|  |
| --- |
| *class* MainWindow : *public* QMainWindow{  *public:*      explicit MainWindow(QWidget \**parent* = nullptr)      : QMainWindow(*parent*){  *auto* edit = new QTextEdit();          edit->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);          setCentralWidget(edit);      }  }; |

这里使用了namespace Qt。在QT中，这个命名空间用来存放QT中用到的枚举对象。这么做是因为QT中的类都是全局可见的，不需要显示的namespace。而QT中是没有枚举类这个特性，所以就采用了这样一种办法。因为，C++中枚举对象是全局可见的。

这里插播下enum。

传统 enum 的问题

在C++11之前，使用传统的enum时，枚举值会泄漏到外层作用域，容易导致命名冲突。例如：

|  |
| --- |
| enum Color { Red, Green, Blue };  enum TrafficLight { Red, Yellow, Green }; // 错误：Red 和 Green 重复定义 |

**​\*\*enum class 的改进\*\***

enum class解决了这个问题，它将枚举值的作用域限制在枚举类型内部，并且不会隐式转换为整数。例如：

|  |
| --- |
| #include <iostream>  // 定义枚举类  enum class Color { Red, Green, Blue };  enum class TrafficLight { Red, Yellow, Green };  int main() {  // 使用枚举类  Color c = Color::Red;  TrafficLight t = TrafficLight::Green;  // 输出枚举值  if (c == Color::Red) {  std::cout << "Color is Red" << std::endl;  }  if (t == TrafficLight::Green) {  std::cout << "Traffic light is Green" << std::endl;  }  // 不会隐式转换为整数  // int i = c; // 错误：不能隐式转换  int i = static\_cast<int>(c); // 需要显式转换  std::cout << "Color as int: " << i << std::endl;  return 0;  } |

**\*\*enum class 的特点\*\***

1. ​**作用域限制**：枚举值必须通过枚举类名::枚举值访问，不会泄漏到外层作用域。
2. ​**类型安全**：enum class不会隐式转换为整数，避免了意外的类型转换。
3. ​**底层类型支持**：可以指定枚举的底层类型（如int、char等），例如：

|  |
| --- |
| enum class Color : char { Red, Green, Blue }; |

好了，回到QT。

Qt::ScrollBarAsNeeded表示侧边滑块出现策略。

窗口中设置一个主控件，然后可以添加一个menubar，menubar里可以继续添加submenu，最后设置信号和槽函数。

|  |
| --- |
| *class* MainWindow : *public* QMainWindow{  *public:*      explicit MainWindow(QWidget \**parent* = nullptr)      : QMainWindow(*parent*){  *auto* edit = new QTextEdit();  *auto* menuBar = new QMenuBar;          setMenuBar(menuBar);          QMenu\* menu = menuBar->addMenu("File");          menu->addSeparator();          QAction\* open = menu->addAction("open");          connect(open, &QAction::triggered, this, &MainWindow::onOpen);          QAction\* save = menu->addAction("save");          QAction\* exit = menu->addAction("save as");            setCentralWidget(edit);      }  *void* onOpen(){          qDebug() << "open";      }  }; |

上面的代码还缺两个connect函数需要完成，但是不用这么做。QT提供了可以将信号和槽函数自动连接在一起的机制。

|  |
| --- |
| *class* MainWindow : *public* QMainWindow{      Q\_OBJECT  *public:*      explicit MainWindow(QWidget \**parent* = nullptr)      : QMainWindow(*parent*){  *auto* edit = new QTextEdit();  *auto* menuBar = new QMenuBar;          setMenuBar(menuBar);          QMenu\* menu = menuBar->addMenu("File");          QAction\* action = menu->addAction("open");          action->setObjectName("open");            setCentralWidget(edit);          QMetaObject::connectSlotsByName(this);      }  private Q\_SLOTS:  *void* on\_open\_triggered(){          qDebug() << "open";      }  }; |

Qt 的反射机制（Reflection Mechanism）是一种在运行时动态获取类信息（如类名、属性、方法等）的能力。Qt 通过其元对象系统（Meta-Object System）实现了反射机制，这是 Qt 框架的核心特性之一。

Qt 反射机制的核心组件有以下几个

​\*\*QMetaObject\*\*：

每个使用 Q\_OBJECT 宏的类都会生成一个对应的 QMetaObject 对象。

QMetaObject 包含了类的元信息，如类名、父类、信号、槽、属性等。

​\*\*QObject\*\*：

Qt 反射机制的基础类，所有支持反射的类都必须直接或间接继承自 QObject。

QObject 提供了 metaObject() 方法，用于获取类的 QMetaObject。

​\*\*Q\_PROPERTY 宏\*\*：

用于声明类的属性，属性可以通过反射机制访问和修改。

​\*\*Q\_INVOKABLE 宏\*\*：

用于标记类的成员函数，使其可以通过反射机制调用。

​\*\*QMetaMethod 和 QMetaProperty\*\*：

分别用于描述类的方法和属性。

要使用反射机制除了继承QObject，还要使用QOBJECT宏。这样，只要槽函数遵循规定的命名机制就可以完成反射。

槽函数命名需要知道对象名字和信号名字，对象默认的名字是个空字符串，使用这个通过继承获取的成员函数可以设置对象名字。至于信号，目前都是QT内置信号，后面会学习自定信号。

action->setObjectName("open");

关于槽的宏QSLOTS，是一个空的宏，因此前面必须带限定符，不然C++编译器展开宏的时候会报错。

关于QOBJECT宏，moc编译器会自动生成一个cpp文件，用来实现我们上面用到的反射机制。简单点来说，就是生成有on\_objectname\_signal组成的字符串，这个字符串对应的就是槽函数。Moc编译器生成的代码会帮我们把这些内容连接在一起。

最后，Cmake文件中引入的moc，就是完成这个事情的。

|  |
| --- |
| *class* MainWindow : *public* QMainWindow{      Q\_OBJECT  *public:*      explicit MainWindow(QWidget \**parent* = nullptr)      : QMainWindow(*parent*){          edit = new QTextEdit();  *auto* menuBar = new QMenuBar;          setMenuBar(menuBar);          QMenu\* menu = menuBar->addMenu("File");          QAction\* action = menu->addAction("open");          action->setObjectName("open");          menu = menuBar->addMenu("Edit");          action = menu->addAction("自动换行");          action->setCheckable(true);          action->setChecked(true);          action->setObjectName("autoWrap");            setCentralWidget(edit);          QMetaObject::connectSlotsByName(this);      }  private Q\_SLOTS:  *void* on\_open\_triggered(){          qDebug() << "open";      }      //信号的参数类型必须和槽的参数类型一致，而且个数也要一致（可以多，不能少）  *void* on\_autoWrap\_triggered(*bool* *checked*){          qDebug() << "autoWrap" << *checked*;          //枚举对象在QTextEdit中,设置按照宽度自动换行          edit->setLineWrapMode(*checked* ? QTextEdit::WidgetWidth : QTextEdit::NoWrap);      }  *private:*      QTextEdit\* edit;  }; |

QT的控件也可以设置状态，上面这段代码就是示例。这次槽函数接受一个参数，这个参数是信号触发后被传递过来的。在QT中，信号传递的参数可以被槽函数接受但是有限制条件，注释有说明。为了演示这个例子，需要保存一个数据成员以便在槽函数中使用它。

接下来，我们看如何持久化保存这些状态，这需要用到QSeetings。

提到QSettings，就必须要说下QVariant。

QVariant 是 Qt 框架中一个非常重要的类，用于存储和操作多种类型的数据。它的主要作用是提供一个通用的容器，可以存储几乎任何类型的值，并在需要时将其转换为特定的类型。QVariant 的核心优势在于它的灵活性和类型安全性。

**\*\***QVariant**的主要特点\*\***

1. ​**通用容器**：  
   QVariant 可以存储多种类型的值，包括基本类型（如 int、double、bool）、Qt 类型（如 QString、QDateTime、QSize）以及自定义类型。
2. ​**类型安全**：  
   QVariant 在存储值时同时记录其类型信息，可以在运行时安全地转换为原始类型。
3. ​**动态类型转换**：  
   可以通过 QVariant::toInt()、QVariant::toString() 等方法将存储的值转换为特定类型。
4. ​**支持自定义类型**：  
   通过 Q\_DECLARE\_METATYPE 和 qRegisterMetaType，QVariant 可以支持自定义类型。
5. ​**广泛用于 Qt 框架**：  
   QVariant 在 Qt 中广泛用于模型/视图框架（如 QAbstractItemModel）、属性系统（如 QObject::property）以及信号槽机制。

**​\*\***QVariant**的常见用法\*\***

**​1. 存储和获取基本类型**

|  |
| --- |
| QVariant v1 = 42; // 存储 int  QVariant v2 = 3.14; // 存储 double  QVariant v3 = "Hello"; // 存储 const char\*  QVariant v4 = QString("Qt"); // 存储 QString  int i = v1.toInt(); // 获取 int  double d = v2.toDouble(); // 获取 double  QString s = v4.toString(); // 获取 QString |

**2. 存储和获取 Qt 类型**

|  |
| --- |
| QVariant v1 = QDateTime::currentDateTime(); // 存储 QDateTime  QVariant v2 = QSize(100, 200); // 存储 QSize  QDateTime dt = v1.toDateTime(); // 获取 QDateTime  QSize size = v2.toSize(); // 获取 QSize |

**​3. 检查类型**

|  |
| --- |
| QVariant v = 42;  if (v.type() == QVariant::Int) {  qDebug() << "v is an int";  }  if (v.canConvert<int>()) {  qDebug() << "v can be converted to int";  } |

**​4. 支持自定义类型**

|  |
| --- |
| #include <QMetaType>  struct MyCustomType {  int id;  QString name;  };  Q\_DECLARE\_METATYPE(MyCustomType) // 注册自定义类型  int main() {  qRegisterMetaType<MyCustomType>(); // 注册到元对象系统  MyCustomType customData = {1, "Qt"};  QVariant v = QVariant::fromValue(customData); // 存储自定义类型  MyCustomType retrievedData = v.value<MyCustomType>(); // 获取自定义类型  qDebug() << "ID:" << retrievedData.id << "Name:" << retrievedData.name;  return 0;  } |

**\*\***QVariant**的应用场景\*\***

1. ​**模型/视图框架**：  
   在 QAbstractItemModel 中，QVariant 用于存储和返回模型数据。
2. ​**属性系统**：  
   在 QObject 中，QVariant 用于存储和获取动态属性。
3. ​**信号槽机制**：  
   QVariant 可以用于传递任意类型的参数。
4. ​**配置文件或数据存储**：  
   QVariant 可以方便地存储和读取多种类型的数据。

**​\*\***QVariant**的局限性\*\***

1. ​**性能开销**：  
   由于 QVariant 需要存储类型信息并进行动态类型检查，可能会带来一定的性能开销。
2. ​**不支持所有类型**：  
   某些复杂类型（如指针或非 POD 类型）可能无法直接存储。
3. ​**类型转换失败**：  
   如果尝试将 QVariant 转换为不兼容的类型，可能会导致运行时错误。