语言

许多 qmake 项目文件使用 和 定义的列表，简单地描述项目使用的源文件和头文件。qmake 还提供了其他运算符、函数和作用域，可用于处理变量声明中提供的信息。这些高级功能允许从单个项目文件为多个平台生成生成文件。`name = valuename += value`

运营商

在许多项目文件中，赋值 （） 和追加 （） 运算符可用于包含有关项目的所有信息。典型的使用模式是为变量分配一个值列表，并根据各种测试的结果追加更多值。由于 qmake 使用默认值定义某些变量，因此有时需要使用 remove （） 运算符来筛选出不需要的值。以下各节介绍如何使用运算符操作变量的内容。`=+=-=`

赋值

运算符为变量赋值：`=`

```
TARGET = myapp
```

上面的行将 TARGET 变量设置为 。这将覆盖以前为 设置的任何值。`myappTARGETmyapp`

追加值

该运算符将新值追加到变量中的值列表中：`+=`

```
DEFINES += USE_MY_STUFF
```

上面的行附加到要放入生成的 Makefile 中的预处理器定义列表中。`USE_MY_STUFF`

删除值

该运算符从变量的值列表中删除一个值：`-=`

```
DEFINES -= USE_MY_STUFF
```

添加唯一值

运算符将值添加到变量中的值列表中，但仅当该值尚不存在时才添加。这可以防止值多次包含在变量中。例如：*=

```
DEFINES *= USE_MY_STUFF
```

在上面的行中，如果尚未定义，则只会将其添加到预处理器定义的列表中。请注意，unique () 函数也可用于确保变量仅包含每个值的一个实例。USE_MY_STUFF

替换值

该运算符将与正则表达式匹配的任何值替换为指定的值：~=

```
DEFINES ~= s/QT_[DT].+/QT
```

在上面的行中，列表中以 开头或替换为 的任何值。QT_DQT_TQT

可变扩展

运算符用于提取变量的内容，可用于在变量之间传递值或将其提供给函数：$$

```
EVERYTHING = $$SOURCES $$HEADERS
message("The project contains the following files:")
message($$EVERYTHING)
```

变量可用于存储环境变量的内容。这些可以在运行 qmake 时进行评估，也可以包含在生成的 Makefile 中，以便在构建项目时进行评估。

要在运行 qmake 时获取环境值的内容，请使用运算符：$$(...)

```
DESTDIR = $$(PWD)
message(The project will be installed in $$DESTDIR)
```

In the above assignment, the value of the environment variable is read when the project file is processed.PWD

To obtain the contents of an environment value at the time when the generated Makefile is processed, use the operator:$(...)

```
DESTDIR = $$(PWD)
message(The project will be installed in $$DESTDIR)
```

```
message(when the Makefile is processed.)
```

In the above assignment, the value of is read immediately when the project file is processed, but is assigned to in the generated Makefile. This makes the build process more flexible as long as the environment variable is set correctly when the Makefile is processed.PWD$(PWD)DESTDIR

Accessing qmake Properties

The special operator can be used to access qmake properties:$$[...]

```
message(Qt version: $$[QT_VERSION])
message(Qt is installed in $$[QT_INSTALL_PREFIX])
message(Qt resources can be found in the following locations:)
message(Documentation: $$[QT_INSTALL_DOCS])
message(Header files: $$[QT_INSTALL_HEADERS])
message(Libraries: $$[QT_INSTALL_LIBS])
message(Binary files (executables): $$[QT_INSTALL_BINS])
message(Plugins: $$[QT_INSTALL_PLUGINS])
message(Data files: $$[QT_INSTALL_DATA])
message(Translation files: $$[QT_INSTALL_TRANSLATIONS])
message(Settings: $$[QT_INSTALL_CONFIGURATION])
message(Examples: $$[QT_INSTALL_EXAMPLES])
```

For more information, see Configuring qmake.

The properties accessible with this operator are typically used to enable third party plugins and components to be integrated in Qt. For example, a *Qt Designer* plugin can be installed alongside *Qt Designer*'s built-in plugins if the following declaration is made in its project file:

```
target.path = $$[QT_INSTALL_PLUGINS]/designer
INSTALLS += target
```

Scopes

Scopes are similar to statements in procedural programming languages. If a certain condition is true, the declarations inside the scope are processed.if

Scope Syntax

Scopes consist of a condition followed by an opening brace on the same line, a sequence of commands and definitions, and a closing brace on a new line:

```
<condition> {
    <command or definition>
    ...
}
```

The opening brace *must be written on the same line as the condition*. Scopes may be concatenated to include more than one condition, as described in the following sections.

Scopes and Conditions

A scope is written as a condition followed by a series of declarations contained within a pair of braces. For example:

```
win32 {
    SOURCES += paintwidget_win.cpp
}
```

The above code will add the file to the sources listed in the generated Makefile when building for a Windows platform. When building for other platforms, the define will be ignored.`paintwidget_win.cpp`

The conditions used in a given scope can also be negated to provide an alternative set of declarations that will be processed only if the original condition is false. For example, to process something when building for all platforms *except* Windows, negate the scope like this:

```
!win32 {
    SOURCES -= paintwidget_win.cpp
}
```

Scopes can be nested to combine more than one condition. For instance, to include a particular file for a certain platform only if debugging is enabled, write the following:

```
macx {
    CONFIG(debug, debug|release) {
        HEADERS += debugging.h
    }
}
```

To save writing many nested scopes, you can nest scopes using the operator. The nested scopes in the above example can be rewritten in the following way: :

```
macx:CONFIG(debug, debug|release) {
    HEADERS += debugging.h
}
```

You may also use the operator to perform single line conditional assignments. For example: :

```
win32:DEFINES += USE_MY_STUFF
```

behaves like a logical AND operator, joining together a number of conditions, and requiring all of them to be true.USE_MY_STUFF:

There is also the operator to act like a logical OR operator, joining together a number of conditions, and requiring only one of them to be true.|

```
win32|macx {
    HEADERS += debugging.h
}
```

If you need to mix both operators, you can use the function to specify operator precedence.if

```
if(win32|macos):CONFIG(debug, debug|release) {
    # Do something on Windows and macOS,
    # but only for the debug configuration.
}
win32|if(macos:CONFIG(debug, debug|release)) {
    # Do something on Windows (regardless of debug or release)
    # and on macOS (only for debug).
}
```

The condition accepts the wildcard character to match a family of values or mkspec names.CONFIG

```
win32-* {
    # Matches every mkspec starting with "win32-"
    SOURCES += win32_specific.cpp
}
```

Note: Historically, checking the mkspec name with wildcards like above was qmake's way to check for the platform. Nowadays, we recommend to use values that are defined by the mkspec in the variable.QMAKE_PLATFORM

You can also provide alternative declarations to those within a scope by using an scope. Each scope is processed if the conditions for the preceding scopes are false. This allows you to write complex tests when combined with other scopes (separated by the operator as above). For example:elseelse:

```
win32:xml {
    message(Building for Windows)
    SOURCES += xmlhandler_win.cpp
} else:xml {
    SOURCES += xmlhandler.cpp
} else {
    message("Unknown configuration")
}
```

Configuration and Scopes

The values stored in the CONFIG variable are treated specially by qmake. Each of the possible values can be used as the condition for a scope. For example, the list of values held by can be extended with the value:CONFIGopengl

```
CONFIG += opengl
```

As a result of this operation, any scopes that test for will be processed. We can use this feature to give the final executable an appropriate name:opengl

```
opengl {
    TARGET = application-gl
} else {
    TARGET = application
}
```

This feature makes it easy to change the configuration for a project without losing all the custom settings that might be needed for a specific configuration. In the above code, the declarations in the first scope are processed, and the final executable will be called . However, if is not specified, the declarations in the second scope are processed instead, and the final executable will be called .application-glopenglapplication

Since it is possible to put your own values on the line, this provides you with a convenient way to customize project files and fine-tune the generated Makefiles.CONFIG

Platform Scope Values

In addition to the , , and values used in many scope conditions, various other built-in platform and compiler-specific values can be tested with scopes. These are based on platform specifications provided in Qt's directory. For example, the following lines from a project file show the current specification in use and test for the specification:win32macxunixmkspecslinux-g++

```
message($$QMAKESPEC)

linux-g++ {
    message(Linux)
}
```

You can test for any other platform-compiler combination as long as a specification exists for it in the directory.mkspecs

Variables

Many of the variables used in project files are special variables that qmake uses when generating Makefiles, such as DEFINES, SOURCES, and HEADERS. In addition, you can create variables for your own use. qmake creates new variables with a given name when it encounters an assignment to that name. For example:

There are no restricitions on what you do to your own variables, as qmake will ignore them unless it needs to evaluate them when processing a scope.

You can also assign the value of a current variable to another variable by prefixing $$ to the variable name. For example:

```
MY_DEFINES = $$DEFINES
```

Now the MY_DEFINES variable contains what is in the DEFINES variable at this point in the project file. This is also equivalent to:

```
MY_DEFINES = $${DEFINES}
```

The second notation allows you to append the contents of the variable to another value without separating the two with a space. For example, the following will ensure that the final executable will be given a name that includes the project template being used:

```
TARGET = myproject_$${TEMPLATE}
```

Replace Functions

qmake provides a selection of built-in functions to allow the contents of variables to be processed. These functions process the arguments supplied to them and return a value, or list of values, as a result. To assign a result to a variable, use the operator with this type of function as you would to assign contents of one variable to another:$$

```
HEADERS = model.h
HEADERS += $$OTHER_HEADERS
HEADERS = $$unique(HEADERS)
```

This type of function should be used on the right-hand side of assignments (that is, as an operand).

You can define your own functions for processing the contents of variables as follows:

```
defineReplace(functionName){
    #function code
}
```

The following example function takes a variable name as its only argument, extracts a list of values from the

```
defineReplace(headersAndSources) {
    variable = $$1
    names = $$eval($$variable)
    headers =
    sources =

    for(name, names) {
        header = $${name}.h
        exists($$header) {
            headers += $$header
        }
        source = $${name}.cpp
        exists($$source) {
            sources += $$source
        }
    }
    return($$headers $$sources)
}
```

Test Functions

qmake provides built-in functions that can be used as conditions when writing scopes. These functions do not return a value, but instead indicate *success* or *failure*:

```
count(options, 2) {
    message(Both release and debug specified.)
}
```

This type of function should be used in conditional expressions only.

It is possible to define your own functions to provide conditions for scopes. The following example tests whether each file in a list exists and returns true if they all exist, or false if not:

```
defineTest(allFiles) {
    files = $$ARGS

    for(file, files) {
        !exists($$file) {
            return(false)
        }
    }
    return(true)
}
```

‹ Platform Notes

Advanced Usage ›