用户简要（user profile）都和人物角色有着很多共同点。例如，他们都企图描述用户与产品的关系。但是人物角色和他们被作为设计工具使用的方法明显地在很多关键方面与用户角色及用户简要不同。

用户角色或者角色模型，正如 Larry Constantine（1999）的定义，是抽象的。它们定义的是用户类和他们的问题之间的关系，包括需要、兴趣、期望和行为模式。Holtzblatt 和 Beyer（1998）在统一的流、文化、物理和序列模型中使用人物角色的方法也类似，都企图将各种抽象的关系同拥有这些关系的人隔离开来。

就作者的观点而言，这些方法是有问题的，因为：

- 与拥有这些关系的人们隔离使合适地标志这些抽象关系更加困难——人类同理心的力量无法轻松地转移到抽象的人上。
- 两种方法几乎都专门关注于任务，并且忽略了目标作为思考和综合设计的组织原则的作用。
- Holzblatt 和 Beyer 的统一模型，虽然在特定范围内有用，并且像百科全书，但很难作为一个连贯的工具用于开发、交流和测度设计决定。

人物角色解决了上面的所有问题。创建得很好的人物角色像用户角色一样集成了相同的关系类型，但在叙述[6]中以目标和例子的方式表达它们。这使得设计师和涉众很容易

---

[6] **译者注** 叙述是作者创建的一种交互设计的技术，见第 6 章。

以更人性化的方式理解设计决定的意义。人物角色也使用目标来提供任务的场景和结构，集成了文化和工作流信息，并且将这些信息转换为行为驱动器。

总的来说，人物角色提供了关于用户及其场景更为完整的模型，而许多其他模型倾向于简化。在任意情况下，人物角色能够和其他建模技术一起组合使用。并且正如我们在本章的末尾所讨论的，一些模型极好地补充了人物角色。

人物角色与用户简要

很多可用性从业人员认为人物角色和用户简要这两个术语是一个意思。如果用户简要真实地从人群统计数据生成并且包括了作者已经描述的足够深度的信息，不会有任何问题。不幸的是，作者经常发现用户简要反映了 Webster 对**特性文件**（profile）的定义“简略的传记性质的框架”。换句话说，用户简要通常只是一个名字（或是一个图片）附着在简短的统计数据上，以及一段功能性的短文章描述诸如人们驾驶的车的类型，有多少个孩子，住在哪，怎么生活等信息。这种类型的用户简要更像是一种用户固定模式，而作为设计工具没有什么用处。虽然我们赋予人物角色名字，有时候甚至给车子和家庭成员起名，但它们只是被用来作为叙述工具帮助人们更好地交流实际的数据，而他们本身并不是最终目标。在人物角色创建过程中支持一些虚构细节只是微不足道的一部分，只是为了让人物角色对设计师和产品组成员显得逼真。

人物角色与市场划分

市场人员可能熟悉人物角色创建的过程，因为它和市场定义过程有着某种相似。市场划分和人物角色设计的主要差别在于前者基于统计数据和销售渠道，而后者基于用户行为和目标。两者不是一回事，并且服务的目标也不同。市场人物角色阐述的是销售过程，而设计人物角色阐述的是开发过程。这也就是说，市场划分在人物角色创建过程中起着作用，它们能帮助确定人群统计数据的范围，在这种范围内形成人物角色假设的框架（见第 4 章）。人物角色的划分是在行为范围内，而不是人口统计或者态度，所以很少存在市场划分到人物角色的一对一映射。更确切地说，市场角色划分可以作为最初的过滤器来限制目标市场内需要访谈的人的范围，如图 5-3 所示。

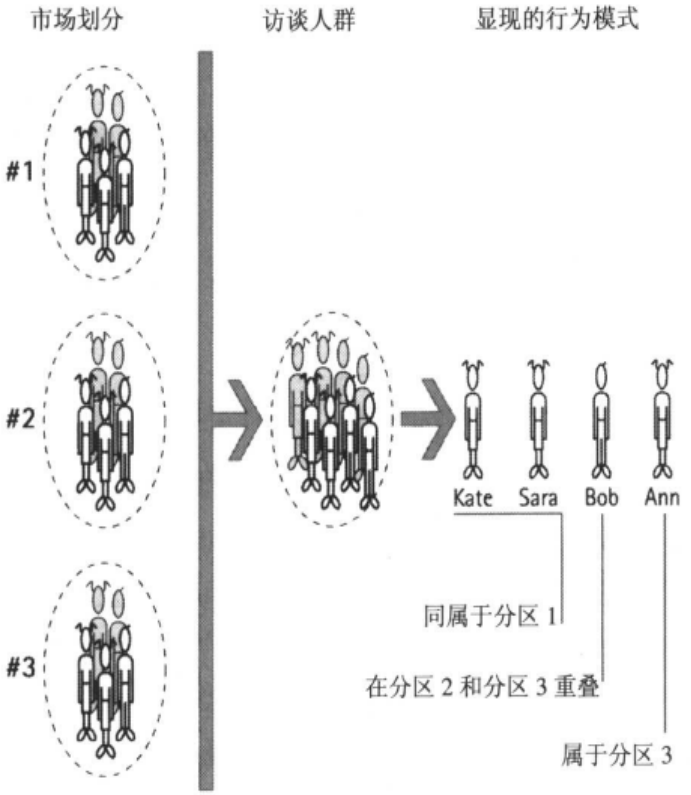


图 5-3 人物角色与市场划分。市场划分能够在研究阶段限制目标市场人物角色的范围。然而，很少存在市场划分和人物角色之间一对一的映射。

用户人物角色与非用户人物角色

一个常见的产品定义错误是将评论、购买和管理产品的人作为目标人群，但这些人并不是最终用户。很多产品是为在消费杂志上评论产品的专栏作家而设计的。购买企业产品的 IT 经理们通常不是产品的用户。在数字产品开发中，为购买者设计是常犯的错误。

虽然 IT 经理的需求不能被忽略，但是如果产品能够很好地为真正的最终用户服务，IT 经理最终也将得到更好的服务。如果最终用户对产品满意并有较高的生产率，IT 经理也同时有成就感。

在特定情况下，诸如需要维护和管理界面的企业系统，创造非用户人物角色是合适的。这要求扩充研究，以包括这些类型的人。这些人物角色可能有他们单独的界面，或者他们只是从修饰学的角度才有用，以描绘产品应该做什么和不应该做什么。非用户人物角色也通常提供了其他的商业目标，这些目标必须与用户人物角色具体表达的用户目标相平衡。

目标

如果人物角色提供了可观察行为集合的场景，目标则是这些行为背后的驱动力。没有目标的人物角色仍然能够作为有用的交流工具，但作为设计工具它们是没有用的。产品的功能和行为必须通过任务来解决目标，通常只有少数任务是必不可少的。必须牢记的是，任务只是达到结果的手段，目标自身才是最终的结果。

目标激发用户模式

人群或者人物角色的目标驱动了他们的行为方式。因此，目标不仅提供了人物角色为什么以及如何期望使用产品的答案，在设计师的心目中也是人物角色复杂的行为和任务的简写。

目标必须来自定性数据

你不能直接问一个人，他的目标是什么；或者他不能描述，或者他不够精确，甚至不完全诚实。人们无法精确地回答这类问题。因此，设计师和研究人员需要认真地从观察到的行为，对其他问题的回答，非动词的暗示，以及诸如书架上的书的标题等环境的暗示中重新构造目标。对人物角色建模的最关键的任务之一是标志它们并且简洁地表达：每个目标应该用一个简单的句子来表达。

目标类型

目标有不同的变体。从以用户为中心的设计角度看，最重要的目标是用户的目标。它们通常在设计中是第一优先级的目标，尤其是在消费类产品的设计中。非用户目标也会起到作用，尤其是在企业环境中。组织、雇主、顾客及合作伙伴的目标如果不能被直接解决，那它需要通过产品的设计来解决。

【用户目标】

用户人物角色有用户目标：这些范围从广泛的期望到高度实用的产品体验。用户目标通常分为三个基本类（Goodwin,2001）：

- 生活目标
- 体验目标
- 最终目标

我们在下面的小节中详细地描述这些目标。

生活目标。生活目标表达了用户的个人期望，这通常超越了产品设计的场景。这些目标是深层次的驱动力和动机，它们能够解释为什么用户极力完成他们期望完成的最终目标。对于理解用户在更广阔的场景中与其他事物的关系，以及从品牌角度理解用户对产品的期望，这些生活目标是有用的。

- 成为做得最好的。
- 快速成长，得到最好的提升。
- 了解一切在这个领域需要知道的知识。
- 做一个道德、模范和被人信任的楷模。

生活目标很少直接映射到具体的界面设计元素中。然而，将它们记在心里是非常值得的。如果用户发现产品能让他们接近生活目标，而不仅是最终的目标，会更果断地赢得他们的心，这超过其他任何市场营销手段。解决用户的生活目标使对产品满意的用户提升为狂热的忠实用户（假设其他目标也得到满足）。

体验目标。体验目标是简单的、统一的和个人的。荒谬的是，这使得人们很难就这一目标进行交流，尤其是在非个人的商业场景下。体验目标表达了一些人在使用产品或者与产品的交互方面的感受。

- 不感觉自身愚蠢。
- 不要犯错误。
- 感到胜任和自信。
- 有乐趣（或者至少不会感到无聊）。

体验目标代表着人们赋予所有软件产品无意识的目标。人们无意识地将这些目标带入场景中，甚至都不需要能够说出这些目标。人们无意识地期望被庄重而有尊严地对待，并且获得支持，而不是得到惩罚。当软件让用户感到愚蠢时，不管其他目标如何，用户的自尊受损，效率降低。他们的不舒适程度和怨恨增加。受够了这种类型的对待，用户会利用一切机会背离这种系统。任何与个人目标不一致的系统最终会失败，而不管这些系统如何声称实现了其他目标。

最终目标。最终目标代表了用户的期望，使用具体产品的切实的成果。当你选择一个蜂窝电话时，你在心目中可能有一个目标。相似地，当你在 Web 上搜索某一项或者某

条信息时，你有着清晰的最终目标。当你用字处理软件打开一个文档时，你在心目中有你期望实现的目标。最终目标必须满足用户，让用户认为产品值得花他们的时间和金钱。产品需要关注的大多数目标是最终目标，诸如：

- 找到最好的价格。
- 确定新闻稿。
- 处理顾客的订单。
- 创建商业的数值模型。

将最终目标和体验目标结合起来。最终目标比体验目标或者生活目标更有吸引力，尤其是对冷静的商人和程序员来说。这是与他们的特征相符的，他们创建的软件虽然绝妙地完成了最终目标，但完全没有满足用户的体验目标。即使最终目标得到识别和满足，如果体验目标最终不能得到满足，用户将对自身和产品感觉很糟。确实，用户完成了自己的目标，但这不是令人愉快或者使人愉悦的体验。另外一方面，如果你的软件忽略了实际问题，而仅仅服务于用户的体验目标，那你设计的只是一个玩具，而不是一个商业应用程序。

【非用户目标】

顾客目标、公司目标及技术目标都是非用户目标。通常，这些目标必须被承认和考虑，但他们不是设计方向的基础。虽然，这些目标需要被解决，不能以用户为代价。

顾客目标。顾客，正如已经讨论的，有着不同于用户的目标。顾客目标的确切性质在消费类产品和企业产品中有很大不同。消费者顾客通常是父母、亲戚或者朋友他们为人物角色购买产品，关心他们的安全和幸福。企业顾客通常是IT经理，他们经常关注的是安全、易于维护，并且容易定制。顾客人物角色也有着他们自身的生活、体验和与产品有关的最终目标。顾客目标始终不应该超越最终目标，但需要在整体设计中被考虑到。

公司目标。商业和其他组织有着对软件的其他需求，并且它们处于与个人目标相同的层次。“增加我们的利润”是董事会和股东的基础要求。设计者使用这些目标以持续关注更重要的问题，并且避免被任务或者其他错误的目标分心。公司目标包括以下内容：

- 增加利润。
- 增加市场占有率。
- 打败竞争者。
- 更有效地使用资源。
- 提供更多的产品或者服务。

研究工作场所的心理学家有一个术语——**卫生因素**（hygienic factor）。Saul Gellerman（1963）将其定义为“有效动机的前提条件，但其自身无驱动力”。例如，办公室的光线就是卫生因素。你不会因为光线亮而去上班，但如果根本就没有光线，你根本就不会去。

公司目标在这种意义上来说就是卫生因素。从公司的角度看，公司有重要的目标。但公司并不做事，他们的雇员才是做事的。雇员更为个人化的生活、体验及最终目标与公司目标同等重要。

然而公司目标也不能被轻视。不能实现这些目标的软件会失败，就像那些无法满足用户目标的软件一样。

技术目标。我们日常生活中使用的大多数基于软件的产品的创建都考虑到了技术目标。很多这些目标减轻了软件创建的任务，这是程序员的目标。这就是为什么它们在用户目标之前被考虑的原因。

- 节约内存。
- 在浏览器中运行。
- 监视数据一致性。
- 增强程序执行效率。
- 使用“酷”的技术或者特性。
- 维护平台范围内的一致性。

技术目标对于开发人员来说尤其重要。在教育阶段的早期强调技术目标需要服务于用户目标和商业目标非常重要。技术目标应该服从于其他更面向人的目标。使用新技术可能是软件公司的任务，但它从来不是用户的目标。就用户而言，他们不关心我们的工作是使用层次数据库、关系数据库、面向对象数据库、平面文件系统，还是魔术来完成。他们考虑的是轻松而又有尊严地很快完成工作。

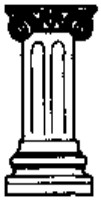
成功的产品必须首先满足用户目标

设计一个好的交互式产品仅仅对于那些在某种场景下带有某种意图使用人工制品的人才有意义。你必须要了解某一具体人的意图。两者是不能分开的。这就是为什么行为设计过程的一个关键工具是人物角色：具有特定目标或者特定意图进行工作的具体人。

需要考虑的最重要的意图或者目标是那些实际使用人工制品或者应用程序的个体，而不必是那些购买者。与产品交互的是实际的人，而不是公司或者 IT 经理，所以你必须将他的个人目标比那些雇用他的公司或者支持他的 IT 经理看得更重要。你的用户会在实现他们个人目标的同时，尽量实现他们雇主的商业目标。用户最重要的目标是始终保留

他的个人尊严：不感觉自身愚蠢。

可以肯定地说，如果我们让用户犯大错误，无法让他们顺利地完成大量工作，或者让他觉得厌烦，我们就会让用户觉得自己很愚蠢。



公理：不要让用户觉得自己很愚蠢。

这可能是最重要的交互设计指导准则。在本书中，我们详细研究当前软件中存在的多种让用户感觉愚蠢的方式，并且我们探索了避免这些陷阱的方法。

好的交互设计的实质是设计交互，实现制造商或者服务提供者的目标以及他们的合作伙伴的目标，而又不违背用户的目标。

构造人物角色

正如前面所讨论的，在与产品的用户和潜在用户的访谈或者观察期间（有时候是一些顾客），人物角色从观察到的模式中推导出。这些数据的缺口通过 SME，涉众及可以获得的文献提供的补充研究和数据来填充。在构造人物角色集合过程中，我们在一系列的可观察的行为变量的范围内（也称为轴或者范围）划分使用。良好创建的人物角色集成了有关目标、态度、工作或者活动流、环境、技巧和技巧层次，以及挫败的信息。

创建值得信任和有用的人物角色需要等量的详细分析和创新综合工作。标准化的过程对这两种活动都有显著的辅助作用。在本节中描述的过程，由 Robert Reimann, Kim Goodwin 和 Lane Halley 开发，是在几十个交互设计项目中持续演化而来的。

1. 再次讨论人物角色假设。
2. 将访谈主体映射为行为变量。
3. 标志重要的行为模式。
4. 综合特征和相关目标。
5. 检查完整性。
6. 展开叙述。
7. 指定人物角色类型。

再次讨论人物角色假设

在你完成了研究并粗略地组织数据之后，下一步你比较在数据中标志的模式与在人物角色假设中做出的猜测。你标志的可能的角色是否真正的不同呢？你标志的行为变量是否有效（见第 4 章）？是否存在其他的你没有预料到的情况，或者你预料的不被你的数据所支持的情况？

列出观察到的行为变量的完整集合。诸如年龄或者技术能力的统计变量看起来也可能会影响行为变量，但警惕不要在人物角色创建阶段关注统计数据，因为行为变量对设计更有影响。例如对企业应用来说，行为和（统计）变量通常与职业角色紧密相关。虽然变量数目因项目而不同，但通常来说对于每个角色发现 15 到 30 个变量很普遍。

如果数据与假设不同，则需要添加、删除或者修改所期望的角色和行为。如果变化非常显著，你可能需要考虑添加访谈来弥补你所发现的新的行为范畴内的一些缺口。

将访谈主体映射为行为变量

在你满意于已经标志出通过访谈主体显示的整个行为变量集合后，下一步是将每一位访谈者映射到合适的变量范畴。这种映射的精确度并不像标志访谈者的相互位置关系那样重要。换句话说，如果访谈者仅在 45%或 50%的程度上精确对应特定刻度，也是可以接受的（通常没有精确度量的方式，你必须依赖于你对所观察的主体的感觉）。这是多个主体聚集在每个重要变量轴上的方式。

识别显著的行为模式

在你已经将该谈主体映射到行为变量后，你可以看到聚集在多个范围或者变量上的特定主体。一些聚集在 6 到 8 个不同的变量上的主体的集合可能会代表一个显著的行为模式，而这个模式会形成人物角色的基础（Goodwin, 2002）。一些特殊的角色可能仅仅会出现在一个重要的模式中，但通常你会发现两个或者甚至三个这样的模式。

模式要有效，在聚集的行为间就必须有逻辑或者因果关系（Goodwin, 2002），而不仅仅是伪造的关联。例如，如果数据显示经常购买 CD 的人也可能下载 MP3 文件，那么存在明显的逻辑关联。但如果数据显示经常购买 CD 的访谈者也喜欢收集邮票，那么可能不存在逻辑关系。

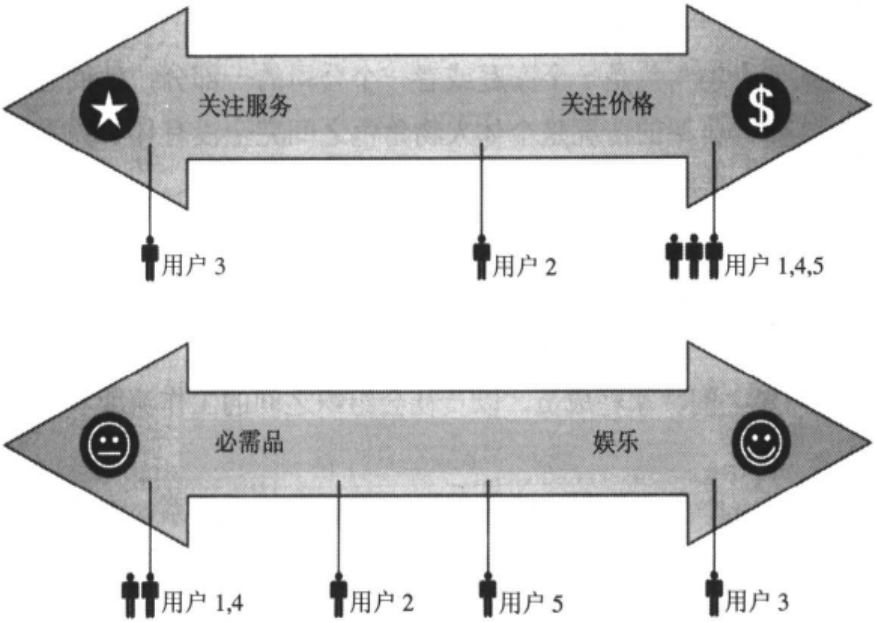


图 5-4 将访谈主体映射为行为变量，这个例子选自电子商务。访谈主体被映射到不同的行为轴上。单一主体绝对位置的精确值没有它相对于其他主体的相对值重要。在多个轴上聚集的主体显示了显著的行为模式。

综合特征和相关目标

对于你识别的每一个显著的行为模式，必须从数据中综合细节。描述潜在的使用环境，典型的工作日（或者其他相关的时间周期），当前的解决方案和挫折，以及与其他模式的相关关系（Goodwin，2002a）。

描述行为特征简略的要点就够了。尽量地按照被观察到的行为来描述。一个或者两个能精确描述人物角色的人格特征就能比较形象地描述人物角色。过多的虚构特质的描述，不仅是一种干扰，而且使得你的人物角色可信度降低（Goodwin，2002a）。记住你是在创造一个设计工具，而不是长篇小说的人物梗概。仅仅是具体的数据才能够在设计及团队做出最终的商业决定时起作用。

在这一阶段，有一个虚构细节是重要的——人物角色的名和姓。人物角色的名字应该是对其类型的体现，而不是倾向于漫画或者固定形式。作者使用婴儿命名书来帮助创建人物角色名字。这时候，你也能够添加一些统计信息，诸如年龄、地理位置、相对收入（如果合适）及职业头衔。这些信息在合成行为细节时，主要帮助你更好地可视化人物角色。从这时开始，你就要用他或者她的名字来描述人物角色。

【人物角色的人际关系】

有时候，某个产品的一系列人物角色是一个家庭或者一个公司的一部分，他们之间有着人际关系和社会关系。然而，典型的情况是个体人物角色之间完全没有什么关系，并且经常来自不同的地理位置和社会族群。

仅仅对于以下情况，才需要理解人物角色之间的社会关系：

1．你没有观察到访谈主体的任何行为变化是与公司大小、产业，或者家庭/社会变化相关的。

2．这么做，对于解释协作者之间、家庭成员，以及社会组织之间的工作流或者社会交互非常关键。

如果你创建的人物角色为同一个公司工作，或者相互之间有着社会关系，在你需要表达的重要目标不属于预先确定的关系时，则可能会遇到麻烦。尽管在人物角色集合中定义单一社会关系，比在个体人物角色及人物角色集合之外的次要人物角色之间定义多种不同的、不相关的社会关系更容易，在最初创建人物角色时，最好还是创建不同的、多样的人物角色，而不是冒险将多个不同的脚本提纲集成到单个社会动态关系中。

【综合目标】

目标是从访谈和行为观察中综合出最关键的细节。目标最好通过分析包含每一个人物角色的行为组来推导出。通过识别每一个人物角色行为的逻辑关联，你开始得到产生这些行为的目标。你能够通过观察行动（在每个人物角色聚集中访谈主体企图完成的事情及其原因）和分析访谈主体对目标导向的问题的回答来推导目标（参见第4章）。

要想作为有效的设计工具，目标在某种程度上必须始终直接与正在设计的产品相关。通常，对于人物角色来说大多数的有用目标是最终目标。大多数人物角色有三到五个与他们相关的最终目标。生活目标对于面向消费者的产品的人物角色来说是最有用的，但是在临时性的职业角色（role）中对于企业人物角色也很有意义。没有生活目标或者一个生活目标对于大多数人物角色来说是合适的。通常的体验目标诸如“不感觉愚蠢”及“不要浪费时间”可以被认为是任一人物角色都隐含的。有时候，有些领域可能要求更详细的体验目标。没有体验目标或者一、两个体验目标对于大多数人物角色来说是合适的。

检查完整性和独特性

从这时开始，你的人物角色应该开始被赋予生命。你应该检查映射集合、人物角色的特征和目标，以确定是否存在重要的缺口需要填充。这又进一步要求进行其他的研究，

以找到在你的行为轴上缺失的行为。你可能也希望检查你的笔记看看是否需要加入行政人物角色以满足涉众的设想或者请求。

如果你发现两个人物角色仅仅在一些统计数据方面存在差异，你可能选择去掉其中一个冗余的人物角色或者调整人物角色特征让他们截然不同。每一个人物角色应该至少在一个显著的行为上与其他人物角色相异。如果你在映射方面做得很好，则应该不是一个问题。

通过确定你的人物角色集合具有差异性和独特性，你能够维护一个可管理的人物角色集合。

展开叙述

你所列出的关键特征要点和目标指出了复杂行为的实质，但留下了更多有待澄清。第三方的叙述在向其他团队成员传达人物角色的态度、需要和问题方面非常有力。它也加深了设计师/作者与人物角色及其动机方面的联系。

典型的人物角色叙述应该不长于一到两页的内容。人物角色叙述不需要包含每一个观察到的细节，因为在理想情况下，设计师也参与了研究工作，而大多数团队以外的人又不需要过多的细节。

因此，在本质上，叙述（narrative）包含一些虚构的事件和反应，但正如前面所讨论的，它并不是一个简短的故事。最好的叙述快速地从职业或者生活方式（企业与消费者）方面介绍人物角色，并且简略地给出他一天的生活梗概，包括焦躁、关注和兴趣等与产品有直接关系的各个方面。细节应该是你特征列表的扩充，以及其他从你所观察和访谈中获得的数据。叙述应该以一个总结的形式表达人物角色对产品的需求。

在你的描述中，必须认真地考虑细节的精确度。细节不应该超出你研究的深度。对于科学学科来说，如果你记录了一个 35.421 米的度量，这意味着你的度量精确到了.001 米。同样，详细的人物角色描述意味着在你研究中相似程度的观察（Goodwin, 2002a）。

当你开始展开叙述时，选择你人物角色的照片。照片让你在展开叙述，或者在你完成后让团队中其他人加入时，显得更加真实。选择照片时应该非常小心。最好的照片不仅捕捉了统计信息、环境线索（护士人物角色应该穿着护士外套，并且在诊所环境下，或许还有一个病人），还捕捉了人物角色的一般态度（承受大量文书工作的秘书照片可能会看起来很吓人）。作者保存了几个可检索的照片库，以备查找合适的人物角色图片。

指定人物角色类型

到目前为止，你应该感觉到你的人物角色看起来非常像你所认识的真实的人。人物角色构造的最后一步完成了将你的定性研究变为强有力的设计工具的过程。

所有的设计都需要一个设计目标——设计所关注的受众。我们已经创建的人物角色，代表着设计目标的可能候选人。单独的一个界面只能为一个专一的人物角色设计。

公理：每一个界面的设计都是为了某个专一的、主要的人物角色。

接下来，我们必须要做的是对我们的人物角色进行优先级排序，以确定首要的设计对象。我们的目标是找到一个人物角色，他的需要和目标能够通过单一界面完整而愉快地被满足，而不会剥夺其他人物角色的权利。我们通过指明人物角色类型的方式完成这一步聚。存在六种类型的人物角色，他们通常大致以下面列出的顺序指定。

- 首要人物角色。
- 次要人物角色。
- 补充人物角色。
- 顾客。
- 所服务的人物角色。
- 负面人物角色。

在下面的小节中，我们从设计师的角度讨论每一个人物角色的类型及其意义。

【首要人物角色】

首要人物角色代表着界面设计的主要目标。单一产品中，每个界面只对应一个首要人物角色。但是，对于某些产品（特别是企业产品）来说，存在多个不同的界面是可能的，其中每个界面都针对不同的首要人物角色（例如，健康信息系统中诊所界面和财务界面可能是分开的，并且每个界面针对不同的人物角色）。

针对集合中任何其他人物角色的设计都不能满足首要人物角色。然而，如果设计针对的目标是首要人物角色，所有其他的人物角色都能至少被部分地满足（因此，他们肯定会高兴）。

选择首要人物角色是一个排除的过程。每个人物角色必须通过将他的目标与其他人

物角色的目标进行比较来测试。如果没有首要人物角色是明显的，则意味着存在两种可能：要么产品需要多个界面，每个界面都针对合适的首要人物角色（这种情况通常都存在于企业和技术产品）；要么产品想实现的目标太多。如果你的消费类产品有多个首要人物角色，产品的范围可能过于宽广。

【次要人物角色】

有时会出现这样一种情况：如果在针对首要人物角色设计的界面中，增加一个或者两个具体的附加需求（这些需求并不为首要人物角色所需要），某个人物角色就能够完全被这一界面所满足。这说明这个人物角色是这一界面的次要人物角色，并且这一界面的设计必须解决这些需求，并不妨碍首要人物角色。通常，一个界面有零到两个次要人物角色。太多的次要人物角色则表明产品的范围可能有问题。

【补充人物角色】

不是首要人物角色或者次要人物角色的用户人物角色都是补充人物角色：他们完全可以被某个主要界面满足。对单一界面，可以存在任意数目的与这一界面相关的补充人物角色。有代表性的一个例子是，行政人物角色——被加进来用以代表涉众的利益——通常成为补充人物角色。

【顾客人物角色】

顾客人物角色解决的是顾客的需要，而不是最终的用户的需要，这一点在本章的前面已经讨论过。通常，顾客人物角色被处理为次要人物角色。然而，在某些企业环境下，一些顾客人物角色能够成为首要人物角色，因为他们有自己的管理界面。

【所服务的人物角色】

所服务的人物角色在某种程度上不同于已经讨论过的人物角色。他们根本就不是产品的用户。然而，他们直接受到产品使用的影响。被放射机治疗的病人不是机器界面的使用者，但她们会因为一个好的界面受到更好的服务。所服务的人物角色提供了一个跟踪产品第二类的社会和物理分支的方式。他们被当成次要人物角色。

【负面人物角色】

像被服务的人物角色一样，负面人物角色不是产品的用户。不像被服务的人物角色，他们的使用纯粹是带修饰色彩的，帮助团队中其他成员进行交流，而确定哪些不应该是产品的设计目标。对于最终用户的企业产品来说，负面人物角色的好候选人通常是 IT 专家，而对于消费者产品来说，好的负面人物角色是了解技术并且很早就使用过产品的人物角色。

其他模型

人物角色是非常有用的工具，但它们确实不是惟一用来帮助人们对用户和它的环境进行建模的工具。Holzblatt 和 Beryer 的场景设计一书提供了将在这里简略讨论的有关模型的丰富信息。工作流或者序列模型对于捕捉组织内的信息流和决策过程非常有用，并且通常被表达为捕捉多种现象的有向图：

- 什么发起了一个过程。
- 用户产生和消费的信息。
- 用户做出的决策。
- 用户采取的行动。
- 从行动中产生的结果。

一个创建得很好的人物角色应该捕捉人物角色工作流，但是工作流模型在捕捉人物角色间和组织内的工作流上仍然非常必要。

人工制品模型（artifact model）正如它的名字所建议的，代表了用户在他们的任务和工作流中采用的不同人工制品。通常这些人工制品是在线或者论文形式的。人工制品模型通常为了在最终的设计中提取和复制最好的实践，捕捉了相似人工制品之间的共同点和显著差异。人工制品在设计过程的后期非常有用，但在直接将纸上系统变换为数字系统需要警惕，不认真分析目标和设计原则的应用范围（特别是本书的第二篇内容），通常会产生可用性问题。

物理模型（physical model）像人工制品一样，企图捕捉用户环境的元素。物理模型关注于捕捉构成用户工作空间的物理对象的布局，它提供了对使用问题的频率和生产率的物理障碍的观察。好的人物角色描述会集成这些信息，但在复杂的物理环境中它会创建具体详细的用户环境的物理模型（地图）而有所帮助（诸如车间和装配线）

人物角色和其他模型让那些大量而困惑的用户数据变得有意义。现在你获得了作为设计工具的完善模型，下一章会显示如何使用这些工具来将用户目标和需要变换为可工作的设计方案。

6

脚本提纲：将目标转换为设计

在前面两章里，我们描述了如何捕捉有关用户的定量信息。通过认真分析这些信息和综合用户模型，我们能够获得有关用户和他们各自目标的清晰图景。我们也解释了如何进行优先级排序来确定哪些用户是最合适的设计目标。这里，我们所缺少的是能够将这些知识转换为连贯的设计方案的过程，而这个设计方案不仅能满足用户的需要，同时也满足业务需要和技术约束条件。

本章将描述一个能弥合研究和设计鸿沟的过程。这个过程使用人物角色作为技术集合中的主要人物，借助这些技术得以快速地以迭代、可重复和可测试的方式获得设计方案。这个过程主要有三个里程碑：定义用户需求；依次使用这些需求来定义产品的基本交互框架；用不断增强的设计细节来完善框架。将这些过程粘合在一起的是叙述：使用人物角色来讲述针对设计的故事。

作为设计工具的叙述

叙述，或者讲故事，是人类最早的活动之一。关于叙述能够表达思想的强大功能已经在很多地方论述过。此外，叙述能够使用和激发创造性视觉技巧，从而可以作为产生

和确认设计思想的一种强大工具（Rheinfrank & Evenson, 1996）。因为交互设计首先是随着时间发生的设计行为，叙述结构和最简单的视觉工具如白板，最适合于构思和表达交互概念。详细的细化需要更复杂的视觉和交互工具，但定义需求和框架的初步工作最好得以灵活而弹性地完成，而较少地去依赖技术，因为技术不可避免地会妨碍构思。

设计中的脚本提纲

对专业的可用性专家来说，**脚本提纲**（scenario）是一个熟悉的术语，通常用于描述通过具体化来构思问题解决方案的方法（Carroll, 2001）：使用具体的故事来构造和讲述设计方案。脚本提纲根据一定具体的模板设计，但也允许灵活性。设计团队中的任何人员都可以按照自己的意愿来修改它。正如 Carroll 在“Making use”中所描述的：

“脚本提纲既具体又粗糙，既切实又灵活……它们含蓄地鼓励在整个团队中以“如果……则会怎么样”的方式思考。它们使在不削弱创新性的情况下理清设计构思成为可能……脚本提纲驱使设计师关注于将促成产品设计的使用。它们能够以不同的细节程度，针对不同的目的来描述情况，从而帮助协调设计方案的各个方面。”

Carrol 有关**基于脚本提纲设计**（scenario-based design）的使用关注于描述用户如何完成任务（Carrol, 2001）。它由一组环境设定值组成，并且包括从用户替身中抽象出的代理或者参与者，它们有着基于角色的名字，如会计或者程序员。

虽然 Carrol 确实理解了在设计过程中脚本提纲的效能和重要性，作者认为在 Carroll 使用脚本提纲的过程中存在以下两个问题。

- Carrol 的脚本提纲在表达人类参与者方面不够具体。不详细理解系统用户的具体细节，就不可能设计系统的合适行为。抽象地面向角色的模型在提供对用户的理解或者同理心方面不够具体。
- Carrol 的脚本提纲过于仓促地转向详细的任务，而没有考虑用户的目标和动机，而正是用户目标驱动和过滤了这些任务。虽然 Carrol 确实较为简单地讨论了目标，目标在这里只是指脚本提纲的目标。这些目标某种程度上随着具体任务的完成被循环地定义。Carroll 的脚本提纲开始于错误地细节层次：应该先考虑用户目标，而不是先标志用户任务，以及对它们进行优先级排序。不解决用户的目标，高层次的产品定义会很困难。

作者认为在基于脚本提纲的设计方法中缺少的重要组成部分是用人物角色的使用。人物角色提供了充分切实的有关用户的表达，能够在脚本提纲设置中作为可信任的代理。

这加强了设计师对用户心智模型和角度的同理心。与此同时，它提供了对用户动机如何影响任务，以及对任务进行优先级排序的探索。因为人物角色是对目标，而不是简单的任务建模，因此脚本提纲解决的问题范围能够扩充到包括产品定义。人物角色帮助回答了以下问题："产品应该是什么样的？"，以及"产品看起来应该如何和应该采取什么样的行为"。作者通过引入**基于人物角色的脚本提纲**——使用了人物角色和目标的脚本提纲，解决了基于任务的脚本提纲产生的问题。

在脚本提纲中使用人物角色

基于人物角色的脚本提纲（persona-based scenarios）是使用产品来实现具体的目标的一个或者多个人物角色的简明叙述性描述。脚本提纲捕获了时间轴上的人工制品和用户之间的非口语的对话（Buxton, 1990），以及交互功能的结构和行为。目标是任务的过滤器，并且在构造脚本提纲的迭代过程中，指导如何组织信息和控件的显示。

脚本提纲的内容和场景是从在研究阶段收集并在建模阶段进一步分析得到的信息中推导出来的。在这些脚本提纲中，设计师扮演为人物角色，成为脚本提纲中的人物（Verplank, et al, 1993），就像演员进行即席创作一样。这种过程导致了对结构和行为的实时合成——通常绘制在白板上，后来形成详细的外观。最后，人物角色和脚本提纲被用来在整个过程中测试设计思想和设计猜想的有效性。在设计过程中的不同阶段，应用了三种不同的基于人物角色的脚本提纲。在每一阶段，都持续而关注在一个狭窄的范围。这些脚本提纲类型——**场景脚本提纲**（context scenarios）、**关键路径脚本提纲**（key path scenarios），以及**确认脚本提纲**（validation scenarios）——将在本章中展开详细描述。

基于人物角色的脚本提纲与用例

脚本提纲和用例（use case）都是用来描述数字系统的方法。然而，它们服务于不同的功能。目标导向的脚本提纲是从具体用户（人物角色）角度定义产品行为的迭代手段。这不仅包括系统的功能，也包括功能的优先级，以及这些功能以用户所看到的和用户与系统交互来表达的方式。

用例，在另外一个方面，是一种从软件工程角度。已被一些可用性专家所采纳的技术。它们通常用的是对系统功能需求的穷尽式描述，通常都具有事务性质，关注于低层次的用户行为和系统响应（Wirfs-Brock, 1993）。系统的精确行为——系统如何精确地响

应——通常不是常规或者具体用例中的一部分。有关系统行为和形式的很多猜想在设计中仍然是隐含的（Constantine and Lockwood, 1999）。用例能实现针对不同用户类型的用户任务的完整分类，但很少提及或者根本不提及到这些任务如何向用户表达或者如何在界面中对它们进行优先级排序。用例可能在标志边缘情况，以及确定产品在功能上是否完整方面非常有用，但它们只适合设计在确认的后期采用。

用基于人物角色的设计来构思设计方案

正如在第 1 章中已经讨论到的，从健壮的模型转换为设计方案实际上包括两个主要阶段：**需求定义阶段**（Requirements Definition）回答的问题主要围绕产品是什么，以及应该是什么；**框架定义阶段**（Frameword Definition）回答产品如何行为，以及它如何被组织以满足用户目标。在本节中，我们将详细讨论这两个阶段，以及由 Robert Reimann, Kim Goodwin, Dave Cronin, Wayne Greenwood 和 Lane Hally 在 Cooper[1]开发的基于人物角色的脚本提纲的方法学。

定义需求

需求定义阶段确定设计是什么：我们的人物角色需要使用什么功能，以及他们完成目标必须访问什么样的信息。下面的五个步骤组成了整个过程：

1. 创建问题和视图声明。
2. 头脑风暴。
3. 标志人物角色的期望。
4. 构建场景脚本提纲。
5. 标志需要。

虽然这些步骤大致按次序执行，但是它们代表着一个迭代的过程。设计师可能期望从步骤 3 到步骤 5 循环几次，直到需求变得稳定。这是这个过程的必要部分，不应该缩短。对每一个步骤的详细描述如下。

【步骤 1：创建问题和视图声明】

在构思的任一过程开始之前。对于设计师来说，有一个对前进方向的清晰的要求是

[1] **译者注** 作者创建的公司名。

非常重要的，即使是一个相当粗糙的、高层次的要求。问题和视图声明提供的正是这种要求，并且在设计过程继续之前，使涉众能达成一致方面非常有用。

在较高层次上，**问题声明**（problem statement）定义设计的目标（Newman& Lamming, 1995）。设计问题声明应该简明地反映需要改变的情况，来服务人物角色和提供产品给人物角色的商业组织。通常因果关系存在于商业关注点和人物角色关注点之间。例如：

设计新的产品X会帮助用户实现目标G，这让用户能以更好地（精确度，效率等）完成X，Y和Z任务，并且不会产生他们现在遇到的A，B，C等问题。这会有力地改善X公司的顾客满意度，并且会增加市场占有率。

X公司的顾客满意率低，市场占有率从去年开始已经下降了10%，因为用户没有充足的工具完成X，Y和Z任务，而完成这些任务则能帮助用户满足他们的目标G。

商业问题和可用性问题之间的关系对于驱动涉众加入设计，以及形成根据用户和商业目标的设计努力方面非常关键。

视图声明（visior statement）作为高层次的设计视图和需求，是问题声明的倒置。你从用户需要开始，转向如何用设计视图满足商业目标：

问题和视图声明中的内容应该直接从研究和用户模型中获得。用户目标和需要应该从首要和次要人物角色中推导出。而商业目标则应该从与涉众访谈中提取。

当你重新设计已有产品时，问题和视图声明最为重要。然而，即使对于新技术产品，或者针对还未开发的市场设计的产品，当你用问题和视图声明系统地表达用户的目标和泪丧时，你也在帮助团队在随后的设计活动中达成一致。

【步骤2：头脑风暴】

在需求定义阶段进行的头脑风暴在某种程度上呈现出一种说反话的意图。作为设计师，你对领域和用户的研究和建模可能已经进行好几天甚至几个星期。如果说你的脑子没有过滤过设计想法，这几乎是不可能的。因此，我们在过程的这个点上进行头脑风暴的原因是，我们至少需要暂时忘记这些想法。头脑风暴的主要作用是在进入脚本提纲阶段之前，尽量消除设计师的偏见，并让设计师准备好在脚本提纲阶段担当首要人物角色。

头脑风暴应该是不受约束、不加评判的。将你所有能想到的（甚至那些没有想到的）想法都说出来，并将这些想法整理、记录下来，妥善保管直到过程的后期。最后，可能不会所有的都有用，但里面可能会有一些精彩的东西适合你后来设计的框架。Holtzblatt &Beryer（1998）描述了一个执行头脑风暴的方法，在启动头脑风暴会议方面非常有用，特别是当团队中包括非设计师时。

不要在头脑风暴中花费太多时间，几个小时（少于半天的时间）对于你和你的团队成员从系统中获得疯狂想法，应该已经足够了。如果你发现你的想法已经开始重复，那

就是停止头脑风暴的好时机。

【步骤 3：标志人物角色的期望】

你的人物角色对产品和使用场景的期望共同构成了产品的人物角色的心智模型。正如我们在第 2 章所讨论的，界面的表达模型——设计的行为和表达——与用户的心智模型尽量匹配是非常重要的。这种匹配甚至比反映产品内部实际上是如何构造的实现模型更重要。

对于每一个首要人物角色，你必须标志：

- 每个人物角色在使用产品体验方面可能有的一般期待和愿望。
- 每个人对产品行为的期待和愿望。
- 影响用户愿望的态度、过去体验、渴望，以及其他社会、文化、环境和认知因素。

从你的人物角色的描述中，可能可以得到充分的信息直接回答其中的一些问题。然而，你还是应该使用研究数据来分析用户主体如何定义和描述对象和动作的语言和语法。这里，对象和动作是用户主体的使用模式的一部分。需要理清的问题如下。

- 主体首先提到的是什么？
- 他们使用哪些动作单词（词组）？
- 对象中哪些中间步骤、任务或者对象他们没有提及到？

在你整理好有关主要人物角色的期望和影响的列表后，对次要人物角色和顾客人物角色也要做类似工作，并且交叉检查相似性和差异。

【步骤 4：构造场景脚本提纲】

脚本提纲是人和他们的活动的故事（Carroll, 2001）。实际上，场景脚本提纲是我们使用的三种类型的脚本提纲中最像故事的那种，它主要关注于人物角色及其心智模型、目标和活动。场景脚本提纲描述了展现使用模式的广泛场景，并且包括了环境和组织（对于企业系统来说）方面的考虑（Kuutti, 1995）。场景脚本提纲建立了在一天中，或者在其他有意义的、能够说明频繁使用和经常使用模式的一段时间段中，首要人物角色和次要人物角色与系统之间（或者通过系统和其他人物角色之间）的主要接触点。正是因为这个原因，场景脚本提纲有时候被称为**日常生活脚本提纲**（day-in-the-life scenarios）。

场景脚本提纲解决了以下的问题（Goodwin, 2002）。

- 产品使用时的设置是什么？
- 它是否会被使用很长一段时间？
- 人物角色是否经常被打断？

- 单个工作站或者设备上是否有多个用户？
- 和它一起使用的其他产品是什么？
- 基于人物角色的技巧和使用的频繁程度，可容许多大的复杂性？
- 为了满足人物角色的目标，人物角色需要完成哪些主要活动？
- 使用产品最终期望的结果是什么？

为了确保场景脚本提纲的有效性，应该让它们范围宽广而浅，应该控制立即探索交互细节的欲望。先绘制出大的图景，并且系统的标志需要是非常重要的。这么做并且遵循相应的步骤能够防止你迷失于设计细节，这些细节以后可能不能一致地结合在一起。

场景脚本提纲不应该像当前这样表达系统行为。这些脚本提纲代表着勇敢的、新的目标导向的产品世界，所以，特别在初始阶段，集中关注目标。不要担心事情如何得以确切完成——你可以在开始阶段将设计当成魔术黑匣子。

有时候可能不只需要一个场景脚本提纲。尤其是当有多个首要人物角色时，但有时候，单个首要人物角色也可能有两个或者多个不同的使用场景。

脚本提纲也完全是文本的。我们还没有讨论形式，而仅仅是用户和系统的行为。这种讨论最好通过文本性的叙述来完成。

一个场景脚本提纲的例子。下面是一个 PDA 和电话合成设备和服务的首要人物角色的场景脚本提纲的第一次迭代的例子。Vivien Strong, Indianapolis 的一个房地产代理商。Vivien 的目标是平衡工作和家庭生活，紧紧抓住每一个交易机会，并且让每一个客户都感觉自己是 Vivien 的惟一客户。

Vivien 的场景脚本提纲可能如下。

1. 在早晨做好准备，Vivien 使用电话来收发电子邮件。它的屏幕足够大，并且网络连接速度很快。因为早上她同时要匆忙地为女儿 Alice 准备带到学校的三明治，这样手机比计算机更方便。

2. Vivien 收到一封 E-mail，来自最新客户 Frank，他想在下午去看房子。Vivien 在几天前已经输入了他的联系信息，所以她现在只需要在屏幕上执行一个简单的操作，就可以拨打他的电话。

3. 在与 Frank 打电话的过程中，Vivien 切换到耳机，这样她能够在谈话的同时看到屏幕。她查看自己的约会记录，看看哪个时段自己还没有安排。当她创建一个新的约会时，电话自动记录下这是与 Frank 的约会，因为它知道她是在与谁交流。谈话结束后，她快速地输入准备看的那处房地产的地址。

4. 将 Alice 送到学校之后，Vivien 前往房地产办公室去收集另一个项目所需要的水管工程的信息。她的电话已经更新了她的 Outlook 约会时间，所以办公室里的其他人知

道她下午在哪。

5. 一天过得很快，当她前往即将看的那处房地产，准备和 Frank 见面时，已经有点晚了。电话告诉她约会将在 15 分钟之后。当她打开电话时，电话不仅显示了约会记录，而且将与 Frank 相关的所有文件，包括电子邮件、备忘录、电话留言、与 Frank 有关的电话日志，甚至包括 Vivien 作为电子邮件的附件发送的房地产的微缩图像。Vivien 按下呼出键，电话自动连接到 Frank，因为它知道 Vivien 马上就要和 Frank 见面。她告诉 Frank 她将在 20 分钟之内到达。

6. Vivien 知道那处房地产的大致位置，但不是很确切。她停在路边，在电话中打开存在约会记录里面的地址。电话直接下载了从她当前地点到目的地的微缩地理图像。

7. Vivien 按时到达了访谈处，并且开始向 Frank 介绍这处房地产。她听到从钱包中传出电话铃声。通常，当她在约会的时候，会自动将电话转接到语音信箱，但 Alice 可以输入密码跨越这一过程。电话知道是 Alice 在打电话，并使用了特别的响铃声。

8. Vivien 拿起了电话——Alice 错过了公交，需要接她。Vivien 给她的丈夫打电话看他能否代劳，可是访问的却是他的语音信箱，他肯定是不在服务区内。她给自己的丈夫留言，告诉他自己和客户在一起，看他能否去接 Alice。五分钟后，电话发出了一个简短的音调，从这个音调，Vivien 可以判断出这个短信是她丈夫发给她的。她看到了丈夫发出的短消息："我会去接 Alice，好运！"

需要注意到的是，这里，脚本提纲处于较高的层次，并没有涉及太具体的有关界面和技术的信息。在技术允许的范围内创建脚本提纲是重要的，但在这一阶段，现实状况的细节内容还不是太重要。总是有机会返回到细节部分，我们最终是企图描述一个最理想的，但仍然可行的体验。值得注意的是，这里，脚本提纲中的活动是如何与 Vivien 的目标相关的，并且尽可能除去可能多的任务。

假设它有魔力。假设界面有魔力是脚本提纲开发早期阶段的一个强大的工具（Cooper, 1999）。如果产品有魔术性效能来满足你的人物角色的目标，界面会多么简单？这种思考方法有助于设计师跳出框架看问题。这种魔术方案显然是不够的，但是，能够以创造性的方式在技术上尽可能地实现接近魔术方案的交互（从人物角色的角度看）是伟大的交互设计的实质。对用户来说，产品实现目标，并只受到最小的侵扰，几乎是魔术般的。前面的脚本提纲中的一些交互看起来有点像魔术一样，但在今天的技术条件下都是可能的。但是，是目标导向的行为，而不仅仅是技术，创造了这种魔术性的效果。

设计技巧 在设计的早期阶段，假设界面有魔术效应。

【步骤 5：标志需要】

在你对你的场景脚本提纲的初稿满意的情况下，你可以开始分析它并且提取人物角色的需求。这些**需求**（needs）包括对象和动作（Shneiderman, 1998），以及场景。作者不倾向于将需求等同于**任务**（tasks）。这里隐含的意思是任务必须通过用户手工完成，而术语**需求**则只是暗示着特定对象需要存在，对于这些对象的特定动作需要在特定场景下发生（无论是由用户发起或是由系统发起）。因此，从 Vivien 的场景脚本提纲角度看，需求可能是：

直接从约会记录（场景）中拨打电话（动作）给某个人（对象）[2]。

如果你习惯于以这种方式提取需求，它能很好地工作；否则，你可以像下面的一些小节描述的那样将它们区分开来。

数据需求 人物角色的数据需求是必须在系统里被描绘的对象和信息。图表、图像、状态标记、文档类型，可以分类、过滤或者操作的属性，以及可以直接操作的图形对象类型（在创作和艺术软件中）都是数据需求的例子。

功能需求 功能需求是针对系统对象必须进行的操作，它们最终会转换为界面控件。功能需求也定义了界面中的对象或者信息应该在何位置和容器内被显示。

场景需求和要求 场景需求描述了对象集合内对象（或者控件集合内集合）之间的关系，以及对象和控件之间可能的关系。这可能包含哪些对象类型在一起显示对于工作流或者满足具体的人物角色目标有意义，以及特定对象如何与其他对象交互（例如，当选择购买的项时，已经选择的项列表需要可见）。其他场景要求包括考虑产品的物理环境（办公室，路上，门内，门外）以及在使用产品时人物角色的技术和能力。

其他要求 在你经历了“假设界面有魔力”这一过程后，获得一个有关你设计的产品的业务和技术的现实要求的坚定想法很重要（虽然，我们希望当技术选择直接影响用户需目标时，设计师能影响技术选择）。

- 业务要求（Business requirements）可能包括开发时间表、规则、价格结构以及商业模型。

---

[2] **译者注** 参见前面 Vivien 的例子。

- 技术要求（technique requirements）可能包括重量、大小、形态要素（form factor）、显示、能量约束和软件平台选择。
- 顾客和合作伙伴要求（customer and partner requirements）可能包括易于使用、维护和配置，承受成本和许可权协议。

现在你的设计团队应该有一个以问题和视图声明形式存在的书面要求；一个产品如何以场景脚本提纲形式满足用户目标的粗略的、创造性的大纲；一个从你的研究、用户模型和脚本提纲中提取出来的需求和要求的简化列表。现在，你可以进一步钻研你的产品行为的细节，并且开始考虑如何表达产品及其功能。你现在可以定义交互的框架了。

定义交互框架

需求定义阶段为设计努力的核心——定义产品的交互框架——提供了舞台。交互框架不仅定义了交互的骨架——它的结构——而且也定义了产品的流和行为。下面的6个步骤描述了定义交互框架的过程。

1. 定义形态要素和输入方法。
2. 定义视图。
3. 定义功能元素和数据元素。
4. 确定功能组和层次关系。
5. 给出交互框架的草图。
6. 构建关键路径脚本提纲。

像前面的过程一样，这不是一个线性的努力过程，而需要反复进行。我们将在下面的小节中，更详细地描述这些步骤。

【步骤1：定义形态要素和输入方法】

创建交互框架的第一步是定义你正在设计的产品的形态要素。它是一个被用来在高分辨率的屏幕上浏览的Web应用吗？它是一个小而轻，分辨率低，以及需要在黑暗的光线下也能看得见的电话吗？它是一个在公共环境中，被很多各种各样的新手使用的公共信息亭吗？对于设计来说这些都意味着什么样的约束条件？回答这些问题为后面的设计努力提供了舞台。

在你定义完了产品的基本姿态（posture，见第8章）后，你应该接着决定对于产品有效的输入方法：键盘、鼠标、键区（keypad）、拇板（thumboard）、接触屏（touch screen）、声音、游戏控制杆（game controller）、远程控制（remote control），以及其他许多种可能

都存在着。哪一种组合对于你的首要人物角色和次要人物角色是合适的？什么是产品的首选输入方法。

【步骤 2：定义视图】

在定义了基本的形态要素和输入方法后，下一步是要考虑产品可以处于哪个首要屏幕或者状态。最初的场景脚本提纲给了你一些感觉，这些首要屏幕或者状态可能是什么样：随着设计的演化（尤其是在步骤 4），它们可能改变或者某种程度被重新调整，但打好初始的基础对于组织你的思想通常很有帮助。如果你知道用户有很多最终的目标和需求，以数据重叠来说它们不是紧密相关的，那么考虑为它们定义不同的视图则是合理的。另一方面，如果你面对的是一个相关需求的集合（例如，为了一个约会，你需要看日历及可能的通信地址），那么，如果形态要素允许，你可以考虑定义一个结合了所有这些的视图。

【步骤 3：定义功能元素和数据元素】

功能元素和数据元素是界面中功能和数据的可见表达。它们是在需求定义阶段标志的功能需求和数据需求的具体表现。在这里，我们特地以真实世界的对象和动作描述这些需求，功能元素和数据元素以用户界面的表达语言的方式描述：

- 屏幕上的窗格、框架和其他容器。
- 屏幕上的控件和物理控件的分组。
- 屏幕上的个体控件。
- 设备上的个体按钮、旋钮和其他物理启示（affordances）[3]。
- 数据对象（图标、列表项、图像、图形）及其相关属性。

在早期的框架迭代中，容器是最需要指定的。随后，当你关注于个体容器的设计时，你会获得更详细的界面元素。

很多人物角色的需求会促使产生多重的界面元素来满足他们的需求。例如，Vivien 需要能够根据联系地址打电话。为了满足这一需求，功能元素必须包括：

- 声音激活（与联系地址相关的声音数据）。
- 可分配的快速拨号按钮。
- 从联系地址列表中选择。
- 从电子邮件头部，约会或者备忘录中选择名字。
- 在合适的场景下自动分配呼叫按钮（约会接踵而至）。

---

[3] **译者注** 作者定义的专门术语，见第 20 章。

多重向量[4]（vector）通常是个好主意，但有时候不是所有可能的向量对于人物角色来说都是有用的。使用人物角色目标，设计原理，模式（参见第 7 章），以及业务和技术约束会筛选掉一些元素列表以满足特定需求。你也需要确定数据元素。Vivien 的数据元素可能包含约会，备忘录，将要做的事的列表和消息。

【步骤 4：确定功能分组和层次关系】

在你有一个好的高层次的功能元素和数据元素的列表之后，你可以开始将它们分为不同的功能单元，并且确定这些功能单元的层次关系（Schneiderman, 1998）。因为这些元素促进具体任务的完成，方法是将这些元素成分组，以在一个任务中和相关任务之间最好地促进人物角色的流[5]。需要考虑的问题包括如下。

- 哪些元素需要占用更多的不动产（屏幕空间），哪些不需要？
- 哪些元素是其他元素的容器？
- 容器应该被如何调整以优化流？
- 哪些元素应该被一起使用，哪些则不？
- 以什么样的顺序使用一组相关的元素？
- 应用什么样的交互模式和设计原则？
- 人物角色的心智模型如何影响组织（Goodwin, 2002）？

最重要的初始步骤是为界面确定高层次的容器元素，以及在给定产品需要的形态要素和输入方法的情况下，它们如何得到最好的安排。必须进行比较或者在一起使用的对象的容器必须相邻。一般来说，代表过程中不同步骤的对象应该相邻，并且顺序排序。在这种汇合处，使用交互设计原则和模式极其有帮助。本书的第 7 章和第二部分提供了在这个阶段起着协助作用的很多原则。

【步骤 5：给出交互框架的草图】

你可能想给出界面中将高层次的容器组织在一起的不同方式的草图。给出草图是一个迭代过程，最好在由一个或两个交互设计师和一个视觉设计师或者工业设计师组成的小的协作团队中进行。界面的视觉化应该在一开始非常简单：框代表每个功能组和/或带有名字并且包含不同区域关系描述的容器。（见图 6-1）。

---

[4] **译者注** 多重向量指多个功能元素，在 GUI 领域，会把一些命令控件称为命令向量，见第 27 章等。

[5] **译者注** 作者定义的术语，一种心理状态，见第 9 章。

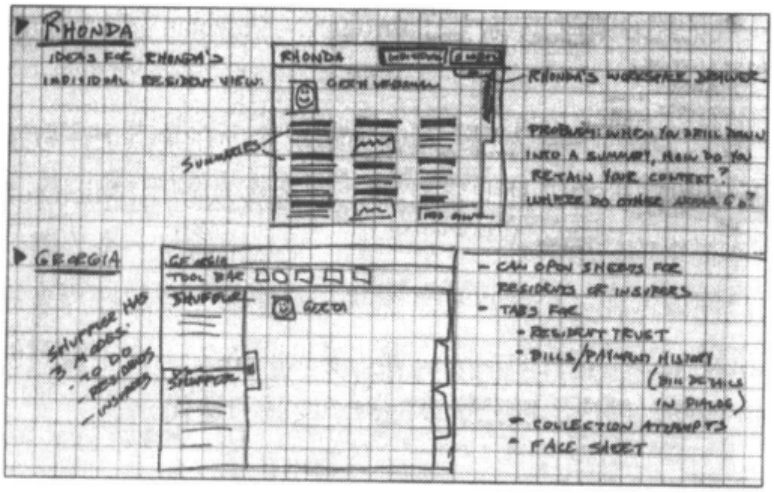


图 6-1 早期框架草图的例子。框架草图应该简单，以矩形、名字、功能区关系的简单描述开始。细节的一些视觉性暗示可以通过提供一些内容的想法给出，但是在这一阶段不要涉及设计细节。

一定要先总观整个高层次的框架，不要分散注意力在某个设计方面的细节上。以后有充足的时间在小件层次细化设计，过于匆忙地进入后面的阶段，可能会带来在后面的设计中缺乏一致性的风险。

【步骤 6：构造关键路径脚本提纲】

关键路径脚本提纲从在场景脚本提纲中暗示而没有解决的细节中得到。关键路径脚本提纲，通过人物角色最频繁使用的界面，通常是每天在任务层次上描述了首要动作和路径。例如，在电子邮件程序中，浏览和写电子邮件是关键路径活动，而配置新的电子邮件服务器则不是。

关键路径脚本提纲通常需要最大的交互支持。新用户必须能够快速地掌握关键路径交互和功能，所以需要内置教学系统来支持他们（参见第 18 章和第 27 章）。然而，因为这些功能被频繁使用，用户不会长久依赖于教学系统：他们很快要求有快捷方式。更进一步，随着用户变得非常有经验，他们希望能定制日常使用的交互，使它们遵从自己的个性化工作风格和偏好。

【脚本提纲和情节串连图板】

不像面向目标的场景脚本提纲，关键路径脚本提纲更倾向于面向任务；它们更关注于在场景脚本提纲中被广泛描述和暗示的任务细节（Kuutti, 1995）。这并不意味着目标被忽略了——目标和人物角色需求是设计过程中不变的尺度，用于剪裁不必要的任务以及对必要的任务流水线化。然而，关键脚本提纲必须以确切的细节描述每一个主要的交互

的精确行为，并且用于对每一个主要的路径进行走查（walkthrough）（Newman &Lamming, 1995）。

通常，关键路径脚本提纲开始于白板，并且达到合理的细节层次。在某点上，取决于界面的复杂性和密度，掌握基于计算机的工具非常有用。作者喜欢使用 Microsoft PowerPoint 作为协助关键路径脚本提纲的情节串连图板的工具。交互中的每一个步骤，无论是在用户与系统间，多个用户间或者在多个用户与系统间，都能以一个幻灯片来描绘，浏览这些幻灯片的过程提供了对交互一致性的现实检查（参见图 6-2）。在不需要创建过多细节的情况下，PowerPoint 能够足够快、低分辨率地用于绘画和迭代。

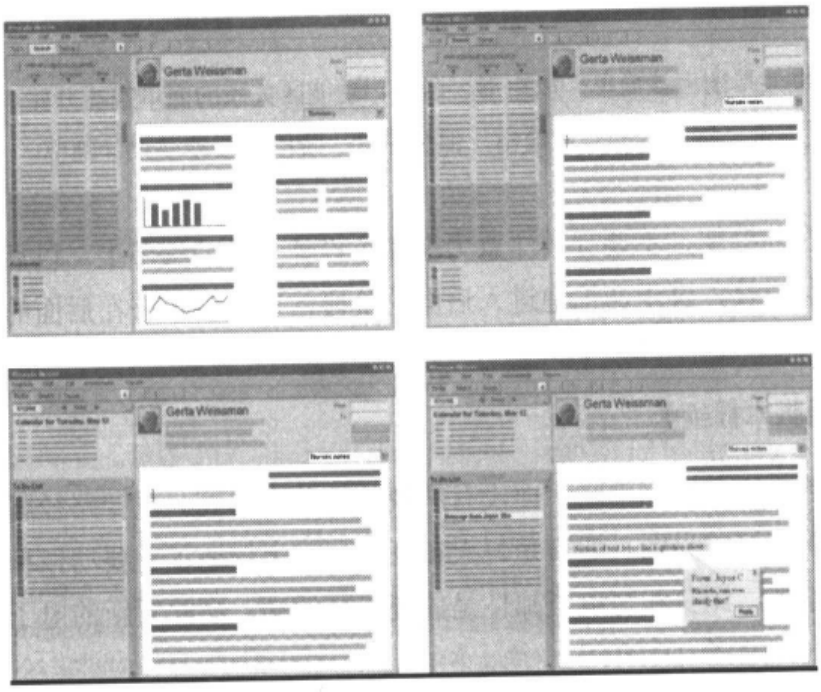


图 6-2 共享的医疗系统软件产品 Orcas 的情节串连图板的例子

【假设系统是人】

就像“假设系统有魔力”是用于创建概念层次的场景脚本提纲的强大工具，“假设系统是人”是适用于关键路径脚本提纲的交互层次的强大工具。该原理非常简单（我们将在第 14 章详细讨论它）：与数字化产品的交互在基调和有益性方面应该类似于与一个礼貌的、体贴的人之间的交互（Cooper, 1999）。在构造交互的时候，你应该问问你自己：首要的人物角色是否得到产品人性化的对待？一个仔细、体贴的交互看起来应该像什么样？软件以何种方式在不妨碍其他的情况下，提供有用信息？如何最小化人物角色实现目标的努力？一个乐于助人的人会怎样做？

【原则和模式】

将关键路径脚本提纲变换为情节串连图板（以及步骤 3 的元素分组）的关键是应用一般的交互原则和详细精确的交互模式。这些工具利用了数年的交互设计知识——不利用这些知识等于重新创造一切。关键路径脚本提纲提供了交互设计固有的自顶而下的方法，从主屏幕到子窗格或者对话框持续地对更为详细的设计结构进行迭代。原则和模式添加了可以平衡这种过程的自底而上的方法。原则和模式可以用于在所有的设计层次组织元素。第 7 章讨论了详细的有关原则和模式的使用和类型，第三篇的章节提供了适用于过程中这个步骤的充分有用的交互原则。

优化形式和行为

当获得了稳定而坚实的框架定义之后，设计师认为后面的设计开始平滑地按序进行：关键路径脚本提纲的每一次迭代都添加了细节，加强了产品的流和整体一致性。在这个阶段，可以顺利地进入优化阶段，在这里设计变换为更具体的形式。优化过程包括三个步骤：

1. 起草外观草图。
2. 构造确认脚本提纲。
3. 设计定稿。

在这一阶段，原则和模型在确保设计有一个好的、正式的行为结果方面仍然很重要。第 19 章和第三部分中的很多章都为优化阶段提供了有用的原则。设计团体在优化阶段的开始就参与进来也非常关键——现在，设计有一个可靠的概念性和交互性的基础，设计师的输入对于能够创建一个忠实于概念的最终设计是至关重要的。

【步骤 1：外观设计】

当交互框架进入关键路径脚本提纲阶段，交互设计师应该和视觉界面设计师一起工作以开发视觉设计的品牌和功能构件（更基本的视觉设计原则，参见第 19 章）。类似地，工业设计师应该加入这个阶段。视觉设计师和工业设计师应该与交互设计师一起走查关键路径脚本提纲。不同的设计角度应该是互补的，并且能产生更好、更合意的产品，只要所有的团体能够与在其基础上创建脚本提纲的人物角色的特征一致。

视觉和工业设计师（或者有这些技巧的界面设计师）应该开始将详细的线框和情节串连图板转换为有代表性的、完整的位图屏幕（参见图 6-3）。

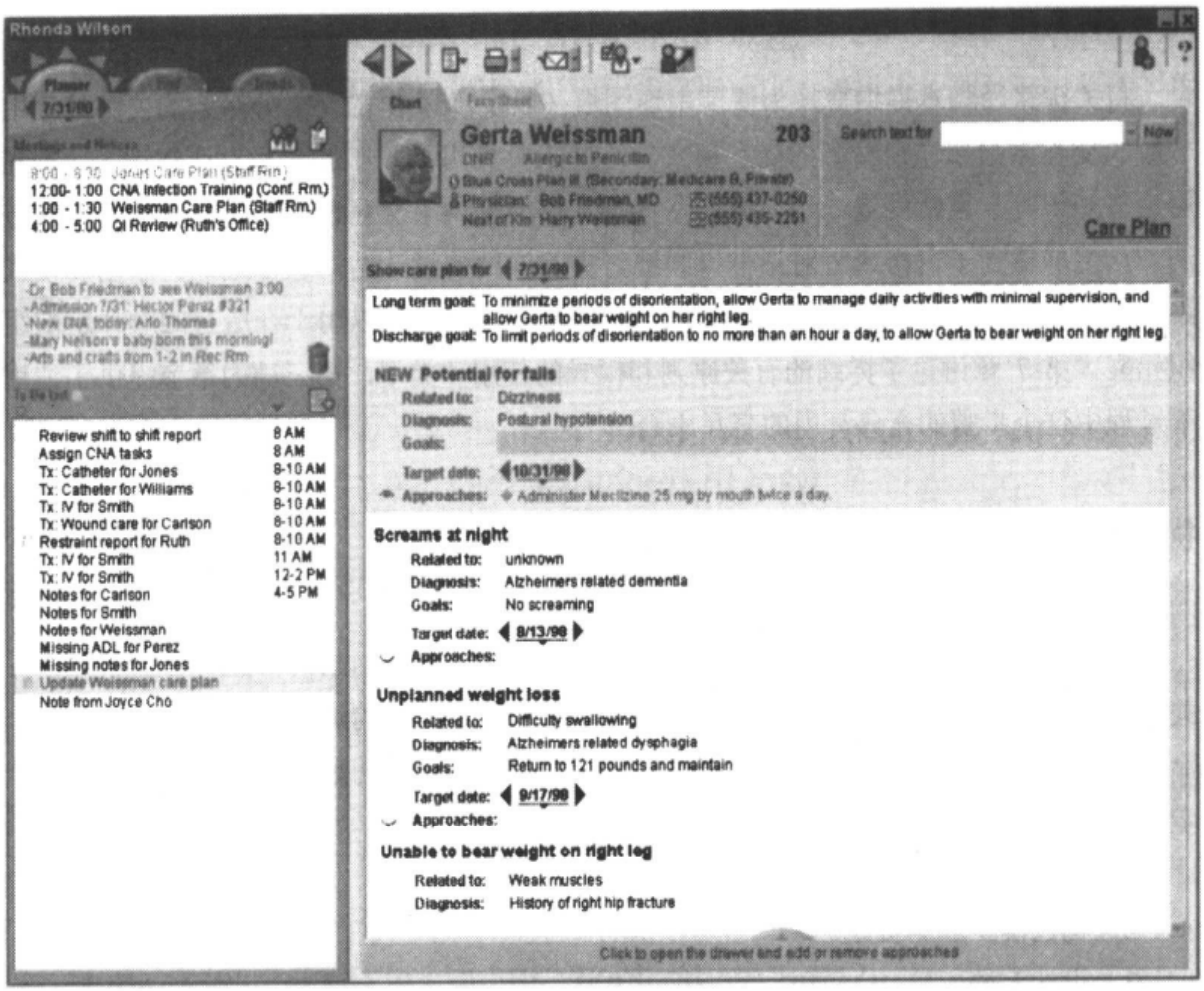


图 6-3　基于前面图中的情节串连图板创建的完整的位图屏幕。注意因为像素和屏幕分辨率的原因，这些布局会有轻微的改变。视觉或者交互设计师需要在这个阶段一起紧密地工作，以确保对设计的视觉改变能够继续加强这种产品行为，并且满足首要人物角色的目标。

每一个首要的视图和对话框都应该被提到。视觉设计师应该创建风格指南，这样开发者能够对低层次和低优先级的界面部分应用一致的设计元素，因为设计师可能缺少时间和资源去亲自完成这些界面。

【步骤 2：构造确认脚本提纲】

如果用户频繁地进行某个任务，它的交互必须被完整地绘制。同样，如果任务非常关键但不频繁地进行，它的交互，虽然设计的时候有不同的目标，也必须有良好的设计。在你对关键路径脚本提纲进行了大量的走查，并制作情节串连图板后，应该将注意力转移到界面中那些较少被使用的边角部分。开发这些脚本提纲的方式与关键路径脚本提纲相同，通常，对它们进行如下优先级排序。

关键路径变体　关键路径变体脚本提纲（key path variant scenarios）是沿着人物角色

决策树上的某一点，从关键路径中分离出来的，较少人使用的交互。这可能包括仍然属于主要的调色板或者工具条上较少被使用的功能；文档较少使用的视图；对于产品的基本操作来说不关键的主要对话框。关键路径变体有着与关键路径脚本提纲相同的要求，但更多地强调教学性，因为它们较少被使用。

必要使用 必要使用脚本提纲（Necessary use scenarios）包括必须完成的所有动作，但它们不会频繁地被使用。清除数据库，配置，以及做出其他异常的请求可能属于这一类。必要使用的交互要求教学性，因为它们很少被使用：用户可能忘记了如何访问这些功能，以及完成相关的功能。然而，用户不会要求存在并行的交互习惯用法，诸如键盘等，因为这些功能很少被使用到。而且因为它们不被频繁使用，必要的功能不包括被用户定制。

边缘情况使用 边缘情况使用脚本提纲（Edge case use scenarios），正如它的名字隐含的，是可选和不频繁使用的活动。程序员通常强调确认脚本提纲处理的边缘情况，因为他们有着将所有的功能都视为同等重要的天然倾向，但边缘情况不应该是设计努力的关注点。设计师不能忽略边缘功能和情况，但他们需要的交互是低优先级的，并且，通常深深隐藏在界面里。虽然代码有可能成功或者失败地处理边缘情况，但产品的成功或者失败则主要看能否成功处理日常使用和必要情况的能力。

【设计定稿】

在你通过确认脚本提纲检查了设计之后，要开始和你的视觉设计团队，以及工业设计团队一起确定产品最后的外观设计。从这时开始，你可以产生一系列的输出，包括一份打印出来的形式和行为规范。形式和行为规范包括屏幕印刷品的陈列（mock-up）和有充分标注的细节可用于编码的情节串连图板。取而代之，同样你可以以 HTML 或者诸如 Flash 或者 Director 创建交互原型而服务于同样的目的。不管你交付什么样的设计选择，你的团队在实现过程中应该继续与编程团队一起工作，确保设计视图精确地从设计文档转变为忠实于视图的最终产品。

7

综合好的设计：原则和模式

在前三章，我们已经讨论了实现出众的交互设计的过程。但什么使得设计出众呢？设计是否满足用户目标和需求（而不牺牲业务目标或者忽略技术约束）是衡量设计是否出众的标准之一。但是，设计的什么属性使得它能够达到这些目标呢？是否是那些设计所拥有的通用的、与场景相关的属性和特性构成一个“好”的设计呢？

作者坚信这些问题的答案在于使用交互设计原则——设计有用且可用的形式和行为的指导准则，以及使用交互设计模式——具体类型的设计问题的可仿效和通用的解决方案。本章将更加详细地阐述这些思想。除了关注于设计的原则和模式以外，我们必须也考虑一些更大的**设计规则**（design imperatives）来为设计过程打好基础。这些会在本章的最后提及。

交互设计原则

交互设计原则是一些通常能够用于解决行为、形式和内容问题的指导准则。它们代表着产品行为的特征，这些特征能够更好地帮助用户实现他们的目标，并且用户在这么做的时候感觉到自己能胜任工作并且自信。在整个设计过程中，都应该应用原则来帮助

我们把在脚本提纲迭代过程产生的任务转变为界面中定型的结构和行为。

原则减少工作量

原则为之服务的主要目标之一是优化用户在使用系统时的体验。在生产率工具和其他非面向企业的产品方面，这种体验的优化意味着减少工作量（Goodwin，2002a）。可以被减少的工作种类包括以下几个部分。

- 逻辑工作——对文本和组织结构的理解。
- 知觉工作——识别形状、大小、颜色和表达的视觉布局和语义。
- 记忆工作——回忆密码、命令向量，数据对象和控件的名字和位置，以及对象之间的其他关系。
- 物理/运动神经工作——敲打键盘的次数，鼠标移动的程度，使用的姿势（单击、拖、双击），用户输入模式之间的切换，需要的导航程度。

本书中大多数原则都尝试在向用户提供更高程度的反馈和对场景有用的信息的同时，减少工作量。

在不同细节层次运作的原则

设计原则在三个不同组织层次上运作：概念层次、交互层次和界面层次。虽然，本书主要关注交互层次的原则，但在一定程度上也谈到了所有的三个层次。

- 概念层次原则帮助我们定义什么是产品，以及它如何符合它的首要人物角色所需要的广泛使用场景。第3，8，9章讨论了概念层次的设计原则。
- 交互层次原则帮助我们定义产品在一般情况和具体情况下应该如何行为，第二、三、五、六、七部分的一些章节讨论了一般的交互层次原则，第八部分讨论了Web和设备界面的交互层次原则。
- 界面层次原理帮助定义了界面的外观。第四部分和第五、六部分的部分章节讨论了界面层次的原理。

大多数交互设计原理跨平台，虽然一些平台，诸如Web和嵌入式系统，因为平台设置的一些特殊约束条件有一些特殊的考虑。

原则与风格指南

风格指导（style guide）根据产品品牌和可用性指导准则严格定义了界面的外观（进一步有关外观的讨论见第 19 章）。他们通常关注于详细的小件（widget）层次：对话框里有多少 Tab 键？按钮突出显示状态看起来应该怎么样？控件和标签之间的像素空间应该有多大？要为产品创建一个好的、协调的外观，这些问题都必须回答。但是，风格指导通常都没有涉及太多，更大的有关产品应该和如何行为的问题。

作者建议设计师在有可用的风格指南，以及调整交互细节时，要注意风格指南，但在行为设计中存在一些更大和更有趣的问题，很难在风格指南中找到位置。本书剩下的大多数章节为设计有良好行为的交互系统提供了这些指导准则。它们包括一些原则和模式，但并没有以任何特定的风格指南的方式。本章的剩下部分及本书的第二篇和第三篇，都将集中讨论怎样才能构成好的行为等规范这些重要问题。

交互设计模式

设计模式主要服务于两个重要的功能。第一个功能是捕捉有用的设计决策并且将这些决策通用化以便在将来解决相似的问题类型（Borcher，2001）。在这个意义上，模式同时代表着对设计知识的捕捉和形式化，这一方式有着多种用途，包括减少在新项目上投入的设计时间和精力，对项目的新设计师进行培训，或者如果模式应用得非常广泛，则可以用以培训这个领域的新设计师。

虽然模式的应用在设计的教学和效率方面非常重要，但使得模式令人激动的关键因素还是它们代表着用户及模式解决的活动类型的最优或者准最优交互。

交互和建筑模式

与流行的模式工程应用相比较，交互设计模式与 Christopher Alexander 在其独创性著作“A Pattern Language”（1997）和“The Timeless Way of Building”（1979）首先构思的建筑设计模式更为密切。Alexander 企图捕获一些他称之为无名的质量的构造块，正是这些建筑设计的精要为居民创造了安康。是人性因素将交互设计模式（以及建筑设计模式）与工程设计模式区分开来，工程设计模式的惟一关注点是代码的复用。

交互设计模式不同于建筑设计模式的惟一而重要的一点是它们的关注点不同，不仅

是元素的结构和组织不同，响应用户活动的元素的动态行为和改变也不同。我们可能忍不住将这些不同简单地视为随着时间而产生的变化，但这些变化是非常有趣的，因为它们响应着人类的活动。这将它们与在广播或者电影媒体（它们有着不同的设计模式集合）中可见的预定的时序变化区分开来。Jan Borcher2（2001）对这些界面设计模式的适当描述是：

（交互设计）模式指的是物理元素之间的关系，以及在这些物理元素上发生的事件。界面设计师，像城市设计师一样，努力创建一个建立了对身在其中的人有正面影响的行为模式的环境。“永恒”的结构可以比得上用户界面诸如“透明”和“自然”的特性。

交互设计模式的类型

像其他大多数设计模式一样，交互设计模式可以采用层次性组织结构，从系统级到单独的界面小件级。像原则一样，它们能够应用于组织的不同层次（Goodwin, 2002a）。

- **姿态模式**（postural pattern）适用于概念级，帮助确定产品相对于用户的整体产品姿态。产品姿态的概念其最显著的模式将在第 8 章中讨论。
- **结构模式**（structural pattern）解决了与信息显示和访问管理相关的问题，以及数据和功能的容器以何种可视化操作方式来最好地满足用户目标和场景。结构模式包括在第 6 章中讨论的视图、窗格和元素组。我们在本章将更详细地讨论一些与结构模式相关的问题。
- **行为模式**（behavioral pattern）解决了与个体功能、数据对象，或者这些对象组的具体交互相关的广泛问题。大多数人们想到的系统和小件行为可以归入此类，并且很多这些低层次的模式会在第二篇和第三篇中讨论。图 16-1 提供了一个 Cooper 公司倡导的比较新的行为模式的极佳例子：一个为基于属性的检索引擎创建的自然语言输出查询构造。

结构模式或许是最少文档化的模式，但它们仍然得以广泛应用。最为常用的高级别结构模式之一是：在 Microsoft Outlook 中，导航窗格在左边，概观窗格在右上方，而详细的窗格在右下方（见图 7-1）。

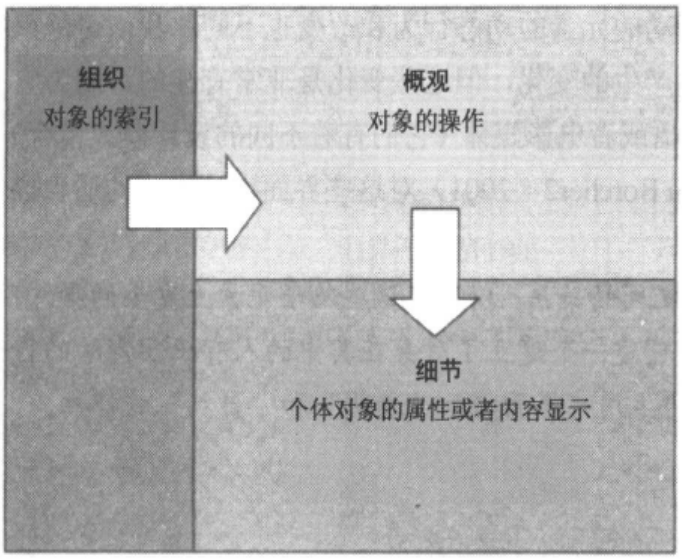


图 7-1 Microsoft Outlook 使用的主要结构模式在工业界的很多不同的产品领域被广泛地使用。左边的垂直窗格提供了导航，并且驱动右上方的概观窗格。对这个窗格进行选择则弹出了右下方详细的或者文本内容的窗格。

这个模式最适合于全屏幕的应用，因为它们要求用户访问不同种类的对象，操作这些成组的对象，显示个体对象或者文档的详细内容或者属性。这一模式使这些能以单屏幕的方式流畅地显示，而不需要其他窗口。很多电子邮件客户程序使用这种模式，并且这种模式的变体出现在很多创作和信息管理工具中。在这些工具中，快速访问和操作多种类型的对象非常普遍。

结构模式、模式嵌套和预定制的设计

结构模式中通常包含一些其他的结构模式。你可能会想像：在给定清晰的用户需求的情况下，一个综合的结构模式目录允许设计师相当迅速地组装连贯的、目标导向的设计。虽然这些论断在某种情况下是真实的，这一点作者在实际中已经观察到，然而模式能够以简单的方式机械地组装，从来都不现实。正如 Christopher Alexander 敏捷地指出的（1979），结构模式是预定制构造的对立面，因为场景在定义模式实际显示的形式方面绝对重要。模式被应用的环境非常关键，包含、组成、毗邻该模式的其他模式也是同样的。对于交互设计来说，这也是事实。每个模式的核心存在于被代表对象之间，以及这些对象和用户目标之间。模式的精确形式当然对于每个事例来说可能不同，并且定义这些模式的对象会自然地因领域不同而不同。但对象之间的关系在本质上仍然是相同的。

交互设计规则

除了需要前面提到的原则类型，作者感觉到也需要一些更为基础的原则在整体上指导

设计过程。下面一些较高层次的设计规则（由 Robert Reimann, Hugh Dubberly,Kim Goodwin David Fore 合 Jonathan Korman）适用于交互设计，但同样也适用于其他设计学科。

交互设计师应该创建具有以下属性的设计方案：

- 伦理的（能体谅人、有帮助）
  不伤害
  改善人的状况
- 有意图的（有用、可用）
  帮助用户实现他们的目标和渴望。
  包容用户场景和能力
- 注重实效的（能生存的、可行的）
  帮助委托的组织实现它们的目标
  满足业务和技术需求
- 优雅的（有效、巧妙的和情感的）
  代表最简单的完整方案
  拥有内部的一致性（自我表白的、可理解的）
  合适的容纳、刺激认知和情感。

伦理的交互设计

当设计师被要求设计对人的生活有着实质性影响的系统时，交互设计师面对着伦理问题。他们可能对系统用户有着直接的影响，或者对那些生活在某种程度上被这些系统影响的人有着间接的影响。因为他们设计工作的成果不是简单的政策的说服性沟通或者产品的营销。实际上，它意味着执行政策或者创建产品本身。设计用户直接使用的系统相对直观，但是系统对对于那些间接影响的人的影响有时候更难计算。

【不要伤害】

在理想情况下，不应该伤害任何人，尤其不能伤害用户。交互系统可能伤害的类型包括：

- 人际伤害（失去尊严、蒙受羞辱）。
- 心理伤害（困惑、不舒适、沮丧、强迫、厌烦）。
- 物理伤害（痛苦、伤害、剥夺、死亡、安全被降低）。
- 环境伤害（污染、生物多样性的减少）。

- 社会和社会学伤害（剥削、不公正的产生和永恒化）。

前三个在某种程度上比后面两个更为容易解决，并且基本上是本书第三部分的主题。前两个要求对研究的领域有较深的理解。他们也要求涉众确定这些仍处于能够通过项目解决的范围。（明显地，最后两个不是大多数产品的问题，但是读者肯定能够想到一些例子与此相关，诸如海底石油钻塔的控制系统或者选举-投票阅读系统）。

【改善人类状况】

不伤害，当然对于一个真正伦理的设计是不充分的。与此同时，它应该改善事物。交互系统可能能够广泛改善的情况类型有：

- 增强理解（个体、社会和文化）。
- 增强个体和团体的效率 / 有效性。
- 改善个体和组织之间的交流。
- 减少个体和组织之间的社会—文化紧张关系。
- 改善公平（财政、社会和伦理）。
- 平衡社会的一致性和文化的多样性。

设计师在从事新的设计项目时，应该将这些问题放在心中。应该考虑做好的机遇，即使他们轻微偏出了范围。

【有意图的交互设计】

本书的主旋律是基于对用户目标和动机的有意识设计。如果没有别的问题，第一部分章节描述的目标导向的过程应该能够帮助你实现有意图的设计。然而，有意识的部分不仅是理解用户目标，也包括理解它们的限制。在这方面人物角色[1]可以起到很好的作用，因为你在研究和创建人物角色时观察到的行为模式为让你很好地理解用户的力量及其盲点。目标导向的设计帮助设计师创建在用户较弱时支持用户，而当用户强壮时，则加强他们的力量的产品。

注重实效的交互设计

在架子上堆满了灰尘的设计规范对任何人都没有用处。设计在创建时应该有价值。一旦被创建，他们需要在真实世界应用。并且一旦被应用，它需要为它的拥有者产生高利润的回报。在设计过程中，考虑业务目标和技术需要是非常紧要的。这并不意味着设

[1] **译者注** 作者创建的目标导向设计的术语，见第 5 章。

计师在表面上必须接受涉众和程序员告诉他们的一切：在商业、工程和设计团对中需要一个积极的“有关边界在哪和什么样的领域的是灵活的”对话。程序员经常宣称设计是不可能的，他们所指的是在给定时间表的情况下设计是不可能的。销售业务组织可能会在不完全理解他们的用户是否会接受它们的计划分支的情况下，创建业务计划。设计师，他们已经收集了详细的用户定性数据，可能会有独特的有关业务模型的观念。当在设计、业务和工程之间存在相互信任和尊重的关系时，设计应该能够最好地工作。

优雅的交互设计

优雅在词典里同时定义为“优雅而受约束的美丽风格”，以及“科学精确、整洁和简洁”。作者相信设计中的优雅，或者交互设计，同时结合了这些思想。

【表达最简单的完整方案】

设计最经典的元素之一是形式的经济，以最少实现最多。在交互设计中，这种经济性扩展到了行为：用户的简单工具集合允许他们实现更多的事情。对于好的设计来说少就是多，并且设计师应该尽力在较少增加形式和行为的情况下解决设计问题，并且与你的人物角色的心智模型一致。这个概念对于程序员来说众所周知，它们承认好的算法应该清晰而简短。

【拥有内部的一致性】

好的设计有着统一整体的感觉，它们所有的部分都平衡而协调。设计糟糕的软件，或者就根本没有设计的软件，通常看起来就像通过随意的编制零散的部件而拙劣地修补在一起。这通常是基于实现模型构造的结果，这种情况下，不同的开发团队在相互不沟通的情况下在不同的界面模块上工作。这与我们想实现的恰恰相反。目标导向的设计过程提供了一个理想的创建内部一致的设计环境，在这种环境中，产品概念在较高地层次被理解为一个整体，并且接着迭代地优化为详细细节。

【合适地包容、刺激认知和感情】

很多传统设计师经常提到期望，以及他们在设计交流和产品中的重要性。这本身没有错，但是作者感觉到单纯地强调情感，（虽然情感本身很复杂），他们可能有时候看到的仅仅是整个图景的一部分。

当设计服务于某种意图，尤其是当产品定位于企业，或者他们的目的是高度技术或者具有专门性，期望是需要满足的较为狭窄的感情。我们几乎不能让操作辐射治疗系统

的技术人员感觉对系统充满期望。相反我们希望他能够对系统控制的相当危险的装置小心或者谨慎。因此，我们作为设计师尽量让他们关注于病人和他们的治疗。因此，对于我们称之为期望的事物，作者相信优雅（就优雅的角度而言）意味着用户无论在什么场景下，都受到认知和情感上刺激或者支持。

本书的剩下章节枚举了在交互设计中作者认为最重要的交互原则——毋庸置疑，你可能会发现更多，但这些超出了你开始时所需要的。本章也包含贯穿全文的设计模式。

第一部分的章节已经讲述了目标导向的交互设计实践背后隐藏的过程和概念。本章将提供有益的设计观点，这些会帮助你将这些知识转换为极佳的设计，而无论你处于什么领域。

第二部分

去除障碍，达到目标

8 软件姿态

9 和谐与流

10 消除附加工作

11 导航和调整

12 理解撤销

13 重新思考“Files”和“Save”

第二篇

设计行为与形式

使用今天的大多数数字产品就像驾驶已经滚下悬崖的汽车一样：你必须从窗口爬出，没有光线，引擎发出令人怀疑的沉闷噪音，薄金属片在不合适的时候飞出去了。在我们的生活中，制造的这些人工制品在集成了越来越多的技术的同时，为什么必须变得越来越难以使用和理解？

在今天的交互设计中，我们有很多明显的实验，可以观察成功和失败——但设计师通常几乎无法就细节问题达成一致，而更不用提更大的问题。在交互设计中奏效的或许是猜测或者模仿。令人困惑的是不应该以这种方式工作。本篇的章节将解决交互设计中一些更大的问题。

8

软件姿态

大多数人都有某种主要的行为姿态来适合自己的工作角色：士兵时刻警惕；收费员冷淡又无趣；演员过着富丽堂皇的生活；服务代表乐于助人。同样，程序也有向用户展现自己的主要方式。

程序可能大胆或者保守，色彩丰富或者单调，但它们这些特性应该有着特定的、目标导向的理由。它的风格不应该来自设计师或者程序员的偏好。程序的表达方式影响用户与它们的关系，而这种关系强烈影响着产品的可用性。如果程序的外观和行为与其意图相冲突，那么看起来会不和谐、不适宜，就像茶杯里的毛发或者婚礼上的小丑。

程序的外观和行为应该反映它们的使用方式，而不能采用其他任意的标准。程序的行为态度——向用户表达自身的方式——就是它的姿态。从姿态的角度看，程序的外观不是审美的选择，而是行为的选择。程序的姿态正是它行为的基础，无论你做出什么样的审美选择，它都应该与姿态协调。

界面姿态体现了它的行为态度，这会对后面的设计做出很多重要的指导。作为一个交互设计师，在设计中，你首先应该确保界面表达的姿态适合程序的行为，以及用户的行为。本章讨论桌面、Web 和设备应用的不同姿态。

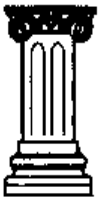
桌面的姿态

桌面应用可以分为四种姿态：**独占**（sovereign）、**暂时**（transient）、**精灵**（daemonic）和**辅助**（auxiliary）。因为每一种姿态都描述了一组不同的行为属性，也描述了不同的用户交互类型。更重要的是，这些分类是设计师设计界面的起点。例如，独占姿态的应用除非采取独占的行为方式，否则用户的感觉会很糟糕。Web 和其他非桌面的应用有着它们特有的姿态，我们会在本章的最后详细讨论。

独占姿态

那些全屏的，长时间内独占用户注意力的程序就是有**独占姿态**（sovereign posture）的应用程序。独占式应用程序提供较多的功能和特性，并且用户会长时间持续使用它们。这种类型应用的典型例子是字处理、电子表单及电子邮件软件。很多垂直应用也是独占式应用程序，因为它们长时间占据屏幕，用户与它们的交互非常复杂，需要全神贯注。使用独占式应用程序的用户经常处于一种称为**流**（flow）的状态。用户通常以最大化的方式使用独占式应用程序（我们会在第 25 章讨论窗口的状态）。例如，很难想像在 3×4 英寸的窗口大小内使用 Outlook——这个尺寸不适合它的主要工作：创建、查看电子邮件和日程安排（见图 8-1）。

独占式应用程序通常由用户长时间地连续使用。作为主要工具，它占据了用户的主要工作流程。例如，在创建幻灯片的整个过程中，你的 PowerPoint 占据整个屏幕，即使需要其他程序辅助完成任务，PowerPoint 也保持独占姿态。



公理：独占式应用程序的用户是永久的中间用户。

独占式行为的含义很微妙，但只要经过思考就会非常清晰。它最重要的含义是独占式程序的用户是中间用户（参见第 3 章）。每个用户只会在一段时间内是新手，与他最终花费在产品使用上的时间相比，这段时间很短暂。可以肯定的是，新手不得不痛苦地克服初始学习曲线的上升阶段，但与用户使用应用程序的时间相比，他熟悉这个程序的时间还是很少的。

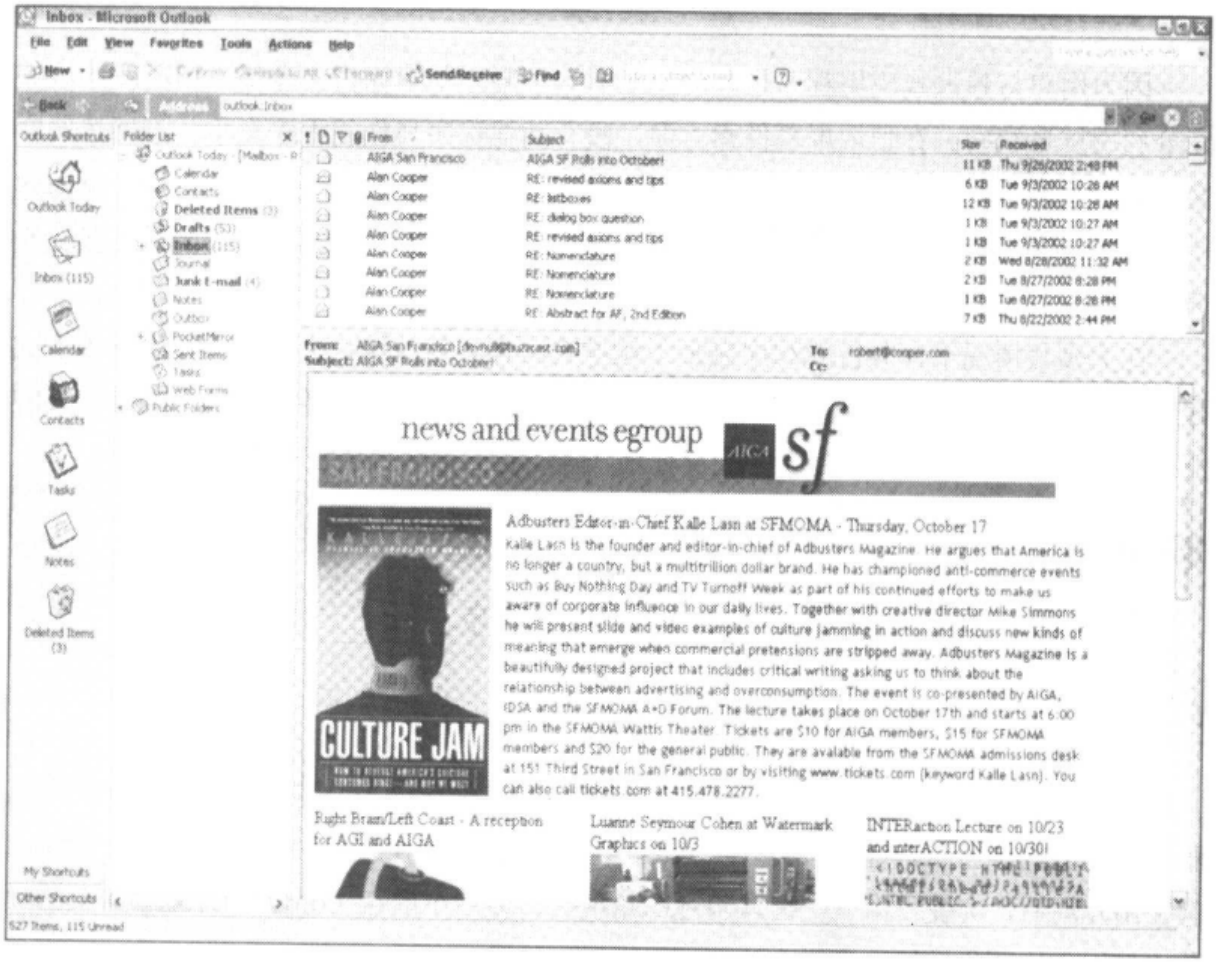


图 8-1 Microsoft Outlook 是一个经典的独占式应用程序例子。它长时间占据屏幕，不间断地与用户交互，并且多个相邻的窗格用于导航和提供帮助信息，它需要占据整个屏幕。

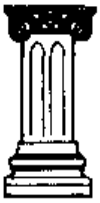
从设计师的角度看，这意味着程序应该为永久中间用户的使用而优化设计，而不为新手（或者专家）牺牲速度和效能，支持笨拙但容易学习的常规做法在这里不适用了，因为只提供了一些简单的效能工具。当然，如果你能在不影响中间用户交互的情况下，提供一些容易学习的习惯用法，那就最好了。

第一次使用的用户和中间用户之间，还存在很多只是偶尔使用的用户。这些偶尔使用的用户不能忽略。然而，独占式应用程序是否成功，仍然取决于频繁使用它的中间用户，除非它能同时满足中间用户和不熟悉的用户。WordStar，一个较早的字处理软件，就是一个好的例子。它在 20 世纪 70 年代后期和 80 年代早期占据了字处理软件市场，因为它能很好地服务于中间用户，虽然它对于新手用户和不熟练用户来说，使用极其困难。WordStar 公司一直非常兴旺，直到它的竞争对手在为中间用户提供了相同能力的同时，也让不熟练的用户觉得好用。WordStar 因为无法跟上竞争，所以很快萎缩而变得无足轻重。

【充分利用像素】

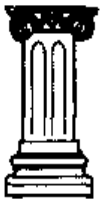
因为用户与独占式应用程序之间的交互占据了用户与计算机的会话，所以程序应该充分利用屏幕空间。在使用屏幕像素方面，没有其他程序与你的程序竞争。不要浪费空间，但也不要不好意思使用你所需要的空间来完成任务。如果需要四个工具条，就用四个工具条。在其他姿态的程序中，四个工具条可能会太复杂，但独占姿态的应用在像素使用方面有着正当的理由。

在大多数情况下，独占式程序以最大化方式运行。除非用户有特殊的要求，独占式程序应该默认为最大化（全屏）形式。程序应该可以调整大小，并且在其他屏幕配置的情况下能够合理工作，但它必须为全屏使用（而不是其他情况）进行优化。



公理：为全屏使用优化独占式应用程序。

因为用户会长时间地盯着独占式应用程序，所以你应该小心地减弱界面视觉的颜色和纹理。颜色少一点，保守一点。大尺寸、色彩丰富的控件让新用户看起来可能很酷，但使用几周后，会变得俗气。但从长远的角度看，使用小虚点（tiny dot）或者加重的颜色比使用大块颜色更好，那样会让控件组织得更紧凑。



公理：独占式界面应该使用保守的视觉风格。

用户可能长时间盯着相同的工具栏、菜单和工具条，会得到一些纯粹因为熟悉而产生的位置感。这样，设计师可以自由地用更少的像素做更多的事情。工具条和其他控件可以比正常的更小。辅助控件，像屏幕分割器、标尺和滚动条可以更小，空间上也可以更紧凑。

【丰富的视觉反馈】

独占式应用程序可以作为一个大平台，为用户创建一个视觉反馈丰富的环境。你可以大量在界面上添加额外的信息。屏幕底部的状态条、边上的滚动条、标题栏，以及程序可视区域的角落，都可以充满程序状态、数据状态、系统状态的视觉指示，以及其他

更多有用的用户行为暗示。然而，一定要小心：在保证丰富视觉反馈的同时，要避免创建混乱的界面。

因为它们在屏幕上微妙的显示方式，第一次使用的新用户不会注意到这些结果，更别提理解它们。然而，稳定使用几个月以后，新手用户会开始发现它们，思考它们的含义，并且试探性地去使用。这个时候，用户愿意花一些精力来学习，如果提供某种方法让用户更容易理解这些结果，他不仅会成为一个更好的用户，而且满意度也会提高。用户使用程序的能力随着理解增长。将这种丰富性添加到界面中，就像将不同的调料添加到肉汤里——让整个食物变得更丰富。我们将在第 34 章讨论丰富的无模态视觉反馈。

【丰富的输入】

独占式应用程序同样也可以享受丰富的输入。每个常用的部分，都应该可以用多种方式操作。直接操作、对话框、键盘助记符和键盘加速键都很合适。你可以使用直接操作习惯用法要求用户精巧的运动技巧。屏幕上的敏感区域可以仅仅是一些像素，因为你能够确信用户的使用环境：舒适地坐在椅子上，手臂放在桌子上，在有弹力的鼠标垫上稳定地滚动鼠标。

设计技巧 独占式应用程序可以使用丰富的输入。

在程序窗口的边边角角使用控件。在一个喷气式飞机的驾驶舱中，用得最多的控件就直接在飞行员面前；那些偶尔使用或者在紧急状态下才使用的控件可以在扶手、头顶及边上的面板上找到。在 Word 中，Microsoft 将最常用的功能放在两个主要的工具条上（见图 8-2），而将经常使用但视觉上会引起混乱的功能放在屏幕底部水平滚动条左边较小的控件上。这些控件改变整个视觉显示的外观——正常视图、页面布局和大纲视图。

新手不会经常使用它们，如果碰巧触发了它们，他们会感到非常困惑。将它们放在屏幕的底部，新手用户几乎不会注意到。它们隔离的位置微妙而无声地提示，使用它们时应该小心。更有经验的用户，在理解和控制程序方面会更有信心，会开始注意到这些控件，而对它们的目的感到好奇。当他们准备好接受可能的结果时，会试探性地使用这些控件。从这些控件放置的位置，可以非常精确而又有用地映射到它们的使用。

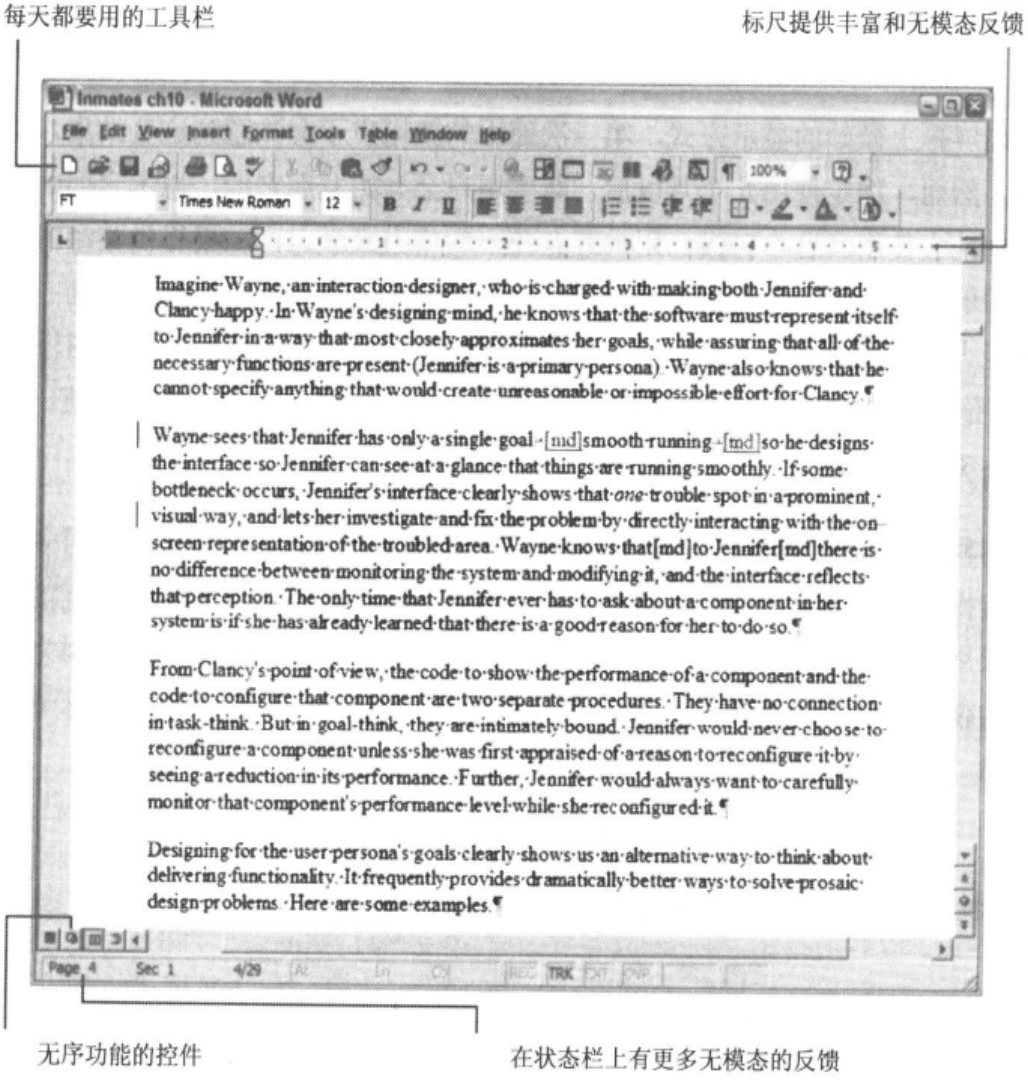


图 8-2 Microsoft Word 将控件同时放在应用程序的顶部和底部。位于顶部的控件与位于底部的控件相比更温和。后者进行了隔离，因为它们可能会导致严重的视觉混乱。

用户不会欣赏延迟的交互。就像鞋子里进了一粒沙，重复几次之后，一两秒的延迟都会让用户感到很痛苦。费时的功能是可以接受的，但在用户正常使用产品的过程中，它们不能是需要经常或反复使用的功能。例如，如果将用户的工作保存到磁盘需要花一段时间，用户很快就会认为这种延迟是不合理的。而如果对矩阵求逆或者改变文档的整个格式花上几秒钟，用户不会愤怒，因为他们明白这是一个很大的任务，另外，用户也不会经常这么做。

【以文档为中心的应用】

“独占式应用程序应该占据整个屏幕”这一公理对于程序内的文档窗口来说也成立。

程序内部包含文档的子窗口应该始终最大化，除非用户有其他要求。

设计技巧：在独占式应用程序中，让文档视图最大化。

很多独占式应用程序程序也是以文档为中心的（它们主要的功能包括创建和观看数据丰富的文档），这很容易在二者之间产生混淆，但实际上它们是不同的。大多数文档是$8\frac{1}{2}$乘以 11 英寸的大小，并且不会完全占据标准的计算机屏幕。要尽量显示尽可能多的数据，自然要求全屏幕的姿态。例如，如果构造的文档是一个 32×32 像素的图标，则以文档为中心的程序不需要占据整个屏幕。程序的独占性不是来自其以文档为中心的特性，也不是来自文档的大小——它来自程序使用的性质。

如果一个程序操作文档时仅进行一些简单的功能，比如扫描图像，那么它不是一个独占式应用程序程序，并且不应该表现出独占式的行为。这样单一功能的应用有着它们自己的暂时姿态。

暂时姿态

暂时姿态（transient posture）的程序打开又关闭，用一套非常有限的附加控件，展现一些单一的功能。通常在使用某个独占式应用程序的过程中，暂时姿态程序在需要的时候调用、出现并完成自身的工作，接着迅速地消失，让用户继续正常的工作。

暂时式应用程序的显著特征在于它们的临时性质。因为它们不会长时间停留在屏幕上，用户不会有机会非常熟悉它们，所以，程序的用户界面不需要细致、清晰而显眼地显示控件，不能有错误。界面必须清楚地说明它在干什么。这里不需要艺术性很强而模糊的图像或图标，需要的是有着精确图例的大按钮，以稍大而容易阅读的字体清楚地显示。

设计技巧：暂时式应用程序必须简单清晰，并且切中要点。

虽然暂时式应用程序可以在你的桌面上单独运行，但它通常对独占式应用程序起着辅助支持作用。例如，在用 Word 编辑文档时，使用浏览器定位和打开另外一个文件就是

典型的暂时场景。设置麦克风音量也是一样（见图 8-3）。因为暂时式应用程序借用了独占式程序的空间，所以它必须考虑独占式程序，除了必要的空间以外，不能占用其他屏幕空间。相对于独占式程序长期占据屏幕，可以周密、充分地计划这一特点而言，暂时式应用程序只可以进行一些类似周末露营的活动，它们不能图形化或者临时性地将自身部署在屏幕上，它是软件世界的出租车。

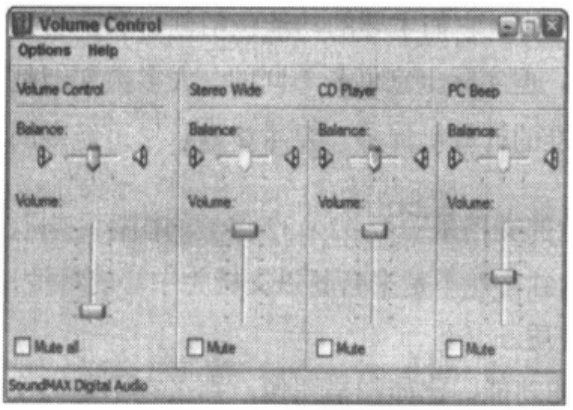


图 8-3 Windows 音量控制就是一个典型的暂时程序例子。它在一些更具有独占性的活动中少量而快速地使用，接着消失。Microsoft 可以采取多种手段来加强面板的暂时性，如保留更多的空白空间，拉长控件，使用色彩更丰富的调色板与独占式应用（通常暂时式应用程序在它上面启动）区分。

【明亮而清晰】

尽管暂时姿态的应用程序必须节约使用屏幕上的空间，但它们界面上的控件与独占应用相比，应该更大。这种色彩斑斓的视觉设计，几个星期就会让独占式应用程序的用户感觉沉闷，但暂时式应用不会长时间地停留在屏幕上，因而不会困扰用户。相反，在程序弹出时，醒目的图形反而有助于用户更快地定位。暂时式应用不应该使用单调的颜色，相反，应该选择明亮的颜色，这有助于区分暂时式应用与它覆盖的独占式程序，如有弱色调的阴影则更好。暂时式应用应使用更明亮的颜色和更显眼的图形来清晰地传递它的意图——就像在高速行驶的情况下，司机需要大而明亮的反射路标，免得他在每小时 100 公里的速度下拐错弯。

暂时式应用应该把指令放在表面上显示。用户可能一个月才会使用一次，很可能会忘记各种选择的含义。与其使用标题为“设置”的按钮，不如将按钮大一些，加上“设置用户偏好”的标题。这使含义更清晰，并且，这样的按钮让人放心。同样，在暂时式应用程序上不应该使用缩写——任何词汇都应该完整地拼写出来，避免用户困惑。如，用户应该能够轻松地判断打印机处于繁忙状态，或者某段音频大约需要播放五秒钟。

【尽量简洁】

在用户调用暂时式应用程序时，他所需要的信息和功能设施应该直接显示在程序单个窗口上。让用户的注意力集中在这个窗口，不要再转到其他窗口或对话框来完成程序

的主要功能。如果在设计暂时式应用程序时，你发现自己在增加一个第二视图或对话框，可能这就意味着设计需要复查。

设计技巧：暂时式应用程序只使用一个窗口和视图。

小的滚动条和琐碎的指向—单击—拖放（point-click-and-drag）界面不适合暂时式应用程序，尽量少要求用户的运动技巧。对于简单的功能来说，简单的按钮就足够了。任何直接可操作的对象必须足够大，而且容易移动：至少 20 个平方像素。控件应偏离窗口的边界。不要使用窗口的底部、状态栏或者暂时式应用程序的边界。相反，将控件聚在一起，个性化地放在窗口的主要部分。

当然应该提供键盘界面，但它必须简单（见图 8-4）。复杂度不能超过 Enter、Escape 和 Tab 键。你可能也需要添加方向箭头，但必须是在需要的情况下。

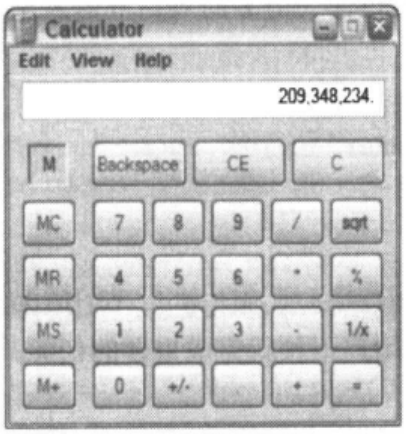


图 8-4 Windows 和 Mac 平台上的计算器(calculator)附件是一个暂时式应用的好例子。它有着大而显眼的按钮和功能，程序也能够使用键盘操作。这很好，因为硬件风格的按钮虽然对于新手用户来说可以接受，但某种程度上难以使用。

当然，暂时式应用程序也存在例外的情况，不止一个主题，但这种情况很少。如果暂时式应用程序完成的不止是单一的功能，则界面应该能可视化地表达。如程序导入或者导出图形，界面应该通过显眼的颜色或者其他图形平均而可视化地分为两半。一半包括导入控件，而另外一半包含导出控件。它们两者都应该清楚地贴上标签。无论你想做什么，都不要添加更多的窗口或者对话框。

记住，任何暂时式应用程序都可能被调用，用来协助管理独占式程序的某些特性。这意味着暂时式应用程序可能会挡住独占程序上很重要的信息，因为它们位于独占式程序之上。这意味着暂时式应用程序必须是可移动的，而且必须有一个标题。

在暂时式应用程序中，尽量地将管理成本保持最低是至关重要的。用户只需要启动程序，调用功能，接着结束程序。在这样的交互中，强迫用户进行与工作无关的窗口管

理任务完全不合理。

【记住状态】

对于暂时式应用程序和独占式应用程序来说，帮助用户最好的方式是让程序拥有记忆。如果暂时式应用程序能够记住它最后一次使用时的状态，那么很好，相同的尺寸和位置对下一次也适用。这好过任何默认设置。无论用户给程序什么样的形状和位置，在程序下一次调用时，原有形状和位置都应该再现。当然，逻辑设置也同样如此。

另外，如果程序使用起来真的很简单，那么干脆直接指定它的形状——别管边框（可以直接调整的窗口边界）。节省工作，尽量简化程序，降低复杂性（但要小心，防止滥用）。这里的目标不是减少程序员的工作——这仅仅是间接的好处——而是尽量减少用户的复杂性。如果程序的功能不要求重新调整大小，并且程序的整体尺寸小，那么“简单就是美”的原理比平常显得更为重要。例如，Windows 和 Mac 平台上的计算器就不能重新调整大小。它始终是确定的大小和形状。

你无疑已经意识到，几乎所有的对话框实际上都是暂时式应用程序。你会发现，前面关于暂时式应用程序的所有指导准则同样适用于对话框的设计（更多的对话框信息见第 30 章和第 31 章）。

精灵姿态

不与用户交互的程序是**精灵姿态**（daemonic posture）的程序。在用户不可见的情况下，这些程序静静地，且不可见地在后台服务，可能不需要用户干预就完成重要的任务。打印机驱动程序就是一个很好的例子。

你可能会想到，对精灵式应用程序的用户界面讨论必然很短。然而，程序员经常让精灵式应用程序拥有更适合独占式程序的全屏控制面板。例如，像 Excel 那样设计传真管理程序就是一个严重的错误。这种情况的另外一个极端是在需要进行调整时，用户经常无法访问精灵式应用程序，结果因为不能调整，用户将感到无休止的挫折。

暂时式应用程序控制功能的执行，而精灵式应用程序通常管理进程。你的心跳无须有意识的控制，相反，它在背后完全自主地进行。像心跳控制的过程一样，精灵式应用程序通常处于完全看不见的状态，只要你的计算机仍在工作，就能正常完成任务。然而，与你的心脏不同的是，精灵程序必须经常安装和卸载，有时候，它们还必须进行调整以适应改变的环境。精灵式应用程序就是在这种情形下与用户交流的。在正常情况下，用户和精灵式应用程序之间的交互在本质上是暂时的，所有暂时式应用程序的设计规则对精灵式应用程序也成立。

对于精灵式应用程序来说，暂时式设计“将程序的意图和用户可获得的选择范围和含义告知用户”的设计原则更加重要。很多情况下，用户甚至意识不到（无意识）精灵式应用程序的存在。如果你承认这一点，那么很容易理解为什么在不合适的上下文下报告程序的状态会导致混乱。因为很多程序要完成的是深奥的功能，如打印机驱动程序或者通信集中器——必须特别小心处理来自它们的消息，防止用户困惑或者误解。

那么，对于其他姿态的程序视为理所当然的一个问题，对于精灵式应用程序而言就显得非常重要：如果程序在正常情况下是不可见的，那么在用户偶尔需要的情况下，应该如何调用用户界面？最常用的方法是用屏幕程序图标来表示它们，例如 Windows 状态区域（系统托盘区）或者 Mac OS 菜单条的最右端。在用户不需要的时候，将图标显眼地放在用户面前实际上表示一种公开的冒犯，就像在某人车子的挡风玻璃上粘贴广告一样。如果你的精灵程序不需要天天配置，那么不要将它的图标放在主屏幕上。现在，Windows XP 已隐藏那些并不活跃的精灵式应用程序。只有会连续地提供有用状态信息的情况下，精灵式应用程序才应该一直使用图标。

Microsoft 做了一些这个方面的折中，在任务栏的最右端设置了一个称为状态栏的区域，属于精灵姿态的程序可能会驻留该区域。这个区域，也称为系统托盘区，程序员们经常滥用它，将它设置为独占式应用程序的快速启动区域。在 Windows XP 中，Microsoft 的标准是：只有状态图标才可以出现在该状态区域（快速启动区域紧挨着任务栏的开始按钮）。除非用户自己选择，否则只会显示那些报告状态改变的活跃图标。其他图标都会隐藏起来。这些决定对于暂态式应用程序非常合适（见图 8-5）。

Mac 和 Windows 上都用控制面板来对精灵式应用程序进行有效配置，它是可以启动的应用程序，可以用来配置精灵式应用程序的暂态程序。这为用户提供了访问这些“进程为中心”应用的一致场所。

图 8-5 Windows XP 任务栏上的状态区域。鼠标指针指向一个图标，该图标代表监视 Windows Messenger 连接的精灵进程。图标提供了无模态的视觉状态，也提供了 Windows Messenger 的启动点（launch point）。它是公理“在有输出的地方允许输入”（第 10 章）的变体。

辅助姿态

混合了暂时和独占特征的程序具有**辅助姿态**（auxiliary posture）。辅助姿态程序像独占姿态程序一样，长时间出现，但它起到的仅仅是辅助作用。它像暂时式应用程序一样，占据的屏幕空间小，通常位于其他应用程序之上。Windows 任务栏、时钟程序、UNIX 平台上的性能监视程序，以及 Mac 的 Stickies 等，都是辅助姿态程序的好例子。经常使用即时通信应用的人们也以辅助的方式使用它们。在 IE 浏览器的 Windows XP 版本中，Microsoft 已经意识到，在用户浏览 Web 时，音频流也能够播放这样一个辅助功能。因此它已经将音频播放器集成到浏览器旁边的窗格上（见图 8-6）。

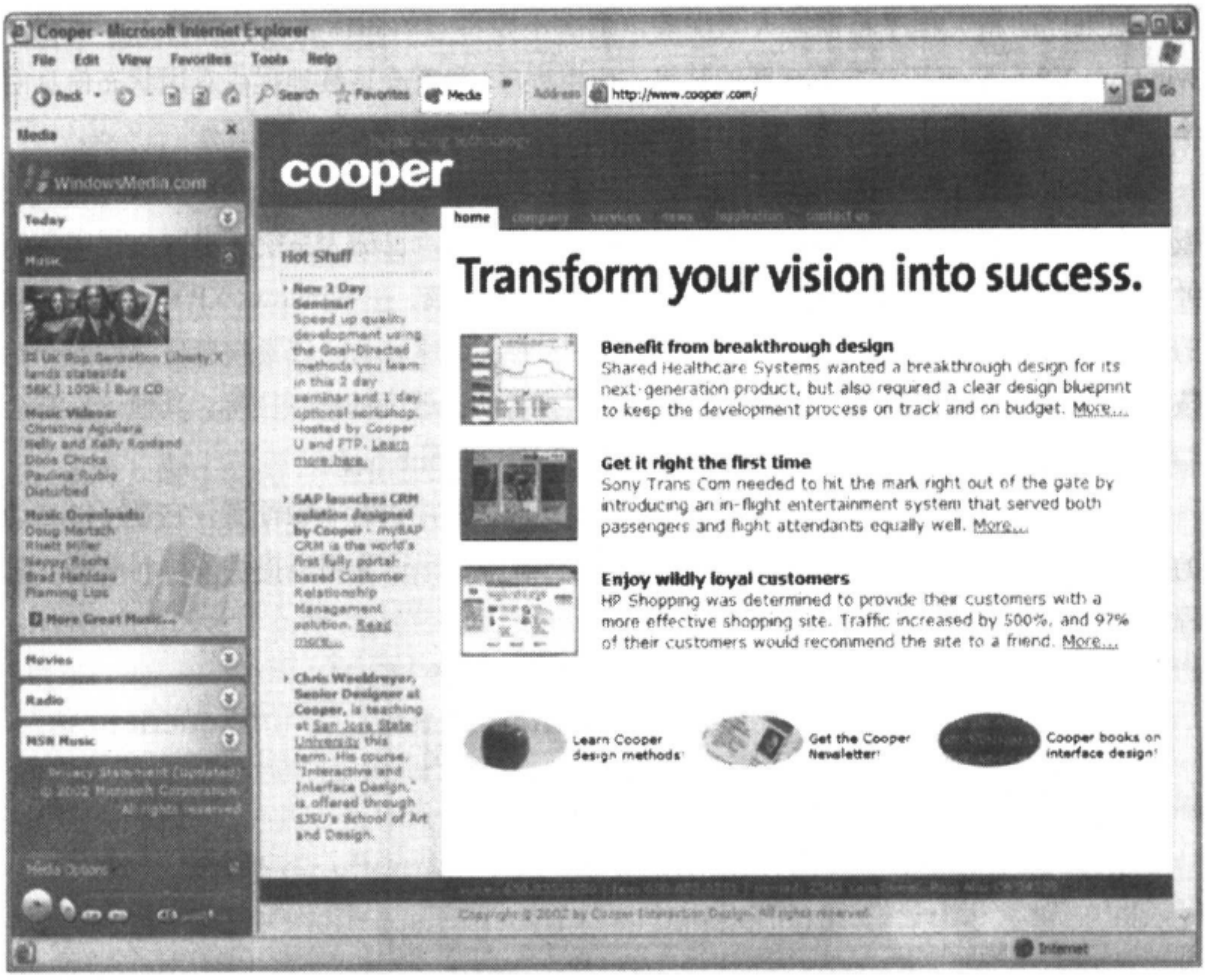


图 8-6 Microsoft 已经意识到浏览 Web 时，音频流起着潜在的辅助角色作用，并且将音频播放器集成到因特网浏览器旁边可折叠的窗格上。音频控制本身也能作为单独的窗口在浏览器窗口内弹出。

辅助程序通常是运行中进程的无声报告者。虽然一些程序，如 Stickies 或者股票报价机，用来显示用户感兴趣的其他数据。在某些情况下，这种报告可能是实际管理进程以

外完成的功能，但也不一定完全如此。如辅助应用可能是监视正在使用或者可以获得的系统资源数量。程序经常不断地显示小的柱状图来反映当前可以获得的资源。

进程报告辅助程序必须简捷，而且在报告信息时非常显眼。它们必须优先考虑重要的独占式应用程序，并且在必要时快速退出。

辅助程序不是用户注意力的焦点，这种特性属于它的宿主应用程序。例如，考察一下自动呼叫分配应用（ACD）[1]。ACD 用来给客服代表团队平均分发呼入，受过培训的客服代表们接受订单，提供支持，或者两者兼顾。每个客服代表在使用的计算机上运行与他工作相关的应用。主要因为系统采购的原因，这个应用是一个独占姿态的应用，ACD 程序是在该独占姿态程序之上的辅助应用程序。例如，销售代理用它来接收来自潜在用户使用免费号码的呼入，服务代表使用的订单输入程序是独占姿态的，而 ACD 程序是辅助姿态的，运行在输入程序之上，对呼叫做出反应。ACD 程序在使用屏幕空间方面应该非常保守，因为它会干扰所覆盖的独占式应用程序。它们长时间占据屏幕，只能够提供一些小的特性。换句话说，辅助应用的控制可以根据独占式应用程序的敏感性来设计。

Web 的姿态

就姿态而言，设计师可能会认为 Web 有必要和桌面应用不同。虽然存在结合不同姿态的变体，但基本的四种姿态确实包含了大多数 Web 站点及 Web 应用的需求（我们将在第 37 章详细地讨论 Web 设计的细节）。

面向信息的站点

纯粹信息化的站点，除了页面之间的导航及有限的搜索外，不需要复杂的事务，但必须平衡两种力量：需要显示合理密度的有用信息，还需要让第一次访问和不常访问站点的用户容易学习和导航。这意味着在信息站点中存在着独占属性和暂时属性之间的平衡关系。以哪种姿态为主，主要取决于目标人物角色是谁，以及他在访问站点时的行为模式：他们是不常访问的或者仅仅访问一次的用户，还是频繁访问的用户，每周或每天都浏览内容？

站点内容的更新频率在某种程度上确实影响行为：与每月更新的站点相比，每天更新的信息站点自然会吸引频繁的用户。不频繁更新的站点可能更多地用于偶然参考（用

[1] **译者注** ACD 为 Automatic Call Distribution 的缩写。

户主观认为信息不太具有专题性），而不是反复使用，因此，应该采取暂时姿态，而不是独占姿态。另外，通过关注特定用户的访问频率，站点自己能够调整到更独占的姿态。

【独占属性】

详细的信息显示最好通过假定独占姿态来实现。通过假定用户使用全屏，设计师能够充分利用可以获得的空间，清晰地表达站点自身的信息，让用户得到指引的导航工具和提示。

Web 上采用独占姿态惟一的美中不足在于确定什么样的全屏分辨率才合适。实际上，这个问题也同样存在于桌面应用软件中。在这一点上，桌面应用软件和 Web 的惟一差别在于，Web 站点没有方法影响用户对屏幕分辨率的选择。然而，投资昂贵的有生产性质的桌面应用软件用户可能会确信，他们应购买合适的硬件来支持软件的需求。因此，Web 设计师需要在早期做出决定，他们希望支持的最小分母（意指最小分辨率）是什么。作为选择，他们必须为更为复杂的站点编码，对高分辨率的屏幕进行优化，但在较低分辨率的显示器上仍然可用（不需要水平滚动）。

【暂时属性】

你的首要人物角色对站点访问得越少，站点就越需要采取暂时姿态。在信息站点中，这要求易用和导航清晰。

用户可能对一些不常参考的站点设置书签，应该让用户能够对任何页面设置书签，这样，用户以后在任何时候都能可靠返回。

用户可能会间断地访问每周或每月更新材料的站点，所以，这些站点的导航应该非常清晰。如果站点能够通过 cookie 或者服务端的方法保留过去用户行为的信息，并且基于用户过去的兴趣来组织信息，那么，这能够显著地帮助不常访问的用户在导航成本最低的情况下发现他们需要的信息。

事务网站和 Web 应用

事务网站、Web 应用与信息站点一样，可能也有独占立场和暂时立场之间的平衡关系。因为交互层次明显更复杂了，因此，这非常具有挑战性。

同样，一个好的切入点是首要人物角色的目标和需要：他们是否是消费者？谁会随意地使用站点，以一周还是一个月为基础？还是一些日常基础工作必须使用这些站点的雇员（相对于企业或者 B2B Web 应用）？作为雇员工作重要成分的事务站点应考虑完全作为独占姿态的应用程序。

另一方面，电子商务、在线银行及其他面向消费者的事务站点，必须像信息站点一样，以类似的方式平衡独占姿态和暂时姿态。实际上，很多消费者事务站点有着明显的信息特征，因为用户喜欢研究和比较产品、投资及其他需要进行事务处理的内容。对于这些类型的站点，导航清晰非常重要，访问支持信息，以及对事务进行流水线化处理也同样重要。通过一次单击订购，良好的搜索和浏览能力，在线的条目评论，推荐列表，持久[2]的购物车，以及跟踪最近浏览条目的能力，Amazon.com 很好地解决了这些问题。如果 Amazon 存在失误之处，那可能就是它想做的太多了，一些接近页面底部的导航链接可能很少得到单击。

Web 门户

早期的搜索引擎允许人们去寻找和访问分布在整个 Web 世界的内容和功能。它们实际上是**门户**（portal）（该单词的原意）——也就是到达其他地方的途径。实际上在**导航门户**（navigational portals）上什么也没有发生。你进入，到达某个目的地，然后再返回。它们专门用于快速访问没有关联的信息和功能。

如果用户不太需要经常通过导航门户访问，那么，暂时姿态是合适的，提供清晰简单的导航控制，接着让开。如果用户需要更频繁的访问，那么，合适的是辅助姿态：小而持久的链接面板（像 Windows 任务栏）。

随着门户站点的演化，它们提供越来越多的内容和功能，远远超越了原先作为迈向其他目标跳板的作用。**消费者导向**（consumer-oriented）的门户提供了与某个专题相关的内容和功能的统一访问方式。**企业门户**（enterprise portals）门户提供了对重要企业信息和商业工具的内部访问。在这两种情况下，门户的意图实质上是创建用户能够访问特定类型信息，以及完成特定类型工作的环境：**环境门户**（environmental portal）。实际的工作在环境门户中完成。信息从各种不同的来源中收集起来，进行处理。不同的工具协同起来完成统一的目的。

环境门户的各个元素需要以某种方式组织起来帮助用户实现具体的目标。当一个门户创建了工作环境，它也创建了某种场所。门户不再是去某些目的地的路径，而是一个目标。环境门户的合适姿态是独占姿态。

在环境门户中，个体元素的功能实质上就像同时运行的小的应用程序一样——因此，元素自身也有自己的姿态：

---

[2] **译者注** 表示可以保存状态。

- **辅助元素**（auxiliary elements）：环境门户中的大多数元素都有辅助姿态。它们通常表示的是用户需要持续访问的一组信息聚集（诸如动态状态监控），或者简单的功能（小的应用程序、链表等诸如此类）。辅助元素实际上是环境门户的关键构造块。因此，独占门户由一些辅助姿态的小应用程序构成。
- **暂时元素**（transient elements）：除了辅助元素，环境门户通常也提供暂时服务。它们很简单，解释性元素丰富，用户在较短的时间内按需使用。设计师应该将暂时姿态赋给任何简短而临时访问的嵌入式门户服务（诸如待办事件列表[3]，或者包跟踪状态显示[4]），这样，它不会和门户自身的独占/辅助姿态冲突。相反，它成为自然的临时扩展。

当将传统的应用迁移为环境门户时，主要的设计挑战之一是，将应用分为合适的具有暂时姿态或者辅助姿态的门户服务。独占的全浏览器应用程序在门户网站中不合适，因为一旦启动，人们不会把它们看做门户的一部分。

其他平台的姿态

手持设备、公用信息亭和基于软件的电器，各自的软件姿态问题稍有不同。像 Web 界面一样，这些平台也通常包含多种姿态之间的平衡关系（与这些平台相关的其他设计问题见第 38 章）。

公用信息亭

公用信息亭大而全屏幕的特性看起来偏向于独占姿态，但有几个理由表明情况并非如此。首先，公共信息亭的用户通常是第一次使用的用户（除了 ATM 用户以外），并且在大多数情况下不是日常用户。第二，大多数人不会在公共信息亭面前花太多时间：他们完成简单的事务：搜索，获取需要的信息，接着离开。第三，大多数公共信息亭在显示屏的边上使用触摸屏或者屏外按钮，没有你期望的支持独占式应用程序的高密度数据输入的机制。第四，公共信息亭的用户很少舒适地坐在位置理想的显示器之前，而是在光线很强和干扰很大的公共场所。这些用户行为和约束条件应该将公共信息亭的姿态偏向暂时，简单的导航，大尺寸的控件，吸引注意力的丰富视觉显示，以及对功能的提示。

---

[3] 译者注　To-do list。

[4] 译者注　packet-tracking status display，作者提到的可能是网络包的跟踪。

教育性和娱乐性公共信息亭和需要严格暂时姿态的事务公共信息亭不同。在这种情况下，对公共信息亭环境的调查比简单地完成事务或者搜索时更重要。在这种情况下，更高的数据密度、更复杂的交互和视觉变化有时候能够引入正面效果，但需要认真考虑输入机制的限制，免得用户不能在界面之间成功导航。

手持设备

为手持设备做设计是存在硬件限制的锻炼：输入机制、屏幕大小、分辨率及电源消耗。很多设计师已经意识到与手持设备相关的一个重要观点："手持设备不是一个孤立的系统"。像 Palm 和 Pocket PC 这样的个人信息管理设备，它们是桌面系统的附属设备，更多地用于浏览信息，而不是自己进行高密度输入。虽然对于很多手持设备来说，能够买到折叠式键盘，这实际上是将它们变为桌面系统（带小屏幕）。对于辅助设备来说，辅助姿态对于大多数经常使用的手持设备应用程序来说是合适的——典型的个人信息管理器（PIM）、电子邮件，以及 Web 浏览。更少或者更临时使用的手持设备应用程序（像闹钟）可以采纳更为暂时的姿态。

手机是一种有趣的手持设备。它们不是辅助设备，而是主要的通信设备。然而，就用户界面姿态而言，手机实际上是暂时的。你可以尽快打一个电话，接着在对话期间放弃某个界面。因此，对于手机来说，最好的界面是非可视化的。语音激活是拨打手机的最完美方式。打开手机的盖子可能是最有效的应答方式（或者再次使用语音激活用于免提使用）。手机界面越暂态，越好。

近些年来，手持数据设备和手机日趋融合。这种融合的设备使得手机越发复杂，而数据操作越发困难。但最近的设备类型，比如 Handspring Treo 实现了中间状态的成功。在某些方面，它们借助设备的附属性质协助电话输入信息，使得电话可用性增强：Treos 使用桌面的联系信息来同步设备的电话簿，因此避免了先前痛苦的数据输入步骤，并且加强了手机功能的暂时性质。在为这些设备做设计时，承认数据功能的辅助性质和手机功能的暂时性质是重要的，使用它们加强相互的可用性（数据拨号应该具有最低限度的暂时性，而数据浏览应该是辅助的）。

电器

大多数电器都有极其简单的界面，并且严重地依赖于硬件按钮和拨号来管理电器的状态。然而在一些情况下，大一些的电器（最明显的是洗衣机和烘烤机）会弹出彩色的

LCD 触摸屏，允许丰富的输出和直接输入。

电器界面，像上节提到的手机界面，应该主要考虑为暂时姿态。这些界面的用户很少了解技术，因此，应该选择最简单、最直接的界面。这些用户也可能习惯于硬件控件。除非通过触摸屏能够获得前所未有的易用性，拨号和按钮（有一些适当的语音反馈，以及一些只读显示或者甚至硬件灯的视觉反馈）可能是更好的选择。很多电器设计犯的错误是，将大量新的且用户不想使用的功能堆在一起，放到新的数字模型里。不将它变得简单，结果，这些简单的 LCD 触摸屏成为不能工作的令人困惑的控件集合。

电器界面采用暂时立场的另一个理由是电器用户将事情尽量交给机器具体去做。和事务型公用信息亭用户一样，他们对探索界面或者获取附加信息一点不感兴趣。他们仅仅想让洗衣机正常工作，或者烹调冰冷的食物。

电器设计的另一个方面需要不同的姿态：显示洗衣机处于什么阶段或 VCR 设置成记录什么内容的状态信息应该提供为精灵图标，在边角处无声地提供最小状态。如果需要更多的状态信息，那么辅助姿态就是合适的。

9

和谐与流

要使软件更具有生产率，我们必须让它的用户更有生产率。要使用户更有生产率，我们必须确保用户应用软件时处于一种和谐的心境中。正是用户的精神状态，最终确定了他们使用软件的效率。本章讨论如何确保软件加强用户的效率和效益，如何避免打断我们希望用户保持的关注生产率状态。

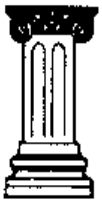
流与透明性

当人们全身心地关注于某活动时，他们不会意识到外界的干扰。这种状态称为流（flow），由 Mihaly Csikszetmihalyi 首创的概念。Mihaly Csikszetmihalyi 是芝加哥大学心理学教授及《流：最佳体验的心理学》一书的作者（HarperCollins , 1991）。

在《人件：高生产率的项目和团队》（Dorset House, 1987）一书中，Tom DeMarco 和 Timothy Lister，将流描述为“深入的，几近完全沉思的状态”。流经常产生“平静的欢娱”，能够让你忘记时间的流逝。尤其是当你从事面向过程的任务时，例如“工程、设计、开发和写作”。今天，这些任务通常在计算机上完成，在这期间，用户与软件交互。因此，我们应该创建促进和加强流的软件交互，而不是可能会打断或者干扰流状态的软件交互。

如果程序不断地打断用户的流状态，那么对于用户来说，回到原来的生产率状态非常困难。

如果不需要你的程序，用户就能够像玩魔法一样实现自己的目标，那么用户就不会使用你的程序[1]。出于同样的原因，如果用户需要你的程序，但可以在不使用用户界面的情况下实现自己的目标，那么他也不会使用界面。用户与软件交互不是一种审美体验（可能游戏、娱乐、面向研究和探索的交互系统除外）。在很大程度上，它是一种注重实效的体验，与软件的交互越少越好。



公理：无论你的界面有多酷，越少越好。

将注意力放在交互本身上只会强调工具的副作用，而不是强调用户的目标。用户界面是人工制品，与用户的目标没有直接关系。下次，你可能就会嘲笑你以前设计的自认为很酷的交互。切记，对于大多数用户的意图来说，最终的用户界面就是没有界面[2]。

要创建流，我们与软件的交互必须成为**透明**（transparent）的。换句话说，用户没有注意到界面是可视的人工制品，相反，界面每时每刻服务用户，在恰当的时候和恰当的地方向用户提供他恰当的服务。有很多使用户界面隐身的极佳方式，如下所示：

1. 遵循心智模型。
2. 引导，但不要讨论。
3. 将工具放在顺手的地方。
4. 提供无模态反馈。

我们现在依次详细地讨论每种方法。

遵循心智模型

我们在第 2 章引入了用户心智模型的概念。对一个处理来说，不同的用户有不同的心智模型，但他们很少能够详细地理解计算机的处理过程。每个用户自然都形成了有关软件如何完成任务的心智愿景。人们的大脑会寻找一些因果模式来理解机器的行为。

---

[1] **译者注** 作者富有创意地阐叙“界面存在的价值在于加强和促进用户的生产率”。

[2] **译者注** 参见第 10 章。作者的意思是界面多少会给用户增加负担。

例如，在一个医院信息系统[3]中，医生和护士都有病人信息的心智模型，这些信息从真实世界中处理的病人记录中获取。因此，用病人的名字作为索引来寻找病人信息是有意义的。每个医生都有相应的病人，所以，在治疗界面中过滤病人有着额外的意义，这样，每个医生都能够在以名字方式排序的名单中选择自己的病人。另外一方面，在医院的业务部门，办事员担心的是延期未付的账单。他们不关心谁支付账单，以及为了什么目的支付账单，而关心的是什么时候支付账单（并且这些账单有多大）。因此，对于业务部门的界面来说，首先按照延期时间或者应付金额大小排序，而将病人的名字作为辅助的组织方式更有意义。

引导，但不要讨论

很多开发人员认为理想的界面应该与用户进行双向交流。然而，大多数用户都不这样想。例如，他们更愿意用和自己的车交互的方式与软件交互。打开车门，上车，然后去目的地。要继续向前时踩油门，想停下来时踩刹车，转弯的时候打方向盘。

这种理想的交互情形不是对话，更像是使用工具。当木匠要敲钉子时，他不想和锤子讨论钉子的问题——他会直接用锤子钉钉子。在车里，如果司机想改变方向，他会直接指挥车子。司机喜欢通过合适的设备从车子和外部环境直接获得反馈：挡风玻璃外面的视野；仪表板的读数；疾驰而过的风声；轮胎压在道路上的声音；对侧向重力的感觉，以及路面传来的振动。木匠也希望有类似的反馈：钉子下沉的感觉，铁互相击打的声音，以及举起锤子的感觉。

司机当然不期望车子通过对话框与自己交互，木匠更不希望看锤子上显示这样的信息（如图 9-1）。

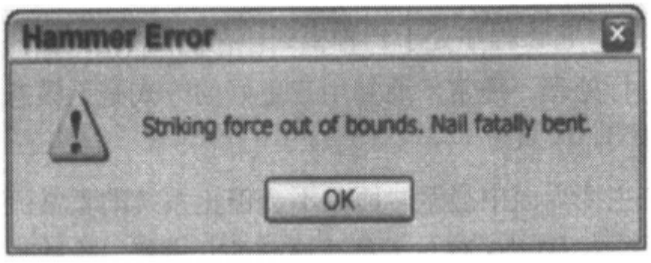


图 9-1　程序员习惯看到这样的消息，并不意味着其他行业的人们也希望这样。没有人希望他要使用的机器来责备它。如果以错误的方式启动机器，我们期望获得错误的响应。当然，这可以防止我们犯严重的错误，但责备不同于保护。

[3] 译者注　医院信息系统要服务于两个首要人物角色，所以包括治疗界面和业务办公界面。

软件常激怒用户的原因之一就是：它们并不像车子或者锤子那样工作。相反，软件经常不合时宜地把我们引到对话框，指出我们的缺点，并且要求我们做出响应。从用户的角度看，软件颠倒了自己的角色——应该是用户提要求，而软件响应。

使用直接操作能够获得我们所想要的效果。如果用户希望将对象从 A 处移动到 B 处，那么用户单击它并把它拖放到合适的位置。更能引导流状态的界面具有丰富而完善的直接操作习惯用法，这是一个通用的原则。

将工具放在顺手的地方

如果要直接操作所有的特性，大多数程序都太复杂了。因此，它们为用户提供了不同的工具集合。这些工具实际上是程序进入的不同行为模式，提供工具是一种复杂性的折中，但我们还是可以努力使工具操作更加容易，而且不干扰流。我们首先必须确保工具信息非常丰富，容易看到，并且简单地实现工具之间的切换快速。

工具应该就手，最好在工具栏和工具条上。这样，用户容易看到，而且单击就可以进行选择。如果用户必须转移注意力去寻找工具，那么他对应用的专注就会受到干扰。这就像他必须从座位上起身，沿着走廊寻找铅笔一样。他永远都不应该四处寻找工具。

无模态反馈

操作工具时，我们经常希望程序报告自身的状态，以及工具正在操作的数据状态。这些信息需要清晰地发布且容易发现，不会干扰或阻止当前的动作。

当程序需要向用户提供信息或者反馈时，可以有多种表达方式。最常见的方式是在屏幕上弹出一个对话框，这种技术就是模态。它让程序处于一种模式，在返回常态之前，也就是在用户能够继续任务之前，必须进行处理。通常，通知用户更好的方式是**无模态反馈**（modeless feedback）。

如果无论在什么时候，用户的信息在主要界面中显示，而且不会停止系统的正常流程和交互，那么这种反馈就是无模态的。在 Word 中，仅仅看屏幕底部的状态栏，你就可以无模态地发现你在哪一页，哪一节，当前文档有多少页，指针在什么位置，现在是什么时间。

如果想知道你的文档有多少个字，那么你将不得不从菜单中调用工具——字数统计对话框（见图 9-2）。对于写杂志文章的人来说，他需要认真考虑字数，该信息最好无模态发布。

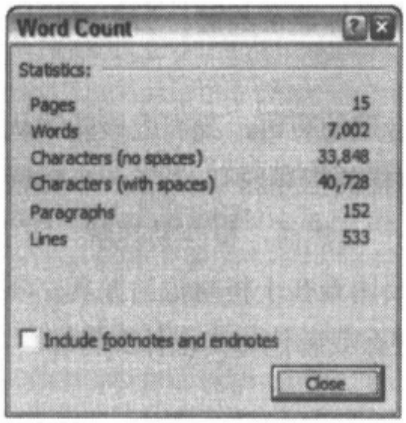


图 9-2 在 Word 中，如果你想知道文档有多少字，你必须选择“字数统计”子菜单。这会打开一个对话框。要返回 Word，你必须首先单击“字数统计”对话框的“关闭”按钮。这种行为恰恰与无模态对话框相反，并且干扰了流。

喷气式战斗机机有头部显示，即 HUD，它将关键的设备信息显示在驾驶舱挡风玻璃的前端，当驾驶员的眼睛盯着敌机时，甚至不需要周围的视觉，就能够看到重要的仪表信息。

我们的软件应该像 HUD 一样显示信息。程序应该能够使用显示屏幕的边缘显示用户在应用程序主要工作区活动的信息。很多绘画程序，如 Adobe Photoshop，已经提供了标尺、缩略图及其他窗口外围的无模态反馈。我们会在第 34 章进一步讨论丰富的无模态反馈类型。

和谐

对一个成功的作家来说，他的作品中见不到构思的痕迹，读者清晰地看到故事和人物，而不会受到作家所运用的写作技巧的干扰。同样，当用户与程序交互良好时，交互机制隐身，让用户直接面对自己的目标，意识不到软件的介入。糟糕的作家在其作品中随处可见，糟糕的设计师在他设计的软件中也随处可见。

公理：和谐的用户界面是透明的。

对于小说家来说，不存在什么“好的语句”，也不存在什么规则说“语句应该如何透明地组织”。一切依赖于故事的主角在做什么，或者作者想创造什么样的效果。小说家知道在某个平静而细致的段落里不能插入一个模糊的词语，就像在弦乐四重奏中不能有刺耳的音符一样。这种规律同样适用于软件。交互设计师必须训练他的耳朵，听出软件交

互弦乐队中发出的刺耳音符。界面中所有的元素一致地实现目标非常关键。当程序与用户的交流非常和谐时，软件交互变得几乎不可见。

Webster 将和谐定义为“协调一致的组织”，这是一个合理的短语，我们应该能够从与软件的交互中体会到。和谐没有固定规则。你不能创建这样的指导原则：“在一个对话框上五个按钮很合适”或者“一个对话框上七个按钮太多了”。然而，很容易理解，“在对话框上有 35 个按钮”这种情况应该避免。这种分析的主要困难在于将问题当做真空环境，没有考虑正在解决的问题，也没有考虑当时用户在干什么或者他想实现什么目标。

增加的技巧：更少，更好

对于很多事情来说，多比少好。然而在界面设计中，反过来才是对的。我们应该不断减少界面元素的数目，但不降低系统的能力。为了实现该目的，我们必须用更少的元素实现更多的目标。在这种情况下，和谐变得很重要。我们必须调整和控制产品的所有功能，而不是让界面成为窗口和对话框的堆积，不是让界面上散布着很少使用的，彼此之间没有联系的控件。

很容易创建复杂而效率低下的界面。这些界面通常允许用户完成单个任务，而不顾及其他相关的任务。例如，大多数桌面软件允许用户命名和保存数据文件，但不同时提供删除、重新命名或者拷贝文件的功能。对话框将这个任务交给操作系统。添加这些功能可能微不足道，但程序员完成这些微不足道的活动，要比用户被迫去完成这些活动要好。直到今日，如果用户想做一些简单的事情，如编辑文件的其他拷贝，必须完成一系列并不简单的动作：去桌面选择文件，双击拷贝文件，改变文件的名字，打开新的文件等。为什么不将这些交互以流水线的方式处理呢？

这并不像看起来得那么困难。和谐并不意味着借助出现的问题进行恐吓，而是对每一种可能性都灵巧地处理。与其在每一个应用程序的“File Open” 对话框中添加文件拷贝和重新命名功能，不如放弃“File Open”对话框，而用 Shell 程序取代。当用户想打开一个文件时，程序方便地调用 Shell 命令，该命令内建了所有的文件操作功能，用户可以双击期望的文档。诚然，使用打开文件对话框能够向用户显示过滤的文件视图（通常受限于能够识别的格式），但为什么不将“通过类型过滤加上类型排序”这些功能添加到 Shell 中呢？

按照逻辑，我们也能够处理“Save As…”对话框，它和“File Open”对话框的逻辑相反。如果我们每次在应用程序中调用 Save as 功能，它用合适的临时文件名将该文件写入临时的目录，并且将控制转交给 Shell，应该有这些功能的 Shell 工具，使我们能随意

移动文件或者对它们重新命名。

要让这些功能无缝连接，程序员不得不创建大量的代码，但考虑一下前面提到的情形。通过简单的设计努力，就可以完全放弃无数的对话框，而成千上万的程序的用户界面会在视觉和功能上更加一致。这就是小技巧。

区分可能性和概率

交互，通常以对话框的形式，不必要地出现在用户界面中，这种情况有许多例子。通常，出现这种情况的原因是程序面临选择。这是因为程序员倾向于从逻辑的角度解决选择问题，这种特征也体现在软件设计中。对于逻辑学家来说，如果一个命题 999 999 次为真，而一次为假，则这个命题是假的——这就是布尔逻辑工作的方式。然而对于其他人来说，这绝对是个真命题。命题存在为假的可能性，但为假的概率极小。更好地编制用户界面的最有效方式之一就是区分可能性和概率。

程序员倾向于把可能性和概率等同起来。如用户可以选择结束程序时保存或放弃他已经工作了六个小时的文档。从数学的角度看，这两种选择都有可能，但是反过来，用户放弃他们工作的概率可能小至千分之一。然而，大多数程序都有询问用户是否想保存工作的对话框，如图 9-3 所示。

图 9-3 中的对话框既不合适也不必要。你选择放弃文档修改的概率是多少？弹出这个对话框就像你的爱人每次吃饭的时候都提醒你不要把汤洒在身上一样。我们将在第 13 章讨论剔除这种对话框的意义。

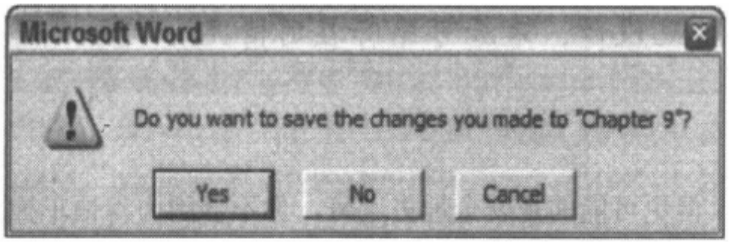


图 9-3 这是图形用户界面世界里最不必要的对话框。我们当然想保存自己的工作。大多数情形都是这样的，不保存违背了常规，应该以某种不常用的对话框处理。这个简单的对话框更多的是强迫用户了解 RAM 和磁盘存储的一些无用而令人困惑的细节，而不是他们与计算机的交互。这样的对话框永远不该出现。

另一个混淆概率和可能性的例子可以在 Microsoft Excel 4.0 版本（现在已经不用了）中找到。当你选择一个或者多个单元格并且按下 Delete 键清除字段时，会弹出一个小的

对话框询问你想删除什么。如图 9-4 所示，该对话框“方便地”允许你清除所选单元格的格式、公式或者注解。

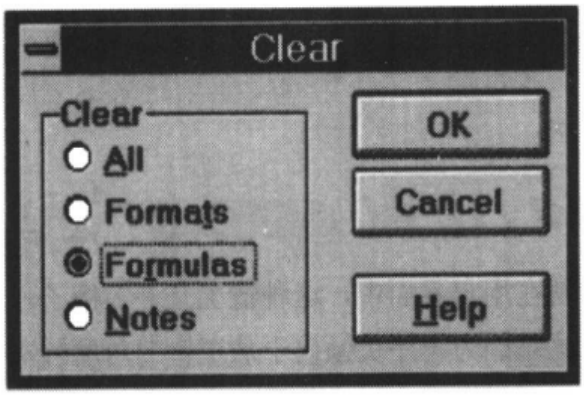


图 9-4　在 Excel 的 4.0 版本中，当你按下删除键时，就会弹出该对话框。如果你是计算机，这相当合理，但如果你是一个人，这意味着当你每次去完成的公式清除（这通常概率很大）时，你不得不处理可能性很小的情形。Excel 中的对话框，就像每次听交响乐休止符时，指挥不得不翻一页。谢天谢地，Microsoft 在新的版本中已经更正了。

该对话框的强迫性让每个用户都要发疯。确实存在三种不同的删除操作：格式、公式和注解。然而，虽然用户删除公式的频率很高，但很少删除格式和注解，这也是事实。某些东西是可能的，但不意味着它很可能发生。一个高级的删除功能，可以以下拉菜单或者弹出菜单的方式获得，能更简捷地解决这类问题。

人们通常以是否有能力创建处理复杂逻辑系统中出现的多种可能性的软件（但几乎不可能）来评价程序员。然而，这并不意味着他们应该直接在用户界面中处理这些不规则的可能性。对程序员设计的用户界面来说，不合时宜出现的一些可能存在的边缘情况是没有价值的友情赠送。一天使用几百次的对话框、控件和选项与一天仅使用一次或从不使用的对话框、控件和选项毫无区别地列在一起[4]。

公理：为很可能发生的情况设计，考虑可能存在的情况。

今天早晨，你可能会被车撞到，但更可能的是安全地去上班，你不会因为害怕车撞而待在家里，所以在界面中没必要让那些可能发生的事情，改变你处理几乎一定会发生的事情的方式。

[4] **译者注**　作者认为这种情况不合理。

提供比较

另外，程序表达信息的方式可能会干扰用户的意识。经常滥用的是定量数据或者数值信息的表达。如果某个应用程序需要显示磁盘空闲空间的大小，它通常会像 Microsoft Windows 3.x 文件管理器那样，向用户提供确切的空闲字节的大小，如图 9-5 所示。

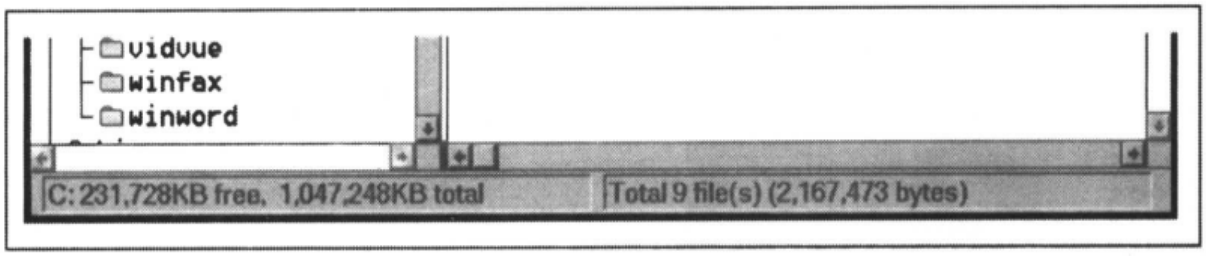


图 9-5 Windows 3.x 文件管理器在报告磁盘文件使用的字节数目方面花了不少精力。这种精确性是否有助于用户确定清理磁盘空间的必要性？当然不会，进一步说，7 位数字是显示磁盘状态的最好方式吗？用比例方式的图形表达（比如饼形图）显示磁盘空间使用情况不是更有意义吗？幸运的是，Microsoft Windows 当前使用饼形图来显示磁盘空间的使用情况。

程序在左下角告诉用户磁盘空间的大小，以及磁盘空闲字节的多少。这些数字难以阅读，也难以理解。在一个有着千百万字节的硬盘空间里，还剩下多少字节对于我们来说已经不重要，然而，当前的用户界面严格地告诉用户磁盘空间还剩下多少 KB。但即使程序精确地告诉我们当前磁盘的状态，它也无法成功地与用户交流。我们真正需要的信息是磁盘空间是否用完，或者我们在安装一个 20 MB 的程序后，是否仍然有充足的工作空间。尽管这些原始的数据很精确，但无助于我们理解现状。

视觉表达专家 Edward Tufte 说定量表达应该回答 “与什么比较？” 的问题。得知磁盘上存在 231 728 KB 空闲空间，得知磁盘全部空间的 22%为空闲空间相比，后者更有用。Tufte 的另外一个格言是“显示数据”，而不是简单地提供文本或者数字。在饼形图中以不同颜色显示硬盘使用空间和空闲空间，更容易理解硬盘空间的使用比例和规模。它会向我们显示 231 728 KB 实际上意味着什么。不应该放弃数字，但应该归入显示的标签状态，而不是显示本身。它们应该以合理而一致的精确方式显示。这些信息的含义应该能够形象化地显示，而数字只不过增加支持而已。

在 Windows XP 里，Microsoft 有进步，也有退步。文件管理器（如图 9-5 所示）已经没有了，取而代之的是如图 9-6 所示的浏览器对话框。它将属性对话框与硬盘联系起来。使用空间显示为蓝色，而剩余空间显示为紫红色。这使得饼形图易读。现在你扫一

眼就可以看到高兴的消息：GranFromage[5]几乎是空的。

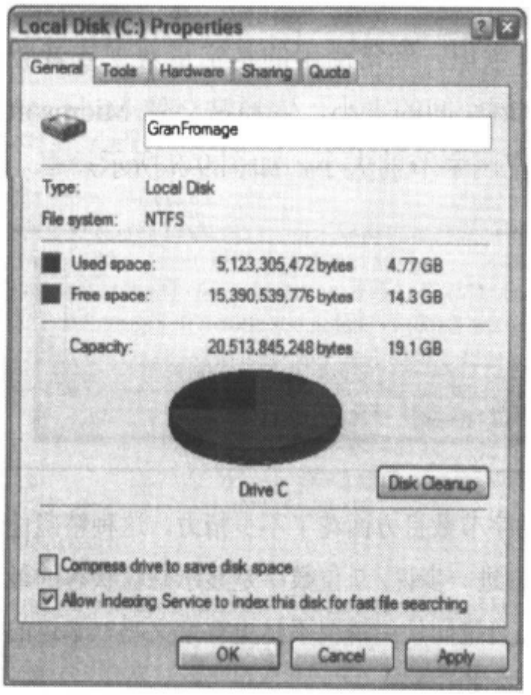


图 9-6　在 Windows XP 中，用“注射”取代了“电椅”[6]，你可以从浏览器调用属性对话框取代文件管理器底部长而难以理解的数字。好的一面是你可以看到以饼形图方式显示的硬盘使用情况，而坏的一面是你不得不停止当前的活动，打开对话框了解应该预先就能够获得的基本信息。在 Windows 2000 中，当选中一个硬盘分区时，图形自动显示在浏览器窗口的左边。XP 的解决方案倒退了一步。

遗憾的是，饼形图没有内建在浏览器界面中。相反，你不得不在菜单项里寻找它。要了解硬盘空间的使用情况，必须弹出一个模态对话框，虽然它会提供这些信息，但不顺手。浏览器是你浏览、拷贝、移动和删除文件的地方，但它并不是你很容易发现是否需要删除一些东西的地方。饼形图应该内建在浏览器里。在 Windows 2000 中，当你选择浏览器窗口时，它显示在左手侧。然而在 Windows XP 里，Microsoft 又退步了，图形再一次归入对话框。饼形图应该和数值数据一起在任何时候都能够在浏览器中见到，除非用户有意选择隐藏它。

---

[5] **译者注**　本地硬盘的名字。

[6] **译者注**　指 Windows XP 在进步的同时，又退步了，始终未能尽善尽美。这些都是死刑的方式！

使用图形化输入

软件经常没有以图形的方式表达数值信息，支持图形输入的情况就更少见了。很多软件让用户输入数字，接着使用命令行将这些数字转换为图形。很少有产品允许用户图形化输入，将图形转换为数字矢量。与此不同的是，大多数现代字处理软件允许你拖动标尺上的标记来设置制表和缩排。在效果上，用户可以说“这就是我希望段落开始的地方”，程序精确地计算从左边边栏 1.347 英尺开始，而不是强迫用户输入 1.347。

像 Microsoft Visio 这样的智能画图程序在这方面更为出色。用户在屏幕上操作的每个多边形在后台由小的单元电子表格表达，行代表每个节点，而列代表每个节点的 *X* 和 *Y* 坐标。拖动多边形的顶点使得电子表格的对应节点数据发生改变，这些值以 *X* 和 *Y* 表示。用户既可以通过图形也可以通过电子表格的方式来访问形状。

这个原理可用于多种情况。当列表中的项目需要重新排序时，用户可能想以字母排序或者个人偏好排序，可能没有相应的算法支持。用户应该能够直接拖动以达到期望的次序，而没有算法干扰这些基本的操作。

反映程序状态

当某人睡着时，他看起来睡着了。当某人忙碌时，他看起来很忙：眼神关注在工作上，身体语言处于封闭状态，全神贯注。当某人闲暇时，他看起来很闲，身体放松，活动着，眼神四顾并且看起来愿意和人接触。人们不仅仅希望彼此之间能够获得这种微妙的反馈，他们也依赖于这种反馈维持社会秩序。

程序也应该以同样的方式工作。当程序休眠时，它应该看起来休眠了。当程序被唤醒时，它看起来就是唤醒的。当程序繁忙时，它看起来很忙。当程序在从事某种耗时的内部活动如格式化磁盘时，我们应该看到一些显眼的外部动作，如磁盘的图标缓慢地从灰色状态变为活跃状态。当计算机在发送传真时，我们应该看到扫描传真和发送传真的标志（或者至少是一个无模态的进度条）。如果程序在等待远程的数据库反应时，它应该在视觉上进行改变，反映这种等待的状态。在本章的前面我们已经简略地提到程序状态最好以无模态反馈的形式提供。更多无模态反馈的例子可以在第 34 章找到。

避免不必要的程序状态报告

对于程序员来说，最重要的是确切地知道在程序进程内发生了什么事，这样能够控制整个进程。对于用户来说，没有必要知道发生的所有细节。如非技术人员听到数据库已经改动时会感到不安。让程序完成必要的状态报告就可以了，在情况正常时，给予提示，让用户放心，而不要告诉用户程序完成任务的细节，以免给用户带来负担。

即使用户无法理解它提供的信息，但很多程序仍然迅速地告诉用户程序的详细进度。程序弹出对话框告诉用户“连接已经建立”，“记录已经发出”，“用户已经登录”，“事务已经记录”这些无用的信息。对于软件工程师来说，这些信息相当于机器的轰鸣声、溪流的潺潺声、拍击在海滩上的波浪声。这些信息提示我们一切正常，实际上，它们也可能用于调试软件。而对于用户来说，这种状态报告就像地平线上出现的怪异光线、夜里的尖叫声和房屋内到处乱飞的东西。

正如前面讨论的，程序应该清晰地显示自身正在工作，但应该以更微妙的方式提供详细的反馈信息。以模态对话框的方式报告状态信息停止了交互，而没有带来其他任何好处。

报告程序的常规状态而不停止程序进行很重要。在报告状态时，不要采用对话框方式。将对话框用于那些超出正常进度的事件的报告。

设计技巧：不要使用对话框报告常规状态。

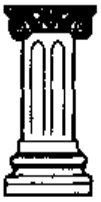
基于同样的原因，不要停止程序的进行而用那些不严重的状态报告干扰用户。如果程序比较繁忙，那么不要使用对话框报告状态。相反，在程序里创建状态指示器，这样的信息对于感兴趣的用户来说很清晰，而对于那些繁忙的用户也不鲁莽。

协调用户交互的关键是采取目标导向的方法。你必须问自己是否存在快速的交互，帮助用户直接实现目标。当前设计的程序在用户没有预先引导的情况下，通常不会采取任何向前的步骤。但用户更愿意看到程序采取“合理”的第一步，接着调整达到所期望的目标。如果采用这种方式，那么程序让用户更接近目标。

避免白板

对用户的需要一无所知，问用户一堆问题来确定他们的需求是很容易的。有多少程序在启动开始时问用户一堆问题？但是用户，不是超级用户而是普通人，对向程序解释自己的行为感到非常不舒服。他们更愿意看到程序考虑正确的假定情况，接着正确地操作。在大多数情况下，你的程序基于过去的经验做出正确的主观设想。例如，当你使用 Microsoft Word 创建一个新的文档时，程序预先为空白文档设置页边距和其他属性，而不是打开一个对话框来定制每一个细节。与之相比，PowerPoint 做得不够。在每次创建新的讲稿时，PowerPoint 让你选择讲稿的基本风格。程序可以通过记忆“用得最多的”或“最近使用的”风格或者模板，并且为新的文档创建默认的设置而做得更好。

仅仅使用“思考”这个词，并不意味着软件需要智能化（以人类的思维来说），需要通过推理来确定最合适的事情。相反，它应该做一些简单的统计意义上的正确的事情，接着提供有用的工具来让用户完成第一次尝试。而不是简单地给用户一个白板，让用户勉为其难。这样，程序不是在请求行动许可，而在事后请求宽恕。



公理：请求宽恕，而不是许可。

对于大多数人来说，完全的白板是一个困难的起点。在别人的基础上开始要容易得多。通常，用户很容易精细地调整程序提供的近似值，从而精确地得到用户所希望的，而没有从头开始那么大的风险。正如我们在第 15 章所讨论的，赋予程序好的记忆力是实现这些目标的最好方法。

命令调用与配置

无论在什么时候，当用户调用带有很多参数的命令时，经常产生另外一个问题。这个问题的产生是因为没有区分功能和它的配置。如果你要求程序完成一个功能，那么程序应该简单地完成这个功能，而不需要向用户询问精确的配置细节。

例如，对于很多程序来说，当你想要打印文档时，会调出一个配置对话框。复杂的对话框要求你指定需要打印多少份？纸张的方向是什么？使用什么送纸器？设置多大的

页面空白？输出应是单色，还是彩色？打印比例是多少？是否使用 Postscript 字符或者本地字符？是否打印当前页？打印的范围是当前选择还是整个文档？是否打印到一个文件？如何命名这个文件？所有这些选项都是有用的，但我们所想的是打印文档，而这些应该包含在我们的请求之中。

更为合理的设计应该是使用一个命令来打印，而另外一个命令用来设置打印。打印命令不能发起任何对话框，而只是直接打印。要么使用前面的设置或者标准的设置。打印设置功能会提供有关纸张、拷贝和字符的所有选择。能够直接从配置对话框完成打印也非常合理。

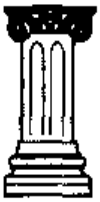
Word 工具条的打印控件提供了不需要调用对话框的直接打印。这对于很多用户来说非常完美，但对于那些有多个打印机或者有网络打印机的用户来说，它提供的信息太少。用户可能在单击控件或者调用对话框改变它之前，想知道程序选择的是哪一个打印机。放在工具条或者状态栏上的简单无模态输出是一个好的候选方案（当前提供在控件的工具提示中，这很好，但反馈比这更好）。Word 的打印设置对话框称为 Print...，可以从文件菜单中调用。它应该有更清晰的名字，虽然按照 GUI 的标准，省略号确实暗示调用对话框。

配置功能和调用功能之间存在很大差别。前者可能包括后者，但后者不应该包括前者。一般来说，任何用户配置命令一次，就会调用命令十次。在十次调用中，让用户显式地配置一次命令，要比用户十次有九次拒绝配置界面要好。

Microsoft 的打印方案是合理的经验方法。将对功能的即时访问放在工具条的按钮上，而对配置功能的对话框访问放在菜单项上。配置菜单是更好的教学工具，而按钮则提供了即时动作。

提问与提供选择

提问不同于提供选择。两者之间的差别就像逛商店和面试。通常认为提问的人与被提问的人相比更优越。权威者提问，而从属者做出响应。向用户提问让他们感觉低一等。



公理：提问不等同于提供选择。

对话框（特别是确认对话框）提问，工具条是提供选择。确认对话框停下来，要求

用户回答问题，直到得到答案才会消失。另一方面，工具条始终存在，安静而礼貌，像琳琅满目的商店一样提供商品，为你提供手指一点就舒适地选择的机会。

与大多数软件开发者所想的不同，问题和选择并不是一定会让用户感觉强大。更普遍的是，这会让用户感到有压力，并为此烦恼。你喜欢汤还是色拉？色拉。你喜欢甘蓝还是菠菜？菠菜。你喜欢法国、千岛群岛，还是意大利的？法国。你喜欢本地的还是普通的？够了！快给我汤！你喜欢杂烩还是鸡肉面汤？

用户不喜欢被问。不断的提问向用户暗示着程序是：

- 无知的。
- 健忘的。
- 功能不强的。
- 不主动的。
- 不能自理的。
- 烦躁的。
- 过分要求的。

这些都是人类不喜欢的性格。为什么我们期望软件这样呢？程序问的不是一些智力问题，也不是期望像朋友们在宴会上那样交流，相反，那是无知的行为，并且表现出不适当的权威。程序对我们的观点不感兴趣。它需要信息，通常是那些一开始并不需要向我们询问的问题（如何避免这些问题的讨论见第 15 章和第 34 章）。

比单个问题更糟的是反复提出那些不必要的问题。你想保存文件吗？你是否想现在保存文件？你真的想保存文件？用户感觉那些很少提问的软件更聪明，更有礼貌。因为如果用户无法知道如何回答，就会觉得自己很愚蠢。

在“The Media Equation”（Cambridge University Press, 1996）一书中，斯坦福大学社会学家得出了引人注目的结论：人类对待和响应计算机及其他交互产品的方式和人类相同。我们是否应该注意产品显现出的人格特性。胜任工作、乐于助人，或者抱怨，唠叨，强迫，或者找借口。我们会在第 14 章讨论如何让软件更有礼貌和体贴。

选择很重要。但基于表达的信息做出自由选择和被程序以模态的方式审问之间存在差异。用户更愿意像在街道上驾驶汽车一样引导他们自己的软件。汽车向用户提供了全面的选择，而不发起一次对话框。想像一下图 9-7 描述的情况。

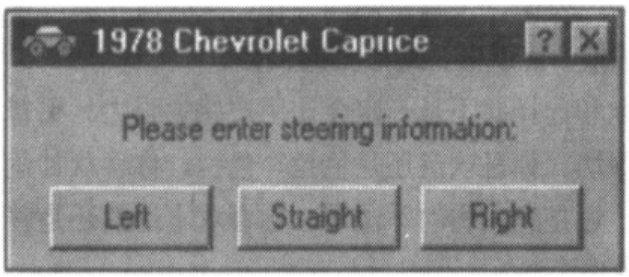


图 9-7 想像一下你不得不单击对话框上的按钮来驾驶汽车。这就会让你感受到普通人对对话框的感受：让人难过，不是吗？

直接操作方向盘对于汽车来说是最合适的习惯用法，它让用户感觉自己处于优越的位置，引导你的汽车去目的地。没有用户愿意像审判席上的嫌疑犯那样被人质疑，然而这就是我们的软件通常要求我们做的。

隐藏弹射座椅控制杆

在每个战斗机的驾驶员座舱下都有一个智能的控制杆，当拉动它时，会启动驾驶员座位底下的小火箭引擎，抛出座位上的驾驶员，他将借助降落伞安全地降落到地面。弹射座椅控制杆只能使用一次，并且它的结果明显，不可逆转。

就像战斗机需要弹射座椅控制杆一样，复杂的桌面应用软件也需要配置设施。变化的业务需求迫使软件适应特殊情况，并且最好能这么做。如果程序不能适应某些公司做事情的方式，可以为定制软件或者成千上万份产品许可付出上百万美元的公司不会仁慈地接受这种无能。程序必须适应，但这种适应要考虑为一次性的过程，或者仅在非常场合由公司 IT 职员完成。换句话说，弹射座椅控制杆可以使用，但不会经常使用。

公理：隐藏弹射座椅控制杆。

程序必须有弹射座椅控制杆，这样在某些情况下，用户能够改变界面上的持久对象（见第 11 章）或者显著地（有时候不可逆转）改变应用的功能或者行为。必须确保不会发生的事情是无意中触动弹射座椅控制杆（见图 9-8）。当用户仅想对程序做微小调整时，界面设计必须确保用户不会触动弹射座椅。

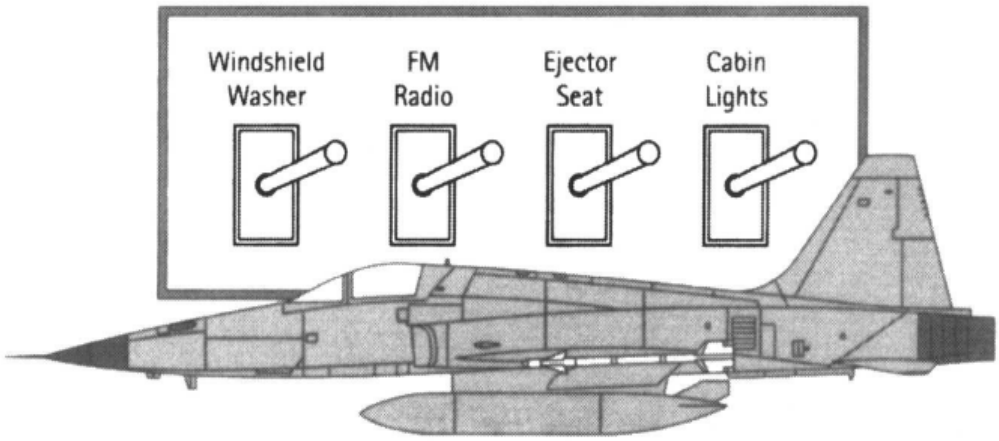


图 9-8 弹射座椅控制杆有着灾难性的后果。前一分钟，驾驶员安全地坐在飞机里，后一分钟，他离开自己的飞机，在蓝色的天空中翻跟斗。对于保证驾驶员的安全来说，弹射座椅是必要的。但需要花大量的设计工作确保弹射座椅不会无意触发。允许信任的用户改变永久的对象来配置程序和在偶然的情况下触发弹射座椅具有一定的可比性。请务必隐藏弹射座椅控制杆。

弹射座椅控制杆有两种基本的变体：一种导致程序明显的视觉混乱（工具和工作区域的布局发生大的改变）；另外一种产生不可逆转的行为。这两种功能都应该对不熟悉的用户隐藏。在这两者之中，后者更危险。对于前者，用户可能会感觉惊讶，对接着发生的事情感觉沮丧。但花一段时候后，用户能够返回。而对于后者，用户和他的同事们可能无法摆脱后果。

牢记流与和谐的主要原则，你的软件会让用户长时间处于高生产率状态。高生产率的用户是愉快的用户。拥有高生产率和愉快的顾客是任何数字产品制造商的目标。下一章，我们会进一步讨论如何消除不必要的障碍改善用户的生产率，而障碍通常是以实现模型思考的结果。

10

消除附加工作

软件经常包括一些交互，对于用户来说，属于头重脚轻的附加工作。程序员通常有意地关注使能技术，以至于他们不从目标导向的角度认真地考虑操作技术的人类的行为。结果是软件需要用户支付太多的体力做**附加工作**（excise），这通常是能感觉到的，甚至是每次使用时的身体劳作。

本章关注这种附加工作的性质，并且讨论可以减少甚至完全消除这些附加工作的手段。

什么是附加工作

当我们决定开车前往办公室时，必须打开车库门，上车，启动发动机，停下来，在动身前往目的地之前关闭车库门。所有这些动作都是为了支持汽车，而不是为了支持我们前往目的地。如果有星际迷航的传输器，我们呼叫相应的目的地的坐标，然后立即出现在那里。不需要车门，不需要发动机，也没有红绿灯。我们的意思不是抱怨开车复杂，而是要区分完成日常任务时需要采取的两种动作类型。

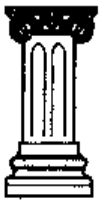
任何大的任务，诸如开车前往办公室，涉及很多较小的任务。有些任务直接实现目标，像操纵方向盘直到你的办公室。另外一方面，附加工作任务（excise tasks）不直接实

现目标，但与此同时，对于实现任务非常必要。这些任务包括打开和关闭车门，开动引擎，在红绿灯面前停车，给车加油及进行周期性的养护。

附加工作是在我们努力实现目标的同时，满足工具或者外部代理需要的其他工作。这两者有时候很难区分，因为我们习惯于附加工作成为任务的一部分。对于经常开车的人来说，区分打开车库门的活动和开车前往目的地的活动非常困难。开关车库门通常是我们为车子做的，而不是我们自己。它不像油门踏板和方向盘那样将我们带向目的地。在红灯面前停下是社会要求我们这么做的，也不能帮助我们实现自己真实的目标（在这种情况下，它确实帮助我们实现安全到达办公室的相关目标）。

软件在目标导向的任务和附加工作任务之间存在相当清晰的分界线。像汽车一样，一些软件的附加任务是微乎其微的，操作它们不需要多大困难。另外一方面，一些软件的附加任务像补车胎一样令人讨厌。安装就是这样，还有配置网络、备份，连接在线服务等都是附加任务。

附加任务的问题是它们在消耗我们的精力，而不是直接实现我们的目标。在消除了附加任务的地方，我们能够让用户更加有效率，更有生产率，并且能够改善软件的可用性。作为一个软件设计师，你应该对附加工作的存在非常敏感，并且象治疗感染的医生一样有激情地采取步骤根除附加工作。



公理：消除附加工作使得用户更加有效率。

有很多小的附加工作例子，尤其在GUI领域。几乎所有窗口管理都归入这类。拖，调整形状，调整大小，重新排序，窗口的层叠和级联等都属于附加工作。

GUI附加工作

对于有经验的计算机用户来说（尤其是那些受过命令行系统培训的用户来说），针对图形用户界面的主要批评之一是因为操作窗口和图标付出的额外努力使得实现目标更慢，更加困难。用户抱怨说，使用命令行时，他们仅仅输入期望的命令，计算机立刻执行。对于视窗系统，为了查找期望的文件或者程序，在启动之前，必须打开多个文件夹。接着它们出现在屏幕上之后，必须拉伸和拖动窗口，直到它们达到期望的位置和外观。

这种抱怨有着深刻的基础。这些额外的窗口管理任务确实是附加工作。它们不能帮

助我们实现目标。在他们设计成对用户的协助之前，程序的要求是一种额外开支。但每个人都知道，GUI 比命令行系统更加容易使用。谁是正确的呢？

因为实际的问题隐藏在后面，所以产生了困惑。命令行界面向用户强加了更为昂贵的附加工作。用户必须首先记住命令行。另外，用户也不能轻松地配置自己的屏幕，以适应个人需求。命令行界面的附加工作仅仅当用户投资了大量时间和精力学习之后，才会变小。

另外一方面，对于临时用户和新手用户来说，GUI 的视觉外在性帮助他们导航和学习什么任务是合适的，以及这些任务什么时候是合适的。GUI 一步一步的性质对于不熟悉系统任务的用户来说，是一个很大的帮助。对于那些有不止一个任务需要完成，并且必须一次使用多个程序的用户来说，非常有益处。

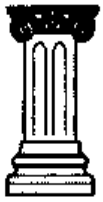
附加工作和专家用户

对于愿意学习命令行界面的任何用户来说，自动合格地成为超级用户。对于使用命令行的超级用户来说，会很快成为任何界面的超级用户，包括 GUI。这些用户会很快了解他们所使用程序的每一细微差别。在启动程序时，他们非常清楚想做什么，以及应该如何完成。对于这类用户，提供给临时用户和新手用户的帮助是一种障碍。

在我们消除附加工作时，必须小心。我们不能仅仅为了适应专家用户来消除它。然而同样，我们也不必强迫超级用户接受我们提供给新手用户或者临时用户的帮助。

训练工具

软件设计师经常无意中引入的大量附加工作领域之一是为新手用户或者临时用户提供支持。为了使新的用户更加容易学习而给程序增加功能设施非常合理。遗憾的是，这些设施在用户熟悉程序之后很快成为附加工作——正如第 3 章讨论的永久中间用户那样。为了培训新手用户而添加的软件功能必须很容易关闭。人们很少长期需要培训工具。培训工具虽然对于新手用户来说是一种福利，当这些工具永久留存下来时，是高级学习和使用的妨碍。



公理：不要固定培训工具。

“纯粹”的附加工作

有一些行为是没有人需要的纯粹附加工作，无论对专家用户，还是新手用户。这些包括计算机能够处理的硬件管理任务，比如告诉程序哪个 COM 端口能够使用。对这些信息的需要应该与用户界面隔离，并且用后台更为智能的程序行为取代。

视觉附加工作

有时候设计师过分依赖视觉隐喻，因此添加了各种附加工作。例如带有电话的桌面、复印机、订书机及传真机……或者抽屉内带有文件夹的文件柜——正是相关的例子。这些视觉隐喻能让人们容易理解程序元素和行为之间的关系。但了解这些基本情况之后，隐喻的管理成了完全的附加工作（有关视觉隐喻限制的讨论见第 19 章）。除此之外，图形消耗了非常多的屏幕空间，尤其是独占姿态的应用软件。我们盯屏幕的时间越长，我们越埋怨程序中已经熟悉的内容所占用的屏幕空间。在很久之前的第一天，那么迷人地告诉我们如何拨电话号码的小电话图标现在成为快速通信的障碍。

与独占姿态程序相比，暂时姿态程序能够容忍更多的培训和解释附加工作。暂时姿态程序不常使用，所以需要一些协助，帮助用户理解程序的功能，以及记住如何控制程序。然而，对于独占姿态应用，随着时间的流逝，最轻微的附加工作也变得苦不堪言。

在 Web 产生之前，第二种视觉附加工作并不是一个很严重的问题。第二种视觉附加工作即过分强调视觉设计元素，以至于妨碍了用户目标和对用户的理解。20 世纪 90 年代后期，Web 领域吸引了大量的图形和新媒体设计师。这些人将这种媒体视为主导的视觉形式，用户对它们的体验通过丰富的——通常是动画的视觉——来定义。虽然这可能（仍然）适合于目录销售网站，它们主要用于间接的销售。对于 Web 事务站点和 Web 应用来说，它完全不合适。正如我们在第 37 章讨论的，从行为的角度看，与多媒体公共信息亭软件或者目录网站相比，后者更像使用独占式桌面的应用，大多数电子商务站点归入此类。

结果是很多在视觉上具有吸引力和创新的站点忽略掉了最为关键的两个元素：理解

用户目标和帮助用户实现目标的流水线化行为。一个好的例子是 Boo.com，最先倒闭的 dot.com 例子之一。这个时装 E-tailor 使用时髦的视觉和基于 flash 的交互代理，但并没有花太多精力解决用户目标。这个站点因为使用 flash 而速度缓慢，视觉上分心，布局混乱，因为多个窗口和困惑的链接难以导航。Boo 高度概念化，但用户的目标非常简单，就是比别的地方更快、更便宜和更容易在线购买。过了一段时间，这些问题得到了弥补，但 Boo 的顾客已经放弃了。人们很想知道，如果采用目标导向的设计，Boo 及很多其他失败的电子商务站点会在多大程度上有所不同。

不幸的是，这种视觉过剩慢慢渗透到桌面应用，因为程序员和设计师从 Web 那儿借用了一些浮华而不合适的习惯用法。我们将在第 19 章更多地讨论合适的视觉界面设计。

确定什么是附加工作

有时候，我们发现，特定的任务例如窗口管理，虽然它们主要用于程序，但对于临时用户或者有特殊偏好的用户来说是有用的。在这种情况下，如果把它强加给用户，而不是按照用户意愿获得，那么这种功能本身只会成为附加工作。

确定功能或者行为是否属于附加工作的惟一方法是，将它与用户的目标进行比较。如果用户需要在屏幕上一次看到两个程序，目的是比较和传送信息，那么能够配置主窗口，使他们共享屏幕空间的能力就不是附加工作。如果用户没有具体的目标，那么用户必须配置主窗口的需求就是附加工作。

停止进度

有一种特殊形式的附加工作，它们是那么的普遍而备受关注。在第 9 章，我们引入了流的概念，在这种情况下，通过和谐地使用工具，用户进入高生产率的精神状态。流是一种自然状态，用户进入状态而不需要刺激。在某人已经进入流状态之后，需要一些努力才能打断。例如电话突然响了，这样会打断流状态，错误消息对话框的出现同样如此。一些打断是不可避免的，而另外一些则不是必要的。但是，轻易打断用户的流而没有任何好处是极端愚蠢地中止用户的进度，而且是最具破坏性的附加工作形式。

公理：不要极端愚蠢地停止进度。

设计糟糕的软件会做出凡是有自尊的人都不会做出的断言。例如，它明确地宣称文件不存在，仅仅因为它太愚蠢，以至于不能在合适的地方找到文件，然后它含蓄地责备你丢失了文件。程序高高兴兴地执行不可能的查询而导致系统挂起，直到你决定重新启动。用户实在有理由将这种行为视为白痴。

错误、通知和确认信息

可能不会有比错误消息和确认消息对话框这样更流行的附加工作元素。它们无处不在，根除它们需要大量的工作。在第 7 部分，我们会详细讨论这些问题，但现在仅仅需要指出“它们是严重的附加工作，并且无论在什么样的情况下都应该消除”就足够了。

通常的错误消息对话框是不必要的。它或者告诉用户不感兴趣的东西，或者要用户改变一些条件，而这些它本身就能够做到，也应该做到。图 10-1 显示了 Adobe Illustrator 6 显示的错误消息对话框，而此时用户想保存文档。我们不能肯定它想告诉我们什么，但它看起来很可怕。

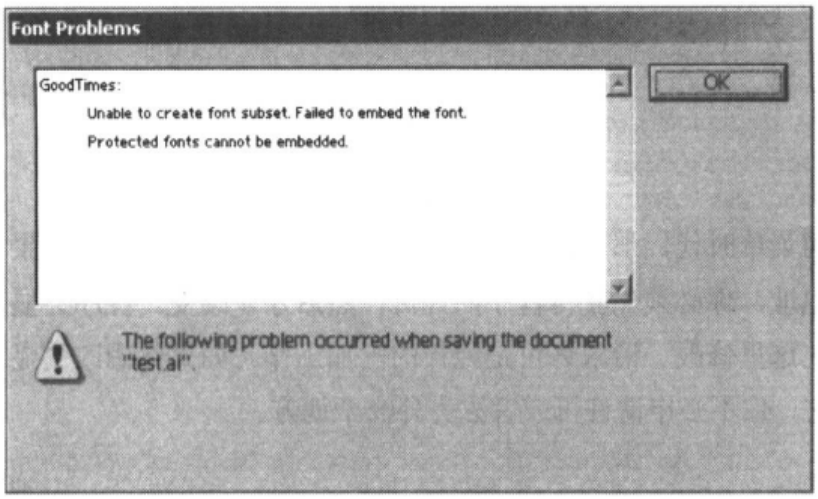


图 10-1 这是一个糟糕的无用错误消息对话框，它极端愚蠢地停止进度。我们不能验证，也不能理解它告诉我们的信息。它也没有给我们其他响应的选项，除了单击“OK”按钮承认我们的过失。仅仅当保存程序的时候，这个消息才出现。这也就是说，当我们委托它做一些简单而直观的事情，程序在没有帮助的情况下甚至无法保存一个文件。它甚至不能告诉我们它需要什么样的帮助。

消息中止已经令人厌烦而耗时良多的进程，会使这个进程显得更长。在告诉程序保存作品之后，用户甚至不能去拿一杯咖啡。因为也许他返回时发现功能没有完成，程序又无意识地停下来等待了。我们会在第 33 章讨论如何消除这类错误消息。

另外一个令人沮丧的例子来自 Microsoft Outlook，如图 10-2 显示。这个对话框要求我们在没有任何信息的基础上，做出可能会花很长时间而不能撤销的决定。如果对话框在你改变一些规则后出现，这是否成为你保存它们的理由呢？如果不这么做，你是否会想要多一点信息？比如说，什么确切的规则存在冲突，并且哪些是最近创建的？当我们按下撤销键时，我们也无法确认什么事将发生。我们是撤销对话框保留不匹配的规则？还是放弃导致不匹配产生的最新改变？这种设计糟糕的交互在用户心中产生的恐惧和不确定性，是完全没有必要的。我们将在第 34 章讨论如何改善这种情形。

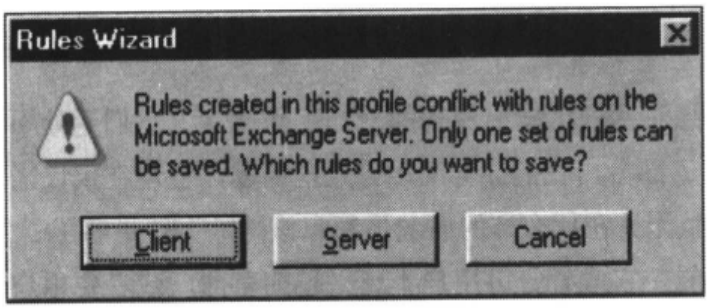
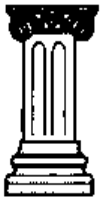


图 10-2　这是一个可怕的确认对话框，愚蠢地了中止程序的进行。如果程序足够聪明，能够监测到差异，为什么它自身不能改正错误呢？对话框给我们的选项是令人惊慌的，它告诉我们可以引爆两个盒子中的一个，一个装着垃圾，而另一个装着家里的狗。但程序没有说是哪一个盒子。如果我们单击撤销，这又意味着什么呢？它是否会继续，然后炸死我们的狗？

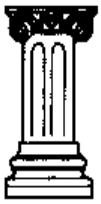
让用户申请许可

回到命令行和基于字符的菜单时代，那时，界面通常间接地给用户提供服务。如果想改变一个项目，例如你的地址，你必须先获得程序的许可，然后才能改变。程序会显示一个屏幕，这里地址可以在这里修改。请求许可是纯粹的附加工作。如果你想改变显示的值，应该能够立刻改变它。你不必申请许可或者去另外一个地方。



公理：别让用户申请许可。

很多文件选择对话框也这样做。它们向你显示建议的目录。但如果你不喜欢显示的选择，你不能实时改变它。相反，你单击一个按钮，比如说浏览，接着获取另外一个对话框，在这个对话框里，你改变它。这是不必要的，对话框应该很容易跟踪这些改变，验证它的合法性，并且确保正确地安装，而不必强迫用户申请许可。



公理：在有输出的地方允许输入。

正如在最后一个例子中，很多程序都有一个显示输出值的地方（诸如文件名、数值及选择的项），而在另外一个地方接受用户的输入。这遵循了实现模型，他们将输入和输出处理为不同的过程。然而，用户的心智模型并不区分这种不同。他认为“这里有一个数字，我只要单击它，就可以输入新的数值。”如果程序不能接受这种想法，也没必要将这种附加工作加入到界面。如果用户可以修改，那么他应该能够在程序显示的地方完成。

在某些环境下，请求许可的反面是有用的。与其让程序发起对话框，不如用户停止对话框，并且让对话框不再出现。用这种方式，即使程序错误地认为自己对用户有帮助，用户也能够让无用的对话框停止骚扰他。Microsoft 现在大量使用这种习惯用法（例如，如果新手无意地关闭了对话框而不知道如何恢复，那么他可以在一个显眼的地方得到帮助：即标有 “恢复所有关闭的对话框”的帮助菜单项。

不必要的保护措施

另外一个经常愚蠢地停止进度的地方是密码保护系统。在专门的环境中，安全是一个重要的问题，但这对于祖母和她用于冲浪的 PC 来说，没有意义。密码并不是对所有人和场景都合适，当他们起着妨碍作用时，应该有方法关掉。

不响应

程序访问如服务器这样的远程设备、打印机、网络、MODEM，或者它在进行大量数据结构的本地处理时，程序会变慢或者没有响应。每个程序执行可能大量消耗时间的任务时，必须保证偶尔检查一下，看用户是否在不停地敲击键盘或者疯狂地单击鼠标，抱怨着："不，不，我不是要重新组织整个数据库，那样要花四百三十万年。"

常见的附加工作陷阱

你应该警惕地发现和查找界面中的任何一个小附加工作。对于用户来说，无数小的附加工作加起来会构成很大的附加工作。下面的列表帮助你发现附加工作。

- 不要强迫用户到另外一个窗口去完成影响本窗口的功能。
- 不要强迫用户记住他将事物放在层次文件系统的哪个位置。
- 不要强迫用户调整窗口大小。当子窗口在屏幕上弹出时，程序应该为它的内容调整合适的大小。不要让它大而空，或者太小而需要不停地滚动。
- 不要强迫用户移动窗口。如果桌面上存在空闲空间，将窗口放在那，而不是直接将其放在已经打开的程序之上。
- 不要强迫用户重新进入个人设置，如果已经设置了字符、颜色、缩进，或者声音，确信她不需要再做一遍，除非她想改变。
- 不要强迫用户在填充字段时满足完整性。如果用户想从事务登录屏幕中忽略一些细节，不要强迫用户输入。假定他不需要再重新输入。数据库的完整性（在大多数情况下）不值得强迫用户。
- 不要强迫用户申请许可。这通常是不允许用户在输出的地方输入的症状。
- 不要让用户确认他的行为（这隐含存在健壮的撤销功能）。
- 不要让用户的行为产生错误。

用户界面中存在附加工作问题，还有导航问题，是用户对基于软件的产品不满意的主要原因。每个设计师和产品管理者应警惕各种形式的 GUI 附加工作，并且花时间和精力来确保将附加工作从产品中清除出去。

11

导航和调整

桌面应用软件、网站及各种设备都有着一个共同点，如果设计不当，都会给可用性造成很大困难，即一般是由于导航设计不当。用户必须能够借助程序、网站或者设备的特性和功能进行有效的导航。当用户在屏幕之间跳转时，必须能在程序中准确定位。

如果用户始终能理解他下一步要做什么，知道程序、网站或者设备的状态，并且能够发现他所需要的工具，那么用户就能导航。本章讨论围绕导航产生的问题，以及如何更好地通过交互产品帮助用户导航。

导航是附加工作

正如第 10 章所暗示的，关于导航，最重要的是要知道，几乎在所有的情况下，它代表着纯粹的附加工作，或者至少很接近附加工作。除了在游戏里，用户目标就是成功地穿越障碍或迷宫来成功导航之外，在软件里，导航不能满足用户的目标、需要或者期望。不必要的或困难的导航会成为用户沮丧的主要原因。实际上，在作者看来，糟糕的导航是任何应用软件或者系统——桌面的、基于 Web 的或者其他系统设计中的头号问题。这

也是程序员的实现模型对用户最透明的地方。作者要强调的是，应用程序或者网站并不能因为对导航结构进行额外注意而获得更多好处。

导航类型

软件中的导航发生在多个层次。下面列举了最常见的导航类型：

- 在多个窗口或者屏幕之间导航。
- 在一个窗口的多个窗格之间导航（或者页面的多个框架之间导航）。
- 在窗格的多个工具或多个菜单之间导航。
- 在窗格或框架显示的信息之间导航（如滚屏、平移、缩放、链接跳转）。

一些读者可能会质疑上面的一些导航类型，不过作者有意使用了广义的导航概念：任何将用户带到新的界面部分，定位对象、工具或者数据的动作都是导航。这么做的原因很简单。当我们开始将这些动作看做导航时，就可以清楚地看到它们是一些附加工作，因此应该尽量减少或消除，我们会更详细地讨论每一种导航类型。

在多个窗口或者页面之间导航

在多个窗口之间的导航对用户来说或许是最容易迷失方向的导航。在窗口之间导航涉及到注意力的整体转移，这会打断用户的流，强迫他们进入新的上下文。导航到另外一个窗口也意味着初始窗口内容的部分或者整个都脱离了视线的焦点。至少这意味着用户需要管理窗口，这是进一步打断用户流的附加任务。如果为了实现目标，用户需要不断地在窗口之间移动，那么生产率会下降，混乱感和挫折感会加深。如果窗口的数目很多，那么用户会完全失去方向感，可能会体验到**导航创伤**（navigational trauma）——他在界面中迷失了。独占式应用程序通过将所有的主要交互放在单个主窗口之中来避免这个问题，这个窗口可能包含多个独立的窗格。

在窗格之间导航

窗口可以包括多个窗格，它们或者相邻，或者通过使用分割器(splitter)（参考第 25 和 26 章）隔离，或者互相重叠，通过标签来识别。相邻的窗格可以通过放置有用的支持功能、链接或者将数据直接与主工作区域或显示区域相邻来解决导航问题，因此几乎可以将导航消减为零。如果对象能够在多个窗格之间导航，那么这些窗格应该互相靠近。

当相邻的支持窗格数目太多，或者它们不是以匹配用户工作流的方式放在屏幕上时，

会出现一些问题。太多相连的窗格会产生视觉混乱和困惑。用户不知道去哪儿查找所需要的内容。同时，过于拥挤也需要引入滚屏，这是另外一个导航问题。单个屏幕内的导航因此成了问题。一些门户网站，想要满足所有人的需要，因此存在这样的导航问题，我们将在第 37 章详细讨论。

在某些情况下，根据我们的工作流需要，可以使用标签窗格。标签窗格带来了导航负担和潜在的方向混乱，因为用户在导航时，它们盖住了屏幕上已有的内容。然而，当需要多个文档或者一个文档的多个视图时，对主工作区域可以采用这种习惯用法（例如在 Microsoft Excel 里，见图 11-1）。

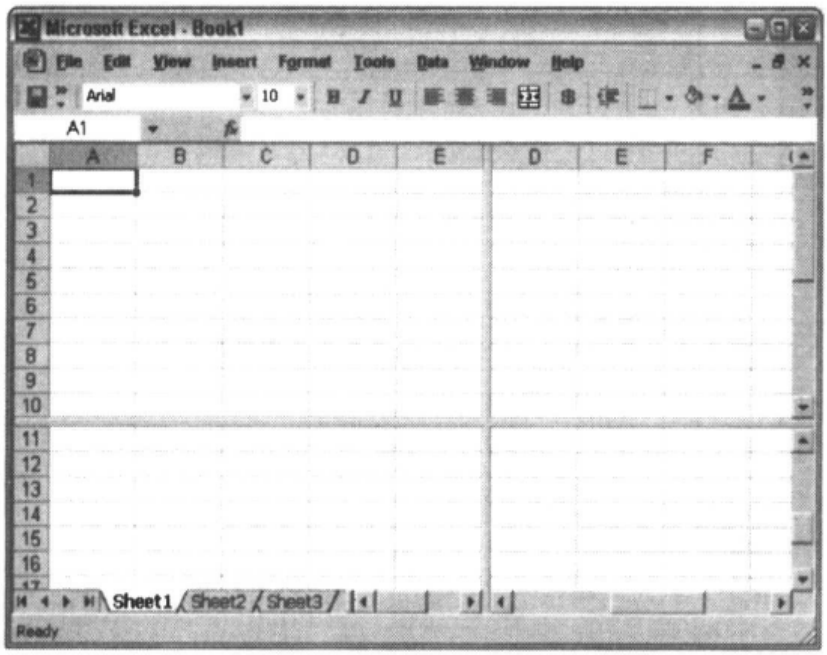


图 11-1 Microsoft Excel 使用标签窗格（在左下方可见），让用户在相应的工作表之间导航。Excel 也用分割器提供相邻的窗格来浏览单个工作表多个分开的部分，而不需要不断滚屏。这两种习惯用法都为 Excel 用户减少了导航附加工作。

一些程序员认为，标签可以将复杂的功能分为每个窗格一组简单的功能。他们的理由是，如果将功能分开，使用起来会更容易。实际上，将每组功能分为单独的窗格，附加工作增加了，而与此相反，用户的理解和方向感减少了。更严重的是，这么做违反了公理：对话框（或者弹出窗口）是另一个空间，去那里需要一个好的理由（第 25 章）。在大多数程序的大多数用户正常使用时，并不需要他们的软件工具存在几十个控件。

当一个工作区域存在多个支持窗格而不同时使用时，标签窗格是合适的。支持窗格也可以堆叠，用户选择适合他当前任务的窗格只需要一次单击。为 Macintosh 开发的 Microsoft Internet Explore 使用了这种堆叠窗格的变体。如果选择没有窗格（用户可以通过单击活跃的标签去掉选择），程序像关掉抽屉一样关掉相邻窗格，仅仅让标签可见（见图 11-2）。如果空间很宝贵，就可以使用这种变体。

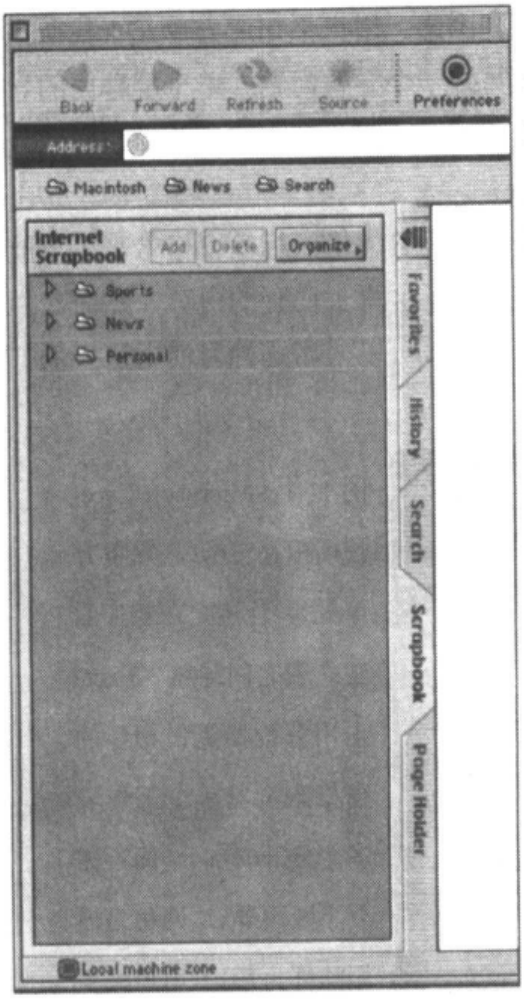


图 11-2 微软为 Mac 设计的浏览器通过垂直的标签条访问标签窗格。PC 版本有着相似的习惯用法，但使用更标准的工具条控件来选择窗格。Mac 版本更多地将这些功能与控件捆绑在一起，但要求用户歪着脑袋来读这些标签(tab label).

在工具和菜单之间导航

另外一种重要但常被忽视的导航形式来自用户需要使用不同的工具、调色板和功能这一需求。窗格或者窗口内的空间组织对于减少额外的鼠标移动非常重要，如果好一点，这些额外的鼠标移动会让用户恼怒和疲劳，如果再糟糕一点，就会产生反复性的劳损。那些经常使用的工具，以及一起使用的工具，应该在空间上组织在一起，并且可以立即获取。对于部分用户来说，菜单需要更多的导航努力，因为它们的内容在单击之前是不可见的。经常使用的功能应该以工具条、调色板，或者等效的方式来提供。菜单应该为那些很少访问的命令保留（我们会在本章的后面讨论控件的组织，在第 29 章详细讨论工具条）。

Adobe Photoshop 6.0 因为强迫用户在多个调色板控件之间导航，而让用户恼怒。例如，Paint Bucket 工具和 Gradient 工具都占用了工具调色板上相同的空间。你必须单击和

保持在一个可见的控件上，它打开一个菜单，让你在两者之间做出选择（如图 11-3 所示）。然而它们都是填充工具，而且需要频繁使用，最好将它们放在调色板邻近的地方，免得不断地破坏流的工具导航。

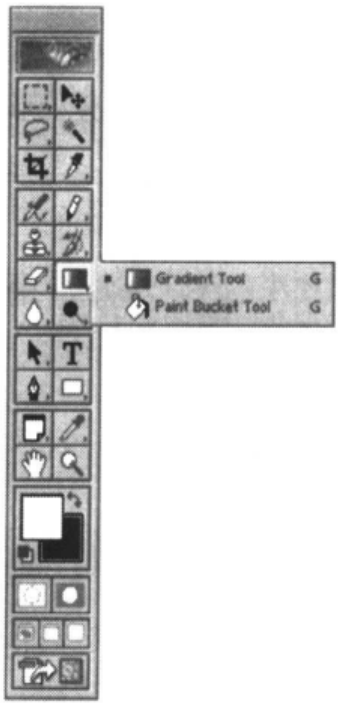


图 11-3 Adobe 将 Paint Bucket 工具隐藏在调色板的组合图标按钮（见第 26 章）中。即使用户经常访问 Gradient tool 和 Paint Bucket，用户任何时候访问这个菜单也都被迫在工具之间切换。

信息导航

信息导航或者窗格和窗口内容的导航能够通过很多方式来实现：滚屏（平移）、链接（跳转）和缩放。前面两种方法很普遍：滚屏在大多数软件中无处不在，而链接在 Web 中无处不在（虽然，链接习惯用法也慢慢在非 Web 应用中用到了）。缩放主要用于可视化的三维数据和详细的二维数据。

滚屏（scrolling）通常是必要的，但应该尽可能地减少这种需要。在换页和滚屏信息之间存在一个平衡：你应该理解用户的心智模型和工作流，来确定对于他们而言什么是最好的。

在二维的可视化及绘图应用中，垂直和水平滚屏非常普遍。这种类型的界面受益于缩略图而易于导航。我们将在本章的后面讨论这种技术及其他视觉标志。

链接（linking）是 Web 上关键的导航范式，因为它是在视觉上引起混乱的活动，所以必须特别小心，要提供视觉和文本上的线索帮助用户定位。我们将在第 37 章详细讨论 Web 导航。

缩放（zooming）和**平移**（panning）是浏览二维和三维信息的导航工具。这些方法在创建二维和三维图画、模型，或者探索真实世界三维环境（如结构走查）的表达方面非常合适。当用于检查超出二维表达的任意或者抽象数据方面时，就存在不足。一些信息视觉工具使用缩放表示“显示对象更多的属性细节”，这是一种逻辑上的缩放，而不是空间的缩放。随着对象视图的增大，属性（经常是文本）出现在它的图形表达上。要更好地对这类交互进行服务，可以使用相邻的支持窗格，相邻窗格能以更标准和更可读的形式显示选择对象的属性。用户发现空间的缩放很难理解，除了可视化研究人员和某些程序员以外，逻辑缩放显得非常神秘。

对于用户来说，平移和缩放一起使用时，产生了更多的导航困难。人们不习惯于在没有限制的三维空间中移动，而且，当三维空间投射到二维屏幕上时，理解起来有一些困难（更多有关三维操作的讨论见第 24 章）。

改善导航

有很多方法可以改善（消除、减少和加速）应用软件、Web 站点和设备的导航。下面是一些最有效的方法：

- 减少目的地数量。
- 提供导航标志。
- 提供总体视图。
- 提供合适的控件——功能映射。
- 调整界面以适应用户需要。
- 避免层次关系。

减少目的地数目

改善导航最有效的方法很显然：减少必需导航的目的地数目。这种所谓的目的地包括模态、表格、对话框、页面、窗口和屏幕。如果模态、页面或者屏幕的数目减至最小，那么用户保持方向感的能力将会显著上升。对于前面提到的四种类型的导航而言，这意味着：

- 将页面和窗口的数目减至最少。一个全屏幕的窗口，加上两三个视图（最多）是最好的。将对话框，尤其是无模态对话框减至最少。如果程序或网站有几十

个不同类型的页面、屏幕、表格，那么它们在任何情况下都是不可导航的。

- 控制窗口或网页的相邻窗格数目，限制到用户需要实现他们目标的最小数目。在独占应用中，最多三个窗格就好了。在网页上，任何超出两个导航区域和一个内容区域的页面将会难以导航。
- 将控件数目的值限制在用户真正需要满足他目标的最佳值。通过人物角色较好地掌握你的用户，会让你避免用户不需要或者不真正想要，因此妨碍他们的控件。
- 应该尽量减少滚屏。这意味着需要让支持窗格有足够的空间来显示信息，这样，就不需要不断地滚屏。二维和三维图形的默认视图和场景（Scene）应该让用户能够确定方向而不需要太多的平移。缩放，尤其是不断地缩放，对于大多数用户来说是最困难的导航类型，所以它的使用应该由用户自由决定，而不是作为一个需求。

很多电子商务网站，因为设计师想用一个通用站点为所有的人提供服务，因此得到了令人困惑的导航。如果一个用户在网站买书，但从不买 CD，对这个用户而言，不应该在他的主窗口强调对网站 CD 内容的访问。这样也为这名用户留下了更多的空间买书，导航也变得更容易了。相反，如果他经常访问账号页面，他的网站版本就应该将账号按钮（或者标签）醒目地呈现在他面前。

提供导航标志

除了减少导航目的地数目，加强用户定位能力的另一种方式是提供更好的参考点——导航标志。就像航海员参考海岸线或者星星来导航，用户参考界面上的持久对象来导航。

在桌面世界里，持久对象始终包括程序的窗口。每个程序都可能有一个主要的顶层窗口。该窗口的特征也可以认为是持久的对象：菜单、工具条、调色板，还有状态栏和标尺这样的可视特征。一般来说，程序的每个窗口都有着独特的外观，这样能很快地识别出来。

在 Web 上，也有类似的规则。最好的 Web 应用，就像 Amazon.com，小心地使用了持久对象，让它们在购物的过程中保持不变，尤其是页面顶端的标签条及左端的搜索和浏览区域。这些区域不仅提供了清晰的导航选项，而且它们的持久存在和固定位置也能够帮助用户定位方向（如图 11-4 所示）。

在设备中，对屏幕也有同样的规则，但硬件控件（hardware control）本身能够起到导航标志（radio button）的作用——当他们能够提供状态的视觉或者触觉反馈时更是如此。

例如，选择单选按钮时，亮灯。如果和软件进行适当集成，即使是刻度盘上的指针位置，也能够提供导航信息。

取决于应用，程序内容的主窗口也能够很容易地识别（尤其是公共信息亭和小屏幕的设备）。某些程序可能对数据提供了一些不同的视图。所以它们屏幕的整体外观随着所选择的视图而改变。然而，桌面应用的不同外观来自于菜单、调色板和工具条的独特组合。这意味着必须把菜单和工具条考虑为导航的辅助。并不需要大量导航标志来成功导航。它们只要是可见就行了。不用说，如果删除了导航标志，它们就不能辅助导航了，所以最好让它们永久地固定在界面上。

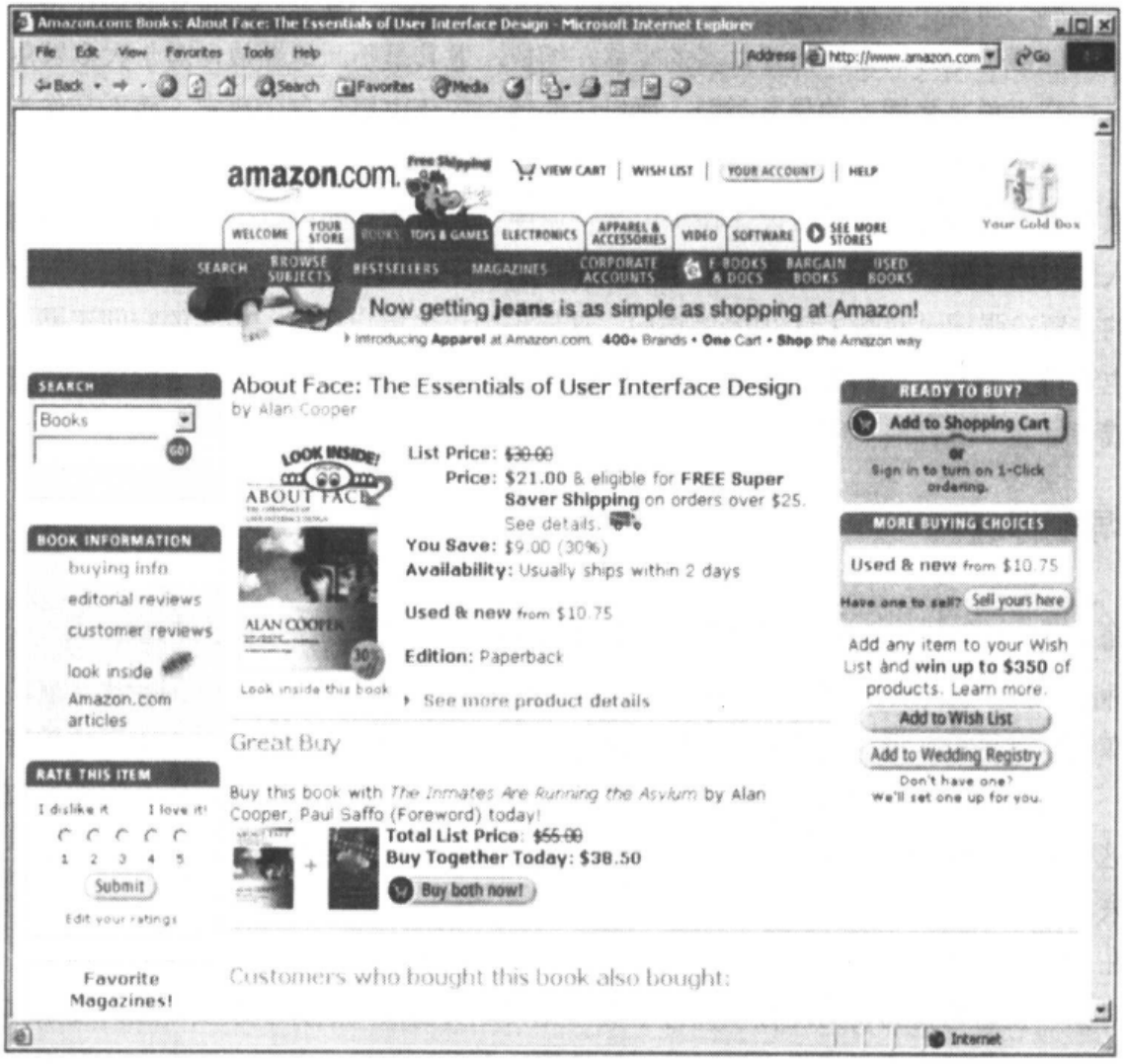


图 11-4　Amazon.com 在大部分页面都使用了很多持久对象区域。如顶部的标签条、旁边的搜索和浏览区域。这不仅能帮助用户判断自己可以去哪里，也帮助用户定位自己的位置。

让网站的每个页面看起来都差不多可能对营销有吸引力，但如果太相似了，就会引起混乱。当然，你应该在页面上一致地使用通用元素，但让不同的空间在视觉上看起来不同——比如让购买页面看起来不同于新的账户页面——会更好地帮助用户定位方向。

【菜单】

程序中最明显的持久对象是主窗口和它的标题，以及菜单条。菜单的部分好处来自它的可靠性和一致性。对程序菜单做出用户预料之外改变会大幅度地减少用户对它们的信任，对菜单项和单独的菜单而言都是这样。是可以将菜单项添加到菜单底部，但对菜单主要的标准项目组合来说，应该只有在非常明显的需要下才能改变。

【工具条】

如果程序有工具条，那么也应该把它当做可识别的标志。因为永久的中间用户更习惯使用工具条，而不是新手用户，改变工具条控件而引发的责难不会像改变菜单项那么强烈。将工具条整个删除当然是引起持久对象混乱的改变。虽然应该提供这么做的能力，但也不应该随意地提供。应该防止用户偶然触发这种改变，一些程序将引起工具条消失的控件就放在工具条上，这是完全不合适的弹射座椅控制杆[1]。

【其他界面标志】

工具调色板和数据显示/编辑的固定区域也应该看做持久对象。它们将使界面导航更容易。明智地使用空白的空间和易读的字体很重要，这样将使标志保持清晰独特。

提供总体视图

界面总体视图起着与导航标志相同的作用：帮助用户定位。不同的是，总体视图帮助用户定位内容，而不是在整个应用中定位。因此，总体视图区域本身应该是持久的，它的内容应该取决于正在导航的数据。

总体视图可以是图形的，也可以是文本的，这取决于内容的性质。一个很好的图形总体视图例子是 Adobe Photoshop 的 Navigator 调色板（见图 11-5）。

[1] 译者注　见第 9 章的解释。

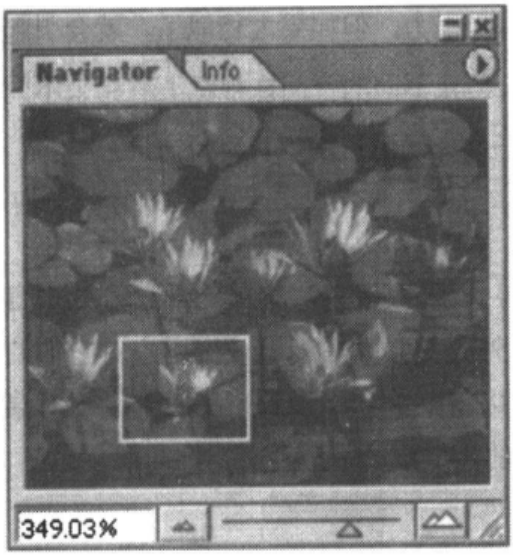


图 11-5 Adobe 使用极佳的总体视图习惯用法。Navigator 调色板提供了大图像的缩略图，而轮廓框代表着在主要显示中当前可见的图像部分。调色板不仅提供了导航背景，而且也可以对主要的显示进行平移和缩放。

在 Web 世界里，总体视图的通用形式是文本的：例如无处不在的层级（breadcrumb）显示（见图 11-6）。而且，大多数层级不仅提供了导航帮助，也提供了导航控件：它们不仅显示用户处于数据结构的什么位置，对于这种链接形式的结构，它们也给用户提供了移动到不同节点的工具。

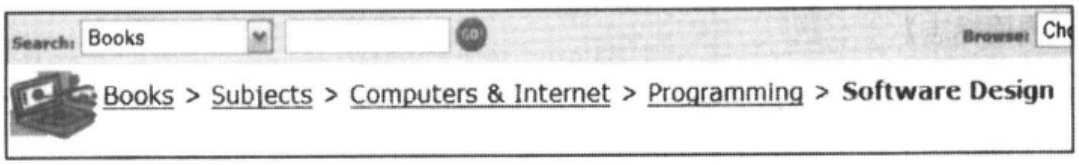


图 11-6 Amazon.com 上的典型层级显示。用户可以看到他们所在的位置，并且可以沿着层级进行点击，从而在链接上导航。

最后一个有关总体视图工具的有趣例子是**注释滚动条**(annotated scrollbar)。注释滚动条对于文本的滚动来说是最有用的。它们巧妙地使用滚动条和文本信息的线性性质，来提供有关选择、突出显示、格式化或者非格式化文本很多其他可能属性的位置信息。当拖动滚动条到适当的位置时，就可以看见这些项目的位置提示信息。当处于注释位置时，可以看到文本的注释特性（见图 11-7）。Microsoft Word 使用了注释滚动条的变体，在滚屏时保持活动的工具提示(tool tip)中显示页码和最近的标题。

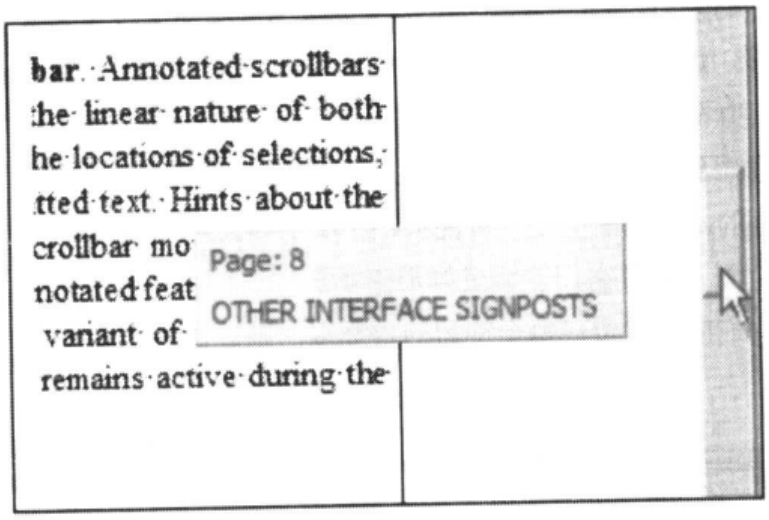


图 11-7 一个注释滚动条。滚动条上的标记表示文本中的位置，如突出显示的章节显示在这里。当滚动条经过该标记时，标记暗示的位置显示在屏幕上。

提供合适的控件——功能映射

映射描述了控件、它影响的事物，以及预期结果之间的关系。当控件与它影响的事物之间没有视觉上或者象征上的关系时，映射关系很不自然。不自然的映射关系影响用户的进度，用户不得不停下来思考控件和它影响的事物之间的关系，从而打断了流状态。控件到功能的不自然映射也增加了用户的认知负担，并且可能会产生严重的用户错误。

一个很好的例子来自煤气灶，而不是数字世界。任何做过饭的人，都会对做饭用的煤气灶旋钮没有合适地映射到它控制的火炉感到恼火。如图 11-8 所示，典型的煤气灶有四个火炉，每个火炉占用正方形的一角。然而，控制火炉的旋钮却在前边的面板上成直线排列。

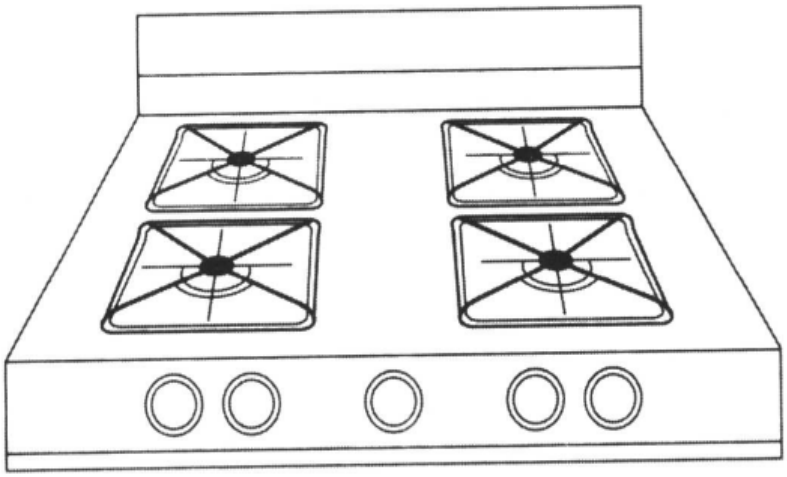


图 11-8 炉顶控件的物理映射很不自然。最左边的旋钮控制的是前端左边还是后端左边的火炉？用户必须在使用火炉时，重新确定它们之间的关系。

在这个例子中，问题是物理映射。使用控件的结果应该显而易见：当你旋转一个旋钮时，会点燃一个火炉。然而，控制的目标——哪个火炉会变暖？——不清楚。旋转最左边的旋钮控制的是左边前端的火炉，还是后端的火炉呢？用户必须通过试验或者参考

旋钮下的图标来确定。这种不自然的映射，迫使用户在每次使用煤气炉时都不得不重新确定映射关系。虽然随着时间推移，这种认知过程会慢慢变成无意识的行为，但它仍然存在。如果用户匆匆忙忙或者注意力不集中（在人们准备膳食时，通常会这样），就会很容易出错。用户因为旋错了旋钮而感到自己很愚蠢，或者直到用户意识到自己的错误之前，食物没有加热，这些都还算好的，最糟糕的是，可能会不小心引起厨房的火灾。

解决方案是调整旋钮的物理位置，这样它们能够更好地表示所控制的火炉。旋钮不必以火炉同样的模式来布局，但它们应该定位成每个旋钮的目标很清晰。图 11-9 中的炉子就是一个有效的控件映射例子。

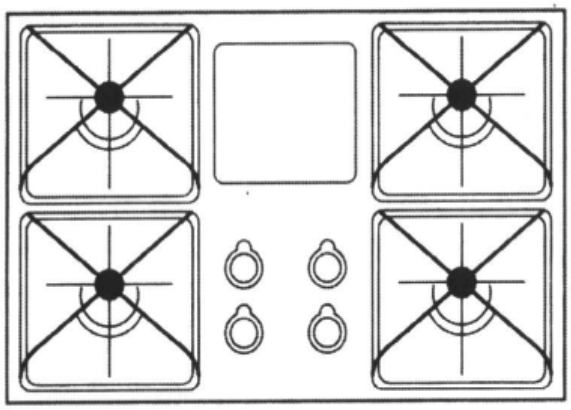


图 11-9 清晰的空间映射。在炉顶上，旋钮和火炉的映射非常清晰，因为旋钮的空间布局清晰地将旋钮与火炉联系起来。

在这种布局中，很清楚，左上端的旋钮控制着左上端的火炉。每个旋钮的位置都在视觉上清晰地显示出它所控制的火炉。Donald Norman(1989)将这种更直觉的布局称为“自然映射”。

另一个不同类型的不自然映射例子如图 11-10 所示。在这种情况下，概念到动作的逻辑映射不清晰。

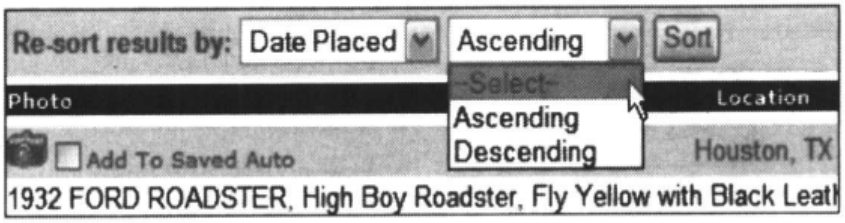


图 11-10 逻辑映射问题的例子。如果用户想首先看到最近的项目，他是选择 Ascending 还是 Descending 呢？这些术语和用户对时间的理解没有很好的映射关系。

这个网站使用成对的下拉菜单，根据时间对搜索结果进行排序。对第一个下拉菜单的选择决定了第二个下拉菜单提供的选择内容。在第一个菜单中，当选择 Date Placed 重新排序时，第二个下拉菜单显示 Ascending 和 Descending 选项。

不像前面的炉子旋钮，这里控件的目标非常清晰：下拉菜单选项会影响下面显示的

列表。然而，使用控件的结果不清晰，如果用户选择上升时，会获得什么样的排列次序呢？

正是这些用来表达日期排序选项的术语，使得用户如果想在列表中先看到最近的项目时，无法确定该怎么选择。上升和向下不能很好地映射到用户对时间的心智模型。人们不认为时间存在上升和向下之分。相反，他认为时间和事件可以分为最近的和最老的。快速修正该问题的方法是，将选项使用的单词改为“首先显示最近的”和“首先显示最老的”，如图 11-11 所示。

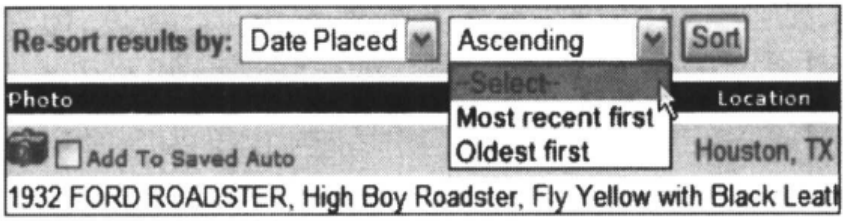


图 11-11 清晰的逻辑映射。使用术语“最近的”和“最老的”使用户容易映射到基于时间的排序。

无论是创建家电、桌面应用还是 Web 站点，你的产品都可能存在映射问题。映射是一个关注细节就会得到回报的领域：即使没有多少时间来做出改变，你仍然可以通过寻找和解决映射问题来让产品得到明显的改善。如果说结果，那就是产品会更加容易理解，并且使用起来更愉悦。

调整界面以适应用户需要

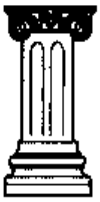
调整界面意味着对界面进行组织来减少导航。在实践中，这意味着将最常用的功能和控件放在立即可获得的，最方便的位置，供用户访问；而把很少使用的功能放在界面的隐蔽处，用户不会不小心触发。很少使用的功能不应该从程序中移除，但应该把它们移除到用户的日常工作区域之外。

公理：为典型的导航调整界面。

适当的界面调整中，最重要的原则是**相称的努力**(commensurate effort)。它可以适用于所有的用户，特别是永久的中间用户。这个原则只不过是说，人们愿意为更有价值的事物努力工作。所获得的当然是奋斗者眼里的价值，这与在技术上实现一个特性有多困

难无关，相反，它完全与用户的目标相关。

如果用户真的想获得某种东西，那他愿意付出更多的努力。如果一个人想成为好的网球运动员，他会到球场上去，努力训练。对于那些不喜欢网球的人来说，大量运动是一种枯燥无味的努力。如果一个用户需要用有趣的标题，优美的格式，有着多个分栏和多种字体的文档来取悦他的老板，他就会有强烈的动机来探索程序的秘密，学习如何实现这些特性，他会在项目中付出相称的努力。如果其他用户只想用单个分栏和一种字体来打印简单的旧式文档，他们就没有学习这些高级格式的动机。



公理：如果回报值得，用户愿意付出相称的努力。

这意味着，如果你在程序中添加一些管理起来非常复杂的特性，只有在回报值得的情况下，用户才愿意忍受这种复杂性。这就是程序的界面不能为了实现简单的功能变得复杂，但可以为了实现复杂的功能而变得复杂的原因（只要这些功能不是经常使用）。

从界面的角度看，用户必须花费一些额外的精力才能调用一些高级的特性是可以接受的，无论是需要在菜单中寻找，打开对话框，还是打开抽屉。相称的努力的原则允许我们这样调整界面：简单而经常使用的功能一直放在手边，随时可以访问，而用户较少访问但有很大的回报的高级特性可以安全地隐藏起来，在他们需要的时候进行调用。一般来说，界面中的控件和显示应该根据三个属性来组织：使用频率、混乱程度和暴露程度。

- **使用频率**（frequency of use）。在典型的日常生活使用模式中，意味着控件、功能、对象或者显示的使用频率。经常使用的项目和工具（每天几次）应该直接可以访问。正如第9章中讨论的，更少使用的项目或者每天使用一两次的项目，应该不超过一到两次点击就可获得，其他项目则应该可以通过两到三次点击获得。
- **混乱程度**（degree of dislocation）。指的是因为调用特殊的功能/命令，使得界面或者应用处理的文档/信息突然改变。一般来说，将这种类型的功能在界面中隐藏起来是一个好主意（见第9章有关弹射座椅控制杆的详细解释）。
- **暴露程度**（degree of exposure）。处理的是不可逆转或者可能会有其他危险的功能。洲际导弹系统要求由两个人在一个房间的对面同时打开钥匙。就像将引起混乱的功能一样，你应该使用户更难发现这些功能。

当然，随着用户慢慢熟悉这些特性，他们会寻找快捷方式，因此你必须提供快捷方

式。如果软件遵循相称的努力的原则，学习曲线不会真的消失，但它会从用户心中消失——这才是恰到好处。

避免层次关系

层次关系是程序员使用时间最长的工具之一。程序中的大部分数据，以及操作这些数据的代码，都是以层次关系存在的。基于这个原因，很多程序员喜欢在界面中展现层次关系（实现模型）。正如我们所见到的，早期的菜单就是层次关系的。但是抽象的层次关系对于用户来说很难导航。这种情况对于程序员来说难以理解，因为他们本身对层次关系非常适应。

大多数人对商业关系和家庭关系中的层次非常熟悉，但是，大多数人在存储或者检索信息时，层次关系并不是一个自然概念。大多数机械存储系统是简单的，不是由简单的存储对象序列（诸如书架）组成的，就是一系列的序列，深度只有一层（好比文件柜）。这种用单个层次的分组来组织事物的方法非常普遍，可以在你的家里或者办公室随处找到。因为它从来不会超过一个嵌套层次，我们将这种存储范式称为**单层分组**（monocline grouping）。

程序员非常熟悉这样的嵌套系统：一个对象的实例存储在同一个对象的另一个实例中。大多数人难以理解这种想法。在机械世界里，复杂的存储系统，根据需要，在不同层次使用不同的机械形式：对于文件柜，你不会在柜子里看到柜子，或者在抽屉里看到抽屉。即使是抽屉内的文件夹——这种不同的嵌套也很少超过两个嵌套层次。在当前大多数窗口系统的桌面隐喻中，你可以无限制地在文件夹中嵌套文件夹。大多数计算机新手用户遇到这种范式时无疑会感到困惑。

基于相同的特征，大多数人将他们的论文（以及其他项目）存储在一系列堆栈或者堆里。Acme 论文放在这里， M 项目的论文放在那里，个人资料放在抽屉里。Donald Norman(1994)称之为堆橱(pile cabinet)。只有在计算机里，人们才会将 M 项目的文档放在活跃客户文件夹里，而活跃客户文件夹又会存储在客户文件夹里，客户文件夹又存储在商业文件夹里。

计算机科学将层次关系结构看做工具，来解决管理大批量数据的实际问题。当这种实现模型以显式模型（manifest model）呈现给用户时（更多的模型知识见第 2 章），用户会感到困惑，因为它与用户有关存储的心智模型相冲突。单层分组是用户有关软件的心智模型。单层分组在计算机以外的世界里那么普遍，因此交互设计师违背该模型时实在需要冒些风险。

使用单层分组对计算机中广泛存在的大量数据进行物理管理是不够的，但这并不意

味着它作为显式模型没有用。这个难题的解决方法在于，根据用户想像的方式，如单层分组来显示结构，但提供只有深层关系组织才能提供的搜索和访问工具。换句话说，与其强迫用户在深而复杂的树形结构中导航，不如给他们一些工具，把合适的信息拿来，放在他们面前。我们将在第 16 章讨论如何实现这种设计方案。

12

理解撤销

撤销是一种非凡的功能，能允许我们恢复前面的动作。简单而优雅，这种功能有着明显的价值。然而当我们从目标导向的视角来看撤销时，在意图和方法方面存在很多变体。撤销当然对用户重要，而且没有我们想像的那么简单。本章研究用户理解撤销和使用撤销功能的不同方式。

用户和撤销

传统上，人们认为撤销是用来对困境中的人们进行营救的，是穿着闪耀盔甲的武士，在山脊上飞驰的骑兵，在最后一刻突然出现的超级英雄。

作为一种计算机功能，撤销没有什么优点。因为计算机本身不会犯错误，它不需要什么撤销功能。而另一方面，人不断犯错误，撤销是为他们而存在的额外功能。根据这些观察，我们可以发现，在程序所有的功能中，对撤销建模时应该尽量不要根据它的构造方法（实现模型），反过来，应该尽量接近用户的心智模型。

人不仅会犯错误，而且犯错误是人们日常行为的一部分，从计算机的角度来看，一

开始就弄错、一次不对的浏览、暂停、打嗝、打喷嚏、咳嗽、眨眼、笑或者发出“嗯”、“你知道”等都是错误，但从人类用户的角度来看，这些太正常了。人的错误是如此普遍，如果认为这些都是错误或者是反常的行为，将会给你的软件设计带来负面的影响。

用户有关错误的心智模型

用户不相信，或者至少不愿相信他们会犯错误。这是用户心智模型通常不包含错误的另外一种说法。遵循用户的心智模型意味着不责备用户。然而，实现模型基于“CPU不会犯错误”这一事实，遵循实现模型意味着将所有的责任归于用户。因此，大多数软件认为自身是无可责备的，任何问题都纯粹是用户的错。

对于用户界面设计师来说，解决方法是完全放弃用户会犯错误的想法——意味着用户所做的一切，在他看来都是合法而合理的。用户不会也不愿意在内心承认错误，所以，程序不应该在与用户交互时与用户的动作抵触。

撤销支持用户探索

如果从“用户所做的一切都不会构成错误”这一立场出发来设计软件，我们立即会从不同的角度来看事情。我们不再将用户想像成代码模块，或者驱动计算机的外围设备，而将用户想像成一个探索者，在探索未知的世界。我们知道，探索，包括不可避免地遇到死胡同和封闭的峡谷，沿着走不通的路，到达枯竭的干洞。对于人类来说很自然，进行尝试，调整行动，探索未知世界以发现它们的边界。只有通过尝试，否则用户又如何知道使用工具能做些什么呢？当然，愿意尝试的程度因人而异，但大多数人至少愿意尝试。

因为他们像计算机一样思考而获得高薪（Cooper,1999）的程序员们，却把这些探索行为当成必须用代码处理的错误。从实现模型看——程序员必然的视图——这种温和而没有恶意的探索代表着一系列的错误。从更具有启蒙性的，心智模型的角度看，这些行为是自然而正常的。程序要么回避这些可理解的错误，要么协助用户探索不同的选择。撤销是支持软件用户界面探索的主要工具。如果用户决定了，撤销允许用户恢复一个或者更多的先前动作。

撤销很大的一个好处纯粹是心理上的，它让用户宽心。如果有信心在任何时候都能返回，那么你更容易进入一个洞穴。撤销功能将绳梯放在地上，让新用户确信能够从闭塞的洞穴里返回，进一步支持用户探索的意愿。

奇怪的是，在他们需要之前，用户通常不会想到要撤销。就像灾难发生之前，家庭

主妇也不会想到保险一样。用户经常在没有完全准备好的情况下冲进洞穴。只有在遇到麻烦的时候，才会寻找绳梯——撤销。

设计撤销功能

虽然用户需要撤销，但撤销并不支持他们任务的特定目标，而是在达到实际目标的过程中提供了一个必要条件：信任。撤销并不直接有助于达到用户的目标，但能防止发生负面情况破坏用户的努力。

取决于各种情况和期望，用户对撤销功能有着不同的形象化理解。如果用户是一个计算机盲，他会将撤销当做一个无条件的应急按钮，而将自己从运气不佳的混乱遭遇中解脱出来。稍有经验的用户可能会将撤销功能理解为已删除数据的存储设施。真正有着逻辑头脑而且更理解计算机的用户，可能将撤销看做程序的堆栈，可以以相反的次序撤销。为了创建更有效的撤销功能，必须尽可能更多地满足我们能够预料的用户心智模型。

成功地设计撤销系统的秘诀在于它支持常用的工具，并且能够避免任何暗示用户失败的信号（不管是视觉、声音或者文本信号）。应该尽少用来恢复错误，而更多地支持探索。错误通常是简单的、不正确的行为。而探索与此相反，是一系列的探查步骤，其中一些需要保留，而另外一些需要放弃。

撤销最好作为程序范围内的全局功能，可以撤销上一次的动作，而不管它是由直接操作还是由对话框完成。这样的话，撤销应用在嵌入对象时存在一些问题。如果用户改变嵌入在 Word 文档中的电子表格，点击 Word 文档，然后调用撤销，最近的 Word 动作被恢复，而不是最近的电子表格动作被恢复。用户无法理解这一点。撤销功能没有无缝地实现电子表格文档和字处理文档之间的接合。这里，撤销不再是全局的，而是模块化的。这本质上不是撤销的问题，而是嵌入技术的问题。

撤销的类型和变体

没有足够的术语来描述已有的撤销类型，这在软件行业已经司空见惯了，它们都统一称为撤销。这种语言的不足也导致了创新的缺乏，没有出现一些更新更好的撤销变体。在本节中，我们定义几种撤销的变体，并且解释它们的不同。

渐增动作和过程动作

首先，考虑撤销针对的对象：用户的动作。典型应用中的典型用户动作有一个过程部件——用户做了什么，还有一个可选的数据组件——影响了什么信息。当用户调用撤销功能时，动作的过程部件进行恢复，如果该动作有一个可选的数据部件——用户添加了数据或者删除了数据——那么，相应地，删除或重新添加数据。剪切、粘贴、画图、键入和删除都是有数据部件的动作，所以对它们的撤销涉及到删除和替换影响到的文本或图像。包含数据部件的这些动作称为**渐增动作**（incremental actions）。

很多能够可撤销的动作没有数据变化，就像在字处理软件中，对段落重新设置格式，或者画图程序中的旋转那样。这些都是对数据的操作，既没有添加数据也没有删除数据。这样的动作（只有一个过程部件）称为**过程动作**（procedural actions）。大多数现有的撤销功能并没有区分渐增动作和过程动作，只是简单地恢复最近一个动作。

隐蔽撤销和解释性撤销

通常，我们通过菜单项，或标签/图标不变的工具栏控件来调用撤销。用户知道触发该习惯用法会撤销最近的操作，但它并没有提示操作是什么，这叫做**隐蔽撤销**（blind undo）。另一方面，对将被撤销的具体操作，如果还有一些文字或者视觉描述，这种撤销叫做**解释性撤销**（explanatory undo）。

例如，如果用户的最近一次操作是输入单词“设计”，菜单上的撤销功能提示撤销对单词“设计”的输入。解释性撤销与盲撤销相比，使用起来更愉悦。将解释性撤销放在菜单项上很容易，但放在工具条控件上会比较困难，虽然将解释性撤销放在工具提示（见第 29 章）上是一种很好的折中。

单次撤销和多次撤销

现在通常用到的两个最熟悉的撤销类型是单次撤销和多次撤销。**单次撤销**（single undo）是一种最简单的变体，恢复最近的用户动作，既可以是过程动作也可以是渐增动作。两次连续的单次撤销通常会撤销已经做出的撤销，并且让系统返回在第一次撤销激活之前的状态。

因为易于操作，所以单次撤销非常有效。用户界面清晰，容易描述和记忆。用户不费力就得到一份免费午餐。这是到目前为止最常实现的撤销功能，并且对于大多数程序来说肯定够了，即使没有达到最好。对于用户来说，缺少简单的撤销功能，可能就会完全放弃某个产品。

用户通常会马上意识到大多数的命令错误：做完某件事，如果感觉糟糕或者看起来不太好，他会停下来评估当前情形。如果程序表达得很清楚，他会看到自己的错误，并且选择撤销功能，将事物的状态恢复到先前正确的状态，接着继续。

多次撤销（multiple undo）可以连续调用好几次，它可以以相反的时序顺序恢复多个操作。任何有简单撤销功能的程序必须记住用户的最近一次操作，如果可以的话，缓存任何改变的数据。如果程序实现多次撤销，它就必须维护一个操作堆栈，堆栈的深度可以根据用户的偏好来设置。每次调用撤销，它执行一次渐增撤销来恢复最近的操作，如果需要的话，替换或者删除数据，并且放弃存储在堆栈中的操作。

【单次撤销的限制】

单层次功能的撤销的最大限制在于，用户会无意中覆盖救命的撤销功能。如果用户没有马上意识到错误，就会出现问题。例如，他删除了文本的六个段落，又删除了一个单词，接着他发现那六段内容的删除是错误的，应该恢复。不幸的是，现在调用撤销只能挽回一个单词，六个段落永远丢失了。撤销功能的失败在于，它行为精确，但并不实用。任何人都能看出六个段落比一个单词更重要，然而程序为了一个单词，完全放弃了六个段落。程序的盲目性让它保留了二角五分的硬币，放弃了五十美元，仅仅因为二角五分硬币是最后提供的。

在一些程序里，任意一次鼠标单击，尽管这些单击很无辜，也会导致单次撤销功能放弃用户最近所做的有意义的事情。虽然多次撤销能解决这些问题，但它自己也存在一些问题。

【多次撤销的限制】

为了避免单层次撤销的弱点，有了相同渐增撤销的多层次实现。程序保存用户做过的每一个动作。通过反复地选择撤销，每个动作都能够以初始调用的相反次序来撤销。在上面的脚本提纲中，用户能够通过第一次调用撤销来恢复删除的单词，接着第二次调用，来恢复删除的前六个段落。一个冗余的单词删除，对于能够恢复六个有价值的段落来说是一个小小的代价。人们一般不会注意到删除一个单词的附加工作，就像我们不会注意到救护车的路费一样。当生命处于危险时，不要在乎一些小的代价。但是，这并没有改变撤销机制建在错误的模型基础上这一事实，在其他一些环境中，严格的 LIFO 次序（后进先出）的撤销功能使得治疗和疾病本身一样痛苦。

想像一下，用户删除了六个段落文本，接着打开另外一个文档，并进行全局的查找和替换功能。为了能够恢复丢失的六个段落，用户必须首先撤销相当复杂的全局搜索和替换操作。这次的介入操作不像先前删除一个单词的例子。这个介入操作复杂而困难，

并且撤销它明显是让人很不愉快的附加工作。因此可以肯定，能够在队列中选择撤销哪一个操作要更好，并且能够将中间插入的（但合法的）操作保持不变。

【多次撤销的模型问题】

多次撤销存在的问题不是因为它的行为，而是因为它的显式模型（manifest model）。大多数撤销功能以一种不友好的功能为中心的方式构建，它们一个功能一个功能记住用户的动作，并且以单个功能为基础区分用户动作。在根据实现模型创建显式模型的传统方法中，撤销系统倾向于对代码和数据结构建模，而不是对用户目标建模。每点击一次撤销按钮，精确地恢复一次以功能为单位的行为。对于大多数因为用户输入错误而产生的问题来说，一个功能接着一个功能进行恢复是合适的心智模型。用户发现错误后立即修正，通常局限在两个或者三个功能之内。例如，Windows 95 的绘图程序有固定的，三个动作的撤销限制。然而，当程序变得更复杂时，渐增的多次撤销模型不能随之扩展。

【指望后进先出】

当用户进入逻辑的死胡同（而不是简单的输入数据错误）时，通常进行了几个复杂的步骤，在意识到自己迷路之前，已经来到了一个未知的世界，他需要返回已知世界。然而，这时他可能已经执行了几个互相交织的操作，其中只有一些是他不想完成的。他可能想保留一些动作，而放弃另外一些动作，不一定是按严格的反向次序。如果用户想输入一些文本，编辑文本，接着决定撤销对文本的输入，但不撤销对它的编辑，会怎么样？这样的操作实现和解释起来都有问题。Neil Rubenking 提供了一个例子：假设用户把 tragedy 全局替换成 catastrophe，接着将 cat 替换成 dog。要撤销第一个操作，而不撤销第二个操作，程序是否能够可靠地修复所有的 dogastrophes？

在这种更复杂的情况下，将撤销表达为线性的后进先出堆栈，不能像简单情况下那样奏效。用户可能希望像研究菜单一样研究自己的动作，并且选择不连续的子集来恢复，并保留其他内容。这要求解释性撤销比正常的多次隐蔽撤销有更健壮的表达。另外，从表达中选择意味着更加复杂。对队列中的操作进行表达，来清晰地显示用户实际上在撤销什么，就更难了。

恢复

恢复（redo）功能是撤销采用实现模型的结果，在实现模型中，撤销操作必须以相反的顺序进行，如果不首先撤销先前所有合法插入的操作，则没有操作可以被撤销。恢复实际上是对撤销进行撤销，如果程序员已经实现了撤销，恢复很容易实现。

在多次撤销中，恢复能避免糟糕情形的出现。如果用户想收回半打操作，他点击多次撤销，等待回到期望的状态。这种情况下，很容易一下按了多次撤销，他马上看到自己做了不应该的撤销，恢复通过允许用户对撤销动作进行撤销，来解决这个问题，使用户能返回最后一次正确的动作。

很多实现了单次撤销的程序将最后一次撤销动作当做可以撤销的动作，实际上，这样，第二次调用撤销功能就成了最小的恢复功能。

分组多次撤销

Microsoft 有一个不同寻常的撤销功能，多次撤销的变体，我们称之为**分组多次撤销**（group multiple undo）。它是多个层次的，在撤销堆栈中显示了每次操作的文字描述。你可以检查过去操作的列表，并且在列表中选择一些操作进行撤销。然而，你不是撤销一次操作，而是所有操作返回原来的状态（见图 12-1）。

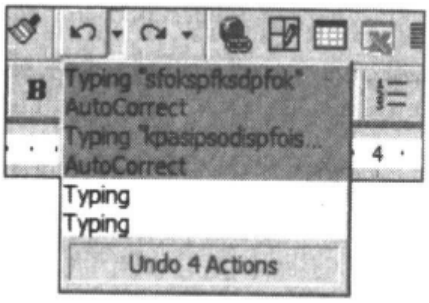


图 12-1 Microsoft 的撤销/恢复功能有点不同寻常，你可以撤销多个动作，只要作为一组。但不能选择撤销你三次动作之前的动作。恢复也是一样的。

实际上，如果不恢复所有的中间插入的操作，你不能恢复六个丢失的段落。在选择了一个或者更多的操作撤销后，撤销操作的列表在恢复控件中以相反的顺序获得。恢复的工作方式和撤销一样。你可以像所期望的那样选择恢复多个操作，那么在指定操作之前的所有操作都可以恢复。

该程序为这种情况提供了两种视觉提示，如果用户选择列表中的第五项，那么该项及在此之前的四项都被选中了。一个文字提示显示："撤销 5 个操作"。设计时不得不添加文本符号的现实告诉我们，无论程序员采取什么样的构造方式，用户采用的都是不同的心智模型。用户以为他可以沿着列表，从过去的动作中挑选一项来撤销。程序没有提供该选项，所以显示了文字提示。这就像一张门的拉动把手上贴上了推的标志，任何人还是去拉一样[1]。

---

[1] **译者注** 指用户的行为遵循自己的心智模型。

类似撤销行为的其他模型

最简单的撤销显式模型遵从了用户的心智模型："我刚才做了一些事，但现在希望刚才没有做就好了，我想点击一个按钮撤销刚才所做的事"。不幸的是，随着情况越来越复杂，这种显式模型很快就偏离了用户的心智模型。在本节中，我们将讨论类似撤销行为的模型，它们的工作方式和更标准的撤销和恢复习惯用法不同。

比较：这看起来怎么样

除了在一定时期内有助于用户做出决定之外，成对的撤销和恢复功能是一个方便的比较工具。比如说，你想比较没有对齐的右边距和对齐的右边距在视觉效果上的差异，开始选择没有对齐的右边距，然后对齐，现在你单击撤销按钮，看没有对齐的右边距，再按恢复，再次看对齐了的边距。在效果上，在撤销和恢复之间进行切换实现了**比较**（comparison）或**如果……又怎么样**(what-if)的功能：它正好以实现模型的形式表达。如果这样的功能添加到遵循用户心智模型的界面上，它可能显示为一个比较控件。该功能让你反复地向前或者向后在视觉上比较两个状态。

一些 Sony 电视机遥控器有一个标为跳转的功能，它在当前频道和先前的频道之间切换——这对于同时看两个节目来说非常方便。跳转功能用一个命令提供了类似一对撤销和恢复的功能，而减少了 50%的附加工作。

作为比较功能使用时，撤销和恢复实际上是一个功能，而不是两个。一个意在"施加改变"，而另外一个意在"不要施加改变"。单个比较按钮可能会更精确地将动作呈现给用户。虽然我们在面向文本的字处理程序场景下已经描述了这种工具，但比较功能可能在图形操作或者绘图程序中更有用，在这里，用户不断地变换图形。快速而容易地比较施加了变换和没有施加变换之间的差异，这对于数字艺术家来说是帮助很大。

毋庸置疑，比较功能是一个高级的功能，正如跳转功能不会为大多数电视用户所使用一样，比较功能也只是大多数经常使用的用户喜欢的特点。因为绘图程序通常只有那些一直使用的人才会频繁使用，这不应该影响它的有用性。对于这类程序，满足频繁使用的用户是一种合理的设计选择。

分类撤销

退格键实际上是一个撤销功能，虽然它非常特殊。当用户输入错误时，退格键"撤销"错误的字符。如果在用户错误输入之后，又进行了一个不相关的操作，如重新设置

段落格式，接着反复地按退格键。错误输入的字符被删除，而段落格式设置会跳过去。看你怎么认为，如考虑到用户能够在任意选择的位置进行不连续的撤销，这样相当方便。你也可以认为这对于用户来说是一个陷阱，因为他可以移动光标并且无意中用退格键删除字符，而这些字符不是最后输入的。

从逻辑上说，后面这种情况有些问题。但经验显示，这对用户来说经常不是问题。这种不连续的渐增撤销很自然，而且容易使用——难以用语言解释，因为所有事物都是可见的：用户能清楚地看到什么被退格键删除了。退格键是经典的渐增撤销的例子。在忽略其他插入动作的同时，仅仅恢复一些数据。然而，如果你想像一个带有指针的撤销功能，指针可以移动，并且可以撤销指针指向的最近功能，你可能会认为这种功能不能得到清晰的管理，会困惑典型的用户，让他感到惊讶。经验告诉我们，退格键不会那样。它表现很良好，是因为它的行为与用户关于光标的心智模型是一致的：光标是添加字符的源头，那么认为它是删除字符的位置也很合理。

借助同样的知识，我们能够创建不同类别的渐增撤销，诸如格式撤销功能，它仅仅撤销之前的格式命令，和其他具体类别相关的撤销动作。如果用户输入一些文本，将它改为斜体，接着输入更多文本，增加段落的缩进，再输入更多的文本，按下格式撤销键，结果仅仅是段落的缩进撤销了。第二次按下格式撤销键会恢复斜体操作。但格式撤销不会影响内容。

分类撤销对于非文本程序来说意味着什么呢？在绘图程序中，例如存在单独的颜料应用工具：变换、剪切和粘贴的撤销命令。在程序中，确实没有理由让每一个具体操作类别不能拥有单独的撤销功能。

颜料使用工具包括所有的绘图实现：铅笔、钢笔、填充、喷雾、刷子，以及所有的形状工具：矩形、直线、椭圆、箭头。变换包括所有的图像操作工具——修剪、锐化、色调、旋转、对比、线段粗细。剪切和粘贴工具包括所有的套索，选取框、克隆、拖动及其他重定位工具。不像字处理程序中的退格键功能，在绘图中撤销颜料使用工具具有时序性，并且与选择无关。这也就是说，首先被删除的颜料会是最近使用的颜料，而不管当前的选择。在文本中，存在一个隐含的从左上角到右下角的次序。从右下角到左上角的删除映射到一种强烈的、固有的心智模型；这看起来很自然。在绘图中，不存在这么常规的次序，所以任何删除次序，除了基于输入顺序的，对于用户来说都是令人不安的。

更好的替换方法可能是仅对当前选择进行撤销。例如，用户选择图形对象，并且请求撤销变换，将会恢复已经应用到所选对象上的最后变换。

大多数软件用户对渐增撤销很熟悉，可能会发现分类撤销新鲜而令人困扰。然而，退格键无处不在的现实显示，渐增撤销是一个习惯行为，用户发现后会有帮助。如果更

多的程序有着模态的撤销工具，用户会很快习惯它们。他们甚至会更加期望在字处理程序中找到的退格键工作方式。

被删除的数据缓冲区

当用户长时间在一个文档上工作，他会更期望存在一个已删除文本的仓库。这不是因为他发现渐增撤销的命令没用，而是因为恢复动作不再和具体的功能很相关。再以丢失的六个段落为例。如果它们和一打复杂的格式命令混在一起，通过撤销来恢复和重新键入一样困难。用户在想："如果程序记住我所删除的东西，把它们放在一个特殊的地方，就可以直接得到我所想要的"。

用户想像的是他动作包含的数据部件库，而不是简单的过程后进先出堆栈——一个已删除数据的缓冲区。用户想得到丢失的文本，而不关心哪些命令删除了它们。通常的显式模型迫使用户不仅要意识到每一个步骤，而且要依次恢复每一个动作。要想创建一个更能经受用户检验的功能，我们除了需要正常的撤销堆栈外，还要创建一个收集所有已删除文本或者数据的独立缓冲区。无论在什么时候，用户能像打开一个文档一样打开这个缓冲区，并且使用标准的剪切粘贴，或者点击拖动等习惯用法来观察和恢复期望的文本。如果已删除数据的缓冲区中的项标有简单的时间戳和文档名字，导航应该会非常简单和形象化。

用户愿意按照自己的意愿而不是按照顺序浏览已删除数据的缓冲区。找到这六个丢失的段落是一个简单的视觉过程，而不用管用户已经完成的复杂插入步骤的数目或者类型。已删除数据的缓冲区应该和常规的渐增多次撤销一起提供，因为它们具有互补性。数据无论如何应该保留在缓冲区中。这种特性对于很多程序非常有用，无论是电子表单，绘图程序还是发票生成器。

里程碑和复原

用户偶尔想返回到很久以前的地方，但当他们这么做时，细粒度的动作不是很重要。渐增撤销的需要仍然存在，但在多数情况下，辨识以往操作的单个部件太过详细了。在第 13 章中讨论的**里程碑**（mile stoning），简单地拷贝整个文档，就像照相机快速将一幅图像在时间上凝固一样。因为里程碑涉及到整个文档，它通常直接通过文件系统来实现。里程碑和其他撤销系统之间的主要差别在于，用户必须显式地调用里程碑，记录一个拷贝或者文档的快照。在用户完成以后，他可以继续安全地修改原始文档。如果他后来觉得自己的改变不合适，它可以返回到保存的状态——文档的先前版本。

有很多工具在源代码中支持里程碑的概念。然而，据作者所知，到目前为止，没有

程序直接将它显示给用户。相反，它们依赖于文件系统的界面，正如我们已经看到的，这一点对于很多用户来说很难理解，如果里程碑以非文件系统的用户模型交付给用户，实现起来应该非常容易，管理也非常简单，单个按钮控件可以保存当前状态的文档。用户可以按照他期望的间隔保存任意数目的版本。要回到先前某个里程碑版本，用户可能希望访问**复原**（**reversion**）功能。

在第 13 章中讨论的复原功能非常简单，或许太简单了。它的菜单项是“复原到里程碑”。这对文件系统来说已经可以了，但作为撤销功能的一部分考虑时应该提供更多信息。如它应该显示一份可用的已保存版本列表，以及对每个版本提供一些有关的信息，如记录的时间和天数，记录人，大小及一些可选的用户输入注释。用户可以选择其中一个版本，程序会加载这个版本，并且放弃其他已经插入的改变。

冻结

冻结（freezing）是里程碑的反面，包括锁定文档中的数据，让它不会被改变。虽然新的数据可以添加，但任何已经输入的信息不能再修改。已有的段落不能改变，但新的段落可以添加到旧段落之间。

该方法对于图形文档来说要比文本文档更有用。这很像艺术家用定影剂来喷射绘画作品。所有在那一刻之前做的标记现在都变成永久的了，然而新的标记可以按照你的意愿添加。已经显示在屏幕上的图像被锁定，并且不能改变。但新的图像可以添加到旧的图像上。Procreate Painter 应用程序用它的 Wet Paint 和 Dry Paint 命令提供了相似的功能。

防撤销操作

一些操作不能撤销，因为它的一些动作调用了不在程序直接控制下的设备。例如，电子邮件发送以后就不存在撤销了。计算机电源关闭了，但没有保存数据，也不能对损失进行撤销。然而，很多操作伪装成防撤销的，但其实它们很容易恢复。例如，当你在大多数程序中第一次保存文档时，你可以选择文档的名字。但几乎没有程序让你对该文件重新命名，当然，你可以使用“另存为”存成另外一个名字，但这只会用新的名字创建另外一个文件，而旧的文件在旧的名字下保持不变。为什么不提供文件名撤销呢？因为它不属于撤销的传统视图，程序员通常不提供改变文件名的真实撤销功能。花一些时间在你的应用程序中找一找，看看能否找到一些应该能被撤销，但现在无法撤销的功能。你会惊讶地发现能找到很多这样的功能。

13

重新思考“Files”和“Save”

如果曾经试过教你的母亲如何使用计算机，你会知道，难度不只来自问题。事情刚开始很顺利，启动字处理软件并且敲入一个字母，她也和你一样。当你最后完成，单击“关闭”按钮时，会弹出一个对话框：“你想保存改变吗？”你和你的母亲都遇到了麻烦。你的母亲看着你问道：“这是什么意思？发生什么事了？”

在现代计算机系统中，用户最难理解的就是文件系统：把程序和数据文件存储到磁盘上的功能。让初学者理解磁盘非常困难。大多数人都不清楚主内存和磁盘存储之间的差别。不幸的是，我们设计软件的方式强迫用户（甚至是你的母亲）理解这种差别。本章提供了设计文件和磁盘交互的不同方式——这种方法与用户的心智模型更加协调。

保存文件改变怎么了

每个程序同时存在于两个位置：内存和硬盘。每个文件都是如此。然而，大多数用户从来都没有领会到内存和磁盘的差异，以及内存和硬盘与她在计算机上完成文档的任务有什么关联。毋庸置疑，文件系统，以及它管理的磁盘存储系统，是引起大多数非计

算机职业人员不满的主要原因。

如图 13-1 所示，当“保存改变?”的对话框打开时，用户抑制着恐惧和困惑，并且出于习惯单击 Yes。始终以同样方式回答的对话框是一个应该消除的冗余对话框。

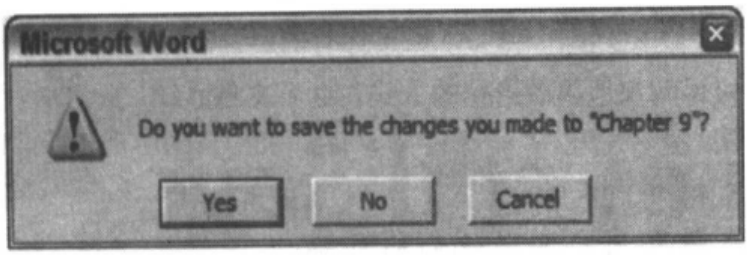


图 13-1 在编辑之后关闭文档时，Word 会问你这个问题。该对话框是程序员将文件系统的实现模型强加到倒霉的用户身上的结果。该对话框的出现对新手用户如此意外，以至于他们常常选择“不”。

保存改变对话框基于一个糟糕的假设：保存和不保存是同样可能的行为。即使 Yes 按钮点击的数量级比 No 按钮更频繁，对话框仍然给予这两个选项同样的权重。正如第 9 章所讨论的，存在令人困惑的可能性和概率情况。用户可能说 No，但用户几乎都会说 Yes。你的母亲会这么想，如果不想要这些改变，为什么要关闭一个包含它们的文档呢？对于她来说，问题是荒谬的。

关于这个对话框，还有一些别的怪异之处。当你完成所有的工作，为什么它只问你是否保存改变呢？为什么它不问你是什么时候做的这些改变？关闭文档和保存改变之间的关系似乎并不自然，即使超级用户们已经非常熟悉它了。

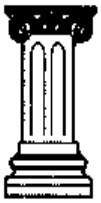
当用户请求关闭或者退出时，程序弹出保存改变对话框，因为这时不得不同步内存里的文档拷贝和磁盘上的文档拷贝。在该功能的技术实现上，将保存改变和关闭退出相关联。但用户看不到这种关联。在现实生活中，当我们离开房间时，不会考虑放弃所有做出的改变。当我们将一本书放回书架时，并不会先删除我们写在空白上的注释。

作为有经验的用户，我们已经学会了使用对话框去达到一些并非出于它们本意的目的。要撤销大量的改变不容易，因此我们在使用保存改变对话框时选择 No 实现该意图。如果发现自己对错误的文件做了大量改变，你使用对话框作为一种逃离方式，将文档返回正确的状态。这很方便，但这是黑客的手法。有一些方法能更好地解决这些问题（如明显的复原功能）。

那么实际的问题是什么呢？现代个人计算机操作系统上的文件系统，如 Windows XP 或 Mac OS X，在技术上都很优秀。你母亲遇到的问题是因为忠实地将优秀的实现模型作为用户界面造成的。

实现模型存在的问题

计算机的文件系统是用户用来管理硬盘数据和程序的工具。这指的是存放大多数信息的硬盘，但也包括软盘，ZIP 盘，CD-ROM 和 DVD，如果你有的话。Mac 上的 Finder 程序和 Windows 的 Explorer 程序极尽辉煌地用图形化的方式表达了文件系统。



公理：磁盘和文件不能帮助用户实现他们的目标。

即使文件系统是一个不应该影响用户的内部功能，但是，由于文件系统对大多数程序界面的影响无处不在，所以它成了一个大问题。交互设计师面对的最困难的问题之一涉及文件系统及其需求。它影响了我们的菜单、对话框、甚至是程序框架。这种影响可能还会继续，除非我们一致协作来改变这种局面。

当前，大多数软件处理文件系统的方式[1]就像操作系统 Shell 处理的方式一样（Explorer, Finder）。这等价于要求你像机械师一样对待你的车子。虽然从交互的角度看，这种方法很不合适，但它是事实上的标准，要改善它存在很多抵制。

关闭和不想要的改变

当犯了错误不知所措时，计算机用户习惯地认为，可以用关闭放弃不想要的改变。这是不正确的，因为拒绝改变的最合适时刻是在做出改变的时候。我们甚至已经有了一个非常好的习惯用法：撤销是删除改变的合适功能。

另存为

当你对保存改变对话框回答 Yes 时，很多程序接着向你显示“另存为”的对话框。一个典型的例子如图 13-2 所示。

---

[1] **译者注** 作者是指软件界面以实现模型的方式表达文件系统。

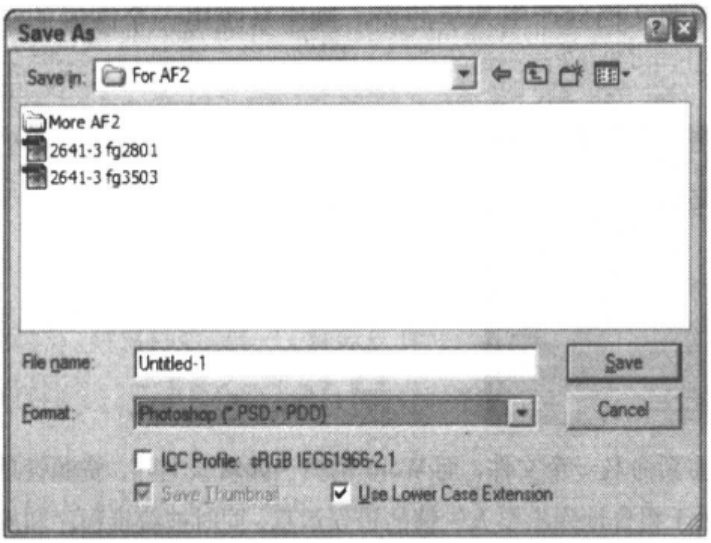


图 13-2 “另存为”对话框提供了两个功能，让你命名文件并且将它放在你选择的文件夹里。然而，用户没有一个保存的概念，所以，对话框的标题不匹配这些功能的心智模型。进一步，如果对话框允许你命名和保存文件，你可能会期望它允许你重新命名和替换文档。不幸的是，我们的期望被糟糕的设计所混淆。

大多数用户不理解手工保存的概念，从他们的角度看，对话框当前的名字没有什么意义。从功能上说，对话框提供了两种功能。他让用户命名文件，并且允许用户选择存放的目录。这些功能都要求了解文件系统的知识。用户必须知道如何构造一个文件名，以及如何在文件目录里导航。很多已经掌握了文件命名的用户，在理解目录树时不得不放弃。他们将文档放在程序默认选择的目录里。所有的文件都存储在同一个目录。偶尔，一些动作会导致程序忘掉默认目录，这些用户不得不求助专家寻找他们的文件。

“另存为”对话框需要确定它的真实意图是什么。如果是命名和保存文件，那么它做得很糟糕。在用户已经命名和保存文件后，它不能改变文件的名字或者目录——至少不能在对话框里改变，而它声称提供命名和保存功能。它也不能使用应用程序的其他工具。实际上，在 Windows XP 里，你可以使用该对话框重新命名其他文件，但不是你当前正在工作的文件。啊？它的意思，我们猜测，是让你重命名该文件先前保存的其他里程碑，因为你不能重新命名当前的文件。但是这两个操作应该都是可能的，也应该允许完成。

新手用户没有那么走运，而有经验的用户知道可以关闭文档，去资源管理器里重新命名文件，再返回应用程序，从文件菜单中调用打开对话框重新打开文档。这个时候你会觉得奇怪，除了上面提到的情况之外，打开对话框并不允许你重新命名或者重新定位文件。

强迫用户去资源管理器重新命名文档比较容易，但这里有一个隐藏的陷阱。问题在于 Windows 很容易支持多个程序同时运行。被这个特性所吸引，用户试图去资源管理器重新命名，而不首先关闭应用程序中的文档。这个非常合理的动作触发了那个陷阱，用户又遇到了新的打击。他被一个粗鲁的如图 13-3 的对话框所拒绝。他没有首先关闭文档，

他怎么知道？试图重新命名一个打开的文档会产生共享冲突，操作系统用一个傲慢的错误消息对话框拒绝了用户。

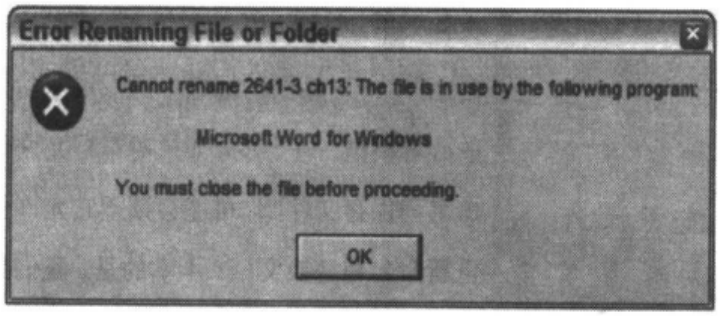


图 13-3 如果用户企图使用资源管理器重新命名一个文件，而 Word 仍然在编辑该文件，资源管理器因为过于愚蠢而无法解决该问题，会过于粗鲁地弹出要人领情的错误消息。同时被编辑程序和操作系统拒绝，这很容易让一个新用户认为文档根本就不能重新命名。

可怜的用户只不过想重新命名他的文档，但发现自己在操作系统神秘的群岛中迷失。讽刺的是，有权威和责任在运行时改变文档名字的程序，而根本就不愿去试一下。

归档

没有显式的文档拷贝或归档功能。用户必须通过“另存为”对话框才能实现该目的，这么做非常易引起混淆。即使存在拷贝的命令，用户也以不同的方式理解。如在处理一个称为 Alpha 的文档时，一些人会想像我们将创建一个称为 Copy of Alpha 的文件，并且以这种方式保存。其他人会想像我们已经放弃了 Alpha，而继续处理 Copy of Alpha。

后面一种情况可能只会发生在那些已经熟悉了文件系统实现模型的有经验用户身上。这就是当前我们使用“另存为”对话框的方式。你已经将文件保存为 Alpha，接着你调用另存为对话框改变文件名字。Alpha 关闭了，并且存储在硬盘上。Copy of Alpha 继续打开，并且进行编辑。从单个文档的视图来看，这种行为没有什么意义，并且为用户设置了一个非常糟糕的陷阱。

下面是可能会导致麻烦的合理脚本提纲。比如说你最近已经花了 20 分钟编辑 Alpha 文档，现在你想将它在硬盘上归档，这样你能够对原来的文档做一些大而实验性的改变。你调用“另存为”对话框，将其保存为 New Alpha。程序将 Alpha 存储在硬盘上并且让你编辑 New Alpha，但 Alpha 并没有保存，所以它被写到磁盘，但没有保存最近 20 分钟做出的改变。这种改变仅仅存在于当前内存中的 New Alpha 的拷贝。当你确信工作已经备份到 Alpha 中，而对 New Alpha 进行剪切和粘贴时，你实际上在修改这一信息的惟一版本。

任何人都知道，你可以使用锤子来敲打螺丝钉，或者钳子来砸钉子。但任何一个有

经验的工匠都知道，错误地使用工具最终会给你带来麻烦。工具坏了，或者工作变得一塌糊涂。“另存为”对话框是拷贝或者管理拷贝的错误工具，最后，用户不得不自己收拾残局。

实现模型与心智模型

文件系统的实现模型和所有用户对于文件系统的心智模型相反。大多数用户将电子文档想像成真实世界中打印的文档，并且完全被这些真实对象的行为特征所迷惑。用最简单的术语说，用户想像所有的文档有两个突出特性：首先，只存在一个文档；其次，这个文档属于他。文件的系统实现模型违反了这两个规则：始终存在文档的两个拷贝；它们同时都属于该程序。

每一个数据文件，文档，甚至应用软件，在由计算机使用时，都同时存在于两个位置：一个在硬盘；一个在内存。然而，用户将他的文档想像成书架上的书。比如，用户偶尔从书架上拿下一本杂志，将某些东西添加在上面。只有一本杂志，或者在书架上，或者在用户的手中。在计算机中，硬盘相当于书架，内存相当于编辑发生的位置，等价于用户的手。但在计算机世界里，杂志没有从书架上拿下。相反，它拷贝了一份，该拷贝驻留在计算机内存中。当用户改变文档时，他实际上是在改变内存中的文档，而原始的拷贝没有改变。当用户完成对文档的修改并且关闭文档时，程序面临两种选择：是用内存中的拷贝取代硬盘上的原始拷贝，还是放弃修改了的拷贝？从程序员的角度看，存在着同样的可能性，两种选择都可以。然而，从用户的角度看，根本就没有什么决定可做，他做出改变，接着放好文档。如果这种情形发生在物理世界的纸质杂志上，用户从书架上拿下书，用铅笔加上一些内容，然后放回书架。而上面的情况仿佛书架突然开口说话，问他是否真的想保留这些改变！

隐藏文件系统的实现模型

说到这里，严肃的程序员马上开始局促不安。他们认为我们践踏神圣的地板：磁盘拷贝是一件很精彩的事情，我们最好不要妄图消除它。请放松！文件系统不存在什么问题。我们只不过需要将它向用户隐藏起来。我们仍然可以在不破坏用户心智模型的情况下，向用户提供磁盘上另一份拷贝的所有好处。

如果打算根据用户的心智模型来展示文件系统，那么我们会得到明显的好处。所有

人都可以教会我们的母亲使用计算机。我们不必回答她所指出的那些无法解释的界面行为问题。我们可以向她展示程序，向她解释程序如何允许她在文档上工作。并且一旦完成，她可以在磁盘上存储文档，仿佛文档是书架上的杂志一样。我们明智的解释不会被“另存为”对话框打断。母亲一类的角色是广大计算机消费者用户的代表。

另一个好处是，交互设计师不必在他们的产品中集成笨拙的文件系统。我们可以根据用户的目标构造程序中的命令，而不是根据操作系统的需要。我们不再需要调用菜单最左边的“**文件**”菜单项。“文件”这种命名方法刺眼地说明了当前的技术是如何侵入到程序界面上的。我们将在本章的后面讨论一些可以替代的方法。

改变**文件**菜单的名字和内容违反了标准，一些虽然不是官方的但是事实上的标准。但改变带来的好处远远超过它可能导致的混乱。作为一个有经验的用户，要习惯新的表达当然需要初始的成本，但这比你所想到的要少得多。这是因为很多超级用户通过学习实现模型已经显示了他们的能力和耐力。对他们来说，学习更好的模型不是什么问题，并且也不存在功能损失问题。对于新用户来说，好处是显而易见的。爬上来以后，计算机专业人员已经忘记了山的高度，但每天新手用户面对计算机世界的珠穆朗玛峰，会深感挫折。我们所做的任何努力，去降低他们必须攀登的山峰的高度，都会带来很大益处，而这一步会制服一些最危险的巅峰。

设计统一的文件表现模型

设计合理的软件总是将文档当做单一实例，而不是磁盘上一份拷贝，内存里一份拷贝：**一个统一的文件模型**（a unified file model）。管理不在内存的信息是文件系统的职责，它通过维护磁盘上的第二个拷贝来实现。这个方法是对的，但正是它的实现细节混淆了用户。应用软件应该和文件系统一起向用户隐藏那些令人不安的细节。

如果文件系统想向用户提示，文件正在由其他程序使用，因此不能改变，把文件名显示为红色，或者挨着它显示一个特殊的符号就够了。新的用户可能会获得一个如图 13-3 这样的错误消息。但至少应该显示一个视觉线索，说明错误突然出现的理由。

不仅在当前的实现模型中，所有数据都有两个拷贝，在程序运行时，所有程序也都有两个拷贝[2]。当用户去任务栏的开始菜单启动字处理软件时，一个对应 Word 的按钮出现在任务栏上。如果他返回任务栏的开始菜单，Word 仍然在那。从用户的角度看，他从工具箱里拿出锤子，没有想到他的工具箱里还有一个锤子。

---

[2] **译者注** 对新手用户而言，存在两个拷贝：一个已经启动，另一个仍然在任务栏的开始菜单中。

这些或许不应该改变，毕竟计算机的力量之--就在于它可以同时运行多个软件拷贝的能力。但软件应该帮助用户理解这种非直觉的行为。例如，开始菜单应该可以对已经运行的程序做些说明。

统一文档管理

对于大多数应用来说，已有的文件管理标准套件包括另存为对话框，保存改变对话框以及打开文件对话框。正如前面已经显示的，整体上来说，这些对话框对于一些任务来说是令人困惑的，对于其他任务，则完全不能胜任。下面是根据用户心智模型管理文档的不同方法。

除了将文档显示为单一实体外，还有用户可能需要的多个目标导向的功能，每一个都应该有它对应的功能：

- 自动保存文档。
- 创建文档的拷贝。
- 创建文档的里程碑/里程碑拷贝。
- 命名和重命名文档。
- 存放和移动文档。
- 指定文档的存储格式。
- 对改变进行复原。
- 放弃所有的改变。

自动保存文档

每个计算机用户都必须学习的重要功能之一是如何保存文档。调用该功能意味着用户接受已经对计算机内存做出的任何改变，并且将它写到文档的硬盘拷贝上。在统一模型中，我们废除了所有用户界面承认的两个拷贝范式，保存功能完全从主流界面中消除。这并不意味着它从程序中消失，相反，它仍然是一个非常必要的操作。

设计技巧：自动保存文档和设置。

程序应该自动保存文档。至少，当用户完成文档，调用关闭功能时，程序应直接继

续将改变写到磁盘，而不需要停下来，调用保存改变对话框来确认请求。

在一个完美的世界里，这就足够了，但是计算机和软件会崩溃，电源可能会断掉，其他不可预测的灾难会凑巧同时发生而丢掉你的工作。如果在你单击“关闭”按钮之前电源断掉了，所有的改变都会丢失，因为包含文档的内存不能工作了。磁盘上的原始数据没有什么问题，但几个小时的工作丢失了。为了防止发生这些情况，程序应该在用户与计算机交互期间定期保存文档。理想的情况下，只要用户做出改变，换句话说，在用户敲击键盘之后，程序都会保存每一点改变。对于大多数程序来说，这是可能的，它们可以在几秒钟之内保存到硬盘中。仅仅对于特定程序，如字处理程序，这种层次的保存会比较困难（但也不是不可能）。

Word 会自动地按照倒计时保存，你可以将这种延迟设置为任意的分钟数。如果你请求每两分钟保存一次，那么在精确的两分钟之后，程序会停止，将你的改变写入磁盘，而不管当时你在干什么。当保存开始时，如果你正在输入，程序会非常现实而令人不安地体现出程序的限制，即强行停滞。这是一种非常令人不愉快的停滞。如果在算法中关注的是用户而不是时钟，这种问题就会消失。没有人会连续地输入。每个人都会停下来整理他的思路，或者翻过一页或者喝一杯咖啡。所有程序需要做的是等待，直到用户停止几秒钟的敲击，这时接着保存。

几分钟一次自动保存，以及在关闭程序时自动保存，对于大多数程序来说足够了。一些人比如说作者，就像妄想狂一样害怕程序崩溃和丢失数据，他们习惯性地在每一个段落后都按下 Ctrl+S，有时甚至在每个句子之后（Ctrl+S 是手工保存功能的键盘快捷键）。所有的程序都应该有手工保存控件，但不应该要求用户调用手工保存。

当前的状况是保存功能明显地放在主要的程序菜单上。当用户请求关闭文档或者关闭/退出程序时，保存对话框被强制施加到文档包含未保存改变的所有用户身上。这些人工制品（指对话框）必须放弃，但手工保存功能能够，也应该像现在一样保留。

创建文档的拷贝

应该有一个叫做快照拷贝(Snapshot copy)的显式功能。单词“**快照**”（snapshot）很明显地说明，该拷贝不同于原始拷贝，而且该拷贝没有与原始拷贝捆绑在一起。这也就是说，对原始的后继拷贝不会对该拷贝有影响。新的拷贝应该自动赋予一个标准形式的名字，如 Copy of Alpha。这里 Alpha 是原始拷贝的名字。如果已经存在一个有这种名字的文档，新的拷贝应该命名为 Alpha 的第二个拷贝。拷贝应该放在原始文档的同一个目录下。

尽管为该命令同时存在的对话框是很诱人的，但不应该打断用户。程序应该悄悄、

有效、聪明地采取行动。而不应强迫用户回答诸如“拷贝一份？”这样的愚蠢问题。在用户心中这是一个简单的命令。如果出现意外，程序应该做出他自己权威的建设性决定。

命名和重命名文档

文档的名字应该显示在应用程序的标题栏上。如果用户决定重新命名文档，可以点击它，并且在合适的地方编辑。还有什么比这更简单更直接的？

存放和移动文档

大多数编辑的文档是已经存在的。它们打开，而不是从头开始创建。这意味着它们在文件系统中的位置已经被确定。尽管我们在创建或者在第一次保存文档时为文档创建主目录，但离开实现模型，这些都没有什么意义。新的文件应该存放在合理的地方，用户能够再一次发现它们。

设计技巧：将文件放在用户能够找到的地方。

如果用户想把文档显式地存放在文件系统层次关系的某个位置，那么他可以从菜单中调用，出现一个类似“另存为”的对话框，对话框中突出显示当前文档。用户能够将文档移动到其他希望的位置。程序自动存放文件，这个对话框仅仅用于移动文件到其他位置。

指定文档的存储格式

图 13-2 给出了另存为对话框上的一个附加功能。对话框底部的组合框允许用户指定文件的物理格式。这个功能不应该位于这个位置。通过将物理格式与保存行为捆绑在一起，用户在保存时遇到了额外而不必要的复杂性。在 Word 中，如果用户无意中改变了文件格式，保存功能和接着的关闭行为都伴随着令人吃惊而且意外的确认对话框。很少发生改变文件物理格式的情况。保存文件是很普遍的情况。这两个功能不应该结合。

从用户的角度看，文档的物理格式——无论它是多信息文本（rich text），ASCII 还是 Word 格式，都是文档的特征，而不是磁盘文件的特征。指定文件的格式不应该和保存文件到磁盘的动作相关。它属于文档属性对话框更合适。

文档的物理格式应该通过从主要菜单中调用小的对话框方式指定。对话框应该将显眼的警告信息放在它的界面上，使用户清晰地了解这些功能可能会引起数据丢失。

对于一些绘图程序来说，用户期望能够将图像文件保存为多种格式，导出对话框（一些画图程序已经支持）对于这种功能比较合适。

对改变进行复原

如果用户无意中做出必须恢复的改变，那么已经有了纠正这种动作的工具：撤销(undo)。文件系统不应该作为撤销的代理来调用。文件系统可以有某种机制来支持这种功能，但这并不意味着它应该用这种术语来向用户展示。直接调用文件系统来撤销改变的概念只不过削弱了撤销的功能。

本章后面描述的里程碑功能显示了能够实现以文件为中心的撤销视图，这样，它们能够在统一的文件模型下工作。

放弃所有的改变

对用户来说，在打开或者创建文档后，决定放弃他所做出的所有改变也不是不常见。所以，应该明确支持这种行为。不用强迫用户理解文件系统来实现他的目标，主菜单上一个简单的**放弃改变**功能就足够了。因为这种功能会丢失数据，因此，应该用清晰的警报标志保护用户。让该功能可以撤销会相对容易实现，也会更加理想。

创建文档的里程碑拷贝

里程碑和拷贝命令非常类似。不同之处在于，里程碑拷贝完成之后由程序管理。用户调用里程碑对话框，该对话框列出了每个里程碑的统计数据，如记录的时间和长度。一次点击，用户就可以选择一个里程碑，同时，他也立即选择了该里程碑作为活跃的文档。当前新的里程碑选择时的文档，它自己也应该成为里程碑，如用 Milestone of Alpha 12/17/03，1：53pm 命名的文档。里程碑实质上是轻量级的版本形式。

新的文件菜单

新的文件菜单看起来就如图 13-4 所示的样子。

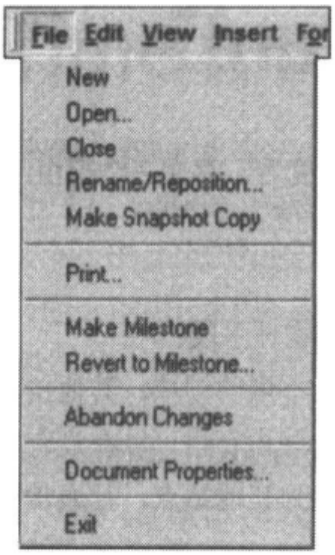


图 13-4 修改后的文件菜单更好地反映了用户的心智模型，而不是程序员的实现模型。仅仅存在一个文件。如果用户愿意，他可以重新命名，放弃它所做出的改变或者改变文件格式。他不再需要理解或者担心 RAM 里的拷贝与磁盘里的拷贝。

New 和 Open 的功能像过去一样，在自动保存改变之后，关闭文档，不会出现对话框或者任何其他令人惊讶的方式。Rename/Move 创建了新的对话框，让用户重新命名当前的文件，或者将它移动到其他目录下。Make Snapshot Copy 创建一个新的文件，即当前文档的拷贝。Print 在一个对话框里收集所有与打印机相关的控件。Make Milestone 类似于拷贝，除非程序通过调用 Revert to Milestone 菜单项来管理这些拷贝。Abandon Changes 放弃自文档打开或创建以来对文档做出的所有改变。Document Properties 打开一个对话框，让用户改变文档的物理格式。Exit 像现在一样关闭文档和应用。

文件菜单的新名字

现在我们展现了统一的存储模型，而不是有关磁盘和 RAM 的分叉实现模型。我们不再需要将应用程序最左端的菜单称为文件菜单——它反映了实现模型，而不是用户模型。有两个合理的替代方案。

可以使用应用处理的文档类型来标志菜单。例如，电子表单应用可能会将它最左端的菜单标志为表单。发票应用可能标志为发票。

另一种选择，可以给左端的菜单一个更为通用的标签，如文档。这对于字处理软件以及绘图应用来说是一个合理的选择。但对于更为专业的程序来说是不合适的。

相反，确实让文件代表磁盘内容的极少数程序——通常是操作系统的 Shell 和实用程序，确定应有一个**文件**菜单，因为它们确实代表文件。

磁盘和文件系统是一个特性

从用户的角度看，磁盘没有存在的理由。从硬件工程师的角度看，存在三个理由：

- 磁盘比内存便宜。
- 一旦写入，硬盘信息不会因停电而丢失。
- 磁盘提供了将信息从一台计算机移动到另一台计算机的物理手段。

第二个理由和第三个理由当然非常有用，但这并不是非磁盘不可的领域。其他技术同样可以，或者更好。有多种 RAM 在停电时不会丢失数据。一些内存也可以在没有电源的情况下保留数据。网络和电话线可以用来物理地将数据传输到其他站点，且通常比可移动的硬盘更容易。

理由一，成本，是硬盘存在的实际理由。不丢失数据的内存比硬盘的成本贵了很多。可靠的宽带网络则更加昂贵。

与 RAM 相比，磁盘有很多缺陷。磁盘比内存慢很多。它们也没有那么可靠，因为依赖于可移动的部件。它们通常消耗更多的能量，并且占用更多的空间。但磁盘更大的问题在于计算机，CPU 不能直接向它们写或者读。它的辅助部件必须在 CPU 能够直接使用之前，将数据读入内存。当 CPU 工作完成，它的辅助部件必须一次性将数据写回硬盘。这意味着涉及到硬盘的处理必然要比普通的 RAM 要慢一个数量级，并且比使用 RAM 更加复杂。

设计技巧：磁盘是黑客手段，而不是设计特性。

使用磁盘的时间和复杂性带来的惩罚如此严重，要不是因为成本问题，我们不会依赖它。磁盘不会让计算机更好，更强大，更快或者更加容易使用。相反，它使计算机更加脆弱，更慢，而且更为复杂。它们是一种折中，对数字计算机体系结构的调整。如果计算机设计者可以使用更加经济的 RAM，而不是磁盘，那么他们会毫不犹豫地这么做。实际上，在最新的手持通信器和 PDA 中，已经使用 Compact Flash，以及类似的晶体内存技术。

无论磁盘技术给我们的软件设计留下了什么样的标志，这么做主要是为了实现上的意图，而不是为了服务用户，或者任何目标导向的设计原理。

改变的时机

只有两个论据支持以文件系统模型来实现应用软件：软件已经以这种方式设计和实现，以及用户习惯于这样。

这两个论据都不合理。第一个是无关紧要，因为以统一的文件模型编写的新程序可以和老的实现模型应用共存。底下的文件系统根本不需要改变。类似于最近几年工具条快速地占领大多数应用的界面一样，统一的文件模型同样可以成功，并且得到用户支持。

第二个论据更加狡猾，因为它的支持者将用户团体作为自己的屏障。更重要的是，如果你问用户，他们会反对这种新的解决方法，因为他们痛恨改变，尤其当改变影响他们花了很长时间才掌握的技术，如文件系统。然而，用户不会始终都是成功设计的最好预测者，尤其当设计与他们所经历的不同时。

在 20 世纪 80 年代，Chrysler 向用户展示了一款完全不同的新汽车设计：小型货车。公众一致指责新的设计。Chrysler 坚持生产了 Caravan，并且使用户相信设计确实很优越。他们是对的，最初抵制这种设计的人们不仅使得这款车成为最畅销的小型货车，而且使得小型货车成为可以自由买卖以来最流行的新汽车原型。

用户不是交互设计师，你不能期望他们理解交互范式改变带来的影响。但市场已经表明，人们很乐意放弃设计糟糕使用痛苦的软件，而追求更容易更好的软件，即使他们并不理解设计原理背后的说明。

第三部分

提供高效能和愉悦

14 设计体贴的软件

15 设计智能的软件

16 改进数据检索

17 改进数据输入

18 为不同的需要进行设计

14

设计体贴的软件

第 9 章曾经提到，斯坦福的两位社会学家，Clifford Nass 和 Byron Reeves 发现，人类好像有一种本能，告诉他们如何与周围有意识的生物交往。一旦任何人工制品表现出足够的交互性——就像一般的软件程序那样——这些本能就会激活。我们对软件的反应是无意识和不可避免的。

这个研究的意义是深远的：如果希望用户喜欢我们的软件，那么我们设计软件的时候，应该让它表现得像一位举止可爱的人。如果希望用户能高生产率地使用我们的软件，那么就应该将它的行为设计成像一个有支持能力的人类同伴。

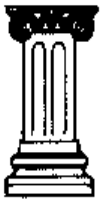
设计体贴的软件

Nass 和 Reeves 认为，软件应该是礼貌的，但是作者更喜欢体贴这个词。尽管礼貌可以解释为一种礼节——说“请”和“谢谢”，但是帮助却并不大——真正的体贴意味着把他人的需求放在第一位。体贴的软件最关心的是用户的目标和需求，其次才是它的基本功能。

如果软件的信息贫乏，过程模糊，迫使用户到处寻找普通的功能，还因为软件自己

的错误而立即责备用户，那么用户不会喜欢这种软件，用户体验也会很不愉快，无论软件多么酷，多么有代表性，视觉隐喻多么好，内容多么充实或者多么拟人化。

另一方面，如果交互是礼貌的，大方的，对人有帮助的，那么用户会喜欢这种软件，使用的时候也会有愉快的体验。同样，无论界面的组成如何，即使是一个绿色屏幕的命令行界面，如果它能做到这些，也会受到欢迎。



公理：软件应该体贴。

什么使软件更体贴

人们赋予体贴的软件许多美好的特征，但它的定义却很模糊。以下列举了一些软件产品（和人）在进行体贴交互时应该具有的特点：

- 体贴的软件对用户感兴趣。
- 体贴的软件是恭顺的。
- 体贴的软件是乐于助人的。
- 体贴的软件具有常识。
- 体贴的软件预见需求。
- 体贴的软件是尽责的。
- 体贴的软件不会因为它自己的问题增加你的负担。
- 体贴的软件会及时通知我们。
- 体贴的软件是有理解能力的。
- 体贴的软件是自信的。
- 体贴的软件不问许多问题。
- 体贴的软件承担责任。
- 体贴的软件知道什么时候调整规则。

我们现在来详细讨论这些特征。

体贴的软件对用户感兴趣

一个体贴的朋友会希望更多地了解你，他记住你的喜恶以便将来可以让你开心。每个人都希望别人能根据自己个性品味来对待自己。

另一方面，大多数软件不知道或不关心是谁在使用它。即使是我们个人拥有的计算机里的个人软件也好像不记得我们，尽管我们一直在反复使用它。

软件应该努力记住我们的工作习惯，特别是所有我们告诉过它的事。对于编写程序的开发人员来说，我们正处在一个信息社会，所以，当程序需要某些特殊信息的时候，只要让用户提供；随后程序将这些信息扔掉，认为一旦需要的话，再次询问就可以了。程序本来就比人更适合记忆，而且作为一个想像中的帮助工具，如果它还忘掉，就更不体贴了。我们将在第 15 章详细讨论软件记忆这个问题。

体贴的软件是恭顺的

一个好的服务提供者以客户为上。她明白她正在服务的人就是老板。当餐馆服务员将我们带到一张桌子面前时，我们会认为他选择的座位只是一种建议，而不是命令。在人不太多的餐馆，如果礼貌地要求换一桌时，我们希望能够得到允许。如果餐馆服务员拒绝了，我们可能就会选择别的餐馆，一家更尊重我们意愿的餐馆。

不体贴的软件对人的活动进行监督和妄加判断。软件有权利认为我们犯了错，但是当它对我们的行为做出判断时不能太过武断。软件可以建议在我们键入自己的电话号码之前不提交输入。它也应该解释一下其中的因果关系，但如果我们希望没有号码的时候也能提交，那么我们期望软件会按我们说的做。（提交这个词和它所代表的概念，和软件应该扮演的恭顺角色相反。软件应该向用户提交，任何预先提供提交按钮的程序都是不礼貌的。注意，网络上几乎所有的事务站点都存在这个问题！）

体贴的软件是乐于助人的

如果向商店服务员询问在哪里可以找到某件商品，我们希望他不仅回答我们的问题，还能义务向我们提供一些有用的间接信息，比如某种更贵质量更好的商品正在以差不多的价格优惠促销。

多数软件不会试图提供相关信息。相反，它只是狭隘地回答我们问它的问题，即使

是与我们的目标明显相关的其他信息，它也不愿提供。当我们告诉文字处理器打印一份文档的时候，它不会告诉我们打印纸不够了，也不会告诉我们前面有四十份文档在排队等待打印，或附近另一台打印机空闲。而一个乐于助人的人会告诉我们这些。

体贴的软件具有常识

在不合适的地方提供不合适的功能是软件产品的一个特点。多数软件产品将经常使用的控件和从不使用的控件放在一起。你很容易地发现：提供简单功能的菜单与不可撤销的专业“弹射座椅控制杆”功能紧邻在一起。这就像你的餐桌座位正靠着打开的烤肉炉。

一些可怕而让人恼怒的故事是，计算机系统反复地发送给用户金额为 0 的支票，或者数字为 957 142 039.58 美元的账单。当这种事件发生的时候，我们认为系统应该提醒收账或付账部门的人，特别是频频发生的时候，但是，多数信息系统却很少具有常识。

体贴的软件预见需求

你的人类助手知道，当你在另一个城市旅行的时候，你将需要一个旅馆房间，甚至在你并没有明确请求的时候。她知道你喜欢哪种房间，你不用提出要求，她就为你预订好了。她能预见你的需求。

在我们仔细阅读网页的时候，Web 浏览器浪费着大量的时间，什么也不做。它可以在我们阅读的时候很容易地预测我们的需求并做好准备。它可以利用这些闲散的时间提前下载所有可见的链接。时机正好，因为我们很快就会请求浏览器检测一个或更多的链接。中断一个不想要的请求很容易，但是，等待请求加载很浪费时间。我们将在第III部分的其余各章进一步讨论更多的方法，让软件利用空闲时间，为我们办事。

体贴的软件是尽责的

一个负责的人会从更长远的角度来认识执行任务的含义。例如，一个负责的人会擦干净柜台，倒空垃圾，而不只是洗刷盘子，因为那些任务与更大的目标相关：清洁厨房。一个负责的人在起草一份报告的时候，还会在报告上加一个漂亮的封面，还会为整个部门影印足够的份数。

这儿有一个例子：如果我们交给 Rodney——我们假想的助手——一个名为 manila 的文件夹，并且告诉他整理好，他检查了文件夹标签上的文字——比如说 MicroBlitz 合同

——然后再在文件柜里寻找。在 M 条目下，让他吃惊的是，那儿也有一个含有同样的 MicroBlitz 合同标签的 manila 文件夹。Rodney 注意到它们之间的差别，并进行研究。他发现先前的文件夹包含一份关于 17 个小部件的合同，这些小部件 4 个月前已经交付到 MicroBlitz。另一方面，新文件夹是有关下个季度生产和交付 32 个齿轮件的合同。有责任心的 Rodney 将老的文件夹改名为“MicroBlitz 小部件合同，7/03”，然后将新文件夹名称改为“MicroBlitz 齿轮合同，11/03”。这种主动性就是我们认为 Rodney 很有责任心的原因。

我们前一个假想的助手 Elliot，完全是个傻瓜。他一点也不负责任，如果他遇到相同的情况，他会不假思索地将新的 MicroBlitz 合同文件夹堆到旧的 MicroBlitz 合同旁边。诚然，他也安全地放好了文件，但他没有做得更好。这也是 Elliot 不再成为我们假想助手的原因。

另一方面，如果我们依靠文字处理软件草拟一份新的齿轮合同，然后想把它保存在 MicroBlitz 目录下，程序提供的选择是要么重写和破坏原先的小部件合同，要么就根本不能保存。程序不仅没有 Rodney 能干，甚至还不如 Elliot。他比傻子还愚蠢。软件是如此之笨，以至于仅仅因为文件的名字相同，它就认为我们的意思是要把老的文件扔掉。

最起码，程序应该将两个文件标上不同的日期，然后保存。即使程序拒绝单方面采取这种极端的行为，它至少应该在保存新文件之前，向我们显示老的文件（让我们重命名那个文件）。程序还可以采取许多更负责任的行为。

体贴的软件不会因为自己的问题而增加你的负担

在服务台前，人们总是希望经纪人对她自己的问题绝口不提，而是充分关注你的问题。也许对于经纪人一方来说有失公平，但是那是服务行业的规则。软件也应该对它自己的问题保持安静，关注我们的问题。因为计算机没有自我或者脆弱的情感，它们应该能胜任这种角色；但是它们总是以相反的方式行事。

软件总是用错误消息向我们抱怨，用确认对话框打断我们，用一些不必要的通知来向我们夸耀（文档成功保存！软件先生，你真好：你有过保存不成功的时候吗？）。我们对程序是否清空垃圾箱的信心问题不感兴趣。我们不想听到不知道该将文件放到磁盘什么地方的抱怨。我们也不需要看到计算机的数据传输率和它的加载顺序。就像我们不需要知道客户服务代表不幸的感情变故一样。软件不仅应该对自己的事情保持安静，而且应该是聪明、自信、自主地解决自己的问题。我们将在 33 和 34 章详细地讨论这些问题。

体贴的软件及时通知我们

虽然我们不希望软件因为它的小恐惧和小成功而不断地纠缠我们，但我们却真的希望它能对关系到我们的事时及时通知。我们不希望本地的酒吧招待向我们抱怨他最近离婚，但我们期待他在显眼的地方标明酒的价格，在黑板上写明何时召开赛前晚会，谁将出场和当前的 Vegas 让球数。获得这些信息的时候，没有人打断我们：无论我们何时需要，它都是显而易见的。同样，软件可以为我们提供大量运行状态的非模态反馈。我们将在第 34 章进行讨论。

体贴的软件有理解能力

现有的多数软件都不是很有理解能力。它对大多数问题的范围理解狭隘。它也许愿意执行艰难的任务，但是只在正确的时间，接收到精确的命令时。例如，如果你请库存查询系统告诉你还有多少小部件，它会忠实地向数据库查询，并报告到你提问时间为止的库存数量。但是，如果二十分钟后，Dallas 办公室的人将所有的库存小部件都清走了，又会怎么样呢?你现在操作就有可能发生尴尬的误解，你的计算机呆在那儿，浪费十亿次的无用指令。它没有理解能力。如果你想再一次知道小部件的数量，那不是一个很好的线索——表明你还会向它了解小部件的数量吗？你可能不希望在你的余生每天都听到有关小部件状态的报告，但是也许你会在本周的剩下几天里想得到有关它们的信息。有理解能力的软件观察用户在做什么，然后利用那些模式来提供相关的信息。

软件也应该观察我们的偏好，并且无需明确的要求就将它们记住。如果我们总是最大化一个应用程序来使用全部的可用屏幕，程序在几次之后就应该知道总是以这种设置启动。对于调色板、默认工具、经常使用的模板设置和其他有用的设置都是同样的道理。

体贴的软件是自信的

软件应该忠于它的信念。如果我们告诉计算机删除一个文件，它不应该问，“你确定吗？ 我们当然确定，否则我们也不会提出请求。不应该在事后劝告我们或者它自己。

另一方面，如果计算机有任何怀疑我们可能发生错误（这也是常事），它也应该指望我们改变主意，等待我们取消删除文件的请求。

曾经多少次，你点击打印按钮后去喝杯咖啡，等你回来的时候却发现一个可怕的对

话框在屏幕中央颤抖，“你真的想打印吗？”这种不安全感简直令人生气，它完全违背了体贴的人类行为。

体贴的软件不问许多问题

正如我们在第 9 章中讨论的，不体贴的软件总是问许多烦人的问题。过多的选择很快就丧失了它的好处，而成为一种痛苦的经验。

选择可以通过不同的方式提供。它们可以以商店橱窗的方式提供。我们可以在有空的时候观看橱窗，考虑，选择，或者不理会所提供的商品——没有问题要问。否则，选择就会像审问一样强加给我们。就像过境时的海关人员：“你有什么东西要申报吗？”我们不知道问题的后果。我们是会被搜身呢，还是不会？软件不应该让用户受到这种胁迫。

体贴的软件失败也不失风度

当你的一个朋友严重失礼的时候，他会试图以后弥补，挽回可以挽回的损失。当程序发现一个致命问题的时候，它可以选择充分利用时间努力弥补过失而不让用户受到损害，或者简单地让系统崩溃。换句话说，它要么像一个有精神病的邮递员一走了之，将大量的工作留给他的同伴和主管，要么整理好它的工作，确保将尽可能多的数据以可恢复的格式保存起来。

多数程序都充满了数据和设置。当它们崩溃的时候，信息通常直接丢弃了。用户只能自认倒霉。比如说，程序正在运行，从服务器下载电子邮件的时候，内部的某个子程序耗尽了内存。和多数桌面软件一样，程序发出一个有效的消息：“你的程序完全瘫痪了”。并且在你点击 OK 之后立即结束。你重新启动程序，或者有时重启计算机，却发现程序丢失了你的电子邮件，当你向服务器查询的时候，又发现它已经删除了你的邮件，因为邮件已经移交给了你的程序。这不是我们所希望的好软件。

在电子邮件的例子中，程序接收来自服务器的电子邮件后，服务器删除了副本——但是却没有确保电子邮件在本地恰当地记录下来。如果邮件程序确保那些信息已经迅速地写到本地磁盘，甚至就是在它通知服务器那些信息已经成功地下载之前，都不会出现这种问题。

即使程序没有崩溃，不体贴的行为也非常普遍，特别是在网上。用户经常需要向网页的一系列表单输入详细信息。在填完十个或十一个字段之后，用户可能按下提交按钮，但由于他的这部分输入有错误或者漏失，站点拒绝他的输入并告诉他纠正。于是用户点

击返回箭头回到页面，可以看到数个有效的输入连同那个无效的输入一起都删除了。还记得那个不可理喻的可恶初中地理老师，Mr. Jones 吗？他把你那篇南美洲的报告全部撕成碎片然后扔掉——只因为你是用铅笔而不是钢笔写的。到今天为止你还厌恶地理吗？Mr. Jones 很可能以前就是个程序员。

体贴的软件知道什么时候调整规则

当手工信息处理系统转化为计算机系统时，在此过程中丢失了一些东西。尽管一个自动订单输入系统可以处理上百万次的订单，比人类员工能处理的要多得多，但人能够以某种被多数自动系统忽略的方式进行工作。在自动系统中几乎没有利用这种功能。

在手工系统中，当员工在销售部的朋友打来电话，解释说加速处理某个订单意味着更多的生意，职员就可以迅速处理那个订单。当另一个订单缺失一些关键信息的时候，职员也可以继续处理它，记住以后去获取和记录这些信息。这种灵活性是自动系统所缺乏的。

多数计算机系统只有两种状态：不存在或者全部一致。不能识别或接受任何中间状态。在任何一个手工系统中，有一个重要的，矛盾的，没有说出，没有记载，但又非常可靠的状态：暂停。一个事务尽管没有全部处理完但也可以接受。人类操作员在他的头脑里、书桌上或者口袋里创建了这种状态。

如数字系统在它保存发票之前需要客户和订单信息。人类职员可以在得到客户的详细信息之前，直接提前登记订单，而计算机系统就会拒绝这个事务，没有详细的客户信息就不允许输入发票。

手工操作系统让人们打破顺序，或者在先决条件满足之前可以执行操作的特点称为规避能力。这是在系统计算机化过程中的损失之一。它的缺失是数字系统缺乏人性的一个关键因素。它是实现模型的自然结果，程序员没有看到任何创建中间状态的理由，因为计算机不需要。但是人类有轻微调整系统的强烈需求。

可规避系统的一个好处是减少错误。通过允许很多小的临时错误进入系统，相信人们在它们造成严重问题之前能够纠正它们，我们可以避免更大、更持久的错误。荒谬的是，多数计算机加强硬性规则来保护这种错误。这些死板的规则使得软件成为人类的敌人，因为人类不能采取规避措施来防止更大的错误，他很快就不会关心对软件的保护措施，也就不能避免真正的大问题。当死板的规则强加在灵活的人类身上时，两面皆输。

从商业上说，阻止人们做他们想做的事总是不好的，而计算机系统也经常不得不消化无效的数据。

在现实世界中，缺失的信息和没有适当填充到标准字段的额外信息都是成功的重要工具。信息处理系统很少处理这种现实世界的数据。它们只能对事务中固定和重复的数据部分，即实际事务的梗概，建模。这些事务可能与大量的会议、旅游和娱乐、配偶和小孩的名字，高尔夫比赛和喜欢的体育明星有关。如果只有终止日期比正式期限延长两星期，某个事务才能完成，这时多数公司宁愿延长最终期限，也不愿看到一个上百万美元的交易化为灰烬。在现实世界中，限制总是可以调整的。体贴的软件需要意识到并且包容这种事实。

体贴的软件承担责任

太多软件采取这种态度："这不是我的责任。"当把工作传递给某些硬件设备的时候，它袖手旁观，任由愚蠢的硬件来完成它。任何用户都可以看出这样的软件不体贴，不负责任。这种软件没有分担责任，来帮助用户变得更加有效。

例如，在一个典型的打印操作中，程序开始发送 20 页报告给打印机，同时打开带有取消按钮的打印过程对话框。如果用户很快意识到自己忘了一个重要的改动，他在打印机刚打出第一页的时候点击取消按钮。程序立即撤销打印操作。但是用户不知道的是，当打印机开始打印第一页的时候，计算机已经把 15 页报告交给了打印机缓存。程序取消的是最后五页，但是打印机对取消操作一无所知；它只知道交给了它 15 页任务，于是它继续打印。同时，程序还自鸣得意地告诉用户打印已经取消。用户可以清楚地看到程序在说谎。

用户对程序与打印机之间的通信问题没有多少同情心。他不关心通信是单向的。他所知道的是，在打印机输出栏里出现第一页纸之前，他已经决定不打印文档，他点击了取消按钮，然后愚蠢的程序继续打印了 15 页，尽管他有足够的时间来终止它。程序甚至接受了用户的取消命令。用户把 15 张废纸扔进垃圾桶的时候，他在抱怨愚蠢的程序。

想像一下，如果程序可以和打印驱动器通信，打印驱动器也可以和打印机通信，用户的体验会怎样呢？如果程序足够聪明，打印工作就可以在第二页纸浪费之前轻易地取消。打印机当然也有取消功能——只是软件懒得用它，因为它的程序员太懒，没有进行连接。

体贴的软件是可能的

我们的软件产品是因为不体贴，而不是因为缺少功能而激怒我们。正如特征列表所示，设计体贴的软件通常不会比设计粗鲁或者不体贴的软件更难。这只是意味着在构想交互的时候，应该模仿一位敏感而关切的朋友。这些特征有与商业计算注重实效的目标是一致的。行为更人性化可能是最实际的目标了。

15

设计智能的软件

因为程序中的每条指令必须通过 CPU 依次执行，为此，我们通常优化我们的代码。程序员尽量最小化指令数量，确保程序的性能。但是我们经常忘记的是，CPU 一完成全部工作，就处于空闲等待状态，直到用户发起另一个命令。我们付出巨大的努力降低计算机的反应时间，但是我们很少甚至没有努力充分利用 CPU 的空闲周期。软件像在军队里一样给 CPU 下指令，交替地命令它行动或者等待。行动是好的，但是等待该换个做法了。

公理：计算机工作，用户思考。

计算机时代的分工是很清晰的：计算机工作，用户思考。计算机科学家用人工智能的幻想引诱我们：计算机能自己思考。但是，用户在思考方面并不真正需要帮助。他们确实需要信息管理方面的帮助，比如寻找信息和组织信息，但是，根据信息做出决策最好还是由人脑（wetware）——我们自己来完成。

关于智能的软件有一些误解。一些天真的观察家认为智能软件确实有智能，但是这个术语的真正含义是，在条件非常困难甚至用户空闲的情况下，这些程序能够努力工作。无论我们梦想的思考型计算机是什么样的，有更好而更直接的方法让计算机更努力地工作。本章讨论一些最重要的方法使软件可以更努力地工作，为人类服务。

利用计算机的空闲周期

在当今的计算机系统中，用户需要记住太多的事情，如文件名和文件在文件系统中的精确位置。如果一个用户想找到季度计划的电子表格，他必须记住文件名字或者进行浏览。在这期间，处理器处于停止状态，浪费了上十亿计的时钟周期。

当前多数软件也不关注场景。例如，当用户在辛苦地赶电子表格的工期时，程序能向他提供的最大帮助最多也就和他在空闲时间玩数字游戏时一样。凭良心说，软件再也不能在用户工作的时候浪费这么多空闲时间了。在日常活动中，我们的计算机应该开始肩负起更多的责任。

浪费的时钟周期

在通常的情况下，多数用户在不足几秒的时间内能做的事情很少，但这足以让典型的桌面计算机执行至少十亿条指令。无疑，这些周期都是空闲的。处理器除了等待，没有做任何事。反对利用这些周期的人总是认为："我们不能做出假设，这些假设可能是错误的"。当前计算机的功能是如此强大，尽管这些说法是对的，但经常没有关系。简单而言，如果程序的假设是错的也没关系，它有足够的空闲能力做出几个假设，而在用户最终做出选择的时候，放弃那些糟糕假设的结果。

在 Windows 和 Mac OS X 的抢占式线程化的多任务系统中，你可以在后台执行额外的工作，而不影响用户当前的任务。程序可以启动查找文件，如果用户开始输入，程序可以放弃，直到下一次空闲时钟周期。终于，用户停下来思考，而程序已经有了足够的时间扫描整个硬盘，用户甚至不会注意到。

程序每次提供一个模态对话框，它就进入空闲等待状态，当用户处理这个对话框时，它不做任何事。实在不应该发生这种情况。对于对话框来说，找到合适的帮助方法不会很困难。用户上次是怎么做的？例如程序可以向用户建议以前的选择。

我们需要以全新的，更主动的方式，来思考软件能够怎样帮助人们实现他们的目标和任务。

更好地利用时钟周期

如果你的程序、网站或者设备能够预测用户下一步要做什么，那么它难道不能提供更好的交互吗？如果你的程序可以预测用户在特定的对话框或表单（form）中的选择，那部分界面难道不可以跳过吗？难道你不认为预测用户会采取什么行动是界面设计卓越的秘密武器吗？

你可以预测你的用户将要做什么。你可以在程序里建立第六感，不可思议地准确地预测用户下一步将要做什么！那些以十亿计的浪费的处理器周期都会得到充分利用：你所需要做的，就是给界面赋予记忆。

给软件赋予记忆

当我们在这种场景下使用记忆（memory）这个术语时，指的不是 RAM[1]，而是指在多次会话中跟踪和响应用户行为的程序工具。如果你的程序简单地记住用户上次做了什么（或者怎么做的），它可以凭此来预测用户下次的行为。正如我们将在本章后面的内容将要见到的那样，你的程序应该记住用户以前的行为，这个简单的原则是设计软件行为最有效的工具之一。

你也许认为为记忆费心是不必要的：每次只要问一下用户会更容易。程序员们飞快地弹出一个对话框来询问任何信息。但是，如我们在第 9 章中讨论的那样，人们不喜欢被问。不断地询问用户不仅是附加工作的一种，而且从心理方面来说，它微妙地表示着对用户权威的质疑。

多数软件是健忘的，每次运行时很少记忆，甚至不记忆任何东西。如果我们的程序很聪明，在使用期间能保留一些信息，但通常这些信息也只是为了使程序员工作更容易，而不是为了用户。程序自动丢弃一些信息，如它的使用方式、改变方式、使用场合、处理的数据内容、用户是谁，程序的不同功能是否用到或者使用的频率如何等等。在这期间，程序用驱动程序名、端口分配和减轻程序员压力的其他细节填充初始化文件。从用户的角度看，也可以使用相同的功能设施来显著地提高软件的智能。

---

[1] **译者注** memory 通常指内存，故作者有这种说法。

任务一致性

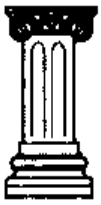
这种记忆真的起作用吗？通过回忆用户上次的行为来预测用户将要做什么是基于任务一致性的原则：我们每天的目标和实现目标的方式（通过任务）通常是相同的。这不仅适用于像刷牙和吃早餐这样的任务，而且对于使用文字处理器、电子邮件程序、手机和电子商务网站这样的任务也适用。

当消费者使用产品时，他使用的功能以及使用方式和上次很可能极为相似。他甚至可能在同一个文档中工作，或者至少相同位置的相同类型的文档上工作。确实他不会每次都做完全相同的事，但是他很可能有少量的重复模式。因为具有明显的可靠性，你可以通过简单的权宜之计——也就是记住用户前几次使用程序的情况来预测他的行为。这样可以让你大幅度减少程序必须向用户提问的次数。

例如，尽管 Sally 使用 Excel 的方式可能和 Kazu 明显不同，但她自己每次使用 Excel 的方式很相似。虽然 Kazu 喜欢 9 磅的 Time Roman 字体，而 Sally 喜欢 12 磅的 Helvetica，但是 Sally 会用 12 磅的 Helvetica 是可靠的规律，并不需要程序去问 Sally 她使用哪种字体？对于 Sally 来说，12 磅的 Helvetica 是一个可靠的起始点，次次如此。

记住选择和默认

确定程序应该记住什么信息有一个简单的原则：如果它值得用户输入，它就值得程序记住。



公理：如果值得用户输入，就值得程序记住。

任何时候，你的程序发现有一个选项，特别是已经向用户提供了该选项时，程序应该每次都记住这些信息。不是使用硬编码的默认值，程序可以使用用户以前的设置作为默认值，这样更可能是用户所希望的；程序应该直接做出与用户上次选择相同的决定，而不是向用户询问；如果错了，可以让用户纠正。无论用户设置的选项是什么，程序都应该记住，这样，一直保留这些选项，直到用户手工将其改变。如果用户跳过程序的这个工具或者将它关闭，它们就不应该再向用户提供了，或者在用户乐意接受它们的时候，仍然能找到它们。

没有记忆能力的程序，最烦人的特点之一是有关文件和磁盘的帮助很少。如果用户只有一个地方需要帮助，那就是文件和磁盘。像 Word 这样的程序记住了用户上次查找文件的地方。遗憾的是，如果用户总是将他的文件放在名为 Letters 的目录下，然后编辑一个文档模板存放在模板目录下，尽管只有一次，之后他的所有书信会存储在模板目录下而不是在书信目录下。所以，程序不仅必须记住文件最后一次访问的位置，而且必须分别记住每种类型的文件上次访问的位置。

窗口的位置也应该记住，如果你上次将文档最大化，那么下次它也应该是最大化的。如果用户将窗口设置为与另一个窗口相邻，那么下次在用户没有给出任何指令的时候，窗口也应该以相同的方式设置。现在，Microsoft Office 应用程序在这方面已经很成功。

记住模式

如果程序有好的记忆能力，用户可以在几个方面受益。记忆减少附加工作，那些用于管理工具而不是完成工作的无用努力。界面附加工作的很大一部分来自向程序解释它应该已经知道的事情。例如在你的文字处理器中，你可能要经常反白文本，在黑色底版上写白字。为了做到这点，你选择一些文本并将字体的颜色改成白色。在没有改变选择的情况下，又将背景颜色设置成黑色。如果引起了程序的足够注意，它会发现你要求的两个格式化步骤中间没有其他的选择。在你看来这是有效的单一操作。如果程序能够在这个独特的模式重复几次的时候，自动创建这种类型的新风格不是很好吗？或者比这更好——创建一个新的反白工具条控件？

多数主流程序允许它们的用户设置默认值，但这并没有记忆有效。就是对超级用户来说，这种设置也是烦琐的过程，许多用户永远不会理解怎样自定制他们喜欢的默认值。

要记住的动作

用户做的每一件事都应该记住。在我们的硬盘驱动器中有充足的空间，对你的程序来说，记忆是对存储空间的很好投资。我们常常认为程序浪费磁盘空间，因为一个大的应用程序可能要占用 30 或者 40 MB 空间。那是程序的典型用法，但是，对于用户数据来说不是这样的。如果你的文字处理程序在每次运行时保存 1KB 的执行笔记，它仍然不会消耗那么多空间。比如，你每个工作日使用文字处理器 10 次，每年大约有 200 个工作日，你每年运行程序 2000 次。净消耗量也仍然只有 2MB，这是对全年的盘点，大概没有超过你桌面上的背景图像所占用的空间。

文件位置

所有打开文件的功能应该记住用户访问文件的位置。对于每个特定程序，多数用户只从极少的几个目录访问文件。程序应该记住这些源目录，在**打开文件**对话的组合框中提供这些目录。用户永远不应该一步一步地通过目录树访问特定的目录。

可推论的信息

软件不应该只是简单地记住这些显式的事实，还应该记住从这些事实可以推论到的有用信息。例如，如果程序记住了文件每次打开时修改的字节数量，那么它能够帮助用户做一些合理的检查。假设文件修改的字节数量的历史数据是 126，94，43，74，81，70，110 和 92。如果某次用户调用这个文件，修改了 100 字节，可能没有意外。但是，如果某次修改字节的数量突然激增到 5000，程序可以认为出了某种错误。虽然用户有可能不小心做了他会后悔的事，但这种可能性非常低，所以用一个**确认对话框**打搅他是不对的。但是，程序有理由在修改 5000 字节之前保留一个重要的副本以防万一。程序可能只需要保留这个副本到用户下次访问这个文件，因为用户一眼就会发现错误，然后他会决定**撤销**操作。

多会话撤销

在用户关闭文档或者程序的时候，多数程序放弃它们的撤销堆栈。这是非常短视的行为。相反，程序可以将撤销堆栈写入文件。当用户重新打开文件的时候，程序可以加载用户上次运行程序时的撤销堆栈，即使那是在一个星期以前。

过去的数据输入

一个有良好记忆能力的程序可以降低用户犯错的次数。原因很简单，用户必须输入的信息大量减少。更多的信息可以通过程序的记忆直接获得。例如，在一个货品计价程序中，如果软件从过去的记忆中得到日期、部门编号和其他标准字段，用户在这些字段犯输入错误的机会就会变得很少。

如果程序记住了用户输入的信息，并且将这些信息用于将来的合理性检查，程序就

可以保留输入的错误数据。设想一个数据输入程序记住了每次输入的邮政编码和城市名称。当用户输入一个熟悉的城市名字和不熟悉的邮政编码时，输入字段可以变成黄色表示匹配的不确定性。当用户输入熟悉的城市名字，而邮政编码却是另外一个城市的时，输入字段可以变成桃红色表示更严重的混淆错误。用户并不一定需要因为这些颜色而采取任何行动，但是如果他需要，就能看到这些警告。

某些 Windows 2000 和 XP 应用程序，特别是 Internet Explorer 有一个相似的特性：命名的数据输入域记住了以前输入的数据，允许用户从组合框里挑选一个值。出于个人的保密考虑，这种特性可以关闭，但对于其他人来说，它既节省时间，又防止错误发生。

程序文件的外部活动

应用程序在调用之间可以启动一个很小的线程。这个小程序密切注视它所处理的文件。它可以追踪文件移动的位置，谁对它进行了读写。这些信息在用户再次运行程序时也许有一定的帮助。当他试图打开一个特殊的文件时，程序可以帮他寻找，即使文件已经移动。程序可以告诉用户在他的文件上执行了什么功能，如是否打印或者传真给某人。的确可能并不需要这种信息，但是，计算机很容易付出这些时间，毕竟那只是所浪费空闲周期的一点点。

赋予应用程序记忆

如果开发人员在软件设计过程中接受任务一致性的作用，那将不同凡响。设计师发现他们的思想呈现全新的面貌。通常在毫无疑问地要弹出对话框的地方会由更多的研究过程取代，设计师提出的问题更加巧妙。比如，程序应该记住多少？应该记住哪些情况？除了上次的设置以外，是否还需要记住其他设置？什么构成了模式的改变？设计师开始想像这样的情况：用户在一行中连续 50 次接受相同的日期格式，然后手工输入一次不同的格式。下次用户输入一个日期，程序应该用哪种格式呢？ 50 次相同的日期格式还是最近的日期格式？一种新格式在成为默认格式之前应该使用多少次？即使存在模棱两可，程序仍然不应该向用户提问。它必须应用自己的主动性做出选择。如果错了，用户可以自由地更改程序的选择。

下面的部分解释了人们进行选择的特征模式，这可以帮助我们解决一些与任务一致性有关的更复杂问题。

缩小决策集合

人们倾向于将无限的决策集合降低到最少的有限决策集合。甚至当你每次做的事并不完全相同时，也常常会从一个重复的较小选项集合中选择你的行为。人们在无意中缩小决策集合，但软件可以记录这些决策集合，并且遵照它行事。

例如，仅仅因为你昨天在 Safeway 购了物，并不意味着你只在 Safeway 购物。但是，当下次你需要日常用品的时候，你可能又到 Safeway 购物。同样，即使你喜欢的中国餐馆菜谱上有 250 道菜，你也经常会从自己偏爱的五六道菜里做出选择。当人们开车上班或回家的时候，也经常是依据交通情况从少量几条熟悉的路径中选择。当然，计算机可以毫不费力地记住这四五个事物。

尽管简单地记住最后一次的动作比不记任何事要好得多，但如果决策集合仅由两个元素组成，它仍可能带来特殊的麻烦。比如你交替地从一个目录里读文件，而在另一个目录保存文件，程序每次提供给你的是上次的目录，那它肯定是错的。解决的方法就是记住多个过去选择。

缩小决策集合使我们想到，程序必须记住的信息应该是用户经常使用的一组信息。和只有一个正确答案的情况不同，有几个选项都是正确的。程序应该寻找更加微妙的线索来区分小的集合中的哪一个是正确的。例如，如果你用支票程序去付账单，程序很快就会知道只有两三个账户是经常使用的。但是它怎样才能知道对于一个特定的账单，哪个是最合适的呢?如果程序记得付款人和每笔账目的数量，就能比较容易地做出选择。每次你付的租金数量是完全相同的！汽车支付也是这样。支付给电气公司的每笔账目可能有变动，变动可能在上次账目的 10%～20%之内。所有这些信息可以帮助程序识别将要做什么，也正是用这些信息帮助用户。

偏好阈值

人们所做的决策一般分为两大类：重要的和不重要的。任何特定的动作都可能与上百次的决策相关，但是，其中只有很少是重要的，其余的都不重要。软件界面可以用偏好阈值来简化用户任务。

在决定购买汽车之后，你不会在意为它筹钱，只要物有所值。在你决定购买日用品之后，选择哪个结账口付款不重要。在决定到 Matterhorn 山间驰骋的时候，你不会在乎乘坐的是哪个雪橇。

偏好阈值证明，不断地向用户询问程序决策细节是不必要的，从而可以在用户界面

设计中给我们指导。用户请求打印之后，不必问他需要多少份，也不必问图像是横向还是纵向。我们可以将这些设置假定为第一次使用的情形，并且为后继的调用记住这些设置。如果用户希望改变，他可以启用打印机选项对话框。

使用偏好阈值，我们可以轻易地跟踪哪些程序功能是用户喜欢调整的，哪些是用户一旦设置好后就不会再改变的。有了这些知识，程序可以在用户想控制的地方提供选择，而不必在用户不感兴趣的时候打搅他。

多数情况下，多数是对的

当然，任务一致性合理但不是绝对地预测用户将会做什么。如果我们的程序依靠该原则，预言的不确定性是很自然的。如果我们能够以80%的准确率预言用户将要做什么，也就是说还有20%的情况是错的。可能有人会认为，这里合适的做法是为用户提供选择，但这意味着用户可能在80%的情况被不必要的对话框打搅。程序应该直接做它认为最恰当的事，然后允许用户覆盖或者撤销，而不是提供选择。如果撤销工具很容易使用和理解，用户就不会受到无谓的干扰。毕竟他十次中只有两次需要使用撤销，取消了八次多余的对话框，对于用户来说这是很划算的。

记忆带来不同

我们的软件常常难以使用的原因是，设计师看似合理而合乎逻辑的假设是错误的。他们认为用户的行为是随机而不可预测的。必须询问用户来确定合适的过程。虽然人类行为不会像数字计算机那样具有确定性，但也很少是随机的。问一些愚蠢的问题会让用户厌烦。

然而，当基于任务一致性原理赋予软件记忆能力的时候，我们会意识到在用户效率和满意度方面的巨大成效。我们都会喜欢聪明、积极主动，有着良好的判断力和敏捷记忆力的助手。有效利用记忆的程序能记住有帮助的信息，和程序每次运行时用户的个人偏好，而不需要询问。它将更像一个积极主动的助手。简单的事情（记忆）可以产生不同的效果：差别就是，用户是忍耐你的产品，还是喜欢你的产品。下次发现程序向你的用户提问时，让它直接问它自己。

16

改进数据检索

在物理世界里，存储和检索之间有着无法分割的联系：将一个东西放在储物架上（保存）也为我们后来找到它提供了方法（检索）。在数字领域里，仅仅把这两个概念之间联系到一起是不够的。只要我们打破传统的思维模式，计算机就能够支持非常复杂的检索技术。本章从交互的观点讨论数据检索的方法，介绍一些更以人为本的方法来解决有用数据的查找问题。

存储和检索系统

存储系统是将货物安全储藏在仓库里的方法。其物理系统由容器和将对象装进容器或者取出容器所需的工具组成。

检索系统是用于在仓库里寻找货物的方法。它是一个逻辑系统，允许根据某些抽象数值，如名字、位置或者某些内容特征将货物定位。

正如我们在第 13 章中所讨论的那样，磁盘和文件通常以实现模型而不是根据用户有关信息如何存储的心智模型来显示，查找存储信息的方法也是这样。这是极其不幸的，因为与机械系统相比，计算机是一种能够提供更好的信息查找方法的工具。在我们讨论

如何改善检索之前，先来简单地了解一下它的工作方式。

物理世界的存储和检索

在家里，如果我们有一本书或者一把锤子，没有必要给它起名字或者存放在一个永久的地方。书可以通过除了名字以外的其他特征标志，如颜色或者形状。然而，收集了我们需要使用和查找的更多条目之后，如果组织得更好，则更有帮助。

一切都各就其位：通过位置存储和检索

对于书和锤子来说，有一个合适的位置是重要的，因为在需要的时候，我们通过位置找到它。我们不能通过吹口哨发现它或者期望它们发现我们。我们必须知道它们在哪，然后去那取。在物理世界里，事物的实际位置意味着找到它们的手段。记住我们把它放在哪——它的位置——对于找到它，以及放好它便于第二次发现都至关重要。例如，当我们想找一个勺子，我们去自己放勺子的地方。我们不会通过勺子的固有特征来找它。类似地，当我们查找书时，我们或者去自己放书的地方，或者猜测它和其他书放在一块。我们不是通过关联来找书。也就是说，我们不会通过它的内容来找书。

在这种模型中，这种方法能在你的家里很好工作，它的存储系统和检索系统是相同的：两者都基于记住位置。它们是双重系统，既存储又检索。

基于索引的检索

系统所有的事物都在它合适的位置上，这听起来很好，但是它也有缺陷：它受到人类记忆水平的限制。尽管它能很好地为你家中存放书本、铁锤和汤匙服务，但是它根本不适用于国会图书馆存放的所有资料。

对于图书馆书架上的书和报纸，我们用另外的工具查找：Dewey 十进制系统（命名源于发明者的名字，美国哲学家和教育学家 John Dewey）。这种思想非常聪明：根据它的主题、标题以及所在书架的数字顺序，给每本书一个惟一的数码。如果你知道号码，就可以轻松地找到书，与它主题相关的其他书也会在附近，这对研究极其有利。惟一遗留的问题是如何确定特定书的数码。当然不会有人会记得每一个数码。

解决的方法是索引，一个允许你通过查找特征条目，如名字来找到条目位置的记录集。传统的图书馆分类卡片提供了三个查找特征：作者，科目和标题。当书本收录进图

书馆系统时，标上了书号，一本书有三类索引卡片，包括所有特征和 Dewey 十进制书号。每个卡片以作者姓名、科目或者标题开头。这些卡片按字母顺序分别归于相应的索引。当你想查找一本书的时候，可以从其中的一类索引中找到它的书号。然后，通过检查标志找到那一行书架，那是书号的范围与你的目标相近。再搜索书架，通过书号的顺序缩小你的视线范围，直到找到你要的那本书。

物理生活中，检索一本书需要通过存储系统，但是，逻辑上找到你要的书只需要通过检索系统。书架和书号是存储系统。卡片索引是检索系统。你用一个系统确定所要的书，用另一个系统索引它。在典型的大学或者专业图书馆，客户是不允许进入书库的。作为一个读者，你通过使用检索系统确定自己想要的书。图书管理员再通过存储系统帮你拿到书。惟一的书号是这两个独立系统之间的桥梁。在现实世界中，检索系统和存储系统都是劳动密集的系统。特别是在过去没有计算机的图书馆，它们都是固定不变的。在过去，如果要加上第四个“基于收录日期”的索引对于图书馆来说将极其困难。

数字世界的存储和检索

和现实世界的书本、仓库和卡片不同的是，在计算机中加一个索引不是难事。具有讽刺意味的是，在终于有可能实现动态关联检索机制的系统中，我们经常不实现任何检索系统。令人惊奇的是，在桌面系统中，我们根本不使用索引。

今天绝大多数计算机系统既没有检索系统，也没有存储系统。如果想找到磁盘上的一个文件，你必须知道它的名字和它的地址。这就好像我们进入图书馆，烧掉卡片目录，然后告诉读者，他们如果能够通过记住印在书脊上的书号，就能轻易地找到他们想要的书。我们将文件检索的负担全部施加到用户的记忆能力上，而同时，CPU 只是悠闲地呆在那里，执行十亿次的 NOP 命令[1]。

尽管桌面计算机可以处理上百种不同的索引，但我们忽略了这种能力，根本就没有指向磁盘存储文件的索引。相反，为了再次找到它们，我们不得不记住文件存放的位置以及文件的名字。这种忽略是现代软件设计中最具破坏性的退步之一。这种失败归因于文件和它们存在的组织系统之间的相互依赖，而在现实世界中，这种相互依赖是不存在的。

---

[1] 译者注　空闲操作，指计算机长时间什么都不干。

检索方法

在计算机上找到一个文档有三种基本方式。你可以记住它在文件目录中的位置，也就是位置检索。你也可以通过记住它的名字来找到它，即标识检索。第三种方法，关联或基于属性的检索，是以文档某些固有的特性为基础来查找文档。如你想找一本有红色封面的书，或一本关于轻轨电车系统的书，或者一本有蒸汽机车图片的书，或者一本提到 Theodore Judah 的书，必须使用的方法就是关联检索。

位置检索和标志检索都是作为存储系统功能使用的方法，在计算机上可以根据名字很好地归类，它们实际上是同一个。关联检索是一种存储系统没有的方法。如果我们的检索系统仅以存储方法作为惟一的基础，那么我们会否认任何基于属性的检索，我们就只能依靠记忆。为了找到需要的信息，用户必须知道他想要的是什么信息以及存放的位置。为了找到计算分期偿还家庭债务的电子表格，用户必须知道他将其保存在 Home 目录下，并且电子表格的名字叫 amort1。如果他忘记了这些事实中的任何一个，那么要想找到文档相当困难。

基于属性的检索系统

对于早期的图形用户界面系统，如最初的 Macintosh，位置检索系统几乎没有意义：桌面隐喻显示了文件的位置（你不必使用索引去查找你桌上的文件），在 144K 的软盘上也不能存储太多的文件。但是，如今的桌面系统可以轻易地存储大小是以前 250，000 倍的文档！但是，我们仍然使用相同的隐喻和检索模型管理我们的数据。我们继续严格按照存储系统的实现模型显示软件的检索系统，而忽略查找文件的系统的效能和易用性，查找文件的系统与保存文件的系统完全不同。

基于属性的检索系统能够让我们根据文档的内容找到它们。例如我们可以找到含有“superelevation”字符串的所有文档。这种检索系统如果真的有效，那么它将找到所有包含该内容的文档位置，所以用户不必说“去查找这样或者那样的目录，发现包含“superelevation”的所有文档”。当然，这个系统必须知道它将要检索的范围，所以它不必在整个因特网上检索包含 “superelevation”的文档，除非我们坚持这么做。

一个设计良好的基于属性的检索系统，还可以让用户按同义词、相关主题或者单个文档的指定属性进行浏览。用户可以动态定义具有重叠属性的文档集。假设一个咨询公司，为每一个潜在用户递交了一份建议书。每份建议书都不同，当然会根据相关客户进行分组。但是，在这些建议书中有一定的关联，因为它们多是为同一个功能服务的：建

议某种商务关系。如果用户可以找到并将所有的计划书收集在一起，同时允许每个计划书保持独立性以及与特定客户的相关性，那会很方便。一个基于位置的文件系统——在它的单一存储位置上——必须借助单个属性，而不是多个特征来存储每个文档。

系统只要留心，就可以知道许多有关文档的信息。如果基于属性的检索系统记住了部分信息，就不必要给用户强加许多负担。例如，程序可以轻易地记住下面这些信息：

- 创建文档的程序。
- 文档类型：文字、数字、表格、图形。
- 上次打开文档的程序。
- 文档是否特别大或特别小。
- 文档是否很长时间没有访问过。
- 文档持续打开的时间。
- 在上次编辑过程中添加或删除的信息数量。
- 文档是否由超过一种类型的程序编辑过。
- 文档是否包含来自其他程序的嵌入式对象。
- 文档是否是从另外的文档中剪切或者拷贝过来的。
- 文档是否经常编辑。
- 文档是否经常浏览，但很少编辑。
- 文档是否打印过以及打印到哪。
- 文档打印的频率，以及在每次打印之前是否做了修改。
- 文档是否传真过以及传给谁了。
- 文档是否通过电子邮件发送，以及发送给谁。

检索系统可以根据这些事实为用户找到文档，而不必用户提前进行任何显式记录。你是否想到其他可以由系统记住的有用属性？

市场上有一种产品为 Windows 提供了这种功能。Enfish 公司销售的一套个人和企业产品，它在你的计算机上动态而隐蔽地创建了信息索引，如果你愿意的话（专业版本），可以跨越局域网（LAN）或者跨越 Web。它可以跟踪文档、书签、内容和电子邮件——摘录一切合理的属性。它还提供强大的归类和过滤功能。它是真正卓越的产品系列。我们将从 Enfish 示例中学到所有的东西。

我们为自己创建的磁盘文件存储系统没有错。惟一的问题在于，我们没有创建足够的磁盘文件检索系统。相反，我们将存储系统交给用户，并称之为检索系统。这就好像是交给他一袋杂货，并说是一顿丰盛的晚餐。没有理由改变我们的文件存储系统，Unix 模型是很好的模型。程序可以轻松地记住它们工作的文件名及位置，所以它们不需要检索系统：需要检索系统的是我们人类用户。

关系数据库与数字汤

使用数据库技术的软件对用户有两个简单的要求：首先，用户必须预先定义数据形式；第二，用户必须遵守该定义。还有两个与人类用户相关的事实：首先，他们从来不会提前知道自己想要什么；其次，即使他们知道，也会经常改变主意。

组织非结构化的事物

我们生活在因特网时代，越来越发现自身经常面对不适用关系数据库要求的信息系统：我们既不能提前定义信息，也不能可靠地坚持我们想像的定义。特别是这两种现象呈指数增长，更证实了这种两难的境地。

第一种现象是电子邮件。数据库中的每个记录有特定的标志，因此属于相同类型的对象表，而电子邮件信息与此不同，不能很好地适应这种范例。我们可以将我们的电子邮件分成接收和发送两类，但这对我们并没有什么帮助。如果你收到来自 Jerry 一封关于 sally 的邮件，提到了 Ajax 计划，以及该计划怎样与董事会上你和 Johns Consulting 的共同报告相关。你可以将它保存在“Jerry”文件夹或者“Sally”文件夹或者“Ajax”文件夹，但是你真正想做的是将它保存在所有的文件夹里。在六个月之内，你可能因为无法预料的原因要找到这条消息，无论原因是什么，你将希望能够找到它。

第二种现象是 Web。就像一个缺乏管理的无限杂乱的过剩硬盘，Web 向结构化提出了挑战。因特网上有大量的可用信息，但是因为它的数量庞大，来源迥异，使得没有哪个规则系统可以强加给它。（我们将看到 W3C 发起的语义 Web 可能有希望成功）。即使 Web 可以组织，方法可能只存在于外部，因为它的内容属于上百万计的个人，而不隶属于任何权威。和数据库中的数据不同，我们不能指望在因特网上的记录中找到可预测的标志信息。

数据库存在的问题

数据库还存在更多的问题：所有数据库记录有单一的预定义的类型，一种记录类型的所有实例被组织在一起。一个记录可能代表一个发票或者一个客户，但是它不能既代表发票又代表客户。同样，记录所在的字段可能是一个名字或者一个社会保险号码，但是它不可能既是一个名字又是一个社会保险号码。这是所有数据库的一个基本概念——它的主要目的是允许我们在存储系统中设置顺序。不幸的是，它不能解决我们的电子邮

件检索问题：来自 Jerry 的电子邮件是“电子邮件”记录类型，这是不够的。我们必须同时可以将它指定为“Jerry”类型、“Sally”型、“Ajax”型、“Jones Consulting”类型和“董事会”类型的记录。我们还必须能够随意添加或者改变它的标志，甚至在记录已经存储之后。再者，一个“Ajax”类型的记录可能是指文档，如一个项目计划，而不是电子邮件信息。因为记录格式是不可预测的，所以，关于 Ajax 记录的标志值本身不能可靠地保存在记录里。这与数据库的工作机制直接矛盾。

数据库确实为我们提供了比匹配简单的记录类型更灵活的检索工具。它也允许我们通过检查内容和匹配搜索条件来查找和提取记录。如我们查找号码为“77329”的提货单或者查找带有标志字符串“Goodyear Tire and Rubber”的客户。但是，这仍然不能解决电子邮件问题。如果我们允许用户向信息记录中输入关键词“Jerry”、“Sally”、“Ajax”、“Jones Consulting”和“董事会”，那么我们必须提前定义这些字段。但是，正如我们前面提到的，提前定义不能保证用户后来会遵守这个定义，例如，他可能正在寻找有关公司野餐的消息。另外，添加一系列的关键词字段会将你引入数据处理的常见困境中：如果给你的用户十个字段，那么有人会想要十一个。

基于属性的替代方案

如果关系数据库技术不对，那么什么是对的呢？如果用户发现很难像数据库要求的那样提前定义信息，是否存在能够工作的替代存储和检索系统呢？

关键是分离存储系统和检索系统。如果将索引作为检索系统使用，那么存储技术仍然只能是数据库。我们可以想像，存储功能是我们存放记录的数字汤（digital soup）。这种数字汤能够接受我们向它倒入的任何记录，无论大小、长短、类型或者内容。无论什么时候输入记录，程序都将返回一个用于检索记录的令牌。我们所要做的是交回令牌，数字汤立即返回我们的记录。但这只是我们的存储系统；还需要一个为我们管理令牌的检索系统。

基于属性的检索系统可以解救我们：我们可以创建一个索引，它存储令牌的副本和键值。然而真正神奇的是，我们能够创建无限数量的索引，每一个代表它自己的键值并包含一个令牌副本。如果我们的数字汤包含我们所有的电子邮件信息，那么我们可以为每个老朋友建立一个索引，“Jerry”、“Sally”、“Ajax”、“Jones Consulting”和“董事会”。现在，当你需要找到关于董事会的电子邮件时，不需要在成打的文件夹中乏味地手工寻找。相反，只要一个查询就会找到需要的所有信息。

当然，这些索引中必须填充某些人或者某些事物，但在交互设计中，这是家常便饭。

有两个组成部分需要考虑。首先，系统必须能够阅读电子邮件消息，和自动摘录和索引恰当的名字，如网址、街道地址、电话号码和其他明显数据的索引信息。其二，系统必须让用户非常容易地添加特殊的消息指针。他可以显式地指定与特定值相关的特定电子邮件信息，无论这个值是否是从消息中按字面引用的。输入是可以的，但是从备选列表中选择，点击拖放以及其他更高级的用户界面习惯用法可以使任务更轻松。

如果存储系统重要性下降，检索系统从存储系统分离出来，且重要性增强，那么将会有很大的好处。某些形式的数字汤可以帮助我们控制日常生活中出现得越来越多的不可预料的信息。我们可以为用户提供功能强大的信息管理工具，而不需要用户提前配置信息或者在将来遵守配置。毕竟他们做不到，所以又何必要坚持呢？

自然语言输出：一种基于属性检索系统的理想界面

在本章的前面，我们已经讨论了基于属性的检索系统的优点。为了真正成功，这种系统需要一个前端，可以允许用户非常容易地理解复杂而相互关联的属性集。

一种替代的方法是使用自然语言处理，用户可以用英语键入他的请求。这种方法存在的问题是，当前普通计算机在多数商务条件下，不可能有效地理解自然语言的提问。在严密控制条件的情况下，它可以在图书馆合理工作，但是不能在充满奇想、方言、口语和误解的现实世界中运行。在任何情况下，自然语言识别引擎的程序超出了普通编程团队所具有的能力和预算。

作者已成功使用的另一个方法是一种称为自然语言输出的技术。使用这种技术，程序为用户提供一系列受限控件，让用户从中选择。控件整队排列，使它们读起来像一个英语句子。用户从有效的可替换的文法中选择，这种设计在本质上是一个自我建档的，有限的查询工具。图 16-1 显示了它的工作方式。

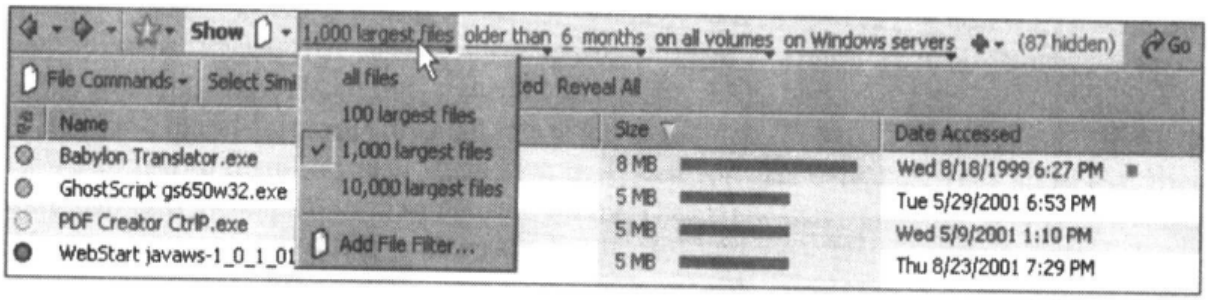


图 16-1 自然语言输出界面的一个例子，具有基于属性的检索工具。它是 Cooper 为 Softek 的 Storage Manager 设计的一部分。对于数据库查询，这些控件产生自然语言输出，而不是接受自然语言输入。当点击每个带下划线的短语时，程序都会提供一个下拉菜单，它带有一个可选择的选项列表。用户从动态的选择系列中构造一个句子，总能保证得到一个有效的结果。

一个自然语言输出界面对于旧的关系数据库查询的表达也是自然的。对于多数人来说，用通常的方式查询数据库很难。因为它需要使用布尔符号和神秘的数据库句法：SQL。我们在第 2 章讨论过布尔符号存在的问题。我们认为，不能因为程序需要理解布尔查询，所以也迫使用户去理解它。

英语不是布尔代数，所以英语句子不需要用 AND 或者 OR 来连接，而是使用像“以下均使用”或者“以下均不使用”的短语。用户发现在这些短语中进行选择是容易的，因为它们非常清楚而且受到限制。当他做出选择之后，他可以像读句子一样检查它的有效性。

从程序角度来说，自然语言输出最具技巧的部分在于，左边控件的选择多数情况下以级联（cascading）的方式改变右边控件选项的内容。这意味着为了有效地实现自然语言输出，必须预先确定选择语法，而且控件必须根据其他控件中选择的内容动态改变或者隐藏。它也意味着控件本身必须能够动态显示，或者至少动态加载数据。

其他相关的问题是本土化。如果你为多种语言设计，词语顺序会有很大的不同（如德语和英语），可能会需要不同的语法设置。

基于属性的检索工具和自然语言输出界面两者都需要巨大的设计和编程努力，但用户在管理数据的效能和灵活性方面会获得极大的好处。因为我们必须管理的数据以指数增长，所以现在无论在管理什么数据，在这些更强大的，目标导向的工具方面进行投资是有意义的。

17

改进数据输入

在第 14 章中，我们讨论了体贴的软件需要在适当的时候能调整规则。软件功能最弱的领域之一是处理数据输入。这是软件开发，尤其是数据库软件开发的结果。本章我们将讨论已有数据输入方式存在的问题，以及一些使数据输入过程更注重人类需求而不是数据库需求的可能方式。

数据完整性与数据免疫

数据输入和数据处理的开发规则很简单：决不让不合法和不清楚的数据进入软件。程序员在用户界面中设立了障碍，这样，坏数据就不会进入系统。这种内部的纯净状态称为数据完整性。

数据完整性的规则将杂乱的信息排除在外部世界里，在任何东西进入计算机之前，都要进行过滤和清洁。软件必须对坏数据保持高度警惕，就好像过境处的海关官员那样（见图 17-1）。所有数据在输入点都要变成有效的。外部的任何事物都是怀疑对象，在经过严格检查允许输入后，它被认为是纯洁的。这样处理的优点在于一旦输入数据库，代码不必不断反复检查数据的有效性和恰当性。

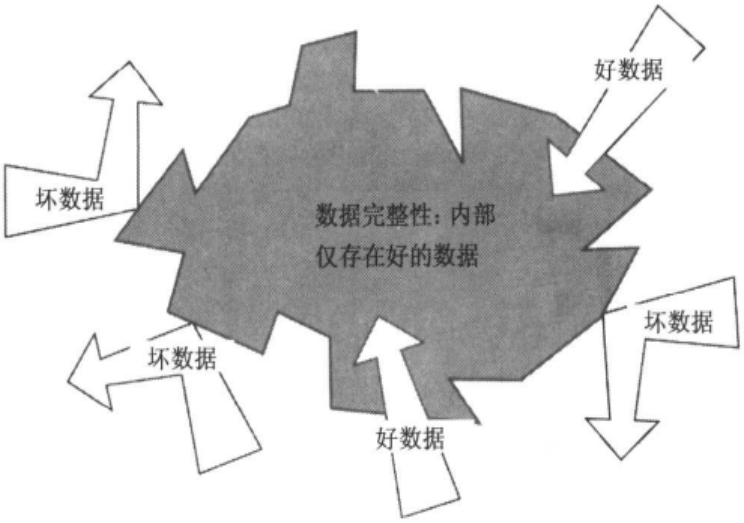


图 17-1 在数据完整性的浮华之下——用认可的数据保护用户和计算机的客观规则——有一个令人烦恼的潜台词：人类恶意地将事情弄糟，一旦有机会，用户可能会输入古怪的垃圾故意摧垮系统。这不是事实。用户可能不小心输入了错误数据，但那与暗示他们故意那样做是截然不同的。用户对这些潜台词很敏感；他们很快意识到程序对他们不信任。数据完整性不仅因不能确定的减轻程序员负担的理由妨碍了系统为用户服务，而且它的高姿态惹怒了用户。它是“需要用户去适应程序，而不是计算机满足用户需要”的又一个例子。

这种方法的缺点很明显：它将数据库的需要置于用户需要之前，用户每次向计算机输入数据都要受到同等的检查。注意，这对于大多数个人软件不是问题：PowerPoint 不知道也不关心你的表达是否正确。但是，你在管理大型企业的时候，无论你是为企业管理系统输入数据的职员，还是在线购买 DVD 的 Web 冲浪人员，你就面临这种类似边境巡逻的检查。

人类，特别是那些每天在工作中需要填写大量表格的人，深知他们得到的数据不是软件所需要的纯洁形式。他们知道信息是不完整的，有时还是错误的。他们甚至知道有时候需要加速处理数据以便让顾客高兴。但是，当遭遇一个对这种事情非常不灵活的系统时，数据处理软件要么必须停下来，要么必须找到某种方法绕过系统以完成数据处理。然而，如果软件意识到人类的存在，并且在界面上体谅他们，那么每个人都会得益。

不犯错误：当软件在输入点检查数据时，它对用户搜身，确认他没有携带任何违禁品进入计算机高度安全的环境，它表达得很清楚。它告诉我们，用户是无关紧要的，程序是全能的——用户是为了程序的方便而工作，而不是程序为用户的方便而工作。这不是我们希望给用户的印象。我们希望用户感觉他们是主管，程序是为他们工作的；用户做决策，而程序工作。

令人高兴的是，有很多方法可以保护软件免受坏数据的侵害。程序员需要让系统具

备对不一致信息和缺陷信息的免疫力，而不是将坏数据置于系统之外。这种方法与写出更智能更复杂的代码有关，它能健壮地处理所有的数据改变，赋予程序数据免疫（data immunity）能力。

数据免疫

为具备数据免疫能力，我们的程序必须训练成能够三思而后行，能够寻求帮助。多数软件盲目地对数字执行算法，而不首先检查数据。程序认为，一个数字字段必须包含一个数字——数据完整性告诉它这些。如果用户输入单词“nine”，而不是数字“9”，程序就会发牢骚，但是，人对这种形式的数据习以为常。如果程序在执行算法之前简单地看看数据，它就会明白。

我们必须训练程序相信，用户输入的是他希望输入的内容，如果用户希望改正，他会那样做，而无需我们狂妄地坚持。但是程序可以寻求帮助。是否有知道如何将字母解释为文本数字的模块？是否存在反映用户意图的修改历史记录？

如果所有这些都没有，程序必须为数据添加一个注释，这样，如果用户来检查问题的时候，他能找到记载了发生的事情和程序采取步骤的准确而完整的记录。

是的，如果用户输入“adsf”代替“9.38”，程序将会得不到满意的结果。但是，立即停止程序来解决问题也不是一个令人满意的过程；输入过程和结果报告同样重要。如果用户界面设计正确，当用户输入“adsf”时，程序就会提供视觉反馈，这样，用户输入上百次坏记录的可能性是极低的。总之，用户只有在程序愚蠢地对待他们的时候，才会愚蠢地行事。

绝大多数情况下，用户输入的不正确数据对于当时的情况来说仍然是合理的。如果程序希望获得一个由两个字母组成的州代码，用户可能偶然输入“TZ”。然而，如果那个相同的用户输入“Dallas”作为城市名，理解这时出现的问题并不需要多少智能。修复缺失的邮政编码对于现代功能强大的计算机来说不算是负担。邮政编码定位程序很少出现错误，而多数人会出现错误。

丢失数据怎么办

显然，信息丢失会违背每个人的初衷。数据输入人员错误键入发票数量，然后又删除了发票，这就带来了真正的问题。但是，程序正当的责任就是停下来指出用户的错误吗？不，不是这样的。你必须考虑情况。如果应用程序是一个桌面生产率程序，用户和

计算机交互，那么他的错误结果很明显。无论如何，用户使用程序就像开汽车，他不会喜欢因为发现雪佛兰的挡风玻璃清洗液已经很少了，就将方向盘上锁。

另一方面，我们假设用户是一个全职的数据输入人员，他在企业数据输入程序中键入表格。我们的员工为了生计做这样一项工作，他花费了上百个——甚至上千个小时——使用程序。他对屏幕上发生的事具有第六感，马上就能知道他是否输入了坏数据，特别是在程序使用微妙的非模态视觉和听觉线索通知他数据状态的情况下。

程序也可以帮助，如必须有效的部分数字不需要键入，而是通过列表视图或者其他受限控件输入。地址和电话号码在能够帮助分析数据的智能字段中输入。程序经常给用户正面反馈，这样程序开始作为伙伴帮助用户了解他的工作状态。数据丢失又会有多严重呢？

在数据输入的情况下，缺失一个字段可能是严重的，但是，这种字段通常是输入得不对而不是遗漏了。程序可以轻易地帮助工作人员检查问题，并且在不停止进度的情况将它变为有效输入。如果工作人员决定省略必要的字段，责任在于工作人员而不是程序。因为缺乏能力或者反社会倾向的职员比例很低。数据输入程序不能仅仅因为百分之一的职员不能胜任工作，就认为所有的数据输入工作职员都不能胜任简单的工作。

我们的信息处理系统多数是能够容忍缺失信息的。一个缺失的名字、密码、数字或者价格几乎总是能够从其他记录数据中重建。如果不行，数据也能够通过询问事务相关的各部分重建。代价是昂贵的，但是，这种代价仍然比技术支持中心的成本低。我们的信息处理系统在缺失数据的情况下也能正常运作。编写这些系统的程序员不喜欢处理缺失数据有关的额外工作，所以他们将数据的完整性作为一条不可违背的神化定律。因此，成千上万的职员必须使用僵化的法西斯式工具以避免数据库崩溃——而不是保证他们的业务不失败。

为防范少数几个人，把你所有的工作人员都当做白痴显然适得其反。它降低每个人的生产率，鼓励快速、昂贵和导致错误的工作方式。它打击士气，增加了想做得更好的职员的无意错误率。它是个自以为是的预言，认为你的信息工人是不值得信任的。

数据输入职员过去陈旧的角色形象正在迅速消失，他们再也不是过去那种没有思想的人，在成堆的纸制表格中机械地敲击键盘，他们再也不是坐在锅炉房和成百的员工一起做同样工作的人。数据输入的任务已经不是大规模生产的工作，正在成为高生产率的工作：一项由有才能，具有专业能力的人员完成的工作。随着电子商务的到来，数据输入直接由客户来操作。换句话说，与数据输入打交道的人越来越不能忍受被当做没有雄心，没受过教育和不聪明的人来对待。用户不能容忍愚蠢的，侮辱他们的软件。直到找到另外的软件销售商向他们展示尊重用户的界面，他们才会按下按钮在网上冲浪。

数据输入和规避能力

当输入系统将坏数据置身于系统之外时，他们几乎从来不允许用户进行规避。没有方法加上边栏评论或给输入字段添加注释。如一个极其必要的数据项缺失，比如说利率。如果没有有效的利率，输入系统就不允许交易，那将阻止公司的生意。如果在借贷程序的利率字段附近，有一个铅笔标注，由银行行长草签“现金支付日的最优惠利率加 3”，又会怎样呢？系统努力做到完美，但最终经受不了现实的考验。

如果一个自动数据处理系统太严格，它就不能模拟现实生活。拒绝现实的系统是没有帮助的，即使所有的字段都有效。在这种情况下，你必须问自己：“数据库和它试图支持的生意哪个更重要？”管理数据库和创建数据输入程序的人只为 CPU 服务。这种利益的明显冲突，只有深知其味但又从开发中分离出来的交互设计可以解决。

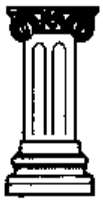
在计算机系统中建立规避能力很困难，因为它需要一个能力更强大的界面。职员不能将一个文档移到队列顶部，除非队列、文档以及文档在队列中的位置都能明显看见。将电子文档从一堆电子文档中取出，并将它放到最顶端的工具也必须功能清晰。规避能力还需要保存暂停记录的工具，而且撤销工具也同样重要。更重要的问题是规避会允许潜在的滥用。

避免滥用的可取方法是计算机有能力轻易地跟踪用户所有动作，为由任何外部的观察者详细做记录。原则很简单：让用户做他们想做的。但是，非常详细地保留动作记录，这样能够做出完整的说明。

审核与校正

许多程序员相信，如果用户在输入数据的时候出错，就有责任通知他。在其他程序出错的时候，程序有责任通知它们是很自然的，但是，这个规则不能延伸到用户。客户永远是对的，所以，无论用户告诉程序什么，它都必须接受，不管程序是知道还是不知道。这是与数据免疫相类似的概念，因为无论用户输入什么，都应该接受，不管程序认为它是多么不正确。

这并不意味着程序可以袖手旁观，说，“好吧，他不想活，就让他淹死。”不能仅仅因为程序必须认为用户总是对的，就认为用户就确实总是对的。人类总是会犯错误，你的用户也不例外。用户的错误不是程序的错，但是，这是程序的责任。你该如何修正它？

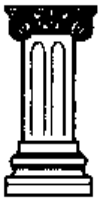


公理：错误可能不是你犯的，但保护用户是你的责任。

程序可以提供警告——只要它们不会愚蠢地停止进度——但是，如果用户选择做一些可疑的事，程序能做的只是接受事实，并努力使用户避免伤害。就像一个忠实的向导，必须跟随他的雇主走进丛林，确保携带一杆枪和足够的子弹。

警告应该使用活动窗口上的非模态技术，通知用户他做了什么，就好像计速器静静地向我们报告超速。然而，用愚蠢的模态停止进度是不合理的。就好像当我们时速超过65英里的时候，计速器切断油门是不对的。例如，校正字段可以突出显示任何用户输入的可疑信息，而不是使用错误信息框。

当用户做了程序确认是错误的事情时，那只有一个方法可以保护他。如果我们校正他的工作而不告知他，他会因为错误的状况而进入丛林，所以我们不能那样做。如果我们校正他的工作，并确保他能够知道，我们可能必须提供错误信息框或者确认对话框，这也是不能接受的。我们惟一的选择就是跟在我们勇敢的用户之后，确保他不会受到伤害。我们跟踪他进入丛林的道路；记住每个动作；确保每个动作都能复原；确保没有间接的信息丢失，以及用户可以领会到我们认为可能存在的问题。本质上说，我们对他的行为保留一份清晰的审核踪迹。因此，存在这样一条公理：审核，而不校正。



公理：审核，而不校正。

如果你不能利用受限控件管理输入的数据，就必须接受用户给你的任何信息。然后记录你得到的和没有得到的，如果有人需要进行整理，你将会有全部的记录。例如，你有数据发生错误的内部记录，用户可以利用这些信息。用户就可以判断是否因为数据的缺失导致了行星偏离轨道。这意味着软件可以追踪是谁，做了什么，在哪，怎么做的，以及什么时候用户做的事情，这样，状况就可以得到修改，纠正或者只是便于日后理解。这种做法比强迫用户以某种武断的格式输入数据要人性得多，因为判断这种武断格式的正确性是看是否与文件模式一致，而不是满足用户需要。

微软Word有一个审核的检查例子，也有很多反例。优秀的例子是处理拼写检查：在你输入的时候，文本下面出现小波浪线，指出字典不能识别拼写和语法的单词（如图

17-2)。右击这些单词会弹出一个可替换的菜单，你可以从中选择，但是，你不一定必须改变，你也不会受对话框或者其他愚蠢的模态干扰。

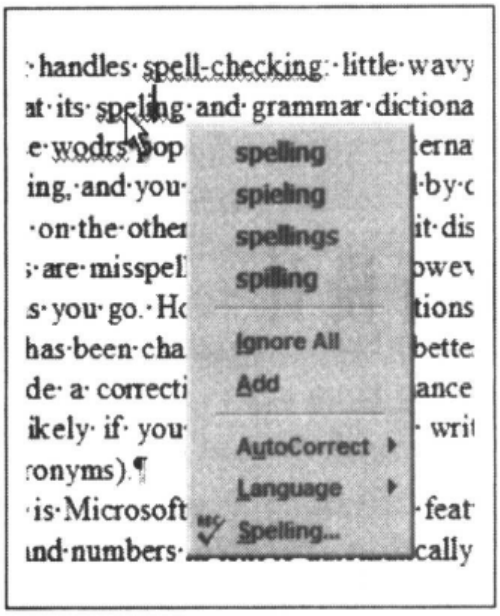


图17-2 微软Word自动拼写检查器检查拼写错误的单词，用下划的红色小波浪线为用户提供非模态反馈。右击带下划线的单词就会打开一个可替换菜单，可以从中选择正确的单词。

另一方面，最初Word的自动纠错功能有一些烦扰。在你打字的时候，它悄悄地将它认为拼写错误的单词改正过来。尽管，这个特征在更正排版时令人难以置信地有用。然而，校正没有审核痕迹，用户不知道他键入的文字已经更改。如果Word能够提供一些标记，来表明已经纠正了的错误，同时还有些错误没有更正，就更好了（比如，你用大量专业术语和缩写词语写技术报告时，这种错误可能会出现得更频繁一些）。

但是，更加令人恐怖的是微软的自动格式化功能。它试图解释用户的行为，如在文本中使用星号和数字来自动地格式化带标号的列表或其他段落格式。当它工作时就像变魔术；但程序经常出错，而且一旦做了又很难取消。自动格式化的问题在于软件有一点自作聪明；它应该留更多的思考机会给人类。幸运的是，这个功能可以关闭。

在现实世界中，人类能够接受部分填充或者不正确填充的文档。我们可以在心中记住，以后再修改，我们通常也是这么做的。如果忘记了，我们也会在最终发现疏忽的时候进行改正。即使我们从来不改正它，有时也能蒙混过关。谁能够说计算机的行为应该与人类不同？程序员就是这种人。他们说，出于对用户的利益考虑，所以拒绝不完整或者不精确的数据，但实际上他们是为自己的利益才这么做的——这样他们不必写那些处理突发事件的难度更大的编码。如果他们试图除零（divide by zero），用户不会死亡，但计算机程序会。

18

为不同的需要进行设计

正如我们在第一部分中讨论的，人物角色（persona）和脚本提纲（scenarios）在基本目标，行为模式和用户需要的确定和设计方面提供了重要的指导。但是，尽管我们为人物角色优化了设计，个人的需要也会随时变化，而且个人之间以及各个用户子群之间的行为也有所差异。这些都是功能真正强大的界面要解决的问题。本章在一个大的设计框架的场景下，探讨处理不同用户不同需求的各种观点和理念。

命令向量和工作集

在根据不同的经验水平将用户的需要进行分类方面，有两个概念特别有用：**命令向量**（Command Vector）和**工作集**（Working Set）。命令向量是允许用户向程序发起指令的特殊技术：直接操作句柄、下拉菜单及弹出菜单、工具条控件和键盘快捷键都是命令向量的例子。

好的用户界面提供多种命令向量。程序中的关键功能是以菜单命令、工具条命令、键盘加速器和直接操作控件形式提供的，它们各自能激活特殊的命令。这种冗余性让不

同技能水平的用户和不同偏好的用户可以根据自己的意愿和能力向程序发出命令。

直接向量和教学向量

直接操作控件，如按钮和工具条控件，是直接向量。在点击按钮和看到功能结果之间没有延迟。直接操作对信息直接产生影响，没有任何中间过程。菜单和对话框都没有这种直接特性，它们都需要中间步骤，有时还不止一个。

一些命令向量为新用户提供更多支持。一般来说，菜单和对话框提供的支持最多，这就是我们之所以把它们叫做教学向量的原因。初学者利用菜单的教学性作为使用新程序的向导，但是，一再重复的中间步骤又常常使他们想去寻找更简洁更直接的向量。

工作集和人物角色

因为每个用户都在无意识地记忆经常使用的命令，持久的中间用户记住了命令和功能的适度子集：工作集。对于不同的个人来说，组成任何用户工作集的命令是独特的，尽管和使用该程序表现出相似使用模式的其他用户相比，它可能和他们的工作集有明显的重叠。例如在 Excel 中，几乎每个用户都会输入公式和标签，指定字体和打印；但是，Sally 的工作集可能包括目标寻找，而 Elliot 的工作集包括链接的电子表单。

严格地说，虽然没有一个标准的工作集可以覆盖所有用户的需求，但是，用户使用模式的研究和建模可以产生一个较小的功能子集，设计师可以确信它是多数用户经常访问的。这个最小工作集可以通过目标导向的设计方法确定：利用脚本提纲来发现你的人物角色所需要的功能。这些需要直接转化成最小工作集的内容。

任何人工作集中的命令都是那些经常使用的命令。用户都希望这些命令能够特别迅速而容易地调用。这意味着，设计师必须至少为程序多数用户的最小工作集提供直接命令向量。

尽管程序的最小工作集几乎都是每个用户完整工作集的一部分，但是，个人喜好和工作需求指定了它需要包括的附加特征。甚至为企业运作而写的定制软件，也可以提供一系列的特征供每个用户挑选。这意味着设计师在为最小工作集提供直接访问的同时，也必须提供将其他命令提升为直接向量的方法。同样，直接命令也需要更多的教学式向量，以确保初学者学会使用界面。这意味着界面中的多数功能必须有多种命令向量。

多种向量规则有一个例外：危险命令（如全部删除，清除撤销，放弃更改等等）不应该有容易的，并行的命令向量。相反，它们需要在菜单和对话框中得到保护（与公理保持一致：隐藏弹射座椅控制杆，见第 9 章）。

让新手用户变为中间用户

Donald Norman（1989）提出了另一个有关命令向量的有用观点。Norman 用词组世界中的信息（information in the world）和头脑中的信息（information in the head）来区分用户访问信息的不同方式。当 Norman 谈到世界中的信息时，他指的是在完成某事时，通过观察环境或者界面可以获得足够信息的情况。例如，公用信息亭张贴的城镇地图就是世界中的信息。我们不必费神去记住 Transamerica Building 的确切位置，因为我们可以通过阅读地图找到它。与之相反的就是你头脑中的信息，指的是你已经学会或者记住的知识，比如胡同口的近道是在任何地图上没有标出的信息。头脑中的信息使用起来比世界中的信息要快捷和容易得多，但是你必须保证已经学过它，而且不会忘记，它也没有过时。世界中的信息使用起来更慢，也更麻烦，但非常可靠。

世界向量和头脑向量

一个教学式向量必须充满世界中的信息，因此它是世界向量。相反，键盘快捷键构成头脑向量，因为使用它需要用户的头脑充满有关功能和等价键的信息。世界向量是初学者需要的，经验丰富的用户在访问高级的或者极少使用的功能时也需要它。头脑向量则由中间用户广泛地使用，甚至更多地由专家用户使用。

例如，当我们首次搬进一个街区，你可能不得不使用地图——世界向量。住了两天以后，你会丢掉地图，因为你已经学会如何回家——头脑向量。另一方面，尽管你已经很熟悉你的房子了，但是，当不得不调整热水器的温度设置时，你可能必须阅读说明书——世界向量——因为搬进来的时候你不必庸人自扰地去记住它们的设置。

我们与软件的关系也是这样。我们找到容易记住的工具和命令，它们是我们经常使用的，而忽略我们很少使用的那些命令细节。这意味着任何经常使用的向量都会自动成为头脑向量的候补者。例如，经常使用之后，我们不再读菜单，而是通过识别模式来找到我们需要的东西：打开第二个菜单，选择倒数第二部分的最末一项。对于人脑来说，模式识别比阅读速度快得多。我们阅读只是为了确认我们的选择。

记忆向量

新用户会为世界向量高兴，但是，随着他们逐渐成为永久的中间用户，他们开始发展自己的工作集，（教学式的）世界向量也开始变得单调乏味。用户喜欢为他们工作集的内容找到更加直接的头脑向量。这是用户自然的和恰当的期望。如果我们的软件希望被

人认为易于使用，那么我们必须满足这种期望。解决方法由两部分组成。首先，我们必须让头脑向量与世界向量并存，其二，我们必须为用户提供让用户学会对应于每个世界向量的头脑向量的途径。这个途径本身就是向量：记忆向量。

有几种方式为用户提供记忆向量。效果最差的方法是只在用户文档中提及向量。稍好一点但仍然没多大用的方法是在程序的在线帮助系统中提及向量。这些方法将找到记忆向量的责任推给用户，用户只能靠自己意识到她首先需要找到记忆向量。

最好的记忆向量就建立在界面上，或者至少在程序界面中以世界向量的方式提供。后者至少可以通过在标准的帮助菜单上添加快捷方式实现。这种方法的好处很明显，因此具有教学作用。新用户可以看到多种命令向量共存，存在易发现的学习资源。所有程序都应该有这种快捷菜单项。

设计技巧：在帮助菜单中提供快捷方式。

实际上，直接将记忆向量集成到主界面存在的问题比想像中要少。在多数程序的菜单中，已经有两个这样的记忆向量。正如微软所定义的那样，一个典型的 Windows 应用程序有两个键盘头脑向量：助记符和快捷键。例如，在微软 Word 中保存的助记符是 Alt+F+S。这种助记符的记忆向量分别通过在菜单标题和菜单项中带下划线的 F 和 S 实现。保存的快捷键是 Ctrl+S。Ctrl+S 在菜单保存项同行右侧清晰地做了注释，这就起到了记忆向量的作用。

这些向量都没有强加给新用户。他甚至没有注意到它们的存在，直到他有机会详细地使用程序——也就是直到他成为中间用户。最后，他会注意到这些视觉提示的存在，也会疑惑它们的含义。多数中等智商的人——多数用户不需要帮助就可以理解与快捷方式的联系。助记符理解起来更难一些，但是，一旦用户得到指点或者偶然将它与 Alt 元键的使用联系起来，这个习惯用法就非常容易记住，并且在出现的任何场合下使用。

如果先看一下第 30 章中的图 30-4，你就会看到一个优秀的技术，小图标为从菜单转到工具条——图标按钮（butcons）提供记忆向量。对应每项功能或设施的图标应该在用户界面涉及到的每个部件上显示出来：每个菜单、每个图标按钮、每个对话框、帮助文本中的每次提及以及每个打印文档中的记载。在界面中形成视觉符号的记忆向量是最有效的技术，但在整个产业中，它仍然是一个未开发的领域。

个性化和配置

用户界面设计师经常面临这样的难题：是否让他们的产品具有用户定制功能。用户按自己的方式做事的需求，和由于熟悉的元素移走或者隐藏而造成的程序导航困难之间，很容易产生矛盾。解决的方法就是从不同的角度看问题。

人们喜欢改变周围的事物，使之适合自己。甚至新手用户，更不用提永久的中间用户，都喜欢在程序上打上自己个性的烙印，将它改变成他们喜欢的外观或行为方式，适合他们的独特口味。人们这样做与用配偶和孩子的照片、植物、喜欢的画、名言和 Dilbert 卡通装饰他们卧室的原因相同。

装饰持久对象——墙壁——在不移动它们的情况下，赋予它们个性。因为悬挂了一幅 M.C.Escher 的海报，可以帮助你从成打同样的走廊中认出它。术语个性化描述的是持久对象的装饰。

个性化使我们工作的地方更可爱，更可亲，使它们更人性和令人愉悦。软件也是这样，给用户提供个性化装饰能力，既有趣，又可以作为有用的导航助手。

另一方面，移动持久对象本身可能妨碍导航。如果设备处人员周末进入你的房间重新布置你的工作室，Dilbert 卡通也不见了，星期一早上要想找到你的办公室可能要颇费周折（持久对象和它们对于导航的重要性在第 11 章中已经讨论过）。

这是明显的矛盾吗？并不是。在持久对象上加上装饰可以帮助导航，而移动持久对象有碍于导航。术语配置概括了移动、添加或者删除持久对象。

配置是更富有经验的用户所期望的。永久的中间用户，在建立了工作集之后，希望能够配置界面，使那些功能可以更容易地找到和使用。他们还会调节程序的速度和易用性，但是，无论如何定制，配置水平应该是适度的。

对于专家用户来说，配置是必需的。传统的导航助手已经不能满足他们的需求，因为他们对产品非常熟悉。专家可能每天都要使用程序数小时；实际上，它可能是完成他们大量工作的主要程序。

移动工具条上的控件是个性化的一种表现形式。但是，工具条最左侧的三个控件在许多程序中对应新建文件，打开文件和保存文件。现在普遍认为它们是持久对象。用户通过移动这些控件来配置他的程序，也就是使工具条个性化的过程。因此，在配置和个性化之间有一个灰色的区域。

改变屏幕对象的颜色是个性化的任务。Windows 在这方面是非常通融的，它允许用户独立地改变窗口界面上的每个组成部分的颜色，包括桌面本身的颜色和模式。Windows 也给了用户改变系统字体的能力。个性化是特殊的模态（见下一小节）；无论人们是喜欢

还是不喜欢，你必须容纳这两种类型的用户。

个性化工具必须简单易用，在用户确定选择之前，给她一个预览的机会。首先它必须容易撤销。让用户改变颜色的对话框应该提供恢复默认设置的功能。

只要程序能够正常工作，多数终端用户不会因不能配置程序而提出抗议。一些真正的专家用户可能会有轻微感觉，但是，如果你的程序能按照他们希望的那样工作，他们会一直使用和欣赏你的程序。

IT 企业的管理者很看重配置功能。它允许他们巧妙地强制企业用户来实行同样的方法。他们欣赏向菜单及工具条添加宏和命令的能力，这样他们在现有的软件基础上能够更加紧密地结合企业建立的过程、工具和标准。许多 IT 管理者购买软件的基本决策就是要买那些有配置能力的软件。如果他们在购买一万，两万份程序，他们肯定认为程序应该能够适应他们特殊的工作风格。因此，微软办公应用程序获得最易配置套装软件的头衔绝对不是偶然的。

特殊的模态行为

很多时候用户测试显示，在一个习惯用法的有效性上，用户相对均等地分为两部分。一半用户明显喜欢一种习惯用法，而另一半更喜欢另一种。人们的喜好明显地分为两个或者更多分组的情况表明人们的喜好是特殊模态的。

对于这个问题，开发组织在情感上也分为两派。一组成为菜单项阵营，而其余的开发人员则是图标按钮阵营。他们就两种方法哪个相对要好一些争论不休，事实上，问题的真正答案一开始就摆在那里：两种方法都要用。

当用户对习惯用法的喜好存在分歧的时候，软件设计时就必须提供两种习惯用法。两组人都应该感到满意。使一部分人感到满意而激怒另外一部分人是没有好处的，无论你或者你的开发人员属于哪一组。

Windows 就如何在菜单实现方面迎合这种特殊模态需求方面提供了一个很好的例子。一些人喜欢起初的 Macintosh 菜单的工作方式。你点击菜单栏的菜单项会使菜单出现；然后仍然按住鼠标按钮沿菜单下拉方向向下拖动，在你选择的选项上释放鼠标。另外一些人认为这个操作规程很难，他们更喜欢不需要在拖动的同时按住鼠标的操作方式。Windows 巧妙地满足了这种需求。它通过让用户在菜单栏的菜单项上点击和释放来使菜单出现。然后用户可以移动鼠标——按钮处于释放状态——到他选择的菜单项。再次点击和释放鼠标来选择菜单项和关闭菜单。用户仍然还可以通过点击和拖动来选择菜单项。这些习惯用法的聪明之处在于彼此可以和平共处。任何用户可以自由地混合使用这两种

习惯用法，也可以坚持使用其中的一种。程序不需要改变。没有最喜欢和最佳选择的设置；它只是工作。

自 Windows 95 开始，微软向菜单行为中加入了第三种特殊形式的模态：用户点击及释放和以前一样，但现在他可以沿着菜单栏拖动鼠标，其他菜单依次激活。令人惊异的是，所有三种习惯用法现在能完美地融合在一起。现在，Mac 也全部支持这三种习惯用法。

本土化和全球化

设计用于不同的语言和文化的程序对于设计师来说存在某些特殊的挑战。但在这里考虑命令向量又可以提供指导。

直接向量，如直接操作和工具条图标按钮是符合习惯的，是视觉的而不是文本的。因此它们能够相当容易地全球化。当然，设计师做大量准备工作来确保这些习惯用法所选择的颜色和符号在不同的文化中没有特殊含义很重要。那些特殊含义是设计师不想要的（例如在日本，复选框里的 X 会被理解为取消选择而是进行选择）。但是总的来说，非隐喻性的习惯用法对于全球化的界面应该是相当安全的。

教学性向量，如菜单项、字段标签、工具提示和说明提示和语言相关，因此必须本土化，翻译为恰当的语言。创建本土化的界面应该谨记以下问题：

- 在某些语言中，单词和词组通常要比其他语言中的单词词组要长（例如，德文标签平均来说要比英文长得多）。
- 某些语种中的单词，特别是亚洲语种，可能很难根据字母顺序分类。
- 年月日的顺序以及是使用 12 小时制还是 24 小时制在不同国家是不同的。
- 数字和货币小数点的表达是不同的（一些国家使用句号和逗号的方式与美国相反）。
- 一些国家以星期计数（例如，第 50 周是十二月中旬），一些国家使用的历法不是西洋新历（Gregorian calendar，1583 年罗马教皇格列高里十三世为改正西洋旧历而制定的历法）。

在翻译菜单项和对话框的时候，需要全盘考虑。确保翻译的界面保持整体的一致性很重要。真空状态下直观翻译的菜单项和标签与其他独立翻译的项组合在一起的时候，可能会造成混乱。界面的语义应该在抽象层次和细节层次得到维护。

库和模板

不是所有使用程序创建文档的用户都能够完全从白手起家建立文档。但是，多数程序为用户提供原子工具，这相当于锤子、锯子和凿子。对于某些用户来说，这就很好了，但是，其他用户需要更多的工具：可供他们打磨和油漆的未完工桌子或椅子。

假设一个程序让你使用因特网上的信息配置个性化的报纸。一些用户可能喜欢将体育新闻放到首页。但多数用户可能会希望以更传统的视角，将世界新闻放在首页，而体育新闻放在后面。甚至这些更传统的用户会喜欢这样的事实：他们可以添加本地新闻和与个人特殊兴趣相关的主题新闻。他们会挑选预先做好的一张报纸，然后稍微修改来达到他们自制版本的需要。对于我们这些几乎不懂新闻的人来说，在白板的基础上创建一份完整的报纸不是个令人愉快的任务。

换而言之，如果用户没有白手起家的需要或者期望的话，任何程序都应该允许用户从候选的设计库中选择一个初始设计或者文档结构。

设计技巧：为用户提供好的解决方案模板库。

某些程序已经提供了预先设计的模板库，但应该有更多的程序这样做。白板使多数人感到害怕，如果用户不想要模板，那么他就不必那样做。提供基本设计库是个很好的解决方法。

第四部分

应用视觉设计原则

19 外观设计

20 隐喻、习惯用法和启示

19

外观设计

在后 Macintosh 时代，人们普遍接受的看法是图形用户界面，或者说 GUI，优于字符界面。然而，虽然某些 GUI 程序外观精美，使用简便，令人惊叹，但多数的 GUI 程序尽管是图形化的，却仍使我们困扰。看来很容易创建出和 UNIX 命令行应用程序一样难以使用的图形用户界面程序。为什么会这样呢？

为了找到这个问题的答案，我们需要更好地理解视觉设计在创建用户界面中的作用。

视觉艺术和视觉设计

视觉艺术和视觉设计人员都采用相同的视觉媒体。他们都必须熟练掌握这种媒体，然后就没什么共同点了。艺术家的目标在于创造一个能够激发美学反应的可见人工制品。因而，艺术是艺术家，有时还是整个社会，在所关心的情感或理性主题上的一种自我表达方法。艺术家所受的束缚越少，创造的独特作品会越多，艺术价值也就越高。

另一方面，设计师的作品必须满足他人的需要，不是他自己的需要；而与此相反，同时代的艺术家关注的只是意识和情感的表达。正如 Kevin Mullet 和 Darrell Sano 在他们的著作“Designing Visual Interfaces”（1995）所指出的：“视觉设计师关心的是寻找最合

适的表现方式，来交流一些的特殊信息。”那么，视觉界面设计师关心的是寻找最适合的表现方式，来表达他们所设计软件的行为。

图形设计和视觉界面设计

用户界面设计并不完全排除对美感的关注，但应将这些关注放到功能框架的约束条件下。因此，根据设计中的界面范围，在**界面上下文**（interface context）中的视觉设计需要几种相关的技能。在界面上工作的任何设计师都必须理解下面几个基本元素：颜色、**版式**（typography）、**形式**（form）和**组合**（composition）。另一方面，也必须对交互以及软件行为有所了解。虽然对于一份真正成功的交互设计而言，这两种类型的视觉洞察力都是必需的，但很难找到这些技能均衡发展的视觉设计师。

图形设计和用户界面

直到最近的 20 年，图形设计这门学科还只是用于包装、广告和文档设计的印刷媒体所统治。旧式教育培养出的图形设计师无法自如地处理数字媒体设计，他们不习惯在像素级处理图像。而在像素级处理图像又是处理多数界面设计问题所必需的。但是，新培养出的一代图形设计师在数字媒体方面得到了训练，能相当成功地将图形设计的概念应用到这种新的，像素化的媒体中去。

典型的图形设计师对视觉原则有着很深刻的理解，而对于随时间变化的软件行为和交互等概念的理解相对薄弱。通晓数字技术的天才图形设计师擅长于提供丰富类型、清晰、视觉一致、审美愉悦而令人兴奋的界面，我们在 Windows XP，Mac OS X 及一些视觉上更成熟的计算机游戏界面和面向消费者的应用程序中见过。这些设计师擅长制造漂亮而合适的界面外观，并负责将公司品牌融入到软件外观中。对于他们来说，设计首先是信息的清晰性和可读性，然后才是品质、风格和传达了品牌信息的框架。最后才通过启示（affordance）（参见第 20 章）表达行为。

视觉界面设计和视觉信息设计

视觉界面设计师需要有图形设计师相似的技能，但他们更注重设计的组织问题，以及用启示向用户表达行为的方式。虽然图形设计师更擅长定义视觉设计的语法——看上去像什么——而视觉界面设计师则更了解交互原则。比如说，他们注重如何使界面的视

觉结构与用户和程序行为的逻辑结构相匹配。视觉界面设计师也关心向用户表达程序的状态，关心用户对各项功能理解（布局、格线、图形-背影问题，等等）的认知问题。

视觉信息设计师履行的职责与内容和导航相关，而不是更多的交互功能。他们的作用在 Web 设计中特别重要。因为 Web 设计中，内容常比功能更重要。他们的焦点在于通过使用视觉语言来控制信息的层次关系。正如视觉界面设计师的工作紧密围绕交互设计一样，视觉信息设计师的工作则紧密围绕信息结构。

工业设计

虽然在任何深度上讨论工业设计问题都超过了本书的范围，但随着交互式电器和手持设备越来越普遍，工业设计在创造新的交互产品中起到越来越重要的作用。正如在图形设计师、视觉界面师和信息设计师之间存在技能上的差异一样，各种层次的工业设计师之间也有类似区别。其中一些人更擅长为目标对象设计吸引人的合适形状和外壳，另外一些人的天赋在于能够设计符合人体工程学，并且富有逻辑性的物理控件，它们能够很好地匹配用户行为，并且表达设备的行为。当越来越多的物理人工制品基于软件，并采用复杂而综合的视觉显示，交互设计师、工业设计师和视觉设计师一起紧密合作，开发可用的产品变得越来越重要。

可视界面设计原则

人的大脑是一个超级模式处理计算机，能理解我们无论何时何地看到的这些高度密集的视觉信息。我们的大脑通过辨别视觉模式，并且对我们所见到的事物建立优先级系统来处理这些杂乱的输入信息。大脑的视觉系统具有在视觉提示（visual cue）的基础上把视野内的部分装配成模式的能力，视觉提示是我们之所以能够如此快速和有效地处理视觉信息的原因。视觉界面设计必须利用我们与生俱来的视觉处理能力，帮助程序向用户表达它们的行为和功能。

用这么一点点篇幅来公正地评价视觉界面设计这个课题实在是太少了，但有一些重要的原则，能帮你将视觉界面设计得在使用时尽可能容易和令人愉快。Kevin Mullet 和 Darrell Sano（1995）对这些原则进行了极好的详细分析。在这里，我们只是对视觉界面设计的一些重要的概念总结一下。

视觉界面应该：

- 避免视觉噪声（visual noise）和杂乱（clutter）。

- 使用对比（contrast）、相似性（similarity）与分层（layering）来区分和组织元素。
- 在每一个组织层次上提供视觉结构和流。
- 使用紧凑、一致而上下文合适的图像。
- 风格一致，功能全面而有目的性。

我们在下面各节分别对这些原理进行详细的讨论。

避免视觉噪音和杂乱

界面的视觉噪音是由多余的视觉元素造成的，它们分散用户的注意力，使我们不能把注意力集中到直接表达软件功能和行为的视觉元素上。想像一下，在极其拥挤、吵闹的饭馆里对话的情景，如果环境太嘈杂，要想进行有效的交流是不可能的。对于用户界面来说，道理也一样。视觉噪音可表现为多种形式，如过分修饰不必要的空间元素、过分使用规则和其他视觉元素来分割控件，在不同的控件间留白不够，不恰当或者过多地使用颜色、纹理（texture）和版式。

杂乱的界面试图在有限的空间里提供过多功能，但实际上却造成了控件之间视觉上相互干扰。视觉上过分修饰呈现巴洛克（baroque）风格，缺乏条理或者屏幕上显得过分拥挤都会给用户带来认知压力，妨碍用户导航时的速度和精确度。

总之，界面，特别是非娱乐性的界面，应该使用简单的几何形状，最少量的轮廓（contours）和最少的饱和色。一个界面的版式变化不宜太大。比如说，不同字号的字体一种或两种就够了。当多种相似的设计元素（控件、窗格和窗口）有相似或相关的逻辑意图时，它们在视觉属性，如字形、字号、颜色、纹理、权重、方向、空间和对齐方面也应近似。要突出的元素需要在视觉上与其他元素形成鲜明对比。

好的视觉界面像任何好的视觉设计一样，在视觉方面是高效的。它们充分利用最少的视觉和功能元素。图形设计师普遍采用的一项技术是试着去掉单个元素，来测试它对清晰表达信息的作用。

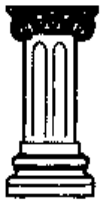
飞行员兼诗人 Antoine de Saint Exupery 曾说过："完美不是没有任何东西可加，而是没有任何东西可减。"创作界面时，你必须不断地在视觉方面将其简化。一个视觉元素越能在完成更多工作的同时表达仍然清晰，就越好。正如阿尔伯特·爱因斯坦认为的那样，事情要尽量简单，而不是简单一些。

另一个相关的概念就是杠杆作用（leverage），在界面中使用多个元素来表达多重相关的意图。列表中，用来表达对象类型的视觉符号就是很好的例子，点击视觉符号，可以打开该类对象的属性对话框。界面可以包括一个发起属性显示的单独控件，但是更经

济和符合逻辑的解决方法是将它与类型标记（type marker）结合在一起。总之，是交互设计师，而不是视觉设计师，最适合给视觉元素分配多重功能。这需要深刻地洞察具体上下文中的用户行为，还有软件的行为和编程问题。

使用对比和分层来区分和组织元素

对比界面中的元素满足了两种需要。首先，将界面中主动的，可操作的界面元素与被动的，不可操作的界面元素进行对比。其次，将主动元素的不同逻辑集合进行对比，以更好地表达它们的不同功能。应避免无意识地，或者模棱两可地使用对比，因为用户对几乎相同的结果会感到迷惑不解。正确地使用对比会产生用户能够记住和识别的视觉模式，使他们更快找到方向。对比也提供了显示界面视觉层次关系中元素地位的手段。换句话说，对比是功能和行为的表达工具。



公理：可视界面基于视觉模式。

【维度对比、色调对比和空间对比】

界面的可操作控件应该在视觉上与非操作区域区分。用伪三维图像给人一种手动启示（manual affordance）的感觉（有关启示的详细讨论见第 20 章），也许是对控件进行对比最有效的形式。如赋予可点击和可拖动的按钮和其他项目突出的外观，而数据输入域（date entry area）如文本域凹进的外观。这些技术提供了维度对比。

除了启示的维度以外，色调、饱和度、值（亮度）都能进行调整，来区分控件和背景，或者将控件在逻辑上组织在一起。多数情况下，使用这些色调对比应该围绕单一的“轴”——色调、饱和度或者亮度，而不是所有元素同时使用。我们必须知道，单纯使用色调对比有风险，因为对颜色有感知缺陷的人无法获取这类信息；而饱和度和亮度是更安全的选择。在单色显示中，色调对比是设计师惟一的选择，或控件的色调对比，或控件背景的色调对比，或两者兼有，可能都是合适的，要根据具体场景而定。

空间对比是逻辑区分控件和数据输入域的另一种方式。空间上将相关元素放在一起，有助于用户了解哪些任务是互相关联的。好的位置分组应该考虑任务和子任务的顺序，以及眼睛看屏幕的方式（多数西方国家是从左至右，从上至下）这些因素。在以后的各小节我们还会继续讨论。形状也是一个重要的对比方式。如复选框是方的，单选框是圆

的。这种设计绝不是偶然。另一类空间对比利用方向：上、下、左、右或是某个角度。Mac 和 Windows 中的图标提供了精细的方向提示。文档图标是垂直的，文件夹的图标是水平的，而应用程序图标至少在原来的 Mac 中有对角线。大小对比也是有用的，特别是在显示定量信息时，它们很容易对比。信息设计将在本章稍后进行讨论。在考虑标题和标签的相对大小以及界面网格模块区的相对大小时，大小对比也非常有用。在这些情况下，大小与范围的广度、重要性、使用频度密切相关。和颜色对比一样，使用空间对比最好在这些变量中选择一个作为“轴”。

【分层】

根据每个元素的视觉提示或主动元素所在的背景进行分层也是组织界面的一种方式。几种视觉属性控制了对分层的理解。颜色影响对分层的理解：深色、冷色调、不饱和色层次下降，相反，亮色、暖色、饱和色层次上升。大小也影响分层：大的元素层次高，而小的元素层次低。位置上相互重叠的元素也许是视觉分层最直接的例子。

为了对元素进行有效分层，你必须使用最少量的对比来维持屏幕同层内各条目间的紧密相似性。在你确定好分组，以及如何在视觉上最好地传达组信息后，再根据背景的重要性，调节组与组之间的对比来决定显示时的重要性。层间的差异要最大化，而同层各条目间的差异要最小化。

【图形和背景】

人类理解视觉模式的负面影响在于图形（用户应该关注的视觉元素）和背景（图形出现的后台环境）之间的冲突。人们习惯于把亮的对象当做图形，而暗的对象当做背景。一个成功的图形和背景设计必须完整地结合：图形元素的位置错误和比例失调可能最终强调的是背景。一个图形和背景结合良好的设计有着比例和视觉权重均衡的图形和背景，图形位于背景的中央。

【斜视检测】

检验视觉界面设计所使用的对比是否有效的一种方法是图形设计师们提到的斜视检测。闭上一只眼睛，另一只眼睛斜视屏幕。看看哪些元素突出，哪些模糊，哪些项好像成一组，是图形还是背景占优势。其他的测试方法还有在镜子里看设计（镜面检测）和倒置检测来发现设计的不平衡。变换你的观察角度经常能发现以前未觉察到的布局和组合问题。

在每个组织层次提供视觉结构和流

你的界面最有可能由成组的视觉元素和行为元素组成，这些元素组合在一起形成窗格，然后依次构成屏幕或页面。如前所述，可以依据位置（或接近关系）、对齐、颜色（亮度、色调、色温、饱和度）、纹理、大小或者形状进行分组。在独占式应用程序中，可能存在几个层次的结构，所以，保持一个清晰的视觉结构对于你的用户按工作流程从界面的一部分轻松地导航到另一部分至关重要。剩下的章节描述帮助实现明快的视觉结构的多种重要特征。

【对齐、网格和用户的逻辑路径】

设计师帮助用户有组织而系统地熟悉产品的一个重要方法就是对齐视觉元素。成组的元素须在水平和竖直方向对齐（如图 19-1 所示）。设计师必须特别注意：

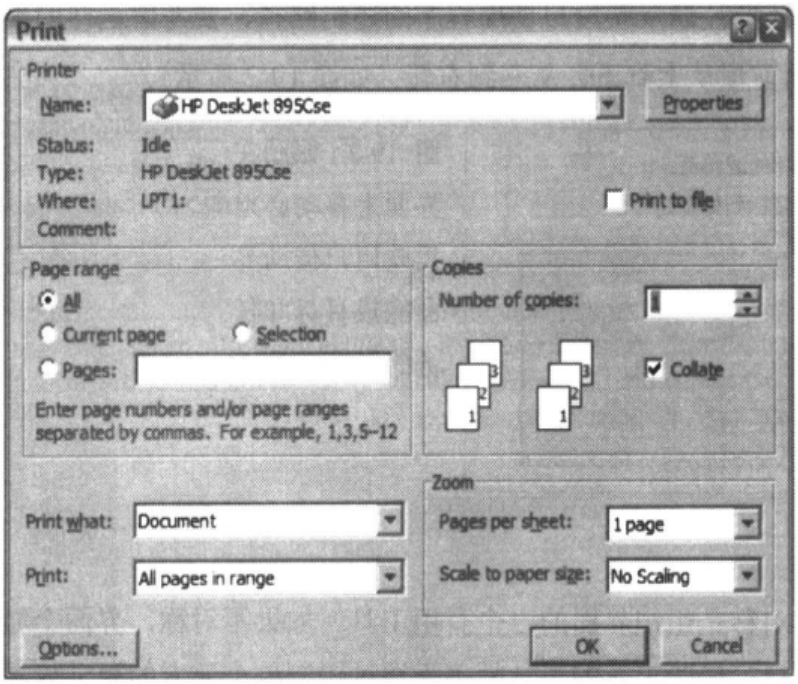


图 19-1 Microsoft Word 的打印对话框是一个很好的例子。它依赖网格将各元素在竖直方向和水平方向精确对齐。注意相关功能元素周围的分组框控制项（group box）。通过使用分组框控制项可以避免界面视觉混乱。打印对话框有着清晰的边界线。通过适当的平衡组内元素之间的空白，常不需要一个明确的方框指出分组情况。

- 对齐标签。控件的标签竖直排列时要彼此对齐；虽然右对齐在视觉上显得更清爽，但在输入表单大小相同的情况下，左对齐使用户浏览起来更方便。（否则看

起来就像棵圣诞树，左右参差不齐。）

- 在控件集合内对齐。相关的复选框、单选框或文本字段组都应该按照规则的网格对齐。
- 控件之间对齐。对齐的控件（如前所述）与其他对齐的控件成组时也应该遵循相同的网格。
- 大规模的元素组、窗格，以及屏幕和较小的控件组一样都应该遵循规则的网格结构。

在定义具有多个视觉或功能复杂层次的界面时，网格结构是极其重要的。在交互设计师为应用程序和它的元素定义了整体框架后，视觉界面设计师将在网格结构上帮助调整布局，恰当地强调高层元素和结构，同时又给更低层或相对不重要的控件留有一定的余地。对于网格来说，重要的是越简单越好。如果原子网格单位太小，网格将太复杂而无法辨认。模糊和复杂都是良好设计的大敌。清晰而简单的网格将有助消除模糊。

在遵循网格的同时，布局还要注意反映用户使用程序的逻辑路径，要考虑这样的事实：（在西方国家）人们的眼睛习惯从上到下，从左到右转（如图 19-2 所示）。

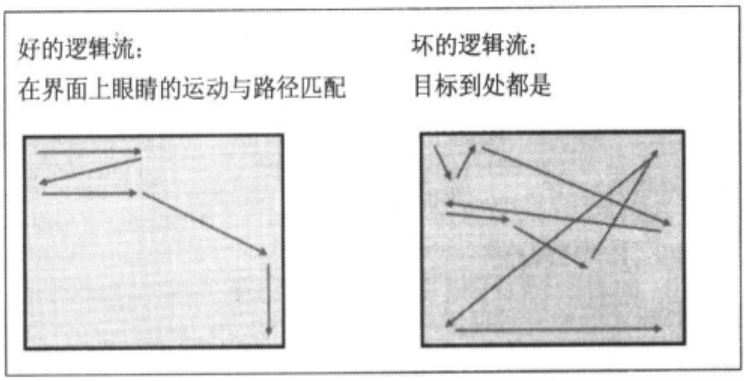


图 19-2 眼睛在界面上移动必须反映用户借助界面完成目标和任务的逻辑路径。

【对称和平衡】

从视觉平衡的观点来说，对称是组织界面的一个有用工具。如果不对称，界面会显得失衡，好像摇摇欲坠倒向一边。经验丰富的设计师善于通过控制单个元素的视觉权重来获得不对称的平衡，就像你能在跷跷板上平衡不同重量的孩子一样。在用户界面的上下文中要得到一个不对称的设计不容易，因为屏幕大小的限制，很难有空白。斜视测验、镜面检测和倒置检测在观察布局是否倾向一方时也有用。

界面中常用到两种对称：垂直轴对称（沿一根垂直线对称，此线常位于一组元素的中央）或对角线对称（沿对角线对称）。多数典型的对话框呈两种对称中的一种，对角线对称最常见（如图 19-3 所示）。

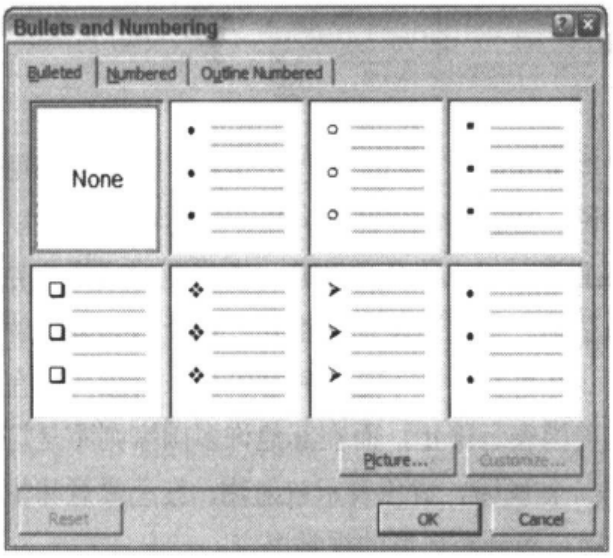


图 19-3 Microsoft Word 的项目与符号对话框呈对角线对称。对称轴从左下角至右上角。

独占式应用程序不会在顶层使用对称（它们一般通过良好设计的网格获得平衡）。但在一个设计得很好的独占式界面中，各元素在某种程度上也会使用对称（如图 19-4 所示）。

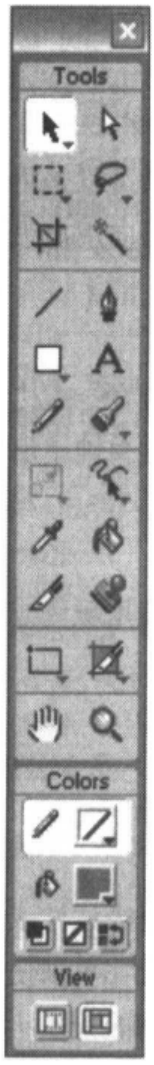


图 19-4 Macromeida Fireworks 4 工具板中的垂直对称

【空间协调与空白】

所谓空间协调，也就是把界面（或至少每一屏）看作一个整体。设计师发现，对于人的眼睛来说，某种比例与其他比例相比看上去会更舒服。最著名的有黄金分割率，这个比率好像是古希腊人发现的，后来由 Leonardo Da Vinci 命名。不幸的是，目前，多数计算机显示器的比例是 1.33:1，给设计师设计整屏应用和独占式应用程序的布局造成不利因素（见第 8 章）。尽管如此，这种对比例的理解仍然在设计舒服的用户界面布局方面非常重要。

界面功能区的恰当尺寸有助于空间协调，元素之间和元素组周围适当的空白也是一样。就像一部设计得很好的书，有合适的页边空白，段落、图形、标题之间有恰当的距离一样。为了使设计的界面不至于给人拥挤不堪的感觉，这样的视觉关注很重要。特别是在设计独占式应用程序时，寻找合适的比例是很重要的，因为用户每次要待几个小时。你不希望用户在每次使用你的产品或服务时感到不舒服。这关键取决于你的布局。堆得像个正方形不好，像两个正方形也不好，要让你的比例醒目、明快而准确。

使用紧凑、一致和上下文适宜的图像

使用图标和其他说明元素可以帮助用户理解界面，但如果做得很糟糕，可能会使用户感到恼怒、迷惑甚至侮辱。对于设计师来说，理解“程序需要向用户传达什么？”以及“用户认为应该传达什么？”这两个问题很重要。对人物角色及其心智模型的理解为界面中使用文字语言和视觉语言奠定了坚实的基础。文化问题也很重要。设计师必须了解颜色、手势、符号在不同文化中的不同含义（红色在中国不是一种警告色，拇指向上在土耳其是一种严重的侮辱性手势，八角形在美国表示停止，而在其他许多国家则不是这样）。设计师还必须了解特殊领域的颜色编码。在医院，黄色表示放射物，红色常意味生命受到威胁。在设计之前先确认是否理解了用户领域和环境的视觉语言。

视觉元素也应该是紧凑而通用的视觉语言的一部分。这意味着相似的元素应该共享视觉属性，如它们所处的位置、大小、线条粗细和整体风格。仅仅对比那些重要信息以区分含义。设计理念就是要创建结合在一起的元素，形成紧凑的、整体的系统。设计达到这点才是完美组合，而没有一点拖泥带水。

【面向功能的图标】

设计代表功能或对象操作的图标带来了一些有趣的挑战。最重要的一个挑战就是用图标的视觉语言代表抽象的概念。在这种情况下，最好依赖习惯用法，而不是向用户强加无法理解的具体表示方法，并且考虑增加工具提示和文本标签。

对于一些更显而易见的具体功能，适用以下指导准则：

- 将动作和动作施加的对象同时表达有助于增进理解。名词和动词结合在一起比单独用动词更易被人理解（例如剪切命令,用一篇文档上加一个 X，比单纯使用剪刀这种隐喻图像更便于理解）。
- 小心使用与你的目标用户想像中不同的隐喻和表现方式。
- 在视觉上将相关功能分组，以提供空间上的上下文，如果不行，就使用颜色或其他的视觉主题提供上下文。
- 保持图标简单，避免过分的视觉细节。
- 尽可能重用，这样用户只需要学习一次。

【对象与视觉符号之间的联想】

在界面中为对象类型创建独特的符号有利于用户认知。这些符号并不总是表达性或隐喻的——它们常是习惯用法（习惯用法的力量详见第 20 章）。这些视觉标记比单独使用文本标签更能帮助用户快速地导航到恰当的对象。为了建立符号和对象之间的关系，

在屏幕上任何表示对象的地方使用符号。

公理：在视觉上区分不同行为的元素。

设计师必须注意让视觉上不同的符号代表不同的对象类型。在一个布满相似图标的屏幕上辨认某个特定图标与在一堆文字里辨认某个字一样困难。还有一点非常重要：在视觉上区分（对比）行为不同的对象尤其重要，包括各种控件的变体，如按钮、滑动块和复选框。

【图标和视觉符号的绘制】

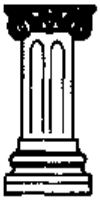
特别是随着彩色屏幕图形能力的增强，人们试图增加细节，在绘制图标和画面时，达到和照片差不多的质量。但是，这种趋势最终并没有为用户的目标服务，特别是对于生产率应用。图标必须保持简单和示意性，使用最少量的颜色和阴影，保持适当大小。最近，Windows XP 和 Mac OS X 都接近采用完全绘制的图标（OS X 更甚，用 128×128 的像素制造近乎照片的图标）。虽然这种图标很好看，但不恰当地将注意力引向了它们本身，而且以小尺寸绘制时效果不佳。这意味着，要使它们易于辨认，必须占据额外的屏幕空间。另外，它们也使界面不紧凑，因为仅有一小部分功能（多与硬件相关）能够用这样具体的写实照片风格的图像来代表。照片图标就像全部大写的文字；图标之间的差异不明显，容易混淆，它的错综复杂会让我们迷失。Mac OS X Aqua 界面充满写实照片的按钮最终分散人的注意力（见图 19-5）。用户对这些服务并不满意。

图 19-5 Mac OS X 中的**写实照片风格**的图标。这些图标的惟一作用就是分散人们对数据和功能控件的注意力。另外，即使在某些场合，详细地绘制人们熟悉的对象是有意义的，但是，绘制人们不熟悉的对象或抽象概念（如网络）又有什么意义呢？有多少用户见过裸露的硬盘驱动器是什么样子（最右边）？用户最终还是依靠附加的文字来理解图标，除非它们经常用到。

【行为的可视化】

不要只用文字描述界面功能的结果（或者更糟，一点说明也没有），要用视觉元素向

用户展示结果是什么。不要将这与控件启示上使用的图标相混淆。而是，除了用文本表示设置和状态，还绘制说明性的图片或图表传达行为。虽然可视化常消耗更多的空间，但它清晰表达的能力值得占用像素。近来，微软已经发现了这样的事实，例如，在 Windows Word 对话框中除了使用文本控件外，也开始可视化地表达它们的含义。Photoshop 和其他图像处理软件早就用缩略图预览视觉处理操作的结果。



公理：可视化地传达功能与行为。

Word 的页面设置对话框提供了标记为预览的图像。这是一个只输出控件，在对话框当前页边距设置下，显示页面视图的缩影。多数用户很难想像左边距 1.2 英寸会是什么样子，预览控件可以实现这些。如果允许预览控件输入而不仅是输出，微软可以做得更好。拖动图片的左边距，观察对应微调控制项中的数值上下变化。

相关的文本字段仍然重要——你不能用可视化域取代它们。文本表示的是设置的精确值，而视觉控件精确地描绘创建页面的外观。

全面而有目的地结合风格和功能

当设计师选择在界面上应用某种风格的元素时，必须从全局角度出发。界面的每个方面都必须从视图的风格考虑，而不是单个控件或其他视觉元素。你不希望界面像涂满油彩的外衣一样。相反，你必须确认程序视觉界面设计的功能特征与产品的视觉品牌（visual brand）协调。你的程序行为是品牌的一部分，用户对产品的体验反映的是形式、内容和行为的适宜平衡。

【形式与功能】

对于许多设计师来说，视觉风格具有一定的诱惑力，但在一个界面中，应该小心地控制视觉元素的风格——特别是在设计独占式应用时。设计师在调整到某种视觉风格的过程中必须小心，不要影响控件的基本形状、视觉行为和控件的视觉启示（见第 20 章）。要点是理解每一个元素的价值。只要能体现你的意图，且不干扰界面的意图或用户与它交互的能力，为元素添加风格就没有错。

这意味着，教育和娱乐性的应用程序，特别是为儿童设计的那些应用，可以去尝试更多的风格。界面和内容的视觉体验是享受这些应用程序的一部分，对控件和内容之间

的主题关系也可能有更多的看法，不过，即使在这样的情况下，也应该保留基本的启示，使用户能真正地容易理解内容。

【品牌和用户界面】

多数成功的公司为建立品牌资产而不惜投入巨资。企业培育出牢固的品牌资产就能为产品博得价格上的优势，也促进了客户的忠诚度。品牌显示出产品的正面特征，意味着与其他产品的区别，也体现了用户的品味。

从最基本的意义上说，品牌的价值在于用户与特定企业交互的总和。这种不断增加的交互通过基于技术的渠道，因此，现在比以往任何时候都更强调用户界面品牌就不足为怪了。如果目标是一致的正面顾客交互，那么语言、视觉和行为的品牌信息就必须一致。

尽管企业已经考虑到品牌内涵，因为它关系到传统的市场销售和未来的交流渠道，但许多企业才刚刚开始从用户界面方面对待品牌问题。为了在用户界面的场景下理解品牌，从下面两个角度考虑会有所帮助：第一印象和长期关系。

正如人际关系，用户界面的第一印象极其重要。第一个五分钟的体验是建立长期关系的基础。为确保第一个五分钟体验成功，界面必须清晰直接地表达品牌。一般来说，视觉设计，通过颜色和图像在达成第一印象方面扮演最重要的角色之一。为你的用户界面选择支持品牌的合适的颜色调色板和图像风格，是为品牌正面建立第一印象的有效手段。

人们形成第一印象后，将开始评估界面行为是否与它的外观一致。通过在第一印象中许下的承诺，来建立品牌资产和长期的客户关系。而交互设计和行为控制，则常常是遵守视觉品牌对用户所许承诺的最好方式。

可视信息设计的原则

和可视界面设计一样，可视信息设计领域也有一些对设计师们有用的原则。信息设计领袖 Edward Tufte 断言，好的视觉设计是“思路清晰则可见”，是通过对浏览者的“认知任务”（目标）及一系列设计原则的理解来达到的。

Tufte 认为，在信息设计中存在两个重要问题：

（1）很难在一个二维表面显示多维信息（多于两个变量的信息）。

（2）显示分辨率不够高，不足以显示高密度信息。计算机提出了一个特别的挑战——虽然它们能够添加动画和交互，但它的信息分布密度比纸张低。

交互和可视界面设计师不能逃避二维显示屏的限制，也不能克服低分辨率显示存在

的问题。但一些通用的设计原则——与语言、文化、时代无关——有助于让信息显示的效率最大化，无论是在纸上还是在数字媒体上。

在他完成的著作 the Visual Display of Quantitative Information（1983）中，Tufte 介绍了七大原则。在下面几节中，将讨论与数字界面及内容相关的部分。

据 Tufte 所说，可视化的显示信息必须：

（1）加强视觉对比。

（2）显示因果关系。

（3）显示多个变量。

（4）在一帧显示中结合文本、图形和数据。

（5）确保内容的品质、相关性和完整性。

（6）在相邻空间上显示事物，而不按时间堆积。

（7）可量化的数据就要量化。

我们将对基于软件的媒体信息设计简单地讨论这些原则。

加强视觉对比

应该为用户提供一些方法在脚本提纲前后对比相关的变量、趋势、前后脚本提纲。对比使信息更有价值，用户更易理解。和其他许多图形工具一样，Adobe Photoshop 经常使用预览，使用户轻易地获得交互前后的对比（见图 19-6、图 19-7 和图 19-8）。

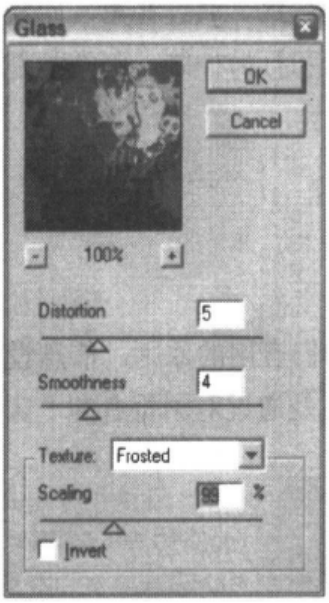


图 19-6　视觉对比。Photoshop 滤镜中提供的交互预览允许用户对操作的结果与操作执行前进行比较。某些 Photoshop 滤镜除了前面对话框中显示的缩略图以外，还允许（用户可以即时点击开关）对整个图像全景预览或对选择部分预览。

显示因果关系

在信息图形中阐明原因与结果。Tufte 在书中举了航天飞机失事的经典例子。如果 NASA 科学家准备的图表条理清晰地表明了着陆点的空气温度与 O-ring 失效的严重性之间的关系，那么悲剧也许可以避免。交互界面中应该使用非模态的视觉反馈（见 34 章）告诉用户他们行为的可能结果，或提供如何完成操作的暗示。

显示多个变量

在不影响清晰度的情况下，提供多个相关变量信息的数据应该同时显示。在交互式显示中，用户可选择开启或关闭变量，使对比更容易，相关性（因果关系）也更清晰。图 19-7 是交互式显示的例子，它允许处理多个变量。

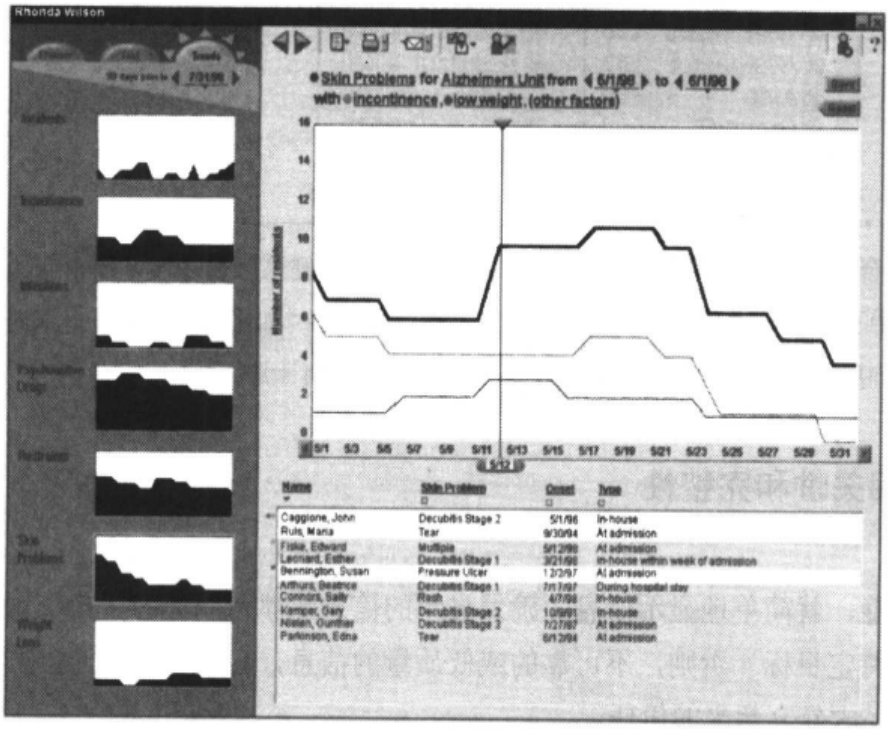


图 19-7 显示多个变量。这是为共享保健系统设计的 Orcas 长期监护系统。它允许护理人员在一帧满屏显示的情况下比较趋势，并且进行相关性分析。长期保健机构的护理人员需要主动了解保健质量。 Cooper 的设计师创建了一个趋势分析工具。护理人员可以任意选择保健问题进行跟踪，并且对其他潜在问题进行相关性分析。左侧的图形作为实时的缩略图显示主要趋势；这些属性和其他属性可以加入到右侧的图表中。滑动的垂直游标确定了图表下方区域中显示的日期，它显示相关居民某日发生的所有事件。点击居民的名字，将向护理人员显示该居民的图表。

在一帧显示中结合文本、图形和数据

对于部分用户来说，因为需要更多认知处理，需要单独的主键或图例才能解释的图像效率较低。阅读和理解图例是另一种和导航有关的附加工作。用户不得不在图表和图例之间来回移动，然后再将两者在脑海中结合起来。图 19-8 是结合了文本、图形、数据，以及输入和输出的交互例子，对用户来说这是高效率的结合。

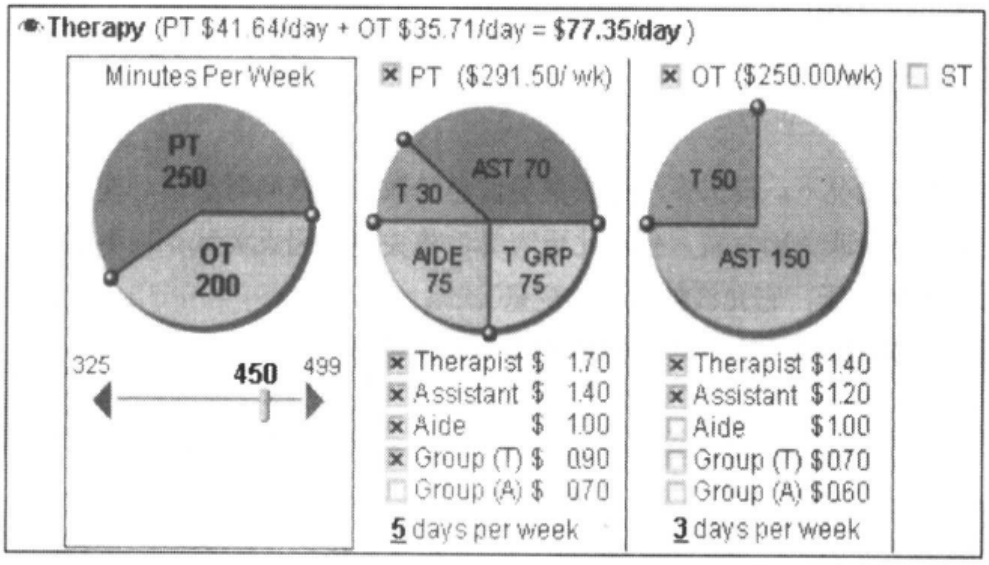


图 19-8 在一帧显示中结合文本、图形和数据。这个界面也是为共享保健系统设计的，便于病历管理员在长期的保健护理中平衡新居民的护理质量和费用。可视化地观察“如果会怎么样”脚本提纲以及详细的文本允许用户更快更好地为每位新居民及其家庭制订最好的计划。

确保内容的品质、相关性和完整性

不要因为技术上可能，就简单地显示信息。确保显示的信息能够帮助你的用户实现与他们的上下文相关的特定目标。否则，不可靠的或低质量的信息，会破坏用户通过产品的内容、行为和视觉品牌建立起来的信任。

在相邻空间上显示事物，而不是按时间堆积

如果你要表达按时间发生的变化，将这些变化安排在相邻的空间上显示，而不要彼此叠加，那么用户会更容易理解。连环画是一个用相邻空间显示时间轴上的流和变化的好范例。

当然，这些建议适用于静态信息显示；在软件中，只要技术允许（如存储器的限制和网络连接速度），使用动画会更有效地显示时间轴上的变化。

可量化的数据就要量化

尽管你想用图形和图表使趋势及其他数量信息更容易理解，但不能放弃数字本身的显示。例如，在 Windows 磁盘属性对话框中，饼形图给用户留下了空闲空间的大概印象，已使用空间和空闲空间的字节数也以数字形式显示出来。

在可视界面中使用文本和颜色

文本和颜色是用户界面视觉语言不可或缺的两大元素（文本一贯都是）。本节讨论与这两个重要的视觉工具相关的一些有用的视觉原则。

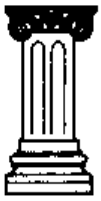
使用文本

人们处理可视信息时比文本信息更容易，这意味着使用视觉元素导航比使用文本元素更快。出于导航的目的，最好把文本当做视觉元素考虑，它们必须简单，便于认知和记忆。

文本形成可识别的形状，而大脑把它当做视觉对象分类。每个单词有一个可识别的形状，这就是为什么全部大写的单词更难辨认的原因——因为大写的单词缺乏熟知的模式匹配暗示，我们不得不花费更多的精力去辨认写的是什么。在界面中要避免全部大写的单词。

识别文字和阅读不同。阅读时，我们只需扫一眼单词就能根据上下文了解它的意思。界面试图通过阅读最少量的文本来实现成功的导航。用户导航遇到有趣的信息后，如果适宜，他会阅读详细的细节。用可视对象提供上下文帮助将有利于用最少的阅读量来实现导航功能。

如果使用视觉符号或习惯用法表达界面对象，那么我们的大脑能迅速区分它们。在视觉上辨认出我们感兴趣的对象类型后，再阅读文本来区分我们寻找的特殊对象。在这种情况下，我们不需要了解那些不感兴趣的对象类型，从而加速导航和减少附加工作。伴随的文本只有在我们认为它重要时才发挥作用。



公理：可视化地显示大致内容，文本化地显示具体对象。

界面上必读的文本适用于以下指导准则：

- 确保文本与背景对比明显，不要使用冲突的颜色，以免影响可读性。
- 选择合适的字形和字号，小于 10 磅的字难于阅读。对于简短的文本，如标签或简短说明，要用清晰的 sans-serif 字体，如 Arial 比较合适。对于成段文本，则用 serif 字体，如 Times 则更恰当。
- 文本组成段落后便于理解，用最少量的文字清楚地表达意思。词组必须明确，避免用缩略语，即使必须用缩略语，也要用标准缩略语。

使用颜色

颜色是多数（技术上支持颜色的）可视界面上的重要组成部分。现在液晶显示器无所不在，用户希望在像 PDA 和电话一类的设备上也用上彩屏。然而，颜色不仅是市场营销上的选项，也是一个强大的信息设计和视觉设计工具，具有很强的效果，也容易被滥用。

颜色作为界面视觉语言的一部分，向用户传达特定的信息。对于非娱乐性应用程序，特别是独占式应用程序来说，颜色应该与视觉语言的其他元素结合，如符号、图标、文本以及在界面上维持的空间关系。恰当地使用颜色，能为可视界面设计的以下目标服务：

- 颜色吸引注意力。颜色是丰富视觉反馈中的重要元素，一致地使用颜色来突出重要信息，提供了一种重要的交流渠道。
- 颜色提升导航和浏览速度。在导航标志中一致地使用颜色能够帮助用户快速导航和自动导向到他们寻找的信息。
- 颜色显示关系。颜色提供了相关对象分组或组织到一起的方式。

颜色的滥用

如果不小心，颜色在界面上就会被滥用。最常见的滥用如下：

- 颜色太多。Human Factors International 的研究表明，一种颜色能大幅度减少搜

索时间。增加另外的颜色提供价值的较少，在增加至七种或者更多种颜色时，搜索功能显著退化。毋庸置疑，在任何界面的导航中，都有相似的模式。

- 互补色的应用。互补色彼此正好相反。当它们相邻设置或作为图形和背景一起使用时，会造成感觉假象，难于辨认和聚焦。这是颜色立体影像（Chromostereopsis）的效果，频谱两端的颜色靠近时，会“振荡”。蓝色背景下的红色文本，或者反过来，都会给阅读造成困难。
- 过分饱和。高饱和的颜色刺眼，吸引过多的注意力。当使用多种饱和色时，常出现颜色立体影像和其他感觉假象。
- 对比度不够。如果图形颜色与背景颜色只是色调不同，而在饱和度和亮度上没有不同，则很难辨认。图形和背景除了色调不同外，饱和度和亮度也应有区别。如果可能，应尽量避免彩色背景的彩色文本。
- 对色彩缺陷注意不够。大约有 10%的男性患某种程度的色盲。因此，用红色和绿色表达重要信息时要特别小心。所用的颜色应该在饱和度和亮度上有所变化以彼此区分。如果你的调色板灰度变化容易区分，那么色盲用户就应该能够区分颜色方案。

一致性和标准化

许多内部可用性组织把自己看做数字产品一致性的看门人。一致性意味着软件产品的不同模块之间有相似的外观和行为，有时，这种特性扩展到开发商售出的所有产品上。对于多数的软件供应商来说，如 Macromedia 和 Adobe 不断地从更小的厂商那里购买新的软件产品，因而对品牌一致性的关注特别紧迫。作为一流的供应商，很明显，他们最大的利益是使购买的软件产品外观就像它们内部开发的产品。此外，苹果公司和微软都有兴趣鼓励他们自己的开发人员和第三方开发人员，创建运行与它们的操作系统平台应用外观相同的应用程序。这样，用户感觉他们各自的平台提供了无缝而舒服的用户体验。

界面标准化的益处

虽然付出了很大的代价，但用户界面标准化带来了益处。当恰当地执行时，标准化明显有利于用户。按照 Jakob Nielsen（1993）的说法：单一界面标准，能够通过提高产出和减少错误，改善用户学习界面的能力和提高生产率。之所以有这些好处，是因为用户在界面的其他部分，或遵循相似标准的其他应用程序的经验基础上，能更容易地预见

程序行为。

同时，界面标准也有利于软件开发商。因为标准带来的一致性改善了易用性和易学性，降低了客户培训和技术支持的费用。因为正式的界面标准提供了现成的界面绘制决定，所以开发团队不必在项目会议上对此进行辩论，降低开发时间和劳动强度。最后，好的标准能够降低维护费用，提高设计和代码的重复利用。

界面标准化的风险

任何标准的主要风险在于根据标准创建的产品不可能比标准更好。正如 Niesen 所说的，在制定标准时，首先特别要注意确保标准规范了一个真实可用的界面，对于必须根据规范来创建界面的开发人员来说，规范是可用的。

把界面标准视为好界面的万能药也是危险的。多数界面标准强调的是界面的语法，它的视觉外观，很少涉及界面更深的行为，或更高层次的逻辑和组织结构。这么做有一个好的理由：一个通用的界面标准没有上下文。它不考虑特定上下文中的具体用户行为和使用模式，而是关注人类感觉和认知的一般问题，有时也关注视觉品牌。这些关注都是重要的，但它们是表达细节，而不是规则依附的交互框架。

标准、指导准则和经验法则

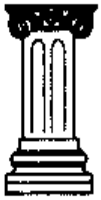
虽然标准无疑是有用的，但它们也需要随技术的演化，以及我们对用户和用户目标理解的演化而演化。一些职业人员和程序员对苹果或微软的用户界面标准奉若神明。两个企业都公布了用户界面标准，而他们又随意而频繁地违背标准，事后更新指导准则。当微软提出一个界面标准，在下一版本它将毫无疑问地改成更好的。这也很自然——界面设计仍不成熟。如果认为标准中的好处是扼杀了真正的创新，那也是错误的。在某些方面，苹果公司从 OS 9 到 OS X 显著的视觉风格变化，改变了 Mac 信徒的“界面标准是花岗岩上的蚀刻”的错误观念。

最初的 Macintosh 因为它超越苹果机以前所有的平台和标准而获得了巨大成功。相反，Mac 的力量来自这样的事实：厂商们接受苹果的领导，使得界面的外观、工作和行为都相似。同样的，许多优秀的 Windows 程序毫不迟疑地模仿 Word、Excel 和 Outlook。

因此，界面标准最适合作为具体的指导准则和经验法则。僵化地遵循界面指导准则或不考虑具体上下文中的用户需求，会强迫应用程序界面去适应不恰当的交互模型。

什么时候打破规则

我们应该如何对待界面指导准则？与其问我们“是否应该遵循标准”，不如问我们“应该什么时候打破标准”更有用。答案是，在，也只有在，我们有一个充足理由的时候。



公理：除非有真正出众的可选方法，否则遵守标准。

但是，什么才算一个充足的理由呢？只有当证明了一个新的习惯用法更好的时候吗？通常，这种测试很难定义，因为它很少能简化至可量化的程度。最好的回答是：当一个习惯用法被目标用户（你的人物角色）试用后，大多数人认为明显更好，这就是将其用于界面的最好理由。这就是工具条，总体视图、标签和其他一些习惯用法出现的原因。研究者也许在实验室对这些人工制品进行了测试，但只有在真实的软件世界里得到有效的应用，才能证实它的成功。

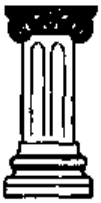
你违背指导准则的理由可能最终证明不够好，并且你的产品可能会因此付出代价。但是，你和其他设计师会从错误中学习。这就是 Christopher Alexander （1964）所谓的“自然过程”，当个体企图改进方案时，内在、未经检查的缓慢而微小进步的过程。新的习惯用法（以及旧习惯用法的新使用）会引起风险，这就是为什么认真、目标导向的设计以及在真实地工作条件下进行真实用户的恰当测试那么重要的原因。

应用程序之间的一致性和标准

当出售多种软件产品的公司决定所有产品在用户界面角度必须完全一致时，使用标准和指导准则将遇到特殊挑战。

正如前面讨论的，从视觉品牌角度看虽然有些错综复杂，但非常有意义。如果人物角色和市场分析表明，两种不同产品的用户毫无重叠，他们的目标和需求也截然不同，你会怀疑开发两种视觉品牌分别针对不同的用户，而不是使用单一的缺少目标性的外观是否有意义。当涉及到软件行为时，这些问题更加紧急。如果客户将应用软件作为套件使用，单一标准可能重要。但即使是这种情况，是否面向图形的表达软件，如 PowerPoint，应该与 Word 这样的文本处理程序保持同样的界面结构呢？微软的意图是好的，但是，在

体现整体风格指导原则方面走得太远。PowerPoint 没有从与 Excel 和 Word 相似的菜单结构中获得多少好处。因遵循不同于用户心智模型的结构而失去了易用性。另一方面，设计师在某些地方又有所区别，PowerPoint 确实有一个按幻灯片分类的显示，一个独一无二的界面。



公理：一致性并不意味着僵化。

设计师应该记住，一致性并不意味着僵化，特别在不恰当的时候。界面和交互风格指南准则必须随它所服务的软件成长和演化。有时，为了更好地服务于用户和他们的目标（有时甚至是你的企业目标），必须打破规则。当这种情况发生时，尽量使改变和增加与标准兼容。指导你的是规则的精神实质，而不是文字本身。

20

隐喻、习惯用法和启示

界面设计师，尤其是有视觉偏好的界面设计师，经常谈到寻找合适的隐喻来进行界面设计。他们认为，如果界面充满用户熟悉的实际对象图案，用户会更容易学习。于是他们创作的界面不是摆满了办公桌、文件夹、电话和参考书的办公室，就是像一沓纸或者一条布满各式建筑的街道。如果你也寻找那种魔术般的隐喻，那么你可能是在一家了不起的公司。界面领域的一些顶尖设计师把选择合适的隐喻作为他们的第一要务。

作者相信这种方法是错误的。刻意遵照隐喻会将界面和物理世界的事物不必要地绑在一起。还有一大堆其他问题：没有那么多好的隐喻，它们的可扩展性很差；用户对隐喻的认知能力经常值得怀疑，特别是在跨越文化边界时。隐喻，尤其是物理隐喻和空间隐喻，在设计大多数基于软件的产品时，有着很大的局限性。本章我们将讨论它的原因，以及一些方法来替代基于隐喻的设计。

界面范例

用户界面的概念和可视化设计中主要有三类范例：实现为中心、隐喻和习惯用法。实现为中心的界面基于对事物工作原理的理解——这本身就很困难。隐喻界面基于对事

物工作原理的直觉理解——一种有风险的方法。而采用习惯用法的界面，则基于人们对如何实现目标的学习——一个自然的，人性化的过程。

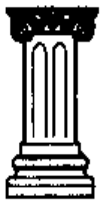
用户界面设计领域经历了从极度重视技术（实现）到同样重视隐喻的过程。虽然只有隐喻这种范例得到了命名和描述，但在当代软件设计中，不乏这三类范例的事例。尽管隐喻是人类彼此交流的强大工具（本书中也充满隐喻），但它们在界面设计中是较弱的工具，经常有碍于创作真正优秀的界面。

实现为中心的界面

实现为中心的用户界面在计算机界遍地都是。这种界面根据它们的构造，也就是软件创建的方式，来表达。为了成功地使用它们，用户必须理解软件内部的工作方式。实现为中心的范例意味着用户界面设计只基于实现模型。

今天，绝大多数的软件程序是以实现为中心的，因为它们毫不迟疑地向用户展示了系统精确的构建方式。每个函数一个按钮，每个代码模块一个对话框，命令和进程都精确地回应着内部的数据结构和算法。

我们可以通过学习如何运行程序来理解实现模型界面的构成。问题在于，反过来也是如此：为了成功使用界面，我们必须了解程序如何运行。



公理：用户喜欢成功，而不是知识渊博。

工程师希望了解程序的运行方式，所以，他们对实现为中心的范例非常满意（再加上易于构建，所以有这么多的软件采用这种范例）。工程师喜欢见到齿轮、控制杆和阀门，因为这样可以帮助他们理解机器内部的工作方式，至于那些让界面变得复杂的东西，看起来只是很小的代价。工程师也许希望理解系统内部的工作方式，但多数用户没有这种渴望，也没有时间。他们更喜欢成功，而不是知识渊博。工程师们很难理解这种倾向。

隐喻界面

隐喻界面依赖于用户在界面视觉提示与功能之间建立的直觉联系。不必了解软件的运行机制，所以隐喻界面与实现为中心的界面相比，是一大进步，但它的能力和有效性被不切实际地夸大了。

当我们在用户界面和交互设计的上下文中谈论隐喻时，我们真正指的是视觉隐喻：

用于描绘事物目的和特征的图片。用户识别隐喻的图像，通过外延理解事物的目的。隐喻的范围从工具条按钮上的小图形到一些程序的整个屏幕——从按钮上表示剪切的小剪刀到 Quicken 中完整大小的支票簿。我们凭直觉理解隐喻，但直觉真正的意思是什么？Webster 词典是这样定义直觉的：

intuition\in-‘tu-wi-shen\名词 1.敏锐的洞察力。2.立即的理解或认知；凭直觉获得的知识或信念；无须明显地理性思考和推理，获得直接知识和感觉的能力。

这个定义指明了直觉不可思议的特性，却没说我们是如何直觉理解事物的。直觉借助推论工作，我们发现不同事物之间的联系，获取相似之处的同时忽略不同的地方。我们之所以理解界面中隐喻控件的含义，是因为在心里把它们与其他我们熟悉的事物联系在一起。这是一种利用大脑进行推理的高效方式，但这种方法也取决于用户特有的大脑。因为用户也许并不具有建立这种联系所必需的语言、知识或推论能力。

【隐喻的局限性】

隐喻是用户界面设计的坚实基础，这种观念——是一种误导。这就像有很多好软件曾经采用过软盘，就崇拜软盘一样。当应用到信息时代的系统中时，隐喻存在很多局限性。

首先，隐喻不具有可扩展性。在一个简单程序的简单过程中有效的隐喻，随着程序的规模和复杂性的增加，可能会失败。当计算机在使用软盘和 10 MB 的硬盘，只有几百个文件的时候，文件图标是个很好的主意，但是，对于当前 60 GB 的硬盘和成千上万的文件，文件图标就显得太笨了，效率不高。

其次，隐喻依赖于设计师与用户之间相似的联想方式。如果用户没有和设计师相似的文化背景，就容易导致隐喻的失败。即使在相同或相似的文化背景下，也可能有明显的误解。试问一架飞机的图片是表示“查问飞机到达信息”还是“预订机票”呢？

最后，虽然隐喻在提高新手用户（第一次用户）学习能力的方面是一个小的进步，但当新手用户成为中间用户之后，却要付出很大代价。为了反映物理世界的机制，大多数隐喻紧紧地把我们的理念和物理世界束缚在一起，永远限制着软件的能力。隐喻的问题在本章的后面将继续讨论。

直觉的定义表明直觉过程不需要理性思维。在计算机工业界，特别在用户界面设计领域，直觉这个词常指易用或易于理解。易用显然非常重要，但决不能把将界面设计的成功归于形而上学。易用也不能贬低形而上学这个单词的精确含义。人们为什么理解某些界面而不理解其他界面存在多种实际的原因。

【直觉、本能与学习】

不通过任何有意识的学习，某些特定的声音、气味和图像也能引起我们的反应。小孩面对一只生气的狗，即使之前没经过任何学习，他也本能地知道露出的犬牙意味着危险。这种认知编码深藏在人脑中。本能是不需要意识参与的固有反映，直觉比本能更高一点，因为尽管直觉也不需要意识思维，但它以有意识学习所构建的知识网络为基础。

人与计算机交互的本能例子，包括屏幕上图像的总体改变会让我们吃惊和害怕；我们的眼睛必然会被网页上闪动的广告所吸引；或者我们对计算机突然发出的声音、CPU突然冒烟的反应。

直觉是有意识地学习和本能地感知事物的中间阶段。如果已经知道烧红的物体会烫伤我们，我们会倾向于将它们归入潜在危险的事物一类，直到证明不是这样。我们不必知道某种特定的烧红物体是危险的，但它为我们在开始探索之前提供了一个安全区域。

我们通常所指的直觉，确切地说，是头脑中新体验与旧知识的一种心智比较（mental comparison）。例如，你直觉地理解废纸篓图标如何工作，是因为你曾经了解真正的废纸篓如何工作，并为日后建立联系做了准备。即使你最初不知道如何使用废纸篓，学习也是件非常容易的事情。人类的大脑具有惊人的学习能力，我们毫不费力地不断学习着新事物。基于这些事实，第三种类型的界面应运而生。

习惯用法界面

习惯用法设计，Ted Nelson 曾称之为“设计原则”（1990），基于我们学习和使用习惯用法的方式，——想想“beat around the bush（拐弯抹角）” 或“cool”这样的习惯用语。习惯用法界面不关注技术知识或者直觉功能，相反，通过学习简单而非隐喻的视觉或者行为习惯用法来完成目标和任务，从而解决了前面两种界面类型存在的问题。

习惯用法表达不会像隐喻那样引起联想。当我们说“beat around the bush”时，不会联想到灌木，也不会想到什么人正在打什么东西。东西不管是凉的，还是热的，只要是令人满意的，我们都可以说“cool”。我们明白习惯用语的意思只是因为已经学过，它很独特，而不是因为我们理解它的字面意思，或者头脑中下意识地联想到什么。是的，我们都可以迅速地记住和使用这些习惯用语；这样做几乎没有意识。

如果不能直觉地理解习惯用法，那么你同样也无法通过推理理解。我们的语言充满成语和习惯用语，如果没学过，你不会明白它是什么意思。当我们说，“Uncle Joe kicked the bucket”，即使和桶或者踢没有什么关系，你也知道我们的意思。你无法通过将脚和咚咚响的桶做各式各样排列组合来理解它的含义。只有从上下文中来了解它的意思，或者你有意识地学过。你记得桶、踢和死亡之间的模糊联系，因为人类善于这样记住事物。

人的大脑确实拥有惊人的容量，它能迅速而轻易地学习和记忆大量的习惯用法，而无须与已知的情形进行比较，或者去了解它们的工作方式，或者它们为什么这么工作。大量记忆是必要的，因为多数习惯用法没有丝毫隐喻含义，其他的即使背后有典故，大多数也由于年代久远而失传了。

【图形界面主要是习惯用法界面】

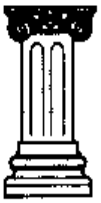
事实证明，在直觉的图形界面上，大多数元素都采用了视觉习惯用法。窗口、标题栏、关闭框、屏幕分割器、超链接和下拉菜单都是我们学习过的习惯用法，而不是具有隐喻意义的直觉。Macintosh 使用垃圾箱弹出软盘或 ZIP 磁盘纯粹是习惯用法（许多设计师认为这是个糟糕的习惯用法），尽管垃圾箱本身有视觉隐喻。

无所不在的鼠标输入设备没有任何隐喻，相反，是习惯用法的学习。在电影 Star Trek IV 中有这样一幕：Scotty 返回 20 世纪的地球，想对鼠标说话。鼠标的自然外观无法显示它的功能和用途，也不能与我们经历的其他任何事物相比较，所以不是靠直觉来学习它。但学习用鼠标点击东西令人难以置信地容易。只要第一次有人教你三秒钟，就能立即学会。我们不知道也不关心鼠标是如何工作的，但甚至小孩都能很好地操作。这就是习惯用法的学习。

具有讽刺意义的是，许多常见的，大家认为具有隐喻含义的图形用户界面元素，实际上是习惯用法。可调窗口（resizable window）和无限嵌套的文件夹这些人工产物并不真的具有隐喻含义——在现实世界中没有对应的东西，只是因为习惯用法的易学性。

【好的习惯用法只需学习一次】

由于在以实现为中心的软件体验中形成了条件反射，我们倾向于认为学习界面很困难。学习以实现为中心的软件界面很困难，因为你只有理解了软件内部的工作机制，才能有效地使用它们。我们知道的许多东西是不需要理解就能学到的：如脸、人际交往、态度、优美的旋律、品牌名称、房间布置、房间或办公室里的家具。我们不了解某人的脸为什么以这种方式组合，但我们认识他的脸。之所以认识，是因为我们见过他，并自然而然（容易地）地记住了他。



公理：所有的习惯用法都需要学习；好的习惯用法只需要学一次。

对习惯用法的重要观察是，习惯用法虽然必须学习，但它非常容易学习。好的习惯

用法只需学一次。学习“neat”、“politically correct”、“the lights are on but nobody's home”、“in a pickle”、“take the red-eye”、“grunge”等成语是非常容易的。人脑只要听一次就能记住它们。学习如单选框（radio button）、关闭框（close boxes）、下拉菜单（drop-down menu）和组合框（combobox）这些习惯用法也一样容易。

【品牌与习惯用法】

给一个简单动作或符号赋予内涵，这种理念销售和广告人员非常理解。毕竟，综合习惯用法是产品品牌的精髓，在这里，企业采用产品或者企业的名称，并赋予它所期望的含义。麦当劳的金色拱门，三菱的三块钻石，奥林匹克的五环，乃至微软的飞行窗口都是非隐喻性习惯用法，它们被赋予了通用的含义，能立即被我们认出。图 20-1 展示的习惯用法品牌例子显示了它的作用。

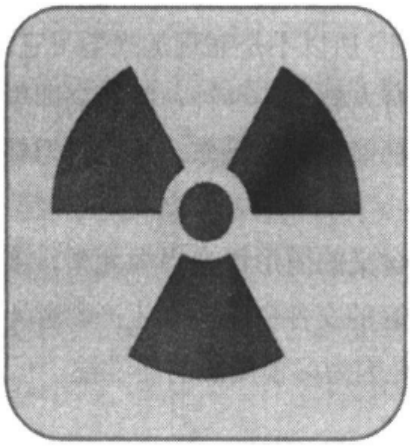


图 20-1　这是个习惯用法符号，从开始使用起就被赋予了有别于其他事物的特殊含义。对于二十世纪五六十年代成长起来的人，这个表面上毫无意义的符号却代表着核辐射，让他们不寒而栗。视觉习惯用法符号，如美国国旗，如果说没有更强大的力量，至少有与隐喻符号同等的效能。这种力量源自我们如何使用它，以及将它与什么联系在一起，而不是因为它与现实世界的事物有什么固有的联系。

隐喻的其他局限性

如果依赖隐喻含义来创建用户界面，我们面对的不仅有前面提到的小问题，而且有两个大问题：很难找到恰当的隐喻，和隐喻局限了我们的思维。

找到好的隐喻

也许寻找物理对象，如打印机和文档这样的视觉隐喻不是件难事，但要为进程、关系、服务和转换——这些软件中最常用的对象找到合适的隐喻体绝非易事，甚至是不可能的。要为买票，改变通道，购买产品，找参考资料，设置格式，旋转工具或者改变分辨率找到有用的视觉隐喻体尤其困难，但这些操作是我们使用软件时最常见的过程类型。

使用全局隐喻所带来的问题

隐喻界面最大的问题在于，它们将界面与机械时代的人工制品束缚在一起。一个极端的例子是 Magic Cap20 世纪 90 年代中期 General Magic 鼓吹的一个手持通信器的界面。它几乎各个方面都依赖于隐喻。可以从桌面的收件箱（inbox）或者笔记本访问你的信息，和门连在一起的走廊，代表二级功能。走出门去，可以访问第三方的服务。如图 20-2 所示，这些服务由街道上的建筑代表。你进入一座楼，配置服务，等等诸如此类。完全依赖于隐喻意味着你可以直觉地理解软件的基本功能，但接下来，在你明白了它的功能之后，隐喻极大地增加了导航的负担。你不得不回到街道才能配置其他服务，你必须沿着走廊进入游戏室才能玩单人纸牌游戏。在物理世界这是正常的，但没有理由在软件世界也这样。为什么不放弃隐喻，使用户更容易访问功能？General Magic 的程序员后来创建了一个书签作为快捷手段，作为拼凑上去的添加软件，但令人惋惜的是，太小也太晚了。

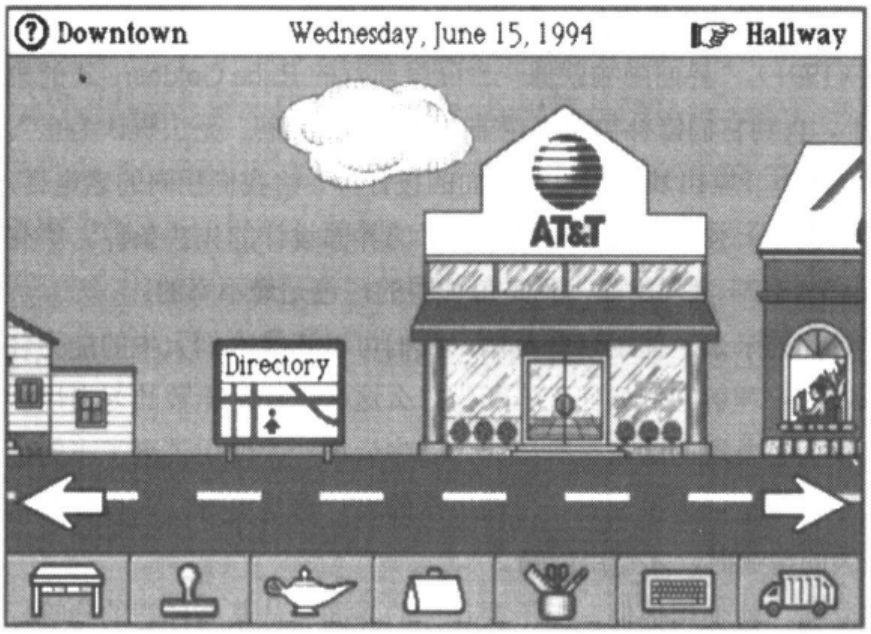


图 20-2 General Magic 的 Magic Cap 界面，在 1990 年代用于 Sony 和 Motorola 的产品中。它是一个标准的漫游隐喻设计。界面中所有的导航以及大多数其他交互附属于空间隐喻和物理隐喻。它设计起来确实很好玩，但当你成为一个中间用户后，特别不好用。这确实很不应该，因为当时一些低层次和非隐喻的数据输入交互已经非常成熟，并且得到了很好的设计。

General Magic 界面依赖所谓的全局隐喻。这是单一的基础性隐喻，它为系统中其他的隐喻搭建了框架。最初的 Macintosh 桌面也是作为全局性隐喻设计的。

全局性隐喻背后隐藏着这样一个错误信念：低层次的隐喻与全局性隐喻一致，通过联想带来了认知的好处。将隐喻超越简单功能的认知诱惑不可抵御：软件中的电话也让

你用按钮拨号，就像使用桌子上的电话一样。我们所看见的电话号码簿软件和我们口袋中的一样。难道就不能更好地超越工业时代技术的限制，发挥计算机的真正作用吗？为什么我们的通信软件不允许多次连接，或者通过组织或附属关系建立连接，或者隐藏电话号码的使用？

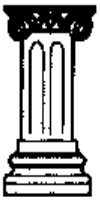
用桌子上的一张电话图片代表拨号服务似乎是聪明的，但它实际上将用户限制在缺乏创意的设计中。如果能够开发出只需指向朋友图片就能与他们通话的电话软件，那些最初的开发者们会更加欣喜。他们无法这样，因为受到现实中沉闷的电路和塑料造型所限制。另一方面，今天只要我们乐意，就能够奢侈地以任何方式绘制通信界面——显示朋友的照片是完全合理的——但我们仍抱着过去的观念不放，用过时的技术来表达。

扩充隐喻有两大陷阱，一个针对用户，另一个针对设计师。用户依靠隐喻来认知之后，他希望隐喻所指的事物与真实世界对象的行为一致。这也就为设计师设置了陷阱，他们为了迎合用户的期望而完全照搬机械时代的参考物来绘制软件。正如我们在第一部分中所看到的，将机械过程照搬到计算机中反而常使它们不如以前。

Brenda Laurel 曾说过（1991），“界面隐喻就像一台隆隆前行的 Rube Goldberg 式的机器，每次损坏后，草草修补，直到它们修补得面目全非，再也不能理解，无法辨认为止”。让我吃惊的是，那些本可以最终开发出梦幻般电话界面的设计师，给我们相同的老电话，只是因为他们记住了这样的教导：强大的全局隐喻，是好的界面设计的先决条件。所有这些错误观念起源于 Xerox PARC，全局隐喻神话是最虚弱的，也是最不幸的。

习惯用法设计是界面设计的未来。使用这些范例，我们可以依靠人们天生的能力轻松快速地学习，只要不强迫用户理解它们如何工作和为什么这么工作。无数的习惯用法等待我们去发明，而等待我们去发现的隐喻是有限的。隐喻为第一次使用带来了小的便利，却在继续使用软件的过程中造成大量浪费。习惯用法的设计总是更好，只有在非常合适，以及发现了非常强大的隐喻时，才使用隐喻。

如果你能找到合适的隐喻，就使用他们，但不要让你的界面屈从某些任意的隐喻标准。



公理：不要让你的界面屈从某个隐喻。

Mac 和隐喻：修正主义观点

在 20 世纪 70 年代中期，Palo Alto Research Center（PARC）在 Xerox 中发明了现代

图形用户界面（GUI）。PARC 定义的图形用户界面由以下部分构成：窗口（Windows）、按钮（button）、鼠标（Mice）、图标（Icons）、视觉隐喻（Visual Metaphor）和下拉菜单（drop-down menu）。由于整体的优越性，它在工业时代形成了牢不可破的地位。

PARC GUI 第一次商业上的成功实现是 Apple Macintosh 和它的桌面隐喻：废纸篓、折叠的纸张（窗口）和文件夹。但 Mac 的成功并不是因为这些隐喻，而是因为其他的几个原因，其中包括对设计和细节的整体关注。它在以下几个方面推动了交互设计的进步：

- 基于简单的鼠标动作集合，为用户定义严格限制但又灵活的与应用程序交互的词汇。
- 它为屏幕上丰富的视觉对象提供了精密的直接操作。
- 它使用高分辨率的方形像素，使屏幕显示与打印结果十分接近，特别在使用苹果公司的新产品，激光打印机时。

隐喻有助于组织这些关键的设计特性，和开辟好的市场份额，但它从来不是主要的吸引力。事实上，Mac 上市早期是相当艰难的，人们需要花时间去习惯这种全新图形用户界面的做事方式。软件开发商们开始对开发这种全然不同的软件环境持怀疑态度（微软除外）。

但是，人们最终还是被 Mac 超出其他系统的能力所征服：WYSIWYG（所见即所得）的桌面版式。所见即所得的界面与高质量的输出打印（借助于激光打印机）相结合开辟了 Apple 和 Mac 数年所拥有的全新市场。对于 Mac 公司的成功，隐喻只起了很小的作用（没别的意思）。

创建习惯用法

首次发明图形用户界面时，它们明显的优越性使众多观察家将成功归功于界面的图形特性。这是情理之中的事，但这个结论不确切，只是猜测。第一代 GUI，如最初的 Mac，之所以更好，主要是因为界面图形特性限定了一系列用户和系统进行交互的词汇。特别是图形用户界面接受来自用户的输入信息，将这些信息从非限制性的命令行转换为一系列严格限制的鼠标活动。在命令行界面，用户可以输入语言中的任何字符的组合——这种组合几乎有无限多种。用户为了正确地输入，必须准确地知道程序期望的是什么。他必须以严格的精度记住每个字母和符号。顺序很重要，有时甚至连大小写问题也不例外。

在现代图形用户界面，用户能用鼠标光标指向屏幕上的图片或单词。这些选择从用户大脑转移到了屏幕上，不需要进行任何记忆。用户利用鼠标上的按钮可以单击、双击和单击并拖动。键盘用于数据输入，而不是特有的命令输入或导航工具。用户输入词汇

的原子元素数量从成打（如果不是成百）降至三个。而与命令行系统相比，图形用户界面程序执行的任务范围并没有受到更多的限制。

交互词汇中的原子元素越多，学习过程所耗费的时间和困难就越多。像英语词汇至少要花费十年才能全部学完，并且它的复杂性要求不断使用才能保持流利，但对于熟练的用户来说，它极富表现力。在原子水平上限制交互词汇的元素数量降低了它的表现力。但是，原子元素极易组成大量复杂的交互体，就像字母组成单词，单词构成句子。

可以用一个倒金字塔代表一个适当结构的交互词汇。所有易学的交流系统遵循图20-3所示的模式。最底层包括原语（primitive），构成语言的所有原子元素。现代图形用户界面中，这些原语包括指向、单击和拖动。

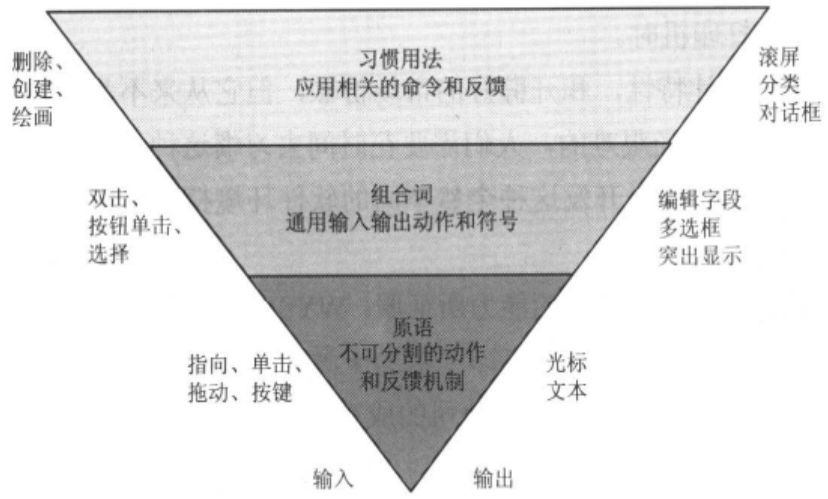


图 20-3 图形用户界面易于使用的主要原因在于推行有限的交互词汇。由指向、单击和拖动这些极少量的原语组成复杂的习惯用法。这些原语构成大量简单的组合词，进而能组合成多种复杂的特殊领域的习惯用法。所有这些都以同样几个易于学习的动作为基础

中间层包含组合词（compound）。它们有更复杂的结构，通过一个或多个原语组合创建。包括一些接受动作和表现状态的简单视觉对象，还有双击（double-clicking）和单击并拖动（click-and-dragging）这样的动作，按钮（pushbutton）、复选框（check-boxes）、超级链接（hyper-links）和直接操作句柄（direct manipulation handles）这些操作对象。

最上层包含习惯用法。使用当前的领域知识，习惯用法结合和组织了组合词。这些领域知识包括用户工作模式和目标，并不是专门的计算机解决方案。一系列的习惯用法提供了大量的词汇，来表达程序试图解决的问题。在图形用户界面中，包括标签按钮（labeled button）、字段（field）、导航条（navigation bar）、列表框（list boxes）、图标（icons）、

甚至成组的字段和控件（group of field and control），或者整个窗格和对话框（pane and dialog）。

不遵循这种方式的任何语言将很难学习。计算机以外的许多有效的交流系统都有类似的词汇。在美国，街道上的标志牌采用了形状和颜色的简单组合模式：黄色三角形表示警告，红色八角形表示强制，绿色长方形表示通告。

手动启示

启示是 Donald Norman（1989）提出的术语，定义为“事物被感觉到的特性和实际特性，主要是确定事物可能使用方式的基本特性”。

尽管定义很好，但它忽略了关键联系：如何知道提供给我们的特性是什么？如果看到某样东西，就明白如何使用它——你理解了它的启示——但你必须采用一些方法建立这种心智联系。

因此我们对 Norman 的定义做了修改，去掉了“实际”这个词。这样，启示就成为一个纯粹的认知词汇，专指我们认为对象能做什么，而不是它实际上能做什么。如果一个按钮安装在一户人家紧邻前门的墙上，它的启示百分之百是门铃。如果我们按下的时候，触发了脚下的陷阱门，我们掉进陷阱，就证明它不是门铃；但它的启示仍然是一个门铃。

那么，我们怎么会认为它是门铃呢？很简单，因为在我们复杂而漫长的社交和成长过程中学会了按门铃的礼节。我们知道这类按钮可以按，是因为看见过周围类似的电子设备，或者因为多年前与父母一起站在门阶上学会了如何进入他人家门。

但这里还有另一种力量在起作用。如果在没有把握的地方，如汽车的引擎盖上看到按钮，我们想像不出它有什么作用，但我们知道它是可以用手指按的对象。我们是怎么知道的呢？毋庸置疑，我们之所以知道是因为我们使用工具的天性。作为人类，看见手指大小的东西在我们所能触及的范围内，就会自动地去按它们。我们看见长而圆的东西，就会合拢手指像抓把手一样抓住它们。这就是 Norman 用启示所要表达的意思。但是，为了清晰起见，我们将这种如何用手操作对象的本能理解称为手动启示。当人工制品的外形明显地适合我们的手和脚时，我们会认为它们能进行直接操作不需要任何说明。事实上，基于外形和手的关系，来理解如何使用工具的行为，是一个明显的直觉理解界面的例子。

Norman 详细讨论了[手动]启示是如何比书写的指令（written instruction）更具有竞争力的。他举的一个典型例子是：在一扇只有用力推才能打开的门上用金属杆做一个把手，

杆的形状、高度和位置正好适合人用手抓。门的[手动]启示好像在说“拉我”。无论某人如何经常使用这扇讨厌的门，他总是试图把它拉开，因为该启示比门上任何“推”的标志的影响都要强烈。

手动启示数量很少。我们用手拉类似把手的东西，如果它体积小，我们会用手指去拉。我们用手或手指推平板样的东西。如果在地板上，我们会用脚来推。我们旋转圆形的东西，小的用手指，例如拨号；大一些的用双手，如操纵方向盘。这些手动启示是多数视觉用户界面设计的基础。

流行的伪 3D 系统设计，如 Windows、Mac OS 和 Motif 依靠阴影和突出显示使屏幕图像显得更立体。这些图像提供实质的手动启示，就像按钮对我们使用工具的大脑说“来按我呀！”一样。

手动启示的语义学

一个未加修饰的虚拟手动启示缺少它所执行的功能信息。我们看到一个按钮样的东西，但如何才能知道当它被按下时，能完成哪些任务？和机械对象不同，你不能只通过跟踪虚拟的杠杆与其他机械的关系，知道它的实际功能。软件用这种方法无法检测其因果关系。相反，我们必须依靠补充的文本和图片，或者更常用到的是以前的知识和体验。滚动条的启示清楚地表明它能操作，但能告诉用户它的作用的惟一东西是箭头，信息都藏在箭头的方向中。为了了解滚动条对文档位置的控制，我们不得不请教别人或者通过体验来学习。

控件须有文本或者图标标签来说明。如果答案不是直接写在控件上，我们只能通过亲自体验或培训这两种方法来学习。我们在其他地方见过，或请教过别人，或者亲自试试看会发生什么。我们不能从本能和直觉中获得任何帮助，只能依靠经验。

实现用户期望的启示

现实世界里，一个对象能做什么取决于它的物理形式和它与其他物理对象之间的联系。锯之所以可以锯木头，是因为它很锋利而且有一个把手。门上的球形旋钮之所以可以开门是因为它与插销相连。但是，在数字世界里，对象之所以可以完成任务，是因为程序员赋予它做某事的能力。通过物理观察，我们可以发现大量关于锯子和旋钮如何工作的信息，不会被所见到的现象所蒙蔽。然而在计算机屏幕上，虽然我们可以见到突起的三维矩形像一个按钮等我们去按，但是并不意味它应该被按。它几乎可以做任何事。我们之所以可能犯错误，是因为它与现实世界不同，我们在屏幕上所见到的，和它背后

的功能没有必然的联系。换而言之，我们也许不知道锯子是如何工作的，甚至因不能有效地操作它而感到沮丧，但决不会被它愚弄，所以如果它没有达到我们的期望，我们也不会提抗议。而在计算机屏幕上，错误的印象很容易发生。

当我们在屏幕上放置一个按钮时，就与用户建立了一个契约，用户按下按钮，按钮在视觉上就会发生变化：用鼠标点击，它会向下凹。进一步，这种契约表明按钮将执行图例中确切说明的某些合理的工作。这听上去显而易见，但令人吃惊的是，有那么多程序提供造成误解的手动启示。（对于按钮，这种情况发生得相对较少，但对于其他控件，这种现象很常见）。确保你的程序通过使用手动启示满足用户的期望。

第三篇

交互细节

发明计算机的科学家给我们设计了一套复杂的语言符号体系，用作与软件交流的工具。此体系的好处是非常精确，但是劳动强度太大，极易出错。所以，该行业内外的一些专家提倡采用人机对话的界面。但是语言，尤其是口语，极易造成误解和没有上下文的引用。相比之下通过示范(demonstration)来交流会比较清晰。用鼠标、指示笔(stylus)或手指指向对象的主意就出于这种理念。因为这些动作更直接，我们可以向计算机（或其他数字产品）展示做什么，而不是试图告诉它做什么。在过去的 25 年中，这种范式的细节得以不断的优化，并被广泛应用于桌面软件、Web（有某些必要的限制下）和嵌入式系统中。这些交互的复杂细节值得我们进一步去了解，这也是本篇要讨论的问题。

第五部分

鼠标和操作

21 直接操作和定点设备

22 选择

23 拖放

24 操作控件、对象和连接

21

直接操作和定点设备

现代图形用户界面建立在直接操作屏幕上的图形对象这一概念上，这些图形对象包括：按钮、滑动块（slider）和其他功能控件，以及图标和其他数据对象的代表物。选择、组合和移动屏幕对象的能力是我们当前界面设计的基础。同时，为了完成这些操作，我们也需要灵活的输入机制。本章将讨论直接操作的基本要素和用于实现直接操作的各种设备。

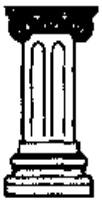
直接操作

1974 年，马里兰大学的计算机学教授 Ben Shneiderman 首次提出了**直接操作**这一说法。根据作者的归纳，shneiderman 所定义的直接操作包括以下三个要素：

- 操作对象的视觉表现。
- 具体的物理动作，而不是文本输入。
- 立即可见的操作效果。

不太严格地说，也可以认为直接操作就是单击和拖动对象，虽然这也是对的，但它很容易让我们忽略 shneiderman 观点中的精妙之处。在他概括的三个要点中，有两个与程

序提供给用户的视觉反馈有关，只有第二点与用户的动作相关。也许将直接操作称为“可视化操作”更为恰当，因为我们在过程中看到目标对象对操作非常重要。不幸的是，很多直接操作习惯用法的例子都没有充足的视觉反馈，这些习惯用法不符合有效直接操作的定义。



公理：丰富的视觉交互是直接操作成功的关键。

除了它显而易见的一面，对直接操作的另一个观察是，我们只能直接操作那些已经显示的信息。对于我们来说，必须是可见的，才能操作。这里强调的又是直接操作的视觉本质。如果你想在软件中创建有效的直接操作，必须注意用精细而丰富的图形细节来绘制数据、对象、控件和光标。

直接操作是简单、直观、易用和易记的。但是，大多数用户在第一次见到一个给定的直接操作习惯用法时，不能独立地意识到或发现它。直接操作习惯必须通过别人教授才能学会，但是教起来不难——通常只须指出它们，而且一旦学过，就很少会忘记。这是一个经典的根据习惯用法设计的例子。也许加上隐喻图片会有所帮助，但你不能指望总能找到恰当的隐喻，而且即使你找到了恰当的隐喻，也不能指望所有的用户都能很好地理解它。你只好承担起教授直接操作习惯用法的任务，聊以自慰的是教它起来还比较轻松。

关于直接操作，苹果公司的人性化界面风格指南（Human Interface Style Guide）这样说：“用户希望感到自己在控制计算机的活动”。这些已出版的指南和 Macintosh 用户界面本身都清楚地表明，苹果公司相信直接操作是优秀用户界面设计的基本原则。但是，倡导设计以用户为中心的领袖 Don Norman（1989）说过，“直接操作的第一人称系统有其自身的缺陷。虽然它们易于使用，富有趣味，令人愉悦，但要使用它们来完成一件真正优秀的工作常常是困难的。它们需要用户直接完成任务，用户也许并不善于这样。”我们应该相信谁呢？

当然，答案是两者都有道理。正如 Apple 所说，直接操作是一个极其强大的工具；也如 Norman 所说，工具必须交到称职的人手里。

这一矛盾正好阐明了两种不同类型的直接操作之间存在的差异。按下按钮是直接操作，画图程序中用画笔画画也是直接操作。虽然几乎所有用户都能按下按钮，但很少有人能够利用画笔画出漂亮的图画（就此而言，真实的笔也是这样的）。这些例子表明有两

种不同的直接操作：**程序操作**和**内容操作**。

程序操作与内容操作

程序操作包括诸如按下按钮和滚屏等动作，所有（正常的）用户都能做到。程序操作不需要特殊的技能或训练，侧重于对程序和程序界面的管理。

而内容操作主要与利用定点设备直接手工创建、更改、移动数据有关，虽然可以由任何人操作，但它的结果总是与操作者的技能和天赋相符。

诸如 Corel Draw!, Adobe Photoshop 和 Macromedia Fireworks 这些侧重于画图操作的应用程序，就满足内容操作的定义。诸如 SketchUp 或 Alias/Wavefront Maya 这些有三维模型的应用程序也是一样。Visio 甚或是 PowerPoint 这样的程序归入此类虽有些勉强，但它们更为结构化的界面仍然是以内容为中心的，需要用户有一些画图天赋。我们将在第 24 章中详细讨论画图习惯用法。

在操作类的程序中，我们发现有 5 种不同的直接操作。

- 选择。
- 拖放。
- 控件操作。
- 调整大小、形状和位置。
- 连接对象。

在第五部分的后面各章中我们将逐一讨论这些直接操作类型。

直接操作过程的三个阶段

我们可以将直接操作过程分为三个阶段。

1. **自由阶段**（Free Phase）：在用户采取任何行动之前。
2. **捕获阶段**（Capture Phase）：在用户开始拖动之后。
3. **结束阶段**（Termination Phase）：在用户释放鼠标按钮之后。

在自由阶段，还没有发生任何事情。界面的任务是在恰当的时候，显示直接操作已经准备好了。

在捕获阶段，界面有两项任务：明确显示直接操作过程已经开始；在视觉上标志动作中的潜在参与者。

在结束阶段，界面必须明白地指示用户动作已经结束，并且显示确切的结果。

直接操作习惯用法的视觉反馈

正如我们已讨论的，直接操作成功的关键是丰富的视觉反馈。让我们更深入地了解一些视觉反馈的方法。

根据所在阶段的不同，有两种不同的光标暗示。在自由阶段，光标经过屏幕上的对象时产生的任何视觉改变都称为**自由光标暗示**（free cursor hinting）。在捕获阶段开始之后，光标的变化则为**捕获光标暗示**（Captive cursor hinting）。许多程序用手形的光标来表示可以拖动的是文档，而不是文档中的信息。

如果在拖动时，同时按下一个元键，拖动的就是对象的拷贝，而不是对象本身，光标则可能会从一个箭头变为一个上面有小加号的箭头，以表示操作是复制而不是移动，很明显，这正是捕获光标暗示的一个例子。

当对象拖动时，光标必须拖动对象的整个代表物或者代表物的一些影像。例如，在一个绘图程序中，将复杂的视觉元素从一个位置拖到另一位置时，对于一个程序来说，（由于计算机性能的限制）要真正拖动一个图片是很困难的，所以经常只是拖动对象的轮廓。

直接操作是有效交互的关键

20 世纪 90 年代初期出现的 Web 和 HTML，在某种程度上代表着产业在界面和交互设计方面的巨大退步。行业人员突然明白这样的观念：仅限于滚动页面和单击一个链接的交互不仅可以被人们接受，而且工作得很好。那何乐而不为呢？还能找到比单击更简单的做法吗？这种看法过去是，现在也仍然是缺乏远见的。虽然对于从一个文档移动到另外一个文档来说，一次单击（几乎）是足够了（毕竟你还需要滚动），但对其他大多数任务来说还是远远不够的，比如：移动、分组或组织事物，绘画和雕刻事物，确定事物的位置或大小，以及其他一些人性化的任务。没有理由将 Web 或者桌面局限在一次单击的水平上，我们期待随着技术的进步，Web 应用程序在交互方面会和桌面应用程序一样丰富。幸运的是，现有的硬件条件已远远超出了直接操作的丰富应用需求，我们只须确保软件能赶上步伐。在本章的下一小节，我们将讨论常用的一些定点设备及它们所提供的能力。

定点设备

二维屏幕上对象的直接操作可以通过使用**定点设备**来实现。当然，要指向某物，最好的办法是用手指。它们总是很方便，很可能现在，在你旁边就有好几个这样的方便定点设备。它们惟一的缺点是末端太粗，在高分辨率的屏幕上无法用于精确指向，高分辨率的屏幕单靠它们自己也不知道是否被指向了。因为这个限制，其他定点设备取代了手指，每种替代设备都有其自身的优缺点。现在，鼠标是用得最普遍的，但不能肯定它是否能永远占据优势。

光笔和阴极射线管（CRT）

第一个计算机定点设备——光笔，是手指的逻辑延伸。你手中拿着光笔，像用钢笔一样指向屏幕。除了它在计算机内完全无用这个可悲的事实以外，对于直接操作来说它是一个很好的工具。

在用笔或其他书写工具时，我们练习对手部肌肉的精细运动控制，以用手指操作指示笔的尖端。为了可靠地做到这点，我们不得不将手的跟部搁在一个固定物上，否则，我们的动作就会漂移。无论我们的手指动作如何精确，动作都会漂动，除非我们为我们的手提供一个牢固的基础。

计算机用大而笨拙的阴极射线管或者精密的液晶显示板作为显示屏。显示屏通常垂直面向我们，而不是像书本和纸张一样平铺在桌上。固定的水平面上，在平铺的纸上用笔写字是很容易的，但在没有支撑的情况下，悬空的手臂和手在垂直表面用同样一支笔做出精确的移动则相当困难。在垂直显示平面上使用钢笔，浪费了我们所说的手指精细运动控制（fine mofor confrol），迫使我们去依赖手臂肌肉的粗略运动控制（gross motor control）。手臂肌肉只适合用于远距离移动，无法达到我们对精确度的期望值。

当我们的手掌边缘靠在一个垂直面上时，要在这个面上绘画也是极其困难的，你不妨在墙上试试。你的腕关节不能向后弯到足够的程度。经常在墙、门和窗等垂直平面上画画的人常用一种叫腕杖的工具——末端有一个衬垫的半米长木杆。画家将衬垫端支在墙上，另一端抓在手里。画画的那只手搁在杆的中间。腕杖帮助她改变了相对绘画平面的入射角度，从纯粹垂直角度转变为她更容易控制手绘画的角度。

不幸的是，腕杖对于计算机用户是不切实际的，因此我们发明了鼠标一类的其他工具。

鼠标和间接操作

在桌面上滚动鼠标时，你看见一个视觉符号——光标，以同样的方式在显示屏上移动。鼠标向左移，光标就左移；鼠标向上移，光标就向上移。第一次使用鼠标的时候，你立刻能意识到鼠标和光标相互联系的，很容易学会，而且很难忘记。这样很好，因为靠观察来了解鼠标如何工作几乎是不可能的。电影 Star Trek IV：The Voyage Home[1] 中有这样一幕：从 24 世纪回到 20 世纪的地球人正在试图使用一台计算机，他拿起鼠标，用嘴对着它说话。这一幕很有趣，因为它潜在地表明了一个事实：鼠标没有表明它是定点设备的视觉启示，直到有人向我们展示光标的移动怎样与鼠标的活动相关联。在这一点上，理解是瞬间完成的。所有习惯用法都必须经过学习才能使用。好的习惯用法只学习一次。就此而言，鼠标当然是一个好的习惯用法。

但鼠标的活动与光标的活动常常不是一对一的，而是成一定比例。在多数 PC 机上，鼠标大约移动 4 厘米，光标跨过 30 厘米的屏幕时。手靠在坚实的桌上，手指可以以很高的精确度移动鼠标。手部肌肉的精细运动控制甚至能以 1 比 8 的比率实现光标的精确定位。那些掌握鼠标存在困难的用户，常常是因为没有将手掌的跟部牢牢地靠在桌面上。

虽然当我们谈论用鼠标指向和移动对象时用了“**直接操作**”这个术语，但实际上，我们操作这些东西是间接的。光笔直接指向屏幕，相对于鼠标，称它为直接操作工具更合适，因为我们确实指向了对象。可是在使用鼠标时，我们只是在桌上操作鼠标，而不是指向屏幕上的对象。

借助于一支纤细的指示笔，我们能获得对位置的精确控制，但用手掌大的鼠标，手指尖的肌肉就起不到对笔那样的作用。这就是为什么我们在实践中无法用鼠标进行手写输入的缘故。虽然在使用鼠标时用到了精细运动控制，但这和我们用笔尖做练习那样的极其精细的控制完全不同。手拿着大得多的鼠标，我们可以轻易地移动鼠标到特定的位置，但我们不能有效地限定形状，不能连续地相对自身移动书写草书或印刷体。因此，鼠标是很好的指向屏幕对象工具，但它很难用于输入图形数据。如果是在水平表面上做数据输入，则指示笔都能很好地完成这两项任务。

---

[1] **译者注** 星际迷航，其中 24 世纪的人可以通过语言直接操纵计算机。

其他定点设备

鼠标是灵巧的工具，它允许我们指向垂直平面上的事物，又解除了我们在垂直表面上指向或绘画的缺陷。在其他情况下，它是不如笔的。你可以用笔写草书，但用鼠标做不到这一点。这一事实足以说明笔比鼠标操作起来更精确。只有在垂直表面上书写时，鼠标才能体现其优越性。

多年来，有许多其他的定点设备与鼠标竞争，但始终未被广泛接受。其中包括轨迹球（trackball）、数字化写字板（digitizing tablets）和触控板（touchpad）。

【轨迹球】

轨迹球存在的时间几乎与微型计算机一样长，但它除了用于游戏和早期的膝上型电脑，从未真正地流行过。原因是，它没有鼠标用起来顺手。轨迹球中最大的问题是按钮的设置。在光标可以到达的范围内，轨迹球的精确度是否和鼠标一样或者更强，这点尚无定论。轨迹球无需滑动，所以就桌面上的占用面积而言更小。但轨迹球上的按钮必需设置到旁边，给单击造成困难。也许，如果按钮（至少对 Mac 来说）能安置到球下，以致轻轻下压球体就能实现单击（与无按钮苹果型鼠标概念相似），则用户对轨迹球的接受度会高一些。

【数字化写字板】

数字化写字板也存在多年了，虽然在艺术家和图形设计师中有一批拥护者，但多数人发现他们几乎不可能用它来导航计算机桌面。部分问题也许是人们习惯于使用鼠标。鼠标是相对指向工具——拿起它，然后在任何地方放下都不会改变光标在屏幕上所处的位置，只有在平面上移动鼠标才会影响光标的位置。写字板是绝对定点设备——写字板上每一位置都直接与屏幕上相应区域对应。所以，如果你从左上角拿起笔放在右下角，光标就会立刻从屏幕左上角跳到右下角。数字化写字板真正的问题在于，显示和数字化区域是分开的。从某种意义上说，这是造成大多数人使用它比较困难的主要原因。再加上数字化写字板价格不菲，你就能明白为什么它从未成为主流了。

【触控板】

触控板（touchpad，因为某些原因，苹果公司称为 track pad），是迄今为止继鼠标之后最成功的一种定点设备。这主要是因为它们在笔记本电脑中得到普遍应用，而笔记本电脑正变得比台式电脑更流行。触控板令人着迷之处在于它结合了鼠标、轨迹球和数字化写字板的行为。它采用一个像写字板一样的小型数字化平面，但和轨迹球或鼠标一样，

你是用手指而不是特制的笔。和鼠标一样，它是相对定点设备——光标的移动只取决于你的手指在触控板表面的活动。和轨迹球相似，它的按钮不得不放在一个不方便的位置，只能用大拇指才能接触到它。近期的许多触控板驱动程序让你通过轻叩或者双叩来实现单击或双击。只要你的手指轻叩时不出现意外的滑动，就能很好地完成任务，否则就会引起光标移动。当然，如果你需要（在 Windows 中）右击[2]，就不得不用按钮了。尽管触控板不够完美，但它有一个绝对的优势：当你带着笔记本电脑旅行的时候，因为它的存在，就不必再带一个鼠标了。

【指杆】

与其他一些品牌一样，IBM 和 Toshiba 的笔记本电脑应用了一种有点奇怪的定点设备，称为指杆。它由键盘上位于 G、H、B 键之间的一个铅笔橡皮大小的小球构成。手指轻推橡皮的头，就可以朝推的方向移动光标。虽然有些人喜欢这种设备，但多数用户发现它难于控制，用户深受和触控板按钮设置同样的问题困扰。

笔的回归

大约在十年前，最早出现可用的，融合了 LCD 显示技术和数字化板技术的产品，就是触摸屏。其中，最早也最著名的无疑是 Apple Newton。这种设备允许用户用指示笔或手指直接操作屏幕上的对象，直接在屏幕上写字，甚至能将手写输入转换成计算机可读的文本（虽然最早的 Newton 因其书写识别存在的问题而臭名昭著）。Newton 的设计是让你能够像拿笔记本那样用手拿着它，因此它提供了用户更需要的水平方向显示。这样用户的操作精度能够达到像使用笔一样。这种设备要足够大，才能适当地支持手的操作。这也许是它衰落的原因之一。许多人发现它体积太大，不方便随身携带，因此削弱了它作为电子笔记本的功能。

在苹果公司之后，掌上电脑（Palm）很快就应运而生。他们创建了一个更小巧的设备，该设备不能识别书写体，却可以识别一种名叫 Graffiti 的简化字母表。奇怪的是，没有第三方软件的支持，你无法在屏幕上直接输入 Graffiti 文字；相反，需要在屏幕下方的数字化区域输入（也许它是 Palm 的交互中最脆弱的部分）。然而，你可以用指示笔或手指在小的方形数字显示屏上直接操作对象。Palm 最成功之处在于它小巧简单。因此它曾经，并且现在仍然获得了巨大的成功。现在 Palm OS 甚至进入到一些手机领域。

---

[2] **译者注** 因为 Windows 鼠标采用两个按钮，而 Mac 鼠标只要用一个按钮。

在你读到本书的时候，微软可能已经推出了它的平板（Tablet）PC 平台。数年前，微软针对基于笔的计算机发布了 Windows 版本，但硬件技术进步太慢，速度太慢，体积太大，阻碍了它的流行。时光流逝，Tablet PC 重量轻，速度快，有高分辨率的显示器和强大的书写识别能力。但最重要的是，它是一种形态要素（Form-factor），人们又回归到用手写字的舒适体验。笔最终会取代鼠标的位置吗？只有时间能够回答。

但目前，我们仍然对鼠标感到满意。让我们进一步探讨它的交互模型。

鼠标的使用

当用鼠标在屏幕上漫游时，在近距离活动和远距离移动之间有一明显的分界线：你的目标可能很近，就可以将手掌跟部放在桌上保持不动，否则你必须抬起手。放下手掌将光标从一个地方移到另一个地方，这时你使用的是手指肌肉的精细运动技能。当你抬起手做更大的移动时，你用的是手臂肌肉的粗略运动技能。粗略运动技能并不比精细运动技能快或慢，但两者之间的转换是困难的，要耗费时间和精力，因为用户必须将两组肌肉整合起来。打字者不喜欢让任何事物追使他们的手离开键盘上的基准[3]位置，因为这需要肌肉群之间的转换。基于同样的原因，也不喜欢跨越屏幕位置来移动鼠标操作控件，因为这样不得不需要在精细运动控制和粗略运动控制之间进行转换，然后又回到精细运动控制。

单击鼠标上的按钮也需要精细运动技能——用你的手指往下压它——如果手没有牢固地靠在桌面上，你在无意地移动鼠标和光标的情况下就无法单击它。控制鼠标移动，在精细运动控制和粗略运动控制之间可能存在折中的方法，但当要直接地单击按钮——用户必须首先将手掌根部置于桌面，进入精细运动控制模式。用鼠标操作控件，用户必须用精细运动控制来实现光标对复选框或按钮的精确定位。但如果光标远离目标控件，则用户必须先用粗略运动控制来移动光标接近控件，然后改成精细运动控制来完成工作。某些控件在交互时混合了这些问题（见图 21-1）。

显而易见，对任何程序来说，如果它可单击的区域超出几个像素的范围，都容易产生问题。如果一个控件需要用户在较近处点击后又点击较远处，那么这个控件的设计是不好的。但无所不在的滚动条就是这样的一个控件。假如想滚动整个文档，你必须使用精确运动控制单击向下箭头数次，直到找到你要找的位置，但你可能点的次数过多，以致错过你的目的地。这种情况下，你必须单击向上箭头回到你想去的地方。当然，移动

[3] **译者注** 即键盘上的（A、S、D、F；J、K、L）。

光标到向上箭头，你必须抬起手掌跟部做粗略运动控制的移动。继而又放下手，做精细运动控制的移动来确定箭头的位置，并且在你单击按钮时稳住鼠标。

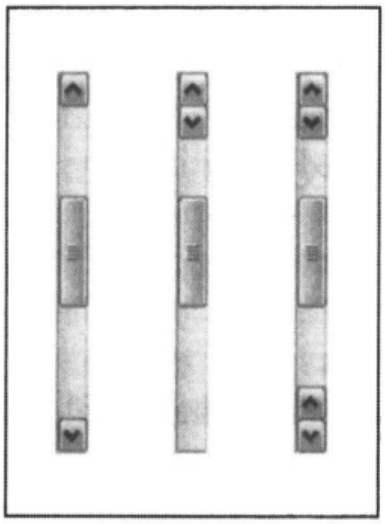


图 21-1 左边显示的是我们熟悉的滚动条，是图形用户界面中一种较难使用的控件。从向上滚屏转变到向下滚屏，你必须从单击按钮所需的精细运动控制，转变成移动手到滚动条的另一端所需的粗略运动控制，然后再换成精细运动控制精确定位鼠标，并且单击按钮。位于图中间的滚动条稍微做了一些调整，两个按钮靠在一起，排除了以上问题（ Macintosh 的滚动条两个箭头按钮被设置在底部）。右边的滚动条在视觉上有点杂乱，但有更为灵活的交互。关于滚动条更多的信息见第 26 章。

为什么滚动条上的箭头要分开在两端呢？是的，采用这种方式，在视觉上显得更对称，但使用起来更困难。如图 21-1 所示。如果两个箭头在滚动条的一端彼此紧连，简单的精细运动控制的移动就能改变滚动方向，而不需要再做精细——粗略——精细动作控制的艰难变换。微软和鼠标制造商已经致力于在硬件方面解决这个问题。他们在鼠标左右键之间加上一个滚轮。向前或向后滚动这个滚轮，活跃窗口的滚动条就会相应地向上或向下移动。

不仅手工操作不很灵敏的用户发现鼠标有问题，有许多经验丰富的计算机用户，特别是打字员，也发现使用鼠标有时存在困难。对于数据密集的任务，键盘比鼠标要好。手不得不离开键盘去操控鼠标，调节光标的位置，然后又不得不回到键盘，这是一件令人沮丧的事。在个人电脑的初期，键盘一统天下，如今又常常是鼠标一统天下。程序应该对所有的移动和选择都提供完整的鼠标和键盘支持。

设计技巧：对移动和选择任务应该同时提供鼠标和键盘操作。

相当多的计算机用户使用鼠标有麻烦，所以如果我们要想成功，则在设计软件中，考虑到老练的鼠标用户的同时，也要照顾到这部分人的情绪。这就意味着每种鼠标的习惯用法都至少存在别的替代方法，当然这不完全可能。例如，试图在画图程序中支持一些面向图片的动作时，不使用鼠标是荒谬的，但这是极少数的情况。多数商业和个人软

件有很好的键盘命令，多数用户将鼠标和键盘命令结合起来使用，有时用鼠标启动命令，用键盘结束命令，或者与之相反。

鼠标按钮知多少

鼠标的发明者想要得出鼠标应该有多少按钮，但他们没有达成一致意见。一些人说一个按钮是对的，另一些则深信应该有两个。还有一部分人鼓吹多个按钮的鼠标可以用来分别或一起单击按钮。比如有 5 个按钮的鼠标，其单击方式就有 32 种不同的组合。最终，苹果公司为它的 Macintosh 安装了一个按钮，微软采用两个按钮，而 UNIX 社团（特别是 Sun Microsystems）采用了三个按钮。

单个按钮的鼠标是 Macintosh 最主要的缺点之一。苹果公司广泛的用户测试显示新手（追溯到 1984 年，谁不是初学者呢？）最适合的按钮数目是一个，因而单按钮的鼠标庄严载入了苹果公司历史的万神殿。当新手用户走出初学阶段，成为永久的中间用户（perpetual intermediate）[4]，鼠标右键发挥作用的时候，单键鼠标就成为一种不幸。为了新手的简洁，它牺牲了大多数计算机用户的效能。

单键鼠标和双键鼠标阵营之间的差异其实不大。创建鼠标左键的目的习惯上定义为“与 Macintosh 鼠标的单键相同”。换句话说，大家普遍认为右键是额外的按钮，左键是用户真正需要的。但是如今，随着右键上下文弹出菜单的演化，这种描述不再正确。

尽管许多 UNIX 工作站系统有三键鼠标可供选择，但中间键很少用到。中间键常认为是给应用程序的保留键。不用说，很少有程序用到它。

鼠标左键

总的来说，鼠标左键用于所有主要的直接操作功能：启动控件、选择、绘画等。推而论之，意味着左键不支持间接功能。间接功能要么通过右键来完成，要么不能通过直接操作访问，只存在于菜单或键盘中。

左键最常用的方式是激活或选择。对于按钮或复选框这样的控件，单击左键意味着按下按钮或者选择选项。如果你在数据上单击，则通常意味着选择。我们在下一章进一步讨论这个问题。

---

[4] 译者注　见第 3 章的解释。

鼠标右键

微软和其他许多公司曾经长期对鼠标右键视而不见。只有少数勇敢的程序员将一些行为与右键关联起来。这些行为被看作额外的、可选择的或高级的功能。当 Borland 国际公司用右键作为进入属性对话框的工具时，业内对这种行为的看法充满矛盾，如他们所说，“充满批判性的欢呼”。当然，大多数可用性方面的评论来自 Macs，它只有单键，微软也蔑视 Borland，所以最初这种理念并未得到应有的流行。在微软最终步 Borland 的后尘，发布 Windows 95 之后，情况才有所改变。今天，鼠标右键担负了重要而又极其有用的角色：支持直接访问属性，和其他针对对象和功能的上下文相关动作。

鼠标中间键

虽然，应用程序供应商希望使用鼠标右键（除了 Mac），但他们不指望鼠标的中间键。因此，多数供应商只能将它当作捷径。事实上，在微软的风格指南中说，中间键“应该分配给界面上已有的操作或功能。”而一度将它定义为右键的储备。

作者有一些使用鼠标中间键的朋友。实际上，他们信赖它，用它作为双击左键的捷径——该特性通过配置鼠标驱动软件来获得，但应用程序仍然愉快地忽略着它。

用鼠标指向和单击

最基本的，你可以利用鼠标做两种原始的操作：移动它指向不同的事物，或者单击按钮。任何超出指向和单击的鼠标动作都是一种或多种动作的组合。鼠标动作词汇是遵循这种规律形成的。鼠标因此成为好的计算机外围设备。

应用元键：Ctrl, Shift 和 Alt 也可以改变鼠标行为。我们将在本章后面部分讨论这些键。不借助元键的条件下，能完成的所有鼠标动作总结如下。为了讨论方便，我们为每种行为取了个简短的名字（见圆括号中显示）。

- 指向（指向）。
- 指向，单击，释放（单击）。
- 指向，单击，拖动，释放（单击拖动）。
- 指向，单击，释放，单击，释放（双击）。

- 指向，单击，单击其他键，释放，释放（合击）。
- 指向，单击，释放，单击，拖动，释放（双拖）。

这些行为中除了合击以外，每一种行为都可以由双键鼠标的任何一个键完成。精通鼠标的用户可能完成过所有这六种行为，但普通用户只见过表中的前五种。对鼠标心存恐惧的用户认为可以接受其中的前三种。Windows 和 Mac OS 设计时，只用到前三种。在这两种操作系统中，用户为避免使用双击，不得不绕道完成他们的目标任务，但至少这是可以做到的。

【指向】

这种简单操作是图形用户界面的基石，也是所有鼠标操作的基础。用户移动鼠标，直到屏幕上的光标指向所期望的对象或置于对象之上。即使没有单击，只是指向，屏幕上的对象也能注意到。直接操作的对象常微妙地改变它的外观。当鼠标光标移到对象上时，对象会显示这种特性。这种行为称为受范性（pliancy），我们将在本章后文中进一步讨论。

【单击】

用户将鼠标停留在固定的位置，按下按键然后释放。通常，这种动作定义为触发控件状态改变或者选择一个对象。在文本或单元表格中，单击意味着“选中此处”。对于一个按钮控件，状态改变意味着当鼠标按键在控件之上按下时，按钮进入或保持按下的状态。当鼠标按键释放，按钮就触发，与之相关的动作就会出现。

设计技巧：一次单击选择数据或改变控件状态。

但是，如果用户在鼠标按键按下时，移动光标离开控件，那么按钮控件就会回到未按状态。尽管直到释放鼠标按键为止，输入焦点仍然在控件上。但当释放鼠标按键后，输入焦点切断了，任何事也没发生。如果用户改变了主意，这也提供了一条方便的逃逸途径。本章后文中将详细讨论单击过程中鼠标按下和鼠标释放的事件。

【单击拖动】

这种通用的操作有许多常用的用法，包括：选择、改变形状、改变位置、绘图、拖放。在第五部分接下来的各章，我们都会讨论到它们。

滚动条是一个有趣的灰色区域，因为它包含了拖动功能，但它是控件，而不是对象（如桌面图标）。在单击鼠标按钮时，Windows 的滚动条保持全部输入状态，允许用户在

没有直接将鼠标置于滚动条上时，也能成功滚动。但是，如果用户的鼠标离滚动条太远，它就会自动回到单击前的位置。这种行为可以理解，因为长距离的滚动需要粗略运动控制，使它难以保持在滚动条控件的范围内。如果拖动得太远，则滚动条做出合理的猜测，即用户并不是首先想滚动。一些程序将这种限制设置得太窄，导致滚屏的行为变幻无常，令人沮丧。

【双击】

如果双击由两次单击组成，逻辑上似乎应该是双击首先要完成与单击相同的事情。当鼠标指向数据的时候确实是这样的。单击选择对象，双击选择对象然后采取动作。

设计技巧：双击意味着单击加动作。

这种基本的理解来自 Macintosh 的上一辈 Xerox Alto/Star，它也是当代图形用户界面的标准。事实上，对于不太灵活的用户来说双击是困难的——一些人感到痛苦,少数人根本做不到——这种现状在很大程度上被忽视了。行业必须正视这个难堪的现实：尽管有相当数量的用户存在问题，但大多数用户双击时不存在问题，而且舒服地使用鼠标工作。我们不能因为相对少数人的缺陷，而处罚大多数人。解决方法在继续探讨中，包括双击习惯用法并确保有对应的单击习惯来实现同样的功能。

尽管在数据上双击有着确定的定义，但在多数控件上双击没有意义（如果你将图标归入数据，而不是控件）。额外的单击忽略不计，或者更普遍的情况下认为它是第二次独立的单击。根据控件的不同，这种动作可能没有危险，也可能造成麻烦。如果控件是切换按钮，你会发现只是回到了它的初始状态（迅速地打开后关闭）。如果控件在第一次单击后已经发挥作用，例如，对话框里的 OK 键，结果就不可预料——无论在按钮下的是什么，获得的都是第二次按钮按下的信息。

【合击】

合击意为同时单击两键，尽管并不真正需要两个按钮精确地同时单击或释放，但为了合击有效，第二个鼠标按钮被单击，必须在第一个鼠标之前。

有两种不同的合击。第一种最简单，用户指向某对象并同时单击两个按钮。这种习惯用法很笨拙，而且在现存的软件中不多见，虽然一些创造欲望强烈的程序员在选择时将其作为 shift 键的替代品。

第二种是使用合击来取消拖动。拖动开始是简单的单按钮拖动，接着用户增加了第二个按钮。虽然这种技术听上去比第一种更晦涩难懂，但它确实在行业中得到更广泛的

接受。它适于取消拖动操作，我们将在下一章更详细地讨论。

【双拖】

这是另一个仅为专业人员所用的习惯用法。无错误地执行双击和拖动就像在拍你头的同时又摸你的肚子。像三击一样，仅在主流，水平，独占式的应用中使用。将它作为选择的扩展使用。例如在 Word 中，你可以双击文本选择整个单词；作为功能的扩充，你可以通过双拖扩展逐个逐个单词的选择。

大的独占应用软件中有很多选择的替换方法，双拖习惯用法在它们中的应用是合适的。但是除非你已经创建了这个怪物，否则还是坚持用更基本的一些鼠标行为为好。

鼠标释放和按下事件

每次用户单击鼠标按钮，程序必须处理两个不连续的事件：鼠标释放事件和鼠标按下事件。因为鼠标管理缺乏一致性，所以鼠标释放和鼠标按下采取的动作定义也根据场景和程序的不同而有差异。这些动作应该严格一致。

当选择对象时，选择总发生在按钮按下时，因为按钮按下是拖动序列的第一步。根据定义，如果不首先选择，你就不能拖动，所以选择必须发生在鼠标按下的时候。否则，用户就不得不需要双拖来完成。

设计技巧：在数据上鼠标按下意味着选择。

另一方面，如果光标指向控件，而不是所选数据，按下按钮的动作是暂时激活控件的状态转换。当控件最后发现按钮释放事件后，它就会执行状态转换。

设计技巧：在控件上鼠标按下意味着预备动作；鼠标释放意味着执行动作。

这种机制允许用户从容退出无意中发生的单击。例如，在按钮操作中，用户只需将鼠标移出按钮，这样即使释放鼠标按键，选择也会无效。对于复选框，意思也相同：一旦鼠标按下，所选择的复选框视觉上已经激活，但直到鼠标释放，选择才会真正有效。

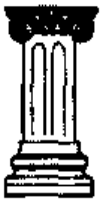
光标

光标是屏幕上鼠标位置的可视代表物。依据惯例，它通常是一个指向左上角的小箭头，但在程序控制下，它能在一个较小的范围内——32×32 像素内改变成很多形状。因为光标必须经常是为了解决单个像素的问题——指向的对象只占一个像素——所以必须有某种方式让光标精确地指示它所要指向的确切像素。解决方法就是在任何光标上设置一个单像素作为指向的确切位置。这个单像素称为热点（hotspot）。对于标准箭头，逻辑上热点是箭头尖部。无论光标的形状如何，它都有一个热点像素。

受范性和暗示

当你在屏幕上移动鼠标，鼠标指向的某些事物是惰性的：鼠标热点位于它们之上时，单击鼠标也没有反应。相反，**受范性的**对象或区域对鼠标活动有反应。可按下的按钮（Push-button）控件是受范性的，因为它能被鼠标光标按下。任何能被选择和拖动的对象都是受范性的；因此文件管理器和浏览器中的任何目录和文件图标都是受范性的。实际上，电子表格中的每个单元和文本文档中的每个字符都是受范性的。

如果屏幕上的对象是受范性的，这个事实就必须告诉用户。如果事实表达不清楚，对于非专家用户来说，这种习惯用法就失去了作用（习惯用法当然是有用的，但总的来说，我们能把更多的信息更好地传达给每一位用户）。



公理：*在视觉上暗示受范性。*

将对象的受范性传达给用户有三种基本方式：通过对象本身的静态视觉启示，动态改变视觉启示，或者在光标经过对象时改变光标的视觉启示。

【静态和动态的视觉暗示】

静态视觉暗示——即对象的受范性由对象本身的静态视觉启示表达——通过对象绘制在屏幕上的方式来提供。例如，三维造型的操作按钮是一种静态视觉暗示，因为它有可供按下的手动启示。

某些具有受范性的视觉对象不太明显，或者因为太小，或者因为隐藏的缘故。如果受范性对象不在程序主窗口的中心区域，并且具有不太通俗的外观，用户也许不会理解到这个对象能够操作。这就要求更积极的视觉暗示：动态视觉暗示。

动态视觉暗示是这样工作的：当光标经过受范性对象时，它用动画改变对象的外观。切记，这一动作出现于鼠标按钮被单击之前，并且仅仅在光标掠过时激活的行为。自 Windows 98 出现以来一个好的例子是工具条上的**图标按钮**（butcon，看起来像图标的按钮）：虽然为了减少工具条视觉上的混乱，图标按钮没有常规的按钮启示，但当光标从任何一个图标按钮上经过时，启示会再次出现。这种结果是有力的暗示：控件具有按钮行为，当然，它确实是这样。这种习惯用法在 Web 上非常普遍。

这种层次的积极视觉暗示足以作为强大的训练工具，还可以提醒用户受范性区域的位置。

【光标暗示】

当光标经过对象时，**光标暗示**通过改变光标的外观来传达对象的受范性。

很多流行的软件将视觉暗示和光标暗示进行任意组合，我们已经习以为常了。例如，按钮呈立体三维，它的阴影清楚地表明对象突起，可以按下，经过突起的按钮时，光标并没有发生改变。另一方面，当光标经过窗口外框的时候，光标变成双箭头，表明窗口边缘可以拉长。这是框架可以拉长的惟一确定的视觉启示。虽然光标暗示常改变光标形状，来显示哪种类型的直接操作可以接受，但它最重要的作用是让用户明白对象是受范性的。要在视觉上提示数据的受范性，而又不影响数据的正常表达是困难的，所以光标暗示是最有效的方法。某些控件很小，用户难以像点击按钮一样点击它，这时光标暗示就是成功的关键。如图 21-2 中所示,微软的 Excel 中的分栏（column divider）和屏幕分割器（screen splitter）就是很好的例子。

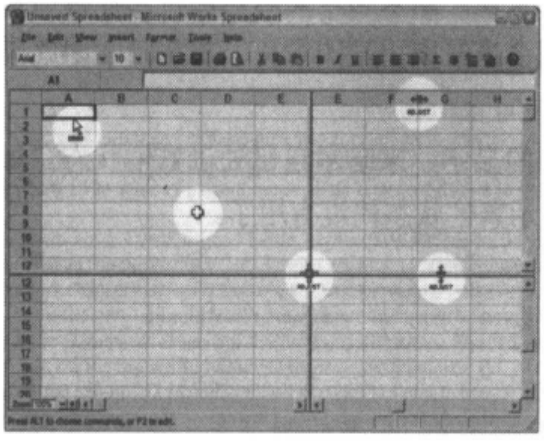


图 21-2 Microsoft Work Spreadsheet 使用光标暗示来强调视觉受范性不明显的控件。列宽和行高能通过拖动相邻两列之间的垂直线来设置，此时光标变成两个头的水平箭头，暗示受范性的同时还指示容许的拖动方向。对于屏幕分割器也是一样。当鼠标经过为未被选择的可编辑单元格时，它显示一个加号光标；当它经过一个选中的单元格，显示为拖动光标。在这些光标上附加文本使它们几乎成了工具提示。

概括来说，控件常提供静态视觉暗示或动态视觉暗示，而受范性的（可操作）数据更常提供光标暗示。我们将在第 23 章中再对暗示进行讨论。

设计技巧：显示受范性是光标暗示最重要的作用。

【等待光标暗示】

存在一种光标暗示的变体，称为等待光标暗示。按人们的说法，无论程序做什么需要花费一定时间的事情——比如访问磁盘或重建目录——程序会改变光标，在视觉上暗示程序变为不响应状态。Windows 操作系统采用的图形是我们熟悉的沙漏。其他操作系统使用过手表，旋转的球和热气腾腾的咖啡杯。当程序变得迟缓时，通知用户是个好主意，但光标不是完成此项工作的合适工具。首先，光标属于每个程序，而不是任何特定的程序。

因为光标属于系统，仅仅在它侵占了程序空间时，才被程序借用，用户界面问题由此产生。在 Windows 或 Mac OS X 这些抢先式多任务系统里，一个程序变得迟缓，没有必要使其他运行程序也变得迟缓[5]。如果用户指向这些程序中的一员，它就需要用到光标。所以，光标不能用于表示某个程序处于繁忙状态。

如果程序必须对用户的单击熟视无睹，应该通过屏幕上程序自身所拥有的指示器告诉用户，而不是光标。它可以图形化地显示相应的功能，并在它的主窗口或者程序运行期间出现的对话框中显示进度，还可以用提供取消操作的方式来实现。用对话框这种习惯用法，效果不如直接在程序主窗口里绘制相同的图形。

Windows 操作系统仅在程序所在的窗口显示沙漏光标，这在逻辑上是正确的。但是，这也意味着繁忙的程序没有提供迟缓状态的视觉反馈。如果用户无意识地将光标移出正忙的程序主窗口，进入另一个运行程序，光标就会恢复成正常的箭头。视觉暗示因此失去了效果。

总之，每个程序必须通过在视觉上改变自身的显示，来暗示自己处于繁忙状态。如果繁忙状态取决于光标所指的位置，用光标来表示繁忙状态就没有用了。

---

[5] **译者注** 因为任务可以抢先，各个程序的调度变得公平。

输入焦点

输入焦点（input focus）是一种模糊的技术状态，它很复杂，以致一些图形用户界面专家都在上面犯了糊涂。

Windows 和多数其他桌面操作系统平台是多任务的，也就是说，在任何指定的时间可能有多个程序在执行有用的工作。但无论有多少程序同时运行，只能有一个程序与用户直接接触。这就是输入焦点这个概念产生的原因。输入焦点表明哪个程序将接受用户下一个输入信息。为了讨论方便，我们可以认为输入焦点与激活相同：某个时刻只有一个程序是活动的。这是纯粹从用户的角度看问题。程序员们通常必须做更多的准备工作。活动的程序有着最突出的标题栏。

最简单的情况下，有输入焦点的程序将接受下一个键盘输入信息。因为普通的键盘输入没有定位元件部件，所以输入焦点不能改变。但是鼠标按键有定位部件，作为正常命令的副作用可以导致输入焦点改变——新焦点单击（new-focus click）。然而，如果你在已经有输入焦点的窗口里单击鼠标——焦点内单击（in-focus click），就不会有输入焦点的改变。

焦点内单击是正常情况，程序会按其他鼠标单击一样处理：如选择数据、移动插入点或者启用命令。新焦点单击引发了一个难题：程序将用它来做什么？程序是否应该在完成输入焦点转换后就放弃它（从功能的角度看），还是应该让它执行双重任务，先转换输入焦点，然后在程序中执行正常任务？

例如，假设屏幕上同时可以看见两个文件夹。其中只有一个可以是活动的。假定文件夹 Foo 的窗口是活动的，它有输入焦点，并且视觉上用突出显示的标题栏显示。键盘输送的信息只进入 Foo 窗口。在已经活动的文件夹 Foo 的窗口中单击鼠标——就是焦点内单击，只能进入 Foo 窗口。

现在，如果你移动鼠标光标到名为 Bar 的文件夹窗口，再单击鼠标，你在告诉 Windows 你想将 Bar 窗口变成活动窗口，接管输入焦点。这个新焦点单击会导致标题栏的颜色发生改变，表明 Bar 文件夹窗口是活动的，同时也会使 Foo 窗口无效。

问题来了：Bar 文件夹窗口应该在自己的场景中解释新焦点单击吗？假如说新焦点单击发生在可见的文件图标上，图标应该是选中，还是在完成输入焦点转换后就作废？如果 Bar 文件夹窗口已经处于活动状态，在焦点内单击相同的图标，文件将会选中。在实际中，文件也确实会选中。窗口将新焦点单击视为合法的焦点内单击。

Microsoft Windows 部分统一地将新焦点单击视为焦点内单击。例如，如果你通过单击拖动标题来改变 Word 的焦点，它不仅获得输入焦点，而且改变位置。但这就是引起不

一致的地方，如果你在 Word 内的一个文档上单击，使输入焦点转换到 Word，Word 获得了输入焦点，但这次单击作废了——它在文档中不会视为焦点内单击。

专家们对理解新焦点单击持矛盾意见，所以没有哪种策略是一定是对的。通常，忽略新焦点单击更安全一些，行动过程也更保守一些。另一方面，从用户角度看，额外的单击构成了附加的工作。如果你像 Word 及 Excel 那样选择忽略单击，那么即使交互感觉很好，也很难解释在窗口框架内的新焦点单击也被认为是焦点内单击的这种矛盾现象[6]。

元键

一起使用元键和鼠标可以扩展直接操作习惯用法。元键包括 Control 键、Alt 键和两个 Shift 键。

在 Windows 世界里，没有像苹果公司对 Macintosh 那样统一用户界面标准，从元键的用法中可以得到证明。虽然微软最后公布了元键的标准，但它的努力就像试图清除阿拉巴马（Alabama）路边的荆棘一样无效。

甚至微软也随意违背自己制定的元键标准。每个程序都“自有一套”。但还是有些方式成为了主流，尤其是那些通常首先由苹果公司定义的。不幸的是，它们的映射却并不完全相同。苹果系列有一个 Clover 键和 Apple 键大致分别与 Ctrl 和 Alt 键相对应。让你自己的界面保持一致，比记住选用哪个元键，或者你选择以哪个程序为模板重要得多。

使用光标暗示动态地显示元键的含义是个好主意。更多的程序应该这样做。当元键按下时，光标应该发生改变来反映习惯用法的新意图。

设计技巧：用光标暗示表明元键的含义。

在第 22 章和第 23 章，我们将讨论 Control 和 Shift 元键的用法和意义，以及它们如何影响选择和拖放。

---

[6] **译者注** Windows 对两种情况的处理存在矛盾。在 Word 和 Excel 中忽略新焦点单击，而在窗口框架中将新焦点单击解释为焦点内单击。

22

选　择

利用鼠标真正能做的只有两件事：挑选一个对象，以及挑选某些东西来处理这个对象。这些挑选动作指的就是选择，它们之间还有许多细微差别。本章讨论选择的不同类型，以及使用它们时应该如何进行视觉指示。

对象-动词的次序和选择

用户界面的一个基本问题是以怎样的顺序执行命令。几乎每一个命令都有一个操作，和一个或多个操作数。**操作**概括了所要发生的动作，而**操作数**指的是操作对象。**操作**和**操作数**是程序员专用的术语；界面设计师更喜欢借用语言学术语，用**“动词”**指代操作，用**“对象”**指代操作数。

你可以先指定动词，然后是对象，或者以相反的次序。通常分别称之为**动词—对象**次序和**对象—动词**次序。这两种次序在现代用户界面中都采用了。

在图形界面首次出现时，动词—对象次序带来的问题就清楚地显现出来了。在交互界面中，如果用户首先选择了一个动词，系统就必须进入一种与常态不同的状态（模式）：等待对象。通常用户随后将选择一个对象，那就不会有问题。但是，如果用户想将动词

施加于多个对象，系统如何能知道呢？只有用户提前告诉系统将有多少个对象输入，它才可能知道。但需要用户去这么做，就违背了“从来不要让用户请求许可”的公理。否则，程序必须接受所有操作数，直到用户输入了某些特殊的对象列表终止（object-list-termination）命令，这也是个非常笨的习惯用法。明白问题所在了吗？在高度结构化的语言环境里工作得非常好的操作，在宽松的交互世界里却会完全土崩瓦解。

如果把命令次序改成对象—动词，我们就不需要担心结果。用户选择要操作的对象，然后指出对对象执行的动词。软件再对所选对象执行所定制的功能。这样的好处是，用户能轻松地在同一复杂选择上执行一系列动词。但要注意一个新的概念悄悄渗透进这个综合体，而它在动词-对象世界中不存在，也没有必要存在。这个新概念就是**“选择”**。

与其让程序在用户选择一个或多个对象时记住动词，不如让程序在用户选择动词时记住一个或多个对象。但这就意味着我们需要一种机制来识别、标记和记住所选的操作数。选择就是这种机制，通过选择用户通知程序哪些对象需要记住。

对象—动词模型可能有点不好理解，但选择是一种非常容易掌握的习惯用法。只要见过一次，就很难忘记。例如在 Outlook 中用 Ctrl-单击选择多个邮件，然后删除，很快就成为根深蒂固的习惯。在英文场景中，我们必须先选择对象是荒谬的（尽管对于说德语的人也许不存在问题）。另一方面，在非语言动作中我们常用到这种模型。我们购买食品的时候，首先挑选对象——把它们放到购物车里——然后再指定执行何种操作：将购物车推到收款台表示我们希望购买。但在对话中我们从来不会说：“玉米片，买”。

在非交互界面，如模态对话框中，并不总是需要选择这个概念。对话框本身就有一个对象列表终止命令：OK 按钮。用户可以先选择功能，然后选对象，或者相反。因为直到单击 OK 按钮进行确认，整个操作都不会真正发生。这并不是说，对象—动词次序在多数对话框中不会用到，只是说明没有哪种特定的命令次序非常好，更确切地说，两种次序各有优缺点，在复杂的用户界面中相互补充。两者都是软件设计师的强大工具，应该应用到最适合的地方。

在它最简单的一种变体中，选择是微不足道的：用户用鼠标光标指向数据对象，然后单击，对象选中。然而，这种操作虽然从表面上看起来很简单，但实际中有很多有趣的变体值得我们去发掘。

离散选择和连续选择

用户通常选择数据对象，而不是动词，动词是针对对象的操作。当启动一个动词（换句话说一个命令）时，你的单击动作可能与选择数据时一样，但命令很少展示为拥有选

择状态的对象。可以选择的对象通常是那些可以操作的对象，即数据。那么选择的变体就取决于可选数据的变体。这里有两大类数据。

一些程序把数据表示为可以相对其他对象独立操作的不同视觉对象。桌面上的图标和画图程序中的图形对象就是很好的例子。这些对象也可以彼此独立地选择。它们是离散数据，针对它们的选择称为**离散选择**。

离散数据不必是同一类的，离散选择也不必是连续的。

相反，有些程序把数据表示为由许多连续的小片数据组成的矩阵。Word 处理器中的文本，或电子表格中的单元格，就是由成百上千相似的小对象组成的连贯整体。这些对象常以连续分组的方式进行选择。所以我们称之为**连续数据**，对连续数据的选择就是**连续选择**。

连续选择和离散选择都支持一次单击选择和单击-拖动选择。一次单击选择最少的离散数据，而单击-拖动选择可选择更多数据。除此之外还有其他一些显著的差别。

Word 文档中的文本有正常的次序——它由连续数据组成。文字次序不规则将影响到文档表达。字符流从头到尾以有意义的方式形成连续的整体。在数据的上下文中选择单词或段落是有意义的。而随机、不连贯的选择通常没有意义。虽然理论上不连续选择是允许的，例如选择几个不连贯的段落，但对选择进行可视化，并避免对选择无意识的不必要操作给用户任务带来的麻烦，远远超过了任务本身的价值。一般来说，如果连续数据的滚动超出屏幕，就不应该对它进行离散选择。

另一方面，离散数据没有内在次序，就像盘中的豌豆；你选择的次序和吃豌豆的次序一样并不重要。在画图程序中，屏幕上不同图形对象是独立的。图形之间的关系对于它们的含义来说没有意义。即使它们的 z-轴次序在屏幕上彼此覆盖也没关系，只有彼此直接覆盖时才具有特殊意义。无论对于什么样的集合图形来说，对象次序的不规则性都没有任何影响（同样除非对象彼此覆盖）。这些对象没有内在的次序，因此，在上下文中，连续选择没有意义，每个对象都是离散地选择。

多数画图程序提供分组功能，允许两个或多个离散对象逻辑上组合在一起，形成单个新的离散对象。这组对象无论包含多少部件，行为上就好像单个离散对象。

当然，你可以选择不止一个离散对象，但它仍然是一系列的独立选择，而不是有序数据的子集。

互斥

通常，当做出一个选择后，以前任何选择都作废了，这种行为称为**互斥**（Mutual

Exclusion），也就是一个选择排斥另一个选择。比如，用户单击一个对象，对象选中。对象保持选中状态，直到用户选择了其他事物为止。互斥对离散选择和连续选择都适用。

某些离散系统允许通过对所选对象的第二次单击取消选择。这可能导致一种奇怪的情况：根本没有东西选中，并且没有插入点。你必须决定这种情况对应用程序是否合适。

添加选择

互斥在连续选择中是必须的。因为如果用户的选择可能会轻易地滚动出屏幕，用户就不可能看见或者知道他的动作会有什么结果。想像一下，如果能够选择一篇长文档的几个独立文本段落，或许这个操作是有用的，但它的操控性并不好。是滚屏，而不是连续选择，造成的问题，但多数管理连续数据的程序都是可滚动的。

但是，如果在离散选择时关闭互斥，用户可以连续单击对象来选择多个独立对象，称之为**添加选择**（Additive Selection）。例如，列表框可以允许用户随心所欲地做出多项选择。条目在第二次单击时取消。用户在选择期望的对象后，最后的动作同时对它们起作用。

多数离散选择系统默认实现互斥，只有使用元键时才允许添加选择。Shift 元键最常用，Ctrl 元键其次。例如在画图程序中，单击选择一个图形对象后，你可以用 Shift 元键加单击在你的选择中增加其他对象。

因为前面讨论的控制问题，连续选择系统通常不允许添加选择（但可以想像有总体视图的界面能对添加选择进行管理）。但是，连续选择系统的确需要使用元键对允许的单一选择进行扩展。不幸的是，是使用 Ctrl 键还是 Shift 键还没有一致意见。在 Word 中，Shift 键使得初始选择和 Shift 键-单击之间的一切内容都选中。很容易发现具有相似的添加选择功能程序，选用的却是不同的元键变体。这种选择实际上并没有什么不同，因此在这里跟随市场潮流是最好的，因为一致性为用户提供了虽然小但实实在在的好处。

某些列表框以及 Windows 的文件视图（两者都是离散数据的例子）用添加选择做的事情有点奇怪。它们用 Ctrl 键实现“正常”的离散添加选择；但又用 Shift 键来扩展选择，仿佛它们是连续的数据，而不是离散数据。多数情况下，这种映射徒增疑惑，因为这与离散数据添加选择的习惯用法相矛盾。也很少有人会想选择列表框中的三个相邻项仅仅因为它们紧密相邻，这是连续选择扩展用法的真正意图，但在这里却被误用了。在文件视图中对这种用法还有一些争论，但成组选择（见下节）能很好地处理这个问题。Macintosh 用更典型也更恰当的 Shift 键-单击映射来在文件视图中不连续地添加选择。如果你也和作者一样，用 Mac 和 Windows 两个系统工作，那么这些不同用法增加了认知负担。

插入和替换

正如我们已经明确的，选择表明在哪些数据上将进行下一步功能操作。如果下一个功能是**写入**命令，输入数据（键盘输入或粘贴命令）将写入已经选中的对象。在离散选择中，一个或多个离散对象选中，输入的数据交给选中的离散对象，它们各自以自己的方式处理数据。这可能会导致一种“**替换**”动作，即输入的数据取代所选对象。或者所选对象将输入的数据作为素材来完成某种标准功能。例如在 PowerPoint 中，如果选择了一种形状，键盘输入的文本就会成为所选形状的文本注释。

但在连续选择中，输入的数据总是替换现有的选中数据。当你在一个文字处理器或文本输入框中输入，输入的字就会替换所选的文本。连续选择还有个特殊倾向：选择能简单明了地表明连续数据两个元素间的位置，而不是数据中特定的元素。这个位置被称为**插入点**（insertion point）。

在文字处理器中，**插入符号**（caret）（通常是闪烁的黑色线段，用来表明下一字符所在的位置）表明文本中两字符之间的位置，而实际上它并没有选择两个字符中的任何一个。通过指向和单击任何其他地方，你可以轻易地移动插入符号。但如果你拖动以扩展选择，插入符号就会消失，并换成文本的连续选择。

插入点也可以认为是一种空选择。从定义上说，向所选对象输入字符，新输入的字符就替换了所选对象，但如果选择是空的，新的字符没有替换任何东西，它们仅仅是插入。换而言之，插入是替换的一个特例。

电子表格中也使用连续选择，但它的实现与文字处理器有些不同。单元格组成一个连续的数据矩阵，所以选择是连续的，但是没有选择两个单元格之间空间的说法。在电子表格中，单击就是选择整个单元格。在电子表格中通常也没有插入点的说法，虽然这样的设计可能会很吸引人（这就是说，选择两个垂直相邻的单元格顶部和底部之间的线，然后插入一行或填充一个单元格）。

将这两个习惯用法混合也是可能的。PowerPoint 中的幻灯片 Slider sorter 视图允许选择插入点，但单幅幻灯片也能进行选择。如果你在幻灯片上单击，你就选定了那张幻灯片，但如果你在两张幻灯片之间单击，就会出现一个闪烁的插入点符号。

如果程序允许插入点，就必须通过单击和拖动来选择对象。即使只是选择文字处理器中的一个字符，也必须拖动鼠标经过它。这就意味着用户在应用程序的正常过程中，将要做很多单击和拖动的动作，这里的副作用是使任何拖放习惯用法更难表达。你可以在 Word 中体会到这些，在 Word 中拖放文本首先涉及到一个单击和拖动操作以做出选择，然后鼠标回到被选文本，再次单击和拖动进行实际移动。做同样的事，Excel 让你在所选

单元格的边界发现特殊的受范区（仅一两个像素宽）。为了移动一个离散选择，用户必须在单个动作里完成对对象的单击和拖动。

为了减轻文字处理器中选择的单击和拖动负担，也可以用其他的直接操作捷径，如双击选择单词。

成组选择

单击和拖动（click-and-drag）操作也是成组选择的基础。对于连续数据，它意味着从鼠标按下到鼠标释放都属于扩展选择。这也可通过元键调整，例如在 Word 中 Ctrl 加单击选择一个完整的句子，那么 Ctrl 加拖动就可以逐句选择。独占式应用程序应该使用这些合适的变体来丰富它们的交互。经验丰富的用户最终会记住和使用它们，只要变体手工操作起来很简单。

在离散对象的集合中，单击和拖动操作通常开始一个拖放移动（drag-and-drop-move）动作。如果鼠标按钮是在两个对象之间单击，而不是在任何一个特定对象上，就有一种特殊的含义。它产生了一个如图 22-1 所示的**拖动矩形**（drag rectangle）。

拖动矩形的大小是动态变化的，它的左上角是鼠标被按下的位置，其右下角是鼠标释放的位置。当鼠标释放时，包含在拖动矩形内的任何对象都作为一组进行选择。

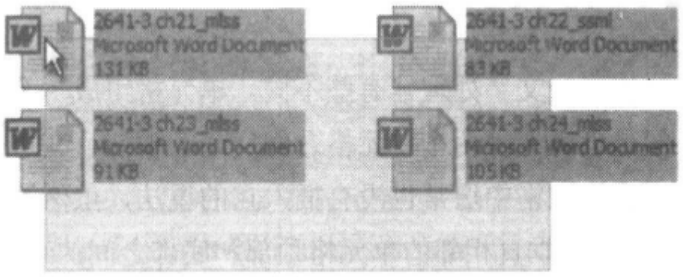


图 22-1 当鼠标按下时，光标不在任何特定的对象上，单击和拖动就产生一个拖动矩形。鼠标释放后包含在矩形里的任何对象都被选定。画图程序和许多文字处理器的用户可能很熟悉这个习惯用法。这个例子取自 Windows XP，在同一个文件夹里选择文件。矩形从右下角被拖动到左上角。

选择的视觉提示

必须清晰、醒目地向用户指出选中的对象。选中状态在拥挤的屏幕上必须容易认出、明确，并且不会使对象通常可见的细节变得模糊。

设计技巧：使选择在视觉上明确醒目。

特别是，你必须确保用户能轻易地辨别哪些项目已经选定，哪些项目没有选定。仅仅让人能看出项目不同是不够的。必须记住，有相当一部分的人是色盲，所以不能单纯用颜色区分不同选择。

反色

反色经常用于完成选择——反转已选对象的像素。在单色屏幕上，这意味着将所有白色的像素变黑和所有黑色的像素变白。但是还有多少人还在使用黑白显示器呢？在 1984 年，最初的 Macintosh 面世的时候，实质上和硬件构造上都是单色显示计算机。因此，苹果公司认为使用反色技术显示选择是恰当的。反色是通过对选择的像素与所有的位为 1 的元素（依赖于处理器，或者所有位为 0）进行异或操作来实现的。异或恰恰是 CPU 执行起来最快的操作，再加上 1984 年当时计算机能力的限制，这是最好的选择。因为布尔逻辑的特性，异或天生速度很快，也能够由重复的异或撤销。微软在 Windows 第一版中延用了异或技术，尽管无论在人们的意识中还是实际上它都从来不是单色显示系统。

异或操作的结果是只有当操作数是二元的时候才能定义：开或关，1 或 0，黑像素或白像素。对于彩色视频进行异或并不合适。的确，不同视频驱动器的颜色定义也不同。蓝色的反色是什么颜色？艺术课上，它是黄色；可是在布尔代数学里，谁知道它是什么？况且，颜色本身带有信息，让用户的视频硬件成为颜色代表含义的判别器是愚蠢的。最后，某些反色可能导致选择变得模糊不清，这是不惜一切代价都要避免的。

微软的 PowerPoint 色彩丰富，幻灯片视图也很少是单色的。当选中了文本对象中的字符用于编辑时，背景就变黑了，对实际字符进行了反色。一致的黑色背景是可以的，而反色的像素有时也没问题。但是，当文本（和通常一样）是白色时，它的背景变黑了，字符也从白色反转为黑色，这就造成编辑困难。程序应该确保文本选中时，任何情况下都容易辨认。

在控件方面，对选择进行反色会产生更多美学和识别方面的问题。例如，在 Windows 的早期版本中，如果用控制面板程序来配置你的屏幕颜色，你设定菜单的颜色为黄色，而不是灰色，它们会反色为蓝色。这当然很容易注意到，但由于引入的识别问题，它并

不是我们一定需要的。

微软意识到了这个问题，Windows 3.0 中定义了两个新的系统颜色设置：color-highlight 和 color-highlight text。当然，这些显示常数代表的是可变颜色，而不是固定颜色。每个用户能改变这些变量的定义，这样可以与所有的应用程序保持一致（直到用户再次将其改变）。连同这些新颜色一起形成相应的应用标准：当一个对象选中时，它的颜色变成 color-highlight 代表的颜色，所选对象的任何文本或与之对照的像素变成 color-highlight text 代表的颜色。如果和文字处理器一样，选择是连续的，那么背景就成了 color-highlight，背景上的文本变成 color-highlight text。这种新标准使选择的视觉行为在彩色平台上正常化了。Mac OS 和大多数 UNIX 窗口系统有相似的设置。

设计技巧：用系统突出显示颜色来显示选择。

选择彩色对象

在画图、绘画、动画和演示文稿程序中我们处理的是彩色对象，颜色很容易因为选择而丢失。最合理的解决办法就是在对象上加选择指示器，而不是改变所选对象的视觉特性来显示选择。多数图形艺术程序采用这种方法，为所选对象加上不会使对象本身模糊的视觉框（visual scaffolding）。这通常通过操作点来处理（在 24 章讨论）：给所选对象周围加上一个小框，提供控制点。在混乱中，特别是在复杂的图形操作程序中，操作点仍然可能丢失，但是有一种方法能保证所选对象无论用什么颜色总能可见：通过移动来指示选择。

Macintosh 首批程序之一的 MacPaint，有一个很好的习惯用法：用简单的虚线描画出被选对象的轮廓，所有的虚线围绕对象同步移动。虚线看上去就像一队蚂蚁，因此这种效果赢得一个形象绰号：行进中的蚂蚁。（也叫做选取框，Marquee，通过对以前电影招牌的理解来展示相似的行为。）

Adobe Photoshop 使用这个习惯用法显示选中的图画区域，它工作得很好（专业用户可以用键盘让它们出现或者消失，这样他们可以没有任何干扰地观察对象）。尽管要小心，动画也不难做到。无论背景密度多高，颜色多复杂，行进中的蚂蚁都能做到。

既然已经理解了选择的基础，你可以进一步了解如何用直接操作来控制选中的对象。本篇的其余各章将讨论拖、放，以及其他可视化操作所选对象的方式。

23

拖　放

在图形用户界面所有的直接操作习惯用法中，拖放操作——在屏幕上移动对象的同时，单击和按住鼠标按钮——的定义最多。令人奇怪的是，拖放的使用并不像我们想像的那样广，而且它的确没有实现所有的潜能。特别是在 Web 流行，以及和 Web 类似的行为成为超级易用性的同义词后，更妨碍了拖放在桌面系统上的发展，因为开发人员错误地在其他不合适的上下文中模仿着浏览器的交互。

拖放的定义

任何鼠标动作都非常有效率，因为它在用户的单个动作中包含两个命令成分：定位功能和一个特定的功能。拖放有双倍效率，因为在一个流畅的动作里加上了第二个定位功能。虽然拖放被视为现代图形界面的一个基石，但值得注意的是，除了画图和绘画专用程序以外，其他程序很少用到拖放。好在随着更多的程序使用这一习惯用法，情况看起来有所改变。

我们可以将拖放定义为“点击一个对象，并将它移动到另一个位置”，虽然对概括一个如此广泛的习惯用法来说，这个定义有些狭窄。对拖放更精确的描述是：单击某个对

象，移动它以暗示一种变换。

Macintosh 是第一个成功提供拖放的系统。它引起了对这个习惯用法的许多期望，而这些期望因为两个简单的原因一直没有实现。首先，拖放不是一个系统范围内的工具，而是程序 Finder 的一个副产品。其次，当时 Mac 只是单任务计算机，在应用程序之间进行拖放的问题多年没有表现出来。

要感谢苹果公司的是，他们在第一版用户界面标准指南中对拖放作了概括。而在另一面，微软不仅没有在早期 Windows 版本中使用拖放，而且甚至没有在程序文件中描述它的过程。但是，微软最终迎头赶上，并且开拓了一些新的用法，比如活动工具条（movable toolbar）和可停靠调色板（dockable palettes）。

内部拖放和外部拖放

从根本上说，你可以在程序内将一个事物从一个地方拖放到另一个地方，也可以把某物从你的程序拖放到其他程序。根据它们之间的不同，分别称之**为内部拖放**和**外部拖放**。

内部拖放无论从概念角度还是从编码角度来说都相当简单。而外部拖放显然需要更完善的支持，因为涉及到的两个程序都必须支持同样的概念，并且必须以兼容的方式实现这些概念。

理解**源—目标**范例对理解拖放习惯用法的两个变体都很重要。

源—目标范例

当用户为了实现某项功能，单击一个离散对象，并将它拖动到另一个离散对象时，这一幕的背后就是源—目标范例的体现。

拖动发起时所在的对象控制着整个过程：它就是**源对象**，一个入口[1]。如果你拖动的是一个图标，图标就是它自己的入口。如果你拖动的是一段文本，包含文本的编辑器就是这个入口。当用户释放鼠标按钮，被拖动的东西就会放在某些**目标对象**上。

源—目标这个术语的主要目的是区分这种操作与画图和绘画程序中的拖放操作，在那些程序中，工具和图形对象在一块打开的画布上拖动。源—目标是逻辑对象的操作代表着幕后处理的一种习惯有法。源—目标拖放最常见的形式是在程序管理器或 Macintosh

[1] **译者注** 见第 24 章的译者注 1。

Finder 中，将图标从一个文件夹移到另一个文件夹。

拖动数据对象到目标功能

你可以将文件或文件夹拖到一个代表功能的控件，而不是将它们拖到另一个文件夹。在我们熟悉的 Macintosh 垃圾箱（trashcan）中，可以证明，这种习惯用法是直接操作最著名的例子，Windows 用回收站模仿（Recycle Bin）了这一习惯用法。当我们用更好的面向对象方法创建软件时，可以把对象拖放到代表删除功能的图标或控件，而不直接删除。苹果公司将这个理念扩展到打印机上。作者注意到文件压缩程序也是以这种方式实现的。在这种习惯用法的开拓方面还有很多创新空间。

这里需要注意的是，上段中的所有习惯用法都涉及到外部拖放，因为目标对象都是独立的程序。在同一个程序里，代码知道哪些对象可以拖动——经常只有一种类型——任何目标控件都很容易处理它。对于外部拖放，源对象可以来自任何程序，而目标控件对源程序和拖放的对象可能没有任何直接知识。目标程序必须在不需要理解拖放对象及其内容的情况下，以某种合理的方式处理未知对象。例如，文件夹能够这么做，因为它只能处理文件。假如交给它的是来自文字处理器的一段文本，它会怎么做呢？如果它不能处理文本，它并不真正具有外部能力。感谢微软，Windows 的回收站真的能接受从 Word 中拖出的段落文本，也能接收来自 Excel 的单元格。

对于一个真正具有**外部能力**的对象，它必须能接收来自任何对象的任何内容，不管源程序是什么。乍听上去，好像实现起来会非常复杂，其实不必那样。主要是要定义接口标准。当把数据拖到一个对象，目标对象要说的只是，“是的，我可以接受拖放”。接着，两个对象必须就格式问题进行磋商，因为不可能期望每个对象都能以其他所有程序专有的格式接收数据。如果源对象是 Excel，比如说最开始它以内部格式提供数据，另一个微软程序可能知道如何解释这种格式，但 X 品牌的产品可能不知道。所以 X 品牌的目标对象客气地提出异议——不是对拖放提出异议，而是对拖放内容的格式提出异议。于是源对象 Excel 必须以更一般的格式（SYLK，CSV，ASCII）再次提供数据。目标对象也许不能接受 SYLK 或者 CSV，但按照惯例，它一定会接受 ASCII，ASCII 是所有平台上最通用的格式。每个具有外部能力的对象都必须至少接受 ASCII、简单的位图、文件指针和我们即将见到的——功能。希望在开放市场获得成功的对象会接受比这更多的格式，但起码这四者可确保与任何其他对象的兼容性。举例说，即使是音频文件，最终也能作为一个简单的磁盘文件指针进行传递。支持这种格式协商的外部拖放协议是**协商拖放协议**（negotiated drag-and-drop protocol）。不能协商拖放格式的协议只能接受已知格式，

因此称之为**已知格式拖放协议（known-format drag-and-drop protocol）**。

拖动功能对象到目标数据

适当的、协商的、外部拖放能力包括把功能拖放到目标数据，也包括把数据拖放到目标功能。在处理连续数据时，定义这类动作的范围可能会引起一些疑问，同时，也会产生动词—对象次序和终止等问题。但在作者看来，这个习惯用法值得进一步研究。

虽然 OLE 和 ActiveX 声称提供这种协议，但到现在为止，还没有一个标准的外部拖放协议，当然也没有协商拖放协议。

源—目标交互

一个设计巧妙的对象会在视觉上提示它的受范性，或者以静态的绘画方式，或者在光标经过时以主动的动画方式。

对象能够拖动的理念很容易学习。在见过可以这么做之后，用户很难忘记图标、选中的文本和其他截然不同的对象能够进行直接操作。他们也许会忘记动作细节（所以在用户单击对象之后，其他的反馈形式非常重要），但直接操作受范性很容易记住。第一次使用的用户和不经常使用的用户可能需要一些额外的帮助。这种帮助可以来自额外的训练，也可以来自内建在界面中的建议。总之，交互友好的程序鼓励用户对程序中的不同对象使用直接操作。

一旦用户在某个对象上单击鼠标按钮后，那个对象就成为拖放过程中的源对象。另一方面，还没有对应的目标对象，因为释放鼠标的位置还没有确定：它可能是另一个对象，也可能是对象之间的开放空间。但是当用户按住鼠标移动的时候，光标会经过源对象程序内部或者外部的一系列对象。如果这些对象支持拖放，它们就可能是目标对象，称为**拖放候选对象（drop candidate）**。

一次拖动只能有一个源对象和一个目标对象，但可以有很多拖放候选对象。根据拖放协议，拖放候选对象可能不知道如何接受特定的拖放内容；它只需要知道如何接受提供的拖放协议。其他协议可能需要候选对象立即识别能否有效处理所提供的源对象。后面这种方法更慢，但能向用户提供更好的反馈。然而，如果协议需要在源对象和每个拖放候选对象之间进行广泛的协商，交互就可能变得非常缓慢，从这点上看也许这样不值得。

拖动时的视觉反馈

每个拖放候选对象惟一的任务就是视觉上表明捕获光标的热点经过了它，也就是说，如果用户释放鼠标按钮，它将接受拖放，或者至少理解它。这种指示本质上是主动视觉暗示。

设计技巧：拖放候选对象必须在视觉上显示它们的接受能力。

视觉上提供拖放接受能力的最弱方式是改变光标。光标的工作是代表被拖动的事物。因此，最好是将对所有拖放候选对象的指示留给拖放候选对象本身。

设计技巧：拖动光标必须在视觉上表示源对象。

不混淆这两种视觉功能很重要。不幸的是微软在 Windows 中就犯了这个错误。这种决策考虑更多的是容易编码，而不是设计。在显示拖放接受能力方面，改变光标比突出显示拖放候选对象要简单得多。光标代表拖动的对象，不应该用于表示拖放候选对象。

这好像还不是最坏的，微软还用讨厌的带斜杠的圆形进行光标暗示，通常这个图标表示不允许。

这个符号是一个令人不愉快的习惯用法，因为它告诉用户不能做什么，是一种负面反馈。用户容易误解为："现在不要移动鼠标，不然将有无法挽回的损失"而不会理解为"释放鼠标，不会有什么事情发生"。无论微软的风格指南如何解释，把表示不允许的符号添加到光标暗示中，是这两种弱习惯用法的糟糕组合。

在用户最后释放鼠标按键后，当前的拖放候选对象就成为目标对象。如果用户在有效的拖放候选对象之间，或无效拖放候选对象上释放鼠标，那么就没有目标对象，拖放操作结束，并不做任何动作。没有声音或视觉上没有变化，是表示这种结果的好方式。确切地说，它不是一种取消，所以也就不需要显示取消符号。

【指示拖动受范性】

用主动光标暗示来显示拖动受范性这种解决方法是有问题的。在日益面向对象的世界里，能够拖动的事物越来越多。光标闪烁和迅速变化更多的是造成了视觉混乱，而不

是帮助。一个解决方法就是假定事物能拖动，然后让用户体验。这种方法在 Windows 浏览器和 Macintosh Finder 的窗口中获得了成功。没有光标暗示，拖动受范性可能是一种难以发现的习惯用法，所以你可以考虑在界面中建立其他一些指示，也许是文本暗示，或者工具提示那样的弹出菜单。

在选择好源对象开始拖动操作后，必须有某些视觉指示。视觉上最生动的方法是使拖动操作完全活动起来，实时地显示整个源对象的移动。但是，这种方法难以实现，速度太慢，所以很可能不是一个恰当的解决方案。问题在于源和目标操作需要相当精确的指针。例如，源对象也许有六平方厘米，但它必须拖放到只有一平方厘米的目标对象上。源对象不能模糊目标对象，因为源对象大到横跨多个拖放候选对象，我们需要用光标热点精确显示它将拖放到哪个拖放候选对象上。这就是说，在源和目标操作中，拖动源对象的透明轮廓要比拖动源对象完全动画的精确图像要好得多。同时这也意味着被拖动对象不会模糊正常的箭头光标。箭头尖端仍需指示确切的热点。

当轮廓相对源对象移动时，拖动轮廓对大多数重新定位也是恰当的，在原始位置仍然可以看到它。

【指示拖放候选对象】

当光标带着源对象的轮廓在屏幕上移动时，经过一个又一个的拖放候选对象。这些拖放候选对象必须在视觉上显示它们已经意识到自己正被当作可能的拖放目标。通过视觉改变，拖放候选对象提醒用户它能够正确处理拖放的对象。

有一点是那么明显，以至于难以发现，也就是只有那些成为拖放候选对象的对象才是当前可见的。一个正在运行的应用在不可见的情况下，没有必要可视化地显示它能够成为目标对象。通常，占据屏幕空间的对象数目很少，最多两打。这意味着实现负担不会过大。

在内部，源对象在经过每个拖放候选对象时，应该与相应的每一个通信。简略地会话应该发生，源对象询问目标对象，它是否能够接受拖放。如果能够，目标对象用视觉暗示显示。目标对象也需要知道什么时候光标离开它的空间范围，这样它能够关闭视觉暗示。

微软不仅不坚持拖放候选对象使用视觉暗示，而且它建议改变光标就很充分。这种决定对于用户来说是不幸的。用户很难理解当前拖动的对象是什么，目标对象是什么，是否目标对象支持源对象的拖放。至少在桌面上，Windows 图标当前通过视觉上突出显示来正确显示它们的拖放候选对象。这是否是 Macintosh Finder 对 Windows 桌面的影响，或者是源对象和目标对象拖放工作方式的标准？我们希望是后面这种情况。

完成拖放操作

当源对象最终拖放到一个候选对象时，该对象就成为真正的目标对象。这时，源对象和目标对象必须进行更详细的对话（相对源对象与其他拖放候选对象之间）。毕竟，用户已经执行了操作，目标对象已经确定。目标对象也许知道如何接受拖放，但那并不意味着它能很好地接受在这次操作中拖放的特定源对象。

这种更详细的对话隐含着该转移可能会失败。那是可以的，显示拖放接受能力和实际拖放失败，比在存在可能成功的情况下不指示拖放接受能力要更好（如果遵守最小的通用格式标准，就不会有物理失败）。如果拖放需要协商，转移格式有待解决。如果是信息转移，源对象和目标对象可能希望协商转移的信息是否采用两者都知道的某种专用格式，或者不得不降低数据精度，转换成某种更弱但更通用的格式，如 ASCII 文本。

插入目标

当用户在屏幕上拖动源对象时，它指向的每个拖放候选对象都在视觉上发生改变，显示它们的拖放接受能力。在某些程序中，源对象可能拖放到其他对象之间的位置。Word 中拖动文本就是这样的操作，在列表和数组中重新排序操作也是这样。

拖放的重要视觉反馈就是如果用户释放鼠标按钮，会显示源对象会落在何处。在源—目标拖放中，拖放候选对象在视觉上突出显示以显示潜在的拖放落点，但在**重新排序拖放**中，潜在拖放落点处根本没有对象。视觉暗示出现在程序的背景或者连续数据中：**插入目标**（insertion targets）。

在 PowerPoint 的视图（slide-sorter）中重新排列幻灯片就是这种类型（内部）拖放的一个典型例子。用户可以选择一幅幻灯片，并将它拖动到不同的位置。在拖动时，插入目标在幻灯片之间出现（一个垂直的黑条，看上去像大的文本编辑符号）。当你拖动文本时，Word 也会显示一个插入目标。不仅加载光标会显现出来，你还可以在两个相邻字符之间看见一个垂直的灰条，灰条显示了精确插入拖放文本的位置。

无论是把什么对象拖放到其他对象之间，程序必须显示一个插入目标。就像源和目标拖放中的拖放候选对象一样，程序必须在视觉上显示拖动的对象可以在什么地方放下。

完成时的视觉反馈

如果目标对象和源对象能达成一致意见，合适的操作就会发生。在这一点上，重要的步骤是视觉反馈能反映操作已经发生。如果操作是一种转移，源对象必须在源位置消失，出现在目标位置上。如果目标对象代表功能，而不是容器（如打印图标），图标必须视觉上暗示它接收了拖放，并且正在打印。这可以使用动画或视觉状态改变来实现。

视觉丰富的源和目标拖放操作，是 GUI 设计师戏法袋里最有魔力的操作。如果工具开发商更好地支持这种习惯用法，它将在应用程序开发者中流行，用户也将受益。

其他拖放交互问题

当我们第一次见到拖放习惯用法，它看上去似乎很简单，但对于常用用户和一些特殊情况来说，它会带来一些并不简单的问题和困难。通常，在软件设计的迭代优化过程中已经暴露了这些缺点，本着发明的精神，聪明的设计师设计了聪明的解决方法。

自动滚屏

当选中的对象拖出封闭应用程序的边界时，程序应该如何解释？当然，正确的解释是对象正在拖动到一个新位置，但新位置是在封闭应用程序之内还是应用程序之外呢？

以 Microsoft Word 为例，当一段选中文本拖动到可见的文本窗口之外时，用户是说“我想将这段文本放到另一个程序中”还是“我想将这段文本放到同一文档的其他位置，但这一位置目前不在屏幕当中？”如果是前者，我们可按前面讨论过的继续。但如果用户期望的是后者，应用程序则必须朝着拖动的方向自动滚屏，定位到一定距离的位置，而不是这个文档中当前可见的位置。

自动滚屏是拖放操作重要的补充。当你实现一个操作时，可能不得不实现另一个操作。无论拖放的目标对象是否在屏幕之外，程序都需要自动滚动。

设计技巧：任何可滚动的拖放目标对象都必须支持自动滚屏。

在早期的实现中，如果你拖到应用程序窗口之外，自动滚屏就起作用。这样有两个

致命的缺点。首先，如果应用程序充满整个屏幕，如何获得应用程序以外的光标？其次，如果你想拖动对象到另一个程序，应用程序该如何分辨这种情况与自动滚屏呢？

针对这个问题，微软开发了非常聪明的解决方法。基本上，它恰恰在应用程序的边界内，而不是到边界外才开始自动滚屏。当拖动光标靠近可滚动窗口的边界时——但仍然在窗口内——朝着拖动方向的滚屏就开始了。如果拖动光标来到文本区底部的三到四毫米之内时，Word 开始向上滚动窗口内容。如果拖动光标接近文本区顶部边界时，Word 就向下滚屏。遗憾的是，Word 的实现没有考虑到处理器的性能，在作者的计算机上，动作太快了以致不起作用（公平的说，有些微软产品自动滚屏实现得确实很好）。除了补偿处理器速度，如图 23-1 所示，实现这个习惯用法的更好方法是使用不同的自动滚屏速度。随着光标距窗口边缘越近，自动滚屏速度就越快。例如，当光标距上边缘 5 毫米时，文本以每秒一行的速度向下滚动。到 4 毫米时，文本以每秒两行的速度滚动，以此类推。这就让用户可以有效地控制自动滚屏。自动滚屏总是要进行调整，因为计算机只会越变越快。

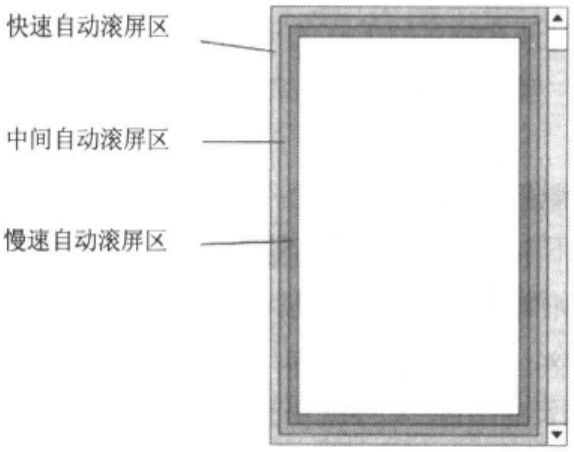


图 23-1 遗憾的是，微软以计算机能够实现的任意速度进行自动滚屏，至少对用户来说，这种速度太快以致无法控制。不仅应该设置自动滚屏的最大速度，移动也应该进行分级，并且应该保证用户可控性。自动滚屏应该随用户距窗口边缘变近而加速。微软做得很好，当光标接近封闭的滚动框内边界，而不是边界外时，自动滚屏的想法的确非常聪明。

自动滚屏需要的另一个重要细节是时间延迟。如果光标一进入边界周围的敏感地带，就立即开始自动滚屏，那么对于动作慢的用户，就太容易在无意中启动自动滚屏。为了防止这种事情发生，应该只有在拖动光标进入自动滚屏敏感带时，对时间进行合理缓冲——约半秒钟——后再开始启动。

如果用户拖动光标，完全超出 Word 的可滚动文本窗口时，就不会发生自动滚动。相反，重新定位操作会在非 Word 的其他程序中结束。例如，如果拖动光标离开 Word 进入 PowerPoint，当用户释放鼠标按钮时，选中的对象就会粘贴到鼠标指定位置的 PowerPoint 幻灯片中。进一步，如果拖动光标移动到 PowerPoint 编辑窗口任意边界的三到四毫米以内，PowerPoint 就会朝合适的方向自动滚屏。这是个非常方便的特点，因为当前屏幕的限制，我们会发现自己经常拖着一个加载的光标，却没有地方可以放置拖动的内容。

避免拖放抖动

当一个对象既可能选择，也可能拖动时，鼠标偏向于选择操作，这非常关键。因为单击时光标不偏移一两个像素非常困难，频繁地选择某物的动作，一定不要让程序错误地把该动作理解为拖放的开始。用户很少想在屏幕上拖动一两个像素。执行拖动所花费的时间通常比执行选择所花费的时间要多。拖动常伴有刷新屏幕的操作，导致屏幕上的对象闪烁（尽管随着计算机速度加快，这种现象会消失）。这种意外的视觉干扰会影响用户期望的简单选择。此外，对象现在可能移动了几个像素。对象可能已经在用户希望的位置上了，所以即使一个像素的移动也不会让用户高兴。

在硬件方面，像按钮那样的控件，有机械接触时就会表现出工程师称之**为颤动**（bounce）的特性，即当有人按压开关的小金属触点，它们就会颤动。对于门铃这样的电路，颤动花去几个毫秒不会有什么关系。但在现代的电子仪器中，这种额外的点击问题重大。如果这种额外的转换发生在第一次操作之后的几毫秒之内，支持这种开关的电路有忽略的它们特殊逻辑，这就避免了在你打开音响后的千分之一秒后又将它关闭的现象。这种情况和鼠标过分敏感的问题有点像，解决的方法也模仿开关制造商，使鼠标**去颤动**（debounced）。

为了避免这种情况，程序应该建立**拖动阈值**（drag threshold）。除非移动超过最小阈值，比如说三个像素，否则，忽略掉鼠标按下事件后的所有鼠标移动消息。这就提供了一种保护机制，避免无意中激活拖动操作。如果用户能够保持鼠标按钮在按下位置的三个像素之内，整个的单击动作被视为选择命令，而所有小幅度的、无意识的移动将被忽略，对象不再颤动。鼠标移动一旦超过三个像素的阈值，程序就变为拖动操作，如图 23-2 所示。无论什么情况下对象选择或者拖动时，拖动操作都应该去颤动。

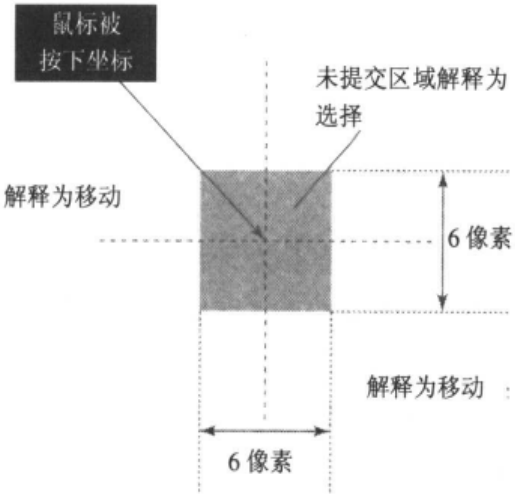


图 23-2　能够同时被选择和拖动的任何对象必须去颤动。当用户单击对象，动作必须解释为选择，而不是拖动，即使是用户在单击和释放期间不小心将鼠标移动了一两个像素的距离。程序必须忽略任何鼠标移动，只要它呆在不受约束的区域（每个方向不超过三个像素）。从鼠标被按下的坐标位置移动超过三个像素，就会变成拖动操作。对象也被认为处于操作中。这就是所谓的拖动阈值，它用于对鼠标去颤动。

设计技巧： 对所有拖动去颤动。

Windows 3x 的程序管理器以一个像素为阈值，这个阈值太小。在你只想选择一个图标时，很容易不经意地移动它的位置。Windows XP 桌面的图标以三个像素为阈值。

某些应用程序可能需要更复杂的拖动阈值。三维应用常需要支持屏幕上三个投射轴上移动的拖动阈值。另一个例子来自我们为客户设计的报告产生器。用户可以通过水平拖动来重新定位报告中的各栏。例如，它可以将“姓”从列表中的任何位置拖动，重新定位到“名字”之后。到目前为止，这是最常用的拖放习惯用法，还有另一个不常用的拖动操作，这种操作允许一栏中的值直接插入到另一栏中——如图 23-3 中的地址栏和州名栏。

Before

	NAME	ADDRESS	CITY
2	Ginger Beef	342 Easton Lane	Waltham
3	C. U. Lator	339 Disk Drive	Borham
4	Justin Case	68 Elm	Albion
5	Creighton Barrel	9348 N. Blenheim	Five Islands
6	Dewey Decimal	1003 Water St.	Freeport

After

	NAME	ADDRESS/CITY	
2	Ginger Beef	342 Easton Lane Waltham	
3	C. U. Lator	339 Disk Drive Borham	
4	Justin Case	68 Elm Albion	
5	Creighton Barrel	9348 N. Blenheim Five Islands	
6	Dewey Decimal	1003 Water St. Freeport	

图 23-3 这个报告生成器程序有一个有趣的特点。它能够通过拖放操作将一栏中的内容插入到另一栏中。这种直接操作行为与常用的对各栏重新排序的拖放操作（比如将城市栏移到地址的左侧）相冲突。我们使用一种特殊的双轴拖动阈值来解决这个矛盾。

我们想遵照用户的心智模型，让他们可以进行这种叠加操作，将一栏的值拖动到另一栏中。但是这种操作与单纯的各栏水平重新排序相冲突。我们通过区分水平拖动和垂直拖动来解决该问题。如果用户将某栏向左或右拖动，就意味着他将该栏作为一个单元重新定位。如果用户上下拖动某栏，则将该栏的值插入到另一栏。

因为水平拖动是用户的主要动作，而垂直拖动较少，所以我们将拖动阈值向水平轴倾斜。我们创建了如图 23-4 所示的线轴形区域取代了方形区域。设定水平移动阈值为四个像素，它不会将大的移动误认为用户正常的水平移动，同时仍可避免用户不经意的垂直移动。为了确认极少用的垂直移动，用户必须沿垂直轴移动八个像素，并且左右偏离不超过四个像素。这个动作相当自然，也容易学习。

这种轴向不对称的阈值在其他方面也能用到。Visio 用了一个相似的习惯用法区分画直线和曲线。

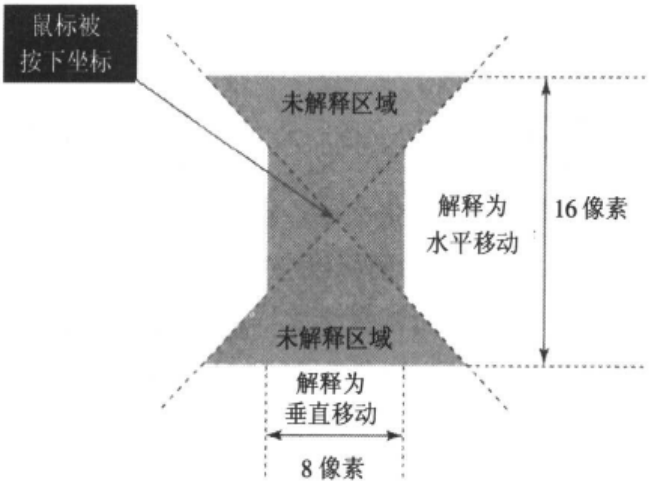


图 23-4 在客户的程序中，这种线轴形的拖动阈在水平拖动上允许偏差。迄今为止，在这种应用程序中水平拖动是最常用的拖动类型。这种拖动阈使用户很难不经意地开始垂直拖动。但是，如果用户确实想垂直拖动对象，一个无论是向上还是向下的明显活动都将导致程序在极小附加工作的情况下确认这种垂直模式。在这种方法创建之前，垂直移动需要使用工具条图标暂时改变模式。

鼠标游标

作为一个精确的指向工具，鼠标的缺陷显而易见，特别是在画图程序中拖动对象的时候特别明显。要想将一个对象拖动到确切的位置上非常困难，特别是当显示屏的分辨率为每英寸 100 个像素以上，而鼠标以 6 比 1 的比例在显示屏上移动时。为了移动光标一个像素，必须精确地将鼠标移动六百分之一英寸，这不容易做到。

解决这个问题的方法就是添加鼠标游标功能。那样用户可以迅速地转换到某种模式，允许以鼠标为基础的对象操作达到更高的分辨率。

在拖动期间，如果用户认为他需要更精确的调整，可以改变鼠标移动与显示屏上对象移动的相对比值。任何要求精确对齐的程序必须提供游标工具。这至少包括所有绘图和绘画程序，演示文稿程序和图像操作程序。

设计技巧：任何要求精确对齐的程序必须提供游标工具。

这种习惯用法有几种可以接受的形式。在拖动操作期间，按下某键，比如 Enter 键，鼠标改变成游标模式。在游标模式时，鼠标每移动十个像素，对象会移动一个像素。

另一个有效的方法是在拖动操作期间激活箭头键。维持鼠标按钮按下的状态下，用户可以操控箭头键将所选对象向上、下、左或右每次移动一个像素。释放鼠标按钮时，拖动操作将立即终止。

这样的游标存在一个问题，那就是释放鼠标的简单动作常会使用户的手移动一两个像素，导致在这个瞬间，放置完美的对象偏离了对齐。这种问题的解决方法是在接受第一个游标按键时，使鼠标不敏感。也就是使鼠标忽略在某个合理阈值（如五个像素）内的所有后继移动。这就意味着用户可以利用鼠标粗略地开始移动，然后利用箭头键精细地确定终末位置，而且在释放鼠标时也不会干扰位置。如果用户希望在游标开始后，做额外的粗略移动，他只需将鼠标移动超过阈值，系统就将脱离游标模式。

如果箭头键不是在界面中，如在画图程序中，它们能用于控制所选对象的游标移动，这意味着用户不必维持鼠标按钮按下的状态。Adobe Illustrator、Photoshop 以及 PowerPoint 就是这样。在 PowerPoint 中，箭头键将所选对象在网格中移动一步——在默认网格的设置状态下，一步约为 2 毫米。如果在使用箭头键的同时按住 Alt 键，每按下一次箭头键，对象移动一个像素。这样做相当不错。

24

操作控件、对象和连接

前两章中，我们讨论了两种重要的直接操作类型：选择和拖放。本章中，我们将讨论其余几种直接操作：控件操作、对象操作和对象连接。

控件操作

通过考察所需要的鼠标行为是**单击**还是**单击-拖动**，我们可以将针对控件的直接操作进行分类。

除了简单的单击控件，大多数直接操作习惯用法都需要单击-拖动操作。这是使用鼠标进行视觉交互的基础，下面我们将对一些细节进行探究。

单击—拖动

在用户单击鼠标键，并在不释放的情况下移动它时，拖动开始。当用户刚开始单击鼠标键时，显示屏上光标所处的坐标位置称为**鼠标-按下点**，而用户释放鼠标键时，光标的坐标位置称为**鼠标-释放点**。鼠标-按下点在任何直接操作的过程中都是确定的。鼠标-

释放点只有在过程结束时才能确定。

拖动开始后，用户和计算机之间的整个交互进入一种特殊的状态。在程序员行话里，我们称系统与用户之间所有的交互被**捕获**，意思是如果拖动操作没有完成，其他程序就不能与用户进行交互。

当鼠标单击了某程序（该程序接收了鼠标-按下事件）之后，用户对鼠标、键盘或其他任何输入设备的操作都将直接作用于该程序——从技术上说是入口[1]。拥有鼠标-按下点的入口，称为**源对象**。如果源对象是连续的数据或控件，那么对它施加的拖动操作意味着选择范围的扩展或控件状态的改变。但是，如果源对象是离散的对象，那么对它施加拖动操作更可能意味着拖放等直接操作的开始，在这种情况下，整个过程主要涉及对交互的捕获[2]。

技术上，无论鼠标在两个动作之间移动的距离有多远，在用户单击鼠标键时捕获状态就立即存在，直到鼠标键释放，捕获状态才告结束。对人来说，简单的单击-释放动作在未作任何移动时似乎是一瞬间的事，但对程序来说，在按钮单击和释放之间可以执行成百上千条指令。如果用户在释放按钮之前不经意地移动了鼠标，捕获状态防止它触发相邻控件。源对象将简单地拒绝这样的错误命令。

从捕获中逃逸

拖动操作中最重要也最容易忽视的一点是：取消拖动操作的机制。用户不仅需要终止拖动的方法，也需要在终止拖动操作时，能够确保他成功地做到了这一点。

如果能醒目而毫不含糊地向用户表明拖动动作已经取消，那么无论情况如何，用户都能放心大胆地使用这种取消习惯用法。然而，多数应用程序还没有任何形式的拖动取消方法。这是用户界面中的重大失误，因为任何优秀的界面都将提供一致且可靠的方法来取消用户不慎引发的动作。

---

[1] **译者注** 原文为 windows，此处并非窗口，因为它可能包含连续数据或者控件。

[2] **译者注** 此处区分了拖动和拖放，前者在特定上下文环境中可能意味着拖放，如源对象是离散对象。

公理：提供取消拖动的方法，并向用户明确表示。

至少，可以认为键盘上的 Escape 键是通用的鼠标操作取消机制，无论是单击还是拖动。如果用户在按下鼠标键的同时，按下 Escape 键，系统应该放弃捕获状态，恢复到此次单击动作之前的状态。当用户随后释放鼠标键时，程序必须在它产生任何副作用之前，放弃处理这个作为用户输入的鼠标释放操作。

因为元键是惟一在拖动过程中有特殊意义的键，所以我们可以用元键以外的任何键来取消鼠标单击，而不仅仅是 Escape 键。但有些程序允许箭头键和鼠标联合使用，所以也存在某些特例有待解决（我们在下一章中讨论）。

一种比较少为人知的习惯用法就是合击，即用户同时单击鼠标左右键。Windows 中有很多通过合击取消拖动操作的例子。例如，用户用左键开始拖动，之后却发现自己并不是真的想完成这样一个操作。他可以单击右键，然后安全地释放双键。这个习惯用法对释放的时间和顺序不敏感，对于用鼠标右键开始的拖动也同样适用。

设计技巧：用合击取消拖动。

微软在 DOS 版的 Word 软件中采用合击取消拖动，但不幸的是，在 Windows 中放弃了这种用法。诚然，这种用法适用于专业人士，但为什么在界面上要迎合新手而妨碍到专家呢？至少现行的 Word 版本把 Escape 键当作取消拖动操作的用法。

Windows XP 中的 Windows Explorer（甚至包括桌面）支持 Escape 和合击作为取消拖动的习惯用法，但使用合击时，会产生一个问题。Windows 设计师选择在鼠标释放时启动上下文弹出菜单（单击鼠标右键），而不是在鼠标按下时。这就意味着当你的手指离开鼠标右键时，你将面对一个自己不希望出现的菜单，你不得不主动离开，用另一次单击来关闭菜单。讨厌！让菜单在鼠标按下时弹出是有理由的！

因为当时微软只是尝试性地推出双键鼠标，所以它也只是尝试用合击作为取消操作的习惯用法。但是现在微软似乎理所当然地认为它的所有用户都至少有两个以上鼠标键，并且认为采用合击作为通用的取消习惯用法是合理的（假设确实成功取消了的话）。写信给国会议员吧。

如果你的程序设计合理，能够让用户利用 Escape 键或者合击取消拖动操作，也还存在一个问题，那就是要让用户确信他现在的操作是安全的。因为光标可能已经发生了改变，提示拖动正在进行，或者所拖动对象的轮廓已经在随着光标移动。取消操作使这些视觉暗示消失，但用户仍然想知道他是否已真正安全。用户可能已经按下了 Escape 键，但他仍然会按住鼠标键不放，因为他不能确认释放按键是否完全安全（也许他已使用了合击，正按下鼠标的两个键）。此刻应该通知用户操作已经有效地取消，可以释放鼠标键了。确保用户确定他得到了可靠的信息只会有益无害。

当然，用户将获得一些视觉反馈，无论他拖动的是什么对象，都已经不再处于拖动状态了，因为它会从光标下面消失或者跳回到原来的位置。可以通过提供简洁的动画来强化视觉反馈，比如，在最初的 Macintosh 中打开和关闭窗口时所伴随的那种动画（这是非常精妙的视觉提示，Windows 应该从中吸取经验。）

作为替代方案或追加方案，可以提供一个类似工具提示的简短信息，让它在光标附近显示一两秒钟。也可以提供短小的“促音”来提示拖动操作已经取消，这有助于强化习惯用法。

受范性反应

让我们回到拖动操作本身：一旦拖动开始，用户动作的含义取决于拖动动作的类型。拖动动作又与程序、上下文和源对象有关。

最简单的情形是，对于连续数据，拖动意味着“扩展选择范围”。文本或者其他数据在鼠标按下点到鼠标释放点之间得到连续选择。

当光标在控件内时单击鼠标，控件必须通过视觉效果表明它的状态正要改变。这种表现是很重要的，但常被那些创建控件的设计师所忽略。这是主动视觉暗示形式的一种：**受范性反应**。

按钮需要在视觉上从凸起状态变为凹陷状态；复选框应该突出显示方框，而不是画勾。对任何控件来说，受范性反应是重要的反馈机制，无论是激活动作还是改变状态，它都可以让用户知道如果释放鼠标按键会触发某些动作。受范性反应还是取消机制的一个重要组成部分。当用户单击按钮，按钮凹陷。如果用户在按住鼠标键的同时移动鼠标离开按钮，按钮则会回到它静止时的凸起状态。这时如果用户释放鼠标，将不会激活按钮（与没有受范性反应时一致）。

单击—拖动控件

许多控件，尤其是菜单，需要相对复杂的单击-拖动操作，而不仅仅是单击。这个直接操作对用户要求更高，因为它结合了粗略动作与精细动作来单击、拖动和释放鼠标。尽管菜单使用频率不如工具条控件高，但它仍然是我们经常要用到的，特别是新手用户或者生疏用户。因此，我们发现 GUI 设计的难题之一就是：菜单是初学者主要的操作控件，而从物理操作性方面来看，它又是一个更难操作的控件。

除非提供附加的习惯用法来完成同样的任务，否则没有什么办法可以解决这个问题。如果一种功能可以通过菜单访问，并且用到的次数较多，则需要提供激活此功能的其他习惯用法——一种不需要进行单击-拖动操作的习惯用法。

Windows 有一个很好的特点，是能够用一系列的单击来操作菜单，而无须单击和拖动，Mac OS 的最新版本也采用了这种做法。单击菜单，它就向下展开。当指向你所期望的菜单项时，单击一次就可以选择和关闭菜单。微软进一步扩展了这种理念，你在任何菜单上一单击，程序就转换为一种**菜单模式**。处在菜单模式时，程序中所有的上级菜单以及这些菜单中的所有项都处于激活状态，就像你在进行单击和拖动一样。当你移动鼠标时，无须使用鼠标键，相应的每个菜单就会向下展开。如果不熟悉这种形式，可能会有些惊慌，但最初的惊慌过后，你会感到高兴，因为这种设计会使你的手腕备感舒适。

调色板工具的行为

画图和绘画程序中，在工具栏选定一种工具或形状后，用户通过拖动来进行操作。工具栏工具有它们自己独特的行为，值得我们单独讨论。它们的行为有两种基本类型：模态工具和加载的光标工具。

【模态工具】

使用**模态工具**时，用户从列表或专门的工具条（通常称为工具箱或者调色板）中选择一种工具。现在，程序的显示区完全处于这种工具的模态中：程序只能完成这一工具的工作。通常用光标的改变来表明当前激活的是什么工具。

当用户在画图区域使用工具单击和拖动时，工具就执行工作。例如，如果激活的工具是喷枪（spray），程序就进入了喷枪模态，只能完成喷墨工作。工具可以反复施用，按需喷墨，直到用户选择其他工具。如果用户希望在画图中使用其他工具，比如橡皮擦

（Eraser），他必须回到工具箱，选择橡皮擦工具。然后程序就进入了橡皮擦模态，在选择其他工具之前，程序只能擦除对象。调色板中通常会提供光标选择工具，使用户可以将光标回复到用于进行选择动作的指针（selection-oriented pointer）状态，例如在 Adobe Photoshop 中就是如此。

模态工具既可以是在画图中执行动作的工具——如橡皮擦，也可以是绘制形状的工具——如椭圆。光标可以变成橡皮擦，擦除先前输入的任何东西，也可以变成椭圆工具，绘制多个新的椭圆。鼠标按下的事件就确定了形状的角度或中心（或是它的边界），用户拖动鼠标可以把图形拉伸到期望的大小和外形。鼠标释放事件则确认这次画图完成。

模态工具在“画图”这样的程序中不会令人烦恼，因为其中的工具数量很少。而在更高级的画图程序如 Adobe Photoshop 中，这种模态具有破坏性。因为随着用户能够熟练使用的光标和工具增多，花费在选择和改变选择工具上的时间和动作——附加工作，显著增加。在这类画图程序中，模态工具提供了非常好的习惯用法来引导用户了解该程序的功能集，但对于熟悉复杂程序的熟练用户来说，模态工具并不太有效。幸好，像 Photoshop 这样的程序充分利用了键盘命令来满足超级用户的要求。

管理模态工具应用的难度更多地来自纯粹的工具数量，而不是模态性。更确切地说，当用户工作时，使用工具的数量太多将会影响工作效率。超过五个模态工具就常常难以管理。例如，如果 Adobe Illustrator 中的必要工具从 24 个下降到 8 个，它的用户界面问题就可能不至于让用户觉得痛苦。

为了补偿模态工具数量太多的副作用，像 Adobe Illustrator 这样的产品采用元键来调节不同的模态。Shift 键常用于受约束的拖动，但在 Illustrator 中添加了很多不标准的元键，并且用了不标准的使用方式。例如，在拖动对象的同时按下 Alt 键，拖走的就是对象的拷贝，但 Alt 键也可以将选择工具从顶点选取功能切换为对象选择功能。这两者之间的区别很微妙：如果单击某物，然后按下 Alt 键，你拖走的是拷贝。相反如果你先按下 Alt 键，然后单击某物，你选择的是整个对象，而不是它的单个顶点。但更令人迷惑的是，此时你必须释放 Alt 键才能拖动对象，否则拖走的就是该对象的拷贝。对于“选择一个完整的对象，并将它拖动到新的位置”这样一件简单的事，你必须按住 Alt 键，再指向对象，进行单击，然后在不移动鼠标的前提下按住鼠标键，先释放 Alt 键，将对象拖到目标位置！这些人在怎么想呢？

必须承认，这些可能的组合非常有用，但是它们难学、难记又难用。如果你是一个专业的图形艺术家，每天要用 Illustrator 工作八小时，你可以把这些缺点转变成优势，就像赛车手在跑道上能将难以驾驭的、精细调节过的汽车变成自己的优势一样。但 Illustrator 的临时用户会像普通司机那样，由于工具超出自己的水平而远远落后。

Adobe Illustrator 牢牢植根于 Macintosh 世界。它的错误之一在于窗口界面的设计拒

绝利用双键鼠标，而这正是微软 Windows 免费而廉价的特性。Illustrator 完全没有使用右键。毋庸置疑，公司的一些人认为与 Mac 的互动操作性更重要。例如，Adobe 可以将所有的选择工具设置在左键上，所有的画图工具设置在右键上。用户只需决定使用哪个鼠标键就可在画图和操作之间来回转换。甚至更好的方案是每个按钮都可以使用三个元键：Alt、Ctrl 和 Shift。

【加载的光标工具】

使用**加载光标**工具时，如果用户在调色板中选择一种工具或形状，光标不会（在用户再次切换前）永久地切换到所选工具的光标形状，而是临时加载了——或负载了——所选对象的单个实例。

此时如果用户在画图区域单击一次，在显示屏的鼠标释放点就会创建一个实例。加载光标对于功能性操作来说不太好用（尽管微软将它无所不在地使用在 Format Painter 功能中），但它确实很适合于处理图形对象。例如，在 PowerPoint 中它就得到了广泛应用。用户从图形工具栏中选择一个矩形，光标就变成了负载一个矩形实例的矩形模态工具。

许多普通的画图程序采用了这种光标工具，同时，作为一种图形直接操作习惯用法，加载光标工具在通常不认为是画图的程序中也非常流行。Visual Basic 就是一个合适的例子。当用户在工具栏中单击控件，光标就加载该控件。用户再次单击，就会在窗体中创建一个控件实例。Borland 的 Delphi 也使用了加载光标，但是，如果你按住 Shift 键在调色板中单击控件，得到的将是创建多个控件实例的模态工具。不错！

在许多使用加载光标的程序中，如 PowerPoint，用户不能通过单击放下对象，而是必须拖动矩形框来决定对象的大小。一些程序如 Visual Basic 中这两种方法都可以。单击一次加载光标将以默认的大小创建一个对象实例。新的对象创建后处于选择状态，几个操作点（我们将在下一小节讨论操作点）环绕着它，对象的形状和大小能立即进行精确调整。毫无疑问，既允许单击创建默认尺寸的对象，又允许拖动矩形框定制对象大小的双重模式，是最灵活和最明显的方式，能令多数用户满意。

有时使用加载光标的程序忘记改变光标的外观。例如，Visual Basic 在光标加载时，会将光标变成十字准线，而 Delphi 则根本不做任何改变。如果认为这时光标是一种模态行为——如果单击它会创建某物——在视觉上指示这种状态是重要的。加载的光标也要有合适的取消操作习惯用法。否则，你如何安全地取消光标的加载状态呢？

对象操作

显示屏上的数据对象，特别是画图和建模程序中的图形对象，像控件一样，都能通

过单击和拖动来操作。对象（不是在前面章节中讨论过的图标）依靠单击一拖动完成三项主要操作：调整位置、调整大小和调整形状。

调整位置

调整对象的位置是一种简单的行为，只须单击对象然后将它拖动到新的位置。关于调整位置，最重要的设计问题就是它侵占了其他直接操作习惯用法的使用空间。调整位置这一功能需要使用单击-拖动动作，结果是这个动作不能再用于其他目的。如果对象可以调整位置，就意味着占用了单击-拖动，不能再用它来在对象上执行其他操作，比如在对象自身中滚屏或按下一个按钮。

要解决这种冲突，最普通的方法是将对象的某个专门区域用于调整位置功能。例如，在 Windows 和 Macintosh 中你可以单击和拖动标题栏来调整窗口位置。窗口的其余部分不是调整位置的受范区域，这样，正如你可能希望的，单击-拖动习惯用法就可以用于窗口内的其他功能。窗口可以拖动的惟一暗示就是标题栏的颜色，这种精妙的视觉暗示纯粹是习惯性的：无法用直觉理解。但它非常有效，证明了基于习惯用法的界面设计的有效性。总之，你需要为受范区域提供更清楚的视觉暗示。例如，标题栏可以用亮度方面的轻微改变作为受范暗示，或者可以用光标暗示。这种解决方法的代价是必须有专用的像素分配给标题栏。标题栏具有多重任务的事实可以减轻这种代价，比如，标题栏可以是程序的标识器，活动状态指示器，其他系统标准控件，如最小化、最大化和关闭功能的所在。

要想移动对象，首先必须选择它。这就是选择必须发生在鼠标按下时的原因：用户可以用拖动，而不是先单击后释放，来选择对象，接着单击一拖动来调整位置。这样比简单地单击然后拖动到希望的位置更自然。

移动连续数据又会产生一个问题。例如在 Word 中，微软使用不灵活的单击-等待-单击操作来拖动大块文本。你必须以单击和拖动来选择一个文本段，然后等待一秒钟左右，并再次单击和拖动来移动文本段。这是令人遗憾的，但对于连续选择又没有其他好的替代方法。如果微软愿意去掉扩展选择操作的元键习惯用法，那些元键就能用于选择句子，并能在一次移动中拖动它。但这仍然不能解决选择和移动任意文本段的问题。

调整大小和调整形状

当谈及 Windows 桌面和其他类似的图形用户界面时，调整大小和调整形状之间真的

没有任何功能上的差异。通过单击和拖动专门的控件，用户调节矩形窗口大小的同时，也改变其外观比例。Macintosh 中在每个窗口的右下角有一个特殊的调整大小控件。拖动这个控件允许用户改变矩形的高度和宽度。Windows3.x 避开了这种习惯用法，使用窗口周围的外围边框来调整大小。它在提供充分视觉暗示的同时，也提供光标暗示，这样就很容易发现。Windows 95 加上了类似于 Macintosh 的右下角用于调整大小和调整形状的控件，尽管这样缩窄了它的窗口边框（通过边框调节大小仍然起作用）。现在，Windows 保留了边框和它的光标暗示，但边框的视觉暗示基本上都没有了。对于注意到光标暗示的用户来说，这种方法是两种方案的完美结合。

这些习惯用法对于调整窗口大小来说是合适的，但当调整的对象是绘图或建模程序中的图形元素时，在对象上永久叠加这样的控件就没法接受。用于调节图形对象大小的习惯用法在视觉上必须很醒目，来将它自身与图画的组成部分，尤其是它所控制的对象区分开来，并且必须考虑对象及其周围区域的用户视图，调节大小的指示器也不能混淆调整大小的动作。

一个流行的习惯用法实现了这些目标；它由分别安置在矩形对象的四个角和四条边中心的八个小黑色正方块组成。这些小黑色正方块，如图 24-1 所示，称为调整大小操作点（或简称为操作点）。

操作点对设计师来说是很实惠，因为它们还可以指示选择。这是个很自然的共生关系，因为通常对象只有选择之后才能调整大小。

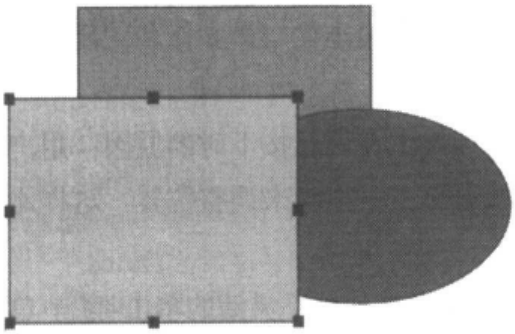


图 24-1　所选对象有八个操作点，每个角上一个，每条边中点一个。操作点在指示选择的同时，也是一个经济的对象大小和形状调整习惯用法。操作点有时通过像素反转来实现，但这样在颜色多的时候可能会淹没在混乱中。

位于每条边中心的操作点只能移动这条边，其他边保持不动。位于角上的操作点可以同时移动它接触的两条边，这种交互在视觉上是非常直观的。

操作点有可能会模糊它们所代表的对象，所以它们不能成为长期停留在屏幕上的控件。这是我们之所以在更高层的可调整大小窗口中见不到它的原因（尽管在 SUN 某些版本的 Open Look GUI 窗口中可以看到）。对于那种情况，外框或角上的调节大小指示器更实用。如果所选对象比显示屏还大，操作点可能看不见。如果它们隐藏在显示屏后面，那么它们不仅不能用于直接操作，而且作为选择指示器也没有用。

必须注意的是，我们在这里讨论的操作点存在一个前提条件，那就是由它调控的对象是矩形，或者容易被矩形包围。当然在 Windows 世界中，矩形对象易于操作，不是矩形的对象也能通过将它包围在矩形边界内操作。如果用户创建一个组织系统图，这也许已经够了，但对于更复杂的对象调整形状又会怎样呢？其实，还有一种功能强大的调整大小操作点变体：顶点操作点。

许多程序在屏幕上画出的对象呈折线形（polyline）。折线是绘图程序的专用术语，专指由一系列顶点定义的多段线条。如果最后的顶点与最初的顶点重合，它就是闭合的，也就形成了一个多边形。这样的对象选中时，程序不是像在矩形图那样设置八个操作点，而是在折线形的每个顶点上设置一个操作点。用户可以独立地拖动任何一个顶点，局部改变对象内部形状，而不会在总体上影响对象。如图 24-2 所示。

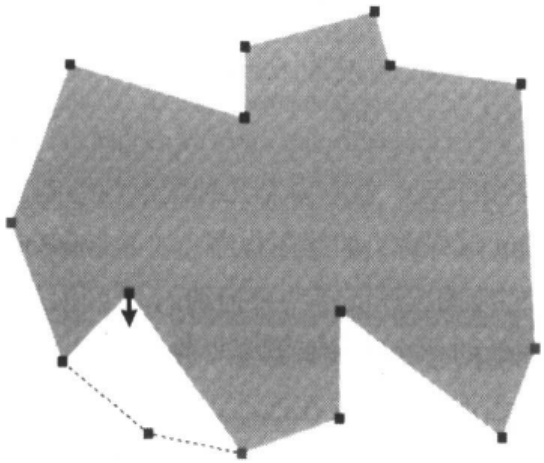


图24-2 图中的黑色小方块为顶点操作点，之所以如此命名，是因为多边形的每个顶点有一个操作点。用户可以单击和拖动任何操作点调整多边形的形状，每次调整一段。这个习惯用法在画图程序中很有用，而且在桌面生产率程序中也有应用。

PowerPoint 中的任意边形也以折线方式绘制。如果单击这种不规则的图形，会出现带有八个标准操作点的矩形外框。如果你随意右击图像，并在背景菜单中选择编辑顶点，这个矩形外框就会消失，由顶点操作点取而代之。这两种习惯用法都很重要，前者用于按比例调节图像，后者用于精细调节图像形状。

调整大小和调整形状的元键变体

在拖动时，元键常用来将拖动约束在正交的方向上。这种类型的拖动称为约束拖动（constrained drag），如图 24-3 所示。

约束拖动的是一种实际路径受到限制的拖动，无论用户的鼠标移向什么方向，对象只能直线地向上、向下、向左、向右或者呈 45° 角拖动。通常使用是的 Shift 元键，但程序之间有不同的惯例。约束拖动在画图程序中非常有用，特别是在绘制整洁的组织结构

图时。最初拖动的几个毫米具有支配意义，它决定了拖动的方向。例如，用户在水平轴上开始拖动，自此以后的拖动就限制在水平轴上。有些程序对约束有不同的解释，它允许用户在拖动期间通过将鼠标拖动超越某个阈值而改变拖动角度。

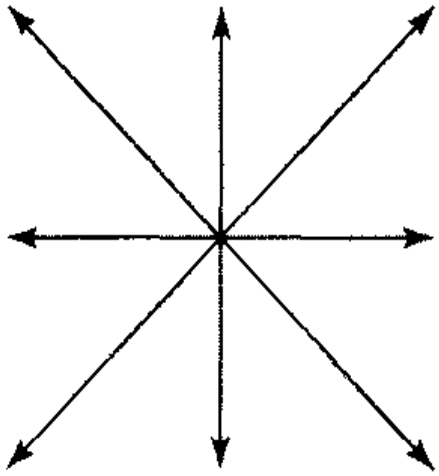


图 24-3 通常通过按下 Shift 键来约束拖动，此时拖动受限，对象只能沿图中所示的四个轴向拖动。程序通过鼠标初始的移动确定移动方向。拖动阈值的实现将在本章后面部分讨论。

Windows 的画图程序 Paint 在移动对象时不约束拖动，但它限制几种图形的绘制，如线段和圆。多数将图形视为对象而不是比特（Paint 这样做）的绘图程序（如 PowerPoint）允许约束拖动。像 Adobe Photoshop 这样更复杂的绘图程序支持约束拖动的习惯用法。

使用元键产生了一个有趣的问题：元键在拖动的什么阶段有意义？换句话说，是必须在拖动开始时——鼠标按键向下时按下元键，还是仅仅需要在拖动过程的某个位置按下元键？是否需要在鼠标释放前一直按下元键？最好的回答是：用户在他开始拖动后的任何时间内都可以通过按下元键，转换到约束拖动并且得到视觉反馈。如果拖动过程中取消了元键，就能返回非约束拖动状态。最后，如果计算机发现在鼠标键释放瞬间元键仍然维持按下状态，约束拖动就会得到确认。PowerPoint 和 Paint 就是这样做的。

在有趣的、基于比特画图的程序 Paint 中，它在它的铅笔工具中也允许约束拖动：在拖动过程中，任何时候按下元键，就会限制所画的对象，鼠标释放时，则会停止流出数字墨水。

三维对象的操作

对于只装备了二维输入设备和显示器的用户来说，精确地处理三维对象无疑对交互提出了重大挑战。用户界面设计领域一些最有趣的研究与此相关，它们试图开发出更好的适合三维输入和控制的范例，但迄今为止仍然没有真正的突破，只是将二维习惯用法演化扩展到了三维世界。

多数三维应用程序与精确绘图（如建筑CAD）或三维动画相关。创建模型的时候，动画出现的问题与绘图时出现的问题相似。使模型随时间发生移动和改变增加了新的复杂性。动画家常在专门的应用程序中创建模型，再将这些模型加载到不同的动画工具中。

对三维操作习惯用法更深入的讨论可能需要整章甚至整卷书的篇幅来描述。因此我们这里只简单地介绍三维操作的一般性问题。

【显示问题和习惯用法】

也许在二维屏幕上实现三维交互最重要的问题就是这种环境缺乏视差——双眼能察觉事物深度的一种能力。设计师们没有求助于昂贵而神秘的眼镜外围设备，而是用一些技巧来解决这些问题。另一个重要的问题是：近距离的对象挡住了远距离的对象。这些导航问题，以及我们将在下节中讨论的输入问题，可能就是虚拟现实（virtual reality）没有成为图形用户界面未来趋势的主要原因。

多重视点（multiple viewpoint） 使用**多重视点**可能是解决以上两个问题最古老的方法。但从交互的观点来说，多重视点在很多方面是效果最差的。虽然如此，多数三维模型应用程序在屏幕上设置了多重视点，每个视点从不同的角度显示同一对象或情景。通常，包括了视图、正视图和侧视图，每个视图与绝对轴对齐，每个轴都可以向内或向外缩放（zoom）。大多数情况下，还会有第四个视图，情景的正交投影或透视投影，用户可以调节每个视图的确切参数。可是，当每个视图都有单独的窗口，并且每个窗口都有自己的外边框和控件时，这个习惯用法就变得非常麻烦：窗口不可避免的相互重叠，彼此影响，有限的屏幕被重复的控件和窗口边框浪费。一个较好的解决方法是启用多窗格窗口，每个窗口中允许配置一个，二个，三个，或者四个窗格（在三个窗格的配置中有一个大窗格和两个较小的窗格）。使用工具条或者键盘快捷键，能尽可能确保一次单击就能配置这些视图。

多重视点显示的缺点在于，需要用户同时看多个地方才能知道对象的位置。迫使用户在复杂的情景中从顶上、侧面和前面各个角度定位对象，并期望他在大脑中实时进行三角测绘实在不容易，即使对于建模专家也是勉为其难的。不过，多重视点对沿特定轴线精确对齐对象还是有帮助的。

基线网格（baseline grid）、**景深效果**（depth cueing）、**阴影**（shadow）和**标杆**（pole）基线网格、景深效果、阴影和标杆是用于解决多重视点所致问题的习惯用法，它们背后的理念是允许用户在正交或透视投影视图中了解三维情景中对象的位置和运动。

基线网格提供了情景的虚拟地板和墙壁，每个轴一个，有助于用户定位方向。当摄

像机视点能自由旋转时（通常情况也是如此），基线网格特别有用。

借助**景深效果**，视野中较深的对象显得暗淡。这种影响通常是连续的，所以给出了对象大小、形状和范围的有用线索，即使在单个对象表面上也能显示景深效果。当在格线上使用景深效果时，能够帮助消除视图中格线方向的歧义。

在某些三维应用程序中定位的方法是**阴影**，所选对象的轮廓投影到格线上，就好像光线垂直照射每个格线。当用户在三维空间移动对象时，可以借助这些阴影或轮廓跟踪她在每个维度上是怎样移动对象（或改变对象大小）的。

阴影能起到很好的作用，但那些阴影和网格在视觉上会互相影响。一个可行的方法就是使用单层**底板网格（floor grid）**和**标杆**。标杆常与水平方向的网格一起发挥作用。当用户选择了一个对象，从对象的中心有一条垂直线扩展到网格。当她移动对象时，标杆跟随移动，但标杆始终保持与水平方向的网格垂直。用户可以通过观察标杆在格线（*x* 和 *y* 轴）表面移动的出发点，以及标杆相对于网格（*Z* 轴）的距离和方位，来了解她在三维空间中移动对象的位置。

准线（guide line）和其他丰富的视觉暗示 前面一节描述的习惯用法全部是丰富的非模态视觉反馈的例子（至于非模态视觉反馈，我们将在 34 章进一步讨论）。但是，对于某些应用程序，过多的格线和标杆则具有破坏性。例如，@ Last Software’s Sketch Up 是一个建筑绘图程序，当用户使用这一程序设计草图时，可以使用卷尺和量角器工具绘制自己的草图线条，也可以用色彩编码提示在恰当的轴上正确定向，还可以启用蓝色梯度的天空和地面颜色来帮助定向。因为应用程序关注的是建筑设计，不是通常意义的三维建模或动画，所以设计师努力实现一个既简洁、功能强大又易学易用的简单界面（如图 24-4）。

线框（wire frame）和**边框**（bounding box） 线框和边框解决了对象可视化问题。在处理器速度非常缓慢的日子里，因为计算机速度不够快，不能实时绘制实心的外观。所有对象都需要用边框代表，那时建模应用程序通常只绘制所选对象的粗糙表面，而用线框代表未选对象。透明度也可以起到一定的作用，但它仍然有很高的计算强度。在高度复杂的情景下，有时只绘制非选择对象的边框是必要的，或者可以满足要求了，虽然不是最完美的。

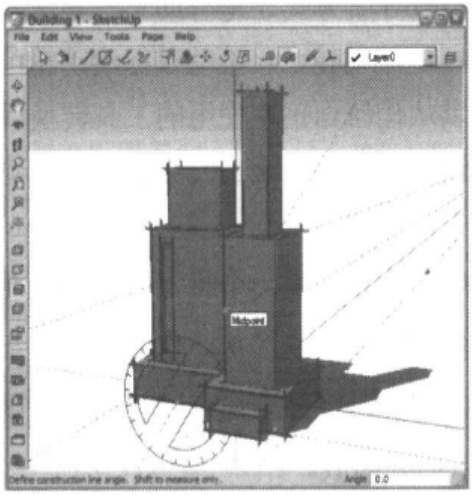


图24-4 @ Last Software's SketchUp是应用程序中的精品。它集强大的三维建筑设计功能、流畅的交互、丰富的反馈和可管理的设计工具集于一身。用户可以根据位置、方向、季节以及一天中的时间设置天空的颜色和现实世界里的阴影。这不仅有助于表达，而且可以帮助用户在创建时更好地定位。用户也可以布置三维格线和度量指南，而就像是在使用二维绘图应用程序一样，图中可看到量角器。摄相机旋转和缩放功能巧妙地映射到鼠标的滚动轮上，这样在使用其他工具时可以方便地使用这些功能。工具提示提供了协助画线和对象对齐的文本提示。

【输入问题和习惯用法】

三维应用程序采用的许多习惯用法，例如，拖动操作点和顶点操作点，是从二维应用中转化而来的。但是，在三维输入中还存在一些特殊的问题。

拖动域 三维场面的二维投影中，使用直接操作的基本问题之一就是如何将屏幕上光标的二维移动，转换为虚拟三维空间内更有意义的移动。

在三维投影中，不同种类的**拖动域**需要区分三个轴，而不是两个轴上的移动。比方说，鼠标上下移动变换成沿一个轴向上的移动，而45度角的拖动常用于表示另一个轴上的运动。当用户沿特定轴平行拖动时，Sketch Up用虚线的形式提供色彩编码提示，同时也采用工具提示来进行暗示。在三维环境中，以光标和其他类型的暗示来显示丰富的视觉反馈是非常必要的。

选取 三维操作的另一个重要问题是**选取**。因为在组合情景时，对象要么在线框里，要么透明，这样当用户的鼠标经过许多重叠的项目时，程序很难知道选择的究竟是哪一个项。突出显示位置有一定的帮助，但还不够，因为对象有可能被其他对象完全遮盖。成组选择就更难处理了。

许多三维应用程序求助于不那么直接的技术，如对象列表或对象层次关系，用户可以在三维视图外从这些列表或关系中选择对象。尽管这种交互有用，但还有更多直接的方法可以选择。

例如，当掠过画面的某一部分时，可以打开一个类似菜单的工具提示，让用户选择一个或多个重叠的对象（这种菜单对于只有单个清晰对象这种简单情况来说是不需要的）。所有可以单独选择的平面、顶点或边，在鼠标经过时都应该有受范性提示。

虽然不能直接解决问题，但是，一种平稳简单的情景导航方式还是有助于改善选取的。Sketch Up 将缩放和**环绕**（orbit）功能都映射到鼠标的滚动轮上。滚动鼠标滚动轮可以围绕三维空间的中心零点进行缩放；按住滚动轮可以从任何你正在使用的工具转换成旋转模态。这种模态下允许图像围绕任何方向的中心轴旋转。这种流畅的导航使操作建筑模型如同在手掌中旋转它一样轻而易举。

对象旋转、摄像机移动、旋转和缩放 对于三维应用程序来说一个特定的问题是能实现多少空间操作功能。可以在三条轴上调整对象的位置、大小和形状。也可以在这三条轴上旋转。此外，摄像机视点可以在适当位置旋转，或者围绕某个焦点旋转，也可以在三条轴上旋转。最后，摄像机的视野可以分别向内或向外缩放。

这不仅意味着在三维应用程序中元键和快捷键的分配很关键（很明显，这些控件的一部分可以放到工具条上，但专业用户几乎全部用键盘来控制这些模式），这里还存在另一个问题：即使摄像机变换和对象变换之间的实际差别很大，但通过观察摄像机视点也很难分辨两者的差别。解决这个问题的方法是在显示屏的一角加入情景绝对视图的缩微图，这个视图在需要时可以放大或缩小，并且，为避免用户迷失在空间中，它提供了一个现实检查和整体导航的方法（注意，这种缩微图在导航大的二维图像时也很有用）。

对象连接

在某些应用程序中，一种功能强大的直接操作习惯用法是连接，用户从一个对象单击拖动到另一个对象，不是将第一个对象拖到第二个对象之上，而是从第一个对象画一条连线或者箭头到第二个对象。

如果用过项目管理或组织图表程序，你肯定熟悉这个习惯用法。例如，在项目经理的网络图（通常称为 PERT 图）中，将一个任务框与另一个任务框连接起来。在这种情况下，连接的方向很重要：鼠标键按下点处的任务是源任务，鼠标键释放点处的任务是目标任务。

当拖动两个对象之间的连接时，它以**橡皮筋**（**rubber-banding**）的形式提供视觉反馈：箭头成为一条从鼠标按下点延伸到光标所在位置的线。线条是动画的，一端随光标移动，另一端停留在鼠标按下点。

在用户释放鼠标键后，就确定了鼠标释放点，程序也就可以决定它是否位于有效的目标位置。如果是这样，程序会在两个对象之间绘制一条永久的线或者箭头。在某些应用程序中，它也在逻辑上把两个对象连接在一起。

当用户在屏幕上拖动箭头的一端，应用就会捕获到这个输入，在这里适用拖动离散

数据的规则。

通常鼠标左键不能触发连接功能，因为那样会与选择和调整位置冲突。在一些程序中，鼠标右键触发连接功能，但 Windows 中鼠标右击启动的是上下文菜单，这样就有问题。更好的解决方法可能是设置一个专用连接区域或操作点，作为拖动连接的始发区域，或者从工具栏中使用模态工具/加载光标工具，或者采用元键映射（比如，可以考虑 Alt 键是不是还没有占用）。

连接本身也完全可以成为对象，具有调整大小的操作点和可以编辑的属性。这种实现方式意味着连接可以进行独立的选择、移动和删除。在某些程序（如项目规划应用程序）中，对象之间的连接需要包含一定的信息，因此在这种情况下，让连接成为“一等公民”很有意义。

连接不像其他习惯用法那样需要光标暗示，因为橡皮筋线的作用清晰可见。但是在对象存在逻辑联系的程序中，显示出箭头当前指向的哪一个对象是有效的目标将很有帮助。换句话说，如果用户拖动一个箭头，在它还没有指向屏幕上某个图标或小部件之前，他如何才能辨别这个图标或小部件是否能合法地连接？答案就是采用某些主动视觉暗示来显示潜在的目标对象。这种潜在目标对象的暗示可以非常微妙，甚至当程序中所有对象对于任意连接都是有效目标时，这种暗示可以完全没有。不过，当连接拖动经过目标对象时，它们都应该突出显示，表示愿意接受。

无可争议，提供便利的连接取消手段极其重要，合击和 Escape 在这个习惯用法中仍起作用。

第六部分

控件及其行为

25 窗口行为

26 使用控件

27 菜单：教学向量

28 使用菜单

29 使用工具条和工具提示

30 使用对话框

31 对话框礼节

32 创建更好的控件

25

窗口行为

任何有关用户界面设计的书都会讨论 windows（是小写的 windows，窗口）。在本章中，我们将首先讨论窗口，这个无处不在的矩形，发展的历史过程，来帮助读者更多地了解它，而不是仅仅沉浸在对它的直观感觉中。随后，在本章的余下小节中，我们将讨论窗口和多窗口应用的行为属性，以及何时使用窗口比较恰当，何时又不恰当。

PARC 和 Alto

当前图形用户界面的外观都源自 Xerox Alto，20 世纪 70 年代中期在 Xerox（施乐）公司的帕洛阿尔托研究中心（Palo Alto Research Center，PARC）开发的一个实验性计算机桌面系统。 PARC 的 Alto 是第一台具有图形界面的计算机，设计它是用来发掘计算机作为桌面商业系统的潜力。Alto 设计成一个网络办公系统，系统中的文档能进行组合、编辑和以 WYSIWYG（What You See Is What You Get，所见即所得）的形式浏览、存储、检索、在工作站之间进行电子转移和打印。Alto 系统为桌面计算术语的改革做出了巨大贡献，如鼠标、矩形窗口、滚动条、按钮（push-button）、“桌面隐喻”（desktop-metaphor）、面向对象编程、下拉菜单（drop-down menu）、以太网和激光打印。

PARC 对工业和当时的计算处理意义深远。苹果和微软的总裁，史蒂夫•乔布斯和比尔•盖茨，两个人分别在 PARC 看见过 Alto，Alto 给他们留下了不可磨灭的印象。

Xerox 试图将 Alto 和后来在其基础上衍生的计算机系统 Star 商业化，但它们都价格昂贵、结构复杂、速度缓慢，因而在商业上没有获得成功。人们普遍认为 Xerox，这家当时的主流复印机公司，它的管理层缺乏远见和魄力，在推销“无纸化办公”的背后，没有付出应尽的努力。PARC 的智囊团意识到 Xerox 先前在鼓吹不实际的市场份额神话，于是纷纷退出，这极大地造福了其他的软件公司，尤其是苹果和微软。

史蒂夫•乔布斯和他的 PARC 逃亡者立即着手用 Lisa 复制 Alto/Star。他们在很多方面非常成功，但在处理现实问题上却也复制了 Star 的失败。Lisa 是引人注目的、容易接近的、令人兴奋的，同时也过于昂贵（在 1983 年销售价为 9995 美元），速度慢得令人感到沮丧。尽管在商业上注定是一个失败，但它点燃了许多人对于微型计算机工业的憧憬和想像，当时这一工业规模很小，但它的前景一片光明。

与此同时，面向对象的表达和通信模型比 Alto/Star 迷人的图形化外表，给比尔•盖茨留下了更深的印象。微软在 20 世纪 80 年代初期的产品，尤其是电子表单软件 Multiplan（Excel 的前身）就反映了这种思想。

Lisa 的失败并没有打倒史蒂夫•乔布斯。他相信 PARC 关于真正的图形个人计算机设想从理念变为现实的时机已经成熟。他综合苹果公司各部门精英，加上 PARC 流亡骨干，试图开发商业上可行的 Alto 的替代品。结果就是富有传奇色彩的，对我们的技术、设计和文化都有巨大影响的 Macintosh 诞生了。Mac 独力唤醒了工业界对设计和审美的认识，它不仅提出了用户友好的标准，而且还将那些因为工业界对技术细节的热衷而排除在计算领域之外的专业人员从各个不同的领域中释放出来。

对 Macintosh 近乎宗教般的氛围也与 Mac 用户界面的许多方面密切相关。下拉菜单、隐喻、对话框、矩形重叠窗口和其他元素都构成了其神秘性的一部分。不幸的是，因为 Mac 的设计获得巨大成功，它的失败之处常常就被忽视了。

PARC 原则

PARC 的研究者，除了开发了一系列革命性的硬件和软件，创造了 Alto 之外，还倡导了许多至今仍被奉为图形用户界面设计和开发领域真理的创新理念。

视觉隐喻

来自 PARC 的想法之一是视觉隐喻。在 PARC，大家认为整体视觉隐喻是用户能否理解系统的关键，因而也是产品及其理念能否成功的关键。本书的第四部分中，我们讨论了完全依赖这种隐喻设计的某些弊端。

避免模态

与现代图形用户界面相关的另一个原则是“应该避免模态”的观念。模态是一种程序能进入的状态，在这种状态下用户动作的效果与正常状态有所差异，它实质上是一种行为转向。

例如，老的程序要求你必须进入一种特殊状态才能输入记录，然后再进入另一种状态才能打印，这些行为状态就是模态，它们非常令人迷惑和沮丧。曾是 PARC 研究员和苹果公司首席科学家的 Larry Falser 最早提倡在软件中废除模态。一本颇有影响的杂志刊登了他的一张照片，在这张照片中，他身穿的 T 恤上印有“不要让我陷入模态（"Don't mode me in."）”这几个醒目的大字。他的牌照上也写着“不要模态（NOMODES）”。在命令行环境下，模态确实是有害的，但在图形用户界面的对象—动词世界里，尽管设计不当的模态可能极端令人沮丧，但模态并不总是有害的。不幸的是，“不要让我陷入模态”的原则已成为我们设计术语中无可争议的一部分。

可以证明，Macintosh 中影响最大的程序是 MacPaint，一个界面完全模态化的程序。这个程序能让用户在计算机屏幕上逐个像素地绘画。用户从有十几个工具的调色板中选择工具，然后在屏幕上画画。选择一种工具就是进入一种模态，并且在选择某个工具时，程序的行为就遵从这个工具的特性，程序同时只能有一种行为方式。

PARC 的研究员没有错，只是有一些误解。和同时代的程序相比，MacPaint 用户界面的好处非常大，但这种好处并不是来自想像中的非模态特性。相反，是因为用户容易知道程序处于哪个模态，并且模态改变起来也毫不费力。

基于实现模型的模态令人困惑。编辑模态（edit mode）与预览模态（preview mode）只是对程序很方便，对用户则不然。基于用户心智模型的模态则通常是无害的，例如，MacPaint 中的喷雾器模态和画笔模态。

层叠窗口

Mac 中另一个源自 PARC 的基本要素是层叠的矩形窗口（它也转移到了 Microsoft Windows 中）。在现代图形用户界面中，矩形占据着统治地位，而且无处不在，因此它常被看作视觉交互成功的关键。

用矩形来显示数据有一些很好的理由。其中，最不重要的一点可能是因为它和我们的显示技术匹配：相对于其他图形，阴极射线管和液晶显示屏处理矩形更快。而“人们常用的数据输出格式绝大多数是矩形的”，这是比较重要的原因：自 Gutenberg（德国活版印刷发明人）开始，我们就是在矩形纸张上阅读文本，大多数其他输出形式，例如照片、电影和视频也符合矩形的形状，矩形图形和图表也是我们最容易理解的，矩形似乎与人类的认知过程有某种关系。矩形在空间利用方面效率也很高。

层叠窗口表明，和输入模糊命令相比，还有更好的方法来实现在当前多个并发程度中切换。

重叠矩形窗口的设计初衷是想用来代表用户桌面上重叠的纸张。没问题，可是为什么呢？答案又要追溯到整体桌面隐喻。如果你的书桌与作者的一样，那么它一定堆满了纸张；当你想阅读或编辑其中之一时，必须从一大堆文稿中找到它，把它放到最上面，然后才能开始。问题是，这种方式的效果最多也就和真正的桌面上一样，它并不是很好，尤其是当你的桌面很小，对角线仅有 17 英寸，并且上面堆满了“纸”时。

重叠窗口的概念是好的，但在真实世界中却不太实际。当你在屏幕上运行三个或更多的应用程序或文档时，这种重叠纸张的隐喻就要经受考验——它不能按比例增长。此外，这个习惯用法还有其他问题。用户在点击鼠标时，如果不小心偏离了期望中的位置，即使只是一个像素，也会发现他的目标程序不见了，由另一个代替了。微软的用户测试显示，一个典型的用户可能会多次运行同一个文字处理器，因为他误认为程序已经“丢失”，必须重新开始。类似这样的原因促使微软引进了任务栏，而苹果公司仍未完全解决这个问题。

关于重叠窗口的另一些困惑来自其他几种也用重叠窗口实现的习惯用法。我们熟悉的对话框是其中一个，还有菜单和浮动调色板（floating tool palette）。这样的重叠在单个应用程序里是自然的，也是一种形式很好的习惯用法。它甚至还有些隐喻的痕迹：就像某人交给你一个重要的备忘录。

微软与平铺窗口

在关注新的 PARC 图形用户界面显著特征的伟大传统中，比尔•盖茨将他东拼西凑的产品命名为视窗（Windows），以回应 Macintosh 的成功。

第一版的 Microsoft Windows 稍微偏离了 Xerox 和 Apple 创立的模式。Windows 1.0 不用层叠矩形窗口代表桌面上相互重叠的纸张，而是依靠所谓的**平铺**技术允许用户在屏幕上同时显示多个应用程序。平铺意味着多个应用程序可以以统一的小方格方式平分可用的像素，平均分割可用的屏幕空间来运行程序。当时认为平铺的发明是解决层叠窗口带来的定向和导航问题的理想方法。平铺窗口之间的导航比层叠窗口之间的导航要容易得多，但像素的浪费非常惊人。另外，只要用户移动整洁的平铺窗口，他必定要重新面对层叠窗口的附加工作（见第 10 章）。平铺终究没有成为主流的习惯用法，尽管在许多有趣的地方仍然能发现它：试一试右击当前 Windows 的任务栏。

全屏应用程序

重叠窗口不能解决在多个运行程序之间的导航问题，于是许多开发商继续致力于寻找新的解决方法。在某些基于 UNIX 的平台上，会话管理器（session manager）的**虚拟桌面**可以将桌面扩展成可视窗口的六倍大。屏幕的一角有这六个桌面叠加的、小的缩略图，六个桌面可以同时运行不同的程序，而且每个桌面可以打开多个窗口。只需点击你所希望激活的桌面就可以在这些虚拟桌面之间进行切换。在一些版本中，你甚至可以将这些微型窗口从一个桌面拖到另一个桌面。

为了在 Windows 2.0 中加入重叠窗口，微软宁愿面对苹果公司提起的违反合同和侵犯专利双重诉讼。在所有这些论战中，最基本的问题好像被人遗忘了：怎样才能让用户在程序之间容易导航？多个窗口共享一个小小的屏幕，不管是重叠还是平铺（尽管它们确实偶尔有一些用处），都不是好的解决方法。我们正在迅速进入全屏程序时代。单个应用程序在占主导地位时，都会占据整个屏幕。任务栏这样的工具，在提供可视化导航方法的同时，只向当前应用程序借用了最少量的像素（有趣的是，这种理念与 Mac 早期的 Switcher 相似，Switcher 能在全屏显示的应用程序之间来回切换）。任务栏是解决导航的一个好方法，相对于其他解决方案，它在像素使用方面更友好，减轻了用户的困惑，尤其是在当前的应用程序需要长时间使用时，它非常适用。在 Windows 中，用户可以根据自己的意愿选择全屏还是层叠来显示他们的应用程序。Apple 则冒着风险坚持只用层叠窗口。

当代的很多软件在开始设计时都有这样的设想：用户界面应该由一系列层叠的窗口组成，没有模态，由整体隐喻提供信息。PARC 的传统就是一个明显的例子。不管这些设想是对还是错，已知的大多数现代图形用户界面设计都源自它们。但是，好的设计师会抛开传统，用崭新的观点来看待软件设计，他们以历史为鉴，而不是以历史为教条。

多窗格应用程序

有一个习惯用法，吸收了平铺窗口的精华，将这些平铺窗口安排在一个独立的全屏应用程序中，那就是**多窗格窗口**（multipaned windows）。多窗格窗口由共享一个窗口的几个独立视图或**窗格**组成，**相邻窗格**由固定的或可移动的间隔条或分隔器隔开（我们将在第 26 章进一步讨论分隔器）。Microsoft Outlook 是多窗格应用程序的一个典型例子。在同一屏幕上，几个独立的窗格分别显示着邮箱列表，选中的邮箱内容和单个选中的消息。在日历视图中，月视图，周视图，任务清单也以同样的方式显示。

多窗格窗口的好处在于，独立而相关的信息可以轻松地在单个独占屏幕上显示，并能将导航和窗口管理的附加工作几乎减少到零。对于任何一个复杂的独占应用程序，相邻窗格的设计尤为必要。特别是在一个窗格中提供导航和/或构造块，以及在相邻窗格中支持数据查看或数据构造的设计，似乎代表一种可重复的有效模式。

相邻窗格的理念也可以在 Web 上以**帧**的方式采用，但由于它的实现非常糟糕，以及 Netscape 和微软之间的标准之争，使得帧既笨拙又复杂。希望随着 Web 技术的进步和高度交互的 Web 应用程序流行，帧代表的理念会在浏览器中再度出现（浏览器本身已经使用了多窗格）。

多窗格的另一种形式是**堆叠窗格**（stacked pane）或**标签**（tab）。尽管这些形式似乎在对话框中用得最多（见第 31 章），它们在独占窗口中有时也非常有用。一个典型例子是 Microsoft Excel，可以通过屏幕底部的反转标签访问相关的电子表格。Excel 同时使用了相邻窗格和堆叠窗格（如图 25-1 所示）。

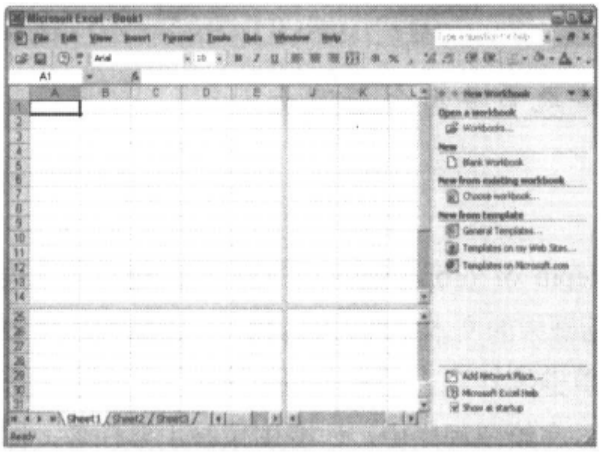


图 25-1 微软的 Excel 是一个既使用了标签窗格又使用了相邻窗格的优秀例子。在标签习惯用法中，反向标签用得很少，也会显得别扭 (见第 31 章)，但在这种背景下工作得很好。Excel 允许你将对单个电子表格的视图分割到多个相邻窗格中。在其他程序，如 Outlook 中，相邻窗格中的信息都是不同的（但通常相关）。该图中右侧窗格就是一个这种类型相邻窗格的例子，它主要用于导航和支持构造。

选择你的窗口

我们的程序由两种窗口组成：主窗口和辅窗口（如文档和对话框）。决定程序的外观很重要的一步就是决定在程序中使用哪种窗口。如果希望创建一个有效的用户界面，我们不能简单地猜测应该使用哪种窗口，而是要仔细地作出选择，并且理解为什么我们必须做出这种选择。

不必要的空间

如果我们将程序想像成一所房子，每个窗口则是一个单独的房间。房子本身用主窗口来表达，每个房间都是一个文档窗口或对话框。在现实生活中，我们不会为房子添加房间，除非它有一个其他房间无法满足的意图。同样，我们也不应该随意为程序添加窗口，除非它有现有窗口不能或者不应该满足的特殊意图。

意图是一个目标导向的术语。它意味着使用房屋与目标相关，但不一定与具体的任务或功能相关。例如，你可能在你家前门与某人握手，这与在厨房或卧室与人握手有着全然不同的含义。

如果刚见面的某人向你伸出手，突然又猛抽回，对你说：“等一下，让我们去厨房握手”，你肯定会觉得非常奇怪。只是为了握手而去另一个房间是不可思议的。这个任务在当前的位置就已经可以很好地完成了！如果在厨房握了手，你们俩又跑回大厅，去继续之前做的事，那就更加荒谬了。

公理：对话框是另一个房间，去之前要有个好理由。

例如，在多数绘图程序中，阴影深度（depth of drop-shadow）通常通过选择某个菜单项，打开一个对话框来设置。然后，在对话框中的文本字段（text field）或类似控件中设置阴影深度。设置完毕后，程序返回主菜单继续绘图。这种顺序如此常见，几乎不会引起注意，但不可否认这是一个拙劣的设计。在绘图程序中，改变图像是主要任务，图像在主窗口中，所以改变它的工具也应该在主窗口中。设定阴影深度不是画图程序的附带任务，而是它不可或缺的重要组成部分。如果用铅笔在纸上画图，画家会带个新工具——橡皮擦，而不会只是为了改变阴影深度换到另一张书桌或纸上。比如，阴影深度可以由工具栏上的控件设定，或者，更好的办法是，让用户通过鼠标点击阴影，然后把阴影拖动到新的位置。

将功能放在对话框中的重点是把它们和主任务分开来。在对话框中调节阴影是可以的，但就交互而言却是蹩脚的。去隔壁的房间握个手同样没有什么问题，但在更广泛的交互背景中则是巨大的浪费。

从程序员的观点看，改变阴影是一项独立的功能，所以他很自然地将它当作一个窗口。但从用户的角度看，它是构成整体所必须的功能，应该囊括在主窗口中。

设计技巧：将功能创建在需要使用它们的窗口中。

在用户界面设计中，这是人们违反得最多的技巧之一。因为程序结构是以功能为中心的，所以用户界面通常也是如此。考虑到创建对话框非常容易，结果就是每个功能一个对话框。现代图形用户界面开发工具倾向于为创建对话框提供便捷的方式，但是，对在文档窗口中添加控件或创建直接操作习惯用法的支持通常没那么好。想做出更好用户界面的开发人员常常必须创建自己的工具，工具开发商给他们提供的帮助很有限。

必要的房间

如果你打算去游泳，却得到一间宾客满座的起居室作为更衣场所，这会是件非常令

人意外的事。礼貌和谦逊是你希望有一间单独更衣室的正当理由，为需要的人提供一个单独的房间是完全恰当的。

当用户执行正常事件序列以外的功能时，程序必须提供特殊场所。例如，清除数据库就不是一种正常的活动。它必须设置和使用不属于数据库程序正常操作的特征和工具。程序中的一般代码支持如输入和测试记录等日常工作，但清除整体记录不是日常事件。单个对话框正适合用于清除操作。对于程序来说，引导用户去一个单独的房间——窗口或对话框——处理清除功能是完全恰当的。

使用目标导向的思维方式，我们可以研究每个功能使之达到最好的效果。如果某人用图形程序画图，他的目标是创建吸引人而有效的图像。所有的绘画工具都与这个目标直接相关。铅笔、画笔和橡皮擦是与目标联系最紧密的功能，这些工具应该紧密地整合为一个工作间，就像传统的画家在制图板上顺手的地方安排好铅笔、钢笔、小刀、镊子、橡皮擦和其他画图工具那样。工具必须在绘画者能顺手拿到的地方，远一点都不行，如果说要画家站起来到另一个房间去就更不用说了。在程序中，画图工具应该排列在画图空间的边沿，单击鼠标就可以使用，用户不必调用菜单或对话框来完成任务。Procreate 的 Painter 7 软件将绘画者的工具放在托盘里，你可以将常用的工具移到托盘的前面。尽管只要你愿意，就可以把不同的托盘或工具栏藏起来，但它们在默认状态下作为画图主窗口的一部分出现。它们也能放在窗口的任何位置。假如你创建了一把画笔，例如特殊红色阴影的瘦炭笔，它是你需要再次使用的，你可以简单地“撕开”工具栏，将它放到工作室里任何你愿意的地方——就像绘画者把炭笔放在托盘里一样。这种工具选择的设计模拟了画图时操作工具的使用方式。

另一方面，如果用户决定导入一幅剪贴画，尽管这项功能与创建一幅好画的目标有关，但它所用的工具与画图无关。剪贴画目录显然与用户画图的目标并不切合——它只是达到最终目的的一种手段。传统的画家也许不会在他的制图板上放一本剪贴画的书，但他可能希望附近有一本，也许就放在靠近制图板的书架上，甚至不必起身就能拿到。在程序中，剪贴画工具应该易于获取，但因为它不属于通常需要的工具，所以它应该放在单独的设施如对话框中实现。

当用户完成了艺术品创作，他就达到了创造一幅有效图像的初始目标。这时，他的目标发生了变化。他新的目标是保存图片，保护它，并通过它交流。钢笔和铅笔的使命已经完成，也不再需要剪贴画了，现在隐藏这些工具无妨。传统画家这时要将图画从画板上取下，拿进大厅，喷定色剂，然后将它卷起来，放进邮寄用的纸筒。他收起画图工具的目的在于——他不希望用定色剂过度喷洒作品，也不希望已经完成的作品被涂料和木炭意外地污损。邮寄用的纸筒不常使用，而且与画图过程无关，所以画家将它们储存在壁橱里。软件中对应的过程是，用户结束画图程序，将画图工具放到一边，在硬盘中

找一个合适的位置存储图像，通过电子邮件传给他人。从用户的目标看，这些功能明显地独立于画图程序。

通过研究用户目标，我们自然能找到程序的恰当形式。不是简单地将每种功能都设置成对话框，相反，我们看到一些功能完全不应该放在对话框中，一些功能应该放到从属于界面主体的对话框中，一些功能则应该从程序中完全分离出来。

窗口污染

一些设计师采取每个对话框实现一个功能的方法。这最终导致了**窗口污染**。

实现多个用户目标需要执行一系列的功能。如果每个功能有一个单独的对话框，那么视觉上很快就会变得拥挤，导航也会变得令人困惑。如图 25-2 所示，CompuServe Navigator（1.0.1 版）就是这样的例子。

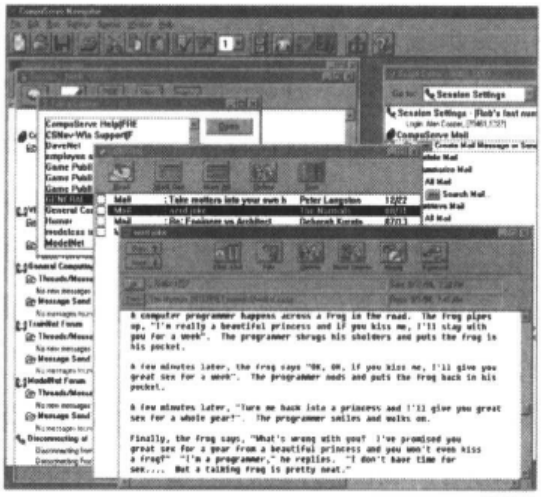


图 25-2 CompuServe's Navigator1.0 版软件的窗口污染问题。正常的电子邮件下载需要打开三个窗口。为了看一个归档的消息需要打开三个以上的窗口。看邮件是一个整体行为，应该占据单一、整体的窗口。这还不是最糟糕的，最糟糕的是用户必须以打开窗口的相反顺序，逐个关闭每个窗口。

给自行车上点润滑油能使踩踏板更省力，并不意味着倒上一加仑油能使踏板自己动起来。Navigator 的设计师之所以不断地增加窗口，也许源于他们对窗口的错误认识：窗口总是好东西。

公理：任何交互习惯用法的效果都和上下文相关。

另一种可能是，Navigator 是在没有整体设计框架的情况下，由很多程序员完成的。每个功能模块都以一个窗口或对话框来表达，这样做仅仅是为了在后期更容易清晰地组

合：经典的实现模型。这个例子我们在 1995 年“About Face”的第一版中已经提到过，希望当前的状况已经有所好转。但只需要看看今天美国在线（AOL）的界面就会知道改变是微乎其微的。AOL，除了因附加工作和困惑给庞大的用户群造成了巨大的损害，它一直是这个星球上最糟糕的窗口污染者。

让我们来看看 Navigator 的一个例子：电子邮件。从用户的角度来看，看一封保存的电子邮件不是三个独立的功能（选择邮箱、对邮箱中的邮件分类和阅读消息），而是一个单独的活动。单一窗口不仅足以完成这个任务，而且更紧密地符合用户的目标和浏览邮件的心智模型。但是，实现它的程序员却忠实地向用户呈现了它的实现过程，这有点像强迫司机开两个方向盘，每个方向盘控制一个前轮，而不是将两项功能组合成单一概念上的整体。

对于 Navigator 的问题，较好的解决方法是创建单个邮箱，然后把一些用于管理搜索的工具，策略性地放在顶部一行——工具条就很理想。搜索的中间结果可以和最终消息一起显示在窗口中。寻找和阅读消息是一个目标，应该由一个对话框来实现。这方面，微软的 Outlook 向前迈进了一大步：多个邮箱、每个邮箱的邮件列表和邮件消息本身都可以通过一个多窗格的独占窗口访问，设计上，这个窗口保持最大化状态。回复消息时打开一个附加窗口。

没有什么好的方法可以明确展示多个窗口之间的联系，所以不要创建太多的窗口。这也是 Visual Basic（VB）中一个特别令人烦恼的问题，VB 很容易创建**窗体**（form）。窗体是独立的顶层窗口，行为方面与非模态对话框相同。将程序创建为多个非模态对话框的集合一直是一项值得怀疑的策略，在 VB 让它容易实现之前，从来没有流行过。容易实现并不意味着它就一定是好的设计。模态对话框总是使你立即返回出发点，所以它不会为你带来其他负担，但窗口不同：每增加一个窗口都会增加用户管理窗口的附加工作。如果是常用程序，那这方面的负担会增加到令人厌恶的程度。

一个 VB 程序员曾经说过，因为他的程序有 57 个窗体，所以设计起来特别困难。没有任何程序可以有效地使用 57 个窗体，每个窗体就其本身也许是优秀的，但组合起来，就太多了，这个道理就和你一口气品尝 57 瓶上好的夏敦埃酒，以及在星期六试开 57 辆私家轿车一样。

窗口状态

程序员将应用程序的主窗口称为**顶层窗口**。顶层窗口有与其他顶层窗口层叠的内在能力，但这并不是它通常的使用方式。根据编码方式的不同（在 Windows 和某些 UNIX

图形用户平台中），每个顶层窗口可以成为三种可能状态中的一种。奇怪的是微软只命名了这三种状态中的两种：最小化和最大化。

UNIX 图形用户界面如 Motif，以及 Windows 95 以前的版本，**最小化**窗口缩小为矩形图标（通常比正常的桌面图标大一点）堆放在桌面上。Windows 95 中，最小化窗口是将窗口变成任务栏上的一个按钮。**最大化**窗口充满整个屏幕，屏蔽任何其他内容。

不知为何，微软设法避免直接提到第三种状态，只是在系统菜单中（点击标题栏左上角可以看到）提示它的名字，菜单里的**还原**（restore）描述了如何获得第三种状态。这个功能将顶层窗口最小化和最大化窗口还原成其他状态。为更加符合情理，我们将第三种状态称为**多元**（pluralized）状态，尽管它也称为还原状态。

多元状态是一种中间状态，窗口既不缩小为图标，也不最大化覆盖整个屏幕。当窗口处于多元状态时，它与另一些图标以及其他在多元状态的窗口共享屏幕。在 Mac OS X 中，如果窗口没有都图标化（Mac OS 9 只允许窗口折叠成标题栏，而不能完全图标化），那么所有的窗口都处在多元状态。多元状态下窗口既可以平铺，也可以层叠。

回顾一下 Windows1.0，最小化和最大化状态分别称为图标化和放大化。这些术语更具描述性，因而也更吸引人。后来 IBM 与微软关系缓和，在误以为美国的主管们会更满意的情况下，要求改为统一的表达方式，更差的名称就这样保留下来了。

独占应用程序的正常状态是最大化。除了支持程序之间的切换，或者在程序和文档之间拖放数据（后者可以用工具栏控件完成）外，几乎没有理由使独占应用程序处于多元状态。一些暂态应用程序，如文件管理器或者浏览器最适合处在多元状态，它们主要用来作为独占程序的跳板。

为什么最小化

Windows 的任何应用程序都可以最小化，但是为什么要这么麻烦呢？在没有任务栏或类似习惯用法的系统中，常需要借助最小化这种不灵活的方法从一个程序切换到另一个程序。先最小化当前活动程序（它可能是最大化的），然后使你所期望的程序图标最大化（或者使之处于多元状态）——如果它还没有最大化或处于多元状态，并且也没有隐藏在先前的应用程序后面。为了切换回来，必须采用相反的顺序。你不得不满屏幕地移动鼠标，整个过程即缓慢、复杂又无聊。

Windows 3x 中 Alt+Tab 键组合是在应用程序之间切换的好方法（甚至 XP 仍然支持这种功能）。但是它比较晦涩，又没有可视化，还要求用户的经验比较丰富。在超级用户之外，很少有人知道这种技能，而且它的操作方式也与 Windows 任何其他习惯用法都不

同。按住 Alt+Tab，你会迅速地直接移到下一个运行程序。按住 Alt 键，再反复按 Tab 键，你可以在每个运行程序之间循环移动。它在屏幕中央的一个小窗口中显示候选程序的名字和图标。这里的诀窍在于：当用户释放 Alt 键时，真正选中的程序才出现在屏幕上，而在 Windows 的其他任何地方，释放换档键时都不会有什么动作。这个习惯用法有些不可思议，大多数人都不知道它，学习起来也比较困难，但学会这个习惯用法后，它会是应用程序间导航的最快方式。除了可以加快程序间切换的速度以外，它带来的最大好处在于：不同的程序可以保持各自的自然状态，无论程序是处在最大化还是多元状态，虽然一般情况下是在最大化状态。

Alt+Tab 习惯用法是一个经典的例子，编程团队就这样巧妙地解决了一个妨碍专家的重大问题。许多聪明的软件设计师试图创建方便的程序切换习惯用法，但没有一个能和它媲美。这种方法是卓越的，但事实上很难发现——它既没有出现在文档中，也没有出现在任何菜单上，所以必须有人告诉你这种用法的存在。这种方法几乎不会带来任何管理负担，但它需要用户能够非常熟悉和灵巧地使用计算机，并且能清醒地控制它——常见的编程人员就是这样。我们真正需要的是一个比 Alt+Tab 更亲切的，不仅仅只是适用于超级用户和电脑黑客的版本。在 Windows 95 中，任务栏正是这样的一个解决方法。

Windows 任务栏最终承认：大多数人希望在最大化的独占窗口中工作，他们需要易于接受的习惯用法来完成这个任务。屏幕上的一个区域为这个始终存在的灰色栏所专用，除了最顽固的程序员之外（他们总是隐藏任务栏），对于人们来说，这个开销是值得的。无论当前所处的状态如何，每个运行程序或打开的文档在任务栏上都有一个按钮（除了精灵姿态程序，甚至有些精灵姿态程序也将它放入这个状态区域）。活动程序的按钮以按下的状态来显示。

任务栏是 Alt+Tab 习惯用法的一种简单的可视化实现方式：你点击想要选择的应用程序相应按钮，它就移动到最前面，处于活动状态。如果它最近一次处于最大化状态，那么它仍然最大化。如果它最近一次处于多元状态，那么它仍处于多元状态，并且所在的位置没有变化，仅仅是移动到一堆窗口的前端。如果它是最小化的，就将打开为先前的最大化或多元状态。可以将运行程序想像为电视机的频道——按下按钮允许你从一个频道跳到另一个频道。

将程序最小化的另一个原因是为了降低屏幕的混乱程度。如果运行了几个处于多元状态的独占程序，你可以将一些程序最小化来简化屏幕。但这种方法治标不治本。如果依次最大化每个应用程序，就不会出现明显的混乱，并且最小化也就不必要了，当然，前提条件是你可以用任务栏或者 Alt+Tab 进行导航。在我们有了任务栏之后，最小化存在的惟一理由是它能帮助我们清空屏幕，以便我们可以使用桌面上的图标。在 Windows 任务栏的快速启动区域，有一个按钮能瞬间最小化所有窗口。另一方面，Windows 2000

和 XP 的任务栏中有一个桌面快捷方式工具栏，这使得我们甚至也不需要这个按钮了（尽管这个特性在默认状态下是关闭的）。

在 Windows 的当前版本中，已经不需要最小化程序这一功能了。已经有足够的工具来像控制电视机频道一样管理程序。长期以来 Alt+Tab 允许超级用户以这种方式工作，而任务栏最终使得超级用户以外的人也能以这种方式工作。

在任务栏出现之前，最小化的应用程序，和 UNIX 图形用户界面平台上的那些最小化应用程序一样，有时也以一些创新的方式来显示动态或动画的状态信息。这种应用在 UNIX 中仍不乏其例，但最终可能会在 Mac OS X 中终结，由于它有最小化的应用程序，但又没有完全最大化的程序。但在 Windows 中，用桌面上的项目来显示有用的信息这种习惯用法几乎已经不存在了，因为程序员意识到最大化的程序窗口几乎总会遮盖这些项目。

为何要多元状态

程序变为多元状态有什么理由吗？也许吧。在有些含糊的情形下，需要两个或更多的程序并列出现在屏幕上。如果用户只是希望依次运行一些独占程序，并且小些的暂态程序也只是暂时覆盖在独占程序之上，那么多元状态就没有存在的必要了。如果用户希望在独占程序之间剪切和粘贴信息，剪贴板就能很好地工作。但是，如果用户希望用到程序之间的拖放功能，那么，涉及拖放的两个程序都必须是可见的，并共享屏幕。换句话说，两个程序必须都处于多元状态。这一点至少在 Windows 任务栏没有出现之前都是成立的，任务栏出现后，拖动并按住对象，放到任务栏上的一个最小化应用，就可以直接打开这个应用。尽管这个习惯用法带来额外的拖动开销，但它消除了窗口管理的附加工作：自己选择吧！

现代的 XGA 计算机屏幕从 640×480 到 1600×1200 像素都有。在 21 世纪初期，1024x768 是最常见的分辨率。在这种限制条件下，现在的一些独占程序，如文字处理器和绘画程序，如果占用的屏幕空间少于半屏，使用起来会非常困难和不舒服。在向媒体和业界精英做展示时，微软骄傲地演示了将电子表格拖放到文字处理器的过程。为了能单独解释这一功能，两个应用程序（Excel 和 Word）的窗口要预先摆放好。将两个窗口变成多元状态，然后手动调节使它们各自获得足够显示空间的管理附加工作，要比使用剪贴板和任务栏以及简单地在两个独占程序间交换的管理附加工作大得多，微软没有告诉你这一点。

程序之间的拖放是一个功能强大的习惯用法，我们可以看到将来它的使用频率还会

增加。可是我们不会看到它频繁使用在独占应用程序之间，除非计算机屏幕能变得比现在大许多，这可能还得等上好几年。但是，程序间拖放为独占程序和暂态程序间的信息移动带来了方便。看看在文字处理器文档中加入一张剪贴画的过程，就能明白为什么。

文字处理器是独占程序，它以最大化状态运行。剪贴画管理器是暂态程序[1]，它通常运行在大约整个屏幕四分之一的固定大小窗口中（窗口的高和宽分别是屏幕的一半）。剪贴画管理器可以任意放置在屏幕上最不显眼的象限内，一旦我们选中目标图像，就可以直接拖动它到文字处理器的合适位置。再次单击，剪贴画管理器就回到任务栏，用户又可以继续编辑。窗口管理成本仅仅是两次单击，一次用来打开管理器，一次用来关闭管理器，以及可能需要一次单击拖动操作，移开管理器使文字处理器的关键区域能更容易看到。两个应用都不需要变为多元状态。

在不提供最小化和多元状态的情况下，也可以有效地创建 Windows 软件。处于多元状态的程序正在消亡，而正是 Windows 在加速这个过程。现在的程序不一定必须经历多元状态。当然，这并不意味着你真的可以完全抛弃这两种状态。为了向后兼容，你必须能够将窗口最小化，当用户需要将程序在屏幕上平铺时，每个最大化窗口必须能够变成多元状态。如果程序不能达到这些基本要求，经验丰富的用户可能会感到失望，即使它更多的是适用性方面的问题，而不是实际的软件设计问题。

出于实际需要，我们只保留了两种程序配置：最大化独占程序和多元暂态程序。独占程序可以允许暂态程序短暂覆盖它。在设计自己的应用程序时，你必须对这一基本设计问题做出决策：这个应用是主导性的还是临时性的。这个决策将决定你在这个应用中使用的主要窗口类型。

MDI 与 SDI

大约 15 年前，微软开始在 Windows 应用程序中实行一种新的方法来组织功能。微软称之为**多文档界面（multiple document interface）**，简称 **MDI**。MDI 满足了某类应用程序，也就是同时处理同类型文档多个实例的应用程序需要。著名的例子是 Excel 和 Word。

微软将代码内建在操作系统中来支持新的标准，MDI 作为一个标准出现是必然的。到八十年代后期和九十年代初期，MDI 被微软公司里的某些人当作包治用户界面百病的良药。所有各式不同的问题都开这个药方。

如今，微软已经对 MDI 进行了反思，接受了单文档界面（single document interface），

[1] 译者注　暂态程序覆盖在文字处理器上，所以文字处理器不需要变为多元状态。

简称为 SDI。毕竟 MDI 没有解决所有的问题。

如果想从电子表格中复制一个单元格，将它粘贴到另一个电子表格，那么你必须依次打开和关闭两个电子表格，这非常令人讨厌。如果同时可以打开两个电子表格就好多了。有两种方法来做到这一点：你可以有一个电子表格程序，包含两个以上的表格实例。或者你可以拥有多个电子表格程序，每个程序只有一个表格实例。第二种选择具有技术优越性，但需要高性能的设备。

在 Windows 早期版本，微软出于简单实用、节约资源的考虑选择了第一种方案。包含多个电子表格（文档）的单个程序与多个程序相比，可以节省字节和 CPU 周期，但却有性能方面的问题。

不幸的是，一个程序多个文档的模型违反了 Windows 早期建立的基本设计规则：每次只能激活一个窗口。这里需要一种方案能够每次激活一个程序，同时每次激活这个程序中的一个文档窗口。MDI 这是实现这个方案的一种黑客工具。

在 MDI 成为标准以后的这些年，出现了两种情况。首先，善意但被误导的程序员灾难性地滥用了这种工具。其次，相对于 MDI 出现的年代，我们的计算机如今已变得强大了很多——具有多个实例，每个实例有一个文档的程序已经非常切实可行。因此微软做出澄清：MDI 即使不是注定消亡的，至少在策略上不再是正确的。

尽管微软的方向改变了，但只要 MDI 不被滥用，它的存在还是很合理的。滥用的主要方式是一个程序中有一种类型以上的文档窗口。图 25-2 就是一个例子。CompuServe Navigator 程序提供了一打以上不同类型的文档窗口，使用户理解起来非常困难（如今 AOL 也是这样）。这是为什么许多设计师愿意看到 MDI 废弃的主要原因之一。但在像文字处理器和电子表格这样的独占程序中，MDI 并无不妥，只要文档只有一种类型，最小化/多元状态的功能得到抑制，更多地支持从窗口菜单或者一系列标签中选择最大化文档。否则，功能变得不再锐利，导航变得沉重，困惑就会产生。加强这种限制有两重原因。首先，当选择不同类型的文档窗口时，菜单必须随之改变。用户依靠菜单的持久性帮助他们在屏幕上定向。改变菜单就会失去这种可靠性。第二，我们前面讨论的最小化、最大化和多元窗口中所描述的一切问题，在 MDI 程序文档窗口中都会加倍严重。用户不得不在一个大窗口中管理小窗口，这是一个真正可恶的附加工作例子。干净利索地从一个窗口切换到另一个窗口要好得多。在完全最大化的电子表格之间切换有力而高效。

如今微软在实现 MDI 和 SDI 方面没有明显区别。在多数微软的应用程序中，你可以通过窗口菜单或任务栏切换电子表格，也可以通过任务栏从 Excel 切换到 Word。

26

使用控件

控件是允许用户和软件进行交流的屏幕对象，它具有可操作性和自包含性。无论你将它们称为控件、小部件（widget）、小配件（gadget）还是小零件（gizmos）——它们是创建图形用户界面的主要构造模块。它们与图形用户界面的发展密切相关，与窗口、菜单和对话框一样，是图形用户界面构造的基础。

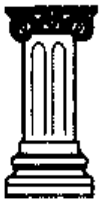
根据用户目标，控件可分为四种基本类型：**命令控件**（imperative control），用于启动功能；**选择控件**（selection control），用于选择选项或数据；**输入控件**（entry control），用于输入数据；**显示控件**（display control），用于可视化地直接操作程序。有些控件组合了一种或者更多的类型。

我们熟悉的大多数控件来自 Windows、Mac OS 和其他通用窗口界面的标准。这些预先定制的控件集合在作用范围和能力方面非常有限。

避免布满控件的对话框

在大多数窗口应用里最容易创建的是对话框。对话框提供了自动工具来指定如何及在什么地方布置控件。对话框的标准定义就是布满控件的模态窗口。很显然，程序员模

仿其他界面在布满控件的对话框基础上创建用户界面，是非常轻松的。同样明显的是，使用其他可视化直接操作习惯用法创建窗口界面，是很困难的。因此大多数本地 GUI 系统把交互分为两大类：极容易实现的预定义控件，包括单选按钮（radio button）、复选框（check box）、下拉菜单（drop-down menu）等等，以及难于实现的直接视觉交互（direct visual interaction）。与此一致的是，大多数已有的文档很好地讲述了预定义控件，而忽略了其他方法。然而布满控件的对话框并不是成功用户界面设计的关键。我们会在第 30 章和第 31 章更多地讨论如何合理使用对话框。



公理：大多数布满控件的对话框并不是好的用户界面设计。

作者并不是建议消除标准控件。然而，虽然使用这些控件能够确保实现起来很容易，但无法确保使用也会很容易。就像所有好的界面元素一样，控件也必须明智地使用而且是在恰当的场景中。

我们会更加详细地讨论四种控件类型——命令控件、选择控件、输入控件和显示控件。

命令控件

在人机交互中，有一种由名词（有时候称为对象）、动词、形容词和副词组成的语言。调用命令时，我们指定了动词——动作声明。当描述动作会影响到什么时，我们在指定组成句子的名词。有时候我们从已存在的列表中选择名词，有时候输入一个新名词。我们分别用形容词和副词来修饰名词和动词。

与动词相对应的控件类型叫做命令控件，因为它产生立即的动作。命令控件采取动作并且立即进行。菜单项（menu item），我们将在第 27 章和第 28 章讨论，也属于命令（imperative）习惯用法。在控件世界里，命令习惯用法的典范是按钮。实际上，它是惟一一种，虽然存在多种不同的外表。单击按钮及其相关动作，也就是动词，将立即执行。

按钮

下压按钮一般通过仿造的三维凸起特征来进行识别。如果控件是矩形（或者有时呈

椭圆形），并凸起显示（根据右侧和底部的阴影，以及顶部和左侧的加亮），那么它就具有命令的视觉启示[1]。用户只要单击后释放鼠标光标，它就立即执行。在对话框中，一个**默认按钮**（default button）常常被突出显示来指示用户可以采取的最合理动作。

可以证明，在设计师的工具箱中，下压按钮是视觉上最引人注目的控件。因此在用户界面中演化出如此多的变体也就不奇怪了。当代人造三维按钮的操作启示促进了它的广泛使用。它是好东西——那么为什么不多用它？甚至 Web 上的超级链接设计师也借用了下压按钮的外观，尽管经常以受范性反馈为代价。

下压按钮的部分启示（affordance）是它的视觉受范性（visual pliancy），显示它的“可按压特性”。当用户指向并单击鼠标键，屏幕上的按钮视觉上会发生改变，从凸起变为凹下，显示它正在激活。正如我们在第 5 部分中讨论过的，这是一个动态视觉暗示的例子。设计糟糕的程序，以及许多网站上绘制的按钮，单击时却没有发生改变。对于开发者来说（特别是在 Web 上），这样做既便宜又容易，但对用户来说则非常令人不安。因为它不禁让人产生疑问：“它确实做了什么吗？”用户希望看到按钮改变——受范性反应——你必须满足他的期望。

这在网页和多媒体应用程序中是很重要的。因为许多网页和多媒体应用程序出于艺术和技术的原因，不得不违背桌面受范性的规则，但仍让一部分显示对单击敏感。

在这些情况下光标暗示是不够的，尽管它是令人满意的补充。即使整个屏幕都布满了拼贴画，如棒球运动的收藏品等，当用户单击 Louisville Slugger[2]来完成某项功能时，球棒也应该在视觉上进行移动，让用户确认这是一个命令按钮——或者在本例中是一个可以按动的球棒。

图标按钮

在发布 Windows 3.0 的同时引入了**工具条**（toolbar）（我们将在第 29 章进一步讨论），这是一个已成事实标准的习惯用法，和菜单栏一样为人熟知。按钮从传统的对话框移居到了工具条。在这个过程中，按钮显著地扩展了它的功能、作用和视觉特征。

在对话框中，按钮呈矩形（Mac 中有弧形的边），有专有的文本标签。当它移到工具条上时，变成了方形，没有文本，而获得了象形文字，一个图标。因此**图标按钮**（butcons）就诞生了：一半是按钮，一半是图标。在 Windows 98 中，图标按钮，或者说工具条按钮，

---

[1] **译者注** 提供物的定义见第 20 章。

[2] **译者注** 美国职业棒球大联邦 MLB 指定球棒。

继续发展，没有了凸起的启示，除非在使用时——这种改变减少了视觉上的混乱，避免工具条过度拥挤。遗憾的是，这也使得初学者难以理解。现在 Windows 2000 的工具栏按钮只有在指向时，才显示它的按钮启示。Windows XP 的图标按钮继续演化远离它的按钮特性。Office XP 应用程序图标按钮有一个令人困惑的视觉暗示，在小的方形平台上把图标从工具条的表面抬高。作者感到奇怪，Microsoft 为什么不适可而止，而继续在视觉上将图标按钮与其按钮功能紧密相关。

在理论上，图标按钮很容易使用：它们总是可见的，不需要花费太多的时间，也不像下拉菜单那样需要一定的灵敏度。因为它很常见，易于记忆，尤其在独占应用程序中。图标按钮的优点很难与工具条的优点区分开来——两者之间有着解不开的联系。始终困扰图标按钮的问题不是来自按钮部分，而是来自图标部分。我们迅速理解视觉启示——它好像在叫“单击我!”（无论如何，在 Windows 98 之前是这样做的）。问题在于图标按钮的表面图像很少会那样清晰。

总之，图标难以确切解释，动词图标比名词图标更难解释。因为图标按钮是命令控件，这些图标代表动词，因而存在问题。真正的问题不是说找到更好的视觉隐喻或更好的图片创造者就可以了，而是，在图标有限视觉分辨率的情况下描绘动作及其关联非常困难，如果不是完全不可能的话。如果你能找到一个恰当的图标，它也具有很好的记忆特性，但是要想向初学者表达图标按钮的意图，常常还是不够的。

图像确实有很好的记忆特性，但这样也会产生进退两难的局面。因为这些特性非常好，视觉图像不仅足以提醒日常用户图标按钮所代表的命令，而且，视觉图标按钮比旧的文本图例按钮（text legend butcons）要节省空间。只要开始的时候有学习途径，它就会成为用户命令工作集（working set）的一部分，用户会把图像当做习惯用法记忆，即使缺乏天生的学习能力也不会有问题。因此要想成功，图标按钮的图像必须在视觉上与其他图标按钮进行区分，并且可以作为习惯用法记忆。

然而，如果没有任何机制可以解释它们的意图，图标按钮和工具条将发挥不出他们本身的用处。正因为这些原因，最初工具条上的图标按钮迅速普及的时候，人们普遍抱怨图标不好理解。为了解决这些问题，一些公司放大他们的图标直到能包含文本图例和图标。然而，其他公司则要求用户选择哪些命令显示为图标按钮，因而在界面上添加了另外一层附加工作。但微软的工具提示（ToolTip）巧妙而一劳永逸地解决了这个图标按钮令人费解的问题。工具提示提供了开始的学习方法，而不会妨碍经常使用的用户。自工具提示出现以来，对图标按钮令人费解的抱怨也逐渐平息下来。

我们将在第 29 章进一步讨论图标按钮、工具条和工具提示。

选择控件

因为命令控件是动词，它需要一个名词来进行操作。选择控件和输入控件是两类用于选择名词的控件。**选择控件**（selection control）允许用户从一组有效的选项中选择一个操作数。

选择控件与动作无关。选择控件可以提出单一选择（用户只说“是”或者“不是”），或者提出一组选择（用户可以根据控件设置，选择一个或多个选项）。列表框（list box）和复选框（check boxes）是选择控件中典型的例子。

复选框

复选框（check boxes）是最早发明的视觉控件习惯用法之一，它因为提供了简单的二者择一而受到宠爱。复选框有强烈的可供单击的视觉启示。它存在受范区域，因为当鼠标经过时会突出显示，或者有三维凹进的视觉处理。用户单击后，可以看到一个检查标记（check marker），你已经学会了让它按你的意愿工作所需要知道的全部知识：单击做标记，再次单击除掉标记。复选框是简单的，可视的，优雅的。

然而，复选框主要是基于文本的控件。作为视觉上可识别的图标，可复选的框（checkable box）紧挨着用于区分它的文本。它的工作方式同于列表框（list box）文本项（text item）左侧的图标，可从视觉上帮助用户区分类型。但与其他列表框输入一样，是图片支持文本，而不是其他方式。复选框是熟悉而有效的习惯用法；但它和菜单一样，存在优点的同时也存在缺点。确切的文本使复选框清楚明确，但也使用户不得不放慢速度阅读，而且占据了数量可观的屏幕空间。

传统上，复选框是方形的。用户通过形状来识别视觉对象，因而方形复选框是一个重要的标准。方形本身没有好坏之分；只是最初选择的形状是方形，而许多用户也已经习惯去识别这种形状。没有充分的理由要摒弃这种模式，无论推销人员和形象艺术设计人员怎么说，也不要将复选框设计成钻石形或圆形。

也许我们能像在菜单中使用图标按钮一样改进复选框。也许我们能够开发一种复选框，用图标代替文本。也许，我们在将复选框图标化方面不会走得太远，但我们可以用另一种演化的习惯用法：用图标按钮来取代复选框。

用图标代替文本，按钮就演化成图标按钮，迁移到工具条上。于是，按钮继续演化为**锁定图标按钮**（latching butcon）（如图 26-1 所示），使用简单的方法允许按钮单击时保持凹进——或者按下状态，再次单击时则凸起。锁定图标按钮状态不再是瞬间的，而是

锁定直到再次单击。控件特征也完全改变，成为截然不同的类型，命令控件成了选择控件。

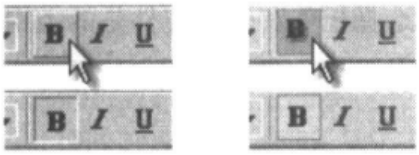


图 26-1 图为 Windows 2000（左侧）和 XP（右侧）中鼠标经过（凸起）和单击时的状态。左侧的按钮启示更明显。图中的控件也是锁定图标按钮的例子，使用简单的方法“在单击后不允许按钮回到凸起状态”而发明的。这个习惯用法的非凡之处在于将图标按钮习惯用法从命令型（指定动词）变为选择型（指定名词）。除了设置状态而不是启动直接命令以外，它具有图标按钮所有习惯用法的特性和节约空间的优点。

微软 Office 程序套件中默认的工具条配置似乎默许区分瞬时命令型图标按钮和锁定选择型图标按钮。一般情况下，微软只将命令型图标按钮置于顶部工具条中，而多数选择型图标按钮则放在其他地方。

锁定图标按钮作为单次选择习惯用法广泛地取代了复选框，特别适合用在非模态交互中，这样不会打断用户做决定时的流状态。锁定图标按钮比复选框更节省空间：因为它们依靠视觉识别而不是文本标签来表明意图，所以体积更小。当然，这也意味着它们存在和命令型图标按钮一样的问题：图标的费解性。工具提示再次挽救了我们：那些小的弹出窗口给了我们足够的文本来澄清对图标按钮的困惑，而不需要长期占据太多像素。

触发按钮：一种应该避免的选择习惯用法

触发按钮是一种用于节约界面空间的最常用控件变体，但这却让用户极度困惑。触发按钮上的动词总是控件所能处在的多个状态之一。例如，如果设计的触发按钮用于控制打印分辨率，它可能一直显示草稿输出，直到你单击它，才可能显示高质量输出。

控件告诉你可以单击它，所以当它显示高质量输出时，意味着单击后你会进入高质量输出模态。然后，按钮将改为显示草稿输出，那就表明单击后应用程序将处于草稿输出模态。这个技术存在的问题在于人们可能认为控件是显示当前状态的指示器。遗憾的是，当你处于高质量输出模态时，它显示草图输出，或者相反。事实上，它总是与你期望的相反。控件或者作为状态指示器，或者作为状态转换的命令控件，但不能同时表示两者（如图 26-2 所示）。

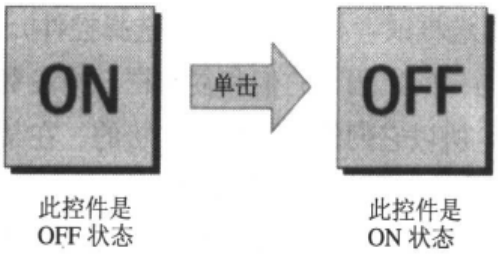


图 26-2 触发按钮控件效率很高。它们通过在一个控件内控制两个相互排斥的选项来节省空间。触发控件存在的问题在于它们无法完成控件的第二种职责——告诉用户它们当前的状态。如果当前处于 off 状态，按钮显示 ON，那么就无法知道状态是什么。如果当前处于 off 状态，它显示 OFF，那么 ON 按钮又在哪里呢？不要使用触发按钮控件，无论是在按钮中还是在菜单中！

这个问题的解决方法要么是在按钮上拼写出动词短语——转换为高质量输出——要么完全采用其他技术。用两个单选按钮取代它是一种常见的选择，这样做的不利因素是消耗了更多的屏幕空间。

另一个方法是图示法（pictorial）。画一张竖向纸张的图片，当用户单击时，它翻转到侧向显示你当前处于横向。这是非常难忘而令人愉悦的，但它并不一定要非常容易发现。取决于程序的其余部分将如何影响用户，他们会不会期望一幅小图片具有受范性而且会有一些效果。光标暗示有所作用，然而，重要的不是将图片放在按钮之上。如果那样做，你只不过是创建了一个象形文字的触发按钮，它和那些文本标签一样传达着冲突的信息。

单选按钮

复选框的一个变体是**单选按钮**（radio button），它的名字说明了一切。当汽车第一次装上收音机时，我们就发现在行驶过程中旋转按钮手动调频会危及生命。所以汽车收音机都提供了一种新奇的仪表盘，由六个镀铬合金的按钮组成，每一个都有事先调好的波段。现在你的视线不需要离开路面，只要按下一个按钮，就可以调到你喜欢的频道。这种习惯用法功能强大，在交互设计中仍有很多实际应用。

单选按钮的行为是**相互排斥**（mutually exclusive）的，这意味着选择一个选项时，以前选择的选项会自动取消。每次只有一个按钮可以选择（技术人员常用**互斥**（mux）这个词作为“相互排斥”这个词组的简称，我们也常用它来指这些控件）。

因为相互排斥，单选按钮总以两个或多个成组出现，而且每组中只有一个单选按钮可以选中（从技术上说，这种相互排斥并不具有强制性，“总有一个选中”的规则也不具强制性，开发人员个人可以自由地打破这些规则）。单个单选按钮没有定义——相反它必

须表现得像复选框一样（在这个例子中你应该使用复选框或者相似的非互斥选择控件）。

单选按钮甚至比复选框更浪费屏幕空间。基于同样的原因，它们和复选框一样浪费同样数量的空间，但单选按钮只有成组时才有意义，所以它们的浪费总是加倍的。在某些情况下，这种浪费是值得的，特别是在向用户显示全部可获得选项的集合时非常重要。听起来它像是模糊的教学法，事实也是如此。单选按钮很适合担当教学的角色，这也意味着它们可以合理地用于不常使用的对话框，但它们不应该出现在独占应用程序的界面中，因为它必须迎合日常用户的需要。

与传统上复选框呈方形的原因相同——就是因为我们已经这样用它——单选按钮是圆形的。除了美学或市场的原因之外，没有理由去改变这种形状，而这些原因都要让位于已经建立的传统（X Windows/Motif 的单选按钮是钻石形的，所以如果你为 UNIX 系统做设计，就可能要考虑遵守这种风格。）。

单选按钮是最古老的图形用户界面习惯用法之一，因此许多设计师认为它比其他新的习惯用法更好，但事实并非如此。在某些情况下，单选按钮正在被更多的现代习惯用法所取代。

你可能会想像，图标按钮就像对复选框那样，也能为单选按钮做些什么：在应用程序界面上取代它。如果两个或两个以上的锁定图标按钮按组放在一起，并且相互排斥——以致每次只能锁定其中之一——它们的行为方式确切地说与单选按钮相同——就构成了一组**单选图标按钮**（radio butcon）。

单选图标按钮像单选按钮一样工作：总有一个选中——锁定——而且无论何时当另一个选中时，先前的按钮就返回正常——凸起——状态。Word 工具条上的对齐控件（alignment control）是一个恰当的单选图标按钮例子，如图 26-3 所示。

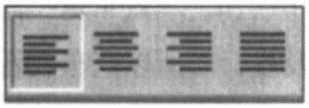


图 26-3 Word 的对齐控件是一组单选图标按钮，和单选按钮相似。总有一个图标按钮选中，当单击另一个图标按钮时，先前的那个按钮就返回正常凸起状态。这种变体是一种节省空间的习惯用法，很适合经常使用的选项。

和所有的图标按钮习惯用法一样，它们有效地利用空间，让经验丰富的用户依靠模式识别（pattern recognition）来分辨它们，同时使用工具提示提醒临时用户它们的用途。有些新手用户很聪明，借助工具提示就可以学会，有些可能会学得更慢一些，但像其他平行的教学命令向量（command vector）一样可以确保用户学会。

组合图标按钮

单选图标按钮的一种变体是下拉版本的。由于它和组合框控件相似，我们称为**组合图标按钮**（combutcon）。如图 26-4 所示。通常，它是一个右侧有向下小箭头的单一锁定图标按钮（在 Windows 中），如果你单击箭头并停住，就会下拉出一个菜单，包含数个（有时是许多，呈二维数组排列）锁定图标按钮。你可以像在下拉菜单中一样滑动光标，在组合图标按钮的下拉列表/数组中选择一项。当你释放鼠标键时，就做出了选择。选定的图标按钮就出现在工具条的箭头旁边（如果单击图标按钮，就会切换图标按钮状态）。像菜单一样，图标按钮的菜单也应该在用户单击箭头再释放后展开，第二次单击才做出选择。

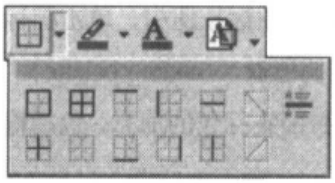


图 26-4 组合图标按钮是一组行为类似组合框的锁定图标按钮。单击组合图标按钮的向下箭头就会下拉图标按钮菜单。移动光标到期望的项然后释放鼠标。新选中的图标按钮作为选择项出现在工具条上。可以把这种习惯用法想像成在单个图标按钮中填进许多相关的锁定图标按钮。与创建单独的图标按钮相比，它产生了更多的用户附加工作。但在空间成本较大或者功能很少使用的情况下，它非常有用。

组合图标按钮的变体包括在组合图标按钮右下角向下箭头处（这可以在微软的工具条中发现）画一个小的向下或者向上的三角。Adobe 产品在它的工具栏控件中使用了这种变体；这种变体也要求一次单击并且按住图标按钮弹出菜单（如图 26-5 所示，在 Adobe 工具栏中，菜单向右而不是向下展开）。你可以稍微调整这种习惯用法，具有创造力的软件设计师总是不断探索，将更多的功能填满总是太小的屏幕。

你可以在 Word 中发现微软的变体，这里指定突出显示颜色和文本颜色的图标按钮看起来更像小调色板，而不是图标按钮堆积的组合图标按钮菜单。从图 26-5 中可以看到，这些菜单可以将很多效能和信息堆积为紧凑的控件。这种设施肯定是为频繁使用的用户，尤其是经常使用鼠标的用户设计的，而不是为新手用户设计的。然而，对于那些基本熟悉已有工具的用户来说，这种习惯用法通常在自己发现或有人示范以后，会马上变得很清晰。对于用户需要长时间交互的独占姿态应用程序来说，这是一个极好的常用控件。使用菜单上较小的目标要求手工充分灵巧，但与去掉工具栏和下拉菜单，选择某项，以

及等待对话框出现，选择对话框上的颜色按着单击“OK”按钮相比，会更快。

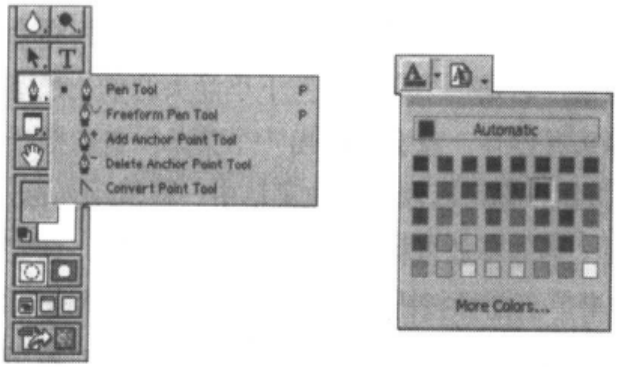


图 26-5 这两个组合图标按钮分别来自 Adobe Photoshop （左）和 Microsoft Word（右）。Photoshop 中组合图标按钮向右侧展开，每个图标按钮都有文本描述。在 Word 的组合图标按钮中，单击图标按钮出现一个可按虚线撕开 （tear-off） 的菜单。它可以成为一个悬浮调色板（floating palette），颜色可以从顶部样本或者文本项中选择，它们弹出颜色选择对话框。这种密集打包的信息，包括输入和输出，显示了 15 年以来优良用户界面的发展方向。

列表控件

列表控件允许用户从有限的文本字符串中进行选择，每个文本字符串代表一个命令、对象或属性。这些控件有时称为**选择列表**（pick list），因为用户可以从它们提供的项目列表中选择一项；根据你所讨论的平台和控件变体，它们称为**列表框**（list box）或**列表视图**（list view）。列表控件是简化交互功能的强大工具，因为它排除了做出错误选择的可能性。

列表控件是一个很小的文本区，右侧有一个垂直滚动条。应用程序在框中以离散的文本行形式显示对象，可以通过滚动条上下移动。用户每单击一次选择一行文本。一种列表控件变体允许多重选择，用户单击鼠标的同时按住 Shift 键或 Ctrl 键一次可以选择多个项目。

早期的列表控件只能处理文本，这种决策至今仍影响着它们的行为。列表控件包含一行接一行没有变化的文本，视觉上就像干燥的沙漠。一些富有冒险精神的程序员将一些老的列表控件改为图片式，但大多数程序员可能不愿那样做。

然而，微软似乎在 Windows 95 中做了很好的工作，它使用了列表视图控件。除了其他特征以外，它允许每行文本点缀一个无须定制编码的图标。这是一个好消息，因为在很多情况下，用户会从紧挨着重要文本项目而带有标志或者提供信息的视觉图标中受益。

图 26-6 显示了带有图标的可修改列表视图控件。

图 26-6 通常选择是一种互斥操作。为了提供多项选择，需要放弃互斥性，如果某些项目滚屏出你的视线，事情会变得令人困惑。耳号是一个解决方法。每个文本项旁边加一个复选框，使用它们代替选择来指示用户的选项。显然复选框是一个非互斥性习惯用法，并且也是人们非常熟悉的图形用户界面习惯用法。用户能很快掌握这种习惯用法。

设计技巧：用生动的图标区分重要的文本项。

列表视图名副其实，对显示列表项有好处，允许用户选择一个或多个项。它也是一个好的提供可拖动项源（source）的习惯用法。如果列表视图中的项可以拖动，那么它就为用户提供了一个好工具，将项按特殊顺序排列（见本章后面的“排序列表”）。

桌面中显示的许多列表不是静态的，它们支持用户通过添加、删除和改变文本条目进行修改。为支持这种功能需要直接进入视图列表，键入新的文本项或者修改现有项。幸运的是，除了图标，列表视图还支持拖放和现场编辑（edit-in-place）的能力。

历史和习惯的力量仍在许多对话框中沿用旧式列表控件。幸好，大多数对话框迅速出现，允许用户选择某个项目，然后很快消失。为了提供更好的交互，程序员应该只用新的视图列表控件或者其他平台上的等价物。

【耳号（Earmarking）】

总的说来，用户选择列表控件中的项目就像输入一些功能。比如说从数种可选字体列表中选择希望的那种字体名字。在列表控件中进行选择的常规方式包括键盘等价物（keyboard equivalent），矩形焦点（focus rectangle），对选中的项目用突出的颜色显示。

但偶尔列表控件也用来选择多个项，这样问题就复杂了。列表控件中的选择习惯用法非常适合单一选择（single selection），但对于多重选择（multiple selection）就相对不太适合了。一般来说，如果同时可以看见整个情景，选择多个离散对象是可以的，就像

桌面上的图标一样。如果同时选定了两个或者两个以上的图标，你可以清楚地看到，因为所有的图标都是可见的。

但如果可用的离散项目集合太大，不适合在单个视图显示，而且其中的一些项目必须滚动才能看见时，选择习惯用法立刻就不适用了。这是列表控件问题中常见的情况。标准的选择模态是相互排斥的，所以当你选择一个事物时，先前选定的事物就会取消。在多重选择时，对于用户来说非常容易发生这样的情况：用户选择了一项，滚动，然后看不见了，再选择第二个项，因为不能再看到前面的项目，用户忘了现在已经取消了对第一个项目的选定。

另一个替代方法也不好：在选择算法中通过编程禁止标准列表控件的相互排斥行为，允许用户愿意选多少项就单击多少项，并保证它们不会取消选定。现在似乎万无一失了（有几分）：用户一个接一个的选择，每个都保持选中状态。美中不足的是没有与标准选择相区分的视觉暗示。很有可能一个用户选择某项，滚动屏幕到其他位置，发现一个更满意的项并选择它，他希望第一个项——这个时候看不见——能够通过标准的相互排斥而自动取消选定。你得选择冒犯前半部分用户，还是后半部分用户。真是个馊主意。

当对象滚动出屏幕，多重选择需要更好更独特的习惯用法。正确做法是采用不同于简单选择的习惯用法，视觉上截然不同。但它究竟是怎样的呢？

碰巧，我们已经有另外一个标准的习惯用法来指示某些事物的选择——复选框。复选框清楚地表达了它们的目的和设置。和所有好的习惯用法一样，它易于学习。复选框也清晰地区分于任何相互排斥的提示。如果我们为列表控件的每个项添加一个复选框，用户不仅清楚地看见哪些项已经选择，哪些项没有选择，而且可以清晰地看见项目之间没有相互排斥。这样就一举两得了。这种可用于多项选择的复选框称为**耳号**（earmarking），图 26-6 就是一个例子。

耳号还解决了多重选择的另一个小问题。多重选择列表控件在创建时没有选中的项。但是，在一些变体中，一旦用户选择了某项，就没有办法返回初始状态（没有选择项）。换句话说，没有不做选择的习惯用法。列表控件用于功能对话框中的操作数的选择时，如果用户改变了主意，取消按钮提供了逃跑路径。但是如果列表控件不是在对话框中，它就僵住了（理想情况下，即使对于这种情况，再次单击所选项也应该取消选定）。耳号采用了不同于选择的规则，列表中的每个项目都是独立的。单击一次做标记；第二次单击取消标记。

有时，在一些画图程序中，如 PowerPoint，在列表的顶部存在“无”或“没有颜色”项目是有意义的。在这种特殊情况下，“无”是一个有效选择，因为你可能需要没有轮廓或填充色的对象。但是不要将这种习惯用法当做列表中取消选定项的通用方法，特别是在用户需要选择某个项才能继续的情况下：用耳号来代替。

【从列表中拖放】

在直接操作习惯用法中，列表控件可以当做调色板来使用。例如，如果列表是撰写报告程序的一部分，你可以单击某个条目，将它拖动到报告上，增加一个代表该字段的栏。在通常意义上，这不是选择，因为它完全是一种捕获（captive）操作。毫无疑问，如果使用支持拖放的列表控件，许多程序将从中获益。

这种可拖动的项目能帮助用户以他期望的方式收集集合。如图 26-7 所示，有两个相邻列表控件，一个显示可用项，另一个显示已选项，这是一种常见的图形用户界面习惯用法。在它们之间有一个按钮，有时是双向的一对按钮，允许选择项目并且从一个框转移到另一个框。如果习惯用法具有支持框之间直接单击拖动期望项目的能力，而不需要选择和功能调用的中间步骤，就会更令人高兴。

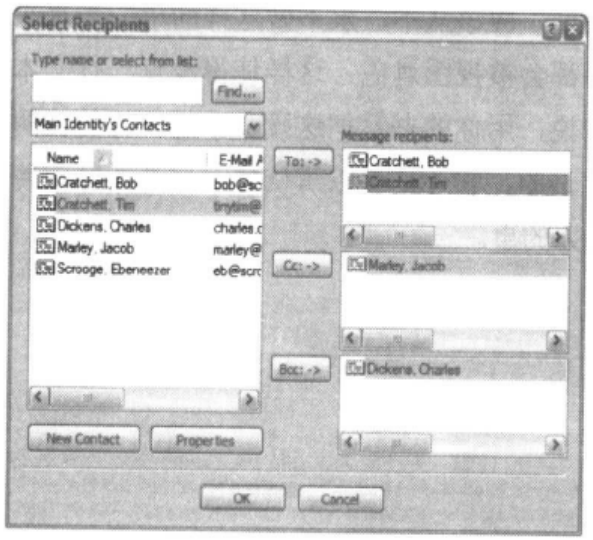


图 26-7 这个对话框来自 Microsoft Outlook Express。它能够从左侧的列表中拖动一个联系人到右侧的 To，Cc 和 Bcc 列表。不过，注意所有的列表域遗憾地使用了水平滚动条（horizontal scrollbar）。不过，左侧的域中，工具提示可以特别显示整行信息。

【排序列表】

有时，在同一个列表控件中需要把项目从一个位置提升另一个位置。事实上，这种需求远远超过多数交互设计师的想像。许多程序为重要的列表提供了自动排序的工具。例如 Windows Explorer 允许按文件名、文件类型、修改日期和文件大小进行排序。这样很好，但是如果能够按照对用户的重要性排序，不是更好吗？从算法方面来说，程序可以按照用户的使用频率进行排序，但那并不总是对的。把最近使用的文件这个因素考虑进去，结果会更接近一些，但仍然不会完全正确（微软在一些应用程序中用这种方式处理字体选择器（font picker），就这个目的而言，它完成得很好）。除了根据下面完整的目录排序，为什么不赋予用户将最重要的文件（对他来说）移到顶端区域的能力，或者单独排序（按字母顺序或者其他任何顺序）？例如，你可能希望对你所在部门的人员列表，按照你所认为他们胜任工作的能力从高到低重新排列。没有这样的自动功能，你不得不

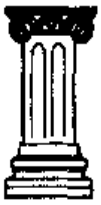
一直拖动，直到正确为止。现在，这是一种定制类型，经验丰富的用户花了几个小时熟悉程序之后希望能这么做。为精细地调节目录需要花很多时间，而且程序必须记住会话（session）之间的确切设置——否则，重新排列的能力就可能没有价值。

在列表控件中，能够将项从一个位置拖动到另一位置很重要，但它要求实现自动滚屏（见第 23 章）。如果在列表中选择了某个项，但你需要拖放的位置位于当前视图之外，那么你必须能够在不放下拖动对象的情况下滚动视图列表。

【水平滚动】

列表控件通常有一个垂直滚动条，用来向上或向下移动列表。它也可以设置水平滚动条。这个特征允许程序员轻松地为列表控件设置超长文本。但是它除了给用户带来痛苦之外，别无它益。

水平滚动（horizontal scrolling）文本很糟糕，应该从不，永不做这样的事。当水平滚动文本列表时，文本显示的每一行前几个字都会被视图遮盖，这样使得没有一行文本具有可读性，文本的连续性彻底破坏了。比方说，用你的书签把这段文字每一行的头两个字遮盖起来，你会发现阅读变得多么困难？是的，它还是可以辨认的，但是会非常费力。而计算机的目的就是要消除人类生活中费力的事。



公理：绝不要水平滚动文本。

如果发现在某种情况似乎需要水平滚动文本，那么赶快寻找替代解决方案。首先要问你自己，列表中的文本为什么会这样长。能减少字数吗？可以将文本换至下一行来减少水平长度吗？可以允许用户为过长的文字选择别名吗？可以用图形条目取代吗？可以用更小一些的字号吗？还应该问自己是否有某些方法扩宽控件。是否可以重新安排窗口或对话框里的事物，在水平方向扩展一些？

最好的答案是将文本换行，对它缩排，使其在视觉上与其他条目不同。也就是说列表控件中的项目会有不同的高度，但这比水平滚动要好。

记住，我们讨论的只是文本。对于图形，一般而言水平滚动条或者水平滚动窗口无所谓。但是提供一个必需水平滚动条的文本列表就像提供一台需要脚踏发电的计算机——这是一个坏消息。

【在列表中输入数据】

为让用户能够在列表控件中直接输入文本而做的工作，历来就非常少。旧式的列表框控件在这方面从没有提供，你得用一些灵巧的代码自己实现。当然，文本输出的地方需要文本输入这很普遍，多数不完善的对话框设计可以直接归因于程序员试图逃避必须编写的现场编辑（Edit-in-place）代码。

然而，Windows 和其他操作系统平台上的现代列表和树形控件都提供了现场编辑工具。Window Explorer 使用了这两种控件。你可以通过重新命名文件和目录来了解它们的编辑方式。要在 Mac OS 或 Windows 95 中重新命名一个文件，可以在希望更改的名字上单击两次——但不要太快（以免错误地解释为双击而打开当前对象）。然后你可以随心所欲地输入文件名。（在 Windows XP 中有一点小变化，在一些视图中你必须右击菜单，选择重新命名才能进入重新命名模态——这是一种进步吗？）其他环境中可编辑的项目在列表控件中显示时，也应该能编辑。

现场编辑成为一个实际问题的边缘情况是为列表添加一个新的条目。多数设计师采用其他习惯用法添加项目：单击一个按钮或选择某个菜单项，为列表添加新的空白条目，用户可以现场编辑它的名字。如果能够，比如说通过双击现有条目之间的空白来创建一个新的空白条目，或者说至少在列表的开始或末端有一个永久的开放空间，在上面有一个“单击添加条目”的标签使它容易发现，那将是更明智的。哈，充满希望的想法……真实世界中解决这个问题的方法是用组合框，我们将在下面谈到。

【组合框】

Windows 3.0 中引入了一种新控件，称为**组合框**（Combo box）。正如名字所示，它是一个列表框和编辑字段的组合。它提供了一个确定的方法在列表控件中输入数据。使组合框成为赢家的另一个特性在于它的弹出变体非常节省屏幕空间。

组合框控件的文本输入部分和列表选择部分之间有明显的区别，减少了用户的困惑。对于单一选择，组合框是非常好的控件。编辑字段可用来输入新的项，它也显示列表中的当前选择。当前选择在编辑字段中显示时，用户可以对它进行编辑——穷人的现场编辑。

因为组合框的编辑字段显示当前选择，所以组合框本质上是单一选择控件。不存在多重选择的组合框，单一选择隐含相互排斥，这也是组合框为什么这么快就取代了由互斥选项组成的单选按钮组的原因之一（在 Windows 有组合框之前，Mac OS 有弹出菜单，它用于取代该平台上大量的单选按钮。然而 Mac 版本不具有组合框编辑特征。）。其他原因包括组合框的空间效率和动态增加项目的能力，以及一些单选按钮不具备的能力。

当使用组合框的下拉变体时，控件显示当前的选择，而不是显示选项列表，以节省

空间。本质上它已经成为一种按需列表（list-on-demand），更像按需提供立即命令列表的菜单，组合框是弹出列表控件。

组合框使用屏幕的高效性允许它为如此复杂的控件做一些不平常的事情：它能合理地永久停留在程序主屏幕上。它甚至能适用于工具条。它是非常有效的控件，适于部署在独占姿态的应用程序中。比如在作者的文字处理器工具条上有四个可见的组合框。它将大量信息高效地填充在一个非常狭小的空间里。在工具条上使用组合框比在菜单上放置相同功能更有效，因为组合框显示当前选择，不需要用户任何动作，如拉下菜单去了解当前状态。该控件用最少量的永久屏幕空间传递最大量信息的优势再次赢得了像素的达尔文战役。

如果在列表中可以实现拖放，那么在组合框中也应该可以实现。例如，能够打开一个组合框，滚动到一个选项，然后拖动选项到一个正在创建的文档中，这是非常强大的习惯用法。因为组合框如此适合用于工具条，该习惯用法真正需要在独占姿态应用程序中添加直接操作的功能。拖放功能应该作为组合框的标准部分。

如果需要多重选择，那么组合框的实用性就会瓦解；该习惯用法完全不能处理，你必须回到普通列表框。常规的列表控件会消耗屏幕上大量的空间，除非有真正的需要，否则不应该永久性地部署它。总之，列表控件最好归为暂时对话框。

树形控件

Mac OS 7 和 Windows 95 都给我们带来了一般意义上的树形控件（tree control），在 UNIX 世界里它的使用已有一段时间了。树形控件是表达层次关系数据的列表视图。它们显示一个侧面的树，每个条目一个图标。条目可以展开或者折叠，类似于文本处理机（outline processor）的工作方式。程序员比较喜欢这种表达方式，它常作为文件导航系统使用。一些人发现这是种有效的显示格式——比分散在桌面上的多个窗口图标要有效得多。遗憾的是，它也给用户带来了问题。因为许多非程序员用户难以理解层次关系数据结构。一般来说，无论树形视图如何有吸引力，只有所表达的事物具有天然的层次关系时（如一个家族）使用它才有意义。如果程序员兴之所至，使用树形视图表达以任意方式组织的任意对象，那么在谈及可用性时会产生大的麻烦。

输入控件

输入控件（entry control）能让用户在程序中输入新的信息，而不仅仅是从已有的列

表中选择信息。

最基本的输入控件是文本编辑字段。和选择控件一样，输入控件向程序表达名词。因为组合框包含一个编辑字段，一些组合框的变体也能作为输入控件。任何允许用户输入数字的控件也是输入控件，如微调控制项（spinner）、标尺（gauges）、滑动块（slider）和调节器（knob）等都属于此类。

有界输入控件和无界输入控件

任何能限制用户输入值大小的控件都是**有界输入控件**（bounded entry control）。例如，一个滑动块从 1 到 100，它就是有界的。使用有界控件时，无论用户行为怎样，程序指定范围之外的数字是不能输入的。因此对于用户来说，他在使用有界输入控件时不可能输入一个无效的值。

相反，一个简单的文本字段可以接受用户键入的任何数据，包括文字与数字的。这种无界输入习惯用法是**无界输入控件**（unbounded entry control）的例子。在使用无界输入控件时，用户很容易会输入无效值。当然，程序在后面可以剔除它，但用户仍可以输入。

简单地说，有界控件应该用在任何需要有数值界限的地方。如果程序只需要 7 到 35 之间的数值，而给用户提供一个可以接受从-1000000 到+1000000 间任何数值的控件，对用户不会有什么好处。她宁愿你提供的是一个上限为 35 而下限为 7 的控件。用户是聪明的，他们会立刻理解和注意到这种限制。

我们指的是输入控件的质量，而不是数据本身，理解这一点非常重要。作为一个有界控件，它需要向用户清晰地传递，最好是可视化的，可接受数据的边界。在用户输入后拒绝的文本字段不是有界控件，而是一种粗鲁的控件。

软件所需的大多数值都是有界的，但许多程序允许数字字段无界输入。当用户无意识地输入一个程序不能接受的值时，程序会发起一个错误信息框（error message box）。这是在以一种“实际上不能却说可能”的方式粗鲁地奚落用户：“您想要什么甜点？我们什么都有，”我们说：“冰淇淋”，你答道：“对不起，我们没有，”我们说：“馅饼呢？”你无辜地问：“没有，”我们说：“小甜饼？”“没有。”“糖果？”“没有。”“巧克力？”“没有。”“那，有什么？”你受伤了，生气地嚷道。“别发疯”，我们愤怒地说：“我们有足够的果盘。”这就是我们在对话框中采用无界编辑字段，而实际上有效值是有界时用户的感受。她输入 17，对于这个无辜的输入，我们奖赏一个错误对话框，告诉她“你只能输入 4 到 8 之间的数值。”这是拙劣的用户界面设计。一个更好的方案是采用有界控件，将输入自动限制在 4，5，6，7 或 8。如果选项的有界集合由文本组成，而不是数字，你仍然

可以使用某种类型的滑动块、组合框或者列表框。图 26-8 所示的是微软在窗口显示设置对话框中使用的一个有界滑动块。它的工作原理类似滑动块或滚动条，但有四个离散位置代表不同的分辨率设置。微软在这种场合也能够轻易地使用非编辑组合框（non-editable combobox）。在很多情况下，滑动块是很好的选择，因为它可以显示有效的输入范围。而组合框并不会更小，但会将一些卡片隐藏起来，直到单击才会显示——这种方式没那么友好。

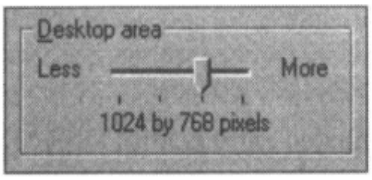


图 26-8 一个只让你输入有效值的有界控件。它不让你输入无效的值，在无效的位置，滑块会拒绝停留。此图显示的是 Windows 95 的显示设置对话框中的有界滑块控件。滑动块有四个离散的位置，当你从左到右拖动滑块时，它下面的图例从“640×480 像素”变为“800×600 像素”，从“1024×768 像素”变为“1280×1024”。它们为什么不使用组合框？你更喜欢哪种？

如果程序需要一个必须保持在指定界限内的数值，呈现给用户的控件应该表明那些限制，并且阻止用户输入界限以外的值。滚动条控件就是这样。尽管滚动条有着明显的缺陷，但在一个方面却是典范：它允许用户用类推的方法输入数量信息。滚动条允许用户用相对值指定数值，而不是直接键入一个数字。也就是说，用户移动滑块，通过相对位置来指示程序内的比例值。它在输入精确数值方面作用较小，尽管许多程序也会这样用。微调控制项这一类的控件在输入具体数值方面更好。

微调控制项

微调控制项（spinners）是常见的键盘或鼠标数字输入控件。如图 26-9 所示，微调控制项包含一个小的编辑字段，附有两个半高按钮。微调控制项使得有界控件和无界控件之间的差别非常模糊。

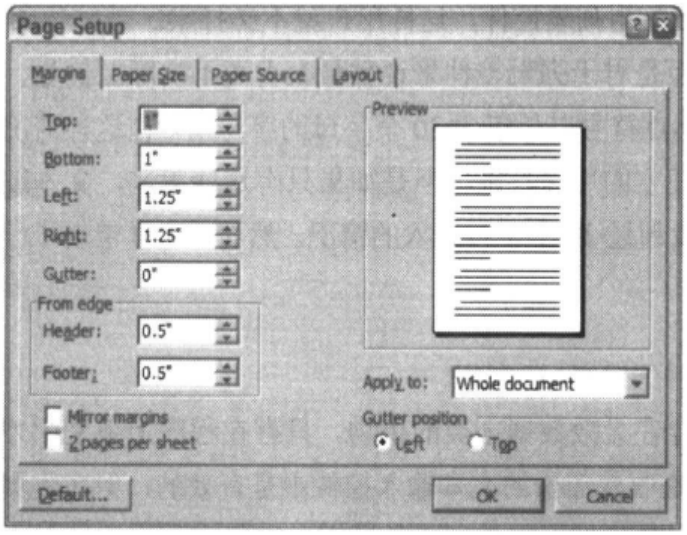


图 26-9 来自 MS Word 的页面设置对话框使用了大量的微调控制项。在左侧的对话框中，你可以看到 7 个这样的新控件，它越来越流行。通过单击小箭头按钮，指定的数值就会一小步一小步地增加或降低。如果用户希望一次做大幅度的改变或输入一个精确设置，可以使用编辑字段直接输入文本。控件的箭头按钮部分有界，而编辑字段则没有。它是有界控件吗？

微调控制项混淆了有界控件和无界控件之间的区别。使用两个小箭头按钮之一可以使用户以离散的小步改变编辑窗口中的数值。每一步都是有界的——数值不会超过程序设置的最上限，也不会低于程序设定的最下限。如果用户希望一次做大幅度的改变，或输入一个精确设定，他可以单击编辑窗口部分直接输入，就像在任何其他编辑字段中输入文本一样。遗憾的是，这个控件的编辑窗口部分是无界的，可以让用户自由输入界限以外的值，甚至是莫名其妙的垃圾。在图中的页面设置对话框中，如果用户输入一个无效的值，程序就会像其他粗鲁程序那样，产生一个错误信息框解释上限和下限（有时），并且还需要用户单击 OK 按钮才能继续。

总体来说，微调控制项是一个很好的习惯用法，可以在多数有界输入中取代普通的编辑字段。在第 7 部分，我们将讨论改善控件错误处理的方法。

无界输入：文本编辑控件

文本编辑控件（text edit controls）是最主要的无界输入控件。这种简单的控件允许用户键入任何文本值，包括文字与数字。编辑字段通常是用户只能输入一两个单词数据的小区域，但它们也可以是非常复杂的文本编辑器。使用鼠标或键盘，用户能够使用标准的连续选择工具编辑里面的文本（正如在第 22 章中所讨论的）。

文本编辑控件常用于数据库应用程序中的数据输入字段（现在常用在连接到数据库的网站中），作为对话框的选项输入字段，或者用于组合框的输入字段。在这些角色中，它们常用来完成有界输入控件的工作。但是如果所需的值是有限的，就不应该使用文本编辑控件。如果可接受的值是数值，应该使用如滑动块或调节器这样的有界输入控件。

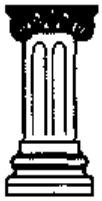
如果可接受值列表由文本行组成，应该使用列表控件，这样用户就不必打字。

有时可接受值的集合是有限的，但是对于列表控件来说数量太大而不实际。例如，一个程序可能需要除了空白、标签和标点符号以外包含 30 个字母的字符串，在这种情况下，文本编辑控件可能无法避免，即使它的用途有限。但是如果只有这些限制，文本编辑控件可以设计成不接受非字母字符及超过 30 个字符输入的情况。然而，这样带来了一个涉及确认（validation）的交互问题。

【确认】

从无界控件的角度来说，确实不存在无效数据之类的事物。只有在程序上下文中才能判断数据是否无效。例如，“1995”在收集年份的文本输入控件中是有效的，但在收集月份的控件中则是无效的。实质上，无界控件不能识别无效数据——只有程序能够做出有效的实际决定。从程序的观点来说，有界输入控件只处理有效输入。因此根据定义，对于程序来说无界输入控件能够返回无效输入。

一个用于收集有界数据的无界控件必须服务于两个目的：控件必须愉快地接受用户键入的任何数据。接着如果程序判断输入无效，控件将被迫告诉用户坏消息。让无界控件接受有界输入是用户对计算机不满的最大原因之一。



公理：为有界输入使用有界控件。

如果数据有界——但不是很严格的有界——程序必须让用户输入数据，以后再剔除无效数据。尽管这样有一些缓和步骤，但终究不是解决问题的好方法。除非……

有一种方法可以解决这个问题：程序继续，接受用户输入的任何数据。换句话说，排除对半有界数据（semibounded data）的处理。要么在有界控件中强制输入正确数据，要么在无界控件中接受所有用户输入的数据。多数程序拒绝这种解决方法。例如，他们认为程序不能够接受在社会安全号字段（social security number）中输入 asdf;lkj。但是，在某些情况下这也许是有意义的。在第 33 章我们会详细讨论这种想法。

程序员处理这种尴尬局面的方法是创建一个**确认控件**（validation control），或者带有内建编辑功能的无界文本输入控件。许多数据输入类型都是常识，包括日期、电话号码、邮递区号和社会安全号等格式，可以获得商业的专门文本编辑控件；你可以购买文本输入控件变体，例如，它们只允许数字、文字或者电话号码，或者拒绝空白和标签。

虽然确认控件是非常广泛的习惯用法，但它很糟糕，因为对于一个给定控件来说，

用户必须依靠更广泛的应用程序场景来确定有效值是什么。尽管从战术上来说，这些控件经常需要；所以我们暂时忽略一些更大的问题，而考虑实际的方法来改进它。成功设计确认控件的关键是为用户提供充分的反馈。一个只是拒绝的输入控件很粗鲁，它肯定会让用户感到气愤和愤慨。

根据第 19 章中的公理：在视觉上区分不同行为的元素，一个基本的改进是使确认控件在视觉上与非确认控件区分开来，无论是文本编辑字段使用的字体，不同的边界颜色，甚至是字段本身不同的背景颜色。

然而改进确认控件的主要方法是为用户提供丰富的状态反馈。正如我们今天所知道的，遗憾的是文本编辑控件实际上没有提供任何种类的反馈内部支持。设计师必须详细地指定这种反馈机制，程序员也可能需要用定制控件来实现。

【主动确认和被动确认】

一些控件在用户输入时拒绝接受按键。在输入过程中控件主动拒绝按键，这就是**主动确认**（active validation）的例子。例如，一个纯文本输入控件可能只接受字母字符，而拒绝数字输入。有些控件除了 0 到 9 几个数字以外拒绝任何其他形式的输入。还有些控件实时地拒绝空白键、标志符、破折号和其他标点符号。一些变体可以更智能，比如基于实时计算拒绝某些数字，除非它们通过了校验和这一算法。

当一个主动确认控件拒绝键入，它必须向用户确认已经这样做了，而且，它还应该提醒用户拒绝的原因。如果提供了解释，用户会不认为拒绝是任意的，这样也有助于用户提供程序所需的信息。

用户希望能够随心所欲地输入数据；这是键盘的特性。如果控件根据它们的值打算拒绝某些键入，它必须向用户做出明确的表示。

有时可能直到用户完成输入程序才能确认数据的范围，而不是在每次键入数据时。只有当控件失去输入焦点时，编辑步骤才发生。也就是说，当用户已经处理完一个字段而移到下一个字段时。如果用户关闭了对话框——或者控件不在对话框中时，调用了另一功能（例如在网页中选择提交）的情况下，编辑步骤也必须发生。如果控件只有等到用户完成数据输入才编辑它的值，这就是被动确认。

例如，控件可以一直等待直到名字输入完毕，再向数据库询问它是否是已经存在的条目。每个单词本身是有效的，但可能总体检验通不过。程序可以努力在每个单词输入时确认名字，但这可能会给网络和服务器添加额外的工作负担。另外，尽管在某个特定时刻，程序会知道名字是否有效，但在名字无效的情况下，用户仍然会继续输入。

处理这种情况的方法是在输入的同时使用递减计时器（countdown timer），每次击键重新计时。如果递减计时器变为零，就开始确认。计时器应设置为约每 400 毫秒一周期，

尽管你也许希望通过用户测试更精确的数字。这样的设置结果是只要用户击键速度快于400毫秒，系统会响应很快。如果用户暂停超过400毫秒，程序理所当然地认为用户已经停止思考（以CPU的速度来衡量，已经过了数月），于是继续完成对当前输入的分析。

为了提供丰富的视觉反馈，可以改变输入字段的颜色，来反映对输入数据有效性的判断。输入字段可以显示桃红色的阴影，直到程序判断数据有效，它再变成白色或绿色。

【暗示框】

对确认控件，另一个好的解决方法是**暗示框**（clue boxes）。这种小型弹出窗口的外观和行为都与工具提示相似（但其背景颜色可以与工具提示区分开来）。它的功能是解释主动确认控件或被动确认控件可接受数据的范围。另外，工具提示只有光标在控件上停留了一会才出现，而暗示框只要控件检测出无效字符就会出现（但像工具提示一样，如果光标在输入字段停留超过一秒以上，它也可以在单侧显示）。如果用户向纯数字字段输入一个非数字字符，程序会在犯规输入的位置附近显示一个暗示框，但不会遮盖所输入的数据。比如说它显示0～9，简短，简洁，而且非常有效。暗示框对被动确认也很有效，如图26-10所示。

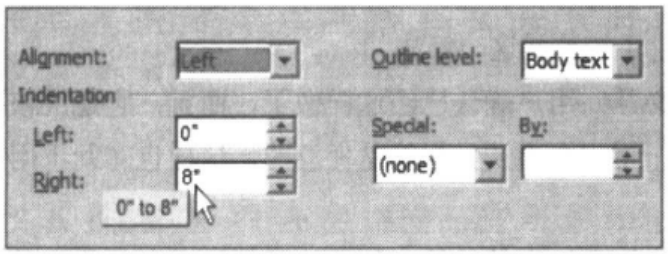


图26-10　工具提示习惯用法如此有效，以致它可以轻易地扩展到其他用途。除了黄色工具提示为图标按钮提供经过标签（flyover label），我们还可以用桃红色工具提示为无界编辑字段提供经过暗示（flyover hint）。这些暗示框也可以消除错误信息框。在本例中，如果用户的输入低于允许值，程序会用允许的最小值代替输入值，并用一个暗示框非模态地解释替换的原因。用户可以输入一个新的值或者接受最小值，而无须被错误对话框打断。

【数据出界的处理】

通常编辑字段用于输入程序所需的数值，如字号磅数。用户可以随心所欲地输入5到500之间的任意数，输入字段将接受输入，并且将该值返回所属程序。如果用户输入的是垃圾，控件必须做出某种决定。例如，在微软Word中输入asdf作为字号磅数，程序会发起一个错误信息框通知你：这是一个无效数值。然后又恢复到先前的字号磅数。错误对话框相当愚蠢，但对无意义的输入做出拒绝是应该的。但是如果你键入的是一个

有效值，9 的英文，又会怎么样呢？程序也会用同样简略的错误信息框拒绝它。如果相反，将控件编程实现为数字输入控件，可能会好一些。如果程序能将 9 的英文转换为 9，这当然不会困扰用户，但它说 9 的英文不是有效值，当然是错误的。毫无疑问它是有效的，程序已经陷入混乱之中。

除非是其他工具，简单地拒绝输入数据比用一个错误信息框拒绝更好。例如，如果被动确认控件只能接受 5 到 25 之间的数字，用户输入 50 ，控件将其改成 25 然后继续运行。如果用户输入 2，控件将其改成 5 然后继续运行。如果用户输入 asdf，控件将恢复到先前的有效值而继续运行。

【单位和度量】

如果文本编辑控件能够识别适当的单位，那就太好了。例如，如果程序要求尺寸，用户输入 5i 或 5in 或 5 英寸，控件不仅报告结果为 5，还能报告单位是英寸。如果输入 5mm，控件应该报告 5 毫米。Sketch Up 是 Windows 和 Mac 平台上一流的建筑绘图应用程序，它支持这种类型的反馈。

比如说这个字段需要一个栏宽（column width）。用户可以输入数字，也可以输入数字加如上所述度量系统的一个单位。用户也可以允许输入**默认**（default）这个单词，程序将设置默认的栏宽。用户也可以输入**最适合**（best fit）这一单词，程序会度量栏中的所有条目，根据环境选择最适合的栏宽。但这种特定情况下也存在一个问题，因为默认和最适合这些单词存在于用户的头脑中，而不是在程序中。但这很容易解决，我们要做的就是通过组合框提供相同的功能。用户可以通过下拉框找到标准宽度与单词（默认和最适合）。微软在 Word 中使用了这种方法，如图 26-11 所示。

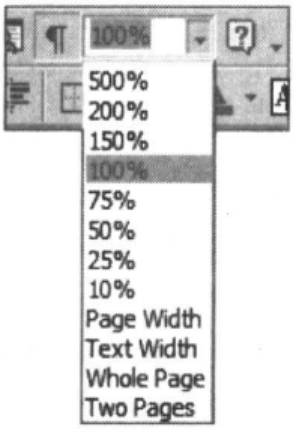


图 26-11 下拉组合框为有界输入字段提供了极好的工具。因为它包含输入值而不是数字。用户无须记住或输入例如页面宽度和整个页面这样的单词，因为它们可以在下拉列表中选择。程序将这些单词解释为合适的数字，每个人都满意了。

用户可以下拉组合框，看到如页面宽度和整个页面这样的项目，选择一个恰当的项。使用这种习惯用法，在可见和可选择的情况下，用户头脑中的信息已经转移

到程序中。

【多信息文本控件】

随着**多信息文本控件**（rich text control）的出现，简单的编辑控件可以承担成熟文字处理器的额外工作了。当编辑字段实现为多信息文本控件时，设计师清楚地意识到他应该向用户展示选项范围，这一点非常重要。对于只希望输入一个单词或数字的简单输入字段，激活整个段落格式化子系统是不恰当的。

对于输入结构化的数据到严格结构化的数据库来说，多信息文本控件并不是很有用。但是，对于编写电子邮件或记笔记这样的任务——换言之，无界文本的逐字输入非常方便。

【插入和改写输入模态】

在大多数文本编辑器中，存在一个用户可设置的选项，在插入模态和改写模态之间切换（toggling）。在插入模态下，输入新文本后，插入点后的文本保留；而在改写模态下，插入点后的文本在用户输入时丢失。这两种模态在某些文字处理器中，如 FORTRAN，无所不在，长盛不衰。插入和改写是导致界面行为显著改变的模态，直到用户完成交互后才能看到，除了相当模糊的击键方式外，也没有清晰的方式进入或退出这些模态（至少在 Windows 中是这样的）。

现在，很难想像在现代图形用户界面文字处理器中使用改写模态，但毫无疑问在此之外有这样的需求。但是对于单行编辑字段来说，添加超出简单插入模态的输入和编辑控件是愚蠢的——潜在的问题会远远大于好处。当然如果你在设计一个文字处理器，那又另当别论。

【使用文本编辑控件输出：一个坏主意】

文本编辑控件，因它熟悉的系统字体和视觉上清晰可见的空白框，给人以可以输入数据的强烈印象。但软件开发人员常将文本编辑控件用做只读输出字段（read-only output）。编辑控件当然可以作为输出字段，但把这种控件用于输出只是欺骗你的用户，用户是不会高兴的。如果你要输出文本数据，使用文本显示控件（text display control），而不要使用文本编辑控件。例如，如果你想显示磁盘剩余空间，不要使用文本编辑字段，因为用户可能会想如果输入一个更大的数值，也许就可以获得更多剩余空间。至少这是控件的肢体语言所告诉他的。

如果你打算输出可编辑的信息，继续用完全可编辑的文本控件输出，那么请在内部完善它，使它表里如一地工作，否则就坚持用下一节描述的显示控件。

设计技巧：仅供输出的文本用非编辑控件（显示控件）显示。

显示控件

显示控件（display control）用于显示和管理屏幕上信息的视觉显示方式。典型的例子包括滚动条（scrollbar）和屏幕分隔器（screen splitters）。管理对象在屏幕上的视觉显示方式的控件及静态显示只读信息的控件都属于这一类。它们包括标页面计数器（paginator）、标尺（ruler）、导向图（guidelines）、网格（grid）、分组框控制项（group box），以及那些称为 dip 和 bump 的三维线条。与其详尽讨论所有这些控件，在这里我们不如重点讨论几个问题较多的控件。

文本控件

最简单的显示控件可能是**文本控件**（text control），它在屏幕的某个位置显示信息。它的管理工作非常普通，只用来做其他控件的标签或者输出一些不能由用户或不应该由用户改变的数据。

文本控件存在惟一的问题是它们经常用在应该使用编辑控件的场合（或者相反）。多数存储在计算机中的信息都可以由用户更改。为什么不允许用户在软件显示的地方改变它呢？为什么输入值的机制与输出值的机制应该不同呢？在许多情况下，程序区分这两种功能是没有意义的。在几乎所有的情况下，程序显示的值可以改变，它应该位于可编辑字段中，用户单击后可以直接更改它。特殊的编辑模态几乎都是附加工作。

多年来，Adobe Photoshop 为了在图像中创建格式化文本时坚持打开一个对话框，结果用户无法看到文本在图像中确切的模样，于是为了得到正确的图像，用户不得不一次又一次地重复这个过程。最后，Adobe 解决了这个问题，让用户以所见即所得的方式直接在图像层编辑格式化文本，它本来就应该这样。

可恨的滚动条

滚动条是一种令人沮丧的控件，它充满困难，难于操作，而且浪费像素。毫无疑问，

滚动条遭到了滥用，而且缺少检查。作为显示控件，当它用于窗口内容和文本的导航器——一个显示控件，是合适的。尽管在很多例子中它用得不合适，只是因为设计师和程序员没有找到更好的办法。这是软件设计方面的一个糟糕理由。

除了它几近统一的可获得性外，滚动条的惟一好处是可以呈比例地显示数值。滚动条上的**滑块**（thumb）是一个可以拖动的小框，可以指示当前位置而且通常显示出可滚动区域的比例。

滚动条可以大致显示比例。例如，用户可以看到一个滚动条的滑块位于两个端点等距离的位置，代表了大约 50%的数值。实际上滚动条不传递有关终端数值的信息这种实际情况很大程度地损害了它作为游离值（sliding value）选择器的作用，但尽管如此，滚动条的比例显示仍然是一个极好的视觉反馈，尽管在实现方面存在一些瑕疵。

滚动条的另一个缺点是它向用户传递的信息太少。它应该慷慨地告诉我们它所管理的信息。当前最好的滚动条用适当大小的滑块来显示当前可见文档的百分比。但滚动条还可以告诉我们：

- 总共有多少页。
- 当我们滚动滑块时，显示页数（记录数、图形）。
- 当我们滚动滑块时，显示每一页的第一个句子（或项目）。

另外，滚动条的功能太少。它管理着文档中的大多数导航；应该为我们提供功能强大的工具，让我们快速而容易地去我们想去的地方。它可以：

- 根据页数/章/节/关键词为我们提供向前跳读的按钮。
- 为我们提供跳到文档开始和末尾的按钮。
- 给我们设置可以快速返回的书签工具。

微软 Word 的最新版本用到的滚动条展示了这些功能中的一大部分。滚动条还要求使用鼠标的高度精确性。在文档中单纯向上或单纯向下滚屏要比上下滚屏简单。你必须非常小心地定位光标，注意滚动的数据。一些滚动条在两端重复设置上下箭头；这样对于可能超出大部分屏幕的文本视图是有帮助的；但对于小一些的窗口，这种控件的重复可能有损害，徒然增加了屏幕的混乱。

滚动条的最后一个烦人之处在于：在最大化窗口中，滚动条在屏幕的最右边，所以你期望将鼠标移动到最右边，单击拖动时，会有些事情发生。但屏幕最右侧的两个像素不属于滚动条，尽管它看上去好像属于滚动条。这意味着你必须使用你的精确运动控制来确保你在滚动之前没有超出到屏幕的边界，作者就做不到。

滑动块和刻度盘

虽然**滑动块**（sliders）和刻度盘主要以有界输入控件的形式使用，它们有时也会误用做更改数据显示的控件。使用滑动块最重要且需要谨记的是：它们是有界的非比例控件。因为滚动条可以轻松地指示滚动数据的数量，而滑动块不能，所以与滑动块相比，对于大多数功能，滚动条能更好地完成在显示中移动数据的工作。

刻度盘（dials）是直接从机械时代旋转调节器的隐喻中借用过来的习惯说法。它们有非常高的空间效率，但用鼠标操作起来极端困难。因为它们包含圆形移动。它们不适合作为有界控件，更不适合作为无界控件，甚至屏幕上刻度盘最好的实现也只是为用户做一些零活。因此要尽量避免使用这种习惯用法。

拇指轮

拇指轮（thumbwheels）是一种刻度盘的变体，但它比刻度盘更容易使用。屏幕上的拇指轮看上去就像鼠标上的滚动轮，行为也非常相似。它们普遍用在某些三维程序中。因为它们是一种紧凑的无界控件，适合平移和缩放。和滚动条不同，它不需要提供任何比例反馈，因为控件范围是无限的。将这种控件映射为某个方向的无限制移动（像缩放），或者在数据的移动中依赖它自身才能返回，才是有意义的。

分隔器

分隔器（splitters）是将独占应用程序分为多个相关窗格的有用工具，每个窗格中的信息都可以浏览、操作或者变换。可移动的分隔器应该借助光标暗示显示它的受范性。尽管将所有分隔器都设置为可移动的会很容易而且又吸引人，但你应该仔细琢磨哪些是能设置成可移动的。一般来说，分隔器的移动方式不能使得窗格中的内容不可用。在窗格需要折叠的情况下，抽屉可能是更好的习惯用法。

抽屉和拉动杆

抽屉（drawer）是在独占应用程序中可以用一次动作打开或者关闭的窗格。如果打开

的抽屉数量用户可以配置，它们可以和分隔器联合使用。抽屉常常通过单击抽屉临近的控件打开。这个控件需要在所有时候都可见，应该是行为相似的锁定按钮/图标按钮或者**拉动杆**（lever），能够旋转指示打开或关闭两种状态。Microsoft Internet Explore 利用锁定图标按钮控制历史抽屉和其他抽屉，而 Mac 上的 IE 使用抽屉边界扩展的垂直方向标签来完成这类工作。标签比较笨拙，占用了浏览器的空间，但它们与抽屉相结合要比在 Windows 程序中与图标按钮结合要好。

抽屉用于存放不常用的控件和功能，但当它们投入使用后，就会和程序的主要工作区联合起来。抽屉有一个好处，就是不像对话框那样遮住主要工作区。属性细节、可搜索的对象列表或组件和历史都可以放入抽屉。

尽管在第 2 部分中讨论的宏观原理非常有助于创建令用户满意和满足的产品，但要谨记“细节就是魔鬼”，即使产品整体的概念是优秀的，令人沮丧的控件也会导致经常发生的低级烦恼。要注重细节，确保控件有适当的行为。

27

菜单：教学向量

带有下拉菜单和对话框的现代图形用户界面作为主流设计习惯用法的时间并不长——仅仅从 1984 年，Macintosh 开始。但现在它已经无所不在，以致人们很容易认为它理所当然。现在值得回顾一下，了解现代对话框和菜单界面的发展历程，来更好地理解菜单的作用和潜在的缺陷。

命令行界面

如果想要和 20 世纪 70 年代 IBM 的大型机对话，你必须用打孔机对一组计算机卡片手工打孔，用一种晦涩的称为 JCL（工作控制语言）的语言告诉计算机如何阅读你的程序，用一个噪音很大的机械卡片阅读器把这组卡片提交给系统。每一行工作控制语言或者程序都必须打孔到一张单独的卡片上。即使是第一台微型计算机，体积小、速度慢，又笨拙，运行着原始的称为 CP/M 的操作系统，它也有着比那些冷藏在玻璃屋里的暴龙[1]好得多的对话风格。你可以在标准键盘上敲入命令，直接与运行 CP/M 的微型机进行交互。

---

[1] **译者注** 指 IBM 的巨型机，译者注

这真是一个奇迹！程序在计算机屏幕上给出一个提示符，就像这样：

A>

然后你可以以命令的方式键入程序的名字，那些程序已经以文件的方式存储在系统中，于是 CP/M 运行它。我们称它为命令行界面，它被公认为人机交互历史上的一次巨大的进步。

惟一的问题是你必须记住命令行的每一个字母。对于熟练用户（那时大多数是程序员）来说，命令行功能非常强大而且高效，因为它提供了最快最有效的途径来完成人们所希望的任务。如果像传统打字员一样在“键盘”上敲击“copy a:*.* b:”命令，磁盘就会复制。如今，如果你有这些知识，对于许多操作来说，命令行仍然要比鼠标快得多。

命令行界面真的把人们从枯燥无味的机器中解放出来了，而随着软件变得越来越强大，越来越复杂，命令行界面附加给用户的记忆任务也变得越来越繁重，因此不得不让位于更好的方式。

顺序层次关系菜单

最后，在 20 世纪 70 年代末期，一些非常聪明的程序员产生了向用户提供选项列表的想法。你可以阅读列表从中选择一项，就像在饭馆里看菜单点菜一样。它的名称也从中而来，于是顺序层次关系菜单时代开始了。

顺序层次关系菜单不需要用户记忆命令行界面所需要的许多命令和参数细节。只需要在屏幕上进行选择，无须记得很清楚。这又是一个奇迹！大约在 1979 年，对程序的评价在很大程度上取决于是否提供菜单。那些坚持命令行的软件开发商最后不得不妥协，并支持更现代的界面范例。

尽管这种界面范例在那时称为基于菜单，而我们称这些菜单为顺序层次关系菜单，与今天流行的菜单区分开。旧式的图形用户界面菜单完全是层次关系的。当你在一个菜单中做出选择后，该菜单就会被另一个菜单取代，再进行选择，然后又是一个，反复深入形成一棵巨大的命令树。

因为屏幕一次只能显示一个菜单，另外那时的软件仍然严重地受到大型机批处理风格的影响，所以层次关系菜单范例在行为上是顺序的。例如，用户可以在高级层次菜单上选择一项主功能，如：

- 输入事务。
- 关闭月份工作簿。
- 打印收入。

- 打印收支平衡表。
- 退出。

在用户选择一种功能后，比如说 1.输入事务，会弹出另一个菜单，该菜单从属于先前的选择，如：

- 输入发货单。
- 输入付款。
- 输入发货单修正数据。
- 输入付款修正数据。
- 退出。

用户将从列表中选择。最可能的情况是：用户必须面对一堆这样的菜单，才可能真正开始他的实际工作。而退出选项只能将他带回到上一级菜单。这就意味着通过菜单树导航是一件真正烦琐的事。

一旦用户做出选择，他就被限定了——无法返回。人们总是免不了要犯错误，所以在那个时代，有一些更进取的开发人员增加了一个确认菜单。程序在接受用户的选择之前，会有另一个菜单发起询问：按 Escape 键改变选择，否则按 Enter 继续。这也带来了不可想像的痛苦，因为无论你是否犯了错误，都必须回答这个笨拙而困惑的原始问题，它可能会让你真的犯下原本希望避免的错误。

按今天的标准来判断，这种基于菜单的界面真是糟透了。它们的主要缺陷是必须深入到各级菜单，这也使它们在与用户交互时不灵活，不清晰。然而，它们比命令行要好，因为在命令行中你必须记住每一个操作数。顺序层次关系菜单减轻了用户的记忆负担，但又迫使他们在混乱的选择和选项中艰难前进（正如现在绝大多数网站所做的那样）。因此，它们也必然为更好的方式所取代。

Lotus 1-2-3 界面

用户界面技术的下一个巨大进步发生在 1979 年，那时 Lotus 公司发布了它最初的 1-2-3 电子表格程序。1-2-3 仍然受深层次关系菜单界面的控制，但 Lotus 做了改进，使它成为当时最成功的软件：可见层次关系菜单（visible hierarchical menu）。

1979 年，计算机屏幕每屏正好显示 2 000 个字符（如图 27-1 所示），排列成 25 个水平行，每行 80 个字符。1-2-3 将菜单水平排列在屏幕顶部，只占用了 25 行中的 2 行。这就意味着菜单可以与实际的电子表格程序在屏幕上共存。与在它之前的层次关系菜单不同，用户不必离开工作屏幕去看一个菜单，可以就在工作屏幕上输入一个菜单命令。这

种理念在 Web 中翻新为层级（breadcrumbs）的形式[2]：一行链接，它不仅显示用户所在站点的各级网页所处的路径，而且允许用户立刻回溯到链接的任何层次。

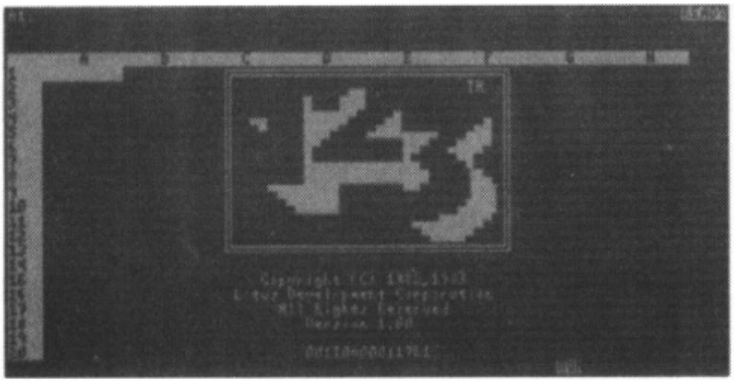


图 27-1 最初的 Lotus 1-2-3，第一次发布是在 1979 年，展示了一个崭新的菜单结构。能与程序工作屏幕共存。那时其他基于菜单的程序都迫使你离开工作屏幕进行菜单选择。和所有伟大的理念一样，它事先并没有料到，但现在看起来很明显。

Lotus 以极大的魄力创建了非凡的层次关系菜单结构。菜单树上有许多节点，几百个独立选项可供选择，每一个选项都可以通过查看屏幕最顶上的行，然后使用 Tab 键向上或向下找到想要的选项。程序通过检查是否有斜线符号（\）来区分电子表格数据和菜单命令。如果用户键入一个斜线符号，其后的击键将看做菜单命令而不是数据。为选择菜单上的某项，你只需要阅读，然后在斜线符号后面键入它的第一个字母。然后子菜单就会取代主菜单出现在最顶行。

经常使用这种界面的用户很快意识到这种模式便于记忆，他们甚至不需要阅读菜单。他们只需要键入/-s 就可以把工作保存到磁盘，而只需要键入/-c-g-x 就可以添加一列数字。基本上，他们可以完全避开菜单的使用。记住了字母命令，用户成为超级用户，并为自己了解了那些晦涩的功能而沾沾自喜。

这种层次菜单现在看来很傻，但它阐明了一个非常强大的观念：一个好的用户界面可以使它的用户以特别的方式逐步从新手用户晋升为专家用户。Lotus 1-2-3 的超级用户可能非常熟悉二十多种功能，但同时他也可能对其他功能一无所知。如果他已经记住了特定的斜线一键序列，就可以继续直接使用该功能，否则，他可以通过阅读菜单找到那些没有记住而不常用的功能。菜单作为发现和学习界面功能工具所表现出的重要性，我们在这一章的后文中还要进一步讨论。

但 Lotus 1-2-3 的层次关系菜单极其复杂。它有太多的命令，每个命令都必须适合单一层次关系菜单结构。程序设计师追本溯源，寻找功能之间的逻辑联系，试图调整层次关系菜单中分配命令的方式。在改革成功和市场优势的狂热下，很容易忽略这些细节。

---

[2] **译者注** 当前的 Web 应用程序借鉴了 lotus1-2-3 的界面形式。

如你所料，Lotus 1-2-3 的成功，使得 Lotus1-2-3 式的界面在 20 世纪 80 年代中期市场上非常流行。始终可见的层次关系菜单应用到了大量的程序中，这个习惯用法真的是基于字符用户界面的最后一根救命稻草，正如 19 世纪 40 年代末期伟大的组装蒸汽机车走向终结一样，是对技术命运的最好诠释。说明这种技术已经命中注定，十年之内柴油机车完全淘汰了蒸汽机车，而图形用户界面淘汰 Lotus 1-2-3 风格的层次关系菜单只花了短短几年的时间。

下拉菜单和弹出菜单

许多概念和技术的同时出现使得图形用户界面成为可能：鼠标，内存映射视频（memory-mapped video），功能强大的处理器，以及弹出窗口。弹出窗口是出现在屏幕上的一个矩形，它会层叠和模糊屏幕上主要部分，直到它完成工作，那时它消失，不会改变原有屏幕的模样。弹出窗口是实现**下拉菜单**（drop-down menus）和对话框的机制。

在现代图形用户界面中，菜单出现在屏幕或窗口最顶行的菜单条（menu bar）中。用户指向并单击菜单条中的一个菜单标题，即可弹出直接属于此标题的选项列表窗口。下拉菜单的一种变体是**弹出菜单**（pop-up menus），当你单击（更常见的是右击）一个对象时，尽管它没有菜单标题，也会弹出一个菜单。

打开一个菜单后，用户通过一次单击或拖动释放做出一个选择。除了菜单不会进入更深和层次之外，这里没有什么别的特殊之处。用户在菜单上做出选择后，要么立即生效，要么会调出一个对话框。这里，菜单只有一级，层次关系扁平化了。换句话说，它最终变成了**单层分组**（monocline grouping）（在第 11 章中已经讨论过）。

可以证明，图形用户界面菜单最重要的进步是：放弃了层次关系，而改为单层分组。另外一种弹出窗口的使用方式，对话框，就是简化菜单的工具。对话框可以让软件设计师将任何菜单的所有子选项都装进一个交互容器中。利用对话框，能够大量减少菜单的层次关系，菜单树所有的细枝末节都可以放到一个对话框中，深度层次关系菜单已经成为过去。

随着图形用户界面显示分辨率的增加，菜单条上能够显示足够多的选项，把程序的所有功能组织成半打有意义的分组，每组由一个单词的菜单标题代表。每组菜单也有足够的空间包含所有的相关功能，增加更多菜单层次的需求也几乎变得多余。

（当然，质疑和非难总伴随着我们，开发者已经创建了将下拉菜单转变为层次关系菜单的方法，它们称为级联菜单，尽管它们偶尔有用，但更多的是诱使开发团队中的不坚定份子打乱他们的菜单而收效甚微。我们将在第 28 章中讨论更多的细节。）

今天的菜单：教学向量

随着现代图形用户界面的演化，两个习惯用法的发展根本性地改变了菜单在用户界面中的作用。这两个习惯用法分别是直接操作（direct manipulation）和工具条（toolbar）。直接操作习惯用法的发展从图形用户界面出现的第一天开始就是一个缓慢而持续推进的过程。相反，工具条的发展则是大约发生在1989年的一场席卷整个产业的革命。在两年之内，几乎所有出售的Windows程序都有了一个充满图标按钮的工具条，而几年以前还没有人见过它。

到同一目的地，一个陌生人可能要走很多弯路，而本地人总是选择最经济的路线。与此相似，程序的熟练用户通常总是以最直接的命令调用功能，而不会选择需要中间步骤的命令。很自然，程序中最常使用的命令是工具条上的图标按钮。菜单项也支持这些功能，但使用它们的大多数是初学者，熟练用户倾向于采用图标按钮的直接向量和直接操作。

这种在经验上的软件使用差异[3]是一个重要的特征，它影响着菜单和对话框的使用方式。菜单和对话框在日常使用中的次数越来越少，而逐步变成了新手用户和临时用户的教学工具。

相对菜单命令而言，工具条上的图标按钮和其他控件通常显得有些冗余。但图标按钮非常直接，而菜单命令相对较慢，也更沉闷。不过菜单命令也有一个巨大好处，它用文字详细说明了各项功能，在相应的对话框中也有控件和数据的详细说明。向用户教授产品功能时，这些详细说明的数据使菜单和对话框成为了最有用的交互技术：**教学向量**[4]（pedagogic vectors）（其他类型的命令向量参见第18章）

有效指导的一个必需元素就是用户能够观察和实验，而无须害怕承担后果。每个对话框中的取消按钮很好地支持了这种功能。与十年前的用户界面相反，菜单和对话框不再是正常用户完成日常功能的主要方法。许多程序员和设计师仍然没有意识到这个事实，他们仍然对菜单命令向量的目标感到困惑。它的主要角色是指导新用户，提醒那些忘记的人或提供发现（和重新发现）不常使用功能的手段。

---

[3] **译者注** 即上段文字中描述的是用菜单项的大多数是初学者，而熟练用户倾向于使用直接向量和操作。

[4] **译者注** 这里给出了为什么把菜单叫做教学向量的原因。

公理：用菜单和对话框提供教学向量。

当用户第一次看见程序，往往不太知道程序能做什么，要想获取应用程序功能和用途的初步印象，最好的方法就是通过它的菜单和对话框来了解可用的功能集合。我们这样做就好比通过贴在餐馆入口的菜单了解食物类型、菜式、餐具和价格。

了解程序功能范围是创造学习气氛的基础特征之一。否则，许多易用的程序也会因为没有简单或者轻松的方式让人了解程序功能，而将用户拒之门外。

工具条和直接操作习惯用法对于新用户来说可能太难理解，但菜单的文本特性有助于解释它的功能。与图标按钮相比，“阅读格式图库”（如图 27-2 所示）对新用户来说更具启迪性（尽管工具提示会有明显的帮助）

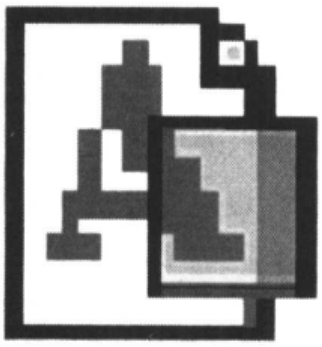


图 27-2 对新手用户来说，一个称为“格式图库”的菜单项可能比这样的图标按钮更具启迪性。但对于中间用户来说，那就完全不同了。

对程序有一点了解的生疏用户而言，下拉菜单或对话框向量的主要任务是作为工具索引，例如，当他知道有这种功能，但不记得它的位置或者名称时可以参考。它的工作原理就和餐馆里的菜单一样，帮助客人回忆一年前他曾点过的可口的咖喱鱼，而不需要记住它的确切菜名。下拉菜单可以让他重新发现自己已经忘记名字的功能。他不必在脑海里记住这些琐事，可以在需要使用它的时候，依靠菜单进行回忆。

如果菜单的主要目的是执行命令，那么就应该精炼，但是如果它的主要目的是教会我们它能做什么，如何得到它，以及可用的捷径是什么，那么精炼就不行了。我们的菜单必须解释特定的功能是什么，而不是在哪里调用。因此菜单中更详细的文本项对我们来说更适合。我们不应该说“打开…”，而应该说“打开报告…”。我们不应该说“自动排列”，而应该说“自动排列图标”。因为菜单的用户还不熟悉功能，应该尽量避免行话。

许多程序也用主窗口最底端的状态栏来显示当前选择菜单项的解释文本。这种习惯用法可以增加命令向量的教育价值——如果用户知道如何寻找它。但它的位置常常让人

注意不到。

教学向量也意味着菜单必须完整，提供程序中完整的动作选择和可用的工具。程序中的每个对话框都应该能从菜单项中访问到，浏览菜单也应该可以了解到程序功能的范围和各种工具的内涵和外延。

还可通过提供指向菜单中其他命令向量的暗示来满足另一个教学目的。将描述键盘等价物的暗示放在这里，能够教会用户可以获得更快执行命令的方法（图 28-1 说明了如何充分利用键盘等价物的暗示）。通过在菜单中正确设置这些信息，用户潜意识里记住了它。它不会强行闯入用户的意识，直到用户愿意学习为止，而这个时候用户会发现它就在手边，而且已经很熟悉了。

28

使用菜单

菜单也许是图形用户界面领域中最古老的习惯用法——被各种各样的迷信和传说环绕着、敬畏着。我们不假思索地认为传统的菜单设计是正确的，因为这么多的现有程序都证明了它的优秀。但这种信仰就像相信弹响指可以赶走老虎一样。你也许会说，这里没有老虎？看，很管用吧！这意味着按照当前结构组织的菜单无论如何也不会很快消失。本章讨论菜单存在的问题以及如何恰当地使用它们。

标准菜单

现在，几乎每个图形用户界面在最左侧的位置至少有一个“文件”菜单和一个“编辑”菜单，在右侧有一个“帮助”菜单。Windows 和 Macintosh，甚至是 Motif 风格指南都一致认为“文件”、“编辑”和“帮助”菜单是标准菜单。你也许又会认为这种事实上的跨平台标准充分地显示了这个习惯用法的正确性。错！这非常明显地说明，开发团体乐意接受平庸的设计，只有在竞争迫使他们做得更好时，才会去改变它们。“文件”菜单

这种名字是思考操作系统工作方式的实现模型[1]的产物；编辑菜单则是基于功能有限的剪贴板[2]；而帮助菜单则很少能为困惑的用户提供有帮助的知识和信息。

这些菜单惯例会诱使我们设计出一些不实用的用户界面。我们熟悉大部分程序中的菜单，但它们是组织功能的好方法吗？像“视图”、“插入”、“格式”和“选项”这样的选择听上去像工具和功能，而不是目标。为什么不以更直接的目标导向[3]的方式组织呢？

你难道听不见程序员在喊：“怎么能改变已经成为标准的事物？人们希望看到文件菜单！”答案很简单：人们可能已经习惯了痛苦和折磨，但这绝不是它们继续存在的理由。如果我们改变“文件”菜单，采用更好，更有意义的模型，用户应用起来不会有什么大问题。不考虑用户目标而武断地改变菜单项才真的是程序员们所担心的大错误。

找到合理菜单结构的关键要回溯到对用户心智模型的理解。他们是怎样思考自己正在做的事情呢？哪些术语对他们最有意义？如果你的用户是位计算机专家，你正在设计桌面应用，那么坚持公认的标准也许是明智的，至少在高层次。但是，如果你在为小朋友或为特殊的受众设计程序，那么它的结构就需要有所不同。

总之，标准菜单结构显然还有它的位置。本章的其余部分将为你在需要设计标准菜单的内部结构时提供一些实际建议。

文件菜单

在第 13 章中我们描述了一个更好的文件菜单。尽管从菜单中移走了保存命令，我们并不是完全不需要这项功能。程序应该可以自动保存，但也允许用户按需保存。保存功能不必像常见的那样，采用覆盖磁盘原始副本的方式，只需要以简单的可恢复方式保存数据，这样的恢复方式独立于应用程序，在程序中不可见。

如果我们从以文件为中心的视图变为以文档为中心，那么也应该把菜单名从“文件”变为“文档”。

微软应用程序中的最近使用（MRU）列表是一个很好的快捷方法。你可以在图 28-1 中看到它。

---

[1] **译者注** 与实现模型对应的是心智模型。

[2] **译者注** 也与系统的实现有关。

[3] **译者注** 作者在有关目标导向设计的这些章节中有更详细的阐述。

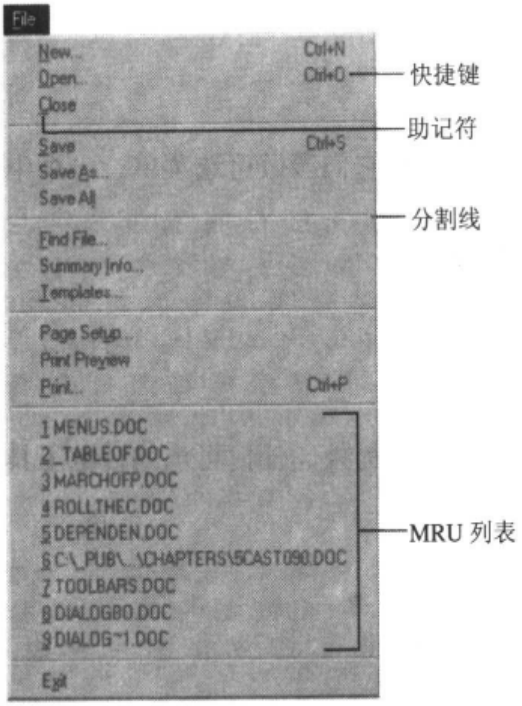


图 28-1 Microsoft Word 的文件菜单展示了它卓越的最近使用文档列表（MRU）。在第 13 章中我们明白了如何重建最前面的六个项来更好地反映用户的心智模型，而不是追随忠于技术的实现模型。

编辑菜单

编辑菜单包含用于剪切、粘贴、导入（import）和导出（export）的工具。不要把它当作一个包罗万象的容器，把其他地方都不适用的工具囊括进来。相反，最好将它们聚集到一个选项或者偏好对话框中，并且从“工具”菜单访问。

窗口菜单

窗口菜单用于对程序打开的多个窗口进行排列、显示和切换。它也可以对屏幕上同时存在的多个文档进行布局。这个菜单上不应该有其他内容。

帮助菜单

当前，帮助菜单粗糙的设计正反映了帮助系统本身的拙劣。我们会在第 35 章对它进行全面的讨论，但在这里应当提到，帮助菜单非常需要一个标名为“捷径”的项，用来解释如何摆脱对菜单的依赖。它应该指向一些更强大的习惯用法，如快捷键（accelerator）、工具条按钮和直接操作习惯用法。

可选菜单

以下各种菜单是十分常用的，但在多数风格指南中都把它们当做可选菜单。一个中度复杂的应用程序至少要用到其中一部分。

视图菜单

视图菜单可以包含影响用户观看程序数据的所有选项。另外，任何可选的视觉工具如标尺、模板或者工具栏都应该归入此处。

插入菜单

插入菜单实际上是对编辑菜单的扩充。如果你只有一两个插入项，可以考虑将它们并入编辑菜单，完全省略插入菜单。

设置菜单

如果你的应用程序中有设置菜单，就是在向用户做出承诺，无论用户何时想改变程序设置，他都可以在这里找到途径。在提供了设置菜单的情况下，就不要在其他菜单中散布设置项或对话框。这也包括打印设置，它常常错误地放在文件菜单中。

格式菜单

格式菜单是最弱的选项菜单之一，因为它几乎专门用于处理可视对象的性质，而不是功能。在更加面向对象的世界里，可视对象的特性是通过更直观的直接操作来控制的，而不是通过功能。菜单服务于它的教学目的，但如果你已经实现了一个更好的面向对象格式性质方案，就可以考虑彻底省略它。

位于文件菜单中的页面设置命令应该放在这里。(注意页面设置和打印机设置截然不同。)

工具菜单

工具菜单，有时模糊地称为选项菜单，其中包含大量强大的功能。像拼写检查和查找这样的功能就可以认为是工具。而工具菜单也应包含那些危险项目（hard-hat item）。

危险的菜单项只供真正的专家用户使用，包括各种高级设置，例如，创建一个查询，客户-服务器数据库程序有易用的直接操作习惯方法，同时程序在后台用适当的结构化查询语言（SQL）语句创建报告。非常明显，为专家用户提供的直接编辑 SQL 语句的功能是绝对的危险项。像这样的功能可能是危险的，也可能会造成混乱，所以在视觉上它们必须与那些更温和的工具分开。在过去，开发者已经将它们与其他菜单项隔离，用图标明显地标示出来只供专家用户使用。另一种可能的方法是将它们放入专家菜单，放在更温和的“工具”菜单右侧。

一些有问题的菜单习惯用法

多年来，程序员总是用一些新的行为习惯用法修饰简单的菜单。其中一些有一定作用，另一些则有害无益。这一部分讨论这些菜单习惯用法。

级联菜单

下拉菜单有一种变体，它允许一个二级菜单与另一个已经激活的下拉菜单并列出现，这种技术称为**级联菜单**（cascading menus）。它给易用性造成了一些严重的问题，但是，多数程序员几乎无法抵挡创建级联菜单的诱惑，他们似乎被级联菜单深深吸引住了。

下拉菜单为我们提供了清晰而又易于导航的单层分组（monocline grouping），而级联菜单则引入了嵌套和层次关系等复杂情况。多级级联菜单不仅为用户记忆菜单项的位置带来困难，而且还要求鼠标精确移动来平滑地实现导航。

在前一章，我们谈到现代图形用户界面如何使层次关系菜单退出历史舞台。程序员却打算让这个躺在坟墓里的习惯用法重新复活，这真是个悲剧。级联菜单只服务于一个目标：它们只是允许将更多的功能填充进一个高层次菜单，而不是为了让它们在屏幕上清晰易读。偶尔，在菜单项太多的情况下，应当将一些更生僻的项设置到第二级菜单，但这应该是“最后一招”（图 28-2 展示了一个组织恰当的一级级联菜单）。级联菜单不要用于常用功能。

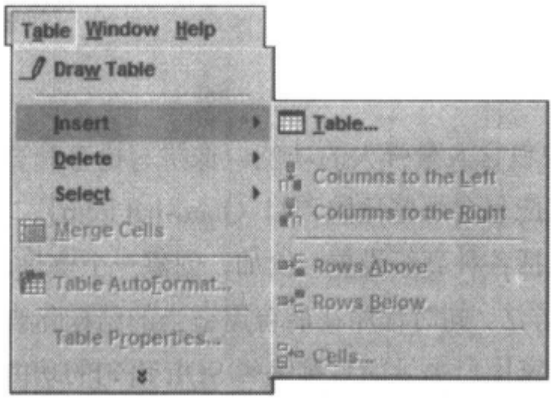


图 28-2 这是微软 Word 中的一个级联菜单。层次关系对于程序员来说有逻辑意义，但对于用户来说基本上没有。级联菜单还对使用鼠标的熟练程度有相当高的要求，这必将困扰生疏用户。微软和多数主要的软件供应商坚持不超过一级级联菜单的标准。本例另一个补救因素就是：级联的菜单项也可以从图标按钮中得到，因此有一个级联菜单项可直接获得的替换方法。此图还显示了菜单项处于失效状态的例子

在 Windows 中，很难说哪一种习惯用法不应该使用，因为应用软件的范围实在太大。但是级联菜单是一种功能不强的习惯用法，它可以在需要时使用，但不能作为优先选择。

Windows 在任务栏（TaskBar）上大量使用级联菜单。“开始”按钮上装载了太多的层次关系菜单，以致它可能不稳定，也可能会没有反应，甚至对鼠标高手也是这样。微软似乎为了避免双击而走到了一个难以置信的极端。

扩展菜单

Windows 2000 带来了一种新的习惯用法，在 Windows XP 中仍有使用：**扩展菜单**（expanding menu）。这种菜单习惯用法在最新的微软 Office 软件中无所不在，尽管它是善意的，却严重损害了下拉菜单的教学功能。微软的设计师试图预测每个菜单中最常用的项，并在初始状态下仅向用户显示那些项。为了看见其余的菜单项，用户被迫单击位于菜单底部的图标（或鼠标在图标上停留）。先前隐藏的项会分散在最初显示的菜单项中间。因为菜单的排列顺序会发生显著的改变，在 Windows 2000 中，微软将通常隐藏的项在视觉上凹进去，将它们与通常显示的项进行区分。这是多么糟糕的令人困惑的习惯用法！它与按钮的受范性响应冲突，并且给菜单增加了混乱的视觉噪音。幸运的是，这在 Windows XP 中已经有所改变（如图 28-3 所示）。尽管如此，这种习惯用法和级联菜单一样糟糕——它仅有的一点补救措施是：如果你确实选择了一个隐藏的项，那么在下次查

看菜单时，它会出现在正常显示的菜单项列表中。

微软显然是想让菜单发挥更大的作用，但在这样做的过程中也给了菜单存在的基本理由致命一击。菜单存在的基本理由就是告诉用户可以获得什么功能。微软应该意识到是工具条给用户提供最常用的项，而不是菜单。幸好，用户可以关闭扩展菜单，但不幸的是它默认状态下总是打开的。

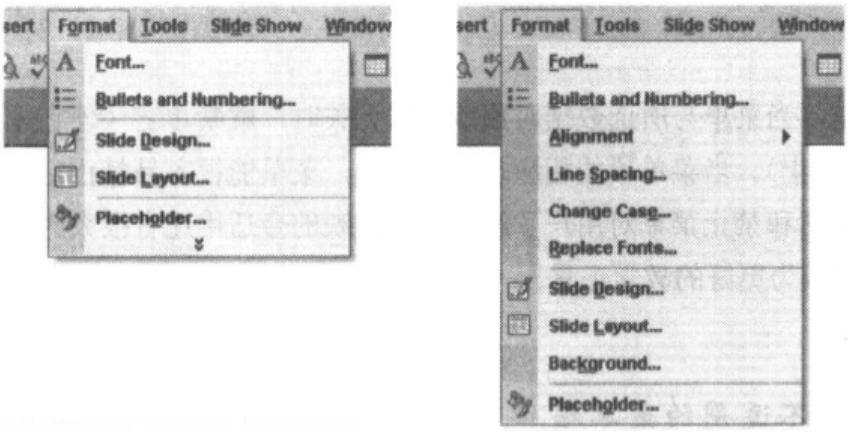


图 28-3 图中显示了 Windows XP 的 PowerPoint 中扩展菜单在扩展前后的对比情况。扩展菜单在交互范围内把所有的事情都弄糟了。它们甚至比级联菜单更烦人。将菜单中的信息隐藏起来破坏了菜单的教学功能，改变菜单项的顺序，干扰了菜单项的位置记忆。在右边的扩展菜单中，左侧稍微深一点点的灰色区域表示这些项在菜单的收缩版本中不存在。所幸的是扩展菜单可以完全关闭。

突然弹出式菜单

在 Mac 和 Windows 早期的程序中有一种菜单变体，后来因为正当的理由放弃了，就是**立即菜单**（immediate menu）或**突然弹出式菜单**（bang menu）。在程序员的行话中，惊叹号意味着突然开始进行，出于习惯，高层次的立即菜单项总是突然弹出（bang）要求执行。正如它的名字所示，菜单标题——在菜单栏上直接与其他菜单标题相邻——行为与下拉菜单的菜单项相同。突然弹出式菜单不是显示一个下拉菜单供选择，而是导致一种功能立即执行！例如，一个用来编辑源代码的突然弹出式菜单标题就是“编辑!”

它的行为是如此出乎意料，以致使人生气。突然弹出式菜单标题实际上没有教学价值，它令人迷惑和不安。工具条上的图标按钮也同样直接，却不会困扰任何人。不同之处在于工具条上的图标按钮说明了它们的直接性。因为它们是按钮，工具条是直接命令应该待的地方。

菜单项惯例及变体

以下介绍菜单项显示和组织的惯例，以及最适合使用它们的时机。

禁止菜单项

一个通行的菜单标准是，当菜单与所选数据对象或项目无关时，就禁止它（使它无效）（菜单项变灰表明已被禁止）。当菜单项的相应功能无效时，菜单能很容易禁止它，你应该充分利用这一点。激活和禁止菜单对用户了解它们所反映的合适用途有很大的帮助。这种功能可以帮助菜单成为更好的教学工具。

设计技巧：禁用不适用的菜单项。

使每个菜单项都像老师一样清楚地表明此刻它是否适合履行职责，非常重要，不要忽略这个细节。

使菜单项具有校验标记

紧挨着菜单项的校验标记（check mark），是一种用户能很快掌握的习惯用法，可能最适合用在菜单结构简单的程序中。它们常用于指示和切换程序界面特性（如打开或关闭工具条）或者与数据对象显示有关的特性（如线框显示与完全绘制图像）。后面这类属性能够非常容易地结合到一个对话框中，这个对话框能容纳一组功能更强大的工具，而与此同时又释放了菜单本身占用的有价值空间。如果是复杂的程序，菜单就需要占用足够的空间。如果是经常需要切换的特性，那么也应该可以通过工具条访问它们。

触发菜单项

你也许会认为下面的雕虫小技可以节省菜单空间。比如，一个功能有两种状态：一种称为显示状态栏；另一种称为隐藏状态栏。你可以创建单个触发菜单项来实现两种状态的切换，但触发菜单项显示的状态总是处于当前没有选中的状态。

这种技术确实节省空间，否则它可能需要两个有互斥校验标记的菜单项。但是，既然菜单的目标是教学清晰，任何妨碍用户理解的做法都不符合要求。触发菜单之所以非常令人迷惑，原因很简单：你无法分辨它是提供一个选择还是描述一个状态。比如说“显示工具条”，它的意思是工具现在已经显示，还是通过选择你可以开始显示工具条？相反，使用单个校验标记菜单项（状态栏要不已经打上标记，要不没有），就可以将意思表达清楚了。

菜单图标

文本项旁边的视觉符号使用户无须阅读文本就可以辨别它们，所以它们的功能就是加快理解。因此，在菜单中添加小图片确实可以提高用户速度。它们还为执行相同任务的控件提供有利的视觉关联。特别需要一提的是一个菜单项应该显示与相应的工具条图标按钮相同的图标。

设计技巧：在平行的命令向量中使用相同的视觉符号。

Windows 为在菜单中设置图片提供了功能强大的工具。很少有程序充分利用它们为用户提供容易的、视觉上的学习技巧。例如，微软办公系列应用程序的工具条都使用空白纸的图标表示新的文档功能。微软将相同的图标放在文件菜单中“新建”-“空白文档”级联菜单项的左侧。用户甚至无须经过思考，很快就可以把它们联系起来。Microsoft Office 应用软件在将用于教学目的的视觉符号融合到菜单中的这一方面有出色的表现。如图 28-4 所示。

图 28-4 微软 PowerPoint 的菜单项是包罗万象的大杂烩。插入菜单向我们展示分隔符（separator）、级联菜单、助记符（mnemonics）、快捷键和菜单图标。正如你所看到的，图标和那些执行同样任务的图标按钮是一致的。多么好的一种学习界面的方法！没有迂腐，没有干扰。

快捷键

快捷键是一种额外的可选方法，即从键盘启动功能。快捷键通常是利用功能键（如F9）或者前面加 Ctrl, Alt 或 Shift 键的击键方式激活某项功能。快捷键显示在下拉菜单项的右侧。在 Windows 和其他平台中它们都有一个确定的标准，但它们的实现取决于每个设计师，而且常常被遗忘了。

成功创建好的快捷键有三个窍门：

- 遵循标准。
- 用于日常使用。
- 标明如何使用它们。

有标准快捷键的地方，就使用它。特别，这里指的是“编辑”菜单中显示的标准编辑集合。用户很快就会知道键入 Ctrl+C 和 Ctrl+V 比将鼠标从起始行移动到下拉编辑菜单，选择“复制”，然后再选择“粘贴”要快，不要让用户在使用你的程序时失望，不要忘记像 Ctrl+P 代表打印和 Ctrl+S 代表保存这样的标准。

确定命令集合中哪些是日常要用的，是一件非常需要技巧的事。你必须选出可能经常使用的功能，并且确保那些菜单项已经给出快捷键。好消息是这组命令数量不大，坏消息是用户与用户之间有着很大的差别。

最好的方法是对可用的功能进行分类。将它们分为三组：那些肯定是每个人日常使用的为一组；那些肯定不是任何人日常所需的为一组；其他的为另一组。第一组必须有快捷键，第二组一定不能有快捷键。最后一组根据具体情况而定，不可避免，它会是数量最大的。你可以将这一组继续分类，依此指定最好的快捷键。如 F2，F3，F4 等指定给这组中最常用的功能。更模糊的快捷键，如 Alt+7 则应该分配给那些至少可能是部分人每天使用的命令。

不要忘记在菜单中显示快捷键。一个快捷键如果只能通过手册或在线帮助找到它，那不会给人们带来任何好处。将它放置在它应该在的地方——相应菜单项的右侧。用户开始不会注意到它，但他们最终会发现的。永久的中间用户会为发现它们感到高兴（见第 3 章），他们会有一种成就感，感觉他们已经入门。这两种感觉让你的客户备受鼓舞。

一些程序提供可自配置的快捷键。有许多实例证明这是个好主意，特别是对于专家用户来说甚至是一种必需。允许用户在独占性应用程序中定制快捷键，对于任何定制工具，必须确保有一个“返回默认值”（Return to Default）的控件。

助记符

助记符是可以为菜单和对话框的直接操作添加平行键盘命令的另一个 Windows 标准（在一些 UNIX 图形用户界面也可以看到）。

微软风格指南描述了助记符（微软现在称之为快速访问键，access key）和快捷键的各个细节，所以我们只简单地强调一下不应该忽视它们。助记符可以使用 Alt 键、箭头键、菜单项或标题中下划线字母访问。按下 Alt 键使应用程序处于助记符模式，用箭头键可以导航到合适的菜单。助记符模式打开以后，按下适当的字母键就可以执行相应的功能。助记符的主要目的在于为每个菜单命令提供一个等效的键盘操作。出于这个原因，助记符必须是完善的，尤其对于面向文本的程序。不要认为它们是键盘的便利设施，必须谨记大多数经验丰富的用户非常依赖他们的键盘，所以保持他们的忠诚，确保助记符前后一致，并且进行慎重的考虑。助记符不是可选的。

对于那些自己不用助记符的设计师（作者就不用）来说，很容易会设置不好的助记符。一个菜单中出现重复字符，或者用不合适的和难于记忆的字符（给它下个定义，称之为创建无助于记忆的助记符）。确保让开发或设计队伍的某个人对这些助记符进行测试和精简。

Windows 系统菜单

系统菜单（System Menu）是 Windows 的标准菜单，设置在所有独立窗口的左上角，它重复了窗口边框控件。在 Windows 3x 中，窗口中一个带水平条的小框启动这个菜单。在 Windows 95 和以后的版本中，它由程序图标代替。一些 UNIX 图形用户界面平台，如 Motif，在相似的位置也有相似的菜单。

这种菜单是一种历史遗迹。它服务于无用的目标。最初它是系统级窗口管理命令的汇集地，但这些命令后来都已经转移到标题栏和窗口边框的直接控件上，之后也没有再添加新的命令。

窗口菜单的存在造成了周边环境的混乱；它没有明显的目的，徒增用户烦恼。系统菜单继续存在的惟一目的是提供移动、调整大小、最大化和最小化窗口等键盘命令的等效物。只要保留了键盘命令，如果删除系统菜单，界面也不会有什么损失。不知何故，它一直保留着，甚至在 Windows XP 中也仍然保留下来了。

29

使用工具条和工具提示

相对来说，工具条是图形用户界面的最新发展之一。虽然不是 Windows 独有的特征，但它确实是从那开始普及的，而许多图形用户界面的其他习惯用法是从 Mac 开始的。工具条有许多优点，也有不足的地方，但它可以和它的伙伴——菜单互补。菜单是完整的工具集，主要用于教学，而工具条是为经常使用的命令设置的，对新手用户帮助不大。

工具条：可见的立即功能按钮

典型的工具条是图标按钮的集合，通常没有文本标签，以水平板的方式置于菜单栏的下方，或者以垂直板的形式紧贴在主窗口的一边。实质上，工具条是单行（或单列）排列的，始终可见的图形化立即菜单项。

图形界面的许多伟大思想同时有很多来源，工具条也不例外。它几乎同时出现在许多程序中，没人能够说清到底是谁首先发明了它。不过有一点是清楚的，它的优点显而易见。工具条的发明一下子解决了下拉菜单存在的问题。工具条的功能总是明白可见，用户只需单击鼠标就可以启动它们。

工具条不是菜单

人们经常认为工具条是菜单的加速版。相似之处很难免：它们都提供对程序功能的访问，经常在屏幕顶端组成一个水平行。在设计师的想像中，工具条不是和菜单平行的命令向量，而是与菜单相同的命令向量。他们认为工具条上可用的功能应该和菜单上的相同。

但工具条的存在目的实际上和菜单完全不同，它们的组成也不必相同。工具条和工具条控件的目的是为已经掌握了程序基础的用户提供快速访问常用功能的途径。工具条不向新手提供帮助，它压根就不是为此设计的。菜单才是新手用户应该求助的地方。

设计技巧：工具条为有经验的用户提供快速访问常用功能的途径。

菜单最大的优点在于它们的完备性和详细说明。用户所需的一切都可以在程序菜单中找到。当然，这种极度的丰富性也意味着它们大而笨重。为了使如此庞大的菜单不浪费太多的像素，它们不得不在大多数时候折叠起来，只在需要时才弹出。弹出动作把菜单从可见的直接命令队列中排除了。菜单是用笨拙而统一的每一小步来换取完整性和效能的。另一方面，工具条上的图标按钮是不完备的，也并不容易理解，但它们与菜单相比，不可否认地具有可见、立即，以及节省空间的优点。

工具条使菜单更限于教学目的

正如我们前面讨论过的，工具条的发明，最终允许菜单集中关注它们的教学目的。在常用功能加到工具条的图标按钮之后，下拉菜单不再是主要的启动功能习惯用法。即使对于经验很少的用户，单击图标按钮也比下拉菜单然后选择一个菜单项要快得多，容易得多——因为后者明显需要更多的技巧和时间。在工具条普及之后，菜单就落入了配角的境地。只有在那些设计拙劣或者不存在工具条的程序中，菜单还承担着启动常用功能的任务。

工具条和工具条控件

工具条产生了图标按钮，它是按钮和图标之间一次愉快的联姻。作为功能的视觉助

记符，图标按钮是优秀的。对于新手用户，它们可能难以理解，但它们不是为新手准备的。

工具条上的图标与文本

如果工具条上的图标按钮和下拉菜单中的菜单项行为相同，那为什么菜单项几乎总是伴有文本显示，而工具条上的按钮总是以小图像显示呢？这种差异存在着充分的理由，尽管我们偶尔会感到迷惑。

文本标签，比如在菜单上的文本标签，可能非常精确而清晰——它们并不总是这样，但精确和清晰是它们的基本目的。为了达到这个目的，需要用户花费时间注意它们，阅读它们。正如我们在第 19 章中所谈到的，阅读比识别图片速度更慢，难度更大。就它们的教学角色来说，菜单必须精确和清晰——一个不能精确和清楚表达的老师是差劲的老师，为了达到教学目，花费额外的时间和精力是合理的权衡。

另一方面，人类容易识别图示符号，但它们常缺乏文本的精确度和条理性。如果你没有确实了解它的意思，象形文字可能会模棱两可。然而，在你了解它的意思之后，就不会轻易忘记它；而即使你的识别快如闪电，但是每次仍然不得不阅读文本。图标在完成为常用工具提供快速访问任务的同时，必须让有经验的用户快速识别它们，这是它们最高的优先权。符号其具有的图形表意的外在形象，比文本更适合担当这种角色。

图标按钮有直接可见的按钮，以及能够快速识别的图像。它们把许多功能挤到一个很小的空间。和往常一样，图标按钮最大的优点也是它最大的不足：图标。

依靠象形文字进行交流是可行的，只要参加者预先对图标的含义达成一致。他们必须做到这一点，因为，任何一种图标就其本质而言，如果不经过学习，它的含义都是模棱两可的。许多设计师认为他们必须发明图标按钮的视觉隐喻，这样才能向新手用户传达足够的信息。这是一种唐吉诃德式的探索，不仅反映了对工具条目的的误解，而且反映了对隐喻魔法徒劳的希望。关于这一点我们在第 20 章已经谈过。

图标按钮中的图像并不需要告诉用户它的目的，而只需要一个醒目而独特的视觉特性。用户通过其他方式会知道它的目的。这并不是说设计师不需要努力实现这两个目标，但不要自欺欺人：那经常很难做到。寻找代表事物的图像要比寻找代表动作或关系的图像容易得多，代表垃圾桶、打印机或图表的图片非常容易理解，但你能用什么样的图标代表应用格式、取消、连接、合并、转换、测量或调整呢？再说，用户也许不能确定打印机图片代表什么意思。它可能表示发现打印机、改变打印机设置或者报告打印机状态。当然，在他学会这个小打印机图标代表“用能用的打印机打印当前文档的一个副本”之后，就不会再有麻烦了。

标签化图标按钮存在的问题

既有文本又有图标的标签图标按钮似乎是个好主意。这不仅在逻辑上有过争议，而且也有过先例。Macintosh 桌面上最初的图标就有文本小标题。甚至到今天，一些网络浏览器也默认这种做法。图标对快速分类有用，但除此以外，还需要文本告诉我们对象的确切用途是什么。

同时用文本和图像的问题在于像素非常宝贵。另外，工具条的功能可能是危险而混乱的，如果太容易访问，就好像将一把装满子弹的手枪放在了咖啡桌上一样。工具条是供那些知道自己在做什么的用户使用的，而菜单供剩下的人使用。

一些用户界面设计师一意孤行，为图标按钮加上标签，或者在对面，或者在下面，然后放上图像。除了偶尔的情况，屏幕空间以这种方式浪费太不值得。这些设计师试图满足两组用户的不同目标：一组希望在温和宽松的环境下学习，另一组知道清晰的边界但有时需要简短的提示。当然，必须有某种方式弥合这两类用户之间的鸿沟。接下来，我们将讨论一些可以解决这个问题，又不必占用宝贵屏幕空间的方法。

解释工具条控件

工具条最大的问题在于：尽管它们的控件速度快而且容易记住，但它们不是一开始就可理解的。如果新用户想要学习图标按钮和其他工具条控件，会怎么样呢？

气球帮助：第一次尝试

苹果公司第一个尝试发明一种工具为 Macintosh 解决这个问题，它称为**气球帮助**（balloon help）。气球帮助是令人困惑不解的办法之一，人人都可以清楚地看到它的好处，但没有人真正想使用它。气球帮助是一种**悬浮**（flyover）（有时也称为**滚过**（rollover）或**鼠标经过**（mouseover））工具，意思是当鼠标光标经过某物而用户并没有按键时会出现，类似于主动视觉暗示。

当激活气球帮助时，类似连环漫画里中的那种小说话泡就会出现鼠标指向的对象旁边。说话泡里面是一两句简短的解释对象功能的话。

气球帮助不受欢迎有着充分的理由。人们发现它的观念是错误的：它认为为了新手用户的利益让日常用户不舒服是可以接受的。气球太大了、太张扬、太琐碎，它们非常碍事。多数用户发现它们太烦人，通常会关闭。于是，当他们忘记一些对象时，不得不求助于菜单，下拉菜单，打开气球帮助，指向不知道的对象，阅读气球，然后又回到菜

单将气球关闭。哎呀，多痛苦啊！

工具提示

微软从来不会牺牲更多的经常用户的利益来换取新手用户的轻松。它发明了一种气球帮助的变体，称为**工具提示**（ToolTips）。它是我们所见过的最聪明和最有效的用户界面习惯用法之一。

首先，工具提示似乎与气球帮助相同，但仔细观察后，你就会发现稍微变换一下出现形式和行为方式，从用户的角度看，就会产生截然不同的效果。和气球帮助不同，工具提示只解释工具条控件的目的（和标志任务栏和应用程序状态栏上的项）。它们并不打算解释屏幕上的其他对象如滚动条、菜单和桌面图标。微软很清楚用户不需要非常基本的解释，也表明了它充分理解了这句话：虽然曾经是新手用户，但我们都会成长为经验丰富的日常用户。

工具提示只包含一个单词或一个非常短的词组。它们并不打算用散文式的语言解释如何使用对象；而是认为你可以从场景中获得剩下的信息。这可能是工具提示与气球帮助相比最重要的进步，这也说明微软与苹果公司设计理念的不同。苹果公司希望它的气球可以教会新手用户。而微软认为新手用户必须艰难地学会事物的工作方式，工具提示仅作为经常用户的记忆唤醒器。

通过在工具条上设置控件，普通用户有了更多的访问方式。微软则使得工具条不再局限于对菜单提供简单的支持。工具提示使工具条成为独占应用程序中发起命令时最主要的习惯用法。它也允许菜单安静地退到幕后，作为新手的命令向量，用于激活不常用的功能。图标按钮自然作为首要习惯用法，而菜单成为辅助习惯用法，使独占式应用程序更容易使用。至于暂时程序，虽然多数用户属于新手用户或生疏用户，因而对图标按钮——捷径——的需求也大大降低。

工具提示窗口非常小，它们沉着而不会遮盖屏幕的主要部分。正如你在图 29-1 中所看到的，它们出现在图标按钮的下方，在解释和作为图标按钮标签的时候不会占用专门标签所需的空间。将光标移到图标按钮上和工具提示出现之间还有一个关键的延迟时间，大约半秒钟。这就有足够的时间进行指向和选择功能，来避免工具提示出现。这种设计决定确保你在移动鼠标经过工具条去完成实际工作时，不会受到弹出的干扰——这是气球帮助早期版本经常发生的问题。那也就是说，如果你忘记了一个很少用的图标按钮用途，只需要花半秒的时间就可以找到它。

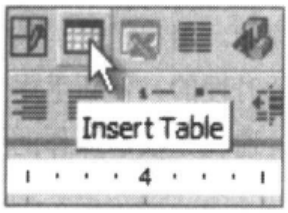


图 29-1 微软的工具提示是解决工具条所存在问题的方法。虽然工具条是为有经验的用户准备的，这些用户有时也会忘记较少使用的命令。只需光标停留一秒，弹出的小文本框就可以提醒用户图标按钮的功能。工具提示之所以成功，不仅因为它的非教学性尊重了用户，而且它也考虑到了像素的价值。工具提示打开了工具条成为独占式应用程序主要控件的大门，同时也让菜单安静地退居幕后，成为纯粹的教学和偶尔使用的命令向量

一幅小的打印机图片可能是模棱两可的，直到你看到旁边“打印”这个单词，然后你的脑海里不再困惑。如果图标按钮用于设置打印机，它会显示“设置打印机”，或只是显示“打印机”，这时指的是外围设备，而不是它的功能，场景告诉了你其余的信息，非常节约像素。

笔者既是经验丰富的日常用户，又是经常使用工具提示的用户。我们从不激活 Mac 机上的气球帮助。微软的工具提示相对气球帮助来说简直就是一种巨大的突破，然而它们实现的功能确实是一模一样的。这又证明了：“细节就是魔鬼”。

公理：在所有工具条和图标控件上使用工具提示。

没有提示的工具条迫使用户不得不详细查看菜单，通过实验学会它们的功能，或者最糟糕的是，要去阅读文档。因为工具条包含中等经验程度用户使用命令的直接版本，它不可避免地包含了一些混乱或危险。所以，如果在屏幕底端状态栏上用一行文本解释图标按钮的目的，不如在你正在查看的位置显示工具提示有效。

不要创建没有工具提示的工具条。事实上，工具提示应该用于所有图标按钮，甚至是对话框中的图标按钮。

禁用工具条控件

如果不能用于当前选择，工具条控件应该禁用。它们一定不能提供模棱两可的状态：例如，如果图标按钮禁止按下，控件本身也应该变成灰色，使禁用状态绝对明显。

一些程序干脆让禁用的工具条彻底消失，这种效果也是令人吃惊的。用户记得工具条所在位置，有了这种现象，受到一贯信任的工具条变成了易变而暂时性的，使新手用户大惊失色，甚至较有经验的用户（像作者这样的）也会困惑不解。通往无模态操作的途径不在于变得更短暂，而在于变得更牢固、持久和可靠。

工具条的演化

在人们开始意识到工具条不仅仅是菜单的快捷方式后，它的成长潜力更明显了。设计师开始意识到除了习惯以外，没有理由将工具条控件限制为图标按钮。不久，设计师开始为工具条发明新的习惯用法。随着这些新构造的出现，工具条真正使自己成为一个主要的控制工具，不同于下拉菜单——而且在许多情况下优于下拉菜单。

图标按钮之后，下一个在工具条上落户的控件是组合框，如 Word 样式、字体和字号控件。这些选择控件落户工具条是很自然的，它们和下拉菜单提供相同的功能，而且还显示了当前选择的属性：当前样式、字体和字号。这个习惯用法能让用户付出的努力更少却得到了更多的信息。

组合框加入工具条后，因为有了先例，像我们在第 26 章中讨论的，各种习惯用法相继出现。这些工具条的一部分如图 29-2 中所示。

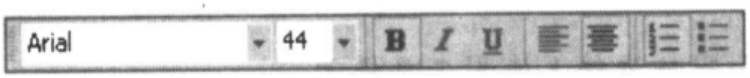


图 29-2　工具条的发展不仅导致了对工具条目的的扩展。它由命令按钮存储库演化为控件能够显示当前选择项状态的场所。这是一个更加面对对象的概念，同时它也使我们的软件功能更加强大。工具条已经成为控件革新的源泉，远远超出了我们对对话框的期望。此图显示了微软 PowerPoint XP 中一个工具条的一部分。你可以看到工具条拖动控件、组合框和锁定图标按钮，锁定图标按钮在用户释放鼠标键后将保持自己的状态（见第 26 章）。

状态指示工具条控件

这种控件变体有益于扩展工具条的用途。当工具条第一次出现的时候，它只提供快速访问常用功能。随着发展，工具条控件开始反映程序数据的状态。与只是简单将单词从无格式（plain）转变为斜体（italic）的图标按钮不同，现在的图标按钮开始通过它的状态显示当前选中的文本是否已经变为斜体。图标按钮不仅控制样式，而且表达了样式选择的状态。这是一个很大的进步，转向更加面向对象的数据表达的进步，因为系统根

据你所选择的对象进行自我调整。

工具条上的菜单

随着工具条上控件种类的增加，我们发现自己已经处于将下拉菜单加入工具条的尴尬局面。图 29-3 显示的是 Word 工具条中撤销的下拉列表。具有讽刺意味的是如此像菜单的一个习惯用法移植到了工具条。直接撤销（immediate undo）当然应该属于菜单，但是显示过去动作历史的相关下拉部分也应该属于菜单吗？至今没有明确的答案。历史列表设置在撤销图标按钮旁边的好处是易于寻找，而工具条是集中常用功能的场所。一个用户需要访问历史列表的频率有多高？最后，让我们回到像素问题。在微软实现工具条时，正是因为其消耗像素少而证明它是合适的。然而，很明显，现代工具条将老式菜单栏进一步变为了辅助命令向量。

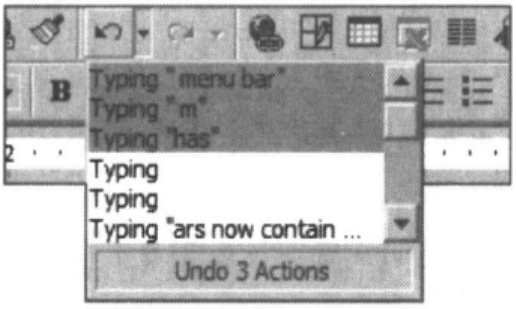


图 29-3 现在的工具条包含下拉列表。这个例子来自 Word 2000 的撤销控件。真是讽刺中的讽刺，下拉菜单已经移居到工具条中，而工具条曾经被用来驳斥老式的下拉菜单栏。

可移动工具条

在将工具条发展成为用户界面习惯用法这一方面，微软比其他任何的软件开发商都做得多。这也反映在它的产品质量上，在它的 Office 套件中，所有工具条都是可定制的。每个程序都有一个标准的**工具条组**，用户可以选择可见或不可见。如果它们是可见的，就可以被动态地放到五个任意的位置之一。它们可以附加在程序主窗口四边的任意一边，称为**停放**（docked）。在工具条图标按钮之间的任何空隙处单击鼠标，把它拖动到靠近边缘的任意位置释放，工具条就会永久地将自己粘在旁边、顶部或底部。如果你把工具条从边缘移开，它会把自己变成一个有小标题栏的悬浮工具条（floating toolbar），如图 29-4 所示。

允许用户移动工具条，也可能会使用户的工具条相互遮挡。微软通过扩展组合图标按钮或下拉菜单方便地解决了这个问题，下拉菜单只在工具条部分遮蔽时才出现，并且可以通过下拉菜单访问隐藏的菜单项，如图 29-5 所示。

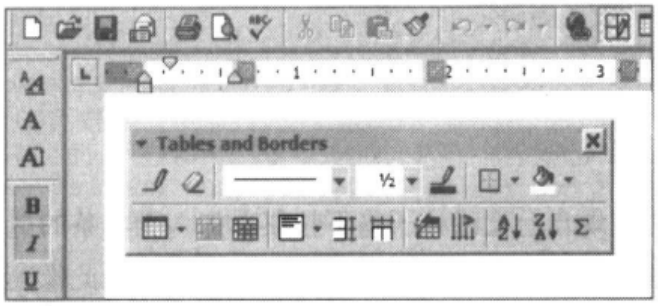


图 29-4 工具条可以水平停放（顶部），垂直停放（左侧），也可以拖动工具条成为一个悬浮调色板。像这样使用可停放的调色板几乎总是比强迫用户对不可停放的工具栏进行窗口管理要好。复杂的调色板也许不能将它们自己组织成像工具条一样的条状，但仍然应该可以停放在独占窗口的边上。

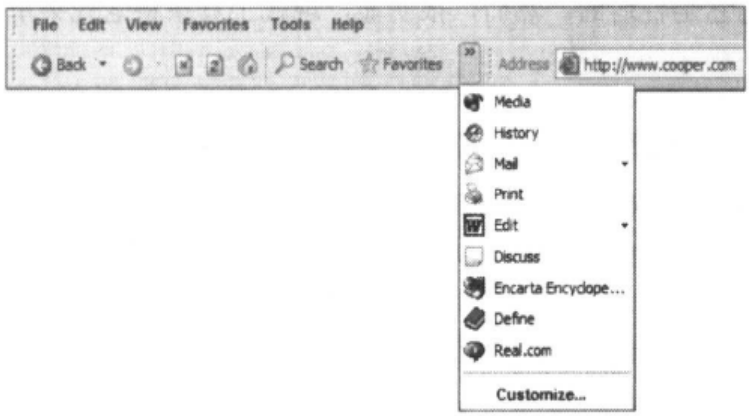


图 29-5 微软的聪明之处在于：允许用户重叠工具条，但仍然可以访问所有功能。这样提供了一种轻量级的定制，超级用户更喜欢实现完全可定制的工具条，这样他可以通过下拉菜单底部的“自定义…”来解决类似的问题。

可定制工具条

微软清楚地意识到这样的两难局面：因为工具条代表所有用户的常用功能，但对于每个用户而言，这些功能是不同的。显然，微软找到了这个问题的解决方法：程序装载典型用户最可能日常使用的控件，其余的让用户定制。但是因为加入了非日常使用的工具，这种解决方法的效果有所减弱。例如，Word 工具条的默认图标按钮套件也包括那些不常用的功能。像插入 Auto text 或插入 Excel 电子表格这样的控件更像是功能清单列表，而不是大多数用户的实际日常选择。尽管它们偶尔会有用，但多数用户不会经常使用它们。

Word 为更高级的用户提供了定制能力，让他们可以随心所欲地配置工具条。为工具条提供这种级别的可定制能力存在某种危险，因为鲁莽的用户可能会创建出一个确实无

法识别而且无用的工具条，那完全是白费力气。人们通常不会为创建一个丑陋而难于使用的东西花费多少力气，他们更可能只会进行一点个性化的改变，数月或者一年才一次。微软扩展了这个习惯用法，这样你可以创建全新而完全个性化的工具条。对于普通用户，这个特征当然是不太必要了，但 IT 管理者可以用它建立“公司”的形象。

微软的定制工具条体现了对细节的惊人关注。你可以将图标按钮向一旁拖动一英寸创建一个小的空隙，这样允许你用好的视觉分隔来对图标按钮进行分组，一些图标按钮是互斥的，所以将它们进行分组很恰当。你也可以选择图标按钮的大小，这对于显示屏分辨率的高低不等来说是一个很好的补偿。显示屏分辨率一般在 640×480 至 1600×1200 或更高一些的范围内。如果大小不能调整，固定大小的图标按钮可能会小得看不清，也可能大得令人讨厌。你还可以选择让图标灰度显示取代彩色显示。最后你还可以关闭工具提示，尽管作者想像不出为什么有人会这样做。

Windows 任务栏：特殊目的的工具条

随着 Windows 95 的发布，微软引入了任务栏（Taskbar），它是一种聪明而有效地管理运行程序和显示系统状态的方法。或许最初为了与 Macintosh OS 7 及其独一无二的菜单栏竞争，任务栏实际上更有用，更适合于多任务的操作系统（那时 Mac OS 还不是多任务的）。通过“开始”菜单，任务栏提供了比较笨重，但容易识别的程序和文件系统的进入点。在后来发布的版本中，任务栏不断演化，添加了一个用户可配置的快速启动区域（quick start area），用来启动普通应用程序和文档。任务栏可以拖动，变得更大，并包含更多的控件。在过去的版本中，任务栏可以停放到屏幕的顶部或者底部。

随着 Windows 新版本的发布，微软似乎想在任务栏中增加新的事物——这种一度有所减缓的趋势不久又有些回头。但是到目前，这些功能的总和使得任务栏成为用户管理桌面的关键工具。

在 Mac OS X 中,我们终于看到了任务栏的一个可怜表亲：Dock，它作为 Macintosh 平台的一个标准出现。不幸的是，它的实现重形式而轻实质。它最大的错误在于在代表可启动应用程序（launchable application）的图标与代表运行进程（running processes）的图标差别不明显，而在任务栏中没有这种问题。

开始菜单

如前面所述，“开始”菜单主要是进入程序、设置和文档的级联菜单结构的入口。它

有一个主要的优点隐藏在一个友好的大按钮中，就是它几乎总是可见，并且用大写字符标为“开始”。问题在于一旦单击它，特别是作为一个新用户，你会有点不知所措，不知道该去哪或下一步该干什么。开始菜单在 Windows XP 中变得更精细，尽管设计师明显希望将项目分成更好理解的几类，但总体效果仍然非常混乱。

“开始”菜单一个好的特征是可以将项目[1]拖动到它上面。但随着快速启动区域（下面我们会谈到）的出现，这种功能又有些多余。既然你可以把项目放到工具条上，为什么还把它放到已经非常拥挤的菜单上来呢？

快速启动工具条

“开始”菜单的右侧是一个可定制的区域，可以从桌面把图标拖到这个区域中。拖来的图标在这里变成启动程序或文档的图标按钮。这是一个伟大的特性，而且做得很好。如果你想移走某项，只需拖离工具条，当你释放它时，它就会变成桌面项的快捷方式，你可以保留，也可以删除。

一些程序员试图在没有征得许可的情况下，就直接在快速启动区域上添加他们的程序链接，这是一种傲慢无理的姿态，用户是不会欣赏的。你可以让他们选择是否这样做，但不要认为他们一定会希望你的程序出现在那。

窗口按钮

窗口按钮是任务栏主要的固定装置，它们代表桌面上的每一个运行窗口。单击其中一个，如果它是关闭的，那么将会打开一个窗口并且接受输入焦点（input focus）。如果单击表示打开窗口的按钮，那么窗口就会关闭（实际上是最小化窗口，程序仍在后台运行）。如果你在打开的窗口上单击，任务栏上的窗口按钮显示状态会反映这些选择，这样任务栏总是可以用来辨别当前窗口。

任务栏按钮包含应用程序的图标、程序的标题和/或显示的当前文档（微软目前似乎遵循“文档标题在前，程序标题在后，中间用破折号隔开”的标准）。只要你没有打开太多的窗口，或任务栏上没有太多的其他控件，这种方式没有问题。因为窗口按钮太多的话，就会变得很狭窄，需要用工具提示提供帮助。

遗憾的是，在 Windows XP 中，微软做了些干扰工具提示的事情：随着空间变得有限，任务栏不是压缩所有的按钮，而是根据程序将它们分组到单个按钮。工具提示只显

[1] **译者注** 这些项通常是代表程序或者文档的图标。

示应用的名字（事实上应用名字代表不止一个窗口，按钮本身也是这样）。为了从一个分组按钮中得到特定文档，用户不得不再单击按钮，然后使用菜单选择。这种新的设计有两个严重问题：首先，在某些情况下不单击窗口按钮就不能分辨哪个文档是打开的；其二，活动窗口按钮和活动窗口之间的映射失去了平衡，变成了多个窗口对应一个分组窗口按钮。谢天谢地，这种病态的行为可以在任务栏属性中关闭。

状态区

自从在 Windows 95 中引入以来，状态区（也称为系统托盘，system tray）已经成了小控件的堆积地，这些小控件可能并不应该在这里出现。状态区是为了让不同的硬件和后台进程提供状态信息而设计的，它也提供一个开启和关闭这些对象的开关。它还显示时间（在工具提示中也显示日期），这很方便，并且复制了 Mac 菜单栏长时间存在的特征。在 Windows XP 中，不活动的图标隐藏在抽屉里（用户可以打开），在它们需要注意时才出现。状态区图标通常在单击时启动一个小的配置对话框，右击时弹出一个菜单。它们应该既简单，体积又小。如果你在设计启动一个应用程序或者复杂级联菜单的状态栏控件，那么把它放到状态区可能就不合适。

一些状态栏图标（如 Windows 2000 中）展现的行为非常有趣，是苹果公司气球帮助概念的非掠过（non-flyover）变体，但是用于完全不同的目的。在 Windows 2000 和 XP 中，状态区图标偶尔会启动一个小小的说话泡，行为上像公告对话框（bulletin dialog）或者进度对话框（process dialog）（第 30 章和第 31 章对这些对话框和其他对话框进行讨论），并且有完整的终止控件[2]（termination control）（类似于 Clippy 的帮助对话框，但远没有它那样烦人）。系统模式对话框也应该像它们一样，不妨碍用户工作的方式。

任务栏上应该有更多的工具条吗

在 Windows 2000 和 XP 中，任务栏包括其他一些工具条，你右击任务栏就可以看见它们——一个让用户输入 URL 启动浏览器；一个让你添加 URL 作为链接；还有一个可以复制所有的桌面图标和文件夹，你甚至可以以任何文件夹或 URL 为基础创建自己的工具条，最后这点似乎尤其惊人。尽管这些对于一些用户来说可能有趣，但微软应该知道工具条不是所有问题的答案。这种将所有东西（除了厨房的下水道）都填充到任务栏的尝试，必然将导致前面提到的窗口按钮混乱的情况。这样做，真的值得吗？

---

[2] **译者注** 对话框的 OK 按钮就是终止控件的一种。

30

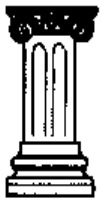
使用对话框

对话框（Dialog box 或者 dialog）不是主程序的一部分。如果说主程序是厨房，那么对话框就是橱柜，橱柜起着辅助的作用，对话框也一样。它们是辅助的参与者（actor），而不是主角。虽然它们可能会逐步推动动作向前，但它们本身不是动力引擎。

对话框暂停了正常交互

对话框叠加在父程序的主窗口上，它是系统与用户进行对话的场地，提供信息和请求输入。当用户完成了查看或者改变了显示的信息时，他可以选择接受或者拒绝这些改变，对话框随即关闭，并让用户返回主程序。

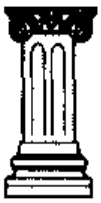
很多用户和程序员认为对话框是图形用户界面的主要习惯用法，很多应用程序也都运用对话框来提供程序的主要交互方法（我们提到的不是那些仅仅由一个对话框构成的简单程序，在这些情况下，对话框确实起到了主窗口的作用）。对于大多数应用程序来说，用户被迫在程序的主窗口和对话框之间来回往返，不可避免地让他感到疲劳和沮丧。



公理：将主要的交互放在主窗口中。

当应用程序显示一个对话框时，它将动作临时移到了主流程之外，摒弃了用户的主要关注点而产生了一个次要的问题。正确理解“对话框是正常处理的暂停”这一点是设计合适对话框的关键。程序的主要交互应该包容在程序的主窗口中，而与此相反，对话框应该仅仅用于辅助的且通常简短的交互。

如果你请宴会上的客人暂时停止喝汤，并进入食品储藏室，这样将会打破流畅的会话流和温暖的友谊，同样，对话框也会打断用户和程序之间顺利交流的和谐。对话框，不管好坏，都打断了交互，并且让用户被动地对程序做出反应，而不是驱动程序。



公理：对话框打断了流。

对话框对于那些交互主流程之外的功能或特性来说是合适的。任何可能会让人困惑、危险或者很少使用的功能放在对话框上可能都会有利。混乱行为（dislocating action）会对屏幕图像产生立即和总体的改变，这种改变在视觉上对用户干扰很大，应该将它与不熟悉的用户隔离。基于这个理由，对话框是非常适合于管理混乱行为的工具。

对话框常适合用于表达不常使用的功能和设置，可以用对话框将这些操作与更为频繁使用的功能和设置隔离开。与其他的主要控件相比，对话框通常更有空间来表现控件。例如，与工具条相比，在对话框中你可以为解释性标签（explanatory label）使用更多的空间。

对话框也非常适合集中与某个主题相关的信息，例如应用程序中一个对象的属性——例如一张发货单或者一名顾客。它也能收集与程序中某个功能相关的所有信息——例如打印报告。当以这种方式使用对话框时，它便成为了一个封装工具，使得你能够打包或者删除一些可能会对程序的正常事件流产生混乱或者危险效果的功能和设置。例如，允许对文档进行整体重新格式化的对话框就应该当做扰乱行为。对话框通过始终显示一个大而友好的撤销按钮（Cancel button），并且提供空间为有风险的控件显示更多保护性和解释性的信息，来防止这种功能的偶然调用。它也能够用缩略图显示改变的结果，向用户显示功能的潜在效果。

可从菜单中调用更多的对话框，所以在菜单和对话框之间存在血缘关系。正如第27章所讨论的，菜单提供了教学命令向量——它的主要目的是向用户教授有关程序的知识。通过扩展，教学向量也频繁地加入到了对话框中。

对话框主要向两个主体服务：熟悉程序的频繁使用用户，用它们来控制更高级或者更危险的设置；不熟悉程序范围和使用的用户，以及使用对话框学习基础知识的用户。这种双重性质意味着对话框必须是紧凑和功能强大的，快速而流利，并且在使用上清晰和具有自我解释性。这两个目标看起来相互矛盾，但实际上它们具有有益的互补性。对话框快速而强大的性质能够直接起到自我解释的效能。

对话框基础

大多数对话框拥有按钮、组合框和其他控件。尽管已经存在一些基本惯例，但是设计师通常根据他们自身的感觉来布置，而不根据任何惯例的规划。对话框可能有边框或标题栏，也可能没有。

所有的对话框都有一个所有者。通常这个所有者是应用程序——创建它的程序——但它可能是窗口系统本身。对话框在视觉上总是位于父程序的上方，尽管可能有其他程序会遮住它。

每个对话框至少有一个**终止命令**（terminating command），即激活时会让对话框关闭或者消失的控件。多数对话框会提供至少两个按钮作为终止命令："OK"和"取消"，虽然在右上角的关闭按钮（Windows 3x 和 Mac OS 9 是左上角）也是一个终止命令的习惯用法。

在技术上，对话框没有终止命令也是有可能的。一些对话框由程序单方面控制显示和消失——例如报告耗时功能的进度——他们的设计师省略了**终止命令**。正如我们会看到的那样，因为多方面的原因，这种设计很糟糕。

模态对话框

有两种类型的对话框：模态对话框和非模态。**模态对话框**（modal dialog boxes）是目前为止最常见的类型，打开一个模态对话框后，拥有它的程序不能继续进行，直到对话框关闭为止。模态对话框在它们的轨道上会停止所有的进度，单击该程序的其他任何窗口，用户都只会听到粗鲁的"嘟嘟声"表示无法反应。拥有它的程序表面所有的控件和对象在模态对话框工作过程中暂停工作。当然，在模态对话框打开的时候，用户可以激

活其他的程序，不过当用户回到原来的程序，对话框还在那等着。

总之，模态对话框是用户（和设计师）最容易理解的。模态对话框的操作非常清晰，对用户说，“现在停下手里的活，来处理我吧。结束之后，就可以回去继续你正在做的事。”模态对话框严格定义的行为意味着它很少被人误解，尽管有时被滥用。也许模态对话框太多，也许显得拙劣和愚蠢，但用户很清楚它们的目的和使用范围。就像死亡和税收，你可能不喜欢模态对话框，但你理解它们的意思。

如果一个模态对话框是面向功能（function-oriented）的，它通常针对整个程序或整个文档进行操作。如果模态对话框是面向过程（process-oriented）或面向属性的（property-oriented），它常是针对当前的选择进行操作的。在任何情况下，在你已经调用一个对话框后，不能改变选择。这是模态对话框与非模态对话框最重要的区别。

事实上，因为模态对话框只停止它们所属的程序，可以把它们更确切地称为**程序模态**（application modal）。也可能创建一种称为**系统模态**（system modal）的对话框，它能使系统中的每一个程序都停止。没有应用程序应该创建这样的对话框，它们惟一目的是用来报告出现的真正影响整个系统的灾难。

设计技巧：绝不创建系统模态对话框。

非模态对话框

另一种对话框称为非模态对话框。它们不如同胞姐妹模态对话框那么常见（尽管自 Adobe 和 Macromedia 开始，随着悬浮调色板在应用程序中广泛使用，它们也变得越来越普遍），也更不好理解。

打开一个非模态对话框后，可以不用打断父程序，无须停止进度，应用程序也不会冻结。主程序的多种工具、控件、菜单和工具条仍保持活动，可以继续调用。尽管非模态对话框的规则比模态对话框的更弱，也更混乱，但它们也有终止命令。

非模态对话框是一个更难使用和理解的怪物，主要是因为它的操作范围不确定。表现在当你调用它时，在它存在的同时，你可以重新回到主程序。这意味着在仍可以看见非模态对话框的情况下，你可以改变选择。如果对话框作用于当前选择，你可以随心所欲地不断选择、改变、选择、改变、选择和改变。例如，微软 Word 的查找和替换对话框允许你在文本中查找一个单词（它会自动选择），并进行编辑，然后再回到对话框，对话框在编辑过程中仍然保持开放状态。

在某些情况下，也可以在主窗口和非模态对话框之间拖动对象。这种特性使它们在画图类型的程序中作为工具或对象调色板时确实很有效。

非模态对话框存在的问题

大多数非模态对话框实现得很笨拙，它们的行为很不一致，令人十分困惑。它们在视觉上与模态对话框非常相似，但功能不同。非模态对话框很少建立行为规范，尤其是终止命令。微软建立了根据不同上下文改变终止按钮图例的先例，一种蹩脚的做法。

多数疑惑的产生是因为我们更熟悉模态对话框，和模态对话框不一致给对话框的使用造成了困难。看到一个对话框时，我们认为它是模态对话框，具有模态对话框的行为方式。如果它是非模态的，用户只有去尝试才能了解它的行为方式。因为非模态对话框没有清晰的原型（archetype）。

更多疑惑的产生是因为用户非常熟悉模态对话框的行为。模态对话框会在调用的瞬间为当前选择调整自己[1]，而且，模态对话框认为在它存在的过程中选择不会变化。相反，在非模态对话框存在的过程中，选择很可能发生改变。那么对话框应该怎样呢？例如，如果使用非模态对话框修改文本，而我们在主窗口选择一些非文本对象，情况会怎么样呢？对话框的小控件（gizmos）应该变灰？冻结？消失？还是应该保持原样，所有控件保持“激活”但操作时没有效果？这些选择都曾经尝试过。尽管各有优点，但究竟哪个对我们有帮助哪个有妨碍还不清楚。我们会在接下来的几页文字中进一步讨论。

非模态对话框也把我们引入了非常复杂的情况。例如，在 Word 的“编辑”菜单打开非模态查找对话框。现在再从格式菜单打开模态的字体对话框。查找对话框就消失了。这是件好事，确实如此，但模态对话框保持在不相关的非模态对话框之上很令人奇怪——查找对话框哪去了？如果你关闭字体对话框，非模态的查找对话框又出现了——整个过程中它藏在你的文档窗口之后！窗口的位置改变造成了一种逻辑感觉：在视觉上尽量让模态对话框和主窗口接近。但是，非模态对话框的消失可能会干扰没有经验的用户。简单的解决方案是完全消除非模态对话框，但那是损人不利己的事。

改进非模态对话框的两个方法

必须找到非模态对话框问题的解决方法。我们提供两种方法：第一种容易理解，是对现存困境的演化；第二种则更激进一些，是一种革命式的跳跃。你可能会怀疑，前者

[1] **译者注** 指根据当前的选择激活或者使某些按钮失效。

不如后者彻底和效率高。你也可能会正确地猜到——我们更喜欢革命式飞跃。

【演化方法】

在演化方法中，我们让非模态对话框更多地保留原有特征，但采纳了两个指导原则，并且将它们一致地应用到所有非模态对话框中。第一个原则就是我们必须在视觉上区分非模态对话框与模态对话框。

设计技巧：在视觉上区分模态对话框和非模态对话框。

如果程序员在 Windows API 中使用标准非模态对话框工具，创建的对话框在视觉上与模态对话框没有区别。我们必须打破这种习惯，设计师必须确保所有的非模态对话框有着清晰醒目的视觉差异。一种可能的方法是在对话框背景中使用独特的色调，或者提供着色的窗口边界，或者在边角插入彩色条纹，或者使用不同颜色或斜体标签使控件具有独特的视觉特征。你还可以在视觉上区分名称/标题栏，例如使它更细，或添加独特的图标。

无论选择哪种方法，你必须进行一贯的坚持。如果开发商能达成一致的标准，那就太好了，但这只是理想主义。如果每个开发商在它的公司范围内坚持非模态对话框标准，问题就会得到显著的改善。

第二个原则是我们必须采用一致、正确的终止命令（terminating command）惯例。似乎每个开发商，甚至是每个程序员，在每个不同的对话框都在采用不同的技术。这种不协调的方法没有任何存在的理由。有的对话框使用“关闭”作为终止命令，有的使用“应用”，有的用“完成”，还有的用“拒绝”，“接受”，“Yes”，甚至有的用“OK”，变化无穷。还有的干脆省略终止按钮，直接依赖标题栏的关闭框。终止非模态对话框的习惯用法应该简单、鲜明和一致，各个程序之间如果不是完全相同，也应该非常相似。

设计技巧：为非模态对话框提供一致的终止命令。

一个非常讨厌的想法就是根据用户在非模态对话框中采取的动作将终止命令的图例从“取消”变为“应用”，或者从“取消”变为“关闭”。这种动态变化，至少会给用户

带来困惑而难于理解；更严重的是，可能会导致用户因为害怕而无法理解。这些图例不应该变化。如果用户没有选择有效的选项而单击了 OK 键，对话框应该认为用户的意思是“什么也不做，关闭对话框”，因为用户确实也是这样做的。模态对话框的“取消”（cancel）按钮为我们提供了直接取消的能力，非模态对话框却经常不提供这种直接习惯用法——我们必须求助于“撤销”[2]（undo）——所以用改变图例提醒用户只会带来混乱。

设计技巧：绝不要动态改变终止按钮的标签。

模态对话框的认知力量在于严格一致的“OK”和“Cancel”按钮。在模态对话框中，“OK”按钮意味着“接受输入，关闭对话框”。问题在于非模态对话框中没有等价的概念。因为非模态对话框里的控件总是处于活动状态，相同的概念云山雾罩。用户不会像在模态对话框中那样因为预料到终止的执行命令，而适当地做一些改变。在模态对话框中，“取消”按钮意味着“放弃输入，关闭对话框”。但在非模态对话框中改变是立即生效的——一单击活动按钮即发生——没有“取消所有动作”的概念。在许多选择中可能有许多独立的动作，在这里，合适的习惯用法是撤销功能，它位于工具条或编辑菜单中，适用于活动应用程序范围内所有的非模态对话框。它在逻辑上也是合适的，因为撤销功能在模态对话框中不可用，而在非模态对话框中可用。

非模态对话框惟一一致的终止动作是“关闭”。每个非模态对话框应该有一个设置在固定位置（如右下角）的关闭按钮。各个对话框应该保持一致：相同的位置和相同的标题。在这一点上的要求不必太严格，但必须使用“关闭”这个词，对话框也决不可使这个标签失效或改变它。

如果关闭按钮激活了关闭对话框以外的其他功能，你应该创建一个遵循模态习惯用法的模态对话框来取代它。

不要忘记非模态对话框经常会有几个按钮，可以立即激活不同的功能。只要单击这些功能按钮中的一个，对话框不应该关闭。因为它可以反复使用，只有单击位置一致的关闭按钮时才会关闭，所以它是非模态的。

非模态对话框也必须特别节约像素。它们驻留在屏幕上，占据前面的中心位置，所以必须特别小心，在不必要的情况下不要浪费像素。出于这个原因，尤其在悬浮调色板的场景下，要得到惟一的关闭控件，标题栏上的关闭框可能是最好的解决方案。

---

[2] **译者注** 因为非模态对话框的改变是立即的。

【更激进的解决方法】

前面描述的只是一个过渡的解决方案，还有一种更激进的方法可以解决非模态对话框问题。

我们当前常用的有两种非模态习惯用法。在两者之间，非模态对话框是较老的一种。另外一种虽然是用户界面的新来者，却取得了前所未有的成功：工具条和图标按钮。因为它的品质，以及方便快捷的特点，工具条习惯用法到现在为止取得了广泛的成功。但究其实质，只不过是一个持久地粘附在程序主窗口（或边上）的非模态对话框。

工具条图标按钮的非模态性是完全可以接受的，因为它不是以对话框的视觉形式传递给我们的。相反，它们表现为工作区周围无所不在的工具。没有对话框的常见陷阱，确信不会与对话框混淆。我们不难理解它们的用途，尽管相同的控件如果出现在对话框时经常会令我们感到迷惑不解。

当在当前选择的场景下不起作用时，工具条图标按钮能够，也应该（如第 29 章中所讨论的那样）变灰。这种行为用户容易理解，同时也解决了另一个非模态对话框问题。

总之，工具条是非模态的，但它们没有非模态对话框存在的问题。它们还有非模态对话框不具有的两个特性：在视觉上与对话框不同，不必担心关闭问题，因为它们无处不在——因此它们不需要终止控件。它们也解决了其他问题，工具条在屏幕空间使用方面有惊人的效率，特别是和对话框相比，而且它们不遮盖当前正在操作的空间！

非模态对话框是自由悬浮窗口，允许用户随心所欲地将它们放在屏幕的任何地方。它们不利的方面是需要窗口管理的附加工作。停放工具条（Docking toolbars）（在前述章节中讨论过的，见图 29-4）是极好的解决方法。你可以单击拖放停放工具条，一离开程序边界它就立刻变成了一个悬浮调色板（也称为调色板窗口）——换句话说成为了一个非模态对话框。你可以让它这样，或者将它拖动到程序主窗口的任何一边，在那儿它又变成工具条，停放在窗口一边。

消除窗口管理附加工作非常重要，特别是当我们谈到例如在 Adobe Photoshop 和其他绘画程序中的大批悬浮调色板时。回到显示屏分辨率只有 640×480 的时代，让这些调色板悬浮有充分的理由——这样用户可以将它们拖到不碍事的地方而且仍然可以访问这些工具。但当显示屏的分辨率逐渐增加，而悬浮控件数量逐渐增多，悬浮调色板碍事的问题又出现了。Adobe 部分地意识到了这个问题，允许调色板对齐主窗口的任何一边，但他们没有做到最后一步——允许它们停放在主窗口而不会隐藏后面的任何东西。另一方面，Macromedia 在 Firework MX 和其他最新的应用程序版本中奉行可停放调色板的理念。停放调色板和工具条消除了窗口管理附加工作，而且仍然给用户提供了在需要时移动个人工具的灵活性。

如图 30-1 所示，这是微软 Word 的一个悬浮工具条。它通常位于主窗口顶部水平行位置，就在菜单栏的下方。

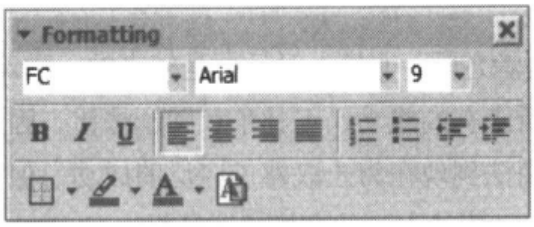


图 30-1　这是微软 Word 的一个悬浮工具条。它也是非模态对话框！更小的标题栏使它呈现与模态对话框不同的视觉外观。上下文相关的不活动图标按钮不会妨碍到任何人，如果所有的非模态对话框都以这种方式显示，大多数由此产生的混乱就会消失。另外，如果你把这个悬浮工具条拖动到应用程序的一边，它就会像熟悉的固定工具条一样停放在程序的一边。想像一下如果你对任何非模态对话框都这样处理——例如查找对话框，那会怎样呢？

现在让我们想像一下 Word 的查找对话框作为悬浮工具条显示情景，如图 30-2 所示。它有一个更小的标题栏取代正常标题栏，没有终止按钮，而是依靠小标题栏上的关闭框。如果我们把这个新的查找对话框拖到主窗口的最上边，会怎么样呢？如果行为一致，它会停放：查找对话框表面的控件应该像格式工具条这样以水平工具条的方式分布。如果对于所有那些格式图标按钮和组合框可以这样，为什么查找对话框不能这样呢？为什么我们的工具条上不能有“查找下一个”的图标按钮和针对不同选项的复选框（如果不是所有功能，至少是大多数常用的功能）？最好的例子是 Google 工具条，一个可以下载，用于 IE 浏览器的工具条控件，它实现了 Google 的搜索功能和惊人数量的附加功能。

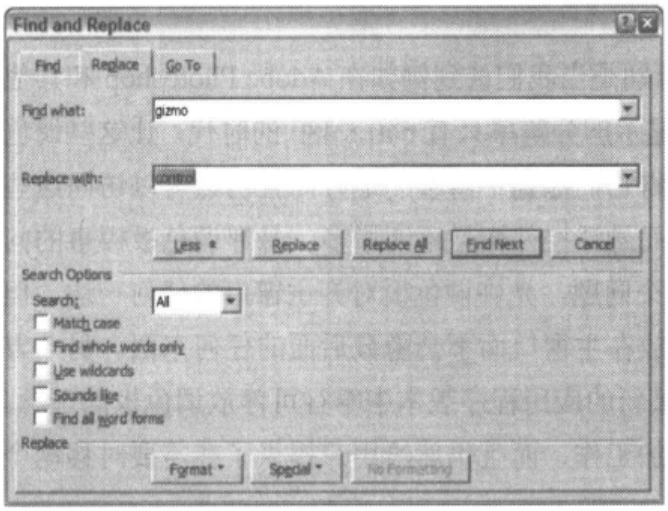


图 30-2　这是微软 Word 中的一个典型非模态对话框。多么混乱！它面积大，遮盖了它需要查找的文本（如果找到的单词位于对话框下面,Word 会推开对话框），它的按钮标签不清楚。例如取消按钮，取消什么？为什么像查找这样的功能不能进入主窗口界面？回答是，它们可以。

微软在它的办公套件中制订了悬浮工具条和停放工具条（floating and docking）习惯用法的标准。所有程序都包括一个根据用户喜好定制工具条的地方。这是用户界面发展的一大进步，工具条是取代非模态对话框的一种新的习惯用法，它应该用在任何可能的地方。

目标导向对话框

模态对话框和非模态对话框的概念来源于程序员的术语。它们影响了我们的设计，但我们也必须从目标导向的观点来考察对话框。就这个角度，有四种基本的对话框：属性对话框（property dialog）、功能对话框（function dialog）、进度对话框（process dialog）和公告对话框（bulletin dialog）。

属性对话框

属性对话框（property dialog）向用户呈现所选对象的属性或者设置，并且允许用户进行改变。有时属性可能与整个程序或文档相关，而不仅仅与一个对象相关。

Word 中的字体对话框（如图 30-3 所示）是一个合适的例子。用户在主窗口中选择一些文本，然后从格式菜单中打开字体对话框，对话框能让用户改变所选字符的字体相关特性。你可以把属性对话框当做针对所选对象的控制面板，它包含一些可调用的配置控件。属性对话框可以是模态的，也可以是非模态的。

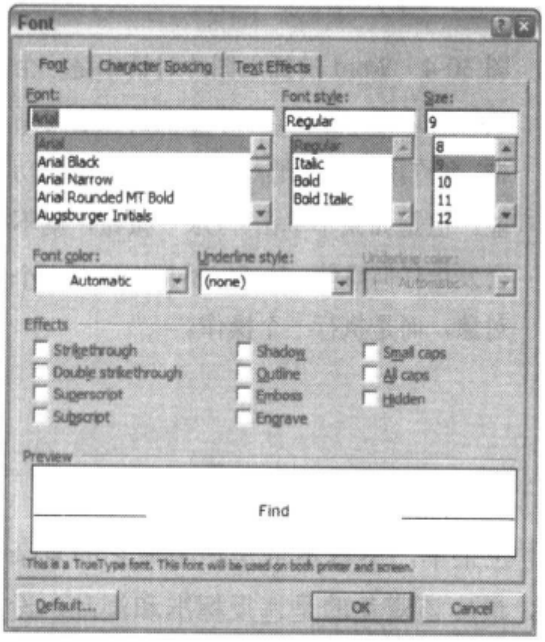


图 30-3 Word 中的字体对话框是一个属性对话框。它反映了当前所选内容的所有排版特性。当用户在这个对话框中操作控件时，预览区选择文本的排版特性会发生改变，但直到单击了“OK”按钮，文档中的属性才会改变。这种操作实质上是一种配置。预览框很大，但为什么左上角的字体字段不能也用真实字体呢？

一般来说，属性对话框控制当前的选择。这遵循的是对象动词形式：用户选择对象，然后通过属性对话框为所选对象选择新的设置。

功能对话框

功能对话框（function dialog）通常从菜单打开，它们是最常见的模态对话框，控制单个功能，如打印、插入对象或拼写检查。

功能对话框不仅允许用户开始一个动作，而且也经常允许用户设置动作的细节。例如在许多程序中，当用户请求打印，它使用打印对话框指定打印多少页，打印多少份，向哪一个打印机输出，以及其他与打印功能相关的设置。对话框上的终止 OK 按钮不仅确认设置，关闭对话框，而且启动打印操作。

这种技术虽然普通，但将两种功能组合成一种：配置功能并调用。仅仅因为一个功能可以配置，并不意味着用户在每次调用之前都想重新配置。因此让这两种功能分开访问会更好。

现代软件的许多功能相当复杂，有许多可配置的选项。相应地，它们的控制对话框也很复杂。

以图 30-4 所示的对话框为例，用户首先通过选择文件来配置操作，然后通过单击终止命令按钮执行配置命令。和本例中设计师所做的一样，用“插入”这一终止按钮取代“OK”按钮看起来很吸引人，但你应该避免这种情况，表面上看起来似乎更符合逻辑，但它付出了失去终止命令按钮一致性的巨大代价。如果对话框的标题栏文本合适的话，它看起来就会像个词组，确切地告诉用户将会发生什么：插入这个图片……

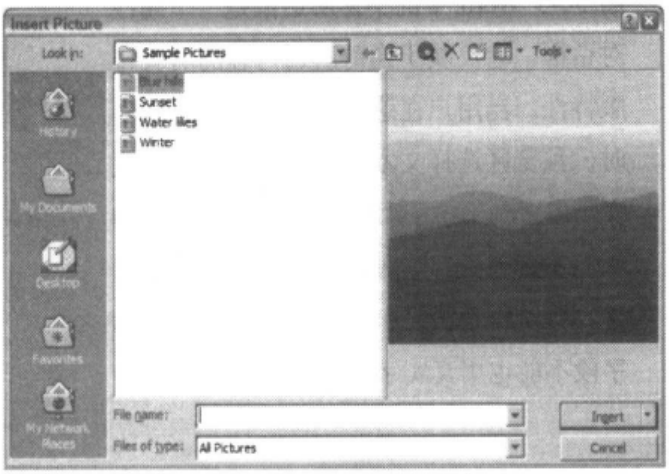


图 30-4 Word 的插入图片对话框是一个功能对话框。它还是一个经典的模态对话框，允许用户首先通过选择文件来配置功能。但是如果不单击“OK”按钮，就不会发生任何事情。对话框不是作用于一个对象，而是执行一个操作。

注意此图中的插入按钮还有一个与之相关的下拉菜单，尽管本意是想隐藏它的复杂性（或许是节约空间），但对于对话框的终止控件，最应该避免的是违反标准和混乱。设

计师应该采取其他不同的方法。

进度对话框

进度对话框（process dialog boxes）是由程序启动，而不是根据用户请求启动的。它们向用户表明当前程序正在忙于某些内部功能，其他功能的性能可能会降低。

进度对话框提醒用户程序不能正常响应，同时也警告用户要耐心，不要为催促程序而反复敲打键盘。

开始某个要占用一定时间（从人们的感觉上）的进程之后，程序必须清楚地表明它正在忙，但一切正常。如果程序没有表明这些，好一点的话，用户可能只会认为它粗鲁；不好的话，会认为程序已经崩溃，必须采取激烈的措施。

设计技巧：当程序将变成无响应状态时，必须通知用户。

正如我们在第 21 章中所讨论的，当前许多程序依靠活动的等待光标暗示，把光标变成沙漏，但是进度对话框是一种更好而且更有信息量的解决方法（我们在本章的后面会讨论更好的解决方法）。

每个进度对话框有四个任务：

- 向用户清楚地表明正在运行一个耗时的过程。
- 向用户清楚地表明一切正常。
- 向用户清楚地表明进程还需多长时间。
- 向用户提供一种取消操作和恢复程序控制的方式。

进度对话框本身就满足了第一个要求，提醒用户某个过程正在发生。第三个要求可以用某种**进度表**（progress meter）来实现，显示已经完成的工作所占的相对比例，以及还有多少需要完成。要满足第三个要求不容易，程序可能会崩溃，对对话框置之不理，也不向用户说明操作状态。进度对话框必须持续显示，通过时间的变动表明一切进度顺利。进度表应该显示相对整个过程所耗费时间的进度，而不是相对整个过程规模的进度。一个过程的前 50%可能与接下来的 50%在时间上差别很大。

用户对计算机执行一个耗时进程的心智模型很像是一个机械转轴。一个表明计算机在读磁盘的静态对话框只能告诉用户一个耗时的进程正在发生，但它并不意味着那是真的。最好的进度显示方式是在对话框中使用动画，在 Windows 中，当移动、复制或删除

文件时，进度对话框会显示一个小的动画图像，纸片从一个文件夹飞到另一个文件夹或垃圾箱（见图 30-5）。这种效果很明显：用户意识到计算机真的在做某件事。这种一切正常的感觉是来自内心，而不是大脑，用户——甚至是专家用户——也会恢复信心。

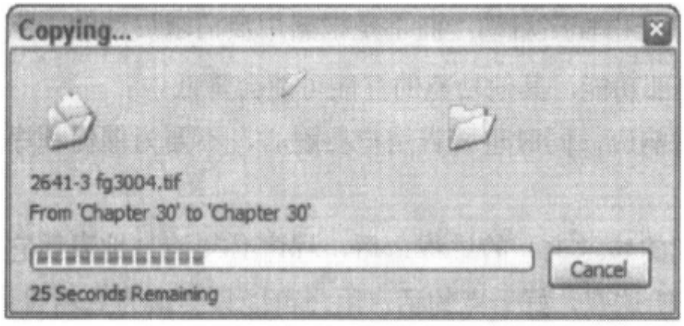


图 30-5 微软的这个进度表基本上是正确的。对于资源管理器中的任何一次移动、复制或删除操作，它们显示了设计合理的进度对话框，它提供了操作剩余时间的线索，还使用动画显示文档从左侧的文件夹飞到右侧的文件夹（或垃圾箱）。用户的心智模型认为计算机内部的某个事物在移动，这个精致的小图片真实地反映了移动的事物。从用户的角度看，这个对话框让人耳目一新，计算机的外部形象反映了计算机的内部操作。不足之处在于没有显示移动中剩余的文件数量，否则它可以方便地提供更好的进度反馈

对第三项要求来说，微软的进度表通过提示过程的剩余时间只能刚刚满足。每一次操作都有一个对话框，但操作可以影响许多文件。因此，对话框也应该动态地显示操作中文件的剩余数量（例如，“37 个文件，还剩下 12 个”）。现在的进度表只显示正在转移的单个文件进度（有趣的是，标准的 Windows 安装过程确实用了一个进度表来指示还有多少个文档）。

注意图 30-5 中的复制对话框还有一个取消按钮。表面上这可以满足第四个要求，有一种方式用于取消操作。用户可能会重新考虑操作所花的时间，决定推迟进行，取消按钮能够让他们做到这一点。但是，如果用户意识到他调用了错误的命令，并且希望取消操作，在这种情况下，他不仅希望停止操作，而且还希望能消除所有的操作痕迹。

如果用户从目录 Alpha 拖动 25 个文件到 Bravo 中，移动的过程中，他意识到其实自己真正想做的是将它们放到目录 Charlie 中，他会按下取消按钮。但不幸的是，取消按钮只能停止移动和放弃移动剩余的文件。换句话说，如果用户在已经复制了 10 个文件后单击取消按钮，剩余的 15 个文件仍在目录 Alpha 中，而前 10 个文件已经到了目录 Bravo 中，这并不是用户所希望看到的。如果按钮标明是“取消”，它就应该代表取消操作，也就是说“我不希望发生任何事情”。如果按钮要确切地表述它的行为，它应该说“停止移

动”或“停止复制”。而与此相反，它说的是“取消”，所以取消操作是它应该做的。这也许意味着需要一些大的缓冲区，取消操作可能比初始的移动、复制或删除消耗更多时间。但是仅仅因为需要额外的时间，这种不多见的情况就不正当了吗？在 Windows Explore 中，程序可以完全撤销一个复制、移动或删除，所以没有理由说取消按钮不能撤销已经执行的部分。

一个好的替代方法是对话框中采用两个按钮：一个标明“取消”，一个标明“停止”，用户可以根据自己的需要选择其中之一。

消除进度对话框

对进度对话框的需要并不是很清楚。在当前的软件中，它们像杂草一样遍地可见，用途也如同杂草。但我们能用什么来取代它呢？

这个问题的答案要看是谁在进行处理。因为一个对话框是一个单独的房间，我们必须了解对话框所报告的过程是否与主窗口的功能互相独立。如果功能是主窗口所显示功能的完整部分，它的状态就应该在主窗口中显示。例如，图 30-5 显示的 Windows 飞动页面对话框就很吸引人，也很合适，难道复制文件不是资源管理器的基础功能吗？在这种情况下，动画可以直接建立在资源管理器的主窗口中。小小的文档页可以飞越状态栏，或者直接从主窗口的一个目录飞到另一个目录。

当然，进度对话框比直接在程序主窗口创建动画要容易得多。它们也提供了便利的“取消”按钮，所以在耗时的任务中显示一个进度对话框是合理的权衡。但我们不能无视这样的事实，这样做相当于“我们将这个房间的功能拿到另一个房间去完成”。这是一个容易的方法，但不是正确的方法。

公告对话框

公告对话框是一种简单的小东西，无疑它是任何图形用户界面中滥用得最多的元素。和进度对话框一样，它无须请求，由程序直接启动。

普遍存在的错误信息框是最典型的公告对话框。通常，程序名在标题栏中显示，一些概括问题的简短文字作为显示主体，一个图像图标表明问题的严重性，还有一个“OK”按钮，通常这些构成了一个统一的整体，有时再加上一个启动在线帮助的按钮。图 30-6 所示的是 Word 中的一个例子。

我们熟悉的信息框通常是一个应用程序模态对话框，它将让程序停止所有的下一步处理，直到用户发起终止命令——比如单击“OK”按钮。这种情况称为阻塞型公告，因

为直到用户回应，程序才能继续。

程序也有可能显示一个对话框，然后又让它消失。这种类型称为临时公告，因为无须用户参与对话框自动消失，程序可以继续。

临时公告有时用于错误报告。显示错误消息来报告问题的程序可能会自己纠正错误，也可以通过其他代理检测问题是否消失。一些程序员会发布错误或通知公告仅作为警示，例如你的磁盘快满了，10 秒钟后它又自动消失。这种类型的行为存在可用性问题。

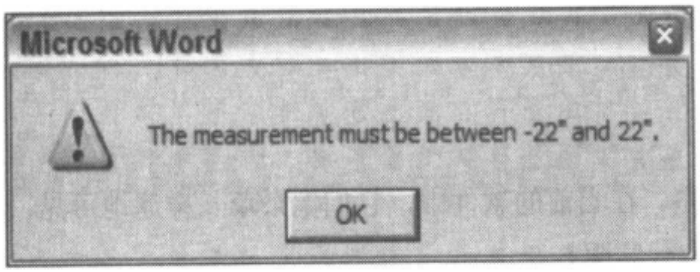


图 30-6 这是一个典型的公告对话框。它从来不会由用户发起，当程序无法完成工作，或只是它想宣告自己已经渡过难关时，由程序单方面发布。程序认为责备用户比继续解决问题要容易得多。用户则会这样理解警告，“度量范围必须在-22 英寸到 22 英寸之间，你连这个最基本的事实都不知道，真是个不可思议的小丑。你太愚蠢了，事实上，我甚至不打算为你改正它！”

一个错误消息或确认信息必须停止程序，否则，用户也许不能仔细阅读它；或者如果他把脸转过去了，要么看不见，要么甚至更糟糕只用眼角瞥到一点。他有理由怀疑错过了一些重要的东西，某种以后会回过头来困扰他的东西。他开始担忧：我错过了什么？那是我不知道就会后悔的重要信息吗？是系统不稳定吗？是不是即将要崩溃？即使问题已经自行解决了，也真的会出现这些顾虑。

如果某件事值得用对话框来表达，那确保用户清楚地获得这些信息，也是值得的。一个临时公告不能保证做到这一点，它不应该用于错误报告或寻求确认。

设计技巧：绝不要用临时对话框作为错误信息框或确认信息框。

属性对话框和功能对话框由用户有意调用——它们为用户服务。但是，程序发起的公告对话框——它们为程序服务，牺牲用户利益。错误、通知（警告）和确认消息都是阻塞型公告对话框。我们将看到，在大多数情况下就算是这些也都能够避免，并且应该避免。我们将在第 33 章和第 34 章中详细讨论这个问题。

31

对话框礼节

上一章我们讨论了对话框方面一些较大的设计问题。本章，我们进一步关注行为良好的对话框应该怎样表现。即使是一个合适的对话框，它所展现的行为也可能出人意料，甚至让人生气。通过注意细节，我们可以把对话框从行为粗鲁的打断者变成礼貌而有帮助的服务员。

礼貌是对话框的美德

你应该还记得，我们将对话框分成四类：属性对话框、功能对话框、公告对话框和进度对话框。这些对话框类型最重要的区别是它们的调用方式。前两种只有在用户明确提出请求时才出现，而后两种由程序单方面发起。这也暗示了在语调和表达方式的不同。如果召唤一个仆人，你希望他步履轻快地走进房间，清楚而明白地直接提供服务。另一方面，当他想向你提出要求时，你希望他谦恭地等在一旁，直到你有好心情，而不是强加他自己的需要而打断你的遐想。就这种精神而言，公告对话框和进度对话框应该比属性对话框和功能对话框表示出更多的尊重。不幸的是事实正好相反。

用户请求的对话框可能体积较大，而且位于屏幕最前方的中心位置。而非请求的对

话框则不应该如此失礼——除非当前用户处于明显的危险之中。非请求的对话框应该体形更小，使用的空间更紧凑，出现在屏幕的一边以避免挡住用户的视线。

标题栏

如果对话框没有标题栏，它就不能移动。所有对话框都应该是可以移动的，那样可以避免遮盖与它们重叠的窗口内容。因此，所有对话框都应该有标题栏。这一点清晰吗？甚至 Windows 风格指南在这一点上也达成了一致，称“通常，应用程序应该只使用可移动的对话框。”

设计技巧：所有的对话框都应该有标题栏。

似乎有这样一种观念，系统模态消息（当然你决不会创建）不需要标题栏，因为它们常用于报告致命的错误。程序员的推理是：“既然系统已经崩溃，那么为什么还要麻烦用户移动对话框呢？”当然，在系统崩溃重新启动之前，你需要看看屏幕上的内容，毕竟，你极可能马上就会丢失所有的屏幕信息。

对话框标题栏上的文本字符串也是一种普遍的疑虑。有人认为它应该是功能名，而另一部分人则认为应该是程序名。折中主义者倾向于两者都使用。正确的答案非常简单：这些都不对。

如果对话框是功能对话框，标题栏上应该有功能性动作——如果你愿意，可以是动词。例如，如果你从插入菜单中请求分隔符，那么对话框的标题栏应该称“插入分隔符”。我们在做什么？我们在插入一个分隔符！我们不是在进行分隔，所以标题栏不能称为“分隔符”。这样的一个单词很容易引起惊慌和迷惑。

设计技巧：在功能对话框的标题栏中使用动词。

在实践中，当在某些选择内容上进行功能操作时，标题栏应该尽可能地指明所选内容。例如，如果你选择了这个句子，“Smilin'Ed is dead”，然后从格式菜单中调用字体菜

单项，所出现的对话框标题栏应该是“Smilin'Ed is dead 的字体格式。”如果你选择的文本太大，不适合放在标题栏上，那也应该显示句子的第一个和最后两个单词，中间用省略号隔开。如果没有任何做选择，那标题应该是“将来文本的字体格式”。

设计技巧：在属性对话框标题栏中恰当地使用对象名。

如果对话框是属性对话框，标题栏上应该有我们正在为它设定属性的对象名或描述。Windows 中的属性对话框遵循这种方式，当你为 Backup 目录调用属性对话框时，标题栏上写着“Backup 属性”。

暂时姿态

如果对话框是独立的程序，那它们是暂时姿态的程序。你可能会希望，对话框在外观和行为上应该像暂态程序，有醒目的视觉习惯用法，明亮的颜色和巨大的按钮。另一方面，对话框占用独占应用程序的空间，所以它们决不能浪费像素。大一点的规则还是小一点的规则不断冲突。一个解决方法是使单个控件大一些，而又要保证对话框本身会不浪费额外的空间。

设计技巧：对话框应该尽可能小，要适度。

Borland 推广的一个标准是在界面上用位图符号（bitmapped symbol）创建特别大的图标按钮：一个大红叉代表取消，一个大绿勾代表 OK，一个蓝色的大问号代表帮助。开始的时候，它们是非常聪明的设计，也非常吸引人。但现在很多人有理由发现它们很浪费空间。图标按钮上的图标在视觉上可以很好地区分它们自己，不需要额外的空间。Borland 现在将同样的位图应用到更常规大小的图标按钮上，这是更好的解决方法。无须浪费珍贵的像素，视觉图像仍能很好地完成工作。

对话框遮盖父窗口的面积应该最小，对话框也绝不能占用超过自己需要的空间。在现代桌面计算机中，像素仍然是最有限的资源，对话框不应该散乱地分布在屏幕上。比

较一下图 31-1 中 CompuServe 的导航对话框与图 31-2 中 Word 对话框的空间利用效率。

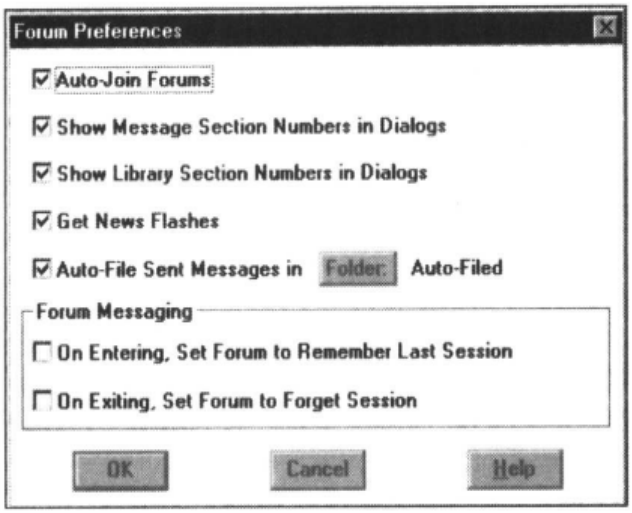


图 31-1 这是来自 Windows（1.0 版）下的 CompuServe 导航器的一个属性对话框。散乱分布的复选框占据了许多空间。不过至少该对话框还有一个标题栏，在它妨碍你时，可以移动。还要提到的是它蹩脚的布局，将“OK”按钮放在最左边而不是右边。

复选框是空间利用效率相对较低的控件：伴随的文本需要许多专用空间。与复选框的文本相比，图标按钮可以像沙丁鱼一样拥挤在一起。但就算是复选框也不需要导航对话框中那么多的空间。

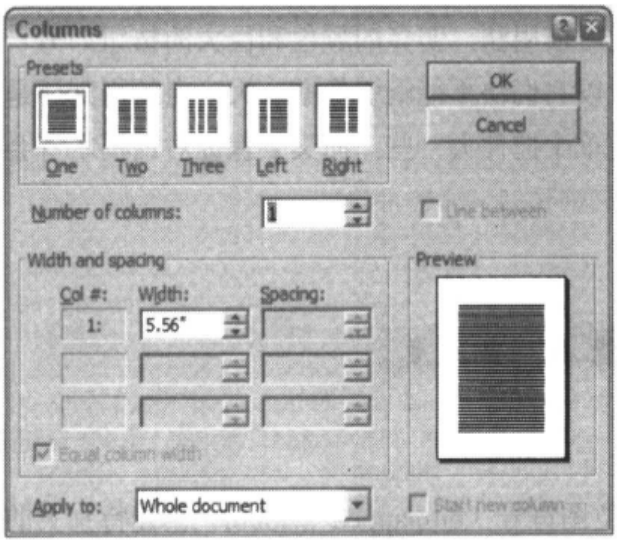


图 31-2 来自微软 Word 的一个典型功能对话框，它充分地利用了空间。控件布局紧凑，非常节省空间。将此图与前图（图 31-1）比较，注意到微软愿意用图形对象代替固定的基于文本的控件，像编辑字段、复选框和按钮。注意右上角终止控件的奇怪位置。在这种情况下，设计师也许在高效利用空间方面有点过头。

减少附加工作

正如我们在第 10 章所讨论的，如果需要很多附加工作（excise）——不必要的开销，那么对话框可能会成为用户的负担。用户很快就会厌烦每次出现时都要重新定位和配置对话框。对话框设计师的职责是确保附加工作最小，特别是因为对话框在交互这场戏中

只是一个配角。

了解需要减少附加工作的位置

在通常情况下，对话框无法降低附加工作是因为它们的位置和状态。对话框应该总是记得上次打开的位置，它们应该自动回到那个位置。大多数对话框每次激活时都会重新开始，一点也不记得上一次的实例化的情形。这是对话框实现方式的产物：作为拥有动态存储的子程序。我们不应该让这些实现细节影响程序的行为方式。对话框应该总是记得它们上次调用的状态，并且应该回到与之相同的状态。如果对话框扩展了或者选中了某个标签，在下一次访问时，它应该回到同样的状态。在第 15 章中，我们详细地讨论了如何使用记忆来解决这类问题。

相同的理念可以用于输入字段的内容。如果上次选中了一个复选框，对话框应该记住，在下次访问时复选框也要选中，上次使用的设置常常也是这次需要使用的。

了解是否需要减少附加工作

对话框（特别是公告对话框和进度对话框）降低附加工作最有效的方法是如果不需要就不出现。如果有某种方式让对话框足够聪明，知道它是否受欢迎，程序应该尽一切办法确定这一点，使用户免得关闭不必要的对话框：一个纯粹是附加工作的动作。

例如在 Word 中，如果你以保存、打印和关闭文档这样的顺序操作（作者总是这样做的），打印中涉及的重新分页会标明文档已经改变。这就意味着当你调用关闭命令时，程序会问你是否需要保存，尽管你已经保存过文档！程序应当注意！在关闭文档之前你当然希望保存文档，不仅根本不需要问这样的问题，而且它应该明白你并没有改变文档，只是程序它自己改变了文档。启用这种对话框完全是附加工作。

公告对话框也是这样。它告诉我们程序已经正常完成某项功能。既然非常正常，程序就不应该求助于对话框这种成为附加工作的东西，愚蠢地停止程序继续进行。

如果你每次请求某项功能时，程序都用对话框提供选项，而你又总是使用相同的选项，程序就不应该发起对话框那么麻烦，它应该可以识别这种模式，将不需要的步骤删除。当然，它应该首先通知你（请使用非模态），这样你就不会奇怪，然后它应该向用户提供选项取代它的决定。

模态对话框的终止命令

每个模态对话框都有一个或更多的终止命令。多数模态对话框有三个：OK（确认）和 Cancel（取消）按钮，以及标题栏上的关闭框。“OK”按钮的意思是“接受我做的任何改变，然后关闭对话框”。“Cancel”按钮的意思是“拒绝我做的任何改变，然后关闭对话框”。这是个简单明显的规则，一个已经建好的标准，因此，任何人违反这条熟悉、值得信赖而常用的途径都是不可思议的。但是出于无法说明的原因，许多用户界面设计师偏要脱离这种简单的规则，损害他们的产品，让用户失望。

模态对话框向用户承诺它将提供等待批准的服务——“OK”按钮，以及一种没有危害的简单退出方式——“Cancel”按钮。这两个按钮不能遗漏，否则就会有损于用户已经对程序建立起来的信任。要得到用户的宽容是极其昂贵的，绝不要遗漏这两个按钮，或改变它们的标签。

设计技巧：为所有的模态对话框提供“OK”和“取消”按钮。

一个同事反对这个技巧，提出如果有一个询问用户是否想“取消保留？”的对话框，那么它与“OK”按钮、“Cancel”按钮一起使用时会有问题。取消“Cancel”究竟是什么意思呢？这是个好问题。解决问题的方法就是决不要那样提问。如果你确实需要问一个那样的问题，也不应该使用与标准终止控件相同的单词。用“取消保留？”这样的话进行询问，用户必须用“取消”回答避免取消。这不是很混乱吗？真的！相反，这个问题应该这样陈述：删除保留？但更好地方法是我们将在第 7 部分中谈到的，如何完全消除确认对话框。

设计技巧：在对话框文本中决不使用与终止命令相同的单词。

“为所有的模态对话框提供 OK 和取消按钮”，这个设计技巧只适合用在功能和属性对话框中。报告错误的公告对话框——这非常可恶——可以只用一个 OK 按钮（好像用户希望与程序一起犯错误一样！）。进度对话框只需要一个取消按钮，这样用户就可以结

束费时的过程。

OK 和取消按钮在任何对话框中都是最重要的控件。这两个按钮在视觉上必须有直接可以辨别的特征，与对话框中的其他控件明显不同，尤其是其他动作按钮。这意味着将几个有相同外观的按钮与 OK 和取消按钮排列在一起是不对的，无论这种做法多么常见。甚至在一些本应该了解这些原则的公司，将 OK 和取消按钮埋葬在其他各组无关的按钮中，并经常令人难过地改变熟悉的标签。

取消按钮对于对话框的教学目的来说尤其重要。当新的用户浏览程序时，他想检查对话框来了解它们的范围和目的，然后再取消操作，就不会带来麻烦。对于更有经验的用户来说，他们认为 OK 按钮比取消按钮更重要。用户调用对话框，做出改变，按下确认的“OK”按钮，然后退出。

有一段时间，微软采用了一种终止按钮的新标准。它规定“OK”按钮位于对话框的右上角，“Cancel”按钮直接放在它下边，下方还有帮助按钮。感谢上帝，微软在最近的版本中已经废除了这种错误的方式。绝大多数用户的阅读习惯是从左上到右下，所以终止按钮位于对话框的右下角更好理解。微软也采用了灰色的外观，终止按钮在视觉上不能通过任何独特颜色、形状或者字体和字号识别。它们与对话框中的其他按钮互相混淆，这样很不好。

“OK”按钮应该放在对话框的右下角，“Cancel”按钮应该放在它的左侧，从而用户可以确信去最右下角可以结束对话框。值得赞扬的是苹果公司的员工从开始就坚持这么做。为什么其他这么多公司还一直有这些烦恼呢？

关闭框

因为对话框是有标题栏的窗口，它们还有另外一个终止习惯用法。单击右上角的关闭框（在 Mac 中位于左上角）会终止对话框。这个习惯用法的问题在于对用户改变的处理不清晰：是接受改变还是拒绝？它是等同于“OK”按钮？还是“Cancel”按钮？因为存在混淆的可能，程序只有一种可能的解释方式，将它等同于“Cancel”按钮。不幸的是，这与非模态对话框的含义存在冲突，因为在非模态对话框中，它与关闭命令相同。非模态对话框而不是模态对话框需要关闭框，因此，为免混淆，模态对话框中不应该有关闭框。

设计技巧：在模态对话框中不要设置关闭框。

如果用户希望有一个“OK”按钮，得到的却是一个“Cancel”按钮，他会觉得奇怪，不得不重新做一遍，然后才会知道。另一方面，如果用户希望的是一个“Cancel”按钮，而得到一个“OK”按钮，他仍然不得不重做，但这次他就会生气了。不要出现这种情况。

帮助按钮

帮助按钮经常和“OK”按钮、“Cancel”按钮同等重要。它请求上下文敏感的帮助，而不是终止对话框，所以它不是终止按钮。它经常和终止按钮放在一起，以致人们产生这样的联想：认为它也和终止按钮一样重要，可是在线帮助没有终止命令重要。将帮助按钮放到它的旁边不太好，但也没有坏处，它也是一种人们熟悉的标准。帮助按钮放在对话框标题栏上可能更好，在许多最新的应用程序版本中，微软已经表明它对这个问题的理解。如图 31-3 所示，微软已经将帮助按钮与“OK”和“Cancel”按钮这两个终止按钮分离，放到了标题栏上。标题栏对于所有对话框来说都是共同的区域，但这样可以与非常特殊的**终止命令**清楚地区分开来（当然，关闭框除外）。

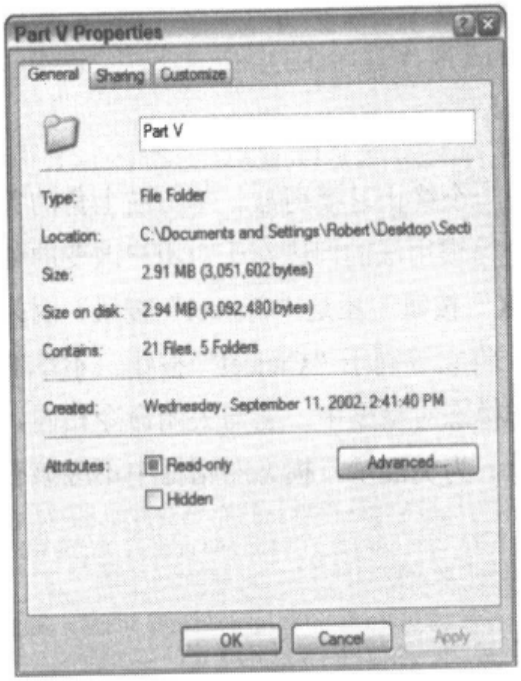


图 31-3　这个 Windows XP 的属性模态对话框表示微软最终意识到帮助不是终止命令。帮助按钮从终止按钮套件移到了标题栏关闭框附近。这当然是个进步，但微软进一步在终止按钮行增加了另一个控件：应用。在这个标签窗格上“应用”功能没有用处，但在“分享”窗格上它是可用的。为什么不把应用按钮放在有意义的窗格上，并且将它与终止按钮分开？

键盘快捷方式

许多对话框提供经常使用或反复使用的工具，如替换和查找工具。随着用户对程序越来越熟悉，他们期望有这些常用对话框的键盘快捷方式。通常有足够多的键，所以没有理由说特定的功能只应该有一种键盘快捷方式。如查找功能应该可以通过 Ctrl+F 调用，也可以通过特殊功能键 F2 调用。替换功能可以是 F3 和 Ctrl+R。

用户可能从帮助系统，也可能从菜单了解到这些快捷方式。通常，这些快捷方式不会引人注意，除非是期望使用它们。新手用户直接使用菜单，只有在他们主动寻找更加快捷的操作方式时，才会发现它们。他们非常感谢你已经提前为他们准备了快捷方式。它们可以真正取悦超级用户群，而这个用户群会对新手用户产生巨大的影响。

尤其受超级用户欢迎的一个特点是定制和配置这些快捷方式的工具，尽管这不是所有的用户（甚至不是多数）会用到的工具。超级用户可能会非常倚重它，因为它允许他们调整控件，最好地匹配他们的工作风格，这总是超级用户感兴趣的事。

标签对话框

给商业软件世界带来一场风暴的另一个用户界面习惯用法是**标签对话框**（tabbed dialog）。在不到两年的时间内，标签对话框，如图 31-4 所示，从一个完全不为人知的习惯用法一跃成为好的设计标准，现在它以一种稍微退化的形式应用到了几乎所有的商业网站。当一个习惯用法有优点的时候，它就会广泛复制，对于对话框设计师来说，标签控件就是这样的幸事。它已经成为所有图形用户界面的标准组成部分。

标签对话框允许对话框本身或者其中的一部分设置成一系列重叠的**窗格**（panes），每一个窗格有一个突出的特征**标签**（tab）。单击标签将使与相关的窗格处在最前面的位置，隐藏其他窗格。标签可以水平排列在窗格的上边或底下，也可以垂直排列在一侧。无疑在这些变体中顶部标签最有用，因为垂直排列在窗格某一边的标签要么需要用户转过头才能读到，要么比标准标签占用更多空间，而底部标签又太容易被忽略。

具有多个属性的许多对象现在可以有相对丰富的属性对话框，而不会因为控件太多使对话框变得太大太拥挤。许多以前塞满控件的功能对话框，现在可以更好地利用空间了。在标签对话框出现以前，这些问题是用扩展对话框和级联对话框笨拙地解决的，我们在以后会简略地提到。

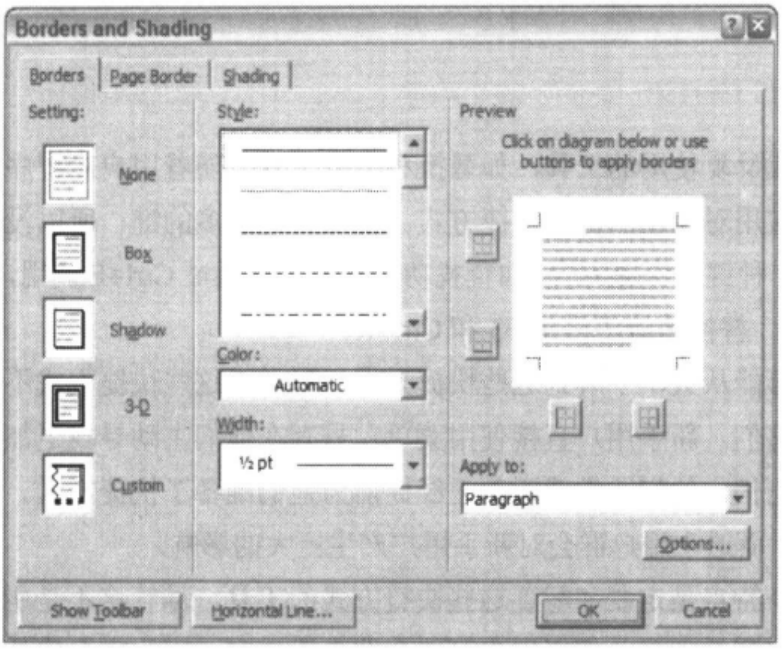


图 31-4 这是微软 Word 的一个标签对话框。如果你有方法方便地将边框（border）和底纹（shading）组合到一个对话框，会很有意义。标签就提供了这种方法。注意设计师正确地将终止控件放在标签面板之外的右下方。如果 OK 按钮是最右边的按钮（苹果公司一开始就这样提倡，但是微软就不同）那就更好了。

每个标签对话框分成两部分：堆叠窗格的部分，即**标签区域**（tabbed area）；窗格之外的对话框剩余部分，即**非标签区域**（untabbed area）。终止命令按钮必须放在非标签区域。如果终止按钮直接设置在标签区域，即使它们在各个窗格中的位置没有改变，它们的意思也会模棱两可。用户很可能会问“如果我按下取消按钮，我是只取消了这个窗格的改变，还是所有窗格中的改变？”把按钮从窗格移到非标签区域，它们的范围在视觉上才会变得清晰。

设计技巧：将终止按钮设置在标签对话框的非标签区域。

广度与深度

标签对话框允许你在一个对话框里填充更多控件，但拥有更多的控件并不一定意味着用户会发现你的界面更易于使用或者功能更强大。对话框中不同窗格的内容必须有放在一起的道理，否则，这种能力就退化为对程序员有利，而不是对用户有利。

对话框中的窗格可以组织为管理在某个专题上的深度增加或广度增加。为了组织广度，每个窗格涉及主题的不同特性，如图 31-4 所示，边框和底纹的显示方法解决了问题，而且加强了文本。在组织深度的例子中，每个窗格以更大的深度探索主题的同一特性。

堆叠标签：滥用对话框

标签之所以如此成功，是因为该习惯用法遵循了用户有关“事物存储”的心智模型：单层分组（monocline grouping）。不同控件组成几个平行窗格，只有一个层次深度。但甚至这种习惯用法也可能（经常如此）被滥用。

因为你可以在标签对话框内填充如此多的控件，在对话框内添加更多窗格的诱惑很大。图 31-5 中所示的 Word 选项对话框就是一个典型的例子。10 个标签数目太多，不能用一行显示，所以它们堆叠成两行。这个习惯用法存在的问题称为堆叠标签。也就是说，如果单击后一行的标签，整行就会前移，另一行就会退后。很少有用户会喜欢这样，因为用户单击标签而标签又在鼠标下移出，让用户感觉不安。它确实很有效，但这又是以什么样的代价呢？

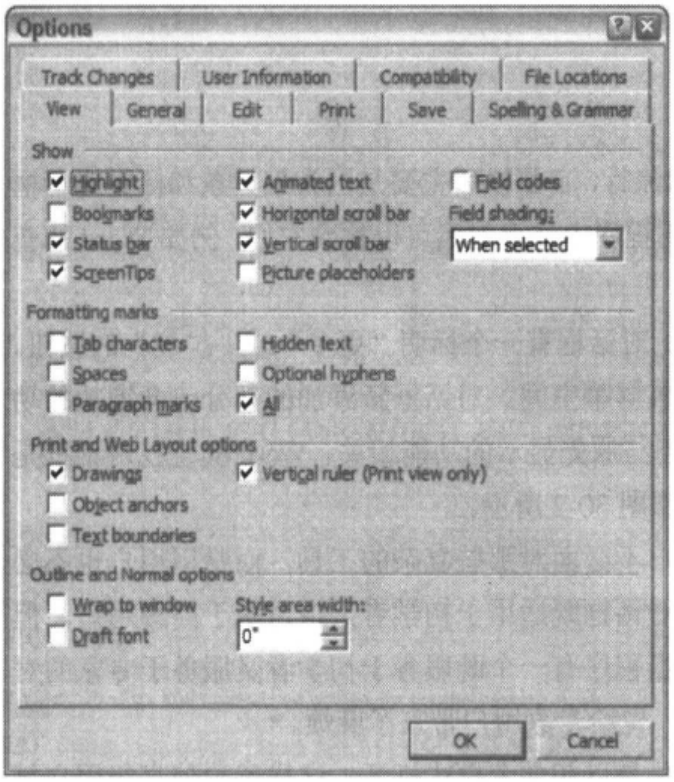


图 31-5 Word 中的选项对话框是一个关于标签的极端例子。当然，许多内容填充到一个对话框是好的。问题在于标签的移动！激活的标签必须位于底部这行，所以如果你单击拼写和语法，这一行就必须滚到底部，另一行则上升到顶部。每个人都憎恨标签在光标下移动，将它分成更小一些的对话框会更好。

公理：所有的习惯用法都有实践性约束。

堆叠标签说明了下面的用户界面设计公理：所有的习惯用法，无论有多少优点，都有它的实践性约束。5 个单选按钮组成一组可能很好，但 50 个单选按钮组成一组就很荒

谬。五六个标签组成一行是好的，但是添加过多的标签导致标签堆叠，就破坏了这个习惯用法的作用。

一个更好的选择是采用几个分别有着少一些标签的对话框。在本例中，选项的分类太广，将所有这些功能混在一块对用户没有任何好处。12 个窗格之间几乎没有联系，所以不需要在它们之间移动。这种解决方法可能在编程中少了那么一点点优雅，但对用户可能更好。

设计技巧：不要堆叠标签。

扩展对话框

扩展对话框大约在 1990 年左右非常流行，自那以后主要是因为工具条和标签对话框的无所不在而走向衰落。在许多主流应用程序中你可能还可以找到它们，例如 Word 的查找对话框。

扩展对话框扩展后会显示更多控件。对话框有一个标明“更多”或“扩展”的按钮。当用户单击它时，对话框变大，占据更多屏幕空间。对话框新添加的部分包含增加的功能，通常是为高级用户提供的，更复杂但与原先显示的功能相关。Word 的查找对话框是这个习惯用法的一个熟悉例子，如前一章图 30-2 所示。

扩展对话框使得生疏用户或新手用户不必面对那些复杂的工具，而常用用户也不必为寻找这些工具烦恼。你可能认为这种对话框既适用于初学者，也适用于高级用户。但是，这种类型的对话框必须小心设计。当程序有一个既服务于初学者又服务于专家的对话框时，经常是不但对初学者傲慢无礼，也给专家用户带来了麻烦。

多数扩展对话框是不完整的，它们总是以初学者模式打开，这样就迫使高级用户每次都不得不扩展对话框。为什么对话框不能以合适的模式出现呢？要想知道哪种模式合适，这很简单：让它维持常用模式。如果用户扩展对话框后关闭，下次调用时它就应该以扩展的形式出现，如果它上次处于收缩状态时关闭，它这次就应该以收缩状态出现。这种简单的特性可以使扩展对话框自动选择用户模式，而不是迫使用户选择对话框模式。Word 的查找对话框就是这样做的。真棒，微软！

当然，也由此产生了一个问题，必须有一个扩展按钮，也必须有一个收缩按钮（或者如查找对话框中那样，有“更多”按钮的同时，也有“更少”按钮）。最普通的方式是

只有一个按钮，在它单击后，文字说明在“更多”和“更少”之间切换。通常，改变按钮的文字说明不太好，因为它没有当前状态的提示，只指示相反的状态。但是，在扩展对话框的情况下，它的视觉特征本身已经非常清晰地显示了对话框所处的状态。

级联对话框

级联对话框是一种糟糕的习惯用法，一个对话框中的控件，通常是按钮，在一个层次关系的嵌套中调用另一个对话框，第二个对话框经常覆盖第一个对话框，有时第二个对话框还可以调用第三个对话框。多么混乱呀！幸好，级联对话框已经失宠，但我们仍然可以找到级联对话框的例子。图 31-6 中的例子来自 Windows XP。

很难理解使用级联对话框会怎么样。一部分问题在于第二个对话框至少会遮盖第一个的一部分。那也不是大问题，毕竟组合框和弹出菜单也是这样，而且某些对话框还可以移动。真正的混乱来自于第二组终止按钮的出现。每个取消按钮的范围是什么？我们确认的是什么？

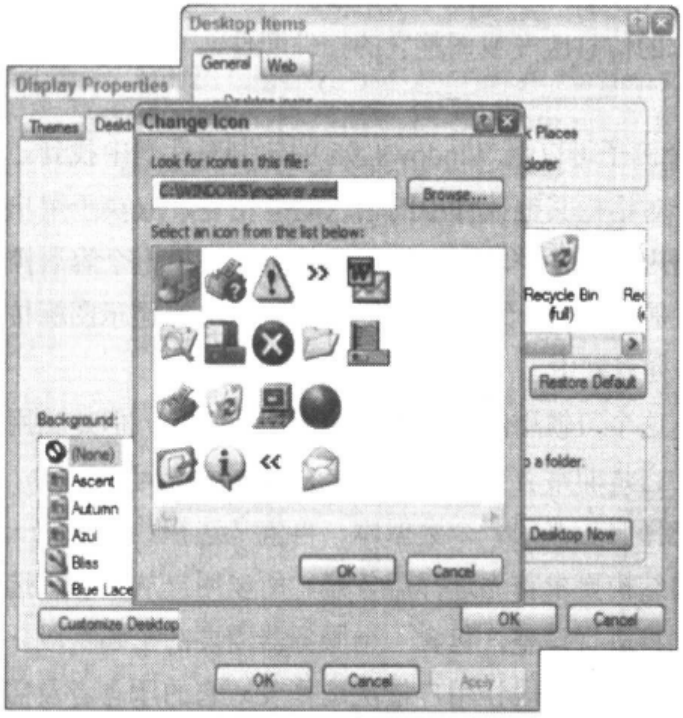


图 31-6 你仍然可以在 Windows 中找到级联对话框。每个对话框有一套终止按钮。产生的附加工作和模糊是没有好处的。

标签对话框的优势在于处理复杂的广度问题，而级联对话框更适合处理深度问题。问题在于层次太深是界面太过复杂时的主要症状，如果发现除了用户通常不需要的模糊信息以外，你的程序仍需要使用级联对话框，那就应该全面审查一下界面的整体复杂性了。

级联对话框的例子很常见。多数打印对话框允许调用打印设置对话框，多数打印设置对话框也允许调用打印驱动设置对话框。每一层对话框都是过程中更深的一个层次。在用户终止上面的对话框时，系统会将控制返回到下一个更低一级的对话框，依次类推。

级联对话框之所以存在，似乎是因为它们对程序员来说非常自然，并且反映了底层的物理过程。有时它们反映了这样一个事实：两个不同的编程小组为界面的不同部分工作，而链接对话框是最经济的集成方法。但这种捆绑和权宜的合理化违背了设计师的初衷——它忽略了用户目标及用户有关过程的心智模型。也许不能完全避免级联对话框，但它们代表非常蹩脚的习惯用法，在采用它们之前必须仔细审查。有时如上文提到的打印对话框的情况需要使用级联对话框。但是即使在这种情况下，作者也会将它们用三个标签组合成单个对话框，或者至少组合头两个对话框，只保留打印机驱动对话框，与打印主对话框分离。三个级联对话框无论对于什么目的来说都是太多了。

动态对话框

多数对话框是静态的，提供固定数目和序列的控件。对话框的一个变体，**动态对话框**（dynamic dialog）可以根据用户输入的信息改变或调整它的控件套件，甚至比扩展对话框改变得更多。

如图 31-7 所示，动态对话框的一个例子可以从 Windows 95 的定制对话框中找到。对话框表面的控件会根据用户在同一个对话框其他控件中的输入动态变化，取决于用户在分类控件中做出的选择，命令控件的内容将发生改变。用不同的东西替换整个容器控件也是可能的；这个对话框的早期版本是用一个分组框控制项（group box）显示图标按钮，但字体和宏在一个填满了文本项的列表框中。

当用户不知道特定控件的位置时，这个习惯用法可能会给他们造成混乱。但是当用户以非常清晰的顺序方式进入设置时，它是非常有效的。例如，在数据库访问程序中，用户必须首先选择一个服务器，然后选择服务器的一个数据库，再就是选择数据库的一张表，动态对话框就非常合适。问题的结构要求首先选择服务器，所以用户从列表中选择一个服务器。选择一旦做出，对话框将进行调整，包含一个服务器需要的密码字段。一旦用户选定了数据库，就会出现一个或多个字段让用户依次选择表、它的所有者及需要的其他信息。

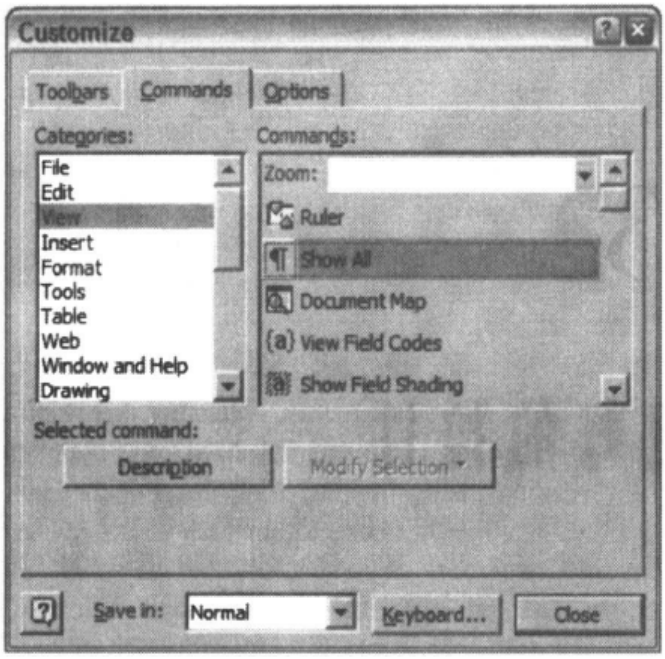


图 31-7 Word 的定制对话框是一个动态对话框的例子。根据你在分类列表框中选择的控件，右侧的命令列表框会包含控件（如图所示）、宏、字体名称或其他项。

对话框可以成为帮助你的用户完成目标的有用助手，而不是在每一步让他们遭受挫败的可怕绊脚石。通过保持对话框的可管理性，并且只有在它们的功能真的属于“另一个房间的情况”下才调用它们，你将很好地维持用户的流状态，保证他们会成功，并且满怀感激。

32

创建更好的控件

控件是与用户交互时很有价值的工具，因为它们将复杂的行为封装在直接可用的软件包里。控件的直接可用性引领我们依靠它们进行用户界面设计。我们使用它们，是因为它们是现成的。遗憾的是，大多数窗口系统的 API 包含的控件集合最多不过是个雏形。控件是发明和创业的竞技场。本章探讨控件创新的一些必要领域。

直接操作控件

对于发明而言，最令人振奋的地方在于有界输入控件（Bounded-entry control）。那么多程序提供带有编辑字段或微调控制项（spinner）收集数据的对话框，通过直接操作更容易地输入数据。利用单击拖动习惯用法取代输入控件不仅使输入更清晰，更容易，而且可以很自然地公布界限，否则（使用标准控件）就会出现无界数据输入问题。

例 1：可拖动的阴影

例如，许多绘画程序，如 PowerPoint，允许用户为填充对象添加阴影。显而易见的实现方法是采用有两个微调控制项控件的对话框，一个用于水平调整，另一个用于垂直调整，合适吗？你还可以包括一个预览图像，这样用户在输入一个新数值时可以直观地看到发生的改变，如果看上去不对，也不必单击撤销。如果用户准备好了，只要单击“OK”按钮，选择的阴影就会添加到对象上。这种界面没有什么不好的地方，但也没有真正有用或者引人注目之处。

但如果取代微调控制项和被动预览区会怎样呢？对话框有一个动态预览区，用户只需通过单击拖动操作抓住阴影，实时移动到他想要的地方。如果根本不用对话框，直接在程序主窗口中操作，用户可以在整个文档上下文中看到阴影的效果，又怎么样呢？他也许可以按下 Enter 键表示确认，而按下 Esc 键表示取消，或者当光标位于阴影的边界框中时，从出现的小控件中选择。如果偏移的实际数值很重要，那也可以作为一个覆盖图在阴影附近显示，或者在应用程序的状态区域，甚至可以在工具条或可停放调色板的可编辑字段上显示。

拖动阴影这种方法与直接操作的理念十分接近。就像木匠使用铁锤一样，用户也可以那样设置阴影，根据输入立即获得清晰的直接反馈。他不必知道三个像素是否太少，或者四个像素是否太多，刚刚好的时候他就能看到。越是充分利用直接操作，你的应用程序就越有用，越吸引人，它们的交互也是这样的。

最后在这个例子中，你还会注意到交互的一点模态性，但它还不像是模态，这是直接操作相对于使用标准对话框的部分优势所在——甚至在需要做决定的时候，它们感觉起来也不会像对话框那样打断你的思路，因为它们更好地集成到了应用程序主要的流里。当然，阴影还可以作为完整的独立效果层来实现，在这种情况下，就完全没有模态了。如果不想要，只需删除那一层就可以了。

例 2：指定网格

和前面的阴影例子一样，网格可以通过带有两个编辑控件的小对话框来控制，这两个编辑控件分别指定网格的水平间隔和垂直间隔。为什么不用动态的预览网格样本来代替这些控件呢？这样用户可以通过直接操作来调整。当用户移动光标到样本上时，光标发生改变，指示样本网格具有受范性。用户可以单击拖动样本的任何位置来调整空间，向下拖动打开垂直间隔，向上拖动可以关闭它，在水平轴上分别向左向右拖动以相同的工作原理工作。为了在调整一个轴时不影响另外一个轴，你可以应用第 23 章中提到的拖

动域，或者让用户按住 Shift 键，它已经成为绘画程序中相当标准的一个轴锁定习惯用法。你也可以想像构建在主屏幕标尺上的直接操作格线控件——抓住操作点，让它伸展或者压缩——这样甚至可以使交互更直接。

阴影控件和网格控件都用图形对象的直接操作取代丑陋而不合适的文本控件，让用户在视觉上恰当地看到他们满意的结果。用户仍可以待在上下文中，即使工具用得很少，完全可以放在对话框中。这两种控件很好地排除了文本输入的需要，而对设置提供了视觉上有界的直接操作。

摘录控件

文本编辑控件最明显的特征之一是它们竟然如此愚蠢。比方说，如果应用程序要求输入一个地址，不存在我们所需要的“地址”控件（Address Control），然而那正是我们所需要的。确实有确认控件，但它们适应不同输入（如一个完整的地址）的能力接近于零。真正的地址控件是一种摘录控件（Validation Control）。

摘录控件是解决格式数据输入的更好途径。摘录控件根据通用输入类的一些规则解析自由形式文本输入控件的内容。例如，你只要创建一个几行高的单一文本输入控件，而不需要分别为街道地址、住宅单元/序列号/邮箱号、城市、州和邮政区号各列一个编辑字段。用户在单一编辑字段中键入他满意的地址，就像他在信封或 rolodex 卡片上一样。控件可以理解地址的不同部分。听起来很好，但我们怎么做呢？

一个普通的文本编辑控件有方法（根据你的语言/编码模型，或者不同输入点，或者值）检查它的内容。通常邮编输入字段的内容是任由用户输入的。一个摘录控件除了传统的方法以外，还有很多其他内容检查方法。它们包括：

- 街道地址。
- 街道。
- 门牌号。
- 地理名称。
- 第二地址。
- 房间号。
- 建筑。
- 住宅单元。
- 邮箱。
- 楼层。

- 城市范围。
- 城市。
- 州。
- 省。
- 区。
- 邮政区号。
- 邮政编码。
- 国家。

不是所有这些值都会填充，根据用户输入的信息，只有那些相关的信息才会填充。控件会尽量确定输入文本的各部分分别属于哪个类别。在这个过程中有三个基本的区分层次。控件能够逐字返回用户所输入的文本；每行地址是分开的：街道地址、第二地址和城市地址；最后，地址的每个元素可以解析成合适的类别。

像这样的控件可以让用户以手工准备信封的相同方式输入地址：成块地输入地址。计算机在数据库程序中将它们有效地分类成不同字段，程序然后将地址按街名或邮政区号分类，即使地址是以人类可读的形式输入。

有用的摘录控件类型包括合适的名称、电子邮件、物理描述和电话号码的摘录控件。一个摘录控件很容易从单一字段中识别出人的教名、名、姓、敬称、级别和头衔，这样用户不必在输入的时候手工将它们分开。

是的，存在一个错误率的问题，但它不会很高，也不会很严重。地址解析算法可以容易地解析大多数地址，如果有人故意输入垃圾，摘录控件可能无法正确识别，但是又有多少用户会故意输入垃圾文本呢？一个购买商业软件的终端用户故意在他自己的系统里输入垃圾数据，当然不能怪你。

例如，当集成到对话框的时候，一个电话号码摘录控件可以通过应用一系列简单的语法和语义规则识别电话号码。字段的输出可以包含用户输入的原始文本，以及一系列可能的电话、传真、手机和呼机号码。如果控件不能从内容中辨别这些数字，那么他就不能识别电话号码，但在大多数情况下，只要人能识别这些数字，软件就可以辨别。让我们看一个例子，比如说我们在电话号码摘录控件中键入以下文本：

415-366-2300w，Home：367-9824 （415）367-9976 Fax 508 2031 pager

这里是一些相当杂乱的信息：不一致、符号缺失，还有着不同的标签。但是你能知道我们输入的是什么吗？当然可以。一个程序也可以！第一个号码是由区号，前缀和主体组成的完整号码。附加的 w 可以合理地解释为办公电话。逗号是个分隔符。第二个号码有一个前缀 Home：它清楚地表明它的性质。缺少一个区号不是大问题，程序很容易地确认它的区号和其他号码的区号一样，是 415。如果有不同，我们肯定会输入。第三个号

码就更棘手一些。当然，它是一个完整的号码，但它是什么意思呢？Fax 这个单词也模棱两可，它可能指的是第三个号码，也可能指的是第四个号码。但因为输入的最后一个单词是 Pager，它肯定指的是第四个号码，所以就可以确定这两个号码，第三个号码是传真号。第四个号码没有连字符，也应该不是问题，因为它仍然是一个可识别的结构良好的电话号码。

如果我们想为难一下控件，可以输入一些更有问题的数据，比如：

4558，1-800-555-1212　25433　555-FLIX

这当然增加了难度，但也不是不可能的。第一个号码 4558，不是一个可识别的电话号码，但是它可以看出是电话号码的一部分。当你在公司内部想通过 PBX（专用分组交换机）给某人打电话时，如果 PBX 的前缀是 488——程序可能已经知道——输入的号码可能就是 488-4558。第二个号码是一个结构良好的号码，它比其他号码都更完整，包括长途前缀 1，还加了五个数字的分机号。我们之所以猜它是分机号，是因为它和 800 不是一组。如果它只有四个数字，我们可能很难区分它是分机号还是一个新的号码。最后一个号码是可以识别的，尽管它不是全部用数字组成。软件要确认 555-FLIX 是个电话号码可能有困难，除非我们正在讨论的字段用于处理电话号码——那是一个很明显的线索。

你们中的许多人可能不能理解摘录控件。它们好像违背了保证数据完整性的传统。但这不是我们在第 17 章中讨论过的那种情况，而且你还可以让用户在提交数据库之前，让用户看见（如果必要也是正确的）摘录的结果。那是微软选择的解决方法。

本书的第一版出版后，微软勇敢地完成了一些摘录控件，像这里描述的一样，见图 32-1。

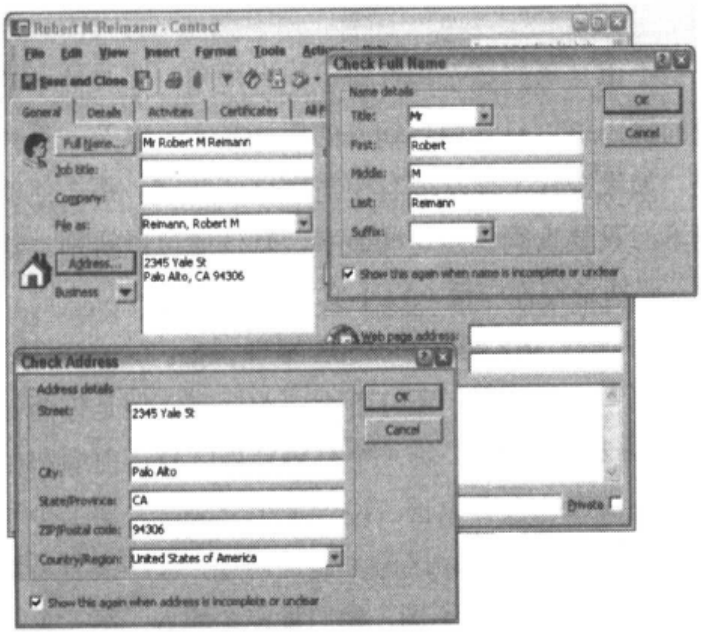


图 32-1　Outlook 中新联系地址对话框包含一些摘录控件；我们来看看主对话框的全名和地址字段。如果想检查结果，你可以单击全名或地址按钮（它们作为字段标签），就会得到如图所示的对话框。Outlook 可以分别正确地解析作者的姓名和地址——但为什么不将它们输入到一个区域中呢？

视觉控件

大多数传统图形用户界面控件都是对文本的封装。复选框、单选按钮、菜单、文本编辑、列表视图和组合框主要是文本，加了一些小的图形装饰。它们并没有利用图形用户界面媒体的全部潜能。

当多数程序为用户提供选项时，它们使用像组合框这样以文本为基础的选择控件。对于抽象功能或功能相关的文本来说这样很好，然而，如果选项可以用视觉上更清晰更吸引人的方法提供，我们就应该抛弃组合框，让用户指向并单击他所想要的图片上，而不是单独用文本描述。

图 32-2 是纯视觉界面的一个很好例子，非常适合纯视觉应用程序 Photoshop。在图层工具栏中，除了图层的名字和施加的操作以外，很难找到文本。但大部分控件都是可视的，最吸引人的是每一层都有一个缩略图。

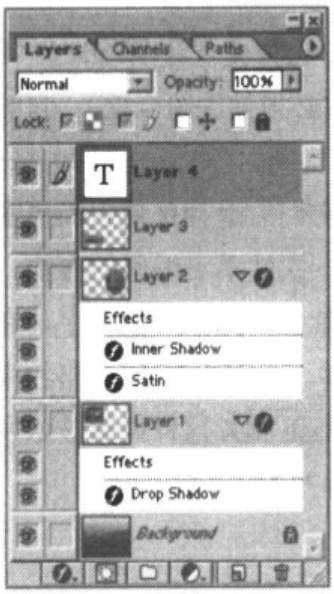


图 32-2 Photoshop 中的图层调色板是一个应用视觉控件的例子，特别是图层本身的实际缩略图可以根据图层的变化实时更新。调色板不仅可以将大量的有用功能压缩进一个很小的空间，而且控件的视觉特性也适应于该应用程序的用户基础：画家、摄影师和其他视觉方面的学者。

在前面一章中，图 31-4 显示了一个使用视觉控件的 Word 对话框。控件让用户单击代表边框的小图示进行复杂的边框选择，而不是通过名字进行选择。在这种情况下，这些控件为微软解决了边框选项数量众多的难题。可以独立地指定上边、底边、左边、右边及段落之间的边框，每一条边还可以有自己的粗细和风格。提供包含几百个像“左边细线，右边细线，上边粗线，底边细虚线”这样的组合框简直有毛病。为 5 个可能的边框分别设置一系列独立组合框的对话框，也会给用户交互带来极大的混乱。微软解决这个问题的方法非常精彩。

然而，我们不必为困难的材料节约视觉控件。在同一张图（图 31-4）的左侧是另一个视觉控件，它为 5 个选项提供简单的选择：方框、阴影、三维边框、自定义边框和无边框。这些很容易实现为单选按钮，但单击小图片更好一些。用户可以单击他希望的图片，而不必单击文本。这样做更清晰，更容易定位，更快，也更直接。

大多数软件厂商用单选按钮代替这样的视觉控件，是因为单选按钮可以从操作系统免费获得，而视觉控件不可以。如果出版商希望使用视觉控件，他们必须雇用设计师和程序员创建它们。相对于其他定制编码，这不算很昂贵，但相对于免费的基于文本的控件，仍然要贵一些。但是从长远来看，这种花费是值得的，因为它解决了可用性问题，否则会产生本土化成本，客户支持电话和所有因用户不满产生的隐性成本。

图 32-3 显示的是 Windows 95 控制面板中一个美丽的图形视觉控件。不是从文本列表中挑选时区（尽管对话框中有一个这样的列表存在），你只需在蓝蓝绿绿的地图上单击，就可选择你所在的时区。当你选择一个时区以后，地图会马上发生滑动，使你所选择的时区位于窗口的中心。微软没必要这样做，就像对一家孤傲的律师行而言，没有理由用大理石地板代替油布地板。但是当你的手从休息厅的柚木和樱桃木装饰的家具滑进柔软舒适的皮制坐椅里，就会真正体会到舒适和奢侈。出于某种原因，这种交互在 Windows 98 及以后的版本中消失了（对话框还在，但时区没有突出显示，你必须使用组合框）。它的消失是个谜，作者很想念它。

矛盾的是，控件是应用程序表面之外的独特对象，但改进它们的途径是使它们更密切地和程序视觉结构成为统一的整体。本章讨论的例子无疑是一次性手工编码的事物。这对于开发商来说是重要的机会：创建通用的视觉控件开发工具包，允许一般程序员在平均预算的基础上为他们的产品创建视觉、动画的直接操作控件。在 HTML 的严格约束下，Web 上已经有了许多视觉控件的革新。像 Macromedia 的 Flash MX 这样的新技术发展，如果得到广泛应用，将来可能会为这样的视觉控件提供一个平台。

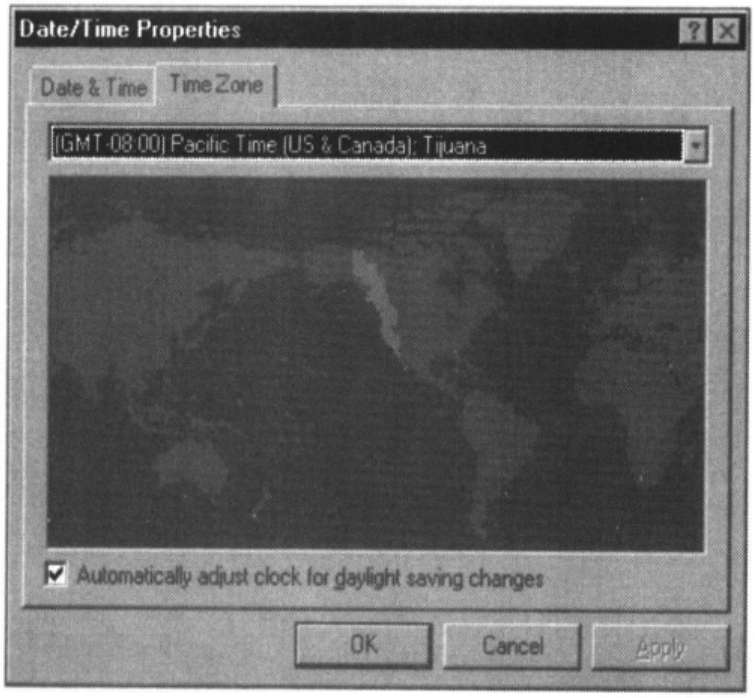


图 32-3 Windows 95 的时区对话框是一个优秀的视觉控件。它为用户提供了一种吸引人的图片选择方式。例如，如果住在美国的东海岸，你所需要做的只是单击东海岸的某个地方，地图就会平滑地移动直到美国东部位于中央，并且突出显示。动画的加速和减速如此恰到好处，几乎达到一种享受的效果，就像走进某人办公室的休息厅发现大理石、胡桃木和皮革代替了灰泥和胶合板。如果希望为你的程序添加一点美感，就让它成为有触觉的艺术品吧。为什么微软在以后的 OS 版本中取消了这种交互，依然是个谜。

第七部分

与用户的交流

33 消除错误

34 通知和确认

35 与用户的其他交流方式

36 安装过程

33

消除错误

在第30章，我们讨论了公告对话框，它是在出现问题或面临自己无法做出的决定时，由程序单方面发起的对话框。换句话说，公告对话框用于错误消息（error message），通知（notifies）和确认（confirmation），这是三种现代图形用户界面设计中滥用得最多的元素。如果设计得当，这些对话框几乎可以完全消除。本章中，我们将探讨如何消除和为什么要消除它们。

错误对话框被滥用了

在图形用户界面的世界里，可能没有比错误对话框滥用得更多的习惯用法了。程序没有权利——甚至是职责——拒绝用户的输入，这种提议被视为异端邪说，因此许多技术人员草率地抛弃了它。但是，如果我们理性地从用户角度——而不是程序员的角度考察这种断言，它不仅是可能的，而且是合理的。

用户从来不想要错误消息。用户真正希望的是避免错误造成的结果，这与他们想得到错误消息的说法有很大不同。这就好像是说人们不想摔断腿，就拒绝滑雪一样。可用性大师Donald Norman（1989）指出用户经常因为产品设计中的错误而自责。不是因为你

没听到用户的抱怨就意味着他们很高兴获得错误消息。

为什么我们有这么多的错误消息

第一代计算机数量不多，功能不强，价格昂贵，也没有为软件的灵敏性提供方便。这些机器的操作者是穿着白大褂的科学家，他们能理解 CPU 的需要，也不会因为错误消息而生气。他们知道计算机工作多么艰难，也不介意获得内核转储（core dump）、炸弹、“取消，重试，失败？”，或者是声名狼藉的“FU”（文件无法获得）信息。这就是软件对待人像对待 CPU 那样的传统。从早期的计算以来，程序员就已经接受了软件与人类交互的合适方式，那就是要求输入，而在人类没有达到 CPU 这样的专业水平时，它就抱怨。

无论什么情况下都是软件要求用户按它的方式做事，而不是软件适应人的需要，这种例子并不少见，但没有比无处不在的错误消息更普遍的了。

错误消息怎么啦

作为阻塞型模态公告板（见第 30 章），错误消息必须用模态对话框停止进度。多数用户界面设计师——同时也身为程序员——认为他们的错误消息框是在提醒用户一些严重的问题。这是一种普遍存在的错误认识。其实大多数错误消息框是在告诉用户程序无法灵活工作，通过图 30-6 这个例子你就可以明白。就用户看来，大多数错误消息框是在向程序承认自己的愚蠢。换句话说，对于大多数用户来说，错误信息框不仅停止了程序的进度，而且还公然违背了第 10 章中的公理：不要用愚蠢的行为停止进度。我们可以通过消除错误信息框来大幅度改善界面的质量。

设计技巧：错误信息框停止进度，简直是白痴。

人们讨厌错误消息

人有情绪和感觉：计算机没有。当一个代码块拒绝另一个代码块的输入时，发送代码块没有感觉；它不会皱眉，不会受到伤害，或者求助于心理咨询。人则不同，当他们平白无故被告知是傻瓜时，会很生气。

当用户看到一个错误信息框，就好像是另一个人说他们很笨。用户讨厌这样（见图33-1）。多数程序员不顾用户的必然反应，只是耸耸肩，然后继续以各种方式设置错误信息框。除此之外，他们不知道如何创建可靠的软件。

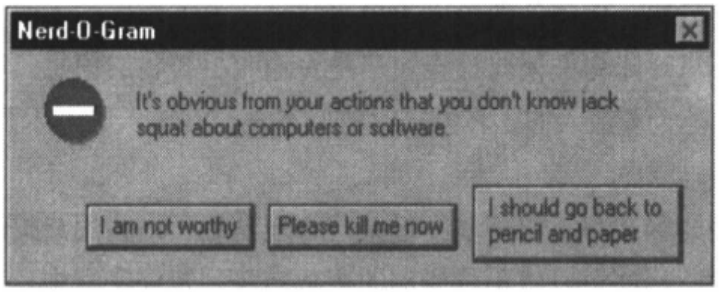


图 33-1 无论你的错误信息框如何用词得当，这就是用户的解释方式。

许多程序员和用户界面设计师误认为当人们错了的时候，喜欢或需要别人告诉他们。这种推断是错误的，理由有以下几点。这种认为人们犯错的时候喜欢知道的假设首先忽视了人的本性。许多人得知自己错了的时候会非常沮丧，宁愿不知道自己做错了。许多人不愿意从别人那里得知自己犯了错误，另一些人则只愿意从另一半或亲密的朋友那里得知，而绝少有人会希望由一个机器告诉他。你可以否认，但这是事实，用户会在责备自己之前，指责通知者。

“用户需要知道他们犯了错误”这一假设也是不对的。知道请求了无效的字体大小会有多重要？大多数程序员可以找到更合理的替代方式。

我们认为当别人失礼时，告诉他这些是很不礼貌的。告诉某些人有菜叶粘在他的牙齿上，或者纽扣松了，都是很让人尴尬的事。聪明的人会设法引起对方注意，而不惊动其他人。但程序员却认为用屏幕正中央巨大而醒目的对话框停止一切活动，并且发出明显的“嘟嘟”声是合适的。

到底是谁的错

传统的看法认为错误信息框的作用是在用户错了的时候通知他们。实际上，大多数错误公告的作用是在程序陷入混乱的时候向用户报告。用户所犯的错误比想像的要少得多。典型的“错误”包括用户输入了一个界限以外的数字，或者在计算机不允许的地方输入了一个空格。当用户输入了一些按计算机的标准不能理解的数据时，这到底是谁的错？是用户因为不知道到底该如何正确使用程序的错，还是程序没有做出选择和说明的错？

信息以不熟悉的顺序输入常被软件认为是个错误，但人们处理这种不熟悉的顺序没有困难。人类知道如何等待时机，直到结束。软件则会草率地下结论，认为违反顺序就

是错误的输入，并且发起讨厌的错误信息框。

例如，当用户创建一张没有客户号码的发货单时，多数程序会拒绝输入。它们愚蠢地停下来，要求用户必须立刻输入正确的客户号码。另一种方式是：程序可以先接受这个任务，期望最终会输入有效的客户号码。例如，他可以为用户建立一个特别提示表明缺少了什么。然后程序在交易结束或者月账簿关闭时检查用户是否输入了必要的信息，确保用户输入了有效的客户号码。这是大多数人的工作方式。他们不是经常输入了“坏”编码，而是软件不接受他们输入的编码顺序。

如果用户忘记了向计算机完整地解释某事，它可以经过某些合理延迟后，为用户提供持续的信号。在一天或一周结束时，程序可以将不能协调的事务转移到暂停账目中。程序不必用错误信息框阻止处理的进行，毕竟，程序会记住这个事务，这样它们可以进行跟踪和修正。这是手工系统的工作方式，那么为什么计算机系统就不能至少做到这样呢？为什么停止整个过程，仅仅是因为遗漏了某些东西？只要用户能够获得账目还需要整理的通知，就不应该有什么问题。诀窍在于通知用户而不需要停止整个过程。我们将在第 34 章中进一步讨论这个问题。

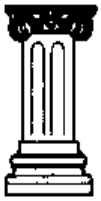
如果程序是一位人类助手，因为我们递交了一份不完整的表格就在会计部门静坐示威，我们会非常不高兴。如果我们是老板，就会考虑解雇这种小题大做、固执伪善的小职员。我们会说，只需要拿起表格指出缺少的信息就可以了。作者曾经使用过 Rolodex 程序，即使已经输入了个人地址，它还要求为电话号码输入区号。合理地猜测一个区号并不需要多高的智力。如果你输入了一个新名字，他的住址在 Menlo Park，程序完全可以通过查阅数据库，发现其他住在 Menlo Park 的 25 个人，他们的区号是 650，而假定他的区号是 650。诚然，如果你输入一个新的地址，比如说 Boise Idaho，程序可能会被难住。但是访问一下网络上的目录，或者创建一个美国 1000 个大城市的区号列表又有何难呢？

程序员可能会反驳说：“程序可能会出错。它不能百分之百肯定，一些城市有多个区号。没有经过用户同意它不能胡乱猜测！”不是这样的。

如果我们请一名人类助手向 Rolodex 输入一个客户的电话联系信息，没有提到区号，他无论如何也会接受，并且希望在信息缺失变得关键时能找到它。他可能会到地址目录中查找。比如说客户住在洛杉矶，因此地址目录不确定：区号可能是 213，也可能是 310。如果这名助手冲进办公室焦急地嚷道“停下你手中的活！这个客户区号模棱两可！”我们会非常想解雇他，而去雇用一个 IQ 比室温稍微高一点的人。为什么软件就应该有不同呢？在这种情况下，人可能会在区号字段里写上“213/310？”。下次我们给那位客户打电话的时候，需要再确定一下该区号是否正确，但与此同时，生活可以继续。

又可能有这样的激烈抗议：“区号字段只能容下三个数字！我没法将‘213/310？’

填进去!”哟，那真是太糟了。你的意思是说根据程序底层的实现模型——严格固定的字段宽度——显示的用户界面，迫使你拒绝自然的人类行为来迎合计算机那令人厌恶，顽固不化的错误信息框。坦率地说，错误信息框来自程序不合理的行为，而不是用户的任何错误。



公理：用户界面不应该是肤浅的。

这个例子阐述了用户界面设计另一个重要的观察成果。它不是肤浅的。在设计中未能解决的问题越过了系统放在用户面前。有很多不同的方式处理软件交互中产生的异常情况——一个具有创造精神的设计师或程序员可能会绞尽脑汁想出半打情况——而多数程序员甚至想都没想过。他们为自己的进度表和偏好所累，因此倾向于根据完美的 CPU 行为而不是不完美的人类行为来构思这个世界。

错误消息不起作用

对于错误信息框，它最后的讽刺是：*它们不能阻止用户犯错误*。在我们的想像中，用户可以脱离麻烦，因为我们相信错误信息框可以指引他们，但这是一个错觉。错误信息框真正所做的只不过是保护程序不陷入麻烦。在大多数软件中，错误信息框像哨兵一样在程序敏感的地方放哨，而不是在用户最危险的地方放哨。它有一种根深蒂固的信念，那就是程序比用户重要。其实无论错误信息的数量和质量如何，用户在使用软件的过程中都会有大量的麻烦，一个错误信息框所能做的就是防止用户不向数字段中输入字母，它与防止用户不输入错误数字无关，这是更难的设计任务。

消除错误消息

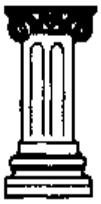
我们不能通过简单地抛弃那些显示实际错误对话框的代码，然后一旦出现问题，就让程序崩溃来消除错误消息。我们必须重新编程使它们不再容易出问题，我们必须用更友好、更礼貌、更茁壮的软件取代错误消息，它防止错误产生，而不是让程序在事情与其预料的不同时，只会发牢骚。就像是预防接种抵御疾病的疫苗，我们使程序对问题具有免疫力，然后我们就可以抛弃报告问题的信息。为了消除错误信息，我们必须首先消

除用户犯错误的可能性。不能认为错误信息是正常的，我们必须把它视为解决罕见问题的异常方法——就像外科手术之于阿司匹林。我们必须把它当做一种类似于最后一招的习惯用法。

每个优秀的程序员都知道，如果模块 A 向模块 B 发送了无效数据，模块 B 应该立刻用一个合适的错误指示器拒绝输入。在模块接口的设计上，不这样做将是个巨大的错误。但人类用户不是代码模块，软件不仅不应该用错误信息框拒绝输入，而且软件设计者必须重新评估“无效数据”的概念。当它来自人类时，软件必须认为输入是正确的，原因很简单，那就是人类比代码更重要。软件不但不能拒绝输入，而且必须努力去理解和调整输入造成的混乱。程序能够理解计算机内部事物的状态，而人类只能理解真实世界的事物状态。从根本上说，真实世界比计算机所思考的要更有意义，也更重要。

使错误不可能

使用户不可能犯错是消除错误消息的最好方法。通过为所有的数据输入使用有界控件，可以防止用户输入错误的数字。与迫使用户键入他的选择不同，让用户从包含可能选项的列表中选择。例如与其让用户输入州编码，不如让用户从有效的州编码列表或者地图中选择。换句话说，使用户不可能输入错误的州。



公理：尽可能使错误不可能出现。

另一个消除错误信息框的好方法是让程序变得足够聪明，使用户不再提出不必要的请求。许多错误信息框显示：“无效的输入。用户必须输入 xxxx。”既然程序知道用户必须输入什么，它为什么不自己直接输入 xxxx ，避免对用户的斥责呢？让程序记住过去访问过的文件，允许用户在列表中选择，取代用户在磁盘中寻找文件的要求，因为那样存在用户选择错误文件的可能。另一个例子是设计从内部时钟获取数据的系统，用来取代请求用户输入。

毫无疑问，所有这些解决方法都会加大程序员的工作量。但是，程序员的工作是为了满足用户要求，而不是相反。如果程序员把用户仅仅当做是另一种输入设备，那么在软件设计中就很容易忘记正确的主次关系。

计算机用户不会同情程序员的困难。他们不会看到错误信息框背后的技术原理，他

们所看到的是程序不愿以人类的方式处理事物。他们将所有的错误信息框都当做图 33-2 所示的对话框变体。

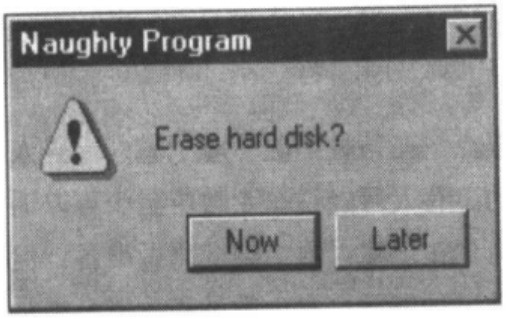


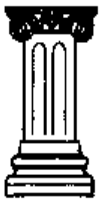
图 33-2 这是大多数用户理解错误信息对话框的方式。用户把这当做卡夫卡式的审问，每个后续的选择都会导致惩罚和悔恨。

错误信息框的一个问题在于它经常是事后诸葛亮。它们显示："坏事发生了，你所能做的就是承认灾难的降临"。这样的报告没有任何帮助，而且这些对话框还总是带有一个"OK"按钮，要求用户成为该错误的同谋。这些错误信息框让人联想到老战争片中的情景：一个倒霉的士兵在经过稻田的时候踩上了地雷，他和他的战友清楚地听到地雷触发机制的滴答声，士兵意识到尽管现在是安全的，但一旦脚离开，地雷就会爆炸，带走他一大部分有用的躯体。用户在看到多数错误对话框的时候都有这种感觉。他们多么希望自己在千里之外，回到真实世界中来。

正面反馈

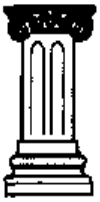
软件难以学习的原因之一在于正面反馈太少。人们从正面反馈中学到的知识比从负面反馈中学到的更多。人们希望正确有效地使用他们的软件，这激励着他们学习如何使软件更好为他们工作。他们需要的不是在失败时的象征性惩罚，而是在成功时受到奖励或至少得到认可。如果他们得到称赞，他们会感觉良好，这种好心情会反映到他们的成果中。

负面反馈的鼓吹者可能举出许多负面反馈有效引导人们行为的例子。这些证据是真的，但是几乎无一例外，有效惩罚反馈的场景是防止用户做他们想做但不应该做的事：例如不应该以超过时速 55 英里的速度行驶，不要欺骗另一半，不要逃避所得税。但是回到帮助人们做他们想做的事时，正面反馈是最好的。想像一下，如果你请的滑雪教练冲你大喊大叫，或者一个餐馆老板当着其他客人的面大声宣布你的信用卡拒付了。



公理：当软件告诉用户失败时，用户会觉得很没面子。

记住我们正在从计算机的角度谈论负面反馈的缺点。在某种情况下对于另外一些人来说，尽管负面反馈使人不愉快，但也可能是正当的。有人可能说军士教员至少可以训练你在战斗中如何保存生命，专横的教授至少可以为你在现实世界中的变化做准备。但是软件产生的负面反馈——无论是什么软件——都是一种侮辱。军士教员和教授至少是人，他们有真实的体验和优点。但软件告诉你失败会让人觉得没有面子。用户有充分的理由讨厌侮辱和贬低。计算机内部发生的任何事情都没有重要到可以羞辱和贬低人类用户的地步。我们只是出于习惯采取负面反馈。



公理：计算机内部没有什么危急时刻值得羞辱人类。

有特例吗

随着我们的技术力量不断增强，计算机硬件的便携性和灵活性也不断增加。现代计算机能在不断电的情况下连接或断开网络和外围设备，这意味着硬件的增加或消失都是正常的，打印机、调制解调器和文件服务器可以像潮水一样来去自由。随着 WiFi 和蓝牙（Blue tooth）等无线网络的发展，我们计算机与网络的连接和断开可以很经常，很容易，而且，很快也会变得很透明。如果你在打印一个文档，却发现没有连接打印机，这是个错误吗？如果你编辑的文件，它所在的驱动器突然不能访问，这是个错误吗？如果你的调制解调器不再插在计算机上，这是个错误吗？

我们涉足互联网越深，变化无常的难题就变得越普遍。很容易把互联网想像成一个无限的硬盘——它不受任何人、任何公司或系统管理员的控制。今天有效的打印机明天就便成无意义的了。这是错误吗？

这些现象没有一个应该看做错误。如果试图打印某件东西，而又没有可用的打印机，你的程序应该将打印信息存储在磁盘中，当打印机重新联机后，打印管理程序应该安静地指示队列中有未打印的文档。这应该是正常的日常计算技术特征，而不是一个错误。

对于文件来说也是同样的道理。如果你在服务器上打开一个文件，开始编辑，然后带着笔记本去餐馆进午餐，程序应该明白文件正常的宿主地不再可用，它必须做些更聪明的事。它可以用内建的 WiFi 卡登录到远程服务器上，或者只是在本地保存一些改动，当你吃完午餐回到办公室，再和服务器上的版本同步。无论如何这是正常的行为，不是错误。

几乎所有的错误信息框都可以消除。如果你从“错误信息框必须消除，以及所有其他的事也得因为这个目标而改变”的角度审视这种情况，你会发现这种判断完全正确。你还会惊讶地发现，原来实现这个目的只需要改变那么少。在程序其他部分需要很大改动的罕见情况下，可以与真实世界折中使用错误信息框。但是程序员们需要开始把这种折中当做程序本身对失败的承认：它已经求助于不正当和暗算的行为，就像代码中的 GOTO 语句。

尽管如此，如果打印机着火了，我们会乐意在屏幕上看到错误消息框。错误提示应该留到这种真正紧急的情况下才使用。

改进错误消息：最后一招

现在我们讨论一些改进错误消息框质量的方法，如果确实需要使用它们的话。使用这些建议只在你用尽所有其他选择之后，作为**最后一招**。

一个组织得很好的错误消息框应该满足下列要求：

- 有礼貌。
- 具有启发性。
- 有帮助。

千万不要忘记，错误消息框是程序报告自己所做工作的失误，而且它正在妨碍用户做事。错误消息框必须永远都是礼貌的，它甚至不能暗示是用户产生的问题，因为从用户的观点看那不是真的。客户永远是对的。

用户确实输入了一些愚蠢的数据，但是程序没有资格争吵和指责。它们最应该做的就是向用户提供他们所要求的，无论多么愚蠢。最重要的是，当程序在用户最终发现他自己错误的时候不能说：“嗯，看你做的蠢事，现在不能恢复。太糟了。”。即使是在用户采取不恰当的行为时，保护用户也是程序的责任。这似乎是非常苛刻的，但是防止程序采取不恰当的行为理所当然不是用户的职责。

错误消息框必须为用户说明问题。这意味着它必须向用户提供做出一个合适的决定来解决问题所需要的信息。它必须澄清问题的范围，可选择的方法是什么，默认的情况下程序会做什么，如果有信息丢失了，也要告之丢失了什么信息。程序应该把这当做一

种忏悔，告诉用户所有的事。

但是如果程序只是将问题堆在用户面前，撒手不管是不对的。在错误消息框中最少应该直接提供一种正确的建议方法，应该提供以不同的方式处理问题的按钮。如果打印机不存在，消息框应该提供延期打印输出的选项或选择另外的打印机。如果数据库已经删除不能使用，它应该重建数据库并让它回到工作状态，包括告诉用户该过程要花费多少时间，以及造成的负面影响。

图 33-3 展示了一个合理的错误消息框例子。注意它是礼貌、启发和有帮助的。它甚至没有暗示用户的行为有任何缺陷。

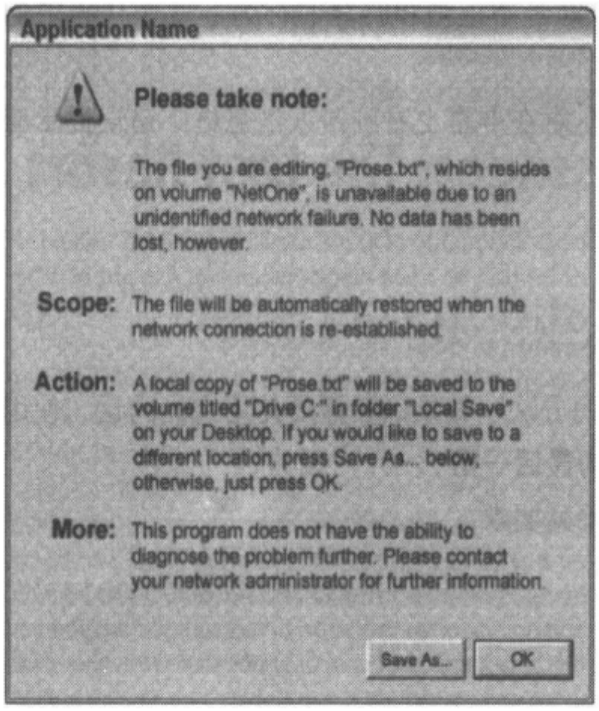


图 33-3 正如你没有充分的理由在编码中使用 GOTO 语句一样，发起错误消息框也没有任何充分的理由。但是和程序员偶尔会妥协使用一两个方便的 GOTO 语句一样，他们偶尔也会发起一个错误消息框。但是，在这种情况下，你的错误消息框的外观应该和该图相似。它礼貌地为用户说明问题，为用户提供帮助好让它从困境中解脱。这个错误公告有四个部分（分别用“请注意”、“范围”、“动作”和“更多”标记）帮助用户清楚地理解可用的选择，以及他为什么要做出每个选择。程序足够聪明，不至于因为卷不可用而丢失文件。对话框还通过“另存为…”按钮提供另一种可选的办法。

错误的结束

错误消息框确认了计算机是正确与否的最终裁决者，用户只是服务于它的数字化王权的思想。这个态度影响了程序员，也影响了用户。它怂恿程序员在设计中做出错误的判断，并在实现中走捷径。这些折中要求使用更多错误消息，同时用户也被错误消息麻痹，以致看不到**无错计算**的可能。

34

通知和确认

前一章我们讨论了错误对话框。在本章中，我们讨论提示对话框（alert dialog）（也称为通知器）和确认对话框（confirmation dialog），还有它们交互的结构，底层的假设，以及在大多数情况下如何消除它们。

提示和确认

和错误对话框一样，提示和确认也愚蠢地停止进度，但是它们不报告故障。提示通知用户程序的行为，而确认给用户赋予一种忽略该行为的权力。这些对话框在大多数程序中像杂草一样丛生，应该像错误对话框一样，使用更有用的习惯用法消除它们。

提示：宣布显而易见的内容

当程序感觉不舒服要行使权威时，会采取措施通知用户它的动作。这就称为提示。提示违反了公理：对话框是另一个房间，去之前要有个好理由（见第 25 章）。即使提示是正当的（它极少正当），为什么要到另一个房间去做呢？如果程序采取了一些不可原谅

的行为，它应该在行为发生地点坦白，而不是在一个单独的对话框里。

概念上，程序应该有勇气确信自己，或者在没有用户的直接引导下不应该采取行动。例如，如果程序把用户的文件自动保存到磁盘上，它应该自信地认为自己做对了。它应该让用户了解程序做了什么，但是它不必愚蠢地停止过程的进行。如果程序不能确认它是否应该保存文件，它就不应该保存文件，将该操作留给用户处理。

相反，如果用户指示程序做某事——例如拖动文件到垃圾箱——它不需要为了告诉用户已经将文件拖到垃圾箱而愚蠢地停下来。程序应该确保这个行为有足够的视觉反馈；如果用户确实做了错误的动作，程序应该悄悄地为用户提供强大的撤销工具，这样用户可以全身而退。

提示的原理在于它们通知用户。这是令人满意的目标，但不能以打断流畅的交互流作为代价。

图 34-1 中所示的提示对话框是提示带来麻烦而不是帮助的例子。查找对话框（下面的那个）迫使用户完成搜索时单击取消按钮，而重叠的提示对话框使它成为了打断流的按钮的藏身之地：首先是提示对话框中的“OK”按钮，随后是查找对话框中的取消按钮。如果把提示信息建立在主查找对话框中，用户的负担会减少一半。这对用户界面设计师来说很经济。

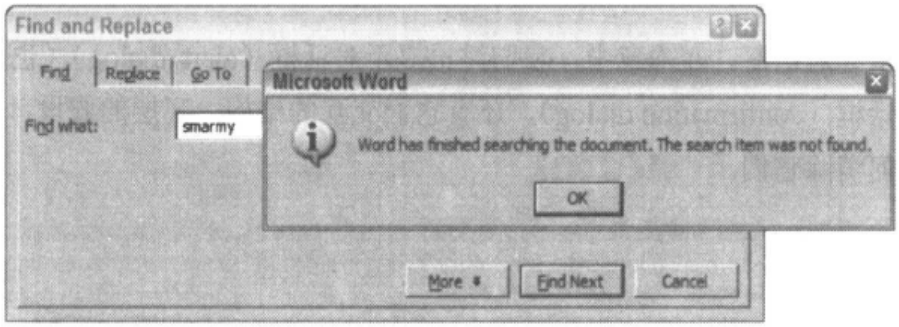


图 34-1　这是一个典型的提示对话框。它不仅不必要、不恰当，而且愚蠢地打断了进度。Word 的查找对话框已经完成了搜寻文档的任务。报告事实是 Word 的另一个功能吗？如果不是，那为什么要用不同的对话框呢？就好像必须到起居室里用叉，到厨房用汤勺一样。图中的“i”图标是笨拙界面设计的一个真正警告。的确，软件必须有效不断地向用户报告它的状态，但是停止进度的提示对话框不是恰当的机制。

因为提示对话框容易创建，所以它们数量众多。很多语言用一行代码就可以提供数种信息框工具。相反，在程序界面上建立动画状态显示需要上千行的代码。在这种情况下不能指望程序员做出正确的选择。因为他们有利益的冲突，所以设计师必须精确地指定信息在应用程序界面上的报告位置。然后设计师必须穷追不舍，确保设计不与快速编

码妥协。想像一下，如果建筑工地上的承包人单方面决定不加浴室，因为水管装置处理起来太麻烦，那后果会怎么样呢？

软件需要一直告诉用户它的行为。它应该在主屏幕上有可视化的指示器，使用户可以获得这种状态信息，用户期望这样。启动提示对话框来宣布一个没有请求的动作已经够糟的了，为一个已经请求的动作再发起一个提示信息就更加荒谬。

软件需要灵活和宽容，但它不必阿谀奉承。图 34-2 是一个经典的“提示对话框应该消除以结束我们的痛苦”的例子。它宣称条目已经添加到电话簿。这个对话框在我们告诉它添加条目到电话簿后立刻出现，它发生在我们向电话簿物理添加条目的几毫秒内。为了宣布这么显而易见的事，它停止了进度。

这好像是程序希望它艰苦的工作获得赞许：“瞧，亲爱的，我已经为你清扫了房间。你不爱我吗？”如果一个人像这样与我们交往，我们会建议他去做心理咨询。

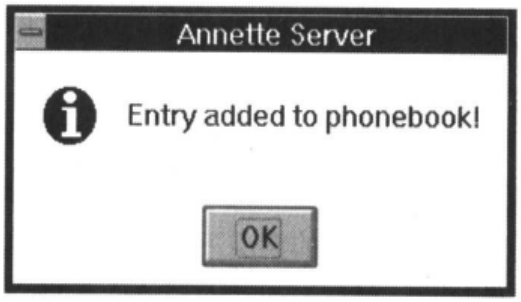


图 34-2 选自 Delrina WinFax Lite 的这个对话框是一种不必要的谄媚。我们为电话簿添加条目，却马上被这个重要的信息框挡住了路。我们真的需要程序浪费我们的时间来详细描述这个显而易见的事实吗？

确认

当程序对自己的行为不自信时，它经常用对话框询问用户来征求许可。如图 34-3 所示的对话框被称为确认对话框。有时程序是在对用户的行为做事后的劝告而提供确认。有时程序觉得没有能力做决定，而用确认对话框让用户替它选择。确认对话框总是来自程序，而不是用户。这意味着它们是实现模型的反映，而不是用户目标的代表。

正如我们在第 2 章中讨论的那样，向用户显示实现模型一定会得到不好的用户界面。这意味着确认信息是不恰当的。当程序员在编码陷入僵局时，就在软件中创建一个确认对话框。通常，他意识到将引导程序采取某种冒险动作，并对要为此承担责任而感到没有自信。有时冒险动作基于程序发现的一些情况，但更多的时候基于用户发起的命令。举一个典型的例子，在用户发起一个不可恢复或者结果将导致异常警报的命令后，就会启动一个确认对话框。

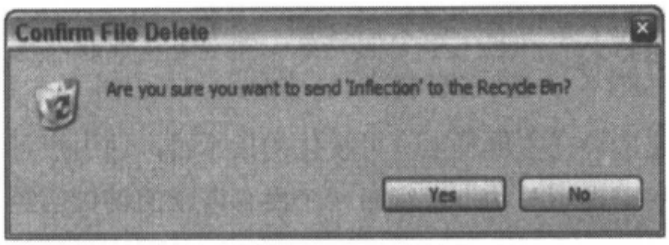


图 34-3 每次我们删除 Windows 文件时，都会见到询问我们是否确定的确认对话框。是的，我们确定，我们总是表示可以确定。如果错了，我们希望 Windows 能够为我们恢复文件。Windows 用它的回收站满足了我们的愿望。既然这样，它为什么还要发起确认信息？当例行公事地发起确认对话框的时候，用户也习惯性地批准。所以当它最终向用户报告一个即将发生的灾难时，他也会出于惯例继续批准它。为你的用户做点好事，不要再创建另外的确认对话框了。

确认对话框是把责任推卸给用户。用户信任程序的工作，程序不仅应该做，而且要确保做对。恰当的解决方法是使动作易于恢复，以及提供足够的非模态反馈，确保用户没有脱离保护。

在开发过程中，随着程序代码的增加，程序员会发现大量他们感到不能充分解决问题的地方。程序员将不知不觉地单方面地在这些位置插入推卸责任的代码。这种趋势应当得到密切关注，因为程序员甚至在用户界面规范已经达成一致的情况下，还在代码中插入对话框。程序员经常不把确认对话框当做用户界面的一部分，但是它们确实是用户界面的一部分。

【对话框在喊“狼来了!”】

确认对话框显示了人类行为的一个有趣怪癖：它们只在意想不到的时候起作用。听起来可能会不以为然，除非你在具体的上下文中体会到它。如果在常规的地方提供确认对话框，用户很快就会习以为常，看也不看就习惯性地打发它。因此关闭确认对话框和发起确认对话框一样成为例行公事。如果在某一时刻，一个真正意外的危险情况发生——一个应该引起用户注意的情况——他会因为已经形成的习惯机械性地关闭确认对话框。就像寓言故事里大叫“狼来了”的男孩，当最后真正有危险的时候，确认对话框不能起作用，因为它在没有危险的时候叫了太多次。

对于确认对话框来说，它们通常只能在用户明确单击了“No”或“Cancel”按钮后才会出现。当用户可能单击“Yes”或“OK”按钮时，永远不应该出现。从这个角度看，它们看上去相当没有意义，对吗？

图 34-4 所示的确认对话框是一个经典的例子。图中确认对话框的讽刺意味在于它很难决定删除或保留哪些样式。如果确认对话框在我们试图删除当前使用的样式时出现，

它至少还有些帮助，因为这种确认显示了更少的例行公事。但是为什么不在正在使用的样式名边上加一个图标来取代确认对话框呢？那样界面可以提供更中肯的状态信息，用户也可以做出更全面的删除决定。

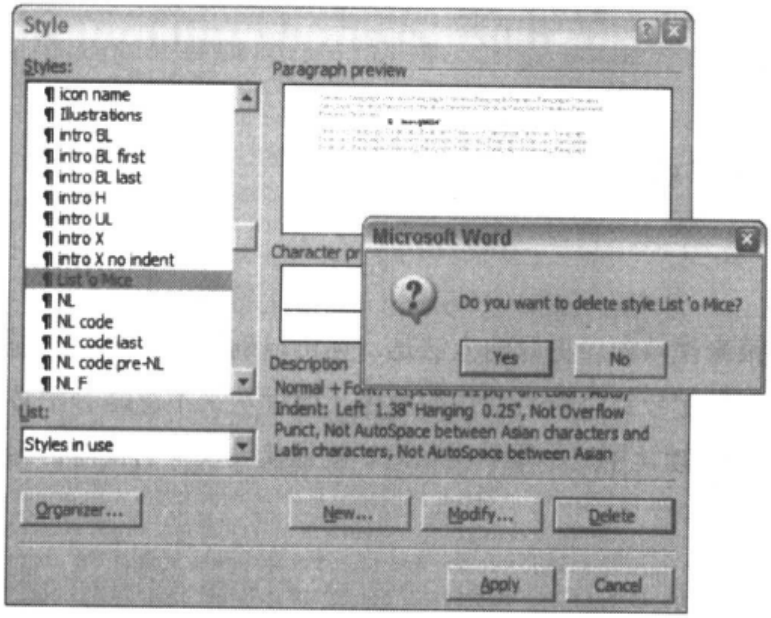
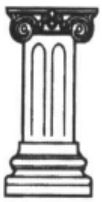


图 34-4 如果你在 Word 的样式对话框中单击“Delete”按钮，就会见到这个确认对话框。多数人会机械地单击“Yes”按钮，而不管是否是他的本意。在样式对话框中，你偶尔可能不经意地删除了一个真正想保留的样式。但是，这种确认并不能起到保护作用，它也不允许恢复。如果它只是在某些条件下出现而不是简单地请求删除时，那么它还有一丝有用的机会。请保证不要再创建这样的对话框，好吗？

【消除确认对话框】

有三个公理告诉我们如何消除确认对话框。最好的方式是遵循简单的格言：做，不要问。当你设计软件时，勇往直前给它确信的力量（如第 4 章中讨论的那样，通过用户研究来支持）。用户会敬佩它的简短和自信。



公理：做，不要问。

当然，如果程序自信地做了用户不乐意的事，必须有能力执行撤销操作。程序行为的每个方面都必须是可撤销的。在那些罕见的情况下，当程序行为不合时宜时，让用户发起停止并撤销（stop-and-undo）的命令，取代预先用确认对话框提问的方式。

当前我们认为不能用撤销保护的大部分情况实际上可以保护得相当好。删除或覆盖文件就是个很好的例子。文件可以移动到暂停目录，在那里保留一个月或者保留到你物理删除之前。Windows 的回收站就采用了这种策略，除了一个月后自动清除的文件部分：用户仍然必须手工清除垃圾。

公理：让所有的动作都可以撤销。

比草率行动迫使用户使用撤销来挽救程序更好的方法是，你可以确保程序为用户提供了足够的信息，以致它不会故意发起导致不恰当动作的命令（或者从来不忽略一个必要的命令）。程序应该用丰富的视觉反馈让用户不断获得信息，就像仪表板上的仪器告诉我们汽车的状态一样。

公理：提供非模态反馈来帮助避免用户犯错误。

有时会出现撤销也不能真正起到保护作用的情形。这是不是使用确认对话框的合理场合？不必。一个更好的方法是为用户提供类似高速公路的保护方式：用一致而清晰的标志。你可以在界面上建立卓越的非模态警示标志。比方说，图 34-5 来自 Adobe Photoshop 的对话框，告诉我们文档比可用的打印区域大。为什么程序要等到现在才告诉我们这个事实呢？如果在页面上显示真实可打印区域的向导任何时候都可以看得到（除非用户把它们藏了起来），那又会怎样呢？如果当用户的鼠标滑过工具栏上的打印按钮时，将那些可打印区域以外部分的图片突出显示，那又会怎样呢？很明显，非模态反馈（见下一节）是解决这些问题的最好方式。

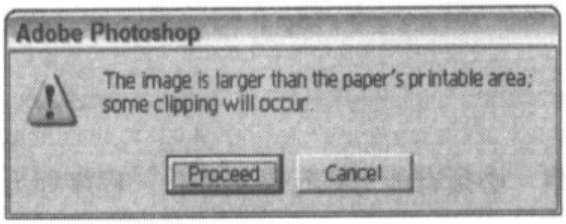


图 34-5　这个对话框提供的帮助太小，也太迟了。如果程序可以在主界面中用虚线向导直接显示可打印区域会怎样呢？没有理由让用户忍受这样的对话框。

比实在不能撤销的动作更常见的是：那些易于撤销，但常规确认对话框不能保护的动作。图 34-3 中的确认对话框是极好的例子。没有任何理由要确认文件到回收站的移动。回收站存在的惟一理由就是实现文件删除的撤销工具。

取代对话框：丰富的非模态反馈

现在家用和办公用计算机多数都有高分辨率显示器和高质量的音频系统，却很少有程序（游戏除外）甚至是粗浅地使用这些工具为用户提供有用信息，让用户大体上了解程序的状态，用户的任务，系统和它的外围设备。这就好像有一个完整的工具箱可用于向用户传递信息，而程序员仍然坚持像以前那样用粗糙的工具——对话框——来交流。不用说，这种方式根本无法与用户交流精细的状态信息，因为甚至最无能的设计师也知道你不希望不断弹出对话框，但持续的反馈是用户确实需要的，只是交流的渠道需要有所改变。

在该节我们将讨论丰富的非模态反馈，可以在程序的主显示区向用户提供信息。它不必停止程序流或用户流，几乎可以消除那些讨厌的对话框。

丰富的视觉非模态反馈

也许最重要的非模态反馈类型是**丰富的视觉非模态反馈**（Rich Visual Modeless Feedback，RVMF）。这种类型的反馈在提供当前应用的对象或过程的状态和属性的深入信息方面非常丰富。因为它惯用屏幕的像素（经常是动态的），所以它是视觉的。它是非模态的，因为这种信息总是预先显示，用户不需要特别的动作或模态变换来浏览和理解这种反馈。

例如，在 Windows 2000 或 XP 中，单击文件管理器窗口中的一个对象，会在文件管理器窗口的左侧自动显示对象的详细信息（在 XP 中，微软有一些退步，它将信息放在各种其他命令和链接的底部。另外，它默认地将详细信息区域实现为需要打开的抽屉，虽然程序至少记住了它的状态）。信息包括标题、文档类型、大小、作者、修改日期，如果它是一个图像或媒体对象，甚至会有缩略图或微型播放器。如果对象是磁盘，它显示一个饼形图和图例来描述磁盘已用了多少空间。确实非常方便！这个交互也许稍微带些模态的色彩，因为它需要选择对象，但是无论如何用户都是需要选择对象的。这种方便的功能排除了显示信息所需的属性对话框。尽管这种信息多数是文本，它仍然适用于这种习惯用法。图 34-6 是来自 Cooper 设计项目中的一个例子。

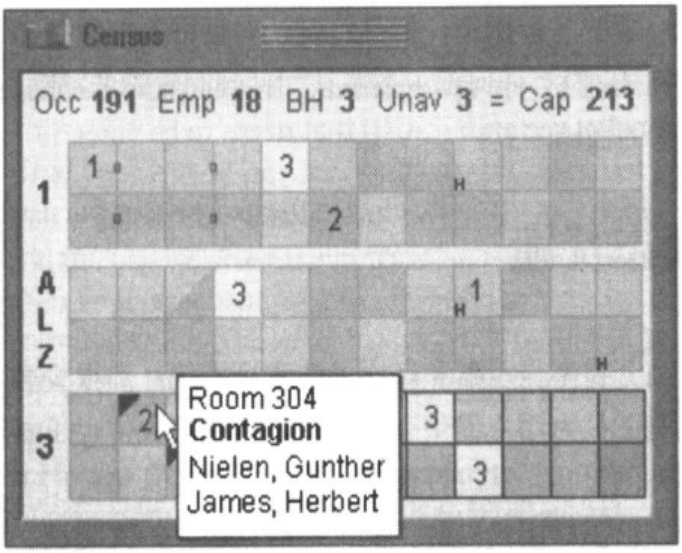


图 34-6 这个面板是 Cooper 设计的一个长期健康信息系统。它是丰富视觉非模态反馈的一个好例子。图表代表设施里的所有房间。颜色编码指示男性、女性、空置或男女混住的房间；数字表示空置的床位；房间之间的小框代表共享的浴室。黑色三角形表示健康问题，小的“H”意为保留的床。工具提示代替丰富视觉非模态反馈显示房间号，房客名字，以及突出显示任何有关房间和房客的重要注意信息。房间、床位和雇员的数值概要位于上部。这种显示有一个短时间的学习过程。一旦熟练掌握，它让护士和设施管理者对设施状态一目了然。

还有另外一个例子，来自 Mac：当你从因特网下载一个文件时，下载的文件作为一个图标出现在桌面上，用一个小的动态更新进度栏在视觉上表明下载的百分比。

丰富视觉非模态反馈的最后一个例子来自计算机游戏世界：Sid Meier's Civilization（席德 梅尔的《文明》）。这个游戏在主界面中提供了许多丰富视觉非模态反馈的例子。它描述的是一幅历史世界的图景，你作为进化的文明社会领导人，可以进行建设和征服。Sid Meier's Civilization 用丰富视觉非模态反馈象征城市的 6 种事物，如果城市更先进，它的结构就会更现代。如果它更大，图标就会更大更富有装饰性。如果有城市暴乱，就会有烟雾从城市升起。人群和市民团体也有视觉化地显示状态，通过小计量器的方式表明团体的健康和力量。甚至风景也有视觉非模态反馈：当团体移动和城市壮大时虚线标记影响范围的变动。地形也是变化的，比如铺设道路，清除森林和大山开矿。尽管游戏中有对话框存在，但大部分理解发生情况所需要的信息无须任何语言或对话框交流。

想像一下这种情形，如果所有对象在你的桌面上都有相关信息，或者你的程序能够以这种方式显示它们的状态。打印机图标可以显示它们将多久完成打印工作。磁盘和可移动的媒体图标可以显示它们已经使用的空间。当选择了一个对象然后拖放，所有可以接收它的位置会从视觉上突出显示，宣布它们的接纳能力。

想想你程序中的对象，它们有多少特性——特别是动态变化的特性——对于你的用户来说哪种状态信息是重要的。找到一种表达方法。用户注意到和学会这个表示方法后，事情的进度状态可以让他一目了然（如果用户需要，还应该有获得完整细节信息的途径）。利用丰富视觉非模态反馈形式把信息放在你的主显示屏，试试看你可以消除多少例行公事的对话框！

创造丰富非模态视觉反馈必须注意的一个要点是：它不是为新手用户创建的。即使你添加了工具提示用文字描述任何一个视觉暗示的细节（这是你应该做的），它也需要用户首先发现它，然后体会它的含义。丰富视觉非模态反馈是用户可以实时发现的东西，当他们发现后，会感到如此奇妙，但是，与此同时他们也需要菜单和对话框的支持来寻找需要的信息。这意味着取代提示（alert）和严重问题警告（warning）的丰富视觉非模态反馈对用户来说必须特别清晰，必须确保这种状态在视觉上比次要的提供信息的丰富视觉非模态反馈更突出强调。

听觉反馈

在数据输入环境里，客户在计算机屏幕面前输入数据一坐就是几个小时。这些用户很可能眼睛盯着原始文档，按指法打印而不看屏幕。如果输入了一些错误数据，他需要通过听觉和视觉两种反馈获得信息。客户在视线停留在原始文档的同时，可以用听觉监测他的输入是否成功。

我们建议的这种听觉反馈不要和错误信息框发出的嘟嘟声相同。事实上，它根本不是嘟嘟声。我们建议作为问题反馈的听觉指示器是没有声音。当前听觉反馈存在的问题是缘于广泛流行的观念：和正面听觉反馈相比，负面反馈才是合情合理的。

【负面听觉反馈：宣告用户错误】

人们经常反对听觉反馈的理由是用户不喜欢。用户讨厌计算机发出的声音，他们不喜欢计算机冲他们发出噪音，对于如今广泛使用计算机声音的方式，这可能是对的——人们已经习惯了以下这些不幸事实：

- 计算机总是使用伴随警告噪音的错误信息框。
- 计算机的噪音总是很大，单调而令人不快。

有问题时发出噪音称为负面听觉反馈。在大部分系统中，错误信息框通常伴随高而刺耳的“嘟嘟声”，听觉反馈因此与它们强烈相关。这种噪音是对用户错误的公然宣告。它向听力范围内的所有人公布你已经做了可恨而愚蠢的事。这是一种如此可恨的习惯用法，因此现在多数软件开发人员都深信听觉反馈是糟糕的，再也不应该看做界面设计的

一部分。没有比这更背离真理的啦！这是负面反馈带来的问题，不关听觉反馈的事。

负面听觉反馈有几个缺点。因为负面听觉反馈是在出问题的时候发起的，它自然承当了警报的角色。出于警报目的，它必然会设计成为吵闹、刺耳和扰人心神的，当房间着火，生命安全受到威胁时，希望它能唤醒熟睡的人们。它们就像保险，我们所有的人都希望从来不要听到。不幸的是，程序经常不能处理用户所做的事情，于是这些动作成了正常交互过程的一部分。警报在这种正常的关系中没有位置，同样，在偶然改换车道而没有使用转向灯时，我们也不希望汽车警报器又不响了。也许负面听觉反馈最受谴责的方面在于它暗含成功只需沉默。人类喜欢听到一切正常，他们也需要知道什么时候做得不太好，但是这不意味着他们喜欢听到这些。负面反馈系统明显不如正面反馈系统受欢迎。

在没声音与有声音的负面反馈之间，人们会选择前者。在没声音与有噪音的正面反馈之间，人们会根据个人情况和品位选择。在没声音与柔和悦耳的正面听觉反馈之间，人们会选择后者。在程序中我们从来没有给用户提供过高质量的正面听觉反馈，所以不奇怪人们会把声音与坏的界面联系在一起。

【正面听觉反馈】

几乎每个软件世界以外的对象和系统都是提供声音来表示成功而不是失败的。在我们关门时，听到滴答声，我们就知道锁上了，而没有声音则表示还不保险。当我们与人交谈，他们说“是”或者“啊哈”，我们知道他们至少最低限度地注意到了我们所说的话。但当他们保持沉默时，我们有理由相信出了什么错。当转动点火装置的钥匙没有听到声音时，我们知道有麻烦了。当我们扳开复印机的开关，它不是发出嗡嗡的声音，而是冰冷的沉默，我们知道出问题了。甚至很多我们认为沉默的设备也会发出噪音：打开炉子时，煤气发出嘶嘶声，点燃炉子时发出动人的“叭”的声音。因为没有声音，电炉天生就不如煤气灶友好，更难使用——它们需要指示灯告诉我们它们的状态。

当工具使用成功时发出声音，它就称为正面听觉反馈。我们的软件工具通常是沉默的；我们能听到的只有静静敲击键盘的声音。嘿！那就是正面听觉反馈。每次你按下键盘，你都可以听到微弱的，但是正面的声音。键盘制造商完全可以制造极其安静的键盘，但是他们没有那样做，因为我们依赖听觉反馈告诉我们工作进行得怎样。反馈不必是复杂的——那些嘀答声并没有告诉我们很多信息——但是它们必须是一致的。如果没听到声音，我们知道没有敲到键。正面听觉反馈真正的价值在于，它的缺失是非常有效的问题指示器。

正面听觉反馈的有效性起源于人类的敏感性。没有人喜欢别人告诉他失败了。错误

信息框是负面反馈，在用户做错了事时通知用户。沉默可以确保用户知道而又没有真的告诉他失败。它有明显的效果。因为软件不必伤害用户感情，而又能完成任务。

我们的软件应该给我们持续小声的听觉暗示，就像我们的键盘一样。如果程序在用户行为正确时发出微弱而又容易辨别的声音，程序会更友好和更容易使用。程序可以每次在用户输入有效数据时，发出一个上声调。如果程序不能理解输入的数据，它可以保持沉默，用户就会立刻知道问题，在不尴尬或伤害自尊的情况下改正输入。无论何时用户开始拖动图标，在对象拖动时，计算机可以发出一个微弱而使人联想到滑动的声音。当它被拖到受范区域时，一个附加的敲击声提示这种碰触。当用户最后释放鼠标的时候，如果成功，他会从扬声器里听到一个柔和高兴的“扑通”声作为奖赏，否则，如果放在了无效区域就不会有声音。

和视觉反馈一样，计算机游戏擅长于正面听觉反馈。Mac OS 9 还用精细的正面听觉反馈在文档保存和拖放时做了很好的工作。当然，听觉反馈必须在正确的音量状态。Windows 和 Mac 提供了一个标准音量控件，所以良好听觉反馈的障碍已经排除。

丰富的非模态反馈是处理交互设计的最伟大工具之一。用精细而强大的非模态交互取代恼人而无用的对话框，可以让用户不屑一顾的程序变成他们爱不释手的程序。思考一下你可以用丰富视觉非模态反馈和其他非模态反馈机制来改造程序的所有方法。

35

与用户的其他交流方式

本书的这部分内容在讨论如何与用户交流。如果不讨论与用户的交流方式那就是我们的失职，这些交流方式不仅对用户有帮助，对你，软件创建者或发行人，维护产品的品牌和标志也不无裨益。在最好的情况下，这些交流方式没有什么争议，而本章提出的建议能够让你充分理解用户交流的两面性。

桌面上的标志

现代桌面屏幕已经非常拥挤了。一个典型的用户要同时运行半打程序，每个程序必须维持自己的标志。当用户要完成相应的工作时，必须能认出程序，你必须因你创建的程序得到应有的信誉。在软件中声明标志有几个惯例。

程序名称

依照惯例，程序名称会在程序主屏幕的标题栏中标明。这个文本就是**标题字符串**（title string），通常是由程序主窗口拥有的单一文本值。微软 Windows 引入了一些复杂性。自

Windows 95 开始，标题字符串在 Windows 界面中担当起更重要的角色，特别是在程序任务栏上的**启动按钮**（launch button）显示的标题字符串。

当打开的程序数目增加时，任务栏上的启动按钮将随着按钮增多而体积减小，随着按钮变短，标题字符串也会相应删减。

最初，标题字符串只包含程序名和公司品牌名。这时就存在冲突了：如果你在程序名上加上公司名，比如说“Microsoft Word,”你会发现任务栏上只能有七八个运行的程序或者打开的文件夹，并且把程序启动按钮文本字符串截成了“Microsoft”。如果还运行了“Microsoft Excel,”你会发现两个相邻按钮有着相同点，但没什么用的标题字符串。它们名字中的不同部分——“Word”和“Excel”——隐藏起来了。

多年来，标题字符串起到了另一个作用。许多程序用它来显示当前活动文档的名字。微软的 Office 套件程序就是这样做的。在 Windows 95 中，微软将活动文档的名字拼接在标题字符串的右端，用一个连字符将它与程序名分开。在后来的版本中，微软颠倒了顺序，文档名在前。就用户看来，文档名是第一位的，当然是更目标导向的选择。该技术并不标准；但是因为微软这样做了，所以它经常被人效仿。这样标题字符串就变得太长，以致不适合用在启动按钮上——但工具提示可以帮忙。

微软所做的是在程序内部数据结构中添加一个新的标题字符串。这个字符串只用于（任务栏上）启动按钮，而原始的标题字符串用于窗口标题栏。这样设计师和程序员能够为任务栏有限的空间裁制合适的启动按钮，而标题字符串总是在有多余空间的标题栏上显示它的全长。

程序图标

程序标志的第二大组成部分是它的图标。Windows 中要关心两个图标：标准的图标是 32×32 像素，小型图标是 16×16 像素。Mac OS 9 和更早的操作系统有相似的安排；OS X 中的图标理论上能够非常大——达到 128×128 像素。Windows XP 似乎也可以使用 64×64 像素，因为今天的显示屏分辨率已经超过 1600×1200 像素。

32×32 像素大小的图标用于桌面，16×16 像素的图标用于标题栏、任务栏、资源管理器，以及 Windows 界面上的其他位置。因为当代图形用户界面的视觉艺术性越来越重要，所以你必须更加注意程序图标的质量，特别是在你希望程序图标从远处可以直接辨认——尤其是微缩版本时。用户不必立刻认出它——虽然这样会很好——但他应该可以直接看出它与其他图标的不同。

图标设计本身就是一门工艺，要做好比表面上看来困难得多。可以证明，Susan Kare

设计的原始 Macintosh 图标创建了行业标准。今天，许多视觉界面设计师专门从事图标设计，任何程序都将受益于图标设计中的天赋和经验。

辅助应用程序窗口

程序的辅助窗口并不真正属于程序功能的一部分，而是作为惯例提供的。这些窗口或者只能根据请求获得，或者程序仅仅提供一次，如程序的致谢字幕。这些由程序单方面显示，只有在程序第一次使用或每次启动时才出现，但是从交流渠道来说，它既能帮助用户，又能更好地传达品牌。

“关于”对话框

习惯上，“关于”对话框（About boxes）是向用户标志程序的一个单独对话框。它也用做程序的致谢字幕，标志创建它的人。具有讽刺意味的是，“关于”对话框告诉用户有关程序的信息并不多。在 Macintosh 中，“关于”对话框可以在苹果弹出菜单（Apple pop-up）顶部调用。在 Windows 中，它几乎总是可以在帮助菜单的底部发现的。

微软在它的程序中采用了一致的关于对话框，它的设计方法简单，正如你在图 35-1 中见到的。微软将关于对话框专门用于标明，类似于软件的驾照。这样很不幸，因为它应该是好奇的用户了解程序概况，而不会使不感兴趣的用户感到唐突的好地方。紧跟微软的设计步伐，并不总是件好事，这就是一个与微软不同时可以提供很大好处的地方。

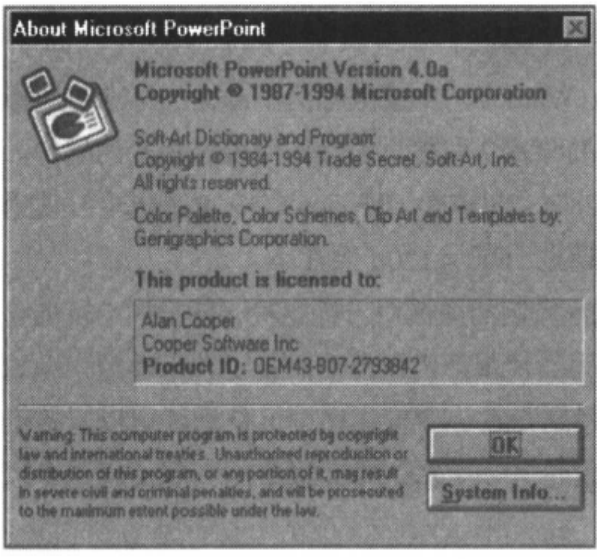


图 35-1 PowerPoint 的“关于”对话框是微软解决方法的典型例子。它告诉用户程序的确切名字和版本，说明相关的版权，发布法律警示（啊！），并显示用户的名字和公司。程序的图标传统上显示在左上角。问题是，如果朋友请你告诉她有关 PowerPoint 的一些信息，你大概不会重复相关的版权，而只会向她描述一下程序的概况。那么这张图片怎么了？

微软解决方法的主要问题在于，关于对话框没有告诉用户有关程序的信息。事实上它是标志框（identification box）。它通过名字和版本号标志程序，它标志程序的各种版权；标志用户和用户的公司。它确实是一个有用的功能，但只是对于微软的客户支持更有用，而不是对用户更有用。

让关于对话框更有用的期待是很强烈的——否则我们就不会看见它上面的内存使用和系统信息按钮。通过采取更具目标导向的方法，我们可以在关于对话框中加入一些信息，使它真正对用户有所帮助。关于对话框可以告诉用户的最重要的事情是程序的范围。最广泛地说，它应该告诉用户程序能做什么和不能做什么。多数程序的作者忘了许多用户一点也不了解 InfoMeister 3000 4.0 版本的实际功能，这是给他们友善提示的好地方。

关于对话框也是让新用户成功地开始第一课的好地方。例如，如果存在对用户交互非常关键的新习惯用法，如直接操作方法，这是简短地告诉他这些信息的好地方。另外，关于对话框可以引导新用户去访问有助于她理解程序的其他信息源。

因为当前这种工具的设计只是陈述了难懂的条文，而没有提供程序信息，从这点来说它应该称为标志框（identity box），而不是关于对话框。标志框为用户标志程序，图 35-1 中的对话框极好地符合了这个定义。它告诉了我们所有律师需要的材料，以及技术支持人员需要知道的材料。显然，微软认为标志框非常重要，而可以牺牲真正的关于对话框。

正如我们所看到的，标志框必须提供基本的标志信息，包括出版商的名字、程序图标、程序的版本号，以及作者的姓名。另外一个有用的项目是出版商的技术支持电话号码。

许多软件出版商没有把程序与某次具体的软件构建联系在一起进行区别。出于市场的原因，一些销售商甚至为非常不同的程序冠上相同的版本号。但是标志对话框或关于对话框的版本号主要用于客户支持。一个误导性的版本号码将会耗费出版商更多的电话支持时间，只为找出用户所用程序的精确版本。不管用什么方案，只要数字非常确定就可以了。

报告版本号的一个重要部分是告诉用户它取代的是以前哪个版本。知道这是版本 3.2 没有绝对的意义。但是知道版本 3.2 是版本 3.1 的改进，并取代了所有的 2.x 版本是有用的。出售商致力于提升他们的软件，每个版本都试图取代某些以前的版本，发布小型的增补版本是为弥补以前版本的不足，但不能完全取代前者，同样，一种特殊的版本允许与某种新硬件或软件兼容，这都应该有所说明。

如果你打算显示一个信息丰富的版本号，在这个对话框中解释一下版本号的细节无伤大雅。多数用户会忽视它，但是会受到数以千计的 IT 公司经理们的欣赏。

许多程序用序列号来惟一地标志。当然，这里是显示号码的地方。用户需要用这个号码与出版商联系，或者用于公司记录，所以程序应该让用户看到它，并且可以选择它

用于拷贝。

关于对话框（不是标志对话框）绝对是说明产品团队名字的好地方。作者坚信应该感谢软件设计、开发和测试整个过程中所有有贡献的人。程序员、设计师、管理者和测试者都应该看到他们的名字。文档撰写者有时会把名字写上手册，但是其他人则只有程序本身而已。关于对话框是与主屏幕没有功能重叠的极少数几个对话框之一，所以没有理由说它不能太大，可以腾出空间告诉大家谁做了贡献。尽管一些程序员并不关心在屏幕上看到他们名字，但许多程序员会因此受到极大的鼓舞，真正欣赏这样做的管理者，有什么理由不为创建程序的这些聪明而勤奋工作的人们署名呢？

最后一个问题是指向 Bill Gates 的（就像在本书 1995 年第一版中一样），他有一个企业内部规定：程序员不可以将他们的名字写进程序的关于对话框。他觉得很难知道这群个体的最后界限。但是作为娱乐产业指示器的现代电影的片头字幕却没有这样的担心。事实上，在游戏软件中开发致谢极为常见。也许现在微软大量投入游戏软件开发和销售，事情会有所改变——但别期望太高。

微软的政策是令人烦恼的，因为它的惯例广泛地应用着。结果，它不公布程序员名字的政策也被盲目效仿的公司广泛复制。

闪屏

闪屏（splash screen）是在程序第一次加载进内存时显示的对话框。有时它和关于对话框或标志框一样，自动显示，但更常见的是，出版商会创建一个单独的更动人、视觉上更生动的闪屏。

只要用户启动程序，闪屏就应该显示在屏幕上，这样在程序加载为自身运行做准备时，用户可以看到它。几秒钟后自动消失，程序开始执行任务。如果在闪屏使用期限内，用户按下任意键或单击任何鼠标按钮，闪屏应立刻消失。即使它只显示几毫秒时间，程序也应该表示出对用户时间的尊重。

闪屏是建立好印象的极佳机会。它可以用于加强用户的想法：购买你的产品是一个好的选择。它通过显示公司标志（Logo）、产品标志、产品图标和其他合适的视觉符号帮助建立视觉品牌。

闪屏也是很好的工具，可以引导新手用户去访问那些不常使用的培训资源。如果程序有内建的教程和配置选项，闪屏可以提供按钮直接访问这些工具（在这种情况下，闪屏应该保持开放直到手工关闭）。

因为闪屏将会被新手用户看见，如果你有什么要向他们交代，这是个很好的地方。

另一方面，你向新手用户提供的信息会惹恼有经验的用户，所以后来的闪屏实例应该更通用。无论你说什么，都必须简洁明了，不要拖沓冗长。闪屏上的一条恼人的信息就像鞋里的石子，如果不能很快清除它，马上就会带来痛苦。

共享软件闪屏

如果你的程序是共享程序，闪屏可能对你来说是最重要的对话框（尽管不是用户最重要的对话框）。它是你通知用户产品的用途和付款方式的机制。一些人将共享软件闪屏当做**内疚屏**（guilt screen）。当然，这些信息也应该放在用户能看到的地方，通过程序每次加载时向用户显示，可以强化程序应该付费。另外，万一产品销售到偏远地区，你也需要提供联系途径。最好的方法是创建一个优秀的产品，而不是让潜在的用户在使用过程中感觉内疚。

在线帮助

在线帮助就像是打印的文档，为永久的中间用户提供参考工具。最终，在线帮助并不重要，就像汽车的用户手册并不重要一样。如果你发现自己需要一个手册，那意味着你车子设计很糟糕。设计才是真正重要的。

一个有许多特点和功能的复杂程序应该有一个参考文档：希望拓宽产品认识视野的用户能够在这里找到答案。这个文档可能是打印手册或者在线帮助。打印手册内容详尽、可浏览、友好而且可以方便地带在身边。在线帮助可以搜索，比较详尽，非常轻，也很经济。

索引

因为你不会像读一本小说一样阅读手册，所以成功的关键和参考文档的有效性在于搜索工具的质量。这基本上取决于索引质量，打印手册后面有一个索引，在线帮助有一个自动索引搜索工具。

作者怀疑见到的在线帮助工具很少由专业索引师编撰。但是，索引里有许多条目，可能会使数字加倍。更甚者，生成索引需要研究程序和特征，而不是帮助文本。这很不容易，因为它要求技术很高的索引编写人员非常熟悉程序的特点。改善界面要比创建一个真正的好索引容易得多。

索引条目列表无可争议地要比条目文本更重要。用户会更容易接受撰写糟糕的条目，但无法接受某个条目的缺失。索引应该有尽可能多的主题同义字，要做好准备，工作可能非常艰巨。需要解决问题的用户会想“我怎样才能把这个单元格变黑？”而不是“我怎样把这个单元格设置成100%的底纹？”如果该条目列在底纹中，那么这个索引对于用户来说是失败的。你思考的方式越目标导向，索引就会越好地与用户寻找某物时最可能浮现在他脑海里的事物匹配。最有效的索引模型是 Irma S. Rombaur Marion 和 Rombaur Becker（Bobbs-Merrill,1962）在 *The Joy of Cooking* 中使用的一个索引模型，那是作者用过的最完善、最实用的一个索引。

快捷方式和总览视图

几乎每个帮助系统都缺少**快捷方式**（shortcut）的选项；帮助菜单中的项选中时，以摘要的形式反映程序不同特征的所有工具和键盘命令，这是任何在线帮助非常必要的组成部分，因为它向永久的中间用户提供了最重要的特性：功能访问。与详细的用法说明相比，永久中间用户更需要工具和命令。

另外，在线帮助系统还缺少**总体视图**（overview）。用户希望知道 Enter Macro 命令如何工作，而帮助系统做一些无用的解释，说它是让你在系统中输入宏的工具。我们想知道的是它的范围、效果、能力、优势、缺点，以及从绝对和相对其他销售商类似产品的角度，我们为什么要使用这个工具。@Last Software 为建筑画图应用程序 Sketch Up 提供在线的流视频教程。这是非常棒的总体视图，特别是如果它们可以通过 CD-ROM 访问的话。

不是为新手用户准备的

许多帮助系统认为它们的任务是为新手用户提供帮助，这是不对的。新手用户一般会远离帮助系统，因为它通常和程序一样复杂。另外，任何通过实验无法掌握基本功能的程序都是不能接受的，大量帮助文本也起不了作用。在线帮助应该忽略新手用户，关注那些已经能够成功使用产品，而且希望成为永久中间用户的用户。

非模态和交互式帮助

工具提示是非模态在线帮助，它们的效果好得不可思议。另一方面，标准帮助系统

是在单独的程序中实现的，为了提供帮助它遮盖了程序的大部分。如果你向某人请教如何执行一个任务，他会用手指着屏幕上的对象加以解释。单独的帮助程序遮盖了主程序，做不到这一点。苹果公司采用了一个全新的帮助系统，通过用户顺序激活突出显示的菜单和按钮，指导用户逐步完成任务。尽管这不是完全非模态的，但它是交互式的，和用户期望执行的任务密切联系成为整体，不像参考帮助系统那样需要单独的房间。

向导

向导（wizards）是微软发布的一个习惯用法。它迅速博得了程序员和用户界面设计师的普遍喜爱。向导通过一系列对话框一步一步引导用户，试图确保成功。这些对话框与复杂的步骤平行，通常用于管理程序的功能。例如，一个向导可以帮助用户在PowerPoint上创建一个幻灯片。

程序员喜欢向导，因为他们像对待外围设备一样对待用户。每一个向导对话框会问用户一两个问题，最后无论请求的是什么任务，都会执行。它是由程序做出询问的一个很好例子，它们违背了公理：提问不等同于提供选择（见第9章）。

向导写成一步步的例程，而不是用户和程序之间的对话。用户就像机器人管弦乐队的指挥，挥动着指挥棒控制节奏，但对进展没有任何影响。以这种方式，向导很快退化为确认的消息，用户也学会了简单点击每一屏上的“下一步”按钮，无须严格地分析为什么。

向导在一些很少使用的场合能起到作用，例如，安装和初始配置。在交互较弱的HTML世界里，它们也成为Web上几乎所有事务的标准习惯用法——更好的浏览器技术会最终改变这一点。

创建向导更好的方式是变成简单的自动功能，不向用户提问而直接继续执行功能。比如，如果要创建一个演示文稿，它应该直接创建，然后让用户选择，或者后来使用标准工具来进行改变。典型向导的询问策略不友好、不可靠，也没有帮助。向导通常并没有向用户解释事情进行得怎么样。

向导的设计目的在于改进用户界面，但在许多情况下，它有着相反的效果，使得程序员可以在复杂的功能上添加原始的实现模型界面，并且温和地保证“使用向导会使它变得很容易。”这些让人联想到把责任推卸给用户的通常做法：“我们会把它写进文档。”

“智能”代理

也许没有太多必要谈到 Clippy 及其近亲，既然微软已经在 Windows XP 的营销中反对创建它。（但请注意，XP 中并没有真正删除 Clippy）。Clippy 是微软研究 BOB 时创建的一个遗迹。BOB 是一个“直觉”的真实世界，一个充满隐喻的界面，与第 20 章中讨论的 General Magic 的 Magic Cap 界面极其相似。BOB 广泛地使用了拟人化的动画人物与用户交流，引导完成任务。这是微软给人印象最深的界面失败之一。Clippy 是这些帮助代理的后裔，和它们一样令人讨厌。

智能动画代理一个显著的问题在于拟人化，软件将赌注押在用户对代理智能的期望上。如果它不能满足这些期望，很快就会激怒用户，就像专卖店里的销售人员向用户宣称产品里有一个专家，但是在几个简单问题之后，他证明自己没有根据。

这些东西很快就变得倒胃口，令人心神不宁。微软办公软件的用户试图完成一些任务，而不是要在滑稽而错误百出的帮助系统中寻找娱乐。多数程序需要更直接，少一些干扰，更值得信任的获取帮助的方式。

36

安装过程

安装过程是一个更值得评论的软件交流形式，因为它是程序与客户的第一次交流。

多数软件开发经理被安装程序的非生产性所蒙蔽，所以他们并没有把安装程序看做一种机会（或危险）。如果你的安装过程是有效界面设计的展示柜，用户在开始使用你的程序时就会感觉很好。相反，如果你的安装过程是事后添加的补丁，用户可能会对你的产品吹毛求疵。

最好的安装就是无须安装

有时最好的交流就是压根没有交流。任何称职的 IT 管理员知道这对于安装来说是真的。安装意味着维护更新。它意味着呼叫中心会被各种问题和麻烦包围。它也意味着不得不恢复用户使用讨厌而愚蠢的安装界面时无意造成的破坏。

对安装的憎恨正是以 HTML 编写的基于浏览器的企业应用程序在公司办公室无所不在的主要原因之一。没有哪个理智的 IT 经理希望把时间花在无止境的安装和升级上。对于 IT 经理来说，无须安装的软件是天赐之物。从不介意以浏览器为基础的 Web 应用程序速度慢、交互差，造成用户无尽的烦恼——对于 IT 专业人士来说，它们跨平台运行、自

我更新的事实具有很强的说服力。他们中的大部分人不必使用他们购买的软件。

具有讽刺意味的是，IT 管理者甚至是开发人员将基于浏览器（browser-based）和免安装（installation-free）混为一谈。创建不以浏览器为基础、但基于因特网的应用程序是完全可能的，它具有全桌面应用程序（full desktop application）的所有交互优点，还和你通过浏览器访问的任何应用程序一样具有免安装的特性。微软办公应用程序在 Windows XP 中通过自动更新已经支持一些这样的程序。它向通过因特网和企业内部网实现完全自动安装前进了一小步。网络应用程序（Network-savvy）从不局限于浏览器技术，但是基于 HTML 的软件易用性在很大程度上让程序员忽略了这一事实。能够创建客户-服务器模型应用程序，并提供基于浏览器的应用程序的所有好处，但与此同时，也允许同样丰富的交互（根据程序员能够管理多少客户端事务而定），而这种丰富性在 HTML 降低我们的期望之前，是我们曾经期望从应用程序获得的。

接二连三的厄运

回到手工安装的时代，不管是从 CD-ROM 安装，还是文件服务器，或者因特网，你所经历的过程基本相似，包括两个步骤。首先，软件必须拷贝或者加载到你的本地硬盘。其次，软件通常需要一些初始的配置，这样你和软件都能够顺利工作。

Edward Tufte，《量化信息的可视化显示》（Graphic Press, 1983）一书的作者讨厌手工安装软件的过程，把这称为“接二连三的厄运”。它没有帮助用户实现目标，也没有帮助程序完成自己的功能。

大多数程序现在使用标准的安装脚本，如 Windows 平台上的 InstallShield 或者 Mac Installer。这些程序与开发商从头开始创建的时代相比，有了很大的改善，并且现在的用户已经在某种程度上熟悉了这些标准的格式。但即使在这些框架里，也很少从目标导向的角度考虑交互问题。因此，安装程序不断盲目地询问用户，迫使他们做出没有告知的决定，并且自大地对计算机的使用方式进行假设。

安装程序中体现出的常见问题，总的来说，是那些最讨厌的软件界面设计问题的一个缩影。大多数安装程序至少表现以下设计错误的其中几种：

- 在不通知后果的情况下，要求用户做出响应。
- 不告诉用户动作的影响范围。
- 提出用户不可能知道答案的问题。
- 向用户询问计算机自身能够回答的问题。
- 不做准备。

- 不提供卸载方法。
- 忽略先前的活动。
- 滥用系统范围内的文件。
- 将文件放在不妥当的地方。
- 覆盖共享文件。
- 不向用户提供程序的任何信息。
- 混淆配置和安装。
- 要求用户主动参与。

下面我们逐一详细讨论每一种设计错误。

在不通知后果的情况下，要求用户做出响应

毫无疑问，在安装程序的所有罪行之中，这是最常见的。安装程序提供一个看起来像图 36-1 这样的对话框。

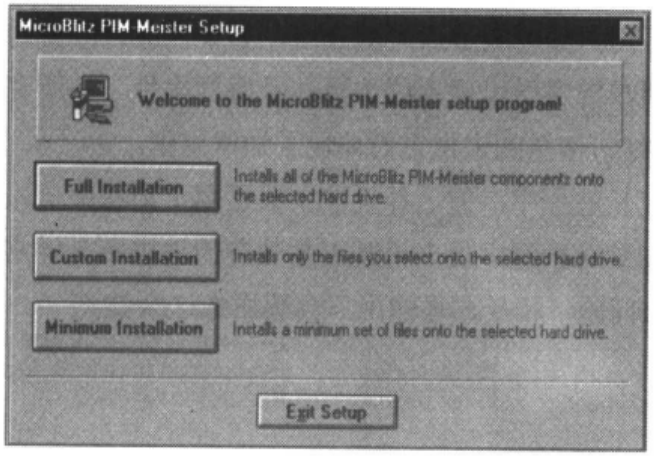


图 36-1 这是典型安装程序的第一个对话框。当你播放一个视频游戏时，只有你的智慧才能指导你。完整安装对我来说是不是太多了？我是不是够聪明，能定制这个程序？如果选择最小安装，会不会显得我很没用？

程序一开始就向你提出一个有着全局影响的问题，这个过程可能是不可逆转的，并且有着你并不理解的后果。一些更高级的安装程序，特别是微软和苹果的安装程序，它们会合理地告诉你，你的选择会消耗多少硬盘空间。然而，你仍然需要猜测你所做选择的含义。用户需要确切知道如果选择最小安装，他会失去哪些功能。仅仅知道消耗多少硬盘空间是不够的，他也必须知道选择的含义。

类似的问题涉及到系统层次的资源，如通信端口、视频驱动程序等。这令人特别恼火，因为错误的选择会立即锁住系统，使程序崩溃，丢失数据，有时候甚至需要用启动盘重新引导系统，并且通过安装程序手工地改正这些错误。虽然错误选择的后果是严重的，但安装程序很少让用户意识到这不能求助于猜测——甚至不能求助于有根据的推测。

用这种问题来考量用户交互，软件应该体现知情（informed consent）原则。应该只向用户询问他们知道后果的问题。特别是，他必须理解对于他来说的后果，而不仅仅是对计算机来说的后果。如果程序提供配置选择，必须让用户完全了解不同的配置将如何影响程序的能力来帮助实现他的目标。

例如，创建知情（informed consent）气氛的合适方式是提供逐条记载的，“以它们能为用户做什么表示的”特征清单，并且这些清单要么包括要么排除在多种选择中。另外，从用户角度的有关整体图景的散文式描述是必要的。例如以下的文字就很好：

最小的安装是为膝上型电脑和笔记本电脑设计的，需要不到 100 MB 的硬盘空间。MicroBlitz PIM-Meister 会消耗比完整安装少 40%的硬盘空间，而不会牺牲任何重要的功能。你所牺牲的功能最多但不是全部包括：在线帮助文本；新手用户的教学程序；7 个自动程序中的 4 个，以及大多数更为晦涩的导入和导出功能。如果想选择最小安装，但感觉到缺少一个或更多的功能，你能够很容易地选择带有特殊选项的最小安装，并且添加期望的功能。而且你可以通过重新运行安装程序来改变已有的安装。

首先，以上声明描述了用户选择最小安装选项的主要理由。其次，它详细地描述了采用最小安装会牺牲哪些功能。第三，如果用户需要，它告诉用户如何覆盖这些设置，以及为什么覆盖这些设置。第四，他告诉用户如何在最后时刻改变，如果他应该改变的话，这让用户放心。这就是知情原则，用户能够做出理性的选择并且感到愉悦。在所有的场合中，Apple 的安装程序没有这么详细，但总的来说它们体现了知情原则。

通常要避免在编程周期中创建详细冗长的程序，但这种说法主要适用于为程序员服务的编程工具以及经常性的程序交流。安装对于程序员来说既不是独有的，也不是经常性的，所以不要害怕向用户详细解释安装问题，相反，你的用户会感谢你这么做。

不告诉用户动作的影响范围

典型地安装程序不会在程序员称为“与用户闲聊”上花时间，但这种闲聊对于缓解用户的不安非常重要。想像一下，如果一个家电修理工来到你家，一句话也不说，就开始拧开水管、拆开冰箱的情形。如果你的修理工开始的时候能给出如下简单的描述，你会感觉更好：

你的冰箱压缩机已经完全不能工作了，因为发动机已经失灵了，我必须彻底更换它。在我的卡车里有可置换的发动机，需要一个半钟头的修理时间，其中包括以环保的方式向系统补充制冷剂。你的质保单会支付部分费用，但不包括我的劳务费。我每小时收费 45 美元。管道设备还需要稍微修理，因为制冷器和你的管道直接相连。任何影响铁制管

道的时候，就会有其他地方漏水的可能。我会努力不移动管道，降低这种风险。

你现在已经知道操作所需要的时间，需要花费多少钱，失败的风险是什么。你已经知道了操作的范围。那么，如果安装程序能告诉我们下面这些信息不是很好吗：

我将在你的系统上安装 MicroBlitz PIM-Meister，意思是说把程序从光盘上拷贝到你的硬盘，解压文件，然后根据你的特殊需要配置程序。从你的处理器判断，我估计完整安装需要 7 分钟，最小安装只需 2 分钟。程序将根据你选择的配置占用 45 MB ~ 124 MB。换句话说，它可能占用你硬盘现有容量的 2.7% ~ 4.7%。你当前的硬盘还没有用到一半，所以可利用的空间会降低 4.6% ~ 8%。

我将在新创建的特殊目录中存放所有程序和相关信息。你可以选择存放目录及目录名称，但你也可以使用默认的 PIM-Meister。我必须至少在 Windows 注册表中添加一项。这个条目限制为一个参数行，对系统中的其他程序绝对没有影响，即使你后来决定卸载 PIM-Meister 也是一样。如果你选中复选框，我会在开始菜单的程序文件夹里创建一个图标启动 PIM-Meister。

这个安装程序维持一份内部状态日志，这样万一它崩溃了，如果你只是重新运行，它也会聪明地恢复正常。

任何时候你都可以通过按下卸载按钮卸载这个程序。程序会从你的系统中移除，留下尽可能少的痕迹。但是你用 PIM-Meister 创建的任何数据文件将保持不变。如果你愿意，也可以请求节约空间的卸载，它意味着所有程序都会删除，但是会保留你的个人设置，以后再执行安装程序可以根据保留的个人设置恢复 PIM-Meister。

这段独白是冗长的，有经验的用户不会想去读它。但新用户会发现了解安装过程所蕴涵的意义会非常令人安心，他们会很高兴得知将要进行的过程会影响哪些范围。

提出用户不可能知道答案的问题

安装程序可能会让用户指定串口。大多数用户不知道串口为何物，或者他们有多少个串口，更不用说哪一个端口最适合程序。游戏和声卡安装程序经常询问用户可用的中断向量，这个问题大多数计算机工程师都回答不了，更不用说典型家庭用户。商业软件安装程序会询问有关网络支持的问题，以及正在使用的鼠标类型，大很多用户不知道如何回答这些问题。

向用户询问他们不能回答的问题是一个糟糕的主意。首先，这样程序得不到它需要的问题答案，相反，它得到的只是一个猜测。其次，它让用户在机器面前感觉很不好，只能证明自己不能满足程序的要求。这多令人难堪！

如果程序需要了解串行端口，它应该测试它们，并且看看哪些已经占用。它能够在很多配置文件中搜索当前端口使用情况，它甚至可以让用户移动鼠标，并且查看多个端口上的活动来减少可能性。

如果程序需要知道中断，程序能够观察或者侦听端口，以确定它们是否在用或者是否可用。通过推断或者查看系统信息文件和注册表（如果存在的话），安装程序能够很好地猜测（可能比用户的猜测更为可靠）。通过记录程序自身的发现，并且接着告诉用户它准备做的事可能会锁定计算机，用户能够关闭所有其他正在运行的程序，并且为程序的错误做好准备，用户能够重新运行程序，并且会发现程序早先的记录。它现在知道什么无效，哪些足以推断出正确的选择。不要将这种选择强加于用户，不能期望他们能够知道答案。

向用户询问计算机自身能够回答的问题

在所有的软件中，这是一个非常普遍的问题，但安装程序的作者必定会因此变得声名狼藉。程序向用户询问计算机的显卡类型，而它很容易通过检测系统来回答这个问题。程序询问用户有多少可用的硬盘空间，而它能通过询问文件系统很容易获得确切答案。当很容易通过搜索硬盘发现时，程序还问你另外的程序或者文件的位置。当你只有一个硬盘时，程序竟然还会问你将程序安装在哪个硬盘上。

计算机的任务是减少我们生活中不必要的琐碎之事，诸如此类的问题只能增加一些无意义的琐事，并且令人不快。任何程序需要的大部分信息能够，也应该能够不询问人类用户就能确定。

不做准备

安装程序另一个最令人讨厌的无礼行为是没有意识到它自身的存在。安装程序快乐地安装已经装好程序的相同版本。或者，它不知道自己正在用来改变配置，所以它竟然再次把文件从光盘拷贝到硬盘。

安装程序的设计师应该能够列出程序需要的所有环境，包括RAM、视频、硬盘、麦克风、游戏操作杆、鼠标、调制解调器和话筒。安装程序应该接着检测，以便在继续之前确认这些主观臆想是否真实。程序应该在开始工作之前常识性地检查系统，它应该查看先前的拷贝及其他需要的软件，应该检查致命或危险的情况，如内存或者硬盘存储空间不足。

大多数中间用户试图配置和个性化自己的独占姿态应用程序，让它使用起来更方便。当应用程序的升级忽略了这些个性化设置时，会很令人沮丧。很多商业管理人员每年差不多都要购买新的计算机，他们发现必须痛苦地重新手工设置自己使用的每个主要应用程序。好的安装程序应该知道如何从其他计算机及其他版本导入这些设置。

不提供卸载方法

虽然很多软件开发商看起来并没有意识到这一点，但顾客经常想从他们的计算中删除软件。顾客删除软件，要么因为其他计算机上需要这个软件，而这台机器上不再需要了，或者因为其他更重要的程序需要它们所占用的空间，或者因为用户认为程序不好，无须保留。每个软件开发商都应该提供删除程序的工具，就像它提供了安装工具一样。卸载工具应该像安装工具一样健壮，并且具有所有特性。它也应该遵循知情的原则，告诉用户程序的范围和后果。在程序合适地安装好后，Windows 平台上的添加和删除程序控制面板及程序注册表提供这些功能。程序安装器也应该能够提供这些功能，如图 36-2 所示。因为某种原因，苹果从来没有意识到卸载的重要，MAC 仍然使用费力的手工卸载过程。

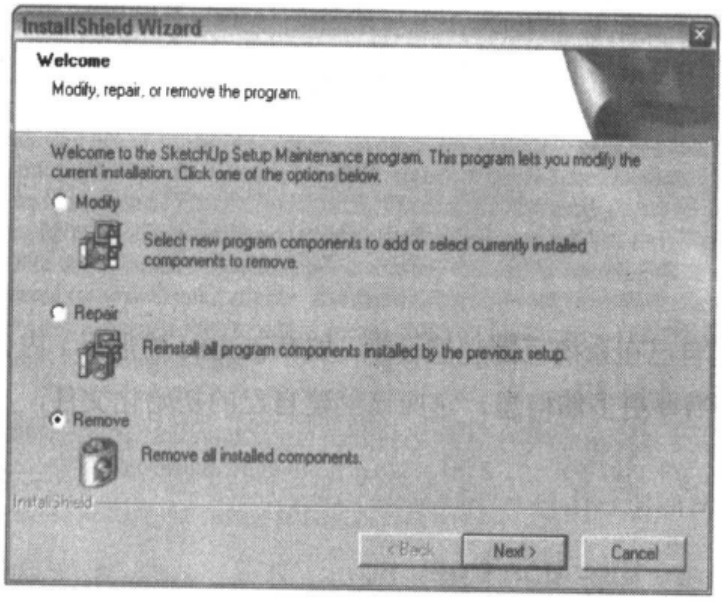


图 36-2 如果在这种环境下启动的话，好的安装程序能够意识到软件已经安装，并且提供了维护和卸载功能。在提供有用的信息方面，这个安装器可以做得更好一点，例如，安装了哪些文件，安装位置，它们占用多少空间及剩余多少空间（这些信息在显示后面的屏幕上，但用户需要在早期做出决定）。不过，总的来说，这个安装程序还不错，至少，InstallShield 的向导能够让用户重新返回，或者在他们确定不想安装或者卸载的情况下撤销。

这种主观假设是合理的：用户并不是憎恨你的程序，只不过想通过删除不再需要的教程或者一些已经证明没有必要的子系统来回收一些空间。卸载程序应该赋予用户这种能力，卸载程序的一些组成部分，而不影响应用的主要功能。用户也可能想将程序移动

到一个不同的硬盘或同一硬盘的不同位置。卸载程序应该知道如何转移程序，并更新所有的引用情况，改变以后，程序仍然顺利工作。

如果一个用户需要临时收回程序占用的磁盘空间（比方说她需要额外的空间用于两个星期的公务旅行），卸载器应该很容易能做到。它应该删除所有占用空间较大的文件，但是应该保留目录、配置文件和在其他系统文件中的记录。当用户公务旅行回来，安装程序可以不用覆盖，也能恢复磁盘上的大文件，也不会忘记个人设置。程序可以像以前一样重新安装。

一些程序，如网络、外围驱动程序和打印机共享软件不仅需要在硬盘上安装，也需要在引导时通过文件激活，因此它们成为了操作系统的永久部分。这类软件对卸载有特殊的要求，卸载器必须能够禁止程序而在物理上又不删除它。例如，如果想运行需要所有可用内存的游戏程序，在这期间用户可以禁止网络驱动程序，但不将它从磁盘上删除。

忽略先前活动

安装程序应该在用户的硬盘上保留活动日志。日志告诉程序以前做过什么和现在做什么。如果程序想从上次的执行中知道某些东西，不管是测试系统，还是询问用户，它应该在这里记录下来，这样可以加快重新处理，用户也不必再次受到打搅。Windows 的安装工具有这种智能恢复（Smart Recovery）的特点，如果软件已经安装，安装程序应该知道，并且在那种情况下调用时，它应该提供卸载和更新选项（见图 36-2）。

滥用系统范围内的文件

Windows 应用程序应该限制它们自己在系统范围功能设施（如 Windows 注册表）内的条目，不超过两到三个。如果程序需要更多的信息，它应该创建自己的初始化文件，在那里保存信息。

将文件放在不妥当的地方

程序应该在它自己的目录中操作。如果它需要多个目录，这些应该放在程序主目录的子目录中。程序永远不应该将文件放到其他目录下，除非其他特殊平台需要（如 Mac 上的扩展）。否则，如果用户希望安装一个新的 OS 版本时，在这个过程中程序文件必须删除。这样造成的故障和困惑令人非常不快，这完全可以避免。

应用程序经常忽视操作系统重装或升级的可能性。当重装 OS 时，程序经常不可避免地要依赖那些即将消失的系统文件中的条目。大多数程序需要完全重新安装，包括冗余地重新复制文件。

尽管程序的所有支持文件应该位于一个子目录中，但对于它创建的数据文件不一定要这样。将程序文件和数据文件单独保存是一种很好的实践，安装程序应该以那个前提开始。当前，在许多 Windows 程序的实践中，将所有的数据文件都归入“我的文档”文件夹。这里的意图很好，但是当许多程序的数据文件都归到这个目录中时，就发生了混乱。如果程序的数据文件可以以默认的形式存储在我的文档下的有名字的子目录中可能会好得多。

覆盖共享文件

许多程序会用到某种类型的运行库。特别是 Visual Basic 应用程序，为程序中使用的每一个可安装控件使用 VBRUN 动态连接库（DLL）、VBX 或 OCX DLL。当程序安装 DLL 或 VBX 时，它可能会用同名文件覆盖其他程序安装的文件。例如，如果程序 A 用名为 GRID.VBX 的商业 VBX 网格控件，另一个厂家的程序 B 用专门的 VBX 网格，名字也是 GRID.VBX，安装程序就会产生问题。即使名字是相同的，功能和界面可能完全不一样。当安装程序 B 时，它必须确保程序不会主观地假设这是一个早期版本而将 GRID.VBX 文件覆盖。如果做出这种假设，程序 A 就会神秘地突然崩溃，结果只会让用户困惑和生气。

如果两个不同的程序用相同 DLL 的不同版本，也会发生类似的问题。想像一下，程序 A 和 B 都用 DLL 的 1.0 版本名为 DATBASE.DLL。然后用户购买了最新版本的程序 B，其中包括最新 2.0 版本的 DATBASE.DLL。B 的安装程序可能认为它可以轻松地用版本 2.0 取代版本 1.0 的 DATBASE.DLL。但是程序 A 不知道如何处理新版本的库，程序崩溃就不可避免。这种崩溃特别隐蔽，因为用户可能在一月份安装了新版本的程序 B，到六月份才会运行程序 A。他对于是什么原因导致的问题没有任何线索。如果有的话，他也只会抱怨完全无辜的程序 A。尽管严格说来厂商并没有错，但这个问题可能影响到所有的厂商，意味着你必须采取有效的措施避免这种问题。

可以通过遵循两个简单的原则来避免这类问题。第一，使用不与其他开发商冲突的名字。例如用 XGRID，GRD 甚至 G7QL 代替 GRID 库的名字。用户可能从来不会看到这些名字，所以他们无须记忆。第二，附加一个惟一的号码表示库的版本。例如第一版命名为 XGRD1，第二版命名为 XGRD2，依次类推。

不向用户提供有关程序的任何信息

安装程序应该随时通知用户在做什么，还需要做什么。当前的安装程序在屏幕上设置一个对话框，用一个小的进度条显示已经完成的百分比。但是因为对它们代表的意义缺乏充分的说明，这些进度条多半是令人灰心和迷惑的。进度条是表示文件的进度吗？还是代表这个部分？这个光盘？全部的安装？它代表的是时间？还是复制的字节？用户已经习惯了无意义进度条的煎熬，他们已经知道不用去管它们。他们知道安装程序就是这么难懂。

混淆配置和安装

因为多数程序在运行之前需要一些基本配置，所以安装过程通常包括配置步骤。这是合理的，但是配置过程可能需要执行一次以上，而从光盘复制文件通常是一次性的操作。安装设计师经常忘了这一点，将这两个过程混在一起，以致没有复制操作（其实这是不必要的）就不能完成配置。安装程序应该聪明地意识到它是在重新运行，为用户提供程序现有实例的配置选择，而无须重复进行复制。

要求用户主动参与

一些安装程序不合理地要求你的时间和注意力。它们需要你主动地参与安装过程，即使你宁愿将这项工作授权给软件。例如，Windows 的 WordPerfect 安装程序会不时向用户提问，而打断程序到硬盘的复制。换句话说，它问一个问题，然后安装几个文件，然后再问你另一个问题，接着安装几个文件。

因此你被迫自觉地守着整个过程。安装程序应该在开始的时候提出所有相关的问题，然后任由用户在安装的过程中起身走开。如果一个新的光盘需要插入，它也应该产生听得见的提示，将你从对屏幕的监视中解放出来。

无疑，那些根本不需要用户参与的安装过程是最好的。

第八部分

超越桌面的设计

37 Web 设计

38 嵌入式系统的设计

37

Web 设计

万维网（world wide web）的出现对于交互设计师来说是福音，也是灾难。一方面，Web 的普及同时也普及了易用性非常重要这一观念，甚至比苹果计算机和 Macintosh 的威力还要大。以用户为中心的设计可能是图形用户界面创建以来第一次得到公司决策者的理解和接受。另外一方面，作为历史演化的自然结果，Web 交互的限制让交互设计退步了至少十年。在利用桌面交互的习惯用法（如拖放）方面，Web 设计师才刚刚开始，而这些习惯用法在最早的 Web 站点出现之前就使用了多年。

本章不想讨论 Web 设计的细节问题，很多其他的书已经讨论过，如 Steve Krug 的“Don't make Me think!(2000)”，Louis Rosenfeld 和 Peter Morville 的《信息架构》(1998)，特别是 Jeffery Veen 的《Web 设计的科学和艺术》（2000）一书以清晰而直观的语言探讨了 Web 设计的主要元素（Jakob Nielsen 的 useit.com 网站也是一个很好的资源）。相反，本章打算将第一部分讨论的过程和 Web 设计领域关联起来，从交互设计的观点提供一些看法。本章特别关注 Web 应用程序出现的一个新分支：基于因特网技术的事务性程序，它具有复杂的交互行为，且必然打破浏览器的限制。

好消息和坏消息

虽然 Web 的流行对交互设计师和可用性从业人员来说明显是好消息，他们的命运和工业一起沉浮，但就 Web 媒体的设计演化而言，也证明它是一把双刃箭。在 Web 繁荣的早期，工业界到处都是新鲜的艺术学校毕业生和传统的图形设计师，他们将 Web 看做令人激动和利润丰厚的机遇：通过新形式的交互视觉表达来创建有竞争力的交流。的确，Web 的早期有很多大好机遇，当时最大的挑战是克服媒体（最初用来共享科学论文和论文上的图表）的约束，用甚至是较低级的视觉和版式组织方式来创建文档。大量需求和供应不足的矛盾产生了对技能不足人员的依赖，他们通过猜测进行设计，拼凑式地工作。

即使那样，图形设计师也意识了到新的设计问题来自对文档中超链接的支持：内容的设计、组织和结构。Louis Rosenfeld（1998）创造的一个术语：**可发现能力**（findability）是简要描述这个设计问题的合适方式。新的设计师类别，信息架构师（information architect）创建了一个新的学科和实践，来解决逻辑结构和内容流的非视觉设计问题。信息架构，解决围绕内容的设计问题，作为一个学科仍然在继续发展和演化。

在 Web 的早期，图形设计经验处理当时的 Web 设计问题已经足够了。随着 Web 设计的增长，简单的 Web 站点演变为了大的信息交易场所，初露端倪的信息架构学科开始用来解决页面导航和内容组织问题。尽管技术约束随着更好的浏览器和标记语言的出现逐步变松，大多数 Web 设计师的关注点仍然是视觉表达和内容访问问题。只有一部分设计师注意到了软件的第三个突出特性：允许人们完成复杂事务的交互行为，由于历史的偶然，它是 Web 世界的迟到者。

今天的 Web 技术使得很多桌面应用程序和 Web 应用程序之间的差别非常模糊，但很多老式 Web 设计师仍然生活在 20 世纪 90 年代中期的世界里，那时视觉和信息设计是 Web 体验最显著的元素。随着 Web 交付的应用和 Web 服务（web_delivered application and service）市场的兴起（如 Microsoft 的.NET 和 Macromedia 的 Flash MX 支持的 B2B 电子商务和 CRM），基于因特网的软件设计变得越来越像桌面应用设计。虽然在可以预见的将来，Web 浏览器和基于浏览器的内容仍然会保留，但基于 Web 的应用设计时必须更多地关注浏览器内外的行为。浏览器的专制时代正在结束。.Net 和其他新技术支持的基于因特网的成熟应用将赋予人们更好、更丰富、更富有创造性的用户体验。当然，好的行为不会因为好技术的到来而自然到来，还需要做很多设计工作。第一步是识别一些盛行的 Web 设计神话，并且将它们放在特定的场景中研究。

Web 设计的普遍神话

围绕 Web 及新经济的热潮，产生了很多有关 Web 内容和基于 Web 应用设计的流行神话。我们能够理解这种神话的产生，大部分流行的技术革新都会这样。在 19 世纪末期，当电力在商业和工业成功中应用时，人们通常认为它不仅是改变商业和社会的神奇力量，而且可以通过“恢复生命力”，能够治愈几乎所有可察觉的疾病。像电力一样，Web 已经改变了我们的社会，虽然它创造的时间还很短暂。然而电力的应用花了 50 年才变得成熟，无疑我们要花很多年才能发现和认识 Web 带来的所有好处和 Web 的社会经济学。

商业和社会组织中一些最常见的 Web 神话如下所述。这些神话在某种程度上已经被管理者、市场、工程师，甚至是一些 Web 设计师所接受。在你的组织中纠正这些神话，越快越好。

神话 1：Web 本质上让事物变得更易用。在 Web 早期，全部交互就是单击文本或图片进入一个新的网页（或同一网页的不同部分）。因为行为极其简单，这种难以置信的交互匮乏严重限制了 Web 上可能完成的任务。在这个意义上，早先的 Web 确实容易使用，尽管结构和布局拙劣，会轻易地破坏用户体验。但是因为后来添加的一些结构如帧、表单、脚本动作和嵌入式 applet（小程序），这些表面的易用性丧失了。设计师现在面对的是和桌面应用程序同样丰富的系统。但是发展滞后的技术既不能支持我们期待的所有桌面应用程序习惯用法，也不能提供流畅的交互流和桌面软件支持的丰富反馈。对于 Web 设计人员来说，解决这些问题存在实质性的挑战。

神话 2：Web 设计是崭新而不同的。对于许多 20 世纪 90 年代中期进入 Web 设计领域的印刷品图形设计师来说，Web 当然是崭新和不同的。它有着更多的限制，更多的流动性。它也引入了一个崭新的概念（对他们来说）：文本超链接。但是对于多年来关注软件界面设计的软件设计师、图形用户界面程序员和可用性专家来说，Web 并没有什么新意。这个领域新的地方在于强调形式胜过强调行为与内容，因此**信息架构**学迅速弥补了知识上的差距。但是，Web 设计的关键已经从表达转变成事务，行为再次成为关注的焦点，虽然内容仍然重要。Web 设计过程与软件程序的设计过程没有真正不同，虽然内容的需要和某些技术的限制影响了设计最终采取的方法。

神话 3：Web 设计就是关于 HTML、布局和版式。自从 Web 出现以来，在 Web 设计周期中存在一个不幸的术语问题。转向 Web 设计的印刷和图形设计师带来了关注媒体角度的设计。传统的艺术和设计学派相当重视媒体的作用，这是对的，但是，这种面向媒体的设计思路应用在简单结构的设计时效果最好，因为不需要担心架构问题。直接编写网页，跳过了重要的规划阶段，而信息架构师很快意识到，对于复杂度超过一定页面

的 Web 站点来说，规划阶段是必需的。这就再一次体现了软件应用程序领域设计师的经验。在软件程序领域，计划和原型是主要的设计活动，后继的实现由程序员完成。在过去，前端的 Web 实现相对容易，经常和设计过程混为一团。对于在 Web 兴盛时期充实到设计队伍中的许多图形设计师来说，图像的设计也意味着实现。但是，今天为生产出更好的产品，多数商业 Web 站点需要在**设计/规划**和**实现**之间进行劳动分工。

神话 4：Web 设计仅仅是前端的东西。正如我们在本书的其他部分中希望说明的那样，设计关系到整个产品或服务，不仅是它做什么，而是包括它是什么及为什么目标服务。这方面 Web 没有什么不同；用户体验以及用户与系统的交互和感知意味着系统必须作为一个整体：在前端如何表达形式、内容和行为，在后端那些行为意味着什么。Web 设计在近乎相同的程度上与视觉设计、信息设计和交互设计密切相关。特别是后两者，对于事务系统的后端设计可能会有明显的影响。

神话 5：Web 设计和浏览器相关。尽管当前绝大多数 Web 程序在浏览器内运行，但是事情已经有所改变。基于因特网的软件程序已经存在了很长时间，随着.NET 技术的出现，存在于浏览器之外但基于因特网的程序数量会迅速增加。除了桌面应用程序之外，随着像 Handspring Treo 这样设备的出现，集成了无线通信的基于设备的应用程序也已经开始流行。各种基于 Web 的应用程序承诺发布和 Web 同样丰富的信息和媒介访问，而不受浏览器界面的限制。由交互设计师和产品团队决定什么平台最适合于新的产品或服务：浏览器或独立应用。浏览器适合于浏览，而复杂的事务则需要更完善的具有丰富习惯用法的交互模型。

神话 6：Web 设计与 Web 页面相关。在电子商务普及以前，Web 设计非常类似于页面设计。当然即使是那时，也有总体站点结构、逻辑流和导航。今天，Web 站点相对合理的静态页面层次关系（以及其他有向图）视图仍然存在。这种观念被信息架构学科出版的大量著作所推崇。很明显，今天从页面的角度考虑信息 Web 站点仍然有一定道理。同时，这种范例与高度事务化的站点根本不一致，如实时从数据库动态检索信息。随着前端站点技术有着越来越完善的交互能力，把动态生成的屏幕当做页面来处理的概念将越来越弱化。高度事务化的 Web 应用程序最好按它的本来面目：客户-服务器程序来对待。打破面向页面的思维限制将允许设计师更好地实现事务 Web 应用程序用户的真实目的和需求。

神话 7：Web 和因特网是同义词。早期的 Web 技术（特别是 HTTP 和 HTML）设计用来以异步和请求驱动的方式，传输和格式化 ASCII 文本及少数二进制图像。Web 浏览器和服务器很适合这种操作，但并不适合以同步或流模式交换大量的异构数据。因特网与支持浏览器访问的 Web 相比，是更丰富的网络。使用因特网协议（IP，TCP）的分布应用程序比基于浏览器的 Web 应用程序更灵活，功能更强大。

神话 8：Web 应用与本地桌面应用相比更容易也更快创建。HTML 支持快速原型，某些面向信息的 Web 站点行为简单，因而支持快速构建。但是具有复杂行为的事务性 Web 应用程序的开发和本地代码一样，具有相同的工程挑战。同样，它不可避免地需要健壮的设计和开发方法学。

Web 站点与 Web 应用

Web 浏览器最初理解为共享和链接文档的手段，而不需要麻烦的协议和文件传输工具，如 FTP，Gopher 和 Archie。没有多少人预见到 Web 的流行性，起初它是为个人交流和社区创建，后来又为商业交流和服务，因此 Web 协议是从微不足道的开始中成长起来的，浏览器和这些协议所支持的相对低水平的交互，也受到了最初理解的 Web 意图的限制。

Web 站点

本章，我们谈到的 Web 站点是指在 Web 上以信息为中心的服务，其交互层次主要与搜索及相应的链接相关。大多数 Web 站点或者包含静态的网页，或者由数据库提供的完整文章和文档。如前面所述，很容易把 Web 站点理解为按顺序、层次关系，或者有向图组织起来的一系列页面或文档，它有一个导航模型，将用户从一个页面带到另一个页面，也有一个搜索工具，提供更为目标导向的特殊文档定位。Web 站点以多种形式存在，有个人 Web 站点，公司营销和支持 Web 站点，以及信息中心的企业内部网。Web 站点设计主要关注内容、组织（信息层次关系）和条理性。行为包含的只不过是一致的导航计划（不要轻视它；正确导航是站点容易使用的关键），以及适当使用启示（affordance）和习惯用法来提示用户哪些元素是活跃的。

Web 应用

相反，Web 应用具有很强的事务性。大多数（如果不是所有）屏幕同时动态显示来自不同数据库的不同信息和表单。屏幕的某些区域甚至可以通过 applet 程序实时传送数据，或者支持客户端实时处理数据。Web 应用程序可能存在于浏览器中，也可能作为独立的应用程序运行。因为 Web 应用的本质是事务性的，它主要关注行为而不是内容。也就是说，事实上必须将两者联合起来考虑，因为对于多数 Web 应用程序——比如

Amazon.com 等电子商务站点——内容驱动着事务。但是，正如我们所见到的电子商务程序一样，即使内容（商品）表达得很好并且存在需求，如果事务困难，不能令人满意甚至令人厌烦，用户就会放弃他们的努力。在电子商务领域，到处都是潜在客户遗弃的购物车，他们一看到收款台附近 800 磅的大猩猩就打消了购物的念头。

Web 应用在电子商务以外的许多领域存在。近年来，像 SAP、Oracle 和 Siebel 提供的企业资源规划系统（ERP）和客户关系管理系统（CRM）已经移到 Web 平台。许多财政规划、在线银行和个人信息管理系统都有 Web 版本可以使用，它们的优点在于用户可以从任何一台与因特网连接的计算机（甚或是 PDA）登录。

那些碰巧运行在浏览器窗口中的企业 Web 应用或其他特殊 Web 应用可以像桌面应用程序那样呈现给用户，只要小心地设计交互来反映工程技术约束，就不会有问题。目前，因特网上的商务站点可能更接近于传统的 Web 站点，因为在这些站点中，有一定比例的站点，正是由大部分静态信息页面组成的。

【暂态 Web 应用】

Web 应用和桌面应用一样，存在独占应用程序和暂态应用程序两种。典型的暂态 Web 应用以交互式 applet（小程序）的形式嵌入标准的 HTML 网页。这种暂态 Web 应用程序的一个好例子是微软的 MSN 网站。暂态 Web 应用程序通常是实用程序（utility），要么用得很少，要么用得很频繁，但他们不像独占应用程序那样，用户一盯就是数小时。暂态 Web 应用程序除了需要遵循暂态桌面应用程序相同的规则以外（见第 8 章），还需要遵循另外的一些规则：

- **暂态 Web 应用应该清晰显示它们的功能**。因为 Web 站点的用户习惯了有限的交互，暂态 Web 应用程序在这种上下文中，就必须特别小心，提供清晰的启示和提示，如果必要，可以以简短指令的形式存在，这些指令要设计得与应用程序完整地结合在一起。

- **暂态 Web 应用必须简单、直接而中肯**。暂态的嵌入式 Web 应用程序没有多少回旋余地，必须确保你的实用程序以最小的花销成功地满足用户目标。

- **暂态 Web 应用应该符合用户的心智模型和 Web 站点上下文中的流**。你必须开发密切关注用户心智模型和工作流的暂态程序，不仅应该在他们的领域（domain）上下文中，而且应该在 Web 站点使用的上下文中。如果你的程序无法实现得与其他内容相符，就不要像链接广告那样链接实用工具。用户习惯于忽略任何类似于标语和在线广告的东西，特别是动画。

- **仔细考虑用户数据访问问题**。出于技术的原因，多数暂态 Web 应用程序无法在客户端保存用户数据，用户可能需要登录来检索和操作他们的数据。同样，你必须要求

用户手工保存他们输入的数据。这些操作应该是无缝的，应尽可能直观。

【电子商务 Web 应用】

当考虑事务站点的姿态时，电子商务有它的特殊性。电子商务站点的使用对于多数用户来说是短暂活动：他们来到站点（根据个人情况和场景），针对购买项目做一些数量不等的研究，然后要么购买，要么两手空空一无所获，然后一段时间都不再上来。电子商务站点很有趣，因为它们结合了 Web 站点的信息元素（以在线分类的方式）和商务事务元素。许多情况下，Web 站点也可以取代销售员，具有围绕顾客服务所必需的品牌标志和价值特性。电子商务事务必须仔细设计，因为如果在线交易过程有一点繁琐或者混乱，用户很容易受到阻碍。

电子商务站点，如 Amazon.com，在保留标准网页外观的同时，融合了一些类似应用程序的功能。这些功能包括指向最近一段时间内浏览过的项目的链接和显示小结（subtotal）的准持久购物车（但不幸的是，没有评估税费和运输费）。这些功能类型，记录和反映了用户的场景，因而简化和丰富了在线购物的体验，为用户将来购物提供了促进因素。

电子商务 Web 站点的底线是它的付款过程，它也是（并非巧合）最终使大部分用户受挫和感到迷惑的地方。设计师在创建结账界面时，应该充分利用暂态应用程序的规则。输入简单，清楚的启示和指令，视觉反馈清晰，从开始到结束的流程明显，以及用户改正错误的方法清楚，所有这些对于电子商务交易的成功和用户的满意度至关重要。

【独占姿态的 Web 应用】

独占姿态的 Web 应用应该而且正在致力于与它们的桌面同类不分彼此。这种 Web 应用程序的一个好例子是微软的 Outlook Web Access（OWA ）客户机程序。它尽可能地复制桌面版本的外观和感觉，不同的是从浏览器内部。如图 37-1 所示，OWA 在浏览器技术和拨号带宽的限制下复制了多数 Outlook 界面。

与面向页面的 Web 站点设计不同，最好的独占性 Web 应用设计方法就是将这些程序当做桌面程序。设计师也应该清楚地了解这种媒体的技术限制和开发组织的合理进度以及预算情况。和独占性桌面应用程序相似，独占性 Web 应用应该是全屏应用程序，布满了控件和数据对象。他们应该利用特殊的窗格或者其他屏幕区域，将相关的功能和对象进行分组。用户应该感觉他们是在一个环境中，而不是从一个页面导航到另一个页面，或者从一个地方导航到另一个地方。信息刷新或者重新显示的工作应该最小化（和 Web 站点的行为相反，Web 站点上几乎任何一个动作都需要完全刷新）。

将独占性 web 程序当做桌面应用程序，而不是网页集合的好处在于，这也允许设计

师打破浏览器交互面向页面模型的限制，更好地满足客户-服务器应用程序需要的复杂行为。对于获取你所需要的信息来说，Web 站点是一个很有效的地方，就像电梯能成功地将你带到建筑物的特定楼层一样，但你不会试图在电梯里做实际的工作；类似，用户也不能通过迫使他们使用浏览器，访问基于页面的 Web 站点来进行实际的事务工作。

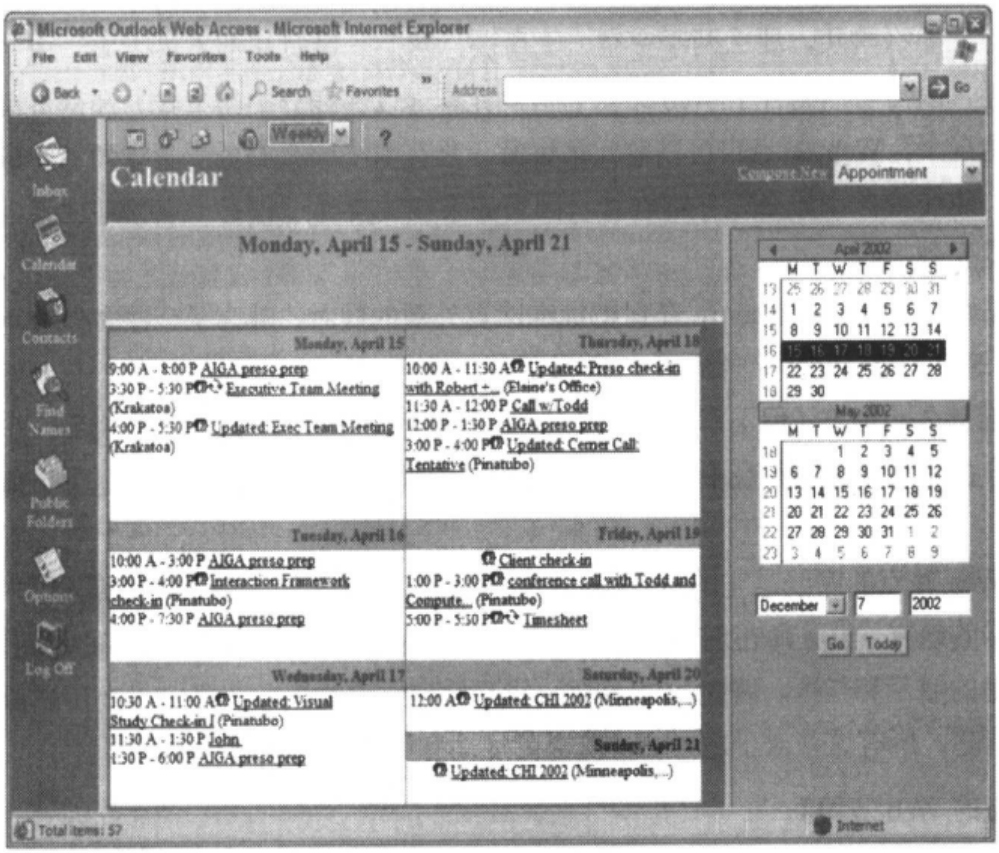


图 37-1 微软 Outlook Web Access 是一个在浏览器中运行的独占性 Web 应用软件的好例子。在 OWA 的情况下，将浏览器当做容器很有意义，因为产品的目的是允许与连接到因特网的计算机访问 Outlook。这种需要对用户并不重要或不太实际的企业应用程序，可以考虑使用浏览器外基于因特网的技术。这些技术能更好地支持丰富的交互、客户端处理和存储，这些是基于浏览器的应用程序难以支持或者不能支持的。基于浏览器的独占姿态 Web 应用设计师，也应该考虑隐藏标准的浏览器控件。浏览器的前进和后退按钮在 Web 应用程序中可能会产生不可预料的结果，从用户的视图中消除它们有利于加强应用程序的心智模型，而不是网页的心智模型。在用户桌面上创建链接或者浏览器链接工具条有助于解决初始访问问题。

【基于浏览器与基于因特网的应用程序】

创建独占性 Web 应用程序有两种可能的途径。第一种是创建基于浏览器的应用程序，实质上接管浏览器窗口，将所有浏览器控件隐藏起来，在浏览器内容区域提供丰富的交互。用于窗口的技术可以是 HTML、DHTML、JavaScript、 Flash 和 Java applet 的结合，

它们最适合这种环境。这种方法能给你布局和显示方面的自由，但会将交互限制在浏览器支持的范围。如果有多个浏览器平台需要支持，那又会带来兼容性问题。因为每个浏览器有它不可忽视的独特怪癖。这使得最小通用方法（lowest-common-denominator）成为最诱人的方法，尽管从用户的角度它不是一个好的解决方法。（这种最小通用方法并不总是必需的，后面将会讨论。）基于浏览器的应用程序也鼓励将界面限制在单个主窗口里，因为浏览器缺乏任何独立子窗口（如非模态对话框）的真正支持。

独占性 Web 应用的第二种方法是完全抛弃浏览器，而创建不基于浏览器的，基于因特网的应用程序。基于因特网的应用程序是具有 Web 意识的桌面程序，它充分利用如 Flash, Java, ActiveX, TCP/IP 或 Microsoft.Net 等多种技术，和传统 GUI 库一起使用，或者取代传统的 GUI 库。通过把应用程序放到浏览器之外，你可以在不失去 Web 上数据访问能力的前提下，提供丰富，清楚，和复杂的交互。你可以改进交互，不仅可以借助健全的 GUI 支持，没有与浏览器控件相冲突的危险，而且因为基于因特网的应用程序不受浏览器瘦客户机模型的约束。客户可以保存程序状态和数据，允许程序记忆用户行为，并且对用户的行为进行反应，存储经常使用的信息，提供独占性桌面应用程序所有其他的交互好处。

不能确定采用哪种方法的时候，该怎么办呢？这里有一些指导方法：

- 如果从任何计算机（与因特网连接）访问是你人物角色必需的特征，那么基于浏览器的应用程序是合适的。
- 如果你的客户在特殊的浏览器（特别 IE 5 或者更高版本）和/或插件方面是标准的，那么用基于浏览器的应用程序创建丰富的体验会容易得多。
- 通常，考虑基于浏览器方法的主要原因是容易安装和维护。但是，基于因特网的应用程序（例如，微软 Windows XP 和 RealNetworks 的 RealOnePlayer）能够很好地通过网络更新。设计这个过程的关键是尽可能地对用户透明。
- 基于浏览器的应用程序不太容易自动保存数据，或在客户系统中存储数据。如果你的应用程序设计需要这些行为，基于因特网的桌面应用程序更合适。支持本地数据储存也意味着服务器较少需要连接，能显著地提高应用程序的响应时间。
- 人物角色和脚本提纲是帮助你克服Web技术约束的关键技术。通过对用户建模，并且逐步逼近用户工作场景以确定用户需求，你会更深入地了解到哪种技术方法会最适合用户和顾客的实际情况。

【企业内部网、企业外部网和因特网】

正如在第 8 章中所讨论的那样，Web 站点和应用程序的行为是否恰当取决于它们的使用上下文。影响这种上下文的一个主要因素是：Web 站点是为客户通过因特网使用设

计，还是为企业内部使用设计。

在本章的前面谈到，信息和电子商务 Web 站点需要维持独占元素和暂态元素平衡的设计。这些 Web 站点应该显示适合独占性应用程序密度的信息，但是主要导航元素和事务元素应该维持暂态应用程序的简单明了，它们不会经常用到，而且用户不需要在每次会话之间记住它们的行为和控件。

另一方面，面向企业访问公司内部网或者电子商务信息和事务的软件可以采取更明确的独占姿态。这些应用程序的用户可能成为经常用户和长期用户，他们很快就会成为永久的中间用户。增加信息和功能的密度可以更加充分地利用整个屏幕的空间。如果该企业在浏览器或其他基于因特网的应用程序的技术方面实现了标准化，它可以很好地利用丰富的视觉非模态反馈和其他 GUI 习惯用法。

这种美好交互图景的一个例外是外部网设计：因为它难以预料你客户的客户的配置，所以外部网的设计要么求助于因特网风格的设计方法，要么求助于避开了浏览器兼容问题，而完全基于因特网应用程序的方法。因为许多客户知道基于浏览器的访问是合意的（我们讨论过，这只是在某些情况下是真的），至少对于现在来说，后者可能是棘手的买卖。

还有一些设计技巧可以帮助你为企业内部网、外部网和因特网创建满意的 Web 应用程序：

- 因特网站点和应用程序必须几乎总是为多个首要人物角色服务，他们典型地分为两类：定期来访者和新来/一次性来访者。因此，为广泛的因特网访问创建的 Web 站点和应用程序在独占性和暂态性之间存在冲突。这种平衡取决于精确的领域和服务上下文。它可以通过在第一部分中描述的研究和建模技术创建，清晰一致的导航，以及设计恰当的搜索会很有帮助。
- 因为基于浏览器的应用程序需要连接服务器加载数据，当用户选择某些数据部分时，试着把你的应用程序设计成批量下载相关数据。你必须小心地保持优化的一次下载数据量和批量下载期间系统响应时间之间的平衡。可能需要测试来决定（从响应时间的角度）可接受的数据块大小。
- 注意在 Web 站点上使用适合用户**心智模型**的语言，而不是**实现模型**或者以往 Web 的陈腐语言。象 Submit 这样的按钮标题不仅贬低身份而且不能提供信息。
- 当为浏览器设计时，必须知道用户经常同时打开多少个浏览器窗口。尽量将所有的交互限制在一个窗口里。
- 因为不能保证（服务器端）基于浏览器的应用程序自动保存数据，所以需要用户手工完成。你的应用与桌面应用程序越类似，对于用户来说烦恼就越多。因此，你必须采用和用户心智模型一致的方式使这种手工保存流畅完整地和应用程序流结合。

【Web 应用和视觉设计】

因为从 Web 发展的历史来看，许多商业 Web 站点开始是作为营销工具，所以价值在于 web 的消息组件。但对于 Web 应用程序来说，这种市场信息主要通过系统的行为和交互来执行，它必须反映品牌价值。在 Web 应用程序中，有使用丰富视觉效果的诱惑，但最主要的是加载时间延长了，并且干扰了程序的真实目的：满足用户目标。在 Web 应用程序中，简单明了的色调和清晰的版面比视觉醒目更重要。炫目的图像不仅转移注意力，而且用户已经习惯于忽视 Web 上那些看上去像广告或者营销手法的任何东西。

在设计过程中，交互设计师和视觉设计师需要密切合作，使用在品牌场景下工作的可视化语言，有效地表达 Web 站点的行为及功能。同样，信息设计师必须与交互设计师紧密合作，使内容与行为相匹配。

【同时包含设计和工程的努力】

在你的组织准备实现 Web 应用程序之前的最后一个告诫是：要成功地实现独占性 Web 应用程序，无论前端，还是后端，都需要经验丰富的专业程序员和软件工程师，他们不仅能沉着地应对工程难题，而且理解和支持严格的设计和工艺过程。可以迅速和相对简单地建立 Web 站点；Web 和基于因特网的应用程序需要密切注意软件体系结构，还有信息结构和交互设计。小心不要随便尝试设计或者策划一个复杂的 Web 应用程序。和任何其他种类的软件应用程序一样，那样做只会带来麻烦。

尽管 Web 应用程序在**交互范围**和**工程范围**方面比面向页面的 Web 站点会复杂得多，它给在 web 上或者通过因特网真实工作的用户带来的价值是值得付出汗水的。Web 应用程序和 Web 服务，在如.NET 这样的技术支持下，将成为 Web 的未来。和任何复杂的软件一样，它的风险也会相当大，但很好地规划和执行交互设计的方法和原理会减少这种风险，并且为用户提供更高级的在线体验。

38

嵌入式系统的设计

前一章讨论了 Web 平台上的设计问题。嵌入式系统（embedded system），或者集成在那些我们自然地认为不是计算机的设备（如手机、电视机、微波炉、汽车仪表板、照相机、银行机器，以及实验设备）中的软件系统，有它们自身的机会和限制。设计嵌入式系统时，你必须知道这些。没有精心的设计，向设备和电器中添加数字智能只会导致产品的行为更像桌面计算机，而不是用户所期望和需要的设备（Cooper,1999）。

本章讨论嵌入式系统的类型，从目标导向设计的角度介绍这些设备一些有用的设计原理。

一般设计原则

尽管嵌入式系统包含软件交互，但它们有一些与桌面系统不同的独特性质。在设计任何嵌入式系统的时候，无论是智能电器（smart appliance），公用信息亭系统（kiosk system）或者手持设备（handheld device），都必须谨记这些基本原理：

- 不要把你的产品当成计算机。
- 软件设计和硬件设计相结合。

- 上下文驱动设计。
- 明智地使用模态。
- 限制范围。
- 平衡导航和显示密度。
- 为你的平台定制。

接下来我们对这些原理逐一进行讨论。

不要把你的产品当成计算机

也许在设计嵌入式系统时应该遵循的最重要原则就是，你正在设计的不是计算机，尽管它的界面类似于计算机的位图显示（bitmap display）。用户对你的产品功能有着特殊的期待（如果它是电器或者熟悉的手持设备），或者期待很少（如果你设计的是公用信息亭系统）。你不能将桌面计算机世界中的所有行李——习惯用法和术语——带到像照相机或者微波炉这样的简单家用电中来。同样，科学和其他技术设备的用户希望迅速而直接地访问他们领域里的数据和控件，而不是通过计算机操作系统或者文件系统艰难地寻找。

程序员，特别是那些设计桌面平台的程序员，即使他们在设计时也很容易忘记，他们不是在为通常意义上的计算机设计：彩色的大屏幕，桌面隐喻，大量电源与内存，完整尺寸的键盘和鼠标设备。对于多数嵌入式设备来说，这些假设，即使有的话，也很少是有效的。例如，电视遥控上的菜单按钮有意义吗？当然，你需要访问功能，但菜单是来自计算机世界的一个术语，而不是电视机。同样，“取消”这个术语对于关闭微波炉计时器也不合适。

比将嵌入式系统硬塞进计算机界面更好的方法是了解你设计这种设备的目的，然后再考虑如何应用数字化技术，以加强用户体验。微软在 PocketPC 界面上有很多教训，不过和所有的微软产品一样，每一个新版本都在改善。

结合硬件和软件设计

从交互的观点看，嵌入式系统的一个定义特征是，在界面中软件和硬件部分常常紧密相关。不像桌面计算机用户关注的重点是高分辨率的彩色大屏幕，与之不同的是，多数嵌入式系统提供的支配用户注意力的硬件控件必须与用户任务和谐地结合。由于成本、能力和形态要素（form factor）的限制，基于硬件的导航和输入控件经常必须替代屏幕，因此它们需要特别裁剪，以适应界面软件部分，用户目标和人机工程学（ergonomic）的

需要。

因此关键是要从目标导向和人机工程学的角度同时设计系统界面的硬件和软件元素，以及它们之间的交互。在标准的开发过程中这种情况很少发生，一般是硬件工程师团队将完整的机械和工业设计交给软件工程师团队，软件工程师团队必须适应他们，而不管从用户的角度看这是不是最好的。

作为一名设计人员，你必须说服你的团队在硬件平台完成之前开始界面设计过程。现在可以获得的顶尖、最具创新意义的数字设备（如 Handspring Treo, Apple iPod）中的大多数都出自这种全盘考虑，无缝地结合硬件和软件为用户创建高效、引人注目的体验（见图 38-1）。

图 38-1 此图为 Cooper 设计的一个智能桌面电话，体现了硬件和软件控件的完美结合。用户可以使用触摸屏（touch screen）和拇指轮（thumbwheel）轻松地调节音量/扩音器，拨新号码，用硬件控制语音信息重放，以及管理已知的联系方式/电话号码、输入电话，电话记录，语音信息和会议功能。设计力求使最常用和最重要的电话特征更容易使用，而不是在系统里加载太多的功能。注意屏幕上有手指大小的触摸区，并使用文本提示加强了交互。

上下文驱动设计

嵌入式系统和桌面应用程序之间另一个明显的区别在于上下文环境的重要性。虽然桌面应用程序也关注上下文问题，但设计师通常认为多数在桌面上运行的软件会在相对安静和私人位置的计算机上使用。尽管这已经变得不那么真实，因为膝上型电脑（laptop）既有桌面电脑的功效，又有无线接收功能，，但是仍然存在这种情况，用户出于形态要素的需要，即使使用膝上型电脑时也在固定而安静的环境中。

许多嵌入式系统与这种情况恰恰相反，它们或者为随身携带使用（手持设备），或者固定在公共活动场所（如公用公用信息亭）。甚至是绝大多数固定和隔绝的嵌入式系统（如家用电器）也有很强烈的场景元素：主人为宴会准备热菜势必会分散精力，一个智能微波炉绝不能需要大量控件导航。同样，生产电梯的技师不必在测试设备控件上花费太多精力——这种干扰在某种情况下可能会威胁生命安全。

因此嵌入式系统设计必须紧密适应使用的上下文环境。对于手持设备来说，环境就是设备手持的位置和操作方式。它怎样拿着的？是一只手还是两只手使用的设备？在不立即使用的时候放在哪里？用户在使用设备的同时还会从事什么其他活动？在什么环境下使用？是喧闹的地方，还是比较明亮或者比较黑暗的地方？如果在公共场合使用，用户对别人看见或者听到这种设备的使用会怎么想？稍后我们会进一步讨论这些问题。

对于公用信息亭，上下文关注主要集中在公用信息亭的物理环境及社会问题：公用信息亭在环境中扮演什么角色？公用信息亭是否位于交通要道？它是否提供辅助的信息，或者它是否是主要的吸引源？在合适的时候建筑环境是否会引导人们找到公用信息亭？每次有多少人想使用公用信息亭？是否有足够数量的公用信息亭而无需长时间的等待？对于公用信息亭和公用信息亭交通在不影响用户交通的情况下是否有足够的空间？我们会简短地谈到这些问题和一些其他问题。

明智地使用模态

桌面计算机应用经常有丰富的模态：软件可能处于许多不同的状态，在不同的状态下输入控件和其他控件映射为不同的行为。工具调色板就是很好的例子：选择一个工具，鼠标和键盘活动映射为特定工具指定的功能；选择另一个新工具，同样的输入将产生不同的行为。

多数嵌入式系统出于以下这些原因很难支持大量的模态：

- 屏幕空间有限，因此改变模态的能力有限。
- 嵌入式系统的用户（就像公用信息亭的情况那样）常常是新手用户，而不是中间用户，没有时间去熟悉模态以及模态之间的导航。
- 在输入机制受限时，使用模态产生导航附加工作。

嵌入系统设计总体上应该避免模态的使用。如果必须使用模态，在数量上也应该受到限制；理想的模态切换应该由使用场景的条件改变决定。例如，对于 PDA/电话接收设备，当收到一个电话时转换到电话模态，当电话挂断后又恢复初始模态是有意义的（允许打电话的同时访问其他数据是一个更好的选择。）。如果确实需要模态的话，在界面中必须能够清晰地访问，并且退出途径必须明了而快捷。在很多 Palm OS 手持设备中有四个硬件应用程序按钮，这就是清楚标记模态的好例子。

相反，很多手机对过多的模态提供很少的导航，层次关系结构组成的模态通常需要多次按键访问。手机的大多数用户只使用拨号和访问电话簿功能，如果他们试图访问其他功能，很快就迷路了。甚至像静音这样的重要功能，一般电话用户也常常不能掌握。

通过在电源按钮的旁边提供明显标记的铃声模态开关就能轻松解决这个问题，每个手机都应该模仿这种界面特征。

模态的可用性完全取决于输入和显示机制。与桌面计算机比较，嵌入式系统的输入和显示设备明显更小，而且更少。这就意味着，在嵌入式系统设备中实现的模态在理解和导航方面存在困难。因此模态应该在数量上减少，并且降低复杂度。

限制范围

大多数嵌入式系统都用于特殊的场景和特殊的目的。不要把这些系统当做通用计算机。与在一个地方完成太多完全不同的任务的设备相比，能够更有效地完成一系列特定任务的设备能够更好地为用户服务。像微软 Pocket PC 这样的设备，后来试图模仿完整的桌面系统，甘愿冒着界面繁杂充满各种功能疏远用户的风险，仅仅因为这些功能当前在桌面系统中存在。

许多设备与桌面系统共享信息。从以桌面为中心的观点设计这种系统是有意义的：这些设备是桌面系统的延伸和附属设备，在桌面系统不能使用的地方提供关键的信息和功能。对于这些辅助设备，脚本提纲可以帮助你决定什么功能是有用的。

平衡导航与显示密度

大多数嵌入式系统（公用信息亭除外）受到有限的显示空间限制。手持设备受到形态要素和电源的限制。对于电器来说，成本通常是决定因素。无论什么原因，设计师创建的设计必须充分利用可用的显示技术，同时满足用户的信息需求。尽管大显示屏的布局可以松散一些，但是对于嵌入式系统来说，每像素和每平方毫米的显示都是重要的。这种显示空间的限制需要权衡显示信息的清晰度和导航的复杂性。通过恰当地限制功能范围，你可以稍微改善这种情况，但是显示和导航之间总是存在某些程度的紧张。

你必须仔细地规划嵌入式系统的信息显示，创建信息的层次关系。确定什么是最重要的信息，使这种功能最显眼，然后再看哪些附属信息适合在屏幕上显示。尽量避免通过屏幕闪烁来显示不同信息。例如，带有数字控件的微波炉通过两个数值间的闪烁来显示你设置要达到的温度及到达指定温度的距离。但是，这种方法容易混淆每个数字的准确含义。一个更好的解决方法是微波炉显示设定的温度，在它旁边用一个小的条形图表示微波炉此时的温度与期望温度的距离。你还必须留出空间显示相关硬件控件的状态；或者更好的方法是使用能显示自身状态的控件，比如带指示灯的控件按钮，或者使用可

以维持物理状态的控件（例如，双态元件（toggle）、开关（switch）、滑块（slider）、旋钮（knob）即可）。

将输入复杂性最小化

几乎所有嵌入式系统的共同特点是只有简单的输入系统，而不是键盘或者桌面风格的指向设备。这意味着任何输入系统——特别是文本输入——对于用户来说都是笨拙的，速度缓慢而困难。甚至最复杂的输入系统——触摸屏、语音识别、手写识别和与之相关的拇指板（thumboard）——与完整的键盘和鼠标相比都更笨拙。因此尽可能限制和简化输入非常重要。

像 RIM 的 Blackberry 和 Danger Hiptop 设备有效地利用拇指轮（thumbwheel）作为主要的选择机制：快速地翻滚可能的选择，按压拇指轮（或附近的按钮）选择给定项。当需要输入文本数据的时候，这两种设备也充分利用了拇指板。

与之相反，Handspring 的 Treo 则是利用触摸屏和拇指板。如果你可以用手指接触有效地激活 Treo 屏，这将是很有效的。但是，多数 Palm 屏幕上的小控件太小，需要你用指示笔（stylus）来做精确选择。这意味着你必须在指示笔和拇指板之间切换，使输入增加了不必要的麻烦。Palm 的 Tungsten 设备用附加的四个方向的方向垫（four-way directional pad）和选择按钮来解决类似的问题，它支持不使用指示笔的屏幕控件导航。

公用信息亭的屏幕通常很大，虽然如此也应该尽可能避免文本输入。如果触摸屏足够大，它可以显示软键盘；每个虚拟键应该大到不让用户输入错误。触摸屏也应该避免拖动相关的习惯用法；单叩（single-tap）习惯用法对于新手用户来说更容易，控制也更明显。

为你的平台定制

在桌面领域，软件开发商有时试图开发跨越多个平台的界面，例如，程序外观和行为与 Mac OS，Windows 和 Unix 相同（或相近）。这不仅需要程序员大量的编程工作开销，而且最终也得不到用户的认可。用户宁愿使用他们本地平台的应用，本地平台的应用会充分利用适合他们个人工作环境的独特特征、设备和界面习惯用法。

对于嵌入式系统软件，问题更复杂了。直接将桌面产品移植到设备平台是不可能的，对于不同的操作系统和硬件的设备，企图开发跨平台界面同样是徒劳的。在嵌入式系统领域，直接针对每个平台编程和定制最简单。

手持设备的设计

手持设备向交互设计师提出了特殊的挑战。因为它们是特别为满足移动用途而设计的，所以必须小巧，重量轻，电源消耗小，朴实，在忙而有干扰的情况下容易拿着和操作。特别是手持设备，交互设计师、工业设计师、程序员和机械工程师进行紧密合作是非常必要的。尤其需要关注的是显示屏的大小和清晰度，容易输入和控制，对上下文环境敏感。本部分详细讨论这些焦点以及解决这些焦点问题的有效方法。以下是设计手持设备最有用的交互和界面原则：

- **考虑设备会怎样手持和携带**。物理模型对理解设备怎样操作是很重要的。模型至少应该反映设备的大小、形状和结构（翻盖等），将重量也考虑进去，模型会更有效。设计师在上下文中和场景的关键路径脚本提纲中使用这些模型来确认建议的**形态要素**。
- **尽早决定设备是一只手操作还是两只手操作**。脚本提纲应该明确用户在不同的上下文中可接受哪些模态。只要不是常用的功能，将主要用于单手使用的设备支持一些需要双手使用的高级功能也可以。例如，一个手持的账目工具，允许所有用单手操作的计算但需要双手提交输入数据是无益的，因为提交功能是常用脚本提纲的一部分。
- **考虑设备是作为辅助设备还是独立设备**。多数手持数据设备是作为桌面数据系统的辅助设备设计的。Palm 和 Symbian 设备都是最成功的桌面系统或服务器通信便携系统。它们不是复制所有的桌面功能，而主要用于访问和浏览信息，只提供少量输入和编辑特征（完全大小的可折叠键盘也可以用于这种系统；但事实上，在使用键盘的时候这些设备又转换成非常紧凑的桌面系统）。最新的手持设备将附属概念扩展到无线连接领域，使附属设备理念更加合适和功能强大。

另一方面，一些设备比如标准的手机，要设计成真正的独立设备。从 PC 机到手机下载电话号码是可能的，但是多数用户因交互复杂从来不会这么做。这种独立设备最成功的地方就是关注范围较小的功能，但又为那些功能提供一流的行为。RIM 的 Blackberry 是小范围内关注，数据为中心的手持设备的最好例子；它擅长实时无线接收，浏览和发送企业电子邮件。Blackberry 设备包括其他 PIM（个人信息管理）功能的支持，但是这些功能不太常用，是无线电子邮件定义了该产品的特征。

- **避免使用多用途窗口和弹出窗口**。在小的低分辨率屏幕上，一般来说不能使用悬浮窗口。在这点上，界面应该像独占性应用程序（见第 8 章），利用屏幕全部的空间。无论如何应该避免使用非模态对话框，如果可能，模态对话框和错误对话框应该尽量用第 33 章和第 34 章中讨论的技术取代。
- **努力集成功能使导航最小化**。手持设备用于各种特殊的上下文场景。通过研究

上下文场景，你可以清楚地了解需要集成什么功能，以提供无缝的**目标导向**体验。

大多数汇聚多种功能的设备，因试图做的事太多，而冒着没有一个人满意的风险。如 Treo 这样的通信器，在集成无缝的通信相关功能体验方面是最好的。如今这些设备集成电话和地址簿是合理的：当接进一个电话，你可以从地址簿中看见全名；你还可以通过选择电话簿中的名字直接拨号。但是，这种集成可以更进一步。单击地址簿中的名字可以显示所有通信器知道的文档：与这个人相关的约会、电子邮件、电话日志，包括呼叫者名字的备忘录，以及与他相关的网站等。单击其中一个文档，可以将你带到与那个人相对应的界面。通信器最新的应用程序，如 iambic 有限公司的 Agendus，开始采用这种方法将那些曾经不同的应用程序集成到符合用户目标的无缝流中。

- **屏幕上的控件应该大而明亮**。如果在你的设备中使用触摸屏，控件应该大到可以用于指接触指示。笔也可能会迷失，因此少年用户常在使用笔时扔掉它。手持屏幕常处于节能状态，亮度和对比度较低，所以在某些照明条件下，小型的图标和控件也很难看见。最后，因为手持设备的界面必须划分为几个屏幕（在特定的场景中只能使用其中一部分），所以手持设备的屏幕应该模拟暂时姿态应用程序（见第 8 章），采用大而对比强烈的彩色控件（如果条件允许）。

- **使用大号的无衬线字体（sans-serif font）**。衬线字体（serif font）在分辨率低的情况下难于阅读；无衬线字体应该用于低分辨率的手持显示器。

- **无须拖动**。触摸屏可以支持拖动，但是用户操作起来比较困难，也有刮坏屏幕的危险。理想的硬件控件，如 RIM 的 Blackberry 拇指轮或者 Palm 的向上向下按钮应该是滚动的主要机制。拖放习惯用法也应该避免使用。

- **无须转换输入模态**。如前所述，输入应该尽可能简单。当设备使用键盘或者拇指板的时候，它应该不需要在使用单手、双手或者用指示笔轻叩屏幕之间进行繁杂的切换。允许两种类型的独立输入是好的，但是在使用过程中需要不断转换输入模态可能让用户感到厌烦。

- **当在屏幕外有更多数据的时候应该有明显的提示**。许多人对需要滚动才能看到信息的小屏幕还不习惯。如果有超过一屏的数据，确保有醒目的提示表明有更多的数据可用，最好是关于如何访问的提示。

设计公用信息亭

表面上，公用信息亭和桌面界面有许多共同点：大的彩色屏幕和背后强大的处理器。但是就用户交互而言就不同了。与独占性桌面应用程序用户相比较，用户使用公用信息

亭的次数有限，对于特定的公用信息亭来说，更典型的情况是可能只会使用一次。而且用户在使用公用信息亭的时候，要么有一个非常具体的目标，要么没有直接目标。公用信息亭用户一般不会访问键盘或者指向设备。即使他们访问了，也不能有效地使用。最后，公用信息亭用户一般是在公共场合，充满喧闹和干扰的环境下，可能和其他的同伴一前一后地使用公用信息亭。每种环境都给公用信息亭的设计施加了压力。

公用信息亭的姿态和导航

因为公用信息亭的用户永远是不确定的，主要用于一个较短的时间段，界面应该侧重于第 8 章描述的暂态界面。这意味着大而色彩丰富的界面，有清晰的控件启示，硬件控件（如果有）和它们对应的软件功能有清晰的映射关系，导航需求最小化。和设计手持设备一样，悬浮窗口和对话框应该避免；任何信息或行为最好集成在单一全屏中（作为独占性的应用程序）。因此，在两种最常见的桌面姿态之间，公用信息亭处在一个有趣的中间地带。

因为公用信息亭常通过一屏接一屏的信息或过程引导用户，上下文导航比整体导航更重要。重要的导航范例是“我从这里可以到哪里去？”，而不是“我可以去哪？”也就是说，对于主要目的是事务的公用信息亭来说，重要的是提供逃逸出口，允许用户撤销事务和在任意位置开始。

事务与探索

通常公用信息亭分为两种类型：事务型和探索型。事务型的公用信息亭提供一定范围内的事务或服务。这些包括银行取款机（ATM 机），用于飞机场、火车或公交车站和一些影院的售票机。甚至自动贩卖机也可以看做是一种简单的事务型公用信息亭。事务型公用信息亭的用户心中有特定的目的：取得现金，买到票，Tootsie Roll 糖果，或者某条具体信息。这些用户只对尽可能快速而无障碍地完成目标感兴趣。

探索型公用信息亭在博物馆中最常见。面向教育和娱乐的公用信息亭不是主要的吸引源，但是它为用户提供附加的信息和更丰富的体验，用户可以看到其他可能相关的（但不是必需的）展览。探索型公用信息亭与事务型公用信息亭不同，用户在使用事务型公用信息亭时一般没有什么期望。但用户在使用探索型公用信息亭时，他们很好奇，或者很希望获得愉悦或者启发，而没有明确的目标。对于探索型公用信息亭，探索行为必须由用户执行，所以公用信息亭的界面不仅要有清楚和易于掌握的导航，而且必须具有艺

术美感，视觉上（和听觉上）使用户感到兴奋。每个屏幕必须有趣，而且鼓励用户对系统进一步探索。

公共环境下的交互

事务型公用信息亭，通常不需要用特殊的诱惑来吸引用户。但是，它们需要放在理想的地点，既显眼，又能很好处理它产生的用户交通流量。使用路标和信号系统将这些公用信息亭连接起来是最有效的方法。某些事务型公用信息亭，特别是 ATM 机，需要考虑安全问题。如果它们的位置可能不安全（或者事实上不安全），用户不会使用它们，或者冒险使用它们。事务型公用信息亭的结构规划应该和交互及工业设计规划同时进行。

和事务型公用信息亭一样，探索型公用信息亭也应该放在理想的地点，与路标系统一起使用。它们一定不能妨碍任何吸引源，但必须和相关的吸引源足够近，必须有足够的空间让人们聚集：探索型公用信息亭最可能被一群人使用（如一家人）。特殊的挑战在于在一个位置安装恰当数量的公用信息亭——公司常根据用户流量调查来决定事务型公用信息亭的理想数量。人们不会在事务型公用信息亭滞留太长时间，他们也常常愿意排队等待，因为他们有具体的最终目的。探索型公用信息亭则不同，它鼓励逗留，这使得它对旁观者没有吸引力。因为潜在的用户对于探索型公用信息亭的内容没有期待，要让他们排队等待使用一台机器是困难的。设想多数人只有在探索型公用信息亭有空时才能使用是可靠的。

当设计公用信息亭界面时，要仔细考虑声音的使用。探索型公用信息亭为用户提供丰富的听觉反馈和内容似乎很自然，但是音量应该进行限制，不至于侵占这种公用信息亭支持的主要吸引体验。听觉反馈应该少量应用于事务型公用信息亭；但它也是有用的，例如帮助提醒用户取回银行卡或者更改交易。

管理输入

大多数公用信息亭要么使用触摸屏，要么使用映射为屏幕对象和功能的硬件按钮和键盘。在触摸屏的情况下，和其他触摸屏界面有相同的使用原则：

- 可触摸的对象应该足够大，能用手指操作，对象应该对比明显，色彩鲜艳，在屏幕上很好区分，能很好地避免无意的选择。
- 在带有触摸屏的公用信息亭使用屏幕上的键盘（on-screen keyboard）输入数据很有诱惑力。但是，这种输入机制应该是最后手段——例如当需要密码时（这种情况下

键盘可能更有效）或者某些不能从列表选择的其他信息。

- 应该避免使用拖放习惯用法，除非在绝对需要的时候，否则任何一种滚动的操作都应该避免。

因为触摸屏更昂贵，也更容易错误，一些公用信息亭使用映射为屏幕功能（on-screen function）的硬件按钮取代触摸屏。和手持系统一样，关键的问题在于保持映射的一致性，不同屏幕上的相似功能对应相同的按钮。这些按钮不应该离屏幕太远，或者立体地布置，那样映射就不明显了（有关映射问题讨论的更多细节见第 11 章）。

听觉界面的设计

听觉界面，如语音信息系统和自动呼叫中心的界面，存在一些特殊的挑战。导航是最重要的挑战，因为没有视觉工具，在层次关系的功能树中很容易迷路（这些系统几乎总是存在功能树）。以下是设计可用听觉界面的一些简单原则：

- **根据用户的心智模型组织和命名功能**。在任何设计中这都是重要的，但是当功能只是进行口头描述，以及仅存在于在当前功能的上下文中时，就更加重要。要研究上下文场景决定哪种功能是最重要的，并且使它们最容易获取。
- **总是标明当前可用的功能**。系统应该在每个用户动作之后，重申当前可用的功能，并且说明怎样激活它们。
- **总是提供返回和前进的方法**。界面应该在每个动作之后，告诉用户怎样返回到功能结构的前一步（通常是功能树的上一个结点），以及如何到达功能树的更顶端层次。
- **总是提供与人说话的方法**。如果可能，界面应该在每个动作之后，尤其在用户似乎有困难时，向用户介绍如何转向人类助手。
- **给用户提供足够的反应时间**。系统通常需要口头或电话键盘输入信息。应该进行测试来决定恰当的等待时间。记住：用电话键盘输入文本会非常不灵活，也非常慢。

附录 A　本书公理集

1　目标导向设计

交互设计不是凭空猜测。

2　实现模型和心智模型

用户界面应该避免实现模型，而支持用户心智模型。

目标导向的交互反映了用户的心智模型。

不要在界面上只复制机械时代的产品，而不进行信息时代的加强。

重要的改变必须比原来更好。

3　新手、专家和中间用户

没有人愿意停留在新手级别。

为中间用户优化。

将用户想像成非常聪明但非常忙的人。

5　用户建模：人物角色和目标

不要让用户觉得自己很愚蠢。

每一个界面的设计都是为了某个专一的、主要的人物角色。

8　软件姿态

独占式应用程序的用户是永久的中间用户。

为全屏使用优化独占式应用程序。

独占式界面应该使用保守的视觉风格。

9　和谐与流

无论你的界面有多酷，越少越好。

和谐的用户界面是透明的。

为很可能发生的情况设计，考虑可能存在的情况。

请求宽恕，而不是许可。

提问不等同于提供选择。

隐藏弹射座椅控制杆。

10 消除附加工作

消除附加工作使得用户更加有效率。

不要固定培训工具。

不要极端愚蠢地中止用户进程。

别让用户申请许可。

在有输出的地方允许输入。

11 导航和调整

为典型的导航调整界面。

如果回报值得，用户愿意付出相称的努力。

13 重新思考“Files”和“Save”

磁盘和文件不能帮助用户实现他们的目标。

14 设计体贴的软件

软件应该体贴。

15 设计智能的软件

计算机工作，用户思考。

如果值得用户输入，就值得程序记住。

17 改进数据输入

错误可能不是你犯的，但保护用户是你的责任。

检查，而不校正。

19 外观设计

可视界面基于视觉模式。

在视觉上区分不同行为的元素。

可视化地传达功能与行为。

可视化地显示大致内容，文本化地显示具体对象。

除非有真正出众的可选方法，否则遵守标准。

一致性并不意味着僵化。

20 隐喻、习惯用法和启示

用户喜欢成功，而不是知识渊博。

所有的习惯用法都需要学习；好的习惯用法只需要学一次。

不要让你的界面屈从某个隐喻。

21 直接操作和定点设备

丰富的视觉交互是直接操作成功的关键。

在视觉上暗示受范性。

24 操作控件、对象和连接

提供取消拖动的方法，并向用户明确表示。

25 窗口行为

对话框是另一个房间，去之前要有个好理由。

任何交互习惯用法的效果都和上下文相关。

26 使用控件

大多数布满控件的对话框并不是好的用户界面设计。

绝不要水平滚动文本。

为有界输入使用有界控件。

27 菜单：教学向量

用菜单和对话框提供教学向量。

29 使用工具条和工具提示

在所有工具条和图标控件上使用工具提示。

30 使用对话框

将主要的交互放在主窗口中。

对话框打断了流。

31 对话框礼节

所有的习惯用法都有实践性约束。

33 消除错误

用户界面不应该是肤浅的。

尽可能使错误不可能出现。

当软件告诉用户失败了时，用户会觉得很没面子。

计算机内部没有什么危急时刻值得羞辱人类。

34 通知和确认

做，不要问。

让所有的动作都可以撤销。

提供非模态反馈来帮助避免用户犯错误。

附录 B　本书设计技巧集

2　实现模型和心智模型

用户不理解布尔逻辑。

5　用户建模：人物角色和目标

在设计的早期阶段，假设界面有魔术效应

8　软件姿态

独占式应用程序可以使用丰富的输入。

在独占式应用程序中，让文档视图最大化。

暂时式应用程序必须简单清晰，并且切中要点。

暂时式应用程序只使用一个窗口和视图。

9　和谐与流

不要使用对话框报告常规状态。

13　重新思考“Files”和“Save”

自动保存文档和设置。

将文件放在用户能够找到的地方。

磁盘是黑客手段，而不是设计特性。

18　为不同的需要进行设计

在帮助菜单中提供快捷方式。

为用户提供好的解决方案模板库。

21　直接操作和定点设备

对移动和选择任务应该同时提供鼠标和键盘操作。

（一次单击）选择数据或改变控件状态。

双击意味着单击加动作。

在数据上鼠标按下意味着选择

在控件上鼠标按下意味着预备动作；鼠标释放意味执行动作。

显示受范性是光标暗示最重要的作用。

用光标暗示表明元键的含义。

22 选择

使选择在视觉上明确醒目。

用系统加亮颜色来显示选择。

23 拖放

拖放候选对象必须在视觉上显示它们的接受能力。

拖动光标必须在视觉上表示源对象。

任何可滚动的拖放目标对象都必须支持自动滚动。

对所有拖动去颤动。

任何要求精确对齐的程序必须提供游标工具。

24 操作控件、对象和连接

用合击取消拖动。

25 窗口行为

将功能创建在需要使用它们的窗口中。

26 使用控件

用生动的图标区分重要的文本项。

为仅供输出的文本用非编辑控件（显示控件）显示。

28 使用菜单

禁用不适用的菜单项。

在平行的命令向量中使用相同的视觉符号。

29 使用工具条和工具提示

工具条为有经验的用户提供快速访问常用功能的途径。

30 使用对话框

绝不创建系统模态对话框。

在视觉上区分模态对话框和非模态对话框。

为非模态对话框提供一致的终止命令。

绝不要动态改变终止按钮的标签。

当程序将变成无响应状态时，必须通知用户。

绝不要用临时对话框作为错误信息框或确认信息框。

31 对话框礼节

所有的对话框都应该有标题栏。

在功能对话框的标题栏中使用动词。

在属性对话框标题栏中恰当地使用对象名。

对话框应该尽可能小，但不能再小。

为所有的模态对话框提供确认和取消按钮。

在对话框文本中决不使用与终止命令相同的单词。

在模态对话框中不要设置关闭框。

将终止按钮设置在标签对话框的非标签区域。

不要堆叠标签。

33 消除错误

错误信息框停止进程，简直是白痴。

跋：给同行的话

自 1995 年本书第一版出版以来，数字产品世界当然发生了改变。Web 改变了我们访问信息的方式，.com 也是几度盛衰。正如 Moore 定律所说的那样，硬件运算速度最先得到提高，其次存储容量大幅度增加。手持数据设备，曾经是稀罕物，现在已是无处不在，而且开始进入无线电话和数码相机领域。

但是所有这些新鲜出炉的伟大技术的存在，也没有显著的改善我们的生活。计算机仍然很难使用。照相机和电话有时出现崩溃，需要重新启动，在你希望打电话或者拍照时，却做不到。现在我们的 DVD 播放器和 VCR（家庭摄像机）在功率损耗后，还闪烁着 12:00。

我们什么时候才能意识到那不是技术的错，而是产品行为的缺陷？难道仅仅因为硬件和宽带的进步，情况就会不同吗？愚蠢、粗鲁、不合时宜的软件和基于软件的产品的老问题仍然存在，和以前一样。

最终，我们通过研究和满足用户目标能改善数字产品。仅仅依靠将产品转移到新的平台或者提高技术不能在根本上改进产品。我们的技术已经非常先进。1995 年缺乏的，现在仍然缺乏，那就是对人类更好的关怀。

需要的是不止是更伟大的设计。我们需要的是孕育伟大设计的产品开发组织。设计师不仅要成为为用户服务的倡导者，还要成为组织内部革新的倡导者。基于技术限制这种陈旧假设的旧思维模式已经渗透到许多产品开发组织内部。这是职业人员必须积极采取步骤加以克服的。

匮乏论

从 PARC 范例开始我们已优化了半个世纪。我们知道怎样创建好的错误消息、确认对话框和提示。但是创建支持用户的丰富而统一的视觉界面还没有多少经验。对于创建健壮的数据一致性和完善层次关系的文件系统，我们已经有数年的经验，但是创建数据免疫系统和基于属性的检索系统，我们还毫无经验。

问题很简单：我们对计算机所有的了解都是错误的！五十年前，地球上所有的计算能力还不如今天的一只手表。如今你的家庭轿车的计算能力超过了当时的航天飞机。25 年前，计算机是珍贵的商品：极其昂贵、数量有限、能力也差。1975 年，代表技术发展水平的 IBM 370/135 大型机的主内存只有 144KB。它有两个 100 MB 的硬盘驱动器，每

个驱动器都有电冰箱那样大。它还有一个读卡机，一个链式打印机。它安置在自己的房间里，深藏于特制的建筑。

计算资源总是非常缺乏。内存、存储、CPU 周期、带宽从来就不够。如果你想精通你所做的，你必须努力将匮乏的资源最大化——你必须确保 CPU 获得所有的中断。你开发的系统最大化地利用磁盘、RAM，甚至是穿孔卡。

经历了那个时代的学术界和商业界的有经验的程序员和计算机专家，习惯从计算机资源的方面考虑问题：匮乏论。正是这些骨子里深知计算机资源匮乏的人仍然引领当今的软件产业。

丰富论

在短短的 20 年以内，我们将匮乏的世界抛在了身后，进入了丰富的世界，甚至超越了我们的视线。我们的计算机和我们所希望的那样功能强大。我们拥有设计真正为人们服务的软件所需的所有比特、字节以及 CPU 周期。

与匮乏论相对的是丰富论，好的交互设计师有这种敏感性。丰富论将设计师从对内存，存储或 CPU 周期的担忧中解脱出来。相反，他们所要关心的是用户，他们有创建提高用户效率的界面和行为的设计天赋和培训。

如何改变人们关于这些问题的观点？一个方法来自权威的说服。采用专家的视点可能只是起步，采取的步骤还包括逐步积累经验或者经过专门训练。

培训成为一个设计师

交互设计师通常不是来自各级程序员，尽管如此，但他们也是精通技术的人。非技术人员无法想像计算机能为我们做的美好新事物。非技术人员无法理解 CPU 的时间与用户下次击键之前计算机迅速地完成千万次指令之间的精细平衡。非技术人员会让我们的计算机像过去那样糟糕地对待我们，但却有可爱的图片。计算机究竟能为我们做什么并不是显而易见的。它需要天才、技能、训练和经验。

寻求专业训练

在过去的几年，各国已经开始计划设置交互设计的课程。即使这样，交互设计作为一门新兴学科在学术设置上仍然受到局限。全世界仅有几个学院提供专门的交互设计研究生课程。尽管它们课程类似，但没有实现标准化（有充分的理由证明标准化是件好事）。在更大的学院，这些课程中的大多数以及其他计算机相关课程：人机交互（HCI）或新媒

体设计要么从艺术系要么从技术系（常是建筑或者计算机科学系）派生。它们各有其自身的历史，观点和教学方法的理念。

（尽管令人高兴的是这些已经开始在改变）但是在学术界对于交互设计课程应该以什么为核心元素以及如何教授这些课程仍没有达成一致意见。艺术学校主张交互设计方法作为个性或者品牌表达的方式，而不是解决产品定义和可用性问题的方法；技术系认为应该从探索和实现技术的角度，而不是发现和解决人类目标的角度教授交互设计这门学科。强调 HCI 技术的课程倾向于注重认知理论和用户研究，而不重视设计方法和实践（设计工艺）。许多设计项目仍然重视工具，而不是方法，但正在发生改变。

多数大学仍然在教授经验主义的可用性工程和 HCI，而不是教授设计。但这也正在改变，每天越来越多的交互设计师从大学毕业。

许多拥有新的或已经制定了交互设计和 HCI 课程的学院开始认识交互设计以及交互设计师需要的素质和技能。拥有先进思想的这些学院包括：

- Carnegie Mellon 大学；
- Illinois 工艺学院，设计学院；
- Ivrea 交互设计学院；
- New York 大学，电讯交互项目；
- North Carolina State 大学；
- Helsinki 艺术设计大学；
- California 大学，伯克利设计学院；
- Virginia 联邦大学。

其他途径

从事交互设计你真的需要硕士或者 Ph.D.学位吗？严格而广泛（艺术、商务、人文学科和自然科学）的科班训练的优势是肯定的。但是一些东西，在任何学科中也不能轻易教授。对用户的同理心和抽象概括能力（将它们冷静地提炼出来）是教不会的技能。一些雇主寻找有这样的天才的人，无论他们的正式学历怎样。

如果你想选择交互设计作为可能的事业，有一些事情你需谨记：

- 设计师很少编码——如果你喜欢编程，你具有所有的能力；世界需要更多设计敏感的程序员。但是除非你可以完全控制你的项目，否则你同时设计和开发将对不起你的用户——这存在利益冲突。所以，如果你不能容忍放弃编程的想法，交互设计可能不适合你。
- 可用性研究是极其重要的，但是它不是设计。它能确定问题所在，却不能解决

问题（除了在一些细节问题上）。你是能够想象、广泛提炼和详细解决问题，还是更擅长从已知的状态中提取事实？如果后者更适合你，可用性研究可能更是你兴趣的重点。

- 气质是重要的。最好的交互设计师对所有事都感兴趣，愿意（甚至渴望）沉浸在自己不熟悉的领域学习和吸收有用的元素。他们也非常关心个体的人和总体的人性条件。

- 所有的设计师都需要一些基本的技能；交互设计师应该有很好的绘画或写作能力（精通两者是很难得的，也是珍贵的），他们还必须和同事和客户都能很好的交流。最难获得的技能是将创造性的远见和分析思维结合起来，这也是伟大的交互设计师的特点。

无论你接受的基础训练是什么，作为一个设计专业人员，你必须与庞大的开发队伍中扮演其他角色的人密切合作。你要尽一切可能促进你的组织以目标导向的模式思考。这是个艰巨的任务，但也是有巨大的个人回报和组织回报的事业。你的权威得到承认是关键，但是这意味着你将需要承担几种职责。

和产品团队合作

当设计师们综合集体的智慧形成一个方案后，程序员们常常难以接受他们的权威，而擅自改变。任何与程序员一起工作过一段时间的人都会有这样的体验：团队开会，每个人都对行动过程达成一致意见。人人都认可了各自的任务以及程序将来的外观。两个星期过后，当小组重新集合的时候，一个程序员说——毫无讽刺之意——“啊，我决定以这种方式来做。我想这样会更好，”此时团队中的其他人就只能免开尊口了。即使那样做真的更好（实际上常常并非如此），当一个团队依靠你的时候，单方面的改变事情仍然是错误的。

严格的开发过程可以明显地改变这种状况，在开发过程中设计与工程，销售和商务管理作为平等参与者——并明确责任和各组的权限。责任进一步细分，与平分的权限保持均衡，可以显著提高整个开发周期及其远期的设计成功率和组织对产品的支持率（Korman，2001）。你应该在你自己的组织中鼓励这些：

- 设计团队负责用户对产品的满意度。当前多数组织无人承担这个责任。为实现这个责任，设计师必须有权力决定产品应具有怎样的行为。他们也需要收集信息：他们必须观察和通过交谈了解潜在用户的需要，与工程师交流技术的机会和限制，与销售人员交流市场供求，还要与管理人员交流组织会认可的产品种类。

- 工程团队有管理用户看不见的系统技术（实现细节）的权力。因为设计有着它的好处，工程师必须负责按规范的那样建立设计师定义的行为，同时要遵守预算按期完

成。因此工程师需要一份产品行为的清晰描述，这可以指导他们创建系统、分配时间以及成本预算。这种描述必须来源于设计团队。

- 销售团队负责产品对客户的吸引，所以他们必须有权力与客户进行全面的交流（谨记客户并不等同于用户；客户购买的是产品，但并不需要和用户那样使用它们。）为了做到这一点，团队需要掌握包括设计师的研究结果和他们自己的调查结果。
- 管理者负责产品的利润率，因此他们有权决定其他各组的工作。为了做决定，管理人员必须从其他各组那里获得确切的信息：设计产品的定义，市场销售计划和工程师创建产品的时间和成本的计划。

和工程师团队合作

为了产品的成功，设计和工程团队必须有健康的工作关系（Korman,2001）。许多工程师（从以前的不幸经历中）已经认识到设计师不是完全值得信赖的。当软件工程师不信任设计师的判断的时候，为了控制他们甚至声称不可能。因此，设计团队需要付出额外的努力来了解开发团队。

设计师有义务足够清楚地了解技术平台确保工程师能够实现他们的设计。但是，设计师不应该过多地考虑实现的容易性，而应该考虑可能性。基于容易实现的决定应该是管理要求，只有在设计基础上完成工程时间和成本估计之后出现。这也意味着管理人员在他们从工程师那得到时间和成本预算之前，不应该设置实现期限。

在设计的过程中，设计师必须吸收工程领域中的知识来理解技术的机会和限制。这将影响到产品可以采纳的行为。设计师还必须创作清晰恰当的细节形式和行为说明；一个理由充分的谨慎的设计说明书会消除工程师的恐惧心理，因为它没有遗留无答案的界面问题。因此，它降低了改变要求的可能性，也减少了打断工程进度的失误。

设计师把设计说明书交给工程师之后，他们仍然是工程师的资源。任何说明书不可能预期所有的可能行为和状态，所以设计师们必须在工程师创造产品的过程中支持说明书的解释，详尽阐述和临时决定设计细节。这也作为对设计师的检查，确保他们交付的说明书尽可能清晰。

实际上，在交付设计说明书过程中存在一个逆转关系：在交付之前，工程师为设计师服务，提供技术咨询。在交付之后，设计师为工程师服务，在整个实现阶段提供设计咨询。

展望未来

在数字技术领域，事情可以不同。软件和数字产品有着无穷的潜能。它们的设计是从帮助用户实现目标出发，而不是为计算机的方便而进行编程。

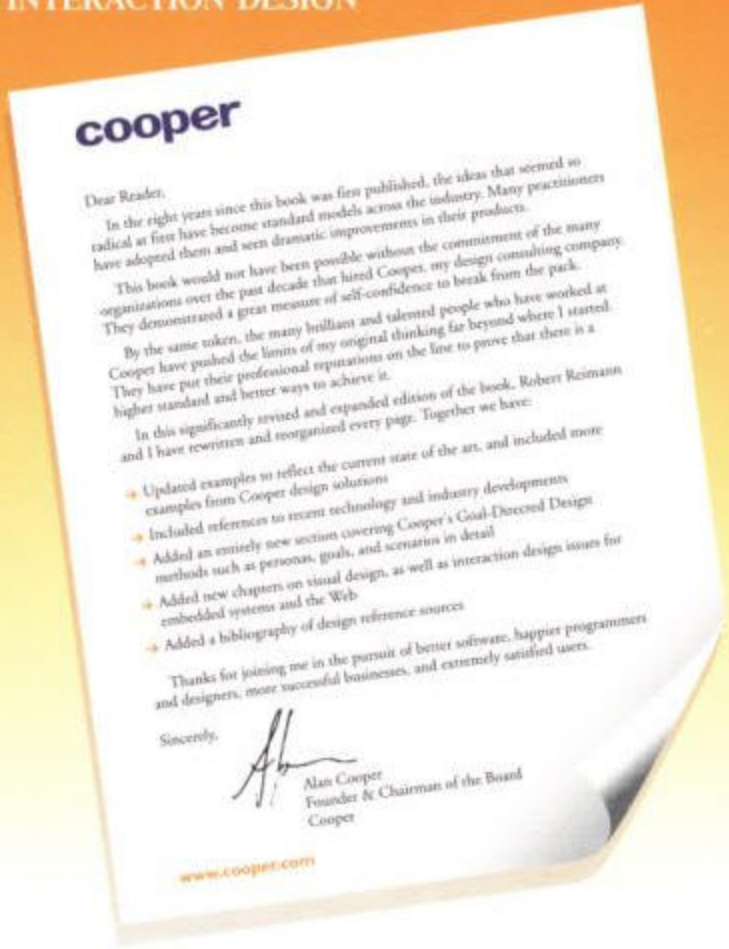
新经济的起落已经冷静地唤醒高技术市场。技术销售已经下降。人们已经意识到他们已经具有所有的动力，超出自从有 PC 和蜂窝电话已来所能想到的特性。他们没有获得，不能从其他人获得的是能够使用起来感到愉悦的产品。工业界已经收到了来自消费者没有信心的选票，他们厌倦于没有用的附加功能，并且无效果地寻找，寻求可能是解决真实问题的解决方案。通过合适的行为满足用户需要的产品设计是产业界的未来——惟一有意义的未来。新经济必须准备满足真实人们的真实需要，而不是技术无情的前进，将我们都踩在脚下。

我们希望本书对你有意义。对那些将它放在书架上的职业人员有意义。对于我们这些交互设计人员来说，确定我们的目标是一个艰难的斗争，并且挑战没有结束。但只要有像我们这样看到人性技术重要，并且不愿意让技术论者忘记人性技术的重要的人们，我们就有着更明亮、更人性化的未来。

Copyrighted Material

ABOUT FACE 2.0

THE ESSENTIALS OF INTERACTION DESIGN



cooper

Dear Reader,

In the eight years since this book was first published, the ideas that seemed so radical at first have become standard models across the industry. Many practitioners have adopted them and seen dramatic improvements in their products.

This book would not have been possible without the commitment of the many organizations over the past decade that hired Cooper, my design consulting company. They demonstrated a great measure of self-confidence to break from the pack.

By the same token, the many brilliant and talented people who have worked at Cooper have pushed the limits of my original thinking far beyond where I started. They have put their professional reputations on the line to prove that there is a higher standard and better ways to achieve it.

In this significantly revised and expanded edition of the book, Robert Reimann and I have rewritten and reorganized every page. Together we have:

- Updated examples to reflect the current state of the art, and included more examples from Cooper design solutions
- Included references to recent technology and industry developments
- Added an entirely new section covering Cooper's Goal-Directed Design methods such as personas, goals, and scenarios in detail
- Added new chapters on visual design, as well as interaction design issues for embedded systems and the Web
- Added a bibliography of design reference sources

Thanks for joining me in the pursuit of better software, happier programmers and designers, more successful businesses, and extremely satisfied users.

Sincerely,

Alan Cooper
Founder & Chairman of the Board
Cooper

www.cooper.com

Alan Cooper is a pioneering software inventor, programmer, designer, and theorist. He is credited with creating what many regard as the first serious business software for microcomputers, and is widely known as the "Father of Visual Basic." For the last decade, Alan's interaction design consulting firm, Cooper, has helped companies invent powerful, usable, desirable software and improve digital product behavior through the use of Alan's unique methodology — the Goal-Directed process. A cornerstone of this method, the use of *personas*, has been widely adopted since it was first described in Alan's second book, *The Inmates are Running the Asylum*. A best-selling author and popular speaker, Alan is a tireless advocate for integrating design into business practice and for humanizing technology.

Robert Reimann has spent the past 15 years pushing the boundaries of digital products as a designer, writer, lecturer, and consultant. He has led dozens of design consulting projects for startups and Fortune 500 clients alike. Upon joining Cooper in 1996, Robert led the development and refinement of many Goal-Directed Design methods described in *About Face 2.0*. He has lectured at major universities and to international industry audiences, and he is a member of the industry advisory board for the Institute of Design at the University of California, Berkeley.

"About Face 2.0 is one of the very rare design books that's fun to read, even though it rocks fundamental beliefs and packs the page with useful information. It's a must-read for anyone who wants to understand what the software design process should be (but usually isn't). The perspective is unique: intellectually rigorous enough for academics while remaining focused on helping practitioners. I'd recommend this book to anybody in the business."

— Harley Manning, Research Director, Forrester Research

7 85555 86509 9

Visit www.wiley.com/compbooks

WILEY
wiley.com

$35.00 US
$52.99 CAN
£24.50 UK

LEVEL
Beginning to Advanced

SHELVING CATEGORY
User Interfaces

ISBN 0-7645-2641-3

9 780764 526411 53500

Copyrighted Material