Gyp语法规则参考 & 工具的使用

by Markdown, 8/14/2014 10:17:28 AM

翻译自 https://code.google.com/p/gyp/wiki/GypLanguageSpecification

目的和背景

Google使用过很多处理平台无关的项目构建系统，比如 Scons ， CMake 。在实际使用中这些并不能满足需求。开发复杂的应用程序时，在Mac上Xcode更加适合，而Windows上Visual Studio更是无二之选。
gyp是为Chromium项目创建的项目生成工具，可以从平台无关的配置生成平台相关的Visual Studio、Xcode、Makefile的项目文件。这样一来我们就不需要花额外的时间处理每个平台不同的项目配置以及项目之间的依赖关系。

概述

Gyp大致有如下的特点：

- 配置文件都是以*.gyp*结尾
- 每个*.gyp*文件都描述了如何去构建项目
- 每个*.gyp*文件都能生成若干个合适特定平台的项目文件:
  - 在Mac上，*.gyp*文件会生成Xcode项目文件，包含了如何构建项目。一个*.gyp*文件会生成的一个.xcodeproj
  - 在Windows上，*.gyp*文件会生成Visual Studio项目文件。一个*.gyp*文件会生成一个.sln，并为每个targets生成一个.vcproj
  - 在Linux上，*.gyp*文件会生成Makefile文件
- *.gyp*文件的语法是Python数据格式(Json格式)
- *.gypi*文件可以被*.gyp*文件包含，方便统一设置
- 配置中数据是键-值对的形式

详细设计

设计准则：

- 关键字一致性：所有的关键字都和平台项目的项目配置字段相同
- 通过前缀表明配置属于的特定平台。比如： `msvs_disabled_warnings` , `xcode_framework_dirs`

样例

```
{  'target_defaults': {    'defines': [      'U_STATIC_IMPLEMENTATION',
['LOGFILE', 'foo.log',],    ],    'include_dirs': [      '..',    ],  },
'targets': [    {      'target_name': 'foo',      'type': 'static_library',
'sources': [        'foo/src/foo.cc',        'foo/src/foo_main.cc',      ],
'include_dirs': [          'foo',          'foo/include',      ],      'conditions':
[          [ 'OS==mac', { 'sources': [ 'platform_test_mac.mm' ] } ]      ],
'direct_dependent_settings': {        'defines': [          'UNIT_TEST',        ],
'include_dirs': [          'foo',          'foo/include',        ],      },    },
],}
```

结构元素

*.gyp*文件中定义了一些targets和构建规则。

下面的关键字可以定义在最顶层：

- conditions: 条件定义。见 *****条件(conditions)*****

- includes: 包含.*gyp*文件的列表

```
{
'targets':[
{
'target_name': 'Thread',
'type': 'executable',
'includes': [
'../common.gypi',
'./thread.gypi',
]
...
}
],
}
```

- target_defaults: 默认的项目配置，每个项目(targets)的配置都需要从这个配置继承
- targets: 项目列表

```
{
'targets': [
{
'target_name': 'hello1',
'product_extension': 'stuff',
'type': 'executable',
'sources': [
'hello.c',
],
},
{
'target_name': 'hello2',
'target_extension': 'stuff',
'type': 'executable',
'sources': [
'hello.c',
],
}
]
}
```

- variables: 定义了键值对，可以被其他地方以 `<(varname)` 的方式引用

```
{
```

```
'variables': {

'pi': 'import math; print math.pi',

'third_letters': "<(other_letters)HIJK",

'letters_list': 'ABCD',

'other_letters': '<(letters_list)EFG',

'check_included': '<(included_variable)',

'check_lists': [

'<(included_variable)',

'<(third_letters)',],

'check_int': 5,

'check_str_int': '6',

'check_list_int': [

7,

'8',

9,],

'not_int_1': ' 10',

'not_int_2': '11 ',

'not_int_3': '012',

'not_int_4': '13.0',

'not_int_5': '+14',

'negative_int': '-15',

'zero_int': '0',

},

...

}
```

项目(targets)

*.gyp*文件定义的一套构建项目的规则。targets中也可以包含 `includes` 、 `conditions` 和 `variables` 。下面的是一些targets中的专有字段：

- target_name: 指定定义的target的名称
- type: target的类型。支持 `executable` 、 `static_library` 、 `shared_library` 和 `none` 。其中 `none` 类型也很有用，可以作为处理资源、文档等特别项目的类型。例如，在Windows中，`none` 将作为工具集类型的项目存在
- product_extension: 指定target生成目标的扩展名，不包含'.'
- product_name: 指定tareget生成目标的文件名，与 `product_extension` 组成一个文件全名
- dependencies: 指定target依赖的其他target

```
{

'targets': [

{

'target_name': 'a',

'type': 'static_library',
```

```
'dependencies': [
'b/b.gyp:b',
'c/c.gyp:*'
],
'sources': [
'a.c',
],
},
],
}
```

- defines: 定义了预处理宏。类似于C/C++命令行编译中的-D或/D选项

```
{
'targets': [
{
'target_name': 'defines',
'type': 'executable',
'defines': [
'FOO',
'VALUE=1',
'PAREN_VALUE=(1+2+3)',
'HASH_VALUE="a#1"',
],
'sources': [
'defines.c',
],
},
]
}
```

- include_dirs: 指定了包含文件的查找目录。类似于C/C++命令行编译中的-I或/I选项

```
{
'targets': [
{
'target_name': 'includes',
'type': 'executable',
'dependencies': [
'subdir/subdir_includes.gyp:subdir_includes',
],
```

```
'cflags': [

'-Ishadow1',

],

'include_dirs': [

'.',

'inc1',

'shadow2',

'subdir/inc2',

],

'sources': [

'includes.c',

],

},

],

}
```

- sources: 列出了项目中的代码文件和一些项目相关的文件。 `sources!` 段中可以指定被排除的文件
- configurations: 为targets定义的一套构建配置。见 *****配置(configurations)*****
- link_settings: 指定target需要链接的库。 `executable` 和 `shared_library` 类型的target需要指定链接库。这个段内可以指定target中可包含的除了 `configurations` 、 `target_name` 和 `type` 的所有配置。可以与 `all_dependent_setting` 、 `direct_dependent_setting` 做对比
- direct_dependent_settings: 指定依赖本target的target设置。这个段内可以指定target中可包含的除了 `configurations` 、 `target_name` 和 `type` 的所有配置。可以与 `all_dependent_setting` 、 `link_settings` 做对比
- all_dependent_settings:
- libraries: 指定target依赖的库，见 `link_settings>libraries`

```
...

'link_settings': {

'libraries': [

'libiconv.dylib',

'<(ZLIB_DIR)contrib/minizip/libminizip.a',

'libcurl.dylib',],

},

...
```

- actions: 针对输入的文件，定义了一组自定义的构建动作。见 *****动作(actions)*****
- copies: 定义了一套拷贝动作。见 *****拷贝(copies)*****
- rules: 见 *****规则(rules)*****
- target_conditions: 类似于conditions，但是起左右的时间比conditions晚
- msvs_precompiled_header: 指定预编译头文件。只能用于Visual Studio
- msvs_precompiled_source: 指定预编译源文件。只能用于Visual Studio
- msvs_prebuild: 生成之前事件。只能用于Visual Studio
- msvs_postbuild: 生成之后事件。只能用于Visual Studio

```
'msvs_postbuild': r'copy "\((OutDir)\)(TargetName)" "C:\$(TargetName)"
```

- msvs_props: 指定target的属性页文件(.vsprops) 。只能用于Visual Studio

- msvs_cygwin_shell: 指定action运行在cygwin下。只能用于Visual Studio
- msvs_cygwin_dirs: 指定cygwin的目录。只能用于Visual Studio
- xcode_config_file: 在xcode中，指定target的配置文件(.xcconfig)。只能用于xcode
- xcode_framework_dirs: 在xcode中，指定框架的目录。只能用于xcode

配置(configurations)

`configurations` 段可以在 `targets` 和 `target_defaults` 段中。 `configurations` 不能够重写由 `target_defaults` 中指定的项。

在有些时候我们需要为项目指定多个配置，比如 `Debug` 、 `Release` 。 下面的一段为 *Windows中的常规配置*:

```
...
'configurations': {
'Debug': {
'msvs_settings': {
'VCCLCompilerTool': {
# 多线程调试 DLL (/MDd)
'RuntimeLibrary': '3',
# 不优化 /Od
'Optimization': '0',
# 用于"编辑并继续"的程序数据库 (/ZI)
'DebugInformationFormat': '4',
},
'VCLinkerTool': {
'GenerateDebugInformation': 'true',
'GenerateMapFile': 'false',
# 'SubSystem': '1',
},
},
}, # Debug
'Release': {
'msvs_settings': {
'VCCLCompilerTool': {
# 多线程 DLL (/MD)
'RuntimeLibrary': '2',
# 完全优化 /Os
'Optimization': '2',
# 使用内部函数 /Oi
'EnableIntrinsicFunctions': 'true',
# 程序数据库 (/Zi)
'DebugInformationFormat': '3',
},
```

```
'VCLinkerTool': {

'GenerateDebugInformation': 'true',

'GenerateMapFile': 'false',

},

},

}, # Release

},

...
```

条件(conditions)

`conditions` 和 `target_conditons` 可以出现在 `.gyp文件` 的任何位置。 `conditions` 语句在加载 `.gyp` 文件后即进行处理， `target_conditions` 语句在处理完所有依赖项后在处理。
在加载完全局和局部的数据后，使用python的 `eval()` 函数对条件字符串做处理。

动作(actions)

`actions` 提供了自定义处理输入输出的功能，其中的每一项都有下面的字段：

- action_name: action的名称，某些平台可能会忽略这个字段
- inputs: 输入信息，作为增量构建时使用
- outputs: 输出信息，作为增量构建时使用
- action: 构建命令
- message: 构建时显示的信息

构建系统会比较 `inputs` 和 `outputs` 中的文件是否是修改过，只有在修改过的情况下才会运行 `action` 。
在Xcode中，是通过shell脚本实现的；在Visual Studio中通过FileConfiguration包含一个自定义的VCCustomBuildTool实现。

```
'sources': [

# libraries.cc is generated by the js2c action below.

'<(INTERMEDIATE_DIR)/libraries.cc',],

'actions': [

{

'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)'],

},],
```

规则(rules)

`rules` 提供了自定义构建的功能，每一项都有下面的字段：

- rule_name: rule的名称，某些平台可能会忽略这个字段
- extension: 在本 `target` 中，所有以此为扩展名的源文件在构建时都使用这个规则
- inputs: 依赖的构建规则
- outputs: 输出信息，作为增量构建时使用。可以访问下面的 `RULE_INPUT_*` 变量
- action: 构建命令。可以访问下面的 `RULE_INPUT_*` 变量
- message: 构建时显示的信息。可以访问下面的 `RULE_INPUT_*` 变量

下面的这个变量可以在 `outputs` 、 `action` 和 `message` 中访问：

- RULE_INPUT_PATH: 当前输入的全路径
- RULE_INPUT_DIRNAME: 当前输入的目录
- RULE_INPUT_NAME: 当前输入的名称
- RULE_INPUT_ROOT: 当前输入的去掉扩展名
- RULE_INPUT_EXT: 当前输入的扩展名

```
{  'targets': [    {      'target_name': 'program',      'type': 'executable',
'msvs_cygwin_shell': 0,      'sources': [        'main.c',        'prog1.in',
'prog2.in',      ],      'rules': [        {          'rule_name': 'make_sources',
'extension': 'in',          'inputs': [            'make-sources.py',          ],
'outputs': [            '<(INTERMEDIATE_DIR)/<(RULE_INPUT_ROOT).c',
'<(INTERMEDIATE_DIR)/<(RULE_INPUT_ROOT).h',          ],          'action': [
'python', '<(_inputs)', '<(RULE_INPUT_NAME)', '<@(_outputs)',          ],
'process_outputs_as_sources': 1,        },      ],    },  ],}
```

`rules` 与 `actions` 有些相似之处。

拷贝(copies)

`copies` 提供了简单的文件拷贝功能，每一项都有下面的字段：

- destination: 拷贝的目的地文件夹
- files: 需要拷贝的文件列表

`copies` 会在 `destination` 创建相同名称的文件。

```
{  'targets': [    {      'target_name': 'copies1',      'type': 'none',
'copies': [        {          'destination': 'copies-out',          'files': [
'file1',          ],        },      ],    },    {      'target_name': 'copies2',
'type': 'none',      'copies': [        {          'destination':
'<(PRODUCT_DIR)/copies-out',          'files': [            'file2',          ],
},      ],    },    # 拷贝目录    {      'target_name': 'copies3',      'type':
'none',      'copies': [        {          'destination': '<(PRODUCT_DIR)/copies-
out',          'files': [            'directory/',          ],        },      ],
},  ],}
```

代码文件的命名

平台相关的文件命名规则为 `*_win.{ext}` 、 `*_mac.{ext}` 、 `*_linux.{ext}` 和 `*_posix.{ext}` 。例如：

- _win: foo_win.cpp
- _mac: foo_mac.cpp,foo_mac.mm
- _linux: foo_linux.cpp
- _posix: foo_posix.cpp

gyp的源代码

对于文档中一些未列出和有争议地方最可靠的是去看 `gyp` 的源代码和例子。 `gyp` 的测试代码很详

细，可以作为学习的例子���参考。

使用IDE调试生成的Makefile项目

在Linux上调试一般都是使用gdb，但是对于不态熟悉的就不好直接上手。这里推荐QtCreator，当然Netbeans和Eclispe CDT也可以，但是不如QtCreator方便和高效。

通用配置 *common.gypi*

```
{  'includes': [    'env.gypi'    ],  'include_dirs': [    '.',  ],
'configurations': {    'Debug': {      'defines': [        'DEBUG',      ],    },
'Release': {      'defines': [        'NDEBUG',      ],    },  }, # configurations
'conditions': [    ['OS=="win"', {      'defines': [        'WIN32',
'UNICODE',        '_UNICODE',        'OS_WIN',      ],
'msbuild_configuration_attributes': {        'IntermediateDirectory':
'$(OutDir)__BuildTemp\\$(ProjectName)\\',      },      'msbuild_toolset': {
#'PlatformToolset': 'v100'      }, # msbuild_toolset      'msvs_settings': {
'VCCLCompilerTool': {          'WarningLevel': '3',
'DisableSpecificWarnings': ['4251','4996'],          'WarnAsError': 'true',
},        'VCLinkerTool': {          'AdditionalDependencies': [
'kernel32.lib',            'user32.lib',          ],
'AdditionalLibraryDirectories': [],        },      }, # msvs_settings
'configurations': {        'Debug': {          'msvs_settings': {
'VCCLCompilerTool': {              # 多线程调试 DLL (/MDd)
'RuntimeLibrary': '3',              # 不优化 /Od              'Optimization': '0',
# 用于"编辑并继续"的程序数据库 (/ZI)              'DebugInformationFormat': '4',
},            'VCLinkerTool': {              'GenerateDebugInformation': 'true',
'GenerateMapFile': 'false',              # 'SubSystem': '1',            },
}, # msvs_settings        }, # Debug        'Release': {          'msvs_settings':
{            'VCCLCompilerTool': {              # 多线程 DLL (/MD)
'RuntimeLibrary': '2',              # 完全优化 /Os              'Optimization': '2',
# 使用内部函数 /Oi              'EnableIntrinsicFunctions': 'true',              # 程
序数据库 (/Zi)              'DebugInformationFormat': '3',            },
'VCLinkerTool': {              'GenerateDebugInformation': 'true',
'GenerateMapFile': 'false',            },          }, # msvs_settings        }, #
Release      }, # configurations    }], # Windows    ['OS=="mac"', {
'make_global_settings': [        ['CC', '/usr/bin/clang'],        ['CXX',
'/usr/bin/clang++'],        ],            'defines': [        'OS_POSIX',
'OS_MACOSX',      ],      'xcode_settings': {        'ALWAYS_SEARCH_USER_PATHS':
'NO',        #'i386', 'x86_64'        'ARCHS': [ 'x86_64' ],
'MACOSX_DEPLOYMENT_TARGET': '10.6',        'CC': 'clang',        'GCC_VERSION':
'com.apple.compilers.llvm.clang.1_0',        'CLANG_CXX_LANGUAGE_STANDARD':
'c++0x',        # libstdc++, c++11, libc++        'CLANG_CXX_LIBRARY': 'libstdc++',
'GCC_ENABLE_OBJC_GC': 'unsupported',        #'LIBRARY_SEARCH_PATHS': [],
'GCC_ENABLE_CPP_EXCEPTIONS': 'YES',        'GCC_SYMBOLS_PRIVATE_EXTERN': 'NO',
'DEBUG_INFORMATION_FORMAT': 'dwarf-with-dsym',        #'DEPLOYMENT_POSTPROCESSING':
'YES',        'OTHER_CFLAGS': [          '-fno-eliminate-unused-debug-symbols',
'-mmacosx-version-min=10.6',          # compile use oc++          '-x objective-
c++',        ],        'WARNING_CFLAGS': ['-Wno-deprecated-declarations'],
'WARNING_CFLAGS!': ['-Wall', '-Wextra',],        'WARNING_CXXFLAGS': ['-Wstrict-
aliasing', '-Wno-deprecated',],      }, # xcode_settings      'link_settings': {
'libraries': [
'$(SDKROOT)/System/Library/Frameworks/Foundation.framework',        ],      },
'libraries': [],      'mac_framework_dirs': [],      'configurations': {
'Debug': {          'xcode_settings': {            'GCC_DEBUGGING_SYMBOLS': 'full',
'STRIP_INSTALLED_PRODUCT': 'YES',            'GCC_OPTIMIZATION_LEVEL': '0',
'OTHER_CFLAGS': ['-g',],            'OTHER_CXXFLAGS': ['-g',],          }, #
xcode_settings        }, # Debug        'Release': {          'xcode_settings': {
'GCC_OPTIMIZATION_LEVEL': 's',          }, # xcode_settings        }, # Release
}, # configurations    }], # Mac    ['OS=="linux"', {      'defines': [
'OS_POSIX',        'OS_LINUX',      ],      'cflags': [        # Support 64-bit
shared libs (also works fine for 32-bit).        '-fPIC',        '-std=c++11',
'-fstrict-aliasing',        '-Wall',        '-Wextra',        '-Wshadow',        '-
Wconversion',        #'-Wpadded',        '-Wstrict-aliasing=2',        '-Wstrict-
overflow=4',      ],      'ldflags': [],      'configurations': {        'Debug': {
'cflags': [            '-g',            '-C',          ],        },
'Release': {          'cflags': [            '-O2',          ],        },      }, #
configurations    }], # Linux  ], # conditions}
```

gyp命令行

- `--depth`：Chromium历史遗留问题，需要设置为 `.`
- `--generator-out`：设置生成项目的输出文件夹
- `--f`：设置生成项目的类型，不设置则根据平台决定生成的项目类型。可用的值有：`msvs`、`xcode`、`make`

- `--G` : 设置生成的其他标记参数，这个值和具体的平台有关。
  - msvs_version: 指定生成的Visual Studio项目的版本，不设置则会根据系统中安装的最新的Visual Stuio生成项目。可用的值有 `2005` 、 `2005e` 、 `2008` 、 `2008e` 、 `2010` 、 `2010e` 、 `2012` 、 `2012e` 、 `2013` 、 `2013e` 。例如： `--Gmsvs_version=2010`
- `--toplevel-dir` : 设置源代码的跟目录，默认将取 `--depth` 指定的值
- `-D` : 传入变量到.gyp，可以使用 `<(varname)` 这种方式读取。例如： `-Dbuildtype=share_library`
- `--no-circular-check` :

写API文档是一件很蛋疼的事情，以后继续添加。

标签: 开发工具

FelixZYY
关注 - 0
粉丝 - 1
+加关注

0 0

(请您对文章做出评价)

posted @ 2014-08-15 08:49 FelixZYY 阅读(1494) 评论(1) 编辑 收藏

评论列表

#1楼 2016-01-25 17:37 Beny

请问一下，actions 生成的vs 工程文件，把双引号生成“"”（html标签）了，请问如何解决。
actions**如下**:
'actions': [{
'action_name': 'copy to plugins',
'msvs_cygwin_shell': 0,
'inputs': [],
'outputs': [
'<(PRODUCT_DIR)/plugins/test.dll',
],
'action': [
'xcopy /efy',
'<(PRODUCT_DIR)/test.dll',
#'$(OutDir)',
'<(PRODUCT_DIR)/plugins/test.dll',
],
},],

生成的vs的内容如下:
<Command>call call copy \efy "$(OutDir)test.dll"
"$(OutDir)plugins\test.dll"
if %errorlevel% neq 0 exit /b %errorlevel%</Command>

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:

- 据说阿里巴巴投了东南亚版“亚马逊”，对，就是靠“抄袭”起家的那家
- 李一男在受审 牛电科技却宣称融资3000万美元
- 攻击者利用百度广告API传播恶意程序
- 迪士尼手游部门主管：过度扩张等于自杀
- 美飞船携流星相机升空 5月将“纵火”返地球

» 更多新闻...

最新知识库文章:

- 如何运维千台以上游戏云服务器
- 架构漫谈（一）：什么是架构?
- 架构的本质
- 谷歌背后的数学
- Medium开发团队谈架构设计

» 更多知识库文章...