高级用法

添加新的配置功能

qmake 允许您创建自己的，这些名称可以通过将项目文件的名称添加到 CONFIG 变量指定的值列表中来包含在项目文件中。功能是文件中自定义函数和定义的集合，这些文件可以驻留在许多标准目录之一中。这些目录的位置在多个位置定义，qmake 在查找文件时将按以下顺序检查每个目录：features.prf.prf

1. 在环境变量中列出的目录中，该目录包含由平台的路径列表分隔符分隔的目录列表（Unix 为冒号，Windows 为分号）。QMAKEFEATURES
2. 在属性变量中列出的目录中，该属性变量包含由平台的路径列表分隔符分隔的目录列表。QMAKEFEATURES
3. 位于目录中的功能目录中。 目录可以位于环境变量中列出的任何目录的下面，该环境变量包含由平台的路径列表分隔符分隔的目录列表。例如：。mkspecsmkspecsQMAKEPATH$QMAKEPATH/mkspecs/<features>
4. 位于 QMAKESPEC 环境变量提供的目录下的功能目录中。例如：。$QMAKESPEC/<features>
5. 位于该目录中的功能目录中。例如：。data_install/mkspecsdata_install/mkspecs/<features>
6. 在作为环境变量指定的目录的同级存在的功能目录中。例如：。QMAKESPEC$QMAKESPEC/../<features>

在以下功能目录中搜索功能文件：

1. features/unix、或，具体取决于所使用的平台features/win32features/macx
2. features/

例如，请考虑项目文件中的以下分配：

```
CONFIG += myfeatures
```

添加变量后，qmake 将在解析完项目文件后，在上面列出的位置搜索文件。在Unix系统上，它将查找以下文件：CONFIGmyfeatures.prf

1. $QMAKEFEATURES/myfeatures.prf（对于环境变量中列出的每个目录）QMAKEFEATURES
2. $$QMAKEFEATURES/myfeatures.prf（对于属性变量中列出的每个目录）QMAKEFEATURES
3. myfeatures.prf（在项目的根目录中）。项目根目录由顶级文件确定。但是，如果将文件放在子目录或子项目的目录中，则项目根目录将成为子目录本身。.pro.qmake.cache
4. $QMAKEPATH/mkspecs/features/unix/myfeatures.prf和（对于环境变量中列出的每个目录）$QMAKEPATH/mkspecs/features/myfeatures.prfQMAKEPATH
5. $QMAKESPEC/features/unix/myfeatures.prf和$QMAKESPEC/features/myfeatures.prf
6. data_install/mkspecs/features/unix/myfeatures.prf和data_install/mkspecs/features/myfeatures.prf
7. $QMAKESPEC/../features/unix/myfeatures.prf和$QMAKESPEC/../features/myfeatures.prf

> **注意：** 文件的名称必须为小写。.prf

安装文件

在Unix上，通常也使用构建工具来安装应用程序和库;例如，通过调用 。出于这个原因，qmake有一个的概念，一个对象，它包含有关项

```
documentation.path = /usr/local/program/doc
documentation.files = docs/*
```

成员通知 qmake 应将文件安装在 （路径成员） 中，并且成员指定应复制到安装目录的文件。在这种情况下，目录中的所有内容都将复制到 。path/usr/local/program/docfilesdocs/usr/local/program/doc

一旦安装集被完全描述，你可以用如下行将其附加到安装列表中：

```
INSTALLS += documentation
```

qmake 将确保将指定的文件复制到安装目录。如果需要对此过程进行更多控制，还可以为对象的成员提供定义。例如，以下行告诉 qmake 为此安装集执行一系列命令：extra

```
unix:documentation.extra = create_docs; mv master.doc toc.doc
```

作用域确保这些特定命令仅在 Unix 平台上执行。可以使用其他作用域规则定义其他平台的相应命令。unix

在执行对象的其他成员中的指令之前，将执行成员中指定的命令。extra

如果将内置安装集附加到变量中，并且未指定 或 成员，qmake 将决定需要为您复制的内容。目前，支持 和 安装集。例如：INSTALLSfilesextratargetdlltarget

```
target.path = /usr/local/myprogram
INSTALLS += target
```

In the above lines, qmake knows what needs to be copied, and will handle the installation process automatically.

Adding Custom Targets

qmake tries to do everything expected of a cross-platform build tool. This is often less than ideal when you really need to run special platform-dependent commands. This can be achieved with specific instructions to the different qmake backends.

Customization of the Makefile output is performed through an object-style API as found in other places in qmake. Objects are defined automatically by specifying their *members*. For example:

```
mytarget.target = .buildfile
mytarget.commands = touch $$mytarget.target
mytarget.depends = mytarget2

mytarget2.commands = @echo Building $$mytarget.target
```

The definitions above define a qmake target called , containing a Makefile target called which in turn is generated with the command. Finally, the member specifies that depends on , another target that is defined afterwards. is a dummy target. It is only defined to echo some text to the console.mytarget.buildfiletouch.dependsmytargetmytarget2mytarget2

The final step is to use the variable to instruct qmake that this object is a target to be built:QMAKE_EXTRA_TARGETS

```
QMAKE_EXTRA_TARGETS += mytarget mytarget2
```

This is all you need to do to actually build custom targets. Of course, you may want to tie one of these targets to the qmake build target. To do

Member	Description
commands	The commands for generating the custom build target.
CONFIG	Specific configuration options for the custom build target. Can be set to to indicate that rules should be created in the Makefile to call the relevant target inside the sub-target specific Makefile. This member defaults to creating an entry for each of the sub-targets.`recursive`
depends	The existing build targets that the custom build target depends on.
recurse	Specifies which sub-targets should be used when creating the rules in the Makefile to call in the sub-target specific Makefile. This member is used only when is set in . Typical values are "Debug" and "Release".`recursiveCONFIG`
recurse_target	Specifies the target that should be built via the sub-target Makefile for the rule in the Makefile. This member adds something like . This member is used only when is set in .`$(MAKE) -f Makefile.[subtarget] [recurse_target]recursiveCONFIG`
target	The name of the custom build target.

Adding Compilers

It is possible to customize qmake to support new compilers and preprocessors:

```
new_moc.output  = moc_${QMAKE_FILE_BASE}.cpp
new_moc.commands = moc ${QMAKE_FILE_NAME} -o ${QMAKE_FILE_OUT}
new_moc.depend_command = g++ -E -M ${QMAKE_FILE_NAME} | sed "s,^.*: ,,"
new_moc.input = NEW_HEADERS
QMAKE_EXTRA_COMPILERS += new_moc
```

With the above definitions, you can use a drop-in replacement for moc if one is available. The command is executed on all arguments given to the variable (from the member), and the result is written to the file defined by the member. This file is added to the other source files in the project. Additionally, qmake will execute to generate dependency information, and place this information in the project as well.`NEW_HEADERSinputoutputdepend_command`

Custom compiler specifications support the following members:

Member	Description
commands	The commands used for for generating the output from the input.
CONFIG	Specific configuration options for the custom compiler. See the CONFIG table for details.
depend_command	Specifies a command used to generate the list of dependencies for the output.
dependency_type	Specifies the type of file the output is. If it is a known type (such as TYPE_C, TYPE_UI, TYPE_QRC), it is handled as one of those type of files.
depends	Specifies the dependencies of the output file.
input	The variable that specifies the files that should be processed with the custom compiler.
name	A description of what the custom compiler is doing. This is only used in some backends.
output	The filename that is created from the custom compiler.
output_function	Specifies a custom qmake function that is used to specify the filename to be created.
variables	Indicates that the variables specified here are replaced with $(QMAKE_COMP_VARNAME) when referred to in the pro file as $(VARNAME).
variable_out	The variable that the files created from the output should be added to.

The CONFIG member supports the following options:

Option	Description

explicit_dependencies	The dependencies for the output only get generated from the depends member and from nowhere else.
dep_existing_only	Every dependency that is a result of .depend_command is checked for existence. Non-existing dependencies are ignored. This value was introduced in Qt 5.13.2.
dep_lines	The output from the .depend_command is interpreted to be one file per line. The default is to split on whitespace and is maintained only for backwards compatibility reasons.
no_link	Indicates that the output should not be added to the list of objects to be linked in.

Library Dependencies

Often when linking against a library, qmake relies on the underlying platform to know what other libraries this library links against, and lets the platform pull them in. In many cases, however, this is not sufficient. For example, when statically linking a library, no other libraries are linked to, and therefore no dependencies to those libraries are created. However, an application that later links against this library will need to know where to find the symbols that the static library will require. qmake attempts to keep track of the dependencies of a library, where appropriate, if you explicitly enable tracking.

The first step is to enable dependency tracking in the library itself. To do this you must tell qmake to save information about the library:

```
CONFIG += create_prl
```

This is only relevant to the template, and will be ignored for all others. When this option is enabled, qmake will create a file ending in .prl which will save some meta-information about the library. This metafile is just like an ordinary project file, but only contains internal variable declarations. When installing this library, by specifying it as a target in an INSTALLS declaration, qmake will automatically copy the .prl file to the installation path.`lib`

The second step in this process is to enable reading of this meta information in the applications that use the static library:

```
CONFIG += link_prl
```

When this is enabled, qmake will process all libraries linked to by the application and find their meta-information. qmake will use this to determine the relevant linking information, specifically adding values to the application project file's list of DEFINES as well as LIBS. Once qmake has processed this file, it will then look through the newly introduced libraries in the variable, and find their dependent .prl files, continuing until all libraries have been resolved. At this point, the Makefile is created as usual, and the libraries are linked explicitly against the application.`LIBS`

The .prl files should be created by qmake only, and should not be transferred between operating systems, as they may contain platform-dependent information.

‹ qmake Language

Using Precompiled Headers ›