Qt 信号槽的实现

原文地址：http://woboq.com/blog/how-qt-signals-slots-work.html

Qt 因其信号槽机制闻名遐迩。但是信号槽是怎样工作的？本文将解释 QObject 和 QMetaObject 的内部实现，以及探索如何在这种机制下实现信号槽。

本文将展示 Qt5 的部分代码，不过有时会为格式化以及简洁性而有所修改。

信号和槽

首先，我们通过官方示例回忆下信号槽是如何工作的。

我们的头文件是这样的：

```
1
2
3
4
5
6
7
8
9
10
11
class Counter : public QObject
{
    Q_OBJECT
    int m_value;
public:
    int value() const { return m_value; }
public slots:
    void setValue(int value);
signals:
    void valueChanged(int newValue);
};
1
2
3
4
5
```

```
6
7
void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}
```

之后，我们可以这样使用 Counter 对象：

```
1
2
3
4
5
Counter a, b;
QObject::connect(&a, SIGNAL(valueChanged(int)),
                 &b, SLOT(setValue(int)));
a.setValue(12);  // a.value() == 12, b.value() == 12
```

这是最初的语法，从 1992 年 Qt 发明依赖就没有变化。

但是，即使从一开始最基本的 API 没有改变，其底层实现被修改过好多次。在底层添加了许多新的特性，也有很多内容发生了变化。不过，即便如此，也并没有引入任何魔法，本文即将向您展示这些是如何发生的。

MOC —— 元对象编译器

Qt 的信号槽和属性系统基于在运行时进行内省的能力。内省意味着，我们可以列出对象的方法和属性列表，并且能够获取有关它们的所有信息，例如其参数类型。没有这种内省能力，QtScript 和 QML 就很难实现。

C++ 原生并没有提供内省，所以 Qt 提供了一个工具来支持它。这个工具就是 MOC。这是一个代码生成器（但它**不是**某些人所称的预编译器）。

它处理头文件，生成额外的 C++ 文件，这些文件将同程序剩下的部分一起编译。这些生成的 C++ 文件包含了内省所需要的所有信息。

由于这个额外的代码生成器，Qt 有时会被某些语言纯正癖者批评。我们可以通过 Qt 文档，看看如何回应这种批评。代码生成器并没有任何问题，MOC 提供了极大的帮助。

带有魔法的宏

你能指出哪些*关键字*不是纯 C++ 的关键字吗？signals、slots、Q_OBJECT、emit、SIGNAL 和 SLOT。这些都是 Qt 对 C++ 的扩展。它们事实上就是简单的宏，在 qobjectdefs.h 中定义：

```
#define signals public#define slots /* nothing */
```

是的，信号和槽就是普通的函数：编译器就把它们同其它函数一样对待。这些宏还有另外一个目的：MOC 能够看到它们。

在 Qt4 及之前的版本中，signals 被展开成 protected。Qt5 则变成 public，用以支持新的语法。

```
#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 **); \
    QT_TR_FUNCTIONS /* translations helper */ \
private: \
    Q_DECL_HIDDEN static void qt_static_metacall(QObject *, QMetaObject::Call, int, void **);
```

Q_OBJECT 定义了一系列函数和一个静态的 QMetaObject 对象。这些函数由 MOC 在生成的文件中实现。

```
#define emit /* nothing */
```

emit 是一个空的宏。甚至 MOC 也不会处理它。换句话说，emit 其实是可选的，没有什么含义（除了提醒开发者）。

```
8
9
Q_CORE_EXPORT const char *qFlagLocation(const char *method);
#ifndef QT_NO_DEBUG# define QLOCATION "\0" __FILE__ ":" QTOSTRING(__LINE__)# define
SLOT(a)     qFlagLocation("1"#a QLOCATION)# define SIGNAL(a)   qFlagLocation("2"#a
QLOCATION)#else# define SLOT(a)     "1"#a# define SIGNAL(a)   "2"#a#endif
```

这些宏仅由预处理器使用，将参数转换成字符串，并且在之前添加一个代码。

在调试模式下，我们还会将这些字符串追加上所在文件的位置作为信号无法正常连接的警告信息。这是 Qt 4.5 以兼容的形式引入的。为了知道哪些字符串具有行信息，我们使用了 qFlagLocation，这个函数将在一个表格中注册字符串地址等两项。

MOC 生成的代码

下面我们将浏览下 Qt5 中 moc 生成的部分代码。

```
const QMetaObject Counter::staticMetaObject = {
    { &QObject::staticMetaObject, qt_meta_stringdata_Counter.data,
      qt_meta_data_Counter,  qt_static_metacall, 0, 0}
};
const QMetaObject *Counter::metaObject() const
{
    return QObject::d_ptr->metaObject ? QObject::d_ptr->dynamicMetaObject() : &staticMetaObject;
}
```

我们可以看到，这里有 Counter::metaObject() 和 Counter::staticMetaObject 的实现。它们在 Q_OBJECT 宏中被声明。QObject::d_ptr->metaObject 仅供动态元对象（QML 对象）使用，所以一般而言，虚函数 metaObject() 仅返回这个类的 staticMetaObject。

staticMetaObject 构建为只读数据。QMetaObject 在 qobjectdefs.h 定义：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
14
15
16
struct QMetaObject
{
    /* ... Skiped all the public functions ... */
    enum Call { InvokeMetaMethod, ReadProperty, WriteProperty, /*...*/ };
    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 **relatedMetaObjects;
        void *extradata; //reserved for future use
    } d;
};
```

间接定义 d 目的是表示，所有成员都应该是私有的。它们实际不是 private 的，目的是保持这个结构体是 POD，并且允许静态初始化。

QMetaObject 通过将父对象（本例中就是 QObject::staticMetaObject）作为 superdata 进行初始化。stringdata 和 data 由本文之后介绍的某些数据初始化。static_metacall 是一个指向 Counter::qt_static_metacall 的函数指针。

内省表

首先，我们分析下 QMetaObject 的整型数据：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
16
17
18
19
20
21
22
23
24
25
26
27
static const uint qt_meta_data_Counter[] = {
// 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,    1,   24,    2, 0x05,
// slots: name, argc, parameters, tag, flags
       4,    1,   27,    2, 0x0a,
// signals: parameters
    QMetaType::Void, QMetaType::Int,    3,
// slots: parameters
    QMetaType::Void, QMetaType::Int,    5,
       0        // eod
};
```

前 13 个 int 由头（header）组成。我们的表格有两列，第一列是总数，第二列是在这个数组中描述开始的索引。在这个例子中，我们有两个函数，函数描述的开始位置是索引 14。

函数描述由 5 个 int 组成。第一个是名字，这实际是其在字符串表（我们会在后面看到字符串表的细节）的索引位置。第二个整型是参数的个数，接下来是一个索引，表明在哪里可以找到这个参数的描述。现在我们先忽略 tag 和 flag 两个数据。对每一个函数，moc 还会保存每一个参数的返回类型、类型以及名字的索引。

豆子注：原文点到为止。我们再仔细看看这个数组各部分的含义。qt_meta_data_Counter 是一个 uint 数组，生成代码的时候已经为我们分为五个部分：第一部分 content，也就是内容，分为 9 行。第一行

revision，指明 moc 生成代码的版本号（Qt4 的 moc 生成的代码，该值是 6，也就是相当于 moc v6；Qt5 则是 7）。第二个 classname，也就是类名。这是一个索引，指向字符串表的某一个位置（本例中就是第 0 位）。后面便是类信息 classinfo、函数位置等的信息。总体来说，这个表就是一个索引表。

```
    QByteArrayData data[6];
    char stringdata[47];
};
#define QT_MOC_LITERAL(idx, ofs, len) \
    Q_STATIC_BYTE_ARRAY_DATA_HEADER_INITIALIZER_WITH_OFFSET(len, \
    offsetof(qt_meta_stringdata_Counter_t, stringdata) + ofs \
        - idx * sizeof(QByteArrayData) \
    )
static const qt_meta_stringdata_Counter_t qt_meta_stringdata_Counter = {
    {
QT_MOC_LITERAL(0, 0, 7),
QT_MOC_LITERAL(1, 8, 12),
QT_MOC_LITERAL(2, 21, 0),
QT_MOC_LITERAL(3, 22, 8),
QT_MOC_LITERAL(4, 31, 8),
QT_MOC_LITERAL(5, 40, 5)
    },
    "Counter\0valueChanged\0\0newValue\0setValue\0"
    "value\0"
};
#undef QT_MOC_LITERAL
```

这主要是一个 QByteArray 的静态数组。QT_MOC_LITERAL 宏创建了一个静态的 QByteArray，引用了后面字符串的特性索引位置。

信号

MOC 同时实现了信号。它们就是普通的函数，创建了一个指向参数的指针的数组，并将这些传给 QMetaObject::activate 函数。数组的第一个元素是返回值。在我们的例子中，这个值是 0，因为返回值是 void。传给 activate 的第三个参数是信号的索引（本例中是 0）。

```
1
2
3
4
5
6
// SIGNAL 0
void Counter::valueChanged(int _t1)
{
    void *_a[] = { 0, const_cast(reinterpret_cast(&_t1)) };
```

```
    QMetaObject::activate(this, &staticMetaObject, 0, _a);
}
1
2
3
4
5
6
7
8
9
void Counter::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id, void **_a)
{
    if (_c == QMetaObject::InvokeMetaMethod) {
        Counter *_t = static_cast(_o);
        switch (_id) {
        case 0: _t->valueChanged((*reinterpret_cast< int(*)>(_a[1]))); break;
        case 1: _t->setValue((*reinterpret_cast< int(*)>(_a[1]))); break;
        default: ;
        }
```

指向参数的指针数组同信号中的是一样的。_a[0] 永远不会被调用，因为在这里，所有的返回值都是 void。

有关索引的注意事项

在每一个 QMetaObject 中，槽、信号以及其它该对象可调用的函数都会分配一个从 0 开始的索引。它们是有顺序的，信号在第一位，然后是槽，最后是其它函数。这个索引在内部被称为**相对索引**。它们不包含父对象的索引位。

一般而言，我们并不想知道一个比特定类更一般的索引，但是却想包含在继承链中其它函数的索引。为了实现这一点，我们在相对索引的基础上添加一个偏移量，得到**绝对索引**。这是在公开 API 中使用的索引，由 QMetaObject::indexOf(Signal, Slot, Method) 这样的函数返回。

连接机制使用以信号为索引的向量。但是在向量中，所有的槽也会占有一定空间，通常在一个对象中，槽的数量要比信号多。所以从 Qt 4.6 开始，使用的是一种仅包含信号索引的新的内部实现。

在使用 Qt 开发时，你只需要知道函数的绝对索引。但是当你浏览 Qt 的 QObject 源代码时，你必须注意这三者之间的区别。

连接是怎样工作的?

在开始连接是，Qt 所要做的第一件事是，找出所需要的信号和槽的索引。Qt 会去查找元对象的字符串表来找出相应的索引。

然后，创建一个 QObjectPrivate::Connection 对象，将其添加到内部的链表中。

每一个连接都需要保存什么信息呢？我们需要一种方法，能够快速访问给定信号索引的连接。因为允许多个槽连接到同一个信号，我们需要为每一个信号添加一个已连接的槽的列表。每一个连接都必须包含接收对象和槽的索引。我们也想实现，在接收对象销毁的时候，这些连接也能够被自动销毁。所以每一个接收对象都需要知道谁连接到它自己，以便能够清理连接。

这是在 qobject_p.h 中 QObjectPrivate::Connection 的定义：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
```

```
struct QObjectPrivate::Connection
{
    QObject *sender;
    QObject *receiver;
    union {
        StaticMetaCallFunction callFunction;
        QtPrivate::QSlotObjectBase *slotObj;
    };
    // 单独连接的 ConnectionList 的 next 指针
    Connection *nextConnectionList;
    // senders 链表
    Connection *next;
    Connection **prev;
    QAtomicPointer argumentTypes;
    QAtomicInt ref_;
    ushort method_offset;
    ushort method_relative;
    uint signal_index : 27; // 信号范围（参考 QObjectPrivate::signalIndex()）
    ushort connectionType : 3; // 0 == auto, 1 == direct, 2 == queued, 4 == blocking
    ushort isSlotObject : 1;
    ushort ownArgumentTypes : 1;
    Connection() : nextConnectionList(0), ref_(2), ownArgumentTypes(true) {
        // ref_ 赋值为 2，以便内部列表使用，同时供 QMetaObject::Connection 使用
    }
    ~Connection();
    int method() const { return method_offset + method_relative; }
    void ref() { ref_.ref(); }
    void deref() {
        if (!ref_.deref()) {
            Q_ASSERT(!receiver);
            delete this;
        }
    }
};
```

每一个对象都有一个连接向量：将每一个信号与一个 QObjectPrivate::Connection 的链表关联起来。

每一个对象还有一个该对象所连接到的连接的反向列表，以便以后自动删除。这是一个双向链表。

之所以使用链表，是因为它们能够快速地添加、删除对象。它们在 QObjectPrivate::Connection 中，由指向下一个/上一个节点的指针来实现。

注意，senderList 的 prev 指针是一个指针的指针。这是因为我们并不是真的指向上一个节点，而是指向上一个节点中的 next 指针。这个指针仅在连接销毁时使用，并且不能向后遍历。它允许不为第一个元素

添加特殊处理。

发出信号

当我们调用信号时，实际是调用 MOC 生成的代码，而这部分代码是调用了 QMetaObject::activate。

下面是在 qobject.cpp 相关的实现代码：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
```

35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76

```
77
78
79
80
void QMetaObject::activate(QObject *sender, const QMetaObject *m, int local_signal_index,
                           void **argv)
{
    activate(sender, QMetaObjectPrivate::signalOffset(m), local_signal_index, argv);
    /* 这里我们仅转发到下一个函数。我们传递的是元对象的信号偏移量，而不是 QMetaObject 本身。
     * 它被分割成两个函数，因为 QML 内部需要另外的调用。 */}
void QMetaObject::activate(QObject *sender, int signalOffset, int local_signal_index, void **argv)
{
    int signal_index = signalOffset + local_signal_index;
    /* 我们所做的第一件事，是快速检查一个 64 位的位蒙版 bit-mask。如果为 0，
     * 我们就知道没有连接到该信号的东西，可以迅速返回，     * 这意味着，发送一个没有与槽连接的信号是相当迅速的。*/
    if (!sender->d_func()->isSignalConnected(signal_index))
        return; // nothing connected to these signals, and no spy
    /* ... 跳过调试信息和 QML 调用，以及一些合理性检查 ... */
    /* 使用互斥锁，因为 connectionList 中的所有操作都是线程安全的 */
    QMutexLocker locker(signalSlotLock(sender));
    /* 获取该信号的 ConnectionList。此处做了一些简化。真实的代码还为列表添加了引用计数和一些合理性检查 */
    QObjectConnectionListVector *connectionLists = sender->d_func()->connectionLists;
    const QObjectPrivate::ConnectionList *list =
        &connectionLists->at(signal_index);
    QObjectPrivate::Connection *c = list->first;
    if (!c) continue;
    // 我们需要最后一次检查，确保在信号发出的过程中添加的信号不会在本次发出过程被触发。
    QObjectPrivate::Connection *last = list->last;
    /* 遍历槽 */
    do {
        if (!c->receiver)
            continue;
        QObject * const receiver = c->receiver;
        const bool receiverInSameThread = QThread::currentThreadId() == receiver->d_func()-
>threadData->threadId;
        // 确定该连接应该立即发出，还是放入事件队列
        if ((c->connectionType == Qt::AutoConnection && !receiverInSameThread)
            || (c->connectionType == Qt::QueuedConnection)) {
            /* 从根本上说，就是复制参数，发出事件 */
            queued_activate(sender, signal_index, c, argv);
```

```
            continue;
        } else if (c->connectionType == Qt::BlockingQueuedConnection) {
            /* ... 跳过 ... */
            continue;
        }
        /* 助手结构体，设置 sender()（并且在超出作用域之后重新设回 */
        QConnectionSenderSwitcher sw;
        if (receiverInSameThread)
            sw.switchSender(receiver, sender, signal_index);
        const QObjectPrivate::StaticMetaCallFunction callFunction = c->callFunction;
        const int method_relative = c->method_relative;
        if (c->isSlotObject) {
            /* ... 跳过 ...  Qt5 风格的指向函数指针的连接 */
        } else if (callFunction && c->method_offset metaObject()->methodOffset()) {
            /* 如果存在 callFunction（指向由 moc 生成的 qt_static_metacall 的指针，
             * 调用该函数。还需要检查已保存的 metodOffset 是否依旧可用          * （因为我们可能在析构函数中调用） */
            locker.unlock(); // 实际调用时不能持有锁
            callFunction(receiver, QMetaObject::InvokeMetaMethod, method_relative, argv);
            locker.relock();
        } else {
            /* 动态对象 */
            const int method = method_relative + c->method_offset;
            locker.unlock();
            metacall(receiver, QMetaObject::InvokeMetaMethod, method, argv);
            locker.relock();
        }
        // 检查该对象是否被槽析构
        if (connectionLists->orphaned) break;
    } while (c != last && (c = c->nextConnectionList) != 0);
}
```

结论

我们看到了连接是怎样工作的，信号和槽是如何触发的。不过我们还没有了解 Qt5 新语法的实现，我们会在之后章节中详细讨论。