第 3 章

Python GUI 工具包：PyQt6

PyQt 是一个创建 Python GUI 应用程序的工具包，是 Qt 和 Python 结合的产物，可以认为是为了将 Qt 的功能用于 Python 开发的一个 Qt 的 Python 包装器。PyQt 由 Riverbank Computing 公司开发，是最强大的 GUI 库之一。PyQt6 是基于 Digia 公司 Qt6 的 Python 接口，是 PyQt6 的最新版本，也是目前最流行的基于 Python 的 GUI 工具包。本章将介绍 PyQt6 的主要功能以及一些常用 API 的使用方法。本书中所有的 GUI 应用都使用 PyQt6 实现，所以本章的内容非常重要。

3.1 Python 中主要的 GUI 工具包

Python 中有非常多的 GUI 工具包，这些工具包主要有 Tkinter、PyQT/PySide、wxPython、Kivy、PySimpleGui、Flexx、Toga 等，它们的优缺点如下。

(1) Tkinter。

优点：是 Python 的默认标准 GUI 库，使用简单。

缺点：设计的界面比较简陋。

(2) PyQT/PySide。

优点：作为 Python 的插件实现，功能非常强大，可以用 Qt 开发多个漂亮的界面。

缺点：商业授权上存在一些问题。

(3) wxPython。

优点：跨平台支持性很好。

缺点：使用相对较少。

(4) Kivy。

优点：跨平台支持性很好，可以用于移动端开发。

缺点：学习曲线较陡峭。

(5) PySimpleGui。

优点：使用简单，易于上手。

缺点：功能相对较少。

（6）Flexx。

优点：纯 Python 工具箱，使用 Web 技术进行渲染，更适合应用于 Web 应用中。

缺点：使用相对较少。

（7）Toga。

优点：适用于 macOS、Windows、Linux(GTK)以及 Android 和 iOS 等移动平台。

缺点：使用相对较少。

PyQT 和 PySide 之所以放在一起，是因为它们都是 Python 的 QT 绑定库，它们的 API 非常相似，最大的区别是 License。PyQT 采用了 GPL 协议，QT 公司拥有商业许可证；而 PySide 则使用 LGPL 协议，可以免费使用，不需要开源代码。目前 PyQT 由 Riverbank Computing 公司维护，比较稳定，开发社区也比较大；而 PySide(又名 Qt for Python)现由 Qt 公司维护，比 PyQt 更“年轻”一些。

本章会主要讲解 PyQt6，因为其功能非常强大，几乎可以完成任何工作，而且是目前最流行的基于 Python 的 GUI 工具包，技术社区非常繁荣。

如果读者没有安装 PyQt6，可以使用下面的命令进行安装。

```
pip3 install PyQt6
```

3.2　创建窗口

创建窗口时需要使用 QWidget 类。QWidget 类是所有用户界面对象的基类。它提供了一个默认的构造方法，可以创建一个空的窗口。QWidget 类提供了一些基本的功能，例如设置窗口标题、设置窗口图标、设置窗口大小、设置窗口位置等。此外，QWidget 类还提供了一些事件处理函数，如鼠标事件、键盘事件、绘制事件等。

如果想创建自己的用户界面对象，可以从 QWidget 类派生出一个新的类。在这个新的类中，可以添加自己的成员变量和成员函数，并且可以重载 QWidget 类中的一些函数来实现自己的功能。

本节的例子会利用 QWidget 类创建一个窗口，并设置窗口的尺寸、位置、窗口标题、窗口图标以及红色的背景颜色。完成这个例子需要使用 QWidget 类的 setGeometry 方法设置窗口的尺寸和位置，使用 setWindowTitle 方法设置窗口标题，使用 setWindowIcon 方法设置窗口图标，使用 setStyleSheet 方法设置背景颜色。这些方法的原型和参数含义如下。

1. setGeometry 方法

该方法的原型如下：

```
setGeometry(self, x: int, y: int, w: int, h: int) -> None
```

参数含义如下。

（1）x：窗口左上角的横坐标。

(2) y：窗口左上角的纵坐标。

(3) w：窗口的宽度。

(4) h：窗口的高度。

2. setWindowTitle 方法

该方法的原型如下：

```
setWindowTitle(self, str) -> None
```

str 参数表示窗口的标题。

3. setWindowIcon 方法

该方法的原型如下：

```
setWindowIcon(self, icon) -> None
```

icon 参数表示窗口图标，为 QIcon 对象。

注意：QWidget 类和 QApplication 类都有 setWindowIcon 方法，使用方法完全相同。不过由于 macOS 并不支持窗口标题，所以在 macOS 下使用 QWidget.setWindowIcon 方法是无效的，只能使用 QApplication.setWindowIcon 方法将图标放到 macOS 最下方的 Dock(程序坞)中。而在 Windows 和 Linux 中，使用 QWidget.setWindowIcon 和 QApplication.setWindowIcon 都可以，为了让程序可以同时在 macOS、Windows 和 Linux 下正常运行，所以本例使用 QApplication.setWindowIcon 方法设置标题图标。

4. setStyleSheet 方法

setStyleSheet 方法用于设置 PyQt6 应用程序中的样式表。样式表是一种用于设置组件外观的机制，类似于 CSS。可以使用 setStyleSheet 方法来设置组件的背景颜色、字体、边框等。例如，以下代码将 QPushButton 组件的背景颜色设置为蓝色：

```
QPushButton {
    background-color: blue;
}
```

可以在样式表中使用 CSS 选择器来选择要应用样式的组件。例如，以下代码将所有 QLabel 组件的字体大小设置为 30 像素：

```
QLabel {
    font-size: 30px;
}
```

setStyleSheet 方法的原型如下：

```
setStyleSheet(self, str,) -> None
```

str 参数表示样式字符串。

下面的代码完整地演示了这个例子的实现过程。

代码位置：src/gui/pyqt6/create_window.py

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget
from PyQt6.QtGui import QIcon
class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        # 设置窗口位置和尺寸
        self.setGeometry(300, 300, 300, 220)
        # 设置窗口标题
        self.setWindowTitle('窗口标题')
        # 设置窗口背景色
        self.setStyleSheet("background-color: red")
        self.show()
if __name__ == '__main__':
    app = QApplication(sys.argv)
    # 设置窗口图标
    app.setWindowIcon(QIcon('images/tray.png'))
    ex = Example()
    sys.exit(app.exec())
```

运行程序，会看到如图 3-1 所示的窗口。在 macOS 中，标题栏图标并不显示在标题栏中，而是将图像显示在 Dock 上，如图 3-1 所示窗口下方最右侧的图标就是本例创建的窗口的标题栏图标。Windows 窗口的标题栏图标会显示在标题栏的左侧，如图 3-2 所示。Linux 比较复杂，使用不同的窗口管理器时，会有不同的效果。有的显示图标，有的不显示图标。

图 3-1　在 macOS 下创建的窗口

图 3-2　在 Windows 下创建的窗口

注意：运行代码之前，要保证 images 目录中有 tray.png 文件。

3.3 布局

PyQt6 提供了多种布局方式，包括水平布局、垂直布局、网格布局和表单布局。水平布局和垂直布局是最常用的两种布局方式，它们分别将组件从左到右或从上到下排列。网格布局将组件放置在一个网格中，而表单布局可以在窗口中放置多个组件，并且可以自动调整组件的大小和位置。在 PyQt6 中，可以使用 QHBoxLayout 和 QVBoxLayout 创建水平布局和垂直布局，使用 QGridLayout 创建网格布局，使用 QFormLayout 创建表单布局。这些布局管理器都是 QWidget 的子类，可以通过 addWidget 方法将组件添加到其中。如果需要在一个布局中添加另一个布局，则可以使用 addLayout 方法。

下面的例子演示了如何使用这 4 种布局。首先用 PyQt6 创建一个窗口，然后将窗口垂直分成 4 个区域，分别演示水平布局、垂直布局、网格布局和表单布局。第 1 个区域使用水平布局，水平放置 5 个按钮。第 2 个区域使用垂直布局，垂直放置 3 个按钮。第 3 个区域使用网格布局，放置 9 个按钮（文本从 1 到 9），排成 3×3 形式。第 4 个区域使用表单布局放几个组件，用于演示使用表单布局。

代码位置：src/gui/pyqt6/layout.py

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget, QGridLayout, QVBoxLayout, QHBoxLayout,
QFormLayout, QPushButton
class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        # 水平布局
        hbox = QHBoxLayout()
        for i in range(5):
            hbox.addWidget(QPushButton(str(i)))
        # 垂直布局
        vbox = QVBoxLayout()
        for i in range(3):
            vbox.addWidget(QPushButton(str(i)))
        # 网格布局
        grid = QGridLayout()
        positions = [(i,j) for i in range(3) for j in range(3)]
        for position, name in zip(positions, ['1', '2', '3', '4', '5', '6', '7', '8', '9']):
            button = QPushButton(name)
            grid.addWidget(button, *position)
        # 表单布局
        form = QFormLayout()
        form.addRow('Name:', QPushButton('Button'))
        form.addRow('Age:', QPushButton('Button'))
        form.addRow('Job:', QPushButton('Button'))
```

```
        # 整体垂直布局
        vbox_all = QVBoxLayout()
        vbox_all.addLayout(hbox)
        vbox_all.addLayout(vbox)
        vbox_all.addLayout(grid)
        vbox_all.addLayout(form)

        self.setLayout(vbox_all)

        self.setGeometry(300, 300, 300, 150)
        self.setWindowTitle('布局演示')
        self.show()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec())
```

运行程序，会看到如图 3-3 所示的窗口。

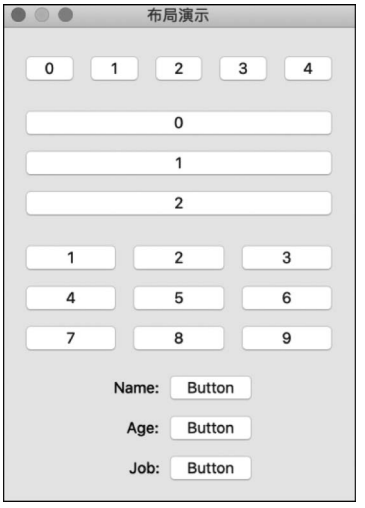

图 3-3　布局

3.4　常用组件

PyQt6 提供了一些常用的组件，如按钮、标签、文本框等，方便用户进行界面设计和交互。PyQt6 中常用的组件包括 QCheckBox（复选框）、QPushButton（按钮）、QSlider（滑杆）、QProgressBar（进度条）、QCalendarWidget（日历）、QComboBox（下拉列表框）、QLabel（标签）、QLineEdit（行文本编辑器）、QTextEdit（文本编辑器）、QSpinBox（整数步长调节器）、QDoubleSpinBox（浮点数步长调节器）、QDateEdit（日期编辑器）、QTimeEdit（时间编辑器）、QDateTimeEdit（日期和时间编辑器）等。

大多数组件的使用方法类似，每一个组件都对应于一个类，类名就是前面给出的组件英文名，如 QLineEdit、QPushButton 等。创建一个组件，其实就是创建这个组件类的对象，然后将组件放到窗口上，例如，下面的代码创建一个 QPushButton 组件。

```
button = QPushButton('Click me', window)
```

下面的例子会创建一个窗口，用 PyQt6 创建一个窗口，并放置 3 个 QLineEdit 组件、3 个 QLabel 组件和 3 个复选框组件。根据这些复选框组件是否选中，控制前面的 QLineEdit 组件是否可以输入文本，如果不可输入文本，将 QLineEdit 组件的背景色设置为红色，否则恢复为白色。在组件的下方还会放置一个按钮组件和一个 QLabel 组件，单击按钮组件，会将这 3 个 QLineEdit 组件中输入的内容组合成一个字符串，显示在最下方的 QLabel 组件中。

代码位置：src/gui/pyqt6/components.py

```
from PyQt6 import QtWidgets, QtCore
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QLineEdit, QCheckBox, QVBoxLayout,
QHBoxLayout, QPushButton
from PyQt6.QtGui import QPalette, QColor
class Window(QWidget):
    def __init__(self):
        super().__init__()
        self.setGeometry(300, 300, 300, 220)
        self.setWindowTitle("组件演示")
        # 创建布局
        layout = QVBoxLayout()

        # 创建姓名行
        name_layout = QHBoxLayout()
        name_label = QLabel('姓名')
        self.name_line_edit = QLineEdit()
        self.name_check_box = QCheckBox()
        self.name_check_box.setChecked(True)
        self.name_check_box.stateChanged.connect(self.on_name_check_box_state_changed)
        name_layout.addWidget(name_label)
        name_layout.addWidget(self.name_line_edit)
        name_layout.addWidget(self.name_check_box)

        # 创建年龄行
        age_layout = QHBoxLayout()
        age_label = QLabel('年龄')
        self.age_line_edit = QLineEdit()
        self.age_check_box = QCheckBox()
        self.age_check_box.setChecked(True)
        self.age_check_box.stateChanged.connect(self.on_age_check_box_state_changed)
        age_layout.addWidget(age_label)
        age_layout.addWidget(self.age_line_edit)
        age_layout.addWidget(self.age_check_box)
```

```
        # 创建收入行
        income_layout = QHBoxLayout()
        income_label = QLabel('收入')
        self.income_line_edit = QLineEdit()
        self.income_check_box = QCheckBox()
        self.income_check_box.setChecked(True)
        self.income_check_box.stateChanged.connect(self.on_income_check_box_state_changed)
        income_layout.addWidget(income_label)
        income_layout.addWidget(self.income_line_edit)
        income_layout.addWidget(self.income_check_box)

        # 添加姓名、年龄、收入到布局中
        layout.addLayout(name_layout)
        layout.addLayout(age_layout)
        layout.addLayout(income_layout)

        # 创建按钮和显示标签
        button = QPushButton('显示')
        button.clicked.connect(self.on_button_clicked)
        self.display_label = QLabel()

        # 添加按钮和显示标签到布局中
        layout.addWidget(button)
        layout.addWidget(self.display_label)

        # 设置窗口布局
        self.setLayout(layout)
    # 姓名复选框的槽函数
    def on_name_check_box_state_changed(self):
        if self.name_check_box.isChecked():
            self.name_line_edit.setReadOnly(False)
            palette = self.name_line_edit.palette()
            palette.setColor(QPalette.ColorRole.Base, QColor('white'))
            self.name_line_edit.setPalette(palette)
        else:
            self.name_line_edit.setReadOnly(True)
            palette = self.name_line_edit.palette()
            palette.setColor(QPalette.ColorRole.Base, QColor('red'))
            self.name_line_edit.setPalette(palette)
    # 年龄复选框的槽函数
    def on_age_check_box_state_changed(self):
        if self.age_check_box.isChecked():
            self.age_line_edit.setReadOnly(False)
            palette = self.age_line_edit.palette()
            palette.setColor(QPalette.ColorRole.Base, QColor('white'))
            self.age_line_edit.setPalette(palette)
        else:
            self.age_line_edit.setReadOnly(True)
            palette = self.age_line_edit.palette()
            palette.setColor(QPalette.ColorRole.Base, QColor('red'))
            self.age_line_edit.setPalette(palette)
```

```
        # 收入复选框的槽函数
        def on_income_check_box_state_changed(self):
            if self.income_check_box.isChecked():
                self.income_line_edit.setReadOnly(False)
                palette = self.income_line_edit.palette()
                palette.setColor(QPalette.ColorRole.Base, QColor('white'))
                self.income_line_edit.setPalette(palette)
            else:
                self.income_line_edit.setReadOnly(True)
                palette = self.income_line_edit.palette()
                palette.setColor(QPalette.ColorRole.Base, QColor('red'))
                self.income_line_edit.setPalette(palette)
        def on_button_clicked(self):
            name = self.name_line_edit.text()
            age = self.age_line_edit.text()
            income = self.income_line_edit.text()
            display_text = f'姓名:{name},年龄:{age},收入:{income}'
            self.display_label.setText(display_text)

    if __name__ == '__main__':
        app = QApplication([])
        window = Window()
        window.show()
        app.exec()
```

运行代码，显示如图 3-4 所示的窗口，在 QLineEdit 组件中输入一些文本，单击“显示”按钮，这 3 个 TLineEdit 组件中输入的内容就会显示在按钮下方。

图 3-4　组件演示

3.5　列表组件（QListWidget）

PyQt6 有两个列表组件——QListView 和 QListWidget。QListView 是一个用于显示列表的组件，采用了 MVC 模式。与 QListWidget 不同，它不会自动创建一个内部模型来管理列表项。相反地，需要手动创建一个模型，并将其与视图关联。这使得 QListView 更加灵活，因为可以使用任何类型的模型来管理数据。向 QListView 列表中添加列表项的代码如下：

```
list_view = QListView()
# 创建字符串列表模型,用于向列表中添加字符串列表项
model = QStringListModel()
model.setStringList(['Item 1', 'Item 2', 'Item 3'])
# 为 QListView 组件设置模型
list_view.setModel(model)
```

QListWidget 是 QListView 的升级版本，已经建立了一个数据存储模型 QListWidgetItem，操作方便，直接调用 addItem 方法即可添加列表项。而 QListView 是基于 Model 的，需要自己建模（如建立 QStringListModel、QSqlTableModel 等），保存数据，这样就大大降低了数据冗余，提高了程序的效率，但是需要我们对数据建模有一定的了解。向 QListWidget 列表中添加列表项的代码如下：

```
list_widget = QListWidget()
item1 = QListWidgetItem("item1")
list_widget.addItem(item1)
item2 = QListWidgetItem("item2")
list_widget.addItem(item2)
```

下面的例子在窗口上放置一个 QListWidget 组件和两个按钮（“添加”按钮和“删除”按钮），QListWidget 组件默认添加 6 个列表项。单击“添加”按钮，会向 QListWidget 组件中添加 1 个新的列表项，如果某一个列表项被选中，单击“删除”按钮，会删除当前的列表项。

代码位置：src/gui/pyqt6/list.py

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QPushButton,
QListWidget, QListWidgetItem
class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        # 创建一个 QListWidget 组件
        self.listWidget = QListWidget(self)
        # 创建两个按钮
        self.addButton = QPushButton('添加', self)
        self.delButton = QPushButton('删除', self)
        # 将按钮添加到水平布局中
        hbox = QHBoxLayout()
        hbox.addWidget(self.addButton)
        hbox.addWidget(self.delButton)
        # 将 QListWidget 和水平布局添加到垂直布局中
        vbox = QVBoxLayout()
        vbox.addWidget(self.listWidget)
        vbox.addLayout(hbox)
        # 设置窗口布局
        self.setLayout(vbox)
```

```
            # 添加 6 个列表项
            for i in range(6):
                item = QListWidgetItem('列表项 %d' % i)
                self.listWidget.addItem(item)
            # 连接按钮的单击事件
            self.addButton.clicked.connect(self.addItem)
            self.delButton.clicked.connect(self.delItem)

            # 设置窗口大小和标题
            self.setGeometry(300, 300, 350, 300)
            self.setWindowTitle('QListWidget')
        def addItem(self):
            # 向 QListWidget 中添加一个列表项
            item = QListWidgetItem('新列表项')
            self.listWidget.addItem(item)

        def delItem(self):
            # 获取当前选中的列表项
            currentItem = self.listWidget.currentItem()
            if currentItem is not None:
                # 删除当前选中的列表项
                row = self.listWidget.row(currentItem)
                self.listWidget.takeItem(row)
    if __name__ == '__main__':
        app = QApplication(sys.argv)
        ex = Example()
        ex.show()
        sys.exit(app.exec())
```

运行程序，会显示如图 3-5 所示的窗口，读者可以单击“添加”按钮和“删除”按钮来体验 QListWidget 组件插入和删除列表项的效果。

图 3-5　列表组件

3.6　下拉列表组件（QComboBox）

QComboBox 是一个下拉列表组件，允许用户从预定义的选项中选择一个值。使用 QComboBox.addItem 方法可以向该组件中添加列表项，代码如下：

```
combo_box.addItem('New Item')
```

如果想删除当前选定的选项，可以使用 removeItem 方法，代码如下：

```
combo_box.removeItem(combo_box.currentIndex())
```

下面的例子在窗口上放置一个 QComboBox 组件，里面默认添加 3 个 item。QComboxBox 组件下方水平放置两个组件（Add 按钮和 Delete 按钮），单击 Add 按钮，向 QComboxBox 组件添加一个列表项。单击 Delete 按钮，会删除当前选中的列表项。

代码位置：src/gui/pyqt6/combobox.py

```
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QPushButton, QComboBox
import sys
class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        vbox = QVBoxLayout()
        hbox = QHBoxLayout()
        # 创建 QComboBox 组件
        combo_box = QComboBox()
        # 向 QComboBox 组件添加 3 个默认列表项
        combo_box.addItems(['Item 1', 'Item 2', 'Item 3'])
        vbox.addWidget(combo_box)

        add_button = QPushButton('Add')
        add_button.clicked.connect(lambda: combo_box.addItem('New Item'))
        hbox.addWidget(add_button)

        remove_button = QPushButton('Remove')
        remove_button.clicked.connect(lambda: combo_box.removeItem(combo_box.currentIndex()))
        hbox.addWidget(remove_button)

        vbox.addLayout(hbox)

        self.setLayout(vbox)
        self.setGeometry(300, 300, 350, 250)
        self.setWindowTitle('QComboBox Example')
        self.show()
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec())
```

运行程序，单击 Add 按钮会添加新的列表项，单击 Remove 按钮会删除当前列表项。如图 3-6 所示。

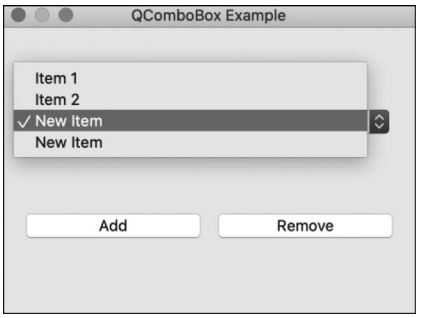

图 3-6　下拉列表组件

3.7　表格组件(QTableWidget)

QTableWidget 是 PyQt6 中的一个表格组件，可以用于显示和编辑表格数据。QTableWidget 是一个基于 QTableView 的类，提供了一些额外的功能，如行和列的插入和删除，以及单元格的合并。若要添加表头，可以使用 QTableWidget 的 setHorizontalHeaderLabels 方法来设置水平表头标签，使用 setVerticalHeaderLabels 方法来设置垂直表头标签。若要添加表格项，可以使用 setItem 方法来设置单元格的内容。若要删除表格项，可以使用 removeRow 或 removeColumn 方法来删除行或列。

在下面的例子中，在窗口放置一个 QTableWidget 组件和两个按钮(“添加”按钮和“删除”按钮)。两个按钮要放到 QTableWidget 组件的下面，默认添加 4 列的两个表格项。单击“添加”按钮，会添加一个新表格行；单击“删除”按钮，会删除当前选中的表格行。

代码位置：src/gui/pyqt6/tablewidget. py

```
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
class Table(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('QTableWidget 组件演示')
        self.resize(500, 300)
        # 创建表格
        self.table = QTableWidget()
        self.table.setColumnCount(4)
        self.table.setHorizontalHeaderLabels(['header1', 'header2', 'header3', 'header4'])
        self.table.setRowCount(2)
        for i in range(self.table.rowCount()):
            for j in range(self.table.columnCount()):
                item = QTableWidgetItem(str(i + j))
                self.table.setItem(i, j, item)

        # 创建添加和删除按钮
        add_button = QPushButton('添加')
```

```
        add_button.clicked.connect(self.add_row)
        delete_button = QPushButton('删除')
        delete_button.clicked.connect(self.delete_row)

        # 创建布局
        button_layout = QHBoxLayout()
        button_layout.addWidget(add_button)
        button_layout.addWidget(delete_button)
        table_layout = QVBoxLayout()
        table_layout.addWidget(self.table)
        table_layout.addLayout(button_layout)

        # 设置布局
        self.setLayout(table_layout)

    def add_row(self):
        row_count = self.table.rowCount()
        self.table.insertRow(row_count)
        for j in range(self.table.columnCount()):
            item = QTableWidgetItem(str(row_count + j))
            self.table.setItem(row_count, j, item)

    def delete_row(self):
        row = self.table.currentRow()
        if row != -1:
            self.table.removeRow(row)

if __name__ == '__main__':
    app = QApplication([])
    app.setFont(QFont('Helvetica'))
    table = Table()
    table.show()
    app.exec()
```

运行程序，会看到如图 3-7 所示的界面，单击“添加”按钮会添加新的行，单击“删除”按钮，会删除当前行。

图 3-7　表格组件

3.8 树形组件(QTreeWidget)

QTreeWidget 是 PyQt6 中的一个树形组件,用于显示树形结构的数据。QTreeWidget 类根据预设的模型显示树中的数据。QTreeWidget 使用类似于 QListView 类的方式提供一种典型的基于 item 的树形交互类,该类基于 QT 的"模型/视图"结构,提供了默认的模型来支撑 item 的显示,这些 item 类为 QTreeWidgetItem 类。如果不需要灵活的"模型/视图"框架,可以使用 QTreeWidget 来创建有层级关系的树形结构。如果把标准 item 模型结合 QTreeView 使用,可以得到更灵活的使用方法,从而把"数据"和"显示"分离开。QTreeWidget 组件可以添加多列:使用 setColumnCount 方法设置列数,使用 setHeaderLabels 方法设置列标签。

下面的例子在窗口上放置一个 QTreeWidget 组件和两个按钮(Add 按钮和 Delete 按钮)。QTreeWidget 组件默认会随机添加一些节点。单击 Add 按钮,会在当前选中的节点添加一个子节点,单击 Delete 按钮,会删除当前选中的节点。

代码位置:src/gui/pyqt6/treewidget.py

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtGui import QFont

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("QTreeWidget 组件演示")
        # 创建树形组件
        self.tree_widget = QTreeWidget()
        self.tree_widget.setColumnCount(2)
        self.tree_widget.setHeaderLabels(["Name", "Value"])

        # 创建 Add 按钮
        self.add_button = QPushButton("Add")
        self.add_button.clicked.connect(self.add_node)

        # 创建 Delete 按钮
        self.delete_button = QPushButton("Delete")
        self.delete_button.clicked.connect(self.delete_node)

        # 创建水平布局
        self.layout = QHBoxLayout()
        self.layout.addWidget(self.tree_widget)
        self.layout.addWidget(self.add_button)
        self.layout.addWidget(self.delete_button)

        # 设置布局
        self.setLayout(self.layout)
```

```
            # 随机添加一些节点
            for i in range(10):
                item = QTreeWidgetItem([f"Node {i}", f"Value {i}"])
                self.tree_widget.addTopLevelItem(item)
        # 添加节点
        def add_node(self):
            # 获取当前选中的节点
            current_item = self.tree_widget.currentItem()
            if not current_item:
                return

            # 创建新的子节点
            new_item = QTreeWidgetItem(["New Node", "New Value"])
            current_item.addChild(new_item)
        # 删除节点
        def delete_node(self):
            # 获取当前选中的节点
            current_item = self.tree_widget.currentItem()
            if not current_item:
                return

            # 删除节点
            current_item.parent().removeChild(current_item)


    if __name__ == '__main__':
        app = QApplication(sys.argv)
        app.setFont(QFont('Helvetica'))
        main_window = MainWindow()
        main_window.show()
        sys.exit(app.exec())
```

运行程序，会显示如图 3-8 所示的窗口，单击 Add 按钮，会在当前选中的节点下面添加子节点；单击 Delete 按钮，会将选中的节点（包括该节点的子节点）删除。

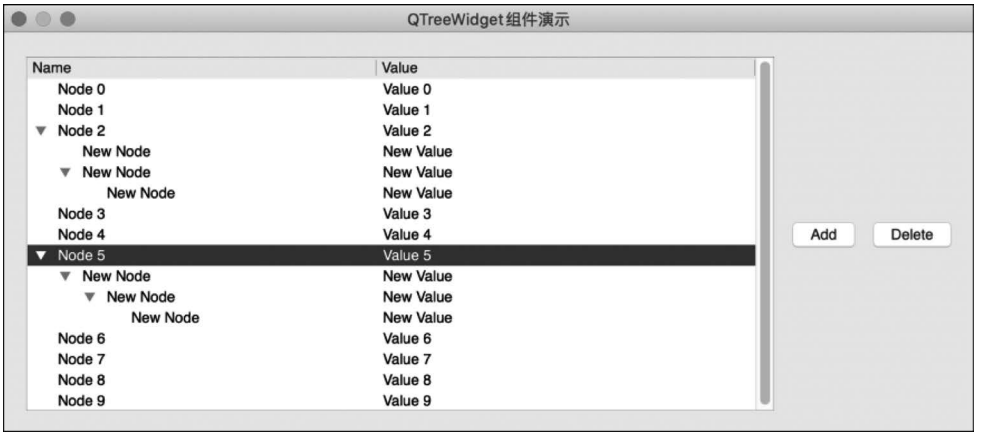

图 3-8　树形组件

3.9 菜单

PyQt6 支持的主要菜单类型是主菜单和弹出菜单。其中，主菜单是应用程序的主要导航方式，通常包含文件、编辑、视图、帮助等菜单项。弹出菜单是在用户请求时显示的菜单，通常包含与当前上下文相关的操作，一般在组件上右击鼠标，弹出菜单就会显示出来。

要实现这两种菜单，可以使用 QMenuBar 和 QMenu 类。QMenuBar 类提供了一个水平菜单，可以在其中添加 QMenu 对象。QMenu 类提供了一个垂直菜单，可以在其中添加 QAction 对象。

下面的例子在窗口放置一个标签组件，然后创建一个主菜单，菜单项包括新建、打开、保存、分割线、退出。单击“退出”菜单项，会退出程序；单击其他菜单项，会将菜单项文本显示在标签组件中。右击窗口或标签，会显示与主菜单一模一样的弹出菜单。菜单项的动作也完全相同。

代码位置：src/gui/pyqt6/menu_demo.py

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
from PyQt6.QtCore import *
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setGeometry(200,200,200,200)
        # 创建标签组件
        self.label = QLabel(self)
        self.label.setText("这是一个标签")
        self.label.setGeometry(50, 50, self.width(), 30)
        self.label.setContextMenuPolicy(Qt.ContextMenuPolicy.CustomContextMenu)
        self.label.customContextMenuRequested.connect(self.showMenu)
        # 创建主菜单
        menu_bar = self.menuBar()
        file_menu = menu_bar.addMenu("文件")
        # 向主菜单添加
        self.addMenuItem(file_menu)
    # 显示弹出菜单
    def showMenu(self, pos):
        menu = self.createPopupMenu()
        menu.exec(self.mapToGlobal(pos))
    # 添加菜单项
    def addMenuItem(self,menu):
        # 创建弹出菜单的菜单项
        new_action = QAction("新建", self)
        open_action = QAction("打开", self)
        save_action = QAction("保存", self)
        separator_action = QAction(self)
```

```
            separator_action.setSeparator(True)
            exit_action = QAction("退出", self)

            # 将菜单项添加到弹出菜单中
            menu.addAction(new_action)
            menu.addAction(open_action)
            menu.addAction(save_action)
            menu.addAction(separator_action)
            menu.addAction(exit_action)

            # 连接退出菜单项的信号和槽函数
            exit_action.triggered.connect(self.close)

            # 连接其他菜单项的信号和槽函数
            new_action.triggered.connect(lambda: self.label.setText("新建菜单项被单击"))
            open_action.triggered.connect(lambda: self.label.setText("打开菜单项被单击"))
            save_action.triggered.connect(lambda: self.label.setText("保存菜单项被单击"))
        # 创建弹出菜单
        def createPopupMenu(self):
            menu = QMenu(self)
            # 向弹出菜单添加菜单项
            self.addMenuItem(menu)
            return menu
        # 关闭应用程序
        def closeEvent(self, event):
            event.accept()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

运行程序，会显示一个窗口。如果在 macOS 系统中，主菜单会显示在 macOS 的菜单栏上，如图 3-9 所示。如果在 Windows 系统中或 Linux 系统中，主菜单会直接显示在窗口上。

在窗口或标签组件上右击鼠标，会显示如图 3-10 所示的弹出菜单。

图 3-9　主菜单和弹出菜单

图 3-10　弹出菜单

在前面的代码中，使用 QMainWindow.createPopupMenu 方法创建窗口的弹出菜单。当主窗口接收到上下文菜单事件时，会自动调用此方法生成一个菜单，并返回这个菜单（Menu 对

象)。通常会在 QMainWindow 的子类中重写 createPopupMenu 方法来创建弹出菜单。

另外,由于主菜单与弹出菜单完全相同,所以本例使用 addMenuItem 函数为主菜单和弹出菜单添加菜单项。

3.10 对话框

PyQt6 支持多种对话框类型,常用的对话框包括 QMessageBox、QInputDialog、QFileDialog、QColorDialog、QFontDialog、QPrintDialog 和 QProgressDialog 等。下面分别介绍这些对话框的功能和使用方法。

1. QMessageBox

QMessageBox 是最常用的对话框之一,可以用于显示消息、警告和错误等信息。使用 QMessageBox 的方法如下:

```
msgBox = QMessageBox()
msgBox.setIcon(QMessageBox.Information)
msgBox.setText("This is a message box")
msgBox.setWindowTitle("Message Box")
msgBox.setStandardButtons(QMessageBox.StandardButton.Ok | QMessageBox.StandardButton.Cancel)
msgBox.exec()
```

上述代码中,setIcon 方法用于设置图标,setText 方法用于设置文本,setWindowTitle 方法用于设置标题,setStandardButtons 方法用于设置消息对话框的按钮种类,exec 方法用于显示对话框。

setStandardButtons 方法可以设置的按钮种类如下。

(1) QMessageBox.StandardButton.Ok:确定。
(2) QMessageBox.StandardButton.Open:打开。
(3) QMessageBox.StandardButton.Save:保存。
(4) QMessageBox.StandardButton.Cancel:取消。
(5) QMessageBox.StandardButton.Close:关闭。
(6) QMessageBox.StandardButton.Discard:放弃。
(7) QMessageBox.StandardButton.Apply:应用。
(8) QMessageBox.StandardButton.Reset:重置。
(9) QMessageBox.StandardButton.RestoreDefaults:恢复默认值。
(10) QMessageBox.StandardButton.Help:帮助。
(11) QMessageBox.StandardButton.SaveAll:全部保存。
(12) QMessageBox.StandardButton.Yes:是。
(13) QMessageBox.StandardButton.YesToAll:全部是。
(14) QMessageBox.StandardButton.No:否。
(15) QMessageBox.StandardButton.NoToAll:全部否。

（16）QMessageBox.StandardButton.Abort：中止。

（17）QMessageBox.StandardButton.Retry：重试。

（18）QMessageBox.StandardButton.Ignore：忽略。

上述代码中，setStandardButtons 方法设置了两个按钮——OK 和 Cancel。这些按钮的作用是由按钮的角色决定的，例如，OK 按钮的角色是 AcceptRole，Cancel 按钮的角色是 RejectRole。如果用户单击了 OK 按钮，则返回 QMessageBox.Ok；如果用户单击了 Cancel 按钮，则返回 QMessageBox.Cancel。

2. QInputDialog

QInputDialog 可以用于获取用户输入的信息。使用 QInputDialog 的方法如下：

```
text, ok = QInputDialog.getText(None, "Input Dialog", "Enter your name:")
if ok:
    print(f"Your name is {text}")
```

上述代码中，getText 方法用于弹出一个输入对话框，第 1 个参数为父窗口，第 2 个参数为标题，第 3 个参数为提示信息。如果用户单击 OK 按钮，则返回输入的文本和 True；否则，返回空字符串和 False。

3. QFileDialog

QFileDialog 可以用于打开和保存文件。使用 QFileDialog 的方法如下：

```
filename, _ = QFileDialog.getOpenFileName(None, "Open File", "", "Text Files (*.txt);;All
Files (*)")
if filename:
    print(f"Filename is {filename}")
```

上述代码中，getOpenFileName 方法用于弹出一个打开文件对话框，第 1 个参数为父窗口，第 2 个参数为标题，第 3 个参数为默认目录，第 4 个参数为文件过滤器。如果用户选择了文件，则返回文件名和 True；否则，返回空字符串和 False。

4. QColorDialog

QColorDialog 可以用于选择颜色。使用 QColorDialog 的方法如下：

```
color = QColorDialog.getColor()
if color.isValid():
    print(f"Selected color: {color.name()}")
```

上述代码中，getColor 方法用于弹出一个颜色选择对话框。如果用户选择了颜色，则返回颜色对象；否则，返回无效的颜色对象。

5. QFontDialog

QFontDialog 可以用于选择字体。使用 QFontDialog 的方法如下：

```
font, ok = QFontDialog.getFont()
if ok:
    print(f"Selected font: {font.family()}, {font.pointSize()}pt")
```

上述代码中，getFont 方法用于弹出一个字体选择对话框。如果用户选择了字体，则返回字体对象和 True；否则，返回默认字体对象和 False。

6. QPrintDialog

QPrintDialog 类提供了一个打印对话框，用于选择打印机和打印选项。它的构造方法原型如下：

```
QPrintDialog(parent: QWidget = None, flags: Union[Qt.WindowFlags, Qt.WindowType] =
Qt.WindowFlags())
```

其中，parent 参数表示可选的 QWidget 对象，flags 参数表示可选的窗口标志。使用 QPrintDialog 类的方法如下：

```
printer = QPrinter(QPrinter.HighResolution)
dialog = QPrintDialog(printer)
if dialog.exec_() == QDialog.Accepted:
    # 打印文档
```

7. QProgressDialog

QProgressDialog 类提供了一个进度对话框，用于显示长时间运行的操作的进度。它的构造方法如下：

```
QProgressDialog(labelText: str, cancelButton: Union[QWidget, NoneType] = None, minimum:
int = 0, maximum: int = 100, parent: QWidget = None, flags: Union[Qt.WindowFlags,
Qt.WindowType] = Qt.WindowFlags())
```

其中，labelText 参数是进度对话框中显示的文本；cancelButton 参数是可选的 QWidget 对象，用于取消操作；minimum 和 maximum 参数分别是进度条的最小值和最大值；parent 参数是可选的 QWidget 对象；flags 参数是可选的窗口标志。使用 QProgressDialog 类的方法如下：

```
progress_dialog = QProgressDialog("Copying files...", "Cancel", 0, len(files), self)
progress_dialog.setWindowModality(Qt.WindowModal)
for i, file in enumerate(files):
    if progress_dialog.wasCanceled():
        break
    progress_dialog.setValue(i)
    # 复制文件
progress_dialog.close()
```

下面的例子完整地演示了这 7 种对话框的使用方法，单击相应的按钮，会有对应的对话框弹出，选择后，会在按钮下方的标签组件中看到选择的结果。

代码位置：src/gui/pyqt6/dialogs.py

```
import sys
from PyQt6.QtWidgets import *
```

```
from PyQt6.QtPrintSupport import QPrintDialog
class Example(QWidget):
    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        vbox = QVBoxLayout()

        # 添加 4 个按钮
        btn1 = QPushButton('QMessageBox', self)
        btn2 = QPushButton('QInputDialog', self)
        btn3 = QPushButton('QFileDialog', self)
        btn4 = QPushButton('QColorDialog', self)
        btn5 = QPushButton('QFontDialog', self)
        btn6 = QPushButton('QPrintDialog', self)
        btn7 = QPushButton('QProgressDialog', self)
        # 添加 4 个标签
        self.label1 = QLabel(self)
        self.label2 = QLabel(self)
        self.label3 = QLabel(self)
        self.label4 = QLabel(self)
        self.label5 = QLabel(self)
        self.label6 = QLabel(self)
        # 将 4 个标签添加到垂直布局中
        vbox.addWidget(btn1)
        vbox.addWidget(self.label1)
        vbox.addWidget(btn2)
        vbox.addWidget(self.label2)
        vbox.addWidget(btn3)
        vbox.addWidget(self.label3)
        vbox.addWidget(btn4)
        vbox.addWidget(self.label4)
        vbox.addWidget(btn5)
        vbox.addWidget(self.label5)
        vbox.addWidget(btn6)
        vbox.addWidget(self.label6)
        vbox.addWidget(btn7)

        # 连接 4 个按钮到槽函数
        btn1.clicked.connect(lambda: self.showMessageBox("messagebox"))
        btn2.clicked.connect(lambda: self.showInputDialog())
        btn3.clicked.connect(lambda: self.showOpenFileDialog())
        btn4.clicked.connect(lambda: self.showColorDialog())
        btn5.clicked.connect(lambda: self.showFontDialog())
        btn6.clicked.connect(lambda: self.showPrintDialog())
        btn7.clicked.connect(lambda: self.showProgressDialog())
        self.setLayout(vbox)
    # 显示消息对话框
    def showMessageBox(self, text):
```

```
            sender = self.sender()
            msgBox = QMessageBox()
            msgBox.setWindowTitle("Message Box")

            msgBox.setText(f"The {sender.text()} button has been pressed.")
            msgBox.setInformativeText(f"You selected {text}")
            msgBox.setStandardButtons(QMessageBox.StandardButton.Ok | QMessageBox.StandardButton.Cancel)
            msgBox.setDefaultButton(QMessageBox.StandardButton.Ok)
            ret = msgBox.exec()
            if ret == QMessageBox.StandardButton.Ok:
                self.label1.setText("ok")
            else:
                self.label1.setText("cancel")
        # 显示输入对话框
        def showInputDialog(self):
            ret = QInputDialog.getText(self, 'Input Dialog', 'Enter your name:')[0]
            self.label2.setText(ret)
        # 显示打开文件对话框
        def showOpenFileDialog(self):
            ret = QFileDialog.getOpenFileName(self)[0]
            self.label3.setText(ret)
        # 显示颜色对话框
        def showColorDialog(self):
            col = QColorDialog.getColor()
            if col.isValid():
                self.label4.setStyleSheet(f"background-color: {col.name()}")
        # 显示字体对话框
        def showFontDialog(self):
            font, okPressed = QFontDialog.getFont()
            if okPressed:
                self.label5.setText(font.toString())
        # 显示打印对话框
        def showPrintDialog(self):
            printDialog = QPrintDialog()
            if printDialog.exec() == QDialog.DialogCode.Accepted:
                self.label6.setText("Success")
        # 显示进度条对话框
        def showProgressDialog(self):
                progressDialog = QProgressDialog(self)
                progressDialog.setLabelText("Copying files...")
                progressDialog.setRange(0, 100)
                progressDialog.setValue(47)
                progressDialog.show()
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec())
```

运行程序，会显示如图 3-11 所示的窗口。如果单击某一个按钮，会有相应的对话框弹出，完成选择（或输入）后，就会在对应按钮下方看到相应的内容。

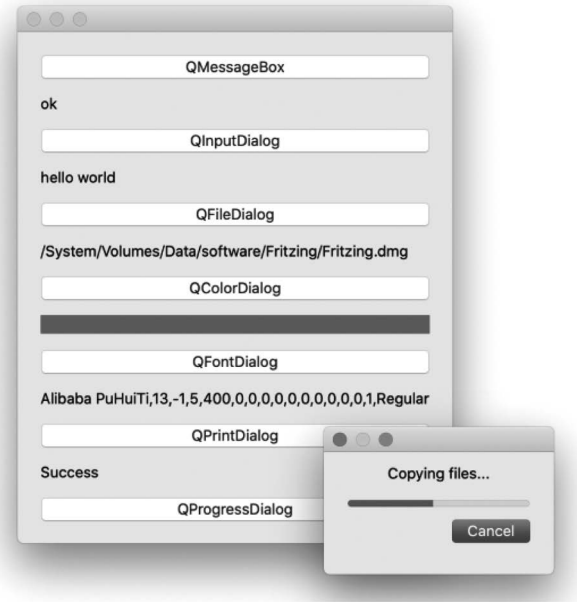

图 3-11　对话框示例

这 7 种对话框在 macOS、Windows 和 Linux 下的显示效果并不相同，但其功能是一样的。例如，图 3-12 是 macOS 的“打开文件”对话框。图 3-13 是 macOS 的“选择颜色”对话框。

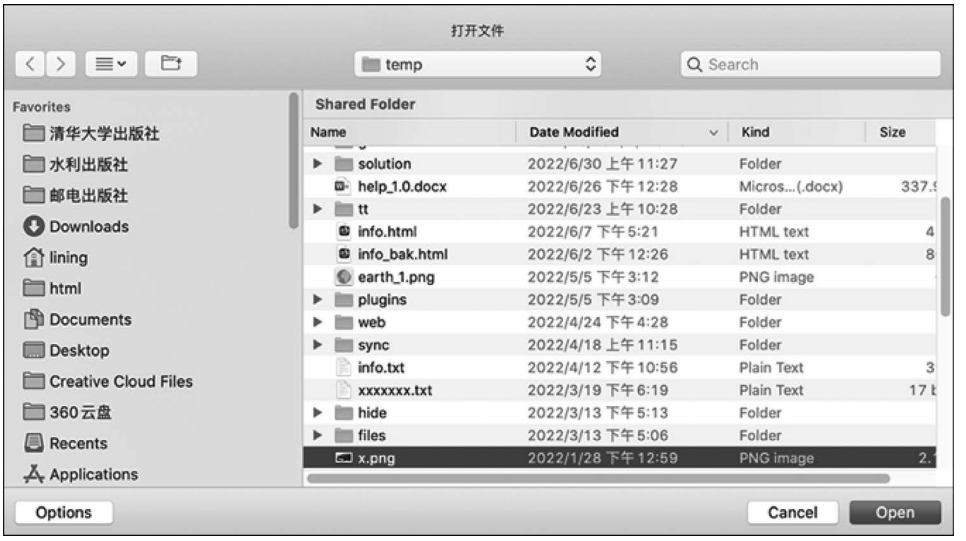

图 3-12　“打开文件”对话框

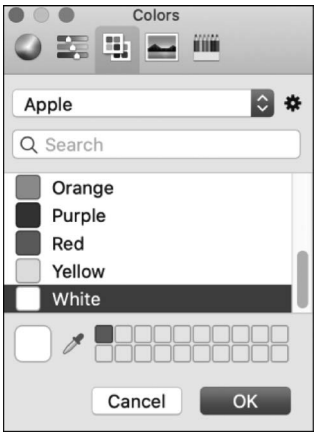

图 3-13 “选择颜色”对话框

3.11 自由绘画

本节会使用 Canvas 组件实现一个有趣的功能：自由绘画。基本功能是可以通过鼠标在窗口上画出任意平滑的曲线，并可以设置曲线的颜色，还可以将绘制的曲线保存成图像文件。

可以使用 PyQt6 的 QPainter 和 QPen 类来实现绘图功能。QPainter 类用于在 QWidget 组件上执行绘图操作，其功能类似于一个绘图工具，为大部分图形界面提供了高度优化的函数，使 QPainter 类可以绘制从简单的直线到复杂的图形等。QPen 类则用于设置画笔的颜色、宽度等属性。使用 QColorDialog 类来弹出“选择颜色”对话框，以设置画笔的颜色。可以在 QWidget 组件上重载 paintEvent 函数，并在该函数中使用 QPainter 类进行绘制。使用 mousePressEvent 和 mouseMoveEvent 函数来实现鼠标自由绘制线条。要实现平滑曲线，可以使用贝塞尔曲线或样条曲线等算法。

下面的例子将完整地实现这个绘图程序，通过单击窗口下方的“画笔颜色”按钮，可以弹出“选择颜色”对话框，并选择画笔的颜色。移动鼠标就可以在按钮下方的区域自由绘制曲线。单击“保存图像”按钮，会弹出“保存文件”对话框，输入文件名，就能将绘制的图形保存成图像文件。

代码位置：src/gui/pyqt6/drawing. py

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
from PyQt6.QtCore import *

class Window(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("画图程序")
        self.setGeometry(100, 100, 500, 500)
```

```
        self.setFixedSize(500, 500)
        layout = QVBoxLayout()
        self.setLayout(layout)
        self.pen_color = Qt.GlobalColor.black
        self.pen_width = 3
        self.canvas = QLabel()
        layout.addWidget(self.canvas)
        pixmap = QPixmap(500, 450)
        pixmap.fill(Qt.GlobalColor.white)
        self.canvas.setPixmap(pixmap)
        color_button = QPushButton("画笔颜色")
        color_button.clicked.connect(self.select_color)
        layout.addWidget(color_button, alignment=Qt.AlignmentFlag.AlignHCenter)

        save_button = QPushButton("保存图像")
        save_button.clicked.connect(self.save_image)
        layout.addWidget(save_button, alignment=Qt.AlignmentFlag.AlignHCenter)
        self.last_x, self.last_y = None, None
    # 选择颜色
    def select_color(self):
        color = QColorDialog.getColor(initial=self.pen_color)
        if color.isValid():
            self.pen_color = color
    # 鼠标移动时触发
    def mouseMoveEvent(self, e):
        if self.last_x is None:
            self.last_x = e.position().x()
            self.last_y = e.position().y()
            return
        pixmap = self.canvas.pixmap().copy()
        painter = QPainter(pixmap)
        pen = QPen(self.pen_color, self.pen_width, Qt.PenStyle.SolidLine, Qt.PenCapStyle.RoundCap,
                   Qt.PenJoinStyle.RoundJoin)
        painter.setPen(pen)
        painter.drawLine(int(self.last_x - 15), int(self.last_y ), int(e.position().x() - 15),
int(e.position().y()))
        painter.end()
        self.canvas.setPixmap(pixmap)
        self.last_x = e.position().x()
        self.last_y = e.position().y()

    def mouseReleaseEvent(self, e):
        self.last_x = None
        self.last_y = None
    def save_image(self):
        file_name, _ = QFileDialog.getSaveFileName(self, "保存图像", "", "PNG Image (*.png)")
        if file_name:
            self.canvas.pixmap().save(file_name)
app = QApplication(sys.argv)
window = Window()
window.show()
app.exec()
```

运行程序，在显示的窗口上绘制一些曲线，并设置画笔的颜色，效果如图 3-14 所示。

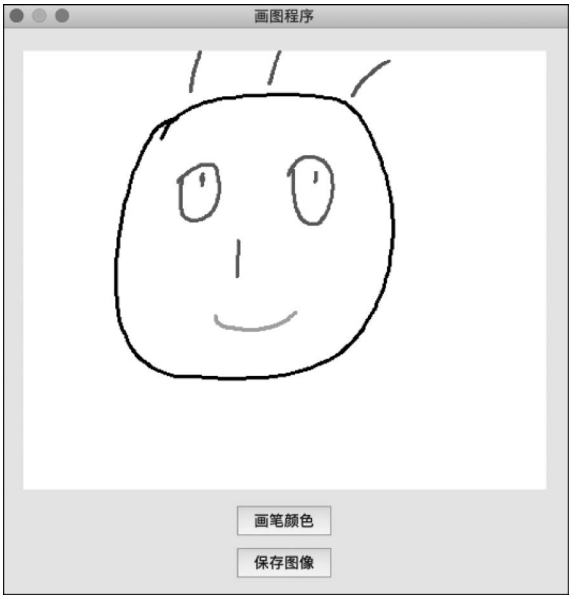

图 3-14　画图程序

3.12　图像旋转器

本节会通过 PyQt6 实现一个可以打开图像、旋转图像和保存旋转结果的项目。实现这个项目涉及如下 4 种技术。

- 打开图像；
- 显示图像；
- 旋转图像；
- 保存图像。

这 4 种技术可以完全通过 PyQt6 提供的 API 实现。

1. 打开图像

使用 QPixmap 对象可以打开图像文件，代码如下：

```
pixmap = QtGui.QPixmap('/resources/file.png')
```

2. 显示图像

可以使用 QLabel 组件显示图像，通过 QLabel.setPixmap 方法为标签组件指定 QPixmap 对象，代码如下：

```
image_label = QtWidgets.QLabel(self)
image_label.setPixmap(self.pixmap)
```

3. 旋转图像

使用 QTransform.rotate 方法可以设置图像的旋转角度。如果 rotate 方法的参数值是正数，为顺时针旋转；如果为负数，则为逆时针旋转。其实 QTransform.rotate 方法只是设置旋转角度，需要调用 QPixmap.transformed 方法，并传入 QTransform 对象，才会通过 QPixmap.transformed 方法返回旋转后的图像，最后再将旋转后的图像重新显示在 QLabel 组件上，代码如下：

```
# 顺时针旋转 40 度
transform = QtGui.QTransform().rotate(40)
pixmap = pixmap.transformed(transform)
image_label.setPixmap(pixmap)
```

4. 保存图像

通过 QPixmap.save 方法可以保存图像，代码如下：

```
pixmap = image_label.pixmap()
pixmap.save('/resources/myimage.png')
```

下面是这个例子的完整代码，单击“打开图像”按钮会弹出“打开图像”对话框，选择图像文件后，会在窗口下方显示该图像，如图 3-15 所示。单击“逆时针旋转”按钮，会弹出一个“输入文本”对话框，输入旋转角度后，图像会向逆时针旋转一定角度，如图 3-16 所示。“顺时针旋转”按钮也有类似的操作。最后，单击“保存图像”按钮，会弹出“保存图像”对话框，输入文件名后，会将旋转后的图像保存为新的图像文件。读者可以打开旋转后的图像，效果如图 3-17 所示。

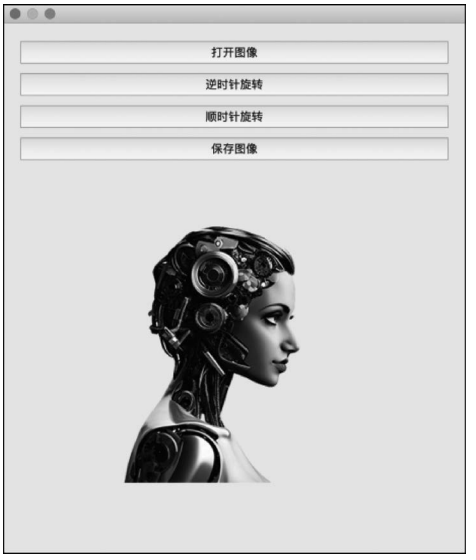

图 3-15 打开图像

图 3-16 逆时针旋转图像

图 3-17　旋转后的图像

代码位置：src/gui/pyqt6/rotate_image.py

```
import sys
from PyQt6 import QtWidgets, QtGui, QtCore
from PyQt6.QtWidgets import QFileDialog

class Window(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        # 设置窗口大小
        self.resize(500, 500)

        # 创建打开图像按钮
        self.open_button = QtWidgets.QPushButton('打开图像', self)
        self.open_button.clicked.connect(self.open_image)

        # 创建逆时针旋转按钮
        self.rotate_ccw_button = QtWidgets.QPushButton('逆时针旋转', self)
        self.rotate_ccw_button.clicked.connect(self.rotate_ccw)

        # 创建顺时针旋转按钮
        self.rotate_cw_button = QtWidgets.QPushButton('顺时针旋转', self)
        self.rotate_cw_button.clicked.connect(self.rotate_cw)

        # 创建保存图像按钮
        self.save_button = QtWidgets.QPushButton('保存图像', self)
        self.save_button.clicked.connect(self.save_image)

        # 创建用于显示图像的组件
        self.image_label = QtWidgets.QLabel(self)
        self.image_label.setAlignment(QtCore.Qt.AlignmentFlag.AlignCenter)

        # 创建布局并添加组件
        layout = QtWidgets.QVBoxLayout(self)
        layout.addWidget(self.open_button)
        layout.addWidget(self.rotate_ccw_button)
        layout.addWidget(self.rotate_cw_button)
        layout.addWidget(self.save_button)
        layout.addWidget(self.image_label)
```

```
    def open_image(self):
        # 打开文件对话框选择图像文件
        file_name, _ = QFileDialog.getOpenFileName(self, '打开图像', '', '图像文件 (*.png
*.jpg *.jpeg *.bmp)')

        if file_name:
            # 加载并显示图像
            self.pixmap = QtGui.QPixmap(file_name)
            self.image_label.setPixmap(self.pixmap)

    def rotate_ccw(self):
        # 弹出输入对话框获取旋转角度
        angle, ok = QtWidgets.QInputDialog.getInt(self, '逆时针旋转', '输入旋转角度(度)')
        if ok:
            # 逆时针旋转图像并显示结果
            transform = QtGui.QTransform().rotate(-angle)
            pixmap = self.pixmap.transformed(transform)
            self.image_label.setPixmap(pixmap)
    def rotate_cw(self):
        # 弹出输入对话框获取旋转角度
        angle, ok = QtWidgets.QInputDialog.getInt(self, '顺时针旋转', '输入旋转角度(度)')
        if ok:
            # 顺时针旋转图像并显示结果
            transform = QtGui.QTransform().rotate(angle)
            pixmap = self.pixmap.transformed(transform)
            self.image_label.setPixmap(pixmap)
    def save_image(self):
        # 打开保存文件对话框选择保存位置
        file_name, _ = QFileDialog.getSaveFileName(self, '保存图像', '', 'PNG 图像
(*.png);;JPEG 图像 (*.jpg *.jpeg);;BMP 图像 (*.bmp)')
        if file_name:
            # 保存图像
            pixmap = self.image_label.pixmap()
            pixmap.save(file_name)
if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    window = Window()
    window.show()
    sys.exit(app.exec())
```

3.13　点对点聊天

点对点聊天的关键点就是建立两端的通信，Python 通过 Socket 在两台机器之间建立连接和互相传递文本数据的基本步骤如下。

(1) 一台机器作为服务器端，创建一个 Socket 对象，绑定一个 IP 地址和端口号，然后调用 listen 方法开始监听客户端的连接请求。

示例代码如下：

```
s = socket.socket()                    # 创建 socket 对象
host = socket.gethostname()            # 获取本机名称
port = 12345                           # 设置端口号
s.bind((host, port))                   # 绑定 IP 地址和端口号
s.listen(1)                            # 开始监听客户端连接请求,最多允许一个连接
```

(2) 另一台机器作为客户端，创建一个 Socket 对象，指定服务器端的 IP 地址和端口号，然后调用 connect 方法尝试连接服务器。

示例代码如下：

```
s = socket.socket()                    # 创建 socket 对象
host = '192.168.1.100'                 # 设置服务器端的 IP 地址
port = 12345                           # 设置服务器端的端口号
s.connect((host, port))                # 尝试连接服务器
```

(3) 如果连接成功，服务器端的 Socket 对象会返回一个新的 Socket 对象，用于与客户端进行通信。客户端的 Socket 对象也可以与服务器端的新 Socket 对象进行通信。

(4) 通信过程中，可以使用 send 和 recv 方法发送和接收文本数据。也可以使用 sendall 和 makefile 方法发送和接收大量数据。

发送数据示例代码如下：

```
s.send('hello world')                  # 将内容编码后发送给客户端
```

接收数据示例代码如下：

```
while True:                            # 循环接收消息
    data = s.recv(1024)                # 接收客户端发送的数据,最多 1024 字节
    if data:                           # 如果接收到数据
        s = data.decode()              # 在文本框中显示客户端发送的消息
        print(s)
    else:                              # 如果没有接收到数据
        break                          # 跳出循环
```

(5) 通信结束后，可以使用 close 方法关闭 Socket 对象。

示例代码如下：

```
s.close()                          # 关闭 socket 对象
```

下面的例子会实现一个点对点的聊天程序，在窗口上方大部分区域是一个列表组件，用于显示聊天记录和状态信息。最下方从左到右分别是文本输入框、“发送”按钮和“连接”按钮。单击“连接”按钮，会弹出 IP 输入框，输入 IP 后，如果连接成功，会在另一台机器上同样程序的列表组件一开始显示连接成功信息。如果两台机器的点对点聊天程序连接成功，可以在文本输入框中要发送的内容，然后单击“发送”按钮，将输入的内容发送到另一台机器的聊天程序中，并在列表中显示，效果如图 3-18 所示。

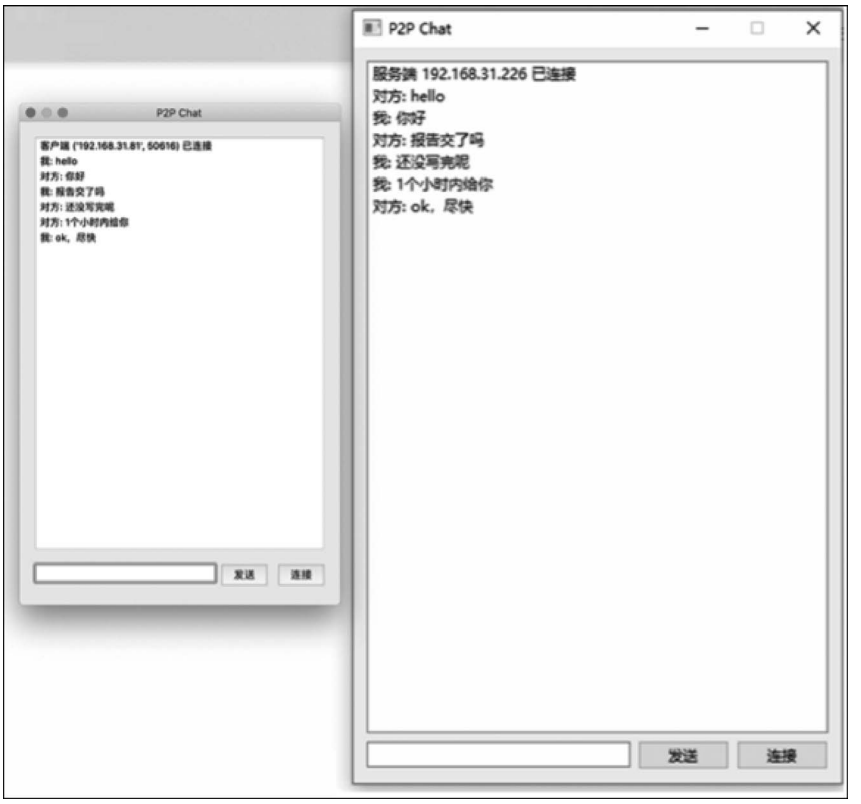

图 3-18　点对点聊天

代码位置：src/gui/pyqt6/p2pchat.py

```
import sys
import socket
from PyQt6.QtWidgets import *
from PyQt6.QtNetwork import QTcpServer, QTcpSocket
import threading
class ChatWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
        # 创建服务端 socket
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # 获取本机 ip 地址
        self.host_ip = socket.gethostbyname(socket.gethostname())
        # 设置通信端口号
        self.port = 8888
        # 绑定 ip 地址和端口号
        self.server_socket.bind((self.host_ip, self.port))
        # 开始监听客户端连接请求
        self.server_socket.listen()
```

```
        # 创建客户端 socket
        self.client_socket = None

        # 启动服务端线程
        threading.Thread(target=self.start_server).start()

    def initUI(self):
        self.setGeometry(300, 300, 400, 700)
        self.setFixedSize(400, 600)
        self.setWindowTitle('P2P Chat')

        vbox = QVBoxLayout()
        self.setLayout(vbox)

        self.listWidget = QListWidget()
        vbox.addWidget(self.listWidget)

        hbox = QHBoxLayout()
        vbox.addLayout(hbox)

        self.lineEdit = QLineEdit()
        hbox.addWidget(self.lineEdit)

        connectButton = QPushButton('发送')
        connectButton.clicked.connect(self.send_message)
        hbox.addWidget(connectButton)

        sendButton = QPushButton('连接')
        sendButton.clicked.connect(self.connect_to_server)
        hbox.addWidget(sendButton)
    def start_server(self):
        # 等待客户端连接
        self.client_socket, address = self.server_socket.accept()
        # 显示客户端地址
        self.listWidget.addItem( f"客户端 {address} 已连接")
        # 接收客户端消息
        self.receive_message()

    def connect_to_server(self):
        ip, ok = QInputDialog.getText(self, '连接', '请输入服务端的 ip 地址')
        if ok:
            try:
                # 创建客户端 socket
                self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                # 连接到服务端 socket
                self.client_socket.connect((ip, self.port))
                # 显示服务端地址
                self.listWidget.addItem(f"服务端 {ip} 已连接")
                # 接收服务端消息
                threading.Thread(target=self.receive_message).start()
```

```
                except:
                    # 显示连接失败的信息
                    self.listWidget.addItem(f"无法连接到 {ip}")
        def send_message(self):
            # 获取文本输入框中的文本
            message = self.lineEdit.text()
            # 如果文本不为空
            if message:
                try:
                    # 通过 socket 发送文本给对方
                    self.client_socket.send(message.encode('utf-8'))
                    # 在聊天记录中显示自己发送的消息
                    self.listWidget.addItem( f"我: {message}")
                    # 清空文本输入框中的文本
                    self.lineEdit.clear()
                except:
                    # 显示发送失败的信息
                    self.listWidget.addItem(f"无法发送消息")
        def receive_message(self):
            while True:
                try:
                    # 从 socket 中接收对方发送的消息
                    message = self.client_socket.recv(1024).decode()
                    # 如果消息不为空
                    if message:
                        # 在聊天记录中显示对方发送的消息
                        self.listWidget.addItem(f"对方: {message}")
                except:
                    # 显示接收失败的信息
                    self.listWidget.addItem(f"无法接收消息")
                    break
    if __name__ == '__main__':
        app = QApplication(sys.argv)
        chatWindow = ChatWindow()
        chatWindow.show()
        sys.exit(app.exec())
```

本例需要在两台机器上运行，在其中一台机器上单击“连接”按钮，会弹出输入 IP 对话框；输入 IP 后，如果 IP 是正确的，会成功连接到另一台机器运行的点对点聊天程序，然后就可以进行聊天了。

3.14　小结

PyQt6 是目前最流行的 Python GUI 工具包，而且功能相当强大。PyQt6 中不仅提供了与 GUI 相关的 API，还提供了很多非 GUI 的 API。例如，QtMultimedia 提供了与多媒体相关的 API；QtSql 提供了与数据库相关的 API；QtWebSockets 提供了与 WebSocket 相关的 API。在本书后面的部分，还会有多处使用 PyQt6 以及其他第三方库创建各种有趣的应用。因此，学好 PyQt6 是能否顺利理解这些案例的基础。