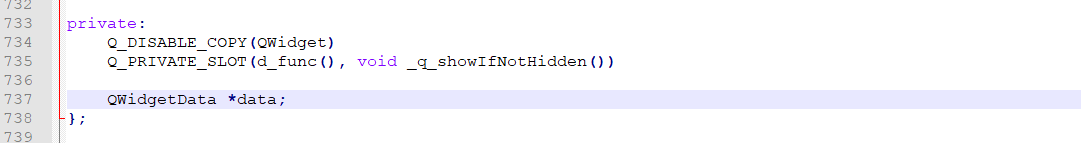
[Qt源码解析之-从PIMPL机制到d指针](https://www.cnblogs.com/sherlock-lin/p/12862425.html)



**一、PIMPL机制**

**PIMPL** ，即Private Implementation，作用是，实现 **私有化，力图使得头文件对改变不透明**，以达到解耦的目的

pimpl 用法背后的思想是把客户与所有关于类的私有部分的知识隔离开。由于客户是依赖于类的头文件的，头文件中的任何变化都会影响客户，即使仅是对私有节或保护节的修改。pimpl用法隐藏了这些细节，方法是将私有数据和函数放入一个单独的类中，并保存在一个实现文件中，然后在头文件中对这个类进行前向声明并保存一个指向该实现类的指针。类的构造函数分配这个pimpl类，而析构函数则释放它。这样可以消除头文件与实现细节的相关性

该句出自 ***超越 C++ 标准库--boost程序库 导论***

**（1）实例说明**

假设现在有一个需求，你需要写一个类，来完成产品的信息保存和获取，这个需求看起来非常的简单，我们只需要一分钟就能写好

**Product.h**

class Product

{

public:

string GetName() const;

void SetName(const string& mName);

float GetPrice() const;

void SetPrice(float mPrice);

private:

string mName;

float mPrice;

};

**Product.cpp**

string Product::GetName() const

{

return this->mName;

}

void Product::SetName(const string &name)

{

this->mName = name;

}

float Product::GetPrice() const

{

return this->mPrice;

}

void Product::setPrice(float price)

{

this->mPrice = price;

}

利用PIMPL机制，将私有成员隐藏起来，使得只有接口不变，那么头文件就不会改变，已达到解耦的目的。PIMPL机制的好处之一就是避免头文件依赖，提高编译速度。

那利用PIMPL机制，上面的问题如何解决呢？

**（2）利用PIMPL机制**

基于原来的需求，代码设计如下：

**Product.h**

class ProductData;

class Product

{

public:

Product();

~Product();

string getName() const;

void setName(const string& name);

float getPrice() const;

void setPrice(float price);

private:

ProductData\* data;

};

**Product.cpp**

class ProductData

{

public:

string name;

float price;

};

Product::Product()

{

data = new ProductData();

}

Product::~Product()

{

delete data;

}

string Product::getName() const

{

return data->name;

}

void Product::setName(const string &name)

{

data->name = name;

}

float Product::getPrice() const

{

return data->price;

}

void Product::setPrice(float price)

{

data->price = price;

}

可以看出来，Product 类除了必要的接口函数外，就只有一个ProductData指针了，而ProductData又是使用的前置声明，在cpp中实现，这样，只要接口不变，那么内部私有成员或者逻辑改变，并不会影响client

上面的代码只是最简单的实现，其中还存在很多问题，而实际的项目中可能要复杂的多。尽管如此，我们也能看出PIMPL机制的优点：

* 降低耦合度；
* 隐藏模块信息；
* 降低编译依赖，提高编译速度；
* 接口和实现真正分离；

**二、Qt源码中的d指针**

了解PIMPL机制之后，我们可以看看优秀的C++库中是如何实现PIMPL机制的，以Qt框架为例。读过Qt源码的同学对Qt中的d指针想必不会陌生，我们来详细讲解一下

**（1）QThread中的PIMPL机制**

我们随便选取一个Qt中的模块，以QThread为例分析一下Qt中是如何实现PIMPL机制的

首先，找到QThread类的头文件 qthread.h，我们可以看到，QThread 类的什么中，除了对外的接口外，根本看不到能够猜测内部实现方法或者变量，而且其private的成员只有下面几个

class Q\_CORE\_EXPORT QThread : public QObject

{

...

private:

Q\_DECLARE\_PRIVATE(QThread)

friend class QCoreApplication;

friend class QThreadData;

};

那么，QThread内部使用的方法和属性都去哪里了呢

我们先找到它的构造函数，实现如下：

QThread::QThread(QObject \*parent)

: QObject(\*(new QThreadPrivate), parent)

{

Q\_D(QThread);

d->data->thread = this;

}

**（2）Q\_D宏**

**Q\_D()** 是Qt中的一个宏定义

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

**Q\_D(QThread);** 展开如下：

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

这也是上面代码中的 **d** 指针的由来，可以看到，d其实是一个QThreadPrivate指针，

Q\_D(QThread); //QThread\*

Q\_D(const QThread); //const QThread\*

**（3）d\_func()**

这里有一个方法 **d\_func()**，我们可以查看到它的声明

#define Q\_DECLARE\_PRIVATE(Class) \

inline Class##Private\* d\_func() { \

return reinterpret\_cast<Class##Private \*>(qGetPtrHelper(d\_ptr));} \

inline const Class##Private\* d\_func() const { \

return reinterpret\_cast<const Class##Private \*>(qGetPtrHelper(d\_ptr));} \

friend class Class##Private;

上面这段代码会生成方法 d\_func()，而在QThread 头文件类声明中，可以看到此宏

class Q\_CORE\_EXPORT QThread : public QObject

{

...

private:

Q\_DECLARE\_PRIVATE(QThread)

...

};

**（4）Q\_DECLARE\_PRIVATE宏**

**Q\_DECLARE\_PRIVATE(QThread)** 宏展开如下：

class Q\_CORE\_EXPORT QThread : public QObject

{

...

private:

inline QThreadPrivate\* d\_func(){

return reinterpret\_cast<QThreadPrivate\*>(qGetPtrHelper(d\_ptr));

}

inline const QThreadPrivate\* d\_func() const {

return reinterpret\_cast<QThreadPrivate\*>(qGetPtrHelper(d\_ptr));

}

friend class QThreadPrivate;

};

可以看出，这个宏其实是在QThread内部定义了两个 inline 方法和一个友元类，d\_func() 方法也来源于此

**（5）qGetPtrHelper()方法**

这里的 **qGetPtrHelper()** 方法可以找到，我给它重新排版一下，如下：

template <typename T> static inline T \*qGetPtrHelper(T \*ptr) {

return ptr;

}

template <typename Wrapper> static inline typename Wrapper::pointer qGetPtrHelper(const Wrapper &p) {

return p.data();

}

这是一个模板方法，如果只是一个普通的类指针，则返回该指针；

而如果是一个类模板方法，则调用data()方法，并返回结果

所以，我们看到上面的 **d\_func()** 方法中的，替换了 **qGetPtrHelper()** 方法后，如下：

class Q\_CORE\_EXPORT QThread : public QObject

{

...

inline QThreadPrivate\* d\_func(){

return reinterpret\_cast<QThreadPrivate\*>(d\_ptr.data());

}

inline const QThreadPrivate\* d\_func() const {

return reinterpret\_cast<QThreadPrivate\*>(d\_ptr.data());

}

};

**（6）d\_ptr**

那么，这里的 **d\_ptr** 又是哪里来的呢，它其实是在 **QObject** 对象内定义的

class Q\_CORE\_EXPORT QObject

{

...

protected:

QScopedPointer<QObjectData> d\_ptr;

};

QObject::QObject(QObject \*parent)

: d\_ptr(new QObjectPrivate)

{

…

}

我们都知道，Qt 中所有对象都是继承自 **QObject** 的，所以 **QThread** 是可以使用 d\_ptr 的

**QScopedPointer** 是Qt中封装的智能指针，相当于stl中的std::unique\_ptr，所以，上面代码中的 **d\_ptr.data()** 作用是获取智能指针管理的指针，等同于std::unique\_ptr中的 **get()** 方法，也就是这里的 **QObjectData** 指针。

所以，**d\_func()** 方法的作用是，获取 **QThread** 中继承的 **QObject** 中的 **QObjectData** 指针，并使用强制类型转换为 **QThreadPrivate** 指针类型。而为什么能转换，因为他们之间是有继承关系的 **QThreadPrivate -> QObjectPrivate -> QObjectData**

而QObjectData，是用来保存标注位，当前类的实例，父类和子类们等：

class Q\_CORE\_EXPORT QObjectData {

public:

virtual ~QObjectData() = 0;

QObject \*q\_ptr;

QObject \*parent;

QObjectList children;

uint isWidget : 1;

uint blockSig : 1;

uint wasDeleted : 1;

uint isDeletingChildren : 1;

uint sendChildEvents : 1;

uint receiveChildEvents : 1;

uint isWindow : 1; //for QWindow

uint deleteLaterCalled : 1;

uint unused : 24;

int postedEvents;

QDynamicMetaObjectData \*metaObject;

QMetaObject \*dynamicMetaObject() const;

};

上面说的所有东西最终都是在分析 **Q\_D(QThread);**，现在我们知道，这句宏定义最后会得到 **QThreadPrivate** 指针，而这个类的作用就是我们之前讲的 PIMPL机制中的用作保存 **QThread** 类私有成员，以达到解耦的目的.