Qt原理 — 信号槽

跟踪源码

1. 程序流程不走的分支，不分析
2. 删掉部分用不到的成员

分析小例子

代码

```
/*****************************************************
* file: mytest.h
*****************************************************/
#ifndef MYTEST_H
#define MYTEST_H

#include <QObject>

class MyTest : public QObject
{
    Q_OBJECT
public:
    explicit MyTest(QObject *parent = nullptr);

signals:
    void signal_1_test();
    void signal_2_test(int value);
    int signal_3_test(int value);
    void signal_4_test(double value1, int value2);

public slots:
    void slot_1_test();
    void slot_2_test(int value);
    int slot_3_test(int value);
    void slot_4_test(double value1, int value2);

public:
    void test();
};

#endif // MYTEST_H

/*****************************************************
* file: mytest.cpp
*****************************************************/
#include "mytest.h"
#include <iostream>

using namespace std;
```

```cpp
MyTest::MyTest(QObject *parent) : QObject(parent)
{

}

void MyTest::slot_1_test()
{
    cout << "void slot_1_test()" << endl;
}

void MyTest::slot_2_test(int value)
{
    cout << "void slot_2_test(int value)" << endl;
}

int MyTest::slot_3_test(int value)
{
    cout << "int slot_3_test(int value)" << endl;
}

void MyTest::slot_4_test(double value1, int value2)
{
    cout << "void slot_4_test(double value1, int value2)" << endl;
}

void MyTest::test()
{
    emit signal_1_test();
    emit signal_2_test(0);
    emit signal_3_test(0);
    emit signal_4_test(3.14, 0);
}


/*************************************************
* file: main.cpp
*************************************************/
#include <QCoreApplication>
#include "mytest.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    MyTest t;
    QObject::connect(&t, SIGNAL(signal_1_test()), &t, SLOT(slot_1_test()));
    QObject::connect(&t, SIGNAL(signal_2_test(int)), &t,
SLOT(slot_2_test(int)));
    QObject::connect(&t, SIGNAL(signal_3_test(int)), &t,
SLOT(slot_3_test(int)));
    QObject::connect(&t, SIGNAL(signal_4_test(double,int)), &t,
SLOT(slot_4_test(double,int)));

    t.test();

    return a.exec();
}
```

项目编译过后，会生成`moc_mytest.cpp`文件

moc_mytest.cpp文件

qt_meta_stringdata_MyTest和qt_meta_data_MyTest

在此文件中，有两个静态成员`qt_meta_stringdata_MyTest`和`qt_meta_data_MyTest`

- `qt_meta_stringdata_MyTest`

```
static const qt_meta_stringdata_MyTest_t qt_meta_stringdata_MyTest = {
    {
QT_MOC_LITERAL(0, 0, 6), // "MyTest"
QT_MOC_LITERAL(1, 7, 13), // "signal_1_test"
QT_MOC_LITERAL(2, 21, 0), // ""
QT_MOC_LITERAL(3, 22, 13), // "signal_2_test"
QT_MOC_LITERAL(4, 36, 5), // "value"
QT_MOC_LITERAL(5, 42, 13), // "signal_3_test"
QT_MOC_LITERAL(6, 56, 13), // "signal_4_test"
QT_MOC_LITERAL(7, 70, 6), // "value1"
QT_MOC_LITERAL(8, 77, 6), // "value2"
QT_MOC_LITERAL(9, 84, 11), // "slot_1_test"
QT_MOC_LITERAL(10, 96, 11), // "slot_2_test"
QT_MOC_LITERAL(11, 108, 11), // "slot_3_test"
QT_MOC_LITERAL(12, 120, 11) // "slot_4_test"

    },
    "MyTest\0signal_1_test\0\0signal_2_test\0"
    "value\0signal_3_test\0signal_4_test\0"
    "value1\0value2\0slot_1_test\0slot_2_test\0"
    "slot_3_test\0slot_4_test"
};
```

此成员是`qt_meta_stringdata_MyTest_t`结构体

```
struct qt_meta_stringdata_MyTest_t {
    QByteArrayData data[13];    // typedef QArrayData QByteArrayData;
    char stringdata0[132];      // 存放了字符串的字符数组
                               //这些数组包含了类 名、信号函数和槽函数的名字和参数的名字
};

struct Q_CORE_EXPORT QArrayData
{
    QtPrivate::RefCount ref;
    int size;   // 字符串的长度
    uint alloc : 31;
    uint capacityReserved : 1;

    qptrdiff offset; // in bytes from beginning of header
                     // 字符串到本结构体对象首地址的偏移

    // method member.....
}
```

`QT_MOC_LITERAL`宏的参数解读：

此宏用来填充`QByteArrayData`结构体

1. 第一个参数是编号
2. 第二个参数是字符数组`stringdata0`的下标索引
3. 第三个参数是字符串所包含的字符的个数

例如：

```
QT_MOC_LITERAL(9, 84, 11), // "slot_1_test"
/*
编号为9
位于数组索引第84位（索引从0开始）
长度为11
*/
```

- `qt_meta_data_MyTest`

```
static const uint qt_meta_data_MyTest[] = {

 // content:
       7,       // revision
       0,       // classname
       0,    0, // classinfo
       8,   14, // methods
       0,    0, // properties
       0,    0, // enums/sets
       0,    0, // constructors
       0,       // flags
       4,       // signalCount

 // signals: name, argc, parameters, tag, flags
       1,    0,   54,    2, 0x06 /* Public */,
       3,    1,   55,    2, 0x06 /* Public */,
       5,    1,   58,    2, 0x06 /* Public */,
       6,    2,   61,    2, 0x06 /* Public */,

 // slots: name, argc, parameters, tag, flags
       9,    0,   66,    2, 0x0a /* Public */,
      10,    1,   67,    2, 0x0a /* Public */,
      11,    1,   70,    2, 0x0a /* Public */,
      12,    2,   73,    2, 0x0a /* Public */,

 // signals: parameters
    QMetaType::Void,
    QMetaType::Void, QMetaType::Int,    4,
    QMetaType::Int, QMetaType::Int,    4,
    QMetaType::Void, QMetaType::Double, QMetaType::Int,    7,    8,

 // slots: parameters
    QMetaType::Void,
    QMetaType::Void, QMetaType::Int,    4,
    QMetaType::Int, QMetaType::Int,    4,
    QMetaType::Void, QMetaType::Double, QMetaType::Int,    7,    8,
```

```
        0        // eod
};
```

此成员是`uint`数组，此数组分为两个部分，第一个部分是**// content:**注释部分，第二部分是除去第一部分的其余部分，描述了信号槽函数的相关信息。

此例中，信号有四个，槽也有四个，信号和槽的参数信息也分别有四个

```
 // signals: name, argc, parameters, tag, flags
       1,    0,   54,    2, 0x06 /* Public */,

/*
1:  qt_meta_stringdata_MyTest结构中编号为1，信号是signal_1_test
0:  参数个数是0
54: 位于qt_meta_data_MyTest头部的偏移，即 QMetaType::Void，从这里可看出返回值为void，没有参数，如果后面有数字，则是参数的名字，即 qt_meta_stringdata_MyTest_t结构中数组的索引

例如:
QMetaType::Void, QMetaType::Int,    4,
QT_MOC_LITERAL(4, 36, 5), // "value"
*/
```

信号

虽然信号是一个没有函数实现的函数声明，但是在此文件中，一并生成了信号的函数实现

```
// SIGNAL 0
void MyTest::signal_1_test()
{
    QMetaObject::activate(this, &staticMetaObject, 0, nullptr);
}

// SIGNAL 1
void MyTest::signal_2_test(int _t1)
{
    void *_a[] = { nullptr, const_cast<void*>(reinterpret_cast<const void*>
(&_t1)) };
    QMetaObject::activate(this, &staticMetaObject, 1, _a);
}

// SIGNAL 2
int MyTest::signal_3_test(int _t1)
{
    int _t0{};
    void *_a[] = { const_cast<void*>(reinterpret_cast<const void*>(&_t0)),
const_cast<void*>(reinterpret_cast<const void*>(&_t1)) };
    QMetaObject::activate(this, &staticMetaObject, 2, _a);
    return _t0;
}

// SIGNAL 3
void MyTest::signal_4_test(double _t1, int _t2)
{
```

```
    void *_a[] = { nullptr, const_cast<void*>(reinterpret_cast<const void*>
(&_t1)), const_cast<void*>(reinterpret_cast<const void*>(&_t2)) };
    QMetaObject::activate(this, &staticMetaObject, 3, _a);
}
```

元对象

对象的信息保存在成员`staticMetaObject`中，通过`metaObject`函数获取

```
const QMetaObject MyTest::staticMetaObject = {
    { &QObject::staticMetaObject, qt_meta_stringdata_MyTest.data,
      qt_meta_data_MyTest,  qt_static_metacall, nullptr, nullptr}
};


const QMetaObject *MyTest::metaObject() const
{
    return QObject::d_ptr->metaObject ? QObject::d_ptr->dynamicMetaObject() :
&staticMetaObject;
}
```

`QMetaObject`结构如下

```
struct Q_CORE_EXPORT QMetaObject
{
    // ...

    enum Call {
        InvokeMetaMethod,
        ReadProperty,
        WriteProperty,
        ResetProperty,
        QueryPropertyDesignable,
        QueryPropertyScriptable,
        QueryPropertyStored,
        QueryPropertyEditable,
        QueryPropertyUser,
        CreateInstance,
        IndexOfMethod,
        RegisterPropertyMetaType,
        RegisterMethodArgumentMetaType
    };

    // ...

    struct { // private data
        const QMetaObject *superdata;       // &QObject::staticMetaObject
        const QByteArrayData *stringdata;   // qt_meta_stringdata_MyTest_t的data
数组
        const uint *data;   // qt_meta_data_MyTest数组
        typedef void (*StaticMetacallFunction)(QObject *, QMetaObject::Call,
int, void **);
        StaticMetacallFunction static_metacall; // qt_static_metacall函数地址
```

```
        const QMetaObject * const *relatedMetaObjects;
        void *extradata; //reserved for future use
    } d;
};
```

`qt_static_metacall`函数，信号函数会在这里被调用

```
void MyTest::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id, void
**_a)
{
    if (_c == QMetaObject::InvokeMetaMethod) {
        MyTest *_t = static_cast<MyTest *>(_o);
        Q_UNUSED(_t)
            // 通过id调用信号
        switch (_id) {
        case 0: _t->signal_1_test(); break;
        case 1: _t->signal_2_test((*reinterpret_cast< int(*)>(_a[1]))); break;
        case 2: { int _r = _t->signal_3_test((*reinterpret_cast< int(*)>
(_a[1])));
            if (_a[0]) *reinterpret_cast< int*>(_a[0]) = std::move(_r); }
 break;
        case 3: _t->signal_4_test((*reinterpret_cast< double(*)>(_a[1])),
(*reinterpret_cast< int(*)>(_a[2]))); break;
        case 4: _t->slot_1_test(); break;
        case 5: _t->slot_2_test((*reinterpret_cast< int(*)>(_a[1]))); break;
        case 6: { int _r = _t->slot_3_test((*reinterpret_cast< int(*)>(_a[1])));
            if (_a[0]) *reinterpret_cast< int*>(_a[0]) = std::move(_r); }
 break;
        case 7: _t->slot_4_test((*reinterpret_cast< double(*)>(_a[1])),
(*reinterpret_cast< int(*)>(_a[2]))); break;
        default: ;
        }
    } else if (_c == QMetaObject::IndexOfMethod) {
        int *result = reinterpret_cast<int *>(_a[0]);
        {
            typedef void (MyTest::*_t)();
            if (*reinterpret_cast<_t *>(_a[1]) == static_cast<_t>
(&MyTest::signal_1_test)) {
                *result = 0;
                return;
            }
        }
        {
            typedef void (MyTest::*_t)(int );
            if (*reinterpret_cast<_t *>(_a[1]) == static_cast<_t>
(&MyTest::signal_2_test)) {
                *result = 1;
                return;
            }
        }
        {
            typedef int (MyTest::*_t)(int );
            if (*reinterpret_cast<_t *>(_a[1]) == static_cast<_t>
(&MyTest::signal_3_test)) {
                *result = 2;
```

```
                return;
            }
        }
        {
            typedef void (MyTest::*_t)(double , int );
            if (*reinterpret_cast<_t *>(_a[1]) == static_cast<_t>
(&MyTest::signal_4_test)) {
                *result = 3;
                return;
            }
        }
    }
}
```