# SCons入门

## SCons介绍

scons是一个Python写的自动化构建工具，从构建这个角度说，它跟GNU make是同一类的工具，是一种改进，并跨平台的gnu make替代工具，其集成功能类似于autoconf/automake。scons是一个更简便，更可靠，更高效的编译软件。

scons是一个Python写的自动化构建工具，从构建这个角度说，它跟GNU make是同一类的工具。它是一种改进，并跨平台的gnu make替代工具，其集成功能类似于autoconf/automake。scons是一个更简便，更可靠，更高效的编译软件。

Scons是一个开放源码、以Python语言编码的自动化构建工具，可用来替代make编写复杂的makefile。并且scons是跨平台的，只要scons脚本写的好，可以在Linux和Windows下随意编译。

SCons 的设计目标就是让开发人员更容易、更可靠和更快速的建造软件。

与传统的 make 工具比较，SCons 具有以下优点：

* 使用 Python 脚本做为配置文件；
* 对于C, C++ 和 Fortran, 内建支持可靠自动依赖分析，不用像 make 工具那样需要 执行"make depends"和"make clean"就可以获得所有的依赖关系。
* 内建支持 C, C++, D, Java, Fortran, Yacc, Lex, Qt，SWIG 以及 Tex/Latex。 用户还可以根据自己的需要进行扩展以获得对需要编程语言的支持。
* 支持 make -j 风格的并行建造。相比 make -j, SCons 可以同时运行 N 个工作，而 不用担心代码的层次结构。
* 使用 Autoconf 风格查找头文件，函数库，函数和类型定义。
* 良好的夸平台性。SCons 可以运行在 Linux, AIX, BSD, HP/UX, IRIX, Solaris, Windows, Mac OS X 和 OS/2 上。

## hello world

scons由Sconstruct 作为入口，控制如何进行编译操作。Sconstruct 本身是一个python文件，故需要遵循python的语法，以及能使用一些python的方法。(如我们可以用print 来debug）

这有一段很简单的hello.cpp

#include <iostream>

int main() {

std::cout << "hello world" << std::endl;

}

以及一个很简单的Sconstruct

Program("hello.cpp")

Program是Scons中的一个编译方法(builder\_method)， 告诉Scons 我们想要把hello.cpp 编译成一个可执行文件。

保证Sconstruct 和hello.cpp 在同一个文件夹下，执行scons，就可以完成编译，生成可执行文件hello。

可以看到，我们只指定了一个cpp文件，scons会默认给可执行文件一个名字，以及完成.o文件的生成，非常智能。当然，我们也能指定输出文件的名字，Program("target\_name", hello.cpp")

另外，除了Program ，还有其他很多builder\_method， 如Object, SharedLibrary，StaticLibrary，LoadableModule，StaticObject，CFile

## 编译多个文件

编译多个文件非常简单

* 直接使用list Program(["a.cpp", "b.cpp", "c.cpp"])即可。
* 也可以使用Glob方法
* source = Glob("src/\*.cpp")
* print source # python语法，可以打印出来debug
* Program(["hello.cpp"] + source)

用这种方式把src下的文件都加进来

* Program(Split("a.cpp, b.cpp c.cpp") Split 也是SCons提供的一个方法，顾名思义了。

## 编译一个库

把Program改为 Library (或者StaticLibrary，这两者是一样的)即可。

Library("t", Glob("src/\*.cpp"))

这样就能得到一个静态库了。 如果如要一个动态库，则可以使用SharedLibrary。

## 链接一个库

上边我们学会如何编译一个库了，那么如何链接呢？也很简单，加个参数即可

source = Glob("src/\*.cpp")

SharedLibrary("t", source)

Program(["hello.cpp"], LIBS=["t"], LIBPATH=".")

Program 可以理解为python 的一个方法，很多参数都有默认值，我们要做的只用覆盖其默认值即可。如上，我们指定引入LIBS。同样的，LIBS参数也可以是个str，LIBPATH也可以是个list，放上所有要查找的路径，如['/usr/lib', '/usr/local/lib']，这里就不赘述了。

## 略微高级一点的功能

判断是否需要重新编译

SCons很智能，只会编译需要编译的内容。比如我刚执行完scons，再次执行，则会提示scons: . is up to date.。 那么他是如何做到的呢？也不复杂，依赖一个Decider的方法，以及一个.sconsign.dblite文件。

默认情况下，如果文件的md5值改变了，才会重新编译。每次编译，SCons都会把md5存起来，再次执行时，如果md5没变，则不需要rebuild。

如果我们不希望使用md5，而是使用文件修改时间呢？很简单，增加Decider('timestamp-newer') （默认情况下为md5）。也可以使用'MD5-timestamp，则他们一起变了才会rebuild。

我们前面也说到，Decider是一个方法，那很显然，我们也可以自己写个decider方法，详细的可以看[scons-user.pdf](https://scons.org/doc/production/PDF/scons-user.pdf) 6.1.5，这里不细写了。。

## env

env分为三种

* external enviroment 保存在os.environ 中，和scons其实没太大关系，保存着一些系统定义的环境变量，如PATH之类
* construction environment

这个比较重要，我们可能希望不同的源文件使用不同的编译参数，这时候我们就能设置不同的编译环境，指定哪个文件用哪种编译。construction environment 中保存了构建相关的一些参数。

使用方式如下

env1 = Environment(CXX = 'gcc') # 创建一个env

print env1["CXX"] # 获取参数

env2 = env1.Clone(CXX = 'g++') # 复制一个env

env1.Replace(CXX = 'g++') # 修改参数

env1["CXX"] = "clang++" #再修改参数

env1.MergeFlags("-g") #增加一个flag

就可以构建出一个env，除了CXX 被修改以外，其他参数均不变。另外，我们也可以像一个dict一样，去获取env中的内容。

* execution enviroment

这其实就是construction environment 中的一个变量ENV

env = Environment()

print env["ENV"]

env2 = Environment(ENV=os.environ)

env3 = Environment(ENV = {"PATH" : os.environ["PATH"]})

几个简单的例子，一看就明白了

## 命令行输入的参数

比如我们想通过一个debug字段来控制是否开启debug模式，怎么做呢？可以通过ARGUMENTS

env = Environment()

debug = ARGUMENTS.get("debug", 0)

if int(debug):

print "in debug mode"

scons debug=1就可以了。