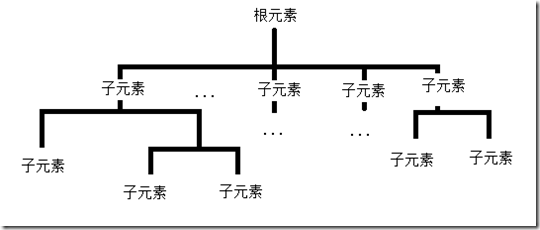
在使用Qt之前，需要了解几个概念。

1. qml：https://en.wikipedia.org/wiki/QML

qml是一种脚本语言（类似python），和python一样，在使用时也要import很多模块。QML将界面分成一些更小的元素，这些元素可以组成一个组件，QML语言描述了UI的形状和行为，并且可以使用JavaScript修饰。总的来说QML的结构有点像HTML，其语法和CSS比较近似。

要使用QML进行界面的布局，首先需要理解QML元素的层次结构。QML的层次结构很简单，是一个树形结构，最外层必须有一个根元素，根元素里面可以嵌套一个或多个子元素，子元素里面还可以包含子元素。如果用图形画出来的话大概是这个样子。



QML的坐标系采用的屏幕坐标系，原点在屏幕左上角，x轴从左向右增大，y轴从商到下增大，z轴从屏幕向外增大。子元素从父元素上继承了坐标系统，它的x,y总是相对于它的父元素坐标系。这一点一定要记住，非常重要。

一个简单的例子：<https://www.cnblogs.com/csulennon/p/4485768.html>

2. Qt quick：<https://en.wikipedia.org/wiki/Qt_Quick>

是Qt框架下的一个子框架，旨在UI。

Qt quick的两种启动例子：<http://www.cnblogs.com/mcumagic/p/5320324.html>，一个是用QQuickView，一种是用QQmlApplicationEngine。

使用QQuickView显示QML文档，对窗口的控制权（比如设置窗口标题、Icon、窗口的最小尺寸等）在C++代码；而使用QQmlApplicationEngine加载以Window为根对象的QML文档，QML文档则拥有窗口的完整控制权，可以直接设置标题、窗口尺寸等属性。

Qt例子中的places\_map和mapviewer（MapPlginDemo）分别用到了这两种方式。

关于QQmlApplicationEngine：

<https://www.devbean.net/2016/02/qt-study-road-2-extending-qml/>

在 Qt Creator 中，我们创建 Qt Quick Application 项目，打开 Qt Creator 自动帮我们生成的 main.cpp，可以看到类似下面的代码（由于版本问题，这段代码可能会有所不同）：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #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();  } |

在这段代码中，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。QQmlApplicationEngine需要一个 QML 文件，将其加载作为应用程序的入口点。在这个例子中，这个文件就是 main.qml。Qt Creator 帮我们生成的 QML 文件被作为资源文件，因此需要使用“qrc”前缀访问到。

3. 插件

Qt有两种与插件有关的API。一种用来扩展Qt本身的功能，如自定义数据库驱动，图像格式，文本编解码，自定义分格，等等，称为Higher-Level API 。另一种用于应用程序的功能扩展，称为Lower-Level API。前一种是建立在后一种的基础之上的。这里讨论的是后一种，即用来扩展应用程序的Lower-level API。（注：前者就是Qt Creator中“帮助”->“关于插件”中的插件？后者就是字节写的插件，比如MapPlginDemo中的GeoServiceProviderFactory

）

Qt中的地图插件：

为了通过Qt的API使用不同地图服务商的接口，Qt提供了很多插件，包括

<https://doc.qt.io/qt-5/qtlocation-index.html>

可以使用Esri、HERE、Mapbox、OSM的地图服务。（没有google、bing和国内的百度、高德等）。

places\_map使用插件方法：

Rectangle {

anchors.fill: *parent*

//! [Initialize Plugin]

Plugin { //定义一个Plugin类型

id: *myPlugin*

name: "osm" // "mapboxgl", "esri", ...

//specify plugin parameters if necessary

//PluginParameter {...}

//PluginParameter {...}

//...

}

Map {

id: *map*

anchors.fill: *parent*

plugin: *myPlugin*; //或者直接plugin: "osm";

center: *locationOslo*

zoomLevel: 13

MapItemView {

model: *searchModel*

delegate: MapQuickItem {

coordinate: place.location.coordinate

anchorPoint.x: *image*.width \* 0.5

anchorPoint.y: *image*.height

sourceItem: Column {

Image { id: *image*; source: "marker.png" }

Text { text: title; font.bold: true }

}

}

}

}

为了使用google、高德等厂商的地图服务，就需要自己写插件（比如QGroundControl和MapPlgindemo），在MapPlgindemo的qml中：

Window {

Map {

id: *\_map*

anchors.fill: *parent*

zoomLevel: 18

// center: wps84\_To\_Gcj02(43.8868593, 125.3247893, 0.0) //RenMin square

center: *wps84\_To\_Gcj02*(43.9688305, 125.3763277, 0.0) //Jinxing Buiding

gesture.flickDeceleration: 3000

// 地图插件

plugin: Plugin { name: "Gaode" }

}

}

Window是根元素，Map是子元素。Map中包含了plugin这个属性，它的值表示数据源，比如

plugin: Plugin { name: "Gaode" }

第二个Plugin是一个QML类型，那么是它的一个属性：

|  |
| --- |
| **name** : string |

This property holds the name of the plugin. Setting this property will cause the Plugin to only attach to a plugin with exactly this name.

这里面的“Gaode”就是Plugin的名字。

如何创建自己的插件（这里主要关注low level插件）：

https://doc.qt.io/qt-5/plugins-howto.html

Writing a plugin involves these steps:

1. Declare a plugin class that inherits from [QObject](https://doc.qt.io/qt-5/qobject.html) and from the interfaces that the plugin wants to provide.
2. Use the [Q\_INTERFACES](https://doc.qt.io/qt-5/qobject.html" \l "Q_INTERFACES)() macro to tell Qt's [meta-object system](https://doc.qt.io/qt-5/metaobjects.html) about the interfaces.
3. Export the plugin using the [Q\_PLUGIN\_METADATA](https://doc.qt.io/qt-5/qtplugin.html" \l "Q_PLUGIN_METADATA)() macro.
4. Build the plugin using a suitable .pro file.

比如说qgroundcontrol在QGEOSERVICEPROVIDERQGC.h中：

class QGeoServiceProviderFactoryQGC: public QObject, public QGeoServiceProviderFactory

{

Q\_OBJECT

Q\_INTERFACES(QGeoServiceProviderFactory)

Q\_PLUGIN\_METADATA(IID "org.qt-project.qt.geoservice.serviceproviderfactory/5.0" FILE "qgc\_maps\_plugin.json")

public:

QGeoCodingManagerEngine\* ***createGeocodingManagerEngine*** (const QVariantMap &parameters, QGeoServiceProvider::Error \*error, QString \*errorString) const;

QGeoMappingManagerEngine\* ***createMappingManagerEngine*** (const QVariantMap &parameters, QGeoServiceProvider::Error \*error, QString \*errorString) const;

QGeoRoutingManagerEngine\* ***createRoutingManagerEngine*** (const QVariantMap &parameters, QGeoServiceProvider::Error \*error, QString \*errorString) const;

QPlaceManagerEngine\* ***createPlaceManagerEngine*** (const QVariantMap &parameters, QGeoServiceProvider::Error \*error, QString \*errorString) const;

};

MapPlginDemo只是少了 QGeoCodingManagerEngine。

上面这些public的方法都是什么？

<https://doc.qt.io/qt-5/qtlocation-geoservices.html>。都是[QGeoServiceProviderFactory](https://doc.qt.io/qt-5/qgeoserviceproviderfactory.html)的方法，用户可以重载这些方法。

上面在定义plugin时，有一个json文件，这个文件描述了plugin的能力和版本，其格式与插件有关，与QLocation GeoService插件有关的，见

<https://doc.qt.io/qt-5/qtlocation-geoservices.html>。

{

"Keys": ["maps"],

"Provider": "Gaode",

"Version": 101,

"Experimental": false,

"Features": [

"OnlineMappingFeature",

"OnlineGeocodingFeature",

"ReverseGeocodingFeature"

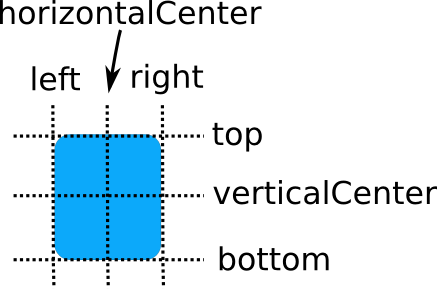
]

}

4. anchors

<http://doc.qt.io/qt-5/qtquick-positioning-anchors.html>

是Qtquick中放置item的一个工具。每个item都有几个不可见的线：

这样就可以描述两个item的相对关系了，比如：

Rectangle { id: rect1; ... }

Rectangle { id: rect2; anchors.left: rect1.right; ... }

In this case, the left edge of *rect2* is bound to the right edge of *rect1*, producing the following:



还有anchors.fill用于设置item和target一样，比如：

MouseArea {

id: *redSquareMouseArea*

anchors.fill: *parent*

…

}

就表示MouseArea的区域和parent一样。

5. MouseArea

学习MouserArea：Qt Creator中例子，mousearea。

MouseArea本身是一个不可见的item，用于为某个可见的item提供鼠标事件交互。一个简单的例子：

MouseArea

{

id:*parentArea*

anchors.fill: *parent*

hoverEnabled: true

onPositionChanged: *par*.text = 'Moving'

}

Text {

id: *par*

anchors.bottom: *info*.top;anchors.horizontalCenter: *parent*.horizontalCenter;anchors.margins: 30

}

鼠标进入到MouseArea上面的window或者rectangle中并进行移动时，就会触发positionChanged([MouseEvent](https://doc.qt.io/qt-5/qml-qtquick-mouseevent.html) *mouse)*这个*signal*，对应的onPositionChanged是一个handler：更改Text的内容。

6. Qobject

The [QObject](http://doc.qt.io/qt-5/qobject.html)-based version has the same internal state, and provides public methods to access the state, but in addition it has support for component programming using signals and slots.

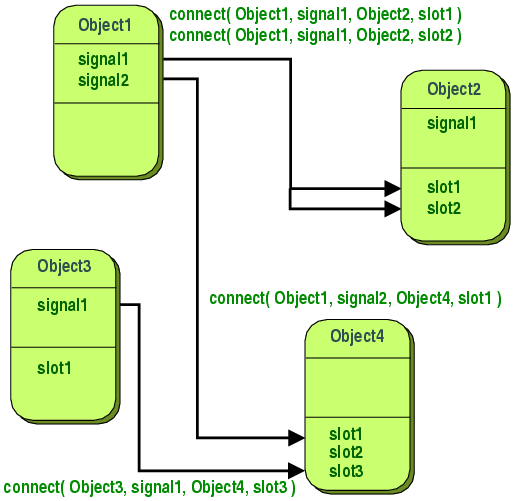
All classes that contain signals or slots must mention [Q\_OBJECT](http://doc.qt.io/qt-5/qobject.html" \l "Q_OBJECT) at the top of their declaration. They must also derive (directly or indirectly) from [QObject](http://doc.qt.io/qt-5/qobject.html).

7. 信号与槽

signal & slot是Qt发明的、在不同object之间通信的机制，也是Qt的最大的特点。

<http://doc.qt.io/qt-5/signalsandslots.html>

其他的通信机制：callback，或者[observer interface＋ registration](https://www.zhihu.com/question/21219705/answer/17612611)。 callback就是将一个handler的函数指针传递给某个函数，当这个函数发生某个事件时，就会自动调用callback来响应。Qt采用signal&slot的方式：

connect语句中的Object1、signal1、Object2、slot1必须都是指针。一般connect语句都在类的构造函数中编写。

信号与槽的参数类型必须是匹配正确的，因此是类型安全的。

signals可以从任意位置emit，但qt建议在定义signal的类或者子类中emit。

signal是由MOC自动产生的，没有返回类型，只能声明，不能在cpp文件中实现。

slots是C++函数，可以被正常调用，唯一特别的就是有signals连接到slots。和callback相比，signals & slots会慢10倍。

8. 在Qwidget中调用qml

http://blog.csdn.net/u014597198/article/details/70258498

Qt电子地图笔记

资源：

1. QGroundControl，在github上下载，Readme.md中有交代Qt版本要求

2. Qt高得地图插件：<http://download.csdn.net/download/linge1995/9798193>，就是一个window显示高得地图，和Qt中的相关内容差不多，可以在此基础上进行开发

3. Qt的几个example：mapviewer、places\_map。

4. Qt的web API：<http://blog.csdn.net/d759378563/article/details/76576550>

5 Google的web map：<https://github.com/google/earthenterprise>

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 地图 | Plugin | Qquick启动形式 |
| QgroundControl | Google、Bing、MapQuest、OpenStreetMap | 自己编写的QGeoServiceProviderFactoryQGC | QqmlApplicationEngine |
| MapPlginDemo | 高德 | 自己编写的GeoServiceProviderFactory | QqmlApplicationEngine |
| mapviewer | ESRI、HERE、Mapbox、OpenStreetMap | 官方的 | QqmlApplicationEngine |
| places\_map | 同mapviewer | 官方的 | QQuickView |

其中QGroundControl是最贴近我们需求的，因此以它为蓝本。MapPlginDemo是把它地图的部分拿出来的，可以在此基础上开发第一个小程序。

第一个小程序：

在MapPlginDemo的基础上，实现：

1. 移动鼠标时输出经纬度（高度）;

2. 点击鼠标时生成一个Marker点;

3. 将该点的坐标通过串口输出一帧。

一、移动鼠标时输出经纬度（高度）

鼠标移动应该是一个事件，Qt应当能监听到这一事件。MouseArea的onPositionChanged句柄能满足这一要求。

获得经纬度需要用到Map QML类型的**toCoordinate**：

|  |
| --- |
| coordinate **toCoordinate**(QPointF *position*, bool *clipToViewPort*) |

Returns the coordinate which corresponds to the *position* relative to the map item.

再看coordinate QML变量：

The coordinate type represents and stores a geographic position.

This type is a QML representation of [QGeoCoordinate](http://qgeocoordinate.html/) and represents a geographic position in the form of [latitude](http://qml-coordinate.html/" \l "latitude), [longitude](http://qml-coordinate.html/" \l "longitude) and [altitude](http://qml-coordinate.html/" \l "altitude) attributes.

其中altitude是指above sea level，可能还不是wgs84坐标系的，这个回头再考证。

总之可以用onPositionChanged: *cord*.text = *map*.toCoordinate(*Qt*.point(*mouse*.x,*mouse*.y)).latitude+' ' +

*map*.toCoordinate(*Qt*.point(*mouse*.x,*mouse*.y)).longitude + ' ' + *map*.toCoordinate(*Qt*.point(*mouse*.x,*mouse*.y)).altitude

来实现了。

二、 点击鼠标时生成一个Marker点

点击鼠标同样可以用onClicked句柄来实现。

Marker点需要有一个图标：把marker.png放到工程的qml.qrc中。

将这个图标放置到Map上，需要用到的QML类型是MapQuickItem。在Map中添加：

https://stackoverflow.com/questions/34637133/qml-maps-get-coordinates-when-tap-the-screen

MapQuickItem {

id:marker

sourceItem: Image{

id: image

source: "marker.png"

}

coordinate: map.center

anchorPoint.x: image.width / 2

anchorPoint.y: image.height / 2

}

三. 将该点的坐标通过串口输出一帧，发出的帧和接收的帧应当在另外一个窗口显示。

输出一帧，应该是一个函数，可以用c++写（也可以用qml写吗？）。