SCons 用户指南

andyelvis

http://blog.csdn.net/andyelvis/article/details/7055377

目录

第一章：编译和安装 SCons

1、安装 Python

因为 SCons 是用 Python 编写的，所以你必须在使用 SCons 之前安装好 Python。你在安装 Python 之前，应该注意查看 Python 是否在你的系统里已经可用了（在系统的命令行中运行 python -V 或 python --version）。

$python -V

Python 2.5.1

在一个 Windows 系统里，

C:\>python -V

Python 2.5.1

如果 Python 没有安装，你会看到一条错误消息比如“command not found”(在 UNIX 或 Linux 里)或"python is not recognized as an internal or external command, operable program or batch file"(在 Windows 里)。在这种情况下，在你安装 SCons 之前需要先安装 Python。

有关下载和安装 Python 的信息可以从 http://www.python.org/download/得到。

2、从预编译包中安装 SCons

2.1、在 Red Hat(或者基于 RPM)Linux 系统里安装 SCons

在使用 RPM(Red Hat Package Manager)的 Red Hat Linux，Fedora 或者任何其他 Linux 发行版里，SCons 是预编译好的 RPM 格式，准备被安装的。你的发行版可能已经包含了一个预编译好的 SCons RPM。

如果你的发行版支持 yum 安装，你可以运行如下命令安装 SCons：

#yum install scons

如果你的 Linux 发行版没有包含一个特定的 SCons RPM 文件，你可以下载 SCons 项目提供的通用的 RPM 来安装。这会安装 SCons 脚本到/usr/bin 目录，安装 SCons 库模块(library modules)到/usr/lib/scons。

从命令行安装，下载合适的.rpm 文件，然后运行：

#rpm -Uvh scons-2.1.0-1.noarch.rpm

2.2、在 Debian Linux 系统里安装 SCons

如果你的系统已经连上了因特网，你可以运行如下命令来安装最新的官方 Debian 包：
#apt-get install scons

2.3、在 Windows 系统里安装 SCons

SCons 提供了一个 Windows installer，使得安装变得非常容易。从 http://www.scons.org/download.php 下载 scons-2.1.0.win32.exe。然后你需要做的就是执行这个文件。

3、在任何系统里编译和安装 SCons

如果你的系统里没有一个预编译的 SCons 包，你可以使用本地 python distutils 包很容易地编译和安装 SCons。
第一步就是下载 scons-2.1.0.tar.gz 或 scons-2.1.0.zip，地址 http://www.scons.org/download.html。
解压下载的文件，会创建一个叫 scons-2.1.0 的目录，进入这个目录执行如下命令安装 SCons：
#cd scons-2.1.0
#python setup.py install
这将会编译 SCons，安装 scons 脚本到 python 目录（/usr/local/bin 或 C:\Python25\Scripts），同时会安装 SCons 编译引擎到 python 使用的库目录（/usr/local/lib/scons 或 C:\Python25\scons）。因为这些都是系统目录，你可能需要 root 或管理员权限去安装 SCons。

3.1、编译和安装多个版本的 SCons

SCons 的 setup.py 脚本有一些扩展，这些扩展支持安装多个版本的 SCons 到不同的位置。这让下载和体验不同版本的 SCons 变得很容易。
安装 SCons 到指定版本的位置，调用 setup.py 的时候增加--version-lib 选项：
#python setup.py install --version-lib
这将会安装 SCons 编译引擎到/usr/lib/scons-2.1.0 或 C:\Python25\scons-2.1.0 目录。

3.2、安装 SCons 到其他的位置

你可以安装 SCons 到其他的位置，而不是默认的位置，指定--prefix=选项：

#python setup.py install --prefix=/opt/scons
这将会安装 scons 脚本到/opt/scons/bin，安装编译引擎到/opt/scons/lib/scons。
你可以同时指定--prefix 和--version-lib，这个时候 setup.py 将会安装编译引擎到相对于指定 prefix 的特定版本的目录，在刚才的例子上加上--version-lib，将会安装编译引擎到 /opt/scons/lib/scons-2.1.0。

3.3、没有管理员权限的情况下编译和安装 SCons

如果你没有权限安装 SCons 到系统目录，使用--prefix 选项安装到你选择的其他的位置。例如，安装 SCons 到相对于用户$HOME 目录的合适的位置，scons 脚本安装到$HOME/bin，编译引擎安装到$HOME/lib/scons，使用如下命令：
#python setup.py install --prefix=$HOME

第二章：简单编译

1、编译简单的 C/C++程序

这是一个用 C 语言编写的著名的"Hello,World!"程序：

```
int main()
{
    printf("Hello, World!\n");
}
```

用 SCons 编译它，需要在一个名为 SConstruct 的文件中输入如下命令：

```
Program('hello.c')
```

这个短小的配置文件给了 SCons 两条信息：你想编译什么（一个可执行程序），你编译的输入文件（hello.c）。Program 是一个编译器方法（builder_method），一个 Python 调用告诉 SCons，你想编译一个可执行程序。
现在运行 scons 命令编译这个程序。在 Linux 或 Unix 系统上，你会看到如下输出：

```
% scons
scons: Reading SConscript files...
scons: done reading SConscript files.
scons: Building targets...
cc -o hello.o -c hello.c
```

```
cc -o hello hello.o
scons: done building targets.
```

在一个带有微软 Visual C++编译器的 Windows 系统上，你会看到如下输出：

```
C:\>scons
scons: Reading SConscript files...
scons: done reading SConscript files.
scons: Building targets...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target,source,env)
scons: done building targets.
```

首先，你仅仅需要指定源文件，SCons 会正确地推断出目标文件和可执行文件的名字。
其次，同样的 SConstruct 文件，在 Linux 和 Windows 上都产生了正确的输出文件：在 POSIX 系统上是 hello.o 和 hello，在 Windows 系统上是 hello.obj 和 hello.exe。这是一个简单的例子，说明了 SCons 使得编写程序编译脚本变得很容易了。

2、编译目标程序

Program 编译方法是 SCons 提供的许多编译方法中一个。另一个是 Object 编译方法，告诉 SCons 从指定的源文件编译出一个目标文件：

```
Object('hello.c')
```

现在运行 scons 命令编译，在 POSIX 系统里它仅仅编译出 hello.o 目标文件：

```
% scons
scons: Reading SConscript files...
scons: done reading SConscript files.
scons: Building targets...
cc -o hello.o -c hello.c
scons: done building targets.
```

在 Windows 系统里编译出 hello.obj 目标文件：

```
C:\>scons
scons: Reading SConscript files...
scons: done reading SConscript files.
scons: Building targets...
cl /Fohello.obj /c hello.c /nologo
scons: done building targets.
```

3、简单的 JAVA 编译

SCons 同样使得编译 Java 也很容易了。不像 Program 和 Object 两个编译方法，Java 编译方法需要你指定一个目录，这个目录是用来存放编译后的 class 文件的，以及一个存放.java 源文件的目录：

```
Java('classes', 'src')
```

如果 src 目录仅仅包含一个 hello.java 文件，那么运行 scons 命令的输出会如下所示（在 POSIX 系统里）：

```
% scons
scons: Reading SConscript files...
scons: done reading SConscript files.
scons: Building targets...
javac -d classes -sourcepath src src/hello.java
scons: done building targets.
```

4、编译之后清除

使用 SCons，编译之后想要清除不需要增加特殊的命令或目标名。你调用 SCons 的时候，使用-c 或--clean 选项，SCons 就会删除合适的编译产生的文件。

```
% scons -c
```

5、SConstruct 文件

如果你使用过 Make 编译系统，你应该可以推断出 SConstruct 文件就相当于 Make 系统中的 Makefile。SCons 读取 SConstruct 文件来控制程序的编译。

5.1、SConstruct 文件是 Python 脚本

SConstruct 文件实际上就是一个 Python 脚本。你可以在你的 SConstruct 文件中使用 Python 的注释：

```
# Arrange to build the "hello" program.
Program('hello.c')                #"hello.c" is the source file.
```

5.2、SCons 的函数是顺序无关的

重要的一点是 SConstruct 文件并不完全像一个正常的 Python 脚本那样工作，其工作方式更像一个 Makefile，那就是在 SConstruct 文件中 SCons 函数被调用的顺序并不影响 SCons 你实际想编译程序和目标文件的顺序。换句话说，当你调用 Program 方法，你并不是告诉 SCons 在调用这个方法的同时马上就编译这个程序，而是告诉 SCons 你想编译这个程序，例如，一个程序由一个 hello.c 文件编译而来，这是由 SCons 决定在必要的时候编译这个程序的。

SCons 通过打印输出状态消息来显示它何时在读取 SConstruct 文件，何时在实际编译目标文件，进而来区分是在调用一个类似 Program 的编译方法还是在实际地编译这个程序。

看下面这个例子：

```
print "Calling Program('hello.c')"
Program('hello.c')
print "Calling Program('goodbye.c')"
Program('goodbye.c')
print "Finished calling Program()"
```

执行 SCons，我们看到 print 语句的输出是在读取 Sconstruct 文件消息之间，说明了那才是 Python 语句执行的时候：

```
% scons
scons: Reading Sconscript files...
Calling Program('hello.c')
Calling Program('goodbye.c')
Finished Calling Program()
scons: done reading SConscript files...
scons: Building targets...
cc -o goodbye.o -c goodbye.c
cc -o goodbye goodbye.o
```

```
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

6、使 Scons 输出更简洁

你已经看到过 SCons 编译的时候会打印一些消息，那些消息围绕着实际用来编译程序的命令：

```
C:\scons
scons: Reading SConscript files...
scons: done reading SConscript files.
scons: Building targets...
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
scons: done building targets.
```

这些消息反映了 SCons 工作时候的顺序。

一个缺点就是，这些消息使得输出看起来很混乱。当调用 SCons 的时候使用-Q 选项，可以屏蔽掉那些与实际编译程序命令无关的消息：

```
C:\>scons -Q
cl /Fohello.obj /c hello.c /nologo
link /nologo /OUT:hello.exe hello.obj
embedManifestExeCheck(target, source, env)
```

第三章：编译相关的一些事情

1、指定目标文件的名字

当你调用 Program 编译方法的的时候，它编译出来的程序名字是和源文件名是一样的。下面的从 hello.c 源文件编译一个可执行程序的调用

将会在 POSIX 系统里编译出一个名为 hello 的可执行程序，在 windows 系统里会编译出一个名为 hello.exe 的可执行程序。

```
Program('hello.c')
```

如果你想编译出来的程序的名字与源文件名字不一样，你只需要在源文件名的左边声明一个目标文件的名字就可以了：

```
Program('new_hello','hello.c')
```

现在在 POSIX 系统里运行 scons，将会编译出一个名为 new_hello 的可执行程序：

```
% scons -Q
cc -o hello.o -c hello.c
cc -o new_hello hello.o
```

2、编译多个源文件

通常情况下，你需要使用多个输入源文件编译一个程序。在 SCons 里，只需要就多个源文件放到一个 Python 列表中就行了，如下所示：

```
Program(['prog.c','file1.c','file2.c'])
```

运行 scons 编译：

```
% scons -Q
cc -o file1.o -c file1.c
cc -o file2.o -c file2.c
cc -o prog.o -c prog.c
cc -o prog prog.o file1.o file2.o
```

注意到 SCons 根据源文件列表中的第一个源文件来决定输出程序的名字。如果你想指定一个不同的程序名称，你可以在源文件列表的右边指定程序名，如下所示指定输出程序名为 program：

```
Program('program',['prog.c','file1.c','file2.c'])
```

3、使用 Glob 指定文件列表

你可以使用 Glob 函数，定义一个匹配规则来指定源文件列表，比如*,?以及[abc]等标准的 shell 模式。如下所示：

```
Program('program', Glob('*.c'))
```

4、指定单个文件以及文件列表

有两种方式为一个程序指定源文件，一个是文件列表：

```
Program('hello', ['file1.c', 'file2.c'])
```

一个是单个文件：

```
Program('hello', 'hello.c')
```

也可以把单个文件放到一个列表中，

```
Program('hello', ['hello.c'])
```

对于单个文件，SCons 函数支持两种方式。实际上，在内部，SCons 把所有的输入都是看成列表的，只是在单个文件的时候，允许我们省略方括号。

5、使文件列表更易读

为了更容易处理文件名长列表，SCons 提供了一个 Split 函数，这个 Split 函数可以将一个用引号引起来，并且以空格或其他空白字符分隔开的字符串分割成一个文件名列表，示例如下：

```
Program('program', Split('main.c   file1.c   file2.c'))
```

或者

```
src_files=Split('main.c   file1.c   file2.c')
Program('program', src_files)
```

同时，Split 允许我们创建一个文件列表跨跃多行，示例如下：

```
src_files=Split("""main.c
                   file1c
                   file2.c""")
Program('program', src_files)
```

6、关键字参数

SCons 允许使用 Python 关键字参数来标识输出文件和输入文件。输出文件是 target，输入文件是 source，示例如下：

```
src_files=Split('main.c   file1.c   file2.c')
Program(target='program', source=src_files)
```

或者

```
src_files=Split('main.c   file1.c   file2.c')
Program(source=src_files, target='program')
```

7、编译多个程序

如果需要用同一个 SConstruct 文件编译多个文件，只需要调用 Program 方法多次：

```
Program('foo.c')
Program('bar', ['bar1.c', 'bar2.c'])
```

8、在多个程序之间共享源文件

在多个程序之间共享源文件是很常见的代码重用方法。一种方式就是利用公共的源文件创建一个库文件，然后其他的程序可以链接这个库文件。另一个更直接，但是不够便利的方式就是在每个程序的源文件列表中包含公共的文件，示例如下：

```
Program(Split('foo.c   common1.c   common2.c'))
Program('bar', Split('bar1.c   bar2.c   common1.c   common2.c'))
```

如果程序之间共享的源文件过多，可以简化：

```
common=['common1.c', 'common2.c']
foo_files=['foo.c'] + common
bar_files=['bar1.c', 'bar2.c'] + common
Program('foo', foo_files)
Program('bar', bar_files)
```

第四章：编译和链接库文件

1、编译库文件

你可以使用 Library 方法来编译库文件：

```
Library('foo', ['f1.c', 'f2.c', 'f3.c'])
```

SCons 会根据你的系统使用合适的库前缀和后缀。所以在 POSIX 系统里，上面的例子会如下编译：

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
```

如果你不显示指定目标库的名字，SCons 会使用第一个源文件的名字。

1.1、使用源代码或目标文件编译库文件

除了使用源文件外，Library 也可以使用目标文件，如下所示：

```
Library('foo', ['f1.c', 'f2.o', 'f3.c', 'f4.o'])
```

1.2、使用 StaticLibrary 显示编译静态库

Library 函数是用来编译静态库的。如果你想显示指定需要编译静态库，可以使用 StaticLibrary 替代 Library：

```
StaticLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

1.3、使用 SharedLibrary 编译动态库

如果想编译动态库（在 POSIX 系统里）或 DLL 文件（Windows 系统），可以使用 SharedLibrary：

```
SharedLibrary('foo', ['f1.c', 'f2.c', 'f3.c'])
```

在 POSIX 里运行 scons 编译：

```
% scons -Q
cc -o f1.os -c f1.c
cc -o f2.os -c f2.c
cc -o f3.os -c f3.c
cc -o libfoo.so -shared f1.os f2.os f3.os
```

2、链接库文件

链接库文件的时候，使用$LIBS 变量指定库文件，使用$LIBPATH 指定存放库文件的目录：

```
Library('foo', ['f1.c', 'f2.c', 'f3.c'])
Program('prog.c', LIBS=['foo', 'bar'], LIBPATH='.')
```

注意到，你不需要指定库文件的前缀（比如 lib）或后缀（比如.a 或.lib），SCons 会自动匹配。

```
% scons -Q
cc -o f1.os -c f1.c
cc -o f2.os -c f2.c
cc -o f3.os -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog.o -c prog.c
cc -o prog prog.o -L. -lfoo -lbar
```

3、$LIBPATH 告诉去哪里找库

默认情况下，链接器只会在系统默认的库目录中寻找库文件。SCons 也会去 $LIBPATH 指定的目录中去寻找库文件。$LIBPATH 由一个目录列表组成，如下所示：

```
Program('prog.c', LIBS='m', LIBPATH=['/usr/lib', '/usr/local/lib'])
```

使用 Python 列表的好处是可以跨平台。另一种可选方式是，把库目录用系统特定的路径分隔符连接成一个字符串：
在 POSIX 系统里：
LIBPATH='/usr/lib:/usr/local/lib'
在 Windows 里：
LIBPATH='C:\\lib;D:\\lib'
当链接器执行的时候，SCons 会创建合适的 flags，使得链接器到指定的库目录寻找库文件。上面的例子在 POSIX 系统里编译：

```
% scons -Q
cc -o prog.o -c prog.c
cc -o prog prog.o -L/usr/lib -L/usr/local/lib -lm
```

第五章：节点对象

1、编译方法返回目标节点列表

所有编译方法会返回一个节点对象列表，这些节点对象标识了那些将要被编译的目标文件。这些返回出来的节点可以作为参数传递给其他的编译方法。
例如，假设我们想编译两个目标文件，这两个目标有不同的编译选项，并且最终组成一个完整的程序。这意味着对每一个目标文件调用 Object 编译方法，如下所示：

```
Object('hello.c', CCFLAGS='-DHELLO')
Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(['hello.o', 'goodbye.o'])
```

这样指定字符串名字的问题就是我们的 SConstruct 文件不再是跨平台的了。因为在 Windows 里，目标文件成为了 hello.obj 和 goodbye.obj。
一个更好的解决方案就是将 Object 编译方法返回的目标列表赋值给变量，这些变量然后传递给 Program 编译方法：

```
hello_list = Object('hello.c', CCFLAGS='-DHELLO')
goodbye_list = Object('goodbye.c', CCFLAGS='-DGOODBYE')
Program(hello_list + goodbye_list)
```

这样就使得 SConstruct 文件是跨平台的了。

2、显示创建文件和目录节点

在 SCons 里，表示文件的节点和表示目录的节点是有清晰区分的。SCons 的 File 和 Dir 函数分别返回一个文件和目录节点：

```
hello_c=File('hello.c')
Program(hello_c)
classes=Dir('classes')
Java(classes, 'src')
```

通常情况下，你不需要直接调用 File 或 Dir，因为调用一个编译方法的时候，SCons 会自动将字符串作为文件或目录的名字，以及将它们转换为节点对象。只有当你需要显示构造节点类型传递给编译方法或其他函数的时候，你才需要手动调用 File 和 Dir 函数。

有时候，你需要引用文件系统中一个条目，同时你又不知道它是一个文件或一个目录，你可以调用 Entry 函数，它返回一个节点可以表示一个文件或一个目录：

```
xyzzy=Entry('xyzzy')
```

3、打印节点文件名

你可能需要经常做的就是使用一个节点来打印输出这个节点表示的文件名。因为一个编译方法调用返回的对象是一个节点列表，你必须使用 Python 脚本从列表中获得单个节点。例如，如下的 SConstruct 文件：

```
hello_c=File('hello.c')
Program(hello_c)
classes=Dir('classes')
Java(classes, 'src')
object_list=Object('hello.c')
program_list=Program(object_list)
print "The object file is:", object_list[0]
print "The program file is:", program_list[0]
```

4、将一个节点的文件名当作一个字符串

如果你不是想打印文件名，而是做一些其他的事情，你可以使用内置的 Python 的 str 函数。例如，你想使用 Python 的 os.path.exists 判断一个文件是否存在：

```
import os.path
```

```
program_list=Program('hello.c')
program_name=str(program_list[0])
if not os.path.exists(program_name):
    print program_name, "does not exist!"
```

在 POSIX 系统里执行 scons：

```
% scons -Q
hello does not exist!
cc -o hello.o -c hello.c
cc -o hello hello.o
```

5、GetBuildPath：从一个节点或字符串中获得路径

env.GetBuildPath(file_or_list)返回一个节点或一个字符串表示的路径。它也可以接受一个节点或字符串列表，返回路径列表。如果传递单个节点，结果就和调用 str(node)一样。路径可以是文件或目录，不需要一定存在：

```
env=Environment(VAR="value")
n=File("foo.c")
print env.GetBuildPath([n, "sub/dir/$VAR"])
```

将会打印输出如下：

```
% scons -Q
['foo.c', 'sub/dir/value']
scons: . is up to date.
```

有一个函数版本的 GetBuildPath，不需要被一个 Environment 调用，它是基于 SCons 默认的 Environment 来使用的。

第六章：依赖性

到目录为止，我们已经看到了 SCons 是如何一次性编译的。但是 SCons 这样的编译工具的一个主要的功能就是当源文件改变的时候，只需要重新编译那些修改的文件，而不会浪费时间去重新编译那些不需要重新编译的东西。如下所示：

```
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
scons: '.' is up to date.
```

第二次执行的时候，SCons 根据当前的 hello.c 源文件判断出 hello 程序是最新的，避免了重新编译。

1、决定一个输入文件何时发生了改变：Decider 函数

默认情况下，SCons 通过每个文件内容的 MD5 签名，或者校验和来判断文件是否是最新的，当然你也可以配置 SCons 使用文件的修改时间来判断。你甚至可以指定你自己的 Python 函数来决定一个输入文件是否发生了改变。

1.1、使用 MD5 签名来决定一个文件是否改变

默认情况下，SCons 根据文件内容的 MD5 校验和而不是文件的修改时间来决定文件是否改变。如果你想更新文件的修改时间，来使得 SCons 重新编译，那么会失望的。如下所示：

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% touch hello.c
% scons -Q hello
scons: `hello' is up to date.
```

上面的例子中即使文件的修改时间变了，SCons 认为文件的内容没有改变，所以不需要重新编译。但是如果文件的内容改变了，SCons 会侦测到并且重新编译的：

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% edit hello.c
    [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
```

你也可以显示指定使用 MD5 签名，使用 Decider 函数：

```
Program('hello.c')
Decider('MD5')
```

1.1.1、使用 MD5 签名的衍生

使用 Md5 签名去决定一个输入文件是否改变，有一个好处：如果一个源文件已经改变了，但是由它重新编译出来的目标文件的内容和由它修改前编译出来的目标文件一样，那么那些依赖这个重新编译的但是内容没变的目标文件的其他目标文件是不需要重新编译的。

例如，一个用户仅仅改变了 hello.c 文件中的注释，那么重新编译出来的 hello.o 文件肯定是不变的。SCons 将不会重新编译 hello 程序：

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% edit hello.c
    [CHANGE A COMMENT IN hello.c]
% scons -Q hello
cc -o hello.o -c hello.c
scons: `hello' is up to date.
```

1.2、使用时间戳（Time Stamps）来决定一个文件是否改变

SCons 允许使用两种方式使用时间戳来决定一个输入文件是否已经改变。

最熟悉的方式就是 Make 使用时间戳的方式：如果一个源文件的修改时间比目标文件新，SCons 认为这个目标文件应该重新编译。调用 Decider 函数如下：

```
Object('hello.c')
Decider('timestamp-newer')
```

并且因为这个行为和 Make 的一样，你调用 Decider 函数的时候可以用 make 替代 timestamp-newer：

```
Object('hello.c')
Decider('make')
```

使用和 Make 一样时间戳的一个缺点就是如果一个输入文件的修改时间突然变得比一个目标文件旧，这个目标文件将不会被重新编译。例如，如果一个源文件的一个旧的拷贝从一个备份中恢复出来，恢复出来的文件的内容可能不同，但是目标文件将不会重新编译因为恢复出来的源文件的修改时间不比目标文件文件新。

因为 SCons 实际上存储了源文件的时间戳信息，它可以处理这种情况，通过检查源文件时间戳的精确匹配，而不是仅仅判断源文件是否比目标文件新。示例如下：

```
Object('hello.c')
Decider('timestamp-match')
```

1.3、同时使用 MD5 签名和时间戳来判断一个文件是否改变

SCons 提供了一种方式，使用文件内容的 MD5 校验和，但是仅仅当文件的时间戳改变的时候去读文件的内容：

```
Program('hello.c')
Decider('MD5-timestamp')
```

使用 Decider('MD5-timestamp')的唯一缺点就是 SCons 将不会重新编译一个目标文件，如果 SCons 编译这个文件后的一秒以内源文件被修改了。

1.4、编写你自己的 Decider 函数

我们传递给 Decider 函数的不同的字符串实际上是告诉 SCons 去选择内部已经实现的决定文件是否改变的函数。我们也可以提供自己的函数来决定一个依赖是否已经改变。

例如，假设我们有一个输入文件，其包含了很多数据，有特定的格式，这个文件被用来重新编译许多不同的目标文件，但是每个目标文件仅仅依赖这个输入文件的一个特定的区域。我们希望每个目标文件仅仅依赖自己在输入文件中的区域。但是，因为这个输入文件可能包含了很多数据，我们想仅仅在时间戳改变的时候才打开这个文件。这个可以通过自定义的 Decider 函数实现：

```
Program('hello.c')
def decide_if_changed(dependency,target,prev_ni):
    if self.get_timestamp()!=prev_ni.timestamp:
        dep=str(dependency)
        tgt=str(target)
        if specific_part_of_file_has_changed(dep,tgt):
            return True
    return False
Decider(decide_if_changed)
```

在函数定义中，depandency（输入文件）是第一个参数，然后是 target。它们都是作为 SCons 节点对象传递给函数的，所以我们需要使用 str()转换成字符串。第三个参数，prev_ni，是一个对象，这个对象记录了目标文件上次编译时所依赖的签名和时间戳信息。prev_ni 对象可以记录不同的信息，取决于 dependency 参数所表示的东西的类型。对于普通的文件，prev_ni 对象有以下的属性:
.csig：target 上次编译时依赖的 dependency 文件内容的内容签名或 MD5 校验和
.size：dependency 文件的字节大小
.timestamp：dependency 文件的修改时间
注意如果 Decider 函数中的一些参数没有影响到你决定 dependency 文件是否改变，你忽略掉这些参数是很正常的事情。
以上的三个属性在第一次运行的时候，可能不会出现。如果没有编译过，没有 target 创建过也没有.sconsign DB 文件存在过。所以，最好总是检查 prev_ni 的属性是否可用。
以下是一个基于 csig 的 decider 函数的例子，注意在每次函数调用的时候，dependency 文件的签名信息是怎么样通过 get_csig 初始化的：

```
env = Environment()

def config_file_decider(dependency, target, prev_ni):
    import os.path

    # We always have to init the .csig value...
    dep_csig = dependency.get_csig()
    # .csig may not exist, because no target was built yet...
    if 'csig' not in dir(prev_ni):
        return True
    # Target file may not exist yet
    if not os.path.exists(str(target.abspath)):
        return True
    if dep_csig != prev_ni.csig:
        # Some change on source file => update installed one
        return True
    return False
```

```
def update_file():
    f = open("test.txt","a")
    f.write("some line\n")
    f.close()

update_file()

# Activate our own decider function
env.Decider(config_file_decider)

env.Install("install","test.txt")
```

1.5、混合使用不同的方式来决定一个文件是否改变

有些时候，你想为不同的目标程序配置不同的选项。你可以使用 env.Decider 方法影响在指定 construction 环境下编译的目标程序。
例如，如果我们想使用 MD5 校验和编译一个程序，另一个使用文件的修改时间:

```
env1=Environment(CPPPATH=['.'])
env2=env1.Clone()
env2.Decider('timestamp-match')
env1.Program('prog-MD5','program1.c')
env2.Program('prog-timestamp','program2.c')
```

2、决定一个输入文件是否改变的旧函数

SCons2.0 之前的两个函数 SourceSignatures 和 TargetSignatures，现在不建议使用了。

3、隐式依赖：$CPPPATH Construction 变量

现在假设"Hello,World!"程序有一个#include 行需要包含 hello.h 头文件：

```
#include <hello.h>
int main()
{
    printf("Hello, %s!\n",string);
```

}

并且，hello.h 文件如下：

```
#define string "world"
```

在这种情况下，我们希望 SCons 能够认识到，如果 hello.h 文件的内容发生改变，那么 hello 程序必须重新编译。我们需要修改 SConstruct 文件如下：

```
Program('hello.c', CPPPATH='.')
```

$CPPPATH 告诉 SCons 去当前目录('.')查看那些被 C 源文件（.c 或.h 文件）包含的文件。

```
 % scons -Q hello
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
% edit hello.h
   [CHANGE THE CONTENTS OF hello.h]
% scons -Q hello
cc -o hello.o -c -I. hello.c
cc -o hello hello.o
```

首先注意到，SCons 根据$CPPPATH 变量增加了-I.参数，使得编译器在当前目录查找 hello.h 文件。

其次，SCons 知道 hello 程序需要重新编译，因为它扫描了 hello.c 文件的内容，知道 hello.h 文件被包含。SCons 将这些记录为目标文件的隐式依赖，当 hello.h 文件改变的时候，SCons 就会重新编译 hello 程序。

就像$LIBPATH 变量，$CPPPATH 也可能是一个目录列表，或者一个被系统特定路径分隔符分隔的字符串。

```
Program('hello.c', CPPPATH=['include', '/home/project/inc'])
```

4、缓存隐式依赖

扫描每个文件的#include 行会消耗额外的处理时间。

SCons 让你可以缓存它扫描找到的隐式依赖，在以后的编译中可直接使用。这需要在命令行中指定--implicit-cache 选项：

```
% scons -Q --implicit-cache hello
```

如果你不想每次在命令行中指定--implicit-cache 选项，你可以在 SConscript 文件中设置 implicit-cache 选项使其成为默认的行为：

SetOption('implicit-cache', 1)
SCons 默认情况下不缓存隐式依赖，因为--implicit-cache 使得 SCons 在最后运行的时候，只是简单的使用已经存储的隐式依赖，而不会检查那些依赖还是不是仍然正确。在如下的情况中，--implicit-cache 可能使得 SCons 的重新编译不正确：
1>当--implicit-cache 被使用，SCons 将会忽略$CPPPATH 或$LIBPATH 中发生的一些变化。如果$CPPPATH 的一个改变，使得不同目录下的内容不相同但文件名相同的文件被使用，SCons 也不会重新编译。
2>当--implicit-cache 被使用，如果一个同名文件被添加到一个目录，这个目录在搜索路径中的位置在同名文件上次被找到所在的目录之前，SCons 将侦测不到。

4.1、--implicit-deps-changed 选项

当使用缓存隐式依赖的时候，有些时候你想让 SCons 重新扫描它之前缓存的依赖。你可以运行--implicit-deps-changed 选项：
% scons -Q --implicit-deps-changed hello

4.2、--implicit-deps-unchanged 选项

默认情况下在使用缓存隐式依赖的时候，SCons 会注意到当一个文件已经被修改的时候，就会重新扫描文件更新隐式依赖信息。有些时候，你可能想即使源文件改变了，但仍然让 SCons 使用缓存的隐式依赖。你可以使用
--implicit-deps-unchanged 选项：
% scons -Q --implicit-deps-unchanged hello

5、显示依赖：Depends 函数

有些时候一个文件依赖另一个文件，是不会被 SCons 扫描器侦测到的。对于这种情况，SCons 允许你显示指定一个文件依赖另一个文件，并且无论何时被依赖文件改变的时候，需要重新编译。这个需要使用 Depends 方法：
hello=Program("hello.c")
Depends(hello,'other_file')
注意 Depends 方法的第二个参数也可以是一个节点对象列表：
hello=Program('hello.c')

```
goodbye=Program('goodbye.c')
Depends(hello,goodbye)
```

在这种情况下，被依赖的对象会在目标对象之前编译：

```
% scons -Q hello
cc -c goodbye.c -o goodbye.o
cc -o goodbye goodbye.o
cc -c hello.c -o hello.o
cc -o hello hello.o
```

6、来自外部文件的依赖：ParseDepends 函数

SCons 针对许多语言，有内置的扫描器。有些时候，由于扫描器实现的缺陷，扫描器不能提取出某些隐式依赖。

下面的例子说明了内置的 C 扫描器不能提取一个头文件的隐式依赖：

```
#define FOO_HEADER <foo.h>
#include FOO_HEADER
int main()
{
      return FOO;
}
% scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% edit foo.h
% scons -Q
scons: '.' is up to date.
```

显然，扫描器没有发现头文件的依赖。这个扫描器不是一个完备的 C 预处理器，没有扩展宏。

在这种情况下，你可能想使用编译器提取隐式依赖。ParseDepends 可以解析编译器输出的内容，然后显示建立所有的依赖。

下面的例子使用 ParseDepends 处理一个编译器产生的依赖文件，这个依赖文件是在编译目标文件的时候作为副作用产生的：

```
obj=Object('hello.c', CCFLAGS='-MD -MF hello.d', CPPPATH='.')
SideEffect('hello.d',obj)
ParseDepends('hello.d')
```

```
Program('hello', obj)
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
cc -o hello hello.o
% edit foo.h
% scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
```

从一个编译器产生的.d 文件解析依赖有一个先有鸡还是先有蛋的问题，会引发不必要的重新编译：

```
 % scons -Q
cc -o hello.o -c -MD -MF hello.d -I. hello.c
cc -o hello hello.o
% scons -Q --debug=explain
scons: rebuilding `hello.o' because `foo.h' is a new dependency
cc -o hello.o -c -MD -MF hello.d -I. hello.c
% scons -Q
scons: `.' is up to date.
```

第一次运行的时候，在编译目标文件的时候，依赖文件产生了。在那个时候，SCons 不知道 foo.h 的依赖。第二次运行的时候，目标文件被重新生成因为 foo.h 被发现是一个新的依赖。

ParseDepends 在调用的时候立即读取指定的文件，如果文件不存在马上返回。在编译过程中产生的依赖文件不会被再次自动解析。因此，在同样的编译过程中，编译器提取的依赖不会被存储到签名数据库中。这个 ParseDepends 的缺陷导致不必要的重新编译。因此，仅仅在扫描器对于某种语言不可用或针对特定的任务不够强大的情况下，才使用 ParseDepends。

7、忽略依赖：Ignore 函数

有些时候，即使一个依赖的文件改变了，也不想要重新编译。在这种情况下，你需要告诉 SCons 忽略依赖，如下所示：

```
 hello_obj=Object('hello.c')
hello = Program(hello_obj)
Ignore(hello_obj, 'hello.h')
```

```
% scons -Q hello
cc -c -o hello.o hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
% edit hello.h
  [CHANGE THE CONTENTS OF hello.h]
% scons -Q hello
scons: `hello' is up to date.
```

上面的例子是人为做作的，因为在真实情况下，如果 hello.h 文件改变了，你不可能不想重新编译 hello 程序。一个更真实的例子可能是，如果 hello 程序在一个目录下被编译，这个目录在多个系统中共享，多个系统有不同的 stdio.h 的拷贝。在这种情况下，SCons 将会注意到不同系统的 stdio.h 拷贝的不同，当你每次改变系统的时候，重新编译 hello。你可以避免这些重新编译，如下所示：

```
hello=Program('hello.c', CPPPATH=['/usr/include'])
Ignore(hello, '/usr/include/stdio.h')
```

Ignore 也可以用来阻止在默认编译情况下文件的产生。这是因为目录依赖它们的内容。所以为了忽略默认编译时产生的文件，你指定这个目录忽略产生的文件。注意到如果用户在 scons 命令行中请求目标程序，或者这个文件是默认编译情况下另一个文件的依赖，那么这个文件仍然会被编译。

```
 hello_obj=Object('hello.c')
hello = Program(hello_obj)
Ignore('.',[hello,hello_obj])
```

```
% scons -Q
scons: `.' is up to date.
% scons -Q hello
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello
scons: `hello' is up to date.
```

8、顺序依赖：Requires 函数

有时候，需要指定某一个文件或目录必须在某些目标程序被编译之前被编译或创建，但是那个文件或目录如果发生了改变，那个目标程序不需要重新编译。这样一种关系叫做顺序依赖（order-only dependency）因为它仅仅影响事物编译的顺序，它不是一种严格意义上的依赖关系，因为目标程序不需要随着依赖文件的改变而改变。

例如，你想在每次编译的时候创建一个文件用来标识编译执行的时间，版本号等等信息。这个版本文件的内容在每次编译的时候都会改变。如果你指定一个正常的依赖关系，那么每个依赖这个文件的程序在你每次运行 SCons 的时候都会重新编译。例如，我们可以使用一些 Python 代码在 SConstruct 文件中创建一个新的 version.c 文件，version.c 文件会记录我们每次运行 SCons 的当前日期，然后链接到一个程序：

```
import time
version_c_text="""
char *date="%s"
""" % time.ctime(time.time())
open('version.c', 'w').write(version_c_text)
hello=Program(['hello.c', 'version.c'])
```

如果我们将 version.c 作为一个实际的源文件，那么 version.o 文件在我们每次运行 SCons 的时候都会重新编译，并且 hello 可执行程序每次会重新链接。

```
% scons -Q hello
cc -o hello.o -c hello.c
cc -o version.o -c version.c
cc -o hello hello.o version.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
cc -o hello hello.o version.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
cc -o hello hello.o version.o
```

我们的解决方案是使用 Requires 函数指定 version.o 在链接之前必须重新编译，但是 version.o 的改变不需要引发 hello 可执行程序重新链接：

```
  import time

version_c_text = """
char *date = "%s";
""" % time.ctime(time.time())
open('version.c', 'w').write(version_c_text)

version_obj = Object('version.c')

hello = Program('hello.c',
                LINKFLAGS = str(version_obj[0]))

Requires(hello, version_obj)
```

注意到因为我们不再将 version.c 作为 hello 程序的源文件，我们必须找到其他的方式使其可以链接。在这个例子中，我们将对象文件名放到$LINKFLAGS 变量中，因为$LINKFLAGS 已经包含在$LINKCOM 命令行中了。

通过这些改变，当 hello.c 改变的时候，hello 可执行程序才重新链接：

```
 % scons -Q hello
cc -o version.o -c version.c
cc -o hello.o -c hello.c
cc -o hello version.o hello.o
% sleep 1
% scons -Q hello
cc -o version.o -c version.c
scons: `hello' is up to date.
% sleep 1
% edit hello.c
   [CHANGE THE CONTENTS OF hello.c]
% scons -Q hello
cc -o version.o -c version.c
cc -o hello.o -c hello.c
cc -o hello version.o hello.o
% sleep 1
% scons -Q hello
```

```
cc -o version.o -c version.c
scons: `hello' is up to date.
```

9、AlwaysBuild 函数

当一个文件传递给 AlwaysBuild 方法时，

```
hello=Program('hello.c')
AlwaysBuild(hello)
```

那么指定的目标文件将总是被认为是过时的，并且被重新编译：

```
 % scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q
cc -o hello hello.o
```

AlwaysBuild 函数并不意味着每次 SCons 被调用的时候，目标文件会被重新编译。在命令行中指定某个其他的目标，这个目标自身不依赖 AlwaysBuild 的目标程序，这个目标程序仅仅当它的依赖改变的时候才会重新编译：

```
 % scons -Q
cc -o hello.o -c hello.c
cc -o hello hello.o
% scons -Q hello.o
scons: `hello.o' is up to date.
```

第七章：环境

一个环境就是能够影响一个程序如何执行的值的集合。SCons 里面有三种不同类型的环境：

External Environment（外部环境）：
外部环境指的是在用户运行 SCons 的时候，用户环境中的变量的集合。这些变量在 SConscript 文件中通过 Python 的 os.environ 字典可以获得。

Construction Environment（构造环境）：
一个构造环境是在一个 SConscript 文件中创建的一个唯一的对象，这个对象包含了一些值可以影响 SCons 编译一个目标的时候做什么动作，以及决定从那一

个源中编译出目标文件。SCons 一个强大的功能就是可以创建多个构造环境，包括从一个存在的构造环境中克隆一个新的自定义的构造环境。

Execution Environment（执行环境）：
一个执行环境是 SCons 在执行一个外部命令编译一个或多个目标文件时设置的一些值。这和外部环境是不同的。
与 Make 不同，SCons 不会自动在不同的环境之间拷贝或导入值。这是一个刻意的设计选择，保证了不管用户外部环境的值是怎么样的，编译总是可以重复的。这会避免编译中的一些问题，比如因为一个自定义的设置使得使用了一个不同的编译器或编译选项，开发者的本地代码编译成功，但是 checked-in 后编译不成功，因为使用了不同的环境变量设置。

1、使用来自外部环境的值

当执行 SCons 的时候，外部环境的值通过 os.environ 字典获得。这就以为着在任何一个你想使用外部环境的 SConscript 文件需要增加一个 import os 语句：

```
import os
```

2、构造环境

在一个大型复杂的系统中，所有的软件都按照同样的方式编译是比较少见的。例如，不同的源文件可能需要不同的编译选项，或者不同的可执行程序需要链接不同的库。SCons 允许你创建和配置多个构造环境来控制和满足不同的编译需求。

2.1、创建一个构造环境：Environment 函数

一个构造环境由 Environment 方法创建：

```
env=Environment()
```

默认情况下，SCons 基于你系统中工具的一个变量集合来初始化每一个新的构造环境。
当你初始化一个构造环境时，你可以设置环境的构造变量来控制一个是如何编译的。例如：

```
import os
    env=Environment(CC='gcc', CCFLAGS='-O2')
    env.Program('foo.c')
```

2.2、从一个构造环境中获取值

你可以使用访问 Python 字典的方法获取单个的构造变量：

```
env=Environment()
print "CC is:", env['CC']
```

一个构造环境实际上是一个拥有方法的对象。如果你想直接访问构造变量的字典，你可以使用 Dictionary 方法：

```
env=Environment(FOO='foo', BAR='bar')
dict=env.Dictionary()
for key in ['OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
    print "key=%s, value=%s"   %   (key,dict[key])
```

如果你想循环并打印出构造环境中的所有变量：

```
env=Environment()
for item in sorted(env.Dictionary().items()):
    print "construction variable = '%s', value = '%s'"   %   item
```

2.3、从一个构造环境中扩展值：subst 方法

另一种从构造环境中获取信息的方式是使用 subst 方法。例如：

```
env=Environment()
print "CC is:", env.subst('$CC')
```

使用 subst 展开字符串的优势是结果中的构造变量会重新展开直到不能扩展为止。比如获取$CCCOM：

```
env=Environment(CCFLAGS='-DFOO')
print "CCCOM is:", env['CCCOM']
```

将会打印出没有扩展开的$CCCOM：

```
% scons -Q
CCCOM is: $CC $CCFLAGS $CPPFLAGS $_CPPDEFFLAGS
$_CPPINCFLAGS -c -o $TARGET $SOURCES
scons: '.' is up to date.
```

调用 subst 方法来获取$CCCOM：

```
env=Environment(CCFLAGS='-DFOO')
print "CCCOM is:", env.subst('$CCCOM')
```

将会递归地扩展所有的构造变量：

```
% scons -Q
```

```
CCCOM is: gcc -DFOO -c -o
scons: '.' is up to date.
```

2.4、处理值扩展的问题

如果扩展一个构造变量的时候，发生了问题，默认情况下会扩展成"空字符串，不会引起 scons 失败。

```
env=Environment()
print "value is:", env.subst('->$MISSING<-')

% scons -Q
value is: -><-
scons: '.' is up to date.
```

使用 AllowSubstException 函数会改变默认的行为。当值扩展的时候发生了异常，AllowSubstExceptions 控制了哪一个异常是致命的，哪一个允许安全地发生。默认情况下，NameError 和 IndexError 这两个异常允许发生。如果需要所有的构造变量名字存在，调用 AllowSubstExceptions：

```
AllowSubstExceptions()
env=Environment()
print "value is:", env.subst('->$MISSING<-')

% scons -Q
value is:
scons: *** NameError `MISSING' trying to evaluate `$MISSING'
File "/home/my/project/SConstruct", line 3, in <module>
```

也可以用来允许其他的异常发生，使用${...}构造变量语法。例如，下面的代码允许除零发生：

```
AllowSubstExceptions(IndexError, NameError, ZeroDivisionError)
env = Environment()
print "value is:", env.subst( '->${1 / 0}<-' )

% scons -Q
value is: -><-
scons: `.' is up to date.
```

2.5、控制默认的构造环境：DefaultEnvironment 函数

我们已经介绍过的所有的 Builder，比如 Program 和 Library，实际上使用一个默认的构造环境。

你可以控制默认构造环境的设置，使用 DefaultEnvironment 函数：

```
DefaultEnvironment(CC='/usr/local/bin/gcc')
```

这样配置以后，所有 Program 或者 Object 的调用都将使用/usr/local/bin/gcc 编译目标文件。

注意到 DefaultEnvironment 返回初始化了的默认构造环境对象，这个对象可以像其他构造环境一样被操作。所以如下的代码和上面的例子是等价的：

```
env=DefaultEnvironment()
env['CC']='/usr/local/bin/gcc'
```

DefaultEnvironment 函数常用的一点就是用来加速 SCons 的初始化。为了使得大多数默认的配置能够工作，SCons 将会搜索本地系统已经安装的编译器和其他工具。这个搜索过程会花费时间。如果你知道哪一个编译器或工具你想配置，你可以控制 SCons 执行的搜索过程通过指定一些特定的工具模块来初始化默认的构造环境：

```
env=DefaultEnvironment(tools=['gcc','gnulink'], CC='/usr/local/bin/gcc')
```

上面的例子告诉 SCons 显示配置默认的环境使用 GNU 编译器和 GNU 链接器设置，使用/usr/local/bin/gcc 编译器。

2.6、多个构造环境

构造环境的真正优势是你可以创建你所需要的许多不同的构造环境，每一个构造环境对应了一种不同的方式去编译软件的一部分或其他文件。比如，如果我们需要用-O2 编译一个程序，编译另一个用-g，我们可以如下做：

```
opt=Environment(CCFLAGS='-O2')
dbg=Environment(CCFLAGS='-g')
opt.Program('foo','foo.c')
dbg.Program('bar','bar.c')

% scons -Q
cc -o bar.o -c -g bar.c
cc -o bar bar.o
cc -o foo.o -c -O2 foo.c
cc -o foo foo.o
```

我们甚至可以使用多个构造环境去编译一个程序的多个版本：

```
opt=Environment(CCFLAGS='-O2')
dbg=Environment(CCFLAGS='-g')
opt.Program('foo','foo.c')
dbg.Program('foo','foo.c')
```

这个时候 SCons 会发生错误：

```
% scons -Q
scons: *** Two environments with different actions were specified for the same target: foo.o
File "/home/my/project/SConstruct", line 6, in <module>
```

这是因为这两个 Program 调用都隐式地告诉 SCons 产生一个叫做 foo.o 的目标文件。SCons 无法决定它们的优先级，所以报错了。为了解决这个问题，我们应该显示指定每个环境将 foo.c 编译成不同名字的目标文件：

```
opt=Environment(CCFLAGS='-O2')
dbg=Environment(CCFLAGS='-g')
o=opt.Object('foo-opt','foo.c')
opt.Program(o)
d=dbg.Object('foo-dbg','foo.c')
dbg.Program(d)
```

2.7、拷贝构造环境：Clone 方法

有时候你想多于一个构造环境对于一个或多个变量共享相同的值。当你创建每一个构造环境的时候，不是重复设置所有共用的变量，你可以使用 Clone 方法创建一个构造环境的拷贝。

Environment 调用创建一个构造环境，Clone 方法通过构造变量赋值，重载拷贝构造环境的值。例如，假设我们想使用 gcc 创建一个程序的三个版本，一个优化版，一个调试版，一个其他版本。我们可以创建一个基础构造环境设置$CC 为 gcc，然后创建两个拷贝：

```
env=Environment(CC='gcc')
opt=env.Clone(CCFLAGS='-O2')
dbg=env.Clone(CCFLAGS='-g')
env.Program('foo','foo.c')
o=opt.Object('foo-opt','foo.c')
opt.Program(o)
d=dbg.Object('foo-dbg','foo.c')
dbg.Program(d)
```

2.8、替换值：Replace 方法

你可以使用 Replace 方法替换已经存在的构造变量：

```
env=Environment(CCFLAGS='-DDEFINE1');
env.Replace(CCFLAGS='-DDEFINE2');
env.Program('foo.c')
```

你可以安全地针对那些不存在的构造变量调用 Replace 方法：

```
env=Environment()
env.Replace(NEW_VARIABLE='xyzzy')
print "NEW_VARIABLE = ", env['NEW_VARIABLE']
```

在这个例子中，构造变量被添加到构造环境中去了：

```
%scons -Q
NEW_VARIABLE = xyzzy
scons: '.' is up to date.
```

变量不会被扩展知道构造环境真正被用来编译目标文件的时候，同时 SCons 函数和方法的调用是没有顺序的，最后的替换可能被用来编译所有的目标文件，而不管 Replace 方法的调用是在编译方法的前后：

```
env=Environment(CCFLAGS='-DDEFINE1')
print "CCFLAGS = ", env['CCFLAGS']
env.Program("foo.c")
env.Replace(CCFLAGS='-DDEFIN2')
print "CCFLAGS = ", env['CCFLAGS']
env.Program("bar.c")
```

```
  % scons
scons: Reading SConscript files ...
CCFLAGS = -DDEFINE1
CCFLAGS = -DDEFINE2
scons: done reading SConscript files.
scons: Building targets ...
cc -o bar.o -c -DDEFINE2 bar.c
cc -o bar bar.o
cc -o foo.o -c -DDEFINE2 foo.c
cc -o foo foo.o
scons: done building targets.
```

因为替换发生在读取 SConscript 文件的时候，foo.o 编译的时候$CCFLAGS 变量已经被设置为-DDEFINE2，即使 Relapce 方法是在 SConscript 文件后面被调用的。

2.9、在没有定义的时候设置值：SetDefault 方法

有时候一个构造变量应该被设置为一个值仅仅在构造环境没有定义这个变量的情况下。你可以使用 SetDefault 方法，这有点类似于 Python 字典的 set_default 方法：

```
env.SetDefault(SPECIAL_FLAG='-extra-option')
```

当你编写你自己的 Tool 模块将变量应用到构造环境中的时候非常有用。

2.10、追加到值的末尾：Append 方法

你可以追加一个值到一个已经存在的构造变量，使用 Append 方法：

```
env=Environment(CCFLAGS=['-DMY_VALUE'])
env.Append(CCFLAGS=['-DLAST'])
env.Program('foo.c')
```

Scons 编译目标文件的时候会应用-DMY_VALUE 和-DLAST 两个标志：

```
% scons -Q
cc -o foo.o -c -DMY_VALUE -DLAST foo.c
cc -o foo foo.o
```

如果构造变量不存在，Append 方法将会创建它。

2.11、追加唯一的值：AppendUnique 方法

有时候仅仅只有在已经存在的构造变量没有包含某个值的时候，才会增加这个新值。可以使用 AppendUnique 方法：

```
env.AppendUnique(CCFLAGS=['-g'])
```

上面的例子，仅仅只有在$CCFLAGS 没有包含-g 值得时候才会增加-g。

2.12、在值的开始位置追加值：Prepend 方法

对于一个存在的构造变量，你可以使用 Prepend 方法追加一个值到它的值的开始位置。

```
env=Environment(CCFLAGS=['-DMY_VALUE'])
env.Prepend(CCFLAGS=['-DFIRST'])
env.Program('foo.c')

% scons -Q
cc -o foo.o -c -DFIRST -DMY_VALUE foo.c
cc -o foo foo.o
```

如果构造变量不存在，Prepend 方法会创建它。

2.13、在前面追加唯一值：PrependUnique 方法

仅仅在一个构造变量存在的值中没有包含将要增加的值的时候，这个值才被追加到前面，可以使用 PrependUnique 方法;

```
env.PrependUnique(CCFLAGS=['-g'])
```

3、控制命令的执行环境

当 SCons 编译一个目标文件的时候，它不会使用你用来执行 SCons 的同样的外部环境来执行一些命令。它会使用$ENV 构造变量作为外部环境来执行命令。这个行为最重要的体现就是 PATH 环境变量，它决定了操作系统将去哪里查找命令和工具，与你调用 SCons 使用的外部环境的不一样。这就意味着 SCons 将不能找到你在命令行里执行的所有工具。

PATH 环境变量的默认值是/usr/local/bin:/bin:/usr/bin。如果你想执行任何命令不在这些默认地方，你需要在你的构造环境中的$ENV 字典中设置 PATH。

最简单的方式就是当你创建构造环境的时候初始化这些值：

```
path=['/usr/local/bin', '/usr/bin']
env=Environment(ENV={'PATH':PATH})
```

以这种方式将一个字典赋值给$ENV 构造变量完全重置了外部环境，所以当外部命令执行的时候，设置的变量仅仅是 PATH 的值。如果你想使用$ENV 中其余的值，仅仅只是设置 PATH 的值，你可以这样做：

```
env['ENV']['PATH']=['/usr/local/bin','/bin','/usr/bin']
```

注意 SCons 允许你用一个字符串定义 PATH 中的目录，路径用路径分隔符分隔：

```
env['ENV']['PATH']='/usr/local/bin:/bin:/usr/bin'
```

3.1、从外部环境获得 PATH 值

你可能想获得外部的 PATH 来作为命令的执行环境。你可以使用来自 os.environ 的 PATH 值来初始化 PATH 变量：

```
import os
env=Environment(ENV={'PATH':os.environ['PATH']})
```

你设置可以设置整个的外部环境：

```
import os
env=Environment(ENV=os.environ)
```

3.2、在执行环境里增加 PATH 的值

常见的一个需求就是增加一个或多个自定义的目录到 PATH 变量中。

```
env=Environment(ENV=os.environ)
env.PrependENVPath('PATH','/usr/local/bin')
env.AppendENVPath('LIB','/usr/local/lib')
```