QObject Class

QObject 类是所有 Qt 对象的基类。更多的...

Header:	**#include < QObject>**
CMake:	find_package(Qt6 REQUIRED COMPONENTS Core) target_link_libraries(mytarget PRIVATE Qt6::Core)
qmake:	QT += core
Inherited By:	Q3DObject, Q3DScene, Q3DTheme, QAbstract3DAxis, QAbstract3DInputHandler, QAbstract3DSeries, QAbstractAnimation, QAbstractAxis, QAbstractDataProxy, QAbstractEventDispatcher, QAbstractGrpcClient, QAbstractHttpServer, QAbstractItemDelegate, QAbstractItemModel, QAbstractItemModelTester, QAbstractNetworkCache, QAbstractOAuth, QAbstractOAuthReplyHandler, QAbstractSeries, QAbstractState, QAbstractTextDocumentLayout, QAbstractTransition, QAccessiblePlugin, QAction, QActionGroup, QAmbientSound, QAudioDecoder, QAudioEngine, QAudioInput, QAudioListener, QAudioOutput, QAudioRoom, QAudioSink, QAudioSource, QAxBaseObject, QAxFactory, QAxScript, QAxScriptManager, QBarSet, QBluetoothDeviceDiscoveryAgent, QBluetoothLocalDevice, QBluetoothServer, QBluetoothServiceDiscoveryAgent, QBoxSet, QButtonGroup, QCamera, QCanBus, QCanBusDevice, QCandlestickModelMapper, QCandlestickSet, QClipboard, QCoapClient, QCompleter, QCoreApplication, QCustom3DItem, QDataWidgetMapper, QDBusAbstractAdaptor, QDBusAbstractInterface, QDBusPendingCallWatcher, QDBusServer, QDBusServiceWatcher, QDBusVirtualObject, QDesignerFormEditorInterface, QDesignerFormWindowManagerInterface, QDnsLookup, QDrag, QDtls, QDtlsClientVerifier, QEventLoop, QExtensionFactory, QExtensionManager, QFileSelector, QFileSystemWatcher, QFutureWatcher, QGenericPlugin, QGeoAreaMonitorSource, QGeoCodeReply, QGeoCodingManager, QGeoCodingManagerEngine, QGeoPositionInfoSource, QGeoRouteReply, QGeoRoutingManager, QGeoRoutingManagerEngine, QGeoSatelliteInfoSource, QGeoServiceProvider, QGesture, QGraphicsAnchor, QGraphicsEffect, QGraphicsItemAnimation, QGraphicsObject, QGraphicsScene, QGraphicsTransform, QGrpcOperation, QHelpEngineCore, QHelpFilterEngine, QHelpSearchEngine, QHttpMultiPart, QIconEnginePlugin, QImageCapture, QImageIOPlugin, QInputDevice, QInputMethod, QIODevice, QItemSelectionModel, QJSEngine, QLayout, QLegendMarker, QLibrary, QLocalServer, QLowEnergyController, QLowEnergyService, QMaskGenerator, QMediaCaptureSession, QMediaDevices, QMediaPlayer, QMediaRecorder, QMimeData, QModbusDevice, QModbusReply, QMovie, QMqttClient, QMqttSubscription, QNearFieldManager, QNearFieldTarget, QNetworkAccessManager, QNetworkCookieJar, QNetworkInformation, QObjectCleanupHandler, QOffscreenSurface, QOpcUaClient, QOpcUaGdsClient, QOpcUaHistoryReadResponse, QOpcUaKeyPair, QOpcUaNode, QOpcUaProvider, QOpenGLContext, QOpenGLContextGroup, QOpenGLDebugLogger, QOpenGLShader, QOpenGLShaderProgram, QOpenGLTimeMonitor, QOpenGLTimerQuery, QOpenGLVertexArrayObject, QPdfDocument, QPdfPageNavigator, QPdfPageRenderer, QPdfWriter, QPieSlice, QPlaceManager, QPlaceManagerEngine, QPlaceReply, QPluginLoader, QQmlComponent, QQmlContext, QQmlEngineExtensionPlugin, QQmlExpression, QQmlExtensionPlugin, QQmlFileSelector, QQmlImageProviderBase, QQmlPropertyMap, QQuick3DObject, QQuickAttachedPropertyPropagator, QQuickImageResponse, QQuickItem, QQuickItemGrabResult, QQuickRenderControl, QQuickTextDocument, QQuickTextureFactory, QQuickWebEngineProfile, QRemoteObjectAbstractPersistedStore, QRemoteObjectNode, QRemoteObjectPendingCallWatcher, QRemoteObjectReplica, QScreen, QScreenCapture, QScroller, QScxmlDataModel, QScxmlInvokableService, QScxmlInvokableServiceFactory, QScxmlStateMachine, QSensor, QSensorBackend, QSensorReading, QSessionManager, QSettings, QSGTexture, QSGTextureProvider, QSharedMemory, QShortcut, QSignalMapper, QSignalSpy, QSocketNotifier, QSoundEffect, QSpatialSound, QSqlDriver, QSqlDriverPlugin, QStyle, QStyleHints, QStylePlugin, QSvgRenderer, QSyntaxHighlighter, QSystemTrayIcon, Qt3DAnimation::QAbstractAnimation, Qt3DAnimation::QAnimationController, Qt3DAnimation::QAnimationGroup, Qt3DAnimation::QMorphTarget, Qt3DCore::QAbstractAspect, Qt3DCore::QAspectEngine, Qt3DCore::QNode, Qt3DCore::Quick::QQmlAspectEngine, Qt3DInput::QInputDeviceIntegration, Qt3DInput::QKeyEvent, Qt3DInput::QMouseEvent, Qt3DInput::QWheelEvent, Qt3DRender::QGraphicsApiFilter, Qt3DRender::QPickEvent, Qt3DRender::QRenderCapabilities, Qt3DRender::QRenderCaptureReply, Qt3DRender::QStencilOperationArguments, Qt3DRender::QStencilTestArguments,

Header:	**#include < QObject>**

Qt3DRender::QTextureWrapMode, QTcpServer, QTextDocument, QTextObject, QTextToSpeech, QThread, QThreadPool, QTimeLine, QTimer, QTranslator, QUiLoader, QUndoGroup, QUndoStack, QValidator, QValue3DAxisFormatter, QVideoSink, QVirtualKeyboardAbstractInputMethod, QVirtualKeyboardDictionary, QVirtualKeyboardDictionaryManager, QVirtualKeyboardInputContext, QVirtualKeyboardInputEngine, QVirtualKeyboardObserver, QVirtualKeyboardTrace, QWaylandClient, QWaylandObject, QWaylandQuickShellIntegration, QWaylandSurfaceGrabber, QWaylandView, QWaylandXdgPopup, QWaylandXdgToplevel, QWebChannel, QWebChannelAbstractTransport, QWebEngineContextMenuRequest, QWebEngineCookieStore, QWebEngineDownloadRequest, QWebEngineHistory, QWebEngineNavigationRequest, QWebEngineNewWindowRequest, QWebEngineNotification, QWebEnginePage, QWebEngineProfile, QWebEngineUrlRequestInterceptor, QWebEngineUrlRequestJob, QWebEngineUrlSchemeHandler, QWebSocket, QWebSocketServer, QWidget, QWindow, and QWinEventNotifier

- 所有成员的列表，包括继承的成员
- 已弃用的成员

注意：该类中的所有函数都是reentrant。

注意：这些功能也thread-safe:

- connect()
- connect()
- connect()
- connect()
- connect()
- disconnect()
- disconnect()
- disconnect()
- deleteLater()

特性

- **objectName** : QString

公共方法

	QObject(QObject **parent* = **nullptr**)
virtual	**~QObject**()
QBindable	**bindableObjectName**()
bool	**blockSignals**(bool *block*)
const QObjectList &	**children**() const
QMetaObject::Connection	**connect**(const QObject **sender*, const char** signal, *const char* ***method*, Qt::ConnectionType *type* = Qt::AutoConnection) const

	QObject(**QObject** ***parent*** **= nullptr**)
bool	**disconnect**(const char **signal* = nullptr, const QObject** receiver* = nullptr, const char **method* = nullptr) const
bool	**disconnect**(const QObject **receiver*, const char** method* = nullptr) const
void	**dumpObjectInfo**() const
void	**dumpObjectTree**() const
QList	**dynamicPropertyNames**() const
virtual bool	**event**(QEvent **e*)
virtual bool	**eventFilter**(QObject **watched*, QEvent** event*)
T	**findChild**(const QString &*name* = QString(), Qt::FindChildOptions *options* = Qt::FindChildrenRecursively) const
QList< T>	**findChildren**(const QString &*name*, Qt::FindChildOptions *options* = Qt::FindChildrenRecursively) const
QList< T>	**findChildren**(Qt::FindChildOptions *options* = Qt::FindChildrenRecursively) const
QList< T>	**findChildren**(const QRegularExpression &*re*, Qt::FindChildOptions *options* = Qt::FindChildrenRecursively) const
bool	**inherits**(const char **className*) const
void	**installEventFilter**(QObject **filterObj*)
bool	**isQuickItemType**() const
bool	**isWidgetType**() const
bool	**isWindowType**() const
void	**killTimer**(int *id*)
virtual const QMetaObject *	**metaObject**() const
void	**moveToThread**(QThread **targetThread*)
QString	**objectName**() const
QObject *	**parent**() const
QVariant	**property**(const char **name*) const
void	**removeEventFilter**(QObject **obj*)
void	**setObjectName**(const QString &*name*)
void	**setObjectName**(QAnyStringView *name*)
void	**setParent**(QObject **parent*)
bool	**setProperty**(const char **name*, const QVariant &*value*)

	QObject(QObject **parent* = nullptr)
bool	**signalsBlocked**() const
int	**startTimer**(int *interval*, Qt::TimerType *timerType* = Qt::CoarseTimer)
int	**startTimer**(std::chrono::milliseconds *time*, Qt::TimerType *timerType* = Qt::CoarseTimer)
QThread *	**thread**() const

公共槽

void	**deleteLater**()

信号

void	**destroyed(QObject **obj* = nullptr)**
void	**objectNameChanged**(const QString &*objectName*)

静态公共成员

QMetaObject::Connection	**connect(const QObject sender*, const char signal, *const QObject **receiver*, const char **method*, Qt::ConnectionType *type* = Qt::AutoConnection)**
QMetaObject::Connection	**connect**(const QObject **sender, *const QMetaMethod &*signal*, const QObject** receiver, *const QMetaMethod &*method*, Qt::ConnectionType *type* = Qt::AutoConnection)
QMetaObject::Connection	**connect**(const QObject **sender*, PointerToMemberFunction *signal*, const QObject** receiver*, PointerToMemberFunction *method*, Qt::ConnectionType *type* = Qt::AutoConnection)
QMetaObject::Connection	**connect**(const QObject **sender*, PointerToMemberFunction *signal*, Functor *functor*)
QMetaObject::Connection	**connect**(const QObject **sender*, PointerToMemberFunction *signal*, const QObject** context*, Functor *functor*, Qt::ConnectionType *type* = Qt::AutoConnection)
bool	**disconnect**(const QObject **sender*, const char** signal, *const QObject **receiver*, const char **method*)
bool	**disconnect**(const QObject **sender, *const QMetaMethod &*signal*, const QObject** receiver, *const QMetaMethod &*method*)
bool	**disconnect**(const QMetaObject::Connection &*connection*)
bool	**disconnect**(const QObject **sender*, PointerToMemberFunction *signal*, const**

	QObject receiver*, PointerToMemberFunction *method*)
QMetaObject::Connection	**connect**(**const QObject sender*, const char signal**, *const QObject **receiver*, const char **method*, **Qt::ConnectionType** *type* = **Qt::AutoConnection**)
const QMetaObject	**staticMetaObject**
QString	**tr**(const char **sourceText*, const char** disambiguation* = nullptr, int *n* = -1)

protected function

virtual void	**childEvent**(**QChildEvent** ***event***)
virtual void	**connectNotify**(const QMetaMethod &*signal*)
virtual void	**customEvent**(QEvent **event*)
virtual void	**disconnectNotify**(const QMetaMethod &*signal*)
bool	**isSignalConnected**(const QMetaMethod &*signal*) const
int	**receivers**(const char **signal*) const
QObject *	**sender**() const
int	**senderSignalIndex**() const
virtual void	**timerEvent**(QTimerEvent **event*)

相关非成员

	QObjectList
T	**qobject_cast**(QObject **object*)
T	**qobject_cast**(const QObject **object*)

宏

QT_NO_NARROWING_CONVERSIONS_IN_CONNECT
Q_CLASSINFO(*Name*, *Value*)
Q_EMIT
Q_ENUM(...)
Q_ENUM_NS(...)
Q_FLAG(...)
Q_FLAG_NS(...)

QT_NO_NARROWING_CONVERSIONS_IN_CONNECT

Q_GADGET

Q_GADGET_EXPORT(*EXPORT_MACRO*)

Q_INTERFACES(...)

Q_INVOKABLE

Q_MOC_INCLUDE

Q_NAMESPACE

Q_NAMESPACE_EXPORT(*EXPORT_MACRO*)

Q_OBJECT

Q_PROPERTY(...)

Q_REVISION

Q_SET_OBJECT_NAME(*Object*)

Q_SIGNAL

Q_SIGNALS

Q_SLOT

Q_SLOTS

详细说明

QObject 是 Qt 的核心Object Model。该模型的核心特征是一种非常强大的无缝对象通信机制，称为signals and slots。您可以将信号连接到插槽connect() 并破坏与的连接disconnect()。为了避免永无休止的通知循环，您可以暂时阻止信号blockSignals()。受保护的函数connectNotify（） 和disconnectNotify() 使得跟踪连接成为可能。

QObjects 组织自己object trees。当您创建一个以另一个对象作为父对象的 QObject 时，该对象将自动将其自身添加到父对象的 QObject 中children（） 列表。父对象拥有该对象的所有权；即，它将自动在其析构函数中删除其子级。您可以按名称查找对象，也可以使用类型来查找对象findChild（） 或者findChildren()。

每个对象都有一个objectName() 及其类名可以通过相应的metaObject（） （看QMetaObject::className()）。您可以使用以下方法确定对象的类是否继承了 QObject 继承层次结构中的另一个类inherits（） 功能。

当一个对象被删除时，它会发出一个destroyed（） 信号。您可以捕获此信号以避免对 QObject 的悬空引用。

QObjects 可以通过以下方式接收事件event() 并过滤其他对象的事件。看installEventFilter（） 和eventFilter（）了解详情。一个便利的处理者，childEvent()，可以重新实现以捕获子事件。

最后但并非最不重要的一点是，QObject 在 Qt 中提供了基本的计时器支持；看QTimer对定时器的高级支持。

请注意，Q_OBJECT宏对于任何实现信号、槽或属性的对象都是必需的。您还需要运行Meta Object Compiler在源文件上。我们强烈建议在 QObject 的所有子类中使用此宏，无论它们是否实际使用信号、槽和属性，因为不这样做可能会导致某些函数表现出奇怪的行为。

所有 Qt 小部件都继承 QObject。便利功能isWidgetType() 返回一个对象是否实际上是一个小部件。它比qobject_cast[QWidget](https://doc-qt-io.translate.goog/qt-6/qwidget.html?_x_tr_sl=auto&_x_tr_tl=zh-CN&_x_tr_hl=zh-CN&_x_tr_pto=wapp)*(*obj*) 或*obj* ->inherits（“QWidget”）。

一些QObject函数，例如children()，返回一个QObjectList。QObjectList是一个类型定义QList<QObject *>。

线程亲和力

据说 QObject 实例具有*线程关联性*，或者说它*存在*于某个线程中。当 QObject 接收到queued signal或一个posted event，槽或事件处理程序将在对象所在的线程中运行。

注意：如果 QObject 没有线程亲和性（也就是说，如果thread() 返回零），或者如果它位于没有运行事件循环的线程中，则它无法接收排队信号或发布的事件。

默认情况下，QObject 存在于创建它的线程中。可以使用以下方式查询对象的线程关联性thread() 并更改为使用moveToThread()。

所有 QObject 必须与其父对象位于同一线程中。最后：

- setParent如果涉及的两个 QObject 位于不同的线程中，() 将失败。
- 当 QObject 被移动到另一个线程时，它的所有子对象也将自动移动。
- moveToThread如果 QObject 有父对象，() 将失败。
- 如果 QObjects 创建于QThread::run()，他们不能成为QThread对象因为QThread不存在于调用的线程中QThread::run()。

注意： QObject 的成员变量*不会*自动成为其子变量。父子关系必须通过传递一个指向子对象的指针来设置constructor，或通过致电setParent()。如果没有这一步，对象的成员变量将保留在旧线程中moveToThread（） 叫做。

无复制构造函数或赋值运算符

QObject 既没有复制构造函数，也没有赋值运算符。这是设计使然。实际上，它们是被声明的，但是在`private`带有宏的部分中Q_DISABLE_COPY()。事实上，所有从 QObject 派生（直接或间接）的 Qt 类都使用此宏来声明其复制构造函数和赋值运算符为私有。推理可以在以下讨论中找到Identity vs Value在Qt上Object Model页。

主要结果是您应该使用指向 QObject（或 QObject 子类）的指针，否则您可能会想使用 QObject 子类作为值。例如，如果没有复制构造函数，则无法使用 QObject 的子类作为要存储在容器类之一中的值。您必须存储指针。

自动连接

Qt 的元对象系统提供了一种自动连接 QObject 子类及其子类之间的信号和槽的机制。只要使用合适的对象名称定义对象，并且槽遵循简单的命名约定，就可以在运行时通过QMetaObject::connectSlotsByName（） 功能。

uic*生成调用此函数的代码，以启用使用Qt Designer*创建的表单上的小部件之间执行自动连接。*有关使用Qt Designer*自动连接的更多信息，请参阅Using a Designer UI File in Your Application*Qt Designer*手册的部分。

动态属性

从 Qt 4.2 开始，可以在运行时向 QObject 实例添加和删除动态属性。动态属性不需要在编译时声明，但它们提供与静态属性相同的优点，并且使用相同的 API 进行操作 - 使用property() 阅读它们并setProperty() 来写它们。

从 Qt 4.3 开始，支持动态属性Qt Designer，并且标准 Qt 小部件和用户创建的表单都可以被赋予动态属性。

国际化 (I18n)

所有 QObject 子类都支持 Qt 的翻译功能，从而可以将应用程序的用户界面翻译成不同的语言。

为了使用户可见的文本可翻译，必须将其包含在对tr（） 功能。这在Writing Source Code for Translation文档。

也可以看看QMetaObject,QPointer,QObjectCleanupHandler,Q_DISABLE_COPY（），和Object Trees & Ownership。

属性文档

[bindable]objectName : QString

注：该属性支持QProperty绑定。

该属性保存该对象的名称

您可以使用以下命令按名称（和类型）查找对象findChild()。您可以找到一组对象findChildren()。

```
qDebug("MyClass::setPrecision(): (%s) invalid precision %f",
       qPrintable(objectName()), newPrecision);
```

默认情况下，该属性包含一个空字符串。

也可以看看metaObject（） 和QMetaObject::className()。

成员函数文档

*[invokable]QObject::QObject(QObject *parent = nullptr)*

使用父对象构造一个对象*parent*。

对象的父对象可以被视为该对象的所有者。例如，一个dialog box是 的父级**OK**和**Cancel**它包含的按钮。

父对象的析构函数会销毁所有子对象。

环境*parent*构造`nullptr`一个没有父对象的对象。如果该对象是一个小部件，它将成为一个顶级窗口。

注意：该函数可以通过元对象系统和 QML 调用。看Q_INVOKABLE。

也可以看看parent(),findChild（），和findChildren()。

[virtual]QObject::~QObject()

销毁对象，删除其所有子对象。

传入和传出该对象的所有信号都会自动断开，并且该对象的任何挂起的已发布事件都会从事件队列中删除。然而，使用通常更安全deleteLater() 而不是删除QObject直接子类化。

警告：所有子对象都将被删除。如果这些对象中的任何一个位于堆栈或全局中，那么您的程序迟早会崩溃。我们不建议从父对象外部保存指向子对象的指针。如果您仍然这样做，destroyed() 信号使您有机会检测对象何时被销毁。

警告：删除QObject当挂起的事件正在等待传递时可能会导致崩溃。您不得删除QObject如果它存在于与当前正在执行的线程不同的线程中，则直接执行。使用deleteLater() 而是，这将导致事件循环在所有待处理事件传递给对象后删除该对象。

也可以看看deleteLater()。

bool QObject::blockSignals(bool block)

如果*block*为 true 时，该对象发出的信号将被阻止（即，发出信号不会调用与其连接的任何对象）。如果*block*为 false，则不会发生此类阻塞。

返回值是之前的值signalsBlocked()。

请注意，destroyed即使该对象的信号已被阻止，() 信号也会被发射。

被阻止时发出的信号不会被缓冲。

也可以看看signalsBlocked () 和QSignalBlocker。

*[virtual protected]void QObject::childEvent(QChildEvent *event)*

该事件处理程序可以在子类中重新实现以接收子事件。该事件在*event*范围。

QEvent::ChildAdded和QEvent::ChildRemoved添加或删除子项时，事件会发送到对象。在这两种情况下，您只能依靠孩子作为QObject， 或者如果isWidgetType() 返回`true`, aQWidget。 （这是因为，在ChildAdded在这种情况下，孩子还没有完全构造出来，并且在ChildRemoved如果它可能已经被破坏了）。

QEvent::ChildPolished当子项被完善或添加完善的子项时，事件将发送到小部件。如果您收到子级抛光事件，则子级的构建通常已完成。但是，这并不能得到保证，并且在小部件的构造函数执行期间可能会传递多个抛光事件。

对于每个子小部件，您都会收到一个ChildAdded事件，零个或多个ChildPolished事件，以及一个ChildRemoved事件。

这ChildPolished如果添加子项后立即将其删除，则事件将被忽略。如果一个子项在构建和销毁过程中被多次抛光，您可能会收到同一个子项的多个子项抛光事件，每次都使用不同的虚拟表。

也可以看看event()。

const QObjectList &QObject::children() const

返回子对象的列表。这QObjectList类在头文件中定义<QObject>如下：

```
typedef QList<QObject*> QObjectList;
```

添加的第一个孩子是first列表中的对象，最后添加的子对象是last列表中的对象，即新的子对象附加在末尾。

请注意，列表顺序会在以下情况发生变化：QWidget孩子们是raised或者lowered。升起的小部件将成为列表中的最后一个对象，降下的小部件将成为列表中的第一个对象。

也可以看看findChild(),findChildren(),parent（），和setParent()。

*[static]QMetaObject::Connection QObject::connect(const QObject **sender*, const char** *signal, const QObject **receiver, const char *method, Qt::ConnectionType type = Qt::AutoConnection)*

创建给定的连接*type*来自*signal*在里面*sender*反对*method*在里面*receiver*目的。返回连接的句柄，稍后可用于断开连接。

指定时必须使用SIGNAL()和宏SLOT()*signal*和*method*，例如：

```
QLabel *label = new QLabel;
QScrollBar *scrollBar = new QScrollBar;
QObject::connect(scrollBar, SIGNAL(valueChanged(int)),
                 label,  SLOT(setNum(int)));
```

此示例确保标签始终显示当前滚动条值。请注意，信号和槽参数不得包含任何变量名称，只能包含类型。例如，以下内容将不起作用并返回 false：

```
// WRONG
QObject::connect(scrollBar, SIGNAL(valueChanged(int value)),
                 label, SLOT(setNum(int value)));
```

一个信号也可以连接到另一个信号：

```
class MyWidget : public QWidget
{
    Q_OBJECT

public:
    MyWidget();

signals:
    void buttonClicked();

private:
    QPushButton *myButton;
};

MyWidget::MyWidget()
```

```
{
    myButton = new QPushButton(this);
    connect(myButton, SIGNAL(clicked()),
            this, SIGNAL(buttonClicked()));
}
```

在此示例中，`MyWidget`构造函数中继来自私有成员变量的信号，并使其在与 相关的名称下可用`MyWidget`。

一个信号可以连接到许多插槽和信号。一个插槽可以连接多个信号。

如果一个信号连接到多个插槽，则当信号发出时，这些插槽将按照连接时的相同顺序被激活。

该函数返回一个QMetaObject::Connection如果成功将信号连接到插槽，则表示连接的句柄。如果无法创建连接，连接句柄将无效，例如，如果QObject无法验证两者的存在*signal*或者*method*，或者如果他们的签名不兼容。您可以通过将句柄转换为布尔值来检查句柄是否有效。

默认情况下，您建立的每个连接都会发出一个信号；对于重复连接会发出两个信号。您可以用一个命令来断开所有这些连接disconnect（） 称呼。如果您通过了Qt::UniqueConnection *type*，只有在不重复的情况下才会建立连接。如果已经存在重复项（相同对象上完全相同的插槽的完全相同的信号），连接将失败并且连接将返回无效QMetaObject::Connection。

注意： Qt::UniqueConnections 不适用于 lambda、非成员函数和函子；它们仅适用于连接到成员函数。

可选的*type*参数描述要建立的连接类型。特别是，它确定特定信号是立即传送到时隙还是稍后排队传送。如果信号排队，参数必须是 Qt 元对象系统已知的类型，因为 Qt 需要复制参数以将它们存储在幕后的事件中。如果您尝试使用排队连接并收到错误消息

```
QObject::connect: Cannot queue arguments of type 'MyType'
(Make sure 'MyType' is registered using qRegisterMetaType().)
```

称呼qRegisterMetaType() 在建立连接之前注册数据类型。

注：该函数是thread-safe。

也可以看看disconnect(),sender(),qRegisterMetaType(),Q_DECLARE_METATYPE（），和Differences between String-Based and Functor-Based Connections。

[static]QMetaObject::Connection QObject::connect(const QObject **sender, const QMetaMethod &signal*, const QObject** receiver, const QMetaMethod &method*, Qt::ConnectionType type = Qt::AutoConnection)*

创建给定的连接*type*来自*signal*在里面*sender*反对*method*在里面*receiver*目的。返回连接的句柄，稍后可用于断开连接。

如果无法创建连接，则连接句柄将无效，例如参数无效。您可以检查是否QMetaObject::Connection通过将其转换为 bool 来使其有效。

该函数的工作方式`connect(const QObject *sender, const char *signal, const QObject *receiver, const char *method, Qt::ConnectionType type)`与QMetaMethod指定信号和方法。

也可以看看connect（const QObject *发送方、 const char *信号、 const QObject *接收方、 const char *方法、 Qt::ConnectionType 类型）。

*QMetaObject::Connection QObject::connect(const QObject **sender*, const **char** signal, const char **method, Qt::ConnectionType type = Qt::AutoConnection) const*

该函数重载了 connect()。

连接*signal*来自*sender*对象到这个对象的*method*。

相当于连接(*sender*,*signal*, `this`,*method*,*type*）。

您建立的每个连接都会发出一个信号，因此重复的连接会发出两个信号。您可以使用以下命令断开连接disconnect()。

注：该函数是thread-safe。

也可以看看disconnect()。

*[static]template < typename PointerToMemberFunction> QMetaObject::Connection QObject::connect(const QObject **sender*, **PointerToMemberFunction signal, const QObject** receiver*, PointerToMemberFunction method, Qt::ConnectionType type = Qt::AutoConnection)*

该函数重载了 connect()。

创建给定的连接*type*来自*signal*在里面*sender*反对*method*在里面*receiver*目的。返回连接的句柄，稍后可用于断开连接。

该信号必须是在标头中声明为信号的函数。槽函数可以是任何可以连接到信号的成员函数。如果信号至少具有与槽一样多的参数，并且信号和槽中相应参数的类型之间存在隐式转换，则槽可以连接到给定信号。

例子：

```
QLabel *label = new QLabel;
QLineEdit *lineEdit = new QLineEdit;
QObject::connect(lineEdit, &QLineEdit::textChanged,
                 label,  &QLabel::setText);
```

此示例确保标签始终显示当前行编辑文本。

一个信号可以连接到许多插槽和信号。一个插槽可以连接多个信号。

如果一个信号连接到多个插槽，则当信号发出时，这些插槽将按照与建立连接的顺序相同的顺序激活

如果该函数成功地将信号连接到槽，则返回一个连接句柄。如果无法创建连接，连接句柄将无效，例如，如果QObject无法验证是否存在*signal*（如果它没有声明为信号）您可以检查是否QMetaObject::Connection通过将其转换为 bool 来使其有效。

默认情况下，您建立的每个连接都会发出一个信号；对于重复连接会发出两个信号。您可以用一个命令来断开所有这些连接disconnect（） 称呼。如果您通过了Qt::UniqueConnection *type*，只有在不重复的情况下才会建立连接。如果已经存在重复项（相同对象上完全相同的插槽的完全相同的信号），连接将失败并且连接将返回无效QMetaObject::Connection。

可选的*type*参数描述要建立的连接类型。特别是，它确定特定信号是立即传送到时隙还是稍后排队传送。如果信号排队，参数必须是 Qt 元对象系统已知的类型，因为 Qt 需要复制参数以将它们存储在幕后的事件中。如果您尝试使用排队连接并收到错误消息

```
QObject::connect: Cannot queue arguments of type 'MyType'
(Make sure 'MyType' is registered using qRegisterMetaType().)
```

确保声明参数类型Q_DECLARE_METATYPE

重载的函数可以通过以下方法解决qOverload。

注：该函数是thread-safe。

也可以看看Differences between String-Based and Functor-Based Connections。

*[static]template <typename PointerToMemberFunction, typename Functor> QMetaObject::Connection QObject::connect(const QObject *sender, PointerToMemberFunction signal, Functor functor)*

该函数重载了 connect()。

从以下位置创建连接*signal*在*sender*反对*functor*，并返回连接的句柄

该信号必须是在标头中声明为信号的函数。槽函数可以是任何可以连接到信号的函数或函子。如果信号具有至少与槽函数一样多的参数，则槽函数可以连接到给定信号。信号和槽中相应参数的类型之间必须存在隐式转换。

例子：

```
void someFunction();
QPushButton *button = new QPushButton;
QObject::connect(button, &QPushButton::clicked, someFunction);
```

还可以使用 Lambda 表达式：

```
QByteArray page = ...;
QTcpSocket *socket = new QTcpSocket;
socket->connectToHost("qt-project.org", 80);
QObject::connect(socket, &QTcpSocket::connected, [=] () {
        socket->write("GET " + page + "\r\n");
    });
```

如果发送者被破坏，连接将自动断开。但是，您应该注意，当发出信号时，函子中使用的任何对象仍然处于活动状态。

重载的函数可以通过以下方法解决qOverload。

注：该函数是thread-safe。

*[static]template <typename PointerToMemberFunction, typename Functor> QMetaObject::Connection QObject::connect(const QObject **sender*, PointerToMemberFunction signal, const QObject *context*, Functor functor, Qt::ConnectionType type = Qt::AutoConnection)*

该函数重载了 connect()。

创建给定的连接*type*从*signal*在*sender*反对*functor*被放置在特定的事件循环中*context*，并返回连接的句柄。

注意： Qt::UniqueConnections 不适用于 lambda、非成员函数和函子；它们仅适用于连接到成员函数。

该信号必须是在标头中声明为信号的函数。槽函数可以是任何可以连接到信号的函数或函子。如果信号具有至少与槽函数一样多的参数，则槽函数可以连接到给定信号。信号和槽中相应参数的类型之间必须存在隐式转换。

例子：

```
void someFunction();
QPushButton *button = new QPushButton;
QObject::connect(button, &QPushButton::clicked, this, someFunction, Qt::QueuedConnection);
```

还可以使用 Lambda 表达式：

```
QByteArray page = ...;
QTcpSocket *socket = new QTcpSocket;
socket->connectToHost("qt-project.org", 80);
QObject::connect(socket, &QTcpSocket::connected, this, [=] () {
        socket->write("GET " + page + "\r\n");
    }, Qt::AutoConnection);
```

如果发送者或上下文被破坏，连接将自动断开。但是，您应该注意，当发出信号时，函子中使用的任何对象仍然处于活动状态。

重载的函数可以通过以下方法解决qOverload。

注：该函数是thread-safe。

[virtual protected]void QObject::connectNotify(const QMetaMethod &signal)

当某些东西连接到时会调用此虚函数*signal*在这个对象中。

如果你想比较*signal*对于特定信号，您可以使用QMetaMethod::fromSignal () 如下：

```
if (signal == QMetaMethod::fromSignal(&MyObject::valueChanged)) {
    // signal is valueChanged
}
```

警告：该函数违反了面向对象的模块化原则。但是，当您仅在某些东西连接到信号时才需要执行昂贵的初始化时，它可能很有用。

警告：该函数是从执行连接的线程调用的，该线程可能与该对象所在的线程不同。该函数也可以用QObject内部互斥体已锁定。因此，不允许重新输入任何内容QObject功能，包括isSignalConnected()，来自您的重新实现。如果您在重新实现中锁定互斥锁，请确保您不会调用QObject函数的互斥体保存在其他地方，否则会导致死锁。

也可以看看connect（） 和disconnectNotify()。

*[virtual protected]void QObject::customEvent(QEvent *event)*

可以在子类中重新实现此事件处理程序以接收自定义事件。自定义事件是用户定义的事件，其类型值至少与QEvent::User的项目QEvent::Type枚举，通常是QEvent子类。该事件在*event*范围。

也可以看看event（） 和QEvent。

[slot]void QObject::deleteLater()

安排删除该对象。

当控制权返回到事件循环时，该对象将被删除。如果调用此函数时事件循环未运行（例如，之前在对象上调用了deleteLater()）QCoreApplication::exec())，一旦事件循环启动，该对象就会被删除。如果在主事件循环停止后调用deleteLater()，则该对象将不会被删除。从 Qt 4.8 开始，如果对存在于没有运行事件循环的线程中的对象调用deleteLater()，则该对象将在线程结束时被销毁。

请注意，进入和离开新的事件循环（例如，通过打开模式对话框）将不会*执行*延迟删除；对于要删除的对象，控件必须返回到调用deleteLater() 的事件循环。这不适用于在先前的嵌套事件循环仍在运行时删除的对象：Qt 事件循环将在新的嵌套事件循环启动后立即删除这些对象。

注意：多次调用该函数是安全的；当第一个延迟删除事件被传递时，该对象的任何待处理事件都将从事件队列中删除。

注：该函数是thread-safe。

也可以看看destroyed（） 和QPointer。

*[signal]void QObject::destroyed(QObject *obj = nullptr)*

该信号在物体之前立即发出*obj*在发生任何事件后被销毁QPointer已通知，无法屏蔽。

该信号发出后，所有对象的子对象都会立即被销毁。

也可以看看deleteLater（） 和QPointer。

*[static]bool QObject::disconnect(const QObject **sender*, const char *signal, const QObject **receiver, const char *method)*

断开连接*signal*在对象中*sender*从*method*在对象中*receiver*。`true`如果连接成功断开则返回；否则返回`false`.

当涉及的任何一个对象被销毁时，信号槽连接就会被删除。

connect() 通常以三种方式使用，如以下示例所示。

1. 断开与对象信号连接的所有连接：

```
disconnect(myObject, nullptr, nullptr, nullptr);
```

相当于非静态重载函数

```
myObject->disconnect();
```

2. 断开连接到特定信号的所有连接：

```
disconnect(myObject, SIGNAL(mySignal()), nullptr, nullptr);
```

相当于非静态重载函数

```
myObject->disconnect(SIGNAL(mySignal()));
```

3. 断开特定接收器的连接：

```
disconnect(myObject, nullptr, myReceiver, nullptr);
```

相当于非静态重载函数

```
myObject->disconnect(myReceiver);
```

`nullptr`可以用作通配符，分别表示“任何信号”、“任何接收对象”或“接收对象中的任何槽”。

这*sender*也许永远不会`nullptr`。（您无法在一次调用中断开来自多个对象的信号。）

如果*signal*是`nullptr`，它断开连接*receiver*和*method*来自任何信号。如果没有，则仅断开指定的信号。

如果*receiver*是的`nullptr`，它会断开连接到的任何东西*signal*。如果不是，则插入除*receiver*没有断开连接。

如果*method*是的`nullptr`，它会断开所有连接到的连接*receiver*。如果没有，则仅指定插槽*method*将被断开连接，并且所有其他插槽将保持不变。这*method*必须是`nullptr`如果*receiver*被遗漏，因此您无法断开所有对象上特定命名的插槽。

注意：断开所有信号槽连接也会断开QObject::destroyed() 信号（如果已连接）。这样做可能会对依赖此信号来清理资源的类产生不利影响。建议仅断开由应用程序代码连接的特定信号。

注：该函数是thread-safe。

也可以看看connect()。

*[static]bool QObject::disconnect(const QObject **sender, const QMetaMethod &signal*, const QObject** receiver, const QMetaMethod &method*)*

断开连接*signal*在对象中*sender*从*method*在对象中*receiver*。`true`如果连接成功断开则返回；否则返回`false`.

`disconnect(const QObject *sender, const char *signal, const QObject *receiver, const char *method)`该函数提供了与但使用相同的可能性QMetaMethod来表示要断开的信号和方法。

此外，如果出现以下情况，该函数将返回 false，并且不会断开任何信号和槽：

1. *signal*不是发送者类或其父类之一的成员。
2. *method*不是接收者类或其父类之一的成员。
3. *signal*实例不代表信号。

QMetaMethod() 可以用作通配符，含义为“任何信号”或“接收对象中的任何槽”。以同样的方式`nullptr`可以用于*receiver*意思是“任何接收对象”。在这种情况下，方法也应该是 QMetaMethod()。*sender*参数应该是 never `nullptr`。

注意：断开所有信号槽连接也会断开QObject::destroyed() 信号（如果已连接）。这样做可能会对依赖此信号来清理资源的类产生不利影响。建议仅断开由应用程序代码连接的特定信号。

也可以看看disconnect（const QObject *发送者，const char *信号，const QObject *接收者，const char *方法）。

*bool QObject::disconnect(const char **signal* = nullptr, const QObject** receiver* = nullptr, const char *method = nullptr) const*

该函数重载disconnect()。

断开连接*signal*从*method*的*receiver*。

当涉及的任何一个对象被销毁时，信号槽连接就会被删除。

注意：断开所有信号槽连接也会断开QObject::destroyed() 信号（如果已连接）。这样做可能会对依赖此信号来清理资源的类产生不利影响。建议仅断开由应用程序代码连接的特定信号。

注：该函数是thread-safe。

*bool QObject::disconnect(const QObject **receiver*, const char** method* = nullptr) const*

该函数重载disconnect()。

断开该对象中的所有信号*receiver*的*method*。

当涉及的任何一个对象被销毁时，信号槽连接就会被删除。

[static]bool QObject::disconnect(const QMetaObject::Connection &connection)

断开连接。

如果*connection*无效或已断开连接，不执行任何操作并返回 false。

也可以看看connect()。

[static]template < typename PointerToMemberFunction> bool QObject::disconnect(const QObject **sender*, PointerToMemberFunction signal, const QObject* receiver*, PointerToMemberFunction method)*

该函数重载disconnect()。

断开连接*signal*在对象中*sender*从*method*在对象中*receiver*。`true`如果连接成功断开则返回；否则返回`false`.

当涉及的任何一个对象被销毁时，信号槽连接就会被删除。

connect() 通常以三种方式使用，如以下示例所示。

1. 断开与对象信号连接的所有连接：

```
disconnect(myObject, nullptr, nullptr, nullptr);
```

2. 断开连接到特定信号的所有连接：

```
disconnect(myObject, &MyObject::mySignal(), nullptr, nullptr);
```

3. 断开特定接收器的连接：

```
disconnect(myObject, nullptr, myReceiver, nullptr);
```

4. 断开从一个特定信号到特定插槽的连接：

```
QObject::disconnect(lineEdit, &QLineEdit::textChanged,
                    label,  &QLabel::setText);
```

`nullptr`可以用作通配符，分别表示“任何信号”、“任何接收对象”或“接收对象中的任何槽”。

这*sender*也许永远不会`nullptr`。（您无法在一次调用中断开来自多个对象的信号。）

如果*signal*是`nullptr`，它断开连接*receiver*和*method*来自任何信号。如果没有，则仅断开指定的信号。

如果*receiver*是的`nullptr`，它会断开连接到的任何东西*signal*。如果不是，则仅断开指定接收器中的插槽。与非空断开连接（）*receiver*还断开与连接的插槽功能*receiver*作为他们的上下文对象。

如果*method*是的`nullptr`，它会断开所有连接到的连接*receiver*。如果没有，则仅指定插槽*method*将被断开连接，并且所有其他插槽将保持不变。这*method*必须是`nullptr`如果*receiver*被遗漏，因此您无法断开所有对象上特定命名的插槽。

注意：无法使用此重载来断开连接到函子或 lambda 表达式的信号。那是因为无法对它们进行比较。相反，使用需要 a 的重载QMetaObject::Connection

注：该函数是thread-safe。

也可以看看connect()。

[virtual protected]void QObject::disconnectNotify(const QMetaMethod &signal)

当某些东西与网络断开连接时，将调用此虚函数*signal*在这个对象中。

看connectNotify() 作为如何比较的示例*signal*具有特定的信号。

如果所有信号都与该对象断开连接（例如，信号参数disconnect() 是`nullptr`），disconnectNotify() 只调用一次，并且*signal*将是无效的QMetaMethod（QMetaMethod::isValid() 返回`false`）。

警告：该函数违反了面向对象的模块化原则。但是，它对于优化对昂贵资源的访问可能很有用。

警告：该函数是从执行断开连接的线程调用的，该线程可能与该对象所在的线程是不同的线程。该函数也可以用QObject内部互斥体已锁定。因此，不允许重新输入任何内容QObject功能，包括isSignalConnected()，来自您的重新实现。如果您在重新实现中锁定互斥锁，请确保您不会调用QObject函数的互斥体保存在其他地方，否则会导致死锁。

也可以看看disconnect（） 和connectNotify()。

void QObject::dumpObjectInfo() const

将此对象的有关信号连接等的信息转储到调试输出。

注意：在 Qt 5.9 之前，该函数不是 const。

也可以看看dumpObjectTree()。

void QObject::dumpObjectTree() const

将子树转储到调试输出。

注意：在 Qt 5.9 之前，该函数不是 const。

也可以看看dumpObjectInfo()。

QList[QByteArray](https://doc-qt-io.translate.goog/qt-6/qbytearray.html?_x_tr_sl=auto&_x_tr_tl=zh-CN&_x_tr_hl=zh-CN&_x_tr_pto=wapp)

QObject::dynamicPropertyNames() const

返回使用动态添加到对象的所有属性的名称setProperty()。

*[virtual]bool QObject::event(QEvent *e)*

该虚函数接收对象的事件，如果事件发生则应返回 true*e*被识别并处理。

可以重新实现 event() 函数来自定义对象的行为。

确保为所有未处理的事件调用父事件类实现。

例子：

```
class MyClass : public QWidget
{
    Q_OBJECT

public:
    MyClass(QWidget *parent = nullptr);
    ~MyClass();

    bool event(QEvent* ev) override
    {
        if (ev->type() == QEvent::PolishRequest) {
            // overwrite handling of PolishRequest if any
            doThings();
            return true;
        } else  if (ev->type() == QEvent::Show) {
            // complement handling of Show if any
            doThings2();
            QWidget::event(ev);
            return true;
        }
        // Make sure the rest of events are handled
        return QWidget::event(ev);
    }
};
```

也可以看看installEventFilter(),timerEvent(),QCoreApplication::sendEvent（），和QCoreApplication::postEvent()。

*[virtual]bool QObject::eventFilter(QObject **watched***, **QEvent** event**)*

如果此对象已作为事件过滤器安装，则过滤事件*watched*目的。

在您重新实现此函数时，如果您想过滤*event*out，即停止进一步处理，返回true；否则返回 false。

例子：

```
class MainWindow : public QMainWindow
{
public:
    MainWindow();

protected:
    bool eventFilter(QObject *obj, QEvent *ev) override;

private:
    QTextEdit *textEdit;
};

MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);

    textEdit->installEventFilter(this);
}

bool MainWindow::eventFilter(QObject *obj, QEvent *event)
{
    if (obj == textEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent*>(event);
            qDebug() << "Ate key press" << keyEvent->key();
            return true;
        } else {
            return false;
        }
    } else {
        // pass the event on to the parent class
        return QMainWindow::eventFilter(obj, event);
    }
}
```

请注意，在上面的示例中，未处理的事件被传递给基类的 eventFilter() 函数，因为基类可能出于其自身的内部目的重新实现了 eventFilter()。

一些事件，例如QEvent::ShortcutOverride必须明确接受（通过调用accept() ）以防止传播。

警告：如果在此函数中删除接收者对象，请务必返回 true。否则，Qt 会将事件转发给已删除的对象，程序可能会崩溃。

也可以看看installEventFilter()。

template < typename T> T QObject::findChild(const QString &name = QString(), Qt::FindChildOptions options = Qt::FindChildrenRecursively) const

返回该对象的子对象，该对象可以转换为类型 T 并称为*name*，或者`nullptr`如果不存在这样的对象。省略*name*参数导致所有对象名称匹配。搜索是递归执行的，除非*options*指定选项 FindDirectChildrenOnly。

如果有多个子代与搜索匹配，则返回最直接的祖先。如果有多个直接祖先，则未定义将返回哪一个。在这种情况下，findChildren应使用()。

此示例返回name`QPushButton`的子级，即使该按钮不是父级的直接子级：`parentWidget``"button1"`

```
QPushButton *button = parentWidget->findChild<QPushButton *>("button1");
```

此示例返回`QListWidget`的子级`parentWidget`：

```
QListWidget *list = parentWidget->findChild<QListWidget *>();
```

`QPushButton`此示例返回名为的`parentWidget`（其直接父级）的子级`"button1"`：

```
QPushButton *button = parentWidget->findChild<QPushButton *>("button1",
Qt::FindDirectChildrenOnly);
```

此示例返回 的`QListWidget`子级`parentWidget`，即其直接父级：

```
QListWidget *list = parentWidget->findChild<QListWidget *>(QString(),
Qt::FindDirectChildrenOnly);
```

也可以看看findChildren()。

template < typename T> QList< T> QObject::findChildren(const QString &name, Qt::FindChildOptions options = Qt::FindChildrenRecursively) const

返回具有给定值的该对象的所有子对象*name*可以转换为 T 类型，如果没有这样的对象，则转换为空列表。空值*name*参数导致所有对象都匹配，空对象仅匹配那些objectName是空的。搜索是递归执行的，除非*options*指定选项 FindDirectChildrenOnly。

以下示例显示如何查找`QWidget`指定`parentWidget`name的 child 列表`widgetname`：

```
QList<QWidget *> widgets = parentWidget.findChildren<QWidget *>("widgetname");
```

此示例返回`QPushButton`的所有子代`parentWidget`：

```
QList<QPushButton *> allPButtons = parentWidget.findChildren<QPushButton *>();
```

此示例返回 的所有`QPushButton`直接子代`parentWidget`：

```
QList<QPushButton *> childButtons = parentWidget.findChildren<QPushButton *>
(Qt::FindDirectChildrenOnly);
```

也可以看看findChild()。

[since 6.3]template < typename T> QList< T> QObject::findChildren(Qt::FindChildOptions options = Qt::FindChildrenRecursively) const

这是一个过载功能。

返回此对象的所有可转换为类型 T 的子对象，如果没有此类对象，则返回空列表。搜索是递归执行的，除非*options*指定选项 FindDirectChildrenOnly。

该功能是在 Qt 6.3 中引入的。

也可以看看findChild()。

template < typename T> QList< T> QObject::findChildren(const QRegularExpression &re, Qt::FindChildOptions options = Qt::FindChildrenRecursively) const

该函数重载了findChildren()。

返回此对象的子对象，这些子对象可以转换为类型 T 并且其名称与正则表达式匹配*re*，如果不存在此类对象，则为空列表。搜索是递归执行的，除非*options*指定选项 FindDirectChildrenOnly。

*bool QObject::inherits(const char *className) const*

返回`true`此对象是否是继承的类的实例*className*或一个QObject继承的子类*className*; 否则返回`false`.

类被认为继承了自身。

例子:

```
QTimer *timer = new QTimer;         // QTimer inherits QObject
timer->inherits("QTimer");          // returns true
timer->inherits("QObject");         // returns true
timer->inherits("QAbstractButton"); // returns false

// QVBoxLayout inherits QObject and QLayoutItem
QVBoxLayout *layout = new QVBoxLayout;
layout->inherits("QObject");        // returns true
layout->inherits("QLayoutItem");    // returns true (even though QLayoutItem is not a QObject)
```

如果您需要确定一个对象是否是特定类的实例以进行强制转换，请考虑使用qobject_cast改为 <Type *>(object)。

也可以看看metaObject（） 和qobject_cast()。

*void QObject::installEventFilter(QObject *filterObj)*

安装事件过滤器*filterObj*在这个物体上。例如：

```
monitoredObj->installEventFilter(filterObj);
```

事件过滤器是一个接收发送到该对象的所有事件的对象。过滤器可以停止事件或将其转发到该对象。事件过滤器*filterObj*通过其接收事件eventFilter（） 功能。这eventFilter如果事件应该被过滤（即停止），() 函数必须返回 true；否则它必须返回 false。

如果单个对象上安装了多个事件过滤器，则首先激活最后安装的过滤器。

这是一个KeyPressEater类，它吃掉其监视对象的按键：

```
class KeyPressEater : public QObject
{
    Q_OBJECT
    ...

protected:
    bool eventFilter(QObject *obj, QEvent *event) override;
};

bool KeyPressEater::eventFilter(QObject *obj, QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        qDebug("Ate key press %d", keyEvent->key());
        return true;
    } else {
        // standard event processing
        return QObject::eventFilter(obj, event);
    }
}
```

以下是如何将其安装在两个小部件上：

```
KeyPressEater *keyPressEater = new KeyPressEater(this);
QPushButton *pushButton = new QPushButton(this);
QListView *listView = new QListView(this);

pushButton->installEventFilter(keyPressEater);
listView->installEventFilter(keyPressEater);
```

这QShortcut例如，class 使用此技术来拦截快捷键按下。

警告：如果您删除了接收器对象eventFilter()函数，一定要返回true。如果返回 false，Qt 会将事件发送到已删除的对象，程序将崩溃。

注意，过滤对象必须与该对象在同一个线程中。如果*filterObj*在不同的线程中，该函数不执行任何操作。如果其中之一*filterObj*或者在调用此函数后该对象被移动到不同的线程，事件过滤器将不会被调用，直到两个对象再次具有相同的线程亲和力（它不会被*删除*）。

也可以看看removeEventFilter(),eventFilter（），和event()。

[since 6.4]bool QObject::isQuickItemType() const

`true`如果该对象是一个则返回QQuickItem; 否则返回`false`.

调用此函数与调用 等效`inherits("QQuickItem")`，只是速度要快得多。

该功能是在 Qt 6.4 中引入的。

[protected]bool QObject::isSignalConnected(const QMetaMethod &signal) const

返回`true`如果*signal*连接到至少一个接收器，否则返回`false`。

*signal*必须是该对象的信号成员，否则行为未定义。

```
static const QMetaMethod valueChangedSignal = QMetaMethod::fromSignal(&MyObject::valueChanged);
if (isSignalConnected(valueChangedSignal)) {
    QByteArray data;
    data = get_the_value();       // expensive operation
    emit valueChanged(data);
}
```

正如上面的代码片段所示，您可以使用此函数来避免昂贵的初始化或发出无人监听的信号。但是，在多线程应用程序中，在此函数返回之后和发出信号之前，连接可能会发生变化。

警告：该函数违反了面向对象的模块化原则。特别是，不得从重写中调用此函数connectNotify（） 或者disconnectNotify()，因为它们可能会从任何线程调用。

bool QObject::isWidgetType() const

返回`true`该对象是否是一个小部件；否则返回`false`.

调用此函数与调用 等效`inherits("QWidget")`，只是速度要快得多。

bool QObject::isWindowType() const

返回`true`对象是否为窗口；否则返回`false`.

调用此函数与调用 等效`inherits("QWindow")`，只是速度要快得多。

void QObject::killTimer(int id)

使用计时器标识符杀死计时器，*id*。

定时器标识符由以下命令返回startTimer() 当定时器事件启动时。

也可以看看timerEvent （） 和startTimer()。

*[virtual]const QMetaObject *QObject::metaObject() const*

返回指向该对象的元对象的指针。

元对象包含有关继承的类的信息QObject，例如类名、超类名、属性、信号和槽。每一个QObject子类包含Q_OBJECT宏将有一个元对象。

信号/槽连接机制和属性系统需要元对象信息。这inherits() 函数也利用了元对象。

如果您没有指向实际对象实例的指针，但仍想访问类的元对象，则可以使用staticMetaObject。

例子：

```
QObject *obj = new QPushButton;
obj->metaObject()->className();             // returns "QPushButton"

QPushButton::staticMetaObject.className();  // returns "QPushButton"
```

也可以看看staticMetaObject。

*void QObject::moveToThread(QThread *targetThread)*

更改该对象及其子对象的线程关联性。如果该对象有父对象，则无法移动该对象。事件处理将在*targetThread*。

要将对象移动到主线程，请使用QApplication::instance() 检索指向当前应用程序的指针，然后使用QApplication::thread() 检索应用程序所在的线程。例如：

```
myObject->moveToThread(QApplication::instance()->thread());
```

如果*targetThread*是`nullptr`，该对象及其子对象的所有事件处理都会停止，因为它们不再与任何线程关联。

请注意，该对象的所有活动计时器都将被重置。计时器首先在当前线程中停止，然后在下一个线程中重新启动（以相同的间隔）。*targetThread*。因此，在线程之间不断移动对象可能会无限期地推迟计时器事件。

AQEvent::ThreadChange事件在线程关联性更改之前发送到此对象。您可以处理此事件以执行任何特殊处理。请注意，发布到该对象的任何新事件都将在*targetThread*，前提是它不是nullptr：当它是时nullptr，该对象或其子对象不会发生任何事件处理，因为它们不再与任何线程关联。

警告：该函数不是*线程*安全的；当前线程必须与当前线程关联性相同。换句话说，该函数只能将对象从当前线程“推”到另一个线程，而不能将对象从任意线程“拉”到当前线程。然而，这一规则有一个例外：没有线程关联的对象可以“拉”到当前线程。

也可以看看thread()。

[private signal]void QObject::objectNameChanged(const QString &objectName)

该信号在对象名称更改后发出。新对象名称传递为*objectName*。

注意：这是一个私人信号。它可以用于信号连接，但不能由用户发出。

注意：属性的通知程序信号objectName。

也可以看看QObject::objectName。

*QObject *QObject::parent() const*

返回指向父对象的指针。

也可以看看setParent（） 和children()。

*QVariant QObject::property(const char *name) const*

返回对象的值*name*财产。

如果不存在这样的属性，则返回的变体无效。

有关所有可用属性的信息通过metaObject（） 和dynamicPropertyNames()。

也可以看看setProperty(),QVariant::isValid(),metaObject（），和dynamicPropertyNames()。

*[protected]int QObject::receivers(const char *signal) const*

返回连接到的接收器的数量*signal*。

由于时隙和信号都可以用作信号的接收器，并且相同的连接可以进行多次，因此接收器的数量与该信号建立的连接的数量相同。

调用该函数时，可以使用SIGNAL()宏来传递特定信号：

```
if (receivers(SIGNAL(valueChanged(QByteArray))) > 0) {
    QByteArray data;
    get_the_value(&data);       // expensive operation
    emit valueChanged(data);
}
```

警告：该函数违反了面向对象的模块化原则。但是，当您仅在某些东西连接到信号时才需要执行昂贵的初始化时，它可能很有用。

也可以看看isSignalConnected()。

*void QObject::removeEventFilter(QObject *obj)*

删除事件过滤器对象*obj*来自这个对象。如果尚未安装此类事件过滤器，则该请求将被忽略。

当该对象被销毁时，该对象的所有事件过滤器都会自动删除。

即使在事件过滤器激活期间（即从eventFilter（） 功能）。

也可以看看installEventFilter(),eventFilter（），和event()。

*[protected]QObject *QObject::sender() const*

如果在由信号激活的槽中调用，则返回指向发送信号的对象的指针；否则返回`nullptr`。该指针仅在从该对象的线程上下文调用该函数的槽执行期间有效。

如果发送者被销毁，或者槽与发送者的信号断开连接，则该函数返回的指针将变得无效。

警告：该函数违反了面向对象的模块化原则。但是，当许多信号连接到单个插槽时，访问发送器可能会很有用。

警告：如上所述，当通过 a 调用该槽时，该函数的返回值无效Qt::DirectConnection来自与该对象线程不同的线程。请勿在此类场景中使用此功能。

也可以看看senderSignalIndex()。

[protected]int QObject::senderSignalIndex() const

返回调用当前执行槽的信号的元方法索引，该信号是由返回的类的成员sender()。如果在由信号激活的槽之外调用，则返回 -1。

对于具有默认参数的信号，此函数将始终返回包含所有参数的索引，无论使用哪个参数connect()。例如，信号`destroyed(QObject *obj = \nullptr)`将有两个不同的索引（带参数和不带参数），但此函数将始终返回带参数的索引。当使用不同参数重载信号时，这并不适用。

警告：该函数违反了面向对象的模块化原则。但是，当许多信号连接到单个插槽时，访问信号索引可能会很有用。

警告：当通过 a 调用该槽时，该函数的返回值无效Qt::DirectConnection来自与该对象线程不同的线程。请勿在此类场景中使用此功能。

也可以看看sender(),QMetaObject::indexOfSignal（），和QMetaObject::method()。

void QObject::setObjectName(const QString &name)

将对象的名称设置为*name*。

注意：属性的 Setter 函数objectName。

也可以看看objectName()。

[since 6.4]void QObject::setObjectName(QAnyStringView name)

这是一个过载功能。

注意：属性的 Setter 函数objectName。

该功能是在 Qt 6.4 中引入的。

*void QObject::setParent(QObject *parent)*

使该对象成为以下对象的子对象*parent*。

也可以看看parent（） 和children()。

*bool QObject::setProperty(const char *name, const QVariant &value)*

设置对象的值*name*财产给*value*。

如果该属性是在类中定义的Q_PROPERTY那么成功时返回 true，否则返回 false。如果未使用定义该属性Q_PROPERTY，因此未在元对象中列出，它被添加为动态属性并返回 false。

有关所有可用属性的信息通过metaObject（） 和dynamicPropertyNames()。

可以使用再次查询动态属性property() 并可以通过将属性值设置为无效来删除QVariant。更改动态属性的值会导致QDynamicPropertyChangeEvent发送到对象。

注意：以“*q*”开头的动态属性保留用于内部目的。

也可以看看property(),metaObject(),dynamicPropertyNames（），和QMetaProperty::write()。

bool QObject::signalsBlocked() const

`true`如果信号被阻塞则返回；否则返回`false`.

默认情况下，信号不会被阻止。

也可以看看blockSignals（） 和QSignalBlocker。

int QObject::startTimer(int interval, Qt::TimerType timerType = Qt::CoarseTimer)

启动计时器并返回计时器标识符，如果无法启动计时器则返回零。

计时器事件将每隔*interval*毫秒直到killTimer（） 叫做。如果*interval*为 0，则每当没有更多的窗口系统事件需要处理时，定时器事件就会发生一次。

虚拟的timerEvent() 函数被调用QTimerEvent定时器事件发生时的事件参数类。重新实现此函数以获取计时器事件。

如果多个定时器正在运行，QTimerEvent::timerId() 可用于找出哪个计时器被激活。

例子：

```
class MyObject : public QObject
{
    Q_OBJECT

public:
    MyObject(QObject *parent = nullptr);

protected:
    void timerEvent(QTimerEvent *event) override;
};

MyObject::MyObject(QObject *parent)
    : QObject(parent)
{
    startTimer(50);     // 50-millisecond timer
    startTimer(1000);   // 1-second timer
    startTimer(60000);  // 1-minute timer

    using namespace std::chrono;
    startTimer(milliseconds(50));
    startTimer(seconds(1));
    startTimer(minutes(1));

    // since C++14 we can use std::chrono::duration literals, e.g.:
    startTimer(100ms);
    startTimer(5s);
    startTimer(2min);
    startTimer(1h);
}

void MyObject::timerEvent(QTimerEvent *event)
{
```

```
    qDebug() << "Timer ID:" << event->timerId();
}
```

注意QTimer的准确性取决于底层操作系统和硬件。这*timerType*参数允许您自定义计时器的精度。看Qt::TimerType有关不同计时器类型的信息。大多数平台支持20毫秒的精度；有些提供更多。如果 Qt 无法交付所请求数量的计时器事件，它将默默地丢弃一些事件。

这QTimer类提供了带有单次定时器和定时器信号而不是事件的高级编程接口。还有一个QBasicTimer比这个更轻量级的类QTimer并且比直接使用计时器 ID 更简洁。

也可以看看timerEvent(),killTimer（），和QTimer::singleShot()。

int QObject::startTimer(std::chrono::milliseconds time, Qt::TimerType timerType = Qt::CoarseTimer)

这是一个过载功能。

启动计时器并返回计时器标识符，如果无法启动计时器则返回零。

计时器事件将每隔*time*间隔直到killTimer（） 叫做。如果*time*等于`std::chrono::duration::zero()`，那么每当没有更多的窗口系统事件需要处理时，计时器事件就会发生一次。

虚拟的timerEvent() 函数被调用QTimerEvent定时器事件发生时的事件参数类。重新实现此函数以获取计时器事件。

如果多个定时器正在运行，QTimerEvent::timerId() 可用于找出哪个计时器被激活。

例子：

```
class MyObject : public QObject
{
    Q_OBJECT

public:
    MyObject(QObject *parent = nullptr);

protected:
    void timerEvent(QTimerEvent *event) override;
};

MyObject::MyObject(QObject *parent)
    : QObject(parent)
{
    startTimer(50);     // 50-millisecond timer
    startTimer(1000);   // 1-second timer
    startTimer(60000);  // 1-minute timer

    using namespace std::chrono;
    startTimer(milliseconds(50));
    startTimer(seconds(1));
    startTimer(minutes(1));

    // since C++14 we can use std::chrono::duration literals, e.g.:
    startTimer(100ms);
    startTimer(5s);
```

```
    startTimer(2min);
    startTimer(1h);
}

void MyObject::timerEvent(QTimerEvent *event)
{
    qDebug() << "Timer ID:" << event->timerId();
}
```

注意QTimer的准确性取决于底层操作系统和硬件。这*timerType*参数允许您自定义计时器的精度。看Qt::TimerType有关不同计时器类型的信息。大多数平台支持20毫秒的精度；有些提供更多。如果 Qt 无法交付所请求数量的计时器事件，它将默默地丢弃一些事件。

这QTimer类提供了带有单次定时器和定时器信号而不是事件的高级编程接口。还有一个QBasicTimer比这个更轻量级的类QTimer并且比直接使用计时器 ID 更简洁。

也可以看看timerEvent(),killTimer（），和QTimer::singleShot()。

*QThread *QObject::thread() const*

返回对象所在的线程。

也可以看看moveToThread()。

*[virtual protected]void QObject::timerEvent(QTimerEvent *event)*

该事件处理程序可以在子类中重新实现，以接收对象的计时器事件。

QTimer为计时器功能提供更高级别的接口，以及有关计时器的更多一般信息。定时器事件被传递到*event*范围。

也可以看看startTimer(),killTimer（），和event()。

*[static]QString QObject::tr(const char **sourceText*, const char disambiguation* = nullptr, int n = -1)*

返回翻译版本*sourceText*，可选地基于*disambiguation*字符串和值*n*对于包含复数的字符串；否则返回QString::fromUtf8（*sourceText*）如果没有合适的翻译字符串可用。

例子：

```
void SpreadSheet::setupMenuBar()
{
    QMenu *fileMenu = menuBar()->addMenu(tr("&File"));
    ...
```

如果一样的话*sourceText*用于同一上下文中的不同角色，可以传入一个附加的标识字符串*disambiguation*（nullptr默认情况下）。在 Qt 4.4 及更早版本中，这是向翻译人员传递注释的首选方式。

例子：

```
MyWindow::MyWindow()
{
    QLabel *senderLabel = new QLabel(tr("Name:"));
    QLabel *recipientLabel = new QLabel(tr("Name:", "recipient"));
    ...
```

看Writing Source Code for Translation有关 Qt 翻译机制的一般详细说明，以及Disambiguate Identical Text有关消歧信息的部分。

警告：仅当在调用此方法*之前*安装了所有转换器时，此方法才是可重入的。不支持在执行翻译时安装或删除翻译器。这样做可能会导致崩溃或其他不良行为。

也可以看看QCoreApplication::translate（） 和Internationalization with Qt。

成员变量文档

const QMetaObject QObject::staticMetaObject

该变量存储类的元对象。

元对象包含有关继承的类的信息QObject，例如类名、超类名、属性、信号和槽。每个包含以下内容的类Q_OBJECT宏还将有一个元对象。

信号/槽连接机制和属性系统需要元对象信息。这inherits() 函数也利用了元对象。

如果你有一个指向对象的指针，你可以使用metaObject() 检索与该对象关联的元对象。

例子：

```
QPushButton::staticMetaObject.className();  // returns "QPushButton"

QObject *obj = new QPushButton;
obj->metaObject()->className();             // returns "QPushButton"
```

也可以看看metaObject()。

相关非成员

*template < typename T> T qobject_cast(QObject *object)*

*template < typename T> T qobject_cast(const QObject *object)*

返回给定的*object*如果对象属于 T 类型（或子类），则强制转换为 T 类型；否则返回`nullptr`. 如果*object*那么`nullptr`它也会回归`nullptr`。

类 T 必须继承（直接或间接）QObject并与Q_OBJECT宏。

类被认为继承了自身。

例子：

```
QObject *obj = new QTimer;          // QTimer inherits QObject

QTimer *timer = qobject_cast<QTimer *>(obj);
// timer == (QObject *)obj

QAbstractButton *button = qobject_cast<QAbstractButton *>(obj);
// button == nullptr
```

这qobject_cast() 函数的行为与标准 C++ 类似`dynamic_cast()`，优点是不需要 RTTI 支持并且可以跨动态库边界工作。

qobject_cast()也可以与接口结合使用；看到Plug & Paint示例以了解详细信息。

警告：如果 T 没有用Q_OBJECT宏，该函数的返回值未定义。

也可以看看QObject::inherits()。

QObjectList

同义词QList[QObject](https://doc-qt-io.translate.goog/qt-6/qobject.html?_x_tr_sl=auto&_x_tr_tl=zh-CN&_x_tr_hl=zh-CN&_x_tr_pto=wapp)*。

宏文档

QT_NO_NARROWING_CONVERSIONS_IN_CONNECT

当使用基于 PMF 的语法连接信号和槽时，定义此宏将禁用信号携带的参数与槽接受的参数之间的缩小和浮点到整数转换。

也可以看看QObject::connect。

Q_CLASSINFO(Name, Value)

该宏将额外信息与类相关联，可以使用QObject::metaObject()。Qt 仅在以下情况中有限地使用了此功能Qt D-Bus和Qt QML模块。

额外信息采用以下形式*Name*字符串和一个*Value*文字字符串。

例子：

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("Author", "Pierre Gendron")
    Q_CLASSINFO("URL", "http://www.my-organization.qc.ca")

public:
    ...
};
```

也可以看看QMetaObject::classInfo(),Using Qt D-Bus Adaptors， 和Extending QML。

Q_EMIT

当您想要将`emit`Qt Signals and Slots 与3rd party signal/slot mechanism。

宏通常在与文件中的变量`no_keywords`一起指定时使用，但即使未*指定*也可以使用。`CONFIG``.pro``no_keywords`

Q_ENUM(...)

该宏向元对象系统注册一个枚举类型。它必须放在具有以下属性的类中的枚举声明之后Q_OBJECT,Q_GADGET或者Q_GADGET_EXPORT宏。对于命名空间使用Q_ENUM_NS （） 反而。

例如：

```
class MyClass : public QObject
{
    Q_OBJECT

public:
    MyClass(QObject *parent = nullptr);
    ~MyClass();

    enum Priority { High, Low, VeryHigh, VeryLow };
    Q_ENUM(Priority)
    void setPriority(Priority priority);
    Priority priority() const;
};
```

使用 Q_ENUM 声明的枚举有其QMetaEnum已在附件中注册QMetaObject。您还可以使用QMetaEnum::fromType() 得到QMetaEnum。

已注册的枚举也会自动注册到 Qt 元类型系统，从而使它们为人所知QMetaType无需使用Q_DECLARE_METATYPE()。这将启用有用的功能；例如，如果用于QVariant，您可以将它们转换为字符串。同样，将它们传递给QDebug会打印出他们的名字。

int请注意，枚举值在元对象系统中以带符号的形式存储。通过元对象系统访问枚举时，使用有效值范围之外的值注册枚举int将导致溢出和潜在的未定义行为。例如，QML 确实通过元对象系统访问已注册的枚举。

也可以看看Qt's Property System。

Q_ENUM_NS(...)

该宏向元对象系统注册一个枚举类型。它必须放置在具有以下名称的命名空间中的枚举声明之后Q_NAMESPACE宏。它是一样的Q_ENUM但在命名空间中。

使用 Q_ENUM_NS 声明的枚举有其QMetaEnum已在附件中注册QMetaObject。您还可以使用QMetaEnum::fromType() 得到QMetaEnum。

已注册的枚举也会自动注册到 Qt 元类型系统，从而使它们为人所知QMetaType无需使用Q_DECLARE_METATYPE()。这将启用有用的功能；例如，如果用于QVariant，您可以将它们转换为字符串。同样，将它们传递给QDebug会打印出他们的名字。

int请注意，枚举值在元对象系统中以带符号的形式存储。通过元对象系统访问枚举时，使用有效值范围之外的值注册枚举int将导致溢出和潜在的未定义行为。例如，QML 确实通过元对象系统访问已注册的枚举。

也可以看看Qt's Property System。

Q_FLAG(...)

该宏注册一个flags type与元对象系统。它通常用在类定义中来声明给定枚举的值可以用作标志并使用按位 OR 运算符进行组合。对于命名空间使用Q_FLAG_NS（） 反而。

宏必须放置在枚举声明之后。标志类型的声明是使用Q_DECLARE_FLAGS() 宏。

例如，在QItemSelectionModel， 这SelectionFlags标志通过以下方式声明：

```
class QItemSelectionModel : public QObject
{
    Q_OBJECT

public:
    ...
    enum SelectionFlag {
        NoUpdate       = 0x0000,
        Clear          = 0x0001,
        Select         = 0x0002,
        Deselect       = 0x0004,
        Toggle         = 0x0008,
        Current        = 0x0010,
        Rows           = 0x0020,
        Columns        = 0x0040,
        SelectCurrent  = Select | Current,
        ToggleCurrent  = Toggle | Current,
```

```
        ClearAndSelect = Clear | Select
    };

    Q_DECLARE_FLAGS(SelectionFlags, SelectionFlag)
    Q_FLAG(SelectionFlags)
    ...
}
```

注意： Q_FLAG 宏负责向元对象系统注册各个标志值，因此无需使用Q_ENUM() 除了这个宏之外。

也可以看看Qt's Property System。

Q_FLAG_NS(...)

该宏注册一个flags type与元对象系统。它用于具有以下名称的命名空间Q_NAMESPACE宏，声明给定枚举的值可以用作标志并使用按位 OR 运算符进行组合。它是一样的Q_FLAG但在命名空间中。

宏必须放置在枚举声明之后。

注意： Q_FLAG_NS 宏负责向元对象系统注册各个标志值，因此没有必要使用Q_ENUM_NS() 除了这个宏之外。

也可以看看Qt's Property System。

Q_GADGET

Q_GADGET 宏是一个更轻量级的版本Q_OBJECT不继承自的类的宏QObject但仍然想使用一些提供的反射功能QMetaObject。就像Q_OBJECT宏，它必须出现在类定义的私有部分。

Q_GADGET 可以有Q_ENUM,Q_PROPERTY和Q_INVOKABLE，但它们不能有信号或槽。

Q_GADGET 使类成员`staticMetaObject`可用。`staticMetaObject`属于类型QMetaObject并提供对声明的枚举的访问Q_ENUM。

也可以看看Q_GADGET_EXPORT。

[since 6.3]Q_GADGET_EXPORT(EXPORT_MACRO)

Q_GADGET_EXPORT 宏的工作方式与Q_GADGET宏。然而，`staticMetaObject`可用的变量（参见Q_GADGET）与提供的一起声明*EXPORT_MACRO*预选赛。如果需要从动态库导出对象，但封闭类作为一个整体不应该这样做（例如，因为它主要由内联函数组成），那么这很有用。

例如：

```
class Point {
    Q_GADGET_EXPORT(EXPORT_MACRO)
    Q_PROPERTY(int x MEMBER x)
    Q_PROPERTY(int y MEMBER y)
    ~~~
```

该宏是在 Qt 6.3 中引入的。

也可以看看Q_GADGET和Creating Shared Libraries。

Q_INTERFACES(...)

该宏告诉 Qt 该类实现了哪些接口。这在实现插件时使用。

例子：

```
class BasicToolsPlugin : public QObject,
                         public BrushInterface,
                         public ShapeInterface,
                         public FilterInterface
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.Examples.PlugAndPaint.BrushInterface" FILE
"basictools.json")
    Q_INTERFACES(BrushInterface ShapeInterface FilterInterface)

public:
    ...
};
```

请参阅Plug & Paint Basic Tools示例以了解详细信息。

也可以看看Q_DECLARE_INTERFACE(),Q_PLUGIN_METADATA（），和How to Create Qt Plugins。

Q_INVOKABLE

将此宏应用于成员函数的声明，以允许通过元对象系统调用它们。宏写在返回类型之前，如下例所示：

```
class Window : public QWidget
{
    Q_OBJECT

public:
    Window();
    void normalMethod();
    Q_INVOKABLE void invokableMethod();
};
```

该`invokableMethod()`函数使用 Q_INVOKABLE 进行标记，使其在元对象系统中注册并允许使用以下命令调用它QMetaObject::invokeMethod()。由于`normalMethod()`函数不是以这种方式注册的，因此无法使用以下方式调用它QMetaObject::invokeMethod()。

如果一个可调用成员函数返回一个指向QObject或一个子类QObject它是从 QML 调用的，适用特殊的所有权规则。看Data Type Conversion Between QML and C++了解更多信息。

[since 6.0]Q_MOC_INCLUDE

Q_MOC_INCLUDE 宏可以在类内部或外部使用，并告诉Meta Object Compiler添加包含。

```
// Put this in your code and the generated code will include this header.
Q_MOC_INCLUDE("myheader.h")
```

如果用作属性或信号/槽参数的类型是前向声明的，这非常有用。

该宏是在 Qt 6.0 中引入的。

Q_NAMESPACE

Q_NAMESPACE 宏可用于添加QMetaObject命名空间的功能。

Q_NAMESPACE 可以有 Q_CLASSINFO,Q_ENUM_NS,Q_FLAG_NS ，但他们不能有 Q_ENUM,Q_FLAG,Q_PROPERTY,Q_INVOKABLE、信号或槽。

Q_NAMESPACE 使外部变量`staticMetaObject`可用。`staticMetaObject`属于类型QMetaObject并提供对声明的枚举的访问Q_ENUM_NS/Q_FLAG_NS。

例如：

```
namespace test {
Q_NAMESPACE
...
```

也可以看看Q_NAMESPACE_EXPORT。

Q_NAMESPACE_EXPORT(EXPORT_MACRO)

Q_NAMESPACE_EXPORT 宏可用于添加QMetaObject命名空间的功能。

它的工作原理与Q_NAMESPACE宏。`staticMetaObject`但是，在命名空间中定义的外部变量是使用提供的声明的*EXPORT_MACRO*预选赛。如果需要从动态库导出对象，这非常有用。

例如：

```
namespace test {
Q_NAMESPACE_EXPORT(EXPORT_MACRO)
...
```

也可以看看Q_NAMESPACE和Creating Shared Libraries。

Q_OBJECT

Q_OBJECT 宏必须出现在声明其自己的信号和槽或使用 Qt 元对象系统提供的其他服务的类定义的私有部分中。

例如：

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT

public:
    Counter() { m_value = 0; }

    int value() const { return m_value; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

注意：该宏要求该类是以下类的子类QObject。使用Q_GADGET或者Q_GADGET_EXPORT而不是 Q_OBJECT 来启用元对象系统对非类中枚举的支持QObject子类。

也可以看看Meta-Object System,Signals and Slots， 和Qt's Property System。

Q_PROPERTY(...)

该宏用于声明继承的类中的属性QObject。属性的行为类似于类数据成员，但它们具有可通过Meta-Object System。

```
Q_PROPERTY(type name
           (READ getFunction [WRITE setFunction] |
            MEMBER memberName [(READ getFunction | WRITE setFunction)])
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [REVISION int | REVISION(int[, int])]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [BINDABLE bindableProperty]
           [CONSTANT]
           [FINAL]
           [REQUIRED])
```

属性名称、类型和READ功能是必需的。类型可以是任何支持的类型QVariant，或者它可以是用户定义的类型。其他项目是可选的，但WRITE功能是通用的。这些属性默认为 true USER，但默认为 false。

例如：

```
Q_PROPERTY(QString title READ title WRITE setTitle USER true)
```

有关如何使用此宏的更多详细信息及其使用的更详细示例，请参阅有关的讨论Qt's Property System。

也可以看看Qt's Property System。

Q_REVISION

将此宏应用于成员函数的声明，以使用元对象系统中的修订号来标记它们。宏写在返回类型之前，如下例所示：

```
class Window : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(int normalProperty READ normalProperty)
    Q_PROPERTY(int newProperty READ newProperty REVISION(2, 1))

public:
    Window();
    int normalProperty();
    int newProperty();
public slots:
    void normalMethod();
    Q_REVISION(2, 1) void newMethod();
};
```

当使用元对象系统将对象动态公开给另一个 API 时，这非常有用，因为您可以匹配其他 API 的多个版本所期望的版本。考虑以下简化示例：

```
    Window window;
    int expectedRevision = 0;
    const QMetaObject *windowMetaObject = window.metaObject();
    for (int i=0; i < windowMetaObject->methodCount(); i++)
        if (windowMetaObject->method(i).revision() <= expectedRevision)
            exposeMethod(windowMetaObject->method(i));
    for (int i=0; i < windowMetaObject->propertyCount(); i++)
        if (windowMetaObject->property(i).revision() <= expectedRevision)
            exposeProperty(windowMetaObject->property(i));
```

使用与前面的示例相同的 Window 类，只有当预期版本为或更高版本时，newProperty 和 newMethod 才会在此代码中公开2.1。

由于如果未加标签，所有方法都被视为修订版，因此 或 的0标签无效并被忽略。`Q_REVISION(0)``Q_REVISION(0, 0)`

您可以将一两个整数参数传递给`Q_REVISION`. 如果传递一个参数，则仅表示次要版本。这意味着主要版本未指定。如果传递两个，第一个参数是主版本，第二个参数是次版本。

元对象系统本身不使用此标记。目前这仅被使用QtQml模块。

有关更通用的字符串标签，请参阅QMetaMethod::tag()

也可以看看QMetaMethod::revision()。

Q_SET_OBJECT_NAME(Object)

该宏指定*Object*这objectName“目的”。

是否并不重要*Object*是否是一个指针，宏会自己计算出来。

也可以看看QObject::objectName()。

Q_SIGNAL

这是一个附加宏，允许您将单个函数标记为信号。它非常有用，特别是当您使用不理解 `asignals`或`Q_SIGNALS`groups 的第 3 方源代码解析器时。

当您想要将`signals`Qt 信号和槽与3rd party signal/slot mechanism。

宏通常在与文件中的变量`no_keywords`一起指定时使用，但即使未*指定*也可以使用。`CONFIG``.pro``no_keywords`

Q_SIGNALS

当您想要将`signals`Qt 信号和槽与3rd party signal/slot mechanism。

宏通常在与文件中的变量`no_keywords`一起指定时使用，但即使未*指定*也可以使用。`CONFIG``.pro``no_keywords`

Q_SLOT

这是一个附加宏，允许您将单个函数标记为槽。它非常有用，特别是当您使用不理解 `aslots`或`Q_SLOTS`groups 的第 3 方源代码解析器时。

当您想要将`slots`Qt 信号和槽与3rd party signal/slot mechanism。

宏通常在与文件中的变量`no_keywords`一起指定时使用，但即使未*指定*也可以使用。`CONFIG``.pro``no_keywords`

Q_SLOTS

当您想要将`slots`Qt 信号和槽与3rd party signal/slot mechanism。

宏通常在与文件中的变量`no_keywords`一起指定时使用，但即使未*指定*也可以使用。`CONFIG``.pro``no_keywords`