Thinking in Qt ( ⅩⅢ )



之前有被吐槽文章没深度

确实，浅显的文章就好比 github 上的各种带 **Awesome** 的 **Markdown**

[awesome-qt](https://github.com/JesseTG/awesome-qt)

[awesome-qml](https://github.com/mikalv/awesome-qt-qml)

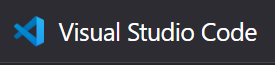
虽然收获了很多 star，却不能给人更长时间的思考，收藏即吃灰



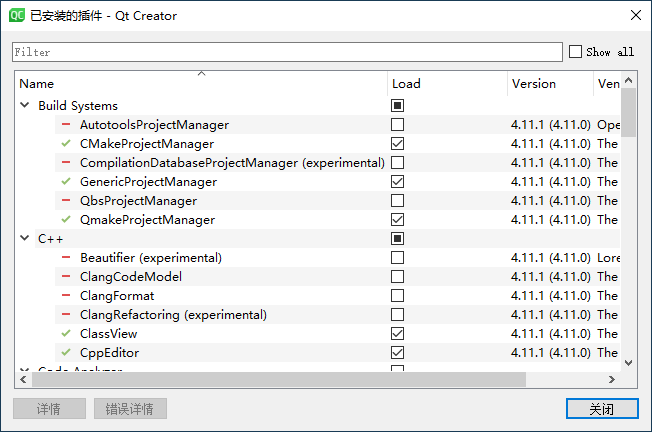
## **Plugin Framework**

很多软件用到了“插件”，有的叫 plug-in、plugin，

有的叫 add-in、addin、add-on、addon



VS Code 使用了插件框架



Qt Creator 使用了插件框架

Qt Creator 也有自己的插件框架，我们可以从清华提供的 Qt 镜像网站上下载到源码

**[Qt Creator 源码](https://mirrors.tuna.tsinghua.edu.cn/qt/development_releases/qtcreator/8.0/8.0.0-rc1/)**

因为开源的缘故，[很多人](https://www.jianshu.com/p/b9e63f8b4db2) 都对代码剖析过了，虽然 [很多](https://blog.csdn.net/skymanwww/article/details/86502003) 都是蜻蜓点水。

但是学习嘛，学多学少都是有收获的。

我这次想做的是“学以致用”，就是参考这份代码，**设计出自己的插件框架**。

换句话说，就是我们如果用 Qt 写了一个程序后，

也拥有一套类似的系统，能够通过插件扩展我们写的程序

## **为什么要这么做？为什么要设计插件系统？用插件有什么好处？**

试想一个场景

多人合作开发一款软件，有的人负责底层硬件通信协议的实现，

有的人负责算法库的实现，有的人负责多种测试场景的实现，

而我们需要整合他们的动态库，在 PC 上用 Qt 开发一个带界面的软件。

我们不用插件系统当然也能“一把梭”

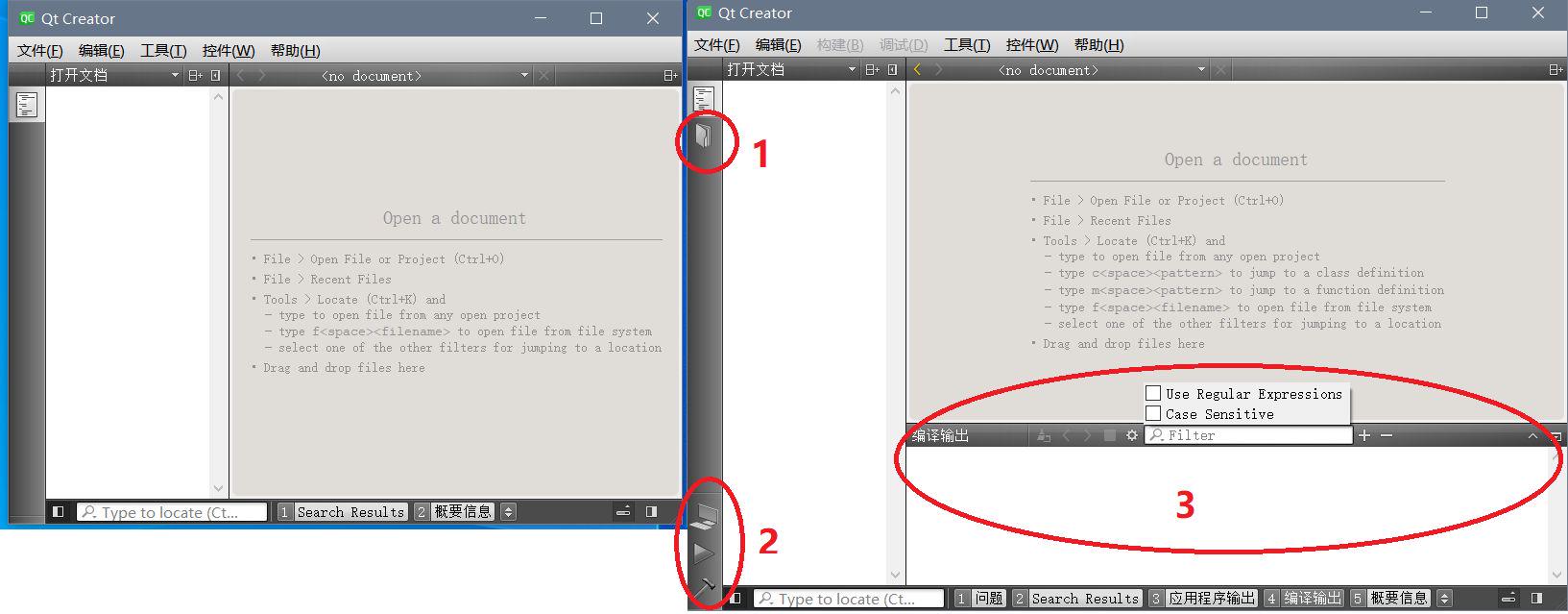


**但如果项目初期硬件通信接口经常变，算法库不停更新，测试场景不停增加呢？**

有了插件框架，我们就可以为“不同模块”或“同一模块的不同版本”编写插件，

把编译出的插件丢进主程序的 Plugins 文件夹下，

不需要编译主程序，就能在主程序上呈现新的界面，新的窗口



Qt Creator 的插件加载前后

其实上面提到的只是插件或者说动态库的好处

而之所以叫框架，就是它能解决不同插件的依赖问题，

并且设计了插件的生命周期，还能在运行时载入插件、卸载插件

比如，你能在程序运行的时候，换一个算法库的版本等等……

总结一下：

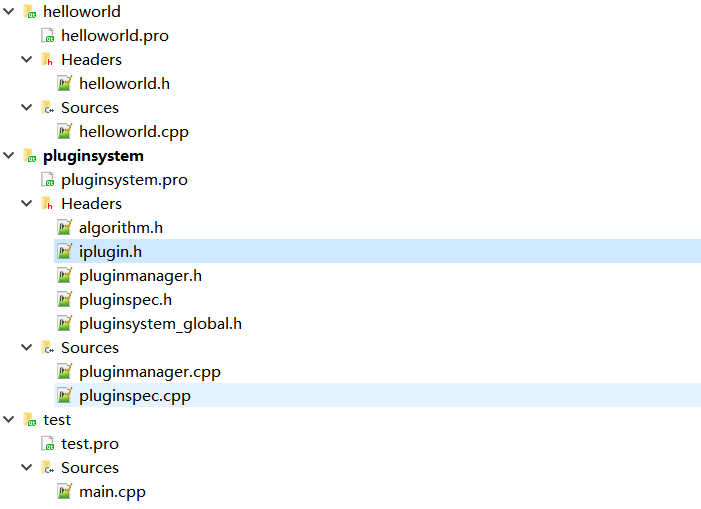
1. **插件动态加载，无需编译主程序**
2. **插件之间有依赖关系时，能自动控制加载顺序**
3. **主程序不需要关闭，热加载插件或卸载插件**
4. **多个不同的版本的插件可以共存，通过配置来选择加载哪些控件**
5. **方便程序的持续集成和持续发布**

## **自制插件框架**

我设计的框架参考了 Qt Creator 的源码

原始的 Extension System 依赖 **Aggregation** 和 **Utils** 模块

还用了 Qt 源码一贯用的 **PIMPL** 模式 和 一堆宏定义 **我一并优化掉了**



pluginsystem 是插件系统，helloworld 是插件，test 是开发的主程序

源码我传到百度网盘了

**[Custom Plugin System](https://pan.baidu.com/s/1U8oITfXj0EPOHbVQuoeOsA?pwd=6p42)**

## **怎么使用这个框架？怎么实现的？原理是什么？**

原理一般都有点长。简而言之，Qt 提供了两种 API 来创建插件

**[plugins how to](https://doc.qt.io/qt-5/plugins-howto.html)**

一种是**高阶API**，一种是**低阶API**。区别在于前者已经定义好了接口而后者需要自己编写。

比如上一篇文章里提到的 QStylePlugin，就属于高阶 API

我们可以派生一个 MyStylePlugin，重写 create 这个接口

1. class MyStylePlugin : public QStylePlugin
2. {
3. Q\_OBJECT
4. Q\_PLUGIN\_METADATA(IID "org.qt-project.Qt.QStyleFactoryInterface" FILE "mystyleplugin.json")
5. public:
6. QStyle \*create(const QString &key);
7. };

然后在 main.cpp 里加入下面这段代码，实现换肤。

1. QApplication::setStyle(QStyleFactory::create("MyStyle"));

具体的例子可以参考

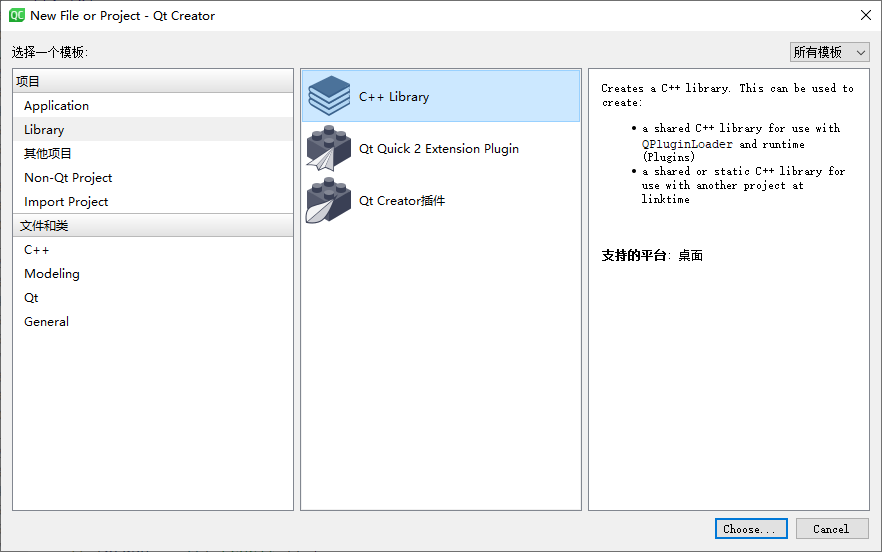
**[style plugin example](https://doc.qt.io/qt-5/qtwidgets-tools-styleplugin-example.html)**

**低阶API** 需要自己定义接口

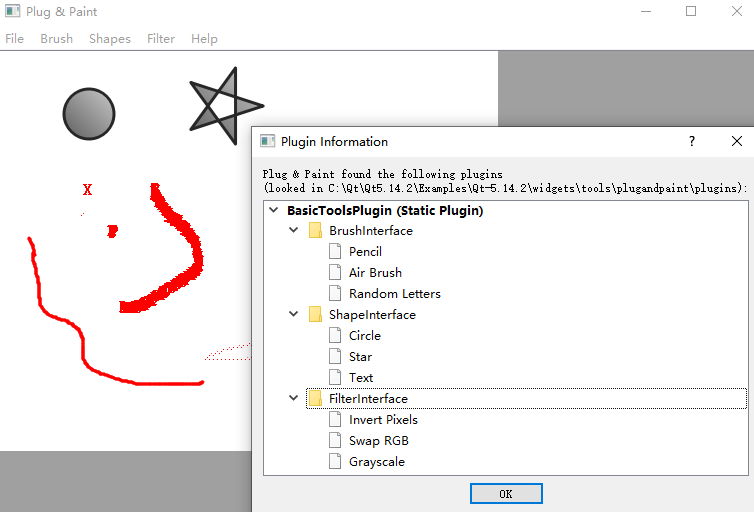
比如我们有个画图程序，自定义了一些接口

1. //interface.h
2. class ShapeInterface
3. {
4. public:
5. virtual ~ShapeInterface() = default;
6. virtual QStringList shapes() const = 0;
7. virtual QPainterPath generateShape(const QString &shape, QWidget \*parent) = 0;
8. };
9. #define ShapeInterface\_iid "org.qt-project.Qt.Examples.PlugAndPaint.ShapeInterface/1.0"
10. Q\_DECLARE\_INTERFACE(ShapeInterface, ShapeInterface\_iid

我们就可以新建一个 C++ Library 的工程，去实现这些接口



具体的例子可以参考



**[plug and paint app example](https://doc.qt.io/qt-5/qtwidgets-tools-plugandpaint-app-example.html)**

我的框架以低阶 API 为基础，没有引入 Q\_INTERFACES 宏

1. #pragma once
2. #include "pluginsystem\_global.h"
3. #include <QObject>
4. class PluginSpec;
5. class PLUGINSYSTEM\_EXPORT IPlugin : public QObject
6. {
7. Q\_OBJECT
8. public:
9. IPlugin() {}
10. ~IPlugin() {}
11. virtual bool initialize(const QStringList &arguments, QString \*errorString) = 0;
12. virtual void extensionsInitialized() {}
13. virtual bool delayedInitialize() { return false; }
14. private:
15. friend class PluginSpec;
16. PluginSpec \*pluginSpec;
17. };

当我们编写插件时，都必须继承 **IPlugin**

1. //pro 里需要加入 TEMPLATE = lib 和 CONFIG += shared dll
2. #pragma once
3. #include "iplugin.h"
4. #include "pluginsystem\_global.h"
5. #include <QWidget>
6. class HelloWorldPlugin final : public IPlugin
7. {
8. Q\_OBJECT
9. Q\_PLUGIN\_METADATA(IID "hello.world" FILE "helloworld.json")
10. public:
11. HelloWorldPlugin(){}
12. ~HelloWorldPlugin() override {
13. delete w;
14. }
15. bool initialize(const QStringList & arguments, QString \* errorString) final {
16. w = new QWidget;
17. w->show();
18. return true;
19. }
20. void extensionsInitialized(){}
21. void shutdown(){}
22. QWidget \*w {nullptr};
23. };

以上是创建了一个名为 HelloWorld 的动态库（即插件）

把它放入我们应用程序（ 即 Test ）的 Plugins 目录，我们的程序就会在运行时加载它

而加载的原因是 pluginspec.cpp 下的这几行代码

1. QPluginLoader loader;
2. loader.setFileName(filePath);
3. loader.load();
4. auto \*pluginObject = qobject\_cast<IPlugin\*>(loader.instance());
5. IPlugin \*plugin = nullptr;
6. plugin = pluginObject;
7. plugin->initialize(arguments, &err);

QPluginLoader 就是 Qt 设计用来加载动态库的跨平台的类

在 Windows 上，底层调用 **LoadLibrary** 和 **GetProcAddress**，

前者用于获得 DLL 的句柄，后者用于获得 DLL 中例程的地址

Unix 中使用 dlopen / dlsym

多个 QPluginLoader 的实例并不会导致同一个动态库被多次加载。如果有多个实例 load 了同一个插件库，那么只有在最后一个实例执行 unload 后才能将动态库卸载，前几个实例的 unload 方法都会返回false，动态库也不会被卸载

需要注意的是，release 的程序只能加载 release 的动态库，

同样，debug 的应用程序也只能加载 debug 版本的插件

在我的插件框架中，PluginManager 被设计为单例类，用来管理应用程序的所有插件

1. #include <QApplication>
2. #include "pluginmanager.h"
3. int main(int argc, char \*argv[])
4. {
5. QApplication a(argc, argv);
6. qPluginManager->setSettings("app.ini");
7. qPluginManager->setPluginPath("Plugins");
8. qPluginManager->loadPlugins();
9. return a.exec();
10. }

**qPluginManager->setPluginPath("Plugins")** 的意思是递归遍历

应用程序同级目录下的 Plugins 文件夹

解决它们的循环依赖问题，然后加载它们，通过绝对路径 new 一个 PluginSpec

保存到 PluginManager 成员变量 QList<PluginSpec \*> pluginSpecs 中

**qPluginManager->loadPlugins()** 的意思是遍历所有 PluginSpec

因为每个 PluginSpec 都有成员变量 QPluginLoader loader

loader.instance 得到插件对象的指针，然后再通过 qobject\_cast 强转成 IPlugin 的指针

接着就能执行不同插件的 initialize(arguments,&err) 来初始化不同插件了

最后我们再说明下编译步骤：

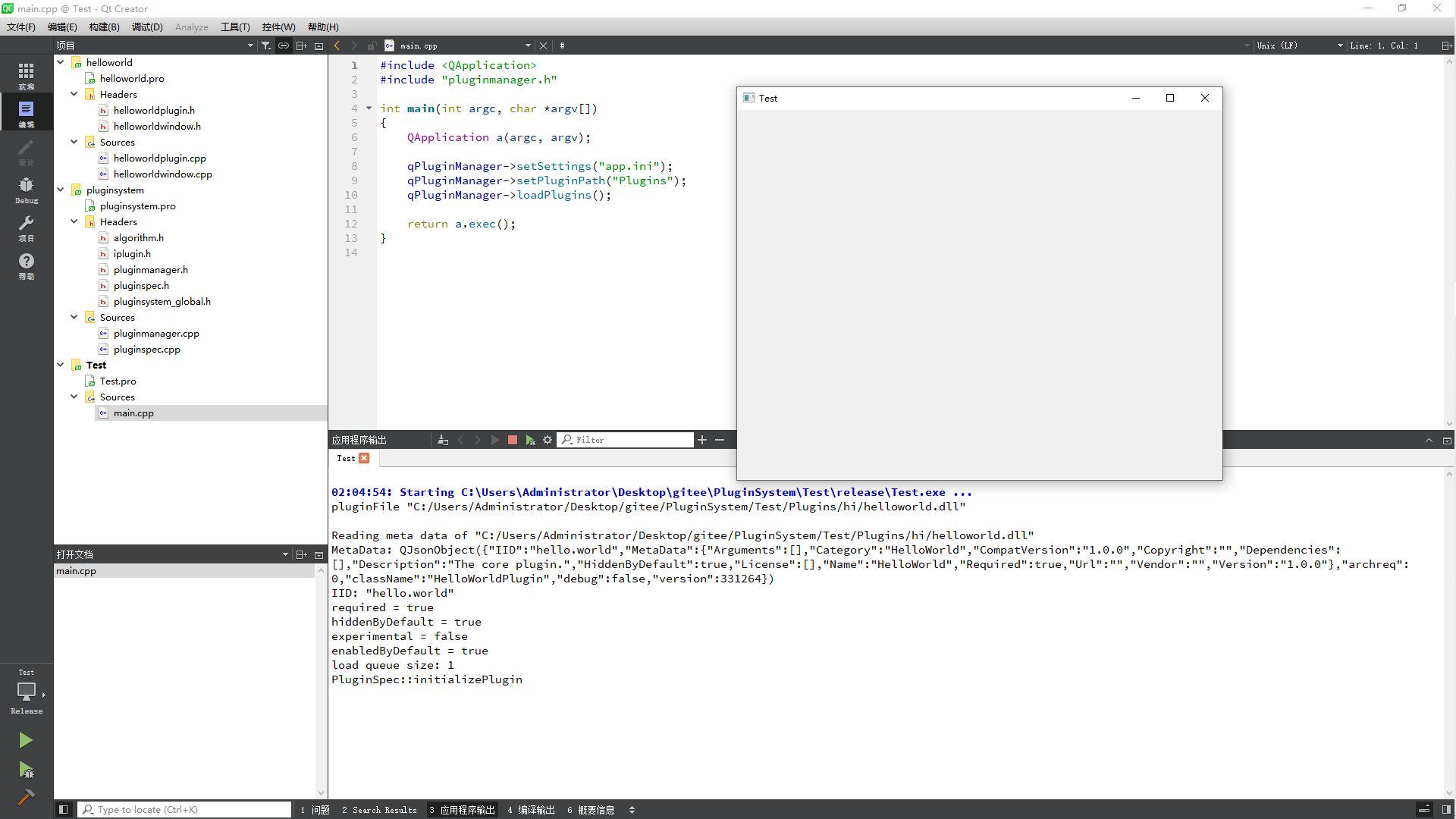
**使用 Qt5.14.2 和 VS2015 的编译器编译。所有的编译都选择 release（或者都选择 debug）模式，并且把 Shadow build 的 √ 去掉**

**先编译 pluginsystem 项目，得到 PluginSystem.lib 和 PluginSystem.dll**

**再编译 helloworld 项目，把 PluginSystem.lib 放入 helloworld/lib 目录下，得到 helloworld.dll**

**最后编译 test 项目，把 PluginSystem.lib 放入 test/lib 目录下，把 PluginSystem.dll 放入 test/release 目录下，把 helloworld.dll 放入 test/Plugins 目录下**

**运行如下**



一个“普普通通”的窗口

终于写完了……其实文章只花了几个小时的时间，代码却花费了我1个礼拜

文章代码上传百度云了 [下载链接](https://pan.baidu.com/s/1hMV5ySRIxmMgqpdGtdfprA?pwd=czwe)

喜欢各位看官们能喜欢