<http://blog.csdn.net/king523103/article/details/46125129>

QT文档中这样解释：

sendEvent(QObject\* receiver,QEvent\* event)

使用notify()函数直接给receiver发送事件。

postEvent(QObject\* receiver, QEvent\* event)

向事件队列中添加receiver和event。

简单说，sendEvent使用的是同步处理事件，postEvent使用的异步处理事件

## sendEvent代码分析

**[cpp]** [view plain](http://blog.csdn.net/king523103/article/details/46125129) [copy](http://blog.csdn.net/king523103/article/details/46125129)

1. **inline** **bool** QCoreApplication::sendEvent(QObject \*receiver, QEvent \*event)
2. {  **if** (event) event->spont = **false**; **return** self ? self->notifyInternal(receiver, event) : **false**; }

直接调用notifyInternal接口，注意中间设置自发消息标志位为false，同时还需要判断self是有有效（QCoreApplication是否启动）

**[cpp]** [view plain](http://blog.csdn.net/king523103/article/details/46125129) [copy](http://blog.csdn.net/king523103/article/details/46125129)

1. **bool** QCoreApplication::notifyInternal(QObject \*receiver, QEvent \*event)
2. {
3. // Make it possible for Qt Jambi and QSA to hook into events even
4. // though QApplication is subclassed...
5. **bool** result = **false**;
6. **void** \*cbdata[] = { receiver, event, &result };
7. **if** (QInternal::activateCallbacks(QInternal::EventNotifyCallback, cbdata)) {
8. **return** result;
9. }
11. // Qt enforces the rule that events can only be sent to objects in
12. // the current thread, so receiver->d\_func()->threadData is
13. // equivalent to QThreadData::current(), just without the function
14. // call overhead.
15. QObjectPrivate \*d = receiver->d\_func();
16. QThreadData \*threadData = d->threadData;
17. ++threadData->loopLevel;
19. **bool** returnValue = notify(receiver, event);
20. --threadData->loopLevel;
21. **return** returnValue;
22. }

删除了一些不重要的代码，notifyInternal的主要作用是activateCallbacks，直接看notify

**[cpp]** [view plain](http://blog.csdn.net/king523103/article/details/46125129) [copy](http://blog.csdn.net/king523103/article/details/46125129)

1. **bool** QCoreApplication::notify(QObject \*receiver, QEvent \*event)
2. {
3. Q\_D(QCoreApplication);
4. // no events are delivered after ~QCoreApplication() has started
5. **if** (QCoreApplicationPrivate::is\_app\_closing)
6. **return** **true**;
7. **if** (receiver == 0) {                        // serious error
8. qWarning("QCoreApplication::notify: Unexpected null receiver");
9. **return** **true**;
10. }
11. **return** receiver->isWidgetType() ? **false** : d->notify\_helper(receiver, event);
12. }

这个接口在QT文档上有注释。注意其中当receiver为控件时，不进行处理。

**[cpp]** [view plain](http://blog.csdn.net/king523103/article/details/46125129) [copy](http://blog.csdn.net/king523103/article/details/46125129)

1. **bool** QCoreApplicationPrivate::notify\_helper(QObject \*receiver, QEvent \* event)
2. {
3. // send to all application event filters
4. **if** (sendThroughApplicationEventFilters(receiver, event))
5. **return** **true**;
6. // send to all receiver event filters
7. **if** (sendThroughObjectEventFilters(receiver, event))
8. **return** **true**;
9. // deliver the event
10. **return** receiver->event(event);
11. }

在这里可以看到事件是如何处理的：

* 先送入Application的事件过滤器，看看是否在事件过滤器中处理
* 再查看receiver是否有此事件的过滤器
* 最后，将事件送入receiver的event接口。

从整个过程来看，可以认为sendEvent直接调用了receiver的event接口。因此，可以认为处理方式为同步处理方式。

## PostEvent分析

**[cpp]** [view plain](http://blog.csdn.net/king523103/article/details/46125129) [copy](http://blog.csdn.net/king523103/article/details/46125129)

1. **void** QCoreApplication::postEvent(QObject \*receiver, QEvent \*event)
2. {
3. postEvent(receiver, event, Qt::NormalEventPriority);
4. }

**[cpp]** [view plain](http://blog.csdn.net/king523103/article/details/46125129) [copy](http://blog.csdn.net/king523103/article/details/46125129)

1. **void** QCoreApplication::postEvent(QObject \*receiver, QEvent \*event, **int** priority)
2. {
3. ...
4. QThreadData \* **volatile** \* pdata = &receiver->d\_func()->threadData;  //得到线程信息
5. QThreadData \*data = \*pdata;
6. **if** (!data) {
7. // posting during destruction? just delete the event to prevent a leak
8. **delete** event;
9. **return**;
10. }
12. // lock the post event mutex
13. data->postEventList.mutex.lock();
15. // if object has moved to another thread, follow it
16. **while** (data != \*pdata) {                        <span style="font-family: Arial, Helvetica, sans-serif;">//在这里判断receiver线程信息是否发生变化。（有可能是另外一个线程调用用receiver->moveToThread）</span>
17. data->postEventList.mutex.unlock();
19. data = \*pdata;
20. **if** (!data) {
21. // posting during destruction? just delete the event to prevent a leak
22. **delete** event;
23. **return**;
24. }
26. data->postEventList.mutex.lock();
27. }
28. //这里postEventList还是被锁着的。
29. // if this is one of the compressible events, do compression
30. **if** (receiver->d\_func()->postedEvents
31. && self && self->compressEvent(event, receiver, &data->postEventList)) {
32. data->postEventList.mutex.unlock();//这个事件有可能被压缩（实际上是发现队列中有这个事件还没有被处理，且这个事件是可以被压缩的，例如paintevent）
33. **return**;
34. }
36. event->posted = **true**;
37. ++receiver->d\_func()->postedEvents;
38. **if** (event->type() == QEvent::DeferredDelete && data == QThreadData::current()) {
39. // remember the current running eventloop for DeferredDelete
40. // events posted in the receiver's thread
41. event->d = **reinterpret\_cast**<QEventPrivate \*>(quintptr(data->loopLevel)); //receiver即将被析构？
42. }
43. //将事件添加到postEventList中，注意这里的优先级第一个最高，最后一个优先级最低
44. **if** (data->postEventList.isEmpty() || data->postEventList.last().priority >= priority) {
45. // optimization: we can simply append if the last event in
46. // the queue has higher or equal priority
47. data->postEventList.append(QPostEvent(receiver, event, priority));
48. } **else** {
49. // insert event in descending priority order, using upper
50. // bound for a given priority (to ensure proper ordering
51. // of events with the same priority)
52. QPostEventList::iterator begin = data->postEventList.begin()
53. + data->postEventList.insertionOffset,
54. end = data->postEventList.end();
55. QPostEventList::iterator at = qUpperBound(begin, end, priority);
56. data->postEventList.insert(at, QPostEvent(receiver, event, priority));
57. }
58. data->canWait = **false**;
59. data->postEventList.mutex.unlock();//在这里解除锁
60. //receiver所在的线程调用eventDispatcher处理postEventList
61. **if** (data->eventDispatcher)
62. data->eventDispatcher->wakeUp();
63. }

从上面可以看出，postEvent实际上是将事件添加到receiver所在线程中的一个队列中，至于这个队列所在的线程什么时候处理这个事件，postEvent是无法控制的