BuildSystem

Table of Contents

1 BuildSystem

words from huangjun@baidu.com(leemars528)

- 内部构建系统*必须*要求所有模块都使用这个构建系统。
- 模块的依赖*不应该*通过额外的系统来管理。

- *应该*能够指定足够细致的粒度。
- 就*应该*只依赖于需要生成接口所需要的最少内容。

1.1 GYP

gyp(generate your project)是chromium的构建系统，地址在**http://code.google.com/p/gyp/**。

关于GYP和CMake的对比在**http://code.google.com/p/gyp/wiki/GypVsCMake**。

文档建设还是比较差的，并且个人使用一个很简单的例子都没有work成功。虽然wiki有UserDocumentation但是里面介绍非常粗略，基本上可以认为是一个没有成熟产品。

虽然没有比较好的使用文档，主页wiki里面还有有一些关于gyp本身比较好的描述，以及设计的初衷。 通过学习这些内容，可以对构建系统有更加深入的认识。感谢huangjun@baidu.com(leemars528)给我的建议， 他透露这个可能是gg内部的一个构建系统原型。并且之前yangliu@baidu.com(from google)给我举构建系统例子的时候， 表达方式上和gyp也非常相似，所以有理由相信gyp很像现在google内部的构建系统。

1.1.1 设计目标

gyp设计针对目标就是为了解决chromium浏览器构建问题，最重要的就是支持多平台的构建。因此生成的后端可能是 Scons/Make(Unix/Linux)，Xcode(Mac)或者是Visual Studio(Windows).并且因为chromium内部都是C/C++文件， 因此主要考虑方便C/C++程序的构建。设计时候还考虑到下面这些问题：

- debug vs. release.
- cross compile.
- toolchain interface.

1.1.2 构建文件

构建文件名字不固定，但是后缀通常是.gyp和.gypi(gyp included).构建文件内容就是python的一个数据结构(可以认为是json,不过允许#作为注释并且允许trailing的).这样做的一个方便结果就是为了读取构建文件信息，只需要eval一下文件的内容即可，就可以得到这个构建文件的描述了。

下面是一个example：

```
{
    'target_defaults':{
        'defines':['DEBUG'],
    },
    'targets':[
        {
            'target_name':'test', #生成的文件.
            'type':'executable', #可执行程序.
            'sources':['test.cc'],
            'defines':['FOO']
        }
    ]
}
```

在后面部分会详细解释构建文件里面的每个element。

1.1.3 .gyp文件剖析

整个构建文件最顶层是一个字典，包含了下面这些key：

- conditions //条件判断
- includes //包含的构建文件
- $target_{defaults}$ //构建目标默认属性
- targets //构建目标列表
- variables //构建文件使用的变量

1.1.3.1 conditions

conditions分为两种行为。普通的conditions就在load构建文件之后立即计算，另外一种是$target_{conditions}$ 是在计算完成依赖之后然后来进行计算的，两个过程分别就是early and late phases阶段。对于conditions写法非常简单：

```
'conditions':[
    ['OS==Linux',{'sources':['linux_interface.cc']}]
]
```

对于condition的判断，主要还是为了能够修改一些描述属性。从文档上来看的话，默认提供的条件就是OS判

断，其他判断应该都是变量的判断。

1.1.3.2 targets

target部分的话会对target$_{defaults里面设定的内容默认进行merge。比如上面例子的话，对于target}$/test来说， 使用的defines就会是-DDEBUG -DFOO。当然对于这种东西是可以进行其他策略选择的，比如如果修改成下面格式，那么就是直接替换：

```
'defines=':['FOO']
```

生成的defines就是-DFOO了。或者是可以剔除掉：

```
'defines!':['DEBUG']
```

生成的defines就没有任何内容了。通过在选项key后面添加操作符号来达到自定义目的的(相对于全局环境).

对于一个target包括了下面这些重要属性：

- actions(list) //执行命令
- all$_{dependentsettings}$(dict) //如果依赖这个target的话，需要使用的设置
- configurations(dict) //配置
- defines(list) //对于C/C++的defines
- dependencies(list) //依赖对象。如果是本文件的话那么直接引用，如果是其他文件的话，使用path/xxx.gyp:target.
- direct$_{dependentsettings}$(dict) //直接依赖这个taregt的话，需要使用的设置
- include$_{dirs}$(list) //头文件目录
- libraries(list) //目标需要链接的库
- link$_{settings}$(dict) //依赖这个target，需要使用的链接参数
- sources(list) //源文件
- target$_{conditions}$(list) //和conditions类似，但是是在完全计算之后然后来判断
- target$_{name}$(string) //名字
- type //目标类型，现在只是支持static$_{library}$,shared$_{library}$,executable和none

1.1.3.3 includes

gyp倾向的组织就是在toplevel上面存在一个gyp文件，可以存在子目录下面，但是子目录下面并不存放一个完整的构建文件， 通常只是存放构建文件的片段。为了区分，后缀为gypi。本身来说，这个gypi并不可以直接被gyp所接受生成native构建系统文件， 唯一的作用就是被toplevel的gyp文件进行include。如果对于Linux系统来说，最终生成的Makefile应该是一份大Makefile并且没有递归make的操作。 关于构造一个没有递归的Makefile是非常有价值的，不管是对于调试还是提升编译速度方面。可以参考文章Recusive Make Considered Harmful.

一旦我们允许include子目录的gypi文件进来，我们就必须规定哪些字段应该是文件。原因是假设存在src目录下面有src/BUILD.gypi这样一个文件， sources内容如下:

```
'sources':['src.cc']
```

而在上层BUILD.gyp文件里面，使用includes语法：

```
'includes':['./src/BUILD.gypi']
```

那么在生成大Makefile的时候，我们必须清楚'sources'字段里面内容都是文件，不应该直接使用src.cc， 相反应该加上目录前缀src，最终应该使用src/src.cc这样一个文件。关于哪些字段里面内容是路径， 这个在gyp里面有详细规定，在后面小节里面我们也会提到。

1.1.3.4 actions

actions是targets里面的一个特殊属性，主要是用来进行target的自定义操作的。关于rule的部分， 应该问问huangjun@baidu.com,因为他实际操作过gyp并且阅读过Chrmoium里面的.gyp文件。

每个action是一个dict，主要包含4个属性：

- action$_{name}$(string). //操作名称
- input(list) //输入文件
- outputs(list) //输出文件
- actions(list) //命令

有了这些属性就可以做一个完整的操作抽象。

1.1.3.5 variables

variables这个小节里面是进行变量的定义，格式是dict。下面是一个例子：

```
'variables':{
    'common_files':['src/common.cc','src/interface.cc'],
}
```

为了引用变量，我们可以这样编写：

```
<(common_files)
<@(common_files)
>(common_files)
>@(common_files)
```

总之引用变量必须加上(),同时在前面加上<,<@,>,>@的4种中一种前缀符号。关于前缀符号的含义， 会在后面的operator小节里面说明。

对于变量类型，一共分为3类：

- predefined variables //预定义变量
- user-defined variables //用户定义变量
- automatic variables //自动变量

预定义变量比如OS(系统环境),EXECUTABLE$_{SUFFIX}$(可执行文件后缀).用户自定义变量就不再赘述。

自动变量类似于Makefile里面的$@,$这样的变量，好比反射。比如在target$_{conditions部分的话}$， 我们根据不同类型程序来做不同的condition：

```
'target_conditions':[
['_type=='static_library',{'sources':['func.cc']}]
]
```

这样对于target为static$_{library都会联编func}$.cc这个文件了，自动变量是就是属性名称之前加上$_{构成的}$。

存在自动变量非常必要。有时候我们在全局环境中，希望根据不同的条件来定义不同的行为，并且是在计算的同时在来做条件判断的。 这样就提出一个要求就是，条件判断部分必须有能力知道，当前到底在计算什么东西(反射)。

1.1.4 early and late phases

对于变量展开和条件判断有两个不同的阶段：

- 载入文件之后进行，就是early phase。
- 计算完成之后进行，就是late phase。

对于两个阶段允许不同操作是非常必要的。对于early phase这个肯定需要，而对于late phase的话， 有时候我们是希望了解到gyp处理完成某个target之后所有信息，然后进行判断的。

ps:comake2在设计的时候，就没有考虑late phase这个功能。造成没有办法在应用层添加延迟计算这样一个特性。 最终只能够是修改comake2代码来完成需求。

1.1.5 operator

关于每个操作符号的含义：

- x= //字段内容进行覆盖
- x? //如果字段没有定义，那么就进行覆盖
- x+ //字段内容进行merge
- <(x) //early phase计算变量x，并且以string类型返回结果
- >(x) //late phase计算变量x，并且以string类型返回结果
- <@(x) //early phase计算变量x，并且以list类型返回结果
- >@(x) //late phase计算变量x，并且以list类型返回结果
- x! //从已有的x字段中排除部分
- x/ //操作允许使用include/exclude，内容是一个正则表达式来进行包含和排除列表里面内容
- <!(x) //认为x是一个shell command，得到执行结果作为string类型返回
- <!@(x) //认为x是一个shell command，得到执行结果作为list类型返回

1.1.6 路径内容属性

在includes这个小节提到了，gyp规定了某些属性的内容必须为路径。这些属性是：

- files.
- include_dirs.
- inputs.
- libraries.
- outputs.
- sources.

但是gyp对于里面的内容也做了一些特殊处理。对于内容来说，如果以下面这些字符开头：

- / //绝对路径
- $ //变量
- - //链接参数比如-lm
- <,>,! //operator
- 其他作为相对路径

1.1.7 总结

gyp文档缺乏导致在分析这个系统的时候，也没有完全使用只是通过阅读文档来完成的，没有一个可以run的Makefile。 同时Makefile也想当难读(尽管如此，还是稍微看了一下生成的Makefile).最好可以结合chromium源代码来看看gyp是如何使用的(时间有限,我没有做).

gyp虽然是构建系统，并且通常来说构建文件倾向于使用描述性语言来完成，但是类json风格的描述性语言很容易形成过深的嵌套不利于阅读。 可以考虑使用其他方式描述性语言来完成。如果可以的话，结合过程语言也未尝不可。

在运行时上gyp区分early和late phase两个阶段，同时允许通过target_defaults这样一个section来设定全局属性。 通过automatic variables这样的机制提供了反射功能，并且允许自定义操作，并且允许从外部shell中读取内容。 此外gyp允许提供跨模块之间的依赖管理(一个模块有如何这个模块被依赖的话，那么依赖这个模块的模块应该使用哪些属性). 这些都是一个强大构建系统所必须的。

从gyp层面就考虑了如何避免调用recursive make，通过规定某些属性内容只允许为路径名称并且允许include其他目录的.gpyi文件， 理论上可以生成不需要recurisve make的Makefile。同时在生成Makefile上面考虑了cross compile,out-of-source build问题。

1.2 Scons

地址在**http://www.scons.org/**。

总体来说算是一个比较成熟的产品，也发布过相当多的版本经过很长时间的演化。 这里是一些已经使用scons的项目列表**http://www.scons.org/wiki/SconsProjects**。 可以看到比较有名的两个就是MongnoDB和v8。

1.2.1 Make缺陷

SCons对于Make缺陷也有一个总结，链接是**http://scons.org/wiki/FromMakeToScons**。 这里稍微总结一下，在设计Make代替品BuildSystem时候需要考虑到：

- make的命令是不可移植的，也就是说，对于*nix和win32必须单独编写不同的make命令。
- make对于递归目录处理会牺牲速度。同时从目录角度来说，可能会发生重复检查的情况。
- make语法十分ugly。
- make使用环境变量，这样造成很多问题不可复现。
- make对于一些检查过于简单，造成很多时候你必须make clean然后重新make。
- make缺乏自动分析编译依赖。
- make难以调试，主要是在规则冲突方面。

客观地说除了3之外，其他都是非常严重的问题。对于3的话，要看你使用Makefile到什么程度了。 除了这些之外，如果按照Ninja的说法来说，Makefile还有一个速度过慢的缺点。

1.2.2 构建文件

scons的构建文件名称是统一的都称为SConstruct。当然为了方便目录的组织，也允许在各个目录下面存放SConscript， 然后最上面SConstruct收集这些SConscript组织成为一个大的构建文件。

和gyp所不同的是，scons并没有生成一个所在平台的构建系统文件(linux->Makefile)，而是直接解释SConstruct来进行构建。 避开native build system文件的生成在一定程度上可以简化工作量，但是自己需要来完成整个构建活动。 同时速度可能会打一些折扣(比如现在google推出的ninja的构建系统构建系统很好，而scons就不能有效利用这个工具,而gyp只需要稍加改动就支持). 个人观点来看的话，生成native build system文件可能是一个更好的选择。

下面是一个简单例子

```
Program('main.cc')
```

然后使用方式是

```
[zhangyan@tc-cm-et18.tc.baidu.com test-scons]$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o main.o -c main.cc
g++ -o main main.o
scons: done building targets.
```

使用scons默认用来构建里面所有的内容。当然也可以使用scons <target>来构建某个或者是某些target。scons -c相当于make clean,scons -Q的话可以安静地构建，只是打印出真正执行的命令。

如果是需要编译多个文件的话,可以使用python的列表格式。同时这些scons内置函数也是支持python的keyword argument这样的形式的：

```
Program(target='hello',source=['main.cc','hello.cc'])
```

然后我们执行

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

我们可以看到，在这个SConstruct这个Python Script里面Program等东西都实现成为了scons的内置函数，然后在表达上文件列表就是使用python列表形式。这样在一定层面上这就让用户接触到了[]这样的格式， 用户可能就会意识到当前可能是一个脚本，而不是以一种透明方式展现出来。不过通常来说,这样问题并不是很大，用户也能够忍受列表以及keyword argument这样的东西，同时对于高级用户(熟悉python)的人也感到比较亲切。

同时我们必须注意到，scons在构建目录下面存放了一个.sconsign.dblite这样一个文件。这个文件应该是 使用sqlite这样关系数据库的。因为scons很多特性需要依赖存储数据，比如cache隐式依赖，记录文件的md5等等信息， 单独使用一个辅助文件存放信息是一个不错的选择。

1.2.3 SConstruct文件剖析

1.2.3.1 Builder

SConstruct主要支持的开发语言有C/C++以及Java。对于Java有专门的Builder，这里我们不谈.对于C/C++的Buidler有下面这几种：

- Program //可执行程序
- StaticLibrary //静态库
- SharedLibrary //动态库
- Object //目标文件

当然为了完备，还提供了一个万能的Buidler就是Command。对于Command来说的话，使用方式如下：

```
Command('hello',['main.cc','hello.cc'],'g++ -o $TARGET $SOURCES')
```

撇开具体使用方法我们不说，对于自定义命令来说，我们最主要关心三个方面：

- target
- dependencies
- commands

在Command这个Builder很完整。同时为了方便还允许用户使用automatic variables，比如$TARGET和$SOURCES这两个变量， 类似于Makefile里面的$@和$。关于Command讨论就到这里，接下来看看上面三个主要的Builder.

Builder。

1.2.3.1.1 编译参数和自动分析编译依赖

默认情况下面如果这样写的话，那么是不能够制定任何编译参数的：

```
Program('main.cc')
```

要不使用下一节方法一样构建一个环境，要不就需要显示地构造Object对象：

```
obj=Object('main.cc',CCFLAGS='-DDEBUG')
Program('main',obj)
```

默认情况下面是不会自动分析编译依赖的，也就是说,如果main.h头文件发生变化的话，是不会发生重新构建的。 为了强制能够自动分析编译依赖，需要显示写明CPPPATH，就是include path：

```
obj=Object('main.cc',CCFLAGS='-DDEBUG',CPPPATH=['.'])
Program('main',obj)
Decider('make')
```

Decider的意义是说按照时间戳更新的方式来检查，这个在检查依赖这节会提到。在做法上，个人猜测scons 也是使用g++ -MM -MG这样的方式来分析编译依赖。通常来说耗时会比较长。因为scons允许在调用参数中缓存上次编译依赖， 强制刷新编译依赖等，这样可以在耗时和功能上达到一个比较好的折衷。

1.2.3.1.2 源文件使用不同参数编译

假设对于

```
env1=Environment(CCFLAGS='-DDEBUG')
env1.Program(target='debug_main',source=['main.cc'],)
env2=Environment()
env2.Program(target='release_main',source=['main.cc'])
```

关于Environment会在后面讲到，因为scons对于main.cc直接生成main.o这样文件，如果main.cc使用不同参数来进行编译的话如上， 那么就可能出错。scons可以检测到对于同一个target使用不同参数编译，那么在执行时：

```
[zhangyan@tc-cm-et18.tc.baidu.com test-scons]$ scons
scons: Reading SConscript files ...

scons: *** Two environments with different actions were specified for the same target: mai
File "/home/zhangyan/test-scons/SConstruct", line 4, in <module>
```

为了解决这个问题，可以使用Object这样的Builder来指定产生不同的文件名：

```
env1=Environment(CCFLAGS='-DDEBUG')
obj1=env1.Object(target='debug_main',source=['main.cc'])
env1.Program(obj1)
env2=Environment()
obj2=env2.Object(target='release_main',source=['main.cc'])
env2.Program(obj2)
```

执行之后

```
[zhangyan@tc-cm-et18.tc.baidu.com test-scons]$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o debug_main.o -c -DDEBUG main.cc
g++ -o debug_main debug_main.o
g++ -o release_main.o -c main.cc
g++ -o release_main release_main.o
scons: done building targets.
```

1.2.3.1.3 库链接

可执行程序需要进行库链接，使用方式也是非常简单。Program这个Builder下面提供LIBPATH和LIBS这两个keyword arguments， 可以用来指定所需要链接的库。

这样来写链接库首先是比较符合用户习惯的，但是这样会造成一个问题，这个在comake2中出现过，也是导致comake2并不提倡这个做法的原因， 那就是这样写的话不好构建依赖。假设下面两种写法：

```
Program('main.cc',LIBS='./lib/libcm.a')
Program('main.cc',LIBS='cm',LIBPATH=['./lib'])
```

很明显第1中方式不仅仅写了所链接的库，scons本身还可以从中知道依赖什么文件，而comake2的做法就是希望用户显式依赖某个库。

但是在scons中绕过了这个问题，是因为scons还有两个函数用来显示声明依赖的。对于显示声明依赖，个人看法是这个东西必须做为一个可选项， 对于小型程序显然用户希望构建系统自己生成依赖，而对于大型程序，显示声明依赖可以减少分析时间提高构建效率。

1.2.3.2 Dependencies

对于scons，除了在Builder里面提到的implicit dependencies这种隐式依赖，其他的依赖都必须显示说明

对于scons，除了在Builder里面提到的implicit dependencies这种隐式依赖，其他的依赖都必须显示说明的，或者是从指定的文件中进行分析。

1.2.3.2.1 分析依赖

关于依赖共有两种类型，一种是普通的依赖就是A->B，另外一种是顺序依赖。顺序依赖的意思是说，其实A->B并不是真正构建A依赖于构建B。 只不过我们希望在构建顺序上，B在A构建之前。如果B发生变化，A不一定发生构建。这两种依赖分别对应于scons里面的Depends和Requires函数， 当然这个在Makefile里面也有方法表现出来。scons还提供一个函数ParseDepends可以分析g++ -MM -MG产生的.d文件，然后来判断依赖。

1.2.3.2.2 判断依赖变化

判断依赖变化上，scons允许自己定义函数来判断文件是否发生变化。内置判断函数有下面几种：

- Decider('make') //依赖对象修改时间是否比target修改时间大
- Decider('timestamp-match') //依赖对象修改时间相对上次来说没有改变
- Decider('MD5') //依赖对象内容的md5没有发生改变
- 或者是以上这些判断方式的组合

同样依赖变化还包括构建方式的参数变化，这点是值得学习的。实现起来也不是很难，可以在数据库中记录前一个构建某个 target的输入文件以及构建参数，如果构建参数发生变化的话，那么重新构建并且在数据库中进行记录。

1.2.3.3 Environment

对于scons来说存在这么几个环境：

- External Environment //外部环境，可以通过import os获得
- Construction Environment //创建环境，我们可以在SConstruct里面填写
- Execution Environment //执行环境，scons在执行执行时所拥有的

和make有点不同的是，scons有意不将External Environment导入到Construction Environment里面来。理由简单也非常合理，那就是如果一旦将环境变量也纳入scons的逻辑的话，那么构建过程是不可重复的。 可就是说，在某个人机器上搞得定但是另外一个机器上搞不定仅仅是因为环境变量不同。

默认情况下面scons有一个默认的环境DefaultEnvironment()可以获得。然后我们可以在一个环境下面定义Builder等内容。 同时scons针对环境这个对象有相当多的操作，比如Clone,Repliace,Merge等。

1.2.3.4 Hierarchical Builds

层级目录之间的构建，在一开始也说过了，是通过导入其他目录下面的SConscript这样的文件来完成的， 最终也是在top-level directory上面发生构建。和gyp一样，SConscript也需要处理这样的问题， 就是定义哪些元素是file和directory,然后在解释的时候必须加上目录前缀。

此外，相对于gyp,scons还引入了变量的export和import机制。某个SConscript可以export一部分变量出来， 然后另一个SConscript文件可以import这个变量进去。实现方式也非常简单，就是scons假设存在一个全局的变量池 (variable pool),组织形式是一个字典。然后export就是写入这个字典，而import就是从这个字典导入内容。 跨文件之间的变量共享是非常必要的，这个在comake2里面共享编译依赖(已经做到),并且还有很强烈的需求就是共享编译参数(这点没有做到).

1.2.4 其他问题

scons提供了install方法，允许把某个文件安装到某个目录下面去，以相同或者是不同的名字安装。

scons提供了平台无关的文件操作函数，比如copy,delete,rename,mkdir,touch等。这些函数做成为一个延迟函数对象. 有一个Execute函数可以接受这些对象立即执行。其实关于这文件操作函数，个人认为可能完全没有必要，平台无关是非常好的概念， 但是对于文件系统实在是难于抽象出来。其他不说，首先考虑文件系统的名称就够费神半天，个人觉得把大部分细节屏蔽掉就ok。 对于需要平台相关操作,那就平台相关命令吧，scons对于命令操作也包装成了一个对象，这样可以延迟执行。

scons支持out-of-source的编译方式，实现方式是通过copy源代码到另外一个地方然后进行编译，关于这么做的原因在下面地址也说了。 个人感觉，可能out-of-source的编译也不一定需要使用这种方式来完成(Android Build System就是out-of-source编译的,但是没有copy source). **http://www.scons.org/doc/2.1.0.alpha.20101125/HTML/scons-user/c3348.html**

scons支持configure这样的功能，包括：

- 检查头文件是否存在
- 检查库是否存在
- 检查typedef

- 检查函数是否存在

当然这只是配置中一部分。阅读了MongnoDB的SConstruct这个文件，发现其实还有相当多的检测是需要自己来完成的， 同时可能不希望按照scons内置的configure规则来进行检查。对于configure这样一个东西是非常需要，但是相当难做。 GNU Autoconf个人感觉来说相对更加成熟一些，如果一定要做configure这样功能的话，可以首先参考一下GNU Autoconf.

scons还支持cache编译产出物这样的功能，不过个人还是觉得没有必要这么做，因为这样在scons内部实现了一个cache系统， 不仅仅加重了工作量，而且并没有做好，因为相对ccache来说，scons提供过于简单了。至于cache效率的话，是另外一个问题。 关于cache系统确实完全独立实现并且通用。

1.2.5 总结

scons是一个文档相对来说比较完整，已经被一些大型项目所使用的，比较成熟和完善的产品。

scons本身使用Python Script来做描述文件，相对于gyp来说可读性会更好，表达功能上相对于gyp也更加强大， 但是容易造成一个问题就是速度会过慢，尤其对于scons这种执行方式。因为scons每次都是读取SConstruct/SConscript文件， 而不像gyp一样一次就生成Makefile，大部分时候是不需要变动Makefile，只是需要make的。

scons对于编译依赖来说，提供了隐式分析和显示说明两种方式来描述依赖，然后针对隐式分析编译依赖也做了一些有效的优化。 comake2在这点上也是提供了隐式分析和显示分析，但是却没有做一些优化，这样就造成每次生成Makefile时间过长。 如果是大型项目，耗时会非常长，在这些情况下面显示编写依赖可能会更好。对于依赖变化检测，也是提供了很多种方法。 并且如果构建参数发生变化，也是认为依赖发生变化，重新构建的。

scons对于构建系统构建方式，和gyp一样认为构建必须从top-level开始进行。 和gyp一样，允许每个子目录下面存放SConscript描述文件，然后在上层最终收集上来。

scons提供了环境这个概念，这点对于构建不同参数的版本作用非常大。并且scons对于环境这个概念引入也非常自然， 直接env.Program(..)和没有使用环境的Program(..)在编写内容上相差无几，同时也提供了很多关于环境的操作，并且提供一个默认环境。

和其他构建系统一样，scons还提供了平台无关文件操作，install方法，out-of-source编译，configure，以及编译产物的cache， 不过个人看来其实每个部分要不就做得相当粗糙(out-of-source编译,configure)，要不就是鸡肋(文件操作，编译产物cache，install)， 当然也可能是因为自己对于scons了解不够深入引起的。

1.3 Ninja

地址在**https://github.com/martine/ninja/blob/master/manual.asciidoc**。

ninja是也是为了加速Chromium的编译(gyp也是，不过gyp提供的是一个前端，而ninja是实际描述文件。 现在已经有一个patch for gyp来生成ninja文件了)。定位非常清晰，就是达到更快的构建速度。对比的对象是没有任何递归的Makefile。 作者给出了一个实测数据,30000个文件，make使用10seconds才开始进行构建，ninja只用了1s中就开始进行构建了。

ninja的设计是对于make的缺陷的考虑，认为make有下面几点造成编译速度过慢：

- 隐式规则，make包含很多默认推导规则。
- 变量计算，比如编译参数应该如何计算出来。
- 依赖对象计算。

ninja认为描述文件应该是这样的：

- 依赖必须显式写明(为了方便可以产生依赖描述文件)
- 没有任何变量计算
- 没有默认规则，没有任何默认值

针对这点所以基本上可以认为ninja就是make的最最精简版。

但是ninja相对于make增加了下面这些功能，可以看到都非常有用，并且这些在上层是不能够完成的，只有在这层做：

- 如果构建命令发生变化，那么这个构建也会重新执行。
- 所依赖的目录在构建之前都已经创建了，因为如果不是这样的话，我们执行命令之前都要去生成目录。
- 每条构建规则，除了执行命令之外，还允许有一个描述，真正执行打印这个描述而不是实际执行命令。
- 每条规则的输出都是buffered的，也就是说并行编译，输出内容不会被搅和在一起。

ninja也支持order dependencies，还有一系列辅助工具比如查询某个target的依赖以及构建规则，打印出整个构建图等功能。 关于ninja文档里面没有提到如何判断依赖变化，但是如果目的是提高速度的话选用方式应该还是时间戳(学习scons之后,考虑一下时间戳的匹配可能会更快)。

1.4 ABS

ABS(Android Build System).关于这个BuildSystem的文档可以在android的源代码的build/core/build-system.html里面找到。 ABS是虽然是使用make的，但是做了相当多的规范。首先看看ABS所规定的原则和策略有哪些：

- 支持多平台目标
- 使用non-recursive make
- 支持环境和配置的设置
- out-of-source的编译方式
- 依赖能够自动分析出来
- 隐藏执行命令输出
- 允许使用GLOB语法来包含文件
- 允许在一个directory里面编译多个targets
- 允许每个目录有各自描述文件

希望以后支持下面这些特性：

- 对于不同的平台可以同时编译
- 如果头文件删除,那么依赖可以检测到
- 能够自动进行所有平台构建

之前做过关于ABS的编译优化，稍微看过一下ABS是如何组织的。大致组织方式是在：

- 在top-level包含Makefile。内容很简单，就是include build/core/main.mk。
- 在build/core/main.mk里面include了每个目录下面的.mk文件。
- 然后在build/core下面系统一系列的辅助.mk文件，比如executable.mk(编译可执行程序)，均被包含到main.mk里面。
- 对于在每个子目录下面的Android.mk文件里面，只需要负责设置一些变量。
- 然后在make时候，子目录变量收集起来，配合辅助.mk文件，生成一系列规则。
- 然后进行构建。

1.5 CMake

下面是CMake的有用链接：

- CMake主页，**http://www.cmake.org/**
- 为什么KDE会从SCons切换到CMak，.**http://lwn.net/Articles/188693/**
- 台湾人写的CMake overview.ppt风格不错，**http://wenku.baidu.com/view/e27778db6f1aff00bed51e60.html**
- CMake-behind scenes of code development，**http://wenku.baidu.com/view/c1dbc3eb998fcc22bcd10dff.html**
- cmake –help-html cmake-help.html，可以查看CMake的内置HTML帮助页，难读但是更像是一个Specification。

cmake的全称是cross-platform makefile generator，但是猜想后来后端很多但是名字保留下来了。 现在的定位应该是cross-platform build system generator，通过读取源代码目录下面的CMakeLists.txt文件， 然后进行configure同时生成native build system所需要的构建文件。在Unix下面主要生成：

- unix makefile.
- code blocks - unix makefile.
- eclipse cdt4 - unix makefile.
- kdevelop3 - unix makefile.

cmake相对于其他构建工具优越的几点有：

- C/C++的头文件自动分析

- 生成Makefile进行make时候比较漂亮，能够显示编译进度
- 生成Makefile的target比较丰富
- cmake可以配合其他工具进行无缝集成，比如CTest/CDash
- KDE/OepnCV已经使用了CMake，对于质量比较有保证

1.5.1 构建文件

编写一个简单的CMakeLists.txt非常简单：

```
project(main)
add_executable(main main.cc)
```

然后进行configure并且同时生成Makefile：

```
[zhangyan@tc-cm-et18.tc.baidu.com test-cmake]$ cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /home/zhangyan/utils/bin/gcc
-- Check for working C compiler: /home/zhangyan/utils/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zhangyan/test-cmake
```

输入参数指定source位置，这样来进行out-of-source的构建。cmake在第一次分析CMakeLists.txt的时候，会对环境进行分析然后将一系列得到的环境内容保存在CMakeCache.txt文件里面，得到一系列的变量，相当于配置的选项。 然后外部允许通过自己指定来改写这些默认得到的环境，但是这个环境依然不能够被source所使用，比较好地可以被程序使用的方式， 就是像GNU Build System一样生成的config.h。关于如何生成config.h，会在特定的节里面提到。

```
[zhangyan@tc-cm-et18.tc.baidu.com test-cmake]$ make
[100%] Building CXX object CMakeFiles/main.dir/main.cc.o
Linking CXX executable main
[100%] Built target main
```

cmake本身也提供了类似于gyp的功能,将编译依赖输出graphviz的图,方便查看.

1.5.2 CMakeLists.txt解析

cmake里面提供的描述语言，内置了函数，宏，条件判断，流程控制，同时还有变量(不清楚是否有嵌套作用域)， 还有一些简单的数据结构比如整数字符串以及列表，相当的完整。

关于CMake有相当多的语法和属性，这里就不一一讲解，只是谈谈相对于其他构件系统比较独特的地方。

add$_{customcommand}$允许用户进行自定义的命，通过指定output,input,command，同时允许指定这些命令在：

- dependencies构建之前执行
- dependencies构建之后，但是target构建之前执行
- target构建之后执行

add$_{test}$允许全局环境增加test用例，允许自己使用test的启动命令。

build$_{command}$能够获得某个target的构建命令。

file定义了一系列文件操作方法，为多平台的文件操作提供了抽象。

find提供了一系列查找文件，库，package，program的方法，简化配置的生成。 对于find$_{package}$内置很多检测库的方法，并且有大量的内置变量可以被使用。

set$_{property}$/get$_{property}$设置和获取属性。

include包含另外一个文件进来，语义和Make的include是一样的。而add$_{subdirectory}$添加一个子目录，对应的Makefile就是"make -C ..."。 从效果上说，cmake并没有提供类似于gyp和scons的import功能来方便进行项目的整体构建，还是将一个个目录单独进行构建。

1.5.3 变量和属性

变量和属性其实是一个概念，我们使用$<...>来进行变量引用，而使用set来设置变量。对于里面的内容，语法如下：

```
OPTION:target
```

引用target下面的某个OPTION。这些就可以针对每个target来进行一些参数定制了。比如：

```
SET(cppflags:libecho.a,"-DDEBUG -W -Wall")
```

1.5.4 注释

注释也作为属性存在于target或者是全局环境中.对于target的注释来说，在进行build的时候，cmake能够让注释在执行命令之前就显示出来。 这样一方面可以利于开发者来进行调试，同时产生的输出信息对于用户也更加友好。

1.5.5 编译类型

cmake的编译类型一共分为4类，可以简化编译选项的设置：

- Debug //调试版本
- Release //发布版本
- RelWithDebInfo //发布版本但是带上调试信息
- MinSizeRel //发布版本但是简化二进制大小

可以通过-DCMAKE$_{BUULDTYPE}$=...在命令行里面指定.使用一个编译类型会配置上一系列的编译选项，用户可以在内部继续修改这些选项。

写到这里，关于选项这种东西，有一些想法。在设计选项方面，我们必须考虑下面这几个功能：

- 选项必须进行分配，然后可以进行批量开启。
- 允许用户在描述文件中覆盖和增加选项。
- 允许用户在命令行也指定这些选项。

这样在易用，功能方面能够达到比较好的折衷。

1.5.6 生成config.h

cmake生成config.h的方法也是套用模板的方式。里面提供了一个函数，可以将类似于：

```
#define JPEG_FOUND %{jpeg_found_value}d
```

然后在cmake运行时环境里面存在jpeg$_{foundvalue这样变量进行替换，然后生成config}$.h。 非常不错的idea(猜想GNU Build System应该也是这样完成的).

1.5.7 Makefile

生成的Makefile，包含下面这样几个大的target：

- make //默认构建
- make <target> //构建某个target
- make clean //删除中间构建结果
- make depend //重新计算依赖
- make <source>.ii / <source>.s // <source>.o //生成中间文件

cmake能够自动分析C/C++的编译依赖。同时如果CMakeLists.txt发生变化的话，那么也会重新生成Makefile。 因为时间有限，对于生成的Makefile也没有仔细分析。

1.5.8 总结

是一个相当成熟的项目，并且有一系列的外围和cmake形成一个整体，内置了相当多的辅助函数来帮助使用其他开源的库， 所以在设计和实用性上可以看得出来是相当严肃的。

因为语法是自己定义的语言，所以学习起来有一定的成本。在描述能力上来说还是不错的， 但是缺乏一些高级的数据结构比如字典可能会造成某些情况下面使用比较麻烦。

同时cmake还提供一系列检测环境的能力，然后通过生成config.h来进行source file configuration。

最后生成的Makefile在构建时候，用户友好方面不错。并且在CMakeLists.txt变化以及源文件依赖的.h文件变化的话， cmake会重新生成Makefile并且进行编译的。不过因为时间有限也没有对Makefile进行仔细的分析。

1.6 Others

关于BuildSystem一些有用的链接，在这里给出：

- Build Management Entry

http://www.dmoz.org/Computers/Software/Build_Management/ 这个是关于一个构建管理项目的入口，里面有很多解决构建项目遇到问题的工具和方案。

- Make Alternatives

http://freshmeat.net/articles/make-alternatives Make的代替品。

- A.A.P

http://www.a-a-p.org/tools_build.html 这是VIM作者编写的一个构建系统框架，是一个可以扩展的系统。

- Boost.Build

http://www.boost.org/doc/tools/build/index.html http://www.boost.org/boost-build2/index.html Boost使用的BuildSystem，也是旨在解决C++跨平台构建问题。

- Makeit

http://www.dscpl.com.au/projects/makeit/ 时间限制，没有仔细研究。

- Waf

http://code.google.com/p/waf/ 基本上可以认为是和SCons的一个sibling，似乎SCons和Waf都是从某个项目分支出来的。 语法上都采用了Python Script解决方案，但是Waf看上去似乎没有SCons简单。

- SDS

http://sds.sourceforge.net/index.html 时间限制，没有仔细研究。

- MSBuild

http://msdn.microsoft.com/zh-cn/library/dd393574.aspx M$的构建系统，时间限制没有仔细研究，并且对于Windows下面的开发环境也不是很熟悉。 唯一了解就是MSBuild也使用XML来进行描述。

- Maven

http://maven.apache.org/ Java项目开发最常使用的构建系统工具之一。Maven描述方法是使用XML来描述，虽然也可以描述一些过程化的内容， 但是感觉就是没有Python Script简洁和清晰。Maven相对于之前这些构建系统，有下面几点重大改进的地方：

- 规范项目目录结构
- 引入Maven Source Code Repository概念

规范目录结构使得大家开发Maven很容易，而Maven Source Code Repository使得协作开发非常简单， 只要配置好Source Code Repo，然后Maven能够自动下载编译所需要的依赖模块并且进行编译，能够很好地进行多模块之间的持续集成开发。

1.7 个人感觉

关于构建系统，个人感觉这里是一个大坑，想要满足所有人的要求是非常困难的。对于构建系统，我们希望有下面这些点(一部分是我觉得应该这样做)：

- 构建文件表达能力够强，使用Python Script，可以借鉴SCons这样的语法。
- 生成native build system文件，一方面可以提高速度，另外一方面可以简化工作量如cmake。
- 提供丰富变量，包括编译时和运行时。只有提供了这些变量，才能够提供更加灵活的功能如cmake。
- 允许跨文件之间的export和import，比如使用gyp这样的方式而不是使用scons这样的方法(比较笨拙)。
- 如果是生成Makefile，避开生成递归的Makefile，比如scons和gyp所提倡的，而cmake似乎没有这样做。
- 允许out-of-source编译，并且避开scons这样的copy整个源代码的实现方式如cmake。
- 允许自动分析编译依赖，并且是在make的时候自动分析，而不是在生成Makefile就写死如cmake。

- 支持检测编译选项的改变，如果编译选项改变自动发生构建如ninja和scons.
- 支持多种检测文件变化方式如scons。
- 如果有定位为跨平台的构建系统的话，需要提供平台无关的文件操作，但是个人觉得会费神。
- 提供configure的功能吗，像cmake一样提供很多内置的configure功能，但允许scons一样定制。

还有很多其他的点，一时总结的不是很完全，以后有空会继续补充(TODO)。

考虑到BuildSystem是一个相当大的坑，其实我们完全没有必要自己去完成一个构建系统，最坏情况下面使用Makefile或者是ninja就OK了。 对于baidu面临最大的问题，就是需要提供一种类似于maven的source coderepository机制，允许下载和编译所需要的依赖， 同时可以从source code中知道这个模块的依赖来满足基于主干的开发。而在底层构建的话，选用SCons或者是CMake都没有太大问题。

如果仅仅考虑上面这个问题，我们完全可以这样做：

- 对于每个模块的最外层，提供一个COMAKE文件
- COMAKE文件里面只是写明CONFIGS这样的东西。
- 然后comake2 -U & comake2 -B可以帮助用户下载和搭建环境。
- comake2能够生成一份构建系统辅助文件。

比如我们需要使用public/ub的话,我们在COMAKE里面写好

```
WORKROOT('http://www.cnblogs.com/../')
CONFIGS('public/ub')
```

然后生成的COMAKE.mk如下

```
PUBLIC_UB_PATH="http://www.cnblogs.com/../public/ub"
PUBLIC_UB_INCLUDE_CFLAGS="-Ihttp://www.cnblogs.com/../public/ub/output/include"
PUBLIC_UB_INCLUDE_CXXFLAGS="-Ihttp://www.cnblogs.com/../public/ub/output/include"
PUBLIC_UB_LIBS="http://www.cnblogs.com/../public/ub/output/lib/libub.a"
```

然后在用户自己的Makefile里面通过include来使用这些变量：

```
include "COMAKE.mk"
```

Author: zhangyan04@baidu.com

Date: 2011-05-26 22:09:00 CST