Qt GUI编程入门（2）

计算机系

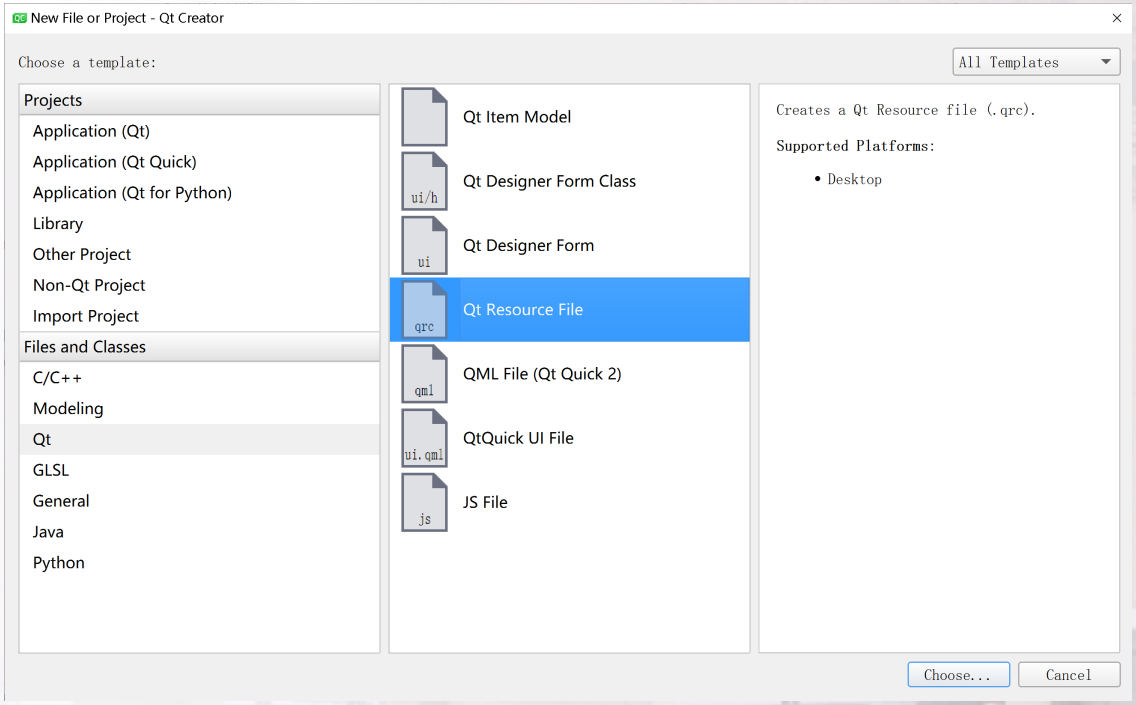
课程主要内容

- Qt资源管理
- QAction
- Qt事件处理
- Qt绘图

Qt资源管理

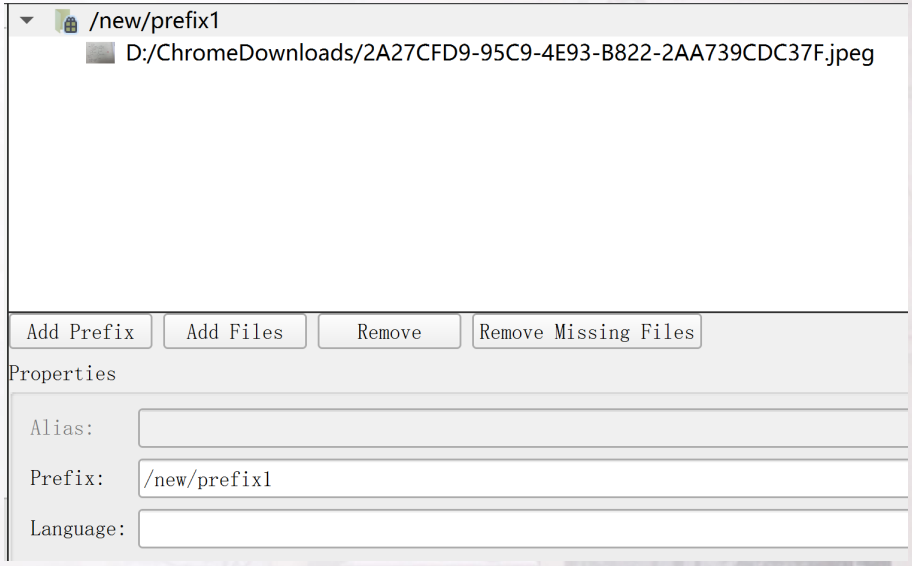
资源管理

- Qt使用资源文件(.qrc) 管理资源
- 可用的资源包括图片、音频、视频等
- 在新建文件时，可以选择新建资源文件



资源管理

- 右击资源文件，选择*Open in Editor*，可以用资源编辑器（Resource Editor）打开
- 可以添加*Prefix*（用于通过代码访问资源），新的目录、新的文件等
- 如图为添加新目录和文件后的示例



- 参考文档：https://doc.qt.io/qt-6/designer-resources.html

资源管理

- 将图片、音频等资源放进一个资源文件中，Qt会将它们内嵌进可执行文件
  - 例如，将图片素材添加进资源文件，再使用QPixmap在QLabel中显示出来
  - 避免部署多个文件
  - 不需要关心资源的路径位置
  - 编译构建系统会进行自适应配置

资源管理

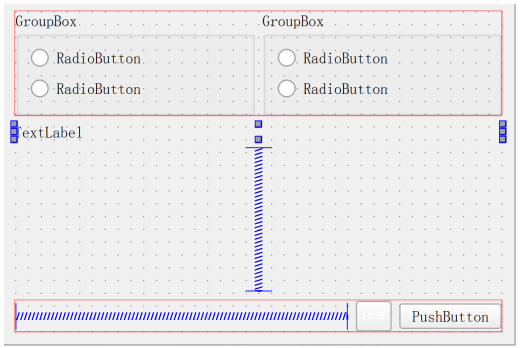
- 通过代码引用资源，在路径和文件名前添加“:”
  - 对于具有复杂逻辑的项目，一般使用代码管理

```
QPixmap pm(":/images/logo.png");
```

- 也可以通过Qt设计器引用资源
  - 添加一个部件
  - 为部件属性（如图标icon）设置资源
  - 在弹出的资源管理器中选择对应的资源

资源管理示例

- 以使用QLabel显示图片为例



1 拖入QLabel部件

label_2 : QLabel

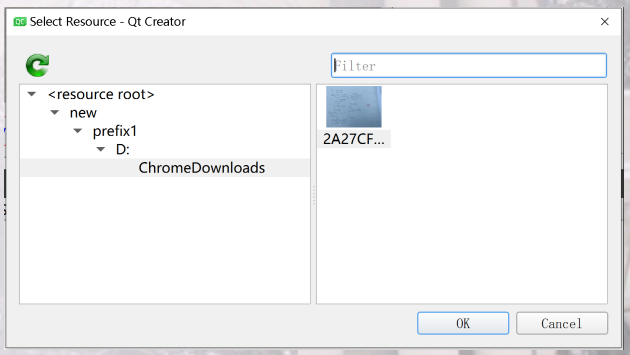
Property	Value
lineWidth	1
midLineWidth	0
QLabel	
text	TextLabel
translatable	☑
disambiguation	
comment	
textFormat	AutoText

2 删除属性中的text文本

label_2 : QLabel

Property	Value
translatable	☑
disambiguation	
comment	
textFormat	AutoText
pixmap	
scaledContents	☐
alignment	AlignLeft, AlignVCenter
wordWrap	☐

3 选择属性中的pixmap，单击“...”选择资源

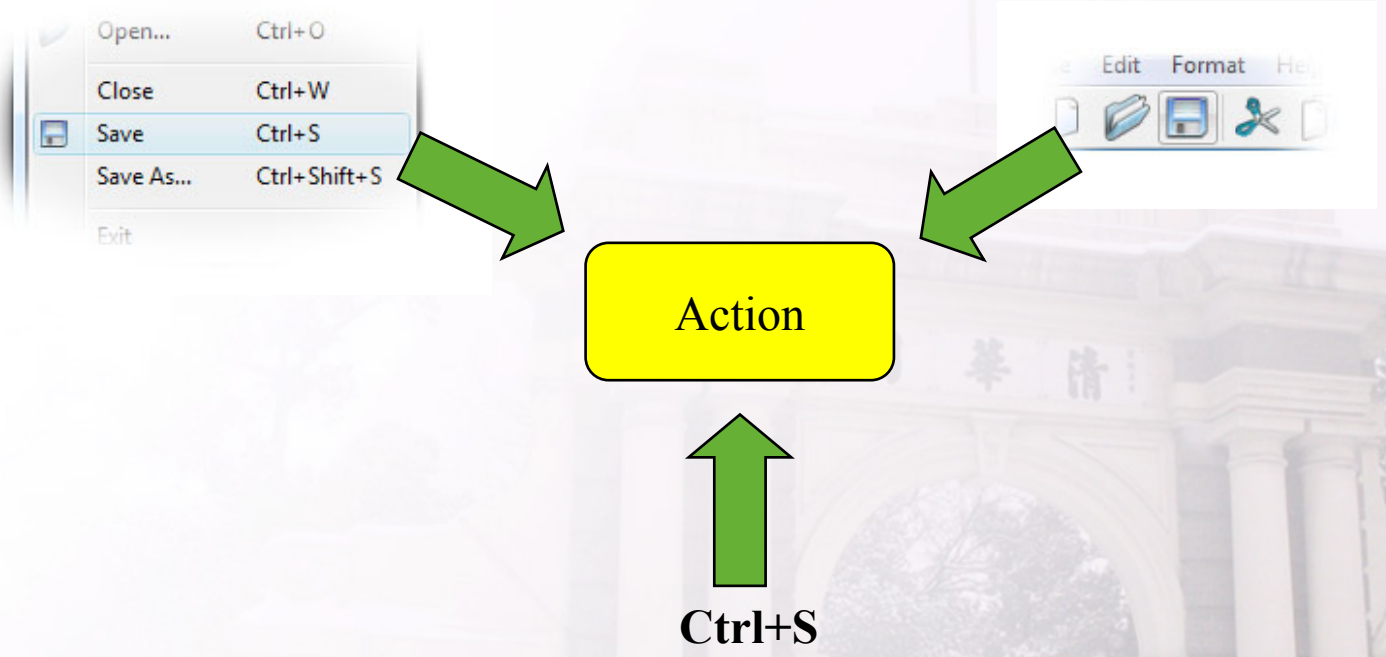


4 选中所需资源后，调整属性使其正确显示

QAction

QAction介绍

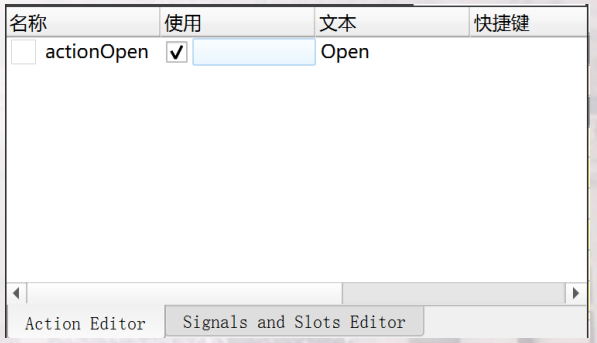
- 菜单栏和工具栏经常具有相同的行为（action）



- QAction封装所有菜单、工具栏和快捷键需要的设置，包括工具和状态栏提示

QAction介绍

- 用Qt设计器进行界面设计时，会自动为菜单项创建QAction对象
- 可使用设计器中的Action Editor进行编辑，编写其triggered()信号对应的槽函数，可实现相应行为



- QAction常用属性
  - text – 文本
  - icon – 图标
  - shortcut – 快捷键
  - checkable/checked – 当前操作是否可选中/已选中
  - toolTip/statusTip – 工具栏提示文本(鼠标在工具栏上悬停时触发)和状态栏提示文本
  - objectName – 对象名

基于纯代码或使用Qt设计器

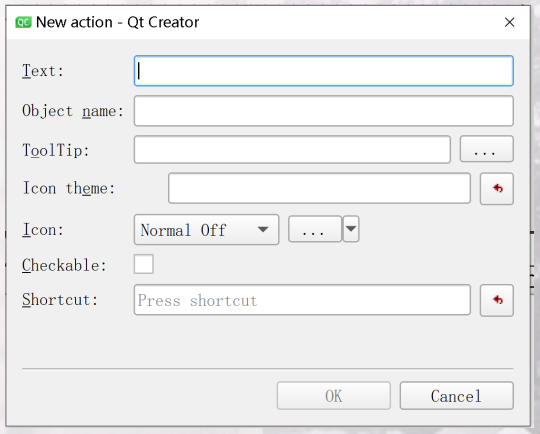
```
QAction *action = new QAction(parent);

action->setText("text");

action->setIcon(QIcon(":/icons/icon.png"));

action->setShortcut(QKeySequence("Ctrl+G"));

action->setData(myQVariantData);
```

生成新的action

设置文本，图标和快捷键

QAction对象可以携带用户设置的QVariant数据

- 也可以通过Qt设计器编辑（在Action Editor中右键点击QAction对象）

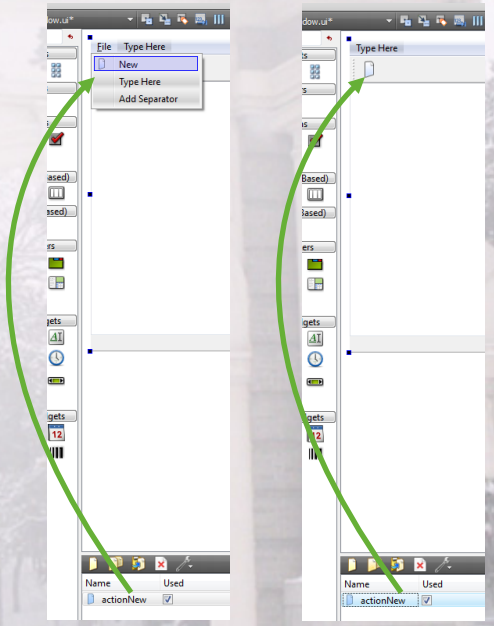


添加QAction对象

- 纯代码形式

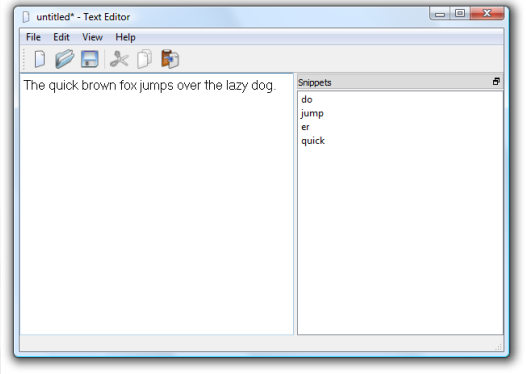
```
myMenu->addAction(action);
myToolBar->addAction(action);
```

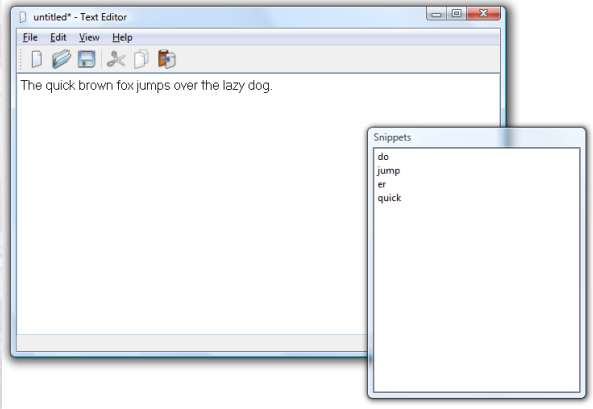
- 通过Qt设计器
  - 在菜单栏编辑菜单项，会在*Action Editor*中自动创建QAction对象
  - 在窗体区域点击右键，选择“添加工具栏”，然后可将*Action Editor*中的QAction对象拖放到工具栏，也可以拖放到菜单栏



可停靠部件

- 可停靠部件是放置于QMainWindow窗体边缘区域的可拆分的部件
- 在Qt设计器，可将部件拖放到QDockWidget中
- 采用纯代码方式，可以调用QMainWindow::addDockWidget向窗体添加可停靠部件





可停靠部件代码

```
void MainWindow::createDock()
{
    QDockWidget *dock = new QDockWidget("Dock", this);

    dock->setFeatures(QDockWidget::DockWidgetMovable |
                      QDockWidget::DockWidgetFloatable);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea |
                          Qt::RightDockWidgetArea);
    dock->setWidget(actualWidget);

    ...

    addDockWidget(Qt::RightDockWidgetArea, dock);
}
```

新建带标题的dockWidget

可以移动或者漂浮

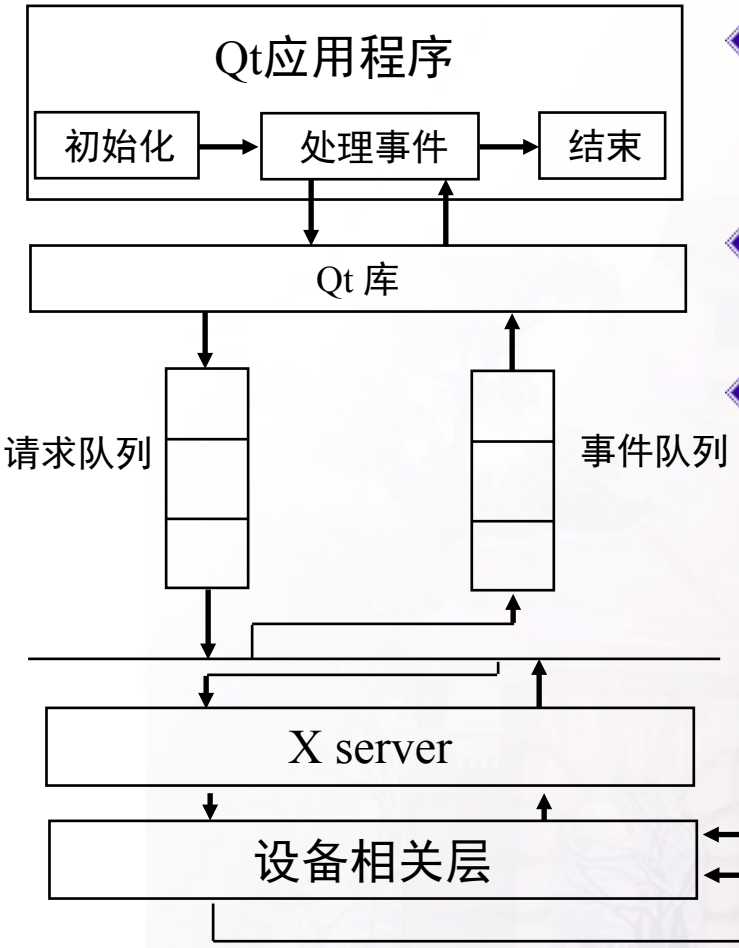
实际部件

可以停靠在左右边缘

将dockWidget添加进窗体

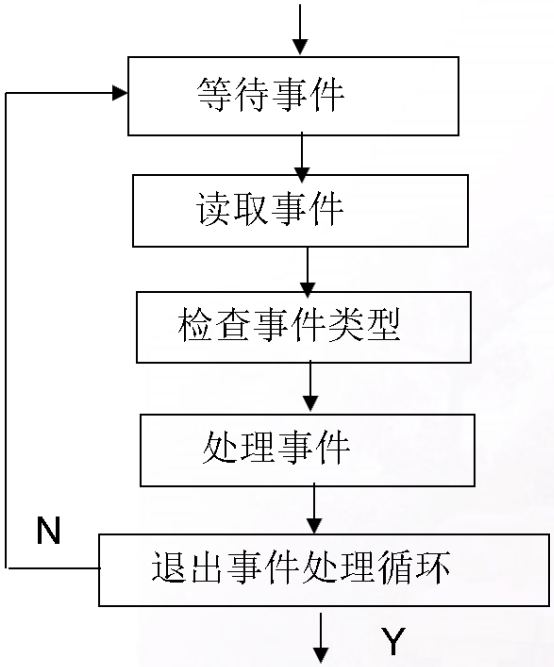
Qt事件处理

Qt事件驱动应用程序



- 在Qt中，所有事件（敲击键盘，鼠标指针在窗口中移动，鼠标按键等事件）都发送到Qt事件队列中
- 在Qt中，事件是一个被发送到事件处理函数的QEvent子类对象
- QEvent子类包括QMouseEvent, QKeyEvent, QPaintEvent, QTimerEvent等

Qt事件循环



- Qt执行QApplication::exec函数进入主事件循环
- 在事件循环中，从事件队列取出事件，并进行处理
- 为提升效率，Qt可能会合并事件队列中的事件
  - 只有最后一个鼠标移动事件(QMouseMoveEvent)被处理
  - 多个绘图事件(QPaintEvent) 也可能合并为一个
- 然后检查是否退出事件循环，如是，则退出
- 如果在处理事件时，去执行耗时的计算、或阻塞的等待，会导致GUI响应慢，甚至界面无响应

Qt事件处理机制

- Qt将从事件队列中取出的事件封装成QEvent对象
- 然后调用QObject::event(QEvent* e)虚函数进行处理，该虚函数根据事件类型，将事件分发到特定事件处理器（如mousePressEvent）
- 对于QWidget应用，基于C++虚函数机制处理事件
  - 在QWidget的子类中重载event虚函数可以在事件分发到特定事件处理器前预先处理
  - 也可以在QWidget子类中重载特定事件处理器，如keyPressEvent

事件处理方式一：重载虚函数event()

- event()函数返回值是bool类型
  - 如果传入事件已被识别并且处理，返回true（不再分发该事件）
  - 否则，调用QWidget::event分发下去处理
- 示例：在窗口中按下tab键，将焦点移动到下一组件，而不是让具有焦点的组件处理
  - MyWidget是QWidget的子类，继承了QObject类的event虚函数

```
bool MyWidget::event(QEvent *event) {
        // QEvent::type()函数返回枚举类型QEvent::Type
        if (event->type() == QEvent::KeyPress) {
                QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
                if (keyEvent->key() == Qt::Key_Tab) {
                        // 在此处处理Tab键，移动到下一个组件
                        return true;
                }
        }
        return QWidget::event(event);
}
```

事件处理方式二：重载特定事件处理器

- 特定事件处理器
  - 响应按键事件

    void keyPressEvent(QKeyEvent*)
  - 响应时钟事件

    void timerEvent(QTimerEvent*)
  - 响应鼠标事件

    void mousePressEvent(QMouseEvent*)

    void mouseDoubleClickEvent(QMouseEvent * event)
  - 响应布局改变事件

    void resizeEvent(QResizeEvent*)

    void moveEvent(QMoveEvent*)

示例：重载特定事件处理器

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

事件处理方式三：安装事件过滤器

- 过滤器是一个监视对象，是QObject子类的对象，需要重载eventFilter函数
  - virtual bool QObject::*eventFilter* (QObject * *target*, QEvent * *event*)
  - 如果被监视对象target安装了过滤器，则当有事件发生时，将自动调用*eventFilter*函数
- 安装过滤器
  - 被监视对象需要调用如下函数安装过滤器

    void QObject::installEventFilter (QObject * *filterObj*)
  - 可将监视对象*filterObj*安装到任何QObject的子类的对象上
  - 如果一个组件对象安装了多个过滤器，则最后一个安装的过滤器将最先调用（类似于堆栈行为）

示例：安装事件过滤器

```
bool MainWindow::eventFilter(QObject *obj, QEvent *event)
{ // MainWindow为监视对象，  ui->textEdit为被监视对象
    if (obj == ui->textEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent*>(event);
            qDebug() << "Ate key press" << keyEvent->key();
            return true; // 停止对该事件的响应
        } else {
            return false;
        }
    } else {
        // pass the event on to the parent class
        return QMainWindow::eventFilter(obj, event);
    }
}
//…
MainWindow::MainWindow(…)… { ui->textEdit->installEventFilter(this); }
```

计时器事件

- QTimer可以产生计时器事件，便于开发计时应用
- 示例一：定时执行某个动作

```
MyClass(QObject *parent) : QObject(parent)
{
    QTimer *timer = new QTimer(this);
    timer->setInterval(5000);
    connect(timer, SIGNAL(timeout()), this, SLOT(doSomething()));
    timer->start();
}
```

5000ms，即5s

- 示例二：延迟一个动作的执行

```
QTimer::singleShot(1500, dest, SLOT(doSomething());
```

示例：关闭窗口事件

- 通过拦截关闭窗口事件，弹出警告窗口
  - 可以重载如下函数

```
void QWidget::closeEvent ( QCloseEvent * event )
[virtual protected]
```

```
#include <QCloseEvent>
void MainWindow::closeEvent(QCloseEvent * event) {
    int ret = QMessageBox::warning(0, tr(“App"),
                                   tr("您真的想要退出？"),
              QMessageBox::Yes | QMessageBox::No);
    if (ret == QMessageBox::Yes) {
            event->accept(); //确认关闭
    } else {
            event->ignore(); //不关闭
    }
}
```

实例讲解
事件过滤器

Qt 2D绘图

Qt 2D绘图

- Qt二维绘图：使用QPainter在绘图设备（如QWidget、QPixmap）上，通过绘制点、线、圆等基本形状组合成图形
  - QPainter：画点，画线，填充，变换，alpha通道等。
  - QPaintDevice：是QPainter用来绘图的绘图设备，Qt提供预定义的绘图设备，如QWidget，QPixmap，QImage等，都从QPaintDevice继承而来
  - QPaintEngine：提供QPainter在不同设备上绘制的统一接口，通常对开发人员是透明的。
- 此外，Qt还提供了Graphics View架构，使用QGraphicsView、QGraphicsScene和各种QGraphicsItem绘图，每个组件是可选择、可交互的
  - 对于有兴趣的同学，可作为课外自学内容

Qt 2D绘图

- 处理绘制事件，只需要重载QWidget::paintEvent函数，并在该函数中实例化一个QPainter对象进行绘制

```
class MyWidget : public QWidget
{
    ...

protected:
    void paintEvent(QPaintEvent*);
```

```
void MyWidget::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);
    //在this指针所指向的MyWidget对象上进行绘制
}
```

Qt绘制事件处理

- 当应用程序收到绘制事件时，会调用QWidget::paintEvent()虚函数
  - 该函数是进行图形绘制的位置
- 有两种手工代码方式触发绘制事件，重绘窗口
  - 调用update()函数 – 把重绘事件添加到事件队列中
    - 重复调用update() 会被Qt合并为一次
    - 不会产生图像的闪烁，但可能有延迟
  - 调用repaint()函数 – 立即产生绘制事件
    - 一般情况下不推荐使用该方法，除非需要立即重绘的特效情况
  - 两种方法都可以可带参数指定重绘特定区域

QPainter

- QPainter三要素
  - 画笔：QPen对象，可以设置线条颜色、宽度、线型等，绘制线条和轮廓
  - 画刷：QBrush对象，可以设置一个线条轮廓内区域的填充方式，如填充颜色、填充方式、渐变特性、填充图片等
  - 字体：QFont对象，当绘制文字时，Qt使用指定字体属性，如果没有匹配的字体，将使用最接近的字体
- QPainter::RenderHint属性指定是否进行反锯齿操作
  - QPainter::Antialiasing：在可能的情况下进行边的反锯齿
  - QPainter::TextAntialiasing：在可能的情况下进行文字的反锯齿绘制
  - QPainter::SmoothPixmapTransform：使用平滑的pixmap变换算法(双线性插值算法)，而不是近邻插值算法

QPainter的绘图函数

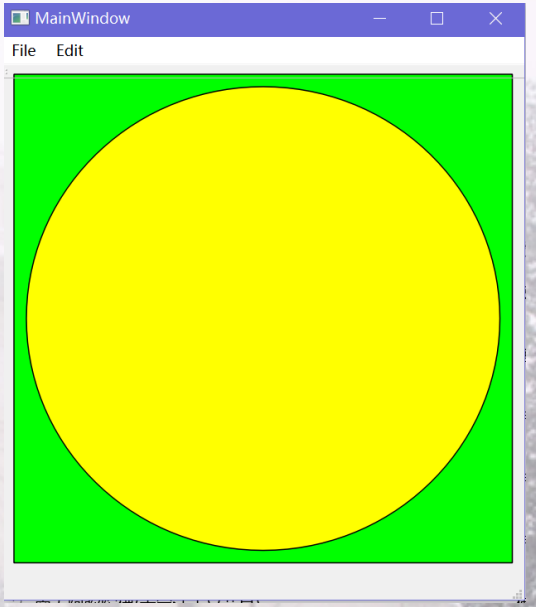
- 详细的接口和其他函数参见Qt文档

drawArc()	弧
drawChord()	弦
drawConvexPolygon()	凸多边形
drawEllipse()	椭圆
drawImage()	QImage表示的图像
drawLine()	线
drawLines()	多条线
drawPath()	路径
drawPicture()	按QPainter指令绘制
drawPie()	扇形
drawPixmap()	QPixmap表示的图像
drawPoint()	点
drawPoints()	多个点
drawPolygon()	多边形
drawPolyline()	多折线
drawRect()	矩形
drawRects()	多个矩形
drawRoundRect()	圆角矩形
drawText()	文字
drawTiledPixmap()	平铺图像
drawLineSegments()	绘制折线

基本图形绘制示例

重载paintEvent函数

```
void MainWindow::paintEvent(QPaintEvent* e)
{
    QPainter p(this);
    p.setRenderHint(QPainter::Antialiasing);
    p.setPen(Qt::black);
    p.setBrush(Qt::green);
    p.drawRect(10, 30, width()-20, height()-60);
    p.setBrush(Qt::yellow);
    p.drawEllipse(20, 40, width()-40, height()-80);
}
```



画笔

画笔

- 画笔的属性包括线型、线宽、颜色等。画笔属性可以在构造函数中指定，也可以使用setStyle()，setWidth()，setBrush()，setCapStyle()，setJoinStyle()等函数设定
- Qt中，使用Qt::PenStyle定义了6种画笔风格，分别是
  - Qt::SolidLine，Qt::DashLine，Qt::DotLine，Qt::DashDotLine，Qt::DashDotDotLine，Qt::CustomDashLine。
  - 自定义线风格(Qt::CustomDashLine)，需要使用QPen的setDashPattern()函数来设定自定义风格。

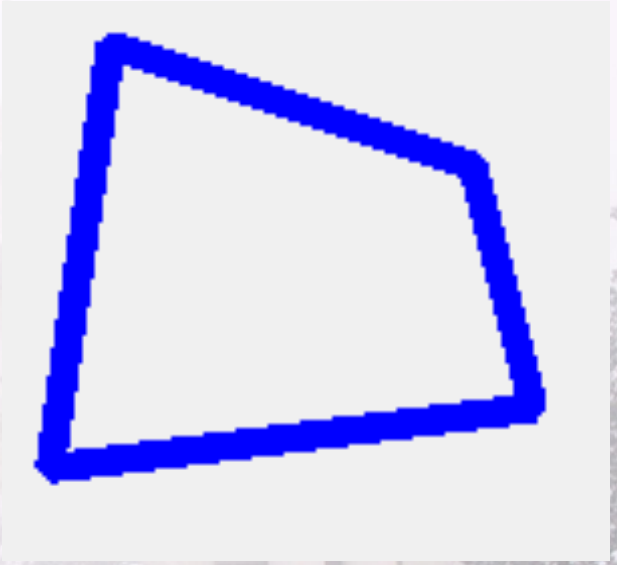
画笔

- **端点风格(cap style)**
  - 端点风格决定了线的端点样式，只对线宽大于1的线有效。
  - Qt定义了包括三种端点风格枚举类型Qt::PenCapStyle
    - Qt::SqureCap (default)
    - Qt::FlatCap
    - Qt::RoundCap

- **连接风格(Join style)**
  - 连接风格是两条线如何连接，连接风格对线宽大于等于1的线有效。
  - Qt定义了包括四种连接方式的枚举类型Qt::PenJoinStyle
    - Qt::BevelJoin (default)
    - Qt::MiterJoin
    - Qt::RoundJoin
    - Qt::SvgMiterJoin

画笔示例

```
static const QPointF points[4] = {
     QPointF(10.0, 80.0),
     QPointF(20.0, 10.0),
     QPointF(80.0, 30.0),
     QPointF(90.0, 70.0)
 };

 QPainter painter(this);
 QPen pen(Qt::blue, 5);
 painter.setPen(pen);
 painter.drawPolygon(points, 4);
```



画刷

画刷

- 在Qt中，通过填充颜色和风格(填充模式)设置画刷。
- 颜色使用QColor类表示，QColor支持RGB，HSV，CMYK颜色模型。
  - RGB颜色由红绿蓝三种基色混合而成

```
QColor( int r, int g, int b, int a )
```

    - r（red）, g（green）, b（blue）, a（alpha）的取值范围为0-255
    - Alpha控制透明度，255表示不透明，0表示完全透明
    - QColor对象还可以用SVG1.0中定义的任何颜色名作为参数初始化
  - HSV/HSL模型比较符合人对颜色的感觉，由色调(0-359)，饱和度(0-255)，亮度(0-255)组成，主要用于颜色选择器。
  - CMYK模型由青，洋红，黄，黑四种基色组成。主要用于打印机等硬件设备。每个颜色分量的取值是0-255。
- 基本模式填充包括有各种点、线组合的模式。

QColor

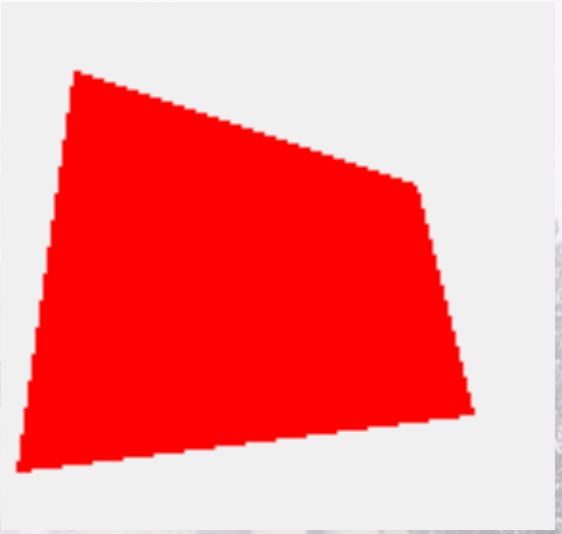
- Qt预定义枚举类型的颜色，如Qt::black

white	black	cyan	darkCyan
red	darkRed	magenta	darkMagenta
green	darkGreen	yellow	darkYellow
blue	darkBlue	gray	darkGray
lightGray			

- 颜色可以通过如下函数进行微调
  - QColor::lighter(int factor)
  - QColor::darker(int factor)

画刷示例

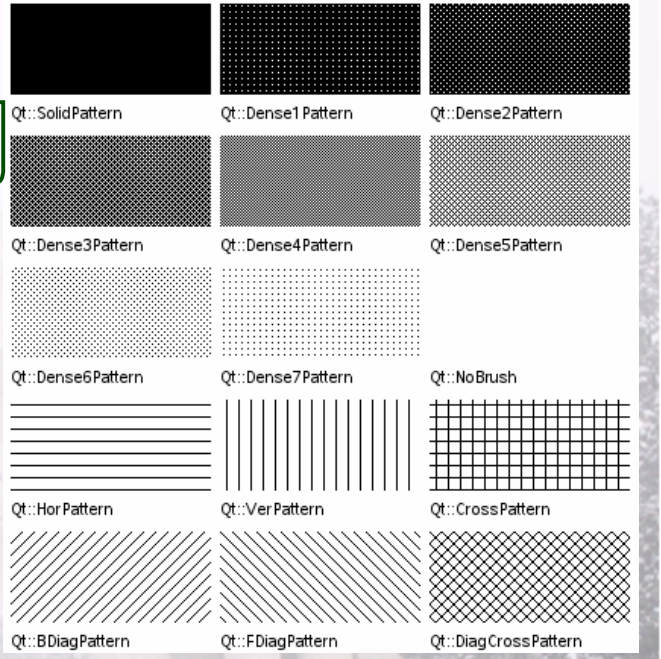
```
static const QPointF points[4] = {
     QPointF(10.0, 80.0),
     QPointF(20.0, 10.0),
     QPointF(80.0, 30.0),
     QPointF(90.0, 70.0)
 };

 QPainter painter(this);
 painter.setPen(Qt::NoPen);
 painter.setBrush(Qt::red);
 painter.drawPolygon(points, 4);
```



模式画刷与带纹理画刷

- 模式画刷

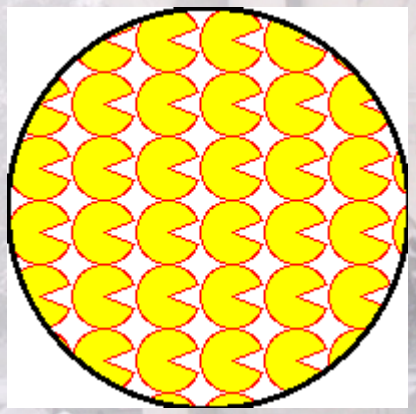
```
QBrush(const QColor &color, Qt::BrushStyle style)
```



- 带纹理画刷
  - 以QPixmap为参数
  - 如果使用黑白的pixmap，则用画刷颜色
  - 如果使用彩色pixmap，则用pixmap的颜色

```
QBrush( const QPixmap &pixmap )
```

```
QPixmap pacPixmap("pacman.png");

painter.setPen(QPen(Qt::black, 3));
painter.setBrush(pacPixmap);
painter.drawEllipse(rect());
```



绘制文本

文本绘制

- 使用QPainter进行文本绘制
  - 基本文本绘制

```
drawText( QPoint, QString )
```

  - 带选项的文本绘制

```
drawText( QRect, QString, QTextOptions )
```

  - 带返回信息的文本绘制

```
drawText( QRect, flags, QString, QRect* )
```

- 字体：QFont类
  - Font family，Size，Bold / Italic / Underline / Strikeout / …
  - 通过QApplication::setFont() 设置应用程序的默认字体
  - 字体信息可以通过QFontInfo取出，并可用QFontMetrics取得字体的相关数据

文本绘制举例

```
QPainter p(this);

QFont font("Helvetica");
p.setFont(font);
p.drawText(20, 20, 120, 20, 0, "Hello World!");

font.setPixelSize(10);
p.setFont(font);
p.drawText(20, 40, 120, 20, 0, "Hello World!");

font.setPixelSize(20);
p.setFont(font);
p.drawText(20, 60, 120, 20, 0, "Hello World!");

QRect r;
p.setPen(Qt::red);
p.drawText(20, 80, 120, 20, 0, "Hello World!", &r);
```

Hello World!

Hello World!

Hello World!

Hello World!

r返回文本
外边框的矩形区域

Font Family/Size

- 在QFont构造函数中指定字体
  - 如果没有对应的字体，Qt将寻找一种最接近的已安装字体
  - 如果字体中不存在某个字符，则绘制一个空心的正方形

```
QFont font("Helvetica");
font.setFamily("Times");
```

- 得到可用字体列表

```
QFontDatabase database;
QStringList families = database.families();
```

- 字体尺寸可以用像素尺寸（pixel size）或点阵尺寸（point size）

```
font.setPointSize(14); // 14 points high
// depending on the paint device's dpi
font.setPixelSize(10); // 10 pixels high
```

字体效果和测量文本大小(选讲)

- 字体效果

Hello Qt!

Hello Qt!

Hello Qt!

~~Hello Qt!~~

Hello Qt!

Hello Qt!

Normal
bold
Italic
strike out
underline
overline

- QWidget::font函数和QPainter::font函数可以取得现有字体

```
QFont tempFont = w->font();
tempFont.setBold( true );
w->setFont( tempFont );
```

- QFontMetrics可用于测量文本和font的大小
- boundingRect函数可用于测量文本块的大小

```
QImage image(200, 200, QImage::Format_ARGB32);
QPainter painter(&image);
QFontMetrics fm(painter.font(), &image);
qDebug("width: %d", fm.width("Hello Qt!"));
qDebug("height: %d", fm.boundingRect(0, 0, 200, 0,
    Qt::AlignLeft | Qt::TextWordWrap, "Hello Qt!").height());
```

中文显示问题(选讲)

使用QTextCodec类

In main.cpp

```
#include <QTextCodec>
…
QTextCodec *codec = QTextCodec::codecForName("GB2312");
// or QTextCodec *codec = QTextCodec::codecForName("UTF-8");
QTextCodec::setCodecForLocale(codec);
```

In mainwindow.cpp

```
int ret = QMessageBox::warning(0, tr(“App"), tr("您真的想要退
   出? "), QMessageBox::Yes | QMessageBox::No);
```

图像处理

图像处理

- Qt提供了4个处理图像的类：QImage，QPixmap，QBitmap，QPicture
  - QImage优化了I/O操作，可以直接存取操作像素数据。
  - QPixmap优化了在屏幕上显示图像的性能。
  - QBitmap从QPixmap继承，只能表示两种颜色。
  - QPicture是可以记录和重启QPrinter命令的类。
- 可以在QImage和QPixmap之间转换

```
QImage QPixmap::toImage();
QPixmap QPixmap::fromImage( const QImage& );
```

图像文件的读写与绘制

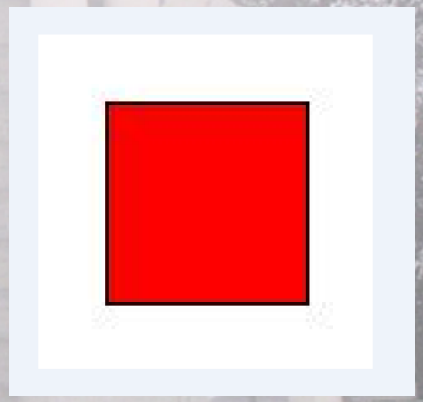
- 读写图像文件
  - 如下代码使用QImageReader和QImageWriter类进行，可通过文件扩展名自动确定文件格式

```
QPixmap pixmap( "image.png" );
pixmap.save( "image.jpeg" );
```

```
QImage image( "image.png" );
image.save( "image.jpeg" );
```

- 作为QPaintDevice的子类， QImage可以由QPainter绘制

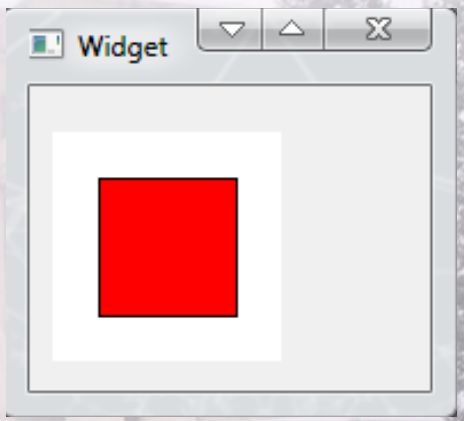
```
QImage image( 100, 100, QImage::Format_ARGB32 );
QPainter painter(&image);
painter.setBrush(Qt::red);
painter.fillRect( image.rect(), Qt::white );
painter.drawRect(
    image.rect().adjusted( 20, 20, -20, -20 ) );
image.save( "image.jpeg" );
```



通过QPixmap进行屏幕绘制

- QPixmap主要用于屏幕绘制

```
void MyWidget::imageChanged(const QImage &image)
{
    pixmap = QPixmap::fromImage( image );
    update();
}

void MyWidget::paintEvent( QPaintEvent* )
{
    QPainter painter( this );
    painter.drawPixmap( 10, 20, pixmap );
}
```



坐标系统与坐标变换

坐标系统

- Qt绘图设备默认坐标原点是左上角，X轴向右增长，Y轴向下增长，
  - 默认的单位在基于像素的设备上是像素，可以通过window、width和height函数取得绘制窗口的尺寸，并在窗口上直接绘图
  - 在打印机设备上默认单位是1/72英寸(0.35毫米)
- QPainter的逻辑坐标与QPainterDevice的物理坐标之间的映射由QPainter的变换矩阵worldMatrix()、视口viewport() 和窗口window()处理
  - 未进行坐标变换的情况下，逻辑坐标和物理坐标是一致的，可以直接进行绘图

坐标值的表示方法

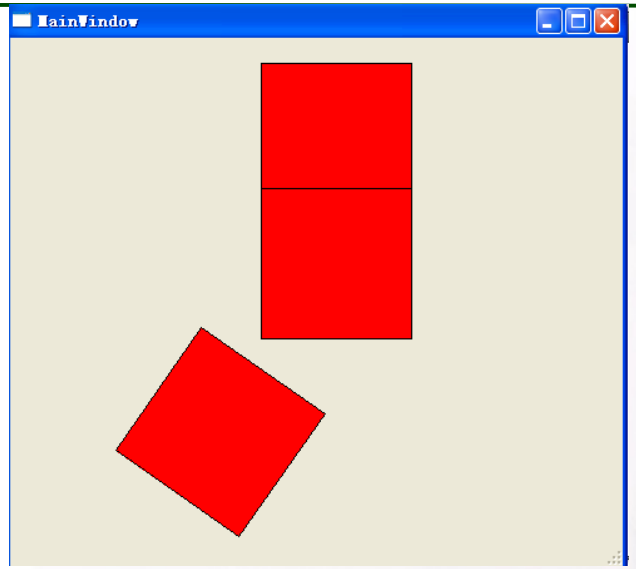
- 使用QPoint, QSize和QRect表示坐标值和区域
  - QPoint: point(x, y)
  - QSize: size(width, height)
  - QRect: point 和 size (x, y, width, height)
- QPointF/QSizeF/QRectF用于表示浮点数坐标

QPainter的坐标变换

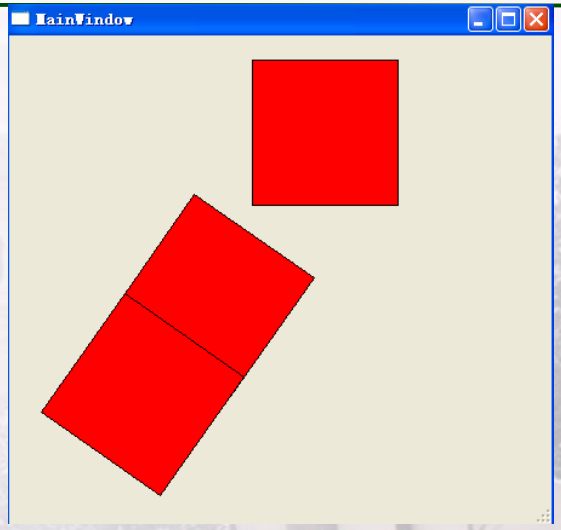
- QPainter支持坐标变换
  - QPainter::scale()函数：比例变换
  - QPainter::rotate()函数：旋转变换
  - QPainter::translate()函数：平移变换
  - QPainter::shear()函数：图形进行扭曲变换
- 变换操作对应的变换矩阵可通过QPainter::wordMatrix()函数取出
- 不同的变换矩阵可以使用堆栈保存和恢复
  - 用QPainter::save()保存变换矩阵到堆栈，用QPainter::restore()函数将其弹出堆栈

坐标变换

- 坐标变换的顺序很重要
- 在做平移变换、旋转变换和扭曲变换时，原点也很重要




```
p.setBrush(Qt::red);
p.drawRect(200, 20, 120, 120);
p.translate(0, 100);
p.drawRect(200, 20, 120, 120);
p.rotate(35);
p.drawRect(200, 20, 120, 120);
```




```
p.setBrush(Qt::red);
p.drawRect(200, 20, 120, 120);
p.rotate(35);
p.drawRect(200, 20, 120, 120);
p.translate(0, 100);
p.drawRect(200, 20, 120, 120);
```

坐标变换的保存和恢复

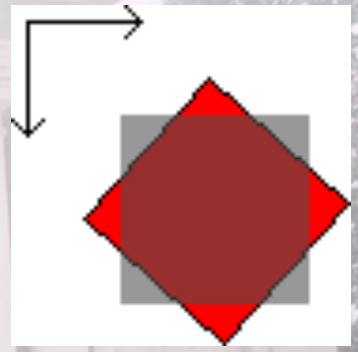
- 通过save和restore函数，可以将坐标变换的状态保存和恢复

```
QPoint rotCenter(50, 50);
qreal angle = 42;

p.save();
p.translate(rotCenter);
p.rotate(angle);
p.translate(-rotCenter);

p.setBrush(Qt::red);
p.setPen(Qt::black);
p.drawRect(25,25, 50, 50);

p.restore();

p.setPen(Qt::NoPen);
p.setBrush(QColor(80, 80, 80, 150));
p.drawRect(25,25, 50, 50);
```

应用变换

画红色矩形

画灰色矩形



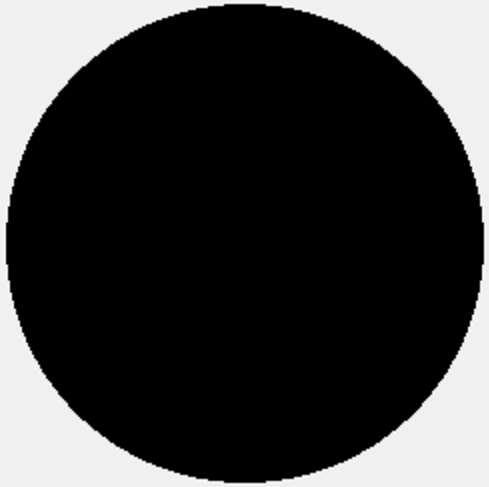
绘图举例：表盘

表盘

- 自定义绘制
- 可以与键盘和鼠标交互

表盘

画表盘的背景



```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);

    int extent;
    if (width()>height())
        extent = height()-20;
    else
        extent = width()-20;

    p.translate((width()-extent)/2, (height()-extent)/2);

    p.setPen(Qt::white);
    p.setBrush(Qt::black);

    p.drawEllipse(0, 0, extent, extent);

    ...
```

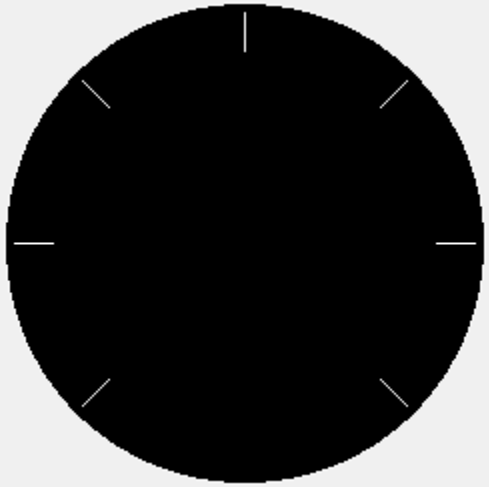
将油表放在中心位置

画背景圆形

表盘

画表盘的刻度

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    ...

    p.translate(extent/2, extent/2);
    for(int angle=0; angle<=270; angle+=45)
    {
        p.save();
        p.rotate(angle+135);
        p.drawLine(extent*0.4, 0, extent*0.48, 0);
        p.restore();
    }

   ...
```



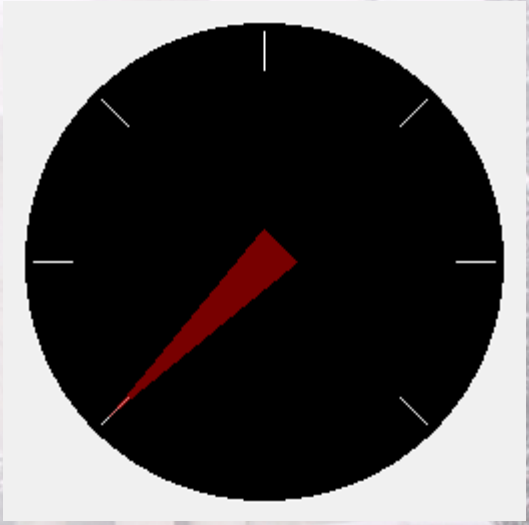
注意save和restore函数

简单调用rotate(45)会增大舍入误差

表盘

画表盘的指针

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    ...

    p.rotate(m_value+135);
    QPolygon polygon;
    polygon << QPoint(-extent*0.05, extent*0.05)
            << QPoint(-extent*0.05, -extent*0.05)
            << QPoint(extent*0.46, 0);
    p.setPen(Qt::NoPen);
    p.setBrush(QColor(255,0,0,120));
    p.drawPolygon(polygon);
}
```



响应事件

- 除了paintEvent，还有
  - 键盘事件
  - 鼠标事件
  - 窗口事件
  - 定时器事件
- 键盘事件：
  - 重写keyPressEvent
  - 键按下时响应
  - 将未处理的按键传给基类处理

```
void CircularGauge::keyPressEvent(QKeyEvent *ev)
{
    switch(ev->key())
    {
    case Qt::Key_Up:
    case Qt::Key_Right:
        setValue(value()+1);
        break;
    case Qt::Key_Down:
    case Qt::Key_Left:
        setValue(value()-1);
        break;
    case Qt::Key_PageUp:
        setValue(value()+10);
        break;
    case Qt::Key_PageDown:
        setValue(value()-10);
        break;
    default:
        QWidget::keyPressEvent(ev);
    }
}
```

响应鼠标事件

- 鼠标事件通过重写如下函数来处理
  - mousePressEvent和mouseReleaseEvent
  - mouseMoveEvent：除非mouseTracking为真，否则只有鼠标按键按下时才被调用
- setValueFromPos是一个私有函数，将点转换为角度

```
void CircularGauge::mousePressEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
void CircularGauge::mouseReleaseEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
void CircularGauge::mouseMoveEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
```

为表盘添加事件过滤器

- 按键0时，时油表指向0

```
class KeyboardFilter : public QObject ...

bool KeyboardFilter::eventFilter(QObject *o, QEvent *ev)
{
    if (ev->type() == QEvent::KeyPress)
        if (QKeyEvent *ke = static_cast<QKeyEvent*>(ev))
            if (ke->key() == Qt::Key_0)
                if (o->metaObject()->indexOfProperty("value") != -1 )
                {
                    o->setProperty("value", 0);
                    return true;
                }

    return false;
}
```

返回true，停止
对该事件的响应

- 安装过滤器

```
CircularGauge w;
KeyboardFilter filter;
w.installEventFilter(&filter);
```

加速绘制（选讲）

- 用户调用update或repaint函数时，可以调用无参数版本
- 然而，用户一般知悉哪些区域需要重绘，因此可以调用有参数版本，指定重绘区域，该重绘区域被封装到QPaintEvent类对象中。

```
void update(int x, int y, int w, int h)
void update(const QRect &rect)
void update(const QRegion &rgn)
```

- paintEvent函数有一个QPaintEvent类对象作为参数，在paintEvent函数中重绘时，可以通过QPaintEvent类取得指定区域，计算并重绘指定区域内部的图形，不需要重绘整个窗口，从而加速绘制过程
- QPaintEvent类有如下方法取得重绘区域
  - QRect rect()：返回需要重绘的矩形
  - QRegion region()：返回需要重绘的区域

谢谢！

Q&A？