leo115的专栏

个人资料

eten

访问： 331984次

积分： 4468

等级： BLOG 5

排名： 第2340名

原创： 105篇 转载： 121篇

译文： 0篇 评论： 66条

Qt事件机制详解

分类： Linux Qt QSS 2012-03-11 18:47 471人阅读 评论(0) 收藏 举报

qt delete 多线程 user object 通讯

1. Qt 中event()

本章内容也是关于Qt事件。或许这一章不能有一个完整的例子，因为对于事件总是感觉很抽象，还是从底层上理解一下比较好的吧！

前面说到了事件的作用，下面来看看我们如何来接收事件。回忆一下前面的代码，我们在子类中重写了事件函数，以便让这些子类按照我们的需要完成某些功能，就像下面的代码：

```
void MyLabel::mousePressEvent(QMouseEvent * event)
{
	if(event->button() == Qt::LeftButton) {
		// do something
	} else {
		QLabel::mousePressEvent(event);
	}
}
```

上面的代码和前面类似，在鼠标按下的事件中检测，如果按下的是左键，做我们的处理工作，如果不是左键，则调用父类的函数。这在某种程度上说，是把事件向上传递给父类去响应，也就是说，我们在子类中“忽略”了这个事件。

我们可以把Qt的事件传递看成链状：如果子类没有处理这个事件，就会继续向其他类传递。其实，Qt的事件对象都有一个accept()函数和ignore()函数。正如它们的名字，前者用来告诉Qt，事件处理函数“接收”了这个事件，不要再传递；后者则告诉Qt，事件处理函数“忽略”了这个事件，需要继续传递，寻找另外的接受者。在事件处理函数中，可以使用isAccepted()来查询这个事件是不是已经被接收了。

事实上，我们很少使用accept()和ignore()函数，而是想上面的示例一样，如果希望忽略事件，只要调用父类的响应函数即可。记得我们曾经说过，Qt中的事件大部分是protected的，因此，重写的函数必定存在着其父类中的响应函数，这个方法是可行的。为什么要这么做呢？因为我们无法确认父类中的这个处理函数没有操作，如果我们在子类中直接忽略事件，Qt不会再去寻找其他的接受者，那么父类的操作也就不能进行，这可能会有潜在的危险。另外我们查看一下QWidget的mousePressEvent()函数的实现：

```
void QWidget::mousePressEvent(QMouseEvent *event)
{
	event->ignore();
	if ((windowType() == Qt::Popup)) {
		event->accept();
		QWidget* w;
		while ((w = qApp->activePopupWidget()) && w != this){
			w->close();
			if (qApp->activePopupWidget() == w) // widget does not want to dissappear
				w->hide(); // hide at least
		}
		if (!rect().contains(event->pos())){
			close();
		}
	}
}
```

请注意第一条语句，如果所有子类都没有覆盖mousePressEvent函数，这个事件会在这里被忽略掉，这暗示着这个组件不

关心这个事件，这个事件就可能被传递给其父组件。

不过，事情也不是绝对的。在一个情形下，我们必须使用accept()和ignore()函数，那就是在窗口关闭的时候。如果你在窗口关闭时需要有个询问对话框，那么就需要这么去写：

```
void MainWindow::closeEvent(QCloseEvent * event)
{
        event->ignore();
    }
}

bool MainWindow::continueToClose()
{
    if(QMessageBox::question(this,
                             tr("Quit"),
                             tr("Are you sure to quit this application?"),
                             QMessageBox::Yes | QMessageBox::No,
                             QMessageBox::No)
        == QMessageBox::Yes) {
        return true;
    } else {
        return false;
    }
}
```

这样，我们经过询问之后才能正常退出程序。

2. Qt中 event() 二

今天要说的是event()函数。记得之前曾经提到过这个函数，说在事件对象创建完毕后，Qt将这个事件对象传递给QObject的event()函数。event()函数并不直接处理事件，而是将这些事件对象按照它们不同的类型，分发给不同的事件处理器(event handler)。

event()函数主要用于事件的分发，所以，如果你希望在事件分发之前做一些操作，那么，就需要注意这个event()函数了。为了达到这种目的，我们可以重写event()函数。例如，如果你希望在窗口中的tab键按下时将焦点移动到下一组件，而不是让具有焦点的组件处理，那么你就可以继承QWidget，并重写它的event()函数，已达到这个目的：

```
bool MyWidget::event(QEvent *event) {
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            // 处理Tab鍵
            return true;
        }
    }

    return QWidget::event(event);
}
```

event()函数接受一个QEvent对象，也就是需要这个函数进行转发的对象。为了进行转发，必定需要有一系列的类型判断，这就可以调用QEvent的type()函数，其返回值是QEvent::Type类型的枚举。我们处理过自己需要的事件后，可以直接return回去，对于其他我们不关心的事件，需要调用父类的event()函数继续转发，否则这个组件就只能处理我们定义的事件了。

event()函数返回值是bool类型，如果传入的事件已被识别并且处理，返回true，否则返回false。如果返回值是true，QApplication会认为这个事件已经处理完毕，会继续处理事件队列中的下一事件；如果返回值是false，QApplication会尝试寻找这个事件的下一个处理函数。

event()函数的返回值和事件的accept()和ignore()函数不同。accept()和ignore()函数用于不同的事件处理器之间的沟通，例如判断这一事件是否处理；event()函数的返回值主要是通知QApplication的notify()函数是否处理下一事件。为了更加明晰这一点，我们来看看QWidget的event()函数是如何定义的：

```
bool QWidget::event(QEvent *event) {
    switch (e->type()) {
    case QEvent::KeyPress:
        keyPressEvent((QKeyEvent *)event);
        if (!((QKeyEvent *)event)->isAccepted())
            return false;
        break;
    case QEvent::KeyRelease:
        keyReleaseEvent((QKeyEvent *)event);
```

```
            if (!((QKeyEvent *)event)->isAccepted())
                return false;
            break;
            // more...
        }
        return true;
}
```

QWidget的event()函数使用一个巨大的switch来判断QEvent的type，并且分发给不同的事件处理函数。在事件处理函数之后，使用这个事件的isAccepted()方法，获知这个事件是不是被接受，如果没有被接受则event()函数立即返回false，否则返回true。

另外一个必须重写event()函数的情形是有自定义事件的时候。如果你的程序中有自定义事件，则必须重写event()函数以便将自定义事件进行分发，否则你的自定义事件永远也不会被调用。关于自定义事件，我们会在以后的章节中介绍。

3. Qt中事件过滤器

Qt创建了QEvent事件对象之后，会调用QObject的event()函数做事件的分发。有时候，你可能需要在调用event()函数之前做一些另外的操作，比如，对话框上某些组件可能并不需要响应回车按下的事件，此时，你就需要重新定义组件的event()函数。如果组件很多，就需要重写很多次event()函数，这显然没有效率。为此，你可以使用一个事件过滤器，来判断是否需要调用event()函数。

QOjbect有一个eventFilter()函数，用于建立事件过滤器。这个函数的签名如下：

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

如果watched对象安装了事件过滤器，这个函数会被调用并进行事件过滤，然后才轮到组件进行事件处理。在重写这个函数时，如果你需要过滤掉某个事件，例如停止对这个事件的响应，需要返回true。

```
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);
        }
 }
```

上面的例子中为MainWindow建立了一个事件过滤器。为了过滤某个组件上的事件，首先需要判断这个对象是哪个组件，然后判断这个事件的类型。例如，我不想让textEdit组件处理键盘事件，于是就首先找到这个组件，如果这个事件是键盘事件，则直接返回true，也就是过滤掉了这个事件，其他事件还是要继续处理，所以返回false。对于其他组件，我们并不保证是不是还有过滤器，于是最保险的办法是调用父类的函数。

在创建了过滤器之后，下面要做的是安装这个过滤器。安装过滤器需要调用installEventFilter()函数。这个函数的签名如下：

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

这个函数是QObject的一个函数，因此可以安装到任何QObject的子类，并不仅仅是UI组件。这个函数接收一个QObject对象，调用了这个函数安装事件过滤器的组件会调用filterObj定义的eventFilter()函数。例如，textField.installEventFilter(obj)，则如果有事件发送到textField组件是，会先调用obj->eventFilter()函数，然后才会调用textField.event()。

当然，你也可以把事件过滤器安装到QApplication上面，这样就可以过滤所有的事件，已获得更大的控制权。不过，这样做的后果就是会降低事件分发的效率。

如果一个组件安装了多个过滤器，则最后一个安装的会最先调用，类似于堆栈的行为。

注意，如果你在事件过滤器中**delete**了某个接收组件，务必将返回值设为**true**。否则，**Qt**还是会将事件分发给这个接收组件，从而导致程序崩溃。

事件过滤器和被安装的组件必须在同一线程，否则，过滤器不起作用。另外，如果在install之后，这两个组件到了不同的线程，那么，只有等到二者重新回到同一线程的时候过滤器才会有效。

事件的调用最终都会调用QCoreApplication的notify()函数，因此，最大的控制权实际上是重写QCoreApplication的notify()函数。由此可以看出，Qt的事件处理实际上是分层五个层次：重定义事件处理函数，重定义event()函数，为单个组件安装事件过滤器，为QApplication安装事件过滤器，重定义QCoreApplication的notify()函数。这几个层次的控制权是逐层增大的。

3. **Qt**自定义事件

这部分将作为Qt事件部分的结束。我们在前面已经从大概上了解了Qt的事件机制。下面要说的是如何自定义事件。

Qt允许你创建自己的事件类型，这在多线程的程序中尤其有用，当然，也可以用在单线程的程序中，作为一种对象间通讯的机制。那么，为什么我需要使用事件，而不是使用信号槽呢？主要原因是，事件的分发既可以是同步的，又可以是异步的，而函数的调用或者说是槽的回调总是同步的。事件的另外一个好处是，它可以使用过滤器。

Qt中的自定义事件很简单，同其他类似的库的使用很相似，都是要继承一个类进行扩展。在Qt中，你需要继承的类是QEvent。注意，在**Qt3**中，你需要继承的类是**QCustomEvent**，不过这个类在**Qt4**中已经被废除**(**这里的废除是不建议使用，并不是从类库中删除**)**。

继承QEvent类，你需要提供一个QEvent::Type类型的参数，作为自定义事件的类型值。这里的QEvent::Type类型是QEvent里面定义的一个enum，因此，你是可以传递一个int的。重要的是，你的事件类型不能和已经存在的type值重复，否则会有不可预料的错误发生！因为系统会将你的事件当做系统事件进行派发和调用。在Qt中，系统将保留0 - 999的值，也就是说，你的事件type要大于999. 具体来说，你的自定义事件的type要在QEvent::User和QEvent::MaxUser的范围之间。其中，QEvent::User值是1000，QEvent::MaxUser的值是65535。从这里知道，你最多可以定义64536个事件，相信这个数字已经足够大了！但是，即便如此，也只能保证用户自定义事件不能覆盖系统事件，并不能保证自定义事件之间不会被覆盖。为了解决这个问题，Qt提供了一个函数：registerEventType(),用于自定义事件的注册。该函数签名如下：

```
static int QEvent::registerEventType ( int hint = -1 );
```

函数是static的，因此可以使用QEvent类直接调用。函数接受一个int值，其默认值为-1，返回值是创建的这个Type类型的值。如果hint是合法的，不会发生任何覆盖，则会返回这个值；如果hint不合法，系统会自动分配一个合法值并返回。因此，使用这个函数即可完成type值的指定。这个函数是线程安全的，因此不必另外添加同步。

你可以在QEvent子类中添加自己的事件所需要的数据，然后进行事件的发送。Qt中提供了两种发送方式：

- static bool QCoreApplication::sendEvent(QObjecy * receiver, QEvent * event)：事件被QCoreApplication的notify()函数直接发送给receiver对象，返回值是事件处理函数的返回值。使用这个函数必须要在栈上创建对象，例如：

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

- static bool QCoreApplication::postEvent(QObject * receiver, QEvent * event)：事件被QCoreApplication追加到事件列表的最后，并等待处理，该函数将事件追加后会立即返回，并且注意，该函数是线程安全的。另外一点是，使用这个函数必须要在堆上创建对象，例如：

  ```
  QApplication::postEvent(object, new MyEvent(QEvent::registerEventType(2048)));
  ```

  这个对象不需要手动delete，Qt会自动delete掉！因此，如果在post事件之后调用delete，程序可能会崩溃。另外，postEvent()函数还有一个重载的版本，增加一个优先级参数，具体请参见API。通过调用sendPostedEvent()函数可以让已提交的事件立即得到处理。

如果要处理自定义事件，可以重写QObject的customEvent()函数，该函数接收一个QEvent对象作为参数。注意，在Qt3中这个参数是QCustomEvent类型的。你可以像前面介绍的重写event()函数的方法去重写这个函数：

```
void CustomWidget::customEvent(QEvent *event) {
        CustomEvent *customEvent = static_cast<CustomEvent *>(event);
        // ....
}
```

另外，你也可以通过重写event()函数来处理自定义事件：

```
bool CustomWidget::event(QEvent *event) {
      if (event->type() == MyCustomEventType) {
            CustomEvent *myEvent = static_cast<CustomEvent *>(event);
            // processing...
            return true;
      }

    return QWidget::event(event);
}
```

这两种办法都是可行的。

好了，至此，我们已经概略的介绍了Qt的事件机制，包括事件的派发、自定义等一系列的问题。下面的章节将继续我们的学习之路！

本文出处：http://devbean.blog.51cto.com/448512/232314 在此感谢该博主对我在Qt事件学习上的帮助。

上一篇 Qt 中C++ static_cast 和 reinterpret_cast的区别

下一篇 同步与异步的区别

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场