Jam/RM 入门教程

Lura Wingerd
Perforce software
Perforce user conference 2010

摘要

Jam/MR(“Jam — make(1) Redux"）是一个软件建构工具。它的命令行语言和处理逻辑已经足以胜任建构任务。Make 也是一种建构工具，尽管它有很大的优势，但是新手们在上手时存在很大难度。本文中，我将会在一系列有趣的示例中逐渐的向大家展示一些 Jam 最有用的特性。从这些示例中透漏出来的基本知识将会给予你一些启示，方便您利用 Jam 先进特性组建一个非常强大的建构系统。很快地，你就会发现手头有一个这样的 Jam 文档将会非常有用。

1 背景资料

Jam 由 Christopher Seiwald 编写，它本身可以免费使用，它的 C++源码可以免费获得，在商业软件和学术软件中的建构中得到了广泛的应用，并且它本身还存在一些变体。它甚至默认安装在 Mac OS X 系统中。但是它并没有能够取代 *make*，关于 Jam 的 O'Reilly 书籍还没有出版。

Jam 最具优势的地方是它的可移植性和速度。它的可移植性受益于 Jam 命令语言与平台依赖性规范的分离，例如：编译和链接命令。至于速度，则部分归功于它的单独调用的处理逻辑。Jam 可执行程序仅需执行一次，这一点有别于 *make*，make 必须递归的自我调用以收集，分析和对出入内容作出处理。

2 Jam 的工作原理

Jam 是一个独立的可执行程序。当你运行该程序时，它分三个步骤来执行任务。第一步，读入包含你的指令的“Jamfiles"（类似于“Makefiles"）。第二步，自动扫描相关的目录（有时可以是一些文件），找出在你的指令中定义的“目标"；Jam“目标”是在建构过程中使用，创建或者更新的文件。第三步，执行系统命令以创建或者更新目标。Jam 处理过程中的三个步骤分别称为语法解析，目标定位，更新修改。在本文结尾，你可能会更加深刻的体会到深入了解这三个步骤的重要。

2.1 运行 Jam

如果你有一个可以运行的“Jam"，你可以自己尝试运行在本文中列出的 Jam 的例子。把下面的内容放入一个文件中，然后运行：

```
jam -f yourfile
```

（通常，你需要把 Jam 命令放到一个“Jamfile”文件中，如果你不使用“-f”标记，Jam 将会默认查找该文件。而且，Jam 默认调用一个称为 Jambase 的设置文件，这可能会为这里的例子增加不必要的复杂性。）

你可以使用很多 Jam 命令行标记来显示有启发意义的诊断信息：

-d2 诊断输出等级 2：显示用于创建或者更新被编译文件的系统命令。
-d5 诊断输出等级 5：显示你的 Jamfiles 的解析过程（哪个步骤正在运行，哪个变量正在进行设置，等等）。尝试在下面的示例中使用该标记。
-n 运行而不实际执行任何更新的命令。
-a 重建所有的目标，不管是否需要。

你可以对这些标记进行组合使用，例如：

```
Jam -nd5a   -fyourfile
```

3 Jam 语言

在开始介绍 Jamfiles 的示例之前，让我们来看一些 Jam 简单却不直观的语言的有趣的方面。

3.1 Jam 语法

Jam 语言有一个非常直观的语法，尽管这样，它仍然会使新手们产生混淆。该语言最不容易理解的语法规则如下：

- 区分大小写。
- 声明内容（成为：“标识符”）必须以空格隔开。
- 每一个声明必须以一个分号结尾。

下面是一些示例：

```
X = foo.c ;
```

这是一个单独的 Jam 声明，把“foo.c”赋值给“X”。

```
X = foo.c ; x = bar.c ;
```

这一对声明给两个 Jam 变量进行了赋值，首先把“foo.c”赋值给“X”，然后把“bar.c”赋值给“x”。

```
X = foo.c;
```

这是一个不完整的声明语句，它将会产生语法错误，可能还会导致无法解释的错误。当 Jam 读取到这个语句时，它将会把“foo.c;”赋值给“X”，而不去理会后一行会产生什么结果。为什么呢？因为在分号前面缺少了一个空格，而分号用来表示声明语句的结束。

Jam 语言中的声明用来赋值或者调用一个“规则”。（一个“规则”可以说是一个程序）你可以通过一个等价的标记来区分该声明是赋值语句还是调用语句。例如，上面所有的声明都是赋值语句。下面是一些调用语句的声明：

```
X foo.c ;
```

该语句调用了一个“X”规则，并且给它传递了一个参数，“foo.c”。

```
X=foo.c ;
```

这是可以完全接受的 Jam 声明，但是它不是你想象的那样。该声明执行了一个没有参数“X=foo.c” 规则。为什么它不是一个赋值声明呢？因为在“=”号后面不存在空格。

通常，Jam 也可以识别第三种类型的声明。以确定的关键字开头的声明被称为“控制流”

（follow-of-control）声明。这些关键字很容易识别，通常会包含“if”，“for”，“switch”和“include”。

在下面的例子中将要累介绍控制流声明和 Jam 语言的附带语法。在本节结尾，你可能已经对 Jam 的工作机制有了足够的了解，而且在你阅读了 Jam 文档之后，你会对本文中没有涉及到的 Jam 的语法和功能有一个更加深入的理解。

3.2 常字符（Literals）

在 Jam 语言中常字符不需要被引用。在 Jam 中，除了变量名，规则名，或者操作符之外的就是常字符了，并且常字符都是字符串。（在 Jam 中除此之外就没有其他的数据类型了!）例如：

```
X = foo.c ;
Foo.c = X ;
```

上面两行代码的意思是：把变量“X”赋值为常字符量“foo.c”，然后给变量“foo.c”赋值为“X”。

然而，当涉及到的常字符中包含空格，等号，或者其他字符时，引用就是必需的，并且 Jam 将会给出不同的解释。例如：

```
X = "this ; and ;this" ;
```

把字符串“this ; and ; this”赋值给变量 X。

3.3 变量(Variables)

Jam 中的变量值是一些字符串。一个单独的 Jam 变量可以被赋予一个字符串列表：

```
X = a b 1 "2 3" ;
```

把一个字符串列表，“a”，“b”，“1”和“2 3”赋值给变量 X。

Jam 变量名也是字符串。你可以为一个变量进行任意的命名：

```
"My dog has fleas!" = yes ;
```

这里的变量名是“My dog has fleas!”，而它的值是一个单独的字符串“yes”。

你可以一次为多个变量进行赋值：

```
My dog has fleas! = yes ;
```

变量“My”，“dog”，“has”，和“fleas!”被赋值为“yes”字符串。

3.3.1 变量展开(Variable Expansion)

在 Jam 中队变量值的访问称为展开。最简单的方式是，你可以使用“$(变量名)”来展开该变量。例如：

```
X = This is a message. ;
Echo $(X) ;
```

调用 Jam 的内建“Echo”规则来输出赋值给 X 的字符串列表。换句话说，就是输出：

```
This is a message.
```

你也可以在赋值声明的左边展开变量：

```
X = Hello ;
$(X) = Bye ;
```

把“Hello”赋值给 X，然后把“Bye”赋值给一个名为“Hello”的变量。

如果一个变量值列表中不止包含一个元素，那么它的展开结果为一个列表：

```
X =  A B C ;
$(X) = Hi there ;
```

把字符串“A",“B"，和“C”赋值给 X。然后把“Hi”和“there”字符串列表分别赋值给变量 A，B，和 C。

你可以使用一个“脚注”来标识特殊的列表，语法为“$(name[subscript])"：

```
X =  A B C ;
Echo $(X[2]) ;
```

输出为“B"。

你可以使用“+=”操作符为一个列表添加元素：

```
X =  A B C ;
X += $(X) ;
```

现在 X 包含了：A B C A B C。

3.3.2 变量展开积(Variable Expansion Products)

在一个单独的 Jam 标识符中，当变量组合展开或者同一个常字符一起展开时，它们展开为积的形式，并且这个展开积本身也是一个列表，示例如下：

```
X =  A B C ;
Y =  E F ;
Echo $(X)$(Y) ;
```

输出结果为：AE AF BE BF CE CF。

```
X =  A B C ;
Y =  test_$(X).result ;
```

Y 现在包含了一下字符串列表：“test_A.result”，“test_B.result”,和“test_C.result”.

```
X =  A B C D E F G H ;
Selected = 3 7 8 ;
Echo $(X[$(Selected)]) ;
```

输出结果为：C G H。

值得注意的是，一个 Jam“标识符”是一个被空格所限制的声明元素。通常可以使用引用来识别出包含空格的单独的标识符，比较下面的两个示例：

```
X = Bob Sue Pat ;
Echo "Hello $(X)!" ;
```

输出结果为：Hello Bob! Hello Sue! Hello Pat!

```
X = Bob Sue Pat ;
```

```
Echo Hello $(X)! ;
```

输出结果为：Hello Bob! Sue! Pat!

当 Jam 声名中的标识符包含了一个值为空的变量时，该变量展开的结果是个空的列表。（可以认为它和零相乘的结果为零。）一个没有赋值的变量，其值为空。你也可以明确的把一个变量赋值为空。需要注意的是，空的列表和包含一个或多个空字符串的列表是不同的。示例如下：

```
X = Bob Sue Pat ;
Echo Hello $(X)$(Y) ;
```

Y 没有被赋值，所以输出结果只有：Hello。

```
X = Bob Sue Pat ;
Y = "" "" ;
Echo Hello $(X)$(Y) ;
```

Y 的值为两个空字符组成的列表，所以输出结果为：Hello Bob Sue Pat Bob Sue Pat。

```
Y Z = test ;
X = Bob Sue Pat ;
Y = ;
Z = $(X)$(Y) ;
```

由于 Y 没有被赋值，所以(X)(Y)之积为空。结果，Z 的值被清空了。

3.3.3 变量展开修饰符(Variable Expansion Modifiers)

Jam 变量展开的“修饰符”可用于修改结果值。修饰符的语法是“$(变量名:修饰符)"。例如，你可以使用“U”和“L”修饰符来强制对展开结果进行大小写转换：

```
X = This is ;
Y = A TEST ;
Echo $(X:U) $(Y:L) ;
```

输出结果为：THIS IS a test

多数的 Jam 修饰符是特别为处理文件名和目标路径而设计的。它们之中，有些用于裁剪输出结果，有些用于替换输出结果。例如，“S=后缀”修饰符可以替换为文件名后缀：

```
X = foo.c ;
Y = $(X:S=.obj) ;
```

把“foo.c”赋值给 Y。

你可以把修饰符与修饰符，修饰符与脚注列表，修饰符与展开积进行组合使用。下面是一些示例：

```
X = foo.c bar.c ola.c ;
Y = .c .obj .exe .dll ;
Echo $(X[2]:S=$(Y):U) ;
```

输出结果为：BAR.C BAR.OBJ BAR.EXE BAR.DLL

3.3.4 在分析中变量的展开

还记的 Jam 运行过程中的三个步骤么？而你的 Jamfiles 中变量的展开就发生在分析这一步骤中，在它浏览你的文件系统之前和它运行系统命令之前。这就意味着，你不能够为 Jam 变量赋值为任何系统命令输出结果(例如,“ls”或者“find”)！

3.4 规则（Rules）

Jam“规则”是在分析步骤中解释和执行的程序。它们可以和参数一起被调用。每一个参数都是一个列表；而且参数之间要以冒号标识符隔开。例如：

```
Depends a : b c d ;
```

调用具有两个参数的“Depends”内建规则。第一个参数是具有一个元素的列表，“a”。第二个参数是具有三个元素的列表，“b”，“c”，和“d”。

下面是一个规则定义的例子：

```
rule MyRule
{
    Echo First arg is $(1) ;
    Echo Second arg is $(2) ;
    Echo Third arg is $(3) ;
}
```

并且下面是调用的方式：

```
MyRule a : b c : d e f ;
```

输出结果为：

```
First arg is a
Second arg is b c
Third arg is d e f
```

为了实现向后兼容，在规则中允许使用$(<)和$(>)来代替$(1)和$(2)。在老的 Jamfiles 中，你可以使用下面的规则定义方式：

```
rule MyRule
{
    Echo First arg is $(<) ;
    Echo Second arg is $(>) ;
}
```

3.5 操作符(actions)

Jam 中的“操作符”允许在 Jam 的更新步骤中，它是用于解析系统命令的特殊规则。关键字“actions”用于定义操作符。操作符定义中包含了系统命令，而不是 Jam 语言声明。但是，它可以包含 Jam 变量。下面是一个简单的操作符定义的例子：

```
Actions MyAction
{
    tough $(1)
```

```
    cat $(2) >> $(1)
}
```

如果该操作符调用方式如下：

```
MyAction ola : foo bar ;
```

该指令序列将会传递给系统的命令行解释程序 Shell 以更新“ola”：

```
tough ola
cat foo bar >> ola
```

操作符由于具有以下特性而区别于规则：

- 它们只接受两个参数。换句话说，你可以涉及到$(1)和$(2)，但是在操作符中不能够存在$(3)。
- 所有传递给操作符的参数都应该是目标对象。（参看下面内容。）
- 尽管规则在 Jam 的解析步骤中运行，但是操作符在它的更新步骤中运行。而且操作符中的 Jam 变量在其传递给系统的 Shell 之前展开。

3.6 目标对象（targets）和依赖关系（dependencies）

在运行 Jam 时，它假定你希望创建一个或多个目标对象。我曾经说过，Jam 的“目标对象”是一个文件系统对象，例如文件，或者库成员。Jam 也可以识别出“抽象的目标”，而它可以用于组织依赖关系。Jam 提供了一个内建的“all”抽象目标。如果你在 Jam 命令行中没有指定目标对象，Jam 将会尝试构建所有的目标对象（“all”）。最好使用一个例子来介绍它的情况，把下面的命令放到“test”文件中：

```
rule MyRule
{
    TouchFile $(1) ;
}

actions TouchFile
{
    touch $(1)
}
MyRule test.output1 ;
MyRule test.output2 ;
MyRule test.output3 ;
```

接下来运行：

```
jam -ftest
```

输出结果如下：

```
don't know how to make all
...found 1 traget(s)...
...can't find 1 target(s)...
```

为了让 Jam 可以识别出你真正希望构建的目标，你可以在命令行中指定目标对象：

```
jam -ftest test.output2
```

那么这次输出结果如下：

```
...found 1 target(s)...
...updating 1 target(s)...
TouchFile test,output2
...updated 1 target(s)...
```

但是最有效的能够让 Jam 识别出更新内容的方式是使用内建的“Depends”规则把所有目标对象的依赖关系定义为“all"：

```
rule MyRule
{
    TouchFile $(1) ;
    Depends all : $(1) ;
}

actions TouchFile
{
    touch $(1)
}

MyRule test.output1 ;
MyRule test.output2 ;
MyRule test.output3 ;
```

现在你可以构建自己的文件而不需要在命令中指出任何目标：

```
jam -ftest
```

由于一个目标先前的测试中已经完成了构建，所以只有两个需要进行构建。下面是 Jam 的输出结果：

```
...found 4 target(s)...
...updating 2 target(s)...
TouchFile test.output1
TouchFile test.output3
...updated 2 target(s)...
```

当 Jam 正在构建的内容处于依赖关系之外时，它会依据自身情况来指出这一情况。例如，如下更改你的测试文件：

```
rule MyRule
{
    TouchFile $(1) ;
    Depends all : $(1) ;
```

```
}

actions TouchFile
{
    touch $(1)
}

MyRule test.output1 test.output test.output3 ;
```

接下来运行下面的命令来重建里面的一个文件（“-a”用于使 Jam 在该文件存在的情况下也进行重建该文件）：

```
jam -ftest -a test.output2
```

输出结果显示，虽然 Jam 得新增目标不存在于你需要目标的依赖关系中，但是它们仍然要被构建：

```
...found 1 target(s)...
...updating 1 target(s)...
warning: using independent target test.output1
warning: using independent target test.output3
TouchFile test.output1 test.output2 test.output3
...updated 1 target(s)...
```

3.7 隐式操作符调用

如果操作符和规则同名，Jam 将会隐式的调用操作符，而且使用和该规则调用中同样的参数。下面是一个隐式的操作符调用的示例：

```
rule MyRule
{
    Depends all : $(1) ;
}

actions MyRule
{
    p4 info > $(1)
}

MyRule info.output ;
```

在一个单独的声明中调用了“MyRule”，“MyRule”规则将会在解析步骤中运行，而“MyRule”操作符将会在更新步骤中运行。但是两者的参数是相同的，都是“info.output”。

3.8 特殊目标变量（Target-specific Variables）

还有一种为 Jam 变量赋值的语法允许为特定的单独目标赋值。当然，通过这种方式赋予的变

量值只可以在操作符中展开。该语法为“variable on target =”。示例如下：

```
X on foo = A B C ;
X on bar = 1 2 3 ;
```

这是一种很有用的方法：

```
rule MyRule
{
    CMD on $(1) = $(2) ;
    PORT on $(1) = $(3) ;
    Depends all : $(1) ;
    MyTest $(1) ;
}

actions MyTest
{
    p4 -p$(PORT) $(CMD) > $(1)
}

MyRule test1.output : info ;
MyRule test2.output : info : mars:1666 ;
MyRule teset3.output : users : mars:1666 ;
```

运行 Jam，你会得到如下结果：

```
...found 4 target(s)...
...updating 3 target(s)...
MyTest test1.output
    p4 info > test1.output
MyTest test2.output
    p4 -pmars:1666 info > test2.output
MyTest test3.output
    p4 -pmars:1666 users > test3.output
...updated 3 target(s)...
```

在这个例子中，CMD 和 PORT 这两个变量分别被赋予了不同的目标。同一个操作符更新了不同的目标，但是当操作符展开后，对于每个目标，它们的结果是不同的。

4 运行示例：Jam 作为测试程序

为了讲解的方便，我搜集了一系列用于实现一个简单的测试程序的 Jamfiles。为了展示 Jam 不同的能力，我对每一个示例都作了相应的改进。如果你有可用于工作的“Jam”和“p4”，你可以亲自运行这些示例。

但是需要注意的是，这些示例和传统的编译-链接没有任何关系。之所以选择这些例子，是因为它们都比较小，而且可以独立使用。在你熟悉了这些例子，对 Jam 的工作机制有一个完

整的理解之后，你就可以开始学习由 Jambase 提供的编译-链接规则了，而且利用这些规则，你就可以实现大型的编译系统了，但是它们要比本文中的例子复杂的多。

4.1 简单的命令测试程序（Simple Commander Tester）

我们的第一个示例是一个用于测试“p4”命令的非常简单的 Jamfile。它仅仅运行每一个测试，然后把结果输出到一个文件中。对于每个测试，测试的结果文件就是所要更新的目标对象，而用于更新的操作符就是所要测试的“P4”命令。

本示例展示了 Jam 在操作符运行失败之后的行为。当 Jam 把用于更新文件的操作符传递给系统时，它将会检查最终命令的结果。如果结果显示命令运行失败，Jam 将会移除刚刚更新的文件。（这就是 Jam 通常的行为；它并不是在特殊示例中的那样。）这样，结果文件只能由成功的示例测试产生。对于测试程序来说，这样当然很好，因为运行之后，只返回了失败的测试；但是它也是很糟糕的，因为它没有为失败的测试留下任何的痕迹。

该示例也介绍了 Jam 的“本地”声明，它用于限制声明和调用了该变量的规则中变量的范围。

```
rule Test
{
    local f = $(1:S=.out) ;
    Depends all : $(f) ;
    RunTest $(f) ;
    CMD on $(f) = $(1) ;
}
actions RunTest
{
    p4 $(CMD) > $(1)
}
Test info ;
Test users ;
Test clients ;
```

4.2 捕获失败的命令（Capturing Failed Commands）

如果一个测试程序没有显示任何失败的测试结果，那么它就不是很有用。在本示例中，对前面的例子做了修改以捕获“p4”测试命令的错误消息。该示例中引入了“ignored”操作修饰符。（可以查看 Jam 文档以获取其他操作修饰符的内容。）“ignored”在操作符运行失败时改变 Jam 的运行情况：Jam 将不会移除目标文件而是继续编译其他依赖于它的目标对象。

无论测试是否成功，每个测试都会有个结果文件。然而除了查看结果文件之外，没有其他的方法可以检查是否测试成功。并且，如果结果文件存在，Jam 认为测试已经结束；重新运行 Jam 将不会重运行失败的测试。

```
rule Test
{
    local f = $(1:S=.out) ;
```

```
    Depends all : $(f) ;
    RunTest $(f) ;
    CMD on $(f) = $(1) ;
}
actions ignore RunTest
{
    p4 $(CMD) > $(1) 2>&1
}
Test info ;
Test clients ;
Test users ;
```

4.3 与标准结果相比较(Comparing Canonical Results)

一个更有用的测试程序需要把测试结果和先前储存的“标准”结果进行比较。本节中的示例程序做了一些增强以运行每个测试，比较测试输出与标准结果，然后把结果输出到“匹配”文件。在测试之后，通过匹配文件来显示成功的测试，而“jam -nd1”显示失败的测试。

需要注意的是本示例中的依赖关系。“All”依赖于匹配文件，每个匹配文件依赖于它所反馈的测试结果文件，而测试结果文件则依赖于它的标准文件。通过这些依赖关系，只有和标准相比较的测试才可以运行，而且其中只有先前没有创建匹配文件的才可以运行。

同时，仔细查看一下传送给操作符的所有目标对象。匹配文件是“RunTest”和“DiffResults”共同的目标对象，而“RunTest”创建了测试结果文件，“DiffResults”对结果文件和标准结果进行了对比。如果匹配文件丢失，Jam 将会调用这两个操作符对其予以重新创建。然而，匹配文件对“RunTest”操作符完全没有影响，“RunTest”只用来创建结果文件。如果匹配文件丢失时，这将会诱使 Jam 返回测试结果。换句话说，“DiffResults”操作符只以匹配文件作为目标对象。如果比较失败，只有匹配文件被删除；而结果文件将会留待测试程序检查。

```
rule Test
{
    local f = $(1:S=.out) ;
    CMD on $(f) = $(1) ;

    local canon = $(1:S=.canon) ;
    local match = $(1:S=.match) ;

    Depends all : $(match) ;
    Depends $(match) : $(f) ;
    Depends $(f) : $(canon) ;

    RunTest $(f) $(match) ;
    DiffResults $(match) : $(f) $(canon) ;
}
```

```
actions ignore RunTest
{
    p4 $(CMD) > $(1) 2>&1
}

actions DiffResults
{
    diff $(2) > $(1) 2>&1
}
Test info ;
Test clients ;
Test users ;
```

4.4 捕获标准结果(capturing canonical results)

在这个示例中，测试程序再次做了增强：它可以对标准文件进行创建或者更新。而正是“CAPTURE”变量的引入启动了这一附加行为。由于 Jamfile 中没有设置为 CAPTURE，它必须在运行 Jam 之前进行设置。你可以在运行环境中设置 CAPTURE，也可以在 Jam 命令行中进行设置：

```
jam -sCAPTURE=1 ...
```

在 CAPTURE 设置后，Jam 将会遵循完全不同的逻辑和不同的依赖关系。“All”现在依赖于标准结果文件，如果任何标准结果丢失，它们将会再次从相应的结果文件中拷贝过来。丢失的结果文件可以通过再次运行测试予以创建。

（写一段把结果文件拷贝到标准结果文件脚本显然是非常复杂的选择，但是它主要解决了两个问题：给 Jam 变量赋值，和条件逻辑。）

```
rule Test
{
    if $(CAPTURE)
    {
        CaptureCanon $(1) ;
    }
    else
    {
        DoTest $(1) ;
    }
}
rule CaptureCanon
{
    local canon = $(1:S=.canon) ;
    local result = $(1:S=.out) ;
    CMD on $(result) = $(1) ;
```

```
    Depends all : $(canon) ;
    Depends $(canon) : $(result) ;

    RunTest $(result) ;
    CopyResult $(canon) : $(result) ;
}

actions CopyResult
{
    cp $(>) $(<)
}

rule DoTest
{
    local f = $(1:S=.out) ;
    CMD on $(f) = $(1) ;

    local canon = $(1:S=.canon) ;
    local match = $(1:S=.match) ;

    Depends all : $(match) ;
    Depends $(match) : $(f) ;
    Depends $(f) : $(canon) ;

    RunTest $(f) $(match) ;
    DiffResults $(match) : $(f) $(canon) ;
}

actions ignore RunTest
{
    p4 $(CMD) > $(1) 2>&1
}

actions DiffResults
{
    diff $(2) > $(1) 2>&1
}
Test info ;
Test clients ;
Test users ;
```

4.5 移除旧的结果文件和标准文件（removing old results and canons）

现在我为本示例做了增强来演示“Clean”规则。该规则用于把以构建的目标对象从文件系

统中移除。我为 clean 规则添加了两个调用，一个用于清除结果和 match 文件，一个用于清除 canon 文件。需要注意的是抽象目标，“clean"。没有什么文件依赖于“clean"，或者被“clean”依赖，这样唯一可以激活 Clean 操作符的方法是在 Jam 命令行中使用“clean”作为目标。换句话说，就是运行：

```
jam clean
```

以移除结果文件或者 match 文件，或者：

```
Jam -sCAPTURE=1 clean
```

以移除 canon 文件。

Clean 操作符有许多行为修饰符：

- ignore（你可能看到过它）。在本节中它基本没有什么用途，除了在文件权限阻止了它们的移除时，用于抑制“failed to build”消息。
- together 告诉 Jam 该操作符仅需要运行一次，而且在操作符定义中的$(1)可以在所有调用了它的目标中展开。（如果没有 together，操作符在每个调用中仅仅可以运行一次。结果是相同的，但是对于一定的操作符，例如一个编译器，together 可以使构建更加有效。）
- existing 告诉 Jam，当在操作符中展开$(1)时，不要包含在文件系统中不存在的目标对象。
- piecemeal 告诉 Jam，如果展开的操作符相对于系统命令 Shell 过大，则把目标进行分组，然后分别运行操作符。

```
rule Test
{
    if $(CAPTURE)
    {
        CaptureCanon $(1) ;
    }
    else
    {
        DoTest $(1) ;
    }
}

rule CaptureCanon
{
    local canon = $(1:S=.canon) ;
    local result = $(1:S=.out) ;
    CMD on $(result) = $(1) ;

    Depends all : $(canon) ;
    Depends $(canon) : $(result) ;

    RunTest $(result) ;
```

```
    CopyResult $(canon) : $(result) ;
    Clean clean : $(canon) ;
}

actions CopyResult
{
    cp $(>) $(<)
}

rule DoTest
{
    local f = $(1:S=.out) ;
    CMD on $(f) = $(1) ;

    local canon = $(1:S=.canon) ;
    local match = $(1:S=.match) ;

    Depends all : $(match) ;
    Depends $(match) : $(f) ;
    Depends $(f) : $(canon) ;

    RunTest $(f) $(match) ;
    DiffResults $(match) : $(f) $(canon) ;
    Clean clean : $(f) $(match) ;
}

actions piecemeal together existing Clean
{
    rm $(2)
}

actions ignore RunTest
{
    p4 $(CMD) > $(1) 2>&1
}

actions DiffResults
{
    diff $(2) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

4.6 编写可移植的操作符（Writing Portable Actions）

本节的例子用于讲解最后的一些改进，这些改进用于阐述：如何让相同的操作符调用可以在不同的系统中运行。在定义操作符之前，我为特别的系统命令设置了一些变量。在操作符定义中我使用了这些变量来代替硬编码命令。在一些情况下，用以执行等价操作符的命令在不同系统下会有很大不同，仅仅更改命令名将不会产生任何的作用。在这种情况下，你可以为根据系统来定义操作符，如下所示：

```
if $(NT)
{
    REMOVE = del/f/q ;
    COPY = copy ;

    actions DiffResults { echo n|comp $(2) > $(1) 2>&1 }
}

if $(UNIX)
{
    REMOVE = rm ;
    COPY = cp ;

    actions DiffResults { diff $(2) > $(1) 2>&1 }
}

rule Test
{
    if $(CAPTURE)
    {
        CaptureCanon $(1) ;
    }
    else
    {
        DoTest $(1) ;
    }
}

rule CaptureCanon
{
    local canon = $(1:S=.canon) ;
    local result = $(1:S=.out) ;
    CMD on $(result) = $(1) ;

    Depends all : $(canon) ;
    Depends $(canon) : $(result) ;
```

```
    RunTest $(result) ;
    CopyResult $(canon) : $(result) ;
    Clean clean : $(canon) ;
}

actions CopyResult
{
    $(COPY) $(>) $(<)
}

rule DoTest
{
    local f = $(1:S=.out) ;
    CMD on $(f) = $(1) ;

    local canon = $(1:S=.canon) ;
    local match = $(1:S=.match) ;

    Depends all : $(match) ;
    Depends $(match) : $(f) ;
    Depends $(f) : $(canon) ;

    RunTest $(f) $(match) ;
    DiffResults $(match) : $(f) $(canon) ;
    Clean clean : $(f) $(match) ;
}

actions piecemeal together existing Clean
{
    $(REMOVE) $(2)
}

actions ignore RunTest
{
    p4 $(CMD) > $(1) 2>&1
}

Test info ;
Test clients ;
Test users ;
```

5. 总结

尽管我还没有完全展示出 Jam 的功能，但是现在你应该对 Jam 有了一个足够清晰的理解了，你可以通过阅读 Jam 文档来补充那些没有提到的内容。我建议您阅读所有在 Jam 程序中提供的文档：

- *Jam/MR - Make(1) Redux* - 描述了 Jam 语言和可执行的命令标识符。
- *Using Jamfiles and Jambase* - 概述了如何利用 Jam 程序已经创建的规则。(这些规则可用于实现一个集编译，归档，链接等于一体的对象构建系统。而且，这些规则还提供了一些在层次目录中管理源文件和生成文件的方法。)
- *Jambase Reference* - 对这些规则的简明介绍。

我还建议大家去阅读 Jambase 本身的源文件。尽管有点长，但是它阐明了许多你可以在自己的 Jamfiles 中使用的技巧。

Note:本文参照 Getting Started with Jam 进行翻译，限于译者水平，可能在翻译中会存在翻译不妥当或者错误的地方，还希望大家多多包涵，提出指正。Jam 是 Haiku（BeOS 系统的开源替代系统）系统的构建软件，用于执行对 Haiku 系统的编译，链接等操作。本文是为了方便中文的 Haiku 爱好者能够很好的理解 Haiku 的源代码，参与到 Haiku 的开发和推广之中而进行翻译的。（译者：Pengphei Han）

【1】Jam 是开放源代码的软件，您可以通过下面的链接拿到它的源码，并且获得相关的文档：http://public.perforce.com/wiki/Jam 。

【2】Haiku 是开放源代码的操作系统，有许多优秀的特性，您可以在下面的 Haiku 官方网站上找到更加详细的您感兴趣的内容：http://www.haiku-os.org 。

【3】本文是 Haiku 中文翻译小组的成果，如果您对 Haiku 非常感兴趣，希望为 Haiku 在中国的推广贡献自己的力量，并且你有很好的计算机方面的能力，英语水平也可以的话，你可以参与到 Haiku 中文化中来。Haiku 中文项目的 wiki 在：http://dev.haiku-os.org/wiki/i18n/zh_CN/Info 。