**OpenDoc** Series'

# Using the Rake Build Language

V1.0

作者：Martin Fowler
译者：DigitalSonic

# 文档说明

**参与人员：**


| 作者 | 联络 |
|---|---|
| DigitalSonic | ding_x_f@yahoo.com.cn |
| | |


**(at) 为 email @ 符号**

**发布记录**

| 版本 | 日期 | 作者 | 说明 |
|---|---|---|---|
| 1.0 | 2007.06.1 | DigitalSonic | 翻译 |
| 1.0 | 2007.06.28 | 夏昕 | 文档格式编排 |
| | | | |
| | | | |
| | | | |
| | | | |

**OpenDoc 版权说明**



**Open Doc Series 目前包括以下几份文档：**

- **Spring 开发指南**
- **Hibernate 开发指南**
- **ibatis2 开发指南**
- **Webwork2 开发指南**
- **持续集成实践之 CruiseControl**

**以上文档可从http://www.redsaga.com获取最新更新信息**

# Using the Rake Build Language

*http://www.martinfowler.com/articles/rake.html*
*Martin Fowler*

Rake 是一种构建工具，用途和 Make、Ant 差不多。同它们一样，Rake 也是一种领域专用语言(Domain Specific Language，DSL)，区别在于 Rake 是用 Ruby 语言编写的内部 DSL。在本文中，我将介绍 Rake，并讲讲我在用 Rake 构建这个网站的过程中碰到的一些有趣的事情：依赖模型、综合任务、自定义构建规程(custom build routines)和调试构建脚本。

最后更新：2005 年 8 月 10 日

我使用 Ruby 已经有很多年了。我喜欢它那简洁但功能强大的语法，还有它那些写得很不错的库。几年前我把我的很多站点生成器从 XSLT 转成 Ruby 的，我对这个改变非常满意。

如果你是我的一个老读者，那么对于我的整个站点是自动构建的这一事实一定不会太惊讶。我以前使用 Ant (Java 世界中非常流行的构建环境)来完成这一工作，因为它很合适 Java 的 XSL 处理器。因为现在我多数时间在用 Ruby，所以使用 Rake 更多些，Rake 是 Jim Weirich 开发的基于 Ruby 的一种构建语言。最近我在构建过程中完全用 Rake 替代了 Ant。

刚开始时，我用一种与 Ant 相似的方式来用 Rake。后来，我尝试了一些不同的方式以便能研究下 Rake 的某些有趣的功能。结果就有了这篇深入研究其中某些方面的文章。Rake 是我用过的第三种构建语言。我很多年前用过 Make(我都不记得那是多久前了)。在过去的 6 年多时间里我大多在用 Ant。Rake 拥有这些语言的很多功能，还有一些新的特性(对我而言)。虽然 Rake 是用 Ruby 写的并且大量使用了 Ruby 的语言特性，但你仍然可以在各种自动构建过程中使用它。对于简单的构建脚本，你不需要知道 Ruby，但随着事情变得越来越有趣后你需要了解下 Ruby，好让 Rake 充分发挥功效。

这有点偏题，我不打算写一个 Rake 的教程——我将关注我觉得有意思的东西而不是完全覆盖所有内容。我不想假定读者了解 Ruby、Rake 或者任何构建语言。我会在进行过程中稍微解释下 Ruby 相关的内容。如果你已经用过这些东西，或者对不同的计算模型有兴趣，这篇文章值得一读。

## 基于依赖编程(Dependency Based Programming)

我前面讲到的“不同的计算模型”对于一种构建语言而言是否太大了？其实不然，我用过的所有构建语言(Make、Ant (NAnt)和 Rake)都使用基于依赖风格的计算而不是通常的命令风格。这一点引导我们从不同的角度去思考如何编写它

们。大多数人并没有察觉到这一点，因为绝大部分构建脚本都很短，但它的确是一个重要的不同点。

下面来看个例子。让我们想象下，我们要写段程序来构建一个项目，构建过程中有这么几步：

- CodeGen：获得数据库配置文件，并使用它们生成数据库结构和代码来访问数据库。
- Compile：编译应用程序代码。
- DataLoad：将测试数据写入数据库。
- Test：运行测试。

我们需要保证这些任务能够单独执行并正常工作。在执行完前几个步骤后才能进行测试。Compile 和 DataLoad 需要先运行 CodeGen。这些规则需要如何来表示呢？

如果用命令式的方式，可以把每个任务写成一个 Ruby 过程。

```
# this is comment in ruby
def codeGen  #def introduces a procedure or method
  # do code gen stuff
end

def compile
  codeGen
  # do compile stuff
end

def dataLoad
  codeGen
  # do data load stuff
end

def test
  compile
  dataLoad
  #run tests
end
```

请注意，这里有个问题。如果我调用 test，那么 codeGen 将会被执行两次。这不会引起什么错误，因为 codeGen 这一步是(假定)幂等的——即调用多次和调用一次没有区别。但这样会花些时间，好在构建是少数的有充足时间可以让它运行的过程。

要修正这个问题可以把几个步骤分解为公共过程和内部过程：

```
def compile
  codeGen
  doCompile
end
```

```
def doCompile
  # do the compile
end

def dataLoad
  codeGen
  doDataLoad
end

def doDataLoad
  #do the data load stuff
end

def test
  codeGen
  doCompile
  doDataLoad
  #run the tests
end
```

现在可以了，不过代码看上去有些乱。这也是个演示基于依赖的系统的好例子。在一个命令模型中，每个过程都调用程序中的几个步骤。而在一个基于依赖的系统中，我们有几个任务并且为它们指定了前置必要条件(pre-requisite)(它们的依赖)。当你调用一个任务时，它会先检查有什么必要条件并进行安排，保证每个必要条件任务被执行，且仅执行一次。所以，我们的例子看起来就会是这样的：

```
task :codeGen do
  # do the code generation
end

task :compile => :codeGen do
  #do the compilation
end

task :dataLoad => :codeGen do
  # load the test data
end

task :test => [:compile, :dataLoad] do
  # run the tests
end
```

(希望你能明白这段代码的意思，我一会儿会适当地解释下它的语法。)

现在如果你调用 compile，系统会检查 compile 任务，发现它依赖于 codeGen。接着系统再看 codeGen 任务，没有前置必要条件。所以系统先运行 codeGen 随后是 compile。这个和命令风格的情况下一样。

一个有趣的案例是 test。此处系统发现 compile 和 dataLoad 都依赖于 codeGen，因此安排 codeGen 先运行，然后是 compile 和 dataLoad(顺序随意)，最后是 test。从本质上讲，任务运行的实际顺序是在运行时由执行引擎指定的，而不是由编写脚本的程序员在设计时决定的。

这个基于依赖的计算模型很适合构建过程，这也是为什么三种构建语言都是用它的原因。用任务和依赖的方式来考虑构建过程是很自然的，构建过程中的大多数步骤是幂等的，我们也不希望多余的工作减缓构建过程。我估计很少有些构建脚本的人会意识到他们正在用自然的计算模型编程，但它的确就是这样。

# 构建领域专用语言(Domain Specific Language for Builds)

我用过的三个构建语言还有另一个共性——他们都是一种领域专用语言(DSL)。只是他们是不同种类的DSL。在我之前用到的术语中：

- Make 是一种使用自定义语法的外部 DSL。
- Ant(和 NAnt)是一种使用基于 XML 语法的外部 DSL。
- Rake 是一种使用 Ruby 的内部 DSL。

Rake 是一个针对通用语言的内部 DSL，这一事实是它与其他两者的重要区别。它允许我随时随地发挥 Ruby 的威力，唯一的代价就是做些看似不太寻常的事情来保证 Rake 脚本是合法的 Ruby 程序。因为 Ruby 不是一个唐突的语言，所以它没有太多古怪的语法。而且 Ruby 是一个完整的语言，我不需要退出 DSL 去做某些有趣的事情——这是 Make 和 Ant 所不能做的。实际上我逐步发现构建语言和内部 DSL 还真是天造地设的一对，因为你往往需要完整的语言才能满足实际的需求，而且你也不再需要很多编写构建脚本的非程序员了。

# Rake 任务(Rake Tasks)

Rake 定义了两种任务。普通任务就和 Ant 里的任务差不多，文件任务和 Make 里的任务相似。如果你对两者都没有概念也不用担心，我会向你解释的。

普通任务最容易解释。这里是一个从我测试环境的构建脚本里取出的一段代码。

```
task :build_refact => [:clean] do
  target = SITE_DIR + 'refact/'
  mkdir_p target, QUIET
  require 'refactoringHome'
  OutputCapturer.new.run {run_refactoring}
end
```

第一行定义了这个任务。在这个语言里，task 是一个很有用的关键词，它引

入一个任务定义。:build_refact 是任务的名字。命名的语法有些麻烦，要用冒号开头，这是因为 Rake 是内部 DSL。

在任务名称后是前置必要条件。这里只有一个:clean。语法是 => [:clean]。我们可以在方括号内放多个依赖，用逗号分开。在之前的例子里你可以看到如果只有一个任务那么可以省略方括号。如果根本就没有依赖也可以省略它(当然也可能有其他原因，稍候我会讲到这个有趣的主题)。

通过在 do 与 end 间写入 Ruby 代码就可以定义任务主体。在这个块中能加入任何我们喜欢的合法 Ruby 代码——我不想在此费神解释这段代码，因为你不需要去理解它。

作一个构建脚本，Rake 脚本(Ruby 爱好者称其为 rakefile)可读性十分高。如果写成等价的 Ant 脚本会像下面这段代码一样：

```
<target name = "build_refact" depends = "clean">
<-- define the task -->
</target>
```

现在你可以把这段代码看作一个 DSL 并模仿它，既然它是一个内部 DSL 你可能会对它是如何作为合法的 Ruby 程序运行感兴趣。实际上 task 并不是一个关键字，而是一个程序调用，它接受两个参数。

第一个参数是一个 hash(相当于 map 或者 dictionary)。Ruby 的 hash 有特别的语法。通常语法是{key1 => value1, key2 => value2}。如果只有一个键值对大括号可以省略，所以在定义 Rake 任务时不用写大括号，这样能简化 DSL。那什么又是键和值呢？此处的键是一个符号——用冒号开头的 Ruby 标识符。你也能用其他的，一会儿我们将看到字符串，你也能用变量和常量——它们用起来似乎更方便些。值是一个数组——相当于其他语言里的列表。我们罗列几个其他任务的名字。如果不用方括号就直接用一个值代替——Rake 会处理单个数组或者直接量的，我必须说这十分方便。

那第二个参数在哪里呢？其实就是在do和end之间的东西——一个块，Ruby中的闭包(Closure)。Rake运行它来构建一个关于这些任务对象的对象图，通过依赖链连接每个对象，每个对象在恰当的时候执行一个块。所有的任务都被创建后，Rake系统靠依赖链来判断需要执行哪个任务，按照什么顺序执行，随后依次调用每个任务的块。闭包的一个关键特性就是它们在计算时无需被运行，它们能被存起来以备后用——即使它们在实际执行时引用了不在范围内的变量。

我们所看到的都是合法的 Ruby 代码，我承认它看起来有些怪，但它让我们能有一个可读性高的 DSL。Ruby 的语法很精炼——像过程参数不需要括号之类的这些语法让这个 DSL 保持紧凑。闭包是另一个重点——它们经常被用来书写内部 DSL，因为闭包允许我们用另一种控制结构来封装代码。

# 文件任务(File Tasks)

我上面说到的任务和 Ant 里的任务很相似。Rake 还支持些不同的任务，称为文件任务，它和 Make 中任务的概念差不多。下面是另一个例子，也来自我 web 站点的 rakefile，稍经简化。

```
file 'build/dev/rake.html' => 'dev/rake.xml' do |t|
  require 'paper'
  maker = PaperMaker.new t.prerequisites[0], t.name
  maker.run
end
```

你可以通过 file 来引用实际文件而不是任务名。所以'build/dev/rake.html'和'dev/rake.xml'是实际文件。html 文件是这个任务的输出，xml 文件是输入。你可以认为一个文件任务是告诉构建系统如何构造输出文件——实际上这就是 Make 里的思想。你可以罗列想要的输出文件，告诉构建程序如何构造这些文件。

文件任务的一个重要部分就是它只有在需要时才会运行。构造系统会检查文件，只有在输出文件不存在或者修改日期早于输入文件时才会运行任务。因此，文件任务在你考虑文件或者以文件为基础的事情时候用起来很顺手。

它与 task 的一个不同之处在于我们把任务对象本身作为一个参数传入闭包——也就是那个|t|。我们现在可以在闭包中引用任务对象并且调用它的方法。我用这种方法来避免文件名重复。可以用 t.name 来获得任务（就是输出文件）的名字。同样也能用 t.prerequisites 来获得前置必要条件。

Ant 里没有与文件任务的近似的东西，而是由每个任务自己执行相同的必要性检查。XSLT 转换任务需要一个输入文件、一个样式文件和一个输出文件，并且只在输出文件不存在或者修改时间比任一输入的文件晚时才执行转换。唯一的问题就是在什么地方处理这个检查——是在构建系统里还是在任务里。Ant 主要用 Java 写好的固定任务，Make 和 Rake 都依靠构建程序作者来给任务写代码。所以把任务的作者从检查文件是否是最新状态的工作里解放出来是很有意义的。

不过在 Rake 里执行最新状态的检查倒是十分容易的，代码看上去就是这个样子的。

```
task :rakeArticle do
  src = 'dev/rake.xml'
  target = 'build/dev/rake.html'
  unless uptodate?(target, src)
    require 'paper'
    maker = PaperMaker.new src, target
    maker.run
  end
end
```

Rake(通过fileutils包)提供了一系列简单的Unix风格的文件操作命令，比如cp、mv、rm等等。还有uptodate?，可以用它来执行此类检查。

我们可以看到要完成这个工作有两种方法，一种是使用文件任务，另一种是使用带 uptodate?的普通任务。该选择哪种呢？

必须承认，我没有一个很好的答案，两者看上去都不错。我决定在新的 rakefile 里尽量使用条理清晰的文件任务。我之前不这么做是因为我知道它是最好的方法，想看看使用它会有什么结果。事情往往就是这样，遇到什么新的东西，反复使用它来发现它的使用边界是个不错的主意。这是个很合理的学习策略。这也是为什么人们总是要反复使用新技术或者是早期的技术。人们常常批评这种做法，但它的确是学习的自然组成部分。如果不超出使用范围进行尝试，那你怎么能知道它的边界呢？重要的是要在一个相对可以控制的环境中进行尝试，这样一来在你找到边界时可以进行适当修复。(总之，在我尝试后我觉得 XML 是构建文件的理想语法。)

我还要说一点就是我还没有发现使用文件任务或者有条理的普通任务有什么问题。我也许会再考虑一年或者两年，但就目前为止我对此比较满意。

# 依赖定义后置 (Defining Dependencies Backwards)

到目前为止，我们已经讨论了 Rake 是如何处理 Ant 和 Make 里的相似事件的。这是一个很棒的组合——把两者的能力和唾手可得的 Ruby 的强大力量组合在一起，但如果仅仅如此还足以让我写这篇文章。引起我兴趣的是那些 Rake 的不同之处。其中的第一项就是 Rake 允许在很多地方定义依赖。

在 Ant 里通过将它们声明为依赖任务的一部分来定义依赖。只要向下面我的 Rake 范例中做的那样就可以了。

```
task :second => :first do
  #second's body
end

task :first do
  #first's body
end
```

Rake(像 Make 那样)允许你在任务初始声明后再添加依赖。这样一来你就能在多个地方描述一个任务。我能像这样来添加近似于前置必要任务的依赖。

```
task :second do
  #second's body
end

task :first do
  #first's body
end
task :second => :first
```

如果任务在构建文件中是一个挨着一个的话，那这种做法没什么区别，但如果是在很长的构建文件里它的确增加了一点灵活性。从本质上来说，这种做法允许你按照常规思路来考虑依赖，或是在添加前置必要任务时添加依赖，亦或者是在第三个地方处理依赖。

和通常一样，这个灵活性同样带来了新问题，究竟在什么地方定义依赖最合适呢？我现在还没有答案，但在我的构建文件里我遵循了两条经验方法。当我在考虑一个任务要在另一个任务之前运行时，我会写依赖任务的时候定义依赖，这个是常规做法。然而我经常会用依赖来对相关任务进行分组，例如不同的勘误页面。在用依赖进行分组时(这是一个结构化构建文件的一个常见部分)，把依赖放在前置依赖任务的附近比较有意义。

```
task :main => [:errata, :articles]

#many lines of build code

file 'build/eaaErrata.html' => 'eaaErrata.xml' do
```

```
  # build logic
end
task :errata => 'build/eaaErrata.html'
```

我并不需要用 task 关键字定义:errata 任务，只要把它作为:main 的依赖就足以定义该任务了。随后再添加单一的勘误文件并将它们分别加到组中。针对此类分组行为，这看起来是个很合理的方法(即使我不用真的在我的构建文件里这么做，一会儿我们就会看到了)。

还有个问题，就是当依赖分散在整个构建文件里时我们怎么才能一下子找到所有的依赖呢？这是个好问题，就让 Rake 来告诉你答案吧，用 rake –p 可以显示每个任务的前置必要任务。

# 综合任务(Synthesized tasks)

综合任务可以让你在定义了一个任务后为它添加依赖，提供完整的 Ruby 支持，为构建引入些额外的技巧。

在我解释综合任务前，我需要先介绍些与构建过程相关的重要原则。构建脚本趋向于两类构建——完整构建(clean builds)和增量构建。完整构建用在输出区为空的时候，在这种情况下所有的东西都由源代码(经过版本控制的)开始构建。这是构建文件所能做的最重要的事情，有个正确的完整构建是最高优先级的。

完整构建很重要，但它们却要花费很多时间。所以，多数情况下增量构建很有用。你的输出目录里已经有了些东西。增量构建会计算出如何以最少的工作量将输出目录内容更新为最新代码。此处会有两个可能发生的错误。第一个，也是最严重的，是重建遗失(missing rebuild)，意思就是一些应该已被构建的项目没有被构建。这种情况是十分糟糕的，原因是它造成输出和输入不匹配，与完整构建的结果不同。另一个稍微好一些的错误是不必要的重建，构建了一个不需要被构建的元素。这个错误相对而言不是那么严重，因为它不会影响正确性，但这个问题增加了增量构建的时间。与增加的时间一样，它有时还会增加些困扰——当我运行 Rake 脚本时，我只期待看到我改变了的东西被构造，否则我会想“它怎么被改动了？”

要保持一个好的依赖结构的重点之一就是保证增量构建能正常工作。我要对我的站点进行增量构建，只要执行'rake'——调用默认任务。我只构建那些我想构建的东西。

这就是我需要的，一个有趣的问题就是将这些运用在我的 bliki 上。我 bliki 的源码是 bliki 目录里的一堆 xml 文件。输出就是每个项对应一个文件，再加上些概要页面——其中主 bliki 页面是最重要的。我希望的就是对一个源文件做的任意修改都能触发 bliki 构建。

我能用以下方式对所有文件进行命名，从而实现目的。

```
BLIKI = build('bliki/index.html')

file BLIKI => ['bliki/SoftwareDevelopmentAttitude.xml',
               'bliki/SpecificationByExample.xml',
               #etc etc
              ] do
```

```
  #logic to build the bliki
end

def build relative_path
 # allows me to avoid duplicating the build location in the build file
 return File.join('build', relative_path)
end
```

这么做的确挺乏味的，而且我还会忘了在添加新文件时把它加入列表中。幸好，我还可以这么做。

```
BLIKI = build('bliki/index.html')

FileList['bliki/*.xml'].each do |src|
        file BLIKI => src
end

file BLIKI do
  #code to build the bliki
end
```

FileList是Rake的一部分，它会根据传入的一部分信息生成文件列表——此处它建立了bliki源代码目录的所有文件列表。每个方法都是一个内部的迭代器，允许我遍历他们并将其中的每一个单独地加入文件任务。每一个方法都是一个集合闭包方法(collection closure method)。

我还为 bliki 任务添加了一个符号任务。

```
   desc "build the bliki"
   task :bliki => BLIKI
```

这样一来我能通过简单调用 rake bliki 来构建 bliki。我不太肯定是不是真的还需要这么做。如果所有的依赖都(像现在这样)被正确地建立起来了，那么可以就用默认的 Rake 任务，并且没有不必要的重建。但目前还是把它放着吧。desc 方法能为后面紧跟的方法定义简短的描述，在我运行 rake –T 时能获得一个包含所有拥有 desc 定义的任务列表。通过这种方法我能知道哪个任务是我需要的。

如果你曾经用过 Make，那你也许会认为这是以前 Make 的一个伟大特性——指定模式规则(pattern rule)来自动生成某类文件。看一个比较常见的例子，你想要通过对所有 foo.c 文件运行 C 编译器构建对应 foo.o 文件。

```
%.o : %.c
      gcc $< -o $@
```

%.c 会匹配每一个以'.c'结尾的文件。$<代表了目标文件，而$@则代表规则中的源文件。这个模式规则意味着你不必为项目中的每个文件添加编译规则，模式规则会告诉 Make 如何构造它需要的任意*.o 文件。(事实上你都不需要在 Make 中写这个规则，因为 Make 自带了很多类似的模式规则。)

Rake 也有类似的机制。我不想深入讨论它，只是点到为止，原因是我还没有发现自己需要它。综合任务已经可以满足我所有的需要了。

# 块域任务(Block Scoping Tasks)

我在使用文件名和依赖时发现了一个问题，就是你不得不重复其中的文件名，举例来说：

```
file 'build/articles/mocksArentStubs.html' =>
'articles/mock/mocksArentStubs.xml' do |t|
 transform t.prerequisites[0], t.name
end
task :articles => 'build/articles/mocksArentStubs.html'
```

在上例中'build/articles/mocksArentStubs.html'在代码中出现了两次。我能在动作块中用任务对象来避免重复，但还是要在建立对整个 articles 对象的依赖时重复它。我不喜欢重复，因为如果我改变了文件名这可能给我带来麻烦。要找个办法只定义一次文件名。我可以就声明一个常量，但这样一来我这个常量就在我 rakefile 里处处可见，而我只想在这个小节里使用它。我比较倾向于变量作用域越小越好。

我可以用上面提到的 FileList 类来解决这个问题，不同的是这次我只用一个文件。

```
FileList['articles/mock/mocksArentStubs.xml'].each do |src|
  target = File.join(BUILD_DIR + 'articles', 'mocksArentStubs.html')
  file target => src do
    transform src, target
  end
  task :articles => target
end
```

这样一来，我定义了只作用于块内代码的 src 和 target 变量。注意一点，它只帮助我在此处定义:articles 任务的依赖。如果我想在:articles 任务的定义处定义依赖，我会需要一个常量以获得跨越整个 rakefile 的可见性。

Jim Weirich 在读了这篇文章的草稿时指出，如果你觉得 FileList 语句太冗长了，你也可以定义一个方法来做件事：

```
  def with(value)
    yield(value)
  end
```

随后做如下调用：

```
  with('articles/mock/mocksArentStubs.xml') do |src|
    # whatever
  end
```

# 构建方法(Build Methods)

将构建语言作为一个完整编程语言的内部 DSL 有一个很大的好处就是我能写程序去处理常见情况。子程序是构造一个程序的最基本方法，缺乏方便的子程

序机制是 Ant 和 Make 的问题之一，特别是在写复杂的构建时。

这里就有一个这样的通用构建方法——我用来将一个 XML 文件转换为 HTML。所有我新写的东西都用 Ruby 来做这个转换，但我还有些旧的 XSLT 文件，没有什么必要急着去改动它们。在写了几个处理 XSLT 的任务后，我很快发现有些重复，所以我为它们定义了一个子程序。

```
def xslTask src, relativeTargetDir, taskSymbol, style
  targetDir = build(relativeTargetDir)
  target = File.join(targetDir, File.basename(src, '.xml') + '.html')
  task taskSymbol => target
  file target => [src] do |t|
    mkdir_p targetDir, QUIET
    XmlTool.new.transform(t.prerequisites[0], t.name, style)
  end
end
```

头两行指出目标目录和目标文件。接着我把目标文件作为依赖添加给提供的任务符号。创建一个带操作的文件任务来建立目标目录(如果需要的话)，并用我的 XmlTool 实施 XSLT 转换。如此一来，当我想建立一个 XSLT 任务时，我只要调用这个方法就好了。

```
xslTask 'eaaErrata.xml', '.', :errata, 'eaaErrata.xsl'
```

这个方法很好的封装了所有的常用代码，并将我此时所需要的所有变量参数化了。我发现把父任务传入子程序很有用，因为这样子程序就能很方便地替我建立依赖——这可以算是另一个灵活建立依赖的方法。我有一个类似的直接从源目录复制文件到构建目录的任务，我用来处理图片和 pdf 的。

```
def copyTask srcGlob, targetDirSuffix, taskSymbol
  targetDir = File.join BUILD_DIR, targetDirSuffix
  mkdir_p targetDir, QUIET
  FileList[srcGlob].each do |f|
    target = File.join targetDir, File.basename(f)
    file target => [f] do |t|
      cp f, target
    end
    task taskSymbol => target
  end
end
```

copyTask 高级一些，因为它能让我指定一系列要复制的文件，可以这样复制文件：

```
copyTask 'articles/*.gif', 'articles', :articles
```

如此可以将源文件的 articles 子目录里的所有 gif 文件复制到构建目录中。它为每个文件建立了独立的文件任务，并让它们都成为:articles 任务的依赖任务。

# 平台依赖的 XML 处理(Platform Dependent XML Processing)

在用 Ant 构建我的站点时，我用基于 Java 的 XSLT 处理器。现在开始用 Rake 了，我便决定改用本地 XSLT 处理器(native XSLT processors)。我同时用 Windows 和 Unix(Debian 和 MacOS)系统，两个系统下都有能很方便得到的 XSLT 处理器。当然它们是不同的处理器，我要分别调用——当然我希望这点在调用 Rake 时对 rakefile 和我而言是透明的。

这又是一个能直接使用完整语言的好处。我可以方便地写一个 XML 处理器根据平台信息进行选择。

让我们从我这个工具的接口部分开始——XmlTool 类。

```
class XmlTool
  def self.new
    return XmlToolWindows.new if windows?
    return XmlToolUnix.new
  end
  def self.windows?
    return RUBY_PLATFORM =~ /win32/i
  end
end
```

在 Ruby 里能够通过调用类的 new 方法建立一个对象。与专门的构造子不同，你能够覆盖这个 new 方法——甚至是返回个不同类型的对象。因此，当我调用 XmlTool.new 时得到的不是一个 XmlTool 的实例，而是我脚本正在运行的平台上的对 XML 处理工具。

两个工具中 Unix 版本的比较简单。

```
class XmlToolUnix
  def transform infile, outfile, stylefile
    cmd = "xsltproc #{stylefile} #{infile} > #{outfile}"
    puts 'xsl: ' + infile
    system cmd
  end
  def validate filename
    result = `xmllint -noout -valid #{filename}`
    puts result unless '' == result
  end
end
```

你会注意到此处有两个针对 XML 的方法，一个是 XSLT 变换，另一个是 XML 验证。在 unix 版本里每个方法都调用一个命令行。如果你不熟悉 Ruby，那就注意下 Ruby 中用#{变量名}来向一个字符串中插入变量。实际上你能在这里插入任何 Ruby 表达式——这真的很方便。在验证方法中我用了倒单引号——执行命令行并返回结果。puts 命令是 Ruby 的标准输出。

Windows 版本的就复杂些，因为要用 COM 而不是简单地用命令行。

```
class XmlToolWindows
  def initialize
    require 'win32ole'
  end
  def transform infile, outfile, stylefile
    #got idea from
http://blog.crispen.org/archives/2003/10/24/lessons-in-xslt/
    input = make_dom infile
    style = make_dom stylefile
    result = input.transformNode style
    raise "empty html output for #{infile}" if result.empty?
    File.open(outfile, 'w') {|out| out << result}
  end
  def make_dom filename, validate = false
    result = WIN32OLE.new 'Microsoft.XMLDOM'
    result.async = false
    result.validateOnParse = validate
    result.load filename
    return result
  end
  def validate filename
    dom = make_dom filename, true
     error = dom.parseError
    unless error.errorCode == 0
      puts "INVALID: code #{error.errorCode} for  #{filename} " +
        "(line #{error.line})\n#{error.reason}"
    end
  end
end
```

require 'win32ole'语句引入了使用 COM 所需的 Ruby 库。这只是程序的常规部分，在 Ruby 里可以先写好导入语句，而库只会在需要时被加载。随后就能像其它脚本语言那样操作 COM 对象了。

你应该已经注意到了，这三个XML处理类没有什么类型关系。之所以能成功进行XML操作是因为Windows和Unix版本的XmlTools都实现了transform和validate方法。这就是Ruby爱好者所谓的动态类型(duck typing)——如果它走起来像只鸭子，叫起来也像只鸭子，那么它肯定就是一只鸭子。没有编译时的检查来保证这些方法的存在。如果一个方法不正确，它会在运行时失败——在测试时这个问题应该会被发现并得到处理。我不想卷入动态对静态类型检查的争论中，只是指出一下这是一个使用动态类型的例子。

如果你正在使用一个 Unix 系统，你需要用任何可以找到的包管理系统去下载我用的 Unix xml 命令(在 Mac 上我用 Fink)。XMLDOM DLL 通常 Windows 中自带，但也有可能需要自己下载。

# 一个意料之外的问题(Going Pear-Shaped)

关于编程，有一件事是肯定的，那就是程序总是会出错。无论你多努力，你所告诉电脑的和它所听到的内容总有些出入。看下这段 Rake 代码吧(这是从我亲身经历过的问题中简化来的)。

```
src = 'foo.xml'
target = build('foo.html')
task :default => target
copyTask 'foo.css', '.', target
file target => src do
  transform src, target
end
```

发现什么错误吗？我也没有发现。我只知道 build/foo.html 总是被构造，即使是在不需要的时候，这是一个非必要的重建。我不知道问题出在什么地方。两个文件的时间戳都是对的，就算我把目标文件的时间调整到晚于源文件，它还是会进行重建。

我调试的第一步是使用 Rake 的追踪功能(rake --trace)。通常只靠它就能确定那些奇怪的调用的原因，但这次却完全没有效果。它只是告诉我'build/foo.html'任务被执行了，但没有告诉我原因。

有人可能会想责怪 Jim 为什么不提供些调试工具。也许这时咒骂能让我感觉好些："你母亲是只克利夫兰来的母狼，而你父亲是根浸湿了的胡萝卜"。(译注：反正是句骂人的话，可能有些文化差异，没有看明白就直译了)

但我有个更好的选择。Rake 是 Ruby 写的，而任务不过就是对象而已。我能获得这些对象的引用，随后去一探究竟。Jim 可能不会把这些调试代码放进 Rake，但我能自己添加。

```
class Task
  def investigation
    result = "------------------------------\n"
    result << "Investigating #{name}\n"
    result << "class: #{self.class}\n"
    result <<  "task needed: #{needed?}\n"
    result <<  "timestamp: #{timestamp}\n"
    result << "pre-requisites: \n"
    prereqs = @prerequisites.collect {|name| Task[name]}
    prereqs.sort! {|a,b| a.timestamp <=> b.timestamp}
    prereqs.each do |p|
      result << "--#{p.name} (#{p.timestamp})\n"
    end
    latest_prereq = @prerequisites.collect{|n| Task[n].timestamp}.max
    result <<  "latest-prerequisite time: #{latest_prereq}\n"
    result << "................................\n\n"
    return result
  end
```

```
end
```

这里是部分代码，如果你不是一个 Ruby 爱好者，你也许会觉得有些奇怪，我为 Rake 的 task 类增加了一个方法。这就类似于一个面向切面的引入(aspect-oriented introduction)，它在 Ruby 里是合法的。和许多 Ruby 技巧一样，它会让你困惑，但只要你够仔细，你会发现它有多棒。

现在我能调用它了，来看看究竟发生了些什么。

```
src = 'foo.xml'
target = build('foo.html')
task :default => target
copyTask 'foo.css', '.', target
file target => src do |t|
  puts t.investigation
  transform src, target
end
```

我把这部分打印出来：

```
------------------------------
Investigating build/foo.html
class: Task
task needed: true
timestamp: Sat Jul 30 16:23:33 EDT 2005
pre-requisites:
--foo.xml (Sat Jul 30 15:35:59 EDT 2005)
--build/./foo.css (Sat Jul 30 16:23:33 EDT 2005)
latest-prerequisite time: Sat Jul 30 16:23:33 EDT 2005
..............................
```

起先我对时间戳感到疑惑，输出文件的时间是 16:42，那为什么任务说是 16:23？后来我意识到任务的类是 Task 而不是 FileTask。Task 类不做日期检查，如果你进行调用，它们始终会被执行。所以我进行了如下的修改。

```
src = 'foo.xml'
target = build('foo.html')
file target
task :default => target
copyTask 'foo.css', '.', target
file target => src do |t|
  puts t.investigation
  transform src, target
end
```

我在该任务被上下文中其他任务涉及前将其声明为了文件任务。问题就这样解决了。

这件事告诉我们，在使用此类内部 DSL 时你可以进入对象内部去探究正在发生的一切。当遇到类似的奇怪问题时，这将是一个很有效的方法。我在遇到的其它非必要构建时也使用这个方法。

(顺便说下，我的调试方法在输出文件不存在时结束，例如一个完整构建。我没有花精力去解决这个问题，因为我只在文件存在时才会去调用它。)

# 用 Rake 来构建非 Ruby 的应用程序(Using Rake to build non-ruby applications)

尽管 Rake 是用 Ruby 写的，但没有理由拒绝用它来构建其他语言写的应用程序。任何构建语言都是用来完成构建工作的脚本语言，你能很轻松地使用由另一种语言写的工具来构建一个环境。(一个很好的例子就是当我们用 Ant 来构建一个微软的 COM 项目时，我们所要做的就是不要让微软的顾问知道而已。)使用 Rake 时最好能懂 Ruby，这样就能做一些比较高级的事情。我认为任何一个专业的程序员需要会至少一种脚本语言，这样就能用它来处理所有的零活。

虽然我本人没有这样做过，但我们有一些在较大型的 Java 项目上使用 Ruby 作为构建语言的经验。当时那个构建脚本已经到了无法使用 Ant 来实现的地步，所以我们用 Ruby 定制了一个构建系统。这个系统很不错，它当然也成了我们在更复杂的项目中的构建系统的又一个选择。我们喜欢它的一个原因是能有一个完整(且可扩展)的语言可以让我们用程序员的术语来描述整个构建：层、模块、jar 和 ear。而且它也更快些。

在结合Java一起使用时，Java虚拟机启动会花费很长时间，这是该构建的一个严重的问题。(该问题也困扰着别的非JVM构建工具。)我的同事Jon Tirsen为了解决这一问题，建立了一个监听socket的antServer程序，Rake程序向服务器发送要执行的Ant任务定义。用这种方法你可以只启动一次Ant重复向它发送命令。他的博客里有更多的信息，但我想应该没有提供代码。他没有用Rake，但这对Rake同样有效。

# 最后的一些思考(Final Thoughts)

综上所述，我发现 Rake 是一个强大易用的构建语言。当然，这与我喜欢用 Ruby 也有点关系，但 Rake 使我确信一个完整语言的内部 DSL 来做构建系统是很有意义的。脚本从很多方面来看都适合用于构建，Rake 只是添加了足够的特性，从而成为一个基于良好语言的优秀构建系统。还有另一个优势就是 Ruby 是一个能运行在任何我需要的平台上的开源语言。

我对那灵活的依赖声明所带来的结果感到惊讶。我能做很多可以减少重复的事情——这让我觉得今后对我的构建的维护工作能变得更容易些。我发现很多放到单独文件中的公共功能被 martinfowler.com 和 refactoring.com 的构建脚本所共享。

如果你正在使用自动构建，那你应该看看 Rake。记住它能被用在各种环境中，而不局限于 Ruby。

# 扩展阅读材料(Further Reading)

你能从rubyforge获得Rake。Rakefile说明是最好的文档之一。Jim Weirich的博客中也包括了些能帮助理解Rake的好文章，但是你需要按倒序来阅读它们(先读早的文章)。这些文章更详细地阐述了我带过的内容(比如规则)。

在脚本的帮助下你能在bash中为Rake建立命令行补全功能。

Joe White有一篇关于Rake库特性的好文章。

如果你喜欢使用内部DSL实现构建工具的想法，但更偏向于Python，那么你也许想看看SCons。

# 致谢(Acknowledgements)

感谢 Jason Yip、Juilian Simpson、Jon Tirsen 和 Jim Weirich 对这篇文章草稿提出意见。感谢 Dave Smith 和 Ori Peleg 在文章发表后对其提出的一些修改建议。

但最重要的是要感谢 Jim Weirich 写了 Rake。我的网站感谢你。

# 重要修改(Significant Revisions)

2005 年 8 月 10 日：第一次发表