信号与槽是观察者模式的一种实现，信号相当于一个能够被观察的事件，一个槽就是一个观察者。

信号与槽是多对多的关系，一个槽可以监听多个信号，一个信号也可以连接多个槽，甚至信号与信号也可以进行连接。

注意：如果一个信号连接了多个槽，当信号发射时，相应的槽函数的执行顺序是随机的。

Qt中的信号与槽原理分析：

参考：<https://blog.51cto.com/9291927/2070398>

<https://www.cnblogs.com/swarmbees/p/10816139.html>

如果希望在类中使用信号与槽，则需要在类中声明Q\_OBJECT宏，如果包含了该宏，则会对应的生成了一个moc\_xxx.cpp的文件。

MOC（元对象编译器）：以FP/incude/object.h中的类Object为例，代码如下：

class Object : public QObject

{

Q\_OBJECT

protected:

QString name\_;

int age\_;

int score\_;

public:

explicit Object(QString name, QObject \*parent = 0);

void setAge(const int& age);

void setScore(const int& score);

signals:

void ageChanged(int age);

void scoreChanged(int score);

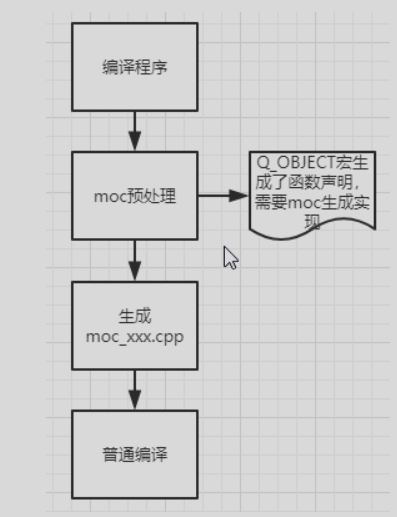
public slots:

void onAgeChanged(int age);

void onScoreChanged(int score);

};

在类Object中添加了Q\_OBJECT的宏，在编译程序时首先会调用MOC对程序进行预处理，生成moc\_object.cpp文件。



打开moc\_object.cpp会发现其包含一些变量和函数，其中比较重要的有以下几个：

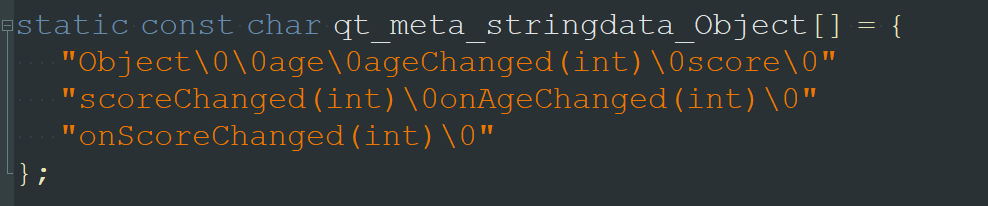
1. 元数据表

包括一个无符号整型数组构成的内省表和一个字符串表。

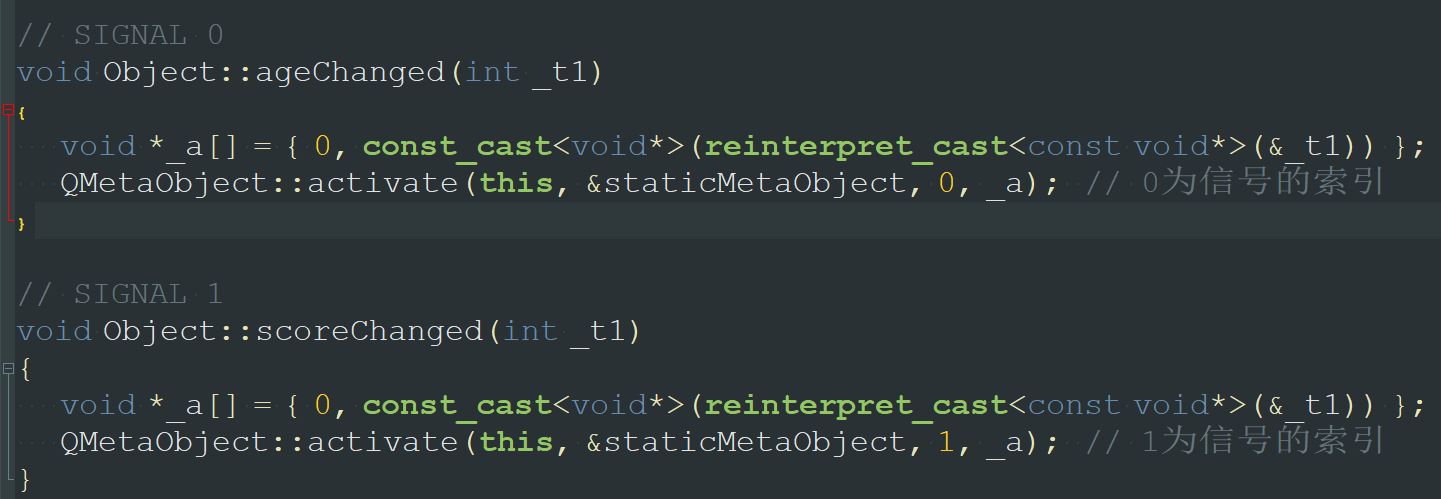
内省表：以qt\_meta\_data\_开头，本例中为qt\_meta\_data\_Object



字符串表：qt\_meta\_stringdata\_Object。

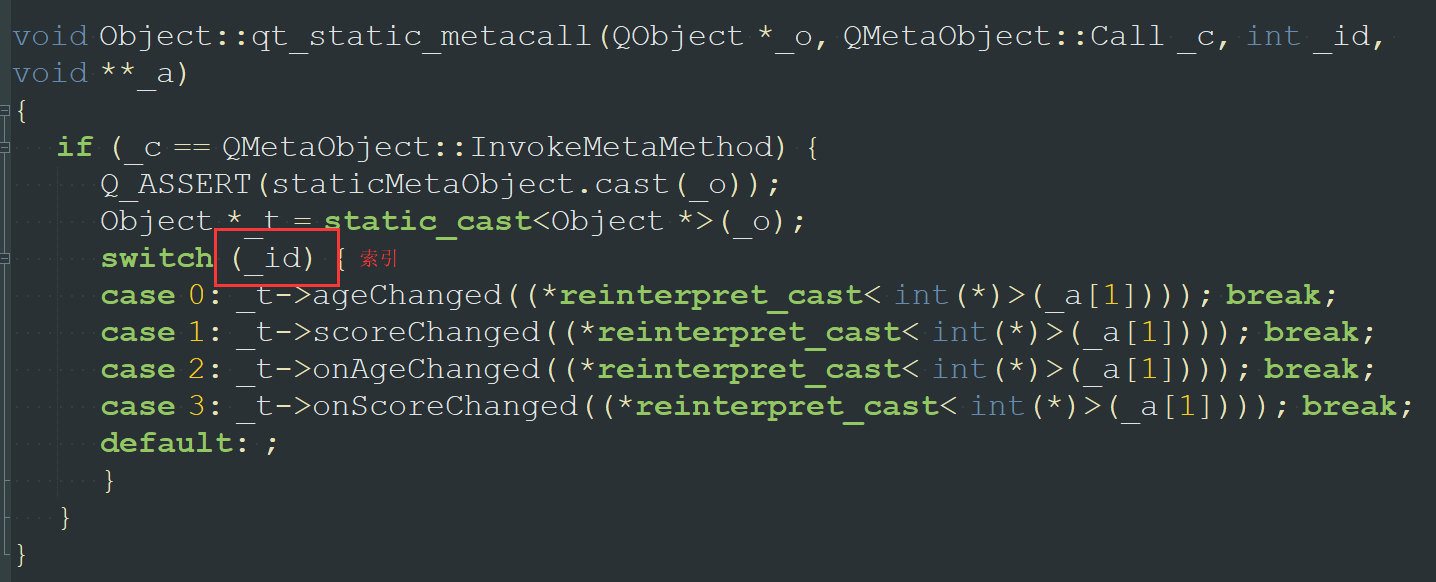


1. 实现信号



从上面的代码可以看出，信号其实也是一个函数。

1. 槽函数的调用：主要在qt\_static\_metacall函数中

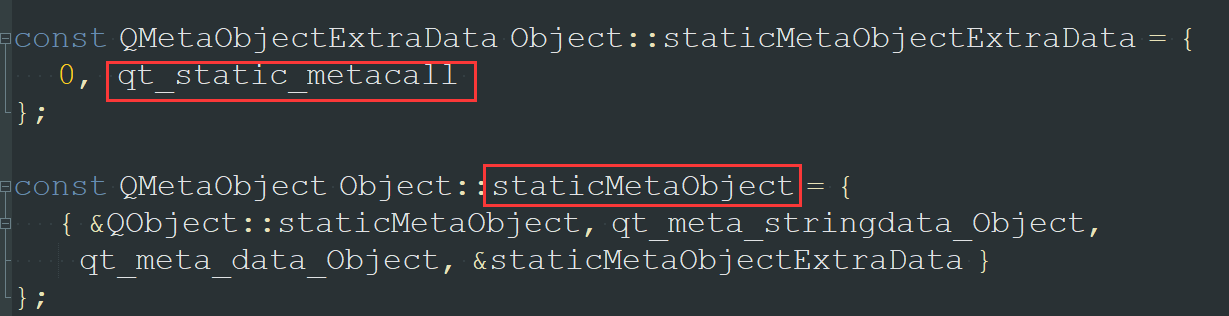


元对象中的索引：

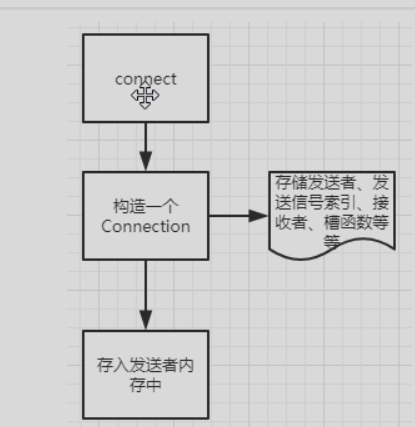
在每一个QMetaObject中，信号、槽以及其它可调用对象都会分配一个索引，从0开始计算。只针对信号和槽来说，一般是信号在前，槽函数在后，这个索引称为相对索引，不包括父对象的索引位置。

为了实现包含在继承链中的其它函数的索引，在相对索引的基础上添加一个偏移量，得到绝对索引。

最终得到一个类型是QMetaObject的元对象staticMetaObject，里面包含了上述相关的信息。

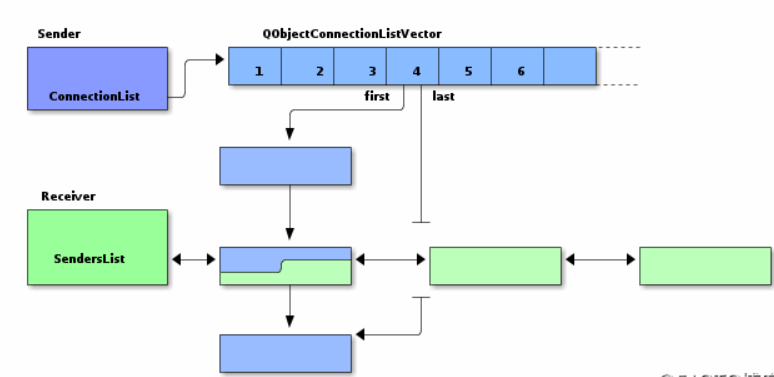


Qt中信号与槽的连接机制：通过QObject::connect函数来实现，过程如下所示。

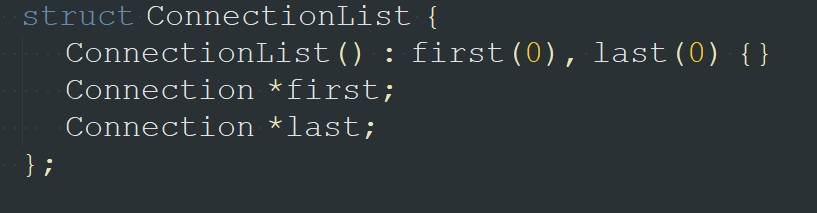


开始连接时，Qt首先通过QObject::connect函数找出所需要的信号和槽的索引，通过查找元对象的字符串表来找出相应的索引，最后调用QMetaObjectPrivate::connect将信号与槽进行连接。

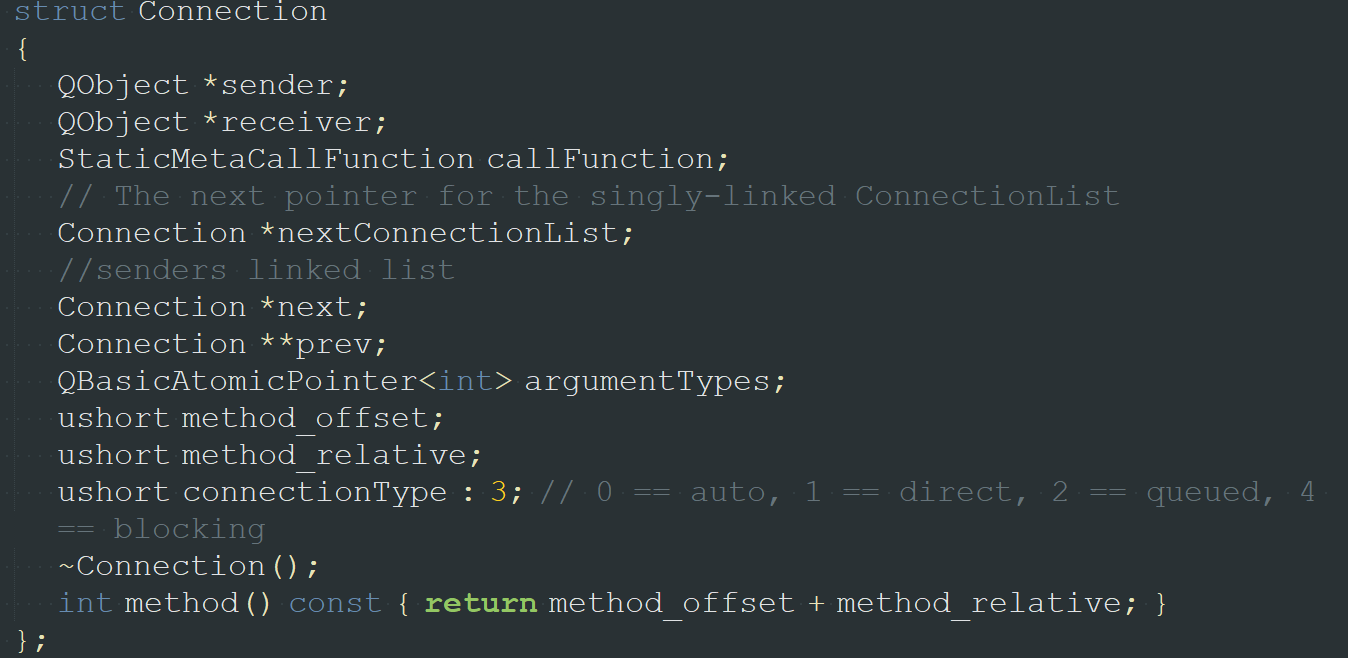
QObject及其派生类对象的元对象在创建时就有一个QObjectConnectionListVector连接链表容器，该容器的索引就是信号的索引，每一个信号都与一个QObjectPrivate::ConnectionList链表相关联，之所以每个信号都有链表，是因为运行一个信号关联多个槽。QObject::connect将新的连接添加到信号发送者附属的元对象的连接链表容器的相应信号的连接链表中。



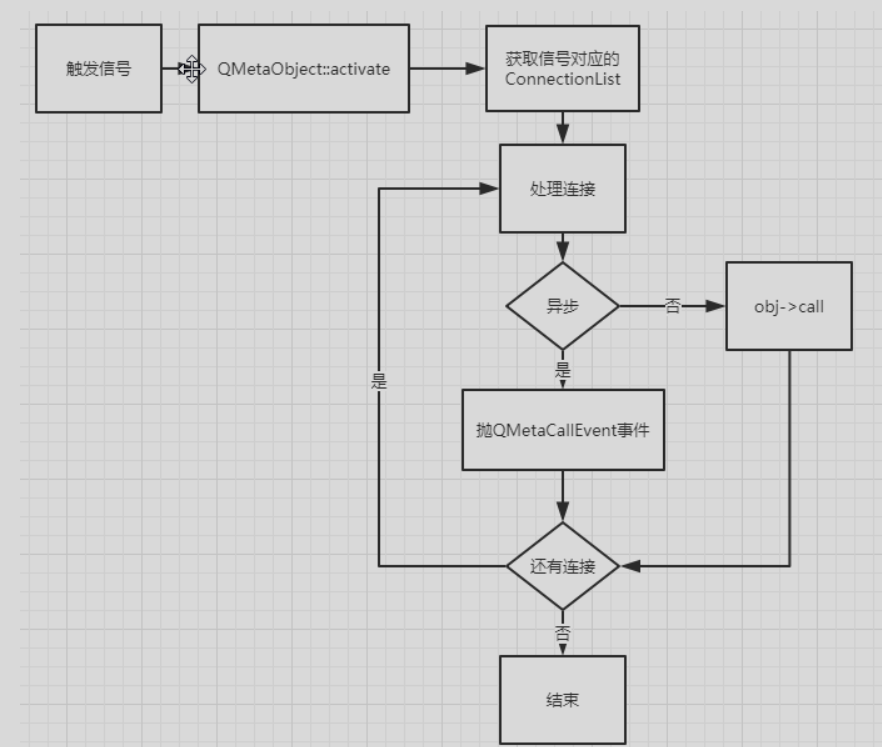
QObjectPrivate::ConnectionList是一个双向链表，其定义如下：



链表中元素的类型为Connection，其定义如下：



Qt中信号的触发：触发信号后，执行流程如下



主要执行的函数：QMetaObject::activate的代码如下

void QMetaObject::activate(QObject \*sender, **const** QMetaObject \*m,

int local\_signal\_index,void \*\*argv)

{

int signalOffset;

int methodOffset;

computeOffsets(m, &signalOffset, &methodOffset);

int signal\_index = signalOffset + local\_signal\_index;

**if** (!sender->d\_func()->isSignalConnected(signal\_index))

**return**; *// 如果发送的信号没有槽连接，直接返回*

**if** (sender->d\_func()->blockSig)

**return**;*//如果阻塞，直接返回*

int signal\_absolute\_index = methodOffset + local\_signal\_index;

void \*empty\_argv[] = { 0 };

**if** (qt\_signal\_spy\_callback\_set.signal\_begin\_callback != 0)

{

qt\_signal\_spy\_callback\_set.signal\_begin\_callback(sender, signal\_absolute\_index,

argv ? argv : empty\_argv);

}

Qt::HANDLE currentThreadId = QThread::currentThreadId();

QMutexLocker locker(signalSlotLock(sender));

*//获取发送者的连接链表容器*

QObjectConnectionListVector \*connectionLists = sender->d\_func()->connectionLists;

**if** (!connectionLists)

{

locker.unlock();

**if** (qt\_signal\_spy\_callback\_set.signal\_end\_callback != 0)

qt\_signal\_spy\_callback\_set.signal\_end\_callback(sender, signal\_absolute\_index);

**return**;

}

++connectionLists->inUse;

*//从发送者的连接链表容器中使用信号索引作为索引，获取相应的连接链表*

**const** QObjectPrivate::ConnectionList \***list**;

**if** (signal\_index < connectionLists->count())

**list** = &connectionLists->at(signal\_index);

**else**

**list** = &connectionLists->allsignals;

**do** {

*//索取发送的信号的连接链表的第一个连接*

QObjectPrivate::Connection \*c = **list**->first;

**if** (!c) **continue**;*//如果连接为空，继续*

*// We need to check against last here to ensure that signals added*

*// during the signal emission are not emitted in this emission.*

QObjectPrivate::Connection \*last = **list**->last;

**do**

{

**if** (!c->receiver)

**continue**;*//如果连接的接收者为空，继续*

QObject \* **const** receiver = c->receiver;

**const** bool receiverInSameThread = currentThreadId == receiver->d\_func()->threadData->threadId;

*// determine if this connection should be sent immediately or*

*// put into the event queue*

**if** ((c->connectionType == Qt::AutoConnection && !receiverInSameThread)

|| (c->connectionType == Qt::QueuedConnection))

{

queued\_activate(sender, signal\_absolute\_index, c, argv ? argv : empty\_argv);

**continue**;

*#ifndef QT\_NO\_THREAD*

}

*//阻塞队列连接类型*

**else** **if** (c->connectionType == Qt::BlockingQueuedConnection)

{

locker.unlock();

**if** (receiverInSameThread)

{

qWarning("Qt: Dead lock detected while activating a BlockingQueuedConnection: "

"Sender is %s(%p), receiver is %s(%p)",

sender->metaObject()->className(), sender,

receiver->metaObject()->className(), receiver);

}

QSemaphore semaphore;

QCoreApplication::postEvent(receiver, **new** QMetaCallEvent(c->method\_offset, c->method\_relative,

c->callFunction,

sender, signal\_absolute\_index,

0, 0,

argv ? argv : empty\_argv,

&semaphore));

semaphore.acquire();

locker.relock();

**continue**;

*#endif*

}

QObjectPrivate::Sender currentSender;

QObjectPrivate::Sender \*previousSender = 0;

**if** (receiverInSameThread)

{

currentSender.sender = sender;

currentSender.signal = signal\_absolute\_index;

currentSender.ref = 1;

previousSender = QObjectPrivate::setCurrentSender(receiver, ¤tSender);

}

*//获取连接的回调函数指针*

**const** QObjectPrivate::StaticMetaCallFunction callFunction = c->callFunction;

**const** int method\_relative = c->method\_relative;

*//如果连接的方法的偏移小于接收者的元对象的方法的偏移*

**if** (callFunction && c->method\_offset <= receiver->metaObject()->methodOffset())

{

*//we compare the vtable to make sure we are not in the destructor of the object.*

locker.unlock();

**if** (qt\_signal\_spy\_callback\_set.slot\_begin\_callback != 0)

qt\_signal\_spy\_callback\_set.slot\_begin\_callback(receiver, c->method(), argv ? argv : empty\_argv);

*//根据接收者的方法偏移，接收者等参数调用qt\_static\_metacall回调函数*

callFunction(receiver, QMetaObject::InvokeMetaMethod, method\_relative, argv ? argv : empty\_argv);

**if** (qt\_signal\_spy\_callback\_set.slot\_end\_callback != 0)

qt\_signal\_spy\_callback\_set.slot\_end\_callback(receiver, c->method());

locker.relock();

}

**else**

{

**const** int method = method\_relative + c->method\_offset;

locker.unlock();

**if** (qt\_signal\_spy\_callback\_set.slot\_begin\_callback != 0)

{

qt\_signal\_spy\_callback\_set.slot\_begin\_callback(receiver,

method,

argv ? argv : empty\_argv);

}

*//根据接收者、接收者的方法索引等参数调用发送元对象的metacall*

metacall(receiver, QMetaObject::InvokeMetaMethod, method, argv ? argv : empty\_argv);

**if** (qt\_signal\_spy\_callback\_set.slot\_end\_callback != 0)

qt\_signal\_spy\_callback\_set.slot\_end\_callback(receiver, method);

locker.relock();

}

**if** (receiverInSameThread)

QObjectPrivate::resetCurrentSender(receiver, ¤tSender, previousSender);

**if** (connectionLists->orphaned)

**break**;

} **while** (c != last && (c = c->nextConnectionList) != 0);

**if** (connectionLists->orphaned)

**break**;

} **while** (**list** != &connectionLists->allsignals &&

*//start over for all signals;*

((**list** = &connectionLists->allsignals), **true**));

--connectionLists->inUse;

Q\_ASSERT(connectionLists->inUse >= 0);

**if** (connectionLists->orphaned)

{

**if** (!connectionLists->inUse)

delete connectionLists;

}

**else** **if** (connectionLists->dirty)

{

sender->d\_func()->cleanConnectionLists();

}

locker.unlock();

**if** (qt\_signal\_spy\_callback\_set.signal\_end\_callback != 0)

qt\_signal\_spy\_callback\_set.signal\_end\_callback(sender, signal\_absolute\_index);

}

可以看出，首先函数中有锁存在，可以保证多线程使用，当信号触发时，会遍历信号对应的ConnectionList，并执行其中所有连接的槽函数。

Qt信号与槽的优缺点：

Qt信号与槽相对于回调函数机制更灵活，简洁易懂

缺点：

1. 运行效率相比于回调函数有所降低
2. 如果使用不当，会造成死循环，例如槽函数中再次发射所接收的信号。
3. 一个信号与多个槽相关联，信号发射后，槽函数执行顺序是随机的
4. C++中的宏定义，构造函数，嵌套类不能定义在信号或槽区域内