<https://blog.csdn.net/u010155023/article/details/50826102>

原文标题：[d指针在Qt上的应用及实现](http://blog.csdn.net/rabinsong/article/details/9474859)

原文链接：http://blog.csdn.net/rabinsong/article/details/9474859

正文：

Qt为了使其动态库最大程度上实现二进制兼容，引入了d指针的概念。那么为什么d指针能实现二进制兼容呢？为了回答这个问题，首先弄清楚什么是二进制兼容？所谓二进制兼容动态库，指的是一个在老版本库下运行的程序，在不经过编译的情况下，仍然能够在新的版本库下运行；需要经过编译才能在新版本下运行，而不需要修改该程序源代码，我们就说该动态库是源代码兼容的。要使一个dll能达到二进制兼容，对于一个结构，对于一个对象，其数据模型应该不变，若有变动，比如在类中增加数据成员或删除数据成员，其结果肯定影响对象的数据模型，从而导致原有数据程员在对象数据模型里的位移发生变化，这样的话编译后的新版本库很可能使程序发生崩溃，为了使在增加和添加项后不使对象数据模型大小发生变化，一种做法是预先分配若干个保留空间，当要添加项时，使用保留项。如下：

1. class A {
3. private:
4. int a;
5. int reserved[3];
6. };
8. class B {
9. private:
10. int a;
11. quint32 b : 1;
12. quint32 reserved : 31;
13. };

这样的话，当样增加项的时候，只需要利用reserved空间，这样的话，对象模型就不会改变。但是这种做法很呆板，因为你不知道未来到底会有多少扩展项，少了不满足要求，多了浪费空间。那麽有没有一种更灵活的方法呢？如下：

1. class Data {
2. public:
3. int a;
4. };
6. class A {
8. private:
9. Data \*d\_ptr;
10. };

将A中的成员a放入Data 中，A中放入Data的一个指针，这样的话，无论你向Data中添加多少数据，A的对象模型始终是4个字节的大小（d\_ptr指针的大小），这种做法是不是比上面的做法更灵活呢？d\_ptr就是我们今天所要说的d指针，Qt为了实现二进制兼容，绝大数类中都包含有这样的指针，下面我们一起来看看Qt的d指针是怎么实现的：

如上图,这个是Qt根结点的指针的一般形式，下面来看看非根结点的一般形式，

注意这里QWidge派生自QObject,它里面没有d\_ptr,但是它的成员函数可以访问d\_ptr，因为 d\_ptr是保护成员，且它的对象模型包含 d\_ptr(这是因为派生类继承父类的所有成员)。

下面我们来看看Qt对上述两种情况是怎么实现的：

qobject.h文件：

1. QObjectData {
2. public:
3. QObject \*q\_ptr;
4. ...
5. };
7. class Q\_CORE\_EXPORT QObject
8. {
9. ...
10. Q\_DECLARE\_PRIVATE(QObject)
11. public:
12. Q\_INVOKABLE explicit QObject(QObject \*parent=0);
13. virtual ~QObject();
14. ...
16. protected:
17. QObject(QObjectPrivate &dd, QObject \*parent = 0);
18. ...
20. protected:
21. QScopedPointer<QObjectData> d\_ptr;
22. ...
23. };

如上，在这里我算去了其他的项，只保留了于d\_ptr有关的项，首先来看看Q\_DECLARE\_PRIVATE(QObject)是什么：

1. #define Q\_DECLARE\_PRIVATE(Class) \
2. inline Class##Private\* d\_func() { return reinterpret\_cast<Class##Private \*>(qGetPtrHelper(d\_ptr)); } \
3. inline const Class##Private\* d\_func() const { return reinterpret\_cast<const Class##Private \*>(qGetPtrHelper(d\_ptr)); } \
4. friend class Class##Private;

根据宏定义，则Q\_DECLARE\_PRIVATE(QObject)翻译如下：

1. inline QObjectPrivate \*d\_func()
2. {
3. return reinterpret\_cast<QObjectPrivate \*>(qGetPtrHelper(d\_ptr));
4. }
5. inline const QObjectPrivate \*d\_func() const
6. {
7. return reinterpret\_cast<const QObjectPrivate \*>(qGetPtrHelper(d\_ptr));
8. }
9. friend class QObjectPrivate;

再来看看qGetPtrHelper的定义：

1. template <typename T> static inline T \*qGetPtrHelper(T \*ptr)
2. {
3. return ptr;
4. }

再来看QScopePointer，它类似于智能指针，这样不用关心 d\_ptr的释放，当离开QScopePointer的作用范围，QScopePointer会自动释放d\_ptr指向的堆内存，那麽这个指针是什么时候生成的呢？q\_ptr又是什么时候赋值的呢？让我们来看看qobject.cpp的实现：

1. QObject::QObject(QObject \*parent)
2. : d\_ptr(new QObjectPrivate)
3. {
4. Q\_D(QObject);
5. d\_ptr->q\_ptr = this;
6. ...
7. }
9. QObject::QObject(QObjectPrivate &dd, QObject \*parent)
10. : d\_ptr(&dd)
11. {
12. Q\_D(QObject);
13. d\_ptr->q\_ptr = this;
14. ...
15. }

我们看第一个构造函数，对于根结点的d\_ptr指向new QObjectPrivate，而QObjectPrivate派生自QObjectData，那麽Q\_D(QObject)宏表示什么意思呢？

#define Q\_D(Class) Class##Private \* const d = d\_func()

Q\_D(QObject)；翻译如下：

QObjectPrivate \* const d = d\_func();

不难看出Q\_D(QObject)；定义了一个QObjectPrivate的常量指针，指向d\_func() 的返回值，而该返回值，正是d\_ptr（见头文件 d\_func()的定义），因此同过Q\_D宏我们就可以访问d指针了。

对于第二个构造函数稍后介绍，下面来看看非根结点的d\_ptr的实现情况：

头文件：

1. class Q\_CORE\_EXPORT QObjectPrivate : public QObjectData
2. {
3. Q\_DECLARE\_PUBLIC(QObject)
4. ...
5. };
7. class Q\_GUI\_EXPORT QWidgetPrivate : public QObjectPrivate
8. {
9. Q\_DECLARE\_PUBLIC(QWidget)
10. ...
11. };
13. class Q\_GUI\_EXPORT QWidget : public QObject
14. {
15. ...
16. Q\_DECLARE\_PRIVATE(QWidget)
17. ...
18. public:
19. ...
20. explicit QWidget(QWidget\* parent = 0, Qt::WindowFlags f = 0);
21. ...
22. };

我们首先来看看Q\_DECLARE\_PUBLIC宏：

1. #define Q\_DECLARE\_PUBLIC(Class) \
2. inline Class\* q\_func() { return static\_cast<Class \*>(q\_ptr); } \
3. inline const Class\* q\_func() const { return static\_cast<const Class \*>(q\_ptr); } \
4. friend class Class;

根据宏定义，Q\_DECLARE\_PUBLIC(QObject)翻译如下：

1. inline QObject \*q\_func()
2. {
3. return static\_cast<QObject \*>(q\_ptr);
4. }
5. inline const QObject \*q\_func() const
6. {
7. return static\_cast<const QObject \*>(q\_ptr);
8. }
9. friend class QObject;

Q\_DECLARE\_PUBLIC(QWidget)翻译如下：

1. inline QWidget \*q\_func()
2. {
3. return static\_cast<QWidget \*>(q\_ptr);
4. }
5. inline const QWidget \*q\_func() const
6. {
7. return static\_cast<const QWidget \*>(q\_ptr);
8. }
9. friend class QWidget;

注意这里的q\_ptr是在QObjectData里公有声明的，QObjectPrivate,QWidgetPrivate都派生或间接派生自QObjectData,所以可以访问q\_ptr。

接下来看Q\_DECLARE\_PRIVATE(QWidget)的翻译：

1. inline QWidgetPrivate \*d\_func()
2. {
3. return reinterpret\_cast<QWidgetPrivate \*>(qGetPtrHelper(d\_ptr));
4. }
5. inline const QWidgetPrivate \*d\_func() const
6. {
7. return reinterpret\_cast<const QWidgetPrivate \*>(qGetPtrHelper(d\_ptr));
8. }
9. friend class QWidgetPrivate;

接下来看看QWidget的构造函数的实现：

1. QWidget::QWidget(QWidget \*parent, Qt::WindowFlags f)
2. : QObject(\*new QWidgetPrivate, 0)
3. {
4. ...
5. }

看到QObject(\*new QwidgetPrivate, 0)这里调用了QObject的第二个构造函数，将d\_ptr指向new QWidgetPrivate所指向的堆内存。