Scons 使用教程

蓬萊道人 2019-07-09 21:23:58 5983 收藏 20

分類專欄： C++ & C

1. 簡單編譯
2. SConstruct 文件
3. 編譯多個源文件
4. 編譯和鏈接庫文件
5. 節點對象
6. 依賴性
7. 環境

1.簡單編譯

源文件：hello.cpp

```
#include<iostream>
using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

用 SCons 編譯它，需要在一個名為 SConstruct 的文件:

```
Program('hello.cpp')
```

這個短小的配置文件給了 SCons 兩條信息：你想編譯什麼（一個可執行程序），你編譯的輸入文件（hello.cpp）。Program 是一個編譯器方法（builder_method），一個 Python 調用告訴 SCons，你想編譯一個可執行程序。 Program 編譯方法是 SCons 提供的許多編譯方法中一個。另一個是 Object 編譯方法，告訴 SCons 從指定的源文件編譯出一個目標文件，在 SConstruct 中為：

```
Object('hello.cpp')
```

使用 SCons，編譯之後想要清除不需要增加特殊的命令或目標名。你調用 SCons 的時候，使用-c 或--clean 選項，SCons 就會刪除合適的編譯產生的文件。

```
$ scons              //編譯源文件（自動讀取 SConstruct 中的內容）
$ scons -c           //清除 scons 編譯的文件
```

當你調用 Program 編譯方法的的時候，它編譯出來的程序名字是和源文件名是一樣的。下面的從 hello.cpp 源文件編譯一個可執行程序的調用將會在 POSIX 系統裡編譯出一個名為 hello 的可執行程序，在 windows 系統裡會編譯出一個名為 hello.exe 的可執行程序。如果你想編譯出來的程序的名字與源文件名字不一樣，你只需要在源文件名的左邊聲明一個目標文件的名字就可以了：

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

2. SConstruct 文件

如果你使用過 Make 編譯系統，你應該可以推斷出 SConstruct 文件就相當於 Make 系統中的 Makefile。SCons 讀取 SConstruct 文件來控制程序的編譯。

- SConstruct 文件實際上就是一個 Python 腳本。你可以在你的 SConstruct 文件中使用 Python 的註釋：

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

- 重要的一點是 SConstruct 文件並不完全像一個正常的 Python 腳本那樣工作，其工作方式更像一個 Makefile，那就是在 SConstruct 文件中 SCons 函數被調用的順序並不影響 SCons 你實際想編譯程序和目標文件的順序。換句話說，當你調用 Program 方法，你並不是告訴 SCons 在調用這個方法的同時馬上就編譯這個程序，而是告訴 SCons 你想編譯這個程序：

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

- 指定默認的目標文件

```
Default(targets)
env.Default(targets)
```

指定了默認的 target，如果在命令行中沒有顯示指定 target，那麼 scons 將編譯默認的 target，多次調用 Default 是合法的，實例：

```
Default('foo', 'bar', 'baz')
env.Default(['a', 'b', 'c'])
hello = env.Program('hello', 'hello.c')
env.Default(hello)
```

如果在 Default 中傳入參數 None，那麼將會清楚所有默認的 target：

```
Default(None)
```

3. 編譯多個源文件

- 通常情況下，你需要使用多個輸入源文件編譯一個程序。在 SCons 裡，只需要就多個源文件放到一個 Python 列表中就行了，如下所示：

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

- 你可以使用 Glob 函數，定義一個匹配規則來指定源文件列表，比如*,?以及[abc]等標準的 shell 模式。如下所示：

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

- 為了更容易處理文件名長列表，SCons 提供了一個 Split 函數，這個 Split 函數可以將一個用引號引起來，並且以空格或其他空白字符分隔開的字符串分割成一個文件名列表，示例如下：

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

或者：

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

- SCons 允許使用 Python 關鍵字參數來標識輸出文件和輸入文件。輸出文件是 target，輸入文件是 source，示例如下：

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

或者：

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

- 如果需要用同一個 SConstruct 文件編譯多個程序，只需要調用 Program 方法多次：

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

- 多個程序之間共享源文件是很常見的代碼重用方法。一種方式就是利用公共的源文件創建一個庫文件，然後其他的程序可以鏈接這個庫文件。另一個更直接，但是不夠便利的方式就是在每個程序的源文件列表中包含公共的文件，示例如下：

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

4．編譯和鏈接庫文件

（1）編譯靜態庫：

- 你可以使用 Library 方法來編譯庫文件：

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

- 除了使用源文件外，Library 也可以使用目標文件

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

- 你甚至可以在文件 List 裡混用源文件和目標文件

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

- 使用 StaticLibrary 顯示編譯靜態庫

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

（2）編譯動態庫：

如果想編譯動態庫（在 POSIX 系統裡）或 DLL 文件（Windows 系統），可以使用 SharedLibrary：

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

（3）鏈接庫文件：

- 鏈接庫文件的時候，使用**$LIBS** 變量指定庫文件，使用**$LIBPATH** 指定存放庫文件的目錄：

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

注意到，你不需要指定庫文件的前綴（比如 lib）或後綴（比如.a 或.lib），SCons 會自動匹配。

- 默認情況下，鏈接器只會在系統默認的庫目錄中尋找庫文件。SCons 也會去$LIBPATH 指定的目錄中去尋找庫文件。$LIBPATH 由一個目錄列表組成，如下所示：

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

5. 節點對象

- **編譯方法返回目標節點列表**

所有編譯方法會返回一個節點對象列表，這些節點對象標識了那些將要被編譯的目標文件。這些返回出來的節點可以作為參數傳遞給其他的編譯方法。例如，假設我們想編譯兩個目標文件，這兩個目標有不同的編譯選項，並且最終組成一個完整的程序。這意味著對每一個目標文件調用 Object 編譯方法，如下所示：

```
1.  Object('hello.cpp', CCFLAGS='-DHELLO')
2.  Object('goodbye.cpp', CCFLAGS='-DGOODBYE')
3.  Program(['hello.o', 'goodbye.o'])
```

這樣指定字符串名字的問題就是我們的 SConstruct 文件不再是跨平台的了。因為在 Windows 裡，目標文件成為了 hello.obj 和 goodbye.obj。一個更好的解決方案就是將 Object 編譯方法返回的目標列表賦值給變量，這些變量然後傳遞給 Program 編譯方法：

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

- **顯示創建文件和目錄節點**

在 SCons 裡，表示文件的節點和表示目錄的節點是有清晰區分的。SCons 的 File 和 Dir 函數分別返回一個文件和目錄節點：

```
1.  hello_c=File('hello.cpp')
2.  Program(hello_c)
```

通常情況下，你不需要直接調用 File 或 Dir，因為調用一個編譯方法的時候，SCons 會自動將字符串作為文件或目錄的名字，以及將它們轉換為節點對象。只有當你需要顯示構造節點類型傳遞給編譯方法或其他函數的時候，你才需要手動調用 File 和 Dir 函數。有時候，你需要引用文件系統中一個條目，同時你又不知道它是一個文件或一個目錄，你可以調用 Entry 函數，它返回一個節點可以表示一個文件或一個目錄：

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

- **將一個節點的文件名當作一個字符串**

如果你不是想打印文件名，而是做一些其他的事情，你可以使用內置的 Python 的 str 函數。例如，你想使用 Python 的 os.path.exists 判斷一個文件是否存在：

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

- **GetBuildPath：從一個節點或字符串中獲得路徑**

env.GetBuildPath(file_or_list)返回一個節點或一個字符串表示的路徑。它也可以接受一個節點或字符串列表，返回路徑列表。如果傳遞單個節點，結果就和調用 str(node)一樣。路徑可以是文件或目錄，不需要一定存在：

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

將會打印輸出如下：

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

6. 依賴性

- **隱式依賴：$CPPPATH Construction 變量**

```
#include <iostream>
#include "hello.h"
using namespace std;

int main()
{
    cout << "Hello, " << string << endl;
    return 0;
}
```

並且，hello.h 文件如下：

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

在這種情況下，我們希望 SCons 能夠認識到，如果 hello.h 文件的內容發生改變，那麼 hello 程序必須重新編譯。我們需要修改 SConstruct 文件如下：

```
Program('hello.cpp', CPPPATH='.')  #CPPPATH 告訴 SCons 去當前目錄('.')查看那些被 C 源文件(.c 或.h 文件)包含的文件。
```

就像$LIBPATH 變量，**$CPPPATH** 也可能是一個目錄列表，或者一個被系統特定路徑分隔符分隔的字符串。

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

7．環境

（1）外部環境

外部環境指的是在用戶運行 SCons 的時候，用戶環境中的變量的集合。這些變量在 SConscript 文件中通過 Python 的 os.environ 字典可以獲得。你想使用外部環境的 SConscript 文件需要增加一個 import os 語句。

（2）構造環境

一個構造環境是在一個 SConscript 文件中創建的一個唯一的對象，這個對象包含了一些值可以影響 SCons 編譯一個目標的時候做什麼動作，以及決定從那一個源中編譯出目標文件。SCons 一個強大的功能就是可以創建多個構造環境，包括從一個存在的構造環境中克隆一個新的自定義的構造環境。

- **創建一個構造環境：Environment 函數**

默認情況下，SCons 基於你系統中工具的一個變量集合來初始化每一個新的構造環境。當你初始化一個構造環境時，你可以設置環境的構造變量來控制一個是如何編譯的。例如：

```
1.  import os
2.  env=Environment(CC='gcc', CCFLAGS='-O2')
3.  env.Program('foo.c')
4.  或者
5.  env=Environment(CXX='/usr/local/bin/g++', CXXFLAGS='-02')
6.  env.Program('foo.cpp')
```

- **從一個構造環境中獲取值**

你可以使用訪問 Python 字典的方法獲取單個的構造變量：

```
1.  env=Environment()
2.  print "CC is:", env['CC']
3.  print "CXX is:", env['CXX']
```

一個構造環境實際上是一個擁有方法的對象。如果你想直接訪問構造變量的字典，你可以使用 Dictionary 方法：

```
1.  env=Environment(FOO='foo', BAR='bar')
2.  dict=env.Dictionary()
```

```
3.          for key in ['OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
4.              print "key=%s, value=%s"  %  (key,dict[key])
```

- **默認的構造環境：DefaultEnvironment 函數**

  你可以控制默認構造環境的設置，使用 DefaultEnvironment 函數：

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

  這樣配置以後，所有 Program 或者 Object 的調用都將使用/usr/local/bin/gcc 編譯目標文件。注意到 DefaultEnvironment 返回初始化了的默認構造環境對象，這個對象可以像其他構造環境一樣被操作。所以如下的代碼和上面的例子是等價的：

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

- **多個構造環境**

  構造環境的真正優勢是你可以創建你所需要的許多不同的構造環境，每一個構造環境對應了一種不同的方式去編譯軟件的一部分或其他文件。比如，如果我們需要用-O2 編譯一個程序，編譯另一個用-g，我們可以如下做：

```
1.          opt=Environment(CCFLAGS='-O2')
2.          dbg=Environment(CCFLAGS='-g')
3.          opt.Program('foo','foo.cpp')
4.          dbg.Program('bar','bar.cpp')
```

- **拷貝構造環境：Clone 方法**

  有時候你想多於一個構造環境對於一個或多個變量共享相同的值。當你創建每一個構造環境的時候，不是重複設置所有共用的變量，你可以使用 Clone 方法創建一個構造環境的拷貝。Environment 調用創建一個構造環境，Clone 方法通過構造變量賦值，重載拷貝構造環境的值。例如，假設我們想使用 gcc 創建一個程序的三個版本，一個優化版，一個調試版，一個其他版本。我們可以創建一個基礎構造環境設置$CC 為 gcc，然後創建兩個拷貝：

```
1.          env=Environment(CC='gcc')
2.          opt=env.Clone(CCFLAGS='-O2')
3.          dbg=env.Clone(CCFLAGS='-g')
4.          env.Program('foo','foo.cpp')
5.          o=opt.Object('foo-opt','foo.cpp')
6.          opt.Program(o)
7.          d=dbg.Object('foo-dbg','foo.cpp')
8.          dbg.Program(d)
```

- **替換值：Replace 方法**

  你可以使用 Replace 方法替換已經存在的構造變量：

```
1.          env=Environment(CCFLAGS='-DDEFINE1');
2.          env.Replace(CCFLAGS='-DDEFINE2');
3.          env.Program('foo.cpp')
```

- **在沒有定義的時候設置值：SetDefault 方法**

有時候一個構造變量應該被設置為一個值僅僅在構造環境沒有定義這個變量的情況下。你可以使用 SetDefault 方法，這有點類似於 Python 字典的 set_default 方法：

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

- **控制目標文件的路徑：env.Install 方法**

```
1.  test = env.Program('test.cpp')
2.  env.Install('bin', 'test.exe') #表示要將 test.exe 放到 bin 目錄下
```

- **執行 SConscript 腳本文件**

```
1.  SConscript(scripts, [exports, variant_dir, duplicate])
2.  env.SConscript(scripts, [exports, variant_dir, duplicate])
3.  SConscript(dirs=subdirs, [name=script, exports, variant_dir, duplicate])
4.  env.SConscript(dirs=subdirs, [name=script, exports, variant_dir, duplicate])
```

調用該 SConscript 函數有兩種方法：

第一種方法是明確指定一個或多個 scripts 作為第一個參數。可以將單個腳本指定為字符串；多個腳本則必須指定為列表（顯式或由函數創建 Split），例子：

```
1.  SConscript('SConscript')                          #在當前目錄中運行 SConscript
2.  SConscript('src / SConscript')                    #在 src 目錄中運行 SConscript
3.  SConscript(['src / SConscript','doc / SConscript']) # 執行多個腳本
```

第二種方法是將（子）目錄名稱列表指定為 dirs=subdirs 參數。在這種情況下，scons 將 在每個指定目錄中執行名為 SConscript 的輔助配置文件 。您可以 通過提供可選的 name = keyword 參數來指定除了名為 SConscript 以外 script。例子：

```
1.  SConscript(dirs ='.')      #在當前目錄中運行 SConscript
2.  SConscript(dirs ='src')    #在 src 目錄中運行 SConscript
3.  SConscript(dirs = ['src','doc'])
4.  SConscript(dirs = [' sub1','sub2'],name ='MySConscript')
```

可選 exports 參數提供變量名稱列表或要導出到 script 的命名值字典。這些變量僅在本地導出到指定 script(s)的變量，並且不會影響 Export 函數使用的全局變量池 。子腳本 Script 必須使用 Import 函數來導入變量。例子：

```
1.  foo = SConscript('sub/SConscript',exports ='env')
2.  SConscript('dir/SConscript',exports = ['env','variable'])
3.  SConscript(dirs ='subdir',exports ='env variable' )
4.  SConscript(dirs = ['one','two','three'],exports ='shared_info')
```

如果提供 variant_dir 參數，Sconscript 位於源碼目錄之下，就像位於 variant_dir 目錄下一樣，例子一：

```
1.  SConscript('src/SConscript', variant_dir = 'build')
2.  等價於：
3.  VariantDir('build', 'src')      # 指定 obj 文件的目錄
```

```
4.          SConscript('build/SConscript')
```

例子二：

```
1.          SConscript('SConscript', variant_dir = 'build')
2.          等價於：
3.          VariantDir('build', '.')
4.          SConscript('build/SConscript')
```

如果沒有提供 variant_dir 參數，那麼參數 duplicate 參數將會被忽略，這個參數表示是否備份目標文件。

（3）執行環境

一個執行環境是 SCons 在執行一個外部命令編譯一個或多個目標文件時設置的一些值。這和外部環境是不同的。

- **控制命令的執行環境**

當 SCons 編譯一個目標文件的時候，它不會使用你用來執行 SCons 的同樣的外部環境來執行一些命令。它會使用$ENV 構造變量作為外部環境來執行命令。這個行為最重要的體現就是 PATH 環境變量，它決定了操作系統將去哪裡查找命令和工具，與你調用 SCons 使用的外部環境的不一樣。這就意味著 SCons 將不能找到你在命令行裡執行的所有工具。PATH 環境變量的默認值是 /usr/local/bin:/bin:/usr/bin。如果你想執行任何命令不在這些默認地方，你需要在你的構造環境中的$ENV 字典中設置 PATH，最簡單的方式就是當你創建構造環境的時候初始化這些值：

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

以這種方式將一個字典賦值給$ENV 構造變量完全重置了外部環境，所以當外部命令執行的時候，設置的變量僅僅是 PATH 的值。如果你想使用$ENV 中其餘的值，僅僅只是設置 PATH 的值，你可以這樣做：

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

注意 SCons 允許你用一個字符串定義 PATH 中的目錄，路徑用路徑分隔符分隔：

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

- **從外部環境獲得 PATH 值**

你可能想獲得外部的 PATH 來作為命令的執行環境。你可以使用來自 os.environ 的 PATH 值來初始化 PATH 變量：

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

你設置可以設置整個的外部環境：

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

- **在執行環境裡增加 PATH 的值**

常見的一個需求就是增加一個或多個自定義的目錄到 PATH 變量中：

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

本文參考：https://blog.csdn.net/andyelvis/article/category/948141