第一章 SWT 和 JFace 概览_1

本章涵盖内容

■ SWT 和 JFace 的目的

■ 它们形成的理由

■ 这两个类库如何区别于 Swing

■ 许可证和平台支持

在 2004 年三月，Java 开发者年度大会宣布了由其读者选择的最佳 Java 组件年度奖结果，超过 15,000 位开发者投票选举诸多 Java 工具套件（当然包括不少声名显赫的供应商如 Oracle 和 Apple 等）之一。但最终，Eclipse 的标准小部件工具套件轻易地击败诸多强手，如同其在 2003 年一样赢得了年度最佳。

虽然 Eclipse 是 Java 开发领域的迟到者，但其同样在 JavaWorld, JavaPro 和 LinuxWorld 社区获得了荣耀。一路而来如潮的掌声和好评彰显了 Eclipse 对于 Java 开发的巨大冲击和影响。在过去的每一天，全球的 Java 开发者们不断被 SWT 和 JFace 强劲的功能和部署新插件和独立应用程序的能力所折服。

本书的主旨就是在于向你展示这一功能套件的功效和你如何将其应用于你现成的项目之中去。需要指出的是：

■ 开发基于 SWT/JFace 的应用程序

■ 用 SWT 内置的图形语义环境生成定制化图形

■ 理解 SWT 和 JFace 后面的结构和方法论

■ 开阔你的 GUI 设计的知识面

■ 构建和部署 Eclipse 的 SWT 和 JFace 应用程序或独立应用程序

最重要的是 GUI 开发应当是一项充满乐趣的工作！没有一个编程工作能给你这样子的快感。因此，我们将 SWT 和 JFace 的理论用实例代码贯穿始终来显示 GUI 开发实践。但在开始之前，我们需要向您简要阐明该技术是什么和它能帮你作什么。

1.1 什么是 SWT/JFace?

虽然我们都知道 SWT 和 JFace 是工具（套件），但更科学地讲它们是软件类库。它们由包含 java 类和接口的文件包组成。但又是什么使得这些组件能让你任意组合 GUI 呢？你的应用程序可以快速运行，高效运用计算机内存并有着和操作系统界面相同的界面体验，没有其他的 GUI 构建体系可以如此。虽然 SWT 和 JFace 实现了相同的功能，但它们产生用户界面的机理却是迥然不同的。

我们试图通过汽车驱动机理来类比这一情况：SWT 开发就像标准的汽车驱动模式，它给了你更多的控制权力，并有机会接触系统内部的东西，但是其使用是相当复杂的；而 JFace 情况下，就像是汽车的自动驱动模式，你不必深入太多，但是你丧失了灵活性。当然，实际情况肯定比这个比喻要更为复杂。所以我们需要更进一

步的讨论这两个类库。

1.1.1 用 SWT 构建 GUIs

每一个操作系统都会有大量的图形组件来构成其默认的用户界面。这一些包括有：按钮、窗口、菜单以及诸如此类。SWT 的目标就是给予你直接获得这些组件的途径，然后如你所愿地将它们定位和设置。你不必担心最终用户的操作系统如何，当你在应用程序中加入一个按钮，它就会在 Windows 中表现得如同 Windows 得按钮，在 Mac 中如 Mac 的按钮，当然在 Linux 中亦是如此。用户会认为你这是为他们的机器定制，而他们并不清楚事实上你仅用 SWT 写了一遍代码。

除了图形组件，SWT 还提供事件处理，这意味着你可以追踪你的用户按下了哪个按钮或是选择了哪个菜单项目。这一强劲的功能使得对用户任意形式输入产生反应成为可能。接下来我们会花大量篇幅来演示这是如何运作的。

最后，如你想在你的应用程序中加入图形，SWT 提供大量的工具来产生图形，处理新字体或是绘制形状。这些图形可以使得你不尽可以构建图形，还可以让你控制图形何时、何地和如何地在你的 GUI 中显现。本书会向你演示 SWT 如何管理颜色、绘图、字体、图案，并贡献了大量的实例代码。

SWT 提供了构建用户界面的巨大能力，但是将如你在本书中所见，代码将会变得冗长而且复杂。正因如此，Eclipse 的设计者推出了 GUI 开发的第二个类库：JFace。

1.1.2 用 JFace 来简化 GUI 开发

为避免一遍遍地使用 SWT 来写重复的代码，Eclipse 的设计者用 Eclipse 工作台产生了 JFace。这个类库提供大量的快捷方式以削减因单独使用 SWT 而大量耗费的时间，但另一方面，JFace 不能完全取代 SWT，许多 GUI 的开发还需要两个套件的特性。

Jface 高效的一个例证就是其事件处理。在许多用户界面中，你或许要处理诸多不同的事件，如：点击按钮、敲击键盘或者菜单项选择，而事实上如上事件都在实现同一功能，在此情况下，SWT 的处理方式是需要对每一个事件单独安排接受和处理过程；而在 JFace 中允许你将他们组合成一个单一对象，这样你可以集中精力于如何应对事件（对象）的处理，而不必理睬事件是如何激发的。这一简单但强力的概念使得你可以在你的 GUI 中加入菜单、工具条或甚至是调色板而不需加入一大堆代码。

Jface 另一个有助益的地方就是当你在构建大型的多窗口、多图形的 GUI 时，它可以通过其特有的注册类来帮你组织 SWT 部件和管理内存开销。举个例子，在 SWT 中，你需要把你应用程序中的字体和位图的生成和分布要予以说明；而在 JFace 中，你大可使用其内置的 FontRegistry 和 ImageRegistry 对象来处理这些乏味的问题。

现在你应该基本理解了这两个类库的特性，接下来我们要更进一步以阐明它们的一些设计概念。

我们将快速浏览如下这些论题：它们是如何在操作系统中表现的，初始产生又是如何的等等。

1.2 揭开其面纱

在用户街面上添加组件、事件和图形并不是新鲜玩意儿了，然而 SWT 和 JFace 何以会激起千层浪呢？在这里

你需要理解设计者的孤心苦诣了。而这又要牵涉到深层次的 Java GUI 开发的原则问题以及如何使用这些类库等等。而在我们深入探讨 SWT/JFace 和 JFace 之前，我们还是有必要介绍一下 Swing。SWT 和 JFace 的产生也正是 Swing 类库（的不争气）而产生，通过比照这 Swing 和 SWT 两个类库的涉及哲学，你会更折服与 SWT 和 JFace 的运行机制；通过这些，你或许也会加入到 Java 开发的 Swing 和 SWT/JFace 两大阵营日益白热化的争论之中。

1.2.1 Swing：老的替代品

当 sun 在 1998 年发布 Swing 类库时，Java 开发阵营相当的欢欣鼓舞。因为最终 Sun 能够以这一工具套件的平台无关的特性来实践其“编写一次，到处运行”的信条了。Swing 也成为之后 Java GUI 开发的最为流行的工具。

但随着时间的推移，许多开发人员又失望了：Sun 在造就 Swing 迷人特性的同时，也使得程序开发日见复杂，运行效率日益低下。正因如此，很少有看到成功的 Java GUI 开发的桌面应用。

Swing 的渲染

为了确保界面在跨平台是能保持一致，Swing 对用户界面的每一个细节的渲染进行完全控制，也就是 Java 虚拟机指令到其组件的每一个象素和具体行为。虽然 Swing 也同底层平台保持通信，但是却不使用操作系统任何已内置的对象，即其白手起家，创建每一个对象。因为这些组件都运行于较高层次，它们被引喻为轻量级组件。这些组件在任何 Java 支持的平台上表现一致。

但是这种方式是有缺陷的，因为 JVM 要管理细化到 GUI 的每一个细节表现和行为，相对于直接依于操作系统的应用程序运行就十分缓慢了。而用户来说，他们绝大多数都偏好于操作系统的界面特性。

Swing 的自动垃圾回收机制

为了确保 Java 的可靠性运算保证，Swing 沿用了 Java 的自动垃圾回收机制——AGC。这产生了一个威胁和隐患，因为在应用程序层面上，对于对象的内存分配起始已经不需要了。在程序独立于开发人员而运行时 AGC 被激发，其一个重要的能力就是如果程序员不释放他们的数据，其他程序就无法为其对象调用这块内存。

AGC 的一个好处就是开发人员只需关注编程而不必关心每一个对象的生命周期，而另一方面，内存分配基制是你无从知晓它会何时发生，另外 AGC 的能力也会因 JVM 身处的操作系统平台的不同而不同。因此，由于大型应用程序内对象产生和丢弃的时效特性，程序的表现在各个不同系统内就无规律可循了。

Swing 的设计架构

Swing 通过 MVC 模式来指导 GUI 设计进程。MVC 将用户接口组件分解为三块：状态信息、显示和对外界事件的反应能力，就是俗称的模型、视图和控制。Swing 的设计者修改了这一方法论并产生了模型代理架构，如图 1.2。这一架构整合了部件的视图和控制部分成为一个 UI 代理模块。所以对于每一个单独的用户界面组件如按钮、框架、和 label-Swing 分配内存均包含这组件状态信息和 UI 代理，也即其既控制着表征和事件反应。

通过将模型信息从表征分离出来，Swing 提供了一个编程方法论，它确保了灵活、可重用编程。但这一能力对于每一个部件都产生了多个对象。当 GUI 变得复杂是，这一额外的分派和丢弃做法会对处理器产生巨大的

负担。

第一章 SWT 和 JFace 概览_2

1.2.2 新贵：SWT/JFace

Eclipse 的设计者注意到了 Swing 的灵活性和其执行问题。他们想要一个套件可以确保 Java 的用户可以象使用操作系统一样运行一个桌面程序。在实际上，他们是如此之迫切需要，以至于他们编制了他们自己的类库：SWT 和 JFace。

无论 Swing 和 SWT/JFace 都会产生一个基于 Java 的平台无关的 GUI，但他们的实现方法又是迥异的。

SWT 和 JFace 的最显著的特征是其介入了直接调取操作系统，使用底层平台的重量级组件，而不是自己重建。这一决策使得 SWT 和 JFace 的表征和运行速度接近于地层平台，在下一章我们会就此展开更为深入的讨论。当然在此的短暂描述也是有所裨益的。

由于当初 Java 的初创者一开始就意识到 Java 应用程序最总会需要使用到传统代码或是操作系统，所以他们提供了从 Java 类内部去调取其他语言（如 C 或 Fortran）调取过程的类库。

SWT/JFace 依靠 JNI 来管理操作系统的渲染而不是由其自己来实施。

SWT/JFace 的资源管理

SWT/JFace 的另一个重要特性就是其不依赖于垃圾自动回收机制。一开始，这很容易让人以为这会产生错误代码。然而，你只需在接触到操作系统资源是加以小心，一般不会出现重大问题。Eclipse 决定将 AGC 从 SWT/JFace 中移走主要有两方面的原因：

■ 内存自动分配的过程在程序运行时是不可预测的，没有任何征兆说明资源何时被释放（可得），当处于分配过程中时，如果异常情况发生，则过程就可能无法关闭。如果你仅是在处理一些简单数据结构时，可能这是小事一件。但，如果是在处理一些大型的图形程序时，内存的分配和回收就是不得不要小心谨慎对待的重大问题了。

■ 对操作系统资源使用 AGC 是困难的，由于 Swing 在高端层次建立了其轻量级的组件，所以这还不成为一个问题。然而，如果在低端层次如 SWT 小部件同样去使用自动回收机制，那么在跨平台环境下错误的易发性和不确定性将会是个灾难。如果要花费太多的时间去清理对象，那么内存泄漏问题就会拖垮程序；或者如果这些资源是以一种错误的方式来释放资源的，那么整个操作系统也要跟着倒霉（宕机）了。

为了预防此类与对象自动回收机制相关联的错误的发生，SWT/JFace 让你自己来自主决定何时释放资源。这一工具套件通过其组件类中内置的 dispose()方法来简化这一处理过程。同时，一旦你释放了某一父（上级）资源，其子（下级）资源也会被自动释放。

在以后的章节中，你将看到这意味着在大部分的应用程序中几乎不需要直接的内存释放的必要。你也可以以半自动方式来调取 SWT/JFace 来管理内存。

设计和开发的简化

在 swing 中 GUI 是以模型代理架构产生的，由此会对 GUI 的每一个组件产生不同的对象以表征其不同的特性。但这一复杂方法不是任何情况下都适用。对于那些刚开始起步学习构建按钮、标签的新手来说不需要如此复杂。而在其他的极端情况下，程序员在构建复杂的图形编辑器和计算机辅助设计工具是，需要与 GUI 功能更多地分离，以取得不同的试图和设计模式。？

SWT 和 JFace 对其组件的设计架构没有强加任何规则，这意味着你可以随心所欲地极端复杂化地或是极端简化地构筑你的 GUIs。因为 Eclipse 极易扩展，同时其源代码又是开放的，你可以随你所愿地添加工具或作修改。事实上，已经有大量的插件为 SWT 和 JFace 组件的 MVC 封装而开发出来了。

1.2.3 SWT/Swing 之争

随意在网上搜索一下 SWT 和 Swing，都会看到关于这两者孰优孰劣的热烈争论。这一论战是毫无必要也是没有实际意义的。SWT 是作为 Swing 之外的另一选择而产生的，而不是为了去替代 Swing。我们撰写本节的目的并不在于褒扬某一工具较另一工具有优势，而是为了解释说明如何运作，为何运作的一种机理。在 Java 开发者中的这种争斗只会伤害到为构建自由的平台无关的应用程序的热忱。这世界给予了 SWT 和 Swing 同样广阔的天地，我们希望这两大阵营求同存异，共创 Java 社区的大同世界。

1.3 SWT/JFace 的许可证和支持平台

在正是开始编写代码之前，我们很乐意谈及 SWT 和 JFace 开发应用程序的两个要点：第一、如同在通用公共许可协议中概要的那样，Eclipse 和其开发库的条件（？）太缺乏，当你在考虑开发商业应用程序时很重要；第二、需要关注 Eclipse 和 SWT/JFace 当前所支持的平台。

1.3.1 CPL 通用许可协议

Eclipse 软件基金会是在 CPL 框架下向公众发布 Eclipse 的。这一许可协议和开放源代码组织的许可协议方案是完全兼容的，并且通过授权免特许权使用费的代码允许作完全的商业使用并可以在全球范围内再发布。这意味着任何人可以使用源代码，修改源代码并且销售其最终产品。

虽然某些平台部件是在特殊许可条件下发布的，但是由于整个 SWT 和 JFace 是在 CPL 框架范围内的，所以这使得在全部可支持平台内开发商业 SWT 和 JFace 应用程序成为了可能。

1.3.2 支持平台

在撰写本书时，在不少平台上都已经开始支持 SWT 和 JFace 开发了。因为其依赖于特殊的窗体功能，因此在某些平台需要多个 SWT 应用实施。表 1.1 列举了支持 SWT 和 JFace 的操作系统和用户界面。

在 linux 操作系统中，尚不能支持 KDE，然而但是如果在 KDE 系统内安装有 GTK 的运行库，那么 SWT/JFace 应用程序同样可以运行在 KDE 桌面。由于 KDE 是构建在 Trolltech Qt 套件基础之上的，而 Qt 套件的发布许可协议比 CPL 有着更多的限制。如果在将来开发了一个 KDE 版本的 SWT 类库，那么现存的所有的 SWT/JFace 应用程序就可以支持它，并继承获得 KDE 的原生的外观。

SWT 还有一个秘密武器就是其对微软 Pocket PC 2002 系统的支持。SWT 发行版提供了对于基于 StrongARM 体系处理器的 Pocket PC 2002 和 Smartphone 2002 设备的支持。正是由于 SWT 的极大

灵活性，使得 Pocket Pc 版本的 SWT 可以运行于大家熟知的 J2SE（标准 Java）和 J2SE 连接首先设备配置的嵌入式设备上。关于和 IBM J9 虚拟机相联系的如何在 CLDC 上构建 SWT 库的问题不在本书讨论范围之内。详情可见 Eclipse 基金会新闻组的网址(news://news.eclipse.org/eclipse.platform.swt)。

对于 Windows 平台的支持还有一个预想不到的好处：你可以在 SWT 容器小部件内直接嵌入 ActiveX 控制。Eclipse 平台使用这一工具方式嵌入微软网页浏览器控制来获得嵌入的网页浏览功能。你可以在附录 B 的“在 SWT/JFace 中 OLE 和 ActiveX”中获得更多的关于 ActiveX 的技术细节。

Table 1.1 SWT/JFace 支持的平台

Operating system	User interface
Windows XP, 2000, NT, 98, ME	Windows
Windows PocketPC 2002 Strong ARM	Windows
Windows PocketPC 2002 Strong ARM (J2ME)	Windows
Red Hat Linux 9 x86	Motif, GTK 2.0
SUSE Linux 8.2 x86	Motif, GTK 2.0
Other Linux x86	Motif, GTK 2.0
Sun Solaris 8 SPARC	Motif
IBM PowerPC	Motif
HP-UX 11i hp9000 PA-RISC	Motif
QNX x86	Photon
Mac OS	Carbon

1.4 The WidgetWindow

学习 SWT/JFace 工具套件的最佳途径就是通过使用器类来构建 GUIs。有了这种倾向性的意识，我们将努力以一个扩张性的项目来触及 SWT/JFace 开发的各个不同方面。最先，我们需要构建一些激动人心的东西，如一个 web 驱动的数据库显示。

但细一考虑我们又觉得不妥，因为这样一来我牵涉到太多无关的代码，这无形之中给了读者一个不小的负担。

因此，我们选择了一个简单的应用程序来合成尽可能夺得 GUI 元素，以此来减少代码数量。我们认为选项卡折叠器对象（在第三章我们会详细讨论）是在本书中演示用的最简洁的方式。然后在书中接下来的每一章，然后我们就可以一个新的添加含有该章的主题内容选项卡。图 1.4 展示了一个设计好的程序的全部，我们称其为小部件窗口。

小部件窗口的开发有着诸多功效。当然，表面上看它仅是一个简单的应用程序集成有 SWT 和 JFace 类库不同的组件。另一方面，它事实上是给了你一个可重用的 SWT 和 JFace 代码的知识库。由于它是一个多类单一项目，相对于单一类多库，小部件窗口可以确保你可以在你自己的用户界面中重用它的每一个部份。

图 1.4 小部件窗口应用程需这一扩张性的项目将会集成本书中谈到的每一个 GUI 和图形组件。

1.5 小结

SWT 和 JFace 类库对于构建用户界面是高效的，但对于它们自己而言，它们并不能作任意简化。就像在其他工具套间中一样，还是需要定位和操控诸如按钮、容器、标签和菜单等。好在，这一工具套件背后蕴涵的哲学使得其具有革命性。

SWT 和 JFace 或许并不完全遵循 Java 意识形态的各项规则，但通过其假疏丝组织的发展，它比 Java 在更大程度上贯彻和实践了开放源代码软件的宗旨与目标。

SWT/JFace 不仅无需要任何的许可证或是授权，还允许你程序员可以基于它作开发并收取费用。如果你开发了一个新的操作系统并且需要一套开发工具来吸引程序员到你的平台上来，那么你所需作的就是为你的系统适当裁减 SWT 和 JFace。如果你正在构建一个新的编程语言，并且不满足于简单的行命令或是连接，那么带有 SWT 和 JFace 的 Eclipse 平台对于你的任务来说是相当合适的。

当 java 开发者们在争论相较与其他套件 SWT/JFace 的优点时，他们事实上仅考虑到当前或者时接下来 6 个月的能力。这一思维定式忽视了 SWT 和 JFace 的这么一个事实，就是：象 Eclipse 一样，因为有着来自全球的公司的、个人的贡献，Eclipse，SWT/JFace 的发展几近于一个流行市场。如果在将来继续积聚开发人员的大量时间，

SWT 和 JFace 就可以象它正在进行的革新一样，唾手可得取得胜利。

历史上，软件开发始终未能成为 IBM 的强项，但我们对于 IBM 的这种通过向开放源代码贡献力量来提升 IBM 硬件价值的理念心存感激。

Eclipse 和 SWT/JFace 的自由和灵活性以及其开发人员的激情，使得我们有理由相信这一工具套件会在今后数年内不断助益开放源代码开发社区。

闲语少说，我们开始我们的编程吧！

第二章 SWT 和 JFace 编程起步_1

本章涵盖内容

- SWT 的重要类：Display 和 Shell
- 一个 SWT 编程实例
- JFace 的重要类: ApplicationWindow

■ 一个 SWT/JFace 编程实例

GUI 编程是软件开发中最有意义的部分之一，但当你一旦依赖于图形界面而非命令行时，就有一个问题浮现出来了：你是怎样接近这些小部件、容器以及操作系统的事件的？那些软件类代表着 GUI 的不同组件，你又是如何操控它们的？

本章的目的就是要回答上述第一个问题并着手准备第二个。我们将讨论 SWT 和 JFace 库的基本类以及它们是如何接近操作系统资源的。本章贡献了两个主源代码 HelloSWT.java 和 HelloSWTJFace.java 来演示在有和没有 JFace 库的情况下如何使用 SWT。我们已检查过这些程序并对它们的基本结构得出了结论。本章开始我们将逐步在小部件窗口项目中加入代码。通过这一图形界面我们将融合本书讨论的 SWT 和 JFace 的各个主题。我们在此建立其框架并在今后每章中予以更新。由于每一章都会有代码加入，我们建议你紧跟这一开发进程。

2.1 SWT 编程

虽然我们很快就能应用 JFace，但是本节我们将集中精力于单独的 SWT 编程。首先，我们要贡献一段基本的 GUI 代码并检查其结构；然后，本节会详述该套件的两个基本类：Display 和 Shell。这两个类提供了部件、容器和事件加入的基础。

请注意：为了能编译和执行本书中的代码，你需要在项目中加入 SWT/JFace 类库，并使程序能找到原生图形库。附录 A“生成 SWT/JFace 项目”中予以了详尽的介绍。

2.1.1 HelloSWT 程序

在开始细节探讨 SWT 理论之前，最好还是早一点取证明它是如何工作的。由于这个原因，我们贡献了第一个 SWT 的 GUI，HelloSWT.java。我们鼓励你把这个加入到 com.swtjface.Ch2 包中，并执行程序。

```
package com.swtjface.Ch2;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

public class HelloSWT {

        public static void main (String [] args) {

                Display display = new Display();

                Shell shell = new Shell(display);

                Text helloText = new Text(shell, SWT.CENTER);

                helloText.setText("Hello SWT!");

                helloText.pack();
```

```
            shell.pack();
            shell.open();
            while (!shell.isDisposed()){
                if (!display.readAndDispatch()) display.sleep();
            }
            display.dispose();
        }
}
```

虽然 HelloSWT 是一个简单的 GUI，但是绝大部分的 SWT 应用程序都是有着如下的三段式的结构：

第一部分以生成一个 Display 和 Shell 类的实例开始。我们马上就会讲到，这一过程是 GUI 获取底层平台的资源并为其他小部件开辟了一个主窗口。

第二部分是在 Shell 上加入一个文本小部件，虽然这很简单，但是对于 SWT 应用程序所作的事情可一样也不能少。它需要处理添加和设置 GUI 功能生效的阻断。小部件和（容器内）的小部件分组作为 Shell 的子部件而添加入。

每一个小部件的监听器和事件都做了定义，以便用户的就此开始动作。本节中达代码也对小部件、容器和事件设定了参数以确保它们按要求预定的显示和行为。在此，pack()方法告诉 Shell 和文本组件使用它们所需的空间。

最后一个部分代表这 GUI 的操作，直到这一刻为止，整个应用程序的代码除了初始化变量还没有干任何事情。但是一旦 Shell 的 open()方法被调取，应用程序的主窗口就成形其子部件在 display 中被渲染。只要 Shell 保持 open 状态，Display 的实例就会使用其 readAndDispatch()方法来追踪在操作系统事件队列中与用户相关的事件。当某一个动作促使窗口关闭时，与 Display 对象（包括 Shell 以及其子部件等）相联系的资源就全部释放。

图 2.1 显示了 GUI 是如何显现的（基于 Linux/GTK）。

祝贺你！你现在已经用 SWT 库产生了你自己的用户图形界面了。在继续开始下一个使用 SWT 和 JFace 的应用程序之前，理解我们所使用的方法和找到设置它们的途径相当重要。

2.1.2 Display 类

虽然 Display 类并不是可见的，但它负责监管着 GUI 的资源并管理着和操作系统的通信。也即意味着它不光要关注着它自己的窗口是如何显示、移动和重画的，还同时要确保诸如鼠标点击、键盘敲击等事件送达小部件并去处理它们。

Diplay 类的操作

虽然 Display 类在你的代码中仅有区区几行，但是正视和理解其操作相当重要。它是任何 SWT 和 JFace 应用程序的承载着，无论你是用 SWT/JFace 开发或是单用 SWT 开发，你必须在你的程序中包含这个类的一个实例。唯有如此，你的用户界面才能使用操作系统的小部件和容器并对用户动作作出反应。虽然大部分的应用程序对于 Display 类所作甚少，或是仅调用了其很少的方法，但是这个类所扮演的角色之重要性还是只得我们大书特书的。

Display 类的主要任务就是负责将你的代码重的 SWT 和 JFace 命令翻译成底层的命令来调取操作系统。这一过程负责将两部分组合并随着 Display 类的实例的产生而开始。首先，Display 对象构建一个代表着操作系统平台的 OS 类的实例。这个类通过一系列被称之为原生方法的特殊 Java 处理过程提供了接触计算机底层资源的途径。

然后，像个总机接线员一样，这个 Display 对象使用这些方法来直接指令操作系统并向应用程序传达用户动作。如下的关于原生方法的例子中就是对于 OS 中 SetFocus()方法的申明：

```
public static final native int SetFocus (int hWnd);
```

这一方法根据其句柄，hWnd 在窗口上设定了定位。因为是原生的方法，此间不存在 Java 代码来指明其操作。反过来，这一关键字告诉了编译器这个方法是用另外一种“其他语言”编写的，并且存在于“另外的文件”之中。本例的 HelloSWT.java 和其他所有的 SWT 和 JFace 应用程序的这一“其他语言”就是 C，这一“另外的文件”就是包含于你的项目文件之中的原生图形库。与 SetFocus()方法相联系的图形库中的 C 代码如下：

```
JNIEXPORT jint JNICALL OS_NATIVE(SetFocus)

(JNIEnv *env, jclass that, jint arg0) {

jint rc;

NATIVE_ENTER(env, that, "SetFocus\n")

rc = (jint)SetFocus((HWND)arg0);

NATIVE_EXIT(env, that, "SetFocus\n")

return rc;

}
```

如你所见，在 Java 代码中的 SetFocus()方法应用的 C 程序调取了操作系统的 SetFocus()函数。很巧，（此处两个 SetFocus 正好同名，但决不是一体的），相吻合的 SWT 命令和操作系统调用使得 GUI 的调试简洁明了。只要你能走出你操作系统 API 的谜局，你就能明晰在你的代码中到底发生了什么。在这个例子中使用的是 Windows 的操作系统，但处理的过程在其他 Eclipse 支持的平台上都是相似的。

需要考虑的另一个要点：如果你的操作系统具有的某类（某一）特性和 SWT 不兼容，此时你可以使用 Java 原生接口来自行加入。一切所需的仅是一个 SWT 中的 Java 原生方法和调取操作系统的在原生图形库中的

C 函数。

Display 类的方法

表 2.1 列明并描述了属于 Display 类的大量方法，这不是一个完全列表，但都是一些使得 SWT/JFace 发挥作用或是在应用程序中实现特殊能力的至关重要的方法。

有两个在任何基于 SWT 的 GUI 中必须用到的方法，第一个是 Display()，它产生一个类的实例并将其和 GUI 相联系；第二个是 getCurrent()，它返回一个应用程序的主线程，用户界面线程。第二个方法通常通常是以 dispose()方法来终结 Display 的操作。

表中接下来的两个方法是帮助程序在一旦用户采取了与 GUI 相关的行动后，接受来自于操作系统的通知。在第四章我们会深入探讨时间处理、句柄和监听器方法，但是理解 readAndDispatch()方法是相当重要的，因为这一方法可以接触到操作系统的事件队列并决定着任何用户的动作是否和 GUI 有关。使用这一方法，HelloSWT 类可以知道是否用户决定要丢弃 Shell 对象。如果是，那么方法就回归一个值“真”，然后应用程序结束。否则，Display 对象就调用其 sleep()方法，应用程序就持续等待。

虽然 Display 类是重要的，但是没有办法可以直接观察到其运作的效果。取而代之的办法是你可以使用这些类的可视表现。

综上，所有最重要的类应当是 Shell。

2.1.3 Shell 类

好比 Display 类提供了窗口管理一样，Shell 类扮演着 GUI 主窗口的角色。和 Display 对象不同的是，一个 Shell 实例是一个可视化的应用，在图 2.1 中，Shell 类在某种程度上通过 OS 类的接触到操作系统，并仅对主窗口的打开、激活、最大化、最小化和关闭保持追踪。Shell 类的主函数为整合在 GUI 内的容器、小部件和事件提供了一个通用的接入点。从这一点讲，Shell 的作用象这些组件的父类。图 2.2 中显示了在一个应用程序的操作系统、Display、Shell、和其他小部件之间的关系。

表 2.1 Display 类中的重要方法和它们的功能

Display method Function

Display() 分派平台资源并产生一个 Display 对象

getCurrent() 返回一个用户界面线程

readAndDispatch() Display 对象翻译事件并将它们传递给接受其

sleep() Display 对象等待事件

每一个 SWT/JFace 应用程序都是基于其主 Shell 对象上的小部件的，但是在应用程序中可能还存在着其他的 Shell 对象，它们通常是和一些临时窗口或是对话框相联系的（这些将在第 10 章里讨论）。这些 Shell 并不附着于 Display 实例，因而被看作是从属性的 Shell。附着于 Display 的 shell 被成为顶层 shell。在 HelloSWT 应用程序的 Shell 实例有着大量的属性，这些属性可以允许用户修改状态值或是读取信息。这样的特性形成了组件的风格。你也可以通过向 Shell 的构造器加入第二个参数来控制 Shell 的风格。之前我们

看到的是在 HelloSWT 的 Shell 声名中仅有一个参数，那就是 display，这时 Shell 接受了一个顶层窗口的默认风格，称为“SHELL_TRIM”。

着组合了大量的单个风格元素，并告诉应用程序窗口应该有一个标题栏(SWT.TITLE)和用户可以最小化 SWT.MIN)、最大化(SWT.MAX)、改变尺寸(SWT.RESIZE)和关闭(SWT.CLOSE)shell。至于其他的 shell 的默认风格，DIALOG_TRIM 则确保着对话框的 shell 有一个标题栏、一个活动区的边界(SWT.BORDER)和被关闭的能力。

在你的 GUI 之内，你可以设定 shell 或其他小部件的风格参数值，若是多个值则可以用“|”相连。除了提到的属性，你还可以确定 shell 的形态，以限定用户修改 shell 的状态。有一个形态对话框会将除对话框以外的动作全部阻断以获取用户的关注。它不能被移动或是改变尺寸，只可以使用给予的按钮关闭或是取消。

最后，由于不是每一个系统平台都可以对 GUI 的组件进行渲染，所以你应当理解 SWT 仅是将这些风格设置作为一种指导而不是严格的限定。

第二章 SWT 和 JFace 编程起步_2

2.2 SWT/JFace 程序

有着对与 SWT 的清晰理解，我们就可以直入主题学习 JFace 了。虽然应用 SWT 和 JFace 的编程相对于单独使用 SWT 的结构全然不同，但是这两个库蕴涵的概念是相近的。象先前的章节一样，本节会提供给读者一个 SWT/JFace 的实例代码并阐释器结构。再进一步，我们会深入转眼 JFace 类库里提供的一个重要类：ApplicationWindow。

在第一章，我们解释了构造 JFace 的目的是为了简化 SWT 的开发。现在我们深入下去来演示它的主类是如何工作的。

2.2.1 基于模型的适配器

Eclipse 文档中使用了两个术语来指代和 SWT 小部件协同的 JFace 类：助手类和基于模型的适配器。在本书中我们选择了后者。关于这一点，确实是比较容易引起混淆的，因为在 Java 中一个适配器是指一个为小部件提供额外的事件处理能力的类，而没有一个有自尊的程序员会使用助手类这一词眼，因此我们就将此称为基于模型的适配器或是 JFace 适配器。

这些适配器可以被分为四大类，如在表 2.2 中列示的那样。在随后的章节中我们会深入剖析，但在此我们仅是作大致描述。

Table 2.2 Categories of JFace adapters

适配器分类 功能

阅读器 分离小部件的外观和信息

动作和实施 简化和组织事件处理进程

图案和字体注册 管理和分配/释放图案、字体资源

对话框和向导 扩展 SWT 对话框和用户互动的能力

第一个也是广为使用的大类包含有 Viewer（阅读器）类，这方面的内容在第 9 章会有全部的叙述。在 SWT 中，一个 GUI 组件的信息和外观通常是被绑定在一起的。然而，viewer 将这些方面分离了，并允许在不同的表单中使用相同的信息。例如，在 SWT 中的树的信息是不能被从树对象中分离出来的，但是在 JFace 的一个树阅读器中的相同信息却可以在一个表阅读器或是列表阅读器中显示。

第二个大类包含有动作和实施，这在本书第 4 章中有描述。这些适配器简化了事件的处理，并将针对用户命令的反应从引发那种反应的 GUI 事件中分离出来。这一点可以通过实例得以很好地说明。在 SWT 中，如果有四个不同的按钮都是去关闭一个对话框的，然后你必须要写上四遍不同的事件处理规程（代码）即便它们完成着相同的结果。在 JFace 中，这四个规程可以整合在一个动作中，然后 JFace 可以自动会将这四个按钮作为该行为的实施者。

第三大类包含有图案和字体的注册，这写在本书第 7 章有叙述。在 SWT 中，如何保持字体和图案的内存资源分配以最小是相当重要的，因为它们在消耗着操作系统的资源。但是由于有了 JFace 的注册机制，这些资源可以按需分配和释放。因此如你使用多个的图案和字体，你不必关注如何地手工回收垃圾。

最后一个大类包含有 JFace 的对话框和向导，本书的第 10、11 章会展开描述。这一些由于它们拓展了 SWT 对话框的能力，应该是最容易能够被理解的。JFace 提供的对话框可以展示信息、提示错误、显示某一过程的进度，额外的，JFace 还提供了一组特别的对话框称为：向导。向导指引着用户完成一组任务，如：安装软件或是设置一个输入文件等。

2.2.2 HelloSWT_JFace 程序

学习 JFace 的最佳途径就是用其类库写上一段代码。如下显示了 HelloSWT_JFace 类的代码，执行的结果和 HelloSWT 是相近的，但是程序的结构是完全不同的。

```
package com.swtjface.Ch2;

import org.eclipse.jface.window.*;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

public class HelloSWT_JFace extends ApplicationWindow {

public HelloSWT_JFace() {

super(null);

}

protected Control createContents(Composite parent) {

Text helloText = new Text(parent, SWT.CENTER);
```

```
helloText.setText("Hello SWT and JFace!");

parent.pack();

return parent;

}

public static void main(String[] args) {

HelloSWT_JFace awin = new HelloSWT_JFace();

awin.setBlockOnOpen(true);

awin.open();

Display.getCurrent().dispose();

}

}
```

虽然 HelloSWTJFace.java 程序代码比 HelloSWT.java 稍长，但是其结构有三个类的方法分隔而较为明晰。

第一个方法是 HelloSWT_JFace()，它构造了主类的一个实例。任何的分配过程中需要执行的设置和通信动作都需在此编写代码。而在 HelloSWT_JFace 类中这倒是不需要的，因为这个类仅调用了其超类的一个构造器。

第二个方法 createContents()处理窗口的设计。由于 ApplicationWindow 的可视化部分不是直接可得的，这一方法将与一个被成为合成器的小部件容器相联系来控制 GUI 的外观。这个容器对象作为任何 GUI 容器的父对象并需要被加入到应用程序中。在所有的小部件生成、设置以及加入父对象后，createContents()方法将该合成器返回主窗口进行显示。

这个应用程序的最后一部分是 main()方法，该方法负责照看 GUI 的真实运作。在 ApplicationWindow 的资源分配完毕后，这个方法就负责设置窗口的显现，直至 setBlockOnOpen()方法以一个真值被调用时关闭。然后，ApplicationWindow 的 open()方法被调用，根据 createContents()方法返回的合成器来显示窗口。而 open()方法之后的代码仅当窗口关闭时才发生作用。再后，程序使用 dispose()方法来释放 GUI 的 Display 实例资源。因为在 HelloSWT_JFace 中的每一个小部件都是 display 的子对象，丢弃了 display 对象也释放了程序中每一个 GUI 组件。

应用程序中的代码一经编译运行，效果应如图 2.3。

2.2.3 JFace and SWT/JFace 内的编程

到这一刻，对单独使用 SWT 编程的 HelloSWT.java 和应用 SWT/JFace 变成的 HelloSWT_JFace.java 的代码进行比照应当对我们有所帮助了。主要的区别在于 SWT 整合了 GUI 的外观和操作在其 Shell 类内，

而相对应地，SWT/JFace 将这些分离为若干部分。这一模式鼓励代码重用，并可以往使得一个程序员在考虑窗口的行为时，另一个程序员同时设计窗口的视图。由合成器控制着的外观可以用 createContents()方法来进行设置，而其运作部分主要是通过 ApplicationWindow 类的实例来实现。由于这个类对于 SWT/JFace 的应用程序如此之重要，所以需要细细检查其功效。

2.2.4 The ApplicationWindow 类

虽然我们已经提到过，HelloSWT_JFace 中的 ApplicationWindow 如何不同于 HelloSWT 中的 Shell 对象，但是这两个应用程序都需要 Shell 和 Display 对象来和操作系统进行通信。一个 SWT 应用程序需要一个独立的 Display 实例而 ApplicationWindow 则无论是在构造是带有空参数都会产生其自己的 Shell。这个类的关系在图 2.4 中有显示。表面上看来，无需如此复杂，但是当你是在构建大型的用户界面时使用 JFace 窗口的好处是显而易见的。

如同在本节开始提到的基于模型的适配器一样，ApplicationWindow 作为基于 Shell 类的 JFace 适配器有两个好处：其一，之前已经提到的，ApplicationWindow 把 GUI 的外观从其行为中分离开来；其二，它提供了对于设计人员相当有帮助的如何设定窗口的额外方式。虽然 Shell 类有方法可以改变其尺寸和风格，而 ApplicationWindow 所具有的方法对于定制化是大有裨益的。这些方法具体见图 2.3。如同在表中所见，ApplicationWindow 对象所具有的方法使得 GUI 编程大大简化。你可以快速地设定窗口的菜单栏、状态栏或是状态栏。这些方法还能设定应用程序的 exceptoin 句柄以及其默认图案。在 SWT 中，这样的能力需要对你产生的 shell 逐个设定，而在 JFace 中，这都是自动的。

表 2.3 ApplicationWindow 类中的设置方法

ApplicationWindow 方法 功能

addMenuBar() Configures the window with a top-level menu

addToolBar() Adds a toolbar beneath the main menu

addStatusLine() Creates a status area at the bottom of the window

setStatus(String) Displays a message in the status area

getSeparator() Returns the line separating the menu from the window

setDefaultImage(Image) Displays an image when the application has no shell

setExceptionHandler(IExceptionHandler)

Configures the application to handle exceptions according to the specified interface

2.3 开始 WidgetWindow（小部件窗口）应用程序

虽然 HelloSWT 和 HelloSWT_JFace 类对于学习基础 SWT/JFace 编程很有帮助，但是这一工具套件提供了如此多的功能，尚需我们进一步地浏览。为了避免多项目情况下的重复代码编写，我们考虑通过建立单一项目并在每章添加代码的方式应是最佳的。

为减少小部件窗口设计的负责度，我们决定同时使用 SWT 和 JFace。在本章中，我们将生成基本的窗口，我们强烈建议你把这个类加入到你的 com.swtjface.Ch2 程序包中。

```
package com.swtjface.Ch2;

import org.eclipse.swt.widgets.*;

import org.eclipse.jface.window.*;

public class WidgetWindow extends ApplicationWindow {

public WidgetWindow() {

super(null);

}

protected Control createContents(Composite parent) {

getShell().setText("Widget Window");

parent.setSize(400,250);

return parent;

}

public static void main(String[] args) {

WidgetWindow wwin = new WidgetWindow();

wwin.setBlockOnOpen(true);

wwin.open();

Display.getCurrent().dispose();

}

}
```

图 2.5 显示了虽然平淡但是重要的小部件窗口类的执行输出。presents the unexciting but important

2.4 小结

虽然你不得不要等到下面的章节才有机会作一些有趣和激动人心的图形界面，但是现在为止你应当已经相当稳固地掌握了 SWT 和 JFace 的内在机理。这些类库使得以平台独立的方式去接近平台的特定资源成为可能，

而且理解这些成就这一可能的对象也相当重要。

重要的对象是 Display，它在后台与操作系统保持着通信。这一通信确保你的 SWT/JFace 应用程序可以使用原生组件并处理事件。虽然 Display 自己没有任何外观，但其它的小部件都需要它的运作才得以成形。

SWT 提供一个 Shell 类作为大一统的 GUI 应用程序和对话框的容器。Shell 形成 GUI 的父窗口并为子小部件们和 Display 提供通信。使用风格值，你可以定制 Shell 的外观和行为。

与上面相对照的，JFace 的应用程序使用 ApplicationWindow 作为它们的主容器。不同于 Shell，ApplicationWindow 没有内置的 form。这意味着你可以随心所欲地指定你的顶级窗口；另外，这些对象提供的方法可以轻易与其他特性集成，如菜单、工具条或是状态栏等。

如你所见的，Shell 类和 ApplicationWindow 类中，SWT 和 JFace 类库提供了相近的功能。这种冗余（重复）性使得笔者在决定本书结构式颇费了一番周章。

最先，我们考虑最好是提供两个相互独立的部分，一部分涵盖 SWT，另一部分则讲述 SWT/JFace。但我们随即意识到这一方式不可行：首先，因为 JFace 提供几乎很少的小部件和容器，这两个部分的绝大部分代码会有重复；其次，在我们以后的章节中可以看到，如果要在 SWT 中实现复杂的功能，则需要惊人的代码量，而同样过程如过有了 JFace 则会大大简化。

因此，本书决定同步展示 SWT 和 SWT/JFace 开发。为了彻底阐明基本概念，我们打算在 SWT 中演示如何应用 GUI 特性，但是对于在小部件窗口应用程序中加入功能，我们强烈建议用两个类库编程。如此做法，我们认为不仅会增进你对 SWT 和 JFace 的理解，还能提升你对如何整合协同两套工具套件的感悟。

无论如何，提升你的感悟的最好方式就是用真实的小部件来构建真实的 GUI。看看它们是如何运作的吧！

第三章 小部件起步_1

本章内容涵盖：

- 小部件和控制
- 标签
- 按钮
- 合成器
- 更新小部件窗口

高兴之余，略现不足的是小部件的称谓仿佛是在暗示这是一个小玩意或是小发明之类不作为正式场合的应用。但是在学习 SWT 和 JFace 时，你必须很严肃地看待小部件。它们是你调色板上的颜料，是你碗橱里的调料。你对这个主题的理解会决定着你的应用程序表现和运行的优劣。

Eclipse 的设计者将小部件定义为：“任何 UI 对象都可以置于另一个小部件之内”。然而，撇开这一个递归的定义，我们拿出我们自己的定义，一个小部件是“在用户图形界面之中的任何一个对象，能够显示信息并且/或者允许用户和程序互动”

之前，处于某种目的我们已经使用了“对象”这个词，由于在 SWT/JFace 的 GUI 中的每一个小部件都是一个类的实例的可视化显现。本章的目的是展示这些类并演示你能如何设定它们的外观。此外，本章将会涵盖三个 SWT 的重要小部件类。我们将以最常用的小部件——标签开始。然后再在一个标签上添加输入能力并学习按钮类。最终，我们会讨论合成器，就是能够包容其他小部件的小部件。

但首先，我们要检查 Widget 类，这个类在整个小部件类层次的顶端；此外还有其最重要的子类，Control。

3.1 Widget 和 Control 类介绍

虽然我们的小部件定义作为纯概念这一点有所帮助，但是在写程序时几乎就一无是处了。因此，本节就试图开始描述在概念背后的类。我们将以 Widget 类和其相关的若干方法开始。然后，我们聚焦于 Control 类以及它的许多方法如何使得 GUI 设计人员象我们的生活般轻松。

3.1.1 理解 Widget 类

作为本书中所有小部件的先辈，Widget 类是你在学习 SWT 和 JFace 时非常重要的。它是所有在 SWT 和 JFace 中所有显示信息以及和用户互动的类的超类。然而不仅它是一个抽象类，另外 Eclipse.org 组织也强烈反对由其产生子类，因为其间牵涉太多的复杂性。因此，你不能从 Widget 类继承或是在你的代码中直接使用它。事实上，这个类的重要性是在于它把所有的小部件统合在一个结构下。

Widget 类中的方法代表了任何 SWT/JFace 小部件内在的基本能力。表 3.1 列示了这些方法和功能的一个重要子集。

SetData()方法允许应用程序以一个对象的形式向一个小部件附加信息。这在多个不同类要共享一个小部件或是小部件需要包含超越于通常其类所能提供的信息时可能特别有用；另外在一小部件有着全局域并且又必须要提供彼此不能直接通信的进程间提供信息时也相当有用。这个方法四通过将对象添加一个字符值来起作用的，而这个值当小部件被丢弃时释放。

下面的四个方法允许应用程序获取 Widget 的信息。

首先是 getData()，它返回小部件上所有经 setData()方法设定的数据值，相对于第二个方法（getStyle）仅返回字符串值。表中的下一个方法是 getStyle()，它返回一个整数值以代表该特殊小部件对象的外观设定。第四个方法是 getDisplay()，它返回与 GUI 相联系的 Display 对象。最后一个方法是 toString()，它返回一个与该小部件对应的类的字符串值。

这些能力是重要的，但是在实际的应用程序中 GUI 设计人员需要更多来设置小部件。由于这一原因我们需要研究 Control 类。

表 3.1 Widget 类的重要方法及它们的功能

小部件方法 功能

setData(String, Object) 将一个对象附着于小部件，并以 String 联系

getData() 返回小部件内所有于对象相关联的数据

getData(String) 返回和 String 相关的数据对象

getStyle() 返回和小部件风格对应的整数

getDisplay() 返回和小部件相联系的 Display 对象

toString() 返回代表小部件类的字符串值

dispose() 释放小部件及其资源

isDisposed() 依小部件是否被释放状态返回一个布尔值

3.1.2 和 Control 对象协同

我们在第 2 章曾经论及，SWT 和 JFace 使用又操作系统提供的小部件来渲染其图形应用程序。而由于不同的平台提供不同系列的 GUI 组件，SWT 仅能完全支持这些部件中的一个子集。

在 Control 类中的对象在操作系统中有一个直接的对应物，你可以通过类里的句柄域加以接近它。然而在 Control 类之外，SWT 还提供了不少的小部件。这一结构在图 3.1 中给予了描绘。你将要使用的小部件的主体部分，如：标签、按钮和合成器都是 Control 类的成员。作为一个结果，由于它们都和句柄相关，你可以使用在通用的 Widget 类中不可得得诸多方法来操控和设置这些对象。虽然本书中我们并不覆盖以上全部方法，但我们还是会给出两大类得方法来帮助你获得和指定一给定 Control 对象的特征（见表 3.2 和 3.3）。Control 对象的一个最为基本的属性就是它的尺寸。在表 3.2 中的第一个方法 getSize()返回一个 Point 格式的值。接下来的两个方法通过以宽度和高度抑或通过一个 Point 实例代表这些维度来设定小部件的尺寸。

余下来表中的方法处理一个 Control 的优选尺寸。指示 Control 需要显示其内容（可能综合了图案、文本、或其他小部件）所需要的最小尺寸。然后，程序通过 pack()方法来重新确定小部件的尺寸。你也可以通过使用这些带布尔值参数的方法来告诉布局管理器小部件的属性发生了改变。

注意：由于平台渲染和分辨率的差异，我们推荐你如有可能尽量使用 pack()方法，而非 setSize()。因为容器大小尺寸是由操作系统控制的，pack()方法可以确保你的容器的外观和内容较为大小得体。此外，你应注意必须在小部件们都已加入容器后才去调用 pack()方法。

表 3.2 获取和操控 Control 尺寸的方法

Control 方法 功能

getSize() 返回一个代表小部件尺寸的 Point 对象

setSize(int, int) 基于给定的长度和宽度设定小部件的尺寸

setSize(Point) 根据 Point 对象设定小部件的尺寸

computeSize(int, int) 返回需要显示全部小部件的需要的维度尺寸

computeSize(int, int, boolean)

返回需要显示全部小部件的需要的维度尺寸，并指明其特征是否被改变 pack() 重新改变尺寸到小部件的优选尺寸

pack(boolean) 重新改变尺寸到小部件的优选尺寸，并指明其特征是否被改变

表 3.3 设定 Control 位置的方法

Control 方法 功能

getLocation() 返回小部件相对于其父部件的位置

setLocation(int, int) 设定小部件相对于其父部件的位置

getBounds() 返回小部件的尺寸和相对于其父部件的位置

setBounds(int, int, int, int)

设定小部件的尺寸和相对于其父部件的位置

toControl(int, int) 变换相对于 display 的坐标为相对于 control 的 Point

toControl(Point) 变换一个相对于 display 的 Point 为相对于 control 的 Point

toDisplay(int, int) 变换相对于 display 坐标为一个相对于 control 的 Point

toDisplay(Point) 变化一个相对于 display 的 Point 为一个相对于 control 的 Point

表 3.3 中的 getLocation()方法返回一个 Point 包含有 Control 相对于其周围的小部件的坐标值。该坐标值可以通过 setLocation()方法来指定。下面的两个方法指代了小部件的边界，边界同时又是和其位置和尺寸息息相关的。GetBounds()方法返回了 Control 的 x、y 坐标和它的宽度和高度。

类似的，setBounds()方法需要四个整数来代表这些指标。当需要描述位置是，你需要一个参考点。对于 getLocation()方法，这个参考点是其小部件容器（通常是它的 Shell）的左上角。对于一个 Shell 对象，getLocation()方法返回其相对于用户控制台的坐标，并有 Display 对象代表。

使用这些图中的维度，shell.getLocatoin()返回(72, 66)，而 button.getLocation()返回(60, 40)。

使用表中的最后一个方法，Control 还能得到它们相对于 Display 对象的位置。在本例中，应用程序变换相对于 Shell 的坐标，称为相对于 control 的坐标；变换相对于 Display 的坐标，称为相对于 display 的坐标。这一翻译由 toDisplay()方法执行。与之相反地，转换相对于 display 的坐标为相对于 control 的坐标，是由 toControl()方法来实现的。

虽然我们没有叙述全部（甚至还没有过半）的与 Control 类相联系方法，但我们已经做了足够多来演示你能如何在一个用户界面中操控这些对象。现在你需要知道的是你能在你的应用程序中加入什么具体的 Control 子类。

3.2 标签

标签类是 Control 类中最简单的类。标签显示了 GUI 中静态的信息，诸如一个字符串或是图案，并且不接受用户输入。因为它们经常被使用，所以你需要熟悉它们的属性。本节将叙述标签背后的风格和方法，以及它们在代码中如何被应用。

3.2.1 风格和分隔符

字体和图案会在第 7 章里全部阐释，在此我们将关注在标签上显示基本信息的若干方法。决定一个标签的外观的基本参数是其风格，即在构造时指定的一个整数值。这和第 2 章里描述的 Shell 类的风格相似。

标签对象的文本相关的风格处理着字符串的排列，其值有 SWT.CENTER、SWT.LEFT 和 SWT.RIGHT。

额外的，你也可以用 SWT.SEPARATOR 风格设定标签单行显示。这些分割符可以设定为水平或垂直（SWT.VERTICAL、SWT.HORIZONTAL）；阴影或非阴影（SWT.SHADOW_IN、SWT.SHADOW_OUT 和 SWT.SHADOW_NONE）。

为了让你对标签（文本和分隔符标签）在程序中如何编写代码的有具体认识，考虑如下代码：

```
Label shadow_label = new Label(shell, SWT.CENTER);

shadow_label.setText("SWT.SHADOW_OUT");

shadow_label.setBounds(30,60,110,15);

Label shadow_sep = new Label(shell, SWT.SEPARATOR | SWT.SHADOW_OUT);

shadow_sep.setBounds(30,85,110,5);
```

第一个标签声名生成一个文本居中排列的标签对象。接下来两个方法设定了标签的显示字符串、尺寸以及在 Shell 中的位置。接下来的声名使用 SWT.SEPARATOR 风格来生成一个分割符组合 SWT.SHADOW_OUT 和 SWT.HORIZONTAL 以控制其外观。代码中唯一的标签指定方法是 setText()，它告诉对象该显示哪个字符串。

此外，还有一些与标签协同的方法值得书写。

3.2.2 标签方法

表 3.4 中列示了在 GUI 中操控标签对象的主要方法。如你所见，大部分都是直接的。

有一点比较重要需要牢记的是一个应用程序可以在完成构造标签后再行设定文本排列。此外，当我们在之后的章节中讨论图案对象时，getImage()和 setImage()方法会证明非常有帮助。

标签在 GUI 中经常使用并很有用。但是它们有时又是惹人烦的。如果它能和用户进行互动那会更加有趣，SWT 中用一个标签接口和组合调用一个按钮来实现这一能力。

3.2.2 标签方法

表 3.4 中列示了在 GUI 中操控标签对象的主要方法。如你所见，大部分都是直接的。

有一点比较重要需要牢记的是一个应用程序可以在完成构造标签后再行设定文本排列。此外，当我们在之后的章节中讨论图案对象时，getImage()和 setImage()方法会证明非常有帮助。

标签在 GUI 中经常使用并很有用。但是它们有时又是惹人烦的。如果它能和用户进行互动那会更加有趣，SWT 中用一个标签接口和组合调用一个按钮来实现这一能力。

3.3 用按钮来和用户互动

除了菜单外，在其他类型的组建中和用户作用更频繁的当数按钮了。按钮对象也是用户界面组建中最简单的了，因为它们功能被严格限定了：非开即关。本章中的按钮尚不能对用户的选择作出反应。但当我们在下一章中讨论 SWT/JFace 的事件模型时，你就可以使用监听器和适配器来对用户动作作出反应了。象 Shell 和标签一样，按钮在 GUI 中的外观依赖于其生成时的风格值。五个可用的风格可以导致按钮的外观和行为迥然不同，这些在接下来的子章节中会有描述。

表 3.4 获取和操控 Control 显现的方法

标签方法 功能

getText() 返回和标签相关联的字符串

setText(String) 将标签和一个字符串关联并显示

getAlignment() 返回一个整值以代表标签文本的排列方式

setAlignment(int) 根据 SWT 常量指定文本的排列方式

getImage() 返回与标签相关联的图案对象

setImage(Image) 将一个图案对象和标签相关联

3.3.1 用下按按钮和 SWT.PUSH 来引发动作

在 GUI 中最通常的按钮时下按按钮，其风格由 SWT.PUSH 常量来指定。这是 Button 类的默认风格值。和标签对象一样，一个按钮的文本是由 setText()方法指定的，并可由 getText()方法来获取。生成和设定一个下按按钮相对简单，如下为实例代码：

The most common type of Button in a GUI is the push button, whose style is

```
Button push = new Button(shell, SWT.PUSH);
```

```
push.setText("PUSH");

push.pack();
```

在设定按钮的文本之外，一个应用程序还可以控制按钮中字符串的排列和外观。按钮构造器可以用“|”操作符将任一的 SWT.LEFT、SWT.CENTER 和 SWT.RIGHT 风格与 SWT.PUSH 组合。在按钮资源分配好后，你还能使用 setAlignment()方法。这些不同的排列方式在图 3.4 中有列示。

图 3.4 还有一个可用的风格使所有按钮表面上浮。而 SWT.FLAT 风格 style 则将按钮表面与 GUI 平面持平。

3.3.2 继续箭头按钮和 SWT.ARROW

有的时候，一幅简单的图画可能比一段文字更能恰当地表达按钮的目的。常用的图画就是箭头，它能告诉用户可以继续浏览一篇文档、图画或是地图。箭头按钮也很容易生成，你可以通过把 SWT.ARROW 和 SWT.UP、 SWT.DOWN、SWT.LEFT 及 SWT.RIGHT 四者之一的组合来指明箭头的方向。这里有一段简短的代码：

```
Button push = new Button(shell, SWT.ARROW | SWT.RIGHT);

push.setText("RIGHT");

push.pack();
```

这些风格在图 3.5 中有图示。象下按按钮一样，箭头按钮也可以使用 SWT.FLAT 来扁平化。当然，如果你想在一个按钮对象中定制一幅图画，可以使用 setImage()方法附件一个图案对象并显示。在第 7 章中，你可以生成并操控这些图案。

3.3.3 用勾选按钮和 SWT.TOGGLE 改变状态

下按按钮和箭头按钮是用以执行动作，但是有的时候你所要的组件是用以对一个执行状态进行追踪。SWT 通过使用勾选按钮和在分配资源时设定 SWT.TOGGLE 风格来提供这一能力。勾选按钮和下按按钮相类似，但是它们可以追踪应用程序的状态信息而不是执行一个规程。此外，当一旦点选后，勾选按钮保持选中状态知道再下一次的点选。（见图 3.6）

勾选按钮是我们所讨论的第一个可以在外观上选中后维持一个状态的组件。在本例中，一个应用程序可以使用 setSelection(boolean)方法来设定按钮的状态，此处，布尔值为真代表选中按钮，为非代表未选中按钮。其他两种选中按钮和收音机按钮共享着这一能力，关于这一点，下文会有叙述。

3.3.4 用选中按钮和 SWT.CHECK 作选择

选中按钮的工作机理和勾选按钮相似，但是它们还需要以列表的形式协同生效。用户可以在一个系列里标记选中按钮选择一个或多个选择项。由于它们的方形的选择区域，这些组件通常也被成为选中盒。为了方便处理，我们推荐以数组的方式使用选中按钮，如下为实例代码：

```
Button[] checks = new Button[2];
```

```
checks[0] = new Button(shell, SWT.CHECK);

checks[0].setText("Choice 1");

checks[0].setLocation(10,5);

checks[0].pack();

checks[1] = new Button(shell, SWT.CHECK);

checks[1].setText("Choice 2");

checks[1].setLocation(10,30);

checks[1].pack();
```

通过生成数组，一个应用程序可以通过调用每一个按钮上的 getSelection()方法来遍历按钮值。此外，如果程序需要预先设定任何选择，可以用 setSelection()方法来标记默认的选择项。在图 3.7 中，setSelection()方法被用来标记在 Shell 里的第一盒第三个选择项。

3.3.5 用收音机按钮和 SWT.RADIO 作单一选择

有的时候，GUI 只需要作一个单一的选择，在这种情形下，选中按钮的开发属性，即允许用户随心所欲选择或多或少的选择项，是不可接受的。

你需要一种象收音机上的按钮一样的能力，当你选中一个的时候，另一个立刻不被选中。很自然地，这一能力可以通过由 SWT/JFace 调用产生自 SWT.RADIO 风格生成的收音机按钮来实现。

把收音机按钮集合于数组

象选中按钮一样，收音机按钮通常也是置于一个集合之中。操控这些按钮对象的一个常用的方法就是把它们放入数组中去，这项技术允许应用程序遍历每一个对象来获取或设定其参数，如下的代码即是：

```
Button[] radios = new Button[3];

radios[0] = new Button(shell, SWT.RADIO);

radios[0].setSelected(true);

radios[0].setText("Choice 1");

radios[0].setLocation(10,5);

radios[0].pack();

radios[1] = new Button(shell, SWT.RADIO);
```

```
radios[1].setText("Choice 2");

radios[1].setLocation(10,30);

radios[1].pack();

radios[2] = new Button(shell, SWT.RADIO);

radios[2].setText("Choice 3");

radios[2].setLocation(10,55);

radios[2].pack();

for (int i=0; i<radios.length; i++)

if (radios[i].getSelected()) System.out.println(i);
```

将这些代码放入 Shell 过程运行得如图 3.8。

许多套件提供了可以集合与管理收音机按钮组的特别组件。SWT/JFace 也提供了相似的能力，但这既不是在 Widget 类里面，也不是在 org.eclipse.swt.widgets 程序包内。RadioGroupField-Editor()类隐藏于 org.eclipse.jface.preference 程序包中，改类能够提供一种方法把收音机按钮们包容于一个单一对象之中。在本书之后的章节中我们还会有机会更深入地论及这一主题，但现在我们仅能提供一个简单的实例。样例代码生成一个 RadioGroupFieldEditor 对象，并植入三个收音机按钮标签：

```
RadioGroupFieldEditor rgfe = new RadioGroupFieldEditor("UserChoice", "Choose an option:",
1, new String[][] {{"Choice1", "ch1"}, {"Choice2", "ch2"}, {"Choice3", "ch3"}}, shell, true);
```

在构造器中的第一个参数提供了由编辑器返回的值的类型名；第二、第三个参数则指定了组的标签和其栏位的数目；第四个参数生成用相关的值生成一系列选择象的名字。用这种方式 RadioGroupFieldEditor 可以显示一系列的收音机按钮而无需分配按钮对象资源。第五个参数把编辑器加入到一个 Shell 对象，如图 3.9。

在 radioGroupFieldEditor 构造器中的最后一个参数指定了该收音机按钮是否和一组对象协同。这个对象被归类为一个容器小部件，因为它集合了一系列的小部件，并将他们在其边界以内显示。在 SWT/JFace 中，这一类容器对象由合成器类提供

3.4 用合成器来包含组件

虽然容器小部件不像它的内容那样有趣，但在任何 SWT/JFace 应用程序中它们都是必须的。它们组成 GUI 的背景结构并提供模块代码，这意味着多个合成器对象可以整合成一个单一的合成器。纵览这本书，我们样例代码中的类都是在扩展合成器类。

在本节中，我们将深度解析这些容器小部件。首先我们要探讨的是 Composite 类，其特征和其参数；然后

我们描述三个最为突出的合成器子类：分组、分栏和选项夹。最后一个子类因为担当这小部件窗口应用程序背景结构的缘故，在本书中显得特别重要。

3.4.1 理解合成器类

在处理第 2 章中 HelloSWT_JFace 应用程序的父对象时，曾提到过合成器。在那儿，ApplicationWindow 生成一个单一的合成器来作为其可视化部分。然而，合成器通常是用以在 GUI 和应用程序代码中组织小部件。象任何 Control 一样，它们可以在一个应用程序中改变尺寸和重新定位。合成器另外还有很多特性，具体详见表 3.5。

Table 3.5 Methods provided by the Composite class

Composite method Function

getChildren() Returns an array of Control objects

getLayout() Returns the layout associated with the Composite

setLayout(Layout) Specifies the layout associated with the Composite

getTabList() Returns an array of Control objects according to their tab

order

setTabList(Control[]) Specifies the tab order of the widgets within the Composite

这些方法给予了应用程序以管理在一个合成器内的小部件的能力。第一个方法是 getChildren()，它把一个合成器内的子小部件列表成 Control 对象的一个数组。接下来的两个方法是 getLayout()和 setLayout()，它们处理布局对象，而布局对象又指定了小部件如何在空间上排列（第 6 章我们会讨论布局）。getTabList()和 setTabList()方法获取和指定在一个合成器内选项卡的顺序，这个顺序就是用户在连续重复按 Tab 键时小部件被选中的顺序。

如图 3.1 所示，合成器类是可滚动类的一个直接子类。这意味着在 SWT/JFace 中所有的合成器对象都具有滚动条。另外，所有的可滚动对象都能使用表 3.6 中的方法来接近它们的维度和滚动条对象。

首先两个方法处理的是 GUI 设计最为关注的问题。虽然 getSize()能告诉你 Control 的总面积，但它却不能告诉你这面积中标题栏、滚动条和状态栏各占据了多少。这个合成器内不可编辑的部分称为它的 trim，即该区域是作为客户区域来使用的。这可以通过 getClientArea()方法来接近。在另一方面，如果你想知道你的应用程序到底需要多少客户区域（面积），你可以通过使用 computeTrim()方法来返回满足你的要求的合成器的尺寸值。

图 3.10 图形化的显示了合成器类、可滚动类以及它们的后代之间的关系。

3.4.2 分组

对于所有的合成器的子类，分组是最容易协同的，因为它事实上自己不执行任何动作，但它的确可以改进一个应用程序的外观和组织。必要时，它需要一一个给定的标签加上画一个包围其子小部件的矩形作明示。该

标签你可以用 setText()方法来指定文本。

分组边界用分割符紧密组合，具体有：SWT.SHADOW_IN、SWT.SHADOW_OUT 和 SWT.SHADOW_NONE 风格。

然而，你可以进一步地通过应用 SWT.SHADOW_ETCHED_IN 或者 SWT.SHADOW_ETCHED_OUT 风格得蚀刻来定制阴影效果。

表 3.6 可滚动类（合成器父类）提供的方法

合成器方法 功能

getClientArea() Returns the available display area of a Scrollable object

computeTrim(int, int,int, int)

Returns the necessary dimensions of the Composite for the desired client area

getHorizontalBar() Returns the horizontal ScrollBar object

getVerticalBar() Returns the vertical ScrollBar object

象许多小部件子类一样，分组类不能被扩展。因此，我们的 Ch3_Group，作为合成器类的子类，在其间生成一个分组对象。在本章末，这个子类将会和小部件窗口应用程序集成，所以我们推荐生成一个名为 com.swtjface.Ch3 的程序包，并插入 Ch3_Group 中。

```
package com.swtjface.Ch3;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

public class Ch3_Group extends Composite {

public Ch3_Group(Composite parent) {

super(parent, SWT.NONE);

Group group = new Group(this, SWT.SHADOW_ETCHED_IN);

group.setText("Group Label");

Label label = new Label(group, SWT.NONE);

label.setText("Two buttons:");
```

```
label.setLocation(20,20);

label.pack();

Button button1 = new Button(group, SWT.PUSH);

button1.setText("Push button");

button1.setLocation(20,45);

button1.pack();

Button button2 = new Button(group, SWT.CHECK);

button2.setText("Check button");

button2.setBounds(20,75,90,30);

group.pack();

}

}
```

这一段简洁的代码如图 3.11 所示，生成一个简洁的容器。

Ch3_Group 类不能被直接执行：它需要一个应用程序来调取它的构造器方法并在一个 Shell 或是 ApplicationWindow 中加入其对象。由于本书中的样本代码都结构化为合成器类，因此你需要生成一个简短的应用程序来观察合成器对象。该应用程序被成为 CompViewer，这个类生成一个 Ch3_Group 的实例，并将其添加如窗口的父合成器在 ApplicationWindow 中显示。

```
package com.swtjface.Ch3;

import org.eclipse.jface.window.*;

import org.eclipse.swt.widgets.*;

public class CompViewer extends ApplicationWindow {

public CompViewer() {

super(null);

}

protected Control createContents(Composite parent) {
```

```
Ch3_Group cc1 = new Ch3_Group(parent);

return parent;

}

public static void main(String[] args) {

CompViewer cv = new CompViewer();

cv.setBlockOnOpen(true);

cv.open();

Display.getCurrent().dispose();

}

}
```

分组类实现了一个合成器对象的主要功能，而其他容器提供了附件的功能。这可以从分割窗中得以体现，因为分割窗允许用户操控合成器子部件的尺寸。

3.4.3 分割窗

虽然分组容器适合于静态显示，而有时应用程序需要可以动态确定尺寸的控制。这样的 GUI 需要在一个有限空间内展现多个面板，并能使用户拉大一个尺寸的同时缩减另一个的尺寸。分割窗通过在其子小部件间设立可移动的栅栏来实现这一能力。这个被称为框格的栅栏允许用户在扩大一个小部件的尺寸的同时去缩减其他合成器内小部件的尺寸。

Sash 类和你的大部分的小部件一起位于 org.eclipse.swt.widgets 包内，但是 SashForm 类却在

org.eclipse.swt.custom 包内。

和分割窗相关的风格和方法主要都是用于确定栅栏位置以及子小部件膨胀和缩减的程度。你可以使用 SWT.HORIZONTAL 或是 SWT.VERTICAL 来指明栅栏的方向，或者你也可以使用 setOrientation()方法。分割窗类提供了一个被称为 getMaximizedControl()的方法，它将返回一个获得最大膨胀的 Control 对象。相似地，getWeight()方法则返回一个包含有各个分割窗子小部件权重数据的整数数组；而 setWeight()方法则使用一个整数数组来设定在合成器内的各个小部件的权重。

如下的 Ch3_SashForm 即为本章贡献的第二段关于合成器的样本代码。这个类以两个箭头按钮间用栅栏分割来展示 SashForm 的功能。

```
package com.swtjface.Ch3;

import org.eclipse.swt.*;
```

```
import org.eclipse.swt.custom.SashForm;

import org.eclipse.swt.widgets.*;

public class Ch3_SashForm extends Composite {

public Ch3_SashForm(Composite parent) {

super(parent, SWT.NONE);

SashForm sf = new SashForm(this, SWT.VERTICAL);

sf.setSize(120,80);

Button button1 = new Button(sf, SWT.ARROW | SWT.UP);

button1.setSize(120,40);

Button button2 = new Button(sf, SWT.ARROW | SWT.DOWN);

Button2.setBounds(0,40,120,40);

}

}
```

当这个合成器在 CompViewer 应用程序中例示是，你能的到结果如图 3.12。这幅图也显示了 GUI 如何在抬高和降低栅栏对象时的反应。

分割窗赋予你了控制 GUI 上的 control 的能力，但是有可能你想要的不仅局限于扩大缩小子小部件的尺寸。例如，如果在一个给定的 display 里有着太多的 GUI 元素，此时你需要一个合成器来把它们安排到一个逻辑分组内。这个将由我们本章讲述的最后一个合成器选项夹提供此功能。

3.4.4 选项夹

选项夹扩展了合成器的能力，即把多个合成器对象置于一个单一的容器内。本类的实例在一个象文件柜一样的结构中持有其他的容器，每一个都可以标号索引得到。

生成和组装选项夹的过程是简单的。在生成好主实例后，应用程序在选项夹内为每一个页面都构造了一个短小突出的对象称之为标号。然后它将调用带字符串阐述 setText()方法来设定 tab 的标签。最终使用 setControl()方法，当该 tab 被选中时，应用程序将此 Control 显示。

如下是关于此设置的样例代码。虽然选项夹构造器带有一个整型参数，但是该类没有特别的风格值。

```
TabFolder folder = new TabFolder(parent, SWT.NONE);
```

```
TabItem item1 = new TabItem(folder, SWT.NONE);

item1.setText("Tab Label");

item1.setControl(new SashForm(folder));
```

分割窗允许用户指定合成器子部件的相对尺寸。

另外，除了用于设定 tab 的方法，选项夹类提供如下方法以获取 TabItem 的信息：

- getItemCount() —获取在选项夹内的 TabItems 数目
- getItems() —返回 TabItem 对象的一个数组
- getSelection() —决定用户选中哪一个 TabItem
- setSelection() —在应用程序内作出选择

因为这一功能和模式在新的合成器内可添加，我们决定我们的小部件窗口应用程序就基于选项夹来开发。因此，我们将会把这个类加入到本书的主应用程序，而不是仅在此显示这一段简短的选项夹样例代码。

3.5 更新小部件窗口

从这一刻起，每一张都会都会包含一个小节的内容的图形话组件加入到生成自第二章的小部件窗口的 GUI 中。我们将执行两个任务来更新这一应用程序。首先我们将从本章添加两个合成器（Ch3_Group 和 Ch3_SashForm）到最终容器——Ch3_Composite 中。然后我们在小部件窗口中生成一个选项夹并形成第一个 TabItem。

3.5.1 生成 Ch3_Composite 类

现在你已熟知了合成器的编程，那么将两个容器集成于一个大的对象中就显得容易了。

```
package com.swtjface.Ch3;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

public class Ch3_Composite extends Composite {

public Ch3_Composite(Composite parent) {

super(parent, SWT.NONE);

parent.getShell().setText("Chapter 3 Composite");
```

```
Ch3_Group cc1 = new Ch3_Group(this);

cc1.setLocation(0,0);

cc1.pack();

Ch3_SashForm cc2 = new Ch3_SashForm(this);

cc2.setLocation(125,25);

cc2.pack();

pack();

}

}
```

如上，在一个合成器中加入另一个合成器就如同加入一个通常的 Control 一样容易。每一个容器都包含有起特性和功能。若想看到最终结果如何，我们还需要给小部件窗口应用程序添加一个该类的实例。

3.5.2 Creating the WidgetWindow TabFolder

就像你所看到的，选项夹的工作机理是一个直接了当的过程。你只需执行两步就可以把 Ch3_Composite 类加入到小部件窗口中去：导入 com.swtjface.Ch3 程序包，生成一个选项夹（该选项夹带有一个代表着 Ch3_Composite 类的 TabItem）。应用如下 com.swtjface.Ch2 的代码就可以实现小部件创扩程序的更新了。

```
package com.swtjface.Ch2;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.jface.window.*;

import com.swtjface.Ch3.*;

public class WidgetWindow extends ApplicationWindow {

public WidgetWindow() {

super(null);

}
```

```
protected Control createContents(Composite parent) {

TabFolder tf = new TabFolder(parent, SWT.NONE);

TabItem chap3 = new TabItem(tf,SWT.NONE);

chap3.setText("Chapter 3");

chap3.setControl(new Ch3_Composite(tf));

getShell().setText("Widget Window");

return parent;

}

public static void main(String[] args) {

WidgetWindow wwin = new WidgetWindow();

wwin.setBlockOnOpen(true);

wwin.open();

Display.getCurrent().dispose();

}

}
```

当你执行小部件窗口程序，结果就如同图 3.13。虽然程序相对简单不足以激动人心，但记住我们的话，该小部件窗口会在之后的每章后变得越来越有趣。

3.6 小结

本章深度钻研了 Widget 类和其许多的子类、方法。另外讲述了一些独立的组件外，我们清晰了在 SWT/JFace 小部件背后的类的结构。由于组件在谱系结构中所处的位置决定了相关的方法是否可用，这一点对我们很重要。本章的内容还相对简单，主要是集中于标签小部件、按钮小部件和系列的合成器子类。在每一处，我们都用代码样例展示了使用在这些小部件之上可用的风格和方法。除了简洁明了的样例展示，这些类同样可以为任何严肃认真的 GUI 开发人员所使用。

之后的章节我们假定你已经对之上的这些材料彻底理解了。

本章的主要缺陷在于其 GUI 都是静态的。用户按下按钮后，应用程序没有对此作出任何反应。本章的这些部件如同是图案一样。相似地，一个仅有一个 tab 的选项夹看上去就像是一个平实的合成器。

我们所需要的就是事件：我们需要应用程序能激活而不是被动消沉。这个话题下隐含的理论是复杂的，但其

提供的功能值得我们花上大力气研究一番。

让我们开始 SWT/JFace 的事件模型吧。

第四章 事件处理_1

本章覆盖内容

- SWT 的事件处理
- 类型化和非类型化的监听器
- 鼠标和键盘事件
- JFace 的事件处理
- 动作和实施

没有了事件，我们所看到的小部件和容器都是装饰品而已。

本章将聚焦于如何设置这些组件来理解和对用户的动作作出反应。值得一提的是，本章还描述了 SWT/JFace 获取这些动作并将它们翻译为所谓的“事件”的软件构件框架。对于这些事件的产生、接收和反应的过程被称为事件模型。许多关于 GUI 的书本都会将事件模型留到最末部分加以阐述，但我们这个主题很有必要早一点介绍引入。

本章第一部分描述了 SWT 促成应用程序处理事件的数据结构。这其间包括有当用户实施一个动作时产生的事件类，接收事件对象的监听器。将这些恰如其分地予以拼接组合，一个应用程序就能对可能发生的几乎每一种形式的事件提供多种反应。然而，SWT 的强有力的事件处理机制可能使得程序代码过于复杂。基于这样的原因，我们需要检验一下 JFace 是如何简化这一过程的。本章的第二部分就将研究使用 SWT 和 JFace 与用户接口的问题。JFace 的类库将使用事件和监听器取代在 SWT 中相对应部分的动作和实施，这两者有着异曲同工之妙。新的类通过将事件处理方法从 GUI 外观中分离出来而简化程序的事件处理过程。虽然 SWT 的动作和实施在窗口导向型的界面运行中功效也与之趋同，但是这在无形之中会缩小开发人员的视野范围并加重其编程负担。

4.1 SWT 中的事件处理

SWT 的事件处理循环如图 4.1 所示。它开始以操作系统记录和列示用户动作的事件队列。一旦当一个 SWT 应用程序开始运行，它的 readAndDispatch()方法对该队列检索排序，并检查和处理底层的操作系统消息队列？。如果它发现某些东西是和用户动作有关的，就会将该事件送达顶层的 Shell 对象，由 Shell 对象哪个小部件来接收这一事件。Shell 把该事件送达用户想要作用的小部件，也即把把信息传达到相关的用户界面，这就是一个监听器。监听器的某一个方法执行必要的过程或是调用另一个方法来处理用户的动作，被称为一个事件的处理器。

要想使一个小部件对应于一个事件，GUI 的设计人员的主要任务就是要确定哪一些事件需要作用，要产生和联系特定的监听器来感应这些事件，最终要构建事件处理器来执行必要的过程。本小节便显示了如何应用在 org.eclipse.swt.events 程序包内的 SWT 数据结构完成这些任务。

4.1.1 使用类型化监听器和事件

在 SWT 中绝大多数的监听器仅对某一特别系列的用户动作有针对反应。由于这样的原因它们被称为类型化的监听器，且它们就继承自 TypedListener 类。相似地，对应于这些特殊动作的事件被称为类型化事件，他们是 TypedEvent 类的子类。例如：一个鼠标单击或是双击就被一个鼠标事件取代。用户的键盘动作就被翻译为由 KeyListerner 侦听的键事件。表 4.1 中就列举了全部的这些类型化事件和监听器。

为了发挥功效，这些监听器必须和 GUI 上的组件相联系。例如：一个树监听器如果只和一个树对象相联系那么就只能接收树事件。但是并不是每一个 GUI 对象都能使用每一个监听器。例如，在 GUI 组件表栏内，一个 Control 组件能比一个追踪器对象广播多得多的类型的事件。还有一些事件如：菜单监听器、树监听器只能附着于特定的小部件。这种附着可以通过调用组件带有类型化监听器参数的 add...Listener()方法来实现。

Table 4.1 SWT 事件类和它们的相关监听器

事件 监听器 监听器方法 GUI 组件

ArmEvent ArmListener widgetArmed() MenuItem

ControlEvent ControlListener controlMoved() Control

controlResized() TableColumn

Tracker

DisposeEvent DisposeListener widgetDisposed() Widget

FocusEvent FocusListener focusGained() Control

focusLost()

HelpEvent HelpListener helpRequested() Control

Menu

MenuItem

KeyEvent KeyListener keyPressed() Control

keyReleased()

MenuEvent MenuListener menuHidden() Menu

menuShown()

ModifyEvent ModifyListener modifyText() Ccombo

Combo

Text

StyledText

MouseEvent MouseListener mouseDoubleClick() Control

mouseDown()

mouseUp()

MouseMoveEvent MouseMoveListener mouseMove() Control

MouseTrackEvent MouseTrackListener mouseEnter() Control

mouseExit()

mouseHover()

PaintEvent PaintListener paintControl() Control

SelectionEvent SelectionListener widgetDefaultSelected() Button

widgetSelected() CCombo

Combo

CoolItem,

CtabFolder

List

MenuItem

Sash

Scale

ScrollBar

Slider

StyledText

TabFolder

Table

TableCursor

TableColumn

TableTree

Text

ToolItem

Tree

ShellEvent ShellListener shellActivated() Shell

shellClosed()

shellDeactivated()

shellDeiconified()

shellIconified()

在表 4.1 的事件栏里列举的这些类型化事件的子类由 Display 和 Shell 对象送达类型化监听器。虽然程序员并不直接操控这些类，类里包含的成员领域提供有事件发生的信息。这一信息有利于事件处理器获取环境信息。如表 4.2 所示的即为从 TypedEvent 和 EventObject 类继承而来的领域。除了这些，许多事件类还提供其他的用户动作信息以外的领域。例如，MouseEvent 类也会包含一个按钮领域，它会告诉哪个鼠标按键被按下，以及鼠标动作时相对于小部件的鼠标 x，y 坐标值。ShellEvent 类包含一个布尔领域成为 doit，它能告诉你给定的动作是否导致了想要的结果发生。最后，PaintEvent 类提供了附加的方法我们会在第七章内讨论。

使用监听器编程

有两类主要的方法可以把监听器编入代码中。第一种是产生一个匿名类于组件的 add...Listener()方法，这会将监听器的范围缩小到仅该组件为止。这一方法在如下代码片断中有显示：

```
Button button = new Button(shell, SWT.PUSH | SWT.CENTER);

button.addMouseListener(new MouseListener() {

public void mouseDown(MouseEvent e) {

clkdwnEventHandler();

}
```

```
public void mouseUp(MouseEvent e) {

clkupEventHandler();

}

public void mouseDoubleClick(MouseEvent e) {

dblclkEventHandler();

}

});

static void dblclkEventHandler() {

System.out.println("Double click.");

}

static void clkdwnEventHandler() {

System.out.println("Click - down.");

}

static void clkupEventHandler() {

System.out.println("Click - up.");

}
```

在第一行中，生成一个按钮小部件并加入应用程序的 Shell 中。然后 addMouseListener()产生一个匿名的鼠标监听器接口并将其与按钮相联系。该接口包含三个方法：mouseDown()、mouseUp()和 mouseDoubleClick()能应用于鼠标监听器的任一实例。如果用户按下鼠标按钮、释放鼠标按钮或是双击，则一个鼠标事件就会送达这三个方法之一，然后会调用恰当的事件处理方法。这些事件处理器通过向控制台发送消息来完成事件处理。

虽然本例中的事件处理较为简单，但它们通常需要比事件处理的其他方面更多的努力。如果你需要接近一个外部类中的对象（由关键字 final 声名）时，一个匿名接口就显得很有帮助了。然而监听器却不能绑定于其他部件。为了解决这一问题，你可以通过声名一个单独的继承于 MouseListener 的接口。如下是样例代码：

```
Button button = new Button(shell, SWT.PUSH | SWT.CENTER);

button.addMouseListener(ExampleMouseListener);
```

```
MouseListener ExampleMouseListener = new MouseListener() {

public void mouseDoubleClick(MouseEvent e) {

System.out.println("Double click.");

}

public void mouseDown(MouseEvent e) {

System.out.println("Click - down.");

}

public void mouseUp(MouseEvent e) {

System.out.println("Click - up.");

}

};
```

之前的样例代码声明了 MouseListener 的所有三个成员方法。但是如果你仅关注于双击事件，并且你只需处理 mouseDoubleClick()方法即可，可是在使用 MouseListener 接口时，你却不得不声明它所有的方法。

幸运的是，你可以有机会使用一种叫做适配器的特殊类来消除这些不必要的代码。

4.1.2 适配器

适配器作为一种抽象类，是监听器接口的实现，并能提供每一个接口要求的方法作为默认的实现。这意味着当你把一个适配器，而非监听器，绑定于一个小部件时，你只需为你感兴趣的方法写上代码即可。虽然这谈不上极大的简化，但是它确确实实能给你省下大量的编程时间，由其是你在处理复杂的 GUI 时更是如此。

注意：本小节提到的适配器和我们在第二章内第一次提到的基于模型的适配器是绝对不同的。在此处，适配器削减了生成监听器接口时所需的代码数量；在 4.2 小节种你会看到基于模型的适配器可以简化事件处理，另外还提供了 GUI 编程的其他方面的帮助。

适配器进对拥有不止一个方法的监听器的事件才有意义。表 4.3 中列示了这些全部的类和相关的监听器类。

适配器对象编码容易，且由相同的 add...Listener()方法来生成即可。

```
button.addMouseListener(new MouseAdapter() {
```

```
public void mouseDoubleClick(MouseEvent e) {

dblclkEventHandler();

}

)};

static void dblclkEventHandler() {

System.out.println("Double click.");

}
```

就像我们看到的，使用 MouseAdapter 类允许你忽略与 MouseListener 接口相关的其他方法，把精力集中在鼠标双击事件即可。和监听器接口相似，适配器可以以匿名类或是本地类编程方式进行。

Table 4.3 SWT 适配器类和与之相关的监听器接口

Adapter Listener

ControlAdapter ControlListener

FocusAdapter FocusListener

KeyAdapter KeyListener

MenuAdapter MenuListener

MouseAdapter MouseListener

MouseTrackAdapter MouseTrackListener

SelectionAdapter SelectionListener

ShellAdapter ShellListener

TreeAdapter TreeListener

4.1.3 键盘事件

虽然绝大多数的如表 4.1 中的事件都是简洁明了易于使用和理解，键盘事件类还是需要进一步的解释。特别地，这些事件类包括有任一键击引发事件的 KeyEvent 类，以及它的两个子类 TraverseEvent 和 VerifyEvent。一个 TraverseEvent 引发自用户敲击箭头键或是 Tab 键来对焦于下一小部件的行为。而一个 VerifyEvent 则是因为一个用户敲击了一段文字需要程序来进行下一步动作前的验证而激起的。除了继

承自 TypedEvent 和 EventObject 类的领域外，KeyEvent 类还有三个成员领域提供由反映键触发事件的信息：

■ character —提供一个字符以代表所敲击的键；

■ stateMask —返回一个整数值代表键盘修改键状态。通过检验这一整数值，程序可以确定当前是否敲击了 Alt、Ctrl、Shift 和 Command 键的一个、部分或全部；

■ keyCode —提供对应于称为“键代码”的类型化键对应的 SWT 公共常量。这些公共常量在表 4.4 中有说明。

如下的代码片断展示了如何使用 KeyListener 来接收和处理一个键事件。它同样还使用到诸如(character、stateMask 和 keyCode)等领域来获取所按下的键的信息：

```
Button button = new Button(shell, SWT.CENTER);

button.addKeyListener(new KeyAdapter() {

public void keyPressed(KeyEvent e) {

String string = "";

if ((e.stateMask & SWT.ALT) != 0) string += "ALT-";

if ((e.stateMask & SWT.CTRL) != 0) string += "CTRL-";

if ((e.stateMask & SWT.COMMAND) != 0) string += "COMMAND-";

if ((e.stateMask & SWT.SHIFT) != 0) string += "SHIFT-";

switch (e.keyCode) {

case SWT.BS: string += "BACKSPACE"; break;

case SWT.CR: string += "CARRIAGE RETURN"; break;

case SWT.DEL: string += "DELETE"; break;

case SWT.ESC: string += "ESCAPE"; break;

case SWT.LF: string += "LINE FEED"; break;

case SWT.TAB: string += "TAB"; break;

default: string += e.character; break;
```

```
}

System.out.println (string);

}

});
```

这段代码使用了键事件的领域和公共常量来生成一个字符串来显示按键的数量和相关的修改符键。在整个事件处理器操作中的第一步包括检查一个事件的 stateMask 领域以观察 Alt、Ctrl、Shift 和 Command keys 是否被按下。如果是，修改器键的名称就会加入到这一字符串。之后该方法就不断检查时间的 keycode 是否和支持的键的字母字符相对应。在其他案例中，键的名称会附加于字符串并送达控制台。

当用户按下按键来从一个组件处理到另一个组件时如一组按钮或是选择框 TraverseEvent 被激发了。本类中包含的两个领域可以让你控制横向动作是否可以将对焦移向另一个 control，或是该对焦继续停留在激发该事件的小部件上。最简单的领域，doit，是一个布尔值，它可以允许（TRUE）或是禁止（FALSE）给定小部件上的横移。TraverseEvent 类的第二个领域，detail，稍微复杂。它是一个整数值，代表着引发事件的不同的按键。例如，如果用户按下 Tab 键来转换到一个新的组件，detail 领域将会包含 SWT 常量 TRAVERSE_TAB_NEXT。

每一个类型的 control 对应于给定的横向键都会由一个不同的默认行为。例如，一个由 TRAVERSE_TAB_NETX 的动作导致的 TraverseEvent，默认情况虾，如果是一个收音机按钮就能引发一个横向移动，而对于画布对象则不能。因此，通过设定 doit 领域为 TRUE，你就可以不必考虑默认的设置情况而允许用户进行横向移动；而将该领域设定为 FALSE，则可以将对焦固定于该组件。

VerifyEvent 的使用和 TraverseEvent 相似。其目标是为了预先决定是否用户的动作会导致通常的或是默认的行为。在本例中，你可以检查用户的文本以决定它在应用程序中是应当更新还是被删除。该类的两个领域：start_and_end，指明了输入的范围，而文本领域则包含了检查过的输入字符串。在检查了用户的文本之后，你可以设定 doit 的领域值：TRUE 允许、FALSE 禁止该动作。

4.1.4 用非类型化事件来定制事件处理

类型化的事件和监听器可以用清楚地表白了其任务的类和接口来处理事件。更进一步的说，类型化的监听器提供了特别的方法来接受和处理这些事件。籍以缩小监听器和事件的处理范围来面对特定的动作，类型化的组件减少了错误代码发生的可能性。

然而，你如更爱好编程的灵活性胜过其安全性，那么 SWT 还提供了所谓的非类型化的事件和监听器。当一个代表着非类型化监听器的监听器类和 GUI 的某一组件相联系时，它就能接受该组件所能发送的任一类事件。因此，你将不得不操控这由 Event 类代表的捕获的全部事件，决定用户执行的那个动作。然后，正确的事件处理方法就被调用。

需要注意的是 Eclipse.org 并不推荐使用非类型化的事件和监听器，实际上，原话是这样说得“不倾向于在应用程序中使用”。这一机制也并未同其类型化的部件包含 org.eclipse.swt.events 程序包内。

事实上，非类型化的监听器、接口和事件类都位于 org.eclipse.swt.widgets 程序包内。

尽管入席，由 Eclipse 网站提供的 SWT 代码片断专门使用了非类型化的监听器和事件。由于你可以生成一个针对特定系列事件而反应的定制化的监听器，所以编程相对简易。如下为样例代码：

```
Listener listener = new Listener () {

public void handleEvent (Event event) {

switch (event.type) {

case SWT.KeyDown:

if (event.character == 'b')

System.out.println("Key"+event.character);

break;

case SWT.MouseDown:

if (event.button == 3)

System.out.println("Right click");

break;

case SWT.MouseDoubleClick:

System.out.println("Double click");

break;

}

}

};

Button button = new Button(shell, SWT.CENTER);

button.addListener(SWT.KeyDown, listener);

button.addListener(SWT.MouseDown, listener);

button.addListener(SWT.MouseDoubleClick, listener);
```

在这段代码中，监听器对象发送任何事件实例给它的单个方法，handleEvent()。然后，该事件的类型领域决定需要作出什么样的处理。如果该事件是 SWT.Keydown 类型，且字符是字母 b，然后一段声明被送往

控制台。如果是 SWT.MouseDown 类型且第三个鼠标键被按下（即单击鼠标右键），然后展示了右击的声明。如果一个 SWT.MouseDoubleClick 事件被激发，然后鼠标双击被显示。

当然，你也可以通过使用类型化的事件和监听器来获取这一能力，但是处理起来就显得麻烦了。按钮需要添加有对应适配器的鼠标监听器和键监听器。然后你需要在何时的监听器方法内放置事件处理的规定。很明显地，非类型化事件处理在本例中不仅方便，而且减少了处理该事件所需要的类的数量。

为了取代类型化事件，Event 类包含了在每个类型化事件中所有的领域。它有着和 KeyEvent 一样的字符领域以及和 MouseEvent 一样的按键领域。在前面的这段代码中，它还有着一个叫做类型（type）的领域，该领域指代着该事件种类，表 4.5 中累世了这些类型（type）。

表 4.5 SWT 中 Event 类的类型值

type 领域的值

SWT.Activate SWT.FocusIn SWT.KeyUp SWT.Move

SWT.Arm SWT.FocusOut SWT.MenuDetect SWT.None

SWT.Close SWT.Expand SWT.Modify SWT.Paint

SWT.Collapse SWT.HardKeyDown SWT.MouseDoubleClick SWT.Resize

SWT.Deactivate SWT.HardKeyUp SWT.MouseEnter SWT.Selection

SWT.DefaultSelection SWT.Help SWT.MouseExit SWT.Show

SWT.Deiconify SWT.Hide SWT.MouseHover SWT.Traverse

SWT.Dispose SWT.Iconify SWT.MouseMove SWT.Verify

SWT.DragDetect SWT.KeyDown SWT.MouseUp

4.1.5 一个 SWT 监听器/事件应用程序

在我们讨论 JFace 事件模型之前，我们将献上一个 SWT 的合成器，它集成和总结了所涵盖的内容。下面显示的这个类包含了两个按钮，一个标签以及必要的事件处理。我们推荐在你的项目中生成一个 com.swtjface.Ch4 的程序包，并将该类添加进去。

```
package com.swtjface.Ch4;

import org.eclipse.swt.events.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.*;
```

```
public class Ch4_MouseKey extends Composite {
Label output;
Ch4_MouseKey(Composite parent) {
super(parent, SWT.NULL);
Button typed = new Button(this, SWT.PUSH);
typed.setText("Typed");
typed.setLocation(2,10);
typed.pack();
typed.addKeyListener(new KeyAdapter() {
public void keyPressed(KeyEvent e) {
keyHandler();
}
});
Button untyped = new Button(this, SWT.PUSH);
untyped.setText("Untyped");
untyped.setLocation(80,10);
untyped.pack();
untyped.addListener(SWT.MouseEnter, UntypedListener);
untyped.addListener(SWT.MouseExit, UntypedListener);
output = new Label(this, SWT.SHADOW_OUT);
output.setBounds(40,70,90,40);
output.setText("No Event");
pack();
}
```

```
Listener UntypedListener = new Listener() {
public void handleEvent(Event event) {
switch (event.type) {
case SWT.MouseEnter:
output.setText("Mouse Enter");
break;
case SWT.MouseExit:
output.setText("Mouse Exit");
break;
}
}
};
void keyHandler() {
output.setText("Key Event");
}
}
```

第一个按钮与一个匿名的类型化监听器相联系以接受键盘事件；一个非类型化的监听器接口加入到第二个按键以捕获当鼠标指针进入或退出按钮区域的事件。无论何时由哪个按钮激发某一事件，一个字符串将送达标签。通过将该合成器集成到上一章中的 CompViewer 应用程序中，组装了的 Shell 显示如图 4.2。

本代码中的 SWT 结构允许一个小部件接收许多类型的事件并提供了许多不同的反应。但在大多数的 GUI 中，这并非必需。在这些案例中，SWT 宽泛的能力只能导致时间处理编程的复杂度提高。以能力换取简易性的愿望终于从 JFace 的事件处理模型中得到了满足。

第四章 事件处理_3

4.2 JFace 中的事件处理

一个监听器接口能够为不同的 control 提供相同的事件处理，但是其使用依赖于发起这一事件的组件。接收鼠标事件的监听器不能用以菜单栏的选择。即便是非类型化的事件也只是在程序指定了哪一类的 control

来触发事件后才有用。

但是当你处理复杂的用户界面时，将事件处理能力从产生事件的 GUI 组件中分离出来的做法就很有帮助了。这将允许一组人单独关注于 GUI 的事件处理，而同时另一组人关注与外观的设计。另外，如果一个监听器能够添附与任意一个组件，那么其代码就有重用的可能。最后，如果程序的一段代码是紧密地集中于 GUI 的外观，而另外的是仅关注与事件的处理。那么代码就相对容易开发和理解。

Jface 用其 Action 和 ActionContributionsItem 类提供了这样的分离。一个 ActionContributionItem 组合了一个 GUI 组件的功能和添附于其上的监听器类。当用户与其互动时，将会触发与其相联系的动作类，以负责处理该事件。虽然这看上去和 SWT 的监听器/事件模型相似，但是这些类却更加抽象，易于使用，范围相对专一。

图 4.2Ch4_MouseKey 合成器

这个样例结合了 SWT 类和接口的许多类型的事件处理。

因为这些类比 SWT 的更加抽象，所以需要假以时日来体会其优点。然而，一旦你懂得和理解了它们，我们确信你就会在处理重复性的事件处理时去经常使用它们。这个可以通过程序样例代码作很好得说明。但是，首先我们在之前还是要做一个技术性的介绍。

4.2.1 理解动作和实施

虽然你能处理 TraverseEvent 和 ArmEvent 这一点令人兴奋，但事实上极少有程序去使用它们。另外，试想如果一个小部件上可以添附多个监听器和事件处理器，这将是令人着迷的；但是 GUI 的组件通常仅能对应于某一单一的输入作出单一功能执行。因为 SWT 的结构提供了每一个可接受的组件并将其和事件相结合，甚至连最简单的监听器/事件都会很复杂。

如果存在这样一组套件，它仅关注于平时经常使用的小部件和事件，并将这一切的使用尽可能作简化，那将极大地简化事件编程。JFace 的事件处理结构所做的正是如此：它的目标就是为了时事件处理更加简介明了，它允许程序员仅使用少量的几行代码来接收和处理通常的事件。为了达成这一目标，JFace 做了如下三个假设：

■ 用户的动作将限于按钮、工具条和菜单

■ 每一个组件将仅有一个事件与之相联系

■ 每一个事件仅有一个事件处理器

在考虑了这些假定之后，JFace 极大地简化了事件处理机制。

第一个假定意味着实施只需选取三种形式之一。第二个假定提供了实施从其相联系的动作的分离，即，每一个组件仅触发一个事件而并不管是由什么动作触发或是那个组件激发了这一事件。第三个假定意味着每一个动作仅需要一个事件处理的规程即可。图 4.3 显示了这个 SWT/JFace 简化的事件模型。象 SWT 的事件模型一样，界面以 Display 类追踪操作系统事件队列来开始。此时，它将信息传递给包含有 Display 的 Shell 对象的应用程序。该应用程序产生一个 Actionb 类然后将它传送到产生原始事件的实施。该实施然后调用 Action 类中 run()方法来作为单一的事件处理器。

Action 类的行为类似于 SWT 的 Event 类，但是 contribution 的功能相对复杂。有两个主要的 contribution 类是 ContributionItem 类和 ContributionManager 类。ContributionItem 类提供独立的 GUI 组件来触发动作，而 ContributionManager 则负责产生包含有 ContributionItem 的对象。因为这两个都是抽象类，真正的事件是由它们的子类来完成的。

图 4.4 显示了这个继承关系图

虽然 ActionContributionItem 是 ContributionItem 的诸多具体型的子类之一，但它是最重要的。这个类被用以在一个应用程序内将一个动作联系到 GUI 上。它虽没有设定好的外观，但是依赖于你使用的 fill() 方法，却可以帮助一个按钮、菜单栏和工具栏的成形。

要在一个应用程序中配合 contributions 需要使用到 ContributionManager 子类。这些子类可看作是 ContributionItems 的容器，整合它们可以改善 GUI 的组织和简化编程。MenuManager 类在一个窗口的顶级菜单整合 ContributionItems，而 ToolBarManager 类则将这些对象放置在位于菜单之下的工具条中。

4.2.2 生成动作类

如下将从抽象类 Action 中生成一个子类称为“Ch4_StatusAction”。这个类的作用是当事件被触发时将一个字符串送达应用程序窗口的状态栏内。我们推荐你将此类加入到你的项目目录中去。因为这个类将被应用到一个工具条中去，且它需要和图案相联系。最简单的做法是进入到$ECLIPSE_HOME/plugins/

org.eclipse.platform_x.y.z 目录中，拷贝文件 eclipse.gif 将其粘贴到当前的项目文件夹。

```
package com.swtjface.Ch4;

import org.eclipse.jface.action.*;

import org.eclipse.jface.resource.*;

public class Ch4_StatusAction extends Action {

StatusLineManager statman;

short triggercount = 0;

public Ch4_StatusAction(StatusLineManager sm) {

super("&Trigger@Ctrl+T", AS_PUSH_BUTTON);

statman = sm;

setToolTipText("Trigger the Action");

setImageDescriptor(ImageDescriptor.createFromFile

(this.getClass(),"eclipse.gif"));
```

```
}

public void run() {

triggercount++;

statman.setMessage("The status action has fired. Count: " +

triggercount);

}

}
```

观察该类的第一件事情是它还有什么没展示。虽然构造器接收到一个 StatusLineManager 对象进行输出展示，但是 Ch4_StatusAction 对于什么样的组件激发了其动作一无所知。因此任何能产生动作的 control 都可以与 Ch4_StatusAction 相联系而无需额外的代码。另外，仅有一个事件处理规程 run()，反观 SWT 中的事件却有多个处理器与之联系。

Run()方法处理事件，但是事实上在本类中的主要工作是由构造器完成的。首先，它从其超类中调用构造器，动作并初始化其 TEXT 和 STYLE 领域。这种方式下，如果 Ch4_StatusAction 是和一个菜单协同，那么菜单项目也可以荼毒触发器。在 T 字母之前的&符号意味着这个字母将作为该动作的快捷键。而在 TEXT 领域内的“Ctrl+T”确保了当用户在同时按下 Ctrl 键和 T 键时该动作就会被激发。

在 action 构造器之下，还需要调用进一步的方法来设置其 GUI 的外观。弱国这是在一个合成器内实施的，那么 Ch4_StatusAction 类建会根据 AS_PUSH_BUTTON 风格，而不是 AS_RADIO_BUTTON 或 AS_CHECK_BUTTON 风格来成形。接下来，setToolTipText()方法将初始化类的 TOOL_TIP_TEXT 领域，一旦当鼠标指针在工具条单元上盘旋就会生成一个字符串。最终，构造器会将一幅图案和 Ch4_StatusAction 类相联系，使其在工具栏额按钮上显现。

每次当 Ch4_StatusAction 被生成，run()方法就被调用。在本例中，触发计数累加器被更新，然后一则消息被送达 StatusLineManager 对象。在绝大多数的应用程序中，该方法被频繁使用以作为事件处理之需。

4.2.3 在一个应用程序中应用 contributions

因为动作和实施只能和按钮、工具条单元、和菜单单元相联系，任何应用程序必须依赖这些组件来证明其能力。所以，虽然正式的对这些小部件的介绍不得不在之后的章节中奉献，我们必须在此处将它们叙上一段。如下展示了 ContributionItem 和 ContributionManager 类被添加到一个窗口。ActionContributionItem、 MenuManager 和 ToolBarManager 这三个 contributor 类都能够触发 Ch4_StatusAction。这一动作将会发送一则消息到窗口底部的状态栏。我们推荐你在 com.swtjface.Ch4 中生成 Ch4_Contributions 类然后用同目录中的 Ch4_StatusAction 类来执行之。

注意：在许多平台上，除非 OSGi 库函数被添加，否则 Contribution 会无法运作。基于这个原因，我们推荐你生成一个 OSGI_LIB 变量并将之同位于$ECLIPSE/plugins/osgi_x.y.z/内的 osgi.jar 相匹配。附录 A 中详细介绍了如何添加 classpath 变量。

OSGi 指的是 Open_Services_Gateway_Initiative，即用以形成和智能消费类电子设备、汽车和家用设

备的联网。虽然在写本书的时候其广泛采用还未能非常确定，但是有一点是肯定的，就是 IBM 极希望其能成功。

```
package com.swtjface.Ch4;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.jface.window.*;

import org.eclipse.jface.action.*;

public class Ch4_Contributions extends ApplicationWindow {

StatusLineManager slm = new StatusLineManager();

Ch4_StatusAction status_action = new Ch4_StatusAction(slm);

ActionContributionItem aci = new

ActionContributionItem(status_action);

public Ch4_Contributions() {

super(null);

addStatusLine();

addMenuBar();

addToolBar(SWT.FLAT | SWT.WRAP);

}

protected Control createContents(Composite parent) {

getShell().setText("Action/Contribution Example");

parent.setSize(290,150);

aci.fill(parent);

return parent;

}
```

```
public static void main(String[] args) {
Ch4_Contributions swin = new Ch4_Contributions();
swin.setBlockOnOpen(true);
swin.open();
Display.getCurrent().dispose();
}
protected MenuManager createMenuManager() {
MenuManager main_menu = new MenuManager(null);
MenuManager action_menu = new MenuManager("Menu");
main_menu.add(action_menu);
action_menu.add(status_action);
return main_menu;
}
protected ToolBarManager createToolBarManager(int style) {
ToolBarManager tool_bar_manager = new ToolBarManager(style);
tool_bar_manager.add(status_action);
return tool_bar_manager;
}
protected StatusLineManager createStatusLineManager() {
return slm;
}
}
```

在 JFace 的应用程序和先前章节中的应用程序间的区别在于对动作和实施的介绍。在累得声明下，程序用一个 StatusLineManager 对象作为参数构造了一个 Ch4_StatusAction 的实例。然后，它产生一个 ActionContributionItem 对象并以 Ch4_StatusAction 实例来区分。此时 contribution 还未成形，仅是

将一个动作联系于用户界面的高层次的一个简单的方法。构造器方法生成一个应用程序对象并将一个菜单、工具条和状态栏添加进去。

createContents()方法设定了窗口的标题和尺寸然后调用 aci.fill()方法，该方法之所以重要是其将 ActionContributionItem 对象放到了 GUI 中去。在本例中因为 fill()方法的参数是个合成器对象，contributor 负责按钮的成形，该按钮如被按下则触发一个 StatusEvent 事件。

在 ch4_Contributions 中的最后三个方法也是简洁明了的。Main()方法负责生成和打开窗口然后调配 GUI 资源。然后，createMenuManager()方法在窗口顶端生成一个菜单实例。由于它是 ContributionManager 的子类，那样就可以和一个 Action 对象绑定，因此 status_action 对象就可以用 add()方法绑定了。该方法也可用于 createToolBarManager()方法和 action 实例的绑定。在这两个案例中，一个 ActionContributionItem 已经暗中以菜单项形式添加到了菜单，以工具项的形式添加到了工具条。

图 4.5 展示了 Ch4_Contributions 的图形界面。在底部的状态栏中一个持续运行的计数器在计量着触发事件的 Ch4_StatusAction 的数目。

4.2.4 和 contributions 互动

有两个方法可以将 ActionContributionItem 整合到一个 GUI 中去。第一个是使用 ContributionManager 子类的 add()方法，如同在 Ch4_Contributions 应用程序中的 MenuManager 和 ToolBarManager 的行为一样。第二个方法就是使用绑定于 ActionContributionItem 类的 fill()方法，并辅以一个 SWT 小部件作为其参数。如果该参数是个 Ch4_Contributions 内一样的合成器，那么 contributor 的呈现将会由该动作的
STYLE 属性来决定。如果该参数是一个 SWT 菜单对象，那么 contributor 将会形成一个菜单项。最后，如果该参数是一个 SWT 的工具条对象，那么 contributor 将会形成一个工具条内的工具项。表 4.6 内展现了 fill()方法的特性。

图 4.5 Ch4_Contributions 应用程序显示了一个 ContributionItem 能够整合到一个窗口中的三种方法。

表 4.6 ActionContributionItem 的 fill()方法重载和其绑定的外观

fill() GUI 应用执行（外观）

fill(Composite) 更具 Action 的 STYLE 属性

fill(Menu, index) 带索引位置的菜单项

fill(ToolBar, index) 带索引位置的工具项

作为 ContributionManager 类的一个有趣的特征就是其重载的 add()方法既接受 Action 类作为参数，也接受 ActionContributionItem 类作为参数。所以，你可以隐式地将一个 ContributionManager 绑定于一个 ContributionItem（使用 Action），或者也可以显式地绑定（使用 ActionContributionItem）。但其间有一个重要的区别：你可以重复性的使用相同的 Action 对象隐式地绑定于 contribution，如同我们在

Ch4_Contributions 类中看到的那样。而显示地 contribution 绑定则只能执行一次。

表 4.7 Action 类的重要方法

Action 方法 功能

run() 执行绑定于 Action 的事件处理

Action() 默认构造器

Action(String) 初始化 TEXT 领域的构造器

Action(String, ImageDescriptor) 初始化 TEXT 领域并将一图案绑定于 Action 的构造器

Action(String, int) 设定 TEXT 和 STYLE 领域的构造器

4.2.5 浏览 Action 类

虽然 Ch4_StatusAction 易于编程和理解，但你还需要对 Action 类的其他方面了然于胸。Action 类内含了大量有助于提升你的用户界面能力的方法。这些方法划分类别并在下面列示。在表 4.7 中显示第一系列的方法对于 Action 类的任何应用都是重要的。第一个也是最重要的方法是 run()。正如我们早先提到的那样，在一个 Action 类中，这是一个单一的事件处理规程，当动作被触发事，它每次都会被调用。另外，构造器初始化绑定于 Action 类的成员领域，这一点我们很快会谈到。在 Ch4_StatusAction 的代码样例中，一个 Action 类的实例包含有许多领域以提供关于在 GUI 中 Action 表现的信息。你可以使用表 4.8 中列举的这些方法来接触和操控这些领域。有前两个方法设定的 TEXT 领域包含有一个显示为标题或菜单项的字符串。挨下来的两个方法处理的事 DESCRIPTION 领域，可以向一个状态栏写入以提供额外帮助信息。当用户的鼠标指针停靠于某一个 contributor 上，则在 TOOL_TIP_TEXT 领域中的字符串就会被显示。表中的最后两个方法设定和涉及了 Action 类的 IMAGE 的属性，该属性代表着 ImageDescription 类的对象。我们将在第七章中进一步解释一个 ImageDescriptor 并不是一个图案，而是一个持有如何生成一个图案的信息的对象。在 Action 类中包含的最后一个领域是 STYLE。该整数值有一个构造器设定并有位列于表 4.9 的顶端的 getStyle()方法存取。接下来的两个方法，setEnabled()和 getEnabled()决定了被 Action 对象绑定的组件能否被用户动作。如果不能，默认情况下它们编程灰色虚影。最后的一组方法，setChecked()和 isChecked()，如果是 Action 对象绑定于一个收音机按钮或是一个选中盒，会比较有用。它们被用于设定按钮的默认状态或是决定是否用户已经选中了它。

Table 4.8 Action 类的属性方法

Action 属性方法 功能

setText(String) 设定 TEXT 领域

getText() 返回 TEXT 领域

setDescription(String) 设定 DESCRIPTION 领域

getDescription() 返回 DESCRIPTION 领域

setToolTipText(String) 设定 TOOL_TIP_TEXT 领域

getToolTipText() 返回 TOOL_TIP_TEXT 领域

setImageDescriptor(ImageDescriptor) 设定 IMAGE 领域

getImageDescriptor() 返回 IMAGE 领域

getStyle() 返回 STYLE 领域

setEnabled(boolean) 设定 ENABLED 领域

getEnabled() 返回 ENABLED 领域

setChecked(boolean) 设定 CHECKED 领域

isChecked(void) 返回 CHECKED 领域

表 4.10 中显示了处理快捷键和键盘转换的方法。快捷键属于键盘上的快捷方式，完成者和鼠标单击相同的功能。正如同我们在 4.1.4 节中谈到的那样，按键由一系列的整数键代码来代表，这里面包括有所有的字母和编辑键（如：Alt、Ctrl、Shift 和 Command）。第一个方法为 Action 对象产生一个快捷键并用一个 SWT 键代码绑定它。接下来的方法为 Action 的快捷键提供了键代码。后面两个方法来回不停地在键的键代码和器所代表的字符串之间转换。RemoveAcceleratorKey()方法解析文本并删除 Action 的快捷键。最后的四个方法提供了键盘上字符键和编辑键的代表字符串和它们的 SWT 代码代表之间的转换。表 4.11 中的首两个方法负责绑定或释放 PropertyChangeListeners 你可以使用定制化的 listener/event

表 4.10 Action 类的快捷键和键盘方法

键盘方法 功能

setAccelerator(int) 设定键代码作为 Action 的快捷键

getAccelerator() 返回 Action 快捷键的键代码

convertAccelerator(int) 将快捷键转换为一个字符串

convertAccelerator(String) 转换一个字符串为一个快捷键

removeAcceleratorText(String) 从给定字符串移除快捷键

findKeyCode(String) 转换一个键名为一个 SWT 键代码

findKeyString(int) 转换一个键名为一个键代码

findModifier(String) 转换一个编辑状态名为一个编辑键代码

findModifierString(int) 转换一个编辑键代码为一个编辑状态名

虽然 JFace 使用了 action 来取代 SWT 中的监听器/事件机制，但是在特殊情况下 Action 类却仍然能和监听器协同工作。这些方法在表 4.11 中有显示它们通常只关注用户界面的 Iproperty-ChangeListener 接口。这一接口关注于用户定制的 PropertyChangeEvents，即当一个给定的对象按照你所定义的方式变化为另一不同的对象。虽然处理属性变化看上去有些负责，但是它们可以允许你创立定制化的监听器/事件关系，而非由 SWT 提供的关系。在表 4.11 中的首先两个方法负责绑定或是释放 PropertyChangeListener。你可以使用接下来的两个方法，通过触发基于预定义的事件类或是给定对象内的指定变化之属性改变来测试这些监听器。该表内的最后一个方法与 HelpListeners 相关，该监听器处理用户需要获取给定组件的信息时提供帮助。

表 4.12 列举了一组不同的包含有 Action 类的方法。首四个方法用于获取 Action 类和其定义的标识符；接下来的两个方法 setMenuCreator()和 getMenuCreator()，和 IMenuCreator 接口一起可被绑定于一个 Action 对象。该接口在由一个特定的事件触发后提供一个简单的生成下拉或是弹出菜单的方法。当一个 Action 的 ENABLED 领域被设定为 FALSE 时，你可以通过 setDisabledImageDescriptor()方法来指定哪一个图案代表该动作并用 getDisabledImageDescriptor()方法来抓取该图案。另外，当鼠标指针盘旋与之上时，若你想改变该图案，则 setHoverImageDescriptor()方法可以设定该属性。

表 4.11 Action 类的监听器方法

Action 监听器方法 功能

addPropertyChangeListener(IPropertyChangeListener) 将一属性改变监听器绑定于 Action

removePropertyChangeListener(IPropertyChangeListener) 从 Action 移除属性改变监听器

firePropertyChange(Event) 根据一事件改变一个属性

firePropertyChange(String, Object, Object) 根据新旧对象改变一个属性

setHelpListener(HelpListener) 将一个帮助监听器绑定于 Action

getHelpListener() 返回一个绑定于 Action 的帮助监听器

表 4.12 Action 类的杂项类方法

方法 描述

setID(String) 设定一个 Action 标识符

getID() 返回一个 Action 标识符

setActionDefinitionID(String) 设定一个 Action 定义的标识符

getActionDefinitionID() 返回一个 Action 定义的标识符

setMenuCreator(IMenuCreator) 为 Action 设定一个菜单生成器

getMenuCreator() 从 Action 返回一个菜单生成器

setDisabledImageDescriptor(ImageDescriptor) 设定禁用 Action 图案

getDisabledImageDescriptor() 返回禁用的 Action 图案

setHoverImageDescriptor(ImageDescriptor) 设定鼠标悬浮图案

getHoverImageDescriptor() 返回鼠标悬浮图案

有了这些方法，JFace 系列套件较之简单的 Ch4_StatusAction 类极大地拓宽了 Action 类的功能。虽然你或许并不需要它们，但是了解它们的功能以及它们在应用程序中如何发挥的功效还是相当重要的。

4.3 更新小部件窗口

为了继续发展丰富小部件窗口应用程序，本章提供了一个合成器的子类，其包含了可以接收和应激用户动作的小部件。它会同之前章节的程序代码集成。

4.3.1 构建第四章的合成器

如下呈现了 Ch4_Composite 类，该类子类化了 4.1 节中的 Ch4_MouseKey 类并执行了在 4.2 节中开发的 Ch4_Contributions 类。我们推荐你将此类加入到 com.swtjface.Ch4 程序包中去。

```
package com.swtjface.Ch4;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.events.*;

public class Ch4_Composite extends Ch4_MouseKey {

public Ch4_Composite(Composite parent) {

super(parent);

Button launch = new Button(this, SWT.PUSH);

launch.setText("Launch");

launch.setLocation(40,120);
```

```
launch.pack();

launch.addMouseListener(new MouseAdapter() {

public void mouseDown(MouseEvent e) {

Ch4_Contributions sw = new Ch4_Contributions();

sw.open();

}

});

}

}
```

Ch4_Composite 的运作简单而易于理解。通过扩展 Ch4_MouseKey 类，集成了类型化和非类型化的绑定于该合成器的 SWT 监听器。另外还加上了一个第三按钮标签。当按下它，该按钮就会就会生成一个使用了 actions 和 contributions 来执行事件处理的 JFace 窗口的实例。

4.3.2 在小部件窗口中加入 Ch4_Composite

在小部件窗口的选项夹中加入下一个 tab 其包含了本章生成的一个合成器。如下展示的是小部件窗口的主应用程序。

```
package com.swtjface.Ch2;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.jface.window.*;

import com.swtjface.Ch3.*;

import com.swtjface.Ch4.*;

public class WidgetWindow extends Window {

public WidgetWindow() {

super(null);

}
```

```
protected Control createContents(Composite parent) {

TabFolder tf = new TabFolder(parent, SWT.NONE);

TabItem chap3 = new TabItem(tf,SWT.NONE);

chap3.setText("Chapter 3");

chap3.setControl(new Ch3Comp(tf));

TabItem chap4 = new TabItem(tf,SWT.NONE);

chap4.setText("Chapter 4");

chap4.setControl(new Ch4_Composite(tf));

getShell().setText("Widget Window");

return parent;

}

public static void main(String[] args) {

WidgetWindow wwin = new WidgetWindow();

wwin.setBlockOnOpen(true);

wwin.open();

Display.getCurrent().dispose();

}

}
```

一旦更新后，小部件窗口应呈现相似如图 4.6。当按下执行按钮出现的是 Ch4_Contributions。

4.4 小结

事件处理说来简单做起来难。很明显，当用户按下按钮或是键入文本，软件应有规程对应。但是追踪那个小部件激发了事件，哪一类事件发生以及该执行哪段软件程序等过程都不是很清晰，而需要付诸努力。在某种程度上，努力的程度决定于该工具套件。如果该套件提供了尽可能多的事件处理和小部件，那样你就为此广阔范围不得不付出程序结构复杂化的代价。

这是 SWT 事件模型的真实情况。因为有如此之多的事件类型，你需要表 4.1 或 4.5 来写对应的代码。因为有如此之多的来针对事件的反应，所以一个独立的适配器成了必须。该事件处理还需要大量的理解，但当你

需要追踪鼠标右击事件以及用户是否横移一个小部件时，SWT 就是最佳的可得工具套件。

在另外一个方面，JFace 的开发人员在设计工具套件时应用 Pareto（28 原理）法则。该法则应用于 GUI 编程即是 80%的程序代码是在处理仅 20%的事件。相似地，占主体的那些事件仅是被一小部分的小部件所激发。遵循这一法则，JFace 的开发人员得出结论，即无需要监听器、适配器或是小部件。取而代之的，JFace 的事件处理是使用 actions——用户在 GUI 动作时所激发的；和 contributors——通过激发一个单一的 action 所能形成的多种形式。

很明显的，对于复杂的图形界面必须综合使用两种事件处理机制。虽然 JFace 可以提供菜单、工具条和按钮的快速编程，但是我们还需要 SWT 来处理键盘事件以及和 Shell、表等小部件相关的事件。另外，JFace 类也无法区分一个鼠标右击和左击事件。因此，对于一个 GUI 变成人员事实上是在寻找一条综合使用两个套件以最少代码获取最大功能的途径。

如同本章内的这些表格，高效的事件编程依赖于对无数规则、类和细节的明了。正因为这样的复杂性，我们对于如何提供本书材料也是苦思冥想。一开始我们计划在稍后的章节描述 SWT/JFace 的事件模型，但是所有之前的代码将是静态的。所以，为了确保将来的样例代码对于读者更加有所帮助，我们决定提前介绍这一令人费解的主题。让我们开始构建动态的 GUI！

第五章 小部件续_1

本章涵盖内容

- SWT 文本小部件
- JFace 文本小部件
- 组合框
- 工具条
- 滑块
- 进度指示器

现在你应该已经知道小部件是如何同事件和监听器联系挂钩的了。挨下来我们将继续我们的 JFace/SWT 小部件之旅。本章我们讨论的 controls 将会勾画初你的小部件工具盒并助你了解将来你在 GUI 应用程序中使用到的大部分 control。

在本章中我们将浏览两个独立的关于文本编辑的途径。首先我们将会在一定深度上讨论如何构建在 SWT 内的文本小部件。在此讨论之后我们会纵览在 JFace 中的高级文本（编辑）支持。虽然 JFace 的文本程序包提供了很多高级的功能项，但它们还是难以使用。

一旦我们完成文本编辑细节的描述，我们将对若干个常用的小部件进行示范，其中涵盖：组合框、工具条、滑块、和进度指示器，当然还有 coolbar，也即可以重组工具条让用户方便地自定义。

完成这一切之后，我们还尚不能在小部件窗口应用程序中构建一个简单的样例。由于我们所论及的小部件的

内容如此之宽泛，我们将针对每一个简单的小部件或是概念生成若干个简单的样例。这些样例的结构就像之前你看到的那些合成器一样，并能象其他例子一样插入到小部件窗口之中。

5.1 用 SWT 编辑文本

SWT 提供了你进行文本编辑所需要的两个 controls：Text 可以允许文本不带任何格式地输入；StyledText 则让你在输入文本时可以改变颜色或是风格。虽然 StyledText 和 Text 控制风格的方法非常相似，但是它们除了都是扩展的合成器外就毫无瓜葛了。

这些类提供的编辑工具都是原始未加以发展加工的。它们提供简便的方法以拷贝文本到粘贴板或是从粘贴板复制到文本，但你需要在恰当的时候编写代码调用这些方法。

5.1.1 基本文本小部件

文本 control 允许用户输入未经格式化的文本。Text 可以用其最基本的形式例示或使用；然而，它还有一些较有趣的特性可以使用。Text 会受若干个事件影响，听过侦听这些事件，你可以影响到该小部件的行为。下面例示的 Ch5_Captilaizer 可以将用户输入的文本大写化。

```
package com.swtjface.Ch5;

import org.eclipse.swt.SWT;

import org.eclipse.swt.events.VerifyEvent;

import org.eclipse.swt.events.VerifyListener;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.Composite;

import org.eclipse.swt.widgets.Text;

public class Ch5Capitalizer extends Composite {

public Ch5Capitalizer(Composite parent) {

super(parent, SWT.NONE);

buildControls();

}

private void buildControls() {

this.setLayout(new FillLayout());
```

```
Text text = new Text(this, SWT.MULTI | SWT.V_SCROLL);

text.addVerifyListener(new VerifyListener() {

public void verifyText(VerifyEvent e) {

if( e.text.startsWith("1") ) {

e.doit = false;

}

else {

e.text = e.text.toUpperCase();

}

}

});

}

}
```

由于这个样例的原因，假定在将所有文本大写化之外，我们还需要禁止输入任何以 1 打头的东西。这个样例针对这两项任务都使用了 VerifyListener 接口。当文本被修改或是新文本被插入时，任何已注册的 VerifyListener 就被调用。

首先，VerifyListener 会检查以确认新文本并非以 1 开始。如果是，则该事件的 doit 领域会被设定为 false，导致编辑行为被 Text 的 control 所拒绝。对于使用键盘的用户，该方法会随着每一次的键击而调用，从而有效地阻止任何 1 的输入，而同时允许其他字符通过。然而，当一个时间内插入了多个字符时，或是程序性地产生或是粘贴文本时，监听器只能针对整块的文本调用一次。所以，它就能阻止第一个是 1 的文本而通过其他。

在验证了哪些文本应该通过之后，我们将由分配新文本进入事件的文本领域来使其大写化。最初该领域持有插入的字符串，但就像你所看到的，我们可以将该字符串修改为任意我们想要的。为了运行该样例，需要添加如下几行代码到 createContent()方法，使其加入到小部件窗口中。

```
TabItem chap5Capitalizer = new TabItem(tf, SWT.NONE);

chap5Capitalizer.setText("Chapter 5 Capitalizer");

chap5Capitalizer.setControl(new Ch5Capitalizer(tf));
```

表 5.1 总结了控制文本实例的若干重要方法。这些方法允许你修改文本，控制其外观，并添加你所感兴趣

的事件监听器。

表 5.1 重要的 Text 方法

方法 描述

addModifyListener() Adds a listener to be notified when the text is modified

addSelectionListener() Adds a listener to be notified when this control is selected

addVerifyListener() Adds a listener to validate any changes to the text

append() Appends the given String to the current text

insert() Replaces the current contents with the given String

copy(), cut(), paste() Moves the current selection to the clipboard, or replaces the current selection with whatever currently is in the clipboard

setSelection(),

selectAll() Programmatically modifies the current selection

setEchoCharacter() Displays the character passed to this method instead of the text typed by the user (useful for hiding passwords, for example)

setEditable() Turns editing on or off

setFont() Sets the font used to display text, or uses the default if passed null (the font can only be set for the widget as a whole, not for individual sections)

对于绝大部分，这些方法都是简洁明了的。你所需关注的唯一的事情就是 insert()方法会取代小部件的全部内容——因为它并不允许在现存的文本中插入文本。

现在我们已经完成了对简单文本条目的叙述，然后我们将使用 StyledText 小部件来观察一些更有趣的可视化的选择项。

5.1.2 StyledText 小部件

虽然对于文本条目 Text 是有用的，但是当你在展示文本时可能经常会用到更多的 control。最终 SWT 提供了 StyledText 小部件。

StyledText 小部件提供了文本展示的所有方法并附加了修改显示字体、文本颜色、字体风格以及更多的能力。额外地 StyledText 提供了编辑操作如剪切、粘贴基本运作的支持。StyledText 包含了一系列的应用到该小部件的预定义的动作，这些是常规的东西如：剪切、粘贴、移动至下一个词、移动至文末。代表这些动作的常量在 org.eclipse.swt.custom 程序包中的 ST 类中有定义。这些常量在两种情况下发挥功效：首先，你可以使用它们程序性地使用 invokeAction()方法调用任一的这些方法；其次，你也可以使用

setKeyBinding()方法来将它们绑定于键击行为。setKeyBinding()选定一个键（可以通过诸如 Shift 或是 Ctrl 之类的编辑键来修改 SWT 常量之一）绑定于指定的动作。如下的例子中组合键 Ctrl-Q 绑定于粘贴动作。引起注意的是这并不意味着会将默认键的绑定清除，该两个绑定都会生效。

```
StyledText.setKeyBinding( 'Q' | SWT.CONTROL, ST.PASTE );
```

StyledText 同样也会向你所监听的事件进行广播。除了 Text 定义的外，StyledText 还加入了画行背景以及行风格。You can use them to modify the style or background color of an

entire line as it's drawn by setting the attributes on the event to match the way you wish the line to be displayed. 然而，你需要保持清醒的是，当你使用一个 LineStyleListener 时，则在 StyledText 实例中将无法再调用 get/setStyleRange()方法（具体将在下一节中讨论）。同样地，使用一个 LineBackgroundListener 意味着你将不能调用 getLineBackground()和 setLineBackground()方法。你可以通过使用 StyleRanges 来修改一个 StyledText 显示的风格。

StyledText 使用 StyleRange 类来管理当前显示的各个不同的 style。一个 StyleRange 持有一系列文本的风格属性的信息。所有的 StyleRange 领域都是公共、公开的，切可以自由修改，但是经修改的风格一直要到 StyledText 实例调用了 setStyleRange()方法后才生效。

StyleRanges 通过使用一个开始偏移量和长度来指明文本的区域。StyleRanges 追踪背景和前景颜色（或以 null 来使用默认设定）以及一个或为 SWT.NORMAL，或为 SWT.BOLD 的字体风格。StyleRange 也有一个 similarTo()方法，使用该方法你可以检查两个 StyleRanges 彼此是否相似。两个 StyleRange 所包含的前景色、背景色、字体风格属性若相同，则定义为它们是相似的。当你在合并两个相近的 StyleRanges 为一个单一实例时，这一方法相当有效。

为了说明 StyleRange 的使用，我们将奉上选自一个简单文本编辑器的程序片段，该编辑器具有持久化一个文件中的文本和风格信息能力。由于空间的限制，我们无法在此显示全部的代码，但你可以从网上下载包含该代码的程序。

我们首先考虑如何持久化 style 的信息。在我们保存该文本后，可以调用 styledText.getStyleRange()方法来得到该 style 的信息，该方法将返回代表在该文档中当前每一个 style 的一 StyleRange 数组。

因为这是一个简单的例子，我们假定唯一可能得风格是粗体文本。我们穿过数组并保存每一个 StyleRange 的起始偏移量和长度到我们的文件中。这个例子可以很容易地扩展成对每一个 StyleRange 进行查询并持久化诸如前景、背景色等附加信息。如下演示的为程序片断：

```
StyledText styledText = ...

StyleRange[] styles = styledText.getStyleRanges();

for(int i = 0; i < styles.length; i++) {

printWriter.println(styles[i].start + " " + styles[i].length);
```

```
}

style information:

StyledText styledText = ...

String styleText = ... //read line from the file

StringTokenizer tokenizer = new StringTokenizer(styleText);

int startPos = Integer.parseInt(tokenizer.nextToken());

int length = Integer.parseInt(tokenizer.nextToken());

StyleRange style = new StyleRange(startPos, length, null, null, SWT.BOLD);

styledText.setStyleRange(style);
```

一旦 styles 被保存，当文件再次打开时，它们将会被载入。我们在一个时间内阅读一行并解析每行内容以获得 style 的信息。

重复一遍，在本例中唯一可能的 style 是粗体文本，所以我们能假定每一个 sytle 的行代表着一段长度的文本应该是粗体显示并以给定的偏移量起始。我们例示了一个新的 StyleRange 并使用读自文件的偏移量和长度值，标记其为粗体，将其加入到我们的 StyledText 控制中。二者选一地，我们其实可以建立一个待用的所有 StyleRange 数组并使用 StyleRanges()方法来一次性地应用它们。

如下的方法，toggleBold()，可以当输入文本时在粗体和正常字体间转换。当 F1 键被按下时侦听该事件的 KeyListener 会调用该方法。

```
private void toggleBold() {

doBold = !doBold;

styledText = ...

if(styledText.getSelectionCount() > 0) {

Point selectionRange = styledText.getSelectionRange();

StyleRange style = new StyleRange(selectionRange.x,

selectionRange.y,

null, null,

doBold ? SWT.BOLD
```

```
: SWT.NORMAL);

styledText.setStyleRange(style);

}

}
```

当 toggleBold()方法在当前文本模式间转换后，它会检查当前是否有被选定的文本，若有，它会确认选定的文本是否符合新模式。GetSelectionRange()方法会返回一个 Point 对象，该对象的 X 值域代表这当前选定文本的起始偏移量，其 Y 值域代表着选定的文本的长度。我们应用这些值来产生一个 StyleRange，而且我们还将其应用于选定的文本。

最后，还有一个疑问就是我们该如何将文本粗体化，在此我们将再一次应用 ExtendedModifyListener：

```
public void modifyText(ExtendedModifyEvent event) {

if(doBold) {

StyleRange style = new StyleRange(event.start,

event.length,

null, null,

SWT.BOLD);

styledText.setStyleRange(style);

}

}
```

当新文本插入后，modifyText()方法被调用。如果当前是粗体模式（按 F1 键选定），我们将使用包含于该事件内的最新修改信息来产生一个新的带粗体文本属性的 StyleRange，并将其应用于本文档。调用 setStyleRange()可以将我们的新 style 应用于文档。StyledText 可以追踪相邻文本的 style，而且如有可能还可以将多个小片段整合成一个单个的大片断。

一个 StyledText 样例

我们的详细的 StyledText 样例展示了你如何能使用 StyledText 发布的事件来应用 undo/redo 功能。样例显示了一个带滚动条的文本区域，该区域内用户可以键入。按下 F1 键可以 undo 最后一次编辑，而按下 F2 键可以 redo 刚才最后一次的 undo 操作。需要注意的是，在我们的程序中剪切、复制和粘贴功能都隐含地自动提供了，在我们的平台中它已经和标准的键盘快捷操作相联系了。

ExtendedModifyListener 有别于 ModifyListener 的地方在于它还出现于 StyledText，并作为事件得部分信息。然而 ExtendedModifyListener 提供了（事件）发生的全部细节，ModifyListener 则是在修改

发生时获得消息而并无具体细节。

为了使程序代码保持简练，该样例假定了所有的编辑发生于缓存至末尾，因为在缓存的其他地方进行的文本插入都会使得 undo/redo 动作感觉很奇怪。留待读者要做的事情就是追踪实际编辑的位置和关于 style 的信息了。

```
package com.swtjface.Ch5;

import java.util.LinkedList;

import java.util.List;

import org.eclipse.swt.SWT;

import org.eclipse.swt.custom.*;

import org.eclipse.swt.events.KeyAdapter;

import org.eclipse.swt.events.KeyEvent;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.Composite;

public class Ch5Undoable extends Composite {

private static final int MAX_STACK_SIZE = 25;

private List undoStack;

private List redoStack;

private StyledText styledText;

public Ch5Undoable(Composite parent) {

super(parent, SWT.NONE);

undoStack = new LinkedList();

redoStack = new LinkedList();

buildControls();

}
```

```
private void buildControls() {
this.setLayout(new FillLayout());
styledText = new StyledText(this, SWT.MULTI | SWT.V_SCROLL);
styledText.addExtendedModifyListener(
new ExtendedModifyListener() {
public void modifyText(ExtendedModifyEvent event) {
String currText = styledText.getText();
String newText = currText.substring(event.start,
event.start + event.length);
if( newText != null && newText.length() > 0 ) {
if( undoStack.size() == MAX_STACK_SIZE ) {
undoStack.remove( undoStack.size() - 1 );
}
undoStack.add(0, newText);
}
}
});
styledText.addKeyListener(new KeyAdapter() {
public void keyPressed(KeyEvent e) {
switch(e.keyCode) {
case SWT.F1:
undo(); break;
case SWT.F2:
redo(); break;
```

```
default: //ignore everything else
}
}
});
}
private void undo() {
if( undoStack.size() > 0 ) {
String lastEdit = (String)undoStack.remove(0);
int editLength = lastEdit.length();
String currText = styledText.getText();
int startReplaceIndex = currText.length() - editLength;
styledText.replaceTextRange(startReplaceIndex, editLength, "");
redoStack.add(0, lastEdit);
}
}
private void redo() {
if( redoStack.size() > 0 ) {
String text = (String)redoStack.remove(0);
moveCursorToEnd();
styledText.append(text);
moveCursorToEnd();
}
}
private void moveCursorToEnd() {
```

```
styledText.setCaretOffset(styledText.getText().length());

}

}
```

这是本例的关键部分：一个 ExtendedModifyListener 加入到 StyledText 对象这样我们可以对编辑事件进行追踪。当文本被编辑时，每一次都会调用 ExtendedModifyListener。经历的这些事件包含有新插入文本的信息。在本例中，我们我们使用开始偏移量以及长度从 StyledText 中获取新的文本并且保存它预防用户之后需要 undo 其编辑。如有可能，事件还在其 replacedText 域内提供了被替换的文本的信息。还有一个更为健壮的应用就是如果发生编辑被 undo 则可以将新编辑信息和该文本信息一起保存以备再次插入。

一个侦听键击的 KeyListener 使用 KeyEvent 来汇报。我们使用 KeyCode 域来检查它是否和我们所关注的这些键之一相吻合。这些键的常量都已在 SWT 类里面有了定义。另外，我们可以查询诸如 Ctrl、Alt 的状态，具体是通过遮盖事件的 stateMask 域来针对由 SWT 类预先定义的常量。

Undo 会弹出在 undo 信息堆栈的顶端条目，该条目持有所有已发生的编辑中的一条记录信息。然后，我们使用方法 replaceTextRange()以空字符串来替换缓存中最后 n 个字符，而 n 则是我们从堆栈中获取的编辑的长度值。

要 redo 一次编辑，我们弹出 redo 堆栈的顶端条目。然后它会以 append()方法插入到文档末尾。

如下几行代码加入到小部件窗口程序中可以让你测试该可 undo 编辑器：

```
TabItem chap5Undo = new TabItem(tf, SWT.NONE);

chap5Undo.setText("Chapter 5 Undoable");

chap5Undo.setControl(new Ch5Undoable(tf));
```

在 SWT 中的文本编辑有一些复杂，但是当你一旦理解了由小部件广播的事件后，在你应用程序中加入基本类型的确认或控制逻辑就并不困难了。然而某些特性若要使用 SWT 提供的工具开实施还是有难度的。作为 JFace 文本编辑，虽说是更为复杂，当然也更强大，它可以提供一大群的功能选项，这一些在下节中我们会有讨论。

5.2 JFace 文本支持

作为使用由 SWT 提供的 StyledText 控制的另一个替换方案，JFace 提供了一个更为广阔的文本编辑框架。超过 300 个类和接口程序分布于 7 个 jface.text 程序包和子包之中。鉴于我们的空间有限我们不可能蜻蜓点水般地对全部内容作粗浅介绍，我们将对在 org.eclipse.text 程序包中的关键类作概要介绍，并开发一个小型程序来展示一些高级特性。

5.2.1 获取 JFace 文本程序包

在你能使用 JFace 文本程序包之前，你需要从你的 eclipse 安装程序中解压获得一些 jar 文件：位于 $ECLIPSE_HOME/plugins/org.eclipse.text_x.y.z 的 text.jar，以及 jfacetext.jar 位于

$ECLIPSE_HOME/plugins/org.eclipse.jface.text_x.y.z。另外在你测试本节提供的样例程序前，请确认这两个文件在你的类路径中。

5.2.2 文本阅读器和文档

Jface 的文本支持是由一系列的核心类以及一系列添加高级特性的扩展为参数组成。我们首先会讨论核心部分，然后提供关于可得的扩展的概要介绍。

两个接口形成了 JFace 的文本支持的核心：IDocument 和 ITextViewer。每一个都由 JFace 提供默认的执行。

一个 IDocument 的实例持有被编辑的真实的文本信息。IDocument 的主要应用在于类 Document，AbstractDocument 提供了一个部分应用，且你也可一自行按需扩展。在设定和获取文本的标准方法之外，IDocument 也借由 IDocumentListener 接口允许监听器来接收编辑内容的通知。

Idocument 也支持如下几个高级特性：

■ 位置——你可以分派一个“粘附”标记作为一段文本区域的一个坐标位置。当它被分派到文本时该坐标对象就包含有一个偏移量和文本长度值。当文档的文本被更新，该坐标值就会同文本保持同步，以确保其总是指向相同的文本章节位置而不管该章节再文档中如何移动。你可以将此应用于诸如“书签”等特性，这样就可以允许用户跳跃到一个文档中标记的位置。基本的 Position 类对于诸如偏移量和文本长度之类跟踪信息提供有限，所以通常你需要将其子类化使其在你的应用程序中构建有效的行为。

■ 分区内容类型——理论上讲，一个文档是由一个或多个分区组成，而分区又由 ITypedRegion 接口代表。每一个分区可以有不同的内容类型，诸如平格式文、富格式文或超文本。要使用这一特性，你需要生成一个 IDocumentPartitioner 并将其分配给你的文档。然后该文档的 partitioner 就负责对有关文档中指定位置的内容类型查询作出反应，而且它必须使用 computePartitioning()方法来返回一个文档提供的所有 ITypeRegions 的数组。当然，在你的文档中应用 partitioner 并不是必须的，如果你没有作分派，整个文档就会用类型 IDocument.DEFAULT_CONTENT_TYPE 来将其视作一个单一的区域。

■ 搜索——IDocument 通过使用 search()方法为用户提供了搜索功能。虽然她不能支持正则表达式或其他搜索方式，但它可以在一下方面给予你控制：搜索起始位置，搜索方向，大小写敏感以及是否整个词语匹配。ItextViewer 则倾向于将一个标准的文本小部件转变为一个基于文档的文本小部件。默认的应用是 TextViewer，该阅读器使用一个显示数据遮盖下的 StyledText。ITextViewer 支持对应于文本修改和可视化事件的监听器，例如改变当前文本的可视区域（观察点）。虽然作为 ITextViewer 的默认应用，TextViewer 允许如你所愿地修改显示来接近 StyledText，但作为手册建议你使用 TextPresentation，因为它可以收集该文档中带有的各个不同的 StyleRanges。

ITextViewer 也支持大量的不同类型的插件，那些插件可以用以修改小部件的行为。这些功能的定制包括有：包括通过 IUndoManager 的 undo 的支持，通过 ITextDoubeClickStrategy 的怎样处理鼠标双击，通过 IAutoIndentStrategy 的自动缩进，以及通过 ITextHover 的鼠标停留于文本某章节断时文本如何显示（的策略）。你可以通过分配一个文本阅读器接口的合适的实例以及调用 activatePlugins()方法来使用这些插件。

最终，程序包 org.eclipse.jface.text 的一系列不同的子程序包提供了有用的扩展，它们总结如表 5.2

Table 5.2 org.eclipse.jface.text 程序子包提供一系列高级功能

程序包	描述
org.eclipse.jface.text.contentassist	提供一个当文本被键入时的自动完成的框架，例如在很多 Java 集成开发环 IContentAssistant 和 IcontentAssistantProcessor 协同工作在恰当的 IcompletionProposals。
org.eclipse.jface.text.formatter	提供格式化文本的功能。 IcontentFormatter 注册不同内容类型的 IformattingStrategy 的实例。当文本需要作格式化，则可以给出代表需要 文本一个字符串来代表何时的格式化策略。
org.eclipse.jface.text.presentation	用以当发生改变时更新文档的视觉外观。在改变后，一个 IPresentation 被用以计算需要重新刷屏的文本区域，然后该信息将同一个 TextPresent 送达给一个 IpresentationRepairer 来重置被破坏的区域的 style。
org.eclipse.jface.text.reconciler	用来把一个文档和其文本的外部存储保持同步。默认的 Reconciler 在后台 运行，当它发现一个“脏”区域需要保持同步时会和 IreconcilingStrate 进行沟通。
org.eclipse.jface.text.rules	根据可定制的 IRules 来定义类实现扫描和匹配文本。该框架用以实现 pre 程序包以及文档 partitioner 并包括由通常情况的匹配规则，诸如单词、数 以及行末。
org.eclipse.jface.text.source	用来向文本添加可视化标记，诸如在 Eclipse 中红色的 X 表示编译错误。要 特性，你必须使用 ISource-Viewer 而不是 ItextViewer。你将需要 子类化注解来勾画恰当的图案。

5.2.3 一个 JFace 样例

我们现在将要构建一个简单的文本编辑器，该编辑器使用了一些 TextViewer 的特性。受 OpenOffice 以及其他一些 word 处理程序的特性启发，该编辑器会对用户键入的单个词进行跟踪。在任何时候，用户可以按下 F1 健来获取关于如何完整其正在键入的内容的建议列表，开始以当前键入的文本后续以用户所有曾经键入过的词语。

为了实现这一功能，我们将使用 org.eclipse.jface.text.contentassist 中的类。我们曾经一建立过一个功能类称之为 WordTracker，该类负责追踪用户最近键入过的绝大部分词语并能建议如何完成该字符串。RecentWordContentAssistProcessor 是 IcontentAssistProcessor 的一个实例，可以提供该框架完整的可能方案。最终， CompletionTextEditor 成为我们的主类：它负责设置 TextViewer 并添附合适的监听器。我们将会详细探讨每一个这些类，并随以完整的源代码。

一个 ContentAssistant 负责向用户建议完整词语的可能方案。每一个 ContentAssistant 有一个或多个注册的 IContentAssistantProcessor，且又和不同的内容类型相联系。当一个 TextViewer 要求建议时，ContentAssistant 会授权与当前文档区域的内容类型相适应的协处理器。

你可能会经常使用到 ContentAssistant，然而我们需要定义一个 IContentAssistProcessor。当方法 computeCompletionProposals()被调用，该处理器主要负责提供可能的完整方案的数组。我们的应用相当简洁明了：给出当前光标的偏移量到文档中，它会查找第一次空白区域出现的地方并决定当前词语的片断，

如果有，则在文档中逐个字符地向后移动：

```
while( currOffset > 0

&& !Character.isWhitespace(

currChar = document.getChar(currOffset)) )

{

currWord = currChar + currWord;

currOffset--;

}
```

一旦有了当前词，它会要求获得来自词语跟踪器中的完整策略然后使用那些完整策略来例示在 buildProposals()方法中 ICompletionProposal 的一个数组：

```
int index = 0;

for(Iterator i = suggestions.iterator(); i.hasNext();)

{

String currSuggestion = (String)i.next();

proposals[index] = new CompletionProposal(

currSuggestion,

offset,

replacedWord.length(),

currSuggestion.length());

index++;

}
```

每一个提议都包含有建议性的文本，在哪个位置插入文本的偏移量，替换的字符数目，后面光标的位置。ContentAssistant 会使用这一数组来显示给予用户的选择项，用户一旦选择了其中一个就会插入正确的文本。在本例中，我们总是通过一个键击事件监听器来程序性地激活 ContentAssistant。然而，IContentAssistProcessor 也包含有允许你指定一系列字符作为建议方案自动触发显示的若干方法。你可以应用 getCompletionProposalAutoActivationCharacters()方法来返回你预定的作为触发器的若干字

符。如下是 IContentAssistProcessor 完整应用的样例。

```
package com.swtjface.Ch5;

import java.util.Iterator;

import java.util.List;

import org.eclipse.jface.text.*;

import org.eclipse.jface.text.contentassist.*;

public class RecentWordContentAssistProcessor implements

IcontentAssistProcessor {

private String lastError = null;

private IContextInformationValidator contextInfoValidator;

private WordTracker wordTracker;

public RecentWordContentAssistProcessor(WordTracker tracker) {

super();

contextInfoValidator = new ContextInformationValidator(this);

wordTracker = tracker;

}

public ICompletionProposal[] computeCompletionProposals(

ITextViewer textViewer, int documentOffset) {

IDocument document = textViewer.getDocument();

int currOffset = documentOffset - 1;

try {

String currWord = "";

char currChar;

while( currOffset > 0 && !Character.isWhitespace(
```

```
currChar = document.getChar(currOffset)) ) {
currWord = currChar + currWord;
currOffset--;
}
List suggestions = wordTracker.suggest(currWord);
ICompletionProposal[] proposals = null;
if(suggestions.size() > 0) {
proposals = buildProposals(suggestions, currWord,
documentOffset - currWord.length());
lastError = null;
}
return proposals;
}
catch (BadLocationException e) {
e.printStackTrace();
lastError = e.getMessage();
return null;
}
}
private ICompletionProposal[] buildProposals(List suggestions,
String replacedWord,
int offset) {
ICompletionProposal[] proposals =
new ICompletionProposal[suggestions.size()];
```

```
int index = 0;
for(Iterator i = suggestions.iterator(); i.hasNext();) {
String currSuggestion = (String)i.next();
proposals[index] = new CompletionProposal(
currSuggestion,
offset,
replacedWord.length(),
currSuggestion.length());
index++;
}
return proposals;
}
public IContextInformation[] computeContextInformation(
ITextViewer textViewer, int documentOffset) {
lastError = "No Context Information available";
return null;
}
public char[] getCompletionProposalAutoActivationCharacters() {
//we always wait for the user to explicitly trigger completion
return null;
}
public char[] getContextInformationAutoActivationCharacters() {
//we have no context information
return null;
```

```
}

public String getErrorMessage() {

return lastError;

}

public IcontextInformationValidator getContextInformationValidator() {

return contextInfoValidator;

}

}
```

我们在文档中逐个字符向后移动，直到遇到文档开始的空白处。每一个建议都包含有类似于诸如在文档何处插入的信息等。理论上你可以在你想要的地方插入该建议性文本，然而如果文本并没有插入在当前的光标位置上则容易引起用户的混淆。

WordTracker 是一个功能类，用以维持和搜索单词列表（见样例 5.4）。我们的应用并不特别有效率，但是它足够简单，而且速度快捷可以满足我们所需。每个单词都加入到列表中去，当需要给出建议方案时，列表就会来回寻找凡以给定字符串开头的的项目。WordTracker 并不包含任何的 SWT 和 JFace 代码，所以我们无需细节性地展开讨论。

```
package com.swtjface.Ch5;

import java.util.*;

public class WordTracker {

private int maxQueueSize;

private List wordBuffer;

private Map knownWords = new HashMap();

public WordTracker(int queueSize) {

maxQueueSize = queueSize;

wordBuffer = new LinkedList();

}

public int getWordCount() {
```

```
return wordBuffer.size();
}
public void add(String word) {
if( wordIsNotKnown(word) ) {
flushOldestWord();
insertNewWord(word);
}
}
private void insertNewWord(String word) {
wordBuffer.add(0, word);
knownWords.put(word, word);
}
private void flushOldestWord() {
if( wordBuffer.size() == maxQueueSize ) {
String removedWord =
(String)wordBuffer.remove(maxQueueSize - 1);
knownWords.remove(removedWord);
}
}
private boolean wordIsNotKnown(String word) {
return knownWords.get(word) == null;
}
public List suggest(String word) {
List suggestions = new LinkedList();
```

```
for( Iterator i = wordBuffer.iterator(); i.hasNext(); ) {

String currWord = (String)i.next();

if( currWord.startsWith(word) ) {

suggestions.add(currWord);

}

}

return suggestions;

}

}
```

Ch5CompletionEditor 包含有我们所讨论的这些部件。在方法 buildControls()中，我们例示并设置了一个 TextViewer。生成一个 ConterntAssistant 并把我们定制的处理器分配于默认的内容类型：

```
final ContentAssistant assistant = new ContentAssistant();

assistant.setContentAssistProcessor(

new RecentWordContentAssistProcessor(wordTracker),

IDocument.DEFAULT_CONTENT_TYPE);

assistant.install(textViewer);
```

一旦该 assitant 被设置完毕，它会安装于阅读器。注意到该 assistant 是把自己安装与阅读器而非如你所预期的那样由阅读器去接受一个 ContentAssistant。

为了对所编辑的内容引起关注，我们使用一个 ITextListener，它同 ExtendedModifyListener 相似并归 StyledText 所使用。

```
textViewer.addTextListener(new ITextListener() {

public void textChanged(TextEvent e) {
```

```
if(isWhitespaceString(e.getText())) {
wordTracker.add(findMostRecentWord(e.getOffset() - 1));
}
}
});
```

在这儿我们使用了和 StyledText 相同的监听器来侦听键击事件。当我们发现触发健被敲击，我们就能程序性的调用 content assistant：

```
case SWT.F1:
assistant.showPossibleCompletions();
```

ContentAssistant 从该点起完成所有的工作，展示可能的完整方案和插入选定内容到文档，如样例 5.5：

```
package com.swtjface.Ch5;
import java.util.StringTokenizer;
import org.eclipse.jface.text.*;
import org.eclipse.jface.text.contentassist.ContentAssistant;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.KeyAdapter;
import org.eclipse.swt.events.KeyEvent;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
public class Ch5CompletionEditor extends Composite {
private TextViewer textViewer;
private WordTracker wordTracker;
private static final int MAX_QUEUE_SIZE = 200;
```

```
public Ch5CompletionEditor(Composite parent) {
super(parent, SWT.NULL);
wordTracker = new WordTracker(MAX_QUEUE_SIZE);
buildControls();
}
private void buildControls() {
setLayout(new FillLayout());
textViewer = new TextViewer(this, SWT.MULTI | SWT.V_SCROLL);
textViewer.setDocument(new Document());
final ContentAssistant assistant = new ContentAssistant();
assistant.setContentAssistProcessor(
new RecentWordContentAssistProcessor(wordTracker),
IDocument.DEFAULT_CONTENT_TYPE);
assistant.install(textViewer);
textViewer.getControl().addKeyListener(new KeyAdapter() {
public void keyPressed(KeyEvent e) {
switch(e.keyCode) {
case SWT.F1:
assistant.showPossibleCompletions();
break;
default:
//ignore everything else
}
}
```

```
});
textViewer.addTextListener(new ITextListener() {
public void textChanged(TextEvent e) {
if(isWhitespaceString(e.getText())) {
wordTracker.add(findMostRecentWord(
e.getOffset() - 1));
}
}
});
}
protected String findMostRecentWord(int startSearchOffset) {
int currOffset = startSearchOffset;
char currChar;
String word = "";
try {
while(currOffset > 0 && !Character.isWhitespace(
currChar = textViewer.getDocument().getChar(currOffset))) {
word = currChar + word;
currOffset--;
}
return word;
}
catch (BadLocationException e) {
e.printStackTrace();
```

```
return null;

}

}

protected boolean isWhitespaceString(String string) {

StringTokenizer tokenizer = new StringTokenizer(string);

//if at least 1token, this string is not whitespace

return !tokenizer.hasMoreTokens();

}

}
```

每个 TextViewer 需要一个 IDocument 来存储其文本。在这儿我们使用默认的 Document 类，该类足够满足我们大部分之需。你必须在使用 TextViewer 之前将文本设之上，否则就会产生 NullPointerExceptions。每一个 ContentAssistant 会被分配系列的 IContentAssistProcessor；然后会有文档的内容类型选择一个合适的。在此我们分派我们的 processor 到默认的内容类型，该内容类型由 IDocument 接口定义。

当我们检测到正确的键被按下，我们会程序性地调用 ContentAssistant。我们检查每一个编辑动作。当我们发现一个编辑仅包含有空白时，我们假定一个新词被加入了，然后从文档中获取它并存入我们的 WordTracker。

在此我们从当前编辑位置开始向后逐个字符地遍历文档。当我们发现空白处，我们为 WordTracker 抓取该单词。要观察这一动作，需要在小部件窗口内加入如下代码：

```
TabItem chap5Completion = new TabItem(tf, SWT.NONE);

chap5Completion.setText("Chapter 5 Completion Editor");

chap5Completion.setControl(new Ch5CompletionEditor(tf));
```

如同你所看到的那样，SWT 和 JFace 提供了一个广泛系列的文本编辑功能选项。虽然我们仅接触到 JFace 提供的一小部分，但通过对这一设计整体的理解，你应该能不是太困难地使用这一扩展功能。现在我们将继续几个不太复杂的小部件，即如下的组合框。

第五章 小部件续_3

5.3 组合框小部件

组合框 control 用以产生一个组合盒。典型的，一个组合 control 可以让用户从待选列表中选取一个选项。有如下三类组合 control：

■ 简单式——包含有一个在顶端的可编辑的文本域以及一个底部的列表盒。这是组合的默认风格

■ 下拉式——一个可编辑的文本域并在右侧有箭头指示。点击该箭头会展现一个选择项列表由用户选择其中之一。

■ 只读式——一个下拉式的组合框但是文本域不可编辑。该风格适用于你想限定用户输入。只读组合框默认的是空选择，所以绝大部分时间你需要调用 select(0)方法使得改组合框的第一选择项可用。

这些风格都是在构造器中通过 STYLE.*来设定的，它们又是相互排斥的。图 5.1 显示了组合的可得风格；你可以使用样例 5.6 中的代码来生成这些结果。

图 5.1

从左至右组合的风格：简单式、下拉式和只读式。

```
package com.swtjface.Ch5;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.RowLayout;

import org.eclipse.swt.widgets.Combo;

import org.eclipse.swt.widgets.Composite;

public class Ch5ComboComposite extends Composite {

public Ch5ComboComposite(Composite parent) {

super(parent, SWT.NONE);

buildControls();

}

protected void buildControls() {

setLayout(new RowLayout());
```

```
int[] comboStyles = { SWT.SIMPLE,SWT.DROP_DOWN,SWT.READ_ONLY };

for (int idxComboStyle = 0;idxComboStyle < comboStyles.length;

++idxComboStyle) {

Combo combo = new Combo(this,comboStyles[idxComboStyle]);

combo.add("Option #1");

combo.add("Option #2");

combo.add("Option #3");

}

}

}
```

可以通过如下代码加入到小部件窗口来运行该样例：

```
TabItem chap5Combos = new TabItem(tf, SWT.NONE);

chap5Combos.setText("Chapter 5 Combos");

chap5Combos.setControl(new Ch5ComboComposite(tf));
```

第五章 小部件续_4

5.4 工具条管理器

工具条管理器是 JFace 的一个类，该类通过使用我们第四章中讨论的动作框架来简化工具条的构造。它是工具条的相当于 MenuManager 的等价物，而且结构相似。该类也是源自于 ConttributionManager 类。因为如此，无论是应用 IAction 还是 IContribution 接口的对象都可以加载到 ToolBarManager 上。如果有需要 ToolBarManager 将会生成合适的 SWT Controls，所以你不比介入到麻烦琐碎的细节中去。绝大部分时间你都会把 Action 对象加到 ToolBarManager 上，然后 ToolBarManager 会自动生成 Toolbar 和 ToolItem 的实例，这一点我们将在后面展开讨论。你可以通过调用应用程序窗口的 createToolBarManager()方法在你的应用程序中轻易地生成一个工具条。不同于 MenuManager 的是 createToolBarManager()方法需要一个风格参数。该风格参数决定了在工具条上的按钮的风格：平板式的或是通常的按钮式。我们在后面也会谈到，使用同一个动作来在菜单和工具条上生成项目较为便捷，例如打开文件这样常见的动作。通过重用动作，你不仅可以简化代码，而且也可确保菜单和工具条总是能保持同步。

5.4.1 ControlContribution

除了和 MenuManager 协同工作的 ContributionItems 之外，还有一个新的 ContributionItem 仅适用于 ToolBarManager：ControlContribution。这是一个很酷的类，它包含了任意的 Control 并允许使用于工具条。你甚至可以封包一个合成器并把它法制到工具条上。要使用 ControlContribution 类，你必须扩展你自己的类并应用一个名为 createControl()的抽象方法。如下的代码片断示范类一个该类的简单应用。我们生成一个定制的 ControlContribution 类，该类可以被 JFace 的 ToolBarManager 所使用：

```
toolBarManager.add(new ControlContribution("Custom") {

protected Control createControl(Composite parent) {

SashForm sf = new SashForm(parent, SWT.NONE);

Button b1 = new Button(sf, SWT.PUSH);

b1.setText("Hello");

Button b2 = new Button(sf, SWT.PUSH);

b2.setText("World");

b2.addSelectionListener(new SelectionAdapter() {

public void widgetSelected(SelectionEvent e) {

System.out.println("Selected:" + e);

}

});

return sf;

}

});
```

注意到如果你想要应对可能发生的任何事情，你必须要在你的 controls 中应用 SelectionListener。对于所有的目的和动机，ControlContribution 类都能允许你把你想要的任何东西放到工具条上。

5.4.2 手工生成工具条

虽然使用一个 ToolBarManager 来产生一个工具条相对简单，但有时你或许偶尔也要手工生成它，特别是当工具条并不复杂或是你的开发环境中没有使用 JFace。

在本例中，你将需要使用两个类：ToolBar 和 ToolItem。ToolBar 是一个持有多个 ToolItem 的合成器。

ToolBar 被渲染成由小型图标化按钮的组成的条带，典型的是 16×16 的点阵图案。每一个这些按钮都同一个 ToolItem 对应，关于这一点我们会在下章节中讨论。

通过按下按钮，用户会触发一个代表该 ToolItem 的动作。一个工具条可以是水平排列也可以是垂直排列，默认情况下是水平的。另外，ToolItem 环绕包含并形成额外的行也是可能的。

典型的，ToolBar 被用以组织和代表一系列关联的动作。例如，会有一个工具条代表所有的文本操作，例如：行排列、类型、字符大小等。

ToolItem 代表了工具条的一个单一项目。它在工具条中所代表的角色和 MenuItem 在菜单中的角色相似。但是区别在于，ToolItem 本质上是图标而非文本。正因如此，总是需要一个图案分配给一个 ToolItem。ToolItem 会忽略文本标签而且如果未被分配一个图案则只显示一个小的红方块。当用户从菜单选择乐意个 ToolItem，它会将该事件向所有已注册的 SelectionListener 广播。而你的应用程序应该对每一个 ToolItem 注册一个监听器并且使用这些监听器来任何于菜单 item 逻辑相关的动作。

第五章 小部件续_5

5.5 酷工具条 CoolBar

酷工具条类似于带有更新功能的 ToolBar。它们之间的主要区别在于一个酷工具条上的 item 可以在程序运行期间重新定位或是改变尺寸大小。每一个这样的 item 由 CoolItem 代表，CoolItem 包含由任意类型的 control。酷工具条最多用的就是持有工具条或是按钮。

接下来的程序片段展现了一个酷工具条的生成和其如何持有多个工具条。每一个子工具条都包含有功能相近组合的 item。在本例中，我们有一个工具条带有文件功能，另一个工具条带有格式功能，第三个工具条带有搜索功能，它们都被包含于一个酷工具条的 control 内。该程序样例是酷工具条的典型范例，有许多方法可以排列合组织 control 来生成激动人心的用户接口。

图 5.2 展示了工具条在四处移动前的样子，特别留意到开始文件和搜索 item 是彼此相邻的。

如下代码可以生成酷工具条：

```
String[] coolItemTypes = {"File", "Formatting", "Search"};

CoolBar coolBar = new CoolBar(parent, SWT.NONE);

for(int i = 0; i < coolItemTypes.length; i++) {

CoolItem item = new CoolItem(coolBar, SWT.NONE);

ToolBar tb = new ToolBar(coolBar, SWT.FLAT);
```

```
for(int j = 0; j < 3; j++) {

ToolItem ti = new ToolItem(tb, SWT.NONE);

ti.setText(coolItemTypes[i] + " Item #" + j);

}

}
```

注意到每一个 CoolItem 在其左侧都有一个句柄：双击该句柄则可将 CoolItem 扩展至 kugjt 的最大宽度，若有需要可把其他的 CoolItem 缩至至小。通过单击并拖曳它，用户可以移动 CoolItem 到酷工具条的不同位置。要生成额外的行，则把一个 CoolItem 由当前 CoolBar 拉下即可。要重新排列一个 CoolItem，则把它拖曳到新位置而其它的 Coolitem 会挪开腾出一个空档来安置它。

图 5.3 展示了我们重新定位 CoolItem 后的样例

第五章 小部件续_6

5.6 滑块

你再窗口中所看到的滑块和滚动条较为相似。虽然看上去股东条是滑块的应用，但它们是有区别的。滚动条绑定于它们所要滚动的项目，且不能在此环境之外使用。正因为此才产生了滑块。

你可以将滑块用作为在一个整数范围内选择任意值的一个 control。该范围的设定是通过 setMinimum() 和 setMaximum()方法。你可以点击并拖曳矩形的滑块（被称之为拇指）。你可以用 setThumb()方法设定拇指的大小；它应当是一个整数。

视觉上，拇指的尺寸是由整个范围的一定百分比例实际确定的。这样，如果范围是从 0 到 100 而拇指的尺寸是 10，则拇指会占据滑块的 10%比例。

注意到：某些操作系统有其原生的滚动条并配有一个常量尺寸的拇指。在这些平台上，拇指的尺寸因为视觉原因而忽略但是使用了其他的计算方式。

在两端箭头间拇指的移动是由一套作为增量的数据确定。你可以通过 setIncrement()方法来设定该增量。另外若你在介乎于拇指和端点箭头间位置点击鼠标则会引起鼠标的大幅移动。该移动的幅度被指为页增量并由 setPageIncrement()方法设定。图 5.4 展示了一个典型的滑块；注意到它和一个垂直的滚动条较为相似。另外还有一个较为方便的方法是调用 setValues()，该方法可以一次性地完成所有这些值的确定。该方法的特征如下：

```
void setValues( int selection, int minimum, int maximum, int thumb,

int increment, int pageIncrement)
```

selection 是拇指的开始位置。它也是有一个在滑块的范围以内的一个整数值指定的。如下的样例说明了一

个范围在 400 到 1600 之间的滑块：

```
package com.swtjface.Ch5;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.Composite;

import org.eclipse.swt.widgets.Slider;

public class Ch5Slider extends Composite {

public Ch5Slider(Composite parent) {

super(parent, SWT.NONE);

setLayout(new FillLayout());

Slider slider = new Slider(this, SWT.HORIZONTAL);

slider.setValues(1000, 400, 1600, 200, 10, 100);

}

}
```

以上代码设定了 selection 为 1000，最小是 400，最大为 1600，拇指的大小尺寸是 200，递增量为 10，页增量为 100。滑块页应用了一个风格让你来指定是否它是垂直的还是水平的。默认情况下是构造一个水平的滑块。如下代码可以将一个滑块样例添加到小部件窗口应用程序中：

```
TabItem chap5Slider = new TabItem(tf, SWT.NONE);

chap5Slider.setText("Chapter 5 Slider");

chap5Slider.setControl(new Ch5Slider(tf));
```

第五章 小部件续_6

5.6 滑块

你再窗口中所看到的滑块和滚动条较为相似。虽然看上去股东条是滑块的应用，但它们是有区别的。滚动条

绑定于它们所要滚动的项目，且不能在此环境之外使用。正因为此才产生了滑块。

你可以将滑块用作为在一个整数范围内选择任意值的一个 control。该范围的设定是通过 setMinimum() 和 setMaximum()方法。你可以点击并拖曳矩形的滑块（被称之为拇指）。你可以用 setThumb()方法设定拇指的大小；它应当是一个整数。

视觉上，拇指的尺寸是由整个范围的一定百分比例实际确定的。这样，如果范围是从 0 到 100 而拇指的尺寸是 10，则拇指会占据滑块的 10%比例。

注意到：某些操作系统有其原生的滚动条并配有一个常量尺寸的拇指。在这些平台上，拇指的尺寸因为视觉原因而忽略但是使用了其他的计算方式。

在两端箭头间拇指的移动是由一套作为增量的数据确定。你可以通过 setIncrement()方法来设定该增量。另外若你在介乎于拇指和端点箭头间位置点击鼠标则会引起鼠标的大幅移动。该移动的幅度被指为页增量并由 setPageIncrement()方法设定。图 5.4 展示了一个典型的滑块；注意到它和一个垂直的滚动条较为相似。另外还有一个较为方便的方法是调用 setValues()，该方法可以一次性地完成所有这些值的确定。该方法的特征如下：

```
void setValues( int selection, int minimum, int maximum, int thumb,

int increment, int pageIncrement)
```

selection 是拇指的开始位置。它也是有一个在滑块的范围以内的一个整数值指定的。如下的样例说明了一个范围在 400 到 1600 之间的滑块：

```
package com.swtjface.Ch5;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.Composite;

import org.eclipse.swt.widgets.Slider;

public class Ch5Slider extends Composite {

public Ch5Slider(Composite parent) {

super(parent, SWT.NONE);

setLayout(new FillLayout());

Slider slider = new Slider(this, SWT.HORIZONTAL);

slider.setValues(1000, 400, 1600, 200, 10, 100);
```

```
}

}
```

以上代码设定了 selection 为 1000，最小是 400，最大为 1600，拇指的大小尺寸是 200，递增量为 10，页增量为 100。滑块页应用了一个风格让你来指定是否它是垂直的还是水平的。默认情况下是构造一个水平的滑块。如下代码可以将一个滑块样例添加到小部件窗口应用程序中：

```
TabItem chap5Slider = new TabItem(tf, SWT.NONE);

chap5Slider.setText("Chapter 5 Slider");

chap5Slider.setControl(new Ch5Slider(tf));
```

第五章 小部件续_7

5.7 进度条

进度条可以让你把一个进度转化为一个长度性的操作。它是一个简化的计数器，进度指示器，并在多种场合下推荐使用。偶尔的，你有可能会需要较一个进度指示器更为复杂的 control，如果你决定了你直接需要一个使用一个进度条，那么你就需要负责该条的外观显示。如下为样例的程序片断：

```
//Style can be SMOOTH, HORIZONTAL, or VERTICAL

ProgressBar bar = new ProgressBar(parent, SWT.SMOOTH);

bar.setBounds(10, 10, 200, 32);

bar.setMaximum(100);

...

for(int i = 0; i < 10; i++) {

//Take care to only update the display from its own thread

Display.getCurrent().asyncExec(new Runnable() {

public void run() {

//Update how much of the bar should be filled in

bar.setSelection((int)(bar.getMaximum() * (i+1) / 10));
```

```
}

});

}
```

当你检查这一串代码时，注意到要额外地计算数值来更新进度条，对于方法 setSelection()的调用可致使小部件每次都得以更新。该行为方式不同于进度指示器或是进度监控对话框，后两者仅在对于最终用户而言的数量变化时才会更新视觉外观。

象你看到的那样，进度条的使用相对于其他我们讨论过的小部件还需要更多的工作，而且通常我们不推荐使用进度条，除非你别无选择。然而，一个进度条偶尔有的时候也是必需的，例如当你需要凹陷一个工具条，则对于那些高等级的 control 就无计可施了。

5.8 进度指示器

进度指示器小部件允许你展示一个进度条而无需担心如何来满足它。类似于进度监控对话框，它支持抽象单元工作——你只需用你预期的工作总数量来初始化该进度指示器以及它何时完成就可：

```
ProgressIndicator indicator = new ProgressIndicator(parent);

...

indicator.beginTask(10);

...

Display.getCurrent()display.asyncExec(new Runnable() {

public void run() {

//Inform the indicator that some amount of work has been done

indicator.worked(1);

}

});
```

象在本例中显示的那样，使用进度指示器需要执行两个步骤。首先你需要通过调用 beginTask()方法来让指示器指导你打算要做的工作量有多大。仅有当该方法被调用之后 control 才会显示。然后你在每次部分工作完成时就调用 worked()方法。就像我们在第四章讨论的那样，在此我们需要关注以下几个线程。在 UI 上真实工作的线程会锁定显示并阻止在初识位置使用进度指示器。然而，你并不允许从一个非 UI 线程来更新小部件。在此的解决方案就是使用 asyncExec()方法来确定更新小部件的代码进度使之从一个 UI 线程来

运行。

进度指示器还提供一个活动模式，该模式下工作总量是未知的。此事工具条不断充满、清空，直至调用了done()方法才结束。要使用活动模式，需调用beginAnimatedTask()方法，而不是beginTask()；并且此事无需使用worked()方法。假定你的工作在一个非UI线程内正确完成了，这也意味着你也无需担心调用asyncExec()方法了。

5.9 小结

SWT和JFace提供了诸多的文本编辑功能选项。SWT的易于使用，但是在这些简单文本编辑之上在扩展复杂应用将会很快编程是一桩痛苦的事情。而在lingyifm，JFace的可对复杂的文本编辑器提供足够强劲的功能，就像在Eclipse环境下一样。然而，它们又相对太过复杂难以理解。

现在我们已经讨论了不少小部件。我们已经研究过组合框和工具条，酷工具条的组合，添加滑块以及向用户展现一个任务进展的若干方式。你或许也注意到，样例程序的代码也变得更为复杂并越来越向实际使用中那样靠近。我们谈到的有关线程问题很重要，需要我们时常谨记在心，而非仅在使用进度条和指示器时如此。

在下面的章节中，我们将探讨布局并试图向你解释如何在一个GUI应用程序中从整体来控制布局。

第六章 布局_1

第六章布局

本章涵盖

■ 充满式布局

■ 行式布局

■ 网格式布局

■ 表格式布局

■ 生成定制布局

在本书中，我们从头到底都使用了布局。现在我们已经对小部件和控制control有了牢固的掌握，我们将钻研如何使用布局来安排小部件成一个悦目的界面的复杂事情。

布局被绑定于一个合成器并用以帮助组织control。如何来理解这一概念呢？你可以想象这个过程就如同小部件们是书本，而布局则是书架。你可以在书架上堆叠书也可以竖起书本靠成一水平行。布局可以组合分割物来分割书本或者新加隔板来使放置更为有效。但是不同于真实世界的书架，SWT的布局是动态的：容器和其内容尺寸可变，可回流，并且根据你所确定的规则来放置。这些规则在SWT中理解为限制。建一个木质的书架所需的工作量不少，会耗时数天或数周。虽然生成一个UI也是耗时间的活，好在SWT的布局中有不少功能项可用这样对工作大有裨益。你或许还没做成功过一个漂亮的橡树书架，但本章中会向你展示如何在生成虚拟橱柜的过程中联系你的想象能力。

注意到：在我们进入到细节之前，有必要对 SWT 的实现方式和 Swing 的设计原则作一个比较。Swing 也有布局，诸如无处不在的边界布局，而且 Swing 的知识也无助于平缓其学习曲线。布局的算法有着一种显而易见的不同感受。SWT 的方式最小化布局，使用小部件的属性和编辑位来控制它们的位置。对应地，Swing 使用一个递归的方法来安置布局。缺点是该方式下的安置会很快变深，导致效率低下以及资源耗费巨大。相对于 Swing，SWT 的布局需要你对小部件的走向作详细规划。所以建议平时预先多备好纸笔来记录你的主意。而对于 Swing，可采用分而治之的策略，一次细分若干个小的布局，然而在放置这些小布局来形成整体的 GUI。最终，两个套件选择了不同的主打方向：设计时间简化 vs 运行简化。所以权衡两者都没有非常显著的优势，但这一不同形成了 GUI 设计师的通途。

6.1 填充式布局

我们的桌面上经常散乱着大堆的书本；在很多情况下这样的状况是简单、容易和可用的。填充式布局是堆书的布局等价物。它是一个简单的布局将各个子部件按照等间距放置充满到一个合成器空间中。默认情况下，control 被一个个肩并肩的从左至右排放。每个 control 被赋予所需的空间，在合成器的任何剩余空间都在子 control 中分割。图 6.1 展示了在填充式布局中的按钮；以下的代码说明了布局的产生。你在合成器中加入一系列的按钮，布局会调整它们的大小并以此占据可得的空间。

```
package com.swtjface.Ch6;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.layout.*;

public class Ch6FillLayoutComposite extends Composite {

public Ch6FillLayoutComposite(Composite parent) {

super(parent, SWT.NONE);

FillLayout layout = new FillLayout( SWT.VERTICAL);

setLayout(layout);

for (int i = 0; i < 8; ++i) {

Button button = new Button(this, SWT.NONE);

button.setText("Sample Text");

}

}

}
```

注意到调用的 setLayout()方法。在合成器中的该方法用以安排所有的子 control。通过这一调用，SWT 并不知道如何确定子 control 的尺寸和位置，所以什么都不能显示。（如果你的小部件显示有麻烦，则通常的原因是你忘了设定布局了）

对窗口尺寸的调整大小会改变按钮的外观，就如同图 6.2 显示。在这两个图中视觉上略有细微区别，因为填充式布局总会扩展按钮直至充满当前空间。

你可以不带参数或是带一个单一的风格参数地调用填充式布局构造器。默认的构造器使用 SWT.HORIZONTAL 风格，在此情况下布局会从左及右地安排子 control。如果使用 SWT.VERTICAL 会导致 control 从顶部到底部地安排。加入如下代码到小部件窗口就可以看到填充式布局是如何工作的：

```
TabItem fillLayoutItem = new TabItem(tf, SWT.NONE);

fillLayoutItem.setText("Chapter 6 FillLayout");

fillLayoutItem.setControl(new Ch6FillLayoutComposite(tf));
```

象堆书本一样，填充式布局仅适用与简单情况。当你的书本（control）变得越来越多时以 JFace/SWT 的方式就变得难以管理。混乱中会有书本丢失，你的视线会变得杂乱。为了高效组织，你需要类似书柜的 GUI。

6.2 行式布局

如果填充式布局象堆书一样，那么行式布局就是一个基本的书柜了。为了不局限于一堆书，你可以组织 control 为若干行，就像隔板一样。由于默认情况下行式布局会将子 control 安排为单一的行，你需要设定 SWT.WRAP 来获取附加行的功能。行式布局通过赋予你边界和空余选项来提供额外的定制功能。（注意到：所谓的行式布局略微有点用词不当，因为你可以选择水平行还是垂直行）。

让我们看看多个行如何促使用户界面包含大量的 control。如下为在小部件窗口中的程序代码：

```
package com.swtjface.Ch6;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.layout.*;

public class Ch6RowLayoutComposite extends Composite {

public Ch6RowLayoutComposite(Composite parent) {

super(parent, SWT.NONE);

RowLayout layout = new RowLayout(SWT.HORIZONTAL);

setLayout(layout);
```

```
for (int i = 0; i < 16; ++i) {

Button button = new Button(this, SWT.NONE);

button.setText("Sample Text");

}

}

}
```

将如下三行代码加入到小部件窗口中，重新确定窗口大小以显示按钮：

```
TabItem rowLayoutItem = new TabItem(tf, SWT.NONE);

rowLayoutItem.setText("Chapter 6 RowLayout");

rowLayoutItem.setControl(new Ch6RowLayoutComposite(tf));
```

如你所见，不同于以往的将所有空间用于每一个子小部件，行式布局会整合多个按钮到每一行中。布局动态地完成这些，所以当你减小窗口宽度，则按钮会向下移位，如图 6.4 所示。

行式布局的其他大多数行为特性都是通过属性值来设定的。我们仅关注如下的这些属性：

■ 折叠——一个布尔值且默认情况下为真。你可能为保持其默认状态。若将其设置为非会导致所有的 control 都安置在一个单一行内，这样会使在父合成器可视边界以外的该行末尾部分被切割掉。

■ 塞满——一个布尔值默认情况下为真。该属性值会保持子 control 们会相同尺寸，这在一个行布局情况下通常是用户想要的。你通过设定塞满来获取平均的行；在另一方面，保持 pack 属性为非会让 control 们维持其原生尺寸。

■ 调整版面——一个布尔值默认情况下为假。该属性会分配 control 平均分布于父合成器的宽度空间内。如果调整版面的属性设为真且父合成器调整了尺寸，则所有的子 control 们获取冗余间隔并重新分配空间使其右平均地占据空余空间。

6.2.1 定制个别的布局单元

你已经观察到了如何控制布局整个行为特性。然而，有的时候可能也会需要用 RowData 类来对布局中的个别子 control 尺寸进行局部修正。

图 6.4 在重新定尺寸后，行布局重新安排按钮成两列

许多布局使用布局数据机制（layout data）。具体就是你可以使用 setLayout()方法绑定于每一个 control 来帮助指导父布局，该方法可以生成一个 LayoutData 的实例。通过观察类谱系可以看到 RowData 是源自于 LayoutData，因此所有的布局都有潜能可以理解所绑定的布局数据。在实践中，你毕竟是针对每一个布局使用确切的布局尺寸类以便得到切实的结果。布局会忽略来自它们不认识的布局数据的提示。有一个约

定俗成的说法，每个布局类都支持其子control的布局数据，且其数据类的命名有规律可循：例如FooLayout有数据类称为“FooData”，诸如此类。

生成行数据线索对于行布局是简单的。所有的信息都会传递到行数据构造器。让我们给予第一两个control以更多的空间来扩展我们的小部件窗口。在如下代码中加入粗体代码：

```
package com.swtjface.Ch6;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.layout.*;

public class Ch6RowLayoutComposite extends Composite {

public Ch6RowLayoutComposite(Composite parent) {

super(parent, SWT.NONE);

RowLayout layout = new RowLayout(SWT.HORIZONTAL);

setLayout(layout);

for (int i = 0; i < 16; ++i) {

Button button = new Button(this, SWT.NONE);

button.setText("Sample Text");

button.setLayoutData(new RowData(200 + 5 * i, 20 + i));

}

}

}
```

图已经说明了使用行数据设定的效果。记得，学习这些功能项做了什么以及它们是如何相互作用的最好途径就是对程序代码的不断细微改进。经常性地，为了得到你想要的，你会需要一个风格线索、属性和布局数据的信息组合。

第六章 布局_2

6.3 网格布局

网格布局通过允许你明白地生成多个行和列来从行布局模型扩展。实际上，网格布局假以多隔板和清晰分割形成良好分区使得你可以进一步地组织你的 control。由于网格数据对象的灵活性因素，以及最终表现，网格布局已经成为一个最经常和广泛使用的布局。图 6.5 显示了一系列的按钮，这一次是由一个网格布局控制的。

如下程序是一个在小部件窗口框架内的简单网格布局的样本。我们用一个四列的网格布局并允许布局生成尽可能多的行。

```
package com.swtjface.Ch6;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.layout.*;

public class Ch6GridLayoutComposite extends Composite {

public Ch6GridLayoutComposite(Composite parent) {

super(parent, SWT.NONE);

GridLayout layout = new GridLayout(4,false);

setLayout(layout);

for (int i = 0; i < 16; ++i) {

Button button = new Button(this, SWT.NONE);

button.setText("Cell " + i);

}

}

}
```

注意到构造器采用了两个参数：列的数目和一个指代是否这些列平均分配空间宽度的布尔值。如果将该布尔值设为假，则就告诉布局对于每一个列都使用最小数量的空间。

你可以通过加入如下行来运行此合成器：

```
TabItem gridLayoutItem = new TabItem(tf, SWT.NONE);
```

```
gridLayoutItem.setText("Chapter 6 GridLayout");

gridLayoutItem.setControl(new Ch6GridLayoutComposite(tf));
```

起先，一个使用网格的构思和一个灵活布局有冲突。关键则在于使用网格布局要理解一个单一的子 control 在同时可以占据多个网格。你可以用布局数据 layout data 来实现。本例中，我们来看一下 GridData 对象，该对象为网格布局提供了如何安置一个 control 的额外信息提示。

使用 GridData 风格

GridData 在许多方面都和我们在上一节中研究的 RowData 对象相似。构造器采用了一系列的风格 style 常量，这些常量组合起来可以决定布局如何定位该单个的小部件。这些风格 style 最终可以归为三类：FILL、HORIZONTAL_ALIGN 和 VERTICAL_ALIGN。不同的 FILL 风格决定了是否可用的单元空间被充满。具体是：FILL_HORIZONTAL 指明了单元空间是水平方向充满的；FILL_VERTICAL 是垂直方向充满的；FILL_BOTH 则会导致单元空间被全部充满。在另外一个方面，ALIGN 风格决定了 control 在单元中是如何定位的。具体的值是 BEGINNING、END、CENTER 和 FILL。BEGINNING 将 control 定位于单元的左边或是最上面，而 END 将 control 放在了右边或是最下面。CENTER 将 control 居中，然后 FILL 将使 control 充满可得的空间。

使用 GridData 尺寸属性不同于 RowData，GridData 有一系列的公共属性用以设定 control 的行为。当不同的 style 设定后，有一些布尔值会自动经受管理，所以不需要直接操控他们。当然，有一些的整型数据要精确的控制单个单元的尺寸。这些属性在表 6.2 内有总结。

有两个属性 horizontalSpan 和 verticalSpan 特别重要。如同我们早先提到的那样，通过设定特定的 control 覆盖多个单元，你可以让你的 UI 看上去不再象所宣称的那样是个电子表格了。

图 6.6 说明了这一概念。我们将生成一个带有三个行和列的布局。文本区域在左下角并覆盖两个行和两个列，并允许比按钮更多的覆盖区域。注意到在上方的按钮 2 以及右侧的按钮 4、5 被分别设定为 FILL_HORIZONTAL 和 FILL_VERTICAL。作为参考，如下为设定文本区域的程序代码片断：

```
Text t = new Text(this, SWT.MULTI);

GridData data = new GridData(GridData.FILL_BOTH);

data.horizontalSpan = 2;

data.verticalSpan = 2;

t.setLayoutData(data);
```

我们设定 span 属性为 2 并告诉 GridData 我们想要尽可能地在两个方向扩展。

表 6.1 总结了 GridData 对象的各种 style 的组合

Style 常量	描述

FILL_HORIZONTAL	扩展单元来水平地充满任何空余空间。 指 HORIZONTAL_ALIGN_FILL
FILL_VERTICAL	扩展单元来垂直地充满任何空余空间。 指 VERTICAL_ALIGN_FILL
FILL_BOTH	水平和垂直地扩展单元空间。 等价于 FILL_HORIZONTAL | FILL_VERTICAL.
HORIZONTAL_ALIGN_BEGINNING	居左排列单元内容。
HORIZONTAL_ALIGN_END	居右排列单元内容。
HORIZONTAL_ALIGN_CENTER	水平居中排列单元内容。
HORIZONTAL_ALIGN_FILL	扩展单元空间充满单元内水平空余空间。
VERTICAL_ALIGN_BEGINNING	排列单元内容于单元顶部。
VERTICAL_ALIGN_END	排列单元内容于单元底部。
VERTICAL_ALIGN_CENTER	排列单元内容于垂直中央。
VERTICAL_ALIGN_FILL	扩展单元空间充满单元内垂直空余空间。

Table 6.2 GridData 尺寸属性

属性	描述	默认值
widthHint	列的最小宽度。SWT.DEFAULT 指明了没有最小宽度。	SWT.DEFAULT
heightHint	行的最小高度。SWT.DEFAULT 指明了没有最小高度。	SWT.DEFAULT
horizontalIndent	单元左侧边际与 control 之间的象素间隙数量。	0
horizontalSpan	单元所覆盖的网格的列的数目。	1
verticalSpan	单元所覆盖的网格的行的数目。	1

第六章 布局_3

6.4 表格式布局

你已看到了我们所讨论的填充式布局、行式布局和最终的网格布局在功能上稳步发展的过程。这些布局幕后都共享着同样的布局算法——按行和列来布置 control；尽管它们的复杂程度各异。而表格式布局则是这一发展路径中的分支。不同于区域划分方法，表格式布局允许你依据 control 之间的相对定位或是相对于父合成器定位来整个 control 形成一个 UI。

这样是的生成一个可调整大小的 control 表格极为容易。例如一个典型的对话框，有一个大的中央文本区域和两个按钮位于其下和其右。在本例中，理解 control 如何定位的最自然的方式就是设想他们彼此是如何相对定位的。告诉 SWT “按钮应该在文本区域下方，并且 Cancel 按钮在 Ok 按钮的右边。”尤其来关注细节而不是计算 control 之间到底有多少个行和列距离。图 6.7 显示了一个样例，在以下的章节中，我们将会讨论生成该 UI 所需的要素。

类 formLayout 其实很简单。唯一的设置选项来自于控制该布局的横向和纵向与边界的距离。以及空白属性这一属性可以让你指定在所有 control 间的距离（以象素计量）。和我们以前探讨过得布局类似，你也可以使用 FormData 实例来设置单个的 control。

6.4.1 使用 FormData

很典型地，一个 FormData 实例在一个合成器内被帮定于各个子 control。因为对于一个表格布局其核心思想就是指明各个子 control 的相对位置，所以不同于其他的布局，给予各个子 control 提供设置用 data 非常重要。如果一个给定的 control 没有一个 FormData 实例来描述它，则它会被默认为放置于合成器的右上角，而这种位置是你极少期望的。宽度和高度属性用象素来指明一个 control 的方位。顶部、底部和左右属性较为重要，且都持有一个 FormAttachment 实例。这些 attachment 描绘了在一个合成器内 control 间的关系。

6.4.2 使用 FormAttachment 指明关系

理解 FormAttachment 是使用表格布局的一个非常重要的方面。就像早先提起的，每一个 FormAttachment 实例描述了一个 control 某一面的位置。你可以以两种不同的方式使用 FormAttachment。

首先，你可以使用父合成器的百分比来指明一个 FormAttachment。例如，如果一个左侧的 FormData 被设定为 50%，则 control 的右边际会处于合成器的水平中央。同样地，如果设定顶端边界为 75%则 control 会处于合成器自上而下的四分之三处。表 6.3 总结了用以指定百分比的 FormAttachment 构造器。以百分比的形式来指明 FormAttachment 是有用的，但你不能总是应用这种方法。将你的所有 control 通用百分比方式作说明和用绝对的象素点来指明他们没有太大的区别：因为当合成器被重定大小时，如何快速定位每一个元素会变得相当困难，因为 control 并不会如你所愿地在该位置上。使用表格式布局的关键点是在于确定 control 间的相互位置，而这正是 FormAttachement 所允许的。

Table 6.3 基于百分比 FormAttachment 构造器

构造器	描述
FormAttachment(int numerator)	假定分母为 100，意味着参数即被视为一个百分比。 仅在 SWT 3.0 中可用。

FormAttachment(int numerator, int offset)	假定分母为 100，意味着参数即被视为一个百分比。 偏移量 offset 是在百分比定位的基础上再行偏移的象素数目。
FormAttachment(int numerator, int denominator, int offset)	假定分母为 100，意味着参数即被视为一个百分比。 偏移量 offset 是在百分比定位的基础上再行偏移的象素数目。

FormAttachment 第二系列的构造器是基于对其他 control 的参照。它们常常将一个 control 的边缘与相邻的 control 相对定位。通过为 button1 设定 FormData 右属性到一个基于 button2 而构建的 FormAttachment，你可以说 button1 应该总是定位与 button2 的右侧。将你的绝大多数 control 依照这种方式去定位有多种好处。你的布局代码目的就变得很容易理解：在过去的象素或是百分壁基础上的那个 control 与哪个 control 相邻的表达方式被取代后，就变得很明显了，例如：名为 foo 的 control 应该位于工具条之下；其次，表格式布局也容易维持你的这种布局意图。无论合成器的尺寸如何大小变化，它总是能够维持其正确的相对位置。

重提一次，作为指定相对位置之用的 FormAttachment 构造器有若干种形式，具体总结在表 6.4。

Table 6.4 FormAttachment constructors that accept relative positions

构造器	描述
FormAttachment(Control control)	将现有小部件添于邻接的 control 一侧的参数。
FormAttachment(Control control, int offset)	将现有小部件添于邻接的 control 一侧的参数，并且有 offset 数量象素的偏移量。
FormAttachment(Control control, int offset, int alignment)	排列 alignment 必须为 SWT.TOP、 SWT.BOTTOM、SWT.LEFT、 SWT.RIGHT 或 SWT.CENTER 其中之一。 将现有小部件添于邻接的 control 一侧的参数，并且有 offset 数量象素的偏移量。

6.4.3 使用一个表格布局管理来放置 control

现在我们已经讨论了驱动一个表格布局工作的类，我们来看一下用以生成图 6.7 的程序代码。例 6.5 生成一个文本区域和两个按钮。文本区域定位合成器的左上部。两个按钮位于文本 control 的下方，并且 Ok 按钮苦放置于 Cancel 按钮的左侧。

例 6.5 Ch6FormLayoutComposite.java

```
package com.swtjface.Ch6;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.layout.*;

public class Ch6FormLayoutComposite extends Composite {
```

```
public Ch6FormLayoutComposite(Composite parent) {
super(parent, SWT.NONE);
FormLayout layout = new FormLayout();
setLayout(layout);
Text t = new Text(this, SWT.MULTI);
FormData data = new FormData();
data.top = new FormAttachment(0, 0);
data.left = new FormAttachment(0, 0);
data.right = new FormAttachment(100);
data.bottom = new FormAttachment(75);
t.setLayoutData(data);
Button ok = new Button(this, SWT.NONE);
ok.setText("Ok");
Button cancel = new Button(this, SWT.NONE);
cancel.setText("Cancel");
data = new FormData();
data.top = new FormAttachment(t);
data.right = new FormAttachment(cancel);
ok.setLayoutData(data);
data = new FormData();
data.top = new FormAttachment(t);
data.right = new FormAttachment(100);
cancel.setLayoutData(data);
}
```

}

在此，我们定位了文本小部件，它是主要的 control，其他的都以此来作现对定位。通过设定顶部和左侧域的 FormAttachments 为百分比 0，我们锚定了文本小部件在左上角。右侧域是 100，所以文本小部件在（合成器）的水平方向空间全部充满；而且底部域的值为 75，这导致其在垂直方向占据了四分之三的空间。

Ok 按钮必须遵循两条规则：它必须总是立即处于文本区域下方并且立即处于 Cancel 按钮的左侧。我们使用顶部和右侧的 FormData 来设定这一切，即给予每一个 FormAttachment 实例以一个恰当的 control 作为一个参照。

在将 Cancel 按钮定位于文本区域下方之后，我们利用一个百分之一百的 FormAttachment 强制其处于合成器的右侧。无论你怎样重定窗口大小尺寸，按钮和文本其余总能维持它们正确的位置。

你或许可能注意到：虽然我们在为 Ok 按钮设定数据时

data.left = new FormAttachment(ok);

使用了行：

data.right = new FormAttachment(cancel);

但是却没有为 Cancel 按钮作任何相应的声明。因为第二行潜在地应用了前者，SWT 禁止你生成循环的 attachment。无论何时你有一个名为 foo 的 control 用 bar 这个 control 来参照，则 bar 决不可以再返过来使用 foo 来参照。根据 SWT 的文档，如果你有这样子的 attachment，则结果布局运算法则将无法定义，虽然 SWT 也保证过徽终止程序避免你的程序陷于一个无穷的循环之中。

将如下程序代码行加入到小部件窗口，可以观察到表格式布局是如何工作的：

TabItem formLayoutItem = new TabItem(tf, SWT.NONE);

formLayoutItem.setText("Chapter 6 FormLayout");

formLayoutItem.setControl(new Ch6FormLayoutComposite(tf));

第六章 布局_4

6.5 定制布局

标准的布局可以以不同的角色来服务于程序，并在绝大多数情况下适用。但有的时候，你需要建立一个定制套工艺精良的细木家具，一个定制化的布局可以使其内涵的东西看上去更棒。

要你再现有布局基础上生成一个定制布局来处理绝大部分情况是极少的，特别是当它们再窗口被改变尺寸大小后使用一个 ResizeListener 来调整小部件的位置时。所以只能是生成一个新的布局类来重用相同的布局逻辑于你的应用程序内，或者是在尺寸变化事件之后手工调整位置（这种做法已被证实看上去比其初始时候

精确定位时更显笨拙、丑陋）。

要演示生成一个布局管理器的过程，我们将生成一个 RadioLayout 类，该类放置其小部件成一个圆形。你或许在你的程序中不常使用这一布局，但是它确实是一个简洁易懂的运算法则，足以促使其自身成为一个很好的样例。当我们完成这一切，最终的结果会如图 6.8 所示。定制化的布局是由抽象布局类演绎而来。你只需书写两个方法：

computeSize()和 layout()。（这些是仅由布局类定义的，它们某种程度上更像是一个接口而不是一个真实的抽象类。）当父合成器被实例化并计算布局需要多少空间时，computeSize()方法即被调用。接下来便是调用一个 layout()来定位所有的 control。

6.5.1 计算布局尺寸大小

我们所要检查的第一方法时 computeSize()。相关代码如下显示：

```
protected Point computeSize(Composite composite,int wHint, int hHint,

boolean flushCache){

Point maxDimensions =calculateMaxDimensions(composite.getChildren());

int stepsPerHemisphere =

stepsPerHemisphere(composite.getChildren().length);

int maxWidth = maxDimensions.x;

int maxHeight = maxDimensions.y;

int dimensionMultiplier = (stepsPerHemisphere + 1);

int controlWidth = maxWidth * dimensionMultiplier;

int controlHeight = maxHeight * dimensionMultiplier;

int diameter = Math.max(controlWidth, controlHeight);

Point preferredSize = new Point(diameter, diameter);

... // code to handle case when our calculations

// are too large

return preferredSize;

}
```

在该方法中的参数都是简洁明了的：

■ 合成器——我们要想生成植入的对象。在该方法一经调用之时，它便有子对象，但是在屏幕上无论是合成器还是其子对象都没有定义尺寸和位置；

■ wHint 和 hHint——分别是高度和宽度的建议值。这些值代表着布局所需求的最大尺寸。它们也有可能是特殊值，如：SWT.DEFAULT，即意味着布局可以使用任意大小尺寸。

■ flushCache——一个简单的特征值，用以告诉布局使用其持有的缓存值是否安全。在我们的样例中我们不缓存如何东西，所以忽略这一特征值是安全的。

ComputeSize()的目的是在于计算我们所放置的合成器大小。特别的，这个方法不能修改任何部件的尺寸——当是时候来定位其子部件并调用 layout()方法时系统会设定父合成器的尺寸。因为我们的样例中布局放置 control 成为一个圆形，我们需要计算出一个何时的半径来适合所有的 control 以避免交迭，并为合成器返回一个适合安置所有小部件的圆的尺寸。

计算的过程是简单的。我们首先通过方法 calculateMaxDimensions()来找到最大的子部件，该方法会询问每一个子小部件它们的偏好尺寸并最终返回最大值。为了保持代码精简，我们假定每一个子部件都和最大的一样大。（这个方法在所有的对象尺寸大体相当的时候工作起来不错，但是如果在实际系统中如果某些部件比其他的要大很多的话那就会带来麻烦。）一旦我们的子对象们都已有了一个尺寸，我们以子部件总数的一半乘以该尺寸。因为一个圆的半球可以包含一半的子对象，这给出了我们该圆的直径。我们生成一个 Point 对象代表这个尺寸（另加部分余量）的平方，并返回它作为我们合成器偏好的尺寸。

6.5.2 放置小部件

既然我们已经推荐了我们合成器的一个尺寸，则 layout()方法就被调用。这是我们将所有子对象放入父合成器的一个提示。此事参数都比较简单。我们得到一个合成器并且 flushCashe 标记也是之前未改动过的。然而其逻辑则相对复杂，因为我们将不得不计算所有子对象的确切位置。要完成这个，我们使用一个或许你还记得的来自于几何学的：

$X^2 + Y^2 = R^2$

我们可以很容易地计算出 R（半径），所以我们可以选择任意的 X 坐标，计算得到 Y 坐标：

$Y = \pm$

从圆的最左端开始，layout 方法贯穿子对象的列表，通常由 X 轴方向间隔排开并以 X 坐标计算 Y 坐标。该项工作由方法 calculateControlPositions()来完成，并由 layout()调用。如下为总结的代码：

```
private Point[] calculateControlPositions(Composite composite){
```

```
... // set up control counts, max width, etc.
Rectangle clientArea = composite.getClientArea();
int radius = (smallestDimension / 2) - maxControlWidth;
Point center = new Point(clientArea.width/2, clientArea.height/2);
long radiusSquared = radius * radius;
int stepXDistance = ...
int signMultiplier = 1;
int x = -radius;
int y;
Control[] controls = composite.getChildren();
for(int i = 0; i < controlCount; i++) {
Point currSize = controls[i].getSize();
long xSquared = x * x;
int sqrRoot = (int)Math.sqrt(radiusSquared - xSquared);
y = signMultiplier * sqrRoot;
... // translate coordinates to be relative to
// actual center, instead of the origin
positions[i] = new Point(translatedX - (currSize.x / 2),
translatedY - (currSize.y / 2) );
x = x + (signMultiplier * stepXDistance);
//we've finished the upper hemisphere, now do the lower
if(x >= radius) {
x = radius - (x - radius);
signMultiplier = -1;
```

```
}

}

return positions;

}
```

这个方法应该是我们之前提到过的运算法则的简洁应用。唯一棘手的部分是我们一次防止一个半园。一旦 X 的值到达圆的最右端，我们需要翻转 X 坐标并沿着原来的路径在 Y 轴之上（即我们公式中标明的＋/－部分）。参数 signMultiplier 会为我们照看这一切。它去值 1 或-1，并控制着 X 值的增减以及 Y 值的正负。

其他在代码中的 “gotcha”会记录下我们假定圆的中心处于原点。因为需要将真是的圆心位置和每个点相对位置翻译过来。一旦我们让 caculateControlPositions()方法工作，写 layout()就容易了。我们将计算得来的位置应用倒这些父合成器的子对象上：

```
protected void layout(Composite composite, boolean flushCache) {

Point[] positions = calculateControlPositions(composite);

Control[] controls = composite.getChildren();

for(int i = 0; i < controls.length; i++) {

Point preferredSize = controls[i].computeSize(SWT.DEFAULT,

SWT.DEFAULT);

controls[i].setBounds(positions[i].x, positions[i].y,

preferredSize.x, preferredSize.y);

}

}
```

因为整个的类已经变得非常大，我们会要求每一个 control 计算其偏好的尺寸并使用该值，加上早先计算得到的位置，放置到合成器内。给每个 control 一个尺寸是关键的：如果不予以设定，conrol 的默认高度和宽度均为 0，即不可见。

6.5.3 更新小部件窗口

完整的 RadialLayout 程序代码在例 6.6 中显示。样例代码较长，但我们已对细节部分进行了检查，所以很容易照做。

例 6.6 RadialLayout.java

```
package com.swtjface.Ch6;

import org.eclipse.swt.SWT;

import org.eclipse.swt.graphics.Point;

import org.eclipse.swt.graphics.Rectangle;

import org.eclipse.swt.widgets.*;

public class RadialLayout extends Layout {

public RadialLayout() {

super();

}

protected Point computeSize(Composite composite, int wHint, int hHint,

boolean flushCache) {

Point maxDimensions =

calculateMaxDimensions(composite.getChildren());

int stepsPerHemisphere =

stepsPerHemisphere(composite.getChildren().length);

int maxWidth = maxDimensions.x;

int maxHeight = maxDimensions.y;

int dimensionMultiplier = (stepsPerHemisphere + 1);

int controlWidth = maxWidth * dimensionMultiplier;

int controlHeight = maxHeight * dimensionMultiplier;

int diameter = Math.max(controlWidth, controlHeight);

Point preferredSize = new Point(diameter, diameter);

if(wHint != SWT.DEFAULT) {

if(preferredSize.x > wHint) {
```

```
preferredSize.x = wHint;
}
}
if(hHint != SWT.DEFAULT) {
if(preferredSize.y > hHint) {
preferredSize.y = hHint;
}
}
return preferredSize;
}
protected void layout(Composite composite, boolean flushCache) {
Point[] positions = calculateControlPositions(composite);
Control[] controls = composite.getChildren();
for(int i = 0; i < controls.length; i++) {
Point preferredSize = controls[i].computeSize(SWT.DEFAULT,
SWT.DEFAULT);
controls[i].setBounds(positions[i].x, positions[i].y,
preferredSize.x, preferredSize.y);
}
}

private Point[] calculateControlPositions(Composite composite) {
int controlCount = composite.getChildren().length;
int stepsPerHemisphere = stepsPerHemisphere(controlCount);
```

```
Point[] positions = new Point[controlCount];
Point maxControlDimensions =
calculateMaxDimensions(composite.getChildren());
int maxControlWidth = maxControlDimensions.x;
Rectangle clientArea = composite.getClientArea();
int smallestDimension = Math.min(clientArea.width,
clientArea.height);
int radius = (smallestDimension / 2) - maxControlWidth;
Point center = new Point(clientArea.width / 2,
clientArea.height / 2);
long radiusSquared = radius * radius;
int stepXDistance = calculateStepDistance(radius * 2,
stepsPerHemisphere);
int signMultiplier = 1;
int x = -radius;
int y;
Control[] controls = composite.getChildren();
for(int i = 0; i < controlCount; i++) {
Point currSize = controls[i].getSize();
long xSquared = x * x;
int sqrRoot = (int)Math.sqrt(radiusSquared - xSquared);
y = signMultiplier * sqrRoot;
int translatedX = x + center.x;
int translatedY = y + center.y;
```

```
positions[i] = new Point(translatedX - (currSize.x / 2),
translatedY - (currSize.y / 2) );
x = x + (signMultiplier * stepXDistance);
//we've finished the upper hemisphere, now do the lower
if(x >= radius) {
x = radius - (x - radius);
signMultiplier = -1;
}
}
return positions;
}

private Point calculateMaxDimensions(Control[] controls) {
Point maxes = new Point(0, 0);
for(int i = 0; i < controls.length; i++) {
Point controlSize =
controls[i].computeSize(SWT.DEFAULT, SWT.DEFAULT);
maxes.x = Math.max(maxes.x, controlSize.x);
maxes.y = Math.max(maxes.y, controlSize.y);
}
return maxes;
}
private int stepsPerHemisphere(int totalObjects) {
return (totalObjects / 2) - 1;
```

```
}
private int calculateStepDistance(int clientAreaDimensionSize,
int stepCount) {
return clientAreaDimensionSize / (stepCount + 1);
}
}
```

现在我们已经有了自己定义的布局，使用它也很容易，如下所示：

```
package com.swtjface.Ch6;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
public class Ch6RadialLayoutComposite extends Composite {
public Ch6RadialLayoutComposite(Composite parent) {
super(parent, SWT.NONE);
setLayout(new RadialLayout());
for(int i = 0; i < 8; i++) {
Button b = new Button(this, SWT.NONE);
b.setText("Cell " + (i + 1));
}
}
}
```

当你用如下代码将这个类加入到小部件窗口，它将生成形成圆形的一系列按钮，就如你早先看到的那样：

例 6.7 Ch6RadialLayoutComposite.java

```
TabItem radialLayoutItem = new TabItem(tf, SWT.NONE);

radialLayoutItem.setText("Chapter 6 RadialLayout");

radialLayoutItem.setControl(new Ch6RadialLayoutComposite(tf));
```

6.6 总结

当你在使用 SWT 时，选择一个布局经常变成是一个复杂性和灵活性之间的博弈。因为可供选择的布局既有简单的填充式布局，该布局会使每一个 control 都变得尽可能地大来充满可得空间；也有行式布局，该布局会让你依照行和列来定位；还有更为复杂的网格式布局和表格式布局，它们通过使用更多的程序代码来允许使用更为高级的定位特性。

对于全部的情况，任意一个单一的布局都不会是一概的正确选择，但是通过理解这些布局你可以为你的程序选择较为恰当的布局方式。在 UI 不复杂时，你可以使用简单布局来快速得到结果，二高档的布局可以生成所需的较为美观的用户界面。

在谈到的布局内容之外，我们将使用一系列的 data 类。每个 data 类与特定的知道其如何使用的布局相关。这些 data 类的实例将粘附与单个的 control，并由布局类内置的运算法则来调谐到恰当位置。

第七章 图形_1

第七章图形

本章涵盖内容

■ 图形环境

■ 颜色

■ 字体编程

■ 图案处理

发展 SWT/JFace 的主要原因是因为其使用了操作系统的原生小部件。绝大多数用户都已习惯了他们的操作系统。他们想要小部件的外观呵行为方式相似。但有的时候，开发者需要使用内置部件并生成他自己的组件。定制化的 control 会针对一个用户接口加入独立的感观，如在视觉导向的应用程序中图案是必需的。在这种情况下，需要理解 SWT/JFace 工具套件的图形能力。

本章的目的是在于提供这样子的一种理解。为了达到这一目标，我们将从特定程序的通常概念入手。第一节将描述使得工具套件的图形能力成为可能的一个类：图形环境（the graphic context）。然后，我们会阐释 SWT 是如何同颜色一起工作以及 JFace 是如何作简化的。第三节我们将展示 SWT 和 JFace 是如何允许应用程序来在文本中使用不同的字体和图形属性。最终，我们将展示 SWT 和 JFace 的库文件是如何生成和修改图案，以及何时来在其他库中使用一个库的方法。

7.1 图形环境

图形环境功能就像是在一个 control 的顶部画上一块板。它将允许你向 GUI 组件中加入定制化的形状、图案以及多字体文本。并且在 control 的视觉发生更新时向这些图案加入事件处理功能来作为控制。

在 SWT/JFace 中，图形环境被封装与 GC 类。GC 对象添附于现存的 control 并使其可添加图形。这一章将会处理这一重要的类及其方法是如何运作的。

7.1.1 生成一个 GC 对象

构建一个图形导向的应用程序的第一步是生成一个图形环境并将其绑定于一个组件。GC 构造器方法执行了两个任务。表 7.1 中显示的是两个可用的构造器方法。

表 7.1 GC 构造器方法以及其功能

颜色构造器	功能
GC(Drawable)	生成一个 GC 并为可作图对象作设定
GC(Drawable, int)	生成冰舌定一个 GC，然后设定为文本展示风格

在第二个构造器中的风格常量决定了文本在 display 中文本是如何显现的。它有两个值：RIGHT_TO_LEFT 和 LEFT_TO_RIGHT，而默认情况下是 LEFT_TO_RIGHT。

第一个参数要求一个继承 Drawable 接口的对象。这个接口包含的方法与一个图形环境的内部有关。

SWT 提供了三个继承了 Drawable 接口的类：Image，Device 和 Control。除非你生成了你自己的 Drawable 对象，你只能向这些类或其子类的实例添加图形。在图 7.1 中的图表描述了这些关系。既然图案对象会在之后的章节中涉及，我们将在此讨论 Device 和 control。

Device 类代表着能够显示 SWT/JFace 对象的任意机制。如果你考虑倒它的两个主要子类：Printer 和 Display 的话，理解它就相对容易。Printer 代表着打印设备；Display 代表着系统控制台。关于 Display 类，因为它是任何一个 SWT/JFace 应用程序的基本类，早在第二章中就有描述。但是由于本章是在处理如何向单独的组件加入图形，我们将会把我们的 GC 对象绑定到第三类 Drawable 类——Control。

如同我们在第三章中提到的，一个 Control 对象是任意一个在后台的操作系统内有对应部分的小部件。它的类或子类都是能够被改变大小、移动并帮定于事件和图形。

图 7.1 仅有自 Drawable 接口继承而来的类可以被图形环境绑定

图 7.1 展示了由 SWT 提供的一些 Control 子类。虽然它们都能包含图形，但只有一个类是特别适合 GC 对象的：在图 7.1 的底部的 Canvas。这个类不仅提供了一个合成器的限制性属性，还可以定制一系列的风格来决定图形在其区域内如何显示。

因为这个，本章中的代码将集中于在 Canvas 对象中生成图案。由于我们已有一个方法来生成图形（GC 类）并且有一个方法来看到它们的显示（Canvas 类），那就让我们来看看这些类在一起是如何工作的。

7.1.2 在一个 Canvas 中画形状

在例 7.1 中，展示的是一个全图形的应用程序，DrawExample.java。它使用了一个 GC 对象来在一个 Canvas 实例上画线条和形状。

例 7.1 DrawExample.java

```
package com.swtjface.Ch7;

import org.eclipse.swt.SWT;

import org.eclipse.swt.graphics.*;

import org.eclipse.swt.widgets.*;

public class DrawExample {

public static void main (String [] args) {

Display display = new Display();

Shell shell = new Shell(display);

shell.setText("Drawing Example");

Canvas canvas = new Canvas(shell, SWT.NONE);

canvas.setSize(150, 150);

canvas.setLocation(20, 20);

shell.open ();

shell.setSize(200,220);

GC gc = new GC(canvas);

gc.drawRectangle(10, 10, 40, 45);

gc.drawOval(65, 10, 30, 35);

gc.drawLine(130, 10, 90, 80);

gc.drawPolygon(new int[] {20, 70, 45, 90, 70, 70});

gc.drawPolyline(new int[] {10,120,70,100,100,130,130,75});

gc.dispose();

while (!shell.isDisposed()) {
```

```
if (!display.readAndDispatch())

display.sleep();

}

display.dispose();

}

}
```

图 7.2 展示了由 DrawExample 类生成的 GUI。

这个样例演示了在你处理 GC 对象是需要牢记脑中的两个重要概念。首先，程序在调用 shell.open()方法之前构造了 Canvas 对象；之后在生成和使用 GC 对象。因为 open()方法清空了 Canvas 的显示所以这一顺序是需要的。这也意味着图形环境必须和 Shell 对象在同一个类内生成。其次，程序在用毕 GC 对象后会即可清空其所占的内存空间。这样作可以快速释放计算机的内存资源并不至于影响到作图过程。

除了在 DrawExample.java 中使用外，GC 类提供了大量的可以在 Drawable 对象上作图和填充图形。这些都罗列在表 7.2 中。

表 7.2GC 类中的 Drawing 方法

方法	功能
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)	用给定的起点和参数画一条曲线
fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)	用背景色画一条圆弧
drawFocus(int x, int y, int width, int height)	用给定的顶点画出一个 focus？矩形
drawLine(int x1, int y1, int x2, int y2)	在给定的并行线间画一条直线
drawOval(int x, int y, int width, int height)	用给定的中心点和尺度画出一个椭圆
fillOval(int x, int y, int width, int height)	用给定的尺度填充一个椭圆
drawPolygon(int[] pointArray)	用给定的顶点画出一个闭合的图形（多边形）
fillPolygon(int[] pointArray)	填充一个给定点的闭合图形（多边形）
drawPolyline(int[] pointArray)	经由多个节点画一直线至指定的终点
drawRectangle(int x, int y, int width, int height)	用给定的起点和坐标位置画一个矩形
fillRectangle(int x, int y, int width, int height)	画出并填充一个给定坐标的矩形
drawRectangle(Rectangle rect)	基于一个对象画出一个矩形
fillRectangle(Rectangle rect)	基于一个对象填充一个矩形
drawRoundRectangle(int x, int y, int width, int height, int arcWidth, int arcHeight)	用给定的宽度、高度以及弧度来生成一个圆角的矩形
fillGradientRectangle(int x, int y, int width, int height, boolean vertical)	画出一个矩形并填充以前景色至背景色的渐变色

DrawExample 的一个问题是每次当社 shell 被遮盖或是最小化时其形状就会被抹去。这很重要需要引起极大关注，因为我们需要确认在窗口变化时图形要保持可见。由于这个原因，SWT 在一个 Drawable 对象被刷新后让你自行控制。这个更新的过程就被称为 painting。

7.1.3 Painting 和 PaintEvents

当一个 GC 方法在一个 Drawabel 对象上画出一个图案，它仅执行这个 painting 过程一次。如果用户改变对象尺寸或是用另一个窗口去覆盖它，则图形会被消除。因此，应用程序能否在外界时间影响下维持其外观这一点相当重要。

这些外部事件被称为 PaintEvents，接收它们的程序接口是 PaintListener。一个 Control 在任何时候当其外观被应用程序或是外界活动改变都会触发一个 PaintEvent。这些类对于事件和监听器的使用方式都和我们在第四章内提到的类似。如下的程序片断显示了一个样例；因为一个 PaintListener 仅有一个 eventhandling 方法，所以不需要适配器类：

```
Canvas canvas = new Canvas(shell, SWT.NONE);

canvas.setSize(150, 150);

canvas.setLocation(20, 20);

canvas.addPaintListener(new PaintListener() {

public void paintControl(PaintEvent pe) {

GC gc = pe.gc;

gc.drawPolyline(new int[] {10,120,70,100,100,130,130,75});

}

});

shell.open();
```

使用 PaintListener 的一个有趣的方面是每一个 PaintEvent 对象都包含有其自己的 GC。有两方面很重要的原因：首先，因为 GC 实例由事件生成，而由 PaintEvent 负责将其销毁。其次，应用程序可以在 shell 打开之前生成 GC，这意味着图形可以在一个独立的类里面进行设置。

SWT 在 PaintListener 接口内优化 painting 过程，并且 SWT 的开发者强烈建议 Control 的 painting 仅对 PaintEvent 作出反应。如果一个应用程序因为其他原因必须更新其图形，则他们推荐使用 control 的 redraw()方法，这会在队列中加入一个 paint 请求。之后，你可以调用 update()方法来处理所有的绑定于该对象的 paint 请求。

需要牢记的是，虽然对于 Control 对象推荐在一个 PaintListener 内 painting，但是由于 Device 和 Image 对象并不能在该接口内使用。如果你需要在一个 image 货 device 内生成图形，你必须单独地生成一个 GC 对象并在使用结束后将其销毁。

7.1.4 Clipping 和 Canvas 风格

默认情况下给予一个图形环境的可得面积和被绑定的 Control 的相等。然而，GC 还提供了为其图形区域建立边界的方法。SetClipping()方法会指明 GC 图形的限定范围，而 getClipping()方法则返回剪辑区域的坐标。

当你在处理 PaintEvent 时，剪辑的概念也同样重要。不仅是因为一个 Drawable 对象被另一个窗口覆盖时这些事件被激发，而且还确保区域概念被模糊时保持清晰。也即，如果一个用户用第二个窗口覆盖了部分的 Canvas，PaintEvent 会根据被隐蔽的封闭矩形确定哪块区域被剪辑并给出 x 和 y 坐标以及高度、宽度。因为 repainting 仅是刷新被剪辑区域而非整个对象，因此上述步骤就显得必须了。如果是一个 Control 对象的多个片断被覆盖，默认情况下，对象会合并这些片断从一个单一的区域并对其 repaint。

然而，如果一个应用程序要求每个被覆盖的区域独立地发出刷新请求，则 Control 应该通过设定的 NO_MERGE_PAINTS 风格来进行构造。这是第一次出现的被绑定于 Composite 类但却是特别供 Canvas 对象使用。余下的风格在表 7.3 中有显示。

表 7.3 Canvas 对象的风格选项

风格	功能
NO_MERGE_PAINTS	在各个 paint 独立请求时保持协调
NO_FOCUS	指明 Canvas 不能被聚焦
NO_REDRAW_RESIZE	指明 Canvas 在 Control 被重定大小时不 repaint 自己
NO_BACKGROUND	指明 Canvas 没有默认的背景色

通常，当一个用户点击一个窗口，则任何的键盘输入将会指向它。这个特性被称为聚焦行为（focus behavior），而且你可以借助使用 NO_FOCUS 风格构建对象来从一个 Canvas 对象中移除这一特性。相似地，当一个 Canvas 被重定大小，默认情况下一个 PaintEvent 会被激发然后 display 被 repaint。你可以通过使用 NO_REDRAW_RESIZE 风格来改变这一默认行为。有一点很重要需要引起注意：使用这个风格可能情况下会导致一个重定大小操作期间的图形假相。

在一个图形环境画出其图案之前，其 Canvas 使用其 shell 的颜色来，即默认的背景色来 paint 它自己。这些 paint 操作有可能在某些显示器上产生屏幕闪烁现象。你可以通过使用 NO_BACKGROUND 风格来预防这个，该风格可以阻止首次的 painting。没有了一个背景色的存在，则图形环境必须要覆盖至 Canvas 的每一个象素点，否则就会出现 shell 后面的屏幕外观。

现在，我们已经开始讨论了图形环境相关的颜色问题，那么让我们就次展开细致探讨。

第七章 图形_2

7.2 颜色编程

作为任意一个图形工具套件的基本方面之一是其对颜色的使用。在颜色背后的理论说来简单，但是它们的实际应用还需要多加阐释。

本节将讨论在 SWT 中颜色是如何被展示的，以及在应用程序中如何分派和回收它们。另外还会讨论有 JFace 类苦提供的两个类是如何简化这方面处理过程的。

7.2.1 SWT 的颜色开发

因为显示器使用光线来提供颜色，它使用光的红、绿、蓝三基色组合成一个 display 的颜色生效的。这样子的颜色系统是添加式的，这意味着颜色是在黑色区域通过添加不同比例的红、绿、蓝的元素而成的。例如，如果在某一点上使用 24bit 来指代 RGB 值，则黑色（无光线）的十六进制值表示为 0x000000，而白色（全部光线的合成）表示为 0xFFFFFF。SWT 依此惯例提供了类和方法来获取和使用 RGB 对象。

这个概念看上去简单，但是当将其应用到多平台上时 SWT 的设计者面临了一系列严峻的挑战。这个问题涉及到要在不同的显示分辨率和颜色管理策略下如何提供一套颜色的标准。

首先，SWT 通过使用 display 的 getSystemColor()方法提供了一套 16 基本色（称为系统颜色）。这个方法采用了一个整数来代表 SWT 的颜色常量之一并返回一个 Color 对象。这些常量在表 7.4 中予以了列举并辅以它们的 RGB 代表。

表 7.4 SWT 提供的默认系统颜色

SWT 颜色常量	Color 十六进制值
SWT.COLOR_BLACK	0x000000
SWT.COLOR_DARK_GRAY	0x808080
SWT.COLOR_GRAY	0xC0C0C0
SWT.COLOR_WHITE	0xFFFFFF
SWT.COLOR_RED	0xFF0000
SWT.COLOR_DARK_RED	0x800000
SWT.COLOR_MAGENTA	0xFF00FF
SWT.COLOR_DARK_MAGENTA	0x800080
SWT.COLOR_YELLOW	0xFFFF00
SWT.COLOR_DARK_YELLOW	0x808000
SWT.COLOR_GREEN	0x00FF00
SWT.COLOR_DARK_GREEN	0x008000
SWT.COLOR_CYAN	0x00FFFF
SWT.COLOR_DARK_CYAN	0x008080
SWT.COLOR_BLUE	0x0000FF
SWT.COLOR_DARK_BLUE	0x000080

如果你所使用的颜色在该系列之外，则你必须要根据其 RGB 值来分派一个 Color 对象。你可以通过调用如表 7.5 之内显示的 Color 类绑定的两个构造器方法之一来完成自选色。如果一个 display 的分辨率太低无法显示这一颜色，则它会选用与其 RGB 值最为相近的系统色。

在这两个构造器中，第一个参数是一个 Device 类的对象。其后，时一个 color 的三个介于 0~255 之间的 RGB 值，或者时一个 RGB 类的实例。这个 RGB 类的构造器是 RGB(int, int, int)，即用器元素的值来描述的一个 color。有一点很重要需要牢记的是生成的一个 RGB 实例不能生成一个 color，并且一个 RGB 对象不需要作销毁处理。

表 7.5 Color 类的构造器方法

Color 构造器	功能
Color(Device, int, int, int)	根据单独的 RGB 值分派一个 color
Color(Device, RGB)	根据给定的 RGB 对象分派一个 color

在例 7.2 中的代码生成一个 Canvas 显示了两个彩色的形状。此刻我们推荐你在小部件窗口项目内生成一个名为 com.swtjface.Ch7 的程序包并添加 Ch7_Colors 类。

```
package com.swtjface.Ch7;

import org.eclipse.swt.SWT;

import org.eclipse.swt.graphics.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.events.*;

public class Ch7_Colors extends Canvas {

public Ch7_Colors(Composite parent) {

super(parent, SWT.NONE);

setBackground(this.getDisplay().

getSystemColor(SWT.COLOR_DARK_GRAY));

addPaintListener(drawListener);

}

PaintListener drawListener = new PaintListener() {

public void paintControl(PaintEvent pe) {
```

```
Display disp = pe.display;

Color light_gray = new Color(disp, 0xE0, 0xE0, 0xE0);

GC gc = pe.gc;

gc.setBackground(light_gray);

gc.fillPolygon(new int[] {20, 20, 60, 50, 100, 20});

gc.fillOval(120, 30, 50, 50);

light_gray.dispose();

}

};

}
```

这些代码演示了应用程序可以得到和使用 color 的两个途径。在构造器中，getSystemColor()方法将返回一个基本色 SWT.COLOR_DARK_GRAY，这一基本色无需作销毁。由 PaintEvent 生成的图形环境使用 Color()构造器来为一个新的 color 分派资源。这个 color，即 light_gray，是使用了三个十六进制数值对应于红、绿、蓝三色生成的。在最后一次的使用后，light_gray color 即被收回。在图 7.3 中显示了这些颜色。

在这两个案例中，要生成一个 color 需要一个 Display 对象。这由绑定于 Canvas 的 getDisplay()方法来实施。但是由于 PaintListener 接口不能接触到构造器成员，所以，取而代之的它使用了 PaintEvent 的 display 域。

两个案例中使用的 setBackground()在分派 color 中扮演了一个重要的角色。在类构造器中，该方法设定了 Canvas 的背景色，即 DARK_GRAY。这个方法重用来把 color 加入到 PaintEvent 的 GC，这个即是三角形和椭圆的 color。值得注意的是 setForeground()方法并不是完全必须。

SWT 的 color 运作是个简洁的过程，但是还有办法使其更为简化。为此，JFace 提供了如何减少管理 color 工作量的类。

图 7.3 程序使用系统色 SWT.COLOR_DARK_GRAY 并根据 RGB 值来生成 light_gray 的 color。

7.2.2 JFace 的 color 附件功能

Jface 和 SWT 使用一样的 color 方法。它还提供两个有趣的类来简化 color 的操控：JFaceColor，位于 org.eclipse.jface.resource 程序包；ColorSelector，位于 org.eclipse.jface.preference 程序包。

JFaceColors 类

JFaceColors 类包含有大量的静态方法，你可以在一个 Eclipse 的 Workbench 应用程序中用其得到 color。

GetBannerBackground()返回一个应用程序标题栏的 color，而 getBannerBackground()返回显示错误信息的小部件边界颜色。同时还有返回不同类型 text 的 color 的方法。

JfaceColor 还提供一个在 SWT 或者 JFace 应用程序中都可以被调用的游泳方法：setColors()，使用该方法你可以一次性地设定某一个小部件的前景色或是背景色。如下的程序代码片断可以使得按钮的前景色为红，其背景色为绿：

```
Button button = new Button(parent, SWT.NONE);

red = display.getSystemColor(SWT.COLOR_RED);

green = display.getSystemColor(SWT.COLOR_GREEN);

JFaceColors.setColors(button,red,green);
```

还有一个叫做 disposeColor()的方法，它尽管名义上的功能是立刻销毁所有的 color，但是在 Color 类中它还是不能取代 dispose()方法。取而代之地，它是在 workbench 要销毁其 color 资源时执行的一个额外任务。

ColorSelector 类

这个由 JFace 工具套件提供的另外一个类可以让用户在一个应用程序中选择颜色。

虽然 ColorSelector 时 JFace 的选择框架的一部分，我们觉得有必要在此提及其功能。本质上，该类香一个 SWT 的颜色对话框类实例添加一个按钮。图 7.4 中展示了一个样例。

图 7.4 ColorSelector 允许用户选择一个 RGB 对象

ColorSelector 根据用户的选择设定和得到 RGB 值。方法 setColorValue()在对话框生成之时设定了默认的选择。方法 getColorValue()将用户的选择换算成一个 RGB 对象以便于用于分派一个 color。如上即为如下的程序片断：

```
ColorSelector cs = new ColorSelector(this);

Button button = cs.getButton();

RGB RGBchoice = cs.getColorValue();

Color colorchoice = new Color(display, RGBchoice);
```

Color 使得一个 GUI 得以改观，但是它们并不能表达有用的信息。一个良好的用户界面需要和用户进行沟通。这意味着需要加入文本，也即意味着需要使用控制文本如何显示的资源。这些资源被称为字体（font）。

第七章 图形_3

7.3 用字体显示文本

象使用 color 一样，字体变成简单而易于理解，但是在此还有重要的细节需要考虑。本节将会演示应用 SWT 工具套件来选择、分派和销毁字体的方式。然后，我们将向你演示如何应用 JFace 简化这些任务。

为了达到维持原生操作系统行为和外观特性的目标，SWT/JFace 工具套件主要依赖操作系统提供的字体。不幸的是，这些字体在不同的平台上是有变化的。因此，当本节中描述某一个字体的名字时，它意味着这是安装于你的系统之上的某一字体名字。

7.3.1 SWT 中使用字体

SWT 提供了一系列的字体选择类来完成上述的三个功能之一。首先谈及的是字体 management-allocating 和 deallocating Font 对象。第二个功能是在对象中应用字体来改变所要显示的文本。最终，SWT 还提供方法来在图形应用程序中测量文本的尺度。

字体管理

就如同 RGB 对象包含有用以生成 color 对象的信息一样，FontData 对象也提供了用以生成 Font 实例所需的基本数据。这些数据由三部分组成，也即通常的 FontData 构造器方法的三个参数：

FontData(String name, int height, int style)

第一个参数代表了字体的名字，诸如 Times New Roman 或 arial 之类。第二个 height 指字体中的点数，style 代表这字体表面的类型：SWT.NORMAL（通常）、SWT.ITALIC（斜体）、或 SWT.BOLD（粗体）。

另外，你可以通过指明其 locale（即应用车舞女刮须的地理位置以及所需使用的字符集）来定制化一个 FontData 对象。一个字体的 locale 可以借助调用方法 getLocale()来确定和利用方法 setLocale()来指明。

RGB 和 FontData 实例都不需要做销毁处理，但是 Font 对象要求作分派和销毁。表 7.6 代表着 Font 类可用的构造器方法。

表 7.6 Font 类的构造器方法

字体构造器	功能
Font(Device, FontData)	根据 FontData 对象来分派一个 font
Font(Device, FontData[])	根据 FontData 对象数组来分派一个 font
Font(Device, String, int, int)	根据名字、尺寸和风格来分派一个 font

对于 Font 类只有一个唯一的销毁方法：dispose()。你应该在最后一次使用 Font 之后马上就调用它。

在对象中应用字体

在 SWT 中，font 通常被帮定于两个 GUI 对象之一：Control 和 GC。当你使用 setFont()方法将其帮定于 Control 时，任何使用 setText()方法来显示的文本将使用该指定的字体。图形环境也会使用 setFont()方法，但是它们会提供大量不同的方法来为其剪辑区域勾画文本。这些方法在表 7.7 中显示。

Table 7.7 Text methods of the graphic context (GC) class

图形环境文本方法	功能
drawString(String, int, int)	在给定坐标显示字符串
drawString(String, int, int, Boolean)	在给定坐标和使用背景显示字符串
drawText(String, int, int)	在给定坐标显示字符串
drawText(String, int, int, Boolean)	在给定坐标和使用背景显示字符串
drawText(String, int, int, int)	在给定坐标和使用标记显示字符串

因为 drawString()和 drawText()方法被频繁使用，这个表需要略作解释。虽然方法 drawString()和 drawText()有相同的参数类型和功能，但是区别在于 drawText()可以处理回车符和 tab 扩展，而 drawString()则不行。另外，在 string 之后的整数代表着文本 display 的坐标。

在第二及第四个方法的布尔型参数指代了是否文本背景为透明。如果该值设定为 TRUE，则包含文本的矩形区域的颜色就无法被更改。如果值为 FALSE，则该矩形的颜色就设定为图形环境的背景。

在最后一个 drawText()方法第三个整数参数代表着改变文本 display 的标记。这些标记如下：

■ DRAW_DELIMITER—如有需要将文本分多行显示

■ DRAW_TAB—在文本中扩展 tab

■ DRAW_MNEMONIC—加下划线？

■ DRAW_TRANSPARENT—决定文本的背景是否和其绑定的对象颜色相同

许多这些标记在下面的程序代码中将会被用到。

测量字体参数

当在 GUI 中整合文本时，你或许想要知道文本的大小尺度，也即意味着知道一个给定的字体的度量。SWT 通过其 FontMetrics 类来提供这一信息，该类包含很多方法来确定这些参数。这些在表 7.8 中显示。

这个类没有构造器方法。取而代之地 GC 对象必须调用其 getFontMetrics()方法。它会为在图形环境中使用的字体返回一个 FontMetrics 对象并让你使用列表中的方法。每一个方法根据象素点的数目返回一个给定维度上测量得到的整数。

表 7.8 FontMetrics 类中的测量方法

FontMetrics 文本方法	功能
getAscent()	从字符顶部的基准线返回距离值
getAverageCharWidth()	返回字符的平均宽度
getDescent()	从字符底部的基准线返回距离值
getHeight()	返回递增、递减和首要区域的总和
getLeading()	返回字符顶部至凸起标记间的距离

现在我们已经讨论了字体的管理、集成和测量问题，在实际编程中这些类都很重要。

7.3.2 字体编程

在例 7.3 中，类 Ch7_Fonts 延伸了 Canvas 并生成一个选定字体文本的图形环境。当用户点击按钮，一个 FontDialog 实例被打开。该对话框决定了在当前系统上那些字体可用并让用户选择在 Canvas 中文本的（字体）名字，尺寸和风格。

一旦用户作出了选择，图形环境将调用其 getFontMetrics()方法来显示文本维度。

这个类将会被加入到小部件窗口中，所以我们推荐你将类 Ch7_Fonts 放入到 com.swtjface.Ch7 程序包中。

```
package com.swtjface.Ch7;

import org.eclipse.swt.SWT;

import org.eclipse.swt.graphics.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.events.*;

public class Ch7_Fonts extends Canvas {

static Shell mainShell;

static Composite comp;

FontData fontdata;

public Ch7_Fonts(Composite parent) {

super(parent, SWT.BORDER);

parent.setSize(600, 200);

addPaintListener(DrawListener);

comp = this;

mainShell = parent.getShell();

Button fontChoice = new Button(this, SWT.CENTER);

fontChoice.setBounds(20,20,100,20);
```

```
fontChoice.setText("Choose font");
fontChoice.addMouseListener(new MouseAdapter() {
public void mouseDown(MouseEvent me) {
FontDialog fd = new FontDialog(mainShell);
fontdata = fd.open();
comp.redraw();
}
});
}
PaintListener DrawListener = new PaintListener() {
public void paintControl(PaintEvent pe) {
Display disp = pe.display;
GC gc = pe.gc;
gc.setBackground(pe.display.
getSystemColor(SWT.COLOR_DARK_GRAY));
if (fontdata != null) {
Font GCFont = new Font(disp, fontdata);
gc.setFont(GCFont);
FontMetrics fm = gc.getFontMetrics();
gc.drawText("The average character width for this
font is " + fm.getAverageCharWidth() + " pixels.", 20, 60);
gc.drawText("The ascent for this font is " +
fm.getAscent() + " pixels.", 20, 100, true);
gc.drawText("The &descent for this font is " +
```

```
fm.getDescent()+ " pixels.", 20, 140,

SWT.DRAW_MNEMONIC|SWT.DRAW_TRANSPARENT);

GCFont.dispose();

}

}

};

}
```

一旦用户惦记了选择字体按钮，则 MouseEvent 处理器生成一个 FontDialog 实例并通过调用对话框的 open()方法来使其可见。这个方法返回一个 FontData 对象，该对象在 DrawListener 接口中使用并维图形环境生成一个 font。这个由 PaintEvent 生成的 GC 对象然后调用器 getFontMetrics()方法来测量该 font 的参数。

当图形环境设定器前景色维 SWT.COLOR_DARK_GRAY 后，着通常意味着所有通过 GC 生成的文本都会被这中颜色环绕。

然而，当你看到图 7.5，仅有第一个 drawText()方法是被该前景色所环绕；这是因为第二及第三个召用被认为是透明的并且采用了底下的 Canvas 的颜色。第三个 drawText()方法也启用了快捷字符，即在一个字母前的&符号导致该字符被下划线突出。这在图 7.5 中在第三句中极为 d 下添加了下划线。

图 7.5 Ch7_Fonts.java 的用户界面。该应用程序整合了 SWT 字体操控的许多元素。

在 drawListener 接口中，仅当 FontDialog 设定了 FontData 对象后大量的处理过程才开始执行。这种安排是必要的，因为一旦 FontData 参数为空就会导致出错。同样，因为图形环境仅在一个 PaintEvent 后才写出文本，需要调用 redraw()方法才能终止 MouseAdapter（redraw 方法可导致 Canvas 去 repaint 自己）。

7.3.3 用 JFace 改进字体管理

我们在 7.3.1 中谈到 SWT 的图形主要功能之一是提供字体管理——即分派和销毁字体资源。这可以借助 Font 构造器和 dispose()方法来完成，但是在一个单独的应用程序中却没有更为有效的手段来管理多字体情形。而 JFace 则用其 FontRegistry 类来提供这一功能，该类位于 org.eclipse.jface.resource package 程序包。

通过应用一个 FontRegistry，你就不必为了 Fonts 的生成和销毁而劳神了。事实上，FontRegistry 的 put()方法可以让你将一个字符串值同一个 FontData[]对象对应匹配。这个方法可以多次调用来想注册表加入多个 Font。然后，当一个应用程序需要一个新的字体来改变文本显示时，就调用注册表的 get()方法，该方法可以通过基于字符串值参数返回一个 Font 对象。如下即为程序样例：

```
FontRegistry fr = JFaceResources.getFontRegistry();
```

```
fr.put("User_choice", fontdialog.getFontList());

fr.put("WingDings", WDFont.getFontData());

Font choice = fr.get("User_choice");
```

为了不至于去白手起家去生成一个空的注册表，该程序代码使用了一个之前存在的使用 JFace 绑定的 FontRegistry 并加入两个字体。注册表的第一个字体是一个 FontDialog 的选择的结果，而第二个则是取自于应用程序之前已经存在的一个字体。在最后一行，FontRegistry 将帮定于 FontDialog 的 FontData[] 对象转化为一个 Font 实例。

就像 FontRegistry 管理这 font 的生成一样，它也执行这 font 的销毁，包括在注册表的其他字体莫不如此。

在 fontRegistry 中的字体也包括那些应用于 Eclipse 中 Workbench 的标题和对话框中字体。然而你需要使用 JFaceResouces 类来取得它们。如下的代码展示了这是如何完成的。需要注意的是调用注册表的 get() 方法的 String 字符串，包括 FontRegistry 自身都应该是 JFaceResources 的成员域：

```
FontRegistry fr = JFaceResources.getFontRegistry();

Font headFont = fr.get(JFaceResources.HEADER_FONT);

Font dialogFont = fr.get(JFaceResources.DIALOG_FONT);
```

表 7.9 中列举了带有字符串值的这些字体以及其功能

表 7.9 在 JFace 的 FontRegistry 中可得的字体

字符串	字体功能
JfaceResources.BANNER_FONT	在 JFace 标题中用的字体
JfaceResources.DEFAULT_FONT	标准 JFace 字体
JfaceResources.DIALOG_FONT	在 JFace 对话框中用的字体
JfaceResources.HEADER_FONT	在 JFace 头部用的字体
JfaceResources.TEXT_FONT	在 Workbench 文本中用的字体

虽然在左栏中的字符串值在不同平台中保持一致，但是其指向的具体字体却有不同。例如，在 Linux 中，标题栏字体是 Adobe Courier，印刷黑体字，14 度倾斜。而在 MacOS 中默认的标题栏字体则是 Lucida Grande，粗体，和 12 度倾斜。

经历了颜色和字体编程，接下来我们需要讨论的是图形，它可以传达更多的信息。毕竟，一张图胜过千言。

第七章 图形_4

7.4 在图形中集成图案

虽然操控图案这个课题相对于字体和颜色而言更为复杂，但它却可以减少对于平台依赖性的关注。不同的操

作系统和应用程序或许支持着不同的文件类型，但是许多图案格式变得相当流行以至于其在几乎所有平台上都得到支持。在本节中的程序样例特别使用了这些常见的图案类型。随着本节内容的深入，关于图案编程事实上和字体编程相类似。如同为字体处理一样的方式 SWT 也为图案管理、集成提供了类和方法。另外，JFace 提供了内置的资源和注册表一减轻图案管理工作的复杂度。

7.4.1 为图案分配资源

绝大多数的应用程序仅是向用户界面添加图案文件方式来生成一个 Image 对象。在本例中，你将会使用第一个也是最为简单的一个 Image 构造器方法：

```
Image(Device, String)
```

在一个 GUI 中寻求手段来表现这一图案的应用程序会调用该方法，而且来使用 Display 对象作为第一参数，另外第二个参数则是图案文件的路径。在写作本书时，SWT 已接受了*.jpg、*.gif、*.png、*.bmp 和*.ico 文件类型。

如果图案文件停留在已知类的相同目录内，则一个 InputStream 可以用类的 getResourceAsStream() 方法来生成。借助这个 InputStream，你可以使用第二个构造器方法：

```
InputStream is = KnownClass.getResourceAsStream("Image.jpg");

Image Knownimage = new Image(Display, is);
```

经常使用的 Image 构造器方法全部列表在表 7.10 中有显示。第三和第四个构造器方法生成空 Image 实例且带有由方法参数设定的尺度。两个整数指明了图案的 x 和 y 坐标，在第四个方法中的 Rectangle 对象则勾画除了图案的边界。第五个方法会生成一个基于另一个 Image 实例的 Image 并且一个整型的标记决定了是否该图案是不予显示或是以灰色梯度显示。

最后两个构造器方法使用类 ImageData 对象来构造 Image 实例。这个 ImageData 类提供了有关一个 Image 对象的且与 device 无关的信息，并且包含有可操控该图案的方法。就像 FontData 类，ImageData 实例并不使用操作系统资源和考虑如何销毁。然而 Image 实例再也不使用时却需要调用它们的 dispose() 来销毁。

ImageData 类和其集成图案的能力将会很快予以详述。首先，很重要的是你需要理解图案是如何与应用程序集成的。

表 7.10 Image 类的构造器方法

构造器方法	功能

Image(Device, String)	使用现有文件生成一个 Image
Image(Device, InputStream)	使用一个取自于现有文件的 InputStream 生成一个 Image
Image(Device, int, int)	用给定的尺度生成一个 Image
Image(Device, Rectangle)	用一个矩形尺度生成一个空的 Image
Image(Device, Image, int)	用另一个 Image 和给定参数生成一个 Image
Image(Device, ImageData)	根据 ImageData 信息生成一个 Image
Image(Device, ImageData, ImageData)	根据一个 ImageData 对象和另一个决定其是否透明的 ImageData 对象来生成一个 Image 或 icon

7.4.2 使用图案进行图形编程

向一个 GUI 加入一个图案的过程是从生成一个图形环境开始的。然后 GC 对象会调用其 drawImage()方法，至于是以两个可能的哪一种情况成形，需要取决于图案是会以其原是尺寸呈现。这个方法再下面的程序中会予以展示。

在第四章中，我们曾经使用国一个标准的 Eclipse 图案来演示 Action 类是如何生效的。为了全面演示 SWT 的图案能力，我们需要使用一幅更大的图案。因此我们推荐你复制图案 eclipse_lg.gif 从其原来的位置 $ECLIPSE_HOME/plugins/org.eclipse.platform_x.y.z 到 com.swtjface.Ch7 package 程序包内。这样再该程序包内的任何类都可以使用这一图案。

然而，如下的程序片断仅仅是为了演示一个 Image 是如何在一个图形环境中工作的：

```
public class ImageTest extends Composite {

public ImageTest(Composite parent) {

super(parent, SWT.NONE);

parent.setSize(320,190);

InputStream is = getClass().getResourceAsStream("eclipse_lg.gif");

final ImageData eclipseData = new ImageData(is).scaledTo(87,123);

this.addPaintListener(new PaintListener() {

public void paintControl(PaintEvent pe) {

GC gc = pe.gc;

Image eclipse = new Image(pe.display, eclipseData);

gc.drawImage(eclipse, 20, 20);

gc.drawText("The image height is: " + eclipseData.height +
```

```
" pixels.",120,30);

gc.drawText("The image width is: " + eclipseData.width +

" pixels.",120,70);

gc.drawText("The image depth is: " + eclipseData.depth +

" bits per pixel.",120,110);

eclipse.dispose();

}

});

}

}
```

这段代码开始以使用一个 InputStream 方式构造一个 ImageData 对象。在本例中，由于 Image 对象无法被重定大小和改变颜色，因此开始以 ImageData 实例就变得有意义了。重定大小的过程是由 scaleTo() 方法来实施的，该方法会为 GUI 而收缩图案。这个新的图案在图 7.6 中有显示。

图 7.6 虽然一个 Image 对象可以在一个窗口内显示，但是信息提供却来自于 ImageData 实例。

当一个 PaintEvent 发生了，程序会调用 paintControl()方法。该方法会生成窗口的图形环境和一个基于 ImageData 的 Image 对象。在 image 的右侧，三个声明会根据 ImageData 实例的域提供信息。值得注意的是通过改变坐标，你还可以在图案上添加文本（或任意图形）。程序代码显示了你任何能使用 ImageData 类来获取关于图案的信息和改变它们的尺寸。不仅如此，这个类能做得还有很多。但在我们讨论 ImageData 任何生成图案之前，你需要全面理解这个类和其如何显示图案的。

7.4.3 用 ImageData 生成位图

学习 ImageData 的最简单途径就是设计、构建和显示一个该类的实例。在本例中，你将会生成一个位图并使用它来形成一个图案。这一过程会涉及到许多和 ImagaData 类相关的域、方法等，并能给你直观的感受：为什么这个类是必须的。

第一步涉及到确定该使用哪一种颜色。由于考虑到本书是按灰度级印刷的，我们将把颜色的选择框定在灰色。同时为了减少编程的工作量，我们试图将使用的不同颜色数目保持在最少。基于这样的考虑，本例使用了三个颜色——白、黑和灰并把它们组合喂一个幅赛车旗的位图。这看上去不是很令人兴奋，但是足以向你显示

如何来产生 ImageData。

图 7.7 在图案中每一个象素点都可以用缩进量、跨行数和颜色值来标明。

要告诉 ImageData 你将使用的颜色，你需要生成一个 PaletteData 类的实例。该对象包含有一个在图案内 RGB 值数组。对于在图 7.7 内显示的图案，构成的元素为：0x000000 （黑）、0x808080 （灰）和 0xFFFFFF （白）。在图案中的每一个象素点有三段信息：其 x 坐标或是缩进值、其 y 坐标或是跨越行数以及象素点的颜色，即其值或索引。因为这个图案仅由三个象素点颜色构成，你不必为每一个象素点去赋予具体的 RGB 值（0x000000、 0x808080、 0xFFFFFF）。换一种方式，使用在 PaletteData 数组中的颜色索引并赋予象素点值(0, 1, 2)将会简化并降低对于系统内存的负担。这一象素点值和颜色之间的简化映射被援引作一个“带索引的调色板”。例如，因为在上一个程序代码片断中使用的 eclipse_lg.gif 的象素点色深仅为 8 个 bit，即每个象素点的 RGB 值被赋予介于 1 至 2^8（255）之间的值。

然而，对于象素点色深大于 8 个 bit 的图案，额外的将颜色索引和颜色值之间的转换，对内存减负而言就变得得不偿失了。因此对于这些图案将直接使用调色板，即直接对于象素点赋予其 RGB 值。在 PaletteData 类中的 isDirect()方法也即告诉了是使用直接还是带索引的转换。

如果你理解了 PaletteData 类是如何工作的，那么如同例 7.4 那样，对于一个位图的编程就相当轻松了。因为这个 ImageData 对象就会被集成到一个鲜活的图形中，我们推荐你将这个 FlagGraphic 类加入到程序包 com.swtjface.Ch7 中去。

例 7.4 FlagGraphic.java

```
package com.swtjface.Ch7;

import org.eclipse.swt.graphics.*;

public class FlagGraphic {

public FlagGraphic() {

int pix = 20;

int numRows = 6;

int numCols = 11;

PaletteData pd = new PaletteData(new RGB[] {

new RGB(0x00, 0x00, 0x00),

new RGB(0x80, 0x80, 0x80),

new RGB(0xFF, 0xFF, 0xFF)

});
```

```
final ImageData flagData = new ImageData(pix*numCols,

pix*numRows, 2, pd);

for(int x=0; x<pix*numCols; x++) {

for(int y=0; y<pix*numRows; y++) {

int value = (((x/pix)%3) + (3-((y/pix)%3))) % 3;

flagData.setPixel(x,y,value);

}

}

}

}
```

这个样例开始以生成一个与黑、灰和白色对应的 RGB 对象数组的 PaletteData 对象。然后，构造一个带有图案尺度的 ImageData 实例，图案的深度和 PaletteData。因为有三种可能的颜色，深度就被设为 2，也即意味着可以支持到 2^2 的 4 中颜色。如果一个图案的颜色深度为 1、2、4 或是 8，则一个应用程序需要在其内存初始化（分派）之时为 ImageData 对象生成一个带索引的调色板；而对于深度更大的图案，应用程序将直接使用调色板。

注意到：如果用户企图生成一个深度超乎集合{1,2, 4, 8, 16, 24, 32}之外的调色板，则计算机会抛出一个名为 ERROR_INVALID_ARGUMENT 的错误。

在本例中，方法 setPixel()会给 220x120 图案的每一象素点分配一个值。这仅是 ImageData 类提供的诸多位图方法中的一个，在表 7.11 中有全系列的列表显示。

表 7.11 ImageData 类的位图方法

方法	功能
getPixel(int, int)	在指定坐标返回象素点的值
getPixels(int, int, int, int[], int)	在指定的偏移量和跨行数位置返回指定数目的象素点
getPixels(int, int, int, byte[], int)	在指定的偏移量和跨行数位置返回指定数目的象素点
getRGBs()	在带索引的调色板中返回 RGB 对象数组；若使用的是直接调色板则返回空
setPixel(int, int, int)	设定在指定坐标位置的象素点的值
setPixels(int, int, int, int[], int)	设定在指定的偏移量和跨行数位置上一定数目的象素点的值
setPixels(int, int, int, byte[],	设定在指定的偏移量和跨行数位置上一定数目的象素点的值

int)	

很重要需要牢记的是，因为 flagData 只能同 RGB、PaletteData 和 ImageData 对象一起工作，所以不需要调用 dispose()方法。仅当 ImageData 实例在用来生成一个 Image 或是 RGB 值被用来生成 Color 时，才需要进行销毁处理。然而，如果一个 Image 是构造自 flagData 的信息，它将会如图 7.8 般组装。

Figure 7.8 ImageData 对象的一个位图样例

现在，我们已经将内容覆盖了 ImageData 对象的基本知识，我们可以深入一步讨论更高级的专题。当然一个 SWT 的应用程序要和已投入商用的照片编辑工具展开竞争还需要假以时日，但是 SWT 工具套件已经提供了相当多令人印象深刻的图案操控能力。

7.4.4 用 ImageData 操控图案

不仅限于所述的位图方法，ImageData 类还包含这提供图形功效的方法。你可以在一个图案中设定象素点为透明而不是具体颜色。使用 alpha 合成法，两个图案可以合成为一个包含有两个图案元素的一个图案。最终，使用 ImageData 和 ImageLoader 类，你可以将多个图案序列化以模拟 GIF 动画文件。

透明

假以足够的颜色深度，RGB 系统可以提供可见光谱内的任意颜色。然而，一旦你想将图案中某一部分编程透明则上述方法就毫无帮助了。

无论将红、绿和蓝元素任何组合叠加，都无法产生一种可看透的颜色。所以你需要设定一个特殊的象素点值来代表透明。这种方式下，任何带有该值的象素点将会采用在背景之后的颜色。这一功能是由 ImageData 类的 transparentPixel 域来提供的。在具体程序代码中这也很简单，就像如下的片断一样：

```
flagData.transparentPixel = 2;

Image flagImage = new Image(pe.display, flagData);

gc.drawImage(flagImage, 20, 20);
```

在这段代码中，任何象素点的 FlagImage 值为 2（代表者白色）将会采用图案背景的颜色。这在图 7.9 左边的图案中有显示；而右边的图案则是将 transparentPixel 值设为 1（代表者灰色）的结果。

图 7.9 图案的透明。在左侧，所有的白色象素点都成了透明。而在右侧这是灰色象素点成了透明。

除 transparentPixel 域外，ImageData 类还包含大量提供透明的信息的方法。getTransparencyMask()方法返回一个 ImageData 对象，该对象的 mask 数组范围以内则是透明象素点；getTransparencyType()方法则返回一个整型代表着所使用的透明的类型。在许多图案边际的工具套件中，一个程序可以指明在一个图案中透明的程度。然而，由于由于在写作本书时，还没有 setTransparencyType()这个方法，因此这一特性尚未集成到 SWT 中去。

透明是一个有用的功能，但它仍是静态的。设想如果将一系列的图案组合以较短的时间间隔显示来提供连续运动的图像错觉。我们现在就进入到这个号称计算机图形效果之王：动画特级的话题。

保存和制作动画

在许多的图案类型之中，仅有 GIF 格式时支持动画的。因此，在我们深度讨论动画特级之前，我们需要描述一下 SWT 的 Image 对象是如何保存为图案文件的。这意味着要深入探究一下类 ImageLoader。

象 imageData 构造器一样，ImageLoader 类包含有可以接收图案文件或流并返回 ImageData[]对象的方法。然而，这个类的主要功用是将 ImageData[]实例转化为输出流和图案文件。这样，图形可以得以持久而不是被同它们的 Image 对象一块而被销毁。表 7.12 展示了在 SWT 中转载和保存的文件格式类型。

表 7.12 SWT 中可接收的图案文件格式

SWT 常量	图案格式 Image type
SWT.IMAGE_JPEG	Joint Photographic Experts Group (*.jpg)
SWT.IMAGE_GIF	Graphics Interchage File (*.gif)
SWT.IMAGE_PNG	Portable Native Graphic (*.png)
SWT.IMAGE_BMP	Windows Bitmap (*.bmp)—No compression
SWT.IMAGE_BMP_RLE	Windows Bitmap (*.bmp)—RLE compression
SWT.IMAGE_ICO	Windows Icon format (*.ico)

在 SWT 中构建一个图案文件的过程有两步：

1、应用程序生成一个 ImageLoader 类的实例并设定它的 Data 域等于包含着图案信息的 ImageData 或是 ImageData[]对象。

2、通过调用 ImageLoader 的 save()方法来生成图案文件。该 save()方法也可被用于生成该图案的输出流。

然而，一个应用程序在设法生成一个动画效果的 GIF 文件时必须执行额外的几个任务：

3、在序列中的每个 ImageData 对象或是帧必须被设定为按一定时间量长度来显示自己并恰当的销毁。在程序中，这是使用了 ImageData 类的 delayTime 和 disposalMethod 域来达成的。

4、应用程序必须组合这些帧在一个 ImageData 数组内并通过一个 ImageLoader 来装载该数组。

5、应用程序必须初始化 ImageLoader 实例的 repeatCount 域来指明动画序列应当重复自己多少次数。

6、ImageLoader 的 save()方法通过用 SWT.IAMG_GIF 标签来保存图案数组为一个动画 GIF 文件。

在例 7.5 中的 Ch7_Image 类演示了集成多个 ImageData 类实例到一个单独的动画 GIF 中的过程。

注意：如果你在一个支持动画的的浏览器中打开 GIF 文件效果是最好的。

例 7.5 Ch7_Images.java

```
package com.swtjface.Ch7;
```

```
import java.io.*;
import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
public class Ch7_Images {
public static void main(String[] args) {
int numRows = 6, numCols = 11, pix = 20;
PaletteData pd = new PaletteData(new RGB[] {
new RGB(0x00, 0x00, 0x00),
new RGB(0x80, 0x80, 0x80),
new RGB(0xFF, 0xFF, 0xFF)
});
ImageData[] flagArray = new ImageData[3];
for(int frame=0; frame<flagArray.length; frame++) {
flagArray[frame]= new ImageData(pix*numCols, pix*numRows, 4, pd);
flagArray[frame].delayTime = 10;
for(int x=0; x<pix*numCols; x++) {
for(int y=0; y<pix*numRows; y++) {
int value = (((x/pix)%3) + (3 - ((y/pix)%3)) +
frame) % 3;
flagArray[frame].setPixel(x,y,value);
}
}
}
ImageLoader gifloader = new ImageLoader();
```

```
ByteArrayOutputStream flagByte[] = new ByteArrayOutputStream[3];
byte[][] gifarray = new byte[3][];
gifloader.data = flagArray;
for (int i=0; i<3; i++) {
flagByte[i] = new ByteArrayOutputStream();
flagArray[0] = flagArray[i];
gifloader.save(flagByte[i],SWT.IMAGE_GIF);
gifarray[i] = flagByte[i].toByteArray();
}
byte[] gif = new byte[4628];
System.arraycopy(gifarray[0],0,gif,0,61);
System.arraycopy(new byte[]{33,(byte)255,11},0,gif,61,3);
System.arraycopy(new String("NETSCAPE2.0").getBytes(),0,gif,64,11);
System.arraycopy(new byte[]{3,1,-24,3,0,33,-7,4,-24},0,gif,75,9);
System.arraycopy(gifarray[0],65,gif,84,1512);
for (int i=1; i<3; i++) {
System.arraycopy(gifarray[i],61,gif,1516*i + 80,3);
gif[1516*i + 83] = (byte) -24;
System.arraycopy(gifarray[i],65,gif,1516*i + 84,1512);
}
try {
DataOutputStream in = new DataOutputStream
(new BufferedOutputStream(new FileOutputStream
(new File("FlagGIF.gif"))));
```

```
in.write(gif, 0, gif.length);

}

catch (FileNotFoundException e) {

e.printStackTrace();

}

catch (IOException e) {

e.printStackTrace();

}

}

}
```

这一代码的复杂性是由于这样的事实：ImageLoader.save()方法不能将一个 ImageData 数组转化为一个动画 GIF。然而这个方法缺课一从一个独立的 ImageData 对象生成一个 GIF 输出流，并且这一能力可以重复使用域数组中的每一个图案。这样通过大量的操控动作，这三个输出流融合为一个输出流并最终产生一个 FlagGIF.gif 文件。

如果这个文件在 Eclilpse 中没有即刻生成，则可以尝试右击项目名字（WidgetWindow)并选择 Refresh 功能项。文件就会在项目中出现。

现在，我们已经通过卓绝的努力讨论了 SWT 的图形处理能力，那么下来让我们看看 JFace。虽然 JFace 库不能产生令人难以置信的特殊效果，但是它可以极大简化图案处理过程。

7.4.5 用 JFace 管理图案

就象 JFace 的 FontRegistry 类简化了字体管理那样，ImageRegistry 类可以让你集成多个图案于一个应用程序内而无需为资源回收而劳神。它使用了和 FontRegistry 类相同的处理机制。要将一个图案放入注册表，你需要使用带有一个 Image 对象和一个字符串的 put()方法。当图案需要被显示时，get()方法会基于注册键返回该图案。这儿是使用 elipse_lg.gif 文件的一段样例：

```
ImageRegistry ir = new ImageRegistry();

ir.put("Eclipse", new Image(display, "eclipse_lg.gif"));

Image eclipse = ir.get("Eclipse");
```

在这个案例中，你仍需要为 Image 对象分配资源。这在如果是应用程序注册了大量的图案而仅显示一小部

分时会产生问题。因为这个原因，JFace 使用了 ImageDescriptor 类。就像 SWT 的 ImageData 类，ImageDescriptor 包含了需要一个图案的信息，而不需要作系统资源分配。帮定于 ImageRegistry 类的方法 get()和 put()也可适用于 ImageDescriptor 对象，就象下面的代码显示的那样：

```
ImageRegistry ir = new ImageRegistry();

ImageDescriptor id = createFromFile(getClass(),"eclipse_lg.gif");

ir.put("Eclipse", id);

Image eclipse = ir.get("Eclipse");
```

如果你使用 ImageDescriptor，put()和 get()方法的操作可无需为 Image 对象分配资源。这样，你可以向应用程序的 ImageRegistry 加入大量的 ImagDescriptor 而无需担心会生成图案（耗用资源）。最终，由于当 Display 对象被关闭时，一个 ImageRegistry 会丢弃其内容，你就不必担心图案资源的回收了。

7.5 更新小部件窗口

要向小部件窗口应用程序加入图形，在本节中你将生成一个包含颜色、字体和图形的合成器子类。这个容器集成了 Ch7_Colors 和 Ch7_Fonts 类，以及生成动画图案的 Ch7_Images 类。

7.5.1 构建第七章的合成器

例 7.6 代表着 Ch7_Composite 类，该类扩展了 Canvas 类并整合了来自于 7.2.1 节的图画；来自于 7.3.2 节的字体对话框和来自于 7.4.4 节的动画图案。为了使这个正确生效，你必须将文件 FlagGIF.gif 加入到程序包 com.swtjface.Ch7 中去。

```
package com.swtjface.Ch7;

import java.io.*;

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.events.*;

import org.eclipse.swt.graphics.*;

public class Ch7_Composite extends Canvas {

public Ch7_Composite(Composite parent) {
```

```
super(parent, SWT.BORDER);
Ch7_Colors drawing = new Ch7_Colors(this);
drawing.setBounds(20,20,200,100);
Ch7_Fonts fontbox = new Ch7_Fonts(this);
fontbox.setBounds(0,150,500,200);
Ch7_Images flagmaker = new Ch7_Images();
addPaintListener(new PaintListener() {
public void paintControl(PaintEvent pe) {
Display disp = pe.display;
GC gc = pe.gc;
InputStream is=getClass().getResourceAsStream
("FlagGIF.gif");
Image flag = new Image(disp, is);
gc.drawImage(flag, 255, 10);
flag.dispose();
}
});
}
}
```

Ch7_Composite 的操作简单而易于理解：它在 Canvas 左上角生成一个 Ch7_Color 合成器，在 display 的底部生成一个 Ch7_Fonts 合成器。另外在窗口的右上角构造一个图形环境来显示 FlagGIF.gif 文件。

7.5.2 向小部件窗口加入 Ch7_Composite

让我们向小部件窗口的标签夹加入一个标签来整合本章中生成的 Ch7_Composite 类。和前面的章节一样，

剩下需要更新的代码重要声明和 createContents()方法。为了节约空间，例 7.7 仅做了摘选。

```
...Previous import statements...

import com.swtjface.Ch6.*;

import com.swtjface.Ch7.*;

protected Control createContents(Composite parent) {

getShell().setText("Widget Window");

TabFolder tf = new TabFolder(parent, SWT.NONE);

...Previous tab items...

TabItem chap7 = new TabItem(tf,SWT.NONE);

chap7.setText("Chapter 7");

chap7.setControl(new Ch7_Composite(tf));

return parent;

}
```

一旦得到更新，小部件窗口中应当出现同图 7.10 相似的 GUI。不幸的是，虽然 FlagGIF.gif 文件已是一个动画了，但是应用程序装载的图案仍然是静态的。

7.6 总结

SWT/JFace 库允许应用程序显示比第三、五章的小部件更为丰富的内容。这一能力虽不完美，但是对于新生的工具套件来说已属不易。不仅仅在于它可以允许你管理和操控颜色、图案和字体，还可以提供大量的类来持有它们的信息。这对于运行于有限资源系统上的应用程序是极有帮助的。

要明了 ImageData、ImageDescriptor 和 ImageLoader 对象是困难的，但是 SWT/JFace 将图形类的角色做了清晰分类。同样，虽然由 JFace 提供的注册表看上去复杂，但是它们的功效是有助益的，所以我们希望你在构建你自己的应用程序时记得使用它们。

SWT/JFace 的特殊功效还在开发之中，但是其潜能是巨大的。我们希望若干年内 Eclipse 应用程序可以成为产生和操控专业质量图形编辑的工具，而且不仅限于图案和动画，还能整合声频、视频的全部方面。

图 7.10 更新后的小部件窗口已整合了 SWT、JFace 的颜色、字体和图形处理能力

第八章 树和列表_1

第八章树和列表

本章涵盖内容

■ 阅读器框架

■ SWT 树

■ JFace 树

■ SWT 列表

■ JFace 列表

在本章和下章中，我们将浏览由 JFace 提供的最为有用的小部件。然而，伴随着这些小部件的框架也是我们本书中最为复杂的部分之一，所以要提起精神来。

特定小部件的主要目的是为了显示数据的集合。虽然我们所谓的数据可以是从一个数据库内的行到你经常播放的一个 mp3 列表之类的任意东西，所以如何组织这些数据以一定方式来显示是一个经常发生的事情。即便到了最终的显示是多么的千差万别，但是在此之前都需要大量的任务完成数据准备。数据必须从一个源头得到，它也经常以一定次序排列或是过滤得出；而你需要以一种方式将一个文本字符串或是一个图案绑定于每一个域对象。JFace 提供了阅读器 Viewer 框架来处理这些通常的事务；在进入论述该特定小部件话题之前我们需要讨论该框架那的常规要素。

在我们对阅读器及其类有了基本的理解之后，本章将向你显示如何再一个列表或是数中显示数据。在第九章我们还会后续有表格 table 的讨论。

8.1 阅读器以及阅读器框架

Jface 提供了阅读器框架使对于特定小部件的操控更加容易。这一框架包括许多关系复杂的类和接口，如同图 8.1 所示。随着我们在每一节中循序渐进地探讨这一框架，你或许会发现经常参阅这一图表是很有帮助的。

图 8.1 阅读器框架总览图

在我们讨论这些类时，在脑海中保持这样一幅图画很重要。阅读器框架是模型－视图－控制器(MVC)设计模式的一个应用。核心的理念是将域对象（模型 Model）从用户界面（视图 View）和控制它们的逻辑（控制器 Controller）中分离出来。

在框架的核心是抽象类 Viewer 和其子类。在继承关系的谱系中我们关注的是 ContentViewer 和 StructuredViewer。

一个恰当的阅读器提供有一个介于小部件和显示和形成 MVC 三元之一的视图部分的数据之间的抽象层，为一个域对象生成一个 display。对于客户程序而言，一个阅读器提供了比小部件本身更为自然的操控用户界面。在另外一个方面，小部件已经赋予其获取数据的能力而无需担心数据源自何处。通过有 Viewer 提供的接口来操控一系列的对象，小部件能够在其内部使用时重新排列数据成最为便捷的形式而不破坏数据的原有

结构。

ContentViewer 通过使用系列的接口获取了处理以域对象形式出现的数据的能力，这些接口按照 JFace 的惯例都在其名字前冠以 I。我们将在本章中作相近描述的这些接口可以让你用来将你的应用程序定制的逻辑插入到框架中。

StructureViewer 在由 ContentViewer 提供的数据上家了一个结构。结构的细节变动可能很广泛，但是执行诸如排列、过滤之类的通常任务都是在这一层继承层面应用的。每一个小部件——树、列表、表格等都有对应的 Viewer 子类，诸如 ListViewer 或 TableViewer。小部件总是趋向于同至多一个其阅读器的实例相匹配，而一旦该阅读器被帮定于小部件，则所有的操控就由阅读器而不是直接由小部件来实施。如果试图在小部件和其阅读器之间混合调用会导致难以预测的后果。所有的具体的 Viewer 子类都提供一个构造器来将合适的小部件的实例绑定于其阅读器。

作为选择，一个构造器仅获得一个合成器提供，该合成器将实例化一个小部件来作为给定合成器的一个子对象并将其绑定于阅读器。

一旦绑定了小部件，典型地视图会提供若干个方法。这里最为重要的莫过于是添加元素至或是获取、移除元素自由小部件显示的数据集合。另外，设定标签提供者的方法也在该层面应用。在下节中我们将会更细节性地讨论的标签提供者可以从域对象生成恰当的 UI 文本。IBaseLabelProvider 接口是作为所有的 label 提供者必须应用的通用接口的。更多的特定接口是是引申自 IBaseLabelProvider 并应用到独立的小部件类型。抽象类 ContentViewer 定义了这些方法来调取 IBaseLabelProvider 接口；但是在此应用的这些方法时，你可以插入检查来确保仅有 IBaseLabelProvider 应用恰当地加诸于对于给定的小部件，而通用的逻辑是由超类来实施的。

在这些基本类中定义的方法是不能被你的程序代码所直接调用的。为此你需要提供一个带有阅读器的类来应用这些接口之一，然后阅读器会在恰当的时候调用你类中的方法。表 8.1 总结了你需要熟悉的方法以正确地使用阅读器。

表 8.1 重要的阅读器方法

方法 定义于。。。

getControl() Viewer

getSelection() Viewer

refresh() Viewer

setInput() Viewer

setContentProvider() ContentViewer

setLabelProvider() ContentViewer

addFilter() StructuredViewer

reveal() StructuredViewer

setSorter() StructuredViewer

8.1.1 Providers

Viewer、ContentViewer 的第一个子类为处理由一个小部件显示的数据加入了功能，即在 MVC 中提供了 M（模型 Model）。这是一个 provider 执行程序——给小部件作一块数据——例如：返回一个文本以便作为给定列表来显示。有两类 provider：LabelProviders 和 ContentProviders。我们讨论的类谱系已在图 8.1 中显示；在我们浏览不同的类和接口时，你或许会发现该图标的参考较为有用。

Label providers

Label providers 应用了 ILabelProvider 和 ITableLabelProvider。ILabelProvider 和 ITableLabelProvider 基本类似，差别仅在于 ITableLabelProvider 还能处理表列，反观 ILabelProvider 则被假定为是单维的数据。

LabelProvider 的逻辑周转出现于三个方法。其中第一个方法是 isLabelProperty()，它在一个 IBaseLabelProvider 内定义。给予一个对象和特性名称，该方法就可以返回一个布尔值以指征给定特性若发生变化是否需要一个对应的可视化标签也作更新。这不是强制的，但是特性遵从了典型的 JavaBean 命名规则：那就是，如果 bean 有一个特性名称，则一个 getName()和可能地一个 setName()方法则是作为为该对象定义的。当一个对象被一个编辑器（见下节）所更新时，isLabelProperty()即被调用来优化作图操作。如果 isLabelProperty()返回一个 false 值，ContentViewer 就知道不需要重画该小部件。

IbaseLabelProvider 的子接口在不同情况下也提供有 getText()和 getImage()方法。每个都给以一个对象，然后就是 provider 的责任来为该对象返回文本或图案作显示。如果返回值为空 null 则导致没有文本或是图案可供显示。在标准的 label-provider 和 tabel-label-provider 之间唯一的区别就是在 ITableLabelProvider 内的方法还采用了一个附加的参数来指代增加列的索引；余则两者相同。

Jface 提供了一个 ILabelProvider 的默认应用称为 LabelProvider，它会为所有的图案返回空值 null，会为基于给定对象文本返回调用 toString()的结果。这在 debugging 时比较有用并可以快速得到一个原型来运行，但是你通常需要将 LabelProvider 子类化或是提供一个适合于执行你的应用程序的恰当逻辑的接口的应用。通常由 toString()返回的数据并不要求用户去看。

Content providers

除了 label-provider 外，ContentViewer 会关注于 content provider。相对于一个 label-provider 的提供文本和图案为某元素来显示，一个 content-provider 则提供真实的元素来显示。IStructuredContentProvider 所定义的 getElements()方法是将一个给定的对象作为输入并返回一个对象的数组在小部件内显示。当 setInput()在阅读器上被调用，作为一个参数给出的对象被传送到 content-viewer。然后 content-viewer 会负责使用该输入参数来返回在小部件内显示的域对象集合。

一个简单的例子是关于一个 content-provider 如何展示有关来自于一个 XML 文件的信息。它可能采取将一个输入流作为一个参数，从流中解析 XML，然后返回代表着不同 XML 数据的元素给阅读器显示。在此使用 content-provider 并不是必须的，然而；如果你想 add()这些元素来显示的话，它工作起来还是不错的。

IstructuredContentProvider 的另外两个方法经常被留为空。第一个 dispose()是当阅读器要被销毁时调用的方法；你可以用此方法来清空所有该 content-provider 所持有的已分配内存资源。第二个方法则是 inputChanged(Viewer viewer, Object oldInput, Object newInput)，它是阅读器用来通知该 content-provider 原始的输入对象已经发生了改变。虽然很多应用程序可以安全地忽略这一方法，但是 JavaDocs 还是建议要使用它。假定你的应用程序已经包含了广播事件的域对象，诸如一个发送不可得通知得网络资源。当阅读器得输入由这些对象之一变更为另一个时，inputChanged()方法可以用来接触 content-provider 对于旧得输入对象的监控转而注册为针对新对象的事件。

8.1.2 监听器

不同的阅读器为广泛系列的事件监听器提供支持。基本的 Viewer 类提供对于帮助请求（help request）和选择改变（selection changed）事件的通知。观察谱系图中再往下细节，StructuredViewers 加入了（鼠标）双击事件的支持，而 AbstractTreeViewer 则加入了默认的选择和树事件的支持。关于事件和监听器我们早再第四章已经细节性地予以讨论；早先所说的原则在此同样适用。监听器被应用于应用程序后的逻辑；它们构成了 MVC 的 Controller 部分。

第八章 树和列表_2

8.1.3 过滤器和排列器

如同我们早先提到的，通常我们总是在将元素显示之前做好排序工作。这个排序任务可以以无尽的参数来完成，从在地址簿里的按字母对联系人信息排序到按接收的数据对 email 进行列表。要执行排序或是相似的数据操控，元素需要某种类型的结构。关于这个结构的知识来自于 StructuredViewer 类。由 StructuredViewer 提供的关键功能是在某过滤器内通过对象并在显示之前排列它们。

过滤器是个绝佳的主意，它可以缓冲因为要决定一大堆数据中该显示哪一些的压力。首先我们的第一反应会是仅生成需要显示的元素。然而这种方式缺乏灵活性。对于每一套要显示的对象，你必须重写读取的逻辑。另外，处于效率方面的考虑，一次读取全部对象并将它们缓存起来是更有意义的做法。向一个数据库往返重复读写常量很明显会减缓程序运行直至龟行。

为了使用一个过滤器，你需要一次装载整套数据，或是使用一个 ContentProvider 来加入对象。当到了显示数据的时候，你就调用 StructuredViewer 的 addFilter()方法，给它以一个 ViewerFilter 的应用来指明仅接收需要显示的元素。

例如，假定我们有一串单词的列表。用户可以选择以某一特定字符串打头的单词来显示。代码是简单的。它首先定义了一个过滤器：

```
public class SubstringFilter extends ViewerFilter {

private String filterString;

public SubstringFilter( String s ) {

filterString = s;
```

```
}

public boolean select( Viewer viewer, Object parentElement,

Object element) {

return element.toString().startsWith( filterString );

}

}
```

现在我们可以使用这一阅读器了：

```
StructuredViewer viewer = ...

//set content provider, etc for the viewer

SubstringFilter filter = new SubstringFilter( "" );

viewer.addFilter( filter );
```

默认情况下，这会显示每一个元素，因为所有的字符串都以空字符串开头。当用户输入字符串进行过滤，如下的代码行会更新 display：

```
viewer.removeFilter( filter );

filter = new SubstringFilter( userEnteredString );

viewer.addFilter( filter );
```

调用 addFilter()方法会自动地触发元素的重过滤，现在仅有以用户输入的字符串打头的才得以现死活。注意到没有必要为原始的对象集合而担心。阅读器始终维持着整个集合；它选择来显示的仅当 select()方法被调用，过滤器返回值为真的。很有可能在一个阅读器上存在多个过滤器；若如此，只有通过所有过滤器的元素才得以显示。

有一个警信需要清醒认识的是当你在使用一个循环出现过滤器的设计是：虽然装载整个数据集合和让过滤器处理选择在概念上很简单，但是如果你的集合可能包含着数以百万的项目是这个方法就不一定非常奏效了。在本例中，你可能不得不回复到所谓的“装载你所需要的”方法。所以，要总是考虑到你的特定应用程序的需求。

和过滤器的历年相似，StructuredViewer 也允许使用 setSorter()和一个 ViewerSorter 来定制化排列它的元素。在所有元素被过滤后，排列器就有机会来在显示之前对元素重新排序。要对早先的样例后续跟进，你可以使用一个排序器来将单词按字母排序。默认的 ViewerSorter 的应用对每一个元素排序标签时是以大小写敏感的方式进行的。对于你自己的 ViewerSorter 最简单的应用是覆盖方法 compare()，该方法和在

java 程序包 java.util.Compraator 中的 compare()方法是迥异的。Compare()给予了两个对象并返回一个负值整数，零，或是一个正值整数，来分别指明是小于、等于或是大于。

如果一个给定的变化不能改变排列顺序，则你可以使用 isSorterProperty()方法来避免重新排列。如果你需要更为复杂的比较，你可以使用 category()方法来打乱元素成不同的类别，然后每个类别又独立进行排序。例如，如果你有一列 Order 对象，category()会对内部的 order 返回值 1，对于外部的 order 返回 2；反观 compare()方法则基于 order 数进行排序。列表则会将所有的内部 order 组合在一块。然后跟着所有的外部的 order，具体排序则是由 order 的数字决定。如果与不同的类别对应也又视觉的标记，则这种技术应该是最为有效的了。

如下是使用了这种方式的应用案例之代码：

```
public class OrderSorter extends ViewerSorter {

public int category(Object element) {

//assumes all objects are either InboundOrder

//or OutboundOrder

return (element instanceof InboundOrder) ? 1 : 2;

}

public int compare(Viewer viewer, Object e1, Object e2) {

int cat1 = category(e1);

int cat2 = category(e2);

if( cat1 != cat2 ) return cat1 – cat2;

//Order is the superclass of both InboundOrder

//and OutboundOrder

int firstOrderNumber = ((Order)e1).getOrderNumber();

int secondOrderNumber = ((Order)e2).getOrderNumber();

return firstOrderNumber – secondOrderNumber;

}

}
```

注意到该例中是手工调用了 category()方法。因为我们已经覆盖了方法 compare()所以这是需要的，所以

我们如果不自己去调用 category()，则它就不会被调用到。不像过滤器，在一个给定的 StructuredViewer 上它只可能一次有一个排序器。多个排序器其实是无效的，因为么一个都会将前面其他排序器的工作推倒重来。

默认的 compare()方法的应用会基于那些已生成的标签为每个条目生成一个标签。考虑到这个因素，阅读器忽略了 compare()方法。通过将 Viewer 推向成形为一个 ContentViewer，label-provider 能够被使用的方法 getLabelProvider()获取并且用以得到为给定元素而展示的文本。在之前的案例中，如果 label-provider 被应用且可以返回为字符串格式的 order 数，则对方法 compare()的函数覆盖就不必要了。在那种情况下，你就可以幸免而避开应用 category()方法来作 inbound 合 outbound 的 order 的区分，而采信于默认的 compare()方法来作 order 的正确分组。然而，这样作由于如果 label-provider 发生变化不一定带来 order 的正确排序，所以需要引入 label-provider 和 sorter 的联接。这一问题的严重程度因不同的应用程序而异。

现在，我们对背景情况已作了大致了解，让我看看使用这些小部件的一些实例。

第八章 树和列表_3

8.2 树 Trees

一个树的 control 展示了按等级格式的数据，允许用户可以清晰地看到不同元素之间的彼此关系。你可能会较熟悉于 Windows 的 Explorer 或是你的平台上类似工具来浏览你机器上的文件系统，即每一个子文件夹都在其父文件夹下显示。这一层级可以被扩展或是隐藏，从而允许你集中精力于你感兴趣的文件系统。一个 tree 的 control 可以向你提供类似的功能给凡是象文件夹和子文件夹那样有着父/子关系的任意分组对象。图 8.2 展示了一个简单的树。

我们首先要讨论 SWT 的树小部件，接下来是来自于 JFace 的 TreeViewer，它可以简化 Tree 的使用。

8.2.1 SWT trees

树并没有特别有用的用户界面。它扩展了 Scrollable 并提供了在表 8.2 中所列的操作。

方法	描述
addSelectionListener()	使选择事件的通知功能启用 events.
AddTreeListener()	当树的某一层面发生扩展和收缩时 TreeListener 接口提供通知回调功能
select()/deselect()	修改当前选择
getSelection()	获取当前选择
show()	强制 control 滚动直至给定的条目可见

表 8.2 在一个 tree 中可进行的操作

在树中的条目有趣了点了。

树条目 TreeItem

TreeItem时用来象一个树加入条目的类。除了用以显示内容外，TreeItem还承担着维系其父条目和子条目关系的责任。一个给定条目的父条目可以通过 getParentItem()获取，而如果条目已处于树的根部则返回空值。与此同时，getItems()则会以一个TreeItem 数组的形式返回其子条目。

图 8.2 一个树展示了其父/子关系

与树有关的有两个风格选项。第一个是在 SWT.SINGLE 和SWT.MULTI 间二者选一，它将影响到同一时间内有多少个条目可被选中。默认情况下树都被设为 SWT.SINGLE，也即意味着每次当选中一个条目，之前选中的旋即消失。使用 SWT.MULTI 则可以让用户以无论何种该操作系统所支持的方式（通常是 Ctrl-或是 Shift-健组合）选中多个选项。

第二个风格是 SWT.CHECK，它可以导致在树的每一条目的左侧均添加上一个检验框。如果检验框生效了，则任一给定的 TreeItem 的状态都可以使用 getChecked()方法检视取得，该方法将返回一个布尔类型值来指代该条目是否已被检验。注意到在某些平台，一个条目被选中而无需被检验。

一个父条目如果被作 TreeItem 的构造器，则无论从那一层关系而言其都不能作修改。因为维持这种关系的方式，要从一个树中移去一个单独的条目也很困难：你必须先调用方法 removeAll()来清空树并重建其内容，来减去你想移除的条目。

TreeItem提供了的修改待显示的文本或是图像的方法，如：setText()和 setImage()。一个 TreeItem 可以使用 setExpanded(boolean)来被强制扩展或是收缩其自己。

很乐意您通过直接生成和操控 TreeItem 来构建和展示一个树，但是这样作将不得不迫使你在一个较低层面来处理小部件（而事实上你大可不必如此）。通过使用一个 TreeViewer 就可以处理你的树了，而你就可以集中精力于应用程序的逻辑而不必为了用户界面元素细节而劳神了。

8.2.2 JFace 的 TreeViewers

一个 TreeViewer 提供了如同一个 label-provider 一样的能力，就像所有的 Viewer 一样具有过滤和排序的能力。另外，一个 TreeViewer 可以使用一个 ITreeContentProvider 来根植其自身。ITeeContentProvider 接口扩展了 IStructuredContentProvider，加入了可以查询一个给定节点的父或子节点的方法。

早先我们有过关于通常的 Viewer 特性的讨论，一个 content-provider 提供了一个？？对象关系接口。例如：假定你需要在树中展示一个 XML 文档的元素，并允许用户在期间自由浏览。如果直接采用 Tree 和 TreeItem 的方式，则就需要你作贯穿该文档的事件循环并手工建立条目。使用 JDK1.4 中的 DOM 解析工具，结果的程序代码就大致如下：

```
Document document = ... //parse XML

Tree tree = ...

NodeList rootChildren = document.getChildNodes();

for(int i = 0; i < rootChildren.getLength(); i++) {

Element rootElement = (Element)rootChildren.item(i);
```

```
TreeItem item = new TreeItem(tree, NODE_STYLE);
item.setText(rootElement.getTagName());
buildChildren(rootElement, item);
}
...
/*
Recursively builds TreeItems out of the child
nodes of the given Element
*/
private void buildChildren(Element element, TreeItem parentItem) {
NodeList children = element.getChildNodes();
for(int i = 0; i < children.length(); i++) {
Element child = (Element)children.item(i);
TreeItem childItem = new TreeItem(parentItem, NODE_STYLE);
buildChildren(child, childItem);
}
}
```

作为对比，如下代码使用的一个content-provider且应用了在ITreeContentProvider中定义的琐碎方法：

```
Document document = ... //parse XML document
TreeViewer viewer = ...
viewer.setContentProvider(new XMLContentProvider());
viewer.setInput(document);
viewer.setLabelProvider(XMLLabelProvider());
```

```
...
class XMLContentProvider implements ItreeContentProvider {
public Object[] getChildren(Object parentElement) {
return toObjectArray(((Node)parentElement).getChildren());
}
public Object[] getElements(Object inputElement) {
return toObjectArray(((Node)inputElement).getChildren())
}
private Object[] toObjectArray(NodeList list){
Object[] array = new Object[list.getLength()];
for(int i = 0; i < list.getLength(); i++) {
array[i] = list.item(i);
}
return array;
}
public Object getParent(Object element) {
return ((Node)element).getParentNode();
}
public boolean hasChildren(Object element) {
return ((Node)element).getChildNodes().getLength() > 0;
}
... //additional methods with empty implementations
}
```

第一眼看去，content-provider 的代码占据了更多的空间，而且根据代码行数它显得更长。然而，我们须引起注意的是 content-provider 的概念简单且易于维护。getChildren() 和 getElements()在当前节点

调用 getChildren()方法，并将结果转换为一个数组。使用树，可将被迫对顶端元素采取与其他条目不同的处理手段，要生成两段不同的代码并按需要作更新。更为重要的是，通过使用一个 TreeViewer，content-provider，h 和 label-provider 你就在你的域对象上进行了直接操作（在本例中，是一个 XML 文档的节点）。如果文档发生变化，则需要调用 viewer 上的 refresh()方法来更新 display。在程序运行中，如果每个节点的细节都要作展示，则一个备用的 ILabelProvider 的应用可以被分配给 viewer。如果你是手工直接生成一个 Tree 和 TreeItem，那么上述的这些变数情况将要求你要么手动地贯穿 tree 始末来找到并更新相关的节点，要么白手起家重建整个 Tree。就整体而言，在设计中使用一个 content-provider 既简单又不失灵活。

还有值得概述的是方法 hasChildren()。它提供了一个树优化的线索。通常我们可以通过调用方法 getChildren()并检查返回数组的大小，但是某些情况下要获取个定元素的子条目可能代价不菲。如果一个 content-provider 能够不经计算所有的子条目而确定是否有子条目，然后若确实没有任何子条目可供显示则回 false 告诉 tree 来跳过调用 getChildren()的步骤。如果没有便捷的方式来计算这个，安全起见，即便是没有子条目可供显示，也可从方法 hasChildren()返回 true 值并让 getChildren()返回一个空数组。

第八章 树和列表_4

8.3 使用 List 小部件

一个 List 小部件代表者一个序列的项目。一个 mp3 播放器可以使用一个 List 来表现给用户以播放列表，而 Eclipse 在你选择菜单上的“打开”类型时则使用一个 List 来显示可能匹配的类。用户可以从列表中选择一个或多个值，就如同图 8.3 所示的那样。

图 8.3 一个简单列表

再说一遍，我们将会讨论到使用基本的 SWT 类来构建列表，然后再继续钻研有 ListViewer 提供的更为强劲的功能。

8.3.1 SWT 的 lists

因为该小部件是如此简单，所以很容易直接使用一个 List 而不使用一个附加的 Viewer 同样可以得到有用的结果。例如，构建一个字符串列表仅需如下的代码即可：

```
List list = new List(parent, SWT.SINGLE);

for( int i = 0; i < 20; i++ ) {

list.add( "item " + i);

}
```

象 tree 那样，List 支持 SWT.SINGLE 和 SWT.MULTI 来控制同一时间内可以有多少个条目被选中。没有其他的风格（除了由超类支持的之外）为 Tree 所支持。

注意到：如果你在开发一个运行于 Motif 的 SWT 应用程序，你应该意识到不太可能绝对地阻止一个垂直滚动条在一个列表中生成。因为，在 Motif 中，SWT.V_SCROLL 可以加入到你为一个 List 定义的随便其他

风格中以确保风格值与所要显示的东西相吻合。

List 从其超类 Scrollable 继承了滚动的能力。除非你特别指明要更改它，都则 style 总是被假定为 SWT.V_SCROLL。这也意味着如果列表的条目树超过了可得空间，则一个垂直的滚动条将会出现来帮助用户浏览列表。一般水平的滚动条不可得到除非你在风格中加入了 SWT.H_SCROLL。

我们样例的缺点在于 List 只接收 String 的实例。因此，在列表中没有办法来显示一个图案，而且更新一个域对象要求遍历整个 List 来查找其旧值，移除并用新的值来取代之。这种方式对于之前演示的简单情况可以奏效，但是实际上你极有可能想要在你的 List 上玩出更有趣的东西来。由此，你需要使用一个 ListViewer。

8.3.2 Jface 的 ListViewers

使用一个 ListViewer 是同一个 List 互动的较佳途径。在其最为基本的水平上，使用 viewer 可以取得控制小部件行为的更多功能选项，诸如在每一个元素中加入图案或是匆忙改变条目的顺序而不必重建整个列表。它也允许你的模型数据从其显示的方式中分离出来。

一个 ListViewer 由一个父合成器和一个 SWT 风格实例化而来。阅读器支持和基本的 List 一样的风格：SWT.SINGLE 和 SWT.MULTI，它们都指代了在同一时间下有多少个条目可被一起选中。

虽然一个 ListViewer 提供了一个 add()方法你可以用来直接在列表中插入对象，使用一个 ContentProvider 如同我们处理 TreeViewer 那样，是个好主意。ListViewer 使用了 IStructuredContentProvider 接口。这一接口是简单的，通常仅要求应用到方法 Object[]getElements(Object inputElement)。在阅读器内的 setInput()方法调用后，getElements()方法被调用来传递被设定为输入到阅读器的同一对象。

有了所有的不同帮助类运行于 viewer/content-provider，label-provider，过滤器和排序器，则理解它们的相互作用相当重要。

所有一切都自 content-provider 开始，而它返回了可能被显示的整个条目的集合。这一集合然后传递给被帮定于阅读器的任意过滤器，此时过滤器有机会开始移除条目。通过所有过滤器的全部条目然后进行排序并最终给了 label-provider 来确定该显示什么。

用 IStructuredSelection 来获取条目

直到这一点，我们还没有讨论到如何从一个 ListViewr 或是 TreeViewer 中获取条目。无论何时你若想查询哪些条目被选中，JFace 会提供接口 IstructuredSelection 来管理结果。

由 structuredviewer 类提供的 getSelection()返回一个 IStructuredSelection 的实例。作为一个 IStructuredSelection 暗指数据有某种结构存在，或者说是一个 order。接口提供了一个方法来获取一个选定条目的？？，即和 Collections 框架内相同的对象。？？返回一个同展现在列表上相同次序的条目。典型地，你作如下所示的条目的事件循环：

```
IStructuredSelection selection = (IstructuredSelection)viewer.getSelection();

for( Iterator i = selection.iterator();
```

```
i.hasNext(); ) {

Object item = i.next();

//process item

}
```

如果有必要，然而，接口也提供了 toArray()和 toList()方法来立即获取选定条目的整个集合。

8.4 更新小部件窗口

让我们在小部件窗口内加入两个新的合成器，一个是为了说明 tree 而另一个是为了说明 list。首先加入 Ch8TreeComposite，即如下的例 8.1。

例 8.1 Ch8TreeComposite.java

```
package com.swtjface.Ch8;

import java.util.ArrayList;

import java.util.List;

import org.eclipse.jface.viewers.ITreeContentProvider;

import org.eclipse.jface.viewers.TreeViewer;

import org.eclipse.jface.viewers.Viewer;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.Composite;

public class Ch8TreeComposite extends Composite {

public Ch8TreeComposite(Composite parent) {

super(parent, SWT.NULL);

populateControl();

}
```

```
protected void populateControl() {
FillLayout compositeLayout = new FillLayout();
setLayout(compositeLayout);
int[] selectionStyle = {SWT.SINGLE, SWT.MULTI}; 1 风格
int[] checkStyle = {SWT.NONE, SWT.CHECK};
for(int selection = 0; selection < selectionStyle.length;
selection++) {
for(int check = 0; check < checkStyle.length; check++) {
int style = selectionStyle[selection] | checkStyle[check];
createTreeViewer(style);
}
}
}
private void createTreeViewer(int style) {
TreeViewer viewer = new TreeViewer(this, style);
viewer.setContentProvider(new ITreeContentProvider() {2 ContentProvider
public Object[] getChildren(Object parentElement) {
return ((TreeNode)parentElement).getChildren().toArray();
}
public Object getParent(Object element) {
return ((TreeNode)element).getParent();
}
public boolean hasChildren(Object element) {
return ((TreeNode)element).getChildren().size() > 0;
```

```
}
public Object[] getElements(Object inputElement) {
return ((TreeNode)inputElement).getChildren().toArray();
}
public void dispose() {}
public void inputChanged(Viewer viewer, Object oldInput, Object
newInput) {}
});
viewer.setInput(getRootNode()); 3 setInput
}
private TreeNode getRootNode() { 4 getRootNode
TreeNode root = new TreeNode("root");
root.addChild(new TreeNode("child 1")
.addChild(new TreeNode("subchild 1")));
root.addChild(new TreeNode("child 2")
.addChild( new TreeNode("subchild 2")
.addChild(new TreeNode("grandchild 1"))) );
return root;
}
}
class TreeNode { 5 TreeNode
private String name;
private List children = new ArrayList();
private TreeNode parent;
```

```
public TreeNode(String n) {

name = n;

}

protected Object getParent() {

return parent;

}

public TreeNode addChild(TreeNode child) {

children.add(child);

child.parent = this;

return this;

}

public List getChildren() {

return children;

}

public String toString() {

return name;

}

}
```

1. 这两个互不相关的风格集合覆盖了对于 Tree 来说所有可能的风格。代码循环并组合它们来生成若干个样本树实例，以此演示说明所有不同的风格。
2. 在此代码定义了一个 ContentProvider，它被用来向 TreeViewer 提供数据。注意到它能假定每个方法的参数是恰当的域对象（本例中是树节点）并代码中相应加入。
3. 通过调用阅读器上 setInput()的方法开始了使用给定数据的根植树的过程。
4. 该方法构建了域对象的初始集合
5. 这个简单类在本例中作为服务于域对象。

这一窗格生成一个三层深的简单树。我们使用类 TreeNode 来作为域对象。TreeNode 的唯一功能是维持一个子条目的列表。需要注意的是关键方法是 createTreeViewer()，它会生成一个新的 TreeViewer 的实例并将其分配给一个 ITreeContentProvider。这个 content-provider 接收 TreeNode 而且知道任何来为每个节

点返回子条目。因为域对象很自然地知道它们自己的关系，应用这个 content-provider 包含有琐碎的任务：不停地查询每个节点关于其父/子条目情况并在适当时候调用 toArray()方法。你可以返回全部的域对象并让 label-provider（在本例中，是调用方法 toString()的 BaseLabelProvider）来关注任何显示它们。

在分派了 content-provider 后，你必须记得调用 setInput()并将其传递给 TreeNode 来用作树的根基。这一步将真实的域对象关联给了阅读器，否则阅读器就不知道该显示那个对象。这个根对象传给 getElements()来获取第一层的子条目。由 getElement()返回的数组中每一个元素依次传递给 getChildren()来构建再下一层的层级。这个过程延续到直至 hasChildren()返回 false 或者是 getChildren()不再返回子条目了。图 8.4 显示了当你运行该例所得结果。

图 8.4 Tree pane

为了运行该样例，你必须加入如下三行到小部件窗口：

```
TabItem chap8Tree = new TabItem(tf, SWT.NONE);

chap8Tree.setText("Chapter 8 Tree");

chap8Tree.setControl(new Ch8TreeComposite(tf));
```

接下来，例 8.2 展示的 Ch8ListComposite 使用了一些阅读器的高级特性。

例 8.2 Ch8ListComposite.java

```
package com.swtjface.Ch8;

import java.util.ArrayList;

import java.util.List;

import org.eclipse.jface.viewers.IStructuredContentProvider;

import org.eclipse.jface.viewers.LabelProvider;

import org.eclipse.jface.viewers.ListViewer;

import org.eclipse.jface.viewers.Viewer;

import org.eclipse.jface.viewers.ViewerFilter;

import org.eclipse.jface.viewers.ViewerSorter;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.FillLayout;
```

```
import org.eclipse.swt.widgets.Composite;
public class Ch8ListComposite extends Composite {
public Ch8ListComposite(Composite parent) {
super(parent, SWT.NULL);
populateControl();
}
protected void populateControl() {
FillLayout compositeLayout = new FillLayout();
setLayout(compositeLayout);
int[] styles = {SWT.SINGLE, SWT.MULTI}; 1 style
for(int style = 0; style < styles.length; style++) {
createListViewer(styles[style]);
}
}
private void createListViewer(int style) {
ListViewer viewer = new ListViewer(this, style);
viewer.setLabelProvider(new LabelProvider() {
public String getText(Object element) {
return ((ListItem)element).name;
}
});
viewer.addFilter(new ViewerFilter() { 2 Filter
public boolean select(Viewer viewer, Object parent,
Object element){
```

```
return ((ListItem)element).value % 2 == 0;
}
});
viewer.setSorter( new ViewerSorter() { 3 Sorter
public int compare(Viewer viewer, Object obj1, Object obj2){
return ((ListItem)obj2).value - ((ListItem)obj1).value;
}
});
viewer.setContentProvider(new IStructuredContentProvider() {
public Object[] getElements(Object inputElement) {
return ((List)inputElement).toArray();
}
public void dispose() {}
public void inputChanged(Viewer viewer, Object oldInput,
Object newInput) {}
});
List input = new ArrayList();
for( int i = 0; i < 20; i++ ) {
input.add(new ListItem("item " + i, i));
}
viewer.setInput(input);
}
}
class ListItem { 4 ListItem
```

```
public String name;

public int value;

public ListItem(String n, int v) {

name = n;

value = v;

}

}
```

1. 这两个是列表所仅有的。程序代码生成了其中之一。
2. 这个简单的 ViewerFilter 仅选择值域为偶数的条目。如果该值可被 2 整除，则返回 true，也即允许该条目可被显示。
3. 这个 ViewerSorter 按照域对象的值域由高到低进行排列。
4. 在本例中 ListItem 是作为域对象的。

这个代码生成一个类 ListItem 扮作域对象。ListItem 存储了一个名字和一个整数值，以用作排序和过滤。

因为一个 List 仅可处理元素间没有彼此关系的简单数据，IStructuredContentProvider 的应用仅要求在 getElements()中一个简单联系。为了弥补这个 content-provider 令人厌烦的应用，我们在此加入了一个 label-provider，一个过滤器和一个排序器。我们将按照其执行的次序逐个加以考虑。

在阅读器从 content-provider 获取了条目列表后，过滤器首先对这些条目进行了处理。由于本例的原因，我们决定仅显示那些值域是偶数的条目。因为调用对象上 toString()所生成的结果并不漂亮，你生成一个 label-provider 返回每一个 ListItem 的名字域。最终结果显示如图 8.5；将如下列加入到小部件窗口，编译、运行之：

```
TabItem chap8List  = new TabItem(tf, SWT.NONE);

chap8List.setText("Chapter 8 List");

chap8List.setControl(new Ch8ListComposite(tf));
```

图 8.5 列表窗口

当你考虑到所有这些操作都清楚地相互隔离则阅读器设计的优雅之处就明显了。Content-provider 并不介意它所提供的对象会被如何处理。过滤器之间并不需要彼此了解或者条目是如何排列的。Label-provider 显示对象而无需关注排序器是如何排列它们的。以上任一可以换为截然不同的应用而不会对程序代码的其余部分造成任何影响。

8.5 小结

理解不同的小部件之间的关系以及它们的阅读器是有效地使用这些 control 的关键。对于 control 的简单应用

可以无需借助阅读器，但是能够应用过滤器、排序器、label-provider 以及 content-provider 将会帮助你更为精练地分离对应用程序细节的关注。综上，可以确定你是否有必要去使用阅读器，并可以坚定自己的信念。如果将 TreeItem（或者其他类型的条目类）的直接生成方式和使用一个 content-provider 方式混同则会导致难以预期的结果并是你的程序代码晦涩难懂。

第九章 表和菜单_1

第九章表和菜单

本章涵盖内容

■ SWT 表

■ JFace 表

■ 编辑表数据

■ 生成菜单

如同我们每每出去吃饭，我们总发现我们自己会坐在车内，绞尽脑汁在考虑该上哪儿去。我们通常以不同食物类型的命名来收场：

“日本菜？”“不赖，但不是我想要的”

“意大利菜？”“不，不是今晚”

“印度菜？”“好主意，让我们再想想。。”

特别是我们在饥饿难耐是，我们对于就近的餐馆考虑再三并有了一个好的选择。

最终，我们有了如此的一个计划：在某一午后，当我们并非饥饿且有空闲思考时，我们写下了该片区域内的一个饭馆列表并按照价位或是食品类型加以组织。现在当我们决定外出时，看一眼这个列表我们就可以在讨论时有了个明确、形象的选择项。这个在我们对别的东西有喜好时可能无所助益，但是确实可以使得去哪儿的决策更容易些。

在一个软件应用程序中，一个菜单 menu 提供了和我们的饭馆列表相近的功效。

一个有限的选择项列表可以给用户作为他们确定从事哪项任务的导向。就像有的时候我们会发现一处我们从未造访过的却是上好的吃饭地方一样，用户通过看到下拉菜或是前后菜单找到之前它们并不知晓的你程序中的功能。

本章我们会涵盖两个任务。首先，我们将以最后一个基本阅读器小部件，表 table 来继续讨论有关上一章中的 Veiwer 框架。那些适用于树和列表的的概念同样适合于表，但是 JFace 也以单元编辑器形式提供了高级功能来使得用户可编辑表的应用更为容易。一旦你熟悉了编辑框架，我们将会回顾在第四章所学的 Actions 并展示任何来应用其生成菜单，由此你可以向你的用户展示你程序的功能而不是听任他们妄加猜测或是仅凭记忆来了解你的程序到底有何功用。最终，我们在本章中的样例将假以一个小的用户可编辑小部件展示如何来应用一个

表格的 context 菜单，即在一个关系数据库中编辑数据。

9.1 表 Tables

对于用户而言，一个表格看上去象一个由很多单元组成的二维网格。这也经常是一个诸如展示来自于一个数据库的逐行查询结果的便捷方法。以及你即将会看到的由 JFace 提供的编辑表数据的高级功能。

9.1.1 理解 SWT 表格

继续 SWT 习惯的直观命名方式，一个 table 即由一个名为 Table 的类来代表。其实 Table 类并不非常有趣。通常，如果你使用 JFace，则可以使用由一个 TableViewer 提供的接口而免除了和 Table 直接互动之苦。然而，如果你想直接操控当前选中的表格单元，又或者你并不在使用 JFace，则你将不得不使用隐藏在后面的 Table 了。

当看到 Table 的可得方法时，你将要注意的第一件事是：虽然有那么多的辅助方法来查询 Table 的情况，但是很清楚缺乏有效手段来定制化你的 Table。实际上，不同于直接向 Table 中加入数据或是列，一旦当依赖类被实例化，你将 pass 一个 Table 的实例给合适的依赖类；这一点类似于合成器被 pass 给其他小部件而不是将小部件加入合成器。不同于一些简洁的显示属性，如标题的可见性，关于应用 Table 时需要注意到的关键方法已在表格 9.1 中具体做了总结。表 9.1 Table 的重要方法

方法	描述
addSelectionListener()	当表的选择发生变化时通知你
select()/deselect()	以集中方式重载来让你程序化地加入或移除一个或多个单元的选择
getSelection()	获取一个当前选中单元的数组
remove()	从表中移除单元
showItem()/showSelection()	强制表格滚动到单元或选择可见

还有同样重要的是记得 Table 其实是扩展了 Scrollable 并且也因此会自动开启 scrollable 功能，除非你特地将其关闭。

TableItem

要向一个表格中加入数据，你必须使用一个 TableItem。每个 TableItem 的实例代表着在表格中的某一整行。每个 TabelItem 负责控制在其行中每个列单元内文本或是图案的显示。这些值可以通过 setText()和 setImage()方法来设定，这两个方法采用的一个整型参数即指明了需要修改的行。

如同我们提到的，TableItem 在它们的构造器中被绑定于一个 Table，即如下显示：

```
Table t = ...

//Create a new TableItem with the parent Table

//and a style

TableItem item = new TableItem(t, SWT.NONE);
```

```
item.setText(0, "Hello World!");

...
```

根据 Javadocs，对于一个 TableItem 没有风格 style 可供设定，但是构造器却可以接受一个风格参数。这看上去对我们而言无甚必要，但它却可以使 TableItem 和我们看到的其他小部件保持一致。

TableColumn

在和 table 直接互动时你需要接触的最后一个类就是 TabelColumn 了，它将生成一个表中的独立列。就像对待 TableItem 一样，你必须怕 pass 一个 Table 给 TableColumn 的构造器来使这两个对象绑定。

每个 TableColumn 的实例控制着表中的一个列。对你所需要的 TableColumn 进行实例化是必要的，否则默认情况下被设定为仅有一列。对于每个列的行为和外观进行控制的方法有多个，诸如宽度、文本排列方式，以及是否列可重定大小。你可以通过使用 setText()方法来加入标题文本。为避免在一个列上直接设定属性之累，我们选择使用一个 TableLayout 这种简单方法。通过调用 TableLyaout 的 addColumnData()方法，你可以轻松地描述表格每一个列的外观。传递 ColumnWeightData 的 addColumnData()实例的能力是关键的，这样做法可以令你为每一列指明其一个相对权重值而无需太多关注于每个列的实际象素点的多少。如下的程序片断展示了如何使用一个 TableLayout 来生成一个表。程序生成了三个等宽列并在两行内填充数据。程序产生的表格和图 9.1 类似。

```
//Set up the table layout

TableLayout layout = new TableLayout();

layout.addColumnData(new ColumnWeightData(33, 75, true));

layout.addColumnData(new ColumnWeightData(33, 75, true));

layout.addColumnData(new ColumnWeightData(33, 75, true));

Table table = new Table(parent, SWT.SINGLE);

table.setLayout(layout);

//Add columns to the table

TableColumn column1 = new TableColumn(table, SWT.CENTER);

TableColumn column2 = new TableColumn(table, SWT.CENTER);

TableColumn column3 = new TableColumn(table, SWT.CENTER);

TableItem item = new TableItem(table, SWT.NONE);

item.setText( new String[] { "column 1", "column 2", "column 3" } );
```

```
item = new TableItem(table, SWT.NONE);

item.setText( new String[] { "a", "b", "c" } );
```

要想做的第一件事是使用一个 TableLayout 来为该表设定结构。每次你调用方法 addColumnData()，它将向 table 加入一个新的列。我们将要有三个列，所以我们加入一个 ColumnWeightData 来描述每一列。在此我们在构造器中使用的参数有 weight，minimumWidth 和 resizeable。Weight 指代了该列在屏幕上的宽度值，且以整个表的可得空间的百分比形式出现。MinimumWidth 就如同其名字的意义一样，是用作该列的最小宽度象素值。而 resizeable 标记决定了是否用户可以重定该列的大小。

图 9.1 一个简单的三列表

在我们设定了 table 后，我们需要将该三个列进行实例化，这样它们才可以加入到表中。有一点很重要需要牢记的是添加列是一个两步操作的过程：生成一个 TableLayout 来描述每一个列将会有多大，然后生成该列自己。因为我们允许 TableLayout 控制尺寸，则在它们生成之后我们就不需要使用列了。

9.1.2 Jface 的 TableViewers

虽然在你的程序代码中直接使用一个 Table 也是可能的，但就像你看到的那样，这样做既不直观也不简便。和表 List 相类似，JFace 也提供了一个阅读器 viewer 类来简化 table 的应用。如下程序片断演示了一个基本的 TableViewer 是如何来显示来自于一个数据库中的数据的。这期间也有着我们在第八章中讨论的一模一样的 filters，sorters 和 label-provider。另外，因为在前一章中的参数在此同样应用，我们将使用一个 ContentProvider 来向我们的表提供数据。

首先，表必须被设定。这个和你在之前一节中见到的设定一个 Table 相类似，即为每一个即将生成的列使用一个 addColumnData()：

```
final TableViewer viewer = new TableViewer(parent, SWT.BORDER |

SWT.FULL_SELECTION);

//configure the table for display

TableLayout layout  = new TableLayout();

layout.addColumnData(new ColumnWeightData(33, true));

layout.addColumnData(new ColumnWeightData(33, true));

layout.addColumnData(new ColumnWeightData(33, true));

viewer.getTable().setLayout(layout);
```

```
viewer.getTable().setLinesVisible(true);

viewer.getTable().setHeaderVisible(true);
```

一旦表被设置，我们将添加卡动的 provider。在本例中最重要的一个是 content-provider，他是负责着从数据库中获取数据并 pass 回给阅读器。注意到你没有从 getElements()返回空——而是在确实没有子单元时，返回一个空的数组：

```
viewer.setContentProvider(new IStructuredContentProvider() {

public Object[] getElements(Object input) {

//Cast input appropriately and perform a database query

...

while( results.next() ) {

//read results from database

}

if(resultCollection.size() > 0) {

return new DBRow[] { ... };

}

else {

return new Object[0];

}

}

//... additional interface methods

});

viewer.setLabelProvider(new ITableLabelProvider() {

public String getColumnText(Object element, int index) {

DBRow row = (DBRow)element;
```

```
switch(index) {

//return appropriate attribute for column

}

}

//... additional interface methods

});
```

一旦 provider 们被设定，我们就能加入列了。在表被显示时，在每个列上我们设定的文本将被作为列标题而显示：

```
TableColumn column1 = new TableColumn(viewer.getTable(), SWT.CENTER);

column1.setText("Primary Key");

TableColumn column2 = new TableColumn(viewer.getTable(), SWT.CENTER);

column2.setText("Foreign Key");

TableColumn column3 = new TableColumn(viewer.getTable(), SWT.CENTER);

column3.setText("Data");
```

最终，我们需要提供一个输入来驱动 content-provider。输入对象（在本例中，是一个字符串描述的一个查询）在 viewer 上被设定，并且当被显示到表上是被 pass 给 content-provider：

```
viewer.setInput(QUERY);
```

本例模拟了从数据库中获取多行数据并显示结果。然而，它足以支持我们关于 content-provider 的观点了。有关 IStructuredContentProvider 应用的角色是简洁明了的：给定一个输入元素，返回所有要被显示的子元素。一个表并不维持父/子关系，所以这个方法仅调用一次且只给予当前的输入对象。当在使用一个 content-provider 时需要保持清醒的问题是它总是在 UI 的线程内执行的。这也意味着界面的更新必须等到你的方法执行完毕，所以你当然不应该通过查询一个数据库来得到你的更新。Content-provider 应该遍历已装载的域对象找出其中一幅图画选择以作为合适的显示内容。

关于错误处理的一段话

当你在使用 JFace 时——特别是在使用由小部件在内部调用的 provider 时候需要对有关的错误处理特别谨慎对待。当 JFace 在回调你的类时，若遇到所有的例外情况 exceptions，典型做法是在一个 try/catch 的阻断中。

Jface 在将该例外进行对外传播之前会作一些检查看看它自己是否知道该处理这个例外。然而不幸的是这些检查都是依赖于平台 Platform 的类，这些类都是紧密结合于 Eclipse 的；而在实践中在未运行 Eclipse 情况虾，

几乎不大可能对平台进行正确初始化。当 JFace 在 Eclipse 之外使用平台时，就导致了内部声明的失败，而这些例外将以标记为你自己类的失败而告终。

在实际情况中，你不应该让一个例外抛出预给定的方法之外。如果发生这种情况，你将会接收到这样子奇怪的消息：“The application has not been initialized.”所以一旦你看到这些，请仔细检查你代码中诸如 ClassCastException 之类难以侦测的地方，而且如果 JFace 隐藏在后台运行则就更加难以定位了。

编辑表数据 table data

就小部件自己而言，展示数据当然是一个有用的功能，但是事实上有时你并不想让用户来编辑它。经常，启用编辑功能的最友好方式是允许用户直接改变在表格上显示的内容。JFace 则通过 CellEditors 来提供这一编辑功能的支持。

如同我们在本章概述内容中提到的那样，CellEditors 之存在是为了帮助域模型从编辑过程中解放出来。另外，使用这些编辑器可以使你的 UI 更加友好：用户不会输入你的程序难以理解的值，这样就可以避免错误信息扰人耳目。该框架假定了每个域对象都有一个命名属性的数字。通常，你应该遵从 JavaBeans 的规则，对于属性 foo 应该应用 getFoo()和 setFoo()发拿过发；但是由于你可以仅通过其名字来分辨是哪个属性，所以这样作并非严格必须。你将开始以向你的 TableViewer 添加一个 ICellModifier 实例。ICellModifier 负责接受一个对象的给定属性值，确定一个属性当前是否应该被编辑，并在编辑完成后将该更新值应用到对象上。如果允许真实的编辑将由一个 CellEditor 来执行。JFace 通过复选框、组合框、弹出对话框、或是直接键入新文本值来提供 CellEditor 给实施编辑。另外，如果你需要一种新形式的编辑器，还可以子类化 CellEditor 获得。在注册了 CellEditor 后你将一个属性绑定于每一列。当用户点击某一单元格来改变其值，JFace 就动用其匹配正确列的魔力来编辑和显示正确的编辑器，然后等编辑完成通知你的 ICellModifier。

我们将通过样例展示这一过程中的重要部分。本节中其余的程序片断都采自于 Ch9TableEditorComposite，这个类的代码在本章末全文展示。

第一个程序片断设定了代码其余部分需要引用的数据。在 VALUE_SET 内的字符串数组持有将要被我们的 ComboBoxCellEditor 显示的值。我们将要在索引和值之间进行多次转换（详见本章后面的讨论）：

```
private static final Object[] CONTENT = new Object[] {

new EditableTableItem("item 1", new Integer(0)),

new EditableTableItem("item 2", new Integer(1))

};

private static final String[] VALUE_SET = new String[] {

"xxx", "yyy", "zzz"

};

private static final String NAME_PROPERTY = "name";
```

```
private static final String VALUE_PROPERTY = "value";
```

我们的类包含这几个不同的方法，每一个都负责设定单元编辑器的不同方面。它们从 buildControls 依次调用。这个方法第一件所作的事就是设定阅读器要求的表格和类：

```
protected Control buildControls() {

final Table table = new Table(parent, SWT.FULL_SELECTION);

TableViewer viewer = new TableViewer(table);

... //set up a two column table
```

一旦表格被初始化，我们继续向我们的阅读器添加 ITableLableProvider 的一个实例。这个主意和我们在第八章内讨论的 label-provider 类似。然而，因为一个表的每一行都有许多列，则我们的方法的标记需要略有变化。除了元素外，每一个方法现在都采取了一个要求的列的整数型的索引。Label-provider 因此必须要包含一个列索引和域对象之间对应逻辑。接下来的程序片断展示了这是如何做的：

```
viewer.setLabelProvider(new ITableLabelProvider() {

public String getColumnText(Object element, int columnIndex) {

switch(columnIndex) {

case 0:

return ((EditableTableItem)element).name;

case 1:

Number index = ((EditableTableItem)element).value;

return VALUE_SET[index.intValue()];

default:

return "Invalid column: " + columnIndex;

}

}

});

attachCellEditors(viewer, table);
```

```
return table;

}
```

attachCellEditores()方法是我们设定 ICellModifier 的地方，它负责将一个属性名字翻译为可显示的数据，确定一个给定的属性是否能被编辑，然后将用户所作的无论如何变化付诸应用。当用户双击一个单元格要编辑它，canModify()被调用来确定该编辑是否被允许。如果被允许，接下来就调用 getValue()来获取被编辑的属性之当前值。一旦编辑完成，modify()就被调用；modify()负责着将用户所作的变化应用到原始的域对象。在 getValue()和 canModify()这些方法内将参数直接作用到于域对象是安全的，但是对于方法 modify()来说则不然。modify()接收来自行中显示的 TableItem。这个 TableItem 用域对象来设定为其数据，所以在我们更新它之前必须要使用 getData()来获取它：

```
private void attachCellEditors(final TableViewer viewer, Composite parent) {

viewer.setCellModifier(new ICellModifier() {

public boolean canModify(Object element, String property) {

return true;

}

public Object getValue(Object element, String property) {

if( NAME_PROPERTY.equals(property))

return ((EditableTableItem)element).name;

else

return ((EditableTableItem)element).value;

}

//method continues below...
```

当 modify()完成对域对象的更新，我们必须阅读器知道并去更新 display。阅读器的 refresh()方法就是用以这个目的。调用带有改变的域对象参数的 refresh()会导致阅读器重画给定的行。如果我们跳过这一步，则一旦被编辑的单元格不再选中激活用户就永远不会观察到它们的变化：

```
public void modify(Object element, String property, Object value) {

TableItem tableItem = (TableItem)element;

EditableTableItem data = (EditableTableItem)tableItem.getData();
```

```
if( NAME_PROPERTY.equals( property ) )
data.name = value.toString();
else
data.value = (Integer)value;
viewer.refresh(data);
}
});
```

在 cellEditor 数组中给出的项目将和后台表中的列按顺序相匹配：

```
viewer.setCellEditors(new CellEditor[] { new TextCellEditor(parent),
new ComboBoxCellEditor(parent, VALUE_SET )
});
```

接下来，在 setColumnProperties()中的字符串就是在我们的域对象上可编辑属性的名称了。它们也同样和表的列按顺序匹配，所以在我们的样例中点击列 0 将会视图编辑名字属性，列 1 则会编辑值属性：

```
viewer.setColumnProperties(new String[] {
NAME_PROPERTY, VALUE_PROPERTY
});
}
}
class EditableTableItem {
... //name and value properties
}
```

象我们在此使用一个 ComboBoxCellEditor 是需要技巧的。该编辑器的构造器会采纳一个字符串数组来作为供用户选择的值。然而，当编辑结束编辑器会等待从 getValue()返回的整数并返回一个整数给 modify()。这

些值应该和 pass 给 ComboBoxCellEditor 构造器的数组中选定值之索引相对应。在这个简单例子中，我们将整数直接保存在值域内，但是在实际的应用程序中你可能需要在索引和值之间轻松实现来回转换的工具。

再说一遍，使用 CellEditor 是一个极为有趣且你需要关注于你的方法应用和错误处理的领域。特别是当不同的方法应用到不同的对象上时，如同在 ICellModifier 中，很容易编译出错。鉴于 JFace 的例外处理方式，就像我们之前提及的，这些问题同样会引起关于“Application not initialized”的含糊其词的运行时错误提示，而你又不知道如何查找，所以要追溯很困难。

第九章 表和菜单_3

9.2 生成菜单 menus

每一个图形化程序都会应用到菜单之类的东西。如果你经常在你应用程序顶部作诸如查找文件、编辑之类的动作。这些菜单就扮演着一个重要的角色，因为它们提供了一个用户可以浏览你的应用程序所含有的功能的处所。

我们首先要讨论利用 SWT 来生成菜单。然后再会涉及到我们在第四章中提及的 JFace 的 Action 类，通过它我们可以找到一条允许共享通用代码来生成菜单的备选途径。

9.2.1 加速键

在我们还为太深入到菜单的特定细节之前，让我们先讨论一下 SWT 是如何处理加速键的。所谓的加速键是指无需用户用鼠标去点击而直接由键盘上的快捷方式来激活某一小部件的方式。一个普遍存在的绝好例子就是 Ctrl-C

（若你在使用苹果机，则或者是打开 Apple-C）将文本拷贝到粘贴板，这一功能和绝大多数应用程序中的你选定编辑菜单中的 Copy 功能一模一样。

对于常规任务提供快捷键可以极大地提高资深用户的工作效率，因为他们的手无需在键盘和鼠标间频繁转换。对于某一项目的快捷键的键击方式通常会出现在紧邻于应用程序下拉菜单的该项目名字边上，这样可以很明晰地提示用户来学习如何联系操作这一快捷键。在 SWT 和 JFace 中，快捷键是通过来自于 SWT 类中的常量来表述的。这个概念和风格 sytle 如出一辙：所有的常量都二进制化来确定最终键的组合。另外，字符被用以代表键盘上的字母和数字。因为一个 Java 的字符会自动转化为一个整型，字符就可以如 SWT 的 style 一样用来构建一个位图。这个位图被传递给在一个 Menu 上的 setAccelerator()方法来注册将来会该菜单项目的键组合。例如，一个 MenuItem 的快捷键若被设定为 SWT.CONTROL | SWT.SHIFT | 't'，则同时按下 Ctrl、Shift 和 T 键就能激活它了。

9.2.2 在 SWT 中生成菜单

当你在使用 SWT 生成菜单时，你一般仅用到两个类：Menu 和 MenuItem。虽然这两个类本身并不复杂，但是一旦你启用它们马上有几个复杂问题就会浮现。

Menu 可视作是 MenuItem 的容器。Menu 扩展了 Widget 并包含了一组添加 MenuItem 和控制其可见性以及在菜单中位置的方法。Menu 也广播事件来应用 MenuListener 接口，该接口可以接收菜单显示和隐藏是的通知。

Menu 支持三种不同的 style，它们控制着视觉外观以及菜单的生成类型：

■ SWT.POP_UP— 当你在一个应用程序中右击鼠标时弹出一个自由浮动类型菜单

■ SWT.BAR— 在一个应用程序窗口顶部生成一个菜单工具条。一个菜单工具条并没有典型意义的可选择菜单单元；而是，它是作为一个包含着类型 SWT.DROP_DOWN 的菜单容器。

■ SWT.DROP_DOWN— 生成文件、编辑以及其他我们熟悉的下拉式菜单。这些菜单可能包含着一个 MenuItem 和 submenu 混合体。

一个 MenuItem 是一个既可以被最终用户选择的小部件也可一被显示作另一个菜单。一个 MenuItem 总是作为一个 Menu 的子部件来生成的。对于 MenuItem 来说可以使用一系列的风格：

■ SWT.PUSH— 生成一个标准的不带虚饰的菜单单元

■ SWT.CHECK, SWT.RADIO— 加入一个复选框或是收音机按钮，正常情况下，每次当项目选中在会在开和关间翻转。

■ SWT.SEPARATOR—对 menu item 进行视觉上的分割。它会展现为你平台上的一个标准分割符（通常是一根细直线）并且不能被用户选中。

■ SWT.CASCADE—生成一个子菜单。当一个层叠 menu item 被分配给某一个 menu，则当高亮某一 item，submenu 就会展现

所有的 MenuItem 除了分隔符外都广播可以被侦听到的 SelectionEvent 事件。图 9.2 显示了不同的 menu 风格。

图 9.2 Menu 类型。由顶及底分别是 SWT.CHECK、SWT.CASCADE、SWT.PUSH 和 SWT.RADIO。

生成 Menu 是简洁的。类被实例化并作恰当设置即可，然后分配给小部件们作准备的显示。如下的程序片断显示了如何生成一个 File 菜单并添加到你的应用程序主窗口：

```
Composite parent = ... //get parent

Menu menuBar = new Menu(parent.getShell(), SWT.BAR);

MenuItem fileItem = new MenuItem(menuBar, SWT.CASCADE);

fileItem.setText("&File");

Menu fileMenu = new Menu(fileItem);

fileItem.setMenu(fileMenu);

parent.getShell().setMenuBar(menuBar);
```

注意到你必须首先生成一个根菜单栏然后加入一个猜度那项目到各个下拉菜单以备将来在其上显示。在这时，

我们已经有了一个显示为 File 的菜单栏，但是却是空的。我们下一步任务将是在其上植入：

```
MenuItem open = new MenuItem(fileMenu, SWT.PUSH);

open.setText("Open...");

open.setAccelerator(SWT.CONTROL | 'o');

open.addSelectionListener(new SelectionListener() {

public void widgetSelected(SelectionEvent event) {

... //handle selection

}

};
```

点击 File 将会出现一个带有 Open 项的下拉菜单。如果 Open 被选中，则我们之前定义的选中监听器 selection listener 将会调用并显示一个打开文件对话框或是其他的对于本应用程序而言是恰当的动作。我们已经通过调用方法 setAccelerator()以及带有我们想要分配键的位遮罩参数来将 Ctrl-O 作为快捷键设定给该项。结果就是按下 Ctrl-O 就会调用 selection listener，这一点和用鼠标的点选无异。

生成一个弹出菜单和我们已经在做的相似，但是会费一些小周章。我们其实并不需要一个菜单栏，所以我们就开始以该如下的弹出方式 pop-up：

```
Composite parent = ... //get composite

final Menu popupMenu = new Menu(parent.getShell(), SWT.POP_UP);
```

注意到我们已经声明了一个 Menu 实例是 final 的了。这是重要的，因为我们需要在后面的一个监听器中引用到它。

生成 MenuItem 和一个下拉菜单一样。有所不同的是，我们将要展现如何来生成一个 menu item 使其在被高亮时能释放出一个子菜单。在此过程中的须注意的重要点是在子菜单生成后，它必须使用 setMenu()来将其分配给父 menu，就像我们早先在处理 menu bar 时一样：

```
MenuItem menuItem = new MenuItem(popupMenu, SWT.CASCADE);

menuItem.setText("More options");

Menu subMenu = new Menu(menuItem);

menuItem.setMenu(subMenu);

MenuItem subItem = new MenuItem(subMenu, SWT.PUSH);
```

```
subItem.setText("Option 1");

subItem.addSelectionListener( ... );
```

不同于一个菜单栏，一个弹出菜单默认情况下是不显示的——所以你必须确定何时该显示它。典型作法是将其作为鼠标键右击的反应，所以我们在父合成器上使用一个 MouseListener。这就是我们需要弹出菜单实例 final 化的地方，而我们将在我们的匿名内部类中引用到它：

```
parent.addMouseListener(new MouseListener() {

public void mouseDown(MouseEvent event) {

if(event.button == 2) {

popupMenu.setVisible(true);

}

}

... //other MouseListener methods

});
```

MouseEvent 包含有关于被点击的鼠标键信息。且按键并编号为：1 为鼠标左键；2 为鼠标右键。如果 2 键被按下，我们就使弹出菜单可见；且它所显示的位置就在刚才点击的地方。按下 Esc 键或是在菜单之外的任意地方再行点击鼠标则自动会隐藏该弹出菜单。

现在你已看到 SWT 是如何处理菜单的了，现在我们将注意力转向由 JFace 提供的菜单功能项。

9.2.3 使用 JFace 的 actions 来加入菜单

我们已经在第四章中讨论了 JFace 的 Action 的设计。为了做简要回顾，一个 action 封装了一个单一应用程序层面事件的反应，诸如“Open a file”或是“Update the status bar.” 而这个 action 可以被在不同的环境中重用和激发，诸如一个工具条按钮或是一个菜单项。我们在此将讨论这最后一个案例。通过使用 action 来生成你的菜单，而不是手工去做，你可以简化你的程序设计并重用通用逻辑。

在一个 menu 中使用 action 和在其他任意地方的使用是相似的。记得一个 IContributionsManager 是负责着组装单个的 Action 然后将它们转形成为可以提供给用户以显示的一种形式。对于菜单而言，我们将使用 IContributionManager 的 MenuManager 应用。在向 MenuManager 加入了所需的 action 后，我们可以要求它来生成一个新的 menu 或是向其他的 menu 添加 action 了。程序代码看上去就像下面的：

```
Shell shell = ... //obtain a reference to the Shell

MenuManager fileMenuManager = new MenuManager("File");
```

```
IAction openAction = new OpenAction(...);

... //create other actions as appropriate

fileMenuManager.add(openAction);

... //add other actions

Menu menuBar = new Menu(shell, SWT.BAR);

fileMenuManager.fill(menuBar, -1);

shell.setMenuBar(menuBar);
```

虽然，我们仍旧在手工地生成菜单栏，我们可以向 manager 加入 action 了并且让它来关注 menu 是如何构建的。在本例中，我们将以向窗口的菜单栏添加一个 File 菜单来结束，因为这是我们在实例化 MenuManager 时所给予的名字。这样做而不是手工来构建菜单的好处是 action 类可以轻松地在其他地方得以重用。例如，如果我们有一个包括着一个按钮的工具条并可以让用户来打开文件，则我们就可以同样地应用一个名为 OpenAction 的类。

若你在使用 menu manager 时，你必须在脑子中保持警醒：一旦在一个给定的实例上调用了 fill()或是 createXXX()，则 Menu 和 MenuItem 的实例就会在内部生成并进入缓存。这样做时必要的因为 manager 就可以对 menu 实施必要的更新了。然而，这也意味着你将不能在使用 fill()和 create()方法了，特别是对于不同类型的 menu。例如假设在之前的程序代码后我们调用了在 fileMenuManager 上的 createContextMenu()。则当我们视图向一个合成器添加菜单时就会得到例外 exception，因为如果使用了类型 SWT.CASCADE 而不是 SWT.POP_UP（这个时 context menu 所要求的），menu 就会成为进入缓存的实例

9.3 更新小部件窗口

我们在本章的小窗口组合了一个 table-viewer，cell-editor，和一个 context-menu。我们将扩展早先我们讨论的一个数据库编辑器的程序代码片断并加入一个右键菜单来让用户添加一个新的行。最终的产品如图 9.3。

图 9.3 我们的数据库表编辑器

例 9.1 比我们其他章节的小窗口程序代码都要长，所以在通篇阅读之前，我们先要指出最为精彩之处。第一个需要注意的是 NewRowAction 内部类。该类持有一个向表插入一个新行的逻辑；它将被添加到我们在 createPane()中生成的 MenuManager 上。

接下来则是 createPane()方法了，它是类的入口点。在放置了一个表后，添加了一个 label-provider、content-provider 和 cell-editor，我们实例化了一个 MenuManager 并用其来构建了一个 context-munu 然后将其添加到新生成的 Table 中去。最终，我们将初始化的 content 传递给阅读器 vierwer。

在 createPane()后上场的是私有工具方法 private utility method。我们最重要的须关注的方法应该是

attachCellEditors()，它包含着允许编辑独立表单元的逻辑。注意到这些修改都是直接作用于域对象之上的。

在例末是 EditableTableItem 类，它在本例中用作域对象并且方便期间包含于相同的文件之中。

例 9.1 Ch9TableEditorComposite.java

```
package com.swtjface.Ch9;

import org.eclipse.jface.action.*;

import org.eclipse.jface.viewers.*;

import org.eclipse.swt.SWT;

import org.eclipse.swt.graphics.Image;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.*;

public class Ch9TableEditorComposite extends Composite {

private static final Object[] CONTENT = new Object[] { 1 Initial content

new EditableTableItem("item 1", new Integer(0)),

new EditableTableItem("item 2", new Integer(1))

};

private static final String[] VALUE_SET = new String[] {

"xxx", "yyy", "zzz"

};

private static final String NAME_PROPERTY = "name";

private static final String VALUE_PROPERTY = "value";

private TableViewer viewer;

public Ch9TableEditorComposite(Composite parent) {

super(parent, SWT.NULL);

buildControls();
```

```
}
private class NewRowAction extends Action { 2 NewRowAction class
public NewRowAction() {
super("Insert New Row");
}
public void run() { 3 run() method
EditableTableItem newItem =
new EditableTableItem("new row", new Integer(2));
viewer.add(newItem);
}
}
protected void buildControls() {
FillLayout compositeLayout = new FillLayout();
setLayout(compositeLayout);
final Table table = new Table(this, SWT.FULL_SELECTION);
viewer = buildAndLayoutTable(table);
attachContentProvider(viewer);
attachLabelProvider(viewer);
attachCellEditors(viewer, table);
MenuManager popupMenu = new MenuManager(); 4 Build menu
IAction newRowAction = new NewRowAction();
popupMenu.add(newRowAction);
Menu menu = popupMenu.createContextMenu(table);
table.setMenu(menu);
```

```
viewer.setInput(CONTENT);

}

private void attachLabelProvider(TableViewer viewer) {

viewer.setLabelProvider(new ITableLabelProvider() {

public Image getColumnImage(Object element, int columnIndex) {

return null; 5 getColumnText() method

}

public String getColumnText(Object element, int columnIndex) {

switch(columnIndex) {

case 0:

return ((EditableTableItem)element).name;

case 1:

Number index =

((EditableTableItem)element).value;

return VALUE_SET[index.intValue()];

default:

return "Invalid column: " + columnIndex;

}

}

public void addListener(ILabelProviderListener listener) {

}

public void dispose(){

}

public boolean isLabelProperty(Object element, String property){
```

```
return false;
}
public void removeListener(ILabelProviderListener lpl) {
}
});
}
private void attachContentProvider(TableViewer viewer) {
viewer.setContentProvider(new IStructuredContentProvider() {
public Object[] getElements(Object inputElement) {
return (Object[])inputElement; 6 getElements() method
}
public void dispose() {
}
public void inputChanged(Viewer viewer, Object oldInput,
Object newInput) {
}
});
} 7 buildAndLayoutTable()
private TableViewer buildAndLayoutTable(final Table table) {
TableViewer tableViewer = new TableViewer(table);
TableLayout layout = new TableLayout();
layout.addColumnData(new ColumnWeightData(50, 75, true));
layout.addColumnData(new ColumnWeightData(50, 75, true));
table.setLayout(layout);
```

```
TableColumn nameColumn = new TableColumn(table, SWT.CENTER);
nameColumn.setText("Name");
TableColumn valColumn = new TableColumn(table, SWT.CENTER);
valColumn.setText("Value");
table.setHeaderVisible(true);
return tableViewer;
}
private void attachCellEditors(final TableViewer viewer, Composite parent) {
viewer.setCellModifier(new ICellModifier() {
public boolean canModify(Object element, String property){
return true;
}
public Object getValue(Object element, String property) {
if( NAME_PROPERTY.equals(property))
return ((EditableTableItem)element).name;
else
return ((EditableTableItem)element).value;
} 8 modify() method
public void modify(Object element,String property,Object value) {
TableItem tableItem = (TableItem)element;
EditableTableItem data =
(EditableTableItem)tableItem.getData();
if( NAME_PROPERTY.equals( property ) )
data.name = value.toString();
```

```
else
data.value = (Integer)value;
viewer.refresh(data);
}
});
viewer.setCellEditors(new CellEditor[] {new TextCellEditor(parent),
new ComboBoxCellEditor(parent, VALUE_SET )
});
viewer.setColumnProperties(new String[] {
NAME_PROPERTY, VALUE_PROPERTY
});
}
}
class EditableTableItem { 9 EditableTableItem class
public String name;
public Integer value;
public EditableTableItem( String n, Integer v) {
name = n;
value = v;
}
}
```

1. 这些常量所持有的数据将会为我们的初始化 content 所用。在实际的应用程序中，这些数据可能取自于数据库或是其他的外部源。
2. 这个类含有向数据集插入新行的逻辑，它扩展了 Action，所以 MenuManager 才可用。
3. 为了执行需要的逻辑，我们覆盖了在 Action 中定义的 run()函数。Action 框架可以确保该方法在适当的时候被调用。我们的应用生成一个新的域对象并调用在 table-viewer 上的 add()。绝大部分真

实应用程序中会需要额外的逻辑来管理域对象集合。

4. 我们通过生成一个新的 MenuManager 来构建一个简单的 context-menu 并且添加了我们想使用的 action。在本例中我们直接向 Table 添加了 menu。如果如果该 tab 内存在有不止该 table 的 control，则该菜单仅会在 table 范围内用户右击鼠标才会出现。如果你想使用户的 tab 内其他任意区域的右击同样生效，则你需要将 menu 加入到父合成器中。
5. 这是一个标准的 LabelProvider 应用，类似于你早先看到的。返回同要求的列相匹配的任一属性值。
6. 我们的 content-provider 假定无论何输入皆给出一个对象数组。执行一个恰当的方法应用并返回结果。
7. 在此我们构造了表格。我们加入两个列并设定了标题文本。
8. modify()方法是我们的 CellModifier 应用中最为重要的部分。要素参数包含了对于刚发生改变的 cell 的 TableItem。被帮定于该 item 的域对象可用方法 getData()获取。当我们在检查 propertyName 参数来确定什么属性被修改时，我们使用值参数来更新域对象上匹配的属性，而这包含着用户键入的数据。
9. 这个类在本例中用作为于域对象。

通过加入如下列到小部件窗口可以运行该例：

```
TabItem chap9TableEditor = new TabItem(tf, SWT.NONE);

chap9TableEditor.setText("Chapter 9");

chap9TableEditor.setControl(new Ch9TableEditorComposite(tf));
```

当你载运行本例时，初始的窗口内包含着带有样例数据的两个行。右击带来一个菜单允许你在表中插入新行。双击一个单元可以编辑数据，并可采用键盘键入或是下拉菜单选择。

9.4 Summary

在 table 和 TableViewer 中你所见到的绝大部分应该和第八章中内容类似。Viewer 和 provider 的基本概念和我们早先讨论的是一致的。因为表格驱动着一个二维结构的数据，它们需要比其他我们检验过的小部件更多的设置工作。TableLayout 和 TableColumn 类为每个表生成这种结构并控制着如何向用户展现表的细节。

经过这两章，你应该有能力处理这些 viewer 或是在 JFace 中的 TableTreeViewer 之类深奥的类的要求了。

而 cellEditor 对于 TableViewer 来说是有用的特性。CellEditor 提供了处理表中特定单元更新的一个框架，预定义的 CellEditor 类提供了一条给用户作选择离散项的便捷之路。

关于任意的应用程序都会用到的菜单栏，通常也会要仅显示和用户当前操作相对应的功能项。例如，在一个子处理器中右击鼠标很自然会带来一个与文本格式化相关的菜单项。SWT 使得生成这些菜单容易而 JFace 则通过加入 action 框架来促使无论何种情况下逻辑的复用。我们已经在第四章讨论了隐藏在 action 之后的理论，而我们给出的样例应该使你对实践中如何应用有了良好的感觉。

第十章 对话框_1

第十章对话框

本章涵盖内容

■ SWT 内涵的对话框

■ JFace 内涵的对话框

■ 进度指示器

■ 在 JFace 中定制对话框

如果你想被人关注，那么你必须找到一条你自己引起关注的途径。无论你的目标是俘获一颗爱心，抑或是推广一项新发明，不付诸努力是不可能取得成功的。这条准则同样适用与软件业。如果你的程序要从用户这里得到点什么，则它必须要找到可吸引用户眼球的东西。条条大路通罗马，但最常用的要算是提供给客户以对话框了。一个对话框是独立于你的应用程序主窗口的窗口。当这个窗口被置于应用程序之前，用户被强迫去注意你在此展示的什么东西，无论它是一个状态信息或是一个输入的请求，诸如一个要保存的文件名。因为展示一个对话框是一个如此通常和必要的任务，SWT 和 JFace 提供了轻松处理对话框的支持。

我们需要讨论两个独立的对话框类：org.eclipse.swt.widgets.Dialog 和 org.eclipse.jface.dialogs.Dialog。如同它们的包名字指示的那样，第一个类是 SWT 部分，而第二个则来自于 JFace。当然在构建一个应用程序时进使用 SWT 对话框也是可能的，但是这样做可并不有趣。好在，SWT 提供了若干个预建的对话框来执行一些特定的任务。我们将首先讲述使用 SWT 对话框，后续有一个对比使用 JFace 对话框。本章我们将在本章生成一个用户名/密码提示对话框来演示如何写一个定制对话框。

10.1 SWT 对话框

抽象类 Dialog 提供了 SWT 内所有对话框的基础。而作为它自己，这个类根本就是无趣的，因为它并提供更多地有关行为方面东西。如你愿意将 SWT 的 Dialog 子类化也是可能的。然而，将 JFace 提供的 Dialog 子类化可以使你的工作更加简化。因此，我们将在本章内 JFace 章节内讨论如何生成定制化的对话框。SWT 为通用任务提供了便于和易于预制的对话框。我们那将讨论如何和何时在解下来的章节中使用这些。

10.1.1 颜色对话框

假定你正在编写一个文本编辑器，而且你想让用户来指定在屏幕上文本的颜色。如果你的目标受众是由编程高手构成，或许可能让他们来手工指明 RGB 值，但是常规用户只想看到可用的功能项或是只需一个点击选中动作即可。SWT 就是通过其类 ColorDialog 来帮你达成这一切。

由于考虑到本书的灰度级的特质无法来欣赏彩色图案，但是图 10.1 展示了用户是如何能够轻松地从一个 ColorDialog 来选择一个颜色。

图 10.1 ColorDialog 允许用户从可得的颜色中进行选取

展示这个对话框很简单。只需实例化它，然后调用 open()即可：

```
ColorDialog dialog = new ColorDialog(shell);
```

```
RGB color = dialog.open();
```

如果你想要预先选择一个颜色，则在打开对话框之前调用 setRGB()。

调用 open()阻断，意味着在用户点击 OK 或是 Cancel 之前不会返回任何值。如果有颜色被选中，则将从 open()返回一个 RGB 的实例。如果用户点击了 Cancel 则返回 null。该被选中的颜色在之后还可使用 getRGB()来获取。

10.1.2 目录对话框 DirectoryDialog

DirectoryDialog 将选择一个目标目录——例如，要想保存一组文件的位置。如下代码片断展示了该目录选择对话框，且附有解释用户被认为将要作甚么的信息：

```
DirectoryDialog dialog = new DirectoryDialog(shell);

dialog.setMessage("Choose a save directory");

String saveTarget = dialog.open();

if(saveTarget != null) {

java.io.File directory = new java.io.File(saveTarget);

...

}
```

该对话框看上去就像是图 10.2。

图 10.2 DirectoryDialog 等待着用户选择一个目录

一旦用户选中了一个目录，调用 open()方法则返回一个表征着所选定的目录绝对路径的字符串。该字符串可用作生成一个 java.io.File 并且能据此根植。而如果用户点选 Cancel，则 open()返回 null。

10.1.3 文件对话框 FileDialog

FileDialog 在诸多方面和 DirectoryDialog 有着相似之处。首先，除了 DirectoryDialog 展示的是一串目录列表外，对话框外观几乎相同；FileDialog 会在当前显示的目录内加入一个文件，就如同图 10.3 显示的那样。

图 10.3 FileDialog 在保存模式下允许用户输入一个文件名来保存

接下来的程序片断展示了如何让用户来进行文件的多选操作，条件是只要文件名后缀是 txt：

```
FileDialog dialog = new FileDialog(shell, SWT.MULTI);

dialog.setFilterExtensions(new String[] {"*.txt"});
```

```
dialog.open();
```

通常，open()阻断并返回给文件一个要么是全路径的字符串，要么是 null。不同于你之前见到的对话框，FileDialog 支持如下三种不同的风格 style：

■ SWT.SAVE—将对话框作为一个保存对话框对待，允许用户要们选择一个现存的文件，要么输入一个新文件名来生成。

■ SWT.OPEN—让用户选择一个单一的现存文件来打开。

■ SWT.MULTI—让用户一次性地打开多个文件。即使用户在一个 MULTI-style 的对话框内选择了多个文件，事实上 open()仅返回一个。在 open()返回值之后 getFileNames()必须用来获取包含选中的所有文件名的一个数组。

对于 FileDialog 内还有需要关注的是方法 setFilterPath()、setFilterNames()和 setFilterExtention()。当它们在对话框打开之前被调用，这些方法将会用来限定显示给用户看的文件列表。SetFilterPath()接受一个单一的字符串将其用作默认应该显示的目录的路径。余下来的两个方法接受的则是字符串数组，这些数组用来组装成有效的文件名和后缀。我们前面一个样例就是过滤掉了除了后缀以 txt 的其他任何东西。注意的是展示给用户看的过滤器其实是可编辑的，因此不能保证它们会如你所愿地选出一个文件或是类型。

10.1.4 字体对话框 FontDialog

就如同你的文本编辑器的用户会要给其文本选定一种颜色一样，他们也会选择一种字体来显示。处于这个目的，我们有了 FontDialog。它的使用几乎和我们之前讨论的 ColorDialog 一模一样：

```
FontDialog dialog = new FontDialog(shell);

FontData fontData = dialog.open();
```

FontDialog 会自动选择在用户系统上对 SWT 而言可得的字体。这可能会导致较为复杂的显示，如图 10.4 显示的那样。

图 10.4 FontDialog 展示了在用户系统上安装的所有字体

返回的 FontData 可以用来被实例化并纠正显示字体，如同第七章所述的那样。

10.1.5 消息对话框 MessageBox

如果你的应用程序遇到一个错误而无法移除，则典型的做法是显示一个文本消息给用户并退出。由于这个和其他你需要展示一个消息对话框的原因，SWT 提供了 MessageBox。如下的代码行将会给用户一个简单的对话框来提示是否继续：

```
MessageBox dialog = new MessageBox(shell, SWT.OK | SWT.CANCEL);

dialog.setMessage("Do you wish to continue?");

int returnVal = dialog.open();
```

当你在使用 MessageBox 时，有两个重要的操作步骤。相对容易的部分是用 setMessage()来设定需要显示的文本信息。较为复杂的则是设定风格 style 了。使用一个 icon style 会给你的消息对话框加入一个恰当的图标。Style 可以用作控制 MessageBox 显示什么样的按钮。然而，某些 style 需要和其他 style 组合才能有效；允许的组合方式在表 10.1 中有展示。这些都可以自由地和 icon style 进行组合，表 10.2 中有列示。如果没有特为指定 button 或是有效的 button style，就使用 SWT.OK，这会最终在对话框内仅出现 OK 按钮。Open()会返回一个和被点击按钮的 SWT 常量相对应的整数。

Dialog button style combinations
SWT.OK
SWT.OK | SWT.CANCEL
SWT.YES | SWT.NO
SWT.YES | SWT.NO | SWT.CANCEL
SWT.RETRY | SWT.CANCEL
SWT.ABORT | SWT.RETRY | SWT.IGNORE

表 10.1 有效的 button style 组合

对话框图标	指代....
SWT.ERROR_ICON	发生了一个错误 An error has occurred.
SWT.ICON_INFORMATION	对话框正在展示给用户一个非互动的信息
SWT.ICON_QUESTION	对话框要求从用户处得到一个答案（通常是 OK 或是 Cancel）
SWT.ICON_WARNING	用户正准备执行一个有潜在危害的动作
SWT.ICON_WORKING	程序正在任务执行之中

表 10.2 对话框图标

第十章 对话框_2

10.2 JFace 对话框

标准的 JFace 的 dialog 类简洁易用，因为它们同其它的 UI 工具套件如 Swing 等工作原理相似。主要区别是静态方法来展示消息对话框、错误对话框之类的；而 JFace 则将这些不同的对话框类型进行子类化，最终通过

所有的 JFace 对话框都扩展自抽象类 org.eclipse.jface.dialogs.Dialog，而它自己恰恰又是扩展自 org.eclipse.jface.window.Window。图 10.5 展示了标准的对话框类之间的关系。就象你看到的，这些关系绝大部分是简单的；Error 和 Input 对话框则是依赖于基本谱系大纲之外的接口。

图 10.5 JFace dialog 继承谱系图

Dialog 的任意子类设计都会面临一些必要的考虑。你通过重写方法 buttonPressed()来改变对话框的行为。只要任意的按钮被点击（哪怕是你自己定义添加的非标准按钮）默认情况下都会立即关闭对话框。如果你想改变这一行为或是在对话框关闭之前作一些处理，你必须重写这些方法。时刻牢记如果你重写了 buttonPressed()方法，则在你自己的应用末调用方法 super.buttonPressed()之前，该对话框都不会被关闭。你也可以通过重写 okPressed()或是 cancelPressed()得到支持这些按钮的对话框内某些特定按钮的

hook。再说一遍，默认情况下，这些方法仅是关闭对话框而已——除非你打算自行添加 action，当然还得确保你在完成之后调用父方法。

最终，方法 createButtonsForButtonBar()控制着任意给定对话框所生成的按钮。如果你想改变这些按钮，那就有施展身手的机会了。有一个例外情况是 MessageDialog——因为你是如此频繁地改变在一个消息对话框内的按钮，构造器就提供了一个快捷方式来指明按钮该如何显示而无需将其子类化。

建议一个对话框应是模式化的，意味着一旦其被打开，那么其他窗口就不能被激活，除非对话框被关闭。使用一个模式化的对话框的代码编写通常是简单的，因为你基本可以确保只要对话框一经显示，你的用户就不再关注到你应用程序的其余部分。所有的本节内探讨的基本对话框都遵从这一准则。这将给你的程序代码带来两方面的影响。首先，若没有父窗口而去打开一个对话框是没有意义的，由于这我们的代码样例终就包含了一个父应用程序窗口。其次，你必须记得在你的代码中去调用 open()阻断，也即意味着对话框无路以哪种方式结束前方法都不返回值。Open()返回一个整型值，这是一个零基的被点选按钮索引，而－1 则代表着对话框是被除了点击一个按钮之外的其他一些方式关闭的（例如按 Esc 键退出）。

10.2.1 消息对话框 Message dialogs

一个消息对话框被用以向用户显示一则消息。由于点击任意按钮都会关闭该对话框所以互动性不大。

如下的程序片断展示了如何显示一则消息——如你所见，显示对话框本身仅需两行代码而已。这和在 Swing 中的调用 JOptionPane.showMessageDialog()大体相当：

```
MessageDialog dialog = new MessageDialog( testWindow.getShell(),

"Greeting Dialog", //the dialog title

null,

"Hello! How are you today?", //text to be displayed

MessageDialog.QUESTION, //dialog type

new String[] { "Good",

"Been better",

"Excited about SWT!" }, //button labels

0);

dialog.open();
```

消息对话框有若干钟不同的类型，皆在 MessageDialog 类内定义做静态常量。这个类型确定了所显示的图案；在我们的样例中，我们得到一幅作为问题标志的图案，因为我们业已经声明其为 QUESTION 类型的对话框了。其他类型包括 ERROR、INFORMATION、WARNING 和 NONE。在你的操作系统上每一个类型都有其标准的图案（除了 NONE 类型外，事实上它无图案可供显示）。

你也可以传递给一个字符串数组来作为按钮上的标签方式来自动生成按钮。对于每一个标签，MessageDialog 会生成一个对应的按钮。默认情况下，所有的按钮行为方式一致，都能关闭对话框。而在实践中，一个消息对话框通常都只有 OK 和 Cancel 按钮，除此之外属罕见。

构造器也可接收一个待显示的图案。

10.2.2 错误对话框 Error dialogs

Error dialog 在许多地方和 MessageDialog 相似，或者说它是向用户显示了一则错误的消息。你甚至可以通过生成一个类型为 ERROR 的 MessageDialog 来效仿一个 ErrorDialog。然而，ErrorDialog 通过使用接口 IStatus 来允许你显示更为深度的错误细节。IStatus 持有每一个所发生错误的质态的细节信息。结果可在图 10.6 中看得到。

图 10.6 一个带有多 IStatus 实例的 error dialog

图 10.6 中的 error dialog 是由如下代码生成的。我们生成一个 ErrorDialog 的实例然后传递一个持有错误信息的 IStatus 对象给它。根 IStatus 握有若干个其他 IStatus 实例可以提供关于该错误的递增性质的粒度信息：

```
...

ErrorDialog errorDialog = new ErrorDialog(testWindow.getShell(),

"Test Error Dialog",

"This is a test error dialog",

testWindow.createStatus(),

IStatus.ERROR | IStatus.INFO ); 1 Severity mask

...

public IStatus createStatus() {

final String dummyPlugin = "some plugin";

IStatus[] statuses = new IStatus[2];

statuses[0] = new Status(IStatus.ERROR, 2 Status

dummyPlugin, 3 Plug-in

IStatus.OK,

"Oh no! An error occurred!",
```

```
new Exception 4 Exception


statuses[1] = new Status(IStatus.INFO,

dummyPlugin,

IStatus.OK,

"More errors!?!?",

new Exception() );

5 MultiStatus

MultiStatus multiStatus = new MultiStatus(dummyPlugin,

Istatus.OK 6 Severity

statuses,

"Several errors have occurred.",

new Exception() );

return multiStatus;

}
```

1. Istatus 定义了若干个质态相关的常量。通过将其数位化，我们生成了一个位遮罩来描述在显示中所关注的质态。在每个独立的 IStatus 对象中设定的质态会与其遮罩相比对，而且仅当对照吻合时才会显示该对象的细节。通过将本例中的 INFO 改变位 WARNING，我们第二个状态对象的细节被抑止住了。
2. 在此我们生成一个状态实例，该实例应用了 IStatus 接口。这一思路就是封装在该类中所有关于一个 error 的信息然后让 ErrorDialog 或是其他用户程序根据语义环境来确定哪些内容适合被显示。
3. Istatus 要求一个插入式标识符，并且是具备唯一性质的字符串。这个标识符从未被 ErrorDialog 所使用过，因此我们给予该对象一个虚构值。
4. 装体存储了给出的例外 exception。例外包含在状态的 toString()方法内并可用 getException()方法获取。
5. 多状态也应用了 IStatus 接口并将多个的 IStatus 实例分组在一起。
6. 在此设定质态被用作去选择一幅合适的图案在对话框内显示。这个选择工作在一个 MessageDialog 内也是一样。

这串代码会显示一个带有消息“Several errors hava occurred”的错误对话框，如同在图 10.6 中见到的那样。点击 Detail 按钮会在对话框底部打开一个小窗口并显示出我们之前定义的两个状态对象的消息。

由于给予对话框的根 IStatus 对象是一个 MultiStatus 而且在方法 isMultiStatus()中返回 true，所以 Detail 按钮就出现了。除看到的外，它还处理一个 MultiStatus，ErrorDialog 调用 getChildren()来获取状态信息的细节。如果根 IStatus 向方法 isMultiStatus()返回一个 false，则 Detail 按钮将不会出现。一个 MultiStatus 的子对象可能是 MultiStatus 它们自己，允许你根据需要构建一个任意复杂的树。一个 MultiStatus 的质态等于其子对象的最大质态。JavaDocs 定义了一个无子对象的 MultiStatus 默认位一个 OK 的质态。

如你所见，ErrorDialog 提供了一个比 ERROR 类型下的 MessageDialog 更为高级的错误汇报机制。当然主要的缺点是其和 Eclipse 平台捆绑太紧密，而不是我们想要的那样较纯的基于 JFace 的小部件。不同的与 IStatus 相关的类不仅仅来自于 org.eclipse.core.runtime，而且还要求有插件标识符作为参数——这是一个在 Eclipse 而非 JFace 中的概念。关于所述的这个缺陷如何严重还是一个争议中的话题，在本例中之所以可以正常使用是因为是由于传递了哑值，因为该值从未用过。然而你却不可以传递 null，因为 Status 会检查插件值，如果是 null 的情况它会抛出一个内部申明失败的异常。这样使用这些类就可以减小到最低混乱度，而在非基于 Eclipse 的环境中，通常会产生拙劣的应用程序设计。如果你不需要为你的错误信息提供扩展的细节，则最好避免使用 ErrorDialog 而换作 ERROR 类型的 MessageDialog。然而，如果你确实需要细节功能支持，那就需要在你的已写就的类之程序设计中略作妥协、调整了。

10.2.3 输入对话框 Input dialogs

所谓字如其人，一个 InputDialog 就是用以允许用户输入文本的。这些对话框设计来的目的是为了处理相对较小数目文本（通常最多一行）。它们和 Swing 中的 JOptionPane.showInputDialog()扮演着相同的角色。

一个 InputDialog 所提供的关键功能是在于可选用一个 IInputValidator，它负责着校验一个输入的字符串。如下代码会提示用户输入 5 至 12 个字符长的字符串：

```
IInputValidator validator = new IInputValidator() {

public String isValid(String text) { //return an error message,

if(text.length() < 5) //or null for no error

return "You must enter at least 5 characters";

else if(text.length() > 12)

return "You may not enter more than 12 characters";

else

return null;

}

};

InputDialog inputDialog = new InputDialog( testWindow.getShell(),
```

"Please input a String", //dialog title

"Enter a String:", //dialog prompt

"default text", //default text

validator); //validator to use

inputDialog.open();

isValid()方法应用了对于用户输入文本的有效性检查。方法的语义提示可能第一眼看上去有点奇怪，但是它可以让你和用户沟通状态信息是更显灵活。方法传递给文本进行校验，然后若文本有效则返回 null。若无效，方法则返回一个字符串给用户以作为一个错误信息。在不同条件下返回不同的错误信息是允许的，这样有助于帮用户厘清思路：到底你在期望用户输入什么样的东西？

注意到该方法被频繁调用——即每次文本被改动（每次键击），它都会检查有效性。这意味着你的程序应当干尽可能少的事这样才可以快速反应。如果在这个环节花费太多时间会导致你的 UI 慢得难以忍受。

你或许会提供一个默认得字符出来输入到对话框。你可以传递 null，若文本预左侧为空。校验器也会传递空，此时任何得输入都会被接受。

InputDialog 被传递了一个标题，一个提示符和可能得某写默认文本以及一个 IInputValidator 的实例。对话框有 OK 和 Cancel 按钮显示。OK 按钮会根据 IInputValidator 提供的 isValid()方法所返回的值在任何时候被启用和禁用；如果没有 validator，则按钮总是被启用。一旦对话框被关闭，你可以用 getValue()来获取所键入的值。

第十章 对话框_3

10.2.4 进度指示器对话框 Progress monitor dialogs

应用程序经常会执行一些耗时长久的任务。当你的应用程序正忙时，让用户知道你正处于工作状态很重要，否则他们会赶到挫败。JFace 使得完成这项工作相对简化，因为它提供了了一个展示给用户看的任务状态的框架。

你必须理解几个接口来高效地使用 ProgressMonitorDialog。图 10.7 展示了这些接口之间的关系。

图 10.7 进度指示器类

在此最重要的应该是 IRunnableContext 和 IRunnableWithProgress 接口。IRunnableContext 提供了一个任意长时间运行任务的环境，而 ProgressMonitorDialog 则应用了这一接口。
IRunnableWithContext 使用一个 IRunnableWithProgress 的实例，而这意味着是被一个长时运行任务所应用。IRunnableWithProgress 因此是你的类必须应用的接口。最终，IRunnabelWithProgess 由 IProjessMonitor 的提供一个实例，它通常都会提供一个执行积极度的报告。

在和一个 IProgressMonitor 互动时，必须依次调用。该过程开始以调用 IRunnableContext 上的 run()方法，给它一个 IRunnableWithProgress 实例。在做完了必要的初始化工作后，IRunnableContext 调

用在 IRunnableWithProgress 上 run()，并传递一个 IProgressMonitor。IRunnableWithProgress 开始以调用 beginWork()并附预期总工作量的数字参数。它会周期性地调用 worked()和 subTask()对话框来通知进度指示器当前进度。它也会调用 isCanceled()来检查是否已被取消。最后，当任务完成后 done()被调用。在调用了 done()后，IProgressMonitor 就不再调用其他方法了。

IprogressMonitor 假定你的任务可以被分割成某一工作的抽象单元。然后它提供给你一个回调方法以让你通知 x 个工作单元已经完成。典型的，这个通知工作会在循环的最后一步被调用。然而，如果你的任务是一串系列的缓慢工作，诸如连接数据库或是网络，你可以给每一个单项工作分配一个值作为整项要做的工作的百分比。这样做就可以从手工编程通知用户还剩余多少待完成工作的繁琐中解放出来了。在我们的样例中，ProgressMonitorDialog 应用了 IRunnableContext 并提供了一个 IProgressMonitor 的实例；但是因为我们仅处理接口，这些可以由任何的 IRunnableContext 应用所使用。在 SWT/JFace 中，IRunnableContext 的唯一其他应用就是 ApplicationWindow 了，可以想见的是你可以为你自己特定的应用程序来应用自己的状态通知窗口——例如，对于一个确实耗时长久的任务，恰当的做法是在每一个阶段即将结束之前发送一封邮件或许是一个恰当的方式。在这种情况下，你可以用到这些接口。

我们的下一段程序将提供一个使用了一个 ProgressMonitorDialog 的样例。注意道显示给用户的状态文本是由 IRunnableContext 在调用了 beginTask()和 subTask()后控制的，而更新屏幕上的进度条则是由 IProgressMonitor 在接收了其 worked()方法后处理的：

```
ProgressMonitorDialog progressMonitor = new ProgressMonitorDialog(shell);

try {

IRunnableWithProgress runnable = new IRunnableWithProgress() {

public void run( IProgressMonitor progressMonitor )

throws InterruptedException {

progressMonitor.beginTask("Doing hard work...",

100); 1 Total work

while(!taskCompleted()) {

progressMonitor.worked(10); 2 Amount worked

progressMonitor.subTask("sub task: " + getCurrentTask());

... //Perform some long task

if( progressMonitor.isCanceled() ) { 3 Canceled check

throw new InterruptedException();

}
```

```
}

progressMonitor.done(); 4 Done

}

};

progressMonitor.run( true, 5 Fork

true, 6 Cancellable

runnable );

}

catch (Exception e) {

e.printStackTrace();

}
```

1. 我们必须通知 ProgressMonitor 我们已经开始了该项任务。另外，我们必要还要告诉它：我们期望执行的工作有多少个单元，抑或是在无法知晓情况下使用 IProgressMonitor.UNKNOWN。这个值也会给 Ui 一个提示：如果我们执行的工作量超过了预期值，则进度条会自己清空并重新开始直至我们通知其已告结束。
2. 在此我们通知了 ProgressMonitor 我们将执行一定的工作量。如果增幅足够大到视觉上有明显差异，就会引起它更新展示给用户看的进度条。在我们的案例中，每一个工作的单元大小都占据总量的 10%，所以每次调用该方法都会引起进度条有明显增幅。
3. 如果我们设定允许用户干预来取消该项工作，则我们必须周期性地检查是否用户声明了要取消。IRunnableContext.run()指明了如果任务被取消，则在完成必要的清空之后须抛出一个名为 InterruptedException 异常。这些异常最终传送向 IRunnableContext.run()的最初调用者。
4. 当结束时，我们必须通知 IProgressMonitor。
5. 当我们使用一个 IRunnableContext 开始一个长时间运行的任务时，我们被允许来指明我们是否应在一个独立的线程内运行该任务。若你运行于一个独立线程内，则你应确保在一个线程－安全 thread-safe 内全部资源都是可得的——详见第四章之 SWT 和线程讨论。
6. 我们也可以指明该项任务是否可由用户来取消。在一个 ProgressMonitorDialog 案例中，该值将决定了一个 Cancel 按钮是否会被显示。通常如果你的任务耗时过久则需要在首要位置用到一个 ProgressMonitor，你可以允许用户尽可能地取消它。注意到点击 Cancel 按钮对于正在执行的任务而言仅是一个推荐方案，任务有时可能并不能被强制停止。这取决于任务正确地检查取消请求并处理之——当然也可以不去管它，如果你忘了在代码中加入检查 check，它默认情况下还是会做的。

给更多的 control 使用 ProgressIndicator

ProgressMonitorDialog 提供了一个轻松方式来通知用户当前进度。然而，有些时候在展现一个进度条时会涉及到更多的 control。若有此需要，SWT 会让你实例化并直接使用小部件来控制进度条，这些我们下

文会谈及。

ProgressIndicator 小部件允许展现一个进度条而无需关注它到底如何运作。像 ProgressMonitorDialog 它支持工作的抽象单元——你只需用你预计的工作总量参数来初始化 ProgressIndicator：

```
ProgressIndicator indicator = new ProgressIndicator(parent);

...

indicator.beginTask(10); //total work to be done. Control is

...

//not displayed until this method is called

//use asyncExec() to update in the UI thread

Display.getCurrent()display.asyncExec(new Runnable() {

public void run() {

indicator.worked(1); //inform the ProgressIndicator that

//some amount of work has completed

}

});
```

在 progressIndicator 接收通知时，它假定是由通过计算已完成的展总的工作量的百分比来更新进度条的视觉外观的。

在工作总量情况不详的情况下，ProgressIndicator 也提供一个机动模式。在此模式下，你将调用的是方法 beginAnimatedTask()而非 beginTask()；并且也没有必要去调用方法 worked()。假定你的工作在一个非 UI 线程内正确完成，这也暗指你无需担心要调用 asyncExec()。

偶尔地，你或许会用到更多 control 甚至超过了一个 ProgressIndicator 所允许的。在你需要某一较低层面植入小部件时，SWT 会提供你 ProgressBar。

如果你想直接使用一个 ProgressBar，那你就需要负责照看进度条的更新了。如下代码即为示例：

```
//styles are SMOOTH, HORIZONTAL, or VERTICAL

ProgressBar bar = new ProgressBar(parent, SWT.SMOOTH);

bar.setBounds(10, 10, 200, 32);
```

```
bar.setMaximum(100);

...

for(int i = 0; i < 10; i++) {

//use asyncExec() to do updates in the UI thread

Display.getCurrent()display.asyncExec(new Runnable() {

public void run() {

//update how much of the bar should be filled in

bar.setSelection((int)(bar.getMaximum() * (i+1) / 10));

}

});

}
```

注意到除了要你自己计算进度条更新数外，调用方法 setSelection()会引起小部件每次都会更新；这个行为不同于 ProgressIndicator 或是 ProgressMonitorDialog，该两者仅会更新在变化足够达到引起最终用户的视觉差异才会做更新动作。

如你所见，使用 ProgressBar 比我们所讨论的其他小部件需要介入到更多的工作，而通常我们都会劝你避免趟这个混水，除非你别无选择。然而，偶尔可能会去这样用——如，你想让进度条呈凹陷状展示，则这些高级 control 就无能为力了。

10.2.5 自定义对话框 Custom dialogs

虽然我们讨论的对话框可以覆盖如此之多的常规任务，但是你经常还会发现你的应用程序会调用到某些独特的对话框甚至连 JFace 的设计者都不曾预见的。如果你想给你的应用程序生成一个新类型的对话框，我们推荐你扩展使用 JFace 的对话框类而非 SWT 的。因为 JFace 框架下的结构使得你的工作更加简便，而使用 SWT 的话你将不得不自行撰写绝大部分的常规功能项。

因为 JFace 提供的框架可以管理对话框的开启和关闭，你的工作就主要集中于定义在页上显示的 control。对话框提供了若干手段来定义你的对话框的布局，当然这取决于你使用的 control 的所处层次。在我们讨论何时该重写特定的方法之前我们将会研究该已何种顺序来展开。

因为对话框扩展了 Window，就像第二章中讨论得那样任何东西都会开始以 createContent()方法。在此之后作一些初始化工作，然后调用 createDialogArea()。这个方法构建了对话框的顶部。当 createDialogArea()方法返回后，createButtonBar()被调用来为在对话框底部的进度条生成一新的合成器和布局。最终 createButtonBar()调用 createButtonForButtonBar()来实例化按钮并在对话框上显示。

默认情况下会生成 OK 和 Cancel 按钮。

你可以随意地在整个过程的任意点上通过重写合适的方法来介入控制。通常，你可以限定自己来应用有关的 createDialogArea()或是 createButtonsForButtonBar()。CreateDialogArea()采用一个合成器来作为任何你所生成的 control 的父部件，并要求必须返回一个 Control 其 layout data 是 GridData 的一个实例。完成这一约定的最方便的途径是在你做自己的工作之前调用默认的应用：

```
Composite composite = (Composite)super.createDialogArea(parent);

...//add custom controls

return composite;
```

除了这么些的限定条件，你就可以自由地向你的对话框添加任意恰当的 control 了。

象 createDialogArea()一样，createButtonBar()必须返回一个反映自己布局数据 GridData 的一个 Control。不过，胜于重写 createButtonBar()，应用 createButtonForButtonBar()就相对简单了，你只需关注生成你所需的按钮而不必担心有关布局问题。按钮是由 createButton()方法生成的。例如，默认的 createButtonsForButtonBar()使用了如下代码来生成 OK 和 Cancel 按钮的：

```
createButton( parent, IdialogConstants.OK_ID, IDialogConstants.OK_LABEL,

true ); //make this button the default

createButton( parent, IdialogConstants.CANCEL_ID,

IdialogConstants.CANCEL_LABEL, false );
```

createButton()采纳了按钮的父部件、一个整型 ID、字符串来作为标签，而且采用一个标记来指示是否该按钮是作为默认生成的。除了将按钮加入其内部列表并恰当的更新布局数据外，默认情况下 createButton()也会向按钮添加一个 SelectionListener，该监听器在某编号 ID 的按钮被按下时可致对话框的 buttonPressed()b 被调用。如果你对于你的对话框的要求比较特殊（诸如向所有生成的按钮添加特定风格 style),你甚至还可以自由地重写方法 createButton().

10.3 更新小部件窗口

你已经目睹了调用 SWT 和 JFace 下每一个对话框的样例，以及本章我们还进一步深度演示如何做一个定制化的对话框。例 10.1 生成一个 Dialog 的子类来显示两个文本输入区域，一个用于输入 username，另一个则是输入 password。我们还添加了第三个按钮来清空任何已输入的文本。图 10.8 展示了该对话框的样子。

Figure 10.8 The custom password dialog

要完成这些，我们要重写 Dialog 的方法 createDialogArea()、createButtonsForButtonBar()和 buttonPressed()。注意到 createDialogArea()是所有三个方法中最见复杂的了。

Listing 10.1 UsernamePasswordDialog.java

```
package com.swtjface.Ch10;

import org.eclipse.jface.dialogs.Dialog;

import org.eclipse.jface.dialogs.IDialogConstants;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.GridData;

import org.eclipse.swt.layout.GridLayout;

import org.eclipse.swt.widgets.*;

public class UsernamePasswordDialog extends Dialog {

private static final int RESET_ID = IDialogConstants.NO_TO_ALL_ID + 1;

private Text usernameField;

private Text passwordField;

public UsernamePasswordDialog(Shell parentShell) {

super(parentShell);

}

protected Control createDialogArea(Composite parent) {

Composite comp = (Composite)super.createDialogArea(parent);

1 super.createDialogArea() method

GridLayout layout = (GridLayout)comp.getLayout();

super.createDialogArea()

layout.numColumns = 2;

Label usernameLabel = new Label(comp, SWT.RIGHT);

usernameLabel.setText("Username: ");

usernameField = new Text(comp, SWT.SINGLE);
```

```
GridData data = new GridData(GridData.FILL_HORIZONTAL);
usernameField.setLayoutData(data);
Label passwordLabel = new Label(comp, SWT.RIGHT);
passwordLabel.setText("Password: ");
passwordField = new Text(comp, SWT.SINGLE | SWT.PASSWORD);
data = new GridData(GridData.FILL_HORIZONTAL);
passwordField.setLayoutData(data);
return comp;
}
protected void createButtonsForButtonBar(Composite parent) {
2 createButtonsForButtonBar() method
super.createButtonsForButtonBar(parent);
createButton(parent, RESET_ID, "Reset All", false);
}
protected void buttonPressed(int buttonId) { 3 buttonPressed() method
if(buttonId == RESET_ID) {
usernameField.setText("");
passwordField.setText("");
}
else {
super.buttonPressed(buttonId);
}
}
}
```

1. 我们重写了方法 createDialogArea()来实例化本对话框中所需的 control。首先我们在超类中调用该方法已处理所有的布局信息。超类会保证返回一个带有一 GridLayout 的合成器，所以将方法应用到该返回的对象上是安全的。在设定布局有两个列后，我们生成一对要展示的标签和文本域。
2. 加入我们的 Reset 全部按钮仅需一行。我们确保在调用 super.createButtonsForButtonBar()后能够得到标准的 OK 和 Cancel 按钮。
3. 最终，我们要对用户的点击作出反应。当 createButton()被调用，它绑定于每一个按钮之上的正确监听器来确保 buttonPressed()被调用。我们会检查按钮的 ID 来看是否和我们早先生成的 Reset 所有按钮的 ID 匹配否。若匹配，我们重置文本域内的文本。否则就和超类通信由其正常处理按钮点击动作。

例 10.2 贡献了一个合成器，它并不很有趣。主要目的是在于投放我们的对话框。因为我们说过不能在 tab 上直接显示对话框，所以转由合成器来生成一个按钮。当按钮被按下，建廷器会立即实例化并加以显示我们之后会显示的 UsernamePasswordDialog。

Listing 10.2 Ch10CustomDialogComposite.java

```
package com.swtjface.Ch10;

import org.eclipse.swt.SWT;

import org.eclipse.swt.events.SelectionEvent;

import org.eclipse.swt.events.SelectionListener;

import org.eclipse.swt.layout.FillLayout;

import org.eclipse.swt.widgets.Button;

import org.eclipse.swt.widgets.Composite;

public class Ch10CustomDialogComposite extends Composite {

public Ch10CustomDialogComposite(Composite parent) {

super(parent, SWT.NONE);

buildControls();

}

protected void buildControls() {

FillLayout layout = new FillLayout();

setLayout(layout);
```

```
Button dialogBtn = new Button(this, SWT.PUSH);
dialogBtn.setText("Password Dialog...");
dialogBtn.addSelectionListener(new SelectionListener() {
public void widgetSelected(SelectionEvent e) {
UsernamePasswordDialog dialog =
new UsernamePasswordDialog(getShell());
dialog.open();
}
public void widgetDefaultSelected(SelectionEvent e){}
});
}
}
```

在生成了一个按钮后，我们向其添加一个 SelectionListener。当该监听器接收到一个 widget-selected 事件，它会生成一个新的 UsernamePasswordDialog 并打开，就如同在 JFace 中预建的那些对话框一样。

你可以向小部件窗口加入如下行来运行之：

```
TabItem chap10 = new TabItem(tf, SWT.NONE);
chap10.setText("Chapter 10");
chap10.setControl(new Ch10CustomDialogComposite(tf));
```

10.4 总结

使用由 JFace 提供的对话框通常是简洁易用的，虽然你在些任意代码前需要仔细考虑你的应用程序的设计。

MessageDialog 和 InputDialog 提供了 Swing 之外同样功能组件的又一选择。不同域 JOptionPane 的静态方法，在 JFace 中的 dialog 类们可以被子类化并加以定制来满足你应用程序独特的要求，而且它们还提供在简单情况下的易用功能项。

ErrorDialog 提供了更为高级的错误汇报功能，而与此同时还在你的应用程序中引入了额外的库依赖性，诸如你之前从未见到的参数等。在你使用 ErrroDialog 之前需要仔细考虑这一类问题；但我们想无论如何这个类总是有其价值所在的，特别是当你能将 org.eclipse.core.runtime 相关的类和参数与你自己的应用程序

有效的加以隔离的话。

在另一方面 IProgressMonitor 框架则提供了一个干净、可扩展的类的集合，且在你的代码中即刻使用。它还可以直接植入较低层面的 control，但我们推荐还是坚持使用框架内的类和接口，除非是在特殊情况下。通过写

IrunnableWithProgress 和 IProgressMonitor 接口，你可以简化并在新环境下重用代码。

最终，当你在生成一个定制化的对话框时，你最好是开始以一个由 JFace 提供的 dialog 类，以避免继承 SWT 的基本类。JFace 的框架提供的结构使生成一个新的对话框相对容易。所有需要你做的是重写若干个方法来定义用在你的对话框内的特定组件。

第十一章 向导_1

第十一章向导

本章涵盖内容

■ 多页对话框

■ JFace 向导框架

■ 持久对话框设定

今天，绝大多数的人都已熟悉了一个应用程序为某些任务而提供向导这一概念。通过将一个复杂任务分解为一系列的步骤，每次执行一个，这样就有可能将原来是一个换句话说是变相的胁迫性质的功能项集合浓缩成为相对终端用户友好的经历了。在 Eclipse 内能找到的较好例证就是当新建一个项目。Eclipse 支持一个广泛系列的语言开发，对于新项目而言每一个语言都有许多不同的功能项需要设置。不同于某些做法如让你深陷于充满组合框、文本输入框、复选框等的对话框，Eclipse 通过一个一步步生成一个新项目的系列过程来导向用户。你可以给你的新项目选择一种语言，然后是存放位置，再后是设置语言的特别设定。每次在你提供了极少的信息后，Eclipse 可以智能推断绝大部分的默认设置，你就可以点击 Finish 按钮来告诉程序向前一步生成该项目。如果你选择了你自己来设置细节，你可以自由的在个步骤间前进、后退来改变你之前作出的抉择。整个过程是赏心悦目的，因为你可以看到在永久固化下来之前它们是如何影响到你的选择的。

Jface 提供由一个框架来帮助你在自己的应用程序中生成并使用向导。该框架由一个三层的谱系图构成。每一层都可以获取多个实例——一个容器包含有向导，而向导则包含有页。每一层定义有一个接口和默认的该接口的应用。通常最方便的是将该应用子类化，但是为了考虑最大可能的灵活性框架设计为只由接口来引用对象。任何一个向导有关的接口的完整应用都可以毫无问题地自由混入现存类中。

图 11.1 提供了一个用来生成典型向导的类的概览图。根据前一章我们有关对话框的讨论，你应该可以认出其中某些类来。

图 11.1 向导类

图 11.1 展示了向导框架内的类和接口是如何整合在一起的。在此最为重要的是每一层的实体类——Wizard

和 WizardDialog 仅依赖于下一层的接口和类，而不是默认的应用。这相同的属性无论在那个方向衍化都会保留——虽然 IWizardPage 使用了 IWizard，且不管是否由 Wizard 或是一些其他类来应用了 IWizard。我们要从该图表的底部开始，再行向上研究，依次讨论每一个其中的类。然后我们将展示如何是它们一起工作。

11.1 多页向导 Multipage dialogs

我们之前章节中讨论的所有对话框都仅由一页构成。所有的可得功能项均一次全部显示，而对话框也只会开启和关闭。然而，一个向导需要比一个单一的对话框要显示更多的内容。JFace 提供了一个通用接口来将多页对话框使作向导特定接口的父容器。在将注意力转至向导之前，我们将简要叙述这一通用接口。

11.1.1 IDialogPage

IdialogPage 是在任意多页对话框内页的基本接口。接口提供方法来设定一个给定页全部属性，诸如其标题、描述或是要显示的图案。最重要的方法是 createControl()，该方法在当页要生成其内容时就会被调用。JFace 用抽象类 DialogPage 提供了一个默认的 IDialogPage 的应用，这个抽象类给在接口内除 createControl()外所有方法提供了默认的应用。就其自己而言，IDialogPage 既不有趣也无用处，所以我们干脆就继续讨论其最广泛使用的子接口：IWizardPage。

11.1.2 IWizardPage

向导的基本元素是一个页。一个页应该代表着你在指引用户的无论那个过程中的某一步。生成一个有用的向导的关键在于要较好地定义这些步骤——如果在某一页上装了太多信息反而于事无补，用户可能会感到混乱；另一方面，设置太多的单独步骤可能也适得其反，这样会惹烦了用户。

Jface 使用 IWizardPage 接口来代表在向导中的某一单一页。这个接口内的一系列方法得到了定义，其中一些最为重要的都总结在表 11.1 中。

表 11.1 由 IWizardPage 定义的重要方法

方法	描述
getName()	每一页都必须有其唯一的名字。该方法经常被用来从向导中获取一个特定的页。
GetNextPage(), getPreviousPage()	这些方法在用户点击 Next 或是 Previous 按钮来前后页移动时会被调用。正确的要移向的页必须被返回（当然结果会随着用户所作的选择不同而异）。
IsPageComplete()	指明用户是否已经填制完成了本页上必要的内容了。
CanFlipToNextpage()	指明 Next 按钮是否可用。典型地，这个方法在本页操作已完成而且在向导内至少还有一页的步骤时会返回 true 值。

几个其他的较为简洁易用的 getter 和 setter 方法也在 IWizardPage 中有定义。要对这些方法悉数加以应用很快另编程工作变得相当乏味。幸运的是，JFace 通过一个默认的应用 WizardPage 类来拯救你于水深火热之中了。

11.1.3 WizardPage

WidzardPage 应用了 IWizardPage 接口并提供了一个页的基本逻辑。你只需要应用 IDialogPage 的方法 createControl()来构建你页上的恰当 control，当然如你愿意修改页的行为特性也可以去重写一系列的其

他方法来达成。

例 11.1 展示了一个 WizardPage 的一个应用实例。页代表着一个单一的复选框，询问用户是否使用默认的目录（可能正在设定一个新的 Java 项目，或是其他类似的任务）。就其自身而言，该类并不提供有趣的行为特性。在本章后面，我们将向你展示任何组合多个 IWizardPage 应用来构建一个完整的向导。

Listing 11.1 DirectoryPage.java

```
package com.swtjface.Ch11;

import org.eclipse.jface.wizard.WizardPage;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.GridLayout;

import org.eclipse.swt.widgets.*;

public class DirectoryPage extends WizardPage {

public static final String PAGE_NAME = "Directory";

private Button button;

public DirectoryPage() {

super(PAGE_NAME, "Directory Page", null); 1 Constructor

}

public void createControl(Composite parent) {2 createControls() method

Composite topLevel = new Composite(parent, SWT.NONE);

topLevel.setLayout(new GridLayout(2, false));

Label l = new Label(topLevel, SWT.CENTER);

l.setText("Use default directory?");

button = new Button(topLevel, SWT.CHECK);

setControl(topLevel); 3 setControl() method

setPageComplete(true); 4 setPageComplete() method

}
```

```
public boolean useDefaultDirectory() {5 useDefaultDirectory() method

return button.getSelection();

}

}
```

1. WizardPage 构造器采用一个页名字（在该向导内这个名字应该是唯一的）——即页的标题，以及（可选的）一个要显示的图案的描述符。
2. 你的页必须应用该方法。在此你生成所有在页上显示的 control。对于本例，我们生成一个简单的标签和一个复选框。
3. 当你生成你的 control 完毕，你必须调用 setControl()来让超类知道你所生成的东西。这一操作如果失败会导致 JFace 声明为 runtime 的失败。
4. 该方法被用来发出信号：该页是否已经获取了足够的信息来允许用户继续下一步。简单起见，我们在此设定其为 true。而绝大多数的应用程序中，你需要在你的 control 上添加监听器来等待用户已经输入全部必要数据的确认信号事件。
5. 在此我们设定一个方法开放给所有的复选框状态作查询。其他在向导内的类也可查询到是否用户想要使用默认的位置和行为。

第十一章 向导_2

11.2 向导

一个向导起步于一个单独的页。一个向导组合一个页的集合来代表用户试图执行的全部任务。另外，为了成组这些页，向导的主要职责在于搞清楚是否整项任务已获取足够信息来结束以及在任务结束时该作哪些必要的处理。

类似于 wizard page，向导有一个接口和一个默认的应用。

11.2.1 IWizard

Iwizard 接口有很多的方法，绝大多数都是简洁的功能设置项。尚不值得大书特书，但是我们在表 11.2 中予以了描述。

表 11.2 IWizard interface 定义的重要方法

方法	描述
canFinish()	周期性地调用来检查当前的向导是否可以结束。结果将决定了是否可以启用 Finish 按钮。注意到即便是返回 true 值也并不意味着是可以立即关闭向导了，只有当用户点击了 Finish 按钮；任何他没有输入的都将给出合理的默认值。
CreatePageControls()	倾向于让向导来预先为其所有页生成 control，以便于其估算所需的最大尺寸从而避免在页面间来回切换时要重定尺寸大小的尴尬。该方法的 Wizard 应用调用在该向导内的所有页上方法 createControl()，这也是你通常所要干的。然而如果你想延迟生成某些页面，该方法也可以被重写，特别是生成的动作较为缓慢或根本每次必要。

PerformCancel()	当用户要求取消向导时提供通知。由于取消即可进行因此任何的清空或是其他过程必须被完成。这个方法将返回一个布尔值，若是 false 会发出信号给框架：当前取消动作并不被允许。
PerformFinish()	当用户成功地完成向导后发出通知。所有与该向导相关的逻辑应在此执行。和 performCancel()一样该方法也会返回一个布尔值，而且返回的 false 值告诉了框架结束的请求被拒绝。

对于 wizard page 而言，JFace 通过一个形为 Wizard 的类默认的应用来将你从应用所有的定义在 IWizard 中方法的苦差事中解救出来。

11.2.2 Wizard

继续我们的例子，我们将展示 Wizard 的使用，JFace 的默认 IWizard 应用。我们的子类是简单的，因为 Wizard 已经为我么做了绝大部分工作。注意到 Wizard 提供了一个光发系列的设置功能项（诸如设定图案和颜色）。

我们的样例 wizard 继续我们之前讨论设定的项目，使用我们自行开发的 DirectoryPage 类。我们在此仅讨论该类的纲要并在本章末给出该例的全部代码：

```
public class ProjectWizard extends Wizard {

public ProjectWizard() {

super();

}

public void addPages() { 1 addPages() method

addPage(new DirectoryPage());

//... add other pages as needed

}

public boolean performFinish() { 2 performFinish() method

DirectoryPage dirPage =

(DirectoryPage)getPage(DirectoryPage.PAGE_NAME);

if(dirPage.useDefaultDirectory()) {

...

}
```

```
return true;

}

public boolean performCancel() { 3 performCancel() method

//... perform cancel processing

return true;

}

}
```

1. 该方法被调用来告诉向导可添加所需的任何页。页通常以它们添加的顺序显示。要改变此行为特性，你必须重写方法 getNextPage()和 getPreviousPage()。
2. 在此我们浏览了生成我们的项目的真实过程。我们早先添加的目录页通过其页名 page name 来调取，然后它也可以被查询到我们感兴趣的数据。这也展示了为什么在向导中的每一页都必须要有其唯一的名字——若有重复，则方法 getPage()将无法确定该返回哪一页。
3. 当用户取消时，任何的清空都必须要作。

如你所见，Wizard 将会替你完成大部分的工作。除了一些设定功能项外，以及应用 performFinish()以及你想做的 performCancel()就所剩无几了。

11.3 合并总成

最终我们来到控制整个向导的这一层面：向导容器。虽然在看到的第一眼会觉得有些突兀，狐疑为何将它孤立于向导之外，因它允许一个容器成组多个向导并在期间切换。

11.3.1 Wizard 容器

一个向导容器意味着是一个或多个向导的主人。IWizardContainer 接口自身并不有趣——它提供以方法来得到当前页，若干个方法来更新容器窗口的方方面面，以及一个方法来程序性地改变当前显示页。绝大部分的真实动作来源于 WizardDialog，即 IWizardContainer 的应用。

11.3.2 WizardDialog

客户可以自由地提供它们自己基于 IWizardContainer 的应用，但是 WizardDialog 对于绝大部分的需求已足可以满足。典型意义上，WizardDialog 甚至都不必要进行子类化。我们首先将要展示如何使用 WizardDialog 本身，之后再是演示生成一个确定在运行时是否显示于特定页上的子类。

首先，例 11.2 显示的是标准的 WizardDialog。

例 11.2 WizardDialogDemo.java

```
package com.swtjface.Ch11;
```

```
import org.eclipse.jface.window.ApplicationWindow;

import org.eclipse.jface.wizard.WizardDialog;

import org.eclipse.swt.widgets.Display;

public class WizardDialogDemo {

public static void main(String[] args) {

ApplicationWindow testWindow = new ApplicationWindow(null);

testWindow.setBlockOnOpen(false);

testWindow.open();

ProjectWizard wizard = new ProjectWizard();

WizardDialog wizardDialog = new WizardDialog(

testWindow.getShell(),

wizard);

wizardDialog.create();

wizardDialog.open();

}

}
```

如同我们的 demo 程序显示的那样，如果所有你想做的仅是依次显示各向导页而已，则使用一个 WizardDialog 就很简单。只需将 IWizard 传递给对话框的构造器并调用 open()方法，则你的向导页们就会依照当初添加它们的顺序依次显示。

一个关于初始化错误的警告

一段程序代码会如我们呈现的那样运行，但是如果你在你自己程序中测试一个向导的话需要搞清楚一件事。当我们在最初测试向导的各项功能时，我们生成一个 WizardDialog 的子类并添加了一个 main()方法。不幸的是，由于在 SWT 和 java 的 classloader 之间存在的一个微妙互动关系，原来的就失效了。当你键入“java TestWizardDialog”，Java 虚拟机就会装载并初始化 TestWizardDialog 类，然后它会寻找并执行在此定义的 static void main()方法。为了初始化 TestWizardDialog，虚拟机需要初始化所有其超类，这也包括了 org.eclipse.jface.dialogs.Dialog。而 Dialog 有其静态的初始化编码来试图从 ImageRegistry 中获取特定的图案。而在此时系统还未完全的初始化（记得，main()方法还未执行呢），则从注册表获取数据失败，抛出一个 NullPointerException 的异常并导致主线程终止。在一般应用程序中这不成为问题，因为 main()方法通常位于一个类中或是应用程序窗口的子类中。然而值得引起注意的是在

本例以及其他的 SWT 类中这是一个潜在的问题，加入你被此问题所困扰。如出错的症状非常奇怪，而解决的方法相当简单——把你的 main()方法放到一个不是扩展子 SWT 类的类中即可。

第十一章 向导_3

11.4 组合向导

偶然的，你或许会遇到这种情况，即要求从几个可能的向导中选择一个。这边有一个很好的范例就是在 Eclipse 中当你选择 File->New->Other。你会看到一个含有系列选项的向导供你选择一个你想生成的新项目。无论选中哪个都会对应的运行一个独立的向导来指引你下一步操作。JFace 就是以类 WizardSelectionPage 和接口 IWizardNode 提供了这方面的支持。

11.4.1 WizardSelectionPage

WizardSelectionPage 扩展了 WizardPage 并和在一个向导内的其他页一样发挥作用。对于该类来说有一个额外的方法很重要：setSelectedNode()，它采用一个 IWiazardNode 作为参数。当一个节点被选中，其子类会适当时候会调用该方法。如果是在你直接子类化 WizardPage 的案例中，你必须在一个 WizardSelectionPage 的子类中应用 createControl()。该方法被应用来代表对于用户的可得选择——经常会以树的形式出现，但是 JFace 的设计者选择了不对你的情况下何种方式表现最佳作出假定。每个可得的选择应该帮定于 IWizardNode 的一个实例。

11.4.2 IWizardNode

一个 IWizardNode 总是倾向于作为为一个向导的真实实例一个占位符。WizardSelectionPage 传递这些实例给 setSelectedNode()并在选择页完成后从 getSelectedNode()中获取它们。这个接口包含两个重要方法（见表 11.3）。

表 11.3 IWizardNode 接口定义的重要方法

方法	描述
getWizard()	获取绑定于该节点上的 wizard。它假定在该方法第一次调用之前 wizard 不会被生成，而且如果该方法被调用多次，返回的页仅是被缓存的同一各 wizard，而不是每次都会返回一个新的。
IsContentCreated()	查询一个 IWizardNode 的状态并检查它是否已经实例化了一个 wizard。

通常，你通过子类化 WizardDialog 来使用这些类。在一个 WizardSelectionPage 完成后你将调用方法 getSelectedNode()来获取用户选择的节点。你然后可以在该节点上使用 getWizard()来获取向导 wizard 并能够将此 wizard 的实例传递给在 WizardDialog 内的 setWizard()。

11.5 持久化向导数据

有的时候在数次调用之间会需要保存一个向导的数据。例如，在 Eclipse 内的 Create New Java Class 向导包括有一系列的复选框来生成诸如一个 public static void main()或者在超类中默认应用抽象方法等。这些复选框的状态在应用后会被保存，所以如果你未选中复选框来生成一个 main()方法，则你就不必每次都进行更改了。

Jface 提供用 DialogSettings 类来轻松管理这些持久化的设置。虽然这些技术经常应用于向导，但是又有谁能说没理由应用到其他也想持久化状态的对话框内呢？

11.5.1 DialogSettings

DialogSettings 提供了一个 IDialogSettings 接口的应用，并使用了一个被某 XML 文件所支持的哈希表。这对于绝大多数的需求以可满足；但是当你发现需要将来某时需要转换应用时，通常你会根据接口而非实体应用来引用对象。

使用 IDialogSettings 是简单的：

```
IDialogSettings settings = new DialogSettings("mydialog");

settings.put("checkboxOneChecked", true);

settings.put("defaultName", "TestDialog");

settings.save("settings.xml");

IDialogSettings loadedSettings = new DialogSettings(null);

loadedSettings.load("settings.xml");

loadedSettings.getBoolean("checkboxOneChecked");

loadedSettings.get("defaultName");
```

当在运行时，这段代码会向当前目录写入一个简单的 XML 文件，看上去象下面的：

```
<?xml version="1.0" encoding="UTF-8"?>

<section name="mydialog">

<item key="defaultName" value="TestDialog"/>

<item key="checkboxOneChecked" value="true"/>

</section>
```

将值存入 XML 这个方式有着如下好处：方便手工编辑，或是测试值的不成对组合或是存入无效值情况下作紧急修补。

存储值 Storing values

值被存入 settings 对象。它或许是有给 save()方法一个文件名来存入该文件中或是存入 java.io.Writer。对应的它可由 load()方法或是（一个 java.io.Reader）来从文件中读出。因为 DialogSettings 使用 XML 来装载和保存，因此需要在你的类路径 classpath 中包含 xercesImpl.jar 和 xmlParserAPIs.jar（来自于

$ECLIPSE_HOME/plugins/org.apache.xerces_x.y.z）。

你传递给 DialogSettings 构造器的名字将生成一个片断。片断是成组在整个 dialog-settingss 内相关数据的方式。你可以使用 getSection()来按名字获取一个 section 片断，而 getSection()将返回另一个 IDialogSettings 的实例。正如你从装载 settings 的代码中看到的那样，在装载时是不需要指明 section 的名字的；它们会自动从文件获得。

获取值 Retrieving values

如果对于给定的键没有作过设定，则调用 get()或是 getArray()将返回 null。然而，如果你视图获取一个非设定值（或者是该文件已被手工编辑而因此不再含有有效的数字），则不同的数字的 get()会抛出名为 NumberFormatExceptions 的例外，所以当有可能某些值未被设定时，你必须要对处理此类情况有所准备。

这些究竟如何有用处将取决于你的目标平台。在 Java1.4 中，位于 java.util.prefs 包内的类已经提供了相近的功能。从一个设计的观点来看，采用由基本平台提供的工具通常应该是一个较好的选择，但是如果你的应用程序所支持的是 Java1.3 或是更早的版本恐怕就无福消受了。处于这样的情况下，IDialogSettings 就可以成为较为方便的备选方案了。

第十一章 向导_4

11.6 更新小部件窗口

要生成功能型向导，对于小部件窗口，我们需要比通常更多的类。在本章早先你已经目睹了 DirectroyPage。为了完成这一样例，我们需要加入更多的页，完成 ProjectWizard 的应用，并加入一个合成器的子类。

首先，让我们看一下 ChooseDirectoryPage。该页在用户拒绝使用默认目录时被调用。该页呈现了一个文本输入框给用户输入一个目录选择。ChooseDirectoryPage 在例 11.3 中予以了演示。

注意：很重要的是该设计纯粹是基于为了演示多个向导页如何工作的目的。在真实的应用程序中，你应该让用户在使用默认复选框的同一页上输入他选择的目录，你或许可能象早些章节中讨论的那样使用一个 DirectoryDialog。

Listing 11.3 ChooseDirectoryPage.java

```
package com.swtjface.Ch11;

import org.eclipse.jface.wizard.WizardPage;

import org.eclipse.swt.SWT;

import org.eclipse.swt.layout.GridData;

import org.eclipse.swt.layout.GridLayout;

import org.eclipse.swt.widgets.*;
```

```
public class ChooseDirectoryPage extends WizardPage {

public static final String PAGE_NAME = "Choose Directory";

private Text text;

public ChooseDirectoryPage() {

super(PAGE_NAME, "Choose Directory Page", null);

}

public void createControl(Composite parent) {

Composite topLevel = new Composite(parent, SWT.NONE);

topLevel.setLayout(new GridLayout(2, false));

Label l = new Label(topLevel, SWT.CENTER);

l.setText("Enter the directory to use:");

text = new Text(topLevel, SWT.SINGLE);

text.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

setControl(topLevel);

setPageComplete(true);

}

public String getDirectory() {

return text.getText();

}

}
```

这一页和 DirectoryPage 相似。输入框被呈现给用户，而且应用程序的其余部分都可以使用一个公共的方法来查询用户作出的选择。

我们样例中的最后页是关于用户选择的总结。在该页上没有用户互动内容；它显示了一段文本字符串来说明已做的选择。总结页在例 11.4 中显示。

Listing 11.4 SummaryPage.java

```
package com.swtjface.Ch11;
import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
public class SummaryPage extends WizardPage {
public static final String PAGE_NAME = "Summary";
private Label textLabel;
public SummaryPage() {
super(PAGE_NAME, "Summary Page", null);
}
public void createControl(Composite parent) {
Composite topLevel = new Composite(parent, SWT.NONE);
topLevel.setLayout(new FillLayout());
textLabel = new Label(topLevel, SWT.CENTER);
textLabel.setText("");
setControl(topLevel);
setPageComplete(true);
}
public void updateText(String newText) {
textLabel.setText(newText);
```

```
}

}
```

在某些方面，这个类和前面的两个恰恰相反。该类不是给客户提供查询当前页状态的方法，而是提供方法来更新显示的文本。作为强制、显性地更新页状态的替代方案，它在需要显示自己时，还可能让这个类查询某些共享状态，这种情况譬如一个代表着处于被构建过程中的项目的 Project 对象。这个方式就要求重写在 DialogPage 内定义 setVisible()方法。当 setVisible(true)被调用，textLabel 和其他任何相关的小部件会被刷新。通常，你应当选择这一方式，因为它仅需要如何显示在总结页 SummaryPage 这一局部知识。我们在此贯彻这一做法就是为了避免不得不写一个 Project 类和保持样例代码短小。

我们接下来要在例 11.5 中奉上 ProjectWizard 的完全版应用。这段代码较之本章早先的代码片断有了两个提高。首先，我们扩展了 addPages()应用来加入本向导内所需的全部页。更重要的是，我们还扩展了 getNextPage()逻辑。

Listing 11.5 ProjectWizard.java

```
package com.swtjface.Ch11;

import org.eclipse.jface.wizard.IWizardPage;

import org.eclipse.jface.wizard.Wizard;

public class ProjectWizard extends Wizard {

public void addPages() {

addPage(new DirectoryPage());

addPage(new ChooseDirectoryPage());

addPage(new SummaryPage());

}

public boolean performFinish() {

DirectoryPage dirPage = getDirectoryPage();

if (dirPage.useDefaultDirectory()) {

System.out.println("Using default directory");

}

else {
```

```
ChooseDirectoryPage choosePage = getChoosePage();
System.out.println( "Using directory: " +
choosePage.getDirectory());
}
return true;
}
private ChooseDirectoryPage getChoosePage() {
return (ChooseDirectoryPage) getPage( ChooseDirectoryPage.PAGE_NAME);
}
private DirectoryPage getDirectoryPage() {
return (DirectoryPage) getPage(DirectoryPage.PAGE_NAME);
}
public boolean performCancel() {
System.out.println("Perform Cancel called");
return true;
}
public IWizardPage getNextPage(IWizardPage page) {
if (page instanceof DirectoryPage) {
DirectoryPage dirPage = (DirectoryPage) page;
if (dirPage.useDefaultDirectory()) {
SummaryPage summaryPage =(SummaryPage)
getPage(SummaryPage.PAGE_NAME);
summaryPage.updateText("Using default directory");
return summaryPage;
```

```
}

}

IWizardPage nextPage = super.getNextPage(page);

if (nextPage instanceof SummaryPage) {

SummaryPage summary = (SummaryPage) nextPage;

DirectoryPage dirPage = getDirectoryPage();

summary.updateText( dirPage.useDefaultDirectory()

? "Using default directory"

: "Using directory:" +

getChoosePage().getDirectory());

}

return nextPage;

}

}
```

该类的有趣玩意包含于方法 getNextPage()内。在此我们控制着在页间的通行。我们必须正确地处理好两套方案。首先是在用户离开 DirectoryPage 的情况下，它可以选择使用默认的文件目录。传递给 getNextPage()的参数就是用户所来自的页，所以我们检查它是否是 DirectoryPage。若是，则在将参数施加于 IWizardPage 的正确应用后，我们那查询复选框的状态。若被选中，我们将跳过直奔状态页，所以我们使用 getPage()接收并返回它。

如果之前的页不是 DirectoryPage，或者用户没有作复选框的选中，我们就退回到原来的默认行为并通过调用 super.getNextPage()来确定接下来的该是哪一页。然而，如果接下来的也是 summary page 总结页，我们需要确保更新文本来反映用户的当前选择。在这种情况下，我们将 IWizardpage 施加于一个 SummaryPage 并接收需要的其他页来确定待显示的正确文本信息。如同在我们讨论 SummaryPage 之后的要点，这个逻辑是由我们将一个共享状态 shared state 隔离于 SummaryPage 之外导致的；通常我们应该避免它，因为这会在一个多页向导内是复杂度陡然增加。

最后一个类来圆满我们的样例的是由小部件窗口使用的合成器，如同在例 11.6 显示的那样。和之前章节的合成器们一样，这个也不是非常有趣。它奉上了一个按钮，当点击它，可以初始化并显示一个带有我们自己的 ProjectWizard 的 WizardDialog。

例 11.6 Ch11WizardComposite.java

```
package com.swtjface.Ch11;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
public class Ch11WizardComposite extends Composite {
public Ch11WizardComposite(Composite parent) {
super(parent, SWT.NONE);
buildControls();
}
protected void buildControls() {
final Composite parent = this;
FillLayout layout = new FillLayout();
parent.setLayout(layout);
Button dialogBtn = new Button(parent, SWT.PUSH);
dialogBtn.setText("Wizard Dialog...");
dialogBtn.addSelectionListener(new SelectionListener() {
public void widgetSelected(SelectionEvent e) {
WizardDialog dialog = new WizardDialog(
parent.getShell(),
```

```
new ProjectWizard());

dialog.open();

}

public void widgetDefaultSelected(SelectionEvent e) {}

});

}

}
```

这个类中要引起注意的部分是 widgetSelected()，在此我们初始化对话框。

注意到 WizardDialog 如同我们想要的那样不折不扣地工作着。我们可以将我们的 ProjectWizard 的一个新实例传递给它，然后让它该干什么就干什么去。

11.7 总结

向导框架构建于对话框类之上，这一点我们通过在一个单一对话框上添加多页支持的讨论已得出了结论。在向导框架的谱系大纲中包含着三个层级。IWizardContainer，由 WizardDialog 所应用，包含着 IWizard，通常更是 Wizard 的实例。挨下来，一个 IWizard 包含多个 IWizardPage。

一个向导的生命周期并不复杂。每一个 IWizardPage 都应用 createControl()方法来构建所需要显示的 control。在 IWizard 上调用的 addPages()被用来实例化并跟踪 page。在默认的 Wizard 应用中，这个没有其他多余的要求只需给每个页调用 addPage()即可。当每个页面被显示，方法 isPageComplete()和 canFlipToNextPage()被调用来确定是否启用 Next 和 Previous 按钮。当用户无论按下 Next 或 Previous 按钮，getNextPage()和 getPreviousPage()会在当前页上调用来确定接下来该显示哪一页。最终，一旦用户按下 Finish 或是 Cancel 按钮，wizard 自己上的 performFinish()和 performCancel()被调用来发出信号通知：恰当的过程应该予以执行。

除了在任一 wizard 内都会应用的三个标准的接口外，JFace 提供一个在谱系大纲之外的层级，利用它你可以组装多个 wizard。一个 WizardSelectionPage 使用 IWizardNode 的实例向用户提供多向导功能，允许来选择其中之一。每个节点都被设定来一旦选中就实例化向导，由此避免了去做多余的工作。
IDialogSettings 接口提供了一条持久化 settings 数据的方便途径，只要这些 settings 可以由 Java 的原型数据开代表。其中默认的应用，DialogSettings，会串行化对象并存入一个 XML 文件。

第十二章 高级特性_1

第十二章 高级特性

本章涵盖内容

■ 与系统粘贴板互动

■ 应用拖曳－投放操作

■ 保存用户喜好

■ 向标签加入图案

■ 在你的应用程序内嵌入一个 web 浏览器

现在为止，你应该已经熟悉了当你要构建一个正常工作的 SWT/JFace 应用程序所需的一切了。我们讲述的内容涵盖了所需的必要小部件们，像你展示如何在屏幕上将它们定位，讨论有关如何牢记确保一个良好设计的软件等话题。然而，在你逐步开展使用 SWT 时你还需要对一些杂项内容熟悉。

本章包括了一个广泛系列的题材。我们将开始以讨论如何从/向底层的操作系统传输数据，然后我们将展示如何使用这一能力来在你的应用程序中使用拖曳－投放、拷贝－粘贴等功能。接下来我们将讨论由 SWT 提供的用以管理用户偏好的两套框架和用 viewer 类来提高标签的显示。最终，我们将以粗略浏览在 SWT3.0 中新提供的小部件浏览器来结束本章，浏览器将使你可以在你的应用程序内控制用户的网页浏览。

因为这其间某一些类被设计为直接同底层操作系统互动，则它们会因处于不同的平台儿在功能上有所差异。另外，某些小部件在现时还不能在所有平台上得到全面支持。我们随后会指出这些特定平台的相关问题。

12.1 数据传输

虽然你可能尚未注意到，几乎所有你使用的应用程序都在屏幕之后来回不断地进行数据迁移。每次你做 cut 和 paste，应用程序必须要和暂存数据系统的 clipboard 进行互动。拖曳－投放项目要求类似的通信机制，因为应用程序需要让系统知道它能提供什么类型的数据，以及它又能接收那些给定类型的数据。

SWT 能够自动处理大部分的情况。例如，在早先的章节中你就看到，可以使用你操作系统上标准的键盘快捷操作方式来在一个文本 control 中剪切和粘贴文本。然而某些时候，如果你想要支持特定应用程序格式的拖曳－投放操作或是剪切、粘贴操作时，你需要手工来处理这些了。本节将讲述如何来进行操作，就像我们构建一个原型文件浏览器可以支持内部拖、放以及复制－拷贝操作以及同原生的操作系统工具进行互动。

图 12.1 文件浏览器，显示两个不同目录下的内容

在我们深度探讨这些技术细节之前，请看一下图 12.1，改图展示了我们所想要构建的东西。其中使用到了两个 ListViwer，每个均展示特定目录下的内容。从某一个列表中拖动一个或多个文件到另一列表中会导致一个对应的磁盘上文件拷贝动作。Copy 和 Paste 按钮会对应地分别将左侧列表内选定的文件拷贝到系统粘贴板，或者是从粘贴板上粘贴文件到左侧列表显示的当前目录中去。

12.1.1 Transfer 类

为了使数据能在元件间拷贝和移动，必须有某种途径来使这些元件们就数据是什么样的格式达成一致。如果应用程序不能理解某一给定格式，则将该数据导入就毫无意义——例如，一个文本编辑器根本无法处理拖曳－投放给它的一个图案。SWT 通过使用类 Transfer 以及其子类们来给元件提供了一个相对简单的方式以确

定什么样的数据格式属于可接收的。

每一个 Transfer 的子类都代表这一个特定的数据类型并且知道如何在一个 Java 请求和底层操作系统请求之间转换数据。SWT 应用 Transfer 子类们来处理文件、简单文本和 RTF 格式文本。如果这些不能满足你的要求，则你需要书写你自己的 Tranfer 应用，而这将超出本书论述的范围了。如果你确实需要自行定制这样的一个应用，那么不妨查询 org.eclipse.swt.dnd.Transfer 和 org.eclipse.swt.dnd.ByteArrayTransfer 的 JavaDoc。我们的目的是在于将 Transfer 实例视作为代表某一数据类型的黑盒子。每个子类都含有一个工厂方法 getInstance()来获取类的一个实例。我们可以将这些实例传给我们所感兴趣的数据类型，但是我们从不需要自行调用它们上面的任何方法。恰当时候，在后台，SWT 调用 javaToNative()和 nativeToJava()来传输数据。

表 12.1 显示了 SWT 提供的默认常规目的传输代理

表 12.1 SWT 提供的默认 transfer 代理

Transfer 类名字	描述
FileTransfer	传输一个或多个文件。数据是一个字符串数组，每个字符串都是某一文件或目录的路径。
TextTransfer	传输简单文本。数据是一个字符串。
RTFTransfer	传输 RTF 格式文本。数据是一个带有富文本格式的字符串。

12.1.2 拖曳－投放功能

允许用户从当前位置拖曳一个项目到它们希望去的任意地方可以帮助你的应用程序界面直观且易于使用。然而，要达成这一切，需要在屏幕之后运作大量的工作。

首先，你的应用程序必要要让操作系统知道它能提供和知道如何接收的数据类型。这些数据类型都单独为每个小部件做设置——只因为一个对象可以接受拖曳－投放给它的对象们并不意味着它就可以提供供拖曳－投放数据的数据了（进不等于出）。一旦系统清楚了在你的应用程序中不同的小部件的功能之后，这些小部件就可以接收一个拖曳－投放动作激发的事件然后应用恰当的逻辑。

拖－放操作的类型

当用户从某处拖曳一个项目到另外处时，典型意义上，可以被意读为多个 action 动作，例如：可能是一个 copy 动作，抑或是一个 move 动作。每个操作系统都会有不同的键盘操作惯例用来对应这些操作。然而，你的小部件也需要告诉 SWT 它们支持什么样的操作。一个只读的 display 或许只支持拖出时的拷贝项目功能，而不能允许用户移动项目。 org.eclipse.swt.dnd.DND 中类的常量可以指明这些操作功能的支持，表 12.2 对此做了具体的总结。

表 12.2 transfer 操作的类型

操作常量	描述
DROP_COPY	当拖进/出该 control 时，项目被拷贝。

DROP_MOVE	项目被移出当前位置到被拖曳－投放到的位置。
DROP_LINK	投放项目时生成一个原始位置上项目的链接。
DROP_NONE	当项目被投放是，什么也不做。

拖曳项目到一个应用程序中去

你可以注册一个 control，通过使用一个 DropTarget 的实例来促使该 control 可以接收拖曳－投放于其上的数据。DropTarget 存储了一个小部件可以选择的数据类型以及在该小部件上可执行的合法操作。在项目拖经该小部件时，操作系统可使用这些信息来提供视觉反馈：该项目是否可以投放在此处。一旦目标已经注册，则任何的 DropTargetListener 将能接收由用户企图拖曳－投放 control 内某些东西的动作触发的 DropTargetEvent 事件了。

生成一个 DropTarget 是简单的。你以一个小部件和一系列的操作使其实例化，然后调用 setTransfer() 方法来设定允许的数据对象。添加一个包含某逻辑的监听器，并在有东西被投放时执行之。如下即为程序片段：

```
int operations = DND.DROP_MOVE | DND.DROP_COPY;

DropTarget target = new DropTarget(control, operations);

Transfer[] transfers = new Transfer[] { TextTransfer.getInstance(),

RTFTransfer.getInstance() };

target.setTransfer(transfers);

target.addDropListener( new DropTargetListener(){...} );
```

如果你正在使用一个 viewer，那你必须在该 viewer 实例上调用 addDropSupport()方法而不是去试图直接地植入一个 control。接下来的程序片断是取自于我们的文件浏览器样例，展示了我们是如何为向一个 list viewer 内添加投放文件操作的支持的：

```
Transfer[] types = new Transfer[] {

FileTransfer.getInstance()

};

viewer.addDropSupport(DND.DROP_COPY, types, new FileDropListener(this));
```

注册的过程并不复杂。其中最为重要的部分数应用 DropTargetListener 接口了。当用户向一个 control 拖曳－投放入某项目 item 时，该接口内方法是以如下特定的顺序被调用的：

1. dragEnter()-当拖曳－投放一个项目时，光标进入 control 边界内。
2. dragOver()—当拖曳－投放一个项目时，光标移经 control。
3. dragOperationChanged()—无论何时用户改变准备进行的操作的类型，在执行操作时该方法可能

被调用多次。这在用户按下或是释放某些编辑键如 Ctrl 或是 Option 时最常发生。

4. dropAccept()—用户已经将一个项目投放到某 control 之内了。这是应用程序的最后机会来拒绝这一投放操作或者改变被执行的操作类型。
5. drop()—数据已经被投放。监听器必须应用正确的逻辑来处理给予的数据。

每个方法都给予了一个包含有当前操作信息的 DropTargetEvent。最重要的时，这个事件还包含了一个列表是关于数据源所能支持的数据类型、将要被投放的当前数据之类型、将被执行的可得操作，和准备执行的当前操作。你可以通过修改 currentDataType 和 detail field 来分别改变要使用的数据类型和要执行的操作。

第六个方法是 dragLeave()，它可在 dropAccept()之前的任意时间内被调用。该方法让应用程序知道用户已经将光标移出 control，并且没有投放动作会发生了。

除非你需要动态的更改数据类型或是操作，余则你所需的唯一的执行逻辑的方法就是 drop()了。SWT 和操作系统会处理除此之外的其他细节；如果不能就合适的数据类型和操作达成一致，则投放动作就不被允许，而且你的监听器不会侦听到这些事件。例 12.1 展示了我们如何为一个文件浏览器样例应用一个 DropTargetListener。

例 12.1 FileDropListener.java

```
package com.swtjface.Ch12;

import org.eclipse.swt.dnd.DropTargetEvent;

import org.eclipse.swt.dnd.DropTargetListener;

final class FileDropListener implements DropTargetListener {

private final FileBrowser browser;

FileDropListener(FileBrowser browser) {

this.browser = browser;

}

public void dragEnter(DropTargetEvent event) {}

public void dragLeave(DropTargetEvent event) {}

public void dragOperationChanged(DropTargetEvent event) {}

public void dragOver(DropTargetEvent event) {}

public void dropAccept(DropTargetEvent event) {}

public void drop(DropTargetEvent event) {
```

```
String[] sourceFileList = (String[])event.data;

browser.copyFiles(sourceFileList);

}

}
```

应用投放操作的逻辑是简单的。我们仅支持文件传输类型，所以当在这个方法被调用时假定数据类型时 FileTransfer，即以字符串数组为参数是安全的。如果我们要支持其他数据类型，我们将需要加入额外的逻辑来依照数据类型的不同而作对应不同的反应。同样的，我们假定操作是一个拷贝动作。因此，所有我们的监听器需要作的就是一个文件名的列表和告诉 FileBrowser 组件取拷贝这（个）些文件。

从你的应用程序中拖曳出项目

允许数据从你的应用程序中拖曳出遵循的操作流程和你刚才所见大体相似。一个 DragSource 被生成来注册一个 control 为一个数据源。DragSourceListener 的一个应用会接收用户开始一个拖曳操作的事件并在项目被投放后负责执行相应的逻辑。关于注册的代码几乎与上述的相同。第一个程序代码片断显示了如何来手工生成一个 DragSource：

```
int operations = DND.DROP_MOVE | DND.DROP_COPY;

DragSource source = new DragSource(control, operations);

Transfer[] transfers = new Transfer[] { TextTransfer.getInstance(),

RTFTransfer.getInstance() };

source.setTransfer(transfers);

source.addDragListener( new DragSourceListener(){...} );
```

就像和 DropTarget 一样，当你正在一个 viewer 时，viewer 上的一个方法会为你处理一些事项。接下来的摘录显示了在文件浏览器样例中的拖曳注册：

```
Transfer[] types = new Transfer[] {

FileTransfer.getInstance()

};

...

viewer.addDragSupport(DND.DROP_COPY, types, new FileDragListener(this));
```

DragSourceListener 接口比处理拖放操作的接口要简单得多；它只由三个方法组成，并依如下顺序调用：

1. dragStart()—用户已经开始从此 control 中拖曳数据。如果拖曳操作被允许执行，则事件得 doit 域必须被设定为 true。
2. dragSetData()—一个投放动作已被执行。该方法必须通过将数据置入事件得数据域来提供作为投放的对象。
3. dragFinished()—投放动作已成功完成。任何残存的清空动作，如为一个 move 操作删除原始数据等应在此实施。

每个方法接收附带有准备拖曳的数据的一个 DragSourceEvent。不同于 DropTargetEvent，这个事件若非特别提出无需作修改。

例 12.2 显示了我们如何在文件系统浏览器中应用这个监听器。

例 12.2 FileDragListener.java

```
package com.swtjface.Ch12;

import org.eclipse.swt.dnd.DragSourceEvent;

import org.eclipse.swt.dnd.DragSourceListener;

public class FileDragListener implements DragSourceListener {

private FileBrowser browser;

public FileDragListener(FileBrowser browser) {

this.browser = browser;

}

public void dragStart(DragSourceEvent event) {

event.doit = true; 1 Drag has started

}

public void dragSetData(DragSourceEvent event) {

event.data = browser.getSelectedFiles(); 2 Provide data to be

} transferred

public void dragFinished(DragSourceEvent event) {} 3 Clean up

}
```

1. 当用户试图从某 control 中拖曳一个项目时，该方法被调用。如果该拖曳动作被允许，则 doit 域必须被设定为 true。我们的样例总是允许拖曳，所以我们也总是将该域设定为 true。
2. 项目已被投放到一个接收器内，而且数据必须要提供。这个数据必须要符合来自于 dataType 域内当前数据类型的要求。我们只支持 FileTransfer，所以我们的应用会从浏览器中得到当前选定的文件并将它们插入事件中。
3. 操作已成功完成。如果该操作是一个 move 动作，那么原始的数据应该被删除。同样的，如果是与其他操作类型相关的任何清空动作都应该在这个方法内执行。因为我们的样例仅支持拷贝，故我们的应用为空。

12.1.3 使用粘贴板 clipboard

从系统粘贴板上拷贝数据或是拷贝数据向系统粘贴板的过程和前面所述的拖曳－投放几近相似。它曳使用了 Transfer 子类来从/向操作系统拷贝/粘贴数据。最主要的区别在于是使用系统粘贴板不需要如你作拖曳－投放动作那样预先作注册。在任何时候你的应用程序决定要作剪切、拷贝或是粘贴动作，通常对应于一个某种类型的 Action，它能够使用 org.eclipse.swt.dnd.Clipboard 类并做它所想要做的一切。

每个 Clipboard 的实例都生成并伴随一个 Display。记得在某些平台上，读取 clipboard 或许会要使用到原生的资源。所以极为重要的是一旦完成使用，你需要使用 dispose()来销毁 Clipboard。

将数据置入 clipboard

将数据放到系统的粘贴板上仅需要调运农个一个简单的方法 setContents()。而它所有要做的就是以恰当的 Transfer 来传递数据并译读之。这个程序片断展示了在我们的文件系统浏览器中这一切是怎么做的：

```
Clipboard clipboard = new Clipboard(getDisplay());

FileTransfer transfer = FileTransfer.getInstance();

clipboard.setContents( new Object[] { browser.getSelectedFiles()},

new Transfer[] { transfer });

clipboard.dispose();
```

注意到我们既传递了一个 Object 数组，也传递了一个 Transfer 数组。这些数组必须大小相同，而且索引号为 i 的 Transfer 实例必须能够处理在 Object 数组中索引号为 i 的 Object。这种方式下，所有可支持格式的数据可以一次性地置入粘贴板，而且你的应用程序不必担心何时它会被移除。

虽然我们的样例应用了一个拷贝操作，而不是剪切，但是 Clipboard 并不介意这之间的区别。它会接收数据，而是否从其原始源移除数据则取决于应用程序本身，并会独立于 Clipboard 之外进行处理。

从粘贴板 clipboard 粘贴数据

相同地，从粘贴板拷贝数据也是一项简单工作。当你的应用程序希望获取当前存储于粘贴板的数据时，它会

使用到两个方法。

GetAvailableTypeNames()返回一个字符串数组，这些字符串即为当前所有粘贴板能提供支持数据类型名字。这些值是与操作系统相关的，而且一个给定数据类型所返回的字符串会随着平台的不同而不同。因此，这个方法常用作辅助 debug 并且不便与使用在生产实践的编程中。然而当你在 debug 或是想要知道当前粘贴板上是什么数据类型是，这个方法就意义非凡了。

GetContents()采用一个 Transfer 参数并从粘贴板返回由 Transfer 给定格式的数据，或者如果无给顶格式的数据可提供就返回 null。

如果你的应用程序支持多个数据格式，通常你会需要重复调用 getContents()，且每次传递以一个不同的 Transfer 类型，直至你发现可处理的数据。

如下代码在一个文件浏览器样例中应用了一个 paste：

```
Clipboard clipboard = new Clipboard(getDisplay());

FileTransfer transfer = FileTransfer.getInstance();

Object data = clipboard.getContents(transfer);

if (data != null) {

browser.copyFiles((String[]) data);

}

clipboard.dispose();
```

你应该总是在调用 getContents()后检查是否为 null。因为很多情况下在粘贴板上的当前数据并不能转换为你的应用程序所能理解的格式，或者是粘贴板当前为空。忘记作此检查最终可导致 NullPointerException，而这通常是用户所不愿看到的。

第十二章 高级特性_4

12.1.4 文件系统浏览器

现在你已看到了实现我们的文件浏览器所使用来和操作系统互动的所有代码了。有些片断甚至已经超出了本书涵盖范围，然而对于 FileBrowser 类本身你尚未能一睹芳容。功德圆满起见，我们在此续上全部代码以方便编译和运行本样例。

首先，例 12.3 展示了构建视觉组件的合成器。该类实例化视觉 control 并给两个按钮添加监听器用来处理当按钮按下后的拷贝和粘贴逻辑。

例 12.3 Ch12FileBrowserComposite.java

```
package com.swtjface.Ch12;

import org.eclipse.swt.SWT;

import org.eclipse.swt.dnd.*;

import org.eclipse.swt.events.SelectionEvent;

import org.eclipse.swt.events.SelectionListener;

import org.eclipse.swt.layout.RowLayout;

import org.eclipse.swt.widgets.Button;

import org.eclipse.swt.widgets.Composite;

public class Ch12FileBrowserComposite extends Composite {

private FileBrowser browser;

public Ch12FileBrowserComposite(Composite parent) {

super(parent, SWT.NONE);

RowLayout layout = new RowLayout(SWT.HORIZONTAL);

setLayout(layout);

Button copyButton = new Button(this, SWT.PUSH);

copyButton.setText("Copy");

copyButton.addSelectionListener(new SelectionListener() {

public void widgetSelected(SelectionEvent e) {

Clipboard clipboard = new Clipboard(getDisplay());

FileTransfer transfer = FileTransfer.getInstance();

clipboard.setContents(

new Object[] { browser.getSelectedFiles()},

new Transfer[] { transfer });

clipboard.dispose();
```

```
}
public void widgetDefaultSelected(SelectionEvent e) {}
});
Button pasteButton = new Button(this, SWT.PUSH);
pasteButton.setText("Paste");
pasteButton.addSelectionListener(new SelectionListener() {
public void widgetSelected(SelectionEvent e) {
Clipboard clipboard = new Clipboard(getDisplay());
FileTransfer transfer = FileTransfer.getInstance();
Object data = clipboard.getContents(transfer);
if (data != null) {
browser.copyFiles((String[]) data);
}
clipboard.dispose();
}
public void widgetDefaultSelected(SelectionEvent e) {}
});
browser = new FileBrowser(this);
new FileBrowser(this);
}
}
```

接下来，例 12.4 展示了 FileBrowser 类。每个 FileBrowser 的实例生成并管理一个 ListViewer。一个 ContentProvider 读取了当前目录的内容，然后我们添加一个排序器和一个 LabelProvider 来是显示更加明晰。（在此我们用到的是第八、九章中的这些组件，所以就不展开进行细节讨论了。）FileBrowser 还包含有公共的方法来获取当前选定文件列表或是拷贝一个文件的列表到当前目录中去。

这个代码并非 SWT 相关的；如果你对挨下来发生的不甚熟悉，那么我们推荐你查询 java.io 程序包之相关文档。

Listing 12.4 FileBrowser.java

```
package com.swtjface.Ch12;

import java.io.*;

import java.util.*;

import org.eclipse.jface.viewers.*;

import org.eclipse.swt.dnd.*;

import org.eclipse.swt.widgets.Composite;

public class FileBrowser {

private ListViewer viewer;

private File currentDirectory;

public FileBrowser(Composite parent) {

super();

buildListViewer(parent);

Transfer[] types = new Transfer[] {FileTransfer.getInstance()};

viewer.addDropSupport(DND.DROP_COPY,

types,

new FileDropListener(this));

viewer.addDragSupport(DND.DROP_COPY,

types,

new FileDragListener(this));

}

private void buildListViewer(Composite parent) {
```

```
viewer = new ListViewer(parent);
viewer.setLabelProvider(new LabelProvider() {
public String getText(Object element) {
File file = (File) element;
String name = file.getName();
return file.isDirectory() ? "<Dir> " + name : name;
}
});
viewer.setContentProvider(new IstructuredContentProvider() {
public Object[] getElements(Object inputElement) {
File file = (File) inputElement;
if (file.isDirectory()) {
return file.listFiles();
}
else {
return new Object[] { file.getName()};
}
}
public void dispose() {}
public void inputChanged(Viewer viewer, Object oldInput,
Object newInput) {}
});
viewer.setSorter(new ViewerSorter() {
public int category(Object element) {
```

```
return ((File) element).isDirectory() ? 0 : 1;
}
public int compare(Viewer viewer, Object e1, Object e2) {
int cat1 = category(e1);
int cat2 = category(e2);
if (cat1 != cat2)
return cat1 - cat2;
return ((File) e1).getName().compareTo(
((File) e2).getName());
}
});
viewer.addDoubleClickListener(new IdoubleClickListener() {
public void doubleClick(DoubleClickEvent event) {
IStructuredSelection selection =
(IStructuredSelection) event.getSelection();
setCurrentDirectory((File)selection.getFirstElement());
}
});
setCurrentDirectory(File.listRoots()[0]);
}
private void setCurrentDirectory(File directory) {
if (!directory.isDirectory())
throw new RuntimeException(directory + " is not a directory!");
currentDirectory = directory;
```

```
viewer.setInput(directory);
}
String[] getSelectedFiles() {
IStructuredSelection selection =
(IStructuredSelection) viewer.getSelection();
List fileNameList = new LinkedList();
Iterator iterator = selection.iterator();
while (iterator.hasNext()) {
File file = (File) iterator.next();
fileNameList.add(file.getAbsoluteFile().toString());
}
return (String[]) fileNameList.toArray(
new String[fileNameList.size()]);
}
void copyFiles(String[] sourceFileList) {
for (int i = 0; i < sourceFileList.length; i++) {
File sourceFile = new File(sourceFileList[i]);
if (sourceFile.canRead() && currentDirectory.canWrite()){
File destFile = new File(
currentDirectory, sourceFile.getName());
if (!destFile.exists()) {
FileOutputStream out;
FileInputStream in;
try {
```

```
out = new FileOutputStream(destFile);
in = new FileInputStream(sourceFile);
byte[] buffer = new byte[1024];
while ((in.read(buffer)) != -1) {
out.write(buffer);
}
out.flush();
out.close();
in.close();
viewer.refresh();
}
catch (FileNotFoundException e) {
e.printStackTrace();
}
catch (IOException e) {
e.printStackTrace();
}
}
else {
System.out.println( destFile +
" already exists, refusing to clobber");
}
}
else {
```

```
System.out.println("Sorry, either your source file is not

readable " +

"or the target directory is not

writable");

}

}

}

}
```

要运行这段代码，需要加入如下行到小部件窗口中：

```
TabItem ch12Files = new TabItem(tf, SWT.NONE);

ch12Files.setText("Chapter 12 File Broswer");

ch12Files.setControl(new Ch12FileBrowserComposite(tf));
```

第十二章 高级特性_5

12.2 偏好

任何不同寻常的应用程序都有用户可以自行设置的 settings。虽然可修改的特定功能项不计其数且紧密绑定与特定的应用程序，设定这些功能项的过程最终通常就浓缩为一系列的诸如点击复选框、选择列表项、或是选择目标目录等互动操作。JFace 提供了一个框架来简化用户选择偏好的存储，获取以及显示给用户作后续的修改。

如同 JFace 的其他领域，偏好 preference 框架被划分为系列的接口，其中任一都有一个默认的应用。你可以自由地使用这些实体类，或者因你特殊需要也可应用你白手起家制作的接口。在 JFace 提供的应用之后，我们会考虑每一个接口。

Preference 框架实际上是 JFace 的 dialog 框架的一个扩展。

只有 IPreferencePage 接口是扩展自 dialog 包中的接口，但是这个是假定 preference 将是显示于一个模型化的 dialog 之内的。通过书写你自己特别的 IPreferencePageContainer 的应用是有可能更改这个行为特性的，关于这方面的内容我们本章稍后会展开讨论。

12.2.1 preference 页

Preference 通常是按照相关的集合来分组，而并非散乱分布的，这样的好处是用户可以方便地找出他所想要的功能项。在 JFace 中，这些集合被分配道各个独立的 preference 页中，这样可以一次性地呈现给用户。

如果功能项并不是太多，则无需再进行分割，但是无论如何你至少应该有一个 preference 页。

IPreferencePage

IpreferencePage 接口扩展自 IDialogPage。除了我们在第 11 章所讨论的 IDialogPage 方法，尚有七个新的方法在此定义（见表 12.3）

表 12.3 在 IPreferencePage 接口内定义的方法

方法	描述
setContainer()	将一个 IPreferencePageContainer 实例绑定于该页。
computeSize(), setSize()	处理在屏幕尚显示的 control 的尺寸。这些方法被传递并返回一个 org.eclipse.swt.graphics.Point 的实例。该域内的 Point 对象所代表的应被分别译读为宽度和高度，而不是代表一个(x,y)坐标。
OkToLeave()	当用户希望翻转到另外一页时调用该方法。若返回 false 值则可阻止用户离开本页。
IsValid()	指代了当前页是否有效。确切地说所谓的“有效”是主观的，但是通常它是一个指示器，指示了在当前状态下是否可能离开该页或是关闭对话框。
PerformOk(), performCancel()	分别表征着 OK 或是 Cancel 按钮已经被点击。在这两个方法中任何相关事件的处理过程应予实施，且该两个方法都会返回一个布尔值来指代是否事件被允许发生。

PreferencePage

抽象类 PreferencePage 形成了由 JFace 提供的 IPreferencePage 的所有应用的基础。扩展自 DialogPage，PreferencePage 提供了在一个 dialog 内显示 preference 的大部支持，包括一个标题 title 和一个可选的图案 image。

如果你正在直接的子类化 PreferencePage，你必须必须应用抽象的 createControl()方法来实例化本页所需的 control。再说一遍，这和其他的 DialogPage 是一样的。两个按钮，Apply 和 Defaults 会自动地添加到你 control 的父合成器中，除非在 control 生成之前被调用了 noDefaultAndApplyButton()方法。典型地，若有需要这会在你的子类构造器中予以实施。

默认情况下，无论何时 isValid()返回 true 值，PreferencePage 会为 okToLeave()返回 true 值。除非你使用了 setValid()来改变当前页的有效性，一个 PreferencePage 会总是认为其自己是有效的。这也意味着用户在任何时候被允许翻转页面或是关闭对话框。

方法 performOk()、performCancel()、performApply()和 performDefaults()都可以被重写来对合适的事件发生作出反应。默认情况下，这些方法什么事都不干，所以你确定了想让你的页面做一些有用的事情，则需要重写这些方法。

FieldEditorPreferencePage

由 JFace 所提供的 PreferencePage 的唯一的子类，以及在大部分情况中你将需要用到的唯一的类，应该是 FieldEditorPreferencePage 了。FieldEditorPreferencePage 假定你的 preference 是由一系列的可以独立修改的实体域构成的。 FieldEditorPreferencePage 意欲使编辑本页上的 preference 所需要的全部 FieldEditor 一网打尽变得易如反掌。如此，它可应用并重写我们在之前章节中述及的 PreferencePage

全部方法。典型地，你将发现留给你必须要应用的方法就一个而已。

一旦页面准备要对编辑器进行布局时，createFieldEditors()就会被调用。？？All the method does is add the editors to be displayed using addField().？？<郁闷！>编辑器们然后依照它们早先被添加的顺序依次布局安排在待显示的 control 上。FieldEditorPreferencePage 只向出现在 PreferencePage 上的那些开放一小部分的公共方法，而通常客户几乎不许要调用到它们。

第十二章 高级特性_6

12.2.2 域编辑器 Field editors

一个 field 编辑器负责展现和编辑一个单一的偏好值。编辑器种类繁多，有可能是一个允许用户键入的文本域，亦有可能是允许用户选择一个有效值的复杂对话框，等。JFace 包含着九个实体 FieldEditor 子类，这些我们都会讨论。因为它们基本上已经覆盖了你所需要的方方面面。

如果想要定义你自己的 FieldEditor 子类，你必须遵从一些步骤：

1. 考虑到你要使你的编辑器产生功用所需的基本 control。你将会应用 getNumberOfControls()来返回 control 的数字。而 FieldEditor 则用这个值来计算出如何来布局你的 control。
2. 在 createControl()的应用中，FieldEditor 调用抽象方法 doFillIntoGrid()。这将被用来实例化你的 control 并添加到传递给该方法的合成器上。
3. 如果你的编辑器包含一个 label，则 FieldEditor 会包括有一个内置的功能支持来存储该 label 的文本和这个 label 的 control。在本例中你应该使用 getLabelControl(Composite parent)，而不是去生成你自己的 Label。
4. 应用方法 doLoad()、doLoadDefault()和 doStore()来从绑定于你的编辑器的 PreferenceStore 中读取装载或是持久保存。这个存储可以使用方法 getPreferenceStore()来获取，如果使对于该编辑器尚没有持久的存储则将返回 null。

除了应用所有的抽象方法，你可以在你编辑器属性发生更改时激发事件。FieldEditor 还提供了一个名为 fireValueChanged()的方法，它可以采用一个属性名称、旧值和新值来自动调用已注册的任意 PropertyChangeListener。如果你的 control 没有什么属性会引起外界监听器的关注那这就不是必须的了——在 JFace 所包含的绝大部分的 FieldEditor 子类中并不倾向域激发这些事件，但如果你有需要它还是提供这一支持的。

无论你是在应用一个全新的编辑器抑或是在使用一个内置的，向你的 FieldEditor 添加有关有效性校验功能总是比较有用的。你可以通过重写 isValid()和 refreshValidState()这两个方法来达成这一目的。默认情况下，isValid()总是返回一个 true 值，而 refreshValidState()什么事情也不做。IsValid()其实很简单：它根据是否你的编辑器当前含有一个有效的值去保存来返回 true 或是 false。RefreshValidState()则稍微有点复杂。这个方法会查询 isValid()方法并且，如果值发生改变，则会为 FieldEditor.IS_VALID 属性激发一个 valuechanged 事件。RefreshValidState()会在不同时间由 FieldEditor 框架调用，特别是在装载之后和视图保存值之前。

应用你自己的 FieldEditor 这听上去好像有点复杂，但是你其实并不是经常需要去干这个。JFace 已经提供了九种类型的 FieldEditor（详见表 12.4），而你大部分情况下应该是使用其中之一的子类。

表 12.4 JFace 提供的 Field 编辑器

编辑器类	描述
BooleanFieldEditor	依复选框格式来展现她的 preference 偏好，选中即代表是 true 而不选中为 false。默认情况下，复选框总是出现在任何给出的 babel 的左侧，但如果想让标签处于复选框的左侧，则可以使用风格 style：BooleanFieldEditor.SEPARATE_LABEL 达成。
ColorFieldEditor	让用户来选择一种颜色。展现中有一个按钮，当被按下，会打开另外一个对话框并允许用户看到可得的颜色而且通过鼠标可以点选一个颜色。被选中的颜色被保存为一个 org.eclipse.swt.graphics.RGB 值。
DirectoryFieldEditor	让用户选择在文件系统上的任意目录。一个文本框显示着当前选择的目录。该值可作恰当的更改，或者显示的 Browse 按钮让用户浏览文件系统并依照图形显示选中一个目录。
FileFieldEditor	让用户选中一个文件名或是位置。你可以通过使用 setFileExtentions()方法在浏览文件系统时过滤显示的文件类型，而该方法会采用一个字符串数组作为参数。文件必须匹配该数组内所列的后缀名之一，否则不予显示。
FontFieldEditor	让用户选择一种字体，包括大小和粗体、斜体属性。点击 Change 按钮会打开一个显示所有当前可得字体选项的对话框。说明被选定字体的文本被显示；你可以在 FontFieldEditor 的构造器中设定用作该文本的字符串。该编辑器的值会作为一个 org.eclipse.swt.graphics.FontData 对象而返回。
IntegerFieldEditor	确保任何输入的值都是一个整数。你还可以通过使用 setValidRange()方法来强制输入的值介乎于某一特定范围。
PathEditor	让用户选中多个目录路径。当前选中的路径会被显示在左侧的一个列表中；而用以 add 添加、remove 移除、或者更改路径排列顺序的按钮在右侧。
RadioGroupFieldEditor	展示一个相互排斥项的集合，强迫用户在其中选一。可得选择项的标签在构造器中予以指明，一并附带任意一个被选中时返回的值。这些被传递为一个二维的字符串数组，具体如下： RadioGroupFieldEditor editor = new RadioGroupFieldEditor(/*some other parameters*/, new String[][] { {"Option One", "Value1"}, {"Option Two", "Value2"} }, /*more parameters*/);
StringFieldEditor	提供用户一个文本框来输入一段字符串。这个编辑器有两功能选项 VALIDATE_ON_FOCUS_LOST 和 VALIDATE_ON_KEY_STROKE 支持对输入文本的有效性校验；你可以使用 setValidateStrategy()来在其间勾选。默认情况下，StringFieldEditor 接收任何有效的字符串。若要加入你自己的有效性检验，需要重写被保护方法 protected method——doCheckState()。

	要限定输入文本的长度，可以使用 setTextLimit()方法。

12.2.3 偏好页容器 Preference page containers

正如同向导页 wizard page 由一个向导容器主导 wizard container 一样，偏好页 preference pages 是由一个偏好页容器 preference page container 来显示的。

IPreferencePageContainer

IpreferencePageContainer 接口必须被想要主导 preference page 的任何类应用。该接口是简洁易懂的；它仅有四个方法（见表 12.5）。

表 12.5 IPreferencePageContainer 接口定义的方法

方法	描述
getPreferenceStore()	由 preference pages 来获取一个持久存储的值
updateButtons(), updateMessage(), updateTitle()	让页来请求：容器更新其显示来满足当前活动页

虽然如你所需应用 IpreferencePageContainer 较为容易，但这样作其实并不需要。典型地，preference page 是在一个对话框内显示的，而 PreferencePageDialog 也会提供一个默认的应用来处理这种情况。

IPreferencePageNode

相对于在一个向导对话框页内显示都以一定规律有序排列，preference page 或许会依用户所愿的任意序列去充满。如果强迫一个用户点击经过若干由其不感兴趣的选项构成的页而而仅为了修改其中区区一项，这会不太友好。然而，要避免这种必经的关卡就必须要向用户提供一个页的列表（供其跳选）。

一个 prefernece node 偏好节点就是扮演着这种角色。一个 prefernce node 将一个 preference page 同一个唯一的 ID、一个可选的标题和图案捆绑在一起。标题和图案会在位于对话框左侧的一个树中展示给用户；当其中之一被点击，对应页就显示在右侧。以这种方式，用户可以快速浏览寻找他们所感兴趣的一组 settings。另外 IPreferenceNode 还使用了 add()方法来提供支持使得一个节点成为另外一个节点的子节点；这个时候，当父节点在树中扩展时子节点就得以显示。PreferenceNode 提供了一个 IPreferenceNode 的默认应用，但你几乎没有必要对此接口自行进行扩展应用。

在 preferenceNode 上的绝大部分方法都由框架来使用。你可以在实例化 PreferenceNode 时设定一个标题和图案。一般情况下，在你要管理一个给定节点的子节点时，你所需要调用的其他不多的方法就是 add()和 remove()了。

PreferenceManager

preferenceManager 是 JFace 用来组织 preference node 的一个工具类。它引入了一个路径 path 的概念，该概念用来辨识在一个层次结构图中的节点。一个 path 的组成是由一个或若干个节点（中间以分割符划分）的 ID 组合而成的字符串而构成的。默认情况下这个分割符是句号（.），但是你可以假以传递给 PreferenceManager 构造器的任意字符来改变之。字符串表征于分隔符之上，并且每个从根开始搜索每个

ID，然后找到第一个节点下的子节点，在后是下一个节点的子节点，如此这般，直到找到在路径 path 中的最后一个节点。你可以在根 root 下加入节点或是在路径 path 的当前结构中任意节点下作为其子节点再行添加。

PreferencePageDialog

PreferencePageDialog 是 JFace 的一个 IPreferencePageContainer 应用。它扩展了 Dialog 并加入了显示 preference page 的支持。可得的页在左侧的树中被显示，同时当前激活的页被显示于对话框的主区域。一旦你用一个 PreferenceManager 的实例来将对话框实例化并使用 setPreferenceStore()绑定一个持久化的 store，则你可以调用 open()方法了；余下来的一切皆由 PreferencePageDialog 来照料。

第十二章 高级特性_7

12.2.4 持久化偏好 Persistent preferences

如果每次应用程序启动时 preference 被重置，那么它们就基本没什么大用了。通过 IPreferenceStore，在 JFace 中有了一个持久化你的 preference 的较好途径。

IPreferenceStore

一个 IPreferenceStore 映射着 preference 名字和值这两者。而每个命名过的 preference 或许都会有两个值：一个默认值和一个当前值；如果没有当前值，则将返回一个默认值。Preference 可以是任意 Java 原型（也可参见关于在一个 IPreferenceStore 中方便存储特定 JFace 对象的 PreferenceConverter 的讨论）。每个在此接口中定义的 get 和 set 方法都采用 preference 的名字为参数进行运作。另外，还有一些方法是用来给一个给定的 preference 设定默认值的或是将一个 preference 重置为默认值，而且一个“脏指针”来检查是否该 store 已经发生改变。一个子接口，IPersistentPreferenceStore，加入一个 save()方法来持久化该值。

PreferenceStore

Jface 中包含有 PreferenceStore，即基于 java.util.Properties 类的 IPreferenceStore 的一个应用。

PreferenceStore 仅存储和默认值不想等的属性，由此可以使得必须写入磁盘的数据总量趋于最小。这些值都使用标准的属性文件格式（名字－值成对出现，以“＝”分隔）来持久。在使用 PreferenceStore 时，你可以有两个选择来装载和保存你的数据。最简单的方式就是在实例化一个实例时指明一个文件名：

```
PreferenceStore store = new PreferenceStore( "some_file_name" );

store.load();
```

或者，第二种方式就是在装载和保存时你可以给 store 一个流 stream 使用：

```
PreferenceStore store = new PreferenceStore();

FileInputStream in = new FileInputStream( "some_file_name" );
```

```
store.load( in );

...

FileOutputStream out = new FileOutputStream( "some_file_name" );

store.save( out, "Custom Header" );
```

你必须记得在向一个 PreferenceDialog 传递你的 PreferenceStore 之前显性地调用 load()，因为对话框并不会代表你去调用该方法。可是，在恰当的时候它会自动调用 save()。

注意到当在 PreferenceStore 构造器内没有指明文件名时调用无参数的方法 load()和 save()会导致一个 IOException。因为 PreferenceDialog 调用的是无参数的方法，你必须总是要使用一个带一文件名参数的构造器；只有你需要将这些值拷贝到一个备份的流 stream 时，才使用过载版本的 load()和 save()。

PreferenceConverter

PreferenceConverter 是 JFace 提供的另外一个工具。它由一系列的用来设定和获取常规的 SWT 对象的静态方法组成，而且这些方法不能和一个 IPreferenceStore 一起用。在幕后，出于要长久保存的目的 PreferenceConverter 会串行化对象自/向一个流格式。这些值都如下这般设定和获取的：

```
IPreferenceStore store = ...

PreferenceConverter.setValue( store, "color_pref", new RGB(0, 255, 0) );

...

RGB color = PreferenceConverter.getColor( store, "color_pref" );
```

12.3 标签装饰器 Label decorators

在我们早先关于 ILabelProvider 的讨论中，我们提到了有一个 IBaseLabelProvider 的替换应用。该应用在 ILabelDecorator 内，这是一个设计用来协同基本的 label-provider 一起提供额外信息的接口。

label decorator 总是倾向于用一些当前改订对象当前状态的视觉提示等去“装饰”该对象的外观。在 Eclilpse 的 Package Explorer 中我们可以找到很好的样例。Package Explorer 在一个树中展示了当前项目中所有的 Java 类，并以类们归属的包为单位组织。一个 label provider 显示了每个对象的名字以及一个用以指明该对象是一个类还是包的 icon 图标。标签装饰在标准图标的顶部被添加来指明异常条件，诸如如果有一个编译错误则会显示一个红色 X。

这种设计的主要好处在于其鼓励退耦的方式。继续我们的 Eclipse 样例，对于一个 Java 类的标准 label provider 只需要知道如何来获取来获取类的名字并画一个基本图标。覆盖 error 图标（或是警告图标或是其他可变状态）的逻辑散布于装饰器 decorator，在此 decorator 中它可以被应用于 pakage 包，也可应用于任何其他恰当类型的对象。相同地，因为 Java 类的 label provider 并不被显示状态部标 status icon

的代码所束缚，它可以轻松的在其他不需要状态图标的环境中得以重用。

12.3.1 ILabelDecorator

你用来应用装饰器 decorator 功能的主要接口是 ILabelDecorator。ILabelDecorator 是扩展于 IBaseLabelProvider 并且和 ILabelProvider 相似。有两个方法得到了定义：decorateText()和 decorateImage()。每个都会有伴随着当前显示的文本或图案的一个域对象作为参数传递。每个方法会针对给定的域对象赶回新的文本或是图案。

当你在应用 decorateImage()时，记得每个图案都会消耗相对并不宽裕的系统资源。因此避免无谓地去生成新图案很重要。在我们本章稍后内容中将会使用到 ImageRegistry，它有助于避免生成不需要的图案实例。关于使用图案的最佳实践我们已在第七章中有过讨论。

12.3.2 DecoratingLabelProvider

一旦你应用了 decorator，我们就要确保在执行了装饰后，有为之一振的效果。不同于提供方法来显性化地在 viewer 中加入装饰器之做法，JFace 提供了 DecoratingLabelProvider 类。该类扩展自 LabelProvider 并由此应用了 ILabelProvider。舍弃了自行提供 label 的做法，DecoratingLabelProvider 在起构造器采用了一个 ILabelProvider 实例和一个 ILabelDecorator。它首先代表 label provider 来调用 getText()和 getImage()，接着又代表 label decorator。然后 DecoratingLabelProvider 被帮定于 vierwer，而不是直接地用 ILabelProvider 来调用 setLabelProvider()。因为它是 ILabelProvider 的一个实例，所以你可以通过向构造器中传递适当 DecoratingLabelProvider（而非 LabelProvider）实例来轻松地将装饰器 decorator 串接起来。每个 decorator 被依次调用来构成最终的结果。这一技巧显示如下：

```
DecoratingLabelProvider firstDecorator = new DecoratingLabelProvider(

new MyLabelProvider(),

new FirstLabelDecorator() );

DecoratingLabelProvider secondDecorator = new DecoratingLabelProvider(

firstDecorator,

new SecondLabelDecorator() );

viewer.setLabelProvider(secondDecorator);
```


12.3.3 一个样例

我们现在将展示一个我们讨论的 decorator 概念的样例，所谓的 decorator 概念是之构建一个显示家庭成员关系的一个树。每个人都被装饰以其家庭名字和一个代表其性别的图标。本例的基础架构和我们之前讨论的 TreeViewer 较为相似。出于节省篇幅的考虑，我们将不再通篇展示这个样例，取而代之的是于 label

decorator 相关的片断。

第一步是生成代表着树上各个节点的 TreeNode 类。成员变量和构造器如下：

```
public class TreeNode {

private String firstName;

private boolean isMale = false;

private String familyName;

private List children = new ArrayList();

private TreeNode parent;

public TreeNode(String firstName, String familyName, boolean male) {

this.firstName = firstName;

this.familyName = familyName;

isMale = male;

}

//accessor methods

...

}
```

每个属性皆有一个可得值，所以我们的 decorator 就能查询 TreeNode 来取得所需数据。

我们的 ILabelDecorator 的应用是简洁易懂的。在此我们方便其间扩展了 LabelProvider，这样可以应用在 IBaseLabelProvider 内定义的方法：

```
public class FamilyDecorator extends LabelProvider implements IlabelDecorator {

private static final String MALE_IMAGE_KEY = "male";

private static final String FEMALE_IMAGE_KEY = "female";

private ImageRegistry imageRegistry;

public FamilyDecorator(Shell s) {
```

```
imageRegistry = new ImageRegistry(s.getDisplay());
Image maleImage = new Image(s.getDisplay(), "male.gif");
Image femaleImage = new Image(s.getDisplay(), "female.gif");
imageRegistry.put(FEMALE_IMAGE_KEY, femaleImage);
imageRegistry.put(MALE_IMAGE_KEY, maleImage);
}
public Image decorateImage(Image image, Object element) {
if(element == null) return null;
TreeNode node = (TreeNode)element;
if(node.isMale()) {
return imageRegistry.get(MALE_IMAGE_KEY);
}
else {
return imageRegistry.get(FEMALE_IMAGE_KEY);
}
}
public String decorateText(String text, Object element) {
if(element == null) return null;
TreeNode node = (TreeNode)element;
return text + " [" + node.getFamilyName() + "]";
}
}
```

构造器生成我们所需要的图案然后将其保存在 ImageRegistry 以备将来之用。当 decorateImage()被调用，它会检查 TreeNode 对象上的 isMale()方法并依名字从注册表中获取正确的图案；就这样返回图案供显示。（注意：本段已被简化；在一个真实的应用程序中，一般你会需要在另一个图案的顶部勾画你的装饰

然后返回合成的结果。）

decorateText()也是简洁的。每个节点的家庭名字被获取然后后缀在已经显示的文本上，所得结果就返回为被显示的文本。

最终，我们用一个我们的 decorator 实例来生成一个 DecoratingLabelProvider 并告诉 viewer 来使用这一新的实例而非默认的 label provider：

```
...
viewer.setLabelProvider(
new DecoratingLabelProvider(
(ILabelProvider)viewer.getLabelProvider(),
new FamilyDecorator(getShell())));
...
```

注意到在此我们获取了 viewer 的默认 label provider 并将其作为 DecoratingLabelProvider 的基础来传递使用。在你自己的应用程序中，这段代码会经常被一个定制的 label provider 的实例所取代。运行 demo 则形成如图 12.2 所示的形状。

默认的 label provider 使用 toString()方法在每个节点上仅添加了其 first name 字符的文本。而图标和家庭名字则是在我们自己的 label provider 生效后添加上去的。

图 12.2 一个装饰过的 TreeViewer

注意：Eclipse 中 label decorator 的使用——有其是在一个 Eclipse 的插件 plug-in 中使用 label decorator 时，你必须认识到有些小问题，诸如你不要不小心去碰到内建的 decorator。另外还有一个你可用的特性是可以在 plugin.xml 文件中设置你的 decorator。关于此类主题的细节讨论已经逾越了本书的范畴；但是你可以不妨看一下 Balaji Krish-Sampath 的精彩文章《Understanding Decorators in Eclipse》，具体查询网址：www.eclipse.org/articles/Article-Decorators/decorators.html，若你正在 Eclipse 内使用 decorator，相信它会使你受益匪浅。

第十二章 高级特性_9

12.4 浏览器小部件

伴随着互联网的兴起，HTML 业已成为生成用户界面的最为重要的技术了。它有着诸多优势：书写简单、易于理解和快速修改以适应不同设计。当你在使用诸如 SWT 以及 Swing 等厚客户端技术开发一个应用程序时，通常你会发现你自己会羡慕于由一个基于 HTML 的接口所影响到的快速开发周期。考虑到在 HTML 中

所蕴涵的如此巨大的信息量，有时在你的应用程序中集成显示 HTML 的功能就变得有意义了。

谢天谢地，Eclipse 的 3.0 发布版中,SWT 小组提供了一个 Browse 小部件来使这一切简化。通过使用 Browser，你可以嵌入用户的 web 浏览器到你的应用程序中并使用它来显示 HTML 而无需再写一个定制化的渲染引擎。另外，因为浏览器使用是在你的平台上功能齐备的原生 web 浏览器，你可以获取处理 JavaScript、XML 和任意为浏览器所理解的格式的能力。

在我们更深入地探讨 Browser 小部件之前，几条忠言可以先听听：

■ Browser 目前为止并不是在所有平台上得到支持的。它在微软的 Windows 工作没问题，但是在 Linux 上使用要求有 Mozilla1.5。OS X 以及其他平台还不能支持 Browser；如果你试图将其实例化，则结果就是一个 SWTError。代码正在积极开发中，但是，若你有考虑使用 Browser，应当检查一下你打算支持的平台下状态如何。

■ 在写作本书之时，Browser 的 API 还被认为不甚稳定。当 SWT3.0 推出发行时它会最终定稿，还有可能到时它又会和我们在此讨论的代码样例不匹配。这样情况下，你应当查询你发行版中的小部件版本来了解区别何在。

Browser 的 API 是简单的，它由若干方法组成：用来装载采自于一个 URL 的文档，浏览页时前进和后退，刷新当前的 URL，停止载入当前页。另外，Browser 还广播若干其特有的事件。在当前，这些事件包括打开、关闭、隐藏窗口、改变位置和指示页面载入进度。

使用 Browser 小部件或许还要用到额外的原生库，这取决于你所使用的操作系统平台。对于 Windows 而言没有什么额外的依赖性，但是在使用 Linux/GTK 发布版的 SWT 中 Browser 需要将 libswt-mozillagtk 库包含到你的 LD_LIBRARY_PATH 中。在其他支持的平台上，你应当检查一下是否有原生库包含你系统平台上的 web 浏览器。

我们接下来提供一个简单的 Browser 样例作为实战。；运行时，例 12.5 中的代码会打开一个浏览器窗口。一个文本框允许用户输入一个 URL，而当在小部件窗口的 Open 按钮被点击，对应的位置在浏览器中被打开。前进 Forward 和后退 Back 按钮让用户控制 WidgetViwer 内浏览器的浏览。运行样例得到如图 12.3 的平面截图。

图 12.3 内置 Mozilla 的一个 SWT 应用程序

Listing 12.5 Ch12WebBrowserComposite.java

```
package com.swtjface.Ch12;

import org.eclipse.swt.SWT;

import org.eclipse.swt.browser.Browser;

import org.eclipse.swt.events.SelectionEvent;

import org.eclipse.swt.events.SelectionListener;
```

```
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.*;
public class Ch12WebBrowserComposite extends Composite {
private Browser browser;
public Ch12WebBrowserComposite(Composite parent) {
super(parent, SWT.NONE);
GridLayout layout = new GridLayout(2, true);
setLayout(layout);
browser = new Browser(this, SWT.NONE); 1 Create Browser instance
GridData layoutData = new GridData(GridData.FILL_BOTH);
layoutData.horizontalSpan = 2;
layoutData.verticalSpan = 2;
browser.setLayoutData(layoutData);
browser.setUrl(
"http://www.manning.com/catalog/view.php?book=scarpino");
final Text text = new Text(this, SWT.SINGLE); 2 Open URL
layoutData = new GridData(GridData.FILL_HORIZONTAL);
text.setLayoutData(layoutData);
Button openButton = new Button(this, SWT.PUSH);
openButton.setText("Open");
openButton.addSelectionListener(new SelectionListener() {
public void widgetSelected(SelectionEvent e) {
browser.setUrl(text.getText());
```

```
}
public void widgetDefaultSelected(SelectionEvent e) {}
});
Button backButton = new Button(this, SWT.PUSH);
backButton.setText("Back");
backButton.addSelectionListener(new SelectionListener() {
public void widgetSelected(SelectionEvent e) {
browser.back(); 3 Forward and Back buttons
}
public void widgetDefaultSelected(SelectionEvent e) {}
});
Button forwardButton = new Button(this, SWT.PUSH);
forwardButton.setText("Forward");
forwardButton.addSelectionListener(new SelectionListener() {
public void widgetSelected(SelectionEvent e) {
browser.forward();
}
public void widgetDefaultSelected(SelectionEvent e) {}
});
}
}
```

1. Browser 象其他 control 一样带有一个父合成器以及一个风格 style 参数而被实例化。当前，Browser 尚不支持任何 style。
2. 通过将 URL 传给 setUrl()方法任意有效的 URL 都可以被打开。很明显就是浏览器当前载入页上的

东西。

3. 通过调用恰当的方法浏览器可以被要求根据浏览历史来向前或是退后。

如果你是在一个支持的平台上，你可以向小部件窗口加入如下列来运行改样例：

```
TabItem ch12WebBrowser = new TabItem(tf, SWT.NONE);

ch12WebBrowser.setText("Chapter 12 Web Browser");

ch12WebBrowser.setControl(new Ch12WebBrowserComposite(tf));
```

12.5 小结

在本章内我们已经论述了若干重要的命题，信息容量看上去不容小觑。但是不必急着第一眼就想掌握它；许多我们讨论的概念并不是我们日常 SWT 和 JFace 编程所必须的。最重要的是要了解它们所能给出的功能，所以当你一旦想要用到有关拖曳—投放操作或者是用户偏好时，你可以知道该要用上什么。

你也应该牢记 SWT 还在持续不断的开发之中。虽然我们现时在写作本书时，在诸多平台上 Browser 的小部件还不能全部被支持，但是当你着手你自己的 SWT 应用程序时情况或许就有了改观。记得多看看 www.eclipse.org 网站上的文章，了解在本书交付印刷之后由有了哪些改进。

How to design graphical applications with Eclipse 3.0

SWT/JFace IN ACTION

Matthew Scarpino
Stephen Holder
Stanford Ng
Laurent Mihalkovic

MANNING