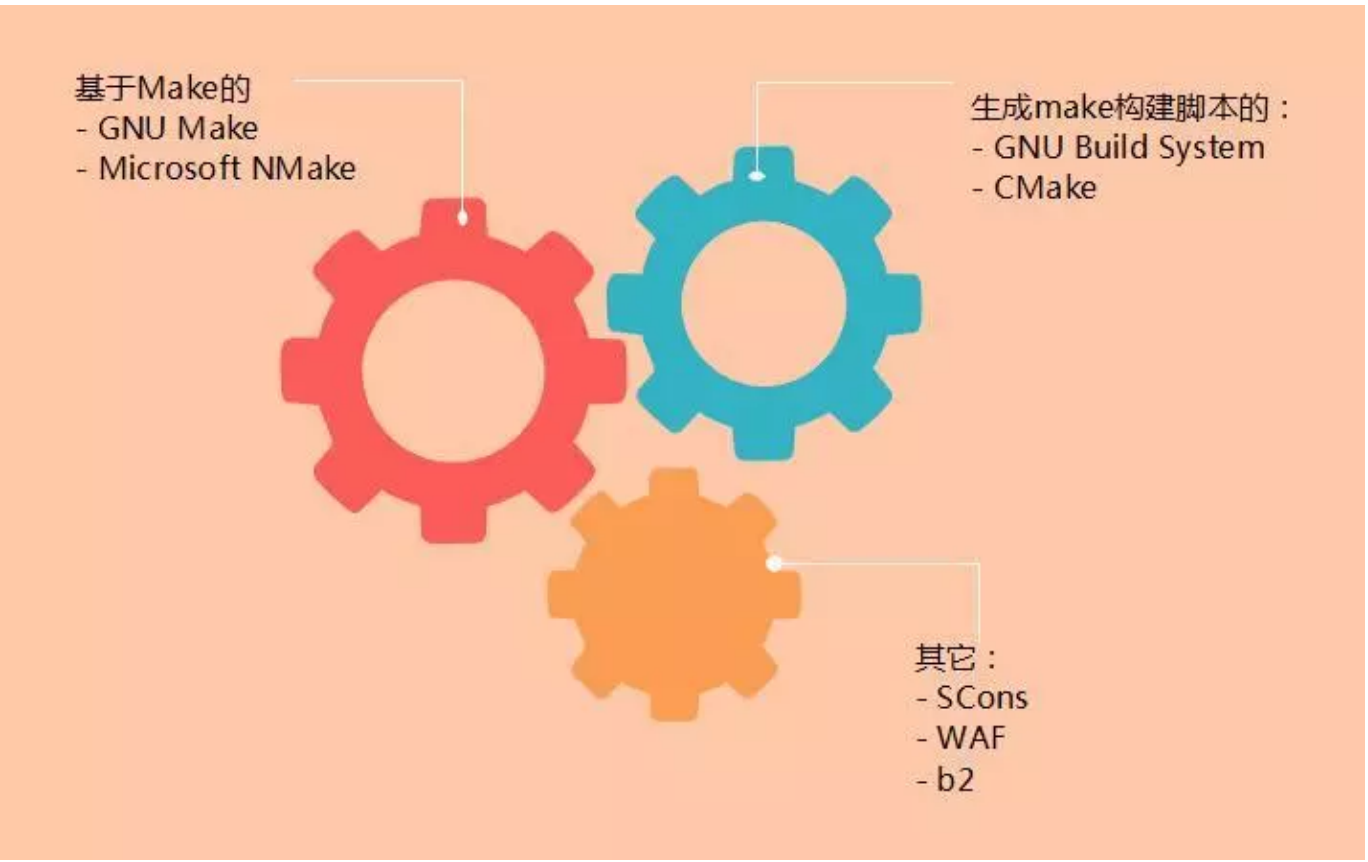
C++构建系统的选择

0.179 2016.11.02 16:15:14 字数 3528 阅读 3995

C++少说也用了十年了，从简单的Hello World到200万行的游戏项目，编译和构建的工具也经历了各种升级。最终的开发环境，选择了Clang+GDB+CMake。当然不断改进和升级开发工具的脚步尚未停止，只要能提高开发效率，怎样折腾都是值得的。

期间经历了：

1. 直接调用编译和链接命令
2. 使用Makefile
3. 使用CMake
4. 不断尝试其他构建系统，如：b2、WAF、SCons



C++构建系统

对构建系统的要求

由于C/C++本身的特性，如：跨平台、高性能等、编写复杂等，对构建系统也是提出了一定的要求：

- **支持并行编译**：构建系统能否支持并行编译？对于编译速度的要求，我给自己定的目标是<10min，超过10min要么换机器，要么想办法优化代码依赖。上百万行的代码，并行编译时必须的，否则一不小心改一行代码等个把小时，这样开发时间白白浪费在编译上太不值得了。
- **自动生成依赖**：构建系统是否仅仅编译刚修改过的及其依赖的文件？代码的依赖关系，要我们自己去手动写脚本（一般gcc/clang的话，使用 `gcc -M xx.cpp` ）？
- **跨平台**：构建系统能否仅写一份构建脚本，支持多种平台？有些项目需要进行交叉编译，测试环境和运行环境是在不同的平台环境下。
- **支持自定义构建目标**： 构建系统必须支持扩展，支持自定义Target等。如：protobuf文件可以根据依赖规则自动生成.h、.cpp；自定义一些用于打包或测试的命令（ `make pack` 、 `make test` ）。

本文下面大概介绍一下刚提到的构建系统，具体用法不赘述，官方网站是最好的开始地方。若有必要会另起文章详细讲解如何使用及其工作原理。

基于make的

GNU Make

对于玩Linux的人来说，这是太熟悉不过的东西了。小规模的项目或仅自己玩的项目，手写Makefile完全就足够了。

GNU Make 是一个控制源码生成可执行文件或其他文件的工具。需要一个叫Makefile的文件来说明构建的目标和规则。

最简单的规则大概是这样的：

```
target:   dependencies ...
          commands
          ...
```

意思是：生成 `target` ，依赖于 `dependencies` ，如果 `dependencies` 有修改或者 `target` 不存在，就逐个执行下面的 `commands` 去生成 `target` 。

下面贴一个复杂的Makefile感受下：

```
CXX      = g++
CXXFLAGS = -g -I../proto.client -I../common
LDFLAGS  = -L../common  -L../proto.client/ -lproto.client -L/usr/local/lib -lzmq -lprotobuf -ltinyworld

OBJS = main.o

SRCS = $(OBJS:%.o=%.cpp)
DEPS = $(OBJS:%.o=.%.d)

TARGET=gateserver

.PHONY: all clean

all : $(TARGET)

include $(DEPS)
$(DEPS): $(SRCS)
    @$(CXX) -M $(CXXFLAGS) $< > $@.$$$$; \\
        sed 's,\\($*\\)\\.o[ :]*,\\1.o $@ : ,g' < $@.$$$$ >$@; \\
        rm -f $@.$$$$

$(OBJS): %.o: %.cpp
    $(CXX) -c $(CXXFLAGS) $< -o $@

$(TARGET): $(OBJS) ../common/libtinyworld.a
    $(CXX) $(OBJS) -o $@ $(CXXFLAGS) $(LDFLAGS)

clean:
    @rm -rf $(TARGET)
```

Microsoft NMake

在Windows下面做开发，Visual Studio基本上完全胜任。微软自己的IDE功能强大，对于项目构建的管理IDE帮着你搞定了。VS的构建的管理其实用的是微软自己的Make，叫NMAKE。脚本还是IDE，各有千秋：IDE好处就是它什么都帮你干了，简单方便；坏处就是对构建的方式和过程了解的比较浅，自由度没那么大，遇到大型项目的特殊需求时要各种查资料。

MSDN上面的NMAKE脚本示例：

```
# Sample makefile

!include <win32.mak>
```

```
all: simple.exe challeng.exe

.c.obj:
  $(cc) $(cdebug) $(cflags) $(cvars) $*.c

simple.exe: simple.obj
  $(link) $(ldebug) $(conflags) -out:simple.exe simple.obj $(conlibs) lsapi32.lib

challeng.exe: challeng.obj md4c.obj
  $(link) $(ldebug) $(conflags) -out:challeng.exe $** $(conlibs)
```

自动生成make脚本的

手动写make脚本自由度大，为了自由度，它的设计比较简单，有许多上述对构建系统的要求它没法支持。如：GUN Make没法自己知道代码的依赖，需要借助编译器来自己写脚本；跨平台就更不可能了。

还有一个重要的影响就是对于环境的自动检测。如果你的代码发布出去，任何一个人下载下来需要进行编译，他的编译器、操作系统环境、依赖的第三方库的位置和版本都会有差异，如何进行编译？难到要下载你代码的人去手动修改你的Makefile吗？当然不是，这个时候在编译之前还需要一步：检测当前编译环境、操作系统环境、第三方库的位置等，不满足要求就直接报错，检测到所有依赖后再根据这些信息生成适合你当前系统的Makefile，然后才能进行编译。

GNU Build System

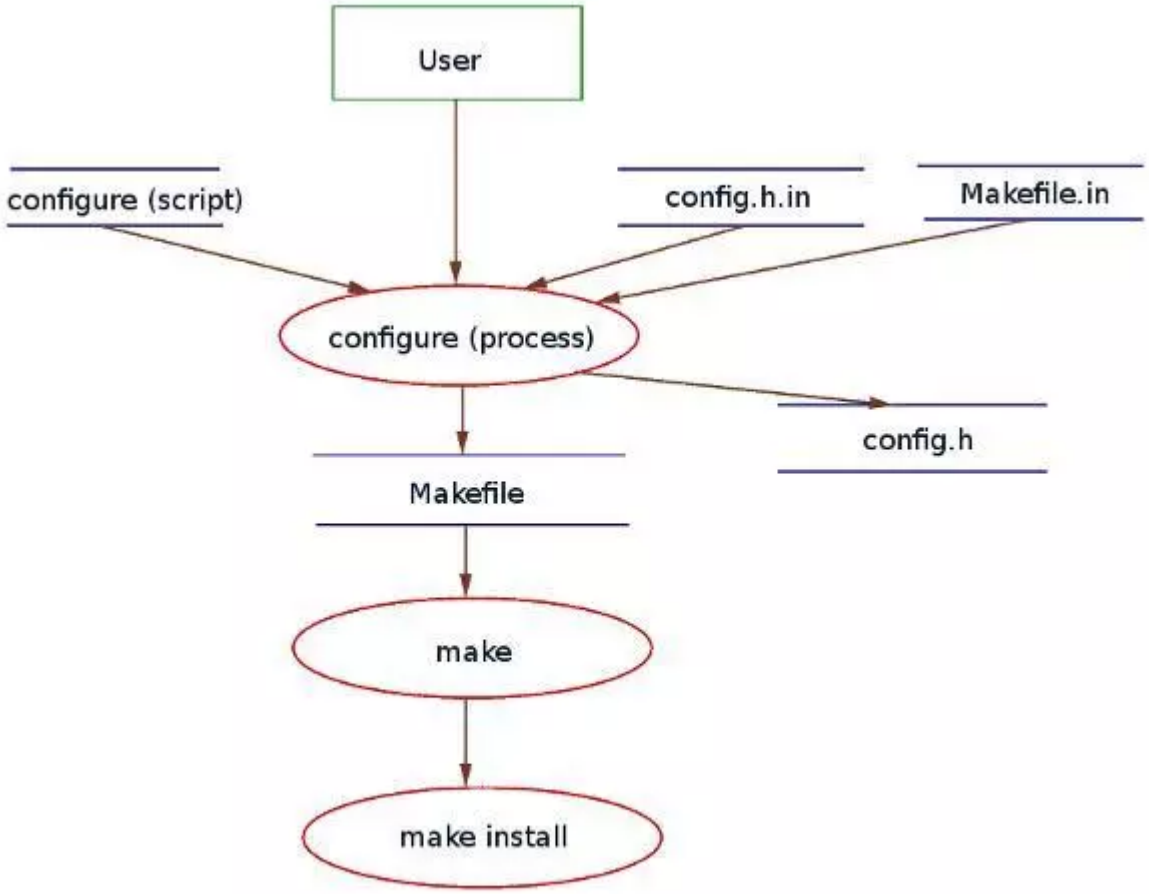
认识GNU Build System可以从两个角度入手：使用者和开发者。主要包含三大模块：

- Autoconf
- Automake
- Libtool

站在使用者的角度，GNU Build System为我们提供了源码包**编译安装**的方式：

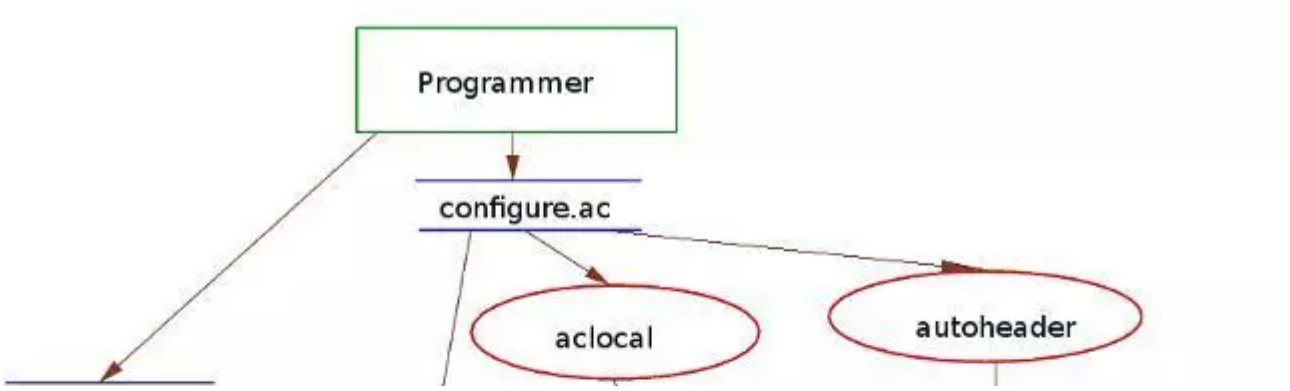
```
tar -xvzf package-name.version.tar.gz # tar -xvjf package-name.version.tar.bz2
cd package-name.version
./configure --prefix=xxx
make
make install
```

其中的 `configure` 就是检测环境，生成Makefile的脚本。大概的过程如下：



构建和安装

站在开发者的角度，GNU Build System 为我们广大程序员提供了**编写构建规则和检查安装环境**的功能。



GNU Build System

要发布自己的源码，首先需要一个Autoconf的 `configure.ac` ，最简单的长这样：

```
AC_INIT([hello], [1.0])
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADERS(config.h)
AC_PROG_CC
AC_CONFIG_FILES(Makefile)
AC_PROG_INSTALL
AC_OUTPUT
```

其次还需要一个Automake的 `Makefile.am` 来描述构建规则，看起来这这样的：

```
AC_INIT([hello], [1.0])
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADERS(config.h)
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_CONFIG_FILES(Makefile)
AC_PROG_INSTALL
AC_OUTPUT
```

定义好检查环境和配置的 `configure.ac` 和描述构建规则的 `Makefile.am` ，生成一个可以发布的源码包大概过程如下：

```
aclocal
autoconf
autoheader
touch NEWS README AUTHORS ChangeLog
automake -a
./configure
make
make dist
```

CMake

CMake

CMake是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装（编译过程）。他能够输出各种各样的Makefile或者project文件，能检查编译器所支持的C++特性，类似UNIX下的automake。CMake 并不直接建构出最终的软件，而是产生标准的建构脚本（如Unix 的 Makefile 或 Windows Visual C++ 的 projects/workspaces），然后再使用相应的工具进行编译。

CMak的特点主要有：

1. 开放源代码， 使用类 BSD 许可发布。 http://cmake.org/HTML/Copyright. html
2. 跨平台， 并可生成 native 编译配置文件， 在 Linux/Unix 平台， 生成 makefile， 在苹果平台， 可以生成 xcode， 在 Windows 平台， 可以生成 MSVC 的工程文件。
3. 能够管理大型项目， KDE4 就是最好的证明。
4. 简化编译构建过程和编译过程。 CMake 的工具链非常简单： `cmake+make` 。
5. 可扩展， 可以为 cmake 编写特定功能的模块， 扩充 cmake 功能。

其实CMake工具包不仅仅提供了编译，还有：支持单元测试的CTest，支持不同平台打包的CPack，自动化测试及其展示的CDash。有兴趣的访问官方网站学习：https://cmake.org/

一般，在每个源码目录下都有一个 CMakeLists.txt，看起来是这样的：

```
cmake_minimum_required (VERSION 2.6)
project (Tutorial)
# The version number.
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)
```

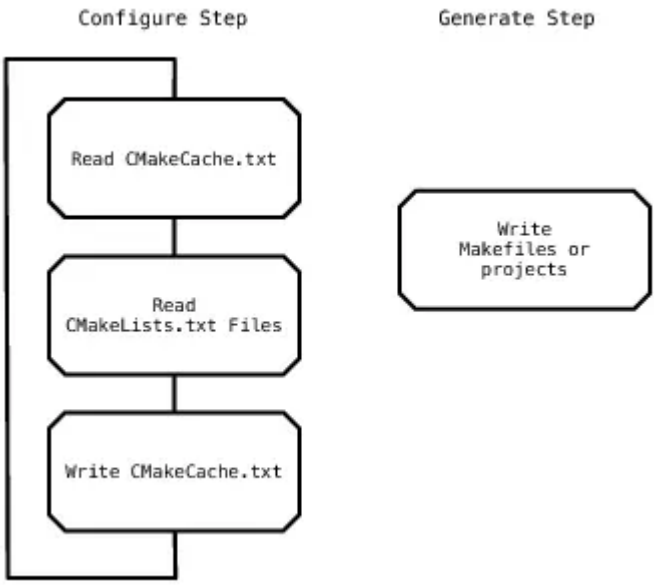
```
# configure a header file to pass some of the CMake settings
# to the source code
configure_file (
  "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
  "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  )

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
include_directories("${PROJECT_BINARY_DIR}")

# add the executable
add_executable(Tutorial tutorial.cxx)
```

使用的时候：

第一步：根据CMakeLists.txt生成Makefile，命令如下：

```
mkdir path-to-build
cd path-tob-build
cmake path-to-source
```

cmake的过程可以分为**配置**和**生成**过程。配置的时候优先从CMakeCache.txt中读取设置，然后再扫一遍CMakeList.txt中的设置，该步骤会检查第三方库和构建过程的变量；生成步骤则根据当前的环境和平台，生成不同的构建脚本，如Linux的Makefile，Windows的VC工程文件。



CMake的过程

第二步：编译。没啥好说的，Linux下直接 `make -jxx` ，其他的操作系统的IDE直接打开点一下build按钮即可。

非基于make的

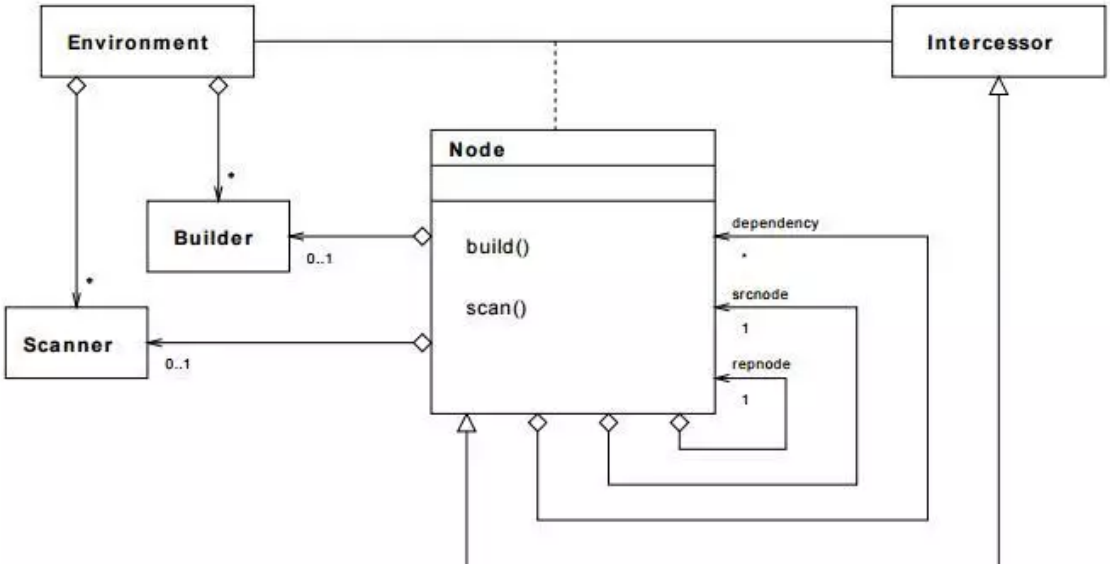
非基于make的构建系统五花八门，这里只大概介绍一下我所知的几个。

SCons

SCons 是一个开放源代码、以 Python 语言编写的下一代的程序建造工具。作为下一代的软件建造工具，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++,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。

SCons架构：



SCons架构

SCons的脚本名为SConstruct， 内容看起来是这样的：

```
Program('helloscons2', ['helloscons2.c', 'file1.c', 'file2.c'],
    LIBS = 'm',
    LIBPATH = ['/usr/lib', '/usr/local/lib'],
    CCFLAGS = '-DHELLOSCONS')
```

其中，

- LIBS： 显示的指明要在链接过程中使用的库，如果有多个库，应该把它们放在一个列表里面。这个例子里，我们使用一个称为 m 的库。
- LIBPATH： 链接库的搜索路径，多个搜索路径放在一个列表中。这个例子里，库的搜索路径是 /usr/lib 和 /usr/local/lib。
- CCFLAGS： 编译选项，可以指定需要的任意编译选项，如果有多个选项，应该放在一个列表中。这个例子里，编译选项是通过 -D 这个 gcc 的选项定义了一个宏 HELLOSCONS。

编译命令：

```
$ scons -Q
 gcc -o file1.o -c -DHELLOSCONS file1.c
 gcc -o file2.o -c -DHELLOSCONS file2.c
 gcc -o helloscons2.o -c -DHELLOSCONS helloscons2.c
 gcc -o helloscons2 helloscons2.o file1.o file2.o -L/usr/lib -L/usr/local/lib -lm
```

Waf

SCons项目小的话还好，规模一大，依赖分析速度急速下降，而且自动配置功能很弱 （跨平台构建能力不足），Waf尝试去解决SCons所暴露的问题。Waf也是基于Python的配置、编译、安装程序。主要特性：

- 构建顺序自动化：输入输出文件的构建顺序自动化识别。
- 依赖自动分析：根据文件或命令自动进行依赖分析。
- 性能：任务都是并发执行的。
- 灵活性：可以方便地通过添加新的子类创建新的命令或任务，特定构建过程中的瓶颈可以动过方法的动态重载来消除。
- 可扩展性：默认支持多种编程语言和编译器，有需求新加的也可以通过插件进行支持。
- IDE支持：Eclipse, Visual Studio and Xcode project generators (waflib/extras/)
- 文档详细：入门到深入可以阅读：《Waf Book》
- Python兼容：cPython 2.5 to 3.4, Jython 2.5, IronPython, and Pypy

一个简单的C++构建脚本wscript，先睹为快：

```
#! /usr/bin/env python
# encoding: utf-8
# Thomas Nagy, 2006-2010 (ita)

# the following two variables are used by the target "waf dist"
VERSION='0.0.1'
APPNAME='cxx_test'

# these variables are mandatory ('/' are converted automatically)
top = '.'
out = 'build'

def options(opt):
    opt.load('compiler_cxx')

def configure(conf):
    conf.load('compiler_cxx')
    conf.check(header_name='stdio.h', features='cxx cxxprogram', mandatory=False)

def build(bld):
    bld.shlib(source='a.cpp', target='mylib', vnum='9.8.7')
    bld.shlib(source='a.cpp', target='mylib2', vnum='9.8.7', cnum='9.8')
    bld.shlib(source='a.cpp', target='mylib3')
    bld.program(source='main.cpp', target='app', use='mylib')
    bld.stlib(target='foo', source='b.cpp')
```

```
    # just a test to check if the .c is compiled as c++ when no c compiler is found
    bld.program(features='cxx cxxprogram', source='main.c', target='app2')

    if bld.cmd != 'clean':
        from waflib import Logs
        bld.logger = Logs.make_logger('test.log', 'build') # just to get a clean output
        bld.check(header_name='sadlib.h', features='cxx cxxprogram', mandatory=False)
        bld.logger = None
```

Boost.Build(b2)

在编译Boost库的时候，会用到 `b2` 命令，其实就是 `Boost.Build` 的缩写。编译C++/C代码时，只需要指定要编译那些可执行文件或库，然后列出相关的源码，Boost.Build帮你搞定其他事情，支持Windows、OSX、Linux和商业的Unix系统。

HelloWorld项目的jamroot.jam脚本（Jamfiles，一种不同于Makefile的构建脚本，有兴趣自google）：

```
exe hello : hello.cpp ;
```

Boost.Build是一个高级编译系统，它能尽可能容易的管理C++项目集。其思想是在配置文件中指定编译程序的要素。例如，它不需要告诉Boost.Build如何使用某个编译器。Boost.Build支持多个编译程序，并知道如何使用它们。如果你创建一个配置文件，你只需要告诉Boost.Build在何处寻找源文件，调用哪些可执行文件，Boost.Build使用哪个编译器。然后，Boost.Build将尝试查找编译器并自动生成程序。

Boost.Build支持许多不包含任何编译器特定选项的编译器的配置文件。配置文件完全是编译器独立的。当然，可以设置选项是否应该优化代码。这些选项都是boost.build语言写的。一旦选择编译器去编译程序, Boost.Build会将配置文件中的选项翻译成相应编译器的命令行选项。这样就有可能写一次配置文件，在不同的平台上用不同的编译器构建程序。

Boost.Build只支持C++和C项目。它是为在不同平台上用不同编译器编译和安装Boost C++库而创造的。

小结

各种构建系统各有优缺点，需要深入研究和使用才能了解。没有那个是最好的，只有最适合的。一般：

- 一两个源文件的C++代码，完全没必要用构建系统，直接使用编译器命令直接搞定；
- 自己用的小项目，直接手动写Makefile即可

- 大型C++项目建议使用CMake，GNU Build System比较年龄大了，规则有些复杂，写起来没有CMake那么舒服，跨平台的话就根本没戏。
- 偶尔突破一下，想尝试一下新鲜的构建系统，SCons、Waf、B2等等，等着你玩。
- 某天感觉构建系统也不过如此，闲的无聊你也可以尝试写一个，这不500行代码搞定：http://www.aosabook.org/en/500L/contingent-a-fully-dynamic-build-system.html

参考资料

- https://www.softprayog.in/tutorials/understanding-gnu-build-system
- https://www.gnu.org/software/make/
- http://www.cmake.org/
- http://scons.org/
- http://scons.org/doc/production/PDF/scons-design.pdf
- https://waf.io
- http://www.boost.org/build/tutorial.html