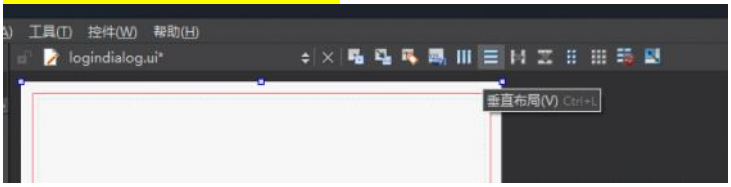
JstChat1

2025年5月11日 17:21

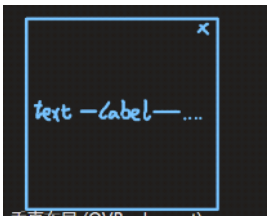
---------- 登录界面 ----------

》》》》什么是QT中的垂直布局和水平布局?



Qt中的水平布局（QHBoxLayout）和垂直布局（QVBoxLayout）是控制部件（如按钮、文本框、标签等）在界面中排放方式的布局管理器。

1. 水平布局 (QHBoxLayout)
   - 在水平布局中，界面上的部件会沿着水平方向（从左到右）依次排列。
   - 每个控件会水平摆放在一个行内，从左到右的顺序，控件之间的间隔可以通过布局管理器进行设置。



2. 垂直布局 (QVBoxLayout)
   - 在垂直布局中，界面上的部件会沿着垂直方向（从上到下）依次排列。
   - 每个控件会垂直摆放在一列中，从上到下的顺序，控件之间的间隔同样可以通过布局管理器进行设置。

总结：
- 水平布局将部件按水平（从左到右）的顺序排列。
- 垂直布局将部件按垂直（从上到下）的顺序排列。

》》》》QT中的资源文件：.qrc 可以在同目录下存在多个吗？比如resources之下有：Icons.qrc，Textures.qrc多个 .qrc 文件

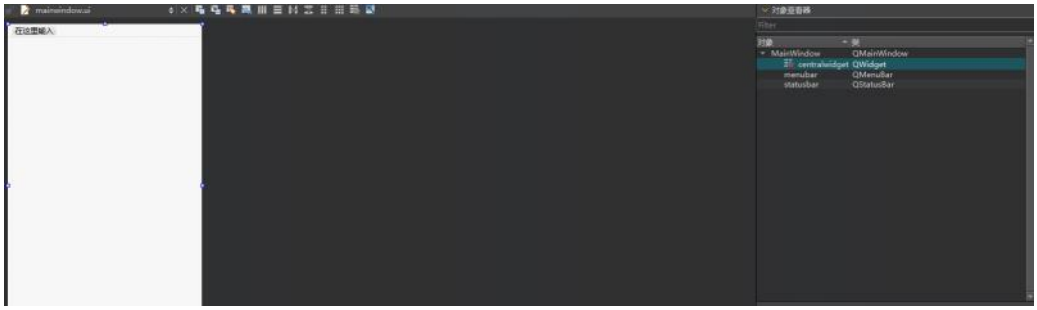
可以。
1. 每个 .qrc 文件是一个资源集合，它可以包含多个文件，比如图片、图标、样式表等。
2. 多个 .qrc 文件可以分门别类，例如你可以将图标资源放在 Icons.qrc 文件中，将纹理资源放在 Textures.qrc 文件中，这样可以更好地组织资源。

》》》》**Align : 对齐**

》》》》setCentralWidget()

setCentralWidget 是 Qt 框架中 QMainWindow 类的一个成员函数，它用于设置主窗口的中央部件。
具体来说，setCentralWidget 允许你指定一个部件（比如一个小部件或一个布局）作为主窗口的中心区域，一般用来指定主窗口的内容。

比如：主窗口为空，但有几个组件，其中之一为 : centralwidget，我们通过 SetCentralWidget() 函数将自定义的窗口指定于主窗口 (main window) ，然后使用 show() 函数展示。



maniWindow.cpp

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

    this->setWindowIcon(QIcon(":resources/chat4.png"));  // 将资源中的 Icon 设置在窗口上
}
```

》》》》什么是 QT 中的 signals 和 slots？

signals

signals 关键字是 Qt 中特有的，用于声明信号。
信号是 Qt 的对象间通信机制之一，signals 通常与槽（slots）配合使用。信号用于在对象之间发送通知，信号和槽的连接由 Qt 的 QObject::connect() 函数处理。
例如，当某个事件发生时，发出一个信号，其他对象可以响应该信号。

```
class MyClass : public QObject {
    Q_OBJECT  // 必须包含此宏才能使用信号和槽

public:
    MyClass() : QObject() {}

signals:
    void valueChanged(int newValue);  // 信号声明

public slots:
    void onValueChanged(int newValue) {
        qDebug() << "Value changed to" << newValue;
    }
};
```

在这个示例中，valueChanged 是一个信号，表示值已经变化。信号的发送通常通过 emit 关键字进行，如：

```
emit valueChanged(42);
```

slots

slots 也是在类中定义的，不同于 signals 的是， slots 可以放在类的 public、protected 或 private 部分，具体取决于你想如何控制访问权限。

public slots:	如果槽是公共的，那么外部代码可以直接调用这些槽函数。这在很多情况下是需要的，因为槽函数可能是信号的响应函数。
private slots:	如果槽是私有的，那么它们只能在类内部被调用，外部代码无法直接调用。通常用于仅在内部响应某些信号的场景。
protected slots:	保护槽可以在类的派生类中访问。

```
class MyClass : public QObject {
    Q_OBJECT

public:
    MyClass() {}

signals:
    void valueChanged(int newValue);

public slots:
    void onValueChanged(int newValue) {
        qDebug() << "Value changed to" << newValue;
    }

private slots:
    void internalSlot() {
        // 仅在内部调用的槽
    }
};
```

Signals 的作用域（是否存在修饰符？）

在 Qt 中，signals 不支持像 slots 那样使用 public或 private 进行前向修饰，因为信号的作用是：让类的外部对象能够触发它们，并与某个对象进行通信。
因此 signals 在类中默认是 public 的，如果信号被声明为私有或受保护的，外部对象就无法连接到它，这违背了信号与槽机制的设计目的。

```
class MyClass : public QObject {
    Q_OBJECT

public:
    MyClass() {}

signals:  // 这里的信号默认是 public 的，不需要显式指定
    void valueChanged(int newValue);  // 信号是 public 的，外部可以连接和触发

private:
    void privateMethod() {}
};
```

在这个示例中，valueChanged 信号默认是 public 的，你不需要显示地使用 public signals。Qt 也不允许将其声明为 private。

slots 和 signals 的区别：

signals：	通常是 public 的，用于与外部对象通信。信号不会有访问限制，因此能够被外部对象通过 connect() 连接和触发。
slots：	可以是 public、protected 或 private，这取决于设计者是否希望在外部访问，或只在类内部调用。

》》》》注意：两个不同的 LoginDialog

```
// 该 LoginDialog 属于 UI 命名空间下，继承于 Ui_LoginDialog
namespace Ui {
    class LoginDialog;
}

// 该 LoginDialog 属于全局范围下，继承于 QDialog
class LoginDialog : public QDialog              // LoginDialog 继承于 QDialog，所以有一些默认的成员函数可以使用（比如 show()）
{
    Q_OBJECT

public:
    explicit LoginDialog(QWidget *parent = nullptr);
    ~LoginDialog();

private:
    Ui::LoginDialog *ui;                        // 注意，此处使用的类型是 UI::LoginDialog，而非 ::LoginDialog

signals:
    void switchRegister();
```

》》》》 qss 文件编写格式文档
(参考网址)[https://doc.qt.io/qt-6/stylesheet-syntax.html]

.qss 文件是 Qt Style Sheets 文件，用于在 Qt 应用程序中定义界面元素的样式和外观，类似于网页中的 CSS（Cascading Style Sheets）。
Qt Style Sheets 允许开发者控制 Qt 小部件（例如按钮、标签、文本框等）在多种情况下的外观、颜色、字体等属性。

》》qss 中的 # 是什么意思?
在 CSS 或 QSS 中，# 是一个选择符，表示选择具有特定 ID 的元素。

```
eg.
QDialog#LoginDialog {
    background-color: lightblue;
}
```

在 QSS 中使用 # 来选择这个 objectName 为 LoginDialog 的 QDialog ，并手动将其背景颜色设置为蓝色。

》》》》关于：QLatin1String 类
QLatin1String 不是一个函数，而是一个 Qt 类。它用于表示 Latin-1 编码（ISO 8859-1）的一种优化形式，通常用于字符串的存储和传递。

QLatin1String 类的作用：QLatin1String 主要用于处理 Latin-1 编码 的字符串，它通常比使用普通的 QString 更高效，特别是在处理大量 Latin-1 编码的数据时，因为 QLatin1String 使用的是一个只读的字节数组。

```
if(qss.open(QFile::ReadOnly))
{
    qDebug("Qss open success.");
    QString str = QLatin1String(qss.readAll());
    a.setStyleSheet(str);
    qss.close();
}
```

---------- 内存修复&qss ----------

》》》》树形管理机制：

Qt 有一个父子窗口的管理机制，这种机制有助于统筹窗口的生命周期，并对内存进行管理。

具体来说，Qt 的 父子关系 是基于 对象树形结构 来组织的，所有窗口控件（包括对话框、窗口等）都可以设计成这种结构（我猜想这可能是通过父类与子类之间的继承，来进行设计）。

树形结构及其特性

1. 树形结构：	Qt 中的对象（尤其是界面元素）是通过父子关系组织成一个树形结构的。 每个控件（比如 QWidget）都可以指定一个父控件（比如 MainWindow），它将成为该控件的父节点。如果某个控件没有父控件，它就是一个顶级控件（根控件）。 例如： • MainWindow 是根控件，它没有父控件。 • LoginDialog 是 MainWindow 的子控件，它的父控件是 MainWindow。 这个树形结构的组织方式类似于一个 父节点和子节点 之间的关系，所有控件都会被包含在这个树中。
2. 内存管理和生命周期：	当父控件被销毁时，所有子控件会自动被销毁，确保没有内存泄漏。这个机制是通过 QObject 的析构函数实现的。 例如： • 当 MainWindow 被销毁时，LoginDialog 和 RegisterDialog 作为它的子控件会自动销毁。 • 你不需要显式地删除 LoginDialog 或 RegisterDialog，Qt 会自动处理。
3. 可视化管理：	通过父子关系，Qt 可以控制控件的显示和隐藏。如果父控件被隐藏，所有子控件也会自动隐藏。如果父控件显示，子控件也会显示。 例如，MainWindow 隐藏时，LoginDialog 和 RegisterDialog 会自动隐藏，无需单独操作。
4. 信号和槽机制：	父子控件之间还可以通过 Qt 的 信号和槽机制 进行通信。子控件可以发射信号，父控件可以连接这些信号，并作出响应。

》》》》关于一些代码的调用流程，这是我的理解

```
m_LoginDialog->setWindowFlags(Qt::CustomizeWindowHint);
```

当我们使用 m_LoginDialog 的成员函数时，以下为该函数的使用思路

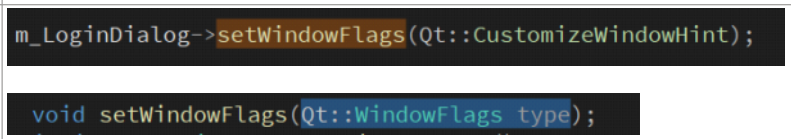
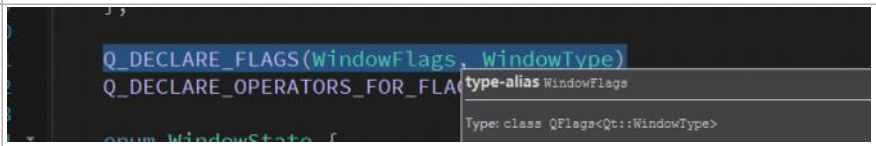
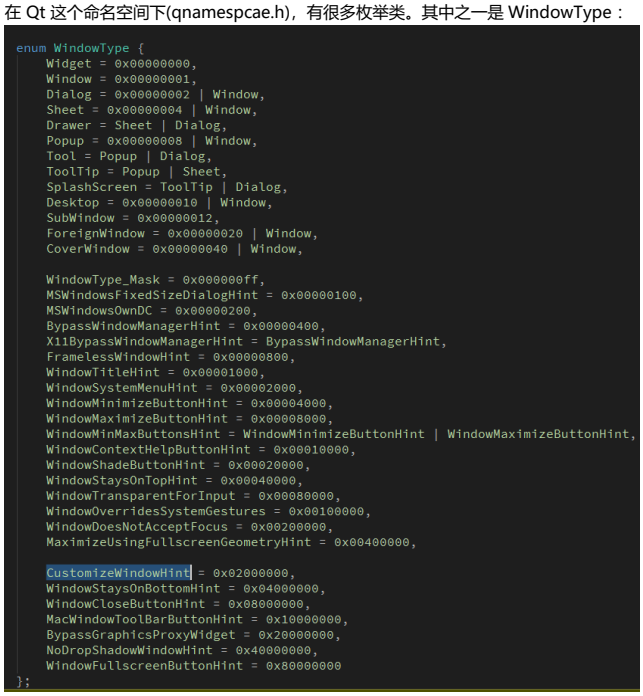
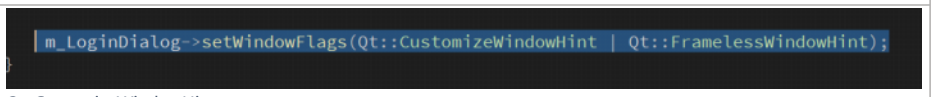
Tips:（鼠标单击某一个类名或者函数，当光标位于字段之上时，使用 F2 快捷键，可以快速转到定义文件）

首先LoginDialog 是我们在 MainWindow.h 中定义的成员	`private:`
	`  Ui::MainWindow *ui;`
	`  LoginDialog* m_LoginDialog;`
	`  RegisterDialog* m_RegisterDialog;`

这个成员属于 LoginDialog 类，而 LoginDialog 属于父类 ::Qdialog	`class LoginDialog : public QDialog`
	`{`
	`  Q_OBJECT`

::Qdialog 又属于 ::Qwidget 类	`class Q_WIDGETS_EXPORT QDialog : public QWidget`
	`{`
	`  Q_OBJECT`
	`  friend class QPushButton;`

因此 m_LoginDialog 成员是 ::Qwidget 类的派生，可以使用该类的成员函数，比如 setWindowFlags()	`#endif`
	
	`  QWidget *parentWidget() const;`
	
	`  void setWindowFlags(Qt::WindowFlags type);`
	`  inline Qt::WindowFlags windowFlags() const;`
	`  void setWindowFlag(Qt::WindowType, bool on = true);`
	`  void overrideWindowFlags(Qt::WindowFlags type);`
	
	`  inline Qt::WindowType windowType() const;`

当我们使用 m_LoginDialog 调用这个函数时，需要填入参数类型：QT::WindowFlags	
打开定义我们发现这样的宏定义： （这表明：Q_DECLARE_FLAGS 宏定义了一个位标志类型，它允许你将多个 WindowType 枚举值组合成一个 WindowFlags 类型的变量。）	
因此我们在使用该函数时，需要在参数中填写诸如：WindowType1 | WindowType2 | WindowType3 这样的组合。 然而 WindowType 的定义就在这个宏定义（Q_DECLARE_FLAGS）的上方。	在 Qt 这个命名空间下(qnamespcae.h)，有很多枚举类。其中之一是 WindowType： 
因此最终，我们可以在 MainWindow.cpp 中这样调用函数：	 1. Qt::CustomizeWindowHint： 这个标志允许用户修改窗口的默认外观，特别是窗口的边框和标题栏等。 2. Qt::FramelessWindowHint： 这个标志使窗口成为无边框窗口，意味着窗口没有默认的窗口边框、标题栏、最小化、最大化和关闭按钮。

》》》》Unpolish 和 Polish 函数

对于 QWidget 对象，我们使用其成员函数获取 Widget 的 Style，style() 函数返回 QStyle 类型的值。因此我们可以调用 QStyle 的成员函数 Polish/Unpolish()。

关于这两个函数的解释：

unpolish()：	void QStyle::unpolish(QWidget *widget); • 作用：将指定控件从样式中移除，停止对控件的样式处理。也可以理解为“撤销”该控件的样式应用。 • 参数：一个 QWidget 指针，表示要从样式中撤销的控件。 • 用法：当你需要清除控件的样式状态，重新调整控件外观时，使用这个函数。
polish()：	void QStyle::polish(QWidget *widget); • 作用：将控件重新应用样式，确保控件根据当前的样式进行渲染。 • 参数：一个 QWidget 指针，表示要重新应用样式的控件。 • 用法：通常在控件的外观或样式发生了变化时（比如样式设置更改、控件属性修改等）调用此方法，使得控件能够重新渲染。

》》》》setProperty() 函数

setProperty() 是 Qt 中 QObject 类的一个成员函数，它用于为对象设置自定义的属性。

这是 Qt 提供的一种机制，使用该函数，可以动态地为任何继承自 QObject 的对象（包括控件和其他类）添加或修改属性。

setProperty()

函数签名：	`bool QObject::setProperty(const char *name, const QVariant &value);`
参数：	○name：自定义的属性的名称，类型为 const char*，即该属性的字符串标识符。 ○value：要设置的属性值，类型为 QVariant。（QVariant 是 Qt 中用于封装各种类型数据的类，它使得你可以将任何类型的数据存储在一个对象中。） 例如：`widget->setProperty("customProperty", 42);  // 设置一个名为 "customProperty" 的自定义属性，值为 42`
返回值：	返回 true 表示属性设置成功，返回 false 表示设置失败。
作用：	setProperty() 允许你为 QObject 的派生类（如 QWidget）动态添加或修改属性。 你可以使用这种机制随时存储和检索与对象相关的数据，而不必定义额外的成员变量，并且这些属性可以通过对象的名字在代码中动态设置和获取。这对于需要在运行时，通过字符串动态访问属性的场景非常有用。

例一：

```
QWidget *widget = new QWidget;
widget->setProperty("customProperty", 42);          // 设置一个名为 "customProperty" 的自定义属性，值为 42

QVariant value = widget->property("customProperty");    // 获取属性 "customProperty" 的值
```

例二：

```
QWidget *widget = new QWidget;
widget->setProperty("customColor", QColor(255, 0, 0));      // 设置一个自定义颜色属性

// 在样式表中使用这个自定义属性
widget->setStyleSheet("background-color: qproperty(customColor);");
```

》》》》手动为一个按钮编写一个单击的触发事件 和 直接在QT Creator中通过“转到槽”->选择Click()事件 有什么不同?

两者在呈现效果上并没有什么不同。

只不过手动编写 signals 和 slots 没有 QT 的自动化功能快捷。但手动编写事件触发函数可以带来更加灵活的方式。

》》》》什么是正则表达式？正则表达式的规则是什么?

》》》》正则表达式

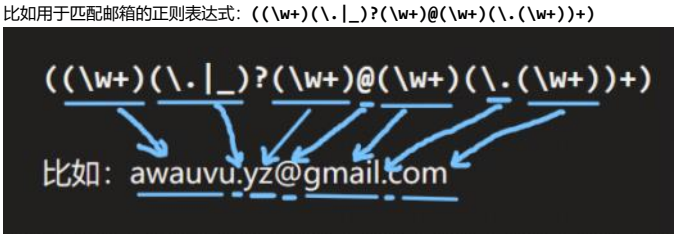
定义：

正则表达式（Regular Expression，简称 regex 或 regexp）是一种用于匹配字符串的模式。它由一些字符组成，利用这些字符可以定义复杂的匹配规则。并可以用来检查、查找、替换或操作字符串中的文本。

正则表达式的规则表：

<table>
<tr><td>普通字符：</td><td>字母、数字和标点符号等普通字符代表它们自己，如 a、1、# 等。</td></tr>
<tr><td>特殊字符：</td><td>
<table>
<tr><td>.：</td><td>匹配除换行符以外的任何单个字符。</td></tr>
<tr><td>^：</td><td>匹配输入字符串的开始位置。</td></tr>
<tr><td>$：</td><td>匹配输入字符串的结束位置。</td></tr>
<tr><td>*：</td><td>匹配前一个字符零次或多次。</td></tr>
<tr><td>+：</td><td>匹配前一个字符一次或多次。</td></tr>
<tr><td>?：</td><td>匹配前一个字符零次或一次。</td></tr>
<tr><td>[]：</td><td>字符集，匹配括号内的任何字符，例如 [abc] 匹配 a、b 或 c。</td></tr>
<tr><td>|：</td><td>表示“或”操作符，例如 a|b 匹配 a 或 b。</td></tr>
</table>
注意：
1 -> “.”是一个有意义的特殊字符，如果需要匹配"."，则需要对齐进行转移。
2 ->"*" , "+" , "?" 指的是匹配前一个字符一次或者多次，而不是前面所有字符一次或多次。
（比如((\w+)(\.|_)?)，这里的 “? ” 仅作用于(\.|_)，即仅作用于"." "_" 二者的匹配。）
</td></tr>
<tr><td>转义字符：</td><td>\：用来转义特殊字符，使其失去特殊含义，或者用于表示一些特殊字符，如：
<table>
<tr><td>\d：</td><td>匹配一个数字，等价于 [0-9]。</td></tr>
<tr><td>\w：</td><td>匹配一个字母、数字或下划线，等价于 [A-Za-z0-9_]。</td></tr>
<tr><td>\s：</td><td>匹配一个空白字符（包括空格、制表符、换行符等）。</td></tr>
</table>
</td></tr>
<tr><td>量词：</td><td>
<table>
<tr><td>{n}：</td><td>匹配前一个字符恰好出现 n 次，例如 a{3} 匹配 aaa。</td></tr>
<tr><td>{n,}：</td><td>匹配前一个字符至少出现 n 次，例如 a{2,} 匹配 aa、aaa、aaaa 等。</td></tr>
<tr><td>{n,m}：</td><td>匹配前一个字符出现 n 到 m 次，例如 a{2,4} 匹配 aa、aaa 或 aaaa。</td></tr>
</table>
</td></tr>
<tr><td>分组与捕获：</td><td>
<table>
<tr><td>()：</td><td>用于分组，可以将多个字符组合成一个单元，进行整体匹配。分组还可以用于捕获匹配的内容，例如 (abc) 匹配 abc，并且可以获取匹配到的字符串。</td></tr>
</table>
</td></tr>
</table>

比如用于匹配邮箱的正则表达式：**((\w+)(\.|_)?(\w+)@(\w+)(\.(\w+))+)**



》》对于末尾的理解

```
((\w+)(\.|_)?(\w+)@(\w+)(\.(\w+))+)
```

这里的 “+” 指的是对 (\.(\w+)) 需要匹配多次，比如对于这个邮箱：user@mail.example.co.uk，我们需要对 (\.(\w+)) 匹配多次，因为我们会多次获取 .example, .co, .uk

》》》》QRegularExpression 是什么类型

QRegularExpression

QRegularExpression 是一个类，用于表示正则表达式对象。

（它是 Qt 5.0 引入的，用于替代早期版本中的 QRegExp 类，提供了更加现代的正则表达式支持，并且符合 ECMAScript 标准（JavaScript 的正则表达式语法）。

》》》》match 和 hasMatch 函数

match() 函数

<table>
<tr><td>函数原型：</td><td>QRegularExpressionMatch QRegularExpression::match(const QString &str, int offset = 0) const;</td></tr>
<tr><td>功能：</td><td>match() 用于检查字符串 str 是否与正则表达式模式匹配。它返回一个 QRegularExpressionMatch 对象，包含匹配结果的详细信息。</td></tr>
<tr><td>参数：</td><td>○ str：要匹配的字符串。
○ offset：指定从哪个位置开始匹配（默认为 0）。</td></tr>
<tr><td>返回值：</td><td>返回一个 QRegularExpressionMatch 对象，如果匹配成功，则 QRegularExpressionMatch 对象包含匹配的细节，否则返回一个无效的匹配对象。</td></tr>
<tr><td>示例：</td><td>

```
QRegularExpression re("\\d+");  // 匹配一个或多个数字
QRegularExpressionMatch match = re.match("1234");
if (match.hasMatch()) {
    qDebug() << "Matched!" << match.captured(0);  // 输出匹配的内容
}
```

</td></tr>
</table>

hasMatch() 函数

函数原型：	bool QRegularExpression::hasMatch() const;
功能：	hasMatch() 用于检查正则表达式是否与输入的字符串匹配。它返回一个布尔值 true 或 false，指示是否有匹配。
返回值：	如果匹配成功，返回 true；否则返回 false。

示例：	`QRegularExpression re("\\d+");  // 匹配一个或多个数字`
	`QRegularExpressionMatch match = re.match("abc123");`
	`if (match.hasMatch()) {`
	`    qDebug() << "Match found!";`
	`} else {`
	`    qDebug() << "No match.";`
	`}`

》》》》QT 中的 tr

```
#ifndef QT_NO_TRANSLATION
// full set of tr functions
#  define QT_TR_FUNCTIONS \
    static inline QString tr(const char *s, const char *c = nullptr, int n = -1) \
        { return staticMetaObject.tr(s, c, n); }
#else
// inherit the ones from QObject
# define QT_TR_FUNCTIONS
#endif
```

Tr 是一个函数，当QT_TR_FUNCTIONS 这个宏被调用之后，tr 得以被定义。

tr 的功能是：	tr 用于实现字符串的国际化（i18n）。它将字符串标记为需要翻译的文本，并将其与应用程序的翻译文件进行关联。
用例：	tr 通常用于 Qt 中的类（特别是 QObject 的子类）的方法和构造函数中。例如： `QString translatedText = tr("Hello, World!");` 在这个例子中，"Hello, World!" 会被标记为一个待翻译的字符串。 在应用程序运行时，如果存在相应语言的翻译文件（如 app_zh_CN.ts），这个字符串会被翻译成相应语言。

》》》》一些误解：

这个是宏函数：	#define MAX(a, b) ((a) > (b) ? (a) : (b))
这个是宏：	# define QT_TR_FUNCTIONS static inline QString tr(const char *s, const char *c = nullptr, int n = -1) { return staticMetaObject.tr(s, c, n); }
	这个宏的作用是在调用 QT_TR_FUNCTIONS 时，会将 QT_TR_FUNCTIONS 与其之后的函数定义进行文本替换，从而在文件中定义一个函数。