Qt 事件处理机制 （上篇）

在Qt中，事件被封装成一个个对象，所有的事件均继承自抽象类QEvent. 接下来依次谈谈Qt中有谁来产生、分发、接受和处理事件。

作者：佚名 来源：互联网 | 2011-07-01 14:14 收藏 分享

本篇来介绍**Qt 事件**处理机制 。深入了解**事件**处理系统对于每个学习**Qt**人来说非常重要，可以说，**Qt**是以**事件**驱动的UI工具集。 大家熟知Signals/Slots在多线程的实现也依赖于**Qt**的**事件**处理机制。

在**Qt**中，**事件**被封装成一个个对象，所有的**事件**均继承自抽象类QEvent. 接下来依次谈谈**Qt**中有谁来产生、分发、接受和处理**事件**：

1、谁来产生事件： 最容易想到的是我们的输入设备，比如键盘、鼠标产生的

keyPressEvent，keyReleaseEvent，mousePressEvent，mouseReleaseEvent事件（他们被封装成QMouseEvent和QKeyEvent），这些事件来自于底层的操作系统，它们以异步的形式通知Qt事件处理系统，后文会仔细道来。当然Qt自己也会产生很多事件，比如QObject::startTimer()会触发QTimerEvent. 用户的程序可还以自己定制事件。

2、谁来接受和处理事件：答案是QObject。在Qt的内省机制剖析一文已经介绍QObject 类是整个Qt对象模型的心脏，事件处理机制是QObject三大职责（内存管理、内省(intropection)与事件处理制）之一。任何一个想要接受并处理事件的对象均须继承自QObject，可以选择重载QObject::event()函数或事件的处理权转给父类。

3、谁来负责分发事件：对于non-GUI的Qt程序，是由QCoreApplication负责将QEvent分发给QObject的子类-Receiver. 对于Qt GUI程序，由QApplication来负责。

接下来，将通过对代码的解析来看看QT是利用event loop从事件队列中获取用户输入事件，又是如何将事件转义成QEvents，并分发给相应的QObject处理。

```
#include <QApplication>
#include "widget.h"
//Section 1
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Widget window; // Widget 继承自QWidget
    window.show();
    return app.exec(); // 进入Qpplication事件循环，见section 2
}
// Section 2:
int QApplication::exec()
{
   //skip codes
   //简单的交给QCoreApplication来处理事件循环=〉section 3
   return QCoreApplication::exec();
}
// Section 3
int QCoreApplication::exec()
```


编辑推荐

头条 沉浸式故事：AR和VR如何改变2021年的市场营销?

热点 iOS版微信8.0.3更新了什么 朋友圈和自定义表情突破限制

热点 微信一口气更新了 12 个功能

聚焦 合理规划：如何为APP选择正确的数据库?

关注 为什么手机厂商越来越少做白色前面板的手机呢?

24H热文 一周话题 本月获赞

- iOS 14.5新功能大盘点：个个都是绝技
- 从安卓设备转移数据到iOS的几种方法，掌握...
- iPhone 12价格走势：2个月降10%，5个月...
- 推荐两款iOS端磁力下载工具
- 轻松找到微信接收文件存储位置
- 快捷指令怎么用？玩转iOS14快捷指令全攻略
- 一招搞定手机和电脑的多屏协同
- 合理规划：如何为APP选择正确的数据库?

订阅专栏 +更多

16招轻松掌握PPT技巧

GET职场加薪技能

共16章 | 晒书包

289人订阅学习

```
{
    //得到当前Thread数据
    QThreadData *threadData = self->d_func()->threadData;
    if (threadData != QThreadData::current()) {
        qWarning("%s::exec: Must be called from the main thread", self->metaObject()->classNam
        return -1;
    }
    //检查event loop是否已经创建
    if (!threadData->eventLoops.isEmpty()) {
        qWarning("QCoreApplication::exec: The event loop is already running");
        return -1;
    }
    ...
    QEventLoop eventLoop;
    self->d_func()->in_exec = true;
    self->d_func()->aboutToQuitEmitted = false;
    //委任QEventLoop 处理事件队列循环 ==> Section 4
    int returnCode = eventLoop.exec();
    ....
    }
    return returnCode;
}
// Section 4
int QEventLoop::exec(ProcessEventsFlags flags)
{
   //这里的实现代码不少，最为重要的是以下几行
   Q_D(QEventLoop); // 访问QEventloop私有类实例d
        try {
        //只要没有遇见exit，循环派发事件
        while (!d->exit)
            processEvents(flags | WaitForMoreEvents | EventLoopExec);
    } catch (...) {}
}
// Section 5
bool QEventLoop::processEvents(ProcessEventsFlags flags)
{
    Q_D(QEventLoop);
    if (!d->threadData->eventDispatcher)
        return false;
    if (flags & DeferredDeletion)
        QCoreApplication::sendPostedEvents(0, QEvent::DeferredDelete);
    //将事件派发给与平台相关的QAbstractEventDispatcher子类 =>Section 6
    return d->threadData->eventDispatcher->processEvents(flags);
}
#include <QApplication>
#include "widget.h"
//Section 1
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Widget window;  // Widget 继承自QWidget
    window.show();
    return app.exec(); // 进入Qpplication事件循环，见section 2
}
// Section 2:
int QApplication::exec()
{
   //skip codes
   //简单的交给QCoreApplication来处理事件循环=〉section 3
   return QCoreApplication::exec();
}
// Section 3
int QCoreApplication::exec()
{
    //得到当前Thread数据
    QThreadData *threadData = self->d_func()->threadData;
```

```
    if (threadData != QThreadData::current()) {
        qWarning("%s::exec: Must be called from the main thread", self->metaObject()->classNam
        return -1;
    }
    //检查event loop是否已经创建
    if (!threadData->eventLoops.isEmpty()) {
        qWarning("QCoreApplication::exec: The event loop is already running");
        return -1;
    }
    ...
    QEventLoop eventLoop;
    self->d_func()->in_exec = true;
    self->d_func()->aboutToQuitEmitted = false;
    //委任QEventLoop 处理事件队列循环 ==> Section 4
    int returnCode = eventLoop.exec();
    ....
    }
    return returnCode;
}
// Section 4
int QEventLoop::exec(ProcessEventsFlags flags)
{
   //这里的实现代码不少，最为重要的是以下几行
   Q_D(QEventLoop); // 访问QEventloop私有类实例d
        try {
        //只要没有遇见exit，循环派发事件
        while (!d->exit)
            processEvents(flags | WaitForMoreEvents | EventLoopExec);
    } catch (...) {}
}
// Section 5
bool QEventLoop::processEvents(ProcessEventsFlags flags)
{
    Q_D(QEventLoop);
    if (!d->threadData->eventDispatcher)
        return false;
    if (flags & DeferredDeletion)
        QCoreApplication::sendPostedEvents(0, QEvent::DeferredDelete);
    //将事件派发给与平台相关的QAbstractEventDispatcher子类 =>Section 6
    return d->threadData->eventDispatcher->processEvents(flags);
}

// Section 6，QTDIR\src\corelib\kernel\qeventdispatcher_win.cpp
// 这段代码是完成与windows平台相关的windows c++。 以跨平台著称的Qt同时也提供了对Symi
// 其事现分别封装在QEventDispatcherSymbian和QEventDispatcherUNIX
// QEventDispatcherWin32派生自QAbstractEventDispatcher.
bool QEventDispatcherWin32::processEvents(QEventLoop::ProcessEventsFlags flags)
{
    Q_D(QEventDispatcherWin32);
    if (!d->internalHwnd)
        createInternalHwnd();
    d->interrupt = false;
    emit awake();
    bool canWait;
    bool retVal = false;
    bool seenWM_QT_SENDPOSTEDEVENTS = false;
    bool needWM_QT_SENDPOSTEDEVENTS = false;
    do {
        DWORD waitRet = 0;
        HANDLE pHandles[MAXIMUM_WAIT_OBJECTS - 1];
        QVarLengthArray<MSG> processedTimers;
        while (!d->interrupt) {
            DWORD nCount = d->winEventNotifierList.count();
            Q_ASSERT(nCount < MAXIMUM_WAIT_OBJECTS - 1);
            MSG msg;
            bool haveMessage;
```

```
        if (!(flags & QEventLoop::ExcludeUserInputEvents) && !d->queuedUserInputEvents.isE
            // process queued user input events
            haveMessage = true;
            //从处理用户输入队列中取出一条事件
            msg = d->queuedUserInputEvents.takeFirst();
        } else if(!(flags & QEventLoop::ExcludeSocketNotifiers) && !d->queuedSocketEvents.isE
            // 从处理socket队列中取出一条事件
            haveMessage = true;
            msg = d->queuedSocketEvents.takeFirst();
        } else {
            haveMessage = PeekMessage(&msg, 0, 0, 0, PM_REMOVE);
            if (haveMessage && (flags & QEventLoop::ExcludeUserInputEvents)
                && ((msg.message >= WM_KEYFIRST
                    && msg.message <= WM_KEYLAST)
                || (msg.message >= WM_MOUSEFIRST
                    && msg.message <= WM_MOUSELAST)
                || msg.message == WM_MOUSEWHEEL
                || msg.message == WM_MOUSEHWHEEL
                || msg.message == WM_TOUCH
#ifndef QT_NO_GESTURES
                || msg.message == WM_GESTURE
                || msg.message == WM_GESTURENOTIFY
#endif
                || msg.message == WM_CLOSE)) {
                // 用户输入事件入队列，待以后处理
                haveMessage = false;
                d->queuedUserInputEvents.append(msg);
            }
            if (haveMessage && (flags & QEventLoop::ExcludeSocketNotifiers)
                && (msg.message == WM_QT_SOCKETNOTIFIER && msg.hwnd == d->internalH
                // socket 事件入队列，待以后处理
                haveMessage = false;
                d->queuedSocketEvents.append(msg);
            }
        }
        ....
            if (!filterEvent(&msg)) {
                TranslateMessage(&msg);
                //将事件打包成message调用Windows API派发出去
                  //分发一个消息给窗口程序。消息被分发到回调函数，将消息传递给windows系统
              DispatchMessage(&msg);
            }
        }
    }
  } while (canWait);
    ...
  return retVal;
}
// Section 6，QTDIR\src\corelib\kernel\qeventdispatcher_win.cpp
// 这段代码是完成与windows平台相关的windows c++。 以跨平台著称的Qt同时也提供了对Symi
// 其事现分别封装在QEventDispatcherSymbian和QEventDispatcherUNIX
// QEventDispatcherWin32派生自QAbstractEventDispatcher.
bool QEventDispatcherWin32::processEvents(QEventLoop::ProcessEventsFlags flags)
{
  Q_D(QEventDispatcherWin32);
  if (!d->internalHwnd)
    createInternalHwnd();
  d->interrupt = false;
  emit awake();
  bool canWait;
  bool retVal = false;
  bool seenWM_QT_SENDPOSTEDEVENTS = false;
  bool needWM_QT_SENDPOSTEDEVENTS = false;
  do {
    DWORD waitRet = 0;
    HANDLE pHandles[MAXIMUM_WAIT_OBJECTS - 1];
```

```
218.      QVarLengthArray<MSG> processedTimers;
219.      while (!d->interrupt) {
220.          DWORD nCount = d->winEventNotifierList.count();
221.          Q_ASSERT(nCount < MAXIMUM_WAIT_OBJECTS - 1);
222.          MSG msg;
223.          bool haveMessage;
224.          if (!(flags & QEventLoop::ExcludeUserInputEvents) && !d->queuedUserInputEvents.isE
225.              // process queued user input events
226.              haveMessage = true;
227.              //从处理用户输入队列中取出一条事件
228.              msg = d->queuedUserInputEvents.takeFirst();
229.          } else if(!(flags & QEventLoop::ExcludeSocketNotifiers) && !d->queuedSocketEvents.isE
230.              // 从处理socket队列中取出一条事件
231.              haveMessage = true;
232.              msg = d->queuedSocketEvents.takeFirst();
233.          } else {
234.              haveMessage = PeekMessage(&msg, 0, 0, 0, PM_REMOVE);
235.              if (haveMessage && (flags & QEventLoop::ExcludeUserInputEvents)
236.                  && ((msg.message >= WM_KEYFIRST
237.                       && msg.message <= WM_KEYLAST)
238.                      || (msg.message >= WM_MOUSEFIRST
239.                          && msg.message <= WM_MOUSELAST)
240.                      || msg.message == WM_MOUSEWHEEL
241.                      || msg.message == WM_MOUSEHWHEEL
242.                      || msg.message == WM_TOUCH
243. #ifndef QT_NO_GESTURES
244.                      || msg.message == WM_GESTURE
245.                      || msg.message == WM_GESTURENOTIFY
246. #endif
247.                      || msg.message == WM_CLOSE)) {
248.                  // 用户输入事件入队列，待以后处理
249.                  haveMessage = false;
250.                  d->queuedUserInputEvents.append(msg);
251.              }
252.              if (haveMessage && (flags & QEventLoop::ExcludeSocketNotifiers)
253.                  && (msg.message == WM_QT_SOCKETNOTIFIER && msg.hwnd == d->internalH
254.                  // socket 事件入队列，待以后处理
255.                  haveMessage = false;
256.                  d->queuedSocketEvents.append(msg);
257.              }
258.          }
259.          ....
260.              if (!filterEvent(&msg)) {
261.                  TranslateMessage(&msg);
262.                  //将事件打包成message调用Windows API派发出去
263.                    //分发一个消息给窗口程序。消息被分发到回调函数，将消息传递给windows系统
264.                DispatchMessage(&msg);
265.              }
266.          }
267.      }
268.    } while (canWait);
269.      ...
270.    return retVal;
271. }
272.
273. // Section 7 windows窗口回调函数 定义在QTDIR\src\gui\kernel\qapplication_win.cpp
274. extern "C" LRESULT QT_WIN_CALLBACK QtWndProc(HWND hwnd, UINT message, WPARA
275. {
276.    ...
277.    //将消息重新封装成QEvent的子类QMouseEvent ==> Section 8
278.     result = widget->translateMouseEvent(msg);
279.    ...
280. }
281.
282. // Section 7 windows窗口回调函数 定义在QTDIR\src\gui\kernel\qapplication_win.cpp
283. extern "C" LRESULT QT_WIN_CALLBACK QtWndProc(HWND hwnd, UINT message, WPARA
```

```
{
    ...
    //将消息重新封装成QEvent的子类QMouseEvent ==> Section 8
     result = widget->translateMouseEvent(msg);
    ...
}
```

从Section 1~Section7， **Qt**进入QApplication的event loop，经过层层委任，最终QEventloop的processEvent将通过与平台相关的QAbstractEventDispatcher的子类QEventDispatcherWin32获得用户的用户输入**事件**，并将其打包成message后，通过标准Windows API，把消息传递给了Windows OS，Windows OS得到通知后回调QtWndProc，至此**事件**的分发与处理完成了一半的路程。

小结：**Qt 事件**处理机制 （上篇）的内容介绍完了，在下文中，我们将进一步讨论当我们收到来在Windows的回调后，**事件**又是怎么一步步打包成QEvent并通过QApplication分发给最终**事件**的接受和处理者QObject::event.请继续看Qt 事件处理机制 （下篇）。***希望本文能帮你解决问题！

【编辑推荐】

1. Qt For Symbian截获程序前后台切换事件
2. 初学者文档 QT中窗口刷新事件
3. Qt 多线程之逐线程事件循环 下篇
4. 详解 QT 源码之 Qt 事件机制原理
5. QT源码之Qt信号槽机制与事件机制的联系
6. 详解 Qt 事件过滤器

【责任编辑：李程站 TEL：（010）68476606】