[**Qt的内存管理机制**](https://www.cnblogs.com/apocelipes/p/9991845.html)

当我们在使用Qt时不可避免得需要接触到内存的分配和使用，即使是在使用Python，Golang这种带有自动垃圾回收器（GC）的语言时我们仍然需要对Qt的内存管理机制有所了解，以更加清楚的认识Qt对象的生命周期并在适当的时机加以控制或者避免进入陷阱。

这篇文章里我们将学习QObject & parent对象管理机制，以及QWidget与内存管理这两点Qt的基础知识。

**QObject和内存管理**

在Qt中，我们可以大致把对象分为两类，一类是QObject和它的派生类；另一类则是普通的C++类。

对于第二种对象，它的生命周期与管理和普通的C++类基本没有区别，而QObject和它的派生类则有以下的显著区别：

* QObject和其派生类可以使用SIGNAL/SLOT机制
* 它们一般会有一个parent父对象的指针，用于内存管理（后面重点说明）
* 对于QWidget和其派生类来说，内存管理要稍微复杂一些，因为QWidget需要和eventloop高度配合才能工作（后面也会重点说明）

signal和slot一般来说并不会对内存管理产生影响，但是对close()槽的处理会对QWidget产生一些影响，所以我们放在后面讲解。

那么先来看一下QObject和parent机制。

**QObject的parent**

我们时常能看到QWidget或者其他的控件的构造函数中有一项参数parent，默认值都为NULL，例如：

QLineEdit(const QString &contents, QWidget \*parent = nullptr);

QWidget(QWidget \*parent = nullptr, Qt::WindowFlags f = ...);

这个parent的作用就在于使当前的对象实例加入parent指定的QObject及其派生类的children中，当一个QObject被delete或者调用了它的析构函数时，所有加入的children也会全部被析构。

如果parent设置为NULL，会有如下的情况：

* 如果是构造时直接指定了NULL，那么当前实例不会有父对象存在，Qt也不能自动析构该实例除非实例超出作用域导致析构函数被调用，或者用户在恰当的实际使用delete操作符或者使用deleteLater方法；
* 如果已经指定了非NULL的parent，这时将它设置成了NULL，那么当前实例会从父对象的children中删除，不再受到QObject & parent机制的影响；
* 对于QWidget，parent为NULL时代表其为一个顶层窗口，也可以就是独立于其他widget在系统任务栏单独出现的widget，对于永远都是顶层窗口的widget，例如QDialog，当parent不为NULL时他会显示在父widget中心区域的上层；
* 如果QWidget的parent为NULL或是其他值，在其加入布局管理器或者QMainWindow设置widget时，会自动将parent设置为相应的父widget，在父控件销毁时这些子控件以及布局管理器对象会一并销毁。

所以我们可以看出，QObject对象实际上拥有一颗类实例关系树，在树中保存了所有通过指定parent注册的子对象，而子对象里又保存有其子对象的关系树，所以当一个父对象被销毁时，所有依赖或间接依赖于它的对象都会被正确的释放，使用者无需手动管理这些资源的释放操作。

基于此原理，我们可以放心的让Qt管理资源，这里有几个建议：

1. 对于QObject及其派生类，如果彼此之间存在一定联系，则应该尽量指定parent，对于QWidget应该指定parent或者加入布局管理器由管理器自动设置parent。
2. 对象只需要在局部作用域存在时可以选择不进行内存分配，利用局部作用域变量的生命周期自动清理资源。
3. 对于非QWidget的对象来说，如果不指定非NULLparent，则需要自己管理对象资源。QWidget比较特殊，我们在下一节讲解。
4. 对于在局部作用域上创建的父对象及其子对象，要注意对象销毁的顺序，因为父对象销毁时也会销毁子对象，当子对象会在父对象之后被销毁时会引发double free。

**QWidget和内存的释放**

QWidget也是QObject的子类，所以在parent机制上是没有区别的，然而实际使用时我们更多的是使用“关闭”(close)而不是delete去删除控件，所以差异就出现了。

先提一下widget关闭的流程，首先用户触发close()槽，然后Qt向widget发送QCloseEvent，默认的QCloseEvent会做如下处理：

1. 将widget隐藏，也就是hide()；
2. 如果有设置Qt::WA\_DeleteOnClose，那么会接着调用widget的析构函数

我们可以看到，widget的关闭实际是将其隐藏，而没有释放内存，虽然我们有时会重写closeEvent但也不会手动释放widget。

看一个因为close机制导致的内存泄漏的例子，我们在button被单击后弹出某个自定义对话框：

button.ConnectClicked(func (\_ bool) {

dialog := NewMyDialog()

dialog.Exec()

})

因为dialog在close时会被隐藏，而且没有设置DeleteOnClose，所以Qt不会去释放dialog，而用户也无法回收dialog的资源，也行你会说golang的GC不是能处理这种情况吗，然而遗憾的是GC并不能处理cgo分配的资源，所以如果你期望GC做善后的话恐怕要失望了，每次点击按钮后内存用量都会增加一点，没错，内存泄露了。

那么给dialog设置一个parent，像这样，会如何呢？

dialog.SetParent(self)

遗憾的是，并没有什么区别，因为这样只是把dialog加入父控件的children，并没有删除dialog，只有父对象被销毁时内存才会真正释放。

解决办法也有三个。

第一种是使用deleteLater，例如：

dialog.DeleteLater()

这会通知Qt的eventloop在下次进入主循环的时候析构dialog，这样一来确实解决了内存泄露，不过缺点是会有不可预测的延迟存在，有时候延迟是难以接受的。

第二种是手动删除widget，适用于parent为NULL的场合：  
C++：

delete dialog;

golang：

dialog.DestroyMyDialog()

说明一下，DestroyType也是qtmoc生产的帮助函数，因为golang没有析构函数的概念，所以goqt使用生成的该帮助函数显示调用底层C++对象的析构函数。

第三种比较简单，对于单纯显示而不需要和父控件做交互的widget，直接设置DeleteOnClose即可，close时widget会被自动析构。

当然对于PyQt5来说并不会存在如上的问题，sip库能很好的与python的GC一起工作。唯一需要注意的是有时底层C++对象已经被释放，但是上层python对象依然存在，这时使用该对象将导致抛错。

**总结**

Qt提供了一套方便的机制帮助我们进行内存和资源管理，使我们从繁重的劳动中得到了部分的解放，但同时也要注意到那些很容易坑，这样才能写出健壮的正确执行的程序。

如有错误之处，欢迎批评指正。