

O'Reilly & Associates 公司介绍

为了满足读者对网络和软件技术知识的迫切需求,世界著名计算机图书出版机构 O'Reilly & Associates 公司授权中国电力出版社,翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly & Associates 公司是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司, 同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为二十世纪最重要的 50 本书之一) 到 GNN (最早的 Internet 门户和商业网站), 再到 WebSite (第一个桌面 PC 的 Web 服务器软件), O'Reilly & Associates 一直处于 Internet 发展的最前沿。

许多书店的反馈表明, O'Reilly & Associates 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比, O'Reilly & Associates 公司具有深厚的计算机专业背景, 这使得 O'Reilly & Associates 形成了一个非常不同于其他出版商的出版方针。O'Reilly & Associates 所有的编辑人员以前都是程序员, 或者是顶尖级的技术专家。O'Reilly & Associates 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家, 而现在编写著作, O'Reilly & Associates 依靠他们及时地推出图书。因为 O'Reilly & Associates 紧密地与计算机业界联系着, 所以 O'Reilly & Associates 知道市场上真正需要什么图书。

目录

序	1
前言	7
第一章 Ant 入门	15
文件和目录	15
Ant 的构建文件	16
运行 Ant	19
Ant 命令行参考	22
构建文件轮廓	24
继续学习	25
第二章 安装和配置	26
发布	26
安装	27
配置	36

第三章 构建文件	40
为什么用 XML?	41
Ant 构建块	42
一个示例工程及构建文件	47
构建文件执行处理	63
Ant 并非脚本语言	75
构建文件授权问题	78
第四章 Ant DataType	81
已定义 DataType	82
XML 属性约定	82
argument DataType	85
environment DataType	88
filelist DataType	90
fileset DataType	93
patternset DataType	96
filterset DataType	98
path DataType	100
mapper DataType	102
第五章 用户编写任务	108
定制任务的需要	109
Ant 的任务模型	110
任务生命期	120
通过分析看示例： jar 任务	124
关于任务的其他内容	143
第六章 用户编写监听者	147
BuildEvent 类	148
BuildListener 接口	150

一个例子：XmlLogger	152
并行问题	156
第七章 核心任务	158
任务总结	159
常用类型和属性	162
工程和目标	165
核心任务参考	167
第八章 可选任务	261
任务汇总	261
可选任务参考	265
附录一 Ant 的未来	351
附录二 Ant 解决方案	355
词汇表	369

序

必须承认，我从没有想过像 Ant 这样一个小小的构建工具能够发展得如此迅速，而且在 Java 开发者社区中会取得如此显著的地位。在我编写第一版 Ant 时，它只不过是有助于解决我所遇到的跨平台构建问题的一个简单工具。现在它已成长壮大起来，而且正得到世界各地成千上万开发人员的使用。其间的奥秘在哪里呢？这样一个小程序何以最终被如此众多的人所采纳呢？也许由 Ant 的发展历程可稍见端倪。

早在加入 Apache 的 CVS 服务器之前，Ant 就已经被编写出来。1998 年中期，我在 Sun 公司负责创建 Java Servlet 2.1 规范以及相应的参考实现。我将此参考实现命名为 Tomcat，它是一种全新的代码基，这是因为原来的参考实现所基于的是 Java Web 服务器的代码，这是一种由 JavaSoft 迁移到 iPlanet 的商业产品。而新的实现必须是 100% 纯 Java 的。

为了使参考实现得到 100% 纯 Java 的证书，即便我们是 Sun 公司的内部人员，而且使用 Java 平台工作，也必须向 Key Labs（一个独立的证书公司）表明它可以在三种不同平台上运行。为了确保 servlet 参考实现能够随处运行，我选择了 Solaris、Windows 和 Mac OS。另外，我不仅希望 Tomcat 能够在这三种平台上运行，而且希望它在所有这三种平台以及 Linux 上还能够构建和开发。我尝试使用了 GNU Make、shell 脚本、批处理文件，以及各种各样其他的工具。每种方

法都存在其自己的问题。这些问题均源于一点，即所有现有的工具都无一例外地借鉴了C程序的构建。将这些做法应用于Java确实也能工作，但是很慢。即使Java程序本身可以很好地完成工作，与Java虚拟机相关的启动开销也相当大。而且，当Make为每个需要编译的文件创建一个新的VM（虚拟机）实例时，编译时间会基于一个工程中的源文件数成线性增长。

我尝试了许多方法来编写一个make文件，从而对于需要重编译的工程，使得其中的所有源文件一次即传递给javac。但是，无论我怎样努力，无论我采用了多少种Make向导工具，仍然无法得到一种能够在多个平台上采用同一方式工作的方法。make文件中的!#\$%#制表符格式实在令我太厌烦了。即便我是Emacs的忠实支持者，我仍然无法忍受一个工具需要用Emacs来编写其文件，以确保不会出现非有意放置的空格（注1）。

那是在欧洲参加完一个会议的返航途中，我终于彻底厌倦了，对于创建在任何环境下都能采用同样方式工作的make文件，我已不打算再去尝试了。我决心“建立”一个自己的工具：这个工具应当能够检查一个工程中的所有Java源文件，将它们与所有已编译的类相比较，并把需要编译的源文件清单直接传给javac。另外，它还应完成其他的一些工作，例如将所有的类填入一个JAR文件中，并复制另外一些文件从而建立该软件的一个可发行版本。为了确保在每个得到支持的平台上都能采用相同方式工作，我决定用Java来编写这个工具。

数小时后，我就有了一个实用的工具。它非常简单和原始，只包括为数不多的类。其中用到了java.util.Properties的功能作为其数据层。它确实可以工作，而且相当不错，我的编译时间下降了一个数量级。回到美国后，我又在Solaris、Linux和Mac OS上对它做了测试，在所有这些平台上它均能很好地工作。当时它最大的问题就是其所能完成的工作仅限于编译文件和复制文件，而且此功能是硬编码的。

注1：据称，make文件格式原来的设计者在第一周之后便认识到这些制表符可能会成为一个问题。但是当时他已经有了数十个用户，因此不想再破坏其兼容性。

数周之后，我向我的朋友 Jason Hunter 展示了这个名为 Ant 的工具（注 2）（之所以如此命名，是因为尽管它是一个小东西，却能够做大事，这与“蚂蚁”有着异曲同工之处），Jason 是 O'Reilly 公司出版的《Java Servlet Programming》（译注 1）一书的作者。Jason 认为，作为一个工具它已像模像样，但并不觉得它会有多重要的作用。当时我正在考虑使用 Java 的反射功能来提供一种简洁的方法对 Ant 的功能加以扩充，从而使程序员可以编写其自己的任务来实现扩展，也就是说，在我提到这种想法之前，Jason 并不认为 Ant 有太大的意义。而在此之后他的疑问顿消，于是我有了第一位 Ant 用户兼 Ant 的热心传播者。Jason 还有一个非凡的能力，他能够很快地在软件的任何一部分中找出 bug，并帮助我解决了不少问题。

有了反射层之后，我又编写了更多的任务，这样 Ant 对 Sun 公司的其他小组也能助一臂之力。不过，构建文件的格式稍显繁琐。特性（property）文件并未确实做到层次式分组，随着任务的引入又带来了目标（任务集合）的概念。我采用了许多不同方法来解决这一问题，不过同样又是在一次从欧洲的返航途中有了解决问题的灵感。这一解决方案即为令“工程 - 目标 - 任务”层次遵循 XML 文档层次结构。以往的反射工作需要将 XML 标签名与任务实现相关联，而该解决方案还可以减轻这项工作的负担。

最后，在横越大洋上空时，我完成了最棒的代码编写。海拔高的位置是否对产生灵感有所帮助，对此我一直感到疑惑。也有可能是到欧洲的旅行行为我带来了创造力，这一点只有更多的实践才能得到验证。

正如我们所知，Ant 已经推出。你在当前使用的各版本 Ant 中所见的一切（包括好的和不好的），都是当时所做决定的结果。可以肯定，尽管自此已经做了许多修改，但是基础仍保持不变。2000 年末，连同 Tomcat 一同加入到 Apache 的 CVS 工具库中的基本上即为此源代码。其后我又转向其他一些工作，主要是作为 Sun 公司 Apache 软件基金会（Apache Software Foundation）的代表，以及从事 XML 规范（如 Sun 公司的 JAXP 和 W3C 的 DOM）方面的工作。

注 2： 另外，ANT 还代表“Another Neato Tool”的字母缩写。我知道这样解释有点傻，但事实确实如此。

译注 1： 本书中文版《Java Servlet 编程》已由中国电力出版社引进出版。

令人难以置信的是，世界各地的人们都开始谈论 Ant。最早发现它的是那些在 Apache 上使用 Tomcat 的人。然后他们开始向他们的朋友介绍，而这些朋友又向他们的其他朋友推广，如此继续。此时，更多的人们已经了解到了 Ant，并且使用率甚至多于 Tomcat。在 Apache 上开发和使用 Ant 的人已经形成了一个强大的群体，而在此发展道路上也已经对该工具做了许多改进。人们现在使用它来构建各种形式的工程，从很小的应用到极其庞大的 J2EE 应用都包括在内。

我得知 Ant 被载入史册是在 2001 年的 JavaOne 大会上。当时我正在参加一个主题介绍，其中要演示一个主要数据库软件公司所推出的新的开发工具。演示者展示了通过在方框之间进行连线可以何等容易地设计软件，然后单击了构建 (Build) 按钮。在显示屏幕上展现的居然是每个 Ant 用户所熟识并视为基础的那些方括号。对此我是如此震惊，同时简直无法相信。

Ant 用户的数目仍在增长。显然我所开发的这个小东西已经为全世界 Java 开发人员所共享。而且还不仅仅限于 Java 开发人员。最近我还发现了一个 NAnt，这是 Ant 思想针对 .NET 开发的一种实现（注 3）。

要是我原先知道 Ant 会取得如此巨大的成功，最初可能就会花费更多时间来增强其功能，使它不仅仅像现在这样只作为一个简单工具。不过，这样也许会将最初使之脱颖而出的一些特性埋没。Ant 可能会设计得过于复杂。如果我花费太多时间来使之能够完成超出我的需求之外的工作，它就有可能成为一个过于庞大的工具，而且也可能过于“笨重”以至于难以使用。这种情况在软件行业屡见不鲜，特别是目前所提出的许多 Java API 更是如此。

Ant 成功的秘诀可能就在于它原本没有打算成功。它只是针对许多人所遇到的一个明显问题而提出的一个简单解决方案。作为解决这一问题的幸运儿，我感到颇为自豪。

你手中的这本书将指导你如何使用当前的 Ant。Jesse 和 Eric 将教你如何高效地使用 Ant，以及如何对其扩展，并告诉你将如何使用各种不同任务（包括内置任务

注 3：可在 <http://nant.sourceforge.net/> 上找到 NAnt。

和广泛使用的可选任务)。另外,对于某些Ant设计决策,作者还将提供一些技巧从而避免其本身的缺陷。

在由他们接手为你提供有力的指导之前,我还有最后一句话要告诉你:只要可能,就应自行解决自己的切肤之痛。如果已有的工具不能满足你的需求,就应当另找一个工具。如果不存在这样的工具,那么就创建一个。而且要确保与大家共享。其他数以千计的人可能有着与你类似的难题。

—— James Duncan Davidson
旧金山, 加利福尼亚州, 2002 年 4 月

前言

对于许多基于 Java 的工程而言，对所有 JavaTM 源文件进行编译已不再是构建这些工程所需的惟一步骤。对于典型的 HelloWorld 程序、书中的例子以及简单 applet，源文件的编译就已经足够了。但是还有一些复杂的基于 Java 的工程，如 Web 应用或基于 Swing 的程序（如 JBuilder），要求做更多的工作。必须根据资源控制得到最新的资源；未由 Java 编译器自动处理的依赖关系也需要得到管理；各种类必须被捆绑并交付到多个位置，有时是作为 JAR 或 WAR 文件进行交付；某些 Java 技术，诸如 EJB（Enterprise Java Bean，企业 Java Bean）和 RMI（Remote Method Invocation，远程方法调用）类，则需要单独的编译和代码生成步骤，这些均并非由 Java 编译器完成。shell 脚本和 GNU Make 通常是完成这些任务的首选工具，从“完成工作”的角度来说，这些工具可以很好地达到目的，但从长远来看，它们却是不太好的选择。

虽然 GNU Make 可以提供许多功能，但在易用性方面，却存在许多缺陷。makefile 有其自己的语言语法，这就要求编写 makefile 的人具备此项专门的知识。GNU Make 还缺乏平台无关性，因此对于同一个 makefile，需要维护和分发多个版本（每一个版本对应于一个目标平台）。由于 shell 脚本和 GNU Make（要记住，GNU Make 只是现有 shell 基础上的一个语言扩展）所固有的性质，使得对于任何一个非专家级用户来说，在操作系统之间（甚至 shell 之间）进行迁移都是很困难的，

甚至是不可能的。如果使用 GNU Make，那么对于当前基于 Java 的工程，遵循这种做法所需的时间和维护开销是相当高的，这种情况可谓屡见不鲜。

Sun 公司对其所有 SDK 工具均提供了 Java 版本。诸如 *javac* 等可执行程序只是执行 Java 代码的包装器。其他厂商的工具，如 BEA 公司的用于 WebLogic 的 EJB 编译器、JUnit 和 Jakarta 工具以及库均采用 Java 编写。GNU Make 只能从命令行调用可执行程序。例如，为了调用一个 Java 类，GNU Make 必须使用 *java* 命令来调用 JVM，再将类名作为命令行参数进行传递。Make 不能在程序中使用 Java 工具的任何库，如异常和错误对象等，而这些库则允许更为灵活的构建过程。对于用 Java 编写的工具（如 WebLogic 的 *ejbc* 编译器），它们可以与同一 JVM 中可用的其他对象（如 Ant 任务对象）共享来自于异常和错误的信息。较之于命令行返回代码和事后的错误消息字符串解析而言，这样做可以改进构建过程。

GNU Make 所存在的问题和用 Java 编写构建工具的可能性促使 James Duncan Davidson 写出了 Ant。Ant 将 Java 编译器作为一个类运行，而不是来自命令行的一个调用。保持在 JVM 中则允许用特定的代码处理错误，还可对 Sun 公司通过其编译器提供的结果采取操作。Ant 用 XML 作为其构建文件语法，因此只是会增强开发人员和工程管理人员的技能，而不是令其因学习新知识而过于疲劳。Ant 对构建过程加以扩展，使其不只是运行程序，因此称之为构建环境（environment）比构建工具更为恰当。

本书的结构

本书涵盖了初识 Ant 的人所需的全部知识。对于 Ant 专家来说，本书可作为一本参考书，其中提供了 Ant 核心任务的详细定义；讨论了 Ant 的主要特性；提供了用 Ant 来管理工程的一些最佳实践；还解释了某些 Ant 问题的解决方法。

第一章，Ant 入门。这一章逐步介绍了一个非常基本的 Ant 构建文件示例，其目的是使你能够很快上手。我们展示了如何创建目录、编译代码以及生成一个 JAR 文件，但是对于每一项工作如何进行的具体细节则未做深究。这一章还包括了 Ant 命令行使用的详细信息。最后得到一个构建文件的大致轮廓，以此作为初始模板。

第二章，安装和配置。这一章介绍了如何得到、安装以及在 Windows 和 Unix 平台上配置 Ant。我们列出了在这些开发平台上发现的一些缺陷，并提供了解决方法和解决方案。

第三章，构建文件。这一章介绍了一个示例工程上下文中的 Ant 构建文件例子。我们对此构建文件的主要部分和结构加以了剖析和描述，还对一些问题做了解释，如 Ant 引擎的一般流程和 Ant 使用 XML 的好处等，并强调了构建文件的主要部分。

第四章，Ant DataType。这一章详细描述了各种 Ant DataType。尽管在前面的章节中已经用到过 DataType，但这里才对其做深入研究。我们介绍了如何使用环境变量和如何传递命令行参数以进行处理，还说明了如何利用各种文件和模式。

第五章，用户编写任务。这一章涵盖了 Ant 的一个最佳功能，即能够对 Ant 进行扩展。由此编写扩展的功能，就能够处理特定工程可能需要的任何事情。作为一个意外收获，你还可以在将来的工程中重用这些任务，对于你所付出的努力，你的收益将并不仅限于最初的实现。你的任务还可以得到共享并公开展示，这样即使是你不认识的人也能够由于你的工作而受益。

第六章，用户编写监听者。这一章介绍了如何设计和开发你自己的构建事件监听者。据此，你就可以编写类来完成一些操作，这些操作所基于的是与构建文件处理相关的流程。操作的范围很广，从特定复杂任务完成时的发送邮件，到将同样的事件重定向到一个集中的“构建监听框架”等均属于此范畴。正如用户编写任务一样，可能的用户编写监听者也是无法计数的。这一章还包括对监听者的进一步扩展：用户编写日志工具（logger）。利用这些日志工具，可以改进甚至替换 Ant 默认的日志系统。

第七章，核心任务。这一章是对全部核心 Ant 任务的详尽参考。对于每个任务，在此都可以看到一个描述、一组支持此任务的 Ant 版本以及对所有任务属性的定义。从这里还可以得到有关任务使用的有用示例。

第八章，可选任务。这一章的形式类似于第七章，是对 Ant 丰富的可选任务所提供的参考。

附录一, Ant 的未来。这一部分所讨论的正如其标题所示。我们介绍了 Ant 未来的方向和将出现的新功能, 另外还提供了有关建议, 你可通过采取所建议的步骤来避免使用很快就要过时的功能。

附录二, Ant 解决方案。Ant 可用于解决不同的构建问题, 这一部分则深入研究了其中所采用的更为常见的方法。除此以外, 我们还谈到如何使用带有级联工程结构的构建文件。这些工程结构有一个主工程目录以及许多子工程子目录。每个子工程包括其自己的构建文件, 而且主工程有一个主构建文件, 能够构建所有子工程。

本书的读者对象

本书主要面向 Java 开发人员, 特别是那些开发企业级 Java 应用的人员。另外, 如果某些人需要一个健壮的构建工具, 并要求这种工具不只是调用命令行编译器和实用程序, 那么这本书对其尤为适用。对于大型工程的构建管理人员 (以及负责构建管理的工程管理人员) 来说, 这本书也很有用。

预备知识

对于本书的大多数内容, 仅需要对 Java 和 XML 有基本的了解即可。有关为 Ant 编写扩展的章节则要求你还应对 Java 继承和接口有明确的理解。Ant 最适合用于构建和部署基于 Java 的工程。有些 Ant 任务可以提供功能来编译和运行其他语言, 如 Perl、Python、C 和 C#, 尽管这些任务也可用, 但本书所强调的是 Java 在 Ant 中的使用。

平台和版本

作为 Apache 的 Jakarta 项目下的一个开源项目, Ant 得到“每晚 (nightly)”代码修订和构建。这些每晚构建即创建了 Ant 的“非稳定版本”。主要维护人员时常会以一个每晚构建作为发布版, 并宣布其功能和稳定性。在写这本书时, 共有 5 个

此类版本，即 1.1、1.2、1.3、1.4 和 1.4.1。这本书主要关注 1.4.1 版本，它于 2001 年 10 月发布。有些任务（如 `copydir`）在 1.2 版本以后已经不用，但在本书中仍会涉及，因为它们并未从核心任务表中被完全去除。

本书英文排版约定

本书英文采用以下排版约定：

斜体 (*italic*)

用于 Unix 和 Windows 命令、文件名和目录名、强调以及技术术语的首次使用。

等宽字体 (constant width)

用于代码示例以及展示文件内容，还用于正文中出现的 Java 类名、Ant 任务名、标签、属性名和环境变量名。

等宽斜体 (constant width *italic*)

用于语法描述以指出用户定义的项。

等宽黑体 (constant width bold)

对于同时显示输入和输出的例子，此字体用于用户的输入。

术语

为一致起见，本书中将 Ant 指令文件称为构建文件 (buildfile)。在其他与 Ant 相关的论坛和文献中，你可能还会看到诸如 `build.xml` 和 `antfile` 的提法。这些词可以互换，不过构建文件更为合适。

在谈到 XML 时，我们采用了以下约定，即标签 (tag) 指构建文件中由括号括起来的标记。例如，`<path>` 是一个标签。元素 (element) 则指标签及其孩子（如果有孩子的话）。以下 XML 记法即为 `<path>` 元素的一个例子。标签和元素之间

的差别在于，标签仅指 `<path>`，而元素则指从 `<path>` 到 `</path>` 间的全部内容。

```
<path>
  <fileset dir="src">
    <includes name="**/*.java"/>
  </fileset>
</path>
```

XML 元素和标签定义了 Ant 的任务和构建文件中的 *DataType*。任务 (task) 完成操作，并相当于 Ant 引擎中的模块部件。*DataType* 则为 Ant 引擎定义了复杂的数据分组，通常是路径或文件集。

文件名和路径约定

Ant 是一个 Java 程序，并采纳了 Java “不明确的” 文件系统观点。在运行时，Ant 会检查底层 JVM 所提供的路径分隔符和目录分隔符，并使用这些值。它会成功地解释构建文件中的 “;” 或 “:”。例如，在一个 Unix 机器上运行时，Ant 将路径 `dir;dir\subdir`（注意转义符 “\”）正确地解释为 `dir:dir/subdir`。在同一个值类型中，分隔符的使用必须一致，对于字符串 `dir;dir/subdir`，其中包括一个 Windows 的路径分隔符 (;)，又包括一个 Unix 的目录分隔符 (/)，因此不是一个好的形式。在本书中，Unix 和 Windows 的文件路径约定都会出现在例子中，以此来强调一点，即 Ant 并不关心你所用的平台。

Ant 不处理跨平台的驱动器盘符。在 Ant 路径元素中使用驱动器盘符将限制构建文件在 Windows 环境下的使用。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

你还可以给我们发电子消息。要加入邮件列表或索要书目，请发电子邮件到：

info@oreilly.com

要询问技术问题或对本书发表评论，请发电子邮件到：

bookquestions@oreilly.com

info@mail.oreilly.com.cn

我们为本书建有一个网站，在此你可以找到一些例子和勘误表（包括以前报告的错误及公开的修正），可由以下网页访问：

<http://www.oreilly.com/catalog/anttdg/>

要了解本书和其他书的信息，请访问 O'Reilly 网站：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

Jesse 的致谢

我要先感谢我的妻子 Melissa 以及两个孩子 Natalie 和 Peter，在许多个周末他们都

无法享受天伦之乐。没有他们的爱和支持，我绝不可能完成这本书。还要感谢 Keyton Weissinger，是他最初鼓励我写这本书的。在 Eric 和我写这本书的全过程中，整个 Ant 社区在技术支持上起到了很重要的作用。我要特别感谢 Stefan Bodewig 和 Conor MacNeil，是他们在百忙之中帮助我理解了 Ant 的一些高级函数，并乐意提供其信息，在此向他们所付出的时间表示感谢。

另外，我还要感谢我的技术审校：Diane、Dean、Jeff 和 Paul。由于他们的贡献使这本书增色不少。我总是告诫自己“批评只会使这本书做到更好”，而这也正是他们所做的。

最后，我要感谢位于乔治亚州 Roswell 的 Caribou 咖啡店的员工，每个周六，我都会在这家咖啡店呆上 4 到 8 个小时，并占用他们的一张桌子（还有电），对此他们都能容忍。因为有可口的咖啡，还有他们这些友好的人，使得在那里写书成为一件快意的事情。

Eric 的致谢

我要感谢我的家人，有了他们的帮助，这本书才得以写成。感谢我的妻子 Jennifer，谢谢你忍受了我在写这本书时的那些夜晚和周末。感谢我的儿子 Aidan，无论我有多少工作要做，我都会找时间带你去动物园。我爱你们。

还要感谢每一位技术审校为这本书做出的贡献，Diane Holt、Dean Wette、Jeff Brown 和 Paul Campbell 从他们的业余安排中挤出了大量时间来帮助完成这本书，对此我表示深切的谢意。

第一章

Ant 入门

你可能已经下载并安装了 Ant，现在打算看一个例子来了解它是如何工作的。如果是这样，那么这一章正是为你准备的。在此我们会介绍一个非常基本的构建文件示例。其后将对 Ant 的命令行选项做全面的描述。如果你想先按部就班完成安装过程，那么可以先跳到第二章，然后再来阅读这一章。

我们不打算在这一章中对构建文件的每一处细节都加以解释。请参见第三章中一个更复杂的例子。

文件和目录

对于这个例子，我们先从如图 1-1 所示的目录和文件结构开始介绍。阴影框表示文件，未加阴影的框表示目录。

注意： 可以由本书网页下载此例，网址为 <http://www.oreilly.com/catalog/anttdg/>。

Ant 的构建文件 *build.xml* 位于工程的基目录中。这是一般情况，不过你完全可以使用其他的文件名，或者将此构建文件放在其他的位置上。*src* 目录包括了组织为一种普通包结构的 Java 源代码。一般来说，这些源文件的内容并不重要。不过需要指出，*PersonTest.java* 是一个单元测试，生成的 JAR 文件中不包括此文件。

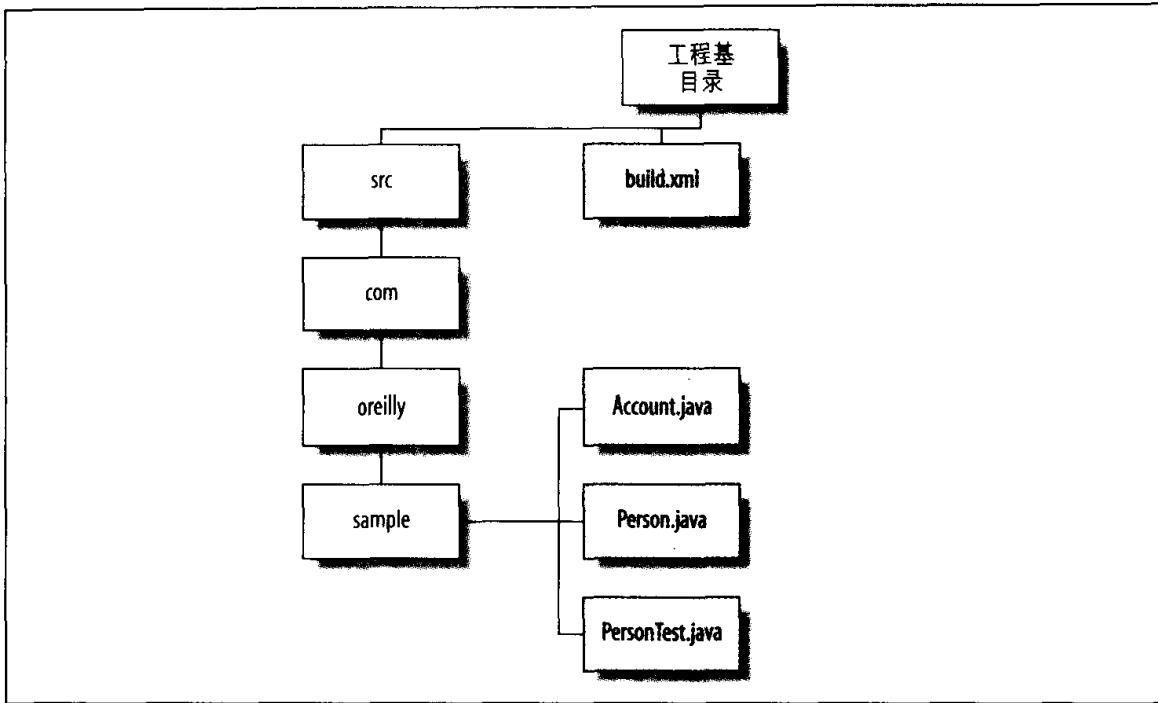


图 1-1：示例构建文件的起点

我们这个示例构建文件将使 Ant 创建一个目录树及一些文件，如图 1-2 虚线阴影框中所示。它还会编译 Java 源代码、创建 *oreilly.jar* 并且提供一个“clean”目标来删除所有生成的文件和目录。

下面来看使这一切成为可能的构建文件。

Ant 的构建文件

Ant 的构建文件是用 XML 编写的。例 1-1 显示了此例中完整的 Ant 构建文件。与大多数实际应用中的构建文件相比，它要简单一些，但确实能够说明几乎每个 Java 工程所需的一些核心概念。

例 1-1：build.xml

```
<?xml version="1.0"?>

<!-- build.xml - a simple Ant buildfile -->
<project name="Simple Buildfile" default="compile" basedir=". " >
```

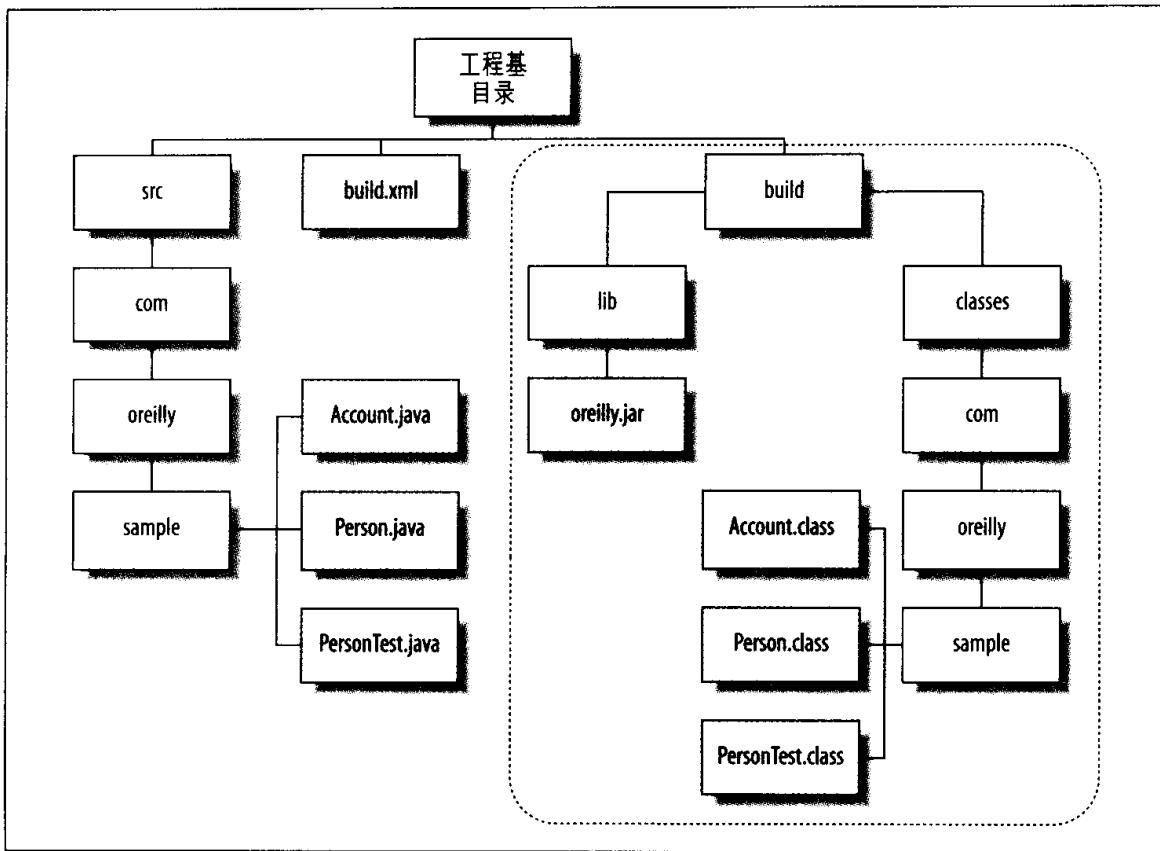


图 1-2：示例构建文件创建的目录和文件

```
<!-- The directory containing source code -->
<property name="src.dir" value="src"/>

<!-- Temporary build directories -->
<property name="build.dir" value="build"/>
<property name="build.classes" value="${build.dir}/classes"/>
<property name="build.lib" value="${build.dir}/lib"/>

<!-- Target to create the build directories prior to the -->
<!-- compile target. -->
<target name="prepare">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${build.classes}" />
    <mkdir dir="${build.lib}" />
</target>

<target name="clean" description="Removes all generated files.">
    <delete dir="${build.dir}" />
</target>
```

```
<target name="compile" depends="prepare"
       description="Compiles all source code.">
  <javac srcdir="${src.dir}" destdir="${build.classes}"/>
</target>

<target name="jar" depends="compile"
       description="Generates oreilly.jar in the 'dist' directory.">
  <!-- Exclude unit tests from the final JAR file -->
  <jar jarfile="${build.lib}/oreilly.jar"
       basedir="${build.classes}"
       excludes="**/*Test.class"/>
</target>

<target name="all" depends="clean,jar"
       description="Cleans, compiles, then builds the JAR file."/>

</project>
```

XML 注意事项

Ant 构建文件是 XML 文件，可以用任何文本编辑器来创建。在创建你自己的构建文件时，需要谨记以下几点：

- 第一行为 XML 声明。如果有，则必须出现在 XML 文件的第一行；在其之前不允许出现空行。实际上，即便在 `<?xml` 之前只有一个空格也会导致 XML 解析程序失败。
- XML 对于大小写、引号以及正确的标签语法很挑剔。如果其中任何一项不正确，Ant 就会失败，这是因为其底层的 XML 解析程序会失败。

以下是一个错误的例子，如果将 `</project>` 结束标签错误地键入为 `</Project>`，则会出现这个错误：

```
Buildfile: build.xml

BUILD FAILED

C:\antbook\build.xml:41: Expected "</project>" to terminate
element starting on line 4.

Total time: 2 seconds
```

构建文件描述

我们的构建文件包括多个 XML 注释、所需的 `<project>` 元素以及许多特性 (property)、任务 (task) 和目标 (target)。`<project>` 元素为这个工程建立工作目录 “.”。这就是包含此构建文件的目录。它还指定了默认的目标，在此为 “compile”。稍后在描述如何运行 Ant 时，即会明确默认目标的目的。

特性定义允许我们避免在构建文件中将目录名硬编码。这些路径通常都是相对于 `<project>` 元素所指定的基目录。例如，以下标签设置了源文件目录名。

```
<property name="src.dir" value="src"/>
```

其后，这个构建文件定义了多个目标。每个目标都有一个名字，如 “prepare”、“clean” 或 “compile”。开发人员从命令行调用 Ant 时要结合这些目标名。每个目标定义了 0 个或多个依赖关系 (dependency)，同时还有一个可选的 `description` 属性 (attribute)。依赖关系指定了在当前目标执行之前，Ant 必须先执行的目标。例如 “prepare” 必须在 “compile” 之前执行。对于 Ant 将显示的目标，`description` 属性为其提供一个适合阅读的描述。

在目标中存在任务，这些任务完成构建中的具体工作。Ant 1.4.1 内置有 100 多个核心的和可选的任务；所有任务都将在第七章和第八章做详细描述。这些任务完成从创建目录到构建结束时播放音乐（注 1）等大量功能。

运行 Ant

我们假设 Ant 已经正确安装。如果还有疑问，可以先阅读第二章，并保证一切可以正常运行起来。

注 1： 请参见第八章中的 `sound` 任务。

示例

为了执行默认目标 `compile` 中的任务，应从包括这个示例 `build.xml` 文件的目录键入以下命令：

```
ant
```

Ant 将打开一个默认的构建文件，即 `build.xml`，并执行此构建文件的默认目标（在这种情况下为 `compile`）。假设目录为 `antbook`，那么应当可以看到以下输出：

```
Buildfile: build.xml

prepare:
  [mkdir] Created dir: C:\antbook\build
  [mkdir] Created dir: C:\antbook\build\classes
  [mkdir] Created dir: C:\antbook\build\lib

compile:
  [javac] Compiling 3 source files to C:\antbook\build\classes

BUILD SUCCESSFUL

Total time: 5 seconds
```

Ant 运行时，会显示出每一个所执行的目标的名字。如示例输出所示，Ant 先执行了 `prepare`，其后才执行 `compile`。这是因为 `compile` 是默认目标，而它对 `prepare` 目标存在一个依赖关系。Ant 打印出每个任务名，并括以方括号，另外还显示了特定于各任务的其他消息。

注意： 在这里的示例输出中，`[javac]` 为 Ant 任务名，而不必为 Java 编译器的名字。例如，如果你所用的是 IBM 的 Jikes，那么仍会显示 `[javac]`，因为这才是在后台运行 Jikes 的 Ant 任务。

若调用 Ant 时未指定一个构建文件名，Ant 会在当前的工作目录中搜索一个名为 `build.xml` 的文件。你不必受此默认文件的限制，对于构建文件可以使用你所希望的任何名字。例如，如果要调用我们的构建文件 `proj.xml`，则必须键入以下命令：

```
ant -buildfile proj.xml
```

还可以显式地指定一个或多个要运行的目标。例如，可以键入 **ant clean** 来删除所有生成的代码。如果我们的构建文件名为 *proj.xml*，而且我们希望执行 **clean** 目标，则应键入 **ant -buildfile proj.xml clean**。其输出如下所示：

```
Buildfile: proj.xml

clean:
[delete] Deleting directory C:\antbook\build

BUILD SUCCESSFUL

Total time: 2 seconds
```

还可以用一条命令执行多个目标：

```
ant clean jar
```

它会在调用 **clean** 目标后调用 **jar** 目标。由于在此示例构建文件中存在目标依赖关系，所以 Ant 会按以下顺序执行目标： **clean**、**prepare**、**compile** 和 **jar**。**all** 目标利用了这些依赖关系，使我们可以通过键入 **ant all** 来完成清除和重新构建：

```
<target name="all" depends="clean,jar"
       description="Cleans, compiles, then builds the JAR file." />
```

all 依赖于 **clean** 和 **jar**，而 **jar** 进一步依赖于 **compile**，**compile** 则依赖于 **prepare**。这样，一条简单的命令 **ant all** 即可按适当的顺序完成所有目标。

获得帮助

你可能注意到了，有些目标包括有 **description** 属性，而另外一些则没有。这是因为 Ant 会区别主目标和子目标。包括有描述的目标为主目标，而没有描述的则被认为是子目标，尽管在表述上有所区别，主目标和子目标的行为却是相同的。从这个工程的基目录键入 **ant -projecthelp** 可得到如下输出：

```
Buildfile: build.xml
Default target:
```

```
compile Compiles all source code.

Main targets:

all      Cleans, compiles, then builds the JAR file.
clean    Removes all generated files.
compile  Compiles all source code.
jar      Generates oreilly.jar in the 'dist' directory.

Subtargets:

prepare

BUILD SUCCESSFUL

Total time: 2 seconds
```

对于包括有数十个目标的大型工程来说，这个工程帮助功能尤其有用，不过前提是你要花时间来增加有意义的描述。

要得到 Ant 命令行语法的总结，可键入 **ant -help**。你将看到有关 Ant 命令行参数的一个简要描述，稍后将对此加以介绍。

Ant 命令行参考

从命令行调用 Ant 的语法如下所示：

```
ant [option [option...]] [target [target...]]

option := {-help
           |-projecthelp
           |-version
           |-quiet
           |-verbose
           |-debug
           |-emacs
           |-logfile filename
           |-logger classname
           |-listener classname
           |-buildfile filename
           |-Dproperty=value
           |-find filename}
```

语法元素说明如下：

-help

显示描述 Ant 命令及其选项的帮助信息。

-projecthelp

显示包含在构建文件中的、所有用户编写的帮助文档。即为各个 `<target>` 中 `description` 属性的文本，以及包含在 `<description>` 元素中的任何文本。将有 `description` 属性的目标列为主目标（“Main target”），没有此属性的目标则列为子目标（“Subtarget”）。

-version

要求 Ant 显示其版本信息，然后退出。

-quiet

抑制并非由构建文件中的 `echo` 任务所产生的大多数消息。

-verbose

显示构建过程中每个操作的详细消息。此选项与 `-debug` 选项只能选其一。

-debug

显示 Ant 和任务开发人员已经标志为调试消息的消息。此选项与 `-verbose` 只能选其一。

-emacs

对日志消息进行格式化，使它们能够很容易地由 Emacs 的 `shell` 模式 (`shell-mode`) 所解析；也就是说，打印任务事件，但并不缩排，在其之前也没有 `[taskname]`。

-logfile filename

将日志输出重定向到指定文件。

-logger classname

指定一个类来处理 Ant 的日志记录。所指定的类必须实现了 `org.apache.tools.ant.BuildLogger` 接口。

-listener classname

为Ant声明一个监听类，并增加到其监听者列表中。在Ant与IDE或其他Java程序集成时，此选项非常有用。可以阅读第六章以了解有关监听者的更多信息。必须将所指定的监听类编写为可以处理Ant的构建消息接发。

-buildfile filename

指定Ant需要处理的构建文件。默认的构建文件为*build.xml*。

-Dproperty=value

在命令行上定义一个特性名-值对。

-find filename

指定Ant应当处理的构建文件。与*-buildfile*选项不同，如果所指定文件在当前目录中未找到，*-find*就要求Ant在其父目录中再进行搜索。这种搜索会继续在其祖先目录中进行，直至达到文件系统的根为止，在此如果文件还未找到，则构建失败。

构建文件轮廓

下面是一个通用的构建文件，它很适合作为一个模板。构建文件包括*<project>*元素，以及其中嵌套的*<target>*、*<property>*和*<path>*元素。

```
<project default="all">
  <property name="a.property" value="a value"/>
  <property name="b.property" value="b value"/>

  <path id="a.path">
    <pathelement location="${java.home}/jre/lib/rt.jar"/>
  </path>

  <target name="all">
    <javac srcdir=". ">
      <classpath refid="a.path"/>
    </javac>
  </target>
</project>
```

关于构建文件有几点需要记住：

- 所有构建文件都要有`<project>`元素，而且至少有一个`<target>`元素。
- 对于`<project>`元素的`default`属性并没有默认值。
- 构建文件并不一定要被命名为`build.xml`。不过`build.xml`是Ant要搜索的默认文件名。
- 每个构建文件只能有一个`<project>`元素。

继续学习

在本章中，我们只是触及了一些皮毛，不过对于Ant的构建文件是什么样子，以及如何运行命令行`ant`脚本，以上内容确实可使你有所认识。前面已经提到，第三章提供了一个更为复杂的例子，并对每一步要做的工作有更多的解释。

第二章

安装和配置

这一章将介绍在哪里可以得到 Ant，并会解释不同发布（distribution）之间的区别，还涵盖了最常见的安装情况。作为一个可移植的 Java 应用，Ant 在许多不同平台上均能一致地工作。相应于特定于平台的 *make* 实用程序，既然 Ant 作为其替代产品，那么这一点自然不足为奇。大多数差别体现在 Ant 的启动脚本中，它在 Windows 和 Unix 系统上有所不同，这是可以理解的。一旦 Ant 得到了安装和配置，它就能够完成绝妙的工作，从而完全屏蔽了不同平台之间的差别。

发布

Ant 是来自 Apache 软件基金会 (Apache Software Foundation) 的一个开源软件，其二进制形式和源代码形式均有提供（注 1）。由 Ant 主页（网址为 <http://jakarta.apache.org/ant/>）即可得到；你可以选择版本构建（release build）或每晚构建（nightly build）。为了方便安装，为 Unix 和 Windows 系统提供了不同的发布。

每晚构建列表的直接链接为 <http://jakarta.apache.org/builds/jakarta-ant/nightly/>。每晚构建包括最新的 bug 修正和功能，并且会不断调整。大多数软件开发小组都应优先选择这种构建，而不是 Ant 的版本构建，其二进制发布可由以下 URL 获得：

注 1：对于 Apache Software License 的情况，可访问 <http://www.apache.org/licenses/LICENSE/>。

<http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/>

二进制发布。

<http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/src/>

相应于当前二进制发布的源代码发布（注 2）。

注意：要得到 Ant 的早期发行版，只需将这些 URL 中的版本号代之以 1.1、1.2 或 1.3 即可。

每个目录都包括 *.tar.gz* 文件（面向 Unix 用户）、*.zip* 文件（面向 Windows 用户）、*.asc* 文件和一个 *.jar* 文件。*.asc* 文件包括 PGP 签名，它对于确定发布的真实性很有用。通常情况下，你可以很安全地忽略这些文件。*jakarta-ant-1.4.1-optional.jar* 文件包含了 Ant 的可选任务，在本章后面将对此加以介绍。建议你下载此文件。

安装

无论是何种平台，要安装 Ant，第一步都是要下载此软件。可以将这些文件下载到一个临时目录中，然后再解压至任何所需的目录下。下载步骤完成之后，取决于你下载的是二进制发布还是源代码发布，其后的过程将有所区别。

警告：Windows 用户应当避免在目录中出现空格，如“Program Files”，因为这会导致所提供的批处理文件出现问题。

Ant 文档中警告不要在 Java 的 *lib/ext* 目录中安装 Ant 的 JAR 文件（注 3）。如果这样做，那么对于一些 Ant 任务，很可能遇到类加载问题。相反，应当将 Ant 的 JAR 文件放在 Ant 的发布目录中。

注 2：最新的 Ant 源代码可以由一个可公开访问的 CVS 仓库得到。如果你需要最近提交的 bug 修正和功能，或者对于组成 Ant 的文件，希望查看对其所做调整的全部历史情况，那么获得最新的源代码就很有用。关于如何访问此仓库的详细指令，请参见 Ant 的网站。要注意没有人担保最新的源代码的运行甚至是对其编译！

注 3：Java 的 *lib/ext* 目录用于放置 Java 的“可选包”，它们可以扩展核心 Java 平台的功能。Sun 公司的 JDK 中所包括的文档对此做了详细解释。

Ant 并没有提供一个安装程序，它可以在你所选择的任意位置上运行来复制文件和目录。表 2-1 列出了在你的主 Ant 目录下最终创建的目录。

表 2-1: Ant 提供的目录

目录	描述
bin	用于运行 Ant 的批处理文件、Perl 脚本以及 shell 脚本
docs	Ant 文档
lib	Ant 运行所需的库
src	Ant 的源代码。仅在源代码发布中提供 ^a

a: 在 Ant 1.3 之前，二进制发布中也包括源代码。

二进制安装

我们将首先介绍二进制安装，它可以满足大多数用户的要求。“二进制”(binary)这个词意味着所有一切都已经得到编译，并且打包到 JAR 文件中，这是为了方便执行，你无需再由源代码编译 Ant。本章后面将介绍的源代码发布则必须在使用前编译。

二进制发布的安装可分为以下 4 个快捷的步骤：

1. 将发布解压 (unzip 或 untar) 至所需的目录。
2. 设置 ANT_HOME 环境变量以指向此位置。
3. 设置 JAVA_HOME 环境变量以指向 JDK 的位置。
4. 在系统的 PATH 环境变量中增加 ANT_HOME/bin。

由于文件名的限制，Unix 发布必须用一个与 GNU 兼容的 tar 版本展开；Solaris 和 Mac OS/X 中所包括的 tar 实用程序将无法工作。GNU tar 可以从 <http://www.gnu.org/software/tar/tar.html> 获得。在 OS X 下，你可以使用 gnutar 命令。展开 Ant 1.4.1 发布的命令为：

```
tar xzvf jakarta-ant-1.4.1-bin.tar.gz
```

一旦安装，再键入 **ant -version** 来验证 Ant 是否位于此路径上。如果工作正常，则表示 Ant 已经正确安装。你可以看到如下输出：

```
Ant version 1.4.1 compiled on October 11 2001
```

与其他 Apache Java 工具类似，Ant 依赖于一些关键的环境变量。你在运行 *ant* 命令时，实际上运行的是在 *ANT_HOME/bin* 目录中所找到的一个 shell 脚本或批处理文件。这也是为什么 PATH 环境变量必须包括此目录的原因。

ant 脚本要使用 *ANT_HOME* 和 *JAVA_HOME* 环境变量，以此来配置运行 Ant 的 JVM 所用的 *CLASSPATH*。如果这些变量没有设置，启动脚本就会推测正确的值，这取决于操作系统的限制。例如，在 Windows NT/2000/XP 上，*ant.bat* 使用 %~dp0 批处理文件变量来确定其包含目录。如果需要，则会把 *ANT_HOME* 的默认值设置为父目录。这一技巧对 Windows 9x 则不能奏效，因为其中不支持 %~dp0。正确地设置 *ANT_HOME* 和 *JAVA_HOME* 是避免出现问题的最佳方法（还要考虑到 *CLASSPATH*，这将在本章稍后讨论）。

Ant 1.1 和 1.2 版本（不包括以后版本）的二进制发布包括有源代码，这对跟踪 bug 以及学习如何编写任务是很有用的。不过，这些二进制发布却不包括构建 Ant 所必需的构建文件和脚本。为此，必须下载稍后将介绍的源代码发布。但是在讨论这个内容之前，有必要先谈谈可选任务。

可选任务安装

下载 Ant 时，还要确保下载并安装了可选任务 JAR 文件。可以在 Ant 网站的二进制发布目录中找到它。在 1.1 和 1.2 版本中，其名为 *optional.jar*。对于 1.3 和 1.4.1 版本，则分别改名为 *jakarta-ant-1.3-optional.jar* 和 *jakarta-ant-1.4.1-optional.jar*。要进行安装，需要下载合适的可选任务 JAR 文件，并将它复制到你的 *ANT_HOME/lib* 目录中。

在许多情况下，可选任务需要另外的库和程序才能工作。例如，*junit* 任务需要

junit.jar (注 4)，那么你必须在运行 Ant 之前将此文件复制到 `ANT_HOME/lib`，或者将它增加到你的 `CLASSPATH` 环境变量中。有些时候，由于许可限制，Apache 无法发布这些库。其他任务之所以被标记为可选的，可能是因为它们要处理特定工具，而这些工具只有少数人使用（或者是专用的）。不过仍存在许多有帮助的可选任务，而且大多数开发小组都会用到它们，所以，如上安装可选任务 JAR 文件是明智之举（注 5）。

源代码安装

安装 Ant 的源代码发布比安装二进制发布要稍微多做一些工作。自然第一步仍是下载和解压此发布。通常你希望将这些源文件放在与任何现有 Ant 安装所不同的一个目录中。其次，确保 `JAVA_HOME` 指向 JDK 发布。对于二进制安装，你还应该设置 `ANT_HOME` 并更新你的 `PATH`。

准备可选任务

你现在必须确定需要编译哪些可选任务。Ant 试图编译所有的可选任务，但会忽略那些不要编译的任务。为了使一个可选任务成功地进行编译，必须将所需的 JAR 文件加到 `CLASSPATH` 中，或者将它们复制到 Ant 的 `lib` 目录中。一旦完成这些工作，就可以继续进行构建。同样，针对每个可选任务的 Ant 文档中都指出了需要哪些库。

构建 Ant 二进制发布

现在你已经准备好要编译 Ant 了。如果使用的是 Ant 1.3 或 1.4.1 版本，则由源代码发布目录键入以下命令：

注 4： 在这种特定情况下，Ant 可选任务 JAR 文件包括 `org.apache.tools.ant.taskdefs.optional.junit.JUnitTask`，它实现此任务本身。JUnitTask 进一步依赖于 *junit.jar* 中所找到的文件，*junit.jar* 与 Ant 分开发布。

注 5： 对于可选任务 JAR 文件要求的最新列表，请参考 Ant 发布所带的用户手册。查找“Installing Ant”（安装 Ant）一节，再查找“Library Dependencies”（库依赖关系）标题。

```
build -Ddist.dir=destination_directory dist (Windows)
build.sh -Ddist.dir=destination_directory dist (Unix)
```

build 脚本会在指定的目标目录中创建一个完整的二进制发布。若未指定目标目录，则 *dist.dir* 默认为 *build*。

除非已经安装了所有的可选任务 JAR 文件，否则你可能会看到大量有关缺少导入库和类的警告。可以放心地忽略这些警告，当然，除非你需要构建和使用那些可选任务。以下是一个错误消息示例，当 *bsf.jar* (*script* 任务所需) 未被包括在 CLASSPATH 中时，就会显示这一消息：

```
C:\ant1.4.1src\src\main\org\apache\tools\ant\taskdefs\optional\Script.java:56:
  Package com.ibm.bsf not found in import.
  import com.ibm.bsf.*;
```

如果你不打算使用 *script* 任务，就无需构建它，因此不必担心这一警告。不过，如果你希望构建和使用 *script* 任务，就需要将 *com.ibm.bsf* 包的 JAR 文件放入 CLASSPATH (或 Ant 的 *lib* 目录) 中，再重新构建。

如果希望直接安装到 ANT_HOME，也可用以下的命令行：

```
build install (Windows)
build.sh install (Unix)
```

这种方法仅在 ANT_HOME 得到设置的情况下才可行，而且必须小心使用。使用 *install* 选项时，ANT_HOME 总是用作目标，即便你指定了 *-Ddist.dir* 也不例外。如果 ANT_HOME 指向你现有的 Ant 安装，那么使用 *install* 选项不是一个特别安全的做法，因为你现有的安装会被新的构建所重写。如果重写了当前的构建，而新的构建不能工作，你就无法轻松地退回到原来的构建。

避免生成 JavaDoc

生成 JavaDoc 文档相当耗费时间。可以用 *dist-lite* 或 *install-lite* 选项来代替 *dist* 或 *install* 选项，从而避免生成 JavaDoc。除了不进行 JavaDoc 生成外，这些操作与其对应操作的作用是相同的。避免 JavaDoc 可以得到更快速的构建，如果你对 Ant 做了调整，而且不想等待 JavaDoc 生成，那么这是很有用的。

构建 Ant 1.2 版本

在 Ant 1.3 和 1.4.1 版本中，*build* 脚本会自动“引导”(bootstrap) Ant，这说明它会首先创建 Ant 的一个最小构建。然后再使用此最小 Ant 构建来完成余下的构建过程。传递给 *build* 脚本的参数将传递给 Ant 的构建文件。对于 Ant 1.2，则必须首先通过键入以下命令来手工地引导 Ant：

```
bootstrap (Windows)  
bootstrap.sh (Unix)
```

一旦完成此工作，就可以使用一个稍有不同的系统特性名继续处理 *build* 脚本：

```
build -Dant.dist.dir=destination_directory dist (Windows)  
build.sh -Dant.dist.dir=destination_directory dist (Unix)
```

与 Ant 1.3 和 1.4.1 不同，目标目录默认为 *./build* 而不是 *build*。还可以使用 *install* 选项将构建结果复制到 *ANT_HOME*，不过这同样存在类似的问题，即现有安装将被重写。一旦 Ant 已经构建，其行为与二进制发布是相同的，就好像你下载的是二进制发布一样。

Windows 安装问题

Ant 是一个相对非入侵的应用。它不会以任何方式更改 Windows 的注册表，而是依赖于前面所述的环境变量以及批处理文件。

设置环境变量

如果对于你的机器你有管理权限，就可以利用系统特性 (System Property) *applet* 来设置 *ANT_HOME*、*JAVA_HOME* 和 *PATH* 环境变量，双击控制面板 (Control Panel) 中的系统 (System) 图标可找到它。否则，你也可以创建一个简单的批处理文件，用它来设置这些变量。例 2-1 即显示了一个此类文件。

例 2-1：配置 Ant 环境的批处理文件

```
@echo off  
REM This batch file configures the environment to use Ant 1.4.1
```

```
set ANT_HOME=C:\ant\ant_1.4.1  
set JAVA_HOME=C:\java\jdk1.4  
set PATH=%ANT_HOME%\bin;%PATH%
```

尽管这样的批处理文件也能工作，但可以对此方法加以改进。其主要缺点在于，在执行 Ant 之前，你必须手工地执行此文件。虽然我们可以在批处理文件最后增加代码从而直接执行 Ant，但是更简单的做法可能是直接编辑 *ant.bat*，此文件包括在 Ant 发布中。作为意外收获，*ant.bat* 已经考虑到了 Windows 9x 和 Windows NT/2000/XP 之间的许多差异。

注意：很少需要编辑 *ant.bat*，除非你没有足够的访问权限以便用 Windows 的控制面板来设置环境变量。

如果确实要修改 *ant.bat*，则只需在此批处理文件的最前面硬编码这三个环境变量。可能你还希望将此批处理文件复制到一个已存在于系统路径中的新位置，那么你可以由任何命令提示键入 *ant*，这样就避免了在系统路径中包括 %ANT_HOME%/bin。

避免 CLASSPATH 问题

出于不同的原因，你可能希望对 *ant.bat* 进行编辑。默认情况下，此批处理文件将构建一个名为 LOCALCLASSPATH 的环境变量，其中包括 ANT_HOME/lib 目录中的所有 JAR 文件。然后将当前的系统 CLASSPATH 追加到 LOCALCLASSPATH，再把得到的路径传递给运行 Ant 的 Java 进程。调用 Java 的命令如下所示：

```
java -classpath %LOCALCLASSPATH% ...
```

Ant 使用当前的系统 CLASSPATH，这一点对于多人开发的工程来说尤其麻烦，因为每个人可能都有一个不同的系统 CLASSPATH。对一个开发人员来说可以成功工作的构建，对于另一个开发人员则可能失败，而仅仅是因为其 CLASSPATH 不同。如果这就是你的工程所存在的问题，则可以编辑 *ant.bat* 中的以下行（注 6）：

注 6：Ant 1.1 和 1.2 脚本在 %CLASSPATH% 周围还带有引号。

```
set LOCALCLASSPATH=%CLASSPATH%
```

只需将它修改为：

```
set LOCALCLASSPATH=
```

采用这种做法，当前用户的系统 CLASSPATH 对 Ant 就是不可见的。这一点很不错，因为 Ant 会通过在 `ANT_HOME/lib` 目录中查找来找到自己的 JAR 文件。如果你的工程需要在其 CLASSPATH 中有其他的 JAR 文件和目录，则应将它们列到构建文件中，而不要依赖于用户在运行 Ant 之前建立 CLASSPATH。请参见例 4-1 中的 `<path>` 部分，这是此技术的一个例子。

在 Ant 1.2 中，Windows NT/2000/XP 用户可能希望在 `ant.bat` 的开始和结尾处增加 `setlocal` 和 `endlocal` 命令。这样可以确保在此批处理文件中所设置的任何环境变量的作用域仅限于该批处理文件范围内。

定制 Ant 1.3 和 1.4.1 版本

从 Ant 1.3 版本开始，如果运行在 Windows NT、Windows 2000 或 XP 平台上，则 `ant.bat` 包括有 `setlocal` 和 `endlocal` 命令。它还增加了一个新的功能，允许你在 `ant.bat` 的开始处执行一个批处理文件，并在结尾处执行另一个批处理文件。如果希望在运行 Ant 前配置一些环境变量，而在 Ant 运行之后再将它们恢复为原来的值，那么这个功能是很有用的。

在运行 Ant 之前，`ant.bat` 会搜索一个名为 `%HOME%\antrc_pre.bat` 的文件。如果此文件存在，则在任何其他事情发生之前执行。这是在 Ant 运行前用于建立环境的钩子（hook）。在构建过程结束时，`ant.bat` 会搜索 `%HOME%\antrc_post.bat`，如果找到，则执行该文件。这是将一切恢复为原状的钩子。

Ant 没有附带这些批处理文件，而且不太可能设置 `HOME` 环境变量。如果你希望使用这些文件，就必须创建它们，然后配置 `HOME` 以使其指向包括这些文件的目录。不过，一旦完成这些工作，当 `ant.bat` 运行时，两个批处理文件就都会自动得到执行。

你可能希望设置环境变量ANT_OPTS。这个变量的值要作为一个JVM参数传递。其常见用途是指定系统特性。在这个简单的例子中，我们将`log.dir`系统特性传递给运行Ant的JVM：

```
$ set ANT_OPTS=-Dlog.dir=C:\logs  
$ ant run
```

现在此特性在构建文件中就可用了，例如：

```
<echo>Log directory is set to: ${log.dir}</echo>
```

如果此构建文件运行了一个Java应用，则可以从中访问此特性，如下所示：

```
String logDir = System.getProperty("log.dir");
```

ANT_OPTS的另一个常见用途是设置堆的最大容量。以下为使用Sun公司的JDK时，如何将堆的最大容量设置为128 MB：

```
set ANT_OPTS=-Xmx128m
```

最后，你可能会为JAVACMD环境变量指定一个值。它默认为%JAVA_HOME%\bin\java，并通常调用Sun公司的JDK所提供的JVM。Ant为那些希望指定另一个JVM的人提供了JAVACMD。

Unix 安装问题

Ant提供了一个名为`ant`的Bourne-shell脚本，其工作方式与Windows的批处理文件`ant.bat`很类似。正如其Windows版本一样，`ant`使用了同样的环境变量，包括：ANT_HOME、JAVA_HOME、CLASSPATH、ANT_OPTS和JAVACMD。每个环境变量在Unix下的工作方式与Windows下完全相同。

与Windows中有所不同，Ant在Unix中没有一组预执行和后执行的文件。在Unix下，Ant启动时只能选择执行一个命令文件。如果`ant`在当前用户的主目录中发现一个名为`.antrc`的文件，此文件将在`ant` shell脚本的任何其他部分之前执行。这

就是定制 Ant 所用的环境变量的钩子。以下代码节选自 *ant shell* 脚本，其中展示了用于调用 *.antrc* 的代码：

```
#!/bin/sh
if [ -f $HOME/.antrc ] ; then
    . $HOME/.antrc
fi
```

由于 Unix 是一个多用户操作系统，所以 Ant 通常安装在一个共享目录中，以便在系统范围使用。例如，Ant 发布可能安装在 */opt*，而 *ant Bourne-shell* 脚本可能安装在 */opt/bin*（有时用 */usr/local* 和 */usr/local/bin* 代替）。这种安装几乎必然要求你是系统管理员，不过，这样确实易于为一组开发人员配置 Ant。

与 Windows 批处理文件一样，Unix shell 脚本也向 Ant 进程传递当前的系统 CLASSPATH。你可能希望删除对 CLASSPATH 的引用，从而确保每个开发人员都用同样的配置。在此重申，应当由你的构建文件定义 CLASSPATH，而不要依赖于用户在运行 Ant 之前建立其自己的 CLASSPATH。

注意：如果你想使用 Perl 或 Python，那么 Ant 还包括有 *runant.pl* 和 *runant.py*，它们完成与 Unix 和 Windows 脚本类似的任务。

配置

一旦 Ant 得到正确安装，就只需很少的其他配置。假设 *ant.bat* 批处理文件（或 *ant shell* 脚本）已经安装在一个目录上，而该目录包括在系统路径中，则从任何命令提示符都能够运行 Ant。

你不必过于刻板地非要用所提供的 *ant* 或 *ant.bat* 脚本来运行 Ant。只要以下各项得到配置，就可以手工地运行 Ant（注 7）：

- 系统 CLASSPATH 包括 *ant.jar* 和任何与 JAXP 兼容的 XML 解析程序。

注 7：设置这些项正是 *ant.bat* 和 *ant* 的工作。

- 对于 JDK 1.1，必须将 *classes.zip* 加到 CLASSPATH 中。对于 Java 2，则必须加入 *tools.jar*。这对于像 *javac* 这样的任务是必要的。
- 许多任务要求将 Java 系统特性 *ant.home* 设置为 Ant 安装目录。这是通过在启动 JVM 时加上 -D 标志完成的，稍后即会说明。
- 对应可选任务的 JAR 文件必须加到 CLASSPATH 中。

假设这些项均得到正确的设置，就可以使用以下命令来调用 Ant（注 8）：

```
java -Dant.home=pathToAnt org.apache.tools.ant.Main
```

注意：如果你要定制 Ant，并且希望使用你自己的 IDE 调试器来运行它，那么理解如何手工地建立和运行 Ant 是很有用的。

XML 问题

每个 Ant 版本（1.4 版本之前）都包括有 Sun 公司的 JAXP（Java API for XML Parsing，面向 XMU 解析的 Java API）1.0 版本。Ant 1.4 及以后版本中则附带了 JAXP 1.1，正如下一段所述。JAXP 是一个 API，它允许 Java 程序以一种可移植的方式使用来自不同厂商的 XML 解析程序。JAXP JAR 文件 *jaxp.jar* 和 *parser.jar* 可在 *ANT_HOME/lib* 中找到。这些文件（以及此目录中的任何其他文件）都由 *ant* 或 *ant.bat* 自动添加到 Ant 所用的 CLASSPATH 中。*jaxp.jar* 包括了 JAXP API，而 *parser.jar* 则是由 Sun 公司提供的一个 XML 解析程序实现。

如果需要 DOM Level（注 9）2 或 SAX 2.0（注 10）的 XML 支持，那么 JAXP 1.0 就不够了。对于这一级别的 XML，应当对 Ant 升级以支持 JAXP 1.1（注 11），可

注 8：或者也可以使用 *-classpath* 命令行选项来指定 CLASSPATH。

注 9：文档对象模型（Document Object Model），参见 <http://www.w3.org/DOM/>。

注 10：面向 XML 的简单 API（Simple API for XML），参见 <http://www.saxproject.org/>。

注 11：对于 JAXP 1.1，其缩写现在代表“Java API for XML Processing”（面向 XML 处理的 Java API）。

以从 <http://java.sun.com/xml/> 得到。如果你所用的是 Ant 1.4 或以后版本，那么这不会对你产生影响，因为它本身就附带了 JAXP 1.1。由于 JAXP 1.1 这是向前兼容 JAXP 1.0 的，所以可以在你的 ANT_HOME/lib 目录中安全地将 *jaxp.jar* 和 *parser.jar* 文件替换为 JAXP 1.1 发布中的 *jaxp.jar* 和 *crimson.jar*。如果你更偏向于另一个与 JAXP 兼容的 XML 解析程序，如 Apache 的 Xerces (<http://xml.apache.org>)，则可以用其 JAR 文件来代替 *crimson.jar*。

对 XSLT (XSL Transformation, XSL 转换) 的支持是 JAXP 1.1 的另一个功能。Ant 通过 *style* 任务支持 XSLT，它可以利用任何与 JAXP 1.1 兼容的 XSLT 处理程序。与 XML 解析程序一样，也要将 XSLT 处理程序的 JAR 文件复制到 ANT_HOME/lib，或者增加到 CLASSPATH 中。JAXP 1.1 发布包括有 Apache 的 *xalan.jar*，你也可以选择另外的处理程序。

最后要说明的是，Sun 公司于 2002 年 2 月发布了 J2SE 1.4。Java 的这一版本包括了 JAXP 1.1、一个 XML 解析程序以及一个 XSLT 处理程序。由于目前它们都是 Java 的核心部分，所以可以由“引导”类加载器来加载。这说明，如果在 J2SE 1.4 下运行，那么 ANT_HOME/lib 中相应的 XML 库甚至不会被用到。要安装 J2SE 1.4 下更新的 XML 支持，请参见“正式标准 (Endorsed Standards)”文档，位于 <http://java.sun.com/j2se/1.4/docs/guide/standards/>。

注意： XML 解析程序和 XSLT 处理程序正在被迅速改进。可以安装 Sun 公司的 JAXP 1.1 参考实现中的 *jaxp.jar*、*crimson.jar* 和 *xalan.jar*，尽管这是实现 XML 和 XSLT 功能最简单的方法，但你仍应当对更新的 XML 解析程序和 XSLT 处理程序加以关注，从而获得优化的性能。

代理配置

像 *get* 这样的 Ant 网络任务要求有网络连接来进行操作。对拨号用户来说，这说明在这一类任务能够运行前，调制解调器必须拨号并连接。不过，许多公司用户会在公司的防火墙内运行 Ant。如果是这种情况，那么在到达远程计算机之前，往往必须配置 JVM 来使用一个代理服务器。

通过设置 JVM 系统特性 proxySet、proxyHost 和 proxyPort，这一点很容易做到。可以通过修改 Ant 的启动脚本来设置这些特性，也可以使用 ANT_OPTS 环境变量来设置。以下例子展示的 Windows 命令即是使用 ANT_OPTS 来指定这些特性，并在其后调用 Ant：

```
set ANT_OPTS=-DproxySet=true -DproxyHost=localhost -DproxyPort=80  
ant mytarget
```

在 Unix 上也可以采用同样的方法，不过语法稍有不同，这要取决于你所用的 shell：

```
$ export ANT_OPTS="-DproxySet=true -DproxyHost=localhost -DproxyPort=80"  
$ ant mytarget
```

可以从命令行、*antrc_pre.bat* (Windows) 或 *.antrc* (Unix) 发出命令来设置 ANT_OPTS。

第三章

构建文件

用 Ant 构建工程要注意两点：工程组织和 Ant 构建文件。它们绝不是彼此独立的两个焦点，而是事物的正反两面；这两个主题是紧密相关的，对其中任何一个方面的设计决定都会影响到另一个方面的设计和实现。在这一章中，我们将介绍如何用 Ant 设计一个工程来管理构建过程，以及如何为该工程编写一个构建文件。在逐步介绍示例工程时，我们将解释如何做出特定的工程布局 (layout) 决策，还会说明如何确定构建文件的各个部分。并非每个工程都适合我们所提供的模型，不过许多工程确实可以；我们希望通过这个练习可以使你准备好编写其他开发工程中的构建文件。

不过，在开始之前，你应当首先理解构建文件自身的功能。构建文件是 Ant 最重要的一个方面，而且有许多细节需要加以解释。构建文件使用了 XML，对此的理解非常重要。有了这个基础，你就可以更好地研究构建文件的主要部分。为了达到这一目的，在本章中，我们将先来看一看 Ant 使用 XML 的原因所在，并由此展开讨论。然后我们将介绍示例工程及其相应的布局，并且确定构建需求。这些元素加在一起即可创建出我们的示例构建文件。

有了写好的构建文件，我们就可以分析 Ant 如何读取构建文件，以及如何执行构建文件中所定义的步骤了。你可以看到 Ant 的灵活性是如何导致一个复杂的过程的。我们会解释这一过程，这样你就不仅能够在编写构建文件时应用此知识，而

且在扩展 Ant 时（例如，用一个用户编写任务来扩展）也可以使用这些知识。最后，我们会介绍一些 Ant 的特殊问题，将谈到其遗漏的内容，还会讨论如何避免其导致的某些问题。那么，现在就开始吧！

为什么用 XML？

其他构建工具使用的模型是用依赖关系（dependency）来定义的，而 Java 工程与此不同，它是用包（package）和组件（component）来描述的，这遵循了 Java 语言的包和对象模型。在最初开发 Ant 的时候，没有任何一种构建工具采用这种做法，由于需要为 Java 提供比已有构建工具更好的工具，这种想法才应运而生。因为 Sun 公司为其工具提供了一个 Java 库，这样就使用户可以通过程序来访问 Java 编译器、jar 工具等等，所以对于一个新的 Java 工程构建引擎来说，可选择的最好语言就是 Java。有了计划和语言，Ant 设计者 James Duncan Davidson 所要做的惟一一件事就是选择构建文件描述文件语法。

James 对构建文件语法有许多需求。首先，此语法对新用户来说必须很简单，从而使他们可以轻松上手。其次，必须有可用的（可免费读取的）Java 库，这样这种新的基于 Java 的构建引擎才可以很容易地实现和维护。最重要的是，编写一个新引擎应该不要求编写一个新的语法解析程序——现有库是至关重要的。另一个设计目标是要能够表述一种层次树的构建结构。另外，此语法应能够用组件和包的方式描述构建，还同时能用操作的方式来描述。下面我们就来看看为什么 XML 能够满足这些需求。

开发人员能理解 XML，这是因为在许多 Java 开发的领域中都用到了 XML，如 EJB（Enterprise Java Beans，企业 Java Beans）、JSP（Java Server Pages，Java 服务页面）以及像 SOAP（Simple Object Access Protocol，简单对象访问协议）这样的数据传输等等。在 Java 世界之外，XML 同样得到了普遍认同，这样就为 Ant 提供了一个覆盖面很广的潜在用户群。XML 的解析程序和模型库可以作为 Java 库免费获得。文档也不是问题；有数百种图书、杂志以及网站致力于 XML 技术的研究。作为一种通用描述语言，XML 可以满足前面提出的复杂的实

用需求。它可以描述操作、数据类型、数据值以及工程布局。XML的这些属性与Ant的设计需求极为匹配，因此，XML是Ant的最佳选择。

Ant 构建块

有了XML元素和标签，我们可以来了解Ant构建文件的主要组件，它们即作为其组件或构建块。我们要使用这些块来构建构建文件。有些部分有专门用途，而另一些是通用的，而且经常被使用。下面来看Ant构建文件的主要组件。

工程

对于一个XML文件，从根元素（在此为`<project>`）到最底层的嵌套标签这样一组标签和元素，我们称之为DOM（document object model，文档对象模型）。任何构建文件的第一个元素（即根元素）总是`<project>`标签。所有构建文件都不能没有此标签，而且也不能有多个`<project>`标签。DOM以树型层次放置元素，这使得构建文件更像是一个对象模型，而不仅仅是一个简单的过程描述文档。以下示例显示了一个合法的工程标签：

```
<project name="MyProject" default="all" basedir=".">
  ...
</project>
```

`<project>`标签有3个属性：`name`、`default`和`basedir`。`name`属性为工程提供一个名字。工程名对于识别日志输出（以了解正在构建的工程）非常有用。对于管理构建文件的系统（如可以读取构建文件的一个IDE），工程名就相当于此构建文件的一个标识符。`default`属性是指构建文件中的一个目标名。如果运行Ant时在命令行未指定一个目标，Ant就会执行默认目标。如果默认目标不存在，Ant将返回一个错误。尽管`default`的值不必为一个合法的目标名（即对应为构建文件中的一个具体目标名），不过我们不鼓励这种做法。建议令默认目标编译所有内容，或者显示使用构建文件的帮助。`basedir`属性定义了工程的根目录，一般情况下为“`.`”，这也是此构建文件所在目录，而不论你在运行Ant时在哪个目录之下。不过，`basedir`还可以定义不同的参考点。例如，作为一个层次工程结构中

一部分的构建文件就需要一个不同的参考点，它指向此工程的根目录。你可以使用 `basedir` 来指定此参考点。

目标

目标 (target) 直接映射为构建的需求规范中所提出的广义目标 (broad goal)。例如，为包 `org.jarkarta` 编译最新源代码，并将其放入一个 JAR 中，这就是一个广义目标，因此也将是构建文件中的一个目标。目标包括做具体工作的任务，这些任务将完成最终的目标。

以下目标将编译一组文件，并将它们打包到一个名为 `finallib.jar` 的文件中。

```
<target name="build-lib">
    <javac srcdir="${src.ejb.dir}:${src.java.dir}"
           destdir="${build.dir}"
           debug="on"
           deprecation="on"
           includes="**/*.java"
           excludes="${global.exclude}">
        <classpath>
            <pathelement location=". "/>
            <pathelement location="${lib.dir}/somelib.jar"/>
        </classpath>
    </javac>
    <jar jarfile="${dist}/lib/finallib.jar" basedir="${build.dir}"/>
</target>
```

如果需要，目标可以有更细的粒度，如下例所示，其中包括一个编译源代码的目标，还有另一个用来打包 JAR 文件的目标：

```
<target name="build-lib">
    <javac srcdir="${src.ejb.dir}:${src.java.dir}"
           destdir="${build.dir}"
           debug="on"
           deprecation="on"
           includes="**/*.java"
           excludes="${global.exclude}">
        <classpath path="${classpath.compile}" />
    </javac>
</target>
```

```
<target name="package-lib">
    <jar jarfile="\${dist}/lib/lib.jar" basedir="\${build.dir}"/>
</target>
```

这种细粒度可能是必要的，例如，如果一个任务（如编译源代码的任务）失败，则不会使另一个相关任务（如构建JAR的任务）的执行停止。在此例中，无论编译目标是否成功，库JAR都会被构建。

一般来说，目标为粗粒度的操作更为合适。任务可以比目标更好地解决细粒度问题。编写构建文件时，尽管并不一定都要遵循在此所示的模型，不过如果至少在目标中维护一致的粒度，那么你就会得到更好的结果。任意编写构建文件意味着你以后需要做更多的工作，因为当工程中增加新功能或新的构建目标时，工程小组中的每个成员都会来找你来提供指导（因为你是构建文件的原作者）。你的目标应当是创建一种只需少量调整（或不需调整）的构建文件，而这就要从目标设计开始。

任务

任务是构建文件中最小的构建块，它们解决构建中更小粒度的目标。这些任务完成具体的工作，包括编译源代码、对类打包、由CVS获取修订情况或者是复制文件和（或）目录。Ant并不像其他一些构建工具那样提供对底层shell的直接通道，而是将所有操作包装为任务定义，每个任务对应为Ant对象模型中的一个Java对象。在Ant中不存在没有相对对象的任务。就此比较而言，shell不仅能运行可执行程序（这与Ant的任务对象模式类似），还可运行不对应为可执行程序的命令（例如，Win32 shell的dir命令）。“每个任务都是对象”这种体系结构为Ant提供了灵活的扩展性，对此我们将在第五章和第六章加以讨论。

以下任务示例使用了copy任务，从而将工程www源目录的jsp中的所有文件（及子目录）复制到系统的WebLogic安装的jsp目录中。路径分隔符“/”在Windows和Unix中都能工作，这也是Ant的一个好处：

```
<copy todir="\${weblogic.dir}/\${weblogic.server.home}/public_html/jsp">
    <fileset dir="\${src.www.dir}/jsp"/>
</copy>
```

任务可以完成其实现者所设计的任何工作。你甚至可以编写一个任务来删除构建文件以及工程中的所有目录。这样做并无好处（至少我们认为如此），但 Ant 不会禁止这样做。

尽管任务（作为 Java 对象以及作为 XML 标签）遵循对象层次结构，这些层次结构却并不相关。例如，对于 copy 任务，就不能因为它有嵌套的 <fileset> 元素而认为 Fileset 对象是 Copy 对象的子类。反之亦然，尽管 Jar 对象由 zip 对象扩展而来，但这并不表示 <zip> 标签可以嵌套一个 <jar> 标签。

数据元素

数据元素可能是 Ant 中最容易混淆的方面。部分变量和部分抽象数据类型等元素表示的是数据而不是要完成的任务。数据元素可以分为两类：特性和 DataType。为了避免混淆，先来明确本章（以及书中余下部分）中所用的一些术语：

特性 (*property*)

在构建文件中由 <property> 标签表示的名 - 值对。

DataType

一类表示复杂数据集合的元素，例如 fileset 和 path。

数据元素 (*data element*)

这个词涵盖了特性和 DataType。

在第四章中，我们将更为详细地介绍 Ant 的 DataType 如何工作，以及如何在构建文件中使用 DataType。

特性

这两种数据元素中，特性 (*property*) 较为简单。它们只表示字符串数据的名 - 值对。除了字符串以外，任何其他数据类型均不能与特性相关联。Java 程序员会意外地发现，特性可间接地与 Java SDK 中的 `Property` 对象相关。这说明通过使用诸如特性文件 (*property file*) 或 JVM 命令行特性设置，你可以在构建时动态地定义特性。

以下是在一个构建文件中使用 `<property>` 标签设置的一些特性示例。前两个元素均将一个特性设置为一个给定值，而第 3 个 `<property>` 元素则加载了一个特性文件。此代码将在 `<project>` 元素的 `basedir` 特性所指定的目录中查找该特性文件。

```
<property name="my.first.property" value="ignore me"/>
<property name="my.second.property" value="a longer, space-filled string"/>
<property file="user.properties"/>
```

要引用特性（或者更确切地说，即引用其值），可以通过使用 `${<property-name>}` 语法实现，如下例所示：

```
<property name="property.one" value="one"/>
<property name="property.two" value="${property.one}:two"/>
```

在本章后面的“工程级数据元素和任务”中，我们将描述 Ant 如何使用特性，以及特性如何满足处理模式。

与 `DataType` 相比，特性的一个好处是其值是类型不可知的 (type-agnostic)，也就是说，它们总是字符串。这是什么意思呢？举个例子，某个特性表示一个目录名。此特性并不知道其值是一个目录，而且它也不关心此目录是否确实存在。如果你需要表示临时构建目录的名字，而这些目录仅在构建过程中存在，那么这一点就很不错。不过，特性并不总是表示路径的最佳数据元素；有时你可能希望对定义路径有更多的控制。对此就可以使用 `DataType`。

DataType

将路径和文件列表定义为特性相当麻烦，而且很容易出错。例如，假设你的工程有一个库目录，其中包括 25 个 JAR。用一个路径字符串来表示这些文件，你就会得到一个非常长的特性定义，如下所示：

```
<property name="classpath" value="${lib.dir}/j2ee.jar:${lib.dir}/activation.jar:
${lib.dir}/servlet.jar:${lib.dir}/jasper.jar:${lib.dir}/crimson.jar:${lib.dir}/jaxp.
jar"/>
```

在这个库中增加和删除 JAR 都意味着你必须对此路径字符串做相应的增加和删

除。还有一种更好的方法。可以使用 fileset DataType 而不是特性中的长路径字符串。例如：

```
<path id="classpath">
    <fileset dir="${lib.dir}">
        <include name="j2ee.jar"/>
        <include name="activation.jar"/>
        <include name="servlet.jar"/>
        ...
    </fileset>
</path>
```

更好的做法是，由于所有 JAR 都在同一个目录下，所以你可以使用通配符，并指定一个 <include> 模式（特性不能使用模式）。例如：

```
<path id="classpath">
    <fileset dir="${lib.dir}">
        <include name="**/*.jar"/>
    </fileset>
</path>
```

这就更容易了！除了明显可以减少键入以外，较之于使用 property 标签，fileset DataType 的使用还有另一个优点。无论在工程的目录中有 2 个还是 25 个 JAR，fileset DataType（如以上两个示例所示）都会通过设置 classpath 对它们全部加以表示。另一方面，每次增加或修改一个 JAR 时，你仍然需要修改路径 - 特性值，即增加或修改 JAR 文件名。

有些 DataType（但并不是全部）可以在构建文件 DOM 的“工程级”定义，这说明它们嵌套在 <project> 元素中。此功能是 Ant 所固有的，不能进行修改，除非你想维护你自己的 Ant 版本。有关 DataType 的更多信息请参见第四章，对于如何使用 DataType 完成特定任务请参见第七章和第八章。

一个示例工程及构建文件

为了向本书提供示例构建文件，我们需要一个示例工程。在此使用了名为 irssibot 的已有工程（一个基于 GNU Make 的工程），它是由 Matti Dahlbom 所编写的一

个 IRC bot（注 1）（源代码可参见 <http://dreamland.tky.hut.fi/Irssibot>）。此工程需要一个典型构建的所有功能：编译源代码、对类打包、清除目录以及部署应用。作为练习，我们取此工程，并将它进行转换以使用 Ant。

理解工程结构

先来看如何为此 irssibot 工程配置目录。Java 的工程组织方法根据工程的不同而不同，有时差别相当大（例如，Web 应用的工程结构与 GUI 工具的工程结构就大相径庭）。很多情况下，工具规定了工程的结构。有些 IDE（例如 VisualAge 3.5 以前的版本）就要求所有源代码都在一个文件中。EJB 和 CORBA 编译器要求源文件和目录遵循命名约定。无论哪一种情况，工程模型都应该满足 RCS（Revision Control System，版本控制系统）的需求（你应该已经使用了版本控制系统吧？）。由于存在这些不同的需求和依赖关系，所以并没有最佳的工程组织模式，而且我们在此也不会提出此类的建议。不过我们所描述的布局和组织很简单，足以用于许多工程当中，而且特别适合于使用 Ant。

设计和实现一个工程结构绝非易事，因此不要认为仅花 1 个小时就能很好地完成了工作。这个工作并不是很困难，但相当繁琐。大多数 Java 程序都有跨平台功能，你在考虑如何组织工程时应记住这个目标。以往只是会考虑到不同操作系统之间和/或不同硬件配置之间的情况。不过，在开发小组中，不同的平台还意味着异构工作站之间小到工具集的差别。功能要做到明显分离，要能够做到自包含，而且没有外来的需求，这些都应作为 Java 工程的目标。为你的工程实现这样的结构，其好处可能不会立即体现出来，但是随着越来越多的开发人员使用你的构建系统，而且随着功能逐渐加入到你的工程中来，你会为提前做了考虑而感到万分庆幸。与修改一个有 45 个目录和 1000 个类的、已建成的工程相比，修改构建文件要简单得多。

注 1：IRC（Internet Relay Chat，Internet 中继聊天）由一系列服务器组成，这些服务器允许用户使用 IRC 客户实时地进行通信。人们在通道（channel）中通信，或“聊天”。这些通道通常有“bot”，或自动的 IRC 客户，它可以管理此通道并保证它打开。否则，如果通道里没有人，它就会关闭。irssibot 就是这样一种 bot。

图 3-1 中的目录展示了我们为满足如上所述的目标而为此示例工程所设计的目录和文件结构。

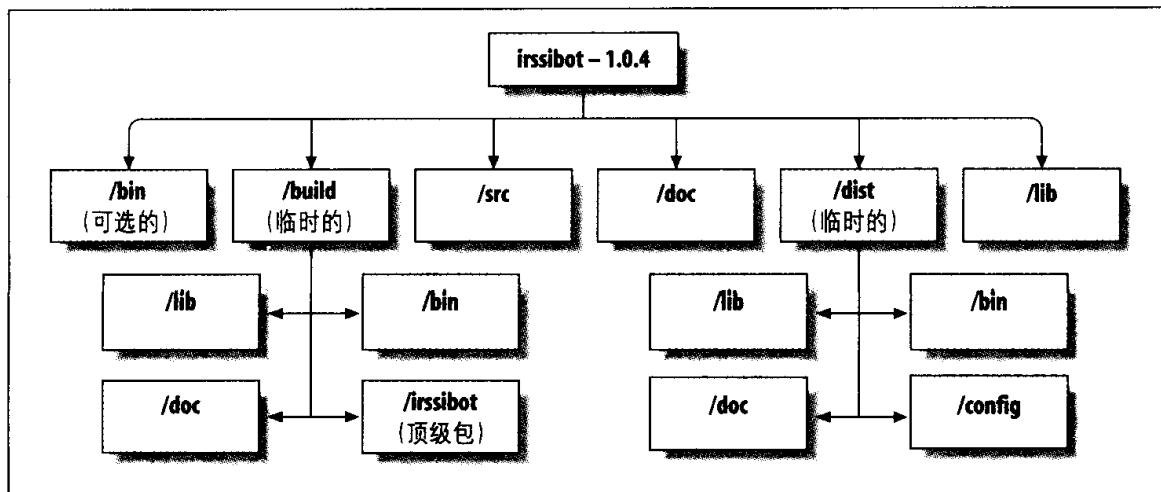


图 3-1: irssibot 目录结构

先从最上面的 *build.xml* 开始介绍，这就是构建文件（注 2）。将此构建文件放在工程的根目录上，这样就使我们能够在数据元素和特性中对工程目录的定义使用相对路径。要避免使用绝对路径，因为这样会破坏工程的可发布性。Java 源代码包以 */src* 目录为根目录。这种组织使我们可以将源代码与得到的类文件相分离。

类文件置于 */build* 中。有时（但在这个工程中却并非如此），有必要将类再进行分组，例如，可分为库和应用。应当在 */src* 和 */build* 目录下做此分离，而不是在根目录下分离。一方面，这样可以减少工程根目录下的混乱情况。另一方面，从更为技术性的角度来说，适当的分离可以在很大程度上使文件管理更容易。例如，如果删除了 */build* 目录，就会删除所有已编译的类。无论如何分解工程，这个方法都保持有效。你总是可以增加目标和任务来处理更为特定的问题，但是却不能总对工程的布局进行调整。

对于没有作为工程一部分而构建的库中的类，其 JAR 和目录放在 */lib* 目录下。再发布库可能比较麻烦，但是不要忽视这个问题。你可能会假定，通过某些

注 2：要记住，*build.xml* 为默认的构建文件名。如果在命令行调用 Ant 时未指定构建文件，Ant 就会假定构建文件名为 *build.xml*。

README文件就能够解释哪些库是必要的,以及从哪里可以得到这些库,从而将所有一切都留给开发人员完成。请务必避免这种做法!(注 3) 开发人员可能由于还开发有其他工程而拥有某个已知库的各个版本,这些版本会存放在其系统的多个位置上。你无法推断出他们到底有什么。将你的工程所用的库再进行发布将对这些开发人员有所帮助。这样当他们在机器上运行你的应用时问题就会少一些,因为你已经为其提供了合适的库。再发布库会增加你的应用包的大小,但是这样做利大于弊。

我们将应用的脚本(无论是安装还是执行脚本)放在`/bin`目录下。此示例工程分别为Windows和Unix提供了运行IRC bot的脚本,即`bot.bat`(Windows)和Bourne Shell 脚本`bot.sh`(Unix)。有时,工程中存在一些构建时所需的可执行程序,而这些程序可能很难找到或者是定制的,这些也都放在`/bin`中。尽管依赖于可执行程序来完成当前Ant任务所不支持的一些功能对你来说最为容易,但还是应该考虑编写一个定制的任务而非可执行程序,因为可执行程序往往会削弱Ant的跨平台功能。

对于文档,我们将非JavaDoc文档放在`/doc`目录中。这可能包括工程的README文档、终端用户文档以及对应于所包括库的文档。基本上就是构建所不能生成的所有文档。

`/dist`目录是我们发布最终产品的位置。非归档的类包、JAR、WAR、EAR和TAR,还有其他一些文件均放在此处。对于JAR及新构建的应用所需的其他包文件,我们在`/dist`目录下设有一个`lib`目录(`/dist/lib`)。如果需要,为分布式文档和生成的javadoc还建有一个`dist/doc`目录。`dist/bin`目录用于存放令应用更易于运行的脚本和可执行程序。发布目录可以方便安装,因为在大多数情况下,安装只是将文件从`/dist`目录复制到文件系统中的其他指定位置上。

设计和编写示例构建文件

既然有了目录结构,下面来为此示例工程设计和编写构建文件。为了更好地描述

注 3: 这不是一个严格的规则,但适用于大多数情况。即便是像Tomcat和JBoss这样的大工程也会附带一些在别处可用的库。

工程目标和构建文件中各部分之间的关系，我们将在定义和描述特定目标之后展示相应的构建文件语法。在开始编写构建文件之前，最好是首先描述和设计你的构建。

要为工程设计和实现一个构建文件，一种方法就是提出一组问题。对这些问题的答案即构成了构建文件的各个部分，把它们加在一起也就形成了完整的解决方案。以下列出了这些问题，在此没有特定的顺序：

- 构建文件如何开始？
- 应该定义哪些特性和 DataType 以用于整个构建中？
- 在对目标进行编译或打包之前，需要创建哪些目录？
- 整个程序由哪些内容组成？库的情况如何？用于安装或执行的脚本是怎样的？静态的和生成的文档是怎样的？
- 修改文件后如何重新构建工程？需要删除所有的类文件吗？需要删除所生成的 JAR 吗？
- 在应用准备发布之前，需要创建哪些目录？需要将源代码与应用同时发布吗？应用的发布由哪些内容组成？

在日常工作中，你可能还会在自己的自由讨论会议中提出更多的问题。你应该会预料到这一点，因为许多问题都与工程的特定条件直接相关。例如，对于这个工程，你从未问过有关构建 EJB 的问题，但是对于你自己的工程则有可能会问到。在开始的工程设计阶段削减功能并缩小规模，这要比在后面增加某个特定功能或步骤容易得多。因此不要害怕提出更多的问题。你完全可以在后面将问题加以合并或剔除。下面我们将回答前面所提出的问题，并着手编写一个组织合理的构建文件。

工程描述文件

我们的第一个问题是：

- 构建文件如何开始？

所有构建文件都由一个工程描述文件（project descriptor）开始，它指定了诸如工程名、默认的构建文件目标以及工程基目录等内容。我们将此工程称为irssibot，默认的目标为all；后面将编写此目标，使之完成应用的编译并将其打包到一个JAR文件中。对于这个工程，其当前的工作目录即为构建文件的基目录，表示为“.”。<?xml?>标签是XML文件的标准，并由XML解析程序库，而不是Ant自身来使用。下面是构建文件中的第一行，而且不应包括任何尾缀空格。

```
<?xml version="1.0"?>

<!-- Comments are just as important in buildfiles, do not -->
<!-- avoid writing them! -->
<!-- Example build file for "Ant: The Definitive Guide" -->
<!-- and its sample project: irssibot -->

<project name="irssibot" default="all" basedir=".">
    /

```

全局变量

我们要考虑的下一个问题是：

- 应该定义哪些特性和DataType以用于整个构建中？

所有工程目录都由根工程目录展开。这样，所有与目录相关的数据元素都应当是相对的。我们为每个主要的子目录定义一个特性，并根据它的作用为之命名。这样做可以保证如果目录名有所调整，那么只需在构建文件中修改一处即可。

```
<!-- Project-wide settings. All directories are relative to the -->
<!-- project root directory -->

<!-- Project directories -->
<property name="src.dir" value="src"/>
<property name="doc.dir" value="doc"/>
<property name="dist.dir" value="dist"/>
<property name="lib.dir" value="lib"/>
<property name="bin.dir" value="bin"/>

<!-- Temporary build directory names -->
<property name="build.dir" value="build"/>
<property name="build.classes" value="${build.dir}/classes"/>
<property name="build.doc" value="${build.dir}/doc"/>
```

```
<property name="build.lib" value="${build.dir}/lib"/>
```

除了全局地定义目录名外，对于某些任务，对特性也适于采用全局定义。在此，我们定义了一个全局特性，它将通知 javac 任务是否生成带调试信息的字节码（译注 1）。javac 任务的所有实例都会使用此特性。

```
<!-- Global settings -->
<property name="javac.debug" value="on"/>
```

我们所设置的下一个特性为 build.compiler。在此其值为 modern，它表示 javac 使用 Java SDK 工具包（即 Java SDK 1.3 及更高版本）中可用的、最新版本的 Sun 编译器。这是一个“魔法特性”（magic property），本章后面将讨论此类特性的一些负面作用。即使你要在所编写的每个构建文件中使用这个值，也最好还是将其用途记入文档。对于许多刚开始接触 Ant 的人，如果发现此处有该特性，而该特性在构建文件中从来没有再用到，就会感到困惑。

```
<!-- Global "magic" property for <javac> -->
<property name="build.compiler" value="modern"/>
```

在深入到定义（和满足）工程的主要目标之前，我们还有最后一步需要完成。irssibot 工程附带了一组库，即 mysql.jar 和 xerces.jar。我们定义了一个全局可用的类路径（classpath），它包括了这些库，而且还可以加入我们（或其他开发人员）将来可能增加的库。对于与路径兼容的任务（如 javac）、文件集（file set）和包含（include）模式（'**/*.jar'）表示：库目录（/lib）及其子目录中的所有文件都应构成其适用的路径（注 4）。

```
<path id="classpath">
  <fileset dir="${lib.dir}">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

译注 1：原文此处为“调试指针”，但在 Java 编译中并无此概念，分析其含义应为“调试信息”。

注 4：与路径兼容性的任务能够对一组目标或文件进行操作，而不仅仅是操作一个目录或文件。这些任务通常对应为展示相同行为的工具，如 javac 或 rm。

目录创建

现在我们需要回答以下问题：

- 在对目标进行编译或打包之前，需要创建哪些目录？

对于我们这个工程，与编译相关的目录（Ant 将所有已编译的类保存在这个目录中）即为构建目录 build 及其子目录（如果有的话）。我们将定义一个准备目标来创建此构建目录。

另外，对此准备步骤我们将增加一些内容，并记录构建的时间，这对于自动的构建极为有用。

```
<!-- Target to create the build directories prior to a compile target -->
<!-- We also mark the start time of the build, for the log. -->
<target name="prepare">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${build.lib}" />
    <mkdir dir="${build.classes}" />
    <mkdir dir="${build.classes}/modules"/>

    <tstamp/>

    <echo message="\${TSTAMP}" />
</target>
```

编译

要编译这个工程，需要回答几个问题：

- 整个程序由哪些内容组成？
- 库的情况如何？
- 用于安装或执行的脚本是怎样的？
- 静态的和生成的文档是怎样的？

对这些问题，我们分别用一个目标逐一加以解决。“整个程序”这个词可以表示许

多含义。对于大多数工程而言（也包括我们的这个工程），其答案是很简单的。整个应用由所有已编译的类、执行应用的脚本以及程序的配置文件组成。

首先，我们要编译此应用，并将它完全打包到一个 JAR 中。在某些情况下，你可能希望将编译和打包至 JAR 的过程分为两步。为了简单起见，在此例中我们将在一个目标中完成这些工作。

```
<!-- Build the IRC bot application -->
<target name="bot" depends="prepare">
    <!-- Compile the application classes, not the module classes -->
    <javac destdir="${build.classes}"
        debug="${debug.flag}"
        deprecation="on">
        <!-- We could have used javac's srcdir attribute -->
        <src path="${src.dir}"/>
        <exclude name="irssibot/modules/**"/>
        <classpath refid="classpath"/>
    </javac>
    <!-- Package the application into a JAR -->
    <jar jarfile="${build.lib}/irssibot.jar"
        basedir="${build.classes}" >
        <exclude name="irssibot/modules/**"/>
    </jar>
</target>
```

irssibot 还包括一组模块，它们扩展了 bot 的功能。将模块类文件和应用类文件相分离，这样可以使得更新应用更容易一些。将来，开发人员很有可能修改和增加模块，而不是主应用的某些部分。通过对包的分离，可使开发人员能够只更新需要更新的类文件。在此将模块编译并打包为一个单独的 JAR。

```
<!-- Build the IRC bot modules -->
<target name="modules" depends="prepare,bot">
    <!-- Compile just the module classes -->
    <javac destdir="${build.classes}/modules"
        debug="${debug.flag}"
        deprecation="on">
        <!-- We could have used javac's srcdir attribute -->
        <src path="${src.dir}"/>
        <include name="irssibot/modules/**"/>
        <classpath refid="classpath"/>
    </javac>

    <!-- Bundle the modules as a JAR -->
```

```
<jar jarfile="\${build.lib}/irssimodules.jar"  
      basedir="\${build.classes}/modules" >  
  <include name="irssibot/modules/**"/>  
</jar>  
</target>
```

irssibot脚本在构建中无需处理，因此我们没有提供目标来处理它们。对配置文件也同样如此。对于脚本和配置文件，即使我们没有编写目标来对其修改或打包，从长远来看，对它们有所考虑仍然是很重要的。在你自己的构建中，这种考虑可能会改变其他目标的实现。

在开始这些工作时，我们提到过此构建文件的默认目标叫做 `all`。这个目标只是利用了 Ant 的依赖关系机制，要求 `bot` 和模块目标都得到运行，从而构建应用。如果你调用 `ant` 时未带有任何参数，则 Ant 会执行 `all` 目标。对于 `all`，我们只需编写如下代码：

```
<target name="all" depends="bot,modules"/>
```

在你自己的构建文件中，并不总是需要有类似于 `all` 这样的目标。还有一个选择是提供一个不做任何事情的默认目标。我们的建议是编写一个帮助目标（即使它不是默认的，也起码要有一个这样的目标）。如果用户调用 `ant` 时不带任何参数，则会得到你的构建文件的帮助文档。例如，可能显示如下内容：

```
Build the foo application with Ant. Targets include:  
  full - build the entire application and its libraries  
  app - build just the application (no libraries)  
  lib - build just the libraries (no application)  
  install - install the application. Read README for details  
  help - display this information
```

如果你对控制台程序的“使用说明 (usage statement)”很熟悉，那么对于我们现在所讨论的内容就会有所认识。我们在附录二中给出了一个构建文件目标示例，它将创建一个使用说明。

当前问题的最后一部分是关于文档的，它需要一个为工程生成 JavaDoc 的目标。JavaDoc 是工程中管理起来比较棘手的一个概念。JavaDoc 工具无法处理不能编译的代码。另外，与编译相比，JavaDoc 的处理非常慢。你不会希望开发人员每

次构建时都必须等待 JavaDoc 的处理。在编写你自己的 JavaDoc 目标时请考虑这些问题。

```
<!-- Generate the API documentation irssibot and the -->
<!-- modules -->
<target name="javadoc" depends="bot">
    <mkdir dir="${doc.dir}/api"/>
    <javadoc packagenames="irssibot.*"
              sourcepath="${src.dir}"
              destdir="${doc.dir}/api"
              author="true"
              version="true"
              use="true" >
        <classpath refid="classpath"/>
    </javadoc>
</target>
```

清除

问及以下问题时，作为结果，可能需要有一个或多个清除目标：

- 修改文件后如何重新构建工程？
- 需要删除所有的类文件吗？
- 需要删除所生成的 JAR 吗？

开发人员有时会忘记清除。这就可能导致问题，因为 Java 编译器的依赖关系检查程序并不能很好地确定类之间的每个依赖关系。而且，为了完成其自己的依赖关系检查，javac 任务要对已编译的类文件（相对于其对应的源代码文件）进行时间戳检查。尽管大多数情况下这都是有效的，但时间戳检查并不是最佳的。对于无依赖关系的类（注 5）、带有静态最终处理（final）的类以及其他特殊情况，即使编译步骤遗漏了某些类，仍可能导致成功的构建（从 Ant 的角度看是成功的）。正因如此，开发人员应该总是能够将构建过程中生成的所有内容都予以删除，并重新开始构建。只有这样才能保证所有需要编译的内容确实会得到编译。我们将这样的构建称为干净的构建（clean build）。

注 5： 在构建 Ant 自身时，这是一个严重的问题，因为会使用自省（introspection）来调用大多数类；对于这些任务不存在直接的依赖关系。

以下示例定义了两个目标，它们可用于确保干净的构建。

```
<!-- Delete class files built during previous builds. Leave directories -->
<target name="clean">
    <delete>
        <fileset dir="${build.classes}" includes="**/*.class"/>
    </delete>

</target>

<!-- Delete any created directories and their contents -->
<target name="cleanall" depends="clean">
    <delete dir="${build.dir}" />
    <delete dir="${dist.dir}" />
    <delete dir="${doc.dir}/api" />
</target>
```

在这些目标中，我们为irssibot构建提供了两种不同的干净构建解决方案。clean目标会删除类文件，对于在编译步骤中确保成功的依赖关系检查来说，这一步应该已经足够。cleanall目标会删除由以前的构建所生成的所有内容，实际上就是将工程还原到未做任何构建前的状态。

注意：在这个例子中，cleanall并不需要依赖于clean。不过，在实际应用中，clean目标可能不仅仅是删除文件。在这种情况下，我们希望Ant在cleanall中处理clean。为安全起见，默认地包括此依赖关系是一种不错的做法。

有时，可能需要在工程构建文件中包括一个distribution clean(发布干净)目标。此目标会删除所有生成的文件、目录以及所有源代码。听上去可能匪夷所思（从某种意义上讲，确实如此），但对于处在版本控制之下的工程来说，这是非常有用的。因此，如果你的工程不在版本控制之下，就不要删除源代码！从理论上讲，将一个工程仅发布为一个构建文件，并由其中的目标来获取或更新来自于某个版本控制系统（如CVS）的源代码（注6），这是有可能的。在我们这个例子中并没有提供一个distribution clean目标，因为irssibot不在版本控制之下。

注6：如果对你的发布服务器有严格的带宽限制，而对CVS服务器则无此限制，那么这确实会很方便。

发布

为此示例工程编写构建文件时，所要考虑的最后一件事就是如何发布此工程。我们需要回答以下问题：

- 在应用准备发布之前，需要创建哪些目录？
- 是否需要将源代码与应用同时发布？
- 应用的发布由哪些内容组成？

只需定义一个目标即可达到这些问题的要求。此工程的目录布局为我们提供了所需的最终结果。发布目录已经存在，构建所要做的只是将文件复制到这些目录中。

以下目标将创建发布目录，并复制类文件、脚本和最终应用的其他组件：

```
<!-- Deploy the application in a "ready-to-run" state -->
<target name="deploy" depends="bot,javadoc">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist.dir}" />
    <mkdir dir="${dist.dir}/bin"/>
    <mkdir dir="${dist.dir}/lib"/>
    <mkdir dir="${dist.dir}/doc"/>
    <mkdir dir="${dist.dir}/config"/>

    <!-- Copy the primary program and modules -->
    <copy todir="${dist.dir}/lib">
        <fileset dir="${build.classes}" />
        <fileset dir="${build.lib}" includes="irssibot.jar"/>
        <fileset dir="${build.lib}" includes="irssimodules.jar"/>
        <fileset dir="${lib.dir}" includes="*.jar"/>
    </copy>

    <!-- Copy the documentation -->
    <copy todir="${dist.dir}/doc">
        <fileset dir="${doc.dir}" />
    </copy>

    <!-- Copy the pre-fab configuration files -->
    <copy todir="${dist.dir}/config">
        <fileset dir="${lib.dir}" includes="*.xml"/>
    </copy>

    <!-- Copy the running scripts -->
    <copy todir="${dist.dir}/bin">
        <fileset dir="${bin.dir}" includes="bot.sh"/>
```

```
<fileset dir="${bin.dir}" includes="bot.bat"/>
</copy>
</target>
```

可以注意到这里在 bot 和 javadoc 目标上设置了目标依赖关系。我们只是要求在部署前应用是最新的。在所有目标中，deploy 最充分地利用了 Ant 的 fileset，因为此目标的任务要完成大量的文件操作。每个 fileset 都将只对我们需要部署的文件加以分组。例如，请看如下复制配置文件的任务：

```
<!-- Copy the pre-fab configuration files -->
<copy todir="${dist.dir}/config">
    <fileset dir="${lib.dir}" includes="*.xml"/>
</copy>
```

此任务只复制 XML 文件。配置目录（表示为 \${lib.dir}）中的其余内容均保持不动。

例 3-1 给出了这个完整的构建文件。

例 3-1: irssibot 工程的完整构建文件

```
<?xml version="1.0"?>

<!-- Comments are just as important in buildfiles, do not -->
<!-- avoid writing them! -->
<!-- Example build file for "Ant: The Definitive Guide" -->

<project name="irssibot" default="all" basedir=". ">

    <!-- Project-wide settings. All directories are relative to the -->
    <!-- project directories -->
    <property name="src.dir" value="src"/>
    <property name="doc.dir" value="doc"/>
    <property name="dist.dir" value="dist"/>
    <property name="lib.dir" value="lib"/>
    <property name="bin.dir" value="bin"/>

    <!-- Build directories -->
    <property name="build.dir" value="build"/>
    <property name="build.classes" value="${build.dir}/classes"/>
    <property name="build.doc" value="${build.dir}/doc"/>
    <property name="build.lib" value="${build.dir}/lib"/>
```

```
<!-- Global settings -->
<property name="debug.flag" value="on"/>
<property name="java.lib" value="${java.home}/jre/lib/rt.jar"/>

<!-- Global property for <javac> -->
<property name="build.compiler" value="modern"/>

<path id="classpath">
    <fileset dir="${lib.dir}">
        <include name="**/*.jar"/>
    </fileset>
</path>

<target name="prepare">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.lib}"/>

    <tstamp/>

    <echo message="\${TSTAMP}"/>
</target>

<target name="all" depends="bot,modules"/>

<!-- Build the IRC bot application -->
<target name="bot" depends="prepare">
    <mkdir dir="${build.classes}"/>
    <javac destdir="${build.classes}"
        debug="\${debug.flag}"
        deprecation="on">
        <!-- We could have used javac's srcdir attribute -->
        <src path="\${src.dir}"/>
        <exclude name="irssibot/modules/**"/>
        <classpath refid="classpath"/>
    </javac>
    <jar jarfile="\${build.lib}/irssibot.jar"
        basedir="\${build.classes}">
        <exclude name="irssibot/modules/**"/>
    </jar>
</target>

<!-- Build the IRC bot modules -->
<target name="modules" depends="prepare,bot">
    <mkdir dir="\${build.classes}/modules"/>
    <javac destdir="\${build.classes}/modules"
        debug="\${debug.flag}"
        deprecation="on" >
```

```
<!-- We could have used javac's srcdir attribute -->
<src path="\${src.dir}"/>
<include name="irssibot/modules/**"/>
<classpath refid="classpath"/>
</javac>
<jar jarfile="\${build.lib}/irssimodules.jar"
      basedir="\${build.classes}/modules"
      manifest="MANIFEST.MF" >
<manifest>
    <attribute name="ModuleType" value="irssibot"/>
</manifest>
<include name="irssibot/modules/**"/>
</jar>
</target>

<!-- Deploy the application in a "ready-to-run" state -->
<target name="deploy" depends="bot,javadoc">
    <!-- Create the distribution directory -->
    <mkdir dir="\${dist.dir}"/>
    <mkdir dir="\${dist.dir}/bin"/>
    <mkdir dir="\${dist.dir}/lib"/>
    <mkdir dir="\${dist.dir}/doc"/>
    <mkdir dir="\${dist.dir}/config"/>

    <!-- Copy the primary program and modules -->
    <copy todir="\${dist.dir}/lib">
        <fileset dir="\${build.classes}"/>
        <fileset dir="\${build.lib}" includes="irssibot.jar"/>
        <fileset dir="\${build.lib}" includes="irssimodules.jar"/>
        <fileset dir="\${lib.dir}" includes="*.jar"/>
    </copy>

    <!-- Copy the documentation -->
    <copy todir="\${dist.dir}/doc">
        <fileset dir="\${doc.dir}"/>
    </copy>

    <!-- Copy the pre-fab configuration files -->
    <copy todir="\${dist.dir}/config">
        <fileset dir="\${lib.dir}" includes="*.xml"/>
    </copy>

    <!-- Copy the running scripts -->
    <copy todir="\${dist.dir}/bin">
        <fileset dir="\${bin.dir}" includes="bot.sh"/>
        <fileset dir="\${bin.dir}" includes="bot.bat"/>
    </copy>
</target>
```

```
<!-- Generate the API documentation for the IRC library and the -->
<!-- IRC bot using the library -->
<target name="javadoc" depends="bot">
    <mkdir dir="${doc.dir}/api"/>
    <javadoc packagenames="irssibot.*"
              sourcepath="${src.dir}"
              destdir="${doc.dir}/api"
              classpath="${lib.dir}/xerces.jar:${lib.dir}/mysql.jar"
              author="true"
              version="true"
              use="true" />
</target>

<!-- Delete class files built during previous builds. Leave directories -->
<target name="clean">
    <delete>
        <fileset dir="${build.classes}" includes="**/*.class"/>
    </delete>
    <delete dir="${doc.dir}/api"/>
</target>

<!-- Delete any created directories and their contents -->
<target name="cleanall" depends="clean">
    <delete dir="${build.dir}"/>
    <delete dir="${dist.dir}"/>
    <delete dir="${doc.dir}/api"/>
</target>

</project>
```

构建文件执行处理

我们已经有了构建文件，但是 Ant 运行时会发生什么呢？理解 Ant 如何解析构建文件并执行目标，这对于写出好的、可靠的构建文件来说是至关重要的。

错误处理

Ant 会解释构建文件的 XML，这说明它会在解析元素时对其进行处理。Ant 所用的 XML 库表示了一个层次树结构；Ant 在处理时也遵循此树的路径。在工程级 (project level, 即 `<project>` 元素内的 XML 级)，Ant 会对 XML 元素做宽度优

先遍历。这说明，它会先加载并处理处在 `<project>` 元素下一级上的所有元素，然后再移至第一个目标。在目标内，Ant 要完成深度优先遍历。这说明，Ant 从目标的第一个元素开始，在移至下一个元素之前会尽可能向下（尽可能深）地处理各个元素。

在试图理解 Ant 如何处理自己的错误（相对于不正确的编译或失败的文件复制等错误而言）时，理解这种设计非常重要。在工程级，Ant 在对任何元素具体处理之前进行一种语法检查。一般来说，在谈及 Ant 处理某个元素时，我们是指 Ant 对该元素全生命周期的处理。假设有一个语法正确的元素声明，那么从外部看处理就是原子性的，在 Ant 解析此元素以及 Ant 完成相关操作（这些操作构成了元素定义的基础）之间，你不能插入任何操作。不过，处理元素时，在两个阶段中会出现错误，理解这些阶段有助于澄清认识。

工程级错误

在工程级上，Ant 会加载构建文件中的所有元素。它将处理除目标以外的每种元素。这说明任何工程级任务或 DataType 都将得到处理。处理目标意味着要运行该目标内的任务和 DataType。你并不希望在 Ant 加载构建文件时就执行所有目标。相反，可以认为 Ant 只是为将来的使用建立了一个列表。这个列表中包括有目标名和属性，还包括特定项中任何导致 Ant 失败的非法值。

对于工程级任务和 DataType，正如所料，会出现错误。读取一个 DataType 元素或执行 DataType 操作时出现的错误为构建错误，对于这些错误，Ant 将做如下处理：如果 Ant 发现一个未“预计”到的特定元素（如发现一个 `<notatag>` 作为 `<project>` 的子元素），这就是一个错误，构建将停止并失败。对于这些错误，要记住重要的一点：默认情况下，Ant 遇到第一个错误时即中断。可能在构建文件中有 100 个属性和元素错误，那么 Ant 会在每次执行时逐个地发现错误（你也如此）。另外，Ant 没有“回滚”的概念，因此 Ant 会立即中断构建，并可能带来可怕的后果。对此无从捕获，也没有办法消除。必须用一个监听者来扩展 Ant，从而对控制错误有所影响。出于这些原因，在写构建文件时务必要小心，特别是在编辑一个稳定工程的当前构建文件时更要注意。语法错误或处理错误都可能将你

的工程置于一种非确定的状态，这就要求你（或者更糟的情况下，还要求你的开发人员）全部重新构建。如果你的构建过程很长，就会浪费大量的时间。

目标级错误

目标级的错误也有与工程级错误类似的影响，只不过错误出现的时间稍有差别。在此 Ant 不是将一个 `<target>` 元素中所嵌套的各个元素全部都加载（即创建一个类似于 Ant 目标表的列表），而是逐个地加载和处理各个元素。当然，这就使得其顺序很重要。如果 Ant 达到第二个元素，则认为相应于第一个元素的操作是成功的，而且认为对于已处理完的元素，与其相关或由此创建的数据、文件或工程状态均是成功的。相反，若在工程级任务或 DataType 中出现一个错误，Ant 会认为其后的元素是未知的。

错误处理示例

下面用几个非法的构建文件来说明错误处理概念。我们将从以下构建文件开始讨论：

```
<project name="mybad" basedir=". " default="all">
  <property naame="oblivion" value="nil"/>
  <notarealtag/>
</project>
```

当 Ant 处理此构建文件时会发生什么情况呢？由于 `property` 是一个工程级 DataType，其非法的属性 `naame` 将导致 Ant 在试图调用与 `naame` 属性相关的设置方法却找不到时失败。Ant 不会显示关于 `<notarealtag/>` 元素的任何消息，这是因为 Ant 在第一次出现失败时即停止了。还要注意，尽管在此将 `<project>` 元素的默认值设置为 `all`，但这个构建文件并没有 `all` 目标。一旦修正了前两个错误（即非法的属性 `naame` 和非法的 `<notarealtag/>`），第三次运行时还会得到一个错误，表示没有 `all` 目标。Ant（以及你）每次只会发现一个错误。

以下是另一个不正确的构建文件，它基于前一个示例：

```
<project name="mybad" basedir=". " default="all">
  <target name="all">
    <notarealtag/>
```

```
</target>
<property name="oblivion" value="nil"/>
</project>
```

当Ant处理此构建文件时会发生什么情况呢？我们将property DataType移到了新加入的默认目标all之下。Ant在发现property DataType的非法属性之前能看到非法标签吗？答案是否定的。在目标级，Ant认为all完全正确，并直接移至非法的属性错误。当然，一旦你修正了此属性错误，Ant自然会通知你无法处理<notarealtag/>。

对以上示例加以调整，我们将修正属性和目标错误。另外，在此增加一个新的目标chaos，其中包括有非法的元素<notarealtag/>。以下是得到的代码段：

```
<project name="mybad" basedir"." default="all">
<property name="oblivion" value="nul"/>
<target name="all">
    <echo message="Hello there, all you happy people."/>
</target>
<target name="chaos">
    <notarealtag/>
</target>
</project>
```

现在Ant会做什么呢？Ant会按我们所要求的那样显示出消息：“Hello there, all you happy people.”。在此没有错误，是不是很令人惊讶？除非令chaos是all目标的一个依赖关系，或者直接从命令行调用chaos目标，否则Ant会忽略此chaos目标中的错误。这就是所谓的“腐烂”错误(fester error)的一个例子。诸如此类的错误会在很长一段时间内不被注意，而在不适当的时候“大发淫威”。要提前并经常进行测试，以避免这些腐烂错误。

对于并非与构建相关的错误，以上就是Ant的处理方法。既然已经知道了错误可能来自哪里，并且知道了如何避免，下面来看看一切正常时Ant所做的工作。

工程级数据元素和任务

Ant在执行任何目标前，会处理在工程级定义的所有元素和任务。当然，正如上一节所述，Ant还会建立一个目标列表，但对于目前而言，这并不重要。

注意: 工程级任务和数据元素很少。引入一个工程级任务或数据元素需要对核心 Ant 引擎做许多调整，因此将来也不太可能增加很多。目前，仅考虑以下工程级元素：property、path、taskdef、patternset、filterset、mapper 和 target。

对于我们的工程示例，工程级数据元素包括定义目录的特性、对应于 javac 任务的全局属性以及作为一个 path DataType 的编译类路径。Ant 按其出现顺序加以处理，使之对构建文件的其余部分全局可用。可以看到，顺序对于相关的特性来说非常重要。

下面花一点时间来讨论特性。特性有两个主要特点。它们是不可变的，而且不论其在哪里定义，总是有全局作用域。不可变性说明一旦 Ant 首次处理了特性名-值对，那么该特性的值就不能再改变。在设计工程和编写构建文件时，这是要记住的很重要的一点。许多刚开始接触 Ant 的人都会错误地把特性看作是脚本中的变量，并期望它们有类似的作用。特别是，Ant 允许重新声明特性，因此在你修改其值时不会抛出错误，而这更是雪上加霜。Ant 定义了声明特性的优先顺序。在 Ant 命令行声明的特性总是比在其他位置定义的特性有更高的优先级。在此之后，Ant 会根据其何时首次发现所声明特性来确定优先级。

不可变性影响着如何解析特性值。我们将利用以下代码示例来加以说明：

```
<property name="property.one" value="${property.two}:one"/>
<property name="property.two" value="two"/>
```

property.one 的值是什么呢？由于 Ant 的特性解析是有顺序的，所以其值为 \${property.two}:one，而不是 two:one。通常情况下，在定义深度渐增的目录时，可以依赖于此做法。如果突然发现你在创建一个名为 \${property.two} 的目录，可能会非常不安。要记住这是顺序所致，你并没有出错。

特性的另一个重要特点是它们都有全局作用域。一个特性有全局作用域说明它是一个全局变量。来看以下的构建文件片段：

```
<property name="prop1" value="one"/>

<target name="target1"> <property name="prop2" value="two"/>
```

```

<echo message="\$\{prop1\}:\$\{prop2\}" />
</target>

<target name="target2" depends="target1">
    <echo message="\$\{prop1\}:\$\{prop2\}" />
</target>

```

target1 定义了特性 prop2。由于所有特性都有全局作用域，所以一旦 Ant 处理了 target1，prop2 对构建文件中的其余部分都将可用。

级联构建文件

级联构建文件可以改变特性不可变性和作用域的规则。开发人员有时在大型工程中使用级联构建文件，这些工程有许多子工程，而每个子工程又有自己的构建文件。位于工程根的主构建文件执行一个或多个子工程构建文件，从而构建部分或全部工程。希望构建单个子工程的开发人员可在该子工程的目录上运行构建文件，这样可以在其日常工作中合法地忽略其他的子工程（这也是之所以要进行设计的原因）。对于这种使用级联构建文件的工程，一个公开的例子就是 Jakarta 的 taglibs。附录二中，我们提供了有关编写级联构建文件的一节，另外对于特性的不可变性（及可能的可变性）会带来的问题，还将提供对其进行管理的技巧。

目标

运行 *ant* 时若未带任何参数，Ant 将读取 `<project>` 元素，并使用 `default` 属性来得到要执行的第一个目标名。在我们这个例子中，该目标名为 `all`。`all` 目标相应地依赖于 `bot` 和 `modules` 目标（译注 2）。这说明，Ant 在运行 `all` 内部的任何元素之前，会先执行这两个目标（目前暂不考虑 `all` 目标不包含任何元素）；而且这些目标必须按顺序成功地完成，这样 Ant 才能开始处理 `all`。由于在此 `all` 目标中没有任何元素，因此 `bot` 和 `modules` 目标的成功就相当于 `all` 目标的成功。

译注 2：原文此处为 `module`，有误，因为目标名为 `modules`，而不是 `module`，后面有多处此类错误均已做修改。

bot 目标

由于这是 all 目标列表中的第一个依赖关系，所以 bot 目标将先运行。bot 目标的用途是编译应用，并将其打包到一个 JAR 文件中。bot 目标也有一个依赖关系：prepare 目标。prepare 目标创建编译步骤所需的临时构建目录。它所用的 mkdir 任务总能成功，即使 mkdir 试图创建的目录已经存在也不例外。只有当 I/O 系统由于文件权限、空间限制或其他硬件或操作系统错误等原因抛出一个异常时，mkdir 任务才会失败。除了创建目录外，prepare 目标还会使用 tstamp 任务来记录构建的时间。tstamp 任务没有属性，也不向控制台或日志输出任何内容。相反，它会设置特性，而这些特性将在后面用到，主要是在 echo 任务中使用，另外任何其他需要日期和时间的任务也将用到这些特性。对于 tstamp 任务的详细内容请参见第七章。

javac 任务将编译 Java 源代码。下面对 javac 任务详细加以分析，它在 bot 目标中定义如下：

```
<javac destdir="\${build.classes}"  
       debug="\${debug.flag}"  
       deprecation="on">  
  <src path="\${src.dir}"/>  
  <exclude name="irssibot/modules/**"/>  
  <classpath refid="classpath"/>  
</javac>
```

对于每个 javac 任务，有三个必要的设置：

- 源目录
- 类路径
- 目标目录

我们将源目录 (source directory, 存储 Java 源代码的位置) 指定为嵌套的 DataType src (注 7)。也可以使用 srcdir 属性，但为了演示之用，在此选择使用一个

注 7： 通过自省方法的一个小技巧，javac 任务类隐藏了一个事实，即 <src> 只是名字有所不同的一个 <path> 元素。对于其他任务，没有可用的、称为 src 的 DataType，不过其他任务也可利用 javac 的这个编程技巧。

`DataType`。在实际应用中，可能使用 `srcdir` 属性更为常见。我们用类似的方法指定编译器的类路径，在此使用了 `classpath` `DataType`。这里，我们使用了一个引用 ID 来引用前面的路径定义。在该构建文件的前面，我们曾定义了一个类路径，其中包括了 `/lib` 工程目录中的所有 JAR，在此将它的引用 ID 置为 `classpath`。为了在以后使用该路径（如同在 `javac` 任务中一样），我们声明了一个类似的 `DataType`，它有一个属性 `refid`。这里将 `refid` 置为另一个 `DataType` 的引用 ID（该 `DataType` 为前面定义的类路径 `path` `DataType`）。Ant 对这些 `DataType` 的值进行管理，从而使你可以一次定义一个 `DataType`，而在别处多次引用。需要说明的重要一点是，`DataType` 引用不同于特性，只在同一个构建文件中起作用（注 8）。

对于已编译类的目标目录，我们使用了 `destdir` 属性来指定此信息。由于目标目录总是一个单个目录，而不是一组文件或一个目录路径，因此这里使用的是一个属性（attribute）和一个特性（property），而不是 `DataType`。

到此为止，我们已经讨论了对于 `javac` 所需的设置，不过，你可以注意到，这里还指定了一组可选的属性和 `DataType`。可选属性为 `debug` 和 `deprecation`；可选的 `DataType` 为 `exclude`。

由于我们仍在开发 `irssibot`，所以很可能会试图在一个调试程序中运行它。这就需要在编译时打开调试标志，对此用 `javac` 的 `debug` 属性来表示。由于我们需要它作为一个全局选项，因此使用了一个特性，并在构建文件开始处设置一次。注意 `yes|no` 和 `true|false` 值适用于诸如 `debug` 这样的布尔（Boolean）属性。

默认情况下，不同的 Java 编译器并不提供有关废弃方法调用的详细信息（注 9）。假设 `irssibot` 使用了一个废弃的方法或字段，编译器只在普遍意义上提示我们使用了废弃调用。它并不会通知哪个方法或者哪个类使用了废弃调用。为了得到详细的信息，在此使用了 `javac` 的 `deprecation` 属性，并将其置为“`true`”（或“`yes`”）。

注 8：Ant 1.5（预计在本书出版后发布）可能对跨构建文件环境引用 `DataType` 提供一个解决方案。

注 9：关于废弃方法和字段的更多内容请参见 David Flanagan 所著的《Java in a Nutshell》（O'Reilly）。本书中文版《Java 技术手册》已由中国电力出版社引进出版。

为了区别模块代码和应用代码，类的包结构被分解为两个子包，其中之一即为 *modules*。我们并不希望这些类作为应用 JAR 的一部分，因此使用了 `<excludes>` 元素来通知 `javac` 不要对它们进行编译。这里的 `<excludes>` 元素将通知 `javac` 排除其 `fileset` 中的所有文件，在这种情况下，即为 *modules* 包目录下非依赖的源代码。

综上所述，我们将通知 `javac` 完成以下工作：

- 编译 `${src.dir}` 中的源代码，但排除 *modules* 包中的 Java 文件。
- 将新构建的类文件发送到构建目录中，构建目录由 `${build.dir}` 特性定义。
- 在类文件中包括调试信息以用于调试程序。
- 提供详细的废弃错误消息，以表明哪些类和调用是废弃的。
- 如果 `javac` 中存在任何操作失败，则导致 `bot` 目标失败。

再回过头来考虑大约 11 行 XML，我们在构建中定义了一个步骤，它总能用正确的类路径编译正确的 Java 文件，而不论将来增加或删除了多少源文件或库。除非此工程的需求（而不仅是参数）有所变化，否则绝对不会对构建文件的这一部分构建文件进行修改。如果确实需求改变，则可重新建立目标（goal），并相应地修改目标。这是可以预料的。作为意外收获，XML 具有详细且适合阅读的特点，这就创建了一个易于维护的构建描述文件。要记住，一个新的目标意味着对构建文件的编辑，但小的工程修改则无需对构建文件进行修改。如果发现需要频繁地修改自己的构建文件，那么就应花一些时间来重新进行设计。最终的目的是只需一次编写构建文件，并尽可能地不再考虑它。

依赖关系检查

即使 `javac` 任务指定了要排除的情况，你也可能会注意到，编译器仍对模块子包中的源代码做了编译。从构建的角度看，如果应用中的代码引用了模块中的代码，那么这一点是无法避免的（注 10）。根据 Java 编译器规范，Java 编译器要负责在

注 10： 我们当然可以通过编写代码来避免这种循环的依赖关系。在此选择这个特定的应用是因为它能体现出这种相互依赖的情况，使我们可以对之加以讨论。

编译时解析所有依赖关系。在编译各个类时，它要在逐个类的基础上完成依赖关系检查。换句话说，如果类 A 依赖于类 B 和类 C，那么在编译 A 时，Java 编译器必须找到已编译的 B 和 C。如果它们不存在，编译器就必须找到类 B 和类 C 的源代码，并在编译类 A 之前对其进行编译。

对于创建其工程对象模型的开发人员来说，管理依赖关系的任务完全落在他们的肩上。因此，Java 类依赖关系及其管理方法等概念就超出了本书的范围。对于 Ant 而言，依赖关系检查是一个自动完成的工作。

打包类文件

Ant 编译了源文件后，就会生成类文件，bot 目标将使用 jar 任务来将这些类文件打包到一个 JAR 中。我们只需 4 行 XML 代码即可完成此工作：

```
<jar jarfile="${build.lib}/irssibot.jar"
      basedir="${build.classes}" >
  <exclude name="irssibot/modules/**"/>
</jar>
```

jar 任务将 *build.classes* 目录中的所有文件（除了模块包目录下的文件）放在一个称为 *irssibot.jar* 的文件中。

modules 目标

modules 目标几乎等同于 bot 目标。javac 和 jar 任务也几乎用了相同的属性和 DataType。惟一的差别在于 javac 和 jar 所排除的 DataType 有所不同。对于 bot 目标，我们显式地排除了模块子包目录下的文件。而在 modules 目标的情况下，我们显式地包括了模块目录中的文件，这就间接地排除了所有其他文件。

将模块子包目录中的文件包括进来（实际上就是排除了其他的文件），其结果是我们的构建将产生两个 JAR，分别有互斥的两组类文件。这一结果满足了前面设置的需求，即表示我们需要两个包：一个用于应用，另一个用于模块。

modules 和 bot 目标均会默认地运行，这是因为 all 目标对它们有依赖关系。all

目标并不包括对发布、文档或清除的依赖，因此 Ant 不会执行这些目标，除非用户在运行时显式地在命令行上做出要求。

其他目标

除了用于编译和打包 irssibot 工程所用的 `bot` 和 `modules` 目标外，我们的构建文件还有其他的目标，分别用于生成文档、构建后的清除以及部署。

javadoc 目标

`javadoc` 目标用 JavaDoc 工具编译动态生成的代码文档。`javadoc` 任务的操作类似于 `javac` 任务。它们都对 Java 代码进行语法检查：对于 `javac`，这是一个预编译步骤，而对于 `javadoc`，则是为了确保文档至少表示了将编译的代码。大多数 JavaDoc 来自于开发人员所编写的类、字段和方法注释，但也有一些是动态生成的，而这也正是为什么代码必须编译的原因。

对于这个目标，我们向现有的文档目录 `doc/` 中增加动态文档，该目录在另一个称为 `api/` 的、单独的目录下。这样，在发布目标执行时，只需对 `doc/` 目录下的内容进行打包或复制即可。有了 `javadoc` 目标，我们还为发布目标提供了一个依赖关系。这对于发布很有帮助，可以确保 `javadoc` 的运行，从而提供最新的代码文档，而且如果不能创建最新的文档就会失败。当然，前面已经提到过，其他目标不应依赖于 `javadoc` 目标。JavaDoc 工具要完成文档生成的时间相当长，比编译步骤本身要多出数倍。

清除

清除工程目录的目标是任何构建中最重要的目标，其重要性甚至超过了编译目标。为什么这么说呢？因为软件开发是一个确定性的操作。无论你的工程如何简单或如何复杂，都会以一种确定性的方式运行。你的构建也不例外。如果在工程中未做其他修改，那么在早上 8 点完成的构建若与在 9 点完成的构建有所不同，则对此将绝对无法解释。这种确定性的行为应当正是最初创建一个构建过程的原因。

清除目标通过为你和你的开发人员提供一种“重置开关”来达到这一目标。你可以（而且总是能够）将工程返回到编译前的状态。为此我们使用了两个目标，因为从理论上讲存在两个开始点。第一个即为全新的工程（fresh project）。当你首次下载zip/tar/jar或由版本控制系统运行一个下载版本（checkout）后，就存在这种工程状态。当工程增长到包括700个以上的类，而且包含了多个包和子包时，对你所做的所有修改进行跟踪可能变得极为麻烦。有一个可以有效地对工程进行重置的构建步骤是非常重要的，甚至是必不可少的。如果没有这样一个步骤，开发人员就必须对构建进行逆向工程（reverse-engineer）操作以找出对系统所做的所有修改。

部署和安装

部署和安装Java工程可能是一件很麻烦的事情。实际上，我们建议，如果你正要开始管理工程和编写构建文件，那么应当暂且不写安装目标，直到工程处于稳定状态时才做此工作。如果我们要为某个平台（例如一个Linux的RedHat发布）编写程序，则有一个很容易的安装目标。可以建立一个RPM（部署步骤），并运行一些RPM命令（安装步骤）。而对于Java开发人员来说，问题就没有这么简单了。注意，对于Ant以及某些Ant发布（其自己的构建文件中有一个install目标），我们专门有一章来介绍有关其安装和配置的问题。对于所有安装，关键是，你作为工程管理人员很少知道其他管理人员和开发人员如何管理其自己的服务器和工作站。Ant工程则确实使这个工作变得很容易。它假设在工作站上只有一个JRE（Java Runtime Environment，Java运行时环境），并有一些应当运行在许多平台上的脚本。安装只需一个诸如/usr/local/ant或c:\ant的根安装目录，然后就万事大吉了。

对于irssibot，我们要为deploy目标创建一个可发布的包，但要由使用此程序的人来确定如何使用此包安装。为了使问题简单化，我们并不打算理解其他工作站的结构。你可以说irssibot是自包含的，它并不试图做超出其工程目录之外的任何事情。我们创建了一个dist/目录，并在其中放置所有JAR、文档以及组成最终程序的脚本。作为练习，你可以将一个安装目标的编写考虑为类似于Ant的编写。需要在命令行设置一些特性（安装目录），目标会用它来将dist/目录下的所有内容复制到安装目录。

到目前为止，安装看上去都很容易，你可能会奇怪为什么在我们自己的工程中没有一个这样的目标。其原因在于另外一部分 Java 开发人员，即服务器端的 Java 开发人员。直到部署这一步，我们的示例已经涉及了 Java 开发的所有方面。对于 Web 应用或 EJB 应用，`deploy` 目标会构建 WAR 和 EAR。当然并非所有应用服务器都支持 WAR 和 EAR（如 BEA 的 WebLogic 5.1 就不支持 EAR）。对于这些开发人员来说，安装就非常困难，而且我们不希望它看上去是个简单的步骤。更好的做法是，由你的构建创建一组可部署的目录和包文件，然后先停下来。此时你可以查看将如何安装应用，并确定是否可以继续。

Ant 并非脚本语言

在了解了示例工程和构建文件之后，你可能会把 Ant 看作是一种用于构建的脚本语言。有了这种看法，你可能会基于这一概念“勇往无前”地编写构建文件，直到出现了问题而无法继续，此时你会疑惑为什么 Ant 不能像脚本语言那样按你所期望的工作。其原因在于，XML 不适于提供一种好的脚本语言。

从某种意义上说，把 Ant 看成一种 XML 脚本语言和相应的解析程序也是情有可原的。但区别在于，作为一种脚本语言，Ant 并不是很好。实际上，它相当糟糕。这种理解上的小偏差可能导致许多误解和失败。要把构建看成是一种设计，而不是一系列步骤，这样可有助于减少误解。我们支持这种制作技术。那么作为一种脚本语言，Ant 的 XML 语法究竟哪里有问题呢？

居然没有数据结构！

关于 Ant 语法中的怪异之处，一个更具体的例子是其对数据的管理。在此，与语言变量最为相近的术语是 `<property>` 标签。这样当然就完全忽略了 XML 丰富的数据功能，而 Ant 的开发人员也知道这一点。除了 `property` 之外，Ant 中还存在一个数据元素的概念，如 `path DataType`。其限制在于，你无法像在脚本语言中那样，“在该语言中” 创建 `DataType`，实际的做法是，要在 Ant 中表示一种新的数据类型，就必须编写一个类（或一组类）。比起对各组数据值简单加以封装来

说，这样可能更为费劲。如果你把Ant看成是一种基于XML的构建脚本语言，并且希望创建你自己的数据元素，那么很快就会陷入此困境。

那么如何避免呢？你当然可以通过程序来修正其中一些缺陷，但是只有在你乐于努力为Ant编写扩展和任务时这才有可能。作为选择，如果你不想或不能通过程序扩展Ant，那么你能做的其他工作将很有限。可以利用Ant邮件列表并阅读有关文档，从而了解Ant开发人员在重构(refactor)这种设计限制时所做的其他更多工作。这些资源很不错，而且重构势必很快就会实现。例如，Ant开发人员仅在6个月内就在两个版本间引入了path DataType的概念(2000年4月的Ant 1.1到2000年10月的Ant 1.2)。作为一个开源的工程，这意味着Ant开发人员的进展很快，并在数月内即能完成工程的重构。

DTD 在哪里？

如果说Ant不是一种脚本语言，而且它使用了XML，那么在Ant解析构建文件时我们就应当能够对它进行验证。这种验证需要一个DTD。但我们却没有这样的东西，对此存在几个原因。

对运行时和解析时处理的描述非常复杂，不是吗？这是因为Ant的内部处理设计本身就很复杂。与其说Ant是故意这样设计的，不如说这是必要之举。由于Ant使用了一种有着定义完善的语法规则的语言，所以它就必须总是遵循这些规则；它通过使用现有的XML库来加载构建文件，从而实现这一点。构建文件只有在得到加载时才被认为是“良构”的。再来分析这句话。Ant不验证文件的完整性，而是在读取XML元素时对它们进行验证。另外，“语法上正确”和“良构”不是同一个概念。为了在加载时做到语法上正确，XML需要有一个相应的DTD(或模式)。但它并没有，而且也不能有(稍后将对此做更多解释)。作为补偿，Ant在XML中做迭代处理，仅仅解析和执行那些需要执行的元素，并在此过程中检查其语法。这样做有一个好处，即可以使Ant更快，因为如果它试图做完全的语法检查(特别是对一个大型的构建文件来说)，Ant就会很慢，而且更有可能的是，作为一个需要大量内存的程序，它还会比现在需要更多的内存。

如果没有一个 DTD 或模式，良构但语法不正确的 XML 就可能会长期存在而不被发现。在 Jakarta 小组发布 Ant 的一个新版本时，这个问题更为严重了。考虑一个不常使用的目标。对这个目标做修改，并用 Ant 1.4.1 来测试。它能工作得很好，而且一切正常。使用模式表明开发人员每月仅使用此目标一次或两次。3 个月后，Jakarta 小组发布了 Ant 1.5 版本，原来在 1.4.1 版本下工作的任务有了一个新的语法。由于不常使用，该目标将继续保持原样而未得到回归（regression）检查。只有在将来出现一个构建错误后，你才会发现这个目标有问题。

应当有一个 DTD 吗？从理论上讲是不能有的。由于定制的任务模型，Ant 的构建文件 DTD 应当随每个新的任务而有所改变。创建一个 DTD 的任务是存在的（`antstructure`），但它仅创建核心任务模型的 DTD。另外，它会忽略任务所需的属性。确定了 Ant XML 语法后，许多用户都曾使用 `antstructure` 的输出作为起始点，并花大力气来建立自己的 DTD。遗憾的是，由于上述关于新任务的问题，没有任何解决方案堪称完善。要验证你的构建文件，就应当对它进行测试，并经常测试。

流控制

最初考虑构建设计时，你难免会以一种过程流的方式来对待它。你甚至可能会用一个流程图来描述各个步骤。流控制需要两个重要的特征，即条件和迭代，而它们在 Ant 中（几乎）是没有的。

条件允许我们根据设置的值或在运行时改变构建流。例如，你可能希望在另一个目标失败时，构建能够运行某个特定目标。在正常版本的 Ant 中是没有这一级别的通用条件控制的（注 11）。如果某个目标失败，所有依赖于它的目标都将失败。除了重写或重新设计任务来处理这种错误事件外，你无法避免这种情况。更有可能的是，你需要编写一个完全不同的任务，它可理解特定的条件，并执行某些条件下所需的不同构建步骤。对于简单任务而言尚可，但可以想见，重写整组任务（如两个或三个 Java 编译、一些文件复制以及打包为 JAR）是相当困难的。

注 11： 条件任务确实存在，但我们认为它们是实验性的。不要把条件任务和流控制弄混。

在构建中应用迭代，这意味着要基于某个条件或一组条件，将一个任务（如编译文件）或一组任务执行多次。你可能认为，在Ant语法中没有显式的条件，就不可能有迭代。你是正确的。不过，即使有条件，也不能要求Ant在一组正在改变的DataType基础上执行任务。作为迭代所需的操作，一个常见的例子就是递归文件操作。假设有一个带有4个子工程的工程。各个子工程之间惟一的差别在于其子工程的根目录名有所不同。你希望对各个子工程进行编译和打包，而且（基于这本书的帮助）已经建立了此工程的目录，从而可以以一种很高效的方式完成要求。作为一个好的设计人员，你意识到可以将一个子工程中的目标重用于各个子工程中，而每次只是修改少数几个特性。在你实现此解决方案时，却会遭到失败。Ant无法重用诸如此类的目标。你只能将此目标剪切并分别在各个子工程中粘贴3次，然后再显式地定义各个子工程。如果以后删除或增加了一个子工程，还必须重新编辑构建文件。如果使用某种形式的级联构建文件，也同样存在这个问题。某些情况下，你必须显式地定义子工程，这说明有些内容（可能是一个特性文件、一个XML数据文件或是其他构建文件）必须得到编辑从而保证修改是完整的。

如果没有定制任务的概念，Ant的生命力就不会很强。你也许可以用一个XML文件、某些XSLT和一些定制的任务来解决条件或迭代问题，但这仍是你自己的解决方案，而不是Ant的。你创建的构建文件现在不再是可移植的，因为必须用你的构建文件来发布你的Ant修改。这只是小问题，但也同样麻烦。如果不能对Ant扩展，我们所能做的工作就很少。因此你的设计必须考虑到这些限制，从而不至于陷入困境而需要完全重构（或更有可能重写）你的构建文件。

构建文件授权问题

相对于Ant提供的功能而言，本章所介绍的示例构建文件是很简单的。我们的目的是展示如何建立一个工程、组织文件并编写一个构建文件，从而使开发人员和工程管理人员更为轻松。这里所用的工程和步骤都有所夸张和扩展，以便更好地说明如何就有关工程的组织和构建文件的设计做出决定。

除了这些步骤，要写出更好的构建文件的最好捷径莫过于经验，这包括研究现有的构建文件和编写新的构建文件。大多数主要的开源的Java应用目前都使用Ant，

这就相当于提供了一个最佳（和最差）实践的虚拟仓库。作为练习，你可能希望采用其中某个构建文件，而且以这本书为参考，对其加以注释，并指出你认为发生了什么情况，以及为什么会出现这些情况。

以下问题尚未提及，因为它们更应作为 Ant 缺陷的解决方法，而不是构建文件设计指导。在最理想的情况下，Ant 的开发人员要重构设计以消除对这些解决方法的需要。

魔法属性

有些属性得到了设置，但从未在任何目标或任务中显式使用。Ant 的对象模型允许任何构建组件都能看到构建的所有属性。这种构建文件内引用存在一个缺陷，就是要将特性标志为魔法特性（magic property）。作为魔法属性，一个很好的例子就是 `javac` 任务的 `build.compiler` 特性。不存在任何特性来指定 `javac` 任务所用的编译器类型，而是要依赖于定义 `build.compiler` 的构建文件。对于使用了一个魔法特性的任务，我们要在任务定义（附录二中将提供）中加以指定。

在编写你自己的任务时，要尽量避免使用魔法特性，因为这样会使构建文件不可读，而且难于维护。

失败何时可看作为成功

考虑以下构建文件片段：

```
<copy todir="newdir">
    <fileset dir="foo">
        <include name="**/*.xml" />
    </fileset>
</copy>
```

此 `copy` 任务元素将所有 XML 文件由 `foo` 复制到 `newdir`，如果不存在 `newdir` 则创建一个。那么，如果 `foo` 不存在，会发生什么情况呢？或者如果根本没有 XML 文件，又会有什么结果呢？

在不存在源目录（此例中为 `foo`）的情况下，所发生的是 `copy` 失败。若没有要复制的文件，则 `copy` 成功，但不会创建一个目标目录（此例中为 `newdir`）。这种行为会导致一个有意思的问题：如果你的构建只在特定条件下创建 `foo`

目录，并生成 XML 文件，那么会怎样呢？对于未生成 XML 的情况，在完成 copy 任务时，你希望整个构建都失败吗？是？不？还是可能？在 Ant 1.4.1 版本以前，你无法对此加以控制。一种解决办法是手工地创建 *foo* 目录，从而保证任务不会失败。从 Ant 1.4.1 开始，copy 有了一个 failonerror 属性，从而允许你控制其失败状态。通过使用 failonerror，你就可以要求 Ant 将一个失败的复制仍看作是成功的。

可以得出一个教训，即在假设一个任务满足构建流之前，要注意什么会使之失败。你绝不希望早上 4 点钟所做的自动构建失败，从而导致一天的测试尽失，而原因只是你对某个任务的失败状态有所误解。

在设计构建和工程布局时，如果有所困惑，请记住没有哪一种构造工程和编写构建文件的方法是绝对正确的。总是会有一些特定的情况或特定的需求，使你无法采用我们在此（以及其他章）所提供的布局和模式。另外，Ant 还相对年轻，必定要进行调整，这就要求你必须与之共进。对 Ant 的改进以及将来的定制任务，这些都可能使本书所描述的一些技术被废弃；需要提醒一句，这些技术不是不好，只是被废弃了。

使用此示例工程的布局，并采用我们所遵循的步骤，可以帮助你在设计和编写自己的构建文件时无需自己做更多的工作。这里所提供的过程来自于我们的研究与实践，从 Ant 公开发布的第一版起我们就开始使用 Ant 了。你会发现许多工程布局和构建文件技术在诸如 JBoss、Tomcat 甚至 Ant 本身中都是相同的。这并不表示这些工程设计是最佳的，而只是说明它们是流行的，而且还将将在一段时间内继续流行。

第四章

Ant DataType

在前一章的构建文件示例中，我们已经看到，为部署该 irssibot 应用，使用了 `fileset` `DataType` 以标识所要复制的文件组。在使用 Ant 时，`DataType` 非常重要，而且 `fileset` 只是可用 `DataType` 中的一种，以下列出其他可用的 `DataType`:

argument

对于由一个 Ant 构建文件调用的程序，向其传递命令行参数。

environment

对于由一个 Ant 构建文件调用的外部命令或程序，指定向其传递的环境变量。

filelist

定义一个文件的命名列表，这些文件无需确实存在。

fileset

定义一个文件的命名列表，这些文件必须确实存在。

patternset

将一组模式分组在一起。

filterset

将一组过滤器分组在一起。

path

以某种在不同操作系统间可移植的方式指定路径（如类路径）。

mapper

定义一组输入文件和一组输出文件间的一个复杂关系。

下面将深入介绍这些基本的Ant DataType。它们是任务所用的构建块，而且对于合法地使用Ant来说至关重要。在这一章中，我们将详细地对各个DataType逐一加以介绍。不过，在此之前，我们将简要讨论DataType何以适合于Ant的整体设计，并解释本章描述不同DataType的属性时所用的记法。

已定义 DataType

Ant DataType见于org.apache.tools.ant.types包，通常由org.apache.tools.ant.types.DataType基类派生得到。EnumeratedAttribute、CommandLine、Environment和Reference也被看作是DataType，不过它们并非由DataType派生。图4-1包括了一个基本UML类图，它说明了Ant设计的这一方面。

基类org.apache.tools.ant.ProjectComponent提供了日志功能，另外还可以访问Project对象。虽然在此没有显示，不过ProjectComponent也是各个Ant任务的基类。这些任务详见于第七章和第八章。

尽管这个类图有助于解释什么是DataType，但理解Ant的内部结构往往并不必要。大多数情况下，你只是希望编写构建文件并且使用Ant而已。出于这个原因，本章余下的部分将把重点放在如何使用这些类型上，而不是其内部实现是如何工作的。

XML 属性约定

与任务一样，DataType也使用属性来定义。本章中在对各个DataType进行讨论时，还会列出各DataType的全部可用属性。这些列表描述了各个属性，说明了支持它的Ant版本，并且指出了该属性是否必要。属性列表形式如下：

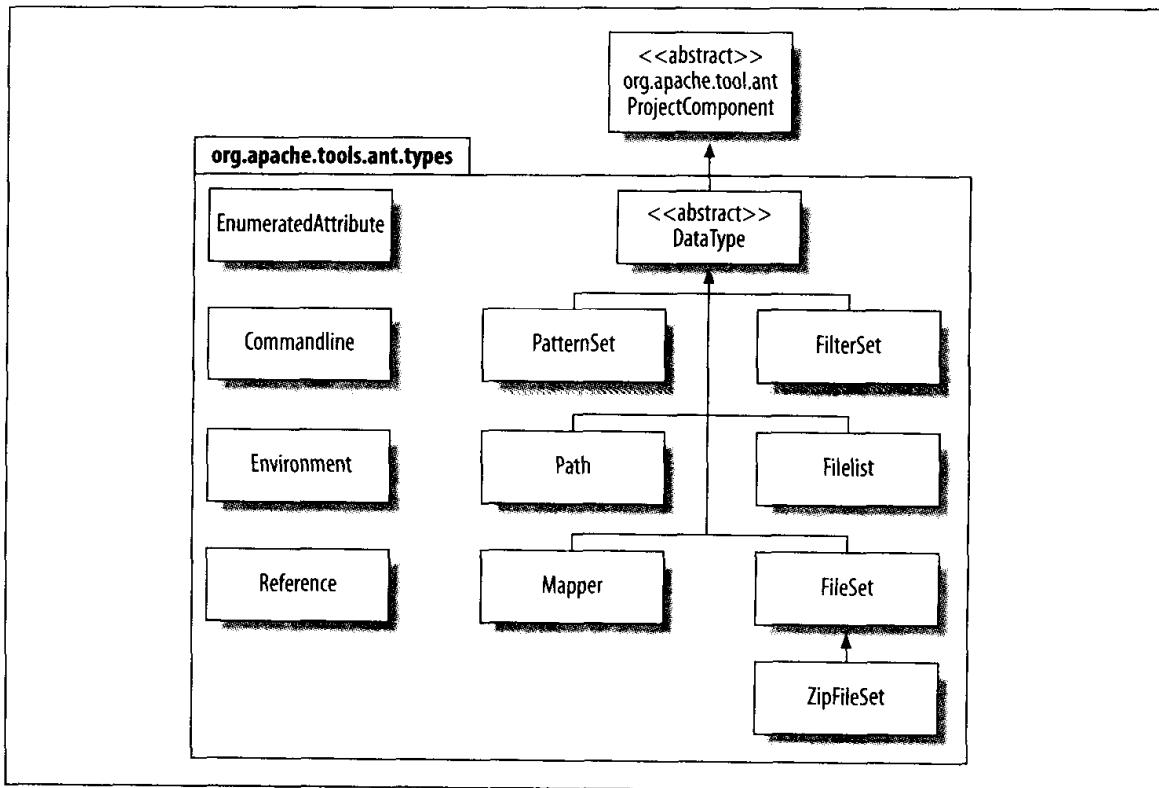


图 4-1: Ant DataType

attribute_name (version, type, required_flag)

这是对一个属性及其功能的描述。

其中：

attribute_name

是属性名。在为一个任务指定属性时，即用此来引用。

version

指出支持该属性的 Ant 版本。“all”表示 Ant 1.2 及以上版本。

type

指出一个属性所能保存的数据类型。例如，String 表示某属性可保存文本数据，如表 4-1 所示。

required_flag

指出在使用任务时某给定属性是否必要。如果此标志是一个星号 (*)，则请参见列表之后的注释。

属性描述

对属性及其功能的描述。

表 4-1 对本章经常引用的属性类型做了总结。在各种情况下，XML 属性的文本都将转换为在此所列出的某个基本类型。“描述”列对各种转换如何完成做了描述。“由 … 实现”列则列出了 Ant 表示各个属性类型所用的 Java 类。

表 4-1: XML 属性类型总结

类型名	由 … 实现	描述
boolean	N/A	完成不区分大小写的字符串比较，将 on、true 和 yes 转换为 true。所有其他值均转换为 false
Enum	org.apache.tools.ant.types.EnumeratedAttribute	用于允许有固定的字符串值集合的情况
File	java.io.File	指定某个文件或目录的名字。除非另做说明，否则文件和目录名都是相对于工程基目录的。稍后将描述的 fileset 和 filelist 允许指定多个文件
int、long 等等	N/A	诸如 java.lang.Integer 等标准 Java 类型包装器类会处理此类转换，即将构建文件中的文本转换为基本类型
Path	org.apache.tools.ant.types.Path	最常用于 classpath 和 sourcepath 属性，表示一个用 “:” 或 “;” 分隔的路径列表。对此将在“Path DataType” 中详细描述
Reference	org.apache.tools.ant.types.Reference	常用于 refid 属性，并包括对某处定义的一个类型 id 的引用。请参见第七章中的 java 任务，其中显示了如何引用一个在构建文件中某处定义的类路径
String	java.lang.String	这是 Ant 中最常用的类型。字符串（及其他属性）要服从 XML 属性限制。例如，< 字符必须写为 <；

argument DataType

apply、exec 和 java 任务均接受嵌套 `<arg>` 元素，可以为其各自的过程调用指定命令行参数。`org.apache.tools.ant.types.Commandline.Argument` 类实现此 DataType（注 1）。如果指定了多个 `<arg>` 元素，则每个元素都分别被看作是过程调用的一个参数。以下是所有 `<arg>` 属性的列表：

`file (all, File, *)`

作为一个参数的文件名。在构建文件中，此文件名相对于当前的工作目录。“当前工作目录”依此类型所用的环境而有所不同。在作为一个参数传递时，此文件名要转换为一个绝对路径。

`line (all, String, *)`

用空格分隔的多个参数的列表。

`path (all, Path, *)`

路径，将在后面的“path DataType”一节中详细解释。

`value (all, String, *)`

一个命令行参数。如果你的参数中有空格，但又想将它作为单独一个值，则使用此属性。

需要有一个（且仅有一个）以上属性。

示例

下面来看一个完整的构建文件，从而使你有一个直观的认识。在例 4-1 中，我们使用了 java 任务来调用 Apache 的 Xalan XSLT 处理程序，通过使用 XSLT（注 2）将一个 XML 文件转换为 HTML。正如所料，java 任务会调用带有 `main()` 方法的任何 Java 类。可使用 `<arg>` 元素向 java 任务传递参数。

注 1： argument 被看作是 DataType，不过它并不是由 DataType 基类派生的。

注 2： style 任务通常用于转换，请参见第七章。

例 4-1: <arg> 的使用

```
<?xml version="1.0"?>
<project name="arg demo" default="xslt" basedir=". ">

<property name="xalan.home" value="C:/java/xalan-j_2_1_0"/>
<property name="xalan.jar" value="${xalan.home}/bin/xalan.jar"/>
<property name="xerces.jar" value="${xalan.home}/bin/xerces.jar"/>

<property name="xmldata" value="familyTree.xml"/>
<property name="stylesheet" value="familyTree.xslt"/>
<property name="result" value="Family Tree.html"/>

<path id="project.class.path">
    <pathelement location="${xalan.jar}"/>
    <pathelement location="${xerces.jar}"/>
</path>

<target name="clean">
    <delete file="${result}" />
</target>

<target name="xslt">
    <echo message="Transforming '${xmldata}' using '${stylesheet}' "/>

    <java fork="true" classname="org.apache.xalan.xslt.Process"
          failonerror="true">
        <arg line="-IN"/>
        <arg value ="${xmldata}"/>
        <arg line="-XSL"/>
        <arg value ="${stylesheet}"/>
        <arg line="-OUT"/>
        <arg value ="${result}"/>
        <classpath refid="project.class.path"/>
    </java>

    <echo message="Success! See '${result}' for the output." />
</target>
</project>
```

本章后面将介绍此构建文件中其他有意思方面。现在，先来关注命令行参数。
如果从 shell 直接调用 Xalan，则命令行形如：

```
java org.apache.xalan.xslt.Process -IN familyTree.xml
-XSL familyTree.xslt -OUT "Family Tree.html"
```

你可以根据需要，自由地使用尽可能多的`<arg>`标签，这些参数将按其在构建文件中所列顺序传递给命令行。还可以在各`<arg>`标签中混合和匹配使用多个属性。你可能会奇怪，为什么我们没有一次指定所有参数，如下所示：

```
<arg line="-IN ${xmldata} -XSL ${stylesheet} -OUT ${result}" />
```

原因在于最后一个参数 "Family Tree.html"。在此例中，文件名包括一个空格。你应该记得，`line`属性允许多个参数以空格分隔，并且将 "Family Tree.html" 看作是两个参数 "Family" 和 "Tree.html"。由于我们希望将整个文件名作为一个参数，且要包括空格，所以必须使用`value`属性：

```
<arg value="${result}" />
```

由于我们将各个文件名定义为 Ant 特性，有些人可能会把 XML 和 XSLT 文件名调整为其他形式以供将来使用。因为这些名字中可能也包括空格，所以对所有这 3 个文件名参数均选用了`value`属性。对于 "-IN"、"-XSL" 和 "-OUT" 参数，我们还可以使用`line`属性，这是因为它们不会包括空格，不过在这种情况下，`value`属性也能得到同样的结果。

你可能还会奇怪，对于这个例子，为什么使用`value`属性而不是`path`。对于`value`属性文本将不做任何改动地传递给正在执行的进程。对于`path`属性，诸如 "familyTree.xml" 的文本在传递给进程之前将被转换为一个平台专有的路径，如 C:\path\to\file\familyTree.xml。对于需要绝对路径名的应用，则要求使用`path`属性。对于这个 Xalan 示例，无论使用`value`还是`path`都没有关系，因为它既可以用绝对路径名，也可以用相对路径名（注 3）。

附加示例

这一节将给出`argument DataType`的几个附加示例。`argument`允许几种变形；它们可以一起使用从而向进程传递多个参数。我们已经提到过，多个参数总是以构建文件中所列的顺序传递。以下为如何向一个进程传递两个不同的命令行参数：

注 3：从理论上讲，Xalan 需要 URL 而不是文件名作为参数。出于这个原因，较之于`value`特性可能得到的相对 URL，由`path`特性所产生的平台专有的文件名则不太适合。

```
<arg line="-mode verbose"/>
```

以下为如何传递一个包含有一个空格字符的命令行参数:

```
<arg value="Eric Burke"/>
```

最后, 是如何将一个路径形式的结构作为一个命令行参数传递:

```
<arg path="/temp;/tmp"/>
```

在 Windows 系统中, 这将转换为 *C:\temp;C:\tmp* (注 4), 在 Unix 系统中则转换为 */temp:/tmp*。

environment DataType

对于执行系统命令的 `apply` 和 `exec` 任务, 需要 0 个或多个嵌套 `<env>` 元素。这些元素指定了哪些环境变量要传递给正在执行的系统命令, 它们由 `org.apache.tools.ant.types.Environment.Variable` 类实现。`<env>` 元素接受以下属性:

file (all, File, *)

作为环境变量值的文件名。此文件名要被转换为一个绝对路径。

key (all, String, Y)

环境变量名。

path (all, Path, *)

作为环境变量的路径。Ant 会将它转换为一个本地约定, “path DataType”一节中将对此进行解释。例如, 在 Windows 平台上, *foo.txt* 将被转换为 *C:\path\to\file\foo.txt*。

value (all, String, *)

环境变量的一个直接量值。

file, path 或 value 只取其一。

注 4: 或者其他驱动器盘符, 这取决于你所在的基目录。

示例

以下示例调用了一个名为 *deploy.bat* 的批处理文件。在此批处理文件中，由于 `<env>` 元素的存在，TOMCAT_HOME 环境变量是可用的：

```
<property name="tomcat.home" value="/path/to/tomcat"/>

<target name="deploy">
    <!-- Call a deployment script, setting up the TOMCAT_HOME -->
    <!-- environment variable.                                     -->
    <exec executable="deploy.bat">
        <env key="TOMCAT_HOME" value="${tomcat.home}" />
    </exec>
</target>
```

在构建文件中使用环境变量

前一个例子显示了如何使用 `exec` 和 `env` 向系统命令传递环境变量。Ant 还允许在你自己的构建文件中使用环境变量。这是一种避免硬编码的绝佳方法，尽管这会限制可移植性。在构建文件中使用环境变量与 `environment DataType` 紧密相关。不过，`environment DataType` 并不用于从 Ant 中访问环境变量。实际做法是，环境变量的这种使用是作为 `property` 任务的一种特殊功能实现的，第七章将对此加以描述。

警告：JDK 1.1.x 应用可以使用 `System.getenv()` 方法访问环境变量。不过，JDK 1.2 不再支持 `System.getenv()`。它已经废弃，并会在调用时抛出一个 `Error`。Sun 公司决定不再使用此方法的原因在于，环境变量并非在由 Java 支持的所有平台上均可用。不过 Ant 的设计者对于支持读取环境变量提供了自己的实现，不过只限于某些平台上。在依赖于这个功能之前，请先在你感兴趣的平台上测试它。

作为一个例子，请考虑例 4-1 中所示构建文件的缺点。请看下一行：

```
<property name="xalan.home" value="C:/java/xalan-j_2_1_0"/>
```

尽管这在你的 PC 上可能可以工作，但在大多数其他开发人员的 PC 上则极有可能无法正常完成。原因在于他们可能把 Xalan 安装在了另一个目录上。如果你的构

建议文件要求开发人员在运行之前先设置 XALAN_HOME 环境变量，情况会好一些。

以下是为了做到这一点而对例 4-1 所做的一些修改：

```
<?xml version="1.0"?>
<project name="arg demo" default="xslt" basedir=".">
  <!-- Set up the 'env' prefix for environment variables -->
  <property environment="env"/>
  <property name="xalan.home" value="${env.XALAN_HOME}"/>
  <!-- Abort the build if XALAN_HOME is not set -->
  <target name="checkXalanHome" unless="env.XALAN_HOME">
    <fail message="XALAN_HOME must be set!"/>
  </target>

  <target name="xslt" depends="checkXalanHome">
    ...
  </target>

```

神奇之处在于下面这一行：

```
<property environment="env"/>
```

现在，通过在变量名前加上前缀“env:”，你就可以引用任何环境变量了。我们还可以增加另一个目标来验证环境变量是否得到设置。如果未设置，则会警告用户，并使构建失败。

```
<target name="checkXalanHome" unless="env,XALAN_HOME">
    <fail message="XALAN_HOME must be set!"/>
</target>
```

filelist DataType

filelist 是一个支持命名的文件列表的 **DataType**, 它由 **org.apache.tools.ant.types.FileList** 实现。包括在一个 **filelist** 中的文件不必确实存在。以下为其可接受的属性:

dir (1.4, File, *)	用于计算绝对文件名的目录。
files (1.4, String, *)	用逗号分隔的文件名列表。
refid (1.4, Reference, N)	对某处定义的一个<filelist>的引用。<filelist>用来定义一个文件列表。如果希望一次定义一组文件，然后再在构建文件中的多处对其引用，那么这就很有用。 注意：从 Ant 1.4 版本开始，使用 filelist Data Type 时，必须同时指定 dir 和 files 属性。也就是说，如果只指定了 dir 或者只指定了 files，那么将无法使用 filelist Data Type。所以，如果一次定义一组文件，然后再在构建文件中的多处对其引用，那么就必须同时指定 dir 和 files 属性。
示例	前面已经简要地提到了如何使用 filelist Data Type，但没有给出具体的例子。filelist Data Type 是在 Ant 1.4 中引入的，同时还引入了 dependset 任务（因为 filelist 只与 dependset 一同使用，所以我们必须对 dependset 任务加以讨论，才能解释 filelist Data Type）。dependset 任务将一个或多个输入文件与一个或多个输出文件加以比较。如果某些输入文件更新一些，则所有输出文件都被删除。另外，如果缺少某些输入文件，所有的输出文件也将被删除。令输出文件与一组可能已经不存在的输入文件相比较，这就是 filelist Data Type 之所以必要的原因。

下面来说明为什么结合使用 filelist Data Type 和 dependset 任务很有意义：在这个示例中，我们将一组 XML 和 XSLT 文件与一个 HTML 文件进行比较。如果缺少任何一个输入文件，或者存在任何一个更新的输入文件，那么 HTML 文件就会被删除。

```
<?xml version="1.0"?>
<project name="filelist demo" default="xslt" basedir=".">
    <!-- 定义两个 filelist，分别包含一组 XSLT 和 XML 文件 -->
    <filelist id="stylesheets" dir=".">
        <!-- 定义三个 XSLT 文件 -->
        <file="header.xslt" />
        <file="footer.xslt" />
        <file="body.xslt" />
    </filelist>
    <filelist id="xmlfiles" dir=".">
        <!-- 定义一个 XML 文件 -->
        <file="employees.xml" />
    </filelist>
    <!-- 定义一个 dependset 任务，将三个 XSLT 文件与一个 XML 文件进行比较 -->
    <dependset>
        <!-- 指定 XML 文件为输出文件 -->
        <output file="output.html" />
        <!-- 指定三个 XSLT 文件为输入文件 -->
        <input file="header.xslt" />
        <input file="body.xslt" />
        <input file="footer.xslt" />
    </dependset>

```

```

<target name="xslt">
    <!-- erase employeeDirectory.html if any of the XML files or
        XSLT stylesheets are newer -->
    <dependset>
        <srcfilelist refid="stylesheets"/>
        <srcfilelist refid="xmlfiles"/>
        <targetfilelist dir=". " files="employeeDirectory.html"/>
    </dependset>
    <echo message="Transforming Files..."/>
    ...
</target>
</project>

```

employeeDirectory.html 依赖于 4 个文件：*header.xslt*、*footer.xslt*、*body.xslt* 和 *employees.xml*。如果这些文件中的任何一个被修改了，*employeeDirectory.html* 都会被 *dependset* 任务删除。如果找不到某个输入文件，那么 *employeeDirectory.html* 也会被删除。

我们定义了两个 *filelist*，一个用于 XSLT 文件，另一个用于 XML 文件。也可以只是简单地定义一个 *filelist* 来包含所有文件，不过如果文件按类型进行了逻辑分组，构建文件就可能更容易理解。在 *dependset* 任务中我们同时引用了这两个 *filelist*:

```

<dependset>
    <srcfilelist refid="stylesheets"/>
    <srcfilelist refid="xmlfiles"/>
    <targetfilelist dir=". " files="employeeDirectory.html"/>
</dependset>

```

<srcfilelist> 标签使用 *refid* 属性指回到构建文件中前面定义的 *filelist*。*<targetfilelist>* 标签显示了另一种可选语法，即允许 *filelist* 得到内联定义。如果打算在一个构建文件中多次引用一个 *filelist*，就应该考虑 *refid* 方法。否则内联地定义 *filelist* 可能更简单。

注意：尽管我们在讨论 *filelist DataType*，但 XML 标签却称为 *<srcfilelist>* 和 *<targetfilelist>*。XML 标签名经常与 *DataType* 名并不匹配。

fileset DataType

fileset DataType 定义了一组文件，并通常表示为 <fileset> 元素。不过，许多 Ant 任务构成了隐式的 (implicit) fileset，这说明它们支持所有 fileset 属性和嵌套元素。与 filelist 类型不同，由 fileset 表示的文件必须存在。fileset 还可能指定为目标级构建文件元素（即 <project> 的子元素），并由其 id 引用。以下为 fileset 属性列表：

dir (all, Path, Y)

fileset 的基目录。

casesensitive (1.4.1, boolean, N)

如果置为 false，那么在匹配文件名时，fileset 不是区分大小写的。其默认值为 true。Ant 1.4.1 以前的版本使用区分大小写的匹配。

defaultexcludes (all, boolean, N)

确定是否使用默认的排除模式。默认为 true。默认的排除模式包括 */*~, **/#*#、 */.##*、 */%*%、 */CVS、 */CVS/**、 */.cvignore、 */SCCS、 */SCCS/** 和 */vssver.scc。

excludes (all, String, N)

用逗号分隔的需要排除的文件模式列表。这是对默认排除模式的补充。

excludesfile (all, File, N)

每行包括一个排除模式的文件的文件名。这是对默认排除模式的补充。

includes (all, String, N)

用逗号分隔的需要包含的文件模式列表。

includesfile (all, File, N)

每行包括一个包含模式的文件的文件名。

除了以上列出的属性外，fileset 还可能包括以下元素：

0 到 n 个嵌套 `<patternset>` 元素: `<exclude>`、`<include>`、`<patternset>` (*all*);
`<excludesfile>`, `<includesfile>`, (1.4)

它们定义了 `fileset` 中包含和/或排除哪些文件。所有这些都将在“`patternset DataType`”一节中做简要描述。除了 `<patternset>` 外, 要使用这些嵌套元素, 而不是其对应的属性。

示例

以下示例将产生同样的结果。由于 `fileset` 相当依赖于 `patternset`, 所以在学习了这些示例后应当继续阅读“`patternset DataType`”一节。第一个示例使用了 `includes` 和 `excludes` 属性来选择 `src` 目录中的所有 `.java` 文件, 而排除目录 `test` 之下所有目录中的 `.java` 文件:

```
<fileset id="sources1" dir="src"
    includes="**/*.java"
    excludes="**/test/**/*.java">
</fileset>
```

下一个示例使用嵌套的 `<include>` 和 `<exclude>` 标签, 取代了 `includes` 和 `excludes` 属性:

```
<fileset id="sources2" dir="src">
    <include name="**/*.java"/>
    <exclude name="**/test/**/*.java"/>
</fileset>
```

通过使用 `<include>` 或 `<exclude>` 元素, 你可以基于特性有选择地包含或排除文件。例如, 你可以使用以下语法有选择地包含文件, 对此将在“`patternset DataType`”一节中简要描述:

```
<!-- Skip unit tests unless the includeTests property is set -->
<exclude name="**/test/**/*.java" unless="includeTests"/>
```

你也可以使用嵌套的 `<patternset>` 达到同样的目的:

```
<fileset id="sources3" dir="src">
    <patternset>
        <include name="**/*.java"/>
```

```
<exclude name="**/test/**/*.java"/>
</patternset>
</fileset>
```

最后，我们在某处定义了一个`<patternset>`，并在两个位置对它做了引用。比起前面的示例来说，这更为有用，因为这样允许你在整个构建文件中重用一个常用的`patternset`：

```
<patternset id="non.test.source">
    <include name="**/*.java"/>
    <exclude name="**/test/**/*.java"/>
</patternset>

<!-- later in the same buildfile -->
<fileset id="sources4" dir="src">
    <patternset refid="non.test.source"/>
</fileset>
<fileset id="sources5" dir="othersrc">
    <patternset refid="non.test.source"/>
</fileset>
```

包含和排除模式语法

Ant 使用模式来包含和排除文件。例如`**/*.java`与任何子目录中的所有`.java`文件相匹配。其语法很简单：

* 与 0 个或多个字符相匹配。`*.java` 就可以与 `Account.java` 和 `Person.java` 匹配，但不匹配 `settings.properties`。

? 与 1 个字符相匹配。`File?.java` 可与 `FileA.java` 和 `FileB.java` 匹配，但不能与 `FileTest.java` 匹配。

** 与 0 个或多个目录相匹配。`/xml/**` 与 `/xml/` 下的所有目录和文件都能匹配。

允许结合使用模式。例如，一种更为复杂的模式 `com/oreilly/**/*Test.java` 就可与以下三个文件匹配：

`com/oreilly/antbook/AccountTest.java`
`com/oreilly/antbook/util/UnitTest.java`
`com/oreilly/AllTest.java`

patternset DataType

`fileset` 是对文件的分组，而 `patternset` 是对模式的分组。它们是紧密相关的概念，因为 `fileset` 依赖于选择文件的模式。`<patternset>` 元素可能作为目标级构建文件元素出现（即作为 `<project>` 的子元素），而且后面将通过其 `id` 被引用。正如前面的例子所示，它也可能作为 `<fileset>` 的一个嵌套元素出现。作为隐式 `fileset` 的任务也支持嵌套 `<patternset>` 元素。

`<patternset>` 元素支持 4 个属性：`includes`、`excludes`、`includesfile` 和 `excludesfile`。它们在前面有关 `fileset` 的一节中已经描述过。除了这些属性外，`patternset` 还允许以下嵌套元素：

0 到 n 个嵌套 `<include>` 和 `<exclude>` 元素

它们支持以下属性：

`name (all, String, Y)`

包含或排除的模式。

`if (all, String, N)`

特性名。只在设置了该特性时 Ant 才使用此模式。

`unless (all, String, N)`

特性名。只在未设置该特性时 Ant 才使用此模式。

0 到 n 个嵌套 `<includesfile>` 和 `<excludesfile>` 元素

它们支持以下属性：

`name (all, String, Y)`

包括有包含和排除模式的文件名，其中每行有一个模式。

`if (all, String, N)`

特性名。只在设置了该特性时 Ant 才读取此文件。

`unless (all, String, N)`

特性名。只在未设置该特性时 Ant 才读取此文件。

示例

现在我们给出 `patternset` DataType 的两种用法。第一个示例所显示的 `patternset` 用于将一组相关文件由一个目录复制到另一个目录。第二个例子显示的 `patternset` 则用于有条件地将文件包含到编译中。

复制文件

以下为如何建立一个 `patternset` 来表示某个目录树中所有与 XML 相关的文件名：

```
<patternset id="xml.files">
    <include name="**/*.dtd, **/*.xml, **/*.xslt"/>
</patternset>
```

下面，我们可以使用 `copy` 任务来将这些文件由一个源目录复制到一个目标目录：

```
<copy todir="${deploy.dir}">
    <!-- select the files to copy -->
    <fileset dir="${src.dir}">
        <patternset refid="${xml.files}" />
    </fileset>
</copy>
```

有条件地包含文件

在此第二个示例中，我们将排除所有单元测试，除非设置了 `includetests` 特性：

```
<?xml version="1.0"?>
<project name="patternset_test_project" default="compile" basedir=". ">

    <!-- exclude tests unless the 'includetests' property is set -->
    <patternset id="sources">
        <include name="**/*.java"/>
        <exclude name="**/*Test.java" unless="includetests"/>
    </patternset>

    ...remainder of buildfile omitted

    <target name="compile" depends="prepare">
        <javac destdir="build">
```

```

<!-- the directory from which the patternset finds files to compile -->
<src path="src"/>

<!-- refer to the patternset which selects the source files -->
<patternset refid="sources"/>
</javac>
</target>

</project>

```

现在，为了将单元测试加入到构建中，可以在从命令行调用 Ant 时设置 `includetests` 特性：

```
$ ant -Dincludetests=true compile
```

filterset DataType

`filterset DataType` 于 Ant 1.4 中引入，并允许定义一组过滤器。这些过滤器（由 `filter` 任务实现）将在文件移动或复制时完成文件中的文本替换。这称为记号过滤（token filtering）。若在输入文件中发现了某些记号则会出现此文本替换。当文件移动或复制时，这些记号被替换为匹配过滤器中所定义的文本。在 Ant 1.4 版本之前，`filter` 任务总是将 @ 字符用作记号分隔符。`filterset` 则允许定制开始和结束记号分隔符。

`filterset DataType` 由 `<filterset>` 元素表示。`<filterset>` 元素可能作为 `copy` 和 `move` 任务中嵌套的内容出现，或者作为目标级构建文件元素出现（即 `<project>` 的子元素）。以下为允许的 `filterset` 属性：

`begintoken (1.4, String, N)`

对于嵌套过滤器所搜索的记号，这是标识其开始的字符串。默认为 @。

`endtoken (1.4, String, N)`

对于嵌套过滤器所搜索的记号，这是标识其结束的字符串。默认为 @。

`id (1.4, String, N)`

对此过滤器的惟一标识符。当过滤器定义为一个目标级构建文件元素，并且需要在以后被引用时，这就是必要的。

refid (1.4, Reference, N)

对构建文件中某处定义的一个过滤器的引用。

filterset 还可以包括以下元素：

0 到 n 个嵌套 <filter> 元素 (1.4)

每个嵌套 **<filter>** 元素定义了一个记号及替换文本。**<filter>** 需要以下属性：

token (1.4, String, Y)

指定要替换的记号，不包括定界符。如果此过滤器要替换 @VERSION@，则将 VERSION 作为此属性值。

value (1.4, String, Y)

指定遇到记号时的替换文本。

0 到 n 个嵌套 <filtersfile> 元素 (1.4)

每个元素指定一个 Java 特性文件，由此可加载另外的过滤器。此文件中每一行包括一个记号，其后为一个冒号 (:)，再后面是一个值。**<filtersfile>** 需要以下属性：

file (1.4, File, Y)

包括过滤器的特性文件的文件名。

示例

此示例目标显示了如何将 %COPYRIGHT! 和 BUILD_DATE! 记号替换为所复制的文件：

```
<target name="tokenFilterDemo" depends="prepare">
    <!-- set up the timestamp -->
    <tstamp>
        <format property="now" pattern="MMMM d yyyy hh:mm aa" />
    </tstamp>
    <copy todir="build" filtering="true">
        <fileset dir="src">
            <include name="**/*.java" />
        </fileset>
```

```
<!-- search for %COPYRIGHT! and %BUILD_DATE! -->
<filterset begintoken "%" endtoken="!">
    <filter token="BUILD_DATE" value="${now}"/>
    <filter token="COPYRIGHT" value="Copyright (C) 2002 O'Reilly"/>
</filterset>
</copy>
</target>
```

注意 filtering="true" 必须在 copy 任务中设置, 以保证发生记号过滤。我们的 filterset 由两个不同的过滤器组成, 而且在此显式地指定了 begintoken 和 endtoken, 因为我们不想使用默认的 @ 字符。

以下为复制前的源文件:

```
// %COPYRIGHT!
// Built on %BUILD_DATE!

public class Whatever {
    ...
}
```

以下为复制操作完成后的目标文件:

```
// Copyright (C) 2002 O'Reilly
// Built on March 12 2002 03:10 PM

public class Whatever {
    ...
}
```

在每个源文件中记号可能出现多次, 它们全都会被替换。对此还有另一个示例, 请参见第七章中的 filter 任务。

path DataType

path DataType 出现很频繁, 而且有时被称为路径形式的 (path-like) 结构。它可能用作为一个属性或一个嵌套元素。path DataType 最常用于表示一个类路径, 不过也可用于表示其他用途的路径。在用作为一个属性时, 路径中的各项用分号

(;) 或冒号(:) 字符隔开，在构建时，此分隔符将代之以当前平台所用的路径分隔符。

注意：与其他DataType类似，path DataType并不总是由<path> XML元素表示。例如，javac任务就接受由path DataType实现的嵌套<classpath>元素。

path DataType用作为一个XML元素时，较之于用作一个属性能够提供更大的灵活性。以下为一组path属性：

location (all, File, *)

表示一个文件或目录。Ant在内部将此扩展为一个绝对文件名（注5）。

path (all, String, *)

一个文件和路径名列表，并以；或:分隔。

refid (all, Reference, *)

对当前构建文件中某处定义的一个path的引用。如果想在构建文件多处引用同一个路径定义，这将是有用的。

location和path均是可选的，除非指定了refid（在这种情况下，location和path都不允许使用）。指定了refid时就不能再有嵌套元素。

path DataType还支持以下嵌套元素：

0到n个嵌套<pathelement>元素（注6）

定义一个或多个要包含在path中的文件。每个嵌套的<pathelement>就像包含它的path DataType一样，还支持location和path属性。

0到n个嵌套<fileset>元素

提供将文件包含在path中的另一种语法。

注5：Ant负责处理路径转换的具体细节，可将路径转换为与所运行操作系统兼容的形式。

注6：<pathelement>由PathElement实现，这是org.apache.tools.ant.types.Path中的一个嵌套类。它是一个辅助类而不是DataType。

0 到 n 个嵌套 <path> 元素

将路径递归地嵌套在其他路径中。

对于一个由两个JAR文件和两个目录所组成的路径，以下即为如何用一个路径形式的结构来表示此路径。这个路径按构建文件中所列的顺序构建：

```
<path>
  <pathelement location="${libdir}/servlet.jar"/>
  <pathelement location="${libdir}/logging.jar"/>
  <pathelement path="${builddir}"/>
  <pathelement path="${utilpath}"/>
</path>
```

`path DataType` 还支持一种简写语法。例如，假设我们在一个任务中用 `<classpath>` 元素来定义一个路径：

```
<!-- The classpath element is implemented with the path DataType -->
<classpath>
  <pathelement path="${builddir}"/>
</classpath>
```

它可以简写如下：

```
<classpath path="${builddir}"/>
```

`location` 属性的工作是类似的。作为最后一种情况，在路径形式的结构中可以嵌套一个或多个 `fileset`：

```
<classpath>
  <pathelement path="${builddir}"/>
  <fileset dir="${libdir}" includes="**/*.jar"/>
</classpath>
```

在此例中，`fileset` 包括了由 `${libdir}` 所指定目录下任何目录中的所有 `.jar` 文件。

mapper DataType

在本章的最后，我们将对 `mapper` 加以讨论，这是在 Ant 1.3 中加入的功能。

`mapper` 定义了一组源文件与一组目标文件如何相关。`<mapper>`（注 7）元素支持以下这些属性：

`classname (1.3, 1.4, String, *)`

实现 `mapper` 的类的类名。当内置 `mapper` 不足以满足要求时，用于创建定制的 `mapper`。

`classpath (1.3, 1.4, Path, N)`

查找一个定制 `mapper` 时所用的类路径。

`classpathref (1.3, 1.4, Reference, N)`

对某处定义的一个类路径的引用。

`from (1.3, 1.4, String, *)`

此属性的含义取决于所用的 `mapper`。后面的例子将展示在何处使用此属性。

`refid (1.3, 1.4, Reference, N)`

对另一个 `mapper` 的引用。如果指定了它，这应当是惟一所列的属性（即不应再指定其他属性）。这就允许将一个 `mapper` 定义一次，而可以在构建文件中的多处用到。后面的例子将展示在何处使用此属性。

`to (1.3, 1.4, String, *)`

此属性的含义取决于所用的 `mapper`。

`type (1.3, 1.4, Enum, *)`

取值为 `identity`、`flatten`、`glob`、`merge` 或 `regexp` 其中之一。它定义了要使用的内置 `mapper` 的类型。

只需一个 `type` 或 `classname` 属性。`from` 和 `to` 属性可能是必要的，这要取决于 `mapper`。

注 7： 在 Ant 1.4.1 中，`mapper` `DataType` 总是由一个 `<mapper>` XML 元素实现。其他 `DataType` 则无此一致性。

示例

在讨论各种特定类型的 mapper 之前，先来看一个简单的示例。例 4-2 给出了一个构建文件，它将对所有.java 文件进行备份复制，并在各文件名后追加一个.bak 扩展名。

例 4-2：用一个 glob mapper 备份文件

```
<?xml version="1.0"?>
<project name="mapper demo" default="backupFiles" basedir=". " >

    <!-- define a mapper for backing up files -->
    <mapper id="backupMapper" type="glob" from="*.java" to="*.java.bak"/>

    <target name="clean">
        <delete dir="bak" />
    </target>

    <target name="prepare">
        <mkdir dir="bak" />
    </target>

    <target name="backupFiles" depends="prepare">
        <copy todir="bak">
            <!-- select the files to copy with a fileset -->
            <fileset dir="src" includes="**/*.java"/>
            <!-- mapper refid="backupMapper" />
        </copy>
    </target>
</project>
```

这个示例还展示了 fileset DataType 的另一种用法，即由 copy 任务用来选择哪些文件要进行复制。copy 任务具体完成文件的复制。嵌套的 fileset 定义了要复制的一组文件。嵌套 mapper 引用构建文件前面所定义的 mapper，还指定文件在复制时如何重命名。当文件被复制时，它们会按照 mapper 所指定的模式被重命名。

这个示例使用了一种称为 *glob mapper* 的 mapper，它会将一种简单的通配模式应用于一组输入文件名，并基于此生成一组目标文件名。有一些可用的 mapper 类型。下面逐一介绍。

identity mapper

将源文件与同名的目标文件进行匹配。这是 copy 任务所使用的默认 mapper，因此很少需要定义自己的 identity mapper。表 4-2 列出了以下 identity mapper 的结果：

```
<mapper type="identity"/>
```

表 4-2: identity mapper 的结果

源文件	目标文件
<i>Customer.java</i>	<i>Customer.java</i>
<i>com/oreilly/data/Account.java</i>	<i>com/oreilly/data/Account.java</i>

flatten mapper

flatten mapper 从文件名中删除所有路径信息。如果希望从多个不同目录中将一组文件复制到一个目标目录中，这就很有用。表 4-3 列出了以下 flatten mapper 的结果：

```
<mapper type="flatten"/>
```

表 4-3: flatten mapper 的结果

源文件	目标文件
<i>Customer.java</i>	<i>Customer.java</i>
<i>com/oreilly/data/Account.java</i>	<i>Account.java</i>

glob mapper

glob mapper 基于简单的通配模式确定目标文件名。若要对已经有一致文件名（如以 *Test.java* 结尾）的一组文件重命名，这就很有用。*to* 和 *from* 定义了模式，其中至多有一个 * 字符。当一个源文件名与 *from* 模式匹配时，即生成目标文件名。*to* 属性的 * 要替换为 *from* 属性的 * 所对应的匹配文本。表 4-4 列出了以下 *glob mapper* 的结果：

```
<mapper type="glob" from="*Test.java" to="*UnitTest.java">
```

表 4-4: glob mapper 的结果

源文件	目标文件
<i>Customer.java</i>	无
<i>com/oreilly/data/Account.java</i>	无
<i>CustomerTest.java</i>	<i>CustomerUnitTest.java</i>
<i>com/oreilly/tests/CustomerTest.java</i>	<i>com/oreilly/tests/CustomerUnitTest.java</i>

表 4-4 前两行中的“无”文本说明，在使用一个 glob mapper 的复制操作中，未得到匹配的文件仅仅是不进行复制。

merge mapper

merge mapper 将所有源文件名与 *to* 属性所指定的相同目标文件相匹配。*from* 属性被忽略。如果想比较一组源文件和某个目标文件的时间戳，*merge mapper* 就很有用。这正是 *uptodate* 任务的工作，如第七章所述。表 4-5 列出了以下 *merge mapper* 的结果：

```
<mapper type="merge" to="oreilly.zip">
```

表 4-5: merge mapper 的结果

源文件	目标文件
<i>Customer.java</i>	<i>oreilly.zip</i>
<i>com/oreilly/data/Account.java</i>	<i>oreilly.zip</i>

regexp mapper

regexp mapper 类似于 *glob mapper*，不过使用的是正则表达式而不是简单的 * 字符。这些正则表达式的具体语法完全取决于所用的底层正则表达式库。稍后将简要介绍 Ant 用于选择此库的机制。

实现org.apache.tools.ant.util.regex.RegexpMatcher接口的类必须由此库提供，而不论你选择使用哪一个表达式库来支持regexp mapper。Ant包括了实现以下库的类：

JDK 1.4

包括在J2SE 1.4中，可在<http://java.sun.com>获得。

jakarta-regexp

可在<http://jakarta.apache.org/regexp/>获得。

jakarta-ORO

可在<http://jakarta.apache.org/oro/>获得。

为了确定要使用哪个库，Ant首先查看ant.regex.matcherimpl系统特性。如果它指定了一个实现RegexpMatcher接口的类，则使用该库。否则，它将对类路径进行搜索，并按所列顺序（从JDK 1.4开始）找到一个合适的库。如果未找到任何库，则任务失败。

第五章

用户编写任务

可以通过定制来扩展 Ant，这个概念一直是（而且仍将是）它最重要也最值得称道的功能。Ant 的创造者为我们提供了一个足够健壮的系统，从而可以与目前可用的许多语言和工具一同使用，另外还能够继续扩充并可与将来的语言和工具协作。例如，目前 Ant 中存在有处理 C# 语言的任务，而在 2000 年 Ant 首次出现时则无此功能。用户已经编写了处理第三方工具的大量任务，从群件产品（如 StarTeam，这是一种版本控制系统）到应用服务器（如 BEA 的 WebLogic 或 JBoss Group 的 JBoss）都在此范围内。这些调整和改进对于 Ant 的核心处理引擎仅有很小的改动（或根本不用修改）。扩展 Ant 而不用调整其核心引擎，这一点非常重要，因为这意味着核心 Ant 引擎的改进和调整可以与扩展开发相分离。这两个领域的开发可以同时进行，这样就能加快对 Ant 的修改，倘若 Ant 是一个集成系统，则绝不会有如此快的修改速度。

所有 Ant 任务都是 Java 类，而且任何程序员都可以通过编写一个新的 Java 任务类来扩展 Ant 的功能。存在一些用户编写的（user-written）任务，它们所使用的 Ant 接口与随 Ant 一起发布的核心任务所用的完全一样。用户编写任务和核心任务之间的差别仅在于任务的创造者和任务所在包有所不同（有时甚至连这两点也是相同的）。除此之外，它们均在同一级别上发挥作用。在这一章中，我们将展示如何编写自己的任务并以此来扩展 Ant。

定制任务的需要

Ant有两个任务: `java`和`exec`, 它们能够在某个系统中执行任何Java类或命令行可执行程序。你也许会感到奇怪, 既然有如此强大的功能, 为什么还需要定制任务呢? 从理论上讲, 你可以使用这些定制任务来处理任何类或运行任何程序。可以证实, 有些定制任务确实只不过就是一个执行包装器, 它们运行Java类或程序的方式与`java`或`exec`任务完全相同。区别仅仅是定制任务更紧密地使用了Ant引擎。定制任务可以提供更详细的消息, 而且可以更准确地处理错误。另一方面,`java`和`exec`任务在处理未预见的错误和向用户提供详细的声明方面功能是有限的。无论事件或错误的本质如何, 对于这两个任务来说, 它们都是一样的, 这样就使你几乎无从控制。

在大多数情况下, 对于扩展Ant功能这个问题, 较之于使用`java`或`exec`任务来说, 定制任务是更好的解决方案。构建错误、事件和消息均由任务发起, 并由Ant引擎管理。Ant对这些事件做出响应, 并以一种受控的方式加以处理, 将它们传递给其自己的监听者, 或者是传递给其他用户编写的监听者(有关用户编写监听者的更多内容请参见第六章)。对于终端用户来说(即某些软件开发人员, 他们需要有关其工程构建过程的更适当的信息), 这种细粒度的任务管理更为合适。另外, 这对于编写定制任务来扩展原有任务的其他开发人员而言, 也是更合适的做法, 这样就可以继承这些任务的功能, 并在一系列相关操作之间建立起一致的行为。单单是这些功能就已经使定制任务看上去很不错了, 不过, 使用定制任务还有其他的优点。

任务很擅长于对简单操作进行抽象, 并可利用一个一致的接口和额外的功能使之更为强大。对于某些常用的跨平台shell功能, 有些Ant任务甚至还能够处理它们之间的非一致性。例如, 复制和删除跨平台的文件和目录是一件很令人头疼的事, 因为命令名和参数在不同shell和不同操作系统之间会存在差异。由于在Ant中能够用任务来抽象文件操作, 这就解决了这个难题, 并为其用户提供了一致的接口。在Ant中, 只有一种复制或删除文件的方法, 而且无论Ant在哪个平台上运行它都将正常工作。这并不是抽象所提供的惟一好处。如果命令行工具没有对功能集加以限制, 那么得到抽象的任务还可以增加可用的功能集。Windows的`del`命令

不能删除所有以 `.java` 结尾的文件，而且对所有以 `Abstract` 开头的文件也不做处理。Ant 的任务 `delete` 则可完成此工作，较之于其命令行方式来说，它表现出了更大的灵活性。更值得一提的是，它可以在任何平台上做此操作。任务设计关注的是构建的需要，而不会将自己局限于工具的功能之上，而工具的设计所关注的是 shell 和操作系统的需要。

由于定制任务类广泛可用，因此你几乎可以对任何工具加以改进。不要把定制任务看成是修补 Ant 缺陷的邦迪创可贴。Ant 及其任务模型更像是垒高拼装玩具。增加任务可以增添并改善 Ant 的功能集，同时不会增加 Ant 的规模。Ant 总可保持其模块性和可扩展性。

Ant 的任务模型

要理解定制任务意味着需要理解任务模型。Ant 作为一个基于 Java 的程序，使用了 Java 的类层次和反射功能来完成其工作。所有 Ant 任务都直接或间接地派生自抽象类 `org.apache.tools.ant.Task`。Ant 引擎在这一级别上管理所有的任务对象，但只对 `Task` 对象进行处理。对于引擎，每个任务都派生自相同的类，而且有与各个其他任务相同的核心方法和特性。XML 解析和方法命名方案的结合使得 Ant 可以使用 `Task` 的所有子类。另外，Ant 以一种固定的方式处理任务，也就是说，Ant 循环地对每个任务进行处理。若要编写简单的任务，那么并没有必要详细了解这个模型和过程，而对于复杂的任务，除非你理解了整个任务模型和执行过程，否则它就会表现出我们所不希望的行为。

编写定制 DataType

除了任务以外，Ant 模型还可以处理 `DataType`。`DataType` 的一个例子就是 `path` 任务。此 `path` 任务不进行直接操作。相反，它会基于 XML 中给定的规则和其他信息来创建一个数据集。对于 Ant 1.4，从理论上讲，用户可以编写其自己的 `DataType`。不过，用来声明 `DataType` 的方法 (`typedef task`) 还存在 bug，尚不能使用。希望 1.5 版本能够提供修正。

任务的组成部分

任务有两个方面。对于一个Ant终端用户来说，任务无非就是构建文件中的XML。你可以剖析此 XML 并从这个角度确定任务的各个部分。不过，对于编写任务的程序员来说，任务则有所不同。尽管 XML 仍然存在，但它只是相当于 Java 代码的一个指南。Java 代码只是冰山的一角。从理论上讲，对于任务还存在许多其他方面。

通用子类

对于所有任务类，都需要由一个超类派生（从某种意义上讲，要从 Task 派生）。Ant 引擎仅对 Task 对象进行操作，而对于开发人员对 Task 类的子类所做的补充则视而不见。不过，这并不意味着你要忽略 Task 类层次。要知道，理解其层次所能提供的帮助与忽视它所能带来的障碍同样多。Task 的子类不仅表示构建文件的任务，它们还表示包括有一些功能的类，而这些功能对于其他任务是有用的。有时，一个子类甚至不是任务。例如，如果你的任务需要使用文件集和模式，就应当扩展 `org.apache.tools.ant.main.taskdef.MatchingTask`。这个类实现了许多这样的文件集和模式操作，从而可以减轻你自行实现它们的负担。有了诸如这个类以及其他一些任务类，对你来说，能够站在如此强大的“巨人”的肩膀上，这是很有好处的。

你应当了解那些设计得与你的需求接近的任务。Ant 中高效重用的一个很好的例子就是 zip 系列的任务。由于 JAR 扩展了 zip 打包模型，所以 jar 任务由 zip 派生，并借用了它的大部分功能，而且只实现了 JAR 专有的操作。更进一步，WAR (Web ARchive) 是一个带有标准目录结构的 JAR，并且还有一个额外且必需的文件，即部署描述文件 `web.xml`。因此，war 任务派生自 jar。对于 war，创建标准目录结构以及验证描述文件的实现都放在 War 任务类中，其他的功能则通过继承得到。在本章后面，我们将分析 jar 任务及其层次，以此作为一个定制任务的例子。

属性

属性 (attribute) 是描述一个特定 XML 标签的名 - 值对。从程序的角度讲，Ant

从 XML 解析并加载属性的名 - 值对，并将它们传递给各个任务对象。Ant 对字符串值进行重定义，使之变成基本类型、File 对象甚至是 Class 对象。通常，属性值表示 boolean 基本类型，相当于任务的处理标志。例如，javac 的 debug 属性就是一个 boolean 类型。若此标志打开，javac 就会编译带有调试信息的类。若此标志关闭，javac 就会正常地编译类。

嵌套元素

或多或少地，嵌套元素与属性总是互斥的选择。它们可以是任务或 DataType。与处理属性一样，任务会显式地处理其嵌套元素。遗憾的是，处理嵌套元素并不像处理名 - 值对那样简单而直接。

嵌套元素的复杂性尚存在疑问，因为关于嵌套元素的使用还没有一个可供设计使用的权威模型。从理论上说，你的定制任务可以将任何任务用作嵌套元素。例如，可以将 javac 作为一个嵌套元素。不过，在你显式地对 javac 的相应类 Javac 的使用进行处理之前，这样一个嵌套元素不会工作。你必须要注意并处理 javac 实现的所有特殊问题，而这绝非易事。即使你确实这样做了，javac 仍有可能会完成某些操作而使你不能将它作为一个嵌套元素。这是因为没有标准的方法来实现任务。既然不能通过程序的方式阻止你将 javac 之类任务用作嵌套元素，那么就只有在构建失败时你才会发现它并不能如期工作。

任务使用自省 (introspection) 调用来处理嵌套元素，就如同处理属性一样。区别仅在于一个嵌套元素的对应类本身包括所有的数据和功能，而属性只是名 - 值对。一个元素有如下需求：其类要被实例化，其属性要被解析和处理，另外其函数要被执行。在此过程中，任何时刻都可能发生错误。

通过比较一个任务对属性的使用和对嵌套元素的使用，可以更好地描述属性和嵌套元素的差别：

```
<copy destdir="newdir/subdir">
    <fileset dir="olddir">
        <include name="**/*.java"/>
    </fileset>
</copy>
```

copy任务带有属性 `destdir` 和嵌套元素 `<fileset>`。copy任务对 `destdir` 的处理很简单。Ant 向这个任务的类传递一个对于目录的 `File` 对象。只需通过一个调用，此属性即得到设置。再来将它与 Ant 如何处理 `<fileset>` 元素加以比较。Ant 有 3 种方法可以向任务的类传递 `Fileset` 对象。无论哪一种情况，Ant 都必须在同一个生命周期中将 `fileset` `DataType` 作为一个任务（因为在这一级别上，对于 Ant 引擎来说，任务和 `DataType` 是一样的）。Ant 对于这些任务和 `DataType` 的处理是一个递归的过程。我们想说明的一点是，Ant 对 `DataType` 的处理过程比起处理一个元素的属性来说要棘手得多。

较之于 `DataType`，尽管属性的使用和理解要容易一些，但它们在可读性和灵活性方面则稍逊一筹。例如，路径就会导致难看且难于维护的属性。路径值可能会很长，而且每次修改路径结构时，都必须对它做相应的修改。嵌套的路径元素则更适合阅读，而且也较易于维护。从它们如何表示路径这个方面来说，当然也更为强大，因为嵌套路径元素可以使用复杂的文件模式（如 `*.*` 对于 `path` `DataType` 是可用的，但不能作为一个 `path` 属性）。

像平常的所有情况一样，要决定是实现任务的属性还是实现其嵌套元素，往往需要进行权衡。尽管在使用 `DataType` 时可以得到可维护性和可读性，但与使用属性相比，在开始的开发时间上则进行损失。使用嵌套元素有多种方法（确切地说，是 3 个方法调用），而每一种都容易出现错误或异常行为，而且它们可能很难进行调试。出于这个原因，一些编写任务的人对这两种方法都提供支持，例如既有一个 `classpath` 属性又有一个 `classpath` 嵌套 `DataType`。

要记住，这对于用户来说可能是一个容易产生混乱的解决方案，因此要相应地对你的任务建立文档。对于以下情况你需要显式地给出定义，即如果一个用户既指定了一个属性，又指定了一个表示相同数据的嵌套元素，那么将会发生什么情况。Ant 并不知道如何确定其差别，因此会试图对它们都加以处理，这样就会产生不确定的后果。

Ant 与任务之间的通信

对任务的各个组成部分有了一定的认识之后，下面可以把注意力转向 Ant 构建引

擎与任务进行通信时所采用的机制了。在编写定制任务时，你需要理解3种通信机制：

Project 类

Project类作为一个公共实例变量，在每个任务中都可用。这个类表示整个构建文件以及其中包含的所有内容，可以使你访问所有任务、目标、特性和其他构建文件部分。

构建异常

构建异常通过BuildException类实现，可以为任务提供一种向Ant构建引擎通知错误情况的机制。

日志系统

日志系统可通过Project类访问，为任务提供一种向用户显示进展信息的方法。

以下三个小节将对这些机制做详细描述。

Project 类

有一个类可帮助完成任务和Ant引擎之间的大多数通信，这就是Project类。其父类Task（注1）中包含有此实例变量，这就使得这种通信成为可能。可以像在任何任务中使用任何实例变量一样地使用它。许多强大的功能都源于Project类，因此要特别注意它能做什么，还要小心可能偶尔会滥用此能力的情况（你当然不会故意滥用其能力吧！）。而且要记住，关于Project，你所做的一些明智之举可能会随着下一版本的Ant的出现而变得不合时宜。要有一个备份设计计划，否则就要准备维护你自己的Ant版本。

Project类表示整个构建文件。此类可以保证对构建文件的每个任务、目标和特性进行访问，甚至还可以访问某些定义构建文件应当如何执行的核心设置。开发

注1：从Ant 1.4起，核心组件变成了ProjectComponent，而不再是Task。Project对象目前是ProjectComponent类的一个保护型实例变量。

人员很少做此访问，不过确实存在有这种功能和能力。从根本上说，任务开发人员可通过`log`方法调用来使用`Project`，从而提供对引擎的核心审计系统的访问。

另外，`Project`为所有任务定义了系统范围的常量和全局方法。常量用于系统调用参数，例如用于日志记录。全局方法则提供了大量功能，包括从将路径翻译为本地形式到为具有`boolean`值的任务属性提供一个`boolean`解释器等等。

在一个任务中，`Project`类的字段名为`project`(相当合适)。以下是对`project`可用的一些常用方法调用和常量：

`project.getGlobalFilterSet()`

返回一个`FilterSet`对象(对于构建来说是全局的)。可以定义一个全局的过滤器集，使每个完成文件或目录操作的任务排除或包含一组文件。如果你的任务需要遵循此全局过滤器，可以利用对`project.getGlobalFilterSet()`的调用来得到。有关`FilterSet`的更多信息请参见 Ant API JavaDoc。

`project.getBaseDir()`

返回`<project>`元素的`basedir`属性。如果你的任务需要在工程目录中(或由工程目录)完成文件操作，那么这是得到该目录路径的最好方法。

`project.translatePath()`

为当前所用的操作系统将一个路径解释为本地格式。构建文件的作者可以用一种通用的方式来写路径和文件名，而忽略像目录分隔符这样的差别。如果你的任务需要完成一个实际的文件操作，就需要本地文件或目录字符串来避免错误。`Project`类中的`translatePath()`方法将把通用路径解释为操作系统专有的路径。`Project`类知道当前所用的平台，并把文件名或目录解释为正确的格式。例如：

```
File f = new File(dir, project.translatePath(filePath));
```

这个例子演示了如何创建一个文件。创建文件的任务并不需要任何平台检测代码来为所用平台(如 Windows 或 Unix)生成一个合法的路径。实际的做法是，编写任务的程序员可调用`translatePath()`，而且很清楚无论 JVM 之下是何种平台，此方法都可正常工作。

```
project.toBoolean()
```

检查一个 boolean 值。有 Boolean 属性（如，一个标志）的任务可以取值为 yes|no、true|false 或 on|off。利用方法 `toBoolean()` 可使这一点成为可能。这样就无需再重写将字符串转换为 Boolean 型的简单方法，而且为所有任务提供了一个一致的接口。带有类标志（flag-like）属性的所有任务都可以使用这 3 组 Boolean 值。例如 `project.toBoolean("yes")` 和 `project.toBoolean("on")` 都将返回 `true`。

如本节所展示的，使用 `Project` 类可以从构建引擎获得信息，除此之外，你还可以用它向构建引擎发送信息。不过这是一个破坏性的用法，而且非常危险。`Project` 类保存有对于构建引擎的许多操作都相当关键的信息，这意味着你可以在认为合适的时候对其加以修改。不过应当只在极端的情况下才做这个工作，或者根本不做（这样会更好）。之所以提到这种能力，只是为了保证我们所述信息的完整性，而不是建议你去实现。与构建引擎进行通信的最安全而且最好的方法是利用构建异常和日志消息。这是因为，一个任务应当完成的唯一一类通信就是提供信息的通信，而不是可能有破坏性的通信。这可以理解为如果出现一个错误，则要为运行时反馈提供状态消息或者妥善地失败。

构建异常

构建异常使用 `BuildException` 类来抛出，并且为任务提供一种向 Ant 构建引擎通知错误情况的机制。你可以在一个任务中的任意位置抛出 `BuildException` 异常。引擎会对任务对象进行方法调用，而每个方法调用都可能会产生一个 `BuildException` 异常。请看以下例子，在此抛出了一个 `BuildException` 异常：

```
if (!manifestFile.exists()) {
    throw new BuildException("Manifest file: " + manifestFile + " does not
exist.", getLocation());
}
```

如果所指定的清单文件在任务要使用它时并不存在，任务就会进入一种错误状态并失败。它会通过抛出一个 `BuildException` 异常向 Ant 引擎通知这种失败，此异常中包括一条错误消息和一个 `Location` 对象（可以使用 `getLocation()` 方法

获得)。Location类包括构建文件名以及引擎当前所解释行的行号。从某种意义上说，它也是一个类似于Project的类，任务可以通过它接收来自引擎的通信。不过，尽管可以用由Location类得到的信息来创建有关消息以放入BuildException异常中，但是大多数开发人员都对其使用都有所限制。

抛出一个BuildException异常将立即停止任务。对于一个目标来说，除非其所有任务都成功，否则该目标就不成功。有了BuildException类，Ant就知道何时使任务、其目标和构建失败。

对于以上规则(除非一个目标的所有任务都成功，否则该目标就不成功)有一个例外，即在一个任务中偶尔会使用一个failOnError属性。使用此属性的任务可以避免抛出BuildException异常，这样就允许构建得以继续。当然，这些工作并不是自动的，而你作为任务的编写者要负责实现这一功能。以下是取自Cvs类的一些代码，展示了如何实现failOnError。

XML为：

```
<cvs failOnErrors="true"  
cvsroot=":pserver:anonymous@cvs.phpwiki.sourceforge.net:/usr/phpwiki"  
dest="${src.dir}"/>
```

其实现(节选自*Cvs.java*源代码)为：

```
/**  
 * 任务的实例变量，表示failOnErrors标志  
 * 如果为true，则在cvs错误退出时终止构建  
 * 默认为false  
 */  
private boolean failOnErrors = false;  
  
...  
// 通过属性设置实例变量  
public void setFailOnErrors(boolean failOnErrors) {  
    this.failOnErrors = failOnErrors;  
}  
  
// 一些方法代码，在此省略
```

```
// 仅当任务要失败时,  
// 由此方法抛出一个构建异常  
public void execute() throws BuildException {  
    // 更多代码……  
  
    // 处理一个错误,  
    // 不过仅当 failOnErrors 为 true 时抛出一个异常  
    if(failOnErrors && retCode != 0) {  
        throw new BuildException("cvs exited with error code " + retCode);  
    }  
    // 更多代码……  
}
```

简单地说，如果 `failOnErrors` 属性为 `false`，`Cvs` 类不会抛出 `BuildException` 异常，并且会为包含此 `cvs` 任务的目标创建一个错误状态。此错误条件至少会生成一些日志消息，从而使终端用户了解到出现了问题，这总比什么都不做要好。例如，一个更好的实现为：

```
// 一些方法代码，在此省略  
// 仅当任务要失败时，抛出一个构建异常  
if(failOnErrors && retCode != 0) {  
    throw new BuildException("cvs exited with error code " + retCode);  
}  
  
if (!failOnErrors && retCode != 0) {  
    log("cvs existed with error code " + retCode);  
}
```

日志系统

`Project` 类允许任务得到有关构建文件的系统范围的信息。它还提供了访问构建引擎审计系统的一些方法，这些方法即为各种形式的 `log()`。所有消息可以根据一个称为消息级（message level）的引擎范围设置加以显示。

消息在以下 5 个级别上显示，在此以其“显示级别”为序：

- ERROR
- WARNING
- INFO

- VERBOSE
- DEBUG

这些级别向 Ant 指示了某个消息应当出现于哪个状态。例如，如果通知 Ant 只显示 INFO 级的消息，那么用 ERROR、WARNING 和 INFO 设置发送的所有消息都会显示在日志中。消息级值可以通过 Project 的以下公共静态字段得到：

```
Project.MSG_ERR  
Project.MSG_WARN  
Project.MSG_INFO  
Project.MSG_VERBOSE  
Project.MSG_DEBUG
```

VERBOSE 和 DEBUG 级很特殊，因为这二者看上去相同，但实际上并非如此。在运行 Ant 时，可以将 VERBOSE 级和 DEBUG 级的消息指定为不同的参数。指定 DEBUG 级的消息并不会显示出 VERBOSE 级的消息，反之亦然。

`log()` 方法向一个构建的已注册日志监听者发送消息。此监听者再根据其设计处理此消息字符串。默认的日志监听者将在控制台上打印出所有内容。`log()` 有 3 种形式：

```
log(message)
```

在任务中，消息通过 Project 类的 `log()` 方法记入日志。默认情况下，对 `log()` 的调用是一个 INFO 级消息（由 `MSG_INFO` 变量指定）。以下例子在默认的 `MSG_INFO` 级向构建引擎发送包含同样信息的消息。

```
project.log("This build step has completed successfully with " + numfiles + " processed");  
  
log("This build step has completed successfully with " + numfiles + " processed");
```

如例子中所示，这里还有一个默认的 `log()` 方法（在 Task 类中定义），这样任务甚至无需使用其 Project 实例变量。使用这个默认的 `log()` 方法是一个不错的想法，因为在将来的 Ant 版本中，可能会取消对 Project 类的任务级访问。

```
log(message, level)
```

`log()`方法的另一版本还有一个消息级参数。这对于发送 DEBUG 级和 VERBOSE 级消息来说很有用。例如：

```
// 使用 project 变量的 log() 方法记录日志消息  
project.log("For loop to process files begins", Project.MSG_DEBUG);  
  
// 使用 Task 的 log() 方法记录日志消息  
log("For loop to process files begins", Project.MSG_DEBUG);
```

注意这里有两种调用 `log()` 的方式。除了 `Project` 类以外，`Task` 类也有一个带有两个参数的 `log()` 实现。你应当尽可能使用 `Task` 的这个带有两个参数的方法 (`log(message, level)`)。

```
log(message, level, task)
```

`Project` 对象的 `log()` 方法还有第 3 个版本，其中有第 3 个参数，即一个 `Task` 对象。在一个用户编写的任务中不应当使用这种调用。它用于构建引擎中，在此提到它只是为了做到全面。

任务生命周期

复杂任务往往要在许多文件上完成操作，还需要依赖于嵌套任务，另外还会使用多个库（例如，可选的 `ejbjar` 任务），而这些就需要我们对任务与 Ant 的关系有深入的理解。请把这一点作为提醒。这一节将深入介绍涉及任务生命周期的具体细节。如果你认为你的定制任务不会达到在此所述的复杂程度，那么可以跳过这一节，并转向我们的例子。你完全可以以后再回来阅读这一节的内容。理解引擎和任务生命周期对于成为一个专业的任务编写者来说非常重要，如果只是要编写相对简单的定制任务，这一点则不是必要的。

Ant 会对所有任务进行同样的处理，它以固定时间间隔来设置属性和处理嵌套元素。我们可以预测一个任务将如何操作，并相应地对其进行设计。一个任务的生命周期可以分为两个主要时间段：解析时 (parse-time) 和运行时 (runtime)。解析时从 Ant 由 XML 读取任务时开始（由引擎逐元素地解释 XML）。运行时则从解析时阶段成功完成时开始。

解析阶段

Ant在其XML解析程序加载元素后对任务进行解析。任务名、任务的属性以及嵌套元素都被包装到一个XML元素对象中，并存储在Ant的内存中（in-memory）DOM中（注2）。在解析时，如果任务XML格式很差，或者任务构造函数中的活动抛出异常，那么操作就会失败。以下详细列出了在解析时Ant对任务完成的活动：

1. 对任务类进行实例化

Ant利用XML元素的名字和自省来对任务的相应类进行实例化。要记住，此时不会设置属性，而且指向构建系统的链接（如project实例变量）此时也不可用。

2. 创建对project和父目标对象的引用

引擎会置一些对象对于任务可用，而任务则利用这些对象与任务引擎进行通信。此时，Ant会创建这些引用，并令它们可用。

3. 增加id引用

Ant在一个内部表中保存一组有id属性的任务。这一步（在其发生时）仅对于其他任务和DataType是重要的。对于那些完成某种形式的并行处理的任务和DataType则尤为重要。有关parallel任务的更多信息请参见第七章，而这是Ant所发布的惟一一个能够完成并行嵌套元素处理的任务。

4. 调用init()

任务对象中的init()方法现在要得到调用。要记住，此时任务属性并不可用。另外，任务需要由嵌套元素得到的信息也不可用。顺便要指出，许多已发布的任务并没有实现这个方法。

注2：如果对于元素、DOM等的编程不太清楚，请参见Brett McLaughlin所著的《Java与XML（第二版）》（O'Reilly）。本书中文版已由中国电力出版社引进出版。

5. 嵌套元素得到解析，并利用 `addXXX()`、`addConfiguredXXX()` 和 `createXXX()` 等方法得到处理。

在整个生命期中，这可能是最重要（也是最困难）的一步。直观地看，你可能会认为 Ant 在解析时会定义和处理任务属性，但实际上并非如此。Ant 在运行时之前并不会对任务属性做任何处理。这也说明，如果构建文件中包含有未得到支持的属性，那么在运行时之前它们并不会被注意到。不过，Ant 会在解析时处理嵌套元素。因此，在捕获任何未得到支持的属性的使用之前，Ant 会先捕获未得到支持的元素的使用。

那么 Ant 如何处理嵌套元素呢？它会对你的任务调用 `createXXX()`、`addConfiguredXXX()` 或 `addXXX()` 方法，在此 `XXX` 是嵌套元素的首字母大写的名字。`createXXX()`、`addConfiguredXXX()` 和 `addXXX()` 方法之间有何区别呢？这取决于你计划如何使用嵌套元素，以及元素相应用对象的性质。如果任务需要实例化元素对象本身，或者如果对象没有默认的构造函数，则使用 `create`；可以把它看成是“你的任务创建了嵌套对象”。如果你的任务需要一个已实例化对象的引用，则使用 `add`；可以把它认看成是“Ant 为你的对象增加了一个对象引用”。如果你需要 Ant 在传递引用前完全地处理元素，则使用 `addConfigured`；可以把它看成是“Ant 为你的任务对象增加了已配置的对象引用”。如果你仍然不明白这些区别，请查看现有的任务实现。Ant 也会偶尔先调用 `createXXX()` 方法。对于一个特定的元素类型，如果你实现了多个方法，Ant 会对它们全部加以调用。这样做的后果可能是可怕的，因此要尽量避免这种做法。

运行时阶段

运行时阶段是完成任务的时刻。它从任务的解析时阶段成功完成时开始。在你的任务进入运行时阶段时，其他目标和任务可能已经成功地运行了。你可能希望依赖于原来期望的特定行为以及任务运行时步骤开始之前的状态设置，绝不要这么做！你的任务应当原子化地进行操作，而且要能够作为构建的第一个或最后一个任务来运行。以下列出了在任务运行时所发生的活动：

1. 任务的所有属性都得到设置

可以把属性考虑为任务的特性。Ant 通过为每个属性调用 `setXXX()` 方法从而为任务对象传递值，在此 `XXX` 是属性的首字母大写的名字。如果缺少了一个设置方法，则会出现 Ant 错误，而且任务和构建将失败。

2. 处理来自 XML 文件的 CDATA 文本

XML 允许你使用 `<![CDATA[]]>` 结构（即字符数据）在一个文件中放置原始文本，可以将此原始文本发送到你的任务。Ant 会调用方法 `addText(String msg)`，在此传递一个 `String` 对象，此对象表示来自 XML 文件的字符数据。以下是一个 CDATA 段的例子：

```
<taskname>
  <![CDATA[Naturalized language to be displayed by an Ant task]]>
</taskname>
```

当 Ant 读取 CDATA 段时，它会对任务调用 `addText("Naturalized language to be displayed by an Ant task")`。如果你的任务（或其父类）并未实现 `addText()` 方法，而你包括了一个 CDATA 元素，构建就会失败。对于字符数据的处理没有默认实现。

许多任务编写者没有使用 CDATA 功能。原始字符数据通常仅在某些任务中有用，这些任务带有消息或者必须结合没有用到转义码的文本。例如，`script` 任务为实际的脚本文本使用了 CDATA，因为对于像 “`<`” 和 “[” 这样典型的编程语言操作符，如果未置于一个 CDATA 段中，就会导致在 XML 中出现问题。

3. 所有嵌套元素的属性都得到设置

Ant 在读取其 XML 时会解析所有元素的属性。不过，直到运行时才会设置属性。这应用于所有的元素，其中也包括任务的嵌套元素。你很少需要担心嵌套元素中的属性状态。在任务执行之前（这一阶段的下一步骤），你可能并不会用到这些属性，只有在任务执行时属性才可用。

4. `execute()` 得到调用

到此为止，所有的工作都主要是数据收集和验证。利用 `execute()` 方法，你的任务才会完成其原来设计要进行的活动。从这一时刻起，你必须处理或提

出所有的错误情况。Ant不期望仅得到一个返回错误码，而不再对任务的方法做更多的调用。

同样，要编写一个任务，你不必完全理解任务生命期。但是如果你的任务中正在做的一项特定工作并未正常完成，而你试图找出其原因，那么对生命期有所了解将对你很有帮助。在某些很少见的情况下，你可能会找到一些方法来利用生命期从而使某些特定的事情发生。但要尽可能避免这样做。任务的某些工作细节不会总保持其目前的方式，它们会有所改变并且不易被觉察。除非计划维护你自己的内部Ant版本，否则你会发现必须坚持某一个Ant版本，因为你的任务只会在一个Ant版本下工作，而在另一版本下则不能正常完成。

生命期是很重要的，这是因为它允许Ant一致地处理任务。这样，从其他任务借用思想和代码就变得非常容易和常见了。

通过分析看示例：jar任务

既然已经有了理论上的基础，下面来看看实际的实现。若要开发你自己的Ant任务，就要编写一个实现你的设计的Java类。任务的复杂性或简单性取决于你。惟一重要的是，你的Java类必须遵循Ant对象模型中提出的约定。

作为一个如何编写任务的例子，我们对一个现有的任务（即jar）进行了分析。jar任务触及到我们需要了解的各个主题。Jar任务类属于深层次结构的一部分，展示了通过继承实现的重用。它派生自zip，而zip则进一步派生自MatchingTask。Jar任务对象并没有它自己的execute()方法实现，而是依赖于zip类中的相应方法实现。这说明对你自己的实现，某些需求是相当宽松的。jar任务还用到了大量的属性和嵌套元素，从而为我们提供了如何处理所有这些功能的好例子。使用一个现有的任务作为示例，这样做可以强调一个概念，即用户编写任务和包括在Ant发布中的任务之间并无区别。

对jar进行分析可以使我们对于如何设计一个任务有一些认识。它有着独一无二且易于理解的设计目标。我们要利用对象重用来设计一个任务，使之将来可以进行扩展。War和Ear派生自Jar，从中得到同样的收益。不过，在此不会谈到实

际 `jar` 任务的各个功能和方方面面。要了解更进一步的信息，可以花点时间来查看源代码发布中的代码。对于所有任务实现（而不仅仅是 `jar` 任务的实现）有更多了解，将使你成为一个更“厉害”的 Ant 任务开发人员。

注意：`Jar`、`Zip` 和 `MatchingTask` 的源代码可以在 Ant 源代码发布中找到 (<http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/src>)。我们分析 `jar` 任务时就利用了取自这些源文件的代码段。如果你不能认可我们的意见，或者无法理解一个代码段如何与相应描述取得一致，那么完全可以根据手边的完整源代码进行分析。

还要知道一点，即从创建一个工作任务的角度来说，我们的分析并不是全面的。我们触及到并解释了设计和编写 `jar` 任务的主要问题，但是对于如何为 JAR 处理输入流等实现细节则未做解释。这是一个分析，而非教程。如果你要按照这里的分析来编写和编译代码，就会发现有些内容不能正常工作。在简洁和完备之间存在着冲突，我们选择了简洁，当然也就会有所牺牲，即未能提供一个完整的用户编写任务教程。不过，我们的分析准确地描述了编写 `jar` 和其他任务所需付出的努力。除了我们所提供的内容以外，如果你还需要更多的信息，那么你只好去学习任务的源代码了。

作为开始，先来假设 Ant 没有 `jar` 任务。没有这个任务，第二章所介绍的 Ant 示例就没有可用的任务来创建其类的 JAR。使用 `java` 或 `exec` 来运行命令行 `jar` 工具过于麻烦，而且容易出现错误（正如本章介绍中所描述的那样）。

设计 `jar` 任务

对于一个创建 JAR 的任务，有哪些需求呢？从命令行工具 `jar` 开始是一个不错的想法。至少，我们的任务需要复制此工具的 JAR 创建 (JAR-creating) 功能（相对于此工具的所有其他功能而言）。这一点差别非常重要。在此不是重新实现 `jar` 工具，而是要为我们的构建创建一个操作，从而仅满足此构建的需求。命令行工具只是有利于达到这个目标。我们的构建要求创建 JAR，因此这个任务的设计就应该关注 JAR 创建，而不是其他的内容。假如将来需要定义其他的需求，如要解包 JAR，则需要为那些功能提供一个实现。

此命令行工具创建了一个与 zip 兼容的压缩文件，此压缩文件带有一种称为 *META-INF* 的特殊目录。命令行工具将一个名为 *MANIFEST.MF* 的特殊文件放入此目录中。在此不打算做过于详细的说明，我们只是将 JAR 描述为一种巧妙的 zip 文件，即此压缩文件不仅能够将一组文件打包到一个文件中，而且还有一种包描述文件（清单）。至少，我们的任务应当创建 JAR，而且如果存在一个用户编写清单文件，还应有其规范。

从构建的角度看，我们的设计应当允许我们利用来自多个目录（并有多种文件类型）的大量文件创建 JAR。由于 JAR 维护了类文件位置的目录结构，所以可能需要对于某些文件如何存储在 JAR 文件中进行调整。有经验的 Ant 用户会把这看成等同于文件集和文件模式（读完本章后，你也同样可以如此认为！）。通过对现有的任务做粗略的研究，可以发现某些任务也有同样的文件集设计，如 *copy* 和 *zip*。

简单地说，对于我们的 *jar* 任务，存在以下需求：

重复命令行工具的 JAR 创建功能

给定一个名字、一个清单文件名以及一组文件或目录，命令行工具就可以创建 JAR。我们的任务也要完成同样的工作。

对一组文件、目录和文件模式进行操作

许多任务都需要能够基于用户定义文件集信息以及用户定义文件模式来运行，我们应当做好准备利用这些功能。

增加和/或修改来自 XML 描述文件的清单文件

较之于相应的命令行工具，任务在功能上会有所扩展，这就是一个例子。在此并不是维护一个单独的清单文件，而是允许将清单设置置于构建文件内，当然要利用 XML 元素实现这一目的。

根据前面的需求分析，我们对于任务的 XML 语法应该有了一定的认识。在为自己的任务定义语法时，如果设计随着工作的进行而有所改变，请不要感到奇怪。

对于我们的任务，其 XML 设计为：

```
<jar jarfile="somefile.jar"
      manifest="somesmanifest.mf"
```

```
basedir="somedir">
<fileset dir="somedir">
    <include name="**/*.class"/>
</fileset>
<manifest>
    <attribute name="SomeAttribute" value="SomeValue"/>
</manifest>
</jar>
```

利用原有的工作

假设在没有 jar 任务的情况下，我们已经穷尽所有其他办法想使构建正常工作，现在我们知道需要编写一个定制任务。在此还必须再做一些研究，即必须确保我们是第一个做此工作的人！已经有数十个定制任务，Ant 发布中仅包括了其中的一些（而不是全部）。自 Ant 1.4 以后，Jakarta 小组在 Ant 网站上一直在维护一个任务列表，从而使用户可以访问一些最常使用的用户编写任务（请参见 <http://jakarta.apache.org/ant/external.html>）。除了这个网站，我们还需要搜索 Ant 邮件列表、Web 或 USENET 来找到可能实现了我们所需功能的现有任务。将来，可能还会有类似于 Perl 的 CPAN 库系统的任务库。

假设我们没有找到已有的 jar 任务。下一步，再来查找是否存在某些已有的任务，而其功能类似于 jar 任务。在实际工作中，你可能并没有足够的经验来了解正在编写的任务与现有任务之间的关系。要确定一个所需任务的功能（或其一部分）是否在当前已有的一些其他任务中存在，请仔细参阅第七章和第八章。

前面已经提到，JAR 只不过是带有一个清单文件和一个不同文件扩展名的 ZIP 文件。正因如此，我们来看 zip 任务是否可能进行重用。zip 任务完成了一个类似的操作，它要由一组模式和规则创建一个打包的文件。实际上，此操作仅在 MANIFEST 的概念以及所得到的文件名方面存在不同（是 .zip 而非 jar）。因此可以做出决定，就从 Zip 来派生我们的对象！

以下是我们的 Jar 类签名：

```
package org.oreilly.ant.tasks;

// 在由此派生之前先需要导入
import org.apache.tools.ant.taskdefs.Zip;
```

```
/*
 * Ant 中对于<jar>任务的实现类
 *
 * 在你的任务中，要确保在此显示任务的几个使用例子
 * 而且，如果计划对实现做其他扩展，
 * 则要描述你的某些方法如何应用，
 * 以及你的任务通常如何工作
 */
public class Jar extends Zip {
    // 实现代码
}
```

由 `Zip` 派生时，所派生的类自动成为 Ant 任务框架的一部分。主任务框架类 `org.apache.tools.ant.Task` 定义了一个任务所需的基本方法（注 3）。这些方法，再加上你在任务实现中提供的那些方法，允许一个任务确定由构建文件中的 XML 元素给定的属性，还可以确定在工程中设置的其他特性。

`org.apache.tools.ant.taskdefs.MatchingTask` 扩展了 `org.apache.tools.ant.Task`，而且对于有文件和目录方法需求的任务，还为其实现了所需的文件和目录方法。诸如 `copy` 和 `zip` 等任务就派生自 `MatchingTask` 以继承这些方法。第四章对于模式和文件集给出了完整的解释。

这里的关键是要找到重用功能。存在一个任务对象模型意味着多种常用功能都可以派生自同样的父任务对象。利用以前的工作并不只是寻找完成同样工作的代码实现，而且还要寻找对其有所补充的对象。这个对象模型相当强大，而且也解释了 Ant 何以在 2 年之内能够发展得如此迅速。在设计和开始阶段的研究上付出大量努力，最后必然会由此受益。对框架做有益的修改可以对所有任务都有帮助，从而只需少量维护甚至无需维护。

实现属性设置方法

Ant 通过一组由任务编写者所定义的设置方法来设置任务的属性。这些方法的名字遵循的约定类似于 JavaBeans 特性设置方法：`set` 后跟着首字母大写的属性名。这些方法必须是公共可见的，而且不向调用者返回任何东西。参数通常是一个

注 3：要记住，对于 Ant 1.4，实际的框架类为 `ProjectComponent`、`DataType` 和 `Task` 可由此派生。不过，`Task` 总是由 `org.apache.tools.ant.Task` 派生。

`String`, 但也可以是如下所列的任何一个对象, 还可以是任何基本类型 (由 `String` 对象转换得到), 也可以是任何用户定义类型 (要求有一个以 `String` 为参数的构造函数)。合法的属性设置方法的参数类型如下:

`String`

最常用的参数。Ant 将由 XML 元素属性得到的原始值传递给设置方法。

`File` 对象

如果 Ant 确定设置方法有一个 `File` 参数, 则试图创建一个 `File` 对象, 它与 `<project>` 元素的 `basedir` 属性中所设置的目录相对应。

`Class` 对象

如果属性值是一个完全限定的类名, Ant 就会试图通过类加载器来加载此类。

在核心和可选的 Ant 1.4.1 发布中, 还没有使用此做法的任务例子 (注 4)。

用户定义对象

如果你的新类有一个构造函数, 而且它仅有一个 `String` 参数, 则可以将此类用于任何设置方法签名。通常, 最好是将你的类作为任务对象的一个私有成员。此类的实现和可视性要由其包含任务所限制。这样, 就可以避免某些人在从一个 JAR 得到的类列表中看到你的对象时, 将它用作一个任务。

要记住, 我们的 `jar` 任务并没有为所有属性实现设置方法, 而仅仅对 `zip` 任务所不能处理的属性提供了设置方法, 或者对需要做不同处理的 `zip` 属性实现了设置方法 (覆盖父对象的设置方法)。表 5-1 列出了此 `jar` 任务的属性 (请参见前面所示的 `jar` 的 XML 示例)。

表 5-1: `jar` 任务的 JAR 专有属性

属性名	描述	是否需要在 <code>Jar</code> 任务对象中实现
<code>jarfile</code>	所得到 JAR 文件的文件名	是, 在 <code>zip</code> 任务对象中不可用
<code>manifest</code>	用于验证和包含的清单文件 的文件名	是, 在 <code>zip</code> 任务对象中不可用

注 4: 尽管是从理论上讲, 但这种技术确实可能有实际用途。在任务执行期间提供一个运行时类实例, 对于仅在运行时给出定义的复杂操作来说可能很有用。

表 5-1: jar 任务的 JAR 专有属性 (续)

属性名	描述	是否需要在 Jar 任务对象中实现
basedir	JAR 文件所来自的根目录	不, Zip 任务对象为此属性实现了设置方法

以下为 `setJarfile()` 属性设置方法的实现。它取一个 `File` 对象作为参数。Ant 通过自省检测此参数，并试图利用由 XML 得到的属性值来创建一个 `File` 对象。创建 `File` 时出现的 XML 错误来自于 Ant 本身，你不必操心处理非法的文件名等等。而且，由于我们要借用 zip 的方法，所以只需调用 zip 的 `setZipFile()` 方法，因为该方法设置了此任务实例的 `File` 对象。

```
/*
 * 设置 JAR 文件名的值
 * 实例变量为 zipFile
 */
public void setJarFile(File pValue) {
    log("Using Zip object 'setZipFile' to identify the JAR filename", MSG_DEBUG);
    super.setZipFile(pValue);
}
```

对于另一个例子，我们将显示另一个属性 `manifest` 的设置方法，此方法仅为 `jar` 所独有。与 `setJarFile()` 类似，`setManifest()` 方法也以一个 `File` 对象作为参数。

```
/*
 * 要打包到 JAR 中的设置清单文件
 * manifest 实例变量可用于
 * 增加新的 manifest 属性项，它对应为
 * jar 任务的嵌套元素
 */
public void setManifest(File manifestFile) {
    // 此属性是一个 File

    // 检查以确保文件确实在于其所声称的位置
    // 如果并非如此，则抛出一个 BuildException 异常，且任务失败
    if (!manifestFile.exists()) {
        throw new BuildException("Manifest file: " + manifestFile + " does not exist.",
getLocation());
    }
}
```

```
// 设置清单文件对象的实例变量
this.manifestFile = manifestFile;

InputStream is = null;
// 加载清单文件
// Conor MacNeil 编写了一个用于处理清单文件的对象，且在 Ant 中可用
// 此对象可以保证清单文件格式正确，
// 而且有正确的默认值
try {
    is = new FileInputStream(manifestFile);
    Manifest newManifest = new Manifest(is);
    if (manifest == null) {
        manifest = getDefaultManifest();
    }
}

manifest.merge(newManifest);
// 下面为任务开发人员记录这个操作
log("Loaded " + manifestFile.toString(), Project.MSG_DEBUG);
} catch (ManifestException e) {
    // ManifestException 异常由 Manifest 对象抛出

    // 与 Manifest 对象类似,
    // 还存在一个定制对象用以警告清单文件错误
    log("Manifest is invalid: " + e.getMessage(), Project.MSG_ERR);
    throw new BuildException("Invalid Manifest: " + manifestFile, e.getLocation());
} catch (IOException e) {
    // IOException 由任何文件 / 流操作抛出, 类似于 FileInputStream 的构造函数
    throw new BuildException("Unable to read manifest file: " + manifestFile, e);
} finally {
    // 由于我们已经将文件读入一个对象中,
    // 下面关闭流
    if (is != null) {
        try {
            is.close();
        } catch (IOException e) {
            // 仅将此异常记入日志, 并不做其他操作
            log("Failed to close manifest input stream", Project.MSG_DEBUG);
        }
    }
}
```

正如属性表所指出的，我们无需提供 `setBasedir()` 方法的实现。

实现嵌套元素处理

实现代码以处理嵌套元素是编写任务时最复杂的一部分。与属性类似，你要通过一些方法来处理嵌套元素，这些方法也遵循命名约定。Ant 取得各个嵌套元素的相应任务对象，并试图调用其 3 个方法中的一个。在这种情况下，方法命名约定为 `addXXX()`、`addConfiguredXXX()` 和 `createXXX()`，在此 `XXX` 为嵌套元素的首字母大写的名字（如，`addFileset()` 处理一个任务的 `<fileset>` 嵌套元素）。要想知道需要实现哪一个方法可能很困难也很容易弄混。各方法之间的微小差别在于 Ant 如何管理各个嵌套元素对象。以下列表对于何时为一个嵌套元素实现 `addXXX()`、`addConfiguredXXX()` 或 `createXXX()` 方法提供了宽松的定义。一般来说，你要选择最满足需求的技术，并实现相应的方法。即使只是理解这些定义如何应用于你的需要，这可能也很困难。不过，后面我们对于 `jar` 任务的分析将有助于理解这个问题。

`addXXX()`

在“增加”一个嵌套元素时，即会通知 Ant 在调用 `addXXX()` 方法之前先实例化该类。如果嵌套元素的相应用对象没有默认构造函数，Ant 就无法做此工作，并抛出一个错误。如果有默认构造函数，Ant 则将实例化的嵌套元素对象传递至你的任务对象，在此可以根据需要对该对象进行处理（例如，将它保存在一个集合中等等）。我们建议，如果仅仅是要避免嵌套元素属性未设置而可能带来的问题，那么就应该在任务的执行阶段再真正使用嵌套元素对象（也就是说，调用方法或由它获取值）。

`addConfiguredXXX()`

你现在可能会认为：“我必须在执行阶段之前使用此嵌套元素！”。幸运的是，Ant 为增加对象提供了一个候选方法。`addConfiguredXXX()` 方法要求 Ant 不仅进行实例化，还会在将嵌套元素对象传递给任务对象之前对它进行配置。换句话说，Ant 可以保证，对于给定的嵌套元素，其属性和嵌套元素在到达任务对象之前均得到设置和处理。由于这种技术打破了任务生命期，所以使用此方法时就存在一些危险，尽管其影响甚微。即使 Ant 为你配置了此元素，仍要记住 Ant 并没有完成对当前任务的配置。你会发现在 `addConfiguredXXX()` 调用期间，父任务的属性均为 `null`。如果你试图使

用这些属性，就会带来错误，最终导致构建运行终止。在方法参数中可用的类型存在一些限制。像 `addXXX()` 方法一样，如果当前对象没有一个默认构造函数，就无法将此嵌套元素的对象作为 `addConfiguredXXX()` 方法的参数。

`createXXX()`

如果 Ant 调用了一个 `createXXX()` 方法，那么对于将嵌套元素解析为任务对象就提供了完全的控制。Ant 没有向任务传递对象，而是需要任务返回嵌套元素的对象。这有一些好的副作用，其中最显著的就是消除了嵌套元素对象必须要有默认构造函数的要求。其缺点在于你要理解元素对象在初始化时是如何工作的。也许你手边并没有文档或源代码，因此这可能是一个极为困难的工作。

这些都是很宽松的定义，因为从程序上不要求你必须使用它们。相应于嵌套元素，只要你对于其 3 种方法中的一种提供了实现，Ant 就能够使用你的任务及其嵌套元素。不过，在你查看 Ant 的源代码发布（特别是用户编写任务的源代码）时，你会注意到开发人员在某些地方并没有遵循这些定义，而且，实际上它们还存在混杂的情况。在编写元素处理方法时没有严格的规则，因此总会有一些使用情况违反前面所指出的定义。

`jar` 任务需要能够设计一组模式以包含和排除一些文件和目录。它还需要有一种方法，以便在 JAR 的清单文件中增加项。在我们的设计中，选择使用嵌套元素来实现这种功能。第一个需求，即模式处理，已经是 `MatchingTask` 对象实现的一部分。第二个需求，即为清单文件指定属性，则需要在 `jar` 的实现中给出显式的处理。再来看此任务的 XML，特别是嵌套元素：

```
<jar jarfile="test.jar"
      manifest="manifest.mf"
      basedir="somedir">
  <manifest>
    <attribute name="SomeAttribute" value="SomeValue"/>
  </manifest>

  <fileset dir="somedir">
    <include name="**/*.class"/>
  </fileset>
</jar>
```

由此示例 XML，我们可以对 jar 任务的嵌套元素建立一个表（请参见表 5-2）。我们给出了其描述，并说明了此类是否必须实现相关功能。要记住每个嵌套元素都有其自己的相应类。在此分析中，我们假设这些类都已经编写并能正常工作。其实现与 jar 任务的实现在概念上稍有差别。

表 5-2: jar 任务的嵌套元素

嵌套元素名	描述	是否需要在 Jar 任务对象中实现
Manifest	向 JAR 的清单文件中增加项 ^a	是，这在 Zip 对象中不可用
Fileset	为 JAR 中的包含和排除创建文件模式	不，MatchingTask 对象实现了这些方法。Zip 由 MatchingTask 继承得到

a: 有关 JAR 及其清单的更多信息，请参见 Sun 公司关于 JAR 规范的文档。

JAR 清单文件

清单文件是 JAR 规范中以往未得到充分利用的部分。利用清单文件，你可以增加关于压缩文件中包括哪些内容的描述。通常，这些描述为版本号或库名。能够在一个构建文件中指定清单项，这可以减少在源代码中管理清单文件的需要。在编写原始的 jar 任务时，开发人员提供了一个 Manifest 对象，由它来管理清单信息（如，其属性及属性的值），而且可以将信息写到磁盘上从而包含在 JAR 中。另外，此 Manifest 对象了解并且可以处理嵌套的 <attribute> 元素。对于我们的用途，在此假设这个类已经存在并能正常工作。

乍一看，我们似乎需要 Ant 在通常的“嵌套元素”阶段处理此 <manifest> 元素。这遵循一般的任务生命期。但是，等待处理 <manifest> 元素意味着由此元素得到的值和数据直到生命期的执行阶段时才可用。这就要求我们具体实现 Jar 任务对象的 execute() 方法，而这是我们力图避免的。在此希望使用 Zip 对象的 execute() 方法，并需要 Ant 在执行阶段前就处理 <manifest> 元素。下面来看 Jar 类的 addConfiguredManifest() 方法：

```
public void addConfiguredManifest(Manifest newManifest)
    throws BuildException {
    if (manifest == null) {
        throw new BuildException();
    }
    manifest.merge(newManifest);
}
```

addConfiguredXXX()系列的方法通知Ant在解析元素时对其处理，而不是等到运行时阶段才做处理。在这种情况下，newManifest参数应当是一个内容完全的Manifest对象。除了完成一些基本的错误检查以及与原有清单文件的内容合并外，此方法不做其他的工作。原有的清单文件来自于jar任务的manifest属性。不过，如果当前清单不存在，merge方法就会要求Manifest创建一个新的清单，这个方法是Manifest对象的一个功能。

文件模式匹配对于许多Ant任务都很常见，因此理解其实现非常重要。你很少需要自行实现代码来处理文件模式。要查看模式匹配的完整实现，请查阅Ant源代码发布中的Zip和MatchingTask源代码。以下是<fileset>嵌套元素处理方法addFileset()的实现：

```
/**
 * 增加一组文件（嵌套元素属性）
 */
public void addFileset(FileSet set) {
    // 向实例变量filesets 增加FileSet 对象,
    // filesets 是一个FileSet 对象向量 (Vector)
    filesets.addElement(set);
}
```

有关生命周期和嵌套元素的介绍是很复杂的，不过你会认为这个问题本身还要更为复杂，不是吗？Ant有一个绝妙之处，这就是它极为依赖于面向对象设计和自省。对象编程的性质意味着设计有时是复杂的，但这会带来编码和代码维护的容易性。正是由于有XML标签 - 类的关系这种独一无二的概念，才使得以前的代码段相当短小。在编写一个类似于jar的任务时，可以假设FileSet对象负责处理全部事情。你只需考虑设计良好的接口。

由于 Jar 类需要维护一个 FileSet 对象列表，所以还需要将它们保存在某处。幸运的是，Java 有大量的集合类，在这种情况下，我们使用了 Vector（注 5）。当然，对于 FileSet 对象向量实际所做的要复杂得多。幸运的是，只需在一处（即 execute() 方法）为 jar 任务编写实现，而且对于 jar 任务，我们甚至无需自己来编写！

实现 execute() 方法

execute() 方法实现任何任务的核心逻辑。在编写任务时，实现任务的 execute() 部分是最容易的部分。Ant 引擎在达到任务处理的最后阶段时会调用 execute()。execute() 方法既没有参数也不返回任何值。这是 Ant 对于任务调用的最后一个方法，因此，此时你的任务类应当已经有了工作所需的全部信息。

在前面的一节中，我们提到 Zip 实现了一个极合适的 execute() 方法版本；我们无需再为 Jar 类编写一个方法实现。这不是逃避责任，而只是高效代码重用的一个很好的例子。为了解释我们为什么无需编写自己的 execute() 方法，下面将继续分析 Zip 的 execute() 方法。在这里的分析中并没有涉及 ZIP/JAR 专有的操作，因为我们的重点是学习如何编写 Ant 任务，而不是如何通过程序来构建和管理 JAR。

对于 execute() 方法，我们将分为 3 个部分来分析：验证、完成实际工作和错误处理。要描述如何实现一个任务的核心操作，这是一个简单而通用的方法。在编写你自己的任务时，要谨记这些部分，因为这可以帮助你设计出一个更好的任务。不过，在逐一分析 execute() 方法的各个部分之前，我们先来看看此方法的签名：

```
public void execute() throws BuildException {
```

在此并无特殊之处。没有需要考虑的参数或返回值。错误通过 BuildException 异常传递，这与在所有其他任务接口方法中一样。

注 5： 你可能会想：“为什么不用 List 或 ArrayList 呢？为什么是同步的 Vector 呢？”Ant 的设计需求要求与 JDK 1.1 兼容。这些集合类在 Java 2 之前并未加入进来，因此要使用 Vector。

验证

在我们的分析中，第一部分是关于验证的。我们需要验证 jar 任务属性的值，另外还必须检查任务是否需要基于属性的值来运行。合法的属性是非 null 的，对于任务如何使用此属性有相应的参数，而合法的属性还应该表示此参数中的值。在大部分情况下，这种验证都发生在设置方法中。不过，由于 Ant 调用设置方法并没有固定的顺序（例如，给定 6 个属性，从技术上说，不可能指定哪一个先设置），两个或更多属性之间的关系验证则必须在 execute() 中完成。所有运行时验证也必须在 execute() 中完成。

在以下代码段中，我们检查了任务中“必要的”属性和元素。在这种情况下，我们仅需要 basedir 属性和 <fileset> 元素。

```
if (baseDir == null && filesets.size() == 0) {  
    throw new BuildException("basedir attribute must be set, " +  
        "or at least one fileset must be given!");  
}
```

在此，我们要进行检查以确保 ZIP 文件（这里是 JAP 文件）的名字是合法的（非 null）。

```
if (zipFile == null) {  
    throw new BuildException("You must specify the " + \  
        archiveType + " file to create!");  
}
```

这就是验证的全部。确实没有多少工作，但是这些短小的代码段对于避免将来出现的错误可谓功绩卓著。如果在任务实现中做了很好的验证，就会为以后节省大量的时间。

完成实际工作

在对 execute() 方法的分析中，第二部分是关于 JAR 文件的创建，在此使用了 Ant 为创建文件集所提供的对象。这里我们要介绍两个辅助对象，即 FileSet 和 FileScanner。它们表示了保存文件和目录集合的不同方法，不过在功能上却并不相同。FileSet 对象直接与 <fileset> 元素及其子元素相关。FileScanner

则是一个能够完成底层文件系统分析的对象，而且与平台无关。它可以将文件集或其他扫描器与自己相比较，以确定文件是否有所修改或遗漏。一旦 Ant 处理了 <fileset> 元素，FileSet 对象就有许多强大的方法可以由所建立的对象抽取出信息。

以下代码段使用了基目录属性 (basedir) 以及文件集来创建一组扫描器。在这种情况下，如果压缩文件存在的话（例如，来自前一次构建），我们将创建一组扫描器从而与压缩文件比较。这是一个最新 (up-to-date) 检查，在可能的情况下，可以消除不必要的工作。getDirectoryScanner 方法来自于 MatchingTask。

```
// 创建扫描器以传递至 isUpToDate()
Vector dss = new Vector ();

// 在基目录下创建一个“可检查的”文件 / 目录列表
if (baseDir != null) {
    // getDirectoryScanner 可由 MatchingTask 对象得到
    dss.addElement(getDirectoryScanner(baseDir));
}

// 创建来自 FileSet 的文件 / 目录的一个“可检查的”列表
// 在此使用 FileSet 的属性
// 将 project 对象传递进来,
// 从而使列表可以将全局过滤器设置包括在工程的特性中
for (int i=0; i<filesets.size(); i++) {
    FileSet fs = (FileSet) filesets.elementAt(i);
    dss.addElement (fs.getDirectoryScanner(project));
}

// 为 isUpToDate 方法创建 FileScanner 数组
int dssSize = dss.size();
FileScanner[] scanners = new FileScanner[dssSize];
dss.copyInto(scanners);

// 如果目标是最新的，则退出
// 这种方法也可以处理空的压缩文件
if (isUpToDate(scanners, zipFile)) {
    return;
}
```

以下代码段发生在一个 try-catch 块中，在此将捕获 IOException 异常，并有一个 finally 子句来关闭 ZIP 压缩文件的文件流（将在分析的下一部分对 catch

块进行分析)。这段代码将文件集以及基目录下的文件增加到作为 ZIP/JAR 压缩文件的输入流中。`addFiles` 的实现则不那么重要。它使用 `FileSet` 对象来得到各个文件名，并将各文件放入一个 `InputStream` 中。

```
try {
    // 将隐式的 fileset 增加到压缩文件中
    // 基目录通过 basedir 属性设置
    if (baseDir != null) {
        addFiles(getDirectoryScanner(baseDir), zOut, "", "");
    }

    // 将显式的 fileset 增加到压缩文件中
    // addFiles 可由 Zip 对象得到
    addFiles(filesets, zOut);
}
```

`try` 块部分提供了创建压缩文件和流的实际功能，在本章中不做介绍。简要地说，它使用了它的辅助对象来为压缩文件创建显式的文件列表。它将清除所有的临时文件，并关闭流和文件对象。如果存在任何错误，`Zip` 对象就必须抛出一个 `BuildException` 异常，从而导致构建失败。这就是为什么要将与文件以及流相关的清除和关闭例程放在 `finally` 子句中的原因所在。这些文件和流必须关闭，而不管构建的错误状态如何。我们将在下一节中更为详细地谈到这个问题。

错误处理

分析的第三部分是关于错误处理的。你可能会认为前面的验证会处理所有的错误，但事实并非如此。因为我们处理的是文件和文件系统，这就存在大量的 `IOException` 异常隐患。我们利用 `BuildException` 异常将错误返回给 Ant，因此，表示错误、`null` 对象和 `IOException` 异常的任何内容最终都会变成一个 `BuildException`。为了与用户进行更好的、更准确的通信，需要对你的错误进行分析，并创建描述性的错误消息。这些就是将出现在构建日志中的消息，因此它们必须是适合阅读的，同时还要提供一个一致的文本布局，从而使你和其他用户能够在日志中进行文本查找。

以下代码段是前一节所示 `try` 块的 `catch` 块。假如处理流或文件时出现一个 `IOException` 异常，此代码将创建一个描述性消息。这包括在删除一些压缩文件

前显示其检查结果。BuildException 异常包括消息、原始错误异常以及位置。要记住，Ant 将一个名为 location 的对象维护为某种执行指针，它有着 XML 的行号以及出现错误的构建文件的名字。

```

} catch (IOException ioe) {
    // 某些 IO (可能是文件) 失败, 下面来进行检查

    // 创建一个描述性消息
    String msg = "Problem creating " + archiveType + ": " + ioe.getMessage();

    // 删除错误的 ZIP 文件
    // 这基本上可以使我们避免部分创建的 zip/jar
    if (!zipFile.delete()) {
        msg += " (and the archive is probably corrupt but I could not delete it)";
    }

    // 此功能处理更新的 jar
    if (reallyDoUpdate) {
        if (!renamedFile.renameTo(zipFile)) {
            msg+=" (and I couldn't rename the temporary file "+
                renamedFile.getName()+" back)";
        }
    }
}

// 消息已经建立。将它发送回 Ant
throw new BuildException(msg, ioe, location);
}

```

编译任务

编译任务涉及到使用当前的 Ant 库文件 (*ant.jar*)，还要用到你的任务的基本包结构。许多人将其定制任务放在 `org.apache.tools.ant.taskdefs.optional` 包中，不过 Ant 并不要求这样做。可以选择一个最适于你的包和工程组织。除非你编写了许多任务，否则以后再修改包也很容易。

你可以编写一个 Ant 构建文件来构建你的任务。以下是一个小的构建文件，我们将由此开始。

```

<!-- Build the custom tasks in this project directory.  We'll
assume that all the custom task classes are packaged under
the 'src' directory and that the results will wind up in

```

```
'dist'. Users must change the value for the Ant directory  
and include any further libraries they choose to use with their  
tasks.  
-->  
<project name="customtasks" basedir=". " default="all">  
    <property name="src.dir" value=". /src"/>  
    <!-- Note the absolute directory. CHANGE THIS BEFORE BUILDING -->  
    <!-- It would be possible to use environment variables, but we do  
        not assume they are set -->  
    <property name="ant.dir" value="/opt/ant"/>  
    <property name="ant.lib" value="${ant.dir}/lib"/>  
  
    <property name="build.dir" value=". /build"/>  
    <property name="dist.dir" value=". /dist"/>  
  
    <!-- Compile all of the task object classes -->  
    <target name="all">  
        <mkdir name="${build.dir}" />  
        <javac srcdir="${src.dir}"  
               destdir="${build.dir}">  
            <classpath>  
                <fileset dir="${ant.lib}">  
                    <include name="**/*.jar"/>  
                </fileset>  
            </classpath>  
        </javac>  
        <copy todir="${dist.dir}">  
            <fileset dir="${build.dir}" />  
        </copy>  
    </target>  
</project>
```

此构建文件将编译你的定制任务对象，可见于子目录 *src* 和相应的包目录中。然后再将所得到的类复制到 *dist* 目录下合适的包结构中。一旦有了类，我们只需部署和定义任务，使之对于 Ant 可见，为此使用了 *<taskdef>* 元素（有关此元素的更多信息可参见下一节，“部署和声明任务”）。

对于本章实现的 jar 版本，诸如以下的一个工程就能正常工作：

```
mytasks/  
build.xml  
dist/  
build/ (temp build directory)  
src/org/myorg/tasks/*.java
```

要让它做到简单。如果只是编写了一个任务，就没有必要超出此目录结构之外来管理你的任务工程。一旦构建了jar任务，则将它放在*dist*目录中的一个JAR中。

在目录和构建文件之间，用你的任务来创建一个新的JAR是轻而易举的。现在要做的全部工作就是部署任务，并使之对于你的构建文件可用。

部署和声明任务

用户编写任务可以用两种方式分别部署为开放的类或JAR，其区别仅在于将来的维护有所不同。为了进行比较，所有内置的任务都被部署为一个JAR，它们是Ant JAR(*ant.jar*)的一部分。在该压缩文件中有一个文件，即*defaults.properties*。在此文件中，维护人员可为Ant可用的各个任务做默认的声明。作为一个特性文件，它是一个名-值对的列表。我们可以将对此特性表进行扩展从而声明我们的定制任务。

如果在Ant的源树中增加一个任务，从理论上说，你可以修改*defaults.properties*文件，并增加你的新任务。在这种情况下，并不是单独地编译你的任务，而是必须完全重新构建Ant，并创建一个新的Ant JAR。这种方法对于Ant的系统范围发布是最佳的，在此需要一个小组中的所有开发人员维护并使用一个同构的开发环境。你的小组必须维护其内部的Ant版本，但是由于有可能已经维护了多种其他工具，因此再多一种影响并不大。

以下是一个例子。如果希望将任务foo（利用相应的对象org.apache.tools.ant.taskdefs.optional.Foo）增加到Ant中的核心任务集合中，就要打开src/main/org/apache/tools/ant/taskdefs中的文件*defaults.properties*，并增加如下一行代码：

```
foo=org.apache.tools.ant.taskdefs.optional.Foo
```

其结果是，下一次构建Ant时，你的任务类及其声明会成为核心任务列表中的一部分。如果对构建Ant的更多详细内容感兴趣，可以参见Ant源代码发布中的docs/manual/install.html#buildingant。

如果你没有使用前面提到的方法，则必须为 Ant 声明一个用户编写任务，在此要通过在每个使用此新任务的构建文件中使用一个`<taskdef>`元素来实现。必须将这些元素放在构建文件的工程级或目标级，这取决于对于你声明的各个定制任务所希望的功能作用域。工程级任务对于一个构建文件中的各个目标都可用，而目标级任务则仅在该特定目标中可用。对于目标级的声明，声明的位置很重要。你不能在声明一个定制目标级任务之前使用它。

以下是一个`<taskdef>`元素的例子，它定义了任务 jar，并将 Jar 指定为实现类：

```
<taskdef name="jar" classname="org.apache.tools.ant.taskdefs.Jar"/>
```

`<taskdef>` 元素有一组属性，由此可确定要使用哪个特性（或特性集）。一般来说，你会使用 `name` 和 `classname` 属性来定义任务的名字（元素名）及其类实现。你还可以指定资源，例如一个特性文件的资源，在此保存一个任务名和任务类列表。对于其所有属性的完整内容请参见第七章中关于 `taskdef` 的文档。

关于任务的其他内容

由于每隔 6 个月就会修改，所以 Ant 绝非处于完美状态。它的某些表现并不总是立即就能显示出来。它还存在一些问题、bug 以及在发布文档中未做说明的隐藏功能。以下几节将描述你在编写自己的任务时需要了解的一些问题。如果你不打算更为稳妥，那么完全可以直接实现你的任务并进行部署，然后会看到将发生什么情况。如果有无法解释的问题，可以再跳回到这一节，看看以下某项内容是否对你有所帮助。除非 JVM 规范做了修改，否则有些问题就不会有异常表现，如 `System.exit()`。其他的一些问题，如魔法特性，则会在将来一些新的任务模型实现发布时出现异样。当然，你可以实现一个任务测试，以此来尽量避免将来可能出现的各种问题。

魔法特性

数月以前出现了 `javac` 任务。许多人认为它很好，但还有其他一些人不赞同。当时，对于主要的 Java 平台（Solaris、Linux 和 Windows），至少有 3 种可用的不

同编译器。这些编译器为 *javac* (及其不同的编译模式)、IBM 的 *jikes* 以及 Symantec 的 *sj*。开发人员并不是将编译器类型定义为 `<javac>` 元素的一个属性，而是认为应当有一个全局的设置，并能影响 *javac* 任务的所有使用。此全局设置应用于 *javac* 的每一次出现，或者应用于由 *Javac* 类派生的任何相关任务的每次出现。例如，只需修改一行，Ant 用户就能从 *jikes* 切换到 *javac*。这很好，不是吗？但是答案却并不确定。

如果有一个全局的编译器标志，对于保证所生成字节码的一致性来说是不错的。一般来讲，你不会用 *jikes* 来编译工程中的一部分，而用 *javac* 编译另一部分。实际上，像编译器标志这样的标志本身是一个很好的想法。但是，其缺点在于这是全包含的。如果你确实需要构建文件中的某些 `<javac>` 元素使用 *jikes* 而另一些元素使用 *javac*，那么情况将如何呢？Ant 的回答将是“绝对不允许”。对于你的任务的设计来说，采用同样的观点则是不好的。因此，既然我们已经知道了魔法特性的后果，那么现在为什么还要来考虑它呢？

令魔法特性成为可能的实现依赖于 Ant 任务模型中的某种设计漏洞（有些人这样认为）。所有任务都有 *Project* 对象的引用。简单地说，*Project* 对象是 Ant 引擎中全能的对象。它拥有所有特性、目标、任务、*DataType* 等等的引用。有了 *Project* 对象，所有任务都可以看到任何特性（包括魔法特性），即使一个任务并未在任务元素的标签中显式声明。只要你以一种合理且可读的方式来使用这种能力，一切都会很正常，至少从编程的角度说是这样。

为了说明魔法特性不是一个好的想法，下面来看从一个构建文件编写者的角度看到的问题，具体来说，就是从构建文件的 XML 标签的角度。在 XML 中，任务是自包含的元素。一个任务的“作用域”由其开始标签开始，并结束于其结束标签。如果加入一些影响到任务操作的特性，但是却在任务的开始和结束标签之外定义，则会打破 XML 的可读性，并丧失了作用域的可见且直观的概念。

有人可能会争论说常见的特性替换（例如，`attribute="${some.predefined.property}"`）也属于我们所描述的问题的一部分，但这里所讨论的完全是另一回事。即使你可能在一个任务的作用域外定义一个特性，或者甚至在构建文件的作用域外定义，使用该特性的位置在任务的 XML 标签中都是非常明显的。即将此

特性用作任务属性的值或一个任务嵌套元素的属性值。无论哪一种情况，属性都在构建文件中很清楚地指出了特性值的用途。与之相反，魔法特性则会仅做一次声明，而再不会提到它。并不要求你将一个魔法特性的声明与使用它的任务相连接。当然，你总是可以在构建文件中增加一些 XML 注释，但是 Ant 并不要求你编写注释。如果某个任务需要，Ant 会要求你设置一个属性。

对于小的构建文件，你可能不会注意到魔法特性的问题。在这些构建文件中，作用域很少会成为问题。在大的工程中，特别是使用了级联工程目录和构建文件的工程，魔法特性就会导致问题。有可能在主构建文件中声明了一个魔法特性，其值级联沿用至其他的构建文件。换句话说，一个构建的行为可能会由于不太明显的特性声明而变化。这就会带来混乱，并导致难以跟踪的错误。

利用 `javac`，如果不对源代码进行修改，并维护你自己的 Ant 版本（这可能又是你极力避免的），那么你所能做的很有限。在使用 `javac` 的魔法特性时，要很好地为之建立文档，并使你的用户了解到为什么构建文件必须使用某种编译器而不是另一个。在编写你自己的任务时，要尽量避免引用工程级特性。

System.exit()的问题

所有美妙的事物都不免会有瑕疵。Java 程序中的一个美中不足便是对 `System.exit()` 的滥用，这一点很常见。Java 开发人员往往不加修改地沿用 C 的编程模型，并在实现其许多程序时使用 `System.exit()` 来终止执行，`System.exit()` 可能用在一个未处理的错误出现时，也可能出现在用户要求程序终止的时候。`System.exit()` 向系统返回一个错误码（更确切地说，是向 JVM 返回错误码）。传统做法是由 0 表示成功或无错误，而任何非 0 值都意味着失败（有些程序为不同的非 0 值指定了相应的含义）。在此存在的问题是，`System.exit()` 会与 JVM 直接通信，而不论类是如何实例化的，也不论一个程序在调用栈中可能的深度如何。人们错误地认为 Java 程序可以处理退出调用，但实际上却并非如此。退出调用是由 JVM 来处理的，仅此而已。那么一般来讲，这种看似不相干的问题对于 Ant 会有何影响呢？具体地，对你又有何影响呢？

如果由某个任务使用的一个任务或类调用了 `System.exit()`, Ant 引擎会由于其 JVM 终止而终止。由于此影响类似于关掉计算机（其实你是在“关掉”一个虚拟机），构建会不带任何错误消息地停止。要说明的是，构建只是简单地停止了。对于作为任务编写者的你来说，若明知某些类要调用 `System.exit()`，你就应该编写使用这些类的任务（注6）。如果不能避免这种调用，则需要使用 `exec` 或 `java` 任务，或者为自己的任务而借用这些任务的实现。`exec` 和 `java` 会创建新进程（`fork`）完成 Ant JVM 的 JVM 处理，这意味着 `System.exit()` 调用不会在 Ant JVM 内部进行。如果认为需要实现类似于此的功能，可以阅读第七章和附录二中的 `java` 任务和创建新进程（`forking`）。你还可以查看 `java` 任务的类（Java）的源代码。

对 `System.exit()` 的调用可能要负责在构建期间处理一些异常的行为。例如，如果你使用 `java` 来调用在 Internet 上找到的新 XSLT 程序，而在程序执行过程中构建异常终止，那么罪魁祸首可能就是新 XSLT 程序中对 `System.exit()` 的调用。要记住，为将来考虑，不要将 `System.exit()` 当作你的朋友。如果必须要有 `System.exit()`，那么也应该存在于类的 `main()` 方法中。

注 6：除非你能够绝对确保可以完全避免此方法调用。

第六章

用户编写监听者

编写日志是 Ant 所固有的功能。正如你所料，此功能是内置的，而且默认情况下总是打开的。不过你想不到的是，完全可以修改 Ant 编写其日志的方式。实际上，对日志记录机制进行修改还不是全部，你不必拘泥于此，对于 Ant 在构建的某些步骤中的行为，其方式均可以修改。Ant 通过事件模型的形式提供了这种强大的灵活性。熟悉 GUI 开发的人可能以前听说过这个词，因为 GUI 编程库就是将事件模型付诸实际的最常用的库。事件模型的概念很简单。Ant 引擎维护了一组对象，这些对象请求“监听”构建的不同“事件”。在处理过程中，Ant 会以 `BuildEvent` 对象的形式向其各个监听者宣布这些事件。这些监听者相应地称为 `BuildListener`。`BuildListener` 是一个 Java 接口。只要你想编写一个新的监听者，就必须在你的新类中实现 `BuildListener` 接口，并为各个接口方法填入相应的逻辑。

编写自己的类来实现 `BuildListener` 接口是一个很简单的工作，特别是与编写一个 Ant 任务相比，简直是小巫见大巫了。通常的用户编写监听者都是某种形式的特定日志工具 (`logger`)，并以此替代 Ant 的内置日志记录机制。了解到这一点，Ant 的开发人员提供了一个 `BuildLogger` 类，它由 `BuildListener` 派生得到，并增加了特殊的功能从而可以直接写入 Ant 的日志。这是很重要的，因为用户可以在构建时控制 Ant 的输出。默认情况下，Ant 将其日志输出定向至 `stdout`，不过也可以通过命令行选项 `-logfile <filename>` 将日志输出定向到一个日志文件。

如果在编写一个 `BuildLogger` 而不是 `BuildListener`，你的类就可以继承此功能来使用 Ant 的输出，从而使开发人员可以在其构建中更容易地使用你的新类。否则，将要求他们不仅要管理 Ant 的输出，还要管理你的类自己的输出。

要记住，监听者并不仅限于替换 Ant 的日志记录系统。利用监听者，你可以在诸如 Bugzilla 的 bug 跟踪系统中结合 Ant 的功能。例如，为达到这个目的，可以编写一个监听者作为 Ant 和 Bugzilla 之间的一座桥。在这座桥的一端，Ant 的构建事件到达以备处理。桥将解释这些事件，并把它们传递给 Bugzilla，从而发出带有正确参数的适当的 HTTP 请求。这里不是修改日志输出，而是监听者将对 Ant 做出修改，或者更确切地说，是要提高其处理能力。其巧妙之处在于，Ant 或 Bugzilla 均没有显式地设计为要彼此集成。这完全是利用 Ant 的监听者 - 生产者事件系统实现的，而且非常易于使用。

为了提供 `BuildListener` 的一个例子，我们（再次）借用了 Ant 的源代码发布中的内容，并将对 `XmlLogger` 做仔细的分析。正如其名所示，像默认的日志工具一样，此监听者也将写出日志输出，只不过它将输出写为 XML 标记。

BuildEvent 类

Ant 及其所有监听者（这其中也包括相应的日志工具）均使用 `BuildEvent` 类进行通信。Ant 可分派 7 种类型的事件，分别表示 Ant 处理一个构建文件所经历的不同阶段。我们将在下一节描述这些事件。要注意这 7 种类型的事件与任务生命周期并没有关系。

对于在 Ant 及事件监听者之间传递的事件，`BuildEvent` 就相当于一个信息容器。Ant 引擎将重要的信息放入一个 `BuildEvent` 对象中，并将其传递至它的监听者，从而使其监听者了解到有关构建的更多信息。有时，由于在实现和设计上存在的约束，Ant 可能会限制这些对象中的信息量。关于这些约束在哪里出现或为什么出现，并没有规章可循。只是要注意确实存在这样的约束，这样在你编写自己的

监听者时，出现某些情况时就不会过于迷惑不解了，例如，你不必奇怪为什么未能得到一个任务的名字（注 1）。

`getProject()`

返回当前正在运行的构建的 Project 对象。此对象要控制构建的所有方面，因此在使用时要务必小心。

`getTarget()`

对于发送事件时的活动目标，返回与之对应的 Target 对象。

`getTask()`

对于发送事件时的活动任务，返回与之对应的 Task 对象。

下一个方法仅在一个任务、目标或构建完成时才可用：

`getException()`

返回发送事件时抛出的活动 BuildException 异常。这对于栈跟踪尤其有用。

以下方法仅在 Ant 正在记录一个消息时才可用：

`getPriority()`

返回消息的优先级。优先级对应为所记录消息的消息级（在 Project 对象中存储为公共静态字段）。对于日志记录级别，请参见第三章中的总结。

`getMessage()`

返回日志记录的消息内容。不要假设记录消息的代码会以某种方式对文本进行格式化。

你所编写的监听者可以对 Ant 传递的 BuildEvent 对象使用以上方法，从而完成各种功能强大的操作。Project、Target 和 Task 对象使你的监听者可以访问关

注 1：你可以对 Ant 做深入研究，并找出为什么没有得到有关信息。如果是由于有人疏于将它增加到 BuildEvent 对象中，那么就非常需要你来修正这个问题，并将修改提交给 Ant 的维护人员。这正是群体开发过程！

于构建的详细信息。如果需要对构建过程有更多的控制（而不只是 XML 元素所提供的控制），那么任务则尤其适于结合监听者来编写。你可以为你的任务类增加更多的公共方法，而你的监听者则可以使用这些增加的方法来完成附加的功能。

BuildListener 接口

通过其事件框架，Ant 可以使用实现了 BuildListener 的监听者类来跟踪各种构建处理事件。BuildListener 接口的设计及其实现所遵循的模式与监听者的 AWT（注 2）概念非常类似。在这两种模型中，均由一个引擎来传递事件，而不论事件是系统事件还是用户驱动事件。希望接收这些事件的类要注册为监听者（在这种情况下即要作为 Ant 引擎的监听者），而且通常要通过接口类型来限制其希望接收的事件类型。当发生某个事件时，引擎会通知其所有监听者（这些监听者均已注册了当前的事件类型）。使用 BuildEvent 对象，Ant 引擎就可以向监听者传递详细的信息。这种通信模型使 Ant 成为可用的而且更为灵活的构建系统，这是因为它并不要求用户依赖于对 Ant 输出的解析（这种解析是很复杂的）。

以下为各种事件类型及其相应的接口方法。

`buildStarted(BuildEvent event)`

Ant 在开始处理构建文件时要激活 buildStarted 事件。实现此方法的监听者可以在构建开始时完成操作。

`buildFinished(BuildEvent event)`

Ant 在结束处理时要激活 buildFinished 事件。在此事件之后，Ant 引擎将不做任何工作。可以把它作为任何给定构建的最后的消息。

`targetStarted(BuildEvent event)`

Ant 在处理一个目标的第一个任务之前要激活 targetStarted 事件。

注 2：AWT (Abstract Windowing Toolkit, 抽象窗口工具包) 是 Java 的跨平台 GUI 库。当前的 GUI 是使用一种称为事件驱动编程的方法编写的。事件驱动程序不是连续地处理信息，而是在一个特定事件做出要求时才完成相应的操作。

```
targetFinished(BuildEvent event)
```

Ant 在处理了一个目标的最后一个任务之后要激活 targetFinished 事件。无论错误状态如何，Ant 都会激活此事件。

```
taskStarted(BuildEvent event)
```

Ant 在开始一个任务或一个 DataType 的生命期之前要激活 taskStarted 事件。

```
taskFinished(BuildEvent event)
```

Ant 在完成了一个任务或一个 DataType 的生命期之后会立即激活 taskFinished 事件。不管任务或 DataType 的错误状态如何，Ant 都将激活此事件。

```
messageLogged(BuildEvent event)
```

Ant 的任意部分调用了某个日志记录方法后，Ant 都会激活 messageLogged 事件。event 参数包含了来自方法调用的消息及其优先级。

注意：请注意对 taskFinished() 和 taskStarted() 的描述。这些事件的名字有点容易导致误解，因为它们都指示任务。这要追溯到早期版本的 Ant，当时构建文件中的每个元素都被认为是一个任务。因此更适于将这些事件考虑为“elementStarted”和“elementFinished”，这表示 Ant 在处理构建文件中的任何元素（而不仅仅是任务）时会调用这些事件。

只要编写了一个实现 BuildListener 接口的类，你就必须为每个接口方法编写实现。即使你并不打算对某个给定事件做任何操作，这一点仍要保证。如果你不想处理某个事件，可以令其接口方法的实现为空。Ant 仍会调用此方法，只不过什么也不会发生。如果你遇到一个错误或者无法处理事件，Ant 事件模型的设计并不要求重新发送各个事件（注 3）。从理论上说，如果在监听者的处理期间发现了任何错误，则要抛出一个 BuildException 异常。不过，如果你的对象恰好是一个日志工具，那么问题的处理则稍有些不同。由日志工具抛出 BuildException 异常不是一个好的做法。你不能由 messageLogged() 对日志记录系统做出调用。

注 3：这一点与某些事件模型不同，如原来的 Java GUI 库就需要事件处理程序传递消息而不是使用这些信息。

实际上，你的日志工具类将调用其自身，从而产生一种循环的操作，并有可能导致一个无限的循环(假设会不断出现错误)。为了避免循环调用和无限循环的可能性，你的`messageLogged()`方法需要向控制台直接显示错误和调试消息(例如，利用`System.err.println()`调用)，或者将消息发送至其他未涉及Ant的机制。

对于监听者模型的使用，最简单也是最常见的方法是改进或替换Ant自己的日志记录系统。Ant自己的默认日志模块从技术上讲就是一个监听者，即类`org.apache.tools.ant.DefaultLogger`。通过此类，Ant可以得到构建事件，并把消息发送到一个输出流；默认的情况下会将消息发送至标准输出。除非偶尔会使用`System.out.println()`调用将消息直接发送至控制台(编写得不好的任务可能会有这种用法)，Ant总是通过构建事件来生成所有的日志消息。用户编写的日志工具通常将消息重定向为不同的消息格式(如XML)，也可能重定向至不同的审计系统(如Log4J)。

在下一节的例子中，我们将更为详细地分析这样的一个用户编写的日志工具：`XmlLogger`。这个类改进了(而不是替换了)Ant中的默认日志工具。其设计很简单，即得到所有构建事件，并使用每个事件过程中可用的信息来创建一些XML标记。XML输出并不遵循某种模式或设计。此实现更像是一个概念证明而不是一个有用工具(注4)。

一个例子：`XmlLogger`

在Ant的每一版源代码发布中都包括有`XmlLogger`的源代码。如果希望采用此代码，则需要从源代码发布中下载。对于通常到Ant的日志输出，`XmlLogger`做了重定向，并将其以XML标记写入到一个文件中。由于它具有简单性，而且源代码广泛可用，所以可以作为一个学习如何编写构建监听者的很好的例子。

注4： 不过，源代码发布附带有一个样式表，此XML日志工具在其输出中引用了此样式表。利用这个样式表，就有可能将XML输出转换为HTML、SVG或可能想像到(和实现)的任何格式。

如果你想了解 XmlLogger 是如何工作的，可以用你的标准 Ant 安装来加以测试。没有必要下载源代码发布，这是因为 XmlLogger 类在所有二进制发布中都有提供。与增加任务的情况不同，你不必在构建文件中使用一个类似于<taskdef>的标记来声明一个监听者。实际做法是，可在命令行对其声明。首先要保证该类对 Ant 可见，为此可以把它加到系统的类路径中，也可以将其打包到一个 JAR 中，再将此 JAR 放在 ANT_HOME/lib 下。然后，将此监听者类指定为 ant 命令的一个参数。*-listener listenerClass* 参数将通知 Ant 引擎，要求它必须将所指定的监听者类加到内部管理的构建监听者列表中。你可以指定多个监听者参数，对于总数并没有限制。不过，应该说几乎没有限制。对于你的 shell，其命令行字节长度限制对此仍适用。为了使用 XmlLogger 监听者，可以如下调用 ant：

```
ant -listener org.apache.tools.ant.XmlLogger
```

基于一个构建文件运行此命令（及其参数），这将导致 Ant 把构建消息写到控制台以及一个名为 *log.xml* 的 XML 标记文件中。此日志工具将 XML 文件写到当前的工作目录下。

以下代码示例显示了 XmlLogger 的 3 个接口方法的实现：taskStarted()、taskFinished() 和 messageLogged()。这些例子只表示了 XmlLogger 类源代码的一部分。大部分 XML 专有的方法调用和类均未列出，一方面是为是了节省篇幅，另一方面还有一个目的，即避免你对日志工具的组成以及构建 XML 文件的代码由哪些内容组成产生混淆。由于缺少一些代码，因此这里的例子不能编译。不过对于我们要说明的问题，XML 专有的类和方法调用并不重要。

TimedElement 类用于管理 XML 数据（以及你将在下面的代码示例中看到的内容），这是一个私有的静态类，其中封装了一个绝对时间值（long 类）和一个 XML 元素对象（Element 类）。在此不再过于深入，可以把 Element 类考虑为表示一个 XML 元素及其属性（如果可用，还包括其嵌套元素）的对象。以下示例显示了 XmlLogger 的 taskStarted() 方法的代码（省略号表示为清楚起见，相应位置上代码有所省略）：

```
package org.apache.tools.ant;
```

```
import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.apache.tools.ant.util.DOMElementWriter;

/**
 * 利用一个 XML 描述（有关构建中所发生情况）
 * 在当前目录中生成一个“log.xml”文件
 *
 * @see Project#addBuildListener(BuildListener)
 */
public class XmlLogger implements BuildListener {

    ...

    static private class TimedElement {
        long startTime;
        Element element;
    }

    ...

    public void taskStarted(BuildEvent event) {
        // 由 BuildEvent 得到任务对象
        Task task = event.getTask();

        // 创建一个新的<task> XML 元素，且
        // 以当前时间为开始时间，
        // 并带有取自 TASK_TAG 的标签“task”
        TimedElement taskElement = new TimedElement();
        taskElement.startTime = System.currentTimeMillis();
        taskElement.element = doc.createElement(TASK_TAG);

        // 由 task 的类名派生得到任务名
        String name = task.getClass().getName();
        int pos = name.lastIndexOf(".");
        if (pos != -1) {
            name = name.substring(pos + 1);
        }

        // 设置<task>元素的属性，并
        // 将其置于元素栈中
        taskElement.element.setAttribute(NAME_ATTR, name);
        taskElement.element.setAttribute(LOCATION_ATTR, \
            event.getTask().getLocation().toString());
    ...
}
```

当 Ant 调用 XmlLogger 的 taskStarted() 方法时，XmlLogger 要取 BuildEvent 对象，并使用其信息来填写元素的日志 XML 标记（利用一个 TimedElement）。由系统时间，XmlLogger 将填写 TimedElement 的开始时间。后面将在 taskFinished() 中利用此时间来计算当前元素的整个处理时间。XmlLogger 由 BuildEvent 对象（event）检查当前执行的任务名，并查找构建文件中任务的物理位置（即行号）。

在 taskFinished() 中，XmlLogger 使用 event 对象来得到 Ant 刚刚处理完毕的元素的名字。利用此名字，它将在由类所维护的一个元素表中获取已创建的 TimedElement。一旦找到此对象，此日志工具就会计算元素的处理时间，并设置适当的属性。以下为 XmlLogger 的 taskFinished() 方法的代码。同样，在此也省略了一些代码，并用省略号指出：

```
public void taskFinished(BuildEvent event) {
    Task task = event.getTask();
    TimedElement taskElement = (TimedElement)tasks.get(task);
    if (taskElement != null) {
        long totalTime = System.currentTimeMillis() - taskElement.startTime;
        taskElement.element.setAttribute(TIME_ATTR, DefaultLogger.
formatTime(totalTime));
    ...
}
```

下面是 XmlLogger 的 messageLogged() 方法。在调用 messageLogged() 之前，Ant 已经确定了日志级别，而并非由你的日志工具来决定何时显示特定的消息。XmlLogger 的 messageLogged() 方法使用取自 event 对象的级别值从而在标记中设置适当的属性。然后，此方法由 event 对象获取消息，并将其置于一个 CDATA 字段中。因此由日志工具得到的 XML 就表示了构建消息字符串，而且形式仍为其原来的字符格式。

```
public void messageLogged(BuildEvent event) {
    Element messageElement = doc.createElement(MESSAGE_TAG);

    String name = "debug";
    switch(event.getPriority()) {
        case Project.MSG_ERR: name = "error"; break;
        case Project.MSG_WARN: name = "warn"; break;
        case Project.MSG_INFO: name = "info"; break;
```

```
        default: name = "debug"; break;
    }
    messageElement.setAttribute(PRIORITY_ATTR, name);

    Text messageText = doc.createCDATASection(event.getMessage());
    messageElement.appendChild(messageText);

    ...
}
```

消息事件与其他事件稍有区别，这是因为 Ant 引擎并非发出消息事件的惟一来源（而其他构建事件则确实如此）。非消息事件均来自 Project 对象，即在 Project 对象进入或离开一个构建文件的元素时就会发出这些事件。日志消息可能来自除 Project 以外的类。这些消息还可以在 Ant 引擎中传送，从而作为事件传递给 messageLogged()。

并行问题

自 1.4 版本以来，Ant 中都包括了一个特定的任务，它可以以并行方式运行其他任务。在 1.4 版本以前，一个目标中的任务会顺序地执行，在大多数情况下，这完全可以，而且也是我们所预料的。不过，对于某些目标（例如，要编译互斥的代码集或要创建无关的目录的目标），如果这些操作通过线程从而同时执行，那么将大为受益。如果用户的系统有多个 CPU，则会看到通过对这些任务进行并行化所带来的性能提升。并行化还能为希望对应用服务器运行单元测试的人带来好处。其应用服务器和测试必须同时运行，在 1.4 版本以前的 Ant 中，这一点实现起来并不容易。遗憾的是，对于编写或已经编写了定制构建监听者的人来说，并行化却可能会破坏其原来可以正常工作的代码。

有些构建事件监听者依赖于以特定顺序出现的一些事件。例如，对于 javac 任务，如果一个监听者希望在 taskStarted() 事件之后看到 taskFinished() 事件，那么假如两个 javac 任务并行地运行，监听者就会失败或者有奇怪的操作。第二个 javac 可能在第一个之前结束。监听者代码正在监听某个事件，而如果 Ant 先完成了第二个 javac 任务，就可能会过早地触发对于第一个 javac 任务的相应

操作，反之亦然。因此，监听者的输出或操作就是错误的，并可能进一步导致其他的问题。如果给出了一个使用 parallel 任务的构建文件，那么最好对你的定制监听者进行测试，以查看非顺序操作是否正常。

作为一个对并行运行的任务加以处理的监听者，XmlLogger 是一个很好的例子。下面来看 XmlLogger 监听一个构建文件中以下一组操作的执行流程：

```
<parallel>
    <copy todir="test">
        <fileset dir=".\\irssibot-1.0.4" includes="**/*.java"/>
    </copy>
    <mkdir dir="testxml"/>
    <mkdir dir="testxml2"/>
    <copy todir="test">
        <fileset dir=".\\oak-0.99.17" includes="**/*.java"/>
    </copy>
    <mkdir dir="testxml3"/>
</parallel>
```

假设引擎是多线程的，它会以某种方式执行任务，从而使它们按以下的顺序完成：

1. MKDIR(TESTXML)
2. MKDIR(TESTXML2)
3. MKDIR(TESTXML3)
4. COPY(irssibot)
5. COPY(oak)

由于编写时考虑到了对无序事件的处理，XmlLogger 所得到的 XML 标记不会显示出任何无序的元素。任务的标记将按以上顺序列出（并保留其嵌套元素）。对于编写多线程监听者，并没有一种“正确”的方法，因此 XmlLogger 在设计上显示出了明智的预见性，这就可以避免将来可能出现的灾难。即便将来对任务库有巨大调整，这种预见性仍可以使所编写的监听者有很强的生命力。

第七章

核心任务

本章列出了 Ant 1.2、1.3、1.4 和 1.4.1 版本的核心任务和属性。在表示版本时若记为“all”，则表示以上 Ant 版本均支持某一给定功能。Ant 1.1 并未考虑在内，本章中对于仅可以在 Ant 1.1 中工作的任务和属性未做描述。

本章由以下主要小节组成：

任务总结

对 Ant 任务提供了一个扼要的总结。

常用类型和属性

描述了属性类型，其后列出了所有 Ant 任务所用的属性。

工程和目标

描述了 `<project>` 和 `<target>` 元素的语法。

核心任务参考

描述了各个核心 Ant 任务。

每个任务的描述包括以下信息：

- 对该任务的简要总结。

- 支持该任务的 Ant 版本列表。
- 实现此任务的 Java 类的类名。
- 该任务的 XML 属性列表。
- 对所允许内容的描述，可能是嵌套 XML 元素，也可能是文本。
- 使用示例。

任务总结

表 7-1 对 Ant 的所有核心任务做了总结。本章余下的内容将对各个任务进行详细的描述。

表 7-1：核心任务总结

任务名	Ant 版本	含义
ant	all	基于另一个构建文件中的一个目标调用 Ant
antcall	all	调用当前构建文件中的一个目标
antstructure	all	为 Ant 构建文件创建一个 XML DTD (Document Type Definition, 文档类型定义)
apply	1.3, 1.4	基于一组文件执行一个系统命令
available	all	如果某资源可用，则设置一个特性
chmod	all	修改文件和目录的权限（仅限于 Unix 平台）
condition	1.4	如果某个条件为 true，则设置一个特性
copy	all	复制文件和目录
copydir	all	在 Ant 1.2 中已经弃用；而代之使用 copy 任务
copyfile	all	在 Ant 1.2 中已经弃用；而代之使用 copy 任务
cvs	all	执行 CVS (Concurrent Versions System, 并发版本系统) 命令
cvspass	1.4	为一个.cvspass 文件增加一个口令；相当于 CVS 的 login 命令

表 7-1：核心任务总结（续）

任务名	Ant 版本	含义
delete	all	删除文件和目录
deltree	all	在 Ant 1.2 中已经弃用；而代之使用 delete 任务
dependset	1.4	管理文件之间的依赖关系，相对于其资源文件，如果存在过时的目标文件，则删除所有这些目标文件
ear	1.4	构建 EAR (Enterprise Application Archive, 企业应用压缩文件) 文件
echo	all	为 Ant 日志或一个文件编写一个消息
exec	all	执行一个本地系统命令
execon	all	在 Ant 1.4 中已经弃用；而代之使用 apply 任务
fail	all	抛出一个 BuildException 异常，导致当前构建终止
filter	all	为当前工程设置记号过滤器
fixcrlf	all	清除源文件中的特殊字符，如制表符、回车符、换行符以及 EOF 字符
genkey	all	在密钥库 (keystore) 中生成一个密钥
get	all	由一个 URL 得到一个文件
gunzip	all	解压缩一个 GZip 文件
gzip	all	创建一个 GZip 文件
jar	all	创建一个 JAR 文件
java	all	执行一个 Java 类
javac	all	编译 Java 源代码
javadoc	all	运行 JavaDoc 实用工具来生成源代码文档
mail	all	使用 SMTP 发送 email
mkdir	all	创建一个目录
move	all	移动文件和目录

表 7-1：核心任务总结（续）

任务名	Ant 版本	含义
parallel	1.4	在并发线程中执行多个任务
patch	all	对原文件应用一个 diff 文件
pathconvert	1.4	将 Ant 路径转换为平台专有的路径
property	all	设置工程中的特性
record	1.4	记录当前构建处理的输出
rename	all	在 Ant 1.2 中已经弃用；而代之使用 move 任务
replace	all	在一个或多个文件中完成字符串替换
rmic	all	运行 rmic 编译器
sequential	1.4	顺序地执行多个任务；设计此任务是为了结合 parallel 任务使用
signjar	all	执行 javasign 命令行工具
sleep	1.4	将构建暂停一个指定的时间间隔
sql	all	利用 JDBC 执行 SQL 命令
style	all	完成 XSLT 转换
tar	all	创建一个 tar 压缩文件
taskdef	all	为当前工程增加定制任务
touch	all	更新一个或多个文件的时间戳
tstamp	all	设置 DSTAMP、TSTAMP 和 TODAY 特性
typedef	1.4	为当前工程增加一个 DataType
unjar	1.3, 1.4	展开一个 ZIP 文件、WAR 文件或 JAR 文件
untar	all	展开一个 tar 文件
unwar	1.3, 1.4	展开一个 ZIP 文件、WAR 文件或 JAR 文件
unzip	1.3, 1.4	展开一个 ZIP 文件、WAR 文件或 JAR 文件
uptodate	all	对于相关的源文件，如果一个或多个目标文件是最新的，则设置一个特性

表 7-1：核心任务总结（续）

任务名	Ant 版本	含义
war	all	创建一个 WAR (Web Application Archive, Web 应用压缩文件)
zip	all	创建一个 ZIP 文件

常用类型和属性

所有 Ant 任务都采用 XML 编写，例如：

```
<copy file="logo.gif" todir="${builddir}"/>
```

在此例中，file 和 todir 即为属性。其属性值 "logo.gif" 和 "\${builddir}" 分别有特定的数据类型。这一节将对任务属性可允许的数据类型做一个总结，其后将列出对于所有任务常用的属性列表。

XML 属性约定

本章中列出了许多 XML 属性，其形式如下：

attribute_name (*version*, *type*, *required_flag*)

此为属性及其功能的描述。

在此：

attribute_name

为属性名。在为一个任务指定属性时将用此来表示一个属性。

version

指示支持此属性的 Ant 版本。"all" 表示 Ant 1.2 及以上版本。

type

表示一个属性可保存的数据的类型。例如，String 表示一个属性应保存文本数据。请参见表 7-2。

required_flag

表示在使用任务时一个给定属性是否是必要的。如果此标志为一个星号(*)，则请参见该列表后面给出的说明。

属性描述

此为属性及其功能的描述。

表 7-2 总结了本章经常引用的属性类型。无论何种情况，XML 属性的文本都会转换为在此列出的某个基本类型。“描述”列对各种转换如何完成做了描述。“由...实现”列则列出了 Ant 表示这些属性类型所用的 Java 类。

表 7-2: XML 属性类型总结

类型名	由 ... 实现	描述
boolean	N/A	完成不区分大小写的字符串比较，将 on、true 和 yes 转换为 true。所有其他值均转换为 false
Enum	org.apache.tools.ant.types.EnumeratedAttribute	用于某些允许有固定的字符串值集的情况
File	java.io.File	指定某个文件或目录的名字。除非另做说明，否则文件和目录名都是相对于工程基目录的。稍后将描述的 fileset (译注 1) 和 filelist 允许指定多个文件
int、long 等等	N/A	诸如 java.lang.Integer 等标准 Java 类型包装器类会处理此类转换，即将构建文件中的文本转换为基本类型
Path	org.apache.tools.ant.types.Path	最常用于 classpath 和 sourcepath 属性，表示一个用;或;分隔的路径列表。对此在第四章的“path DataType”一节中进行了详细描述

译注 1：原文此处为 Fileset，有误。

表 7-2: XML 属性类型总结 (续)

类型名	由 ... 实现	描述
Reference	org.apache.tools.ant.types.Reference	常用于 refid 属性，并包括对某处定义的一个类型 id 的引用。请参见第七章中的 java 任务，其中显示了如何引用在构建文件中某处定义的类路径
String	java.lang.String	这是 Ant 中最常用的类型。字符串（及其他属性）要服从 XML 属性限制。例如，< 字符必须写为 <；

要理解这个表的含义，可以考虑以下任务：

```
<copy file="src/com/oreilly/ejb/manifest.mf"
      tofile="build/META-INF/manifest.mf"/>
```

对于此 copy 任务，file 和 tofile 属性的类型均为 File。Ant 将 XML 属性值（总是字符数据）转换为 java.io.File 对象。知道这一点很有用，因为你需要为这些参数列出合法的文件名。如果未能做到，那么构建将会失败。现在，我们来看对于所有 Ant 任务均可用的 3 个属性。

常用属性

以下列表描述了各个 Ant 任务均支持的属性。由于这些属性对于每个任务都可用，所以它们仅在此列出一次，而不会在对各个任务的描述中再做重复。

id (all, String, N)

一个任务实例的惟一标识符；与 Reference 类型一同使用。

taskname (all, String, N)

任务实例名，它将在日志输出中显示。

description (all, String, N)

关于任务的注释。

工程和目标

<project>和<target>元素不是任务；不过在每个构建文件中都能看到它们。每个构建文件中必须包含一个<project>元素，该<project>元素包含一个或多个<target>元素。

工程

<project>元素在每个构建文件中都可以看到，而且总是根 XML 元素。它为构建文件指定了一个描述性的名字，还指定了默认目标和基目录。另外它还包括构建文件中的所有<target>。

属性

`basedir (all, File, N)`

基目录，工程中的所有相对路径均由此计算得出。它默认为包括构建文件的目录，而且如果此属性未指定，则可以在如下调用 Ant 时设置 basedir 特性：`ant -Dbasedir=mydirectory target`

`default (all, String, Y)`

指定一个目标，如果在 ant 命令行上未指定目标，则执行该目标。

`name (all, String, N)`

Ant 工程的一个描述性的名字。这个名字用于建立文档，而且在键入 `ant -projecthelp` 时也将显示此名字。

内容

0 到 n 个嵌套<description>元素 (1.4)

定义了工程的一个描述以建立文档。每个<description>元素均包括文本内容。在键入 `ant -projecthelp` 时可追加并显示多个描述。

0 到 n 个嵌套<filelist>元素 (all)

定义工程范围的 filelist，它们可以在整个构建文件中得到引用。对于 filelist DataType 的描述请参见第四章。

0 到 n 个嵌套 <fileset> 元素 (all)

定义工程范围的 `fileset`，它们可以在整个构建文件中得到引用。对于 `fileset` `DataType` 的描述请参见第四章。

0 到 n 个嵌套 <filterset> 元素 (1.4)

定义工程范围的 `filterset`，它们可以在整个构建文件中得到引用。对于 `filterset` `DataType` 的描述请参见第四章。

0 到 n 个嵌套 <mapper> 元素 (1.3, 1.4)

定义工程范围的 `mapper`，它们可以在整个构建文件中得到引用。对于 `mapper` `DataType` 的描述请参见第四章。

0 到 n 个嵌套 <path> 元素 (all)

定义工程范围的 `path`，它们可以在整个构建文件中得到引用。对于 `path` `DataType` 的描述请参见第四章。

0 到 n 个嵌套 <property> 元素 (all)

定义工程范围的特性名 - 值对，更多信息请参见 `property` 任务。

1 到 n 个嵌套 <target> 元素 (all)

定义已命名的任务组，以及目标之间的依赖关系。

0 到 n 个嵌套 <taskdef> 元素 (all)

为工程增加定制任务定义。更多信息请参见 `taskdef` 任务。

目标

每个构建文件都包含一个或多个 `<target>` 元素，而它们又分别包含有任务。任务完成实际的构建工作，而目标则定义依赖关系。这在第三章中已进行了详细的解释。

属性

`depends (all, String, N)`

此目标所依赖的其他目标的列表（用逗号分隔）。所列出的每个目标均在此目标执行前按顺序执行。

description (all, String, N)

此目标的一个描述性的名字。此描述用于建立文档，并在键入 **ant -projecthelp** 时可以显示此名字。

if (all, String, N)

指定一个特性的名字。只有设置了所命名的特性时此目标才执行。

name (all, String, Y)

此目标的名字。此即为用户由命令行执行目标时所用的名字，而且可用于列出目标间的依赖关系。

unless (all, String, N)

指定一个特性的名字。只有设置了所命名的特性，此目标才执行。

内容

目标可以包含嵌套 DataType 和任务。DataType 已在第四章进行了描述。现在，我们来介绍 Ant 的 all 核心任务。

核心任务参考

本章余下的部分将提供 Ant 核心任务的详细信息。

ant

基于另一个构建文件中的一个目标调用 Ant

`org.apache.tools.ant.taskdefs.Ant`

基于另一个构建文件中的一个特定目标调用 Ant。对于大型工程来说这尤其有用，这些大型工程往往将构建过程分解为多个 Ant 构建文件，每一个构建文件分别构建整个应用的一小部分。

all

它将实例化一个新的 Ant 工程（作为 `org.apache.tools.ant.Project` 类的一个实例）。基于不同的 Ant 版本，由调用工程向新工程传递特性的方法也有所不同。在 Ant 1.1 中，调用工程的特性在新工程中是可见的。如果两个工程都定义了同

一个特性，则调用工程优先。如本节后面所示，Ant 1.2 中增加了指定嵌套 `<property>` 元素的能力，而在 Ant 1.4 中还增加了 `inheritall` 属性。

此任务将设置新创建 Project 对象中的 `ant.file` 特性，令其值与调用工程中的相应特性值相等，即为构建文件名。

属性

`antfile (all, String, N)`

要调用的构建文件名。默认为 `build.xml`。

`dir (all, File, N)`

新工程所用的基目录；`antfile` 属性相对于 `dir` 所指定的目录。默认为当前的工作目录。

`inheritall (1.4, boolean, N)`

控制特性如何由当前工程传递到新工程。默认为 `true`，这表示当前工程中的所有特性在新工程中都可用。这也正是 Ant 1.4 以前版本的工作方式。如果设置为 `false`，则当前工程中定义的特性不会传递给新工程，除非它们在 `ant` 命令行上定义（也就是说，作为“用户特性”）。由嵌套 `<property>` 元素显式传递的特性不受此属性影响，这说明较之于被调用者的特性，它们总有更高的优先级。

`output (all, String, N)`

文件名，输出即要写至此文件。

`target (all, String, N)`

新工程中要调用的目标名。如果忽略，则调用新工程的默认目标。

内容

`0 到 n 个嵌套 <property> 元素 (all)`

将一个特性传递到新的构建过程。

使用示例

调用当前目录下 *util_buildfile.xml* 中的默认目标。

```
<ant antfile="util_buildfile.xml"/>
```

调用 *gui* 目录下 *build.xml* 中的 *clean* 目标。

```
<ant dir="gui" target="clean"/>
```

调用另一个构建文件，为 *builddir* 特性传递一个新值。即使该特性在调用构建文件的某处定义，此值仍会被显式地设置为 *utiloutput*。

```
<ant antfile="util_buildfile.xml">
  <property name="builddir" value="utiloutput"/>
</ant>
```

参见

对于嵌套 *<property>* 元素可用的属性，请参见 *property* 任务。

antcall

all

调用当前构建文件中的一个目标

[org.apache.tools.ant.taskdefs.CallTarget](#)

调用当前构建文件中的一个目标。通过使用嵌套 *<param>* 元素将特性传递至新目标。对 Ant 源代码进行分析可以看出，*antcall* 要使用当前构建文件来实例化和调用 *ant* 任务。这说明新的工程实例被创建并且特性的工作与在 *ant* 任务中完全一样。

属性

inheritall (1.4, boolean, N)

定义特性如何传递给新目标。默认为 *true*，这说明当前构建过程中的所有特性都将由新目标继承。在 Ant 1.4 之前，仅能采用此做法。若为 *false*，则用户在命令行所设置的特性是惟一要传递给新目标的特性。

target (all, String, Y)

要调用的目标的名字。

内容

0 到 n 个嵌套 <param> 元素 (all)

将一个特性传递给新构建过程。每个 `<param>` 元素都使用 `property` 任务中所用的同样的类实现；`property` 的所有属性均适用。

使用示例

调用 `cleandir` 目标并指定 `dir-to-clean` 特性：

```
<target name="clean">
  <antcall target="cleandir">
    <param name="dir-to-clean" value="javadocs"/>
  </antcall>
</target>
```

删除由 `dir-to-clean` 特性所指定的一个目录：

```
<target name="cleandir">
  <delete dir="${dir-to-clean}" />
</target>
```

参见

对于嵌套 `<param>` 元素可用的属性，请参见 `property` 任务。

antstructure

all

为 Ant 构建文件创建一个 XML DTD

`org.apache.tools.ant.taskdefs.AntStructure`

为 Ant 构建文件创建一个 XML DTD。它使用 Java 的反射为所有任务确定可允许的属性和内容。由于底层 Ant 任务 API 并未指出所需的属性，所以 DTD 将把所有属性均标志为 #IMPLIED (注 1)。

注 1： 在 DTD 中，#IMPLIED 表示可选。

属性

`output (all, File, Y)`

要生成的 DTD 文件的文件名。

内容

无。

使用示例

在当前目录中创建 `project.dtd`:

```
<target name="createdtd">
  <antstructure output="project.dtd"/>
</target>
```

apply

1.3, 1.4

执行一个系统命令

org.apache.tools.ant.taskdefs.Transform

执行一个系统命令。对于 Ant 1.4，已经废弃的 `execon` 任务只是 `apply` 的一个别名。与 `exec` 任务有所不同，此任务需要一个嵌套 `<fileset>` 来指定一个或多个文件和目录作为命令的参数。

属性

`dest (1.3, 1.4, File, *)`

由命令所生成的任何目标文件的目标目录。

`dir (1.3, 1.4, File, N)`

命令的工作目录。

`executable (1.3, 1.4, String, Y)`

要执行的命令名。不包括命令行参数。

`failonerror (1.3, 1.4, boolean, N)`

如果为 `true`, 若命令返回了非 0 值, 则构建失败。默认为 `false`。

`newenvironment (1.3, 1.4, boolean, N)`

如果为 `true`, 则不将现有环境变量传递至新过程。默认为 `false`。

`os (1.3, 1.4, String, N)`

此任务可应用的操作系统的列表。相对于 `System.getProperty("os.name")` 的返回值, 仅当列表中包括有与之匹配的一个字符串时才会执行。

`output (1.3, 1.4, File, N)`

一个文件, 命令输出即要重定向到该文件。

`outputproperty (1.4, String, N)`

存储命令输出的特性的名字。

`parallel (1.3, 1.4, boolean, N)`

如果为 `true`, 则命令仅执行一次, 将所有文件传递为参数。如果为 `false`, 此命令将为各个文件分别执行一次。默认为 `false`。

`skipemptyfilesets (1.4, boolean, N)`

如果为 `true`, 如果未找到源文件, 或者源文件比目标文件更新, 则不执行命令。默认为 `false`。

`timeout (1.3, 1.4, int, N)`

终止命令之前所等待的毫秒数。如果未指定则无限期等待。

`type (1.3, 1.4, Enum, N)`

确定无格式文件或目录的名字是否发送至命令。可允许的值为 `file`、`dir` 或 `both`。默认为 `file`。

`vmlauncher (1.4, boolean, N)`

指定是否试图使用 JVM 的内置命令启动程序, 而不是一个 `antRun` 脚本。默认认为 `true`。

如果指定一个嵌套 `<mapper>`, 则 `dest` 是必要的。

内容

0 到 n 个嵌套 <arg> 元素 (1.3, 1.4)

定义命令行参数。

0 到 n 个嵌套 <env> 元素 (1.3, 1.4)

指定环境变量以传递至命令。

1 到 n 嵌套 <fileset> 元素 (1.3, 1.4)

指定哪些文件和目录要作为参数传递至命令。除非指定了 <srcfile> 元素，否则文件将追加到命令行末尾。

0 或 1 个嵌套 <mapper> 元素 (1.3, 1.4)

若定义，则将目标文件的时间戳与源文件的时间戳加以比较。

0 或 1 个嵌套 <srcfile> 元素 (1.3, 1.4)

若存在，则对由 <fileset> 元素所指定的文件要置于命令行何处加以控制。

<srcfile> 元素并没有任何属性，而且要置于合适的 <arg> 元素之间。

0 或 1 个嵌套 <targetfile> 元素 (1.3, 1.4)

此元素仅在指定了一个 <mapper> 元素和 destdir 属性时才适用。它没有任何属性，并用于标记命令行上目标文件名的位置。其工作与 <srcfile> 元素相同。

使用示例

如果运行 Windows 2000，使用 type 命令显示 build.xml 的内容：

```
<!-- Set vm launcher="false", otherwise this fails when using
     JDK 1.4beta1 on Windows 2000 -->
<apply executable="type" vm launcher="false" os="Windows 2000">
  <fileset dir=".">
    <include name="build.xml"/>
  </fileset>
</apply>
```

参见

对于执行系统命令的另一种方法，请参见 exec 任务，特别是如果你不希望向命令

传递一个文件名列表时，这种方法更合适。有关 `<arg>`、`<env>`、`<fileset>` 和 `<mapper>` 的更多信息请参见第四章。

available

all

如果某资源可用，则有条件地设置一个特性 `org.apache.tools.ant.taskdefs.Available` 在运行时如果某资源可用，则有条件地设置一个特性。此资源可以是一个类、文件、目录或 Java 系统资源。如果资源存在，此特性则置为 `true`，或者置为可选的 `value` 属性所设置的值。否则，不设置此特性。

属性

`classname (all, String, *)`

要查找的一个 Java 类的类名，如 `com.oreilly.book.Author`。

`classpath (all, Path, N)`

在查找一个类名或资源时所用的类路径。

`classpathref (all, Reference, N)`

在构建文件某处定义的一个类路径的引用。

`file (all, File, *)`

要查找的一个文件的文件名。

`filepath (1.4, Path, N)`

文件的路径。

`property (all, String, Y)`

当找到资源时，此任务所设置的特性的名字。

`resource (all, String, *)`

要查找的一个 Java 资源。对于资源组成的更多信息，请参见 `java.lang.ClassLoader` 中的各种 `getResource()` 方法。

`type (1.4, String, N)`

指定 `file` 属性所表示的含义。在 Ant 1.4 中，合法的值为 `"file"` 或 `"dir"`。如果未指定，则 `file` 属性既可能表示文件，也可能表示目录。

value (all, String, N)

当找到资源时，为特性所赋的值。默认为 "true"。

classname、file 或 resource 中必取其一。

内容

0 或 1 个嵌套 <classpath> 元素 (all)

代替 classpath 属性的 Path 元素。

0 或 1 个嵌套 <filepath> 元素 (1.4)

代替 filepath 属性的 Path 元素。

使用示例

如果类路径上有 Java servlet API 2.3 版本（或以上版本），以下例子将把 `Servlet23.present` 特性设置为 true：

```
<available classname="javax.servlet.ServletRequestWrapper"
           property="Servlet23.present"/>
```

它将正常工作，因为 `javax.servlet.ServletRequestWrapper` 类并未包括在 servlet API 以前的版本中。

chmod

修改文件权限

all

org.apache.tools.ant.taskdefs.Chmod

修改一个或多个文件的权限，类似于 Unix 的 `chmod` 命令。此任务仅在 Unix 平台上工作。

属性

defaultexcludes (all, boolean, N)

确定是否使用“默认排除模式”(default excludes)，如第四章“fileset DataType”一节所述。默认为 true。

`dir (all, File, *)`

保存其权限待修改的文件的目录。

`excludes (all, String, N)`

要排除的文件模式的列表（用逗号分隔）。这些文件模式是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件的文件名。

`file (all, File, *)`

要修改权限的文件或目录的名字。

`includes (all, String, N)`

要包含的文件模式的列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`parallel (all, boolean, N)`

如果为 `true`, 则使用一个 `chmod` 命令即修改所有文件的权限。默认为 `true`。

`perm (all, String, Y)`

要应用的新权限，如 `g+w`。

`type (all, Enum, N)`

确定无格式文件或目录名是否发送至命令。可允许的值为 `file`、`dir` 或 `both`。

默认为 `file`。

仅需在 `dir` 或 `file` 中指定其一，或者至少指定一个嵌套 `<fileset>` 元素。

内容

0 到 n 个嵌套 `patternset` 元素: `<exclude>`、`<include>`、`<patternset>` (`all`);
`<excludesfile>`、`<includesfile>` (1.4)

代替其相应属性，它们指定了所包含和所排除的源文件组。

0 到 n 个嵌套 `<fileset>` 元素 (*all*)

指定哪些文件和目录要作为参数传递给命令。

使用示例

对于 JavaDoc 输出树中的所有 HTML 文件，将其权限修改为只读 (444)：

```
<chmod perm="444">
  <fileset dir="${javadocs}">
    <include name="**/*.html"/>
  </fileset>
</chmod>
```

condition

1.4

如果条件为 `true` 则设置一个特性 [org.apache.tools.ant.taskdefs.ConditionTask](#)

如果某个条件为 `true`，则设置一个特性。此任务结合了基本布尔表达式以及 `available` 和 `uptodate` 任务。

属性

`property (1.4, String, Y)`

当条件为 `true` 时，所要设置的特性的特性名。如果条件为 `false`，则不设置相应特性。

`value (1.4, String, N)`

当条件为 `true` 时，为特性所赋的值。默认为 `true`。

内容

以下元素均被认为是条件，在此任务中，必须直接嵌套一个且仅一个条件。而这些条件可能进一步包含如下所列的其他嵌套条件。

`<not>`

仅包含一个嵌套条件，否定其结果。没有任何属性。

<and>

包含任意个嵌套条件，只有当所有嵌套条件均为 true 时才计算为 true。条件的计算是从左至右进行的，而且如果一个条件计算为 false，则整个计算停止（注 2）。没有任何属性。

<or>

包含任意个嵌套条件，如果任何嵌套条件为 true，则计算为 true。条件的计算是从左至右进行的，而且如果一个条件计算为 true，整个计算则停止（注 3）。没有任何属性。

<available>

相当于 available 任务，只不过忽略其 property 和 value 属性。

<uptodate>

相当于 uptodate 任务，只不过忽略其 property 和 value 属性。

<os>

如果当前操作系统为给定类型，则计算为 true。此元素有一个可选的 family 属性，其类型为 String。合法值为 windows、dos、mac 和 unix。dos 属性包括 OS/2 和 Windows 系统。若 family 未指定，则此条件计算为 false。

<equals>

如果两个 String 相等，则计算为 true。不允许嵌套条件。这两个 String 利用 arg1 和 arg2 必要属性来指定。

使用示例

如果类路径上有 servlet API 2.3 版本和 JAXP 1.1 版本，而且 Java 属于以上所列版本，那么此例将把 Environment.configured 特性设置为 true。

```
<condition property="Environment.configured">
  <and>
    <!-- test for servlet version 2.3 -->
```

注 2：与 Java 的 `&&` 操作符作用相当。

注 3：与 Java 的 `||` 操作符作用相当。

```
<available classname="javax.servlet.ServletRequestWrapper"/>
<!-- test for JAXP 1.1 -->
<available classname="javax.xml.transform.TransformerFactory"/>
<or>
    <equals arg1="${java.version}" arg2="1.3.0"/>
    <equals arg1="${java.version}" arg2="1.4.0-beta"/>
    <equals arg1="${java.version}" arg2="1.4.0"/>
</or>
</and>
</condition>
```

整个例子相当于: (servlet 2.3 可用) AND (JAXP 1.1 可用) AND ((Java=1.3.0) OR (Java=1.4.0-beta) OR (Java=1.4.0))。

copy	all
复制文件和目录	org.apache.tools.ant.taskdefs.Copy

将文件和目录复制到新位置。当目标文件不存在或源文件比目标文件更新时，文件才会被复制。

属性

file (all, File, *)

指定要复制的一个文件。使用嵌套 `<fileset>` 来复制多个文件。

filtering (all, boolean, N)

如果为 `true`，将使用某个全局构建文件过滤器进行记号过滤（请参见 `filter` 任务）。使用 `<filterset>` 指定的嵌套过滤器总是可用，而不论此属性如何。默认为 `false`。

flatten (all, boolean, N)

如果为 `true`，则不保留源文件的目录结构，所有文件都将复制到一个目标目录。使用一个嵌套 `<mapper>` 也可以得到同样的结果。默认为 `false`。

includeemptydirs (all, boolean, N)

如果为 `true`，则空目录也将被复制。默认为 `true`。

overwrite (all, boolean, N)

如果为 `true`，则即使目标文件更新也将复制文件。默认为 `false`。

`preservelastmodified (1.3, 1.4, String, N)`

如果为 `true`, 则目标文件将指定为与源文件相同的最近修改时间戳。默认为 `false`。

`todir (all, File, *)`

文件将被复制到的目标目录。

`tofile (all, File, *)`

目标文件, 仅当使用 `file` 属性复制一个文件时可用。

要么必须设置 `file` 属性, 要么至少指定一个嵌套 `<fileset>`。若设置了 `file` 属性, 则 `todir` 或 `tofile` 属性必须设置其一。若使用了嵌套 `<fileset>` 元素, 则只允许设置 `todir`。

内容

0 到 n 个嵌套 <fileset> 元素 (all)

选择要复制的文件。若存在 `<fileset>`, 则必须有 `todir` 属性。

0 到 n 个嵌套 <filterset> 元素 (1.4)

定义文件复制时用于文本替换的记号过滤器。更多信息请参见 `filter` 任务。

0 或 1 个嵌套 <mapper> 元素 (1.3, 1.4)

定义在复制时文件名如何转换。默认情况下, 将完成一个身份转换, 这说明文件名不会被修改。

使用示例

此例将所有 Java 源文件复制到一个新目录, 并将所有 @VERSION@ 的出现替换为 `app.version` 的值。

```
<copy todir="${builddir}/srccopy">
  <fileset dir="${srcdir}">
    <include name="**/*.java"/>
  </fileset>

  <filterset>
```

```
<filter token="VERSION" value="${app.version}" />
</filterset>
</copy>
```

参见

有关 `<fileset>` 和 `<mapper>` 的更多信息请参见第四章。关于 `<filterset>` 元素和记号过滤的信息请参见 `filter` 任务。

copydir

此任务在 Ant 1.2 中已经弃用，而代之使用 `copy` 任务。

copyfile

此任务在 Ant 1.2 中已经弃用，而代之使用 `copy` 任务。

CVS

all

执行 CVS 命令

org.apache.tools.ant.taskdefs.Cvs

执行 CVS 命令。CVS 是一个开源版本控制系统，可在 <http://www.cvshome.org> 处得到。

有关 CVS 的信息，请参见 Gregor N. Purdy 所著的《CVS Pocket Reference》(O'Reilly)。

属性

`command (all, String, N)`

CVS 命令名，默认为 `checkout`。

`cvsroot (all, String, N)`

指定存储库所在位置，相当于 `CVSROOT` 环境变量。

`date (all, String, N)`

指定此命令可应用于指定日期之前(包括指定日期)的文件,相当于-D CVS选项。

`dest (all, File, N)`

指定将写出的文件置于何处。默认为工程的基目录。

`error (all, File, N)`

用于记录来自CVS命令的标准错误输出的文件。默认为使用MSG_WARN日志级别的Ant日志。

`noexec (all, boolean, N)`

若为true,则不对文件系统做任何修改,相当于-n CVS选项,默认为false。

`output (all, File, N)`

用于记录来自CVS命令的标准输出的文件。默认为使用MSG_INFO日志级别的Ant日志。

`package (all, String, N)`

指定要访问的CVS模块。

`passfile (1.4, File, N)`

CVS口令文件的文件名,默认为~/.cvspass。

`port (1.4, int, N)`

CVS同一个服务器通信所用的端口号,默认为2401。

`quiet (all, boolean, N)`

若为true,CVS输出将不显示。相当于-q CVS选项。默认为false。

`tag (all, String, N)`

指定一个CVS标签名。相当于-tag CVS选项。

内容

无。

使用示例

这个简单例子显示了 CVS 的版本：

```
<cvs command="-version"/>
```

以下示例将所有文件用标签 release1.1 写入到 *antbook* 模块中，并将所写入的文件置于 \${builddir} 指定的目录中。由此可以得到提示，即 CVS 如何能够重新构建以前版本的软件包：

```
<cvs dest="${builddir}"  
      cvsroot=":local:C:\cvsrepository\cvsroot"  
      tag="release1.1"  
      package="antbook"/>
```

参见

cvspass 任务。

cvspass

1.4

更新 .cvspass 文件

org.apache.tools.ant.taskdefs.CVSPass

更新 *.cvspass* 文件。相当于执行 *cvs login* 命令。

属性

cvsroot (1.4, String, Y)

指定存储库的位置。相当于 CVSROOT 环境变量。

passfile (1.4, File, N)

指定口令文件的文件名。默认为 *user.home* 目录中的 *.cvspass* 文件。

password (1.4, String, Y)

要增加的口令。

内容

无。

使用示例

此例向当前用户的主目录中的 `.cvspass` 文件增加 `anttester` 口令。

```
<cvspass cvsroot=":local:C:\cvsrepository\cvsroot" password="anttester" />
```

参见

`cvs` 任务。

delete

`all`

删除文件和目录

`org.apache.tools.ant.taskdefs.Delete`

删除一个或多个文件和目录。

这是 Ant 中最危险的任务。利用一个 `<delete dir=". "/>` 标签你就能够轻松地将整个工程全部删除。

属性

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式，如第四章“fileset DataType”一节所述。默认为 `true`。

`dir (all, File, *)`

待删除的目录，包括其所有文件和子目录。有些奇怪的是，此属性与 `file` 属性或嵌套 `<fileset>` 无关。特别是，它未确定 `file` 属性所指定的文件见于哪个目录，实际上，此属性通知任务“蛮力地”删除一棵完整的目录树。

`excludes (all, String, N)`

要排除的文件模式的列表（用逗号分隔）。它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件的文件名。

`failonerror (1.4, boolean, N)`

如果为 `true`，此任务若失败，则构建过程失败。默认为 `true`。

`file (all, File, *)`

要删除的文件的文件名。

`includeemptydirs (1.3, 1.4, boolean, N)`

如果为 `true`, 即使目录为空也将被删除。仅在使用嵌套 `<fileset>` 时才有关系。默认为 `false`。

`includes (all, String, N)`

要包含的文件模式的列表 (用逗号分隔)。

`includesfile (all, File, N)`

每行包括一个包含模式的文件名。

`quiet (1.3, 1.4, boolean, N)`

如果为 `true`, 若某个文件或目录不能被删除也不会失败。默认为 `false`。

`verbose (all, boolean, N)`

若为 `true`, 则在文件删除时显示其名字。默认为 `false`。

`dir` 或 `file` 至少要取其一, 或者要有一个嵌套 `<fileset>`。

内容

0 到 n 个嵌套 `patternset` 元素: `<exclude>`、`<include>`、`<patternset> (all)`;
`<excludesfile>`、`<includesfile> (1.4)`

代替其相应属性, 它们指定了所包含和排除的源文件组。

0 到 n 个嵌套 `<fileset>` 元素 (all)

选择要删除的文件。若 `includeemptydirs=true`, 则仅删除空目录。

使用示例

以下为在每个 Ant 构建文件中几乎都能看到的常用目标。它将删除目录及其所有内容:

```
<target name="clean" description="Remove all generated code">
  <delete dir="${builddir}" />
</target>
```

deltree

此任务在 Ant 1.2 中已经弃用，而代之使用 `delete` 任务。

dependset

1.4

管理依赖关系

org.apache.tools.ant.taskdefs.DependSet

管理文件之间的依赖关系，相对于一组源文件，如果某些目标文件更老，则删除所有这些目标文件。此任务不会完成一个按位置、逐文件的时间戳比较。实际上，它将由一组源文件中取最新的时间戳，并将它与所有目标文件中的最新的时间戳进行比较。

属性

无。

内容

至少需要一个`<srcfileset>`或`<srcfilelist>`, 以及至少一个`<targetfileset>`或`<targetfilelist>`。如果所缺少的文件并不重要，则使用 `fileset` 元素。相反，若使用 `filelist`，缺少任何文件都会导致所有目标文件被删除。

0 到 n 个嵌套 `<srcfileset>` 元素 (1.4)

此 `fileset` 中的所有文件都将与`<targetfileset>`和`<targetfilelist>`元素所指定的所有文件相比较。

0 到 n 个嵌套 `<srcfilelist>` 元素 (1.4)

此 `filelist` 中的所有文件都将与`<targetfileset>`和`<targetfilelist>`元素所指定的所有文件相比较。

0 到 n 个嵌套 `<targetfileset>` 元素 (1.4)

此 `fileset` 中的所有文件都将与`<srcfileset>`和`<srcfilelist>`元素所指定的所有文件相比较。如果存在更老的文件，则所有文件都将被删除。

0 到 n 个嵌套<targetfilelist>元素 (1.4)

此 filelist 中的所有文件都将与<srcfileset>和<srcfilelist>元素所指定的所有文件相比较。如果存在更老的文件，则所有文件都将被删除。

使用示例

如果 Ant 构建文件或任何.java 文件比任何.class 更新，此例将删除构建目录中的所有.class 文件。

```
<dependset>
  <srcfileset dir="${basedir}" includes="build.xml"/>
  <srcfileset dir="${srcdir}" includes="**/*.java"/>
  <targetfileset dir="${builddir}" includes="**/*.class"/>
</dependset>
```

参见

第四章所描述的 fileset 和 filelist 类型。

ear

1.4

创建 EAR 文件

org.apache.tools.ant.taskdefs.Ear

创建 EAR 文件。尽管 jar 任务也能创建 EAR 文件，但 ear 任务可以简化此过程。EAR 文件为 J2EE 应用的部署机制，而且比 JAR（其中包括有定义良好的目录和文件）要稍多些内容。

属性

appxml (1.4, File, Y)

指定部署描述文件的位置，通常部署文件都重命名为所生成 EAR 文件中的 *META-INF/application.xml*。源文件不必命名为 *application.xml*。

basedir (1.4, File, N)

指定基目录，由此将文件增加到 EAR 文件中。

`compress (1.4, boolean, N)`

如果为 `true`, 则压缩 EAR 文件。默认为 `true`。

`defaultexcludes (1.4, boolean, N)`

确定是否使用默认排除模式, 如第四章的“fileset DataType”一节所述。默认为 `true`。

`earfile (1.4, File, Y)`

指定要创建的 EAR 文件的文件名。

`encoding (1.4, String, N)`

指定 EAR 文件中文件名的字符编码, 默认为 UTF-8。Ant 规范警告称, 修改此属性可能会导致得到 Java 无法使用的 EAR 文件。

`excludes (1.4, String, N)`

要排除的文件模式列表 (用逗号分隔), 它们是对默认排除模式的补充。

`excludesfile (1.4, File, N)`

每行包括一个排除模式的文件的文件名。

`filesonly (1.4, boolean, N)`

如果为 `true`, 则不创建空目录, 默认为 `false`。

`includes (1.4, String, N)`

要包含的文件模式的列表 (用逗号分隔)。

`includesfile (1.4, File, N)`

每行包括一个包含模式的文件的文件名。

`manifest (1.4, File, N)`

要使用的清单文件的文件名。

`update (1.4, boolean, N)`

如果为 `true`, 则做出修改时更新现有 EAR 文件, 而不是将其删除再从头创建。默认为 `false`。

`whenempty (1.4, Enum, N)`

无文件匹配时所采取的操作。合法值为 `fail`(终止构建)、`skip`(不创建EAR

文件) 或 create。默认为 create, 这说明无任何文件时创建一个空的 EAR 文件。

内容

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all);
<excludesfile>、<includesfile> (1.4)

代替其相应属性, 它们指定了所包含和排除的源文件组。

0 或 1 个嵌套 <metainf> 元素 (1.4)

定义一个 fileset, 其中包括置于 EAR 文件的 META-INF 目录中的所有文件。如果找到一个名为 MANIFEST.MF 的文件, 则忽略它并发出一个警告。

0 到 n 个嵌套 <fileset> 元素 (1.4)

指定要包含在 EAR 文件中的文件和目录。

0 到 n 个嵌套 <zipfileset> 元素 (1.4)

有关更多信息请参见 zip 任务的文档。

使用示例

以下两个例子将得到同样的结果。第一个例子使用属性:

```
<ear earfile="${builddir}/myapp.ear"
      appxml="ear_deploy_descriptor/application.xml"
      basedir="${builddir}"
      includes="*.jar,*.war"/>
```

下例则使用了嵌套 <fileset> 以代替 basedir 和 includes 属性:

```
<ear earfile="${builddir}/myapp2.ear"
      appxml="ear_deploy_descriptor/application.xml">
    <fileset dir="${builddir}" includes="*.jar,*.war"/>
</ear>
```

参见

请参见 jar 任务。ear 的实现类由 jar 的实现类扩展得到。

echo	all
写至一个日志或文件	org.apache.tools.ant.taskdefs.Echo
将一个消息写至 Ant 日志或一个文件。显示级别默认为 Project.MSG_WARN，表示消息显示在控制台上。	

属性

append (*all, boolean, N*)

如果为 true，则追加到一个现有文件。默认为 false。

file (*all, File, N*)

消息所要写至的文件的文件名。

message (*all, String, **)

要写的文本。

除非文本作为 XML 标签的内容包括在内，否则必须有 message 属性，如以下示例所示。

内容

文本内容 (*all*)

如果未指定 message 属性，则允许文本内容。也允许诸如 \${builddir} 的特性引用。

使用示例

以下例子中的第一个通过使用 message 属性来指定要写的文本。第二个例子则通过将文本放在 <echo>...</echo> 标签之间来指定要写的文本。

```
<echo message="Building to ${builddir}" />
<echo>You are using version ${java.version}
of Java! This message spans two lines.</echo>
```

exec	all
执行一个系统命令	org.apache.tools.ant.taskdefs.ExecTask

执行一个系统命令，类似于 apply 任务，它提供了一种方法来访问 Java 和 Ant 构建环境之外的本地功能。

apply 任务需要一个嵌套 <fileset>，以此指定作为参数传递给系统命令的一个文件和目录列表。exec 任务与此有些区别，它不允许此嵌套 <fileset>。

属性

command (*1.1, CommandLine, **)

要执行的命令，包括参数。在 Ant 1.2 中已经废弃。

dir (*all, File, N*)

命令的工作目录。

executable (*all, String, **)

要执行的命令名。不包括命令行参数。

failonerror (*all, boolean, N*)

如果为 true，则当命令返回非 0 值时构建失败。默认为 false。

newenvironment (*1.3, 1.4, boolean, N*)

如果为 true，则现有环境变量不传递给新过程。默认为 false。

os (*all, String, N*)

任务所应用的操作系统列表。仅当列表中包括与 `System.getProperty("os.name")` 的返回值匹配的一个字符串时才会执行。

output (*all, File, N*)

一个文件，命令输出即要重定向到此文件。

outputproperty (*1.4, String, N*)

存储命令输出的一个特性的名字。

timeout (*all, int, N*)

命令停止前需要等待的毫秒数。如果未指定则无限期等待。

vmlauncher (1.4, boolean, N)

指定是否使用 JVM 的内置命令启动程序，而不是一个 *antRun* 脚本。默认为 true。

从理论上说，只能设置一个 command 或 executable。由于自 Ant 1.2 以后 command 已经废弃，因此推荐使用 executable。

内容**0 到 n 个嵌套 <arg> 元素 (all)**

每个元素指定一个命令行参数，如第四章所述。

0 到 n 个嵌套 <env> 元素 (all)

每个元素指定一个环境变量。

使用示例

在 Windows 2000 平台上，在构建目录中执行 *dir /b*:

```
<exec executable="dir" dir="${builddir}"
      vmlauncher="false" os="Windows 2000">
  <arg line="/b" />
</exec>
```

参见

请参见 apply 任务。命令行参数和环境变量的语法已在第四章进行了描述。

execon

1.2, 1.3 (1.4 中已弃用)

执行一个系统命令

`org.apache.tools.ant.taskdefs.ExecuteOn`

执行一个系统命令。此任务在 Ant 1.4 中已经废弃；而代之使用 apply 任务。

属性

`dir (1.2, 1.3, File, N)`

命令的工作目录。

`executable (1.2, 1.3, String, Y)`

要执行的命令名。不包括命令行参数。

`failonerror (1.2, 1.3, boolean, N)`

如果为 `true`, 则当命令返回非 0 值时构建失败。默认为 `false`。

`newenvironment (1.3, boolean, N)`

如果为 `true`, 则现有环境变量不传递给新过程。默认为 `false`。

`os (1.2, 1.3, String, N)`

任务所应用的操作系统列表。仅当列表中包括与 `System.getProperty("os.name")` 的返回值匹配的一个字符串时才会执行。

`output (1.2, 1.3, File, N)`

一个文件, 命令输出即要重定向到此文件。

`parallel (1.2, 1.3, boolean, N)`

如果为 `true`, 此命令仅执行一次, 将所有文件作为参数传递。如果为 `false`, 则为每个文件执行一次命令。默认为 `false`。

`timeout (1.2, 1.3, int, N)`

命令停止前需要等待的毫秒数。如果未指定则无限期等待。

`type (1.2, 1.3, Enum, N)`

确定无格式文件或目录名是否发送至命令。可允许的值为 `file`、`dir` 或 `both`。默认为 `file`。

内容

请参见 `apply` 任务。

使用示例

请参见 apply 任务。

fail

all

抛出一个异常并终止当前构建

org.apache.tools.ant.taskdefs.Exit

抛出一个 BuildException 异常，导致当前构建失败。

属性

message (*all, String, N*)

指定执行此任务时显示的消息。

内容

文本内容 (1.4)

Ant 1.4 增加了指定嵌套文本的能力。在消息跨多行时这一点非常有用。

使用示例

以下例子终止构建时不带任何描述性消息：

```
<fail/>
```

在这种情况下，Ant 显示以下消息，在此 104 为调用 fail 的构建文件行的行号：

```
BUILD FAILED  
C:\cvssdata\ant\mysamples\build.xml:104: No message
```

以下对 fail 的调用将得到正在显示的一条消息。此消息在 <fail> 和 </fail> 标签间指定。

```
<fail>Java version ${java.version} is not allowed!</fail>
```

下一个例子得到和上例相同的结果；惟一的区别在于，此消息是通过 message 属性指定的。

```
<fail message="Java version ${java.version} is not allowed!"/>
```

参见

使用 echo 任务来写消息而不终止构建。

filter

all

定义记号过滤器

org.apache.tools.ant.taskdefs.Filter

定义记号过滤器。它们用于在复制文件时完成文本替换（即记号过滤）。在 Ant 1.2 和 1.3 中，记号总是形如 @token@。Ant 1.4 则增加了使用一个字符的能力，而不是采用 @ 再加 <filterset> 元素的形式。过滤器不应与二进制文件一同使用。

<filter> 元素可以出现在目标中，或者作为各种复制文件任务中的一个嵌套元素。

属性

filtersfile (all, File, *)

包括记号 / 值对的一个文件，格式化为 Java 特性文件。

token (all, String, *)

源文件中要替换的文本，不包括 @ 字符。

value (all, String, *)

要取代 @token@ 的文本。不保留 @ 字符。

必须指定 filtersfile 属性，或者同时指定 token 和 value。

内容

无。

使用示例

先从以下源文件开始:

```
// %COPYRIGHT!  
  
/**  
 * @version @VERSION@  
 */  
public class Hello {  
    ...  
}
```

我们希望将 %COPYRIGHT! 替换为一个版本提示，并将 @VERSION@ 替换为正确的版本号。以下是一个构建文件中的一个目标，它将完成这一工作：

```
<target name="tokenFilterDemo" depends="prepare">  
    <filter token="VERSION" value="1.0"/>  
    <copy todir="build" filtering="true">  
        <!-- select files to copy -->  
        <fileset dir="src">  
            <include name="**/*.java"/>  
        </fileset>  
        <filterset begintoken "%" endtoken="!">  
            <filter token="COPYRIGHT"  
                value="Copyright (C) 2002 O'Reilly"/>  
        </filterset>  
    </copy>  
</target>
```

第一个<filter>元素负责在复制文件时将 @VERSION@ 替换为 1.0。为此，copy 任务的 filtering 属性必须置为 true。

<filterset>元素是在 Ant 1.4 中新增加的，这对于 %COPYRIGHT! 记号是必需的，这是因为它未将 @ 字符用作分隔符。利用<filterset>，我们可以使用所需的任何记号。<filterset>元素可能包括一个或多个<filter>元素，因此可以将多个<filter>都列为内容。

以下为复制之后的文件：

```
// Copyright (C) 2002 O'Reilly
```

```
/**  
 * @version 1.0  
 */  
public class Hello {  
    ...  
}
```

参见

请参见 copy 任务。

fixcrlf

all

清除特殊字符

org.apache.tools.ant.taskdefs.FixCRLF

清除源文件中的特殊字符，如制表符、回车符、换行符和 EOF 字符。

属性

cr (*all, Enum, N*)

在 Ant 1.4 中已经废弃。指定 CR 字符如何修改。合法的值为 add、asis 和 remove。在 Unix 平台上，默认为 remove，将 Windows 形式的 CRLF 转换为 LF。在 Windows 平台上，默认为 add，将 Unix 形式的 LF 字符转换为 CRLF。

defaultexcludes (*all, boolean, N*)

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。
默认为 true。

destdir (*all, File, N*)

指定“调整”文件置于何处。如果未指定，源文件将被重写。

eof (*all, Enum, N*)

指定如何处理 DOS 形式的 EOF 字符 (Ctrl-Z)。与 cr 属性类似，也支持同样的属性和默认值。若为默认的 remove，则将存在的 EOF 字符删除。若为 add，那么在必要时增加一个 EOF 字符。若为 asis，则什么也不做。

eol (1.4, *Enum, N*)

代替已经废弃的 cr 属性，为 Macintosh 提供更好的支持。合法的值为 asis、

cr、lf 和 crlf。以上各值均指定在“调整”文件中要放置什么 EOL 字符。
在 Unix 中默认为 lf，在 Macintosh 中默认为 cr，而在 Windows 中默认为 crlf。

excludes (all, String, N)

要排除的文件模式列表（用逗号分隔），这些模式是对默认排除模式的补充。

excludesfile (all, File, N)

每行包括一个排除模式的文件的文件名。

includes (all, String, N)

要包含的文件模式的列表（用逗号分隔）。

includesfile (all, File, N)

每行包括一个包含模式的文件的文件名。

javafiles (1.4, boolean, N)

如果为 true，表示嵌套 <fileset> 指定了一组 Java 文件。这样可以确保 Java 字符串和字符常量中的制表符不会被修改。默认为 false。

srcdir (all, File, Y)

包含要调整文件的目录。

tab (all, Enum, N)

控制如何修改制表符。合法的值为 add、asis 和 remove。默认为 asis，这说明，制表符要被保留。若为 add，连续的空格将转换为制表符。若为 remove，则会将制表符转换为空格。

tablength (all, int, N)

一个制表符所表示的空格数。合法的值为 2~80，并包括 2 和 80。默认为 8。

内容

0 到 n 个嵌套 patternset 元素：<exclude>、<include>、<patternset> (all);
<excludesfile>、<includesfile> (1.4)

代替其相应属性，它们指定了所包含和排除的源文件组。

使用示例

以下示例将Java源文件中的制表符转换为4个连续的空格。它保留了现有的EOL和EOF字符：

```
<fixcrlf srcdir="${srcdir}"
           destdir="${builddir}"
           eol="asis"
           tab="remove"
           tablength="4"
           eof="asis"
           includes="**/*.java"
           javafiles="true"/>
```

genkey

all

生成一个密钥对

org.apache.tools.ant.taskdefs.GenerateKey

生成一个密钥对，将它们增加到一个密钥库（keystore）文件中。它实际上就是 *keytool -genkey* 命令的包装器。*keytool* 应用包含在 JDK 中，并管理私钥（private）和公共证书。

属性

alias (all, String, Y)

新密钥库项的身份。

dname (all, String, *)

X.500 标识名及别名。

keyalg (all, String, N)

用于生成此项的算法。

keypass (all, String, *)

用于保护私钥的口令。

keysize (all, String, N)

所生成密钥的大小。

keystore (all, String, N)

密钥库文件名。默认为用户主目录中的 *.keystore*。

sigalg (all, String, N)

用于对证书签名的算法。

storepass (all, String, Y)

用于保护密钥库的口令。

storetype (all, String, N)

密钥库类型。

validity (all, String, N)

所生成证书合法的天数。

verbose (all, boolean, N)

显示模式。默认为 `false`。

仅在未指定 `<dnname>` 内容时才需要 `dnname` 属性。如果私钥的口令与密钥库口令不同，则需要 `keypass`。

内容

0 或 1 个嵌套 `<dnname>` 元素 (all)

可选地使用以代替 `dnname` 属性。包括 0 到 n 个嵌套的 `<param>` 元素，如下例所示。

使用示例

以下示例生成一个新的密钥库项：

```
<genkey dname="CN=Eric Burke, OU=Authors, O=O'Reilly,
          L=Sebastopol, S=California, C=US"
        alias="ericb"
        storepass="aidansdaddy" />
```

下面的例子使用一个嵌套 `<dnname>` 元素完成了同样的任务：

```
<genkey alias="ericb" storepass="aidansdaddy">
  <dbname>
    <param name="CN" value="Eric Burke"/>
    <param name="OU" value="Authors"/>
    <param name="O" value="O'Reilly"/>
    <param name="L" value="Sebastopol"/>
    <param name="S" value="California"/>
    <param name="C" value="US"/>
  </dbname>
</genkey>
```

参见

请参见 Sun 公司的 Java 开发工具包对于密钥库命令行程序的有关文档。

get

由 URL 获取一个文件

all

org.apache.tools.ant.taskdefs.Get

由某 URL 获取一个文件。

属性

dest (all, File, Y)

存储文件所用的本地名。

ignoreerrors (all, boolean, N)

如果为 true，则记录错误但不终止构建。默认为 false。

src (all, URL, Y)

要获取的远程文件的 URL。

usetimestamp (all, boolean, N)

如果为 true，则仅下载时间戳比本地文件新的远程文件。仅用于 HTTP 协议。当此文件下载时，其时间戳设置为远程机器上的时间戳。默认为 false。

verbose (all, boolean, N)

若为 true，每获取 100KB 数据即显示一个 “.”。默认为 false。

内容

无。

使用示例

得到 O'Reilly 的主页:

```
<get src="http://www.oreilly.com/" dest="${builddir}/oreilly_home.html"/>
```

如果在一个防火墙之后，在运行 Ant 之前要使用 ANT_OPTS 环境变量指定代理服务器配置（第二章已做解释）。

gunzip

all

解压缩一个 GZip 文件

org.apache.tools.ant.taskdefs.GUnzip

展开一个 GZip 文件。只有当目标文件不存在或比源文件老时才展开此文件。

属性

dest (all, String, N)

目标文件或目录名。如果忽略，则 dest 默认为包含源文件的目录。若 dest 为一个目录，则目标文件名为 src 名，并去掉任何 .gz 扩展名。

src (all, String, Y)

待解压缩的文件名。

内容

无。

使用示例

将 *manuscript.tar.gz* 展开为同一目录下的 *manuscript.tar*:

```
<gunzip src="manuscript.tar.gz"/>
```

将 *manuscript.tar.gz* 展开为 \${builddir}/*manuscript.tar*:

```
<gunzip src="manuscript.tar.gz" dest="${builddir}"/>
```

对 tar 文件解压缩后，要使用 untar 任务将其展开。

参见

gzip 任务以及 untar 任务。

gzip

all

创建一个 GZip 文件

org.apache.tools.ant.taskdefs.GZip

创建一个 GZip 压缩文件。

属性

src (all, File, Y)

要压缩的文件的文件名。

zipfile (all, File, Y)

要创建的文件的文件名。

内容

无。

使用示例

将 *manuscript.tar* 压缩为 *manuscript.tar.gz*:

```
<gzip src="manuscript.tar" dest="manuscript.tar.gz"/>
```

参见

`gunzip` 任务。

jar	all
创建一个 JAR 文件	<code>org.apache.tools.ant.taskdefs.Jar</code>

由一个或多个源文件和目录创建一个 JAR 文件。

属性

`basedir (all, File, N)`

包括要增加到 JAR 文件中的文件的基目录。

`compress (all, boolean, N)`

如果为 `true`, 则压缩 JAR 文件。默认为 `true`。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式, 如第四章中的“fileset DataType”一节所述。

默认为 `true`。

`encoding (1.4, String, N)`

指定 JAR 文件中的文件名字符编码。默认为 UTF-8。Ant 规范警告称, 修改此属性可能会导致得到 Java 无法使用的 JAR 文件。

`excludes (all, String, N)`

要排除的文件模式列表 (用逗号分隔), 它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`filesonly (1.4, boolean, N)`

如果为 `true`, 则不创建空目录。默认为 `false`。

`includes (all, String, N)`

要包含的文件模式的列表 (用逗号分隔)。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`jarfile (all, File, Y)`

要创建的 JAR 文件的文件名。

`manifest (all, File, N)`

要置于 JAR 文件中的一个现有清单文件的文件名。如果未指定，Ant 会生成一个新的清单文件，其中包含有所用 Ant 的版本。

`update (1.4, boolean, N)`

如果为 `true`，则发生修改时更新现有 JAR 文件，而不是将其删除并从头重新创建。默认为 `false`。

`whenempty (all, Enum, N)`

未找到输入文件时所采取的操作。默认为 `create`。合法值为：

`fail`

终止构建。

`skip`

不创建 JAR 文件。

`create`

若无文件，则创建一个空的 JAR 文件。

内容

`0 到 n 个嵌套 <attribute> 元素 (1.4)`

每个元素指定一个名 - 值，对置于 JAR 文件的清单中“未命名”的段中。清单中各段用空行分隔，并可以有相应的名字。使用 `<section>` 元素来创建命名的清单段。以下为 `<attribute>` 嵌套元素可允许的属性。

`name (1.4, String, Y)`

属性名。

`value (1.4, String, Y)`

属性值。

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all); <excludesfile>、<includesfile> (1.4)

代替其相应属性，它们指定了所包含和排除的源文件组。

0 到 n 个嵌套 <fileset> 元素 (all)

指定包括在 JAR 文件中的文件和目录。

0 或 1 个嵌套 <metainf> 元素 (1.4)

定义一个 fileset，其中包括置于 JAR 文件中 *META-INF* 目录下的所有文件。如果在此 fileset 中找到一个名为 *MANIFEST.MF* 的文件，则其内容将与置于所生成 JAR 文件中的 *MANIFEST.MF* 合并。

0 到 n 个嵌套 <section> 元素 (1.4)

每个元素定义一个命名的清单段。每个 *<section>* 可以包括 0 个或多个嵌套 *<attribute>* 元素。*<section>* 元素需要以下属性：

`name (1.4, String, Y)`

段名。

0 到 n 个嵌套 <zipfileset> 元素 (1.3, 1.4)

更多信息请参见 zip 任务的文档。

使用示例

在构建的目录树中创建包含所有 *.class* 文件的 *sample.jar*:

```
<jar jarfile="\${builddir}/sample.jar"
      basedir="\${builddir}"
      includes="**/*.class"/>
```

此例完成同样的工作，不过使用了一个嵌套 *<fileset>* 元素而不是 *includes* 属性：

```
<jar jarfile="\${builddir}/sample2.jar">
  <fileset dir="\${builddir}" includes="**/*.class"/>
</jar>
```

最后一个例子显示了如何使用 Ant 1.4 的 `<section>` 和 `<attribute>` 元素来创建 JAR 文件的清单：

```
<jar jarfile="build/sample.jar" basedir="src" includes="**/*.java">
  <manifest>
    <attribute name="Version" value="3.2"/>
    <attribute name="Release-Date" value="20 Mar 2002"/>
    <section name="drinks">
      <attribute name="favoriteSoda" value="Coca Cola"/>
      <attribute name="favoriteBeer" value="Amber Bock"/>
    </section>
    <section name="snacks">
      <attribute name="cookie" value="chocolateChip"/>
      <attribute name="iceCream" value="mooseTracks"/>
    </section>
  </manifest>
</jar>
```

以下为所得到的 *META-INF/MANIFEST.MF* 文件：

```
Manifest-Version: 1.0
Release-Date: 20 Mar 2002
Version: 3.2
Created-By: Ant 1.4.1
```

```
Name: snacks
cookie: chocolateChip
iceCream: mooseTracks
```

```
Name: drinks
favoriteBeer: Amber Bock
favoriteSoda: Coca Cola
```

参见

`unzip` 任务。

java

执行一个 Java 类

all

`org.apache.tools.ant.taskdefs.Java`

使用 Ant 的 VM 实例或通过创建一个新的 VM 进程来执行一个 Java 类。如果所执行的应用调用了 `System.exit()`，要确保设置 `fork="true"`，否则 Ant 将退出。

属性

`args (all, String, N)`

Ant 1.2 中已经弃用；而代之使用嵌套 `<arg>` 元素。

`classname (all, String, *)`

要执行的 Java 类的类名。

`classpath (all, Path, N)`

要使用的类路径。除非 `fork="true"`，否则它将增加到 Ant 的类路径中。

`classpathref (all, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`dir (all, File, N)`

VM 的工作目录。除非 `fork="true"`，否则它将被忽略。

`failonerror (all, boolean, N)`

如果为 `true`，则当命令返回非 0 值时构建失败。默认为 `false`。除非 `fork="true"`，否则它将被忽略。

`fork (all, boolean, N)`

如果为 `true`，类将在一个新的 VM 实例中执行。默认为 `false`。

`jar (1.4, File, *)`

要执行的一个可执行 JAR 文件的文件名。此 JAR 文件必须包括一个 Main-Class 清单项，而且 `fork` 必须为 `true`。

`jvm (all, String, N)`

Java 解释器的命令名（可能是命令的一个完全路径名）。默认为 `java`。除非 `fork="true"`，否则它将被忽略。

`jvmargs (all, String, N)`

Ant 1.2 中已经弃用；而代之以使用嵌套 `<jvmarg>` 元素。

`maxmemory (all, String, N)`

为新创建的 VM 分配的最大内存量。除非 `fork="true"`，否则它将被忽略。

相当于 Java 命令行选项 `-mx` 或 `-Xmx`，这取决于所用的 Java 版本。

`output (1.3, 1.4, File, N)`

一个文件名，输出即写至此文件。

`classname` 或 `jar` 二者必取其一。

内容

0 到 n 个嵌套 <arg> 和 <jvmarg> 元素 (all)

分别指定应用和 JVM 的命令行参数。请参见第四章中的“`argument DataType`”一节。

0 到 n 个嵌套 <sysproperty> 元素 (all)

每个元素指定一个系统特性。

0 或 1 个嵌套 <classpath> 元素 (all)

使用 `path` 元素以代替 `classpath` 或 `classpathref` 属性。

使用示例

此例显示了如何将不同命令行参数传递到一个应用：

```
<java classname="com.oreilly.antbook.JavaTest">
  <sysproperty key="oreilly.home" value="${builddir}" />
  <arg value="Eric Burke"/>
  <arg line="-verbose -debug"/>
  <arg path="/home;/index.html"/>
  <classpath>
    <pathelement path="${builddir}" />
  </classpath>
</java>
```

首先，指定了 `oreilly.home` 系统特性。这就相当于调用了以下命令：

```
java -Doreilly.home=build etc...
```

另外，还指定了以下 4 个命令行参数：

- Eric Burke

- -verbose
- -debug
- C:\home;C:\index.html (注 4)

下面的例子显示了如何引用在 Ant 构建文件某处定义的一个类路径:

```
<!-- this is defined at the "target level", parallel to <target>s -->
<path id="thirdparty.class.path">
    <pathelement path="lib/crimson.jar"/>
    <pathelement path="lib/jaxp.jar"/>
    <pathelement path="lib/xalan.jar"/>
</path>

<target name="rundemo">
    <java classname="com.oreilly.antbook.JavaTest">
        <classpath refid="thirdparty.class.path"/>
    </java>
</target>
```

javac	all
编译 Java 源代码	org.apache.tools.ant.taskdefs.Javac

编译 Java 源代码。此任务将 *.java* 文件与 *.class* 文件进行比较。若不存在类文件或源文件比其相应的类文件更新，那么有关的源文件将得到编译。

此任务对于分析源代码或完成逻辑依赖关系分析不做任何工作。例如，如果对于一个基类的源代码已经修改，而在此之后其子类需要编译，则 Ant 不会知道这一点。

支持多种编译器。对于 JDK 1.1/1.2，默认编译器为 *classic*。对于 JDK 1.3/1.4，默认为 *modern*。要选择一个不同的编译器，可如表 7-3 所示设置 *build.compiler* 特性。“别名”列列出了可供选择的特性值，它与“特性”列中所列的值有同样的效果。

注 4： 注意命令行如何转换为平台专有的路径名，这在第四章已做讨论。

表 7-3：编译器选择特性

特性	别名	描述
classic	javac1.1 或 javac1.2	标准 JDK 1.1 或 1.2 编译器
modern	javac1.3 或 javac1.4	标准 JDK 1.3 或 1.4 编译器
jikes		IBM 的 Jikes 编译器
jvc	Microsoft	Microsoft 的 Java SDK 编译器
kjc		kopi 编译器
gcj		gcc 的 gcj 编译器
sj	Symantec	Symantec 编译器
extJavac		在其自己的 JVM 中运行 modern 或 classic

属性

`bootclasspath (all, Path, N)`

要使用的启动（bootstrap）（注 5）类路径。

`bootclasspathref (all, Reference, N)`

对构建文件中某处定义的一个启动类路径的引用。

`classpath (all, Path, N)`

要使用的类路径。除非 `fork="true"`，否则它将增加到 Ant 的类路径中。

`classpathref (all, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`debug (all, boolean, N)`

如果为 `true`，则编译源代码时带有调试信息。默认为 `false`。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。

默认为 `true`。

注 5： 在使用 Sun 公司的 JVM 时，启动类路径包括实现 Java 2 平台的那些类。它们可见于 `jre/lib` 目录中的 `rt.jar` 和 `i18n.jar` 文件。

`depend (all, boolean, N)`

如果为 `true`, 则对于支持依赖关系检查的编译器 (如 `jikes` 和 `classic`), 启用依赖关系检查。默认为 `false`。

`deprecation (all, boolean, N)`

如果为 `true`, 则显示废弃警告。默认为 `false`。

`destdir (all, File, N)`

类文件的目标目录。

`encoding (all, String, N)`

源文件的字符编码。

`excludes (all, String, N)`

要排除的文件模式列表 (用逗号分隔), 它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`extdirs (all, Path, N)`

覆盖 Java 安装可选包的一般位置。

`failonerror (1.3, 1.4, boolean, N)`

如果为 `true`, 则当出现错误时构建失败。默认为 `true`。

`fork (1.4, boolean, N)`

如果为 `true`, 则将 Java 编译器执行为一个单独的进程。若得到设置, 此属性将覆盖 `build.compiler` 特性, 而且 Ant 执行 `JAVA_HOME/bin` 中实际的 Java 可执行程序, 而不是编译器的 Main 类。默认为 `false`。

`includeantruntime (1.3, 1.4, boolean, N)`

如果为 `true`, 则将 Ant 运行时库包括在类路径中。默认为 `true`。

`includejavaruntime (1.3, 1.4, boolean, N)`

如果为 `true`, 则包括来自正在执行的 VM 的默认运行时库。默认为 `false`。

`includes (all, String, N)`

要包含的文件模式的列表 (用逗号分隔)。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`memoryinitialsize (1.4, String, N)`

仅当 `fork=true` 时正常工作。为 VM 指定初始内存大小，例如 64000000、
64000k 或 64m。

`memorymaximumsize (1.4, String, N)`

仅当 `fork=true` 时正常工作。为 VM 指定最大内存大小。

`nowarn (1.4, boolean, N)`

如果为 `true`，则将 `-nowarn` 开关传递给编译器。默认为 `false`。

`optimize (all, boolean, N)`

如果为 `true`，则指示编译器对代码进行优化。默认为 `false`。

`source (1.4.1, String, N)`

如果指定，此属性的文本将作为 `-source` 命令行选项传递给底层的 `javac` 可
执行程序。合法的值为 1.3 和 1.4。若传递 1.4，则允许 JDK 1.4 使用其新声
明的功能。

`srcdir (all, Path, *)`

源代码文件的位置。

`target (all, String, N)`

为一个特定的 VM 版本（如 1.1 或 1.2）生成类文件。

`verbose (all, boolean, N)`

如果为 `true`，则指示编译器生成显示输出。默认为 `false`。

除非指定了嵌套 `<src>` 元素，否则 `srcdir` 属性是必要的。

内容

0 到 n 个嵌套 `patternset` 元素：`<exclude>`、`<include>`、`<patternset>` (`all`)；
`<excludesfile>`、`<includesfile>` (1.4)

代替其相应属性，它们指定了所包含和排除的源文件组。

0 到 n 个嵌套 path 元素: <bootclasspath>、<classpath>、<extdirs> 和 <src> (all)

用以代替其相应的属性。

使用示例

编译 com.oreilly.antbook 包和子包中的所有 Java 源文件，将结果置于 \${builddir} 中：

```
<javac srcdir="${srcdir}"
       destdir="${builddir}"
       includes="com/oreilly/antbook/**"/>
```

javadoc

all

调用 javadoc 实用程序

org.apache.tools.ant.taskdefs.Javadoc

调用 javadoc 实用程序。不同于其他 Ant 任务，此任务不做依赖关系分析，因此每次使用都会生成全部文档。

对于不支持的属性，老版本的 javadoc 只是予以忽略。

属性

access (1.4, Enum, N)

取值为 public、protected、package 或 private 之一。默认为 protected，这说明所有保护型和公共类和成员都包括在输出中。它们直接对应为 JavaDoc 的 -public、-protected、-package 和 -private 命令行标志。

additionalparam (all, String, N)

JavaDoc 命令行的附加参数。对于需要引号的参数要使用 "。

author (all, boolean, N)

如果为 true，则包括 @author 标志。默认为 true。

bootclasspath (all, Path, N)

要使用的启动类路径。

`bootclasspathref (all, Reference, N)`

在构建文件中某处定义的启动类路径的引用。

`bottom (all, String, N)`

每页底部所包括的 HTML。

`charset (all, String, N)`

用于跨平台查看所生成文档的字符集。

`classpath (all, Path, N)`

要使用的类路径。

`classpathref (all, Reference, N)`

在构建文件中某处定义的类路径的引用。

`defaultexcludes (1.4, boolean, N)`

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。

默认为 true。

`destdir (all, File, *)`

所生成文档的目标目录。

`docencoding (all, String, N)`

输出字符编码名，例如，“UTF-8”。

`doclet (all, String, N)`

一个定制 doclet 的类名。它对应于 JavaDoc 的 -doclet 参数。

`docletpath (all, Path, N)`

定制 doclet 的类路径。

`docletpathref (all, Reference, N)`

在构建文件中某处定义的一个 doclet 类路径的引用。

`doctitle (all, String, N)`

包括在包索引页中的 HTML。

`encoding (all, String, N)`

源文件的字符编码。

`excludepackagenames (1.4, String, N)`

要排除的包列表（用逗号分隔）。

`extdirs (all, String, N)`

覆盖 Java 安装可选包的一般位置。

`failonerror (all, boolean, N)`

如果为 `true`，则当命令返回非 0 值时构建失败。默认为 `false`。

`footer (all, String, N)`

包括在每个生成页页脚中的 HTML。

`group (all, String, N)`

在一个总览页面中的组专有的包。此属性指定为一个用逗号分隔的字符串。

每一项包括 HTML 页的标题，其后是一个空格，再后面是一组用冒号分隔的 Java 包名。它遵循 JavaDoc 的 `-group` 命令行参数所指定的语法。

`header (all, String, N)`

包括在每个生成页页眉中的 HTML。

`helpfile (all, File, N)`

帮助链接所链接的文件。

`link (all, String, N)`

创建指向给定 URL 上 JavaDoc 输出的链接。

`linkoffline (all, String, N)`

用空格分隔的两个 URL。使用第二个 URL 上的包列表链接到第一个 URL 上的文档。

`locale (all, String, N)`

要使用的场所名，如 `en_US`。

`maxmemory (all, String, N)`

Java VM 可用的最大的堆大小。

`nodeprecated (all, boolean, N)`

如果为 `true`，则不包括 `@deprecated` 标签。默认为 `false`。

`nodeprecatedlist (all, boolean, N)`

如果为 true，则不包括废弃列表。默认为 false。

`nohelp (all, boolean, N)`

如果为 true，则不生成一个帮助链接。默认为 false。

`noindex (all, boolean, N)`

如果为 true，则不生成索引页。默认为 false。

`nonavbar (all, boolean, N)`

如果为 true，则不生成导航条。默认为 false。

`notree (all, boolean, N)`

如果为 true，则不生成类层次结构。默认为 false。

`old (all, boolean, N)`

如果为 true，则模仿 JDK 1.1 doclet。默认为 false。

`overview (all, File, N)`

包含 HTML 总览文档的文件的文件名。

`package (all, boolean, N)`

如果为 true，则显示包类和成员。默认为 false。

`packagelist (all, String, N)`

包含要处理的包的文件的文件名。

`packagenames (all, String, *)`

用逗号分隔的包名列表，如 com.foo.* , com.bar.*。

`private (all, boolean, N)`

如果为 true，则显示私有类和成员。默认为 false。

`protected (all, boolean, N)`

如果为 true，则显示保护型的类和成员。默认为 true。

`public (all, boolean, N)`

如果为 true，则只显示公共的类和成员。默认为 false。

`serialwarn (all, boolean, N)`

如果为 true，则生成有关 @serial 标签的警告。默认为 false。

`sourcefiles (all, String, *)`

用逗号分隔的源文件列表。

`sourcepath (all, Path, *)`

源代码文件位置。

`sourcepathref (all, Reference, *)`

在某处定义的一个源路径的引用。

`splitindex (all, boolean, N)`

如果为 true，则对 JavaDoc 索引页进行分解，对应每个字母分别有一个 HTML 页。默认为 false。

`stylesheetfile (all, File, N)`

要使用的一个 CSS 文件的文件名。

`use (all, boolean, N)`

如果为 true，则创建类和包使用页。默认为 false。

`useexternalfile (1.4, boolean, N)`

如果为 true，则在执行 javadoc 命令之前，将源文件名和包名写到一个临时文件中，从而使命令行更短一些。默认为 false。

`verbose (all, boolean, N)`

默认为 false。

`version (all, boolean, N)`

如果为 true，则包括 @version 标签。默认为 true。

`windowtitle (all, String, N)`

指定 HTML 页标题。

内容

0 到 n 个嵌套 path 元素: <bootclasspath>、<classpath> 和 <sourcepath> (1.3, 1.4)

用以代替其相应属性。

0 或 1 个嵌套元素, 其中包括 HTML: <bottom>、<doctitle>、<footer> 和 <header> (1.4)

用以代替其相应属性。

0 或 1 个嵌套 <doclet> 元素 (1.3, 1.4)

引用一个定制 doclet。对于 <doclet> 元素支持以下属性:

name (all, String, Y)

doclet 的类名。

path (all, Path, N)

doclet 的类路径。

pathref (all, Reference, N)

在构建文件中某处定义的类路径的引用。

<doclet> 接受任意个嵌套 <param> 元素。它们有 name 和 value 属性, 并用于将命令行参数传递给 doclet。例如:

```
<doclet name="MyDoclet" path="${mydoclet.path}">
  <param name="-loglevel" value="verbose"/>
  <param name="-outputdir" value="${mydoclet.output}"/>
</doclet>
```

0 到 n 个嵌套 <excludepackage> 和 <package> 元素 (1.4)

各元素分别用以代替 excludepackagenames 和 packagenames 属性指定的列表中的一项。例如:

```
<package name="com.oreilly.util.*"/>
<excludepackage name="com.oreilly.test.*"/>
```

0 到 n 个嵌套 <group> 元素 (1.3, 1.4)

用以代替 group 属性。以下为合法的 <group> 元素属性。

`title (all, String, *)`

组标题。

`packages (all, String, *)`

包括在组中的包的列表（用冒号分隔）。

`<group>` 支持 0 到 n 个嵌套 `<title>` 和 `<package>` 元素。它们可以用以代替其相应属性。

0 到 n 个嵌套 <link> 元素 (1.3, 1.4)

用以代替 `link` 属性。以下为合法的 `<link>` 元素属性。

`href (all, String, Y)`

外部文档要链接到的 URL。

`offline (all, boolean, N)`

如果为 `true`, 则生成 JavaDoc 时链接不可用。默认为 `false`。

`packagelistloc (all, File, *)`

包含包列表文件的目录位置。

如果 `offline=true`, 则 `packagelistloc` 是必要的。

0 到 n 个嵌套 <source> 元素 (1.4)

每个元素用以代替 `sourcefiles` 属性所指定列表中的一项。以下为合法的 `<source>` 元素属性:

`file (all, File, Y)`

要建立文档的源文件。

使用示例

此例在 `docs` 目录中创建文档。该目录必须已经存在, 否则构建将失败。这个例子还显示了如何利用 `<bottom>` 元素来包括 HTML 内容。对于 `<doctitle>`、`<footer>` 和 `<header>` 也可采用同样的技术。

```
<javadoc excludepackagenames="com.oreilly.test.*"
          destdir="docs"
```

```
    windowtitle="My Documentation">
<package name="com.oreilly.antbook.*"/>
<package name="com.oreilly.util.*"/>
<sourcepath location="${srcdir}"/>
<classpath location="${builddir}"/>
<bottom>
  <![CDATA[<em>Copyright (C) 2001, O'Reilly</em>
  <br>All Rights Reserved]]>
</bottom>
</javadoc>
```

mail

all

发送 email

org.apache.tools.ant.taskdefs.SendEmail

发送 SMTP email。

属性

files (*all, String, **)

用逗号分隔的文件名列表。每一项指定要加入到邮件体中的一个文本。

from (*all, String, Y*)

发送者的 email 地址。

mailhost (*all, String, N*)

邮件服务器主机名。

message (*all, String, **)

消息内容。

subject (*all, String, N*)

邮件主题。

tolist (*all, String, Y*)

用逗号分隔的接收者 email 地址列表。

files 或 **message** 必取其一。

内容

无。

使用示例

将构建结果作为 email 发送：

```
<property name="my.mailhost" value="mail.oreilly.com"/>

<mail from="ant@foobar.com"
      tolist="developers@foobar.com"
      subject="Build Results"
      mailhost="${my.mailhost}"
      files="buildlog.txt"/>
```

参见

若需要二进制附件，则要使用第八章所列出的 `mimemail` 可选任务。

mkdir

all

创建一个目录

`org.apache.tools.ant.taskdefs.Mkdir`

如果某目录不存在，则创建该目录。如果需要还会创建其父目录。

属性

`dir (all, File, Y)`

要创建的目录。

内容

无。

使用示例

此任务通常用于一个其他目标所依赖的prepare目标。这样可以保证在执行其他目标之前，必要的目标目录已经创建。

```
<target name="prepare">
  <mkdir dir="${builddir}"/>
  <mkdir dir="${deploydir}/docs"/>
</target>
<target name="compile" depends="prepare"> ...
</target>
```

参见

关于删除文件和目录的信息请参见 delete 任务。

move

all

移动文件和目录

org.apache.tools.ant.taskdefs.Move

移动一个或多个文件和目录。

属性

file (*all, File, **)

指定要移动的一个文件。要移动多个文件和目录，则要使用嵌套<fileset>。

filtering (*all, boolean, N*)

如果为true，将使用某些全局构建文件过滤器进行记号过滤。无论此属性如何，嵌套过滤器均可用。默认为false。

flatten (*all, boolean, N*)

如果为true，则源文件的目录结构不保留，将所有文件移动到一个目标目录下。一个嵌套<mapper>也能得到同样的结果。默认为false。

includeemptydirs (*all, boolean, N*)

如果为true，则空目录也将移动。默认为true。

`overwrite (all, boolean, N)`

如果为 `true`, 则即使目标文件更新, 文件也会移动。默认为 `false`。

`todir (all, File, *)`

文件将移动至的目录。

`tofile (all, File, *)`

要移动到的文件。

必须有一个 `file` 或一个嵌套 `fileset` 元素。若使用 `file` 属性, 则 `tofile` 或 `todir` 是必要的。若使用嵌套 `fileset`, 则仅允许 `todir`, 而这也是必要的。

内容

`0 到 n 个嵌套 <fileset> 元素 (all)`

选择要移动的文件。若存在 `<fileset>`, 则 `todir` 属性是必要的。

`0 到 n 个嵌套 <filterset> 元素 (1.4)`

定义文件移动时用于文本替换的记号过滤器。有关的更多信息请参见 `filter` 任务。

`0 或 1 个嵌套 <mapper> 元素 (1.3, 1.4)`

定义文件移动时文件名如何转换。默认情况下, 将完成一种身份转换, 这说明文件名不被修改。

使用示例

将所有 `.class` 文件移至新位置:

```
<move todir="${builddir}/foo">
  <!-- the files to move -->
  <fileset dir="${builddir}">
    <include name="**/*.class"/>
  </fileset>
</move>
```

参见

请参见 copy 任务。

parallel

1.4

包含嵌套任务

org.apache.tools.ant.taskdefs.Parallel

这是其他任务的一个容器。其所包含的每一个任务都在它自己的线程中执行，从而可以潜在地改善整个构建的性能。主构建过程将阻塞，直到所有嵌套任务完成为止。如果任何嵌套任务失败，则在所有线程完成时，parallel 任务也会失败。

只有所包含的任务彼此无关时才应使用 parallel。例如，不要将一个代码生成器与一个试图编译所生成代码的任务并行执行。除非你对多线程概念很熟悉，否则要避免使用此任务。

sequential 任务与 parallel 结合使用，可以顺序地执行多组任务。

属性

无。

内容

任何任务，也包括嵌套 parallel 任务。

使用示例

在此例中，应用的客户和服务器部分彼此无关，而且可以同时编译。不过，在编译客户之前，要复制一些关键的文件，而且要使用一个定制的 Java 程序来生成代码。当在 <sequential> 任务中完成这些工作时，服务器代码再同时进行编译。

```
<parallel>
  <sequential>
    <!-- copy some critical files first... -->
    <copy ... />
```

```

<!-- run a code generator -->
<java ... />

<!-- now compile the client code -->
<javac srcdir="${client_srcdir}"
       destdir="${client_builddir}"
       includes="com/oreilly/client/**" />
</sequential>

<!-- compile the server code in parallel with everything
     contained in the <sequential> task -->
<javac srcdir="${server_srcdir}"
       destdir="${server_builddir}"
       includes="com/acme/server/**" />
</parallel>

```

参见

`sequential` 任务。

patch

`all`

应用一个 `diff` 文件

`org.apache.tools.ant.taskdefs.Patch`

对源文件应用一个 `diff` 文件。CVS 中包括了 `patch` 命令行实用程序，它必须置于此任务执行的路径上。

属性

`backups (all, boolean, N)`

如果为 `true`，则要对未做补丁的文件保存备份。默认为 `false`。

`ignorewhitespace (all, boolean, N)`

如果为 `true`，则在应用补丁文件时忽略空格差异。默认为 `false`。

`originalfile (all, File, N)`

要建立补丁的文件。

`patchfile (all, File, Y)`

包括 `diff` 输出的文件。

`quiet (all, boolean, N)`

如果为 `true`, 除非出现一个错误, 否则工作时不进行显示。默认为 `false`。

`reverse (all, boolean, N)`

如果为 `true`, 则假设补丁文件是通过新老文件交换而创建的。默认为 `false`。

`strip (all, int, N)`

由文件名中去除包含相应数目前导斜线的最小前缀。相当于 `patch` 的 `-p` 选项。

内容

无。

使用示例

应用包括在 `foo.patch` 中的 `diff`, 由 `diff` 输出猜出文件名:

```
<patch patchfile="foo.patch"/>
```

参见

请参见 `cvs` 任务。

pathconvert

1.4

将 Ant 路径转换为平台专有的路径

`org.apache.tools.ant.taskdefs.PathConvert`

将 Ant 路径或 `fileset` 转换为平台专有的路径, 并将结果保存在一个特性中。

属性

`dirsep (1.4, String, *)`

用作目录分隔符的字符, 如“`:`”。默认为当前 JVM 中的 `File.separator`。

pathsep (1.4, String, *)

用作路径分隔符的字符，如“/”。默认为当前JVM中的`File.separator`。

property (1.4, String, Y)

保存所转换路径的特性。

refid (1.4, Reference, *)

待转换路径的引用。

targetos (1.4, String, *)

同时定义pathsep和dirsep的一个快捷方式。合法的值为`unix`和`windows`。`dirsep`和`pathsep`值的选择要与所指定的操作系统相一致。

必须指定`targetos`或同时指定`dirsep`和`pathsep`。必须要有`refid`属性或一个嵌套`<path>`元素。

内容

0 到 n 个嵌套 <map> 元素 (1.4)

每个元素指定了 Unix 和 Windows 之间路径前缀的映射。Ant 仅应用第一个匹配的`<map>`元素。以下为在此环境下的合法`<map>`属性。

from (1.4, String, Y)

要匹配的前缀，如`C:`。当 Ant 在 Windows 上运行时，是不区分大小写的，而在 Unix 上运行时则区分大小写。

to (1.4, String, Y)

`from` 匹配时所用的替换。例如，`/usr`。

0 或 1 个嵌套 <path> 元素 (1.4)

一个`path`元素，用以代替`refid`属性。

使用示例

以下示例定义了一个`fileset`:

```
<fileset id="sources1" dir="src"
    includes="**/*.java"
    excludes="**/test/**/*.java"/>
```

以下为 pathconvert 如何将 fileset 转换为一个 Unix 形式的路径，并将结果保存在 p1 特性中：

```
<pathconvert targetos="unix" property="p1" refid="sources1"/>
```

p1 特性的值现在即为：

```
/home/aidan/src/com/oreilly/Book.java:/home/aidan/src/com/oreilly/Chapter.java
```

property	all
设置当前工程中的特性	org.apache.tools.ant.taskdefs.Property

设置工程中的特性。用户指定的特性总是优先于此任务所定义的特性。使用 ant 任务调用此工程的父工程所定义的特性也同样有此特点。

属性

classpath (1.3, 1.4, Path, N)

查找一个资源时所用的类路径。

classpathref (1.3, 1.4, Reference, N)

在构建文件中某处定义的类路径的引用。

environment (1.3, 1.4, String, *)

用于获取环境变量的前缀。在后面的例子中，我们用它来获取 TOMCAT_HOME 的值。在第四章的“environment DataType”一节中对此也有提及。

file (all, File, *)

特性文件的文件名。按照此特性文件的内容定义了一组特性（名 - 值对）。

location (all, File, *)

将特性值设置为一个绝对文件名。如果此属性包括一个绝对路径，那么任何 "/" "and" "\" 字符都将被转换为当前系统所用的约定。如果路径是相对于工程的，则会展开为一个绝对路径。

name (all, String, N)

要设置的特性的名字。

refid (all, Reference, *)

在工程中某处定义的一个路径或特性的引用。

resource (all, String, *)

一个特性文件的 Java 资源名。使用一个 ClassLoader 来加载此特性文件。

value (all, String, *)

此特性的一个显式值。

若指定 name，则 value、location 或 refid 三者必选其一。否则，在 resource、file 或 environment 中必取其一。

内容

0 或 1 个嵌套<classpath>元素 (1.3, 1.4)

可用以代替 classpath 属性。

使用示例

相对于工程的基目录，定义 builddir 和 srccdir 特性：

```
<property name="builddir" value="build"/>
<property name="srccdir" value="src"/>
```

以下示例基于 com.oreilly.antbook 包中 *test.properties* 的内容定义了两个特性：

```
<property resource="com/oreilly/antbook/test.properties">
  <classpath>
    <pathelement path="${srccdir}" />
  </classpath>
</property>
<!-- display the property values... --&gt;
&lt;echo message="book.title = ${book.title}" /&gt;
&lt;echo message="book.author = ${book.author}" /&gt;</pre>
```

此例首先获取了所有环境变量，在它们前面加上env.前缀。然后将TOMCAT_HOME环境变量的值赋予tomcat.home特性：

```
<property environment="env" />
<property name="tomcat.home" value="${env.TOMCAT_HOME}" />
```

record

1.4

为构建过程创建一个监听者

org.apache.tools.ant.taskdefs.Recorder

为当前构建过程创建一个监听者，将输出记录到一个文件中。对于同一个文件可以存在多个记录器（recorder）。

属性

action (1.4, *Enum*, *N*)

定义记录器是否开始或结束记录。合法的值为start和stop。首次遇到此任务时默认为start。在后面的调用中，记录器的状态将保持不变。

append (1.4, *boolean*, *N*)

如果为true，则当此记录器首次创建时将追加文件，而不是替换文件。默认为true。在同一构建中，以后的调用总是对文件进行追加。

loglevel (1.4, *Enum*, *N*)

确定日志级别。合法的值为error、warn、info、verbose和debug。对于每个记录器实例，级别可能有所不同。

name (1.4, *String*, *Y*)

要记入日志的文件的文件名。

内容

无。

使用示例

此例显示了在编译代码时，如何将详细的日志信息写到一个文件中。

```
<record name="javac.log" loglevel="debug"
        action="start" append="false"/>
<javac srcdir="${srcdir}" destdir="${builddir}"
       includes="com/oreilly/antbook/**">
</javac>
<record name="javac.log" action="stop"/>
```

rename

此任务在 Ant 1.2 中已经弃用；而代之以使用 move 任务。

replace

all

完成字符串替换

org.apache.tools.ant.taskdefs.Replace

在一个或多个文件中完成字符串替换。原文件将被替换而不是复制。

属性

defaultexcludes (all, boolean, N)

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。

默认为 true。

dir (all, File, *)

在指定多个文件时所用的基目录。

excludes (all, String, N)

要排除的文件模式列表（用逗号分隔），它们是对默认排除模式的补充。

excludesfile (all, File, N)

每行包括一个排除模式的文件的文件名。

file (all, File, *)

在其中完成替换的单个的文件。

includes (all, String, N)

要包含的文件模式的列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`propertyfile (1.3, 1.4, File, *)`

指定一个特性文件，其中包含嵌套`<replacefilter>`元素所引用的特性。

`summary (1.4, boolean, N)`

如果为`true`，则显示此操作的一个汇总报告。默认为`false`。

`token (all, String, *)`

要替换的记号。

`value (all, String, N)`

记号的新值。默认为一个空字符串。

在`file`或`dir`中必取其一。如果使用了一个嵌套`<replacetoken>`元素，则需要`token`属性。如果指定了一个嵌套`<replacefilter>`元素的`property`属性，则`propertyfile`属性是必要的。

内容

`0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all); <excludesfile>、<includesfile> (1.4)`

代替其相应属性，它们指定了所包含和排除的源文件组。

`0 到 n 个嵌套<replacefilter> 元素 (1.3, 1.4)`

允许多个替换，且结合`propertyfile`属性工作。`<replacefilter>`属性如下：

`token (1.3, 1.4, String, Y)`

要查找的记号。

`value (1.3, 1.4, String, *)`

替换文本。

`property (1.3, 1.4, String, *)`

特性名，其值要用作为替换文本。

可以指定 value 或 property，也可以均不指定。

0 或 1 个嵌套 <replacetoken> 和 <replacevalue> 元素 (all)

用以代替 token 和 value 属性，支持多行文本。

使用示例

用当前日期替换 @builddate@ 的所有出现：

```
<replace file="${builddir}/replaceSample.txt"
         token="@builddate@"
         value="${DSTAMP}"/>
```

下面的例子完成一个多行记号替换。它使用 XML `<![CDATA[...]]>` 来表示包含一个换行符的字面文本：

```
<replace file="${builddir}/replaceSample.txt">
  <replacetoken><![CDATA[Token line 1
Token line 2]]></replacetoken>
  <replacevalue><![CDATA[Line 1
Line 2]]></replacevalue>
</replace>
```

使用一个包含有记号替换值的特性文件。应用于所有源文件：

```
<replace dir="${srcdir}" includes="**/*.java" propertyfile="tokens.properties">
  <replacefilter token="@vendor@" property="vendor.name"/>
  <replacefilter token="@version@" property="version.name"/>
</replace>
```

rmic

调用 rmic 编译器

all

org.apache.tools.ant.taskdefs.Rmic

调用 rmic 编译器，为 Java 远程方法调用生成 (stub) 和骨架 (skeleton)。

属性

base (*all, File, Y*)

保存已编译文件的位置。

`classname (all, String, N)`

在此类上运行 rmic。

`classpath (all, Path, N)`

使用的类路径。

`classpathref (all, Reference, N)`

在构建文件中某处定义的类路径的引用。

`debug (1.3, 1.4, boolean, N)`

如果为 true，则向 rmic 传递 -g。默认为 false。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。

默认为 true。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔），它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`extdirs (1.4, Path, N)`

覆盖 Java 安装可选包的一般位置。

`filtering (all, boolean, N)`

如果为 true，则记号过滤必须发生。默认为 false。

`idl (1.3, 1.4, boolean, N)`

指示 rmic 生成 IDL 输出。

`idlopts (1.3, 1.4, String, N)`

idl=true 时的附加参数。

`iiop (1.3, 1.4, boolean, N)`

若为 true，则生成一个可移植的 RMI/IIOP 存根。默认为 false。

`iiopopts (1.3, 1.4, String, N)`

iiop=true 时的附加参数。

`includeantruntime (1.4, boolean, N)`

如果为 true，则类路径中包括 Ant 运行时库。默认为 true。

`includejavaruntime (1.4, boolean, N)`

如果为 true，则包括来自正在执行的 VM 的默认运行时库。默认为 false。

`includes (all, String, N)`

要包含的文件模式的列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`sourcebase (all, File, N)`

若指定，则向 rmid 传递 -keepgenerated 选项。生成的源文件移到所指定的目录中。

`stubversion (all, String, N)`

将此设置为 1.1，将向 rmic 传递 -v1.1 选项。

`verify (all, boolean, N)`

若为 true，则在将其传递给 rmic 之前先检验实现 Remote 的类。默认为 false。

内容

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all);
<excludesfile>、<includesfile> (1.4)

用以代替其相应属性，它们指定了所包含和排除的源文件组。

0 或 1 个嵌套 <classpath> 元素 (all)

可以用以代替 classpath 和 classpathref 属性。

0 到 n 个嵌套 <extdirs> 元素 (1.4)

可以用以代替 extdirs 属性。

使用示例

对 com.oreilly.remote 包中的所有类运行 rmic 编译器：

```
<rmic base="${builddir}" includes="com/oreilly/remote/*.class"/>
```

sequential

1.4

包含有序的任务

org.apache.tools.ant.taskdefs.Sequential

这是设计用来与 parallel 任务一同使用的容器任务。可以确保一组任务按顺序执行。

属性

无。

内容

任何嵌套任务。

使用示例

请参见 parallel 任务的示例。

参见

parallel 任务。

signjar

all

执行 jarsigner

org.apache.tools.ant.taskdefs.SignJar

执行 jarsigner 命令行工具。

属性

`alias (all, String, Y)`

指定签名所用的别名。

`internalsf (all, boolean, N)`

如果为 `true`, 则包括签名模块中的 `.SF` 文件。默认为 `false`。

`jar (all, String, Y)`

要签名的 JAR 文件。

`keypass (all, String, *)`

私钥的口令。

`keystore (all, String, N)`

密钥库位置。

`sectionsonly (all, boolean, N)`

如果为 `true`, 则不计算整个清单的散列值。默认为 `false`。

`sigfile (all, String, N)`

`.SF` 或 `.DSA` 文件的文件名。

`signedjar (all, String, N)`

签名 JAR 文件的文件名。

`storepass (all, String, Y)`

密钥库完整性的口令。

`storetype (all, String, N)`

密钥库类型。

`verbose (all, boolean, N)`

如果为 `true`, 则产生显示输出。默认为 `false`。

如果私钥口令不同于密钥库口令, 则 `keypass` 是必要的。

内容

0 到 n 个嵌套 <fileset> 元素 (1.4)

用以代替 jar 属性来对多个文件签名。

使用示例

```
<signjar jar="${builddir}/server.jar" alias="oreilly"  
        storepass="${password}"/>
```

sleep

1.4

暂停构建

org.apache.tools.ant.taskdefs.Sleep

令构建暂停指定的时间。

属性

failonerror (1.4, boolean, N)

如果为 true，当睡眠时出现任何错误构建即会失败。默认为 true。

hours (1.4, int, N)

小时数。

milliseconds (1.4, int, N)

毫秒数。

minutes (1.4, int, N)

分钟数。

seconds (1.4, int, N)

秒数。

内容

无。

使用示例

```
<!-- start a web server, then wait a few seconds for it to initialize -->
<sleep seconds="10"/>
<!-- now start the client unit tests -->
```

sql

执行 SQL 语句

all

`org.apache.tools.ant.taskdefs.SQLExec`

使用 JDBC 执行 SQL 语句。

属性

`autocommit (all, boolean, N)`

如果为 `true`, 则设置 JDBC `autocommit` 标志。默认为 `false`。

`classpath (all, Path, N)`

加载 JDBC 驱动程序时所用的类路径。

`classpathref (all, Reference, N)`

在构建文件中某处定义的类路径的引用。

`delimiter (1.4, String, N)`

用于间隔 SQL 语句的字符串。默认为 “;”。

`delimitertype (1.4, Enum, N)`

合法的值为 `row` 和 `normal`。默认为 `normal`, 这说明只要出现分隔符即会终止 SQL 命令。`row` 表示分隔符必须自己另起一行。

`driver (all, String, Y)`

JDBC 驱动程序的类名。

`onerror (all, Enum, N)`

一条语句失败时控制将发生什么情况。默认为 `abort`。合法的值如下:

`continue`

即使一条或多条语句失败, 此任务仍试图继续执行语句。

`abort`

出现错误时，通过此任务，在终止构建前事务将显式回滚。

`stop`

构建失败，而不会回滚一个失败的事务。不过，一旦 JVM 退出，数据库应当回滚事务。

`output (all, File, N)`

`print=true` 时用于存放结果集的输出文件。默认为 `System.out`。

`password (all, String, Y)`

数据库口令。

`print (all, boolean, N)`

如果为 `true`，则打印所有结果集。默认为 `false`。

`rdbms (all, String, N)`

指定一种 RDBMS，并限制此任务仅能在该 RDBMS 中执行。它应当等于 `DatabaseMetaData` 的 `getDatabaseProductName()` 方法的返回值。

`showheaders (all, boolean, N)`

如果为 `true`，则在打印结果集时显示标题列。默认为 `true`。

`src (all, File, *)`

包含要执行的 SQL 语句的文件。

`url (all, String, Y)`

数据库连接 URL。

`userid (all, String, Y)`

数据库用户 ID。

`version (all, String, N)`

指定一个版本号。只有当 RDBMS 版本与该值匹配时任务才执行。产品版本由 `DatabaseMetaData` 得到。

如果要执行的 SQL 语句指定为标签的文本内容，那么 `src` 属性不是必要的。

内容

文本内容 (*all*)

用以代替 SQL 语句的 `src` 属性。`delimiter` 属性所指定的字符将分隔多条语句。以 `//`、`--` 或 `REM` 开头的行为注释。

0 或 1 个嵌套 `<classpath>` 元素 (*all*)

可以用以代替 `classpath` 属性。

0 到 *n* 个嵌套 `<fileset>` 元素 (1.4)

用以代替 `src` 属性来指定包含 SQL 语句的多个文件。文件将按所列顺序执行。

0 到 *n* 个嵌套 `<transaction>` 元素 (*all*)

每个元素均定义在一个事务中执行的一个命令块。支持一个属性：

`src` (*all, String, **)

包含 SQL 语句的文件的文件名。

如果 SQL 语句作为文本嵌套在 `<transaction>` 元素中，那么 `src` 属性可以被忽略。

使用示例

执行包含在 `report.sql` 中的语句（可能是多条）：

```
<sql driver="${db.driver}"
      url="${db.url}"
      userid="${db.userid}"
      password="${db.password}"
      src="report.sql"/>
```

执行指定为 `sql` 任务内容的 SQL 语句：

```
<sql driver="${db.driver}"
      url="${db.url}"
      userid="${db.userid}"
      password="${db.password}">
    SELECT *
    FROM ReportTbl;
```

```
// 额外的语句……  
SELECT ... ;  
</sql>
```

style	all
完成 XSLT 转换	org.apache.tools.ant.taskdefs.XSLTProcess

完成 XSLT 转换。XSLT 样式表定义了 XSLT 处理程序如何将 XML 转换为其他文本格式。

属性

`basedir (all, File, N)`

指定在哪里找到要转换的 XML 文件。默认为工程的基目录。

`classpath (1.4, Path, N)`

查找 XSLT 处理程序时所用的类路径。

`classpathref (1.4, Reference, N)`

在构建文件中某处定义的类路径的引用。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。
默认为 `true`。

`destdir (all, File, *)`

指定转换结果在哪里存储。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔），它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件的文件名。

`extension (all, String, N)`

转换结果的默认文件扩展名。默认为 `.html`。

force (*1.4, boolean, N*)

如果为 `true`, 即使比其源 XML 或 XSLT 文件还要新, 仍要创建目标文件。
默认为 `false`。

in (*1.3, 1.4, File, N*)

指定用于转换的一个 XML 文件。与 `out` 属性结合使用。

includes (*all, String, N*)

要包含的文件模式的列表 (用逗号分隔)。

includesfile (*all, File, N*)

每行包括一个包含模式的文件的文件名。

out (*1.3, 1.4, File, N*)

指定对应转换结果的文件名。与 `in` 属性结合使用。

processor (*all, String, N*)

所用的 XSLT 处理程序。默认 (推荐) 值为 `trax`。合法的值如下:

trax

支持与 Sun 公司的 JAXP 1.1 兼容的所有处理程序。

xslp

已经废弃。

xalan

支持 Apache Xalan 1.x 版本。

style (*all, String, Y*)

XSLT 样式表名。

除非指定了 `in` 和 `out` 属性, 否则 `destdir` 属性是必要的。

内容

0 到 *n* 个嵌套 `patternset` 元素: `<exclude>`、`<include>`、`<patternset>` (*all*);
`<excludesfile>`、`<includesfile>` (*1.4*)

代替其相应属性, 它们指定了所包含和排除的各组 XML 文件。

0 或 1 个嵌套 <classpath> 元素 (1.4)

可以用以代替 classpath 属性。

0 到 n 个嵌套 <param> 元素 (1.3, 1.4)

每个元素使用以下属性向 XSLT 样式表传递一个参数:

name (1.3, 1.4, String, Y)

XSLT 参数名。

expression (1.3, 1.4, String, Y)

参数值。字面文本必须放在单引号中传递，否则样式表将其当作一个表达式 (注 6)。

使用示例

使用 *customers.xslt* 转换 *customers.xml*，将结果放在构建目录中:

```
<style destdir="${builddir}" style="customers.xslt">
  <param name="date" expression="${DSTAMP}" />
  <include name="customers.xml" />
</style>
```

tar

创建一个 tar 文件

all

org.apache.tools.ant.taskdefs.Tar

创建一个 tar 压缩文件。

属性

basedir (all, File, N)

指定基目录，由此向 tar 文件增加文件。

defaultexcludes (all, boolean, N)

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。

默认为 true。

注 6： 尽管 Ant 手册称字面文本必须用单引号进行转义，但在这种情况下则不尽然。样式表总是将其值当作文本而不是表达式来对待。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔），它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`includes (all, String, N)`

要包含的文件模式的列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`longfile (1.3, 1.4, String, N)`

对于带有长文件名（大于 100 个字符）的文件，对其处理加以控制。合法的值为 `truncate`、`fail`、`warn`、`omit` 和 `gnu`。默认为 `warn`。

`tarfile (all, File, Y)`

要创建的 tar 文件。

内容

0 到 n 个嵌套 `patternset` 元素: `<exclude>`、`<include>`、`<patternset>` (all);
`<excludesfile>`、`<includesfile>` (1.4)

代替其相应属性，它们指定了所包含和排除的源文件组。

0 到 n 个嵌套 `<tarfileset>` 元素 (1.3, 1.4)

每个元素均支持所有 `fileset` 属性和内容 (`includes`、`excludes` 等等)，以及以下附加属性:

`mode (1.3, 1.4, String, N)`

3 位 8 进制字符串，用以指定用户、组和其他模式。

`username (1.3, 1.4, String, N)`

`tar` 项的用户名。

`group (1.3, 1.4, String, N)`

`tar` 项的组名。

使用示例

创建一个 tar 压缩文件，其中包括构建目录中的所有类文件：

```
<tar tarfile="${dist}/classes.tar"
      basedir="${builddir}"
      includes="**/*.class"/>
```

参见

请参见 gzip 任务。

taskdef	all
----------------	------------

增加一个任务

`org.apache.tools.ant.taskdefs.Taskdef`

向当前工程增加任务。它用于定义未在 *ant.jar* 的 *default.properties* 文件中定义的任务。

属性

classname (*all, String, **)

实现此任务的类。

classpath (*all, Path, N*)

所用的类路径。

file (*1.4, File, N*)

包含一个或多个任务定义的特性文件的文件名。每一行的格式如下：

`taskname=full.package.name.TaskClass`

name (*all, String, **)

任务名。

resource (*1.4, String, N*)

包含一个或多个任务定义的特性文件的 Java 资源名。它相当于 **file** 属性，不过在此使用一个 `ClassLoader` 来查找特性文件。

除非指定了 file 或 resource 属性，否则 name 和 classname 属性就是必要的。

内容

0 或 1 个嵌套 <classpath> 元素 (*all*)

可以用以代替 classpath 属性。

使用示例

定义一个定制任务，它可在整个工程中使用：

```
<taskdef name="mycodegen" classname="com.foobar.tasks.MyCodeGen" />
```

touch

all

更新时间戳

org.apache.tools.ant.taskdefs.Touch

更新一个或多个文件的时间戳。

属性

`datetimestamp` (*all, String, N*)

文件的新修改时间，格式为 MM/DD/YYYY HH:MM AM 或 PM。默认为当前时间。

`file` (*all, File, **)

要处理的文件的文件名。如果不存在此文件，则使用嵌套 <fileset> 元素创建一个文件。

`millis` (*all, long, N*)

文件的新修改时间，表示为自 1970 年 1 月 1 日以来的毫秒数。

除非指定了一个嵌套 <fileset>，否则 `file` 属性是必要的。

内容

0 到 n 个嵌套 <fileset> 元素 (1.4)

指定要处理的文件和目录。

使用示例

用当前时间更新 *build.xml* 的时间戳：

```
<touch file="build.xml" />
```

修改构建目录中所有文件和目录的时间戳：

```
<touch datetime="06/25/1999 6:15 AM">
  <fileset dir="${builddir}" />
</touch>
```

tstamp	all
设置时间戳特性	org.apache.tools.ant.taskdefs.Tstamp

设置 DSTAMP、TSTAMP 和 TODAY 特性。另外，每个特性按照表 7-4 所列的格式使用 `java.text.SimpleDateFormat` 进行格式化。

表 7-4: tstamp 格式

特性	格式	示例
DSTAMP	yyyyMMdd	20010916
TSTAMP	HHmm	1923
TODAY	MMMM d yyyy	September 16 2001

属性

无。

内容

0 到 n 个嵌套 <format> 元素 (1.3, 1.4)

支持定制格式。每个结果置于一个特性中。以下为 <format> 元素属性：

property (1.3, 1.4, String, Y)

格式化的时间戳所置于的特性的特性名。

pattern (1.3, 1.4, String, Y)

由 java.text.SimpleDateFormat 定义的格式模式。

offset (1.3, 1.4, int, N)

与当前时间的数字偏移量。

unit (1.3, 1.4, String, N)

定义偏移量参数的单位。合法的值为： millisecond、second、minute、hour、day、week、month 和 year。

locale (1.4, String, N)

构造 SimpleDateFormat 对象所用的场所约定。请参见 java.util.Locale 的文档。

使用示例

生成3个特性，分别包含当前时间、当前时间前一小时以及当前时间后一分钟。均按 September 16 2001 07:37 PM 的格式：

```
<tstamp>
  <format property="now"
    pattern="MMMM d yyyy hh:mm aa"/>
  <format property="hour_earlier"
    pattern="MMMM d yyyy hh:mm aa"
    offset="-1"
    unit="hour"/>
  <format property="minute_later"
    pattern="MMMM d yyyy hh:mm aa"
    offset="1"
    unit="minute"/>
</tstamp>
```

```
<!-- now display one of the values -->
<echo>now = ${now}</echo>
```

typedef

1.4

增加定制 DataType 定义

org.apache.tools.ant.taskdefs.Typedef

向当前工程增加一个或多个定制 DataType 定义。

属性

`name (1.4, String, *)`

要增加的 DataType 的名字。

`classname (1.4, String, *)`

实现 DataType 的 Java 类。

`file (1.4, File, N)`

包含 DataType 定义的特性文件。每行的格式如下：

`name=classname`

`resource (1.4, String, N)`

特性文件的 Java 资源名。使用一个 ClassLoader 来加载特性文件。

`classpath (1.4, Path, N)`

所用的类路径。

除非指定了 file 或 resource 属性，否则 name 和 classname 属性是必要的。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.4)`

可以用以代替 classpath 属性。

使用示例

以下示例创建定制 DataType customer，它由类 com.oreilly.domain.Customer 实现：

```
<typedef name="customer" classname="com.oreilly.domain.Customer"/>
```

unjar

unjar、unwar 和 unzip 任务是等同的。org.apache.tools.ant.taskdefs.Expand 类实现了它们。其属性和示例请参见 unzip 一节。

untar

all

展开一个 tar 文件

org.apache.tools.ant.taskdefs.Untar

展开一个 tar 压缩文件。

属性

`dest (all, File, Y)`

目标目录。

`overwrite (1.4, boolean, N)`

如果为 true，即使文件比 tar 文件中的文件更新，也要重写。默认为 true。

`src (all, File, Y)`

要展开的 tar 文件。

内容

无。

使用示例

```
<untar src="foo.tar" dest="${builddir}" />
```

参见

请参见 tar 任务。

unwar

unjar、unwar和unzip任务是等同的。`org.apache.tools.ant.taskdefs.Expand`类实现了它们。其属性和示例请参见unzip一节。

unzip (以及 unjar 和 unwar)

all

展开一个文件

`org.apache.tools.ant.taskdefs.Untar`

解压缩一个ZIP文件、一个JAR文件或一个WAR文件。

属性

`dest (all, File, Y)`

目标目录。

`src (all, File, Y)`

要展开的文件。

`overwrite (1.4, boolean, N)`

如果为true，即使文件比压缩文件中的还要新，也将对文件进行重写。默认为true。

内容

无。

使用示例

```
<unzip src="dist.jar" dest="${builddir}" />
```

参见

jar、war和zip任务。

uptodate all

如果文件是最新的，则设置一个特性 org.apache.tools.ant.taskdefs.Uptodate

相应于对应的源文件，如果一个或多个目标文件是最新的，则设置一个特性。如果目标比所有源文件都要新，则设置此特性。

属性

property (*all, String, Y*)

要设置的特性。

targetfile (*all, File, **)

要检查的目标文件。

value (*1.4, String, N*)

要为特性设置的值。默认为 true。

除非指定了一个嵌套 `<mapper>`，否则 `targetfile` 属性就是必要的。

内容

0 到 n 个嵌套 `<srcfiles>` 元素 (all)

每个元素均为一个 `fileset`，它定义了要比较的一组源文件。

0 或 1 个嵌套 `<mapper>` 元素 (1.3, 1.4)

定义源文件如何与目标文件相关。如果未指定，此任务使用一个合并 `mapper`，其 `to` 属性设置为 `uptodate` 任务的 `targetfile` 属性的值。

使用示例

如果 `classes.jar` 比工程目录及其所有子目录中所见的 `.class` 文件都要新，则以下示例设置 `jar_ok` 特性。

```
<uptodate property="jar_ok" targetfile="${builddir}/classes.jar">
  <srcfiles dir="${builddir}" includes="**/*.class"/>
</uptodate>
```

下一个例子假设已经有一个定制的代码生成器，它将基于`.template`文件创建`.java`文件。在此使用一个嵌套`<mapper>`，并且只要`.java`文件较之于其相应的`.template`文件更新，则设置`codegen_uptodate`特性。

```
<uptodate property="codegen_uptodate">
  <srcfiles dir="src" includes="**/*.template"/>
  <mapper type="glob" from="*.template" to="*.java"/>
</uptodate>
```

参见

`mappers`已在第四章中进行了讨论。

war	all
创建一个 WAR 文件	<code>org.apache.tools.ant.taskdefs.War</code>
创建一个 WAR 文件。WAR 文件为 servlet 的部署机制。	

属性

`basedir (all, File, N)`

指定基目录，由此向 WAR 文件增加文件。

`compress (all, boolean, N)`

如果为`true`，则压缩 WAR 文件。默认为`true`。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式，如第四章中的“fileset DataType”一节所述。

默认为`true`。

`encoding (1.4, String, N)`

指定 JAR 文件中的文件名字符编码。默认为 UTF-8。Ant 规范警告称，修改此属性可能会导致得到 Java 无法使用的 JAR 文件。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔），它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件的文件名。

`filesonly (1.4, boolean, N)`

如果为 `true`, 则不创建空目录。默认为 `false`。

`includes (all, String, N)`

要包含的文件模式的列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`manifest (all, File, N)`

要使用的清单文件的文件名。

`update (1.4, boolean, N)`

如果为 `true`, 则在做出修改时更新现有 WAR 文件, 而不是将其删除并从头重新创建。默认为 `false`。

`warfile (all, File, Y)`

要创建的 WAR 文件的文件名。

`webxml (all, File, Y)`

部署描述文件名。它置于 `WEB-INF` 目录中, 并且被重命名为 `web.xml`。

`whenempty (all, Enum, N)`

当要包含的文件未找到时所采取的操作。默认为 `create`。合法的值如下:

`fail`

终止构建。

`skip`

不创建 WAR 文件。

`create`

若包含文件不存在, 则创建一个空的 WAR 文件。

内容

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all);
<excludesfile>、<includesfile> (1.4)

代替其相应属性，它们指定了所包含和排除的源文件组。

0 到 n 个嵌套 <classes> 元素 (all)

zipfileset 元素，定义哪些文件置于 WAR 文件的 WEB-INF/classes 目录中。

0 到 n 个嵌套 <fileset> 元素 (all)

fileset 元素，定义哪些文件置于 WAR 文件的顶级目录中。

0 到 n 个嵌套 <lib> 元素 (all)

zipfileset 元素，定义哪些文件置于 WAR 文件的 WEB-INF/lib 目录中。

0 到 n 个嵌套 <metainf> 元素 (1.4)

zipfileset 元素，定义哪些文件置于 WAR 文件的 META-INF 目录中。

0 到 n 个嵌套 <webinf> 元素 (all)

zipfileset 元素，定义哪些文件置于 WAR 文件的 WEB-INF 目录中。

0 到 n 个嵌套 <zipfileset> 元素 (1.3, 1.4)

zipfileset 元素，定义哪些文件置于 WAR 文件的顶级目录中。

使用示例

以下示例创建了一个名为 ecom.war 的 WAR 文件。src/docroot 目录中的文件置于 WAR 文件的根目录中。所有类文件放在 WEB-INF/classes 目录下，JAR 文件放在 WEB-INF/lib 目录下。

```
<war warfile="${builddir}/ecom.war"
      webxml="src/metadata/web.xml">
  <fileset dir="src/docroot"/>
  <classes dir="${builddir}" includes="**/*.class"/>
  <lib dir="${builddir}" includes="*.jar"/>
</war>
```

参见

有关 `zipfileset` 的描述, 请参见 `zip` 任务。

zip	all
创建一个 ZIP 文件	<code>org.apache.tools.ant.taskdefs.Zip</code>
创建一个 ZIP 文件。	

属性

`basedir (all, File, N)`

指定基目录, 由此向 ZIP 文件增加文件。

`compress (all, boolean, N)`

如果为 `true`, 则压缩 ZIP 文件。默认为 `true`。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式, 如第四章中的“fileset DataType”一节所述。
默认为 `true`。

`encoding (1.4, String, N)`

指定 ZIP 文件中的文件名字符编码。默认为当前 VM 所用的编码。

`excludes (all, String, N)`

要排除的文件模式列表 (用逗号分隔), 它们是对默认排除模式的补充。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`filesonly (1.4, boolean, N)`

如果为 `true`, 则不创建空目录。默认为 `false`。

`includes (all, String, N)`

要包含的文件模式的列表 (用逗号分隔)。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`update (1.4, boolean, N)`

如果为 `true`, 则在做出修改时更新现有 ZIP 文件, 而不是将其删除并从头重新创建。默认为 `false`。

`whenempty (all, Enum, N)`

没有匹配的文件时所采取的操作。默认为 `create`。合法的值如下:

`fail`

终止构建。

`skip`

不创建 ZIP 文件。

`create`

不存在文件时, 创建一个空的 ZIP 文件。

`zipfile (all, File, Y)`

要创建的 ZIP 文件的文件名。

内容

`0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all); <excludesfile>、<includesfile> (1.4)`

代替其相应属性, 它们指定了所包含和排除的源文件组。

`0 到 n 个嵌套 <fileset> 元素 (all)`

`fileset` 元素, 定义哪些文件置于 ZIP 文件中。

`0 到 n 个嵌套 <zipfileset> 元素 (1.3, 1.4)`

每个元素均为一个 `zipfileset`, 这是 `fileset` 的扩展。

`zipfileset` 对 `fileset` 增加了以下属性:

`fullpath (String, N)`

只能对一个文件设置。当一个文件增加到压缩文件时, 指定其完全路径名。

`prefix (String, N)`

如果指定，以此路径为前缀的所有文件均置于压缩文件中。

`src (File, *)`

指定一个现有的 ZIP 文件，可以抽取出其内容，再插入到新的 ZIP 文件中。

`dir` 或 `src` 必选其一，且只能选其一 (`dir` 由 `fileset` 继承得到)。

使用示例

创建一个包含所有源代码的 ZIP 文件：

```
<zip zipfile="${builddir}/src.zip" basedir="src"/>
```

参见

请参见 `ear`、`jar`、`tar` 和 `war` 任务，它们都基于 `zip` 所用的同样的代码。

第八章

可选任务

本章列出了 Ant 1.2、1.3 和 1.4 版本中所有可用的可选任务。在此沿用了第七章所用的格式。

任务汇总

表 8-1 对 Ant 所有可选任务做了汇总。

表 8-1：可选任务汇总

任务名	Ant 版本	含义
antlr	1.3, 1.4	运行 ANTLR 解析程序和解释生成工具
blgenclient	1.4	由一个现有的 <i>ejb-jar</i> 文件创建一个客户 JAR 文件。 其名由 “Borland Generated client” 派生得到
cab	all	创建 Microsoft .cab 压缩文件
cccheckin	1.3, 1.4	执行一个 Rational ClearCase <i>checkin</i> 命令
cccheckout	1.3, 1.4	执行一个 ClearCase <i>checkout</i> 命令
ccmcheckin	1.4	执行一个 Continuus ^a <i>ci</i> 命令
ccmcheckintask	1.4	执行一个 Continuus <i>ci default</i> 命令

表 8-1：可选任务汇总（续）

任务名	Ant 版本	含义
ccmcheckout	1.4	执行一个 Continus <i>co</i> 命令
ccmcreatetask	1.4	执行一个 Continus <i>create_task</i> 命令
ccmreconfigure	1.4	执行一个 Continus <i>reconfigure</i> 命令
ccuncheckout	1.3, 1.4	执行一个 ClearCase <i>uncheckout</i> 命令
ccupdate	1.3, 1.4	执行一个 ClearCase <i>update</i> 命令
csc	1.3, 1.4	编译 C# 源代码
ddcreator	all	由文本文件创建串行化 EJB 部署描述文件
depend	1.3, 1.4	基于对内容的分析，确定哪些类已经过时，且对类文件和源文件时间戳加以比较
ejbc	all	执行 BEA WebLogic Server 的 <i>ejbc</i> 工具，生成在该环境下部署 EJB 组件所需的代码
ejbjar	all	创建与 EJB 1.1 兼容的 <i>ejb-jar</i> 文件
ftp	all	实现一个基本 FTP 客户
icontract	1.4	执行 iContract 合约设计 (Design By Contract) 预处理程序
ilasm	1.3, 1.4	汇编 .NET 中间语言 (Intermediate Language) 文件
iplanet-ejbc	1.4	对 iPlanet 应用服务器 (iPlanet Application Server) 6.0 版本编译 EJB 存根和骨架
javacc	all	基于一个语法文件执行 JavaCC (Java compiler compiler) 编译器
javah	1.3, 1.4	执行 <i>javah</i> 工具，由一个或多个 Java 类生成 JNI (Java Native Interface, Java 本地接口) 首部
jdepend	1.4	执行 JDepend 工具
jjtree	all	为 JavaCC 执行 JJTree 预处理程序
jlink	all	构建一个 JAR 或 ZIP 文件，可选地合并现有 JAR 和 ZIP 压缩文件的内容

表 8-1：可选任务汇总（续）

任务名	Ant 版本	含义
jpcovmerge	1.4	将多个 JProbe Coverage 快照合并为一个
jpcovreport	1.4	由一个 JProbe Coverage 快照创建一个报告
junit	all	使用 JUnit 测试框架执行单元测试
junitreport	1.3, 1.4	基于 junit 任务的多个 XML 文件，创建一个格式化的报告
maudit	1.4	执行 WebGain Quality Analyzer 来分析 Java 源代码以找出编程错误
mimemail	1.4	发送带有 MIME 附件的 SMTP 邮件
mmetrics	1.4	在一组 Java 文件上执行 WebGain Quality Analyzer，报告代码复杂性和其他度量指标
mparse	all	基于一个语法文件，执行现在已经废弃的 Metamata MParse compiler compiler 编译器
native2ascii	all	将本地编码的文件转换为包含转义 Unicode 字符的 ASCII 编码文件
netrexxc	all	编译一组 NetRexx 文件
p4change	1.3, 1.4	由一个 Perforce 服务器请求一个新的修改表
p4counter	1.4	取得和设置 Perforce 计数器值
p4edit	1.3, 1.4	由 Perforce 打开文件从而进行编辑
p4have	1.3, 1.4	在当前客户视图中列出 Perforce 文件
p4label	1.3, 1.4	在当前 Perforce 工作区中为文件创建一个标签
p4reopen	1.4	将文件在 Perforce 修改表间移动
p4revert	1.4	回复已打开的 Perforce 文件
p4submit	1.3, 1.4	将文件记入一个 Perforce 库 (Perforce depot)
p4sync	1.3, 1.4	令工作区与 Perforce 库同步
propertyfile	1.3, 1.4	创建或编辑 Java 特性文件

表 8-1：可选任务汇总（续）

任务名	Ant 版本	含义
pvcs	1.4	由一个 PVCS 存储库抽取文件
renameext	all	重命名文件扩展名。此任务在 Ant 1.3 中已经弃用，而代之以使用有一个 glob mapper 的 move 任务
rpm	1.4	构建一个 Linux RPM 文件
script	all	执行一个 BSF 脚本
sound	1.3, 1.4	在构建过程结束时播放一个声音文件
starteam	all	将文件由 StarTeam 写出
stylebook	1.3, 1.4	执行 Apache Stylebook 文档生成程序
telnet	1.3, 1.4	执行一个 telnet 会话
test	1.3, 1.4	执行 org.apache.testlet 框架中的一个单元测试
vsscheckin	1.4	将文件写入到 Visual SourceSafe
vsscheckout	1.4	将文件由 Visual SourceSafe 写出
vssget	all	由 Visual SourceSafe 取得文件
vsshistory	1.4	显示 Visual SourceSafe 中文件和工程的历史
vsslabel	1.3, 1.4	为 Visual SourceSafe 中的文件和工程赋予一个标签
wljspc	all	使用 BEA WebLogic Server 的 JSP 编译器预编译 JSP 文件
wlrun	all	启动 BEA WebLogic Server 的一个实例
wlstop	all	停止 BEA WebLogic Server 的一个实例
xmlvalidate	1.4	使用任何 SAX 解析程序来验证 XML 文档是否良构且合法

a: 尽管 Ant 任务仍表示为 Continuus 命令，但此产品现在称为 Telelogic Synergy，可由 <http://www.telelogic.com> 得到。

可选任务参考

本章余下的部分将对 Ant 的各个可选任务提供详细的信息。属性描述所采用的格式与前一章完全类似。

antlr 1.3, 1.4

运行 ANTLR 解析程序 org.apache.tools.ant.taskdefs.optionalANTLR

运行 ANTLR 解析程序和解释生成工具。要运行此任务必须安装 ANTLR，可在 <http://www.antlr.org> 得到。这个任务将语法文件与目标文件相比较，只有在语法文件更新时才运行 ANTLR。

属性

dir (1.3, 1.4, File, N)

所创建的新JVM的工作目录。仅当 fork=true 时合法。

fork (1.3, 1.4, boolean, N)

如果为 true，则在其自己的 JVM 中运行 ANTLR。默认为 false。

outputdirectory (1.3, 1.4, File, N)

所生成文件的目标目录。默认为包含语法文件的目录。

target (1.3, 1.4, File, Y)

语法文件的文件名。

内容

无。

blgenclient 1.4

创建一个客户 JAR 文件 org.apache.tools.ant.taskdefs.optional.ejb.BorlandGenerateclient

由一个现有的 *ejb-jar* 文件创建一个客户 JAR 文件。blgenclient 派生自“Borland Generated Client”。它设计用于 Borland Application Server v4.5。

属性

`classpath (1.4, Path, N)`

java 任务所用的类路径。仅当模式为 java 时合法。

`classpathref (1.4, Reference, N)`

在构建文件中某处定义的一个类路径的引用。仅当模式为 java 时合法。

`clientjar (1.4, File, N)`

要创建的客户 JAR 的名字。如果省略，此任务将在文件名后追加 client。例如，`foo.jar` 将变成 `fooclient.jar`。

`debug (1.4, boolean, N)`

如果为 `true`，此任务将把 `-trace` 命令行选项传递到底层命令。默认为 `false`。

`ejbjar (1.4, File, Y)`

`ejb-jar` 文件，由此生成客户 JAR。

`mode (1.4, String, N)`

指定命令如何启动。默认为 `java`。合法值如下：

`fork`

使用 `exec` 核心任务。

`java`

使用 `java` 核心任务（注 1）。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.4)`

一个 `path` 元素，用以代替 `classpath` 或 `classpathref` 属性。

注 1：即使在“java”模式下，此任务仍会创建一个新的 JVM。

cab all

创建 Microsoft CAB 文件 org.apache.tools.ant.taskdefs.optional.Cab

创建 Microsoft CAB 压缩文件，在 Windows 平台上，必须在路径上包括 Microsoft *cabarc* 工具。在非 Windows 平台上，可以使用 *libcabinet*，此工具可由 http://trill.cis.fordham.edu/~barbacha/cabinet_library/ 得到。

属性

basedir (*all, File, Y*)

由此压缩文件的目录。

cabfile (*all, File, Y*)

要创建的 CAB 文件。

compress (*all, boolean, N*)

如果为 true，则压缩文件。默认为 true。

defaultexclude (*all, boolean, N*)

确定是否使用默认排除模式。默认为 true。

excludes (*all, String, N*)

要排除的文件模式列表（用逗号分隔）。

excludesfile (*all, File, N*)

每行包括一个排除模式的文件名。

includes (*all, String, N*)

要包含的文件模式列表（用逗号分隔）。

includesfile (*all, File, N*)

每行包括一个包含模式的文件的文件名。

options (*all, String, N*)

指定底层 cabarc 命令的额外命令行参数。

verbose (*all, boolean, N*)

如果为 true，则此任务指示 cabarc 命令以详细模式操作。默认为 false。

内容

0 到 n 个嵌套 `patternset` 元素: `<exclude>`、`<include>`、`<patternset>` (*all*);
`<excludesfile>`、`<includesfile>` (1.4)

用以替代其相应属性，它们指定了包含和排除的源文件组。

0 到 n 个嵌套 `<fileset>` 元素 (*all*)

`fileset` 元素，指定了 CAB 压缩文件中所包含的文件。

cccheckin

1.3, 1.4

执行一个 ClearCase *checkin* 命令 org.apache.tools.ant.taskdefs.optional.clearcase.CCCheckin

执行一个 ClearCase *checkin* 命令。

属性

`cleartoololdir` (1.3, 1.4, String, N)

指定 *cleartool* 所在的目录。

`comment` (1.3, 1.4, String, *)

写入文件时所用的注释。

`commentfile` (1.3, 1.4, String, *)

包含写入文件时所用注释的文件。

`identical` (1.3, 1.4, boolean, N)

如果为 `true`，即使文件与其原始文件相同，此任务也会将此文件写入。默认为 `false`。

`keepcopy` (1.3, 1.4, boolean, N)

如果为 `true`，则此任务保存文件的一个拷贝，并带有一个 `.keep` 扩展名。默认为 `false`。

`nowarn` (1.3, 1.4, boolean, N)

如果为 `true`，则此任务不显示警告消息。默认为 `false`。

`preservetime` (1.3, 1.4, boolean, N)

如果为 `true`，则此任务保留文件修改时间。默认为 `false`。

`viewpath (1.3, 1.4, String, N)`

ClearCase 视图文件或目录的路径。

`comment` 和 `commentfile` 属性是可选的，不过不能同时指定。

内容

无。

`cccheckout`

1.3, 1.4

执行一个 ClearCase `checkout` 命令 `org.apache.tools.ant.taskdefs.optional.clearcase.CCCheckout`

执行一个 ClearCase `checkout` 命令。

属性

`branch (1.3, 1.4, String, N)`

指定写出文件时所用的分支。

`clear 工具 dir (1.3, 1.4, String, N)`

指定 `cleartool` 所在的目录。

`comment (1.3, 1.4, String, *)`

写出文件时所用的注释。

`commentfile (1.3, 1.4, String, *)`

包含写出文件时所用注释的文件。

`nodata (1.3, 1.4, boolean, N)`

如果为 `true`, 此任务将写出文件, 但是并不创建包含可编辑数据的文件。默认为 `false`。

`nowarn (1.3, 1.4, boolean, N)`

如果为 `true`, 则此任务不显示警告消息。默认为 `false`。

`out (1.3, 1.4, String, N)`

指定一个不同的文件名的将文件写出。

`reserved (1.3, 1.4, boolean, Y)`

如果为 `true`, 则此任务原样写出文件。默认为 `true`。

`version (1.3, 1.4, boolean, N)`

如果为 `true`, 则此任务允许写出一个非最新版本的文件。默认为 `false`。

`viewpath (1.3, 1.4, String, N)`

`clearCase` 视图文件或目录的路径。

`Comment` 和 `commentfile` 属性是可选的, 不过不能同时指定。

内容

无。

ccmcheckin

1.4

执行一个 Continuus `ci` 命令 `org.apache.tools.ant.taskdefs.optional.ccm.CCMCheckin`

执行一个 Continuus `ci` 命令。这个任务和其他 `ccm` 任务都是 Continuus Source Manager 产品的包装器。尽管这些任务仍表示为 Continuus, 但此产品在 2000 年 9 月即由 Telelogic 收购, 并称为 Telelogic Synergy, 它可在 `http://www.telelogic.com` 得到。

属性

`ccmdir (1.4, String, N)`

包含 `ccm` 可执行程序的目录。如果未指定, 则任务要搜索路径。

`comment (1.4, String, N)`

此文件的注释。默认为 “`Checkin`” + 当前日期和时间。

`file (1.4, File, Y)`

要写入的文件。

`task (1.4, String, N)`

写入文件时所用的 Continuus 任务号。

内容

无。

ccmcheckintask 1.4

执行一个 Continuus *ci default* 命令 org.apache.tools.ant.taskdefs.optional.ccm.
CCMCheckinDefault

执行一个 Continuus *ci default* 命令。

属性

`ccmdir (1.4, String, N)`

包含 *ccm* 可执行程序的目录。如果未指定，则此任务要搜索路径。

`comment (1.4, String, N)`

写入文件时所用的注释。

`task (1.4, String, N)`

写入文件时所用的 Continuus 任务号。

内容

无。

ccmcheckout 1.4

执行一个 Continuus *co* 命令 org.apache.tools.ant.taskdefs.optional.ccm.CCMCheckout

执行一个 Continuus *co* 命令。

属性

`ccmdir (1.4, String, N)`

包含 *ccm* 可执行程序的目录。如果未指定，则此任务要搜索路径。

`comment (1.4, String, N)`

写出文件时所用的注释。

`file (1.4, File, Y)`

要写出的文件。

`task (1.4, String, N)`

写出文件时所用的 Continuus 任务号。

内容

无。

ccmcreatetask

1.4

执行一个 Continuus *create_task* 命令

org.apache.tools.ant.taskdefs.optional.
ccm.CCMCreateTask

执行一个 Continuus *create_task* 命令。

属性

`ccmdir (1.4, String, N)`

包含 *ccm* 可执行程序的目录。如果未指定，则此任务要搜索路径。

`comment (1.4, String, N)`

此操作所用的注释。

`platform (1.4, String, N)`

/plat 命令行选项。

`release (1.4, String, N)`

/release 命令行选项。

`resolver (1.4, String, N)`

/resolver 命令行选项。

subsystem (*1.4, String, N*)

/subsystem 命令行选项。

task (*1.4, String, N*)

所用的 Continus 任务号。

内容

无。

ccmreconfigure

1.4

执行一个 Continus reconfigure 命令

org.apache.tools.ant.taskdefs.optional.
ccm.CCMReconfigure

执行一个 Continus *reconfigure* 命令。

属性

ccmdir (*1.4, String, N*)

包含 *ccm* 可执行程序的目录。如果未指定，则此任务要搜索路径。

ccmproject (*1.4, String, Y*)

对于此命令的工程的工程名。

recurse (*1.4, boolean, N*)

如果为 *true*，则此任务递归地处理子工程。默认为 *false*。

verbose (*1.4, boolean, N*)

如果为 *true*，则此任务打印详细信息。默认为 *false*。

内容

无。

ccuncheckout	1.3, 1.4
执行一个 ClearCase uncheckout 命令	org.apache.tools.ant.taskdefs.optional .clearcase.CCUnCheckout
执行一个 ClearCase <i>uncheckout</i> 命令。	

属性**cleartooldir (1.3, 1.4, String, N)**指定 *cleartool* 所在的目录。**keepcopy (1.3, 1.4, boolean, N)**如果为 *true*, 此任务保存文件的一个拷贝, 并带有一个 *.keep* 扩展名。默认为 *false*。**viewpath (1.3, 1.4, String, N)**

ClearCase 视图文件或目录的路径。

内容

无。

ccupdate	1.3, 1.4
执行一个 ClearCase update 命令	org.apache.tools.ant.taskdefs.optional .clearcase.CCUpdate
执行一个 ClearCase <i>update</i> 命令。	

属性**cleartooldir (1.3, 1.4, String, N)**指定 *cleartool* 所在的目录。**currenttime (1.3, 1.4, boolean, *)**如果为 *true*, 此任务将把文件修改时间设置为当前系统时间。默认为 *false*。

graphical (1.3, 1.4, boolean, N)

如果为 true，则此任务显示 GUI 窗口。默认为 false。

log (1.3, 1.4, String, N)

日志文件名，ClearCase 应当写到此日志文件。

overwrite (1.3, 1.4, boolean, N)

如果为 true，则此任务重写被截获的文件（注 2）。默认为 false。

preservetime (1.3, 1.4, boolean, *)

如果为 true，则此任务保留来自 VOB (Version Object Base) 的文件修改时间。默认为 false。

rename (1.3, 1.4, boolean, N)

如果为 true，则表示被截获的文件应当重命名为带有一个 .keep 扩展名。默认为 false。

viewpath (1.3, 1.4, String, N)

指定 ClearCase 视图文件或目录的路径。

currenttime 和 preservetime 属性是可选的，但不能同时指定。

内容

无。

CSC

1.3, 1.4

编译 C# 代码

org.apache.tools.ant.taskdefs.optional.dotnet.CSharp

编译 C# 源代码。目前可在包含 *csc.exe* 可执行程序的平台（即各种类型的 Windows 上）工作。

注 2：如果修改了一个文件但未将其写出，则 ClearCase 认为此文件为“被截获”。

属性

`additionalmodules (1.3, 1.4, String, N)`

用分号分隔的附加模块列表，它们均为不包括元数据的 DLL。这相当于 *csc* 的 */addmodule* 参数。

`debug (1.3, 1.4, boolean, N)`

如果为 `true`，则包括调试信息。默认为 `true`。

`defaultexcludes (1.3, 1.4, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`definitions (1.3, 1.4, String, N)`

传递给 *csc.exe* 的定义列表，用“;”、“:”或“,”分隔。例如，`DEBUG;BETA_TEST`。

`docfile (1.3, 1.4, File, N)`

用于生成 XML 文档的文件的名字。

`excludes (1.3, 1.4, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (1.3, 1.4, File, N)`

每行包括一个排除模式的文件名。

`extraoptions (1.3, 1.4, String, N)`

直接传递给 *csc.exe* 命令的附加选项。

`failonerror (1.3, 1.4, boolean, N)`

如果为 `true`，当编译返回一个错误时，此任务将使构建失败。默认为 `true`。

`fullpaths (1.4, boolean, N)`

如果为 `true`，出现错误时，此任务将打印文件的完全路径。默认为 `true`。

`includedefaultreferences (1.3, 1.4, boolean, N)`

如果为 `true`，则此任务包括 .NET beta 1 中以及 *mscore.dll* 的链接中常见的程序集。默认为 `true`。

`includes (1.3, 1.4, String, N)`

要包含的文件模式列表（用逗号分隔）。

`includesfile (1.3, 1.4, File, N)`

每行包括一个包含模式的文件的文件名。

`incremental (1.3, 1.4, boolean, N)`

如果为 `true`，则此任务指示 `csc.exe` 完成一个递增的构建。默认为 `false`。

`mainclass (1.3, 1.4, String, N)`

可执行程序的主类名。

`noconfig (1.4, boolean, N)`

如果为 `true`，则此任务向 `csc.exe` 传递 `/noconfig` 标志。默认为 `false`。

`optimize (1.3, 1.4, boolean, N)`

如果为 `true`，则此任务指示 `csc.exe` 进行优化。默认为 `false`。

`outputfile (1.3, 1.4, File, N)`

目标文件名，如 `mygui.exe`。

`referencefiles (1.3, 1.4, Path, N)`

要包括的引用路径。

`references (1.3, 1.4, String, N)`

所要引用的 `.dll` 文件列表（用分号分隔）。

`srcdir (1.3, 1.4, File, N)`

包含源代码的目录。

`targettype (1.3, 1.4, String, N)`

指定目标类型。可允许的值为 `exe`、`module`、`winexe` 和 `library`。默认为 `exe`。

`unsafe (1.3, 1.4, boolean, N)`

如果为 `true`，则此任务启用 `unsafe` 关键字。默认为 `false`。

`utf8output (1.4, boolean, N)`

如果为 `true`，此任务对输出文件使用 UTF-8 编码。默认为 `false`。

`warnlevel (1.3, 1.4, int, N)`

警告级别，范围为 1 到 4，在此 4 最严格。默认为 3。

`win32icon (1.3, 1.4, File, N)`

所要使用的图标的文件名，如 `foo.ico`。

`win32res (1.4, File, N)`

Win32 资源的文件名，如 `foo.res`。

内容

`0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (1.3, 1.4); <excludesfile>、<includesfile> (1.4)`

用以替代其相应属性，它们指定了包含和排除的源文件组。

ddcreator

`all`

创建 EJB 部署描述文件

`org.apache.tools.ant.taskdefs.optional.ejb.DDCreator`

由文本文件创建串行化 EJB 部署描述文件。此任务为 BEA WebLogic Server 4.5.1 而设计。

属性

`classpath (all, String, N)`

运行 `weblogic.ejb.utils.DDCreator` 类所用的类路径。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`descriptors (all, String, Y)`

包含文本部署描述文件的基目录。

`dest (all, String, Y)`

目标目录。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件的文件名。

`includes (all, String, N)`

要包含的文件模式列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

内容

*0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all);
<excludesfile>、<includesfile> (1.4)*

用以替代其相应属性，它们指定了包含和排除的源文件组。

depend

1.3, 1.4

利用依赖关系查找 / 删除过时的文件 `org.apache.tools.ant.taskdefs.optional.depend.Depend`

将类文件的时间戳与源文件时间戳加以比较，并基于对内容的分析确定哪些类已经过时。然后将过时的类文件删除，其原因有可能是源文件更新，也可能是某些逻辑依赖关系已经改变。例如，修改基类的源代码就会导致所有派生类都被删除，因为它们均对其基类有一个逻辑依赖关系。

属性

`cache (1.3, 1.4, File, N)`

此任务缓存依赖关系信息的目录。如果省略，则不使用缓存。

`classpath (1.4, Path, N)`

指定附加的类（相对于 `destdir`）和 JAR 文件置于何处。此任务将检查对于由此属性指定的类的依赖关系。在这个路径中不能包括第三方库和 JDK 库，

因为它们很少改变，而且会使依赖关系分析速度减慢。例如，如果希望检查对于一个实用程序 JAR 文件的依赖关系，这就很有用。

`classpathref (1.4, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`closure (1.3, 1.4, boolean, N)`

如果为 `true`，此任务仅删除那些直接依赖于过时类的类文件。否则，还将考虑间接的依赖关系。默认为 `false`。

`defaultexcludes (1.3, 1.4, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`destdir (1.3, 1.4, Path, N)`

类文件所在目录。如果省略，则默认为 `srcdir` 指定的值。

`dump (1.4, boolean, N)`

如果为 `true`，此任务将把依赖关系信息写到日志输出中。默认为 `false`。

`excludes (1.3, 1.4, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (1.3, 1.4, File, N)`

每行包括一个排除模式的文件的文件名。

`includes (1.3, 1.4, String, N)`

要包含的文件模式列表（用逗号分隔）。

`includesfile (1.3, 1.4, File, N)`

每行包括一个包含模式的文件的文件名。

`srcdir (1.3, 1.4, Path, Y)`

包含源文件的目录。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.4)`

`Path` 元素，用以代替 `classpath` 和 `classpathref` 属性。

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (1.3, 1.4); <excludesfile>、<includesfile> (1.4)

用以替代其相应属性，它们指定了包含和排除的源文件组。

ejbc

all

执行 WebLogic ejbc 工具

org.apache.tools.ant.taskdefs.optional.ejb.Ejbc

执行 BEA WebLogic Server 的 ejbc 工具，生成在该环境中部署 EJB 组件所需的代码。此任务是为 WebLogic 4.5.1 所设计的。

属性

classpath (all, String, N)

所用的类路径。必须包括所有必需的支持类，如 bean 的远程和本地接口。

defaultexcludes (all, boolean, N)

确定是否使用默认排除模式。默认为 true。

descriptors (all, String, Y)

包含串行化部署描述文件的基目录。

dest (all, String, Y)

存放所生成类、存根和骨架的目标目录。

excludes (all, String, N)

要排除的文件模式列表（用逗号分隔）。

excludesfil (all, File, N)

每行包括一个排除模式的文件的文件名。

includes (all, String, N)

要包含的文件模式列表（用逗号分隔）。

includesfile (all, File, N)

每行包括一个包含模式的文件的文件名。

`keepgenerated (1.3, 1.4, String, N)`

如果为 `true`, 保存所生成的 Java 源代码。默认为 `false`。注意这是一个 `String` 属性, 而不是 Boolean; 合法值为 `true` 和 `false`。

`manifest (all, String, Y)`

要创建的清单文件的文件名。

`src (all, String, Y)`

源树的基, 包括本地接口、远程接口和 bean 实现类。

内容

0 到 n 个嵌套 `patternset` 元素: `<exclude>`、`<include>`、`<patternset>` (`all`);
`<excludesfile>`、`<includesfile>` (1.4)

用以替代其相应属性, 它们指定了所包含和排除的部署描述文件。

ejbjar

all

创建 EJB JAR 文件

org.apache.tools.ant.taskdefs.optional.ejb.EjbJar

创建与 EJB 1.1 兼容的 `ejb-jar` 文件。此任务支持一组通用属性和嵌套元素, 以及多个厂商专有的嵌套元素。

属性

`basejarname (all, String, *)`

用于所生成 JAR 文件名的基本名。

`basenameterminator (all, String, N)`

用于基于部署描述文件名来确定文件名。例如, 假设此属性设置为 “-”。利用这个值, 一个名为 `Customer-ejb-jar.xml` 的部署描述文件就会得到基本名 `Customer`。此属性仅在指定了 `basejarname` 时方可使用。默认为 “-”。

`classpath (1.3, 1.4, Path, N)`

一个类路径, 用于放置要增加到 `ejb-jar` 文件中的文件。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式。默认为 true。

`descriptordir (all, File, N)`

包含部署描述文件的目录树的基。默认为 `srcdir` 指定的值。

`destdir (all, File, Y)`

所生成 JAR 文件的目标目录。文件将放在与部署描述文件子目录相对应的子目录中。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`flatdestdir (all, boolean, N)`

当所生成 JAR 文件不能放在子目录中时，用此来代替 `destdir`。

`genericjarsuffix (all, String, N)`

生成通用 `ejb-jar` 文件时追加到部署描述文件基本名后的名字。默认为 `-generic.jar`。

`includes (all, String, N)`

要包含的文件模式列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`manifest (1.4, File, N)`

指定要包含的一个清单文件。

`naming (1.4, Enum, *)`

配置如何确定 JAR 文件名。默认为 `descriptor`。合法值如下：

`ejb-name`

JAR 文件名基于 EJB 名。例如，`Customer bean` 就会得到 `Customer.jar`。

directory

JAR 文件名基于包含部署描述文件的目录的最后一部分。例如，*com/oreilly/sales/accounting-ejb-jar.xml* 将成为 *sales.jar*。

descriptor

JAR 文件名基于部署描述文件名。例如，*com/oreilly/sales/accounting-ejb-jar.xml* 将变成 *accounting.jar*。

basejarname

只有当 `basejarname` 属性设置时才合法。设置此值时，`ejbjar` 任务在确定 JAR 文件名时将使用 `basejarname` 属性值。例如，若 `basejarname` 为“Book”，则所得到的 JAR 为 *Book.jar*。

srcdir (all, File, Y)

包含组成 bean 的类文件的目录。

仅当 `naming="basejarname"` 时才允许 `basejarname` 属性。

内容

0 到 n 个嵌套 `patternset` 元素：`<exclude>`、`<include>`、`<patternset>` (all);
`<excludesfile>`、`<includesfile>` (1.4)

用以替代其相应属性，它们指定了所包含和排除的部署描述文件。

0 或 1 个嵌套 `<classpath>` 元素 (1.3, 1.4)

用以代替 `classpath` 属性。

0 到 n 个嵌套 `<dtd>` 元素 (1.3, 1.4)

指定 XML 文件中 DTD 引用的本地位置。这很有用，因为由一个本地文件加载 DTD 比远程加载 DTD 要快，而且在与网络断开连接或运行于防火墙之后也能正常。此元素需要以下两个属性：

`location (1.3, 1.4, String, Y)`

DTD 的本地拷贝。要么是一个文件名，要么是一个 Java 资源名。

`publicid (1.3, 1.4, String, Y)`

DTD 的公共 ID。

0 到 n 个嵌套 <support> 元素 (1.3, 1.4)

`fileset` 元素，指定了要包括在所生成 JAR 文件中的附加文件。当生成多个 JAR 文件时，这些支持文件要分别加到各个 JAR 文件中。

`ejbar` 任务支持许多厂商专有的嵌套元素。这就造成了不同 EJB 服务器之间的部署差异。

0 或 1 个嵌套 <borland> 元素 (1.4)

支持 Borland Application Server Version 4.5。生成和编译存根和骨架，创建 JAR 文件，然后验证其内容。属性如下：

`classpath (1.4, Path, N)`

生成存根和骨架时所用的类路径。它要追加到 `ejbjar` 父类中指定的类路径后面。还支持嵌套 `<classpath>` 元素。

`debug (1.4, boolean, N)`

如果为 `true`，则以调试模式运行 Borland 工具。默认为 `false`。

`destdir (1.4, File, Y)`

所生成 JAR 文件的基目录。

`generateclient (1.4, boolean, N)`

如果为 `true`，则生成相应的客户 JAR 文件。默认为 `false`。

`suffix (1.4, String, N)`

追加至部署描述文件基本名后的文本。它用于生成 JAR 文件名。默认为 `-ejb.jar`。

`verify (1.4, boolean, N)`

如果为 `true`，则验证所生成 JAR 文件。默认为 `false`。

`verifyargs (1.4, String, N)`

`verify=true` 时所用的附加参数。

0 或 1 个嵌套<iplanet>元素 (1.4)

支持 iAS (iPlanet Application Server) 6.0 版本。属性如下：

classpath (1.4, Path, N)

生成存根和骨架时所用的类路径。它要追加到 ejbjar 父类中指定的类路径后面。还支持嵌套 <classpath> 元素。

debug (1.4, boolean, N)

如果为 true, ejbjar 任务运行时将把调试信息记入日志。默认为 false。

destdir (1.4, File, Y)

所生成 JAR 文件的基目录。

iashome (1.4, File, N)

iAS 发布的主目录，当 ejbc 实用程序不在路径中时，用于查找 ejbc 实用程序。

keepgenerated (1.4, boolean, N)

如果为 true，则保存所生成的 Java 源文件。默认为 false。

suffix (1.4, String, N)

追加至每个所生成的 JAR 文件名后。默认为 .jar。

0 或 1 个嵌套<jboss>元素 (1.4)

支持 JBoss 服务器。由于 JBoss 支持热部署 (hot deployment)，所以它不需要所生成的存根和骨架。此任务将查找 jboss.xml 和 jaws.xml，将其增加到所生成的 JAR 文件中。支持一个属性：

destdir (1.4, File, Y)

所生成 JAR 文件的基目录。

0 或 1 个嵌套<weblogic>元素 (all)

支持 weblogic.ejbc 编译器。支持以下属性：

args (all, String, N)

weblogic.ejbc 的附加参数。

`classpath (all, Path, N)`

运行 `weblogic.ejbc` 工具时所用的类路径。

`compiler (all, String, N)`

选择一个不同的 Java 编译器。

`destdir (all, File, Y)`

所生成 JAR 文件的基目录。

`genericjarsuffix (all, String, N)`

生成一个中间临时 JAR 文件时所用的文件名后缀。默认为 `-generic.jar`。

`keepgenerated (all, boolean, N)`

如果为 `true`, 则保存所生成的 Java 文件。默认为 `false`。

`keepgeneric (all, boolean, N)`

如果为 `true`, 则保存中间通用 JAR 文件。默认为 `false`。

`newCMP (1.3, 1.4, boolean, N)`

如果为 `true`, 则使用新方法以找到 CMP 描述文件。默认为 `false`。

`noEJBC (1.4, boolean, N)`

如果为 `true`, 则不在 JAR 文件上运行 `weblogic.ejbc`。默认为 `false`。

`rebuild (1.3, 1.4, boolean, N)`

如果为 `true`, 则强制执行 `weblogic.ejbc` 而不检查文件时间戳。默认为 `true`。

`suffix (all, String, N)`

追加至每个所生成的 JAR 文件名后。默认为 `.jar`。

`wlclasspath (1.3, 1.4, Path, N)`

当某个 bean 的本地和远程接口位于运行 WebLogic 6.0 所用的类路径上时, 使用此属性可以避免发出一个警告。设置此属性以包括标准 WebLogic 类, 并使用 `classpath` 属性来包括与 bean 相关的类。

`0 或 1 个嵌套 <weblogictoplink> 元素 (all)`

连同 WebLogic for CMP 使用 TOPLink 时会用到此元素。此任务支持所有 `<weblogic>` 属性, 以及以下附加的属性:

`toplinkdescriptor (all, String, Y)`

本地存储的 TOPLink 部署描述文件的文件名。它与其包含 ejbjar 任务的 `descriptordir` 属性相关。

`toplinkdtd (all, String, N)`

TOPLink DTD 文件的位置。建议是一个本地文件路径 URL，但并非必须如此。默认为 `http://www.objectpeople.com` 处的 DTD。

<code>ftp</code>	<code>all</code>
实现一个 FTP 客户	<code>org.apache.tools.ant.taskdefs.optional.net.FTP</code>
实现一个基本 FTP 客户。此任务依赖于 <code>netcomponents.jar</code> ，可在 <code>http://www.savarese.org/oro/downloads</code> 得到。	

属性

`action (all, String, N)`

要执行的 FTP 命令。合法值为 `send`、`put`、`recv`、`get`、`del`、`delete`、`list` 和 `mkdir`。默认为 `send`。

`binary (all, boolean, N)`

如果为 `true`，则使用二进制模式传输而不是文本模式。默认为 `true`。

`depends (all, boolean, N)`

如果为 `true`，则仅传输新的或已修改的文件。默认为 `false`。

`ignorenoncriticalerrors (1.4, boolean, N)`

如果为 `true`，则忽略由一些 FTP 服务器发送的不重要的错误码。默认为 `false`。

`listing (all, File, *)`

文件名，该文件用于存储由 `list` 活动得到的输出。

`newer (all, boolean, N)`

`depends` 的别名。

`passive (1.3, 1.4, boolean, N)`

如果为 `true`, 则使用被动模式传输。默认为 `false`。

`password (all, String, Y)`

FTP 服务器的登录口令。

`port (all, int, N)`

端口号。默认为 21。

`remotedir (all, String, N)`

远程服务器上的一个目录。

`separator (all, String, N)`

FTP 服务器上的文件分隔符。默认为 “/”。

`server (all, String, Y)`

远程服务器的 URL。

`skipfailedtransfers (1.4, boolean, N)`

如果为 `true`, 即使出现某些失败情况也继续传输文件。默认为 `false`。

`userid (all, String, Y)`

FTP 服务器的登录 ID。

`verbose (all, boolean, N)`

如果为 `true`, 则以详细模式操作。默认为 `false`。

当 `action="list"` 时, `listing` 属性是必要的。

内容

`0 到 n 个嵌套 <fileset> 元素 (all)`

指定传输所要包含和排除的文件和目录。

icontract

1.4

执行 iContract 预处理程序

`org.apache.tools.ant.taskdefs.optional.IContract`

执行 iContract 合约设计预处理程序。iContract 可以在 <http://www.reliable->

`systems.com/tools/` 可以得到。此任务使用 `javac` 任务所用的 Java 编译器。通过设置 `build.compiler` 特性指定其他的 Java 编译器。

属性

`builddir (1.4, File, N)`

已编译且已装配类 (instrumented) 的目标。Ant 用户手册警告称要避免已装配类和未装配类使用相同的目录，因为这会破坏依赖关系检查。

`classdir (1.4, File, N)`

包含已编译、未装配类的源目录。

`classpath (1.4, Path, N)`

装配和编译文件时所用的类路径。

`classpathref (1.4, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`controlfile (1.4, File, *)`

传递到 iContract 的控制文件的文件名。

`defaultexcludes (1.4, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`excludes (1.4, String, N)`

要排除的文件模式列表 (用逗号分隔)。

`excludesfile (1.4, File, N)`

每行包括一个排除模式的文件的文件名。

`failthrowable (1.4, String, N)`

断言非法时抛出的异常或错误的类名。默认为 `java.lang.Error`。

`includes (1.4, String, N)`

要包含的文件模式列表 (用逗号分隔)。

`includesfile (1.4, File, N)`

每行包括一个包含模式的文件的文件名。

`instrumentdir (1.4, File, Y)`

已装配源文件的目标目录。

`invariant (1.4, boolean, N)`

如果为 `true`, 则装配不变量。除非指定了 `controlfile`, 否则默认为 `true`。

`post (1.4, boolean, N)`

如果为 `true`, 则装配滞后条件。除非指定了 `controlfile`, 否则默认为 `true`。

`pre (1.4, boolean, N)`

如果为 `true`, 则装配前提条件。除非指定了 `controlfile`, 否则默认为 `true`。

`quiet (1.4, boolean, N)`

如果为 `true`, 则以安静模式执行。默认为 `false`。

`repbuilddir (1.4, File, N)`

编译存储库类的目标目录。默认为 `repositorydir` 指定的值。

`repositorydir (1.4, File, Y)`

存储库源文件的目标目录。

`srcdir (1.4, File, Y)`

原始 Java 源文件的位置。

`targets (1.4, File, N)`

此任务创建的一个文件的文件名, 其中列出 iContract 将装配的所有类。如果指定, 执行后此文件不会被删除。否则将创建一个临时文件, 然后在执行后删除。

`updateicontrol (1.4, boolean, N)`

如果为 `true`, 则更新当前目录中 iControl 的特性文件, 或者在必要时创建一个新的特性文件。默认为 `false`。

`verbosity (1.4, String, N)`

用逗号分隔的显示级别列表。允许 `error*`、`warning*`、`note*`、`info*`、`progress*` 和 `debug*` 的任意组合。默认为 `error*`。

如果 updateicontrol=true，则 controlfile 属性是必要的。

内容

0 到 n 个嵌套 patternset 元素: <exclude>、<excludesfile>、<include>、<includesfile> 和<patternset> (1.4)

用以替代其相应属性，它们指定了包含和排除的源文件组。

0 或 1 个嵌套 <classpath> 元素 (1.4)

可用以代替 classpath 和 classpathref 属性。

ilasm

1.3, 1.4

汇编 .NET 中间语言文件

org.apache.tools.ant.taskdefs.optional.dotnet.ilasm

汇编 .NET 中间语言文件。仅在 Windows 上工作；路径中必须有 *csc.exe* 和 *ilasm.exe*。

属性

debug (1.3, 1.4, boolean, N)

如果为 true，则包括调试信息。默认为 true。

defaultexcludes (1.3, 1.4, boolean, N)

确定是否使用默认排除模式。默认为 true。

excludes (1.3, 1.4, String, N)

要排除的文件模式列表（用逗号分隔）。

excludesfile (1.3, 1.4, File, N)

每行包括一个排除模式的文件的文件名。

extraoptions (1.3, 1.4, String, N)

直接传递给 *csc.exe* 命令的附加选项。

failonerror (1.3, 1.4, boolean, N)

如果为 true，则当此任务失败时构建将失败。默认为 true。

`includes (1.3, 1.4, String, N)`

要包含的文件模式列表（用逗号分隔）。

`includesfile (1.3, 1.4, File, N)`

每行包括一个包含模式的文件的文件名。

`keyfile (1.4, File, N)`

包含一个私钥的文件的文件名。

`listing (1.3, 1.4, boolean, N)`

如果为 `true`，则产生一个送至当前输出流的列表。默认为 `false`。

`outputfile (1.3, 1.4, File, N)`

目标文件名，如 `mygui.exe`。

`owner (1.3, 1.4, String, N)`

将 `/owner` 参数指定为 `ilasm.exe`。

`resourcefile (1.3, 1.4, File, N)`

要包含的资源文件的文件名。

`srcdir (1.3, 1.4, File, N)`

包含源文件的目录。

`targettype (1.3, 1.4, String, N)`

指定目标类型。可允许的值为 `exe` 和 `library`（创建一个 DLL）。默认为 `exe`。

`verbose (1.3, 1.4, boolean, N)`

如果为 `true`，则以详细模式操作。默认为 `false`。

内容

0 到 n 个嵌套 `patternset` 元素：`<exclude>`、`<include>`、`<patternset>` (1.3, 1.4)；`<excludesfile>`、`<includesfile>` (1.4)

用以替代其相应属性，它们指定了包含和排除的源文件组。

iplanet-ejbc

1.4

为 iPlanet 服务器编译 EJB 存根 org.apache.tools.ant.taskdefs.optional.ejb.IPlanetEjbcTask

为 iPlanet Application Server 6.0 版本编译 EJB 存根和骨架。

属性**classpath (1.4, Path, N)**

生成存根和骨架时所用的类路径。

debug (1.4, boolean, N)

如果为 true，则将附加的调试信息记入日志。默认为 false。

dest (1.4, File, Y)

生成的存根和骨架所置于的基目录。bean 的类文件、本地接口和远程接口也必须在此目录中，而且任务执行前此目录必须存在。

ejbdescriptor (1.4, File, Y)EJB 1.1 部署描述文件的位置，此文件通常名为 *ejb-jar.xml*。**iasdescriptor (1.4, File, Y)**iPlanet EJB 部署描述文件，此文件通常名为 *ias-ejb-jar.xml*。**iashome (1.4, File, N)**

iPlanet 发布的主目录。

keepgenerated (1.4, boolean, N)

如果为 true，则此任务将保存所生成的 Java 源代码。默认为 false。

内容**0 或 1 个嵌套 <classpath> 元素 (1.4)**

可用以代替 classpath 属性。

javacc	all
基于一个语法文件执行 JavaCC	org.apache.tools.ant.taskdefs.optional.javacc.JavaCC
基于一个语法文件执行 JavaCC 编译器。JavaCC 可于 http://www.webgain.com/products/java_cc/ 得到。	

属性

`buildparser (all, boolean, N)`

如果指定，此任务将 BUILD_PARSER 语法选项设置为此属性的值。

`buildtokenmanager (all, boolean, N)`

如果指定，此任务将 BUILD_TOKEN_MANAGER 语法选项设置为此属性的值。

`cachetokens (all, boolean, N)`

如果指定，此任务将 CACHE_TOKENS 语法选项设置为此属性的值。

`choiceambiguitycheck (all, int, N)`

如果指定，此任务将 CHOICE_AMBIGUITY_CHECK 语法选项设置为此属性的值。

`commontokenaction (all, boolean, N)`

如果指定，此任务将 COMMON_TOKEN_ACTION 语法选项设置为此属性的值。

`debuglookahead (all, boolean, N)`

如果指定，此任务将 DEBUG_LOOKAHEAD 语法选项设置为此属性的值。

`debugparser (all, boolean, N)`

如果指定，此任务将 DEBUG_PARSER 语法选项设置为此属性的值。

`debugtokenmanager (all, boolean, N)`

如果指定，此任务将 DEBUG_TOKEN_MANAGER 语法选项设置为此属性的值。

`errorreporting (all, boolean, N)`

如果指定，此任务将 ERROR_REPORTING 语法选项设置为此属性的值。

`forcecheck (all, boolean, N)`

如果指定，此任务将 FORCE_LA_CHECK 语法选项设置为此属性的值。

`ignorecase (all, boolean, N)`

如果指定，此任务将 IGNORE_CASE 语法选项设置为此属性的值。

`javacchome (all, File, Y)`

包含 JavaCC 发布的目录。

`javaunicodeescape (all, boolean, N)`

如果指定，此任务将 JAVA_UNICODE_ESCAPE 语法选项设置为此属性的值。

`lookahead (all, int, N)`

如果指定，此任务将 LOOKAHEAD 语法选项设置为此属性的值。

`optimizetokenmanager (all, boolean, N)`

如果指定，此任务将 OPTIMIZE_TOKEN_MANAGER 语法选项设置为此属性的值。

`otherambiguitycheck (all, int, N)`

如果指定，此任务将 OTHER_AMBIGUITY_CHECK 语法选项设置为此属性的值。

`outputdirectory (all, File, N)`

所生成文件的目标目录。默认为包含语法文件的目录。

`sanitycheck (all, boolean, N)`

如果指定，此任务将 SANITY_CHECK 语法选项设置为此属性的值。

`static (all, boolean, N)`

如果指定，此任务将 STATIC 语法选项设置为此属性的值。

`target (all, File, Y)`

要处理的语法文件。

`unicodeinput (all, boolean, N)`

如果指定，此任务将 UNICODE_INPUT 语法选项设置为此属性的值。

`usercharstream (all, boolean, N)`

如果指定，此任务将 USE_CHAR_STREAM 语法选项设置为此属性的值。

`usertokenmanager (all, boolean, N)`

如果指定，此任务将 USER_TOKEN_MANAGER 语法选项设置为此属性的值。

内容

无。

javah 1.3, 1.4

执行 `javah` 工具 `org.apache.tools.ant.taskdefs.optional.Javah`

执行 `javah` 工具，由一个或多个 Java 类生成 JNI (Java Native Interface, Java 本地接口) 首部。此任务使用默认的 Java 编译器，除非 `build.compiler` 特性设置为其他值，在第七章对 `javac` 任务的描述中已做解释。

属性

`bootclasspath` (1.3, 1.4, Path, N)

所用的启动 (bootstrap) 类路径。

`bootclasspathref` (1.3, 1.4, Reference, N)

在构建文件中某处定义的启动类路径的引用。

`class` (1.3, 1.4, String, Y)

要处理的类名列表 (用逗号分隔)。

`classpath` (1.3, 1.4, Path, N)

所用的类路径。

`classpathref` (1.3, 1.4, Reference, N)

在构建文件中某处定义的一个类路径的引用。

`destdir` (1.3, 1.4, File, *)

所生成文件的目标目录。

`force` (1.3, 1.4, boolean, N)

如果为 `true`，此任务总会写输出文件。默认为 `false`。

`old` (1.3, 1.4, boolean, N)

如果为 `true`，则此任务使用 JDK 1.0 形式的头文件。默认为 `false`。

outputfile (1.3, 1.4, File, *)

如果指定以代替 `destdir`, 则此任务将把所有输出都连接到一个文件中。

stubs (1.3, 1.4, boolean, N)

用于 `old=true` 时。如果为 `true`, 此任务生成 C 声明。默认为 `false`。

verbose (1.3, 1.4, boolean, N)

如果为 `true`, 则此任务以详细模式执行 `javadoc`。默认为 `false`。

`outputfile` 或 `destdir` 中必取其一。

内容

0 或 1 个嵌套 <bootclasspath> 元素 (1.3, 1.4)

可用以代替 `bootclasspath` 或 `bootclasspathref` 属性。

0 到 n 个嵌套 <class> 元素 (1.3, 1.4)

可用以代替 `class` 属性来指定要生成的类。每个 `<class>` 元素都有一个必要的 `name` 属性。例如:

```
<class name="com.oreilly.util.Foobar"/>
```

0 或 1 个嵌套 <classpath> 元素 (1.3, 1.4)

`Path` 元素, 用以代替 `classpath` 或 `classpathref` 属性。

jdepend

1.4

执行 JDepend 工具 `org.apache.tools.ant.taskdefs.optional.jdepend.JDependTask`

执行 JDepend 工具。JDepend 将分析 Java 源文件, 为每个包生成设计质量测量结果。此任务要求 JDepend 1.2 及以后版本, 可在 <http://www.clarkware.com/software/JDepend.html> 得到。

属性

classpath (1.4, Path, N)

所用的类路径。

`classpathref (1.4, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`dir (1.4, File, N)`

JVM 的工作目录。

`fork (1.4, boolean, N)`

如果为 `true`, 则此任务创建一个新的 JVM 实例。默认为 `false`。

`haltonerror (1.4, boolean, N)`

如果为 `true`, 则当出现错误时, 此任务终止构建。默认为 `false`。

`jvm (1.4, String, N)`

此命令用于调用 JVM 的命令, 如果 `fork=false`, 则忽略。默认为 `java`。

`outputfile (1.4, File, N)`

任务将把输出发送至此文件, 如果省略, 则发送到当前输出流。

内容

0 或 1 个嵌套 <classpath> 元素 (1.4)

一个 `path` 元素, 用以代替 `classpath` 或 `classpathref` 属性。

0 到 n 个嵌套 <jvmarg> 元素 (1.4)

如第四章所述的命令行参数。仅当 `fork=true` 时合法。

1 到 n 个嵌套 <sourcespath> 元素 (1.4)

定义 Java 源文件置于何处的 `path` 元素。

jjtree

all

执行 JJTree 预处理程序

org.apache.tools.ant.taskdefs.optional.javacc.JJTree

为 JavaCC 编译器执行 JJTree 预处理程序, 前者可在 http://www.webgain.com/products/java_cc/ 得到。

属性

`buildnodefiles (all, boolean, N)`

如果指定，此任务将 BUILD_NODE_FILES 语法选项设置为此属性的值。

`javacchome (all, File, Y)`

包含 JavaCC 发布的目录。

`multi (all, boolean, N)`

如果指定，此任务将 MULTI 语法选项设置为此属性的值。

`nodedefaultvoid (all, boolean, N)`

如果指定，此任务将 NODE_DEFAULT_VOID 语法选项设置为此属性的值。

`nodefactory (all, boolean, N)`

如果指定，此任务将 NODE_FACTORY 语法选项设置为此属性的值。

`nodepackage (all, String, N)`

如果指定，此任务将 NODE_PACKAGE 语法选项设置为此属性的值。

`nodeprefix (all, String, N)`

如果指定，此任务将 NODE_PREFIX 语法选项设置为此属性的值。

`nodescopehook (all, boolean, N)`

如果指定，此任务将 NODE_SCOPE_HOOK 语法选项设置为此属性的值。

`nodeusesparser (all, boolean, N)`

如果指定，此任务将 NODEUSES_PARSER 语法选项设置为此属性的值。

`outputdirectory (all, File, N)`

所生成文件的目标目录。如果省略，输出将写到包含语法文件的目录。

`static (all, boolean, N)`

如果指定，此任务将 STATIC 语法选项设置为此属性的值。

`target (all, File, Y)`

要处理的 JJTree 语法文件。

`visitor (all, boolean, N)`

如果指定，此任务将 VISITOR 语法选项设置为此属性的值。

`visitorexception (all, String, N)`

如果指定，此任务将 VISITOR_EXCEPTION 语法选项设置为此属性的值。

内容

无。

jlink

`all`

创建 / 合并 JAR 或 ZIP 文件，`org.apache.tools.ant.taskdefs.optional.jlink.JlinkTask`

构建一个 JAR 或 ZIP 文件，可选地与现有 JAR 和 ZIP 压缩文件的内容合并。若存在重复文件，则接受第一个文件，其余的则被忽略。在合并压缩文件时，现有的 `META-INF` 目录将被忽略。

属性

`addfiles (all, Path, *)`

要增加到压缩文件中的一个文件列表。即使它们本身是 JAR 或 ZIP 压缩文件，也要作为单个的文件加入。要合并现有 JAR 或 ZIP 压缩文件的内容，则要使用 `mergefiles` 属性或嵌套元素。

`compress (all, boolean, N)`

如果为 `true`，则压缩输出。默认为 `false`。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件的文件名。

`includes (all, String, N)`

要包含的文件模式列表（用逗号分隔）。

includesfile (all, File, N)

要包含的文件模式列表（用逗号分隔）。

mergefiles (all, Path, *)

要增加到压缩文件中的一个文件列表。*.jar* 和 *.zip* 文件的内容要合并到输出压缩文件中，而不是作为 JAR 和 ZIP 文件增加进来。

outfile (all, File, Y)

目标压缩文件，例如，*myproj.jar*。

addfiles 或 **mergefiles** 至少取其一，或者有相应的嵌套元素。

内容

0 到 n 个嵌套 patternset 元素：<exclude>、<include>、<patternset> (all); <excludesfile>、<includesfile> (1.4)

用以替代其相应属性，它们指定了包含和排除的源文件组。

0 到 n 个嵌套 <addfiles> 元素 (all)

Path 元素，用以代替 **addfiles** 属性。即使以 *.jar* 或 *.zip* 作为后缀，各项也要作为单独的文件加入。例如，*utils.jar* 要作为一个独立的文件增加到压缩文件中，而不会展开。若路径指向目录，则子目录中的所有文件都将递归地加入。

0 到 n 个嵌套 <mergefiles> 元素 (all)

可用以代替 **mergefiles** 属性。除非以 *.jar* 或 *.zip* 作为后缀，否则各项要作为单独的文件加入。在这种情况下（即以 *.jar* 或 *.zip* 作为后缀），压缩文件的内容会被取出，然后增加到新的压缩文件中。*META-INF* 目录将被忽略。

jpcov

1.4

执行 JProbe Coverage 工具

org.apache.tools.ant.taskdefs.optional.sitraka.Coverage

执行 JProbe Coverage 工具，它会运行一个类并分析执行了哪些代码行。此任务设计用于同 JProbe Suite Server Side 3.0 版本一同使用，后者可在 <http://www.sitraka.com> 得到。

属性

`applet (1.4, boolean, N)`

如果为 true，则表示 `classname` 属性指定一个 applet。默认为 false。

`classname (1.4, String, Y)`

要分析的类。

`exitprompt (1.4, String, N)`

对控制台何时提示 “Press Enter to close this window (按回车键关闭此窗口)” 加以控制。合法值为 always、never 和 error。默认为 never。

`finalsnapshot (1.4, String, N)`

配置程序结束时所取快照的类型。合法值为 none、coverage 和 all。默认为 coverage。

`home (1.4, File, Y)`

JProbe 所安装的目录。

`inputfile (1.4, File, N)`

JProbe Coverage 参数文件。如果指定，则所有其他属性均被忽略。

`javaexe (1.4, File, N)`

java 可执行程序的路径。仅用于 `vm="java2"` 时。

`recordfromstart (1.4, Enum, N)`

对程序开始时所完成的分析进行配置。合法值为 none、coverage 和 all。默认为 coverage。

`seedname (1.4, String, N)`

临时快照文件的基本名。如果此属性为 foo，则文件命名为 `foo.jpc`、`foo1.jpc`、`foo2.jpc` 等等。默认为 snapshot。

`snapshotdir (1.4, File, N)`

快照的目标目录。默认为工程的基目录。

`tracknatives (1.4, boolean, N)`

如果为 true，则此任务跟踪本地方法。默认为 false。

`vm (1.4, Enum, N)`

指定任务所运行的 JVM。合法值为 `jdk117`、`jdk118` 和 `java2`。

`warnlevel (1.4, int, N)`

指定警告级别，范围为 0 到 3。默认为 0，这将得到最少数量的警告。

`workingdir (1.4, File, N)`

JVM 的工作目录。

内容

0 到 n 个嵌套 <arg> 元素 (1.4)

命令行应用参数，如第四章所述。

0 或 1 个嵌套 <classpath> 元素 (1.4)

指定类路径的 `path` 元素。

0 到 n 个嵌套 <jvmarg> 元素 (1.4)

命令行 JVM 参数，如第四章所述。

0 或 1 个嵌套 <filters> 元素 (1.4)

为类和方法定义 JProbe 过滤器。例如：

```
<filters>
<include class="com.oreilly.*" method="*"/>
<exclude class="com.oreilly.test.*" method="*"/>
</filters>
```

它支持一个属性：

`defaultexclude (1.4, boolean, N)`

如果为 `true`，则排除所有类和方法。默认为 `true`。

<filters> 支持以下嵌套元素：

0 到 n 个嵌套 <include> 元素 (1.4)

定义要包含哪些类和方法。支持以下属性：

`class (1.4, String, N)`

一个正则表达式，指定所要包含或排除的类。

`enabled (1.4, boolean, N)`

如果为 true，则启用此元素。默认为 true。

`method (1.4, String, N)`

一个正则表达式，指定所要包含或排除的方法。

0 到 n 个嵌套 <exclude> 元素 (1.4)

定义所要排除的类和方法。支持以下属性：

`class (1.4, String, N)`

一个正则表达式，指定所要包含或排除的类。

`enabled (1.4, boolean, N)`

如果为 true，则启用此元素。默认为 true。

`method (1.4, String, N)`

一个正则表达式，指定所要包含或排除的方法。

0 或 1 个嵌套 <socket> 元素 (1.4)

定义对应远程查看的主机和端口号。属性如下：

`host (1.4, String, N)`

要连接的主机。默认为 localhost。

`port (1.4, int, N)`

端口号。默认为 4444。

0 或 1 个嵌套 <triggers> 元素 (1.4)

定义 JProbe 触发器，即为某些特定事件发生时所进行的活动。支持以下嵌套元素：

0 到 n 个嵌套 <method> 元素 (1.4)

每个元素定义一个新的触发器。支持以下属性：

`action (1.4, String, Y)`

要完成的操作。必须取值为 clear、exit、pause、resume、snapshot 或 suspend 之一。

event (1.4, String, Y)

触发操作的事件。必须取值为 enter 或 exit。

name (1.4, String, Y)

作为一个正则表达式的方法的名字。`com.oreilly.util.DateUtil.getCurrentTime` 即为一例。

param (1.4, String, N)

追加到 `-jp_trigger` 标志结尾的可选参数。

jpcovmerge

1.4

合并 JProbe Coverage 快照

org.apache.tools.ant.taskdefs.optional.sitraka.CovMerge

将多个 JProbe Coverage 快照合并为一个。

属性

home (1.4, File, Y)

JProbe 所安装的目录。

tofile (1.4, File, Y)

输出文件名。

verbose (1.4, boolean, N)

如果为 true，则以详细模式操作。默认为 false。

内容

1 到 n 嵌套 <fileset> 元素 (1.4)

定义要合并快照列表的 fileset 元素。

jpcovreport

1.4

生成可打印的 JProbe 报告

org.apache.tools.ant.taskdefs.optional.sitraka.CovReport

生成 JProbe Coverage 快照的一个可打印的报告。

属性

`filters (1.4, String, N)`

用逗号分隔的过滤器列表，各过滤器形如`<package>.class:?`，在此?可为 I（表示包含，即 Include），也可为 E（表示排除，即 Exclude）。

`format (1.4, Enum, N)`

报告的格式。合法值为 xml、html 或 text。默认为 html。

`home (1.4, File, Y)`

JProbe 所安装的目录。

`includesource (1.4, boolean, N)`

如果为 true，则在报告中包含源代码。仅当 `format="xml"` 且 `type="verydetailed"` 时才可用。默认为 true。

`percent (1.4, int, N)`

打印方法的阈值，范围为 0 到 100。默认为 100。

`snapshot (1.4, File, Y)`

要报告的快照的名字。

`tofile (1.4, File, Y)`

要生成的报告的名字。

`type (1.4, Enum, N)`

要生成的报告的类型。合法值为 executive、summary、detailed 和 verydetailed。默认为 detailed。

内容

`0 到 n 个嵌套 <sourcepath> 元素 (1.4)`

指定在哪里寻找源文件的 Path 元素。

`0 或 1 个嵌套 <reference> 元素 (1.4)`

仅当 `format="xml"` 时可用。指定额外的类以检查覆盖信息。`<reference>` 元素可以包含以下内容：

0 或 1 个嵌套 <classpath> 元素 (1.4)

定义在哪里寻找类的 Path 元素。

0 或 1 个嵌套 <filters> 元素 (1.4)

在 jpcov 任务下定义 <filters> 的语法。

junit

all

使用 JUnit 执行测试

org.apache.tools.ant.taskdefs.optional.junit.JUnitTask

使用 JUnit 测试框架执行单元测试。此任务要求 JUnit 3.0 或更高版本，可在 <http://www.junit.org> 得到。

属性

dir (all, File, N)

JVM 的工作目录。仅当 fork=true 时使用。

errorproperty (1.4, String, N)

出现测试错误时所设置的特性。

failureproperty (1.4, String, N)

出现测试失败时所设置的特性。

fork (all, boolean, N)

如果为 true，则为测试创建一个新 JVM。默认为 false。

haltonerror (all, boolean, N)

如果为 true，则当出现一个测试错误时，终止构建。默认为 false。

haltonfailure (all, boolean, N)

如果为 true，则当出现一个测试失败时，终止构建。默认为 false。

jvm (all, String, N)

用于调用 JVM 的命令。默认为 java。仅当 fork=true 时使用。

maxmemory (all, String, N)

fork=true 时所用的最大内存量。

`printsummary (all, Enum, N)`

对每个测试要打印的统计结果加以配置。合法值为 `on`、`off` 和 `withOutAndErr`。`withOutAndErr` 等同于 `on`，只不过测试输出要同时写到标准输出和标准错误输出。默认为 `off`。

`timeout (all, int, N)`

在超时之前，等待一个测试的最大毫秒数。仅当 `fork=true` 时使用。

内容

`0 到 n 个嵌套 <batchtest> 元素 (all)`

基于命名约定定义一个测试集合。支持以下属性：

`errorproperty (1.4, String, N)`

覆盖 `junit` 中指定的 `errorproperty` 属性。默认为 `false`。

`failureproperty (1.4, String, N)`

覆盖 `junit` 中指定的 `failureproperty` 属性。

`fork (all, boolean, N)`

覆盖 `junit` 中指定的 `fork` 属性。默认为 `false`。

`haltonerror (all, boolean, N)`

覆盖 `junit` 中指定的 `haltonerror` 属性。默认为 `false`。

`haltonfailure (all, boolean, N)`

覆盖 `junit` 中指定的 `haltonfailure` 属性。默认为 `false`。

`if (all, String, N)`

指定一个特性。仅当所指定特性设置时才运行测试。

`todir (1.3, 1.4, String, N)`

报告的目标目录。

`unless (all, String, N)`

指定一个特性。除非所指定特性被设置，否则运行测试。

0 或 1 个嵌套 <classpath> 元素 (all)

运行测试时所用的 path 元素。

0 到 n 个嵌套 <formatter> 元素 (all)

配置测试结果如何写出。支持以下属性：

`type (all, Enum, *)`

指定使用何种预定义格式化方式。合法值为 `xml`、`plain` 和 `brief`。

`classname (all, String, *)`

定制格式化类。

`extension (all, String, *)`

输出文件扩展名。与 <test> 的 `outfile` 属性联合使用。

`usefile (all, boolean, N)`

如果为 `true`, 此任务向一个文件发送输出。文件名由测试名确定, 或者由 <test> 元素的 `outfile` 属性指定。默认为 `true`。

`type` 或 `classname` 中必须指定一个。如果指定了 `classname`, 则 `extension` 是必要的。

0 到 n 个嵌套 <jvmarg> 元素 (all)

命令行参数, 如第四章所述。仅当 `fork=true` 时合法。

0 到 n 个嵌套 <sysproperty> 元素 (1.3, 1.4)

环境变量, 如第四章所述。

0 到 n 个嵌套 <test> 元素 (all)

每个元素定义一个。支持以下属性：

`errorproperty (1.4, String, N)`

覆盖 `junit` 中指定的 `errorproperty` 属性。默认为 `false`。

`failureproperty (1.4, String, N)`

覆盖 `junit` 中指定的 `failureproperty` 属性。

`fork (all, boolean, N)`

覆盖 `junit` 中指定的 `fork` 属性。默认为 `false`。

`haltonerror (all, boolean, N)`

覆盖 `junit` 中指定的 `haltonerror` 属性。默认为 `false`。

`haltonfailure (all, boolean, N)`

覆盖 `junit` 中指定的 `haltonfailure` 属性。默认为 `false`。

`if (all, String, N)`

指定一个特性。仅当所指定特性设置时才运行测试。

`name (all, String, Y)`

测试类的类名。

`outfile (all, String, N)`

测试结果的基本名。`<formatter>` 指定的扩展名追加到此名后面。默认为 `TEST-`，其后为 `name` 属性的值。

`todir (1.3, 1.4, File, N)`

报告的目标目录。

`unless (all, String, N)`

指定一个特性。除非所指定特性设置时才运行测试。

junitreport

1.3, 1.4

创建格式化的 JUnit 报告 `org.apache.tools.ant.taskdefs.optional.junit.XMLResultAggregator`

基于来自 `junit` 任务的多个 XML 文件，创建一个格式化的报告。此任务应用一个 XSLT 样式表。必须有 Apache 的 Xalan XSLT Processor (<http://xml.apache.org>) (注 3)。

属性

`todir (1.3, 1.4, File, N)`

XML 文件的目标目录。

注 3： 支持 Xalan 1.2.2，不过建议采用 Xalan 2.x。

tofile (1.3, 1.4, String, N)

目标 XML 文件名。来自 `junit` 任务的各个 XML 文件要聚集到此文件中。默认认为 `TESTS-TestSuites.xml`。

内容

0 到 n 个嵌套 <fileset> 元素 (1.3, 1.4)

`fileset` 元素，它要选择 XML 报告从而合并到 `tofile` 所指定的文件中。它们是由 `junit` 任务得到的输出文件。

0 到 n 个嵌套 <report> 元素 (1.3, 1.4)

每个元素指定如何完成一个 XSLT 转换从而生成一个格式化的报告。支持以下属性：

todir (1.3, 1.4, File, N)

转换结果的目标目录。默认为当前目录。

styledir (1.3, 1.4, File, N)

包含 `junit-frames.xsl` 和 `junit-noframes.xsl` 的目录。如果未指定，任务由 Ant 可选任务 JAR 文件中加载文件。

format (1.3, 1.4, Enum, N)

选择要使用哪些样式表。合法值为 `frames` 或 `noframes`。默认为 `frames`。

maudit

1.4

使用 WebGain Quality Analyzer 查找错误

org.apache.tools.ant.taskdefs.optional
.metamata.MAudit

执行 WebGain Quality Analyzer 来分析 Java 源代码以找出编程错误。此任务要基于 Metamata Audit，后者由 WebGain 于 2000 年 10 月收购。2002 年 1 月 3 日，WebGain Quality Analyzer 集成到 WebGain Studio，而且不再作为一个独立的产品推出。有关的更多信息，请参见 <http://www.webgain.com>。

属性

`fix (1.4, boolean, N)`

如果为 `true`, 则自动修正某些特定类型的错误。默认为 `false`。

`list (1.4, boolean, N)`

如果为 `true`, 则为每个已审计的文件创建一个 `.maudit` 列表文件。默认为 `false`。

`maxmemory (1.4, String, N)`

JVM 的最大内存。

`metamatahome (1.4, File, Y)`

包含 Metamata 发布的目录。

`tofile (1.4, File, Y)`

审计报告的目标 XML 文件。

`unused (1.4, boolean, N)`

如果为 `true`, 则在搜索路径上查找未用的声明。与 `<searchpath>` 嵌套元素一同使用。默认为 `false`。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.4)`

指定类路径的 `path` 元素。

`0 或 1 个嵌套 <fileset> 元素 (1.4)`

指定在哪里查找 `.java` 文件的 `fileset` 元素。

`0 到 n 个嵌套 <jvmarg> 元素 (1.4)`

命令行参数, 如第四章所述。仅当 `fork=true` 时合法。

`0 或 1 个嵌套 <searchpath> 元素 (1.4)`

指定在哪里查找未用的全局声明的 `path` 元素。`unused=true` 时这是必要的。

若指定一个 `<searchpath>` 而 `unused=false`, 此任务将发出一个警告。

0 或 1 个嵌套 <sourcepath> 元素 (1.4)

覆盖 SOURCEPATH 环境变量的 path 元素。

mimemail

1.4

发送带附件的 email

org.apache.tools.ant.taskdefs.optional.net.MimeMail

发送带 MIME 附件的 SMTP 邮件。此任务需要 JavaMail API 和 JavaBeans 活动框架 (JavaBeans Activation Framework)。与核心 mail 任务的区别在于它支持附件。

属性

bcclist (1.4, String, *)

用逗号分隔的 BCC 接收者列表。

cclist (1.4, String, *)

用逗号分隔的 CC 接收者列表。

failonerror (1.4, boolean, N)

如果为 true，则在失败时终止构建。默认为 true。

from (1.4, String, Y)

发送者的 email 地址。

mailhost (1.4, String, N)

邮件服务器名。默认为 localhost。

message (1.4, String, *)

消息体。

messagefile (1.4, File, *)

包含消息体的文件。

messagemimetype (1.4, String, N)

使用 message 或 messagefile 属性时所附带的消息的 MIME 类型。默认为 text/plain。

subject (1.4, String, N)

email 主题行。

tolist (1.4, String, *)

用逗号分隔的 TO 接收者列表。

指定 message 或 messageFile 之一，或者指定一个嵌套 <fileset>。 tolist、cclist 或 bcclist 中必须至少指定一个。

内容

0 到 n 个嵌套 <fileset> 元素 (1.4)

指定要附带的文件的 fileset 元素。

mmetrics

1.4

使用 WebGain Quality Analyzer 报告测量结果

org.apache.tools.ant.taskdefs.optional.metamodel.MMetrics

对一组 Java 文件执行 WebGain Quality Analyzer，报告代码复杂性和其他测试结果。有关 WebGain Quality Analyzer（以前称为 Metamata Quality Analyzer）的背景信息请参见 maudit 任务。

属性

granularity (1.4, String, Y)

指定测量粒度。合法值为 files、types 和 methods。

maxmemory (1.4, String, N)

JVM 可用的最大内存。

metamatahome (1.4, File, Y)

包含 WebGain Quality Analyzer 发布的目录。

tofile (1.4, File, Y)

测量报告的目标 XML 文件。

内容

0 或 1 个嵌套 <classpath> 元素 (1.4)

指定类路径的 path 元素。

0 到 n 个嵌套 <fileset> 元素 (1.4)

指定要分析的源文件的 fileset 元素。

0 到 n 个嵌套 <jvmarg> 元素 (1.4)

命令行 JVM 参数，如第四章所述。

0 或 1 个嵌套 <path> 元素 (1.4)

指定扫描源代码的目录的 path 元素。

0 或 1 个嵌套 <sourcepath> 元素 (1.4)

覆盖 SOURCEPATH 环境变量的 path 元素。

mparse

all

对一个语法文件执行 MParse org.apache.tools.ant.taskdefs.optional.metamata.MParse

对一个语法文件执行 Metamata MParse compiler compiler 编译器。Metamata 由 WebGain 于 2000 年 10 收购，而且无法再得到 MParse。

属性

`cleanup (1.3, 1.4, boolean, N)`

如果为 true，则删除语法文件转换时创建的临时 Sun JavaCC 文件。默认为 false。

`debugparser (1.3, 1.4, boolean, N)`

如果为 true，则启用解析程序调试。默认为 false。

`debugscanner (1.3, 1.4, boolean, N)`

如果为 true，则启用扫描程序调试。默认为 false。

`maxmemory (1.3, 1.4, String, N)`

设置 JVM 可用的最大内存。

`metamatahome (all, File, Y)`

包含 Metamata 发布的目录。

`target (all, File, Y)`

要处理的 `.jj` 语法文件。其时间戳要与所生成的 `.java` 文件时间戳相比较，而且仅当 `.jj` 更新时此任务才运行。

`verbose (1.3, 1.4, boolean, N)`

如果为 `true`，则以详细模式操作。默认为 `false`。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.3, 1.4)`

可用以代替 `classpath` 或 `classpathref` 属性。

`0 到 n 个嵌套 <jvmarg> 元素 (1.3, 1.4)`

命令行 JVM 参数，如第四章所述。

`0 或 1 个嵌套 <sourcepath> 元素 (1.3, 1.4)`

定义源文件位置的 `path` 元素。

native2ascii

`all`

将文件转换为 ASCII 文件

`org.apache.tools.ant.taskdefs.optional.Native2Ascii`

将本地编码的文件转换为包含转义 Unicode 字符的 ASCII 文件。

属性

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`dest (all, File, Y)`

输出的目标目录。

`encoding (all, String, N)`

源文件的字符编码。默认为平台默认编码。

`excludes (all, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`ext (all, String, N)`

重命名输出文件所用的文件扩展名。如果未指定，文件不会重命名。

`includes (all, String, N)`

要包含的文件模式列表（用逗号分隔）。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`reverse (all, boolean, N)`

如果为 `true`，则由 ASCII 转换为本地编码。默认为 `false`。

`src (all, File, N)`

包含要转换文件的目录。默认为工程的基目录。

内容

0 到 n 个嵌套 `patternset` 元素：`<exclude>`、`<include>`、`<patternset>` (`all`)；
`<excludesfile>`、`<includesfile>` (1.4)

用以替代其相应属性，它们指定了所包含和排除的部署描述文件。

0 或 1 个嵌套 `<mapper>` 元素 (1.3, 1.4)

定义文件如何重新命名。如果 `<mapper>` 和 `ext` 属性均已指定，则 `mapper` 优先。`mapper` 在第四章已做描述。

netrexxc

`all`

编译 NetRexx 文件

`org.apache.tools.ant.taskdefs.optional.NetRexxC`

编译一组 NetRexx 文件。NetRexx 是一种“面向人”的编码语言，其目标是要比 Java 更简单，但能够得到 Java 字节码，并与 Java 类无缝地交互。它可以在 <http://www2.hursley.ibm.com/netrexx/> 免费获得。

属性

`binary (all, boolean, N)`

如果为 `true`, 则将直接量看作二进制而不是 NetRexx 类型。默认为 `false`。

`classpath (all, String, N)`

编译所用的类路径。

`comments (all, boolean, N)`

生成 Java 源代码的 NetRexx 编译器。若此标志为 `true`, 则 NetRexx 注释也将被带至所生成的 Java 代码中。默认为 `false`。

`compact (all, boolean, N)`

如果为 `true`, 则显示简洁的（而不是详细的）错误消息。默认为 `false`。

`compile (all, boolean, N)`

如果为 `true`, 则编译所生成的 Java 代码。默认为 `true`。

`console (all, boolean, N)`

如果为 `true`, 则向控制台写消息。默认为 `false`。

`crossref (all, boolean, N)`

如果为 `true`, 则生成变量交叉引用。默认为 `false`。

`decimal (all, boolean, N)`

如果为 `true`, 则在 NetRexx 代码中使用十进制数制。默认为 `true`。

`defaultexcludes (all, boolean, N)`

确定是否使用默认排除模式。默认为 `true`。

`destdir (all, File, Y)`

编译 NetRexx 源文件前将其复制到的目录。

`diag (all, boolean, N)`

如果为 `true`, 生成编译的诊断信息。默认为 `false`。

`excludes(all, String, N)`

要排除的文件模式列表（用逗号分隔）。

`excludesfile (all, File, N)`

每行包括一个排除模式的文件名。

`explicit (all, boolean, N)`

如果为 `true`, 则变量在使用前必须显式声明。默认为 `false`。

`format (all, boolean, N)`

如果为 `true`, 则要对 Java 源代码进行格式化。否则, 要使生成的代码的行号与 NetRexx 源代码匹配从而有利于调试。默认为 `false`。

`includes (all, String, N)`

要包含的文件模式列表 (用逗号分隔)。

`includesfile (all, File, N)`

每行包括一个包含模式的文件的文件名。

`java (all, boolean, N)`

如果为 `true`, 则生成 Java 源代码。默认为 `false`。

`keep (all, boolean, N)`

如果为 `true`, 则保存所生成的 Java 源代码, 且文件扩展名为 `.java.keep`。默认为 `false`。

`logo (all, boolean, N)`

如果为 `true`, 则编译时显示编译器文本标识。默认为 `true`。

`replace (all, boolean, N)`

如果为 `true`, 则编译时替换所生成的 `.java` 文件。默认为 `false`。

`savelog (all, boolean, N)`

如果为 `true`, 则除了将编译器消息写至控制台外, 还要写到 `NetRexxC.log` 文件。默认为 `false`。

`sourcedir (all, boolean, N)`

如果为 `true`, 将类文件与源文件保存在同一个目录中。否则使用工作目录。默认为 `true`。

srcdir (all, File, Y)

NetRexx 源代码所在目录。

strictargs (all, boolean, N)

如果为 true，即使 NetRexx 方法调用没有任何参数，也必须使用括号。默认为 false。

strictassign (all, boolean, N)

如果为 true，则赋值时类型必须匹配。默认为 false。

strictcase (all, boolean, N)

如果为 true，则 NetRexx 代码是区分大小写的。默认为 false。

strictimport (all, boolean, N)

如果为 true，则类必须显式导入。默认为 false。

strictprops (all, boolean, N)

如果为 true，则本地特性必须用 this 显式引用。默认为 false。

strictsignal (all, boolean, N)

如果为 true，则异常必须由类型显式捕获。默认为 false。

symbols (all, boolean, N)

如果为 true，则在所生成的类文件中包括调试符号。默认为 false。

time (all, boolean, N)

如果为 true，则向控制台打印编译时间。默认为 false。

trace (all, String, N)

若指定，则启用某个 NetRexx 跟踪选项。合法值为 trace、trace1、trace2 和 notrace。默认为 trace2。

utf8 (all, boolean, N)

如果为 true，则假设源文件使用 UTF-8 编码。默认为 false。

verbose (all, String, Y)

若指定，以详细模式操作。默认为 verbose3。合法值为从 verbose1 到 verbose5。verbose5 输出最详细的信息。

内容

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (all);
<excludesfile>、<includesfile> (1.4)

用以替代其相应属性, 它们指定了选择要编译的文件时所包含和排除的源文件组。

p4change 1.3, 1.4

请求 Perforce 修改表 org.apache.tools.ant.taskdefs.optional.perforce.P4Change

向一个 Perforce 服务器请求一个新的修改表。

属性

client (1.3, 1.4, String, N)

指定 p4 -c 选项。默认为 p4.client 特性的值 (如果已设置)。

description (1.3, 1.4, String, N)

修改表的注释。默认为 AutoSubmit By Ant。

port (1.3, 1.4, String, N)

指定 p4 -p 选项。默认为 p4.port 特性的值 (如果已设置)。

user (1.3, 1.4, String, N)

指定 p4 -u 选项。默认为 p4.user 特性的值 (如果已设置)。

view (1.3, 1.4, String, N)

此命令所操作的客户、分支或标签视图。

内容

无。

p4counter

1.4

取得 / 设置 Perforce 计数器

org.apache.tools.ant.taskdefs.optional.perforce.P4Counter

取得和设置一个 Perforce 计数器的值。

属性**client** (1.3, 1.4, String, N)

指定 *p4 -c* 选项。默认为 *p4.client* 特性的值（如果已设置）。

name (1.4, String, Y)

计数器名。如果此为惟一指定的属性，则任务将计数器的值打印到标准输出。

port (1.3, 1.4, String, N)

指定 *p4 -p* 选项。默认为 *p4.port* 特性的值（如果已设置）。

property (1.4, String, *)

利用所得到的计数器值设置的特性。

user (1.3, 1.4, String, N)

指定 *p4 -u* 选项。默认为 *p4.user* 特性的值（如果已设置）。

value (1.4, int, *)

若指定，则将计数器设置为此值。

view (1.3, 1.4, String, N)

此命令所操作的客户、分支或标签视图。

不能同时设置 **property** 和 **value**，因为指定 **property** 特性时是在获取一个计数器值，而指定 **value** 时则是在设置一个计数器值。

内容

无。

p4edit 1.3, 1.4

打开 Perforce 文件 org.apache.tools.ant.taskdefs.optional.perforce.P4Edit

由 Perforce 打开文件以进行编辑。

属性

`change (1.3, 1.4, String, N)`

为文件指派此现有的修改表号。

`client (1.3, 1.4, String, N)`

指定 `p4 -c` 选项。默认为 `p4.client` 特性的值（如果已设置）。

`port (1.3, 1.4, String, N)`

指定 `p4 -p` 选项。默认为 `p4.port` 特性的值（如果已设置）。

`user (1.3, 1.4, String, N)`

指定 `p4 -u` 选项。默认为 `p4.user` 特性的值（如果已设置）。

`view (1.3, 1.4, String, Y)`

此命令所操作的客户、分支或标签视图。

内容

无。

p4have 1.3, 1.4

列出当前 Perforce 文件 org.apache.tools.ant.taskdefs.optional.perforce.P4Have

列出在当前客户视图中的 Perforce 文件。

属性

`client (1.3, 1.4, String, N)`

指定 `p4 -c` 选项。默认为 `p4.client` 特性的值（如果已设置）。

port (1.3, 1.4, String, N)

指定 *p4 -p* 选项。默认为 *p4.port* 特性的值（如果已设置）。

user (1.3, 1.4, String, N)

指定 *p4 -u* 选项。默认为 *p4.user* 特性的值（如果已设置）。

view (1.3, 1.4, String, N)

此命令所操作的客户、分支或标签视图。

内容

无。

p4label

1.3, 1.4

标记 Perforce 文件

org.apache.tools.ant.taskdefs.optional.perforce.P4Label

为当前 Perforce 工作区中的文件创建一个标签。

属性

client (1.3, 1.4, String, N)

指定 *p4 -c* 选项。默认为 *p4.client* 特性的值（如果已设置）。

desc (1.3, 1.4, String, N)

标签注释。

lock (1.4, String, N)

如果设置为 *locked*, 则导致标签加锁。默认为一个空字符串。不允许其他值。

name (1.3, 1.4, String, Y)

标签名。

port (1.3, 1.4, String, N)

指定 *p4 -p* 选项。默认为 *p4.port* 特性的值（如果已设置）。

user (1.3, 1.4, String, N)

指定 *p4 -u* 选项。默认为 *p4.user* 特性的值（如果已设置）。

`view (1.3, 1.4, String, N)`

此命令所操作的客户、分支或标签视图。

内容

无。

p4reopen

1.4

在 Perforce 修改表间移动文件 `org.apache.tools.ant.taskdefs.optional.perforce.P4Reopen`

在 Perforce 修改表间移动文件。

属性

`client (1.3, 1.4, String, N)`

指定 `p4 -c` 选项。默认为 `p4.client` 特性的值（如果已设置）。

`port (1.3, 1.4, String, N)`

指定 `p4 -p` 选项。默认为 `p4.port` 特性的值（如果已设置）。

`tochange (1.4, String, Y)`

将文件移动到所指定的修改表。

`user (1.3, 1.4, String, N)`

指定 `p4 -u` 选项。默认为 `p4.user` 特性的值（如果已设置）。

`view (1.3, 1.4, String, N)`

此命令所操作的客户、分支或标签视图。

内容

无。

p4revert

1.4

回复 Perforce 文件

org.apache.tools.ant.taskdefs.optional.perforce.P4Revert

回复打开的 Perforce 文件。

属性**change** (1.4, String, N)

要回复的修改表。

client (1.3, 1.4, String, N)指定 *p4 -c* 选项。默认为 *p4.client* 特性的值（如果已设置）。**port** (1.3, 1.4, String, N)指定 *p4 -p* 选项。默认为 *p4.port* 特性的值（如果已设置）。**revertonlyunchanged** (1.4, boolean, N)如果为 *true*，则仅回复未改变的文件。默认为 *false*。**user** (1.3, 1.4, String, N)指定 *p4 -u* 选项。默认为 *p4.user* 特性的值（如果已设置）。**view** (1.3, 1.4, String, N)

此命令所操作的客户、分支或标签视图。

内容

无。

p4submit

1.3, 1.4

写入 Perforce 文件

org.apache.tools.ant.taskdefs.optional.perforce.P4Submit

将文件写入一个 Perforce 库。

属性

`change (1.3, 1.4, String, Y)`

提交指定的修改表。

`client (1.3, 1.4, String, N)`

指定 `p4 -c` 选项。默认为 `p4.client` 特性的值（如果已设置）。

`port (1.3, 1.4, String, N)`

指定 `p4 -p` 选项。默认为 `p4.port` 特性的值（如果已设置）。

`user (1.3, 1.4, String, N)`

指定 `p4 -u` 选项。默认为 `p4.user` 特性的值（如果已设置）。

`view (1.3, 1.4, String, N)`

此命令所操作的客户、分支或标签视图。

内容

无。

p4sync

1.3, 1.4

与一个 Perforce 库同步 `org.apache.tools.ant.taskdefs.optional.perforce.P4Sync`

将工作区与一个 Perforce 库同步。

属性

`client (1.3, 1.4, String, N)`

指定 `p4 -c` 选项。默认为 `p4.client` 特性的值（如果已设置）。

`force (1.3, 1.4, String, N)`

如果设置为一个非空字符串，则设置 `-f` Perforce 标志。要求刷新文件（注 4）。

注 4： 在这种情况下，Boolean 属性更有意义。Ant 可选任务不如核心任务那样一致。

label (1.3, 1.4, String, N)

如果设置，此任务将使用所指定的标签令工作区与一个Perforce库中的文件取得同步。

port (1.3, 1.4, String, N)

指定 *p4 -p* 选项。默认为 *p4.port* 特性的值（如果已设置）。

user (1.3, 1.4, String, N)

指定 *p4 -u* 选项。默认为 *p4.user* 特性的值（如果已设置）。

view (1.3, 1.4, String, N)

此命令所操作的客户、分支或标签视图。

内容

无。

propertyfile

1.3, 1.4

创建 / 编辑特性文件

org.apache.tools.ant.taskdefs.optional.PropertyFile

创建或编辑 Java 特性文件。此任务增加或编辑文件中的项。它不留现有特性文件中的注释。

属性

comment (1.3, 1.4, String, N)

增加到特性文件中的注释。

file (1.3, 1.4, File, Y)

要创建或修改的特性文件的文件名。

内容

0 到 n 个嵌套 <entry> 元素 (1.3, 1.4)

每一项定义一个要写入特性文件或要修改的一个名 - 值对。支持以下属性：

`default (1.3, 1.4, String, N)`

如果某个特性尚未定义，此为其初始值。对于一个`date`类型，此值可以设置为`now`或`never`来分别表示当前时间或一个空时间。可能有些奇怪的是，如果某个特性不存在，而`default`和`value`都已指定，则将为此特性赋以`value`而不是`default`。

`key (1.3, 1.4, String, Y)`

特性名。

`operation (1.3, 1.4, Enum, N)`

控制`value`属性如何修改`key`属性所指示的特性值。对于所有数据类型均可用`+`或`=`，而且对于`date`和`int`类型还可用`-`。`+`完成加法，`=`完成赋值，`-`完成减法。

`pattern (1.3, 1.4, String, N)`

对`date`和`int`类型如何格式化进行控制。分别使用`SimpleDateFormat`和`DecimalFormat`。

`type (1.3, 1.4, Enum, N)`

合法值为`int`、`date`和`string`。默认为`string`。

`value (1.3, 1.4, String, Y)`

对于由`key`指定的特性，指定一个要增加、减去或赋予的值。与`operation`属性一同使用。对于`date`类型，还可以设置为`now`或`never`。

使用示例

此示例中，如果一个特性文件不存在则创建，并更新多个值：

```
<target name="test_propertyfile">
    <propertyfile comment="Edited by the propertyfile task"
        file="stats.properties">
        <entry key="numRuns" type="int" default="1" operation="+" value="1"/>
        <entry key="lastRun" type="date" operation="=" value="now"
            pattern="MMM dd, yyyy"/>
        <entry key="runBy" operation="="
            value="${user.name}"/>
    </propertyfile>
</target>
```

以下为运行此构建 4 次后，特性文件的形式：

```
#Edited by the propertyfile task  
#Thu Jan 17 10:42:40 CST 2002  
runBy=ericb  
lastRun=Jan 17, 2002  
numRuns=4
```

pvcs

1.4

写出 PVCS 文件 org.apache.tools.ant.taskdefs.optional.pvcs.Pvcs

由一个 PVCS 存储库抽取文件。此任务要求 Merant 的 PVCS Version Manager 系统，可在 <http://www.merant.com> 得到。

属性

force $(1.4, \text{String}, N)$

如果为 yes，则现有文件将被重写。默认为 no。

`ignorereturncode` (*1.4*, *boolean*, *N*)

如果为 true，则当命令失败时构建不终止。默认为 false。

label (1.4, String, N)

指定一个标签。若指定，则只抽取带有指定标签的文件。

`promotiongroup (1.4, String, N)`

指定一个提升组。若指定，则仅抽取属于提升组的文件。

pvcsbin (*1.4, String, N*)

指定 PVCS 发布的 *bin* 目录的位置。

pvcsp project (1.4, String, N)

由此抽取文件的工程。默认为“/”。

repository (1.4. String, Y)

PVCS 存储库的位置。

`updateonly (1.4, boolean, N)`

如果为 `true`, 则只得到比现有本地文件新的文件。默认为 `false`。

`workspace (1.4, String, N)`

文件所要抽取到的工作区。工作区通过 PVCS 客户应用配置。

内容

0 到 n 个嵌套 <pvcsp project> 元素 (1.4)

每个元素都有一个必要的 `name` 属性, 它指定了一个 PVCS 工程, 由此来抽取文件。可以使用多个元素来指定多个工程。这是对由 `pvcsp project` 属性已经指定的所有工程的补充。

renameext

1.2 (1.3 中已经废弃)

重命名文件扩展名

重命名文件扩展名。例如, 可用于将 `*.java` 重命名为 `*.java.bak`。此任务在 Ant 1.3 中已经弃用。而代之以使用有一个 `glob mapper` 的 `move` 任务。

rpm

1.4

创建 Linux RPM 文件

`org.apache.tools.ant.taskdefs.optional.Rpm`

构建一个 Linux RPM 文件。仅在 Linux 平台上能够工作。

属性

`cleanbuilddir (1.4, boolean, N)`

如果为 `true`, 则删除 `BUILD` 目录中所生成的文件。默认为 `false`。

`command (1.4, String, N)`

传递给 `rpm` 可执行程序的一个参数。默认为 `-bb`。

`error (1.4, File, N)`

对应标准错误输出的目标文件。

output (1.4, File, N)

对应标准输出的目标文件。

removesource (1.4, boolean, N)

如果为 true，则删除 SOURCES 目录中的源文件。默认为 false。

removespec (1.4, boolean, N)

如果为 true，则删除 SPECS 目录中的 spec 文件。默认为 false。

specfile (1.4, String, Y)

所用的 spec 文件的文件名。

topdir (1.4, File, N)

目标目录。其中包含 SPECS、SOURCES、BUILD 和 SRPMS 子目录。

内容

无。

script

all

执行一个 BSF 脚本

org.apache.tools.ant.taskdefs.optional.Script

执行一个 BSF (Bean Scripting Framework, Bean 脚本框架) 脚本。此任务需要 IBM 的 Bean Scripting Framework，还需要诸如 Rhino 或 Jython 等所支持的脚本语言。BSF 可在 <http://oss.software.ibm.com/developerworks/projects/bsf> 获得。BSF 发布中可以找到 *ReleaseNotes.html*，它指示了在哪里可以得到所支持的脚本语言。

属性

language (all, String, Y)

脚本语言名。

src (all, String, N)

如果脚本不是内联的，则此为脚本源文件的位置。

内容

`script`任务接受包含内联脚本代码的文本内容。可以替代`src`属性。若脚本中包含非法的 XML 字符（如“<”或“&”），或者必须保留换行符，则必须要有 XML CDATA 段，例如：

```
<script language="javascript"><![CDATA[  
    // some JavaScript code here...  
    if (a < b) {  
        ...  
    }  
]]></script>
```

sound

1.3, 1.4

在构建结束时播放一个声音 org.apache.tools.ant.taskdefs.optional.sound.SoundTask

在构建过程结束时播放一个声音文件。`<sound>`元素必须出现在构建过程中执行的某个 Ant 目标中，不过这个声音文件直到构建结束时才播放。此任务依赖于 Sun 公司的 JMF (Java Media Framework, Java 媒体框架)，它包括在 JDK 1.3 和以后版本中。对于较早的 Java 版本，可以在 <http://java.sun.com/products/java-media/sound/> 下载得到 JMF。

属性

无。

内容

0 或 1 个嵌套`<fail>`元素 (1.3, 1.4)

定义构建失败时要播放的声音。

0 或 1 个嵌套`<success>`元素 (1.3, 1.4)

定义构建成功时要播放的声音。

这两个嵌套元素都由同一个类实现，支持以下属性：

source (1.3, 1.4, File, Y)

要播放的声音文件的文件名。如果是一个目录，则随机选出一个文件。出于这个原因，此目录中的所有文件都必须是声音文件。此任务已经通过 WAV 和 AIFF 文件的测试。

duration (1.3, 1.4, long, N)

播放声音的最大毫秒数。

loops (1.3, 1.4, int, N)

重复播放声音的次数。默认为 0。如果为 1，则声音重复一次，因此会播放两次。即使请求多次播放，此任务所花费的全部时间也不过超过 duration 所指定的值。

使用示例

构建结束时插入欢呼声，或者在出现错误时播放一个气球爆炸的声音：

```
<target name="compile" depends="prepare">
  <sound>
    <!-- limit the applause to 2 seconds -->
    <success duration="2000" source="APPLAUSE.WAV"/>
    <fail source="EXPLODE.WAV"/>
  </sound>

  <javac ...> ...
</target>
```

starteam

1.3, 1.4

写出 StarTeam 文件

org.apache.tools.ant.taskdefs.optional.scm.AntStarTeamCheckOut

由 StarTeam 写出文件，这是一个商业产品，可于 <http://www.starbase.com> 得到。只有 StarTeam 的授权用户才可使用此任务，而且 starteam-sdk.jar 必须在类路径上。

属性

`excludes (all, String, N)`

写出时所要排除的文件列表（用空格分隔），优先于 `includes`。

`foldername (all, String, N)`

工程中的子文件夹，由此写出文件。

`force (all, boolean, N)`

如果为 `true`，则重写现有文件夹。默认为 `false`。

`includes (all, String, N)`

写出时所要包含的文件列表（用空格分隔）。

`password (all, String, Y)`

登录时所用的口令。

`projectname (all, String, Y)`

StarTeam 工程名。

`recursion (all, boolean, N)`

如果为 `true`，则写出时包括子文件夹。默认为 `true`。

`servername (all, String, Y)`

StarTeam 服务器名。

`serverport (all, String, Y)`

服务器端口号。

`targetfolder (all, String, Y)`

文件将写出到此目录。

`username (all, String, Y)`

登录时所用的用户名。

`verbose (all, boolean, N)`

如果为 `true`，则以详细模式操作。默认为 `false`。

`viewname (all, String, Y)`

StarTeam 视图名。

内容

无。

stylebook

1.3, 1.4

执行 Apache Stylebook 生成程序 org.apache.tools.ant.taskdefs.optional.StyleBook

执行 Apache Stylebook 文档生成程序。它依赖于 *stylebook.jar*, 后者可由 <http://xml.apache.org> 得到。Apache Xalan 发布还包括一个合适的 JAR 文件。

属性

`book (1.3, 1.4, File, Y)`

由此生成文档的 book XML 文件。

`classpath (1.3, 1.4, Path, N)`

所用的类路径。

`classpathref (1.3, 1.4, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`skindirectory (1.3, 1.4, File, Y)`

包含 Stylebook skin 的目录。

`targetdirectory (1.3, 1.4, File, Y)`

文档的目标目录。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.3, 1.4)`

可用以代替 classpath 或 classpathref 属性。

telnet

1.3, 1.4

创建 telnet 会话

org.apache.tools.ant.taskdefs.optional.net.TelnetTask

执行一个 telnet 会话。

属性

`initialcr (1.3, 1.4, boolean, N)`

如果为 `true`, 则连接后发送一个回车符。默认为 `false`。

`password (1.3, 1.4, String, N)`

登录口令。

`port (1.3, 1.4, int, N)`

服务器端口号。默认为 23。

`server (1.3, 1.4, String, Y)`

要连接的主机名。

`timeout (1.3, 1.4, int, N)`

超时之前等待的秒数。默认为不超时。

`userid (1.3, 1.4, String, N)`

登录名。

内容

`0 到 n 个嵌套 <read> 元素 (1.3, 1.4)`

每个元素包括一个所等待的文本串 (希望由服务器发送)。

`0 到 n 个嵌套 <write> 元素 (1.3, 1.4)`

每个元素包括一个发送到服务器的文本串。

test

1.3, 1.4

运行一个单元测试

`org.apache.tools.ant.taskdefs.optional.Test`

在 `org.apache.testlet` 框架中执行一个单元测试。

属性

`classpath (1.3, 1.4, Path, N)`

指定所用的类路径。

`classpathref (1.3, 1.4, Reference, N)`

在构建文件中某处定义的一个类路径的引用。

`forceshowtrace (1.3, 1.4, boolean, N)`

如果为 `true`, 则显示任何失败的栈跟踪轨迹。默认为 `false`。

`showbanner (1.3, 1.4, String, N)`

如果指定, 则在启动 testlet 引擎时显示一个标识。

`showsucces (1.3, 1.4, boolean, N)`

如果为 `true`, 则在测试成功时显示一条消息。默认为 `false`。

`showtrace (1.3, 1.4, boolean, N)`

如果为 `true`, 则显示错误 (而不是正常的测试失败) 的一个栈跟踪轨迹。默认为 `false`。

内容

0 或 1 个嵌套 <classpath> 元素 (1.3, 1.4)

指定类路径的 path 元素。

0 到 n 个嵌套 <testlet> 元素 (1.3, 1.4)

每个元素包含要测试的一个类名。例如:

```
<testlet>com.oreilly.util.test.CustomerTestlet</testlet>
```

vsscheckin

1.4

写入 Visual SourceSafe 文件 org.apache.tools.ant.taskdefs.optional.vss.MSVSSCHECKIN

将文件写入 Visual SourceSafe。

属性

`autoresponse (1.4, String, N)`

指定 -I 标志的值。合法值为 Y 和 N。省略时, 任务将 -I 传递给 VSS。否则将传递 -I-Y 或 -I-N。

`comment (1.4, String, N)`

应用于文件的注释。

`localpath (1.4, Path, N)`

覆盖本地工作目录。

`login (1.4, String, N)`

用户名 / 口令组合，形如“用户名，口令”，在此口令是可选的。

`recursive (1.4, boolean, N)`

如果为 `true`，则对子工程递归地操作。默认为 `false`。

`serverpath (1.4, String, N)`

`srcsafe.ini` 所在目录。

`ssdir (1.4, String, N)`

包含 `ss.exe` 的目录。如果省略，则 Ant 搜索 PATH (译注 1)。

`vsspath (1.4, String, Y)`

SourceSafe 工程的路径，没有前导 \$ 字符。

`writable (1.4, boolean, N)`

如果为 `true`，则文件写入后仍可写。默认为 `false`。

内容

无。

vsscheckout

1.4

写出 Visual SourceSafe 文件 org.apache.tools.ant.taskdefs.optional.vss.MSVSSCHECKOUT

由 Visual SourceSafe 写出文件。

译注 1：原文此处为 path，与后面不统一，因此统一为 PATH。

属性

`autoresponse (1.4, String, N)`

指定 -I 标志的值。合法值为 Y 和 N。省略时，任务将 -I 传递给 VSS。否则，传递 -I-Y 或 -I-N。

`date (1.4, String, *)`

写出文件时所用的日期戳。

`label (1.4, String, *)`

写出文件时所用的标签。

`localpath (1.4, Path, N)`

覆盖本地工作目录。

`login (1.4, String, N)`

用户名 / 口令组合，形如“用户名，口令”，在此口令是可选的。

`recursive (1.4, boolean, N)`

如果为 `true`，则对子工程递归地操作。默认为 `false`。

`serverpath (1.4, String, N)`

`srcsafe.ini` 所在目录。

`ssmdir (1.4, String, N)`

包含 `ss.exe` 的目录。如果省略，则 Ant 搜索 PATH。

`version (1.4, String, *)`

写出文件时所用的版本号。

`vsspath (1.4, String, Y)`

SourceSafe 工程的路径，没有前导 \$ 字符。

`version`、`date` 或 `label` 可以指定其一；均为可选的。

内容

无。

vssget all

取得 Visual SourceSafe 文件。 org.apache.tools.ant.taskdefs.optional.vss.MSVSSGET

由 Visual SourceSafe 取得文件。

属性

autoresponse (1.3, 1.4, String, N)

指定 -I 标志的值。合法值为 Y 和 N。省略时，任务将 -I 传递给 VSS。否则，则传递 -I-Y 或 -I-N。

date (all, String, *)

获得文件时所用的日期戳。

label (all, String, *)

获得文件时所用的标签。

localpath (all, Path, N)

覆盖本地工作目录。

login (all, String, N)

用户名/口令组合，形如“用户名，口令”，在此口令是可选的。

quiet (1.4, boolean, N)

如果为 true，则以安静模式操作。默认为 false。

recursive (all, boolean, N)

如果为 true，则对子工程递归地操作。默认为 false。

serverpath (1.4, String, N)

srcsafe.ini 所在目录。

ssdir (all, String, N)

包含 ss.exe 的目录。如果省略，则 Ant 搜索 PATH。

version (all, String, *)

获得文件时所用的版本号。

vsspath (*all, String, Y*)

SourceSafe 工程的路径，没有前导 \$ 字符。

writable (*all, boolean, N*)

如果为 true，文件得到后仍可写。默认为 false。

version、**date** 或 **label** 可以指定其一；均为可选的。

内容

无。

vsshistory

1.4

显示 Visual SourceSafe 工程历史 org.apache.tools.ant.taskdefs.optional.vss.MSVSSHISTORY

显示 Visual SourceSafe 中文件和工程的历史。

属性

dateformat (1.4, *String, N*)

fromdate 和 **todate** 属性所用的格式。此任务使用 `java.text.SimpleDateFormat`，默认为 `DateFormat.SHORT`。

fromdate (1.4, *String, **)

比较的开始日期。

fromlabel (1.4, *String, N*)

比较的开始标签。

login (1.4, *String, N*)

用户名 / 口令组合，形如“用户名，口令”，在此口令是可选的。

numdays (1.4, *int, **)

相对于 **fromdate** 或 **todate** 的天数。可以是一个负数。

output (1.4, File, N)

diff 所写至的目标文件。

recursive (1.4, boolean, N)

如果为 true，则对子工程递归地操作。默认为 false。

serverpath (1.4, String, N)

srcsafe.ini 所在目录。

ssdir (1.4, String, N)

包含 ss.exe 的目录。如果省略，则 Ant 搜索 PATH。

style (1.4, Enum, N)

历史报告的格式。合法值为 brief、codediff、default 或 nofile。默认为 default。

todate (1.4, String, *)

比较的结束日期。

tolabel (1.4, String, N)

比较的结束标签。

user (1.4, String, N)

如果指定，此任务将把一个用户命令（-U 选项）传递给 SourceSafe。

vsspath (1.4, String, Y)

SourceSafe 工程的路径，没有前导 \$ 字符。

fromdate、todate 和 numdays 的任意组合可以为此任务指定时间范围。

内容

无。

vsslabel

1.3, 1.4

标记 Visual SourceSafe 文件 org.apache.tools.ant.taskdefs.optional.vss.MSVSSLABEL

为 Visual SourceSafe 中的文件和工程赋予一个标签。

属性**autoresponse** (1.4, String, N)

指定 -I 标志的值。合法值为 Y 和 N。省略时，任务将 -I 传递给 VSS。否则，传递 -I-Y 或 -I-N。

comment (1.4, String, N)

与此标签关联的注释。

label (1.3, 1.4, String, Y)

应用于文件的标签。

login (1.3, 1.4, String, N)

用户名 / 口令组合，形如“用户名，口令”，在此口令是可选的。

serverpath (1.4, String, N)*srcsafe.ini* 所在目录。**ssdir** (1.3, 1.4, String, N)包含 *ss.exe* 的目录。如果省略，则 Ant 搜索 PATH。**version** (1.3, 1.4, String, N)

要标记的版本。默认为当前版本。

vsspath (1.3, 1.4, String, Y)

SourceSafe 工程的路径，没有前导 \$ 字符。

内容

无。

wljspc all

使用 WebLogic 编译器预编译 JSP org.apache.tools.ant.taskdefs.optional.jsp.WLJspc

使用 BEA WebLogic Server 的 JSP 编译器预编译 JSP 文件。此任务需要 WebLogic Version 4.5.1，而且仅建立文档以用于 Windows NT 4.0、Solaris 5.7 和 Solaris 5.8。

属性

classpath (*all, Path, N*)

编译 JSP 时所用的类路径。

defaultexcludes (*all, boolean, N*)

确定是否使用默认排除模式。默认为 true。

dest (*all, File, Y*)

已编译 JSP 的目标目录。

excludes (*all, String, N*)

要排除的文件模式列表（用逗号分隔）。

excludesfile (*all, File, N*)

每行包括一个排除模式的文件的文件名。

includes (*all, String, N*)

要包含的文件模式列表（用逗号分隔）。

includesfile (*all, File, N*)

每行包括一个包含模式的文件的文件名。

package (*all, String, Y*)

已编译 JSP 的目标包。

src (*all, File, Y*)

包含要编译的 JSP 的文档根目录。

内容

0 到 n 个嵌套 patternset 元素: <exclude>、<include>、<patternset> (*all*);
<excludesfile>、<includesfile> (1.4)

用以替代其相应属性，它们指定了包含和排除的源文件组。

0 或 1 个嵌套 <classpath> 元素 (*all*)

可用以代替 classpath 属性。

wlrun all

启动 WebLogic Server org.apache.tools.ant.taskdefs.optional.ejb.WLRun

启动 BEA WebLogic Server 的一个实例。此任务在服务器停止之前不会返回。

属性

args (*all, String, N*)

WebLogic 实例的附加参数。

beahome (1.3, 1.4, *File, Y*)

包含服务器配置文件的目录。仅应用于 WebLogic 6.0。若指定此属性，此任务即假定有 WebLogic 6.0。

classpath (*all, Path, Y*)

运行服务器所用的类路径。在 WebLogic 6.0 下，这应当包括所有 WebLogic JAR。

domain (1.3, 1.4, *String, Y*)

服务器的域。仅应用于 WebLogic 6.0。

home (*all, File, Y*)

WebLogic 发布目录。

jvmargs (*all, String, N*)

JVM 的附加参数。

name (all, String, N)

WebLogic 主目录中服务器的名字。默认为 myserver。

password (1.3, 1.4, String, Y)

服务器的管理口令。仅应用于 WebLogic 6.0。

pkpassword (1.3, 1.4, String, N)

私钥口令。仅应用于 WebLogic 6.0。

policy (all, String, N)

WebLogic 主目录中安全策略文件的名字。默认为 weblogic.policy。

property (all, String, Y,)

WebLogic 主目录中特性文件的名字。仅应用于 WebLogic 4.5.1 和 5.1。

username (1.3, 1.4, String, N)

服务器的管理用户名。仅应用于 WebLogic 6.0。

weblogicmainclass (all, String, N)

WebLogic 主类名。默认为 weblogic.Server。

wlclasspath (all, Path, N)

WebLogic 服务器所用的类路径。仅应用于 WebLogic 4.5.1 和 5.1。

内容

0 或 1 个嵌套 <classpath> 元素 (1.3, 1.4)

可用以代替 classpath 属性。

0 或 1 个嵌套 <wlclasspath> 元素 (1.3, 1.4)

可用以代替 wlclasspath 属性。

wlstop

all

停止 WebLogic Server

org.apache.tools.ant.taskdefs.optional.ejb.WLStop

停止 BEA WebLogic Server 的一个实例。

属性

`beahome (1.3, 1.4, File, N)`

包含服务器配置文件的目录。若指定此属性，此任务即假定有 WebLogic 6.0。

`classpath (all, Path, Y)`

执行 WebLogic shutdown 命令时所用的类路径。

`delay (all, String, N)`

关闭服务器之前等待的秒数（注 5）。默认为 0。

`password (all, String, Y)`

与指定用户关联的口令。

`url (all, String, Y)`

对某端口的 URL，服务器在相应端口监听 T3 连接。例如，`t3://myserver:7001`。

`user (all, String, Y)`

用于关闭服务所用账户的用户名。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.3, 1.4)`

可用以代替 `classpath` 属性。

`xmlvalidate`

1.4

验证 XML 文件

`org.apache.tools.ant.taskdefs.optional.XMLValidateTask`

验证 XML 文档为良构的，而且使用任何 SAX 解析程序确定其是否合法，后者是可选的。

注 5： 此任务利用 `Integer.parseInt()` 将此 `String` 属性值转换为 `int` 型。

属性

`classname (1.4, String, N)`

要使用的 SAX 解析程序的 Java 类的类名。

`classpath (1.4, Path, N)`

所用的类路径。

`classpathref (1.4, Reference, N)`

构建文件中某处定义的一个类路径的引用。

`failonerror (1.4, boolean, N)`

如果为 true，则当失败时终止构建。默认为 true。

`file (1.4, File, N)`

要验证的 XML 文件。要指定多个文件则要使用嵌套 `<fileset>` 元素。

`lenient (1.4, boolean, N)`

如果为 true，则验证 XML 是否为良构的，但并不验证其是否合法。仅在使用 SAX2 解析程序时才工作。默认为 false。

`warn (1.4, boolean, N)`

如果为 true，则将警告写到日志。默认为 true。

内容

`0 或 1 个嵌套 <classpath> 元素 (1.4)`

一个 path 元素，用以代替 classpath 或 classpathref 属性。

`0 到 n 个嵌套 <fileset> 元素 (1.4)`

一个或多个 fileset 元素，指定要验证的 XML 文件，用以代替 file 属性。

附录一

Ant 的未来

大多数开源工程都有着令人诧异的发展速度，Ant 也不例外。仅仅在过去的 2 年时间内，Ant 就从一个为构建 Tomcat 而设计的原型构建工具一跃成为许多 Java 工程首选的构建工具。好的一方面是，快速的改变意味着：对于开发人员在构建和发布其工程时所面对的问题有了更多的功能和解决方案。但也有不好的一面，即快速的改变也意味着不稳定性，开发人员必须确保新的功能不至于破坏其正在使用的当前功能，为此必须小心翼翼。

Ant 工程的维护人员所关心是其所做努力将对数以千计其他人的工程和工作产生何种影响。在 2001 年初，他们开始着手制定一个重构 Ant 设计的计划。随着时间的推移，Ant 的库已经变得相当庞大。有些任务的功能与其他任务的功能有所重叠。Ant 引擎的开发人员与 Ant 任务和监听者的开发人员之间并没有任何合约。Ant 引擎中的某些实现编写得很不好，因此需要重构；此重构会影响到许多对象中的设计。所有这些修改可能要历时数月，甚至数年。如果一个正在进行中的工程正处于一种稳定的开发状态，令其耽搁这么长的时间是让人难以接受的。正因为如此，维护人员决定另起炉灶，另外创建一个新的工具，即 Ant2。

Ant2

Ant的维护人员从用户和开发人员那里得到了有关重新设计和重构的建议。由这些建议产生了一套新的设计需求和功能要求。对于任务开发人员和Ant引擎开发人员如何协作，将有一个合约加以定义。不再需要对Ant的一些任务做内部调整，相反，对于Ant1而言，这种做法确实用得过于频繁。另一种修改将影响核心任务库（coretask library），这是随Ant附带的任务库。以后将试图以一种“类CPAN”（CPAN-like）的方式（注1）来管理任务库。任务的存储库将在线维护，从而使每个人都能得到。构建文件需要引用在线的库，由此来为特定任务下载JAR或类。开发人员不再需要管理其内部的Ant部署。

除了新的设计需求外，维护人员还试图重构许多原来已知的Ant缺点，特别是XML处理例程中的缺陷。目前，Ant可解释XML，但是仍然要将整个构建文件加载到内存中，这是一种相当低效的方法。大的构建文件将导致Ant的JVM占用过多的系统内存，这会使性能下降。考虑到开发人员在其机器上一天可能会将一个构建运行5次、10次甚至20次，因此构建时间上5分钟的差异就可能导致3个小时的工作量。

总的来说，目标就是要有这样一个系统：相对于原来的Ant而言，要取其精华，去其糟粕。通过阅读*docs/ant2*中的文档，你可以了解到这个新目标，并能跟踪到哪些内容要去除，而哪些内容要加入。自Ant 1.3版本以来，所有Ant发布均包括此文档。希望2002年的某个时候会至少出现Ant2的一个beta版本。

Ant1 RIP 2002?

那么这对于Ant1的用户来说意味着什么呢？Ant2是否就是其终结者呢？并不完全如此。即使前面提到的许多设计建议受到争议和反对（或者希望延迟版本发布），但对于Ant2的设计有一点是不变的：Ant1构建文件在Ant2中不能工作。然而，Ant1已经形成了广泛的用户基础。在许多工程和产品中，Ant1已经建立了难

注1：这指的是Perl公司的已发布CPAN库，这个库允许在没有终端用户干涉的情况下自动地在代码中引用任意数量的模块。

以动摇的地位。例如，IBM 的 VisualAge for Java 目前在其 IDE 中就包括对 Ant 1.2 的支持。WebLogic 6.1 附带了 Ant 1.3 版本的库；所有 Ant 1.3 的示例文档都使用 Ant 来构建所包含的示例。所有 Jakarta 工程都使用 Ant1 构建文件，不过有一些仅在 Ant 1.2 下工作（但在最新版本下也可正常工作）。由于 Ant 当前的使用情况以及其巨大的惯性，即使 Ant2 有了 1.0 或 2.0 版本，也不太可能意味着 Ant1 就会退出历史舞台。更有可能的是，即使确实会发生转变，此过程也会相当缓慢。如果你的工程运转良好，既然本可以正常工作，那么对其改变是没有意义的。由于 Ant1 是开源的，所以其支持绝不会只由某些公司来独立担当。这本书以及大量在线文档都可帮助你基于 Ant1 来维护工程。

这是不是说明 Ant2 不太可能有其地位呢？由于 Ant 是 Jakarta 工程的一部分，因此这一点同样也存在疑问。参与开发 Ant 的一些开发人员也对其他 Jakarta 子工程有所贡献。如果 Jakarta 的新版本或新的子工程出现，那么开发人员就有很好的机会来使用 Ant2。只有时间能证明一切。像对所有开源工具一样，最好是在发生改变时对其时刻加以关注。我们在此列出了一些很可能修改的内容，它们可能会对你现在编写构建文件产生影响。如果你计划将来迁移到 Ant2，那么考虑到 Ant2 的设计可能会使你的迁移更为容易。

- 如果依赖于某些不变的特性（其值设置一次，且仅设置一次），你会发现此设计在 Ant2 将消失（而且，从某种程度上说，Ant 1.5 中就会将其取消）。当前的 Ant2 设计建议要求特性可以在任何时刻设置和重新设置。
- 某些看上去重复的任务将集成为一个超任务（supertask）。例如，jar、unjar、zip、unzip 等等，所有这些对类 ZIP 文件完成操作的任务都将联合成为一个称为 archive 的任务。
- 魔法特性的概念将消失。
- SYSTEM XML 项的使用可允许你通过使用 XML 操作符来动态地包含构建文件段，这一点将由于一个内置的 Ant “包含” 系统的出现而被取消。
- 最重要的是，无论 Ant1 任务如何声明或实现，都不会在 Ant2 中工作。按照设计建议，它们可以作为适配程序和实用程序，从而有利于 Ant1 任务的移植。

Ant2的目标是对Ant1有显著的改善。大多数Ant1所具有的构建和工程设计概念仍将保留。对于由Ant学到的一切，从“构建－设计”的角度来说绝不会浪费。

附录二

Ant 解决方案

时光荏苒，开发人员已经为构建创建了许多很好且一致的解决方案。遗憾的是，这其中的一些解决方案仍然局限于其工程的构建文件，而且很少见诸于文献或文档中。第三章中，我们以一种特定的方式对irssibot工程进行了管理，经证实这种方式在许多其他工程中也取得了成功。我们的示例工程与这个工程有一个类似的特点，即都是将构建文件与文件和目录组织紧密地结合在一起。由于irssibot工程从本质上来说非常简单，而其他一些成功的构建设计（诸如 Tomcat、Jakarta Taglibs 和 Ant 工程中的构建设计）则无此特性，因此我们无法加以说明。不过，在此附录中，我们有机会来讨论这样的一些常见构建解决方案，并对本书其他部分中所用的一些方案进一步澄清。

需要说明，对于以下解决方案，我们并不是从纯学术的角度来介绍的。为了强调此区别，在此将解决方案分解为不同的部分，而且对于每个解决方案均从一个问题开始谈起。如果你发现某个问题与你在编写构建文件时遇到的问题很类似，那么请阅读相应的解决方案，看看这里的建议是否满足你的需要。

测试库的可用性

问题：我们在 Windows 2000 工作站上用的是 Java SDK 1.4 版本，但是 Linux

主机仍然使用 Java SDK 1.2 版本。我们的一些类所使用的类仅在 Java SDK 1.4 版本下可用。现在，这种情况意味着在 Linux 机器上会出现构建错误。如何避免出现这些错误，而且无需编写两个不同的构建文件呢？

回答：使用 `available` 任务来防止出现库版本问题，而且要基于开发人员的环境，只构建应用中必要的部分。在以上情况下，你可以通过检查某些类是否存在来确定 Java SDK 的版本。对于 Java SDK 1.4 及更高版本，`java.lang.CharSequence` 类是惟一的，这个类在 Java SDK 1.4 以前的版本中并没有。Ant 自己的构建文件会完成这一类检查。我们将用 Ant 的构建文件来显示一个示例。

在 Ant 的构建文件中，存在以下目标（已经做了编辑以保留空格）：

```
<target name="check_for_optional_packages">
    <available property="jdk1.2+" classname="java.lang.ThreadLocal" />
    <available property="jdk1.3+" classname="java.lang.StrictMath" />
    <available property="jdk1.4+" classname="java.lang(CharSequence" />
</target>
```

在构建文件的后面部分，如果某些库不可用，我们就不会编译相应的文件：

```
<exclude name="${ant.package}/util/regexp/Jdk14RegexpMatcher.java"
unless="jdk1.4+" />
```

`exclude` 是 Ant 主构建目标的一部分，Ant 的主构建目标将编译 Ant 的所有源代码，而 `exclude` 则排除对 `Jdk14RegexpMatcher.java` 中类的编译，除非 Sun JDK 1.4 版本的库存在。如果你知道开发人员的环境可能大相径庭，那么这种解决方案就非常不错，而 Ant 正是这种情况。只需寥寥几行，这个使用了 `available` 任务并结合 `exclude` Data Type 和 `unless` 属性的构建文件就可以使开发人员更加关注于工程本身，而不必对某个特定但可能不必要的库和工具过分强调。反之亦然，构建管理人员也能够避免某些编译问题，而这些问题是你在检测 `ClassNotFoundException` 异常和错误时通常都会遇到的；对于一些大型的工程（往往有分散的开发小组），这种情况不仅使调试非常困难，而且相当乏味。

与其保证干净，不如适当清除

问题：我们的开发人员在测试应用时出现了问题。例如，两次构建中并没有修改任何代码，但是却得到了不同的行为。从理论上讲，这不应发生，但是由于编译步骤是隐藏的，所以我们不能确认每次构建时会出现什么情况。如何对此加以修正呢？

回答：每次构建都应当能够使工程目录回到其初始状态。看上去这可能是“显而易见”的，但是，对于大多数构建环境（包括 Linux 的核心构建、我们在第二章中的例子、Ant 以及 Tomcat）这一点却并不成立，其原因往往不是很明确。这种模式源于三个常见的构建目标：已编译工程的发布、源代码发布以及测试。

在发布已编译工程时，构建管理人员要求只有安装、运行和支持最终产品所需的文件才对终端用户可用。除了其他原因之外，这样做更是为了使用户能够更为轻松。最好的解决方案是有一个完全分离的空间来创建发布（如 *dist* 目录）。这样就不仅为构建文件提供了一个可以放置可发布组件的位置，而且在需要一个新的发布时还有一个可以进行清除的位置。

发布源代码也是类似的。要使用一个单独的目录来管理将打包和发布的源代码，即要有一个用于复制文件的位置，还要有一个用于删除文件的位置（也就是说，将其全部清除）。较之于将特定的目录复制并打包到工程的源代码中，这样做更有意义。而且，增加和删除一个发布的组件，可以使一个单独的发布阶段更加容易（对于二进制版本这一点也适用）。类似于二进制发布，清除是相当快速的，而且也很容易。尽管你可能认为发布源代码是工程的一个次要目标，但是请做以下考虑：通过令源代码可以发布，你就能够使新的开发人员对于你的工程很轻松地“上手”。如果工程历时时间很长，需要长达数年之久，那么在整个开发过程中开发小组不太可能一成不变。如果能够令新的开发人员更容易地投入到一个进行到一半的工程中，那么这种开发就会使这些开发人员更快地取得更多的成果。

由以上两条规则对解决方案加以结合，可以使我们得到第三条规则，它可使测试更为容易。要进行测试，则需要工程的确定性状态。通过将工程的

发布与其工作和源代码位置加以分离，我们就能够有一种简单的方法来验证所用的是何种版本的工程。如果我们总是不能肯定，认为一个老版本的类会引起混乱，则应该删除此发布再重新创建。无乱则无怪。作为意外收获，我们还可以以一种易于管理的包（如JAR）将这种易于管理的发布发送至其他的开发人员，从而确认错误，并验证测试用例。

使用 Ant 来巩固库

问题：我们的工程使用了至少 15 个不同的 JAR。命令行已经达到了其字节限制，而且构建文件特性也很不好看，同时很难维护。更糟的是，我们可能还要增加更多的 JAR。该如何是好呢？

回答：利用任务的功能来解决问题。很常见的一种情况是，开发人员利用强大的工具来解决简单的问题，然后再根据惯例和“开发人员的主观经验”解决复杂问题。例如，一个 Web 应用可能依赖于大量常用的库。将所有库包含进来的一种快速的解决方案是建立一个 classpath 特性，在一个路径上列出各个 JAR 文件或类目录。运行此 Web 应用时，对于特定的应用服务器，其启动和配置文件中的这一列表当然有不同的目录。对库列表的任何修改都需要对至少两个文件进行修改。

我们可以像下面这样利用一个目标，从而将多个库作为一个 JAR 加以管理：

```
<target name="makesuperjar">
    <jar jarfile="superjar.jar" destdir="${some.common.lib.dir}">
        <zipfileset src="${some.common.lib.dir}/jaxp.jar"/>
        <zipfileset src="${some.common.lib.dir}/jaxen.jar"/>
        <zipfileset src="${some.common.lib.dir}/parser.jar"/>
        <zipfileset src="${some.common.lib.dir}/netcomponents.jar"/>
        <zipfileset src="${some.common.lib.dir}/oracle816JDBC.jar"/>
    </jar>
</target>

<property name="classpath" value="${some.common.lib.dir}\superjar.jar"/>
```

在此通知 Ant 将一个应用所需的所有 JAR 都结合至一个 superJAR 中，该 JAR 将用于整个构建。对于应用所用的所有不同的库，通过取消对它们的

跟踪还可以使发布和安装更加容易。启动脚本和配置文件现在可以表示为 *superjar.jar*，而且我们知道其中包含了工程所需的所有第三方库。

注意：在 Ant 1.2 中，*jar* 任务不能包含一个嵌套的 *<zipfileset>* 元素。

这一技术所带来的好处绝不仅仅是简单的库管理。EJB 有着特别的库管理问题。另外，它对于主应用服务器，处理系统范围的库以及以不同的方式共享 EJB 类并无帮助。你可能需要对同样的类和 JAR 重新打包多次以包含在一个 WAR 或 EAR 中，这要取决于目标应用服务器。有效地使用 *war*、*ear* 和 *jar* 任务可以解决这些问题，而仅需很少的管理开销。

凡事都有好的一面，也有不好的一面，这种解决方案也不例外。诸如 Ant 这样的工程，往往有着众多而分散的开发人员，而且有一个松散管理的开发环境，那么就不能使用这种解决方案。在此若遗漏了 JAR 就会导致 *jar* 任务失败，但是，“可能会遗漏 JAR”这本身就是 Ant 工程设计的一部分。不过，那些有一个紧密管理的库列表的工程，特别是没有“可选”库的工程，将特别受益于这种模式。通过使用一个 JAR 来加强库集合，工程中将不再存在模糊库的可能性（在运行和测试应用时，正是由于这些模糊的库才带来了问题）。它要求第三方库对于应用是分布的，而对于这种情况，这一点很好，因为库的版本和可用性很重要。另外，令人头疼的命令行限制可能会阻碍某些应用服务器管理，而现在也不再是问题。无论是有 2 个还是 20 个 JAR，类路径中都只会定义一个，从而为命令留出更多的空间来放置其他选项和路径。通过提供一个位置来管理应用的所有第三方库，同时通过集中包含这些库并消除版本带来的混乱，可以使构建和工程管理人员更为轻松。

为构建文件目标建立文档

问题：我们有一个复杂的构建文件。当新的开发人员加入到工程时，学习如何使用此构建文件的过程相当长。我们觉得好像更多的时间花在了讲解构建文件做什么上，而不是更好地将新的人员融合到开发队伍中。我们在一个内

部网站上放置了一些文档，它们确实有所帮助，但是我们都知道，并非所有人都会阅读这些文档。我们还能做什么呢？

回答：不要忽略目标的 `description` 属性。运行 Ant 时带有 `-projecthelp` 选项，从而得到更为详细的反馈，这是缩短开发人员熟悉工程的时间以及增加工作效率的另一个办法。

例如，以下目标使用了 `description` 属性从而加入了一些文档：

```
<target name="do-something-really-complex"
        depends="less-complex-stuff"
        description="Perform a lot of checks and tests so you know your project
works right. You wouldn't know this target did this unless we provided a
decription. It's good that you know now, eh?">
...
</target>
```

当然，在此有可能矫枉过正。只应对需要由命令行执行的目标进行描述。`-projecthelp` 列出所有目标时，它会将有描述的和无描述的列表加以分离。许多开发人员可以很快掌握这一点，并了解到要避免运行无描述的目标。对于不应由命令行运行的目标，如果你确实需要对其建立文档，则应在这些目标描述中包括如下的消息：“DON'T RUN THIS FROM THE COMMAND LINE!(不要由命令行运行！)”。

在构建文件外设置特性

问题：我们试图建立一个通用构建文件，但是某些特性值在各个开发人员的工作站上都有所不同，这样就出现了问题。我们不希望开发人员编辑此构建文件，那么该如何做呢？

回答：Ant 的 `property` 任务有一个属性，它可以取一个文件名。如果此文件构建为一个特性文件（每行有一个名 - 值对），则 Ant 将这些值包括在特性表中。如果一个名字与构建文件中指定的名字相冲突，而且在构建文件中指定的定义之前包括了此特性文件（这一点很重要），那么这个值就会覆盖构建文件的值。关键是特性的第一次定义最为优先。

通过声明一个构建所用的外部特性文件，就为开发人员提供了一个办法，从而可以覆盖特性值而无需编辑构建文件。以下构建文件代码段显示了一个例子，即构建文件如何包括一个外部特性文件，在此 codebase 特性需要被覆盖：

```
<project name="ExtenisbleProject" default="all" basedir=".">
    <property name="build.properties"/>
    <property name="codebase" value="/export/home/builduser"/>
    ...the rest of the properties and buildfile
    ...

```

此构建文件在内部设置特性前先由 *build.properties* 导入了特性。如果一个开发人员为 *build.properties* 文件中的 codebase 指定了一个值，那么开发人员所指定的值将更为优先，因为它是最先定义的。否则，如果开发人员没有在其特性文件中指定一个 codebase 值，那么就会使用构建文件中设置的 codebase。

特性文件只是简单的文本文件。以下示例为在特性文件中设置一个特性（在此为 codebase）应当使用的语法：

```
codebase=c:/src
```

要记住 Ant 先处理命令行，因此命令行上 codebase 的值将最为优先。

```
ant -Dcodebase=c:/src
```

在这种情况下，由于 codebase 是在命令行上指定的，所以无论构建文件或特性文件中设置了什么值，其值都会使用 c:/src。

使用 pathconvert

问题：我建立了一组使用 path DataType 的路径。但现在此任务只取作为属性的路径。我需要再使用特性全部重写吗？path 是不是能够解决这个问题呢？

回答：路径最初引入时，尽管合法，但是颇令人烦恼。有些人编写其自己的任务来处理转换，而另一些则费力去维护两组数据，或者坚持使用特性。不过，自从 pathconvert 任务诞生以来，一切都归于正常。

下面来看以下的构建文件示例：

```
<project name="test" default="test" basedir=".">>

    <path id="classpath">
        <pathelement location="lib"/>
        <pathelement location="lib/test.jar"/>
    </path>
    <property name="somepath" value="lib:lib/test.jar"/>
    <target name="test">
        <java classname="org.oreilly.Test" fork="yes" \
            classpath="\${somepath}"/>
    </target>

</project>
```

假设 java 任务必须使用 classpath 属性而不是一个 path DataType (注 1)。惟一的办法就是既用特性又用 DataType 来表示一个路径，从而定义两个路径（这是没有 pathconvert 的情况）。在此例中，看上去并不是大问题，不过这些路径都非常长而且也很复杂。你只希望将其定义一次。以下示例则使用 pathconvert，从而将一个 path DataType 转换为一个特性设置：

```
<project name="test" default="test" basedir=".">>
    <path id="classpath">
        <pathelement location="lib"/>
        <pathelement location="lib/test.jar"/>
    </path>
    <target name="test">
        <pathconvert targetos="windows" property="somepath" \
            refid="classpath"/>
        <java classname="org.oreilly.Test" fork="yes" \
            classpath="\${somepath}"/>
    </target>
</project>
```

利用 pathconvert，我们就不必再有两个路径。pathconvert 任务将 path DataType 转换为一个特性，而它则可由 java 任务来引用。

不过并不是一切都很完美。pathconvert 任务要求定义一个目标操作系统

注 1：这个概念并不是空穴来风。如果构建文件是一个级联构建文件设计的一部分，需要路径的任务就不能依赖于存储在 path DataType 中的路径。它们都需要使用特性，或者是需要重定义路径本身的子工程构建文件。

(或一个路径分隔符)。通常，Ant 会负责做此工作，但遗憾的是，对于 pathconvert 则并非如此。你可能希望使用一个额外的构建文件特性来指出处理构建文件所在的平台。

注意：将 DataType 转换为特性并不是 pathconvert 能够完成的惟一的工作。更详细的内容请查看第七章的文档。

使用声明

问题：在第三章的示例中，你有一个 all 目标来构建所有内容。默认情况下我不希望构建做任何工作，对此有没有一种好方法呢？

回答：当然没有问题，不过要达到这个目的，一个好办法就是建立一个“使用声明”目标。通常，此目标称为 help，它只是显示出构建中常用目标的有关消息。这一操作类似于某些 Unix (和某些 Windows) 的控制台程序。如果对这些控制台程序不熟悉，由命令行调用这些程序时，如果未带参数，它们通常什么也不做。实际上，它们会显示一条消息，以“Usage”文本开头，并显示对此程序可用的各个命令行参数。我们的 help 目标也完成同样的工作，如下例所示：

```
<project name="usage_example" default="help" basedir=".">
    <!-- some properties -->
    <!-- some paths -->

    <target name="build-lib"/>
    <target name="build-app"/>
    <target name="deploy-app"/>
    <target name="makedoc"/>

    <target name="help">
        <echo message="Build the usage_example project"/>
        <echo message="Usage: ant [ant options] <target1> \
                      [<target2 | target3 | ... ]"/>
        <echo message="" />
        <echo message="      build-lib - build just the project's library"/>
        <echo message="      build-app - build the library and \
                      the application"/>
```

```

<echo message=" deploy-app - ready the \
              application for deployment"/>
<echo message=" makedoc - generate all the \
              documentation for the project"/>
<echo message=" -projecthelp - (An Ant option) Display all \
              target descriptions"/>
</target>
</project>

```

现在，如果开发人员从命令行只是调用 *ant*，则会得到如下例所示的消息：

```

src%: ant
Buildfile: build.xml

help:
[echo] Build the usage_example project
[echo] Usage: ant [ant options] <target1> [<target2 | target3 | ... >]
[echo]      build-lib - build just the project's library
[echo]      build-app - build the library and the application
[echo]      deploy-app - ready the application for deployment
[echo]      makedoc - generate all the documentation for the project
[echo]      -projecthelp - (An Ant option) Display all target descriptions

BUILD SUCCESSFUL

Total time: 0 seconds
src%:

```

对于使用你的构建文件的其他人来说，这样的使用说明文本是一个很好的指导，而且有一个 *help* 目标作为默认目标还可以避免意外地执行其他目标。为你的构建文件创建 *help* 目标是一个好的习惯，而且编写这样的目标并不会太费劲。

创建新进程

问题：在构建过程中，我们需要在某些文件上运行一个 Java 实用程序。当此实用程序中出现一些错误时，我们注意到构建已经停止，但没有得到任何成功或失败的消息，而且日志也只是突然结束。我认为可能是第五章提到的 *System.exit()* 问题，但是不知道如何加以修正，我们该怎么做呢？

回答：是的，确实是 *System.exit()* 问题。更确切地说，这个问题是开发人员在

其代码中误用 `System.exit()` 调用造成的。`System.exit()` 调用可以直接与 JVM 通信，这就导致它会立即终止。由于从 Ant 运行的一个 Java 程序是在 Ant 的 JVM 中运行的，所以对 `System.exit()` 的任何调用都会终止 Ant 的 JVM。这一点很不好，不过幸运的是，`java` 任务有一个称为 `fork` 的属性。

```
<java classname="org.oreilly.SpecialTool" fork="yes" />
```

`java` 任务的 `fork` 属性使你能够避免这个问题。此属性通知 `java` 任务在一个单独的 JVM 中运行类。“在一个单独的 JVM 中”，这意味着程序的 `System.exit()` 调用不会终止 Ant 的 JVM。尽管这样做可以保证你的构建不会出乎意料地中断，但是问题并没有完全解决。第 2 个 JVM 仍会不带任何警告地停止。最好还是从根本上解决，即尽可能不用 `System.exit()` 调用。

使用级联工程和构建文件

问题：我们注意到，诸如 JBoss 和 Jakarta 的 Taglibs 等工程没有使用第三章所建议的工程结构。相反，它们看上去好像有多个构建文件，每个构建文件对应于一个子工程。这好像采用的是介绍“级联构建文件”时所讨论的设计。它们到底是什么？应当如何使用它们呢？

回答：对于如何构建一个有多个子工程的大工程，有两种选择。一种是将工程看作是一个集成的整体，因此用一个构建文件来构建全部内容。这个构建文件要定义构建工程所需的所有目标，以及整个构建所需的全部数据元素的其他内容。在这个构建文件中，子工程间的依赖关系可以很容易地得到定义并加以维护。目标的打包和部署可以与各个子工程目标适当地相关。这种描述工程构建的构建文件称为一个集成构建文件(*monolithic buildfile*)。

有些复杂的工程包括许多分离的子工程，这些子工程构成一个应用或框架。其组织结构是每个子工程可以完成其自己的构建，而不用担心其他的子工程。组成这种工程构建的构建文件组称为级联构建文件(*cascading buildfile*)。

对于一些小工程，或者是子工程间存在复杂依赖关系的工程，集成构建文件是上选之策。对于定义良好且子工程是封装的工程，更应采用一个级联的构建文件系统。

Ant 中的级联构建文件得到了诸如 Linux 内核等 Unix 工程的启发。其内核就是使用 `makefile` 的一个级联构建的例子。在 Java 世界中，得到最好组织的级联构建当属 JBoss。JBoss 包括一个根工程目录，其中仅有一个构建文件。利用此构建文件，可以构建组成 JBoss 的所有内容。以下是 JBoss 2.4.4 源树中根目录下的子目录列表：

```
JBoss-2.4.4-src/
    build.xml
    /contrib
    /jboss
    /jboss-j2ee
    /jbosscx
    /jbossmq
    /jnp
    /jbosspool
    /jbosstest
    /jbosssx
```

根目录包括一个构建文件，每个子目录也包括一个构建文件。由于由子目录调用构建文件的模式在各级上都是重复的，因此只需说明根构建文件如何调用其他构建文件即可。除了目录名有所改变，所有内容都是一样的。

以下为 JBoss 的构建文件中的 `build` 目标：

```
<target name="build" depends = "cvs-co,init">
    <ant antfile="src/build/build.xml" dir="jboss" target="main" />
    <ant antfile="src/build/build.xml" dir="jnp" target="src-install" />
    <ant antfile="src/build/build.xml" dir="jbosssx" \
        target="src-install" />
    <ant antfile="src/build/build.xml" dir="jbossmq" \
        target="src-install" />
    <ant antfile="src/build/build.xml" dir="jbosscx" \
        target="src-install" />
    <ant antfile="src/build/build.xml" dir="jbosspool" \
        target="src-install" />
    <ant antfile="src/build/build.xml" dir="jboss-j2ee" \
        target="src-install" />
</target>
```

可以看到，此目标使用了 `ant` 任务来调用各个子工程的构建文件。如果在 JBoss 的主源代码目录下调用 `ant build`，Ant 将试图构建整个工程。`ant` 任务只在其目标构建文件成功的情况下才是成功的。因此，`build` 目标要成功，它调用的所有其他构建文件都必须成功。

至此为止，级联构建文件看上去很简单，但是要以这种方式调用构建文件必须理解一些规则。有一些属性可能会改变 Ant 的行为，而你对此可能未曾预料到。

第一条规则所处理的是特性和特性的作用域。要记住你可以在命令行设置特性，也可以通过 `<property>` 数据元素来设置特性。默认情况下，由父构建文件得到的特性或来自命令行的任何特性都会传递给 `ant` 任务调用的任何子构建文件。如果一个特性在子工程的构建文件中再次声明，那么没有任何影响。特性的不变性仍然适用；根构建文件的特性（以及命令行特性）仍保留。不过，如果在第一个 `ant` 任务上增加以下 `inheritAll` 属性，那么就不再看到这种“不可变特性”的行为：

```
<ant antfile="src/build/build.xml" dir="jboss" \
    inheritAll="false" target="main" />
```

通过将 `inheritAll` 设置为 `false`，就可以通知 Ant 不要令当前构建文件的特性对子工程的构建文件可用，只有一个例外，即命令行特性。可以认为命令行特性具有无上的优先性。这些特性不能改变也不能被忽略。

遗憾的是，级联构建文件的第二条规则不如特性规则那么灵活。`DataType` 引用（带有 `id` 属性的 `DataType`）并不沿着工程树向下传递。仅此而已。若没有利用 `pathconvert` 任务（请见前面的介绍）将路径转换为特性，就没有办法放松这一限制。

这些规则还都指出了一个通用的解决方案，而对于许多编程来说此方案都成立，这就是建立全面的文档，而且要建立很好的文档。如果你的工程使用了级联构建文件，就需要对构建文件增加注释，并编写 `README` 来解释为什么某些特性为第三层子工程设置，而另一些则不是。还要解释为什么某些路径要在每个构建文件中重新定义。若要对 3 个月以来都未碰过的构建文件进行修改，那么这也会对你有所帮助。

词汇表

assemble	汇编	refactor	重构
attribute	属性	reflection	反射
distribution	发布	repository	存储库
DOM (Document Object Model)	文档对象模型	runtime	运行时
EJB (Enterprise Java Beans)	企业 Java Beans	serialize	串行化
library	库	SOAP (Simple Object Access Protocol)	简单对象访问协议
package	包	target	目标
parse-time	解析时	task	任务
property	特性	view	视图
recorder	记录器	XSLT (XSL Transformations)	XSL 转换