## 课程目标

- Qt基础窗口类
- QMainWindow窗口类
- QWidget窗口类
- QDialog窗口类
    - QMessageBox
    - QFileDialog
    - QFontDialog
    - QColorDialog
    - QInputDialog
    - QProgressDialog
- Qt对话框的模态和非模态实现

## 课程实验

- Qt实现QWidget和QMainWindow窗口
- 实现QDialog窗口
- QDialog对话框子类功能实现
- QMessageBox实现
- QFileDialog实现
- QFontDialog实现
- QColorDialog实现
- QInputDialog实现
- QProgressDialog实现

## 课堂引入

Qt可以根据具体的应用需求选择合适的窗口类。这些窗口类为Qt应用程序提供了丰富的用户界面设计和交互功能。

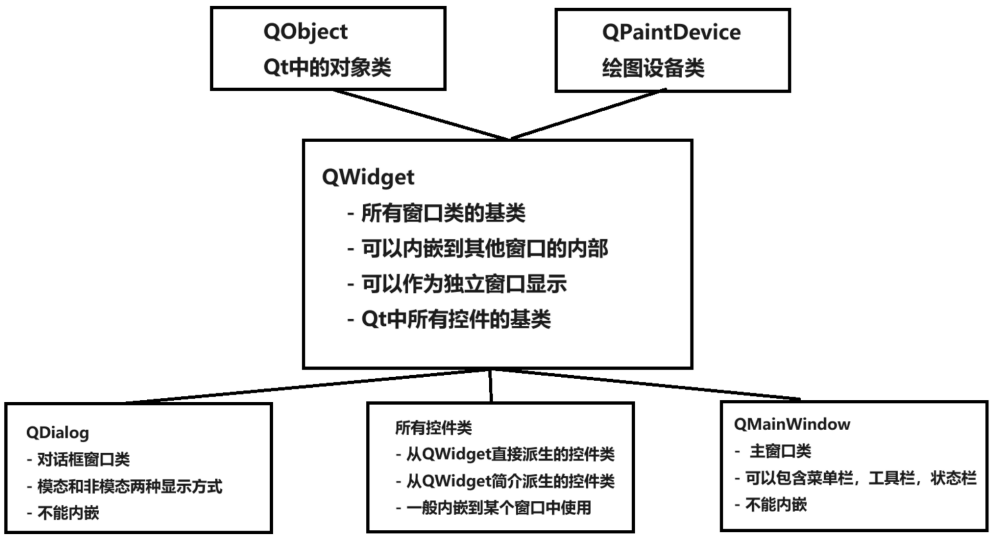
> Qt常用窗口类主要内容包括：
>
> - 窗口类基类QWidget
> - 对话框基类QDialog
> - 带菜单栏工具栏状态栏的QMainWindow
> - 消息对话框QMessageBox
> - 文件对话框QFileDialog
> - 字体对话框QFontDialog
> - 颜色对话框QColorDialog
> - 输入型对话框QInputDialog
> - 进度条对话框QProgressDialog

Qt可以根据具体的应用需求选择合适的窗口类。这些窗口类为Qt应用程序提供了丰富的用户界面设计和交互功能。

# 授课进程

## 一 QWidget类

QWidget类是所有窗口类的父类(控件类是也属于窗口类)，并且QWidget类的父类的QObject，也就意味着所有的窗口类对象只要指定了父对象，都可以实现内存资源的自动回收。




### 1.1 设置父对象

```
// 构造函数
QWidget::QWidget(QWidget *parent = nullptr, Qt::WindowFlags f =
Qt::WindowFlags());

// 公共成员函数
// 给当前窗口设置父对象
void QWidget::setParent(QWidget *parent);
void QWidget::setParent(QWidget *parent, Qt::WindowFlags f);
// 获取当前窗口的父对象, 没有父对象返回 nullptr
QWidget *QWidget::parentWidget() const;
```

### 1.2 窗口位置

```
//------------- 窗口位置 -------------
// 得到相对于当前窗口父窗口的几何信息,边框也被计算在内
QRect QWidget::frameGeometry() const;
// 得到相对于当前窗口父窗口的几何信息, 不包括边框
const QRect &geometry() const;
// 设置当前窗口的几何信息(位置和尺寸信息), 不包括边框
void setGeometry(int x, int y, int w, int h);
void setGeometry(const QRect &);

// 移动窗口, 重新设置窗口的位置
void move(int x, int y);
void move(const QPoint &);
```

窗口位置设定和位置获取的测试代码如下:

```
// 获取当前窗口的位置信息
void MainWindow::on_positionBtn_clicked() {
    QRect rect = this->frameGeometry();
    qDebug() << "左上角: " << rect.topLeft()
             << "右上角: " << rect.topRight()
             << "左下角: " << rect.bottomLeft()
             << "右下角: " << rect.bottomRight()
             << "宽度: " << rect.width()
             << "高度: " << rect.height();
}

// 重新设置当前窗口的位置以及宽度, 高度
void MainWindow::on_geometryBtn_clicked() {
    int x = 100 + rand() % 500;
    int y = 100 + rand() % 500;
    int width = this->width() + 10;
    int height = this->height() + 10;
    setGeometry(x, y, width, height);
}

// 通过 move() 方法移动窗口
void MainWindow::on_moveBtn_clicked() {
    QRect rect = this->frameGeometry();
    move(rect.topLeft() + QPoint(10, 20));
}
```

### 1.3 窗口尺寸

```
//------------- 窗口尺寸 -------------
// 获取当前窗口的尺寸信息
QSize size() const
// 重新设置窗口的尺寸信息
void resize(int w, int h);
void resize(const QSize &);
// 获取当前窗口的最大尺寸信息
QSize maximumSize() const;
// 获取当前窗口的最小尺寸信息
QSize minimumSize() const;
// 设置当前窗口固定的尺寸信息
void QWidget::setFixedSize(const QSize &s);
void QWidget::setFixedSize(int w, int h);
// 设置当前窗口的最大尺寸信息
void setMaximumSize(const QSize &);
void setMaximumSize(int maxw, int maxh);
// 设置当前窗口的最小尺寸信息
void setMinimumSize(const QSize &);
void setMinimumSize(int minw, int minh);


// 获取当前窗口的高度
int height() const;
// 获取当前窗口的最小高度
int minimumHeight() const;
// 获取当前窗口的最大高度
int maximumHeight() const;
```

```
// 给窗口设置固定的高度
void QWidget::setFixedHeight(int h);
// 给窗口设置最大高度
void setMaximumHeight(int maxh);
// 给窗口设置最小高度
void setMinimumHeight(int minh);

// 获取当前窗口的宽度
int width() const;
// 获取当前窗口的最小宽度
int minimumWidth() const;
// 获取当前窗口的最大宽度
int maximumWidth() const;
// 给窗口设置固定宽度
void QWidget::setFixedWidth(int w);
// 给窗口设置最大宽度
void setMaximumWidth(int maxw);
// 给窗口设置最小宽度
void setMinimumWidth(int minw);
```

### 1.4 窗口标题和图标

```
//------------- 窗口图标 -------------
// 得到当前窗口的图标
QIcon windowIcon() const;
// 构造图标对象, 参数为图片的路径
QIcon::QIcon(const QString &fileName);
// 设置当前窗口的图标
void setWindowIcon(const QIcon &icon);

//------------- 窗口标题 -------------
// 得到当前窗口的标题
QString windowTitle() const;
// 设置当前窗口的标题
void setWindowTitle(const QString &);
```

### 1.5 信号

```
// QWidget::setContextMenuPolicy(Qt::ContextMenuPolicy policy);
// 窗口的右键菜单策略 contextMenuPolicy() 参数设置为 Qt::CustomContextMenu, 按下鼠标右键发射该信号
[signal] void QWidget::customContextMenuRequested(const QPoint &pos);
// 窗口图标发生变化, 发射此信号
[signal] void QWidget::windowIconChanged(const QIcon &icon);
// 窗口标题发生变化, 发射此信号
[signal] void QWidget::windowTitleChanged(const QString &title);
```

### 1.6 槽函数

```
//------------- 窗口显示 -------------
// 关闭当前窗口
[slot] bool QWidget::close();
// 隐藏当前窗口
[slot] void QWidget::hide();
```

```
// 显示当前创建以及其子窗口
[slot] void QWidget::show();
// 全屏显示当前窗口，只对windows有效
[slot] void QWidget::showFullScreen();
// 窗口最大化显示，只对windows有效
[slot] void QWidget::showMaximized();
// 窗口最小化显示，只对windows有效
[slot] void QWidget::showMinimized();
// 将窗口回复为最大化/最小化之前的状态，只对windows有效
[slot] void QWidget::showNormal();

//------------- 窗口状态 -------------
// 判断窗口是否可用
bool QWidget::isEnabled() const; // 非槽函数
// 设置窗口是否可用，不可用窗口无法接收和处理窗口事件
// 参数true->可用，false->不可用
[slot] void QWidget::setEnabled(bool);
// 设置窗口是否可用，不可用窗口无法接收和处理窗口事件
// 参数true->不可用，false->可用
[slot] void QWidget::setDisabled(bool disable);
// 设置窗口是否可见，参数为true->可见，false->不可见
[slot] virtual void QWidget::setVisible(bool visible);
```

## 二 QDialog类

### 2.1 常用API

对话框类是QWidget类的子类,处理继承自父类的属性之外,还有一些自己所特有的属性,常用的一些API函数如下:

```
// 构造函数
QDialog::QDialog(QWidget *parent = nullptr, Qt::WindowFlags f =
Qt::WindowFlags());

// 模态显示窗口
[virtual slot] int QDialog::exec();
// 隐藏模态窗口，并且解除模态窗口的阻塞，将 exec() 的返回值设置为 QDialog::Accepted
[virtual slot] void QDialog::accept();
// 隐藏模态窗口，并且解除模态窗口的阻塞，将 exec() 的返回值设置为 QDialog::Rejected
[virtual slot] void QDialog::reject();
// 关闭对话框并将其结果代码设置为r。finished()信号将发出r;
// 如果r是QDialog::Accepted 或 QDialog::Rejected，则还将分别发出accept()或Rejected()
信号。
[virtual slot] void QDialog::done(int r);

[signal] void QDialog::accepted();
[signal] void QDialog::rejected();
[signal] void QDialog::finished(int result);
```

### 2.2 常用使用方法

> 场景介绍:
>
> - 有两个窗口, 主窗口和一个对话框子窗口
> - 对话框窗口先显示,根据用户操作选择是否显示主窗口

关于对话框窗口类的操作

```
// 对话框窗口中三个普通按钮按下之后对应的槽函数
void MyDialog::on_acceptBtn_clicked() {
    this->accept();  // exec()函数返回值为QDialog::Accepted
}

void MyDialog::on_rejectBtn_clicked() {
    this->reject();  // exec()函数返回值为QDialog::Rejected
}

void MyDialog::on_donBtn_clicked() {
    // exec()函数返回值为 done() 的参数, 并根据参数发射出对应的信号
    this->done(666);
}
```

根据用户针对对话框窗口的按钮操作, 进行相应的逻辑处理。

```
// 创建对话框对象
MyDialog dlg;
int ret = dlg.exec();
if(ret == QDialog::Accepted) {
    qDebug() << "accept button clicked...";
    // 显示主窗口
    MainWindow* w = new MainWindow;
    w->show();
}
else if(ret == QDialog::Rejected) {
    qDebug() << "reject button clicked...";
    // 不显示主窗口
    ......
    ......
}
else {
    // ret == 666
    qDebug() << "done button clicked...";
    // 根据需求进行逻辑处理
    ......
    ......
}
```

## 三 QDialog的子类

### 3.1 QMessageBox

QMessageBox 对话框类是 QDialog 类的子类, 通过这个类可以显示一些简单的提示框, 用于展示警告、错误、问题等信息。关于这个类我们只需要掌握一些静态方法的使用就可以了。

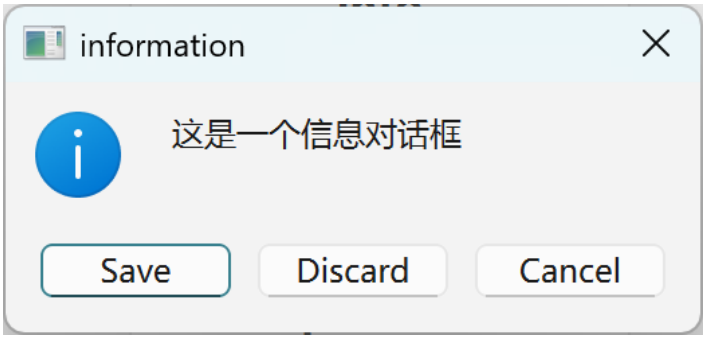
#### 3.1.1 API - 静态函数

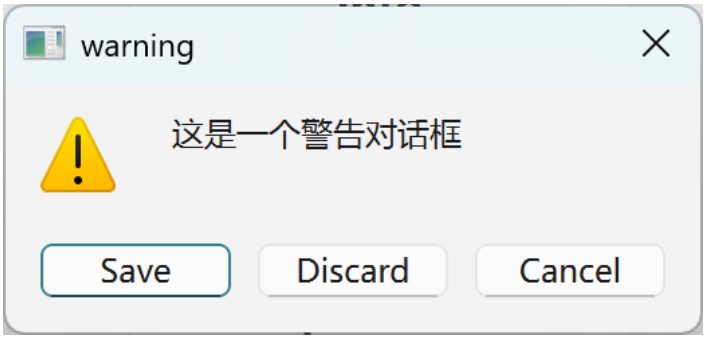
```
// 显示一个模态对话框, 将参数 text 的信息展示到窗口中
[static] void QMessageBox::about(QWidget *parent, const QString &title, const
QString &text);

/*
参数:
- parent: 对话框窗口的父窗口
- title: 对话框窗口的标题
- text: 对话框窗口中显示的提示信息
- buttons: 对话框窗口中显示的按钮(一个或多个)
- defaultButton
    1. defaultButton指定按下Enter键时使用的按钮。
    2. defaultButton必须引用在参数 buttons 中给定的按钮。
    3. 如果defaultButton是QMessageBox::NoButton, QMessageBox会自动选择一个合适的默认
值。
*/
// 显示一个信息模态对话框
[static] QMessageBox::StandardButton QMessageBox::information(
           QWidget *parent, const QString &title,
           const QString &text,
           QMessageBox::StandardButtons buttons = Ok,
           QMessageBox::StandardButton defaultButton = NoButton);

// 显示一个错误模态对话框
[static] QMessageBox::StandardButton QMessageBox::critical(
           QWidget *parent, const QString &title,
           const QString &text,
           QMessageBox::StandardButtons buttons = Ok,
           QMessageBox::StandardButton defaultButton = NoButton);

// 显示一个问题模态对话框
[static] QMessageBox::StandardButton QMessageBox::question(
           QWidget *parent, const QString &title,
           const QString &text,
           QMessageBox::StandardButtons buttons = StandardButtons(Yes | No),
           QMessageBox::StandardButton defaultButton = NoButton);

// 显示一个警告模态对话框
[static] QMessageBox::StandardButton QMessageBox::warning(
           QWidget *parent, const QString &title,
           const QString &text,
           QMessageBox::StandardButtons buttons = Ok,
           QMessageBox::StandardButton defaultButton = NoButton);
```

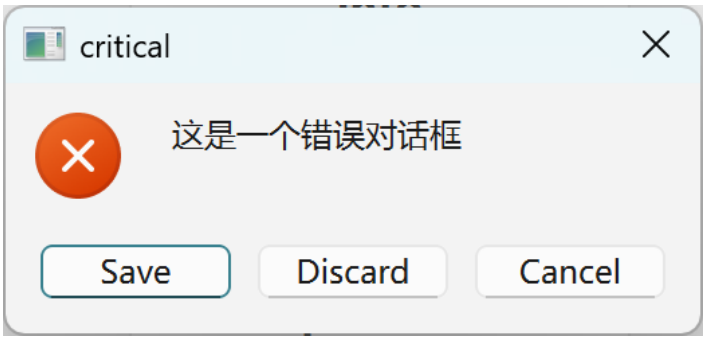
### 3.1.2 测试代码

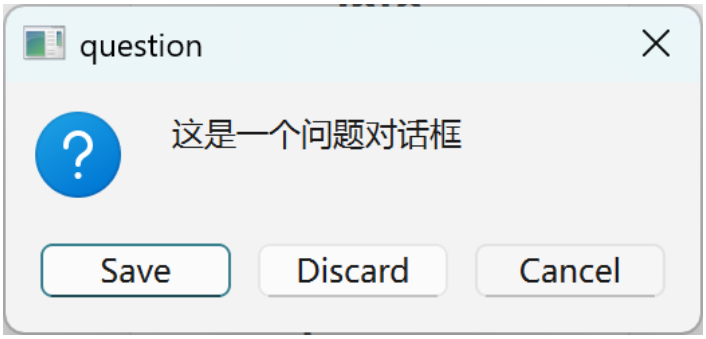
```
void MainWindow::on_msgbox_clicked() {
    QMessageBox::about(this, "about",  "这是一个简单的消息提示框!!!");
    QMessageBox::critical(this, "critical", "这是一个错误对话框-critical...");
    int ret = QMessageBox::question(this, "question",
              "你要保存修改的文件内容吗???",
               QMessageBox::Save|QMessageBox::Cancel,
               QMessageBox::Cancel);
    if(ret == QMessageBox::Save) {
        QMessageBox::information(this, "information", "恭喜你保存成功了, o(*￣▽￣*)o!!!");
    }
    else if(ret == QMessageBox::Cancel) {
        QMessageBox::warning(this, "warning", "你放弃了保存, ╥﹏╥ !!!");
    }
}
```

得到的对话框窗口效果如下图:






### 3.2 QFileDialog

QFileDialog 对话框类是 QDialog 类的子类, 通过这个类可以选择要打开/保存的文件或者目录。关于这个类我们只需要掌握一些静态方法的使用就可以了。

#### 3.2.1 API - 静态函数

```
/*
通用参数:
  - parent: 当前对话框窗口的父对象也就是父窗口
  - caption: 当前对话框窗口的标题
  - dir: 当前对话框窗口打开的默认目录
  - options: 当前对话框窗口的一些可选项,枚举类型, 一般不需要进行设置, 使用默认值即可
  - filter: 过滤器, 在对话框中只显示满足条件的文件, 可以指定多个过滤器, 使用 ;; 分隔
    - 样式举例:
    - Images (*.png *.jpg)
    - Images (*.png *.jpg);;Text files (*.txt)
  - selectedFilter: 如果指定了多个过滤器, 通过该参数指定默认使用哪一个, 不指定默认使用第一个过滤器
*/
// 打开一个目录, 得到这个目录的绝对路径
[static] QString QFileDialog::getExistingDirectory(
                    QWidget *parent = nullptr,
                    const QString &caption = QString(),
                    const QString &dir = QString(),
                    QFileDialog::Options options = ShowDirsOnly);

// 打开一个文件, 得到这个文件的绝对路径
[static] QString QFileDialog::getOpenFileName(
                    QWidget *parent = nullptr,
                    const QString &caption = QString(),
                    const QString &dir = QString(),
                    const QString &filter = QString(),
                    QString *selectedFilter = nullptr,
                    QFileDialog::Options options = Options());

// 打开多个文件, 得到这多个文件的绝对路径
[static] QStringList QFileDialog::getOpenFileNames(
                    QWidget *parent = nullptr,
                    const QString &caption = QString(),
                    const QString &dir = QString(),
                    const QString &filter = QString(),
                    QString *selectedFilter = nullptr,
                    QFileDialog::Options options = Options());
```

```
// 打开一个目录，使用这个目录来保存指定的文件
[static] QString QFileDialog::getSaveFileName(
                  QWidget *parent = nullptr,
                  const QString &caption = QString(),
                  const QString &dir = QString(),
                  const QString &filter = QString(),
                  QString *selectedFilter = nullptr,
                  QFileDialog::Options options = Options());
```

#### 3.2.2 测试代码

打开一个已存在的本地目录

```
void MainWindow::on_filedlg_clicked() {
    QString dirName = QFileDialog::getExistingDirectory(this, "打开目录",
"e:\\temp");
    QMessageBox::information(this, "打开目录", "您选择的目录是: " + dirName);
}
```

打开一个本地文件

```
void MainWindow::on_filedlg_clicked() {
    QString arg("Text files (*.txt)");
    QString fileName = QFileDialog::getOpenFileName(
              this, "Open File", "e:\\temp",
              "Images (*.png *.jpg);;Text files (*.txt)", &arg);
    QMessageBox::information(this, "打开文件", "您选择的文件是: " + fileName);
}
```

打开多个本地文件

```
void MainWindow::on_filedlg_clicked() {
    QStringList fileNames = QFileDialog::getOpenFileNames(
              this, "Open File", "e:\\temp",
              "Images (*.png *.jpg);;Text files (*.txt)");
    QString names;
    for(int i=0; i<fileNames.size(); ++i) {
        names += fileNames.at(i) + " ";
    }
    QMessageBox::information(this, "打开文件(s)", "您选择的文件是: " + names);
}
```

打开保存文件对话框

```
void MainWindow::on_filedlg_clicked() {
    QString fileName = QFileDialog::getSaveFileName(this, "保存文件", "e:\\temp");
    QMessageBox::information(this, "保存文件", "您指定的保存数据的文件是: " +
fileName);
}
```

### 3.3 QFontDialog

QFontDialog类是QDialog的子类, 通过这个类我们可以得到一个进行字体属性设置的对话框窗口, 和前边介绍的对话框类一样, 我们只需要调用这个类的静态成员函数就可以得到想要的窗口了。

#### 3.3.1 QFont 字体类

关于字体的属性信息, 在QT框架中被封装到了一个叫QFont的类中, 下边为大家介绍一下这个类的API, 了解一下关于这个类的使用。

```
// 构造函数
  QFont::QFont();
  /*
  参数:
    - family: 本地字库中的字体名, 通过 office 等文件软件可以查看
    - pointSize: 字体的字号
    - weight: 字体的粗细, 有效范围为 0 ~ 99
    - italic: 字体是否倾斜显示, 默认不倾斜
  */
  QFont::QFont(const QString &family, int pointSize = -1, int weight = -1, bool
italic = false);

  // 设置字体
  void QFont::setFamily(const QString &family);
  // 根据字号设置字体大小
  void QFont::setPointSize(int pointSize);
  // 根据像素设置字体大小
  void QFont::setPixelSize(int pixelSize);
  // 设置字体的粗细程度, 有效范围: 0 ~ 99
  void QFont::setWeight(int weight);
  // 设置字体是否加粗显示
  void QFont::setBold(bool enable);
  // 设置字体是否要倾斜显示
  void QFont::setItalic(bool enable);

  // 获取字体相关属性(一般规律: 去掉设置函数的 set 就是获取相关属性对应的函数名)
  QString QFont::family() const;
  bool QFont::italic() const;
  int QFont::pixelSize() const;
  int QFont::pointSize() const;
  bool QFont::bold() const;
  int QFont::weight() const;
```

如果一个QFont对象被创建, 并且进行了初始化, 我们可以将这个属性设置给某个窗口, 或者设置给当前应用程序对象。

```
// QWidget 类
// 得到当前窗口使用的字体
const QWidget::QFont& font() const;
// 给当前窗口设置字体，只对当前窗口类生效
void QWidget::setFont(const QFont &);

// QApplication 类
// 得到当前应用程序对象使用的字体
[static] QFont QApplication::font();
// 给当前应用程序对象设置字体，作用于当前应用程序的所有窗口
[static] void QApplication::setFont(const QFont &font, const char *className =
nullptr);
```

### 3.3.2 QFontDialog类的静态API

```
/*
参数:
  - ok: 传出参数，用于判断是否获得了有效字体信息，指定一个布尔类型变量地址
  - initial: 字体对话框中默认选中并显示该字体信息，用于对话框的初始化
  - parent: 字体对话框窗口的父对象
  - title: 字体对话框的窗口标题
  - options: 字体对话框选项，使用默认属性即可，一般不设置
*/
[static] QFont QFontDialog::getFont(
            bool *ok, const QFont &initial,
            QWidget *parent = nullptr, const QString &title = QString(),
            QFontDialog::FontDialogOptions options = FontDialogOptions());

[static] QFont QFontDialog::getFont(bool *ok, QWidget *parent = nullptr);
```

### 3.3.3 测试代码

通过字体对话框选择字体, 并将选择的字体设置给当前窗口

```
void MainWindow::on_fontdlg_clicked() {
#if 1
    // 方式1
    bool ok;
    QFont ft = QFontDialog::getFont(&ok, QFont("微软雅黑", 12, QFont::Bold), this,
"选择字体");
    qDebug() << "ok value is: " << ok;
#else
    // 方式2
    QFont ft = QFontDialog::getFont(NULL);
#endif
    // 将选择的字体设置给当前窗口对象
    this->setFont(ft);
}
```

### 3.4 QColorDialog

QColorDialog类是QDialog的子类, 通过这个类我们可以得到一个选择颜色的对话框窗口, 和前边介绍的对话框类一样, 我们只需要调用这个类的静态成员函数就可以得到想要的窗口了。

#### 3.4.1 颜色类 QColor

关于颜色的属性信息, 在QT框架中被封装到了一个叫QColor的类中, 下边为大家介绍一下这个类的API, 了解一下关于这个类的使用。

各种颜色都是基于红, 绿, 蓝这三种颜色调配而成的, 并且颜色还可以进行透明度设置, 默认是不透明的。

```
// 构造函数
QColor::QColor(Qt::GlobalColor color);
QColor::QColor(int r, int g, int b, int a = ...);
QColor::QColor();

// 参数设置 red, green, blue, alpha, 取值范围都是 0-255
void QColor::setRed(int red);       // 红色
void QColor::setGreen(int green);   // 绿色
void QColor::setBlue(int blue); // 蓝色
void QColor::setAlpha(int alpha);   // 透明度, 默认不透明(255)
void QColor::setRgb(int r, int g, int b, int a = 255);

int QColor::red() const;
int QColor::green() const;
int QColor::blue() const;
int QColor::alpha() const;
void QColor::getRgb(int *r, int *g, int *b, int *a = nullptr) const;
```

#### 3.4.2 静态API函数

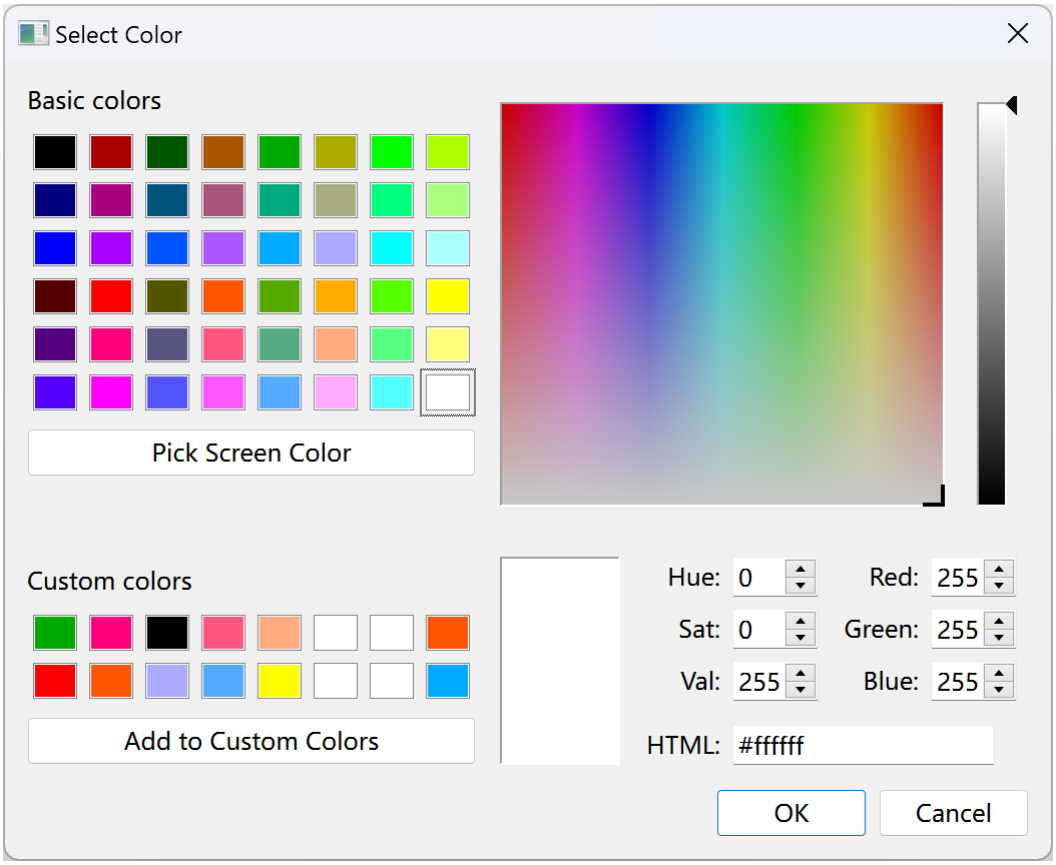
```
// 弹出颜色选择对话框, 并返回选中的颜色信息
/*
参数:
    - initial: 对话框中默认选中的颜色, 用于窗口初始化
    - parent: 给对话框窗口指定父对象
    - title: 对话框窗口的标题
    - options: 颜色对话框窗口选项, 使用默认属性即可, 一般不需要设置
*/
[static] QColor QColorDialog::getColor(
        const QColor &initial = Qt::white,
        QWidget *parent = nullptr, const QString &title = QString(),
        QColorDialog::ColorDialogOptions options = ColorDialogOptions());
```

#### 3.4.3 测试代码

> 场景描述:
>
> 1. 在窗口上放一个标签控件
> 2. 通过颜色对话框选择一个颜色,将选中的颜色显示到标签控件上
> 3. 将选中的颜色的 RGBA 值分别显示出来

```
void MainWindow::on_colordlg_clicked() {
    QColor color = QColorDialog::getColor();
    QBrush brush(color);
    QRect rect(0, 0, ui->color->width(), ui->color->height());
    QPixmap pix(rect.width(), rect.height());
    QPainter p(&pix);
    p.fillRect(rect, brush);
    ui->color->setPixmap(pix);
    QString text = QString("red: %1, green: %2, blue: %3, 透明度: %4")

.arg(color.red()).arg(color.green()).arg(color.blue()).arg(color.alpha());
    ui->colorlabel->setText(text);
}
```

颜色对话框窗口效果展示




### 3.5 QInputDialog

QInputDialog类是QDialog的子类, 通过这个类我们可以得到一个输入对话框窗口, 根据实际需求我们可以在这个输入窗口中输入整形, 浮点型, 字符串类型的数据, 并且还可以显示下拉菜单供使用者选择。和前边介绍的对话框类一样, 我们只需要调用这个类的静态成员函数就可以得到想要的窗口了。

### 3.5.1 API - 静态函数

```
// 得到一个可以输入浮点数的对话框窗口，返回对话框窗口中输入的浮点数
/*
参数:
  - parent: 对话框窗口的父窗口
  - title: 对话框窗口显示的标题信息
  - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
  - value: 对话框窗口中显示的浮点值，默认为 0
  - min: 对话框窗口支持显示的最小数值
  - max: 对话框窗口支持显示的最大数值
  - decimals: 浮点数的精度，默认保留小数点以后1位
  - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
  - flags: 对话框窗口的窗口属性，使用默认值即可
*/
[static] double QInputDialog::getDouble(
            QWidget *parent, const QString &title,
            const QString &label, double value = 0,
            double min = -2147483647, double max = 2147483647,
            int decimals = 1, bool *ok = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags());

// 得到一个可以输入整形数的对话框窗口，返回对话框窗口中输入的整形数
/*
参数:
  - parent: 对话框窗口的父窗口
  - title: 对话框窗口显示的标题信息
  - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
  - value: 对话框窗口中显示的整形值，默认为 0
  - min: 对话框窗口支持显示的最小数值
  - max: 对话框窗口支持显示的最大数值
  - step: 步长，通过对话框提供的按钮调节数值每次增长/递减的量
  - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
  - flags: 对话框窗口的窗口属性，使用默认值即可
*/
[static] int QInputDialog::getInt(
            QWidget *parent, const QString &title,
            const QString &label, int value = 0,
            int min = -2147483647, int max = 2147483647,
            int step = 1, bool *ok = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags());

// 得到一个带下来菜单的对话框窗口，返回选择的菜单项上边的文本信息
/*
参数:
  - parent: 对话框窗口的父窗口
  - title: 对话框窗口显示的标题信息
  - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
  - items: 字符串列表，用于初始化窗口中的下拉菜单，每个字符串对应一个菜单项
  - current: 通过菜单项的索引指定显示下拉菜单中的哪个菜单项，默认显示第一个(编号为0)
  - editable: 设置菜单项上的文本信息是否可以进行编辑，默认为true，即可以编辑
  - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
  - flags: 对话框窗口的窗口属性，使用默认值即可
  - inputMethodHints: 设置显示模式，默认没有指定任何特殊显示格式，显示普通文本字符串
    - 如果有特殊需求，可以参数帮助文档进行相关设置
```

```
*/
[static] QString QInputDialog::getItem(
            QWidget *parent, const QString &title,
            const QString &label, const QStringList &items,
            int current = 0, bool editable = true, bool *ok = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags(),
            Qt::InputMethodHints inputMethodHints = Qt::ImhNone);

// 得到一个可以输入多行数据的对话框窗口，返回用户在窗口中输入的文本信息
/*
参数:
  - parent: 对话框窗口的父窗口
  - title: 对话框窗口显示的标题信息
  - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
  - text: 指定显示到多行输入框中的文本信息，默认是空字符串
  - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
  - flags: 对话框窗口的窗口属性，使用默认值即可
  - inputMethodHints: 设置显示模式，默认没有指定任何特殊显示格式，显示普通文本字符串
    - 如果有特殊需求，可以参数帮助文档进行相关设置
*/
[static] QString QInputDialog::getMultiLineText(
            QWidget *parent, const QString &title, const QString &label,
            const QString &text = QString(), bool *ok = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags(),
            Qt::InputMethodHints inputMethodHints = Qt::ImhNone);

// 得到一个可以输入单行信息的对话框窗口，返回用户在窗口中输入的文本信息
/*
参数:
  - parent: 对话框窗口的父窗口
  - title: 对话框窗口显示的标题信息
  - label: 对话框窗口中显示的文本信息(用于描述对话框的功能)
  - mode: 指定单行编辑框中数据的反馈模式，是一个 QLineEdit::EchoMode 类型的枚举值
    - QLineEdit::Normal: 显示输入的字符。这是默认值
    - QLineEdit::NoEcho: 不要展示任何东西。这可能适用于连密码长度都应该保密的密码。
    - QLineEdit::Password: 显示与平台相关的密码掩码字符，而不是实际输入的字符。
    - QLineEdit::PasswordEchoOnEdit: 在编辑时按输入显示字符，否则按密码显示字符。
  - text: 指定显示到单行输入框中的文本信息，默认是空字符串
  - ok: 传出参数，用于判断是否得到了有效数据，一般不会使用该参数
  - flags: 对话框窗口的窗口属性，使用默认值即可
  - inputMethodHints: 设置显示模式，默认没有指定任何特殊显示格式，显示普通文本字符串
     - 如果有特殊需求，可以参数帮助文档进行相关设置
*/
[static] QString QInputDialog::getText(
            QWidget *parent, const QString &title, const QString &label,
            QLineEdit::EchoMode mode = QLineEdit::Normal,
            const QString &text = QString(), bool *ok = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags(),
            Qt::InputMethodHints inputMethodHints = Qt::ImhNone);
```

### 3.5.2 测试代码

整型输入框

```
void MainWindow::on_inputdlg_clicked() {
    int ret = QInputDialog::getInt(this, "年龄", "您的当前年龄: ", 10, 1, 100, 2);
    QMessageBox::information(this, "年龄", "您的当前年龄: " +
QString::number(ret));
}
```

窗口效果展示:

浮点型输入框

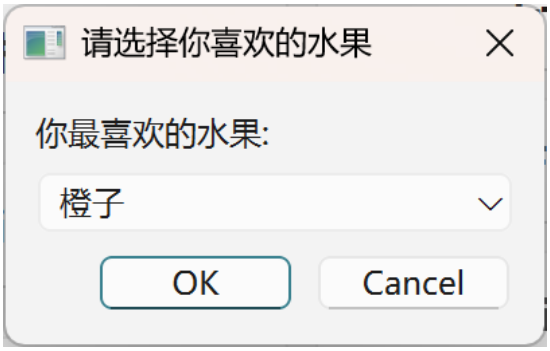
```
void MainWindow::on_inputdlg_clicked() {
    double ret = QInputDialog::getDouble(this, "工资", "您的工资: ", 2000, 1000,
6000, 2);
    QMessageBox::information(this, "工资", "您的当前工资: " +
QString::number(ret));
}
```

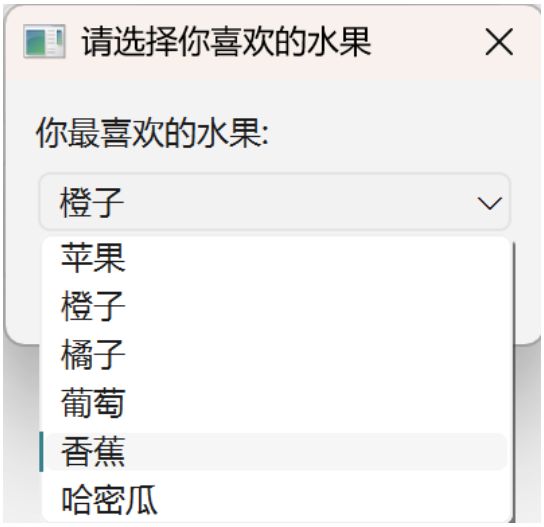
窗口效果展示:

带下拉菜单的输入框

```
void MainWindow::on_inputdlg_clicked() {
    QStringList items;
    items << "苹果" << "橙子" << "橘子" << "葡萄" << "香蕉" << "哈密瓜";
    QString item = QInputDialog::getItem(this, "请选择你喜欢的水果", "你最喜欢的水
果:", items, 1, false);
    QMessageBox::information(this, "水果", "您最喜欢的水果是: " + item);
}
```

窗口效果展示:






多行字符串输入框

```
void MainWindow::on_inputdlg_clicked() {
    QString info = QInputDialog::getMultiLineText(this, "表白",
                                                  "您最想对漂亮小姐姐说什么呢?", "呦吼吼...");
    QMessageBox::information(this, "知心姐姐", "您最想对小姐姐说: " + info);
}
```

窗口效果展示:

单行字符串输入框

```
void MainWindow::on_inputdlg_clicked() {
    QString text = QInputDialog::getText(this, "密码", "请输入新的密码",
                                         QLineEdit::Password, "helloworld");
    QMessageBox::information(this, "密码", "您设置的密码是: " + text);
}
```

### 3.6 QProgressDialog

QProgressDialog类是QDialog的子类, 通过这个类我们可以得到一个带进度条的对话框窗口, 这种类型的对话框窗口一般常用于文件拷贝、数据传输等实时交互的场景中。

#### 3.6.1 常用API

```
// 构造函数
/*
参数:
  - labelText: 对话框中显示的提示信息
  - cancelButtonText: 取消按钮上显示的文本信息
  - minimum: 进度条最小值
  - maximum: 进度条最大值
  - parent: 当前窗口的父对象
  - f: 当前进度窗口的flag属性, 使用默认属性即可, 无需设置
*/
QProgressDialog::QProgressDialog(
```

```
    QWidget *parent = nullptr,
    Qt::WindowFlags f = Qt::WindowFlags());

QProgressDialog::QProgressDialog(
    const QString &labelText, const QString &cancelButtonText,
    int minimum, int maximum, QWidget *parent = nullptr,
    Qt::WindowFlags f = Qt::WindowFlags());


// 设置取消按钮显示的文本信息
[slot] void QProgressDialog::setCancelButtonText(const QString
&cancelButtonText);

// 公共成员函数和槽函数
QString QProgressDialog::labelText() const;
void QProgressDialog::setLabelText(const QString &text);

// 得到进度条最小值
int QProgressDialog::minimum() const;
// 设置进度条最小值
void QProgressDialog::setMinimum(int minimum);

// 得到进度条最大值
int QProgressDialog::maximum() const;
// 设置进度条最大值
void QProgressDialog::setMaximum(int maximum);

// 设置进度条范围(最大和最小值)
[slot] void QProgressDialog::setRange(int minimum, int maximum);

// 得到进度条当前的值
int QProgressDialog::value() const;
// 设置进度条当前的值
void QProgressDialog::setValue(int progress);


bool QProgressDialog::autoReset() const;
// 当value() = maximum()时，进程对话框是否调用reset()，此属性默认为true。
void QProgressDialog::setAutoReset(bool reset);


bool QProgressDialog::autoClose() const;
// 当value() = maximum()时，进程对话框是否调用reset()并且隐藏，此属性默认为true。
void QProgressDialog::setAutoClose(bool close);

// 判断用户是否按下了取消键，按下了返回true，否则返回false
bool wasCanceled() const;


// 重置进度条
// 重置进度对话框。wascancelled()变为true，直到进程对话框被重置。进度对话框被隐藏。
[slot] void QProgressDialog::cancel();
// 重置进度对话框。如果autoClose()为真，进程对话框将隐藏。
[slot] void QProgressDialog::reset();
```

```
// 信号
// 当单击cancel按钮时，将发出此信号。默认情况下，它连接到cancel()槽。
[signal] void QProgressDialog::canceled();

// 设置窗口的显示状态(模态, 非模态)
/*
参数:
    Qt::NonModal  -> 非模态
    Qt::WindowModal -> 模态, 阻塞父窗口
    Qt::ApplicationModal -> 模态, 阻塞应用程序中的所有窗口
*/
void QWidget::setWindowModality(Qt::WindowModality windowModality);
```

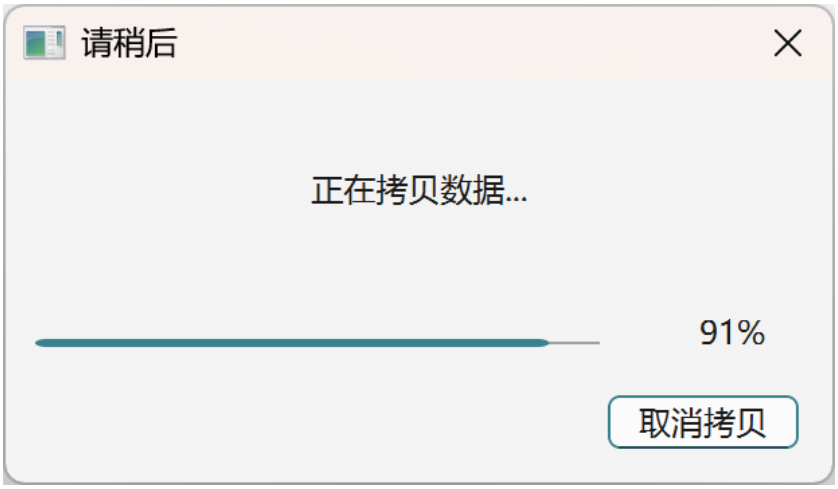
### 2.6.2 测试代码

> 场景描述:
>
> 1. 基于定时器模拟文件拷贝的场景
> 2. 点击窗口按钮, 进度条窗口显示, 同时启动定时器
> 3. 通过定时器信号, 按照固定频率更新对话框窗口进度条
> 4. 当进度条当前值 == 最大值, 关闭定时器, 关闭并析构进度对话框

```
void MainWindow::on_progressdlg_clicked() {
    // 1. 创建进度条对话框窗口对象
    QProgressDialog *progress = new QProgressDialog(
                "正在拷贝数据...", "取消拷贝", 0, 100, this);
    // 2. 初始化并显示进度条窗口
    progress->setWindowTitle("请稍后");
    progress->setWindowModality(Qt::WindowModal);
    progress->show();

    // 3. 更新进度条
    static int value = 0;
    QTimer *timer = new QTimer;
    connect(timer, &QTimer::timeout, this, [=]() {
         progress->setValue(value);
         value++;
         // 当value > 最大值的时候
         if(value > progress->maximum()) {
             timer->stop();
             value = 0;
             delete progress;
             delete timer;
         }
    });

    connect(progress, &QProgressDialog::canceled, this, [=]() {
        timer->stop();
        value = 0;
        delete progress;
        delete timer;
    });
```

```
    timer->start(50);
}
```

进度窗口效果展示:




## 四 Qt 之模式、非模式和半模式对话框

关于“模式”和“非模式”对话框，相信大家都比较熟悉，但其中有一个可能很多人都比较陌生，介于两者之间的状态，我们称之为“半模式“。

### 4.1 模式对话框

阻塞同一应用程序中其它可视窗口输入的对话框。模式对话框有自己的事件循环，用户必须完成这个对话框中的交互操作，并且关闭了它之后才能访问应用程序中的其它任何窗口。模式对话框仅阻止访问与对话相关联的窗口，允许用户继续使用其它窗口中的应用程序。

显示模态对话框最常见的方法是调用其exec()函数，当用户关闭对话框，exec()将提供一个有用的返回值，并且这时流程控制继续从调用exec()的地方进行。通常情况下，要获得对话框关闭并返回相应的值，我们连接默认按钮，例如：”确定”按钮连接到accept()槽，”取消”按钮连接到reject()槽。另外我们也可以连接done()槽，传递给它Accepted或Rejected。

#### 源码

```
MainWindow *pMainWindow = new MainWindow();
pMainWindow->setWindowTitle(QStringLiteral("主界面"));
pMainWindow->show();
CustomWindow *pDialog = new CustomWindow(pMainWindow);
pDialog->setWindowTitle(QStringLiteral("模式对话框"));
// 关键代码
pDialog->exec();
// 关闭模态对话框以后才会执行下面的代码
pMainWindow->setWindowTitle(QStringLiteral("主界面-模式对话框"));
qDebug() << QStringLiteral("关闭模态对话框以后，可以继续向下执行");
```

> 1. 主界面被阻塞，不能进行点击、拖动等任何操作。
> 2. exec()之后的代码不会执行，直到关闭模态对话框。

### 4.2 非模式对话框

和同一个程序中其它窗口操作无关的对话框。在文字处理中的查找和替换对话框通常是非模式的，允许用户同时与应用程序的主窗口和对话框进行交互。调用show()来显示非模式对话框，并立即将控制返回给调用者。

如果隐藏对话框后调用show()函数，对话框将显示在其原始位置，这是因为窗口管理器决定的窗户位置没有明确由程序员指定，为了保持被用户移动的对话框位置，在closeEvent()中进行处理，然后在显示之前，将对话框移动到该位置。

**源码**

```
MainWindow *pMainWindow = new MainWindow();
pMainWindow->setWindowTitle(QStringLiteral("主界面"));
pMainWindow->show();
CustomWindow *pDialog = new CustomWindow(pMainWindow);
pDialog->setWindowTitle(QStringLiteral("非模式对话框"));

// 关键代码
pDialog->show();

// 下面的代码会立即运行
pMainWindow->setWindowTitle(QStringLiteral("主界面-非模式对话框"));
qDebug() << QStringLiteral("立即运行");
```

> 1. 主界面不会被阻塞，可以进行点击、拖动等任何操作。
> 2. show()之后的代码会立即执行。

### 4.3 半模式对话框

调用setModal(true)或者setWindowModality()，然后show()。有别于exec()，show() 立即返回给控制调用者。

对于进度对话框来说，调用setModal(true)是非常有用的，用户必须拥有与其交互的能力，例如：取消长时间运行的操作。如果使用show()和setModal(true)共同执行一个长时间操作，则必须定期在执行过程中调用QApplication ::processEvents()，以使用户能够与对话框交互（可以参考QProgressDialog）。

**源码**

```
MainWindow *pMainWindow = new MainWindow();
pMainWindow->setWindowTitle(QStringLiteral("主界面"));
pMainWindow->show();
CustomWindow *pDialog = new CustomWindow(pMainWindow);
pDialog->setWindowTitle(QStringLiteral("半模式对话框"));

// 关键代码
pDialog->setModal(true);
pDialog->show();

// 下面的代码会立即运行
pMainWindow->setWindowTitle(QStringLiteral("主界面-半模式对话框"));
qDebug() << QStringLiteral("立即运行");
```

> 1. 主界面被阻塞，不能进行点击、拖动等任何操作。
> 2. show()之后的代码会立即执行。

#### 4.5 QDialog,QWidget实现模态和非模态

对于 QDialog 的模态及非模态是直接可以实现的此处总结下。

##### 4.5.1 模态QDialog

方式一：

```
QDialog dlg(this);
dlg.exec();
```

方式二：

```
QDialog *pDlg = new QDialog(this);
pDlg->setModal(true);
pDlg->show();
```

##### 4.5.2 非模态QDialog

```
QDialog *pDlg = new QDialog(this);
pDlg->show();
```

QDialog实现模态非模态很简单，但是对于QWidget有点迷茫，QWidget中没有exec()，也没有setModal()方式，但是想想看，QWidget作为QDialog的基类，而且QWidget作为“窗口”使用也是在平常不过了，所以会意识到QWidget中是否存在一个相对exec()或setModal()更基本的操作来实现模态和非模态呢？就这样，我找到了setWindowModality()，此函数就是用来设置QWidget运行时的程序阻塞方式的，参数解

释如下：

> Qt::NonModal　不阻塞
>
> Qt::WindowModal　阻塞父窗口，所有祖先窗口及其子窗口

> Qt::ApplicationModal 阻塞整个应用程序

看来，`setModal()`也就是使用`setWindowModality()`设置`Qt::ApplicationModal`参数来实现的模态。

如此，要实现QWidget的模态和非模态，只要调用`setWindowModality()`设置阻塞类型就好了：

```
// QWidget *pWid = new QWidget(this);  ----- 注意这样设置不能实现非模态，改成如下
QWidget *pWid = new QWidget (NULL);
pWid->setWindowModality(Qt::ApplicationModal);
// pWid->setAttribute(Qt::WA_ShowModal, true);
pWid->show();
```

但是运行发现并未实现模态效果。这里需要注意，当希望使用`setWindowModality()`将QWidget设置为模态时应该保证QWidget父部件为0，这里修改即可。

`QWidget *pWid = new QWidget(this);`修改为 `QWidget *pWid = new QWidget(nullptr);`

此外，通过`setWindowModality()`设置模态窗口并不是唯一方式，直接设置部件（或窗口）属性也可以：

```
pWid->setAttribute(Qt::WA_ShowModal, true)
```

**总而言之**

是否是模态和QDialog 和QWidget都可以模态和非模态.exec(), show() 等函数无直接关系，只和窗口属性有关,使用以下两种方式都行：

```
setAttribute(Qt::WA_ShowModal, true);        // 属性设置
setWindowModality(Qt::ApplicationModal);     // 设置阻塞类型
```

# 课堂小结

- Qt实现 QWidget 和 QMainWindow 窗口
- 实现 QDialog 窗口
- QDialog 对话框子类功能实现
- 模态和非模态对话框实现

# 随堂作业

# 练习题：文本文件查看器实现（基于 Qt）

**要求说明**：使用 Qt 框架实现一个图形界面程序：

1. **界面布局**
    - 左侧：`QPushButton`按钮，显示文本“打开文件”
    - 右侧：`QLabel`标签，用于显示文件路径（初始为空）
    - `QLabel`标签，显示标题文本“显示文件内容”

- `QTextBrowser` 控件，用于显示文本内容（带滚动条）

2. **核心功能**
   - 点击 "打开文件"按钮时：
     - 弹出标准文件选择对话框（`QFileDialog::getOpenFileName`）
     - 限制文件类型为文本文件（扩展名 `.txt`）
   - 文件选择后：
     - **路径显示**：在 `QLabel` 中显示文件的完整绝对路径
     - **内容加载**：将文件内容完整显示在 `QTextBrowser` 中（保留原格式）
   - 错误处理：
     - 文件读取失败时，弹出错误提示框（`QMessageBox::critical`）