<https://www.cnblogs.com/Braveliu/p/5107550.html#top>

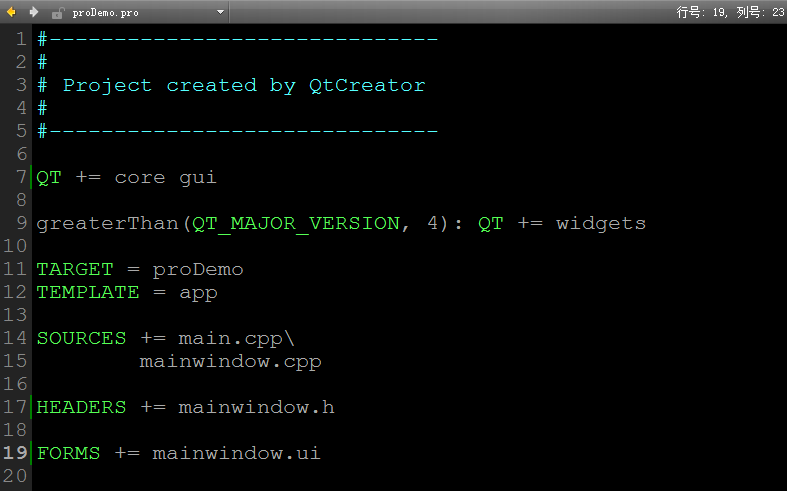
工作中，感觉pro文件的有些内容真不太懂，现系统性的学习一下。于此备录，分享共勉。

为了更好的理解，先创建一个简单的工程作为实践。

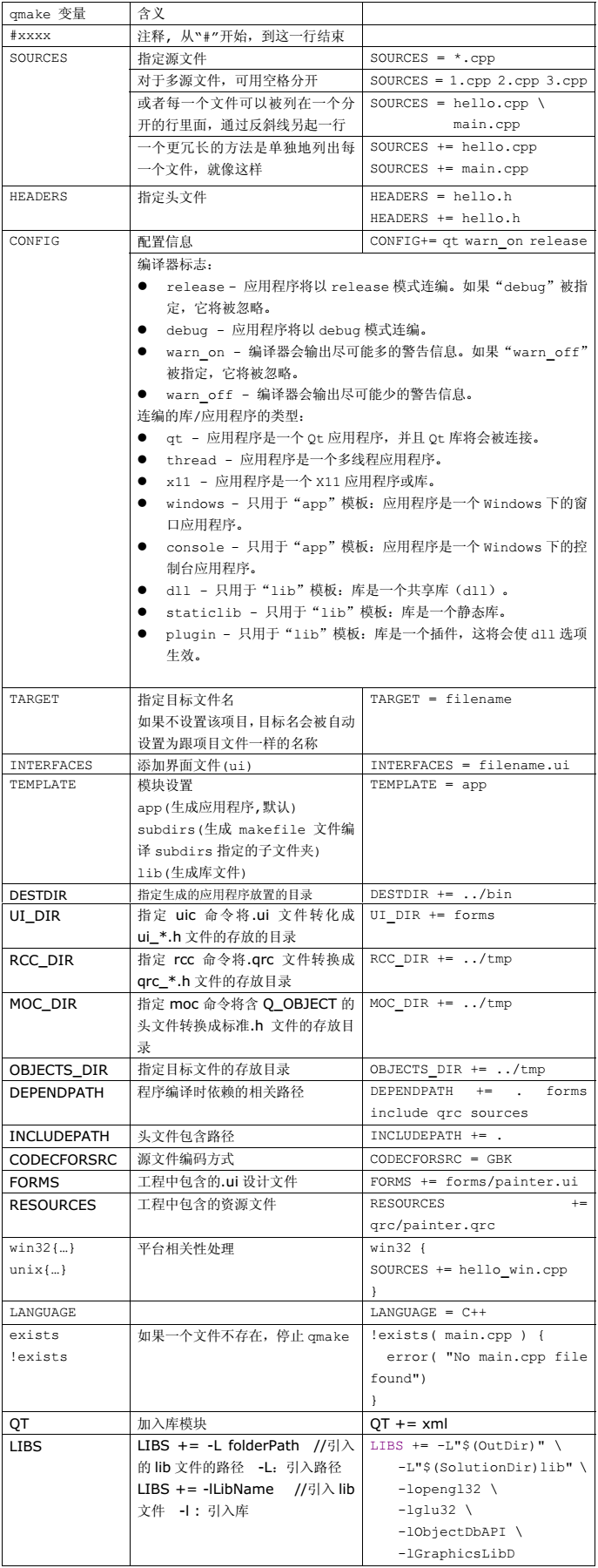
【1】创建一个pro文件

1.1 新建proDemo工程。步骤如下：Qt Creator--->New Project--->应用程序--->Qt Widgets Application--->名称为：proDemo（创建路径自己拟定，本地为F:\Source\proStudy）--->类信息保持不变--->完成

1.2 提取pro文件。在工程目录下，把名称为proDemo且类型为pro的文件找到，内容如下所示：



不懂？呵呵~ 我也不懂，先学习一下，pro文件分析如下：



【2】模板变量

2.1 模板变量 TEMPLATE

模板变量作用告诉qmake为这个应用程序具体生成哪种makefile。下面是模板变量可供选择的值：

[1]app 模板变量的默认值。建立一个应用程序的makefile。

[2]lib 建立一个库的makefile。

[3]vcapp 建立一个应用程序的Visual Studio项目文件。

[4]vclib 建立一个库的VisualStudio项目文件。

[5]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且为一个项目文件生成makefile，还能为它再调用make的makefile。

由以上分析可知，模板变量值不同，生成的makefile文件也会随之改变。那么，默认的同时是最常用的app值，模板请参见下文。

2.2 app模板

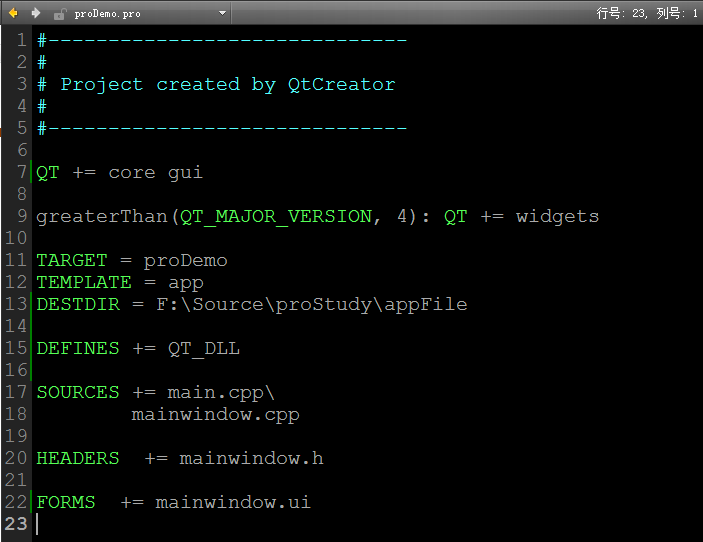
　　app模板告诉qmake为建立一个应用程序生成一个makefile。

当使用这个模板时，设置下面这些qmake系统变量值是有效的。可以在你的.pro文件中使用它们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会为所需的系统变量提供默认值。

例如，上例中的项目pro文件也可写成这样：



注意：如果条目是单值的，比如template或者目的目录DESTDIR（可执行文件或二进制文件的发布目录），我们是用“=”，但如果是多值条目，我们使用“+=”来为这个变量添加现有的条目值。

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DEFINES所有的条目值都将被删除并用QT\_DLL替代。

2.3 lib模板

　　lib模板告诉qmake为建立一个库而生成一个makefile。当使用这个模板时，除了app模板中提到的系统变量外，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。

2.4 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。

【3】配置变量CONFIG

　　配置变量CONFIG 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。

3.1 控制编译器

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（与release互斥）。
* debug\_and\_release - 工程同时用调试和发布模式编译。
* build\_all - 如果指定是debug\_and\_release模式，工程默认是同时用调试和发布模式编译。
* ordered - 使用subdirs模板时，本选项指定了子目录应该按照给出的顺序编译。
* 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（与warn\_on互斥）。

3.2 连编类型

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库将会被链接。
* 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：

https://images2015.cnblogs.com/blog/389111/201601/389111-20160108232439309-343983318.png

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。

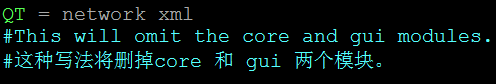
3.3 声明QT库模块

如果CONFIG变量值中包含了qt这个值，qmake支持了qt的程序（因为qmake也可以用在非qt程序的编译）这就要调整一些你程序中使用的qt的模块，而QT变量，正是达到这个目的的。QT是用来声明使用到的一些额外的模块，例如：通过下面的方法使得xml和网络模块有效：

https://images2015.cnblogs.com/blog/389111/201601/389111-20160108232452418-1796347592.png

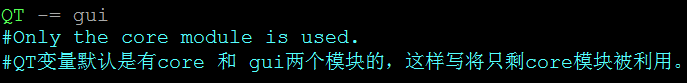
注意：默认情况下，qt包含了core 和 gui 模块，所以上面的声明仅仅是添加xml和网络模块到默认的列表中。

比如，下面的语句就是忽略了默认模块，当编译程序源码时候会导致错误：



假如你想编译一个不需要gui模块的工程，你需要用“-=”操作符来去除包含。

例如：下面的语句就是小型的Qt工程会被编译



下面的罗列显示了QT变量可以使用的选项，并解释了相应的特点：

* core (included by default)   QtCore module 核心模块
* gui (included by default)    QtGui module 界面模块
* network                      QtNetwork module 支持网络模块
* opengl　　　　               QtOpenGL module 支持opengl图像编程
* sql                          QtSql module 支持sql数据库驱动
* svg                          QtSvg module 支持svg矢量图形
* xml                          QtXml module 支持xml模块
* qt3support                   Qt3Support module 支持qt3类

注意：添加opengl到QT变量里面，等价于往CONFIG变量里面添加。

所以对于Qt应用程序来说，没必要同时往QT变量和CONFIG变量里面添加opengl选项。

3.4 关于 CONFIG（debug, debug|release）语法

CONFIG变量可以同时定义 debug 和 release，但只有一个处于active（当两个互斥的值都出现时，最后设置的处于active状态）

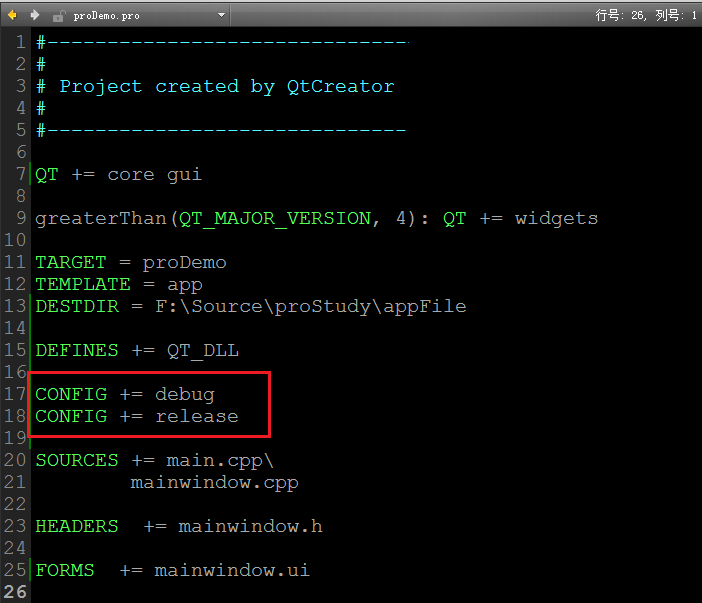
什么意思？怎么理解呢？请看如下：

https://images2015.cnblogs.com/blog/389111/201601/389111-20160108235812762-624480209.png

如上写法，release处于active状态。胡说吧！你咋知道呢？嗯哼？不信咱们用上面的工程测试一下。

[1] 还原测试现场。把工程文件夹proDemo同级的目录及文件删除干净（为了验证准确性，先还原测试实验现场，以下类似做法意义相同）。

[2] 修改内容。把工程中的pro类型文件改为下面的内容：



[3] 编译构建。首先把工程qmake一下，然后构建。

好，完成测试操作。激动人心的时刻到了！这个时候，我们会发现proDemo同级目录生成了appFile文件夹。

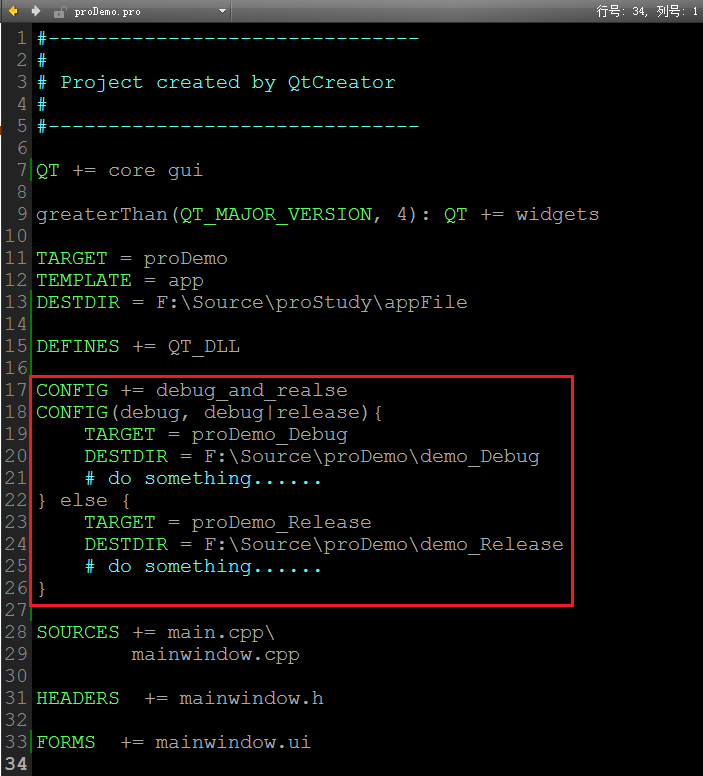
打开此文件夹，发现有两个文件：proDemo.exe 和 proDemo.exe.embed.manifest

坏了！从pro文件来看，我们debug和release版本生成的目标文件名称是相同的（即TARGET值）。肿么办呢？为了验证是release版本的可执行程序！

这样吧！我们在上级目录即proStudy文件夹中搜索\*pdb文件（调试debug版本必生成的文件），没有搜索到。OK！那就说明是Release版本。

其实，另外一种办法，我们会发现proDemo工程文件夹同级目录下也会生成一个build-proDemo-Desktop\_Qt\_5\_3\_MSVC2010\_OpenGL\_32bit-Debug名称的文件夹（说明一点：刚刚构建程序时QT Creator的模式是Debug），而这个文件夹中会有debug和release两个文件夹，你会发现debug文件是空的，而release文件夹中才有内容。这点也可以说明刚刚的确生成的是release版本。

但是，上面的写法，debug 和 release都可以通过测试，而且阅读比较费劲，反正总感觉迷惑性太强，如何处理呢？建议写法如下：



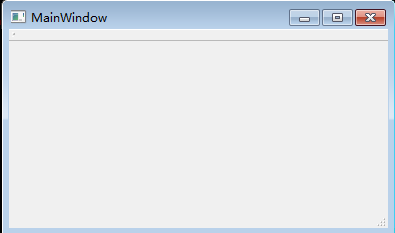
这种情况下，我们再来分析一下：

[1] 还原现场。清空proDemo工程目录下除过proDemo而外的其他文件夹（同上）。

[2] 切换模式。分别切换Debug和Release版本：



[3] 编译构建。各自执行qmake，并进行构建，再运行，可以看到同样的窗体：



好勒~ 尽管，看起来是同样的窗体，但是，要理解是两个完全不同版本的应用程序。

现在看看proDemo目录下会多出这样两个文件夹：demo\_Debug 和 demo\_Release，在其各自目录下，可以分别发现可执行程序proDemo\_Debug和proDemo\_Release，即刚刚两个窗体应用程序。

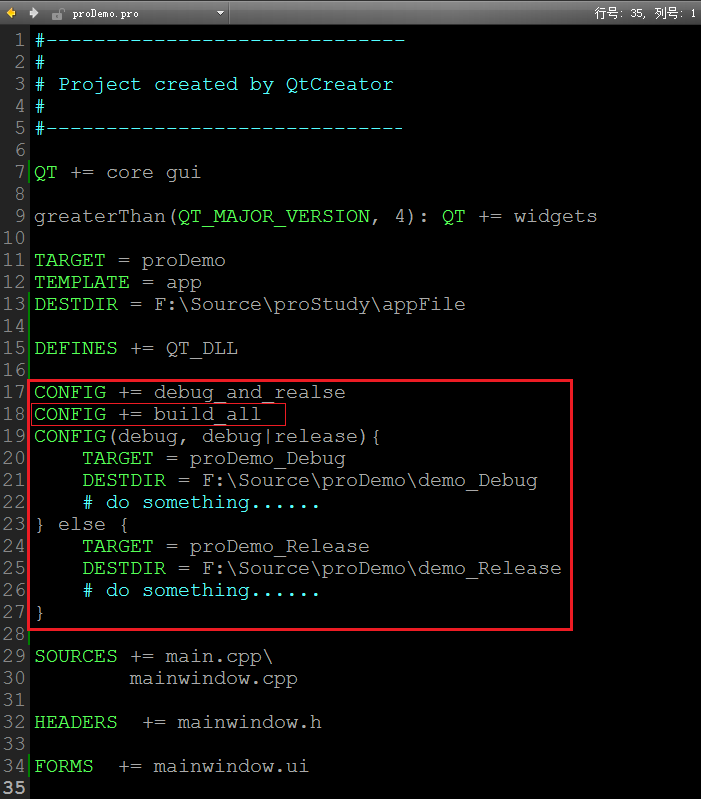
同时，会发现再没有生成appFile文件夹，这点也可以验证DESTDIR 后面的“=”与 “+=”的作用区别。

那么，再回头看 CONFIG（debug, debug|release）这种语法是什么含义呢？

注解：两个参数，前者是要判断active的选项，后者是互斥的选项的一个集合。

有人觉得难道这么麻烦？上面的例子中，生成两种版本的应用程序，我们通过对Qt Creator构建模式进行了切换，那么想直接生成呢？

可以直接加选项build\_all。比如，可以把pro文件改为这样的内容：

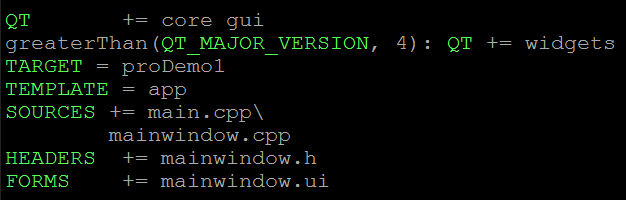


然后清空工程文件夹同级其它目录，再qmake一下，编译后，可以看到不用切换Qt Creator情况下，也我们一次性同时生成了两种版本的可执行程序。

【3】Qt Creator创建工程的pro文件

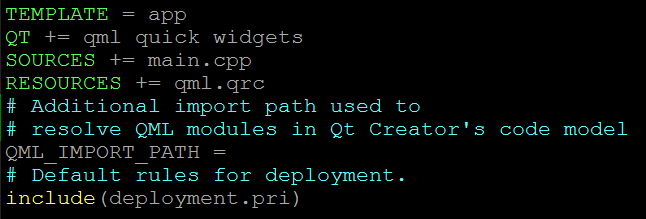
　　下面分别把Qt Creator新建的不同类型工程默认的pro文件罗列一下，可以参考学习：

　　3.1 新建proDemo1工程（注意：模板选择，项目：应用程序；Qt Widgets Application）。步骤如下：Qt Creator--->New Project--->应用程序--->Qt Widgets Application--->名称为：proDemo1（创建路径自己拟定，本地为F:\Source\proStudy）--->类信息保持不变--->完成。对应的pro文件如下：



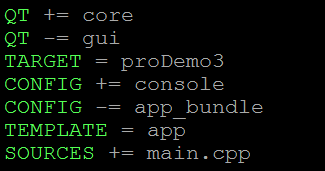
　　此pro文件其实与本文第一张图片相同。因为所建工程相同，默认pro文件也相同。

　　3.2 新建proDemo2工程（注意：模板选择，项目：应用程序；Qt Quick Application）。步骤如下：Qt Creator--->New Project--->应用程序--->Qt Quick Application--->名称为：proDemo2（创建路径自己拟定，本地为F:\Source\proStudy）--->Qt Quick component set : Qt Quick Controls1.2--->完成。对应的pro文件如下：



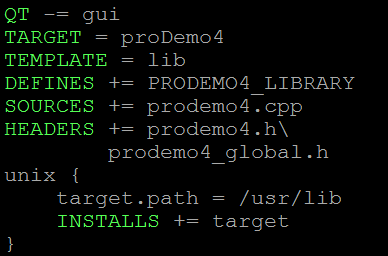
　　此pro文件利用include引入了pri类型的文件。

　　3.3 新建proDemo3工程（注意：模板选择，项目：应用程序；Qt 控制台应用）。步骤如下：Qt Creator--->New Project--->应用程序--->Qt 控制台应用--->名称为：proDemo3（创建路径自己拟定，本地为F:\Source\proStudy）--->类信息保持不变--->完成。对应的pro文件如下：



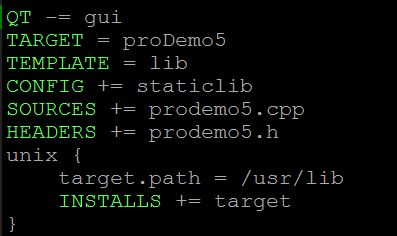
　　此pro文件去掉了CONFIG配置变量默认的app\_bundle项，由于是控制台应用程序。

　　3.4 新建proDemo4工程（注意：模板选择，项目：库；C++库）。步骤如下：Qt Creator--->New Project--->库--->C++ 库--->名称为：proDemo4（创建路径自己拟定，本地为F:\Source\proStudy）--->类型：共享库--->其他项均默认--->完成。对应的pro文件如下：



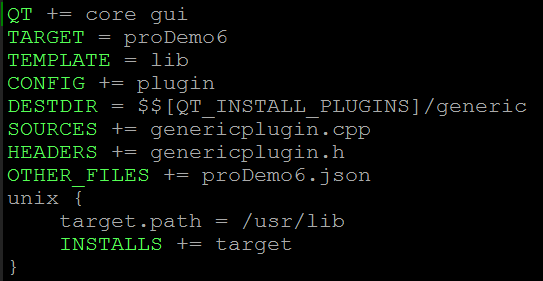
　　此pro文件添加了unix环境的控制。

　　3.5 新建proDemo5工程（注意：模板选择，项目：库；C++库）。步骤如下：Qt Creator--->New Project--->库--->C++ 库--->名称为：proDemo5（创建路径自己拟定，本地为F:\Source\proStudy）--->类型：静态链接库--->其他项均默认--->完成。对应的pro文件如下：



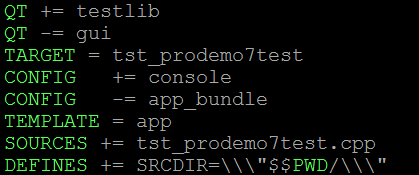
　　此pro文件与3.4工程的最大区别是为配置变量CONFIG添加了staticlib值，因为工程类型选择为静态链接库。

　　3.6 新建proDemo6工程（注意：模板选择，项目：库；C++库）。步骤如下：Qt Creator--->New Project--->库--->C++ 库--->名称为：proDemo6（创建路径自己拟定，本地为F:\Source\proStudy）--->类型：Qt Plugin--->其他项均默认--->完成。对应的pro文件如下：



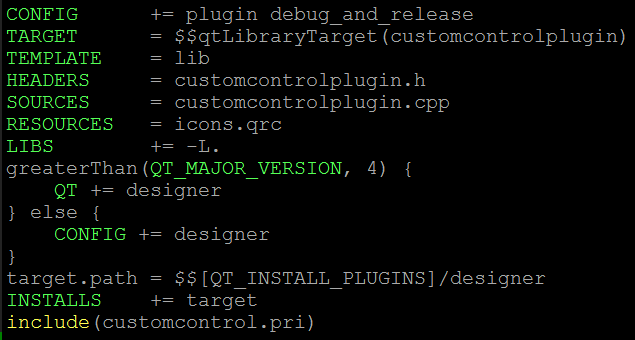
　　此pro文件添加其他文件OTHER\_FILES配置变量，另外，配置变量CONFIG添加了plugin值，因为工程类型选择为Qt Plugin。

　　3.7 新建proDemo7工程（注意：模板选择，项目：其他项目；Qt单元测试）。步骤如下：Qt Creator--->New Project--->其他项目--->Qt单元测试--->名称为：proDemo7（创建路径自己拟定，本地为F:\Source\proStudy）--->其他项均默认--->完成。对应的pro文件如下：



　　此pro文件为QT变量添加testlib值。

　　3.8 新建proDemo8工程（注意：模板选择，项目：其他项目；Qt4设计师自定义控件）。步骤如下：Qt Creator--->New Project--->其他项目--->Qt4设计师自定义控件--->名称为：proDemo8（创建路径自己拟定，本地为F:\Source\proStudy）--->控件类：customControl--->其他项全部默认--->完成。对应的pro文件如下：



　　此pro文件中当Qt版本大于4.0时为QT变量添加值designer。

**备注：整理上文使用本地环境Qt 5.3.2 + Qt Creator3.2.1**