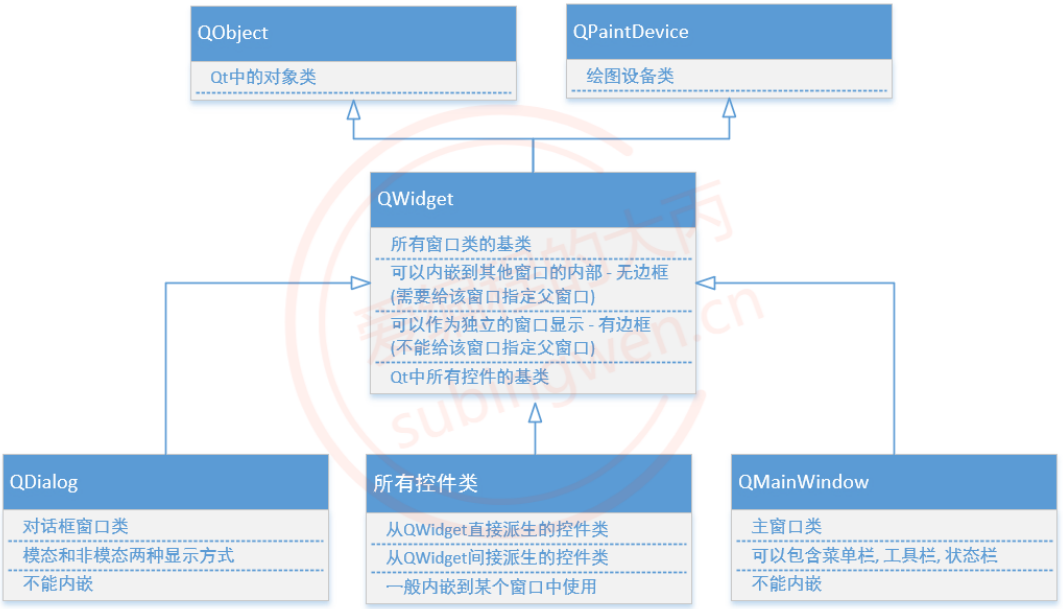
内容摘要：文章中主要介绍了Qt中常用的窗口类, 主要内容包括: `窗口类的基类QWidget` ， `对话框基类QDialog` ， `带菜单栏工具栏状态栏的QMainWindow` ， `消息对话框QMessageBox` ， `文件对话框QFileDialog` ， `字体对话框QFontDialog` ， `颜色对话框QColorDialog` ， `输入型对话框QInputDialog` ， `进度条对话框QProgressDialog` ， `资源文件` 。文章中除了关于知识点的文字描述、代码演示, 还有相关的视频讲解，开始学习。。。

1. QWidget

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。**

在1.1 Qt入门章节中已经为大家介绍了 `QWidget` 的一些特点, 为了让大家能够对这个类有更深入的了解, 下面来说一说这个类常用的一些API函数。

关于这个窗口类的属性介绍, 请参考容器控件之QWidget。



1.1 设置父对象

C++

```cpp
// 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

C++

```cpp
//------------- 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:

C++

```cpp
// 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

C++

```cpp
//------------- 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);

```

1.4 窗口标题和图标

```C++
//------------- 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

```C++
// 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);
```

基于窗口策略实现右键菜单具体操作请参考 Qt右键菜单的添加和使用

1.6 槽函数

```C++
//------------- 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);
```

2. QDialog

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:

C++

```cpp
// 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

TEXT

```
场景介绍:
    1. 有两个窗口，主窗口和一个对话框子窗口
    2. 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

C++

```cpp
// 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。

C++

```cpp
// 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
```

```
    ......
    ......
}
```

3. 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

C++

```cpp
// 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);
```

3.1.2 测试代码

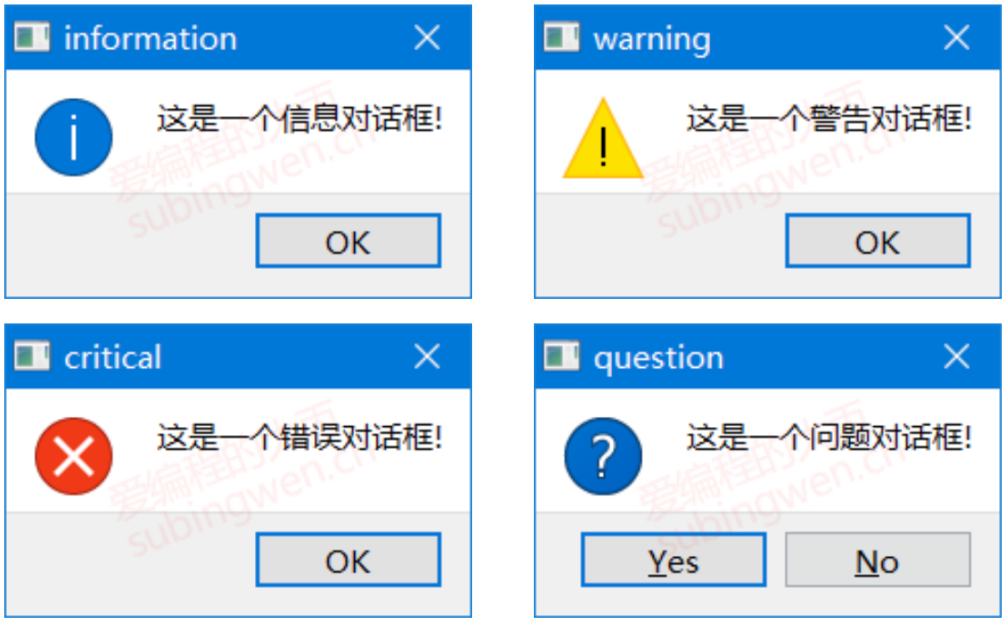
- 测试代码片段

C++

```cpp
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

C++

```
/*
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(),
```

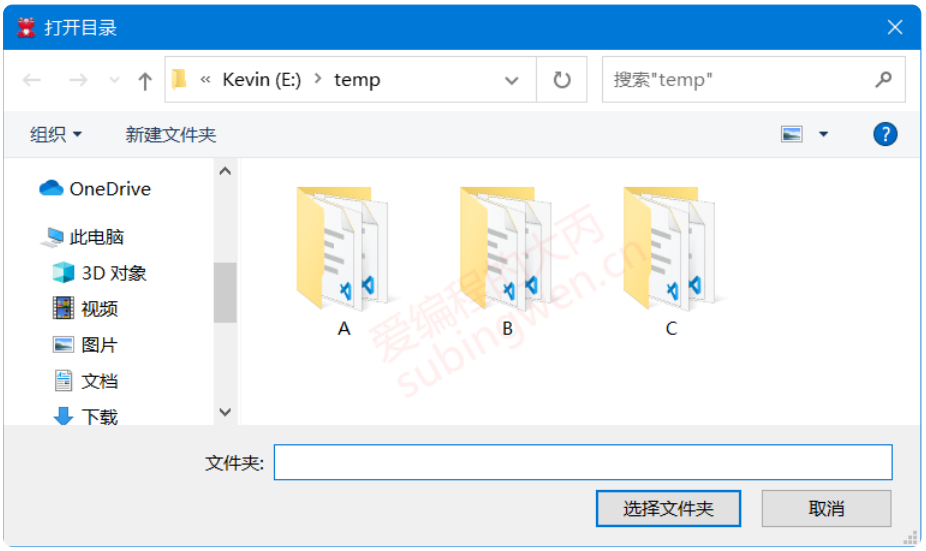
```
34                  const QString &filter = QString(),
35                  QString *selectedFilter = nullptr,
```

3.2.2 测试代码

- 打开 一个已存在的本地目录

```C++
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);
}
```

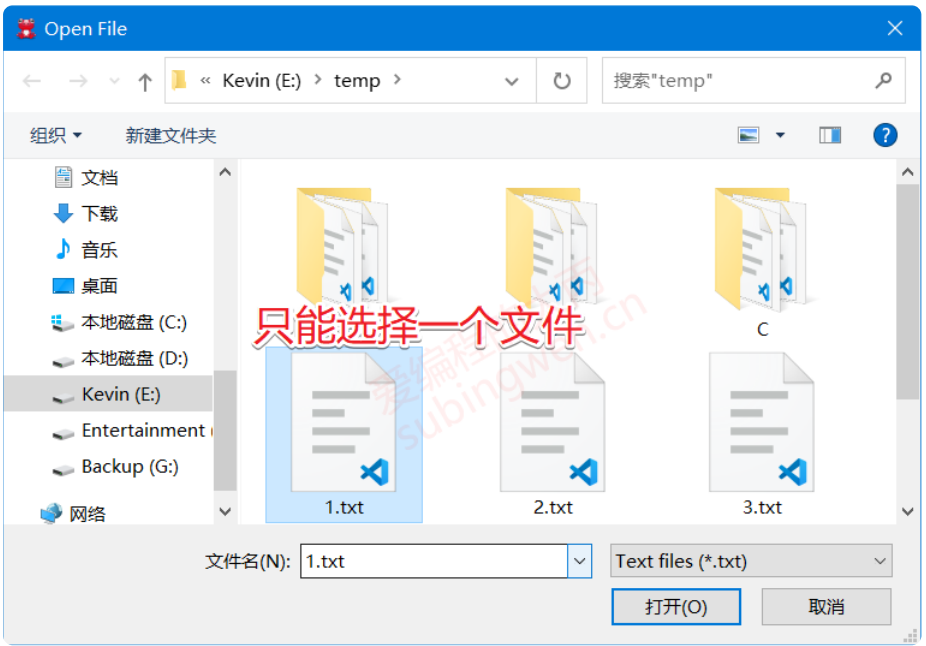
对话框效果如下:



- 打开 一个本地文件

```C++
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);
}
```

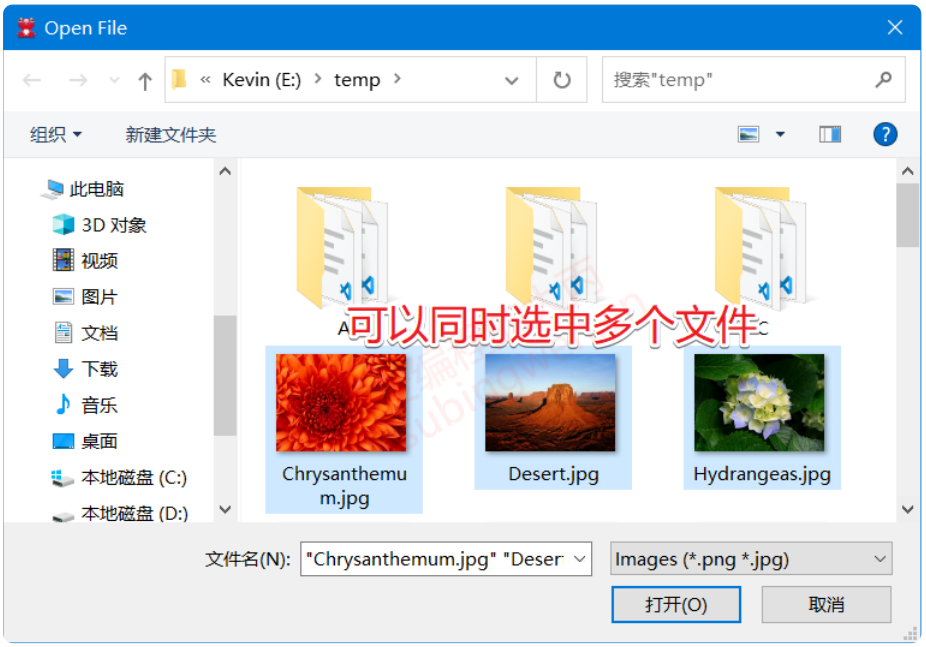
对话框效果如下:



打开 多个本地文件

```C++
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);
}
```

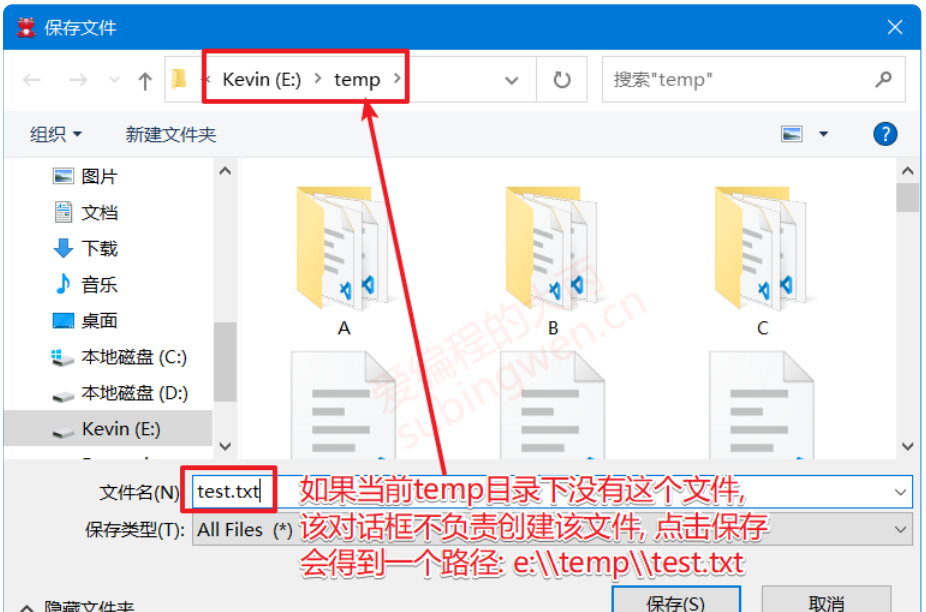
对话框效果如下:



打开 保存文件对话框

```C++
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);
}
```

对话框效果如下:



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。

C++

```
// 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。

C++

```
// 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

C++

```
/*
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);
```

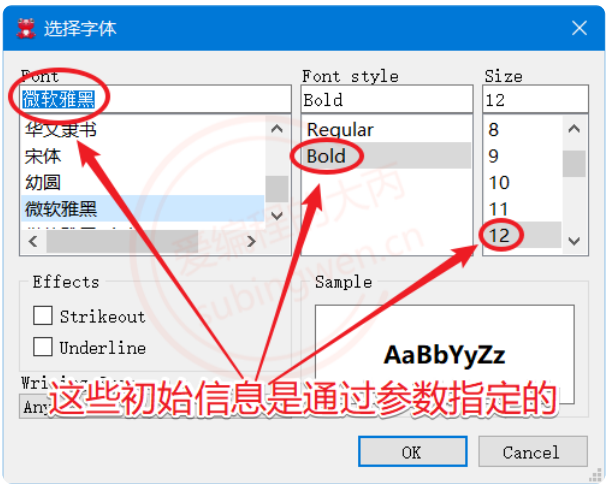
3.3.3 测试代码

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

C++

```cpp
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);
}
```

字体对话框效果展示:



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。

C++

```cpp
// 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

C++

```
  // 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

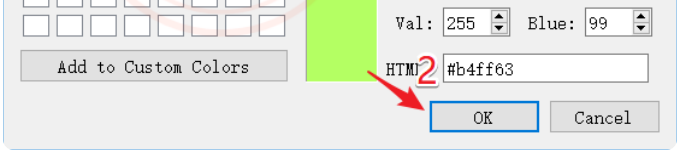
TEXT

```
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
```

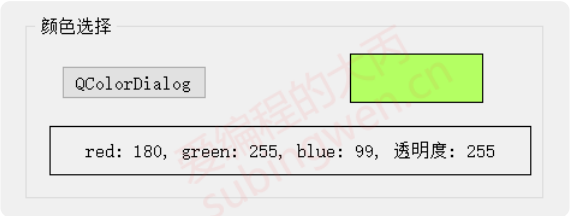
C++

```
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



测试代码效果展示



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

C++

```cpp
// 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(
```

3.5.2 测试代码

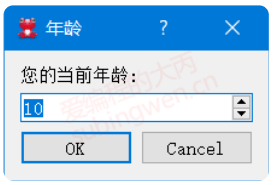
- 整形输入框

C++

```cpp
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:

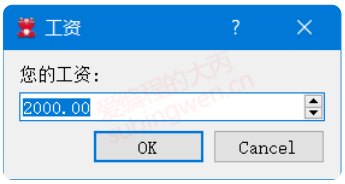


- 浮点型输入框

C++

```cpp
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:

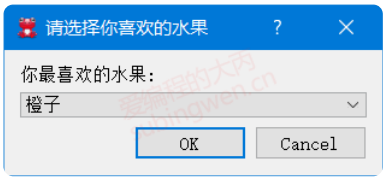


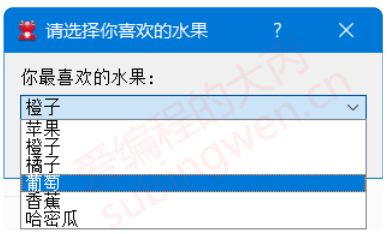
- 带下拉菜单的输入框

C++

```cpp
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
    QMessageBox::information(this, "水果", "您最喜欢的水果是: " + item);
}
```

窗口效果展示:



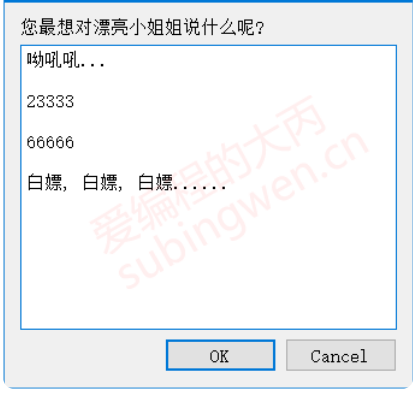


- 多行字符串输入框

C++

```cpp
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?", "
    QMessageBox::information(this, "知心姐姐", "您最想对小姐姐说: " + info);
}
```

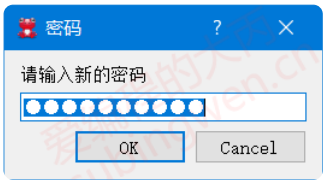
窗口效果展示:



- 单行字符串输入框

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

窗口效果展示:



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

```C++
// 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
```

```
// 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;
```

2.6.2 测试代码

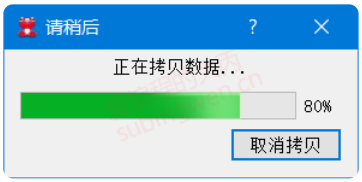
TEXT

```
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
```

C++

```
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;
    });
```

进度窗口效果展示:



4. QMainWindow

`QMainWindow` 是标准基础窗口中结构最复杂的窗口, 其组成如下:

- 提供了 `菜单栏` , `工具栏` , `状态栏` , `停靠窗口`
- 菜单栏: 只能有一个, 位于窗口的最上方
- 工具栏: 可以有多个, 默认提供了一个, 窗口的上下左右都可以停靠

- 状态栏: 只能有一个, 位于窗口最下方
- 停靠窗口: 可以有多个, 默认没有提供, 窗口的上下左右都可以停靠

4.1 菜单栏

- 添加菜单项

关于顶级菜单可以直接在UI窗口中双击, 直接输入文本信息即可, 对应子菜单项也可以通过先双击在输入的方式完成添加, 但是 `这种方式不支持中文的输入` 。

- 常用的添加方式

一般情况下, 我们都是先在外面创建出 `QAction` 对象, 然后再将其拖拽到某个菜单下边, 这样子菜单项的添加就完成了。

- 通过代码的方式添加菜单或者菜单项

```C++
// 给菜单栏添加菜单
QAction *QMenuBar::addMenu(QMenu *menu);
QMenu *QMenuBar::addMenu(const QString &title);
```

```
QMenu *QMenuBar::addMenu(const QIcon &icon, const QString &title);

// 给菜单对象添加菜单项(QAction)
QAction *QMenu::addAction(const QString &text);
QAction *QMenu::addAction(const QIcon &icon, const QString &text);

// 添加分割线
QAction *QMenu::addSeparator();
```

- 菜单项 QAction 事件的处理

单击菜单项, 该对象会发出一个信号

C++

```
// 点击QAction对象发出该信号
[signal] void QAction::triggered(bool checked = false);
```

示例代码

C++

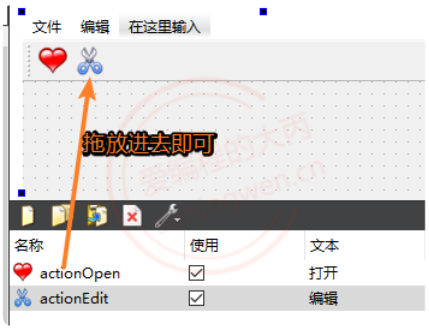
```
// save_action 是某个菜单项对象名, 点击这个菜单项会弹出一个对话框
connect(ui->save_action, &QAction::triggered, this, [=]()
{
      QMessageBox::information(this, "Triggered", "我是菜单项, 你不要调戏我...");
});
```

4.2 工具栏

4.2.1 添加工具按钮

窗口中的工具栏我们经常见到, 并不会为此感到陌生, 那么如何往工具栏中添加工具按钮呢? 一共有两种方式, 这里依次为大家进行介绍。

方式1：先创建QAction对象, 然后拖拽到工具栏中, 和添加菜单项的方式相同



方式2：如果不通过UI界面直接操作，那么就需要调用相关的API函数了

C++

```
// 在QMainWindow窗口中添加工具栏
void QMainWindow::addToolBar(Qt::ToolBarArea area, QToolBar *toolbar);
void QMainWindow::addToolBar(QToolBar *toolbar);
QToolBar *QMainWindow::addToolBar(const QString &title);

// 将Qt控件放到工具栏中
// 工具栏类: QToolBar
// 添加的对象只要是QWidget或者启子类都可以被添加
QAction *QToolBar::addWidget(QWidget *widget);

// 添加QAction对象
QAction *QToolBar::addAction(const QString &text);
QAction *QToolBar::addAction(const QIcon &icon, const QString &text);

// 添加分隔线
QAction *QToolBar::addSeparator()
```

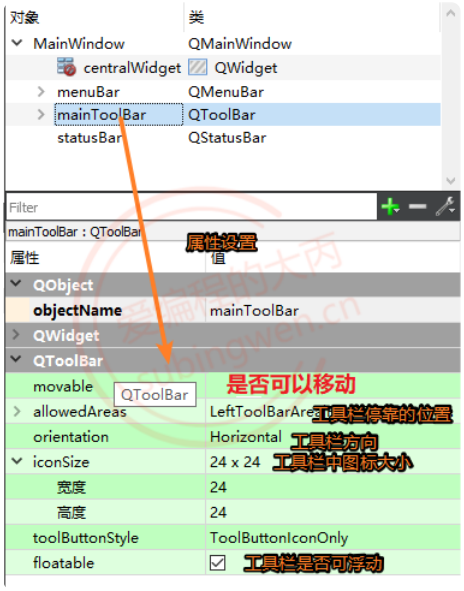
通过代码的方式对工具栏进行操作

C++

```cpp
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // 添加第二个工具栏
    QToolBar* toolbar = new QToolBar("toolbar");
    this->addToolBar(Qt::LeftToolBarArea, toolbar);

    // 给工具栏添加按钮和单行输入框
    ui->toolBar->addWidget(new QPushButton("搜索"));
    QLineEdit* edit = new QLineEdit;
    edit->setMaximumWidth(200);
    edit->setFixedWidth(100);
    ui->toolBar->addWidget(edit);
    // 添加QAction类型的菜单项
    ui->toolBar->addAction(QIcon(":/er-dog"), "二狗子");
}
```

4.2.2 工具栏的属性设置

在UI窗口的树状列表中, 找到工具栏节点, 就可以到的工具栏的属性设置面板了, 这样就可以根据个人需求对工具栏的属性进行设置和修改了。



在Qt控件的属性窗口中对应了一些属性, 这些属性大部分都应了一个设置函数

- 在对应的类中函数名叫什么?
  - 规律: `set+属性名 == 函数名`
- 某些属性没有对应的函数, 只能在属性窗口中设置

4.3 状态栏

一般情况下, 需要在状态栏中添加某些控件, 显示某些属性, 使用最多的就是添加标签 QLabel

C++

```cpp
// 类型: QStatusBar
void QStatusBar::addWidget(QWidget *widget, int stretch = 0);

[slot] void QStatusBar::clearMessage();
[slot] void QStatusBar::showMessage(const QString &message, int timeout = 0);
```

相关的操作代码

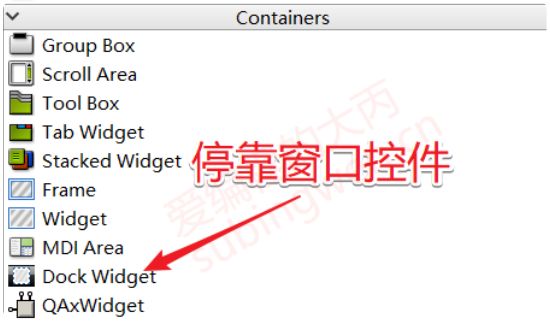
```C++
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // 状态栏添加子控件
    // 按钮
    QPushButton* button = new QPushButton("按钮");
    ui->statusBar->addWidget(button);
    // 标签
    QLabel* label = new QLabel("hello,world");
    ui->statusBar->addWidget(label);
}
```

4.4 停靠窗口

停靠窗口可以通过鼠标拖动停靠到窗口的上、下、左、右，或者浮动在窗口上方。如果需要这种类型的窗口必须手动添加，

如果在非QMainWindow类型的窗口中添加了停靠窗口，那么这个窗口是不能移动和浮动的。

浮动窗口在工具栏中， 直接将其拖拽到UI界面上即可。



停靠窗口也有一个属性面板, 我们可以在其对应属性面板中直接进行设置和修改相关属性。

5. 资源文件 .qrc

资源文件顾名思义就是一个存储资源的文件，在Qt中引入资源文件好处在于他能提高应用程序的部署效率并且减少一些错误的发生。

在程序编译过程中， 添加到资源文件中的文件也会以二进制的形式被打包到可执行程序中，这样这些资源就永远和可执行程序捆绑到一起了，不会出现加载资源却找不到的问题。

虽然资源文件优势很明显，但是它也不是万能的，资源文件中一般添加的都是比较小的资源，比如： 图片，配置文件，MP3 等，如果是类似视频这类比较大的文件就不适合放到资源文件中了。

比如我们需要给某个窗口设置图标, 代码如下:

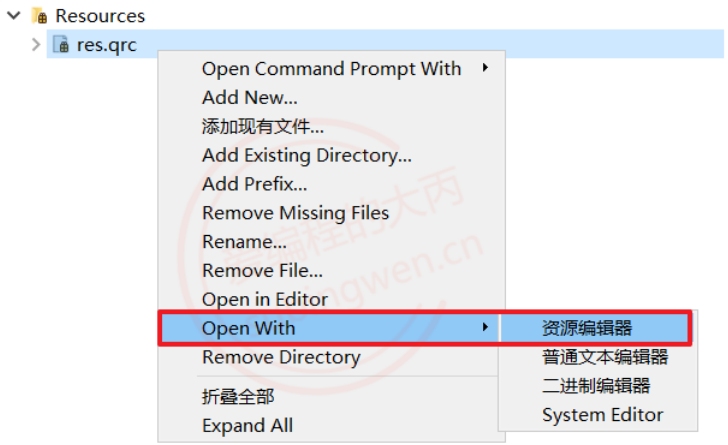
```C++
// 创建图标对象
QIcon::QIcon(const QString &fileName);
// QWidget类的 公共成员函数
void setWindowIcon(const QIcon &icon);

// 给窗口设置图标
// 弊端: 发布的 exe 必须要加载 d:\\pic\\1.ico 如果当前主机对应的目录中没有图片, 图标就无法被加载
// 发布 exe 需要额外发布图片, 将其部署到某个目录中
setWindowIcon(QIcon("d:\\pic\\1.ico"));
```

我们可以使用资源文件解决上述的弊端, 这样发布应用程序的时候直接发布exe就可以, 不需要再额外提供图片了。

下面介绍一下关于资源文件的创建步骤:

资源文件添加完毕之后, 继续给大家介绍资源文件的使用

1 使用资源编辑器 打开资源文件



2 给资源添加前缀

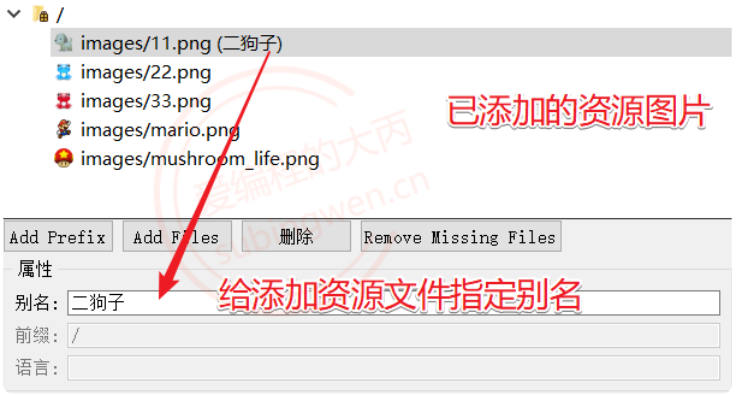
一个资源文件中可以添加多个前缀, 前缀就是添加的资源在资源文件中的路径, 前缀根据实际需求制定即可, 路径以 / 开头

3 添加文件

前缀添加完毕, 就可以在某个前缀下边添加相关的资源了。

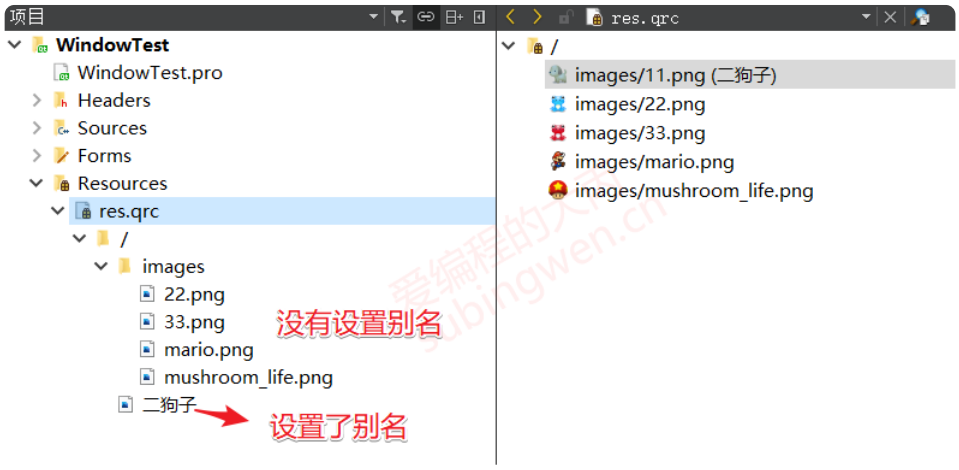
- 弹出以文件选择对话框, 选择资源文件
  - 资源文件放到什么地方?
    - 放到和 项目文件 .pro 同一级目录或者更深的目录中
    - 错误的做法: 将资源文件放到 .pro文件的上级目录, 这样资源文件无法被加载到
- 可以给添加的资源文件设置别名, 设置别名之后原来的名字就不能使用了

```PLAINTEXT
温馨提示:
    1. 在高版本的QtCreator中, 资源文件名字或者别名不支持中文
    2. 如果设置了中文会出现编译会报错
    3. 在此只是演示, 使用过程中需要额外注意该问题
```

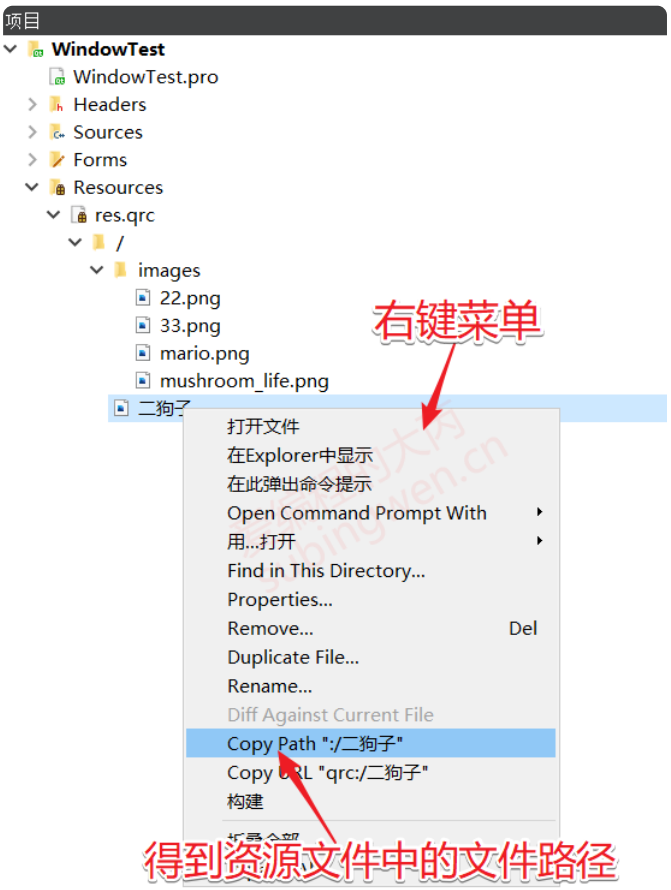


4 如何在程序中使用资源文件中的图片

将项目树中的资源文件节点展开



找到需要使用的资源图片节点, 鼠标右键, 弹出的菜单中选择 `Copy Path ...`



6. 视频讲解

以上知识点对应的视频讲解可以关注 B站-爱编程的大丙