高等程序设计 - Qt/C++

第 3 章：Qt 框架基础知识

王培杰

长江大学地球物理与石油资源学院

2025 年 9 月 29 日

目录

目录

Qt 框架简介

Qt 是什么?

- 跨平台 C++ 应用程序开发框架
- 由 Trolltech 公司开发（现为 Qt Company）
- 提供 GUI、网络、数据库、多媒体等功能
- 支持桌面、移动、嵌入式平台

Qt 开发的优秀软件

- Google Earth
- Autodesk Maya
- Adobe Photoshop
- WPS Office
- Virtual Box
- COMSOL Multiphysics

技术优势

- 跨平台支持
- 丰富的 API
- 优秀的性能
- 活跃的社区
- 完善的文档

开发优势

- 快速开发
- 代码复用
- 维护简单
- 部署方便
- 开源免费

目录

Qt 模块架构

核心模块

- QtCore - 基础核心类（对象模型、事件系统、线程、文件、容器等）
- QtGui - 图形用户界面类（窗口、绘图、事件处理等）
- QtWidgets - 桌面 GUI 组件类（按钮、对话框、布局等）
- QtNetwork - 网络编程类（TCP/IP、UDP、HTTP 等）
- QtSql - 数据库支持类（数据库连接、查询、事务等）
- QtCharts - 图表组件类（图表绘制）
- Qt3D - 3D 图形类（3D 图形渲染）
- QtMultimedia - 多媒体类（音频、视频、图像处理等）
- QtXml - XML 支持类（XML 解析、生成等）
- QtSvg - SVG 支持类（SVG 解析、生成等）
- QtQml - QML 支持类（QML 支持）
- QtQuick - 声明式 UI 类（QML 支持）

Qt 应用程序结构

```cpp
#include <QApplication> // Qt 应用程序类
#include <QMainWindow> // Qt 主窗口类
#include <QLabel> // Qt 标签类
#include <QVBoxLayout> // Qt 垂直布局类
#include <QWidget> // Qt 窗口部件类

int main(int argc, char *argv[]) // 主函数
{
    // 创建应用程序对象
    QApplication app(argc, argv);

    // 创建主窗口
    QMainWindow mainWindow;
    mainWindow.setWindowTitle("Qt Application Example");
    mainWindow.resize(400, 300);

    // 创建中心部件
    QWidget *centralWidget = new QWidget(&mainWindow);
    // 设置主窗口的中心部件
    mainWindow.setCentralWidget(centralWidget);
    mainWindow.setCentralWidget(centralWidget);

    // 创建布局
    QVBoxLayout *layout = new QVBoxLayout(centralWidget);

    // 创建标签
    QLabel *label = new QLabel("Hello, Qt!", centralWidget);
    // 设置标签的文本对齐方式
    label->setAlignment(Qt::AlignCenter);
    // 将标签添加到布局中
    layout->addWidget(label);

    // 显示窗口
    mainWindow.show();

    // 进入事件循环
    return app.exec();
}
```

目录

Qt 桌面 GUI 组件类

常用 Qt 桌面 GUI 组件

- **QWidget**：所有窗口组件的基类，提供窗口管理功能。
- **QDialog**：对话框组件，支持模态和非模态对话框。
- **QMainWindow**：主窗口组件，集成菜单栏、工具栏、状态栏等。
- **QFrame**：框架组件，提供边框和背景。
- **QScrollArea**：滚动区域组件，支持内容滚动。
- **QSplitter**：分割器组件，可拖动分割线调整子窗口大小。
- **QTabWidget**：标签页组件，支持多标签切换。
- **QStackedWidget**：堆叠组件，多个子窗口堆叠显示。
- **QToolBar**：工具栏组件，放置常用操作按钮。
- **QStatusBar**：状态栏组件，显示状态信息。
- **QMenu**、**QMenuBar**：菜单组件，提供菜单栏和下拉菜单。
- **QAction**：动作组件，封装可复用的操作。
- **QButtonGroup**：按钮组组件，管理一组按钮。
- **QGroupBox**：分组框组件，分组相关控件。
- **QLineEdit**：单行文本输入框。
- **QTextEdit**：多行文本输入框。
- **QComboBox**：下拉列表框。
- **QListWidget**：列表控件。
- **QTreeWidget**：树形控件。
- **QTableWidget**：表格控件。
- **QCalendarWidget**：日历控件。
- **QDateEdit**、**QTimeEdit**、**QDateTimeEdit**：日期/时间输入控件。
- **QSpinBox**、**QDoubleSpinBox**：整数/浮点数输入框。
- **QSlider**：滑块控件。
- **QScrollBar**：滚动条控件。
- **QProgressBar**：进度条控件。
- **QRubberBand**：橡皮筋选择框。

QWidget 详解

- **QWidget** 是 Qt 所有用户界面对象的基类，几乎所有可视化控件（如按钮、文本框、窗口等）都直接或间接继承自 QWidget。
- 它提供了窗口的基本管理功能，包括显示、隐藏、移动、调整大小、设置父子关系、事件处理等。
- QWidget 支持层次化的父子结构，子控件会自动跟随父控件移动和显示，便于界面组织和管理。
- 通过重写 QWidget 的 `paintEvent()`、`mousePressEvent()`、`keyPressEvent()` 等虚函数，可以实现自定义绘制和事件响应。
- QWidget 支持多种属性设置，如背景色、字体、光标、焦点策略、透明度等，灵活性极高。
- QWidget 既可以作为顶层窗口（无父对象），也可以作为其他控件的子部件嵌入到界面中。
- 典型用法：自定义控件时，继承 QWidget 并实现相关事件处理和绘制逻辑。

QWidget 使用示例（一）

```cpp
#include <QApplication>
#include <QWidget>
#include <QTimer>
#include <QDebug>
#include <QKeyEvent>
#include <QMouseEvent>
#include <QMimeData>
#include <QDragEnterEvent>
#include <QDropEvent>

class WidgetDemo : public QWidget
{
public:
    WidgetDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        // 初始化设置
        setWindowTitle("QWidget 示例演示");  // 设置窗口
        ↪ 标题
        resize(400, 300);  // 设置窗口大小
        setMinimumSize(200, 150);  // 设置窗口最小大小
        setMaximumSize(800, 600);  // 设置窗口最大大小
        // 窗口属性设置
        setWindowFlags(Qt::Window
        ↪ Qt::WindowCloseButtonHint
        ↪ Qt::WindowMinimizeButtonHint);  // 设置窗口
        ↪ 标志
        setStyleSheet("background-color: lightgray;");
        ↪ // 设置窗口样式
        setMouseTracking(true);  // 设置鼠标跟踪
        setCursor(Qt::ArrowCursor);  // 设置鼠标形状
        setAcceptDrops(true);  // 设置窗口接受拖放
        setAutoFillBackground(true);  // 设置窗口自动填
        ↪ 充背景
        setFocusPolicy(Qt::StrongFocus);  // 设置窗口焦
        ↪ 点策略
        m_timerId = startTimer(1000);  // 启动定时器，每
        ↪ 秒触发一次
        m_step = 10;  // 设置步长
        m_colorIndex = 0;  // 设置颜色索引
    }
```

QWidget 使用示例（二）

```
33 protected:
34     // 事件处理函数
35     void timerEvent(QTimerEvent *event) override {
36         qDebug() << "定时器事件处理";
37         setStyleSheet("background-color: " +
       ↪  m_colors[m_colorIndex] + ";");
38         m_colorIndex = (m_colorIndex + 1) % 3;
39     }
40     // 鼠标点击事件
41     void mousePressEvent(QMouseEvent *event) override {
42         qDebug() << "鼠标点击位置:" << event->pos();
43         m_colorIndex = (m_colorIndex + 1) % 3;
44     }
45     // 按键事件
46     void keyPressEvent(QKeyEvent *event) override {
47         qDebug() << "按键事件:" << event->text();
48         const int step = 10;
49         switch(event->key()) {
50             case Qt::Key_Up:
51                 move(pos() - QPoint(0, step));  // 上移
52                 break;
53             case Qt::Key_Down:
54                 move(pos() + QPoint(0, step));  // 下移
55                 break;
56             case Qt::Key_Left:
57                 move(pos() - QPoint(step, 0));  // 左移
58                 break;
59             case Qt::Key_Right:
60                 move(pos() + QPoint(step, 0));  // 右移
61                 break;
62             case Qt::Key_Space:
63                 resize(size() + QSize(step, step));  //
                ↪  放大
64                 break;
65             case Qt::Key_Control:
66                 resize(size() - QSize(step, step));  //
                ↪  缩小
67                 break;
68             default:
```

QWidget 使用示例（三）

```
69                  QWidget::keyPressEvent(event);
70              }
71          }
72          // 拖放事件
73          void dragEnterEvent(QDragEnterEvent *event)
            ↪ override {
74              qDebug() << "拖放事件:" <<
                ↪ event->mimeData()->text();
75          }
76          // 拖放事件
77          void dropEvent(QDropEvent *event) override {
78              qDebug() << "拖放事件:" <<
                ↪ event->mimeData()->text();
79          }
80
81      private:
82          int m_timerId;  // 用于存储定时器 ID，以便后续停止
83          int m_step;  // 用于存储步长
```

```
84          int m_colorIndex;  // 用于存储颜色索引
85          const QString m_colors[3] = {"red", "blue",
            ↪ "green"};  // 用于存储颜色
86      };
87
88      int main(int argc, char *argv[]) {
89          QApplication app(argc, argv);
90          WidgetDemo mainWindow1;
91          mainWindow1.show();
92          WidgetDemo childWindow(&mainWindow1);
93          childWindow.setWindowTitle("子窗口 1");
94          childWindow.resize(200, 150);
95          childWindow.show();
96          // 坐标转换示例
97          QPoint globalPos1 =
            ↪ mainWindow1.mapToGlobal(QPoint(1001, 1001));
98          QPoint localPos1 =
            ↪ mainWindow1.mapFromGlobal(globalPos1);
```

QDialog 详解

- **QDialog** 是 Qt 中用于实现对话框窗口的基类，广泛用于弹出式交互界面，如设置窗口、消息提示、文件选择等。
- QDialog 支持模态（Modal）和非模态（Modeless）两种显示方式。模态对话框会阻塞父窗口的输入，常用于需要用户立即处理的情况；非模态对话框则允许用户继续操作主窗口。
- QDialog 提供了丰富的窗口管理功能，包括显示（`show()`）、隐藏（`hide()`）、移动（`move()`）、调整大小（`resize()`）、设置父子关系、事件处理等。
- 可以通过重写 QDialog 的虚函数（如 `accept()`、`reject()`、`done()`）自定义对话框的关闭行为和结果返回。
- QDialog 通常配合标准按钮（如“确定”、“取消”）使用，支持信号槽机制，便于与主窗口或其他控件交互。
- QDialog 支持布局管理，可以灵活地添加各种控件，实现复杂的对话界面。
- 典型用法：自定义对话框时，继承 QDialog 并实现界面布局、事件处理和业务逻辑。

QDialog 使用示例

```cpp
#include <QApplication>
#include <QDialog>
#include <QPushButton>
#include <QVBoxLayout>
#include <QLabel>
#include <QDebug>
#include <iostream>
class MyDialog : public QDialog
{
    Q_OBJECT
public:
    MyDialog(QWidget *parent = nullptr) :
    ↪ QDialog(parent)
    {
        setWindowTitle("QDialog 示例");
        resize(300, 150);
        QVBoxLayout *layout = new QVBoxLayout(this);
        QLabel *label = new QLabel("这是一个自定义
        ↪ QDialog 对话框", this);
        layout->addWidget(label);
        QPushButton *okBtn = new QPushButton("确定",
        ↪ this);
        QPushButton *cancelBtn = new QPushButton("取
        ↪ 消", this);
        layout->addWidget(okBtn);
        layout->addWidget(cancelBtn);
        connect(okBtn, &QPushButton::clicked, this,
        ↪ &QDialog::accept);
        connect(cancelBtn, &QPushButton::clicked, this,
        ↪ &QDialog::reject);
    }
};

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyDialog dlg;
    int ret = dlg.exec(); // 模态对话框
    if (ret == QDialog::Accepted) {
        qDebug() << "用户点击了确定";
    } else {
        qDebug() << "用户点击了取消";
    }
    std::cin.get();
    return 0;
}
```

QMainWindow 详解

- **QMainWindow** 是 Qt 中用于实现主窗口的基类，通常用于构建应用程序的主界面。
- QMainWindow 提供了丰富的窗口管理功能，包括菜单栏、工具栏、状态栏、中央窗口区域等。
- 中央窗口区域可以放置各种控件，如 QWidget、QDialog 等。
- QMainWindow 支持布局管理，可以灵活地添加各种控件，实现复杂的界面。
- 典型用法：构建应用程序主界面时，继承 QMainWindow 并实现界面布局、事件处理和数据交互逻辑。

QMainWindow 使用示例

```cpp
#include <QApplication>
#include <QMainWindow>
#include <QMenuBar>
#include <QMenu>
#include <QAction>
#include <QStatusBar>
#include <QToolBar>
#include <QLabel>
#include <QDebug>

class MyMainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MyMainWindow(QWidget *parent = nullptr) :
    ↪  QMainWindow(parent)
    {
        setWindowTitle("QMainWindow 示例");
        resize(600, 400);

        // 创建菜单栏
        QMenuBar *menuBar = new QMenuBar(this);
        setMenuBar(menuBar);

        // 文件菜单
        QMenu *fileMenu = menuBar->addMenu("文件");
        QAction *openAction = fileMenu->addAction("打
        ↪  开");
        QAction *exitAction = fileMenu->addAction("退
        ↪  出");

        // 编辑菜单
        QMenu *editMenu = menuBar->addMenu("编辑");
        QAction *copyAction = editMenu->addAction("复
        ↪  制");
        QAction *pasteAction = editMenu->addAction("粘
        ↪  贴");
```

QFrame 详解

- **QFrame** 是 Qt 中用于实现可定制边框和背景的基类控件，常作为其他控件的容器或分隔线。
- QFrame 支持多种边框样式（如 Box、Panel、HLine、VLine、StyledPanel 等），可通过 `setFrameShape()` 和 `setFrameShadow()` 设置形状和阴影效果。
- 可以通过 `setLineWidth()`、`setMidLineWidth()` 等方法调整边框宽度。
- QFrame 支持设置背景色、背景图片等属性，便于实现美观的界面分隔和装饰。
- QFrame 作为基类，常被用来自定义控件，重写其事件处理函数（如 `paintEvent()`）可实现自定义绘制逻辑。
- 典型用法包括：作为分隔线（水平线/垂直线）、面板容器、装饰性边框等。
- QFrame 也常用于布局中，帮助组织和美化界面结构。

QFrame 使用示例

```cpp
#include <QApplication>
#include <QFrame>
#include <QDebug>
class FrameDemo : public QFrame
{
public:
    FrameDemo(QWidget *parent = nullptr,
    ↪ Qt::WindowFlags f = Qt::WindowFlags())
        : QFrame(parent, f)
    {
        setWindowTitle("QFrame 演示");
        resize(400, 250);
        setFrameShape(QFrame::Box); // 设置边框样式
        // QFrame::Box 边框样式
        // QFrame::NoFrame 无边框
        // QFrame::Panel 面板
        // QFrame::StyledPanel 样式化面板
        // QFrame::HLine 水平线
        // QFrame::VLine 垂直线
        // QFrame::WinPanel Windows 2000 风格面板

        setFrameShadow(QFrame::Raised); // 设置边框阴影
        // QFrame::Plain 无阴影
        // QFrame::Raised 凸起
        // QFrame::Sunken 凹陷

        setLineWidth(3); // 设置边框宽度
        setMidLineWidth(1); // 设置中间线宽度
        qDebug() << "frameRect():" << frameRect();
        qDebug() << "frameShadow():" << frameShadow();
        qDebug() << "frameShape():" << frameShape();
        qDebug() << "frameWidth():" << frameWidth();
        qDebug() << "lineWidth():" << lineWidth();
        qDebug() << "midLineWidth():" <<
        ↪ midLineWidth();
    }
```

QPushButton 详解

- **QPushButton** 是 Qt 中常用的按钮控件，适用于各种窗口和对话框，作为用户交互的主要入口。
- 支持设置文本、图标，或图标与文本组合，灵活满足不同界面需求。
- 可通过 `setCheckable()` 设置为可切换（开关）状态，适合实现切换类功能。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QPushButton，设置属性并连接槽函数，实现交互逻辑。

QPushButton 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QLabel>
#include <QHBoxLayout>
#include <QIcon>
class PushButtonDemo : public QWidget
{
public:
    PushButtonDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QPushButton 示例");
        resize(480, 180);
        QLabel *label = new QLabel("请选择按钮操作",
        ↪ this);
        label->setAlignment(Qt::AlignCenter);
        QPushButton *btnNormal = new QPushButton("普通
        ↪ 按钮", this);
        QPushButton *btnIcon = new
        ↪ QPushButton(QIcon::fromTheme("document-open"),
        ↪ "带图标按钮", this);
        QPushButton *btnCheck = new QPushButton("可切换
        ↪ 按钮", this);
        btnCheck->setCheckable(true);
        connect(btnNormal, &QPushButton::clicked, this,
            [=](){ label->setText("点击了普通按钮"); });
        connect(btnIcon, &QPushButton::clicked, this,
            [=](){ label->setText("点击了带图标按钮");
            ↪ });
        connect(btnCheck, &QPushButton::toggled, this,
        ↪ [=](bool checked){
            label->setText(checked ? "切换按钮：已按下"
            ↪ : "切换按钮：未按下");
        });
        QHBoxLayout *layout = new QHBoxLayout(this);
        layout->addWidget(btnNormal);
        layout->addWidget(btnIcon);
        layout->addWidget(btnCheck);
        layout->addWidget(label, 1);
        setLayout(layout);
    }
};
```

QLabel 详解

- **QLabel** 是 Qt 中用于显示文本或图像的控件，常用于界面中的标签、提示信息、图片展示等场景。
- 支持设置文本、图片、对齐方式、字体、颜色等属性。
- 可以显示富文本（HTML）、超链接，也可用作图片显示控件。
- 常与其他控件（如 QLineEdit、QSpinBox 等）配合，用于描述或提示输入内容。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QLabel，设置内容和样式，作为界面静态信息展示。

QLabel 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>

class LabelDemo : public QWidget
{
public:
    LabelDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QLabel 示例");
        resize(400, 250);

        QLabel *label1 = new QLabel("普通文本标签",
        ↪ this);
        label1->setAlignment(Qt::AlignCenter);

        QLabel *label2 = new QLabel("<font
        ↪ color='blue'> 富文本标签 </font>", this);
        label2->setAlignment(Qt::AlignCenter);

        QLabel *label3 = new QLabel("<a
        ↪ href='https://www.qt.io/'>Qt 官网 </a>",
        ↪ this);
        label3->setAlignment(Qt::AlignCenter);
        label3->setOpenExternalLinks(true);

        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(label1);
        vbox->addWidget(label2);
        vbox->addWidget(label3);
        setLayout(vbox);
    }
};
```

QLineEdit 详解

- **QLineEdit** 是 Qt 中用于单行文本输入的控件，常用于表单、搜索框、命令输入等场景。
- 支持设置占位符文本（placeholder）、最大长度、只读、密码模式等属性。
- 可通过 `setText()`、`text()` 设置和获取内容。
- 支持输入校验（Validator）、自动补全（Completer）、输入掩码（InputMask）等高级功能。
- 常用信号：`textChanged`（文本变化）、`returnPressed`（回车确认）、`editingFinished`（编辑完成）。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QLineEdit，设置属性并连接槽函数，实现数据输入与交互。

QLineEdit 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QLineEdit>
#include <QLabel>
#include <QVBoxLayout>
#include <QHBoxLayout>
class LineEditDemo : public QWidget
{
public:
LineEditDemo(QWidget *parent = nullptr) :
    QWidget(parent)
    {
        setWindowTitle("QLineEdit 示例");
        resize(400, 180);
        QLabel *label = new QLabel("请输入内容", this);
        label->setAlignment(Qt::AlignCenter);
        QLineEdit *lineEdit1 = new QLineEdit(this);
        lineEdit1->setPlaceholderText("普通输入框");
        QLineEdit *lineEdit2 = new QLineEdit(this);
        lineEdit2->setPlaceholderText("只读输入框");
        lineEdit2->setText("只读内容");
        lineEdit2->setReadOnly(true);
        QLineEdit *lineEdit3 = new QLineEdit(this);
        lineEdit3->setPlaceholderText("密码输入框");
        lineEdit3->setEchoMode(QLineEdit::Password);
        // 信号槽连接
        connect(lineEdit1, &QLineEdit::textChanged,
        this, [=](const QString &text){
        label->setText("输入内容: " + text); });
        connect(lineEdit3, &QLineEdit::returnPressed,
        this, [=](){label->setText("密码已输入"); });
        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(lineEdit1);
        vbox->addWidget(lineEdit2);
        vbox->addWidget(lineEdit3);
        vbox->addWidget(label, 1);
        setLayout(vbox);
    }
};
```

QTextEdit 详解

- **QTextEdit** 是 Qt 中用于多行富文本编辑和显示的控件，支持纯文本和富文本（HTML）。
- 可用于实现文本编辑器、日志窗口、富文本输入、代码编辑等场景。
- 支持设置字体、颜色、对齐、段落格式、插入图片和超链接等丰富功能。
- 通过 `setPlainText()`、`setHtml()` 设置内容，`toPlainText()`、`toHtml()` 获取内容。
- 支持撤销/重做、查找替换、只读模式、自动换行、拖拽、剪切板操作等。
- 常用信号：`textChanged`（内容变化）、`cursorPositionChanged`（光标变化）。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QTextEdit，设置属性并连接槽函数，实现多行文本输入与显示。

QTextEdit 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QTextEdit>
#include <QLabel>
#include <QVBoxLayout>
#include <QPushButton>

class TextEditDemo : public QWidget
{
public:
    TextEditDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QTextEdit 示例");
        resize(400, 220);
        auto *label = new QLabel("请输入多行文本",
        ↪ this);
        label->setAlignment(Qt::AlignCenter);
        auto *textEdit = new QTextEdit(this);
        textEdit->setPlaceholderText("支持多行文本和富文
        ↪ 本（HTML）...");
        auto *plainBtn = new QPushButton("显示纯文本",
        ↪ this);
        auto *htmlBtn = new QPushButton("显示富文本",
        ↪ this);
        connect(plainBtn, &QPushButton::clicked,
        this, [=](){
        label->setText(textEdit->toPlainText()); });
        connect(htmlBtn, &QPushButton::clicked,
        this, [=](){
        label->setText(textEdit->toHtml()); });
        auto *btnLayout = new QHBoxLayout;
        btnLayout->addWidget(plainBtn);
        btnLayout->addWidget(htmlBtn);
        auto *mainLayout = new QVBoxLayout(this);
        mainLayout->addWidget(textEdit);
        mainLayout->addLayout(btnLayout);
        mainLayout->addWidget(label);
        setLayout(mainLayout);
    }
};
```

QScrollArea 详解

- **QScrollArea** 是 Qt 中用于实现可滚动区域的控件，适用于内容超出可视范围时的显示需求。
- 支持自动显示和隐藏水平、垂直滚动条，能够根据内容和窗口大小智能调整。
- 可通过设置属性自定义滚动条的样式、位置、行为等，提升界面美观性和交互体验。
- 常用场景：图片浏览器、表单、长文本、复杂控件集合等需要滚动显示的界面。
- 使用方法：将需要滚动的控件（如 QWidget、QLabel 等）作为 QScrollArea 的子控件，通过 `setWidget()` 设置。

QScrollArea 使用示例

```cpp
#include <QApplication>
#include <QScrollArea>
#include <QLabel>
#include <QVBoxLayout>
#include <QWidget>
#include <QPalette>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // 创建内容部件，并设置合适的背景色以提升可读性
    QWidget *contentWidget = new QWidget;
    QVBoxLayout *layout = new QVBoxLayout;
    // 批量添加标签，内容更丰富
    for (int i = 1; i <= 30; ++i) {
        QLabel *label = new QLabel(QString("这是第 %1 行内容 - QScrollArea 演示").arg(i));
        label->setMargin(4);
        layout->addWidget(label);
    }
    layout->addStretch(); // 增加弹性空间，优化滚动体验
    contentWidget->setLayout(layout);
    // 设置内容部件的最小宽度，避免内容过窄
    contentWidget->setMinimumWidth(220);
    // 创建 QScrollArea 并设置内容部件
    QScrollArea *scrollArea = new QScrollArea;
    scrollArea->setWindowTitle("QScrollArea 示例");
    scrollArea->resize(350, 250);
    scrollArea->setWidget(contentWidget);
    scrollArea->setWidgetResizable(true); // 内容自适应滚动区域大小
    // 优化滚动条显示策略
    scrollArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    scrollArea->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
    // 设置滚动区域背景色
    QPalette pal = scrollArea->palette();
    pal.setColor(QPalette::Window, QColor(245, 245, 250));
    scrollArea->setAutoFillBackground(true);
    scrollArea->setPalette(pal);
    scrollArea->show();
    return app.exec();
}
```

QSplitter 详解

- **QSplitter** 是 Qt 中用于实现可拖动分割线的控件，常用于分割窗口或容器。
- QSplitter 支持水平和垂直两种分割方式，可通过设置属性自定义分割线的位置、大小、行为等。
- 支持设置分割线的样式、位置、大小等属性。
- 常用场景：窗口分割、布局调整、界面分隔等。
- 使用方法：将需要分割的控件（如 QWidget、QFrame 等）作为 QSplitter 的子控件，通过 `addWidget()` 添加。

QSplitter 使用示例

```cpp
#include <QApplication>
#include <QSplitter>
#include <QTextEdit>
#include <QLabel>
#include <QFrame>
#include <QVBoxLayout>
#include <QWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // 创建主窗口部件
    QWidget window;
    window.setWindowTitle("QSplitter 示例");
    window.resize(500, 300);
    // 创建水平分割器
    QSplitter *splitter = new QSplitter(Qt::Horizontal, &window);
    // 左侧控件：QTextEdit
    QTextEdit *editor = new QTextEdit;
    editor->setPlainText("左侧：QTextEdit\n可以输入多行文本。");
    // 右侧控件：QFrame + QLabel
    QFrame *frame = new QFrame;
    frame->setFrameShape(QFrame::StyledPanel);
    QVBoxLayout *frameLayout = new QVBoxLayout(frame);
    QLabel *label = new QLabel("右侧：QLabel\n可放置任意控件。");
    label->setAlignment(Qt::AlignCenter);
    frameLayout->addWidget(label);
    // 添加控件到分割器
    splitter->addWidget(editor);
    splitter->addWidget(frame);
    // 设置初始分割比例
```

QListWidget 详解

- **QListWidget** 是 Qt 提供的用于显示和管理条目列表的控件，适合用于简单的列表选择、条目管理等场景。
- 支持单选、多选、拖拽排序、条目编辑、图标显示等功能。
- 可通过 `addItem()`、`addItems()`、`insertItem()` 等方法动态添加条目。
- 支持条目自定义（如设置图标、字体、颜色等），也可通过 `QListWidgetItem` 进行高级定制。
- 常用信号：`itemClicked`（条目点击）、`itemDoubleClicked`（双击）、`currentItemChanged`（当前项变化）等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QListWidget，添加条目并连接槽函数，实现列表数据的交互管理。

QListWidget 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QListWidget>
#include <QVBoxLayout>
#include <QLabel>

class ListWidgetDemo : public QWidget
{
public:
    ListWidgetDemo(QWidget *parent = nullptr) :
    ↪  QWidget(parent)
    {
        setWindowTitle("QListWidget 示例");
        resize(400, 250);

        auto *label = new QLabel("请选择一个项目",
        ↪  this);
        label->setAlignment(Qt::AlignCenter);

        auto *listWidget = new QListWidget(this);
        listWidget->addItems({"苹果", "香蕉", "橙子",
        ↪  "葡萄", "西瓜"});

        connect(listWidget,
        ↪  &QListWidget::currentTextChanged, this,
        ↪  [=](const QString &text){
            label->setText("已选择: " + text);
        });

        auto *mainLayout = new QVBoxLayout(this);
        mainLayout->addWidget(listWidget);
        mainLayout->addWidget(label);
        setLayout(mainLayout);
    }
};
```

QTreeWidget 详解

- **QTreeWidget** 是 Qt 提供的树形结构控件，适合用于分层数据的展示与编辑，如文件夹结构、组织架构等。
- 支持多级节点、节点展开/收起、节点复选框、图标、富文本等功能。
- 可通过 `addTopLevelItem()`、`addChild()` 等方法动态添加节点。
- 支持拖拽排序、节点编辑、节点选择（单选/多选）、右键菜单等高级交互。
- 常用信号：`itemClicked`（节点点击）、`itemChanged`（节点内容变化）、`itemExpanded`（节点展开）等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QTreeWidget，设置列数、表头，添加节点并连接槽函数，实现树形数据的交互管理。

QTreeWidget 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QTreeWidget>
#include <QVBoxLayout>
#include <QLabel>

class TreeWidgetDemo : public QWidget
{
public:
    TreeWidgetDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QTreeWidget 示例");
        resize(420, 300);
        auto *label = new QLabel("请选择节点", this);
        label->setAlignment(Qt::AlignCenter);
        auto *tree = new QTreeWidget(this);
        tree->setColumnCount(2);
        tree->setHeaderLabels(QStringList() << "名称"
        ↪ << "描述");
        auto *root1 = new QTreeWidgetItem(tree,
        ↪ QStringList() << "水果" << "各种水果");
        new QTreeWidgetItem(root1, QStringList() << "苹
        ↪ 果" << "红色或绿色");
        new QTreeWidgetItem(root1, QStringList() << "香
        ↪ 蕉" << "黄色，富含钾");
        auto *root2 = new QTreeWidgetItem(tree,
        ↪ QStringList() << "蔬菜" << "各种蔬菜");
        new QTreeWidgetItem(root2, QStringList() << "西
        ↪ 红柿" << "红色，富含维生素 C");
        new QTreeWidgetItem(root2, QStringList() << "黄
        ↪ 瓜" << "绿色，清爽可口");
        tree->expandAll();
        connect(tree, &QTreeWidget::itemClicked, this,
        ↪ [=](QTreeWidgetItem *item, int){
            label->setText(QString("已选择：%1 -
            ↪ %2").arg(item->text(0),
            ↪ item->text(1)));
        });
        auto *vbox = new QVBoxLayout(this);
        vbox->addWidget(tree);
        vbox->addWidget(label, 1);
        setLayout(vbox);
    }
};
```

QTabWidget 详解

- **QTabWidget** 是 Qt 中用于实现选项卡式界面的控件，适合将多个页面或功能模块整合在同一窗口中，便于用户切换和管理。
- 支持动态添加、删除选项卡，每个选项卡可承载任意 QWidget 子控件，如表单、文本、图表等。
- 可自定义选项卡的标签文本、图标、提示信息，支持设置选项卡的位置（上、下、左、右）、可关闭性等属性。
- 适用场景：多文档界面、设置面板、属性编辑器等需要分组展示内容的应用。
- 常用方法：`addTab()`、`removeTab()`、`setTabPosition()`、`setTabsClosable()`等。

QTabWidget 使用示例

```cpp
#include <QApplication>
#include <QTabWidget>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QTextEdit>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // 创建主窗口部件
    QWidget window;
    window.setWindowTitle("QTabWidget 示例");
    window.resize(420, 260);
    // 创建 QTabWidget
    QTabWidget *tabWidget = new QTabWidget(&window);
    // 第一个选项卡：简单标签
    QWidget *tab1 = new QWidget;
    QVBoxLayout *tab1Layout = new QVBoxLayout(tab1);
    QLabel *label1 = new QLabel("这是第一个选项卡内容。
    ↪  ");
    tab1Layout->addWidget(label1);
    tab1Layout->addStretch();
    // 第二个选项卡：文本编辑器
    QWidget *tab2 = new QWidget;
    QVBoxLayout *tab2Layout = new QVBoxLayout(tab2);
    QTextEdit *textEdit = new QTextEdit;
    textEdit->setPlainText("可以在这里输入多行文本。");
    tab2Layout->addWidget(textEdit);
    // 添加选项卡到 QTabWidget
    tabWidget->addTab(tab1, "选项卡一");
    tabWidget->addTab(tab2, "选项卡二");
    // 设置选项卡可关闭
    tabWidget->setTabsClosable(true);
    QObject::connect(tabWidget,
    ↪  &QTabWidget::tabCloseRequested, tabWidget,
    ↪  &QTabWidget::removeTab);
    // 设置选项卡位置（可选：顶部、底部、左侧、右侧）
    tabWidget->setTabPosition(QTabWidget::North);
    // 主布局
    QVBoxLayout *mainLayout = new QVBoxLayout(&window);
    mainLayout->addWidget(tabWidget);
    window.show();
    return app.exec();
}
```

QStackedWidget 详解

- **QStackedWidget** 是 Qt 中用于实现堆叠式界面的控件，适合将多个子窗口堆叠显示，常用于多步骤向导、标签页等场景。
- 支持动态添加、删除子窗口，每个子窗口可承载任意 QWidget 子控件，如表单、文本、图表等。
- 可自定义子窗口的样式、位置、大小等属性。
- 适用场景：多步骤向导、标签页、属性编辑器等需要分组展示内容的应用。
- 常用方法：`addWidget()`、`removeWidget()`、`setCurrentIndex()` 等。

QStackedWidget 使用示例

```cpp
#include <QApplication>
#include <QStackedWidget>
#include <QPushButton>
#include <QVBoxLayout>
#include <QLabel>
#include <QWidget>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("QStackedWidget 示例");
    window.resize(400, 200);

    QStackedWidget *stack = new
    ↪  QStackedWidget(&window);
    stack->addWidget(new QLabel("这是第一个页面"));
    stack->addWidget(new QLabel("这是第二个页面"));

    QPushButton *prev = new QPushButton("上一页");
    QPushButton *next = new QPushButton("下一页");

    QObject::connect(prev, &QPushButton::clicked,
    ↪  [&](){
        int idx = stack->currentIndex();
        if (idx > 0) stack->setCurrentIndex(idx - 1);
    });
    QObject::connect(next, &QPushButton::clicked,
    ↪  [&](){
        int idx = stack->currentIndex();
        if (idx < stack->count() - 1)
        ↪  stack->setCurrentIndex(idx + 1);
    });

    QVBoxLayout *layout = new QVBoxLayout(&window);
    layout->addWidget(stack);
    layout->addStretch();
    layout->addWidget(prev);
    layout->addWidget(next);
    window.show();
    return app.exec();
}
```

QToolBar 详解

- **QToolBar** 是 Qt 中用于实现工具栏的控件，通常用于主窗口（QMainWindow）中，便于用户快速访问常用操作。
- 工具栏可以包含按钮、下拉菜单、分隔符、控件（如下拉框、输入框等），支持图标和文本的灵活组合。
- 支持动态添加、移除 QAction，可以通过 `addAction()`、`addWidget()`、`addSeparator()` 等方法管理工具栏内容。
- 工具栏可设置停靠位置（上、下、左、右），支持浮动、隐藏、拖拽等交互特性。
- 可通过 `setMovable()` 控制工具栏是否可移动，通过 `setFloatable()` 控制是否可浮动。
- 适用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QMainWindow 中创建 QToolBar，添加常用操作按钮，并与槽函数连接，实现快速操作。

QToolBar 使用示例

```cpp
#include <QApplication>
#include <QMainWindow>
#include <QToolBar>
#include <QAction>
#include <QLabel>

class ToolBarDemo : public QMainWindow
{
public:
    ToolBarDemo(QWidget *parent = nullptr) :
    ↪ QMainWindow(parent)
    {
        setWindowTitle("QToolBar 示例");
        resize(500, 300);

        // 简单工具栏
        QToolBar *toolBar = addToolBar("工具栏");
        QAction *openAction = toolBar->addAction("打
        ↪ 开");
        QAction *saveAction = toolBar->addAction("保
        ↪ 存");

        // 中央窗口
        QLabel *label = new QLabel("中央窗口区域",
        ↪ this);
        label->setAlignment(Qt::AlignCenter);
        setCentralWidget(label);

        // 信号槽
        connect(openAction, &QAction::triggered, this,
        ↪ [=](){
            label->setText("点击了打开");
        });
        connect(saveAction, &QAction::triggered, this,
        ↪ [=](){
            label->setText("点击了保存");
        });
    }
};
```

QStatusBar 详解

- **QStatusBar** 是 Qt 中用于实现状态栏的控件，通常用于主窗口（QMainWindow）中，用于显示应用状态、提示信息等。
- 状态栏可以包含文本、图标、进度条等控件，支持动态更新内容和自定义样式。
- QStatusBar 通常固定在主窗口底部（不可浮动），支持临时消息（`showMessage()`）、永久控件（`addPermanentWidget()`）等功能。
- 常用方法：`showMessage()`、`clearMessage()`、`addWidget()`、`addPermanentWidget()` 等。
- 典型用法：在 QMainWindow 中创建 QStatusBar，显示操作提示、进度信息、状态指示等。

QStatusBar 使用示例

```cpp
#include <QApplication>
#include <QMainWindow>
#include <QStatusBar>
#include <QPushButton>
#include <QLabel>
#include <QTimer>

class StatusBarDemo : public QMainWindow
{
public:
    StatusBarDemo(QWidget *parent = nullptr) :
    ↪ QMainWindow(parent)
    {
        setWindowTitle("QStatusBar 示例");
        resize(480, 220);
        // 创建状态栏
        QStatusBar *statusBar = new QStatusBar(this);
        setStatusBar(statusBar);
        // 显示初始信息
        statusBar->showMessage("欢迎使用 QStatusBar 示
        ↪ 例", 3000);
        // 添加永久信息标签
        QLabel *permanentLabel = new QLabel("就绪",
        ↪ this);
        statusBar->addPermanentWidget(permanentLabel);
        // 添加临时信息按钮
        QPushButton *infoBtn = new QPushButton("显示临时
        ↪ 信息", this);
        statusBar->addWidget(infoBtn);
        // 信号槽：点击按钮显示临时信息
        connect(infoBtn, &QPushButton::clicked, this,
        ↪ [=](){
            statusBar->showMessage("这是临时信息，3 秒后
            ↪ 消失", 3000);
        });
        // 中央窗口区域
        QLabel *centerLabel = new QLabel("主窗口内容区
        ↪ 域", this);
        centerLabel->setAlignment(Qt::AlignCenter);
        setCentralWidget(centerLabel);
    }
};
```

QMenu 详解

- **QMenu** 是 Qt 中用于实现下拉菜单的控件，通常用于主窗口（QMainWindow）中，显示菜单项。
- QMenu 可包含多个 QAction，支持图标与文本灵活组合，也可嵌套子菜单（QMenu）。
- 支持通过 `addAction()`、`addMenu()`、`addSeparator()` 等方法动态管理菜单内容。
- 菜单项可与槽函数连接，实现响应式操作。
- QMenu 可独立弹出（如右键菜单），也可作为菜单栏（QMenuBar）子菜单使用。
- 常见场景：主菜单、右键上下文菜单、功能分组菜单等。
- 典型用法：在 QMainWindow 中创建 QMenuBar，添加 QMenu，再添加 QAction，并与槽函数连接。

QMenuBar 详解

- **QMenuBar** 是 Qt 中用于实现菜单栏的控件，通常用于主窗口（QMainWindow）中，显示菜单项。
- QMenuBar 可包含多个 QMenu，支持图标与文本灵活组合，也可嵌套子菜单（QMenu）。
- 支持通过 `addMenu()`、`addSeparator()` 等方法动态管理菜单栏内容。
- 菜单栏可设置停靠位置（上、下、左、右），支持浮动、隐藏、拖拽等交互特性。
- 可通过 `setMovable()` 控制菜单栏是否可移动，通过 `setFloatable()` 控制是否可浮动。
- 适用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QMainWindow 中创建 QMenuBar，添加 QMenu，再添加 QAction，并与槽函数连接。

QMenu / QMenuBar 使用示例

```cpp
#include <QApplication>
#include <QMainWindow>
#include <QMenuBar>
#include <QMenu>
#include <QAction>
#include <QLabel>
#include <QMessageBox>
class MenuDemo : public QMainWindow
{
public:
    MenuDemo(QWidget *parent = nullptr) :
    ↪ QMainWindow(parent)
    {
        setWindowTitle("QMenu 示例");
        resize(480, 260);
        // 创建菜单栏
        QMenuBar *menuBar = new QMenuBar(this);
        setMenuBar(menuBar);
        // 文件菜单
        QMenu *fileMenu = menuBar->addMenu("文件
        ↪ (&F)");
        QAction *newAction = fileMenu->addAction("新建
        ↪ (&N)");
        fileMenu->addSeparator();
        QAction *exitAction = fileMenu->addAction("退出
        ↪ (&Q)");
        // 编辑菜单
        QMenu *editMenu = menuBar->addMenu("编辑
        ↪ (&E)");
        QAction *copyAction = editMenu->addAction("复制
        ↪ (&C)");
        // 中央窗口
        QLabel *label = new QLabel("请选择菜单项进行操
        ↪ 作", this);
        label->setAlignment(Qt::AlignCenter);
        setCentralWidget(label);
        // 信号槽连接
        connect(newAction, &QAction::triggered,
        this, [=](){ label->setText("新建操作"); });
        connect(exitAction, &QAction::triggered,
        this, [=](){ close(); });
        connect(copyAction, &QAction::triggered,
        this, [=](){ label->setText("复制操作"); });
    }
};
```

QToolButton 详解

- **QToolButton** 是 Qt 提供的专用工具按钮控件，常用于主窗口（QMainWindow）或工具栏（QToolBar）中，作为功能快捷入口。
- 支持同时显示图标和文本，可通过 `setIcon()`、`setText()` 灵活设置。
- 可通过 `setToolButtonStyle()` 控制图标与文本的排列方式（如仅图标、仅文本、图标在上/左等）。
- 支持 `setAutoRaise()`，实现扁平风格和悬浮高亮效果，提升界面美观性。
- 通过 `setPopupMode()` 可设置弹出菜单模式（如即时弹出、延迟弹出、仅菜单）。
- 可用 `setDefaultAction()` 绑定 QAction，实现与菜单栏、工具栏一致的行为。
- 支持 `setCheckable()`，可设置为可切换（开关）状态，适合工具属性切换。
- 常见应用：工具栏按钮、带下拉菜单的操作按钮、属性切换按钮等。

QToolButton 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QToolButton>
#include <QMenu>
#include <QHBoxLayout>
#include <QLabel>
class ToolButtonDemo : public QWidget
{
public:
ToolButtonDemo(QWidget *parent = nullptr) :
    QWidget(parent)
    {
        setWindowTitle("QToolButton 示例");
        resize(480, 180);
        QLabel *label = new QLabel("请选择工具按钮操作",
            this);
        label->setAlignment(Qt::AlignCenter);
        QToolButton *toolBtn = new QToolButton(this);
        toolBtn->setText("工具按钮");
toolBtn->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
toolBtn->setIcon(QIcon::fromTheme("document-open"));
        toolBtn->setAutoRaise(true);
        toolBtn->setCheckable(true);
        QMenu *menu = new QMenu(this);
        QAction *actionA = menu->addAction("操作 A");
        QAction *actionB = menu->addAction("操作 B");
        toolBtn->setMenu(menu);
toolBtn->setPopupMode(QToolButton::MenuButtonPopup);
        connect(toolBtn, &QToolButton::clicked,
        this, [=](bool checked){
        label->setText(checked ? "已按下" : "未按下");
        });
        connect(actionA, &QAction::triggered, this,
            [=](){ label->setText("选择了操作 A"); });
        connect(actionB, &QAction::triggered, this,
            [=](){ label->setText("选择了操作 B"); });
        QHBoxLayout *layout = new QHBoxLayout(this);
        layout->addWidget(toolBtn);
        layout->addWidget(label, 1);
        setLayout(layout);
    }
};
```

QRadioButton 详解

- **QRadioButton** 是 Qt 中用于实现单选按钮的控件，通常用于主窗口（QMainWindow）中，用于显示单选按钮。
- 支持设置文本、图标，或图标与文本组合，灵活满足不同界面需求。
- 可通过 `setChecked()` 设置为选中状态，通过 `isChecked()` 获取选中状态。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QRadioButton，设置属性并连接槽函数，实现交互逻辑。

QRadioButton 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QRadioButton>
#include <QButtonGroup>
#include <QLabel>
#include <QVBoxLayout>
#include <QHBoxLayout>
class RadioButtonDemo : public QWidget
{
public:
    RadioButtonDemo(QWidget *parent = nullptr) : QWidget(parent)
    {
        setWindowTitle("QRadioButton 示例");
        resize(400, 180);
        QLabel *label = new QLabel("请选择一个选项", this);
        label->setAlignment(Qt::AlignCenter);
        QRadioButton *radio1 = new QRadioButton("选项一", this);
        QRadioButton *radio2 = new QRadioButton("选项二", this);
        // 分组，确保单选
        QButtonGroup *group = new QButtonGroup(this);
        group->addButton(radio1, 1);
        group->addButton(radio2, 2);
        // 信号槽连接
connect(radio1, &QRadioButton::toggled, this, [=](bool checked){
        if (checked) label->setText("已选择：选项一");
    });
connect(radio2, &QRadioButton::toggled, this, [=](bool checked){
        if (checked) label->setText("已选择：选项二");
    });
        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(radio1);
        vbox->addWidget(radio2);
        QHBoxLayout *hbox = new QHBoxLayout(this);
        hbox->addLayout(vbox);
        hbox->addWidget(label, 1);
        setLayout(hbox);
    }
};
```

QCheckBox 详解

- **QCheckBox** 是 Qt 中用于实现复选框的控件，通常用于主窗口（QMainWindow）中，用于显示复选框。
- 支持设置文本、图标，或图标与文本组合，灵活满足不同界面需求。
- 可通过 `setChecked()` 设置为选中状态，通过 `isChecked()` 获取选中状态。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QCheckBox，设置属性并连接槽函数，实现交互逻辑。

QCheckBox 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QCheckBox>
#include <QLabel>
#include <QHBoxLayout>
#include <QVBoxLayout>
class CheckBoxDemo : public QWidget
{
public:
    CheckBoxDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QCheckBox 示例");
        resize(400, 180);
    QLabel *label = new QLabel("请选择复选框", this);
    label->setAlignment(Qt::AlignCenter);
    QCheckBox *check1 = new QCheckBox("选项一", this);
    QCheckBox *check2 = new QCheckBox("选项二", this);
    // 信号槽连接
    auto updateLabel = [=]() {
        QStringList checked;
        if (check1->isChecked()) checked << "选项一";
        if (check2->isChecked()) checked << "选项二";
        if (checked.isEmpty())
            label->setText("未选择任何选项");
        else
            label->setText("已选择: " + checked.join(",
            ↪ "));
    };
    connect(check1, &QCheckBox::toggled, this,
    ↪ updateLabel);
    connect(check2, &QCheckBox::toggled, this,
    ↪ updateLabel);
        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(check1);
        vbox->addWidget(check2);
        QHBoxLayout *hbox = new QHBoxLayout(this);
        hbox->addLayout(vbox);
        hbox->addWidget(label, 1);
        setLayout(hbox);
    }
};
```

QComboBox 详解

- **QComboBox** 是 Qt 中用于实现下拉列表的控件，通常用于主窗口（QMainWindow）中，用于显示下拉列表。
- 支持设置文本、图标，或图标与文本组合，灵活满足不同界面需求。
- 可通过 `addItem()` 添加选项，通过 `currentText()` 获取当前选项。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QComboBox，设置属性并连接槽函数，实现交互逻辑。

QComboBox 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QComboBox>
#include <QLabel>
#include <QHBoxLayout>
#include <QVBoxLayout>

class ComboBoxDemo : public QWidget
{
public:
    ComboBoxDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QComboBox 示例");
        resize(400, 180);
        QLabel *label = new QLabel("请选择一个选项",
        ↪ this);
        label->setAlignment(Qt::AlignCenter);
        QComboBox *combo = new QComboBox(this);
        combo->addItem("选项一");
        combo->addItem("选项二");
        combo->addItem("选项三");
        // 信号槽连接
        connect(combo, &QComboBox::currentTextChanged,
        ↪ this, [=](const QString &text){
            label->setText("已选择: " + text);
        });
        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(combo);
        vbox->addWidget(label, 1);
        setLayout(vbox);
    }
};
```

QSpinBox 详解

- **QSpinBox** 是 Qt 中用于实现计数器的控件，通常用于主窗口（QMainWindow）中，用于显示计数器。
- 支持设置文本、图标，或图标与文本组合，灵活满足不同界面需求。
- 可通过 `setRange()` 设置范围，通过 `setSingleStep()` 设置步长。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QSpinBox，设置属性并连接槽函数，实现交互逻辑。

QSpinBox 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QSpinBox>
#include <QLabel>
#include <QHBoxLayout>
#include <QVBoxLayout>

class SpinBoxDemo : public QWidget
{
public:
    SpinBoxDemo(QWidget *parent = nullptr) :
    ↪ QWidget(parent)
    {
        setWindowTitle("QSpinBox 示例");
        resize(400, 180);
        QLabel *label = new QLabel("当前值: 0", this);
        label->setAlignment(Qt::AlignCenter);
        QSpinBox *spinBox = new QSpinBox(this);
        spinBox->setRange(0, 100);
        spinBox->setSingleStep(5);
        spinBox->setValue(0);
        spinBox->setSuffix(" 个");
        // 信号槽连接
        connect(spinBox,
        ↪ QOverload<int>::of(&QSpinBox::valueChanged),
        ↪ this, [=](int value){
            label->setText(QString("当前值:
            ↪ %1").arg(value));
        });
        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(spinBox);
        vbox->addWidget(label, 1);
        setLayout(vbox);
    }
};
```

QDoubleSpinBox 详解

- **QDoubleSpinBox** 是 Qt 中用于实现双精度浮点数计数器的控件，通常用于主窗口（QMainWindow）中，用于显示双精度浮点数计数器。
- 支持设置文本、图标，或图标与文本组合，灵活满足不同界面需求。
- 可通过 `setRange()` 设置范围，通过 `setSingleStep()` 设置步长，通过 `setDecimals()` 设置小数位数。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QDoubleSpinBox，设置属性并连接槽函数，实现交互逻辑。

QDoubleSpinBox 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QDoubleSpinBox>
#include <QLabel>
#include <QHBoxLayout>
#include <QVBoxLayout>

class DoubleSpinBoxDemo : public QWidget
{
public:
    DoubleSpinBoxDemo(QWidget *parent = nullptr) :
    ↪  QWidget(parent)
    {
        setWindowTitle("QDoubleSpinBox 示例");
        resize(400, 180);
        QLabel *label = new QLabel("当前值: 0.00",
        ↪  this);
        label->setAlignment(Qt::AlignCenter);
        QDoubleSpinBox *doubleSpinBox = new
        ↪  QDoubleSpinBox(this);
        doubleSpinBox->setRange(-100.0, 100.0);
        doubleSpinBox->setSingleStep(0.5);
        doubleSpinBox->setDecimals(2);
        doubleSpinBox->setValue(0.0);
        doubleSpinBox->setSuffix(" 元");
        // 信号槽连接
        connect(doubleSpinBox,

            ↪  QOverload<double>::of(&QDoubleSpinBox::valueChan
            this, [=](double value){
            label->setText(QString("当前值:
            ↪  %1").arg(value, 0, 'f', 2));
        });
        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(doubleSpinBox);
        vbox->addWidget(label, 1);
        setLayout(vbox);
    }
};
```

QDateTimeEdit 详解

- **QDateTimeEdit** 是 Qt 中用于实现日期和时间编辑器的控件，通常用于主窗口（QMainWindow）中，用于显示日期和时间编辑器。
- **QDateEdit** 是 Qt 中用于实现日期编辑器的控件，通常用于主窗口（QMainWindow）中，用于显示日期编辑器。
- **QTimeEdit** 是 Qt 中用于实现时间编辑器的控件，通常用于主窗口（QMainWindow）中，用于显示时间编辑器。
- 支持设置日期和时间，或日期和时间组合，灵活满足不同界面需求。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QDateTimeEdit，设置属性并连接槽函数，实现交互逻辑。

QDateTimeEdit 使用示例

```
15        setWindowTitle("QDateTimeEdit 示例");
16        resize(400, 180);
17        QLabel *label = new QLabel("请选择日期和时间",
          ↪  this);
18        label->setAlignment(Qt::AlignCenter);
19        QDateTimeEdit *dateTimeEdit = new
          ↪  QDateTimeEdit(QDateTime::currentDateTime(),
          ↪  this);
20        dateTimeEdit->setDisplayFormat("yyyy-MM-dd
          ↪  HH:mm:ss");
21        dateTimeEdit->setCalendarPopup(true);
22        QDateEdit *dateEdit = new
          ↪  QDateEdit(QDate::currentDate(), this);
23        dateEdit->setDisplayFormat("yyyy-MM-dd");
24        dateEdit->setCalendarPopup(true);
25        QTimeEdit *timeEdit = new
          ↪  QTimeEdit(QTime::currentTime(), this);
26        timeEdit->setDisplayFormat("HH:mm:ss");
27        timeEdit->setCalendarPopup(true);
28        // 信号槽连接
```

```
29        connect(dateTimeEdit,
          ↪  &QDateTimeEdit::dateTimeChanged, this,
          ↪  [=](const QDateTime &dt){
30            label->setText("已选择: " +
              ↪  dt.toString("yyyy-MM-dd HH:mm:ss"));
31        });
32        connect(dateEdit, &QDateEdit::dateChanged,
          ↪  this, [=](const QDate &date){
33            label->setText("已选择: " +
              ↪  date.toString("yyyy-MM-dd"));
34        });
35        connect(timeEdit, &QTimeEdit::timeChanged,
          ↪  this, [=](const QTime &time){
36            label->setText("已选择: " +
              ↪  time.toString("HH:mm:ss"));
37        });
38        QHBoxLayout *hbox = new QHBoxLayout;
39        hbox->addWidget(dateTimeEdit);
40        hbox->addWidget(dateEdit);
41        hbox->addWidget(timeEdit);
42        QVBoxLayout *vbox = new QVBoxLayout;
43        vbox->addLayout(hbox);
44        vbox->addWidget(label, 1);
45        setLayout(vbox);
```

QCalendarWidget 详解

- **QCalendarWidget** 是 Qt 中用于实现日历的控件，通常用于主窗口（QMainWindow）中，用于显示日历。
- 支持设置日期，灵活满足不同界面需求。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：主窗口常用操作入口、编辑器快捷操作、图形界面工具集等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QCalendarWidget，设置属性并连接槽函数，实现交互逻辑。

QCalendarWidget 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QCalendarWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QDate>

class CalendarWidgetDemo : public QWidget
{
public:
    CalendarWidgetDemo(QWidget *parent = nullptr) :
    ↪  QWidget(parent)
    {
        setWindowTitle("QCalendarWidget 示例");
        resize(400, 250);

        QLabel *label = new QLabel("请选择日期", this);
        label->setAlignment(Qt::AlignCenter);
        QCalendarWidget *calendar = new
        ↪  QCalendarWidget(this);
        calendar->setGridVisible(true);
        calendar->setFirstDayOfWeek(Qt::Monday);

        ↪  calendar->setSelectedDate(QDate::currentDate());

        // 信号槽连接
        connect(calendar,
        ↪  &QCalendarWidget::selectionChanged,
        this, [=](){
        QDate date = calendar->selectedDate();
        label->setText("已选择: " +
        ↪  date.toString("yyyy-MM-dd"));
        });

        QVBoxLayout *vbox = new QVBoxLayout;
        vbox->addWidget(calendar);
        vbox->addWidget(label, 1);
        setLayout(vbox);
    }
};
```

QSlider 详解

- **QSlider** 是 Qt 中用于实现滑动条的控件，常用于让用户在一定范围内选择数值。
- 支持水平（Horizontal）和垂直（Vertical）两种方向。
- 可设置最小值、最大值、步长、初始值等属性。
- 支持信号槽机制，常用信号有 `valueChanged(int)`，可用于实时响应用户操作。
- 应用场景：音量调节、进度控制、参数调整等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QSlider，设置属性并连接槽函数，实现交互逻辑。

QSlider 使用示例

```cpp
#include <QApplication>
#include <QWidget>
#include <QSlider>
#include <QLabel>
#include <QVBoxLayout>
#include <QHBoxLayout>
class SliderDemo : public QWidget {
public:
SliderDemo(QWidget *parent = nullptr) : QWidget(parent) {
    setWindowTitle("QSlider 示例");
    resize(400, 180);
    auto *label = new QLabel("当前值: 50", this);
    label->setAlignment(Qt::AlignCenter);
    auto *hSlider = new QSlider(Qt::Horizontal, this);
    auto *vSlider = new QSlider(Qt::Vertical, this);
    hSlider->setRange(0, 100);  vSlider->setRange(0, 100);
    hSlider->setValue(50);  vSlider->setValue(50);
    hSlider->setTickInterval(10);
    vSlider->setTickInterval(10);
    hSlider->setTickPosition(QSlider::TicksBelow);
    vSlider->setTickPosition(QSlider::TicksLeft);
    // 只需一个信号槽即可同步两个滑块
    connect(hSlider, &QSlider::valueChanged, vSlider, &QSlider::setValue);
    connect(vSlider, &QSlider::valueChanged, hSlider, &QSlider::setValue);
    connect(hSlider, &QSlider::valueChanged, label, [=](int value){
    label->setText(QString("当前值: %1").arg(value));
    });
    connect(vSlider, &QSlider::valueChanged, label, [=](int value){
    label->setText(QString("当前值: %1").arg(value));
    });
    auto *hLayout = new QHBoxLayout;
    hLayout->addWidget(vSlider);
    hLayout->addWidget(hSlider, 1);
    auto *mainLayout = new QVBoxLayout(this);
    mainLayout->addLayout(hLayout);
    mainLayout->addWidget(label);
    setLayout(mainLayout);
}};
```

QScrollBar 详解

- **QScrollBar** 是 Qt 中用于实现滚动条的控件，常用于让用户在一定范围内选择数值。
- 支持水平（Horizontal）和垂直（Vertical）两种方向。
- 可设置最小值、最大值、步长、初始值等属性。
- 支持信号槽机制，常用信号有 `valueChanged(int)`，可用于实时响应用户操作。
- 应用场景：音量调节、进度控制、参数调整等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QScrollBar，设置属性并连接槽函数，实现交互逻辑。

QScrollBar 使用示例

```
#include <QApplication>
#include <QWidget>
#include <QScrollBar>
#include <QLabel>
#include <QVBoxLayout>
#include <QHBoxLayout>
class ScrollBarDemo : public QWidget {
public:
ScrollBarDemo(QWidget *parent = nullptr) :
↪  QWidget(parent) {
setWindowTitle("QScrollBar 示例");
resize(400, 180);
auto *label = new QLabel("当前值: 50", this);
label->setAlignment(Qt::AlignCenter);
auto *hScrollBar = new QScrollBar(Qt::Horizontal,
↪  this);
auto *vScrollBar = new QScrollBar(Qt::Vertical, this);
hScrollBar->setRange(0, 100); vScrollBar->setRange(0,
↪  100);
hScrollBar->setValue(50); vScrollBar->setValue(50);
hScrollBar->setPageStep(10);
↪  vScrollBar->setPageStep(10);
connect(hScrollBar, &QScrollBar::valueChanged,
↪  vScrollBar, &QScrollBar::setValue);
connect(vScrollBar, &QScrollBar::valueChanged,
↪  hScrollBar, &QScrollBar::setValue);
connect(hScrollBar, &QScrollBar::valueChanged, label,
↪  [=](int v){ label->setText(QString("当前值:
↪  %1").arg(v)); });
connect(vScrollBar, &QScrollBar::valueChanged, label,
↪  [=](int v){ label->setText(QString("当前值:
↪  %1").arg(v)); });
auto *hLayout = new QHBoxLayout;
hLayout->addWidget(vScrollBar);
hLayout->addWidget(hScrollBar, 1);
auto *mainLayout = new QVBoxLayout(this);
mainLayout->addLayout(hLayout);
mainLayout->addWidget(label);
setLayout(mainLayout);
    }
};
```

QProgressBar 详解

- **QProgressBar** 是 Qt 中用于实现进度条的控件，常用于显示任务进度。
- 支持水平（Horizontal）和垂直（Vertical）两种方向。
- 可设置最小值、最大值、步长、初始值等属性。
- 支持信号槽机制，常用信号有 `valueChanged(int)`，可用于实时响应用户操作。
- 应用场景：任务进度、文件下载、数据传输等。
- 典型用法：在 QWidget 或 QMainWindow 中创建 QProgressBar，设置属性并连接槽函数，实现交互逻辑。

QProgressBar 使用示例

```
14 auto *label = new QLabel("点击“开始”模拟进度", this);
15 label->setAlignment(Qt::AlignCenter);
16 auto *progressBar = new QProgressBar(this);
17 progressBar->setRange(0, 100);
18 auto *startBtn = new QPushButton("开始", this);
19 auto *resetBtn = new QPushButton("重置", this);
20 auto *timer = new QTimer(this);
21 timer->setInterval(50);
22 connect(startBtn, &QPushButton::clicked, this, [=]() {
23     if (!timer->isActive() && value < 100)
24             timer->start(); });
25 connect(resetBtn, &QPushButton::clicked, this, [=]() {
26     timer->stop();
27     value = 0;
28     progressBar->reset();
```

```
29     label->setText("点击“开始”模拟进度"); });
30 connect(timer, &QTimer::timeout, this, [=]() {
31     if (value < 100) {
32         progressBar->setValue(++value);
33         label->setText(QString("当前进度:
       ↪  %1%").arg(value));
34     } else {
35         timer->stop();
36         label->setText("进度完成！");
37     } });
38 auto *layout = new QVBoxLayout(this);
39 layout->addWidget(label);
40 layout->addWidget(progressBar);
41 layout->addWidget(startBtn);
42 layout->addWidget(resetBtn);
43 setLayout(layout);
```

Qt Designer 详解

- **Qt Designer** 是 Qt 官方提供的可视化界面设计工具，专用于快速开发和设计 Qt 应用程序的 GUI 界面。
- 支持拖拽式控件布局，可直观添加、排列和配置各种 Qt 控件（如按钮、标签、输入框等）。
- 内置多种布局管理器（如 QHBoxLayout、QVBoxLayout、QGridLayout），便于实现响应式和自适应界面。
- 可通过属性编辑器设置控件属性，支持信号与槽的可视化连接，简化界面交互逻辑的搭建。
- 支持样式表（QSS）编辑，可实时预览控件美化效果，提升界面美观性。
- 生成的.ui 文件可直接在 Qt Creator 或 C++/Python 项目中加载，支持与代码逻辑分离，便于团队协作和后期维护。
- 适用于原型设计、界面快速迭代、跨平台 GUI 开发等多种场景，是 Qt 开发的重要辅助工具。

QSS 详解

- **QSS** 是 Qt 中用于实现样式表的控件，常用于美化界面。
- 支持设置样式表，灵活满足不同界面需求。
- 支持信号槽机制，常用于触发操作、提交表单、切换状态等场景。
- 应用场景：界面美化、主题切换、样式定制等。

QSS 使用示例（一）

```cpp
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget window;
    window.setWindowTitle("QSS 样式表示例");
    // 创建控件
    QLabel *label = new QLabel("欢迎使用 QSS 美化界面");
    QPushButton *btn1 = new QPushButton("普通按钮");
    QPushButton *btn2 = new QPushButton("高亮按钮");
    btn2->setObjectName("btn2"); // 优化：提前设置 objectName
    // 设置布局
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(label);
    layout->addWidget(btn1);
    layout->addWidget(btn2);
    layout->setSpacing(16); // 优化：增加控件间距
    layout->setContentsMargins(24, 24, 24, 24); // 优化：设置窗口边距
    window.setLayout(layout);
    // 优化：QSS 样式结构更清晰，注释更明确
    QString qss = R"(
        /* 窗口背景 */
        QWidget {
            background-color: #f0f0f0;
        }
        /* 标签样式 */
        QLabel {
            color: #2d8cf0;
            font-size: 20px;
            font-weight: bold;
            padding: 8px;
        }
        /* 普通按钮样式 */
```

QSS 使用示例（二）

```
        QPushButton {
            background-color: #ffffff;
            border: 2px solid #2d8cf0;
            border-radius: 8px;
            color: #2d8cf0;
            font-size: 16px;
            padding: 6px 18px;
            min-width: 100px;
            min-height: 32px;
            transition: all 0.2s;
        }
        QPushButton:hover {
            background-color: #2d8cf0;
            color: #ffffff;
        }
        QPushButton:pressed {
            background-color: #1565c0;
            color: #ffffff;
        }
        /* 高亮按钮样式 */
        QPushButton#btn2 {
            background-color: #ff9800;
            color: #fff;
            border: 2px solid #ff9800;
        }
        QPushButton#btn2:hover {
            background-color: #ffa726;
            border-color: #ffa726;
        }
        QPushButton#btn2:pressed {
            background-color: #ef6c00;
            border-color: #ef6c00;
        }
    )";
    window.setStyleSheet(qss);
    window.resize(360, 220); // 优化：更合适的窗口大小
    window.show();
    return app.exec();
}
```

目录

元对象系统概述

什么是元对象系统?

元对象系统（Meta-Object System）是 Qt 框架中一个核心的、强大的基础设施。它为 Qt 提供了标准 C++ 本身不具备的一些高级功能，使得 Qt 应用程序更加灵活、动态和易于开发。

简单来说，元对象系统是 Qt 在 C++ 基础上”扩展”出来的一套机制，它让对象能够”知道”关于自身的一些信息（如类名、属性、方法等），并支持对象间的动态通信。

元对象系统的作用

- 提供运行时类型信息
- 支持信号槽机制
- 实现属性系统
- 支持动态属性
- 提供反射能力

核心组件

- `Q_OBJECT` 宏
- `moc`（Meta-Object Compiler）
- `QMetaObject` 类
- `QObject` 基类

Q_OBJECT 宏

```cpp
#include <QObject>
#include <QString>

class MyClass : public QObject
{
    Q_OBJECT  // 启用元对象系统

public:
    explicit MyClass(QObject *parent = nullptr);

    // 属性声明
    Q_PROPERTY(QString name READ name WRITE setName
    ↪  NOTIFY nameChanged)
    Q_PROPERTY(int age READ age WRITE setAge NOTIFY
    ↪  ageChanged)

    // 属性访问方法
    QString name() const { return m_name; }
    void setName(const QString &name);

    int age() const { return m_age; }
    void setAge(int age);

    void setAge(int age);

signals:
    // 信号声明
    void nameChanged(const QString &name);
    void ageChanged(int age);

private slots:
    // 槽函数声明
    void onNameChanged(const QString &name);

private:
    QString m_name;
    int m_age;
};
```

Meta-Object Compiler（moc）元对象编译器

moc 的作用

- **moc（Meta-Object Compiler）**是 Qt 框架中专门用于处理元对象系统的工具。
- 它会扫描 C++ 头文件，自动生成实现信号、槽、属性等元对象相关功能的代码。
- 只有包含 `Q_OBJECT` 宏的类才需要 moc 处理。
- moc 生成的代码会被编译并链接到最终的程序中，实现运行时类型信息、信号槽机制等高级特性。
- 开发者无需手动编写这些元对象相关的底层代码，极大提升了开发效率和代码安全性。

moc 的运行过程

- 扫描头文件，找到包含 `Q_OBJECT` 宏的类。
- 生成包含信号、槽、属性等元对象相关功能的代码。
- 将生成的代码编译并链接到最终的程序中。

QMetaObject 类

QMetaObject 类的作用

- QMetaObject 类是 Qt 元对象系统的核心，负责描述和管理 QObject 派生类的元信息。
- 通过 QMetaObject，可以在运行时获取类名、属性、信号、槽等信息，实现反射和动态调用。

QMetaObject 类的常用方法

- `className()`：返回类的名称。
- `indexOfMethod(const char *method)`：返回指定方法的索引。
- `indexOfSignal(const char *signal)`：返回指定信号的索引。
- `invokeMethod(QObject *obj, const char *member, ...)`：在运行时调用对象的方法（槽函数）。
- `propertyCount()`、`methodCount()`：获取属性和方法的数量。
- `property(int index)`、`method(int index)`：通过索引获取属性或方法的元信息。

QObject 基类

QObject 基类的作用

- **QObject** 是 Qt 框架中所有对象的基类，绝大多数 Qt 类都直接或间接继承自它。
- 提供了信号槽机制、事件处理、对象树（父子关系）、动态属性、对象名称等核心功能。
- 支持对象间通信、自动内存管理（父对象析构时自动析构子对象），便于资源管理。
- 通过 `Q_OBJECT` 宏启用元对象系统，支持运行时类型信息和反射。

QObject 基类的常用方法

- `connect()`：连接信号和槽。
- `disconnect()`：断开信号和槽的连接。
- `emit()`：发送信号。
- `parent()`：获取父对象。
- `children()`：获取子对象。
- `setProperty()`：设置属性。
- `property()`：获取属性。
- `metaObject()`：获取元对象。
- `inherits()`：检查是否继承自指定类。
- `findChild()`：查找子对象。
- `event()`：事件处理。

目录

事件

事件的概念

- 事件（Event）是 GUI 编程中用于描述用户操作（如鼠标点击、键盘输入）、系统消息（如窗口重绘、定时器超时）等的对象。
- 事件机制实现了对象与外部交互的基本方式，是 GUI 响应用户操作的基础。

事件的缺陷

- 事件处理函数通常需要重写基类方法，导致代码分散、可维护性差。
- 事件只能在对象内部处理，难以实现对象间的灵活通信。
- 事件机制不支持多对象同时响应同一事件，扩展性有限。
- 事件的传递和处理流程较为复杂，难以实现解耦和灵活的响应机制。
- 难以动态连接和断开事件响应，缺乏运行时的灵活性。

C++ 事件示例

```cpp
#include <iostream>
// 定义一个函数，作为事件发生时的响应（回调函数）
void onButtonClick() {
    std::cout << "Button is clicked!" << std::endl;
}
// 模拟一个“按钮”对象，它有一个点击事件
class Button {
public:
    // 保存事件回调函数的指针
    void (*onClick)() = nullptr;
    // 模拟点击按钮的动作
    void click() {
        std::cout << "Button is being clicked..." <<
        ↪ std::endl;
        if (onClick) {
            onClick();  // 触发事件
        }
    }
};
```

```cpp
// 主函数演示
int main() {
    Button btn;

    // 注册事件：把 onButtonClick 函数赋给 onClick
    btn.onClick = onButtonClick;

    // 模拟用户点击
    btn.click();
    // 等待用户输入
    std::cin.get();
    return 0;
}
```

Qt 事件示例

```cpp
#include <QCoreApplication>
#include <QEvent>
#include <QObject>
#include <QDebug>
// 自定义一个按钮点击事件类型
const QEvent::Type ButtonClickEventType = static_cast<QEvent::Type>(QEvent::User + 1);
// 自定义事件类
class ButtonClickEvent : public QEvent {
public:
    ButtonClickEvent() : QEvent(ButtonClickEventType) {}
};
// 按钮类，继承自 QObject
class Button : public QObject {
    Q_OBJECT
public:
    // 发送点击事件
    void click() {
        qDebug() << "按钮: 正在被点击...";
        // 创建事件对象
        ButtonClickEvent event;
        // 发送事件到自身
        QCoreApplication::sendEvent(this, &event);
    }
protected:
    // 重写事件处理函数
    bool event(QEvent *e) override {
        if (e->type() == ButtonClickEventType) {
            qDebug() << "按钮被点击了！（事件处理）";
            return true;
        }
        return QObject::event(e);
    }
};
int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);
    Button btn;
    // 模拟用户点击
    btn.click();
    return app.exec();
}
```

信号槽概念

- **信号（Signal）**：对象在特定状态或事件发生时发出的通知。
- **槽（Slot）**：用于接收和处理信号的普通成员函数或全局函数，是对信号的响应动作。
- **连接（Connection）**：将信号与一个或多个槽函数绑定起来，信号发出时自动调用对应的槽。
- **解耦**：信号的发送者和槽的接收者无需相互了解，提升代码的灵活性和可维护性。
- **对象间通信**：信号槽机制是 Qt 实现对象间通信的核心方式，支持跨线程、跨对象的消息传递。
- **动态连接与断开**：可以在运行时动态地连接和断开信号与槽，满足复杂场景下的需求。
- **类型安全**：编译器会检查信号和槽的参数类型，避免类型不匹配导致的错误。
- **松耦合设计**：发送者和接收者完全解耦，便于模块化开发和单元测试。
- **支持多对多连接**：一个信号可以连接多个槽，一个槽也可以接收多个信号。
- **自动内存管理**：当对象销毁时，相关的信号槽连接会自动断开。
- **线程安全**：Qt 支持跨线程的信号槽通信，自动进行事件队列投递，简化多线程编程。
- **可扩展性强**：信号槽机制支持自定义信号和槽，适用于各种复杂的应用场景。

信号槽对比事件

	事件机制	信号槽机制
响应方式	通过重写事件处理函数（如 `event()`、`mousePressEvent()`），对象内部响应	通过信号（Signal）和槽（Slot）机制，对象间通信
适用场景	适合处理底层、面向对象自身的消息	适合对象间的灵活响应和协作
扩展性与灵活性	扩展性和灵活性有限，连接固定	支持多对多连接，动态连接和断开，灵活性高
代码维护性	代码分散、耦合度高，维护较难	发送者和接收者解耦，易于维护和扩展

信号槽基本用法

```cpp
#include <QCoreApplication>
#include <QObject>
#include <QDebug>
// 定义一个按钮类，继承自 QObject，支持信号槽
class Button : public QObject
{
    Q_OBJECT
public:
    // 模拟点击按钮的动作
    void click() {
        qDebug() << "按钮：正在被点击...";
        emit clicked();
    }
signals:
    void clicked();
};

```

```cpp
// 定义一个槽函数的接收者
class Receiver : public QObject
{
    Q_OBJECT
public slots:
    void onButtonClicked() {
        qDebug() << "按钮被点击了！";
    }
};

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    Button btn;
    Receiver receiver;
    // 信号与槽连接
    QObject::connect(&btn, &Button::clicked, &receiver,
    ↪  &Receiver::onButtonClicked);
    // 模拟用户点击
    btn.click();
    return app.exec();
}
```

信号槽连接方式（一）

```
#include <QWidget>
#include <QApplication>
#include <QPushButton>
#include <QLabel>
#include <QDebug>
#include <QVBoxLayout>
class SignalSlotExample : public QWidget
{
    Q_OBJECT
public:
    SignalSlotExample(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        setWindowTitle("Signal Slot Example");
        resize(300, 150);
    }
    void setupConnections() {
        m_button = new QPushButton("Test", this);
        m_label = new QLabel("Status", this);
        QVBoxLayout *layout = new QVBoxLayout(this);
        layout->addWidget(m_label);
        layout->addWidget(m_button);
        setLayout(layout);

        // 方法 1: 新语法（推荐）
        connect(m_button, &QPushButton::clicked,
                this,
                ↪ &SignalSlotExample::onButtonClicked);

        // 方法 2: Lambda 表达式
        connect(m_button, &QPushButton::clicked,
        ↪ [this]() {
            qDebug() << "Button clicked via lambda";
        });

        // 方法 3: 旧语法（不推荐）
        connect(m_button, SIGNAL(clicked()),
                this, SLOT(onButtonClicked()));

        // 方法 4: 带参数的连接
        connect(this,
        ↪ &SignalSlotExample::statusChanged,
                m_label, &QLabel::setText);
    }
```

信号槽连接方式（二）

```cpp
int main()
{
    Receiver* receiver1 = new Receiver();
    Receiver* receiver2 = new Receiver();
    Sender* sender1 = new Sender();
    Sender* sender2 = new Sender();
    // 多对一：多个信号连接同一个槽
    QObject::connect(sender1, &Sender::statusChanged,
    ↪  receiver1, &Receiver::onStatusChanged);
    QObject::connect(sender2, &Sender::statusChanged,
    ↪  receiver1, &Receiver::onStatusChanged);
    sender1->doSomething();
    sender2->doSomething();
    // 断开连接
    QObject::disconnect(sender1,
    ↪  &Sender::statusChanged, receiver1,
    ↪  &Receiver::onStatusChanged);
    QObject::disconnect(sender2,
    ↪  &Sender::statusChanged, receiver1,
    ↪  &Receiver::onStatusChanged);
    sender1->doSomething();
    sender2->doSomething();

    // 一对多：一个信号连接多个槽
    QObject::connect(sender1, &Sender::statusChanged,
    ↪  receiver1, &Receiver::onStatusChanged);
    QObject::connect(sender1, &Sender::statusChanged,
    ↪  receiver2, &Receiver::onStatusChanged);
    sender1->doSomething();
    // 断开全部
    QObject::disconnect(sender1,
    ↪  &Sender::statusChanged, nullptr, nullptr);
    sender1->doSomething();
    // 释放对象时，会自动断开信号和槽的连接
    delete receiver1;
    delete receiver2;
    delete sender1;
    delete sender2;
    std::cin.get();
    return 0;
```

目录

Qt 事件系统

事件类型

- 鼠标事件（Mouse Events）
- 键盘事件（Keyboard Events）
- 窗口事件（Window Events）
- 拖放事件（Drag & Drop Events）
- 定时器事件（Timer Events）
- 自定义事件（Custom Events）

事件处理方式

- 事件循环（Event Loop）
- 事件过滤器（Event Filters）
- 事件处理器（Event Handlers）

鼠标事件示例

```cpp
#include <QWidget>
#include <QMouseEvent>
#include <QDebug>
#include <QPainter>

class MouseEventWidget : public QWidget
{
    Q_OBJECT
public:
    MouseEventWidget(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        setMouseTracking(true); // 启用鼠标跟踪
    }

protected:
    void mousePressEvent(QMouseEvent *event) override {
        qDebug() << "鼠标按下，位置:" << event->pos();
        m_lastPos = event->pos();
        update();
    }
    void mouseMoveEvent(QMouseEvent *event) override {
        qDebug() << "鼠标移动，位置:" << event->pos();
        m_lastPos = event->pos();
        update();
    }

    void mouseReleaseEvent(QMouseEvent *event) override
    ↪ {
        qDebug() << "鼠标释放，位置:" << event->pos();
        m_lastPos = event->pos();
        update();
    }

    void paintEvent(QPaintEvent *) override {
        QPainter painter(this);
        painter.setPen(Qt::red);
        painter.drawEllipse(m_lastPos, 5, 5);
    }

private:
    QPoint m_lastPos;
};
```

键盘事件示例

```cpp
#include <QWidget>
#include <QKeyEvent>
#include <QDebug>
#include <QApplication>
#include <QSet>

class KeyboardEventWidget : public QWidget
{
    Q_OBJECT
public:
    KeyboardEventWidget(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        setFocusPolicy(Qt::StrongFocus); // 允许接收键盘
        ↪ 焦点
    }

protected:
    void keyPressEvent(QKeyEvent *event) override {
        m_pressedKeys.insert(event->key());
        // 组合键示例: Ctrl + S
        if (m_pressedKeys.contains(Qt::Key_Control) &&
        ↪ m_pressedKeys.contains(Qt::Key_S)) {
            qDebug() << "检测到组合键: Ctrl + S";
        }
        // 组合键示例: Shift + A
        else if (m_pressedKeys.contains(Qt::Key_Shift)
        ↪ && m_pressedKeys.contains(Qt::Key_A)) {
            qDebug() << "检测到组合键: Shift + A";
        }
        // 只按下单个键
        else {
            qDebug() << "按下键: " <<
            ↪ QKeySequence(event->key()).toString()
            ↪ << "(key code:" << event->key() << ")";
        }
        update();
    }
    void keyReleaseEvent(QKeyEvent *event) override {
        m_pressedKeys.remove(event->key());
        qDebug() << "释放键: " <<
        ↪ QKeySequence(event->key()).toString() <<
        ↪ "(key code:" << event->key() << ")";
    }
private:
    QSet<int> m_pressedKeys;
};
```

窗口事件示例

```
#include <QWidget>
#include <QEvent>
#include <QDebug>
#include <QApplication>
#include <QPainter>
#include <QResizeEvent>
#include <QMoveEvent>
#include <QPaintEvent>
#include <QShowEvent>
#include <QHideEvent>
#include <QCloseEvent>
class WindowEventWidget : public QWidget
{
    Q_OBJECT
public:
    WindowEventWidget(QWidget *parent = nullptr) :
    ↪  QWidget(parent) {}
protected:
    void showEvent(QShowEvent *event) override {
        qDebug() << "窗口显示事件";
        QWidget::showEvent(event);
    void hideEvent(QHideEvent *event) override {
        qDebug() << "窗口隐藏事件";
        QWidget::hideEvent(event);
    }

    void closeEvent(QCloseEvent *event) override {
        qDebug() << "窗口关闭事件";
        QWidget::closeEvent(event);
    }
    void resizeEvent(QResizeEvent *event) override {
        qDebug() << "窗口大小改变:" << event->size();
        QWidget::resizeEvent(event);
    }
    void moveEvent(QMoveEvent *event) override {
        qDebug() << "窗口移动到:" << event->pos();
        QWidget::moveEvent(event);
    }
    void paintEvent(QPaintEvent *event) override {
```

拖放事件示例

```cpp
#include <QWidget>
#include <QDragEnterEvent>
#include <QDropEvent>
#include <QMimeData>
#include <QDebug>
#include <QPainter>
#include <QApplication>

class DragDropWidget : public QWidget
{
    Q_OBJECT
public:
    DragDropWidget(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        setAcceptDrops(true); // 启用拖放
        m_text = "请拖拽文本到此窗口";
    }
protected:
    void dragEnterEvent(QDragEnterEvent *event)
    ↪ override {
        // 判断拖入的数据类型是否为文本
        if (event->mimeData()->hasText()) {
            event->acceptProposedAction();
            qDebug() << "拖拽进入，数据类型为文本";
        } else {
            event->ignore();
        }
    }
    void dropEvent(QDropEvent *event) override {
        if (event->mimeData()->hasText()) {
            m_text = event->mimeData()->text();
            qDebug() << "拖拽释放，接收到文本:" <<
            ↪ m_text;
            event->acceptProposedAction();
            update();
        } else {
            event->ignore();
        }
    }
    void paintEvent(QPaintEvent *) override {
        QPainter painter(this);
        painter.setPen(Qt::blue);
        painter.drawText(rect(), Qt::AlignCenter,
        ↪ m_text);
    }
private:
    QString m_text;
};
```

定时器事件示例

```
#include <QWidget>
#include <QTimerEvent>
#include <QDebug>
#include <QPainter>
#include <QApplication>
class TimerEventWidget : public QWidget
{
    Q_OBJECT
public:
    TimerEventWidget(QWidget *parent = nullptr) :
    ↪ QWidget(parent), m_counter(0) {
        // 启动定时器，间隔 1000 毫秒（1 秒）
        m_timerId = startTimer(1000);
    }
protected:
    void timerEvent(QTimerEvent *event) override {
        if (event->timerId() == m_timerId) {
            m_counter++;
            qDebug() << "定时器事件触发，计数:" <<
            ↪ m_counter;
            update();
        }
    }
    void paintEvent(QPaintEvent *) override {
        QPainter painter(this);
        painter.setPen(Qt::darkGreen);
        painter.drawText(rect(), Qt::AlignCenter,
        ↪ QString("定时器计数: %1").arg(m_counter));
    }
private:
    int m_timerId;
    int m_counter;
};

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TimerEventWidget w;
    w.resize(400, 200);
    w.show();
    return app.exec();
}
```

自定义事件示例

```cpp
#include <QWidget>
#include <QEvent>
#include <QDebug>
#include <QApplication>
#include <QPushButton>
// 定义自定义事件类型
const QEvent::Type MyCustomEventType = static_cast<QEvent::Type>(QEvent::User + 1);
class MyCustomEvent : public QEvent
{
public:
    MyCustomEvent(const QString& msg)
        : QEvent(MyCustomEventType), m_message(msg) {}

    QString message() const { return m_message; }
private:
    QString m_message;
};

class CustomEventWidget : public QWidget
{
    Q_OBJECT
public:
    CustomEventWidget(QWidget *parent = nullptr) : QWidget(parent)
    {
        QPushButton *btn = new QPushButton("发送自定义事件", this);
        btn->setGeometry(100, 80, 200, 40);
        connect(btn, &QPushButton::clicked, this, [this]() {
            // 发送自定义事件
            QCoreApplication::postEvent(this, new MyCustomEvent("Hello, 这是自定义事件!"));
        });
    }
protected:
    // 事件处理
    void customEvent(QEvent *event) override
    {
        if (event->type() == MyCustomEventType) {
            MyCustomEvent *myEvent = static_cast<MyCustomEvent*>(event);
            qDebug() << "收到自定义事件，消息内容:" << myEvent->message();
        }
    }
};
```

事件循环（Event Loop）

事件循环简介

- 事件循环是 Qt 应用程序的核心机制，负责持续接收、分发和处理事件与消息，保证界面响应和程序流畅运行。
- 事件循环通过 `QApplication::exec()` 启动，主线程进入循环，等待并处理各种事件（如鼠标、键盘、定时器、自定义事件等）。

事件循环实现方式

- 通过重写 `event(QEvent *event)` 方法，可以自定义事件的处理逻辑，实现对特定事件的拦截与响应。
- 事件循环自动调用事件处理函数，无需手动轮询，大大简化了事件驱动编程。

事件循环示例

```cpp
#include <QWidget>
#include <QEvent>
#include <QDebug>
#include <QApplication>
#include <QTimer>

// 事件循环演示控件
class EventLoopWidget : public QWidget
{
    Q_OBJECT
public:
    EventLoopWidget(QWidget *parent = nullptr) :
    ↪  QWidget(parent), m_counter(0)
    {
        // 启动定时器，每秒触发一次
        QTimer *timer = new QTimer(this);
        connect(timer, &QTimer::timeout, this,
        ↪  &EventLoopWidget::onTimeout);
        timer->start(1000);
    }

protected:
    // 重写事件处理函数，演示事件循环分发
    bool event(QEvent *event) override
    {
        if (event->type() == QEvent::Timer) {
            qDebug() << "收到定时器事件，计数:" <<
            ↪  m_counter;
        }
        return QWidget::event(event);
    }

private slots:
    void onTimeout()
    {
        m_counter++;
        qDebug() << "定时器超时，发送自定义事件，计数:" <<
        ↪  m_counter;
        // 这里可以发送自定义事件或做其他操作
    }

private:
    int m_counter;
};
```

事件过滤器（Event Filter）

事件过滤器简介

- Qt 事件过滤器允许一个对象监视和拦截另一个 QObject 对象的事件。。
- 常用于全局快捷键、特殊控件行为、日志记录等场景。
- 安装事件过滤器：`target->installEventFilter(filterObject);`
- 移除事件过滤器：`target->removeEventFilter(filterObject);`

事件过滤器实现

- 定义一个自定义类，继承自 `QObject`，并重写 `eventFilter(QObject *watched, QEvent *event)` 方法。
- 创建事件过滤器对象，并通过 `installEventFilter()` 安装到目标对象上。
- 在 `eventFilter` 方法中判断事件类型，进行自定义处理，返回 `true` 表示事件被拦截，`false` 表示继续传递。

事件过滤器示例

```cpp
#include <QWidget>
#include <QEvent>
#include <QKeyEvent>
#include <QDebug>
#include <QApplication>

// 自定义事件过滤器
class MyEventFilter : public QObject {
    Q_OBJECT
public:
    explicit MyEventFilter(QObject *parent = nullptr) :
    ↪ QObject(parent) {}
protected:
    bool eventFilter(QObject *, QEvent *event) override
    ↪ {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent =
            ↪ static_cast<QKeyEvent*>(event);
            qDebug() << "过滤器: 按键:" <<
            ↪ QKeySequence(keyEvent->key()).toString();
            if (keyEvent->key() == Qt::Key_Escape) {
                qDebug() << "过滤器: 拦截 Esc";
                return true;
            }
        }
        return false;
    }
};

// 事件过滤器控件
class EventFilterWidget : public QWidget {
    Q_OBJECT
public:
    EventFilterWidget(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        installEventFilter(new MyEventFilter(this));
        setFocusPolicy(Qt::StrongFocus);
    }
protected:
    void keyPressEvent(QKeyEvent *event) override {
        qDebug() << "收到按键: " <<
        ↪ QKeySequence(event->key()).toString();
    }
};
```

事件处理器（Event Handler）

事件处理器简介

- Qt 事件处理器允许一个对象处理另一个 QObject 对象的事件。
- 通过重写 `event(QEvent *event)` 方法，可以在事件到达目标对象前进行处理或拦截。
- 常用于全局快捷键、特殊控件行为、日志记录等场景。

事件处理器实现

- 定义一个自定义类，继承自 `QObject`，并重写 `event(QEvent *event)` 方法。

事件处理器示例

```cpp
#include <QWidget>
#include <QEvent>
#include <QKeyEvent>
#include <QDebug>
#include <QApplication>

// 事件处理器控件
class EventHandlerWidget : public QWidget
{
    Q_OBJECT
public:
    EventHandlerWidget(QWidget *parent = nullptr) :
    ↪ QWidget(parent) {
        setFocusPolicy(Qt::StrongFocus);
    }
protected:
    bool event(QEvent *event) override {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent =
            ↪ static_cast<QKeyEvent*>(event);
            qDebug() << "事件处理器: 按键:" <<
            ↪ QKeySequence(keyEvent->key()).toString();
            if (keyEvent->key() == Qt::Key_Escape) {
                qDebug() << "事件处理器: 拦截 Esc";
                return true;
            }
        }
        return QWidget::event(event);
    }
};
```

目录

Qt 容器特点

- 隐式共享（Implicit Sharing）：多个对象共享同一块内存，当其中一个对象修改时，会创建一个新的副本。
- 写时复制（Copy-on-Write）：当一个对象被修改时，会创建一个新的副本。
- 内存效率高: 容器类使用内存池管理内存，避免频繁的内存分配和释放。
- 线程安全: 容器类是线程安全的，可以在多线程环境下使用。
- 类型安全: 容器类是类型安全的，可以存储任意类型的数据，可以动态调整大小。
- std 兼容: 容器类是 std 兼容的，可以与 std 容器相互转换。
- 算法支持: 容器类支持算法，可以对容器中的数据进行排序、查找、统计等操作。

Qt 容器类对比

常用容器特性一览

容器类型	底层结构	主要特点
QList/QVector	动态数组	动态增删高效，随机访问快，适合频繁插入/删除
QMap	平衡二叉树	按键有序，查找/插入/删除效率高
QHash	哈希表	键无序，查找/插入/删除极快
QSet	哈希表/平衡树	唯一元素，查找/插入/删除高效
QStack	顺序容器封装	LIFO 结构，只访问栈顶
QQueue	顺序容器封装	FIFO 结构，只访问队首/队尾

QList 使用示例

```cpp
#include <QList>
#include <QString>
#include <QDebug>
#include <algorithm>
#include <iostream>

void listExample()
{
    // 创建 QList 并添加元素
    QList<int> numbers;
    numbers << 10 << 20 << 30;
    numbers.append(40);
    numbers.prepend(5);
    qDebug() << "QList<int> 内容:";
    for (int i = 0; i < numbers.size(); ++i) {
        qDebug() << "索引" << i << ":" <<
        ↪ numbers.at(i);
    }
    // 插入和移除元素
    numbers.insert(2, 15); // 在索引 2 插入 15
    numbers.removeAt(0);   // 移除第 0 个元素
    qDebug() << "修改后的 QList<int> 内容:";
    for (int n : numbers) {
        qDebug() << n;
    }
    // 查找元素
    int idx = numbers.indexOf(30);
    if (idx != -1) {
        qDebug() << "30 的索引为:" << idx;
    }
    // 反转和排序
    std::sort(numbers.begin(), numbers.end());
    std::reverse(numbers.begin(), numbers.end());
    qDebug() << "排序并反转后的 QList<int> 内容:" <<
    ↪ numbers;
    // 支持自定义类型
    QList<QString> names;
    names << "Alice" << "Bob" << "Charlie";
    qDebug() << "QList<QString> 内容:" << names;
}
```

QMap 使用示例

```cpp
#include <QMap>
#include <QString>
#include <QDebug>
#include <iostream>

void mapExample()
{
    // 创建 QMap 并插入元素
    QMap<QString, int> scores;
    scores.insert("Alice", 90);
    scores.insert("Bob", 85);
    scores["Charlie"] = 92;
    scores["David"] = 88;
    qDebug() << "QMap 内容:";
    // 遍历 QMap（按 key 有序）
    QMap<QString, int>::const_iterator it;
    for (it = scores.constBegin(); it != scores.constEnd(); ++it) {
        qDebug() << it.key() << ":" << it.value();
    }
    // 查找元素
    if (scores.contains("Bob")) {
        qDebug() << "Bob 的分数:" << scores.value("Bob");
    }
    // 修改元素
    scores["Alice"] = 95;
    qDebug() << "修改后 Alice 的分数:" << scores.value("Alice");
    // 移除元素
    scores.remove("David");
    qDebug() << "移除 David 后 QMap 内容:" << scores;
    // 获取所有 key 和 value
    QList<QString> keys = scores.keys();
    QList<int> values = scores.values();
    qDebug() << "所有学生:" << keys;
    qDebug() << "所有分数:" << values;
    // 清空 QMap
    scores.clear();
    qDebug() << "清空后 QMap 是否为空:" << scores.isEmpty();
}
```

QHash 使用示例

```cpp
#include <QHash>
#include <QString>
#include <QDebug>
#include <iostream>

void hashExample()
{
    // 创建 QHash 并插入元素
    QHash<QString, int> ages;
    ages.insert("Alice", 25);
    ages.insert("Bob", 30);
    ages["Charlie"] = 28;
    ages["David"] = 22;
    qDebug() << "QHash 内容:";
    // 遍历 QHash（无序）
    QHash<QString, int>::const_iterator it;
    for (it = ages.constBegin(); it != ages.constEnd();
    ↪  ++it) {
        qDebug() << it.key() << ":" << it.value();
    }
    // 查找元素
    if (ages.contains("Bob")) {
        qDebug() << "Bob 的年龄:" << ages.value("Bob");
    }
    // 修改元素
    ages["Alice"] = 26;
    qDebug() << "修改后 Alice 的年龄:" <<
    ↪  ages.value("Alice");
    // 移除元素
    ages.remove("David");
    qDebug() << "移除 David 后 QHash 内容:" << ages;
    // 获取所有 key 和 value
    QList<QString> keys = ages.keys();
    QList<int> values = ages.values();
    qDebug() << "所有人名:" << keys;
    qDebug() << "所有年龄:" << values;
    // 清空 QHash
    ages.clear();
    qDebug() << "清空后 QHash 是否为空:" <<
    ↪  ages.isEmpty();
}
```

QSet 使用示例

```cpp
#include <QSet>
#include <QString>
#include <QDebug>
#include <iostream>
void setExample()
{
    // 创建 QSet 并添加元素
    QSet<QString> fruits;
    fruits << "Apple" << "Banana" << "Orange";
    fruits.insert("Grape");
    fruits.insert("Banana"); // 重复元素不会被添加
    qDebug() << "QSet 内容:";
    for (const QString &fruit : fruits) {
        qDebug() << fruit;
    }
    // 判断元素是否存在
    if (fruits.contains("Apple")) {
        qDebug() << "集合中包含 Apple";
    }
    }
    // 移除元素
    fruits.remove("Orange");
    qDebug() << "移除 Orange 后 QSet 内容:" << fruits;
    // 集合操作：并集、交集、差集
    QSet<QString> tropicalFruits;
    tropicalFruits << "Banana" << "Mango" <<
    ↪  "Pineapple";
    QSet<QString> unionSet =
    ↪  fruits.unite(tropicalFruits);      // 并集
    QSet<QString> intersectSet =
    ↪  fruits.intersect(tropicalFruits); // 交集
    QSet<QString> diffSet =
    ↪  fruits.subtract(tropicalFruits);     // 差集
    qDebug() << "并集:" << unionSet;
    qDebug() << "交集:" << intersectSet;
    qDebug() << "差集:" << diffSet;
    // 清空集合
    fruits.clear();
    qDebug() << "清空后 QSet 是否为空:" <<
    ↪  fruits.isEmpty();
}
```

QStack 使用示例

```cpp
#include <QStack>
#include <QString>
#include <QDebug>
#include <iostream>

void stackExample()
{
    // 创建 QStack 并压入元素
    QStack<QString> stack;
    stack.push("第一步");
    stack.push("第二步");
    stack.push("第三步");
    qDebug() << "QStack 内容（从栈底到栈顶）:";
    for (int i = 0; i < stack.size(); ++i) {
        qDebug() << stack.at(i);
    }

    // 查看栈顶元素
    if (!stack.isEmpty()) {
        qDebug() << "栈顶元素:" << stack.top();
    }

    // 弹出元素
    QString popped = stack.pop();
    qDebug() << "弹出元素:" << popped;
    qDebug() << "弹出后栈顶:" << (stack.isEmpty() ? "空"
    ↪  : stack.top());

    // 判断是否为空
    qDebug() << "QStack 是否为空:" << stack.isEmpty();

    // 清空栈
    stack.clear();
    qDebug() << "清空后 QStack 是否为空:" <<
    ↪  stack.isEmpty();
}
```

QQueue 使用示例

```cpp
#include <QQueue>
#include <QString>
#include <QDebug>
#include <iostream>
void queueExample()
{
    // 创建 QQueue 并入队元素
    QQueue<QString> queue;
    queue.enqueue("任务 1");
    queue.enqueue("任务 2");
    queue.enqueue("任务 3");
    qDebug() << "QQueue 内容（从队首到队尾）:";
    for (int i = 0; i < queue.size(); ++i) {
        qDebug() << queue.at(i);
    }

    // 查看队首和队尾元素
    if (!queue.isEmpty()) {
        qDebug() << "队首元素:" << queue.head();
        qDebug() << "队尾元素:" << queue.back();
    }

    // 出队元素
    QString front = queue.dequeue();
    qDebug() << "出队元素:" << front;
    qDebug() << "出队后队首:" << (queue.isEmpty() ? "空"
    ↪  : queue.head());

    // 判断是否为空
    qDebug() << "QQueue 是否为空:" << queue.isEmpty();

    // 清空队列
    queue.clear();
    qDebug() << "清空后 QQueue 是否为空:" <<
    ↪  queue.isEmpty();
}
```

目录

QString 简介

- **QString** 是 Qt 中用于处理 Unicode 字符串的核心类，支持多语言环境下的高效字符串处理。
- 提供丰富的字符串操作：高效的拼接、查找、替换、分割、格式化、截取、大小写转换、去除空白、插入、移除、重复等。
- 支持与标准 C++ 字符串类型（如 `std::string`、`char*`）的相互转换，便于与 C++ 标准库协作。
- 可以与 Qt 的其他类（如 `QTextStream`、`QFile`、`QByteArray` 等）无缝集成，广泛应用于界面、文件 IO、网络通信和数据处理等场景。
- 内置对 Unicode 的支持，适合国际化应用开发，能够正确处理多种语言和字符集。
- 提供灵活的格式化方法（如 `arg()`、静态格式化等），便于生成动态文本。
- 支持与 `QStringList` 等容器类协作，方便字符串的批量处理和转换。

QString 使用示例（一）

```cpp
#include <QString>
#include <QStringList>
#include <QByteArray>
#include <QDebug>
#include <iostream>
void stringExample()
{
    // 1. 创建和初始化 QString
    QString str1 = QStringLiteral("Hello");
    QString str2 = QStringLiteral("世界");
    QString str3 = QString::fromUtf8("Qt 字符串示例");
    // 2. 拼接字符串
    QString str4 = QString("%1, %2!").arg(str1, str2);
    // 推荐使用 arg 格式化
    str4.append(QStringLiteral(" 欢迎使用 Qt."));
    qDebug() << "拼接后的字符串:" << str4;
    // 3. 格式化字符串
    int year = 2024;
    double pi = 3.14159;
    QString str5 = QStringLiteral("今年是%1, 圆周率约为%2").arg(year).arg(pi, 0, 'f', 2);
    qDebug() << "格式化字符串:" << str5;
    // 4. 查找和替换
    int pos = str4.indexOf(QStringLiteral("Qt"));
    if (pos != -1) {
        qDebug() << R"("Qt" 在 str4 中的位置:)" << pos;
    }
    QString str6 = str4;
    str6.replace(QStringLiteral("Qt"), QStringLiteral("C++"));
    qDebug() << "替换后的字符串:" << str6;
    // 5. 分割和连接
    QString fruits = QStringLiteral("Apple,Banana,Orange");
    QStringList fruitList = fruits.split(',');
    qDebug() << "分割后的 QStringList:" << fruitList;
    QString joined = fruitList.join(QStringLiteral(" | "));
    qDebug() << "用 | 连接后的字符串:" << joined;
```

QString 使用示例（二）

```
37      // 6. 大小写转换
38      qDebug() << "大写:" << str1.toUpper();
39      qDebug() << "小写:" << str2.toLower();
40      // 7. 去除空白
41      QString str7 = QStringLiteral("   Qt is fun!   ");
42      qDebug() << "去除前后空白:" << str7.trimmed();
43      // 8. 转换为标准 C++ 字符串
44      std::string stdStr = str1.toStdString();
45      qDebug() << "转换为 std::string:" <<
        ↪  QString::fromStdString(stdStr);
46      // 9. 转换为数字
47      QString numStr = QStringLiteral("12345");
48      int num = numStr.toInt();
49      qDebug() << "QString 转 int:" << num;
50      // 10. 判断是否为空
51      QString emptyStr;
52      qDebug() << "emptyStr 是否为空:" <<
        ↪  emptyStr.isEmpty();
53      // 11. 字符串长度
54      qDebug() << "str4 的长度:" << str4.length();
55      qDebug() << "str4 的大小:" << str4.size();
```

```
57      // 12. 字符串截取
58      QString str8 = QStringLiteral("Hello, Qt World!");
59      qDebug() << "mid 截取:" << str8.mid(7, 2);   // 从索
        ↪  引 7 开始，取 2 个字符
60      qDebug() << "left 截取:" << str8.left(5);     // 左
        ↪  边 5 个字符
61      qDebug() << "right 截取:" << str8.right(6);   // 右
        ↪  边 6 个字符
62      // 13. 判断字符串内容
63      qDebug() << "str8 是否包含'Qt':" <<
        ↪  str8.contains(QStringLiteral("Qt"));
64      qDebug() << "str8 是否以'Hello'开头:" <<
        ↪  str8.startsWith(QStringLiteral("Hello"));
65      qDebug() << "str8 是否以'World!'结尾:" <<
        ↪  str8.endsWith(QStringLiteral("World!"));
66      // 14. 字符串比较
67      QString cmp1 = QStringLiteral("abc");
68      QString cmp2 = QStringLiteral("Abc");
69      qDebug() << "区分大小写比较:" << (cmp1 == cmp2);
70      qDebug() << "不区分大小写比较:" <<
        ↪  (cmp1.compare(cmp2, Qt::CaseInsensitive) == 0);
```

QString 使用示例（三）

```
72      // 15. 字符串插入和移除
73      QString str9 = QStringLiteral("I Qt!");
74      str9.insert(2, QStringLiteral("love "));
75      qDebug() << "插入后的字符串:" << str9;
76      str9.remove(2, 5); // 从索引 2 移除 5 个字符
77      qDebug() << "移除后的字符串:" << str9;
78      // 16. 重复字符串
79      QString repeated =
        ↪  QStringLiteral("Ha").repeated(3);
80      qDebug() << "重复字符串:" << repeated;
81      // 17. 转换为数字失败的处理
82      QString invalidNum = QStringLiteral("abc");
83      bool ok = false;
84      int invalid = invalidNum.toInt(&ok);
85      qDebug() << "转换非法数字字符串, ok:" << ok << ",
        ↪  value:" << invalid;
```

```
87      // 18. 字符串与 QByteArray 互转
88      QString unicodeStr = QStringLiteral("你好, Qt!");
89      QByteArray ba = unicodeStr.toUtf8();
90      qDebug() << "转为 QByteArray:" << ba;
91      QString fromBa = QString::fromUtf8(ba);
92      qDebug() << "QByteArray 转回 QString:" << fromBa;
93      // 19. 字符串格式化（静态方法）
94      QString formatted = QStringLiteral("%1 + %2 =
        ↪  %3").arg(2).arg(3).arg(2+3);
95      qDebug() << "静态格式化:" << formatted;
96      // 20. 遍历字符串中的字符
97      QString str10 = QStringLiteral("Qt");
98      for (const QChar &ch : str10) {
99          qDebug() << "字符:" << ch;
100     }
101 }
```

QStringList 使用示例（一）

```cpp
#include <QStringList>
#include <QString>
#include <QDebug>
#include <iostream>
void stringListExample()
{
    // 1. 创建和初始化 QStringList
    QStringList list1;
    list1 << "Apple" << "Banana" << "Orange";
    qDebug() << "初始化的 QStringList:" << list1;
    // 2. 通过字符串分割生成 QStringList
    QString fruits =
    ↪  QStringLiteral("Mango,Peach,Pear");
    QStringList list2 = fruits.split(',');
    qDebug() << "分割字符串得到的 QStringList:" <<
    ↪  list2;
    // 3. 拼接 QStringList 为字符串
    QString joined = list1.join(" | ");
    qDebug() << "用 | 连接的字符串:" << joined;
    // 4. 添加和插入元素
    list1.append("Grape");
    list1.prepend("Pineapple");
    list1.insert(2, "Lemon");
    qDebug() << "添加和插入后的 QStringList:" << list1;
    // 5. 查找和移除元素
    int idx = list1.indexOf("Banana");
    if (idx != -1) {
        qDebug() << "\"Banana\"的位置:" << idx;
        list1.removeAt(idx);
    }
    list1.removeAll("Orange");
    qDebug() << "移除元素后的 QStringList:" << list1;
```

QStringList 使用示例（二）

```
31      // 6. 判断是否包含某元素
32      qDebug() << "是否包含\"Apple\":" <<
        ↪  list1.contains("Apple");
33      // 7. 遍历 QStringList
34      qDebug() << "遍历 QStringList:";
35      for (const QString &fruit : list1) {
36          qDebug() << fruit;
37      }
38      // 8. 过滤和排序
39      QStringList filtered = list1.filter("a",
        ↪  Qt::CaseInsensitive);
40      qDebug() << "包含字母 a 的元素:" << filtered;
41      list1.sort();
42      qDebug() << "排序后的 QStringList:" << list1;
```

```
43      // 9. 转换为 QList
44      QList<QString> qlist = list1;
45      qDebug() << "转换为 QList 后第一个元素:" <<
        ↪  qlist.first();
46      // 10. 转换为标准 C++ 容器
47      std::vector<QString> stdVec =
        ↪  std::vector<QString>(list1.begin(),
        ↪  list1.end());
48      if (!stdVec.empty()) {
49          qDebug() << "转换为 std::vector 后第一个元素:"
            ↪  << stdVec.front();
50      }
51      // 11. 清空和判断空
52      list1.clear();
53      qDebug() << "清空后是否为空:" << list1.isEmpty();
54  }
```

QByteArray 简介

- **QByteArray** 是 Qt 中专门用于高效处理原始二进制数据和字节序列的类，底层采用引用计数和写时复制（copy-on-write）机制，内存管理高效。
- 支持与 C++ 标准库的 `std::string`、`char*`、`std::vector<char>` 等类型的无缝互转，便于与标准 C++ 代码协作。
- 可与 Qt 的 `QTextStream`、`QFile`、`QIODevice`、`QDataStream` 等类直接配合，广泛应用于文件读写、网络通信、数据序列化、加密解密等场景。
- 支持多种编码（如 Base64、十六进制、URL 编码等）与解码操作，便于数据的存储与传输。
- 提供丰富的成员函数：如 `append()`、`insert()`、`remove()`、`replace()`、`split()`、`toInt()`、`toHex()`、`fromHex()`、`toBase64()`、`fromBase64()` 等，满足各种二进制和文本数据处理需求。
- 可以与 `QString` 互相转换，便于文本与二进制数据的混合处理。
- 支持直接操作底层数据指针（`data()`、`constData()`），适合高性能场景。

QByteArray 使用示例（一）

```cpp
#include <QByteArray>
#include <QList>
#include <QDebug>
#include <vector>
#include <string>
#include <iostream>
void byteArrayExample()
{
    // 1. 多种方式创建 QByteArray
    QByteArray ba1("Hello, Qt!"); // C 字符串初始化
    QByteArray ba2 =
    ↪  QByteArray::fromStdString(std::string("QByteArray
    ↪  示例")); // std::string 初始化
    QByteArray ba3(10, 'A'); // 10 个'A'
    QByteArray ba4; // 空 QByteArray
    qDebug() << "ba1 原始内容:" << ba1;
    qDebug() << "ba2 原始内容:" << ba2;
    qDebug() << "ba3 原始内容:" << ba3;
    qDebug() << "ba4 是否为空:" << ba4.isEmpty();
    // 2. 追加、插入、前置、替换、移除
    ba1.append(" Welcome.");
    ba1.prepend(">>> ");
    ba1.insert(7, "C++ ");
    qDebug() << "追加、前置、插入后:" << ba1;
    ba1.replace("C++", "Qt/C++");
    ba1.remove(0, 4);
    qDebug() << "替换和移除后:" << ba1;
    // 3. 查找、判断、访问
    int pos = ba1.indexOf("Qt");
    if (pos != -1) {
        qDebug() << "“Qt”在 ba1 中的位置:" << pos;
    }
    qDebug() << "ba1 是否包含 Welcome:" <<
    ↪  ba1.contains("Welcome");
    qDebug() << "ba1 的长度:" << ba1.length();
    if (!ba1.isEmpty())
        qDebug() << "ba1 的第一个字符:" << ba1.at(0);
```

QByteArray 使用示例（二）

```
35      // 4. 分割、拼接、遍历
36      QByteArray ba5("apple,banana,orange,pear");
37      QList<QByteArray> list = ba5.split(',');
38      qDebug() << "分割结果:";
39      int idx = 0;
40      for (const QByteArray &item : list) {
41          qDebug() << QString("第%1 个元素:
            ↪  %2").arg(idx++).arg(QString::fromUtf8(item));
42      }
43      QByteArray joined = list.join(" | ");
44      qDebug() << "用 | 拼接结果:" << joined;
45      // 5. 转换为数字、十六进制、Base64、URL 编码
46      QByteArray ba6("12345");
47      int num = ba6.toInt();
48      double dnum = QByteArray("3.14159").toDouble();
49      qDebug() << "字符串转整数:" << num;
50      qDebug() << "字符串转浮点数:" << dnum;
51      QByteArray hex = ba6.toHex();
52      qDebug() << "十六进制编码:" << hex;
```

```
53      QByteArray base64 = ba6.toBase64();
54      qDebug() << "Base64 编码:" << base64;
55      QByteArray urlEncoded = ba6.toPercentEncoding();
56      qDebug() << "URL 编码:" << urlEncoded;
57      // 6. 解码
58      QByteArray fromHex = QByteArray::fromHex(hex);
59      QByteArray fromBase64 =
        ↪  QByteArray::fromBase64(base64);
60      QByteArray fromUrl =
        ↪  QByteArray::fromPercentEncoding(urlEncoded);
61      qDebug() << "十六进制解码:" << fromHex;
62      qDebug() << "Base64 解码:" << fromBase64;
63      qDebug() << "URL 解码:" << fromUrl;
64      // 7. 与 QString 互转
65      QString str1 = QString::fromUtf8(ba1);
66      QByteArray ba7 = str1.toUtf8();
67      qDebug() << "QByteArray 转 QString:" << str1;
68      qDebug() << "QString 转 QByteArray:" << ba7;
```

QByteArray 使用示例（三）

```
69      // 8. 直接访问底层数据、修改内容
70      const char *raw = ba1.constData();
71      char *data = ba1.data();
72      qDebug() << "底层数据 (只读):" << raw;
73      if (ba1.size() > 0) {
74          data[0] = 'q';
75          qDebug() << "修改后 ba1:" << ba1;
76      }
77      // 9. 截取、清空、判断空
78      QByteArray sub = ba1.mid(2, 5);
79      qDebug() << "mid 截取:" << sub;
80      QByteArray left = ba1.left(4);
81      QByteArray right = ba1.right(6);
82      qDebug() << "left 截取:" << left;
83      qDebug() << "right 截取:" << right;
84      ba1.clear();
85      qDebug() << "清空后 ba1 是否为空:" << ba1.isEmpty();
```

```
86      // 10. QByteArray 与 std::string、std::vector<char>
        ↪  互转
87      std::string stdStr = ba6.toStdString();
88      QByteArray ba8 = QByteArray::fromStdString(stdStr);
89      qDebug() << "QByteArray 转 std::string:" <<
        ↪  QString::fromStdString(stdStr);
90      qDebug() << "std::string 转 QByteArray:" << ba8;
91      std::vector<char> vec(ba5.begin(), ba5.end());
92      QByteArray ba9(vec.data(),
        ↪  static_cast<int>(vec.size()));
93      qDebug() << "QByteArray 转 std::vector<char>:" <<
        ↪  QByteArray(vec.data(),
        ↪  static_cast<int>(vec.size()));
94      qDebug() << "std::vector<char> 转 QByteArray:" <<
        ↪  ba9;
95  }
```

QVariant 简介

- **QVariant** 是 Qt 中用于存储和管理任意类型数据的通用容器类，底层实现了类型安全的类型擦除（type erasure）。
- 可以存储 Qt 内置的各种基本类型（如 `int`、`double`、`bool`、`QString`、`QDateTime`、`QByteArray` 等）、Qt 容器类（如 `QStringList`、`QList<int>` 等），也支持注册自定义类型。
- 常用于通用接口、模型/视图框架（如 `QAbstractItemModel`）、动态属性、信号槽参数等场景，实现类型无关的数据传递和存储。
- 提供类型判断（`type()`、`typeName()`）、类型转换（`toInt()`、`toString()`、`toList()` 等）、类型安全访问（`value<T>()`）、类型设置（`setValue()`）等丰富接口。
- 支持与 C++ 标准类型的互转，便于与标准库协作。
- 通过 `QVariant::isNull()` 和 `QVariant::isValid()` 判断内容是否为空或有效。
- 适合需要存储”任意类型”或”类型不确定”数据的场景，是 Qt 元对象系统和动态特性的基础之一。

QVariant 使用示例（一）

```cpp
#include <QVariant>
#include <QString>
#include <QStringList>
#include <QDateTime>
#include <QByteArray>
#include <QList>
#include <QMap>
#include <QDebug>
#include <iostream>
#if (QT_VERSION >= QT_VERSION_CHECK(5, 7, 0))
#include <QJsonObject>
#endif
// 自定义类型声明与注册（建议放在全局，避免多次注册）
struct MyType {
    int x;
};
```

```cpp

void variantExample()
{
    // 1. 存储基本类型
    QVariant vInt = 42;
    QVariant vDouble = 3.14159;
    QVariant vBool = true;
    QVariant vString = QStringLiteral("Hello
    ↪  QVariant");
    QVariant vDateTime = QDateTime::currentDateTime();
    qDebug() << "int:" << vInt.toInt()
             << "double:" << vDouble.toDouble()
             << "bool:" << vBool.toBool()
             << "QString:" << vString.toString()
```

QVariant 使用示例（二）

```
            << "QDateTime:" <<
            ↪ vDateTime.toDateTime().toString(Qt::ISODate);
    // 2. 存储 Qt 容器类型
    QStringList strList = {"Apple", "Banana",
    ↪ "Orange"};
    QVariant vStrList = strList;
    qDebug() << "QStringList:" <<
    ↪ vStrList.toStringList();
    QList<int> intList = {1, 2, 3, 4, 5};
    QVariant vIntList = QVariant::fromValue(intList);
    qDebug() << "QList<int>:" <<
    ↪ vIntList.value<QList<int>>();
    // 3. 存储 QByteArray
    QByteArray ba("Qt Variant");
    QVariant vByteArray = ba;
    qDebug() << "QByteArray:" <<
    ↪ vByteArray.toByteArray();
    // 4. 类型判断与转换
    if (vInt.typeId() == QMetaType::Int)
        qDebug() << "vInt 是 int 类型";
    if (vString.canConvert<QString>())
        qDebug() << "vString 可以转换为 QString:" <<
        ↪ vString.toString();
    // 5. QVariant 的空值和有效性判断
    QVariant vNull;
    qDebug() << "vNull 是否有效:" << vNull.isValid()
             << "vNull 是否为 null:" << vNull.isNull();
    // 6. QVariant 的类型安全访问
    vInt.setValue(2024);
    qDebug() << "类型安全访问 int:" <<
    ↪ vInt.value<int>();
    // 7. QVariant 的拷贝与比较
    QVariant vCopy = vString;
    qDebug() << "vCopy == vString:" << (vCopy ==
    ↪ vString);
    vCopy = 100;
    qDebug() << "vCopy 赋新值后:" << vCopy;
    // 8. QVariant 的类型名与类型 ID
    qDebug() << "vInt 类型名:" << vInt.typeName()
```

QVariant 使用示例（三）

```
62              << "vString 类型 ID:" << vString.typeId();
63      // 9. QVariant 与 QMap、QList 嵌套使用
64      QMap<QString, QVariant> map{
65          {"name", "Tom"},
66          {"age", 18},
67          {"score", 95.5}
68      };
69      qDebug() << "QMap<QString, QVariant> 内容:";
70      for (auto it = map.cbegin(); it != map.cend();
        ↪  ++it)
71          qDebug() << it.key() << ":" << it.value();
72      QList<QVariant> varList = {1, 2.5, "abc",
        ↪  QDateTime::currentDateTime()};
73      qDebug() << "QList<QVariant> 内容:";
74      for (const QVariant &v : varList)
75          qDebug() << v;
76      // 10. QVariant 的 toJsonValue、toJsonObject、
        ↪  toJsonArray（Qt 5.7+）
77  #if (QT_VERSION >= QT_VERSION_CHECK(5, 7, 0))
78      QVariantMap jsonMap{{"user", "Alice"}, {"id",
        ↪  123}};
79      QJsonObject obj =
        ↪  QJsonObject::fromVariantMap(jsonMap);
```

```
80      qDebug() << "QVariantMap 转 QJsonObject:" << obj;
81  #endif
82      // 11. QVariant 的 swap
83      QVariant a = 1, b = 2;
84      a.swap(b);
85      qDebug() << "swap 后 a:" << a << "b:" << b;
86      // 12. QVariant 的 clear
87      a.clear();
88      qDebug() << "clear 后 a 是否有效:" << a.isValid();
89      // 13. QVariant 的 toStringList、toList、toMap 等
90      QVariant vList =
        ↪  QVariant::fromValue(QList<QVariant>{1, 2, 3});
91      qDebug() << "QVariant 转 QList<QVariant>:" <<
        ↪  vList.toList();
92      QVariant vMap =
        ↪  QVariant::fromValue(QVariantMap{{"k1", 1},
        ↪  {"k2", 2}});
93      qDebug() << "QVariant 转 QVariantMap:" <<
        ↪  vMap.toMap();
```

QVariant 使用示例（四）

```
    // 14. QVariant 的 isNull/isValid 区别演示
    QVariant vEmptyString = QString();
    qDebug() << "空 QString 的 QVariant isNull:" <<
    ↪  vEmptyString.isNull()
             << "isValid:" << vEmptyString.isValid();
    // 15. QVariant 的 setValue 与模板 value<T>() 的配合
    QVariant vAny;
    vAny.setValue(QString("AnyType"));
    qDebug() << "setValue 后 value<QString>:" <<
    ↪  vAny.value<QString>();
    // 16. QVariant 与自定义类型（注册自定义类型）
    // 注册类型（通常在 main 函数或程序初始化时注册一次）
    static bool registered = [](){
        qRegisterMetaType<MyType>("MyType");
        return true;
    }();
    MyType myObj{10};
    QVariant vCustom = QVariant::fromValue(myObj);
    if (vCustom.canConvert<MyType>()) {
        MyType outObj = vCustom.value<MyType>();
        qDebug() << "自定义类型 MyType.x:" << outObj.x;
    }
```

```
#endif
// 自定义类型声明与注册（建议放在全局，避免多次注册）
struct MyType {
    int x;
};
```

QFile 简介

- **QFile** 是 Qt 中用于文件操作的核心类，继承自 `QIODevice`，可用于对本地文件进行读写、创建、删除、重命名、拷贝等操作。
- 支持文本和二进制文件的读写，能够以只读、只写、读写、追加等多种模式打开文件。
- 常用方法包括 `open()`（打开文件）、`close()`（关闭文件）、`read()`、`write()`、`exists()`、`remove()`、`rename()`、`copy()` 等。
- 支持与 `QTextStream`、`QDataStream` 等流类配合，实现高效的文本和二进制数据读写。
- 可通过 `QFileInfo` 获取文件的详细信息（如大小、创建时间、权限等）。
- 跨平台，自动处理不同操作系统下的文件路径和编码问题。
- 适用于配置文件、日志文件、数据文件等各种文件操作场景。

QFile 使用示例（一）

```cpp
#include <QFile>
#include <QTextStream>
#include <QDebug>
#include <QFileInfo>
#include <QDateTime>
#include <iostream>
void fileExample()
{
    // 1. 创建 QFile 对象，指定文件名
    QString fileName = "example.txt";
    QFile file(fileName);
    // 2. 以写入模式打开文件（会覆盖原有内容），写入多行内容
    if (file.open(QIODevice::WriteOnly |
    ↪  QIODevice::Text)) {
        QTextStream out(&file);
        out << "Hello, QFile!\n";
        out << "Qt 文件操作示例。\n";
        out << "当前时间: " <<
        ↪  QDateTime::currentDateTime().toString(Qt::ISODate)
        ↪  << "\n";
        out << "支持中文写入。\n";
        file.close();
        qDebug() << "写入文件成功";
    } else {
        qDebug() << "无法打开文件进行写入";
        return;
    }
    // 3. 以追加模式打开文件，追加内容
    if (file.open(QIODevice::Append | QIODevice::Text))
    ↪  {
        QTextStream out(&file);
        out << "追加一行内容。\n";
        file.close();
        qDebug() << "追加内容成功";
    } else {
        qDebug() << "无法打开文件进行追加";
    }
```

QFile 使用示例（二）

```
34      // 4. 以只读模式打开文件，读取全部内容
35      if (file.open(QIODevice::ReadOnly |
        ↪ QIODevice::Text)) {
36          QTextStream in(&file);
37          qDebug() << "文件内容: ";
38          int lineNum = 1;
39          while (!in.atEnd()) {
40              QString line = in.readLine();
41              qDebug() << QString("第%1 行:
                ↪ %2").arg(lineNum++).arg(line);
42          }
43          file.close();
44      } else {
45          qDebug() << "无法打开文件进行读取";
46      }
47      // 5. 检查文件是否存在，并获取文件信息
48      if (QFile::exists(fileName)) {
49          QFileInfo info(fileName);
50          qDebug() << fileName << "文件存在";
51          qDebug() << "文件大小:" << info.size() << "字
            ↪ 节";
52          qDebug() << "创建时间:" <<
            ↪ info.birthTime().toString(Qt::ISODate);
53          qDebug() << "最后修改时间:" <<
            ↪ info.lastModified().toString(Qt::ISODate);
54          qDebug() << "绝对路径:" <<
            ↪ info.absoluteFilePath();
55          qDebug() << "是否可读:" << info.isReadable() <<
            ↪ "是否可写:" << info.isWritable();
56      } else {
57          qDebug() << fileName << "文件不存在";
58          return;
59      }
60      // 6. 重命名文件
61      QString renamedName = "renamed.txt";
62      if (QFile::rename(fileName, renamedName)) {
63          qDebug() << "文件重命名成功:" << renamedName;
64      } else {
65          qDebug() << "文件重命名失败";
66          return;
67      }
```

QFile 使用示例（三）

```
68      // 7. 拷贝文件
69      QString copyName = "copy.txt";
70      if (QFile::copy(renamedName, copyName)) {
71          qDebug() << "文件拷贝成功:" << copyName;
72      } else {
73          qDebug() << "文件拷贝失败";
74      }
75      // 8. 读取拷贝文件的全部内容为字符串
76      QFile copyFile(copyName);
77      if (copyFile.open(QIODevice::ReadOnly |
    ↪  QIODevice::Text)) {
78          QString allText = copyFile.readAll();
79          qDebug() << "copy.txt 全部内容为: \n" <<
        ↪  allText;
80          copyFile.close();
81      }
```

```
82      // 9. 删除文件
83      if (QFile::remove(renamedName)) {
84          qDebug() << renamedName << "已删除";
85      } else {
86          qDebug() << "删除" << renamedName << "失败";
87      }
88      if (QFile::remove(copyName)) {
89          qDebug() << copyName << "已删除";
90      } else {
91          qDebug() << "删除" << copyName << "失败";
92      }
93  }
```

QDir 使用示例（一）

```cpp
#include <QDir>
#include <QDebug>
#include <QStringList>
#include <iostream>
void dirExample()
{
    // 1. 获取当前工作目录
    QDir currentDir = QDir::current();
    qDebug() << "当前工作目录:" <<
    ↪  currentDir.absolutePath();
    // 2. 创建新目录
    QString newDirName = "test_dir";
    if (currentDir.mkdir(newDirName)) {
        qDebug() << "创建目录成功:" << newDirName;
    } else {
        qDebug() << "创建目录失败或已存在:" <<
        ↪  newDirName;
    }
    // 3. 进入新目录
    if (currentDir.cd(newDirName)) {
        qDebug() << "进入目录:" <<
        ↪  currentDir.absolutePath();
    } else {
        qDebug() << "进入目录失败:" << newDirName;
    }
    // 4. 创建多个文件和子目录
    QFile file1(currentDir.filePath("file1.txt"));
    if (file1.open(QIODevice::WriteOnly |
    ↪  QIODevice::Text)) {
        file1.write("QDir 文件 1 内容\n");
        file1.close();
    }
    QFile file2(currentDir.filePath("file2.txt"));
    if (file2.open(QIODevice::WriteOnly |
    ↪  QIODevice::Text)) {
        file2.write("QDir 文件 2 内容\n");
        file2.close();
    }
    currentDir.mkdir("subdir");
```

QDir 使用示例（二）

```
35      // 5. 列出当前目录下所有文件和子目录
36      qDebug() << "当前目录下的所有文件:";
37      QStringList files =
        ↪  currentDir.entryList(QDir::Files);
38      for (const QString &file : files) {
39          qDebug() << "文件:" << file;
40      }
41      qDebug() << "当前目录下的所有子目录:";
42      QStringList dirs = currentDir.entryList(QDir::Dirs
        ↪  | QDir::NoDotAndDotDot);
43      for (const QString &dir : dirs) {
44          qDebug() << "子目录:" << dir;
45      }
46      // 6. 返回上级目录
47      currentDir.cdUp();
48      qDebug() << "返回上级目录:" <<
        ↪  currentDir.absolutePath();
49      // 7. 删除文件和目录
50      QDir delDir(newDirName);
51      delDir.remove("file1.txt");
52      delDir.remove("file2.txt");
53      delDir.rmdir("subdir");
```

```
54      if (currentDir.rmdir(newDirName)) {
55          qDebug() << "删除目录成功:" << newDirName;
56      } else {
57          qDebug() << "删除目录失败:" << newDirName;
58      }
59      // 8. 获取主目录、临时目录、根目录
60      qDebug() << "主目录:" << QDir::homePath();
61      qDebug() << "临时目录:" << QDir::tempPath();
62      qDebug() << "根目录:" << QDir::rootPath();
63      // 9. 判断某个路径是否为绝对路径
64      QString absPath = "/usr/bin";
65      QString relPath = "relative/path";
66      qDebug() << absPath << "是否为绝对路径:" <<
        ↪  QDir::isAbsolutePath(absPath);
67      qDebug() << relPath << "是否为绝对路径:" <<
        ↪  QDir::isAbsolutePath(relPath);
68      // 10. 拼接路径
69      QDir someDir("/home/user");
70      QString fullPath = someDir.filePath("myfile.txt");
71      qDebug() << "拼接后的完整路径:" << fullPath;
```

QDir 使用示例（三）

```
    // 11. 过滤文件（如只列出 txt 文件）
    QDir filterDir = QDir::current();
    QStringList nameFilters;
    nameFilters << "*.txt";
    QStringList txtFiles =
    ↪  filterDir.entryList(nameFilters, QDir::Files);
    qDebug() << "当前目录下的 txt 文件:";
    for (const QString &file : txtFiles) {
        qDebug() << file;
    }
    // 12. 遍历目录（递归列出所有文件）
    qDebug() << "递归列出当前目录及子目录下所有文件:";
    QStringList allFiles;
    std::function<void(const QDir&)> listAllFiles =
    ↪  [&](const QDir& dir) {
        QStringList files = dir.entryList(QDir::Files);
        for (const QString &file : files) {
            allFiles << dir.absoluteFilePath(file);
        }
        QStringList subdirs = dir.entryList(QDir::Dirs
        ↪  | QDir::NoDotAndDotDot);
        for (const QString &subdir : subdirs) {
            QDir nextDir(dir.absoluteFilePath(subdir));
            listAllFiles(nextDir);
        }
    };
    listAllFiles(QDir::current());
    for (const QString &file : allFiles) {
        qDebug() << file;
    }
```

QDir 使用示例（四）

```
99      // 13. 判断目录是否存在
100     QString checkDir = "test_dir";
101     if (QDir(checkDir).exists()) {
102         qDebug() << checkDir << "目录存在";
103     } else {
104         qDebug() << checkDir << "目录不存在";
105     }
106     // 14. 重命名目录
107     QString oldName = "old_dir";
108     QString newName = "new_dir";
109     QDir().mkdir(oldName);
110     if (QDir().rename(oldName, newName)) {
111         qDebug() << "目录重命名成功:" << oldName << "->" << newName;
112         QDir().rmdir(newName);
113     } else {
114         qDebug() << "目录重命名失败";
115     }
116     // 15. 计算相对路径
117     QString relativePath = QDir::current().relativeFilePath("test_dir");
118     qDebug() << "test_dir 相对于当前目录的相对路径:" << relativePath;
119 }
```

QFileInfo 使用示例

```cpp
#include <QFileInfo>
#include <QDebug>
#include <QDateTime>
#include <iostream>
void fileInfoExample()
{
    // 1. 构造 QFileInfo 对象
    QString filePath = "test.txt";
    QFileInfo info(filePath);
    // 2. 判断文件是否存在
    if (info.exists()) {
        qDebug() << "文件存在:" << info.fileName();
    } else {
        qDebug() << "文件不存在:" << filePath;
        return;
    }
    // 3. 获取文件的绝对路径和相对路径
    qDebug() << "绝对路径:" << info.absoluteFilePath();
    qDebug() << "相对路径:" << info.filePath();
    // 4. 获取文件名、后缀、基本名
    qDebug() << "文件名:" << info.fileName();
    qDebug() << "后缀:" << info.suffix();
    qDebug() << "基本名:" << info.baseName();
    // 5. 获取文件大小
    qDebug() << "文件大小 (字节):" << info.size();
    // 6. 获取文件的创建、修改、访问时间
    qDebug() << "创建时间:" <<
    ↪  info.birthTime().toString();
    qDebug() << "修改时间:" <<
    ↪  info.lastModified().toString();
    qDebug() << "访问时间:" <<
    ↪  info.lastRead().toString();
    // 7. 判断文件类型
    if (info.isFile()) {
        qDebug() << "这是一个普通文件";
    }
    if (info.isDir()) {
        qDebug() << "这是一个目录";
    }
    // 8. 判断文件权限
    qDebug() << "可读:" << info.isReadable();
    qDebug() << "可写:" << info.isWritable();
    qDebug() << "可执行:" << info.isExecutable();
    // 9. 获取上级目录
    qDebug() << "上级目录:" << info.absolutePath();
```

QTextStream 简介

- **QTextStream** 类是 Qt 中用于文本流操作的类，主要用于读写文本文件。
- 继承自 **QIODevice**，可以像操作二进制文件一样操作文本文件。
- 提供了简单易用的接口，支持多种文本编码（如 UTF-8、UTF-16、UTF-32 等）。
- 常用于文件的逐行读取、写入、格式化输出等场景。

QTextStream 使用示例

```cpp
#include <QFile>
#include <QTextStream>
#include <QDebug>
#include <iostream>
void textStreamExample()
{
    // 1. 写入文本文件
    QFile outFile("textstream_test.txt");
    if (outFile.open(QIODevice::WriteOnly |
    ↪  QIODevice::Text)) {
        QTextStream out(&outFile);
        out << "第一行文本" << Qt::endl;
        out << "第二行文本" << Qt::endl;
        out << QString("数字: %1").arg(123) <<
        ↪  Qt::endl;
        outFile.close();
        qDebug() << "写入文件完成";
    } else {
        qDebug() << "无法打开文件进行写入";
        return;
    }
    // 2. 读取文本文件
    QFile inFile("textstream_test.txt");
    if (inFile.open(QIODevice::ReadOnly |
    ↪  QIODevice::Text)) {
        QTextStream in(&inFile);
        qDebug() << "读取文件内容:";
        while (!in.atEnd()) {
            QString line = in.readLine();
            qDebug() << line;
        }
        inFile.close();
    } else {
        qDebug() << "无法打开文件进行读取";
    }
}
```

QDataStream 简介

- **QDataStream** 类是 Qt 中用于二进制数据流读写的类，常用于对象、结构体、基本数据类型的序列化与反序列化。
- 支持多种 Qt 数据类型（如 QString、QList、QMap 等）以及自定义类型的读写。
- 主要用于文件、网络等场景下的数据持久化和传输。
- 通过重载 << 和 >> 运算符实现数据的序列化与反序列化。
- 可以设置版本号，保证不同 Qt 版本间的数据兼容性。

QDataStream 使用示例

```cpp
#include <QFile>
#include <QDataStream>
#include <QDebug>
#include <QString>
#include <QList>
#include <iostream>
void dataStreamExample()
{
    // 1. 写入二进制数据到文件
    QFile outFile("datastream_test.dat");
    if (outFile.open(QIODevice::WriteOnly)) {
        QDataStream out(&outFile);
        out.setVersion(QDataStream::Qt_5_0);
        int a = 42;
        double b = 3.14159;
        QString str = "Qt 数据流测试";
        QList<int> list = {1, 2, 3, 4, 5};
        out << a << b << str << list;
        outFile.close();
        qDebug() << "二进制数据写入完成";
    } else {
        qDebug() << "无法打开文件进行写入";
        return;
    }
    // 2. 从文件读取二进制数据
    QFile inFile("datastream_test.dat");
    if (inFile.open(QIODevice::ReadOnly)) {
        QDataStream in(&inFile);
        in.setVersion(QDataStream::Qt_5_0);
        int a;
        double b;
        QString str;
        QList<int> list;
        in >> a >> b >> str >> list;
        qDebug() << "int:" << a;
        qDebug() << "double:" << b;
        qDebug() << "QString:" << str;
        qDebug() << "QList<int>:" << list;
        inFile.close();
    } else {
        qDebug() << "无法打开文件进行读取";
    }
```

QT 日期与时间

QDateTime 简介

- **QDateTime** 类是 Qt 中用于处理日期和时间的类，提供了一个统一的接口来处理各种日期和时间相关的操作。
- 支持多种时区、时区转换、日期计算、格式化输出等。
- 常用于日期时间计算、格式化输出、数据库操作等场景。
- 支持与 QDate、QTime、QElapsedTimer 等类协作，方便处理日期时间相关操作。

QT 日期与时间示例（一）

```cpp
#include <QDateTime>
#include <QDate>
#include <QTime>
#include <QTimeZone>
#include <QDebug>
#include <QString>
#include <QLocale>
#include <QElapsedTimer>
#include <iostream>
void dateTimeExample()
{
    // 1. 获取当前日期和时间（本地和 UTC）
    QDateTime current = QDateTime::currentDateTime();
    QDateTime currentUtc =
    ↪  QDateTime::currentDateTimeUtc();
    qDebug() << "当前本地日期和时间:" <<
    ↪  current.toString("yyyy-MM-dd hh:mm:ss");
    qDebug() << "当前 UTC 日期和时间:" <<
    ↪  currentUtc.toString("yyyy-MM-dd hh:mm:ss");
    // 2. 获取当前日期和时间的详细信息
    QDate date = current.date();
    QTime time = current.time();
    qDebug() << "当前日期:" <<
    ↪  date.toString("yyyy-MM-dd");
    qDebug() << "当前时间:" <<
    ↪  time.toString("hh:mm:ss");
    qDebug() << "当前星期:" <<
    ↪  QLocale::system().dayName(date.dayOfWeek());
    qDebug() << "今年的第几天:" << date.dayOfYear();
    qDebug() << "今年的第几周:" << date.weekNumber();
    // 3. 构造指定日期和时间
    QDateTime dt(QDate(2024, 6, 1), QTime(12, 30, 0));
    qDebug() << "指定日期和时间:" <<
    ↪  dt.toString("yyyy-MM-dd hh:mm:ss");
    // 4. 日期时间加减
    QDateTime future =
    ↪  current.addDays(5).addSecs(3600);
    QDateTime past =
    ↪  current.addYears(-1).addMonths(-2);
    qDebug() << "5 天后加 1 小时:" <<
    ↪  future.toString("yyyy-MM-dd hh:mm:ss");
    qDebug() << "1 年前 2 个月前:" <<
    ↪  past.toString("yyyy-MM-dd hh:mm:ss");
```

QT 日期与时间示例（二）

```
33      // 5. 时间戳转换
34      qint64 timestamp = current.toSecsSinceEpoch();
35      qint64 msecTimestamp = current.toMSecsSinceEpoch();
36      qDebug() << "当前时间戳（秒）:" << timestamp;
37      qDebug() << "当前时间戳（毫秒）:" << msecTimestamp;
38      QDateTime fromTimestamp =
        ↪  QDateTime::fromSecsSinceEpoch(timestamp);
39      QDateTime fromMsecTimestamp =
        ↪  QDateTime::fromMSecsSinceEpoch(msecTimestamp);
40      qDebug() << "由时间戳（秒）还原:" <<
        ↪  fromTimestamp.toString("yyyy-MM-dd hh:mm:ss");
41      qDebug() << "由时间戳（毫秒）还原:" <<
        ↪  fromMsecTimestamp.toString("yyyy-MM-dd
        ↪  hh:mm:ss");
42      // 6. 字符串与 QDateTime 互转
43      QString dateTimeStr = "2024-06-01 15:20:30";
44      QDateTime parsed =
        ↪  QDateTime::fromString(dateTimeStr, "yyyy-MM-dd
        ↪  hh:mm:ss");
45      qDebug() << "字符串转 QDateTime:" <<
        ↪  parsed.toString("yyyy-MM-dd hh:mm:ss");
```

```
46      QString formatted = parsed.toString(Qt::ISODate);
47      qDebug() << "QDateTime 转 ISO 字符串:" << formatted;
48      // 7. 时区转换
49      QDateTime utc = current.toUTC();
50      qDebug() << "UTC 时间:" << utc.toString("yyyy-MM-dd
        ↪  hh:mm:ss");
51      QTimeZone beijingZone("Asia/Shanghai");
52      if (beijingZone.isValid()) {
53          QDateTime beijing =
            ↪  utc.toTimeZone(beijingZone);
54          qDebug() << "北京时间:" <<
            ↪  beijing.toString("yyyy-MM-dd hh:mm:ss");
55      } else {
56          qDebug() << "无法识别 Asia/Shanghai 时区";
57      }
58      // 8. 日期时间计算
59      QDateTime nextMonth = current.addMonths(1);
60      QDateTime nextYear = current.addYears(1);
61      qDebug() << "下个月:" <<
        ↪  nextMonth.toString("yyyy-MM-dd hh:mm:ss");
62      qDebug() << "明年:" <<
        ↪  nextYear.toString("yyyy-MM-dd hh:mm:ss");
```

QT 日期与时间示例（三）

```
    // 9. 日期时间比较
    if (current > dt)
        qDebug() << "当前日期和时间大于指定日期时间";
    else if (current < dt)
        qDebug() << "当前日期和时间小于指定日期时间";
    else
        qDebug() << "当前日期和时间等于指定日期时间";
    // 10. 日期时间间隔（天、秒、毫秒）
    int daysDiff = current.daysTo(future);
    qint64 secsDiff = current.secsTo(future);
    qint64 msecsDiff = current.msecsTo(future);
    qDebug() << "当前到未来的天数差:" << daysDiff;
    qDebug() << "当前到未来的秒数差:" << secsDiff;
    qDebug() << "当前到未来的毫秒数差:" << msecsDiff;
    // 11. QDate、QTime 常用操作
    QDate leapDate(2024, 2, 29);
    qDebug() << "2024 年 2 月 29 日是否为闰年:" <<
    ↪  leapDate.isLeapYear(leapDate.year());
    QTime t1(10, 20, 30);
    QTime t2 = t1.addSecs(3600);
    qDebug() << "10:20:30 加 1 小时:" <<
    ↪  t2.toString("hh:mm:ss");
    qDebug() << "两个时间的秒数差:" << t1.secsTo(t2);
    // 12. QDateTime 的有效性与空值判断
    QDateTime invalid;
    qDebug() << "未初始化 QDateTime 是否有效:" <<
    ↪  invalid.isValid();
    QDateTime nullDt = QDateTime();
    qDebug() << "空 QDateTime 是否为 null:" <<
    ↪  nullDt.isNull();
    // 13. 计时器
    qDebug() << "计时开始";
    QElapsedTimer timer;
    timer.start();
    for (int i = 0; i < 1000000; i++) {
        QDateTime::currentDateTime();
    }
    qDebug() << "计时结束，耗时（毫秒）:" <<
    ↪  timer.elapsed();
}
```

QProcess 简介

- **QProcess** 类是 Qt 中用于启动和管理外部进程的类，提供了丰富的接口用于与外部程序进行交互。
- 支持异步和同步启动外部进程，可以读取和写入子进程的标准输入、标准输出和标准错误。
- 可用于执行系统命令、脚本、调用其他可执行程序，并获取其输出结果。
- 支持跨平台，自动处理不同操作系统下的进程调用方式。
- 常用方法包括 `start()`（启动进程）、`waitForFinished()`（等待进程结束）、`readAllStandardOutput()`（读取标准输出）、`readAllStandardError()`（读取标准错误）、`write()`（向进程写入数据）等。
- 适用于需要与外部程序协作、自动化脚本、批处理等场景。

QProcess 使用示例

```
#include <QCoreApplication>
#include <QProcess>
#include <QDebug>
#include <QStringList>
#include <iostream>
void processExample()
{
    // 1. 创建 QProcess 对象
    QProcess process;
    // 2. 设置要执行的外部命令（以"ping" 为例，跨平台处理）
#if defined(Q_OS_WIN)
    QString program = "ping";
    QStringList arguments;
    arguments << "127.0.0.1" << "-n" << "2";
#else
    QString program = "ping";
    QStringList arguments;
    arguments << "127.0.0.1" << "-c" << "2";
#endif
    // 3. 启动进程并等待完成
    process.start(program, arguments);
    bool finished = process.waitForFinished(5000); // 最多等待 5 秒
    if (finished) {
        // 4. 读取标准输出
        QString output = process.readAllStandardOutput();
        qDebug() << "进程输出：";
        qDebug().noquote() << output;
        // 5. 读取标准错误
        QString errorOutput = process.readAllStandardError();
        if (!errorOutput.isEmpty()) {
            qDebug() << "进程错误输出：";
            qDebug().noquote() << errorOutput;
        }
        // 6. 获取退出码
        int exitCode = process.exitCode();
        qDebug() << "进程退出码：" << exitCode;
    } else {
        qDebug() << "进程未在规定时间内完成";
    }
}
```

QSettings 简介

- **QSettings** 类是 Qt 中用于读写应用程序配置文件的工具类，支持跨平台，自动适配不同操作系统的配置存储方式（如 Windows 注册表、INI 文件、macOS 的 plist 等）。
- 通过键值对的方式存储和读取各种类型的数据（如 int、double、QString、QVariant 等），支持分组（Group）和层级结构，便于组织复杂配置。
- 常用方法包括 `setValue(key, value)`（写入配置）、`value(key, default)`（读取配置）、`remove(key)`（删除配置）、`contains(key)`（判断配置项是否存在）等。
- 支持自动保存和加载，无需手动管理文件读写，适合保存用户设置、应用参数、窗口状态等。
- 支持多种构造方式，可指定组织名、应用名、文件格式等，灵活适配不同项目需求。
- 典型用法示例见下页代码。

QSettings 使用示例（一）

```cpp
#include <QCoreApplication>
#include <QSettings>
#include <QDebug>
#include <QString>
#include <QVariant>
#include <iostream>
void settingsExample()
{
    // 1. 创建 QSettings 对象（以 INI 文件格式，指定组织和应用名）
    QSettings settings("MyCompany", "MyApp");
    // 2. 写入各种类型的配置项
    settings.setValue("username", "alice");
    settings.setValue("window/width", 800);
    settings.setValue("window/height", 600);
    settings.setValue("volume", 0.75);
    settings.setValue("isAdmin", true);
    // 3. 分组写入
    settings.beginGroup("network");
    settings.setValue("host", "192.168.1.100");
    settings.setValue("port", 8080);
    settings.endGroup();
    // 4. 读取配置项（带默认值）
    QString username = settings.value("username", "defaultuser").toString();
    int width = settings.value("window/width", 1024).toInt();
    int height = settings.value("window/height", 768).toInt();
    double volume = settings.value("volume", 1.0).toDouble();
    bool isAdmin = settings.value("isAdmin", false).toBool();
    qDebug() << "用户名:" << username;
    qDebug() << "窗口宽度:" << width;
    qDebug() << "窗口高度:" << height;
    qDebug() << "音量:" << volume;
    qDebug() << "是否管理员:" << isAdmin;
```

QSettings 使用示例（二）

```
33      // 5. 读取分组内的配置
34      settings.beginGroup("network");
35      QString host = settings.value("host",
        ↪  "localhost").toString();
36      int port = settings.value("port", 80).toInt();
37      settings.endGroup();
38      qDebug() << "网络主机:" << host;
39      qDebug() << "网络端口:" << port;
40      // 6. 判断配置项是否存在
41      if (settings.contains("username")) {
42          qDebug() << "存在用户名配置";
43      }
44      // 7. 删除配置项
45      settings.remove("isAdmin");
46      qDebug() << "删除 isAdmin 后，isAdmin 存在吗？" <<
        ↪  settings.contains("isAdmin");
47      // 8. 枚举所有键
48      QStringList keys = settings.allKeys();
49      qDebug() << "所有配置项键值：";
50      for (const QString &key : keys) {
51          qDebug() << key << ":" <<
            ↪  settings.value(key).toString();
52      }
53  }
```

Ini 文件内容

```
[MyCompany/MyApp]
username=alice
window/width=800
window/height=600
volume=0.75
isAdmin=true
[network]
host=192.168.1.100
port=8080
```

QTimer 简介

- **QTimer** 类是 Qt 中用于定时器的类，提供了简单的定时器功能。
- 支持单次定时和多次定时，可以设置定时器的间隔时间。
- 常用方法包括 `start()`（启动定时器）、`stop()`（停止定时器）、`setInterval()`（设置定时器间隔时间）、`setSingleShot()`（设置定时器是否单次定时）等。
- 适用于需要定时执行某些操作的场景。

QTimer 使用示例

```cpp
#include <QCoreApplication>
#include <QTimer>
#include <QDebug>
#include <iostream>
// 定时器回调槽函数
void onTimeout()
{
    static int count = 0;
    count++;
    qDebug() << "定时器超时，第" << count << "次";
    if (count >= 5) {
        // 超过 5 次后退出应用
        QCoreApplication::quit();
    }
}
```

```cpp
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    // 1. 创建 QTimer 对象
    QTimer timer;
    // 2. 连接 timeout 信号到槽函数
    QObject::connect(&timer, &QTimer::timeout,
    ↪  &onTimeout);
    // 3. 设置定时器间隔（每隔 1 秒触发一次）
    timer.setInterval(1000);
    // 4. 启动定时器
    timer.start();

    qDebug() << "定时器已启动，每秒触发一次，共 5 次后退
    ↪  出";
    app.exec();
    std::cin.get();
    return 0;
}
```

QThread 简介

- **QThread** 类是 Qt 中用于线程的类，提供了简单的线程功能。
- 支持线程的创建、启动、停止、暂停、恢复等操作。
- 常用方法包括 `start()`（启动线程）、`stop()`（停止线程）、`pause()`（暂停线程）、`resume()`（恢复线程）等。
- 适用于需要多线程处理的场景。

QThread 使用示例

```cpp
#include <QCoreApplication>
#include <QThread>
#include <QDebug>
#include <iostream>
// 自定义线程类，继承自 QThread
class WorkerThread : public QThread
{
protected:
    void run() override {
        for (int i = 1; i <= 5; ++i) {
            qDebug() << "工作线程正在运行，计数：" << i;
            QThread::sleep(1); // 线程休眠 1 秒
        }
        qDebug() << "工作线程结束";
    }

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    WorkerThread thread;
    qDebug() << "主线程：启动工作线程";
    thread.start(); // 启动线程，自动调用 run()

    // 等待线程结束
    thread.wait();
    qDebug() << "主线程：工作线程已结束";
    app.exec();
    std::cin.get();
    return 0;
}
```

QtConcurrent 简介

- **QtConcurrent** 类是 Qt 中用于并行计算的类，提供了简单的并行计算功能。
- 支持并行计算，可以设置并行计算的线程数。
- 常用方法包括 `run()`（启动并行计算）、`waitForFinished()`（等待并行计算结束）等。
- 适用于需要并行计算的场景。

QtConcurrent 使用示例

```cpp
#include <QCoreApplication>
#include <QtConcurrent>
#include <QDebug>
#include <QThread>
#include <QVector>
#include <iostream>
// 一个耗时的计算函数
int slowSquare(int x)
{
    QThread::sleep(1); // 模拟耗时操作
    qDebug() << "计算" << x << "的平方，线程 ID:" <<
    ↪ QThread::currentThreadId();
    return x * x;
}

void concurrentExample()
{
    QVector<int> numbers = {1, 2, 3, 4, 5};
    qDebug() << "主线程 ID:" <<
    ↪ QThread::currentThreadId();

    // 1. 并行计算每个元素的平方
    QFuture<int> future = QtConcurrent::mapped(numbers,
    ↪ slowSquare);

    // 2. 等待所有任务完成
    future.waitForFinished();

    // 3. 获取结果
    QVector<int> results = future.results().toVector();
    qDebug() << "所有平方结果:" << results;
}

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    concurrentExample();

    std::cin.get();
    return 0;
```

QPainter 简介

- **QPainter** 类是 Qt 中用于绘图的类，提供了强大的绘图功能。
- 支持绘制各种图形、文本、图像等。
- 支持路径、笔刷、渐变等。
- 支持绘制到各种 QPaintDevice，如 QWidget、QImage 等。
- 常用在 paintEvent() 中进行自定义绘制。
- 支持多种绘制模式，如填充、描边、剪切等。
- 支持多种绘制效果，如阴影、透明、渐变等。

QPainter 使用示例（一）

```cpp
#include <QApplication>
#include <QWidget>
#include <QPainter>
#include <QPen>
#include <QBrush>
#include <QLinearGradient>
#include <QRadialGradient>
#include <QConicalGradient>
#include <QPixmap>
#include <QFont>
#include <QPainterPath>

class PaintWidget : public QWidget {
protected:
    void paintEvent(QPaintEvent *event) override {
        QPainter painter(this);
        // 1. 设置抗锯齿和高质量渲染
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setRenderHint(QPainter::TextAntialiasing,
↪  true);
painter.setRenderHint(QPainter::SmoothPixmapTransform,
↪  true);
        // 2. 绘制不同颜色和线宽的直线
        QPen pen1(Qt::red, 3, Qt::SolidLine,
        ↪  Qt::RoundCap, Qt::RoundJoin);
        painter.setPen(pen1);
        painter.drawLine(20, 20, 180, 20);
        // 3. 绘制带虚线的矩形
        QPen pen2(Qt::blue, 2, Qt::DashLine);
        painter.setPen(pen2);
        painter.drawRect(20, 40, 80, 60);
        // 4. 绘制填充线性渐变的椭圆
        QLinearGradient linearGrad(60, 120, 140, 180);
        linearGrad.setColorAt(0, Qt::yellow);
        linearGrad.setColorAt(1, Qt::green);
        painter.setPen(Qt::NoPen);
        painter.setBrush(QBrush(linearGrad));
        painter.drawEllipse(100, 40, 80, 60);
```

QPainter 使用示例（二）

```
        // 5. 绘制填充径向渐变的圆（带阴影效果）
        QRadialGradient radialGrad(60, 160, 30, 60,
        ↪  160);
        radialGrad.setColorAt(0, Qt::white);
        radialGrad.setColorAt(1, Qt::darkGray);
        painter.setBrush(QBrush(radialGrad));
        painter.drawEllipse(40, 140, 40, 40);
        // 6. 绘制锥形渐变的扇形
        QConicalGradient conicalGrad(180, 180, 0);
        conicalGrad.setColorAt(0, Qt::red);
        conicalGrad.setColorAt(0.5, Qt::yellow);
        conicalGrad.setColorAt(1, Qt::red);
        painter.setBrush(QBrush(conicalGrad));
        painter.setPen(QPen(Qt::darkRed, 2));
        QRectF arcRect(160, 160, 40, 40);
        painter.drawPie(arcRect, 30 * 16, 120 * 16);
        // 7. 绘制多边形
        painter.setPen(QPen(Qt::darkMagenta, 2));
        painter.setBrush(QBrush(Qt::cyan,
        ↪  Qt::Dense4Pattern));
        QPoint points[4] = {QPoint(120, 120),
        ↪  QPoint(180, 120), QPoint(170, 180),
        ↪  QPoint(130, 180)};
        painter.drawPolygon(points, 4);
        // 8. 绘制贝塞尔曲线
        painter.setPen(QPen(Qt::darkBlue, 2,
        ↪  Qt::DashDotLine));
        QPainterPath path;
        path.moveTo(20, 120);
        path.cubicTo(60, 80, 100, 160, 180, 100);
        painter.drawPath(path);
        // 9. 绘制带背景色的文本
        painter.setPen(Qt::black);
        painter.setFont(QFont("微软雅黑", 14,
        ↪  QFont::Bold));
        painter.setBrush(QBrush(Qt::lightGray));
        painter.drawRect(20, 200, 160, 30);
        painter.drawText(25, 222, "QPainter 功能演示");
        // 10. 绘制图片（假设有一张图片在同目录下）
        QPixmap pix("qt_logo.png");
        if (!pix.isNull())
            painter.drawPixmap(120, 10, 60, 60, pix);
```

QPainter 使用示例（三）

```
72          // 11. 保存和恢复状态
73          painter.save();
74          painter.translate(100, 100);
75          painter.rotate(30);
76          painter.setPen(QPen(Qt::darkGreen, 2));
77          painter.setBrush(Qt::NoBrush);
78          painter.drawRect(-20, -20, 40, 40);
79          painter.restore();
80          // 12. 剪切区域
81          painter.setClipRect(150, 150, 40, 40);
82          painter.setBrush(Qt::red);
83          painter.setPen(Qt::NoPen);
84          painter.drawEllipse(140, 140, 60, 60);
85          // 13. 绘制透明图形
86          QColor semiTransBlue(0, 0, 255, 100);
87          painter.setBrush(QBrush(semiTransBlue));
```

```
88          painter.setPen(Qt::NoPen);
89          painter.drawRect(60, 60, 60, 60);
90          // 14. 绘制点和椭圆弧
91          painter.setPen(QPen(Qt::darkCyan, 3));
92          painter.drawPoint(200, 200);
93          painter.setPen(QPen(Qt::darkYellow, 2));
94          painter.drawArc(30, 180, 60, 40, 30 * 16, 120 *
            ↪  16);
95      }
96  };
```

目录

总结

本章要点回顾

- **Qt 框架概述**：Qt 是一个跨平台 C++ 开发框架，提供 GUI、网络、多媒体等模块，支持桌面、移动和嵌入式平台。
- **核心模块与结构**：包括 QtCore、QtGui、QtWidgets 等模块；应用程序基于 QObject 和元对象系统构建。
- **元对象系统**：通过 Q_OBJECT 宏、moc 编译器和 QMetaObject 实现运行时类型信息、信号槽和属性系统。
- **信号槽机制**：Qt 的核心通信方式，支持对象间松耦合，支持多对多连接和跨线程通信，优于传统事件处理。
- **事件系统**：包括鼠标、键盘等事件；通过事件循环、过滤器和处理器实现灵活响应。
- **容器与工具类**：Qt 提供高效容器（如 QList、QMap）和工具（如 QString、QByteArray、QFile、QTimer、QThread），支持隐式共享和线程安全。
- **Qt Widgets 编程**：涵盖基本控件（如 QWidget、QDialog、QMainWindow）和高级组件（如 QPushButton、QLabel、QSlider、QProgressBar），用于构建桌面 GUI。

建议

学习建议

- 多动手实践信号槽和事件机制，尝试自定义信号与槽，深入理解 Qt 的动态特性和对象间通信原理。
- 结合本章示例代码，动手实现常见 Widgets 的创建、属性设置和信号响应，掌握控件的基本用法。
- 练习不同的布局管理器（如 QHBoxLayout、QVBoxLayout、QGridLayout 等），尝试实现复杂界面布局，提升界面设计能力。
- 阅读 Qt 官方文档和 API 手册，查阅控件的更多属性和方法，善用文档解决实际开发中的问题。
- 尝试将多个控件组合，开发小型实用工具或 Demo 项目，巩固所学知识。
- 主动探索高级主题，如多线程编程、网络通信、自定义控件绘制、样式表（QSS）美化等，拓展 Qt 开发视野。
- 参与 Qt 开源社区或查阅优秀开源项目，学习他人代码风格和工程组织方式，提升综合开发能力。