

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 5 日

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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

- 1 Qt 框架概述
- 2 Qt 核心模块**
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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

```
1  #include <QApplication> // Qt 应用程序类
2  #include <QMainWindow> // Qt 主窗口类
3  #include <QLabel> // Qt 标签类
4  #include <QVBoxLayout> // Qt 垂直布局类
5  #include <QWidget> // Qt 窗口部件类
6
7  int main(int argc, char *argv[]) // 主函数
8  {
9      // 创建应用程序对象
10     QApplication app(argc, argv);
11
12     // 创建主窗口
13     QMainWindow mainWindow;
14     mainWindow.setWindowTitle("Qt Application
    ↳ Example");
15     mainWindow.resize(400, 300);
16
17     // 创建中心部件
18     QWidget *centralWidget = new
    ↳ QWidget(&mainWindow);
19     // 设置主窗口的中心部件
20     mainWindow.setCentralWidget(centralWidget);
```

```
20     mainWindow.setCentralWidget(centralWidget);
21
22     // 创建布局
23     QVBoxLayout *layout = new
    ↳ QVBoxLayout(centralWidget);
24
25     // 创建标签
26     QLabel *label = new QLabel("Hello, Qt!",
    ↳ centralWidget);
27     // 设置标签的文本对齐方式
28     label->setAlignment(Qt::AlignCenter);
29     // 将标签添加到布局中
30     layout->addWidget(label);
31
32     // 显示窗口
33     mainWindow.show();
34
35     // 进入事件循环
36     return app.exec();
37 }
```

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统**
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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

```
1  #include <QObject>
2  #include <QString>
3
4  class MyClass : public QObject
5  {
6      Q_OBJECT // 启用元对象系统
7
8  public:
9      explicit MyClass(QObject *parent = nullptr);
10
11     // 属性声明
12     Q_PROPERTY(QString name READ name WRITE setName
13     ⇨ NOTIFY nameChanged)
14     Q_PROPERTY(int age READ age WRITE setAge NOTIFY
15     ⇨ ageChanged)
16
17     // 属性访问方法
18     QString name() const { return m_name; }
19     void setName(const QString &name);
20
21     int age() const { return m_age; }
22     void setAge(int age);
```

```
20     void setAge(int age);
21
22     signals:
23         // 信号声明
24         void nameChanged(const QString &name);
25         void ageChanged(int age);
26
27     private slots:
28         // 槽函数声明
29         void onNameChanged(const QString &name);
30
31     private:
32         QString m_name;
33         int m_age;
34     };
```

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。

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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

```
1  #include <iostream>
2  // 定义一个函数，作为事件发生时的响应（回调函数）
3  void onButtonClick() {
4      std::cout << "Button is clicked!" << std::endl;
5  }
6  // 模拟一个“按钮”对象，它有一个点击事件
7  class Button {
8  public:
9      // 保存事件回调函数的指针
10     void (*onClick)() = nullptr;
11     // 模拟点击按钮的动作
12     void click() {
13         std::cout << "Button is being clicked..." <<
14             ↪ std::endl;
15         if (onClick) {
16             onClick(); // 触发事件
17         }
18     };
19 }
```

```
20 // 主函数演示
21 int main() {
22     Button btn;
23
24     // 注册事件: 把 onButtonClick 函数赋给 onClick
25     btn.onClick = onButtonClick;
26
27     // 模拟用户点击
28     btn.click();
29     // 等待用户输入
30     std::cin.get();
31     return 0;
32 }
```


Qt 事件示例

```
1  #include <QCoreApplication>
2  #include <QEvent>
3  #include <QObject>
4  #include <QDebug>
5  // 自定义一个按钮点击事件类型
6  const QEvent::Type ButtonClickEventType =
↪   static_cast<QEvent::Type>(QEvent::User + 1);
7  // 自定义事件类
8  class ButtonClickEvent : public QEvent {
9  public:
10     ButtonClickEvent() : QEvent(ButtonClickEventType)
↪     {}
11 };
12 // 按钮类, 继承自 QObject
13 class Button : public QObject {
14     Q_OBJECT
15 public:
16     // 发送点击事件
17     void click() {
18         qDebug() << "按钮: 正在被点击...";
19         // 创建事件对象
20         ButtonClickEvent event;
```

```
21         // 发送事件到自身
22         QCoreApplication::sendEvent(this, &event);
23     }
24 protected:
25     // 重写事件处理函数
26     bool event(QEvent *e) override {
27         if (e->type() == ButtonClickEventType) {
28             qDebug() << "按钮被点击了! (事件处理)";
29             return true;
30         }
31         return QObject::event(e);
32     }
33 };
34 int main(int argc, char *argv[]) {
35     QCoreApplication app(argc, argv);
36     Button btn;
37     // 模拟用户点击
38     btn.click();
39     return app.exec();
40 }
```

- **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 <code>event()</code> 、 <code>mousePressEvent()</code> ），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

```
1  #include <QCoreApplication>
2  #include <QObject>
3  #include <QDebug>
4  // 定义一个按钮类，继承自 QObject，支持信号槽
5  class Button : public QObject
6  {
7      Q_OBJECT
8  public:
9      // 模拟点击按钮的动作
10     void click() {
11         qDebug() << "按钮：正在被点击...";
12         emit clicked();
13     }
14     signals:
15         void clicked();
16 };
17
```

```
18 // 定义一个槽函数的接收者
19 class Receiver : public QObject
20 {
21     Q_OBJECT
22 public slots:
23     void onButtonClicked() {
24         qDebug() << "按钮被点击了! ";
25     }
26 };
27
28 int main(int argc, char *argv[])
29 {
30     QCoreApplication app(argc, argv);
31     Button btn;
32     Receiver receiver;
33     // 信号与槽连接
34     QObject::connect(&btn, &Button::clicked,
35                     ↪ &receiver, &Receiver::onButtonClicked);
36     // 模拟用户点击
37     btn.click();
38     return app.exec();
39 }
```

信号槽连接方式 (一)

```
1  #include <QWidget>
2  #include <QApplication>
3  #include <QPushButton>
4  #include <QLabel>
5  #include <QDebug>
6  #include <QVBoxLayout>
7  class SignalSlotExample : public QWidget
8  {
9      Q_OBJECT
10 public:
11     SignalSlotExample(QWidget *parent = nullptr) :
12         QWidget(parent) {
13         setWindowTitle("Signal Slot Example");
14         resize(300, 150);
15     }
16     void setupConnections() {
17         m_button = new QPushButton("Test", this);
18         m_label = new QLabel("Status", this);
19         QVBoxLayout *layout = new QVBoxLayout(this);
20         layout->addWidget(m_label);
21         layout->addWidget(m_button);
```

```
21         setLayout(layout);
22
23         // 方法 1: 新语法 (推荐)
24         connect(m_button, &QPushButton::clicked,
25                this,
26                ↪ &SignalSlotExample::onButtonClicked);
27
28         // 方法 2: Lambda 表达式
29         connect(m_button, &QPushButton::clicked,
30                ↪ [this]() {
31                    qDebug() << "Button clicked via lambda";
32                });
33
34         // 方法 3: 旧语法 (不推荐)
35         connect(m_button, SIGNAL(clicked()),
36                this, SLOT(onButtonClicked()));
37
38         // 方法 4: 带参数的连接
39         connect(this,
40                ↪ &SignalSlotExample::statusChanged,
41                m_label, &QLabel::setText);
42     }
```

信号槽连接方式 (二)

```
26  int main()
27  {
28      Receiver* receiver1 = new Receiver();
29      Receiver* receiver2 = new Receiver();
30      Sender* sender1 = new Sender();
31      Sender* sender2 = new Sender();
32      // 多对一: 多个信号连接同一个槽
33      QObject::connect(sender1, &Sender::statusChanged,
34      ↪ receiver1, &Receiver::onStatusChanged);
35      QObject::connect(sender2, &Sender::statusChanged,
36      ↪ receiver1, &Receiver::onStatusChanged);
37      sender1->doSomething();
38      sender2->doSomething();
39      // 断开连接
40      QObject::disconnect(sender1,
41      ↪ &Sender::statusChanged, receiver1,
42      ↪ &Receiver::onStatusChanged);
```

```
39      QObject::disconnect(sender2,
40      ↪ &Sender::statusChanged, receiver1,
41      ↪ &Receiver::onStatusChanged);
42      sender1->doSomething();
43      sender2->doSomething();
44      // 一对多: 一个信号连接多个槽
45      QObject::connect(sender1, &Sender::statusChanged,
46      ↪ receiver1, &Receiver::onStatusChanged);
47      QObject::connect(sender1, &Sender::statusChanged,
48      ↪ receiver2, &Receiver::onStatusChanged);
49      sender1->doSomething();
50      // 断开全部
51      QObject::disconnect(sender1,
52      ↪ &Sender::statusChanged, nullptr, nullptr);
53      sender1->doSomething();
54      // 释放对象时, 会自动断开信号和槽的连接
55      delete receiver1;
56      delete receiver2;
57      delete sender1;
58      delete sender2;
59      std::cin.get();
60      return 0;
```

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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

```
1  #include <QWidget>
2  #include <QMouseEvent>
3  #include <QDebug>
4  #include <QPainter>
5
6  class MouseEventWidget : public QWidget
7  {
8      Q_OBJECT
9  public:
10     MouseEventWidget(QWidget *parent = nullptr) :
11         QWidget(parent) {
12         setMouseTracking(true); // 启用鼠标跟踪
13     }
14 protected:
15     void mousePressEvent(QMouseEvent *event) override
16     {
17         qDebug() << "鼠标按下, 位置:" << event->pos();
18         m_lastPos = event->pos();
19         update();
20     }
```

```
21     void mouseMoveEvent(QMouseEvent *event) override
22     {
23         qDebug() << "鼠标移动, 位置:" << event->pos();
24         m_lastPos = event->pos();
25         update();
26     }
27     void mouseReleaseEvent(QMouseEvent *event)
28     {
29         qDebug() << "鼠标释放, 位置:" << event->pos();
30         m_lastPos = event->pos();
31         update();
32     }
33     void paintEvent(QPaintEvent *) override {
34         QPainter painter(this);
35         painter.setPen(Qt::red);
36         painter.drawEllipse(m_lastPos, 5, 5);
37     }
38
39 private:
40     QPoint m_lastPos;
41 };
```

键盘事件示例

```
1  #include <QWidget>
2  #include <QKeyEvent>
3  #include <QDebug>
4  #include <QApplication>
5  #include <QSet>
6
7  class KeyboardEventWidget : public QWidget
8  {
9      Q_OBJECT
10 public:
11     KeyboardEventWidget(QWidget *parent = nullptr) :
12         QWidget(parent) {
13         setFocusPolicy(Qt::StrongFocus); // 允许接收键
14         ↪ 盘焦点
15     }
16
17 protected:
18     void keyPressEvent(QKeyEvent *event) override {
19         m_pressedKeys.insert(event->key());
20         // 组合键示例: Ctrl + S
21         if (m_pressedKeys.contains(Qt::Key_Control)
22             ↪ && m_pressedKeys.contains(Qt::Key_S)) {
23             qDebug() << "检测到组合键: Ctrl + S";
24         }
25     }
```

```
22     // 组合键示例: Shift + A
23     else if
24     ↪ (m_pressedKeys.contains(Qt::Key_Shift) &&
25     ↪ m_pressedKeys.contains(Qt::Key_A)) {
26         qDebug() << "检测到组合键: Shift + A";
27     }
28     // 只按下单个键
29     else {
30         qDebug() << "按下键: " <<
31         ↪ QKeySequence(event->key()).toString()
32         ↪ << "(key code:" << event->key() <<
33         ↪ ")";
34     }
35     update();
36
37 void keyReleaseEvent(QKeyEvent *event) override {
38     m_pressedKeys.remove(event->key());
39     qDebug() << "释放键: " <<
40     ↪ QKeySequence(event->key()).toString() <<
41     ↪ "(key code:" << event->key() << ")";
42 }
43
44 private:
45     QSet<int> m_pressedKeys;
46 };
```

窗口事件示例

```
1  #include <QWidget>
2  #include <QEvent>
3  #include <QDebug>
4  #include <QApplication>
5  #include <QPainter>
6  #include <QResizeEvent>
7  #include <QMoveEvent>
8  #include <QPaintEvent>
9  #include <QShowEvent>
10 #include <QHideEvent>
11 #include <QCloseEvent>
12 class WindowEventWidget : public QWidget
13 {
14     Q_OBJECT
15 public:
16     WindowEventWidget(QWidget *parent = nullptr) :
17         QWidget(parent) {}
18 protected:
19     void showEvent(QShowEvent *event) override {
20         qDebug() << "窗口显示事件";
21         QWidget::showEvent(event);
22     }
```

```
22     void hideEvent(QHideEvent *event) override {
23         qDebug() << "窗口隐藏事件";
24         QWidget::hideEvent(event);
25     }
26
27     void closeEvent(QCloseEvent *event) override {
28         qDebug() << "窗口关闭事件";
29         QWidget::closeEvent(event);
30     }
31     void resizeEvent(QResizeEvent *event) override {
32         qDebug() << "窗口大小改变:" << event->size();
33         QWidget::resizeEvent(event);
34     }
35     void moveEvent(QMoveEvent *event) override {
36         qDebug() << "窗口移动到:" << event->pos();
37         QWidget::moveEvent(event);
38     }
39     void paintEvent(QPaintEvent *event) override {
```

拖放事件示例

```
1  #include <QWidget>
2  #include <QDragEnterEvent>
3  #include <QDropEvent>
4  #include <QMimeData>
5  #include <QDebug>
6  #include <QPainter>
7  #include <QApplication>
8
9  class DragDropWidget : public QWidget
10 {
11     Q_OBJECT
12 public:
13     DragDropWidget(QWidget *parent = nullptr) :
14         QWidget(parent) {
15         setAcceptDrops(true); // 启用拖放
16         m_text = "请拖拽文本到此窗口";
17     }
18 protected:
19     void dragEnterEvent(QDragEnterEvent *event)
20     ↪ override {
21         // 判断拖入的数据类型是否为文本
22         if (event->mimeTypeData()->hasText()) {
```

```
23         } else {
24             event->ignore();
25         }
26     }
27     void dropEvent(QDropEvent *event) override {
28         if (event->mimeTypeData()->hasText()) {
29             m_text = event->mimeTypeData()->text();
30             qDebug() << "拖拽释放, 接收到文本:" <<
31                 ↪ m_text;
32             event->acceptProposedAction();
33             update();
34         } else {
35             event->ignore();
36         }
37     }
38     void paintEvent(QPaintEvent *) override {
39         QPainter painter(this);
40         painter.setPen(Qt::blue);
41         painter.drawText(rect(), Qt::AlignCenter,
42             ↪ m_text);
43     }
44 private:
45     QString m_text;
```

定时器事件示例

```
1  #include <QWidget>
2  #include <QTimerEvent>
3  #include <QDebug>
4  #include <QPainter>
5  #include <QApplication>
6  class TimerEventWidget : public QWidget
7  {
8      Q_OBJECT
9  public:
10     TimerEventWidget(QWidget *parent = nullptr) :
        ↳ QWidget(parent), m_counter(0) {
11         // 启动定时器, 间隔 1000 毫秒 (1 秒)
12         m_timerId = startTimer(1000);
13     }
14 protected:
15     void timerEvent(QTimerEvent *event) override {
16         if (event->timerId() == m_timerId) {
17             m_counter++;
18             qDebug() << "定时器事件触发, 计数:" <<
        ↳ m_counter;
19             update();
20         }
21     }
```

```
22     void paintEvent(QPaintEvent *) override {
23         QPainter painter(this);
24         painter.setPen(Qt::darkGreen);
25         painter.drawText(rect(), Qt::AlignCenter,
        ↳ QString("定时器计数: %1").arg(m_counter));
26     }
27 private:
28     int m_timerId;
29     int m_counter;
30 };
31
32 int main(int argc, char *argv[])
33 {
34     QApplication app(argc, argv);
35     TimerEventWidget w;
36     w.resize(400, 200);
37     w.show();
38     return app.exec();
39 }
```

自定义事件示例

```
1  #include <QWidget>
2  #include <QEvent>
3  #include <QDebug>
4  #include <QApplication>
5  #include <QPushButton>
6  // 定义自定义事件类型
7  const QEvent::Type MyCustomEventType =
    ↳ static_cast<QEvent::Type>(QEvent::User + 1);
8  class MyCustomEvent : public QEvent
9  {
10 public:
11     MyCustomEvent(const QString& msg)
12         : QEvent(MyCustomEventType), m_message(msg)
13         ↳ {}
14     QString message() const { return m_message; }
15 private:
16     QString m_message;
17 };
18
19 class CustomEventWidget : public QWidget
20 {
21     Q_OBJECT
22 public:
```

```
23     CustomEventWidget(QWidget *parent = nullptr) :
    ↳ QWidget(parent)
24     {
25         QPushButton *btn = new QPushButton("发送自定义
    ↳ 事件", this);
26         btn->setGeometry(100, 80, 200, 40);
27         connect(btn, &QPushButton::clicked, this,
    ↳ [this]() {
28             // 发送自定义事件
29             QApplication::postEvent(this, new
    ↳ MyCustomEvent("Hello, 这是自定义事
    ↳ 件!"));
30         });
31     }
32 protected:
33     // 事件处理
34     void customEvent(QEvent *event) override
35     {
36         if (event->type() == MyCustomEventType) {
37             MyCustomEvent *myEvent =
    ↳ static_cast<MyCustomEvent*>(event);
38             qDebug() << "收到自定义事件, 消息内容:" <<
    ↳ myEvent->message();
39         }
40     }
```

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

```
1  #include <QWidget>
2  #include <QEvent>
3  #include <QDebug>
4  #include <QApplication>
5  #include <QTimer>
6
7  // 事件循环演示控件
8  class EventLoopWidget : public QWidget
9  {
10     Q_OBJECT
11 public:
12     EventLoopWidget(QWidget *parent = nullptr) :
13         QWidget(parent), m_counter(0)
14     {
15         // 启动定时器，每秒触发一次
16         QTimer *timer = new QTimer(this);
17         connect(timer, &QTimer::timeout, this,
18             ↳ &EventLoopWidget::onTimeout);
19         timer->start(1000);
20     }
21
22 protected:
```

```
21     // 重写事件处理函数，演示事件循环分发
22     bool event(QEvent *event) override
23     {
24         if (event->type() == QEvent::Timer) {
25             qDebug() << "收到定时器事件，计数:" <<
26                 ↳ m_counter;
27         }
28         return QWidget::event(event);
29     }
30 private slots:
31     void onTimeout()
32     {
33         m_counter++;
34         qDebug() << "定时器超时，发送自定义事件，计数:"
35             ↳ << m_counter;
36         // 这里可以发送自定义事件或做其他操作
37     }
38 private:
39     int m_counter;
40 };
```


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

```
1  #include <QWidget>
2  #include <QEvent>
3  #include <QKeyEvent>
4  #include <QDebug>
5  #include <QApplication>
6
7  // 自定义事件过滤器
8  class MyEventFilter : public QObject {
9      Q_OBJECT
10 public:
11     explicit MyEventFilter(QObject *parent = nullptr)
12         ↳ : QObject(parent) {}
13 protected:
14     bool eventFilter(QObject *, QEvent *event)
15         ↳ override {
16         if (event->type() == QEvent::KeyPress) {
17             QKeyEvent *keyEvent =
18                 ↳ static_cast<QKeyEvent*>(event);
19             qDebug() << "过滤器: 按键:" <<
20                 ↳ QKeySequence(keyEvent->key()).toString();
21             if (keyEvent->key() == Qt::Key_Escape) {
22                 qDebug() << "过滤器: 拦截 Esc";
23                 return true;
24             }
25         }
26     }
```

```
20     }
21 }
22 return false;
23 }
24 };
25
26 // 事件过滤器控件
27 class EventFilterWidget : public QWidget {
28     Q_OBJECT
29 public:
30     EventFilterWidget(QWidget *parent = nullptr) :
31         ↳ QWidget(parent) {
32         installEventFilter(new MyEventFilter(this));
33         setFocusPolicy(Qt::StrongFocus);
34     }
35 protected:
36     void keyPressEvent(QKeyEvent *event) override {
37         qDebug() << "收到按键: " <<
38             ↳ QKeySequence(event->key()).toString();
39     }
40 }
```

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

```
1  #include <QWidget>
2  #include <QEvent>
3  #include <QKeyEvent>
4  #include <QDebug>
5  #include <QApplication>
6
7  // 事件处理器控件
8  class EventHandlerWidget : public QWidget
9  {
10     Q_OBJECT
11     public:
12         EventHandlerWidget(QWidget *parent = nullptr) :
13             QWidget(parent) {
14             setFocusPolicy(Qt::StrongFocus);
15         }
```

```
15     protected:
16         bool event(QEvent *event) override {
17             if (event->type() == QEvent::KeyPress) {
18                 QKeyEvent *keyEvent =
19                     ↪ static_cast<QKeyEvent*>(event);
20                 qDebug() << "事件处理器: 按键:" <<
21                     ↪ QKeySequence(keyEvent->key()).toString();
22                 if (keyEvent->key() == Qt::Key_Escape) {
23                     qDebug() << "事件处理器: 拦截 Esc";
24                     return true;
25                 }
26             }
27             return QWidget::event(event);
28         }
```

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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。

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

```
1  #include <QList>
2  #include <QString>
3  #include <QDebug>
4  #include <algorithm>
5  #include <iostream>
6
7  void listExample()
8  {
9      // 创建 QList 并添加元素
10     QList<int> numbers;
11     numbers << 10 << 20 << 30;
12     numbers.append(40);
13     numbers.prepend(5);
14     qDebug() << "QList<int> 内容:";
15     for (int i = 0; i < numbers.size(); ++i) {
16         qDebug() << "索引" << i << ":" <<
            ↪ numbers.at(i);
17     }
18     // 插入和移除元素
19     numbers.insert(2, 15); // 在索引 2 插入 15
```

```
20     numbers.removeAt(0); // 移除第 0 个元素
21     qDebug() << "修改后的 QList<int> 内容:";
22     for (int n : numbers) {
23         qDebug() << n;
24     }
25     // 查找元素
26     int idx = numbers.indexOf(30);
27     if (idx != -1) {
28         qDebug() << "30 的索引为:" << idx;
29     }
30     // 反转和排序
31     std::sort(numbers.begin(), numbers.end());
32     std::reverse(numbers.begin(), numbers.end());
33     qDebug() << "排序并反转后的 QList<int> 内容:" <<
        ↪ numbers;
34     // 支持自定义类型
35     QList<QString> names;
36     names << "Alice" << "Bob" << "Charlie";
37     qDebug() << "QList<QString> 内容:" << names;
38 }
```


QMap 使用示例

```
1  #include <QMap>
2  #include <QString>
3  #include <QDebug>
4  #include <iostream>
5
6  void mapExample()
7  {
8      // 创建 QMap 并插入元素
9      QMap<QString, int> scores;
10     scores.insert("Alice", 90);
11     scores.insert("Bob", 85);
12     scores["Charlie"] = 92;
13     scores["David"] = 88;
14     qDebug() << "QMap 内容:";
15     // 遍历 QMap (按 key 有序)
16     QMap<QString, int>::const_iterator it;
17     for (it = scores.constBegin(); it !=
18         ↪ scores.constEnd(); ++it) {
19         qDebug() << it.key() << ":" << it.value();
20     }
```

```
20     // 查找元素
21     if (scores.contains("Bob")) {
22         qDebug() << "Bob 的分数:" <<
23         ↪ scores.value("Bob");
24     }
25     // 修改元素
26     scores["Alice"] = 95;
27     qDebug() << "修改后 Alice 的分数:" <<
28     ↪ scores.value("Alice");
29     // 移除元素
30     scores.remove("David");
31     qDebug() << "移除 David 后 QMap 内容:" << scores;
32     // 获取所有 key 和 value
33     QList<QString> keys = scores.keys();
34     QList<int> values = scores.values();
35     qDebug() << "所有学生:" << keys;
36     qDebug() << "所有分数:" << values;
37     // 清空 QMap
38     scores.clear();
39     qDebug() << "清空后 QMap 是否为空:" <<
40     ↪ scores.isEmpty();
41 }
```

QHash 使用示例

```
1  #include <QHash>
2  #include <QString>
3  #include <QDebug>
4  #include <iostream>
5
6  void hashExample()
7  {
8      // 创建 QHash 并插入元素
9      QHash<QString, int> ages;
10     ages.insert("Alice", 25);
11     ages.insert("Bob", 30);
12     ages["Charlie"] = 28;
13     ages["David"] = 22;
14     qDebug() << "QHash 内容:";
15     // 遍历 QHash (无序)
16     QHash<QString, int>::const_iterator it;
17     for (it = ages.constBegin(); it !=
18         ↪ ages.constEnd(); ++it) {
19         qDebug() << it.key() << ":" << it.value();
20     }
```

```
20     // 查找元素
21     if (ages.contains("Bob")) {
22         qDebug() << "Bob 的年龄:" <<
23             ↪ ages.value("Bob");
24     }
25     // 修改元素
26     ages["Alice"] = 26;
27     qDebug() << "修改后 Alice 的年龄:" <<
28         ↪ ages.value("Alice");
29     // 移除元素
30     ages.remove("David");
31     qDebug() << "移除 David 后 QHash 内容:" << ages;
32     // 获取所有 key 和 value
33     QList<QString> keys = ages.keys();
34     QList<int> values = ages.values();
35     qDebug() << "所有人名:" << keys;
36     qDebug() << "所有年龄:" << values;
37     // 清空 QHash
38     ages.clear();
39     qDebug() << "清空后 QHash 是否为空:" <<
40         ↪ ages.isEmpty();
41 }
```

QSet 使用示例

```
1  #include <QSet>
2  #include <QString>
3  #include <QDebug>
4  #include <iostream>
5  void setExample()
6  {
7      // 创建 QSet 并添加元素
8      QSet<QString> fruits;
9      fruits << "Apple" << "Banana" << "Orange";
10     fruits.insert("Grape");
11     fruits.insert("Banana"); // 重复元素不会被添加
12     qDebug() << "QSet 内容:";
13     for (const QString &fruit : fruits) {
14         qDebug() << fruit;
15     }
16     // 判断元素是否存在
17     if (fruits.contains("Apple")) {
18         qDebug() << "集合中包含 Apple";
```

```
19     }
20     // 移除元素
21     fruits.remove("Orange");
22     qDebug() << "移除 Orange 后 QSet 内容:" << fruits;
23     // 集合操作: 并集、交集、差集
24     QSet<QString> tropicalFruits;
25     tropicalFruits << "Banana" << "Mango" <<
        ↪ "Pineapple";
26     QSet<QString> unionSet =
        ↪ fruits.unite(tropicalFruits); // 并集
27     QSet<QString> intersectSet =
        ↪ fruits.intersect(tropicalFruits); // 交集
28     QSet<QString> diffSet =
        ↪ fruits.subtract(tropicalFruits); // 差集
29     qDebug() << "并集:" << unionSet;
30     qDebug() << "交集:" << intersectSet;
31     qDebug() << "差集:" << diffSet;
32     // 清空集合
33     fruits.clear();
34     qDebug() << "清空后 QSet 是否为空:" <<
        ↪ fruits.isEmpty();
35 }
```

QStack 使用示例

```
1  #include <QStack>
2  #include <QString>
3  #include <QDebug>
4  #include <iostream>
5
6  void stackExample()
7  {
8      // 创建 QStack 并压入元素
9      QStack<QString> stack;
10     stack.push("第一步");
11     stack.push("第二步");
12     stack.push("第三步");
13     qDebug() << "QStack 内容（从栈底到栈顶）:";
14     for (int i = 0; i < stack.size(); ++i) {
15         qDebug() << stack.at(i);
16     }
```

```
17
18     // 查看栈顶元素
19     if (!stack.isEmpty()) {
20         qDebug() << "栈顶元素:" << stack.top();
21     }
22
23     // 弹出元素
24     QString popped = stack.pop();
25     qDebug() << "弹出元素:" << popped;
26     qDebug() << "弹出后栈顶:" << (stack.isEmpty() ?
    ↪ "空" : stack.top());
27
28     // 判断是否为空
29     qDebug() << "QStack 是否为空:" << stack.isEmpty();
30
31     // 清空栈
32     stack.clear();
33     qDebug() << "清空后 QStack 是否为空:" <<
    ↪ stack.isEmpty();
34 }
```

QQueue 使用示例

```
1  #include <QQueue>
2  #include <QString>
3  #include <QDebug>
4  #include <iostream>
5  void queueExample()
6  {
7      // 创建 QQueue 并入队元素
8      QQueue<QString> queue;
9      queue.enqueue("任务 1");
10     queue.enqueue("任务 2");
11     queue.enqueue("任务 3");
12     qDebug() << "QQueue 内容 (从队首到队尾) :";
13     for (int i = 0; i < queue.size(); ++i) {
14         qDebug() << queue.at(i);
15     }
16
17     // 查看队首和队尾元素
```

```
18     if (!queue.isEmpty()) {
19         qDebug() << "队首元素:" << queue.head();
20         qDebug() << "队尾元素:" << queue.back();
21     }
22
23     // 出队元素
24     QString front = queue.dequeue();
25     qDebug() << "出队元素:" << front;
26     qDebug() << "出队后队首:" << (queue.isEmpty() ?
    ↪ "空" : queue.head());
27
28     // 判断是否为空
29     qDebug() << "QQueue 是否为空:" << queue.isEmpty();
30
31     // 清空队列
32     queue.clear();
33     qDebug() << "清空后 QQueue 是否为空:" <<
    ↪ queue.isEmpty();
34 }
```

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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)

```
1  #include <QString>
2  #include <QStringList>
3  #include <QByteArray>
4  #include <QDebug>
5  #include <iostream>
6  void stringExample()
7  {
8      // 1. 创建和初始化 QString
9      QString str1 = QStringLiteral("Hello");
10     QString str2 = QStringLiteral("世界");
11     QString str3 = QString::fromUtf8("Qt 字符串示例");
12     // 2. 拼接字符串
13     QString str4 = QString("%1, %2!").arg(str1,
14     ↪ str2); // 推荐使用 arg 格式化
15     str4.append(QStringLiteral(" 欢迎使用 Qt."));
16     qDebug() << "拼接后的字符串:" << str4;
17     // 3. 格式化字符串
```

```
18     int year = 2024;
19     double pi = 3.14159;
20     QString str5 = QStringLiteral("今年是%1, 圆周率约
21     ↪ 为%2").arg(year).arg(pi, 0, 'f', 2);
22     qDebug() << "格式化字符串:" << str5;
23     // 4. 查找和替换
24     int pos = str4.indexOf(QStringLiteral("Qt"));
25     if (pos != -1) {
26         qDebug() << R"("Qt" 在 str4 中的位置:)" <<
27         ↪ pos;
28     }
29     QString str6 = str4;
30     str6.replace(QStringLiteral("Qt"),
31     ↪ QStringLiteral("C++"));
32     qDebug() << "替换后的字符串:" << str6;
33     // 5. 分割和连接
34     QString fruits =
35     ↪ QStringLiteral("Apple,Banana,Orange");
36     QStringList fruitList = fruits.split(',');
37     qDebug() << "分割后的 QStringList:" << fruitList;
38     QString joined = fruitList.join(QStringLiteral("
39     ↪ | "));
40     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)

```
1  #include <QStringList>
2  #include <QString>
3  #include <QDebug>
4  #include <iostream>
5  void stringListExample()
6  {
7      // 1. 创建和初始化 QStringList
8      QStringList list1;
9      list1 << "Apple" << "Banana" << "Orange";
10     qDebug() << "初始化的 QStringList:" << list1;
11     // 2. 通过字符串分割生成 QStringList
12     QString fruits =
13     ↪ QStringLiteral("Mango,Peach,Pear");
14     QStringList list2 = fruits.split(',');
15     qDebug() << "分割字符串得到的 QStringList:" <<
16     ↪ list2;
17     // 3. 拼接 QStringList 为字符串
18     QString joined = list1.join(" | ");
19     qDebug() << "用 | 连接的字符串:" << joined;
```

```
18     // 4. 添加和插入元素
19     list1.append("Grape");
20     list1.prepend("Pineapple");
21     list1.insert(2, "Lemon");
22     qDebug() << "添加和插入后的 QStringList:" <<
23     ↪ list1;
24     // 5. 查找和移除元素
25     int idx = list1.indexOf("Banana");
26     if (idx != -1) {
27         qDebug() << "\"Banana\"的位置:" << idx;
28         list1.removeAt(idx);
29     }
30     list1.removeAll("Orange");
31     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)

```
1  #include <QByteArray>
2  #include <QList>
3  #include <QDebug>
4  #include <vector>
5  #include <string>
6  #include <iostream>
7  void byteArrayExample()
8  {
9      // 1. 多种方式创建 QByteArray
10     QByteArray ba1("Hello, Qt!"); // C 字符串初始化
11     QByteArray ba2 =
        ↳ QByteArray::fromStdString(std::string("QByteArray
        ↳ 示例")); // std::string 初始化
12     QByteArray ba3(10, 'A'); // 10 个'A'
13     QByteArray ba4; // 空 QByteArray
14     qDebug() << "ba1 原始内容:" << ba1;
15     qDebug() << "ba2 原始内容:" << ba2;
16     qDebug() << "ba3 原始内容:" << ba3;
17     qDebug() << "ba4 是否为空:" << ba4.isEmpty();
```

```
18     // 2. 追加、插入、前置、替换、移除
19     ba1.append(" Welcome.");
20     ba1.prepend(">>> ");
21     ba1.insert(7, "C++ ");
22     qDebug() << "追加、前置、插入后:" << ba1;
23     ba1.replace("C++", "Qt/C++");
24     ba1.remove(0, 4);
25     qDebug() << "替换和移除后:" << ba1;
26     // 3. 查找、判断、访问
27     int pos = ba1.indexOf("Qt");
28     if (pos != -1) {
29         qDebug() << "“Qt” 在 ba1 中的位置:" << pos;
30     }
31     qDebug() << "ba1 是否包含 Welcome:" <<
        ↳ ba1.contains("Welcome");
32     qDebug() << "ba1 的长度:" << ba1.length();
33     if (!ba1.isEmpty())
34         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:
42     ↪ %2").arg(idx++).arg(QString::fromUtf8(item));
43 }
44 QByteArray joined = list.join(" | ");
45 qDebug() << "用 | 拼接结果:" << joined;
46 // 5. 转换为数字、十六进制、Base64、URL 编码
47 QByteArray ba6("12345");
48 int num = ba6.toInt();
49 double dnum = QByteArray("3.14159").toDouble();
50 qDebug() << "字符串转整数:" << num;
51 qDebug() << "字符串转浮点数:" << dnum;
52 QByteArray hex = ba6.toHex();
53 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 =
60     ↪ QByteArray::fromBase64(base64);
61 QByteArray fromUrl =
62     ↪ QByteArray::fromPercentEncoding(urlEncoded);
63 qDebug() << "十六进制解码:" << fromHex;
64 qDebug() << "Base64 解码:" << fromBase64;
65 qDebug() << "URL 解码:" << fromUrl;
66 // 7. 与 QString 互转
67 QString str1 = QString::fromUtf8(ba1);
68 QByteArray ba7 = str1.toUtf8();
69 qDebug() << "QByteArray 转 QString:" << str1;
70 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)

```
1  #include <QVariant>
2  #include <QString>
3  #include <QStringList>
4  #include <QDateTime>
5  #include <QByteArray>
6  #include <QList>
7  #include <QMap>
8  #include <QDebug>
9  #include <iostream>
10 #if (QT_VERSION >= QT_VERSION_CHECK(5, 7, 0))
11 #include <QJsonObject>
12 #endif
13 // 自定义类型声明与注册 (建议放在全局, 避免多次注册)
14 struct MyType {
15     int x;
16 };
```

```
18
19 void variantExample()
20 {
21     // 1. 存储基本类型
22     QVariant vInt = 42;
23     QVariant vDouble = 3.14159;
24     QVariant vBool = true;
25     QVariant vString = QStringLiteral("Hello
    ↪ QVariant");
26     QVariant vDateTime =
    ↪ QDateTime::currentDateTime();
27     qDebug() << "int:" << vInt.toInt()
28              << "double:" << vDouble.toDouble()
29              << "bool:" << vBool.toBool()
30              << "QString:" << vString.toString();
```

QVariant 使用示例 (二)

```
31         << "QDateTime:" <<
           ↳ vDateTime.toDateTime().toString(Qt::ISODate); 47
32 // 2. 存储 Qt 容器类型
33 QStringList strList = {"Apple", "Banana",
   ↳ "Orange"};
34 QVariant vStrList = strList;
35 qDebug() << "QStringList:" <<
   ↳ vStrList.toStringList();
36 QList<int> intList = {1, 2, 3, 4, 5};
37 QVariant vIntList = QVariant::fromValue(intList);
38 qDebug() << "QList<int>:" <<
   ↳ vIntList.value<QList<int>>();
39 // 3. 存储 QByteArray
40 QByteArray ba("Qt Variant");
41 QVariant vByteArray = ba;
42 qDebug() << "QByteArray:" <<
   ↳ vByteArray.toByteArray();
43 // 4. 类型判断与转换
44 if (vInt.typeId() == QMetaType::Int)
45     qDebug() << "vInt 是 int 类型";
46 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
48 QVariant vNull;
49 qDebug() << "vNull 是否有效:" << vNull.isValid()
51     << "vNull 是否为 null:" <<
       ↳ vNull.isNull();
// 6. QVariant 的类型安全访问
52 vInt.setValue(2024);
53 qDebug() << "类型安全访问 int:" <<
54     ↳ vInt.value<int>();
// 7. QVariant 的拷贝与比较
55 QVariant vCopy = vString;
56 qDebug() << "vCopy == vString:" << (vCopy ==
   ↳ vString);
57 vCopy = 100;
58 qDebug() << "vCopy 赋新值后:" << vCopy;
// 8. QVariant 的类型名与类型 ID
59 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)

```
94 // 14. QVariant 的 isNull/isValid 区别演示
95 QVariant vEmptyString = QString();
96 qDebug() << "空 QString 的 QVariant isNull:" <<
  ↳ vEmptyString.isNull()
97     << "isValid:" << vEmptyString.isValid();
98 // 15. QVariant 的 setValue 与模板 value<T>() 的配
  ↳ 合
99 QVariant vAny;
100 vAny.setValue(QString("AnyType"));
101 qDebug() << "setValue 后 value<QString>:" <<
  ↳ vAny.value<QString>();
102 // 16. QVariant 与自定义类型 (注册自定义类型)
103 // 注册类型 (通常在 main 函数或程序初始化时注册一次)
104 static bool registered = [](){
105     qRegisterMetaType<MyType>("MyType");
106     return true;
107 }();
108 MyType myObj{10};
109 QVariant vCustom = QVariant::fromValue(myObj);
110 if (vCustom.canConvert<MyType>()) {
111     MyType outObj = vCustom.value<MyType>();
112     qDebug() << "自定义类型 MyType.x:" <<
  ↳ outObj.x;
113 }
```

```
12 #endif
13 // 自定义类型声明与注册 (建议放在全局, 避免多次注册)
14 struct MyType {
15     int x;
16 };
```

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)

```
1  #include <QFile>
2  #include <QTextStream>
3  #include <QDebug>
4  #include <QFileInfo>
5  #include <QDateTime>
6  #include <iostream>
7  void fileExample()
8  {
9      // 1. 创建 QFile 对象, 指定文件名
10     QString fileName = "example.txt";
11     QFile file(fileName);
12     // 2. 以写入模式打开文件 (会覆盖原有内容), 写入多行内
    ↪ 容
13     if (file.open(QIODevice::WriteOnly |
    ↪ QIODevice::Text)) {
14         QTextStream out(&file);
15         out << "Hello, QFile!\n";
16         out << "Qt 文件操作示例.\n";
```

```
17         out << "当前时间: " <<
    ↪ QDateTime::currentDateTime().toString(Qt::ISODate)
    ↪ << "\n";
18         out << "支持中文写入.\n";
19         file.close();
20         qDebug() << "写入文件成功";
21     } else {
22         qDebug() << "无法打开文件进行写入";
23         return;
24     }
25     // 3. 以追加模式打开文件, 追加内容
26     if (file.open(QIODevice::Append |
    ↪ QIODevice::Text)) {
27         QTextStream out(&file);
28         out << "追加一行内容.\n";
29         file.close();
30         qDebug() << "追加内容成功";
31     } else {
32         qDebug() << "无法打开文件进行追加";
33     }
```

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 |
78     ↪ QIODevice::Text)) {
79     QString allText = copyFile.readAll();
80     qDebug() << "copy.txt 全部内容为: \n" <<
81     ↪ allText;
82     copyFile.close();
83 }
```

```
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)

```
1  #include <QDir>
2  #include <QDebug>
3  #include <QStringList>
4  #include <iostream>
5  void dirExample()
6  {
7      // 1. 获取当前工作目录
8      QDir currentDir = QDir::current();
9      qDebug() << "当前工作目录:" <<
    ↪ currentDir.absolutePath();
10     // 2. 创建新目录
11     QString newDirName = "test_dir";
12     if (currentDir.mkdir(newDirName)) {
13         qDebug() << "创建目录成功:" << newDirName;
14     } else {
15         qDebug() << "创建目录失败或已存在:" <<
    ↪ newDirName;
16     }
```

```
17     // 3. 进入新目录
18     if (currentDir.cd(newDirName)) {
19         qDebug() << "进入目录:" <<
    ↪ currentDir.absolutePath();
20     } else {
21         qDebug() << "进入目录失败:" << newDirName;
22     }
23     // 4. 创建多个文件和子目录
24     QFile file1(currentDir.filePath("file1.txt"));
25     if (file1.open(QIODevice::WriteOnly |
    ↪ QIODevice::Text)) {
26         file1.write("QDir 文件 1 内容\n");
27         file1.close();
28     }
29     QFile file2(currentDir.filePath("file2.txt"));
30     if (file2.open(QIODevice::WriteOnly |
    ↪ QIODevice::Text)) {
31         file2.write("QDir 文件 2 内容\n");
32         file2.close();
33     }
34     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)

```
72 // 11. 过滤文件 (如只列出 txt 文件)
73 QDir filterDir = QDir::current();
74 QStringList nameFilters;
75 nameFilters << "*.txt";
76 QStringList txtFiles =
    ↳ filterDir.entryList(nameFilters,
    ↳ QDir::Files);
77 qDebug() << "当前目录下的 txt 文件:";
78 for (const QString &file : txtFiles) {
79     qDebug() << file;
80 }
81 // 12. 遍历目录 (递归列出所有文件)
82 qDebug() << "递归列出当前目录及子目录下所有文件:";
83 QStringList allFiles;
84 std::function<void(const QDir&)> listAllFiles =
    ↳ [&](const QDir& dir) {
85     QStringList files =
        ↳ dir.entryList(QDir::Files);
```

```
86     for (const QString &file : files) {
87         allFiles << dir.absoluteFilePath(file);
88     }
89     QStringList subdirs =
    ↳ dir.entryList(QDir::Dirs |
    ↳ QDir::NoDotAndDotDot);
90     for (const QString &subdir : subdirs) {
91         QDir
            ↳ nextDir(dir.absoluteFilePath(subdir));
92         listAllFiles(nextDir);
93     }
94 };
95 listAllFiles(QDir::current());
96 for (const QString &file : allFiles) {
97     qDebug() << file;
98 }
```

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

```
1  #include <QFileInfo>
2  #include <QDebug>
3  #include <QDateTime>
4  #include <iostream>
5  void fileInfoExample()
6  {
7      // 1. 构造 QFileInfo 对象
8      QString filePath = "test.txt";
9      QFileInfo info(filePath);
10     // 2. 判断文件是否存在
11     if (info.exists()) {
12         qDebug() << "文件存在:" << info.fileName();
13     } else {
14         qDebug() << "文件不存在:" << filePath;
15         return;
16     }
17     // 3. 获取文件的绝对路径和相对路径
18     qDebug() << "绝对路径:" <<
19     ↪ info.absoluteFilePath();
20     qDebug() << "相对路径:" << info.filePath();
21     // 4. 获取文件名、后缀、基本名
22     qDebug() << "文件名:" << info.fileName();
23     qDebug() << "后缀:" << info.suffix();
24     qDebug() << "基本名:" << info.baseName();
```

```
24     // 5. 获取文件大小
25     qDebug() << "文件大小 (字节):" << info.size();
26     // 6. 获取文件的创建、修改、访问时间
27     qDebug() << "创建时间:" <<
28     ↪ info.birthTime().toString();
29     qDebug() << "修改时间:" <<
30     ↪ info.lastModified().toString();
31     qDebug() << "访问时间:" <<
32     ↪ info.lastRead().toString();
33     // 7. 判断文件类型
34     if (info.isFile()) {
35         qDebug() << "这是一个普通文件";
36     }
37     if (info.isDir()) {
38         qDebug() << "这是一个目录";
39     }
40     // 8. 判断文件权限
41     qDebug() << "可读:" << info.isReadable();
42     qDebug() << "可写:" << info.isWritable();
43     qDebug() << "可执行:" << info.isExecutable();
44     // 9. 获取上级目录
45     qDebug() << "上级目录:" << info.absolutePath();
```

- **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

```
1  #include <QFile>
2  #include <QTextStream>
3  #include <QDebug>
4  #include <iostream>
5  void textStreamExample()
6  {
7      // 1. 写入文本文件
8      QFile outFile("textstream_test.txt");
9      if (outFile.open(QIODevice::WriteOnly |
10 ↪ QIODevice::Text)) {
11          QTextStream out(&outFile);
12          out << "第一行文本" << Qt::endl;
13          out << "第二行文本" << Qt::endl;
14          out << QString("数字: %1").arg(123) <<
15 ↪ Qt::endl;
16          outFile.close();
17          qDebug() << "写入文件完成";
18      } else {
19          qDebug() << "无法打开文件进行写入";
20          return;
21      }
22  }
```

```
21      // 2. 读取文本文件
22      QFile inFile("textstream_test.txt");
23      if (inFile.open(QIODevice::ReadOnly |
24 ↪ QIODevice::Text)) {
25          QTextStream in(&inFile);
26          qDebug() << "读取文件内容:";
27          while (!in.atEnd()) {
28              QString line = in.readLine();
29              qDebug() << line;
30          }
31          inFile.close();
32      } else {
33          qDebug() << "无法打开文件进行读取";
34      }
35  }
```


- **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

```
1  #include <QFile>
2  #include <QDataStream>
3  #include <QDebug>
4  #include <QString>
5  #include <QList>
6  #include <iostream>
7  void dataStreamExample()
8  {
9      // 1. 写入二进制数据到文件
10     QFile outFile("datastream_test.dat");
11     if (outFile.open(QIODevice::WriteOnly)) {
12         QDataStream out(&outFile);
13         out.setVersion(QDataStream::Qt_5_0);
14         int a = 42;
15         double b = 3.14159;
16         QString str = "Qt 数据流测试";
17         QList<int> list = {1, 2, 3, 4, 5};
18         out << a << b << str << list;
19         outFile.close();
20         qDebug() << "二进制数据写入完成";
21     } else {
22         qDebug() << "无法打开文件进行写入";
23     }
24 }
```

```
24 }
25 // 2. 从文件读取二进制数据
26 QFile inFile("datastream_test.dat");
27 if (inFile.open(QIODevice::ReadOnly)) {
28     QDataStream in(&inFile);
29     in.setVersion(QDataStream::Qt_5_0);
30     int a;
31     double b;
32     QString str;
33     QList<int> list;
34     in >> a >> b >> str >> list;
35     qDebug() << "int:" << a;
36     qDebug() << "double:" << b;
37     qDebug() << "QString:" << str;
38     qDebug() << "QList<int>:" << list;
39     inFile.close();
40 } else {
41     qDebug() << "无法打开文件进行读取";
42 }
```

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)

```
1  #include <QDateTime>
2  #include <QDate>
3  #include <QTime>
4  #include <QTimeZone>
5  #include <QDebug>
6  #include <QString>
7  #include <QLocale>
8  #include <QElapsedTimer>
9  #include <iostream>
10 void dateTimeExample()
11 {
12     // 1. 获取当前日期和时间 (本地和 UTC)
13     QDateTime current = QDateTime::currentDateTime();
14     QDateTime currentUtc =
15         QDateTime::currentDateTimeUtc();
16     qDebug() << "当前本地日期和时间:" <<
17         current.toString("yyyy-MM-dd hh:mm:ss");
18     qDebug() << "当前 UTC 日期和时间:" <<
19         currentUtc.toString("yyyy-MM-dd hh:mm:ss");
20     // 2. 获取当前日期和时间的详细信息
```

```
20     qDebug() << "当前日期:" <<
21         date.toString("yyyy-MM-dd");
22     qDebug() << "当前时间:" <<
23         time.toString("hh:mm:ss");
24     qDebug() << "当前星期:" <<
25         QLocale::system().dayName(date.dayOfWeek());
26     qDebug() << "今年的第几天:" << date.dayOfYear();
27     qDebug() << "今年的第几周:" << date.weekNumber();
28     // 3. 构造指定日期和时间
29     QDateTime dt(QDate(2024, 6, 1), QTime(12, 30,
30         0));
31     qDebug() << "指定日期和时间:" <<
32         dt.toString("yyyy-MM-dd hh:mm:ss");
33     // 4. 日期时间加减
34     QDateTime future =
35         current.addDays(5).addSecs(3600);
36     QDateTime past =
37         current.addYears(-1).addMonths(-2);
38     qDebug() << "5 天后加 1 小时:" <<
39         future.toString("yyyy-MM-dd hh:mm:ss");
40     qDebug() << "1 年前 2 个月前:" <<
41         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)

```
63 // 9. 日期时间比较
64 if (current > dt)
65     qDebug() << "当前日期和时间大于指定日期时间";
66 else if (current < dt)
67     qDebug() << "当前日期和时间小于指定日期时间";
68 else
69     qDebug() << "当前日期和时间等于指定日期时间";
70 // 10. 日期时间间隔 (天、秒、毫秒)
71 int daysDiff = current.daysTo(future);
72 qint64 secsDiff = current.secsTo(future);
73 qint64 msecsDiff = current.msecsTo(future);
74 qDebug() << "当前到未来的天数差:" << daysDiff;
75 qDebug() << "当前到未来的秒数差:" << secsDiff;
76 qDebug() << "当前到未来的毫秒数差:" << msecsDiff;
77 // 11. QDate、QTime 常用操作
78 QDate leapDate(2024, 2, 29);
79 qDebug() << "2024 年 2 月 29 日是否为闰年:" <<
    ↪ leapDate.isLeapYear(leapDate.year());
80 QTime t1(10, 20, 30);
81 QTime t2 = t1.addSecs(3600);
82 qDebug() << "10:20:30 加 1 小时:" <<
    ↪ t2.toString("hh:mm:ss");
```

```
83 qDebug() << "两个时间的秒数差:" << t1.secsTo(t2);
84 // 12. QDateTime 的有效性 与 空值判断
85 QDateTime invalid;
86 qDebug() << "未初始化 QDateTime 是否有效:" <<
    ↪ invalid.isValid();
87 QDateTime nullDt = QDateTime();
88 qDebug() << "空 QDateTime 是否为 null:" <<
    ↪ nullDt.isNull();
89 // 13. 计时器
90 qDebug() << "计时开始";
91 QElapsedTimer timer;
92 timer.start();
93 for (int i = 0; i < 10000000; i++) {
94     QDateTime::currentDateTime();
95 }
96 qDebug() << "计时结束, 耗时 (毫秒):" <<
    ↪ timer.elapsed();
97 }
```

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

```
1  #include <QCoreApplication>
2  #include <QProcess>
3  #include <QDebug>
4  #include <QStringList>
5  #include <iostream>
6  void processExample()
7  {
8      // 1. 创建 QProcess 对象
9      QProcess process;
10     // 2. 设置要执行的外部命令 (以"ping" 为例, 跨平台处
        ↳ 理)
11     #if defined(Q_OS_WIN)
12         QString program = "ping";
13         QStringList arguments;
14         arguments << "127.0.0.1" << "-n" << "2";
15     #else
16         QString program = "ping";
17         QStringList arguments;
18         arguments << "127.0.0.1" << "-c" << "2";
19     #endif
20     // 3. 启动进程并等待完成
21     process.start(program, arguments);
```

```
22     bool finished = process.waitForFinished(5000); //
        ↳ 最多等待 5 秒
23     if (finished) {
24         // 4. 读取标准输出
25         QString output =
        ↳ process.readAllStandardOutput();
26         qDebug() << "进程输出: ";
27         qDebug().noquote() << output;
28         // 5. 读取标准错误
29         QString errorOutput =
        ↳ process.readAllStandardError();
30         if (!errorOutput.isEmpty()) {
31             qDebug() << "进程错误输出: ";
32             qDebug().noquote() << errorOutput;
33         }
34         // 6. 获取退出码
35         int exitCode = process.exitCode();
36         qDebug() << "进程退出码: " << exitCode;
37     } else {
38         qDebug() << "进程未在规定时间内完成";
39     }
40 }
```


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)

```
1  #include <QCoreApplication>
2  #include <QSettings>
3  #include <QDebug>
4  #include <QString>
5  #include <QVariant>
6  #include <iostream>
7  void settingsExample()
8  {
9      // 1. 创建 QSettings 对象 (以 INI 文件格式, 指定组织
      ↪ 和应用名)
10     QSettings settings("MyCompany", "MyApp");
11     // 2. 写入各种类型的配置项
12     settings.setValue("username", "alice");
13     settings.setValue("window/width", 800);
14     settings.setValue("window/height", 600);
15     settings.setValue("volume", 0.75);
16     settings.setValue("isAdmin", true);
```

```
17     // 3. 分组写入
18     settings.beginGroup("network");
19     settings.setValue("host", "192.168.1.100");
20     settings.setValue("port", 8080);
21     settings.endGroup();
22     // 4. 读取配置项 (带默认值)
23     QString username = settings.value("username",
      ↪ "defaultuser").toString();
24     int width = settings.value("window/width",
      ↪ 1024).toInt();
25     int height = settings.value("window/height",
      ↪ 768).toInt();
26     double volume = settings.value("volume",
      ↪ 1.0).toDouble();
27     bool isAdmin = settings.value("isAdmin",
      ↪ false).toBool();
28     qDebug() << "用户名:" << username;
29     qDebug() << "窗口宽度:" << width;
30     qDebug() << "窗口高度:" << height;
31     qDebug() << "音量:" << volume;
32     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

```
1  #include <QCoreApplication>
2  #include <QTimer>
3  #include <QDebug>
4  #include <iostream>
5  // 定时器回调槽函数
6  void onTimeout()
7  {
8      static int count = 0;
9      count++;
10     qDebug() << "定时器超时, 第" << count << "次";
11     if (count >= 5) {
12         // 超过 5 次后退出应用
13         QCoreApplication::quit();
14     }
15 }
```

```
17 int main(int argc, char *argv[])
18 {
19     QCoreApplication app(argc, argv);
20
21     // 1. 创建 QTimer 对象
22     QTimer timer;
23     // 2. 连接 timeout 信号到槽函数
24     QObject::connect(&timer, &QTimer::timeout,
25                     ↪ &onTimeout);
26     // 3. 设置定时器间隔 (每隔 1 秒触发一次)
27     timer.setInterval(1000);
28     // 4. 启动定时器
29     timer.start();
30
31     qDebug() << "定时器已启动, 每秒触发一次, 共 5 次后退
32     ↪ 出";
33     app.exec();
34     std::cin.get();
35     return 0;
36 }
```

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

```
1  #include <QCoreApplication>
2  #include <QThread>
3  #include <QDebug>
4  #include <iostream>
5  // 自定义线程类, 继承自 QThread
6  class WorkerThread : public QThread
7  {
8  protected:
9      void run() override {
10         for (int i = 1; i <= 5; ++i) {
11             qDebug() << "工作线程正在运行, 计数: " << i;
12             QThread::sleep(1); // 线程休眠 1 秒
13         }
14         qDebug() << "工作线程结束";
15     }
```

```
17
18  int main(int argc, char *argv[])
19  {
20      QCoreApplication app(argc, argv);
21
22      WorkerThread thread;
23      qDebug() << "主线程: 启动工作线程";
24      thread.start(); // 启动线程, 自动调用 run()
25
26      // 等待线程结束
27      thread.wait();
28      qDebug() << "主线程: 工作线程已结束";
29      app.exec();
30      std::cin.get();
31      return 0;
32  }
```

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

```
1  #include <QCoreApplication>
2  #include <QtConcurrent>
3  #include <QDebug>
4  #include <QThread>
5  #include <QVector>
6  #include <iostream>
7  // 一个耗时的计算函数
8  int slowSquare(int x)
9  {
10     QThread::sleep(1); // 模拟耗时操作
11     qDebug() << "计算" << x << "的平方, 线程 ID:" <<
        ↳ QThread::currentThreadId();
12     return x * x;
13 }
14
15 void concurrentExample()
16 {
17     QVector<int> numbers = {1, 2, 3, 4, 5};
18     qDebug() << "主线程 ID:" <<
        ↳ QThread::currentThreadId();
19
```

```
20     // 1. 并行计算每个元素的平方
21     QFuture<int> future =
        ↳ QtConcurrent::mapped(numbers, slowSquare);
22
23     // 2. 等待所有任务完成
24     future.waitForFinished();
25
26     // 3. 获取结果
27     QVector<int> results =
        ↳ future.results().toVector();
28     qDebug() << "所有平方结果:" << results;
29 }
30
31 int main(int argc, char *argv[])
32 {
33     QCoreApplication app(argc, argv);
34
35     concurrentExample();
36
37     std::cin.get();
38     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)

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QPainter>
4  #include <QPen>
5  #include <QBrush>
6  #include <QLinearGradient>
7  #include <QRadialGradient>
8  #include <QConicalGradient>
9  #include <QPixmap>
10 #include <QFont>
11 #include <QPainterPath>
12
13 class PaintWidget : public QWidget {
14 protected:
15     void paintEvent(QPaintEvent *event) override {
16         QPainter painter(this);
17         // 1. 设置抗锯齿和高质量渲染
18         painter.setRenderHint(QPainter::Antialiasing, true);
19         painter.setRenderHint(QPainter::TextAntialiasing,
20                               ↪ true);
```

```
20     painter.setRenderHint(QPainter::SmoothPixmapTransform,
21                           ↪ true);
22     // 2. 绘制不同颜色和线宽的直线
23     QPen pen1(Qt::red, 3, Qt::SolidLine,
24              ↪ Qt::RoundCap, Qt::RoundJoin);
25     painter.setPen(pen1);
26     painter.drawLine(20, 20, 180, 20);
27     // 3. 绘制带虚线的矩形
28     QPen pen2(Qt::blue, 2, Qt::DashLine);
29     painter.setPen(pen2);
30     painter.drawRect(20, 40, 80, 60);
31     // 4. 绘制填充线性渐变的椭圆
32     QLinearGradient linearGrad(60, 120, 140,
33                               ↪ 180);
34     linearGrad.setColorAt(0, Qt::yellow);
35     linearGrad.setColorAt(1, Qt::green);
36     painter.setPen(Qt::NoPen);
37     painter.setBrush(QBrush(linearGrad));
38     painter.drawEllipse(100, 40, 80, 60);
```

QPainter 使用示例 (二)

```
36 // 5. 绘制填充径向渐变的圆 (带阴影效果)
37 QRadialGradient radialGrad(60, 160, 30, 60,
    ↪ 160);
38 radialGrad.setColorAt(0, Qt::white);
39 radialGrad.setColorAt(1, Qt::darkGray);
40 painter.setBrush(QBrush(radialGrad));
41 painter.drawEllipse(40, 140, 40, 40);
42 // 6. 绘制锥形渐变的扇形
43 QConicalGradient conicalGrad(180, 180, 0);
44 conicalGrad.setColorAt(0, Qt::red);
45 conicalGrad.setColorAt(0.5, Qt::yellow);
46 conicalGrad.setColorAt(1, Qt::red);
47 painter.setBrush(QBrush(conicalGrad));
48 painter.setPen(QPen(Qt::darkRed, 2));
49 QRectF arcRect(160, 160, 40, 40);
50 painter.drawPie(arcRect, 30 * 16, 120 * 16);
51 // 7. 绘制多边形
52 painter.setPen(QPen(Qt::darkMagenta, 2));
53 painter.setBrush(QBrush(Qt::cyan,
    ↪ Qt::Dense4Pattern));
54 QPoint points[4] = {QPoint(120, 120),
    ↪ QPoint(180, 120), QPoint(170, 180),
    ↪ QPoint(130, 180)};
```

```
55 painter.drawPolygon(points, 4);
56 // 8. 绘制贝塞尔曲线
57 painter.setPen(QPen(Qt::darkBlue, 2,
    ↪ Qt::DashDotLine));
58 QPainterPath path;
59 path.moveTo(20, 120);
60 path.cubicTo(60, 80, 100, 160, 180, 100);
61 painter.drawPath(path);
62 // 9. 绘制带背景色的文本
63 painter.setPen(Qt::black);
64 painter.setFont(QFont("微软雅黑", 14,
    ↪ QFont::Bold));
65 painter.setBrush(QBrush(Qt::lightGray));
66 painter.drawRect(20, 200, 160, 30);
67 painter.drawText(25, 222, "QPainter 功能演
    ↪ 示");
68 // 10. 绘制图片 (假设有一张图片在同目录下)
69 QPixmap pix("qt_logo.png");
70 if (!pix.isNull())
71     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 };
```

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 总结

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)

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QTimer>
4  #include <QDebug>
5  #include <QKeyEvent>
6  #include <QMouseEvent>
7  #include <QMimeData>
8  #include <QDragEnterEvent>
9  #include <QDropEvent>
10
11 class WidgetDemo : public QWidget
12 {
13 public:
14     WidgetDemo(QWidget *parent = nullptr) :
15         QWidget(parent) {
16         // 初始化设置
17         setWindowTitle("QWidget 示例演示"); // 设置窗
18         ↳ 口标题
```

```
17         resize(400, 300); // 设置窗口大小
18         setMinimumSize(200, 150); // 设置窗口最小大小
19         setMaximumSize(800, 600); // 设置窗口最大大小
20         // 窗口属性设置
21         setWindowFlags(Qt::Window
22         ↳ Qt::WindowCloseButtonHint
23         ↳ Qt::WindowMinimizeButtonHint); // 设置窗
24         ↳ 口标志
25         setStyleSheet("background-color:
26         ↳ lightgray;"); // 设置窗口样式
27         setMouseTracking(true); // 设置鼠标跟踪
28         setCursor(Qt::ArrowCursor); // 设置鼠标形状
29         setAcceptDrops(true); // 设置窗口接受拖放
30         setAutoFillBackground(true); // 设置窗口自动
31         ↳ 填充背景
32         setFocusPolicy(Qt::StrongFocus); // 设置窗口
33         ↳ 焦点策略
34         m_timerId = startTimer(1000); // 启动定时器,
35         ↳ 每秒触发一次
36         m_step = 10; // 设置步长
37         m_colorIndex = 0; // 设置颜色索引
38     }
```

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: " +
38             ↪ m_colors[m_colorIndex] + ";");
39         m_colorIndex = (m_colorIndex + 1) % 3;
40     }
41     // 鼠标点击事件
42     void mousePressEvent(QMouseEvent *event) override
43     ↪ {
44         qDebug() << "鼠标点击位置:" << event->pos();
45         m_colorIndex = (m_colorIndex + 1) % 3;
46     }
47     // 按键事件
48     void keyPressEvent(QKeyEvent *event) override {
49         qDebug() << "按键事件:" << event->text();
50         const int step = 10;
51         switch(event->key()) {
52             case Qt::Key_Up:
53                 move(pos() - QPoint(0, step)); // 上
54                 ↪ 移
```

```
55             break;
56             case Qt::Key_Down:
57                 move(pos() + QPoint(0, step)); // 下
58                 ↪ 移
59             break;
60             case Qt::Key_Left:
61                 move(pos() - QPoint(step, 0)); // 左
62                 ↪ 移
63             break;
64             case Qt::Key_Right:
65                 move(pos() + QPoint(step, 0)); // 右
66                 ↪ 移
67             break;
68             case Qt::Key_Space:
69                 resize(size() + QSize(step, step));
70                 ↪ // 放大
71             break;
72             case Qt::Key_Control:
73                 resize(size() - QSize(step, step));
74                 ↪ // 缩小
75             break;
76             default:
```

QWidget 使用示例 (三)

```
69         QWidget::keyPressEvent(event);
70     }
71 }
72 // 拖放事件
73 void dragEnterEvent(QDragEnterEvent *event)
74 ↪ override {
75     qDebug() << "拖放事件:" <<
76     ↪ event->mimeTypeData()->text();
77 }
78 // 拖放事件
79 void dropEvent(QDropEvent *event) override {
80     qDebug() << "拖放事件:" <<
81     ↪ event->mimeTypeData()->text();
82 }
83 private:
84     int m_timerId; // 用于存储定时器 ID, 以便后续停止
85     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",
86     ↪ "green"}; // 用于存储颜色
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 =
98     ↪ mainWindow1.mapToGlobal(QPoint(100, 100));
99     QPoint localPos1 =
100     ↪ 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

```
1  #include <QApplication>
2  #include <QDialog>
3  #include <QPushButton>
4  #include <QVBoxLayout>
5  #include <QLabel>
6  #include <QDebug>
7  #include <iostream>
8  class MyDialog : public QDialog
9  {
10     Q_OBJECT
11 public:
12     MyDialog(QWidget *parent = nullptr) :
13         QDialog(parent)
14     {
15         setWindowTitle("QDialog 示例");
16         resize(300, 150);
17         QVBoxLayout *layout = new QVBoxLayout(this);
18         QLabel *label = new QLabel("这是一个自定义
19         ↳ QDialog 对话框", this);
20         layout->addWidget(label);
21         QPushButton *okBtn = new QPushButton("确定",
22         ↳ this);
23         QPushButton *cancelBtn = new QPushButton("取
24         ↳ 消", this);
```

```
21         layout->addWidget(okBtn);
22         layout->addWidget(cancelBtn);
23         connect(okBtn, &QPushButton::clicked, this,
24         ↳ &QDialog::accept);
25         connect(cancelBtn, &QPushButton::clicked,
26         ↳ this, &QDialog::reject);
27     }
28 };
29
30 int main(int argc, char *argv[])
31 {
32     QApplication app(argc, argv);
33     MyDialog dlg;
34     int ret = dlg.exec(); // 模态对话框
35     if (ret == QDialog::Accepted) {
36         qDebug() << "用户点击了确定";
37     } else {
38         qDebug() << "用户点击了取消";
39     }
40     std::cin.get();
41     return 0;
42 }
```

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

```
1  #include <QApplication>
2  #include <QMainWindow>
3  #include <QMenuBar>
4  #include <QMenu>
5  #include <QAction>
6  #include <QStatusBar>
7  #include <QToolBar>
8  #include <QLabel>
9  #include <QDebug>
10
11 class MyMainWindow : public QMainWindow
12 {
13     Q_OBJECT
14 public:
15     MyMainWindow(QWidget *parent = nullptr) :
        ↳ QMainWindow(parent)
```

```
17     setWindowTitle("QMainWindow 示例");
18     resize(600, 400);
19
20     // 创建菜单栏
21     QMenuBar *menuBar = new QMenuBar(this);
22     setMenuBar(menuBar);
23
24     // 文件菜单
25     QMenu *fileMenu = menuBar->addMenu("文件");
26     QAction *openAction = fileMenu->addAction("打
        ↳ 开");
27     QAction *exitAction = fileMenu->addAction("退
        ↳ 出");
28
29     // 编辑菜单
30     QMenu *editMenu = menuBar->addMenu("编辑");
31     QAction *copyAction = editMenu->addAction("复
        ↳ 制");
32     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

```
1  #include <QApplication>
2  #include <QFrame>
3  #include <QDebug>
4  class FrameDemo : public QFrame
5  {
6  public:
7      FrameDemo(QWidget *parent = nullptr,
8          ↪ Qt::WindowFlags f = Qt::WindowFlags())
9          : QFrame(parent, f)
10     {
11         setWindowTitle("QFrame 演示");
12         resize(400, 250);
13         setFrameShape(QFrame::Box); // 设置边框样式
14         // QFrame::Box 边框样式
15         // QFrame::NoFrame 无边框
16         // QFrame::Panel 面板
17         // QFrame::StyledPanel 样式化面板
18         // QFrame::HLine 水平线
```

```
18         // QFrame::VLine 垂直线
19         // QFrame::WinPanel Windows 2000 风格面板
20
21         setFrameShadow(QFrame::Raised); // 设置边框阴影
22         ↪ 影
23         // QFrame::Plain 无阴影
24         // QFrame::Raised 凸起
25         // QFrame::Sunken 凹陷
26
27         setLineWidth(3); // 设置边框宽度
28         setMidLineWidth(1); // 设置中间线宽度
29         qDebug() << "frameRect()" << frameRect();
30         qDebug() << "frameShadow()" <<
31         ↪ frameShadow();
32         qDebug() << "frameShape()" << frameShape();
33         qDebug() << "frameWidth()" << frameWidth();
34         qDebug() << "lineWidth()" << lineWidth();
35         qDebug() << "midLineWidth()" <<
36         ↪ midLineWidth();
37     }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QPushButton>
4  #include <QLabel>
5  #include <QHBoxLayout>
6  #include <QIcon>
7  class QPushButtonDemo : public QWidget
8  {
9  public:
10     QPushButtonDemo(QWidget *parent = nullptr) :
        ↳ QWidget(parent)
11     {
12         setTitle("QPushButton 示例");
13         resize(480, 180);
14         QLabel *label = new QLabel("请选择按钮操作",
        ↳ this);
15         label->setAlignment(Qt::AlignCenter);
16         QPushButton *btnNormal = new QPushButton("普
        ↳ 通按钮", this);
17         QPushButton *btnIcon = new
        ↳ QPushButton(QIcon::fromTheme("document-open"),
        ↳ "带图标按钮", this);
```

```
18         QPushButton *btnCheck = new QPushButton("可切
        ↳ 换按钮", this);
19         btnCheck->setCheckable(true);
20         connect(btnNormal, &QPushButton::clicked,
        ↳ this,
21             [=]() { label->setText("点击了普通按钮");
        ↳ });
22         connect(btnIcon, &QPushButton::clicked, this,
23             [=]() { label->setText("点击了带图标按钮");
        ↳ });
24         connect(btnCheck, &QPushButton::toggled,
        ↳ this, [=](bool checked) {
25             label->setText(checked ? "切换按钮：已按
        ↳ 下" : "切换按钮：未按下");
26         });
27         QHBoxLayout *layout = new QHBoxLayout(this);
28         layout->addWidget(btnNormal);
29         layout->addWidget(btnIcon);
30         layout->addWidget(btnCheck);
31         layout->addWidget(label, 1);
32         setLayout(layout);
33     }
34 }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QLabel>
4  #include <QVBoxLayout>
5
6  class LabelDemo : public QWidget
7  {
8  public:
9      LabelDemo(QWidget *parent = nullptr) :
10         ↳ QWidget(parent)
11         {
12             setWindowTitle("QLabel 示例");
13             resize(400, 250);
14
15             QLabel *label1 = new QLabel("普通文本标签",
16                 ↳ this);
17             label1->setAlignment(Qt::AlignCenter);
```

```
16
17         QLabel *label2 = new QLabel("<font
18         ↳ color='blue'> 富文本标签 </font>", this);
19         label2->setAlignment(Qt::AlignCenter);
20
21         QLabel *label3 = new QLabel("<a
22         ↳ href='https://www.qt.io/'>Qt 官网 </a>",
23         ↳ this);
24         label3->setAlignment(Qt::AlignCenter);
25         label3->setOpenExternalLinks(true);
26
27         QVBoxLayout *vbox = new QVBoxLayout;
28         vbox->addWidget(label1);
29         vbox->addWidget(label2);
30         vbox->addWidget(label3);
31         setLayout(vbox);
32     }
33 };
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QLineEdit>
4  #include <QLabel>
5  #include <QVBoxLayout>
6  #include <QHBoxLayout>
7  class QLineEditDemo : public QWidget
8  {
9  public:
10     QLineEditDemo(QWidget *parent = nullptr) :
        QWidget(parent)
11     {
12         setWindowTitle("QLineEdit 示例");
13         resize(400, 180);
14         QLabel *label = new QLabel("请输入内容",
        ↳ this);
15         label->setAlignment(Qt::AlignCenter);
16         QLineEdit *lineEdit1 = new QLineEdit(this);
17         lineEdit1->setPlaceholderText("普通输入框");
18         QLineEdit *lineEdit2 = new QLineEdit(this);
19         lineEdit2->setPlaceholderText("只读输入框");
```

```
20         lineEdit2->setText("只读内容");
21         lineEdit2->setReadOnly(true);
22         QLineEdit *lineEdit3 = new QLineEdit(this);
23         lineEdit3->setPlaceholderText("密码输入框");
24         lineEdit3->setEchoMode(QLineEdit::Password);
25         // 信号槽连接
26         connect(lineEdit1, &QLineEdit::textChanged,
27                 this, [=](const QString &text){
28             label->setText("输入内容: " + text); });
29         connect(lineEdit3, &QLineEdit::returnPressed,
30                 this, [=]() {label->setText("密码已输入"); });
31         QVBoxLayout *vbox = new QVBoxLayout;
32         vbox->addWidget(lineEdit1);
33         vbox->addWidget(lineEdit2);
34         vbox->addWidget(lineEdit3);
35         vbox->addWidget(label, 1);
36         setLayout(vbox);
37     }
38 }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QTextEdit>
4  #include <QLabel>
5  #include <QVBoxLayout>
6  #include <QPushButton>
7
8  class TextEditDemo : public QWidget
9  {
10 public:
11     TextEditDemo(QWidget *parent = nullptr) :
12         QWidget(parent)
13     {
14         setWindowTitle("QTextEdit 示例");
15         resize(400, 220);
16         auto *label = new QLabel("请输入多行文本",
17             ↪ this);
18         label->setAlignment(Qt::AlignCenter);
19         auto *textEdit = new QTextEdit(this);
20         textEdit->setPlaceholderText("支持多行文本和富
21             ↪ 文本 (HTML) ...");
```

```
22         auto *plainBtn = new QPushButton("显示纯文本",
23             ↪ this);
24         auto *htmlBtn = new QPushButton("显示富文本",
25             ↪ this);
26         connect(plainBtn, &QPushButton::clicked,
27             this, [=]() {
28                 label->setText(textEdit->toPlainText());
29             });
30         connect(htmlBtn, &QPushButton::clicked,
31             this, [=]() {
32                 label->setText(textEdit->toHtml());
33             });
34         auto *btnLayout = new QHBoxLayout;
35         btnLayout->addWidget(plainBtn);
36         btnLayout->addWidget(htmlBtn);
37         auto *mainLayout = new QVBoxLayout(this);
38         mainLayout->addWidget(textEdit);
39         mainLayout->addLayout(btnLayout);
40         mainLayout->addWidget(label);
41         setLayout(mainLayout);
42     }
43 }
```

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

```
1  #include <QApplication>
2  #include <QScrollArea>
3  #include <QLabel>
4  #include <QVBoxLayout>
5  #include <QWidget>
6  #include <QPalette>
7
8  int main(int argc, char *argv[])
9  {
10     QApplication app(argc, argv);
11     // 创建内容部件, 并设置合适的背景色以提升可读性
12     QWidget *contentWidget = new QWidget;
13     QVBoxLayout *layout = new QVBoxLayout;
14     // 批量添加标签, 内容更丰富
15     for (int i = 1; i <= 30; ++i) {
16         QLabel *label = new QLabel(QString("这是第 %1
17             ↪ 行内容 - QScrollArea 演示").arg(i));
18         label->setMargin(4);
19         layout->addWidget(label);
20     }
21     layout->addStretch(); // 增加弹性空间, 优化滚动体验
22     contentWidget->setLayout(layout);
```

```
23     // 设置内容部件的最小宽度, 避免内容过窄
24     contentWidget->setMinimumWidth(220);
25     // 创建 QScrollArea 并设置内容部件
26     QScrollArea *scrollArea = new QScrollArea;
27     scrollArea->setWindowTitle("QScrollArea 示例");
28     scrollArea->resize(350, 250);
29     scrollArea->setWidget(contentWidget);
30     scrollArea->setWidgetResizable(true); // 内容自适应
31     ↪ 应滚动区域大小
32     // 优化滚动条显示策略
33     scrollArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
34     scrollArea->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
35     // 设置滚动区域背景色
36     QPalette pal = scrollArea->palette();
37     pal.setColor(QPalette::Window, QColor(245, 245,
38         ↪ 250));
39     scrollArea->setAutoFillBackground(true);
40     scrollArea->setPalette(pal);
41     scrollArea->show();
42     return app.exec();
43 }
```

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

```
1  #include <QApplication>
2  #include <QSplitter>
3  #include <QTextEdit>
4  #include <QLabel>
5  #include <QFrame>
6  #include <QVBoxLayout>
7  #include <QWidget>
8
9  int main(int argc, char *argv[])
10 {
11     QApplication app(argc, argv);
12     // 创建主窗口部件
13     QWidget window;
14     window.setWindowTitle("QSplitter 示例");
15     window.resize(500, 300);
```

```
16     // 创建水平分割器
17     QSplitter *splitter = new
    ↪ QSplitter(Qt::Horizontal, &window);
18     // 左侧控件: QTextEdit
19     QTextEdit *editor = new QTextEdit;
20     editor->setPlainText("左侧: QTextEdit\n可以输入多行
    ↪ 文本。");
21     // 右侧控件: QFrame + QLabel
22     QFrame *frame = new QFrame;
23     frame->setFrameShape(QFrame::StyledPanel);
24     QVBoxLayout *frameLayout = new
    ↪ QVBoxLayout(frame);
25     QLabel *label = new QLabel("右侧: QLabel\n可放置任
    ↪ 意控件。");
26     label->setAlignment(Qt::AlignCenter);
27     frameLayout->addWidget(label);
28     // 添加控件到分割器
29     splitter->addWidget(editor);
30     splitter->addWidget(frame);
31     // 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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QListWidget>
4  #include <QVBoxLayout>
5  #include <QLabel>
6
7  class ListWidgetDemo : public QWidget
8  {
9  public:
10     ListWidgetDemo(QWidget *parent = nullptr) :
11         ↳ QWidget(parent)
12     {
13         setWindowTitle("QListWidget 示例");
14         resize(400, 250);
15
16         auto *label = new QLabel("请选择一个项目",
17         ↳ this);
```

```
16         label->setAlignment(Qt::AlignCenter);
17
18         auto *listWidget = new QListWidget(this);
19         listWidget->addItem({"苹果", "香蕉", "橙子",
20         ↳ "葡萄", "西瓜"});
21
22         connect(listWidget,
23         ↳ &QListWidget::currentTextChanged, this,
24         ↳ [=](const QString &text){
25             label->setText("已选择: " + text);
26         });
27
28         auto *mainLayout = new QVBoxLayout(this);
29         mainLayout->addWidget(listWidget);
30         mainLayout->addWidget(label);
31         setLayout(mainLayout);
32     }
33 }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QTreeWidget>
4  #include <QVBoxLayout>
5  #include <QLabel>
6
7  class TreeWidgetDemo : public QWidget
8  {
9  public:
10     TreeWidgetDemo(QWidget *parent = nullptr) :
11         QWidget(parent)
12     {
13         setWindowTitle("QTreeWidget 示例");
14         resize(420, 300);
15         auto *label = new QLabel("请选择节点", this);
16         label->setAlignment(Qt::AlignCenter);
17         auto *tree = new QTreeWidget(this);
18         tree->setColumnCount(2);
19         tree->setHeaderLabels(QStringList() << "名称"
20             << "描述");
21         auto *root1 = new QTreeWidgetItem(tree,
22             QStringList() << "水果" << "各种水果");
```

```
20     new QTreeWidgetItem(root1, QStringList() <<
21         << "苹果" << "红色或绿色");
22     new QTreeWidgetItem(root1, QStringList() <<
23         << "香蕉" << "黄色, 富含钾");
24     auto *root2 = new QTreeWidgetItem(tree,
25         QStringList() << "蔬菜" << "各种蔬菜");
26     new QTreeWidgetItem(root2, QStringList() <<
27         << "西红柿" << "红色, 富含维生素 C");
28     new QTreeWidgetItem(root2, QStringList() <<
29         << "黄瓜" << "绿色, 清爽可口");
30     tree->expandAll();
31     connect(tree, &QTreeWidget::itemClicked,
32         << this, [=](QTreeWidgetItem *item, int){
33             label->setText(QString("已选择: %1 -
34                 %2").arg(item->text(0),
35                     item->text(1)));
36         });
37     auto *vbox = new QVBoxLayout(this);
38     vbox->addWidget(tree);
39     vbox->addWidget(label, 1);
40     setLayout(vbox);
41 }
```

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

```
1  #include <QApplication>
2  #include <QTabWidget>
3  #include <QWidget>
4  #include <QLabel>
5  #include <QVBoxLayout>
6  #include <QTextEdit>
7  int main(int argc, char *argv[])
8  {
9      QApplication app(argc, argv);
10     // 创建主窗口部件
11     QWidget window;
12     window.setWindowTitle("QTabWidget 示例");
13     window.resize(420, 260);
14     // 创建 QTabWidget
15     QTabWidget *tabWidget = new QTabWidget(&window);
16     // 第一个选项卡: 简单标签
17     QWidget *tab1 = new QWidget;
18     QVBoxLayout *tab1Layout = new QVBoxLayout(tab1);
19     QLabel *label1 = new QLabel("这是第一个选项卡内容。
    ↪ ");
20     tab1Layout->addWidget(label1);
```

```
21     tab1Layout->addStretch();
22     // 第二个选项卡: 文本编辑器
23     QWidget *tab2 = new QWidget;
24     QVBoxLayout *tab2Layout = new QVBoxLayout(tab2);
25     QTextEdit *textEdit = new QTextEdit;
26     textEdit->setPlainText("可以在这里输入多行文本。");
27     tab2Layout->addWidget(textEdit);
28     // 添加选项卡到 QTabWidget
29     tabWidget->addTab(tab1, "选项卡一");
30     tabWidget->addTab(tab2, "选项卡二");
31     // 设置选项卡可关闭
32     tabWidget->setTabsClosable(true);
33     QObject::connect(tabWidget,
    ↪ &QTabWidget::tabCloseRequested, tabWidget,
    ↪ &QTabWidget::removeTab);
34     // 设置选项卡位置 (可选: 顶部、底部、左侧、右侧)
35     tabWidget->setTabPosition(QTabWidget::North);
36     // 主布局
37     QVBoxLayout *mainLayout = new
    ↪ QVBoxLayout(&window);
38     mainLayout->addWidget(tabWidget);
39     window.show();
40     return app.exec();
41 }
```

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

```
1  #include <QApplication>
2  #include <QStackedWidget>
3  #include <QPushButton>
4  #include <QVBoxLayout>
5  #include <QLabel>
6  #include <QWidget>
7
8  int main(int argc, char *argv[])
9  {
10     QApplication app(argc, argv);
11     QWidget window;
12     window.setWindowTitle("QStackedWidget 示例");
13     window.resize(400, 200);
14
15     QStackedWidget *stack = new
↪     QStackedWidget(&window);
16     stack->addWidget(new QLabel("这是第一个页面"));
17     stack->addWidget(new QLabel("这是第二个页面"));
18
19     QPushButton *prev = new QPushButton("上一页");
```

```
20     QPushButton *next = new QPushButton("下一页");
21
22     QObject::connect(prev, &QPushButton::clicked,
↪     [&]() {
23         int idx = stack->currentIndex();
24         if (idx > 0) stack->setCurrentIndex(idx - 1);
25     });
26     QObject::connect(next, &QPushButton::clicked,
↪     [&]() {
27         int idx = stack->currentIndex();
28         if (idx < stack->count() - 1)
↪         stack->setCurrentIndex(idx + 1);
29     });
30
31     QVBoxLayout *layout = new QVBoxLayout(&window);
32     layout->addWidget(stack);
33     layout->addStretch();
34     layout->addWidget(prev);
35     layout->addWidget(next);
36     window.show();
37     return app.exec();
38 }
```

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

```
1  #include <QApplication>
2  #include <QMainWindow>
3  #include <QToolBar>
4  #include <QAction>
5  #include <QLabel>
6
7  class ToolBarDemo : public QMainWindow
8  {
9  public:
10     ToolBarDemo(QWidget *parent = nullptr) :
11         QMainWindow(parent)
12     {
13         setWindowTitle("QToolBar 示例");
14         resize(500, 300);
15
16         // 简单工具栏
17         QToolBar *toolBar = addToolBar("工具栏");
18         QAction *openAction = toolBar->addAction("打
19         ↪ 开");
20         QAction *saveAction = toolBar->addAction("保
21         ↪ 存");
```

```
19
20         // 中央窗口
21         QLabel *label = new QLabel("中央窗口区域",
22         ↪ this);
23         label->setAlignment(Qt::AlignCenter);
24         setCentralWidget(label);
25
26         // 信号槽
27         connect(openAction, &QAction::triggered,
28         ↪ this, [=]() {
29             label->setText("点击了打开");
30         });
31         connect(saveAction, &QAction::triggered,
32         ↪ this, [=]() {
33             label->setText("点击了保存");
34         });
35     }
36 };
```

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

```
1  #include <QApplication>
2  #include <QMainWindow>
3  #include <QStatusBar>
4  #include <QPushButton>
5  #include <QLabel>
6  #include <QTimer>
7
8  class StatusBarDemo : public QMainWindow
9  {
10 public:
11     StatusBarDemo(QWidget *parent = nullptr) :
12         QMainWindow(parent)
13     {
14         setWindowTitle("QStatusBar 示例");
15         resize(480, 220);
16         // 创建状态栏
17         QStatusBar *statusBar = new QStatusBar(this);
18         setStatusBar(statusBar);
19         // 显示初始信息
20         statusBar->showMessage("欢迎使用 QStatusBar 示
21         例", 3000);
```

```
22         // 添加永久信息标签
23         QLabel *permanentLabel = new QLabel("就绪",
24         this);
25
26         statusBar->addPermanentWidget(permanentLabel);
27         // 添加临时信息按钮
28         QPushButton *infoBtn = new QPushButton("显示临
29         时信息", this);
30         statusBar->addWidget(infoBtn);
31         // 信号槽：点击按钮显示临时信息
32         connect(infoBtn, &QPushButton::clicked, this,
33         [=]() {
34             statusBar->showMessage("这是临时信息, 3 秒
35             后消失", 3000);
36         });
37         // 中央窗口区域
38         QLabel *centerLabel = new QLabel("主窗口内容区
39         域", this);
40         centerLabel->setAlignment(Qt::AlignCenter);
41         setCentralWidget(centerLabel);
42     }
43 }
```

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

```
1  #include <QApplication>
2  #include <QMainWindow>
3  #include <QMenuBar>
4  #include <QMenu>
5  #include <QAction>
6  #include <QLabel>
7  #include <QMessageBox>
8  class MenuDemo : public QMainWindow
9  {
10 public:
11     MenuDemo(QWidget *parent = nullptr) :
12         QMainWindow(parent)
13     {
14         setWindowTitle("QMenu 示例");
15         resize(480, 260);
16         // 创建菜单栏
17         QMenuBar *menuBar = new QMenuBar(this);
18         setMenuBar(menuBar);
19         // 文件菜单
20         QMenu *fileMenu = menuBar->addMenu("文件
    ↳ (&F)");
```

```
20         QAction *newAction = fileMenu->addAction("新
    ↳ 建 (&N)");
21         fileMenu->addSeparator();
22         QAction *exitAction = fileMenu->addAction("退
    ↳ 出 (&Q)");
23         // 编辑菜单
24         QMenu *editMenu = menuBar->addMenu("编辑
    ↳ (&E)");
25         QAction *copyAction = editMenu->addAction("复
    ↳ 制 (&C)");
26         // 中央窗口
27         QLabel *label = new QLabel("请选择菜单项进行操
    ↳ 作", this);
28         label->setAlignment(Qt::AlignCenter);
29         setCentralWidget(label);
30         // 信号槽连接
31         connect(newAction, &QAction::triggered,
32             this, [=]() { label->setText("新建操作"); });
33         connect(exitAction, &QAction::triggered,
34             this, [=]() { close(); });
35         connect(copyAction, &QAction::triggered,
36             this, [=]() { label->setText("复制操作"); });
37     }
38 }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QToolButton>
4  #include <QMenu>
5  #include <QHBoxLayout>
6  #include <QLabel>
7  class ToolButtonDemo : public QWidget
8  {
9  public:
10 ToolButtonDemo(QWidget *parent = nullptr) :
    ↳ QWidget(parent)
11     {
12         setWindowTitle("QToolButton 示例");
13         resize(480, 180);
14         QLabel *label = new QLabel("请选择工具按钮操
            ↳ 作", this);
15         label->setAlignment(Qt::AlignCenter);
16         QToolButton *toolBtn = new QToolButton(this);
17         toolBtn->setText("工具按钮");
18         toolBtn->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
19         toolBtn->setIcon(QIcon::fromTheme("document-open"));
```

```
20         toolBtn->setAutoRaise(true);
21         toolBtn->setCheckable(true);
22         QMenu *menu = new QMenu(this);
23         QAction *actionA = menu->addAction("操作 A");
24         QAction *actionB = menu->addAction("操作 B");
25         toolBtn->setMenu(menu);
26         toolBtn->setPopupMode(QToolButton::MenuButtonPopup);
27         connect(toolBtn, &QToolButton::clicked,
28                 this, [=](bool checked){
29             label->setText(checked ? "已按下" : "未按下");
            ↳ });
30         connect(actionA, &QAction::triggered, this,
31                 [=]() { label->setText("选择了操作 A"); });
32         connect(actionB, &QAction::triggered, this,
33                 [=]() { label->setText("选择了操作 B"); });
34         QHBoxLayout *layout = new QHBoxLayout(this);
35         layout->addWidget(toolBtn);
36         layout->addWidget(label, 1);
37         setLayout(layout);
38     }
39 };
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QRadioButton>
4  #include <QButtonGroup>
5  #include <QLabel>
6  #include <QVBoxLayout>
7  #include <QHBoxLayout>
8  class RadioButtonDemo : public QWidget
9  {
10 public:
11     RadioButtonDemo(QWidget *parent = nullptr) :
12         QWidget(parent)
13     {
14         setWindowTitle("QRadioButton 示例");
15         resize(400, 180);
16         QLabel *label = new QLabel("请选择一个选项",
17                                     ↪ this);
18         label->setAlignment(Qt::AlignCenter);
19         QRadioButton *radio1 = new QRadioButton("选项
20         ↪ 一", this);
21         QRadioButton *radio2 = new QRadioButton("选项
22         ↪ 二", this);
23         // 分组, 确保单选
```

```
24         QButtonGroup *group = new QButtonGroup(this);
25         group->addButton(radio1, 1);
26         group->addButton(radio2, 2);
27         // 信号槽连接
28         connect(radio1, &QRadioButton::toggled, this,
29                 ↪ [=](bool checked){
30                     if (checked) label->setText("已选择: 选项一");
31                 });
32         connect(radio2, &QRadioButton::toggled, this,
33                 ↪ [=](bool checked){
34                     if (checked) label->setText("已选择: 选项二");
35                 });
36         QVBoxLayout *vbox = new QVBoxLayout;
37         vbox->addWidget(radio1);
38         vbox->addWidget(radio2);
39         QHBoxLayout *hbox = new QHBoxLayout(this);
40         hbox->addLayout(vbox);
41         hbox->addWidget(label, 1);
42         setLayout(hbox);
43     }
44 }
```


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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QCheckBox>
4  #include <QLabel>
5  #include <QHBoxLayout>
6  #include <QVBoxLayout>
7  class CheckBoxDemo : public QWidget
8  {
9  public:
10     CheckBoxDemo(QWidget *parent = nullptr) :
        QWidget(parent)
11     {
12         setWindowTitle("QCheckBox 示例");
13         resize(400, 180);
14         QLabel *label = new QLabel("请选择复选框", this);
15         label->setAlignment(Qt::AlignCenter);
16         QCheckBox *check1 = new QCheckBox("选项一", this);
17         QCheckBox *check2 = new QCheckBox("选项二", this);
18         // 信号槽连接
19         auto updateLabel = [=]() {
```

```
20         QStringList checked;
21         if (check1->isChecked()) checked << "选项一";
22         if (check2->isChecked()) checked << "选项二";
23         if (checked.isEmpty())
24             label->setText("未选择任何选项");
25         else
26             label->setText("已选择: " +
        ↳ checked.join(", "));
27     };
28     connect(check1, &QCheckBox::toggled, this,
        ↳ updateLabel);
29     connect(check2, &QCheckBox::toggled, this,
        ↳ updateLabel);
30     QVBoxLayout *vbox = new QVBoxLayout;
31     vbox->addWidget(check1);
32     vbox->addWidget(check2);
33     QHBoxLayout *hbox = new QHBoxLayout(this);
34     hbox->addLayout(vbox);
35     hbox->addWidget(label, 1);
36     setLayout(hbox);
37 }
38 };
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QComboBox>
4  #include <QLabel>
5  #include <QHBoxLayout>
6  #include <QVBoxLayout>
7
8  class ComboBoxDemo : public QWidget
9  {
10 public:
11     ComboBoxDemo(QWidget *parent = nullptr) :
12         ↳ QWidget(parent)
13     {
14         setWindowTitle("QComboBox 示例");
15         resize(400, 180);
16         QLabel *label = new QLabel("请选择一个选项",
17         ↳ this);
```

```
16     label->setAlignment(Qt::AlignCenter);
17     QComboBox *combo = new QComboBox(this);
18     combo->addItem("选项一");
19     combo->addItem("选项二");
20     combo->addItem("选项三");
21     // 信号槽连接
22     connect(combo,
23         ↳ &QComboBox::currentTextChanged, this,
24         ↳ [=](const QString &text){
25             label->setText("已选择: " + text);
26         });
27     QVBoxLayout *vbox = new QVBoxLayout;
28     vbox->addWidget(combo);
29     vbox->addWidget(label, 1);
30     setLayout(vbox);
31 }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QSpinBox>
4  #include <QLabel>
5  #include <QHBoxLayout>
6  #include <QVBoxLayout>
7
8  class SpinBoxDemo : public QWidget
9  {
10 public:
11     SpinBoxDemo(QWidget *parent = nullptr) :
12         ↳ QWidget(parent)
13     {
14         setWindowTitle("QSpinBox 示例");
15         resize(400, 180);
16         QLabel *label = new QLabel("当前值: 0", this);
```

```
16         label->setAlignment(Qt::AlignCenter);
17         QSpinBox *spinBox = new QSpinBox(this);
18         spinBox->setRange(0, 100);
19         spinBox->setSingleStep(5);
20         spinBox->setValue(0);
21         spinBox->setSuffix(" 个");
22         // 信号槽连接
23         connect(spinBox,
24             ↳ QOverload<int>::of(&QSpinBox::valueChanged),
25             ↳ this, [=](int value){
26                 label->setText(QString("当前值:
27                     ↳ %1").arg(value));
28             });
29         QVBoxLayout *vbox = new QVBoxLayout;
30         vbox->addWidget(spinBox);
31         vbox->addWidget(label, 1);
32         setLayout(vbox);
33     }
34 };
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QDoubleSpinBox>
4  #include <QLabel>
5  #include <QHBoxLayout>
6  #include <QVBoxLayout>
7
8  class DoubleSpinBoxDemo : public QWidget
9  {
10 public:
11     DoubleSpinBoxDemo(QWidget *parent = nullptr) :
12         ↪ QWidget(parent)
13     {
14         setWindowTitle("QDoubleSpinBox 示例");
15         resize(400, 180);
16         QLabel *label = new QLabel("当前值: 0.00",
17         ↪ this);
18         label->setAlignment(Qt::AlignCenter);
19         QDoubleSpinBox *doubleSpinBox = new
20         ↪ QDoubleSpinBox(this);
```

```
18     doubleSpinBox->setRange(-100.0, 100.0);
19     doubleSpinBox->setSingleStep(0.5);
20     doubleSpinBox->setDecimals(2);
21     doubleSpinBox->setValue(0.0);
22     doubleSpinBox->setSuffix(" 元");
23     // 信号槽连接
24     connect(doubleSpinBox,
25
26         ↪ QOverload<double>::of(&QDoubleSpinBox::valueChanged),
27         this, [=](double value){
28             label->setText(QString("当前值:
29             ↪ %1").arg(value, 0, 'f', 2));
30         });
31     QVBoxLayout *vbox = new QVBoxLayout;
32     vbox->addWidget(doubleSpinBox);
33     vbox->addWidget(label, 1);
34     setLayout(vbox);
35 }
```


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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QCalendarWidget>
4  #include <QLabel>
5  #include <QVBoxLayout>
6  #include <QDate>
7
8  class CalendarWidgetDemo : public QWidget
9  {
10 public:
11     CalendarWidgetDemo(QWidget *parent = nullptr) :
12         ↪ QWidget(parent)
13     {
14         setWindowTitle("QCalendarWidget 示例");
15         resize(400, 250);
16
17         QLabel *label = new QLabel("请选择日期",
18         ↪ this);
19         label->setAlignment(Qt::AlignCenter);
20     }
```

```
19     QCalendarWidget *calendar = new
20     ↪ QCalendarWidget(this);
21     calendar->setGridVisible(true);
22     calendar->setFirstDayOfWeek(Qt::Monday);
23
24     ↪ calendar->setSelectedDate(QDate::currentDate());
25
26     // 信号槽连接
27     connect(calendar,
28     ↪ &QCalendarWidget::selectionChanged,
29     this, [=]() {
30         QDate date = calendar->selectedDate();
31         label->setText("已选择: " +
32         ↪ date.toString("yyyy-MM-dd"));
33     });
34
35     QVBoxLayout *vbox = new QVBoxLayout;
36     vbox->addWidget(calendar);
37     vbox->addWidget(label, 1);
38     setLayout(vbox);
39 }
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QSlider>
4  #include <QLabel>
5  #include <QVBoxLayout>
6  #include <QHBoxLayout>
7  class SliderDemo : public QWidget {
8  public:
9  SliderDemo(QWidget *parent = nullptr) :
10     QWidget(parent) {
11     setWindowTitle("QSlider 示例");
12     resize(400, 180);
13     auto *label = new QLabel("当前值: 50", this);
14     label->setAlignment(Qt::AlignCenter);
15     auto *hSlider = new QSlider(Qt::Horizontal,
16     this);
17     auto *vSlider = new QSlider(Qt::Vertical, this);
18     hSlider->setRange(0, 100); vSlider->setRange(0,
19     100);
20     hSlider->setValue(50); vSlider->setValue(50);
21     hSlider->setTickInterval(10);
22     vSlider->setTickInterval(10);
```

```
23     hSlider->setTickPosition(QSlider::TicksBelow);
24     vSlider->setTickPosition(QSlider::TicksLeft);
25     // 只需一个信号槽即可同步两个滑块
26     connect(hSlider, &QSlider::valueChanged, vSlider,
27     [=](&QSlider::setValue));
28     connect(vSlider, &QSlider::valueChanged, hSlider,
29     [=](&QSlider::setValue));
30     connect(hSlider, &QSlider::valueChanged, label,
31     [=](&QSlider::setValue){
32     label->setText(QString("当前值: %1").arg(value));
33     });
34     connect(vSlider, &QSlider::valueChanged, label,
35     [=](&QSlider::setValue){
36     label->setText(QString("当前值: %1").arg(value));
37     });
38     auto *hLayout = new QHBoxLayout;
39     hLayout->addWidget(vSlider);
40     hLayout->addWidget(hSlider, 1);
41     auto *mainLayout = new QVBoxLayout(this);
42     mainLayout->addLayout(hLayout);
43     mainLayout->addWidget(label);
44     setLayout(mainLayout);
45 };
```

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

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QScrollBar>
4  #include <QLabel>
5  #include <QVBoxLayout>
6  #include <QHBoxLayout>
7  class ScrollBarDemo : public QWidget {
8  public:
9      ScrollBarDemo(QWidget *parent = nullptr) :
10         ↳ QWidget(parent) {
11          setWindowTitle("QScrollBar 示例");
12          resize(400, 180);
13          auto *label = new QLabel("当前值: 50", this);
14          label->setAlignment(Qt::AlignCenter);
15          auto *hScrollBar = new QScrollBar(Qt::Horizontal,
16         ↳ this);
17          auto *vScrollBar = new QScrollBar(Qt::Vertical,
18         ↳ this);
19          hScrollBar->setRange(0, 100); vScrollBar->setRange(0,
20         ↳ 100);
21          hScrollBar->setValue(50); vScrollBar->setValue(50);
```

```
22          hScrollBar->setPageStep(10);
23          ↳ vScrollBar->setPageStep(10);
24          connect(hScrollBar, &QScrollBar::valueChanged,
25          ↳ vScrollBar, &QScrollBar::setValue);
26          connect(vScrollBar, &QScrollBar::valueChanged,
27          ↳ hScrollBar, &QScrollBar::setValue);
28          connect(hScrollBar, &QScrollBar::valueChanged, label,
29          ↳ [=](int v){ label->setText(QString("当前值:
30          ↳ %1").arg(v)); });
31          connect(vScrollBar, &QScrollBar::valueChanged, label,
32          ↳ [=](int v){ label->setText(QString("当前值:
33          ↳ %1").arg(v)); });
34          auto *hLayout = new QHBoxLayout;
35          hLayout->addWidget(vScrollBar);
36          hLayout->addWidget(hScrollBar, 1);
37          auto *mainLayout = new QVBoxLayout(this);
38          mainLayout->addLayout(hLayout);
39          mainLayout->addWidget(label);
40          setLayout(mainLayout);
41      }
42  };
```


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、VBoxLayout、GridLayout），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)

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QPushButton>
4  #include <QVBoxLayout>
5  #include <QLabel>
6
7  int main(int argc, char *argv[])
8  {
9      QApplication app(argc, argv);
10
11      QWidget window;
12      window.setWindowTitle("QSS 样式表示例");
13      // 创建控件
14      QLabel *label = new QLabel("欢迎使用 QSS 美化界
        ↳ 面");
15      QPushButton *btn1 = new QPushButton("普通按钮");
16      QPushButton *btn2 = new QPushButton("高亮按钮");
17      btn2->setObjectName("btn2"); // 优化: 提前设置
        ↳ objectName
18      // 设置布局
19      QVBoxLayout *layout = new QVBoxLayout;
```

```
20      layout->addWidget(label);
21      layout->addWidget(btn1);
22      layout->addWidget(btn2);
23      layout->setSpacing(16); // 优化: 增加控件间距
24      layout->setContentsMargins(24, 24, 24, 24); // 优
        ↳ 化: 设置窗口边距
25      window.setLayout(layout);
26      // 优化: QSS 样式结构更清晰, 注释更明确
27      QString qss = R"(
28          /* 窗口背景 */
29          QWidget {
30              background-color: #f0f0f0;
31          }
32          /* 标签样式 */
33          QLabel {
34              color: #2d8cf0;
35              font-size: 20px;
36              font-weight: bold;
37              padding: 8px;
38          }
39          /* 普通按钮样式 */
```

QSS 使用示例 (二)

```
40     QPushButton {
41         background-color: #ffffff;
42         border: 2px solid #2d8cf0;
43         border-radius: 8px;
44         color: #2d8cf0;
45         font-size: 16px;
46         padding: 6px 18px;
47         min-width: 100px;
48         min-height: 32px;
49         transition: all 0.2s;
50     }
51     QPushButton:hover {
52         background-color: #2d8cf0;
53         color: #ffffff;
54     }
55     QPushButton:pressed {
56         background-color: #1565c0;
57         color: #ffffff;
58     }
```

```
59     /* 高亮按钮样式 */
60     QPushButton#btn2 {
61         background-color: #ff9800;
62         color: #fff;
63         border: 2px solid #ff9800;
64     }
65     QPushButton#btn2:hover {
66         background-color: #ffa726;
67         border-color: #ffa726;
68     }
69     QPushButton#btn2:pressed {
70         background-color: #ef6c00;
71         border-color: #ef6c00;
72     }
73     )";
74     window.setStyleSheet(qss);
75     window.resize(360, 220); // 优化: 更合适的窗口大小
76     window.show();
77     return app.exec();
78 }
```

- 1 Qt 框架概述
- 2 Qt 核心模块
- 3 元对象系统
- 4 信号槽机制

- 5 事件系统
- 6 Qt 容器类
- 7 Qt 工具类
- 8 Qt-Widgets 编程
- 9 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。

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。