Add C++ plugins to qml

:art:

取自http://doc.qt.io/qt-5/qtqml-modules-cppplugins.html

- 在 qt creater 上新建一个 qt quick application 工程 (minimal version 5.3), 查看自动生成的 main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}
```

- 其中
- QGuiApplication 封装了有关应用程序实例的相关信息 (比如程序名字、命令行参数等);
- QQmlApplicationEngine 管理带有层次结构的上下文和组件, 它需要一个 QML 文件作为应用程序的入口点;
- QML engine 为 qml 加载插件, 这些插件经常被 qml 的扩展模块提供, 那些导入了这些模块的 qml 文档可以提供各种用途;
- QQmlExtensionPlugin 是一个插件接口, 它使得创建 qml 扩展变得可能, 并且这些扩展可以被 qml 应用动态加载; 这些扩展也使得自定义 qml 类型可以在 qml 引擎中使用;
- 如何创建一个 qml 插件
- qt creater 5.3 可以使用向导创建 qt quick2 extension plugin 了
  - 选择菜单栏, 创建新文件或工程;
  - 选择 library,qt quick2 extension plugin
  - 为自己的 class(这个 class 是自定义 qml 类型的实现类) 和 URI(这个 URI 用于在 qml 中导入使用) 命名, 默认为 MyItem;
  - 点击完成;

- 查看自动生成的 plugin.h 文件

```
#ifndef QMLPLUGINTEST_PLUGIN_H
#define QMLPLUGINTEST_PLUGIN_H

#include <QQmlExtensionPlugin>

class QmlPluginTestPlugin : public QQmlExtensionPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.QQmlExtensionInterface")

public:
    void registerTypes(const char *uri);
};

#endif // QMLPLUGINTEST_PLUGIN_H
```

- plugin.cpp 文件

```
#include "qmlplugintest_plugin.h"
#include "test.h"

#include <qqml.h>

void QmlPluginTestPlugin::registerTypes(const char *uri)
{
    // @uri org.test
    qmlRegisterType<test>(uri, 1, 0, "test");
}
```

- Q_PLUGIN_METADATA 一行强制将该插件识别为一个 QML 扩展插件;
- qmlRegisterType() 函数注册了自定义类 (上面为 test);
- 在实现自定义 qml 类型的实现类前, 先来学习 Q_PROPERTY 宏;
- Q_PROPERTY() 是一个宏, 用来在一个类中声明一个属性 property, 由于该宏是 qt 特有的, 需要用 moc 进行编译, 故必须继承于 QObject 类;
- 语法

```
Q_PROPERTY(type name READ getFunction
  [WRITE setFunction]
```

```
[RESET resetFunction]
[NOTIFY notifySignal]
[DESIGNABLE bool]
[SCRIPTABLE bool]
[STORED bool]
[USER bool]
[CONSTANT]
[FINAL])
```

- property 跟类中数据成员没有什么区别, 但是有几点不一样
  - 必须有一个 read 函数, 它用来读取属性值, 因此用 Const 限定, 它的返回值类型必须为属性类型或者属性类型的引用或者指针;
  - 有一个可选的 write 函数, 它用来设置属性值, 它的返回值必须为 void 型, 而且必须要含有一个参数;
  - 一个可选的 reset 函数把 property 设置成其默认状态, 复位功能必须返回 void, 并且不带参数;
  - 一个可选的 NOTIFY 信号, 如果定义它提供了一个信号, 那么这个信号在值发生改变时会自动被触发;
  - DESIGNABLE 属性表明该 property 是否能在 GUI builder(一般为 Qt Designer) 可见;
  - STORED 属性表明是否一直存在的;
  - USER 属性表明是否可以被用户所编辑;
  - CONSTANT 设定属性是不可修改的, 不能跟 WRITE 或者 NOTIFY 同时出现;
  - FINAL 表明该属性不会被派生类中重写;
- 示例

```
class Test : public QObject
{
  Q_OBJECT
  Q_PROPERTY(bool enabled READ isEnabled WRITE setEnabled)
  public:
  Test(QObject *parent = 0) : QObject(parent) {}
  virtual ~Test(){}
  void setEnabled(bool e) { enabled = e; }
  bool isEnabled() const { return enabled; }

private:
  bool enabled;
};
```

- 根据需要定制自己的实现类;
- To be able to call a method from QML, you must either mark it with Q_INVOKABLE or as a slot;
- 要想在 qml 里面调用一个方法, 这个方法必须用 Q_INVOKABLE 标识或作为一个槽;
- 插件 “其他文件” 里有 qmldir 文件, 这个文件指定了 QML 插件的内容以及插件的 QML 方面的描述;

- 现在构建完成之后在插件构建目录 (就是你创建插件的目录下的对应的 build 目录) 下执行 make install 完成插件的安装;
- 现在如果要使用自定义插件, 只要在 qml 里 import 相应的模块即可
- 比如说你的 qmldir 文档如下

```
module TimeExample
Clock 1.0 Clock.qml
plugin qmlqtimeexampleplugin
```

- plugin.cpp 中这么定义

```
qmlRegisterType<TimeModel>(uri, 1, 0, "Time");
```

- 那么只要在 qml 只需要这么做, 就可以使用上面的插件了

```
import TimeExample 1.0 // import types from the plugin
Time { // this class is defined in C++ (plugin.cpp)
    id: time
}
```

- 使用

```
hours: time.hour
minutes: time.minute
```