**Layout Management（布局管理）**

Qt的布局管理系统使用起来简单方便、功能强大，可以自动的对某个widget下的子widget进行合理化布局，充分的使用所有可用的空间。

**概述**Qt有一系列的布局管理类，用来描述在交互界面上这些子部件如果排列。当子部件的位置、大小改变后，这些个布局管理类会自动处理子部件的空间位置、大小，使之可以整齐的排列起来。所有的QWidget子类都可以使用布局管理类来排列其子widget。调用QWidget::setLayout()函数可以指定该widget使用什么样的布局方式。当某个widget被指定布局时，该布局将会自动的完成如下的动作：

* 定位子widget的位置
* 设置windows的默认大小
* 设置windows的最小大小
* 重新调整大小
* 当如下部分更改时自动调整更新：

(1) 字号、text内容

(2) 显示/隐藏某widget

(3) 删除某widget

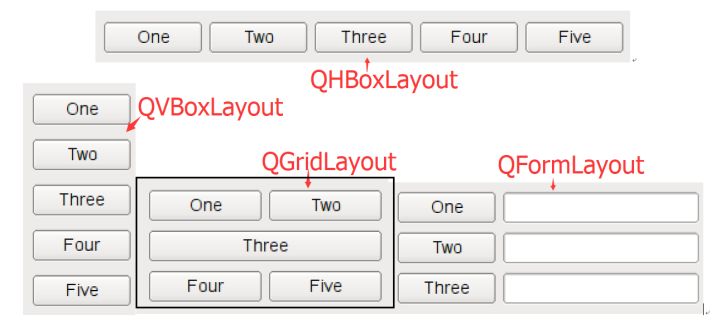
**Qt的布局管理类**

Qt的这些布局管理类是专门为手敲代码进行布局而设计的，并且布局的精度以像素为单位。Qt Designer这种可视化布局的时候也同样使用了这些布局管理类。用Qt Designer设计表单有个好处，就是这些布局避免了编译、链接、循环运行。

* **QGraphicsAnchor**（在QGraphicsAnchorLayout中表示两个item的锚点）；
* **QGraphicsAnchorLayout**（在Graphics View中一个item链接到widget的锚点的布局）；
* **QBoxLayout**（水平/垂直布局）；
* **QHBoxLayout**（水平布局）；
* **QVBoxLayout**（垂直布局）；
* **QFormLayout**（表单布局）；
* **QGridLayout**（网格布局）；
* **QLayout**（布局的基类）；
* **QLayoutItem**（布局中的抽象项目）；
* **QSpacerItem**（布局中的空格）；
* **QWidgetItem**（布局中的widget项目）；
* **QSizePolicy**（布局策略，描述了水平/垂直布局时的策略）；
* **QStackedLayout**（堆栈widget，同一时刻只有一个widget可显示）；
* **QButtonGroup**（管理一堆按钮的容器）；
* **QGroupBox**（有标题的组合框）；
* **QStackedWidget**（堆栈widget）；

水平布局、垂直布局、网格布局、表单布局

最简单的使用布局的方法就是用QHBoxLayout、QVBoxLayout、QGridLayout、QFormLayout来给需要的widget设置布局。这些类都继承于QLayout，而QLayout又继承于QObject。要想设置更加复杂的布局，可以嵌套使用。



**示例代码**

QHBoxLayout

QWidget **\***window **=** **new** QWidget;

QPushButton **\***button1 **=** **new** QPushButton("One");

QPushButton **\***button2 **=** **new** QPushButton("Two");

QPushButton **\***button3 **=** **new** QPushButton("Three");

QPushButton **\***button4 **=** **new** QPushButton("Four");

QPushButton **\***button5 **=** **new** QPushButton("Five");

QHBoxLayout **\***layout **=** **new** QHBoxLayout;

layout**->**addWidget(button1);

layout**->**addWidget(button2);

layout**->**addWidget(button3);

layout**->**addWidget(button4);

layout**->**addWidget(button5);

window**->**setLayout(layout);

window**->**show();

QGridLayout

QWidget **\***window **=** **new** QWidget;

QPushButton **\***button1 **=** **new** QPushButton("One");

QPushButton **\***button2 **=** **new** QPushButton("Two");

QPushButton **\***button3 **=** **new** QPushButton("Three");

QPushButton **\***button4 **=** **new** QPushButton("Four");

QPushButton **\***button5 **=** **new** QPushButton("Five");

QGridLayout **\***layout **=** **new** QGridLayout;

layout**->**addWidget(button1, 0, 0);

layout**->**addWidget(button2, 0, 1);

layout**->**addWidget(button3, 1, 0, 1, 2);*//第二行第一列，占据一行，占据两列*

layout**->**addWidget(button4, 2, 0);

layout**->**addWidget(button5, 2, 1);

window**->**setLayout(layout);

window**->**show();

QFormLayout

QWidget \*window = new QWidget;

QPushButton \*button1 = new QPushButton("One");

QLineEdit \*lineEdit1 = new QLineEdit();

QPushButton \*button2 = new QPushButton("Two");

QLineEdit \*lineEdit2 = new QLineEdit();

QPushButton \*button3 = new QPushButton("Three");

QLineEdit \*lineEdit3 = new QLineEdit();

QFormLayout \*layout = new QFormLayout;

layout->addRow(button1, lineEdit1);

layout->addRow(button2, lineEdit2);

layout->addRow(button3, lineEdit3);

window->setLayout(layout);

window->show();

**注意事项**

**当你使用某个布局时，构建widget的时候不需要指定父对象（parent）**。layout会自动的为其重新分配父对象。layout不是父对象，widget只能令其他widget作为父对象，不可以将layout作为父对象。

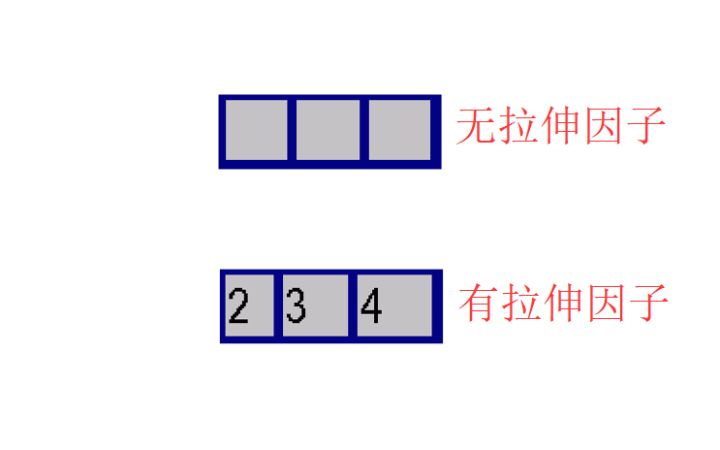
**将widget加入到某个布局中**

当你把某个widget加入到某个布局中时，布局会完成如下动作：

1. 所有的widget都先按照各自的QWidget::sizePolicy()和QWidget::sizeHint()分配空间；
2. 如果某个widget设置了stretch factor（拉伸因子），且大于零，则会按照拉伸因子的比例分配空间；
3. 如果某个widget的拉伸因子等于零，则只有在其他widget不需要空间的时候才会占据这部分空间；
4. 若空间小于某widget的minimum size（最小尺寸）时，会按照这个最小尺寸分配。假如没有设置minimum size，会按照minimum size hint分配最小空间。widgets其实不需要设置minimum size或者minimum size hint，stretch factor（拉伸因子）反而是最为重要的。
5. 若空间大于某widget的maximum size（最大尺寸）时，会按照这个最大尺寸分配。同样的，拉伸因子是最为重要的。

**Stretch Factor（拉伸因子）**

widget一般都没有设置stretch factor，这个是用来设置当前widget和其他widget分配空间的比例。



**为自定义widget设置布局**

当你自定义一个QWidget后，还应该传递出该widget的布局属性。假如你自定义的这个widget没有子widget，或者说你对其手动进行了布局，你可以使用如下的机制来改变widget的行为：

* 重新实现QWidget::sizeHint()来返回首选尺寸
* 重新实现QWidget::minimumSizeHint()来返回允许的最小尺寸
* 调用QWidget::setSizePolicy()设置widget的布局要求

无论什么时候，只要size hint、minimum size hint、size policy改变时，都要调用QWidget::updateGeometry()来使布局重新计算各部分的尺寸来更新。不管调用QWidget::updateGeometry()多少次，只要那三个不变，都只会计算一次。

当自定义的widget的高度取决于实际宽度时（例如具有自动断字功能的label），需要在widget的size policy（尺寸策略）中设置“height-for-width”flag，并重新实现QWidget::heightForWidth()函数。

**布局中的问题**

在label widget中使用rich text的时候会产生一些布局问题。假如这个label是文字，而Qt布局管理处理rich text时就会发生问题。

在一些情况下，父layout是QLayout::FreeResize模式，这就导致布局无法适应其内容而调整尺寸，甚至导致用户无法在小尺寸上进行操作。解决的办法就是子类化出现问题的widget，然后重新实现sizeHint()、minimumSizeHint()函数。在某些情况下，比如设置QDockWidget或者QScrollArea的widget时，必须在widget已经有个布局存在，否则的话这个widget不可见。

**手动布局**如果你自定义个特殊的布局，还可以按照上述方法自定义一个widget。重新实现QWidget::resizeEvent()来计算所需空间，并调用每个子widget的setGeometry()函数。当widget需要重新计算时，它会接收**QEvent::LayoutRequest**类型的事件。重新执行QWidget::event()来处理QEvent::LayoutRequest事件。

**事件系统**

**The Event System**

在Qt中，事件是一种对象，继承自抽象基类QEvent。QEvent代表了，所有来自程序内部或外部发生的事件。事件可以被QObject子类对象接收和处理，此处的子类多为widget 相关。这篇文章讲述了事件在在某一特定应用程序中分发和处理。

**事件是如何分发的**

**How Events are Delivered**

当一个事件发生，Qt创建一个可以代表当前事件的QEvent子类对象，通过调用even()函数分发到QObject或其子类对象中去。

event()函数本身并不处理事件，它会依据事件类型，调用相应的事件处理函数，并依据事件是否被接受或忽略作出回应。 

有一些事件，比如鼠标事件、键盘事件来自windows系统，而有些，比如QTimerEvent,来自其它的资源，有些 则来自应用本身。 

**事件类型**

**Event Types**

很多事件类型拥有自己专有的类，比如 窗口缩放事件，绘图事件，鼠标事件，键盘事件，关窗口事件。这些QEvent的子类添加了专用的事件函数，比如：窗口缩放事件添加了size()和oldSize()函数，从而是窗口知其尺寸的改变。 

有些事件类支持不止一种事件类型，比如，鼠标事件，就支持单击，双击，移动和其它相关的操作。 每一个事件，拥有一个相应的类型，定义在QEvent::Type中，这可以方便的作为运行时信息以便快速的确定给定的事件构建于QEvent的哪一种子类型。 

程序的需要多种而复杂的交互方式，所以Qtr事件分发机制也相当灵活。关于 QCoreApplication::notify() 的文单简明的叙述了整件过程。Qt季刊，从另一个视角重述了上个过程。在此，我们解释95%的应用情形。 

**事件处理函数**

**Event Handlers**

处理分发的事件的常规方式是调用虚函数，比如，绘图事件分发通过调用 QWidget::paintEvent()，虚函数负责相关的回应，常重过重绘窗口。如果你不在覆写的虚函数中实现需求，你需要调用基类的实现。 

比如:在处理checkbox控件时，把左键单击的事件交由覆写函数处理，而其它的点交同其父类的实现。 

void MyCheckBox::mousePressEvent(QMouseEvent \*event)

{

if (event->button() == Qt::LeftButton) {

// handle left mouse button here

} else {

// pass on other buttons to base class

QCheckBox::mousePressEvent(event);

}

}9

如果你想完全取代父类的函数，你需要自实现所有的功能。但是，如果你想扩展父类的功能，你可以实现你想实现的并且调用父类默认你不关心的事件。 

无独有偶，正巧有event函数，或许此函功能并不充足。 许多常见案例中，引入了按压Tab key。通常，窗口截获用于移动窗口焦点，但有一些控件，需要Tab key 为其所用。 

这些对象，可以通过覆写QObject::event()，在常规事件前后处理自己的事件，或是完全自实现。一个非寻常的窗口控件，实现的了截获Tab和自定义事件，代码如下： 

bool MyWidget::event(QEvent \*event)

{

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

QKeyEvent \*ke = static\_cast<QKeyEvent \*>(event);

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

// special tab handling here

return true;

}

} else if (event->type() == MyCustomEventType) {

MyCustomEvent \*myEvent = static\_cast<MyCustomEvent \*>(event);

// custom event handling here

return true;

}

return QWidget::event(event);

}

注意: QWidget::event()依然被调用，用于处理没有被处理的其它情形，通过返回值来表明事件是否被处理，返回true阻止了事件继续被分发。 

**事件过滤器**

**Event Filters**

有时一个对象，需要观测或是拦截被分发到其它对象上的事件。比如，对话框通常会为其它的控件过滤按键，比如，修改Return-key的处理。 

通过函数QObject::installEventFilter()建立一个事件过滤器，可以实现上述功能。通过指定过滤对象的QObject::eventFilter()函数去接收目标对象的事件，事件过滤器先于目标对象处理事件，可依据要求对事件进行检测或丢弃。一个事件过滤器可以通过QObject::removeEventFilter()函数移除。 

当事件过滤器对象的覆写函数eventFilter()被调用，事件可以被接受或丢弃，允许或是禁止更进一步的事件处理。如果所有的事件过滤器允许处理一个事件(其中一个返回false),事件将会被发送到目标对象。如果他中有任何一个停止处理(返回true),目标对象和后序的事件过滤器不会再看到事件。 

bool FilterObject::eventFilter(QObject \*object, QEvent \*event)

{

if (object == target && event->type() == QEvent::KeyPress) {

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

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

// Special tab handling

return true;

} else

return false;

}

return false;

}

上述代码用另外一种方式，截获了发向指定目标控件的Tab 按压事件。在这种情况下，过滤器处理了相关的事件，并且近回true阻止进一步的处理。其它所有的事件忽略，并且过滤器近返回false，允许通过其它按装该事件过滤器，发送到目标控件， 

可以通过按装过滤器到QApplication,QCoreApplication的对象，过滤程序的所有事件。这样的全局过滤器在其它对象过滤器之前被调用。这样强大的功能，却以牺牲事件的分发效率为代价，应用其它的技术来取代它。 

**发送事件**

**Sending Events**

很多程序希望自己创建并发送事件。你可以通过QCoreApplication::sendEvent() and QCoreApplication::postEvent()两个函数，像Qt事件循环构一样建事件对象并发送事件。 

sendEvent()函数立即处理事件，当它返回。事件过滤器，或对象本身己经处理完事件，对于许多事件类来说，有一个函数叫isAccepted()，通知你事件是否被上一个处理函数接受或是拒绝。 

postEvent()函数把事件投到队列中等待处理。在下一轮的Qt事件主循环中，事件被带有某些优化规则的分发。比如，如果有很多resize事件，他们会被压缩为一个。这同样应用于绘图事件，QWidget::update() 调用 postEvent()，可以消除不稳定，减少重绘以加快速度。   
要创建自定义事件类型，你需要定义一个事件号码，该号码要大过QEvent::User，为了传递自定义事件信息，你需为创建一个QEvent 的子类。 