gyp的三大块:变量,条件,目标,另外是默认目标设置及include

条件指定不同的target\_defaults或target或variables等项.这里是指向target\_defaults,同时也可以在targets下使用条件,如指定不同平台的不同源文件,

当然也可以文件名上带上\_linux/\_mac/\_posix/\_win,这样gyp会自动区分平台

对于非target\_defaults的,在targets下单独指定,如'defines'项

gyp中不能用~标示主目录

{

'include':['../common.gypi',], #引用公共的头

'variables':{'library':'static\_library',}, #变量字典

'link\_settings': { #连接方面的参数

'libraries': [

'-lm',

'-lshared\_stuff', # Merged, list item appended

],

'library\_dirs': ['/usr/lib'],#库目录

'conditions': [ #条件列表

['os=="linux"', {'target\_defaults':{'cflags':['-O2','-g'],'ldflags':['-pthread',],'defines':[]},}]

['os=="win"', {'target\_defaults':{'defines':['OW\_WIN','UNICODE']},}]

[ {}] #else

],

'targets': [

{'target\_name':'xxx','type':'static\_library', #每个目标

'include\_dirs':['..',],

'dependencies':['base.gyp:base',"../zlib/zlib.pyp:zlib'],

'libraries':['pthread'],

'sources':['../net/abc.h','../sys/efg.cc'],

}

{'target\_name':'test','type':'executable', #第二个目标

'msvs\_guid': 'D8EC380E-CD33-4634-AE71-8A181A433DC9'#对于window的目标需要这个

...

'dependencies':['xxx','../testing/gtest.gyp:gtestmain',] #依赖本gyp的xxx及外面的gtest的gtestmain

}

]

}

------targets下不常用项---------------

'direct\_dependent\_settings': { #应用到依赖这个目标的其他组件上的dependents上.

'defines': [

'DEFINE\_FOO',

'DEFINE\_ADDITIONAL',

],

'linkflags': [ ],

},

'export\_dependent\_settings': ['../bar/bar.gyp:bar',],

'sources!':[aaa.cc'] #排除某个源文件

'actions': [ #定制编译

{

'variables': {

'core\_library\_files': [

'src/runtime.js',

'src/v8natives.js',

'src/macros.py',

],

},

'action\_name': 'js2c',

'inputs': [

'tools/js2c.py',

'<@(core\_library\_files)',

],

'outputs': [

'<(INTERMEDIATE\_DIR)/libraries.cc',

'<(INTERMEDIATE\_DIR)/libraries-empty.cc',

],

'action': 'python tools/js2c.py <(\_outputs) CORE <(core\_library\_files)',

},

],

<(INTERMEDIATE\_DIR) #使用INTERMEDIATE\_DIR变量

=======================http://www.cnblogs.com/dirlt/archive/2011/05/29/2061937.html===================

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

targetdefaults //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targetconditions 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targetdefaults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(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

alldependentsettings(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.

directdependentsettings(dict) //直接依赖这个taregt的话，需要使用的设置

includedirs(list) //头文件目录

libraries(list) //目标需要链接的库

linksettings(dict) //依赖这个target，需要使用的链接参数

sources(list) //源文件

targetconditions(list) //和conditions类似，但是是在完全计算之后然后来判断

targetname(string) //名字

type //目标类型，现在只是支持staticlibrary,sharedlibrary,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：

actionname(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),EXECUTABLESUFFIX(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targetconditions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staticlibrary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.

includedirs.

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targetdefaults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