GYP(Generate Your Projects)

目录

1 说明

本文档翻译 GYP 的相关文档，并做合并编辑工作。文档原始出处如下：

- GYPUserDocumentation

https://code.google.com/p/gyp/wiki/GypUserDocumentation

GYP (Generate Your Projects)
Status: Draft (as of 2009-05-19)

Mark Mentovai <mark@chromium.org>, Steven Knight <sgk@chromium.org> *et al.*
Modified: 2009-05-19

- GYPLanguageSpecification

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

GYP (Generate Your Projects)
Status: Draft (as of 2009-01-30)

Mark Mentovai <mark@chromium.org> *et al.*
Modified: 2009-02-10

2 介绍

GYP 是 Google 开发的跨平台项目工程生成系统。本文文档第 2-4 章节通过实例和用例介绍 GYP 的基本使用。这部分旨在提供用户级别的 GYP 使用指南。第 5 章则为 GYP 配置文件的语言规格做详细的说明。

GYP 项目同 Google 的其他开源项目一样，托管在 Google Code 上：
https://code.google.com/p/gyp/

3 示例骨架（Skeleton）

本节介绍.gyp 文件的基本骨架。

3.1 Chromium

下面是一个 Chromium 项目的 GYP 配置骨架：

```
{
  'variables': {
    .
    .
    .
  },
  'includes': [
    '../build/common.gypi',
  ],
  'target_defaults': {
    .
    .
    .
  },
  'targets': [
    {
      'target_name': 'target_1',
        .
        .
        .
    },
    {
      'target_name': 'target_2',
        .
        .
        .
    },
  ],
  'conditions': [
    ['OS=="linux"', {
      'targets': [
        {
          'target_name': 'linux_target_3',
            .
            .
```

```
            .
          },
        ],
      }],
      ['OS=="win"', {
        'targets': [
          {
            'target_name': 'windows_target_4',
              .
              .
              .
          },
        ],
      }, { # OS != "win"
        'targets': [
          {
            'target_name': 'non_windows_target_5',
              .
              .
              .
          },
      }],
    ],
  }
```

可以看到，一个 gyp 文件里面实际上就是一个 Python 字典（实际上除了引入 Python 风格注释，在字典和列表后面可以添加逗号外，跟 Json 并无区别）。从上面的例子我们看到，gyp 配置的顶级元素有下面几种：

节点元素	说明	节点值类型
‘variables’	定义可在本文件其他部分使用的变量	Python 字典
‘includes’	本文件所包含的其他.gypi 文件列表	Python 列表
‘target_defaults’	对本文件所有 targets 都生效的设置	Python 字典
‘targets’	本文件所有编译项目设置，每个 Target 一个字典	Python 列表
‘conditions’	根据全局变量的值修改 gyp 其他定义节点	Python 列表

3.2 Execute Target

最简单的目标项（Target）就是一个简单的可执行程序。下面是一个可执行程序的.gyp 配置。

```
{
    'targets': [
```

```
    {
      'target_name': 'foo',
      'type': 'executable',
      'msvs_guid': '5ECEC9E5-8F23-47B6-93E0-C3B328B3BE65',
      'dependencies': [
        'xyzzy',
        '../bar/bar.gyp:bar',
      ],
      'defines': [
        'DEFINE_FOO',
        'DEFINE_A_VALUE=value',
      ],
      'include_dirs': [
        '..',
      ],
      'sources': [
        'file1.cc',
        'file2.cc',
      ],
      'conditions': [
        ['OS=="linux"', {
          'defines': [
            'LINUX_DEFINE',
          ],
          'include_dirs': [
            'include/linux',
          ],
        }],
        ['OS=="win"', {
          'defines': [
            'WINDOWS_SPECIFIC_DEFINE',
          ],
        }, { # OS != "win",
          'defines': [
            'NON_WINDOWS_DEFINE',
          ],
        }]
      ],
    },
  ],
}
```

从上面的配置可以看到，一个典型的 target 一般包含下表所示的子节点：

节点元素	说明	节点值类型

‘target_name’	目标项名称，必须在所有.gyp 文件中都唯一。该名字用来生成 Visual Studio 项目名；XCode 目标项名字；SCons 目标项别名。	字符串
‘type’	目标项类型，此例中为'executable'代表可执行程序。	‘executeable’
‘msvs_guid’	唯一标识目标项，生成 Visual Studio sln 文件所需	GUID
‘dependencies’	列出依赖的其他 Targets 名称列表。 1、 所依赖的 Targets 会先被编译。 2、 如果所依赖的 Target 是一个库，会被链接到本项目。 3、 所依赖的 Targets 的 direct_dependent_settings 节点的设置会被应用到本 Target 的编译和链接。参考下 direct_dependent_settings 的说明	Python 列表
‘defines’	C 预处理定义列表，被加入到编译命令行选项（-D 或/D）	Python 列表
‘include_dirs’	头文件包含路径，被加入到编译命令行选项（-I 或/I）	Python 列表
‘sources’	源文件列表	Python 列表
‘conditions’	根据全局变量的值添加设置	Python 列表

3.3 Library Target

除了可执行程序目标项目，更多的是目标项目是类库。下面是一个类库目标项目的 gyp 设置示例，包含了大部分类库所需的 gyp 设置。

```
{
    'targets': [
      {
        'target_name': 'foo',
        'type': '<(library)'
        'msvs_guid': '5ECEC9E5-8F23-47B6-93E0-C3B328B3BE65',
        'dependencies': [
          'xyzzy',
          '../bar/bar.gyp:bar',
        ],
        'defines': [
          'DEFINE_FOO',
          'DEFINE_A_VALUE=value',
        ],
        'include_dirs': [
          '..',
        ],
        'direct_dependent_settings': {
          'defines': [
            'DEFINE_FOO',
            'DEFINE_ADDITIONAL',
```

```
        ],
        'linkflags': [
        ],
      },
      'export_dependent_settings': [
        '../bar/bar.gyp:bar',
      ],
      'sources': [
        'file1.cc',
        'file2.cc',
      ],
      'conditions': [
        ['OS=="linux"', {
          'defines': [
            'LINUX_DEFINE',
          ],
          'include_dirs': [
            'include/linux',
          ],
        ],
        ['OS=="win"', {
          'defines': [
            'WINDOWS_SPECIFIC_DEFINE',
          ],
        }, { # OS != "win",
          'defines': [
            'NON_WINDOWS_DEFINE',
          ],
        }]
      ],
    ],
  }
```

从上面的配置可以看到，和可执行目标项相比，类库目标项的 gyp 节点多出来几个：

节点类型	说明	节点值类型说明
‘type’	库项目类型，一般定义为’<(library)’，如果需要指明静态或动态，则可以显示使用’static_library’或’shared_library’	‘<(library)’ ‘static_library’ ‘shared_library’
‘direct_dependent_settings’	该节点的设置会被应用到那些直接依赖于该 Target 的 Targets。 参考’dependenties’节点说明。	Python 字典
‘export_dependent_settings’	导出 Targets 列表，这些 Target 的’direct_dependent_settings’节点将会被	Python 列表

	应用到那些直接依赖该 Target 的 Targets	

4 用例

本章详细说明典型用例，这些用例并非完整的 gyp 文件，只列出每个用例相关的 gyp 片段。

4.1 添加一个源文件（Source File）

一个源文件可能是跨平台的，也可能是平台相关的，gyp 分别为这两种需求提供了指定源文件的方法。

4.1.1 添加全平台源文件

```
{
  'targets': [
    {
      'target_name': 'my_target',
      'type': 'executable',
      'sources': [
        '../other/file_1.cc',
        'new_file.cc',
        'subdir/file3.cc',
      ],
    },
  ],
},
```

跨全平台文件直接在'targets'/'sources'里列出。最好是以字母序排序。文件的路径是 gyp 文件所在地路径的相对路径。

4.1.2 添加特定平台源文件

4.1.2.1 指定后缀

为特定平台添加源文件最简单的方式是为源文件添加指定后缀，并在 gyp 文件里添加过滤规则。下表列出了特定平台源文件的约定后缀：

平台	后缀	示例

Linux	_linux	foo_linux.cc
Mac	_mac	foo_mac.cc
Posix	_posix	foo_posix.cc
Win	_win	foo_win.cc

gyp 片段如下：

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'sources': [
        'independent.cc',
        'specific_win.cc',
      ],
    },
  ],
},
```

Chromium 的 gyp 文件都有添加这些后缀源文件的过滤规则，比如本例的 specific_win.cc 文件会自动在非 windows 平台上被过滤掉。

4.1.2.2 条件源文件

如果源文件没有这些特定的后缀，则可以通过‘conditions’节点来特化平台相关的源文件，包括排除平台和指定平台两种办法。

- 排除平台（推荐做法）

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'sources': [
        'linux_specific.cc',
      ],
      'conditions': [
        ['OS != "linux"', {
          'sources!': [ #注意这里是'sources!'
            # Linux-only; exclude on other platforms.
            'linux_specific.cc',
          ]
        }],
      ],
```

```
    },
  ],
},
```

- 指定平台（不推荐）

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'sources': [],
      ['OS == "linux"', {
        'sources': [
          # Only add to sources list on Linux.
          'linux_specific.cc',
        ]
      }],
    },
  ],
},
```

4.2 添加一个可执行目标项（Execute Target）

4.2.1 全平台

```
{
  'targets': [
    {
      'target_name': 'new_unit_tests',
      'type': 'executable',
      'defines': [
        'FOO',
      ],
      'include_dirs': [
        '..',
      ],
      'dependencies': [
        'other_target_in_this_file',
        'other_gyp2:target_in_other_gyp2',
      ],
      'sources': [
        'new_additional_source.cc',
```

```
          'new_unit_tests.cc',
        ],
      },
    ],
  }
```

4.2.2 特定平台

```
{
    'targets': [
    ],
    'conditions': [
      ['OS=="win"', {
        'targets': [
          {
            'target_name': 'new_unit_tests',
            'type': 'executable',
            'defines': [
              'FOO',
            ],
            'include_dirs': [
              '..',
            ],
            'dependencies': [
              'other_target_in_this_file',
              'other_gyp2:target_in_other_gyp2',
            ],
            'sources': [
              'new_additional_source.cc',
              'new_unit_tests.cc',
            ],
          },
        ],
      }],
    ],
  }
```

4.3 添加一个库目标项（Library Target）

4.3.1 全平台

```
{
  'targets': [
    {
      'target_name': 'new_library',
      'type': '<(library)',
      'defines': [
        'FOO',
        'BAR=some_value',
      ],
      'include_dirs': [
        '..',
      ],
      'dependencies': [
        'other_target_in_this_file',
        'other_gyp2:target_in_other_gyp2',
      ],
      'direct_dependent_settings': {
        'include_dirs': '.',
      },
      'export_dependent_settings': [
        'other_target_in_this_file',
      ],
      'sources': [
        'new_additional_source.cc',
        'new_library.cc',
      ],
    },
  ],
}
```

需要注意的是，'type'默认使用'<(library)'，允许用户在 gyp 全局配置里设置具体使用静态还算共享库。或者可以固定使用静态或动态连接库：'static_library'或'shared_library'

4.3.2 特定平台

```
{
  'targets': [
  ],
  'conditions': [
    ['OS=="win"', {
      'targets': [
```

```
        {
          'target_name': 'new_library',
          'type': '<(library)',
          'defines': [
            'FOO',
            'BAR=some_value',
          ],
          'include_dirs': [
            '..',
          ],
          'dependencies': [
            'other_target_in_this_file',
            'other_gyp2:target_in_other_gyp2',
          ],
          'direct_dependent_settings': {
            'include_dirs': '.',
          },
          'export_dependent_settings': [
            'other_target_in_this_file',
          ],
          'sources': [
            'new_additional_source.cc',
            'new_library.cc',
          ],
        },
      ],
    }],
  ],
}
```

4.4 为目标项(Target)添加设置（Settings）

本节详细说明常用目标项设置的配置，本节 gyp 代码都是片段代码。

4.4.1 添加预处理宏定义

预处理宏定义添加到 defines 节点：

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'defines': [
```

```
          'FOO',
          'BAR=some_value',
        ],
      },
    ],
  },
```

4.4.2 添加头文件包含目录

头文件包含目录添加到'include_dirs'节点，该节点可能直接在在 target 下也可以在'conditions'节点下。

```
  {
    'targets': [
      {
        'target_name': 'existing_target',
        'include_dirs': [
          '..',
          'include',
        ],
      },
    ],
  },
```

4.4.3 添加编译 flags

编译 flags 添加到'cflags'节点，由于编译 flags 一般都算平台相关的，所以 cflags 节点一般在 conditions 节点下。

```
  {
    'targets': [
      {
        'target_name': 'existing_target',
        'conditions': [
          ['OS=="win"', {
            'cflags': [
              '/WX',
            ],
          }, { # OS != "win"
            'cflags': [
              '-Werror',
            ],
          }],
```

```
      ],
    },
  ],
},
```

4.4.4 添加链接 flags

链接 flags 是链接器相关的。

在 linux 和非 mac 的 posix 系统上，添加到'ldflags'节点上。

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'conditions': [
        ['OS=="linux"', {
          'ldflags': [
            '-pthread',
          ],
        }],
      ],
    },
  ],
},
```

在 OS X 系统上，添加到'xcode_settings'节点上。
在 Windows 系统上，添加到'msvs_settings'节点上。

4.4.5 添加指定平台设置

每个设置关键字都可以添加感叹号后缀用以移除某些设置，比如 chromium 的 common.gypi 里将 Linux 平台的 -Werror 编译 flags 移除：

```
{
  'targets': [
    {
      'target_name': 'third_party_target',
      'conditions': [
        ['OS=="linux"', {
          'cflags!': [
            '-Werror',
          ],
```

```
        }],
      ],
    },
  ],
},
```

4.5 配置目标项（Target）之间的依赖（Dependencies）

gyp 系统提供了指定目标项之间互相依赖的配置方案。下面分别说明需要配置的步骤。

4.5.1 添加依赖的链接库

在‘dependencies’节点添加依赖的链接库：

```
{
  'targets': [
    {
      'target_name': 'foo',
      'dependencies': [
        'libbar',
      ],
    },
    {
      'target_name': 'libbar',
      'type': '<(library)',
      'sources': [
      ],
    },
  ],
}
```

如果所依赖的目标项在另外一个 gyp 文件，则需要指明 gyp 文件的相对路径：

```
{
  'targets': [
    {
      'target_name': 'foo',
      'dependencies': [
        '../bar/bar.gyp:libbar',
      ],
    },
  ],
}
```

如果更新了 bar.gyp 的名字，则需要更新所有依赖于 bar.gyp 的地方。

4.5.2 指定依赖者所需的设置

目标项可以在’direct_dependent_settings’节点添加依赖者所需设置，这些设置会被链接到依赖者，形成最终的配置文件。

```
{
    'targets': [
      {
        'target_name': 'foo',
        'type': 'executable',
        'dependencies': [
          'libbar',
        ],
      },
      {
        'target_name': 'libbar',
        'type': '<(library)',
        'defines': [
          'LOCAL_DEFINE_FOR_LIBBAR',
          'DEFINE_TO_USE_LIBBAR',
        ],
        'include_dirs': [
          '..',
          'include/libbar',
        ],
        'direct_dependent_settings': {
          'defines': [
            'DEFINE_TO_USE_LIBBAR',
          ],
          'include_dirs': [
            'include/libbar',
          ],
        },
      },
    ],
  }
```

如上所示，目标项 foo 依赖于同文件的 libbar，而后者在’direct_dependent_settings’ 里指定了’defines’和’include_dirs’项，所以 foo 的编译命令最终会加下面选项：

```
-D DEFINE_TO_USE_LIBBAR –I include/libbar
```

需要注意的是，’direct_dependent_settings’里的设置并不影响 libbar 本身的设置，比如 libbar 在自己的’defines’节点有’DEFINE_TO_USE_LIBBAR’宏定义，在自己的’include_dirs’里有‘include/libbar’目录。

4.6 为 Mac OS X bundles 添加支持

```
{
      'target_name': 'test_app',
      'product_name': 'Test App Gyp',
      'type': 'executable',
      'mac_bundle': 1,
      'sources': [
        'main.m',
        'TestAppAppDelegate.h',
        'TestAppAppDelegate.m',
      ],
      'mac_bundle_resources': [
        'TestApp/English.lproj/InfoPlist.strings',
        'TestApp/English.lproj/MainMenu.xib',
      ],
      'link_settings': {
        'libraries': [
          '$(SDKROOT)/System/Library/Frameworks/Cocoa.framework',
        ],
      },
      'xcode_settings': {
        'INFOPLIST_FILE': 'TestApp/TestApp-Info.plist',
      },
    },
```

4.7 在重构中正确的移除文件

TODO（GYP 官方文档还没提供此部分）

4.8 定制编译步骤

TODO（GYP 官方文档还没提供此部分）

5 语言规格

5.1 目标

GYP 工具的目标是为 Chromium 提供一个平台无关的配置系统，GYP 工具通过 gyp 配置系统生成各平台的编译系统。包括 Visual Studio, Xcode, SCons 以及 make files 等。GYP 的目标是让输入格式尽量通用，但不会浪费时间去做到完全正确。

5.2 背景

无论是 Google 内部还算外部，都有很多项目试图创建简单、通用的跨平台编译系统，允许每个平台有一定的灵活性以解决不同平台的差异。事实上，没有哪个系统明显能满足 Chrojmium 的需求，看上去这是一个棘手的问题。GYP 的目标是成功创建一套工具，为 Chromium 高度定制，而不是做一套能处理所有跨平台编译问题的工具。
Mac 系统下的编译环境最复杂。所以我们以 Xcode 为模型做初始配置，然后将配置适配到其他平台。（译注：聪明的做法!）.

5.3 预览

GYP 系统拥有以下几个特点：

- 输入配置写在.gyp 后后缀的文件里。
- 每个.gyp 文件指定了如何编译文件里定义的「组件（Component）」
  - 在 Mac 系统，一个.gyp 文件生成一个 Xcode .xcodeproj bundle，用以编译 Target。
  - 在 Windows 系统，一个.gyp 文件生成了一个 Visual Studio .sln 文件，并为每个 Target 生成一个.vcproj 工程文件。、
  - 在 Linux 系统，一个.gyp 文件生成一个 SCons 文件，为每个 Target 生成一个 makefile 文件。
- .gyp 文件的语法是 Python 数据结构（字典、列表、字符串）。
- 禁止任意使用 Python 代码
  - 对全局和本地.gyp 文件里使用 eval()执行，从而.gyp 里只允许有表达式而非任意 Python 语句。
  - 所有的输入应该与 Json 格式兼容，除了两种情况：1、使用#写单行注释 2、允许在列表和字典后添加逗号。
- 输入是一个 key-value 构成的字典。
- 无效的关键字是非法的，直接被忽略掉。

5.4 设计细节

GYP 的设计原则：

- 一致的重用关键字
- 平台相关的关键字带上平台相关的概念前缀
  - 例如：msvs-disabled_warnings, xcode_framework_dirs
- 输入语法是声明性的，数据驱动的
  - 通过使用 Python 的 eval()而非 exec 来强制这点，前者只解析表达式，后者可以解析任意 Python 语句。
- 关键字的具体语义是递归延迟的，直到所有的文件被读取，并且根据配置选择适当的文件。
- 源文件列表相关
  - 源文件列表是平坦化的，源文件列表的排序遵循通常的开发者规则，当源文件被链接后，源文件列表会保持它们被列出的顺序。
  - 源文件列表不包含逻辑文件夹机制，例如 Visual Studio 的 Filter 元素以及 Xcode 的 Groups。
  - 文件夹嵌套自动映射到文件夹系统。
- 占位符

例子：

```
{
  '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',
        ],
      },
    },
  ],
}
```

5.4.1 顶级元素

gyp 字典的顶级元素如下：

节点关键字	说明
‘conditions’	条件节可以包含任意其他顶级节点
‘includes’	所包含的.gypi 文件列表
‘target_defaults’	默认目标项配置，被所有顶级 targets 继承
‘targets’	target 列表
‘variables’	变量列表，变量定义是字符串键值对

5.4.2 目标项（Targets）

节点关键字	说明
‘actions’	自定义行为列表，每个行为接受输入文件，产生输出文件。
‘all_dependent_settings’	为所有直接或间接依赖的目标项提供统一设置。注意与’direct_dependent_settings’和’link_settings’区别开来。该节点可以包含几乎所有 target 的子元素，除了’configurations’, ’target_name’以及’type’元素。
‘configurations’	编译配置列表
‘copies’	拷贝动作列表
‘defines’	预处理宏定义列表
‘dependencies’	依赖 target 列表，在同一个文件夹内的直接写 target 名字；在其他.gyp 文件里的，需要指明相对路径，例如：../path/to/other.gyp:target_we_want

‘direct_dependent_settings’	所有直接依赖该 target 的其他 target 所需的设置。该节点可以包含几乎所有的 target 子节点，除了’configurations’, ‘target_name’以及’type’节点。注意与’all_dependent_settings’和’link_settings’的区别。
‘include_dirs’	头文件包含目录列表，被加到 C/C++编译选项-I 或/I
‘libraries’	依赖库列表
‘link_settings’	所有链接了该 target 的其他 target 所需的设置。’type’为‘executable’和’shared_library’的 target 是可链接的，所以如果它们依赖于某个不可链接的目标，比如’static_library’，它们将采用不可链接目标的’link_settings’。该节点可以包含几乎所有的 target 子节点，除了’configurations’,’target_name’,’type’等。注意和’all_dependent_settings’以及’direct_dependent_settins’区别。
‘rules’	一个特殊的自定义行为，接受输入文件列表，产生输出文件列表。
‘sources’	源文件列表
‘target_conditions’	类似’conditions’，但是直到他们被合并到实际 target 时才解析。
‘target_name’	目标项名字
‘type’	目标项类型，’executable’,’<(library)’,’static_library’或’shared_library’，如果设置为’none’，则不会被链接。这在资源项目或文档项目时很有用。
‘msvs_props’	Visual Studio 属性文件(.vsprops)列表
‘xcode_config_file’	Xcode 配置文件(.xcconfig)列表
‘xcode_framework_dirs’	Xcode Framework 目录列表

5.4.3 配置（Configurations）

‘congiguration’ 节点可以被’targets’或’target_defaults’节点包含。’configurations’节点是一个列表，因此当前无法覆盖在’target_defaults’里定义的‘configurations’.

特别需要注意到是，应该为同一个工程的每个 target 定义同样的’configurations’集合。即使某个工程跨越了多个.gyp 文件。

一个 Configuration 可以包含几乎所有的’target’子节点，除了’actions’, ‘all_dependent_settins’, ‘configurations’, ‘dependencies’, ‘direct_dependent_settings’, ‘libraries’, ‘link_settings’, ‘sources’, ‘target_name’, ‘type’.

最后，每个 Configuration 必须包含’configuration_name’节点。

5.4.4 条件（Conditionals）

Conditionals 节点可以是.gyp 文件的任何字典的直接子节点。有两种类型，'conditions'节点和'target_conditions'节点，前者在加载.gyp 文件时被处理，后者在所有依赖项合并完后处理。

'conditions'节点和'target_conditions'节点的值是一个列表，每个列表都包含三部分，第一部分是条件表达式；第二部分是条件表达所为 true 时会被融合的设置；第三部分是可选的，表示条件表达式为 false 时的设置。其中，第一部分和第二部分是必须的。

```
'conditions': [
          ['OS=="linux"', {
            'cflags!': [
              '-Werror',
            ],
          },{
            #可选的
          }],
]
```

如左边示例，
黄色是条件表达式
灰色是条件表达式成立时的设置
蓝色是条件表达所不成立时的设置
蓝色部分是可选的。

5.4.5 行为（Actions）

一个行为提供了对输入的一个或多个文件进行处理并产生输出文件的操作。'actions'节点的值是一个列表，列表的每个元素都是一个字典，这个字典的子元素如下：

节点关键字	说明	类型
'action_name'	行为名称，有些编译系统需要唯一的行为名称，有些则完全忽略行为名称。	string
'inputs'	输入文件列表	list
'outputs'	输出文件列表	list
'action'	行为命令行列表。为了最大化跨平台兼容，如果使用 Python 解释器，列表的第一个元素是'python'。	list
'message'	编译过程中需要显示的信息。	string

编译环境会比较输入和输出，如果任意输出文件丢失或比输入文件生成时间早，则会执行该行为。如果所有的输出文件都存在并且生成时间比输入文件早，则输出文件被认为是最新的，行为不会被执行。
Actions 在 Xcode 里被实现为 shell 脚本，并且在编译前执行。Actions 在 Visual Studio 里实现为编译文件的 FileConfiguration 元素的 VCCustomBuildTool，并分别将'action'指定为 CommandLine，'outputs'指定为 Outputs，'inputs'指定为'Additional Dependencies'.
下面是一个例子：

```
      '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)', #gyp 的变量名引用方式
          ],
          'outputs': [
            '<(INTERMEDIATE_DIR)/libraries.cc',
            '<(INTERMEDIATE_DIR)/libraries-empty.cc',
          ],
          'action': ['python', 'tools/js2c.py', '<@(_outputs)', 'CORE',
'<@(core_library_files)'],
        },
      ],
```

5.4.6 规则（Rules）

‘rules’节点提供了自定义编译行为，对给定输入文件列表，产生输出文件列表。rules 节点的值是一个列表，列表的每个元素是一个字典，字典的子节点如下：

节点关键字	说明	类型
‘rule_name’	规则名字，有些编译系统要求唯一的规则名字，有些则完全忽略规则名字。	sting
‘extension’	target 的所有源文件如果标记了该关键字，则都会被当作 rule 的 inputs	string
‘inputs’	输入文件列表	list
‘outputs’	输出文件列表	list
‘action’	命令行列表。为了尽量兼容不同的平台，执行 python 脚本应该在列表的第一个元素指定’python’。	list
‘message’	编译时显示的消息，可以使用 RULE_INPUT_XXX 变量。	string

下面的变量名可以被’output’, ‘action’, ‘message’使用。

变量名	说明
RULE_INPUT_PATH	当前文件的绝对路径
RULE_INPUT_DIRNAME	当前文件所在地目录
RULE_INPUT_NAME	当前文件的文件名
RULE_INPUT_ROOT	当前文件的不带扩展名的文件名
RULE_INPUT_EXT	当前文件的扩展名

Rules 可以看做是 Action 生成器。所有标记了同一个 extension 的源文件都会被同一个 Rules 执行，并指定了同样的’inputs’, ‘outputs’, ‘action’, ‘message’，源文件被添加到’inputs’列表。

5.4.7 拷贝（Copies）

‘copies’节点提供了拷贝动作。’copies’节点是一个列表，列表的每个元素是一个字典，字典的子节点如下：

节点关键字	说明	类型
‘destination’	目标拷贝路径	string
‘files’	待拷贝文件列表	list

5.4.8 生成 XCode .pbxproj 文件

5.4.9 生成 Visual Studio .vcxproj 文件

5.4.10生成 Visual Studio .sln 文件

6 输入格式参考

https://code.google.com/p/gyp/wiki/InputFormatReference