QCoreApplication Class

QCoreApplication 类为没有 UI 的 Qt 应用程序提供事件循环。更多的...

Header:	**#include < QCoreApplication>**
CMake:	find_package(Qt6 REQUIRED COMPONENTS Core) target_link_libraries(mytarget PRIVATE Qt6::Core)
qmake:	QT += core
Inherits:	QObject
Inherited By:	QAndroidService and QGuiApplication

- 所有成员的列表，包括继承的成员

特性

applicationName : QString **applicationVersion : QString** **organizationDomain : QString**	**organizationName : QString** **quitLockEnabled : bool**

public方法

	QCoreApplication(int &*argc*, char ***argv*)
virtual	**~QCoreApplication**()
Qt::PermissionStatus	**checkPermission**(const QPermission &*permission*)
void	**installNativeEventFilter**(QAbstractNativeEventFilter **filterObj*)
virtual bool	**notify**(QObject **receiver*, QEvent** event*)
void	**removeNativeEventFilter**(QAbstractNativeEventFilter **filterObject*)
void	**requestPermission**(const QPermission &*permission*, const QObject **context*, Functor *functor*)

公共槽

void	**exit(int *returnCode* = 0)**
void	**quit**()

信号

void	**aboutToQuit()**
void	**applicationNameChanged**()
void	**applicationVersionChanged**()
void	**organizationDomainChanged**()
void	**organizationNameChanged**()

静态公共成员

void	**addLibraryPath**(**const QString &*path***)
QString	**applicationDirPath**()
QString	**applicationFilePath**()
QString	**applicationName**()
qint64	**applicationPid**()
QString	**applicationVersion**()
QStringList	**arguments**()
bool	**closingDown**()
QAbstractEventDispatcher *	**eventDispatcher**()
int	**exec**()
bool	**installTranslator**(QTranslator **translationFile*)
QCoreApplication *	**instance**()
bool	**isQuitLockEnabled**()
bool	**isSetuidAllowed**()
QStringList	**libraryPaths**()
QString	**organizationDomain**()
QString	**organizationName**()
void	**postEvent**(QObject **receiver*, QEvent** event*, int *priority* = Qt::NormalEventPriority)
void	**processEvents**(QEventLoop::ProcessEventsFlags *flags* = QEventLoop::AllEvents)
void	**processEvents**(QEventLoop::ProcessEventsFlags *flags*, int *ms*)

void	**addLibraryPath**(**const QString &*path***)
void	**removeLibraryPath**(const QString &*path*)
void	**removePostedEvents**(QObject **receiver*, int *eventType* = 0)
bool	**removeTranslator**(QTranslator **translationFile*)
bool	**sendEvent**(QObject **receiver*, **QEvent** event*)
void	**sendPostedEvents**(QObject **receiver* = nullptr, int *event_type* = 0)
void	**setApplicationName**(const QString &*application*)
void	**setApplicationVersion**(const QString &*version*)
void	**setAttribute**(Qt::ApplicationAttribute *attribute*, bool *on* = true)
void	**setEventDispatcher**(QAbstractEventDispatcher **eventDispatcher*)
void	**setLibraryPaths**(const QStringList &*paths*)
void	**setOrganizationDomain**(const QString &*orgDomain*)
void	**setOrganizationName**(const QString &*orgName*)
void	**setQuitLockEnabled**(bool *enabled*)
void	**setSetuidAllowed**(bool *allow*)
bool	**startingUp**()
bool	**testAttribute**(Qt::ApplicationAttribute *attribute*)
QString	**translate**(const char **context*, **const char** sourceText, *const char* ***disambiguation* = nullptr, int *n* = -1)

被重载的私有保护继承方法

virtual bool	**event**(**QEvent** **e*) **override**

相关非成员

void	**qAddPostRoutine**(**QtCleanUpFunction** *ptr*)
void	**qRemovePostRoutine**(QtCleanUpFunction *ptr*)

宏

Q_COREAPP_STARTUP_FUNCTION(**QtStartUpFunction** *ptr*)

Q_DECLARE_TR_FUNCTIONS(*context*)

详细说明

非 GUI 应用程序使用此类来提供事件循环。对于使用 Qt 的非 GUI 应用程序，应该只有一个 QCoreApplication 对象。对于 GUI 应用程序，请参阅QGuiApplication。对于使用 Qt Widgets 模块的应用程序，请参阅QApplication。

QCoreApplication 包含主事件循环，其中处理和分派来自操作系统（例如计时器和网络事件）和其他源的所有事件。它还处理应用程序的初始化和完成，以及系统范围和应用程序范围的设置。

事件循环和事件处理

事件循环通过调用来启动exec()。长时间运行的操作可以调用processEvents() 保持应用程序响应能力。

一般来说，我们建议您创建一个QCoreApplication，QGuiApplication或一个QApplication`main()`尽早在函数中添加对象。exec() 在事件循环退出之前不会返回；例如，当quit（） 叫做。

还提供了几个静态便利函数。QCoreApplication 对象可从instance()。事件可以通过以下方式发送sendEvent() 或发布到事件队列postEvent()。可以使用以下命令删除待处理事件removePostedEvents() 或发送sendPostedEvents()。

该类提供了一个quit() 槽和一个aboutToQuit（） 信号。

应用程序和库路径

一个应用程序有一个applicationDirPath（） 和applicationFilePath()。库路径（参见QLibrary) 可以通过以下方式检索libraryPaths() 并由setLibraryPaths(),addLibraryPath（），和removeLibraryPath()。

国际化和翻译

可以使用以下命令添加或删除翻译文件installTranslator（） 和removeTranslator()。可以使用以下方式翻译应用程序字符串translate()。这QObject::tr() 函数的实现方式为translate()。

访问命令行参数

传递给 QCoreApplication 构造函数的命令行参数应该使用arguments（） 功能。

注意： QCoreApplication 删除了选项`-qmljsdebugger="..."`。它解析 的参数`qmljsdebugger`，然后删除该选项及其参数。

对于更高级的命令行选项处理，创建QCommandLineParser。

区域设置

在 Unix/Linux 上，Qt 默认配置为使用系统区域设置。这可能会在使用 POSIX 函数时导致冲突，例如在浮点数和字符串等数据类型之间进行转换时，因为不同语言环境之间的表示法可能有所不同。要解决此问题，请`setlocale(LC_NUMERIC,"C")`在初始化后立即调用 POSIX 函数QApplication,QGuiApplication或 QCoreApplication 将用于数字格式化的区域设置重置为“C”区域设置。

可以参阅QGuiApplication,QAbstractEventDispatcher,QEventLoop,Semaphores Example， 和Wait Conditions Example。

财产文件

applicationName : QString

该属性保存该应用程序的名称

该值由QSettings使用空构造函数构造类时。这样就不必每次都重复此信息QSettings对象被创建。

如果未设置，应用程序名称默认为可执行文件名称（自 5.0 起）。

访问功能：

QString	**applicationName()**
void	**setApplicationName**(const QString &*application*)

通知器信号：

void	**applicationNameChanged()**

可以参阅organizationName,organizationDomain,applicationVersion， 和applicationFilePath()。

applicationVersion : QString

该属性保存该应用程序的版本

如果未设置，应用程序版本默认为从主应用程序可执行文件或包确定的特定于平台的值（自 Qt 5.9 起）：

平台	来源
Windows（经典桌面）	VERSIONINFO 资源的 PRODUCTVERSION 参数
macOS、iOS、tvOS、watchOS	信息属性列表的 CFBundleVersion 属性
安卓	AndroidManifest.xml 清单元素的 android:versionName 属性

在其他平台上，默认值为空字符串。

访问功能：

QString	applicationVersion()
void	**setApplicationVersion**(const QString &*version*)

通知器信号：

void	applicationVersionChanged()

可以参阅applicationName,organizationName， 和organizationDomain。

organizationDomain : QString

该属性拥有编写该应用程序的组织的 Internet 域

该值由QSettings使用空构造函数构造类时。这样就不必每次都重复此信息QSettings对象被创建。

在Mac上， QSettings如果organizationDomain()不是空字符串，则使用organizationDomain()作为组织；否则它使用organizationName()。在所有其他平台上， QSettings用途organizationName()作为组织。

访问功能：

QString	organizationDomain()
void	**setOrganizationDomain**(const QString &*orgDomain*)

通知器信号：

void	organizationDomainChanged()

可以参阅organizationName,applicationName， 和applicationVersion。

organizationName : QString

该属性保存编写该应用程序的组织的名称

该值由QSettings使用空构造函数构造类时。这样就不必每次都重复此信息QSettings对象被创建。

在Mac上， QSettings用途organizationDomain如果不是空字符串，则将 () 作为组织；否则它使用organizationName()。在所有其他平台上， QSettings使用organizationName()作为组织。

访问功能：

QString	organizationName()
void	**setOrganizationName**(const QString &*orgName*)

通知器信号：

void	**organizationNameChanged()**

可以参阅organizationDomain和applicationName。

quitLockEnabled : bool

该属性保存是否使用QEventLoopLocker功能可能会导致应用程序退出。

默认为`true`.

访问功能：

bool	**isQuitLockEnabled()**
void	**setQuitLockEnabled**(bool *enabled*)

可以参阅QEventLoopLocker。

成员函数文档

*QCoreApplication::QCoreApplication(int &argc, char **argv)*

构建 Qt 核心应用程序。核心应用程序是没有图形用户界面的应用程序。此类应用程序在控制台上使用或作为服务器进程使用。

这*argc*和*argv*参数由应用程序处理，并由应用程序以更方便的形式提供arguments（） 功能。

警告：所引用的数据*argc*和*argv*必须在 QCoreApplication 对象的整个生命周期内保持有效。此外，*argc*必须大于零并且*argv*必须至少包含一个有效的字符串。

[virtual]QCoreApplication::~QCoreApplication()

摧毁了QCoreApplication目的。

[private signal]void QCoreApplication::aboutToQuit()

当应用程序即将退出主事件循环时（例如，当事件循环级别降至零时），将发出此信号。这可能发生在调用quit()从应用程序内部或当用户关闭整个桌面会话时。

如果您的应用程序必须进行最后一秒的清理，则该信号特别有用。请注意，在此状态下无法进行用户交互。

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。

可以参阅quit()。

[static]void QCoreApplication::addLibraryPath(const QString &path)

前置*path*到库路径列表的开头，确保首先搜索库。如果*path*为空或已在路径列表中，则路径列表不会更改。

默认路径列表由一个或两个条目组成。第一个是插件的安装目录，即`INSTALL/pluginsQtINSTALL`的安装目录。第二个是应用程序自己的目录（**不是**当前目录），但仅在QCoreApplication对象被实例化。

当实例化时，库路径将重置为默认值QCoreApplication被破坏了。

可以参阅removeLibraryPath(),libraryPaths（），和setLibraryPaths()。

[static]QString QCoreApplication::applicationDirPath()

返回包含应用程序可执行文件的目录。

例如，如果您已将 Qt 安装在该`C:\Qt`目录中，并且运行`regexp`示例，则此函数将返回“C:/Qt/examples/tools/regexp”。

在 macOS 和 iOS 上，这将指向实际包含可执行文件的目录，该目录可能位于应用程序包内（如果应用程序已捆绑）。

警告：在 Linux 上，此函数将尝试从文件系统获取路径`/proc`。如果失败，则假定`argv[0]`包含可执行文件的绝对文件名。该函数还假设当前目录尚未被应用程序更改。

可以参阅applicationFilePath()。

[static]QString QCoreApplication::applicationFilePath()

返回应用程序可执行文件的文件路径。

例如，如果您已在该`/usr/local/qt`目录中安装了 Qt，并且运行了`regexp`示例，则该函数将返回“/usr/local/qt/examples/tools/regexp/regexp”。

警告：在 Linux 上，此函数将尝试从文件系统获取路径`/proc`。如果失败，则假定`argv[0]`包含可执行文件的绝对文件名。该函数还假设当前目录尚未被应用程序更改。

可以参阅applicationDirPath()。

[static]qint64 QCoreApplication::applicationPid()

返回应用程序的当前进程 ID。

[static]QStringList QCoreApplication::arguments()

返回命令行参数的列表。

通常arguments().at(0)是程序名，arguments().at(1)是第一个参数，arguments().last()是最后一个参数。请参阅下面有关 Windows 的注释。

调用此函数很慢 - 在解析命令行时您应该将结果存储在变量中。

警告：在 Unix 上，此列表是根据传递给 main() 函数中的构造函数的 argc 和 argv 参数构建的。argv 中的字符串数据被解释为QString::fromLocal8Bit(); 因此，无法在以 Latin1 语言环境运行的系统上传递日语命令行参数等。大多数现代 Unix 系统没有这个限制，因为它们是基于 Unicode 的。

在 Windows 上，仅当修改后的 argv/argc 参数传递给构造函数时，才会根据 argc 和 argv 参数构建列表。在这种情况下，可能会出现编码问题。

否则，arguments() 是根据返回值构造的GetCommandLine()。因此，arguments().at(0) 给出的字符串可能不是 Windows 上的程序名称，具体取决于应用程序的启动方式。

可以参阅applicationFilePath（） 和QCommandLineParser。

[since 6.5]Qt::PermissionStatus QCoreApplication::checkPermission(const QPermission &permission)

检查给定的状态*permission*

如果结果是Qt::PermissionStatus::Undetermined那么应该通过以下方式请求许可requestPermission() 来确定用户的意图。

该功能是在 Qt 6.5 中引入的。

可以参阅requestPermission（） 和Application Permissions。

[static]bool QCoreApplication::closingDown()

`true`如果应用程序对象正在被销毁，则返回；否则返回`false`.

可以参阅startingUp()。

*[override virtual protected]bool QCoreApplication::event(QEvent *e)*

重新实现：QObject::event（QEvent *e）。

*[static]QAbstractEventDispatcher *QCoreApplication::eventDispatcher()*

返回指向主线程的事件调度程序对象的指针。如果线程不存在事件调度程序，则此函数返回`nullptr`。

可以参阅setEventDispatcher()。

[static]int QCoreApplication::exec()

进入主事件循环并等待exit（） 叫做。返回传递给的值exit()（如果是 0exit() 被调用通过quit()）。

需要调用该函数来启动事件处理。主事件循环接收来自窗口系统的事件并将这些事件分派给应用程序小部件。

要使您的应用程序执行空闲处理（通过在没有待处理事件时执行特殊函数），请使用QTimer0超时。可以使用以下方式实现更先进的空闲处理方案processEvents()。

我们建议您将清理代码连接到aboutToQuit() 信号，而不是将其放入应用程序的`main()`函数中，因为在某些平台上 exec() 调用可能不会返回。例如，在 Windows 上，当用户注销时，系统会在 Qt 关闭所有顶级窗口后终止进程。`main()`因此，不能保证应用程序在 exec() 调用之后有时间退出其事件循环并在函数末尾执行代码。

可以参阅quit(),exit(),processEvents（），和QApplication::exec()。

[static slot]void QCoreApplication::exit(int returnCode = 0)

告诉应用程序退出并返回代码。

调用此函数后，应用程序离开主事件循环并从调用返回exec()。这exec() 函数返回*returnCode*。如果事件循环未运行，则此函数不执行任何操作。

按照惯例，一个*returnCode*0 表示成功，任何非零值表示错误。

最好始终使用QueuedConnection。如果在控制进入主事件循环之前（例如在“int main”调用之前）发出连接（非排队）到此槽的信号exec()），槽没有任何作用，应用程序永远不会退出。使用排队连接可确保在控制进入主事件循环之前不会调用槽。

请注意，与同名的 C 库函数不同，该函数*确实*返回给调用者——停止的是事件处理。

另请注意，此函数不是线程安全的。它应该只从主线程（该线程QCoreApplication对象正在处理事件）。要要求应用程序从另一个线程退出，请使用QCoreApplication::quit() 或使用 QMetaMethod::invokeMethod() 从主线程调用此函数。

可以参阅quit（） 和exec()。

*void QCoreApplication::installNativeEventFilter(QAbstractNativeEventFilter *filterObj)*

安装事件过滤器*filterObj*对于应用程序在主线程中接收到的所有本机事件。

事件过滤器*filterObj*通过其接收事件nativeEventFilter() 函数，为主线程中接收到的所有本机事件调用该函数。

这QAbstractNativeEventFilter::nativeEventFilter如果事件应该被过滤，即停止，() 函数应该返回 true。它应该返回 false 以允许正常的 Qt 处理继续：然后可以将本机事件转换为QEvent并由标准 Qt 处理event过滤，例如QObject::installEventFilter()。

如果安装了多个事件过滤器，则首先激活最后安装的过滤器。

注意：此处设置的过滤器函数接收本机消息，即MSG 或XCB 事件结构。

注意：当以下情况发生时，本机事件过滤器将在应用程序中禁用：Qt::AA_PluginApplication属性已设置。

为了获得最大的可移植性，您应该始终尝试使用QEvent和QObject::installEventFilter（） 只要有可能。

可以参阅QObject::installEventFilter()。

*[static]bool QCoreApplication::installTranslator(QTranslator *translationFile)*

添加翻译文件*translationFile*到要用于翻译的翻译文件列表。

可以安装多个翻译文件。搜索翻译的顺序与安装的顺序相反，因此首先搜索最近安装的翻译文件，最后搜索第一个安装的翻译文件。一旦找到包含匹配字符串的翻译，搜索就会停止。

安装或移除QTranslator，或更改已安装的QTranslator生成一个LanguageChange活动为QCoreApplication实例。AQApplication实例会将事件传播到所有顶级小部件，其中changeEvent的重新实现可以通过通过传递用户可见的字符串来重新翻译用户界面tr() 函数到相应的属性设置器。Qt Designer 生成的用户界面类提供了`retranslateUi()`可以调用的函数。

该函数`true`在成功时返回，在失败时返回 false。

笔记：QCoreApplication不*拥有*所有权*translationFile*。

可以参阅removeTranslator(),translate(),QTranslator::load（），和Prepare for Dynamic Language Changes。

*[static]QCoreApplication *QCoreApplication::instance()*

返回指向应用程序的指针QCoreApplication（或者QGuiApplication/QApplication） 实例。

如果没有分配实例，`nullptr`则返回。

[static]bool QCoreApplication::isSetuidAllowed()

如果允许应用程序在 UNIX 平台上运行 setuid，则返回 true。

可以参阅QCoreApplication::setSetuidAllowed()。

[static]QStringList QCoreApplication::libraryPaths()

返回应用程序在动态加载库时将搜索的路径列表。

当以下情况发生时，该函数的返回值可能会改变：QCoreApplication被建造。不建议在创建之前调用它QCoreApplication。应用程序可执行文件的目录（**不是**工作目录）是列表的一部分（如果已知）。为了让大家知道QCoreApplication必须按照它用来`argv[0]`查找它的方式进行构建。

Qt 提供了默认的库路径，但也可以使用qt.conf文件。该文件中指定的路径将覆盖默认值。请注意，如果 qt.conf 文件位于应用程序可执行文件的目录中，则可能无法找到它，直到QCoreApplication被建造。如果调用该函数时没有找到，则将使用默认的库路径。

该列表将包括插件的安装目录（如果存在）（插件的默认安装目录是`INSTALL/plugins`，其中`INSTALL`是安装 Qt 的目录）。`QT_PLUGIN_PATH`始终添加以冒号分隔的环境变量条目。当应用程序可执行文件的目录已知时，插件安装目录（及其存在）可能会发生变化。

如果你想迭代列表，你可以使用foreach伪关键字：

```
foreach (const QString &path, app.libraryPaths())
    do_something(path);
```

可以参阅setLibraryPaths(),addLibraryPath(),removeLibraryPath(),QLibrary， 和How to Create Qt Plugins。

*[virtual]bool QCoreApplication::notify(QObject **receiver***, **QEvent** event*)*

发送*event*到*receiver*：*receiver->*事件(*event*) 。返回从接收者的事件处理程序返回的值。请注意，发送到任何线程中任何对象的所有事件都会调用此函数。

`false`对于某些类型的事件（例如鼠标和按键事件），如果接收者对该事件不感兴趣（即，它返回），则该事件将传播到接收者的父对象，依此类推直至顶级对象。

有五种不同的事件处理方式；重新实现这个虚函数只是其中之一。下面列出了所有五种方法：

1. 重新实现paintEvent(),mousePressEvent （） 等等。这是最常见、最简单、也是最弱的方法。
2. 重新实现这个功能。这是非常强大的，提供完全的控制；但一次只能有一个子类处于活动状态。
3. 安装事件过滤器QCoreApplication::instance()。这样的事件过滤器能够处理所有小部件的所有事件，因此它与重新实现notify()一样强大；此外，可以有多个应用程序全局事件过滤器。全局事件过滤器甚至可以看到鼠标事件disabled widgets。请注意，仅针对主线程中的对象调用应用程序事件过滤器。
4. 重新实现QObject::event （） （作为QWidget做）。如果执行此操作，您将按下 Tab 键，并且可以在任何特定于小部件的事件过滤器之前查看事件。
5. 在对象上安装事件过滤器。这样的事件过滤器获取所有事件，包括 Tab 和 Shift+Tab 按键事件，只要它们不改变焦点小部件。

未来方向： Qt 6 中主线程之外的对象不会调用此函数。需要该功能的应用程序应同时找到满足其事件检查需求的其他解决方案。该更改可能会扩展到主线程，从而导致该函数被弃用。

警告：如果重写此函数，则必须确保处理事件的所有线程在应用程序对象开始销毁之前停止执行此操作。这包括您可能正在使用的其他库启动的线程，但不适用于 Qt 自己的线程。

可以参阅QObject::event（） 和installNativeEventFilter()。

*[static]void QCoreApplication::postEvent(QObject **receiver***, **QEvent** *event*, int priority = Qt::NormalEventPriority)*

添加事件*event*，与对象*receiver*作为事件的接收者，放入事件队列并立即返回。

该事件必须在堆上分配，因为事件发布队列将获得该事件的所有权，并在事件发布后将其删除。活动发布后访问该活动并不安全。

当控制权返回到主事件循环时，存储在队列中的所有事件都将使用notify（） 功能。

事件按降序排序*priority*顺序，即事件具有高*priority*在较低的事件之前排队*priority*。这*priority*可以是任何整数值，即介于 INT_MAX 和 INT_MIN 之间（含）；看Qt::EventPriority更多细节。同等事件*priority*将按照发布的顺序进行处理。

注：该函数是thread-safe。

可以参阅sendEvent(),notify(),sendPostedEvents（），和Qt::EventPriority。

[static]void QCoreApplication::processEvents(QEventLoop::ProcessEventsFlags flags = QEventLoop::AllEvents)

根据指定的条件为调用线程处理一些挂起的事件*flags*。

不鼓励使用此功能。相反，更喜欢将长操作从 GUI 线程移至辅助线程，并完全避免嵌套事件循环处理。如果确实需要事件处理，请考虑使用QEventLoop反而。

如果您正在运行一个连续调用此函数的本地循环，而没有事件循环，则DeferredDelete事件将不会被处理。这可能会影响小部件的行为，例如QToolTip, 依赖于DeferredDelete事件才能正常运行。另一种选择是致电sendPostedEvents()从该本地循环内。

调用此函数仅为调用线程处理事件，并在处理完所有可用事件后返回。可用事件是在函数调用之前排队的事件。这意味着在函数运行时发布的事件将排队等待下一轮事件处理。

注：该函数是thread-safe。

可以参阅exec(),QTimer,QEventLoop::processEvents（），和sendPostedEvents()。

[static]void QCoreApplication::processEvents(QEventLoop::ProcessEventsFlags flags, int ms)

该函数重载了processEvents()。

处理调用线程的挂起事件*ms*毫秒或直到没有更多事件需要处理，以较短者为准。

不鼓励使用此功能。相反，更喜欢将长操作从 GUI 线程移至辅助线程，并完全避免嵌套事件循环处理。如果确实需要事件处理，请考虑使用QEventLoop反而。

调用此函数仅为调用线程处理事件。

注：与processEvents() 重载，此函数还处理函数运行时发布的事件。

注意：无论需要多长时间，在超时之前排队的所有事件都将被处理。

注：该函数是thread-safe。

可以参阅exec(),QTimer， 和QEventLoop::processEvents()。

[static slot]void QCoreApplication::quit()

要求应用程序退出。

如果应用程序阻止退出，例如，如果其窗口之一无法关闭，则该请求可能会被忽略。应用程序可以通过处理来影响这一点QEvent::Quit应用程序级别的事件，或QEvent::Close各个窗口的事件。

如果退出未被中断，应用程序将退出并返回代码 0（成功）。

要退出应用程序而不被中断，请调用exit（） 直接地。请注意，该方法不是线程安全的。

最好始终使用QueuedConnection。如果在控制进入主事件循环之前（例如在“int main”调用之前）发出连接（非排队）到此槽的信号exec())，槽没有任何作用，应用程序永远不会退出。使用排队连接可确保在控制进入主事件循环之前不会调用槽。

例子：

```
QPushButton *quitButton = new QPushButton("Quit");
connect(quitButton, &QPushButton::clicked, &app, &QCoreApplication::quit, Qt::QueuedConnection);
```

线程安全注意事项：可以从任何线程调用此函数，以线程安全的方式导致当前运行的主应用程序循环退出。然而，如果QCoreApplication对象同时被销毁。

注：该函数是thread-safe。

可以参阅exit（） 和aboutToQuit()。

[static]void QCoreApplication::removeLibraryPath(const QString &path)

删除*path*从库路径列表中。如果*path*为空或不在路径列表中，则列表不会更改。

当实例化时，库路径将重置为默认值QCoreApplication被破坏了。

可以参阅addLibraryPath(),libraryPaths（），和setLibraryPaths()。

*void QCoreApplication::removeNativeEventFilter(QAbstractNativeEventFilter *filterObject)*

删除事件*filterObject*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 nativeEventFilter() 函数），删除事件过滤器始终是安全的。

可以参阅installNativeEventFilter()。

*[static]void QCoreApplication::removePostedEvents(QObject *receiver, int eventType = 0)*

删除给定的所有事件*eventType*发布的使用postEvent（） 为了*receiver*。

事件不会被调度，而是从队列中删除。您永远不需要调用此函数。如果您确实调用它，请注意终止事件可能会导致*receiver*打破一个或多个不变量。

如果*receiver*是`nullptr`, 的事件*eventType*对于所有对象都被删除。如果*eventType*为 0，则删除所有事件*receiver*。你永远不应该用*eventType*为 0。

注：该函数是thread-safe。

*[static]bool QCoreApplication::removeTranslator(QTranslator *translationFile)*

删除翻译文件*translationFile*从此应用程序使用的翻译文件列表中。（它不会从文件系统中删除翻译文件。）

该函数`true`在成功时返回，在失败时返回 false。

可以参阅installTranslator(),translate（），和QObject::tr()。

*[since 6.5]template < typename Functor> void QCoreApplication::requestPermission(const QPermission &permission, const QObject *context, Functor functor)*

请求给定的*permission*，在上下文中*context*。

当请求准备好后，*functor*将被称为`functor(const QPermission &permission)`，并`permission`描述请求的结果。

这*functor*可以是独立的或静态的成员函数：

```
qApp->requestPermission(QCameraPermission{}, context, &permissionUpdated);
```

拉姆达：

```
qApp->requestPermission(QCameraPermission{}, context, [](const QPermission &permission) {
});
```

或中的一个插槽*context*目的：

```
qApp->requestPermission(QCameraPermission{}, this, &CamerWidget::permissionUpdated);
```

这*functor*将在线程中调用*context*目的。如果*context*在请求完成之前被销毁，*functor*不会被调用。

如果用户明确授予应用程序请求的权限*permission*， 或者*permission*已知在给定平台上不需要用户授权，状态将为Qt::PermissionStatus::Granted。

如果用户明确拒绝所请求的应用程序*permission*， 或者*permission*已知无法访问或不适用于给定平台上的应用程序，状态将为Qt::PermissionStatus::Denied。

请求的结果永远不会是Qt::PermissionStatus::Undetermined。

注意：只能从主线程请求权限。

这是一个重载功能。

该功能是在 Qt 6.5 中引入的。

可以参阅checkPermission（） 和Application Permissions。

*[static]bool QCoreApplication::sendEvent(QObject **receiver***, **QEvent** *event*)*

发送事件*event*直接发送至接收者*receiver*， 使用notify（） 功能。返回从事件处理程序返回的值。

发送事件后，该事件*不会被删除*。正常的方法是在堆栈上创建事件，例如：

```
QMouseEvent event(QEvent::MouseButtonPress, pos, 0, 0, 0);
QApplication::sendEvent(mainWindow, &event);
```

可以参阅postEvent（） 和notify()。

*[static]void QCoreApplication::sendPostedEvents(QObject *receiver = nullptr, int event_type = 0)*

立即调度之前排队的所有事件QCoreApplication::postEvent() 和哪些是针对对象的*receiver*并具有事件类型*event_type*。

来自窗口系统的事件*不是*由该函数调度的，而是由processEvents()。

如果*receiver*是nullptr, 的事件*event_type*为所有对象发送。如果*event_type*为 0，则发送所有事件*receiver*。

注意：该方法必须在其所在线程中调用QObject范围，*receiver*，生活。

可以参阅postEvent()。

[static]void QCoreApplication::setAttribute(Qt::ApplicationAttribute attribute, bool on = true)

设置属性*attribute*如果*on*是真的; 否则清除该属性。

注意：在创建应用程序**之前**必须设置一些应用程序属性QCoreApplication实例。请参阅Qt::ApplicationAttribute文档以获取更多信息。

可以参阅testAttribute()。

*[static]void QCoreApplication::setEventDispatcher(QAbstractEventDispatcher *eventDispatcher)*

将主线程的事件调度程序设置为*eventDispatcher*。仅当尚未安装事件调度程序时，这才可能实现。也就是说，之前QCoreApplication已被实例化。该方法获取对象的所有权。

可以参阅eventDispatcher()。

[static]void QCoreApplication::setLibraryPaths(const QStringList &paths)

设置加载插件时要搜索的目录列表QLibrary到*paths*。所有现有路径将被删除，路径列表将包含中给出的路径*paths*以及应用程序的路径。

当实例化时，库路径将重置为默认值QCoreApplication被破坏了。

可以参阅libraryPaths(),addLibraryPath(),removeLibraryPath（），和QLibrary。

[static]void QCoreApplication::setSetuidAllowed(bool allow)

允许应用程序在 UNIX 平台上运行 setuid，如果*allow*是真的。

如果*allow*为 false（默认值）并且 Qt 检测到应用程序正在使用与真实用户 ID 不同的有效用户 ID 运行，当QCoreApplication实例已创建。

由于 Qt 的攻击面较大，所以它不是 setuid 程序的合适解决方案。然而，由于历史原因，某些应用程序可能需要以这种方式运行。该标志将防止 Qt 在检测到此情况时中止应用程序，并且必须在QCoreApplication实例已创建。

注意：强烈建议不要启用此选项，因为它会带来安全风险。

可以参阅isSetuidAllowed()。

[static]bool QCoreApplication::startingUp()

`true`如果应用程序对象尚未创建则返回；否则返回`false`.

可以参阅closingDown()。

[static]bool QCoreApplication::testAttribute(Qt::ApplicationAttribute attribute)

返回`true`if 属性*attribute*已设置；否则返回`false`.

可以参阅setAttribute()。

*[static]QString QCoreApplication::translate(const char **context*, const char *sourceText, const char **disambiguation = nullptr, int n = -1)*

返回翻译文本*sourceText*，通过查询已安装的翻译文件。翻译文件将从最近安装的文件搜索到第一个安装的文件。

QObject::tr() 更方便地提供此功能。

*context*通常是一个类名（例如“MyDialog”）并且*sourceText*是英文文本或简短的识别文本。

*disambiguation*是一个识别字符串，当相同时*sourceText*在同一上下文中用于不同的角色。默认情况下，它是`nullptr`.

请参阅QTranslator和QObject::tr() 文档，了解有关上下文、消歧和注释的更多信息。

*n*与 with 一起使用%n以支持复数形式。看QObject::tr（）了解详情。

如果没有翻译文件包含以下内容的翻译*sourceText*在*context*，这个函数返回一个QString相当于*sourceText*。

这个函数不是虚拟的。您可以通过子类化来使用替代翻译技术QTranslator。

注：该函数是thread-safe。

可以参阅QObject::tr(),installTranslator(),removeTranslator（），和Internationalization and Translations。

相关非成员

void qAddPostRoutine(QtCleanUpFunction ptr)

添加将从中调用的全局例程QCoreApplication析构函数。此函数通常用于为程序范围的功能添加清理例程。

清理例程以与添加相反的顺序调用。

指定的函数*ptr*不应该接受任何参数并且不应该返回任何内容。例如：

```
static int *global_ptr = nullptr;

static void cleanup_ptr()
{
    delete [] global_ptr;
    global_ptr = nullptr;
}

void init_ptr()
{
    global_ptr = new int[100];      // allocate data
    qAddPostRoutine(cleanup_ptr);   // delete later
}
```

请注意，对于应用程序或模块范围的清理，qAddPostRoutine() 通常不适合。例如，如果程序被分割成动态加载的模块，则相关模块可能会在加载之前很久就被卸载。QCoreApplication析构函数被调用。在这种情况下，如果仍然需要使用 qAddPostRoutine()，qRemovePostRoutine() 可用于防止例程被调用QCoreApplication析构函数。例如，如果在卸载模块之前调用该例程。

对于模块和库，使用引用计数初始化管理器或 Qt 的父子删除机制可能会更好。下面是一个私有类的例子，它使用父子机制在正确的时间调用清理函数：

```
class MyPrivateInitStuff : public QObject
{
public:
    static MyPrivateInitStuff *initStuff(QObject *parent)
    {
        if (!p)
            p = new MyPrivateInitStuff(parent);
        return p;
    }

    ~MyPrivateInitStuff()
    {
        // cleanup goes here
    }

private:
    MyPrivateInitStuff(QObject *parent)
        : QObject(parent)
    {
        // initialization goes here
    }

    MyPrivateInitStuff *p;
};
```

通过选择正确的父对象，通常可以在正确的时刻清理模块的数据。

注意：该函数从 Qt 5.10 开始是线程安全的。

注：该函数是thread-safe。

可以参阅qRemovePostRoutine()。

void qRemovePostRoutine(QtCleanUpFunction ptr)

删除指定的清理例程*ptr*从调用的例程列表中QCoreApplication析构函数。该例程必须先前已通过调用添加到列表中qAddPostRoutine()，否则该函数无效。

注意：该函数从 Qt 5.10 开始是线程安全的。

注：该函数是thread-safe。

可以参阅qAddPostRoutine()。

宏文档

Q_COREAPP_STARTUP_FUNCTION(QtStartUpFunction ptr)

添加将从中调用的全局函数QCoreApplication构造函数。该宏通常用于初始化程序范围功能的库，而不需要应用程序调用库进行初始化。

指定的函数*ptr*不应该接受任何参数并且不应该返回任何内容。例如：

```
// Called once QCoreApplication exists
static void preRoutineMyDebugTool()
{
    MyDebugTool* tool = new MyDebugTool(QCoreApplication::instance());
    QCoreApplication::instance()->installEventFilter(tool);
}

Q_COREAPP_STARTUP_FUNCTION(preRoutineMyDebugTool)
```

请注意，启动函数将在程序结束时运行QCoreApplication构造函数，在任何 GUI 初始化之前。如果函数中需要 GUI 代码，请使用计时器（或排队调用）稍后从事件循环执行初始化。

如果QCoreApplication被删除，另一个QCoreApplication创建完成后，会再次调用启动函数。

注意：此宏不适合在静态链接到应用程序的库代码中使用，因为该函数可能由于被链接器消除而根本不会被调用。

注：该宏是reentrant。

Q_DECLARE_TR_FUNCTIONS(context)

Q_DECLARE_TR_FUNCTIONS() 宏使用`tr()`以下签名声明并实现转换函数：

```
static inline QString tr(const char *sourceText,
                         const char *comment = nullptr);
```

如果您想使用此宏，则很有用QObject::tr() 在不继承自的类中QObject。

Q_DECLARE_TR_FUNCTIONS() 必须出现在类定义的最顶部（在第一个`public:`或之前`protected:`）。例如：

```
class MyMfcView : public CView
{
    Q_DECLARE_TR_FUNCTIONS(MyMfcView)

public:
    MyMfcView();
    ...
};
```

这*context*参数通常是类名，但也可以是任何文本。

可以参阅Q_OBJECT和QObject::tr()。