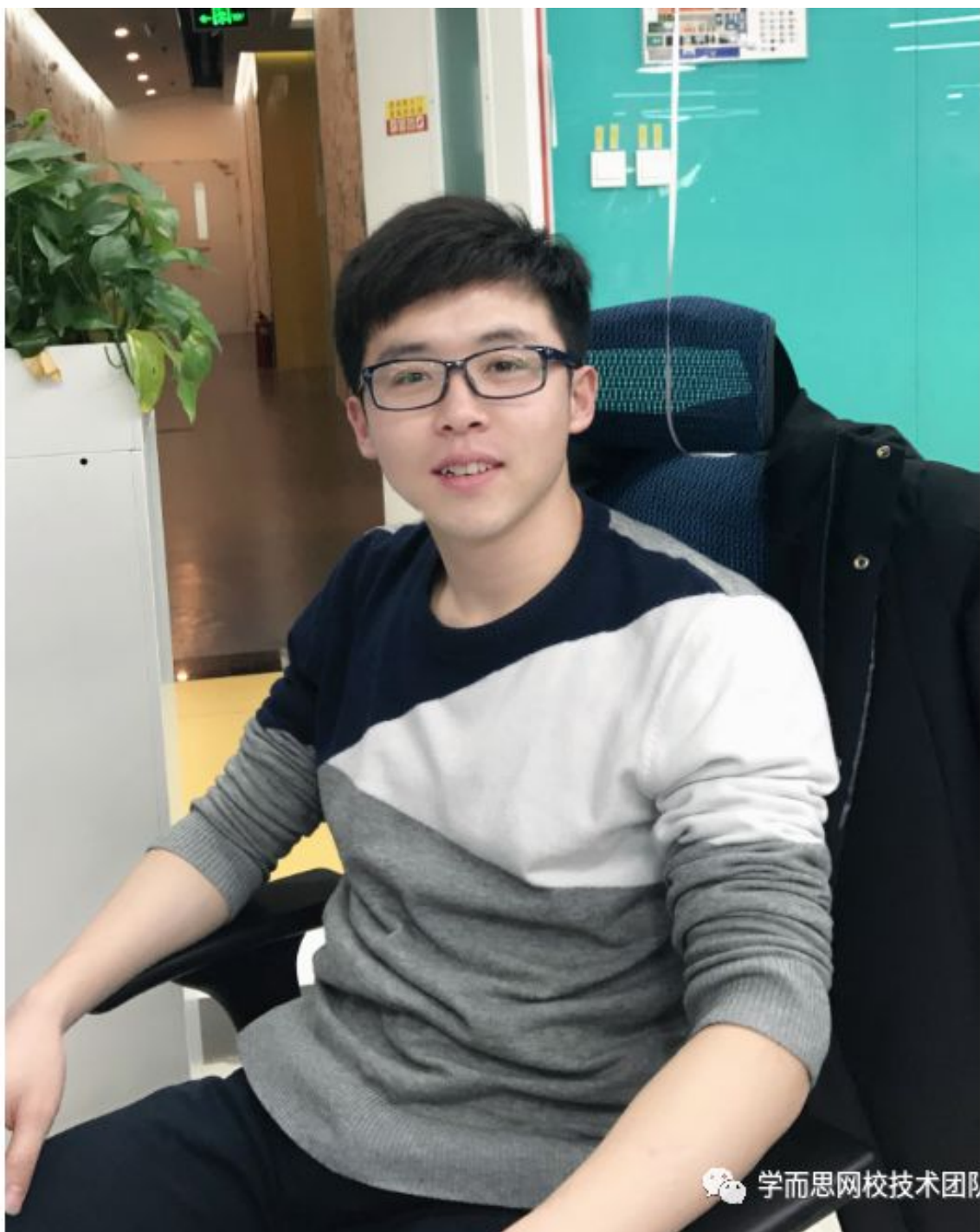


Qt 信号和槽源码分析

曹存 学而思网校技术团队 2019-08-23



分享老师：曹存

引言：

Qt 是一个1991年由Qt Company开发的跨平台C++图形用户界面应用程序开发框架。它既可以开发GUI程序，也可用于开发非GUI程序，比如控制台工具和服务器。Qt是面向对象的框架，使用特殊的代码生成扩展（称为元对象编译器(Meta Object Compiler, moc)) 以及一些宏，Qt很容易扩展，并且允许真正地组件编程。Qt是跨平台开发框架，支持Windows、Linux、MacOS等不同平台；Qt有大量的开发文档和丰富的API，给开发者带来了很大的方便；Qt的使用者也越来越多，有很多优秀的产品都基于Qt开发，如：WPS Office、Opera浏览器、Qt Creator等。Qt的核心机制就是信号和槽，接下来我们通过源代码分析一下实现原理。

基本概念：

信号：当对象改变其状态时，信号就由该对象发射 (emit) 出去，而且对象只负责发送信号，它不知道另一端是谁在接收这个信号。

槽：用于接收信号，而且槽只是普通的对象成员函数。一个槽并不知道是否有任何信号与自己相连接。

信号与槽的连接：所有从 QObject 或其子类（例如 QWidget）派生的类都能够包含信号和槽。是通过静态方法：QObject::connect(sender, SIGNAL(signal), receiver, SLOT(slot)); 来进行管理的，其中 sender 与 receiver 是指向对象的指针，SIGNAL() 与 SLOT() 是转换信号与槽的宏。

实现原理：

1、首先我们搭建好环境，如在Windows系统上：安装Qt5.7（包括源码） + VS2013 及 对应的插件，我们主要是通过VS来进行编译调试的。

2、我们写一个简单实例，然后进行构建，再把Qt安装目录中的QtCored的pdb拷贝到我们的可执行文件目录下面，如下图所示：

Demo.exe	2019/6/27 14:32	应用程序	84 KB
Demo.ilc	2019/6/27 14:32	Incremental Link...	1,163 KB
demo.pdb	2019/6/27 14:32	Program Debug...	2,668 KB
main.obj	2019/6/27 14:32	3D Object	147 KB
MainWindow.obj	2019/6/27 14:32	3D Object	243 KB
moc_MainWindow.cpp	2019/6/27 14:32	C++ Source file	7 KB
moc_MainWindow.obj	2019/6/27 14:32	3D Object	146 KB
Qt5Cored.pdb	2016/6/10 15:20	Program Debug...	26,460 KB

下面是我们要分析的Demo代码：

```
// MainWindow.h
```

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPushButton>

namespace Ui {
class MainWindow;
}

class Test : public QWidget
{
    Q_OBJECT // 要实现信号和槽，必须要有这个宏
public:
    explicit Test(QWidget *parent = 0);
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);

signals:
    void clean();

private slots:
    void onDestory();


private:
    QPushButton *m_button;
};

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void onClean();

private:
    Ui::MainWindow *ui;
    Test *m_testWidget;
};
#endif // MAINWINDOW_H

```

 学网网技术团队

// MainWindow.cpp

```

#include "MainWindow.h"
#include "ui_MainWindow.h"

#include <QPainter>
#include <QDebug>

Test::Test(QWidget *parent)
    : QWidget(parent)
    , m_button(NULL)
{
    m_button = new QPushButton(this);
    m_button->setFixedSize(30, 30);
    connect(m_button, SIGNAL(clicked()), this, SLOT(onDestory()));
}

void Test::paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event)
    QPainter painter(this);
    painter.fillRect(rect(), QColor(255, 255, 255));
}

void Test::resizeEvent(QResizeEvent *event)
{
    m_button->move((width() - m_button->width()) / 2, (height() - m_button->height()) / 2);
}


void Test::onDestory()
{
    emit clean(); // 触发一个信号
    qDebug() << "Test::onDestory over.";
}

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    m_testWidget(NULL)
{
    ui->setupUi(this);
    m_testWidget = new Test(this);
    m_testWidget->setFixedSize(200, 200);
    ui->gridLayout->addWidget(m_testWidget);
    connect(m_testWidget, SIGNAL(clean()), this, SLOT(onClean()));
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::onClean()
{
    qDebug() << "MainWindow::onClean";
}

```

 学而思网校技术团队

我们可以创建一个Qt工程，名称为Demo，编写上面的代码，进行构建，在VS下可以把Qt工程导成VS工程，编译生成，运行结果如下：



点击中间的按钮，我们可以看到控制台打印如下信息：

```
MainWindow::onClean
Test::onDestory over.
```

学而思网校技术团队

第一步：基本结构：

我们分析代码，可以看到在头文件Test和MainWindow类中，都有Q_OBJECT这样的宏，然后我们可以看到上面的可执行文件夹下多出来一个moc_MainWindow.cpp文件，那么我们可以尝试把这两个宏去掉，再进行构建，发现加上了信号和槽的就无法编译过去，我们去掉这些信号和槽后，就不会生成moc开头的这个文件了，当然我们就无法实现信号和槽机了，那么这个宏到底是什么，有了它编译器又会做什么？让我们看看这个宏：

```
#define Q_OBJECT \
public: \
    static const QMetaObject staticMetaObject; \
    virtual const QMetaObject *metaObject() const; \
    virtual void *qt_metacast(const char *); \
    virtual int qt_metacall(QMetaObject::Call, int, void **); \
private: \
    static void qt_static_metacall(QObject *, QMetaObject::Call, int, void **); \
    struct QPrivateSignal {};
```

moc (Meta-Object Compiler) 又称“元对象编译器”完成

学而思网校技术团队

原来这个宏就是一些静态方法和虚方法，但是如果我们将加入到类中，不进行实现，那一定会报错的，为什么还可以正常运行呢？原来Qt帮我们做了很多事情，在编译器编译Qt代码之前，Qt先将Qt自身扩展的语法进行翻译，这个操作是通过moc（Meta-Object Compiler）又称“元对象编译器”完成的。首先moc会分析源代码，把包含Q_OBJECT的头文件生成为一个C++源文件，这个文件的名字会是源文件名前面加上moc_，之后和原文件一起通过编译器处理，那我们想到，这个moc开头的cpp中一定实

现了上面宏里面的方法，以及数据的赋值；接下来我们看看moc_MainWindow.cpp这个文件：

```
#include "../Demo/MainWindow.h"
struct qt_meta_stringdata_Test_t {
    QByteArrayData data[4];
    char stringdata0[22];
};
#define QT_MOC_LITERAL(idx, ofs, len)
Q_STATIC_BYTE_ARRAY_DATA_HEADER_INITIALIZER_WITH_OFFSET(len, \
    qptrdiff(offsetof(qt_meta_stringdata_Test_t, stringdata0) + ofs - idx *
sizeof(QByteArrayData)))
static const qt_meta_stringdata_Test_t qt_meta_stringdata_Test = {{
    QT_MOC_LITERAL(0, 0, 4), // "Test"
    QT_MOC_LITERAL(1, 5, 5), // "clean"
    QT_MOC_LITERAL(2, 11, 0), // ""
    QT_MOC_LITERAL(3, 12, 9) // "onDestory"
},
    "Test\0clean\0\0onDestory"
};
#undef QT_MOC_LITERAL
static const uint qt_meta_data_Test[] = {
    // content:
    7,      // revision
    0,      // classname
    0, 0,  // classinfo
    2, 14, // methods
    0, 0,  // properties
    0, 0,  // enums/sets
    0, 0,  // constructors
    0,     // flags
    1,     // signalCount
    // signals: name, argc, parameters, tag, flags
    1, 0, 24, 2, 0x06 /* Public */,
    // slots: name, argc, parameters, tag, flags
    3, 0, 25, 2, 0x08 /* Private */,
    ...
};
void Test::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id, void
**_a)
{
    if (_c == QMetaObject::InvokeMetaMethod) {
        Test *_t = static_cast<Test *>(_o);
        Q_UNUSED(_t)
        switch (_id) {
            case 0: _t->clean(); break;
            case 1: _t->onDestory(); break;
            default: ;
        }
    }
    ....
}
const QMetaObject Test::staticMetaObject = {
    { &QWidget::staticMetaObject, qt_meta_stringdata_Test.data,
      qt_meta_data_Test, qt_static_metacall, Q_NULLPTR, Q_NULLPTR}
};
const QMetaObject *Test::metaObject() const
{
    return QObject::d_ptr->metaObject ? QObject::d_ptr->dynamicMetaObject() :
&staticMetaObject;
}
void *Test::qt_metacast(const char *_cname)
{
    if (!_cname) return Q_NULLPTR;
    if (!strcmp(_cname, qt_meta_stringdata_Test.stringdata0))
        return static_cast<void*>(const_cast< Test*>(this));
    return QWidget::qt_metacast(_cname);
}
```

```


int Test::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
{
    _id = QWidget::qt_metacall(_c, _id, _a);
    if (_c == QMetaObject::InvokeMetaMethod) {
        if (_id < 2)
            qt_static_metacall(this, _c, _id, _a);
        _id -= 2;
    }
    ...
    return _id;
}
// SIGNAL 0
void Test::clean()
{
    QMetaObject::activate(this, &staticMetaObject, 0, Q_NULLPTR);
}
struct qt_meta_stringdata_MainWindow_t {
    QByteArrayData data[3];
    char stringdata0[20];
};
#define QT_MOC_LITERAL(idx, ofs, len)
Q_STATIC_BYTE_ARRAY_DATA_HEADER_INITIALIZER_WITH_OFFSET(len, \
    qptrdiff(offsetof(qt_meta_stringdata_MainWindow_t, stringdata0) + ofs- idx *
sizeof(QByteArrayData)))
static const qt_meta_stringdata_MainWindow_t qt_meta_stringdata_MainWindow = {{
    QT_MOC_LITERAL(0, 0, 10), // "MainWindow"
    QT_MOC_LITERAL(1, 11, 7), // "onClean"
    QT_MOC_LITERAL(2, 19, 0) // ""
},
    "MainWindow\0onClean\0"
};
#undef QT_MOC_LITERAL
static const uint qt_meta_data_MainWindow[] = {
    // content:
    7,          // revision
    0,          // classname
    0,    0,    // classinfo
    1,    14,   // methods
    0,    0,   // properties
    0,    0,   // enums/sets
    0,    0,   // constructors
    0,         // flags
    0,         // signalCount
    // slots: name, argc, parameters, tag, flags
    1,    0,    19,    2, 0x08 /* Private */,
    ...
};
void MainWindow::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id,
void **_a)
{
    if (_c == QMetaObject::InvokeMetaMethod) {
        MainWindow *_t = static_cast<MainWindow *>(_o);
        Q_UNUSED(_t)
        switch (_id) {
            case 0: _t->onClean(); break;
            default: ;
        }
    }
}
const QMetaObject MainWindow::staticMetaObject = {
    { &QMainWindow::staticMetaObject, qt_meta_stringdata_MainWindow.data,
      qt_meta_data_MainWindow, qt_static_metacall, Q_NULLPTR, Q_NULLPTR }
};
const QMetaObject *MainWindow::metaObject() const
{
    return QObject::d_ptr->metaObject ? QObject::d_ptr->dynamicMetaObject() :
&staticMetaObject;
}
void *MainWindow::qt_metacast(const char *_cname)
{
    ,

```

```

1
    if (!_cname) return Q_NULLPTR;
    if (!strcmp(_cname, qt_meta_stringdata_MainWindow.stringdata0))
        return static_cast<void*>(const_cast< MainWindow*>(this));
    return QMainWindow::qt_metacast(_cname);
}
int QMainWindow::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
{
    _id = QMainWindow::qt_metacall(_c, _id, _a);
    if (_c == QMetaObject::InvokeMetaMethod) {
        if (_id < 1)
            qt_static_metacall(this, _c, _id, _a);
        _id -= 1;
    }
    ...
    return _id;
}

```


 学而思网校技术团队

我们从上面的代码中可以看到，是对Q_OBJECT中的静态数据进行了赋值，并且实现了那些方法，这些都是Qt的moc编译器帮我们生成的，对代码进行了分析，对信号和槽生成了符号，以及特定的数据结构，下面这个主要是记录了类、信号、槽的引用计数、大小、偏移，后面会用到。

```

static const qt_meta_stringdata_Test_t qt_meta_stringdata_Test =
{
    {
        QT_MOC_LITERAL(0, 0, 4), // "Test"
        QT_MOC_LITERAL(1, 5, 5), // "clean"
        QT_MOC_LITERAL(2, 11, 0), // ""
        QT_MOC_LITERAL(3, 12, 9) // "onDestroy"
    },
    "Test\0clean\0\0onDestroy"
};

```


 学而思网校技术团队

通过把QT_MOC_LITERAL这个宏进行替换后，得到如下数据：

```

static const qt_meta_stringdata_Test_t qt_meta_stringdata_Test =
{
    { // -1 size 0 0 Offset
        {-1, 4, 0, 0, 0}, // "Test"
        {-1, 5, 0, 0, 5}, // "clean"
        {-1, 0, 0, 0, 11}, // ""
        {-1, 9, 0, 0, 12} // "onDestroy"
    },
    "Test\0clean\0\0onDestroy"
};

```

 学而思网校技术团队

接下来我们看看下面qt_meta_data_MainWindow这个数组结构：**content**有两列，第一列是总数，第二列是在这个数组中描述开始的索引，如1, 14, // methods，说明有一个methods，我们可以看到slots就是从索引14开始的。


```
static const uint qt_meta_data_MainWindow[] = {
    // content:
    7,          // revision
    0,          // classname
    0,    0,    // classinfo
    1,   14,    // methods
    0,    0,    // properties
    0,    0,    // enums/sets
    0,    0,    // constructors
    0,          // flags
    0,          // signalCount
    // slots: name, argc, parameters, tag, flags
    1,    0,   19,    2, 0x08 /* Private */,
};
```

学而思网校技术团队

从最上面的源代码中我们可以看到再关联信号和槽的时候，用到了SIGNAL和SLOT这两个宏，那么这两个宏到底有什么作用呢？我们分析一下：

```
Q_CORE_EXPORT const char *qFlagLocation(const char *method);
#define QLOCATION "\0" __FILE__ ":" QT_STRINGIFY(__LINE__)
#define SLOT(a)    qFlagLocation("1"#a QLOCATION)
#define SIGNAL(a)  qFlagLocation("2"#a QLOCATION)
```

学而思网校技术团队

分析：

从上面我们可以看到其实这两个就是一个字符串拼接的宏，会在信号(signal)前面拼接"2"，如"2clean()"；会在槽(slots)前面拼接"1"，如"1onClean()"；其中，qFlagLocation这个方法主要是把method存储在QThreadData里面FlaggedDebugSignatures中的const char* locations[Count];表中，用于定位代码对应的行信息。

```
connect(m_testWidget, SIGNAL(clean()), this, SLOT(onClean()));
```

学而思网校技术团队

```
SLOT(onClean()):
```

预编译后如下：

```
connect(m_testWidget, qFlagLocation("2"clean() "\0" "MainWindow.cpp" ":" "57"), this,
qFlagLocation("1"onClean() "\0" "MainWindow.cpp" ":" "57"));
```

学而思网校技术团队

通过上面的一些基本宏、数据结构的介绍，我们知道Qt给我们做了很多工作，帮我们生成了moc代码，给我们提供了一些宏，让我们开发简洁方便，那么Qt又是如何把信号和槽进行关联的呢，就是两个不同的实例，又是如何进行通过信号槽机制进行通信的呢？接下来我们看看信号和槽关联的实现原理：

第二步、信号和槽的关联：

```

#include "MainWindow.h"
#include "ui_MainWindow.h"

#include <QPainter>
#include <QDebug>

QMetaObject::Connection QObject::connect(const QObject *sender, const char
*signal,
                                     const QObject *receiver, const char *method,
                                     Qt::ConnectionType type) {
    // 参数判空
    if (sender == 0 || receiver == 0 || signal == 0 || method == 0) {
        return QMetaObject::Connection(0);
    }
    // 信号
    //1、检查信号是否合法
    if (!check_signal_macro(sender, signal, "connect", "bind"))
        return QMetaObject::Connection(0);
    //2、获取发送者元数据
    const QMetaObject *smeta = sender->metaObject();
    // 3、对信号参数、方法名称进行获取和保存
    QByteArray signalName = QMetaObjectPrivate::decodeMethodSignature(signal,
signalTypes);
    // 4、计算信号索引（包括基类）
    int signal_index = QMetaObjectPrivate::indexOfSignalRelative(smeta,
signalName, signalTypes.size(), signalTypes.constData());
    // 5、对掩码（信号）进行检查，并返回偏移
    signal_index = QMetaObjectPrivate::originalClone(smeta, signal_index);
    signal_index += QMetaObjectPrivate::signalOffset(smeta);


    // 槽（流程和信号同理）
    int membcodes = extract_code(method);
    if (!check_method_code(membcodes, receiver, method, "connect"))
        return QMetaObject::Connection(0);
    QByteArray methodName = QMetaObjectPrivate::decodeMethodSignature(method,
methodTypes);
    const QMetaObject *rmeta = receiver->metaObject();
    int method_index_relative = QMetaObjectPrivate::indexOfSlotRelative(smeta,
methodName, methodTypes.size(), methodTypes.constData());
    // 判断链接类型
    if ((type == Qt::QueuedConnection) && !(types =
queuedConnectionTypes(signalTypes.constData(), signalTypes.size()))) {
        return QMetaObject::Connection(0);
    }
    // 检查signals和method的参数个数和类型是否一致
    QMetaObjectPrivate::checkConnectArgs(signalTypes.size(),
signalTypes.constData(), methodTypes.size(), methodTypes.constData());
    QMetaObject::Connection handle =
QMetaObject::Connection(QMetaObjectPrivate::connect(
    sender, signal_index, smeta, receiver, method_index_relative, rmeta ,type,
types));
    return handle;
}

```

学而思网校技术团队


- 1、先对信号和槽的字符串进行检查，Q_SIGNAL_CODE 是 1；SIGNAL_CODE 是 2。

```
static int extract_code(const char *member) {
    return (((int) (*member) - '0') & 0x3); // // 看到这个就是获取到了前面的数字：信号是
    2，槽是1
}
static bool check_signal_macro(const QObject *sender, const char *signal, const
char *func, const char *op) {
    int sigcode = extract_code(signal);
    if (sigcode != Q_SIGNAL_CODE) { return false;}
    return true;
}
```

 学而思网校技术团队


2、获取元数据（sender和receiver同理）。

```
const QMetaObject *smeta = sender->metaObject();
```

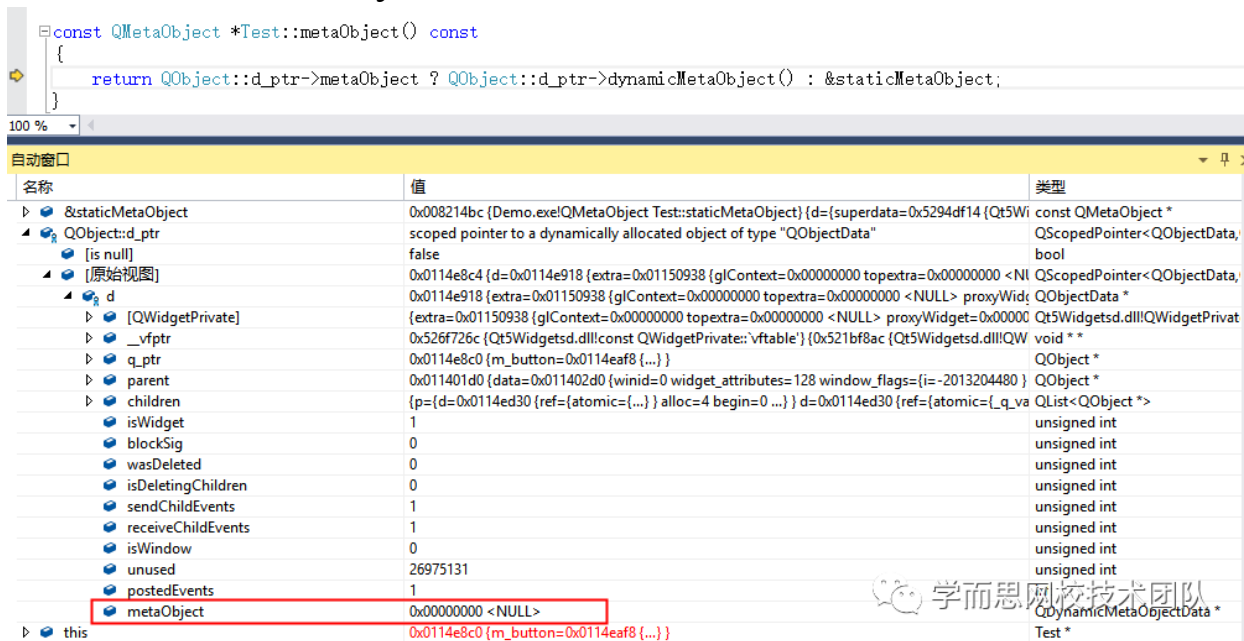
 学而思网校技术团队

这个方法就是我们上面moc_MainWindow.cpp中。

```
const QMetaObject Test::staticMetaObject = {
    { &QWidget::staticMetaObject, qt_meta_stringdata_Test.data,
      qt_meta_data_Test, qt_static_metacall, Q_NULLPTR, Q_NULLPTR}
};
const QMetaObject *Test::metaObject() const {
    return QObject::d_ptr->metaObject ? QObject::d_ptr->dynamicMetaObject() :
    &staticMetaObject;
}
```


 学而思网校技术团队

我们根据调试可以看到QObject::d_ptr->metaObject是空的，所以这样smeta就是上面这个staticMetaObject变量了。



The screenshot shows the Qt Creator debugger interface. The top pane displays the source code of the `Test::metaObject()` function, which returns `QObject::d_ptr->metaObject ? QObject::d_ptr->dynamicMetaObject() : &staticMetaObject;`. The bottom pane shows the 'Automatic' window with a list of variables. The variable `metaObject` is highlighted with a red box, showing its value as `0x00000000 <NULL>`. Other variables like `&staticMetaObject`, `QObject::d_ptr`, and various widget attributes are also listed.

名称	值	类型
<code>&staticMetaObject</code>	<code>0x008214bc (Demo.exe QMetaObject Test::staticMetaObject) {d={superdata=0x5294df14 {Qt5W...</code>	<code>const QMetaObject *</code>
<code>QObject::d_ptr</code>	<code>scoped pointer to a dynamically allocated object of type "QObjectData"</code>	<code>QScopedPointer<QObjectData,</code>
<code>[is null]</code>	<code>false</code>	<code>bool</code>
<code>[原始视图]</code>	<code>0x0114e8c4 {d=0x0114e918 {extra=0x01150938 {glContext=0x00000000 toextra=0x00000000 <NI...</code>	<code>QScopedPointer<QObjectData,</code>
<code>d</code>	<code>{extra=0x01150938 {glContext=0x00000000 toextra=0x00000000 <NULL> proxyWidget=0x00000000 Qt5Widgets.d...</code>	<code>QObjectData *</code>
<code>[QWidgetPrivate]</code>	<code>{extra=0x01150938 {glContext=0x00000000 toextra=0x00000000 <NULL> proxyWidget=0x00000000 Qt5Widgets.d...</code>	<code>QObjectData *</code>
<code>_vfp_ptr</code>	<code>0x526f726c (Qt5Widgets.d.dll const QWidgetPrivate::vftable) {0x521bf8ac (Qt5Widgets.d.dll QW...</code>	<code>void *</code>
<code>q_ptr</code>	<code>0x0114e8c0 {m_button=0x0114eaf8 {...}}</code>	<code>QObject *</code>
<code>parent</code>	<code>0x011401d0 {data=0x011402d0 {winid=0 widget_attributes=128 window_flags={i=-2013204480} QObje...</code>	<code>QObject *</code>
<code>children</code>	<code>{p={d=0x0114ed30 {ref={atomic={...}} alloc=4 begin=0 ...}} d=0x0114ed30 {ref={atomic={_q_va...</code>	<code>QList<QObject *></code>
<code>isWidget</code>	<code>1</code>	<code>unsigned int</code>
<code>blockSig</code>	<code>0</code>	<code>unsigned int</code>
<code>wasDeleted</code>	<code>0</code>	<code>unsigned int</code>
<code>isDeletingChildren</code>	<code>0</code>	<code>unsigned int</code>
<code>sendChildEvents</code>	<code>1</code>	<code>unsigned int</code>
<code>receiveChildEvents</code>	<code>1</code>	<code>unsigned int</code>
<code>isWindow</code>	<code>0</code>	<code>unsigned int</code>
<code>unused</code>	<code>26975131</code>	<code>unsigned int</code>
<code>postedEvents</code>	<code>1</code>	<code>unsigned int</code>
<code>metaObject</code>	<code>0x00000000 <NULL></code>	<code>QDynamicMetaObjectData *</code>
<code>this</code>	<code>0x0114e8c0 {m_button=0x0114eaf8 {...}}</code>	<code>Test *</code>


 学而思网校技术团队

// 首先我们得了解一下这个QMetaObject 和 QMetaObjectPrivate 的定义：

```

struct Q_CORE_EXPORT QMetaObject {
    ...
    static Connection connect(const Object *sender, int signal_index, const Object
*receiver, int method_index, int type = 0, int *types = nullptr);
    static bool disconnect(const Object *sender, int signal_index, const Object
*receiver, int method_index);
    struct { // private data
        const QMetaObject *superdata;
        const QByteArrayData *stringdata;
        const uint *data;
        typedef void (*StaticMetacallFunction)(QObject *, QMetaObject::Call, int,
void **);
        StaticMetacallFunction static_metacall;
        const QMetaObject * const *relatedMetaObjects;
        void *extradata; //reserved for future use
    } d;
};

```

 学而思网校技术团队

在Qt中为了实现二进制兼容性，一般会定义一个私有类，QMetaObjectPrivate就是QMetaObject的私有类，QMetaObject负责一些接口实现，QMetaObjectPrivate具体进行实现，这两个类一般是通过P指针和D指针进行组合式的访问，有一个宏：

```

#define Q_D(Class) Class##Private * const d = d_func()
#define Q_Q(Class) Class * const q = q_func()

```

 学而思网校技术团队

```

struct QMetaObjectPrivate
{
    int revision;
    int className;
    int classInfoCount, classInfoData;
    int methodCount, methodData;
    int propertyCount, propertyData;
    int enumeratorCount, enumeratorData;
    int constructorCount, constructorData; //since revision 2
    int flags; //since revision 3
    int signalCount; //since revision 4
    static inline const MetaObjectPrivate *get(const MetaObject *metaobject);
    static ObjectPrivate::Connection *connect(const Object *sender, int
signal_index,const MetaObject *smeta,
const Object *receiver, int
method_index,const MetaObject *rmeta = 0,
int type = 0, int *types = 0);
    ...
};

```

 学而思网校技术团队

我们看上面的staticMetaObject是一个QMetaObject类型的变量，其中QMetaObject进行了赋值：

- 1) &QWidget::staticMetaObject (父对象的MetaObject) -> superdata
- 2) qt_meta_stringdata_Test.data -> stringdata
- 3) qt_meta_stringdata_Test() -> data
- 4) qt_static_metacall (回调函数) -> static_metacall

其中QMetaObject 是对外的结构，里面的connect方法最终调用的还是QMetaObjectPrivate里面的connect进行实现的。QMetaObject里的d成员填充了上面的staticMetaObject数据，而QMetaObjectPrivate里面的成员填充qt_meta_stringdata_Test数组中的数据，我们可以看到填充前14个数据，这也是moc生成methodData时以14为基数的原因了，转换方法如下：

```
static inline const QMetaObjectPrivate *priv(const uint* data)
{ return reinterpret_cast<const QMetaObjectPrivate*>(data); }
```

 学而思网校技术团队


```
indexOfSignalRelative(&smeta, signalName,
```

3、对信号参数、名称进行获取和保存，如下，把信号的参数保存起来，返回方法名称。

```
QByteArray signalName = QMetaObjectPrivate::decodeMethodSignature(signal, signalTypes);
```

 学而思网校技术团队

```
QByteArray QMetaObjectPrivate::decodeMethodSignature(
    const char *signature, QArgumentTypeArray &types)
{
    Q_ASSERT(signature != 0);
    const char *lparens = strchr(signature, '(');
    if (!lparens)
        return QByteArray();
    const char *rparens = strchr(lparens + 1, ')');
    if (!rparens || *(rparens+1))
        return QByteArray();
    int nameLength = lparens - signature;
    argumentTypesFromString(lparens + 1, rparens, types);
    return QByteArray::fromRawData(signature, nameLength);
}
```

 学而思网校技术团队


4、计算索引（包括基类）。

```
int signal_index = QMetaObjectPrivate::indexOfSignalRelative(&smeta, signalName,
    signalTypes.size(), signalTypes.constData());
```

 学而思网校技术团队

具体实现如下：

```
static inline int indexOfMethodRelative(const QMetaObject **baseObject, const QByteArray &name,
    int argc, const QArgumentType *types)
{
    for (const QMetaObject *m = *baseObject; m; m = m->d.superdata) {
        int i = (MethodType == MethodSignal)? (priv(m->d.data)->signalCount - 1) :
            (priv(m->d.data)->methodCount - 1);
        for (; i >= end; --i) {
            int handle = priv(m->d.data)->methodData + 5*i;
            if (methodMatch(m, handle, name, argc, types)) {
                *baseObject = m;
                return i;
            }
        }
    }
    return -1;
}
```

 学而思网校技术团队

其中 `int handle = priv(m->d.data)->methodData + 5*i`; 我们可以分析, 其实就是 `14+5*i`, 那为什么是5呢? 因为:

```
// signals: name, argc, parameters, tag, flags
```

```
1,  0,  24,  2, 0x06 /* Public */,
```


```
// slots: name, argc, parameters, tag, flags
```

```
3,  0,  25,  2, 0x08 /* Private */,
```

我们可以看到每一个signals或者slots都有5个整形表示。


5、对掩码进行检查。

```
signal_index = QMetaObjectPrivate::originalClone(smeta, signal_index);
```

 学而思网校技术团队

// MethodFlags是一个枚举类型, 我们可以看到 `MethodSignal = 0x04`,
`MethodSlot = 0x08`;

```
enum MethodFlags {
    ...
    MethodSignal = 0x04,
    MethodSlot = 0x08,
    ...
};
```

 学而思网校技术团队

```
// slots: name, argc, parameters, tag, flags
```

```
3,  0,  25,  2, 0x08 /* Private */,
```

6、判断链接类型, 默认是Qt::AutoConnection。

```
enum ConnectionType {
    AutoConnection,
    DirectConnection,
    QueuedConnection,
    BlockingQueuedConnection,
    UniqueConnection = 0x80
};
```

我们介绍一些连接类型:

1、**AutoConnection**: 自动连接: 默认的方式, 信号发出的线程和槽的对象在一个线程的时候相当于: `DirectConnection`, 如果是在不同线程, 则相当于 `QueuedConnection`。

2、**DirectConnection**: 直接连接: 相当于直接调用槽函数, 但是当信号发出的线程和槽的对象不再一个线程的时候, 则槽函数是在发出的信号中执行的。

3、**QueuedConnection**: 队列连接: 内部通过 `postEvent` 实现的。不是实时调用的, 槽函数永远在槽函数对象所在的线程中执行。如果信号参数是引用类型, 则会另外复

制一份的。线程安全的。


4、**BlockingQueuedConnection**：阻塞连接：此连接方式只能用于信号发出的线程和槽函数的对象不再一个线程中才能用，通过信号量+postEvent实现的，不是实时调用的，槽函数永远在槽函数对象所在的线程中执行，但是发出信号后，当前线程会阻塞，等待槽函数执行完毕后才继续执行。

5、**UniqueConnection**：防止重复连接。如果当前信号和槽已经连接过了，就不再连接了。

最后到了信号和槽关联核心的地方了：

首先，我们先得了解以下数据结构：


```
class Q_CORE_EXPORT QObjectData {
public:
    QObject *q_ptr;                // 对象this指针
    ...
    QDynamicMetaObjectData *metaObject;
    QMetaObject *dynamicMetaObject() const;
};
```

 学而思网校技术团队

```

class Q_CORE_EXPORT QObjectPrivate : public QObjectData
{
    Q_DECLARE_PUBLIC(QObject)
public:
    typedef void (*StaticMetaCallFunction)(QObject *, QMetaObject::Call, int, void
**);
    struct Connection {
        QObject *sender;
        QObject *receiver;
        union {
            StaticMetaCallFunction callFunction; // 回调函数指针
        };
        // The next pointer for the singly-linked ConnectionList
        Connection *nextConnectionList; // 单链表
        // senders linked list
        Connection *next;
        Connection **prev;
        Connection() : nextConnectionList(0), ref_(2), ownArgumentTypes(true) {}
    };
    // ConnectionList is a singly-linked list
    struct ConnectionList {
        ConnectionList() : first(0), last(0) {}
        Connection *first;
        Connection *last;
    };
    struct Sender {
        QObject *sender;
        int signal;
        int ref;
    };
    QObjectPrivate(int version = QObjectPrivateVersion);
    void addConnection(int signal, Connection *c);
    void cleanConnectionLists();
    static QObjectPrivate *get(QObject *o) {
        return o->d_func();
    }
    static QMetaObject::Connection connect(const QObject *sender, int
signal_index,
                                           QtPrivate::QSlotObjectBase *slotObj, Qt::ConnectionType
type);
    static bool disconnect(const QObject *sender, int signal_index, void **slot);
public:
    QObjectConnectionListVector *connectionLists;
    Connection *senders; // linked list of connections connected to this
object
    Sender *currentSender; // object currently activating the object
    mutable quint32 connectedSignals[2];
};


```

 学而思网校技术团队


```

class Q_CORE_EXPORT QObject
{
    Q_OBJECT
public:
    Q_INVOKABLE explicit QObject(QObject *parent=Q_NULLPTR);
    virtual ~QObject();
    static QMetaObject::Connection connect(const QObject*sender, const char
*signal,
                                         const QObject*receiver, const char
*member,
                                         ConnectionType = AutoConnection);
    static bool disconnect(const QObject*sender, const char *signal,
                           const QObject*receiver, const char *member);
    ...
protected:
    ObjectData* d_ptr;
};

```


 学而思网校技术团队

上面的这三个数据结构很重要，QObject是我们最熟悉的基类，QObjectPrivate是它的私有类，进行具体实现，QObjectPrivate继承自QObjectData，在QObject里面以组合的形式也进行P指针和D指针的方式进行访问的。在信号和槽关联过程中，数据结构Connection是很重要的数据结构，下面的这个结构是ConnectionList的一个Vector：

```

class QObjectConnectionListVector : public QVector<QObjectPrivate::ConnectionList>
{
public:
    ...
    QObjectPrivate::ConnectionList allsignals;
    QObjectConnectionListVector()
        : QVector<QObjectPrivate::ConnectionList>(), orphaned(false),
dirty(false), inUse(0){ }
    QObjectPrivate::ConnectionList &operator[](int at) {
        if (at < 0) return allsignals;
        return QVector<QObjectPrivate::ConnectionList>::operator[](at);
    }
};

```

 学而思网校技术团队

有了上面的数据结构，我们就可以分析下面的链接过程了，我们看到下面的先是调用的QMetaObjectPrivate的connect，之后又用QMetaObject::Connection进行了指针包装：

```

QMetaObject::Connection handle = QMetaObject::Connection(QMetaObjectPrivate::connect(
    sender, signal_index, smeta, receiver, method_index_relative, rmeta ,type,
    types));

```

 学而思网校技术团队

```

andle = QMetaObject::Connection(QMetaObjectPrivate::connect(

```

```

QObjectPrivate::Connection *QMetaObjectPrivate::connect(const QObject *sender, int signal_index,
const QMetaObject *smeta,
                                const QObject *receiver, int method_index, const
QMetaObject *rmeta, int type, int *types)
{
    // 获取发送者和接收者
    QObject *s = const_cast<QObject*>(sender);
    QObject *r = const_cast<QObject*>(receiver);
    // 获取槽的偏移
    int method_offset = rmeta ? rmeta->methodOffset() : 0;
    // 注册回调函数，这个就是我们最后调用通信的回调函数
    QObjectPrivate::StaticMetaCallFunction callFunction = rmeta ? rmeta-
>d.static_metacall : 0;
    // 构造一个Connection结构:
    QScopedPointer<QObjectPrivate::Connection> c(new QObjectPrivate::Connection);
    c->sender = s;
    c->signal_index = signal_index;
    c->receiver = r;
    c->method_relative = method_index;
    c->method_offset = method_offset;
    c->nextConnectionList = 0;
    c->callFunction = callFunction; // 回调函数
    ...
    QObjectPrivate::get(s)->addConnection(signal_index, c.data());
    return c.take();
}

```

学而思网校技术团队

QObjectPrivate::get(s) 方法其实就是获取了一个QObject里面的QObjectPrivate实例，之后调用addConnection方法添加到链表中：

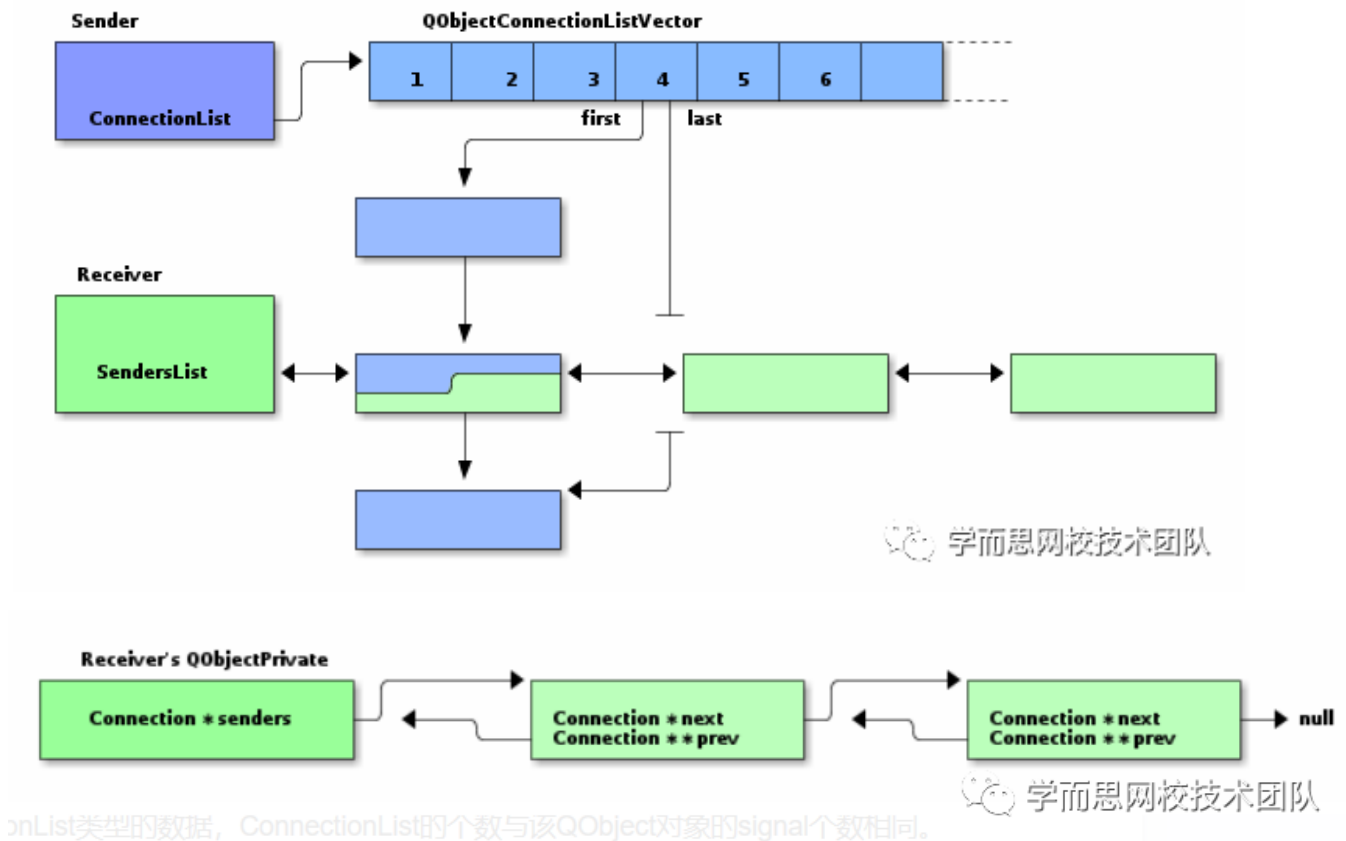
```

void QObjectPrivate::addConnection(int signal, Connection *c)
{
    if (!connectionLists)
        connectionLists = new QObjectConnectionListVector();
    if (signal >= connectionLists->count())
        connectionLists->resize(signal + 1);
    ConnectionList &connectionList = (*connectionLists)[signal];
    if (connectionList.last) {
        connectionList.last->nextConnectionList = c;
    } else {
        connectionList.first = c;
    }
    connectionList.last = c;
    cleanConnectionLists(); // 清除脏数据和不用的数据
    c->prev = &(QObjectPrivate::get(c->receiver)->senders);
    c->next = *c->prev;
    *c->prev = c;
    if (c->next)
        c->next->prev = &c->next;
    if (signal < 0) {
        connectedSignals[0] = connectedSignals[1] = ~0;
    } else if (signal < (int)sizeof(connectedSignals) * 8) {
        connectedSignals[signal >> 5] |= (1 << (signal & 0x1f));
    }
}

```

学而思网校技术团队

结构如下：



分析：

1、每个QObject对象都有一个QObjectConnectionListVector结构，这是一个Vector容器，它里面的基本单元都是ConnectionList类型的数据，ConnectionList的个数与该QObject对象的signal个数相同。每个ConnectionList对应一个信号，它记录了连接到这个信号上的所有连接。前面已经看到ConnectionList的定义中有两个重要成员：first和last，他们都是Connection类型的指针，分别指向连接到这个信号上的第一个和最后一个连接。所有连接到这个信号上的连接以单向链表的方式组织了起来，Connection结构体中的nextConnectionList成员就是用来指向这个链表中的下一个连接的。

2、同时，每个QObject对象还有一个senders成员，senders是一个Connection类型的指针，senders本身也是一个链表的头结点，这个链表中的所有结点都是连接到这个QObject对象上的某个槽的连接。不过这个链表跟上一段提到的链表可不是同一个，虽然他们可能有一些共同结点。


3、每一个Connection对象都同时处于两个链表当中。其中一个是以Connection的nextConnectionList成员组织起来的单向链表，这个单项链表中每个结点的共同点是，他们都依赖于同一个QObject对象的同一个信号，这个链表的头结点就是这个信号对应的ConnectionList结构中的first；另一个链表是以Connection的next和prev成员组织起来的双向链表，这个双向链表中每个结点的共同点是，他们的槽都在同一个QObject对象上，这个链表的头结点就是这个QObject对象的sender。这两个链表会有交叉（共同结点），但他们有不同的链接指针，所以不是同一个链表。

4、在Connect的时候，就是先new一个Connection对象出来，设置好这个连接的信息后，将它分别添加到上面提到的两个链表中；disconnect的时候，就从从这两个链表中将它移除，然后delete掉。而当一个QObject对象被销毁的时候，它的sender指针指向的那个双向链表中的所有连接都会被逐个移除！

第三步、发送信号到接受信号：


1、我们点击上面的button后，然后调用到onDestory槽里面，这是我们写的信号触发的地方：

```
void Test::onDestory()
{
    emit clean();
}
```



2、接下来就进入了moc_MainWindow.cpp里面的代码，调用了QMetaObject的静态方法activate：

```
// SIGNAL 0
void Test::clean()
{
    QMetaObject::activate(this, &staticMetaObject, 0, Q_NULLPTR);
}
```



// 然后进入真正的QMetaObject::activate


```

void QMetaObject::activate(QObject *sender, int signalOffset, int local_signal_index, void **argv)
{
    int signal_index = signalOffset + local_signal_index;

    void *empty_argv[] = { 0 };
    // 上锁 (可能是多线程访问)
    QMutexLocker locker(signalSlotLock(sender));
    // 跟进信号索引获取链表
    QObjectConnectionListVector*connectionLists = sender->d_func()-
>connectionLists;
    const QObjectPrivate::ConnectionList *list;
    if (signal_index < connectionLists->count())
        list = &connectionLists->at(signal_index);
    else
        list = &connectionLists->allsignals;
    do {
        // 循环遍历, 执行所有信号关联的槽
        QObjectPrivate::Connection *c = list->first;
        if (!c) continue;
        QObjectPrivate::Connection *last = list->last;
        do {
            if (!c->receiver)
                continue;
            QObject * const receiver = c->receiver;
            // 如果是QueuedConnection链接方式则不是立即进行回调, 而是放在事件队列中, 等有了
            事件才进行调用。
            // put into the event queue
            if ((c->connectionType == Qt::AutoConnection && !
receiverInSameThread)|| (c->connectionType == Qt::QueuedConnection)) {
                queued_activate(sender, signal_index, c, argv ? argv : empty_argv,
locker);
                continue;
                // 对于阻塞链接方式, 是通过信号里来进行同步的
            } else if (c->connectionType == Qt::BlockingQueuedConnection) {
                locker.unlock();
                QSemaphore semaphore;
                QMetaCallEvent *ev = c->isSlotObject ?
                    new QMetaCallEvent(c->slotObj, sender, signal_index, 0, 0,
argv ? argv : empty_argv, &semaphore) :
                    new QMetaCallEvent(c->method_offset, c->method_relative, c-
>callFunction, sender, signal_index, 0, 0, argv ?
                    argv : empty_argv, &semaphore);
                QCoreApplication::postEvent(receiver, ev);
                semaphore.acquire();
                locker.relock();
                continue;
            }
            // Autoconntion, callFunction就是我们前面注册的回调函数
            if (c->callFunction && c->method_offset <= receiver->metaObject()-
>methodOffset()) {
                callFunction(receiver, QMetaObject::InvokeMetaMethod,
method_relative, argv ? argv : empty_argv);
                locker.relock();
            }
        } while (c != last && (c = c->nextConnectionList) != 0);
    } while (list != &connectionLists->allsignals && ((list = &connectionLists-
>allsignals), true));
}

```


学而思网校技术团队

我们的例子是Autoconntion模式, 所以就会执行下面的代码进行回调:

```
callFunction(receiver, QMetaObject::InvokeMetaMethod, method_relative, argv ? argv : empty_argv);
```


我们终于看到了, 函数进行了回调到moc_MainWindow.cpp里面, 然后调用对应的槽onClean;

```
void MainWindow::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id, void **_a)
{
    if (_c == QMetaObject::InvokeMetaMethod) {
        MainWindow *_t = static_cast<MainWindow *>(_o);
        Q_UNUSED(_t)
        switch (_id) {
            case 0: _t->onClean(); break;
            default: ;
        }
    }
    Q_UNUSED(_a);
}
```

 学而思网校技术团队


最终调用到这里后，打印输出："MainWindow::onClean"

```
void MainWindow::onClean()
{
    qDebug() << "MainWindow::onClean";
}
```

 学而思网校技术团队

最后就是调用完后，会回到onDestory这里：

```
void Test::onDestory()
{
    emit clean();
    qDebug() << "Test::onDestory over.";
}
```

 学而思网校技术团队

注意：如果我们在onClean中进行了对m_testWidget对象的释放操作（delete m_testWidget），再到onDestory()中 emit clean(); 后面进行访问成员，那么一定崩溃，所以要注意。

参考文献：

- 1、<https://woboq.com/blog/how-qt-signals-slots-work.html>
- 2、Qt5.7源码
- 3、自己用C++实现的信号和槽demo：<http://note.youdao.com/noteshare?id=903c8feeda3395f318b872d1a8abab09>

▼ 往期精彩回顾 ▼

[Golang内存分配](#)

[利用DPDK优化容器网络](#)

[剖析页面耗时思路解析](#)

[从Google V8引擎剖析Promise实现](#)



有趣的知识在等你

扫码关注 更多精彩

喜欢此内容的人还喜欢

基于未来云容器实现业务的平滑升级

学而思网校技术团队

