# 构建常见项目类型

本章介绍如何为基于 Qt 的三种常见项目类型设置 qmake 项目文件：应用程序、库和插件。尽管所有项目类型都使用许多相同的变量，但它们中的每一个都使用特定于项目的变量来自定义输出文件。

此处不介绍特定于平台的变量。有关详细信息，请参阅适用于 Windows 的 Qt - 部署和适用于 macOS 的 Qt。

## 构建应用程序

该模板告诉 qmake 生成一个将生成应用程序的生成文件。使用此模板，可以通过向 CONFIG 变量定义添加以下选项之一来指定应用程序的类型：app

| 选择 | 描述: _____ |
| --- | --- |
| 窗户 | 该应用程序是一个视窗 GUI 应用程序。 |
| 安慰 | app仅模板：该应用程序是一个窗口控制台应用程序。 |
| 测试用例 | 该应用程序是一个自动测试。 |

使用此模板时，可以识别以下 qmake 系统变量。应在 .pro 文件中使用它们来指定有关应用程序的信息。有关其他与平台相关的系统变量，您可以查看平台说明。

- 标头 - 应用程序的头文件列表。
- 源 - 应用程序的C++源文件的列表。
- 表单 - 应用程序的 UI 文件列表（使用 Qt 设计器创建）。
- 词典 - 应用程序的 Lex 源文件的列表。
- YACC 源 - 应用程序的 Yacc 源文件列表。
- 目标 - 应用程序的可执行文件的名称。这默认为项目文件的名称。（如果有的话，扩展名将自动添加）。
- DESTDIR - 放置目标可执行文件的目录。
- 定义 - 应用程序所需的任何其他预处理器定义的列表。
- 包含路径 - 应用程序所需的任何其他包含路径的列表。
- 依赖关系路径 - 应用程序的依赖关系搜索路径。
- VPATH - 用于查找提供文件的搜索路径。
- DEF_FILE - 仅限 Windows：要为应用程序链接的 .def 文件。

您只需使用具有其值的系统变量。例如，如果您没有任何额外的包含对象，则无需指定任何附加对象。qmake

```
TEMPLATE = app
DESTDIR  = c:/helloapp
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
DEFINES += USE_MY_STUFF
CONFIG  += release
```

对于单值项目，例如模板或目标目录，我们使用"=";但对于多值项目，我们使用"+="*添加到该*类型的现有项目中。使用"="将变量值替换为新值。例如，如果我们写 ，所有其他定义都将被删除。
DEFINES=USE_MY_STUFF

## 构建测试用例

测试用例项目是旨在作为自动测试运行的项目。任何都可以通过将值添加到变量中来标记为测试用例。
appapptestcaseCONFIG

对于测试用例项目，qmake 会将一个目标插入到生成的生成文件中。此目标将运行应用程序。如果测试以等于零的退出代码终止，则认为测试通过。check

目标通过子DIRS项目自动递归。这意味着可以从 SUBDIRS 项目中发出命令来运行整个测试套件。checkmake check

目标的执行可以通过某些生成文件变量进行自定义。这些变量是：check

| 变量 | 描述: _____ |
| --- | --- |
| 测试运行器 | 附加到每个测试命令前面的命令或 shell 片段。一个示例用例是"超时"脚本，如果测试未在指定时间内完成，它将终止测试。 |
| 睾丸 | 附加到每个测试命令的其他参数。例如，传递其他参数以设置测试中的输出文件和格式（如 QTestLib 支持的选项）可能很有用。-o filename,format |

**注意：** 必须在调用工具时设置变量，而不是在 .pro 文件中设置变量。大多数工具都支持直接在命令行上设置 Makefile 变量：makemake

```
# Run tests through test-wrapper and use JUnit XML output format.
# In this example, test-wrapper is a fictional wrapper script which terminates
# a test if it does not complete within the amount of seconds set by "--timeout".
# The "-o result.xml,junitxml" options are interpreted by QTestLib.
make check TESTRUNNER="test-wrapper --timeout 120" TESTARGS="-o result.xml,junitxml"
```

Testcase projects may be further customized with the following options:CONFIG

| Option | Description |
| --- | --- |
| insignificant_test | The exit code of the test will be ignored during .make check |

failure.testcaseCONFIG+=testcasemake check

# Building a Library

The template tells qmake to generate a Makefile that will build a library. When using this template, the VERSION variable is supported, in addition to the system variables that the template supports. Use the variables in your .pro file to specify information about the library.`libapp`

When using the template, the following options can be added to the CONFIG variable to determine the type of library that is built:`lib`

| Option | Description |
| --- | --- |
| dll | The library is a shared library (dll). |
| staticlib | The library is a static library. |
| plugin | The library is a plugin. |

The following option can also be defined to provide additional information about the library.

- VERSION - The version number of the target library. For example, 2.3.1.

The target file name for the library is platform-dependent. For example, on X11, macOS, and iOS, the library name will be prefixed by . On Windows, no prefix is added to the file name.`lib`

# Building a Plugin

Plugins are built using the template, as described in the previous section. This tells qmake to generate a Makefile for the project that will build a plugin in a suitable form for each platform, usually in the form of a library. As with ordinary libraries, the VERSION variable is used to specify information about the plugin.`lib`

- VERSION - The version number of the target library. For example, 2.3.1.

## Building a Qt Designer Plugin

*Qt Designer* plugins are built using a specific set of configuration settings that depend on the way Qt was configured for your system. For convenience, these settings can be enabled by adding to the QT variable. For example:`designer`

```
QT          += widgets designer
```

See the Qt Designer Examples for more examples of plugin-based projects.

# Building and Installing in Debug and Release Modes

Sometimes, it is necessary to build a project in both debug and release modes. Although the CONFIG variable can hold both and options, only the option that is specified last is applied.`debugrelease`

variable:`debug_and_releaseCONFIG`

```
CONFIG += debug_and_release

CONFIG(debug, debug|release) {
    TARGET = debug_binary
} else {
    TARGET = release_binary
}
```

The scope in the above snippet modifies the build target in each mode to ensure that the resulting targets have different names. Providing different names for targets ensures that one will not overwrite the other.

When qmake processes the project file, it will generate a Makefile rule to allow the project to be built in both modes. This can be invoked in the following way:

```
make all
```

The option can be added to the variable in the project file to ensure that the project is built in both modes by default:`build_allCONFIG`

```
CONFIG += build_all
```

This allows the Makefile to be processed using the default rule:

```
make
```

## Installing in Both Modes

The option also ensures that both versions of the target will be installed when the installation rule is invoked:`build_all`

```
make install
```

It is possible to customize the names of the build targets depending on the target platform. For example, a library or plugin may be named using a different convention on Windows from the one used on Unix platforms:

```
CONFIG(debug, debug|release) {
    mac: TARGET = $$join(TARGET,,,_debug)
    win32: TARGET = $$join(TARGET,,d)
```

The default behavior in the above snippet is to modify the name used for the build target when building in debug mode. An clause could be added to the scope to do the same for release mode. Left as it is, the target name remains unmodified.`else`

‹ Creating Project Files

Running qmake ›