

【文本编辑器】一、界面设计

原创 Qt 学习 Qt 学习 2020-02-09



点击上方 蓝色 文字，快来 关注 我吧！

本篇采用编写代码的方式开发一个文本编辑器的界面，包括菜单栏、工具栏、中心部件和状态栏，预期的效果如下

文件

	新建(N)	Ctrl+N
	打开(O)...	Ctrl+O
	保存(S)	Ctrl+S
	另存为(A)...	
	打印(P)...	Ctrl+P
	打印预览...	
	输出为 PDF(E)...	Ctrl+D
	退出(Q)	Ctrl+Q

编辑



	撤销(U)	Ctrl+Z
	重做(R)	Ctrl+Y
	剪切(T)	Ctrl+X
	复制(C)	Ctrl+C
	粘贴(P)	Ctrl+V

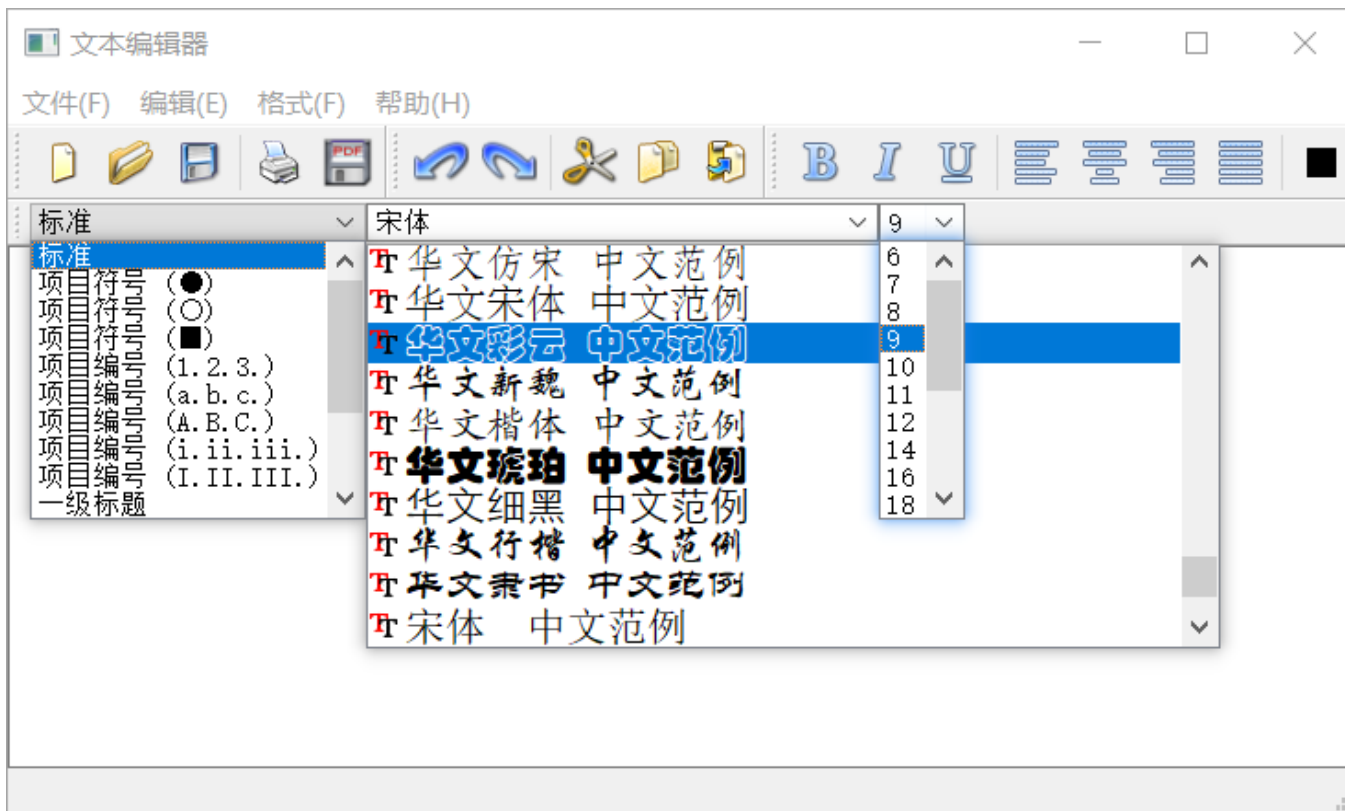
帮助

关于(A)...

关于 Qt(Q)...

格式

		加粗(B)	Ctrl+B
		倾斜(I)	Ctrl+I
		下画线(U)	Ctrl+U
字体(T) ▶		左对齐(L)	Ctrl+L
段落(P) ▶		居中(E)	Ctrl+E
		右对齐(R)	Ctrl+R
颜色(C)...		两端对齐(J)	Ctrl+J



文本编辑器界面设计的完整代码可在后台回复「**文本编辑器-界面设计**」获得下载链接。

本篇目录

1. 添加资源文件
2. “文件”菜单与工具栏的实现
3. “编辑”“格式”和“帮助”菜单与工具栏的实现
4. 中心部件和状态栏

运行环境:

win 10 + Qt 5.12.5 + Qt Creator 4.10

1. 添加资源文件

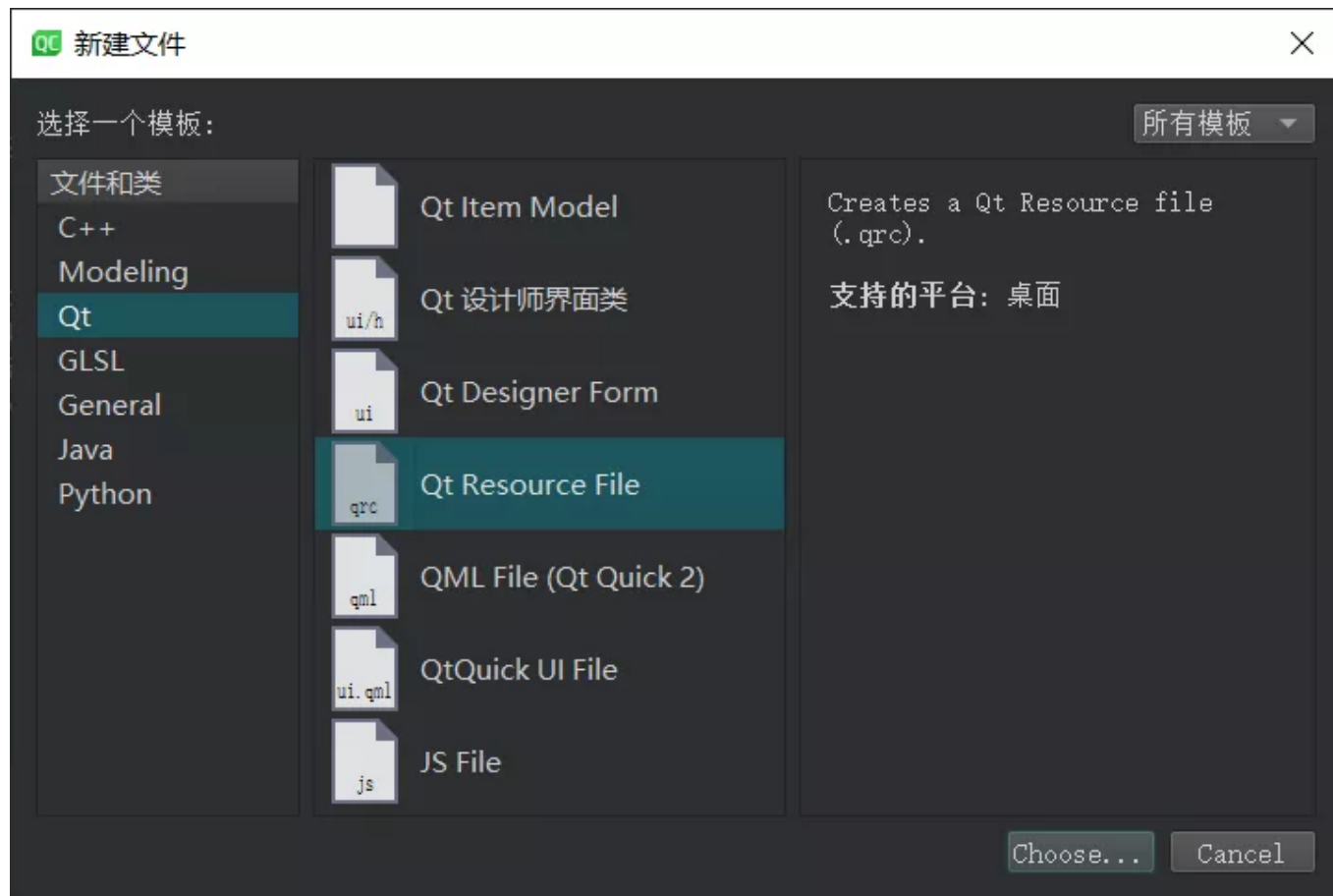
在 Qt Creator 中依次单击『文件』和『新建文件或项目...』，再依次选择 "Application" 和 "QtWidgets Application"，然后

- i. 填写项目名称为 "TextEdit"，基类选择为 "QMainWindow"，类名设置为 "TextEdit"，取消『创建界面』复选框的选中状态；
- ii. 其余选项设置保持不变，点击『下一步』直至最后的『完成』按钮，完成项目工程的建立。

项目工程中包含 main.cpp、textedit.cpp、textedit.h 和 textedit.pro 四个文件，各文件介绍可参考《[UI 文件设计与运行机制](#)》。

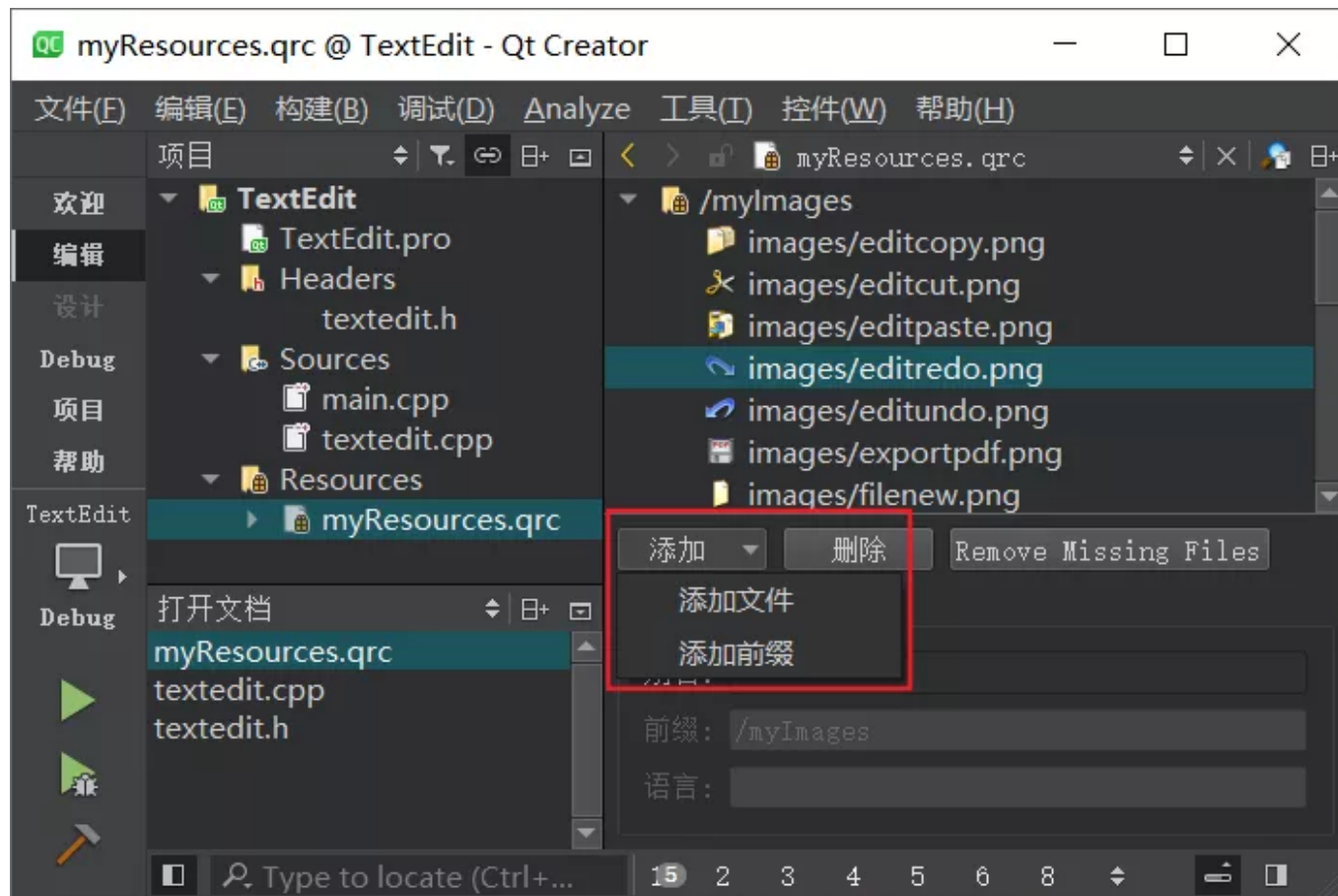
在 Qt 中可以使用资源文件将各种类型的文件添加到最终生成的可执行文件中（推荐），也可以直接加载使用外部文件。本篇推荐使用资源文件，在编译时 Qt 会将资源文件进行压缩，使生成的可执行文件较小。

在项目 "TextEdit" 上右击，单击『Add New...』，在模板中选择『Qt』和『Qt Resource File』，输入名称 "myResources"，完成向项目中添加 Qt 资源文件，如下图



创建完文件后会自动打开该资源文件，如下图所示，点击『添加』按钮，选择『添加前缀』，将默认前缀 `"/new/prefix1"` 修改为 `"/myImages"`（可任意修改，不出现中文字符即可）。然后，点击『添加』，单击『添加文件』，选择需要加载的资源。

在文本编辑器界面设计中，菜单栏和工具栏的设计都需要使用图标。这里将图标文件（均为 png 格式图片，后台回复「**文本编辑器-界面设计**」可获取）统一放置在项目目录的 `images` 文件夹中，按前一段方法将这些图片添加进资源文件。完成后如下图所示



注意：当添加完资源后，需要保存一次所有文件（快捷键 "Ctrl+Shift+S"），否则在后面可能无法显示已经添加的资源。

2. “文件”菜单与工具栏的实现

菜单与工具栏都与 `QAction` 类密切相关，工具栏上的功能按钮与菜单中的部分选项条目相对应，完成相同的功能，使用相同的快捷键、图标和状态栏提示。`QAction` 类为应用程序提供了统一的命令接口，使得从菜单栏或工具栏或通过快捷键触发都调用相同的操作接口，得到相同的触发效果。



完成“文件”菜单栏和工具栏得创建之前，先要对相关得动作 (Action) 进行声明，然后分别创建文件主菜单和工具栏，并将这些动作加入其中，“文件”主菜单各功能项如上图所示。在 "textedit.cpp" 中添加动作、菜单和工具条对应的头文件（加粗部分，下同）

```
1 #include "textedit.h"
2 #include <QAction>
3 #include <QMenu>
4 #include <QMenuBar>
5 #include <QToolBar>
```

同时在 "textedit.h" 中添加类 `QAction` 的前置声明

```
1 #include <QMainWindow>
2 class QAction;
```

在 `TextEdit` 类私有部分声明“文件”主菜单和工具条的实现函数和其下各个功能项动作：

```
1 class TextEdit : public QMainWindow
2 {
3     Q_OBJECT
4
5     public:
6         TextEdit(QWidget *parent = nullptr);
```

```
7     ~TextEdit();
8
9     private:
10        // 文件主菜单和工具条的实现函数
11        void setupFileActions();
12        // 文件主菜单的功能项
13        QAction *actionNew;
14        QAction *actionOpen;
15        QAction *actionSave;
16        QAction *actionSaveAs;
17        QAction *actionPrint;
18        QAction *actionPrintPreview;
19        QAction *actionExportPDF;
20        QAction *actionQuit;
21    }
```

在源文件 "textedit.cpp" 中添加函数 `setupFileActions()` 的实现代码：

```
1 void TextEdit::setupFileActions()
2 {
3     // 文件主菜单动作集
4     actionNew = new QAction(QIcon(rsrcPath + "/filenew.png"), tr("新建(&N)"), this);
5     actionNew->setShortcut(tr("Ctrl+N"));
6     actionNew->setToolTip(tr("新建"));
7     actionNew->setStatusTip(tr("创建一个新文档"));
8     // connect(actionNew, &QAction::triggered, this, &TextEdit::fileNew);
9
10    actionOpen = new QAction(QIcon(rsrcPath + "/fileopen.png"), tr("打开(&O)..."), this);
11    actionOpen->setShortcut(tr("Ctrl+O"));
12    actionOpen->setToolTip(tr("打开"));
13    actionOpen->setStatusTip(tr("打开已存在的文档"));
14    // connect(actionOpen, &QAction::triggered, this, &TextEdit::fileOpen);
15
16    actionSave = new QAction(QIcon(rsrcPath + "/filesave.png"), tr("保存(&S)"), this);
17    actionSave->setShortcut(tr("Ctrl+S"));
18    actionSave->setToolTip(tr("保存"));
19    actionSave->setStatusTip(tr("将当前文档存盘"));
```



```
20 // connect(actionSave, &QAction::triggered, this, &TextEdit::fileSave);
21
22 actionSaveAs = new QAction(tr("另存为(&A)..."), this);
23 actionSaveAs->setStatusTip(tr("以一个新名字保存文档"));
24 // connect(actionSaveAs, &QAction::triggered, this, &TextEdit::fileSaveAs);
25
26 actionPrint = new QAction(QIcon(rsrcPath + "/fileprint.png"), tr("打印(&P)..."), this);
27 actionPrint->setShortcut(tr("Ctrl+P"));
28 actionPrint->setToolTip(tr("打印"));
29 actionPrint->setStatusTip(tr("打印文档"));
30 // connect(actionPrint, &QAction::triggered, this, &TextEdit::filePrint);
31
32 actionPrintPreview = new QAction(tr("打印预览..."), this);
33 actionPrintPreview->setStatusTip(tr("预览打印效果"));
34 // connect(actionPrintPreview, &QAction::triggered, this, &TextEdit::filePrintPreview);
35
36 actionExportPDF = new QAction(QIcon(rsrcPath + "/exportpdf.png"), tr("输出为 PDF(&E)..."), this);
37 actionExportPDF->setShortcut(tr("Ctrl+D"));
38 actionExportPDF->setToolTip(tr("输出为 PDF"));
39 actionExportPDF->setStatusTip(tr("将文档导出为 PDF 格式"));
40 // connect(actionExportPDF, &QAction::triggered, this, &TextEdit::filePrintPdf);
41
42 actionQuit = new QAction(tr("退出(&Q)"), this);
43 actionQuit->setShortcut(tr("Ctrl+Q"));
44 actionQuit->setStatusTip(tr("退出应用程序"));
45 // connect(actionQuit, &QAction::triggered, this, &QWidget::close);
46
47 // 文件主菜单
48 QMenu *menu = menuBar()->addMenu(tr("文件(&F)"));
49 menu->addAction(actionNew);
50 menu->addAction(actionOpen);
51 menu->addSeparator();
52 menu->addAction(actionSave);
53 menu->addAction(actionSaveAs);
54 menu->addSeparator();
55 menu->addAction(actionPrint);
56 menu->addAction(actionPrintPreview);
57 menu->addAction(actionExportPDF);
58 menu->addSeparator();
```

```
59     menu->addAction(actionQuit);
60
61     // 文件工具条
62     QToolBar *fileToolbar = addToolBar(tr("文件"));
63     fileToolbar->addAction(actionNew);
64     fileToolbar->addAction(actionOpen);
65     fileToolbar->addAction(actionSave);
66     fileToolbar->addSeparator();
67     fileToolbar->addAction(actionPrint);
68     fileToolbar->addAction(actionExportPDF);
69 }
```

(左右滑动试试)

以新建动作为例说明，在“文件”主菜单动作集中

- `actionNew = new QAction(QIcon(rsrcPath + "/filenew.png"), tr("新建(&N)"), this);`

创建“新建”动作，并指定此动作使用的图标、名称及父窗口。

- `actionNew->setShortcut(tr("Ctrl+N"));`

设置此动作的快捷键。

- `actionNew->setToolTip(tr("新建"));`

当鼠标的光标移至此动作对应的工具栏按钮上时，在按钮的下方显示“创建一个新文档”的提示。

- `actionNew->setStatusTip(tr("创建一个新文档"));`

当鼠标的光标移至此动作对应的菜单条目或工具栏按钮上时，在状态条上显示“创建一个新文档”的提示。

在文件主菜单中

- `QMenu *menu = menuBar()->addMenu(tr("文件(&F)"));`

调用 `QMainWindow` 的 `menuBar()` 函数得到主窗口的菜单条指针，再调用菜单条 `QMenuBar` 的 `addMenu()` 函数完成插入一个新菜单。

- `menu->addAction(actionNew);`

在主菜单中加入“新建”菜单条目。

- `menu->addSeparator();`

在菜单条目间插入分割条。

在文件工具条中

- `QToolBar *fileToolBar = addToolBar(tr("文件"));`

调用 `QMainWindow` 的 `addToolBar()` 函数得到主窗口的工具条指针，并在主窗口新增一个工具条。

- `fileToolBar->addAction(actionNew);`

在工具条中加入“新建”工具按钮。

- `fileToolBar->addSeparator();`

在工具按钮间插入分割条。

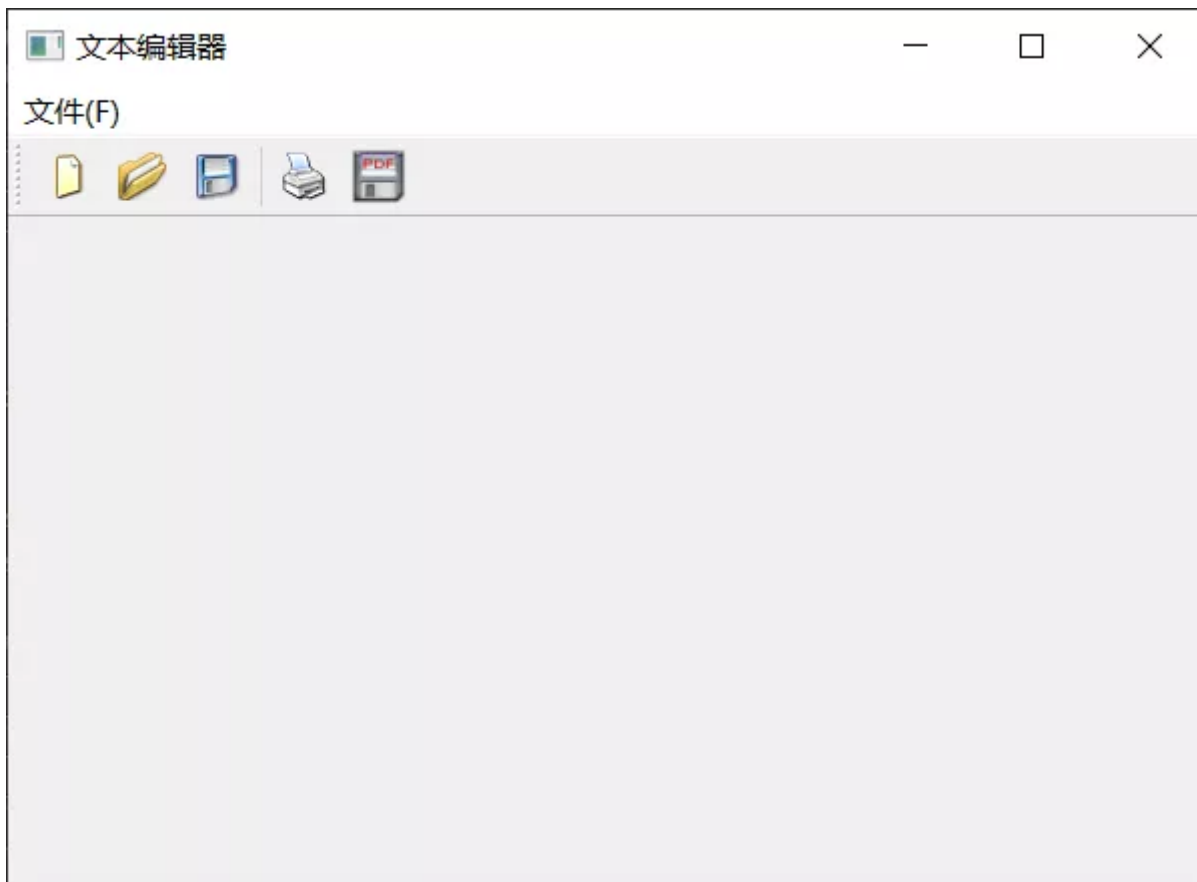
说明：被注释的部分代码将会在后续的推送中做解释，且代码中的 `rsrcPath` 定义如下

```
1  const QString rsrcPath = ":/myImages/images";
```

将创建“文件”菜单和工具条的函数 `setupFileActions()` 添入构造函数中，代码如下

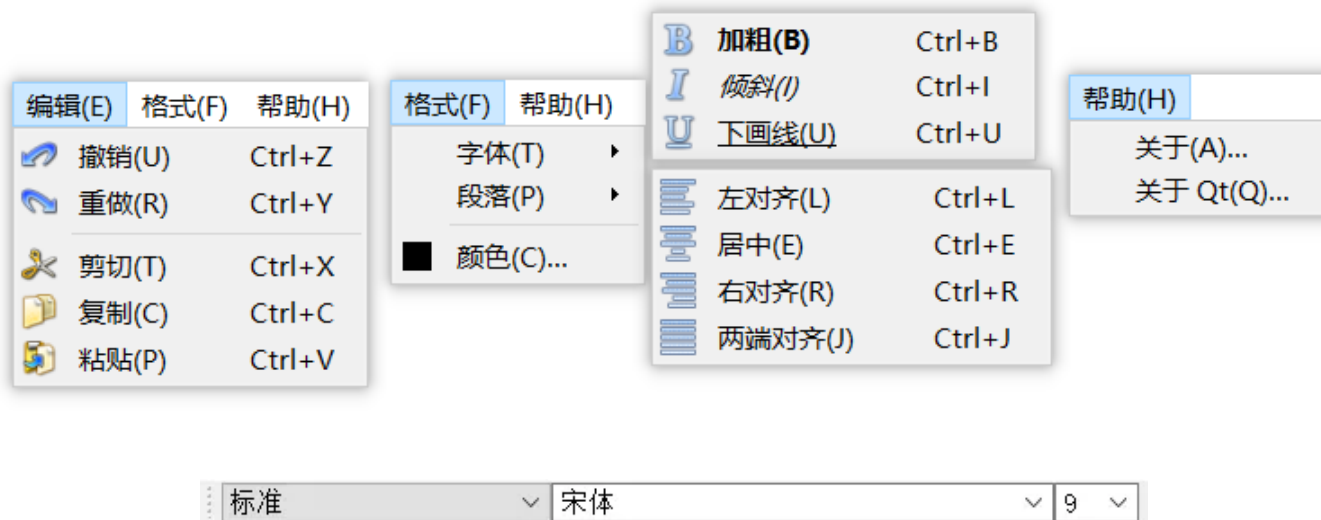
```
1  QTextEdit::TextEdit(QWidget *parent)
2      : QMainWindow(parent)
3  {
4      setWindowTitle(tr("文本编辑器")); // 设置窗体标题
5      setMinimumSize(QSize(600, 400)); // 设置窗体的最小尺寸
6      setupFileActions();
7  }
```

“文件”菜单和工具条的运行效果如下图（更详细的效果可见文末的小视频）



3. “编辑”“格式”和“帮助”菜单与工具栏的实现

“编辑”“格式”和“帮助”菜单与工具条的实现与“文件”菜单与工具条的实现过程类似，各功能项如下图



在 "textedit.h" 中添加类的前置声明：

```
1 class QComboBox;    // 段落和字号组合框
2 class QFontComboBox; // 字体组合框
```

同时记得在 "textedit.cpp" 添加头文件

```
1 #include <QApplication>
2 #include <QComboBox>
3 #include <QFontComboBox>
```

以及在 `TextEdit` 类的私有部分添加主菜单和工具条的实现函数，代码如下：

```
1  ... // 文件主菜单和工具条的实现函数
2  void setupEditActions(); // 编辑主菜单和工具条的实现函数
3  void setupTextActions(); // 格式主菜单和工具条的实现函数
4  void setupHelpActions(); // 帮助主菜单和工具条的实现函数
5
6  ... // 文件菜单栏功能项
7
8  QAction *actionUndo; // 编辑菜单栏功能项
9  QAction *actionRedo;
10 QAction *actionCut;
11 QAction *actionCopy;
12 QAction *actionPaste;
13
14 QAction *actionTextBold; // 格式菜单栏功能项
15 QAction *actionTextItalic;
16 QAction *actionTextUnderline;
17 QAction *actionAlignLeft;
18 QAction *actionAlignCenter;
19 QAction *actionAlignRight;
20 QAction *actionAlignJustify;
21 QAction *actionTextColor;
22
23 QAction *actionAbout; // 帮助菜单栏功能项
24 QAction *actionQtAbout;
25
26 QComboBox *comboStyle; // 段落组合框
27 QFontComboBox *comboFont; // 字体组合框
28 QComboBox *comboSize; // 字号组合框
```

“编辑”、“帮助”菜单和工具条创建函数 `setupEditActions()` 和 `setupHelpActions()` 与“文件”的创建过程一致，下面只着重介绍下 `setupTextActions()` 函数。代码分两部分介绍，第一部分代码如下：

```
1  void TextEdit::setupTextActions()
2  {
```

```
3 // 格式主菜单动作集
4 actionTextBold = new QAction(QIcon(rsrcPath + "/textbold.png"), tr("加粗(&B)"), this);
5 actionTextBold->setShortcut(tr("Ctrl+B"));
6 actionTextBold->setToolTip(tr("加粗"));
7 actionTextBold->setStatusTip(tr("将所选文字加粗"));
8 QFont bold; // 定义加粗字体
9 bold.setBold(true); // 激活字体加粗
10 actionTextBold->setFont(bold); // 将加粗功能赋给加粗动作
11 actionTextBold->setCheckable(true); // 设置为可以选中
12 // connect(actionTextBold, &QAction::triggered, this, &TextEdit::textBold);
13
14 actionTextItalic = new QAction(QIcon(rsrcPath + "/textitalic.png"), tr("倾斜(&I)"), this);
15 actionTextItalic->setShortcut(tr("Ctrl+I"));
16 actionTextItalic->setToolTip(tr("倾斜"));
17 actionTextItalic->setStatusTip(tr("将所选文字倾斜"));
18 QFont italic; // 定义倾斜字体
19 italic.setItalic(true); // 激活字体倾斜
20 actionTextItalic->setFont(italic); // 将倾斜功能赋给倾斜动作
21 actionTextItalic->setCheckable(true); // 设置功能项可以选中
22 // connect(actionTextItalic, &QAction::triggered, this, &TextEdit::textItalic);
23
24 actionTextUnderline = new QAction(QIcon(rsrcPath + "/textunder.png"), tr("下画线(&U)"), this);
25 actionTextUnderline->setShortcut(tr("Ctrl+U"));
26 actionTextUnderline->setToolTip(tr("下画线"));
27 actionTextUnderline->setStatusTip(tr("为所选文字添加下画线"));
28 QFont underline; // 定义字体下画线
29 underline.setUnderline(true); // 激活字体下画线
30 actionTextUnderline->setFont(underline); // 将下画线功能赋给下画线动作
31 actionTextUnderline->setCheckable(true); // 设置功能项可以选中
32 // connect(actionTextUnderline, &QAction::triggered, this, &TextEdit::textUnderline);
33
34 actionAlignLeft = new QAction(QIcon(rsrcPath + "/textleft.png"), tr("左对齐(&L)"), this);
35 actionAlignLeft->setShortcut(tr("Ctrl+L"));
36 actionAlignLeft->setToolTip(tr("左对齐"));
37 actionAlignLeft->setStatusTip(tr("将文字左对齐"));
38 actionAlignLeft->setCheckable(true);
39
40 actionAlignCenter = new QAction(QIcon(rsrcPath + "/textcenter.png"), tr("居中(&E)"), this);
41 actionAlignCenter->setShortcut(tr("Ctrl+E"));
```

```
42     actionAlignCenter->setToolTip(tr("居中"));
43     actionAlignCenter->setStatusTip(tr("将文字居中对齐"));
44     actionAlignCenter->setCheckable(true);
45
46     actionAlignRight = new QAction(QIcon(rsrcPath + "/textright.png"), tr("右对齐(&R)"), this);
47     actionAlignRight->setShortcut(tr("Ctrl+R"));
48     actionAlignRight->setToolTip(tr("右对齐"));
49     actionAlignRight->setStatusTip(tr("将文字右对齐"));
50     actionAlignRight->setCheckable(true);
51
52     actionAlignJustify = new QAction(QIcon(rsrcPath + "/textjustify.png"), tr("两端对齐(&J)"), this);
53     actionAlignJustify->setShortcut(tr("Ctrl+J"));
54     actionAlignJustify->setToolTip(tr("两端对齐"));
55     actionAlignJustify->setStatusTip(tr("将文字两端对齐"));
56     actionAlignJustify->setCheckable(true);
57
58     // 定义动作功能组，且确保 alignLeft 始终位于 alignRight 的左侧
59     QActionGroup *alignGroup = new QActionGroup(this);
60     // connect(alignGroup, &QActionGroup::triggered, this, &TextEdit::textAlign);
61
62     if (QApplication::isLeftToRight()) {
63         alignGroup->addAction(actionAlignLeft);
64         alignGroup->addAction(actionAlignCenter);
65         alignGroup->addAction(actionAlignRight);
66     } else {
67         alignGroup->addAction(actionAlignRight);
68         alignGroup->addAction(actionAlignCenter);
69         alignGroup->addAction(actionAlignLeft);
70     }
71     alignGroup->addAction(actionAlignJustify);
72
73     QPixmap pix(16, 16);
74     pix.fill(Qt::black);
75     actionTextColor = new QAction(pix, tr("颜色(&C)..."), this);
76     actionTextColor->setToolTip(tr("颜色"));
77     actionTextColor->setStatusTip(tr("设置文字颜色"));
78     // connect(actionTextColor, &QAction::triggered, this, &TextEdit::textColor);
79
80     // 格式主菜单
```



```
81     QMenu *menu = menuBar()->addMenu(tr("格式(&F)"));
82     QMenu *fontMenu = menu->addMenu(tr("字体(&T)"));
83     fontMenu->addAction(actionTextBold);
84     fontMenu->addAction(actionTextItalic);
85     fontMenu->addAction(actionTextUnderline);
86     QMenu *alignMenu = menu->addMenu(tr("段落(&P)"));
87     alignMenu->addActions(alignGroup->actions());
88     menu->addSeparator();
89     menu->addAction(actionTextColor);
90
91     // 格式工具条
92     QToolBar *formatToolBar = addToolBar(tr("格式"));
93     formatToolBar->addAction(actionTextBold);
94     formatToolBar->addAction(actionTextItalic);
95     formatToolBar->addAction(actionTextUnderline);
96     formatToolBar->addSeparator();
97     formatToolBar->addActions(alignGroup->actions());
98     formatToolBar->addSeparator();
99     formatToolBar->addAction(actionTextColor);
100
101     ...
102 }
```

其中

- `QActionGroup *alignGroup = new QActionGroup(this);`

定义一个动作组，文本编辑器中将左对齐、居中、右对齐和两端对齐四个动作放入同一个动作组中，在这个动作组中的所有动作在同一时刻只有一个会被选中。

- `QPixmap pix(16, 16);pix.fill(Qt::black);`

创建绘图设备并将其填充为黑色。

函数 `setupTextActions()` 第二部分代码如下：

```
1 void TextEdit::setupTextActions()
2 {
3     ...
4
5     QToolBar *comboToolbar = addToolBar(tr("组合框"));
6     comboToolbar->setAllowedAreas(Qt::TopToolBarArea | Qt::BottomToolBarArea);
7     addToolBarBreak(Qt::TopToolBarArea); // 使工具条在顶部区域分多行显示
8     addToolBar(comboToolbar);
9
10    comboStyle = new QComboBox(comboToolbar);
11    comboToolbar->addWidget(comboStyle); // 将组合框添加至工具条
12    comboStyle->addItem(tr("标准"));
13    comboStyle->addItem(tr("项目符号 (●)"));
14    comboStyle->addItem(tr("项目符号 (○)"));
15    comboStyle->addItem(tr("项目符号 (■)"));
16    comboStyle->addItem(tr("项目编号 (1.2.3.)"));
17    comboStyle->addItem(tr("项目编号 (a.b.c.)"));
18    comboStyle->addItem(tr("项目编号 (A.B.C.)"));
19    comboStyle->addItem(tr("项目编号 (i.ii.iii.)"));
20    comboStyle->addItem(tr("项目编号 (I.II.III.)"));
21    comboStyle->addItem(tr("一级标题"));
22    comboStyle->addItem(tr("二级标题"));
23    comboStyle->addItem(tr("三级标题"));
24    comboStyle->addItem(tr("四级标题"));
25    comboStyle->addItem(tr("五级标题"));
26    comboStyle->addItem(tr("六级标题"));
27    comboStyle->setStatusTip(tr("添加项目符号 ( 编号 ) 或设置段落等级"));
28    //    connect(comboStyle, QOverload<int>::of(&QComboBox::activated), this, &TextEdit::textStyle);
29
30    comboFont = new QFontComboBox(comboToolbar);
31    comboToolbar->addWidget(comboFont);
32    comboFont->setStatusTip(tr("更改字体"));
33    //    connect(comboFont, QOverload<const QString &>::of(&QComboBox::activated), this, &TextEdit::textFamily);
34
35    comboSize = new QComboBox(comboToolbar);
36    comboToolbar->addWidget(comboSize);
37    comboSize->setStatusTip(tr("更改字号"));
38    comboSize->setEditable(true); // 将字号组合框设置为可编辑
```

```
39
40     const QList<int> standardSizes = QFontDatabase::standardSizes();
41     foreach (int size, standardSizes)
42         comboSize->addItem(QString::number(size));
43     comboSize->setCurrentIndex(standardSizes.indexOf(QApplication::font().pointSize()));
44 }
```

工具条是一个可移动的窗口，它可停靠的区域由 `QToolBar` 的 `allowAreas` 决定，包括

Qt::LeftToolBarArea
Qt::RightToolBarArea
Qt::TopToolBarArea
Qt::BottomToolBarArea
Qt::AllToolBarArea

默认为 `Qt::LeftToolBarArea`，启动后默认出现于主窗口的顶部，可通过 `setAllowAreas()` 函数来指定工具条的停靠区域。

调用 `QFontComboBox` 的 `setFontFilters` 接口可过滤只在下拉列表框中显示某一类字体，默认情况下为 `QFontComboBox::AllFonts` 列出所有字体。

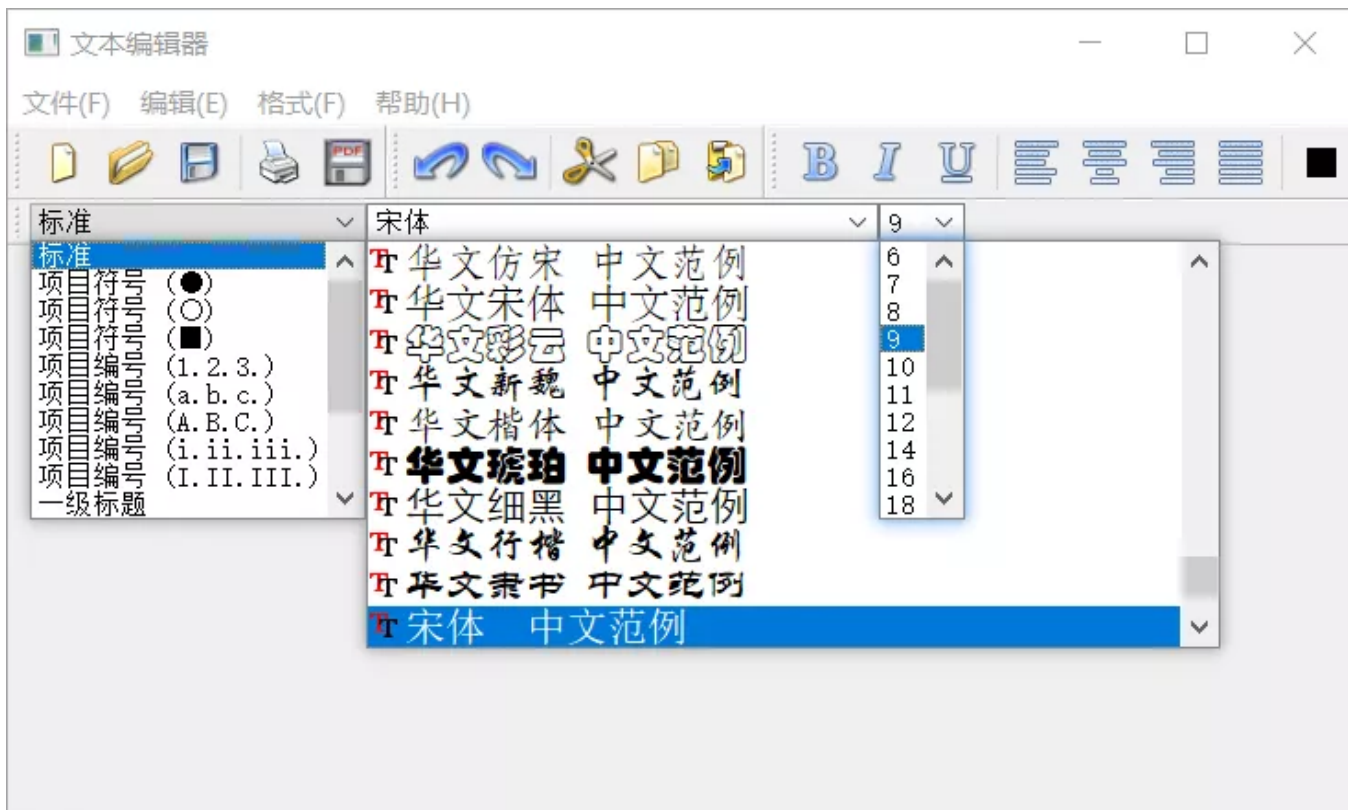
使用 `QFontDatabase` 实现在字号下拉列表框中填充各种不同的字号条目，`QFontDatabase` 类用于表示当前系统中所有可用的格式信息，主要是字体和字号大小。

调用 `standardSizes()` 函数返回可用标准字号的列表，并将它们插入到字号下拉列表框中。文本编辑器中只列出字号。

在构造函数中的 `setupFileActions()`；后面添加“编辑”“格式”和“帮助”菜单与工具条的创建函数如下：

```
1  setupEditActions();  
2  setupTextActions();  
3  setupHelpActions();
```

运行效果如下



4. 中心部件和状态栏

添加中心部件：任何一个文本编辑程序都需要用到 `QTextEdit` 类作为输入文本的容器。首先在 "textedit.h" 中添加头文件

```
1 #include <QTextEdit>
```

和 `QTextEdit` 类的前置声明

```
1 class QTextEdit;
```

然后在类的 `private` 部分添加

```
1 QTextEdit *textEdit;
```

在构造函数中将 `textEdit` 设置为中心部件

```
1 textEdit = new QTextEdit(this);  
2 setCentralWidget(textEdit);
```

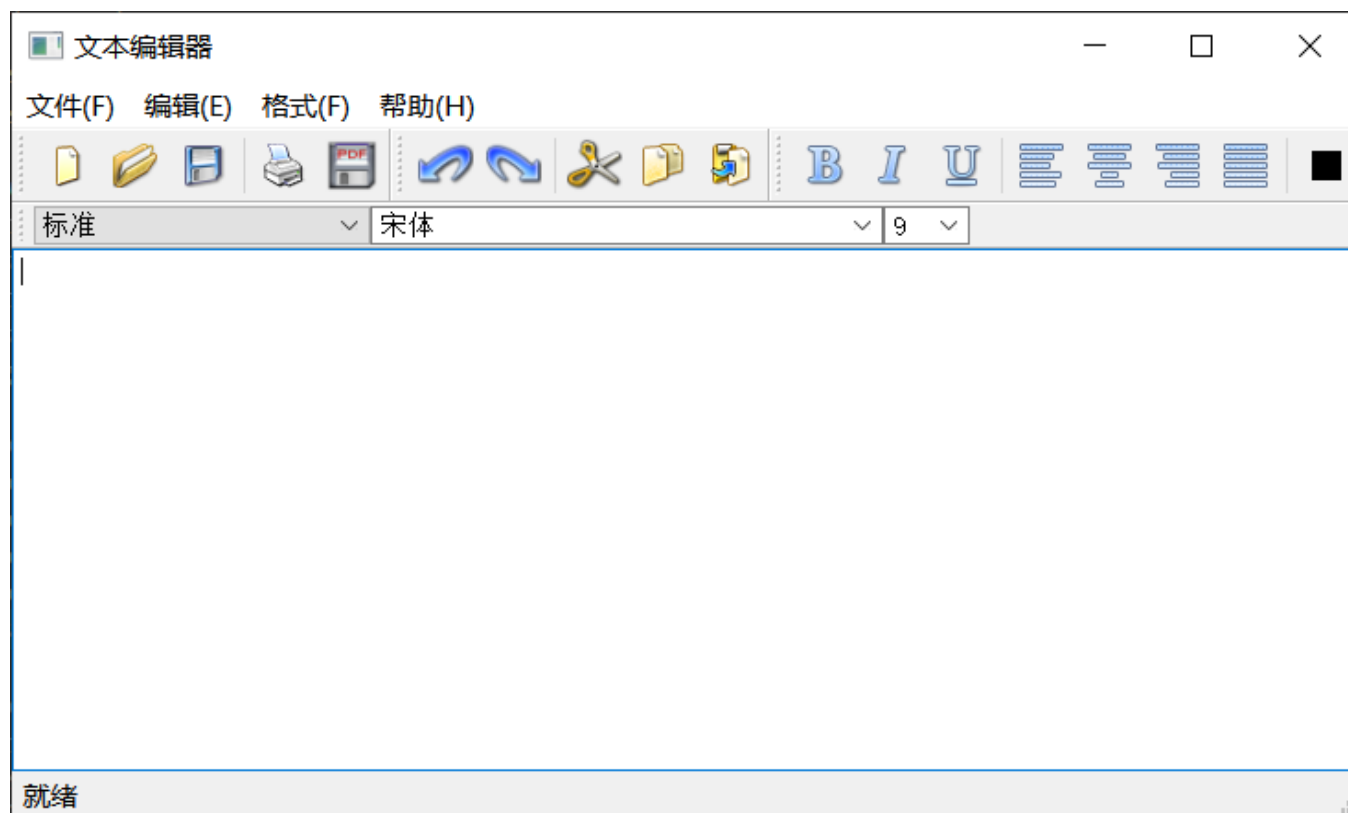
状态栏的添加比较简单，在 "textedit.cpp" 中添加头文件

```
1 #include <QStatusBar>
```

在构造函数中后添加

```
1 statusBar()->showMessage(tr("就绪"));
```

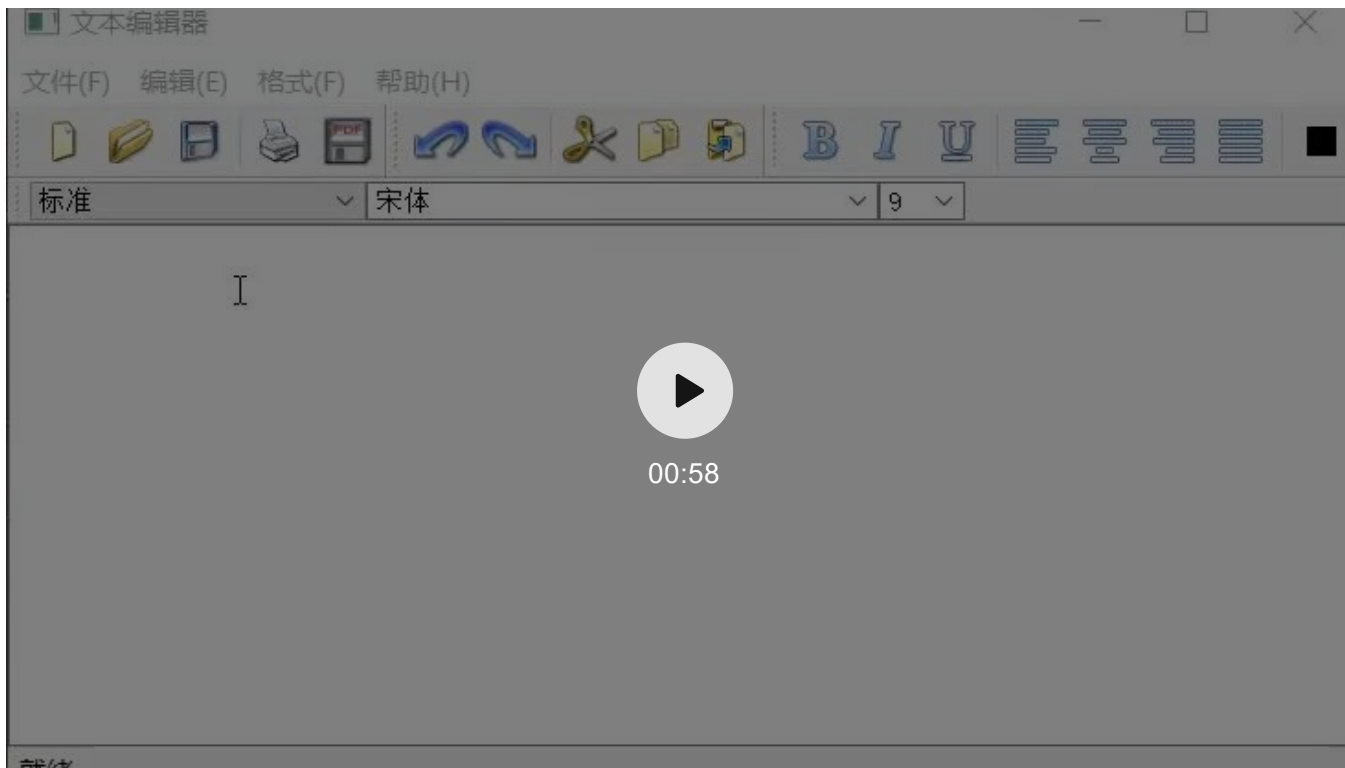
即完成状态栏的添加。最终运行效果如下图



至此就完成了整个文本编辑器的界面设计。

小结

通过一个小短片作为文本编辑器界面设计的总结：



完整的代码可在后台回复「**文本编辑器-界面设计**」获得下载链接。

相关阅读：

《代码化 UI 设计》

《可视化 UI 设计（设计器 Qt Designer 实现）》

《UI 文件设计与运行机制》

《Qt 模块简介》

《主要的窗体类和主窗体构成》

学习 教程

LEARNING TUTORIAL



长按
识别
关注

喜欢此内容的人还喜欢

那个叫韩兴博的17岁少年

狂言Doggy



聊育儿 | 这个被叫停后，该让孩子怎么学？

可涵说

