**事件的传播是在组件层次上面的，而不是依靠类继承机制。（我们定义了一个父组件，这个父组件有很多子类，我们把这些子类依附于这个父组件，在这些组件之间，事件信号在其中随便跑，让这些组件看见自己，而不是我们因为他们是父子类就传播信号。）**

**事件和信号槽区别**

必要的时候，Qt 的事件也可以不进入事件队列，而是直接处理。信号一旦发出，对应的槽函数一定会被执行。但是，事件则可以使用“事件过滤器”进行过滤，对于有些事件进行额外的处理，另外的事件则不关心。总的来说，如果我们**使用**组件，我们关心的是信号槽；如果我们**自定义**组件，我们关心的是事件。因为我们可以通过事件来改变组件的默认操作。比如，如果我们要自定义一个能够响应鼠标事件的EventLabel，我们就需要重写QLabel的鼠标事件，做出我们希望的操作，有可能还得在恰当的时候发出一个类似按钮的clicked()信号（如果我们希望让这个EventLabel能够被其它组件使用）或者其它的信号。

Qt 中的事件都是 protected 的

如果子类没有处理这个事件，就会继续向其父类传递。Qt 的事件对象有两个函数：accept()和ignore()

我们无法确认父类中的这个处理函数有没有额外的操作，所以我们这里默认写法是如果你选择忽略这个事件不是去调用ignore（），而是把它交给父组件的相应函数、

这可能会有潜在的危险。为了避免自己去调用accept()和ignore()函数，而是尽量调用父类实现，Qt 做了特殊的设计：事件对象默认是 accept 的，而作为所有组件的父类QWidget的默认实现则是调用ignore()。如果你实现了自己的响应函数，父类ignore（），事件接受就停留在这一个组件了，变相实现了accept（），如果你调用父类的响应函数，就把父类的ignore（），变成了accpt（），这样就交给上层去处理了。

**特殊情况**

在一个特殊的情形下，我们必须使用accept()和ignore()函数，那就是窗口关闭的事件。对于窗口关闭QCloseEvent事件，调用accept()意味着 Qt 会停止事件的传播，窗口关闭；调用ignore()则意味着事件继续传播，即阻止窗口关闭，当然是我们点击了关闭窗口他就之间关闭，而不是继续把它往上面传递。

事件分发

我们曾经提到event()函数。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分发器源码。我们先阶段不关系事件是如何产生的，event()函数是 protected 的，但是我们可以通过这个简洁版源码窥探事件是如何处理的。实际上我们还是通过switch的操作来完成，针对不同事件处理，

bool QObject::event(QEvent \*e)

{

switch (e->type()) {

case QEvent::Timer:

timerEvent((QTimerEvent\*)e);

break;

case QEvent::ChildAdded:

case QEvent::ChildPolished:

case QEvent::ChildRemoved:

childEvent((QChildEvent\*)e);

break;

// ...

default:

if (e->type() >= QEvent::User) {

customEvent(e);

break;

}

return false;

}

return true;

}

处理

文档中event()是一个集中处理不同类型的事件的地方。如果你不想重写一大堆事件处理器，就可以重写这个event()函数，通过QEvent::type()判断不同的事件。

在event中调用事件对象的accept()和ignore()函数是没有作用的。函数返回值是 bool 类型。如果传入的事件已被识别并且处理，则需要返回 true，否则返回 false。如果返回值是 true，并且，该事件对象设置了accept()，那么 Qt 会认为这个事件已经处理完毕，我们返回true代表accept（），我们可以直接返回false 而不调用父组件，这样我们相当于只处理我们想要的事件。而不是处理父组件提供的所有事件。

事件重写方法

bool CustomWidget::event(QEvent \*e)

{

if (e->type() == QEvent::KeyPress) {

QKeyEvent \*keyEvent = static\_cast<QKeyEvent \*>(e);

if (keyEvent->key() == Qt::Key\_Tab) {

qDebug() << "You press tab.";

return true;

}

}

return QWidget::event(e);

}

**事件拦截**

（事件过滤器不是 protected 的，因此我们可以向任何QObject子类安装事件过滤器；）

我们可以在event()函数中实现拦截的操作。由于event()函数是 protected 的，因此，需要继承已有类。如果组件很多，就需要重写很多个event()函数。这当然相当麻烦，

解决两个问题

QWidget::event()函数是一个 protected 的函数，这意味着我们要想重写event()，必须继承一个已有的类（我们无法通过一个类看到他的protected部分）。试想，我的程序根本不想要鼠标事件，程序中所有组件都不允许处理鼠标事件，是不是我得继承所有组件，一一重写其event()函数？protected 函数带来的另外一个问题是，如果我基于第三方库进行开发，而对方没有提供源代码，只有一个链接库，其它都是封装好的。我怎么去继承这种库中的组件呢？

2.event()函数的确有一定的控制，不过有时候我的需求更严格一些：我希望那些组件根本看不到这种事件。event()函数虽然可以拦截，但其实也是接收到了QMouseEvent对象。我连让它收都收不到。这样做的好处是，模拟一种系统根本没有那个事件的效果，所以其它组件根本不会收到这个事件，也就无需修改自己的事件处理函数。这种需求怎么办呢？

**解决方案**

事件过滤器给我们一种能力，让我们能够完全移除某种事件。事件过滤器可以安装到任意

QObject类型上面，并且可以安装多个。如果要实现全局的事件过滤器，则可以安装到

QApplication或者QCoreApplication上面。这里需要注意的是，如果使用installEventFilter()函数给一个对象安装事件过滤器，那么该事件过滤器只对该对象有效，只有这个对象的事件需要传递给事件过滤器的eventFilter()函数进行过滤，其它对象不受影响。如果给QApplication对象安装事件过滤器，那么该过滤器对程序中的每一个对象都有效，任何对象的事件都是先传eventFilter()函数。

自定义事件

QEvent::Type用于区分各个事件，我们的自定义事件类型不能和已经存在的 type 值重复，

registerEventType()保证自定义事件相互之间不会被覆盖。

函数接受一个 int 值，其默认值是 -1；函数返回值是向系统注册的新的 Type 类型的值。如果 hint 是合法的，也就是说这个 hint 不会发生任何覆盖（系统的以及其它自定义事件的），则会直接返回这个值；否则，系统会自动分配一个合法值并返回

高能预警 （事件发送原理 机制）

事件发送两种方式

1.

receiver就是我们要接受事件的组件，event是一个父类，中间包含各种子类信息，相当于一个类的集合体。

static bool QCoreApplication::sendEvent(QObject \*receiver, QEvent \*event);

直接将event事件发送给receiver接受者，使用的是QCoreApplication::notify()函数。函数返回就是事件处理函数的返回值。在事件被发送的时候，event对象并不会被销毁。通常我们会在栈上创建event对象

调用方式

QMouseEvent event(QEvent::MouseButtonPress, pos, 0, 0, 0);

QApplication::sendEvent(mainWindow, &event);

2.

static void QCoreApplication::postEvent(QObject \*receiver, QEvent \*event);

因为 post 事件队列会持有事件对象，并且在其 post 的时候将其 delete 掉，因此，我们必须在堆上创建event对象。当对象被发送之后，再试图访问event对象就会出现问题（因为 post 之后，event对象就会被 delete）。

(前面两种调用方式再去调用notify，对于notify我们在进入源码进行分析，试探他的原理)实际上都是在调用notify（它是最根本的）进行分发

源码

bool QCoreApplication::notify(QObject \*receiver, QEvent \*event)

{ ........

false : d->notify\_helper(receiver, event);
}

我们看到在notify\_helper中调用了receiver->event(event);所以这两个变量receiver 和event到底是什么了，调用receiver->event(event);这个以后，我们就连接到前面所说的了事件分发了。

bool QCoreApplicationPrivate::notify\_helper(QObject \*receiver, QEvent \* event)

{

// send to all application event filters

if (sendThroughApplicationEventFilters(receiver, event))

return true;

// send to all receiver event filters

if (sendThroughObjectEventFilters(receiver, event))

return true;

// deliver the event

return receiver->event(event);

}