qmake 用户手册

第一章 qmake 的介绍

qmake 的介绍

*qmake*是Trolltech公司创建的用来为不同的平台和编译器书写Makefile的工具。

手写Makefile是比较困难并且容易出错的，尤其是需要给不同的平台和编译器组合写几个Makefile。使用*qmake*，开发者创建一个简单的“项目”文件并且运行*qmake*生成适当的Makefile。*qmake*会注意所有的编译器和平台的依赖性，可以把开发者解放出来只关心他们的代码。Trolltech公司使用*qmake*作为Qt库和Qt所提供的工具的主要连编工具。

*qmake*也注意了Qt的特殊需求，可以自动的包含moc和*uic*的连编规则。

第二章 安装 qmake

安装 qmake

当Qt被连编的时候，默认情况下*qmake*也会被连编。

这一部分解释如何手工连编*qmake*。如果你已经有了*qmake*，可以跳过这里，请看10 分钟学会使用qmake。

手动安装 qmake

在手工连编 Qt 之前，下面这些环境变量必须被设置：

- QMAKESPEC
  这个必须设置为你所使用的系统的平台和编译器的组合。
  举例来说，加入你使用的是Windows和Microsoft Visual Studio，你应该把环境变量设置为*win32-msvc*。如果你使用的是Solaris和g++，你应该把环境变量设置为*solaris-g++*。

  当你在设置 QMAKESPEC 时，可以从下面的可能的环境变量列表中进行选择：

  aix-64 hpux-cc irix-032 netbsd-g++ solaris-cc unixware7-g++ aix-g++ hpux-g++ linux-cxx openbsd-g++ solaris-g++ win32-borland aix-xlc hpux-n64 linux-g++ openunix-cc sunos-g++ win32-g++ bsdi-g++ hpux-o64 linux-icc qnx-g++ tru64-cxx win32-msvc dgux-g++ hurd-g++ linux-kcc reliant-64 tru64-g++ win32-watc freebsd-g++ irix-64 macx-pbuilder reliant-cds ultrix-g++ win32-visa hpux-acc irix-g++ macx-g++ sco-g++ unixware-g hpux-acc irix-n32 solaris-64 unixware7-cc

envvar 是下面之一时，环境变量应该被设置到 qws/envvar：

linux-arm-g++ linux-generic-g++ linux-mips-g++ linux-x86-g++ linux-freebsd-g++ linux-ipaq-g++ linux-solaris-g++ qnx-rtp-g++

- QTDIR
  这个必须设置到Qt被（或者将被）安装到的地方。比如，*c:\qt*和*/local/qt*。

一旦环境变量被设置到qmake目录，*$QTDIR/qmake*，比如*C:\qt\qmake*，现在根据你的编译器运行*make*或者*nmake*。

当编译完成时，*qmake*已经可以使用了。

第三章 10 分钟学会使用 qmake

1 创建一个项目文件

*qmake*使用储存在项目（.pro）文件中的信息来决定Makefile文件中该生成什么。

一个基本的项目文件包含关于应用程序的信息，比如，编译应用程序需要哪些文件，并且使用哪些配置设置。

这里是一个简单的示例项目文件：

```
SOURCES = hello.cpp
HEADERS = hello.h
CONFIG += qt warn_on release
```

我们将会提供一行一行的简要解释，具体细节将会在手册的后面的部分解释。

```
SOURCES = hello.cpp
```

这一行指定了实现应用程序的源程序文件。在这个例子中，恰好只有一个文件，*hello.cpp*。大部分应用程序需要多个文件，这种情况下可以把文件列在一行中，以空格分隔，就像这样：

```
SOURCES = hello.cpp main.cpp
```

另一种方式，每一个文件可以被列在一个分开的行里面，通过反斜线另起一行，就像这样：

```
SOURCES = hello.cpp \
          main.cpp
```

一个更冗长的方法是单独地列出每一个文件，就像这样：

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

这种方法中使用“+=”比“=”更安全，因为它只是向已有的列表中添加新的文件，而不是替换整个列表。

HEADERS 这一行中通常用来指定为这个应用程序创建的头文件，举例来说：

```
HEADERS += hello.h
```

列出源文件的任何一个方法对头文件也都适用。

CONFIG这一行是用来告诉*qmake*关于应用程序的配置信息。

```
CONFIG += qt warn_on release
```

在这里使用“+=”，是因为我们添加我们的配置选项到任何一个已经存在中。这样做比使用“=”那样替换已经指定的所有选项是更安全的。

CONFIG一行中的*qt*部分告诉*qmake*这个应用程序是使用Qt来连编的。这也就是说*qmake*在连接和为编译添加所需的包含路径的时候会考虑到Qt库的。

CONFIG一行中的*warn_on*部分告诉*qmake*要把编译器设置为输出警告信息的。

CONFIG一行中的*release*部分告诉*qmake*应用程序必须被连编为一个发布的应用程序。在开发过程中，程序员也可以使用*debug*来替换*release*，稍后会讨论这里的。

项目文件就是纯文本（比如，可以使用像记事本、vim 和 xemacs 这些编辑器）并且必须存为“.pro”扩展名。应用程序的执行文件的名称必须和项目文件的名称一样，但是扩展名是跟着平台而改变的。举例来说，一个叫做“hello.pro”的项目文件将会在 Windows 下生成“hello.exe”，而在 Unix 下生成“hello”。

2 生成 Makefile

当你已经创建好你的项目文件，生成 Makefile 就很容易了，你所要做的就是先到你所生成的项目文件那里然后输入：

Makefile 可以像这样由“.pro”文件生成：

```
qmake -o Makefile hello.pro
```

对于Visual Studio的用户，*qmake*也可以生成“.dsp”文件，例如：

```
qmake -t vcapp -o hello.dsp hello.pro
```

第四章 qmake 教程

1 qmake 教程介绍

这个教程可以教会你如何使用*qmake*。我们建议你看完这个教程之后读一下*qmake*手册。

2 开始很简单

让我们假设你已经完成了你的应用程序的一个基本实现，并且你已经创建了下述文件：

- hello.cpp
- hello.h
- main.cpp

你可以在*qt/qmake/example*中发现这些文件。你对这个应用程序的配置仅仅知道的另一件事是它是用Qt写的。首先，使用你所喜欢的纯文本编辑器，在*qt/qmake/tutorial*中创建一个叫做*hello.pro*的文件。你所要做的第一件事是添加一些行来告诉*qmake*关于你所开发的项目中的源文件和头文件这一部分。

我们先把源文件添加到项目文件中。为了做到这点，你需要使用SOURCES变量。只要用*SOURCES* +=来开始一行，并且把hello.cpp放到它后面。你需要写成这样：

```
SOURCES += hello.cpp
```

我们对项目中的每一个源文件都这样做，直到结束：

```
SOURCES += hello.cpp
SOURCES += main.cpp
```

如果你喜欢使用像 Make 一样风格的语法，你也可以写成这样，一行写一个源文件，并用反斜线结尾，然后再起新的一行：

```
SOURCES = hello.cpp \
          main.cpp
```

现在源文件已经被列到项目文件中了，头文件也必须添加。添加的方式和源文件一样，除了变量名是 HEADERS。

当你做完这些时，你的项目文件就像现在这样：

```
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

目标名称是自动设置的，它被设置为和项目文件一样的名称，但是为了适合平台所需要的后缀。举例来说，加入项目文件叫做“hello.pro”，在 Windows 上的目标名称应该是“hello.exe”，在 Unix 上应该是“hello”。如果你想设置一个不同的名字，你可以在项目文件中设置它：

```
TARGET = helloworld
```

最后一步是设置*CONFIG*变量。因为这是一个Qt应用程序，我们需要把“qt”放到CONFIG这一行中，这样*qmake*才会在连接的时候添加相关的库，并且保证*moc*和*uic*的连编行也被包含到Makefile中。

最终完成的项目文件应该是这样的：

```
CONFIG += qt
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

你现在可以使用*qmake*来为你的应用程序生成Makefile。在你的应用程序目录中，在命令行下输入：

```
qmake -o Makefile hello.pro
```

然后根据你所使用的编译器输入*make*或者*nmake*。

3 使应用程序可以调试

应用程序的发布版本不包含任何调试符号或者其它调试信息。在开发过程中，生成一个含有相关信息的应用程序的调试版本是很有用处的。通过在项目文件的 CONFIG 变量中添加“debug”就可以很简单地实现。

例如：

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

像前面一样使用*qmake*来生成一个Makefile并且你就能够调试你的应用程序了。

4 添加特定平台的源文件

在编了几个小时的程序之后，你也许开始为你的应用程序编写与平台相关的部分，并且决定根据平台的不同编写不同的代码。所以现在你有两个信文件要包含到你的项目文件中－*hello_win.cpp*和*hello_x11.cpp*。我们不能仅仅把这两个文件放到*SOURCES*变量中，因为那样的话会把这两个文件都加到Makefile中。所以我们在这里需要做的是根据*qmake*所运行的平台来使用相应的作用域来进行处理。

为 Windows 平台添加的依赖平台的文件的简单的作用域看起来就像这样：

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

所以如果*qmake*运行在Windows上的时候，它就会把*hello_win.cpp*添加到源文件列表中。如果*qmake*运行在其它平台上的时候，它会很简单地把这部分忽略。现在接下来我们要做的就是添加一个X11 依赖文件的作用域。

当你做完了这部分，你的项目文件应该和这样差不多：

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
```

像前面一样使用*qmake*来生成Makefile。

5 如果一个文件不存在，停止 qmake

如果某一个文件不存在的时候，你也许不想生成一个Makefile。我们可以通过使用exists()函数来检查一个文件是否存在。我们可以通过使用error()函数把正在运行的*qmake*停下来。这和作用域的工作方式一样。只要很简单地用这个函数来替换作用域条件。对main.cpp文件的检查就像这样：

```
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}
```

“!”用来否定这个测试，比如，如果文件存在，exists(main.cpp)是真，如果文件不存在，!exists(main.cpp)是真。

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}
```

像前面一样使用*qmake*来生成Makefile。如果你临时改变*main.cpp*的名称，你会看到信息，并且*qmake*会停止处理。

6 检查多于一个的条件

假设你使用Windows并且当你在命令行运行你的应用程序的时候你想能够看到qDebug()语句。除非你在连编你的程序的时候使用console设置，你不会看到输出。我们可以很容易地把*console*添加到CONFIG行中，这样在Windows下，Makefile就会有这个设置。但是如果告诉你我们只是想在当我们的应用程序运行在Windows下*并且*当*debug*已经在CONFIG行中的时候，添加*console*。这需要两个嵌套的作用域；只要生成一个作用域，然后在它里面再生成另一个。把设置放在最里面的作用域里，就像这样：

```
win32 {
    debug {
        CONFIG += console
    }
}
```

嵌套的作用域可以使用冒号连接起来，所以最终的项目文件看起来像这样：

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
!exists( main.cpp ) {
    error( "No main.cpp file found" )
```

```
}
win32:debug {
    CONFIG += console
}
```

就这些了！你现在已经完成了*qmake*的教程，并且已经准备好为你的开发项目写项目文件了。

第五章 qmake 概念

1 介绍 qmake

*qmake*是用来为不同的平台的开发项目创建makefile的Trolltech开发一个易于使用的工具。*qmake*简化了makefile的生成，所以为了创建一个makefile只需要一个只有几行信息的文件。*qmake*可以供任何一个软件项目使用，而不用管它是不是用Qt写的，尽管它包含了为支持Qt开发所拥有的额外的特征。

*qmake*基于一个项目文件这样的信息来生成makefile。项目文件可以由开发者生成。项目文件通常很简单，但是如果需要它是非常完善的。不用修改项目文件，*qmake*也可以为为Microsoft Visual Studio生成项目。

2 qmake 的概念

QMAKESPEC 环境变量

举例来说，如果你在Windows下使用Microsoft Visual Studio，然后你需要把QMAKESPEC环境变量设置为*win32-msvc*。如果你在Solaris上使用gcc，你需要把QMAKESPEC环境变量设置为*solaris-g++*。

在qt/mkspecs中的每一个目录里面，都有一个包含了平台和编译器特定信息的*qmake.conf*文件。这些设置适用于你要使用*qmake*的任何项目，请不要修改它，除非你是一个专家。例如，假如你所有的应用程序都必须和一个特定的库连接，你可以把这个信息添加到相应的*qmake.conf*文件中。

项目(.pro)文件

一个项目文件是用来告诉*qmake*关于为这个应用程序创建makefile所需要的细节。例如，一个源文件和头文件的列表、任何应用程序特定配置、例如一个必需要连接的额外库、或者一个额外的包含路径，都应该放到项目文件中。

“#”注释

你可以为项目文件添加注释。注释由“#”符号开始，一直到这一行的结束。

3 模板

模板变量告诉*qmake*为这个应用程序生成哪种makefile。下面是可供使用的选择：

- app - 建立一个应用程序的 makefile。这是默认值，所以如果模板没有被指定，这个将被使用。
- lib - 建立一个库的 makefile。
- vcapp - 建立一个应用程序的 Visual Studio 项目文件。
- vclib - 建立一个库的 Visual Studio 项目文件。
- subdirs - 这是一个特殊的模板，它可以创建一个能够进入特定目录并且为一个项目文件生成 makefile 并且为它调用 make 的 makefile。

“app”模板

“app”模板告诉*qmake*为建立一个应用程序生成一个makefile。当使用这个模板时，下面这些*qmake*系统变量是被承认的。你应该在你的.pro文件中使用它们来为你的应用程序指定特定信息。

- HEADERS - 应用程序中的所有头文件的列表。
- SOURCES - 应用程序中的所有源文件的列表。
- FORMS - 应用程序中的所有.ui文件（由*Qt 设计器*生成）的列表。
- LEXSOURCES - 应用程序中的所有 lex 源文件的列表。
- YACCSOURCES - 应用程序中的所有 yacc 源文件的列表。
- TARGET - 可执行应用程序的名称。默认值为项目文件的名称。（如果需要扩展名，会被自动加上。）
- DESTDIR - 放置可执行程序目标的目录。
- DEFINES - 应用程序所需的额外的预处理程序定义的列表。
- INCLUDEPATH - 应用程序所需的额外的包含路径的列表。
- DEPENDPATH - 应用程序所依赖的搜索路径。
- VPATH - 寻找补充文件的搜索路径。
- DEF_FILE - 只有 Windows 需要：应用程序所要连接的.def 文件。
- RC_FILE - 只有 Windows 需要：应用程序的资源文件。
- RES_FILE - 只有 Windows 需要：应用程序所要连接的资源文件。

你只需要使用那些你已经有值的系统变量，例如，如果你不需要任何额外的INCLUDEPATH，那么你就不需要指定它，*qmake*会为所需的提供默认值。例如，一个实例项目文件也许就像这样：

```
TEMPLATE = app
DESTDIR  = c:\helloapp
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
DEFINES += QT_DLL
CONFIG  += qt warn_on release
```

如果条目是单值的，比如template或者目的目录，我们是用“=”，但如果是多值条目，我们使用“+=”来为这个类型*添加*现有的条目。使用“=”会用新值替换原有的值，例如，如果我们写了DEFINES=QT_DLL，其它所有的定义都将被删除。

“lib”模板

“lib”模板告诉*qmake*为建立一个库而生成makefile。当使用这个模板时，除了“app”模板中提到系统变量，还有一个*VERSION*是被支持的。你需要在为库指定特定信息的.pro文件中使用它们。

- VERSION - 目标库的版本号，比如，2.3.1。

“subdirs”模板

“subdirs”模板告诉 qmake 生成一个 makefile，它可以进入到特定子目录并为这个目录中的项目文件生成 makefile 并且为它调用 make。

在这个模板中只有一个系统变量*SUBDIRS*可以被识别。这个变量中包含了所要处理的含有项目文件的子目录的列表。这个项目文件的名称是和子目录同名的，这样*qmake*就可以发现它。例如，如果子目里是“myapp”，那么在这个目录中的项目文件应该被叫做*myapp.pro*。

4 CONFIG 变量

配置变量指定了编译器所要使用的选项和所需要被连接的库。配置变量中可以添加任何东西，但只有下面这些选项可以被 qmake 识别。

下面这些选项控制着使用哪些编译器标志：

- release - 应用程序将以 release 模式连编。如果“debug”被指定，它将被忽略。
- debug - 应用程序将以 debug 模式连编。
- warn_on - 编译器会输出尽可能多的警告信息。如果“warn_off”被指定，它将被忽略。
- warn_off - 编译器会输出尽可能少的警告信息。

下面这些选项定义了所要连编的库/应用程序的类型：

- qt - 应用程序是一个 Qt 应用程序，并且 Qt 库将会被连接。
- thread - 应用程序是一个多线程应用程序。
- x11 - 应用程序是一个 X11 应用程序或库。
- windows - 只用于“app”模板：应用程序是一个 Windows 下的窗口应用程序。
- console - 只用于“app”模板：应用程序是一个 Windows 下的控制台应用程序。
- dll - 只用于“lib”模板：库是一个共享库（dll）。
- staticlib - 只用于“lib”模板：库是一个静态库。
- plugin - 只用于“lib”模板：库是一个插件，这将会使 dll 选项生效。

例如，如果你的应用程序使用 Qt 库，并且你想把它连编为一个可调试的多线程的应用程序，你的项目文件应该会有下面这行：

```
CONFIG += qt thread debug
```

注意，你必须使用“+=”，不要使用“=”，否则*qmake*就不能正确使用连编Qt的设置了，比如没法获得所编译的Qt库的类型了。

第六章 qmake 高级概念

1 qmake 高级概念

迄今为止，我们见到的*qmake*项目文件都非常简单，仅仅是一些*name = value*和*name += value*的列表行。*qmake*提供了很多更强大的功能，比如你可以使用一个简单的项目文件来为多个平台生成makefile。

2 操作符

到目前为止，你已经看到在项目文件中使用的=操作符和+=操作符。这里能够提供更多的可供使用的操作符，但是其中的一些需要谨慎地使用，因为它们也许会比你期待的改变的更多。

“=”操作符

这个操作符简单分配一个值给一个变量。使用方法如下：

```
TARGET = myapp
```

这将会设置TARGET变量为*myapp*。这将会删除原来对TARGET的任何设置。

“+=”操作符

这个操作符将会向一个变量的值的列表中添加一个值。使用方法如下：

```
DEFINES += QT_DLL
```

这将会把 QT_DLL 添加到被放到 makefile 中的预处理定义的列表中。

“-=”操作符

这个操作符将会从一个变量的值的列表中移去一个值。使用方法如下：

```
DEFINES -= QT_DLL
```

这将会从被放到 makefile 中的预处理定义的列表中移去 QT_DLL。

“*=”操作符

这个操作符仅仅在一个值不存在于一个变量的值的列表中的时候，把它添加进去。使用方法如下：

```
DEFINES *= QT_DLL
```

只用在 QT_DLL 没有被定义在预处理定义的列表中时，它才会被添加进去。

“~=”操作符

这个操作符将会替换任何与指定的值的正则表达式匹配的任何值。使用方法如下：

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

这将会用 QT 来替代任何以 QT_D 或 QT_T 开头的变量中的 QT_D 或 QT_T。

3 作用域

作用域和“if”语句很相似，如果某个条件为真，作用域中的设置就会被处理。作用域使用方法如下：

```
win32 {
    DEFINES += QT_DLL
}
```

上面的代码的作用是，如果在Windows平台上使用*qmake*，QT_DLL定义就会被添加到makefile中。如果在Windows平台以外的平台上使用*qmake*，这个定义就会被忽略。你也可以使用qmake执行一个单行的条件/任务，就像这样：

```
win32:DEFINES += QT_DLL
```

比如，假设我们想在*除了*Windows平台意外的所有平台处理些什么。我们想这样使用作用域来达到这种否定效果：

```
!win32 {
    DEFINES += QT_DLL
}
```

CONFIG 行中的任何条目也都是一个作用域。比如，你这样写：

```
CONFIG += warn_on
```

你将会得到一个称作“warn_on”的作用域。这样将会使在不丢失特定条件下可能所需的所有自定义设置的条件下，很容易地修改项目中的配置。因为你可能把你自己的值放到 CONFIG 行中，这将会为你的 makefile 而提供给你一个非常强大的配置工具。比如：

```
CONFIG += qt warn_on debug
debug {
    TARGET = myappdebug
}
release {
    TARGET = myapp
}
```

在上面的代码中，两个作用域被创建，它们依赖于CONFIG行中设置的是什么。在这个例子中，*debug*在CONFIG行中，所以TARGET变量被设置为*myappdebug*。如果*release*在CONFIG行中，那么TARGET变量将会被设置为*myapp*。

当然也可以在处理一些设置之前检查两个事物。例如，如果你想检查平台是否是 Windows 并且线程设置是否被设定，你可以这样写：

```
win32 {
    thread {
        DEFINES += QT_THREAD_SUPPORT
    }
}
```

为了避免写出许多嵌套作用域，你可以这样使用冒号来嵌套作用域：

```
win32:thread {
    DEFINES += QT_THREAD_SUPPORT
}
```

一旦一个测试被执行，你也许也要做 else/elseif 操作。这种情况下，你可以很容易地写出复杂的测试。这需要使用特殊的“else”作用域，它可以和其它作用域进行组合（也可以向上面一样使用冒号），比如：

```
win32:thread {
    DEFINES += QT_THREAD_SUPPORT
} else:debug {
    DEFINES += QT_NOTHREAD_DEBUG
} else {
    warning("Unknown configuration")
}
```

4 变量

到目前为止我们遇到的变量都是系统变量，比如*DEFINES*、*SOURCES*和*HEADERS*。你也可以为你自己创建自己的变量，这样你就可以在作用域中使用它们了。创建自己的变量很容易，只要命名它并且分配一些东西给它。比如：

```
MY_VARIABLE = value
```

现在你对你自己的变量做什么是没有限制的，同样地，*qmake*将会忽略它们，除非需要在一个作用域中考虑它们。

你也可以通过在其它任何一个变量的变量名前加$$来把这个变量的值分配给当前的变量。例如：

```
MY_DEFINES = $$DEFINES
```

现在 MY_DEFINES 变量包含了项目文件在这点时 DEFINES 变量的值。这也和下面的语句一样：

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

第二种方法允许你把一个变量和其它变量连接起来，而不用使用空格。*qmake*将允许一个变量包含任何东西（包括$(VALUE)，可以直接在makefile中直接放置，并且允许它适当地扩张，通常是一个环境变量）。无论如何，如果你需要立即设置一个环境变量，然后你就可以使用$$()方法。比如：

```
MY_DEFINES = $$(ENV_DEFINES)
```

这将会设置 MY_DEFINES 为环境变量 ENV_DEFINES 传递给.pro 文件地值。另外你可以在替换的变量里调用内置函数。这些函数（不会和下一节中列举的测试函数混淆）列出如下：

join(variablename, glue, before, after)

这将会在*variablename*的各个值中间加入glue。如果这个变量的值为非空，那么就会在值的前面加一个前缀*before*和一个后缀*after*。只有*variablename*是必须的字段，其它默认情况下为空串。如果你需要在*glue*、*before*或者*after*中使用空格的话，你必须提供它们。

member(variablename, position)

这将会放置*variablename*的列表中的*position*位置的值。如果*variablename*不够长，这将会返回一个空串。*variablename*是唯一必须的字段，如果没有指定位置，则默认为列表中的第一个值。

find(variablename, substr)

这将会放置*variablename*中所有匹配*substr*的值。*substr*也可以是正则表达式，而因此将被匹配。

```
MY_VAR = one two three four
MY_VAR2 = $$join(MY_VAR, ″ -L″, -L) -Lfive
MY_VAR3 = $$member(MY_VAR, 2) $$find(MY_VAR, t.*)
```

MY_VAR2 将会包含“-Lone -Ltwo -Lthree -Lfour -Lfive”，并且 MYVAR3 将会包含“three two three”。

system(program_and_args)

这将会返回程序执行在标准输出/标准错误输出的内容，并且正像平时所期待地分析它。比如你可以使用这个来询问有关平台的信息。

```
UNAME = $$system(uname -s)
contains( UNAME, [lL]inux ):message( This looks like Linux ($$UNAME) to me )
```

5 测试函数

*qmake*提供了可以简单执行，但强大测试的内置函数。这些测试也可以用在作用域中（就像上面一样），在一些情况下，忽略它的测试值，它自己使用测试函数是很有用的。

contains(variablename, value)

如果*value*存在于一个被叫做*variablename*的变量的值的列表中，那么这个作用域中的设置将会被处理。例如：

```
contains( CONFIG, thread ) {
    DEFINES += QT_THREAD_SUPPORT
}
```

如果*thread*存在于*CONFIG*变量的值的列表中时，那么QT_THREAD_SUPPORT将会被加入到*DEFINES*变量的值的列表中。

count(variablename, number)

如果*number*与一个被叫做*variablename*的变量的值的数量一致，那么这个作用域中的设置将会被处理。例如：

```
count( DEFINES, 5 ) {
    CONFIG += debug
}
```

error(string)

这个函数输出所给定的字符串，然后会使*qmake*退出。例如：

```
error( "An error has occured" )
```

文本“An error has occured”将会被显示在控制台上并且*qmake*将会退出。

exists(filename)

如果指定文件存在，那么这个作用域中的设置将会被处理。例如：

```
exists( /local/qt/qmake/main.cpp ) {
    SOURCES += main.cpp
}
```

如果*/local/qt/qmake/main.cpp*存在，那么main.cpp将会被添加到源文件列表中。

注意可以不用考虑平台使用“/”作为目录的分隔符。

include(filename)

项目文件在这一点时包含这个文件名的内容，所以指定文件中的任何设置都将会被处理。例如：

```
include( myotherapp.pro )
```

*myotherapp.pro*项目文件中的任何设置现在都会被处理。

isEmpty(variablename)

这和使用count(variablename, 0)是一样的。如果叫做*variablename*的变量没有任何元素，那么这个作用域中的设置将会被处理。例如：

```
isEmpty( CONFIG ) {
    CONFIG += qt warn_on debug
}
```

message(string)

这个函数只是简单地在控制台上输出消息。

```
message( "This is a message" )
```

文本“This is a message”被输出到控制台上并且对于项目文件的处理将会继续进行。

system(command)

特定指令被执行并且如果它返回一个 1 的退出值，那么这个作用域中的设置将会被处理。例如：

```
system( ls /bin ) {
    SOURCES += bin/main.cpp
    HEADERS += bin/main.h
}
```

所以如果命令*ls /bin*返回 1，那么*bin/main.cpp*将被添加到源文件列表中并且*bin/main.h*将被添加到头文件列表中。

infile(filename, var, val)

如果*filename*文件（当它被qmake自己解析时）包含一个值为*val*的变量*var*，那么这个函数将会返回成功。你也可以不传递第三个参数（*val*），这时函数将只测试文件中是否分配有这样一个变量*var*。