Create GUI Applications *with* Python & Qt6

5th Edition | PyQt6

The **hands-on** guide to making apps with Python

Martin Fitzpatrick

引言

如果您想使用 Python 创建图形用户界面（GUI）应用程序，可能会感到无从下手。因为需要理解许多新概念才能让程序正常运行。但就像解决任何编程问题一样，第一步是学会以正确的方式面对问题。本书将从 GUI 开发的基楚原理出发，逐步引导您使用 PyQt6 创建自己的功能齐全的桌面应用程序。

本书的第一版于2016年发布。此后，它已被更新14次，根据读者反馈添加和扩展了章节。现在可用的PyQt资源比我一开始的时候更加丰富，但仍然缺乏深入、实用的指南来构建完整的应用程序。这本书填补了这一空白！

本书以一系列章节的形式呈现，依次探讨PyQt6的不同方面。我将较简单的章节放在前面，但如果您有特定的项目需求，无需拘泥于顺序。每个章节都会先引导您掌握基础概念，随后通过一系列代码示例逐步探索并学习如何自行应用这些理念。

您可以从http://www.pythonguis.com/d/pyqt6-source.zip下载本书的源代码和有关资源

由于本书篇幅有限，无法对整个Qt生态系统进行全面概述，因此书中提供了指向外部资源的链接——既包括网站pythonguis.com上的内容，也包括其他的来源。如果您在阅读时想到"我不知道我能否做到这一点"，最好的办法就是放下这本书，然后去亲自尝试！在整个过程中，请务必定期备份您的代码，这样如果您不小心搞砸了，至少还有东西可以恢复。

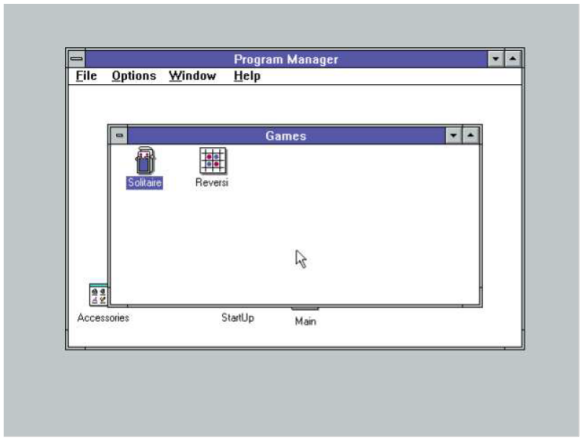
整本书都会有这样的方框，这个标识意味着提供信息、提示和警告。如果您赶时间，可以放心地跳过所有这些方框。但是，阅读它们会让您对Qt 框架知识有更深入、更全面的了解。

1. 图形用户界面（GUI）简史

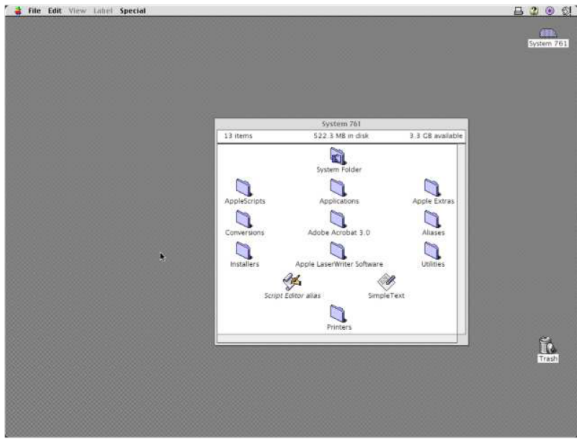
图形用户界面（GUI，Graphical User Interface）有着悠久而古老的历史，可追溯到20 世纪 60 年代。 斯坦福大学的 NLS（ON-Line 系统）引入了鼠标和窗口概念。并在 1968 年首次公开展示。 随后，1973 年施乐 PARC Smalltalk 系统也采用图形用户界面，它是大多数现代图形用户界面的基础。

这些早期系统已经具备了我们如今在现代桌面图形用户界面中习以为常的许多功能，包括窗口、菜单、单选按钮、复选框和图标。这些功能的组合使得我们为这类界面创造了早期缩写词：WIMP（窗口、图标、菜单、指点设备——即鼠标）。

1979年，首款搭载图形用户界面的商用系统——PERQ工作站正式发布。这促使了其他多项GUI开发项目，其中，最引人注目的是1983年发布的*Apple Lisa*，该系统引入了菜单栏和窗口控制概念，以及来自雅达利（GEM）和阿米加的其他系统。在UNIX系统中，*X Window*于1984年问世，而面向个人电脑的Windows首个版本则于1985年发布。



Microsoft Windows 3.1



Apple System 7 (Emulated)

> 图一：微软的Windows3.1系统（1992）的桌面和苹果系统Apple System 7（1991）的桌面

早期图形用户界面并未像人们想象的那样一炮而红，这主要是因为其发布时缺乏兼容的软件，且硬件要求较高——尤其是对家庭用户而言。然而，随着时间的推移，图形用户界面范式逐渐成为与计算机交互的首选方式，而WIMP（窗口、指针、菜单、按钮） 范式也牢固确立为行业标准。这并不意味着桌面环境没有尝试过取代WIMP范式。例如，微软的*Microsoft Bob*（1995）就是微软备受批评的尝试，试图用一个房子来取代桌面。

> 图二：微软试图抛弃桌面范式转而使用一种卡通的房子

自从*Windows 95*（1995）发布以来，一直到*Mac OS X*（2001）、*GNOME Shell*（2011）和*Windows 10*（2015），被誉为革命性的用户界面层出不穷。这些系统均对各自的用户界面进行了全面改造，它们往往伴随着大量宣传，但本质上并没有发生根本性变化。这些用户界面仍然是典型的WIMP系统，它们的运作方式与自20世纪80年代以来图形用户界面的运作方式基本相同。

当革命来临时，它是移动的——鼠标已被触控取代，窗口已被全屏应用取代。但即使在我们所有人都随身携带智能手机的今天，每天仍然有大量的工作是在台式电脑上完成的。WIMP在40多年中的创新中幸存下来，并有希望继续生存下去。

2. 关于Qt的一点小知识

Qt 是一个免费且开源的控件工具包，用于创建跨平台图形用户界面应用程序，可以允许应用程序使用单一代码库针对 Windows、macOS、Linux 和 Android 等多个平台进行开发。但 Qt 远远不止是一个控件工具包，它还内置了对多媒体、数据库、矢量图形和 MVC 接口的支持，因此更准确地说，它是一个应用程序开发框架。

Qt 是由 Eirik Chambe-Eng 和 Haavard Nord 在 1991 年创立的，他们还在 1994 年成立了首家 Qt 公司 *Trolltech*。目前，Qt 由 *The Qt Company* 负责开发，并持续进行定期更新，不断添加新功能并扩展移动设备及跨平台支持。

Qt 和 PyQt6

PyQt6 是由 *Riverbank Computing* 开发的 Qt 工具包的 Python 绑定库。当你使用 PyQt6 编写应用程序时，实际上是在使用 Qt 编写应用程序。PyQt6 库实际上是围绕 C++ Qt 库的封装，这使得它可以在 Python 中使用。

由于这是一个用于访问 C++ 库的 Python 接口，因此 PyQt6 中使用的命名约定并不遵循 PEP8 标准。例如，函数和变量使用混合大小写 `mixedCase` 而不是蛇形大小写 `snake_case` 进行命名。您是否在自己的应用程序中遵循这个标准完全取决于您自己，然而我发现继续遵循 Python 标准来编写自己的代码有助于明确PyQt6 的代码与您自己的代码之间的区别。

最后，虽然有专门针对 PyQt6 的文档，但您经常会发现自己需要阅读 Qt 的官方文档，因为它的内容更加全面。如果您需要将 Qt C++ 代码转换为 Python 代码的建议，可以参考附录B:将c++代码实例转化为 Python

更新和其他资源

本书定期更新。 如果您直接从我这里购买本书，您将会收到自动更新的电子版。 如果您在其他地方购买了本书，请将您的收据发送至 register@pythonguis.com，以获得最新的数字版本，并且注册以获得未来的更新。

您可能也有兴趣加入我的 Python GUI 学院，我在那里提供了以下视频视频教程，内容涵盖本书及其他内容！

> 加入academy.pythonguis.com！

PyQt6 基本功能

现在是时候迈出使用PyQt6创建图形用户界面的第一步了！

在本章中，您将学习PyQt6的基础知识，这些知识是您创建任何应用程序的基础。我们将开发一个简单的桌面窗口应用程序。我们将添加控件，使用布局进行排列并将这些控件与函数连接，使您能够通过图形用户界面触发应用程序的行为。

请以提供的代码为指导，但还请随时自己动手尝试。 这就是学习这些代码如何工作的最好方法。

> 在我们开始之前，您需要安装一个可以正常运行的 PyQt6。如果您还没有安装，请查看 附录A：安装 PyQt6。

> 不要忘记下载本书的源代码http://www.pythonguis.com/d/pyqt6-source.zip!

3. 我的第一个应用程序

让我们来创建第一个应用程序！首先，创建一个新的 Python 文件——您可以随意将其命名为您喜欢的任何名字（例如：`myapp.py`），然后将它保存在一个可以访问的地方。 我们将在这个文件中编写我们的简单应用程序。

> 我们将在此文件中进行编辑，您可能想要回到代码的早期版本，所以记得定期备份。

创建您的应用程序

我们的第一个应用程序的源代码如下所示。 请您逐词键入，注意不要出错。 如果您弄错了，Python 会告诉您哪里出错了。 如果您不想全部手动键入，这个文件包含在这本书的源代码中.

Listing 1. basic/creating_a_window_1.py

```
from PyQt6.QtWidgets import QApplication, QWidget
# 仅用于访问命令行参数
import sys

# 每个应用程序需要一个（且只有一个）QApplication 实例
# 输入 sys.argv，这可以允许应用程序使用命令行参数
# 如果知道不会使用命令行参数,也可以使用 QApplication([])
app = QApplication(sys.argv)
```

```
# 创建一个 Qt widget作为我们的窗口。
window = QWidget()
window.show() # 这很重要!!!!! 默认情况下，窗口是隐藏的。

# 开始事件循环
app.exec()

# 您的应用程序不会到达这里，直到您退出并且事件循环终止
```

首先，启动您的应用程序。 您可以像任何其他 Python 脚本一样从命令行运行它，例如

```
python MyApp.py
```

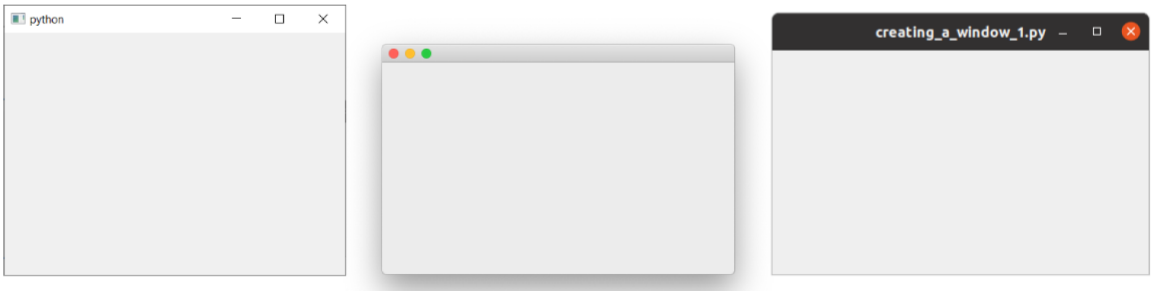
或者，对于 Python 3

```
python3 MyApp.py
```

从现在起，您将看到下面的提示框，提示您运行应用程序并提示您接下来将看到什么。

> 🚀 **运行它吧!** 现在您将看到您的窗口。 Qt 会自动创建一个窗口，您可以拖动它并调整其大小,就像您见过的其他窗口那样。

您看到的内容取决于运行这个示例的平台。下图展示了在 *Windows*、*macOS* 和 *Linux (Ubuntu)* 上显示的窗口。



> 图三: 我们分别在*Windows*、*macOS* 和 *Linux (Ubuntu)* 上显示的窗口

代码中的步骤

让我们逐行地查看代码，以便准确地理解到底发生了什么。

首先，我们导入应用程序所需的 PyQt6 类。 这里我们导入 `QApplication`（应用程序处理程序）和 `QWidget`（基本的空 GUI控件），这两个类都来自 `QtWidgets` 模块。

```
from PyQt6.QtWidgets import QApplication, QWidget
```

Qt 的主要模块包括 `QtWidgets`、 `QtGui` 和 `QtCore`。

> 您可以使用 `from <module> import *`，但这种全局导入在 Python 中通常是不受欢迎的，所以在这里我们将避免使用它

接下来，我们创建一个`QApplication`实例，传入`sys.arg`，即包含传递给应用程序的命令行参数的Python列表

```
app = QApplication(sys.argv)
```

如果您不会使用命令行参数来控制 Qt，您可以传递一个空列表，例如

```
app = QApplication([])
```

接下来，我们使用变量名`window`来创建一个`QWidget`实例。

```
window = QWidget()
window.show()
```

在 Qt 中，*所有*顶层部件都是窗口，也就是说，它们没有父控件，也不嵌套在另一个控件或布局中。这意味着您可以在技术上使用任何您喜欢的控件来创建一个窗口。

> 我看不到我的窗口!
> 没有父类的小工具默认是不可见的。 因此，在窗口对象后，我们必须始终调用`.show()`来使其可见。 您可以移除 .show() 并运行应用程序，但您会无法退出!

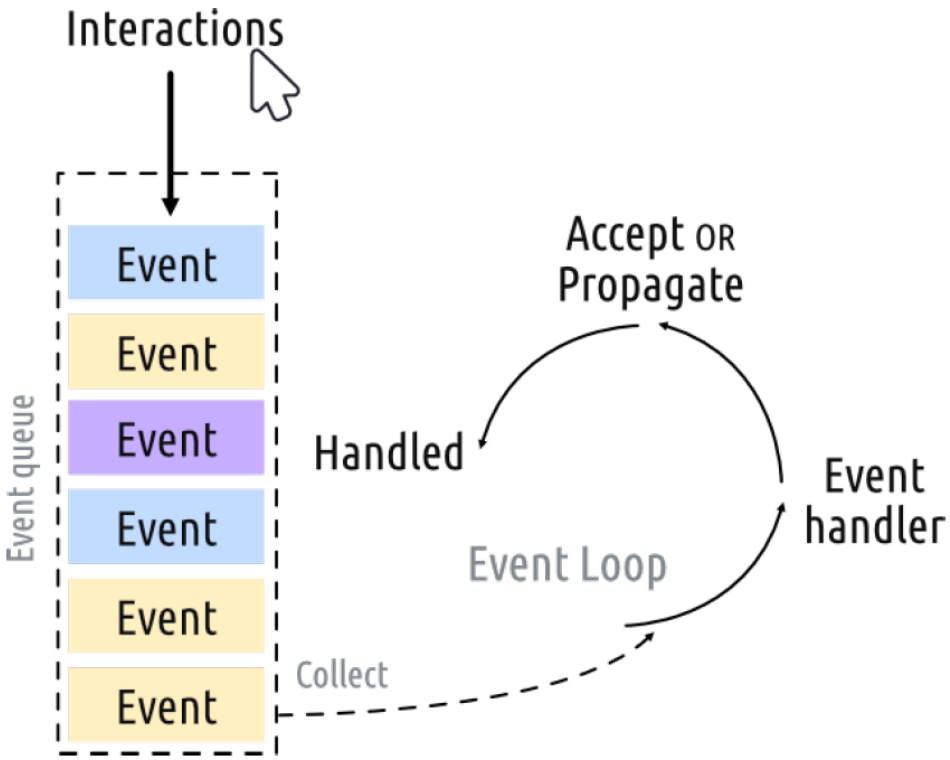
> 什么是窗口?
>
> - 保存应用程序的用户界面
> - 每个应用程序至少需要一个（.....但也可以有更多）
> - 默认情况下，当最后一个窗口关闭时，应用程序将退出

最后，我们调用`app.exec()`来开始事件循环。

什么是事件循环?

在屏幕上显示窗口之前，有几个有关 Qt 世界中应用程序的组织方式的关键概念需要介绍一下.如果您已经熟悉事件循环，就可以放心地跳到下一节。

每个 Qt 应用程序的核心都是 `QApplication` 类。 每个应用程序需要一个，且只需要一个 `QApplication` 对象才可以运行。 该对象包含应用程序的**事件循环**——管理所有图形用户界面交互的核心循环。



> 图4:Qt中的事件循环

与应用程序的每次交互——无论是按键、点击鼠标还是移动鼠标——都会产生一个事件，该事件被置于**事件队列**中。在事件循环中，每次迭代都会对队列进行检查，如果发现正在等候的事件，程序就会将事件和控制权传递给特定的事件处理程序。事件处理程序会处理事件，然后将控制权传递回事件循环，等待处理更多事件。 每个应用程序只能由**一个**事件循环

> 有关 `QApplication` 类.....
>
> - `QApplication` 包含 Qt 事件循环
> - 需要一个 `QApplication` 实例
> - 您的应用程序将在事件循环中等待，直到有操作执行
> - 任何时候都只有**一个**事件循环

`QMainWindow`

正如我们在上一部分中所发现的，在 Qt 中任何控件都可以是窗口。 例如，如果您使用 `QPushButton` 代替 `QtWidget`。 在下面的示例中，您将得到一个有一个可按下的按钮的窗口。

Listing 2. basic/creating_a_window_2.py

```
import sys
from PyQt6.QtWidgets import QApplication, QPushButton

app = QApplication(sys.argv)

window = QPushButton("Push Me")
window.show()

app.exec()
```

这太棒啦，但其实用处不大——您很少需要一个只有一个控件的用户界面！ 但是，正如我们接下来会发现的，使用布局将部件嵌套到其他部件中的功能着您可以在一个空的 `QWidget` 中构建复杂的用户界面。

不过，Qt 已经为您提供了解决方案——`QMainWindow`。 这是一个预制的窗口部件，它提供了大量您可能会使用的标准窗口功能，包括工具栏、菜单、状态栏、可停靠控件等。

我们稍后会了解这些高级功能，但现在，我们将为我们的应用程序添加一个简单的空白 `QMainWindow` 窗口。

Listing 3. basic/creating_a_window_3.py

```
from PyQt6.QtWidgets import QApplication, QMainWindow
import sys

app = QApplication(sys.argv)

window = QMainWindow()
window.show() # 这很重要!!!!! 默认情况下，窗口是隐藏的

# 开始事件循环
app.exec()
```

> 🚀 **运行它吧！** 您会看到您的主窗口。它看上去和之前的那个完全一致！

目前我们的 `QMainWindow` 并不太有趣。我们可以添加一些内容来改善它。如果您想创建一个自定义窗口，最好的方法是继承 `QMainWindow`，然后在 `__init__` 块中包含窗口的设置。这使得窗口的行为可以自包含。我们可以添加我们自己的 `QMainWindow` 子类——为了简单起见，我们称它为 `MainWindow`。

Listing 4. basic/creating_a_window_4.py

```
import sys
from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QPushButton,
) #1

# 创建子类 QMainWindow 来自定义您的应用程序的主窗口
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__() #2

        self.setWindowTitle("My App")
```

```
        button = QPushButton("Press Me!")

        # 设置窗口的中心控件
        self.setCentralWidget(button) #3


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

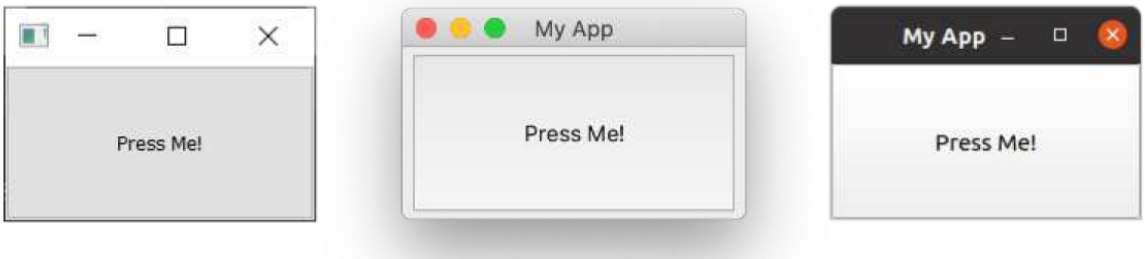
> 1. 常用的 Qt 控件总是从 `QtWidgets` 命名空间导入。
> 2. 我们必须始终调用 `super()` 类的 `__init__`方法。
> 3. 使用 `.setCentralWidget` 在 `QMainWindow` 中放置一个控件。

> 当您子类化一个 Qt 类时，您必须始终调用`super`函数`__init__`以便 Qt 设置对象。

在我们的 `__init__` 块中，我们首先使用 `.setWindowTitle()` 来更改我们主窗口的标题。 然后，我们将第一个窗口控件——一个 `QPushButton` 添加到窗口中间。这是 Qt 中可用的基本部件之一。 在创建按钮时，您可以输入希望按钮显示的文本。

最后，我们在窗口上调用 `.setCentralWidget()`。 这是 `QMainWindow` 特有的函数，用于设置窗口中间的控件。

> 🚀 **运行它吧！** 您会再一次看到您的主窗口，但是这次`QPushButton`控件会显示在中央。按下按钮，但是什么也不会发生，我们将会稍后来调整



> 图五：显示在*Windows*，*macOS*和*Linux*上面的含有一个`QPushButton`的`QMainWindow`

> 渴望使用控件？
> 我们稍后会详细介绍更多控件，但如果您没有耐心，想先睹为快的话，可以看看QWidget 文档。
> 尝试将不同的控件添加到您的窗口！

调整窗口和控件的大小

当前窗口可自由调整大小——您只需用鼠标抓住窗口任何一个角，即可拖动并调整窗口大小至任意尺寸。虽然允许用户调整应用程序大小是件好事，但有时您可能需要对最小或最大尺寸设置限制，或将窗口锁定为固定大小。

在 Qt 中，尺寸通过 `QSize` 对象进行定义。该对象依次接受宽度和高度参数。例如，以下代码将创建一个固定尺寸的 400x300 像素窗口。

Listing 5. basic/creating_a_window_end.py

```
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

# 创建子类 QMainWindow 来自定义您的应用程序的主窗口
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")

        self.setFixedSize(QSize(400, 300)) #1

        # 设置窗口的中心控件
        self.setCentralWidget(button)

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 1. 设置窗口大小

> 🚀 **运行它吧！** 您会看到一个固定大小的窗口——试着去调整大小吧，这不会成功的

> 图六：我们的固定大小的窗口，注意最大化控件在*Windows* 和 *Linux* 上被禁用。 在 *macOS* 上，您可以将应用程序最大化以填满屏幕，但是
> 中央控件不会被调整大小。

除了可以调用 `.setFixedSize()` 方法外，您还可以调用 `.setMinimumSize()` 和 `.setMaximumSize()` 方法分别设置窗口的最小和最大尺寸。您不妨亲自尝试一下！

> 您可以用在任意控件中使用这种方法

在本节中，我们介绍了 `QApplication` 类、`QMainWindow` 类、事件循环，并尝试将一个简单的控件添加到窗口中。在下一节中，我们将了解 Qt 为控件和窗口之间以及控件和窗口与您自己的代码之间的通信提供的机制。

> 请将文件的副本保存为 `myapp.py`，我们稍后还会用到它。

4. 信号与槽

到目前为止，我们已经创建了一个窗口，并添加了一个简单的按钮控件，但该按钮没有任何功能。这完全没有用啊——当您创建图形用户界面应用程序时，通常一定希望它们能够执行某些操作！我们需要一种方法，将按下按钮的操作与执行某些操作联系起来。在Qt 中，信号和槽提供了这种功能。

信号是控件在发生某些事件时发出的通知。这些事件可以是按下按钮、输入框中文本的变化、窗口文本的变化等等任何事情。许多信号是由用户操作触发的，但这并不是一条死板的规则。

除了通知发生的事件外，信号还可以发送数据，并提供有关发生的事件的更多背景信息。

> 您还可以创建自己的自定义信号，我们将在之后的“32. 扩展信号”中进行探讨。

槽是 Qt 用于接收信号的名称。在 Python 中，应用程序中的任何函数（或方法）都可以用作槽——只需将信号连接到它即可。如果信号发送数据，则接收函数也会接收到该数据。许多 Qt 控件也有自己的内置槽，这意味着您可以直接将 Qt 控件连接在一起。

让我们来看看 Qt 信号的基本知识以及如何使用它们将控件连接起来以便在应用程序中实现各种功能。

> 请加载一份新的 `myapp.py` 文件用于本节内容并以新名称保存

`QPushButton`的信号

我们简单的应用程序目前有一个 `QMainWindow`，其中 `QPushButton` 被设置为中央控件。首先，我们将这个按钮与一个自定义的 Python 方法连接起来。在这里，我们创建了一个名为 `the_button_was_clicked` 的简单自定义槽，它接受来自 `QPushButton` 的点击信号。

Listing 6. basic/signals_and_slots_1.py

```
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QPushButton,
) #1

import sys

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__() #2

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_clicked)

        # 设置窗口的中心控件
        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 如果您点击这个按钮您将会在控制台中看到文本”Clicked!“

控制台输出

```
Clicked!
Clicked!
Clicked!
Clicked!
```

接收数据

这真是一个很好的开始！我们已经知道信号还可以发送**数据**，以提供更多关于刚刚发生的事件的信息。`.clicked` 信号也不例外，它还提供了按钮的选中（或切换）状态。对于普通按钮，该状态始终为 `False`，因此我们的第一个槽忽略了这些数据。但是，我们可以让按钮**可选中**，然后看看效果。

在下面的示例中，我们将添加第二个槽用于输出**检查状态**。

Listing 7. basic/signals_and_slots_1b.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QPushButton,
) #1

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__() #2

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_clicked)
        button.clicked.connect(self.the_button_was_toggled)

        # 设置窗口的中心控件
        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")

    def the_button_was_toggled(self, checked):
        print("Checked?", checked)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 如果您点击了这个按钮，您将会看到它在被点击之后高亮了。请再次点击它然后松开，并在控制台中检查状态

控制台输出

```
Clicked!
Checked? True
Clicked!
Checked? False
Clicked!
Checked? True
Clicked!
Checked? False
Clicked!
Checked? True
```

您可以将任意数量的槽连接到一个信号，并可以在槽上同时响应不同版本的信号。

存储数据

通常，将控件的当前状态存储在 Python 变量中非常有用。这样就可以像处理其他 Python 变量一样处理这些值而无需访问原始控件。您可以将这些值存储为单独的变量，或者根据需要使用字典。在下一个示例中，我们将按钮的选中值存储在名为 `button_is_checked` 的变量中。

Listing 8. basic/signals_and_slots_1c.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.button_is_checked = True #1

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_toggled)
        button.setChecked(self.button_is_checked) #2

        # 设置窗口的中心控件
        self.setCentralWidget(button)

    def the_button_was_toggled(self, checked):
        self.button_is_checked = checked #3

        print(self.button_is_checked)
```

> 1. 为变量设置默认值。
> 2. 使用默认值设置控件的初始状态。
> 3. 当控件状态发生变化时，更新变量以匹配。

您可以对任何 PyQt6 控件使用相同的模式。如果控件未提供发送当前状态的信号，则您需要在处理程序中直接从控件检索该值。例如，这里我们正在检查按下处理程序中的**选中**状态。

Listing 9. basic/signals_and_slots_1d.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.button_is_checked = True

        self.setWindowTitle("My App")

        self.button = QPushButton("Press Me!") #1
        self.button.setCheckable(True)
        self.button.released.connect(
            self.the_button_was_released
        ) #2
        self.button.setChecked(self.button_is_checked)

        # 设置窗口的中心控件
        self.setCentralWidget(self.button)

    def the_button_was_released(self):
        self.button_is_checked = self.button.isChecked() #3

        print(self.button_is_checked)
```

1. 我们需要保留对按钮的引用，以便在我们的槽中访问它。
2. 释放信号在按钮释放时触发，但不会发送检查状态。
3. `.isChecked()` 返回按钮的检查状态。

更改界面

到目前为止，我们已经了解了如何接受信号并将输出打印到控制台。但是，当我们点击按钮时，如何在界面中触发某些操作呢？让我们更新槽方法来修改按钮，更改文本并禁用按钮，使其不再可点击。我们还将暂时删除可选状态。

Listing 10. basic/signals_and_slots_2.py

```
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

import sys

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        self.button = QPushButton("Press Me!") #1
        self.button.clicked.connect(self.the_button_was_clicked)

        # 设置窗口的中心控件
        self.setCentralWidget(self.button)

    def the_button_was_clicked(self):
        self.button.setText("You already clicked me.") #2
        self.button.setEnabled(False) #3

        # 我们也来更改窗口标题
        self.setWindowTitle("My Oneshot App")
```

```
app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

1. 我们需要在 `the_button_was_clicked` 方法中访问该按钮，因此我们将其引用保存在 `self` 中。
2. 您可以通过向 `.setText()` 方法传递一个字符串来更改按钮的文本。
3. 要禁用按钮，请调用 `.setEnabled()` 方法并传入 False。

> 🚀 **运行它吧！** 如果您单击按钮，文本将发生变化并且按钮将将变得不可点击。

您并不局限于更改触发信号的按钮，您可以在槽方法中做任何您想做的事情。例如，尝试将以下行添加到 `_button_was_clicked` 方法中，以同时更改窗口标题。

```
self.setWindowTitle("A new window title")
```

大多数控件都有自己的信号，我们用于窗口的 `QMainWindow` 也不例外。

在下面的更复杂的示例中，我们将 `QMainWindow` 上的 `.windowTitleChanged` 信号连接到自定义槽方法 `the_window_title_changed`。该槽还会接收新窗口标题。

Listing 11. basic/signals_and_slots_3.py

```
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

import sys
from random import choice

window_titles = [
    "My App",
    "My App",
    "Still My App",
    "Still My App",
    "What on earth",
    "What on earth",
    "This is surprising",
    "This is surprising",
    "Something went wrong",
] #1


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.n_times_clicked = 0

        self.button = QPushButton("Press Me!")
        self.button.clicked.connect(self.the_button_was_clicked)
```

```
        self.windowTitleChanged.connect(
            self.the_window_title_changed
        ) #2

        # 设置窗口的中心控件
        self.setCentralWidget(self.button)

    def the_button_was_clicked(self):
        print("Clicked.")
        new_window_title = choice(window_titles)
        print("Setting title: %s" % new_window_title)
        self.setWindowTitle(new_window_title) #3

    def the_window_title_changed(self, window_title):
        print("Window title changed: %s" % window_title) #4

        if window_title == "Something went wrong":
            self.button.setDisabled(True)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

1. 使用 random.choice() 从窗口标题列表中进行选择。
2. 将我们的自定义槽方法 `the_window_title_changed` 连接到 windows 的 `.windowTitleChanged` 信号。
3. 将窗口标题设置为新标题。
4. 如果新窗口标题为“Something went wrong”（出现错误），则禁用该按钮。

> 🚀 **运行它吧！** 反复点击按钮，直到标题变为“Something went wrong”（出现错误）且按钮失效。

在这个例子中有几点值得注意。

首先，在设置窗口标题时，`windowTitleChanged` 信号**并不总是**被发出。只有当新标题与之前的标题**不同**时，该信号才会被触发。如果您多次设置相同的标题，该信号只会第一次被触发。

> 请务必仔细检查信号触发的条件，以免在应用程序中使用时出现意外。

其次，请注意我们如何使用信号将事物链接在一起。一个事件的发生——按下按钮——可以触发其他多个事件的发生。这些后续效应无需知道是什么原因导致它们发生，而是简单地遵循一些简单的规则。将效应与触发它们的原因分离是构建图形用户界面应用程序时需要考虑的关键因素之一。我们将在本书中多次提及这一点！

在本节中，我们介绍了信号和槽。我们演示了一些简单的信号，以及如何使用它们在应用程序中传递数据和状态。接下来，我们将介绍 Qt 为您的应用程序提供的控件以及它们提供的信号。

将控件直接连接在一起

到目前为止，我们已经看到了将控件信号连接到 Python 方法的示例。当控件触发信号时，我们的 Python 方法会被调用，并接收来自信号的数据。但您并不总是需要使用 Python 函数来处理信号——您也可以将 Qt 控件直接相互连接。

在下面的示例中，我们将一个 `QLineEdit` 控件和一个 `QLabel` 添加到窗口中。在窗口的 `__init__` 中，我们将我们的行编辑 `.textChanged` 信号连接到 `QLabel` 上的 `.setText` 方法。现在，每当 `QLineEdit` 中的文本发生更改时，`QLabel` 都会将该文本发送到其 `.setText` 方法。

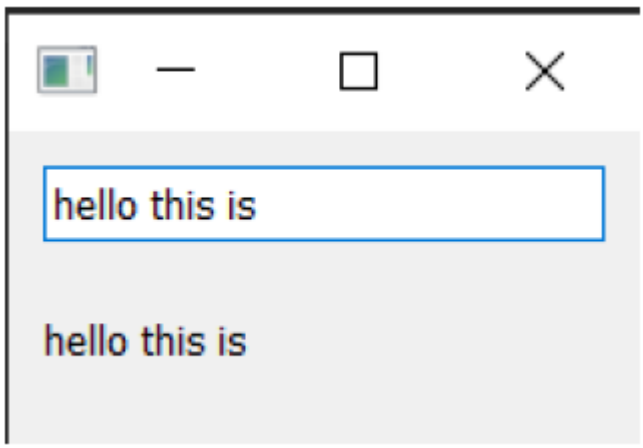
Listing 12. basic/signals_and_slots_4.py

```
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QLabel,
    QLineEdit,
    QVBoxLayout,
    QWidget,
)

import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        self.label = QLabel()
        self.input = QLineEdit()
        self.input.textChanged.connect(self.label.setText) #1

        layout = QVBoxLayout() #2
        layout.addWidget(self.input)
        layout.addWidget(self.label)

        # 设置窗口的中心控件
        self.setCentralWidget(container)


app = QApplication(sys.argv)

window = MainWindow()
window.show()
```

```
app.exec()
```

1. 请注意，要将输入与标签连接起来，输入和标签都必须被定义。
2. 此代码将两个控件添加到布局中，并将其设置在窗口上。我们将在后续章节中详细介绍这一点，现在可以先忽略它。

> 🚀 **运行它吧！** 请在上方的方框中键入一些文字，您就会看到它立即出现在标签上。



> 图七：任何输入进来的文本都会在标签上立即显示

大多数 Qt 控件都有可用的槽，您可以将任何发出与它接受的**类型相同**的信号连接到该槽。控件文档在“公共槽”下列出了每个控件的槽。例如，请参阅 QLabel。

5. 控件

在 Qt 中，**控件**是指用户可以与之交互的用户界面（UI）组件。用户界面由多个控件组成，这些控件被排列在窗口内。Qt 提供了大量可用的控件，甚至允许您创建自己的自定义控件。

在本书的代码示例中，有一个名为 `basic/widgets_list.py` 的文件，您可以运行它来在窗口中显示一组控件。它使用了一些我们稍后会介绍的复杂技巧，所以，现在先不要担心代码的问题。

> 🚀 **运行它吧！** 您将会看到一个包含多个交互式控件的窗口。

> 图八：在*Windows*，*macOS* 和 *Linux(Ubuntu)* 上面展示的控件应用程序的例子

示例中显示的控件如下所示，从上到下依次为：

控件	作用
`QCheckbox`	复选框
`QComboBox`	下拉列表框
`QDateEdit`	编辑日期
`QDateTimeEdit`	编辑日期和时间
`QDial`	可旋转表盘
`QDoubleSpinbox`	浮点数微调框
`QFontComboBox`	字体列表
`QLCDNumber`	相当难看的 LCD 显示屏
`QLabel`	不能互动的标签
`QLineEdit`	输入一行文本
`QProgressBar`	进度条
`QPushButton`	按钮
`QRadioButton`	仅有一个有效选项的选项组
`QSlider`	滑块
`QSpinBox`	整数微调框
`QTimeEdit`	编辑时间

还有更多控件，但它们并不太适合在这里全部展示！完整的列表请参阅 Qt 文档。下面我们将仔细看看一些最有用的控件。

> 请打开一个新的 `myapp.py` 文件并以新名称保存以完成本节内容。

QLabel

我们将从 `QLabel` 开始介绍，它可以说是 Qt 工具箱中最简单的控件之一。这是一个简单的单行文本，您可以将其放置在应用程序中。您可以在创建时通过传递字符串来设置文本——

```
widget = QLabel("Hello")
```

或者，通过使用 `.setText()` 方法——

```
widget = QLabel("1") # 创建的标签文本为 1
widget.setText("2")  # 标签现在显示 2
```

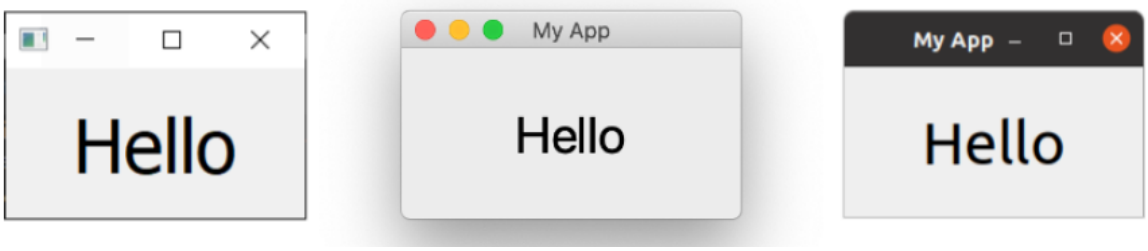
您还可以调整字体参数，例如控件中文本的大小或对齐方式。

Listing 13. basic/widgets_1.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QLabel("Hello")
        font = widget.font() #1
        font.setPointSize(30)
        widget.setFont(font)
        widget.setAlignment(
            Qt.AlignmentFlag.AlignHCenter
            | Qt.AlignmentFlag.AlignVCenter
        ) #2

        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

1. 我们使用 `<widget>.font()` 获取当前字体，对其进行修改，然后将其应用回去。这样可以确保字体与系统字体样式保持一致。
2. 对齐方式通过 `Qt.` 命名空间中的标志来指定。

> 🚀 **运行它吧！** 调整字体参数并查看效果。



图九：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的 `QLabel`

Qt 命名空间（`Qt.`）中包含各种属性，您可以使用这些属性来定制和控制 Qt 控件。我们将在后面的 “35. 枚举和 Qt 命名空间” 中详细介绍这一点。

用于水平对齐的标志包括——

标志	行为
`Qt.AlignmentFlag.AlignLeft`	与左边缘对齐
`Qt.AlignmentFlag.AlignRight`	与右边缘对齐
`Qt.AlignmentFlag.AlignHCenter`	在可用空间内水平居中
`Qt.AlignmentFlag.AlignJustify`	在可用空间内对文字进行调整

用于垂直对齐的标志包括——

标志	行为
`Qt.AlignmentFlag.AlignTop`	与顶部对齐
`Qt.AlignmentFlag.AlignBottom`	与底部对齐
`Qt.AlignmentFlag.AlignVCenter`	在可用空间中垂直居中

您可以使用管道符（`|`）将多个标志组合在一起，但请注意，每次只能使用一个垂直或水平对齐标志。

```
align_top_left = Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop
```

> 🚀 **运行它吧！** 尝试组合不同的对齐标志并观察其对文本位置的影响。

> **Qt 标志**
>
> 请注意，您使用了或运算符（`|`）按照惯例将两个标志组合在一起。这些标志是非重叠的位掩码。例如，`Qt.AlignmentFlag.AlignLeft` 的二进制值为 `0b0001`，而 `Qt.AlignmentFlag.AlignBottom` 的二进制值为 `0b0100`。通过按位或运算，我们得到值 `0b0101`，表示“底部左侧”。
>
> 我们将在后续的 “35. 枚举与 Qt 命名空间” 章节中对 Qt 命名空间和 Qt 标志进行更详细的探讨。

最后，还有一个简写标志，它同时在两个方向上居中——

标志	行为
`Qt.AlignmentFlag.AlignCenter`	水平**和**垂直居中

有趣的是，您也可以使用 `QLabel` 通过 `.setPixmap()` 方法显示一张图片。该方法接受一个像素图（像素数组），您可以通过将图片文件名传递给 `QPixmap` 来创建它。在随本书提供的示例文件中，您可以找到一个名为 `otje.jpg` 的文件，您可以按照以下方式在窗口中显示它：

Listing 14. basic/widgets_2a.py

```
import sys

from PyQt6.QtGui import QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QLabel("Hello")
        widget.setPixmap(QPixmap("otje.jpg"))

        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

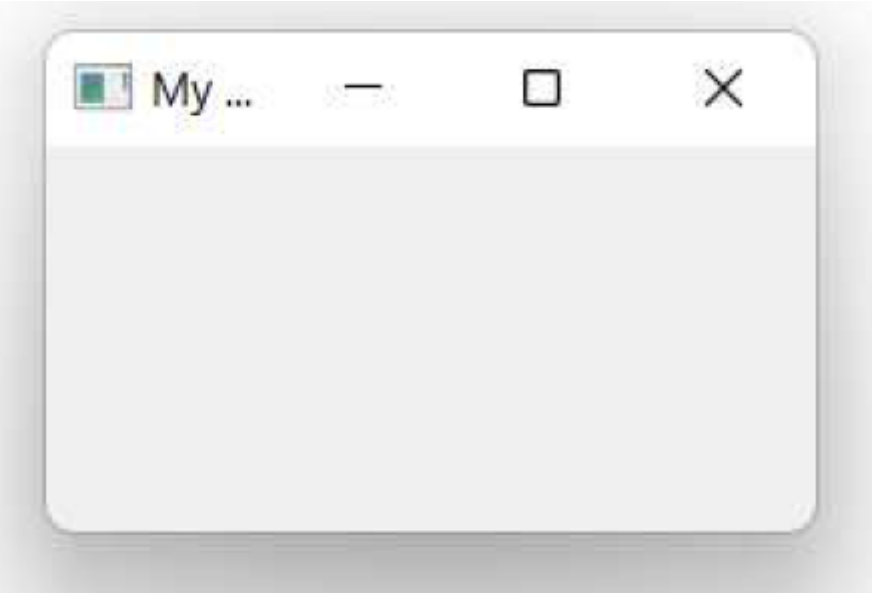
> 图十：叫做“Otje”的猫，太可爱啦

> 🚀 **运行它吧！** 调整窗口大小后，图像会被空白区域包围。

> 没看见图片？继续往下读读看！

在上面的示例中，我们仅使用文件名 `otje.jpg` 来指定要加载的文件。这意味着当应用程序运行时，文件将从当前文件夹中加载。然而，当前文件夹并不一定是脚本所在的文件夹——您可以从任何位置运行脚本。

如果你切换到上级目录（使用 `cd ..`）并再次运行脚本，文件将无法被找到，图像也无法加载。我的老天啊！



> 图十一：猫 Otje 不见了

> 这也是在从IDE运行脚本时常见的问题，因为IDE会根据当前激活的项目来设置路径。

要解决这个问题，我们可以获取当前脚本文件的路径，并利用该路径确定脚本所在的文件夹。我们的图像文件存储在同一文件夹中（或相对于此位置的某个文件夹中），这样也能确定该文件的位置。

文件内置变量 `file` 为我们提供了当前文件的路径。`os.dirname()` 函数从该路径中获取文件夹（或**目录**名称），然后我们使用 `os.path.join` 函数来构建文件的新路径。

Listing 15. basic/widgets_2b.py

```
import os
```

```
import sys

from PyQt6.QtGui import QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow

basedir = os.path.dirname(__file__)
print("Current working folder:", os.getcwd()) #1
print("Paths are relative to:", basedir) #2


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QLabel("Hello")
        widget.setPixmap(QPixmap(os.path.join(basedir, "otje.jpg")))

        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 1. 当前工作目录。
> 2. 我们的基础路径（相对于此文件）。

> 如果您现在还不完全理解，请不要担心，我们将在后面详细说明。

如果您现在运行这个脚本，图像将如预期显示——无论您从哪里运行脚本。脚本还会输出路径（以及当前工作目录），以帮助调试问题。在从应用程序加载任何外部文件时，请务必记住这一点。有关数据文件路径处理的更详细信息，请参阅 "33. 使用相对路径"。

默认情况下，图像在缩放时会保持其宽高比。如果您希望它拉伸并缩放以完全填充窗口，您可以在 `QLabel` 中设置 `.setScaledContents(True)` 方法。

请修改代码，在标签中添加 `.setScaledContents(True)` ——

Listing 16. basic/widgets_2b.py

```
widget.setPixmap(QPixmap(os.path.join(basedir, "otje.jpg")))
widget.setScaledContents(True)
```

> 🚀 **运行它吧！** 调整窗口大小，图片就会变形来自动适应大小。

图十二：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面使用 `Qlabel` 展示的像素图

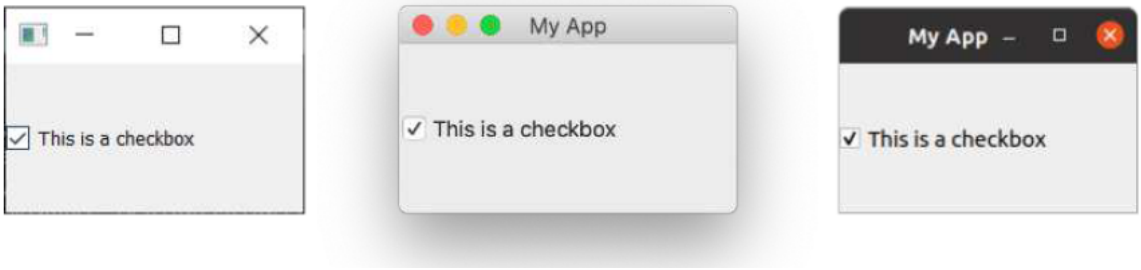
QCheckBox

下一个要介绍的控件是 `QCheckBox`，顾名思义，它为用户提供了一个可选框。然而，与所有 Qt 控件一样，它也有许多可配置的选项来更改控件的行为。

Listing 17. basic/widgets_3.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QCheckBox, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QCheckBox("This is a checkbox")
        widget.setCheckState(Qt.CheckState.Checked)

        # 对于三态：widget.setCheckState(Qt.PartiallyChecked)
        # 或：widget.setTristate(True)
        widget.stateChanged.connect(self.show_state)

        self.setCentralWidget(widget)

    def show_state(self, s):
        print(Qt.CheckState(s) == Qt.CheckState.Checked)
        print(s)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将会看到一个有标签文本的复选框



> 图十三：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QCheckBox`

您可以使用 `.setChecked` 或 `.setCheckState` 通过编程方式设置复选框状态。前者接受 `True` 或 `False`，分别代表已选中或未选中。但是，使用 `.setCheckState` 时，您还可以使用 `Qt.` 命名空间标志指定部分选中状态。

标志	行为
`Qt.CheckState.Checked`	该项已选中
`Qt.CheckState.Unchecked`	该项未选中
`Qt.CheckState.PartiallyChecked`	该项部分选中

支持部分选中状态（`Qt.CheckState.PartiallyChecked`）的复选框通常被称为“三态复选框”，即既非选中也非未选中。处于此状态的复选框通常显示为灰色复选框，并常用于分层复选框布局中，其中子项与父级复选框相关联。

如果您将值设置为 `Qt.CheckState.PartiallyChecked`，复选框将变为**三态**——即具有三种可能的状态。您还可以通过使用 `.setTristate(True)` 来达到相同的效果

> 您可能会注意到，当脚本运行时，当前状态的编号以整数形式显示，其中已选中 = 2，未选中 = 0，部分选中 = 1。您无需记住这些值——它们只是这些相应标志的内部值。您可以通过 `state == Qt.CheckState.Checked` 来测试状态。

QComboBox

`QComboBox` 是一个下拉列表，默认情况下处于关闭状态，需要点击箭头才能打开。您可以从列表中选择一个项目，当前选中的项目将作为标签显示在控件上。组合框适用于从长列表中选择一个选项。

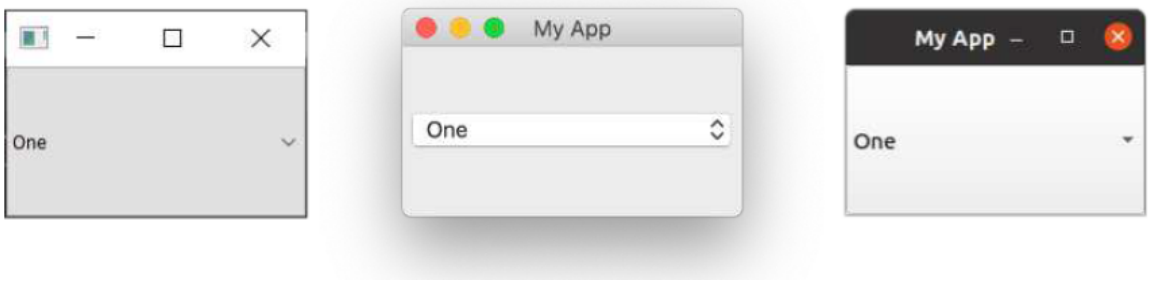
> 您可能在文字处理应用程序中见过用于选择字体样式或字号的组合框。尽管 Qt实际上提供了一个专门用于字体选择的组合框，即 `QFontComboBox`。

您可以通过向 `.addItems()` 方法传递一个字符串列表来向 QComboBox 添加项。项将按您提供的顺序依次添加。

Listing 18. basic/widgets_4.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QComboBox, QMainWindow

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QComboBox()
        widget.addItems(["One", "Two", "Three"])

        widget.currentIndexChanged.connect(self.index_changed)
        widget.currentTextChanged.connect(self.text_changed)

        self.setCentralWidget(widget)

    def index_changed(self, i): # i是一个int型整数
        print(i)

    def text_changed(self, s): # s是一个str型的字符串
        print(s)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到一个包含3个选项的下拉列表框。选择其中一项后，该选项将显示在输入框中。



> 图十四：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QComboBox`

当当前选中的项目被更新时，会触发 `.currentIndexChanged` 信号，默认情况下会传递列表中选中项目的索引。还有一个 `.currentTextChanged` 信号，它提供当前选中项目的标签，这个通常会更加实用。

`QComboBox` 也可以设置为可编辑模式，允许用户输入列表中不存在的值，并可选择将这些值插入列表或直接作为选中项使用。要启用可编辑模式，请加入这行代码：

```
widget.setEditable(True)
```

您还可以设置标志来确定插入操作的处理方式。这些标志存储在 `QComboBox` 类本身中，具体列表如下：

标志	行为
`QComboBox.InsertPolicy.NoInsert`	不允许插入
`QComboBox.InsertPolicy.InsertAtTop`	插入为第一个项
`QComboBox.InsertPolicy.InsertAtCurrent`	替换当前选中的项
`QComboBox.InsertPolicy.InsertAtBottom`	在最后一项之后插入
`QComboBox.InsertPolicy.InsertAfterCurrent`	在当前项之后插入
`QComboBox.InsertPolicy.InsertBeforeCurrent`	在当前项之前插入
`QComboBox.InsertPolicy.InsertAlphabetically`	按字母顺序插入

要使用这些选项，请按以下方式应用标志：

```
widget.setInsertPolicy(QComboBox.InsertPolicy.InsertAlphabetically)
```

您还可以通过调用 `.setMaxCount` 方法来限制盒子中允许的项目数量,例如：

```
widget.setMaxCount(10)
```

`QListWidget`

接下来是 `QListWidget`。该控件与 `QComboBox` 类似，只是选项以可滚动列表的形式呈现。它还支持同时选择多个项目。`QListWidget` 提供了一个 `currentItemChanged` 信号，该信号发送 `QListItem`（列表控件的元素），以及一个 `currentTextChanged` 信号，该信号发送当前项目的文本。

Listing 19. basic/widgets_5.py

```
import sys

from PyQt6.QtWidgets import QApplication, QListWidget, QMainWindow

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QListWidget()
        widget.addItems(["One", "Two", "Three"])
```

```
        widget.currentItemChanged.connect(self.index_changed)
        widget.currentTextChanged.connect(self.text_changed)

        self.setCentralWidget(widget)

    def index_changed(self, i): #不是索引，i 是 QListItem
        print(i.text())

    def text_changed(self, s): # s是一个str型的字符串
        print(s)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到相同的三个项，现在以列表形式显示。选中的项（如果有的话）将被高亮显示。

> 图十五：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QListWidget`

QLineEdit

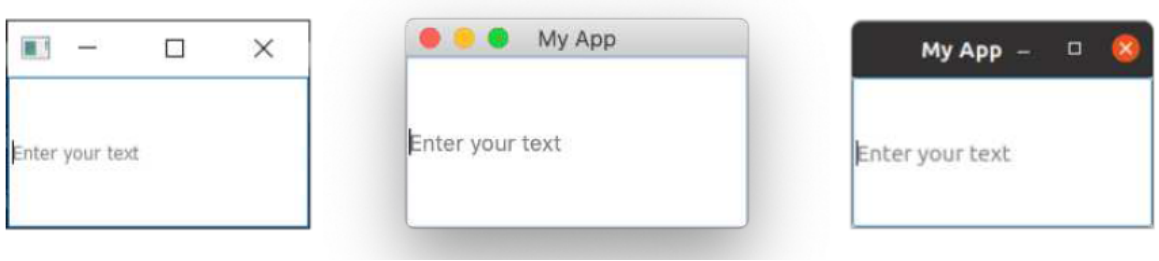
`QLineEdit` 控件是一个简单的单行文本编辑框，用户可以在其中输入内容。这些控件用于表单字段或没有限制有效输入列表的设置。例如，输入电子邮件地址或计算机名称时。

Listing 20. basic/widgets_6.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QLineEdit, QMainWindow

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QLineEdit()
        widget.setMaxLength(10)
```

```
        widget.setPlaceholderText("Enter your text")

        # widget.setReadOnly(True) # 取消注释该行以设置为只读模式

        widget.returnPressed.connect(self.return_pressed)
        widget.selectionChanged.connect(self.selection_changed)
        widget.textChanged.connect(self.text_changed)
        widget.textEdited.connect(self.text_edited)

        self.setCentralWidget(widget)

    def return_pressed(self):
        print("Return pressed!")
        self.centralWidget().setText("BOOM!")

    def selection_changed(self):
        print("Selection changed")
        print(self.centralWidget().selectedText())

    def text_changed(self, s):
        print("Text changed...")
        print(s)

    def text_edited(self, s):
        print("Text edited...")
        print(s)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到一个带有提示的简单文本输入框。



> 图十六：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QLineEdit`

如以上代码所示，您可以通过使用 `.setMaxLength` 方法为文本字段设置最大长度。占位符文本（即在用户输入内容前显示的文本）可通过 `.setPlaceholderText` 方法添加。

`QLineEdit` 为不同的编辑事件提供了一系列信号，包括（用户）按下回车键时、用户选择发生更改时。另外还有两个编辑信号，一个用于框中的文本被编辑的时候，另一个用于文本被更改的时候。这里的区别在于用户编辑和程序更改。只有当用户编辑文本时，才会发送 `textEdited` 信号。

此外，还可以使用输入掩码进行输入验证，以定义支持哪些字符以及在何处支持。这可以应用于字段如下：

```
widget.setInputMask('000.000.000.000;_')
```

上述规则允许使用以句点分隔的3位数字序列，因此可用于验证IPv4地址。

`QSpinBox` 和 `QDoubleSpinBox`

`QSpinBox` 提供了一个带箭头的小数字输入框，用于增加和减少值。 `QSpinBox` 支持整数，而相关的控件`QDoubleSpinBox` 支持浮点数。

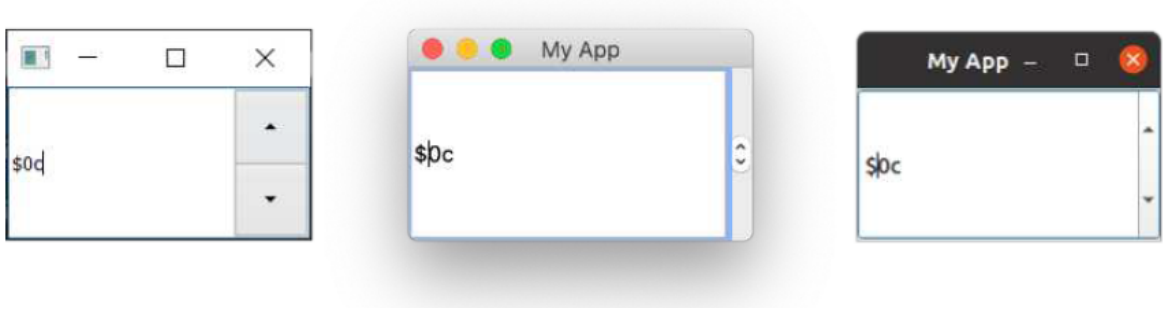
> 双精度(`double`)或双精度浮点数(double float)是 C++ 类型，相当于Python 自己的浮点数(`float`)类型，因此该控件以此命名。

Listing 21. basic/widgets_7.py

```
import sys

from PyQt6.QtWidgets import QApplication, QMainWindow, QSpinBox

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QSpinBox()
        # 或者: widget = QDoubleSpinBox()

        widget.setMinimum(-10)
        widget.setMaximum(3)
        # 或者: widget.setRange(-10,3)

        widget.setPrefix("$")
        widget.setSuffix("c")
        widget.setSingleStep(3) # 或者，对于QDoubleSpinBox，使用0.5
        widget.valueChanged.connect(self.value_changed)
        widget.textChanged.connect(self.value_changed_str)

        self.setCentralWidget(widget)

    def value_changed(self, i):
        print(i)

    def value_changed_str(self, s):
        print(s)


app = QApplication(sys.argv)
```

```
window = MainWindow()
window.show()

app.exec()
```

> 🚀**运行它吧！** 您将看到一个数字输入框。该值显示前缀和后缀单位，且范围限定在+3到-10之间。



> 图十七：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QSpinBox`

上面的演示代码展示了该控件可用的各种功能。

要设置可接受值的范围，您可以使用 `setMinimum` 和 `setMaximum`，或者使用 `setRange` 同时设置两者。值类型的标注支持在数字前添加前缀或在数字后添加后缀，例如使用 `.setPrefix` 和 `.setSuffix` 分别设置货币标记或单位。

点击控件上的向上和向下箭头可增加或减少控件中的值，该值可使用 `.setSingleStep` 进行设置。请注意，这不会对控件可接受的值产生任何影响。

`QSpinBox` 和 `QDoubleSpinBox` 都具有 `.valueChanged` 信号，该信号在其值发生改变时触发。`.valueChanged` 信号发送数字值（整数或浮点数），而单独的 `.textChanged` 信号则将值作为字符串发送，包括前缀和后缀字符。

QSlider

`QSlider` 提供了一个滑动条控件，其内部功能与 `QDoubleSpinBox` 非常相似。它不会以数字形式显示当前值，而是通过滑块在控件长度上的位置来表示。当需要在两个极端值之间进行调整，但不需要绝对精确度时，此控件非常有用。此类控件最常见的用途是音量控制。

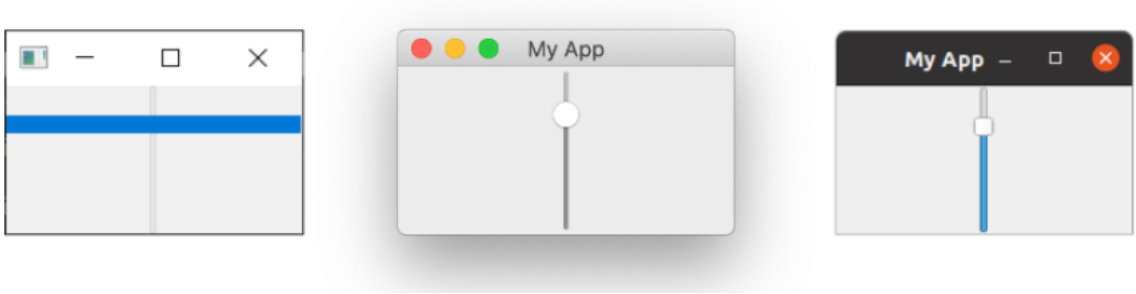
还有一个额外的每当滑块移动位置时触发的 `.sliderMoved` 信号，以及一个每当滑块被点击时发出的 `.sliderPressed` 信号。

Listing 22. basic/widgets_8.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QSlider


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")
```

```
        widget = QSlider()

        widget.setMinimum(-10)
        widget.setMaximum(3)
        # 或者: widget.setRange(-10,3)

        widget.setSingleStep(3)
        widget.valueChanged.connect(self.value_changed)
        widget.sliderMoved.connect(self.slider_position)
        widget.sliderPressed.connect(self.slider_pressed)
        widget.sliderReleased.connect(self.slider_released)

        self.setCentralWidget(widget)

    def value_changed(self, i):
        print(i)

    def slider_position(self, p):
        print("position", p)

    def slider_pressed(self):
        print("Pressed!")

    def slider_released(self):
        print("Released")


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到一个滑块控件。拖动滑块即可更改数值。



> 图十八：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QSlider`。在 *Windows* 中手柄会扩展到控件的大小。

您还可以通过在创建时传递方向来构建垂直或水平方向的滑块。方向标志在 Qt.命名空间中定义。例如——

```
widget.QSlider(Qt.Orientiation.Vertical)
```

或者——

```
widget.QSlider(Qt.Orientiation.Horizontal)
```

QDial

最后，`QDial` 是一个可旋转的控件，功能与滑块相同，但外观为模拟拨盘。它看起来很不错，但从 UI 角度来看并不特别用户友好。然而，它们通常在音频应用程序中用作现实世界中的模拟拨盘的表示。

Listing 23. basic/widgets_9.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QDial, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QDial()
        widget.setRange(-10, 100)
        widget.setSingleStep(1)

        widget.valueChanged.connect(self.value_changed)
        widget.sliderMoved.connect(self.slider_position)
        widget.sliderPressed.connect(self.slider_pressed)
        widget.sliderReleased.connect(self.slider_released)

        self.setCentralWidget(widget)

    def value_changed(self, i):
        print(i)

    def slider_position(self, p):
        print("position", p)

    def slider_pressed(self):
        print("Pressed!")

    def slider_released(self):
        print("Released")


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您会看到一个旋钮，请旋转它以从范围内选择一个数字。

> 图十九：在 *Windows*，*macOS* 和 *Linux(Ubuntu)* 上面的`QDial`

这些信号与 `QSlider` 的信号相同，并保留了相同的名称（例如 `.sliderMoved`）。

以上就是对 PyQt6 中可用的 Qt 控件的简要介绍。要查看可用的控件的完整列表，包括所有信号和属性，请参阅 Qt 文档。

`QWidget`

我们的演示中有一个 QWidget，但您看不到它。我们之前在第一个示例中使用 `QWidget` 创建了一个空窗口。但 `QWidget` 还可以与 布局 一起用作其他控件的容器，以构建窗口或复合控件。我们将在后面更详细地介绍创建自定义控件(22. 自定义控件)。

请记住 `QWidget`，因为您将会频繁地使用它！

6. 布局

到目前为止，我们已经成功创建了一个窗口，并向其中添加了一个控件。但是，通常情况下，您会希望在窗口中添加多个控件，并对添加的控件的位置进行一些控制。在 Qt 中，我们使用布局来排列控件。Qt 中提供了 4 种基本布局，如下表所示。

布局	行为
`QHBoxLayout`	线性水平布局
`QVBoxLayout`	线性垂直布局
`QGridLayout`	在可索引网格XxY中
`QStackedLayout`	堆叠（z）在彼此之前

Qt 中提供了三种二维布局：`QVBoxLayout`、`QHBoxLayout` 和 `QGridLayout`。此外，还有 `QStackedLayout`，它允许您在同一空间内将控件一个叠放在另一个之上，但每次只显示一个控件。

在本章中，我们将依次介绍这些布局，并展示如何使用它们来定位应用程序中的控件。

> *Qt Designer*
>
> 您实际上可以使用Qt Designer以图形方式设计和布局界面，我们将在后续内容中详细介绍。在此我们使用代码，因为这样更便于理解和实验底层系统。

占位符控件

> 请加载一份新的 myapp.py，并将它以新的名字保存下来以供本节使用。

为了更方便地可视化布局，我们将首先创建一个简单的自定义控件来显示我们选择的纯色。这有助于区分我们添加到布局中的控件。请您在与脚本相同的文件夹中创建一个新文件，并将其命名为 `layout_colorwidget.py`，并添加以下代码。我们将在下一个示例中将此代码导入到我们的应用程序中。

Listing 24. basic/layout_colorwidget.py

```
from PyQt6.QtGui import QColor, QPalette
from PyQt6.QtWidgets import QWidget

class Color(QWidget):
    def __init__(self, color):
        super().__init__()
        self.setAutoFillBackground(True)

        palette = self.palette()
        palette.setColor(QPalette.ColorRole.Window, QColor(color))
        self.setPalette(palette)
```

在此代码中，我们子类化 `QWidget` 以创建自己的自定义控件 `Color`。创建控件时，我们接受一个参数——颜色（一个字符串）。首先，我们将 `.setAutoFillBackground` 设置为 `True`，以指示控件自动用窗口颜色填充其背景。接下来，我们将控件的 `QPalette.Window` 颜色更改为我们提供的值 `color` 所描述的新 `QColor`。最后，我们将该调色板应用回控件。最终结果是一个填充了纯色的控件，该颜色是在创建控件时指定的。

如果您觉得以上内容有些难以理解，请不要担心！我们将在后面详细介绍如何创建自定义控件和调色板。目前，您只需了解以下代码即可创建一个实心填充的红色控件即可——

```
Color('red')
```

首先，让我们使用新创建的"颜色"控件将整个窗口填充为单一颜色来测试这个控件。完成之后，我们可以使用 `.setCentralWidget` 将它添加到主窗口，这样就得到了一个纯红色的窗口。

Listing 25. basic/layout_1.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QMainWindow

from layout_colorwidget import Color
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = Color("red")
        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

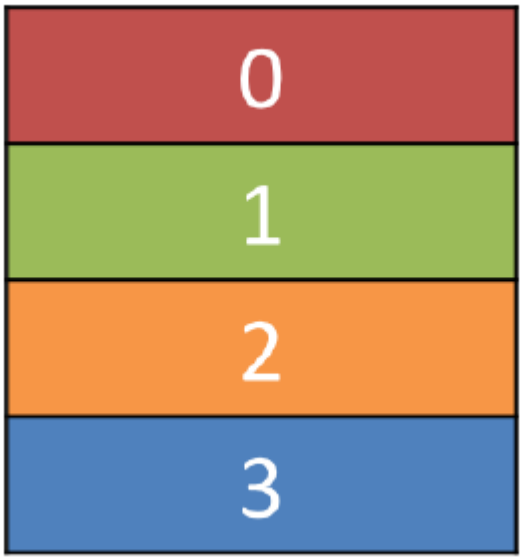
> 🚀 **运行它吧！** 窗口将出现并被完全地填充为红色。请您注意控件如何扩展以填充所有的可用空间。

> 图二十：我们的 `Color` 控件，填充为纯红色。

接下来，我们将依次查看所有可用的 Qt 布局。请注意，要将布局添加到窗口中，我们需要一个占位 `QWidget` 来容纳布局。

`QVBoxLayout` 垂直排列控件

使用 `QVBoxLayout`，您可以将控件线性地排列在彼此之上。添加一个控件会将其添加到列的底部。



> 图二十一：一个按照从上往下顺序填充的 `QVBoxLayout`

将我们的控件添加到布局中。请注意，为了将布局添加到 `QMainWindow`，我们需要将其应用到占位的 `QWidget`。这样，我们就可以使用 `.setCentralWidget` 将控件（和布局）应用到窗口中。我们的彩色控件将在布局中排列，包含在窗口中的 `QWidge` 中。首先，我们像之前一样添加红色控件。

Listing 26. basic/layout_2a.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QVBoxLayout,
    QWidget,
)

from layout_colorwidget import Color

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()

        layout.addWidget(Color("red"))

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
```

```
window.show()

app.exec()
```

> 🚀 **运行它吧！** 请注意，现在红色控件周围显示了边框。这就是布局间距——我们稍后会介绍如何调整它。

> 图二十二：我们在布局中的 `Color` 控件

接下来，在布局中添加一些彩色控件：

Listing 27. basic/layout_2b.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QVBoxLayout,
    QWidget,
)

from layout_colorwidget import Color

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()

        layout.addWidget(Color("red"))
        layout.addWidget(Color("green"))
        layout.addWidget(Color("blue"))

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)


app = QApplication(sys.argv)
```

```
window = MainWindow()
window.show()

app.exec()
```

当我们添加控件时，它们会按照添加的顺序垂直排列。

> 图二十三：三个 `Color` 控件在一个 `QVBoxLayout` 布局中垂直排列

`QHBoxLayout` 水平排列控件

`QHBoxLayout` 与之相同，只是水平移动。添加控件会将其添加到右侧。

> 图二十四：一个从左往右填充的 `QHBoxLayout`

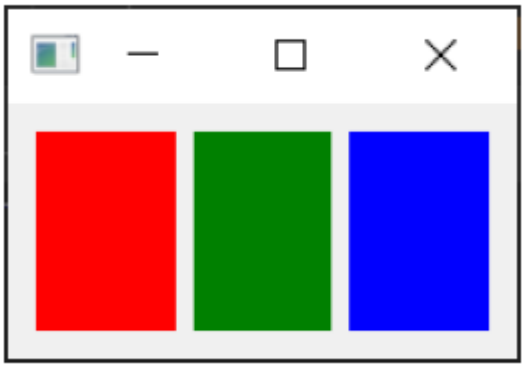
要使用它，我们可以简单地将 `QVBoxLayout` 改为 `QHBoxLayout`。现在，这些框会从左到右排列。

Listing 28. basic/layout_3.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLabel,
    QMainWindow,
    QWidget,
)

from layout_colorwidget import Color

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()
```

```
        layout.addWidget(Color("red"))
        layout.addWidget(Color("green"))
        layout.addWidget(Color("blue"))

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 控件应水平排列



> 图二十五：三个 `Color` 控件在一个 `QVBoxLayout` 布局中水平排列

嵌套布局

对于更复杂的布局，您可以使用 `.addLayout` 在布局中嵌套布局。下面，我们将 `QVBoxLayout` 添加到主 `QHBoxLayout` 中。如果我们将一些控件添加到 `QVBoxLayout`，它们将垂直排列在父布局的第一个槽中。

Listing 29. basic/layout_4.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLabel,
    QMainWindow,
    QVBoxLayout,
    QWidget,
)

from layout_colorwidget import Color


class MainWindow(QMainWindow):
```

```
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout1 = QHBoxLayout()
        layout2 = QVBoxLayout()
        layout3 = QVBoxLayout()

        layout2.addWidget(Color("red"))
        layout2.addWidget(Color("yellow"))
        layout2.addWidget(Color("purple"))

        layout1.addLayout(layout2)
        layout1.addWidget(Color("green"))

        layout3.addWidget(Color("red"))
        layout3.addWidget(Color("purple"))

        layout1.addLayout(layout3)

        widget = QWidget()
        widget.setLayout(layout1)
        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 控件应水平排列成 3 列，第一列还应包含 3 个垂直堆叠的控件。请尝试一下吧！

> 图二十六：嵌套的 `QHBoxLayout` 和 `QVBoxLayout` 布局。

您可以使用 `.setContentMargins` 设置布局周围的间距，或使用 `.setSpacing` 设置元素之间的间距。

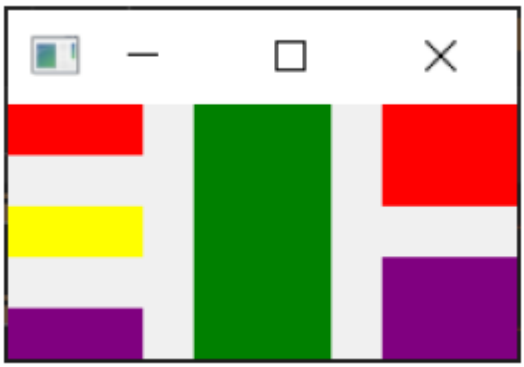
```
layout1.setContentsMargins(0,0,0,0)
layout1.setSpacing(20)
```

以下代码显示了嵌套控件与布局边距和间距的组合。

Listing 30. basic/layout_5.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLabel,
    QMainWindow,
    QVBoxLayout,
    QWidget,
)

from layout_colorwidget import Color

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout1 = QHBoxLayout()
        layout2 = QVBoxLayout()
        layout3 = QVBoxLayout()

        layout1.setContentsMargins(0,0,0,0)
        layout1.setSpacing(20)

        layout2.addWidget(Color("red"))
        layout2.addWidget(Color("yellow"))
        layout2.addWidget(Color("purple"))

        layout1.addLayout(layout2)
        layout1.addWidget(Color("green"))

        layout3.addWidget(Color("red"))
        layout3.addWidget(Color("purple"))

        layout1.addLayout(layout3)

        widget = QWidget()
        widget.setLayout(layout1)
        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

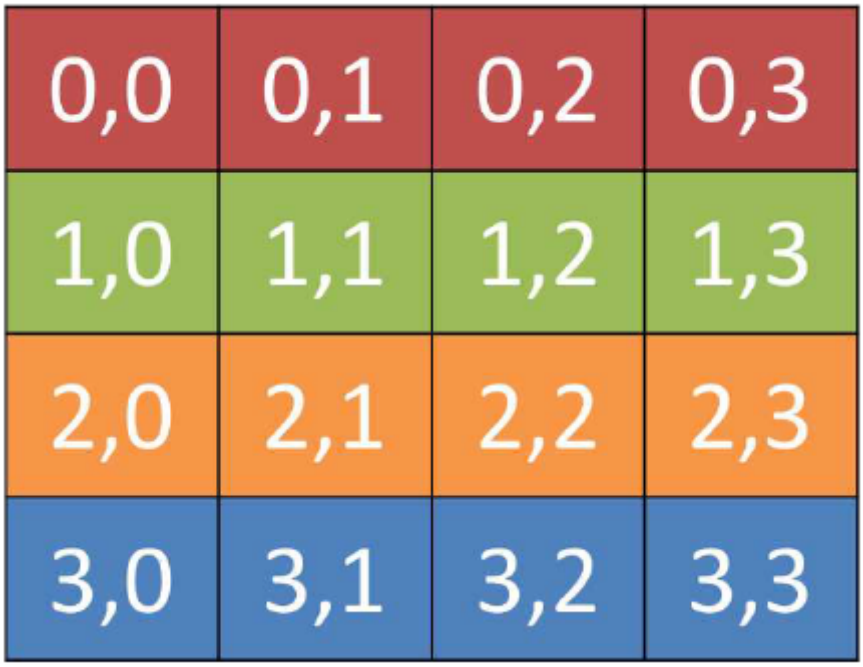
> 🚀 **运行它吧！** 您应该观察间距和边距的效果。请您尝试调整数值，直到您对它们有了一定的把握。



> 图二十七：嵌套的 `QHBoxLayout` 和 `QVBoxLayout` 布局，在控件周围留有间距和边距

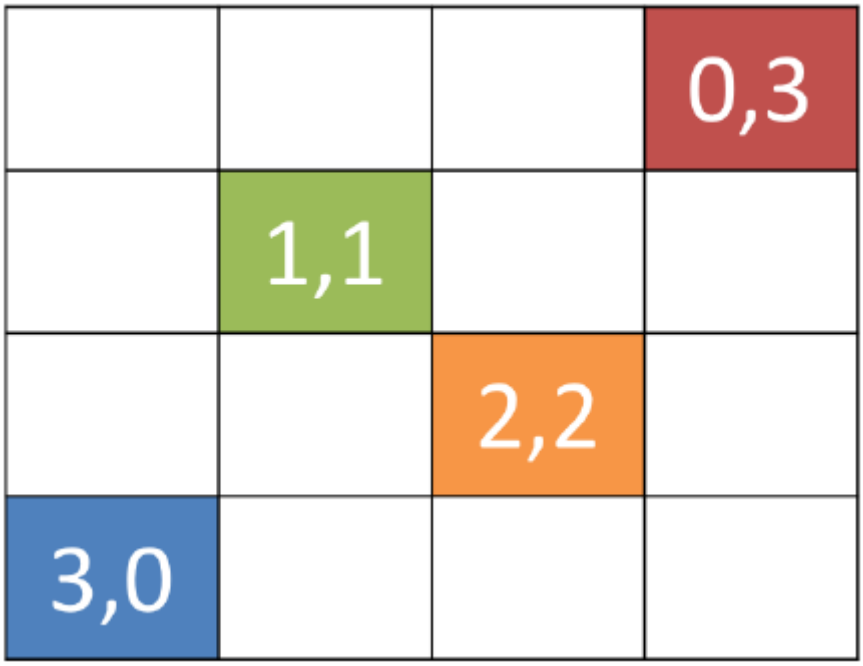
`QGridLayout` 控件以网格形式排列

尽管它们非常有用，但如果您尝试使用 `QVBoxLayout` 和 `QHBoxLayout` 来布局多个元素（例如表单），您会发现很难确保不同大小的控件对齐。解决此问题的办法是使用 `QGridLayout`。



> 图二十八：一个用于显示每个位置的网格位置的 `QGridLayout`

`QGridLayout` 允许您在网格中特定地放置项目。您可以为每个控件指定行和列位置。您可以跳过某些元素，它们将被留空。



图二十九：有未填充槽的 `QGridLayout`

Listing 31. basic/layout_6.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QGridLayout,
    QLabel,
    QMainWindow,
    QWidget,
)

from layout_colorwidget import Color


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QGridLayout()

        layout.addWidget(Color("red"), 0, 0)
        layout.addWidget(Color("green"), 1, 0)
        layout.addWidget(Color("blue"), 1, 1)
        layout.addWidget(Color("purple"), 2, 1)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

```
app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您应该看到控件以网格形式排列，尽管缺少条目，但仍然对齐。

> 图三十：在一个 `QGridLayout` 的四个 `Color` 控件

`QStackedLayout` 在同一空间中放置多个控件

我们将介绍的最后一种布局是 `QStackedLayout`。如上所述，这种布局允许您将元素直接放置在彼此前面。然后，您可以选择要显示的控件。您可以在图形应用程序中使用它来绘制图层，或模仿标签式界面。请注意，还有 `QStackedWidget`，这是一个完全以相同方式工作的容器控件。如果您希望直接将一个栈添加到 `QMainWindow` 中，可以使用 `.setCentralWidget` 方法。

> 图三十一：`QStackedLayout` —— 使用时，只有最上面的控件可见，默认情况下，这是添加到布局中的第一个控件。

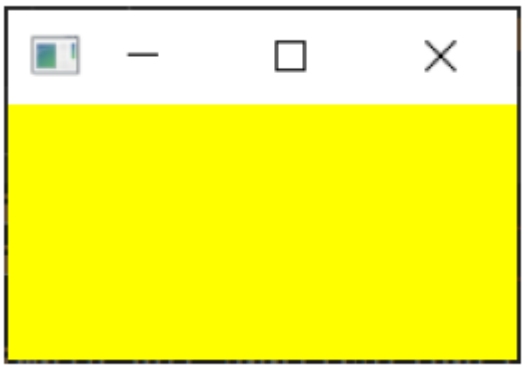
> 图三十二：`QStackedLayout`，可以选择第二个（图中标号为1）控件并将其置于最前面。

Listing 32. basic/layout_7.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QStackedLayout,
    QWidget,
)

from layout_colorwidget import Color


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QStackedLayout()

        layout.addWidget(Color("red"))
        layout.addWidget(Color("green"))
        layout.addWidget(Color("blue"))
        layout.addWidget(Color("yellow"))
```

```
        layout.setCurrentIndex(3)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您只会看到最后添加的控件。



> 图三十三：堆栈控件，仅显示一个控件（最后添加的控件）。

`QStackedWidget` 是应用程序中标签视图的工作方式。任何时候只能看到一个视图（“标签”）。您可以随时使用 `.setCurrentIndex()` 或 `.setCurrentWidget()` 通过索引（按控件添加的顺序）或控件本身来设置项目，从而控制要显示的控件。

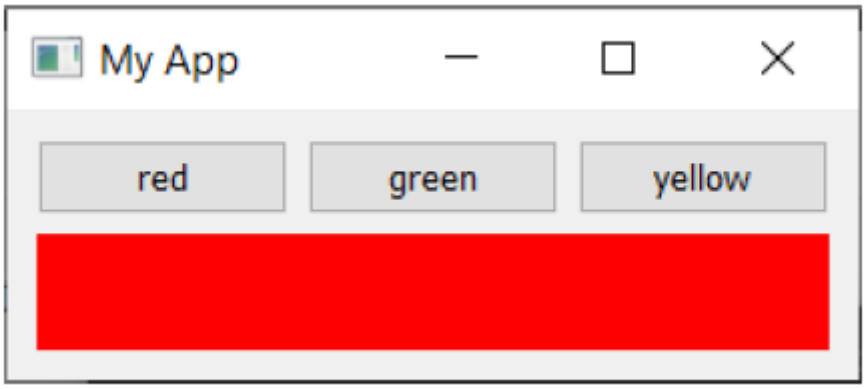
以下是一个简短的演示，使用 `QStackedLayout` 与 `QButton` 结合，为应用程序提供一个类似标签页的界面——

Listing 33. basic/layout_8.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLabel,
    QMainWindow,
    QPushButton,
    QStackedLayout,
    QVBoxLayout,
    QWidget,
)

from layout_colorwidget import Color


class MainWindow(QMainWindow):
```

```
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        pagelayout = QVBoxLayout()
        button_layout = QHBoxLayout()
        self.stacklayout = QStackedLayout()

        pagelayout.addLayout(button_layout)
        pagelayout.addLayout(self.stacklayout)

        btn = QPushButton("red")
        btn.pressed.connect(self.activate_tab_1)
        button_layout.addWidget(btn)
        self.stacklayout.addWidget(Color("red"))

        btn = QPushButton("green")
        btn.pressed.connect(self.activate_tab_2)
        button_layout.addWidget(btn)
        self.stacklayout.addWidget(Color("green"))

        btn = QPushButton("yellow")
        btn.pressed.connect(self.activate_tab_3)
        button_layout.addWidget(btn)
        self.stacklayout.addWidget(Color("yellow"))

        widget = QWidget()
        widget.setLayout(pagelayout)
        self.setCentralWidget(widget)

    def activate_tab_1(self):
        self.stacklayout.setCurrentIndex(0)

    def activate_tab_2(self):
        self.stacklayout.setCurrentIndex(1)

    def activate_tab_3(self):
        self.stacklayout.setCurrentIndex(2)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 现在，您可以使用按钮更改可见控件。



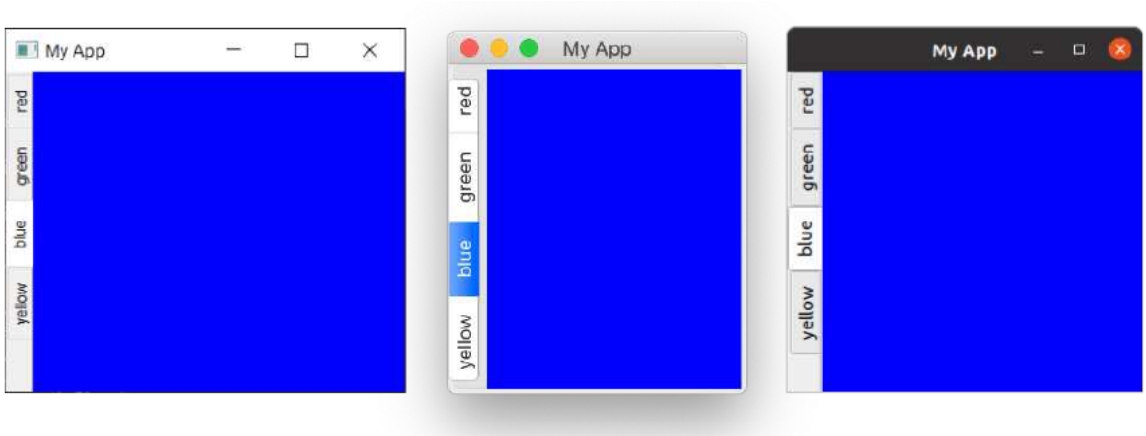
> 图三十四：一个堆栈控件，带有用于控制活动控件的按钮。

Qt 提供了一个内置的选项卡控件，可以提供这种布局，非常方便——尽管它实际上是一个控件，而不是一个布局。下面的选项卡演示是使用 `QTabWidget` 重新创建的——

Listing 34. basic/layout_9.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QTabWidget,
    QWidget,
)

from layout_colorwidget import Color


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        tabs = QTabWidget()
        tabs.setTabPosition(QTabWidget.TabPosition.West)
        tabs.setMovable(True)

        for n, color in enumerate(["red", "green", "blue", "yellow"]):
            tabs.addTab(Color(color), color)

        self.setCentralWidget(tabs)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

如您所见，这种方式更加直观——也更具吸引力！您可以通过设置方向来调整标签的位置，并通过 `.setMoveable` 方法切换标签是否可移动。

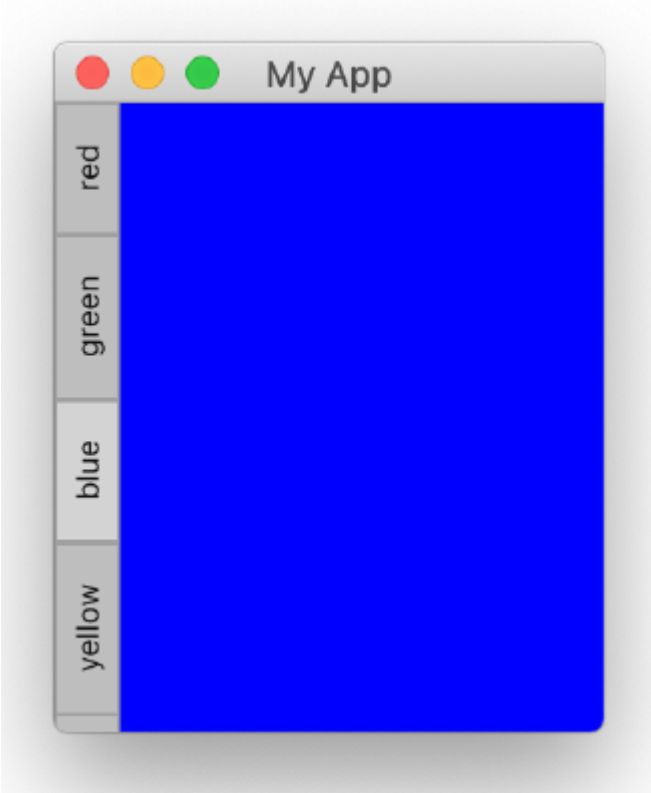


> 图三十五：包含我们控件的 QTabWidget，标签显示在左侧（西侧）。屏幕截图显示了在 *Windows*、*macOS* 和 *Ubuntu* 上的外观。

您会发现 macOS 标签栏与其他平台的标签栏外观差异显著——在macOS 系统中，标签默认采用药丸形或气泡形样式。在 macOS 系统中，此样式通常用于标签式配置面板。对于文档，您可以启用文档模式，以获得与其他平台类似的纤薄标签样式。此选项对其他平台无影响。

Listing 35. basic/layout_9b.py

```
        tabs = QTabWidget()
        tabs.setDocumentMode(True)
```



> 图三十六：在 macOS 上，`QTabWidget` 的文档模式设置为 `True`。

7. 操作、工具栏与菜单

接下来，我们将探讨一些常见的用户界面元素，这些元素您可能在许多其他应用程序中都见过——工具栏和菜单。我们还将探索Qt提供的用于减少不同用户界面区域之间重复性的便捷系统——`QAction`。

工具栏

最常见的用户界面元素之一是工具栏。工具栏是由图标和/或文本组成的条形控件，用于在应用程序中执行常见任务，而通过菜单访问这些任务会显得繁琐。它们是许多应用程序中最为常见的用户界面功能之一。尽管一些复杂的应用程序，特别是微软Office套件中的应用程序，已迁移到基于上下文的"功能区"界面，但对于您将创建的大多数应用程序而言，标准工具栏已足够使用。

> 图三十七：标准图形用户界面元素——工具栏

Qt 工具栏支持显示图标、文本，还可以包含任何标准的 Qt 控件。但是，对于按钮而言，最好的方法是利用 `QAction` 系统将按钮放置在工具栏上。

让我们先为应用程序添加一个工具栏。

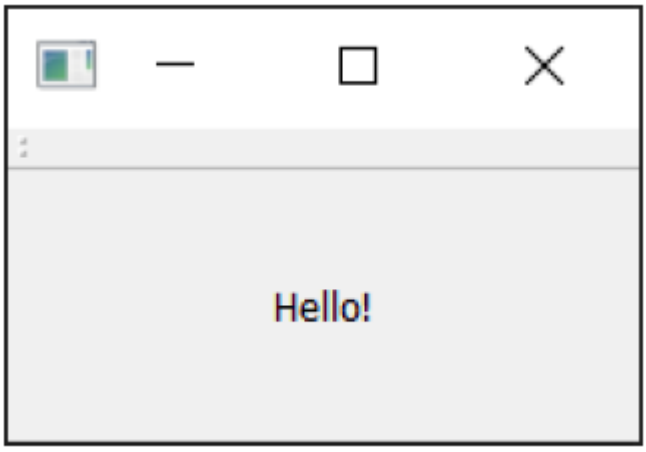
> 请您加载一个全新的 `myapp.py` 副本，并将其保存为新名称以供本节使用。

在 Qt 中，工具栏是通过 `QToolBar` 类创建的。首先，您需要创建该类的一个实例，然后调用 `QMainWindow` 的 `.addToolbar` 方法。将一个字符串作为第一个参数传递给 `QToolBar` 类，即可设置工具栏的名称，该名称将用于在用户界面中识别该工具栏。

Listing 36. basic/toolbars_and_menus_1.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        self.addToolBar(toolbar)

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

> 🚀 **运行它吧！** 您会在窗口顶部看到一条细长的灰色条。这就是您的工具栏。右键点击并点击名称即可将其关闭。



> 图三十八：一个带有工具栏的窗口

> 我无法恢复我的工具栏了！？
>
> 不幸的是，一旦您移除了工具栏，现在就没有地方可以右键点击来重新添加它。因此，作为一个通用的规则，您应该要么保留一个不可移除的工具栏，要么提供一个替代界面来开启或关闭工具栏。

让我们让工具栏变得更有趣一些。我们只需添加一个 `QButton` 控件即可，但 Qt 中还有一种更好的方法可以为您提供一些很酷的功能——那就是通过 `QAction`。`QAction` 是一个提供描述抽象用户界面的方法的类。这意味着，您可以在一个对象中定义多个界面元素，并通过与该元素交互的效果将它们统一起来。例如，工具栏和菜单中通常都会出现一些功能，例如“编辑→剪切”，它既存在于“编辑”菜单中，也以剪刀图标的形式出现在工具栏上，同时还支持键盘快捷键 `Ctrl-X`（macOS 上为 `Cmd-X`）。

如果没有 `QAction`，您必须在多个地方定义此操作。但使用 `QAction`，您就可以只定义一个 `QAction`，定义触发操作，然后将此操作添加到菜单和工具栏中。每个 `QAction` 都有名称、状态消息、图标和可连接的信号（以及更多内容）。

请参阅下面的代码，了解如何添加您的第一个 `QAction`。

Listing 37. basic/toolbars_and_menus_2.py

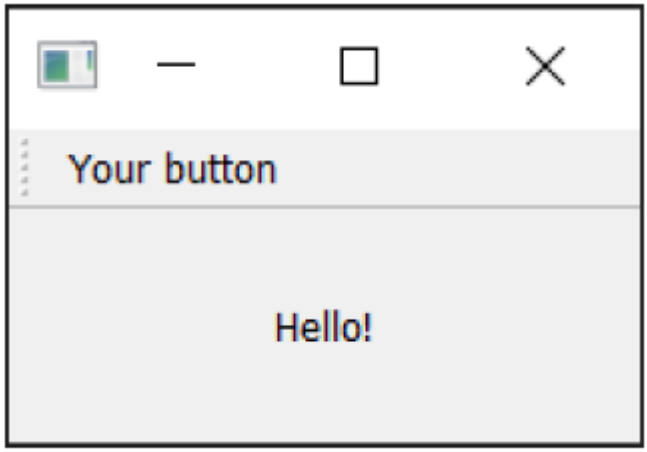
```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        self.addToolBar(toolbar)
```

```
        button_action = QAction("Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        toolbar.addAction(button_action)

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

首先，我们创建一个函数来接受来自 `QAction` 的信号，以便查看它是否正常工作。接下来，我们定义 `QAction` 本身。在创建实例时，我们可以传递一个动作标签和/或图标。你还必须传递任何 `QObject` 作为动作的父对象——这里我们传递 self 作为对主窗口的引用。对于 `QAction` 来说，父对象作为最后一个参数传递，这有点奇怪。

接下来，我们可以选择设置一个状态提示——一旦有状态栏，该文本就会显示在状态栏上。最后，我们将 `.triggered` 信号连接到自定义函数。每当 `QAction` 被"触发"（或激活）时，该信号就会触发。

> 🚀 **运行它吧！** 您应该看到带有您定义的标签的按钮。如果您点击它，我们的自定义函数就会触发"点击"事件并返回按钮的状态。



> 图三十九：我们的 `QAction` 按钮在工具栏中显示出来了

> 为什么信号总是为假？
>
> 传递的信号表明该操作是否被选中，而由于我们的按钮不能被选中——只能点击——因此它总是为假。这就像我们之前看到的 `QPushButton` 一样。

让我们添加一个状态栏。

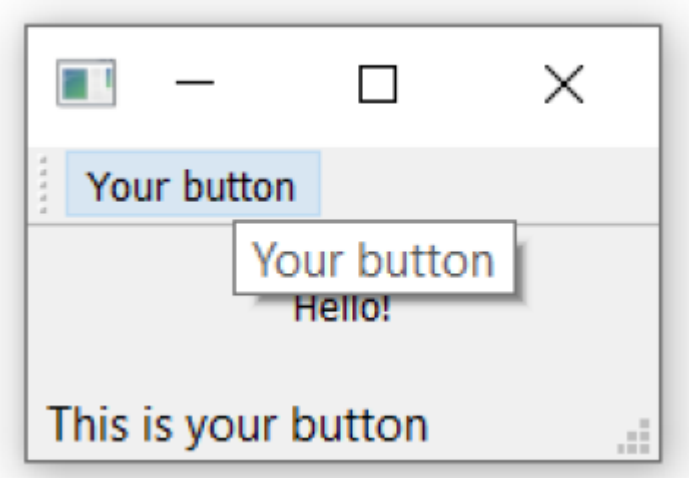
我们通过调用 `QStatusBar` 并将其结果传递给 `.setStatusBar` 来创建状态栏对象。由于我们不需要修改状态栏设置，因此可以在创建时直接将其传递进去。我们可以在一行代码中创建并定义状态栏：

Listing 38. basic/toolbars_and_menus_3.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        self.addToolBar(toolbar)

        button_action = QAction("Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        toolbar.addAction(button_action)

        self.setStatusBar(QStatusBar(self))

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

> 🚀 **运行它吧！** 将鼠标悬停在工具栏按钮上，您将看到状态文本在窗口底部的状态栏中显示。



图四十：状态栏文本会在我们悬停操作时更新。

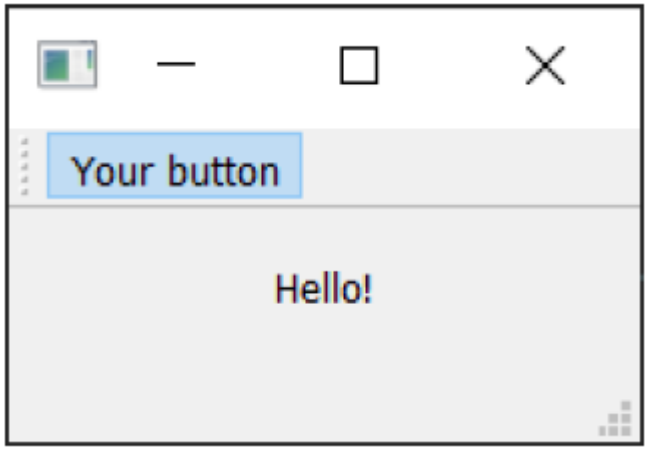
接下来，我们将把 `QAction` 设置为可切换的——点击一次会将其打开，再次点击会将其关闭。要实现这一点，我们只需在 `QAction` 对象上调用 `setCheckable(True)` 方法。

Listing 39. basic/toolbars_and_menus_4.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
```

```
        self.addToolBar(toolbar)

        button_action = QAction("Your button", self)
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        self.setStatusBar(QStatusBar(self))

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

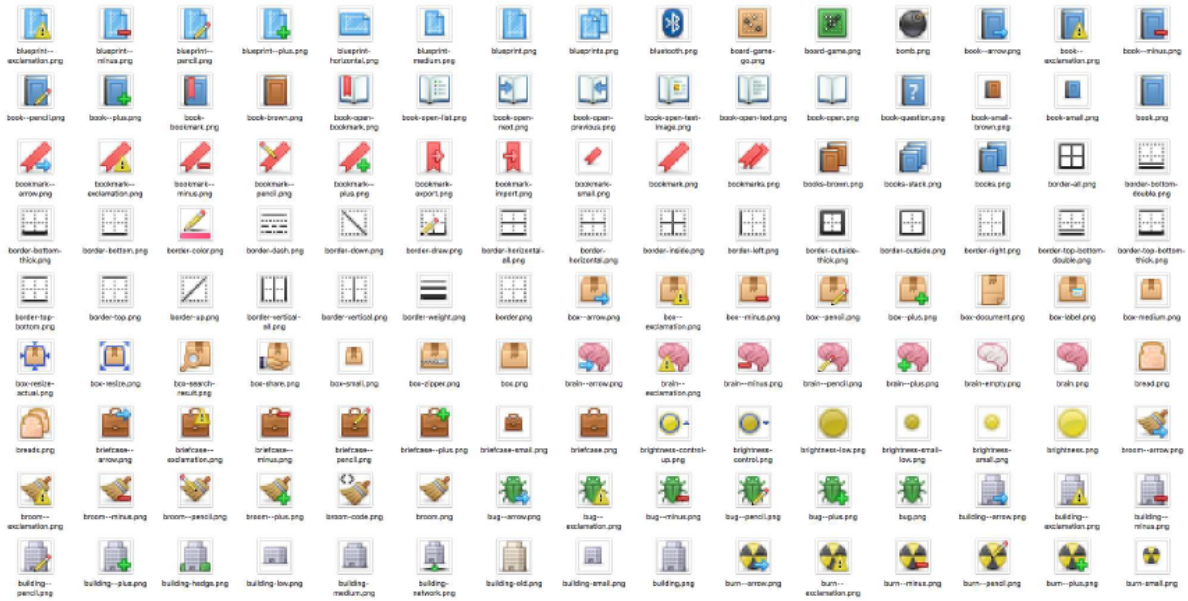
> 🚀 **运行它吧！** 请您点击按钮，查看它从选中状态切换到未选中状态。请注意，我们现在创建的自定义槽函数交替输出 `True` 和 `False`。



图四十一：工具栏按钮已启用

> `.toggled` 信号
>
> 还有一个 `.toggled` 信号，只有在按钮被切换时才会发出信号。但效果相同，因此基本上毫无意义。

目前看起来有点无聊，所以让我们给按钮添加一个图标。为此，我推荐设计师Yusuke Kamiyamane设计的 Fugue图标集。这是一个非常棒的16x16像素图标集，可以为你的应用程序增添专业气质。该图标集可免费使用，只需在分发应用程序时注明出处即可——不过我相信，如果条件允许，设计师也会很乐意收到你的捐赠。



图四十二：设计师 Yusuke Kamiyamane 的作品：Fugue图标集

从图像集（在此示例中我选择了文件 `bug.png`）中选择一张图像，并将它复制到与源代码相同的文件夹中。我们可以创建一个 `QIcon` 对象，通过将文件路径传递给该类来实现。我们使用在 "控件" 一章中学习到的 `basedir` 技术加载图标。这确保无论您从何处运行脚本，都能找到该文件。最后，要将图标添加到 `QAction`（以及按钮）中，只需在创建 `QAction` 时将其作为第一个参数传递即可。

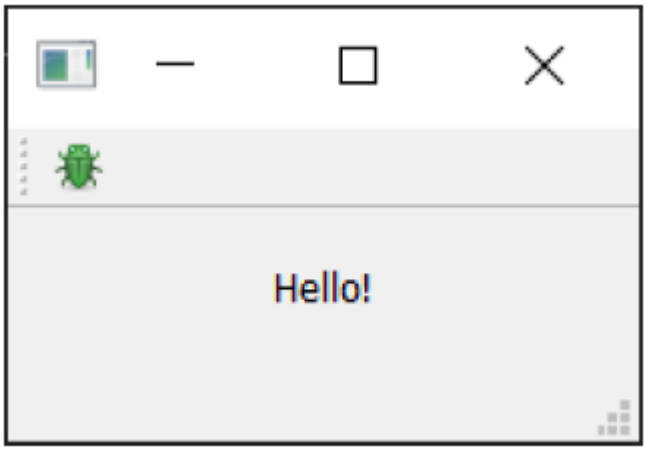
您还需要让工具栏知道你的图标大小，否则您的图标周围会出现大量填充。您可以通过调用 `.setIconSize()` 并传入一个 `QSize` 对象来实现这一点。

Listing 40. basic/toolbars_and_menus_5.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtGui import QAction, QIcon
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QStatusBar,
    QToolBar,
)

basedir = os.path.dirname(__file__)

# tag::MainWindow[]
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)
```

```
        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16, 16))
        self.addToolBar(toolbar)

        button_action = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your button",
            self,
        )
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        self.setStatusBar(QStatusBar(self))

    def onMyToolBarButtonClick(self, s):
        print("click", s)


# end::MainWindow[]

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** `QAction` 现在以图标形式显示。所有功能均与之前完全相同。



图四十三：我们带有一个图标的操作按钮

请注意，Qt 使用操作系统的默认设置来确定是否在工具栏中显示图标、文本或图标和文本。但您可以使用 `.setToolButtonStyle` 覆盖此设置。该槽接受来自 `Qt.` 命名空间的以下任何标志：

标志	行为
`Qt.ToolButtonStyle.ToolButtonIconOnly`	仅有图标，没有文本
`Qt.ToolButtonStyle.ToolButtonTextOnly`	仅有文本，没有图标

标志	行为
`Qt.ToolButtonStyle.ToolButtonTextBesideIcon`	同时存在文本和图标，且文本在图标旁边
`Qt.ToolButtonStyle.ToolButtonTextUnderIcon`	同时存在文本和图标，且文本在图标下面
`Qt.ToolButtonStyle.ToolButtonFollowStyle`	跟随本地桌面的样式

> 我应该使用哪种样式?
>
> 默认值为 `Qt.ToolButtonStyle.ToolButtonFollowStyle`，这意味着您的应用程序将默认遵循应用程序运行所在桌面的标准/全局设置。这通常被推荐以使您的应用程序尽可能地与**系统风格**一致。

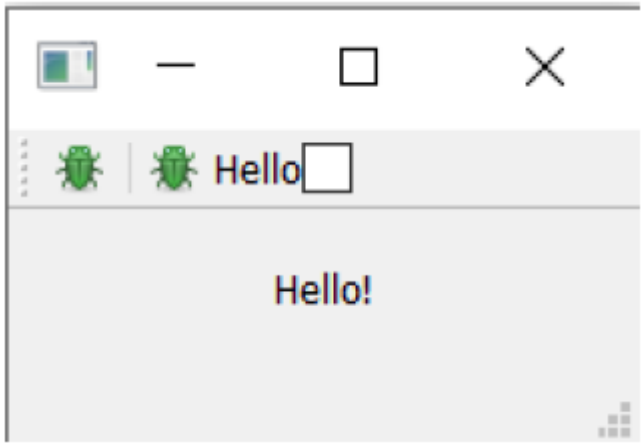
接下来，我们将向工具栏添加一些其他元素。我们将添加第二个按钮和复选框控件。如前所述，您可以在此处添加任何控件，因此请随意发挥创意。

Listing 41. basic/toolbars_and_menus_6.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16, 16))
        self.addToolBar(toolbar)

        button_action = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your button",
            self,
        )
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addSeparator()
```

```
        button_action2 = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your button2",
            self,
        )
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action2)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())

        self.setStatusBar(QStatusBar(self))

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

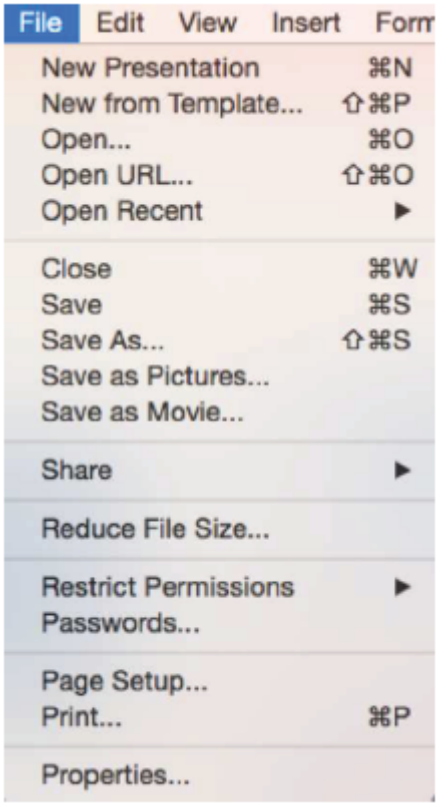
> 🚀 **运行它吧！** 现在您可以看到多个按钮和一个复选框。



> 图四十四：带有一个操作和两个控件的工具栏。

菜单

菜单是用户界面的另一个标准组件。通常它们位于窗口顶部，或在macOS系统中位于屏幕顶部。它们允许访问所有标准应用程序功能。存在一些标准菜单——例如文件、编辑、帮助。菜单可以嵌套以创建功能的分层树结构，并且它们通常支持并显示键盘快捷键以快速访问其功能。



> 图四十五：标准图形用户界面元素——菜单

要创建菜单，我们需要在 `QMainWindow` 上调用 `.menuBar()` 方法来创建菜单栏。我们通过调用 `.addMenu()` 方法并传入菜单名称来在菜单栏上添加菜单。我将其命名为 `'&File'`。这里的 `&` 符号定义了快捷键，按下 Alt 键时可快速跳转到该菜单。

> macOS 上的快捷键
>
> 这在 macOS 上是不可见的。请注意，这与键盘快捷键不同——我们稍后会详细介绍。

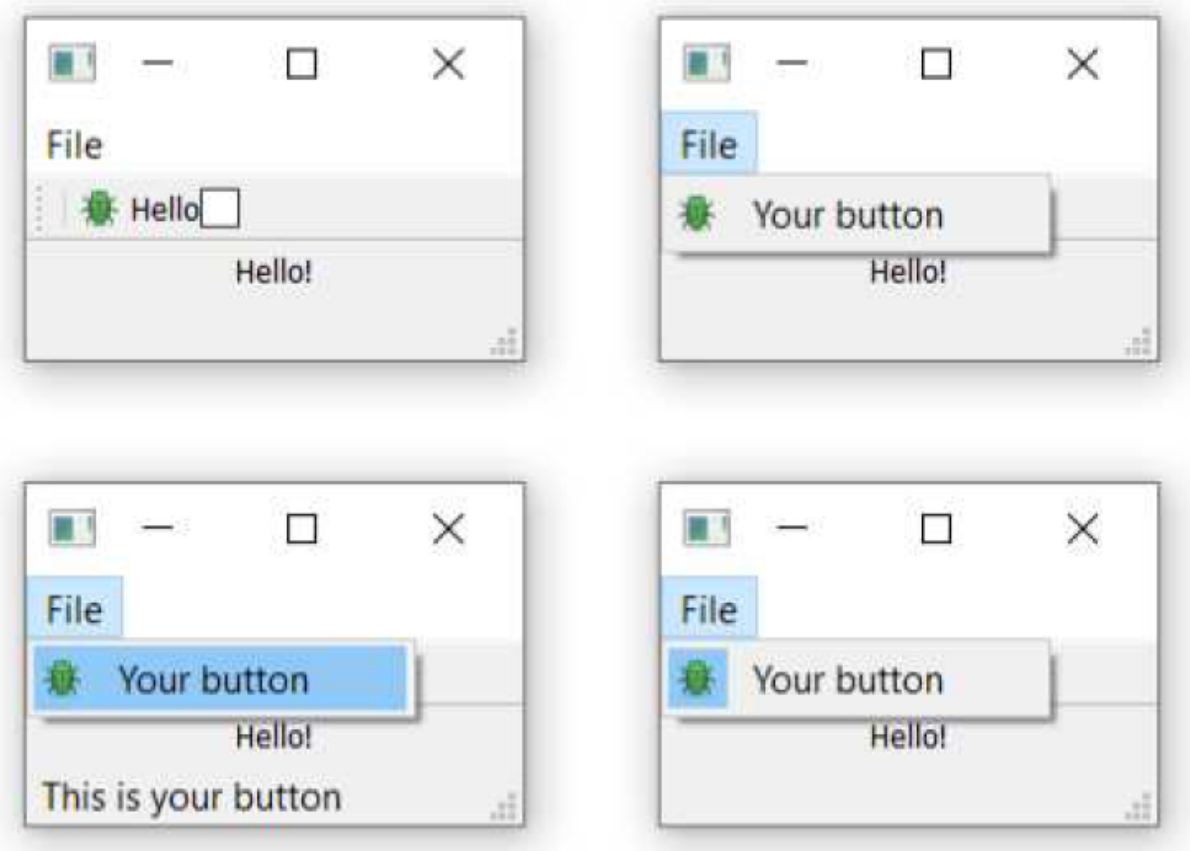
这就是操作功能发挥作用的地方。我们可以复用已有的 `QAction` 来为菜单添加相同的功能。要添加操作，只需调用 `.addAction` 并传入我们定义的操作之一。

Listing 42. basic/toolbars_and_menus_7.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)
```

```
        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16, 16))
        self.addToolBar(toolbar)

        button_action = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "&Your button",
            self,
        )
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addSeparator()

        button_action2 = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your &button2",
            self,
        )
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action2)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())

        self.setStatusBar(QStatusBar(self))

        menu = self.menuBar()

        file_menu = menu.addMenu("&File")
        file_menu.addAction(button_action)

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

点击菜单中的选项时，您会发现该选项可切换状态——它继承了 `QAction` 的特性。



> 图四十六：窗口上显示的菜单 — 在 *macOS* 上，该菜单将位于屏幕顶部。

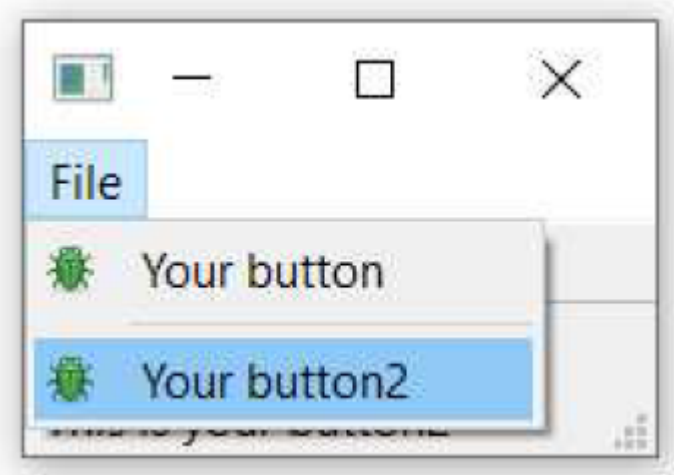
让我们在菜单中添加更多内容。这里我们将为菜单添加一个分隔符，它将在菜单中显示为一条水平线，然后添加我们创建的第二个 `QAction` 。

Listing 43. basic/toolbars_and_menus_8.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16, 16))
        self.addToolBar(toolbar)

        button_action = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "&Your button",
            self,
        )
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)
```

```
        toolbar.addSeparator()

        button_action2 = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your &button2",
            self,
        )
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action2)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())

        self.setStatusBar(QStatusBar(self))

        menu = self.menuBar()

        file_menu = menu.addMenu("&File")
        file_menu.addAction(button_action)
        file_menu.addSeparator()
        file_menu.addAction(button_action2)

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

> 🚀 **运行它吧！** 您应该看到两个菜单项，它们之间应该会有一条分隔线。



> 图四十七：我们的操作在菜单中显示出来了

您还可以使用“&”符号为菜单添加快捷键，以便在菜单打开时，只需按下一个键即可跳转到菜单项。同样，此功能在 macOS 上不适用。

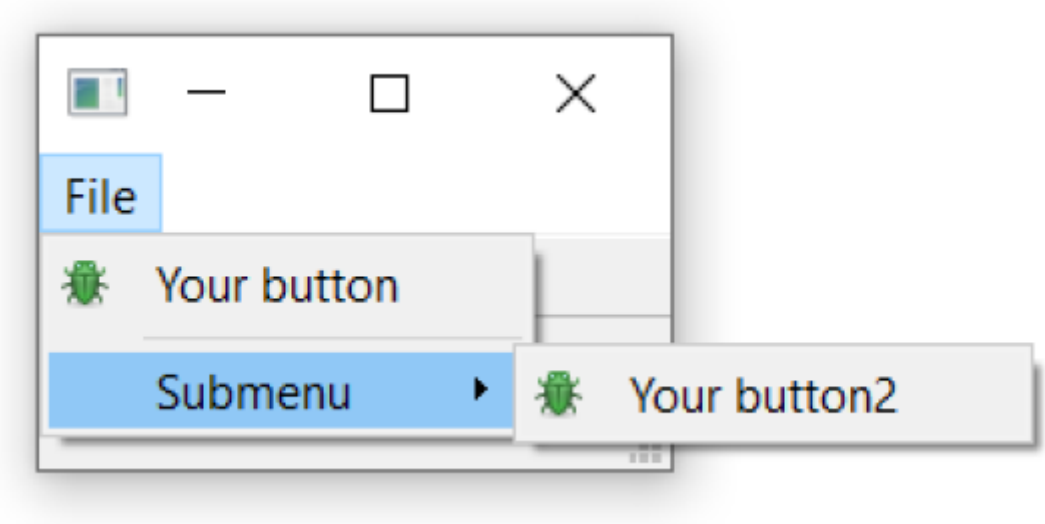
要添加子菜单，只需通过调用父菜单的 addMenu() 方法创建一个新菜单。然后您可以像往常一样向其中添加操作项。例如：

Listing 44. basic/toolbars_and_menus_9.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16, 16))
        self.addToolBar(toolbar)

        button_action = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "&Your button",
            self,
        )
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addSeparator()

        button_action2 = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your &button2",
            self,
        )
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action2)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())

        self.setStatusBar(QStatusBar(self))

        menu = self.menuBar()

        file_menu = menu.addMenu("&File")
        file_menu.addAction(button_action)
        file_menu.addSeparator()

        file_submenu = file_menu.addMenu("Submenu")
        file_submenu.addAction(button_action2)

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

如果您现在运行这个示例，并将鼠标悬停在文件菜单中的子菜单条目上，您会看到一个单条目子菜单出现，其中包含我们的第二个操作。您可以继续向这个子菜单添加条目，与添加顶级菜单条目时的方式相同。



图四十八：文件菜单中的嵌套子菜单。

最后，我们将为 `QAction` 添加一个键盘快捷键。您可以通过调用 `setKeySequence()` 并传入键盘序列来定义键盘快捷键。任何已定义的键盘序列都将显示在菜单中。

隐藏的快捷键

请注意，键盘快捷键与 `QAction` 相关联，无论 `QAction` 是否被添加到菜单或工具栏中，它都有效。

键序列可以通过多种方式定义——作为文本传递、使用Qt 命名空间中的键名，或者使用Qt 命名空间中定义的键序列。请您尽可能使用后一种方式，以确保符合操作系统标准。

以下是完成后的代码，显示了工具栏按钮和菜单。

Listing 45. basic/toolbars_and_menus_end.py

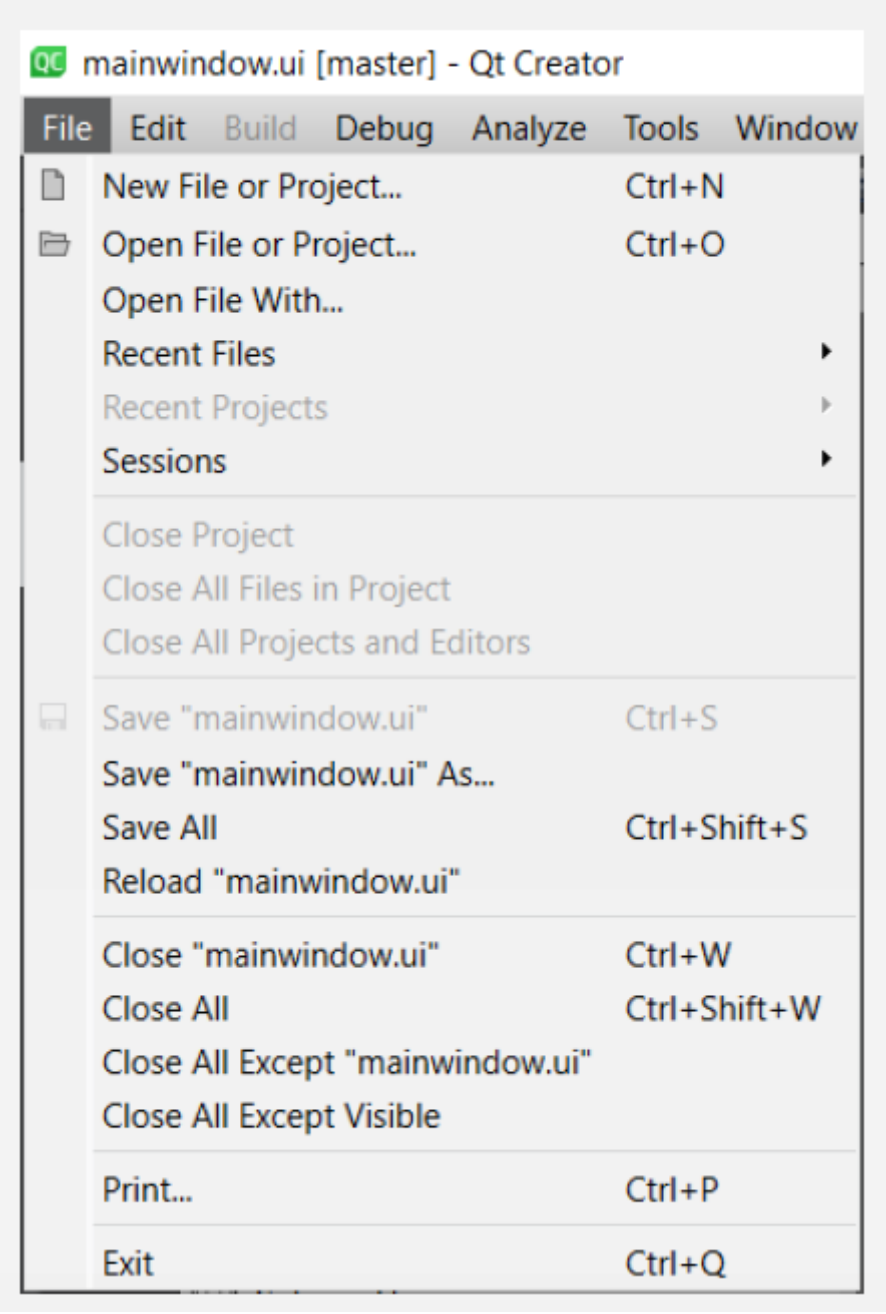
```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        label = QLabel("Hello!")
        # Qt 命名空间有许多用于自定义控件的属性。参见：http://doc.qt.io/qt-5/qt.html
        label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        # 设置窗口的中央控件。默认情况下，控件将扩展以占据窗口中的所有空间。
        self.setCentralWidget(label)

        toolbar = QToolBar("My main toolbar")
        toolbar.setIconSize(QSize(16, 16))
        self.addToolBar(toolbar)
```

```
        button_action = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "&Your button",
            self,
        )
        button_action.setStatusTip("This is your button")
        button_action.triggered.connect(self.onMyToolBarButtonClick)
        button_action.setCheckable(True)

        # 您可以使用键盘名称输入快捷键，例如Ctrl+p
        # Qt.命名空间标识符（例如 Qt.CTRL + Qt.Key_P）
        # 或系统无关标识符（例如 QKeySequence.Print）
        button_action.setShortcut(QKeySequence("Ctrl+p"))
        toolbar.addAction(button_action)

        toolbar.addSeparator()

        button_action2 = QAction(
            QIcon(os.path.join(basedir, "bug.png")),
            "Your &button2",
            self,
        )
        button_action2.setStatusTip("This is your button2")
        button_action2.triggered.connect(self.onMyToolBarButtonClick)
        button_action2.setCheckable(True)
        toolbar.addAction(button_action)

        toolbar.addWidget(QLabel("Hello"))
        toolbar.addWidget(QCheckBox())
        self.setStatusBar(QStatusBar(self))

        menu = self.menuBar()

        file_menu = menu.addMenu("&File")
        file_menu.addAction(button_action)

        file_menu.addSeparator()

        file_submenu = file_menu.addMenu("Submenu")

        file_submenu.addAction(button_action2)

    def onMyToolBarButtonClick(self, s):
        print("click", s)
```

菜单与工具栏的组织管理

如果用户无法找到应用程序的操作，他们就无法充分发挥应用程序的全部功能。让操作易于发现是创建用户友好型应用程序的关键。一个常见的错误是试图通过在应用程序的各个地方添加操作来解决这个问题，结果反而让用户感到困惑和不知所措。

请您将常见且必要的操作放在首位，确保它们易于查找和回忆。想想大多数编辑应用程序中的“文件”>“新建”选项。它位于“文件”菜单顶部，并绑定了一个简单的键盘快捷键 **Ctrl + N**。如果 “新建文档…” **需要通过“文件”>“常用操作”>“文件操作”>“当前文档”>“新建”**或快捷键 **Ctrl +Alt + J** 才能访问，用户将难以找到它。



> Qt Creator 中的文件菜单部分，请注意常见操作位于顶部，不常用的操作则位于下方。

如果您把“文件”>“保存”菜单隐藏得像这样，您的用户就更不可能保存他们的作品，而更有可能丢失它们——这是字面意义上和比喻意义上的！请您看看您电脑上现有的应用程序，来获取灵感。但您要保持批判性眼光，因为市面上充斥着大量设计糟糕的软件。

请您在菜单和工具栏中使用逻辑分组，这样可以更轻松地找到所需内容。在少数几个选项中找到某物比在长列表中更容易。

> Qt Designer 中的分组工具栏

避免在多个菜单中重复相同操作，因为这会让它们的用途变得模糊——“这些操作是不是做同样的事情？”——即使它们的标签完全相同。最后，不要试图通过动态隐藏/移除菜单项来简化菜单。这会导致用户在寻找不存在的选项时感到困惑“……刚才还在这里”。不同状态应通过禁用菜单项、使用独立窗口、清晰区分的界面模式或对话框来表示。

请将菜单按层次结构组织，并逻辑地分组操作。

请将最常用的功能复制到工具栏上。

请在菜单中禁用无法使用的项目。

请勿将同一操作添加到多个菜单中。

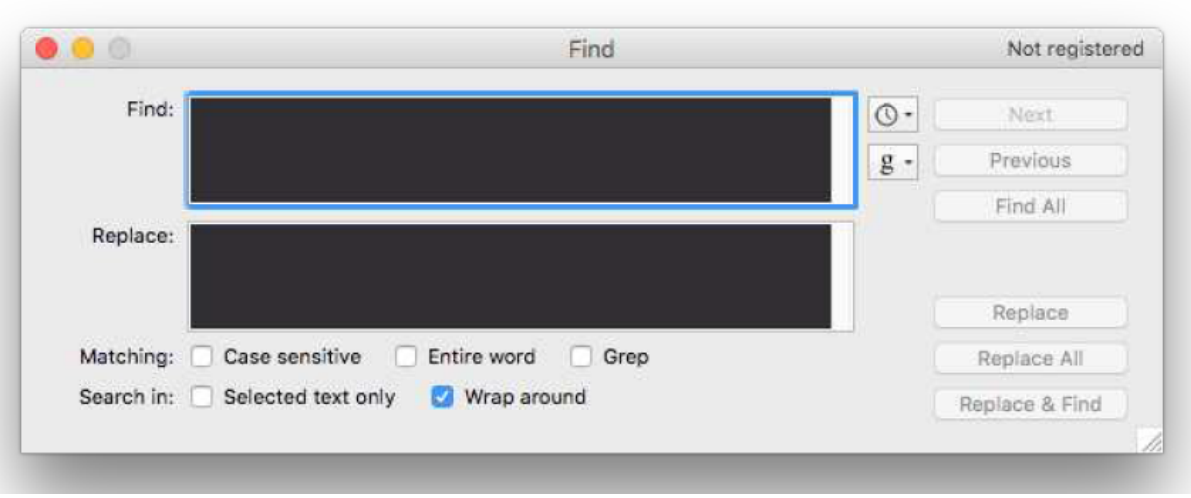
请勿将所有菜单操作都添加到工具栏上。

请勿在不同位置使用相同操作的不同名称或图标。

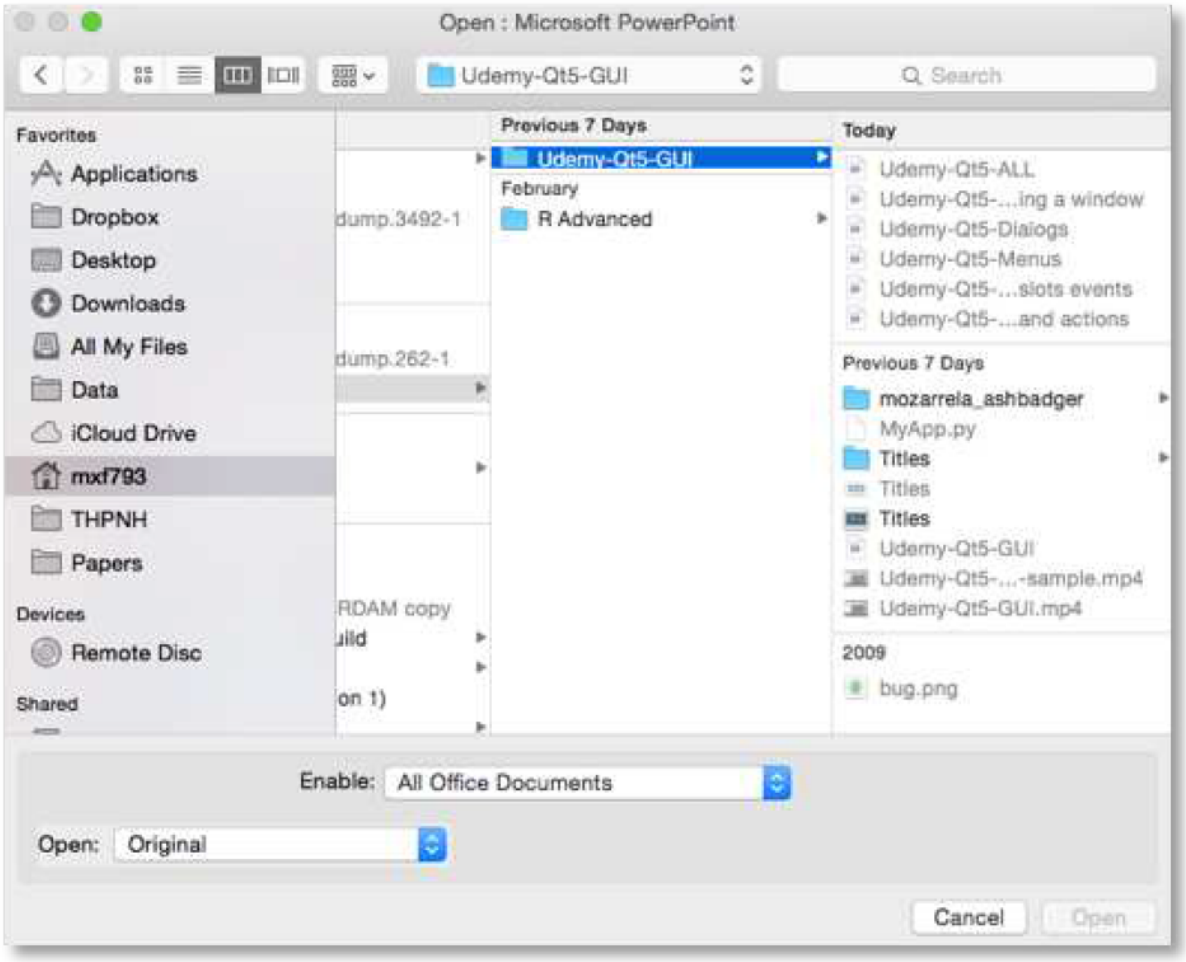
请勿从菜单中删除项目——而是禁用它们。

8. 对话框

对话框是图形用户界面中的有用组件，可让您与用户进行交流（因此得名“对话框”）。它们通常用于文件打开/保存、设置、首选项或应用程序主用户界面中无法容纳的功能。它们是小型模态（或阻塞）窗口，会始终显示在主应用程序窗口前，直到被关闭。Qt实际上为最常见的使用场景提供了多种“特殊”对话框，使您能够提供平台原生的用户体验，从而提升用户体验。



> 图四十九：标准图形用户界面功能——搜索对话框



图五十：标准图形用户界面功能——文件打开对话框

在 Qt 中，对话框由 `QDialog` 类处理。要创建一个新的对话框只需创建一个 `QDialog` 类型的对象，并将其父控件（例如 `QMainWindow`）作为其父对象传递给该对象即可。

让我们创建自己的 `QDialog`。首先，我们从一个简单的框架应用程序开始，该应用程序有一个按钮，该按钮与槽方法相连。

Listing 46. basic/dialogs_start.py

```
import sys

from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press me for a dialog!")
        button.clicked.connect(self.button_clicked)
        self.setCentralWidget(button)

    def button_clicked(self, s):
        print("click", s)


app = QApplication(sys.argv)
```

```
window = MainWindow()
window.show()

app.exec()
```

在槽 `button_clicked`（接收按钮按下的信号）中，我们创建对话框实例，并将我们的 `QMainWindow` 实例作为父窗口传递。这将使对话框成为 `QMainWindow` 的模态窗口。这意味着对话框将完全阻止与父窗口的交互。

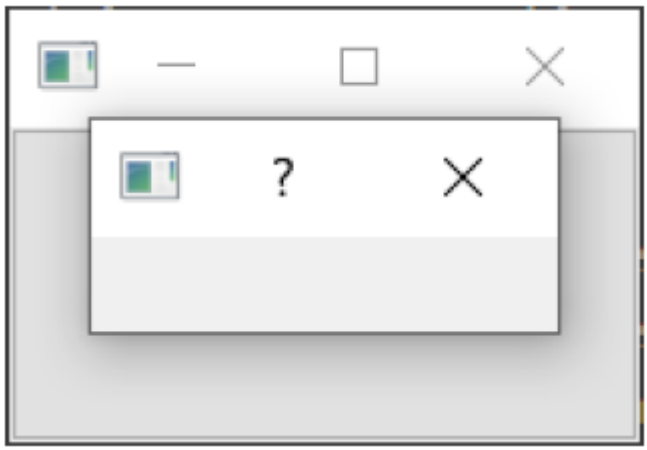
Listing 47. basic/dialogs_1.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QDialog,
    QMainWindow,
    QPushButton,
)

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press me for a dialog!")
        button.clicked.connect(self.button_clicked)
        self.setCentralWidget(button)

    def button_clicked(self, s):
        print("click", s)

        dlg = QDialog(self)
        dlg.setWindowTitle("?")
        dlg.exec()


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 点击按钮后，您将看到一个空的对话框弹出。

一旦创建了对话框，我们使用 `exec()` 函数启动它——就像我们之前使用 `QApplication` 创建应用程序的主事件循环一样。这并非巧合：当您执行 `QDialog` 时，会为对话框专门创建一个全新的事件循环。

> 一个事件循环统领一切
>
> 还记得我提到过，任何时候只能有一个 Qt 事件循环在运行吗？我是认真的！`QDialog` 会完全阻塞你的应用程序执行。不要在启动对话框的同时，还期望应用程序的其他部分继续运行。
>
> 我们稍后将探讨如何利用多线程技术来解决这一难题。



> 图五十一：我们的空对话框覆盖在窗口上。

就像我们的第一个窗口一样，这个窗口也不太有趣。让我们通过添加一个对话框标题和一组“确定”和“取消”按钮来解决这个问题，以便用户可以接受或拒绝该模态窗口。

要自定义 `QDialog`，我们可以继承它。

Listing 48. basic/dialogs_2a.py

```
class CustomDialog(QDialog):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("HELLO!")

        buttons = (
            QDialogButtonBox.StandardButton.Ok
            | QDialogButtonBox.StandardButton.Cancel
        )

        self.buttonBox = QDialogButtonBox(buttons)
        self.buttonBox.accepted.connect(self.accept)
        self.buttonBox.rejected.connect(self.reject)

        self.layout = QVBoxLayout()
        message = QLabel("Something happened, is that OK?")
        self.layout.addWidget(message)
        self.layout.addWidget(self.buttonBox)
        self.setLayout(self.layout)
```

在上述代码中，我们首先创建了 `QDialog` 的子类，并将其命名为 `CustomDialog` 。对于 `QMainWindow`，我们在类中的 `__init__` 块中应用自定义设置，以便在对象创建时应用这些自定义设置。首先，我们使用 `.setWindowTitle()` 为 `QDialog` 设置标题，与我们为主窗口设置标题的方式完全相同。

下一段代码涉及创建和显示对话框按钮。这可能比您预期的要复杂一些。然而，这是由于Qt在不同平台上处理对话框按钮位置时具有灵活性。

> 轻松解决?
>
> 当然，您可以选择忽略这一点，使用布局中的标准 `QButton`，但本文所述的方法可确保对话框遵循主机桌面标准（例如"确定"按钮位于左侧而非右侧）。随意更改这些行为可能会让用户感到极其烦躁，因此我不建议这样做。

创建对话框按钮框的第一步是使用 `QDialogButtonBox` 的命名空间属性定义要显示的按钮。可用的按钮完整列表如下：

Table 1. `QDialogButtonBox` available button types.

按键类型
`QDialogButtonBox.Ok`
`QDialogButtonBox.Open`
`QDialogButtonBox.Save`
`QDialogButtonBox.Cancel`
`QDialogButtonBox.Close`
`QDialogButtonBox.Discard`
`QDialogButtonBox.Apply`
`QDialogButtonBox.Reset`
`QDialogButtonBox.RestoreDefaults`
`QDialogButtonBox.Help`
`QDialogButtonBox.SaveAll`
`QDialogButtonBox.Yes`
`QDialogButtonBox.YesToAll`
`QDialogButtonBox.No`
`QDialogButtonBox.NoToAll`

按键类型
`QDialogButtonBox.Abort`
`QDialogButtonBox.Retry`
`QDialogButtonBox.Ignore`
`QDialogButtonBox.NoButton`

这些应该足以创建任何你能想到的对话框。您可以通过使用管道符 (|) 将多个按钮进行或运算来构建多按钮行。Qt会根据平台标准自动处理按钮的顺序。例如，要显示“确定”和“取消”按钮，我们使用了：

```
buttons = QDialogButtonBox.Ok | QDialogButtonBox.Cancel
```

变量按钮现在包含一个整数值，代表这两个按钮。接下来，我们必须创建一个 QDialogButtonBox 实例来容纳这些按钮。按钮的显示标志作为第一个参数传递。

为了使按钮产生效果，您必须将正确的 `QDialogButtonBox` 信号连接到对话框上的槽。在本例中，我们将 `QDialogButtonBox` 的 `.accepted` 和 `.rejected` 信号连接到 `QDialog` 子类的 `.accept()` 和 `.reject()` 处理程序。

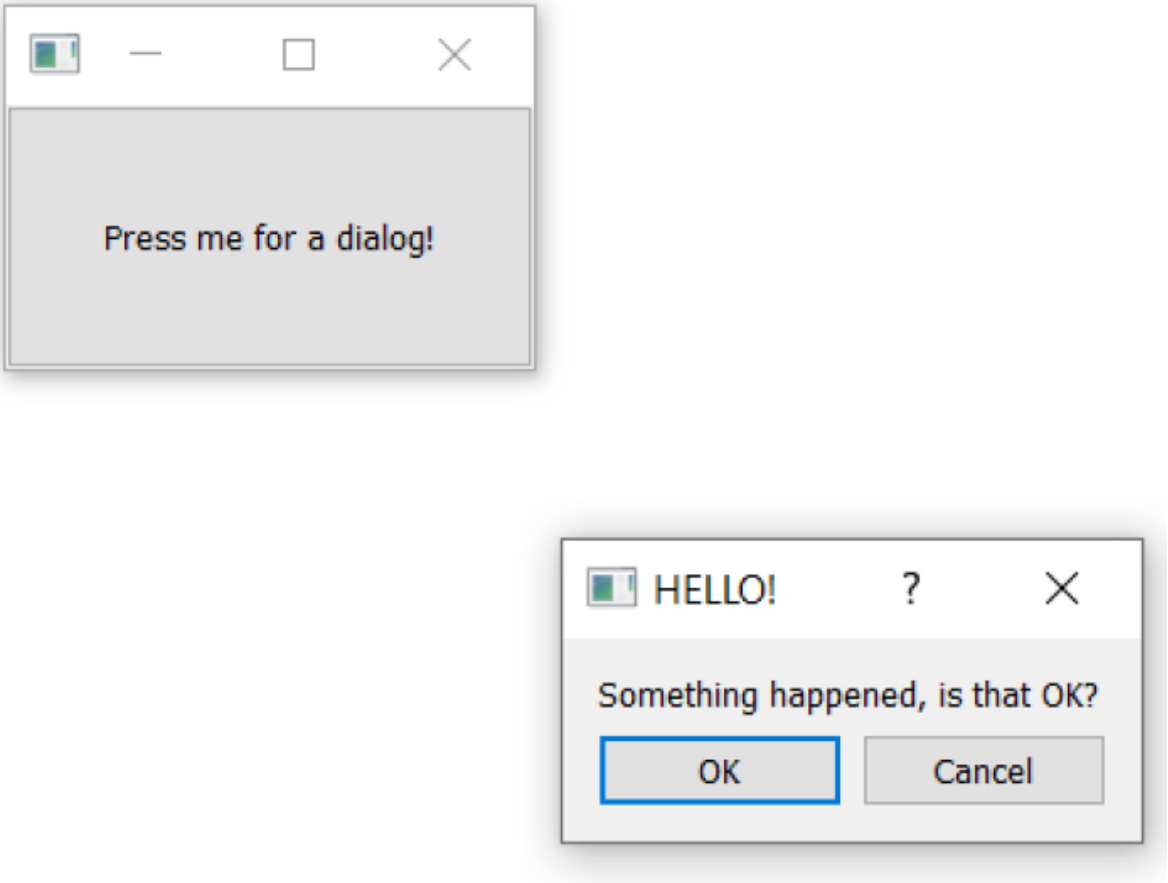
最后，为了使 `QDialogButtonBox` 出现在我们的对话框中，我们必须将其添加到对话框布局中。因此，对于主窗口，我们创建一个布局，并将我们的 `QDialogButtonBox` 添加到其中（`QDialogButtonBox` 是一个控件），然后将该布局设置到我们的对话框上。

最终，我们在 `MainWindow.button_clicked` 槽中启动 `CustomDialog`。

Listing 49. basic/dialogs_2a.py

```
    def button_clicked(self, s):
        print("click", s)

        dlg = CustomDialog()
        if dlg.exec():
            print("Success!")
        else:
            print("Cancel!")
```

> 🚀 **运行它吧！** 点击以启动对话框，您将看到一个包含按钮的对话框。



> 图五十二：我们与标签和按钮的对话框。

当您点击按钮以启动对话框时，可能会发现它出现在父窗口之外——通常位于屏幕中央。通常您希望对话框出现在其启动窗口之上，以便用户更容易找到。要实现这一点，我们需要为对话框指定一个父窗口。如果我们将主窗口作为**父窗口**传递给 Qt，Qt 会将新对话框的位置调整为对话框的中心与窗口的中心对齐。

我们可以修改我们的 `CustomDialog` 类，使其接受一个 `parent` 参数。

Listing 50. basic/dialogs_2b.py

```
class CustomDialog(QDialog):
    def __init__(self, parent=None): #1
        super().__init__(parent)

        self.setWindowTitle("HELLO!")

        buttons = (
            QDialogButtonBox.StandardButton.Ok
            | QDialogButtonBox.StandardButton.Cancel
        )

        self.buttonBox = QDialogButtonBox(buttons)
        self.buttonBox.accepted.connect(self.accept)
        self.buttonBox.rejected.connect(self.reject)

        self.layout = QVBoxLayout()
        message = QLabel("Something happened, is that OK?")
        self.layout.addWidget(message)
        self.layout.addWidget(self.buttonBox)
        self.setLayout(self.layout)
```

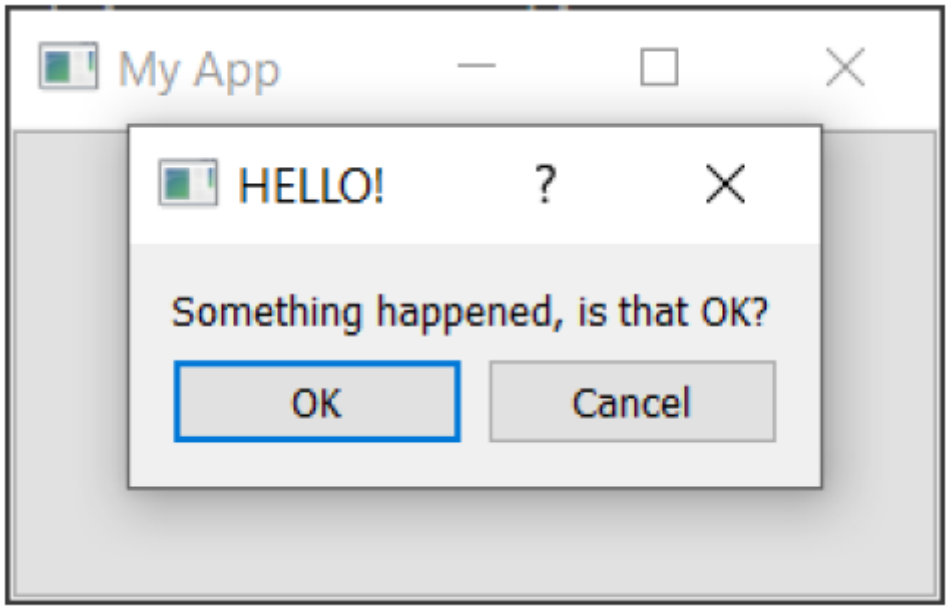
> 1. 我们设置默认值为 `None`，这样我们就可以省略父对象。

然后，当我们创建自定义对话框的实例时，可以将主窗口作为参数传递进去。在我们的 `button_clicked` 方法中，`self` 就是我们的主窗口对象。

Listing 51. basic/dialogs_2b.py

```
    def button_clicked(self, s):
        print("click", s)

        dlg = CustomDialog(self)
        if dlg.exec():
            print("Success!")
        else:
            print("Cancel!")
```

> 🚀 **运行它吧！** 点击以启动对话框，您应该会在父窗口正中央看到对话框弹出。



> 图五十三：我们的对话框位于父窗口的中央。

恭喜！您已成功创建了第一个对话框。当然，您可以继续向对话框中添加任何其他内容。只需像往常一样将其插入到布局中即可。

大多数应用程序都需要一些常见的对话框。虽然您可以自行构建这些对话框，但Qt也提供了许多内置对话框供你使用。这些对话框为您处理了大量工作，设计合理且符合平台标准。

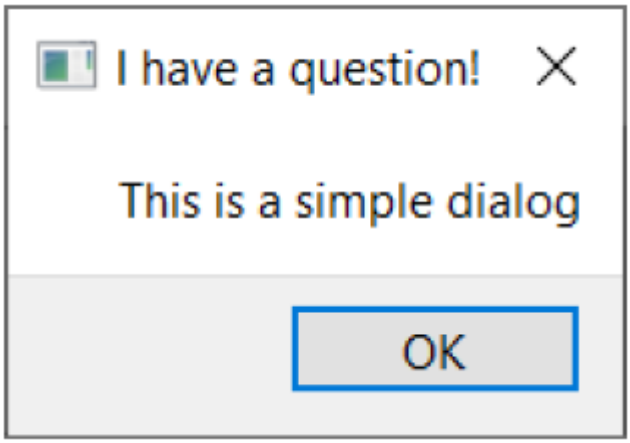
使用 `QMessageBox` 显示消息对话框

我们将首先介绍的内置对话框类型是 `QMessageBox`。它可用于创建信息、警告、关于或问题对话框——类似于我们手动创建的对话框。下面的示例创建了一个简单的 `QMessageBox` 并显示它。

Listing 52. basic/dialogs_3.py

```
    def button_clicked(self, s):
        dlg = QMessageBox(self)
        dlg.setWindowTitle("I have a question!")
        dlg.setText("This is a simple dialog")
        button = dlg.exec()

        # 查找按钮枚举项以获取结果。
        button = QMessageBox.StandardButton(button)

        if button == QMessageBox.StandardButton.Ok:
            print("OK!")
```

> 🚀 **运行它吧！** 您将看到一个带有"确定"按钮的简单对话框。



> 图五十四：一个 `QMessageBox` 对话框。

与我们之前讨论的对话框按钮框类似， `QMessageBox` 上显示的按钮也通过一组常量进行配置，这些常量可以使用 `|` 符号组合以显示多个按钮。可用按钮类型的完整列表如下所示：

Table 2. `QMessageBox` available button types.

按键类型
`QMessageBox.Ok`
`QMessageBox.Open`
`QMessageBox.Save`
`QMessageBox.Cancel`
`QMessageBox.Close`
`QMessageBox.Discard`
`QMessageBox.Apply`
`QMessageBox.Reset`
`QMessageBox.RestoreDefaults`
`QMessageBox.Help`
`QMessageBox.SaveAll`

按键类型
`QMessageBox.Yes`
`QMessageBox.YesToAll`
`QMessageBox.No`
`QMessageBox.NoToAll`
`QMessageBox.Abort`
`QMessageBox.Retry`
`QMessageBox.Ignore`
`QMessageBox.NoButton`

您还可以通过设置以下其中一个图标来调整对话框中显示的图标：

Table 3. QMessageBox icon constants.

图标状态	Description
`QMessageBox.NoIcon`	消息框没有图标
`QMessageBox.Question`	这条消息是在提问
`QMessageBox.Information`	该信息仅供参考
`QMessageBox.Warning`	该消息为警告信息
`QMessageBox.Critical`	该消息表明存在一个严重问题

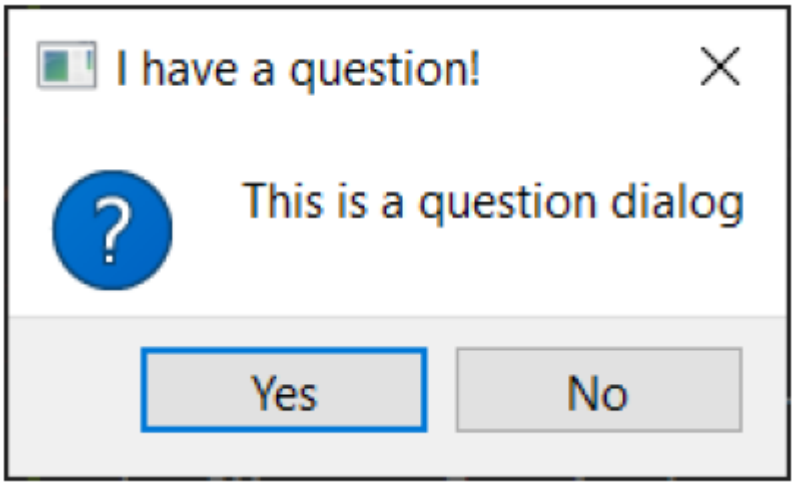
例如，以下代码创建一个带有“**是**”和“**否**”按钮的对话框。

Listing 53. basic/dialogs_4.py

```
from PyQt6.QtWidgets import (
    QApplication,
    QDialog,
    QMainWindow,
    QMessageBox,
    QPushButton,
)

class MainWindow(QMainWindow):
    # __init__ 方法已省略，以提高可读性
    def button_clicked(self, s):
        dlg = QMessageBox(self)
        dlg.setWindowTitle("I have a question!")
        dlg.setText("This is a question dialog")
        dlg.setStandardButtons(
            QMessageBox.StandardButton.Yes
            | QMessageBox.StandardButton.No
        )
```

```
        dlg.setIcon(QMessageBox.Icon.Question)
        button = dlg.exec()

        # 查找按钮枚举项以获取结果
        button = QMessageBox.StandardButton(button)

        if button == QMessageBox.StandardButton.Yes:
            print("Yes!")
        else:
            print("No!")
```

> 🚀 **运行它吧！** 您将看到一个带有“是”和“否”按钮的对话框。



> 图五十五：使用 `QMessageBox` 创建的对话框。

标准的 `QMessageBox` 对话框

为了进一步简化操作，`QMessageBox` 还提供了一系列静态方法，这些方法可用于直接显示此类消息对话框，而无需先创建 `QMessageBox` 实例。这些方法如下所示：

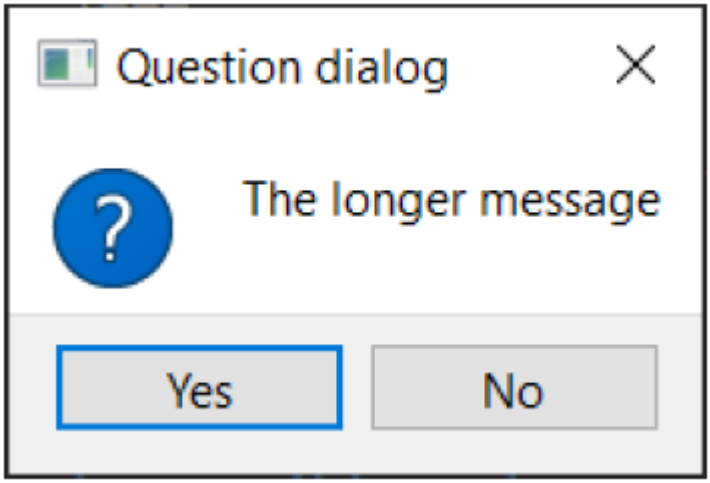
```
QMessageBox.about(parent, title, message)
QMessageBox.critical(parent, title, message)
QMessageBox.information(parent, title, message)
QMessageBox.question(parent, title, message)
QMessageBox.warning(parent, title, message)
```

`parent` 参数是对话框所属的父窗口。如果您是从主窗口启动对话框，可以使用 `self` 引用主窗口对象。以下示例创建一个问题对话框，与之前示例类似，包含“是”和“否”按钮。

Listing 54. basic/dialogs_5.py

```
    def button_clicked(self, s):
        button = QMessageBox.question(
            self, "Question dialog", "The longer message"
        )

        if button == QMessageBox.StandardButton.Yes:
            print("Yes!")
        else:
            print("No!")
```

> 🚀 **运行它吧！** 您会看到相同的结果，这次使用的是内置的 `.question()` 方法。



> 图五十六：内置的提问对话框

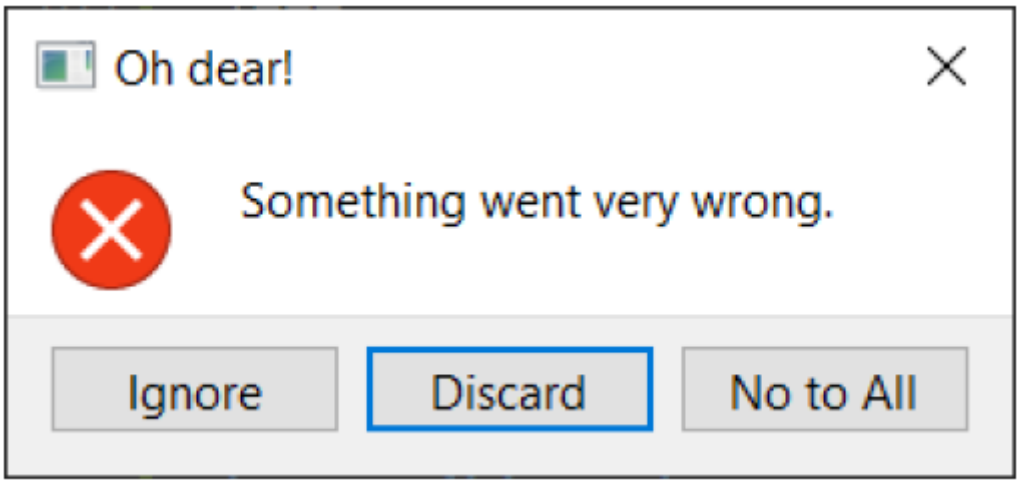
请注意，我们现在不再调用 `exec()` 函数，而是直接调用对话框方法，对话框便会被创建。每个方法的返回值都是被按下的按钮。我们可以通过将返回值与标准按钮常量进行比较，来检测被按下的按钮。

四个信息、问题、警告和关键方法也支持可选的按钮和默认按钮参数，这些参数可用于调整对话框中显示的按钮并默认选择其中一个。不过通常情况下，您可能并不希望更改默认设置。

Listing 55. basic/dialogs_6.py

```
    def button_clicked(self, s):
        button = QMessageBox.critical(
            self,
            "Oh dear!",
            "Something went very wrong.",
            buttons=QMessageBox.StandardButton.Discard
            | QMessageBox.StandardButton.NoToAll
            | QMessageBox.StandardButton.Ignore,
            defaultButton=QMessageBox.StandardButton.Discard,
        )

        if button == QMessageBox.StandardButton.Discard:
            print("Discard!")
        elif button == QMessageBox.StandardButton.NoToAll:
            print("No to all!")
        else:
            print("Ignore!")
```

> 🚀 **运行它吧！** 您将看到一个带有自定义按钮的确认对话框



> 图五十七：严重错误！这是一个糟糕的对话框。

请求单个值

有时您会需要从用户获取单个参数，并希望能够显示一个简单的输入对话框来获取该参数。对于此用例，`PyQt6` 提供了 `QInputDialog` 类。该类可用于获取不同类型的数据，同时还可以对用户输入的值设置限制。

静态方法都接受一个父控件的父参数（通常为 `self`）、一个对话框窗口标题的标题参数以及一个显示在输入框旁边的标签，以及其他类型特定的控件。调用这些方法时，它们会显示一个对话框，关闭后返回一个值和 `ok` 的元组，告知您是否按下了“确定”按钮。如果 `ok` 为 `False`，则对话框已关闭。

首先，我们来看一个最简单的例子——一个按钮，它会弹出一个对话框，从用户那里获取一个整数值。它使用了 `QDialog.get_int()` 静态方法，传递了父级 `self`、窗口标题和输入控件旁边显示的提示信息。

Listing 56. basic/dialogs_input_1.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QInputDialog,
    QMainWindow,
    QPushButton,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button1 = QPushButton("Integer")
        button1.clicked.connect(self.get_an_int)

        self.setCentralWidget(button1)

    def get_an_int(self):
        my_int_value, ok = QInputDialog.getInt(
```

```
            self, "Get an integer", "Enter a number"
        )
        print("Result:", ok, my_int_value)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到一个按钮。按下它后，系统会提示您输入一个数字。

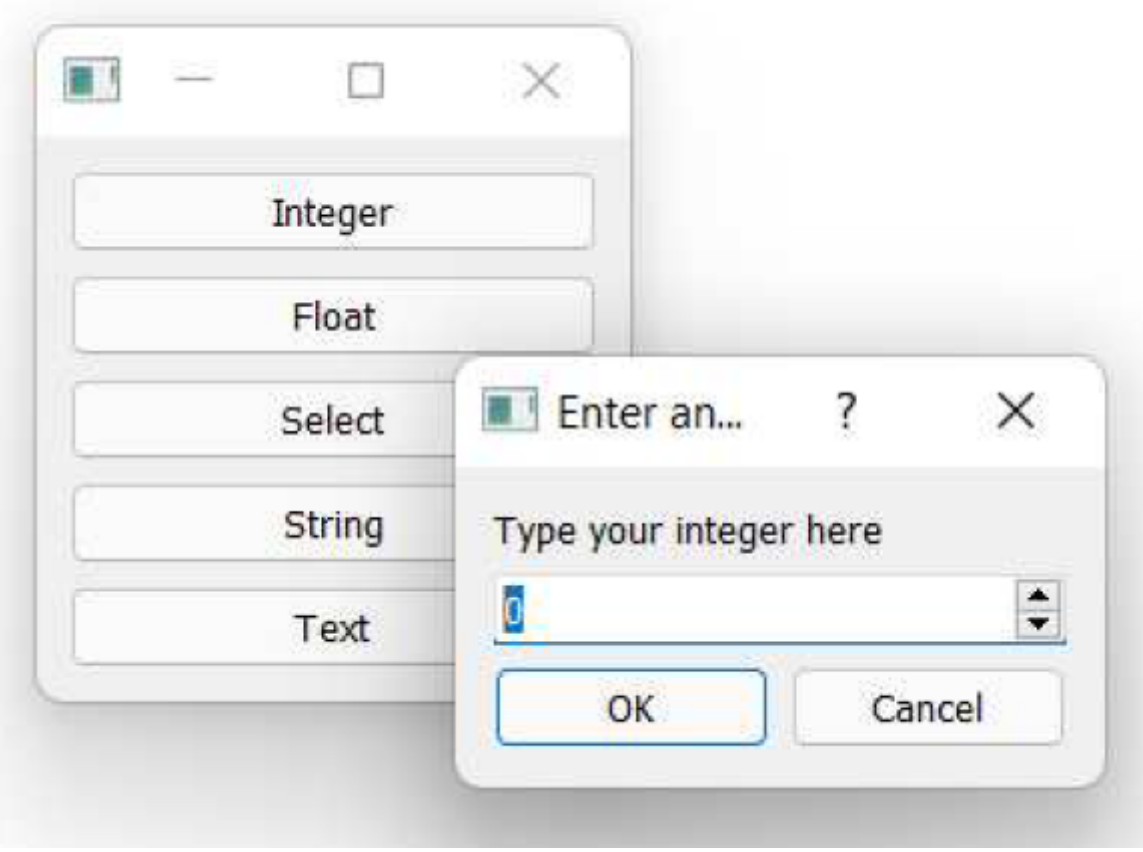
到目前为止，一切都很令人兴奋。让我们扩展这个例子，添加一些按钮，以及它们的处理方法。我们将先将按钮的信号连接到方法槽，然后逐步实现每个输入方法。

Listing 57. basic/dialogs_input_2.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QInputDialog,
    QLineEdit,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()

        button1 = QPushButton("Integer")
        button1.clicked.connect(self.get_an_int)
        layout.addWidget(button1)

        button2 = QPushButton("Float")
        button2.clicked.connect(self.get_a_float)
        layout.addWidget(button2)

        button3 = QPushButton("Select")
        button3.clicked.connect(self.get_a_str_from_a_list)
        layout.addWidget(button3)

        button4 = QPushButton("String")
        button4.clicked.connect(self.get_a_str)
        layout.addWidget(button4)
```

```
        button5 = QPushButton("Text")
        button5.clicked.connect(self.get_text)
        layout.addWidget(button5)

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

    def get_an_int(self):
        my_int_value, ok = QInputDialog.getInt(
            self, "Get an integer", "Enter a number"
        )
        print("Result:", ok, my_int_value)

    def get_a_float(self):
        pass

    def get_a_str_from_a_list(self):
        pass

    def get_a_str(self):
        pass

    def get_text(self):
        pass


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到一组按钮，可用于启动输入功能，但目前仅支持整数输入。



图五十八：对话框启动器演示。点击按钮以启动对话框并输入值。

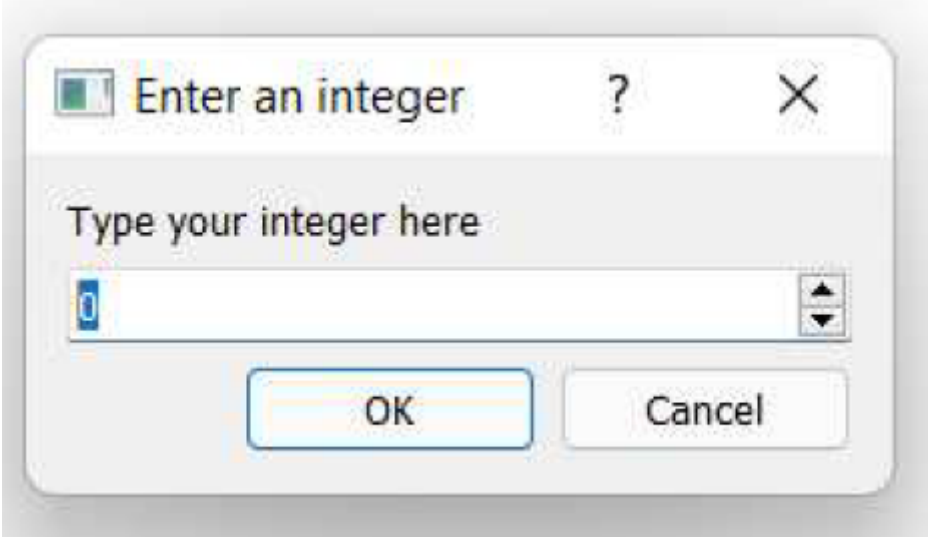
按下按钮将调用我们定义的输入方法之一，让我们接下来实现它们。我们将依次遍历每个 `QInputDialog` 方法，查看可用的配置选项并将其添加到示例中。

整数

如前所述，要从用户获取整数值，可以使用 `QInputDialog.getInt()` 方法。该方法会在对话框中显示一个标准的 Qt `QDoubleSpinBox` 控件。您可以指定初始值、输入的最小值和最大值范围，以及使用箭头控件时的步长。

Listing 58. basic/dialogs_input_3.py

```
    def get_an_int(self):
        title = "Enter an integer"
        label = "Type your integer here"
        my_int_value, ok = QInputDialog.getInt(
            self, title, label, value=0, min=-5, max=5, step=1
        )
        print("Result:", ok, my_int_value)
```



> 图五十九：整数输入的对话框

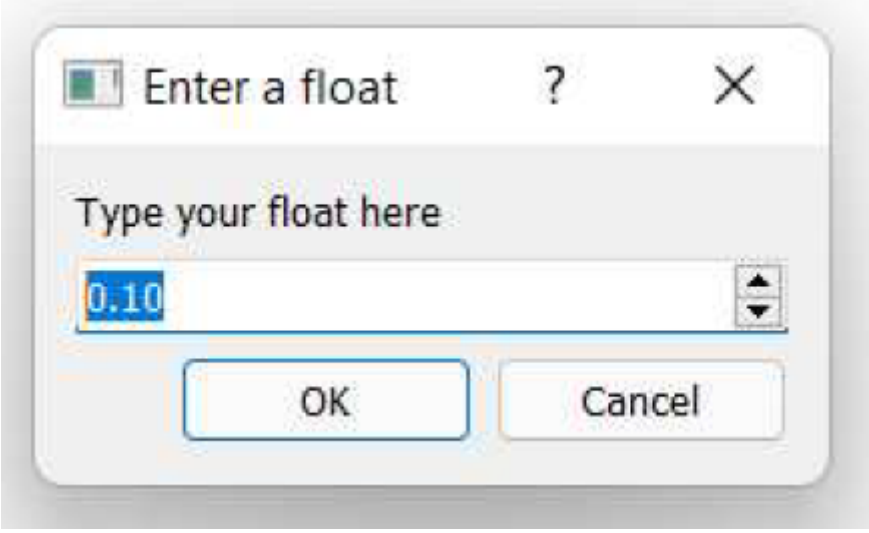
> 即使用户点击“取消”按钮退出对话框，输入的值仍会被返回。在使用该值之前，您应始终先检查 `ok` 返回参数的值。

浮点数

对于浮点数类型，您可以使用 `QInputDialog.getDouble()` 方法—— Python中的 `float` 类型对应着 C++ 中的 `double` 类型。这与上文的 `getInt` 输入完全相同，只是增加了 `decimals` 参数来控制显示的小数位数。

Listing 59. basic/dialogs_input_3.py

```
    def get_a_float(self):
        title = "Enter a float"
        label = "Type your float here"
        my_float_value, ok = QInputDialog.getDouble(
            self,
            title,
            label,
            value=0,
            min=-5.3,
            max=5.7,
            decimals=2,
        )
        print("Result:", ok, my_float_value)
```



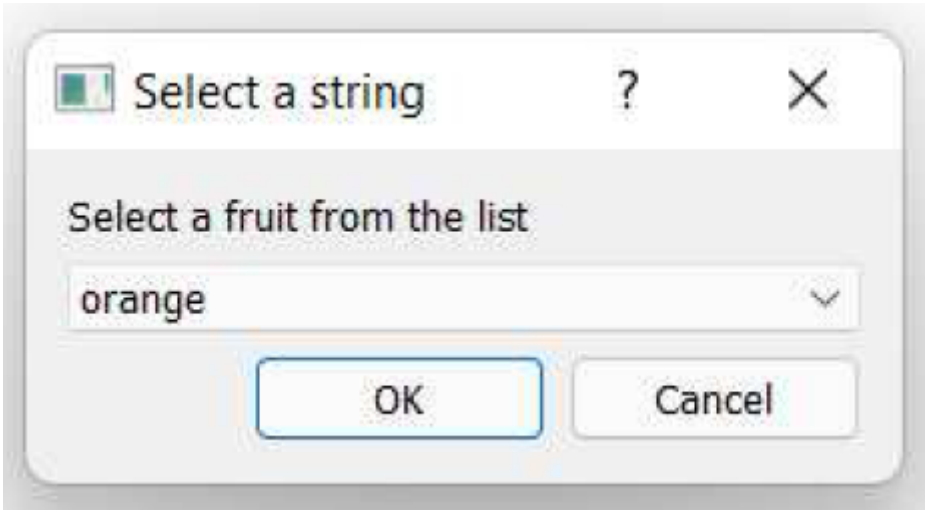
> 图六十：浮点数输入的对话框

从字符串列表中选择

要从字符串列表中选择一个项，可以使用 `QInputDialog.getItem()` 方法。要选择的字符串列表通过 `items` 参数提供。您可以通过将 `current` 参数设置为所选项的索引，来指定初始选中的项。默认情况下，该列表是可编辑的，即用户可以根据需要向列表中添加新项。您可以通过传递 `editable=False` 来禁用此行为。

Listing 60. basic/dialogs_input_3.py

```
    def get_a_str_from_a_list(self):
        title = "Select a string"
        label = "Select a fruit from the list"
        items = ["apple", "pear", "orange", "grape"]
        initial_selection = 2  # orange，从 0 开始索引
        my_selected_str, ok = QInputDialog.getItem(
            self,
            title,
            label,
            items,
            current=initial_selection,
            editable=False,
        )
        print("Result:", ok, my_selected_str)
```



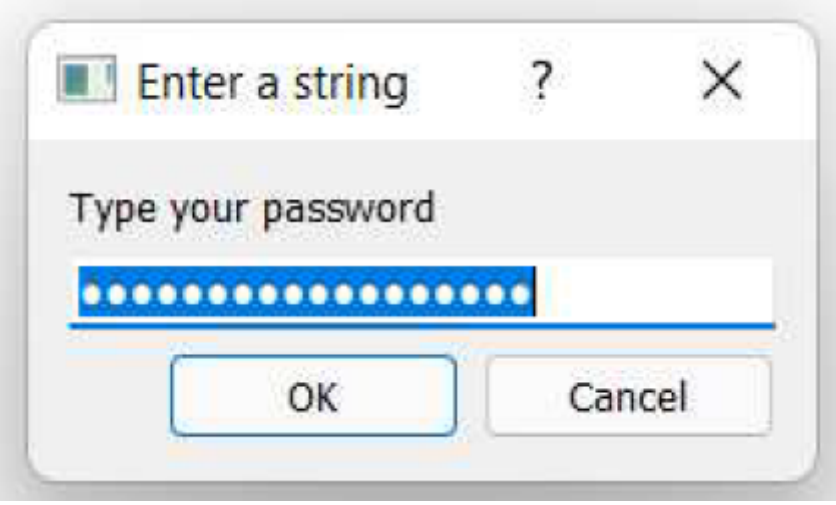
> 图六十一：从字符串列表中选择输入的对话框

单行文本

要从用户获取一行文本，您可以使用 `QInputDialog.getText` 。您可以通过将文本作为参数传递来提供输入的初始内容。模式参数允许您在正常模式和密码模式之间切换，其中输入的文本以星号显示，分别传递 `QLineEdit.EchoMode.Normal` 或 `QLineEdit.EchoMode.Password`。

Listing 61. basic/dialogs_input_3.py

```
    def get_a_str(self):
        title = "Enter a string"
        label = "Type your password"
        text = "my secret password"
        mode = QLineEdit.EchoMode.Password
        my_selected_str, ok = QInputDialog.getText(
            self, title, label, mode, text
        )
        print("Result:", ok, my_selected_str)
```



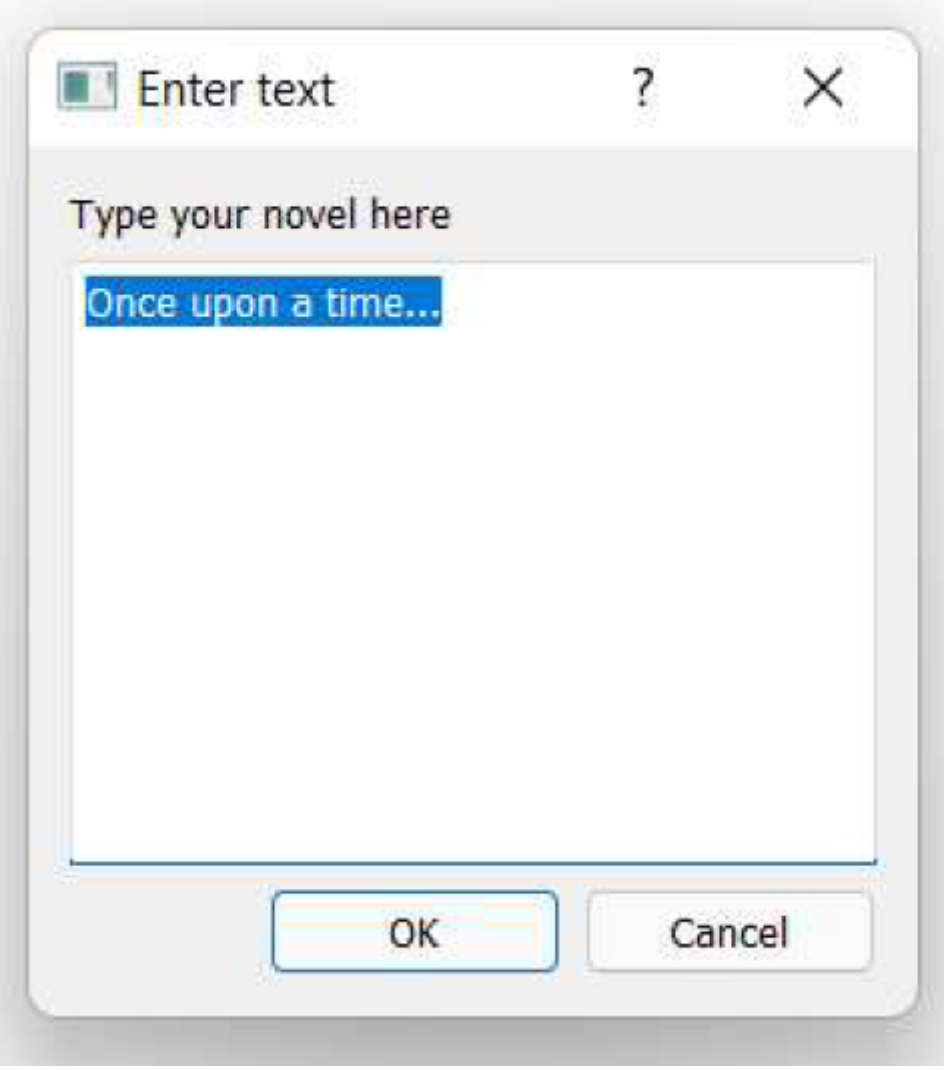
> 图六十二：单行文本输入对话框，密码模式。

多行文本

最后，要输入多行文本，您可以使用 `QLineEdit.getMultiLineText()` 方法。该方法仅接受文本的初始状态。

Listing 62. basic/dialogs_input_3.py

```
    def get_text(self):
        title = "Enter text"
        label = "Type your novel here"
        text = "Once upon a time..."
        my_selected_str, ok = QInputDialog.getMultiLineText(
            self, title, label, text
        )
        print("Result:", ok, my_selected_str)
```



> 图六十三：多行文本输入的对话框

> 🚀 **运行它吧！** 在所有输入方法都已实现后，您现在可以点击每个按钮，查看不同的输入对话框出现。

使用 QInputDialog 实例

上述静态方法适用于大多数使用场景。然而，如果您希望对 `QInputDialog` 的行为进行更精细的控制，您可以创建一个 `QInputDialog` 的实例并在显示前对其进行配置——就像其他对话框类一样。以下是相同的示例，但采用这种方法。

Listing 63. basic/dialogs_input_instance.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QInputDialog,
    QLineEdit,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class MainWindow(QMainWindow):
```

```
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()

        button1 = QPushButton("Integer")
        button1.clicked.connect(self.get_an_int)
        layout.addWidget(button1)

        button2 = QPushButton("Float")
        button2.clicked.connect(self.get_a_float)
        layout.addWidget(button2)

        button3 = QPushButton("Select")
        button3.clicked.connect(self.get_a_str_from_a_list)
        layout.addWidget(button3)

        button4 = QPushButton("String")
        button4.clicked.connect(self.get_a_str)
        layout.addWidget(button4)

        button5 = QPushButton("Text")
        button5.clicked.connect(self.get_text)
        layout.addWidget(button5)

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

    def get_an_int(self):
        dialog = QInputDialog(self)
        dialog.setWindowTitle("Enter an integer")
        dialog.setLabelText("Type your integer here")
        dialog.setIntValue(0)
        dialog.setIntMinimum(-5)
        dialog.setIntMaximum(5)
        dialog.setIntStep(1)

        ok = dialog.exec()
        print("Result:", ok, dialog.intValue())

    def get_a_float(self):
        dialog = QInputDialog(self)
        dialog.setWindowTitle("Enter a float")
        dialog.setLabelText("Type your float here")
        dialog.setDoubleValue(0.1)
        dialog.setDoubleMinimum(-5.3)
        dialog.setDoubleMaximum(5.7)
        dialog.setDoubleStep(1.4)
        dialog.setDoubleDecimals(2)

        ok = dialog.exec()
        print("Result:", ok, dialog.doubleValue())
```

```
    def get_a_str_from_a_list(self):
        dialog = QInputDialog(self)
        dialog.setWindowTitle("Select a string")
        dialog.setLabelText("Select a fruit from the list")
        dialog.setComboBoxItems(["apple", "pear", "orange", "grape"])
        dialog.setComboBoxEditable(False)
        dialog.setTextValue("orange")

        ok = dialog.exec()
        print("Result:", ok, dialog.textValue())

    def get_a_str(self):
        dialog = QInputDialog(self)
        dialog.setWindowTitle("Enter a string")
        dialog.setLabelText("Type your password")
        dialog.setTextValue("my secret password")
        dialog.setTextEchoMode(QLineEdit.EchoMode.Password)

        ok = dialog.exec()
        print("Result:", ok, dialog.textValue())

    def get_text(self):
        dialog = QInputDialog(self)
        dialog.setWindowTitle("Enter text")
        dialog.setLabelText("Type your novel here")
        dialog.setTextValue("Once upon a time...")
        dialog.setOption(
            QInputDialog.InputDialogOption.UsePlainTextEditForTextInput,
            True,
        )

        ok = dialog.exec()
        print("Result:", ok, dialog.textValue())


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 它应该像以前一样工作——请随意调整参数来调整它的行为！

有几点需要注意。首先，当您调用 `exec()` 时，返回值等同于之前返回的 `ok` 值（1 表示 `True`，0 表示 `False`）。要获取实际输入的值，您需要使用对话框对象的类型特定方法，例如 `.doubleValue()`。其次，对于 `QComboBox` 从字符串列表中选择项时，您使用与行输入或文本输入相同的 `.setTextValue()`（设置）和 `.textValue()`（获取）方法。

文件对话框

应用程序中对话框最常见的用途之一是处理文件——无论是应用程序生成的文档，还是希望在应用程序使用之间保留的配置设置。幸运的是，PyQt6内置了用于打开文件、选择文件夹和保存文件的对话框。

如前所述，如果您使用 Qt 的内置对话框工具，您的应用程序将遵循平台标准。在文件对话框的情况下，PyQt6 更进一步，将使用平台的内置对话框进行这些操作，可以确保您的应用程序对用户来说是熟悉的。

> 创建良好的文件对话框非常困难，因此我不建议您尝试自行开发。

在 PyQt6 中，文件对话框是通过 `QFileDialog` 类创建的。为了方便起见，它提供了一系列静态方法，您可以调用这些方法来显示特定的对话框，而无需进行过多配置。以下是一个使用 `QFileDialog.getOpenFileName()` 静态方法获取要打开的文件名的示例。

Listing 64. basic/dialogs_file_1.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QFileDialog,
    QMainWindow,
    QPushButton,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button1 = QPushButton("Open file")
        button1.clicked.connect(self.get_filename)

        self.setCentralWidget(button1)

    def get_filename(self):
        filename, selected_filter = QFileDialog.getOpenFileName(self)
        print("Result:", filename, selected_filter)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 点击按钮以打开文件选择对话框。选择一个文件并点击 **[确定]** 或 **[取消]** 按钮以查看返回的结果。

如您所见，`QFileDialog.getOpenFilename()` 方法会返回两个值。第一个值是所选文件的名称（如果对话框被取消，则为空字符串）。第二个值是当前活动的文件过滤器——用于过滤对话框中可见的文件。默认情况下，该过滤器为“所有文件 (*)”，所有文件均可见。

基于文件的对话框（打开和保存）都接受一个过滤器参数，该参数是一个用分号分隔的过滤器定义字符串列表——这有点奇怪！还有一个 `initialFilter`，它是对话框首次打开时活动的过滤器字符串。让我们看看这些过滤器是如何定义的，以及如何最好地使用它们。

文件过滤器

Qt 文件过滤器的标准格式是一个字符串，其格式如下：用户友好名称可以是任意文本，而 `*.ext` 则是文件匹配过滤器和文件扩展名。该扩展名应在过滤器字符串末尾用括号括起。

```
"User-friendly name (*.ext)"
```

如果您想提供多个过滤器，可以使用 `;;` (两个分号) 将它们分隔开。以下是一个示例，其中包括一个“*所有文件”过滤器。

```
"Portable Network Graphics Image (*.png);;Comma Separated files(*.csv);;All files (*)"
```

接下来我们会更新示例，以将上述示例过滤器提供给 `QFileDialog.getOpenFilename()` 方法。

Listing 65. basic/dialogs_file_2.py

```
    def get_filename(self):
        filters = "Portable Network Graphics files (*.png);;CommaSeparated Values (*.csv);;All files (*)"
        print("Filters are:", filters)
        filename, selected_filter = QFileDialog.getOpenFileName(
            self,
            filter=filters,
        )
        print("Result:", filename, selected_filter)
```

> 您通常会看到 `*.*` 用于所有文件过滤器，但在 Qt 中这不会匹配没有扩展名的文件。

您可以将过滤器写入字符串中，但这样做可能会变得有些繁琐。如果您希望在初始状态下选择特定的过滤器，则需要复制该字符串中的文本（或从中提取）。相反，我建议您将文件过滤器定义存储为字符串列表，然后在传递给对话框方法之前使用 `;;` 将列表连接起来。这样做的好处是，初始过滤器可以通过索引从该列表中选择。

```
FILE_FILTERS = [
    "Portable Network Graphics files (*.png)",
    "Text files (*.txt)",
    "Comma Separated Values (*.csv)",
    "All files (*.*)",
]

initial_filter = FILE_FILTERS[2] # *.csv
# 构建以 ;; 分隔的过滤字符串
filters = ';;'.join(FILE_FILTERS)
```

我们的示例已更新为使用此方法，其中 `FILE_FILTERS` 在文件顶部定义，以便所有文件方法均可使用。

Listing 66. basic/dialogs_file_2b.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QFileDialog,
    QMainWindow,
    QPushButton,
)

FILE_FILTERS = [
    "Portable Network Graphics files (*.png)",
    "Text files (*.txt)",
    "Comma Separated Values (*.csv)",
    "All files (*.*)",
]


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button1 = QPushButton("Open file")
        button1.clicked.connect(self.get_filename)

        self.setCentralWidget(button1)

    def get_filename(self):
        initial_filter = FILE_FILTERS[3] # 从列表中选择一个
        filters = ";;".join(FILE_FILTERS)
        print("Filters are:", filters)
        print("Initial filter:", initial_filter)

        filename, selected_filter = QFileDialog.getOpenFileName(
            self,
            filter=filters,
            initialFilter=initial_filter,
        )
        print("Result:", filename, selected_filter)
```

```
app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

配置文件对话框

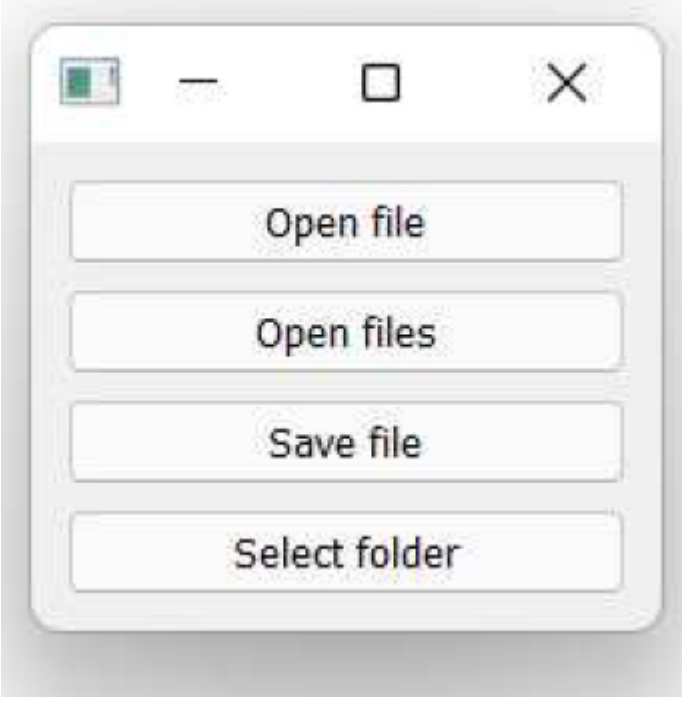
现在我们已经了解了过滤器，让我们扩展我们的示例，为更多类型的文件操作添加处理程序。然后，我们将逐步介绍每个 `QFileDialog` 方法，以了解其他可用的配置选项。下面，我们将会添加一系列按钮，并将它们连接到文件方法槽，以处理显示不同的对话框。

Listing 67. basic/dialogs_file_3.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QFileDialog,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

FILE_FILTERS = [
    "Portable Network Graphics files (*.png)",
    "Text files (*.txt)",
    "Comma Separated Values (*.csv)",
    "All files (*.*)",
]


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()

        button1 = QPushButton("Open file")
        button1.clicked.connect(self.get_filename)
        layout.addWidget(button1)

        button2 = QPushButton("Open files")
        button2.clicked.connect(self.get_filenames)
        layout.addWidget(button2)

        button3 = QPushButton("Save file")
        button3.clicked.connect(self.get_save_filename)
        layout.addWidget(button3)
```

```
        button4 = QPushButton("Select folder")
        button4.clicked.connect(self.get_folder)
        layout.addWidget(button4)

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

    def get_filename(self):
        initial_filter = FILE_FILTERS[3] # 从列表中选择一个
        filters = ";;".join(FILE_FILTERS)
        print("Filters are:", filters)
        print("Initial filter:", initial_filter)

        filename, selected_filter = QFileDialog.getOpenFileName(
            self,
            filter=filters,
            initialFilter=initial_filter,
        )
        print("Result:", filename, selected_filter)

    def get_filenames(self):
        pass

    def get_save_filename(self):
        pass

    def get_folder(self):
        pass

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到一组可用于运行文件方法的按钮——目前仅“打开文件”功能可用



> 图六十四：文件对话框启动器演示。

让我们依次逐一查看每个文件方法，并将它们添加到我们的示例中。

打开一个文件

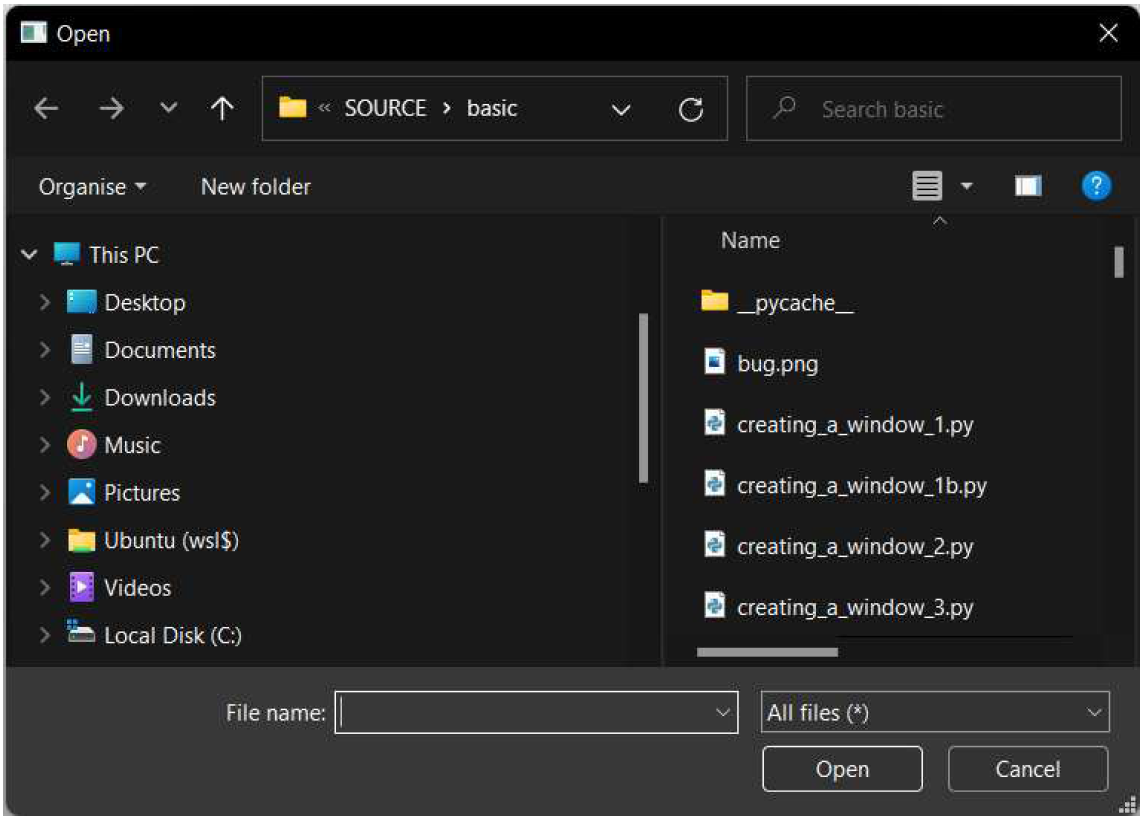
要选择单个文件名以打开文件，可以使用 `QFileDialog.getOpenFileName()` 方法。

静态方法都接受一个父控件的父参数（通常为 `self` ）和一个对话框标题的标题参数。它们还接受一个目录参数，该参数是对话框将打开的初始目录。标题和目录都可以是空字符串，在这种情况下，将使用默认标题，对话框将在当前文件夹中打开。

除了 `caption` 和 'directory' 外，该方法还接受 `filter` 和 `initialFilter` 参数来配置文件过滤器。完成后，它返回所选文件作为字符串（包含完整路径）以及当前选定的过滤器。

Listing 68. basic/dialogs_file_4.py

```
    def get_filename(self):
        caption = "" # 空值使用默认标题。
        initial_dir = "" # 空文件夹使用当前文件夹。
        initial_filter = FILE_FILTERS[3] # 从列表中选择一个。
        filters = ";;".join(FILE_FILTERS)
        print("Filters are:", filters)
        print("Initial filter:", initial_filter)

        filename, selected_filter = QFileDialog.getOpenFileName(
            self,
            caption=caption,
            directory=initial_dir,
            filter=filters,
            initialFilter=initial_filter,
        )
        print("Result:", filename, selected_filter)
```



> 图六十五：标准的Windows打开对话框，处于深色模式。

一旦获得文件名，即可使用标准 Python 进行加载。如果对话框已关闭，文件名变量将为空字符串。

Listing 69. basic/dialogs_file_4b.py

```
        if filename:
            with open(filename, "r") as f:
                file_contents = f.read()
```

打开多个文件

有时您希望用户能够一次加载多个文件——例如将一组数据文件加载到应用程序中。`QFileDialog.getOpenFileNames()` 方法可实现此功能。该方法与上述单文件方法使用相同的参数，唯一区别在于它返回所选文件路径的字符串列表。

Listing 70. basic/dialogs_file_4.py

```
    def get_filenames(self):
        caption = "" # 空值使用默认标题。
        initial_dir = "" # 空文件夹使用当前文件夹。
        initial_filter = FILE_FILTERS[1] # 从列表中选择一个。
        filters = ";;".join(FILE_FILTERS)
        print("Filters are:", filters)
        print("Initial filter:", initial_filter)

        filename, selected_filter = QFileDialog.getOpenFileNames(
            self,
            caption=caption,
            directory=initial_dir,
            filter=filters,
            initialFilter=initial_filter,
```

```
        )
        print("Result:", filenames, selected_filter)
```

您可以通过遍历并加载文件名中的文件，就像在前一个示例中一样。选择单个文件仍然可行，并将返回一个包含单个条目的列表。如果在未选择文件的情况下关闭对话框，文件名将是一个空列表。

Listing 71. basic/dialogs_file_4b.py

```
        for filename in filenames:
            with open(filename, "r") as f:
                file_contents = f.read()
```

保存一个文件

要保存文件，您可以使用 `QFileDialog.getSaveFileName()` 方法。

Listing 72. basic/dialogs_file_4.py

```
    def get_save_filename(self):
        caption = "" # 空值使用默认标题。
        initial_dir = "" # 空文件夹使用当前文件夹。
        initial_filter = FILE_FILTERS[2] # 从列表中选择一个。
        filters = ";;".join(FILE_FILTERS)
        print("Filters are:", filters)
        print("Initial filter:", initial_filter)

        filename, selected_filter = QFileDialog.getSaveFileName(
            self,
            caption=caption,
            directory=initial_dir,
            filter=filters,
            initialFilter=initial_filter,
        )
        print("Result:", filename, selected_filter)
```

同样，您可以使用文件名变量通过标准的 Python 保存到文件。如果对话框在未选择文件的情况下关闭，文件名变量将为空字符串。如果文件已存在，它将被覆盖且现有内容会丢失。

您应始终确认用户是否确实希望覆盖文件。在下面的示例中，我们使用 `os.path.exists()` 函数检查文件是否存在，然后显示一个 `QMessageBox` 对话框，询问用户是否继续覆盖现有文件。如果用户回答“否”，则不会写入文件。如果文件不存在，或用户回答“是”，则写入文件。

Listing 73. basic/dialogs_file_4b.py

```
import os

        if filename:
            if os.path.exists(filename):
                # 已存在文件，请用户确认。
                write_confirmed = QMessageBox.question(
                    self,
                    "Overwrite file?",
                    f"The file {filename} exists. Are you sure youwant to
overwrite it?",
                )
```

```
            else:
                # 文件不存在，始终确认
                write_confirmed = True

            if write_confirmed:
                with open(filename, "w") as f:
                    file_content = "YOUR FILE CONTENT"
                    f.write(file_content)
```

> 始终尝试考虑用户可能犯的错误——例如在保存对话框中点击了错误的文件——并给他们机会自行挽救。

选择一个文件夹

要选择一个现有文件夹，您可以使用 `QFileDialog.getExistingDirectory()` 方法。

```
folder_path = QFileDialog.getExistingDirectory(parent, caption="", directory="",
options=ShowDirsOnly)
```

默认情况下，`QFileDialog.getExistingDirectory` 只会显示文件夹。您可以通过传入参数来更改此设置。

> 还有一些静态方法可用于加载远程文件，这些方法返回 `QUrl` 对象。这些方法包括 `QFileDialog.getSaveFileUrl()`、`QFileDialog.getOpenFileUrls()`、`QFileDialog.getOpenFileUrl()`，以及用于文件夹的 `QFileDialog.getExistingDirectoryUrl()`。有关详细信息，请参阅 Qt文档。

如果您希望对文件对话框的行为有更多控制权，可以创建一个 `QFileDialog` 实例并使用配置方法。以下是相同的文件对话框演示，但与使用上述静态方法不同，我们创建了一个 `QFileDialog` 实例并在启动对话框前对其进行配置。

Listing 74. basic/dialogs_file_2.py

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QFileDialog,
    QLineEdit,
```

```
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

FILE_FILTERS = [
    "Portable Network Graphics files (*.png)",
    "Text files (*.txt)",
    "Comma Separated Values (*.csv)",
    "All files (*.*)",
]


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        layout = QVBoxLayout()

        button1 = QPushButton("Open file")
        button1.clicked.connect(self.get_filename)
        layout.addWidget(button1)

        button2 = QPushButton("Open files")
        button2.clicked.connect(self.get_filenames)
        layout.addWidget(button2)

        button3 = QPushButton("Save file")
        button3.clicked.connect(self.get_save_filename)
        layout.addWidget(button3)

        button4 = QPushButton("Select folder")
        button4.clicked.connect(self.get_folder)
        layout.addWidget(button4)

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

    def get_filename(self):
        caption = "Open file"
        initial_dir = "" # 空文件夹使用当前文件夹。
        initial_filter = FILE_FILTERS[3] # 从列表中选择一个

        dialog = QFileDialog()
        dialog.setWindowTitle(caption)
        dialog.setDirectory(initial_dir)
        dialog.setNameFilters(FILE_FILTERS)
        dialog.selectNameFilter(initial_filter)
        dialog.setFileMode(QFileDialog.FileMode.ExistingFile)

        ok = dialog.exec()
        print(
```

```
            "Result:",
            ok,
            dialog.selectedFiles(),
            dialog.selectedNameFilter(),
        )

    def get_filenames(self):
        caption = "Open files"
        initial_dir = "" # 空文件夹使用当前文件夹。
        initial_filter = FILE_FILTERS[1] # 从列表中选择一个

        dialog = QFileDialog()
        dialog.setWindowTitle(caption)
        dialog.setDirectory(initial_dir)
        dialog.setNameFilters(FILE_FILTERS)
        dialog.selectNameFilter(initial_filter)
        dialog.setFileMode(QFileDialog.FileMode.ExistingFile)

        ok = dialog.exec()
        print(
            "Result:",
            ok,
            dialog.selectedFiles(),
            dialog.selectedNameFilter(),
        )

    def get_save_filename(self):
        caption = "Save As"
        initial_dir = "" # 空文件夹使用当前文件夹。
        initial_filter = FILE_FILTERS[1] # 从列表中选择一个

        dialog = QFileDialog()
        dialog.setWindowTitle(caption)
        dialog.setDirectory(initial_dir)
        dialog.setNameFilters(FILE_FILTERS)
        dialog.selectNameFilter(initial_filter)
        dialog.setFileMode(QFileDialog.FileMode.AnyFile)

        ok = dialog.exec()
        print(
            "Result:",
            ok,
            dialog.selectedFiles(),
            dialog.selectedNameFilter(),
        )

    def get_folder(self):
        caption = "Select folder"
        initial_dir = "" # 空文件夹使用当前文件夹。

        dialog = QFileDialog()
        dialog.setWindowTitle(caption)
        dialog.setDirectory(initial_dir)
        dialog.setFileMode(QFileDialog.FileMode.Directory)

        ok = dialog.exec()
```

```
        print(
            "Result:",
            ok,
            dialog.selectedFiles(),
            dialog.selectedNameFilter(),
        )

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 您将看到与之前相同的对话框启动程序，其中包含相同的按钮。

您会发现，采用这种方法时，对话框之间几乎没有区别——您只需设置适当的模式和窗口标题即可。在所有情况下，我们都通过 `dialog.selectedFiles()` 方法获取选中的文件，该方法返回一个列表，即使只选中了一个文件也是如此。最后，请注意，使用这种方法您可以将过滤器作为字符串列表传递给 `dialog.setNameFilters()`，而不是使用 `;;` 连接它们，尽管如果您更喜欢这种方式的话，您仍然可以使用 `dialog.setNameFilter()` 以 `;;` 连接的方式传递。

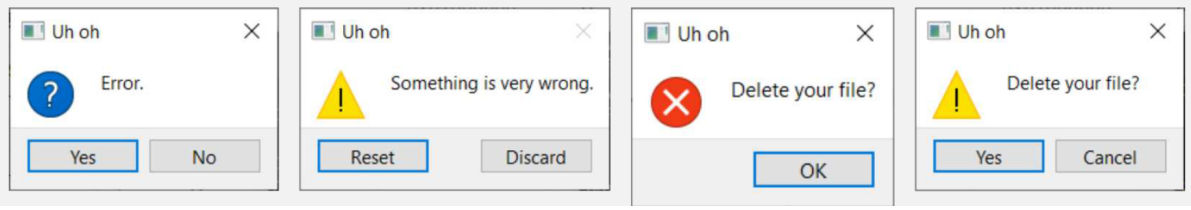
您可以选择任何您喜欢的方法。与之前一样，自定义的 `QFileDialog` 实例具有更高的可配置性（我们在这里只是略微涉及了部分内容）然而，静态方法具有非常合理的默认值，这将为您节省一些时间。

有了这些方法，您应该能够创建应用程序所需的任何对话框了！

> Qt 还提供了一些不太常用的对话框，用于显示进度条（`QProgressDialog`）、一次性错误消息（`QErrorMessage`）、选择颜色（`QColorDialog`）、选择字体（`QFontDialog`）以及显示向导以引导用户完成任务（`QWizard`）。有关详细信息，请参阅 Qt 文档。

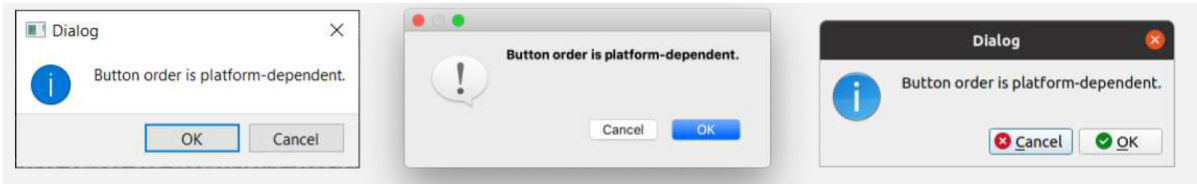
用户友好的对话框

创建糟糕的对话框特别容易。从让用户陷入困惑的选项中无法脱身的对话框，到层层嵌套的无休止弹出窗口，伤害用户的方式不胜枚举。



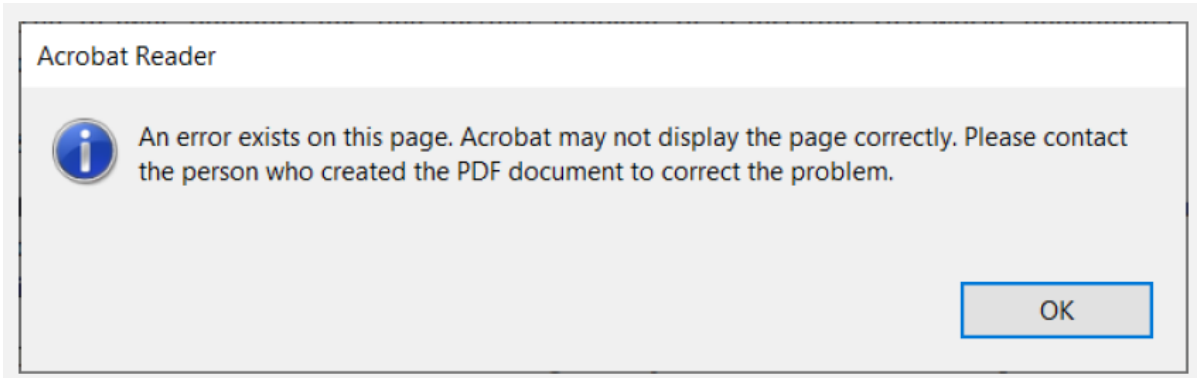
> 一些糟糕的对话示例——您发现第4个地方有什么问题了吗？默认操作具有破坏性！

对话框按钮由系统标准定义。您可能从未注意到 macOS 和 Linux 上的“确定”和“取消”按钮与 Windows 上的位置相反，但您的大脑已经注意到了！



> 对话框按钮的排列顺序取决于平台。

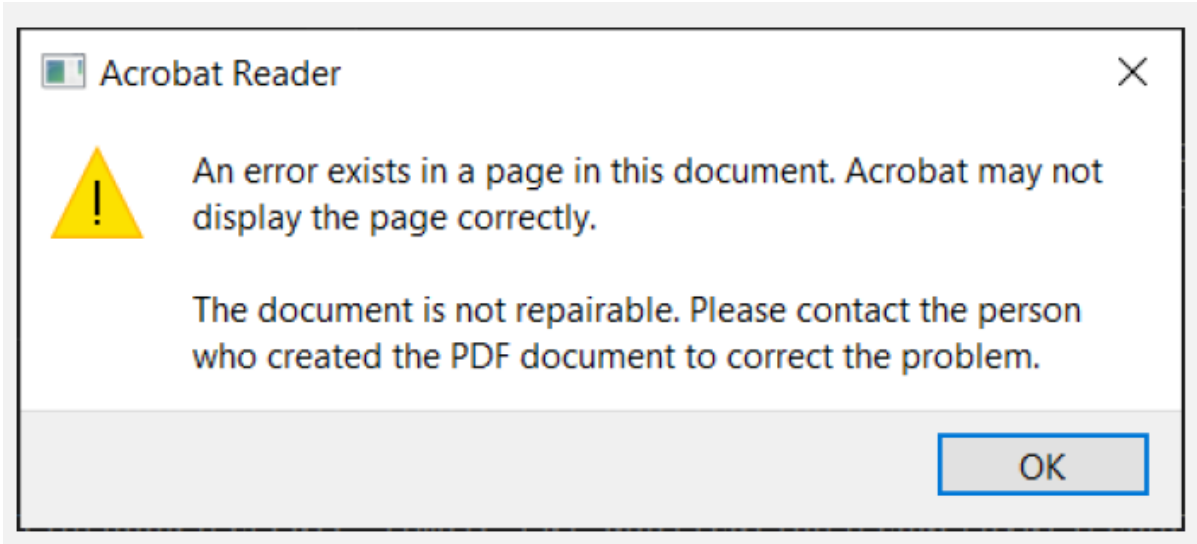
如果您不遵循标准，将会让用户感到困惑并导致他们犯错。使用Qt时，您在使用内置控件时可以免费获得这种一致性。**请务必使用它们**！



> 来自Adobe Acrobat Reader的真实对话框

错误对话框总是让用户感到烦躁。当您显示一个错误对话框时，您是在向用户传达坏消息。当您向某人传达坏消息时，您需要考虑它对他们的影响。

上面是Adobe Acrobat Reader中一个真实的错误对话框。请注意它如何解释存在错误、可能的后果以及潜在的解决方法。但它仍然不够完美。错误以信息对话框的形式显示，且该对话框在每页都会弹出。在文档中移动时无法抑制重复的消息。对话框文本也可进一步优化，以明确指出该错误无法恢复。



> Adobe Acrobat Reader DC 对话框的改进版本

良好的错误信息应说明——

- 发生了什么
- 哪些内容受到了影响
- 由此产生的后果是什么
- 可以采取哪些措施来解决

请务必花时间确保对话框设计合理。

请务必使用真实用户测试错误信息并根据反馈进行优化。

请勿假设用户能够理解编程术语或错误信息。

9. 窗口

在上一章中，我们探讨了如何打开对话框窗口。这些是特殊类型的窗口，默认情况下会捕获用户的焦点，并运行自己的事件循环，从而有效地阻塞应用程序其他部分的执行。

然而，在许多情况下，您可能希望在应用程序中打开第二个窗口，同时不阻塞主窗口——例如，用于显示某个长时间运行的进程的输出，或展示图表或其他可视化内容。或者，您可能希望创建一个应用程序，允许您同时处理多个文档，每个文档都在自己的窗口中。

在 PyQt6 中打开新窗口相对简单，但有几点需要注意以确保其正常工作。在本教程中，我们将逐步演示如何创建新窗口以及如何按需显示和隐藏外部窗口。

创建一个新窗口

要在 PyQt6 中创建一个新窗口，只需创建一个没有父对象的控件对象的新实例即可。该控件可以是任何控件（从技术上讲，可以是 `QWidget` 的任何子类），包括另一个 `QMainWindow`（如果您愿意的话）。

> `QMainWindow` 实例的数量没有限制，如果您需要在第二个窗口上使用工具栏或菜单，您也需要使用 `QMainWindow` 来实现。

与主窗口一样，创建窗口是不够的，您还必须显示它。

Listing 75. basic/windows_1.py

```
import sys
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class AnotherWindow(QWidget):
    """
    此“窗口”是一个QWidget。如果它没有父窗口，它将以自由浮动窗口的形式显示。
    """
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window")
        layout.addWidget(self.label)
        self.setLayout(layout)
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        w = AnotherWindow()
        w.show()


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

您运行这个程序，您会看到主窗口。点击按钮可能会显示第二个窗口，但如果您看到它，它只会显示一瞬间。发生了什么?

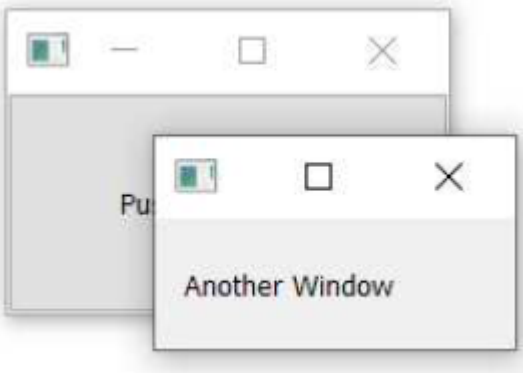
```
    def show_new_window(self, checked):
        w = AnotherWindow()
        w.show()
```

我们在该方法内创建第二个窗口，将其存储在变量 `w` 中并显示出来。然而，一旦离开该方法，Python 会清理 `w` 变量，导致窗口被销毁。为解决此问题，我们需要将窗口的引用保存在某个位置——例如主窗口的 `self` 对象中。

Listing 76. basic/windows_1b.py

```
    def show_new_window(self, checked):
        self.w = AnotherWindow()
        self.w.show()
```

现在，当您点击按钮以显示新窗口时，该窗口将保持打开状态。



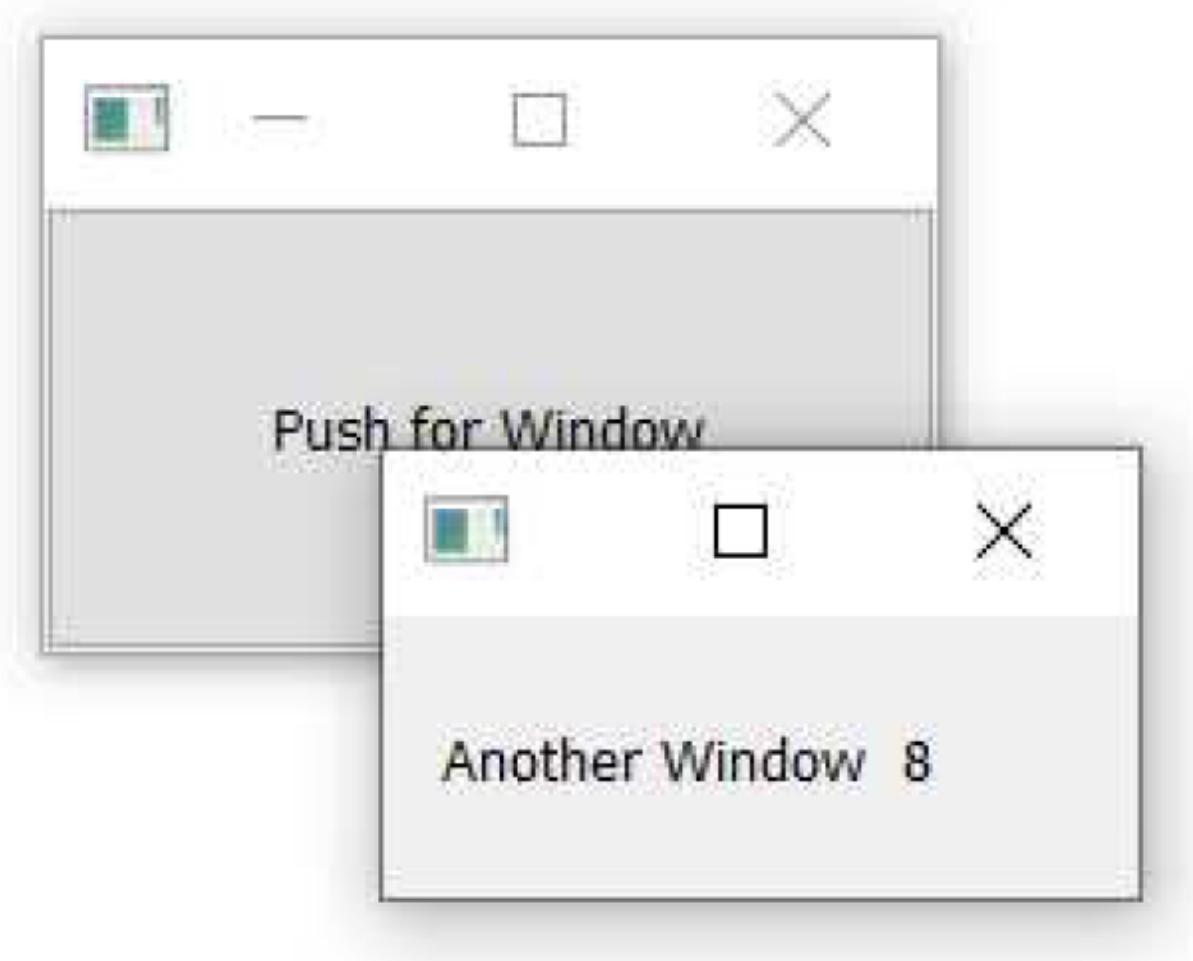
> 图六十六：第二个窗口持续存在。

然而，如果您再次点击按钮会发生什么？窗口将被重新创建！这个新窗口将替换 `self.w` 变量中的旧窗口，而之前的窗口将被销毁。如果您将 `AnotherWindow` 的定义修改为每次创建时在标签中显示一个随机数，您将更清楚地看到这一点。

Listing 77. basic/windows_2.py

```
from random import randint

from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

class AnotherWindow(QWidget):
    """
    此“窗口”是一个QWidget。如果它没有父窗口，它将以自由浮动窗口的形式显示。
    """

    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window % d" % randint(0, 100))
        layout.addWidget(self.label)
        self.setLayout(layout)
```

`__init__` 块仅在创建窗口时执行。如果您继续点击按钮，数字会发生变化，表明窗口正在被重新创建。



> 图六十七：如果再次按下按钮，数字将会改变。

一种解决方案是，在创建窗口之前，先检查该窗口是否已经存在。下面的完整示例展示了这一过程。

Listing 78. basic/windows_3.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.w = None # 目前尚未设置外部窗口。
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        if self.w is None:
            self.w = AnotherWindow()
            self.w.show()
```

这种方法适用于临时创建的窗口，或需要根据程序当前状态进行更改的窗口——例如，您想显示特定的图表或日志输出。然而，对于许多应用程序而言，您可能需要一些标准窗口，希望能够按需显示或隐藏。

在接下来的部分中，我们将探讨如何处理此类窗口。

关闭一个窗口

如前所述，如果没有对窗口的引用被保留，它将被丢弃（并关闭）。我们可以利用这种行为来关闭窗口，将前一个示例中的 `show_new_window` 方法替换为

Listing 79. basic/windows_4.py

```
    def show_new_window(self, checked):
        if self.w is None:
            self.w = AnotherWindow()
            self.w.show()
        else:
            self.w = None # 取消引用，关闭窗口
```

通过将 `self.w` 设置为 `None`（或任何其他值），对窗口的现有引用将被丢失，窗口将关闭。然而，如果我们将它设置为除 None 以外的任何其他值，第一个测试将不再通过，我们将无法重新创建一个窗口。

这仅在您未在其他地方保留此窗口的引用时有效。为了确保窗口无论如何都会关闭，您可能需要显式调用 `.close()` 方法。

Listing 80. basic/windows_4b.py

```
    def show_new_window(self, checked):
        if self.w is None:
            self.w = AnotherWindow()
            self.w.show()

        else:
            self.w.close()
            self.w = None # 取消引用，关闭窗口
```

持久窗口

到目前为止，我们已经探讨了如何按需创建新窗口。然而，有时您会遇到多个标准应用程序窗口的情况。在这种情况下，通常更合理的方法是先创建这些额外窗口，然后在需要时使用 `.show()` 方法将其显示出来。

在以下示例中，我们在主窗口的 `__init__` 块中创建外部窗口，然后我们的 `show_new_window`方法只需调用 `self.w.show()` 即可显示它。

Listing 81. basic/windows_5.py

```
import sys
from random import randint

from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)
```

```
class AnotherWindow(QWidget):
    """
    此“窗口”是一个QWidget。如果它没有父窗口，它将以自由浮动窗口的形式显示。
    """

    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window % d" % randint(0, 100))
        layout.addWidget(self.label)
        self.setLayout(layout)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        self.w.show()


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

如果您运行这个程序，点击按钮会像之前一样显示窗口。注意，窗口只创建一次，对已经可见的窗口调用 `.show()` 方法不会产生任何效果。

显示与隐藏窗口

一旦您创建了持久窗口，您就可在不重新创建的情况下显示或隐藏它。隐藏后，窗口仍存在但不可见，且不会响应鼠标或其他输入。然而，您仍可继续调用该窗口的方法并更新其状态——包括更改其外观。重新显示后，所有更改将立即生效。

下面我们更新主窗口，创建一个 `toggle_window` 方法，该方法使用 `.isVisible()` 方法检查窗口是否当前可见。如果不可见，则使用 `.show()` 方法显示它；如果已可见，则使用 `.hide()` 方法隐藏它。

```
class MainWindow(QMainWindow):

    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.toggle_window)
        self.setCentralWidget(self.button)

    def toggle_window(self, checked):
```

```
        if self.w.isVisible():
            self.w.hide()

        else:
            self.w.show()
```

以下是此持久窗口及显示/隐藏状态切换的完整示例：

Listing 82. basic/windows_6.py

```
import sys
from random import randint

from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

class AnotherWindow(QWidget):
    """
    此“窗口”是一个QWidget。如果它没有父窗口，它将以自由浮动窗口的形式显示。
    """

    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window % d" % randint(0, 100))
        layout.addWidget(self.label)
        self.setLayout(layout)


class MainWindow(QMainWindow):

    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.toggle_window)
        self.setCentralWidget(self.button)

    def toggle_window(self, checked):
        if self.w.isVisible():
            self.w.hide()

        else:
            self.w.show()


app = QApplication(sys.argv)

window = MainWindow()
```

```
window.show()

app.exec()
```

同样，窗口仅创建一次——窗口的 `__init__` 块不会在每次重新显示窗口时重新运行（因此标签中的数字不会改变）。

连接窗口之间的信号

在信号一章中，我们看到了如何直接使用信号和槽将控件连接在一起。我们只需要创建目标控件，并通过变量引用它即可。连接跨窗口的信号时，同样的原则也适用——您可以将一个窗口中的信号连接到另一个窗口中的槽，也就是说，您只需要能够访问该槽即可。

在下面的示例中，我们将主窗口上的文本输入框连接到子窗口上的 `QLabel` 控件。

Listing 83. basic/windows_7.py

```
import sys
from random import randint

from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
    QLineEdit,
)



class AnotherWindow(QWidget):
    """
    此“窗口”是一个QWidget。如果它没有父窗口，它将以自由浮动窗口的形式显示。
    """

    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window") #2
        layout.addWidget(self.label)
        self.setLayout(layout)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.toggle_window)

        self.input = QLineEdit()
        self.input.textChanged.connect(self.w.label.setText) #1
        layout = QVBoxLayout()
```

```
        layout.addWidget(self.button)
        layout.addWidget(self.input)
        container = QWidget()
        container.setLayout(layout)

        self.setCentralWidget(container)

    def toggle_window(self, checked):
        if self.w.isVisible():
            self.w.hide()

        else:
            self.w.show()


app = QApplication(sys.argv)

w = MainWindow()
w.show()

app.exec()
```

1. `AnotherWindow` 窗口对象可通过变量 `self.w` 访问。`QLabel` 可通过 `self.w.label` 和 `.setText` 槽通过 `self.w.label.setText` 访问。
2. 当创建 `QLabel` 时，我们将对其的引用存储在 `self` 上作为 `self.label`，因此可以在对象外部访问它。

> 🚀 **运行它吧！** 在上方框中输入一些文本，您会看到它立即出现在标签上。即使窗口被隐藏，文本也会更新——控件状态的更新并不依赖于它们是否可见。

当然，您也可以将一个窗口上的信号连接到另一个窗口上的方法。只要可以访问，任何操作都是可行的。确保组件可以相互导入和访问是构建逻辑项目结构的一个很好的动机。通常，在主窗口/模块中集中连接组件是合理的，这样可以避免交叉导入所有内容。

10. 事件

用户与 Qt 应用程序之间的每次交互都是一个事件。事件有多种类型，每种类型代表一种不同的交互类型。Qt 使用事件对象来表示这些事件，事件对象打包了关于发生事件的信息。这些事件被传递到发生交互的控件上的特定事件处理程序。

通过定义自定义事件处理程序，您可以更改控件对这些事件的响应方式。事件处理程序与其他方法一样进行定义，但名称是根据它们处理的事件类型来指定的。

控件接收的主要事件之一是 `QMouseEvent` 。`QMouseEvent` 事件是在控件上每次鼠标移动和按钮点击时创建的。以下事件处理程序可用于处理鼠标事件：

事件处理器	被更改的事件
`mouseMoveEvent`	鼠标移动
`mousePressEvent`	鼠标按钮被按下
`mouseReleaseEvent`	鼠标按钮被松开
`mouseDoubleClickEvent`	检测到双击

例如，单击一个控件会触发一个 `QMouseEvent` 事件，该事件将被发送给该控件的 `.mousePressEvent` 事件处理程序。该处理程序可以使用事件对象来查找发生的事情的相关信息，例如触发该事件的原因以及具体发生的位置。

您可以通过继承类并重写处理方法来拦截事件。您可以选择过滤、修改或忽略事件，并将它们传递给事件的正常处理程序，方法是调用父类函数并使用 `super()` 。这些可以添加到您的主窗口类中，如下例所示。在每种情况下，参数 `e` 将接收传入的事件。

Listing 84. basic/events_1.py

```
import sys
from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QTextEdit,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.label = QLabel("Click in this window")
        self.setCentralWidget(self.label)

    def mouseMoveEvent(self, e):
        self.label.setText("mouseMoveEvent")

    def mousePressEvent(self, e):
        self.label.setText("mousePressEvent")

    def mouseReleaseEvent(self, e):
        self.label.setText("mouseReleaseEvent")

    def mouseDoubleClickEvent(self, e):
        self.label.setText("mouseDoubleClickEvent")


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

> 🚀 **运行它吧！** 请您尝试在窗口中移动和点击（以及双击），观察事件的出现。

您会发现，鼠标移动事件仅会在按下按钮时触发。您可以通过调用 `self.setMouseTracking(True)` 方法在窗口上更改此行为。您还可能注意到，按下（点击）和双击事件在按下按钮时均会触发。仅在释放按钮时触发释放事件。通常，要注册用户的点击事件，您应该监听鼠标按下和释放两个事件。

在事件处理程序中，您可以访问一个事件对象。该对象包含有关事件的信息，并可根据具体发生的情况采取不同的响应方式。接下来我们将探讨鼠标事件对象。

鼠标事件

Qt 中所有鼠标事件均通过 `QMouseEvent` 对象进行跟踪，相关事件信息可通过以下事件方法进行读取。

方法	返回的结果
`.button()`	触发此事件的特定按钮
`.buttons()`	所有鼠标按钮的状态（或运算标志）
`.position()` | 控件的相对位置，以 QPoint` 整数表示	

您可以在事件处理程序中使用这些方法来对不同的事件做出不同的响应，或者完全忽略它们。`.position()` 方法以 `QPoint` 对象的形式提供控件的相对位置信息，而按钮则使用 Qt 命名空间中的鼠标按钮类型进行报告。

例如，以下代码允许我们对窗口的左键、右键或中键点击做出不同的响应。

Listing 85. basic/events_2.py

```
    def mousePressEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            # 在此处理左键按下事件
            self.label.setText("mousePressEvent LEFT")

        elif e.button() == Qt.MouseButton.MiddleButton:
            # 在此处理中间按钮的按下操作
            self.label.setText("mousePressEvent MIDDLE")

        elif e.button() == Qt.MouseButton.RightButton:
            # 在此处处理右键按下事件
            self.label.setText("mousePressEvent RIGHT")

    def mouseReleaseEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            self.label.setText("mouseReleaseEvent LEFT")

        elif e.button() == Qt.MouseButton.MiddleButton:
            self.label.setText("mouseReleaseEvent MIDDLE")

        elif e.button() == Qt.MouseButton.RightButton:
            self.label.setText("mouseReleaseEvent RIGHT")

    def mouseDoubleClickEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            self.label.setText("mouseDoubleClickEvent LEFT")

        elif e.button() == Qt.MouseButton.MiddleButton:
            self.label.setText("mouseDoubleClickEvent MIDDLE")

        elif e.button() == Qt.MouseButton.RightButton:
            self.label.setText("mouseDoubleClickEvent RIGHT")
```

按钮标识符在 Qt 命名空间中定义，如下所示：

标识符	值(二进制)	代表的事件
`Qt.MouseButtons.NoButton`	0(`000`)	未按下按钮，或该事件与按下按钮无关。
`Qt.MouseButtons.LeftButton`	1(`001`)	左键被按下
`Qt.MouseButtons.RightButton`	2(`010`)	右键被按下
`Qt.MouseButtons.MiddleButton`	3(`100`)	中间的按键被按下

> 对于右手鼠标，左右按钮的位置是相反的，即按下最右边的按钮将返回 `Qt.MouseButtons.LeftButton` 。这意味着您无需在代码中考虑鼠标的方向。

> 要更深入地了解这一切的工作原理，请参阅后文的 “35. 枚举和 Qt 命名空间”。

上下文菜单

上下文菜单是小型上下文相关菜单，通常在右键单击窗口时出现。Qt 支持生成这些菜单，并且控件有一个用于触发它们的特定事件。在下面的示例中，我们将拦截 `QMainWindow` 的 `.contextMenuEvent` 。每当上下文菜单即将显示时，都会触发此事件，并传递一个类型为 `QContextMenuEvent` 的单一值事件。

要拦截该事件，我们只需用我们的新方法覆盖对象方法，该方法具有相同的名称。因此，在这种情况下，我们可以在 `MainWindow` 子类中创建一个名为 `contextMenuEvent` 的方法，它将接收所有此类型的事件。

Listing 86. basic/events_3.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QAction
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QMenu,
)
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

    def contextMenuEvent(self, e):
        context = QMenu(self)
        context.addAction(QAction("test 1", self))
        context.addAction(QAction("test 2", self))
        context.addAction(QAction("test 3", self))
        context.exec(e.globalPos())

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

如果运行上述代码并在窗口内右键单击，您会看到一个上下文菜单出现。您可以像往常一样在菜单操作上设置 `.triggered` 槽（并重新使用为菜单和工具栏定义的操作）。

> 在将初始位置传递给 `exec()` 方法时，该位置必须相对于在定义时传递的父对象。在此情况下我们传递 `self` 作为父对象，因此可以使用全局位置。

为了完整起见，还有一种基于信号的方法来创建上下文菜单。

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.show()

        self.setContextMenuPolicy(
            Qt.ContextMenuPolicy.CustomContextMenu
        )
        self.customContextMenuRequested.connect(self.on_context_menu)

    def on_context_menu(self, pos):
        context = QMenu(self)
        context.addAction(QAction("test 1", self))
        context.addAction(QAction("test 2", self))
        context.addAction(QAction("test 3", self))
        context.exec(self.mapToGlobal(pos))
```

完全由您决定选择哪一个。

事件层次结构

在 pyqt6 中，每个控件都是两个不同层次结构的一部分：Python 对象层次结构和 Qt 布局层次结构。您对事件做出响应或忽略事件的方式会影响用户界面的行为。

Python 继承转发

通常情况下，您可能希望拦截一个事件，对其进行处理，但仍然触发默认事件处理行为。如果您的对象是从标准控件继承的，则它很可能默认实现了合理的行为。您可以通过调用 `super()` 调用父级实现来触发此行为。

> 这是 Python 的父类，而不是 pyqt6 的 `.parent()`。

```
def mousePressEvent(self, event):
    print("Mouse pressed!")
    super(self, MainWindow).contextMenuEvent(event)
```

该事件将继续按正常方式运行，但您添加了部分不干扰的行为。

布局转发

当您将控件添加到应用程序时，它也会从布局中获得另一个父级。可以通过调用 `.parent()` 来找到控件的父级。有时，您可以手动指定这些父级，例如对于 `QMenu` 或 `QDialog`，但通常情况下，这是自动完成的。例如，当您将控件添加到主窗口时，主窗口将成为该控件的父级。

当您为用户与用户界面的交互创建事件时，这些事件将传递到用户界面中最上面的控件。如果单击窗口中的按钮，该按钮将在窗口之前接收事件。如果第一个控件无法处理事件，或者选择不处理，则事件将向上传播到父控件，该控件将获得处理事件的机会。这种向上传播将一直持续到嵌套控件，直到事件被处理或到达主窗口。

在您自己的事件处理程序中，您可以选择通过调用 `.accept()` 方法将事件标记为已处理。

```
    class CustomButton(Qbutton)
        def mousePressEvent(self, e):
            e.accept()
```

或者，您可以通过调用事件对象的 `.ignore()` 方法将其标记为未处理。在这种情况下，事件将继续向上级层级传递，就像冒泡一样。

```
    class CustomButton(Qbutton)
        def mousePressEvent(self, e):
            e.ignore()
```

如果您希望控件对事件保持透明，则可以放心地忽略您已经以某种方式响应过的事件。同样，您也可以选择接受您未响应的事件，以让它们静默处理。

> 这可能会造成混淆，因为您可能会认为调用 `.ignore()` 会完全忽略该事件。但事实并非如此：您只是忽略了此控件的事件！

Qt Designer(Qt设计师)

到目前为止，我们一直使用 Python 创建应用程序。在许多情况下，这很有效，但随着应用程序的规模越来越大，或者界面越来越复杂，通过编程来定义所有控件会变得有些繁琐。好消息是，Qt 附带了一个图形化编辑器——Qt Designer——其中包含一个拖放式用户界面编辑器。使用 Qt Designer ，您可以可视化地定义你的用户界面，然后只需在后续阶段将应用程序逻辑与之关联即可。

在本章中，我们将介绍使用 Qt Designer 创建用户界面的基础知识、原理、布局和控件都是相同的，因此您可以应用已经学到的所有知识。您还需要了解 Python API，以便稍后连接应用程序逻辑。

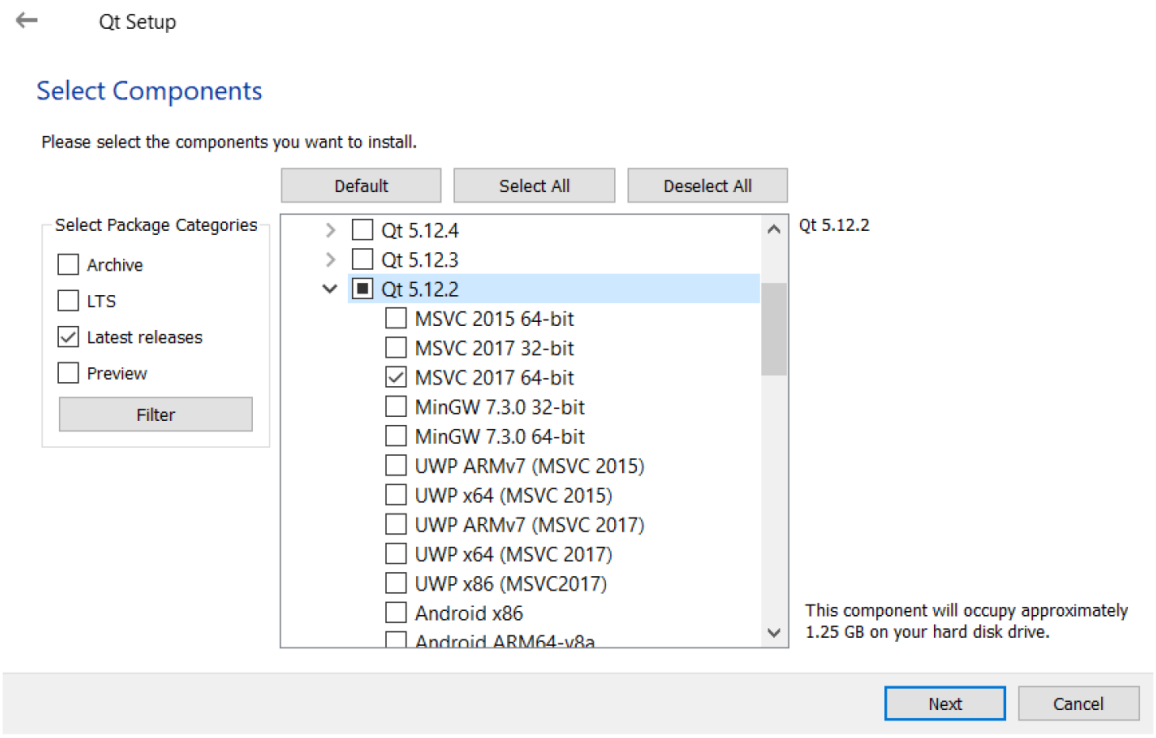
11. 下载 Qt Designer

Qt Designer 包含在 Qt 的安装包中，可从 Qt下载页面获取。请您下载并运行适用于您系统的相应安装程序，并按照以下平台特定的说明操作。安装 Qt Designer 不会影响您的 PyQt6 安装。

> Qt Creator 与 Qt Designer
>
> 您可能还会看到关于 Qt Creator 的提及。Qt Creator 是一个功能齐全的Qt 项目集成开发环境（IDE），而 Qt Designer 是用户界面设计组件。Qt Designer 包含在 Qt Creator 中，因此您可以选择安装它，尽管它对 Python 项目没有额外价值。

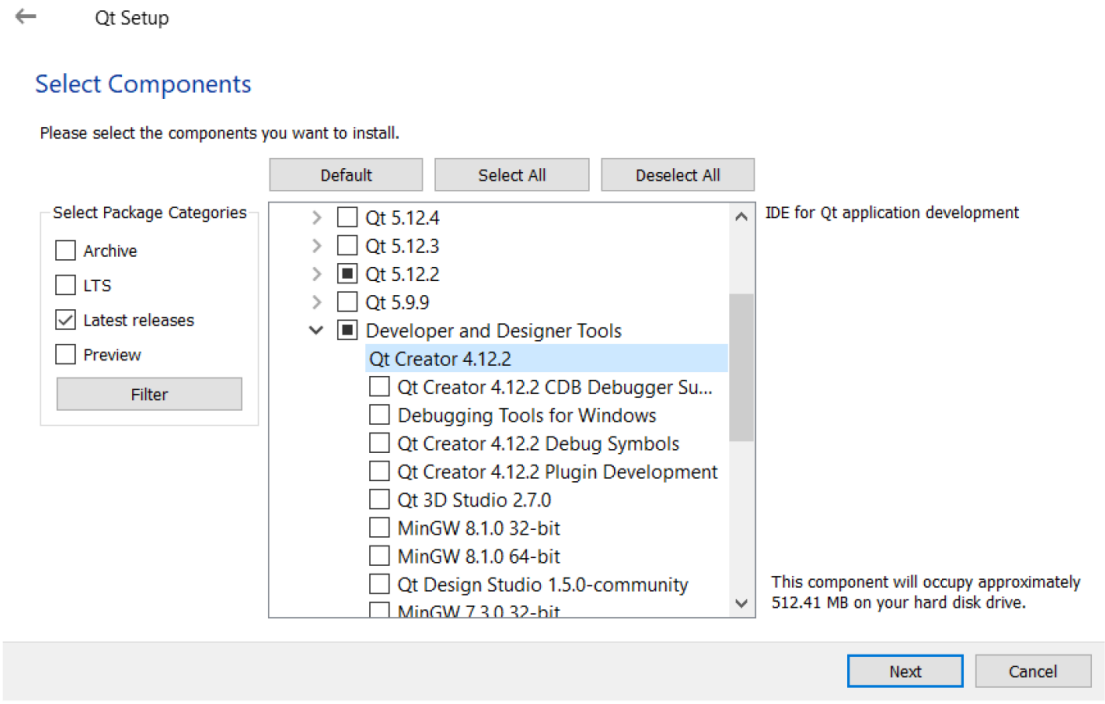
Windows系统

Qt Designer 在 Windows Qt 安装程序中未被提及，但会在安装任何版本的 Qt 核心库时自动安装。例如，在以下截图中，我们选择了安装 MSVC 2017 64 位版本的Qt —— 您的选择不会影响 Designer 的安装。



> 图六十八：安装 Qt 时，也会一并安装 Qt Designer。

如果您想安装 Qt Creator，它列在“开发者和设计师工具”类别下。令人困惑的是，Qt Designer 并未包含在此类别中。



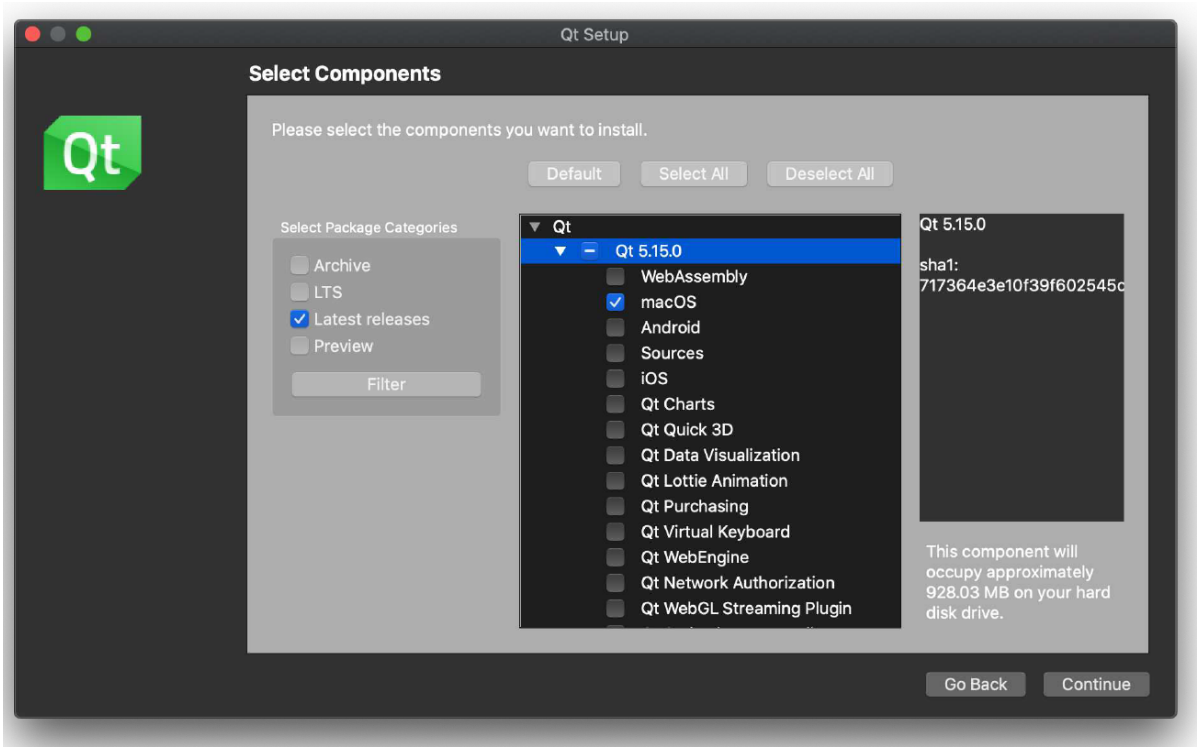
> 图六十九：正在安装 Qt Creator 组件

macOS系统

Qt Designer 在 macOS Qt 安装程序中未被提及，但会在安装任何版本的 Qt 核心库时自动安装。请从 Qt 官网下载安装程序——您可以选择开源版本。

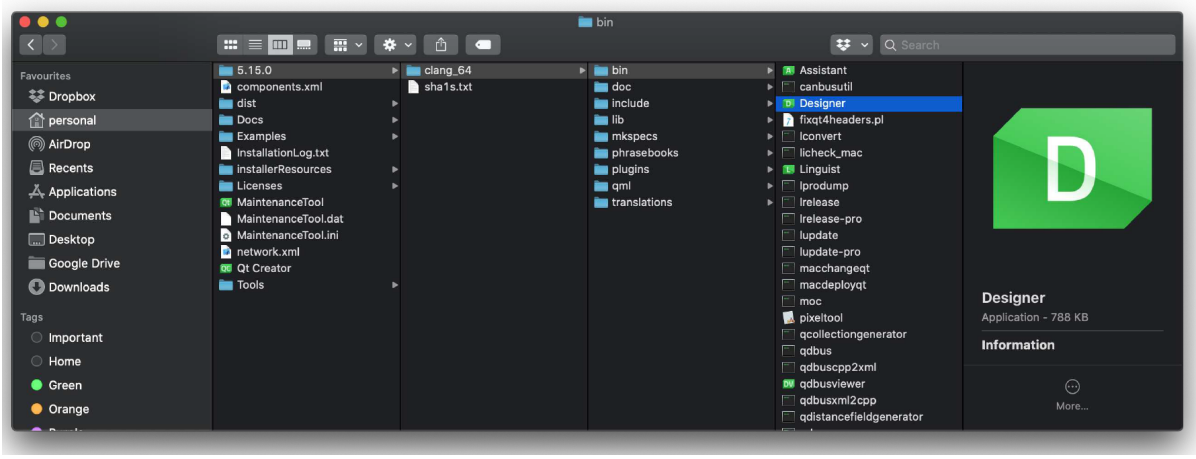
> 图七十：您会在下载的 `.dmg` 文件中找到安装程序。

请您打开安装程序以开始安装。继续操作直至出现选择安装组件的界面。然后，在最新版本的Qt下选择macOS安装包。



> 图七十一：您只需使用最新版本的macOS安装包即可。

安装完成后，请打开您安装 Qt 的文件夹。Designer 的启动程序位于 `<版本>/clang_64/bin` 目录下。您会发现 QtCreator 也安装在 Qt 安装文件夹的根目录中。



> 图七十二：您可以在 `<版本>/clang_64/bin` 文件夹下找到Designer启动器。

您可以直接从当前位置运行Designer，或将其移动到Applications文件夹中，以便通过macOS启动台启动。

Linux (Ubuntu & Debian)

您可以通过包管理器安装 Qt Designer。根据您的发行版和版本，您可以使用 Qt5 Designer 或 Qt6 Designer。您可以使用其中任何一个来开发 PyQt6 的用户界面设计。

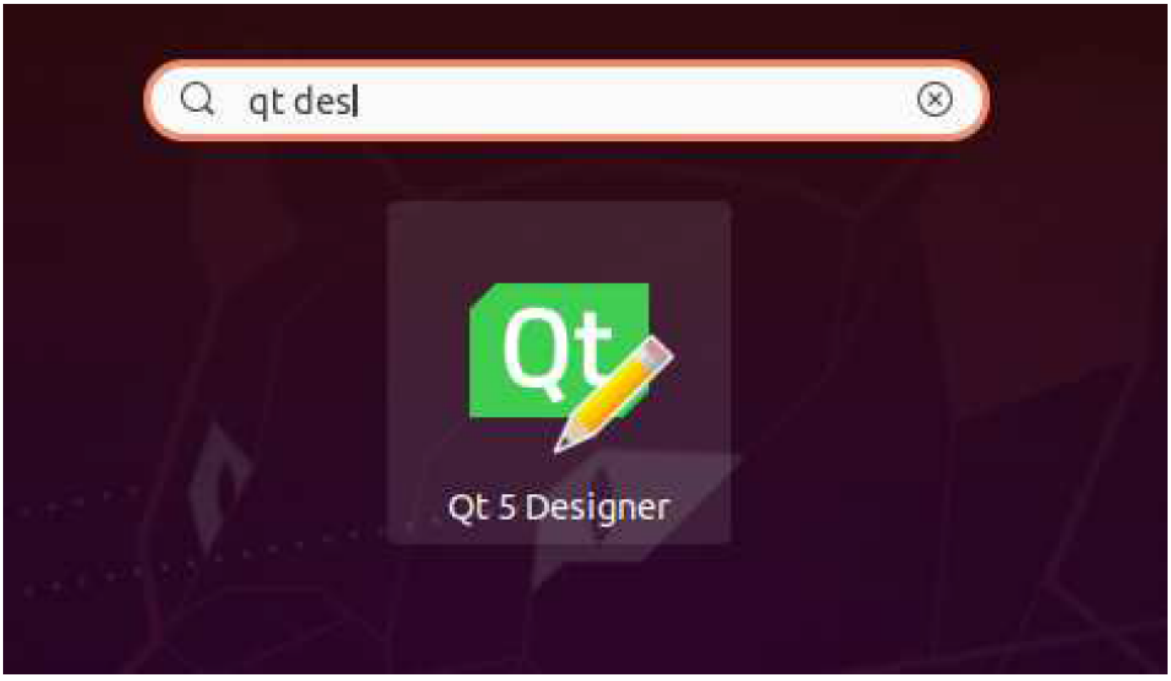
您可以使用以下命令安装 Qt5 Designer：

```
sudo apt-get install qttools5-dev-tools
```

同理，您也可以安装Qt6 Designer，这次用：

```
sudo apt-get install designer-qt6
```

安装完成后，Qt Designer 将出现在启动器中。



> 图七十三：在 Ubuntu 启动器中的Qt Designer

12. 开始使用 Qt Designer

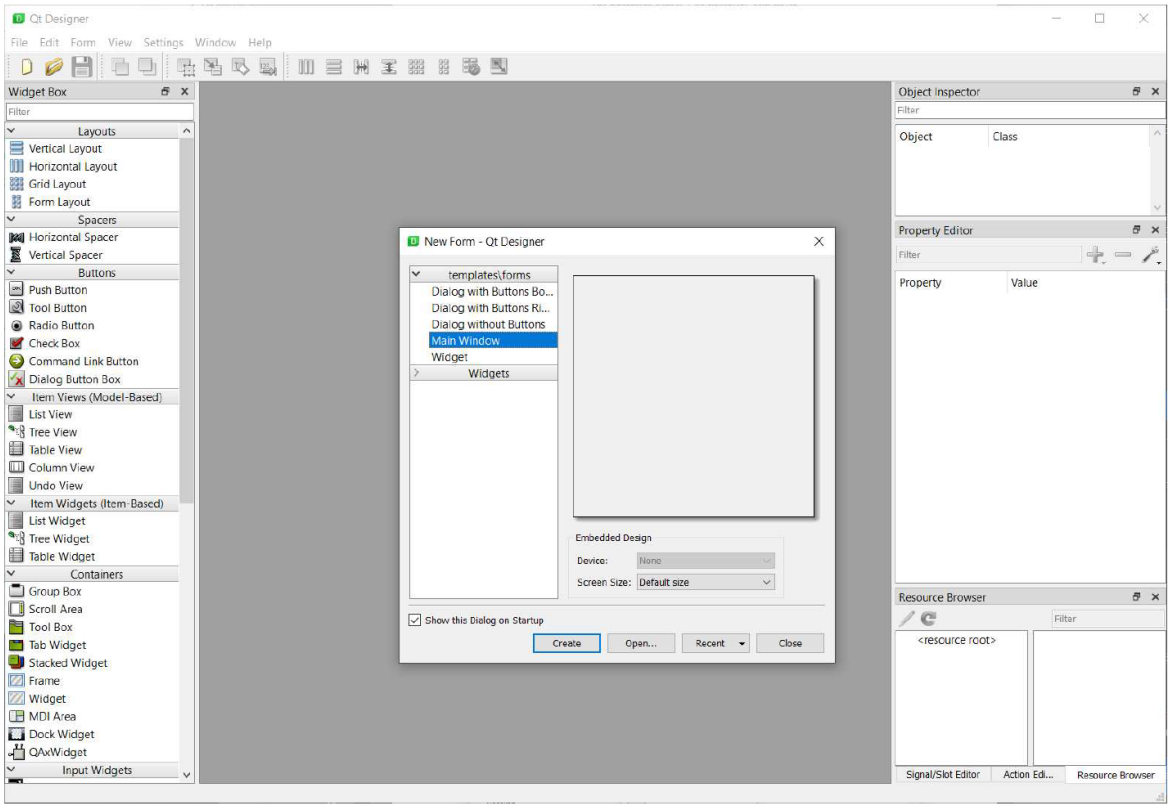
在本章中，我们将快速了解如何使用 Qt Designer 设计用户界面，并将其导出以在您的 PyQt6 应用程序中使用。这里我们仅会简要介绍Qt Designer 的基本功能。一旦您掌握了基础知识，便可以自行进一步探索和尝试。

打开 Qt Designer，您将看到主窗口。设计器可通过左侧的选项卡访问。然而，要激活此功能，您首先需要开始创建一个 `.ui` 文件。

Qt Designer

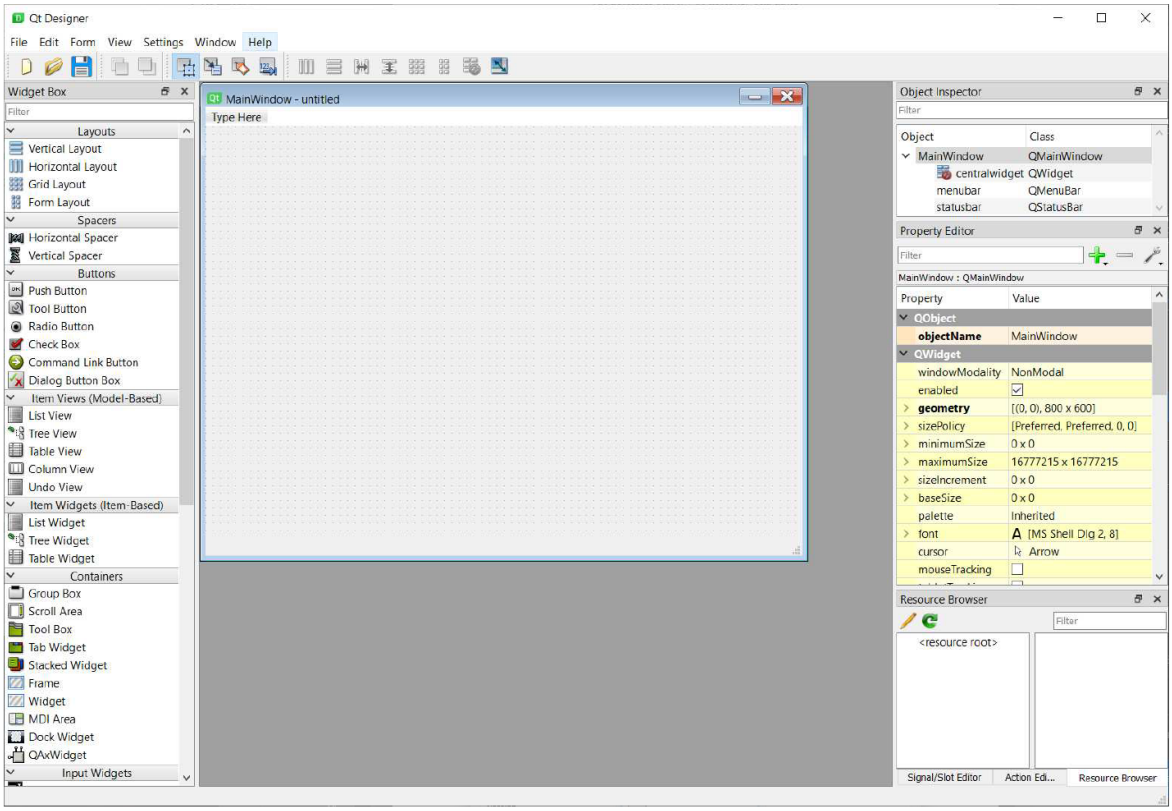
Qt Designer 启动时会显示“新建表单”对话框。在这里，您可以选择要构建的界面类型——这决定了您将构建界面的基础控件。如果您要启动一个应用程序，则“主窗口”通常是正确的选择。但是，您也可以为对话框和自定义复合控件创建 `.ui` 文件。

> 表单是用户界面布局的术语，因为许多用户界面与带有各种输入框的纸质表单相似。



> 图七十四：Qt Designer 界面

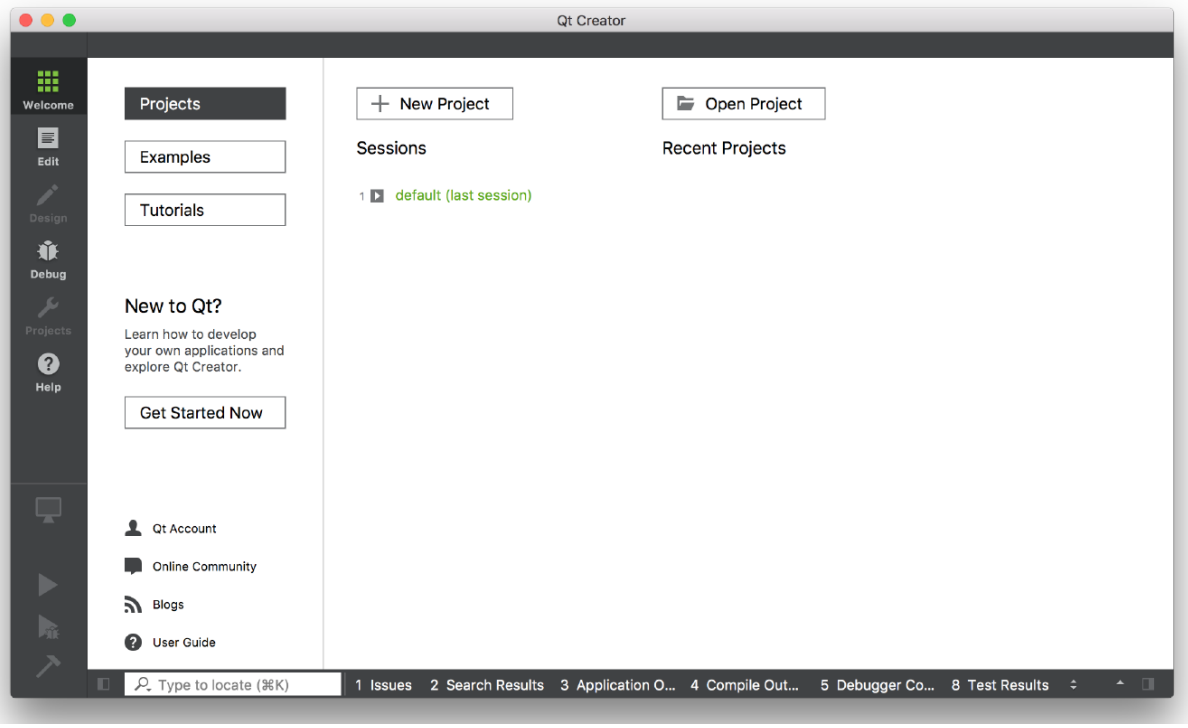
点击“Create”，将创建一个新用户界面，其中包含一个空的控件。现在，您可以开始设计应用程序了。



图七十五：Qt Designer 编辑器界面，其中包含一个空的 `QMainWindow` 控件。

Qt Creator

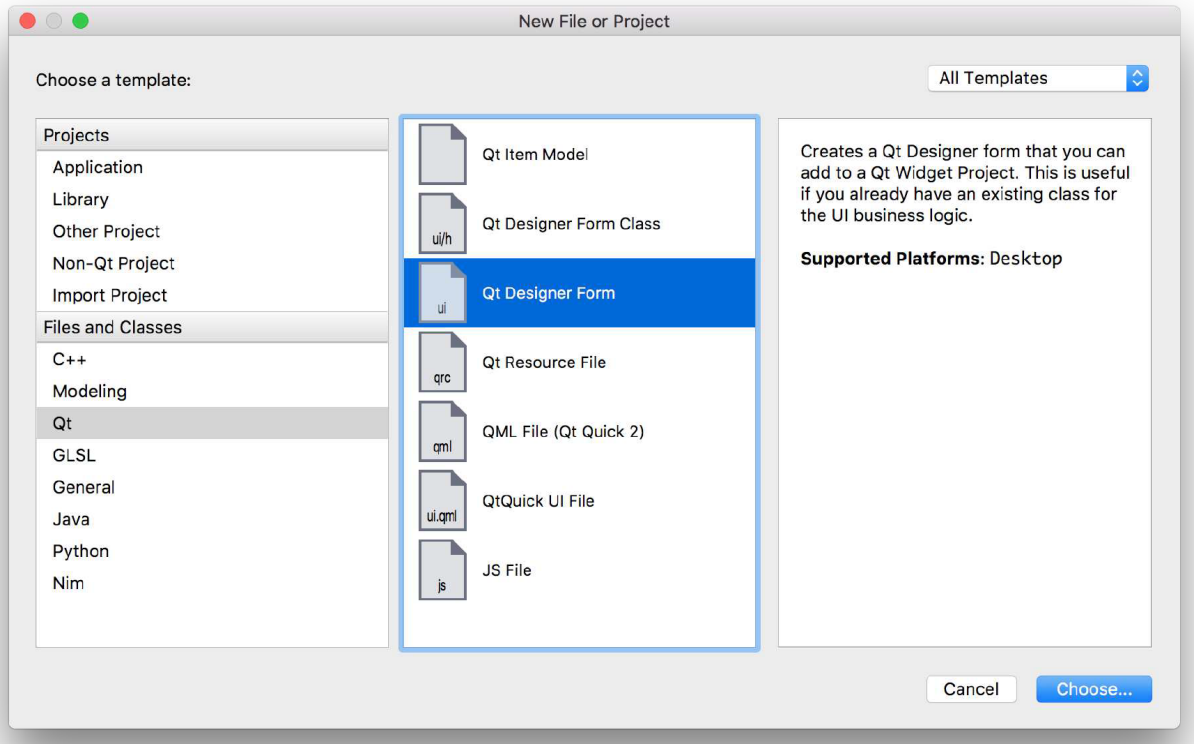
如果您已安装 Qt Creator，界面和流程会略有不同。左侧有一个类似标签的界面，您可以从中选择应用程序的各种组件。其中之一是“Design”，它会在主面板中显示 Qt Designer。



图七十六：Qt Creator 界面，左侧选中“Design”部分。QtDesigner 界面与嵌套的 Designer 界面完全相同。

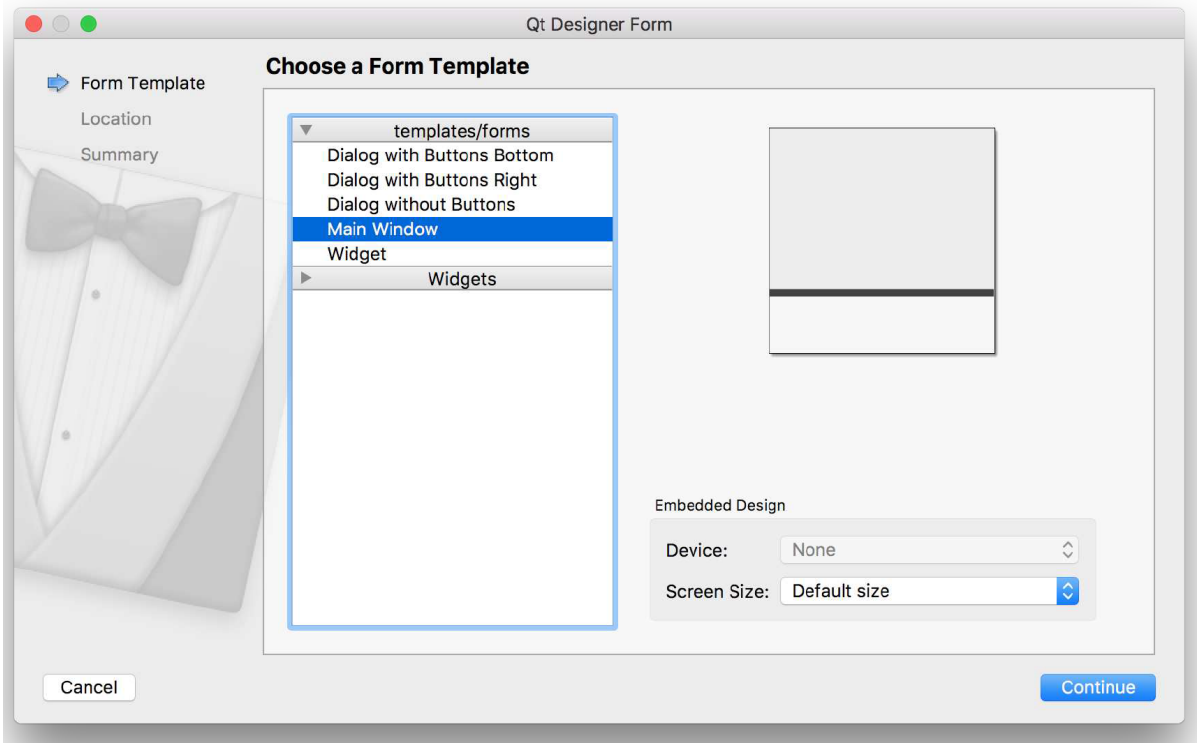
> Qt Designer 的所有功能均可在 Qt Creator 中使用，但用户界面的某些方面有所不同。

要创建一个 `.ui` 文件，请转到“File → New File”或"Project..."在弹出的窗口中在左侧的“Files and Classes”下选择“Qt”，然后在右侧选择“Qt Designer Form”。您会注意到图标上标有“ui”，表明您正在创建的文件类型。



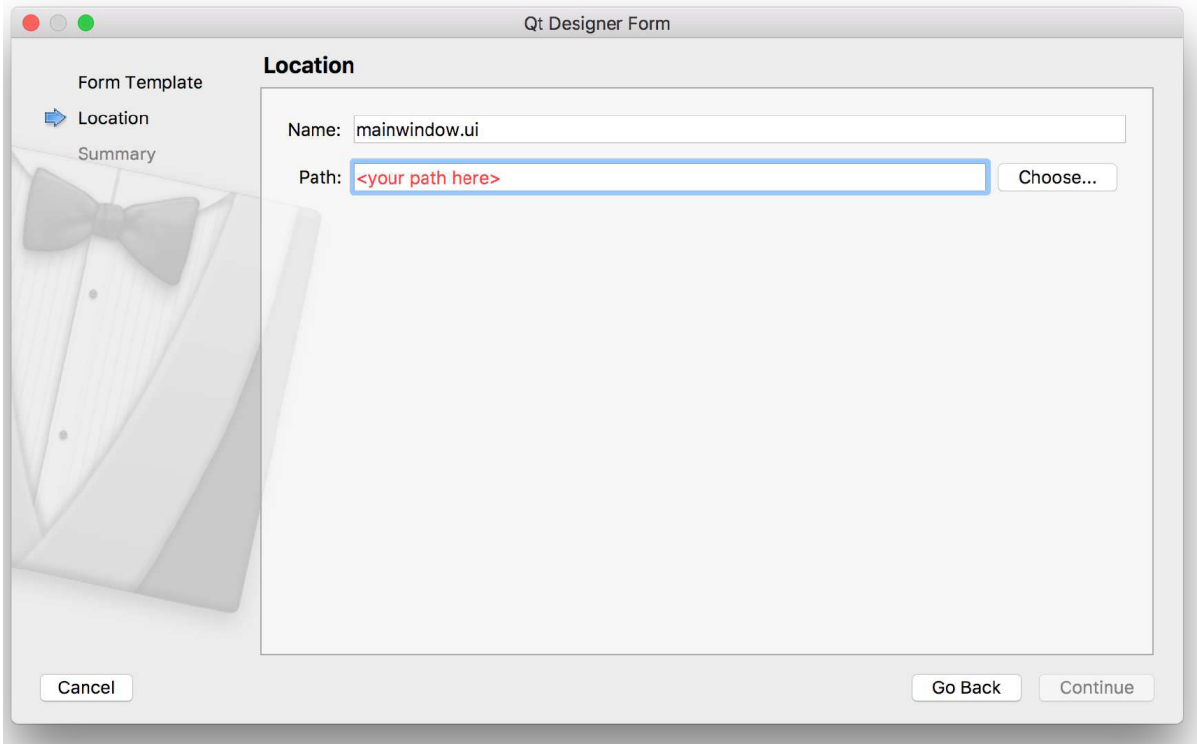
> 图七十七：创建一个新的 Qt .ui 文件。

下一步，系统会询问您要创建哪种类型的用户界面。对于大多数应用程序而言，主窗口"Main Window"是正确的选择。但是，您也可以为其他对话框创建 .ui文件，或使用 `QWidget`（列为“Widget”）构建自定义控件。



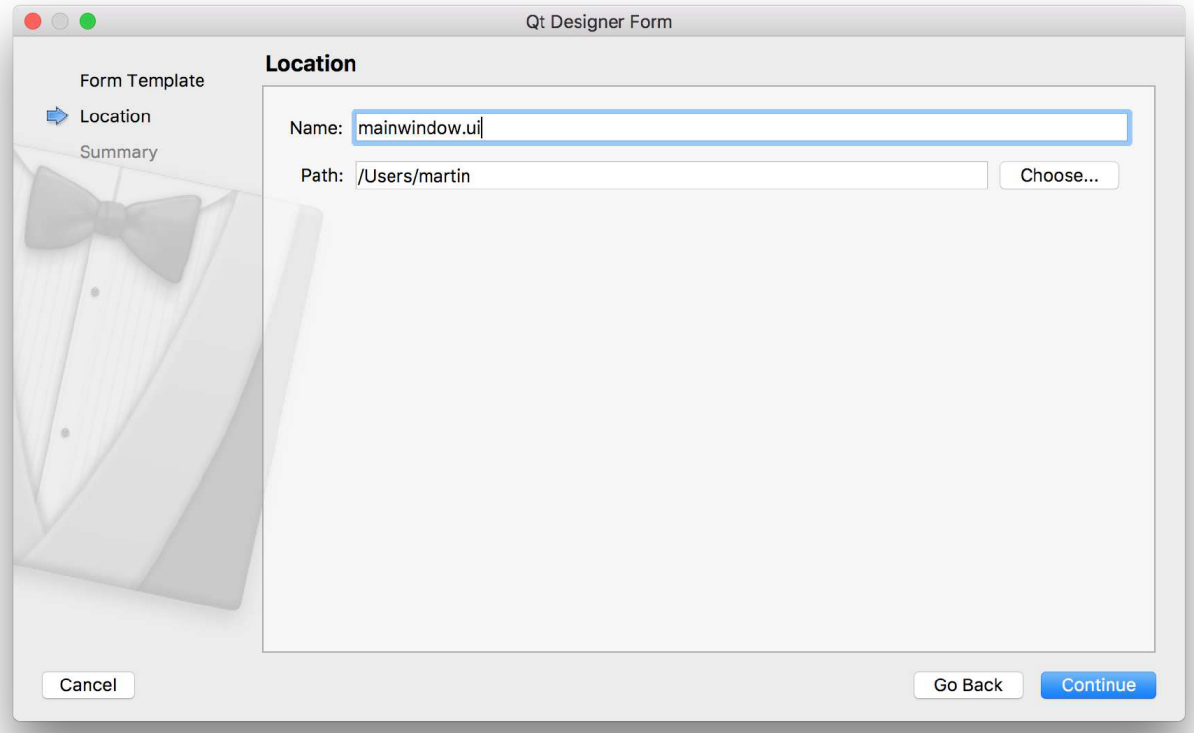
> 图七十八：选择要创建的控件类型，对于大多数应用程序而言，此选项为“Main Window”

接下来，您应该为您的文件选择一个文件名和保存文件夹。将您的 `.ui` 文件保存为与您将要创建的类相同的名称，这样可以使后续命令更加简单。



> 图七十九：选择文件的保存名称和文件夹。

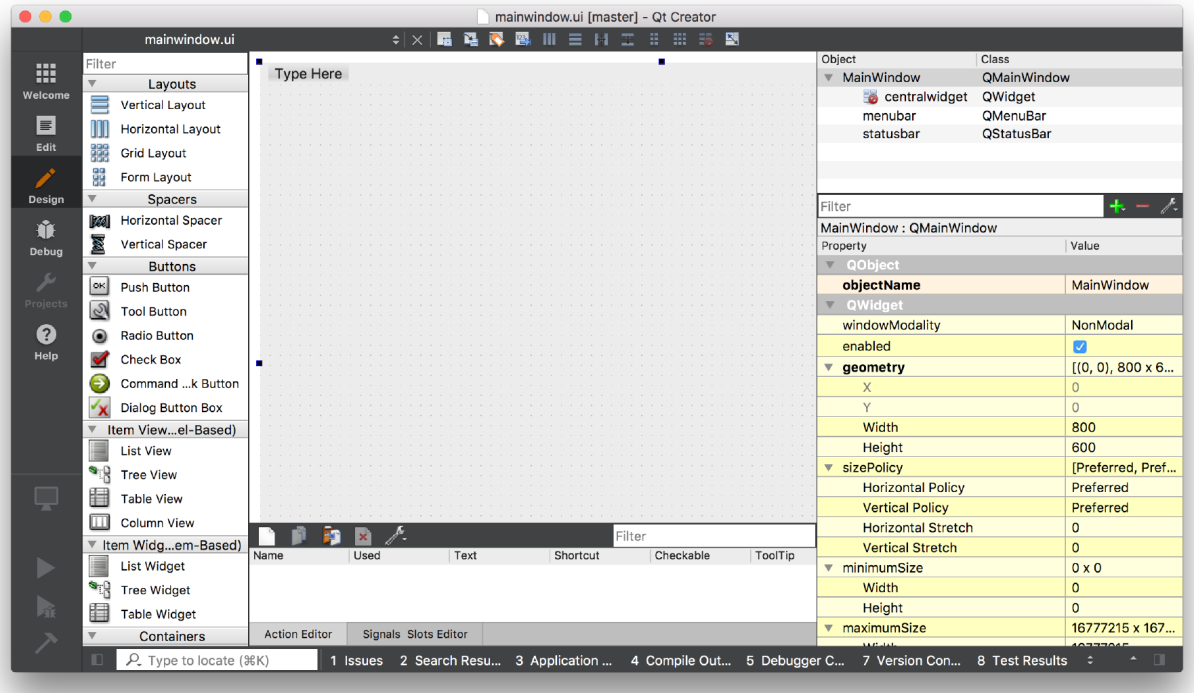
最后，如果您正在使用版本控制系统，您可以选择将文件添加到其中。您可以跳过这一步——这不会影响您的用户界面。



图八十：可以选择将文件添加到您的版本控制系统中，例如 Git。

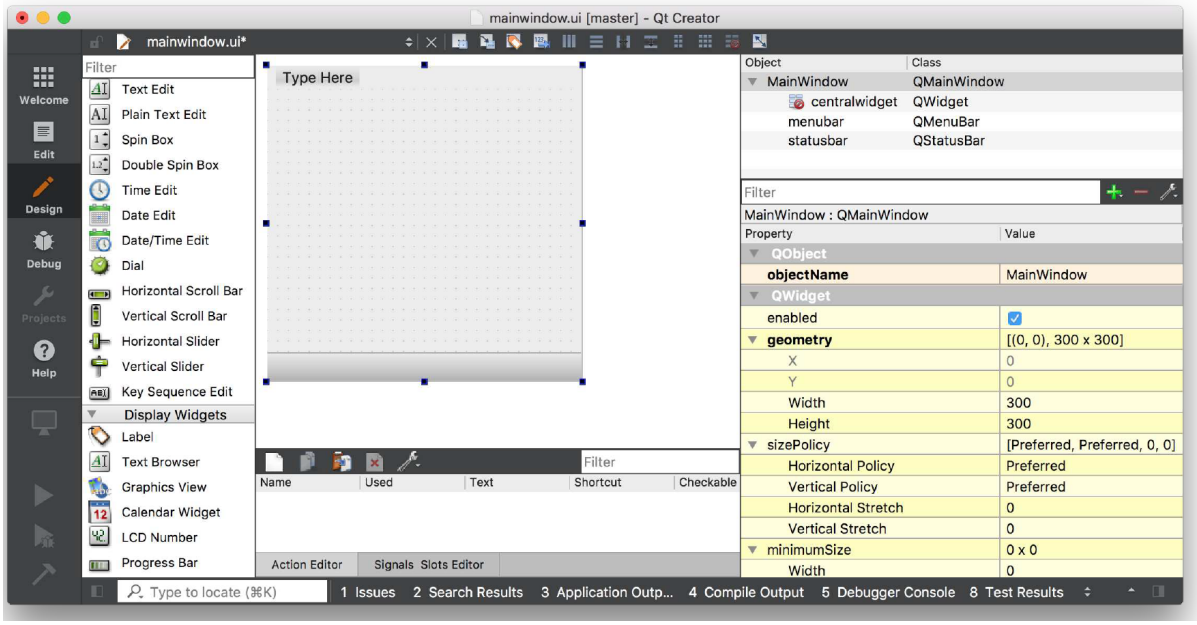
设计您的主窗口布局

您将在用户界面设计器中看到新创建的主窗口。初始状态下没什么可看的，只有一个灰色工作区代表窗口，以及窗口菜单栏的初步框架。



图八十一：创建的主窗口的初始视图。

您可以通过点击窗口并拖动每个角落的蓝色手柄来调整窗口大小。

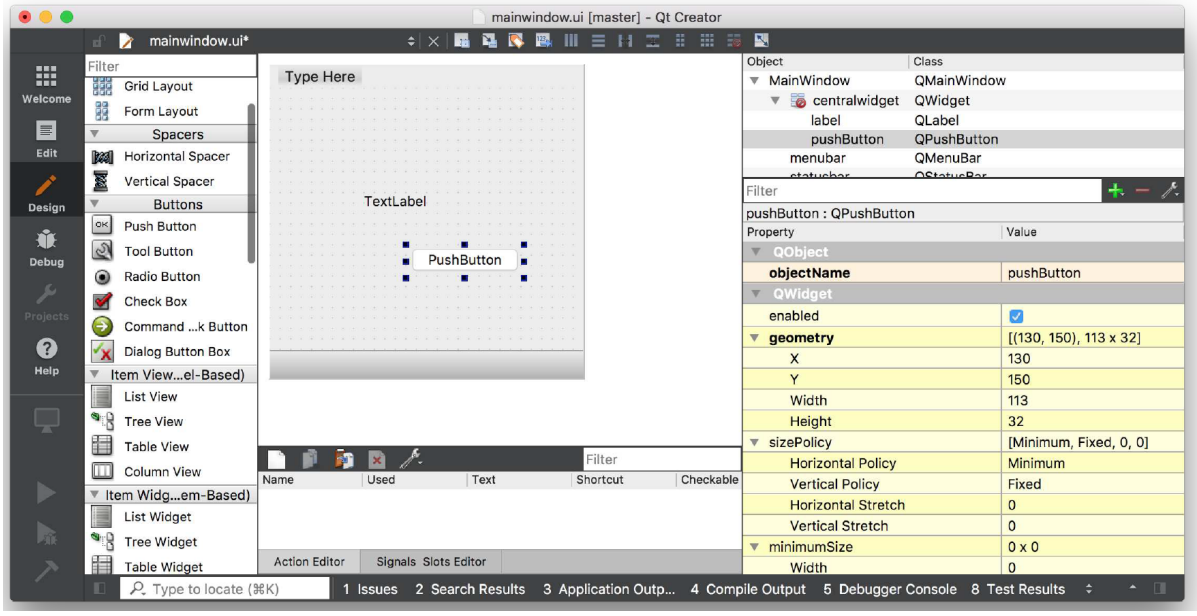


图八十二：主窗口的尺寸被调整为300×300像素。

构建应用程序的第一步是向窗口添加一些控件。在我们的第一个应用程序中，我们了解到，要设置 `QMainWindow` 的中央控件，需要使用 `.setCentralWidget()`。我们还了解到，要添加多个控件并使用布局，需要一个中间 `QWidget` 来应用布局，而不是直接将布局添加到窗口。

Qt Designer 会自动为您处理这一点，尽管它并不特别明显地显示出来。

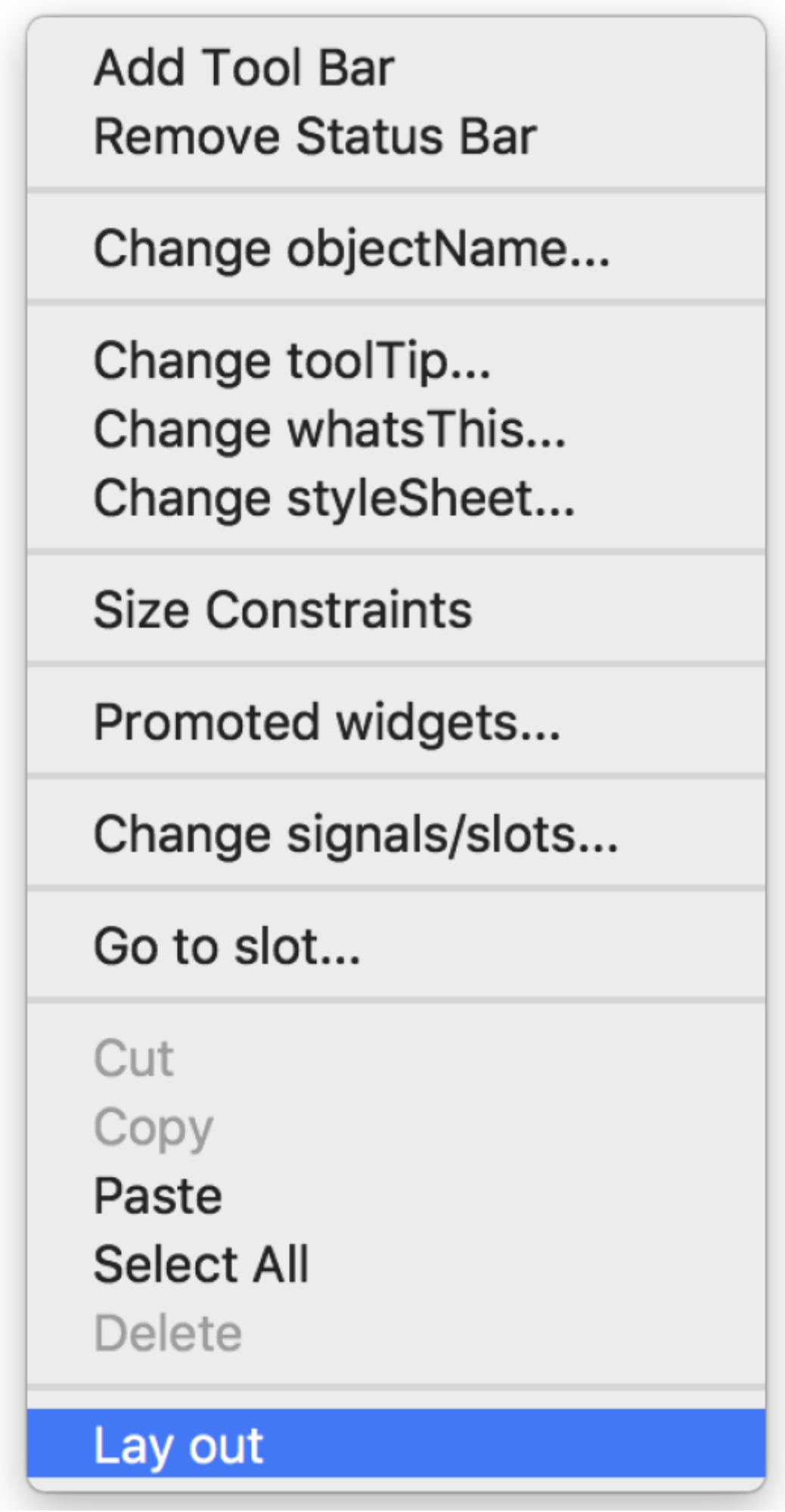
要将多个控件以布局形式添加到主窗口，首先您应该将控件拖到 `QMainWindow` 上。这里我们拖了一个 `QLabel` 和一个 `QPushButton`，将它们放在哪里并不重要。



图八十三：主窗口已添加1个标签和1个按钮。

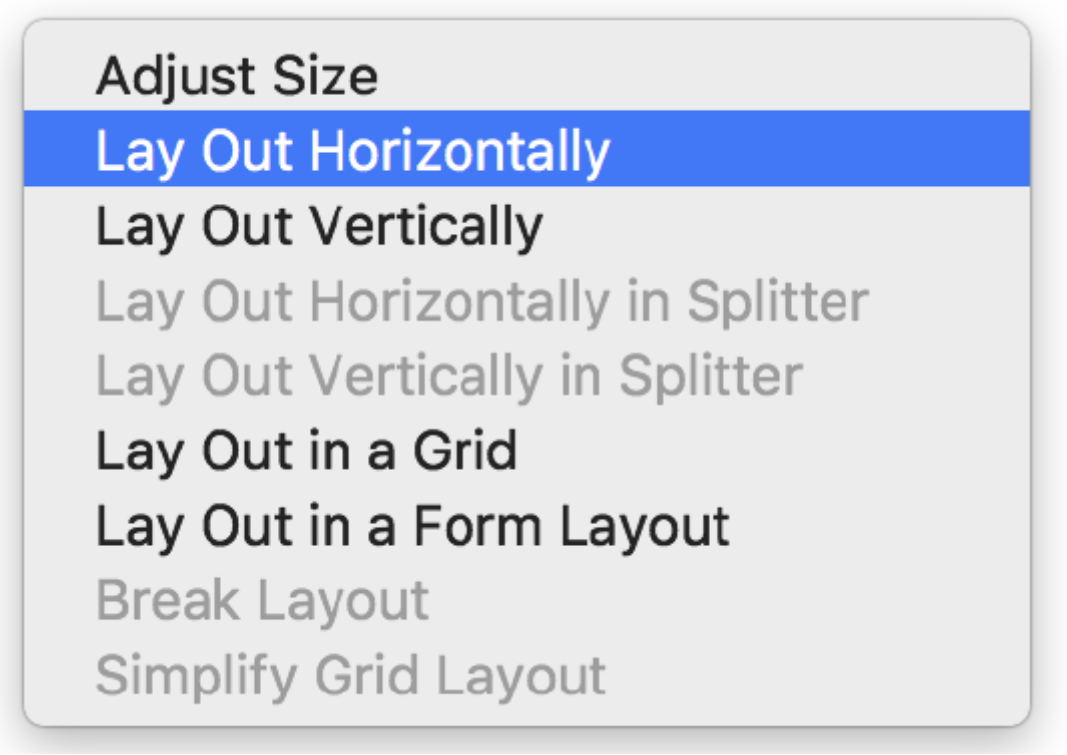
我们将 2 个控件拖到窗口中，这样它们就成为了窗口的子元素。现在，我们可以应用一个布局了。

在右侧面板中找到 `QMainWindow`（它应该位于最上方）。在其下方，您会看到代表窗口中央控件的 `centralwidget`。中央控件的图标显示了当前应用的布局。最初，该图标上有一条红色的圆圈交叉线，表明没有激活的布局。右键单击 `QMainWindow` 对象，并在弹出的下拉菜单中找到“布局”。



图八十四：请您右键点击主窗口，然后选择"lay out."。

接下来，您将看到可应用于窗口的布局列表。选择“Lay Out Horizontally”，该布局将应用于控件。

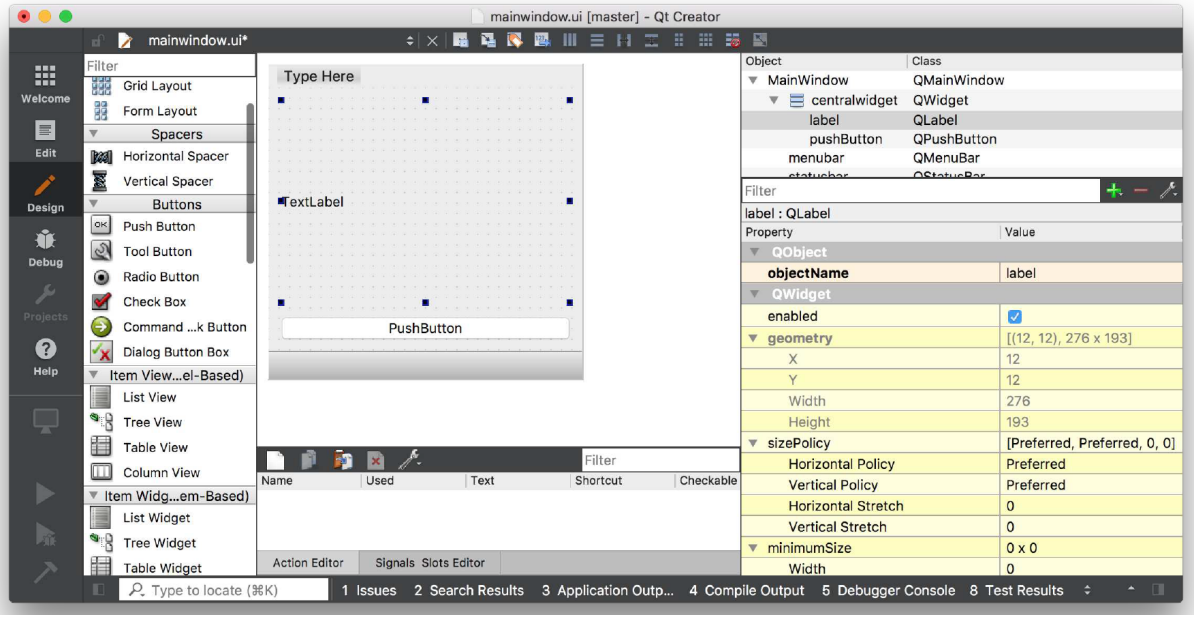


> 图八十五：选择要应用于主窗口的布局。

所选布局将应用于 `QMainWindow` 的中央控件，然后将控件添加到布局中，并根据布局进行布局。

请注意，您可以在布局中拖动和重新排列控件，它们会根据布局限制进行切换和移动。您还可以选择完全不同的布局，这对于原型设计和尝试新想法非常方便。

> 不要在没有控件的情况下尝试添加布局。布局将缩小到零大小，无法选择！



> 图八十六：垂直布局应用于主窗口上的控件。

我们已经使用Qt Designer创建了一个非常简单的用户界面。下一步是将这个用户界面集成到我们的Python代码中，并利用它来构建一个可运行的应用程序。

首先保存您的 `.ui` 文件——默认情况下，它将保存为您在创建时选择的位置，尽管您可以选择其他位置。 `.ui` 文件采用XML 格式。要在 Python 中使用我们的 UI，我们可以直接从 Python 加载它，或首先使用 `pyuic6` 工具将其转换为 Python `.py` 文件。

在 Python 中加载您的 `.ui` 文件

要加载 `.ui` 文件，我们可以使用 PyQt6 附带的 `uic` 模块，具体来说是 `uic.loadUI()` 方法。该方法接受 UI 文件的文件名，并加载它以创建一个功能完整的 PyQt6 对象。

Listing 88. designer/example_1.py

```
import os
import sys

from PyQt6 import QtWidgets, uic

basedir = os.path.dirname(__file__)

app = QtWidgets.QApplication(sys.argv)

window = uic.loadUi(os.path.join(basedir, "mainwindow.ui"))
window.show()
app.exec()
```

要从现有控件（例如 `QMainWindow`）的 `__init__` 块加载用户界面，可以使用 `uic.loadUI(filename, self)`。

Listing 89. designer/example_2.py

```
import os
import sys

from PyQt6 import QtCore, QtGui, QtWidgets, uic

basedir = os.path.dirname(__file__)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        uic.loadUi(os.path.join(basedir, "mainwindow.ui"), self)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

将您的 `.ui` 文件转换为 Python 文件

要生成 Python 输出文件，我们可以使用 PyQt6 命令行工具 `pyuic6` 。我们运行此工具时，需要传入 `.ui` 文件的文件名以及输出目标文件名，并使用 `-o` 参数。以下命令将生成一个名为 `MainWindow.py` 的 Python 文件，其中包含我们创建的 UI。我使用驼峰命名法（CamelCase）来提醒自己这是一个 PyQt6 类文件。

```
pyuic6 mainwindow.ui -o MainWindow.py
```

您可以使用编辑器打开生成的 `MainWindow.py` 文件进行查看，不过不建议您修改此文件——若您进行修改，当您通过 Qt Designer 重新生成用户界面时，所有更改都将丢失。使用 Qt Designer 的优势在于，您可以边开发边对应用程序进行编辑和更新。

构建您的应用程序

导入生成的 Python 文件与导入其他文件的方式相同。您可以按照以下方式导入您的类。`pyuic6` 工具会在 Qt Designer 中定义的对象名称前添加 `Ui_`，而您需要导入的正是这个对象。

```
from MainWindow import Ui_MainWindow
```

要创建应用程序的主窗口，请像往常一样创建一个类，但需要同时继承自 `QMainWindow` 和您导入的 `Ui_MainWindow` 类。最后，在 `__init__` 方法中调用 `self.setupUi(self)` 以触发界面设置。

就是这样。您的窗口现已完全设置完毕。

添加应用程序逻辑

您可以像使用代码创建的控件一样，与通过 Qt Designer 创建的控件进行交互。为了简化操作，`pyuic6` 将所有控件都添加到窗口对象中。

> 对象的名称可以通过Qt Designer找到。您只需在编辑器窗口中单击该对象，然后在属性面板中查找 `objectName` 。

在下面的示例中，我们将使用生成的主窗口类构建一个可工作的应用程序。

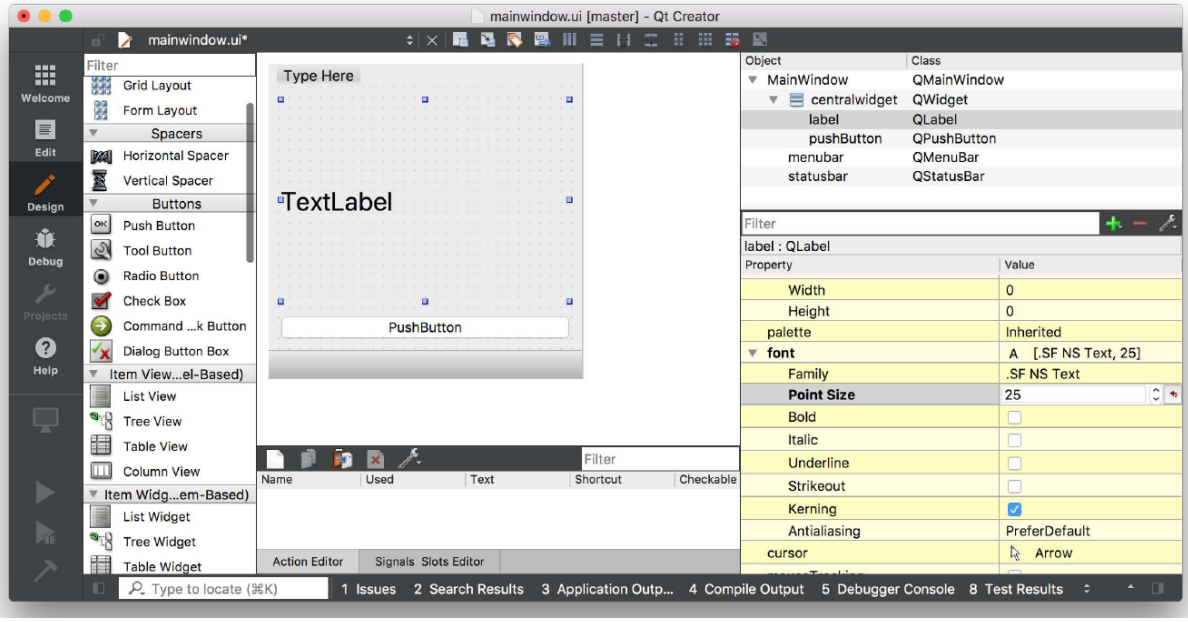
Listing 90. designer/compiled_example.py

```
import random
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QMainWindow

from MainWindow import Ui_MainWindow
```

```
class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.show()

        # 您仍然可以在代码中覆盖 UI 文件中的值
        # 如果可能的话，请在 Qt Creator 中进行设置。请查看属性面板
        f = self.label.font()
        f.setPointSize(25)
        self.label.setAlignment(
            Qt.AlignmentFlag.AlignHCenter
            | Qt.AlignmentFlag.AlignVCenter
        )
        self.label.setFont(f)

        # UI 控件的信号可以照常连接。
        self.pushButton.pressed.connect(self.update_label)

    def update_label(self):
        n = random.randint(1, 6)
        self.label.setText("%d" % n)


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

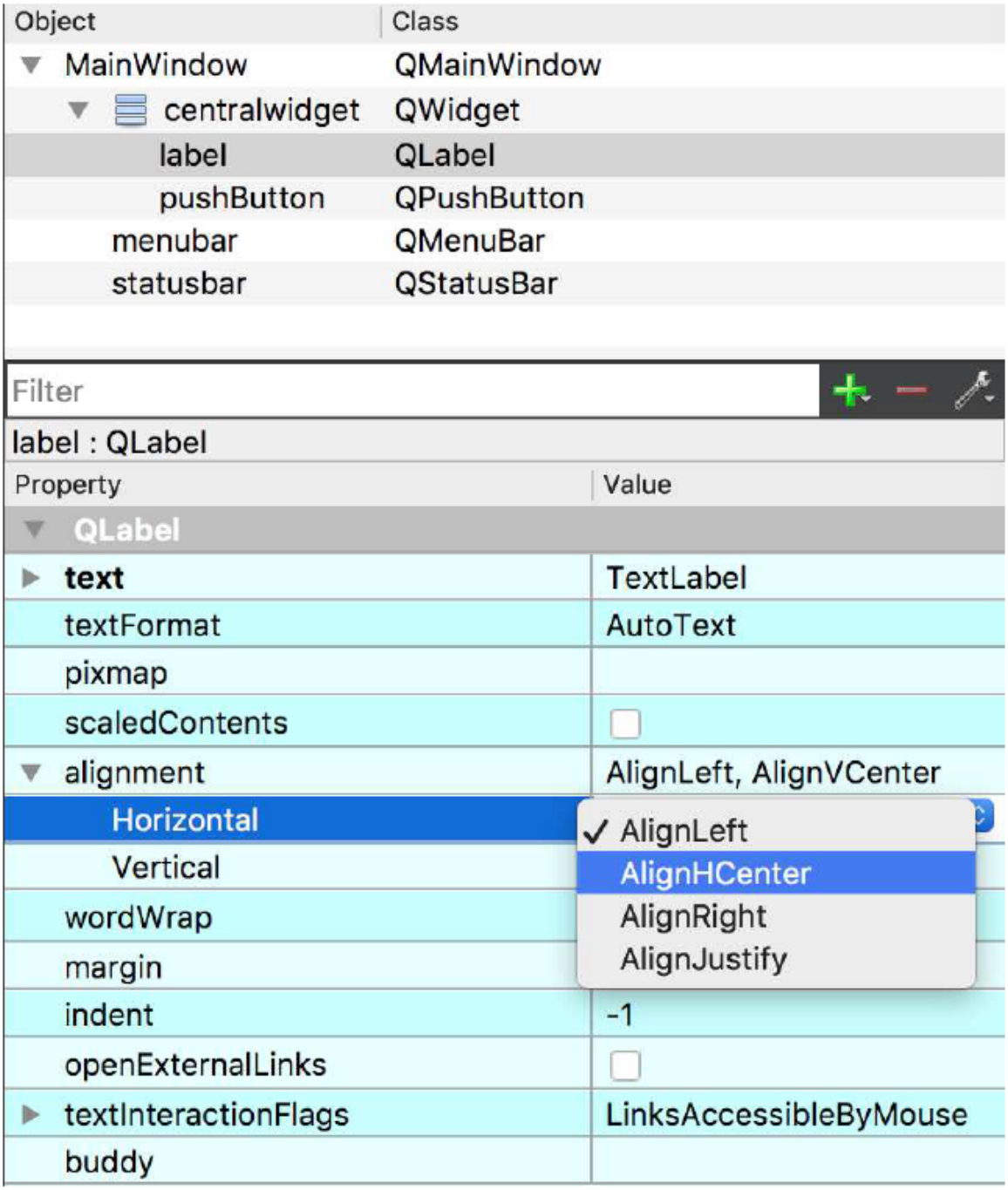
请注意，由于我们在 Qt Designer `.ui` 定义中未设置字体大小和对齐方式，因此必须使用代码手动设置。您可以像之前一样，通过这种方式更改任何控件参数。但是，通常最好在 Qt Designer 本身中配置这些内容。

您可以通过窗口右下角的属性面板设置任何控件属性。大多数控件属性都显示在此处，例如，下面我们正在更新 `QLabel` 控件上的字体大小——



> 图八十七：设置QLabel的字体大小。

您还可以配置对齐方式。对于复合属性（您可以设置多个值，例如左对齐 + 中间对齐），它们是嵌套的。



> 图八十八：详细字体属性。

所有对象属性均可在两个地方进行编辑——具体选择权在您手中，您可以选择在代码中进行特定修改，或在Qt Designer中进行修改。作为一般原则，建议将动态更改保留在代码中，而将基础或默认状态保留在设计的用户界面中。

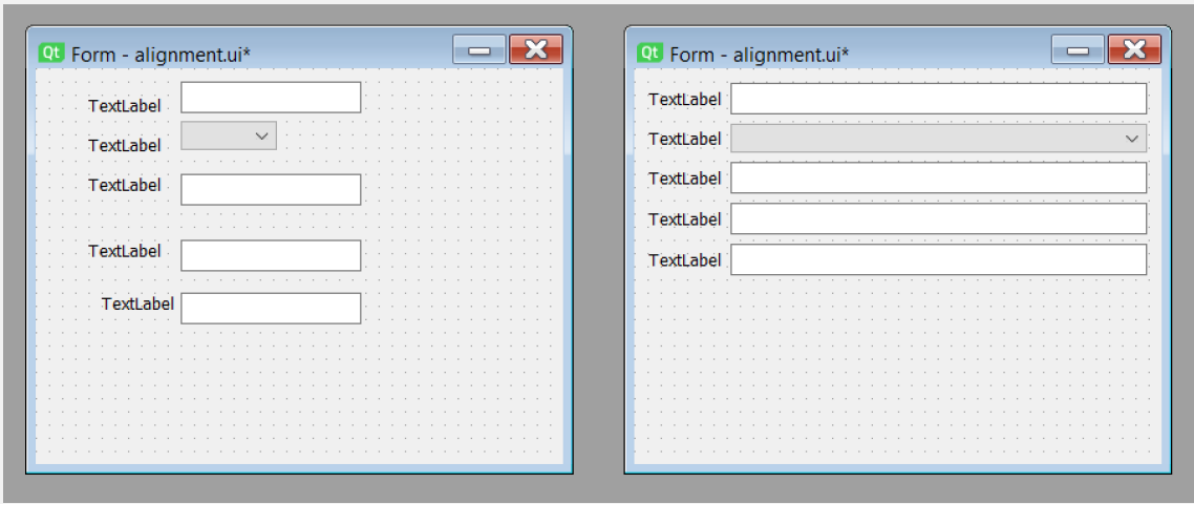
本介绍仅涉及 Qt Designer 功能的一小部分。我强烈建议您深入探索并尝试——请记住，您仍然可以在之后通过代码添加或调整控件。

美学

如果您不是设计师，创建吸引人且直观的界面可能会很困难，甚至您可能不知道它们是什么。幸运的是，有一些简单的规则您可以遵循来创建界面，即使它们不一定漂亮，至少不会难看。关键概念是——对齐、分组和空间。

对齐是为了减少视觉噪声。将控件的角视为对齐点，并尽量减少用户界面中不和谐的对齐点数量。实际上，这意味着确保界面中元素的边缘相互对齐。

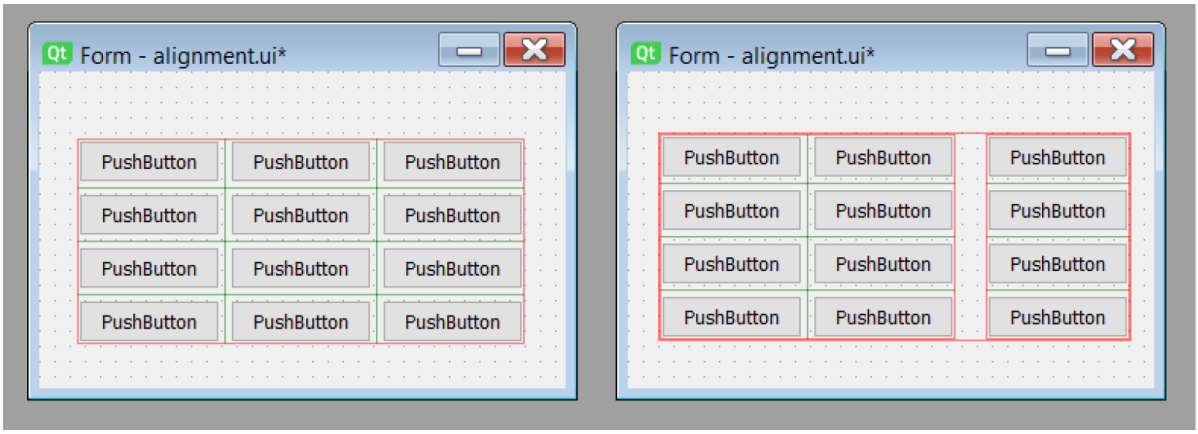
如果输入大小不同，请您将它们与读取的边缘对齐。



> 对齐对界面清晰度的影响

英语是左至右书写的语言，因此如果您的应用程序使用英语，请将文本对齐到左侧。

相关控件**组**会获得上下文，这样会使其更易于理解。所以，在您构建界面时，请将相关元素放在一起。



> 分组元素并在组之间添加间隔

空间是创建界面中视觉上区分的区域的关键——如果没有空间来分隔组件，就无法形成组件！请您务必保持间距的一致性和合理性。

请务必使用对齐功能来减少界面中的视觉干扰。

请务必将相关的控件分组到逻辑集里。

请务必在组之间添加一致的间距，以明确结构。

主题设计

开箱即用的 Qt 应用程序看起来像平台原生应用程序。也就是说，它们采用的是运行所在操作系统的界面风格。这意味着它们在任何系统上都看起来很自然，可以给我们的用户带来自然的使用体验。但这也意味着它们看起来有些乏味。幸运的是，Qt 允许您完全控制应用程序中控件的外观。

无论您是想让您的应用程序脱颖而出，还是正在设计自定义控件并希望它们与应用程序相融合，本章我们将介绍如何在 PyQt6 中实现这些目标。

13. 样式

样式是 Qt 对应用程序进行广泛外观和感觉更改、修改控件显示和行为的方式。Qt 在应用程序在特定平台上运行时会自动应用特定于特定平台的样式，因此，在 macOS 上运行时，您的应用程序看起来像一个 macOS 应用程序，而在 Windows 上运行时，看起来像一个 Windows 应用程序。这些特定于平台的样式利用了主机平台上的本机控件，这意味着它们不能在其他平台上使用。

然而，平台样式并不是您为应用程序设计样式时唯一可选的选项。Qt 还附带了一个名为 Fusion 的跨平台样式，该样式为您的应用程序提供了一致的跨平台、现代的样式。

Fusion

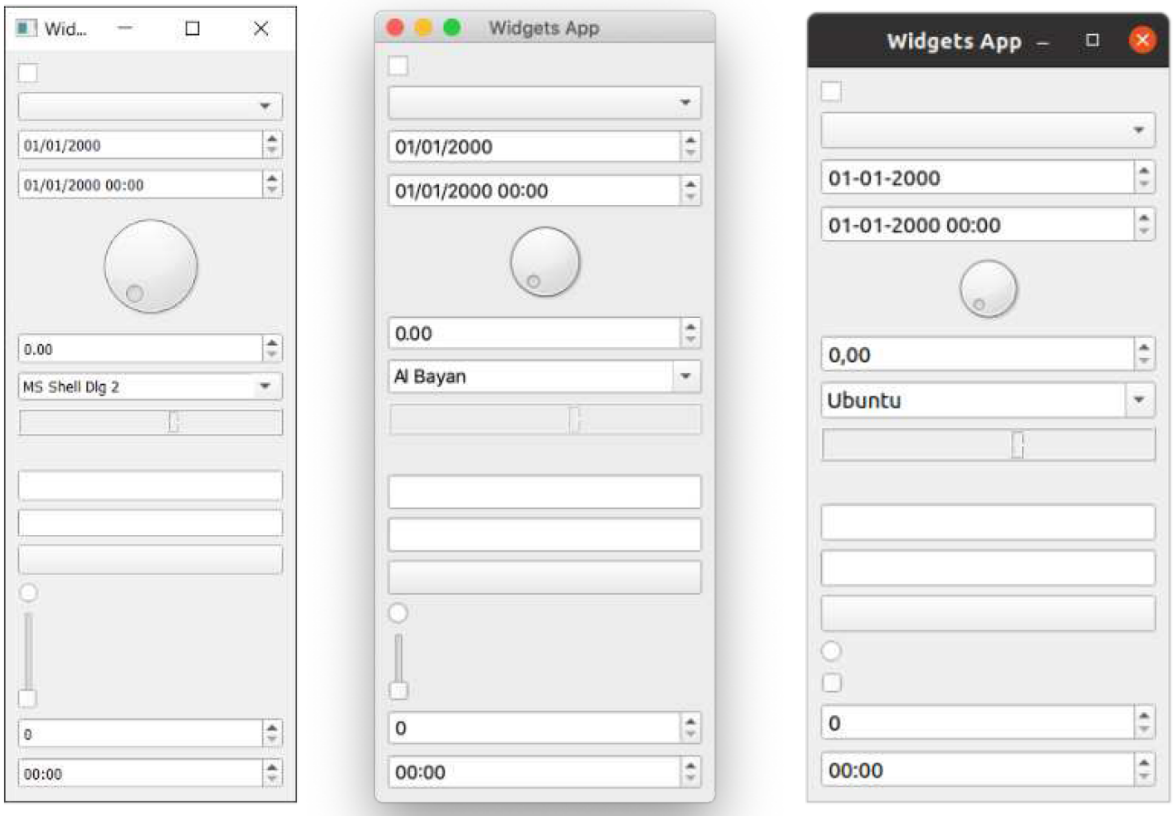
Qt 的 Fusion 样式为您带来了所有系统 UI 一致性的优势，但代价是与操作系统标准的一致性受到了一些影响。哪一个更重要，取决于您对正在创建的 UI 需要多少控制权、您对其进行了多少自定义以及您使用了哪些控件。

> Fusion 样式是一种与平台无关的样式，提供桌面化的外观和感觉。它实现了与 Qt 控件的 Fusion 样式相同的设计语言。
>
> ——Qt官方文档

要启用该样式，请在 `QApplication` 实例上调用 `.setStyle()`，并将样式名称（在本例中为 Fusion）作为字符串传递。

```
app = QApplication(sys.argv)
app.setStyle('Fusion')
#...
app.exec()
```

下面显示了前面提到的控件列表示例，但应用了 Fusion 样式。



> 图八十九："Fusion"样式控件。它们在所有平台上看起来完全相同。

> 在 Qt 文档中还有更多应用了 Fusion 样式的控件示例。

14. 调色板

在 Qt 中用于绘制用户界面的颜色选择被称为调色板。应用程序级和控件特定的调色板都通过 QPalette 对象进行管理。调色板可以在应用程序和控件级别设置，允许您设置全局标准调色板，并根据每个控件的情况进行覆盖。全局调色板通常由 Qt 主题（通常依赖于操作系统）定义，但您可以覆盖它来更改整个应用程序的外观。

活动全局调色板可通过 `QApplication.palette()` 方法或创建一个新的空 `QPalette` 实例来访问。例如——

```
from PyQt6.QtGui import QPalette
palette = QPalette()
```

您可以通过调用 `palette.setColor(role, color)` 来修改调色板，其中 `role` 决定了颜色的用途，`QColor` 决定了要使用的颜色。使用的颜色可以是自定义的 `QColor` 对象，也可以是 `Qt.GlobalColor` 命名空间中的内置基本颜色之一。

```
palette.setColor(QPalette.ColorRole.Window, QColor(53,53,53))
palette.setColor(QPalette.ColorRole.WindowText, Qt.GlobalColor.white)
```

> 在使用调色板时，Windows 10和 macOS 平台的特定主题存在一些限制。

角色(role)种类相当多。主要角色如下表所示——

Table 4. Main roles

常量	值	描述
`QPalette.ColorRole.Window`	10	窗口的背景颜色
`QPalette.ColorRole.WindowText`	0	窗口的默认文本颜色
`QPalette.ColorRole.Base`	9	文本输入控件、组合框下拉列表和工具栏句柄的背景。通常为白色或浅色
`QPalette.ColorRole.AlternateBase`	16	第二种颜色，用于条纹（交替）行——例如 `QAbstractItemView.setAlternatingRowColors()`

常量	值	描述
`QPalette.ColorRole.ToolTipBase`	18	`QToolTip` 和 `QWhatsThis` 悬停指示器的背景颜色。这两个提示均使用“非活动组”（见后文），因为它们不是活动窗口。
`QPalette.ColorRole.ToolTipText`	19	`QToolTip` 和 `QWhatsThis` 的前景色。这两个提示均使用“非活动”组（见后文），因为它们不是活动窗口。
`QPalette.ColorRole.PlaceholderText`	20	控件中占位符文本的颜色。
`QPalette.ColorRole.Text`	6	使用 `Base` 颜色为控件设置文本颜色。必须与 Window 和 Base 形成良好的对比度。
`QPalette.ColorRole.Button`	1	默认按钮背景颜色。此颜色可能与窗口颜色不同，但必须与按钮文字形成良好对比。
`QPalette.ColorRole.ButtonText`	8	按钮上使用的文字颜色，必须与按钮颜色形成足够的对比度。
`QPalette.ColorRole.BrightText`	7	与WindowText颜色差异显著，与黑色形成良好对比的文本颜色。在其他文本和WindowText颜色会导致对比度较差的情况下使用。 注意：不仅可以用于文本。

> 您不必在自定义调色板中修改或设置所有这些选项，这取决于应用程序中使用的控件，有些可以省略。

还有一些较小的角色集，用于控件的 3D 斜角和突出显示选定的条目或链接。

Table 5. 3D bevel roles

常量	值	描述
`QPalette.ColorRole.Light`	2	比 `Button` 颜色更浅
`QPalette.ColorRole.Midlight`	3	颜色深浅在 `Button` 和 `Light` 之间
`QPalette.ColorRole.Dark`	4	比 `Button` 颜色更深
`QPalette.ColorRole.Mid`	5	颜色深浅在 `Button` 和 `Dark` 之间
`QPalette.ColorRole.Shadow`	11	一种非常深的颜色。在默认情况下，阴影颜色为 `Qt.GlobalColor.black`

Table 6. Highlighting & links

常量	值	描述
`QPalette.ColorRole.Highlight`	12	用于指示选中项或当前项的颜色。默认情况下，高亮颜色为 `Qt.GlobalColor.darkBlue`

常量	值	描述
`QPalette.ColorRole.HighlightedText`	13	与高亮文本形成对比的文本颜色。默认情况下，高亮文本的颜色为 `Qt.GlobalColor.white`
`QPalette.ColorRole.Link`	14	未访问超链接的文本颜色。默认情况下，链接颜色为 `Qt.GlobalColor.blue`
`QPalette.ColorRole.LinkVisited`	15	已访问超链接的文本颜色。默认情况下，已访问链接的颜色为 `Qt.GlobalColor.magenta`

> 从技术上讲，对于未分配角色的控件绘制状态，还有一个`QPalette.NoRole`值，在创建调色板时可以忽略它。

对于用户界面中在控件处于活动、非活动或禁用状态时会发生变化的部分，您必须为每个状态设置颜色。要实现这一点，您可以调用 `palette.setColor(group, role, color)` 方法，并传入额外的 `group` 参数。可用的组如下所示

常量	值
`QPalette.ColorGroup.Disabled`	1
`QPalette.ColorGroup.Active`	0
`QPalette.ColorGroup.Inactive`	2
`QPalette.ColorGroup.Normal` "Active"的同义词	0

例如，以下代码将禁用窗口的窗口文本颜色设置为调色板中的白色。

```
palette.setColor(QPalette.ColorGroup.Disabled, QPalette.ColorRole.WindowText,
Qt.GlobalColor.white)
```

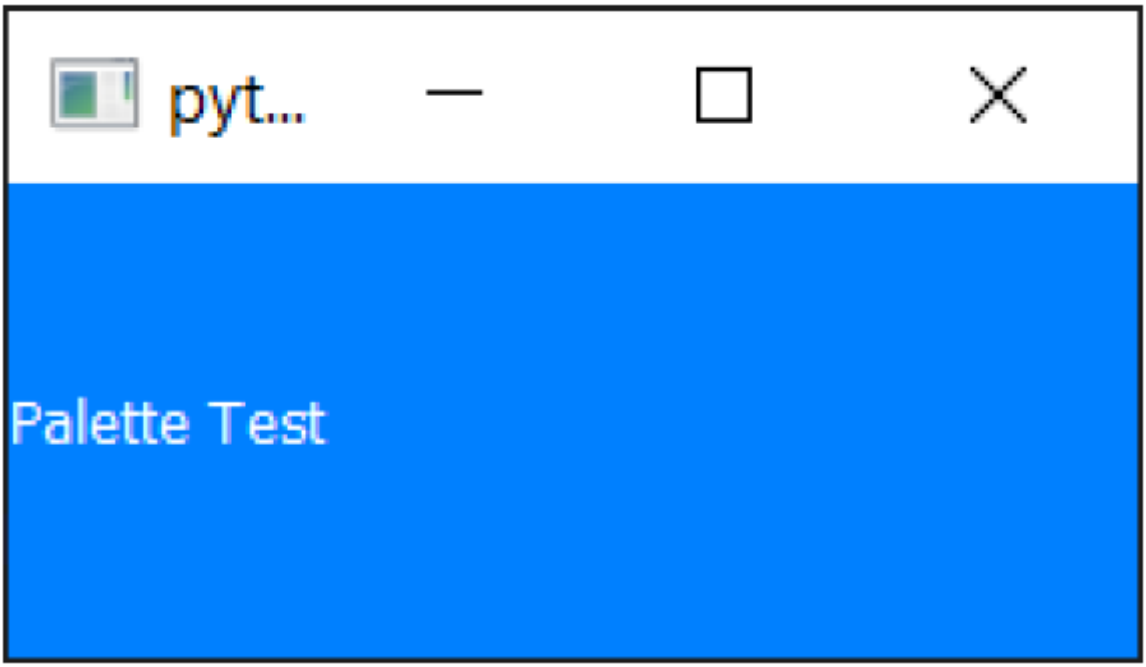
一旦调色板被定义，您可以使用 `.setPalette()` 将它设置到 `QApplication` 对象上，以将其应用到您的应用程序或单个控件上。例如，以下示例将更改窗口文本和背景的颜色（这里使用 `QLabel` 添加了文本）。

Listing 91. themes/palette_test.py

```
from PyQt6.QtWidgets import QApplication, QLabel
from PyQt6.QtGui import QPalette, QColor
from PyQt6.QtCore import Qt

import sys
```

```
app = QApplication(sys.argv)
palette = QPalette()
palette.setColor(QPalette.ColorRole.Window, QColor(0, 128, 255))
palette.setColor(QPalette.ColorRole.WindowText, Qt.GlobalColor.white)
app.setPalette(palette)

w = QLabel("Palette Test")
w.show()

app.exec()
```

运行时，会输出以下内容。窗口背景色变为浅蓝色，窗口文字为白色。



> 图九十：更改窗口和窗口文本的颜色

为了展示配色方案在实际中的应用并了解其局限性，我们将创建一个使用自定义深色配色方案的应用程序。

> 使用此调色板，所有控件都将以深色背景绘制，无论应用程序的深色模式状态如何。有关使用系统深色模式的信息，请参见后面部分。

虽然一般情况下应避免覆盖用户设置，但在某些类型的应用程序中，如照片查看器或视频编辑器，使用明亮的用户界面可能会干扰用户对颜色的判断。以下应用程序骨架使用了 Jürgen Skrotzky 定制的配色方案，为应用程序提供全局深色主题。

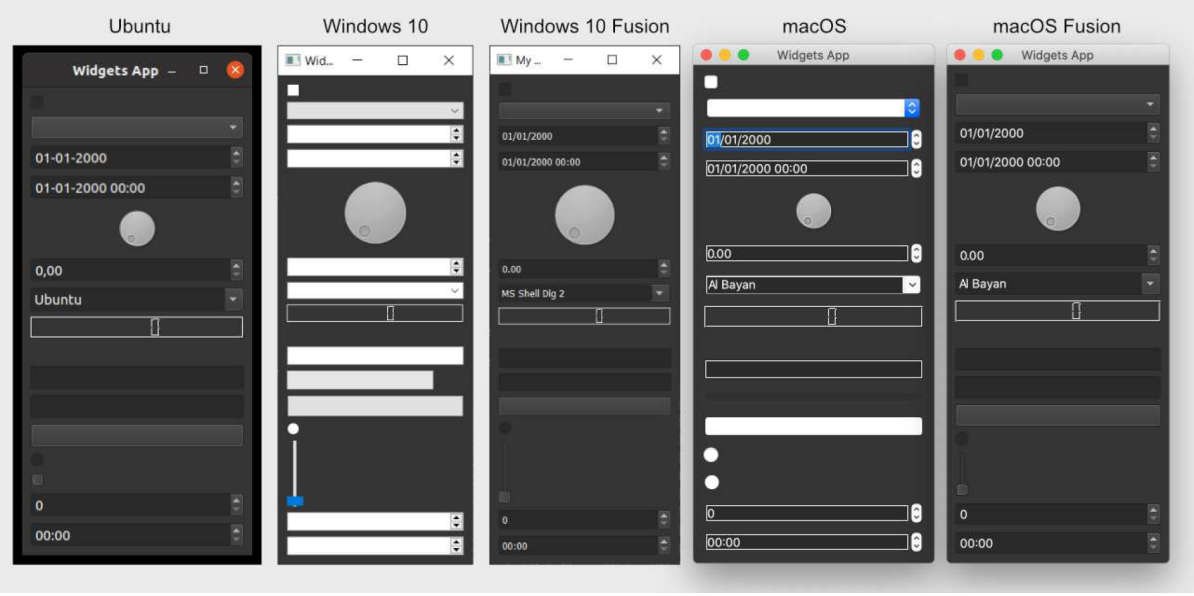
```
from PyQt6.QtWidgets import QApplication, QMainWindow
from PyQt6.QtGui import QPalette, QColor
```

```
from PyQt6.QtCore import Qt

import sys

darkPalette = QPalette()
darkPalette.setColor(QPalette.ColorRole.Window, QColor(53, 53, 53))
darkPalette.setColor(
    QPalette.ColorRole.WindowText, Qt.GlobalColor.white
)
darkPalette.setColor(
    QPalette.ColorGroup.Disabled,
    QPalette.ColorRole.WindowText,
    QColor(127, 127, 127),
)
darkPalette.setColor(QPalette.ColorRole.Base, QColor(42, 42, 42))
darkPalette.setColor(
    QPalette.ColorRole.AlternateBase, QColor(66, 66, 66)
)
darkPalette.setColor(
    QPalette.ColorRole.ToolTipBase, Qt.GlobalColor.white
)
darkPalette.setColor(
    QPalette.ColorRole.ToolTipText, Qt.GlobalColor.white
)
darkPalette.setColor(QPalette.ColorRole.Text, Qt.GlobalColor.white)
221
darkPalette.setColor(
    QPalette.ColorGroup.Disabled,
    QPalette.ColorRole.Text,
    QColor(127, 127, 127),
)
darkPalette.setColor(QPalette.ColorRole.Dark, QColor(35, 35, 35))
darkPalette.setColor(QPalette.ColorRole.Shadow, QColor(20, 20, 20))
darkPalette.setColor(QPalette.ColorRole.Button, QColor(53, 53, 53))
darkPalette.setColor(
    QPalette.ColorRole.ButtonText, Qt.GlobalColor.white
)
darkPalette.setColor(
    QPalette.ColorGroup.Disabled,
    QPalette.ColorRole.ButtonText,
    QColor(127, 127, 127),
)
darkPalette.setColor(QPalette.ColorRole.BrightText, Qt.GlobalColor.
                     red)
darkPalette.setColor(QPalette.ColorRole.Link, QColor(42, 130, 218))
darkPalette.setColor(QPalette.ColorRole.Highlight, QColor(42, 130,
                                                          218))
darkPalette.setColor(
    QPalette.ColorGroup.Disabled,
    QPalette.ColorRole.Highlight,
    QColor(80, 80, 80),
)
darkPalette.setColor(
    QPalette.ColorRole.HighlightedText, Qt.GlobalColor.white
)
darkPalette.setColor(
```

```
    QPalette.ColorGroup.Disabled,
    QPalette.ColorRole.HighlightedText,
    QColor(127, 127, 127),
)

app = QApplication(sys.argv)
app.setPalette(darkPalette)

w = QMainWindow() # 用您的 QMainWindow 实例替换此处。
w.show()

app.exec()
```

与之前一样，调色板构建完成后，必须应用才能生效。这里，我们通过调用 `app.setPalette()` 将它应用到整个应用程序。所有控件都将采用该主题。您可以使用此框架构建自己的应用程序。

在本书的代码示例中，您还可以找到 `themes/palette_dark_widgets.py` ，它使用此调色板再现了控件演示。每个平台上的结果如下所示。



> 图九十一：适用于不同平台和主题的自定义深色配色方案。

您会注意到，使用默认的 Windows 和 macOS 主题时，某些控件的颜色无法正确应用。这是因为这些主题使用平台本机控件来提供真正的本机体验。如果您想在 Windows 10 上使用深色或高度自定义的主题，建议在这些平台上使用 Fusion 样式。

深色模式

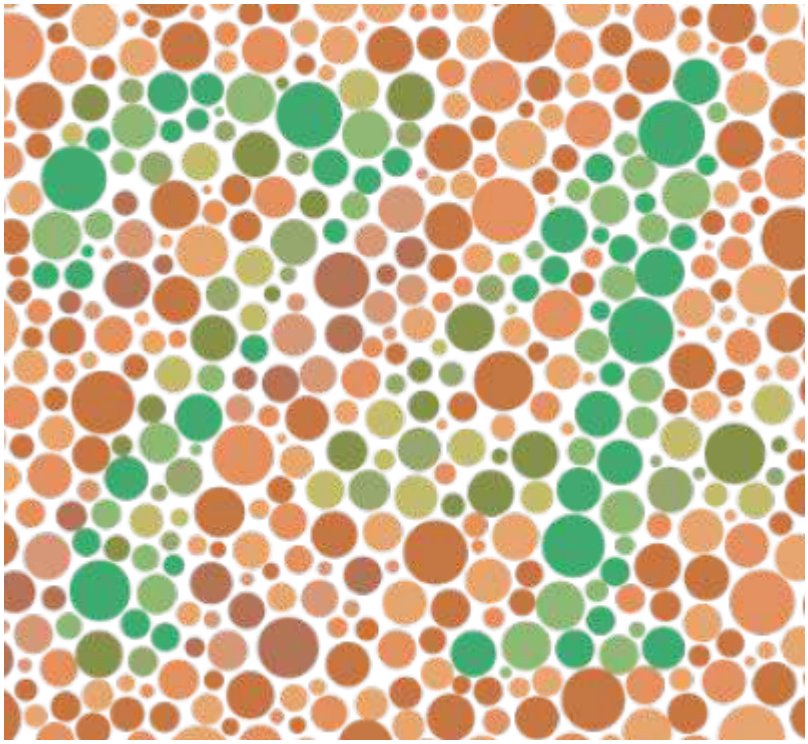
深色主题的操作系统和应用程序有助于减轻眼睛疲劳并减少睡眠干扰，尤其是在晚上工作时。Windows、macOS 和 Linux 均支持深色模式主题，而好消息是，如果您使用 PyQt6 开发应用程序，深色模式支持将自动包含在内。

可访问的颜色

当您开始开发自己的应用程序时，可能会忍不住想在设计中调整颜色——但请稍等！您的操作系统应该有一个标准主题，大多数软件都会遵循这个主题。Qt会自动识别这个颜色方案，并将其应用到您的应用程序中，帮助它们与系统风格保持一致。使用这些颜色有以下优势——

1. 您的应用程序在用户的桌面上将显得非常自然
2. 您的用户已经熟悉上下文颜色所代表的含义

3. 有人已经花时间设计了能够有效工作的颜色方案

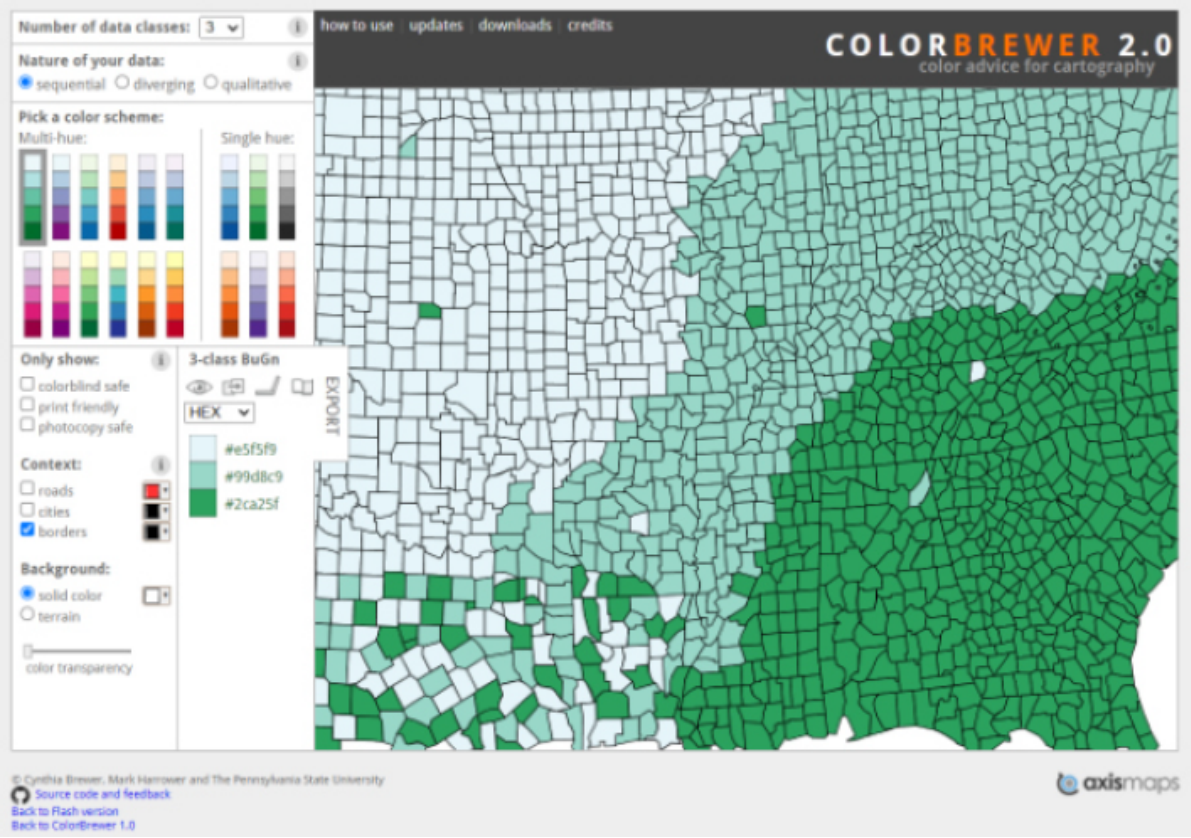


> **不要低估#3的重要性！**设计良好的色彩方案非常困难，尤其是当您考虑无障碍问题时——而您应该考虑！

如果您想替换默认桌面配色方案，请确保收益大于成本，并且您已探索过其他选项，例如目标平台上内置的深色模式。

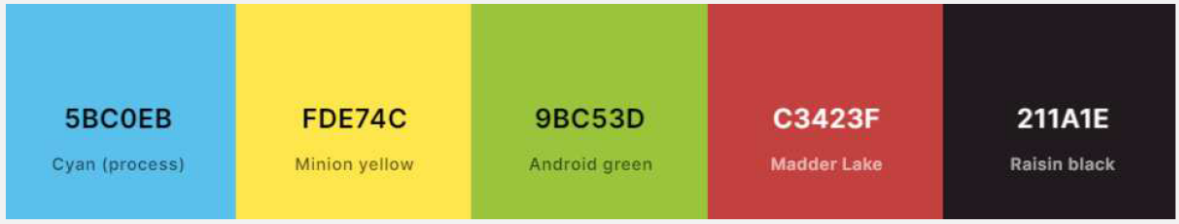
对于**数据可视化**应用，我推荐使用 Cynthia Brewer 的 Color Brewer颜色方案，这些方案兼具定性与定量方案，并专为最大程度的清晰度而设计。

对于**上下文颜色和高亮显示**，或任何其他只需少量颜色的情况——例如状态指示器——coolors.co 网站允许您生成自定义的、协调良好的 4 色主题。



> Colorbrewer2.org 提供定量和定性色彩方案

请充分利用您的配色方案。简单而有效地使用颜色，尽可能限制您的配色方案。如果某些颜色在某个地方有特定含义，那么在所有地方都使用相同的含义。避免使用多种色调，除非这些色调有特定含义。



> 来自 coolors.co 的示例配色方案

请务必在应用程序中使用图形用户界面标准颜色。

使用自定义颜色时，**请务必**定义一种配色方案并坚持使用。

选择颜色和对比度时，**请务必**考虑色盲用户。

请勿将标准颜色用于非标准用途，例如红色代表“OK”。

15. 图标

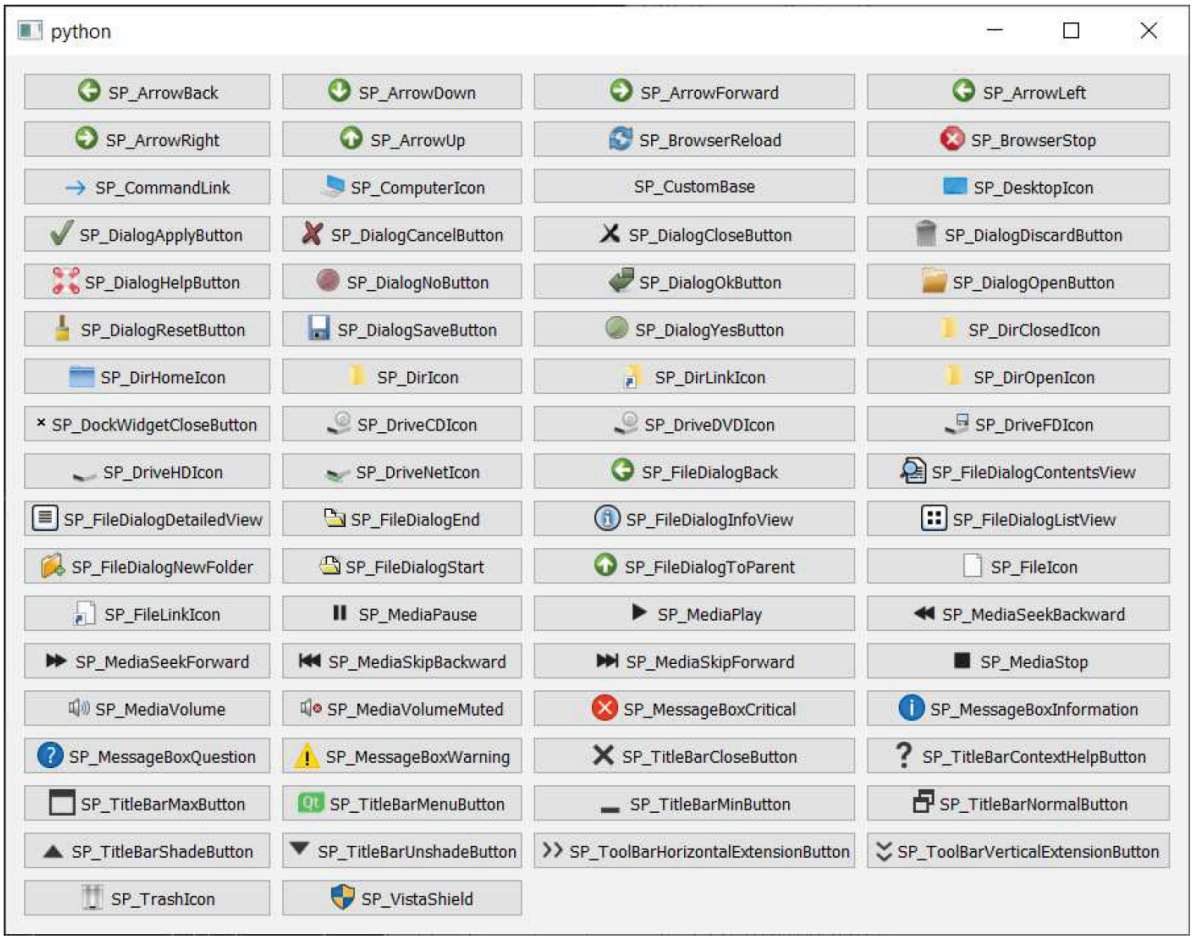
图标是用于辅助导航或理解的小型图片。它们通常出现在按钮上，可以与文本并列显示，或替代文本，也可以与菜单中的操作并列显示。通过使用易于识别的指示符，您可以使界面更易于使用。

在 PyQt6 中，您有多种不同的方式来获取并集成图标到应用程序中。本节将介绍这些选项以及各自的优缺点。

Qt 标准图标

在应用程序中添加简单图标的最简单方法是使用Qt自带的内置图标。这组图标涵盖了多种标准使用场景，包括文件操作、前后箭头以及消息框指示器。

以下是内置图标的完整列表。



> 图九十二：Qt的内置图标

您会发现这组图标的适用范围较为有限。如果这对您正在开发的应用程序来说不是问题，或者您只需要为应用程序使用少量图标，那么这仍然可能是一个可行的选择。

通过当前应用程序样式使用 `QStyle.standardIcon(name)` 或 `QStyle.<constant>` 可以访问这些图标。内置图标名称的完整列表如下所示。

SP_ArrowBack	**SP_DirIcon**	**SP_MediaSkipBackward**
SP_ArrowDown	SP_DirLinkIcon	SP_MediaSkipForward
SP_ArrowForward	SP_DirOpenIcon	SP_MediaStop
SP_ArrowLeft	SP_DockWidgetCloseButton	SP_MediaVolume
SP_ArrowRight	SP_DriveCDIcon	SP_MediaVolumeMuted
SP_ArrowUp	SP_DriveDVDIcon	SP_MessageBoxCritical
SP_BrowserReload	SP_DriveFDIcon	SP_MessageBoxInformation
SP_BrowserStop	SP_DriveHDIcon	SP_MessageBoxQuestion
SP_CommandLink	SP_DriveNetIcon	SP_MessageBoxWarning
SP_CustomBase	SP_FileDialogContentsView	SP_TitleBarContextHelpButton
SP_DesktopIcon	SP_FileDialogDetailedView	SP_TitleBarMaxButton
SP_DialogApplyButton	SP_FileDialogEnd	SP_TitleBarMenuButton
SP_DialogCancelButton	SP_FileDialogInfoView	SP_TitleBarMinButton
SP_DialogCloseButton	SP_FileDialogListView	SP_TitleBarNormalButton

SP_ArrowBack	SP_DirIcon	SP_MediaSkipBackward
SP_DialogDiscardButton	SP_FileDialogNewFolder	SP_TitleBarShadeButton
SP_DialogHelpButton	SP_FileDialogStart	SP_TitleBarUnshadeButton
SP_DialogNoButton	SP_FileDialogToParent	SP_ToolBarHorizontalExtensionButton
SP_DialogOkButton	SP_FileIcon	SP_ToolBarVerticalExtensionButton
SP_DialogResetButton	SP_FileLinkIcon	SP_TrashIcon
SP_DialogSaveButton	SP_MediaPause	SP_VistaShield
SP_DialogYesButton	SP_MediaPlay	SP_DirClosedIcon
SP_MediaSeekBackward	SP_DirHomeIcon	SP_MediaSeekForward

您可以通过以下方式直接通过 `QStyle` 命名空间访问这些图标。

```
icon = QStyle.standardIcon(QStyle.SP_MessageBoxCritical)
button.setIcon(icon)
```

您也可以使用特定控件的样式对象。无论使用哪一个都无所谓，因为我们只是访问内置控件而已。

```
style = button.style() # 从控件中获取 QStyle 对象
icon = style.standardIcon(style.SP_MessageBoxCritical)
button.setIcon(icon)
```

如果您在标准图标集找不到所需的图标，您需要使用以下所述的其他方法之一。

> 虽然您可以混合搭配不同图标集中的图标，但最好在整个应用程序中使用单一样式，以保持应用程序的连贯性。

图标文件

如果标准图标不符合您的需求，或者您需要一些不可用的图标，您可以使用任何自定义图标。图标可以是您平台上 Qt 支持的任何图像类型，尽管对于大多数使用场景，PNG 或 SVG 图像更可取。

> 要获取您平台上支持的图像格式列表，您可以调用 `QtGui.QImageReader.supportedImageFormats()` 方法。

图标集

如果您不是图形设计师，使用现成的图标集可以节省大量时间（和麻烦）。网上有成千上万种这样的图标集，其许可证因在开源软件或商业软件中的使用而异。

在这本书和示例应用程序中，我使用了 Fugue 图标集，该图标集也可免费用于您的软件，但需注明作者。Tango图标集是一个为Linux系统开发的庞大图标集，但它没有许可要求，因此可以在任何平台上使用。

资源	描述	许可证
Fugue byp.yusukekamiyamane	3,570 个 16x16 像素的 PNG 格式图标	CC BY 3.0
Diagona byp.yusukekamiyamane	400 个 16x16 和 10x10 的图标，格式为 PNG。	CC BY 3.0
Tango Icons by The TangoDesktop Project	使用 Tango 项目颜色主题的图标。	公共领域

> 虽然您可以控制菜单和工具栏中使用的图标大小，但在大多数情况下，建议保持默认设置。菜单中图标的标准大小为20x20像素。

> 比这更小的尺寸也可行，图标会居中显示，而不是放大显示。

创造您自己的图标

如果您不喜欢现有图标集中的任何一个，或者希望为您的应用程序打造独特风格，当然可以自行设计图标。图标可使用任何标准图形设计软件创建，并保存为带透明背景的PNG图像。图标应为正方形，且分辨率需确保在应用程序中使用时无需进行缩放。

使用图标文件

一旦您拥有图标文件（无论是来自图标集还是自行绘制），即可通过创建 `QtGui.QIcon` 实例并直接传入图标文件名的方式，在 Qt 应用程序中使用这些图标。

```
QtGui.QIcon("<filename>")
```

虽然您可以使用绝对路径（完整路径）和相对路径（部分路径）来指向您的文件，但绝对路径在分发应用程序时容易出现问题。相对路径只要图标文件存储在与脚本相同的相对位置即可正常工作，尽管在打包时管理这一点也可能较为困难。

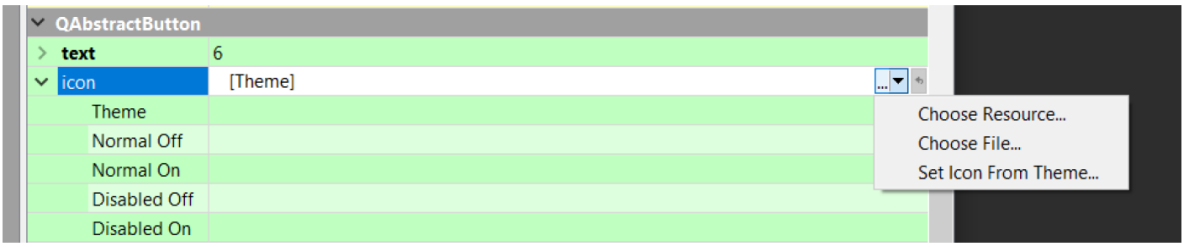
> 为了创建图标实例，您必须已经创建了一个 `QApplication` 实例。为了确保这一点，您可以在源文件的顶部创建应用程序实例，或者在使用它们的控件或窗口的 `__init__` 中创建 `QIcon` 实例。

自由桌面规范图标（Linux）

在 Linux 桌面系统中，有一个名为“自由桌面规范”（Free Desktop Specification）的标准，它为特定操作的图标定义了标准名称。

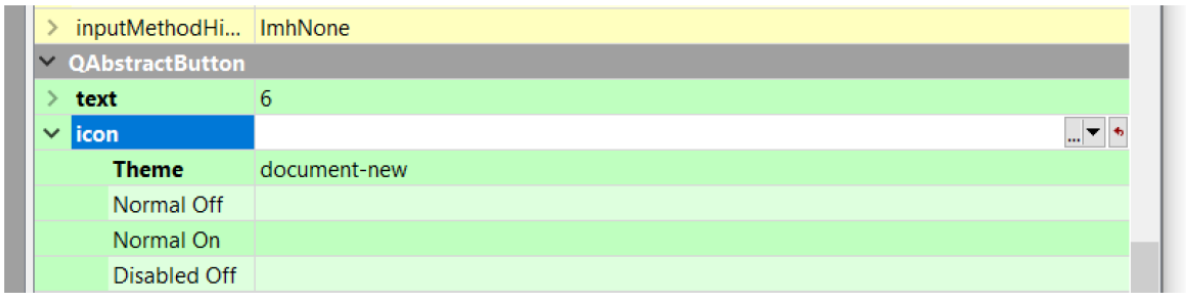
如果您的应用程序使用了这些特定的图标名称（并从主题中加载图标），那么在 Linux 上，您的应用程序将使用当前在桌面上启用的图标集。此处的目标是确保所有应用程序具有相同的用户界面风格，同时保持可配置性。

要在 Qt Designer 中使用这些功能，您需要选择下拉菜单并选择“Set Icon From Theme...”



> 图九十三：选择图标主题

然后，请您输入您要使用的图标名称，例如 `document-new`（参见 有效名称的完整列表）。



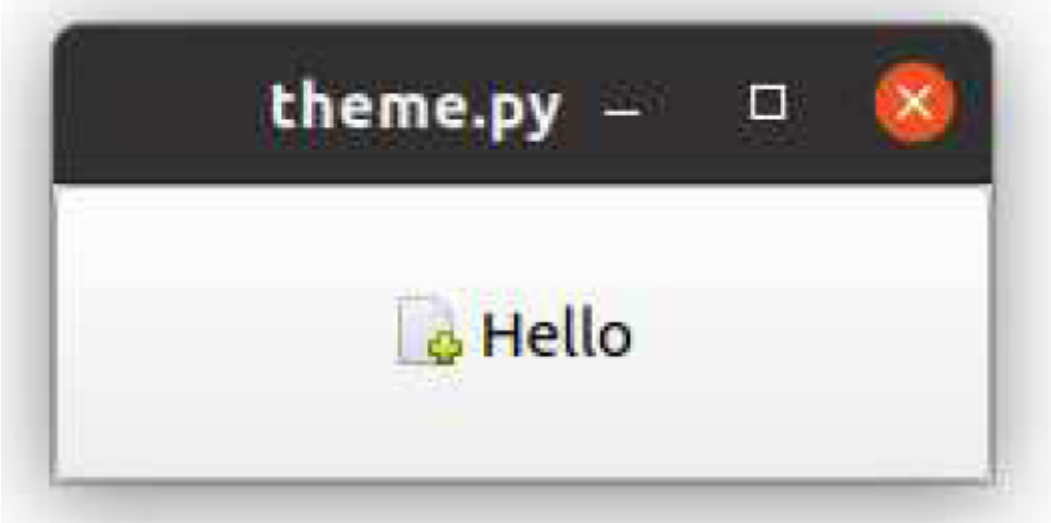
> 图九十四：选择图标主题

在代码中，您可以通过以下方式从当前活动的 Linux 桌面主题中获取图标：`icon =QtGui.QIcon.fromTheme("document-new")`。以下代码片段将生成一个小窗口（按钮），显示当前主题中的“新建文档”图标。

Listing 92. icons/linux.py

```
from PyQt6.QtWidgets import QApplication, QPushButton
from PyQt6.QtGui import QIcon

import sys

app = QApplication(sys.argv)
button = QPushButton("Hello")
icon = QIcon.fromTheme("document-new")
button.setIcon(icon)
button.show()

app.exec()
```

生成的窗口在Ubuntu系统上将呈现如下外观，采用默认的图标主题。



> 图九十五：Linux 自由桌面规范 “文档新建” 图标

如果您正在开发跨平台应用程序，您仍然可以在 Linux 上使用这些标准图标。为此，请为 Windows 和 macOS 使用您自己的图标，并在 Qt Designer 中创建自定义主题，使用自由桌面规范的图标名称。

16. Qt 样式表 (QSS)

到目前为止，我们已经了解了如何使用 `QPalette` 为 PyQt6 应用程序应用自定义颜色。然而，您还可以对 Qt6 中的控件外观进行许多其他自定义。系统提供了名为 Qt 样式表 (QSS) 的功能来实现这种自定义。

QSS 在概念上与用于设置网页样式的层叠样式表 (CSS) 非常相似，具有相似的语法和方法。在本节中，我们将介绍一些 QSS 的示例，以及如何使用它来修改控件的外观。

> 在控件上使用 QSS 会对性能产生轻微影响，因为在重绘控件时需要查找相应的规则。但是，除非您正在执行非常繁重的控件工作，否则这不会产生什么影响。

样式编辑器

为了使 QSS 规则的实际使用更容易一些，我们可以创建一个简单的演示应用程序，允许输入规则并将其应用于一些示例控件。我们将使用它来测试各种样式属性和规则。

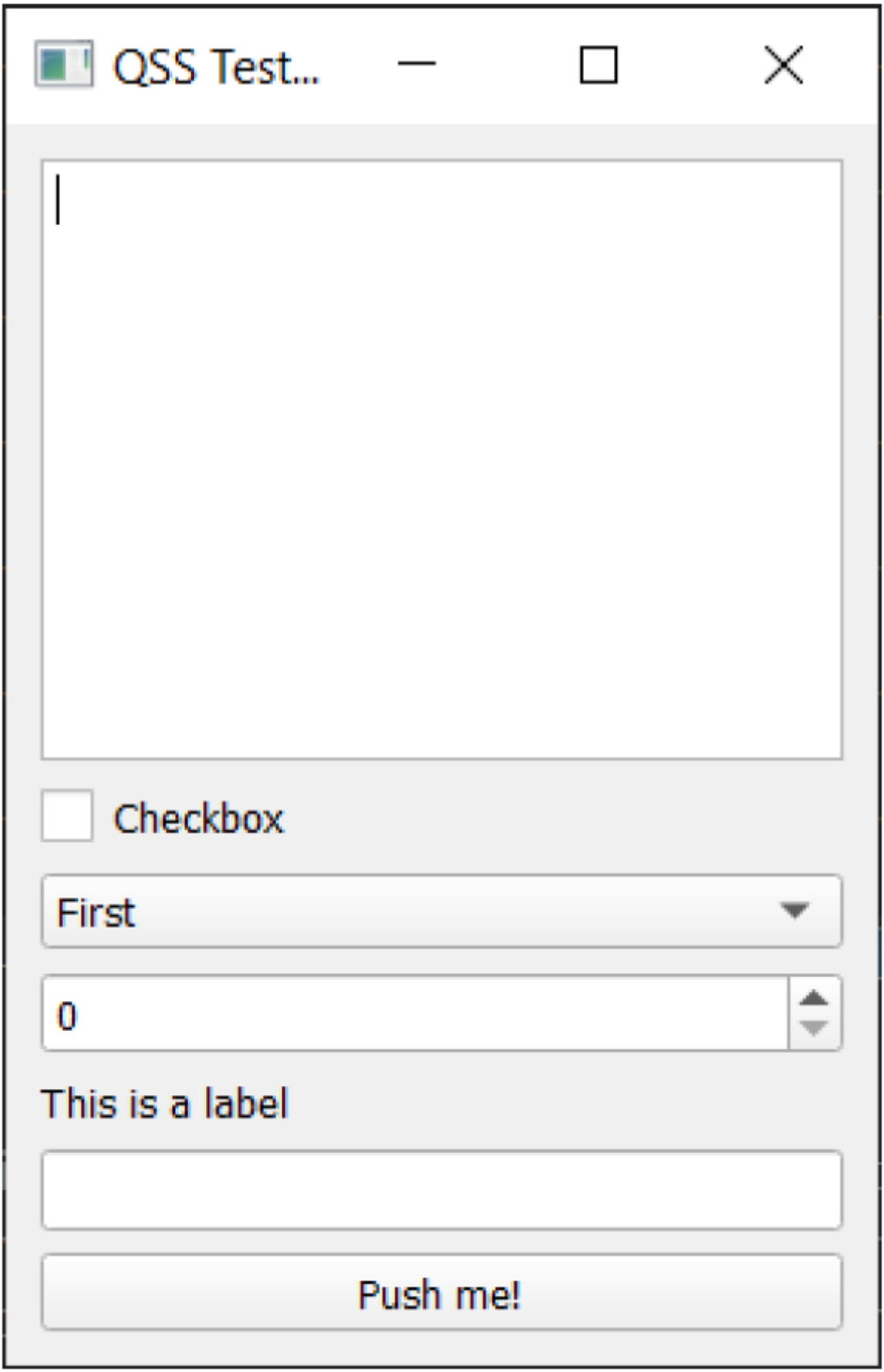
> 样式查看器的源代码如下所示，但也可以从本书的源代码中获得。

Listing 93. themes/qss_tester.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QColor, QPalette
from PyQt6.QtWidgets import (
    QApplication,
    QCheckBox,
    QComboBox,
    QLabel,
    QLineEdit,
    QMainWindow,
    QPlainTextEdit,
    QPushButton,
    QSpinBox,
    QVBoxLayout,
    QWidget,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("QSS Tester")

        self.editor = QPlainTextEdit()
        self.editor.textChanged.connect(self.update_styles)

        layout = QVBoxLayout()
        layout.addWidget(self.editor)

        # 定义一套简单的控件
        cb = QCheckBox("Checkbox")
        layout.addWidget(cb)

        combo = QComboBox()
        combo.setObjectName("thecombo")
        combo.addItems(["First", "Second", "Third", "Fourth"])
        layout.addWidget(combo)
```

```
        sb = QSpinBox()
        sb.setRange(0, 99999)
        layout.addWidget(sb)

        l = QLabel("This is a label")
        layout.addWidget(l)

        le = QLineEdit()
        le.setObjectName("mylineedit")
        layout.addWidget(le)

        pb = QPushButton("Push me!")
        layout.addWidget(pb)

        self.container = QWidget()
        self.container.setLayout(layout)

        self.setCentralWidget(self.container)

    def update_styles(self):
        qss = self.editor.toPlainText()
        self.setStyleSheet(qss)


app = QApplication(sys.argv)
app.setStyle("Fusion")

w = MainWindow()
w.show()

app.exec()
```

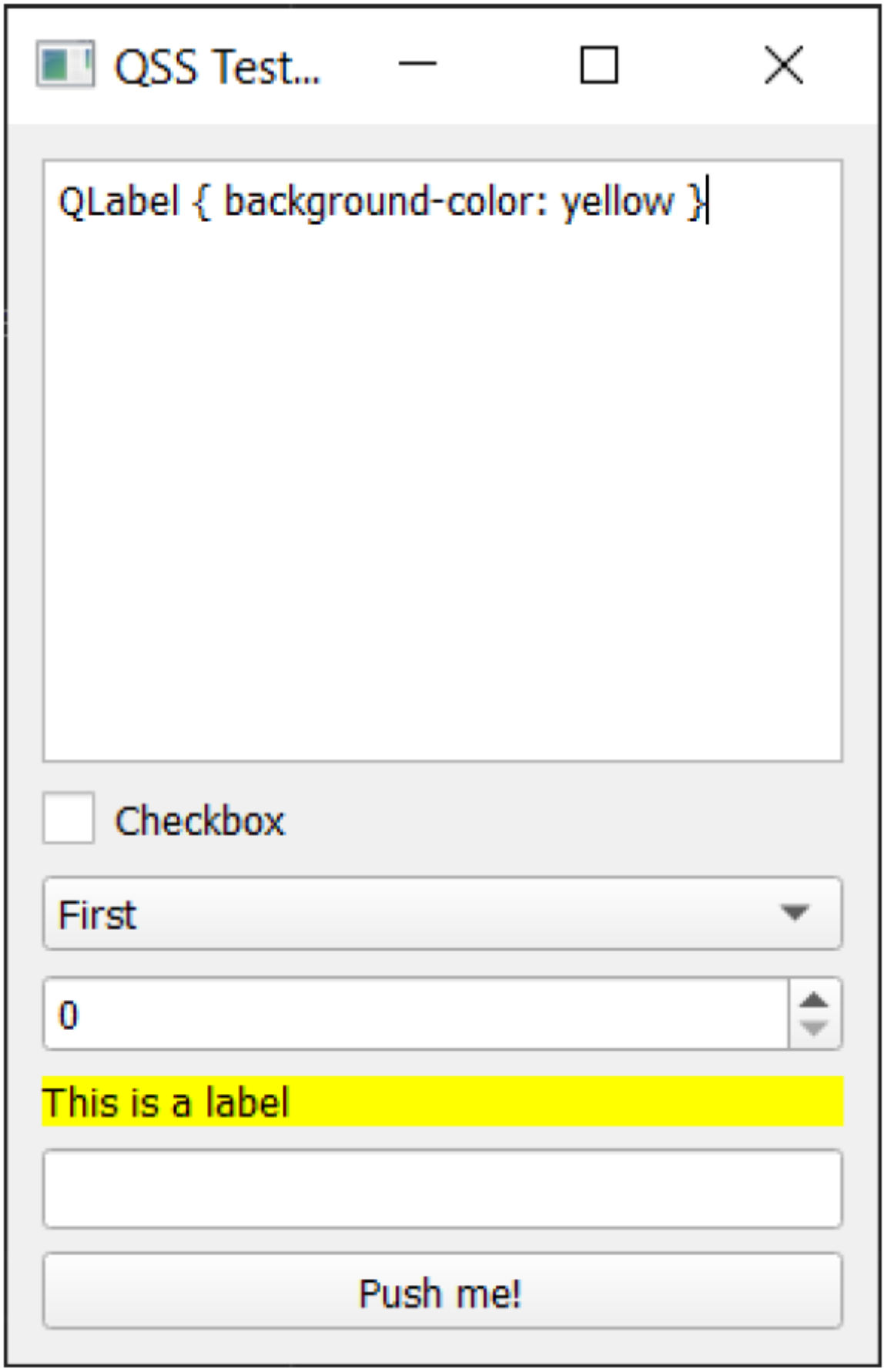
运行此应用程序后，您将看到以下窗口，顶部有一个文本编辑器（您可以在其中输入 QSS 规则），还有一组将应用这些规则的控件——我们稍后将介绍如何应用规则和继承。



> 图九十六：QSS测试应用程序，未应用任何规则。

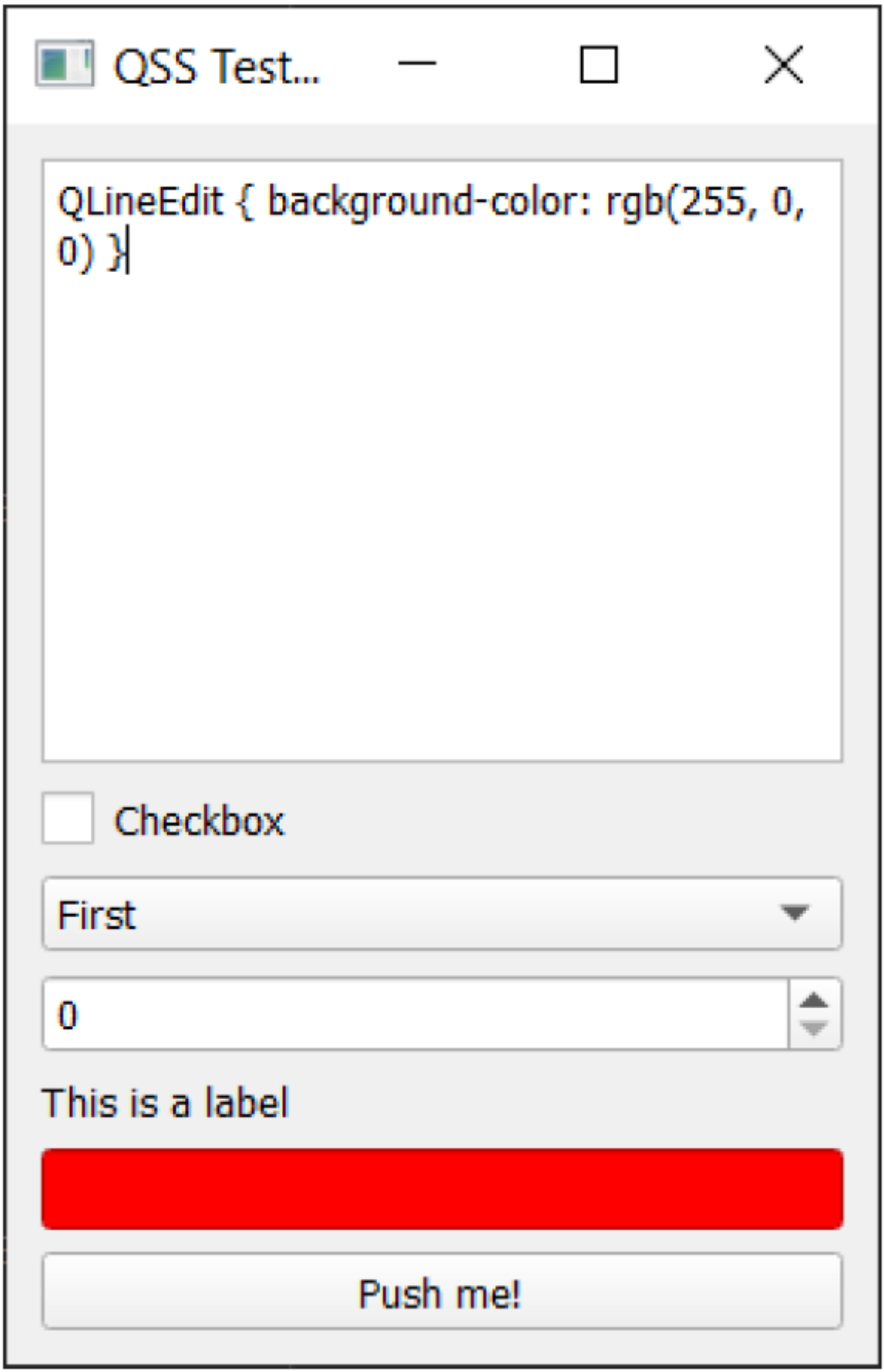
请您尝试在顶部的框中输入以下样式规则，并将结果与截图进行比较，以确保其正常运行。

```
QLabel { background-color: yellow }
```



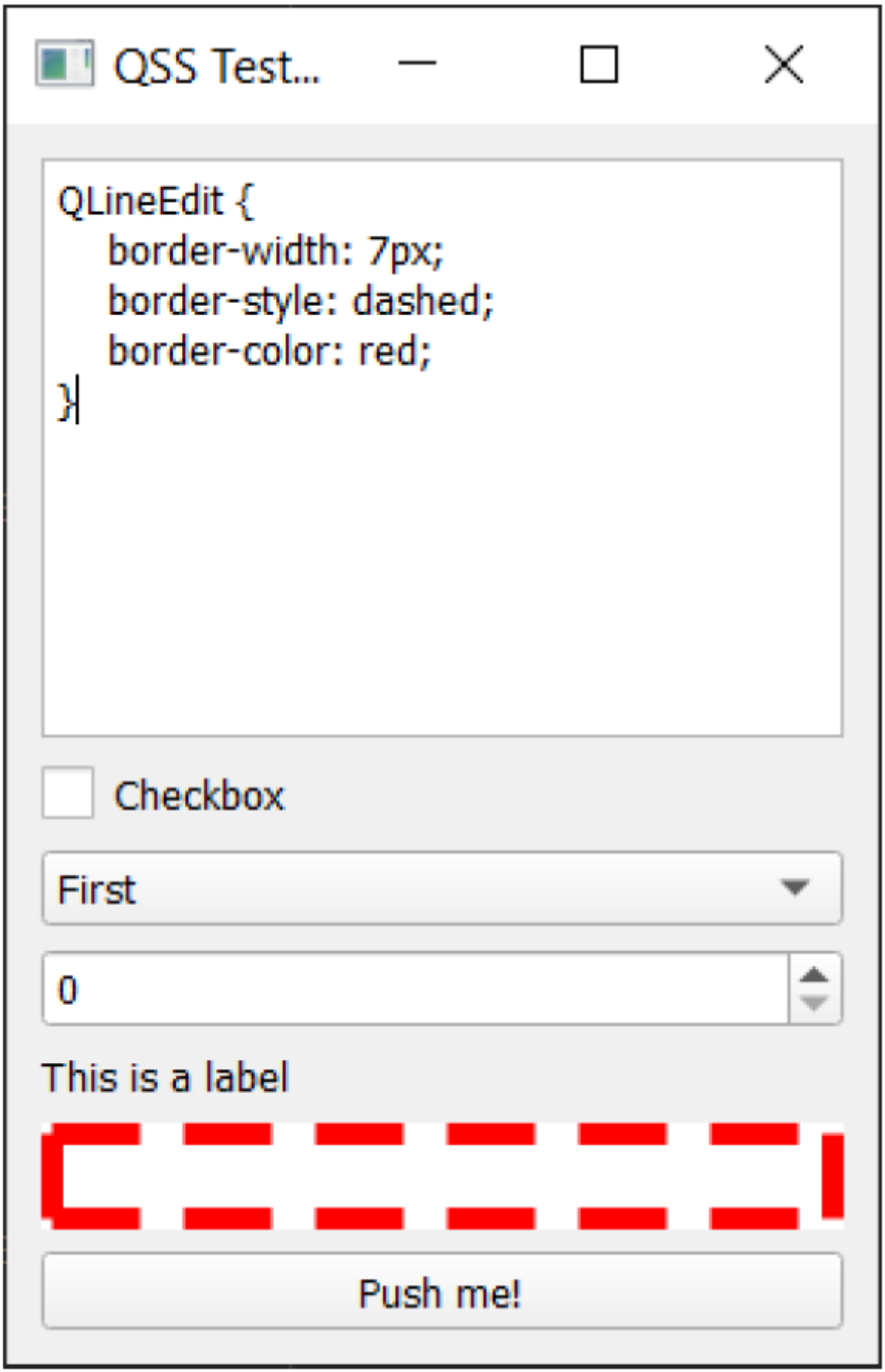
图九十七：为QLabel应用背景颜色：黄色

```
QLineEdit { background-color: rgb(255, 0, 0) }
```



图九十八：为QLineEdit应用背景颜色：rgb(255, 0, 0)（红色）

```
QLineEdit {
    border-width: 7px;
    border-style: dashed;
    border-color: red;
}
```



> 图九十九：为QLineEdit应用虚线红色边框

接下来，我们将详细探讨这些 QSS 规则如何为控件设置样式，并逐步构建一些更复杂的规则集。

> Qt 文档 中提供了可样式化的控件的完整列表。

样式属性

接下来，我们将介绍可用于使用 QSS 设置控件样式的属性。这些属性已按逻辑分段，每段包含相互关联的属性，以便于理解。您可以使用刚刚创建的 QSS 规则测试器应用程序在各种控件上测试这些样式。

以下表格中使用的类型如下所示。其中一些是由其他条目组成的复合类型。

> 您可以暂时跳过此表格，但在解释每个属性的合法值时将需要将其作为参考。

属性	类型	描述
`Alignment`	top, bottom, left, right, center	水平和/或垂直对齐
`Attachment`	scroll, fixed	滚动或固定附件
`Background`	Brush, Url, Repeat, Alignment	刷子类型、URL、重复次数、和对齐方式
`Boolean`	0, 1	True或者False
`Border`	Border Style, Length, Brush	简写边界属性
`Border Image`	none, Url Number, (stretch, repeat)	由九个部分组成的图像（顶部左侧、顶部中央、顶部右侧、中部左侧、中部、中部右侧、底部左侧、底部中央和底部右侧）
`Border Style`	dashed, dot-dash, dot-dot-dash, dotted, double, groove, inset, outset, ridge, solid, none	用于绘制边框的图案

属性	类型	描述
`Box Colors`	Brush	最多可指定四个 Brush 值，分别对应矩形框的顶部、右侧、底部和左侧边界。如果省略左侧颜色，则使用右侧颜色；如果省略底部颜色，则使用顶部颜色
`Box Lengths`	Length	最多可指定四个 Length 值，分别对应矩形框的顶部、右侧、底部和左侧边界。如果省略左侧颜色，则使用右侧颜色；如果省略底部颜色，则使用顶部颜色
`Brush`	Color, Gradient, PaletteRole	颜色、渐变或调色板中的一个条目
`Color`	rgb(r,g,b), rgba(r, g, b, a), hsv(h, s, v), hsva(h, s, v, a), hsl(h, s, l), hsla(h, s, l, a), #rrggbb, Color Name	指定颜色为RGB（红、绿、蓝）、RGBA（红、绿、蓝、透明度）、HSV（色调、饱和度、明度）、HSVA（色调、饱和度、明度、透明度）、HSL（色调、饱和度、明度）、HSLA（色调、饱和度、明度、透明度）或命名颜色。rgb() 或 rgba() 语法可使用 0 到255 之间的整数值或百分比值
`Font`	(Font Style, Font Weight) Font Size	简写字体属性
`Font Size`	Length	字体大小
`Font Style`	normal, italic, oblique	字体样式
`Font Weight`	normal, bold, 100, 200.., 900	字体粗细
`Gradient`	qlineargradient, qradialgradient, qconicalgradient	线性渐变，从起始点到终点。径向渐变，从焦点到围绕该焦点的圆上的终点。锥形渐变，围绕中心点。参见QLinearGradient 文档 以获取语法信息
`Icon`	Url(disabled, active, normal, selected)(on, off)	URL、QIcon.Mode 和 QIcon.State 的列表。例如：`file-icon: url(file.png), url(file_selected.png) selected;`
`Length`	Number(px, pt, em, ex)	一个数字后跟一个测量单位。如果未指定单位，则在大部分情况下使用像素。单位包括：px：像素，pt：一点的大小（即1/72英寸），em：字体的em宽度（即字母'M'的宽度），ex：字体的ex宽度（即字母'x'的高度）。
`Number`	一个十进制整数或实数	例如，123或者12.2312
`Origin`	margin, border, padding, content	请参阅盒模型以获取更多详细信息

属性	类型	描述
`PaletteRole`	alternate-base, base, bright-text, button, button-text, dark, highlight, highlighted-text, light, link, link-visited, mid, midlight, shadow, text, window, window-text	这些值与控件的 QPalette 中的颜色角色相对应，例如：`color: palette(dark);`
`Radius`	Length	一次或两次Length的出现
`Repeat`	repeat-x, repeat-y, no-repeat	repeat-x: 水平重复。 repeat-y: 垂直重复。 repeat: 水平和垂直重复。 no-repeat: 不重复。
`Url`	url(filename)	filename是磁盘上文件的名称，或使用Qt资源系统存储的文件名称。

这些属性和类型的完整详细信息也可在 QSS参考文档 中找到。

文本样式

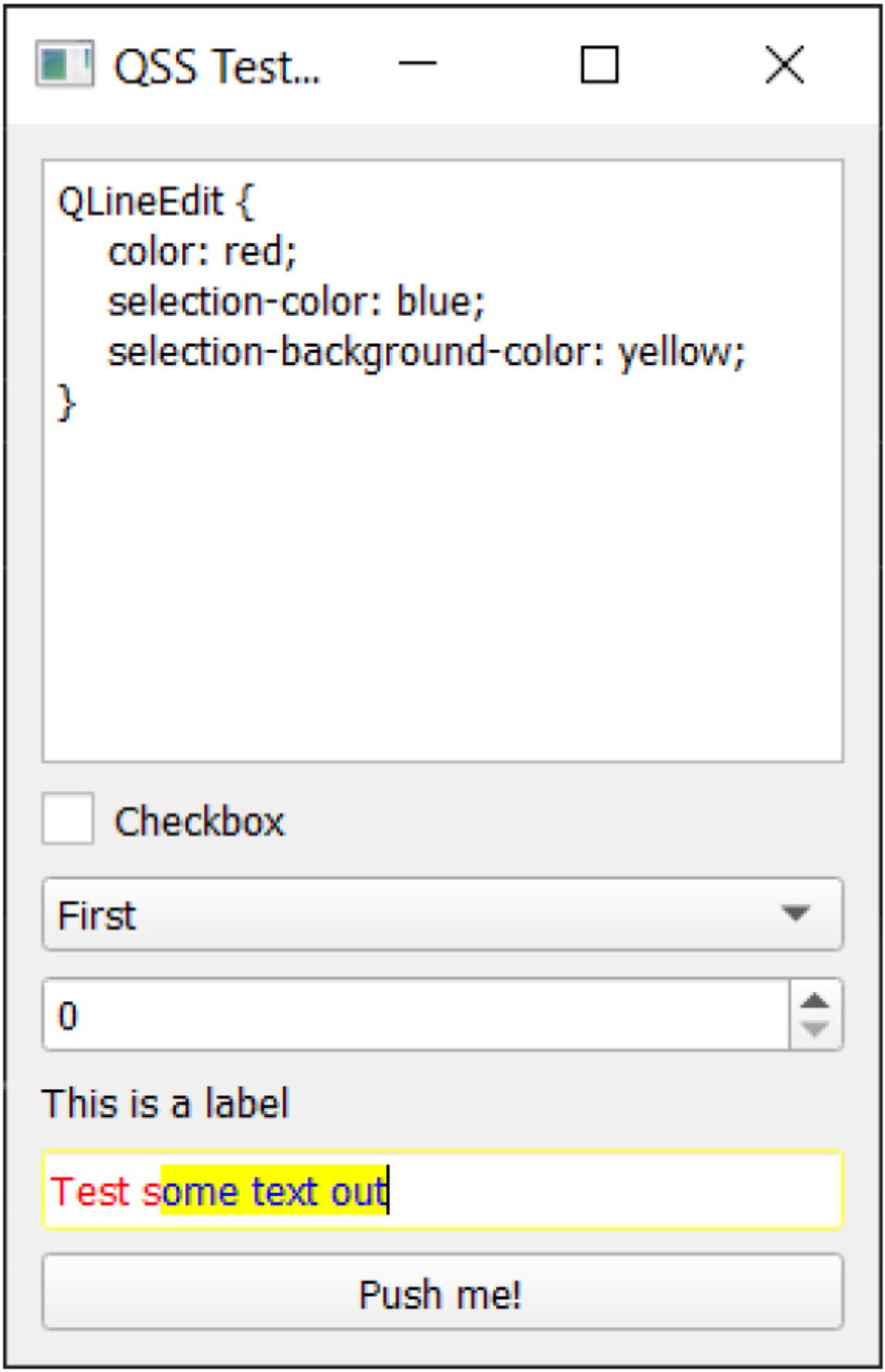
我们首先来介绍文本属性，这些属性可用于修改文本的字体、颜色和样式（粗体、斜体、下划线）。这些属性可应用于任何控件或控件。

属性	类型	描述
`color`	Brush (`QPaletteForeground`)	用于渲染文本的颜色
`font`	Font	设置文本字体的简写符号。相当于指定字体家族、字体大小、字体样式和/或字体粗细
`font-family`	String	字体家族
`font-size`	Font Size	字体大小。在此版本的Qt中，仅支持pt和px单位。
`font-style`	normal, italic, oblique	字体样式
`font-weight`	Font Weight	字体粗细
`selection-background-color`	Brush (`QPaletteHighlight`)	选定文本或项的背景色
`selection-color`	Brush (`PaletteHighlightedText`)	选定文本或项的前景色
`text-align`	Alignment	文本和图标在控件内容中的对齐方式
`text-decoration`	none, underline, overline, line-through	额外的文本效果

下面的示例代码片段将 `QLineEdit` 的颜色设置为红色，选中文本的背景色设置为黄色，选中文本的颜色设置为蓝色。

```
QLineEdit {
    color: red;
    selection-color: blue;
    selection-background-color: yellow;
}
```

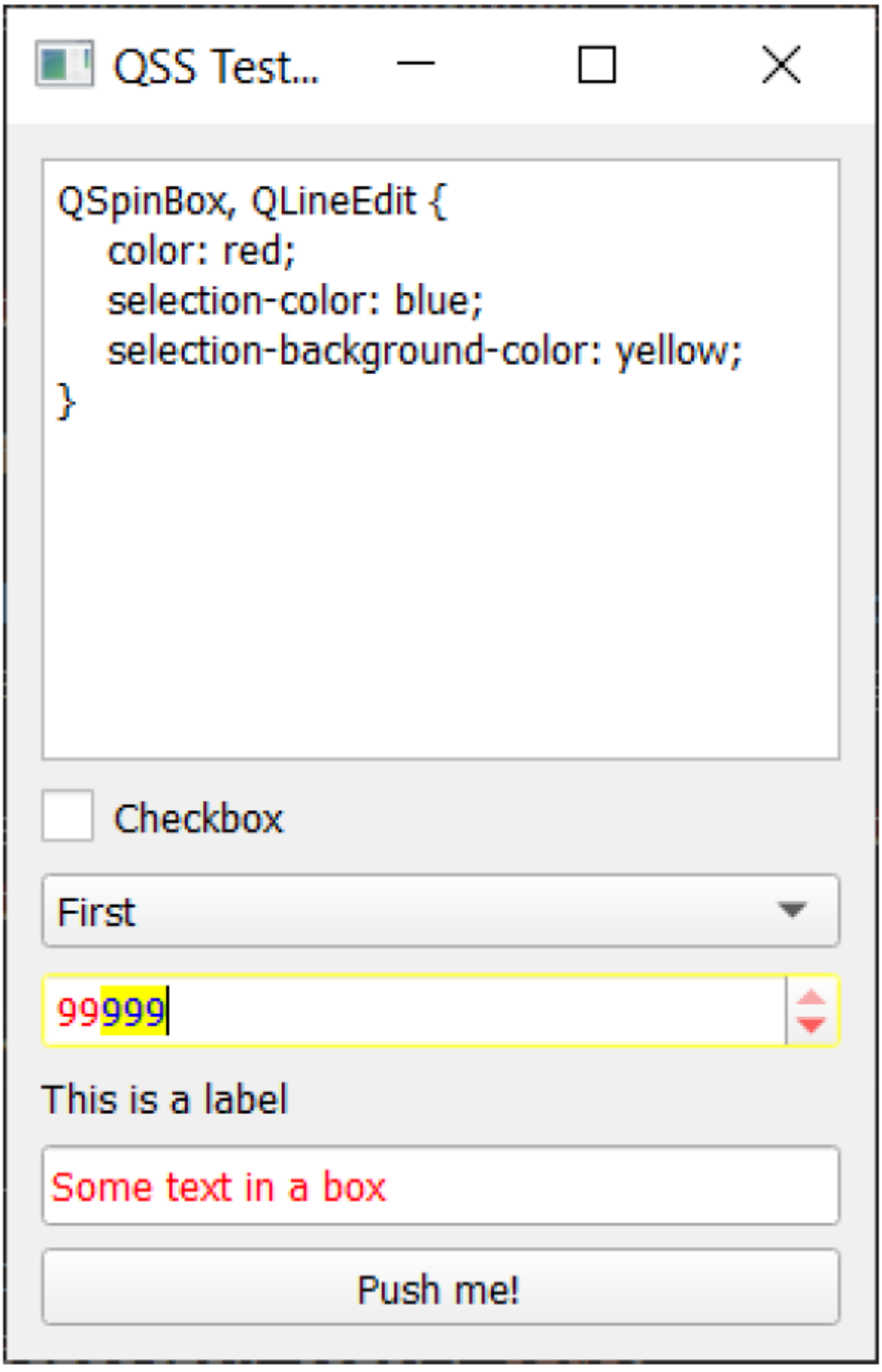
请您在 QSS 测试器中尝试一下，看看对 `QLineEdit` 的影响，结果如下所示。请注意，只有目标控件（`QLineEdit`）受到样式的影响。



图一百：将文本样式应用于 QLineEdit

我们可以将此规则应用于两种不同的控件，将它们都作为目标，并用逗号分隔。

```
QSpinBox, QLineEdit {
    color: red;
    selection-color: blue;
    selection-background-color: yellow;
}
```



> 图101：将文本样式应用于 QLineEdit 和 QSpinBox

在这个最后的例子中，我们将样式应用到 `QSpinBox`、`QLineEdit` 和 `QPushButton` 上，将字体设置为粗体和斜体，并将文本对齐方式设置为右对齐。

```
QSpinBox, QLineEdit, QPushButton {
    color: red;
    selection-color: blue;
    selection-background-color: yellow;
    font-style: italic;
    font-weight: bold;
    text-align: right;
}
```

这会产生如下所示的结果。请注意，`text-align` 属性并未影响 `QSpinBox` 或 `QLineEdit` 的对齐方式。对于这两个控件，必须使用 `.setAlignment()` 方法来设置对齐方式，而不是使用样式。

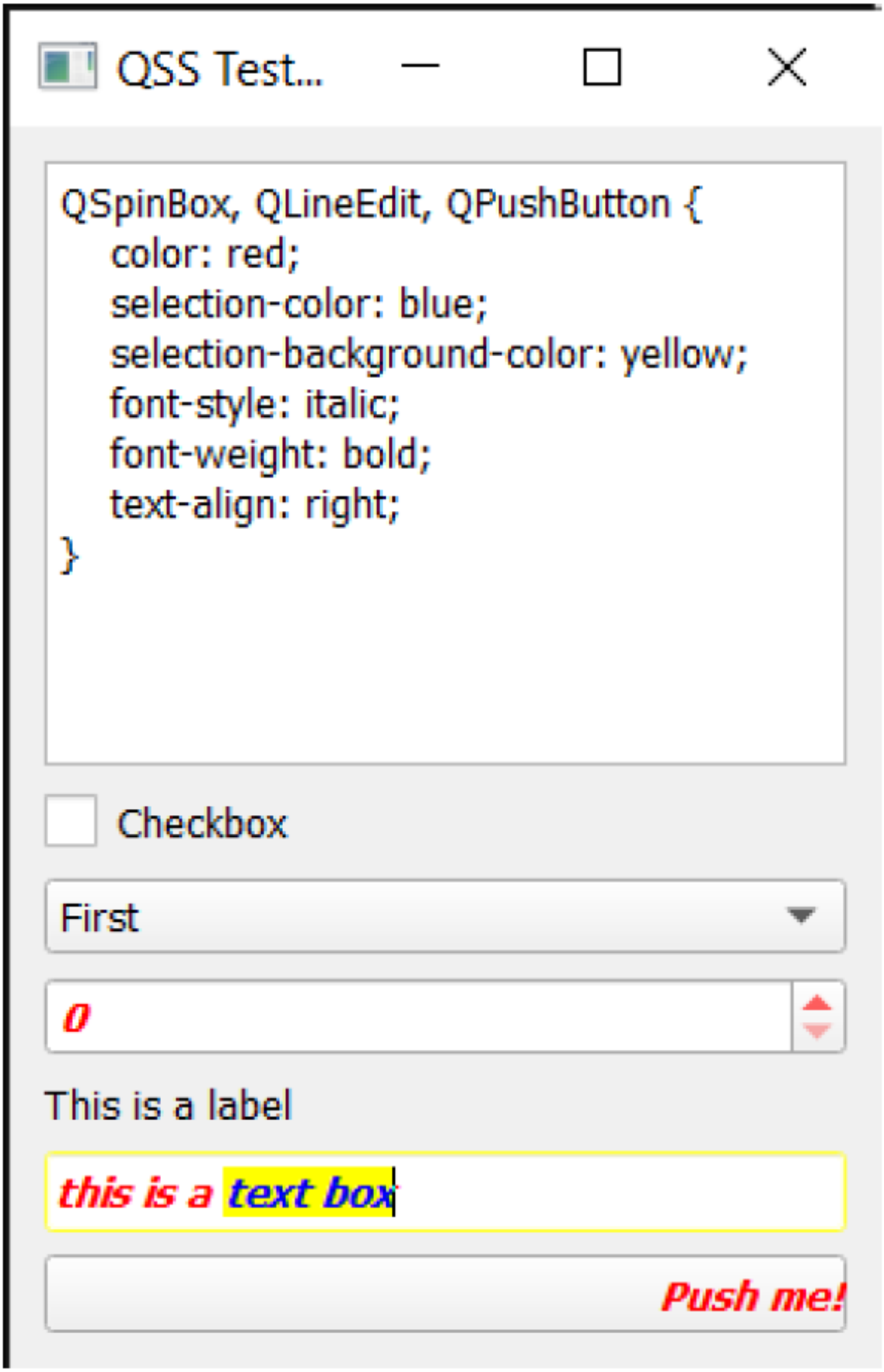


图102：将文本样式应用于 QPushButton、QLineEdit 和 QSpinBox

背景

除了设置文本样式外，您还可以使用纯色和图像设置控件背景的样式。对于图像，还有许多其他属性可用于定义图像在控件区域中的重复和定位方式。

属性	类型（默认）	描述
`background`	Background	简写语法用于设置背景。等同于指定背景颜色、背景图像、背景重复方式和/或背景位置。参见背景起源、选区背景颜色、背景裁剪、背景附着和替代背景颜色
`background-color`	Brush	控件使用的背景颜色
`background-image`	Url	控件使用的背景图像。图像的半透明部分允许背景颜色透过
`background-repeat`	Repeat (both)	背景图像是否以及如何重复以填充背景起始矩形
`background-position`	Alignment (top-left)	背景图像在背景起始矩形内的对齐方式
`background-clip`	Origin (border)	控件的矩形，背景在此处绘制
`background-origin`	Origin (padding)	控件的背景矩形，与背景位置和背景图像一起使用

以下示例将指定的图像应用于我们用于输入规则的 `QPlainTextEdit` 的背景。

```
QPlainTextEdit {
    color: white;
    background-image: url(../otje.jpg);
}
```

图片通过 `url()` 语法引用，传入文件的路径。这里我们使用 `../otje.jpg` 指向父目录中的文件。

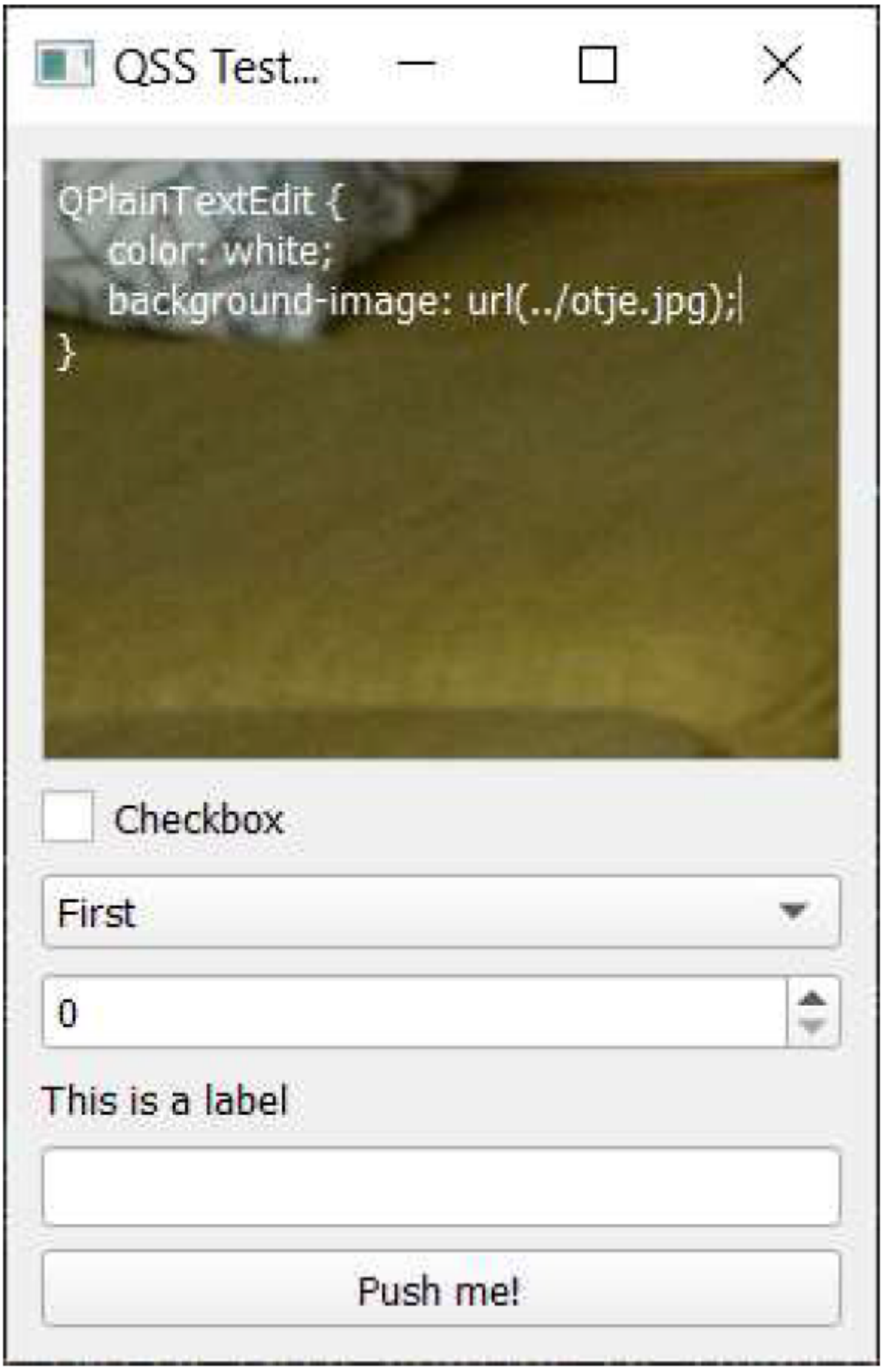
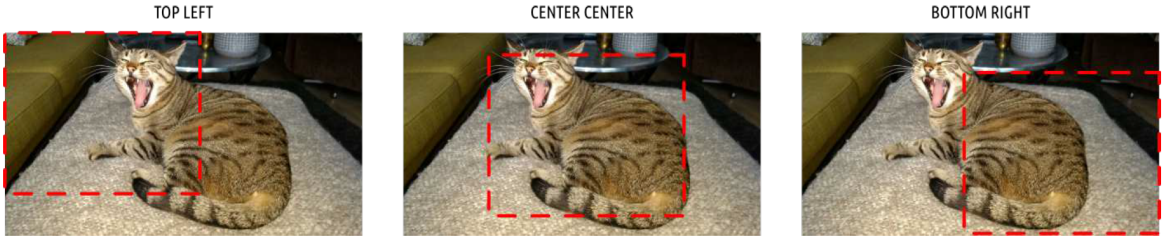

QSS Test...
QPlainTextEdit {
color: white;
background-image: url(../otje.jpg);
}
Checkbox
First
0
This is a label
Push me!

图103：一张背景图片

> 虽然此语法与层叠样式表中使用的语法相同，但远程文件不能使用 URL 加载。

要定位控件中的背景，您可以使用 `background-position` 属性。该属性定义了图像与控件原点矩形上的同一点对齐的点。默认情况下，原点矩形是控件的填充区域。



> 图104：背景位置的示例

居中 `center` 是指图像的中心将沿两个轴与控件的中心对齐。

```
QPlainTextEdit {
    color: white;
    background-image: url(../otje.jpg);
    background-position: center center;
}
```

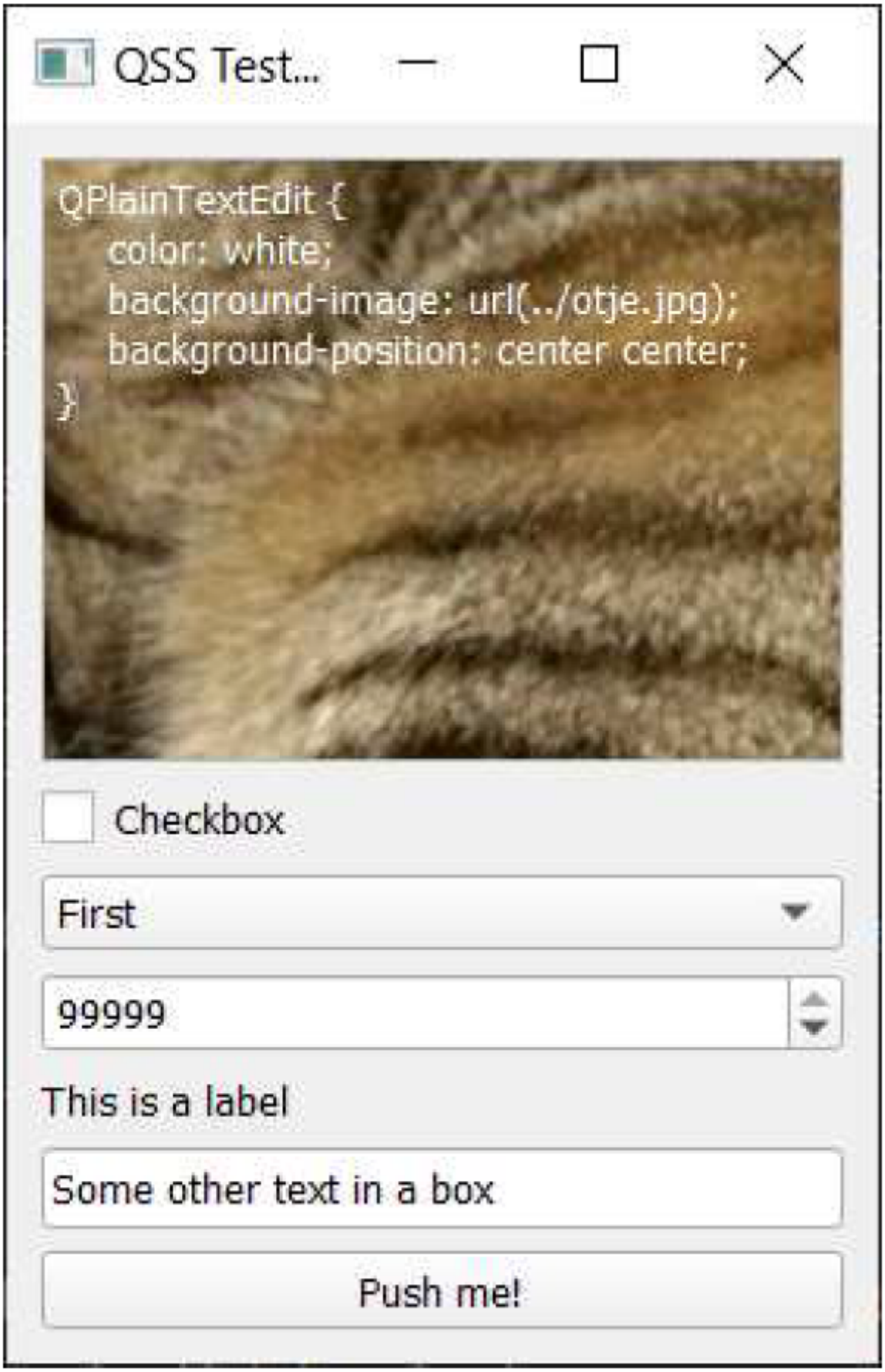


图105：居中的背景图片

要将图像的右下角与控件的原始矩形的右下角对齐，您可以使用以下代码：

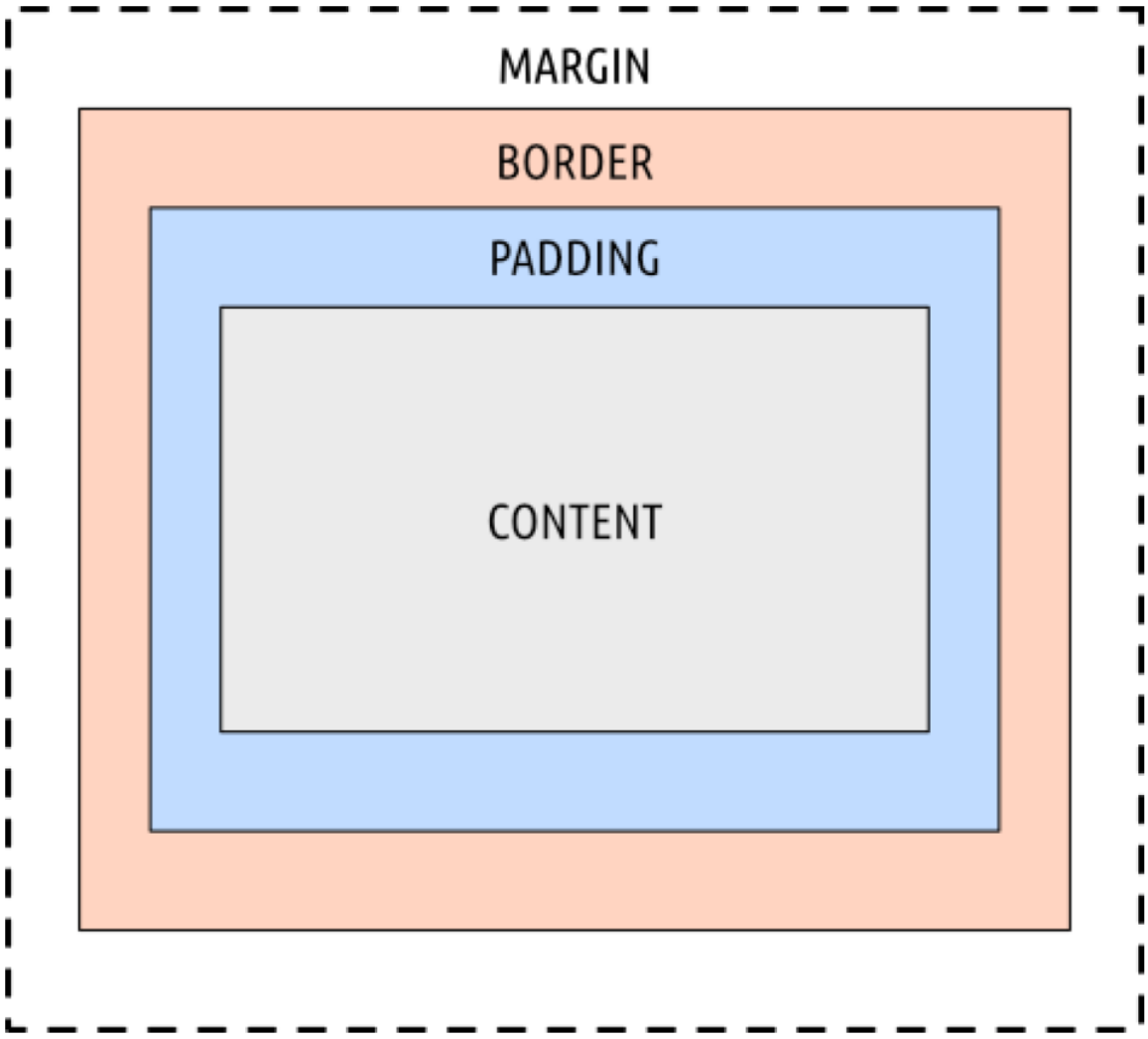
```
QPlainTextEdit {
    color: white;
    background-image: url(../otje.jpg);
    background-position: bottom right;
}
```

`background-origin` 属性可用于修改起始矩形。该属性接受以下值之一：`margin`、`border`、`padding` 或 `content`，用于定义该特定框作为背景位置对齐的参考点。

为了理解这是什么意思，我们需要看看控件框模型。

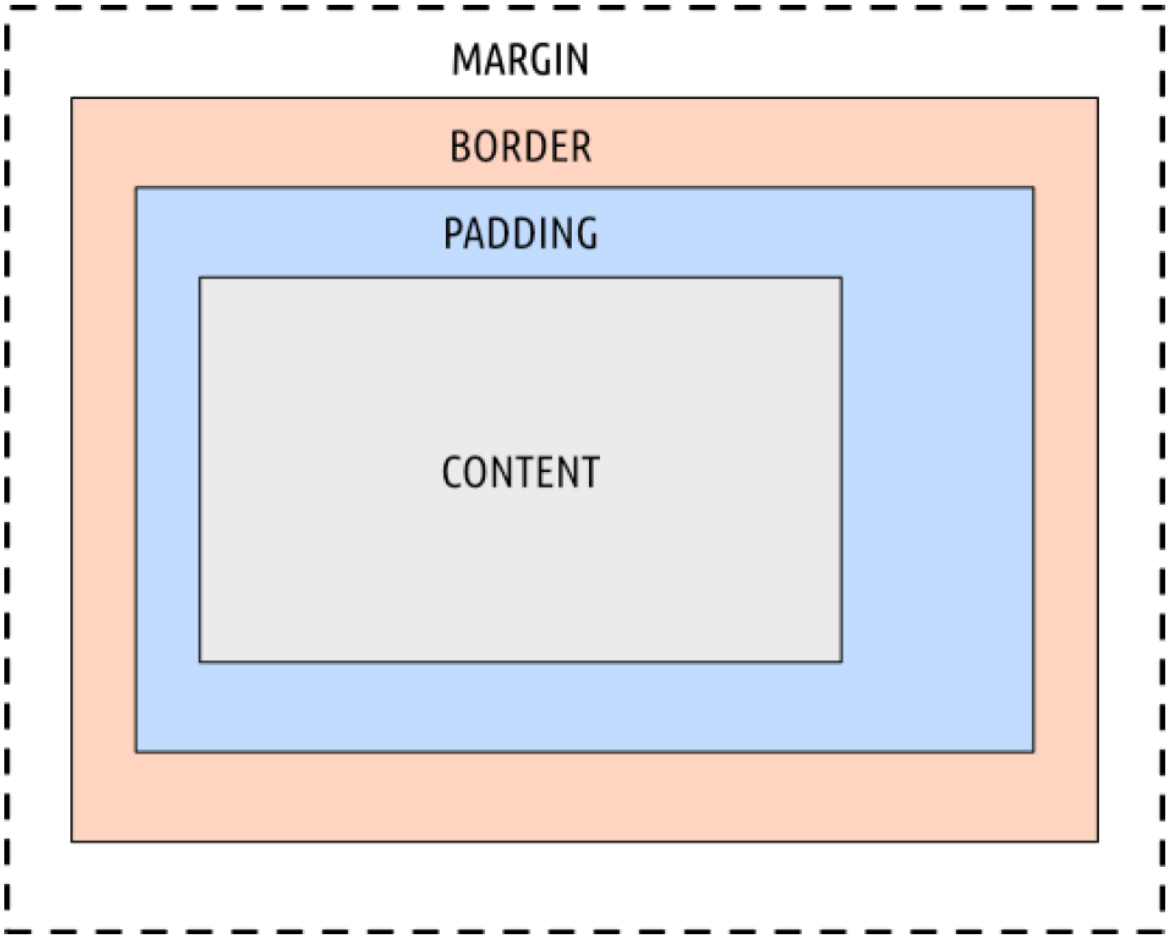
控件盒模型

“盒模型”一词描述了包围每个控件的框（矩形）之间的关系，以及这些框对控件之间的大小或布局的影响。每个 Qt 控件都由四个同心框包围——从内到外，分别是内容、填充、边框和边距。每个框都具有特定的属性，这些属性决定了框的大小、外观和位置。每个框的属性都由 Qt 控件的属性列表定义。



> 图106：盒模型

如果您增加内部框的大小就会增加外部框的大小。例如，增加控件的填充会增加内容和边框之间的空间，同时增加边框本身的尺寸。



> 图107：在右侧添加填充对其他框的影响

可用于修改各个框的属性如下所示

属性	类型（默认）	描述
`border`	Border	设置控件边框的简写符号。相当于指定边框颜色、边框样式和/或边框宽度。还有`border-top`，`border-right`，`border-bottom`和`borderleft`.
`border-color`	Box Colors (`QPaletteForeground`)	边框所有边缘的颜色。也包括：`border-top-color`、`border-right-color`、`border-bottom-color`、`border-left-color`.用于指定特定边缘的颜色。
`border-image`	Border Image	用于填充边框的图像。该图像会被分割成九个部分，并在必要时进行适当拉伸。
`border-radius`	Radius	边框角点的半径（曲线）。也包括`border-top-left-radius`，`border-top-right-radius`，`borderbottom-right-radius`和`border-bottomleft-radius`，用于指定特定角点的半径。
`border-style`	Border Style (none)	所有边框边缘的样式。此外，还有`border-top-style`、`border-right-styleborder-bottom-style`和`border-leftstyle`用于特殊的边缘

属性	类型（默认）	描述
`border-width`	Box Lengths	边框的宽度。也包括 `bordertop-width` 、`border-right-width` 、`borderbottom-width` 和 `border-left-width`
`margin`	Box Lengths	控件的边距。还有 `margin-top`、`margin-right` 、`margin-bottom` 和 `marginleft`
`outline`		围绕对象边界绘制的轮廓
`outline-color`	Color	轮廓的颜色。参见border-color
`outline-offset`	Length	轮廓与控件边框之间的偏移量
`outline-style`		指定用于绘制轮廓的图案。另请参阅border-style
`outline-radius`		为轮廓添加圆角。还有 `outline-bottom-left-radius` , `outline-bottom-right-radius` , `outlinetop-left-radius` 和 `outline-top-right-radius` 填充框长度。还有 `padding-top` , `padding-right` , `padding-bottom` 和 `padding-left` , 填充控件

以下示例修改了 `QPlainTextEdit` 控件的边距、边框和填充。

```
QPlainTextEdit {
    margin: 10;
    padding: 10px;
    border: 5px solid red;
}
```

> 关于单位的说明
>
> 在此示例中，我们使用像素（px）作为填充和边框的单位。边距的值也以像素为单位，因为在未指定单位时，像素是默认单位。您还可以使用以下单位之一：
>
> • px 像素
>
> • pt 一"点"大小（即 1/72 英寸）
>
> • em 字体的 em 宽度（即 'M' 的宽度）
>
> • ex 字体的 ex 宽度（即 'x' 的高度）

在QSS测试器中查看结果时，您可以看到红色边框内的填充物和红色边框外的边距。

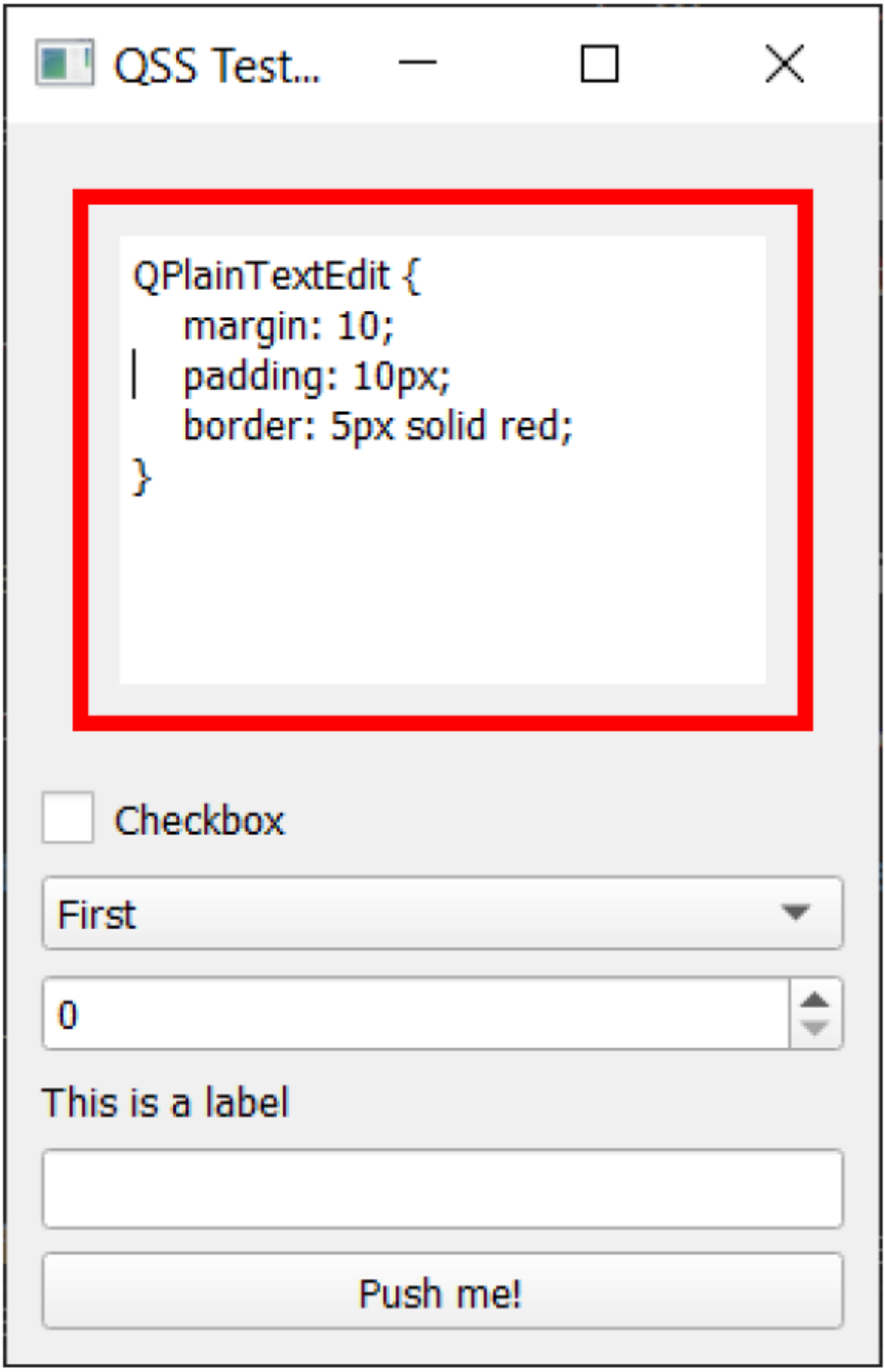


图108：盒模型

您还可以为轮廓添加半径以生成曲线边缘

```
QPlainTextEdit {
    margin: 10;
    padding: 10px;
    border: 5px solid red;
    border-radius: 15px;
}
```

> 图109：边框半径为15像素（曲线）

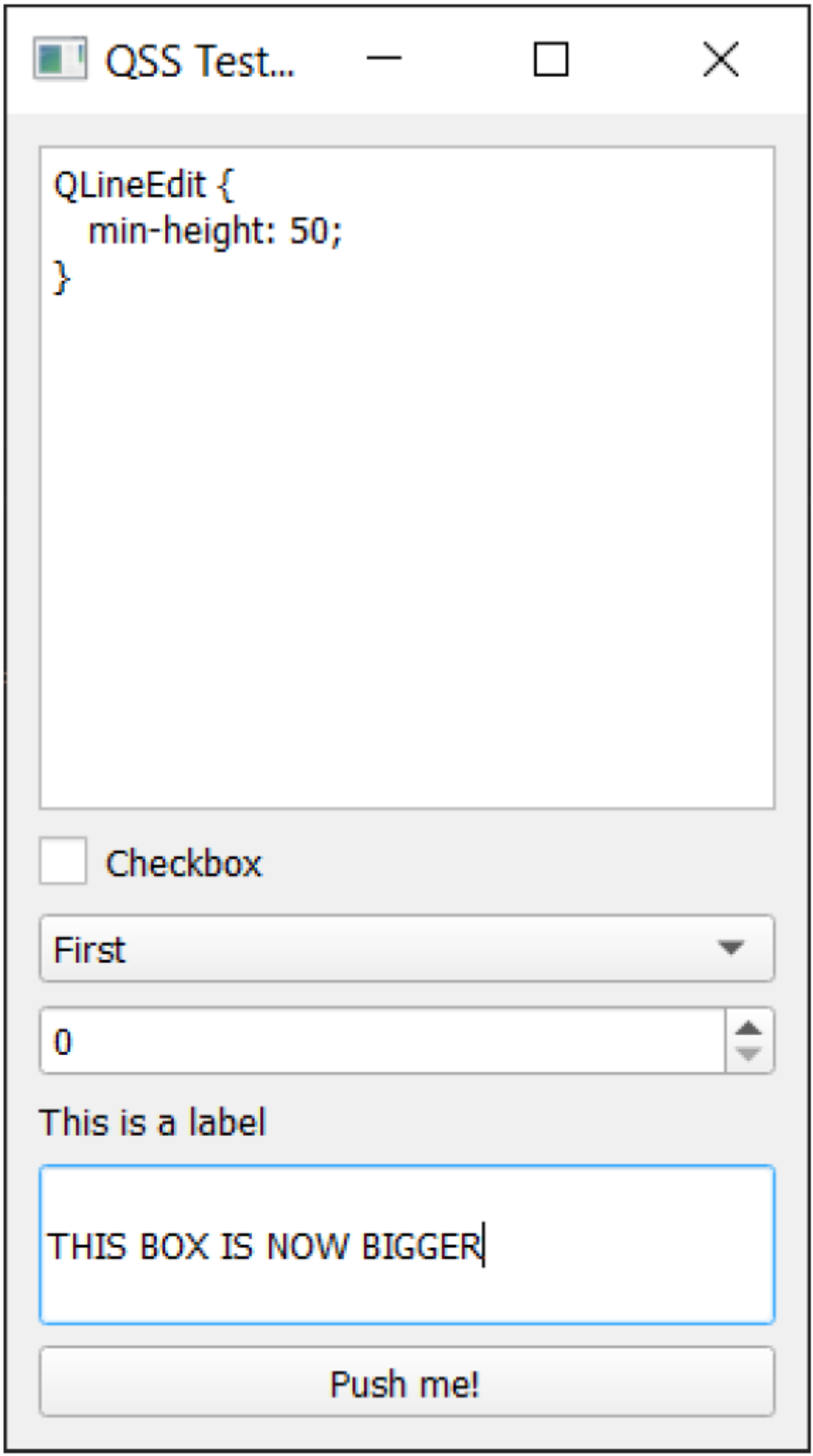
调整控件大小

您可以使用 QSS 控制控件的大小。但是，虽然有特定的 `width` 和 `height` 属性（见下文），但这些属性仅用于指定子控件的大小。要控制控件，您必须使用 `max-` 和 `min-` 属性。

属性	类型（默认）	描述
`max-height`	Length	控件或子控件的最大高度
`max-width`	Length	控件或子控件的最大宽度
`min-height`	Length	控件或子控件的最小高度
`min-width`	Length	控件或子控件的最小宽度

如果您指定的 `min-height` 属性值大于控件通常的高度，则该控件将被放大。

```
QLineEdit {
    min-height: 50;
}
```


QSS Test...
QLineEdit {
min-height: 50;
}
Checkbox
First
0
This is a label
THIS BOX IS NOW BIGGER
Push me!

> 图110：为QLineEdit设置最小高度，以放大其显示区域。

但是，在设置 `min-height` 时，控件当然可以大于此值。要指定控件的确切大小，您可以同时指定尺寸的 `min-` 和 `max-`。

```
QLineEdit {
    min-height: 50;
    max-height: 50;
}
```

这将把控件锁定到此高度，防止它随着内容的变化而改变大小。

> 使用此方法时请小心，否则可能会导致控件无法读取！

控件的特定样式

到目前为止，我们所看到的样式都是通用的，可用于大多数控件。然而，还有许多可设置的控件特定属性。

属性	类型（默认）	描述
`alternate-background-color`	Brush (`QPaletteAlternateBase`)	在 `QAbstractItemView` 子类中使用的替代背景色
`background-attachment`	Attachment (scroll)	确定在 `QAbstractScrollArea` 中，背景图像是否会随着视线焦点滚动或固定
`button-layout`	Number(`SH_DialogButtonLayout`)	按钮在 `QDialogButtonBox` 或 `QMessageBox` 中的布局。可能的值为 0（Windows）、1（Mac）、2（KDE）、3（Gnome）和 5（Android）
`dialogbuttonbox-buttonshave-icons`	Boolean	`QDialogButtonBox` 中的按钮是否显示图标。如果此属性设置为 1，`QDialogButtonBox` 中的按钮将显示图标；如果设置为 0，则不显示图标
`gridline-color`	Color(`SH_Table_GridLineColor`)	`QTableView` 中网格线的颜色
`icon`	Url+	控件图标。目前唯一支持此属性的控件是 `QPushButton`
`icon-size`	Length	控件中图标的宽度和高度
`lineedit-password-character`	Number(`SH_LineEdit_PasswordCharacter`)	作为 Unicode数字的 `QLineEdit` 密码字符
`lineedit-password-mask-delay`	Number(`SH_LineEdit_PasswordMaskDelay`)	`QLineEdit` 密码掩码延迟（以毫秒为单位），即在应用密码掩码字符之前等待的时间
`messagebox-text-interaction-flags`	Number(`SH_MessageBox_TextInteractionFlags`)	消息框中文字的交互行为（来自 `Qt.TextInteractionFlags` ）

属性	类型（默认）	描述
`opacity`	Number(`SH_ToolTipLabel_Opacity`)	控件的不透明度（仅限工具提示） 0-255
`paint-alternating-rowcolors-for-empty-area`	bool	`QTreeView` 是否会绘制数据末尾之后的交替行
`show-decoration-selected`	Boolean(`SH_ItemView_ShowDecorationSelected`)	控制 `QListView` 中的选区是否覆盖整个行，还是仅限于文本的范围
`titlebar-show-tooltipson-buttons`	bool	是否在窗口标题栏按钮上显示工具提示
`widget-animation-duration`	Number	动画应持续的时间（毫秒）

这些仅适用于描述中指定的控件（或其子类）

目标定位

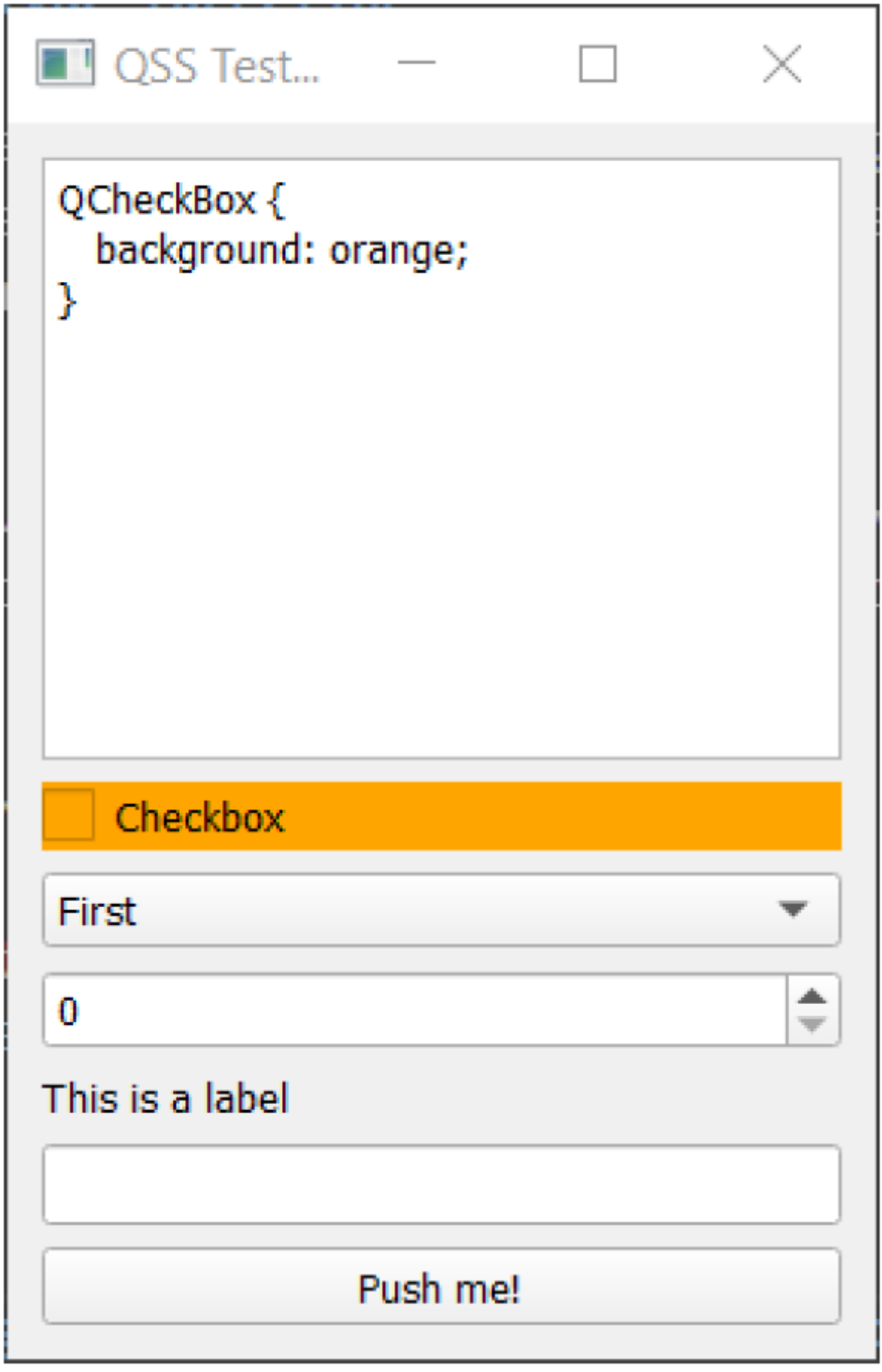
我们已经看到了各种不同的 QSS 属性，并根据它们的类型将它们应用到控件上。但是，如何针对单个控件，Qt 如何决定将哪些规则应用到哪些控件以及何时应用呢？接下来，我们将看看针对 QSS 规则的其他选项以及继承的效果。

类型	例子	描述
通用 (Universal)	`*`	匹配所有控件
类型(Type)	`QPushButton`	`QPushButton` 类或其子类的实例
属性(Property)	`QPushButton[flat="false"]`	非平面的 `QPushButton` 实例。可与支持 `.toString()` 的任何属性进行比较。也可使用 `class="classname"`
属性值包含 (Property contains)	`QPushButton[property~="something"]`	`QPushButton` 的实例，其中属性（`QString` 列表）不包含给定的值
类(Class)	`.QPushButton`	`QPushButton` 的实例，但不是子类
编号(ID)	`QPushButton#okButton`	一个 `QPushButton` 实例，其对象名称为 okButton
后代 (Descendant)	`QDialog QPushButton`	`QDialog` 的子对象（子对象、孙对象等）的 `QPushButton` 实例
子选择器 (Child)	`QDialog > QPushButton`	`QPushButton` 的实例，这些实例是 `QDialog` 的直接子元素

我们将依次查看这些定位规则，并使用我们的QSS 测试工具进行测试。

类型(Type)

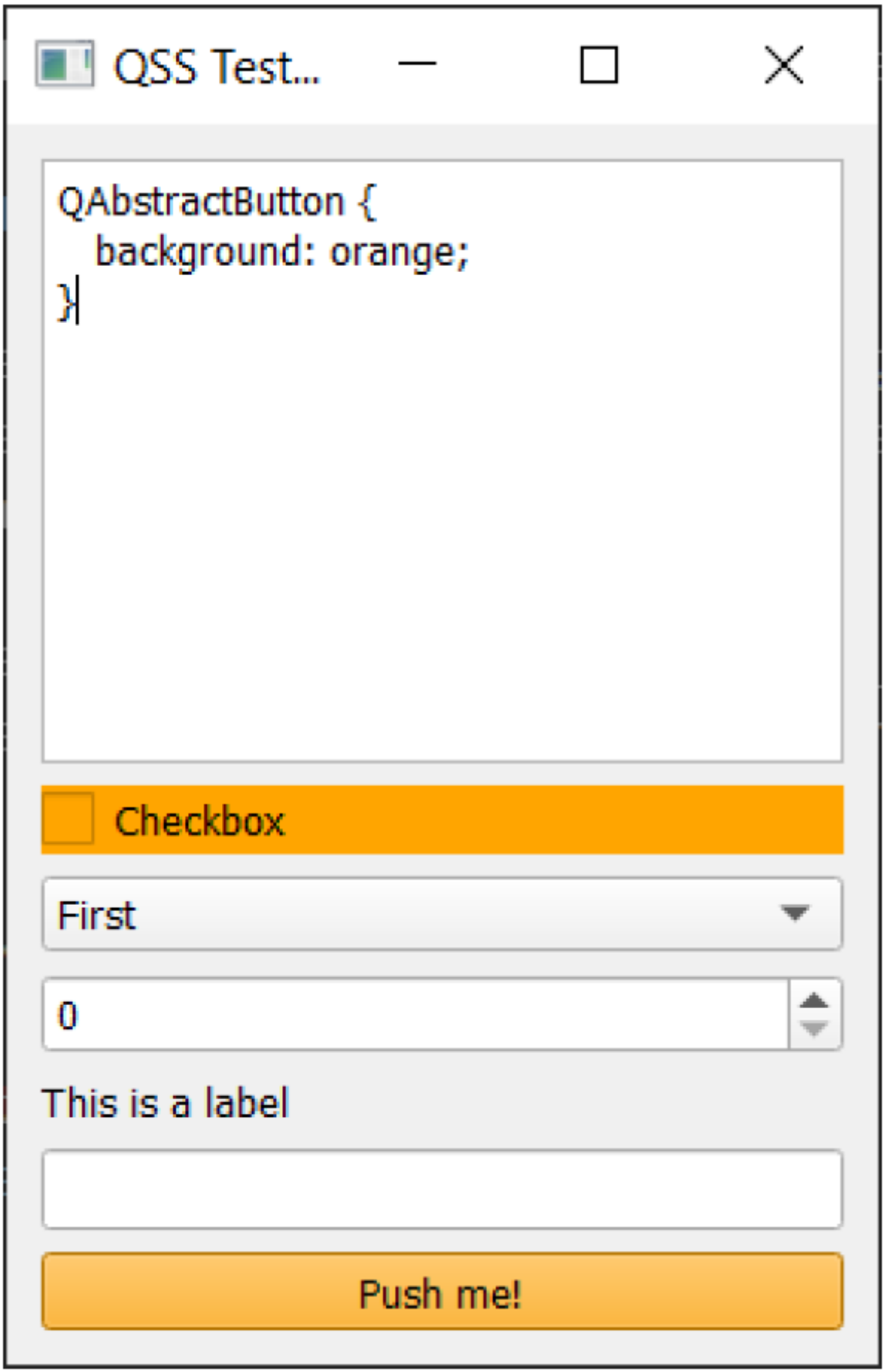
我们在 QSS 测试器中已经看到了类型定位的应用。在这里，我们针对各个控件的类型名称（例如 `QComboBox` 或 `QLineEdit`）设置了规则。



> 图111：针对QComboBox进行操作不会影响其他无关的控件类型。

然而，以这种方式定位类型也会定位该类型的任何子类。例如，我们可以定位 `QAbstractButton` 来定位任何从它派生的类型。

```
QAbstractButton {
    background: orange;
}
```



> 图112：针对 QAbstractButton 进行操作会影响所有子类。

这种行为意味着所有控件都可以使用 `QWidget` 作为目标。例如，以下代码将所有控件的背景设置为红色。

```
QWidget {
    background: red;
}
```

图113：通过父类选择QSS

类(Class)

有时，您可能只想针对特定类别的控件，而不针对任何子类。为此，您可以使用类定位——在类型名称前添加一个`.`

以下针对`QWidget`实例，但不针对任何从`QWidget`派生的类。在我们的 QSS 测试器中，我们唯一的`QWidget`是用于保存布局的中央控件。因此，以下代码将该容器控件的背景颜色更改为橙色。

```
.QWidget {
    background: orange;
}
```

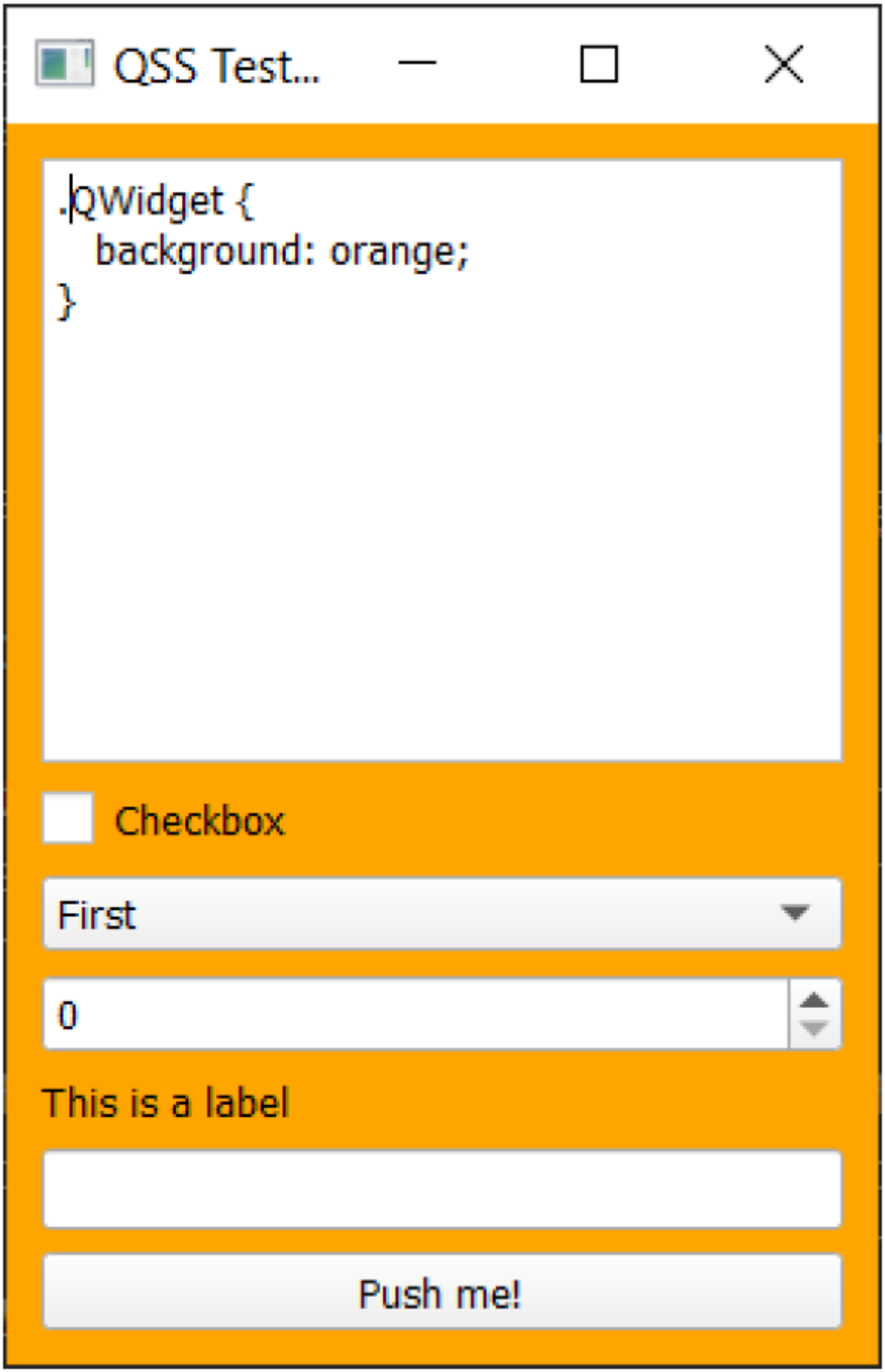


图114：直接针对某个类进行操作不会影响其子类。

ID定位 `#`

所有 Qt 控件都有一个唯一标识它们的对象名称。在 Qt Designer 中创建控件时，您可以使用对象名称来指定该对象在父窗口中的名称。然而，这种关系只是为了方便起见——您可以在自己的代码中为控件设置任何对象名称。这些名称可用于将 QSS 规则直接应用到特定的控件。

在我们的QSS测试应用中，我们为 `QComboBox` 和 `QLineEdit` 设置了ID用于测试。

```
combo.setObjectName('thecombo')
le.setObjectName('mylineedit')
```

属性(Property) `[property="<value>"]`

您可以使用任何可作为字符串（或其值具有 `.toString()` 方法）的控件属性来定位控件。这可用于定义控件上一些相当复杂的状态。

以下是一个简单的示例，通过文本标签定位到一个 `QPushButton`。

```
QPushButton[text="Push me!"] {
    background: red;
}
```

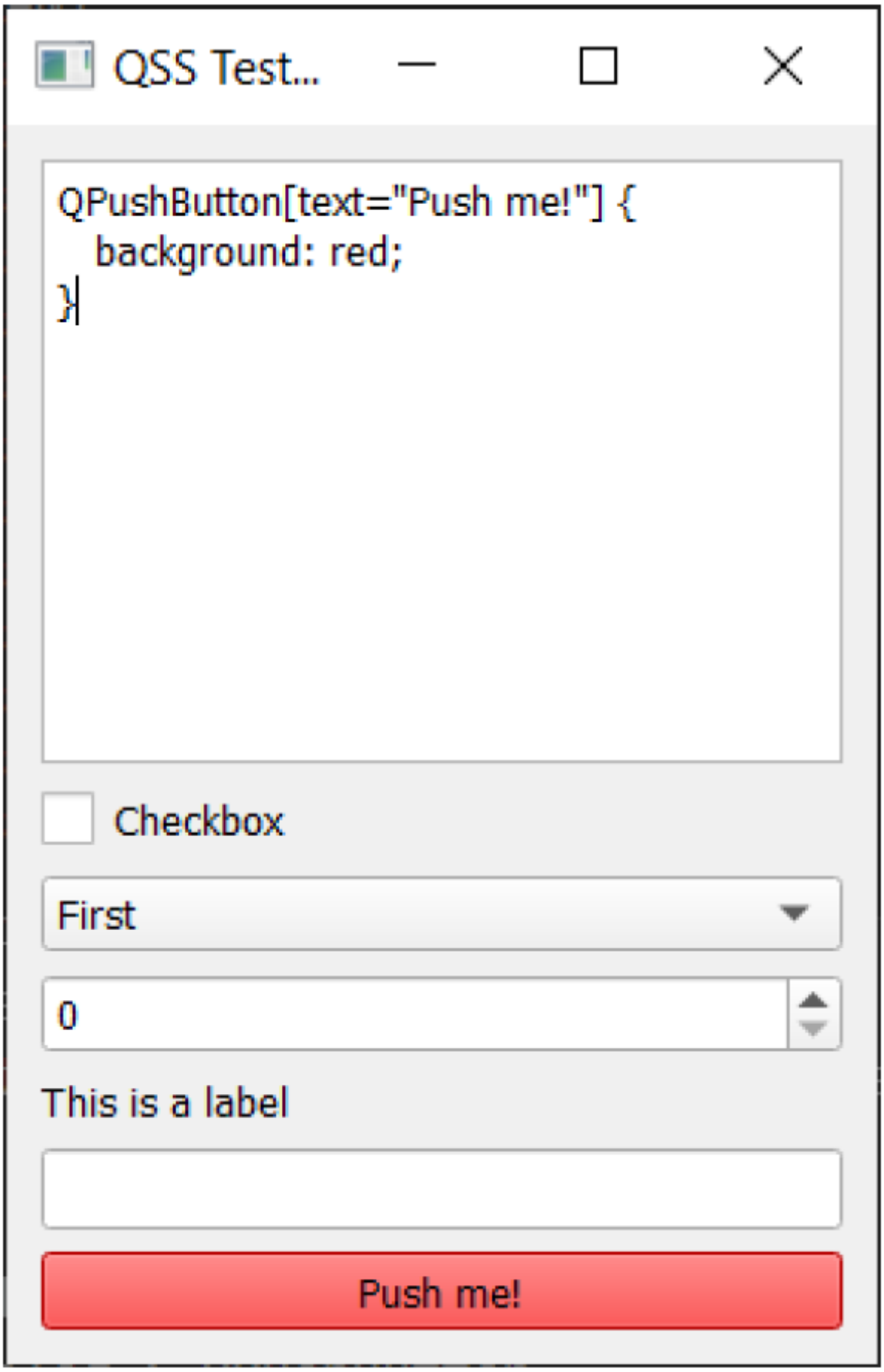


图115：通过标签文本定位 QPushButton

> 通过可见文本来定位控件通常是一个非常糟糕的主意，因为当您尝试翻译应用程序或更改标签时，这会引入错误。

规则在样式表首次设置时应用于控件，不会对属性的更改做出响应。如果 QSS 规则所针对的属性被修改，您必须触发样式表的重新计算才能使其生效——例如，通过重新设置样式表。

后代(Descendant)

要定位给定类型控件的子控件，可以将控件链接在一起。以下示例定位了 `QMainWindow` 的子控件 `QComboBox`，无论它是直接子控件，还是嵌套在其他控件或布局中。

```
QMainWindow QComboBox {
    background: yellow;
}
```

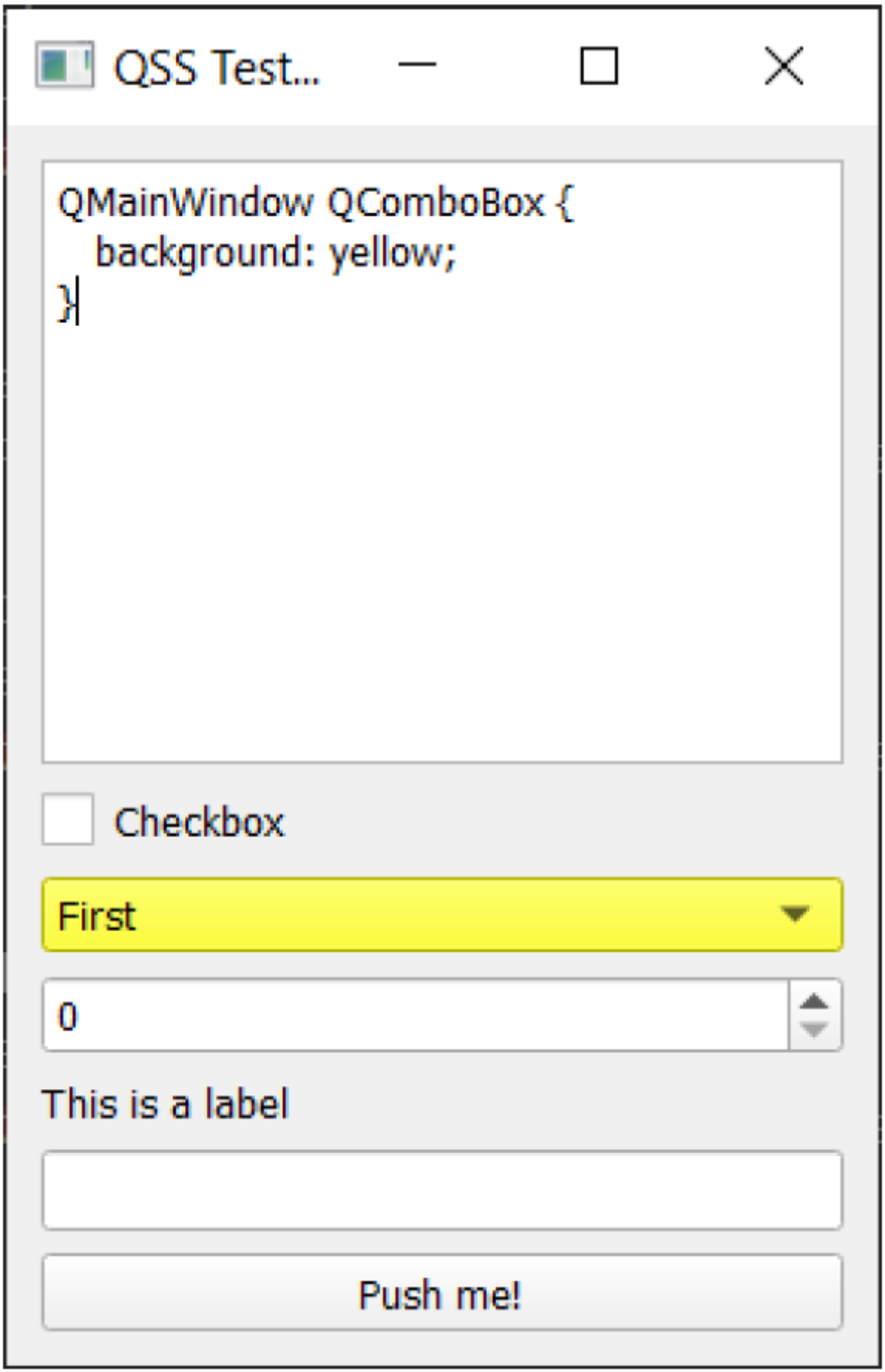


图116：定位一个作为 QMainWindow 子窗口的 QCombobox

要定位所有子元素，您可以将全局选择器作为定位的最后一个元素使用。您还可以将多种类型组合起来，仅定位应用中存在该层次结构的位置。

```
QMainWindow QWidget * {
    background: yellow;
}
```

在我们的 QSS 测试器应用程序中，我们有一个外部 `QMainWindow`，其中有一个 `QWidget` 中央控件来保存布局，然后是该布局中的控件。因此，上面的规则只匹配单个控件（它们都以 `QMainWindow` `QWidget` 为父控件，顺序如上）。

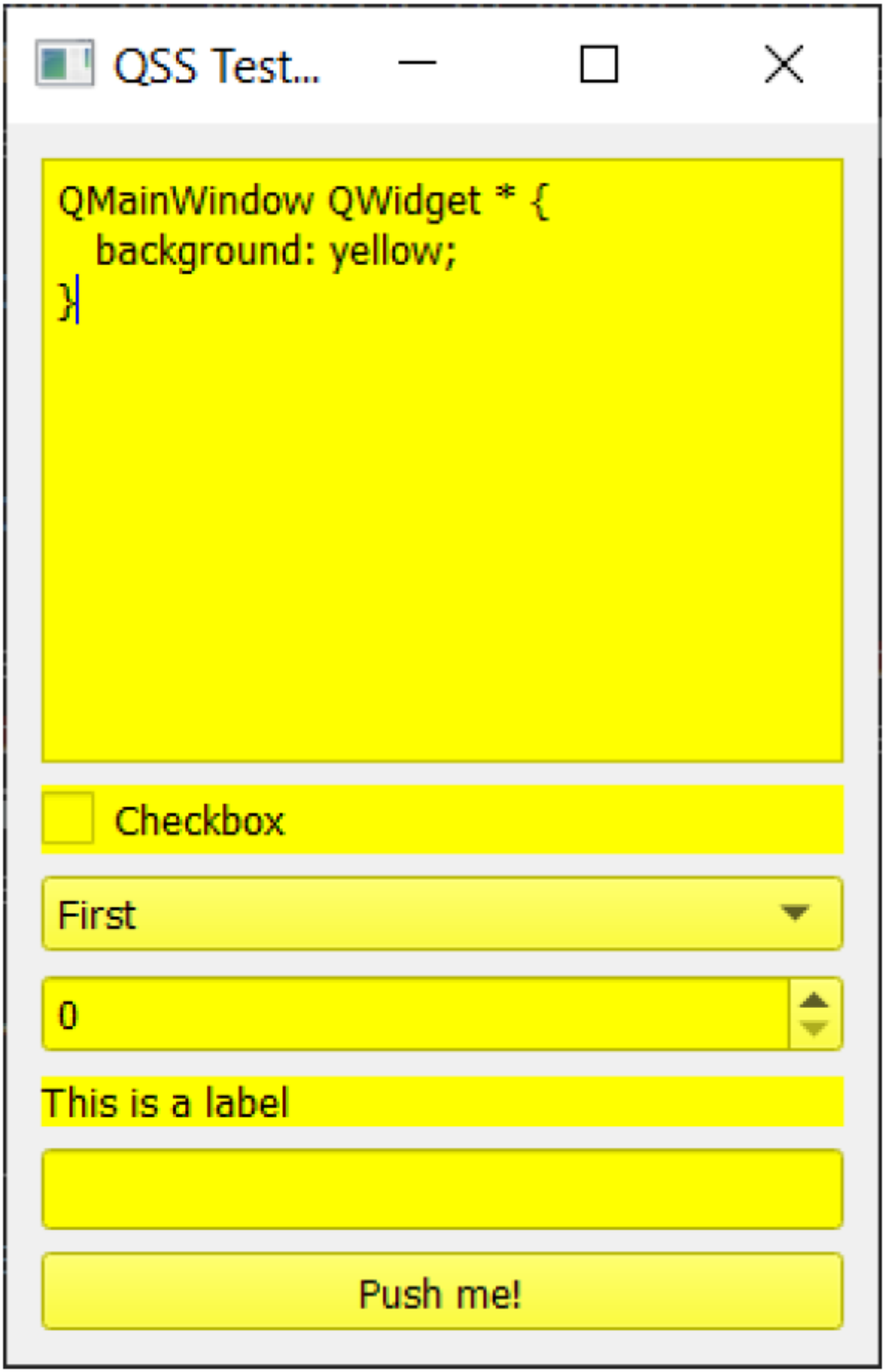


图117：定位一个作为 QMainWindow 子对象的 QComboBox

子选择器(Child) `>`

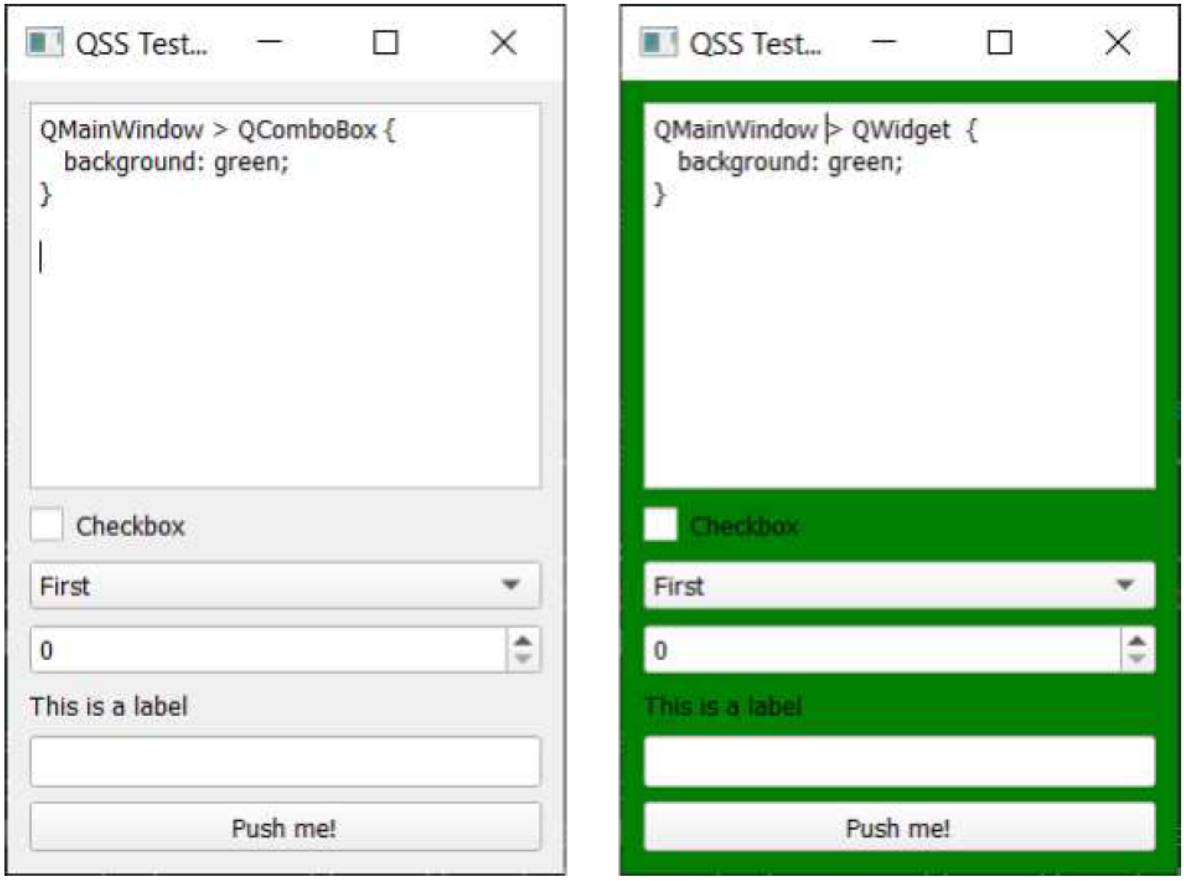
您还可以使用 `>` 选择器来定位另一个控件的直接子控件。这只会匹配完全符合该层次结构的情况。

例如，以下代码仅针对直接位于 `QMainWindow` 下的 `QWidget` 容器。

```
QMainWindow > QWidget {
    background: green;
}
```

但以下内容不会匹配任何内容，因为在我们的 QSS 应用程序中，`QComboBox` 控件不是 `QMainWindow` 的直接子元素。

```
QMainWindow > QComboBox { /* 一个也不匹配 */
    background: yellow;
}
```



> 图118：定位一个作为QWidget的直接子元素的QComboBox

继承

样式表可应用于 `QApplication` 和控件，并适用于样式控件及其所有子控件。控件的有效样式表由其所有祖先（父级、祖父母级……一直到窗口）的样式表以及 `QApplication` 本身的样式表组合而成。

规则按照具体性顺序应用。这意味着，针对特定控件 ID 的规则将覆盖针对该类型所有控件的规则。例如，以下代码将 QSS 测试器应用程序中的 `QLineEdit` 的背景设置为蓝色——特定 ID 覆盖了通用控件规则。

```
QLineEdit#mylineedit {
    background: blue;
}
QLineEdit {
    background: red;
}
```

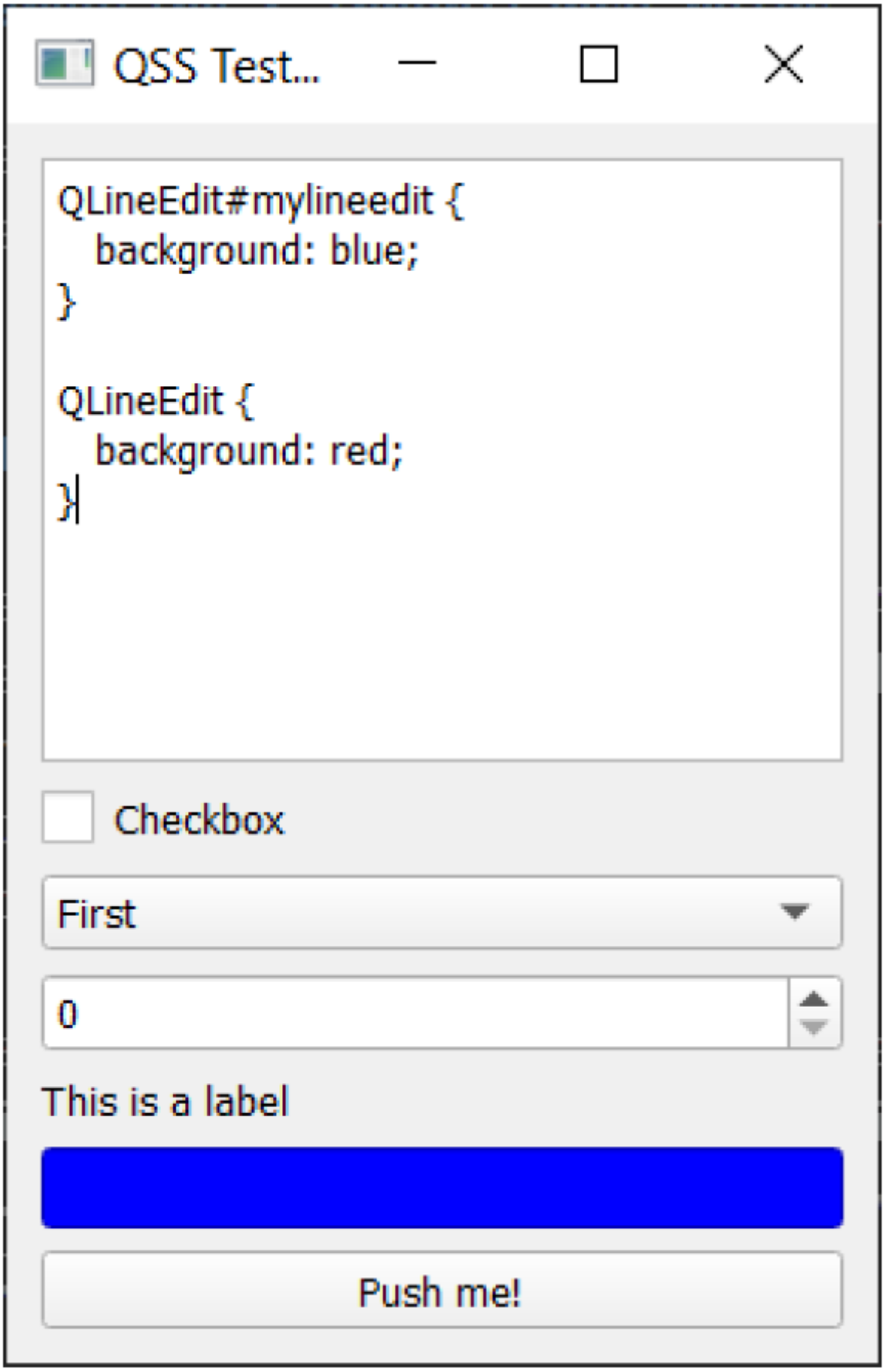


图119：特定 ID 目标优先于通用控件目标。

如果存在两个冲突的规则，则控件自己的样式表将优先于继承的样式，较近的祖先将优先于较远的祖先，例如，父母将优先于祖父母。

无继承属性

控件只受专门针对它们的规则的影响。虽然规则可以设置在父级上，但它们仍然必须引用目标控件才能对其产生影响。以以下规则为例：

```
QLineEdit {
    background: red;
}
```

如果在 `QMainWindow` 上设置，该窗口中的所有 `QLineEdit` 对象将具有红色背景（假设没有其他规则）。然而，如果设置以下内容……

```
QMainWindow {
    background: red;
}
```

…只有 `QMainWindow` 本身会被设置为红色背景。背景颜色本身不会传播到子控件。

图120：QSS 属性不会传播到子元素

> 如果子控件的背景为透明，则红色会透出来

除非被匹配规则所针对，否则控件将使用其默认系统样式值来设置每个属性。控件不会从父控件继承样式属性，即使在复合控件中也是如此，并且控件必须被规则直接针对才能受到规则的影响。

> 这与层叠样式表（CSS）不同，在层叠样式表中，元素可以从其父元素继承值。

伪选择器

到目前为止，我们已经介绍了静态样式，即使用属性来更改控件的默认外观。然而，QSS 还允许您根据动态控件状态来设置样式。一个例子就是当鼠标悬停在按钮上时出现的突出显示——突出显示有助于表明该控件已获得焦点，点击它后会做出响应。

操作样式还有许多其他用途，从可用性（突出显示数据行或特定选项卡）到数据层次结构的可视化。这些都可以通过在 QSS 中使用伪选择器来实现。伪选择器使 QSS 规则仅在特定情况下适用。

您可以对控件应用许多不同的伪选择器。其中一些（如 `:hover`）是通用的，可用于所有控件，而其他一些则是控件专用的。以下为完整列表：

伪状态	描述
`:active`	控件是活动窗口的一部分
`:adjoins-item`	QTreeView 的分支(`::branch`)与项目相邻
`:adjoins-item`	在绘制 `QabstractItemView` 的每一行时，为每隔一行设置颜色（当 `QabstractItemView.alternatingRowColors()` 为 `True` 时）
`:bottom`	位于底部，例如一个将标签显示在底部的 `QTabBar`
`:checked`	该项已选中，例如选中状态的 `QAbstractButton`
`:closable`	该项可以被关闭，例如 `QDockWidget` 具有 `QDockWidget.DockWidgetClosable` 启用
`:closed`	该项处于关闭状态，例如 `QtreeView` 中未展开的项目
`:default`	该项是默认操作，例如默认的 `QPushButton` 或 `QMenu` 中的默认操作
`:disabled`	该项已禁用
`:editable`	`QcomboBox` 可编辑

伪状态	描述
`:enabled`	该项已启用
`:exclusive`	该项属于一个专属项组，例如菜单项在专属的 `QActionGroup` 中
`:first`	该项是列表中的第一个项目，例如 `QtabBar` 中的第一个标签
`:flat`	该项为平面元素，例如一个平面的 `QpushButton` 控
`:floatable`	该项可以浮动，例如 `QDockWidget` 支持浮动功能，即 `QDockWidget.DockWidgetFloatable` 已启用
`:focus`	该项已获得输入焦点
`:has-children`	该项有子项，例如 `QTreeView` 中的项带有子项
`:has-siblings`	该项有兄弟项，例如 `QTreeView` 中的项带有兄弟项
`:horizontal`	该项采用水平布局
`:hover`	鼠标悬停在该项上
`:indeterminate`	该项处于不确定状态，例如 `QCheckBox` 或 `QRadioButton` 部分被选中
`:last`	该项是列表中的最后一项，例如 `QTabBar` 中的最后一个标签
`:left`	该项位于左侧，例如一个 `QTabBar` ，其选项卡位于左侧
`:maximized`	该项已最大化，例如已最大化的 `QMdiSubWindow`
`:middle`	该项位于列表中间，例如 `QTabBar` 中不在开头或结尾的选项卡
`:minimized`	该项已最小化，例如已最小化的 `QMdiSubWindow`
`:movable`	该项可以移动，例如 `QDockWidget` 启用了 `QDockWidget.DockWidgetMovable`
`:no-frame`	该项无边框，例如无边框的 `QSpinBox` 或 `QLineEdit`
`:non-exclusive`	该项属于非独占项组的一部分，例如菜单项在非独占 `QActionGroup` 中
`:off`	可以切换的项，这适用于处于“关闭”状态的项
`:on`	可以切换的项，这适用于处于“开启”状态的项
`:only-one`	该项是列表中的唯一一项，例如 `QTabBar` 中的一个单独标签
`:open`	该项处于展开状态，例如 `QTreeView` 中的展开项目，或带有展开菜单的 `QComboBox` 或 `QPushButton`
`:next-selected`	下一项已被选中，例如 `QTabBar` 中选中的标签位于此项的旁边
`:pressed`	该项正在通过鼠标进行点击操作
`:previous-selected`	上一个选项被选中，例如 `QTabBar` 中的选中标签旁边的标签

伪状态	描述
`:read-only`	该项被标记为只读或不可编辑，例如只读的 `QLineEdit` 或不可编辑的 `QComboBox`
`:right`	该项位于右侧，例如一个 `QTabBar`，其标签位于右侧
`:selected`	该项已被选中，例如 `QTabBar` 中的选中标签或 `QMenu` 中的选中项
`:top`	该项位于顶部，例如一个选项卡位于顶部的 `QTabBar`
`:unchecked`	该项未被选中
`:vertical`	该项采用垂直布局
`:window`	控件是一个窗口（即顶级控件）

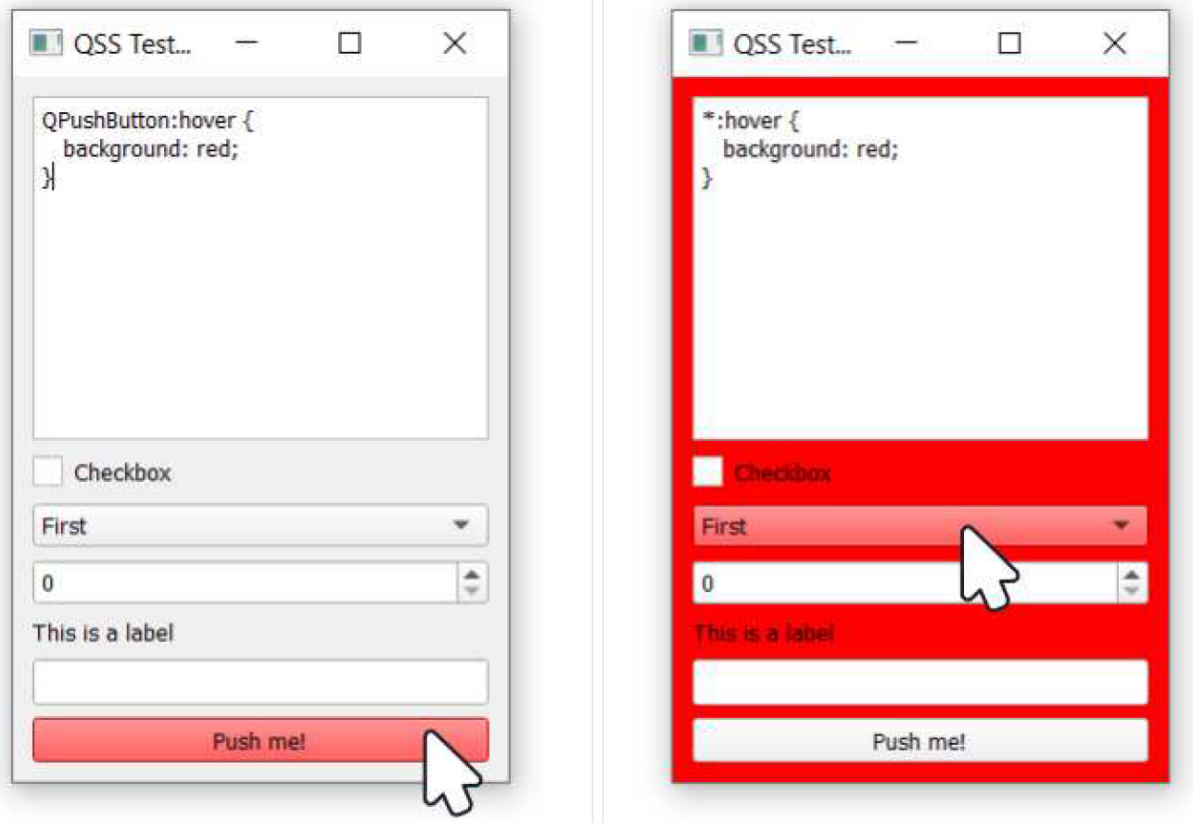
我们可以使用 QSS 测试器来查看伪选择器的运行情况。例如，以下代码将使鼠标悬停在控件上时，`QPushButton` 的背景变为红色。

```
QPushButton:hover {
    background: red;
}
```

以下代码将更改所有控件在有鼠标悬停时的背景

```
*:hover {
    background: red;
}
```

悬停在控件上意味着其所有父级控件也处于悬停状态（鼠标位于其边界框内），如下图所示。



> 图121：左侧，悬停时 QPushButton 被高亮显示。右侧，当悬停一个控件时，所有父控件也会被悬停

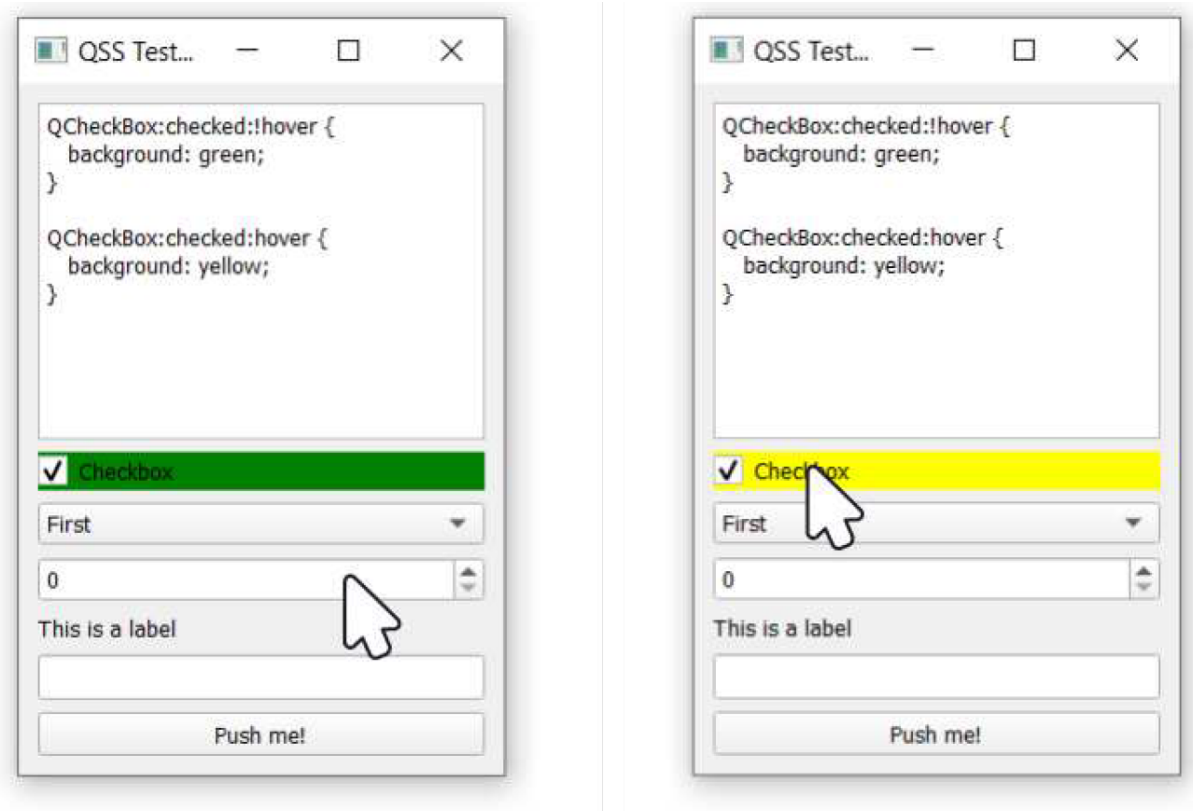
您还可以使用 `!` 来否定伪选择器。这意味着当该选择器不活动时，该规则将生效。例如以下示例...

```
QPushButton:!hover {
    background: yellow;
}
```

...当鼠标未悬停在 QPushButton 上时，会将其设置为黄色。

您还可以将多个伪选择器串联使用。例如，以下代码将设置 `QCheckBox` 在被选中且未悬停时背景为绿色，被选中且悬停时背景为黄色。

```
QCheckBox:checked:!hover {
    background: green;
}

QCheckBox:checked:hover {
    background: yellow;
}
```



> 图122：用于悬停状态的链式伪选择器

对于其他所有规则，您也可以使用 “,” 分隔符将它们串联起来，使定义的规则适用于多个情况。例如，以下代码将使复选框在被选中或悬停时背景变为绿色。

```
QCheckBox:checked, QCheckBox:hover {
    background: yellow;
}
```

样式控件的子控件

许多控件是由其他子控件或控件组合而成。QSS 提供了直接处理这些子控件的语法，因此您可以单独对子控件进行样式更改。这些子控件可以使用 `::`（双冒号）选择器来处理，后跟给定子控件的标识符。

`QComboBox` 就是此类控件的一个很好的例子。以下样式片段将自定义样式直接应用于组合框右侧的向下箭头。

```
QComboBox::drop-down {
    background: yellow;
    image: url('puzzle.png')
}
```

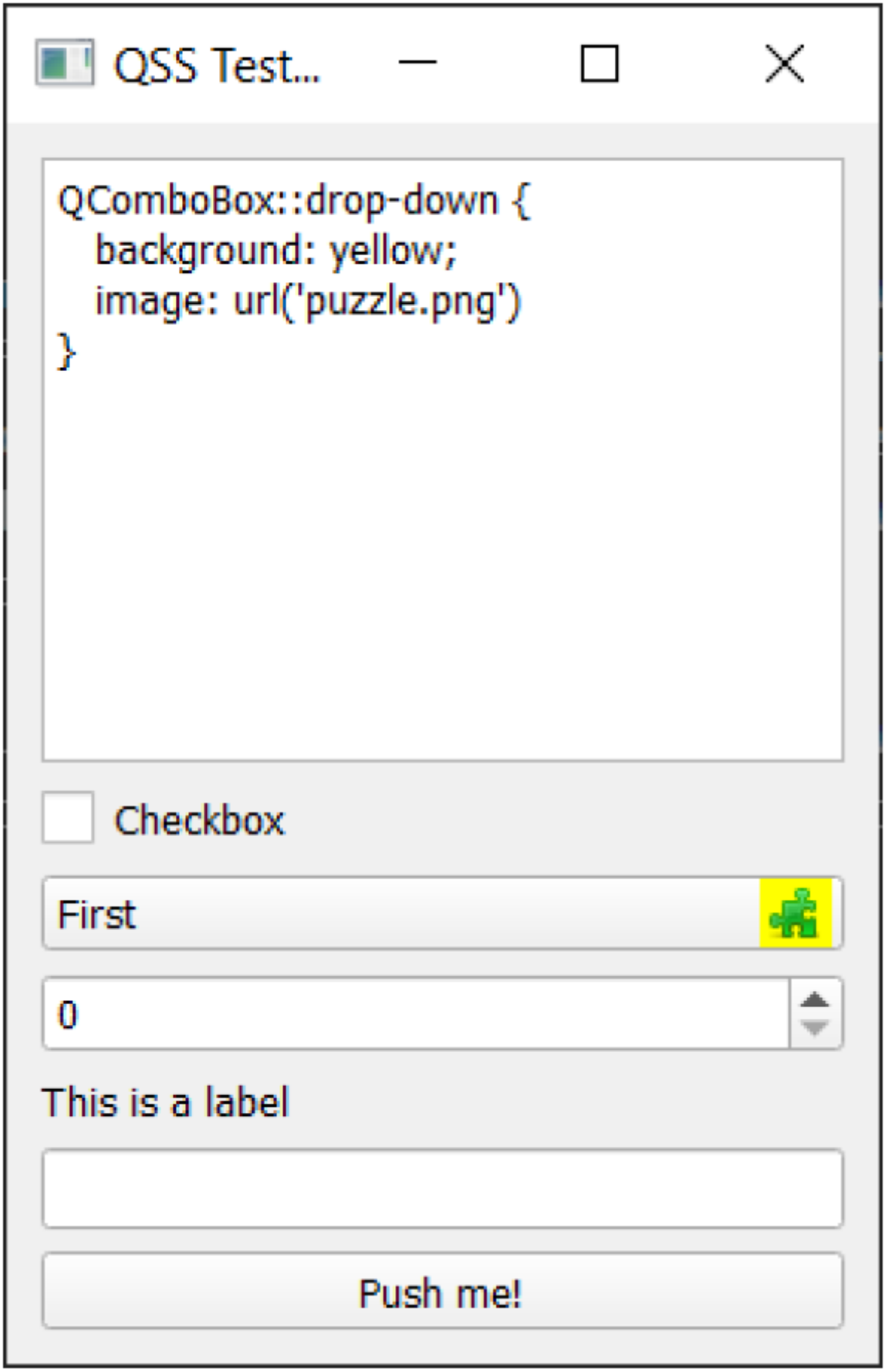


图123：使用QSS为QComboBox下拉列表设置背景和图标。

QSS 中提供了许多子控件选择器，如下所示。您会发现，其中许多仅适用于特定的控件（或控件类型）。

子控件选择器	描述
`::add-line`	在 `QScrollBar` 上用于移动到下一行的按钮
`::add-page`	滑块与添加行之间的间距，适用于 `QScrollBar`
`::branch`	`QTreeView` 的分支指示器
`::chunk`	`QProgressBar` 的进度块
`::close-button`	`QDockWidget` 或 `QTabBar` 的选项卡的关闭按钮
`::corner`	在 `QAbstractScrollArea` 中的两个滚动条之间的区域
`::down-arrow`	`QComboBox`、`QHeaderView`、`QScrollBar` 或 `QSpinBox` 的下箭头
`::down-button`	`QScrollBar` 或 `QSpinBox` 的下拉按钮
`::drop-down`	`QComboBox` 的下拉按钮
`::float-button`	`QDockWidget` 的浮动按钮
`::groove`	`QSlider` 的凹槽
`::indicator`	`QAbstractItemView`、`QCheckBox`、`QRadioButton`、可选中的 `QMenu` 项或可选中的 `QGroupBox` 的指示器
`::handle`	`QScrollBar`、`QSplitter` 或 `QSlider` 的滑块
`::icon`	`QAbstractItemView` 或 `QMenu` 的图标
`::item`	`QAbstractItemView`、`QMenuBar`、`QMenu` 或 `QStatusBar` 的项
`::left-arrow`	`QScrollBar` 的左箭头
`::left-corner`	`QTabWidget` 的左上角，例如控制 `QTabWidget` 中的左上角控件
`::menu-arrow`	带菜单的 `QToolButton` 的箭头
`::menu-button`	`QToolButton` 的菜单按钮
`::menu-indicator`	`QPushButton` 的菜单指示器
`::right-arrow`	`QMenu` 或 `QScrollBar` 的右箭头
`::pane`	`QTabWidget` 的面板（框架）
`::right-corner`	`QTabWidget` 的右上角。例如，此控件可用于控制 `QTabWidget` 中右上角控件的位置
`::scroller`	`QMenu` 或 `QTabBar` 的滚动条
`::section`	`QHeaderView` 的区域
`::separator`	`QMenu` 或 `QMainWindow` 中的分隔符

子控件选择器	描述
`::sub-line`	用于从 `QScrollBar` 中删除一行内容的按钮
`::sub-page`	`QScrollBar` 的滑块与辅助线之间的区域
`::tab`	`QTabBar` 或 `QToolBox` 的选项卡
`::tab-bar`	`QTabWidget` 的选项卡栏。此子控件仅用于控制 `QTabWidget` 中的 `QTabBar` 的位置。要设置选项卡的样式，请使用 `::tab` 子控件
`::tear`	`QTabBar` 的分页指示器
`::tearoff`	`QMenu` 的可撕拉指示器
`::text`	`QAbstractItemView` 的文本
`::title`	`QGroupBox` 或 `QDockWidget` 的标题
`::up-arrow`	`QHeaderView`（排序指示器）、`QScrollBar` 或 `QSpinBox` 的上箭头
`::up-button`	`QSpinBox` 的上箭头按钮

以下操作针对 `QSpinBox` 的上下按钮，分别将背景色设置为红色和绿色。

```
QSpinBox::up-button {
    background: green;
}
QSpinBox::down-button {
    background: red;
}
```

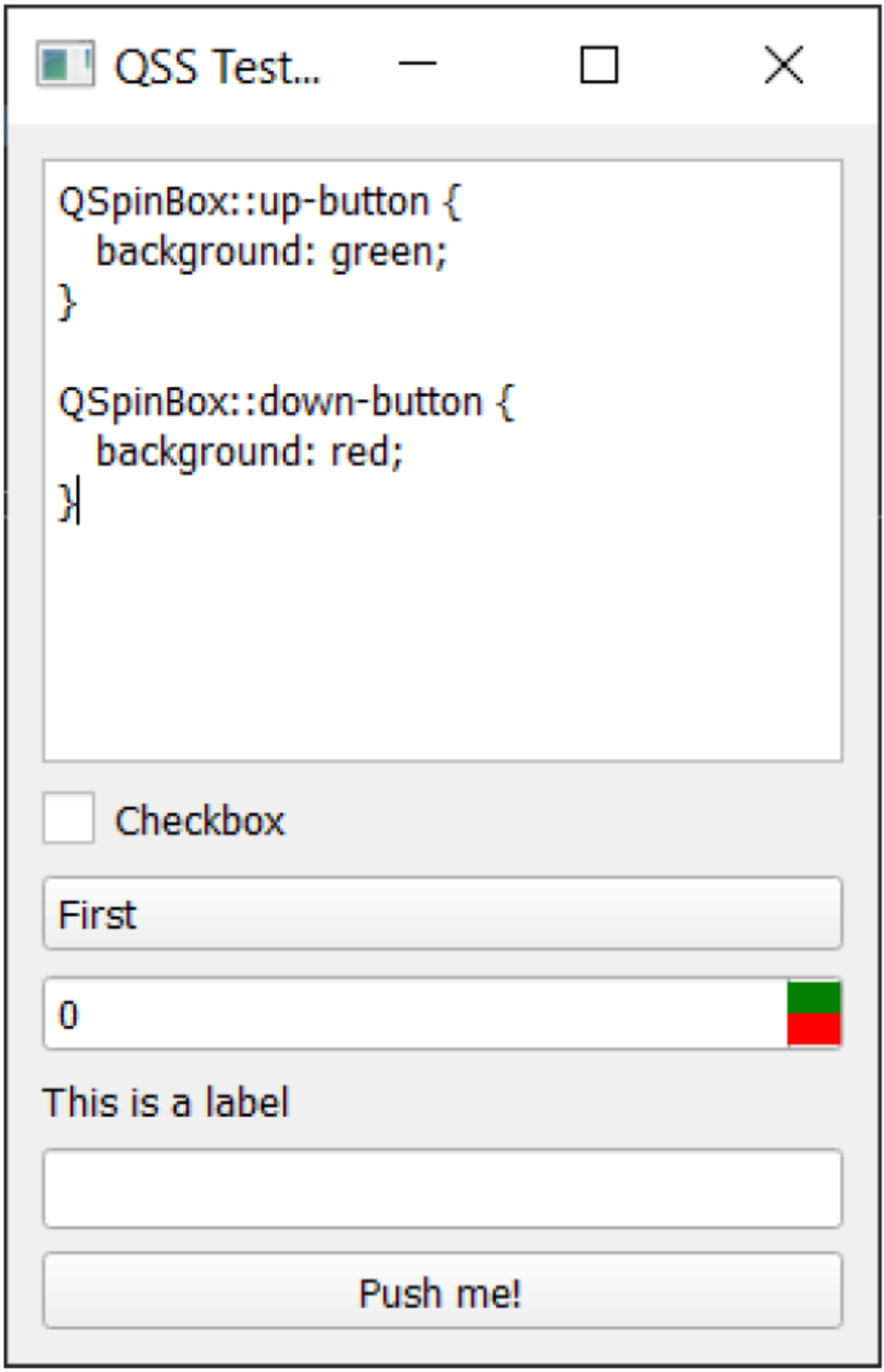


图124：设置QSpinBox上下按钮的背景颜色。

上/下按钮内的箭头也可单独作为目标。下面我们使用自定义加号和减号图标设置它们——注意我们还需要调整按钮大小以适配。

```
QSpinBox {
    min-height: 50;
```

```
}

QSpinBox::up-button {
    width: 50;
}

QSpinBox::up-arrow {
    image: url('plus.png');
}

QSpinBox::down-button {
    width: 50;
}

QSpinBox::down-arrow {
    image: url('minus.png')
}
```

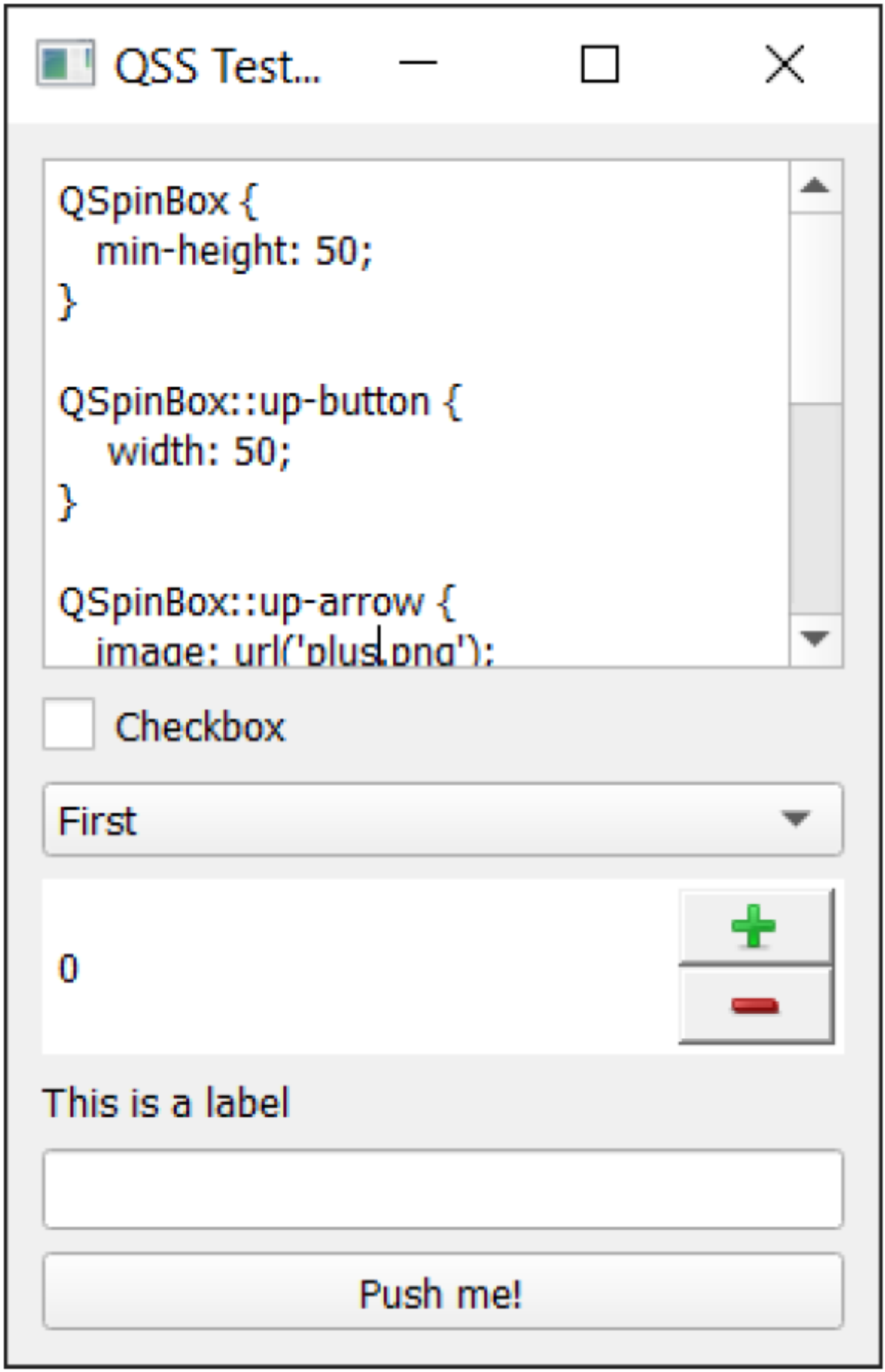


图125：设置QSpinBox上下按钮的背景颜色。

子控件伪状态

您可以像对待其他控件一样，使用伪状态来定位子控件。要做到这一点，只需将伪状态链接到控件之后即可。例如：

```
QSpinBox::up-button:hover {
    background: green;
}

QSpinBox::down-button:hover {
    background: red;
}
```

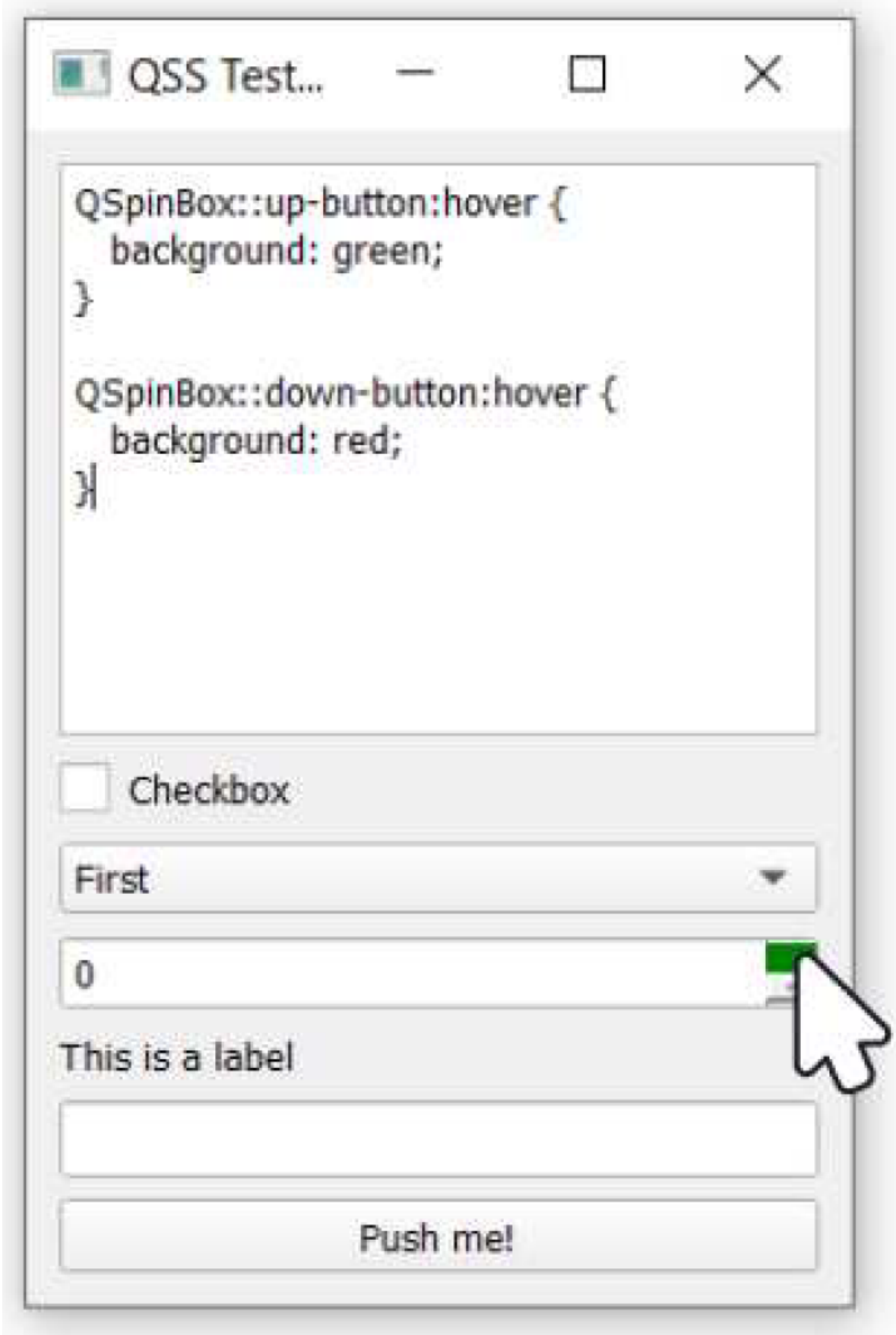


图126：将子元素选择器与伪选择器结合使用。

子控件的定位

使用 QSS，您还可以精确控制控件内部子控件的位置。这些控件允许根据其正常位置或相对于其父控件的绝对位置进行调整。下面我们将介绍这些定位方法。

属性	类型(默认)	描述
`position`	relative, absolute(relative)	使用左、右、上、下指定的偏移量是相对坐标还是绝对坐标
`bottom`	Length	如果位置是相对的（默认值），则将子控件向上移动一定偏移量；指定 `bottom: y` 与指定 `top: -y` 相同。如果位置是绝对的，则 bottom 属性指定子控件的底部边缘相对于父控件底部边缘的位置（另见 `subcontrol-origin`）
`left`	Length	如果位置设置为相对定位，则将子控件向右移动指定的偏移量（即在左侧指定额外空间）。如果位置设置为绝对定位，则指定相对于父控件左边缘的距离
`right`	Length	如果位置设置为相对定位，则将子控件向左移动给定的偏移量（即在右侧指定额外空间）。如果位置为绝对，则指定与父控件右边缘的距离
`top`	Length	如果位置设置为相对定位，则将子控件向下移动给定的偏移量（即在上侧指定额外空间）。如果位置为绝对，则指定与父控件顶部边缘的距离

默认情况下，定位是相对的。在此模式下，left、right、top 和 bottom 属性定义了要添加到相应侧面的额外间距。这意味着，左侧会将控件向右移动。这可能会让人感到困惑。

> 为了帮助您记忆，可以将这些视为“向左添加空间”等等。

```
QSpinBox {
    min-height: 100;
}

QSpinBox::up-button {
    width: 50;
}

QSpinBox::down-button {
    width: 50;
    left: 5;
}
```

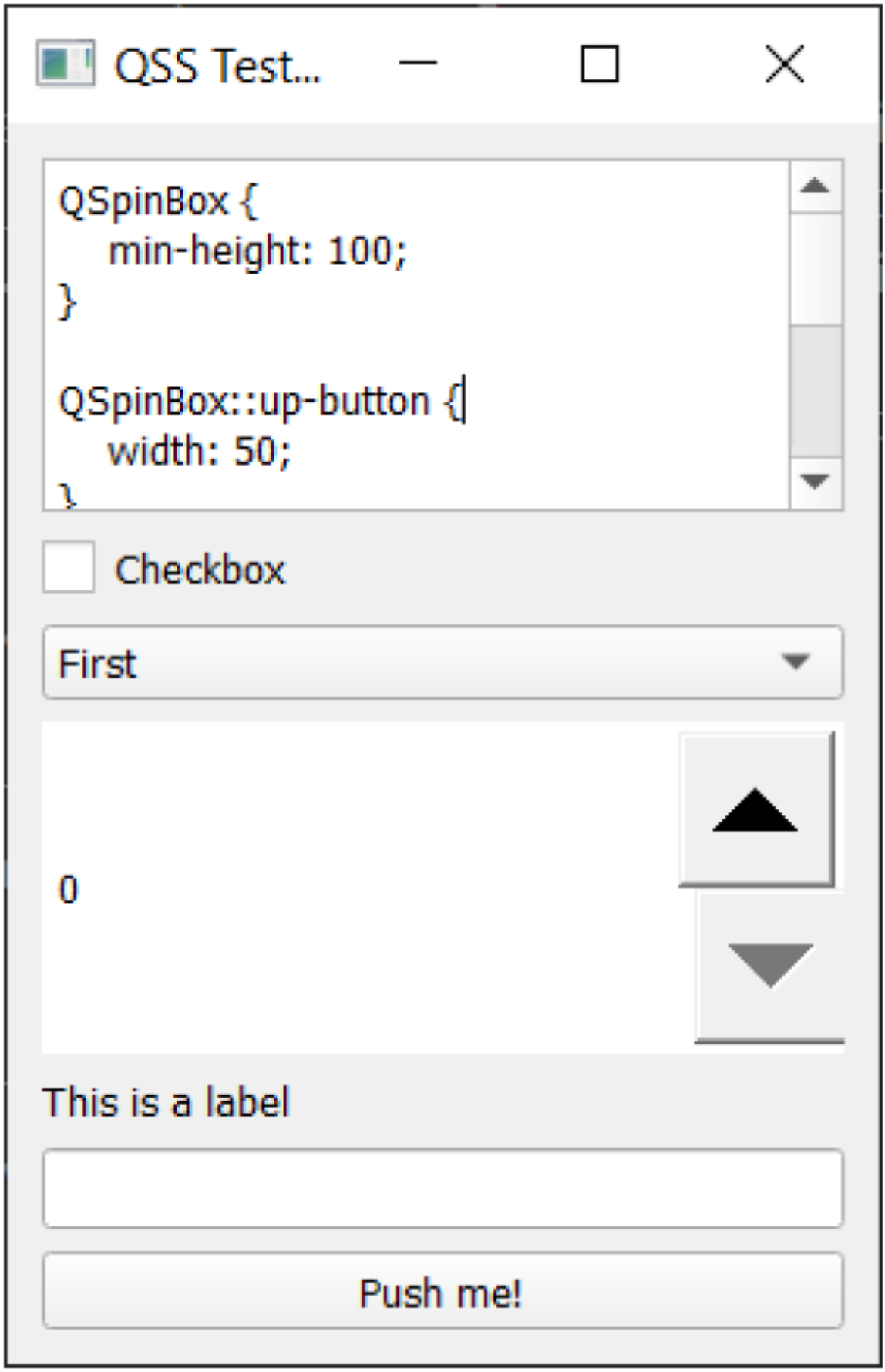


图127：使用left调整子控件的位置

当位置设置为绝对时，左、右、上、下属性定义了控件与其父级相同边缘之间的间距。因此，例如，`top: 5`，`left: 5` 将控件的位置设置为其顶部和左侧边缘距离父级顶部和左侧边缘 5 像素。

```
QSpinBox {
    min-height: 100;
}

QSpinBox::up-button {
    width: 50;
}

QSpinBox::down-button {
    position: absolute;
    width: 50;
    right: 25;
}
```

以下是使用绝对定位将下拉按钮放置在右侧25像素处的效果。

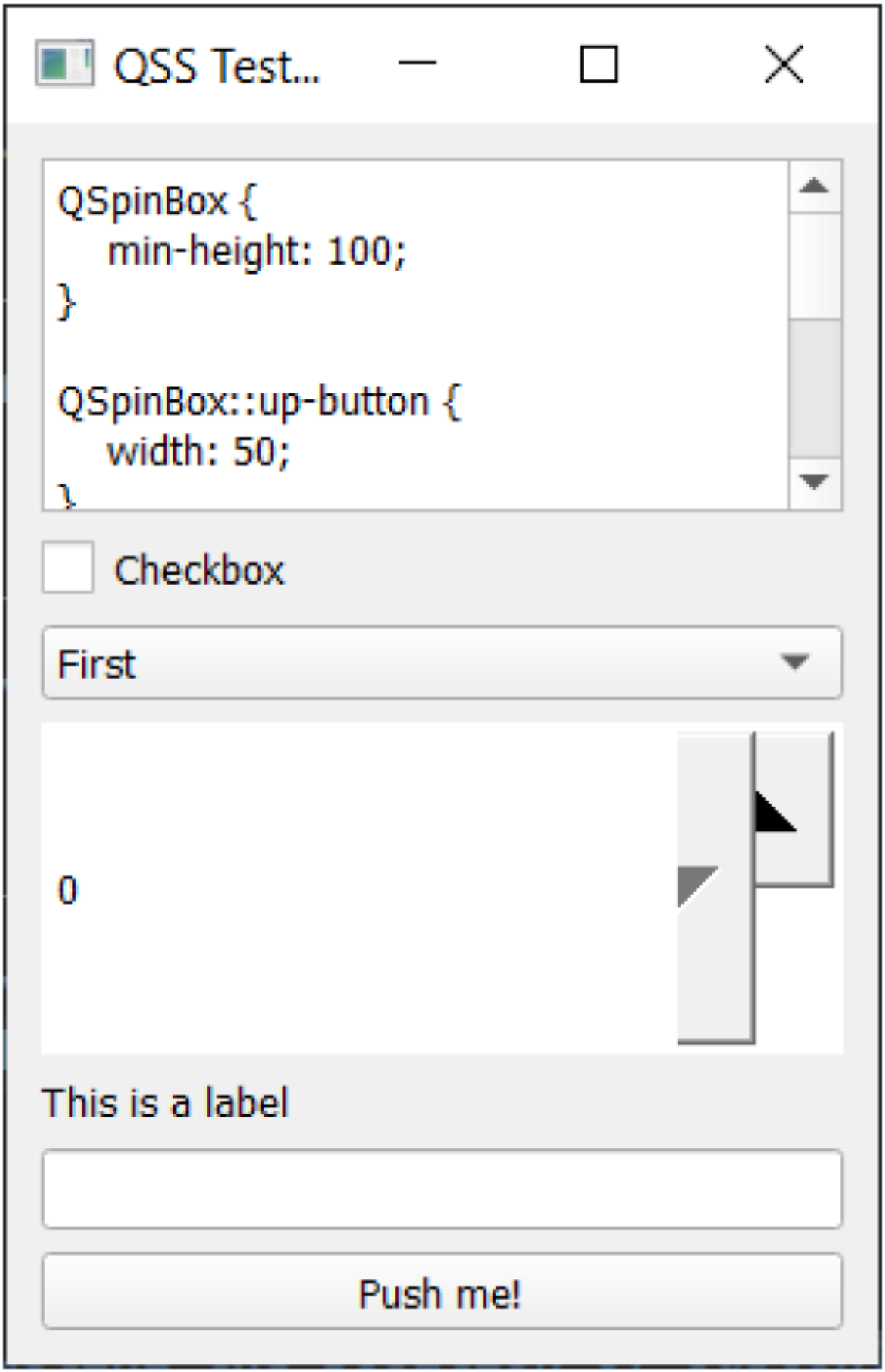


图128：调整子控件的绝对位置

这不是最实用的示例，但它展示了以这种方式定位子控件时的一个限制——你无法将子控件定位在其父控件的边界框之外。

子控件的样式

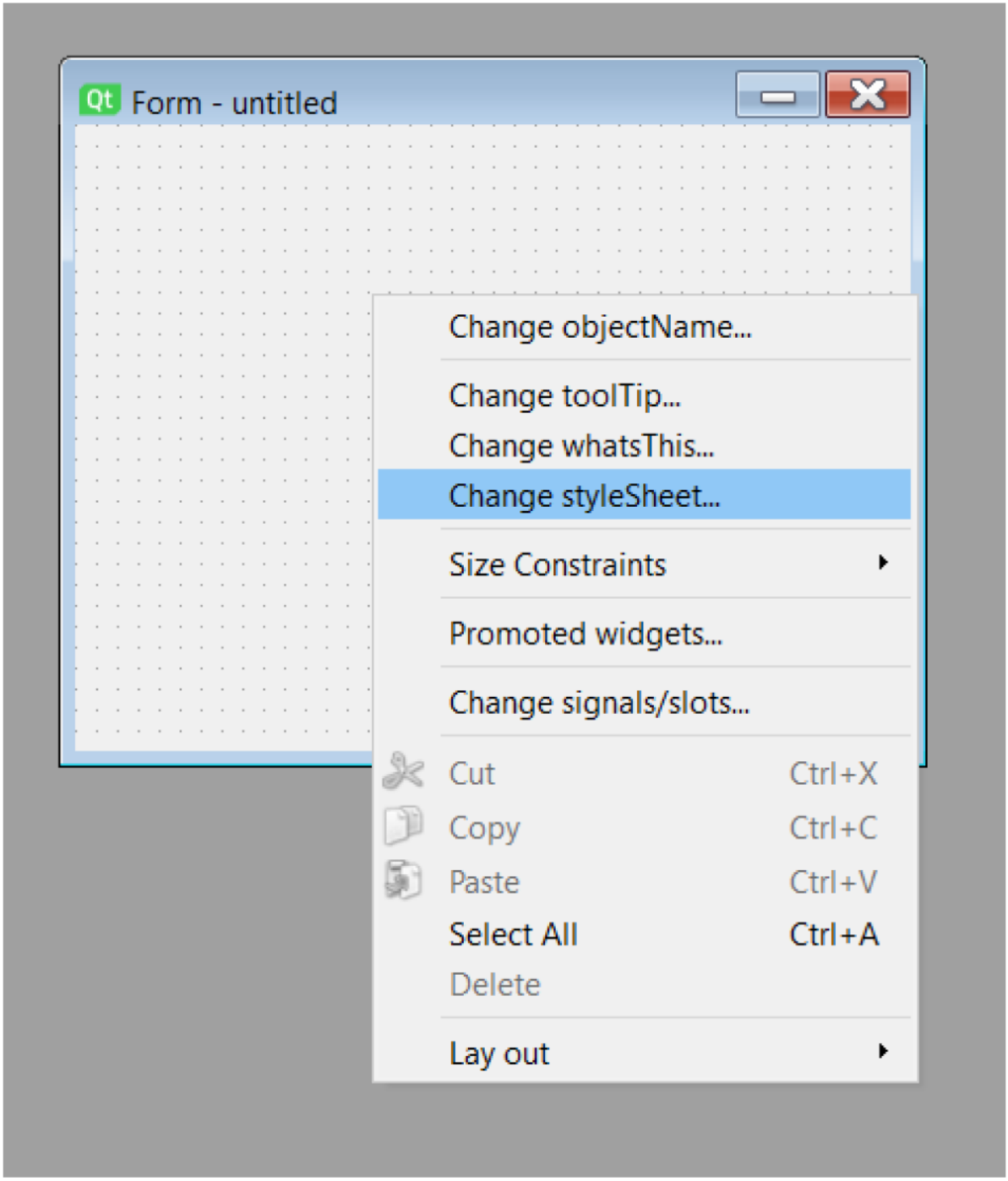
最后，还有许多 QSS 属性专门针对子控件的样式设置。这些属性如下所示——请参阅具体受影响的控件和控件的描述。

属性	类型(默认)	描述
`image`	Url+	在子控件的内容矩形中绘制的图像。设置子控件的图像属性会隐式设置子控件的宽度和高度（除非该图像是 SVG）
`image-position`	alignment	图像的对齐方式。图像的位置可以通过相对定位或绝对定位来指定。请参阅相对定位和绝对定位的说明
`height`	Length	子控件的高度。如果您想要一个高度固定的控件，请将最小高度和最大高度设置为相同的值
`spacing`	Length	控件内的内部间距
`subcontrol-origin`	Origin (padding)	父元素内子控件的起始矩形
`subcontrol-position`	Alignment	子控件在由 `subcontrol-origin` 指定的起始矩形内的对齐方式
`width`	Length	子控件的宽度。如果您想要一个宽度固定的控件，请将最小宽度和最大宽度设置为相同的值

在Qt Designer中编辑样式表

到目前为止，我们看到的示例都是通过代码将 QSS 应用到控件上的。但是，您也可以在 Qt Designer 中为控件设置样式表。

要在 Qt Designer 中为控件设置 QSS 样式表，请右键单击该控件，然后从上下文菜单中选择“Change stylesheet...”。



> 图129：访问控件的 QSS 编辑器

这将打开以下窗口，您可以在其中以文本形式输入 QSS 规则，该规则将应用于此控件（以及符合该规则的任何子控件）。

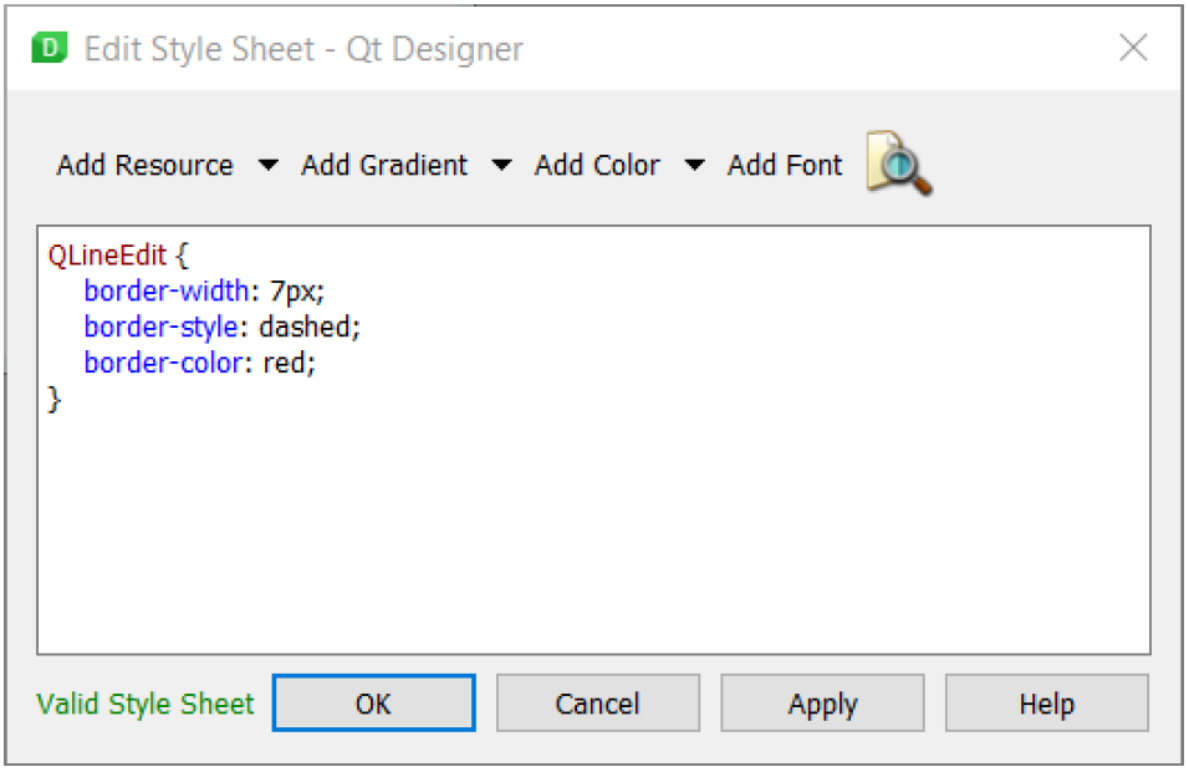


图130：Qt Designer 中的 QSS 编辑器。

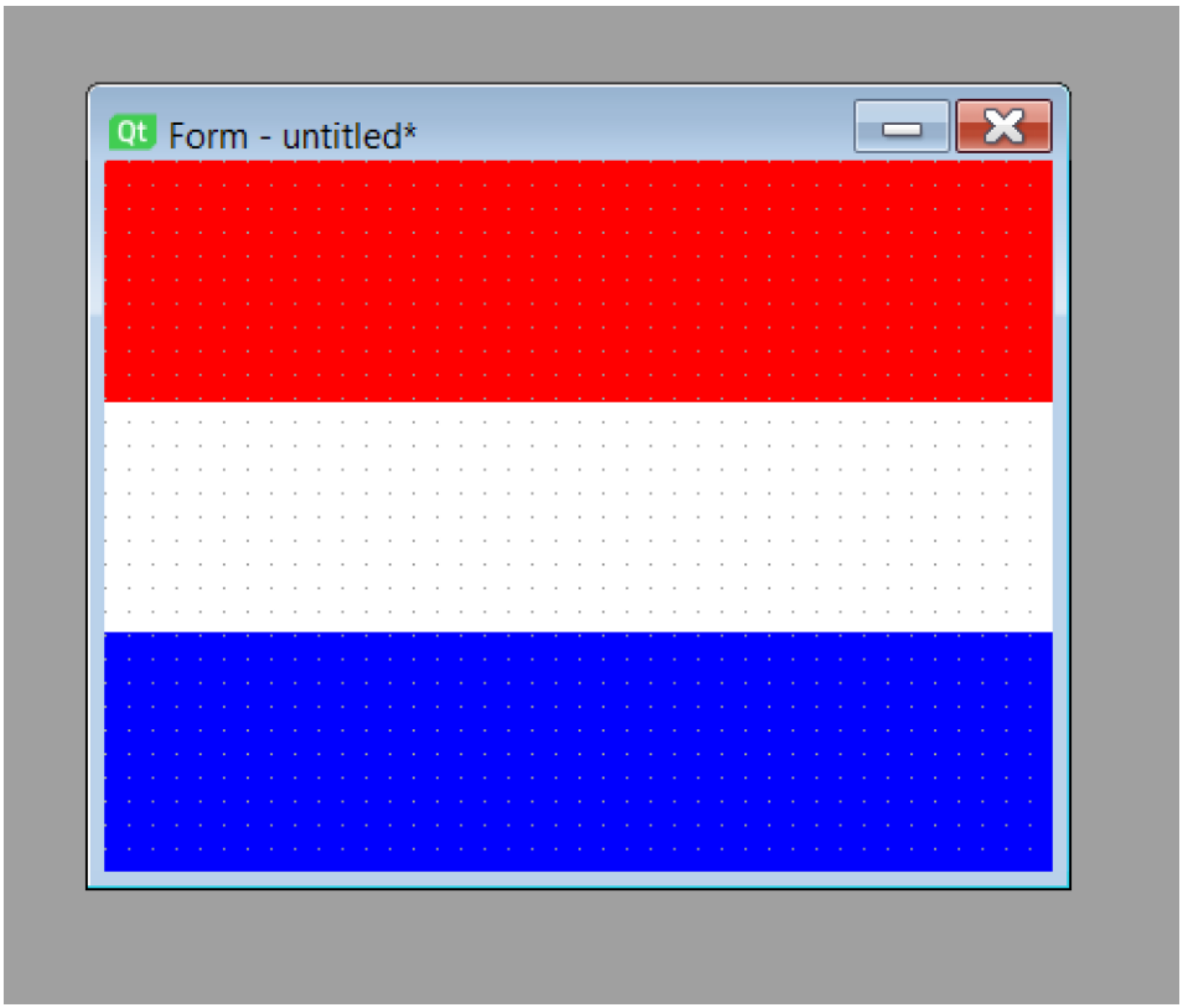
除了以文本形式输入规则外，Qt Designer 中的 QSS 编辑器还提供资源查找工具、颜色选择控件和渐变设计器。该工具（如下图所示）提供许多内置渐变，您可以将其添加到规则中，但您也可以根据需要定义自己的自定义渐变。

图131：Qt Designer 中的 QSS 渐变设计器。

梯度是通过QSS规则定义的，因此您可以将其复制并粘贴到其他地方（包括到您的代码中），以便在需要时重复使用。

Listing 94. The Dutch flag using a QSS qlineargradient rule.

```
QWidget {
    background: qlineargradient(spread:pad, x1:0, y1:0, x2:0, y2:1, stop:0
        rgba(255, 0, 0, 255), stop:0.339795 rgba(255, 0, 0, 255), stop
        :0.339799 rgba(255, 255, 255, 255), stop:0.662444 rgba(255, 255, 255,
            255), stop:0.662469 rgba(0, 0, 255, 255), stop:1 rgba(0, 0, 255,
255))
}
```



> 图132：在Qt Designer中将"荷兰国旗"QSS渐变应用于QWidget

模型视图架构

> ...只要设计得当，这些功能就能以较低的成本实现。
>
> ——丹尼斯·里奇(Dennis Ritchie，C语言之父)

当您开始使用 PyQt6 构建更复杂的应用程序时，您可能会遇到将控件与数据保持同步的问题。

存储在控件（例如简单的 `QListWidget`）中的数据在 Python 中难以操作——更改需要先获取项目、获取数据，然后再将其设置回原位。对此问题的默认解决方案是在 Python 中保留外部数据表示形式，然后将更新复制到数据和控件，或者直接根据数据重写整个控件。当您开始处理较大的数据时，这种方法可能会对应用程序的性能产生影响。

幸运的是，Qt 为此提供了一个解决方案——ModelViews(模型视图架构)。ModelViews 是标准显示控件的强大替代方案，它使用标准化的模型接口与数据源进行交互——从简单的数据结构到外部数据库。这可以隔离您的数据，意味着您可以将其保存在任何您喜欢的结构中，而视图则负责呈现和更新。

本章介绍了Qt模型视图架构的关键方面，并利用它在PyQt6中构建一个简单的桌面待办事项应用程序。

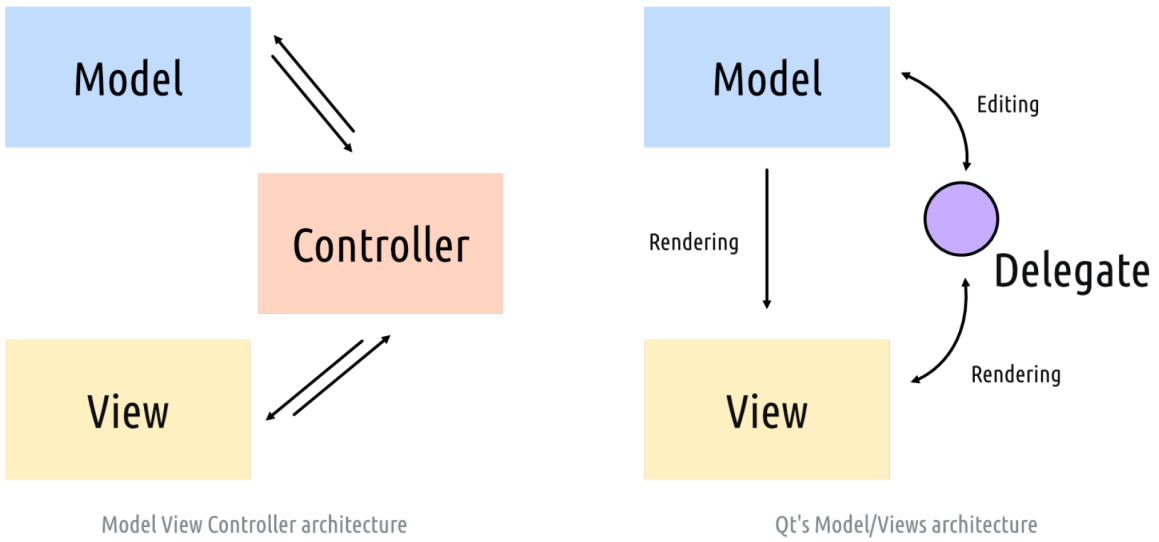
17. 模型视图架构 —— 模型视图控制器

模型-视图-控制器（Model–View–Controller, MVC）是一种用于开发用户界面的架构模式。它将应用程序划分为三个相互关联的部分，将数据的内部表示与数据如何呈现给用户以及如何从用户接收数据分离。

MVC模式将界面划分为以下组件——

- **模型**，包含应用程序正在处理的数据结构。
- **视图**是向用户展示的任何信息表示形式，无论是图形还是表格。同一数据可以有多个视图。
- **控制器**接受用户输入，将其转换为命令并应用于模型或视图。

在 Qt 领域，视图与控制器之间的区别有些模糊。Qt 通过操作系统接受来自用户的输入事件，并将这些事件委托给控件（控制器）进行处理。然而，控件也会将自己的状态滑块呈现给用户，因此它们完全属于视图。与其纠结于如何划分界限，Qt 更倾向于将视图与控制器视为一个整体。然而，控件也处理向用户显示自己的状态，将它们直接放在视图中。与其纠结于如何划分界限，不如在 Qt 中将视图和控制器合并在一起，创建一个模型/视图控制器架构——为了简单起见，称为“模型视图”。



> 图133：MVC 模型与 Qt 模型/视图架构的比较

重要的是，数据本身与数据呈现方式之间的区别得到了保留。

模型视图

模型充当数据存储与视图控制器之间的接口。模型存储数据（或对其的引用），并通过一个标准化的 API (应用程序接口)将这些数据呈现出来，视图随后消费并将其呈现给用户。多个视图可以共享相同的数据，并以完全不同的方式呈现它。

您可以使用任何“数据存储”来构建模型，例如标准的Python列表或字典，或数据库（通过Qt本身或SQLAlchemy）——这完全由您决定。

这两个部分主要负责——

1. **模型**存储数据或对数据的引用，并返回单个或记录的范围，以及相关的元数据或显示指令。

2. **视图**从模型请求数据，并将返回的结果显示在控件上。

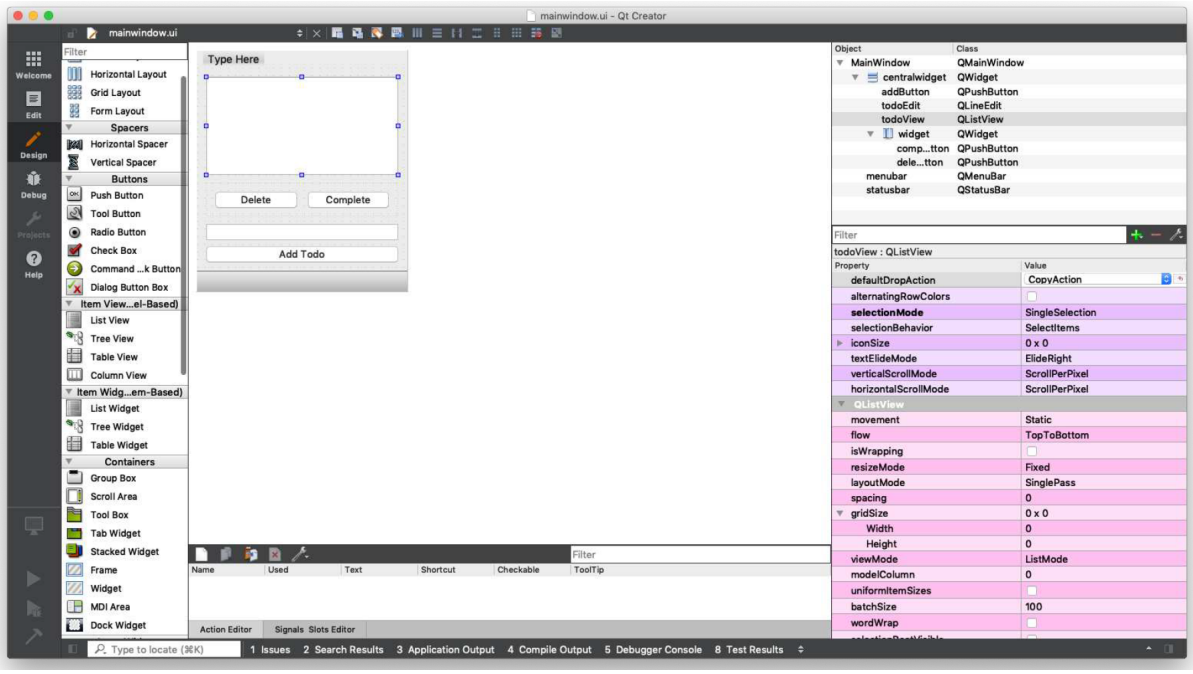
18. 一个简单的模型视图——待办事项列表

为了演示如何在实际中使用 ModelViews ，我们将实现一个非常简单的桌面待办事项列表。该列表将包含一个 `QListView` 用于显示待办事项列表，一个 `QLineEdit` 用于输入新事项，以及一组按钮用于添加、删除或标记事项为已完成。

> 本示例的文件位于源代码中。

用户界面

该简洁的用户界面使用Qt Creator进行布局，并保存为 `mainwindow.ui` 文件。该 `.ui` 文件已包含在本书的下载内容中。



> 图134：在Qt Creator中设计用户界面

请您按照之前所述，使用命令行工具将 `.ui` 文件转换为 Python 文件。

这将生成一个名为 `MainWindow.py` 的文件，其中包含我们在 Qt Designer 中设计的自定义窗口类。该文件可像普通文件一样导入到我们的应用程序代码中——一个用于显示用户界面的基本骨架应用程序示例如下所示：

Listing 95. model-views/todo_skeleton.py

```
import sys

from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt
```

```
from MainWindow import Ui_MainWindow


class MainWindow(QtWidgets.QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 🚀 **运行它吧！** 您会看到窗口弹出，不过目前还无法使用任何功能

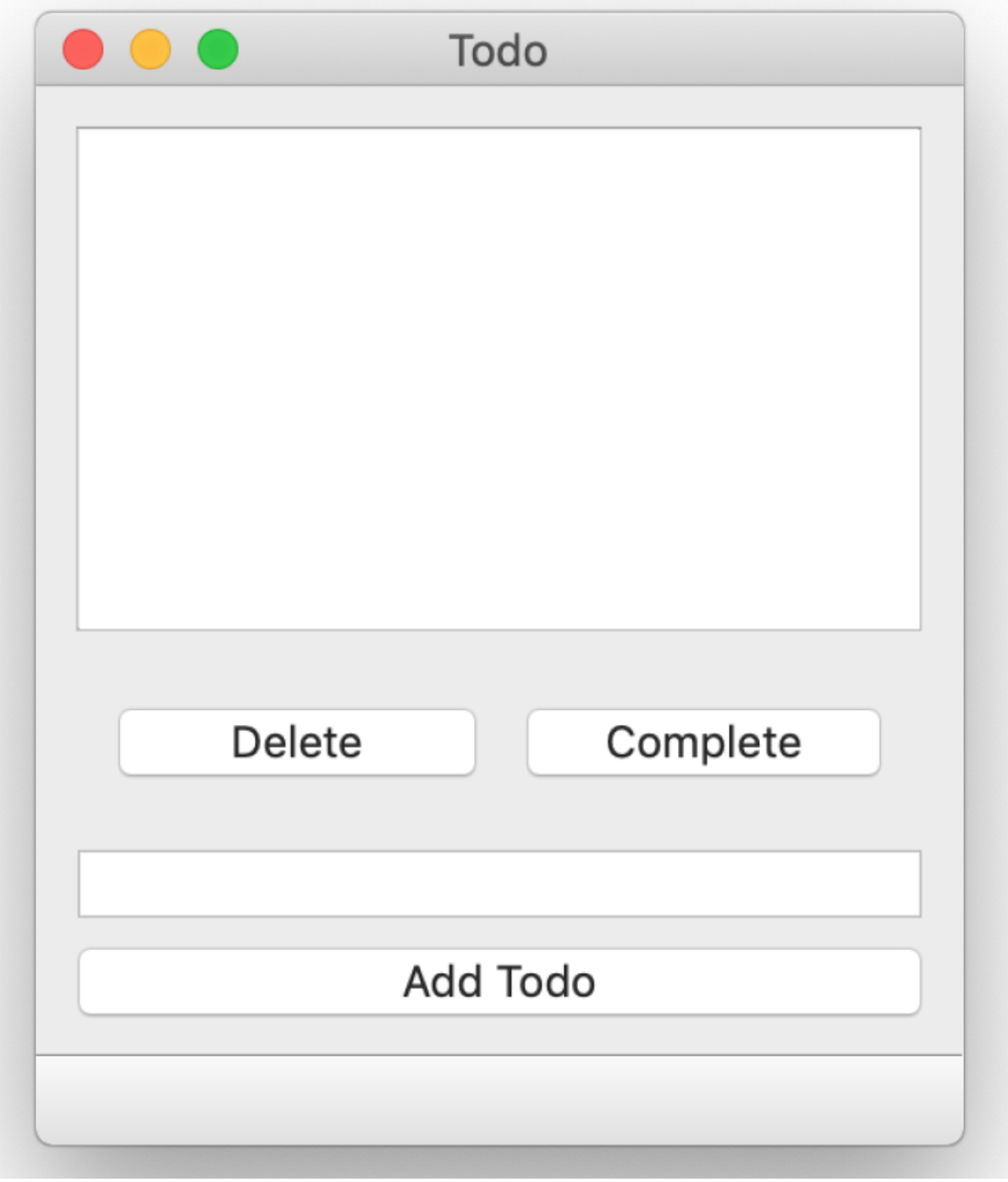


图135：主窗口

界面中的控件被赋予了下表中所示的 ID：

对象名称	类型	描述
`todoView`	`QListView`	当前待办事项列表
`todoEdit`	`QLineEdit`	创建新待办事项的文本输入框
`addButton`	`QPushButton`	创建新的待办事项，将其添加到待办事项列表中
`deleteButton`	`QPushButton`	删除当前选中的待办事项，将其从待办事项列表中移除
`completeButton`	`QPushButton`	将当前选中的待办事项标记为已完成

我们将使用这些标识符在后续步骤中与应用程序逻辑进行关联。

模型

我们通过从基础实现类继承来定义自定义模型，从而能够专注于模型中独特的部分。Qt 提供了多种不同的模型基础类，包括列表、树和表格（适用于电子表格）。

在此示例中，我们将结果显示到 `QListView` 中。与此对应的基础模型是 `QAbstractListModel`。我们的模型轮廓定义如下：

Listing 96. model-views/todo_1.py

```
class TodoModel(QAbstractListModel):
    def __init__(self, todos=None):
        super().__init__()
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            status, text = self.todos[index.row()]
            return text

    def rowCount(self, index):
        return len(self.todos)
```

`.todos` 变量是我们的数据存储。`rowcount()` 和 `data()` 方法是列表模型必须实现的标准模型方法。我们将在下面依次介绍这些方法。

.todos list

我们的模型数据存储为 `.todos`，这是一个简单的 Python 列表，其中我们将存储一个元组，格式为 `[(bool, str), (bool, str), (bool, str)]`，其中 `bool` 表示条目的完成状态，`str` 表示待办事项的文本内容。

我们在启动时将 `self.todo` 初始化为空列表，除非通过 `todos` 关键字参数传入了一个列表。

> 如果提供的 `todos` 为真（即除空列表、布尔值 `False` 或默认值 None 以外的任何值），`self.todos = todos or []` 就会将 `self.todos` 设置为这个值，否则会将其设置为空列表 `[]`。

要创建此模型的实例，我们可以简单地执行以下操作：

```
model = TodoModel() #创建一个空的待办事项列表
```

或者传递一个现有的列表：

```
todos = [(False, 'an item'), (False, 'another item')]
model = TodoModel(todos)
```

.rowcount()

`.rowcount()` 方法由视图调用，用于获取当前数据中的行数。视图需要此信息以确定其可从数据存储中请求的最大索引（`rowcount - 1`）。由于我们使用 Python 列表作为数据存储，该方法的返回值即为列表的 `len()` 值。

.data()

这是模型的核心，它处理来自视图的数据请求，并返回相应的结果。它接收两个参数 `index` 和 `role`。

`index` 是视图请求的数据的位置/坐标，可通过两种方法 `.row()` 和 `.column()` 获取每个维度中的位置。对于列表视图，列可以忽略。

> 对于我们的 `QListView`，列始终为0，可以忽略。但是，对于2D数据，例如在电子表格视图中，您需要使用此列。

`role` 是一个标志，指示视图请求的数据类型。这是因为 `.data()` 方法实际上承担的责任不仅仅是核心数据。它还处理样式信息、工具提示、状态栏等请求——基本上是任何可以由数据本身提供的信息。

`Qt.ItemDataRole.DisplayRole` 的命名有些奇怪，但这表明视图正在向我们请求"请提供要显示的数据"。`data` 还可以接收其他角色，以进行样式设置或请求"可编辑"格式的数据。

角色	值	描述
`Qt.ItemDataRole.DisplayRole`	`0`	以文本形式渲染的关键数据。 QString

角色	值	描述
`Qt.ItemDataRole.DecorationRole`	`1`	要以图标形式渲染为装饰的数据。QColor、QIcon 或 QPixmap
`Qt.ItemDataRole.EditRole`	`2`	以适合在编辑器中编辑的格式呈现的数据。QString
`Qt.ItemDataRole.ToolTipRole`	`3`	项目工具提示中显示的数据. QString
`Qt.ItemDataRole.StatusTipRole`	`4`	状态栏中显示的数据.QString
`Qt.ItemDataRole.WhatsThisRole`	`5`	在"这是什么？"模式下显示的项目数据.QString
`Qt.ItemDataRole.SizeHintRole`	`13`	用于提供给视图的项的尺寸提示. Qsize

要查看可用的所有角色列表，请参阅 Qt ItemDataRole 文档。我们的待办事项列表仅使用 `Qt.ItemDataRole.DisplayRole` 和 `Qt.ItemDataRole.DecorationRole`。

基本实现

下面的代码展示了我们在应用程序骨架中创建的基本模型，该模型包含将模型显示在界面的必要代码——尽管目前它是空的！我们将在此基础上添加模型代码和应用程序逻辑。

Listing 97. model-views/todo_1b.py

```
import sys

from PyQt6.QtCore import QAbstractListModel, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow

from MainWindow import Ui_MainWindow


class TodoModel(QAbstractListModel):
    def __init__(self, todos=None):
        super().__init__()
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            status, text = self.todos[index.row()]
            return text

    def rowCount(self, index):
        return len(self.todos)

class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.model = TodoModel()
        self.todoView.setModel(self.model)
```

```
app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

我们按照之前的方式定义 `TodoModel` 并初始化 `MainWindow` 对象。在 `MainWindow` 的 `__init__` 方法中，我们创建 `TodoModel` 的实例并将其设置为 `todo_view`。将此文件保存为 `todo.py` 并使用以下命令运行：

```
python3 todo.py
```

虽然目前还看不到太多内容，但 `QListView` 和我们的模型实际上已经开始工作了。如果您在 `MainWindow` 类中的 `TodoModel` 中添加一些默认数据，您就会看到它出现在列表中。

```
self.model = TodoModel(todos=[(False, 'my first todo')])
```

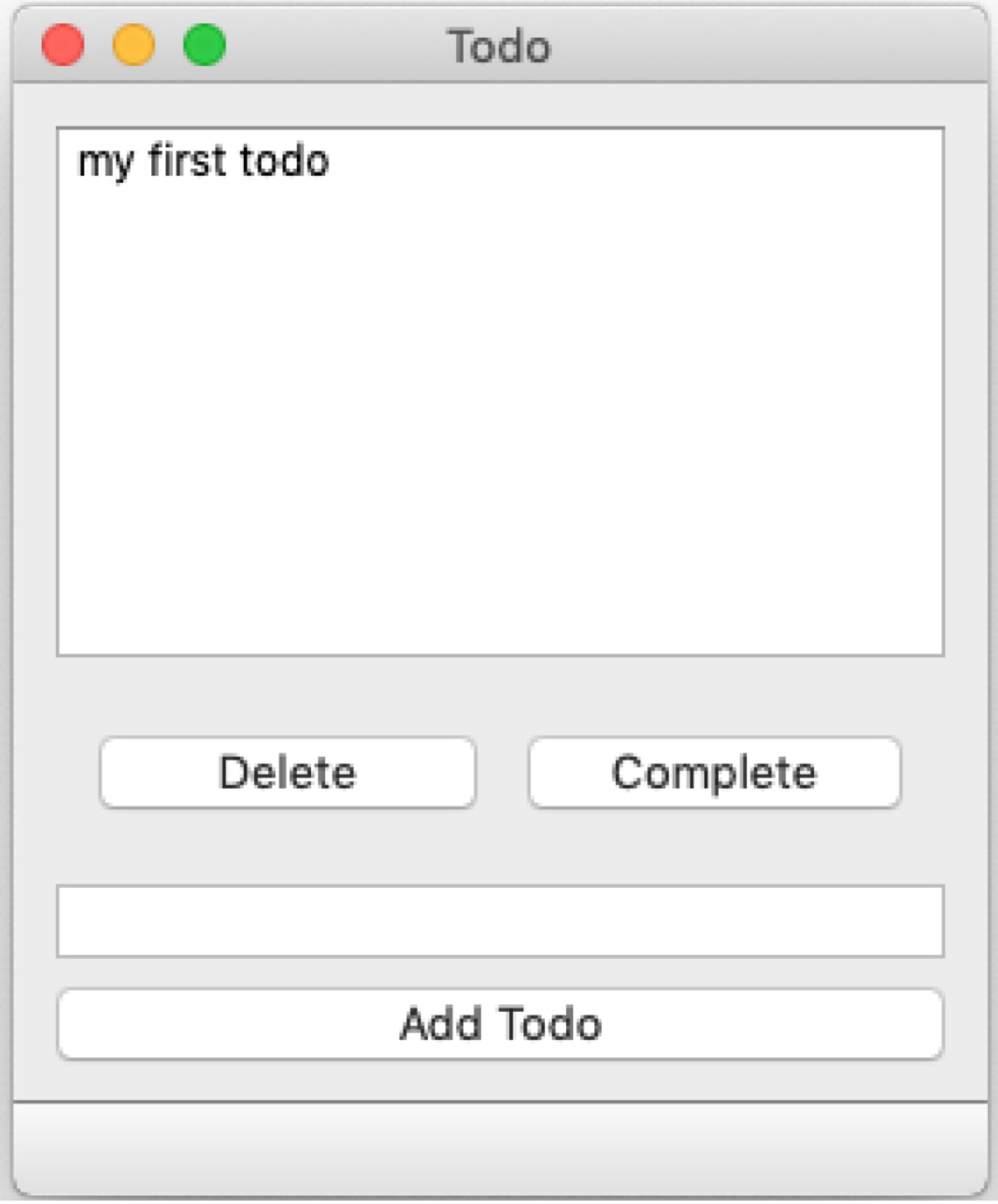


图136：QListView 显示硬编码的待办事项

您可以继续手动添加项目，它们将按顺序显示在 `QListView` 中。接下来，我们将实现从应用程序内部添加项目的功能。

首先在 `MainWindow` 上创建一个名为 `add` 的新方法。这是我们的回调函数，它将负责将输入中的当前文本作为新待办事项添加。将此方法连接到 `__init__` 块末尾的 `addButton.pressed` 信号。

Listing 98. model-views/todo_2.py

```
class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.model = TodoModel()
        self.todoView.setModel(self.model)
        # 连接到按钮.
```

```
        self.addButton.pressed.connect(self.add)

    def add(self):
        """
        将一项内容添加到我们的待办事项列表中，从 QLineEdit .todoEdit 中获取文本，然后清除
它。
        """
        text = self.todoEdit.text()
        # 清楚字符串末尾的尾随空格
        text = text.strip()
        if text: # 不要添加空字符串
            # 通过模型访问该列表
            self.model.todos.append((False, text))
            # 触发刷新
            self.model.layoutChanged.emit() #1
            # 清空输入(input)
            self.todoEdit.setText("")
```

1. 这里，我们发出一个模型信号 `.layoutChanged`，让视图知道数据的形状已经改变。这会触发整个视图的刷新。如果您省略了这一行，待办事项仍然会被添加，但 `QListView` 不会更新。

如果只是数据发生了改变，但行/列数未受影响，则可以使用 `.dataChanged()` 信号。这也会使用左上角和右下角的位置来定义数据中发生改变的区域，以避免重新绘制整个视图。

连接其他操作

现在，我们可以连接按钮的其余信号，并添加辅助函数来执行删除和完成操作。我们像之前一样将按钮信号添加到 `__init__` 块中。

```
        self.addButton.pressed.connect(self.add)
        self.deleteButton.pressed.connect(self.delete)
        self.completeButton.pressed.connect(self.complete)
```

然后定义一个新的 `delete` 方法，如下所示 ——

Listing 99. model-views/todo_3.py

```
class MainWindow(QMainWindow, Ui_MainWindow):

    def delete(self):
        indexes = self.todoView.selectedIndexes()
        if indexes:
            # 索引是一个单选模式下的单项列表
            index = indexes[0]
            # 删除该项并刷新
            del self.model.todos[index.row()]
            self.model.layoutChanged.emit()
            # 清除选中项（因其已不再有效）
            self.todoView.clearSelection()
```

我们使用 `self.todoView.selectedIndexes` 来获取索引（实际上是一个单项列表，因为我们处于单选模式），然后使用 `.row()` 作为索引进入我们模型上的待办事项列表。我们使用 Python 的 `del` 操作符删除索引项，然后触发 `layoutChanged` 信号，因为数据的形状已经发生了改变。

最后，我们清除活动选择，因为您选择的项现在已不存在，且该位置本身可能已超出范围（如果您选择了最后一个项）。

> 您可以让操作更智能一些，改为选择列表中相邻的项。

`complete` 方法如下所示 —

Listing 100. model-views/todo_4.py

```
class MainWindow(QMainWindow, Ui_MainWindow):
    def complete(self):
        indexes = self.todoView.selectedIndexes()
        if indexes:
            index = indexes[0]
            row = index.row()
            status, text = self.model.todos[row]
            self.model.todos[row] = (True, text)
            # .dataChanged 方法接受左上角和右下角坐标，且这两个坐标相等。
            # 对于单一选择。
            self.model.dataChanged.emit(index, index)
            # 清除选中内容（因其已不再有效）。
            self.todoView.clearSelection()
```

这与删除操作使用相同的索引，但这次我们从模型 `.todos` 列表中获取该项，然后将状态替换为 `True`。

> 我们必须进行这种查找替换操作，因为我们的数据以Python元组的形式存储，元组是不可变的。

与标准 Qt 控件的关键区别在于，我们直接对数据进行更改，只需通知 Qt 发生了某些更改即可——控件状态的更新由滑块自动完成。

使用 DecorationRole

如果您运行该应用程序，您会发现添加和删除功能均可正常使用，但尽管完成项的功能正常，视图中却没有相应的显示提示。我们需要更新模型，为视图提供一个指示器，用于显示项已完成的状态。更新后的模型如下所示。

Listing 101. model-views/todo_5.py

```
import os
```

```
basedir = os.path.dirname(__file__)

tick = QImage(os.path.join(basedir, "tick.png"))

class TodoModel(QAbstractListModel):
    def __init__(self, *args, todos=None, **kwargs):
        super(TodoModel, self).__init__(*args, **kwargs)
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            status, text = self.todos[index.row()]
            return text

        if role == Qt.ItemDataRole.DecorationRole:
            status, text = self.todos[index.row()]
            if status:
                return tick

     def rowCount(self, index):
        return len(self.todos)
```

> 我们使用之前介绍的 `basedir` 技术加载图标，以确保无论脚本在何处运行，路径都是正确的。我使用的图标来自 p.yusukekamiyamane 的Fugue图标集。

我们使用一个勾选图标 `tick.png` 来表示已完成的项目，并将该图标加载到名为 `tick` 的 `QImage` 对象中。在模型中，我们实现了对 `Qt.ItemDataRole.DecorationRole` 的处理程序，该处理程序会为状态为 `True`（表示已完成）的行返回勾选图标。

> 除了图标，您还可以返回一种颜色，例如：`QtGui.QColor('green')`，它将被绘制为实心正方形。

运行该应用后，您现在应该能够将项目标记为已完成。

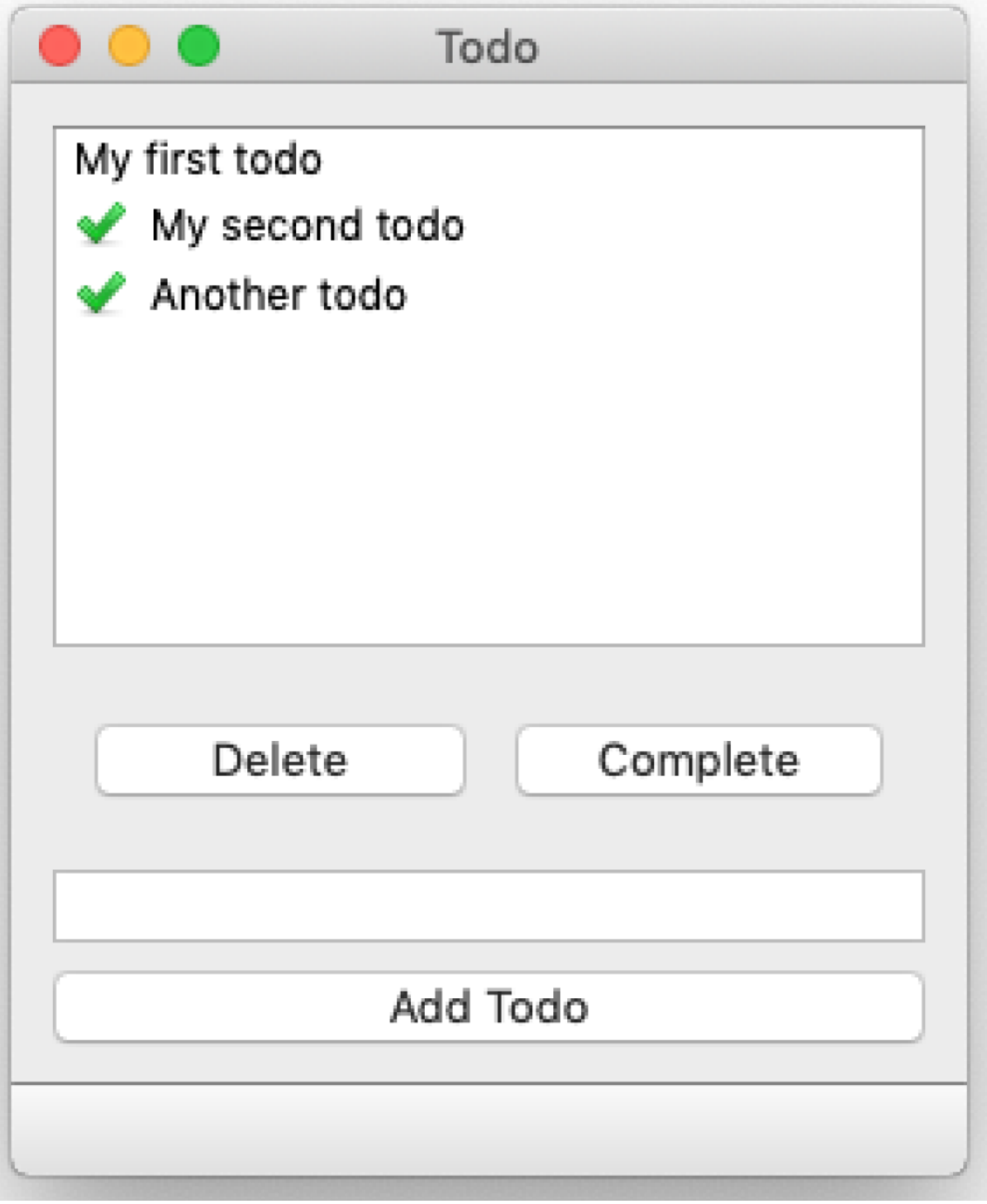


图137：全部完成

持久化数据存储

我们的待办事项应用程序运行良好，但它有一个致命缺陷——它会在您关闭应用程序后立即忘记你的待办事项。虽然认为自己没有事情可做可能会带来短暂的平静感，但从长远来看，这可能不是一个好主意。

解决方案是实现某种持久化数据存储。最简单的方法是使用简单的文件存储，我们在启动时从JSON或Pickle文件中加载项，并写回任何更改。

为此，我们在 `MainWindow` 类中定义两个新方法—— `load` 和 `save` 。这两个方法分别从名为 `data.json` 的 JSON 文件（如果存在，则忽略文件不存在时的错误）中加载数据到 `self.model.todos` ，并将当前的 `self.model.todos` 写入到同一个文件中。

Listing 102. model-views/todo_6.py

```
    def load(self):
        try:
            with open("data.json", "r") as f:
                self.model.todos = json.load(f)
        except Exception:
            pass

    def save(self):
        with open("data.json", "w") as f:
            data = json.dump(self.model.todos, f)
```

要持久化数据的更改，我们需要在任何修改数据的方法末尾添加 `.save()` 处理程序，并在模型创建后在 `__init__` 块中添加 `.load()` 处理程序。

最终的代码如下所示——

Listing 103. mode-views/todo_complete.py

```
import json
import os
import sys

from PyQt6.QtCore import QAbstractListModel, Qt
from PyQt6.QtGui import QImage
from PyQt6.QtWidgets import QApplication, QMainWindow
from MainWindow import Ui_MainWindow

basedir = os.path.dirname(__file__)

tick = QImage(os.path.join(basedir, "tick.png"))


class TodoModel(QAbstractListModel):
    def __init__(self, todos=None):
        super().__init__()
        self.todos = todos or []

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            status, text = self.todos[index.row()]
            return text

        if role == Qt.ItemDataRole.DecorationRole:
            status, text = self.todos[index.row()]
            if status:
                return tick

    def rowCount(self, index):
        return len(self.todos)


class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
```

```
        self.model = TodoModel()
        self.load()
        self.todoView.setModel(self.model)
        self.addButton.pressed.connect(self.add)
        self.deleteButton.pressed.connect(self.delete)
        self.completeButton.pressed.connect(self.complete)

    def add(self):
        """
        将一项内容添加到我们的待办事项列表中，从 QLineEdit .todoEdit 中获取文本，然后清除
它。
        """
        text = self.todoEdit.text()
        # 清楚字符串末尾的尾随空格
        text = text.strip()
        if text: # 不要添加空字符串
            # 通过模型访问该列表
            self.model.todos.append((False, text))
            # 触发刷新
            self.model.layoutChanged.emit() #1
            # 清空输入(input)
            self.todoEdit.setText("")
            self.save()

    def delete(self):
        indexes = self.todoView.selectedIndexes()
        if indexes:
            # 索引是一个单选模式下的单项列表
            index = indexes[0]
            # 删除该项并刷新
            del self.model.todos[index.row()]
            self.model.layoutChanged.emit()
            # 清除选中项（因其已不再有效）
            self.todoView.clearSelection()
            self.save()

    def complete(self):
        indexes = self.todoView.selectedIndexes()
        if indexes:
            index = indexes[0]
            row = index.row()
            status, text = self.model.todos[row]
            self.model.todos[row] = (True, text)
            # .dataChanged 方法接受左上角和右下角坐标，且这两个坐标相等。
            # 对于单一选择。
            self.model.dataChanged.emit(index, index)
            # 清除选中内容（因其已不再有效）。
            self.todoView.clearSelection()
            self.save()

    def load(self):
        try:
            with open("data.json", "r") as f:
                self.model.todos = json.load(f)
        except Exception:
            pass
```

```
    def save(self):
        with open("data.json", "w") as f:
            data = json.dump(self.model.todos, f)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

如果您的应用程序中的数据有可能变得庞大或复杂，您可能更倾向于使用实际的数据库来存储它。Qt 提供了与 SQL 数据库交互的模型，我们稍后将详细介绍。

> 另一个有趣的 `QListView` 示例请参见我的 媒体播放器应用程序示例。该示例使用Qt内置的 `QMediaPlaylist` 作为数据存储，其内容通过 `QListView` 进行显示。

19. 使用numpy和pandas处理模型视图中的表格数据

在上一节中，我们介绍了模型视图（Model View）架构。然而，我们仅涉及了模型视图中的一种——`QListView`。在PyQt6中，还有另外两种模型视图可用——`QTableView` 和 `QTreeView`，它们分别提供表格（类似Excel）和树形（类似文件目录浏览器）视图，且均使用相同的 `QStandardItemModel`。

在本节中，我们将探讨如何使用PyQt6中的 `QTableView` 来建模数据、格式化显示值以及添加条件格式化。

您可以使用模型视图与任何数据源配合使用，只要您的模型能够以Qt能够识别的格式返回数据。在Python中处理表格数据为我们加载和处理数据提供了多种可能性。我们将从一个简单的嵌套列表开始，然后逐步整合您的Qt应用程序与流行的 numpy 和 pandas 库。这将为您构建以数据为中心的应用程序奠定坚实的基础。

`QTableView` 入门指南

`QTableView` 是一个 Qt 视图控件，以类似电子表格的表格视图显示数据。与模型视图架构中的所有控件一样，它使用单独的模型向视图提供数据和呈现信息。模型中的数据可以根据需要进行更新，视图会收到这些更改的通知，从而重新绘制/显示更改。通过自定义模型，可以对数据的呈现方式进行大量控制。

要使用该模型，我们需要一个基本的应用程序结构和一些示例数据。下文展示了一个简单的示例，其中定义了一个自定义模型，并使用一个简单的嵌套列表作为数据存储。

Listing 104. tableview_demo.py

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

```
class TableModel(QtCore.QAbstractTableModel):
    def __init__(self, data):
        super().__init__()
        self._data = data

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            # 请参见下文的嵌套列表数据结构。
            # .row() 方法用于访问外部列表中的元素。
            # .column() 方法用于访问子列表中的元素。
            return self._data[index.row()][index.column()]

    def rowCount(self, index):
        # 外部列表的长度。
        return len(self._data)

    def columnCount(self, index):
        # 以下代码取第一个子列表，并返回其长度（仅在所有行长度相等时有效）
        return len(self._data[0])


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.table = QtWidgets.QTableView()

        data = [
            [4, 1, 3, 3, 7],
            [9, 1, 5, 3, 8],
            [2, 1, 5, 3, 9],
        ]

        self.model = TableModel(data)
        self.table.setModel(self.model)
        self.setCentralWidget(self.table)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

与之前的模型视图示例一样，我们创建 `QTableView` 控件，然后创建自定义模型的实例（我们编写该模型以接受数据源作为参数），然后将模型设置到视图上。这就是我们需要做的全部工作——视图控件现在使用模型来获取数据，并确定如何绘制数据。

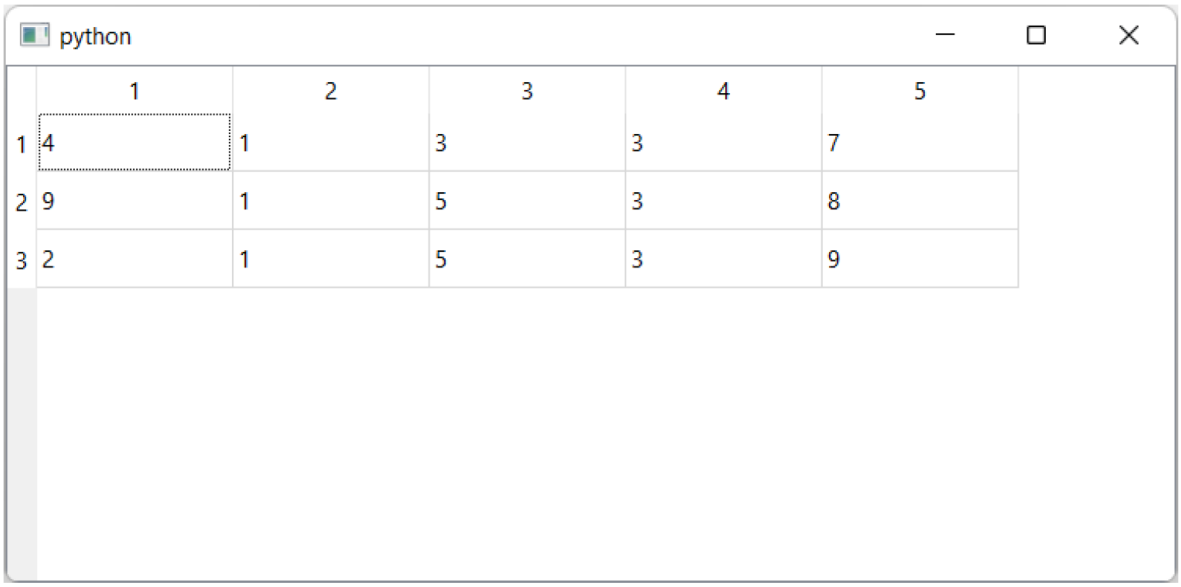


图138：基本表格示例

嵌套 `list` 作为二维数据存储结构

对于一张表格，您需要一个二维数据结构，包含列和行。如上例所示，您可以使用嵌套Python列表来建模一个简单的二维数据结构。我们将花一点时间来查看这个数据结构及其局限性，如下所示——

```
table = [
    [4, 1, 3, 3, 7],
    [9, 1, 5, 3, 8],
    [2, 1, 5, 3, 9],
]
```

嵌套列表是一个“值的列表集合”——一个外层列表包含多个子列表，而这些子列表本身又包含值。由于这种结构，要访问单个值（或“单元格”），必须进行两次索引操作：首先返回一个内部 `list` 对象，然后再次对该 `list` 进行索引。

典型的布局是外层列表存放行，每个嵌套列表存放列的值。采用这种布局时，索引操作首先按行索引，然后按列索引——因此上述示例是一个3行5列的表格。值得注意的是，这种布局与源代码中的视觉布局相匹配。

对该表的第一次索引操作将返回一个嵌套子列表——

```
row = 2
col = 4

>>> table[row]
[2, 1, 5, 3, 9]
```

然后再次对其进行索引以返回值 —

```
>>> table[row][col]
9
```

请注意，使用此类结构无法直接返回整列数据，您需要遍历所有行。不过，您当然可以根据实际需求，将索引与列的关联关系进行调整，即根据是否更适合按列或按行访问数据，将第一个索引作为列进行使用。

```
table = [
    [4, 9, 2],
    [1, 1, 1],
    [3, 5, 5],
    [3, 3, 2],
    [7, 8, 9],
]

row = 4 # 反转
col = 2 # 反转

>>> table[col]
[3, 5, 5]

>>> table[col][row]
9
```

> 该数据结构并未强制要求行或列的长度一致——一行可以包含5个元素，另一行则可能包含200个元素。然而，不一致的情况会导致表格视图出现错误。若您需要处理大型或复杂的数据表，请参阅后文的替代数据存储方案。

接下来，我们将更详细地探讨我们的自定义 `TableModel`，并了解它如何与这个简单的数据结构配合使用，以显示相应的值。

编写自定义的 `QAbstractTableModel`

在模型视图架构中，模型负责提供用于视图显示的数据和展示元数据。为了在数据对象和视图之间进行交互，我们需要编写自己的自定义模型，该模型能够理解数据的结构。

要编写自定义模型，我们可以创建 `QAbstractTableModel` 的子类。自定义表格模型仅需实现三个方法：`data`、`rowCount` 和 `columnCount`。第一个方法返回表格中指定位置的数据（或呈现信息），后两个方法必须返回数据源维度的单个整数值。

```
class TableModel(QtCore.QAbstractTableModel):

    def __init__(self, data):
        super(TableModel, self).__init__()
        self._data = data

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            # 请参见下文的嵌套列表数据结构。
            # .row() 方法用于访问外部列表中的元素。
            # .column() 方法用于访问子列表中的元素。
            return self._data[index.row()][index.column()]

    def rowCount(self, index):
```

```
        # 外部列表的长度。
        return len(self._data)

    def columnCount(self, index):
        # 以下代码取第一个子列表，并返回其长度（仅在所有行长度相等时有效）
        return len(self._data[0])
```

> QtCore.QAbstractTableModel 是一个抽象基类，这意味着它没有实现这些方法。如果您尝试直接使用它，它将无法工作，所以您必须继承它。

在 `__init__` 构造函数中，我们接受一个参数 `data`，并将其存储为实例属性 `self._data`，这样我们就可以从方法中访问它。传入的 `data` 结构是通过引用存储的，因此任何外部更改都会反映在这里。

> 要通知模型发生的变化，您需要使用 `self.model.layoutChanged.emit()` 触发模型的 `layoutChanged` 信号。

`data` 方法带两个参数 `index` 和 `role`。`index` 参数指定当前请求信息的表中位置，并提供 `.row()` 和 `.column()` 两个方法，分别返回视图中的行号和列号。在示例中，数据以嵌套列表形式存储，行号和列号用于按以下方式索引：`data[row][column]`。

视图对源数据的结构一无所知，而模型负责在视图的行和列与您自己的数据存储中的相应位置之间进行转换。

角色参数描述了方法在本次调用中应返回何种信息。为了获取要显示的数据，视图会调用该模型的数据方法，并指定角色为 `Qt.ItemDataRole.DisplayRole`。然而，角色还可以有其他许多值，包括 `Qt.ItemDataRole.BackgroundRole`、`Qt.ItemDataRole.CheckStateRole`、`Qt.ItemDataRole.DecorationRole`、`Qt.ItemDataRole.CheckStateRole`、`Qt.ItemDataRole.DecorationRole`、`Qt.ItemDataRole.FontRole`、`Qt.ItemDataRole.TextAlignmentRole` 和 `Qt.ItemDataRole.ForegroundRole`，每个角色都期望返回特定值（详见后文）。

> `Qt.ItemDataRole.DisplayRole` 实际上期望返回一个字符串，尽管其他基本的 Python 类型，包括 `flaot`、`int` 和 `bool`，也将使用它们的默认字符串表示形式进行显示。然而，将这些类型格式化为自定义字符串通常更可取。

我们将稍后介绍如何使用这些其他角色类型，目前只需知道在返回用于显示的数据之前，您**必须**确保角色类型为 `Qt.ItemDataRole.DisplayRole`。

两个自定义方法 `columnCount` 和 `rowCount` 分别返回数据结构中的列数和行数。在我们这里使用的嵌套列表结构中，行数就等于外层列表中的元素个数，而列数等于内层列表中**某一个**列表的元素个数——假设所有内层列表的元素个数都相同。

> 如果这些方法返回的值过高，您会看到越界错误，如果返回的值过低，您将看到表格被截断。

数字和日期的格式设置

模型返回用于显示的数据预计为字符串。虽然整数（`int`）和浮点数（`float`）也会显示，但它们将使用默认的字符串表示形式，而复杂的 Python 类型则不会。要显示这些类型，或覆盖浮点数、整数或布尔值的默认格式化，您必须自行将这些类型格式化为字符串。

您可能会想通过将数据提前转换为字符串表来实现这一点。然而，这样做会使您难以继续处理表中的数据，无论是进行计算还是更新。

相反，您应该使用模型的数据方法按需进行字符串转换。通过这种方式，您可以继续使用原始数据，同时对数据的呈现方式拥有完全控制权——包括通过配置在运行时动态更改呈现方式。

以下是一个简单的自定义格式化器，它会从我们的数据表中查找值，并根据数据的 Python 类型以多种不同方式显示它们。

Listing 105. tableview_format_1.py

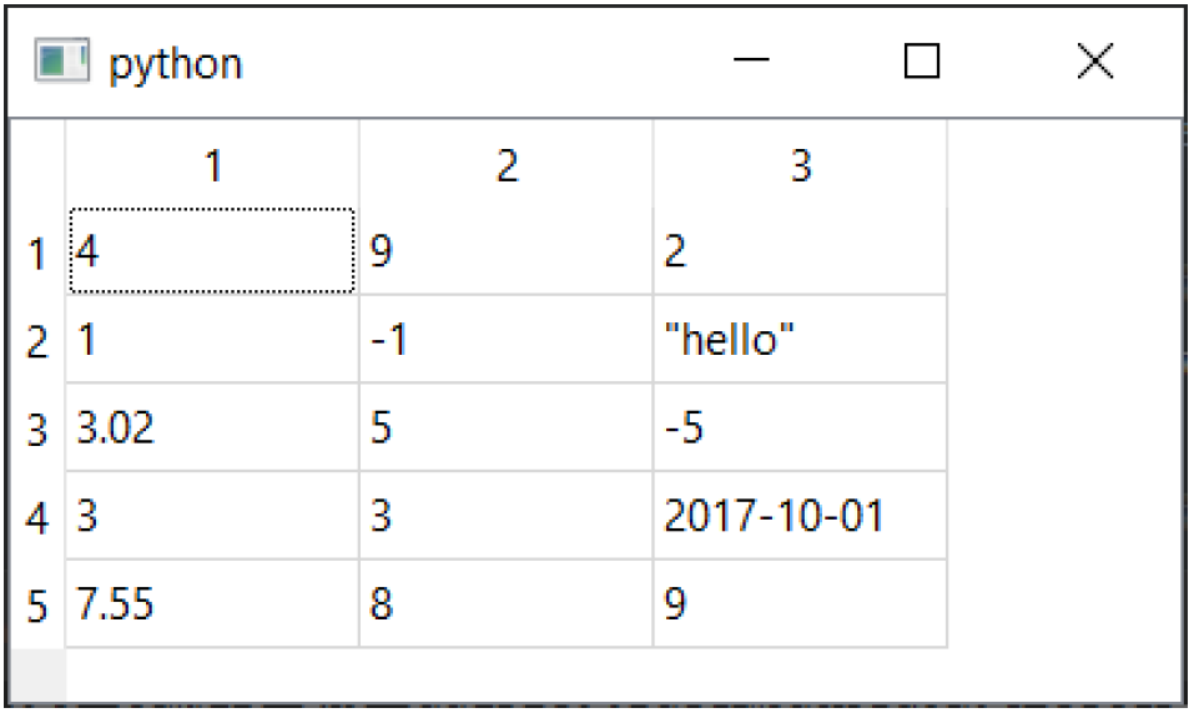
```
import sys
from datetime import datetime #1

from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt


class TableModel(QtCore.QAbstractTableModel):
    def __init__(self, data):
        super().__init__()
        self._data = data

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            # 获取原始值
            value = self._data[index.row()][index.column()]

            # 根据类型进行检查并相应渲染。
```

```
            if isinstance(value, datetime):
                # 渲染时间格式为 YYY-MM-DD。
                return value.strftime("%Y-%m-%d")

            if isinstance(value, float):
                # 将float类型保留小数点后两位
                return "%.2f" % value

            if isinstance(value, str):
                # 使用引号渲染字符串
                return '"%s"' % value

            # 默认值（未在上述内容中捕获的任何内容：例如int）
            return value

    def rowCount(self, index):
        return len(self._data)

    def columnCount(self, index):
        return len(self._data[0])
```

> 请注意文件开头的额外导入语句 `from datetime import datetime`。

请与下方的修改后示例数据配合使用，以查看其实际效果。

```
data = [
    [4, 9, 2],
    [1, -1, 'hello'],
    [3.023, 5, -5],
    [3, 3, datetime(2017,10,1)],
    [7.555, 8, 9],
]
```



> 图139：自定义数据格式化

到目前为止，我们只探讨了如何自定义数据本身的格式。然而，模型接口为用户提供了对表格单元格显示方式的更多控制权，包括颜色和图标。在接下来的部分，我们将探讨如何利用模型来自定义 `QTableView` 的外观。

具有角色的样式和颜色

使用颜色和图标来突出显示数据表中的单元格可以帮助用户更轻松地查找和理解数据，或帮助用户选择或标记感兴趣的数据。Qt 允许从模型中完全控制所有这些功能，通过响应数据方法中的相关角色来实现。

针对各种角色类型，预期返回的类型如下所示：

角色	类型
`Qt.ItemDataRole.BackgroundRole`	`QBrush`（也可以是 `QColor`）
`Qt.ItemDataRole.CheckStateRole`	`Qt.CheckState`
`Qt.ItemDataRole.DecorationRole`	`QIcon`，`QPixmap`，`QColor`
`Qt.ItemDataRole.DisplayRole`	`QString`（也可以是 `int`，`float`，`bool`）
`Qt.ItemDataRole.FontRole`	`QFont`
`Qt.ItemDataRole.SizeHintRole`	`QSize`
`Qt.ItemDataRole.TextAlignmentRole`	`Qt.Alignment`
`Qt.ItemDataRole.ForegroundRole`	`QBrush`（也可以是 `QColor`）

通过响应特定的角色和索引组合，我们可以修改表格中特定单元格、列或行的外观——例如，为第三列中的所有单元格设置蓝色背景。

Listing 106. tableview_format_2.py

```
    def data(self, index, role):
        if(
            role == Qt.ItemDataRole.BackgroundRole
            and index.column() == 2
        ):
            # 请参见下文的数据结构。
            return QtGui.QColor(Qt.GlobalColor.blue)

        # 现有的 `if role == Qt.ItemDataRole.DisplayRole:` 代码块已隐藏，以提高可读性。
```

通过使用索引从我们自己的数据中查找值，我们还可以根据数据中的值自定义外观。下面我们将介绍一些更常见的使用场景。

文本对齐

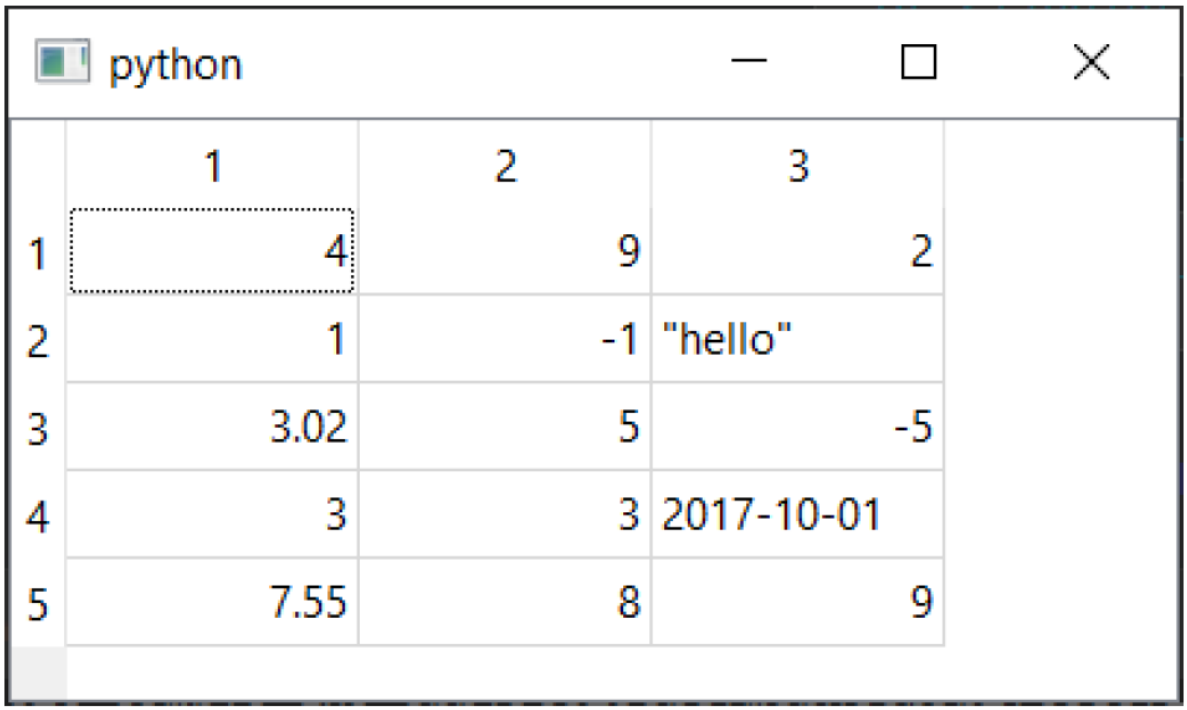
在之前的格式化示例中，我们使用文本格式化将浮点数显示为小数点后2位。然而，在显示数字时，右对齐数字也是一种常见做法，这样可以更方便地比较不同数字列表中的数值。这可以通过在响应 `Qt.ItemDataRole.TextAlignmentRole` 时返回 `Qt.Alignment.AlignRight` 来实现，适用于任何数值类型。

修改后的数据方法如下所示。我们检查 `role == Qt.ItemDataRole.TextAlignmentRole`，并像之前一样通过索引查找值，然后确定值是否为数值。如果是，我们可以返回 `Qt.Alignment.AlignVCenter + Qt.Alignment.AlignRight`，以实现垂直居中对齐和水平右对齐。

Listing 107. tableview_format_3.py

```
    def data(self, index, role):
        if role == Qt.ItemDataRole.TextAlignmentRole:
            value = self._data[index.row()][index.column()]

            if isinstance(value, int) or isinstance(value, float):
                # 右对齐，垂直居中
                return(
                    Qt.AlignmentFlag.AlignVCenter
                    | Qt.AlignmentFlag.AlignRight
                )

        # 现有的 `if role == Qt.ItemDataRole.DisplayRole:` 代码块已隐藏，以提高可读性。
```

> 其他对齐方式也是可能的，包括 `Qt.Alignment.AlignHCenter` 的实现水平居中对齐。您可以通过按位或运算将它们组合在一起，例如 `Qt.Alignment.AlignBottom | Qt.Alignment.AlignRight`。



> 图140：QTableView 的单元格对齐

文本颜色

如果您使用过Excel等电子表格软件，您可能对“条件格式化”这一概念有所了解。条件格式化是指您可以应用于单元格（或行、列）的规则，这些规则会根据单元格的值自动更改其文本和背景颜色。

这有助于可视化数据，例如使用红色表示负数，或通过从蓝到红的渐变色来突出显示数值范围（例如低至高）。

首先，下面的示例实现了一个处理程序，用于检查索引单元格中的值是否为数值且小于零。如果满足条件，则该处理程序返回文本（前景）颜色为红色。

Listing 108. tableview_format_4.py

```
    def data(self, index, role):
        if role == Qt.ItemDataRole.ForegroundRole:
            value = self._data[index.row()][index.column()]

            if(
                isinstance(value, int) or isinstance(value, float)
            ) and value < 0:
                return QtGui.QColor("red")


        # 现有的 `if role == Qt.ItemDataRole.DisplayRole:` 代码块已隐藏，以提高可读性。
```

如果你将此添加到模型的数据处理程序中，所有负数现在将以红色显示。

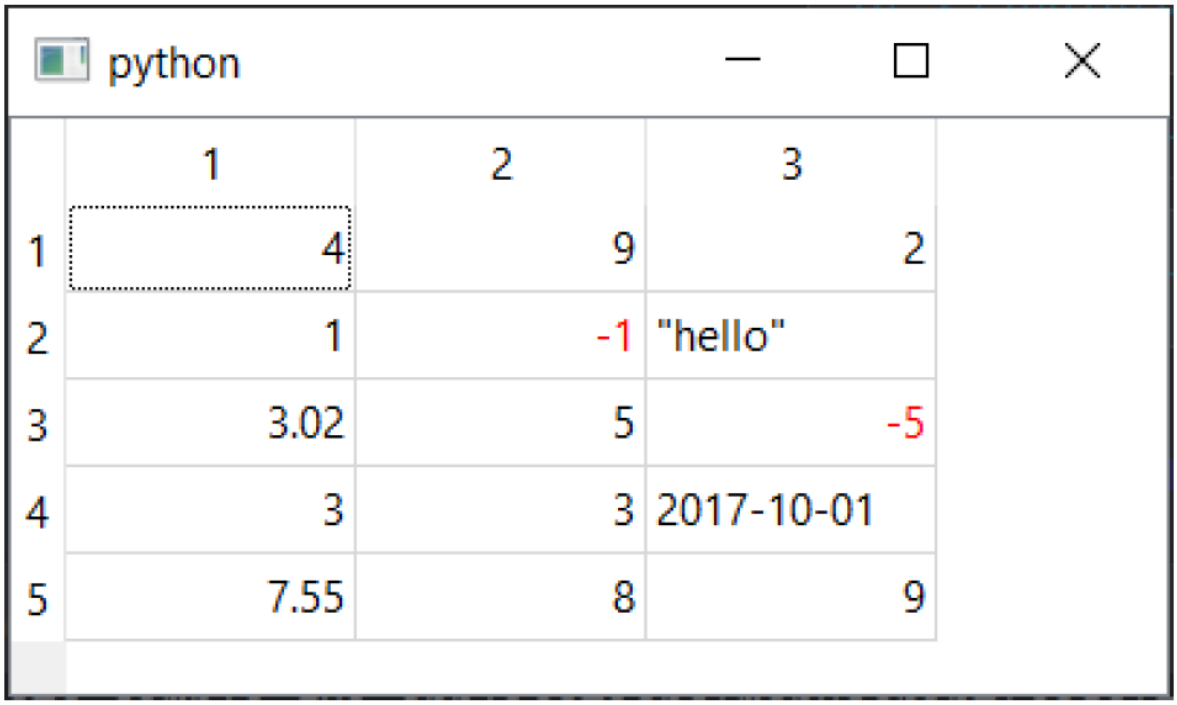


图141：QTableView 的文本格式设置，负数使用红色显示

数值范围渐变

同样的原理可以用于对表格中的数值应用渐变色，例如，突出显示较低和较高的值。首先，我们定义颜色标尺，该标尺来自 colorbrewer2.org。

```
COLORS = ['#053061', '#2166ac', '#4393c3', '#92c5de', '#d1e5f0',
'#f7f7f7', '#fddbc7', '#f4a582', '#d6604d', '#b2182b', '#67001f']
```

接下来，我们定义自定义处理函数，这次针对 `Qt.ItemDataRole.BackgroundRole`。该函数获取指定索引处的值，验证其为数值类型，然后执行一系列操作将其限制在 0...10 的范围内，以便用于索引我们的列表。

Listing 109. tableview_format_5.py

```
    def data(self, index, role):
        if role == Qt.ItemDataRole.BackgroundRole:
            value = self._data[index.row()][index.column()]
            if isinstance(value, int) or isinstance(value, float):
                value = int(value) # 将值转换为整数以进行索引操作。

                # 将范围限制为-5到+5，然后转换为0到10。
                value = max(-5, value) # 值小于-5的变为-5
                value = min(5, value) # 值大于+5时，变为+5
                value = value + 5 # -5变为0，+5变为+10

                return QtGui.QColor(COLORS[value])

        # 现有的 `if role == Qt.ItemDataRole.DisplayRole:` 代码块已隐藏，以提高可读性。
```

这里用于将值转换为梯度的逻辑非常基础，会截取高/低值，且不调整数据范围。然而，您可以根据需要进行调整，只要您的处理函数最终返回的是 `QColor` 或 `QBrush` 即可。

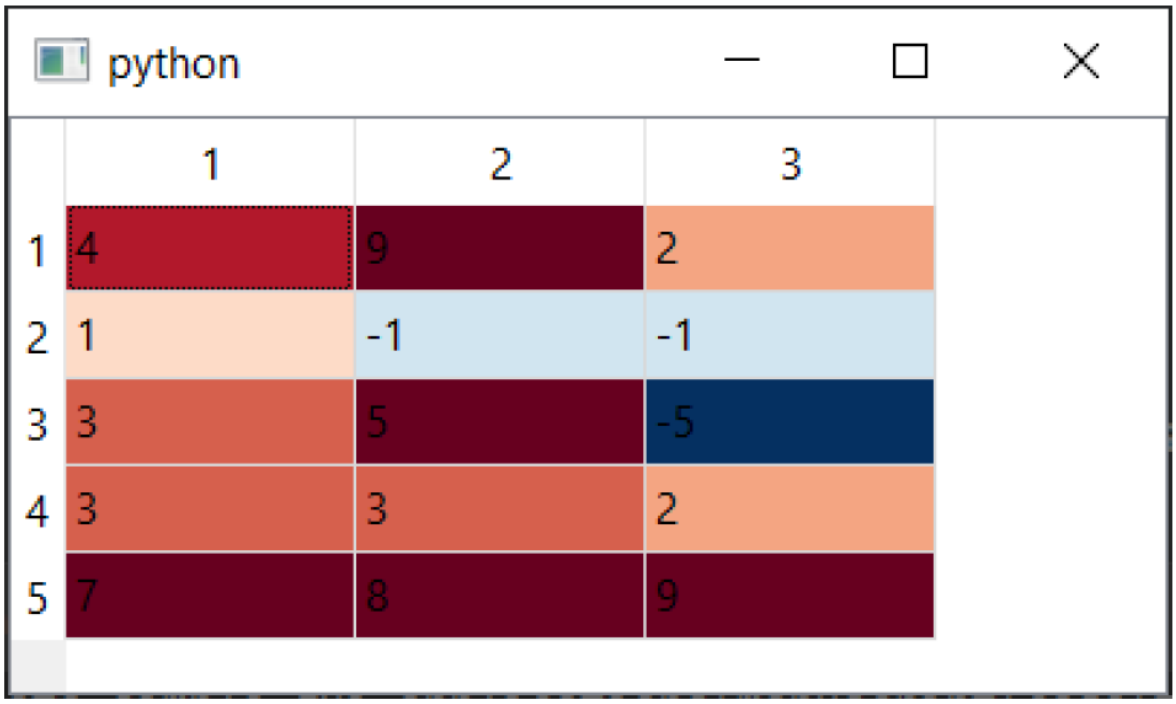


图142：带数字范围颜色渐变的QTableView

图标与图像装饰

每个表格单元格包含一个小型装饰区域，可用于在数据左侧显示图标、图像或纯色块。此区域可用于指示数据类型，例如日历图标表示日期、勾号和叉号表示布尔值，或用于对数值范围进行更微妙的条件格式化。

以下是一些这些想法的简单实现。

使用图标表示布尔/日期数据类型

对于日期，我们将使用 Python 的内置 `datetime` 类型。首先，在文件开头添加以下 import 语句以导入该类型。

```
from datetime import datetime
```

然后，更新数据（在 `MainWindow.__init__` 中设置）以添加日期时间和布尔值（`True` 或 `False` 值），例如。

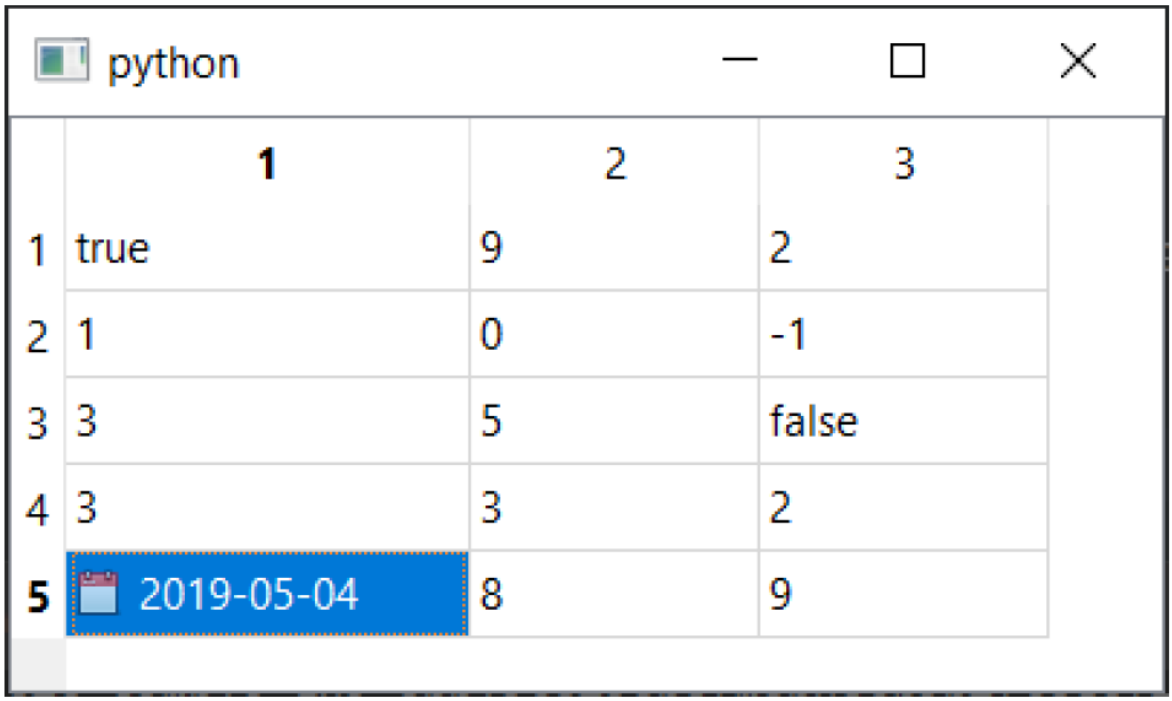
```
data = [
    [True, 9, 2],
    [1, 0, -1],
    [3, 5, False],
    [3, 3, 2],
    [datetime(2019, 5, 4), 8, 9],
]
```

这些设置完成后，您可以更新模型数据方法，以显示图标和格式化日期（适用于日期类型），使用以下代码：

Listing 110. tableview_format_6.py

```
import os

basedir = os.path.dirname(__file__)

class TableModel(QtCore.QAbstractTableModel):
    def __init__(self, data):
        super().__init__()
        self._data = data

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            value = self._data[index.row()][index.column()]
            if isinstance(value, datetime):
                return value.strftime("%Y-%m-%d")

            return value

        if role == Qt.ItemDataRole.DecorationRole:
            value = self._data[index.row()][index.column()]
            if isinstance(value, datetime):
                return QtGui.QIcon(
                    os.path.join(basedir, "calendar.png")
                )

    def rowCount(self, index):
        return len(self._data)

    def columnCount(self, index):
        return len(self._data[0])
```

> 我们使用之前介绍的 `basedir` 技术加载图标，以确保无论脚本如何运行，路径都是正确的。



> 图143：QTableView 的格式化日期并显示指示图标

以下展示了如何使用勾选框和叉号分别表示布尔值的真（`True`）和假（`False`）

Listing 111. tableview_format_7.py

```
    def data(self, index, role):
        if role == Qt.ItemDataRole.DecorationRole:
            value = self._data[index.row()][index.column()]
            if isinstance(value, bool):
                if value:
                    return QtGui.QIcon("tick.png")

                return QtGui.QIcon("cross.png")
```

当然，您可以将以上内容组合在一起，或者将 `Qt.ItemDataRole.DecorationRole` 和 `Qt.ItemDataRole.DisplayRole` 滑块进行任意组合。通常情况下，将每种类型归入同一个 if 分支会比较简单，或者随着模型的复杂程度增加，可以创建子方法来处理每种角色。

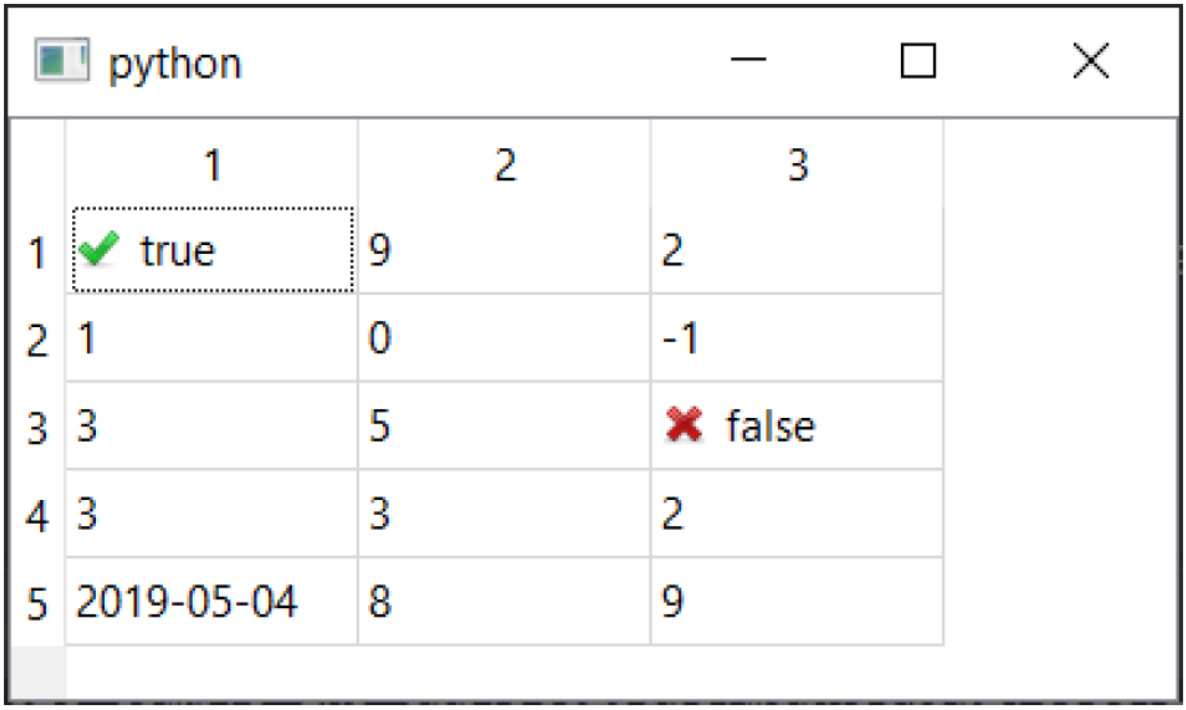


图144：QTableView 的布尔指示器

色块

如果您为 `Qt.ItemDataRole.DecorationRole` 返回 `QColor`，则单元格左侧的图标位置将显示一个小方块。这与之前的 `Qt.ItemDataRole.BackgroundRole` 条件格式化示例完全相同，只是现在需要处理和响应 `Qt.ItemDataRole.DecorationRole`。

Listing 112. tableview_format_8.py

```
    def data(self, index, role):
        if role == Qt.ItemDataRole.DecorationRole:
            value = self._data[index.row()][index.column()]

            if isinstance(value, datetime):
                return QtGui.QIcon(
                    os.path.join(basedir, "calendar.png")
                )

            if isinstance(value, bool):
                if value:
                    return QtGui.QIcon(
                        os.path.join(basedir, "tick.png")
                    )

                return QtGui.QIcon(os.path.join(basedir, "cross.png"))

            if isinstance(value, int) or isinstance(value, float):
                value = int(value)

                # 将范围限制为-5到+5，然后转换为0到10。
                value = max(-5, value) # 值小于-5的变为-5
                value = min(5, value) # 值大于+5时，变为+5
                value = value + 5 # -5变为0，+5变为+10
```

```
            return QtGui.QColor(COLORS[value])
```

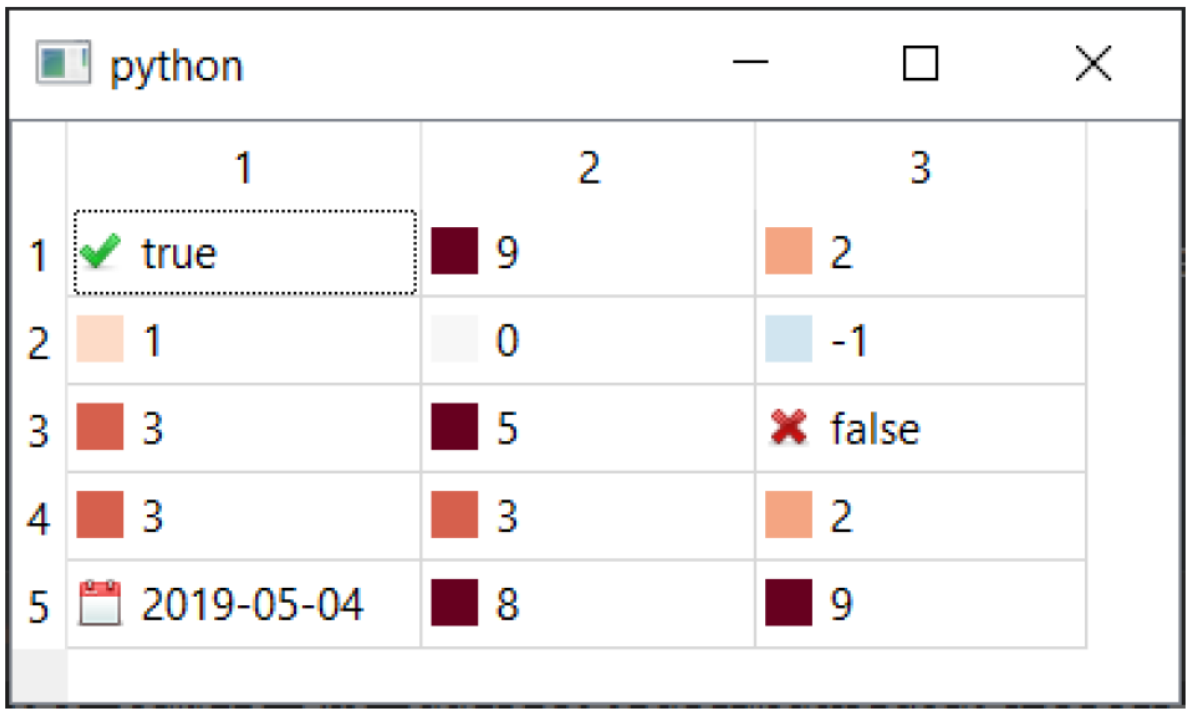


图145：QTableView 颜色块装饰

备选的 Python 数据结构

到目前为止，我们在示例中一直使用简单的嵌套 Python 列表来存储数据以供显示。对于简单的数据表，这种方法是可行的。然而，如果您正在处理大型数据表，Python 还提供了一些其他更好的选项，这些选项还附带额外的好处。在接下来的部分中，我们将探讨两个 Python 数据表库——numpy 和 pandas——以及如何将它们与 Qt 集成。

Numpy

Numpy 是一个库，为 Python 中的大型多维数组或矩阵数据结构提供支持。它对大型数组的高效、高性能处理，使 numpy 成为科学和数学应用的理想选择。这也使 numpy 数组成为 PyQt6 中大型、单类型数据表的良好数据存储。

使用numpy作为数据源

为了支持 numpy 数组，我们需要对模型进行一系列修改，首先修改数据方法中的索引逻辑，然后调整行数和列数的计算方式，以适应 `rowCount` 和 `columnCount` 的计算。

标准的numpy API提供了对2D数组的元素级访问，通过在同一个切片操作中传入行和列，例如 `_data[index.row(), index.column()]`。这比分两步进行索引操作更高效，如列表嵌套列表的示例所示。

在 NumPy 中，数组的维度可以通过 `.shape` 属性获取，该属性会返回一个元组，其中包含沿每个轴的维度。我们可以通过从该元组中选择正确的元素来获取每个轴的长度，例如 `_data.shape[0]` 即可获取第一个轴的大小。

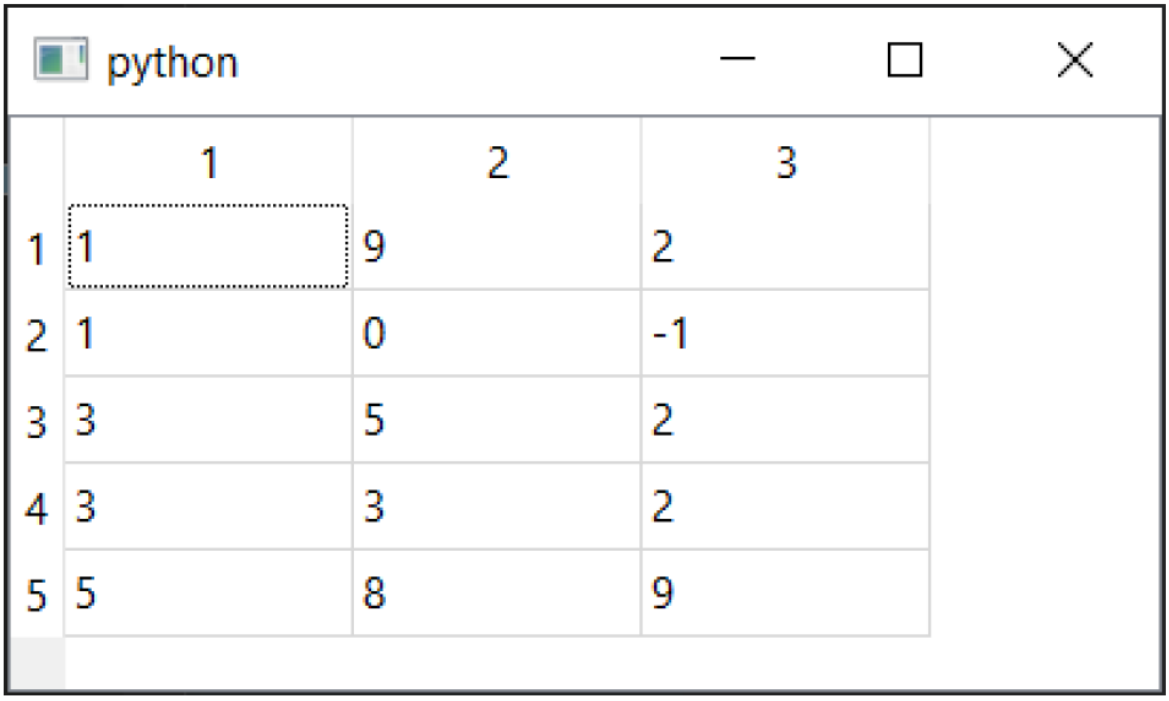
以下完整示例演示了如何使用Qt的 `QTableView` 通过自定义模型显示一个 numpy 数组。

Listing 113. model-views/tableview_numpy.py

```
import sys
```

```python
import numpy as np
from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt


class TableModel(QtCore.QAbstractTableModel):
    def __init__(self, data):
        super().__init__()
        self._data = data

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            # self._data[index.row()][index.column()] 也行
            value = self._data[index.row(), index.column()]
            return str(value)

    def rowCount(self, index):
        return self._data.shape[0]

    def columnCount(self, index):
        return self._data.shape[1]


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.table = QtWidgets.QTableView()
        data = np.array(
            [
                [1, 9, 2],
                [1, 0, -1],
                [3, 5, 2],
                [3, 3, 2],
                [5, 8, 9],
            ]
        )

        self.model = TableModel(data)
        self.table.setModel(self.model)

        self.setCentralWidget(self.table)
        self.setGeometry(600, 100, 400, 200)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 虽然简单的 Python 类型（如 `int` 和 `float`）会直接显示，无需转换为字符串，但 numpy 会使用自己的类型（例如 `numpy.int32`）来表示数组值。为了显示这些值，我们**必须**先将其转换为字符串。



> 图146：使用 numpy 数组的QTableView

> 使用 `QTableView` 只能显示二维数组，然而如果您拥有更高维度的数据结构，您可以将 `QTableView` 与带标签或滚动条的用户界面结合使用，以实现对这些更高维度的访问和显示。

Pandas

Pandas 是一个常用于数据处理和分析的 Python 库。它提供了一个便捷的 API，用于从各种数据源加载二维表格数据并对其进行数据分析。通过将 numpy 的 `DataTable` 用作您的 `QTableView` 模型，您可以直接在应用程序中使用这些 API 加载和分析数据。

使用Pandas作为数据源

将模型修改为与 numpy 兼容的改动相对较小，仅需对 `data` 方法中的索引进行调整，以及对 `rowCount` 和 `columnCount` 进行修改。`rowCount` 和 `columnCount` 的改动与 numpy 完全一致，使用 pandas 中的 `_data.shape` 元组来表示数据的维度。

对于索引操作，我们使用 pandas 的 `.iloc` 方法，用于索引位置——即通过列和/或行索引进行查找。这通过将行和列传递给 `_data.iloc[index.row(), index.column()]` 切片来实现。

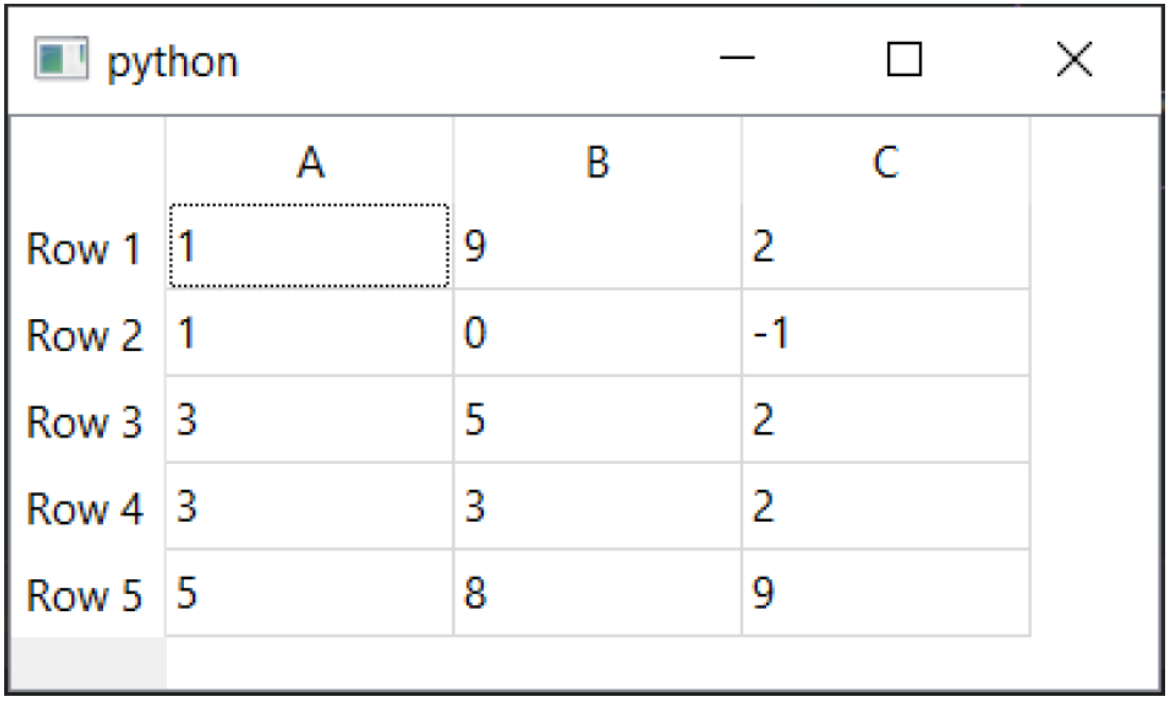
以下完整示例演示了如何通过自定义模型使用Qt 的 `QTableView` 显示 pandas 数据框。

Listing 114. model-views/tableview_pandas.py

```
import sys

import pandas as pd
from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt


class TableModel(QtCore.QAbstractTableModel):
    def __init__(self, data):
        super().__init__()
        self._data = data

    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            value = self._data.iloc[index.row(), index.column()]
            return str(value)

    def rowCount(self, index):
        return self._data.shape[0]

    def columnCount(self, index):
        return self._data.shape[1]

    def headerData(self, section, orientation, role):
        if role == Qt.ItemDataRole.DisplayRole:
            if orientation == Qt.Orientation.Horizontal:
                return str(self._data.columns[section])

            if orientation == Qt.Orientation.Vertical:
                return str(self._data.index[section])


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.table = QtWidgets.QTableView()

        data = pd.DataFrame(
            [
                [1, 9, 2],
                [1, 0, -1],
                [3, 5, 2],
                [3, 3, 2],
                [5, 8, 9],
            ],
            columns=["A", "B", "C"],
            index=["Row 1", "Row 2", "Row 3", "Row 4", "Row 5"],
        )
```

```
        self.model = TableModel(data)
        self.table.setModel(self.model)

        self.setCentralWidget(self.table)
        self.setGeometry(600, 100, 400, 200)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

一个有趣的扩展是使用 `QTableView` 的表格标题来显示行和 pandas 列标题的值，这些值可以分别从 `DataFrame.index` 和 `DataFrame.columns` 中获取。



> 图147：QTableView 支持 pandas DataTable，包含列和行标题。

为此，我们需要在自定义的 `headerData` 方法中实现 `Qt.ItemDataRole.DisplayRole` 处理程序。它接收 `section`，它包含行/列的索引（0...n）、`orientation`（可为 `Qt.Orientations.Horizontal` 用于列标题，或 `Qt.Orientations.Vertical` 用于行标题），以及与数据方法中相同的 `role` 参数。

> `headerData` 方法还接受其他参数，这些参数可用于进一步自定义标题的样式。

总结

在本章中，我们介绍了如何使用 QTableView 和自定义模型在应用程序中显示表格数据的基础知识。随后，我们进一步演示了如何格式化数据以及使用图标和颜色装饰单元格。最后，我们演示了如何使用 ·QTableView· 处理来自 numpy 和 pandas 结构的表格数据，包括显示自定义的列和行标题。

> 如果您想对表格数据进行计算，请查看 25. 使用线程池。

20. 使用Qt模型查询SQL数据库

到目前为止，我们一直使用表格模型来访问应用程序自身加载或存储的数据——从简单的列表列表到 numpy 和 pandas 表格。然而，所有这些方法都有一个共同点，即您所查看的数据必须完全加载到内存中。

为了简化与 SQL 数据库的交互，Qt 提供了一些 SQL 模型，这些模型可以连接到视图，以显示 SQL 查询或数据库表的输出。在本章中，我们将介绍两种方法——在 `QTableView` 中显示数据库数据，以及使用 `QDataWidgetMapper` 将数据库字段映射到 Qt 控件。

您选择哪种模型取决于您是否需要对数据库进行只读访问、读写访问，还是带有关联关系的只读访问（查询多个表）。在接下来的章节中，我们将依次探讨这些选项。

以下示例基于此简单框架，展示了在窗口中显示表格视图，但未设置模型。

Listing 115. databases/tableview.py

```
import os
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QTableView


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.table = QTableView()

        # self.model = ?
        # self.table.setModel(self.model)

        self.setCentralWidget(self.table)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

在连接模型之前，运行此操作只会显示一个空窗口。

> 对于这些示例，我们使用了本书下载中包含的 SQLite 文件数据库示例 `file.sqlite`。

> 您可以使用自己的数据库，包括SQLite数据库或数据库服务器（如PostgreSQL、MySQL等）。有关如何连接到远程服务器的详细说明，请参阅后文的使用 `QSqlDatabase` 进行身份验证。

连接到数据库

要在应用程序中显示数据库中的数据，您必须首先与数据库建立连接。Qt 支持服务器型数据库（如 PostgreSQL 或 MySQL）和文件型数据库（如 SQLite），两者的区别仅在于配置方式不同。

对于所有这些示例，我们使用的是 Chinook示例数据库——一个专为测试和演示设计的示例数据库。该数据库模拟了一家数字媒体商店，包含艺术家、专辑、媒体曲目、发票和客户等表。

> 本书附带了该数据库的SQLite版本副本，命名为 `chinook.sqlite`。您也可以从这里下载最新版本。

```
import os

from PyQt6.QtSql import QSqlDatabase

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()
```

> 您将此代码放置的位置取决于您的应用程序。通常，您希望创建一个数据库连接并在整个应用程序中使用它——在这种情况下，最好创建一个单独的模块，例如 `db.py` 来存放此代码（以及其他相关功能）。

对于所有数据库，操作流程相同——创建数据库对象，设置名称，然后打开数据库以初始化连接。然而，如果您想连接到远程数据库，则需要额外设置一些参数。请参阅后文的使用 `QSqlDatabase` 进行身份验证以获取更多信息。

使用 `QSqlTableModel` 显示表格

将应用程序连接到数据库后，您可以做的最简单的事情就是在应用程序中显示一张表。为此，我们可以使用 `QSqlTableModel`。该模型直接从表中显示数据，并支持编辑功能。

首先，我们需要创建表模型的实例，并传入我们上面创建的数据库对象。然后，我们需要设置要查询数据的源表——这是数据库中表的名称，这里是<表名>。最后，我们需要调用模型的 `.select()` 方法。

```
model = QSqlTableModel(db=db)
model.setTable('<table name>')
model.select()
```

通过调用 `.select()` 方法，我们指示模型查询数据库并保留查询结果，以便后续显示。要将这些数据显示在 `QTableView` 中，只需将其传递给视图的 `.setModel()` 方法即可。

```
table = QTableView()
table.setModel(self.model)
```

数据将以表格形式显示，并可通过滚动条进行浏览。请参见下文的完整代码，该代码加载数据库并在视图中显示专辑表。

Listing 116. tableview_tablemodel.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtSql import QSqlDatabase, QSqlTableModel
from PyQt6.QtWidgets import QApplication, QMainWindow, QTableView

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()


class MainWindow(QMainWindow):
```

```
    def __init__(self):
        super().__init__()

        self.table = QTableView()

        self.model = QSqlTableModel(db=db)

        self.table.setModel(self.model)

        self.model.setTable("Track")
        self.model.select()

        self.setMinimumSize(QSize(1024, 600))
        self.setCentralWidget(self.table)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

运行后，您将看到以下窗口。

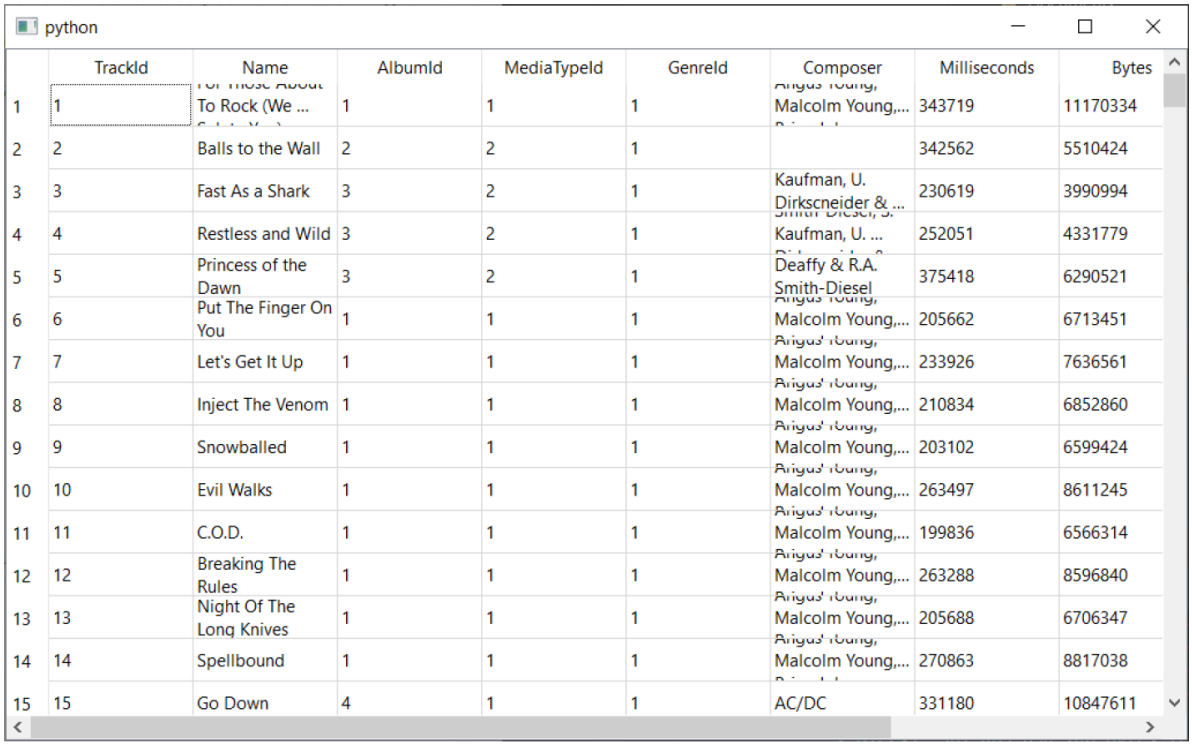


图148：在QTableView中显示的专辑表。

您可以通过拖动右侧边缘来调整列的宽度。通过双击右侧边缘，可以调整列的宽度以适应内容。

编辑数据

数据库中的数据在 `QTableView` 中默认可编辑——只需双击任何单元格，即可修改其内容。修改内容将在编辑完成后立即保存回数据库。

Qt 提供了一些对编辑行为的控制，您可以根据所构建的应用程序类型进行调整。Qt 将这些行为称为"编辑策略"，它们可以是以下之一：

编辑策略	描述
`QSqlTableModel.EditStrategy.OnFieldChange`	当用户取消选中已编辑的单元格时，更改会自动应用。
`QSqlTableModel.EditStrategy.OnRowChange`	当用户选择不同的行时，更改会自动应用。
`QSqlTableModel.EditStrategy.OnManualSubmit`	更改会被缓存在模型中，并仅在调用 `.submitAll()` 时写入数据库，或在调用 `revertAll()` 时被丢弃。

您可以通过调用 `.setEditStrategy` 方法来设置模型的当前编辑策略。例如

```
self.model.setEditStrategy(QSqlTableModel.EditStrategy.OnRowChange)
```

列排序

要按指定列对表格进行排序，我们可以调用模型上的 `.setSort()` 方法，传入列索引和 `Qt.SortOrder.AscendingOrder` 或 `Qt.SortOrder.DescendingOrder`。

Listing 117. databases/tableview_tablemodel_sort.py

```
        self.model.setTable("Track")
        self.model.setSort(2, Qt.SortOrder.DescendingOrder)
        self.model.select()
```

这必须在调用 `.select()` 之前完成。如果您希望在获取数据后进行排序，您可以再次调用 `.select()` 来刷新数据。

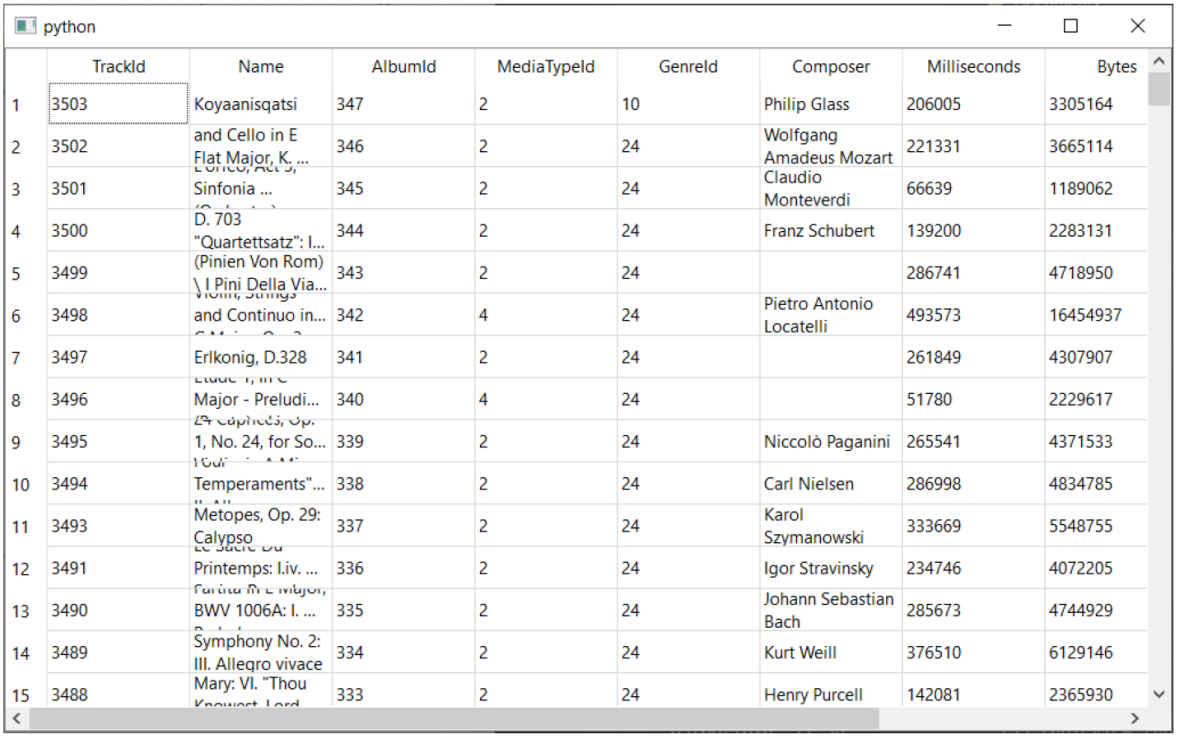


图149：根据列索引2（`album_id`）对专辑表进行排序。

您可能更倾向于使用列名而非列索引对表格进行排序。要实现这一点，您可以通过列名查询列索引。

Listing 118. databases/tableview_tablemodel_sortname.py

```
        self.model.setTable("Track")
        idx = self.model.fieldIndex("Milliseconds")
        self.model.setSort(idx, Qt.SortOrder.DescendingOrder)
        self.model.select()
```

表格现已按毫秒列（`milliseconds`）进行排序。

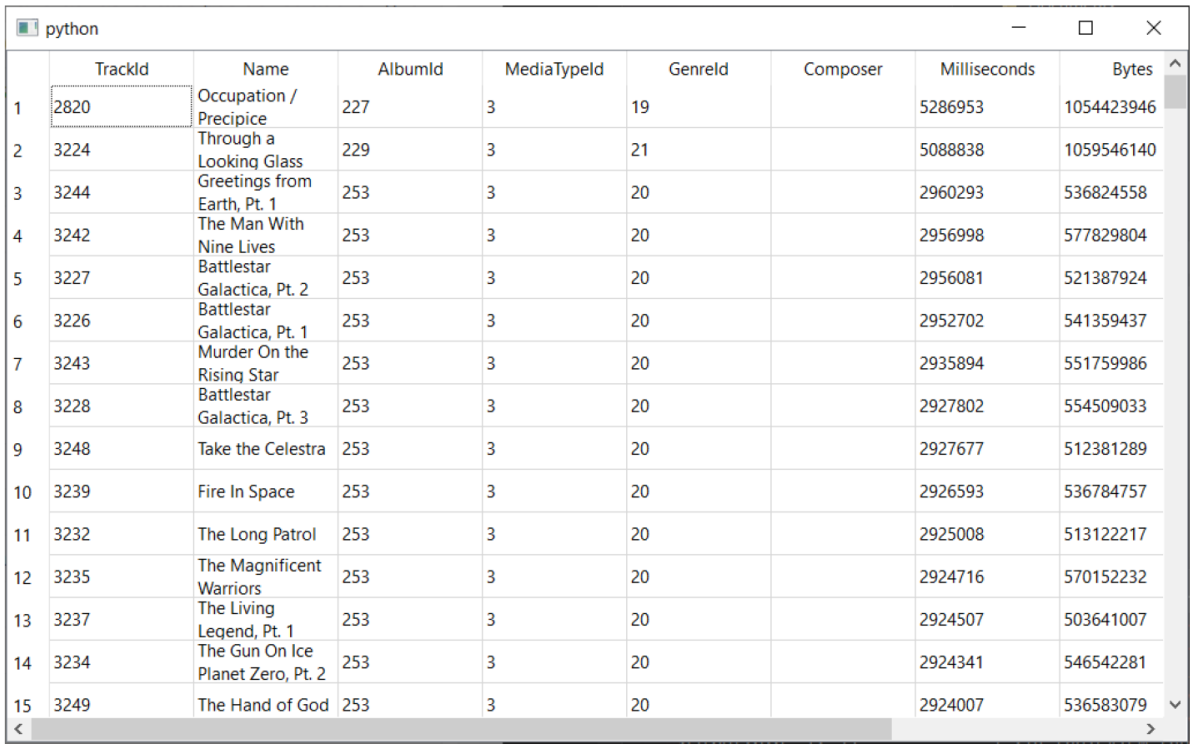


图150：按毫秒列排序的专辑表。

列标题

默认情况下，表格中的列标题来自数据库中的列名。通常这并不太利于用户使用，因此您可以使用 `.setHeaderData` 方法替换它们，传入列索引、方向（水平（顶部）或垂直（左侧）标题）以及标签。

Listing 119. database/tableview_tablemodel_titles.py

```
        self.model.setTable("Track")
        self.model.setHeaderData(1, Qt.Orientation.Horizontal, "Name")
        self.model.setHeaderData(
            2, Qt.Orientation.Horizontal, "Album (ID)"
        )
        self.model.setHeaderData(
            3, Qt.Orientation.Horizontal, "Media Type (ID)"
        )
        self.model.setHeaderData(
            4, Qt.Orientation.Horizontal, "Genre (ID)"
        )
        self.model.setHeaderData(
            5, Qt.Orientation.Horizontal, "Composer"
        )
        self.model.select()
```

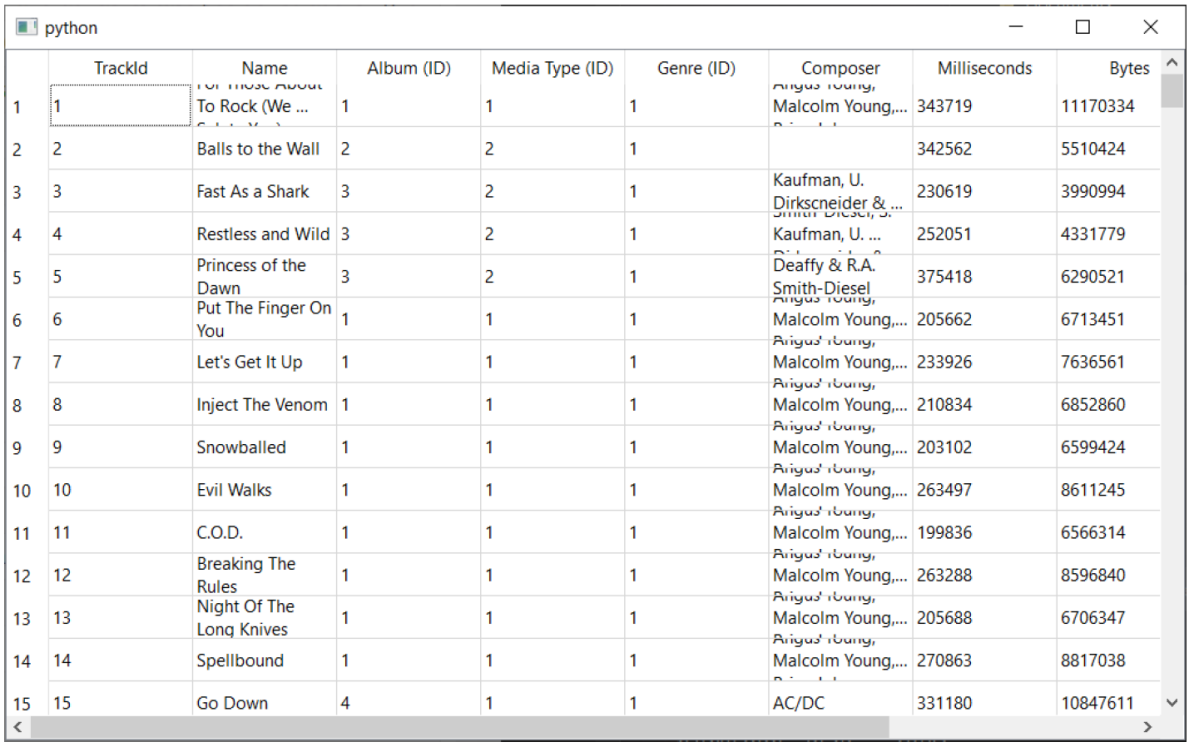


图151：带有更美观的列标题的专辑表。

与排序时类似，使用列索引来实现这一点并不总是方便的。如果数据库中的列顺序发生变化，您在应用程序中设置的名称将与之不一致。

与之前一样，我们可以使用 `.fieldIndex()` 方法来查找给定名称的索引。您可以更进一步，定义一个 Python 字典，其中包含列名和标题，以便在设置模型时一次性应用。

Listing 120. database/tableview_tablemodel_titlesname.py

```
        self.model.setTable("Track")
        column_titles = {
            "Name": "Name",
            "AlbumId": "Album (ID)",
            "MediaTypeId": "Media Type (ID)",
            "GenreId": "Genre (ID)",
            "Composer": "Composer",
        }
        for n, t in column_titles.items():
            idx = self.model.fieldIndex(n)
            self.model.setHeaderData(idx, Qt.Orientation.Horizontal,t)

        self.model.select()
```

选择列

通常您可能不需要显示表格中的所有列。您可以通过从模型中移除列来选择要显示的列。要实现这一点，请调用 `.removeColumns()` 方法，传入要移除的第一列的索引以及后续要移除的列数。例如：

```
self.model.removeColumns(2, 5)
```

一旦删除列，它们将不再显示在表格中。您可以使用与列标签相同的名称查找方法，通过名称删除列。

```
columns_to_remove = ['name', 'something']

for cn in columns_to_remove:
    idx = self.model.fieldIndex(cn)
    self.model.removeColumns(idx, 1)
```

> 以这种方式删除列仅会将其从视图中移除。若需通过SQL过滤列，请参阅下方的查询模型。

筛选表格

我们可以调用模型上的 `.setFilter()` 方法来过滤表格，传入一个描述过滤条件的参数。过滤条件参数可以是任何有效的 SQL `WHERE` 子句，但不需要在前面添加 `WHERE`。例如，`name="Martin"` 用于精确匹配，或 `name LIKE "Ma%"` 用于匹配以 "Ma" 开头的字段。

如果您对 SQL 不熟悉，以下是一些示例搜索模式，您可以使用它们来执行不同类型的搜索。

搜索模式	描述
`field="{}"`	字段与字符串完全匹配。
`field LIKE "{}%"`	字段以给定的字符串开头。
`field LIKE "%{}"`	字段以给定的字符串结尾。

搜索模式	描述
`field LIKE "%{}%"`	字段包含在给定的字符串中。

在每个示例中，`{}` 表示搜索字符串，您必须使用 Python 进行插值：`“{}”.format(search_str)`。与排序不同，过滤操作将自动应用于数据，无需再次调用 `.select()` 方法。

> 如果 `.select()` 尚未被调用，则过滤器将在首次调用时应用。

在下面的示例中，我们添加了一个 `QLineEdit` 字段，并将其连接到搜索轨道名称字段的表。我们将行编辑更改信号连接到构建，并将其应用到模型。

Listing 121. databases/tableview_tablemodel_filter.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtSql import QSqlDatabase, QSqlTableModel
from PyQt6.QtWidgets import (
    QApplication,
    QLineEdit,
    QMainWindow,
    QTableView,
    QVBoxLayout,
    QWidget,
)

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        container = QWidget()
        layout = QVBoxLayout()

        self.search = QLineEdit()
        self.search.textChanged.connect(self.update_filter)
        self.table = QTableView()

        layout.addWidget(self.search)
        layout.addWidget(self.table)
```

```
        container.setLayout(layout)

        self.model = QSqlTableModel(db=db)

        self.table.setModel(self.model)

        self.model.setTable("Track")
        self.model.select()

        self.setMinimumSize(QSize(1024, 600))
        self.setCentralWidget(container)

    def update_filter(self, s):
        filter_str = 'Name LIKE "%{}%"'.format(s)
        self.model.setFilter(filter_str)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

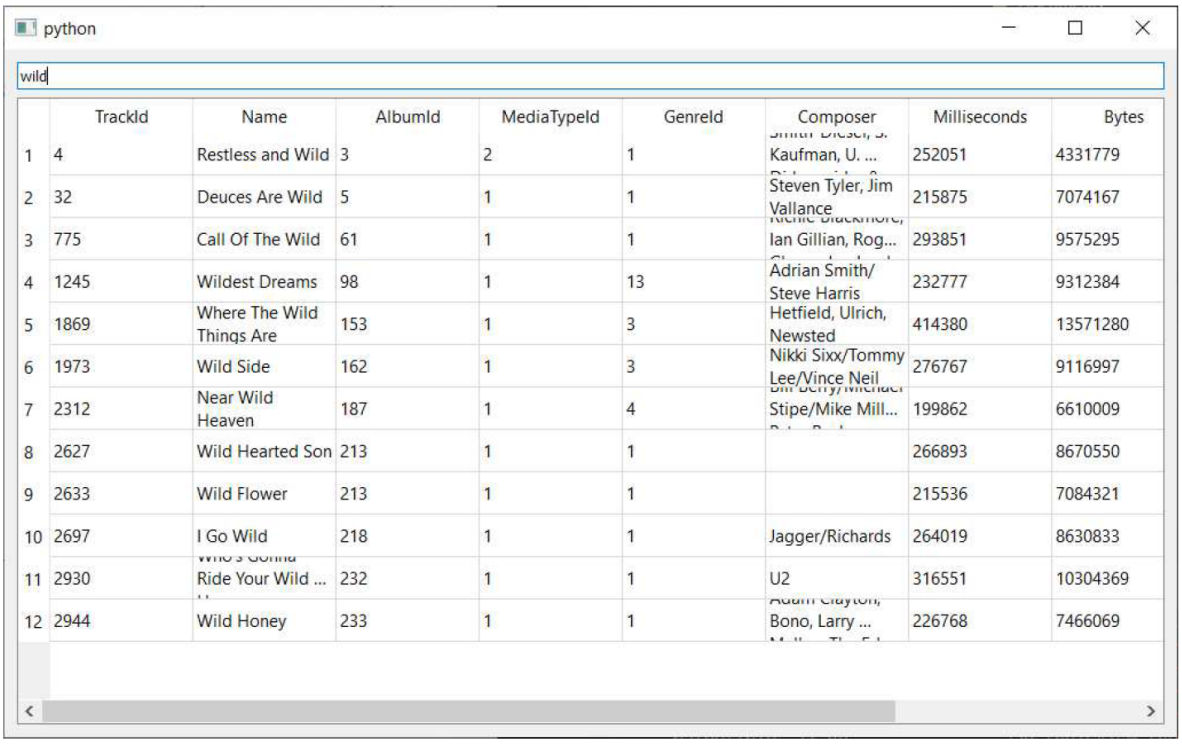


图152：按名称过滤专辑表

这容易受到SQL注入攻击。

虽然这种方法可行，但这其实是一种非常糟糕的在表中启用搜索功能的方式，因为用户可以构造无效或恶意的SQL语句。例如，尝试在搜索框中输入单个字符 `“` ——过滤功能将停止工作，并且在重新启动应用程序之前无法再次正常工作。

这是因为您创建了一个无效的 SQL 语句，例如：

```
'name LIKE "%"%"'
```

解决此问题的理想方法是使用参数化查询，将输入的转义工作交给数据库，以确保不会传递任何危险或格式错误的内容。然而，Qt过滤器接口不支持此功能，我们只能传递一个字符串。

对于简单的纯文本搜索，我们可以直接移除字符串中的任何非字母数字字符或空格。这是否合适将取决于您的具体使用场景。

```
import re

s = re.sub('[\W_]+', '', s)
query = 'field="%s"' % s
```

将上述内容应用到我们示例中的过滤方法中，我们得到以下代码：

Listing 122. databases/tableview_tablemodel_filter_clean.py

```
    def update_filter(self, s):
        s = re.sub("[\W_]+", "", s)
        filter_str = 'Name LIKE "%{}%"'.format(s)
        self.model.setFilter(filter_str)
```

请再次运行示例，并输入 `“` —— 以及您能想到的任何其他垃圾内容。您应该会发现搜索功能仍然正常工作。

使用 `QSqlRelationalTableModel` 显示相关数据

在之前的示例中，我们使用了 `QSqlTableModel` 来显示单个表中的数据。然而，在关系型数据库中，表之间可以存在关联关系，并且通常有必要能够直接查看相关数据。

关系数据库中的关系通过外键进行处理。外键是一个（通常为）数字值，存储在一个表的列中，引用另一个表中行中的主键。

在我们的示例跟踪表中，外键的示例可以是 `album_id` 或 `genre_id`。这两者都是数值，分别指向专辑表和类型表中的记录。将这些值（如 1、2、3 等）显示给用户是没有帮助的，因为它们本身没有意义。

更理想的做法是提取专辑名称或音乐类型，并在表格视图中显示。为此，我们可以使用 `QSqlRelationalTableModel`。

该模型的设置与前一个模型完全相同。要定义关系，我们调用 `.setRelation()` 方法，传入列索引和一个 `QSqlRelation` 对象。

```
from PyQt6.QtSql import QSqlRelation, QSqlRelationalTableModel

self.model = QSqlRelationalTableModel(db=db)

relation = QSqlRelation('<related_table>',
                        '<related_table_foreign_key_column',
'<column_to_display>')
self.model.setRelation(<column>, relation)
```

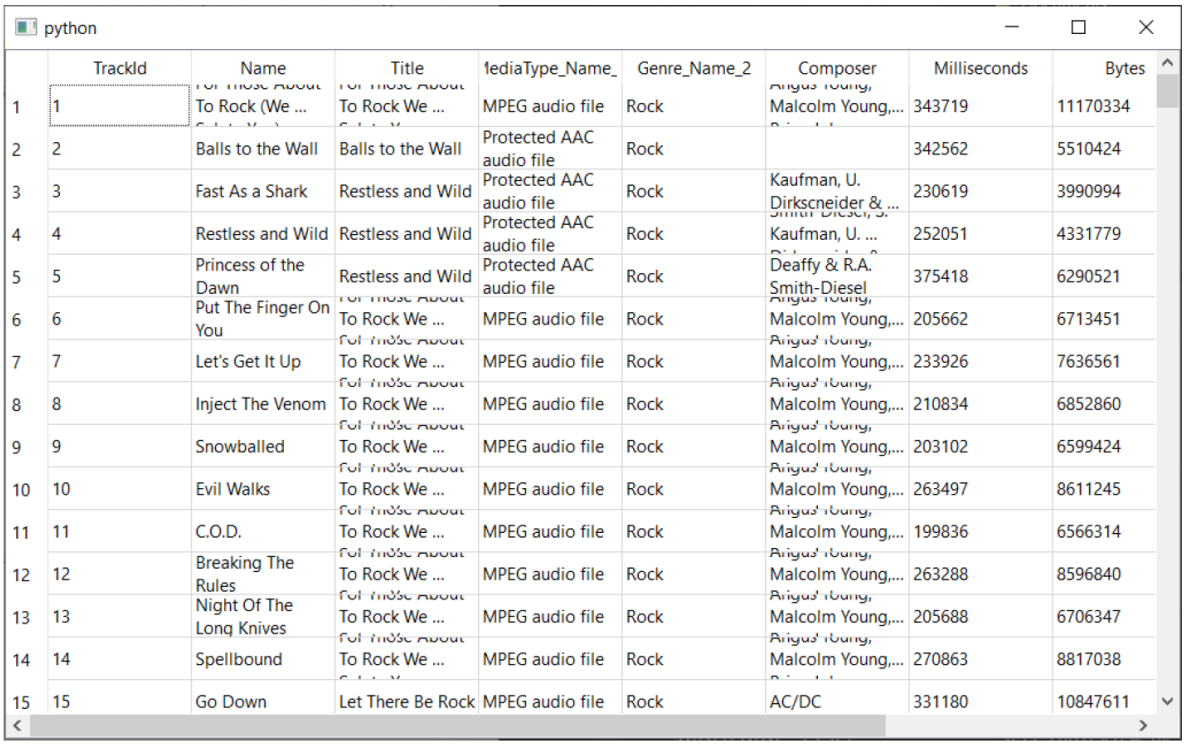
`QSqlRelation` 对象接受三个参数，第一个是我们要从中提取数据的关联表，第二个是该表中的外键列，最后一个是我们要从中提取数据的列。

对于我们的测试数据库中的专辑表，以下操作将从相关表中提取数据，分别对应专辑ID、媒体类型ID和流派ID（对应表中的第3、4、5列）。

Listing 123. databases/tableview_relationalmodel.py

```
        self.model.setTable("Track")
        self.model.setRelation(
            2, QSqlRelation("Album", "AlbumId", "Title")
        )
        self.model.setRelation(
            3, QSqlRelation("MediaType", "MediaTypeId", "Name")
        )
        self.model.setRelation(
            4, QSqlRelation("Genre", "GenreId", "Name")
        )
        self.model.select()
```

运行后，您会发现三个 `_id` 列已被从相关表中提取的数据替换。这些列将采用相关字段的名称，如果不冲突，否则将为其生成一个名称。



	TrackId	Name	Title	MediaType_Name_	Genre_Name_2	Composer	Milliseconds	Bytes
1	1	To Rock (We ...	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	343719	11170334
2	2	Balls to the Wall	Balls to the Wall	Protected AAC audio file	Rock		342562	5510424
3	3	Fast As a Shark	Restless and Wild	Protected AAC audio file	Rock	Kaufman, U. Dirkscneider & ...	230619	3990994
4	4	Restless and Wild	Restless and Wild	Protected AAC audio file	Rock	Kaufman, U. ...	252051	4331779
5	5	Princess of the Dawn	Restless and Wild	Protected AAC audio file	Rock	Deaffy & R.A. Smith-Diesel	375418	6290521
6	6	Put The Finger On You	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	205662	6713451
7	7	Let's Get It Up	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	233926	7636561
8	8	Inject The Venom	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	210834	6852860
9	9	Snowballed	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	203102	6599424
10	10	Evil Walks	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	263497	8611245
11	11	C.O.D.	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	199836	6566314
12	12	Breaking The Rules	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	263288	8596840
13	13	Night Of The Long Knives	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	205688	6706347
14	14	Spellbound	To Rock We ...	MPEG audio file	Rock	Malcolm Young,...	270863	8817038
15	15	Go Down	Let There Be Rock	MPEG audio file	Rock	AC/DC	331180	10847611

图153：显示相关字段的数据。

使用 `QSqlRelationalDelegate` 编辑相关字段。

如果您尝试编辑 `QSqlRelationalTableModel` 中的字段，您会发现一个问题——虽然您可以编辑基础表（这里是 Tracks）中的字段，但您对相关字段（例如 Album Title）所做的任何修改都不会被保存。这些字段目前只是对数据的视图。

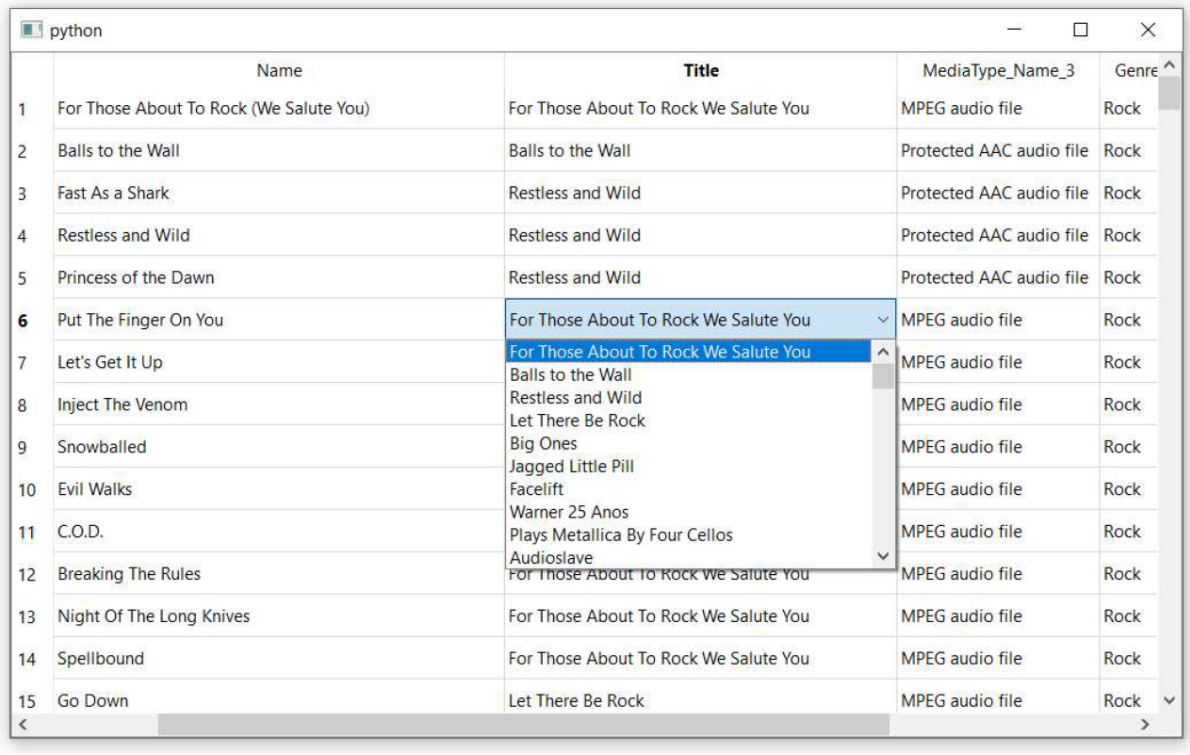
相关字段的有效值受相关表中值的限制 —— 为了获得更多选择，我们需要向相关表中添加另一行。由于选项受到限制，通常有必要将选择项显示在 `QComboBox` 中。Qt 提供了一个模型项委托，可以为我们完成此查找和显示操作 —— `QSqlRelationalDelegate`

Listing 124. databases/tableview_relationalmodel_delegate.py

```
        self.model.setTable("Track")
        self.model.setRelation(
            2, QSqlRelation("Album", "AlbumId", "Title")
        )
        self.model.setRelation(
            3, QSqlRelation("MediaType", "MediaTypeId", "Name")
        )
        self.model.setRelation(
            4, QSqlRelation("Genre", "GenreId", "Name")
        )

        delegate = QSqlRelationalDelegate(self.table)
        self.table.setItemDelegate(delegate)

        self.model.select()
```

该委托会自动处理任何关系字段的映射。我们只需创建一个传递 `QTableView` 实例的委托，然后将生成的委托设置到模型上，一切都会自动完成。

运行此操作时，您将在编辑相关字段时看到下拉菜单。



> 图154：通过下拉列表使相关字段可编辑，使用QSqlRelationalDelegate。

使用 `QSqlQueryModel` 进行通用查询

到目前为止，我们一直在 `QTableView` 中显示整个数据库表，并支持一些可选的列过滤和排序功能。然而，Qt 还允许使用 `QSqlQueryModel` 显示更复杂的查询。在本节中，我们将探讨如何使用 `QSqlQueryModel` 显示 SQL 查询，首先从简单的单表查询开始，然后逐步过渡到关系查询和参数化查询。

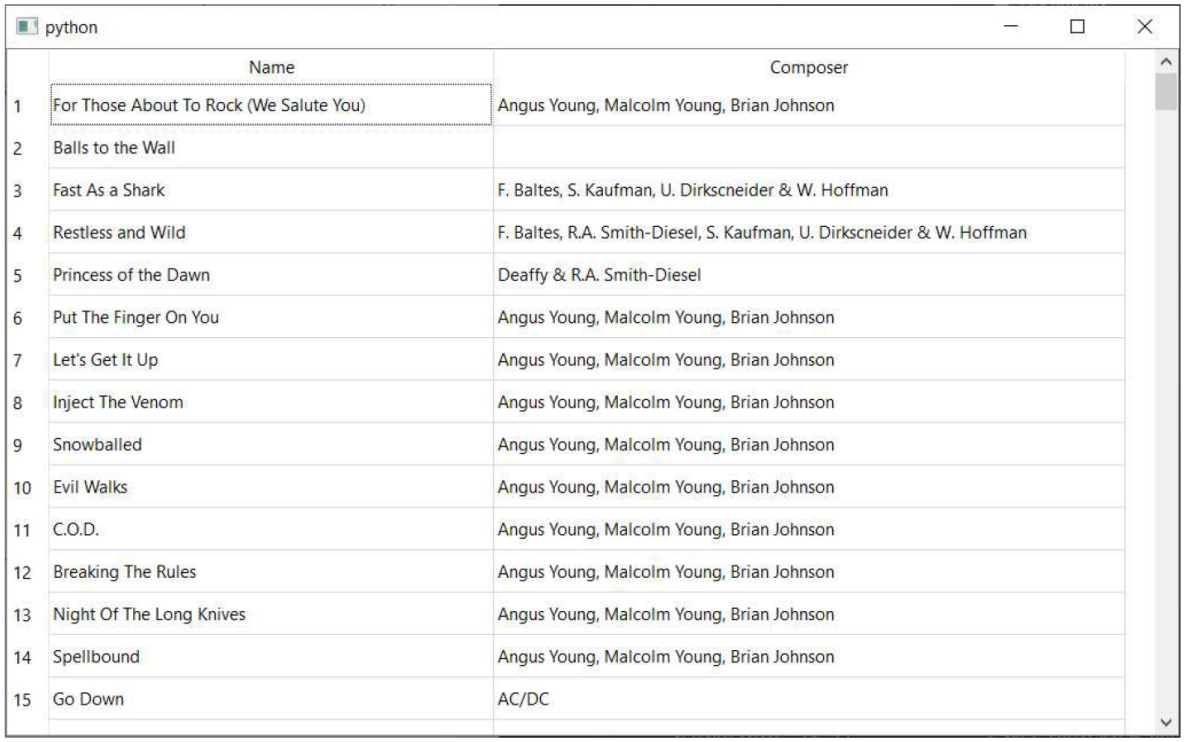
使用此模型进行查询的过程略有不同。与将数据库直接传递给模型构造函数不同，这里我们首先创建一个 `QSqlQuery` 对象，该对象接受数据库连接，然后将该对象传递给模型。

```
query = QSqlQuery("SELECT name, composer FROM track ", db=db)
```

这意味着您可以使用单个 `QSqlQueryModel` 对不同数据库执行查询。该查询的完整示例如下所示：

Listing 125. databases/tableview_querymodel.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtSql import QSqlDatabase, QSqlQuery, QSqlQueryModel
from PyQt6.QtWidgets import QApplication, QMainWindow, QTableView

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.table = QTableView()

        self.model = QSqlQueryModel()
        self.table.setModel(self.model)

        query = QSqlQuery("SELECT Name, Composer FROM track ", db=db)

        self.model.setQuery(query)

        self.setMinimumSize(QSize(1024, 600))
        self.setCentralWidget(self.table)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```



> 图155：执行一个简单的查询

在这个第一个示例中，我们对专辑表执行了一个非常简单的查询，仅返回该表中的两个字段。然而，`QSqlQuery` 对象可用于执行更复杂的查询，包括跨表连接和参数化查询——在参数化查询中，我们可以传递值来修改查询。

> 参数化查询可保护您的应用免受SQL注入攻击。

在下面的示例中，我们扩展了简单的查询，以在专辑表中添加一个相关查找。此外，我们绑定了一个专辑标题参数，该参数用于对专辑表进行包含搜索。

Listing 126. databases/tableview_querymodel_parameter.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtSql import QSqlDatabase, QSqlQuery, QSqlQueryModel
from PyQt6.QtWidgets import QApplication, QMainWindow, QTableView

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()


class MainWindow(QMainWindow):
```

```
    def __init__(self):
        super().__init__()

        self.table = QTableView()

        self.model = QSqlQueryModel()
        self.table.setModel(self.model)

        query = QSqlQuery(db=db)
        query.prepare(
            "SELECT Name, Composer, Album.Title FROM Track "
            "INNER JOIN Album ON Track.AlbumId = Album.AlbumId "
            "WHERE Album.Title LIKE '%' || :album_title || '%' "
        )
        query.bindValue(":album_title", "Sinatra")
        query.exec()

        self.model.setQuery(query)
        self.setMinimumSize(QSize(1024, 600))
        self.setCentralWidget(self.table)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

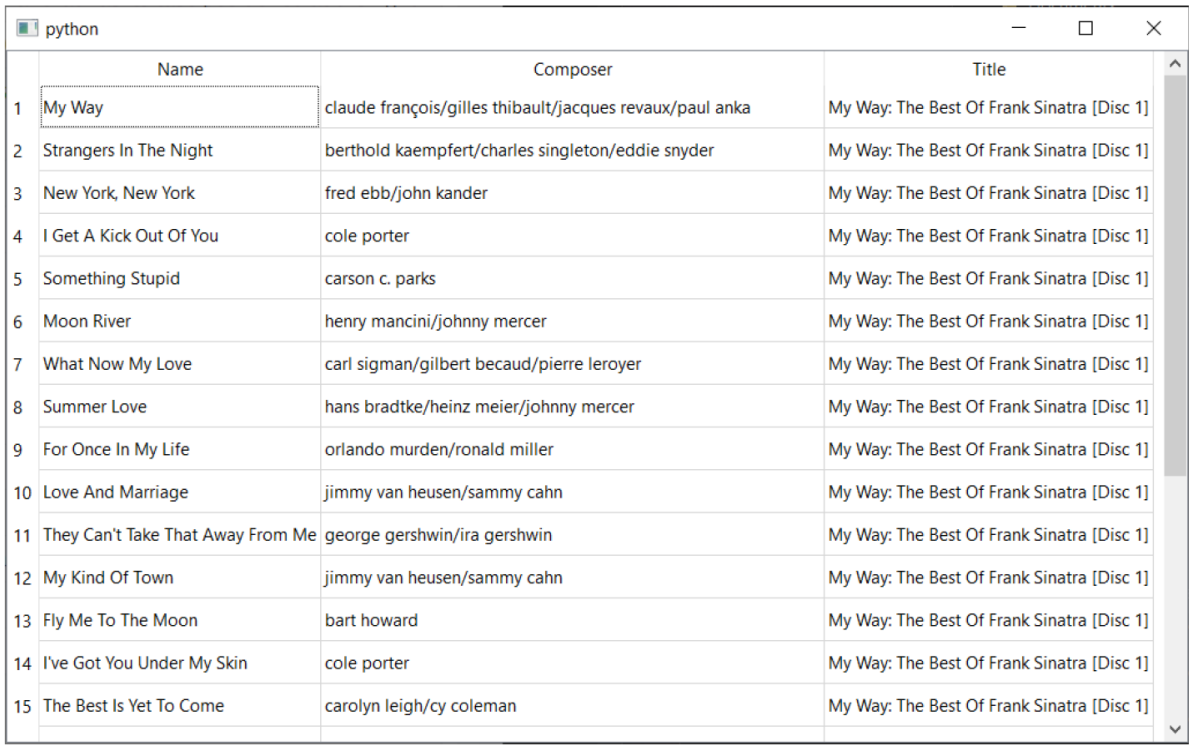
现在我们想要向查询中添加参数，不能直接将查询传递给 `QSqlQuery`，因为这样会立即执行查询，而不会进行参数替换。相反，我们需要将查询传递给 `.prepare()` 方法，告知驱动程序识别查询中的参数并等待值的传入。

接下来，我们使用 `.bindValue()` 方法绑定每个参数，最后调用 `query.exec()` 方法在数据库中执行查询。

此参数化查询等同于以下 SQL 语句：

```
SELECT Name, Composer, Album.Title FROM Track
INNER JOIN Album ON Track.AlbumId = Album.AlbumId
WHERE Album.Title LIKE '%Sinatra%'
```

这将得到以下结果



	Name	Composer	Title
1	My Way	claude françois/gilles thibault/jacques revaux/paul anka	My Way: The Best Of Frank Sinatra [Disc 1]
2	Strangers In The Night	berthold kaempfert/charles singleton/eddie snyder	My Way: The Best Of Frank Sinatra [Disc 1]
3	New York, New York	fred ebb/john kander	My Way: The Best Of Frank Sinatra [Disc 1]
4	I Get A Kick Out Of You	cole porter	My Way: The Best Of Frank Sinatra [Disc 1]
5	Something Stupid	carson c. parks	My Way: The Best Of Frank Sinatra [Disc 1]
6	Moon River	henry mancini/johnny mercer	My Way: The Best Of Frank Sinatra [Disc 1]
7	What Now My Love	carl sigman/gilbert becaud/pierre leroyer	My Way: The Best Of Frank Sinatra [Disc 1]
8	Summer Love	hans bradtke/heinz meier/johnny mercer	My Way: The Best Of Frank Sinatra [Disc 1]
9	For Once In My Life	orlando murden/ronald miller	My Way: The Best Of Frank Sinatra [Disc 1]
10	Love And Marriage	jimmy van heusen/sammy cahn	My Way: The Best Of Frank Sinatra [Disc 1]
11	They Can't Take That Away From Me	george gershwin/ira gershwin	My Way: The Best Of Frank Sinatra [Disc 1]
12	My Kind Of Town	jimmy van heusen/sammy cahn	My Way: The Best Of Frank Sinatra [Disc 1]
13	Fly Me To The Moon	bart howard	My Way: The Best Of Frank Sinatra [Disc 1]
14	I've Got You Under My Skin	cole porter	My Way: The Best Of Frank Sinatra [Disc 1]
15	The Best Is Yet To Come	carolyn leigh/cy coleman	My Way: The Best Of Frank Sinatra [Disc 1]

> 图156：参数化查询的结果

在最后一个示例中，我们添加了三个搜索字段——一个用于歌曲标题、一个用于艺术家、一个用于专辑标题。我们将这些字段的 `.textChanged` 信号连接到一个自定义方法，该方法更新查询的参数。

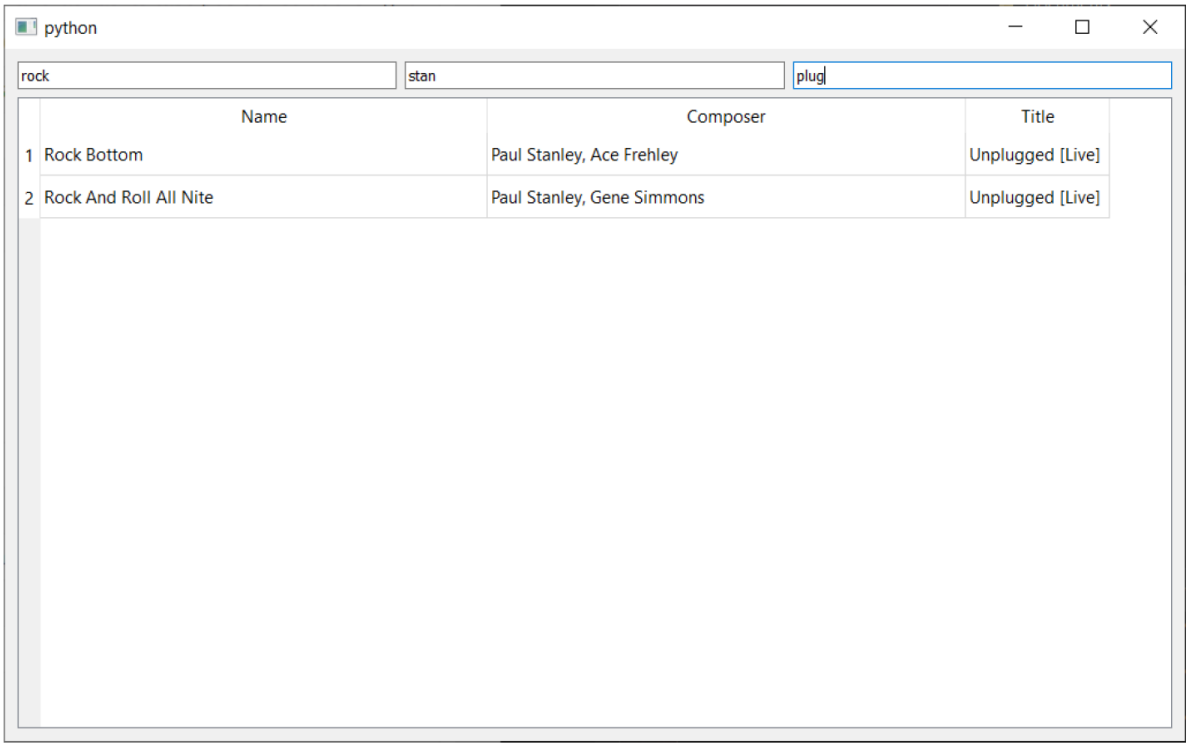
Listing 127. databases/tableview_querymodel_search.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtSql import QSqlDatabase, QSqlQuery, QSqlQueryModel
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLineEdit,
    QMainWindow,
    QTableView,
    QVBoxLayout,
    QWidget,
)

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        container = QWidget()
        layout_search = QHBoxLayout()
```

```
        self.track = QLineEdit()
        self.track.setPlaceholderText("Track name...")
        self.track.textChanged.connect(self.update_query)

        self.composer = QLineEdit()
        self.composer.setPlaceholderText("Artist name...")
        self.composer.textChanged.connect(self.update_query)

        self.album = QLineEdit()
        self.album.setPlaceholderText("Album name...")
        self.album.textChanged.connect(self.update_query)

        layout_search.addWidget(self.track)
        layout_search.addWidget(self.composer)
        layout_search.addWidget(self.album)

        layout_view = QVBoxLayout()
        layout_view.addLayout(layout_search)

        self.table = QTableView()

        layout_view.addWidget(self.table)

        container.setLayout(layout_view)
        self.model = QSqlQueryModel()
        self.table.setModel(self.model)

        self.query = QSqlQuery(db=db)
        self.query.prepare(
            "SELECT Name, Composer, Album.Title FROM Track "
            "INNER JOIN Album ON Track.AlbumId=Album.AlbumId WHERE "
            "Track.Name LIKE '%' || :track_name || '%' AND "
            "Track.Composer LIKE '%' || :track_composer || '%' AND "
            "Album.Title LIKE '%' || :album_title || '%'"
        )

        self.update_query()

        self.setMinimumSize(QSize(1024, 600))
        self.setCentralWidget(container)

    def update_query(self, s=None):

        # 从控件中获取文本值。
        track_name = self.track.text()
        track_composer = self.composer.text()
        album_title = self.album.text()

        self.query.bindValue(":track_name", track_name)
        self.query.bindValue(":track_composer", track_composer)
        self.query.bindValue(":album_title", album_title)

        self.query.exec()
        self.model.setQuery(self.query)
```

```
app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

如果您运行此功能，您可以使用每个字段独立搜索数据库，且每次搜索查询更改时，结果将自动更新。



> 图157：多参数搜索查询的结果

`QDataWidgetMapper`

到目前为止，我们在所有示例中都使用 `QTableView` 以表格形式显示了数据库的输出数据。虽然这种方式通常适合查看数据，但在进行数据输入或编辑时，通常更倾向于以表单形式显示输入内容，这样用户可以直接输入并通过Tab键在字段间切换。

> 这被称为创建、读取、更新和删除（CRUD）操作及接口。

完整的示例代码如下所示。

Listing 128. databases/widget_mapper.py

```
import os
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtSql import QSqlDatabase, QSqlTableModel
from PyQt6.QtWidgets import (
```

```
    QApplication,
    QComboBox,
    QDataWidgetMapper,
    QDoubleSpinBox,
    QFormLayout,
    QLabel,
    QLineEdit,
    QMainWindow,
    QSpinBox,
    QWidget,
)

basedir = os.path.dirname(__file__)

db = QSqlDatabase("QSQLITE")
db.setDatabaseName(os.path.join(basedir, "chinook.sqlite"))
db.open()


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        form = QFormLayout()

        self.track_id = QSpinBox()
        self.track_id.setRange(0, 2147483647)
        self.track_id.setDisabled(True)
        self.name = QLineEdit()
        self.album = QComboBox()
        self.media_type = QComboBox()
        self.genre = QComboBox()
        self.composer = QLineEdit()

        self.milliseconds = QSpinBox()
        self.milliseconds.setRange(0, 2147483647) #1
        self.milliseconds.setSingleStep(1)

        self.bytes = QSpinBox()
        self.bytes.setRange(0, 2147483647)
        self.bytes.setSingleStep(1)

        self.unit_price = QDoubleSpinBox()
        self.unit_price.setRange(0, 999)
        self.unit_price.setSingleStep(0.01)
        self.unit_price.setPrefix("$")
        form.addRow(QLabel("Track ID"), self.track_id)
        form.addRow(QLabel("Track name"), self.name)
        form.addRow(QLabel("Composer"), self.composer)
        form.addRow(QLabel("Milliseconds"), self.milliseconds)
        form.addRow(QLabel("Bytes"), self.bytes)
        form.addRow(QLabel("Unit Price"), self.unit_price)

        self.model = QSqlTableModel(db=db)

        self.mapper = QDataWidgetMapper() #2
```

```
        self.mapper.setModel(self.model)

        self.mapper.addMapping(self.track_id, 0) #3
        self.mapper.addMapping(self.name, 1)
        self.mapper.addMapping(self.composer, 5)
        self.mapper.addMapping(self.milliseconds, 6)
        self.mapper.addMapping(self.bytes, 7)
        self.mapper.addMapping(self.unit_price, 8)

        self.model.setTable("Track")
        self.model.select() #4

        self.mapper.toFirst() #5

        self.setMinimumSize(QSize(400, 400))

        widget = QWidget()
        widget.setLayout(form)
        self.setCentralWidget(widget)

app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 1. 控件必须配置为接受表中的所有有效值。
> 2. 所有控件使用一个 `QDataWidgetMapper`。
> 3. 控件映射到 `_columns`。
> 4. 执行选择以填充模型。
> 5. 将映射器向前移动到第一个记录

运行此示例后，您将看到以下窗口。`self.mapper.toFirst()` 调用选择表中的第一条记录，然后将其显示在映射的控件中。

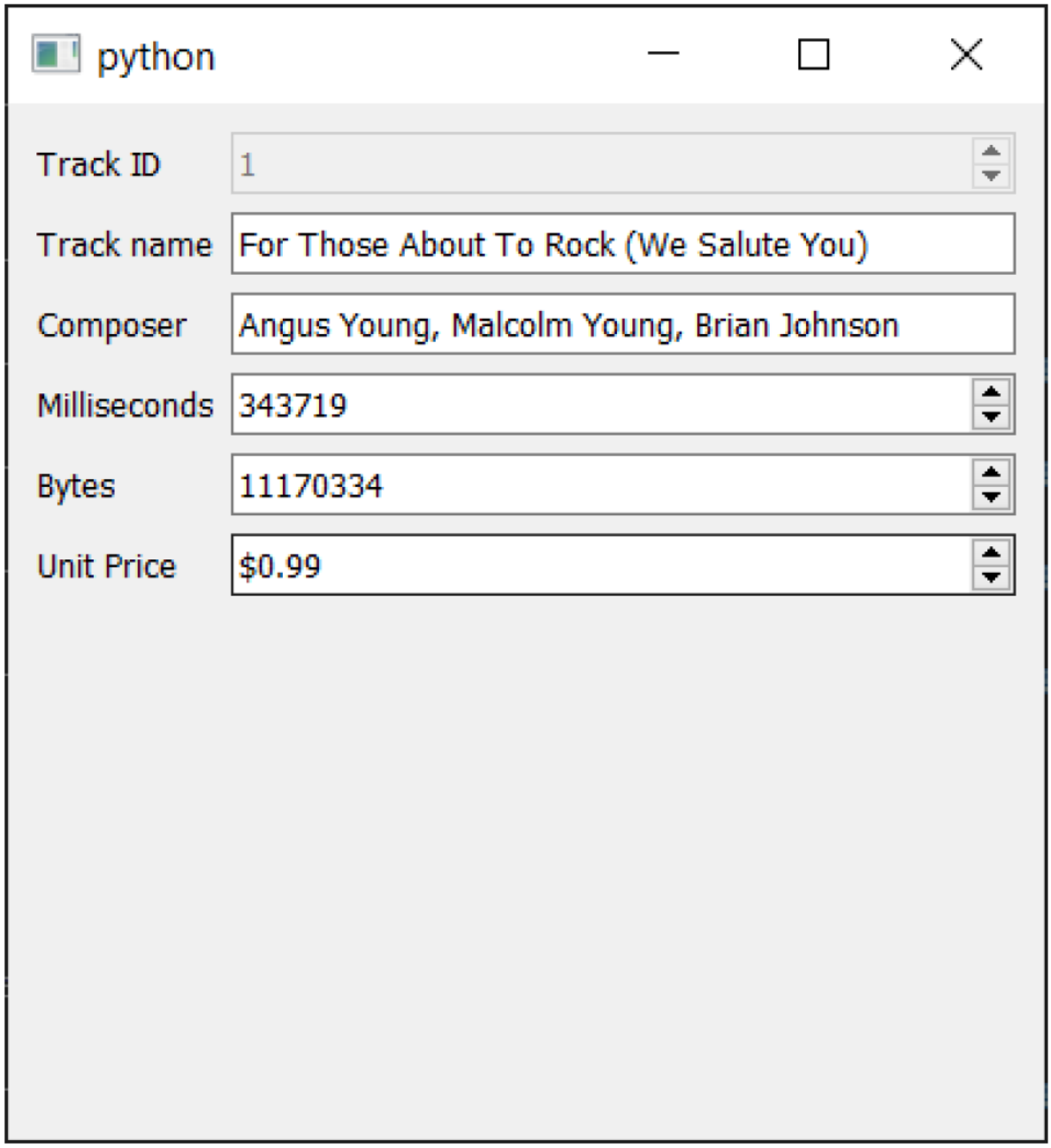


图158：通过映射控件查看记录

目前，我们无法更改正在查看的记录，也无法保存对记录所做的任何更改。为了实现这一点，我们可以添加 3 个按钮——分别用于浏览记录的前一个和下一个，以及保存更改到数据库。为此，我们可以将一些 `QPushButton` 控件连接到映射器槽 `.toPrevious`、`.toNext` 和 `.submit`。

请您更新 `__init__` 方法的结尾，添加以下内容，将控件添加到现有的布局中

Listing 129. databases/widget_mapper_controls.py

```
        self.setMinimumSize(QSize(400, 400))

        controls = QHBoxLayout()

        prev_rec = QPushButton("Previous")
        prev_rec.clicked.connect(self.mapper.toPrevious)

        next_rec = QPushButton("Next")
        next_rec.clicked.connect(self.mapper.toNext)
```

```
        save_rec = QPushButton("Save Changes")
        save_rec.clicked.connect(self.mapper.submit)

        controls.addWidget(prev_rec)
        controls.addWidget(next_rec)
        controls.addWidget(save_rec)

        layout.addLayout(form)
        layout.addLayout(controls)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```

您还需要更新文件顶部的导入语句，以导入 `QPushButton` 和 `QHBoxLayout`

Listing 130. databases/widget_mapper_controls.py

```
from PyQt6.QtWidgets import (
    QApplication,
    QComboBox,
    QDataWidgetMapper,
    QDoubleSpinBox,
    QFormLayout,
    QHBoxLayout,
    QLabel,
    QLineEdit,
    QMainWindow,
    QPushButton,
    QSpinBox,
    QVBoxLayout,
    QWidget,
)
```

现在您可以浏览专辑表中的记录，修改专辑数据并提交这些更改到数据库。此示例的完整源代码位于书籍源代码中的 `databases/widget_mapper_controls.py` 文件中。

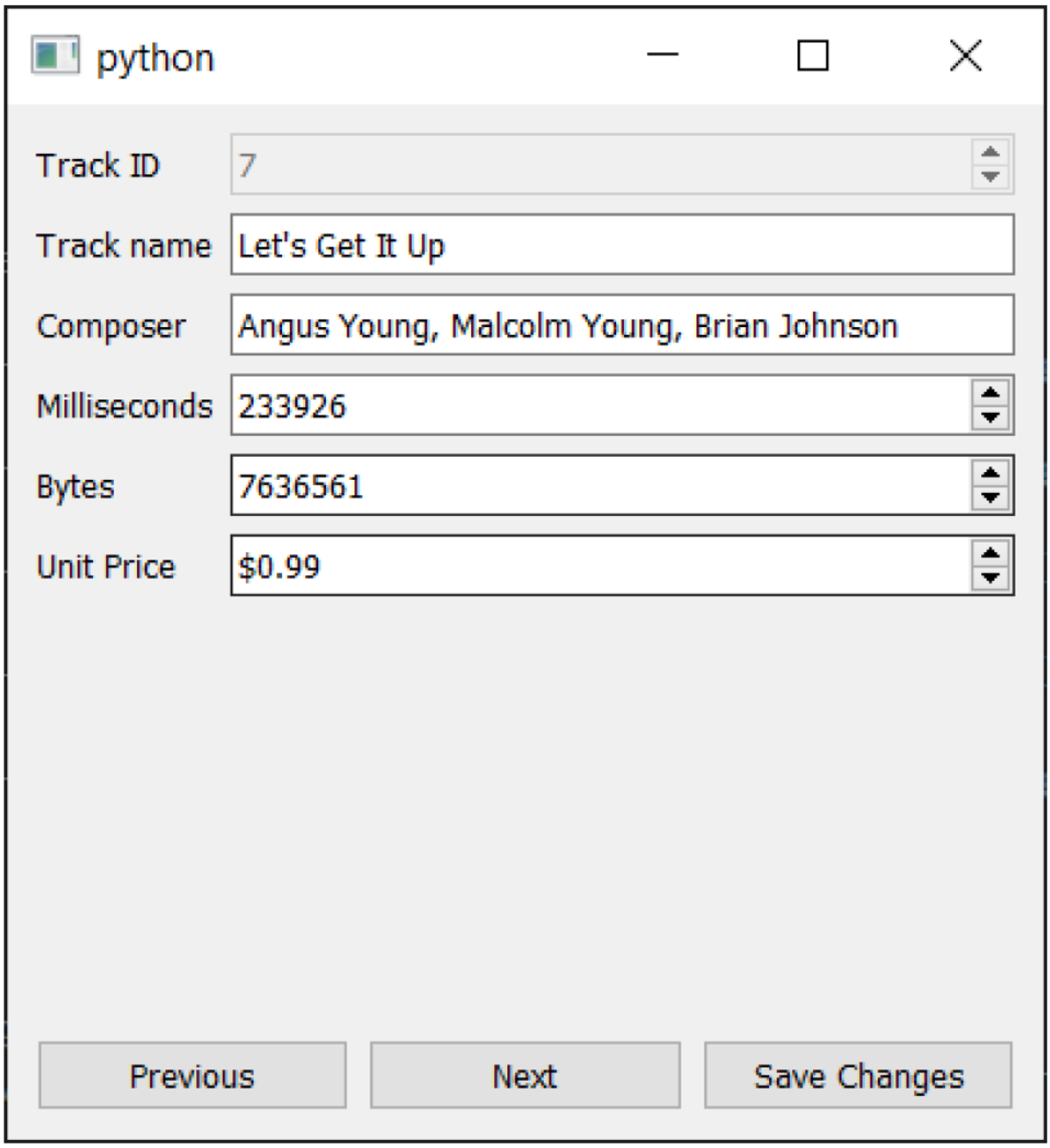


图159：您可以使用上一页/下一页控制按钮来查看记录，还可以点击保存以提交。

使用 `QSqlDatabase` 进行身份验证

到目前为止，我们在示例中使用了 SQLite 数据库文件。但通常情况下，您可能希望连接到远程 SQL 服务器。这需要添加一些额外参数，包括数据库所在的主机名，以及适当的用户名和密码。

```
# 建立数据库连接。
db = QSqlDatabase('<driver>')
db.setHostName('<localhost>')
db.setDatabaseName('<databasename>')
db.setUserName('<username>')
db.setPassword('<password>')
db.open()
```

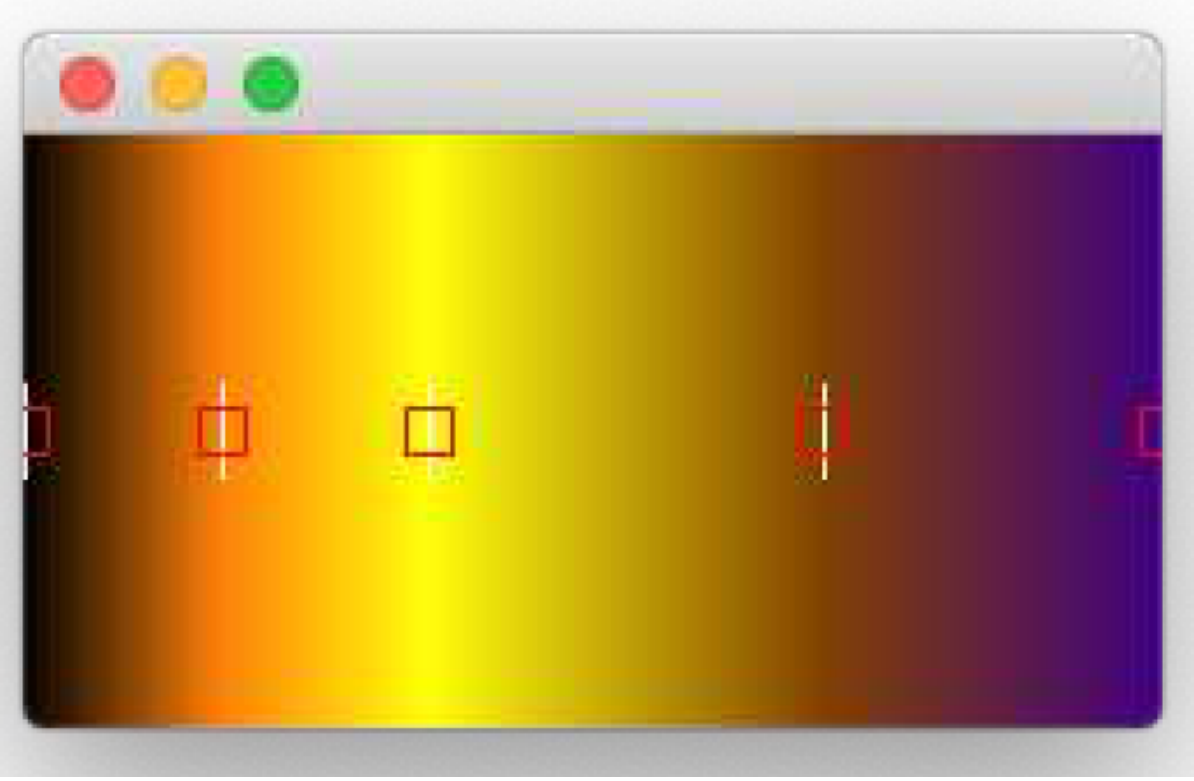
注意：`<driver>` 的值可以是以下任何一个 `['QSQLITE','QMYSQL', 'QMYSQL3', 'QODBC', 'QODBC3', 'QPSQL', 'QPSQL7']`。要获取系统上的此列表，请运行 `QSqlDatabase.drivers()`。

就这样！一旦连接建立，模型将与之前完全相同地运作。

自定义控件

如我们所见，Qt 内置了各种控件，您可以使用这些控件来构建应用程序。即使如此，有时这些简单的控件还是不够用——也许您需要一些自定义类型的输入，或者希望以独特的方式可视化数据。在 Qt 中，您可以自由创建自己的控件，无论是从头开始创建，还是组合现有控件。

在本章中，我们将了解如何使用位图图形和自定义信号来创建您自己的控件



> 图160：自定义颜色渐变输入，我们库中的控件之一

> 您可能还想查看我们的 自定义控件库。

21. Qt 中的位图图形

在 PyQt6 中创建自定义控件的第一步是了解位图（基于像素）图形操作。所有标准控件都以位图的形式绘制在构成控件形状的矩形“画布”上。一旦您了解了其工作原理，就可以绘制任何您喜欢的自定义控件！

位图是由像素组成的矩形网格，其中每个像素（及其颜色）由一定数量的“位”来表示。它们与矢量图形不同，矢量图形中图像以一系列线条（或矢量）绘图形状的形式存储，这些形状用于构成图像。如果您在屏幕上查看矢量图形，它们正在被栅格化——转换为位图图像——以像素形式显示在屏幕上。

在本教程中，我们将介绍 `QPainter`，这是 Qt 用于执行位图图形操作的 API，也是绘制您自己的控件的基础。我们将介绍一些基本的绘图操作，最后将它们整合在一起，创建我们自己的小绘图应用程序。

QPainter

Qt 中的位图绘图操作通过 `QPainter` 类进行处理。这是一个通用接口，可用于在各种表面上绘图，包括 `QPixmap` 等。在本章中，我们将介绍 `QPainter` 的绘图方法，首先在 `QPixmap` 表面上使用基本操作，然后利用所学知识构建一个简单的 Paint 应用程序。

为了便于演示，我们将使用以下存根应用程序，该应用程序负责创建容器（`QLabel`）、创建像素图画布、将像素图画布设置到容器中，并将容器添加到主窗口。

Listing 131. bitmap/stub.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.label = QLabel()
        self.canvas = QPixmap(400, 300) #1
        self.canvas.fill(Qt.GlobalColor.white) #2

        self.setCentralWidget(self.label)
        self.draw_something()

    def draw_something(self):
        pass


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 1. 创建我们将要绘制的 `QPixmap` 对象。
> 2. 用白色填充整个画布（以便我们能看到我们的线条）。

> 为什么使用 `QLabel` 进行绘制？`QLabel` 控件还可以用于显示图像，它是显示 `QPixmap` 的最简单的控件。

我们需要先用白色填充画布，因为根据平台和当前的深色模式，背景颜色可能从浅灰色到黑色不等。我们可以从绘制一些非常简单的内容开始。

Listing 132. /bitmap/line.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QPainter, QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.label = QLabel()
        self.canvas = QPixmap(400, 300) #1
        self.canvas.fill(Qt.GlobalColor.white) #2
        self.label.setPixmap(self.canvas)
        self.setCentralWidget(self.label)
        self.draw_something()

    def draw_something(self):
        painter = QPainter(self.canvas)
        painter.drawLine(10, 10, 300, 200) #3
        painter.end()
        self.label.setPixmap(self.canvas)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 1. 创建我们将要绘制的 `QPixmap` 对象。
> 2. 用白色填充整个画布（以便我们能看到我们的线条）
> 3. 从 (10, 10) 到 (300, 200) 画一条直线。坐标为 (x, y)，其中 (0, 0) 在左上角。

将此内容保存到文件中并运行，您应该会看到以下内容——窗口框架内的一条黑色

线

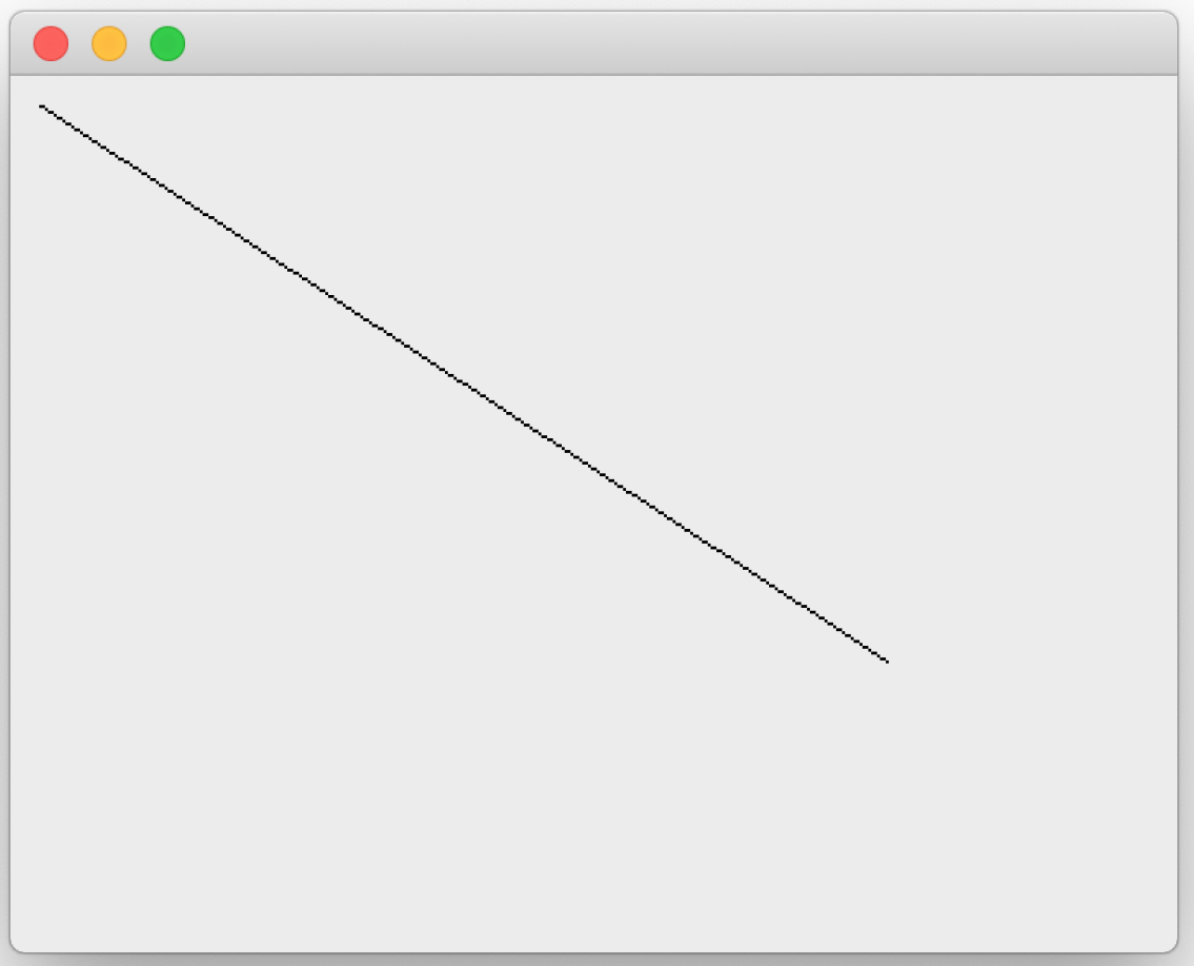


图161：画布上的一条黑色直线。

所有绘制操作均在 `draw_something` 方法中完成——我们创建一个 `QPainter` 实例，传入画布（`self.label.pixmap()`），然后发出绘制直线的命令。最后调用 `.end()` 方法关闭绘图器并应用更改。

> 通常情况下，您还需要调用 `.update()` 来触发控件的刷新，但由于我们在应用程序窗口显示之前就进行了绘制，因此刷新会自动发生。

`QPainter` 的坐标系将 (0, 0) 置于画布的左上角，其中 x 值向右增加，y 值向下增加。这可能与您习惯的图形绘制不同，因为在图形绘制中，(0, 0) 通常位于左下角。

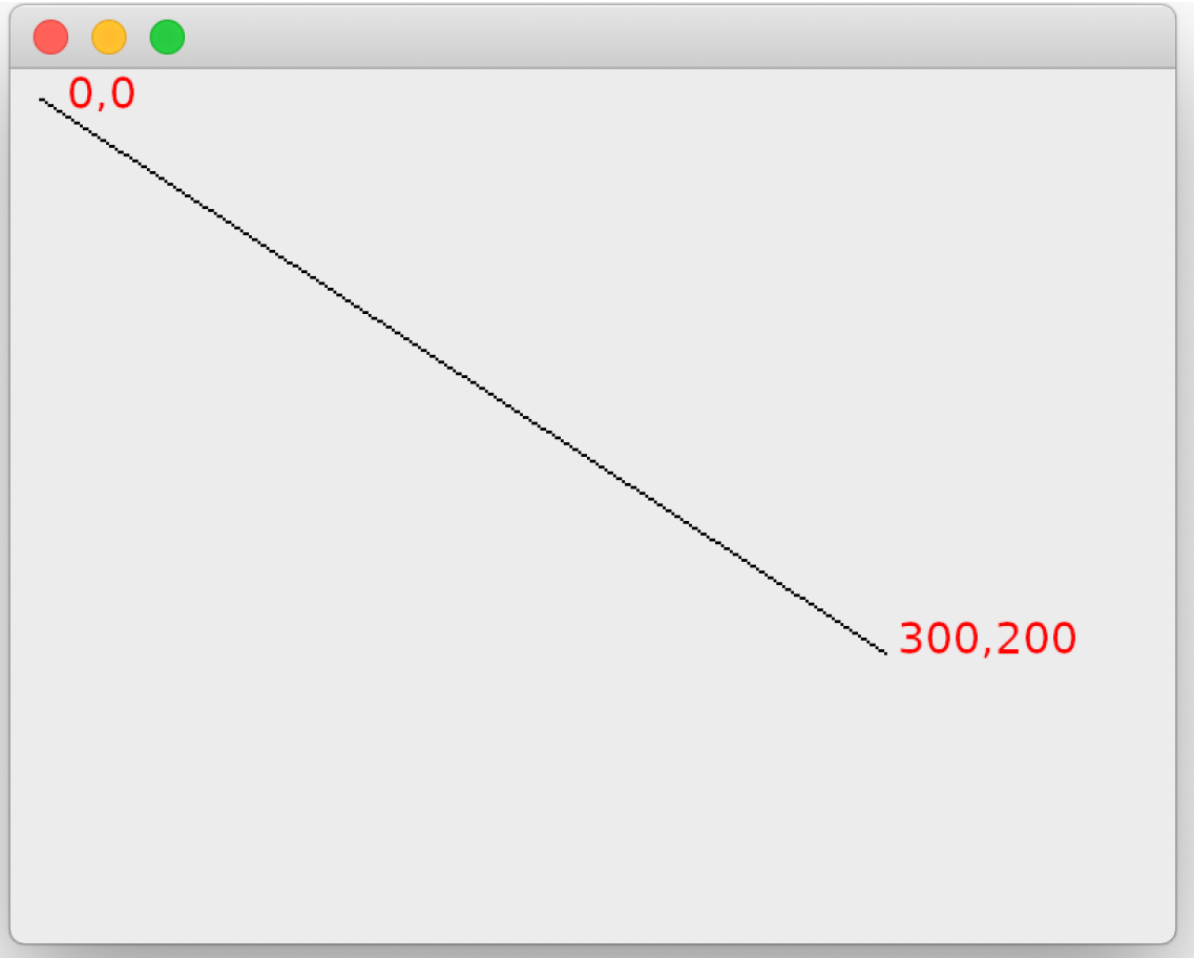


图162：标注有坐标的黑色线条

绘制基本图形

`QPainter` 提供了大量用于在位图表面上绘制形状和线条的方法（在 5.12 版本中，有 192 个 `QPainter` 专用的非事件方法）。好消息是，其中大多数都是重载方法，它们只是调用相同基类方法的不同方式。

例如，有5种不同的 `drawLine` 方法，它们都绘制相同的线，但定义要绘制内容的坐标的方式不同。

方法	描述
`drawLine(line)`	绘制一个 `QLine` 实例
`drawLine(line)`	绘制一个 `QLineF` 实例
`drawLine(x1, y1, x2, y2)`	在 (x1, y2) 和 (x2, y2) 之间画一条直线。（两者均为 `int`）
`drawLine(p1, p2)`	在 (x1, y2) 和 (x2, y2) 之间画一条直线。（两者均为 `QPoint`）
`drawLine(p1, p2)`	在 (x1, y2) 和 (x2, y2) 之间画一条直线。（两者均为 `QPointF`）

如果您在想 `QLine` 和 `QLineF` 有什么区别，那么后者的坐标是浮点数。这在其他计算结果是浮点数时很方便，但其他情况下则不然。

忽略 F 变体，我们有三种独特的方式来绘制一条直线——使用直线对象、使用两组坐标 `(x1, y1)` 和 `(x2, y2)`，或者使用两个 `QPoint` 对象。当您发现 `QLine` 本身被定义为 `QLine(const QPoint &p1, const QPoint & p2)` 或 `QLine(int x1, int y1, int x2, int y2)` 时，您会发现它们实际上是完全相同的东西。不同的调用签名只是为了方便。

> 给定坐标 x1, y1, x2, y2，两个 `QPoint` 对象将被定义为 `QPoint(x1, y1)` 和 `QPoint(x2, y2)`。

因此，排除重复项后，我们得到以下绘图操作：`drawArc`、`drawChord`、`drawConvexPolygon`、`drawEllipse`、`drawLine`、`drawPath`、`drawPie`、`drawPoint`、`drawPolygon`、`drawPolyline`、`drawRect`、`drawRects` 和 `drawRoundedRect`。为了避免被过多的内容淹没，我们将首先专注于基本形状和线条，待掌握基础后再回过头来处理更复杂的操作。

> 对于每个示例，请在您的示例应用程序中替换 `draw_something` 方法，然后重新运行以查看输出结果。

`drawPoint`

这会在画布上的指定位置绘制一个点或像素。每次调用 `drawPoint` 方法都会绘制一个像素。您可以用以下代码替换您的 `draw_something` 代码：

Listing 133. bitmap/point.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        painter.drawPoint(200, 150)
        painter.end()
        self.label.setPixmap(self.canvas)
```

如果您重新运行该文件，您会看到一个窗口，但这次窗口中央有一个单一的点，颜色为黑色。您可能需要移动窗口来找到它。

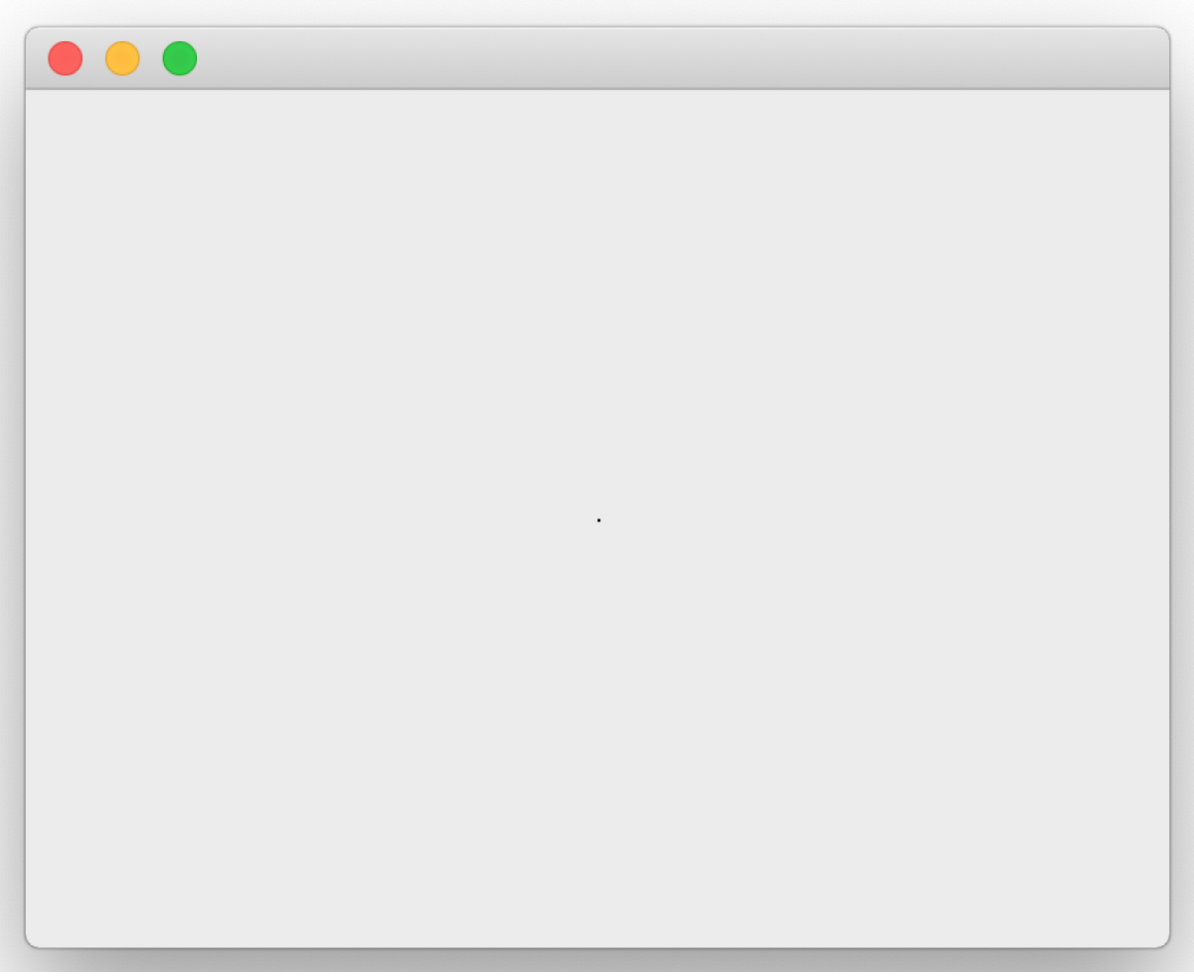


图163：使用QPainter绘制单个点（像素）

这看起来确实没什么特别的。为了让事情更有趣，我们可以更改我们正在绘制的点的颜色和大小。在 PyQt6 中，线条的颜色和粗细是通过 `QPainter` 上的活动画笔来定义的。您可以通过创建一个 `QPen` 实例并应用它来设置这些属性。

Listing 134. bitmap/point_with_pen.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(40)
        pen.setColor(QColor("red"))
        painter.setPen(pen)
        painter.drawPoint(200, 150)
        painter.end()
        self.label.setPixmap(self.canvas)
```

这将产生一个稍显有趣的结果。

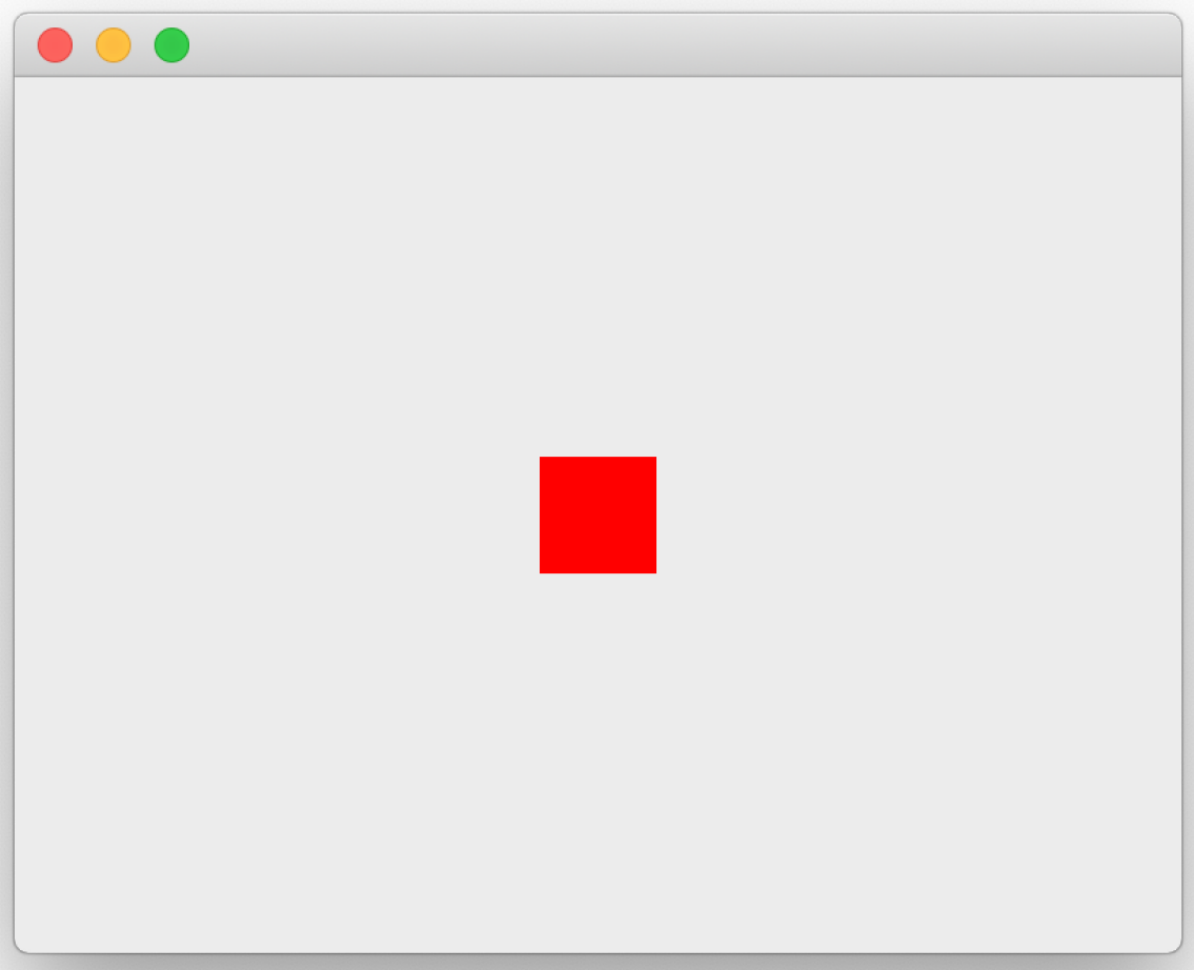


图164：一个大红点

您可以自由地使用 `QPainter` 执行多次绘制操作，直到绘图器结束。在画布上绘制非常快速——这里我们正在随机绘制 10000 个点。

Listing 135. bitmap/points.py

```
from random import choice, randint #1

    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(3)
        painter.setPen(pen)

        for n in range(10000):
            painter.drawPoint(
                200 + randint(-100, 100),
                150 + randint(-100, 100), # x # y
            )
        painter.end()
        self.label.setPixmap(self.canvas)
```

1. 在文件开头添加此导入语句

这些点宽度为3像素，颜色为黑色（默认画笔）。

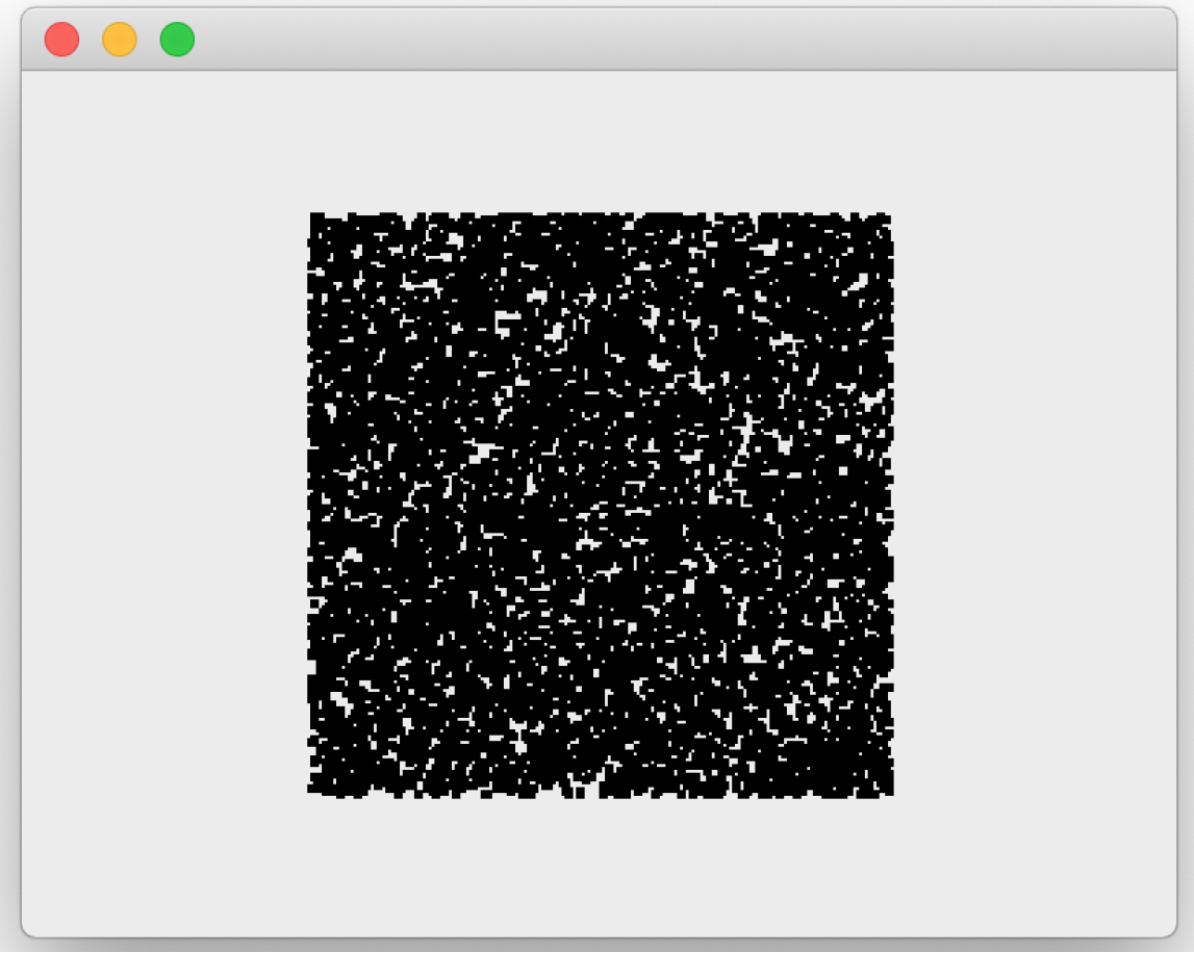


图165：画布上10000个3像素的点

在绘图过程中，您经常需要更新当前的画笔——例如，以不同颜色绘制多个点，同时保持其他特性（如宽度）不变。为了实现这一点，而无需每次都重新创建一个新的 `QPen` 实例，您可以从 `QPainter` 中获取当前活动的画笔，使用 `pen = painter.pen()`。您还可以多次重新应用现有的画笔，每次都对其进行修改。

Listing 136. bitmap/points_color.py

```
    def draw_something(self):
        colors = [
            "#FFD141",
            "#376F9F",
            "#0D1F2D",
            "#E9EBEF",
            "#EB5160",
        ]

        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(3)
        painter.setPen(pen)

        for n in range(10000):
            # pen = painter.pen() 您可以在这里获取活动笔。
            pen.setColor(QColor(choice(colors)))
            painter.setPen(pen)
            painter.drawPoint(
                200 + randint(-100, 100),
```

```
                150 + randint(-100, 100), # x # y
            )
    painter.end()
    self.label.setPixmap(self.canvas)
```

将生成以下输出

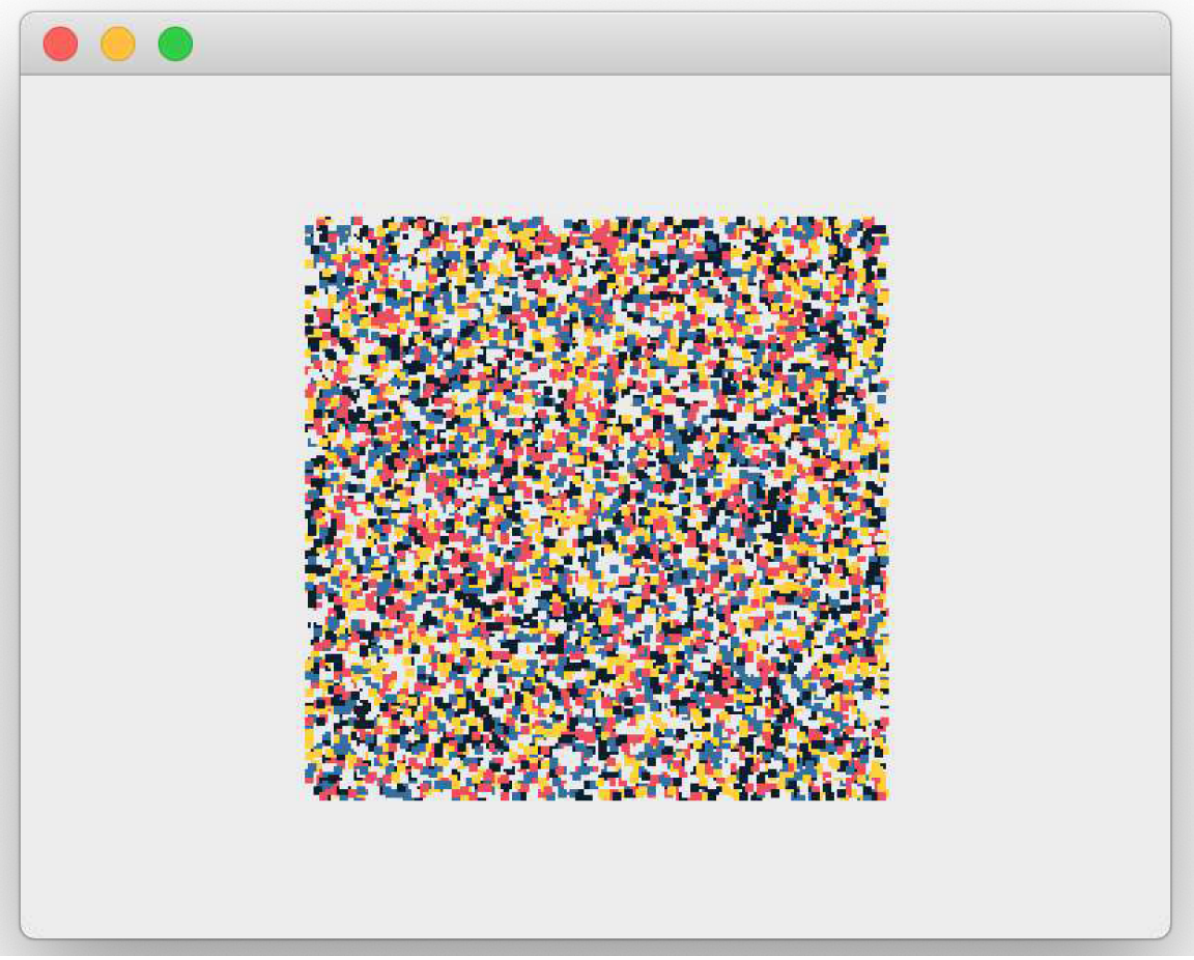


图166：随机分布的三像素宽的点

在 `QPainter` 上只能有一个 `QPen` 处于活动状态——即当前的笔。

这大概就是用画笔在屏幕上画点能带来的全部乐趣了，所以我们接下来要看看其他一些绘图操作。

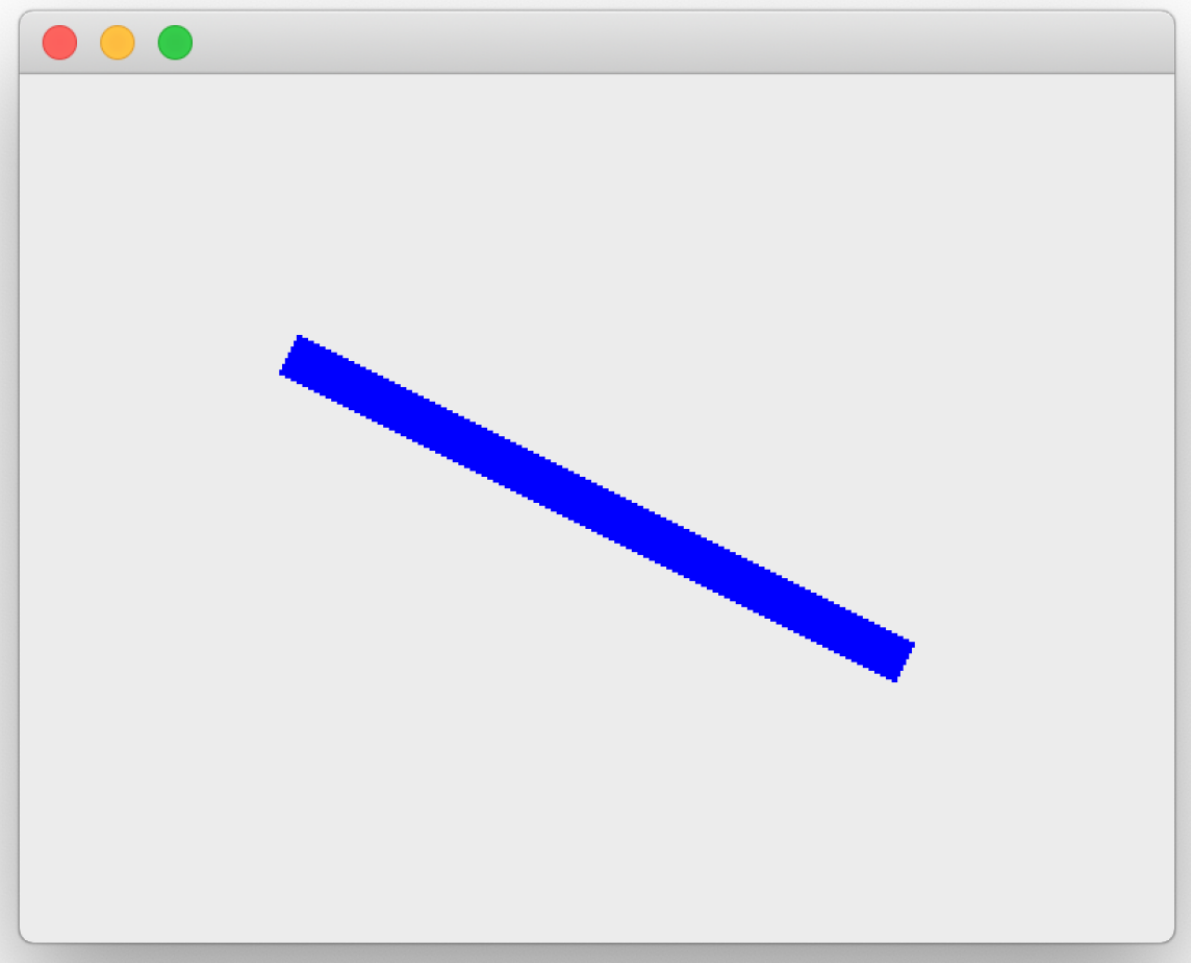
`drawLine`

我们在画布上已经画了一条线来测试功能是否正常。但我们尚未尝试将画笔设置为控制线条外观。

Listing 137. bitmap/line_with_pen.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(15)
        pen.setColor(QColor("blue"))
        painter.setPen(pen)
        painter.drawLine(QPoint(100, 100), QPoint(300, 200))
        painter.end()
        self.label.setPixmap(self.canvas)
```

在此示例中，我们还使用 `QPoint` 来定义要用直线连接的两个点，而不是分别传入 `x1、y1、x2、y2` 参数——请记住，这两种方法在功能上是完全相同的。



> 图167：一个蓝色的粗线

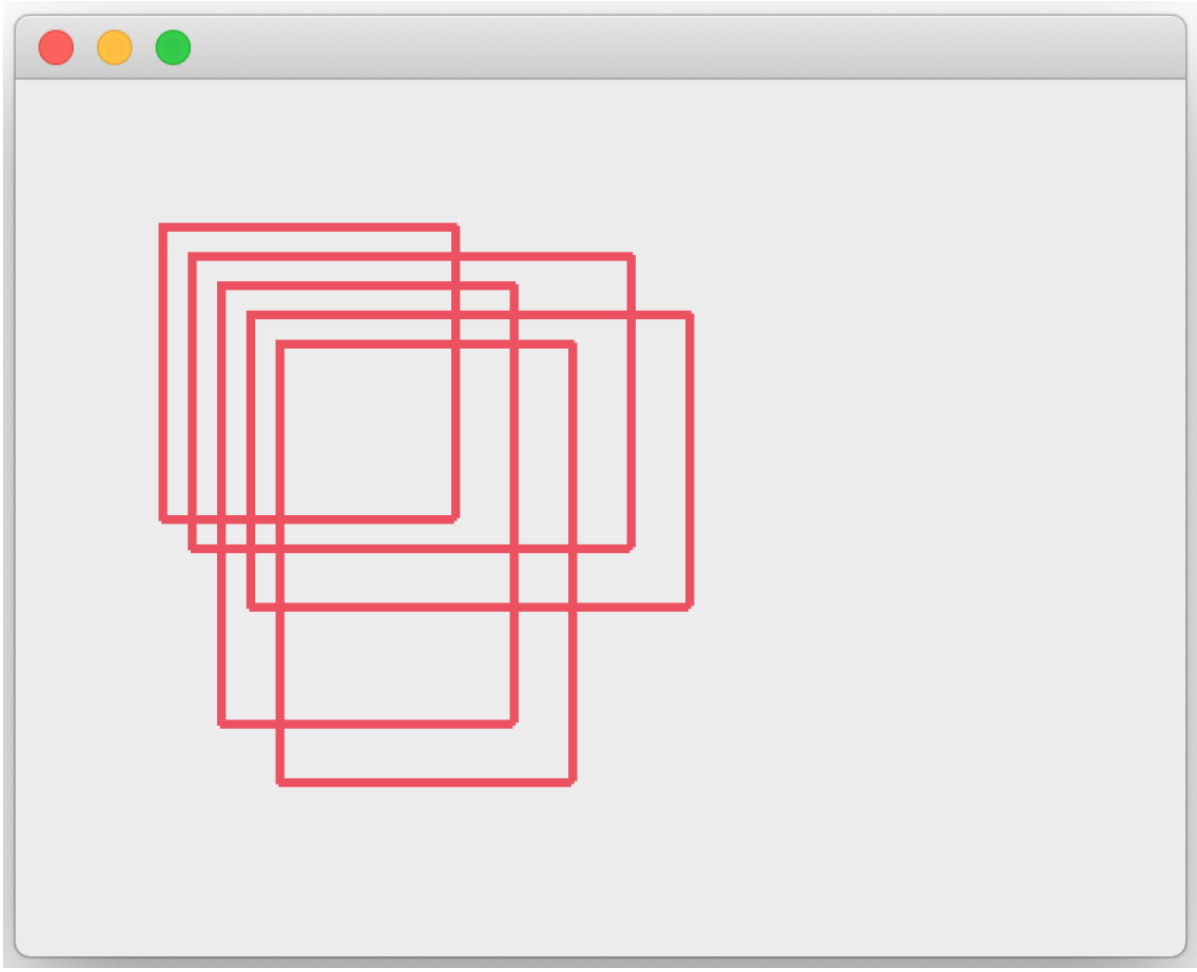
`drawRect`，`drawRects` 和 `drawRoundedRect`

这些函数均用于绘制矩形，矩形可通过一系列点定义，或通过 `QRect` 或 `QRectF` 实例定义。

Listing 138. bitmap/rect.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(3)
        pen.setColor(QColor("#EB5160"))
        painter.setPen(pen)
        painter.drawRect(50, 50, 100, 100)
        painter.drawRect(60, 60, 150, 100)
        painter.drawRect(70, 70, 100, 150)
        painter.drawRect(80, 80, 150, 100)
        painter.drawRect(90, 90, 100, 150)
        painter.end()
        self.label.setPixmap(self.canvas)
```

> 正方形只是一个宽度和高度相等的矩形。



> 图168：画出来的矩形

您还可以将多个对 `drawRect` 的调用替换为对 `drawRects` 的单次调用，并传入多个 `QRect` 对象。这将产生完全相同的结果。

```
painter.drawRects(
    QtCore.QRect(50, 50, 100, 100),
    QtCore.QRect(60, 60, 150, 100),
    QtCore.QRect(70, 70, 100, 150),
    QtCore.QRect(80, 80, 150, 100),
    QtCore.QRect(90, 90, 100, 150),
)
```

在 PyQt6 中，可以通过设置当前活动的绘图刷来填充绘制的形状，将 `QBrush` 实例传递给 `painter.setBrush()` 方法。以下示例将所有矩形填充为带图案的黄色。

Listing 139. bitmap/rect_with_brush.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(3)
        pen.setColor(QColor("#376F9F"))
        painter.setPen(pen)
        brush = QBrush()
        brush.setColor(QColor("#FFD141"))
        brush.setStyle(Qt.BrushStyle.Dense1Pattern)
        painter.setBrush(brush)
        painter.drawRects(
            QRect(50, 50, 100, 100),
            QRect(60, 60, 150, 100),
            QRect(70, 70, 100, 150),
            QRect(80, 80, 150, 100),
            QRect(90, 90, 100, 150),
        )
        painter.end()
        self.label.setPixmap(self.canvas)
```

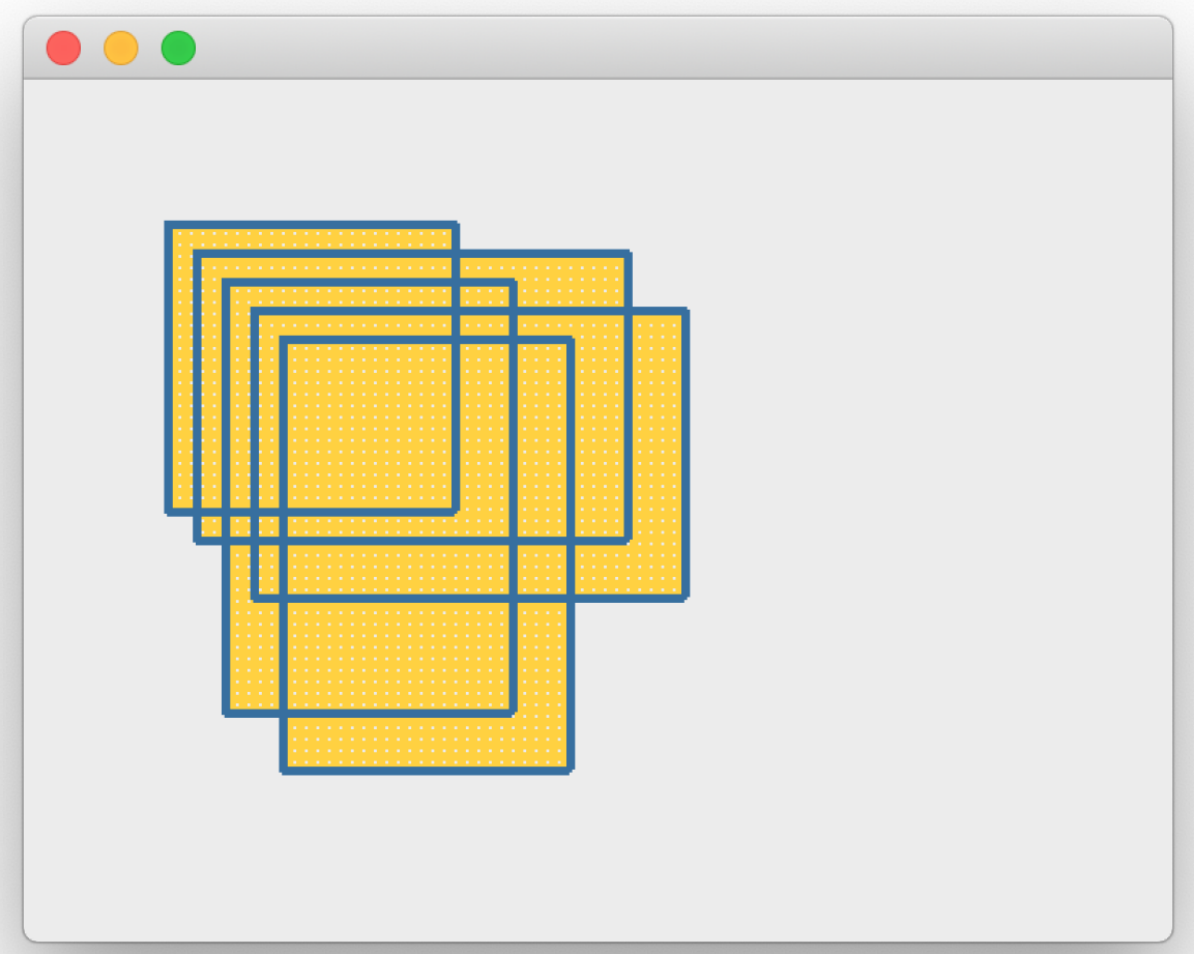


图169：被填上颜色的矩形

至于笔，一个画家只能使用一支画笔，但您可以在绘画时在它们之间切换或更改它们。有 许多画笔样式图案 可供选择。您可能最常使用的是 `Qt.BrushStyle.SolidPattern`。

您必须设置样式才能看到任何填充，因为默认值为 `Qt.BrushStyle.NoBrush`。

`drawRoundedRect` 方法绘制一个矩形，但带有圆角，因此需要额外传入两个参数，分别表示角点的 x 和 y 半径。

Listing 140. bitmap/roundrect.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(3)
        pen.setColor(QColor("#376F9F"))
        painter.setPen(pen)
        painter.drawRoundedRect(40, 40, 100, 100, 10, 10)
        painter.drawRoundedRect(80, 80, 100, 100, 10, 50)
        painter.drawRoundedRect(120, 120, 100, 100, 50, 10)
        painter.drawRoundedRect(160, 160, 100, 100, 50, 50)
        painter.end()
        self.label.setPixmap(self.canvas)
```

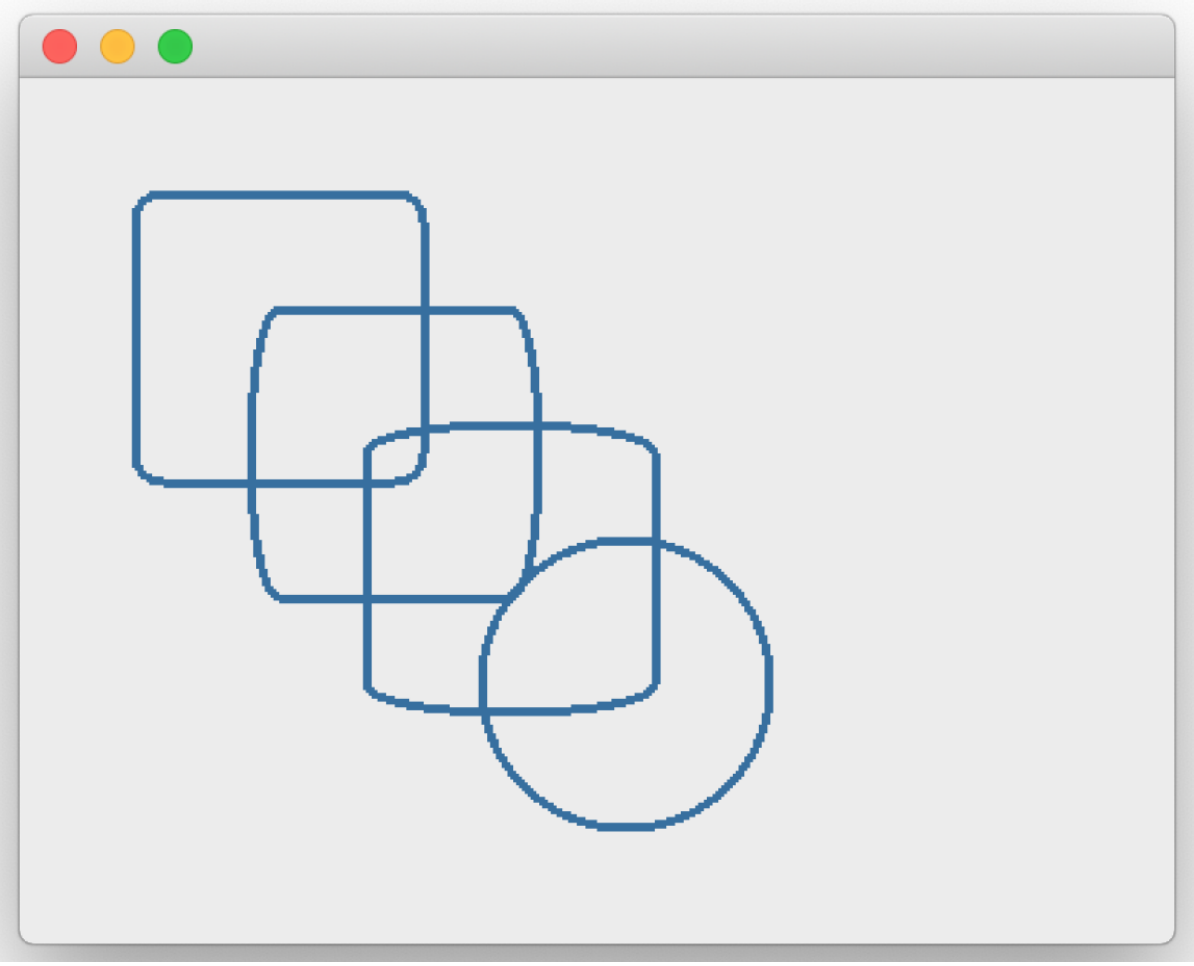


图170：圆角矩形

有一个可选的最后一个参数，用于在 x 和 y 轴的椭圆半径之间切换，这些半径以绝对像素值定义，`Qt.SizeMode.RelativeSize`（默认值）或相对于矩形的大小（作为 0...100 的值传递）。您可以传递 `Qt.SizeMode.RelativeSize` 以启用此功能。

drawEllipse

我们现在要介绍的最后一个原始绘制方法是 `drawEllipse`，它可以用于绘制椭圆或圆。

> 圆只是一个宽度和高度相等的椭圆。

Listing 141. bitmap/ellipse.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)
        pen = QPen()
        pen.setWidth(3)
        pen.setColor(QColor(204, 0, 0)) # r, g, b
        painter.setPen(pen)

        painter.drawEllipse(10, 10, 100, 100)
        painter.drawEllipse(10, 10, 150, 200)
        painter.drawEllipse(10, 10, 200, 300)
        painter.end()

        self.label.setPixmap(self.canvas)
```

在此示例中，`drawEllipse` 方法接受 4 个参数，其中前两个参数是矩形左上角的 x和 y 坐标，该矩形用于绘制椭圆，而后两个参数分别是该矩形的宽度和高度。

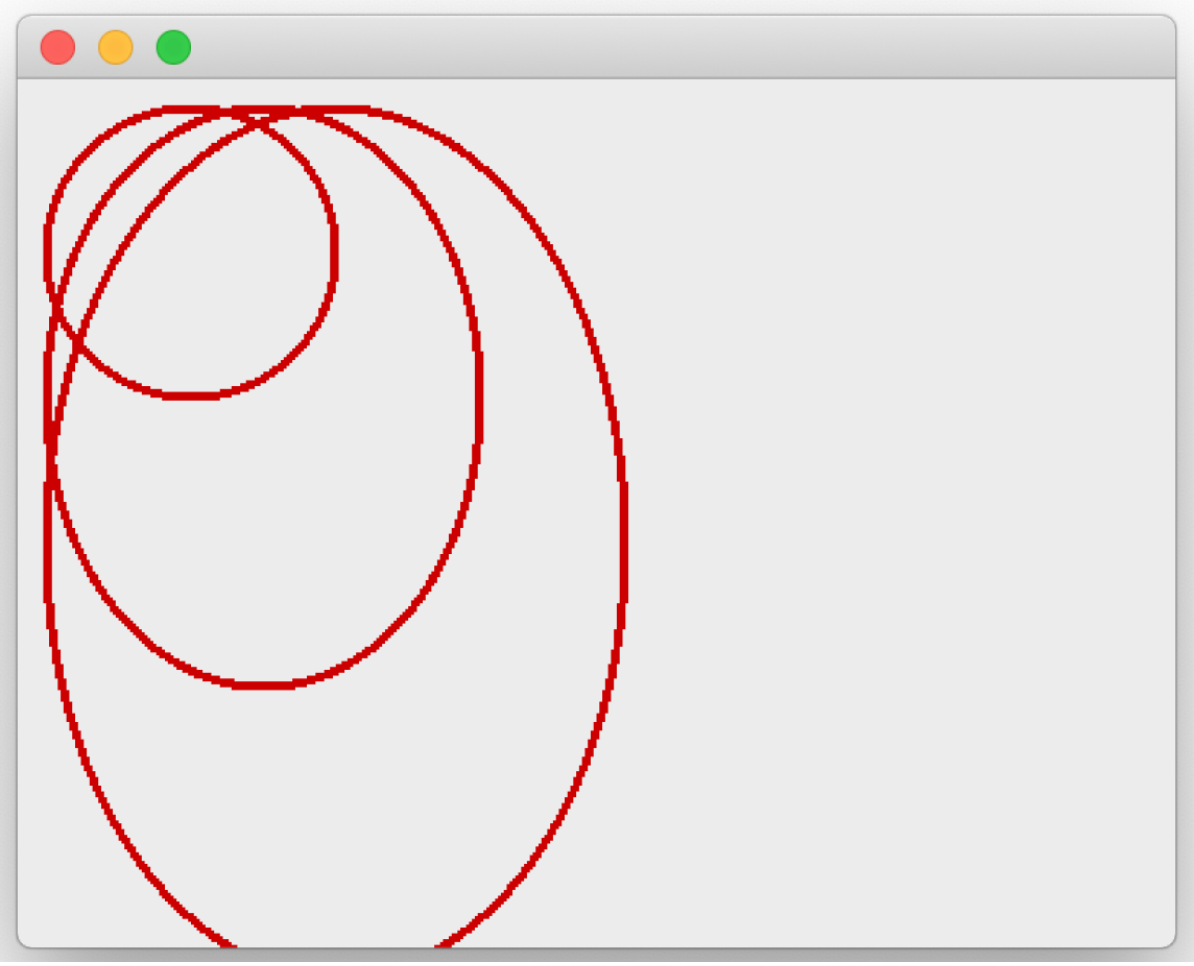


图171：使用x、y、宽度、高度或 `QRect` 绘制椭圆

您可以通过传递一个 `QRect` 来实现相同的效果。

还有另一种调用签名，它将椭圆的中心作为第一个参数，以 `QPoint` 或 `QPointF` 对象的形式提供，然后是 x 和 y 半径。下面的示例展示了它的使用方式

```
painter.drawEllipse(QtCore.QPoint(100, 100), 10, 10)
painter.drawEllipse(QtCore.QPoint(100, 100), 15, 20)
painter.drawEllipse(QtCore.QPoint(100, 100), 20, 30)
painter.drawEllipse(QtCore.QPoint(100, 100), 25, 40)
painter.drawEllipse(QtCore.QPoint(100, 100), 30, 50)
painter.drawEllipse(QtCore.QPoint(100, 100), 35, 60)
```

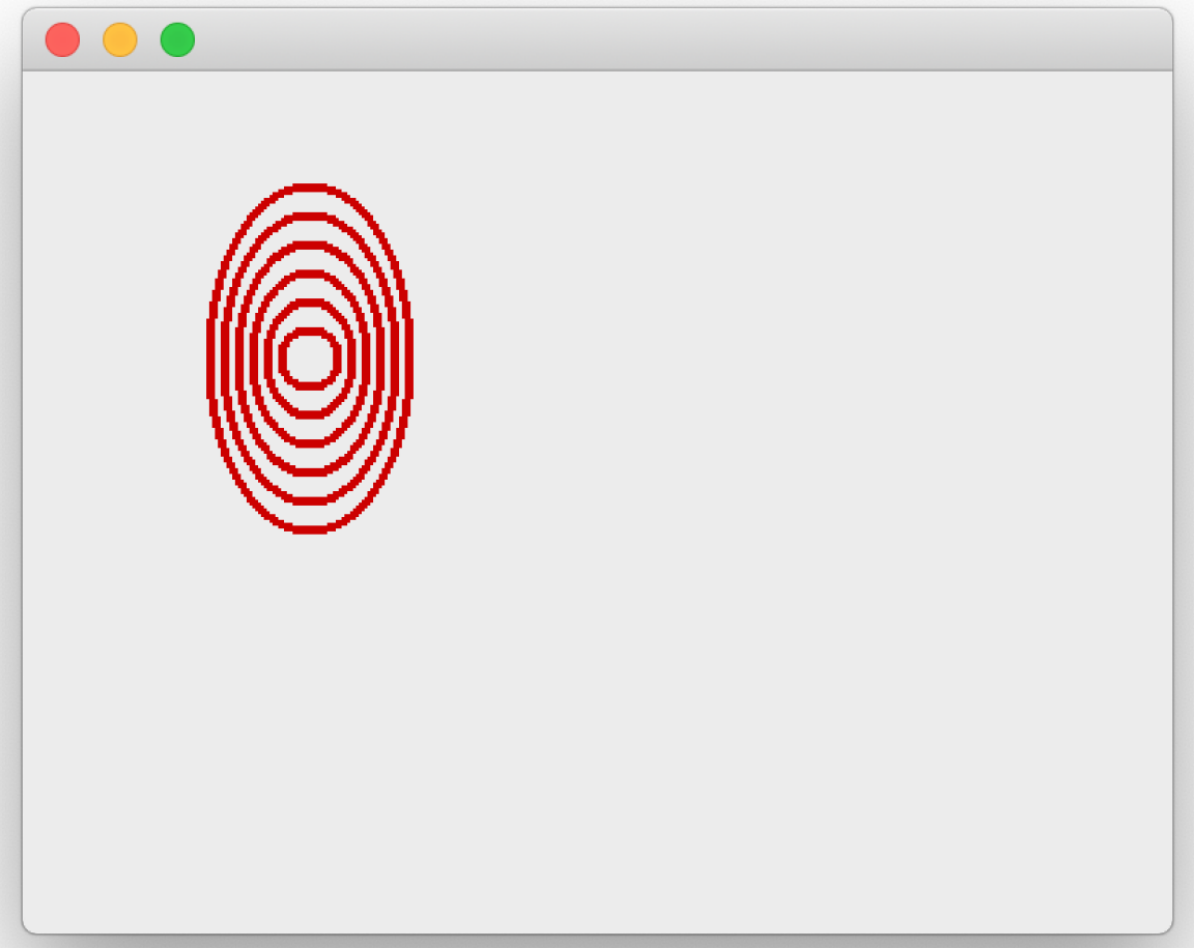


图172：使用点和半径绘制椭圆

您可以使用与矩形相同的 `QBrush` 方法来填充椭圆。

文本

最后，我们将简要介绍 `QPainter` 的文本绘制方法。要控制 `QPainter` 的当前字体，您可以使用 `setFont` 方法并传入一个 `QFont` 实例。通过此方法，您可以控制文本的字体家族、字重和字号（以及其他属性）。文本的颜色仍由当前画笔定义，但画笔的宽度不会产生影响。

Listing 142. bitmap/text.py

```
    def draw_something(self):
        painter = QPainter(self.canvas)

        pen = QPen()
        pen.setWidth(1)
        pen.setColor(QColor("green"))
        painter.setPen(pen)

        font = QFont()
        font.setFamily("Times")
        font.setBold(True)
        font.setPointSize(40)
        painter.setFont(font)

        painter.drawText(100, 100, "Hello, world!")
        painter.end()
        self.label.setPixmap(self.canvas)
```

> 您还可以使用 `QPoint` 或 `QPointF` 指定位置。

> 图173：位图文本“Hello World”示例

还有一些方法可以在指定区域内绘制文本。这里，参数定义了边界框的 x 和 y 位置以及宽度和高度。超出此框的文本将被裁剪（隐藏）。第 5 个参数标志可用于控制文本在框内的对齐方式等其他设置。

```
painter.drawText(100, 100, 100, 100, Qt.AlignmentFlag.AlignHCenter,
                 'Hello, world!')
```

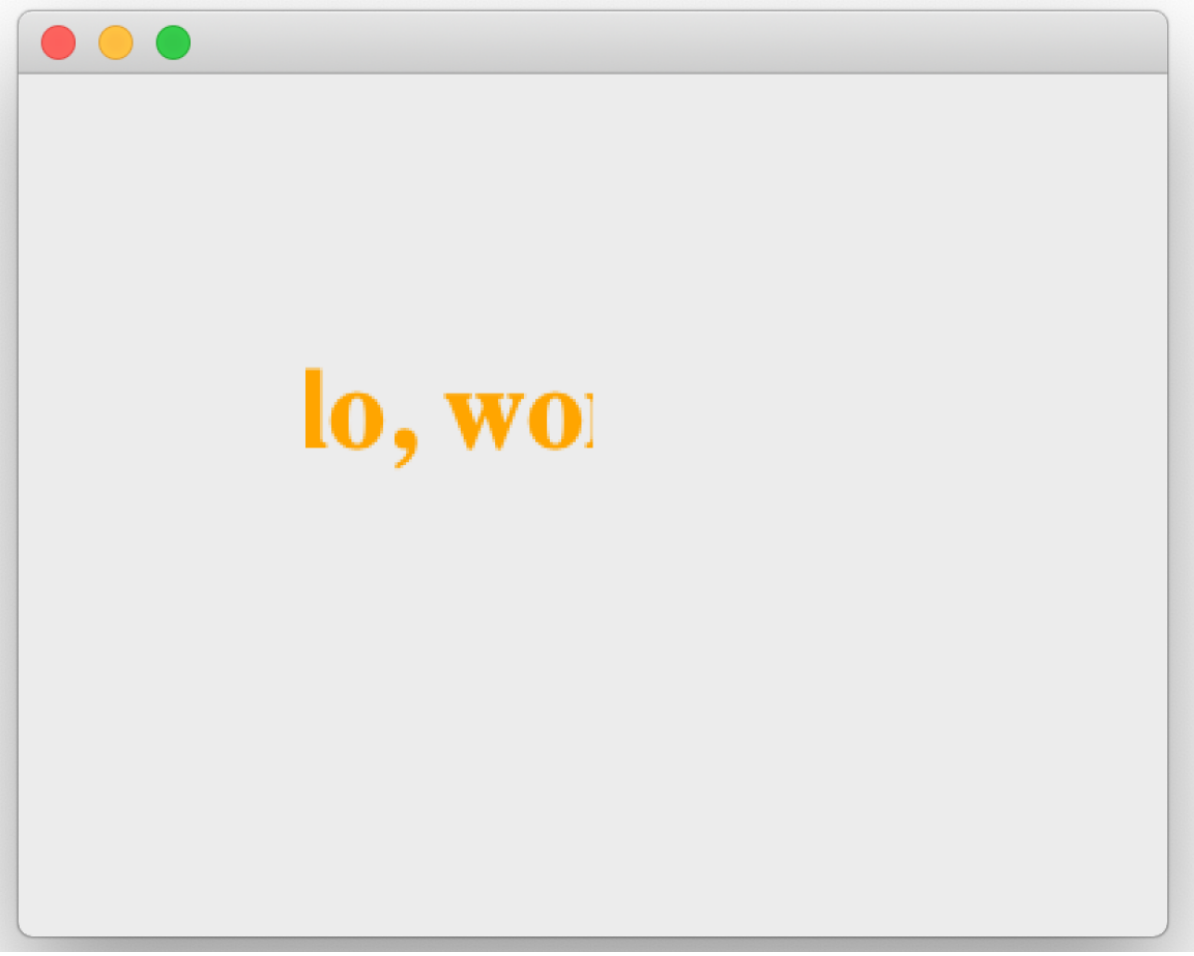


图174：在drawText中裁剪边界框

您可以通过在画家上设置活动字体来完全控制文本的显示。通过 `QFont` 对象设置活动字体。您还可以查看 QFont 文档 以获取更多信息。

用QPainter玩点小花样

这部分内容有点复杂，所以让我们稍作休息，做点有趣的事情。到目前为止，我们一直通过程序化方式定义在 `QPixmap` 表面上执行的绘制操作。但我们同样可以根据用户输入进行绘制——例如允许用户在画布上随意涂鸦。让我们利用迄今为止学到的知识，构建一个简单的绘图应用程序。

我们可以从相同的简单应用程序框架开始，在 `MainWindow` 类中替换 `draw` 方法，添加一个 `mouseMoveEvent` 处理程序。在这里，我们获取用户鼠标的当前位置，并将其绘制到画布上。

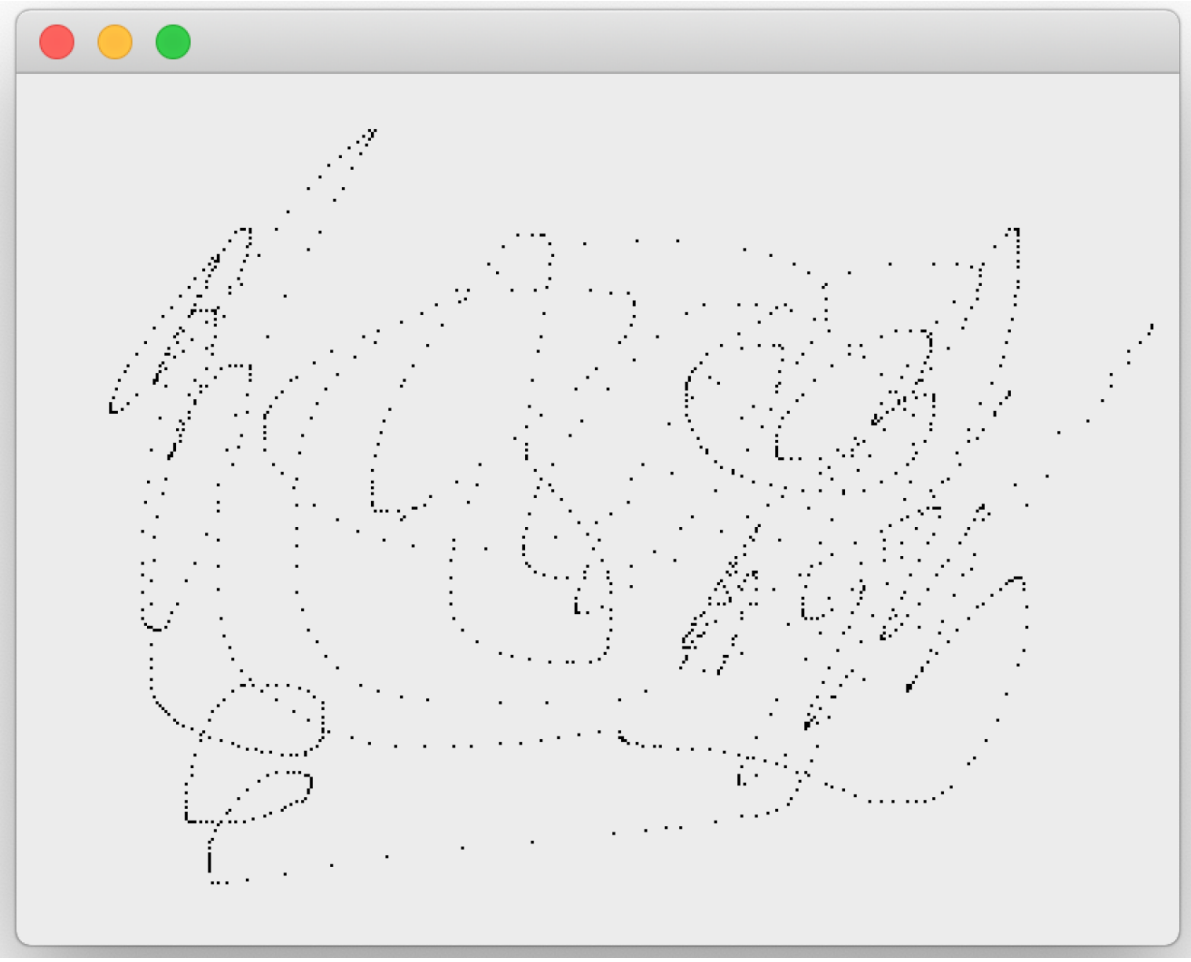
Listing 143. bitmap/paint_start.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QPainter, QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.label = QLabel()
        self.canvas = QPixmap(400, 300)
        self.canvas.fill(Qt.GlobalColor.white)
```

```
        self.label.setPixmap(self.canvas)
        self.setCentralWidget(self.label)

    def mouseMoveEvent(self, e):
        pos = e.position()
        painter = QPainter(self.canvas)
        painter.drawPoint(pos.x(), pos.y())
        painter.end()
        self.label.setPixmap(self.canvas)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 默认情况下，控件仅在按下鼠标按钮时接收鼠标移动事件，除非启用了鼠标跟踪。这可以通过 `.setMouseTracking` 方法进行配置——将该方法设置为 `True`（默认值为 `False`）将持续跟踪鼠标。

如果您保存并运行这个程序，您应该能够将鼠标移动到屏幕上并点击来绘制单个点。它应该看起来像这样——



> 图175：绘制单个鼠标移动事件点

这里的问题是，当您快速移动鼠标时，它实际上会在屏幕上的不同位置之间跳跃，而不是平滑地从一个位置移动到另一个位置。鼠标移动事件（`mouseMoveEvent`）会在鼠标所在的每个位置触发，但这不足以绘制一条连续的线，除非您移动得非常缓慢。

解决这个问题的办法是画线而不是点。在每个事件中，我们只需从我们所在的位置（之前的 `.x()` 和 `.y()`）到我们现在的位置（当前的 `.x()` 和 `.y()`）画一条线。我们可以自己跟踪 `last_x` 和 `last_y` 来做到这一点。

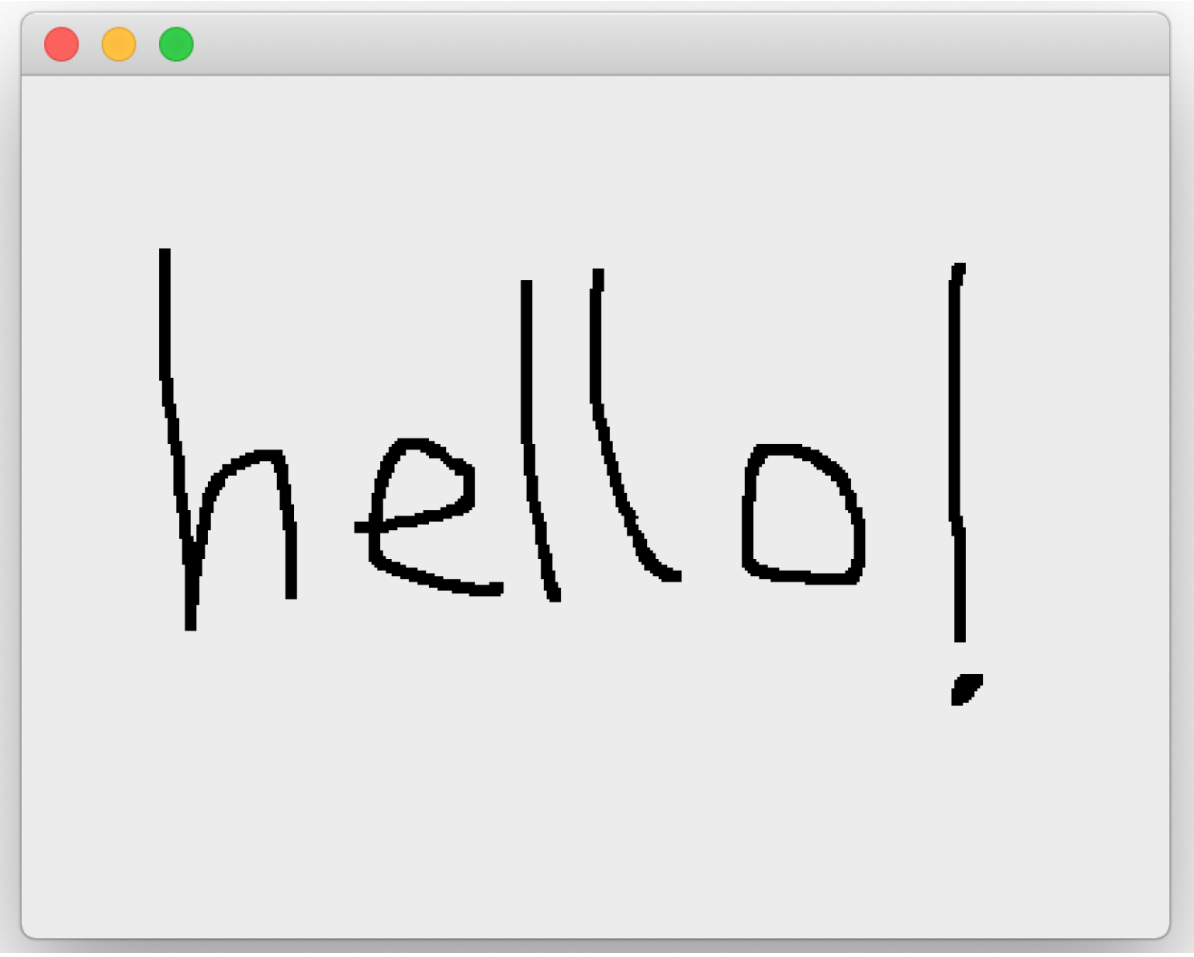
我们还需要在释放鼠标时忘记上一个位置，否则在将鼠标在页面上移动后，我们将从该位置开始再次绘制——即无法断开线条。

Listing 144. bitmap/paint_line.py

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QPainter, QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.label = QLabel()
        self.canvas = QPixmap(400, 300)
        self.canvas.fill(Qt.GlobalColor.white)
        self.label.setPixmap(self.canvas)
        self.setCentralWidget(self.label)

        self.last_x, self.last_y = None, None

    def mouseMoveEvent(self, e):
        pos = e.position()
        if self.last_x is None: # 第一个事件
            self.last_x = pos.x()
            self.last_y = pos.y()
            return # 在第一次时忽略它

        painter = QPainter(self.canvas)
        painter.drawLine(self.last_x, self.last_y, pos.x(), pos.y())
        painter.end()

        self.label.setPixmap(self.canvas)
        # 下次更新时更新源地址
        self.last_x = pos.x()
        self.last_y = pos.y()

    def mouseReleaseEvent(self, e):
        self.last_x = None
        self.last_y = None


app = QApplication(sys.argv)
```

```
window = MainWindow()
window.show()
app.exec()
```

如果您运行这个程序，您应该能够像预期那样在屏幕上涂鸦。



> 图176：使用鼠标进行绘图，采用连续线条。

目前效果还略显单调，因此我们添加一个简单的调色板，以便能够更改笔的颜色。

这需要进行一些重新设计，以确保鼠标位置能够被准确检测到。到目前为止，我们一直在 `QMainWindow` 上使用 `mouseMoveEvent`。当窗口中只有一个控件时，这没问题——只要您不调整窗口的大小，容器和单个嵌套控件的坐标就会对齐。但是，如果我们在布局中添加其他控件，情况就不同了——`QLabel` 的坐标会从窗口偏移，我们就会在错误的位置绘制。然而，如果我们在布局中添加其他控件，情况就不会如此了——`QLabel` 的坐标将与窗口偏移，我们将在错误的位置绘制。

这很容易解决，只需将鼠标处理移到 `QLabel` 本身即可——它的事件坐标总是相对于自身而言的。我们将它作为一个单独的 `Canvas` 对象进行包装，该对象负责创建像素图表面，设置 x 和 y 位置，并保存当前的笔颜色（默认设置为黑色）。

> 这个独立的 `Canvas` 是一个可直接使用的绘图表面，你可以在自己的应用程序中使用它。

Listing 145. bitmap/paint.py

```
import sys

from PyQt6.QtCore import QPoint, QSize, Qt
from PyQt6.QtGui import QColor, QPainter, QPen, QPixmap
from PyQt6.QtWidgets import (
QApplication,
QHBoxLayout,
QLabel,
QMainWindow,
QPushButton,
QVBoxLayout,
QWidget,
)

class Canvas(QLabel):
    def __init__(self):
        super().__init__()
        self._pixmap = QPixmap(600, 300)
        self._pixmap.fill(Qt.GlobalColor.white)
        self.setPixmap(self._pixmap)

        self.last_x, self.last_y = None, None
        self.pen_color = QColor("#000000")

    def set_pen_color(self, c):
        self.pen_color = QColor(c)

    def mouseMoveEvent(self, e):
        pos = e.position()
        if self.last_x is None: # 第一个事件
            self.last_x = pos.x()
            self.last_y = pos.y()
            return # 在第一次时将它忽略

        painter = QPainter(self._pixmap)
        p = painter.pen()
        p.setWidth(4)
        p.setColor(self.pen_color)
        painter.setPen(p)
        painter.drawLine(self.last_x, self.last_y, pos.x(), pos.y())
        painter.end()
        self.setPixmap(self._pixmap)

        # 下次更新时更新源地址
        self.last_x = pos.x()
        self.last_y = pos.y()

    def mouseReleaseEvent(self, e):
        self.last_x = None
        self.last_y = None
```

对于颜色选择，我们将基于 `QPushButton` 创建一个自定义控件。该控件接受一个颜色参数，该参数可以是 `QColor` 实例、颜色名称（“red”、black）或十六进制值。该颜色设置在控件的背景上，以便识别。我们可以使用标准的 `QPushButton.pressed` 信号将其连接到任何操作。

Listing 146. bitmap/paint.py

```
COLORS = [
    # 17种底色 https://lospec.com/palette-list/17undertones
    "#000000",
    "#141923",
    "#414168",
    "#3a7fa7",
    "#35e3e3",
    "#8fd970",
    "#5ebb49",
    "#458352",
    "#dcd37b",
    "#fffee5",
    "#ffd035",
    "#cc9245",
    "#a15c3e",
    "#a42f3b",
    "#f45b7a",
    "#c24998",
    "#81588d",
    "#bcb0c2",
    "#ffffff",
]
class QPaletteButton(QPushButton):
    def __init__(self, color):
        super().__init__()
        self.setFixedSize(QSize(24, 24))
        self.color = color
        self.setStyleSheet("background-color: %s;" % color)
```

定义了这两个新部分后，我们只需遍历颜色列表，为每个颜色创建一个 `QPaletteButton`，并传递颜色即可。然后将它的 `pressed` 信号连接到画布上的 `set_pen_color` 处理程序（通过 `lambda` 间接传递额外的颜色数据），并将其添加到调色板布局中。

Listing 147. bitmap/paint.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.canvas = Canvas()

        w = QWidget()
        l = QVBoxLayout()
        w.setLayout(l)
        l.addWidget(self.canvas)

        palette = QHBoxLayout()
        self.add_palette_buttons(palette)
        l.addLayout(palette)

        self.setCentralWidget(w)

    def add_palette_buttons(self, layout):
```

```
        for c in COLORS:
            b = QPaletteButton(c)
            b.pressed.connect(lambda c=c: self.canvas.set_pen_color(
                c))
            layout.addWidget(b)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

这将为您提供一个功能齐全的多色绘画应用程序，您可以在画布上画线并从颜色调色板中选择颜色。

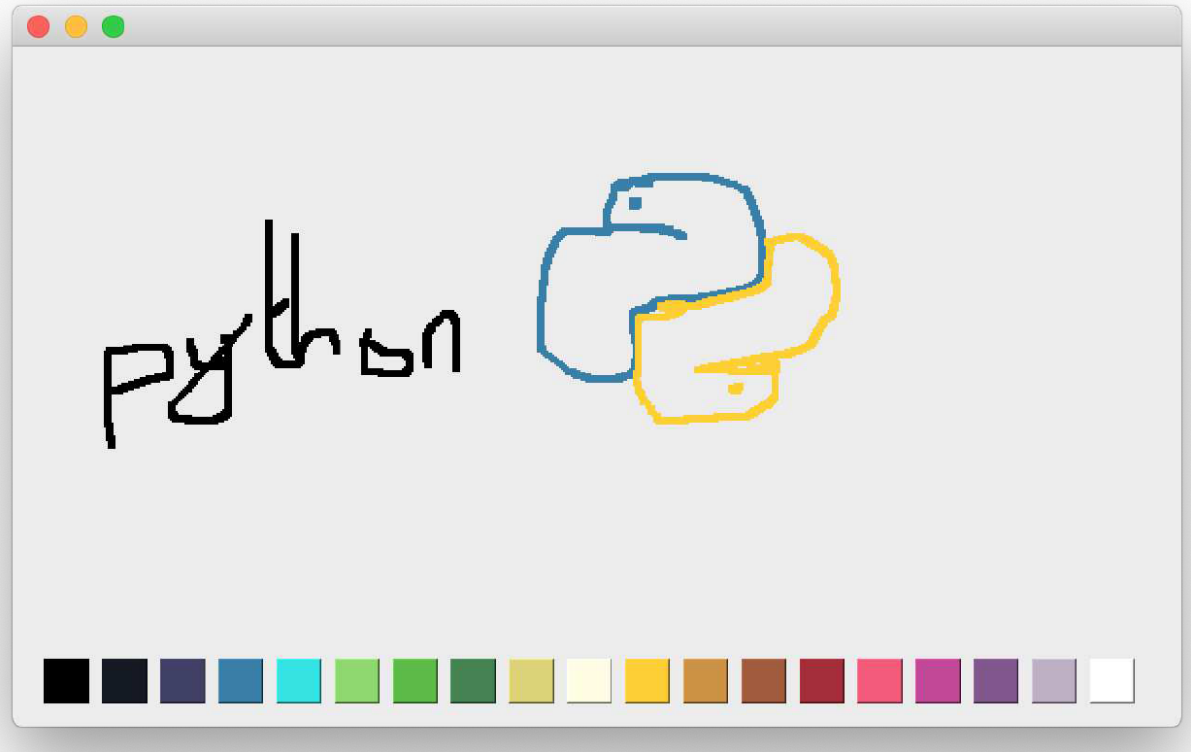


图177：遗憾的是，这并不能让您的画技变得优秀。

喷雾效果

为了增加一点趣味，您可以用以下代码替换 `mouseMoveEvent`，以使用"喷雾罐"效果代替直线绘制。这是通过使用 `random.gauss` 生成当前鼠标位置周围的一系列正态分布点来模拟的，然后我们使用 `drawPoint` 绘制这些点。

Listing 148. bitmap/spraypaint.py

```
import random
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtGui import QColor, QPainter, QPen, QPixmap
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLabel,
    QMainWindow,
    QPushButton,
```

```
    QVBoxLayout,
    QWidget,
)
SPRAY_PARTICLES = 100
SPRAY_DIAMETER = 10

class Canvas(QLabel):
    def __init__(self):
        super().__init__()
        self._pixmap = QPixmap(600, 300)
        self._pixmap.fill(Qt.GlobalColor.white)
        self.setPixmap(self._pixmap)

        self.pen_color = QColor("#000000")

    def set_pen_color(self, c):
        self.pen_color = QColor(c)

    def mouseMoveEvent(self, e):
        pos = e.position()
        painter = QPainter(self._pixmap)
        p = painter.pen()
        p.setWidth(1)
        p.setColor(self.pen_color)
        painter.setPen(p)

        for n in range(SPRAY_PARTICLES):
            xo = random.gauss(0, SPRAY_DIAMETER)
            yo = random.gauss(0, SPRAY_DIAMETER)
            painter.drawPoint(pos.x() + xo, pos.y() + yo)

        self.setPixmap(self._pixmap)
```

> 对于喷雾罐，我们无需追踪上一次的位置，因为我们总是围绕当前点进行喷涂。

我们在文件顶部定义 `SPRAY_PARTICLES` 和 `SPRAY_DIAMETER` 变量，并导入随机标准库模块。下图显示了使用以下设置时的喷雾行为：

```
import random

SPRAY_PARTICLES = 100
SPRAY_DIAMETER = 10
```

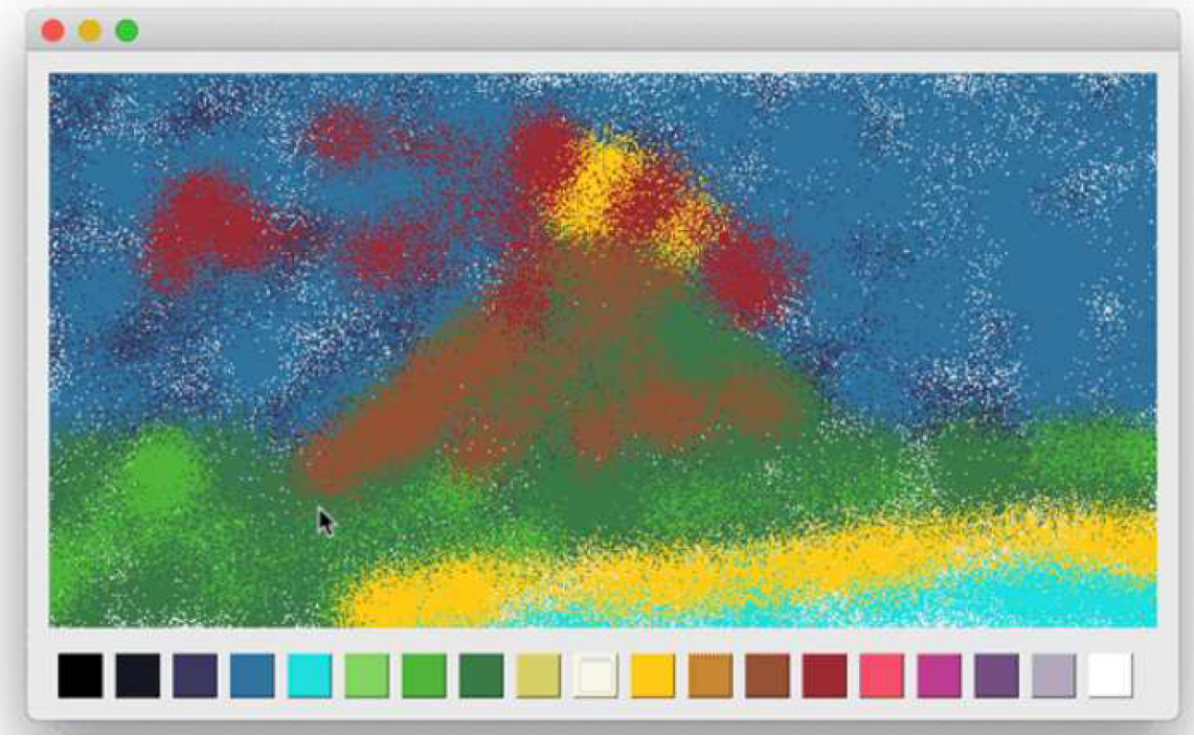


图178：请叫我毕加索

如果您想挑战一下自己，可以尝试添加一个额外的按钮来在绘制和喷涂模式之间切换，或者添加一个输入控件来定义笔刷/喷涂的直径。

> 要获取一个使用 Python 和 Qt 编写的完整功能绘图应用程序，请访问我们在 GitHub 上“Minute apps”仓库中的 Piecasso。

通过本介绍，您应该已经对 `QPainter` 的功能有了一个大致的了解。如上所述，该系统是所有控件绘制的基础。如果您想进一步了解，可以查看控件的 `.paint()` 方法，该方法接收一个 `QPainter` 实例，以允许控件在自身上进行绘制。您在这里学到的相同方法可以在 `.paint()` 中使用，以绘制一些基本的自定义控件。

22. 创建自定义控件

在上一章中，我们介绍了 `QPainter`，并学习了一些基本的位图绘制操作，您可以使用这些操作在 `QPainter` 表面（例如 `QPixmap`）上绘制点、线、矩形和圆。使用 `QPainter` 在表面上绘制图形的过程实际上是 Qt 中所有控件绘制的基础。现在您已经了解如何使用 `QPainter`，您知道如何绘制自己的自定义控件了！在本章中，我们将运用迄今为止所学到的知识，构建一个全新的自定义控件。在本章中，我们将运用迄今为止所学到的知识，构建一个全新的自定义控件——一个可自定义的带刻度盘控制的 `PowerBar` 仪表。

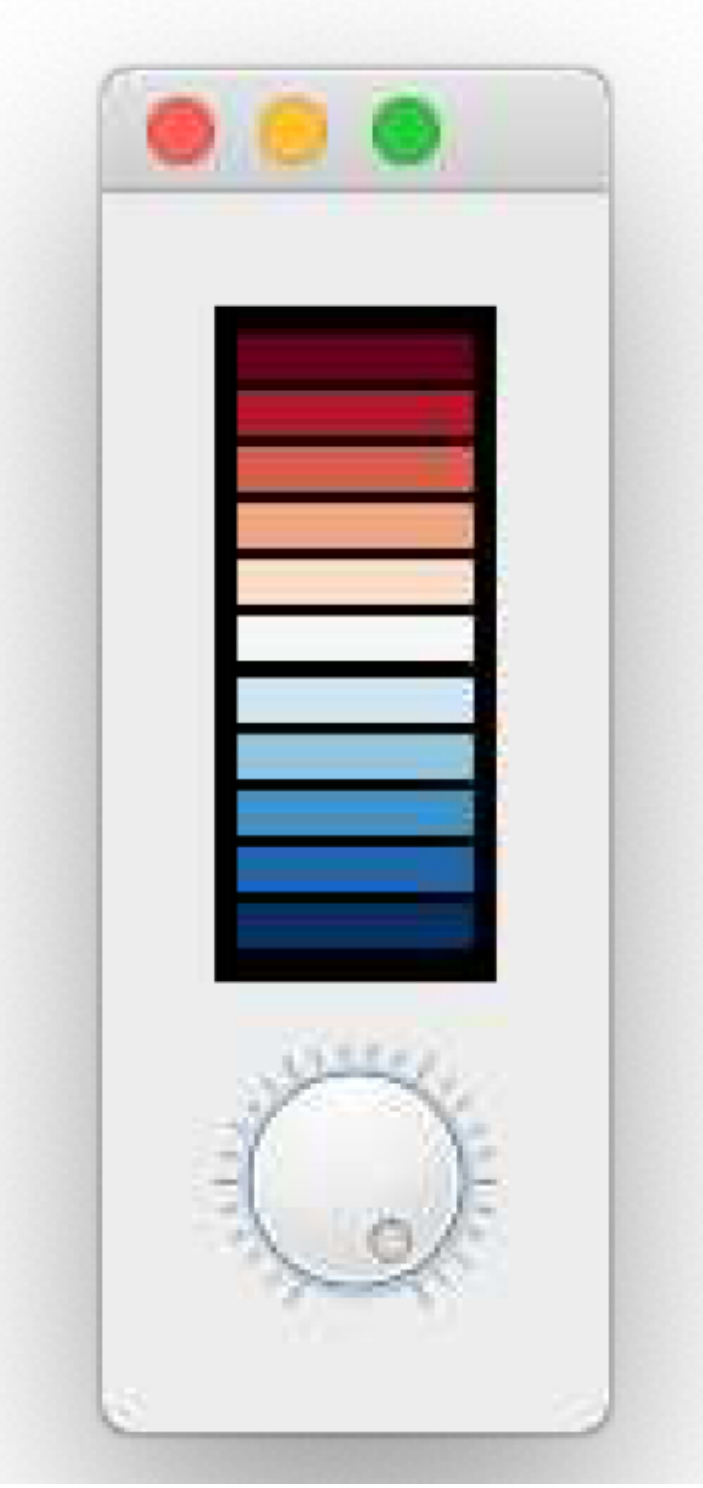


图179： `PowerBar` 仪表

该控件实际上是一个复合控件和自定义控件的混合体，因为我们使用内置的 Qt `QDial` 组件来绘制拨盘，而电源条则由我们自己绘制。然后，我们将这两个部分组合到一个父控件中，该控件可以无缝地放入任何应用程序中，而无需了解其组合方式。最终的控件提供了常见的 `QAbstractSlider` 接口，并添加了一些用于配置条形显示的功能。

按照这个示例，您将能够构建自己的自定义控件——无论它们是内置控件的组合，还是完全新颖的自绘制控件。

开始

正如我们之前所看到的，复合控件只是应用了布局的控件，其本身包含 >1 个其他控件。最终的"控件"可以像其他控件一样使用，内部结构可以根据需要隐藏或显示。

以下是我们 PowerBar 控件的轮廓——我们将从这个轮廓草稿开始逐步构建我们的自定义控件。

Listing 149. custom-widgets/stub.py

```
import sys

from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt


class _Bar(QtWidgets.QWidget):
    pass

class PowerBar(QtWidgets.QWidget):
    """
    用于显示电源条和拨号盘的自定义 Qt 控件。
    演示复合和自定义绘制的控件。
    """

    def __init__(self, parent=None, steps=5):
        super().__init__(parent)

        layout = QtWidgets.QVBoxLayout()
        self._bar = _Bar()
        layout.addWidget(self._bar)

        self._dial = QtWidgets.QDial()
        layout.addWidget(self._dial)

        self.setLayout(layout)


app = QtWidgets.QApplication(sys.argv)
volume = PowerBar()
volume.show()
app.exec()
```

这只是定义了我们的自定义电源条在 `_Bar` 对象中定义——这里只是未更改的 `QWidget` 子类。`PowerBar` 控件（完整的控件）结合了这一点，使用 `QVBoxLayout` 与内置的 `QDial` 一起显示它们。

> 我们无需创建 `QMainWindow`，因为任何没有父控件的控件都是一个独立的窗口。我们的自定义 `PowerBar` 控件将显示为一个普通的窗口。

您可以随时运行此文件来查看控件的运行情况。现在运行它，您应该会看到如下内容：

> 图180：PowerBar 旋钮。

如果将窗口向下拉伸，您会发现拨号盘上方比下方有更多空间——这是由我们的（目前不可见的）`_Bar` 控件占用的。

`paintEvent`

`paintEvent` 处理程序是 PyQt6 中所有控件绘制的核心。控件的每次完整和部分重绘都是通过 `paintEvent` 触发的，该事件由控件滑块处理以绘制自身。`paintEvent` 可以由以下因素触发：

- 调用了 repaint() 或 update()
- 控件被遮挡，现在已露出
- 控件已调整大小

——但还有许多其他原因也会导致这种情况。重要的是，当 `paintEvent` 被触发时，您的控件能够重新绘制它。

如果控件足够简单（就像我们的控件一样），通常只需在发生任何变化时重新绘制整个控件即可。但对于更复杂的控件，这种方法效率非常低。对于这些情况，`paintEvent` 包括需要更新的特定区域。我们将在后面更复杂的示例中使用此方法。

现在，我们将做一件非常简单的事情，用一种颜色填充整个控件。这样，我们就可以看到要绘制条形图的区域了。请您将以下代码添加到 `_Bar` 类中。

Listing 150. custom-widgets/powerbar_1.py

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)
        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor("black"))
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)
```

定位

现在，我们可以看到 `_Bar` 控件，可以调整其位置和大小。如果拖动窗口形状，您应该会看到两个控件改变形状以适应可用空间。这是我们想要的效果，但 `QDial` 也垂直扩展得比应该的更多，留出了我们可以用来放置条形图的空白空间。

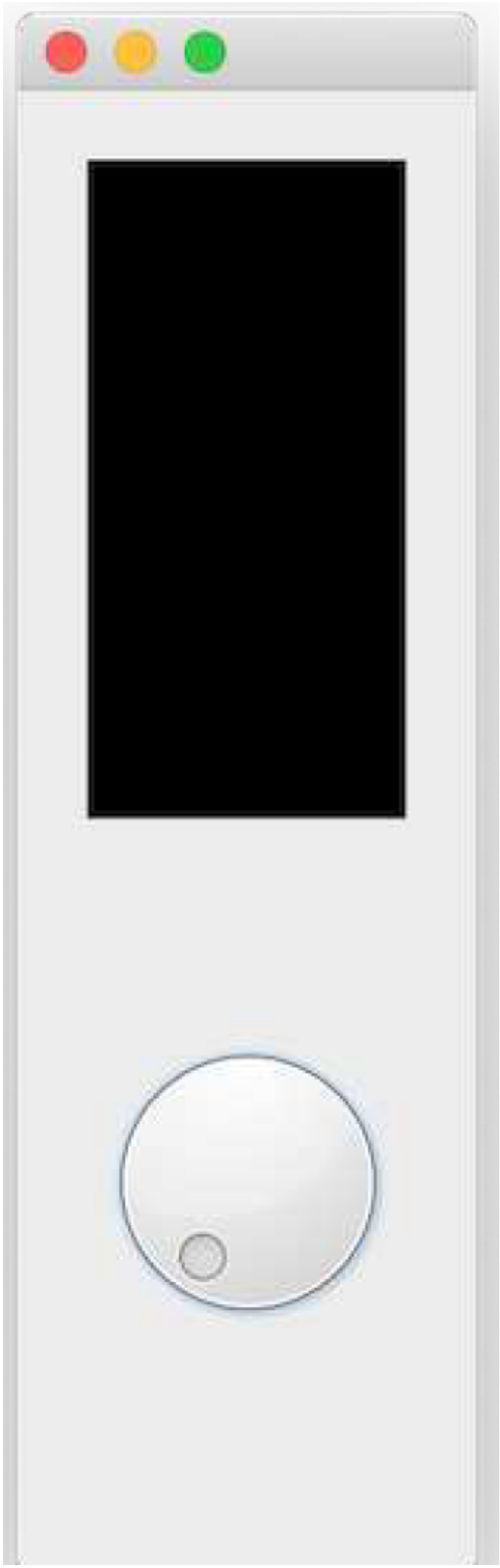


图181：PowerBar 伸展并留出了空间

我们可以对 `_Bar` 控件使用 `setSizePolicy`，以确保它尽可能地扩展。通过使用 `QSizePolicy.MinimumExpanding`，提供的 `sizeHint` 将被用作最小值，控件将尽可能地扩展。

Listing 151. custom-widgets/powerbar_2.py

```
class _Bar(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.setSizePolicy(
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
        )

    def sizeHint(self):
        return QtCore.QSize(40, 120)

    def paintEvent(self, e):
        painter = QtGui.QPainter(self)
        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor("black"))
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)
```

虽然 `QDial` 控件的大小调整还有些不理想，但我们的条形图现在可以扩展到填满所有可用空间了。

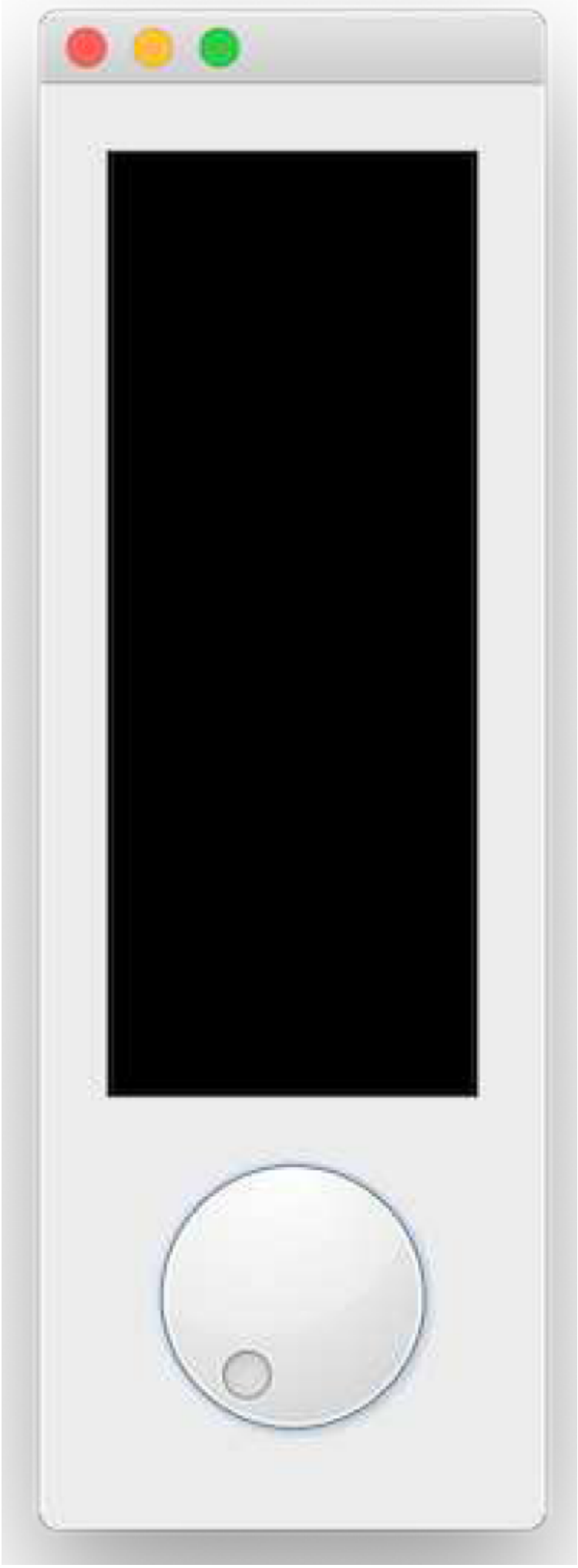


图182：PowerBar 的政策设置为 QSizePolicy.MinimumExpanding。

定位完成后，我们现在可以继续定义绘制方法，在控件顶部（目前为黑色）绘制 PowerBar 计量表。

更新显示

现在，我们的画布已经完全填充为黑色，接下来，我们将使用 `QPainter` 绘制命令在控件上实际绘制一些内容。

在开始绘制条形图之前，我们需要进行一些测试，以确保我们可以使用刻度盘的值更新显示。请将以下代码添加到 `_Bar.paintEvent` 中：

Listing 152. custom-widgets/powerbar_3.py

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor("black"))
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)

        # 获取当前状态。
        dial = self.parent()._dial
        vmin, vmax = dial.minimum(), dial.maximum()
        value = dial.value()

        pen = painter.pen()
        pen.setColor(QtGui.QColor("red"))
        painter.setPen(pen)

        font = painter.font()
        font.setFamily("Times")
        font.setPointSize(18)
        painter.setFont(font)

        painter.drawText(
            25, 25, "{}-->{}<--{}".format(vmin, value, vmax)
        )
        painter.end()
```

这会像之前一样绘制黑色背景，然后使用 `.parent()` 访问我们的父级 `PowerBar` 控件，并通过 `_dial` 访问 `QDial`。在那里我们可以获得当前值以及允许的最小值和最大值。最后，我们使用绘图器绘制这些值，就像之前部分一样。

> 我们在这里将当前值、最小值和最大值的处理交给 `QDial`，但我们也可以自己存储这些值，并使用来自/发往拨盘的信号来保持同步。

运行这个程序，转动旋钮，然后……什么也没发生。尽管我们已经定义了 `paintEvent` 处理程序，但当旋钮发生变化时，我们并没有触发重新绘制。

> 您可以通过调整窗口大小强制刷新，一旦您这样做，您应该会看到文本出现。很巧妙，但用户体验很糟糕——“只需调整应用程序大小即可查看设置！”

要解决这个问题，我们需要将 `_Bar` 控件与 `dial` 上的值变化挂钩，以响应地重新绘制自身。我们可以使用 `QDial.valueChanged` 信号来做到这一点，将它与一个自定义槽方法挂钩，该方法调用 `.refresh()`，从而触发完全重绘。

请您将以下方法添加到 `_Bar` 控件中

Listing 153. custom-widgets/powerbar_4.py

```
    def _trigger_refresh(self):
        self.update()
```

...并将以下内容添加到父级 `PowerBar` 控件的 `__init__` 块中。

Listing 154. custom-widgets/powerbar_4.py

```
        self._dial = QtWidgets.QDial()
        self._dial.valueChanged.connect(self._bar._trigger_refresh)
        layout.addWidget(self._dial)
```

如果您现在重新运行代码，您应该会看到显示屏会自动更新，当您转动旋钮时（用鼠标点击并拖动）当前值将以文本形式显示。

> 图183：PowerBar 以文本形式显示当前值

绘制条

现在我们已经实现了显示屏的更新并显示指针的当前值，我们可以继续绘制实际的条形显示。这部分稍微复杂一些，需要一些数学计算来确定条形的位置，但我们会一步步讲解，以便大家明白其中的原理。

下图显示了我们的目标——一系列 N 个框，从控件边缘内嵌，之间留有空格。

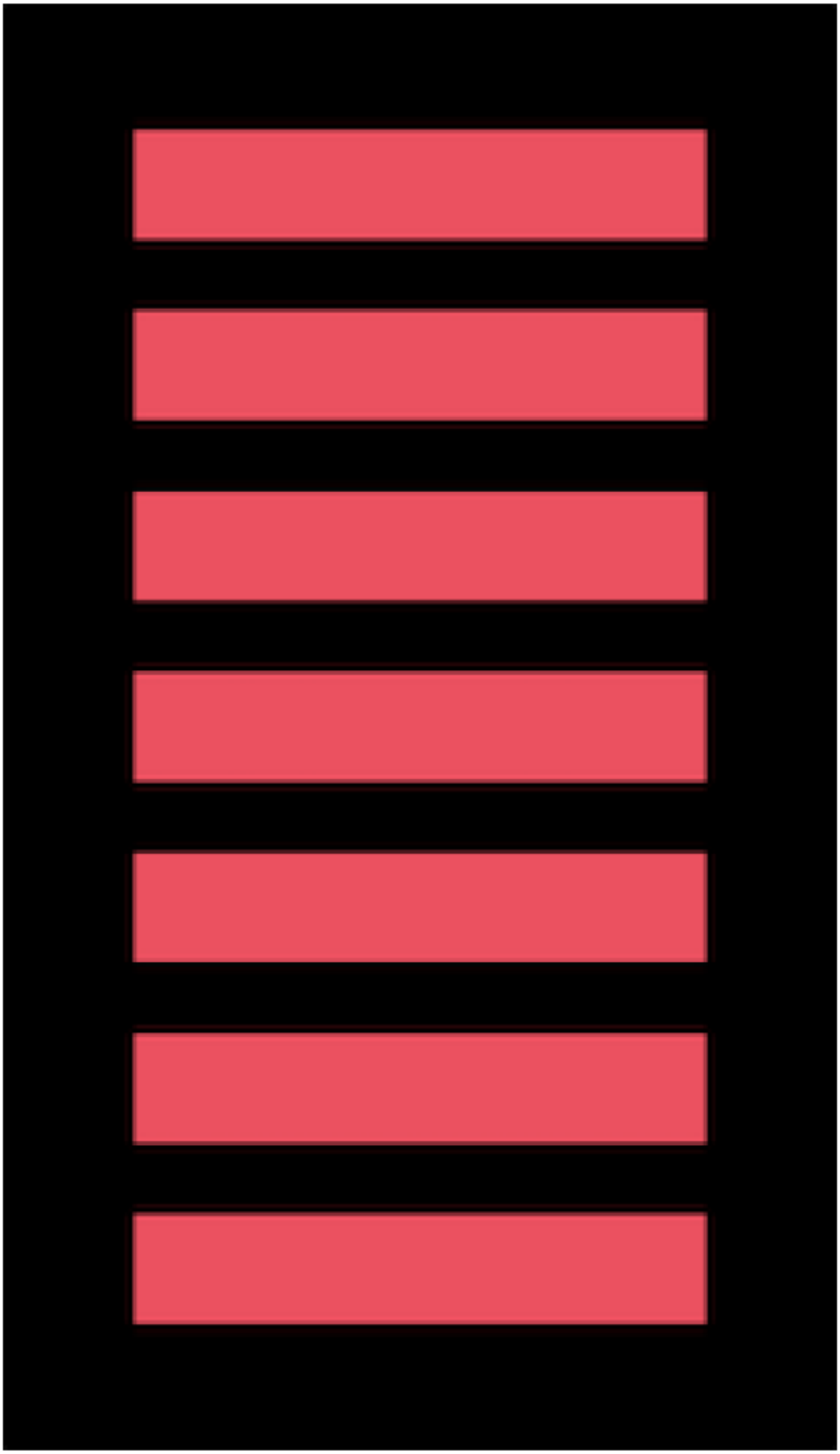


图184：我们正在努力实现的条形图分段及布局方案

计算要绘制的内容

要绘制的框的数量由当前值决定——以及它在 `QDial` 配置的最小值和最大值之间的位置。我们在上面的示例中已经有了这些信息。

```
dial = self.parent()._dial
vmin, vmax = dial.minimum(), dial.maximum()
value = dial.value()
```

如果值位于 `vmin` 和 `vmax` 的中间，则我们希望绘制一半的框。（如果总共有4个框，则绘制2个）。如果值为 `vmax`，则我们希望绘制所有框。

要实现这一点，我们首先将 `value` 转换为0到1之间的数值，其中 `0 = vmin`，`1 = vmax`。我们首先从值中减去 `vmin`，以将可能 `value` 的范围调整为从零开始——即从 `vmin…vmax` 调整为 `0…(vmax-vmin)`。将此值除以 `vmax-vmin`（新的最大值）即可得到一个介于0和1之间的数值。

然后，将这个值（下文称为 `pc`）乘以步长，即可得到一个介于 0 和 5 之间的数值——即需要绘制的方块数量。

```
pc = (value - vmin) / (vmax - vmin)
n_steps_to_draw = int(pc * 5)
```

我们将结果包装为整数以将其转换为整数（向下取整）以去除任何部分的框。

请您在您的绘制事件中更新 `drawText` 方法以写出这个数字。

Listing 155. custom-widgets/powerbar_5.py

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor("black"))
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)

        # 获取当前状态。
        dial = self.parent()._dial
        vmin, vmax = dial.minimum(), dial.maximum()
        value = dial.value()

        pen = painter.pen()
        pen.setColor(QtGui.QColor("red"))
        painter.setPen(pen)

        font = painter.font()
        font.setFamily("Times")
        font.setPointSize(18)
```

```
        painter.setFont(font)

        pc = (value - vmin) / (vmax - vmin)
        n_steps_to_draw = int(pc * 5)
        painter.drawText(25, 25, "{}".format(n_steps_to_draw))
        painter.end()
```

当您转动旋钮时，现在您将看到一个介于0和5之间的数字。

绘制框体

接下来，我们希望将这个数字 0...5 转换为画布上绘制的条形数量。首先，删除 `drawText`、字体和画笔设置，因为我们不再需要这些。

为了准确绘制，我们需要知道画布的大小，即控件的大小。我们还将在边缘添加一些填充，以便在黑色背景上为块的边缘留出空间。

> `QPainter` 中的所有测量单位均为像素。

Listing 156. custom-widgets/powerbar_6.py

```
        padding = 5
        # 定义我们的画布.
        d_height = painter.device().height() - (padding * 2)
        d_width = painter.device().width() - (padding * 2)
```

我们获取高度和宽度，然后从每个值中减去 `2 * padding` ——这里是 2 倍，因为我们同时对左右（以及上下）边缘进行填充。这样就得到了最终的有效画布区域，分别存储在 `d_height` 和 `d_width` 中。

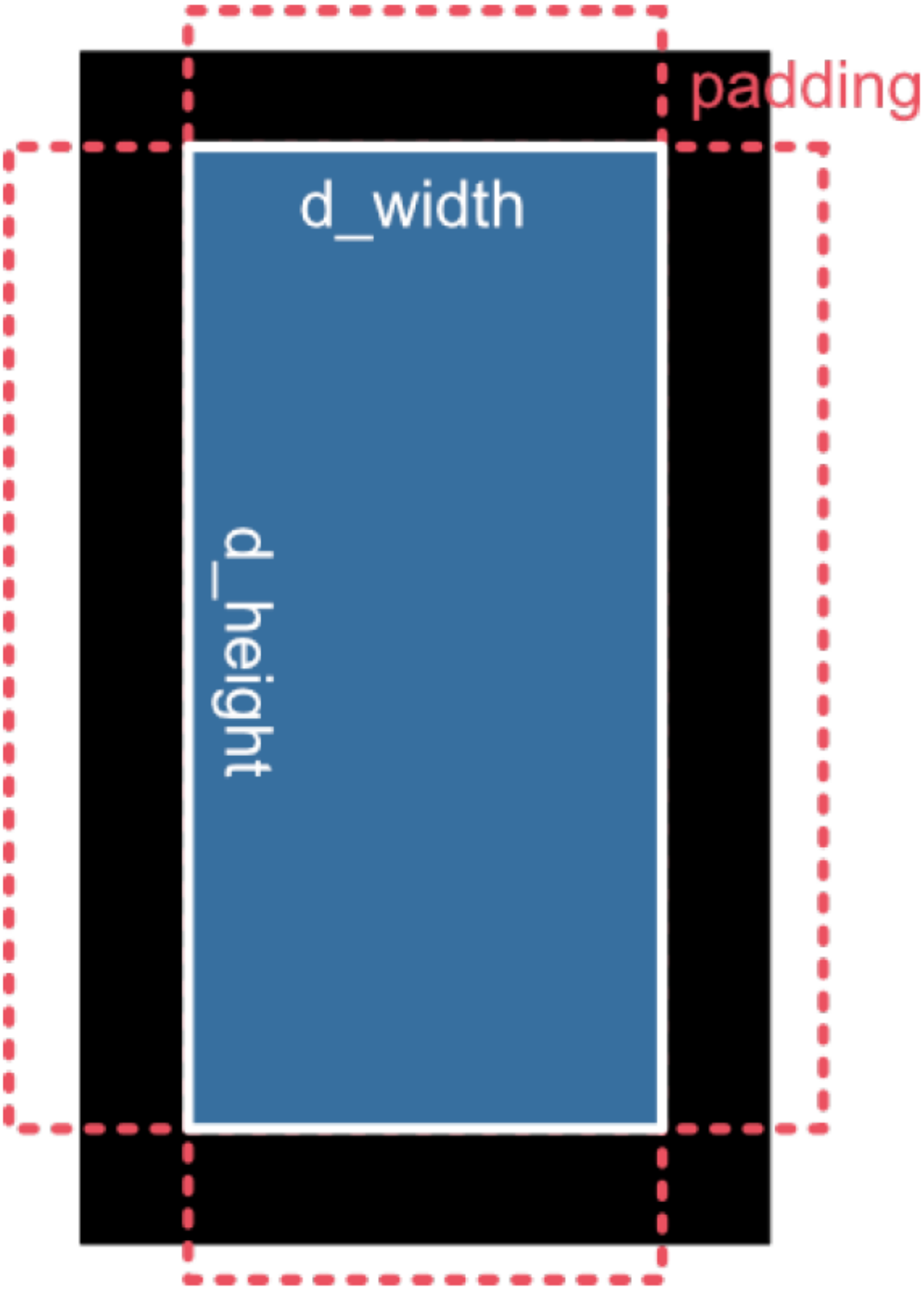


图185：布局外部的填充部分

我们需要将 `d_height` 分成5等份，每份对应一个块——我们可以通过 `d_height / 5` 来计算这个高度。此外，由于我们希望在块之间留有间隙，我们需要计算这个步长中有多少被间隙（顶部和底部，因此减半）占用，有多少是实际的块。

Listing 157. custom-widgets/powerbar_6.py

```
        step_size = d_height / 5
        bar_height = step_size * 0.6
```

这些值是我们绘制画布上方块所需的全部信息。为此，我们使用 `range` 函数从 0 开始计数，直到步数减 1，然后为每个方块绘制一个 `fillRect` 区域。

Listing 158. custom-widgets/powerbar_6.py

```
        brush.setColor(QtGui.QColor("red"))

        for n in range(n_steps_to_draw):
            ypos = (1 + n) * step_size
            rect = QtCore.QRect(
                padding,
                padding + d_height - int(ypos),
                d_width,
                int(bar_height),
            )
            painter.fillRect(rect, brush)
```

在块的放置计算中涉及大量内容，因此我们先逐一分析这些步骤。

用于绘制填充矩形的 `fillRect` 被定义为一个 `QRect` 对象，我们依次向其传入左侧 x 坐标、顶部 y 坐标、宽度和高度。

宽度是整个画布宽度减去填充，我们之前已经计算并存储在 `d_width` 中。左侧 x 同样只是控件左侧的 `padding` 的值（5px）。

我们计算的栏杆高度 `bar_height` 为 `step_size` 的 0.6 倍。

这使得参数 `d_height - ((1 + n) * step_size)` 确定要绘制的矩形的顶部 y 位置。这是在绘制方块时唯一会变化的计算。

> 请记住，`QPainter` 中的 y 坐标从顶部开始，并向下增加。这意味着在 `d_height` 处绘制将对应于画布的最底部。

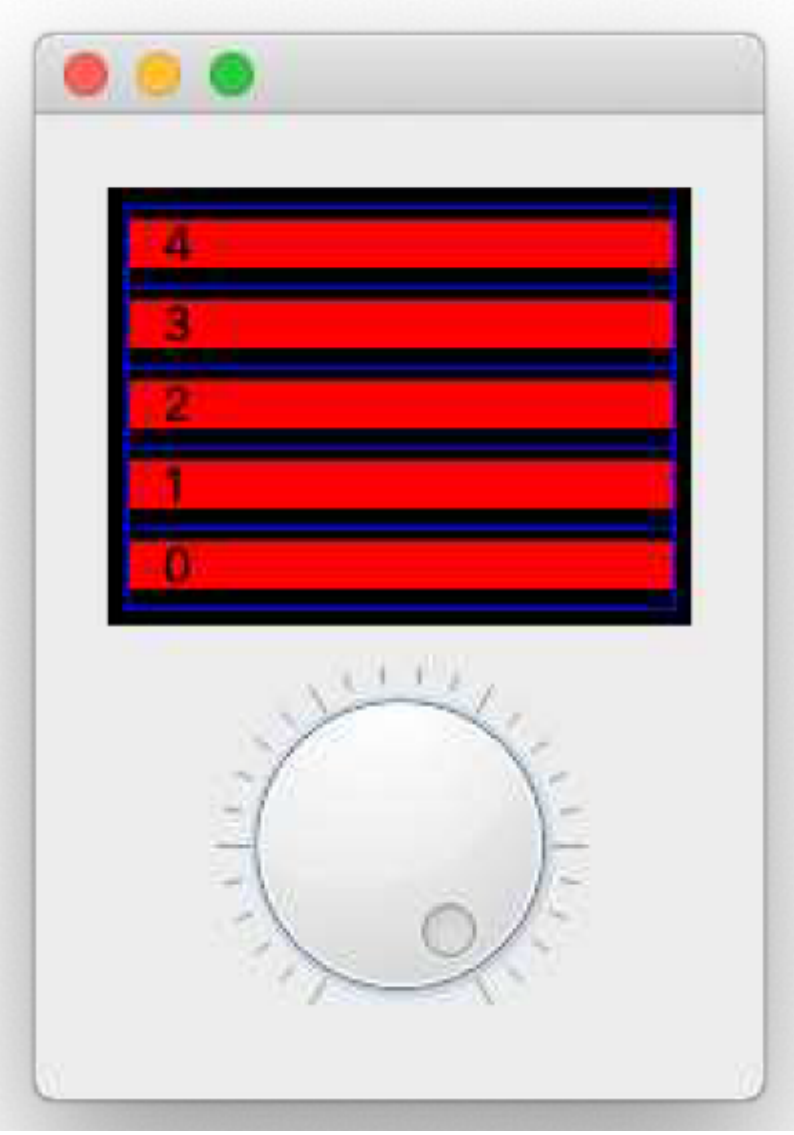
> 要在最底部绘制一个块，我们必须从 `d_height-step_size` 开始绘制，即向上绘制一个块以留出空间向下绘制。

在我们的条计量器中，我们依次绘制方块，从底部开始并向上绘制。因此，我们的第一个方块必须放置在 `d_height-step_size` 处，而第二个方块放置在 `d_height-(step_size*2)` 处。我们的循环从 0 开始向上迭代，因此我们可以使用以下公式实现这一点：

```
ypos = (1 + n) * step_size
y = d_height - ypos
```

这将生成以下布局。

在下图中，当前的n值已被打印在框上，并且一个蓝色的框已被绘制在完整的 `step_size` 周围，这样您可以看到填充和间隔符的实际效果。



> 图186：显示每个段落所占用的整个区域（蓝色部分）

将以上内容整合在一起，就会得到以下代码，运行后将生成一个带红色块的工作电源条控件。您可以来回拖动滚轮，条形将随之上下移动。

Listing 159. custom-widgets/powerbar_6b.py

```
import sys
```

```python
from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt


class _Bar(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()

        self.setSizePolicy(
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
        )

    def sizeHint(self):
        return QtCore.QSize(40, 120)

    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(QtGui.QColor("black"))
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)

        # 获取当前状态.
        dial = self.parent()._dial
        vmin, vmax = dial.minimum(), dial.maximum()
        value = dial.value()

        pc = (value - vmin) / (vmax - vmin)
        n_steps_to_draw = int(pc * 5)

        padding = 5

        # 定义我们的画布.
        d_height = painter.device().height() - (padding * 2)
        d_width = painter.device().width() - (padding * 2)

        step_size = d_height / 5
        bar_height = step_size * 0.6

        brush.setColor(QtGui.QColor("red"))

        for n in range(n_steps_to_draw):
            ypos = (1 + n) * step_size
            rect = QtCore.QRect(
                padding,
                padding + d_height - int(ypos),
                d_width,
                int(bar_height),
```

```
            )
            painter.fillRect(rect, brush)
            painter.end()

    def _trigger_refresh(self):
        self.update()
class PowerBar(QtWidgets.QWidget):
    """
    自定义 Qt 控件，用于显示电源条和拨盘。
    演示复合和自定义绘制的控件.
    """

    def __init__(self, parent=None, steps=5):
        super().__init__(parent)

        layout = QtWidgets.QVBoxLayout()
        self._bar = _Bar()
        layout.addWidget(self._bar)

        self._dial = QtWidgets.QDial()
        self._dial.valueChanged.connect(self._bar._trigger_refresh)
        layout.addWidget(self._dial)

        self.setLayout(layout)


app = QtWidgets.QApplication(sys.argv)
volume = PowerBar()
volume.show()
app.exec()
```

图187：基础版全功能PowerBar

这已经可以完成工作了，但我们可以更进一步，提供更多自定义选项，添加一些用户体验改进，并改进与我们的控件配合使用的 API。

自定义条

现在，我们已经拥有了一个可通过旋钮控制的电源条。但在创建控件时，最好提供一些选项来配置控件的行为，以使其更加灵活。在本部分中，我们将添加一些方法来设置可自定义的分段数、颜色、填充和间距。首先，我们需要创建一个可自定义的电源条。

我们将提供定制化的元素包括——

选项	描述
条数	控件上显示多少个条?
颜色	每个条的颜色各不相同
背景色	绘图画布的颜色（默认黑色）

选项	描述
填充	控件边缘周围的空间，在条和画布边缘之间
条高度 / 条百分比	条中实心部分的比例（0...1）（其余部分为相邻条之间的间隔）

我们可以将这些属性存储在 `_bar` 对象上，并在 `paintEvent` 方法中使用它们来更改其行为。

`_Bar.__init__` 已更新，可接受初始参数，用于指定条形图的条数（作为整数）或条形图的颜色（作为 `QColor` 列表、十六进制值或名称）。如果提供数字，所有条形图将被涂成红色。如果提供颜色列表，条形图的数量将根据颜色列表的长度确定。默认值为 `self._bar_solid_colors`。如果提供数字，所有条形将被设置为红色。如果提供颜色列表，条形数量将根据颜色列表的长度确定。
`self._bar_solid_percent`、`self._background_color`、`self._padding` 的默认值也已设置。

Listing 160. custom-widgets/powerbar_7.py

```
class _Bar(QtWidgets.QWidget):
    def __init__(self, steps):
        super().__init__()
        self.setSizePolicy(
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
        )

        if isinstance(steps, list):
            # 颜色列表。
            self.n_steps = len(steps)
            self.steps = steps

        elif isinstance(steps, int):
            # 条数，默认颜色为红色。
            self.n_steps = steps
            self.steps = ["red"] * steps

        else:
            raise TypeError("steps must be a list or int")

        self._bar_solid_percent = 0.8
        self._background_color = QtGui.QColor("black")
        self._padding = 4 # 边缘周围有n像素的间隙。
```

同样，我们更新 `PowerBar.__init__` 以接受 `steps` 参数，并将其传递下去。

Listing 161. custom-widgets/powerbar_7.py

```
class PowerBar(QtWidgets.QWidget):
    """
    自定义 Qt 控件，用于显示电源条和拨盘。
    演示复合和自定义绘制的控件.
    """
    def __init__(self, parent=None, steps=5):
        super().__init__(parent)

        layout = QtWidgets.QVBoxLayout()
        self._bar = _Bar(steps)
```

```
        layout.addWidget(self._bar)

        self._dial = QtWidgets.QDial()
        self._dial.valueChanged.connect(self._bar._trigger_refresh)
        layout.addWidget(self._dial)

        self.setLayout(layout)
```

现在我们已经准备好参数来更新 `paintEvent` 方法。修改后的代码如下所示。

Listing 162. custom-widgets/powerbar_7.py

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(self._background_color)
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)

        # 获取当前状态.
        dial = self.parent()._dial
        vmin, vmax = dial.minimum(), dial.maximum()
        value = dial.value()

        # 定义我们的画布.
        d_height = painter.device().height() - (self._padding * 2)
        d_width = painter.device().width() - (self._padding * 2)

        # 绘制条.
        step_size = d_height / self.n_steps
        bar_height = step_size * self._bar_solid_percent

        # 根据范围内的值计算y轴停止位置.
        pc = (value - vmin) / (vmax - vmin)
        n_steps_to_draw = int(pc * self.n_steps)

        for n in range(n_steps_to_draw):
            brush.setColor(QtGui.QColor(self.steps[n]))
            ypos = (1 + n) * step_size
            rect = QtCore.QRect(
                self._padding,
                self._padding + d_height - int(ypos),
                d_width,
                int(bar_height),
            )
            painter.fillRect(rect, brush)
```

```
        painter.end()
```

现在您可以尝试为 `PowerBar` 的 `__init__` 方法传入不同的值，例如增加条数或提供颜色列表。以下是一些示例：

> 一个不错的十六进制颜色调色板来源是 Bokeh库。

```
PowerBar(10)
PowerBar(3)
PowerBar(["#5e4fa2", "#3288bd", "#66c2a5", "#abdda4", "#e6f598",
"#ffffbf", "#fee08b", "#fdae61", "#f46d43", "#d53e4f", "#9e0142"])
PowerBar(["#a63603", "#e6550d", "#fd8d3c", "#fdae6b", "#fdd0a2",
"#feedde"])
```

> 图188：一些PowerBar示例

您可以通过修改变量（例如：`self._bar_solid_percent`）来调整填充设置，但提供专门的方法来设置这些参数会更方便。

> 我们遵循 Qt 标准，使用驼峰式命名法为这些外部方法命名，以与从 `QDial` 继承的其他方法保持一致。

Listing 163. custom-widgets/powerbar_8.py

```
    def setColor(self, color):
        self._bar.steps = [color] * self._bar.n_steps
        self._bar.update()

    def setColors(self, colors):
```

```
        self._bar.n_steps = len(colors)
        self._bar.steps = colors
        self._bar.update()

    def setBarPadding(self, i):
        self._bar._padding = int(i)
        self._bar.update()

    def setBarSolidPercent(self, f):
        self._bar._bar_solid_percent = float(f)
        self._bar.update()

    def setBackgroundColor(self, color):
        self._bar._background_color = QtGui.QColor(color)
        self._bar.update()
```

在每种情况下，我们都设置了 `_bar` 对象上的私有变量，然后调用 `_bar.update()` 来触发控件的重绘。该方法支持将颜色更改为单一颜色，或更新颜色列表——设置颜色列表也可用于更改条形图的数量。

> 目前没有方法可以设置条数，因为扩展颜色列表会比较复杂。不过您可以尝试自己添加！

以下是一个示例，使用25像素的填充、实心条形图和灰色背景。

```
bar = PowerBar(["#49006a", "#7a0177", "#ae017e", "#dd3497", "#f768a1",
"#fa9fb5", "#fcc5c0", "#fde0dd", "#fff7f3"])
bar.setBarPadding(2)
bar.setBarSolidPercent(0.9)
bar.setBackgroundColor('gray')
```

使用这些设置，您将获得以下结果。

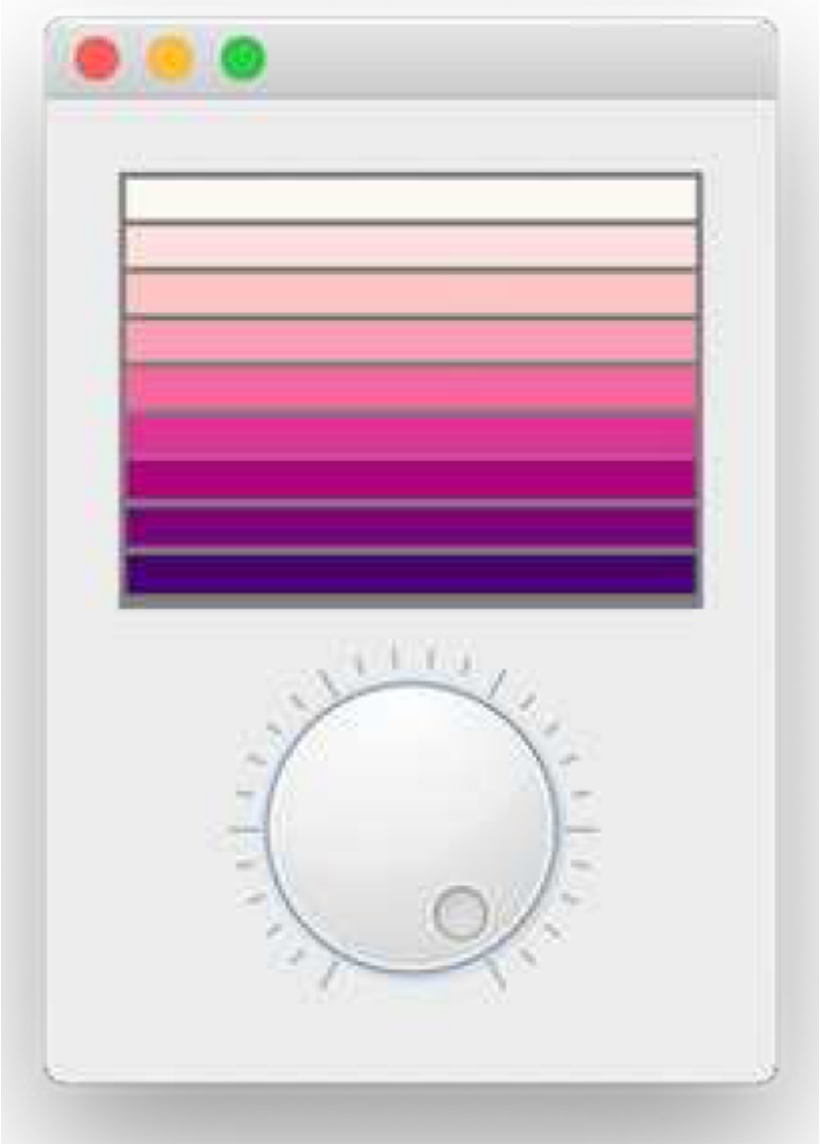


图189：配置PowerBar

添加 QAbstractSlider 接口

我们添加了用于配置电源条行为的方法。但我们目前无法从控件中配置标准 `QDial` 方法，例如设置最小值、最大值或步长。我们可以处理并为所有这些添加包装方法，但很快就会变得非常繁琐。

```
# 单个包装的示例，我们需要30多个这样的包装.
def setNotchesVisible(self, b):
    return self._dial.setNotchesVisible(b)
```

相反，我们可以在外部控件上添加一个小处理程序，以自动查找 `QDial` 实例上的方法（或属性），如果它们不直接存在于我们的类中。这样，我们就可以实现自己的方法，同时仍然可以免费获得 `QAbstractSlider` 的所有优点。

包装器如下所示，通过自定义的 `__getattr__` 方法实现。

Listing 164. custom-widgets/powerbar_8.py

```
    def __getattr__(self, name):
        if name in self.__dict__:
            return self[name]

        try:
            return getattr(self._dial, name)
        except AttributeError:
            raise AttributeError(
                "'{}' object has no attribute '{}'".format(
                    self.__class__.__name__, name
                )
            )
```

访问属性（或方法）时——例如，当我们调用 `PowerBar.setNotchesVisible(true)` 时，Python 在内部使用 `__getattr__` 从当前对象获取属性。该处理程序通过对象字典 `self.__dict__` 完成此操作。我们已覆盖此方法，以提供自定义处理逻辑。

现在，当我们调用 `PowerBar.setNotchesVisible(true)` 时，这个处理程序首先检查当前对象（一个 `PowerBar` 实例）是否存在 `.setNotchesVisible` 方法，如果存在，则使用它。如果不存在，它会调用 `self._dial` 的 `getattr()` 方法，而不是返回在那里找到的内容。这使我们能够从自定义 `PowerBar` 控件访问 `QDial` 的所有方法。这样，我们就可以从自定义的 `PowerBar` 控件访问 `QDial` 的所有方法了。

如果 `QDial` 也没有该属性，并引发 `AttributeError`，我们将捕获该错误，并从我们的自定义控件中再次引发该错误，因为该错误属于该控件。

> 这适用于任何属性或方法，包括信号。因此，标准 `QDial` 信号（如 `.valueChanged`）也可用。

由于这些更改，我们还可以简化 `paintEvent` 中的代码，直接从 `.parent()` 获取当前状态，而不是通过 `.parent()._dial`。这不会改变任何行为，但可以使代码更加易读。

Listing 165. custom-widgets/powerbar_8.py

```
    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(self._background_color)
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)
```

```
        # 获取当前状态.
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        value = parent.value()

        # 定义我们的画布.
        d_height = painter.device().height() - (self._padding * 2)
        d_width = painter.device().width() - (self._padding * 2)

        # 绘制条.
        step_size = d_height / self.n_steps
        bar_height = step_size * self._bar_solid_percent

        # 根据范围内的值计算y轴停止位置.
        pc = (value - vmin) / (vmax - vmin)
        n_steps_to_draw = int(pc * self.n_steps)

        for n in range(n_steps_to_draw):
            brush.setColor(QtGui.QColor(self.steps[n]))
            ypos = (1 + n) * step_size
            rect = QtCore.QRect(
                self._padding,
                self._padding + d_height - int(ypos),
                d_width,
                int(bar_height),
            )
            painter.fillRect(rect, brush)

        painter.end()
```

从仪表显示屏更新

目前，您可以通过旋转拨盘来更新 `PowerBar` 仪表盘的当前值。但如果您还能通过点击电源条上的某个位置，或者上下拖动鼠标来更新该值，那就更好了。为此，我们可以更新我们的 `_Bar` 控件，使其能够处理鼠标事件。

Listing 166. custom-widgets/powerbar_9.py

```
class _Bar(QtWidgets.QWidget):

    clickedValue = QtCore.pyqtSignal(int)
    def _calculate_clicked_value(self, e):
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        d_height = self.size().height() + (self._padding * 2)
        step_size = d_height / self.n_steps
        click_y = e.y() - self._padding - step_size / 2

        pc = (d_height - click_y) / d_height
        value = int(vmin + pc * (vmax - vmin))
        self.clickedValue.emit(value)

    def mouseMoveEvent(self, e):
        self._calculate_clicked_value(e)
```

```
def mousePressEvent(self, e):
        self._calculate_clicked_value(e)
```

在 PowerBar 控件的 `__init__` 块中，我们可以连接到 `Bar.clickedValue` 信号，并将值发送到 `self._dial.setValue`，以设置刻度盘上的当前值。

```
# 获取仪表上的点击事件反馈.
self._bar.clickedValue.connect(self._dial.setValue)
```

现在运行该控件，您就可以在条形区域中点击，值会更新，拨盘也会同步旋转。

最终代码

以下是我们的 PowerBar 计量表控件的完整最终代码，名为 `PowerBar`

Listing 167. custom-widgets/powerbar.py

```
from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6.QtCore import Qt


class _Bar(QtWidgets.QWidget):

    clickedValue = QtCore.pyqtSignal(int)

    def __init__(self, steps):
        super().__init__()

        self.setSizePolicy(
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
            QtWidgets.QSizePolicy.Policy.MinimumExpanding,
        )

        if isinstance(steps, list):
            # 颜色列表.
            self.n_steps = len(steps)
            self.steps = steps

        elif isinstance(steps, int):
            # 整数字条数，默认颜色为红色.
            self.n_steps = steps
            self.steps = ["red"] * steps

        else:
            raise TypeError("steps must be a list or int")

        self._bar_solid_percent = 0.8
        self._background_color = QtGui.QColor("black")
        self._padding = 4 # 边缘周围的n像素间隙.

    def paintEvent(self, e):
        painter = QtGui.QPainter(self)

        brush = QtGui.QBrush()
        brush.setColor(self._background_color)
```

```
        brush.setStyle(Qt.BrushStyle.SolidPattern)
        rect = QtCore.QRect(
            0,
            0,
            painter.device().width(),
            painter.device().height(),
        )
        painter.fillRect(rect, brush)

        # 获取当前状态.
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        value = parent.value()

        # 定义我们的画布.
        d_height = painter.device().height() - (self._padding * 2)
        d_width = painter.device().width() - (self._padding * 2)

        # 绘制条.
        step_size = d_height / self.n_steps
        bar_height = step_size * self._bar_solid_percent

        # 根据范围内的值计算y轴停止位置.
        pc = (value - vmin) / (vmax - vmin)
        n_steps_to_draw = int(pc * self.n_steps)
        for n in range(n_steps_to_draw):
            brush.setColor(QtGui.QColor(self.steps[n]))
            ypos = (1 + n) * step_size
            rect = QtCore.QRect(
                self._padding,
                self._padding + d_height - int(ypos),
                d_width,
                int(bar_height),
            )
            painter.fillRect(rect, brush)

        painter.end()

    def sizeHint(self):
        return QtCore.QSize(40, 120)

    def _trigger_refresh(self):
        self.update()

    def _calculate_clicked_value(self, e):
        parent = self.parent()
        vmin, vmax = parent.minimum(), parent.maximum()
        d_height = self.size().height() + (self._padding * 2)
        step_size = d_height / self.n_steps
        click_y = e.y() - self._padding - step_size / 2

        pc = (d_height - click_y) / d_height
        value = int(vmin + pc * (vmax - vmin))
        self.clickedValue.emit(value)

    def mouseMoveEvent(self, e):
```

```
        self._calculate_clicked_value(e)

    def mousePressEvent(self, e):
        self._calculate_clicked_value(e)


class PowerBar(QtWidgets.QWidget):
    """
    自定义 Qt 控件，用于显示电源条和拨盘。
    演示复合和自定义绘制的控件.
    """
    def __init__(self, parent=None, steps=5):
        super().__init__(parent)

        layout = QtWidgets.QVBoxLayout()
        self._bar = _Bar(steps)
        layout.addWidget(self._bar)
        # 创建 QDial 控件并设置默认值.
        # - 我们为该类提供了访问器方法，以便进行重写.
        self._dial = QtWidgets.QDial()
        self._dial.setNotchesVisible(True)
        self._dial.setWrapping(False)
        self._dial.valueChanged.connect(self._bar._trigger_refresh)
        # 从仪表上的点击事件中获取反馈.
        self._bar.clickedValue.connect(self._dial.setValue)
        layout.addWidget(self._dial)
        self.setLayout(layout)

    def __getattr__(self, name):
        if name in self.__dict__:
            return self[name]

        try:
            return getattr(self._dial, name)
        except AttributeError:
            raise AttributeError(
                "'{}' object has no attribute '{}'".format(
                    self.__class__.__name__, name
                )
            )

    def setColor(self, color):
        self._bar.steps = [color] * self._bar.n_steps
        self._bar.update()

    def setColors(self, colors):
        self._bar.n_steps = len(colors)
        self._bar.steps = colors
        self._bar.update()

    def setBarPadding(self, i):
        self._bar._padding = int(i)
        self._bar.update()

    def setBarSolidPercent(self, f):
        self._bar._bar_solid_percent = float(f)
```

```
        self._bar.update()

    def setBackgroundColor(self, color):
        self._bar._background_color = QtGui.QColor(color)
        self._bar.update()
```

您会注意到，此版本的文件不会创建 `QApplication` 或 `PowerBar` 本身的实例——它旨在作为库使用。您可以将此文件添加到自己的项目中，然后使用 `from powerbar import PowerBar` 导入，以便在自己的应用程序中使用此控件。下面的示例将 `PowerBar` 添加到标准的主窗口布局中。

Listing 168. custom-widgets/powerbar_demo.py

```
import sys
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QVBoxLayout,
    QWidget,
)

from powerbar import PowerBar


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        layout = QVBoxLayout()

        powerbar = PowerBar(steps=10)
        layout.addWidget(powerbar)

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

您应该能够将许多这些想法运用到创建自己的自定义控件中。更多示例，请参阅 学习 PyQt 控件库 ——这些控件均为开源，可免费用于您自己的项目。

23. 在 Qt Designer 中使用自定义控件

在上一章中，我们构建了一个自定义的 `PowerBar` 控件。生成的控件可以像任何内置控件一样，通过导入和添加到布局中，直接在您自己的应用程序中使用。但是，如果您使用 Qt Designer 构建应用程序用户界面该怎么办？您也可以在那里添加自定义控件吗？

答案是——是的！

在本章中，我们将逐步介绍如何将自定义控件添加到您自己的 Qt Designer 应用程序中。这个过程可能会有些复杂，但只要按照以下步骤操作，您就可以在 Designer 中创建的用户界面中使用任何自定义控件。

> 您可以使用相同的方法从其他库（例如 PyQtGraph 或 matplotlib）添加自定义控件。

背景

首先需要了解的是，您无法在 Qt Designer 中加载和显示自定义控件。Designer 中可用的控件是内置的，它无法解释您的 Python 代码来发现您创建的内容。

相反，要将控件插入到用户界面中，您需要添加占位符控件，然后告诉 Designer，您希望在应用程序运行时用自定义控件替换占位符。

在 Qt Designer 中，您会看到占位符。您可以更改与同类型控件相同的参数，这些参数将传递到自定义控件。在 Python 应用程序中加载用户界面时，PyQt6 会将自定义控件替换到相应位置。

在 Qt 中，替换占位控件的过程被称为“推广”(promoting)。内置控件被推广为自定义控件。

编写可推广的自定义控件

通过推广控件，您可以将 Qt Designer 中使用的占位控件替换为自己的自定义控件。在实现自定义控件时，您必须从另一个现有的 PyQt6 控件继承子类，即使该控件是基础 `QWidget` 也是如此。您还必须确保自定义控件实现了您子类的控件的默认构造函数。在大多数情况下，这只是意味着将父类作为 `__init__` 方法的第一个参数接受。您还必须确保自定义控件实现了您子类的控件的默认构造函数。在大多数情况下，这只是意味着将父类作为 `__init__` 方法的第一个参数接受。

> 如果自定义控件引发了错误，请在编译后的 UI 文件中检查 PyQt6 试图传递的参数。

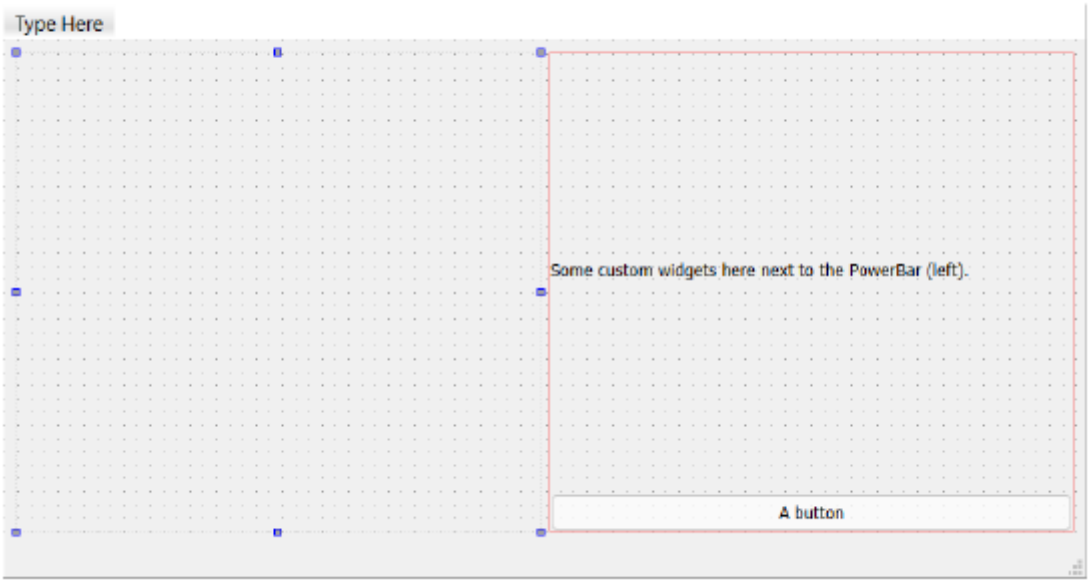
要提升为自定义控件，自定义控件必须位于与编译后的 UI 将被导入的文件分开的文件中。但是，如果您愿意，可以在同一个文件中定义多个自定义控件。

> 此限制是为了避免循环导入——如果您的应用程序文件导入编译后的 UI 文件，而该 UI 文件又反过来导入您的应用程序文件，这种情况将无法正常工作。

在文件中定义自定义控件后，请记下文件名和类名。您需要这些信息来在 Qt Designer 中推广该控件。

在 Designer 中创建和推广控件

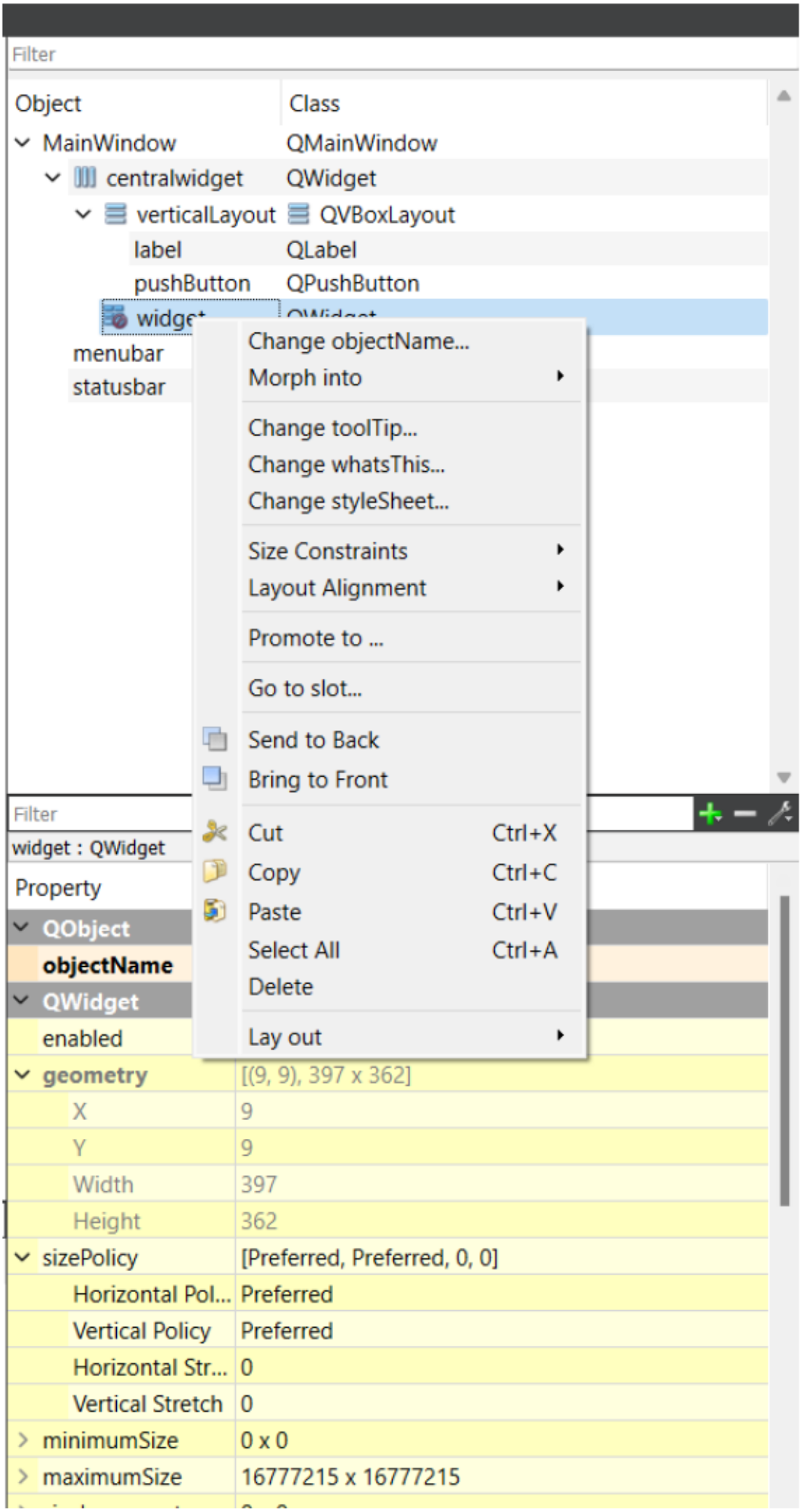
请您选择您希望自定义控件在用户界面中显示的位置，然后添加占位符控件。这里没有固定规则，但通常情况下，如果您的自定义控件继承自另一个 Qt 控件，则使用该控件作为占位符。例如，如果您基于 `QLabel` 创建了一个自定义控件，则使用 Label 作为占位符。这样，您就可以在 Designer 中访问标签的标准属性，以自定义您的自定义控件。



> 图190：简单的 UI 布局，左侧有一个占位符控件

> 您无法在设计器中更改任何自定义控件属性——Qt 设计器对您的自定义控件及其工作原理一无所知。请在代码中进行更改！

添加控件后，您可以对其进行推广。选择您想要推广的控件，右键单击，然后选择“Promote to ...”。


Filter
Object
Class
MainWindow
QMainWindow
centralwidget
QWidget
verticalLayout
QVBoxLayout
label
QLabel
pushButton
QPushButton
widget
menubar
statusbar
Change objectName...
Morph into
Change toolTip...
Change whatsThis...
Change styleSheet...
Size Constraints
Layout Alignment
Promote to ...
Go to slot...
Send to Back
Bring to Front
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Select All Ctrl+A
Delete
Lay out
Filter
widget : QWidget
Property
QObject
objectName
QWidget
enabled
geometry
[(9, 9), 397 x 362]
X
9
Y
9
Width
397
Height
362
sizePolicy
[Preferred, Preferred, 0, 0]
Horizontal Pol...
Preferred
Vertical Policy
Preferred
Horizontal Str...
0
Vertical Stretch
0
minimumSize
0 x 0
maximumSize
16777215 x 16777215

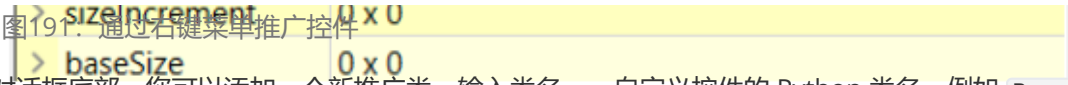


图191：通过右键菜单推广控件

在对话框底部，您可以添加一个新推广类。输入类名——自定义控件的 Python 类名，例如 `PowerBar` ——以及包含该类的 Python 文件作为头文件，省略 `.py` 后缀。

> Qt 会根据类名自动建议文件名，但会添加 `.h` （C++ 标准的头文件后缀）。您必须删除 `.h` ，即使文件名正确。

如果您的自定义控件是在子文件夹中的类中定义的，则提供完整的 Python 点表示法到文件，与您对其他导入操作一样。例如，您可能将文件放置在 `ui/widgets/powerbar.py` 下，然后输入 `ui.widgets.powerbar` 作为头文件

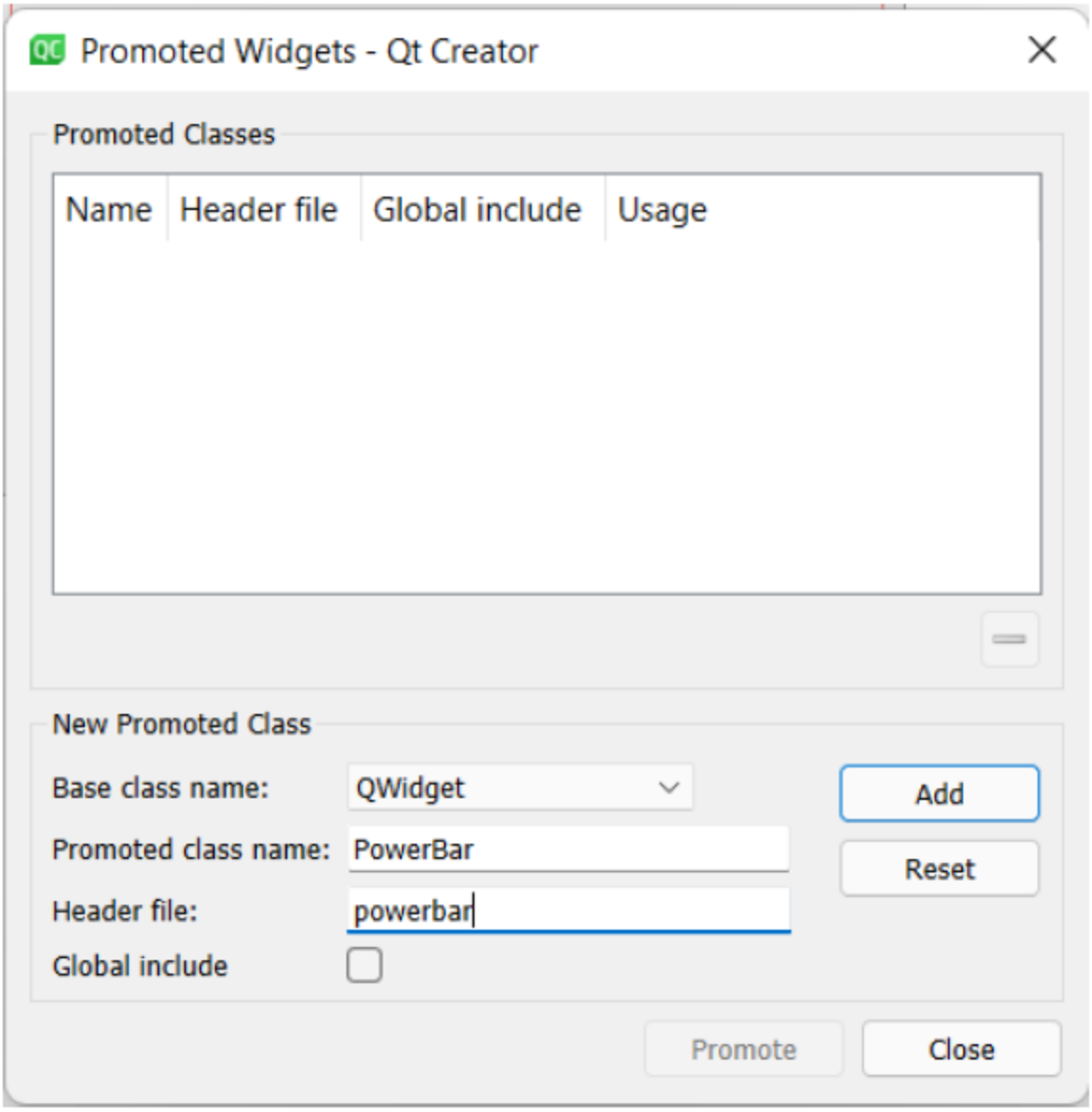
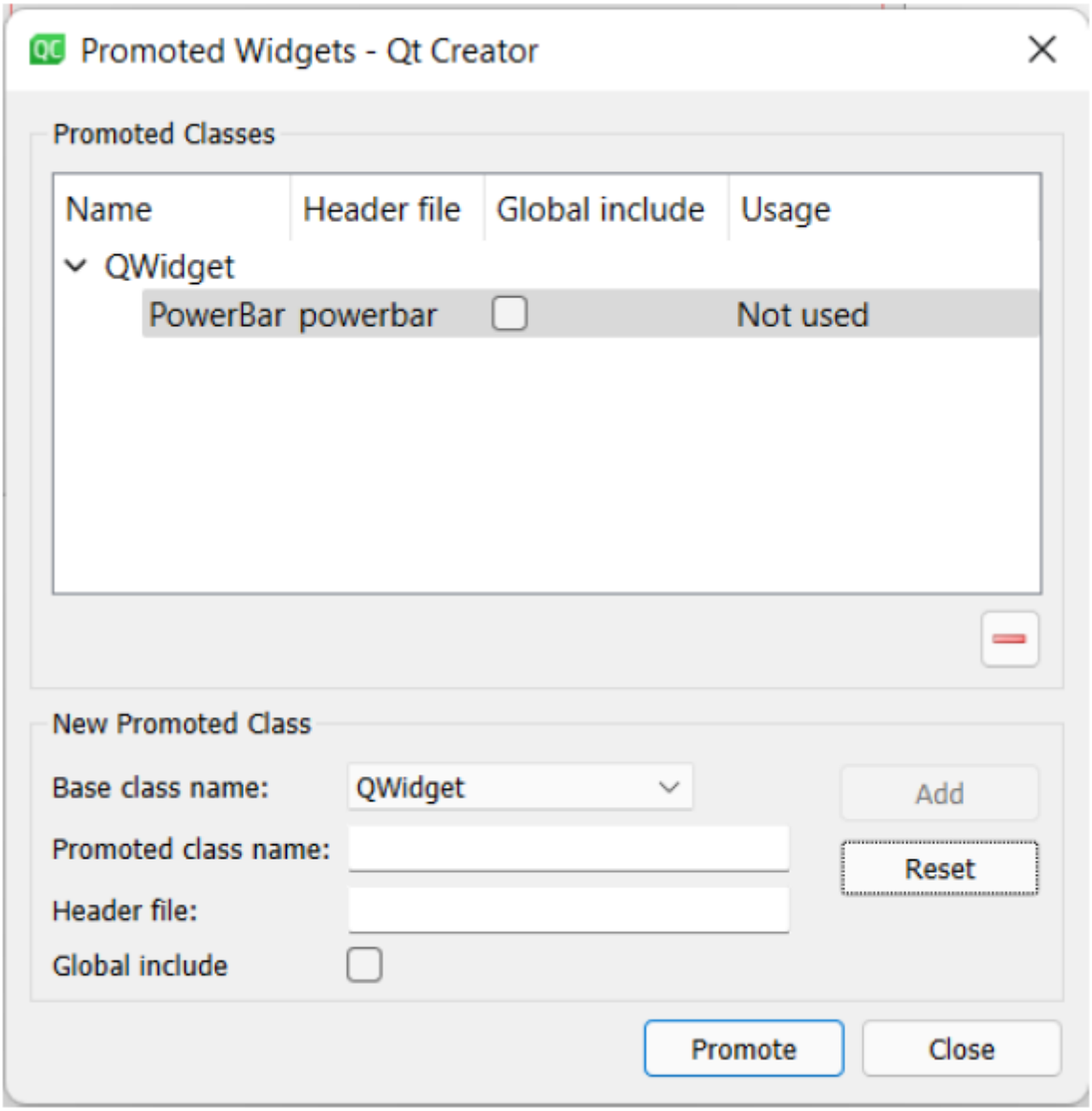


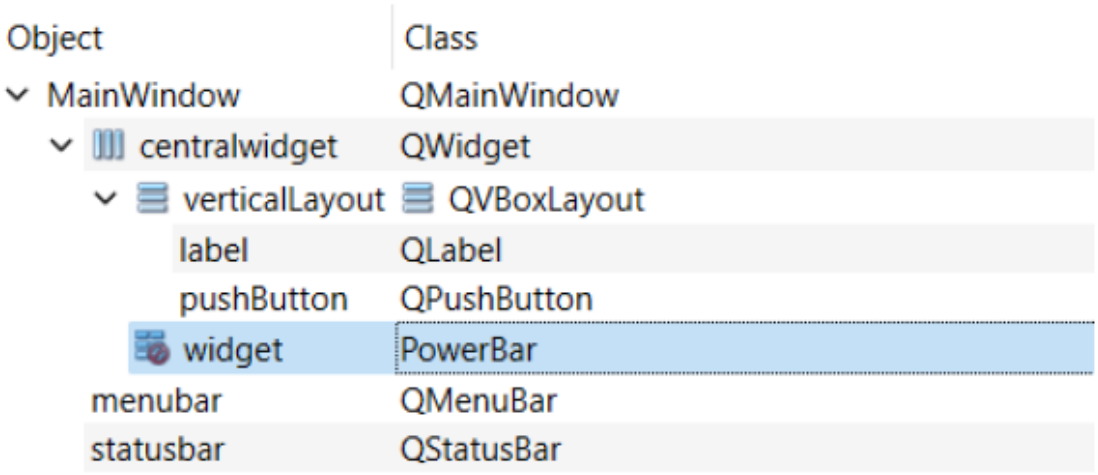
图192：添加类名和头文件

请您点击“Add”来定义推广操作。然后，您可以在顶部的列表中选择促销活动，并点击“Promote”来实际推广您的控件。



> 图193：选择推广操作并将其应用到您的控件中

控件将被推广，并显示其新类名（此处为 `PowerBar`）。



> 图194：在 UI 层叠结构中显示的推广控件

保存 UI 文件，并使用 `pyuic` 工具进行编译，与之前操作相同。

```
pyuic6 mainwindow.ui -o MainWindow.py
```

打开生成的文件，您会看到自定义的 `PowerBar` 类现在被用于在 `setupUi` 方法中构建控件，并且文件底部添加了一个新的导入。

```
class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        # etc...
        self.widget = PowerBar(self.centralwidget)

    # etc...

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow",
                                             "MainWindow"))
        self.label.setText(_translate("MainWindow", "Some custom
                                      widgets here next to the PowerBar
(left)."))

 self.pushButton.setText(_translate("MainWindow", "A button"))
                                      from powerbar import PowerBar
```

您可以像往常一样使用编译后的 UI 文件。您无需将自定义控件导入应用程序，因为编译后的 UI 文件中已经包含了该控件的滑块。

Listing 169. custom-widgets/promote_test.py

```
import random
import sys

from PyQt6.QtCore import Qt
```

```
from PyQt6.QtWidgets import QApplication, QMainWindow

from MainWindow import Ui_MainWindow


class MainWindow(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.show()


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

运行应用程序时，自定义控件将被加载并自动出现在正确的位置。

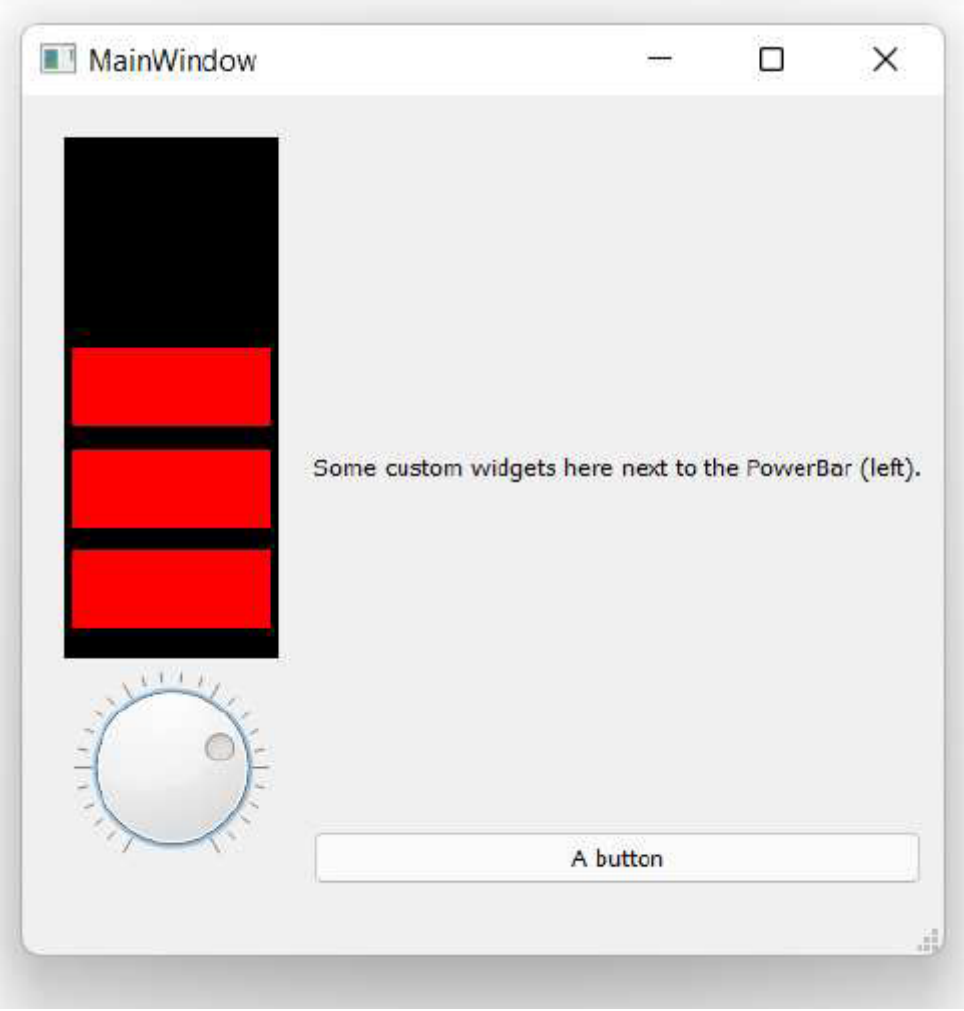


图195：应用程序中显示的 PowerBar 自定义控件

> 您看到的绝大多数错误都与导入相关。第一步应始终是检查编译后UI文件底部的导入语句，以确认其合理性。目标文件是否可达？

第三方控件

您也可以使用相同的技术将其他第三方控件添加到您的应用程序中。该过程完全相同，您只需引用控件的完全限定 Python 导入路径，并使用适当的类名即可。以下是一些常见第三方控件的配置示例

> 我们将在后续章节中详细讲解如何使用这些库！

PyQtGraph

在 Qt Designer 中，将 `PlotWidget` 作为推广的类名，将 `pyqtgraph` 作为头文件。使用 `QWidget` 作为占位控件。PyQtGraph 绘图控件将在生成的 UI 文件中按原样工作。

请参阅本书源代码下载中的 `custom-widgets/pyqtgraph_demo.py` 文件，了解可运行的演示程序。

> 图196：通过控件推广添加了 PyQtGraph 绘图控件

Matplotlib

`matplotlib` 的自定义控件 `FigureCanvasQTAgg` 不能直接在 Qt Designer 中使用，因为构造函数不接受父对象作为第一个参数，而是期望一个 `Figure` 对象。

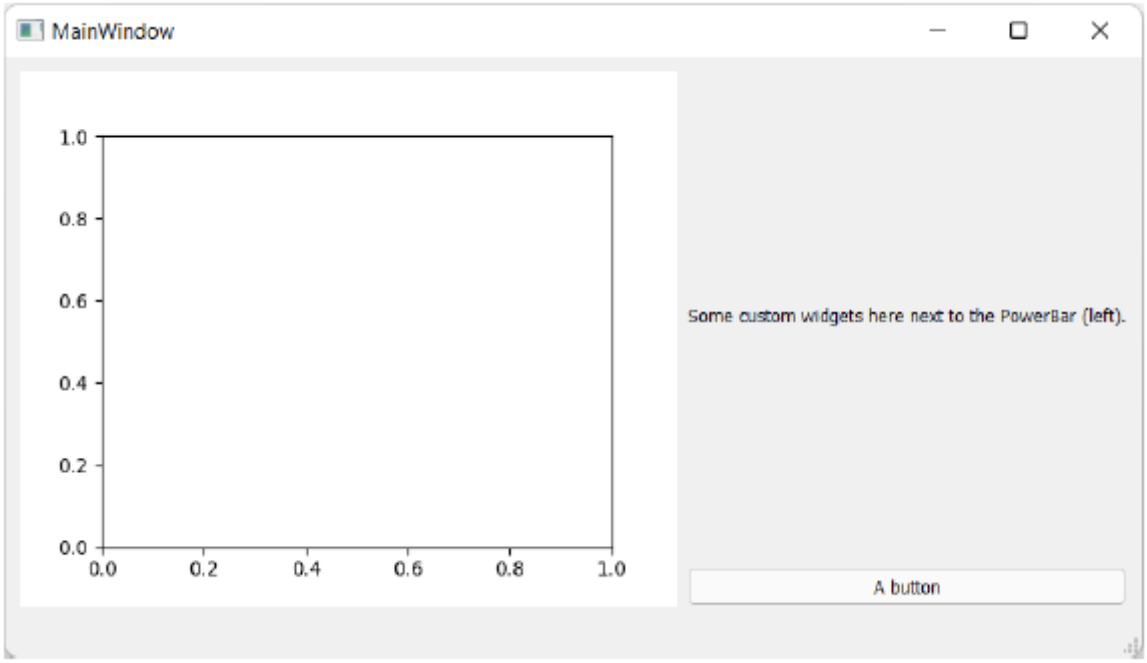
我们可以通过添加一个简单的包装类来解决这个问题，该类定义如下。

Listing 170. custom-widgets/mpl.py

```
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.figure import Figure


class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)
```

请您将此文件添加到名为 `mpl.py` 的项目中，然后在 Qt Designer 中使用 `MplCanvas` 作为推广的类名，使用 `mpl` 作为头文件。使用 `QWidget` 作为占位控件。

请参阅本书源代码下载中的 `custom-widgets/matplotlib_demo.py` 文件，了解可运行的演示程序。



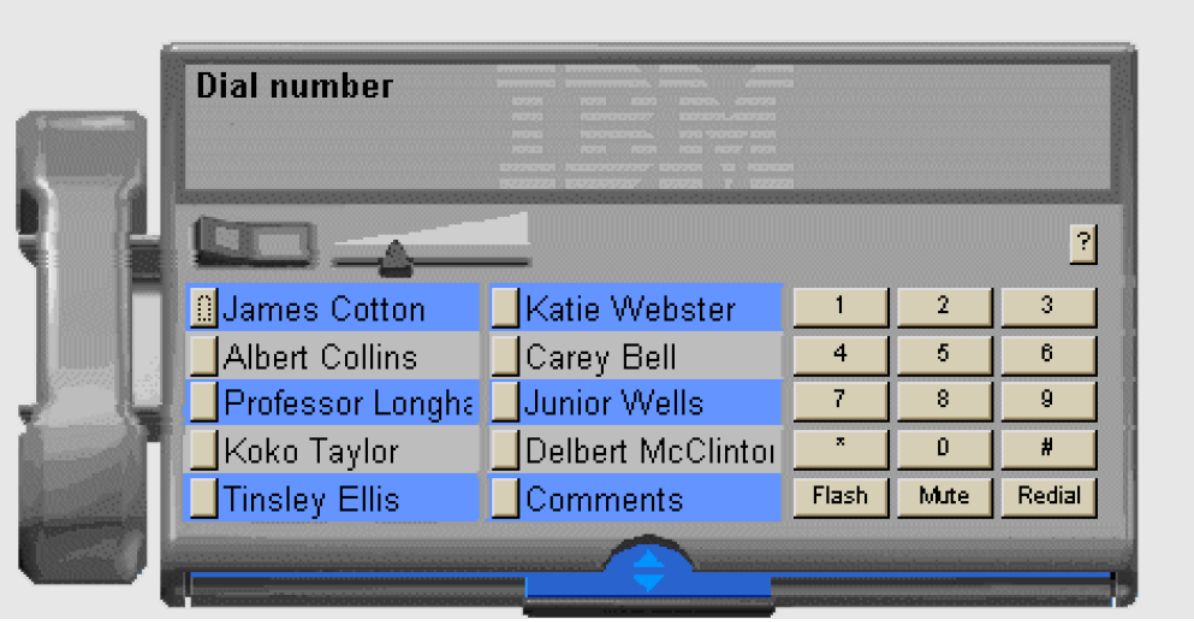
> 图197：通过控件推广添加的 matplotlib 控件

使用这些技术，您应该能够在 PyQt6 应用程序中使用任何自定义控件。

熟悉度与拟物化设计

在构建用户界面时，最强大的工具之一就是熟悉感。也就是说，让用户觉得你的界面是他们之前使用过的。熟悉的界面通常被描述为直观的。在屏幕上移动鼠标指针并点击方形凸起本身并没有什么直观之处。但是，经过多年的重复操作，这种行为本身就变得非常熟悉了。

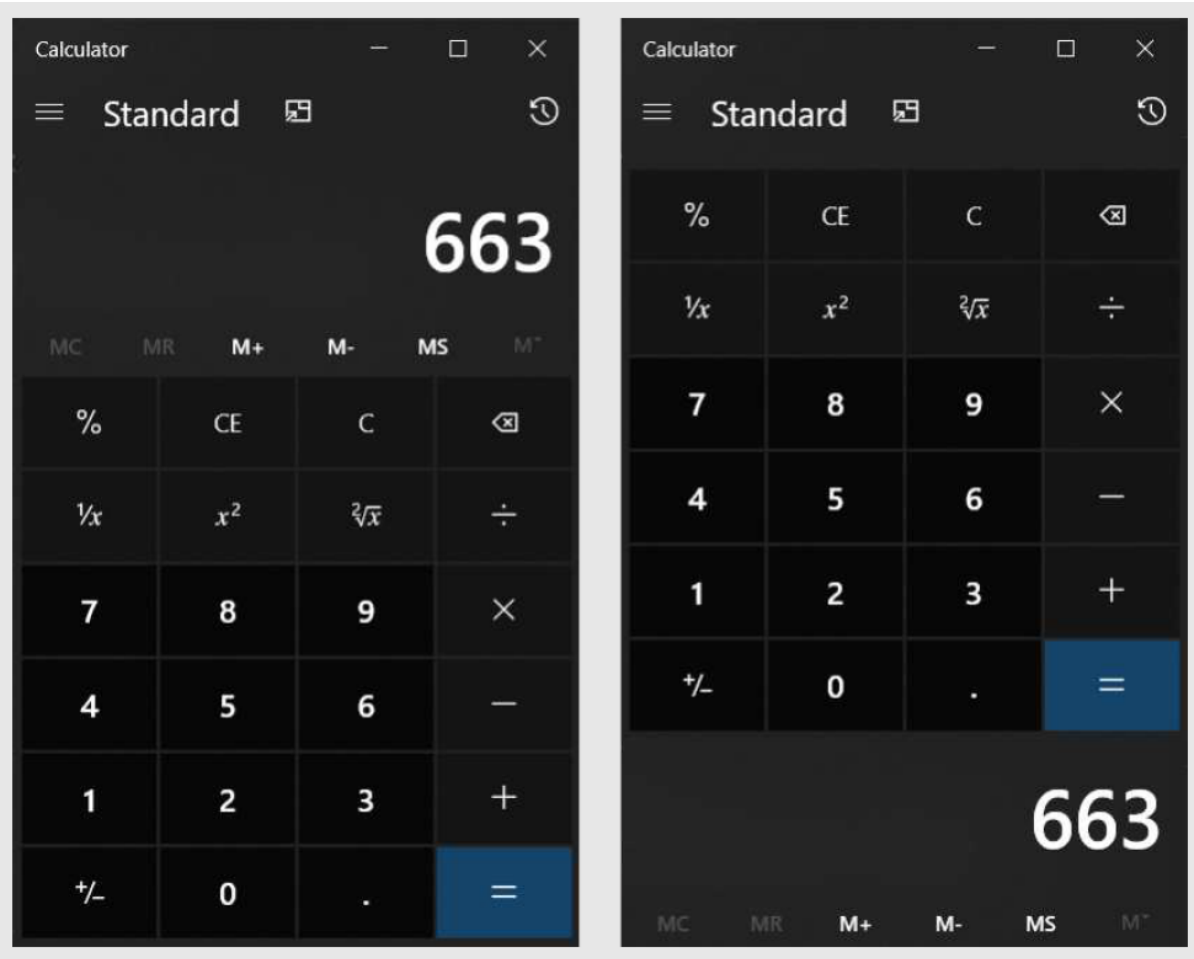
在用户界面中寻找熟悉感导致了拟物化设计。拟物化设计是指将非功能性设计元素应用于物体，而这些设计元素是功能性的。这意味着使用常见的界面元素，或者制作看起来像真实物体的界面。虽然近年来图形用户界面趋势又回到了抽象的“扁平化”设计，但所有现代用户界面都保留了拟物化的元素。



> RealPhone — IBM RealThings™ 系列产品之一

现代桌面计算器就是一个很好的例子。当我们进行计算时，我们会将结果显示在底部。那么为什么计算器的屏幕会位于顶部呢？因为如果屏幕位于底部，它会被您的手挡住。因此，屏幕的位置是出于功能性考虑。

对于计算机上的计算器，这个位置被保留下来，尽管它已经不再具有功能性——鼠标指针不会遮挡屏幕，输入通常通过键盘进行。但如果您打开一个计算器，发现屏幕在底部，您就会感到困惑。它看起来像是倒置的。尽管完全可用，但它看起来奇怪或反直觉。这就是拟物主义的本质——通过利用用户对现有物体的熟悉感，使用户界面感觉更直观。



> 计算器与倒置计算器（Windows 10）

您的软件在这一尺度上的位置由你自行决定。关键在于要了解现有的接口，并在可能的情况下充分利用它们来提升你自己应用程序的易用性。您的用户会因此感激不尽！

- **请务必**在设计自己的界面时适当地借鉴现有界面的设计元素。
- **请务必**在需要提升用户体验的地方适当地加入拟物化设计元素。

并发执行

> 计算机不应浪费您的时间，也不应要求您做超过严格必要的工作。
>
> ——Jef Raskin,，用户界面设计第二定律

通过调用 `QApplication` 对象上的 `.exec()` 启动的事件循环在与 Python 代码相同的线程中运行。运行此事件循环的线程（通常称为图形用户界面线程）还负责处理与主机操作系统之间的所有窗口通信。

默认情况下，事件循环触发的任何执行也将在该线程中同步运行。实际上，这意味着当您的 PyQt6 应用程序在代码中执行某项操作时，窗口通信和图形用户界面交互都会被冻结。

如果您正在执行的操作比较简单，并且能够快速将控制权返回给图形用户界面循环，那么用户不会察觉到这种冻结现象。但是，如果您需要执行较长时间的任务，例如打开/写入大文件、下载一些数据或渲染一些复杂的图像，就会出现问题。对于您的用户来说，应用程序似乎没有响应。由于您的应用程序不再与操作系统通信，操作系统会认为它已经崩溃——在 macOS 上，您会看到“死亡旋转轮”；在 Windows 上，您会看到“蓝屏”。这显然不是理想的用户体验。

解决方案很简单——将工作从图形用户界面线程中移出。PyQt6 提供了直观的界面来完成这项工作。

24. 线程与进程简介

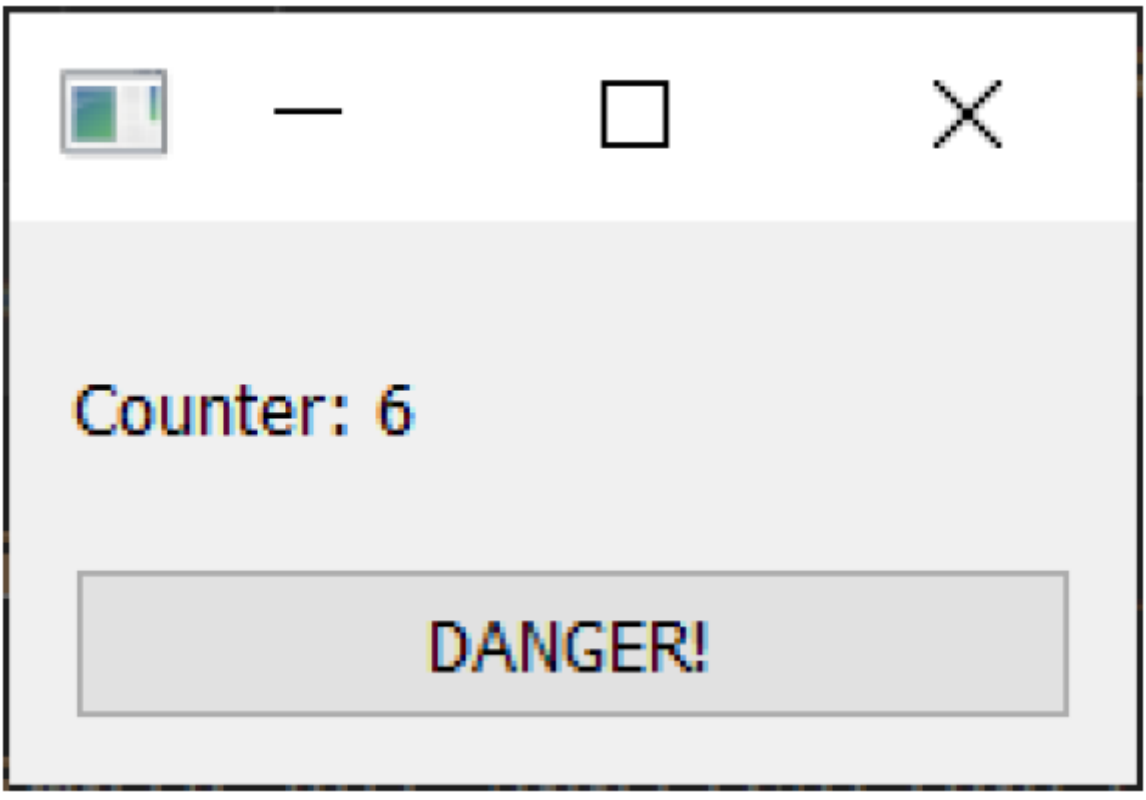
以下是一个用于 PyQt6 的最小示例应用程序，它将使我们能够演示问题并随后进行修复。您可以将此代码复制并粘贴到一个新文件中，并将其保存为适当的文件名，例如 `concurrent.py`。

Listing 171. bad_example_1.py

```
import sys
import time

from PyQt6.QtCore import QTimer
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.counter = 0

        layout = QVBoxLayout()
```

```
        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)

        w = QWidget()
        w.setLayout(layout)
        self.setCentralWidget(w)

        self.show()

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def oh_no(self):
        time.sleep(5)

    def recurring_timer(self):
        self.counter += 1
        self.l.setText("Counter: %d" % self.counter)


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

> 🚀 **运行它吧！** 将出现一个窗口，其中包含一个按钮和一个数字，该数字正在向上计数。



> 图198：该数字将以每秒增加1的速度持续增长，只要事件循环仍在运行。

这是由一个简单的定时器生成的，每秒触发一次。您可以将它视为我们的事件循环指示器——这是一种简单的方式，让我们知道应用程序正在正常运行。还有一个标有“危险！”的按钮。请您试着点击它。

> 图199：按下按钮！！

您会发现每次按下按钮时计数器都会停止计数，而您的应用程序会完全冻结。在Windows系统中，您可能会看到窗口变为浅色，表明其未响应，而在macOS系统中，您可能会看到旋转的“死亡之轮”。

看似冻结的界面实际上是由于 Qt 事件循环被阻塞，无法处理（并响应）窗口事件。您对窗口的点击仍会被宿主操作系统记录并发送至您的应用程序，但由于这些事件被卡在您代码中的时间延迟（`time.sleep`）部分，应用程序无法接受或响应这些事件。因此，应用程序无法响应，操作系统将其解读为冻结或卡死。

错误的方法

解决此问题的最简单方法是在代码内部处理事件。这将允许 Qt 继续响应主机操作系统，而您的应用程序将保持响应。您可以通过使用 `QApplication` 类的静态 `.processEvents()` 函数轻松实现这一点。只需在您的长运行代码块中的某个位置添加类似以下的代码行：

```
QApplication.processEvents()
```

如果我们将长期运行的 `time.sleep` 代码分解为多个步骤，我们可以在其中插入 `.processEvents`。相应的代码如下：

```
def oh_no(self):
    for n in range(5):
        QApplication.processEvents()
        time.sleep(1)
```

现在，当您按下按钮时，您的代码会像以前一样被执行。然而，现在`QApplication.processEvents()`会间歇性地将控制权交还给 Qt，并允许它像往常一样响应操作系统事件。Qt 现在会接受事件并处理它们，然后返回运行您的其余代码。

这确实有效，但有几个原因让它变得糟糕。

首先，当您将控制权交还给 Qt 时，您的代码将不再运行。这意味着您试图执行的任何耗时操作都会花费更长时间。这可能不是您想要的结果。

其次，在主事件循环之外处理事件会导致您的应用程序在循环中分支到处理代码（例如，触发槽或事件）。如果您的代码依赖于/响应外部状态，这可能会导致未定义的行为。下面的代码演示了这种情况。

Listing 172. bad_example_2.py

```
import sys
import time

from PyQt6.QtCore import QTimer
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        c = QPushButton("?")
        c.pressed.connect(self.change_message)

        layout.addWidget(self.l)
        layout.addWidget(b)

        layout.addWidget(c)

        w = QWidget()
        w.setLayout(layout)
```

```
        self.setCentralWidget(w)

        self.show()

    def change_message(self):
        self.message = "OH NO"

    def oh_no(self):
        self.message = "Pressed"

        for _ in range(100):
            time.sleep(0.1)
            self.l.setText(self.message)
            QApplication.processEvents()


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

如果您运行这段代码，您会看到计数器与之前相同。按下“DANGER!”按钮将将显示的文本更改为“Pressed”，正如在 `oh_no` 函数的入口处所定义的。然而，如果您在oh_no仍在运行时按下“?”按钮，您将看到消息发生变化。状态正在从循环外部进行更改。

这是一个简单的示例。然而，如果您在应用程序中有多个长时间运行的进程，每个进程都调用 `QApplication.processEvents()` 来保持系统运行，您的应用程序行为可能会迅速变得不可预测。

线程与进程

如果您退一步思考，想想您在应用程序中希望发生的事情，它可能可以概括为“在其他事情发生的同时发生的事情”。在计算机上运行独立任务有两种主要方法：线程和进程。

线程共享相同的内存空间，因此启动速度快且消耗的资源极少。共享内存使得在线程之间传递数据变得非常简单，然而，不同线程读写内存可能会导致竞争条件或段错误。然而，Python 还存在另一个问题，即多个线程受同一全局解释器锁（GIL）的限制——这意味着不释放 GIL 的 Python 代码只能在单个线程中执行。然而，对于 PyQt6 而言，这并非主要问题，因为大部分时间都花在 Python 之外。

进程使用独立的内存空间（以及完全独立的 Python 解释器）。这避免了与 GIL 相关的潜在问题，但代价是启动时间较长、内存开销较大以及在发送/接收数据时复杂性增加。

为了简化起见，通常使用线程是明智的选择。Qt中的进程更适合于运行和与外部程序通信。在本章中，我们将探讨在Qt内部可用的选项，以将工作转移到单独的线程和进程中。

25. 使用线程池

Qt 提供了一个非常简单的接口，用于在其他线程中运行任务，该接口在 PyQt6 中得到了很好的实现。该接口围绕两个类构建——`QRunnable` 和 `QThreadPool`。前者是用于存放您要执行的任务的容器，而后者则是管理您的工作线程的线程池。

使用 `QThreadPool` 的好处在于，它可以为您处理任务的排队和执行滑块。除了排队作业和检索结果外，几乎无需做其他事情。

使用 QRunnable

要定义自定义 `QRunnable`，您可以继承基础 `QRunnable` 类，然后将希望执行的代码放置在 `run()` 方法中。以下是将我们的长时间 `sleep` 任务实现为 `QRunnable` 的示例。将以下代码添加到 `MainWindow` 类定义之前。

Listing 173. concurrent/qrunnable_1.py

```
class Worker(QRunnable):
    """
    工作线程
    """
    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中
        """
        print("Thread start")
        time.sleep(5)
        print("Thread complete")
```

在另一个线程中执行我们的函数，只需创建一个 `Worker` 的实例，然后将其传递给我们的`QThreadPool` 实例即可。

我们在 `__init__` 块中创建一个线程池的实例。

Listing 174. concurrent/qrunnable_1.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )
```

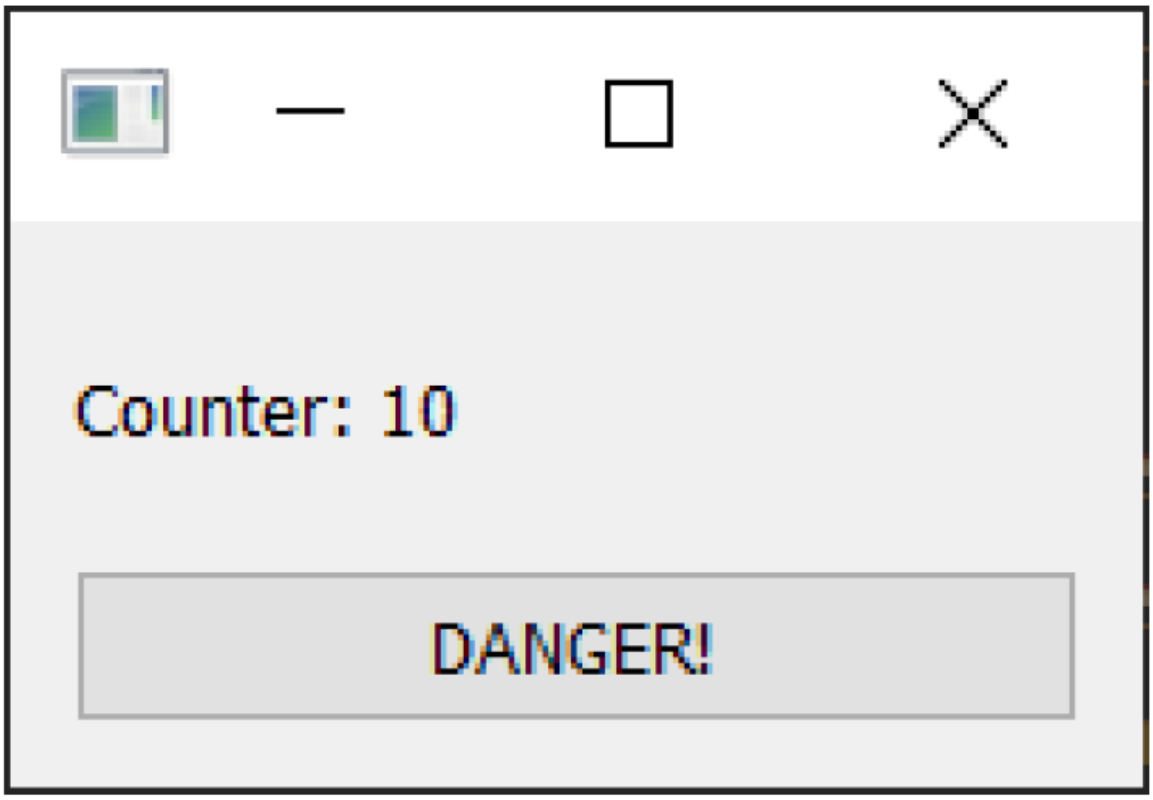
最后，用以下代码替换 `oh_no` 方法，以创建并提交

Listing 175. concurrent/qrunnable_1.py

```
    def oh_no(self):
        worker = Worker()
        self.threadpool.start(worker)
```

现在，点击按钮将创建一个工作进程来处理（长期运行的）进程，并通过 `QThreadPool` 池将其分拆到另一个线程中。如果没有足够的线程来处理传入的工作进程，它们将被排入队列，并在稍后按顺序执行。

> 🚀 **运行它吧！** 您会发现，现在应用程序可以处理您疯狂点击按钮的操作，而不会出现任何问题。



> 图200：简单的 `QRunnable` 示例应用程序。只要图形用户界面线程正在运行计数器就会每秒增加 1

请您查看控制台输出，以观察工作进程的启动和完成情况。

```
Multithreading with maximum 12 threads
Thread start
Thread start
Thread start
Thread complete
Thread complete
Thread complete
```

检查多次点击按钮时会发生什么。您应该看到您的线程立即执行，直到达到 `.maxThreadCount` 报告的数量。如果在已经存在此数量的活跃工作者后再次点击按钮，后续的工作者将被排队，直到有线程可用。

在此示例中，我们让 `QThreadPool` 决定理想的活跃线程数。这个数值在不同计算机上会有所不同，目的是实现最佳性能。然而，有时您可能需要指定特定的线程数——在这种情况下，您可以使用 `.setMaxThreadCount` 方法显式设置此值。此值是针对每个线程池的。

使用 `QThreadPool.start()`

在之前的示例中，我们自己创建了一个 `QRunnable` 对象，并将其传递给 `QThreadPool` 以进行执行。然而，对于简单的用例，Qt通过 `QThreadPool.start()` 提供了一个便捷的方法，该方法可以处理执行任意的 Python 函数和方法。Qt 会为您创建必要的 `QRunnable` 对象，并将它们排入队列。

在下面的示例中，我们将工作放在了 `do_some_work` 方法中，并修改了 `oh_no` 方法，将其传递给线程池的 `.start()` 方法。

Listing 176. concurrent/qthreadpool_start_1.py

```
    def oh_no(self):
        self.threadpool.start(self.do_some_work)

    @pyqtSlot()
    def do_some_work(self):
        print("Thread start")
        time.sleep(5)
        print("Thread complete")

    def recurring_timer(self):
        self.counter += 1
        self.l.setText("Counter: %d" % self.counter)
```

按下按钮将执行我们在 `QThreadPool` 上定义的 `do_some_work` 方法。

> 您可以通过这种方式启动多个线程。尝试按下按钮，直到达到最大并发线程数。在线程池中有空闲空间之前，不会启动新的线程。

对于许多简单的任务来说，这种方法非常有效。在执行的函数中，您可以访问信号，并使用它们来发出数据。您无法接收信号——没有地方连接它们——但您可以通过 `self` 对象与变量进行交互。

更新代码以添加以下 `custom_signal`，并修改 `work` 方法以发出此信号并更新 `self.counter` 变量。

Listing 177. concurrent/qthreadpool_start_2.py

```
class MainWindow(QMainWindow):

    custom_signal = pyqtSignal()

    def __init__(self):
        super().__init__()

        # 将我们的自定义信号连接到处理程序.
        self.custom_signal.connect(self.signal_handler)
        # etc.
    def oh_no(self):
        self.threadpool.start(self.do_some_work)

    @pyqtSlot()
    def do_some_work(self):
        print("Thread start")
        # 发出我们的定制信号.
        self.custom_signal.emit()
        for n in range(5):
            time.sleep(1)
        self.counter = self.counter - 10
        print("Thread complete")
```

```
    def signal_handler(self):
        print("Signal received!")

    def recurring_timer(self):
        self.counter += 1
        self.l.setText("Counter: %d" % self.counter)
```

运行此示例后，您会发现，虽然工作方法在另一个线程中运行（睡眠不会中断计数器），但我们仍然能够发出信号并修改 `self.counter` 变量。

> 您无法从另一个线程直接修改图形用户界面——尝试这样做会导致应用程序崩溃。

> 您可以使用信号修改图形用户界面。例如，尝试将 `str` 信号连接到标签的 `.setText` 方法。

虽然这是一个方便的小界面，但您经常会发现自己希望对正在运行的线程有更多的控制权，或者与它们进行更结构化的通信。接下来，我们将通过一些更复杂的示例，使用 `QRunnable` 来展示什么是可能的。

扩展 QRunnable

如果您想将自定义数据传递给执行函数，您可以配置您的运行器以接受参数或关键字，然后将这些数据存储在 `QRunnable` 的 `self` 对象中。这些数据随后可以在 `run` 方法内部访问。

Listing 178. concurrent/qrunnable_2.py

```
class Worker(QRunnable):
    """
    工作线程
    :param args: 传递给运行代码的参数
    :param kwargs: 传递给运行的关键字参数
    :code
    :
    """
    def __init__(self, *args, **kwargs):
        super().__init__()
        self.args = args
        self.kwargs = kwargs
```

```
    @pyqtSlot()
    def run(self):
    """
    使用传递的 self.args 初始化 runner 函数，
    """
        print(self.args, self.kwargs)

    def oh_no(self):
        worker = Worker("some", "arguments", keywords=2)
        self.threadpool.start(worker)
```

> 由于函数在 Python 中也是对象，您还可以将一个函数传递给运行器以执行。请参阅后文的通用示例来获取一个示例。

线程 I/O

有时，能够从运行中的工作者进程中传递状态和数据会非常有用。这可能包括计算结果、抛出的异常或正在进行的进度（例如进度条）。Qt 提供了信号和槽框架，允许您执行此操作，并且是线程安全的，允许从正在运行的线程直接与图形用户界面进行安全通信。信号允许您发出值，然后这些值由与 `.connect` 链接的槽函数在代码的其他位置拾取。

下面是一个简单的 `WorkerSignals` 类，它包含一些示例信号。信号。

> 自定义信号只能在从 `QObject` 派生的对象上定义。由于 `QRunnable` 并非从 `QObject` 派生，因此我们无法直接在其中定义信号。使用一个用于保存信号的自定义 `QObject` 是最简单的解决方案。

Listing 179. concurrent/qrunnable_3.py

```
class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号。
    支持的信号包括：
    finished
        无数据
    error
        `str` 异常字符串
    result
        `dict` 处理返回的数据
```

```
    """
    finished = pyqtSignal()
    error = pyqtSignal(str)
    result = pyqtSignal(dict)
```

在这个例子中，我们定义了 3 个自定义信号：

1. **完成信号**，没有数据表明任务何时完成
2. **错误信号**，它接收一个由异常类型、异常值和格式化跟踪信息组成的元组。
3. **结果信号**，接收执行函数的任何对象类型

您可能并不需要所有这些信号，但它们被包括进来是为了表明其可能性。在下面的代码中，我们使用这些信号来通知一个简单的计算工作进程的完成和错误。

Listing 180. concurrent/qrunnable_3.py

```
import random
import sys
import time

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号。
    支持的信号包括：
    finished
        无数据
    error
        `str` 异常字符串
    result
        `dict` 处理返回的数据
    """
    finished = pyqtSignal()
    error = pyqtSignal(str)
    result = pyqtSignal(dict)


class Worker(QRunnable):
```

```
    """
    工作线程
    :param args: 传递给运行代码的参数
    :param kwargs: 传递给运行的关键字参数
    :code
    :
    """
    def __init__(self, iterations=5):
        super().__init__()
        self.signals = (
            WorkerSignals()
        ) # 创建信号类的实例.
        self.iterations = iterations

    @pyqtSlot()
    def run(self):
    """
    使用传递的 self.args 初始化 runner 函数,
    """
        try:
            for n in range(self.iterations):
                time.sleep(0.01)
                v = 5 / (40 - n)

        except Exception as e:
            self.signals.error.emit(str(e))

        else:
            self.signals.finished.emit()
            self.signals.result.emit({"n": n, "value": v})


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)

        w = QWidget()
        w.setLayout(layout)
```

```
        self.setCentralWidget(w)

        self.show()

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def oh_no(self):
        worker = Worker(iterations=random.randint(10, 50))
        worker.signals.result.connect(self.worker_output)
        worker.signals.finished.connect(self.worker_complete)
        worker.signals.error.connect(self.worker_error)
        self.threadpool.start(worker)


    def worker_output(self, s):
        print("RESULT", s)

    def worker_complete(self):
        print("THREAD COMPLETE!")

    def worker_error(self, t):
        print("ERROR: %s" % t)

    def recurring_timer(self):
        self.counter += 1
        self.l.setText("Counter: %d" % self.counter)


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

您可以将自己的处理函数连接到这些信号，以接收线程完成（或结果）的通知。\该示例旨在偶尔抛出除以零异常，您将在输出中看到该异常。

```
Multithreading with maximum 12 threads
THREAD COMPLETE!
RESULT {'n': 16, 'value': 0.20833333333333334}
ERROR: division by zero
THREAD COMPLETE!
RESULT {'n': 11, 'value': 0.1724137931034483}
THREAD COMPLETE!
RESULT {'n': 22, 'value': 0.2777777777777778}
ERROR: division by zero
```

在下一节中，我们将探讨这种方法的几种不同变体，这些变体使您能够在自己的应用程序中使用 `QThreadPool` 实现一些有趣的功能。

26. QRunnable 示例

`QThreadPool` 和 `QRunnable` 是以其他线程运行程序的一种非常灵活的方式。通过调整信号和参数，您可以执行任何可以想象到的任务。在本章中，我们将介绍一些示例，说明如何为特定场景构建运行器。

所有示例都遵循相同的总体模式——一个自定义的 `QRunnable` 类，带有自定义的 `WorkerSignals`。区别在于我们传递给运行器的参数、运行器对这些参数的处理方式，以及我们如何连接信号。

Listing 181. concurrent/qrunnable_base.py

```
import sys
import time
import traceback

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import QApplication, QMainWindow


class WorkerSignals(QObject):
    pass


class Worker(QRunnable):
    def __init__(self, *args, **kwargs):
        super().__init__()
        # 存储构造函数参数（用于后续处理）
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        pass


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.show()


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

进度观察器

如果您正在使用线程来执行耗时较长的操作，您应该让用户了解任务的进展情况。一种常见的做法是通过显示一个进度条来实现，该进度条会以从左到右填充的方式，显示任务完成的进度。为了在任务中显示进度条，您需要从 `worker` 中发出当前的进度状态。

为此，我们可以在 `WorkerSignals` 对象上定义另一个名为 `progress` 的信号。该信号在每个循环中发出 0..100 之间的数字，以表示"任务"的进展。该进度信号的输出连接到主窗口状态栏上显示的标准 `QProgressBar`。

Listing 182. concurrent/qrunnable_progress.py

```
import sys
import time

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QProgressBar,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号
    progress
        int 进度完成度，范围为0到100
    """
    progress = pyqtSignal(int)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    """
    def __init__(self):
        super().__init__()

        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
```

```
        total_n = 1000
        for n in range(total_n):
            progress_pc = int(
                100 * float(n + 1) / total_n
            ) # 进度 0-100% ,作为整数
            self.signals.progress.emit(progress_pc)
            time.sleep(0.01)


class MainWindow(QMainWindow):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        layout = QVBoxLayout()

        self.progress = QProgressBar()

        button = QPushButton("START IT UP")
        button.pressed.connect(self.execute)
        layout.addWidget(self.progress)
        layout.addWidget(button)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )

    def execute(self):
        worker = Worker()
        worker.signals.progress.connect(self.update_progress)
        # 执行
        self.threadpool.start(worker)

    def update_progress(self, progress):
        self.progress.setValue(progress)


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

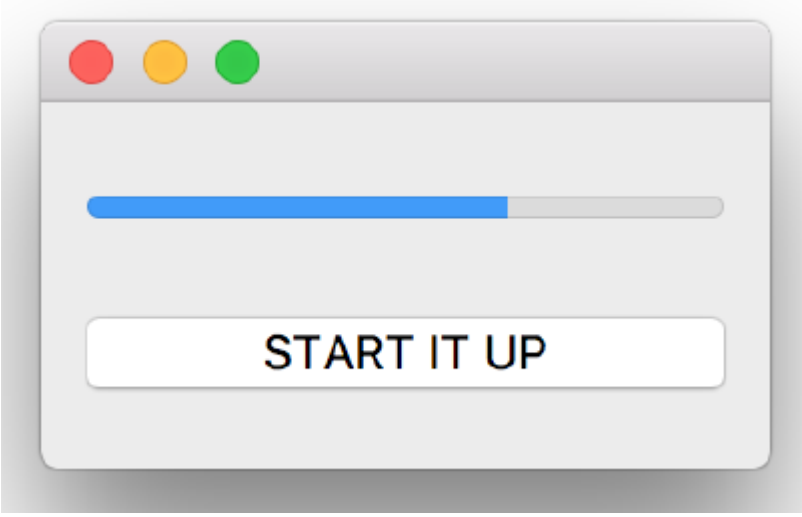


图201：进度条显示长期运行的任务的当前进度

如果您在另一个进程已经运行的情况下按下按钮，您会发现一个问题——两个进程会将它们的进度发送到同一个进度条，因此数值会来回跳动。

使用单个进度条跟踪多个工作者是可行的——我们只需要做两件事：一个用于存储每个工作者进度值的存储位置，以及一个每个工作者的唯一标识符。在每次进度更新时，我们可以计算所有工作者的平均进度，并显示该值。

Listing 183. concurrent/qrunnable_progress_many.py

```
import random
import sys
import time
import uuid

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QProgressBar,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号
    progress
        int 进度完成度，范围为0到100
    """
    progress = pyqtSignal(str, int)
```

```
    finished = pyqtSignal(str)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    """
    def __init__(self):
        super().__init__()
        self.job_id = uuid.uuid4().hex #1
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        total_n = 1000
        delay = random.random() / 100 # 随机延迟值.
        for n in range(total_n):
            progress_pc = int(100 * float(n + 1) / total_n) #2
            self.signals.progress.emit(self.job_id, progress_pc)
            time.sleep(delay)

        self.signals.finished.emit(self.job_id)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        layout = QVBoxLayout()

        self.progress = QProgressBar()

        button = QPushButton("START IT UP")
        button.pressed.connect(self.execute)

        self.status = QLabel("0 workers")

        layout.addWidget(self.progress)
        layout.addWidget(button)
        layout.addWidget(self.status)

        w = QWidget()
        w.setLayout(layout)

        # 字典记录了当前工作器的工作进度。.
        self.worker_progress = {}

        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
```

```
        )

        self.timer = QTimer()
        self.timer.setInterval(100)
        self.timer.timeout.connect(self.refresh_progress)
        self.timer.start()

    def execute(self):
        worker = Worker()
        worker.signals.progress.connect(self.update_progress)
        worker.signals.finished.connect(self.cleanup) #3

        # 执行
        self.threadpool.start(worker)

    def cleanup(self, job_id):
        if job_id in self.worker_progress:
            del self.worker_progress[job_id] #4

            # 如果我们移除了某个值，请更新进度条
            self.refresh_progress()

    def update_progress(self, job_id, progress):
        self.worker_progress[job_id] = progress

    def calculate_progress(self):
        if not self.worker_progress:
            return 0

        return sum(v for v in self.worker_progress.values()) / len(
            self.worker_progress
        )

    def refresh_progress(self):
        # 计算总进度.
        progress = self.calculate_progress()
        print(self.worker_progress)
        self.progress.setValue(progress)
        self.status.setText("%d workers" % len(self.worker_progress))


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

1. 为该任务执行器使用一个唯一的UUID4标识符。
2. 进度以0-100%的整数形式表示。
3. 当任务完成后，需要清理（删除）任务执行器的进度数据。
4. 删除已完成任务执行器的进度数据。

运行此代码后，您将看到全局进度条以及一个指示器，用于显示当前正在运行的活跃工作者数量。

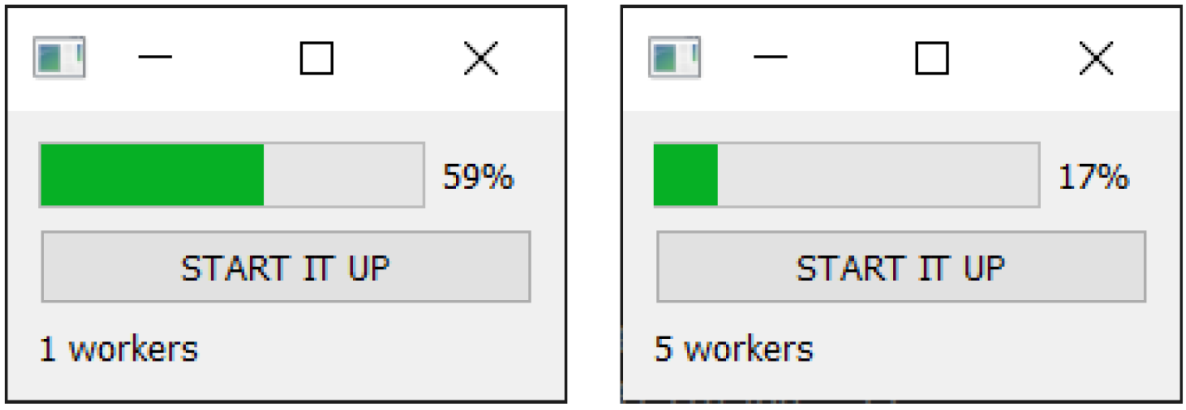


图202：显示全局进度状态的窗口，以及活跃工作者的数量。

通过查看脚本的控制台输出，您可以查看每个独立工作节点的实际状态。

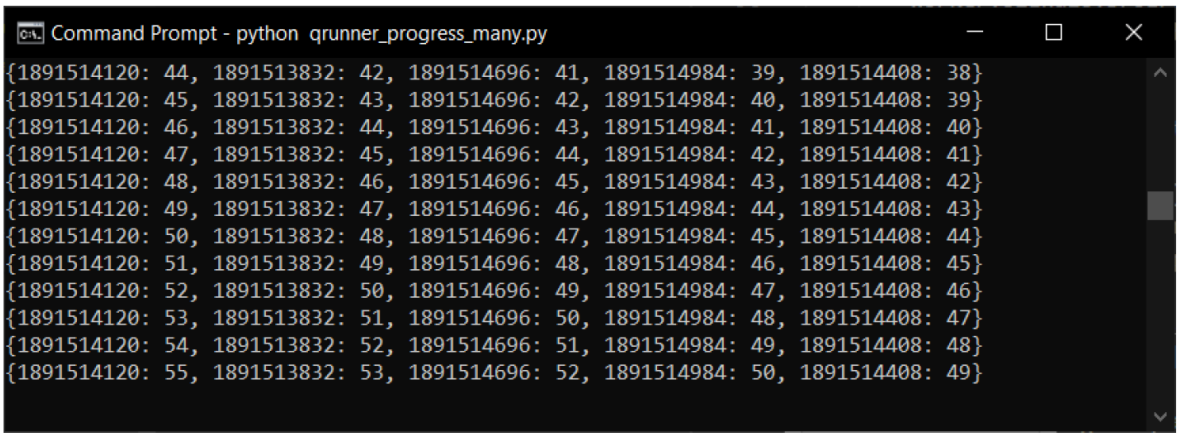


图203：查看 shell 输出以查看每个工作进程的进度。

立即移除工作者意味着进度会倒退。当任务完成时，从平均值计算中移除100会导致平均值下降。您可以推迟清理操作，例如以下代码仅在所有进度条达到100%时移除条目：

Listing 184. concurrent/qrunnable_progress_many_2.py

```
import random
import sys
import time
import uuid

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QProgressBar,
    QPushButton,
    QVBoxLayout,
    QWidget,
)
```

```python
class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号
    progress
        int 进度完成度，范围为0到100
    """
    progress = pyqtSignal(str, int)
    finished = pyqtSignal(str)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    """
    def __init__(self):
        super().__init__()
        self.job_id = uuid.uuid4().hex #1
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        total_n = 1000
        delay = random.random() / 100 # 随机延迟值.
        for n in range(total_n):
            progress_pc = int(100 * float(n + 1) / total_n) #2
            self.signals.progress.emit(self.job_id, progress_pc)
            time.sleep(delay)

        self.signals.finished.emit(self.job_id)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        layout = QVBoxLayout()

        self.progress = QProgressBar()

        button = QPushButton("START IT UP")
        button.pressed.connect(self.execute)

        self.status = QLabel("0 workers")

        layout.addWidget(self.progress)
        layout.addWidget(button)
        layout.addWidget(self.status)

        w = QWidget()
        w.setLayout(layout)

        # 字典记录了当前工作器的工作进度。.
        self.worker_progress = {}
```

```
        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )

        self.timer = QTimer()
        self.timer.setInterval(100)
        self.timer.timeout.connect(self.refresh_progress)
        self.timer.start()

    def execute(self):
        worker = Worker()
        worker.signals.progress.connect(self.update_progress)
        worker.signals.finished.connect(self.cleanup) #3

        # 执行
        self.threadpool.start(worker)

    def cleanup(self, job_id):
        if all(v == 100 for v in self.worker_progress.values()):
            self.worker_progress.clear() # 清空字典

            # 如果我们移除了某个值，请更新进度条
            self.refresh_progress()

    def update_progress(self, job_id, progress):
        self.worker_progress[job_id] = progress

    def calculate_progress(self):
        if not self.worker_progress:
            return 0

        return sum(v for v in self.worker_progress.values()) / len(
            self.worker_progress
        )

    def refresh_progress(self):
        # 计算总进度.
        progress = self.calculate_progress()
        print(self.worker_progress)
        self.progress.setValue(progress)
        self.status.setText("%d workers" % len(self.worker_progress))


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

虽然这可以正常工作，对于简单的用例来说也没问题，但如果这个工作状态（和控制）能够被封装到它自己的管理器组件中，而不是通过主窗口来控制，那就更好了。请参阅后面的“管理器”部分，了解如何做到这一点。

计算器

当您需要执行复杂的计算时，多线程是一个不错的选择。如果您使用的是 Python numpy、scipy 或 pandas 库，那么这些计算也可能释放 Python 全局解释器锁 (GIL)，这意味着您的图形用户界面和计算线程都可以全速运行。

在本例中，我们将创建一些执行一些简单计算的作业。这些计算的结果将返回图形用户界面线程，并在图表中显示。

> 我们在后文的 使用PyQtGraph进行绘图 中对 PyQtGraph 进行详细介绍，目前仅需关注 `QRunnable`。

Listing 185. concurrent/qrunnable_calculator.py

```
import random
import sys
import time
import uuid

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)
import pyqtgraph as pg


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号

    data
        元组数据点 (worker_id, x, y)
    """
```

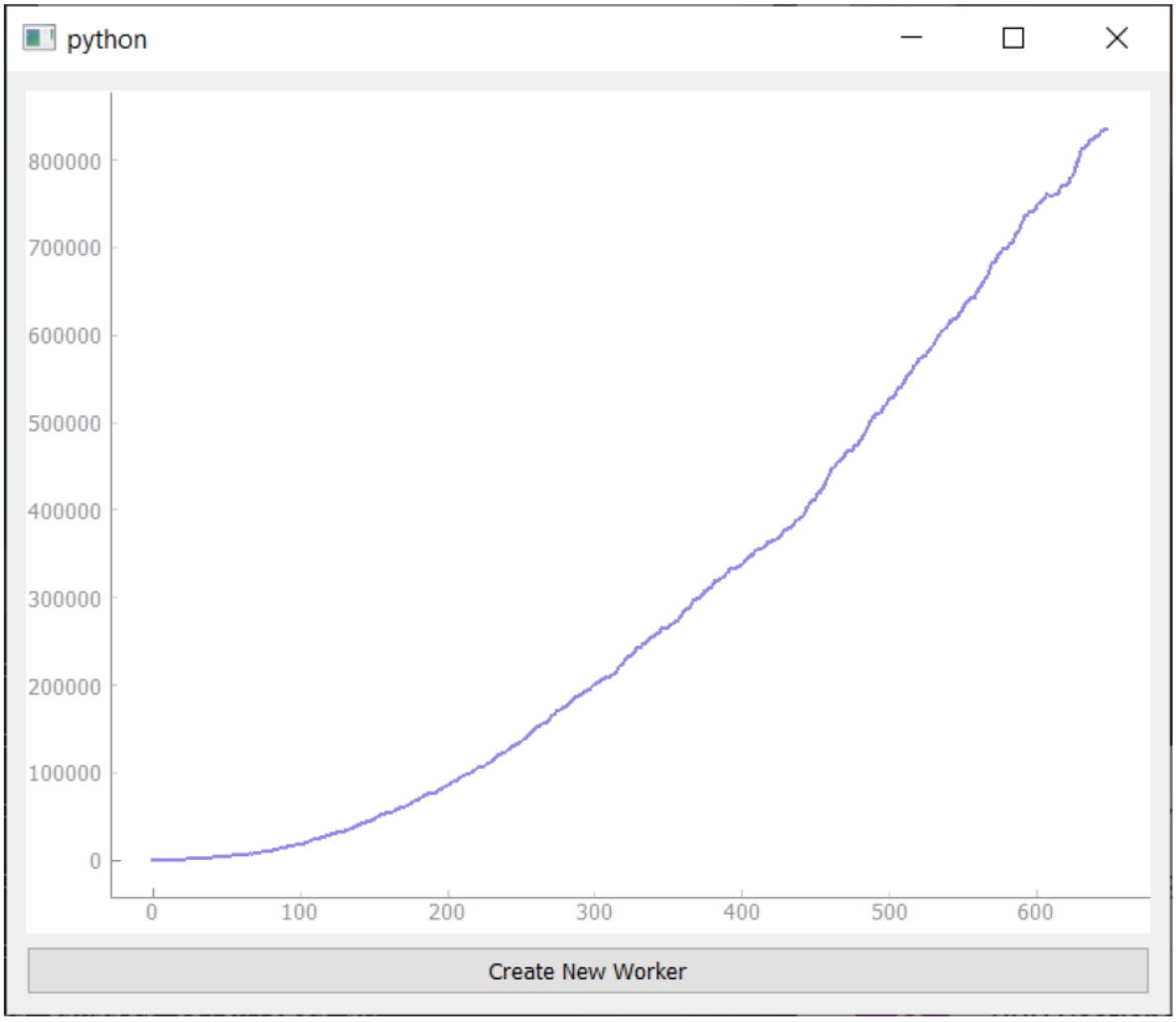
```
    data = pyqtSignal(tuple) #1


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    """
    def __init__(self):
        super().__init__()
        self.worker_id = uuid.uuid4().hex # 此项工作的唯一标识符
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):

        total_n = 1000
        y2 = random.randint(0, 10)
        delay = random.random() / 100 # 随机延迟值.
        value = 0

        for n in range(total_n):
            # 假设计算，每个工作将生产不同的结果值。
            # 由于y和y2的随机值.
            y = random.randint(0, 10)
            value += n * y2 - n * y

            self.signals.data.emit((self.worker_id, n, value)) #2
            time.sleep(delay)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.threadpool = QThreadPool()

        self.x = {} # 保持时间点.
        self.y = {} # 保留数据.
        self.lines = {} # 保留绘制线的引用，以便更新.

        layout = QVBoxLayout()
        self.graphWidget = pg.PlotWidget()
        self.graphWidget.setBackground("w")
        layout.addWidget(self.graphWidget)

        button = QPushButton("Create New Worker")
        button.pressed.connect(self.execute)

        # layout.addWidget(self.progress)
        layout.addWidget(button)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)
```

```
        self.show()

    def execute(self):
        worker = Worker()
        worker.signals.data.connect(self.receive_data)

        # 执行
        self.threadpool.start(worker)

    def receive_data(self, data):
        worker_id, x, y = data #3

        if worker_id not in self.lines:
            self.x[worker_id] = [x]
            self.y[worker_id] = [y]
            # 生成一个随机颜色.
            pen = pg.mkPen(
                width=2,
                color=(
                    random.randint(100, 255),
                    random.randint(100, 255),
                    random.randint(100, 255),
                ),
            )
            self.lines[worker_id] = self.graphWidget.plot(
                self.x[worker_id], self.y[worker_id], pen=pen
            )
            return

        # 更新现有图例/数据
        self.x[worker_id].append(x)
        self.y[worker_id].append(y)

        self.lines[worker_id].setData(
            self.x[worker_id], self.y[worker_id]
        )


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

1. 设置自定义信号以传递数据。使用元组可以发送任何数量的值，这些值被包装在元组中。
2. 这里，我们发出 worker_id、x 和 y 值。
3. 接收器槽解包数据。

一旦您从工作处获取了数据，您可以随心所欲地处理它——例如将其添加到表格或模型视图中。在这里，我们正在将 x 和 y 值存储在以 `worker_id` 为键的字典对象中。这样可以将每个工作的数据保持独立，并允许我们单独绘制它们。

如果您运行这个示例并按下按钮，您会在图表上看到一条线出现并逐渐延长。如果您再次按下按钮，另一个工作将开始运行，返回更多数据并在图表上添加另一条线。每个工作以不同的速率生成数据，每个工作生成100个值。



> 图204：在经过几次迭代后，从单个运行器中输出结果图。

您可以启动新任务，数量最多可达机器上可用的最大线程数。生成100个值后，任务将关闭，接下来排队的任务将启动并将其值作为新行添加。

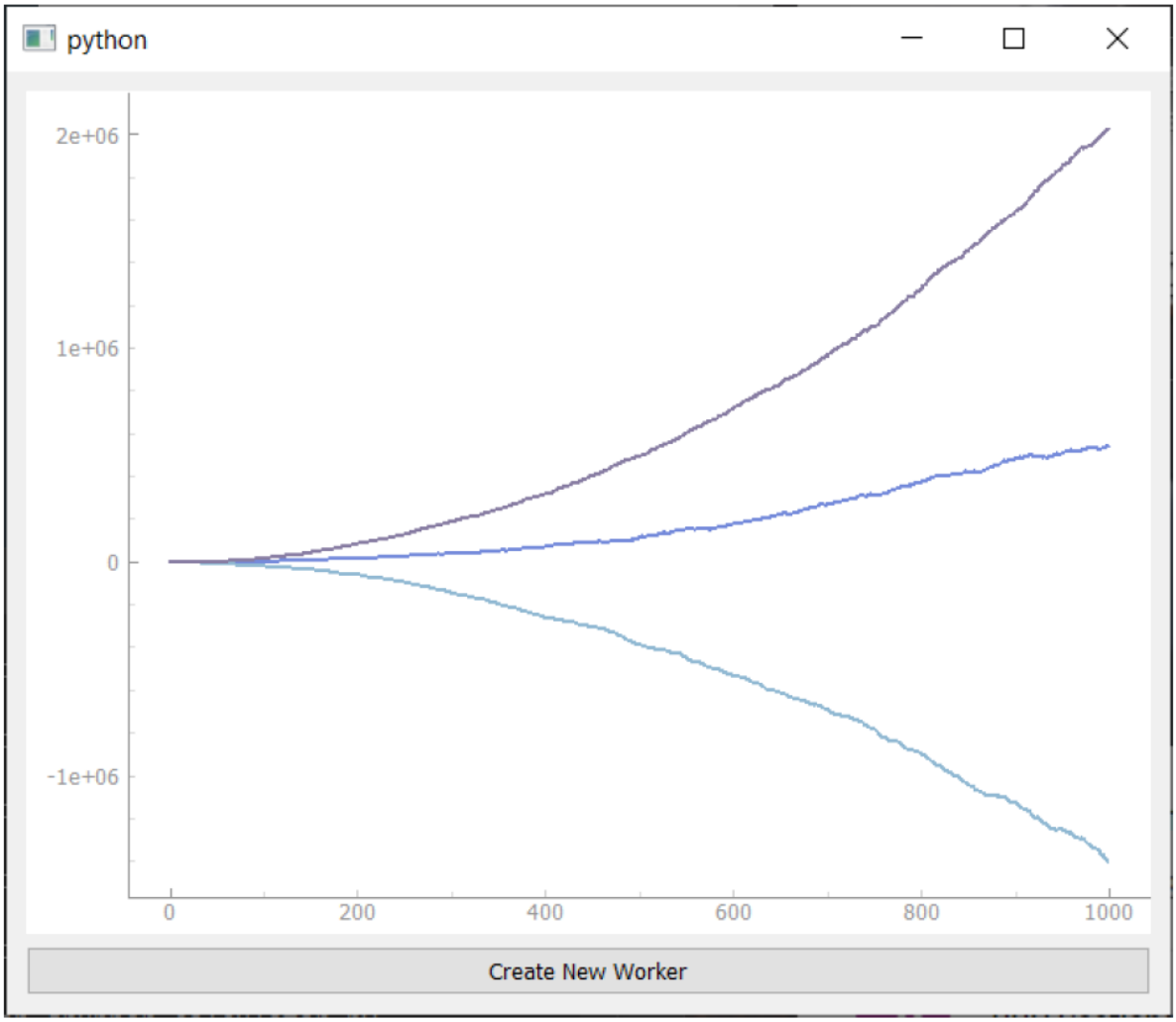


图205：来自多个运行器的数据

当然，元组是可选的，如果您只有一个运行器，或者不需要将输出与源关联，您可以返回裸字符串。通过适当设置信号，您还可以发送字节字符串或任何其他类型的数据。

停止正在运行的QRunnable

一旦启动了 QRunnable，默认情况下无法停止它。从可用性角度来看，这并不理想——如果用户误启动了任务，他们就只能坐等任务完成。遗憾的是，无法直接终止运行器，但我们可以变相地请求其停止。在本示例中，我们将探讨如何通过设置标志位来指示运行器需要停止。

在计算中，标志是用于信号当前状态或状态变化的变量。想想船只如何使用旗帜进行通信。

图206：旗语，“你应该立即停止你的船只。”

下面的代码实现了一个简单的运行器，带有进度条，该进度条每 0.01 秒从左向右增加，以及一个 [停止] 按钮。如果您点击 [停止]，该进程将退出，永久停止进度条。

Listing 186. concurrent/qrunnable_stop.py

```
import sys
import time

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    Qt,
    QThreadPool,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QMainWindow,
    QProgressBar,
    QPushButton,
    QWidget,
)

class WorkerKilledException(Exception):
    pass


class WorkerSignals(QObject):
    progress = pyqtSignal(int)


class JobRunner(QRunnable):
    signals = WorkerSignals()
    def __init__(self):
        super().__init__()
        self.is_killed = False #1

    @pyqtSlot()
    def run(self):
        try:
```

```
            for n in range(100):
                self.signals.progress.emit(n + 1)
                time.sleep(0.1)

                if self.is_killed: #2
                    raise WorkerKilledException

        except WorkerKilledException:
                pass #3

    def kill(self): #4
        self.is_killed = True


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 一些按钮
        w = QWidget()
        l = QHBoxLayout()
        w.setLayout(l)

        btn_stop = QPushButton("Stop")

        l.addWidget(btn_stop)

        self.setCentralWidget(w)

        # 创建状态栏.
        self.status = self.statusBar()
        self.progress = QProgressBar()
        self.status.addPermanentWidget(self.progress)

        # 线程运行器
        self.threadpool = QThreadPool()

        # 创建一个运行器
        self.runner = JobRunner()
        self.runner.signals.progress.connect(self.update_progress)
        self.threadpool.start(self.runner)

        btn_stop.pressed.connect(self.runner.kill)

        self.show()

    def update_progress(self, n):
        self.progress.setValue(n)


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

1. 用于指示是否应终止运行器的标志称为 `.is_killed`。

> 2. 在每个循环中，我们测试 `.is_killed` 是否为 `True`，如果是，则抛出异常。
> 3. 捕获异常，我们可以在这里发出完成或错误信号。
> 4. `.kill()` 是便利函数，这样我们就可以调用 `worker.kill()` 来终止它。

如果您想在不引发异常的情况下停止 `worker`，只需从 `run` 方法中直接返回，例如：

```
    def run(self):
        for n in range(100):
            self.signals.progress.emit(n + 1)
            time.sleep(0)

        if self.is_killed:
            return
```

在上述示例中，我们只有一个工作。然而，在许多应用程序中，您会有更多的工作。当您有多个运行器在运行时，如何停止工作？

如果您希望停止所有工作进程，那么无需进行任何更改。您只需将所有工作进程连接到相同的“停止”信号，当该信号被触发时（例如按下一个按钮），所有工作进程都会同时停止。

如果您想能够停止单个工作，您需要在用户界面的某个位置为每个工作创建一个单独的按钮，或者实现一个管理器来跟踪工作并提供一个更友好的界面来终止它们。请查看后文的 管理器 以获取一个可工作的示例。

暂停一个运行器

暂停一个运行器是一种较少见的需求——通常您希望事情能尽可能快地进行。但有时您可能希望让一个工作进入“睡眠”状态，使其暂时停止从数据源读取数据。您可以通过对停止运行器的方法进行一些小修改来实现这一点。实现这一功能的代码如下所示。

> 暂停的运行程序仍然占用线程池中的一个槽，限制了可运行的并发任务的数量。请谨慎使用！

Listing 187. concurrent/qrunnable_pause.py

```
import sys
import time

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    Qt,
    QThreadPool,
    pyqtSignal,
    pyqtSlot,
)

from PyQt6.QtWidgets import (
```

```
    QApplication,
    QHBoxLayout,
    QMainWindow,
    QProgressBar,
    QPushButton,
    QWidget,
)


class WorkerKilledException(Exception):
    pass


class WorkerSignals(QObject):
    progress = pyqtSignal(int)


class JobRunner(QRunnable):

    signals = WorkerSignals()

    def __init__(self):
        super().__init__()
        self.is_paused = False
        self.is_killed = False

    @pyqtSlot()
    def run(self):
        for n in range(100):
            self.signals.progress.emit(n + 1)
            time.sleep(0.1)

            while self.is_paused:
                time.sleep(0) #1

            if self.is_killed:
                raise WorkerKilledException

    def pause(self):
        self.is_paused = True

    def resume(self):
        self.is_paused = False

    def kill(self):
        self.is_killed = True


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 一些按钮
        w = QWidget()
        l = QHBoxLayout()
        w.setLayout(l)
```

```
        btn_stop = QPushButton("Stop")
        btn_pause = QPushButton("Pause")
        btn_resume = QPushButton("Resume")

        l.addWidget(btn_stop)
        l.addWidget(btn_pause)
        l.addWidget(btn_resume)
        self.setCentralWidget(w)

        # 创建状态栏.
        self.status = self.statusBar()
        self.progress = QProgressBar()
        self.status.addPermanentWidget(self.progress)

        # 线程运行器
        self.threadpool = QThreadPool()

        # 创建一个运行器
        self.runner = JobRunner()
        self.runner.signals.progress.connect(self.update_progress)
        self.threadpool.start(self.runner)

        btn_stop.pressed.connect(self.runner.kill)
        btn_pause.pressed.connect(self.runner.pause)
        btn_resume.pressed.connect(self.runner.resume)

        self.show()

    def update_progress(self, n):
        self.progress.setValue(n)


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

> 1. 如果您不想频繁检查是否到了醒来的时间，可以在 `sleep` 调用中设置一个大于 `0` 的值。

如果您运行这个示例，您会看到一个进度条从左向右移动。如果您点击[暂停]，工作将暂停。如果您接下来点击[继续]，工作将从它开始的地方继续。如果您点击[停止]，工作将永久停止，就像以前一样。

在收到 `is_paused` 信号时，我们不会抛出异常，而是进入一个暂停循环。这会停止工作进程的执行，但不会退出 `run` 方法或终止工作进程。

通过使用 `while self.is_paused:` 循环，当 `worker` 恢复运行时，我们将立即退出循环，并继续之前的工作。

> 您必须包含 `time.sleep()` 调用。这个零秒暂停允许 Python 释放 GIL，因此该循环不会阻塞其他执行。如果没有这个 `sleep`，您将有一个忙循环，它会在不做任何事情的情况下浪费资源。如果您想更少地检查，请增加 `sleep` 值。

通信器

在运行线程时，您经常希望能够实时获取线程正在执行的内容的输出，即在执行过程中获取输出。

在此示例中，我们将创建一个在单独线程中向远程服务器发起请求的运行器，并将其输出转发至日志记录器。我们还将探讨如何将自定义解析器传递给运行器，以从请求中提取我们感兴趣的额外数据。

> 如果您希望从外部进程而非线程中记录数据，请参阅 “运行外部进程” 和 “运行外部命令和进程” 。

数据导出

在这个第一个示例中，我们将使用自定义信号将每个请求的原始数据（HTML）转储到输出中。

Listing 188. concurrent/qrunnable_io.py

```
import sys
import requests

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPlainTextEdit,
    QPushButton,
    QVBoxLayout,
```

```
    QWidget,
)


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号。
    data
        元组 (identifier, data)
    """
    data = pyqtSignal(tuple)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。

    :param id: 该工作节点的ID
    :param url: 用于获取数据的URL
    """

    def __init__(self, id, url):
        super().__init__()
        self.id = id
        self.url = url

        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        r = requests.get(self.url)

        for line in r.text.splitlines():
            self.signals.data.emit((self.id, line))


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.urls = [
            "https://www.pythonguis.com/",
            "https://www.mfitzp.com/",
            "https://www.google.com",
            "https://academy.pythonguis.com/",
        ]

        layout = QVBoxLayout()

        self.text = QPlainTextEdit()
        self.text.setReadOnly(True)

        button = QPushButton("GO GET EM!")
        button.pressed.connect(self.execute)

        layout.addWidget(self.text)
```

```
        layout.addWidget(button)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )

    def execute(self):
        for n, url in enumerate(self.urls):
            worker = Worker(n, url)
            worker.signals.data.connect(self.display_output)

            # 执行
            self.threadpool.start(worker)

     def display_output(self, data):
        id, s = data
        self.text.appendPlainText("WORKER %d: %s" % (id, s))


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

> 译者注：`self.urls`中的网站在国内访问基本都很不稳定，甚至您访问不了Google。您可以试试百度等国内公共网站，或者使用科学上网

如果您运行这个示例并点击按钮，您会看到来自多个网站的HTML输出，这些输出前面会加上获取它们的worker ID。请注意，来自不同工作的输出是交错显示的。

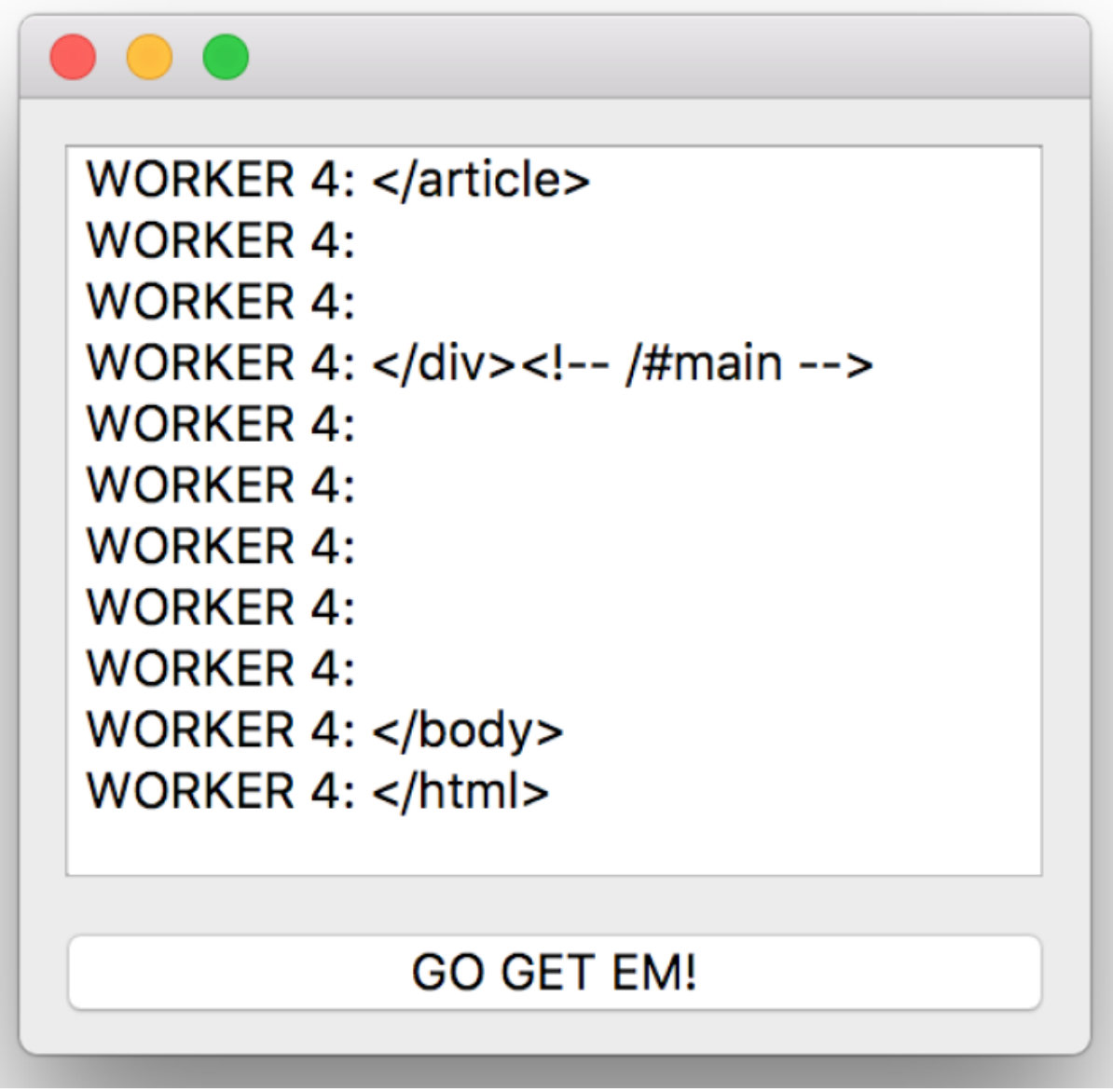


图207：将多个工作线程的输出日志显示在主窗口中

当然，元组是可选的，如果您只有一个运行器，或者不需要将输出与源关联，您可以返回裸字符串。通过适当设置信号，也可以发送字节字符串或任何其他类型的数据。

数据解析

通常，您对线程中的原始数据（无论是来自服务器还是其他外部设备）并不感兴趣，而是希望先对数据进行某种处理。在此示例中，我们创建自定义解析器，这些解析器可以从请求的页面中提取特定数据。我们可以创建多个工作者，每个工作者接收不同的网站列表和解析器。

Listing 189. concurrent/qrunnable_io_parser.py

```
    self.parsers = { #1
                    # 正则表达式解析器，用于从HTML中提取数据。
                    "title": re.compile(
                        r"<title.*?>(.*?)<\/title>", re.M | re.S
                    ),
                    "h1": re.compile(r"<h1.*?>(.*?)<\/h1>", re.M | re.S),
                    "h2": re.compile(r"<h2.*?>(.*?)<\/h2>", re.M | re.S),
                   }
```

1. 解析器被定义为一系列编译后的正则表达式。但您可以按任何方式定义解析器

Listing 190. concurrent/qrunnable_io_parser.py

```
    def execute(self):
        for n, url in enumerate(self.urls):
            worker = Worker(n, url, self.parsers) #1
            worker.signals.data.connect(self.display_output)
            # 执行
            self.threadpool.start(worker)
```

1. 将解析器列表传递给每个工作进程。

Listing 191. concurrent/qrunnable_io_parser.py

```
class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    :param id：该工作线程的 ID
    :param url：要检索的 URL
    """
    def __init__(self, id, url, parsers):
        super().__init__()
        self.id = id
        self.url = url
        self.parsers = parsers

        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        r = requests.get(self.url)

        data = {}
        for name, parser in self.parsers.items(): #1
        m = parser.search(r.text)
        if m: #2
            data[name] = m.group(1).strip()

    self.signals.data.emit((self.id, data))
```

1. 遍历传递给工作线程的解析器列表。对该页面的数据运行每个解析器。
2. 如果正则表达式匹配，将数据添加到我们的数据字典中

运行此代码后，您将看到每个工作进程的输出，其中包含提取的H1、H2和TITLE标签。

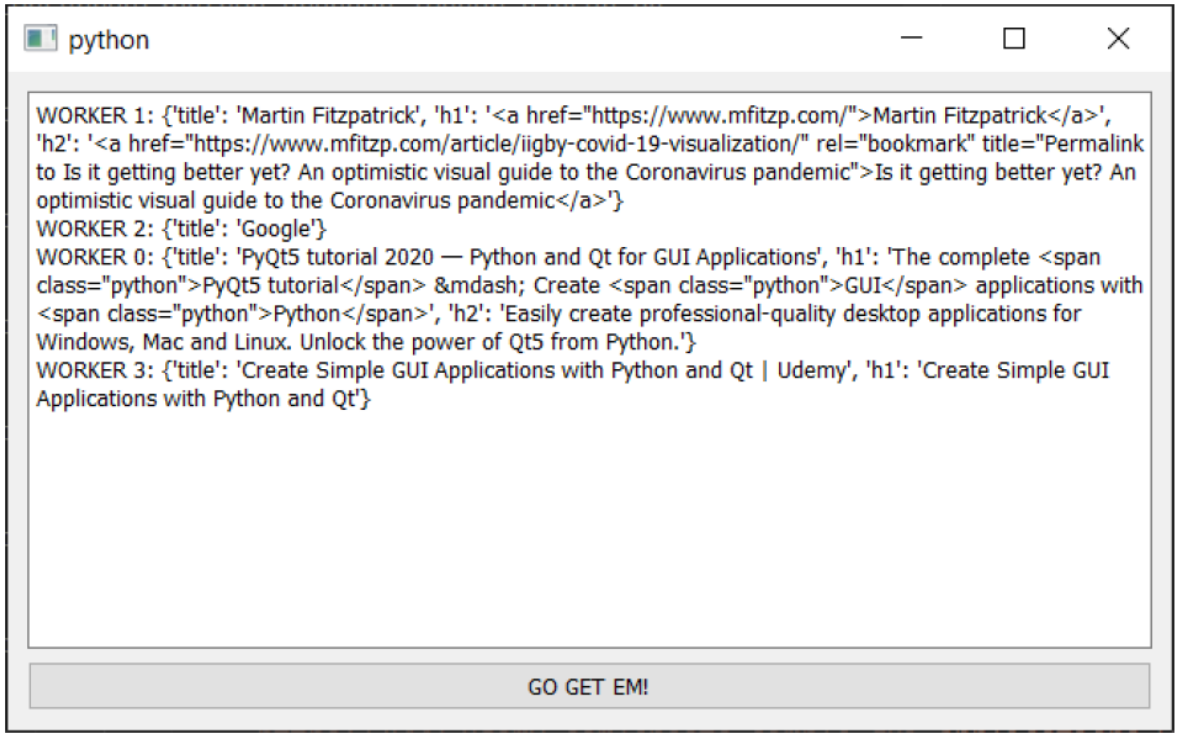


图208：显示多个工作进程解析后的输出结果

如果您正在开发从网站提取数据的工具，建议您使用 BeautifulSoup 4，它比使用正则表达式要强大得多。

通用化

您并不总是能提前知道需要让工作进程执行哪些任务。或者，您可能需要执行多个类似的函数，并希望有一个统一的 API 来运行它们。在这种情况下，您可以利用 Python 中函数是对象这一特性，构建一个通用运行器，该运行器不仅接受参数，还接受要执行的函数。

在下面的示例中，我们创建了一个单一的 Worker 类，然后使用它来运行多个不同的函数。通过这种设置，您可以传入任何 Python 函数，并在单独的线程中执行它。

以下给出了完整的工作示例，展示了自定义的 `QRunnable` 工作以及工作和进度信号。您应该能够将此代码应用于您开发的任何应用程序。

Listing 192. concurrent/qrunnable_generic.py

```
import sys
import time
import traceback

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
```

```
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

def execute_this_fn():
    for _ in range(0, 5):
        time.sleep(1)

    return "Done."


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号
    支持的信号为：
    finished
        无数据
    error
        `元组` (异常类型，值，traceback.format_exc() )
    result
        `对象` 处理后返回的数据，任何类型

    """
    finished = pyqtSignal()
    error = pyqtSignal(tuple)
    result = pyqtSignal(object)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    :param callback: 在此工作线程上运行的回调函数。
    :thread. 提供的 args 和 kwargs 将传递给运行器。
    :type callback: 函数
    :param args: 传递给回调函数的参数
    :param kwargs: 传递给回调函数的关键字参数
    :
    """
    def __init__(self, fn, *args, **kwargs):
        super().__init__()

        # 拆分构造函数参数（用于后续处理）
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()
```

```
    @pyqtSlot()
    def run(self):
        """
        使用传入的参数和关键字参数（args/kwargs）初始化运行器函数。
        """

        # 在此处获取参数（args/kwargs）；并使用它们触发处理流程。
        try:
            result = self.fn(*self.args, **self.kwargs)
        except:
            traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
            self.signals.error.emit(
                (exctype, value, traceback.format_exc())
            )
        else:
            self.signals.result.emit(
                result
            ) # 返回处理结果
        finally:
            self.signals.finished.emit() # 完成


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()
```

```
    def print_output(self, s):
        print(s)

    def thread_complete(self):
        print("THREAD COMPLETE!")

    def oh_no(self):
        # 传递要执行的函数
        worker = Worker(
            execute_this_fn
        ) # 任何其他参数（args）和关键字参数（kwargs）都会传递给run函数。
        worker.signals.result.connect(self.print_output)
        worker.signals.finished.connect(self.thread_complete)

        # 执行
        self.threadpool.start(worker)

    def recurring_timer(self):
        self.counter += 1
        self.l.setText("Counter: %d" % self.counter)


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

通用函数方法增加了一个可能并不明显的限制——运行函数无法访问运行器的 `self` 对象，因此无法访问信号来发出数据本身。我们只能在函数结束时发出函数的返回值。虽然您可以返回一个复合类型（如元组）来返回多个值，但您无法获得进度信号或正在处理的数据。

但是，有一个解决方法。由于您可以将任何内容传递到自定义函数，因此您也可以传递 `self` 或 `self.signals` 对象，以便使用它们。

Listing 193. concurrent/qrunnable_generic_callback.py

```
import sys
import time
import traceback

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    QTimer,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)
```

```
def execute_this_fn(signals):
    for n in range(0, 5):
        time.sleep(1)
        signals.progress.emit(n * 100 / 4)

    return "Done."


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号
    支持的信号为：
    finished
        无数据
    error
        `元组` (异常类型, 值, traceback.format_exc() )
    result
        `对象` 处理后返回的数据，任何类型
    progress
        `整形` indicating % progress

    """
    finished = pyqtSignal()
    error = pyqtSignal(tuple)
    result = pyqtSignal(object)
    progress = pyqtSignal(int)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
    :param callback: 在此工作线程上运行的回调函数。
    :thread. 提供的 args 和 kwargs 将传递给运行器。
    :type callback: 函数
    :param args: 传递给回调函数的参数
    :param kwargs: 传递给回调函数的关键字参数
    :
    """
    def __init__(self, fn, *args, **kwargs):
        super().__init__()
        # 拆分构造函数参数（用于后续处理）
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

        # 将回调函数添加到我们的关键字参数中
        kwargs["signals"] = self.signals

    @pyqtSlot()
     def run(self):
        """
        使用传入的参数和关键字参数（args/kwargs）初始化运行器函数。
```

```
        """

        # 在此处获取参数（args/kwargs）；并使用它们触发处理流程。
        try:
            result = self.fn(*self.args, **self.kwargs)
        except:
            traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
            self.signals.error.emit(
                (exctype, value, traceback.format_exc())
            )
        else:
            self.signals.result.emit(
                result
            ) # 返回处理结果
        finally:
            self.signals.finished.emit() # 完成


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.counter = 0

        layout = QVBoxLayout()

        self.l = QLabel("Start")
        b = QPushButton("DANGER!")
        b.pressed.connect(self.oh_no)

        layout.addWidget(self.l)
        layout.addWidget(b)

        w = QWidget()
        w.setLayout(layout)

        self.setCentralWidget(w)

        self.show()

        self.threadpool = QThreadPool()
        print(
            "Multithreading with maximum %d threads"
            % self.threadpool.maxThreadCount()
        )

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def progress_fn(self, n):
        print("%d%% done" % n)

    def print_output(self, s):
```

```
        print(s)

    def thread_complete(self):
        print("THREAD COMPLETE!")

    def oh_no(self):
        # 传递要执行的函数
        worker = Worker(
            execute_this_fn
        ) # 任何其他参数（args）和关键字参数（kwargs）都会传递给run函数。
        worker.signals.result.connect(self.print_output)
        worker.signals.finished.connect(self.thread_complete)
        worker.signals.progress.connect(self.progress_fn)

        # 执行
        self.threadpool.start(worker)

    def recurring_timer(self):
        self.counter += 1
        self.l.setText("Counter: %d" % self.counter)


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

请注意，要使此功能正常工作，您的自定义函数必须能够接受额外的参数。您可以通过使用 `**kwargs` 定义函数来实现这一点，这样如果额外参数未被使用，它们将被静默地忽略。

```
def execute_this_fn(**kwargs): #1
    for _ in range(0, 5):
        time.sleep(1)

    return "Done."
```

> `signals` 关键字参数被 `**kwargs` 吞噬（掩盖）

运行外部进程

到目前为止，我们已经探讨了如何在另一个线程中运行 Python 代码。然而，有时您需要在另一个进程中运行外部程序——例如命令行程序。

在使用 PyQt6 启动外部进程时，您实际上有多种选择。您可以使用 Python 的内置 `subprocess` 模块来启动进程，或者您可以使用 Qt 的 `QProcess`。

> 有关使用 `QProcess` 运行外部进程的更多信息，请参阅“运行外部命令和过程”章节。

启动新进程总会带来一些执行成本，并会暂时阻塞您的图形用户界面。这通常并不明显，但根据您的使用情况，可能会累积起来，并可能影响性能。您可以通过在另一个线程中启动进程来解决这个问题。

如果您想与进程进行实时通信，则需要一个单独的线程来避免阻塞图形用户界面。 `QProcess` 会在内部为您处理这个单独的线程，但使用Python子进程时，您需要自己完成这项工作。

在这个 `QRunnable` 示例中，我们使用 `worker` 的实例来通过 Python 子进程处理启动外部进程。这使进程的启动成本不会占用图形用户界面的线程，并且允许我们通过 Python 直接与进程交互。

Listing 194. concurrent/qrunnable_process.py

```
import subprocess
import sys

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QPlainTextEdit,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class WorkerSignals(QObject):
    """
    定义了运行中的工作线程可用的信号。
    支持的信号包括：

    finished: 没有数据
    result: str
    """
    result = pyqtSignal(
        str
    ) # 将进程的输出作为字符串返回。.
    finished = pyqtSignal()


class SubProcessWorker(QRunnable):
    """
    ProcessWorker 工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。

    :param command: 使用 `subprocess` 执行的命令.
    """

    def __init__(self, command):
        super().__init__()
        # 存储构造函数参数（用于后续处理）。
```

```
        self.signals = WorkerSignals()
        # 要执行的命令.
        self.command = command

    @pyqtSlot()
    def run(self):
        """
        执行命令，返回结果
        """
        output = subprocess.getoutput(self.command)
        self.signals.result.emit(output)
        self.signals.finished.emit()


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 一些按钮
        layout = QVBoxLayout()

        self.text = QPlainTextEdit()
        layout.addWidget(self.text)

        btn_run = QPushButton("Execute")
        btn_run.clicked.connect(self.start)
        layout.addWidget(btn_run)

        w = QWidget()
        w.setLayout(layout)
        self.setCentralWidget(w)

        # 线程运行器
        self.threadpool = QThreadPool()

        self.show()

    def start(self):
        # 创建一个运行器
        self.runner = SubProcessWorker("python dummy_script.py")
        self.runner.signals.result.connect(self.result)
        self.threadpool.start(self.runner)

    def result(self, s):
        self.text.appendPlainText(s)


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

> 本示例中的“外部程序”是一个简单的 Python 脚本 `python dummy_script.py`。不过，您可以将其替换为任何其他您喜欢的程序。

运行中的进程有两个输出流——标准输出和标准错误。标准输出返回执行过程中的实际结果（如果有的话），而标准错误返回任何错误或日志信息。

在此示例中，我们使用 `subprocess.getoutput` 运行外部脚本。 这会运行外部程序，并在其完成后返回。一旦 程序完成，`getoutput` 会将标准输出和标准错误 一起作为一个字符串返回。

解析结果

您无需直接传递输出结果。如果您需要对命令的输出结果进行后处理，则可以在工作线程中处理该输出结果，以保持其独立性。然后，工作线程可以以结构化的格式将数据返回给图形用户界面线程，以便使用。

在下面的示例中，我们传递了一个函数来后处理示例脚本的结果，将感兴趣的值提取到字典中。这些数据用于更新图形用户界面的控件。

Listing 195. concurrent/qrunnable_process_result.py

```
import subprocess
import sys
from collections import namedtuple

from PyQt6.QtCore import (
    QObject,
    QRunnable,
    QThreadPool,
    pyqtSignal,
    pyqtSlot,
)
from PyQt6.QtWidgets import (
    QApplication,
    QLineEdit,
    QMainWindow,
    QPushButton,
    QSpinBox,
    QVBoxLayout,
    QWidget,
)


def extract_vars(l):
    """
    从行中提取变量，查找包含等号的行，并将其拆分为键值对。
    """
    data = {}
    for s in l.splitlines():
```

```
        if "=" in s:
            name, value = s.split("=")
            data[name] = value

    data["number_of_lines"] = len(l)
    return data


class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号
    支持的信号为:
    finished: 没有数据
    result: 字典
    """

    result = pyqtSignal(dict) # 将输出作为字典返回.
    finished = pyqtSignal()


class SubProcessWorker(QRunnable):
    """
    ProcessWorker 工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。

    :param command: 使用 `subprocess` 执行的命令.
    """

    def __init__(self, command):
        super().__init__()
        # 存储构造函数参数（用于后续处理）。
        self.signals = WorkerSignals()
        # 要执行的命令.
        self.command = command

        # 后处理函数
        self.process_result = process_result

    @pyqtSlot()
    def run(self):
        """
        执行命令，返回结果
        """
        output = subprocess.getoutput(self.command)

        if self.process_result:
            output = self.process_result(output)

        self.signals.result.emit(output)
        self.signals.finished.emit()


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        # 一些按钮
        layout = QVBoxLayout()

        self.name = QLineEdit()
        layout.addWidget(self.name)

        self.country = QLineEdit()
        layout.addWidget(self.country)

        self.website = QLineEdit()
        layout.addWidget(self.website)

        self.number_of_lines = QSpinBox()
        layout.addWidget(self.number_of_lines)

        btn_run = QPushButton("Execute")
        btn_run.clicked.connect(self.start)

        layout.addWidget(btn_run)

        w = QWidget()
        w.setLayout(layout)
        self.setCentralWidget(w)

        # 线程运行器
        self.threadpool = QThreadPool()

        self.show()

    def start(self):
        # 创建一个运行器
        self.runner = SubProcessWorker(
            "python dummy_script.py", process_result=extract_vars
        )
        self.runner.signals.result.connect(self.result)
        self.threadpool.start(self.runner)

    def result(self, data):
        print(data)
        self.name.setText(data["name"])
        self.country.setText(data["country"])
        self.website.setText(data["website"])
        self.number_of_lines.setValue(data["number_of_lines"])


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

在此情况下，简单的解析器会查找包含“=”的任何行，并以此为分隔符分割，以生成名称和值，然后将它们存储在字典中。然而，您可以使用任何您喜欢的工具从字符串输出中提取数据。

由于 `getoutput` 会阻塞直到程序完成，我们无法看到程序的运行情况——例如，获取进度信息。在接下来的示例中，我们将展示如何从正在运行的进程中获取实时输出。

跟踪进度

通常外部程序会将进度信息输出到控制台。您可能希望捕获这些信息，并将其显示给用户，或用于生成一个进度条。

对于执行结果，您通常需要捕获标准输出，对于进度，则需要捕获标准错误。在以下示例中，我们同时捕获两者。除了命令外，我们还向工作传递一个自定义解析函数，以捕获当前工作的进度并将其作为 0-99 之间的数字输出。

这个示例相当复杂。完整的源代码可在随书附带的源代码中找到，但这里我们将重点介绍与较简单版本的关键差异。

Listing 196. concurrent/qrunnable_process_parser.py

```
    @pyqtSlot()
    def run(self):
        """
        使用传入的参数和关键字参数（args/kwargs）初始化运行器函数.
        """
        result = []

        with subprocess.Popen( #1
                              self.command,
                              bufsize=1,
                              stdout=subprocess.PIPE,
                              stderr=subprocess.STDOUT, #2
                              universal_newlines=True,
        ) as proc:
            while proc.poll() is None:
                data = proc.stdout.readline() #3
                result.append(data)
                if self.parser: #4
                    value = self.parser(data)
                    if value:
                        self.signals.progress.emit(value)

        output = "".join(result)

        self.signals.result.emit(output)
```

1. 使用 Popen 运行以获取输出流的访问权限。
2. 我们将标准错误与标准输出一起重定向。
3. 从进程中读取一行（或等待一行）。
4. 将目前收集的所有数据传递给解析器。

解析由这个简单的解析器函数来完成，它接受一个字符串，并匹配正则表达式 ·Total complete: (\d+)%·。

Listing 197. concurrent/qrunnable_process_parser.py

```
progress_re = re.compile("Total complete: (\d+)%")

def simple_percent_parser(output):
    """
    使用 progress_re 正则表达式匹配行，
    返回一个整数表示百分比进度。
    """
    m = progress_re.search(output)
    if m:
        pc_complete = m.group(1)
        return int(pc_complete)
```

解析器与命令一起传递给运行器——这意味着我们可以对所有子进程使用通用运行器，并针对不同的命令以不同的方式处理输出。

Listing 198. concurrent/qrunnable_process_parser.py

```
    def start(self):
        # 创建一个运行器
        self.runner = SubProcessWorker(
            command="python dummy_script.py",
            parser=simple_percent_parser,
        )
        self.runner.signals.result.connect(self.result)
        self.runner.signals.progress.connect(self.progress.setValue)
        self.threadpool.start(self.runner)
```

在这个简单的示例中，我们仅传递进程中的最新一行，因为我们的自定义脚本会输出类似“总完成：25%”的行。这意味着我们只需最新一行即可计算当前进度。

然而，有时脚本可能不太实用。例如，`FFmpeg` 的视频编码器会在处理视频文件时，在开始时输出视频文件的总时长，然后输出当前已处理的时长。要计算进度百分比，您需要这两个值。

要实现这一点，您可以将收集到的输出传递给解析器。在随书附带的源代码中，有一个名为 `concurrent/qrunnable_process_parser_elapsed.py` 的示例，演示了这一过程。

管理器

在之前的示例中，我们创建了多个不同的 `QRunnable` 实现，这些实现可以在应用程序中用于不同目的。在所有情况下，您可以根据需要在同一个或多个 `QThreadPool` 线程池中运行任意数量的这些线程。然而，有时您可能需要跟踪正在运行的线程，以便处理它们的输出，或直接为用户提供对线程的控制权。

`QThreadPool` 本身并不提供访问当前正在运行的线程的接口，因此我们需要自行创建一个管理器，通过该管理器来启动和控制我们的线程。

下面的示例将之前介绍的一些其他线程功能——进度、暂停和停止控制——与模型视图结合起来，以显示个别进度条。这个管理器很可能适用于您运行线程的大多数用例。

> 这是一个相当复杂的示例，完整的源代码可在书的资源中找到。这里我们将依次介绍 `QRunnable` 管理器的关键部分。

工作进程管理器

工作进程管理器类保存线程池、我们的工作进程及其进度和状态信息。它从 `QAbstractListModel` 派生而来，这意味着它也提供了一个类似于 Qt 模型的接口，可以作为 `QListView` 的模型使用，为每个工作进程提供进度条和状态指示器。状态跟踪通过许多内部信号来处理，这些信号会自动附加到每个添加的工作进程上。

Listing 199. concurrent/qrunnable_manager.py

```
class WorkerManager(QAbstractListModel):
    """
    管理器，用于处理我们的工作队列和状态。
    还作为视图的 Qt 数据模型，显示每个工作进程的进度。
    """
    _workers = {}
    _state = {}
    status = pyqtSignal(str)

    def __init__(self):
        super().__init__()

        # 为我们的工作进程创建一个线程池.
        self.threadpool = QThreadPool()
        # self.threadpool.setMaxThreadCount(1)
        self.max_threads = self.threadpool.maxThreadCount()
        print(
            "Multithreading with maximum %d threads" % self
            .max_threads
        )

        self.status_timer = QTimer()
        self.status_timer.setInterval(100)
        self.status_timer.timeout.connect(self.notify_status)
        self.status_timer.start()

    def notify_status(self):
        n_workers = len(self._workers)
        running = min(n_workers, self.max_threads)
        waiting = max(0, n_workers - self.max_threads)
        self.status.emit(
            "{} running, {} waiting, {} threads".format(
                running, waiting, self.max_threads
            )
        )
```

```
    def enqueue(self, worker):
        """
        将一个工作进程加入队列，以便在某个时间点运行，方法是将其传递给 QThreadPool。
        """
        worker.signals.error.connect(self.receive_error)
        worker.signals.status.connect(self.receive_status)
        worker.signals.progress.connect(self.receive_progress)
        worker.signals.finished.connect(self.done)

        self.threadpool.start(worker)
        self._workers[worker.job_id] = worker

        # 将默认状态设置为等待，进度为0.
        self._state[worker.job_id] = DEFAULT_STATE.copy()

        self.layoutChanged.emit()

    def receive_status(self, job_id, status):
        self._state[job_id]["status"] = status
        self.layoutChanged.emit()

    def receive_progress(self, job_id, progress):
        self._state[job_id]["progress"] = progress
        self.layoutChanged.emit()

    def receive_error(self, job_id, message):
        print(job_id, message)

    def done(self, job_id):
        """
        任务/工作进程已完成。将其从活动工作进程字典中移除。
        我们将其保留在工作进程状态中，因为这用于显示过去/已完成的工作进程。
        """
        del self._workers[job_id]
        self.layoutChanged.emit()

    def cleanup(self):
        """
        从 worker_state 中移除所有已完成或失败的工作进程。
        """
        for job_id, s in list(self._state.items()):
            if s["status"] in (STATUS_COMPLETE, STATUS_ERROR):
                del self._state[job_id]
        self.layoutChanged.emit()

    # 模型接口
    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            # 请参见下文的数据结构.
            job_ids = list(self._state.keys())
            job_id = job_ids[index.row()]
            return job_id, self._state[job_id]

    def rowCount(self, index):
        return len(self._state)
```

工作进程在管理器之外构建，并通过 `.enqueue()` 传递进来。这将连接所有信号，并将工作进程添加到线程池中。一旦有线程可用，它就会像正常一样被执行。

工作进程存储在内部字典 `_workers` 中，该字典以任务ID为键。有一个独立的字典 `_state`，用于存储工作进程的状态和进度信息。我们将其分离存储，以便在任务完成后删除任务，保持准确的计数，同时继续显示已完成任务的信息，直到清除

每个提交的工作者的信号都连接到管理器的槽上，这些槽更新 `_state` 字典、打印错误消息或删除已完成的工作。一旦任何状态被更新，我们必须调用 `.layoutChanged()` 来触发模型视图的刷新。`_clear_` 方法遍历 `\_state` 列表，并删除任何已完成或失败的项目。

最后，我们设置了一个定时器，定期触发一个方法，将当前线程数作为状态消息输出。活动线程数是 `_workers` 和 `max_threads` 中的较小值。等待线程数是 `_workers` 减去 `max_threads`（只要大于零）。该消息显示在主窗口的状态栏上。

工作进程

该工作进程本身与我们之前的所有示例遵循相同的模式。我们管理器的唯一要求是添加一个 `.job_id` 属性，该属性在创建工作进程时设置。

工作进程的信号必须包含此工作 ID，以便管理器知道哪个工作进程发送了信号——更新正确的状态、进度和完成状态。

该工作进程本身是一个简单的占位符工作进程，它会迭代100次（每次迭代对应1%的进度），并执行一个简单的计算。该工作进程的计算会生成一系列数字，但其设计会偶尔抛出除以零的错误。

Listing 200. concurrent/qrunnable_manager.py

```
class WorkerSignals(QObject):
    """
    定义运行中的工作线程可用的信号。
    支持的信号为：
    finished
        没有数据

    error
        `元组` (exctype, value, traceback.format_exc() )

    result
        `object` data returned from processing, anything

    progress
        `int` indicating % progress
    """
    error = pyqtSignal(str, str)
    result = pyqtSignal(str, object) # 我们可以返回任何东西.

    finished = pyqtSignal(str)
    progress = pyqtSignal(str, int)
    status = pyqtSignal(str, str)


class Worker(QRunnable):
    """
    工作线程
    从 QRunnable 继承，用于处理工作线程的设置、信号和收尾工作。
```

```
    :param args: 需要传递给工作进程的参数
    :param kwargs: 需要传递给工作进程的关键字

    """

    def __init__(self, *args, **kwargs):
        super().__init__()

        # 存储构造函数参数（用于后续处理）.
        self.signals = WorkerSignals()

        # 为这个任务分配一个唯一的标识符.
        self.job_id = str(uuid.uuid4())

        # 工作进程的参数
        self.args = args
        self.kwargs = kwargs

        self.signals.status.emit(self.job_id, STATUS_WAITING)

    @pyqtSlot()
    def run(self):
        """
        使用传入的参数和关键字参数初始化运行器函数。
        """

        self.signals.status.emit(self.job_id, STATUS_RUNNING)

        x, y = self.args

        try:
            value = random.randint(0, 100) * x
            delay = random.random() / 10
            result = []

            for n in range(100):
                # 生成一些数字.
                value = value / y
                y -= 1

                # 以下情况有时会引发除以零错误
                result.append(value)

                # 分发当前进展情况.
                self.signals.progress.emit(self.job_id, n + 1)
                time.sleep(delay)

        except Exception as e:
            print(e)
            # 我们忽略了这个错误，继续前进.
            self.signals.error.emit(self.job_id, str(e))
            self.signals.status.emit(self.job_id, STATUS_ERROR)

        else:
            self.signals.result.emit(self.job_id, result)
```

```
            self.signals.status.emit(self.job_id, STATUS_COMPLETE)

        self.signals.finished.emit(self.job_id)
```

除了之前看到的进度信号外，我们还有一个状态信号，它会发出以下状态之一。异常被捕获，异常文本和错误状态都会通过错误和状态发出。

Listing 201. concurrent/qrunnable_manager.py

```
STATUS_WAITING = "waiting"
STATUS_RUNNING = "running"
STATUS_ERROR = "error"
STATUS_COMPLETE = "complete"


STATUS_COLORS = {
    STATUS_RUNNING: "#33a02c",
    STATUS_ERROR: "#e31a1c",
    STATUS_COMPLETE: "#b2df8a",
}

DEFAULT_STATE = {"progress": 0, "status": STATUS_WAITING}
```

每个活动状态都分配了颜色，这些颜色将在绘制进度条时使用。

自定义行显示

我们使用 `QListView` 来显示进度条。通常，列表视图会显示每行简单的文本值。为了修改这一点，我们使用 `QItemDelegate`，它允许我们为每行绘制自定义控件。

Listing 202. concurrent/qrunnable_manager.py

```
class ProgressBarDelegate(QStyledItemDelegate):
    def paint(self, painter, option, index):
        # data 是我们状态字典，包含进度、ID 和状态。
        job_id, data = index.model().data(
            index, Qt.ItemDataRole.DisplayRole
        )
        if data["progress"] > 0:
            color = QColor(STATUS_COLORS[data["status"]])

            brush = QBrush()
            brush.setColor(color)
            brush.setStyle(Qt.BrushStyle.SolidPattern)

            width = option.rect.width() * data["progress"] / 100

            rect = QRect(
                option.rect
            ) # rect 的副本，以便我们可以进行修改。.
            rect.setWidth(width)

            painter.fillRect(rect, brush)

        pen = QPen()
```

```
        pen.setColor(Qt.GlobalColor.black)
        painter.drawText(
            option.rect, Qt.AlignmentFlag.AlignLeft, job_id
        )
```

我们从模型中获取当前行的数据，使用 `index.model().data(index,Qt.ItemDataRole.DisplayRole)` 。这调用了 `.data()` 方法，传入索引和角色。在我们的 `.data()` 方法中，我们返回两部分数据——`job_id` 和状态字典，其中包含 `progress` 和 `status` 键。

对于活跃任务（`progress > 0`），状态用于为条形图选择颜色。该条形图以项行大小 `option.rect()` 绘制为矩形，宽度根据完成百分比调整。最后，我们在条形图顶部显示 `job_id` 文本。

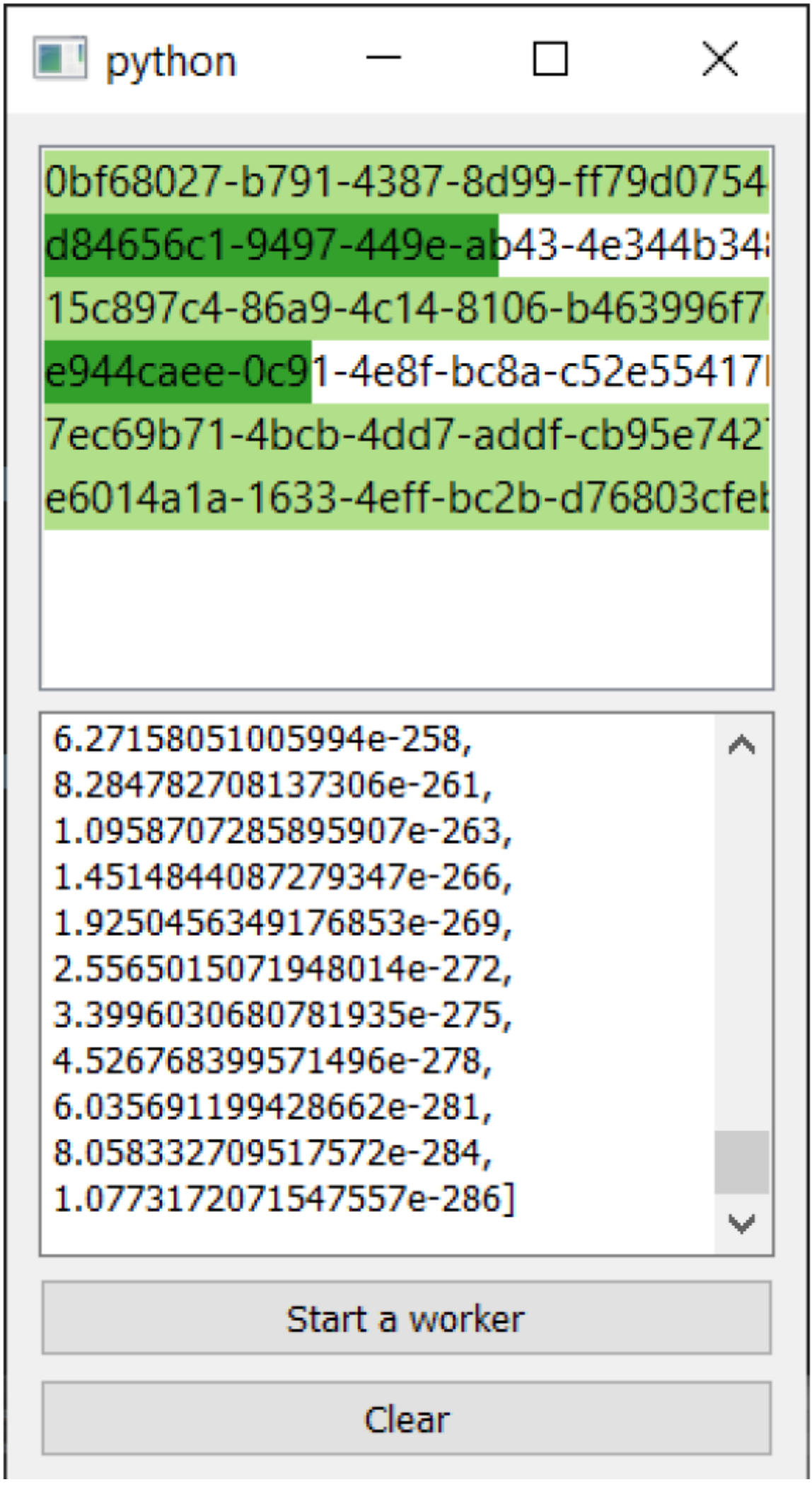
开始一个任务

一切就绪后，我们现在可以通过调用 `.self.worker.enqueue()` 并向工作进程传递参数来排队任务。

Listing 203. concurrent/qrunnable_manager.py

```
    def start_worker(self):
        x = random.randint(0, 1000)
        y = random.randint(0, 1000)

        w = Worker(x, y)
        w.signals.result.connect(self.display_result)
        w.signals.error.connect(self.display_result)

        self.workers.enqueue(w)
```

`.enqueue()` 方法接受一个构造的工作进程，并将内部信号附加到它以跟踪进度。但是，我们仍然可以附加任何其他我们想要的外部信号。


python
0bf68027-b791-4387-8d99-ff79d0754
d84656c1-9497-449e-ab43-4e344b34
15c897c4-86a9-4c14-8106-b463996f7
e944caee-0c91-4e8f-bc8a-c52e55417
7ec69b71-4bcb-4dd7-addf-cb95e742
e6014a1a-1633-4eff-bc2b-d76803cfe
6.27158051005994e-258,
8.28478270813730 6e-261,
1.0958707285895907e-263,
1.4514844087279347e-266,
1.9250456349176853e-269,
2.556501507194801 4e-272,
3.399603068078193 5e-275,
4.526768399571496e-278,
6.035691199428662e-281,
8.058332709517572e-284,
1.077317207154755 7e-286]
Start a worker
Clear

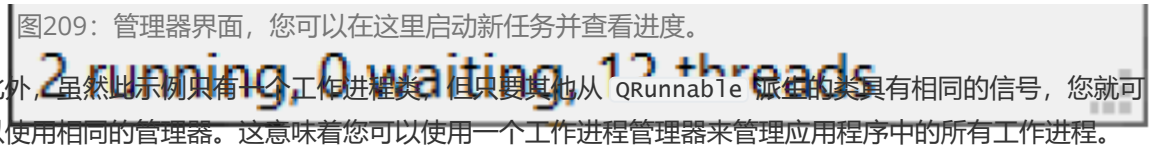


图209：管理器界面，您可以在这里启动新任务并查看进度。

此外，虽然此示例只有十个工作进程类，但只要其他从 `QRunnable` 派生的类具有相同的信号，您就可以使用相同的管理器。这意味着您可以使用一个工作进程管理器来管理应用程序中的所有工作进程。

> 您可以查看本书中的源文件以获取完整代码，并尝试根据需要修改管理器——例如，通用函数运行器尝试添加强制结束和暂停功能

结束任务

我们可以启动任务，其中一些任务可能会因错误而终止。但如果我们想停止那些耗时过长的任务呢？`QListView` 允许我们选择行，并通过选中的行终止特定的工作进程。下面的方法与一个按钮关联，并从列表中当前选中的项中查找工作进程。

Listing 204. concurrent/qrunnable_manager_stop.py

```
    def stop_worker(self):
        selected = self.progress.selectedIndexes()
        for idx in selected:
            job_id, _ = self.workers.data(
                idx, Qt.ItemDataRole.DisplayRole
            )
            self.workers.kill(job_id)
```

除此之外，我们还需要修改委托以绘制当前选中的项，并更新工作进程和管理器以传递强制结束信号。请查看此示例的完整源代码，了解它们是如何配合在一起的。

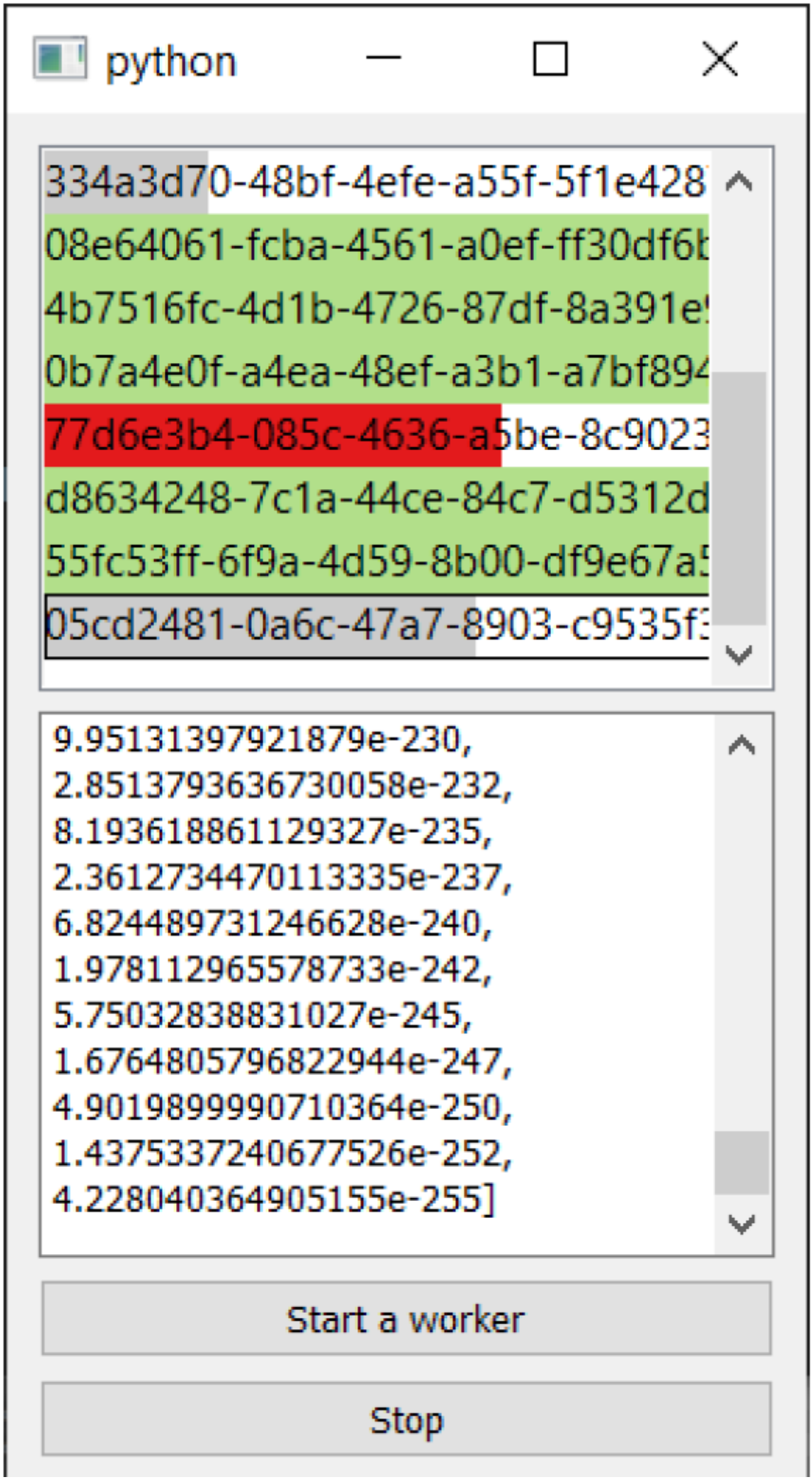
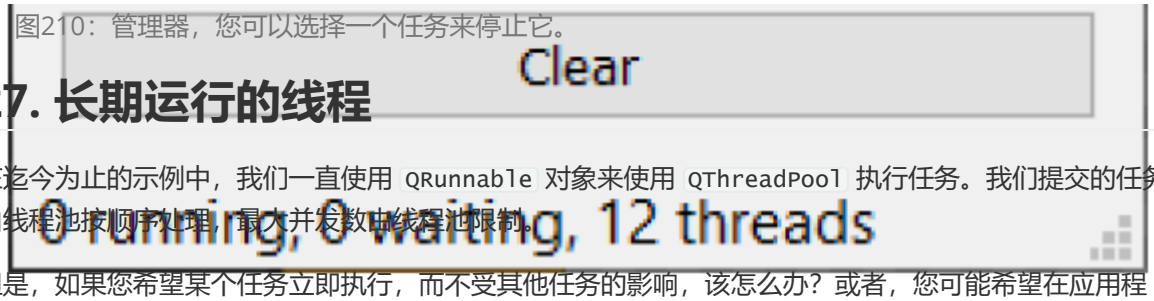

python
334a3d70-48bf-4efe-a55f-5f1e428
08e64061-fcba-4561-a0ef-ff30df6
4b7516fc-4d1b-4726-87df-8a391e
0b7a4e0f-a4ea-48ef-a3b1-a7bf894
77d6e3b4-085c-4636-a5be-8c9023
d8634248-7c1a-44ce-84c7-d5312d
55fc53ff-6f9a-4d59-8b00-df9e67a
05cd2481-0a6c-47a7-8903-c9535f
9.95131397921879e-230,
2.8513793636730058e-232,
8.193618861129327e-235,
2.3612734470113335e-237,
6.824489731246628e-240,
1.97811296557873e-242,
5.75032838831027e-245,
1.6764805796822944e-247,
4.9019899990710364e-250,
1.437533724067752e-252,
4.228040364905155e-255]
Start a worker
Stop

图210：管理器，您可以选择一个任务来停止它。



27. 长期运行的线程

在迄今为止的示例中，我们一直使用 `QRunnable` 对象来使用 `QThreadPool` 执行任务。我们提交的任务由线程池按顺序处理，最大并发数由线程池限制。

但是，如果您希望某个任务立即执行，而不受其他任务的影响，该怎么办？或者，您可能希望在应用程序运行期间始终在后台保持一个线程运行，以与某个远程服务或硬件交互，或传输数据进行处理。在这种情况下，线程池架构可能并不合适。

在本章中，我们将探讨 PyQt6 的持久线程接口 `QThread`。它与您已经见过的 `QRunnable` 对象提供了非常相似的接口，但允许您完全控制线程的运行时间和方式。

使用 `QThread`

与 `QRunnable` 示例一样，`QThread` 类充当了您想要在另一个线程中执行的代码的包装器。它负责启动和将工作转移到单独的线程，以及在线程完成后进行管理和关闭。您只需提供要执行的代码即可。这可以通过子类化 `QThread` 并实现 `run()` 方法来完成。

一个简单的线程

让我们从一个简单的例子开始。下面，我们实现了一个工作线程，它可以为我们执行算术运算。我们为该线程添加了一个信号，我们可以使用它将数据从线程中发送出去。

Listing 205. concurrent/qthread_1.py

```
import sys
import time

from PyQt6.QtCore import QThread, pyqtSignal, pyqtSlot
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow


class Thread(QThread):
    """
    工作线程
    """

    result = pyqtSignal(str) #1

    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中
        """
        print("Thread start")
        counter = 0
        while True:
            time.sleep(0.1)
            # 将数字以格式化字符串的形式输出.
            self.result.emit(f"The number is {counter}")
            counter += 1
        print("Thread complete")
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 创建线程并启动它.
        self.thread = Thread()
        self.thread.start() #2

        label = QLabel("Output will appear here")

        # 连接信号，这样输出就会出现在标签上.
        self.thread.result.connect(label.setText)

        self.setCentralWidget(label)
        self.show()


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

1. 与 `QRunnable` 不同，`QThread` 类继承自 `QObject`，因此我们可以在线程对象本身定义信号。
2. 调用 `.start()` 而不是 `.run()` 来启动线程！

运行此示例后，您将在窗口中看到一个向上计数的数字。这看起来并不

这真的太令人兴奋啦！但计数是在与图形用户界面独立的线程中进行的，结果是通过信号输出的。这意味着图形用户界面不会被正在进行的工作阻塞（尽管正常的 Python GIL 规则仍然适用）。

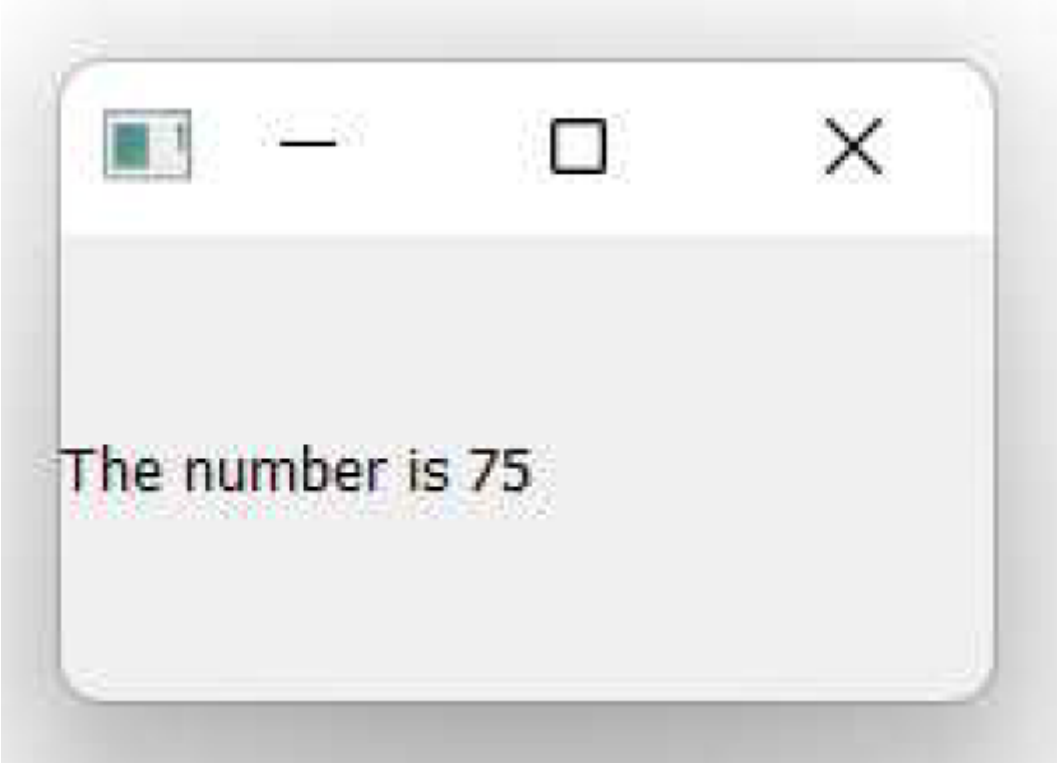


图211：通过信号显示结果的 QThread 计数器

您可以尝试增加 `sleep()` 调用的持续时间，您应该会发现，即使线程被阻塞，主图形用户界面仍然正常运行。

> 如果您通常使用 `numpy` 或其他库，可以尝试使用它们在线程中进行更复杂的计算。

> 通常您会希望在线程中添加某种信号以实现通信。

线程控制

现在我们可以启动线程，但无法停止它。与 `QRunnable` 不同，`QThread` 类内置了 `.terminate()` 方法，可用于立即终止正在运行的线程。这并非干净的关闭操作——线程将直接停止当前执行位置，且不会抛出 Python 异常。

Listing 206. concurrent/qthread_2.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 创建线程并启动它.
        self.thread = Thread()
        self.thread.start()

        label = QLabel("Output will appear here")
        button = QPushButton("Kill thread")
        # 终止（立即杀死）线程.
        button.pressed.connect(self.thread.terminate)

        # 连接信号，这样输出就会出现在标签上.
        self.thread.result.connect(label.setText)
        container = QWidget()
        layout = QVBoxLayout()
        layout.addWidget(label)
        layout.addWidget(button)
        container.setLayout(layout)

        self.setCentralWidget(container)
        self.show()
```

如果您运行这个程序，您会发现我们在线程主循环后添加的“线程完成”消息从未显示。这是因为当我们调用 `.terminate()` 时，执行过程会立即停止，并且永远不会到达代码中的那个位置。

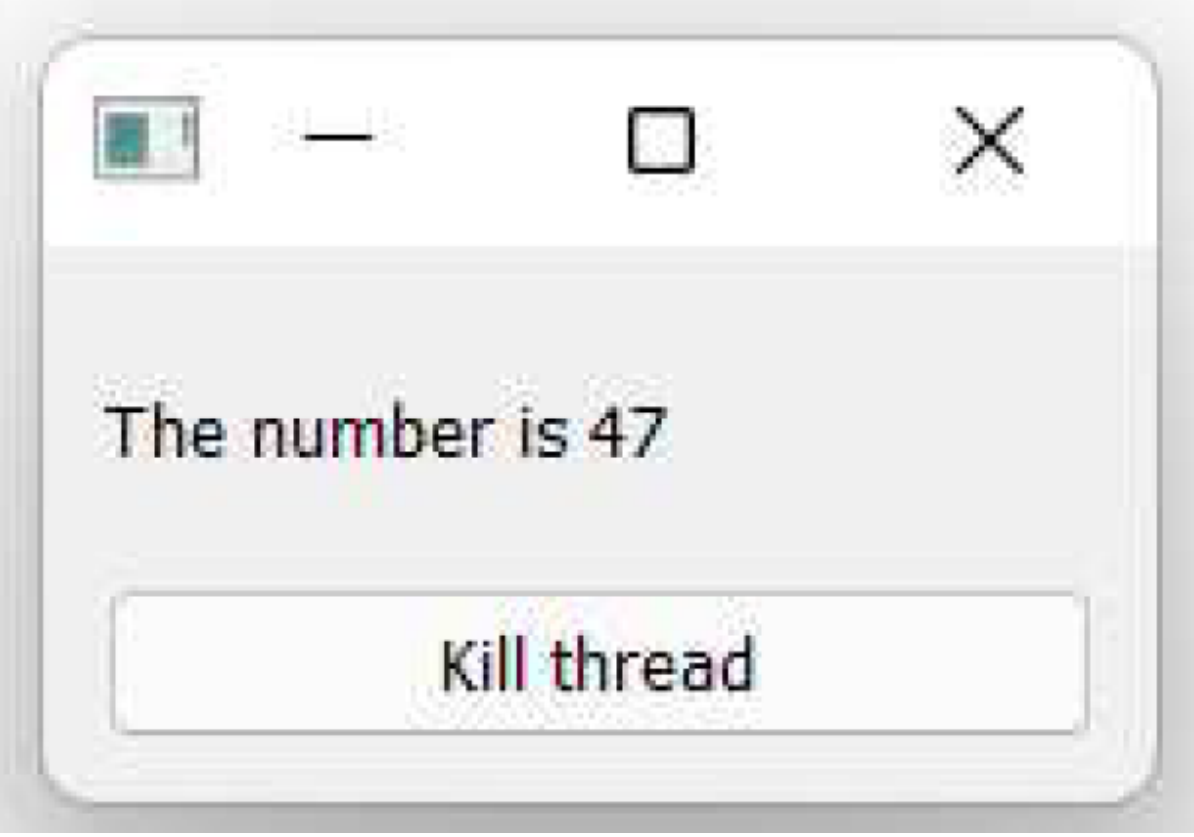


图212：可以通过按钮控制来终止该线程。

然而，`QThread` 有一个完成信号，可用于在线程完成后触发某些动作。无论线程是终止还是正常关闭，该信号都会被触发。

线程对象在线程完成运行后仍会保留，您可以使用它来查询线程状态。然而，请注意——如果线程被终止，与线程对象交互可能会导致您的应用程序崩溃。下面的示例通过尝试在线程被终止后打印一些关于线程对象的信息来演示这一点。

Listing 207. concurrent/qthread_2b.py

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 创建线程并启动它.
        self.thread = Thread()
        self.thread.start()

        label = QLabel("Output will appear here")
        button = QPushButton("Kill thread")
        # 终止（立即杀死）线程.
        button.pressed.connect(self.thread.terminate)

        # 连接信号，这样输出就会出现在标签上.
        self.thread.result.connect(label.setText)
        self.thread.finished.connect(self.thread_has_finished) #1

        container = QWidget()
        layout = QVBoxLayout()
        layout.addWidget(label)
        layout.addWidget(button)
        container.setLayout(layout)
```

```
        self.setCentralWidget(container)
        self.show()

    def thread_has_finished(self):
        print("Thread has finished.")
        print(
            self.thread,
            self.thread.isRunning(),
            self.thread.isFinished(),
        ) #2
```

1. 将完成的信号连接到我们的自定义槽。
2. 如果您终止线程，您的应用程序很可能会在此处崩溃。

虽然您可以从内部终止一个线程，但更干净利落的做法是从 `run()` 方法中返回。一旦您退出 `run()` 方法，线程就会自动结束并被安全地清理，并且完成信号会触发。

Listing 208. concurrent/qthread_2c.py

```
class Thread(QThread):
    """
    工作线程
    """

    result = pyqtSignal(str) #1

    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中
        """
        print("Thread start")
        counter = 0
        while True:
            time.sleep(0.1)
            # 将数字以格式化字符串的形式输出.
            self.result.emit(f"The number is {counter}")
            counter += 1
            if counter > 50:
                return #1
```

1. 在 `run()` 方法中调用 `return` 将退出并终止线程。

当您运行上述示例时，计数器将在 50 处停止，因为我们从 run() 方法返回。如果在此之后尝试按下终止按钮，请注意，您不会再次收到线程完成信号——线程已经关闭，因此无法终止。

发送数据

在前一个示例中，我们的线程正在运行，但无法接收任何来自外部的数据。通常，当您使用长时间运行的线程时，您会希望能够与它们进行通信，无论是为了传递工作，还是以其他方式控制它们的行为。

我们一直在讨论如何干净地关闭线程的重要性。那么，让我们先看看如何与线程通信，告诉它我们希望它关闭。与 `QRunnable` 示例类似，我们可以使用线程中的一个内部标志来控制主循环，只要该标志为 `True`，循环就会继续。

要关闭线程，我们需要修改这个标志的值。下面我们使用一个名为 `is_running` 的标志和自定义方法 `.stop()` 来实现这一点。当调用这个方法时，它会将 `is_running` 标志设置为 `False`。当标志设置为 `False` 时，主循环将结束，线程将退出 `run()` 方法，并关闭。

Listing 209. concurrent/qthread_3.py

```
class Thread(QThread):
    """
    工作线程
    """
    result = pyqtSignal(str)

    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中。
        """
        self.data = None
        self.is_running = True
        print("Thread start")
        counter = 0
        while self.is_running:
            time.sleep(0.1)
            # 将数字以格式化字符串的形式输出.
            self.result.emit(f"The number is {counter}")
            counter += 1

    def stop(self):
        self.is_running = False
```

然后我们可以修改按钮，使其调用自定义的 `stop()` 方法，而不是

Listing 210. concurrent/qthread_3.py

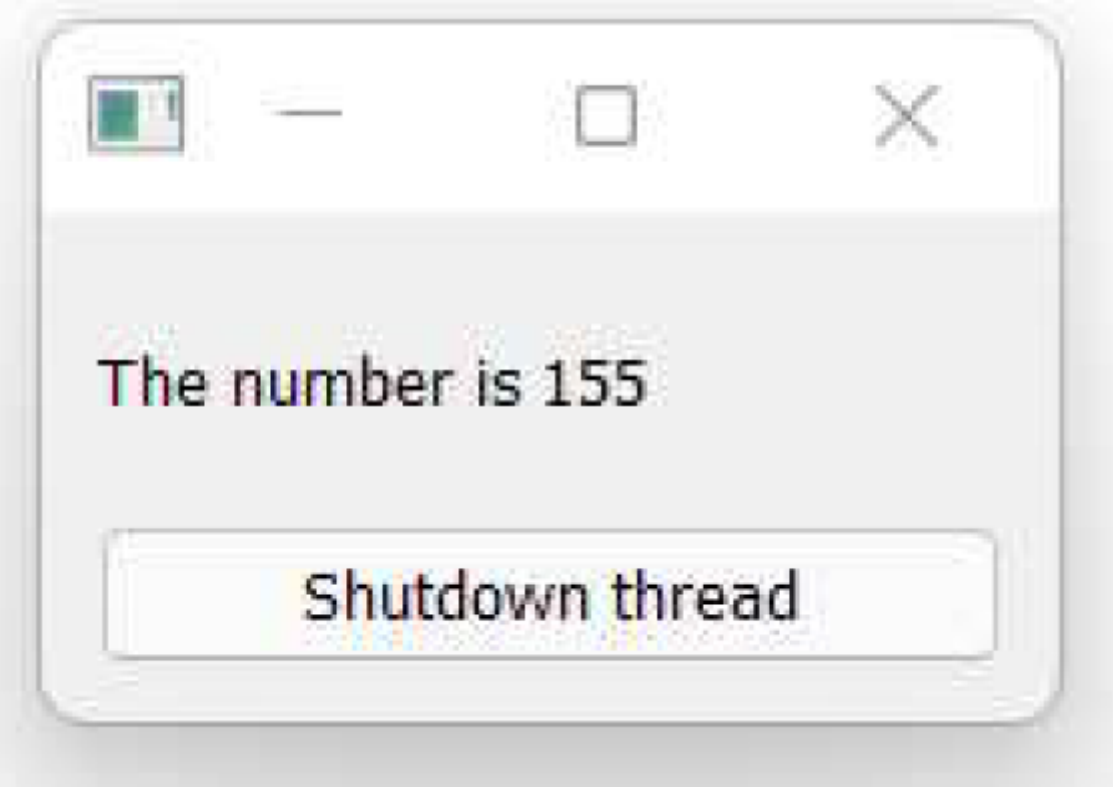
```
        button = QPushButton("Shutdown thread")
        # 优雅地关闭线程.
        button.pressed.connect(self.thread.stop)
```

由于线程已干净地关闭，我们可以安全地访问线程对象，而无需担心它会崩溃。请您将打印语句重新添加到 `thread_has_finished` 方法中。

Listing 211. concurrent/qthread_3.py

```
    def thread_has_finished(self):
        print("Thread has finished.")
        print(
            self.thread,
            self.thread.isRunning(),
            self.thread.isFinished(),
        )
```

如果您运行这个程序，您应该会看到数字像以前一样继续计数，但按下“停止”按钮会立即终止线程。请注意，我们在线程关闭后仍然能够显示该线程的元数据，因为该线程并未发生崩溃。



> 图213：现在可以使用按钮干净地关闭该线程。

我们可以使用相同的基本方法将任何数据发送到我们想要的线程中。下面我们扩展了自定义的 `Thread` 类，添加了一个 `send_data` 方法，该方法接受一个参数，并通过 `self` 将其内部存储在线程中。

使用此方法，我们可以向线程的 `run()` 方法中发送数据，并在该方法中访问这些数据，从而修改线程的行为。

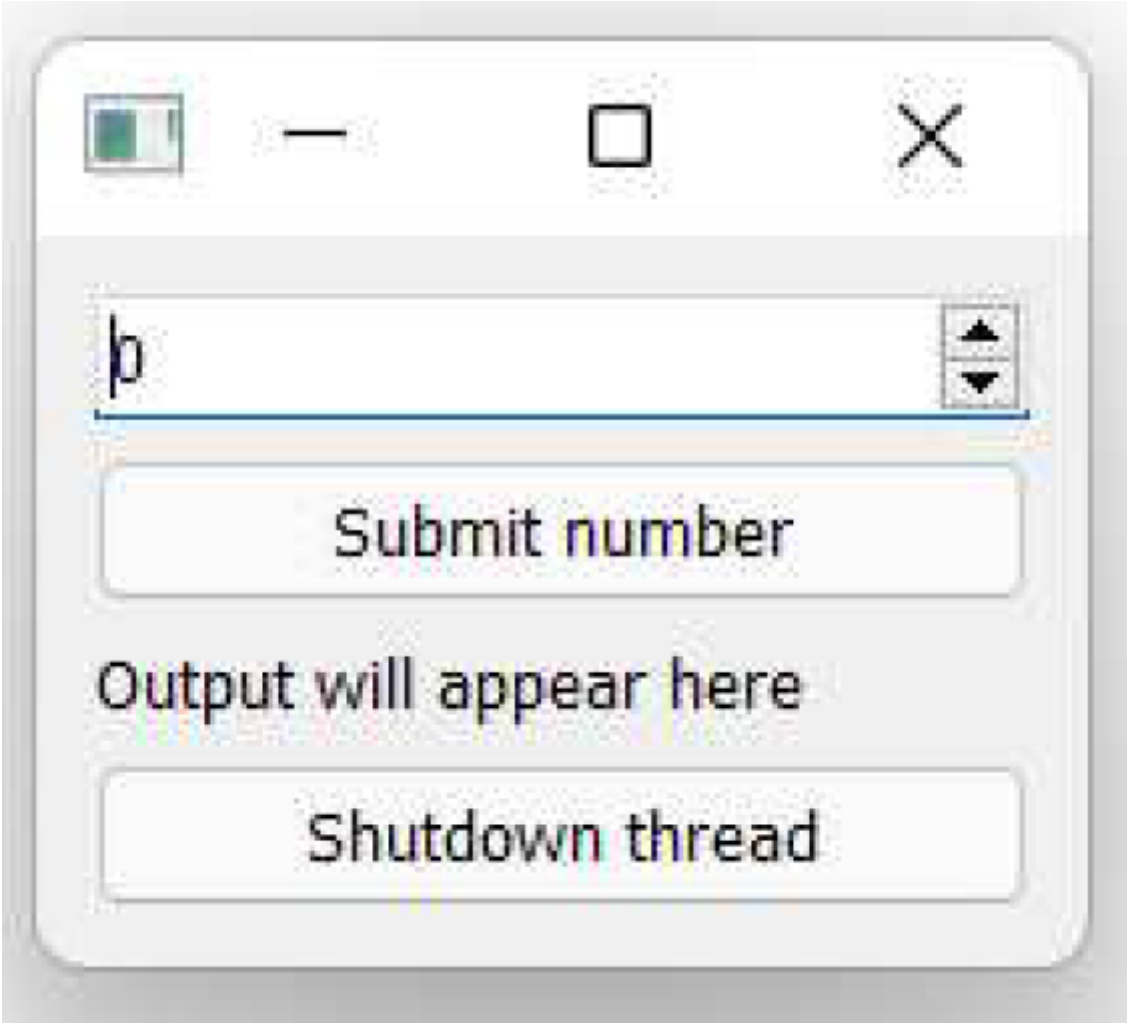
Listing 212. concurrent/qthread_4.py

```
import sys
import time

from PyQt6.QtCore import QThread, pyqtSignal, pyqtSlot
from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QMainWindow,
    QPushButton,
    QSpinBox,
    QVBoxLayout,
    QWidget,
)


class Thread(QThread):
    """
    工作线程
    """
    result = pyqtSignal(str)
```

```
    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中。
        """
        self.data = None
        self.is_running = True
        print("Thread start")
        counter = 0
        while self.is_running:
            while self.data is None:
                time.sleep(0.1) # 等待数据 <1>.
            # 将数字以格式化字符串的形式输出.
            counter += self.data
            self.result.emit(f"The cumulative total is {counter}")
            self.data = None

    def send_data(self, data):
        """
        将数据接收至内部变量
        """
        self.data = data

    def stop(self):
        self.is_running = False


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        # 创建线程并启动它.
        self.thread = Thread()
        self.thread.start()

        self.numeric_input = QSpinBox()
        button_input = QPushButton("Submit number")

        label = QLabel("Output will appear here")

        button_stop = QPushButton("Shutdown thread")
        # 优雅地关闭线程.
        button_stop.pressed.connect(self.thread.stop)

        # 连接信号，这样它的输出就会出现在标签上.
        button_input.pressed.connect(self.submit_data)
        self.thread.result.connect(label.setText)
        self.thread.finished.connect(self.thread_has_finished)

        container = QWidget()
        layout = QVBoxLayout()
        layout.addWidget(self.numeric_input)
        layout.addWidget(button_input)
        layout.addWidget(label)
        layout.addWidget(button_stop)
        container.setLayout(layout)
```

```
        self.setCentralWidget(container)
        self.show()

    def submit_data(self):
        # 将数字输入控件中的值提交给线程
        self.thread.send_data(self.numeric_input.value())

    def thread_has_finished(self):
        print("Thread has finished.")


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

如果您运行这个示例，您会看到以下窗口。您可以使用 `QSpinBox` 选择一个数字，然后点击按钮将其提交给线程。线程将把传入的数字加到当前计数器上并返回结果。



> 图214：现在，我们可以使用 `QSpinBox` 和按钮向我们的线程提交数据。

如果您使用"关闭线程"按钮来停止线程，您可能会注意到一些奇怪的地方。线程确实会关闭，但你可以在它关闭之前再提交一个数字，而计算仍然会进行——试试看！这是因为 `is_running` 检查是在循环的顶部进行的，然后线程会等待输入。

要解决这个问题，我们需要将对 `is_running` 标志的检查移至等待循环中。

Listing 213. concurrent/qthread_4b.py

```
    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中。
        """
        print("Thread start")
        self.data = None
        self.is_running = True
        counter = 0
        while True:
            while self.data is None:
                if not self.is_running:
                    return # Exit thread.
                time.sleep(0.1) # 等待数据 <1>.

                # 将数字以格式化字符串的形式输出.
                counter += self.data
                self.result.emit(f"The cumulative total is {counter}")
                self.data = None
```

如果您现在运行这个示例，您会发现，如果在线程等待时按下按钮，它将立即退出。

> 在您的线程中设置线程退出控制条件时，请务必谨慎，以避免任何意外的副作用。在执行任何新任务/计算之前，以及在输出任何数据之前，请务必进行检查。

通常，您还希望传递一些初始状态数据，例如用于控制后续线程运行的配置选项。我们可以像处理 `QRunnable` 一样，通过在 `__init__` 块中添加参数来传递这些数据。提供的参数必须存储在 `self` 对象中，以便在 `run()` 方法中使用。

Listing 214. concurrent/qthread_5.py

```
class Thread(QThread):
    """
    工作线程
    """
    result = pyqtSignal(str)
    def __init__(self, initial_data):
        super().__init__()
        self.data = initial_data


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        # 创建线程并启动它
```

```
        self.thread = Thread(500)
        self.thread.start()
        # ...
```

使用这两种方法，您可以向线程提供所需的任何数据。在线程中等待数据（使用 `sleep` 循环）、处理数据并通过信号返回数据的模式是 Qt 应用程序中处理长期运行的线程时最常见的模式。

让我们再扩展一个示例，以演示传递多种数据类型。在这个示例中，我们修改线程以使用一个显式锁，名为 `waiting_for_data`，我们可以将其在 `True` 和 `False` 之间切换。您可以使用这个

Listing 215. concurrent/qthread_6.py

```
class Thread(QThread):
    """
    工作线程
    """
    result = pyqtSignal(str)
    def __init__(self, initial_counter):
        super().__init__()
        self.counter = initial_counter


    @pyqtSlot()
    def run(self):
        """
        您的代码应放置在此方法中。
        """
        print("Thread start")
        self.is_running = True
        self.waiting_for_data = True
        while True:
            while self.waiting_for_data:
                if not self.is_running:
                    return # Exit thread.
                time.sleep(0.1) # 等待数据 <1>.

            # 将数字以格式化字符串的形式输出.
            self.counter += self.input_add
            self.counter *= self.input_multiply
            self.result.emit(f"The cumulative total is {self.counter}
")
            self.waiting_for_data = True

    def send_data(self, add, multiply):
        """
        将数据接收至内部变量
        """
        self.input_add = add
        self.input_multiply = multiply
        self.waiting_for_data = False

    def stop(self):
        self.is_running = False
```

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        # 创建线程并启动它.
        self.thread = Thread(500)
        self.thread.start()

        self.add_input = QSpinBox()
        self.mult_input = QSpinBox()
        button_input = QPushButton("Submit number")

        label = QLabel("Output will appear here")

        button_stop = QPushButton("Shutdown thread")
        # 优雅地关闭线程.
        button_stop.pressed.connect(self.thread.stop)

        # 连接信号，这样它的输出就会出现在标签上.
        button_input.pressed.connect(self.submit_data)
        self.thread.result.connect(label.setText)
        self.thread.finished.connect(self.thread_has_finished)

        container = QWidget()
        layout = QVBoxLayout()
        layout.addWidget(self.add_input)
        layout.addWidget(self.mult_input)
        layout.addWidget(button_input)
        layout.addWidget(label)
        layout.addWidget(button_stop)
        container.setLayout(layout)

        self.setCentralWidget(container)
        self.show()

    def submit_data(self):
        # 将数字输入控件中的值提交给线程
        self.thread.send_data(
            self.add_input.value(), self.mult_input.value()
        )

    def thread_has_finished(self):
        print("Thread has finished.")


app = QApplication(sys.argv)
window = MainWindow()
app.exec()
```

您还可以将提交数据的方法拆分为每个值的独立方法，并实现一个显式的计算方法来释放锁。这种方法特别适合在您不需要始终更新所有值的情况下使用。例如，当您从外部服务或硬件读取数据时。

Listing 216. concurrent/qthread_6b.py

```
class Thread(QThread):
    def send_add(self, add):
```

```
        self.input_add = add

    def send_multiply(self, multiply):
        self.input_multiply = multiply

    def calculate(self):
        self.waiting_for_data = False # 解锁并计算.
class MainWindow(QMainWindow):
    def submit_data(self):
        # 将数字输入控件中的值提交给线程.
        self.thread.send_add(self.add_input.value())
        self.thread.send_multiply(self.mult_input.value())
        self.thread.calculate()
```

如果您运行这个示例，您应该会看到与之前完全相同的行为。哪种方法您你的应用程序中最有意义，将取决于该线程正在做什么。

> 不要害怕混合使用您学到的各种线程技术。例如，在某些应用程序中，使用持久线程运行应用程序的某些部分，而使用线程池运行其他部分是有意义的。

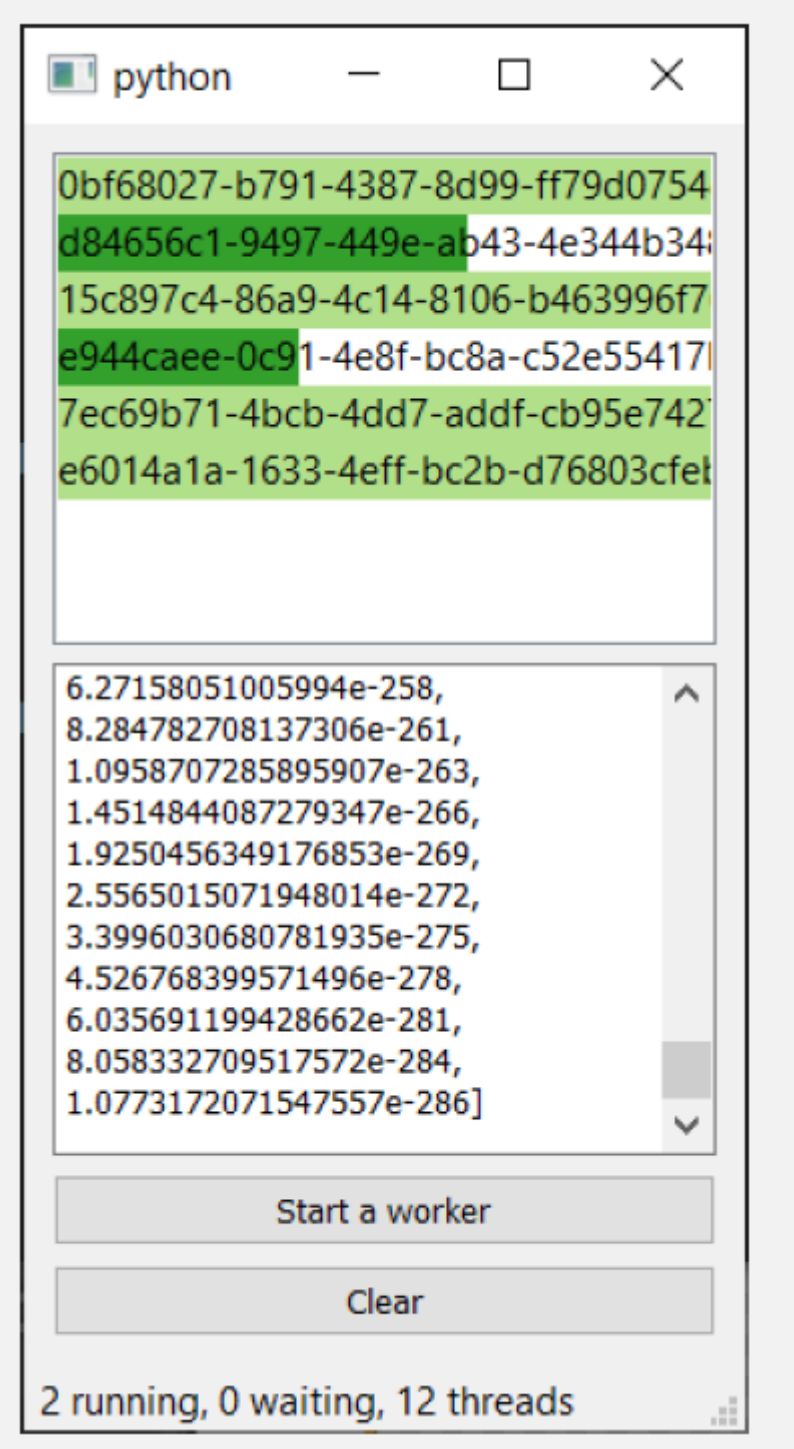
使用过程的感受

当用户在应用程序中执行某项操作时，该操作的后果应立即显现——无论是通过操作本身的结果，还是通过某种指示，表明正在进行的操作将产生相应结果。这一点对于耗时较长的任务（如计算或网络请求）尤为重要，因为缺乏反馈可能导致用户反复点击按钮却得不到任何响应。

> 旋转图标或加载图标可以用于以下情况：当任务的持续时间未知或非常短时。

一种简单的方法是在操作被触发后禁用按钮。但没有其他指示器时，这看起来就像是故障。更好的替代方案是更新按钮，显示“正在处理”的提示，并添加一个活跃的进度指示器，如附近的旋转图标。

进度条是一种常见的方法，用于向用户显示当前正在进行的操作，以及预计需要多长时间。但不要陷入认为进度条总是有用的陷阱！它们只应在能够直观展示任务的线性进展时使用。



> 一些复杂的应用程序可能包含多个并发任务

进度条如果出现以下情况则毫无帮助：

- 进度条会倒退或前进
- 进度条的增长并非与进度成线性关系
- 进度条完成得太快

如果没有这些提示，可能会比完全没有信息更令人沮丧。这些行为可能会让用户感到事情不对劲，从而导致沮丧和困惑——“我错过了哪个对话框？！”这些都不是让用户感到愉快的体验，因此应尽可能避免。

请记住，您的用户并不知道应用程序内部发生了什么——他们唯一的了解渠道是您提供的数据。分享对用户有帮助的数据，并隐藏其他所有内容。如果您需要调试输出，可以将其放在菜单后面。

请务必为耗时较长的任务提供进度条。

请务必在适当情况下提供子任务的详细信息。

请务必在可能的情况下估算任务所需时间。

不要假设用户知道哪些任务耗时长或短。

不要使用上下移动或不规则移动的进度条。

28. 运行外部命令及进程

到目前为止，我们已经探讨了如何在单独的线程中运行程序，包括使用Python的 `subprocess` 模块来运行外部程序。但在PyQt6中，我们还可以利用基于Qt的系统来运行外部程序，即`QProcess`。使用 `QProcess` 创建并执行任务相对简单。

最简单的示例如下所示——我们创建一个 `QProcess` 对象，然后调用 `.start` 方法，传入要执行的命令和一个字符串参数列表。在此示例中，我们正在运行自定义演示脚本，使用 Python 命令：`python dummy_script.py`。

```
p = QProcess()
p.start("python", ["dummy_script.py"])
```

> 根据您的环境，您可能需要指定 `python3` 而不是 `python`

> 您需要在 `QProcess` 实例运行期间，将其引用保存在 `self` 或其他位置。

如果您只是想运行一个程序，而不关心它会发生什么，那么这个简单的例子就足够了。但是，如果您想更多地了解程序在做什么，`QProcess` 提供了一些信号，可以用来跟踪进程的进度和状态。

最有用的事件是 `.readyReadStandardOutput` 和 `.readyReadStandardError`，这些事件会在进程中标准输出和标准错误准备好被读取时触发。所有运行的进程都有两个输出流——标准输出和标准错误。标准输出返回执行结果（如果有），而标准错误返回任何错误或异常。

```
p = QProcess()
p.readyReadStandardOutput.connect(self.handle_stdout)
p.readyReadStandardError.connect(self.handle_stderr)
p.stateChanged.connect(self.handle_state)
p.finished.connect(self.cleanup)
p.start("python", ["dummy_script.py"])
```

此外，还有一个在进程完成时触发的 `.finished` 信号，以及一个在进程状态发生变化时触发的 `.stateChanged` 信号。有效值（在 `QProcess.ProcessState` 枚举中定义）如下所示。

常量	值	描述
QProcess.NotRunning	0	该进程未运行
QProcess.Starting	1	进程已启动，但程序尚未被调用
QProcess.Running	2	该进程正在运行，并已准备好进行读写操作

在下面的示例中，我们将这个基本的 `QProcess` 设置扩展，为标准输出和标准错误添加处理程序。通知数据可用的信号连接到这些处理程序，并使用 `.readAllStandardError()` 和 `.readAllStandardOutput()` 触发对进程数据的请求。

> 这些方法输出原始字节，因此您需要先对其进行解码。

在此示例中，我们的演示脚本 `dummy_script.py` 返回一系列字符串，这些字符串会被解析以提供进度信息和结构化数据。过程的状态也会显示在状态栏上。

完整的代码如下所示：

Listing 217. concurrent/qprocess.py

```
import re
import sys

from PyQt6.QtCore import QProcess
from PyQt6.QtWidgets import (
    QApplication,
    QMainWindow,
    QPlainTextEdit,
    QProgressBar,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

STATES = {
    QProcess.ProcessState.NotRunning: "Not running",
    QProcess.ProcessState.Starting: "Starting...",
    QProcess.ProcessState.Running: "Running...",
}

progress_re = re.compile("Total complete: (\d+)%")


def simple_percent_parser(output):
```

```
    """
    使用 progress_re 正则表达式匹配行，
    返回一个整数表示百分比进度。
    """
    m = progress_re.search(output)
    if m:
        pc_complete = m.group(1)
        return int(pc_complete)


def extract_vars(l):
    """
    从行中提取变量，查找包含等号的行，并拆分为键值对。
    """
    data = {}
    for s in l.splitlines():
        if "=" in s:
            name, value = s.split("=")
            data[name] = value
    return data


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # 保持进程引用.
        self.p = None

        layout = QVBoxLayout()

        self.text = QPlainTextEdit()
        layout.addWidget(self.text)

        self.progress = QProgressBar()
        layout.addWidget(self.progress)

        btn_run = QPushButton("Execute")
        btn_run.clicked.connect(self.start)

        layout.addWidget(btn_run)

        w = QWidget()
        w.setLayout(layout)
        self.setCentralWidget(w)

        self.show()

    def start(self):
        if self.p is not None:
            return

        self.p = QProcess()
        self.p.readyReadStandardOutput.connect(self.handle_stdout)
        self.p.readyReadStandardError.connect(self.handle_stderr)
        self.p.stateChanged.connect(self.handle_state)
```

```
        self.p.finished.connect(self.cleanup)
        self.p.start("python", ["dummy_script.py"])

    def handle_stderr(self):
        result = bytes(self.p.readAllStandardError()).decode("utf8")
        progress = simple_percent_parser(result)

        self.progress.setValue(progress)

    def handle_stdout(self):
        result = bytes(self.p.readAllStandardOutput()).decode("utf8")
        data = extract_vars(result)

        self.text.appendPlainText(str(data))

    def handle_state(self, state):
        self.statusBar().showMessage(STATES[state])

    def cleanup(self):
        self.p = None


app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

在此示例中，我们将进程的引用存储在 `self.p` 中，这意味着我们一次只能运行一个进程。但您可以自由地与应用程序一起运行任意多个进程。如果您不需要跟踪来自这些进程的信息，您可以简单地将进程的引用存储在列表中。

但是，如果您想跟踪进度并单独解析工作进程的输出，您可能需要考虑创建一个管理类来滑块和跟踪所有进程。本书的源文件中有一个示例，名为 `qprocess_manager.py`。

示例的完整源代码可在本书的源代码中找到，但下面我们将重点探讨 `JobManager` 类本身。

Listing 218. concurrent/qprocess_manager.py

```
class JobManager(QAbstractListModel):
    """
    管理器，用于处理活动作业、标准输出、标准错误和进度解析器。
    还作为视图的 Qt 数据模型，显示每个进程的进度。
    """
    _jobs = {}
    _state = {}
    _parsers = {}

    status = pyqtSignal(str)
    result = pyqtSignal(str, object)
    progress = pyqtSignal(str, int)

    def __init__(self):
        super().__init__()

        self.status_timer = QTimer()
        self.status_timer.setInterval(100)
```

```
        self.status_timer.timeout.connect(self.notify_status)
        self.status_timer.start()

        # 内部信号，通过解析器触发进度更新.
        self.progress.connect(self.handle_progress)

    def notify_status(self):
        n_jobs = len(self._jobs)
        self.status.emit("{} jobs".format(n_jobs))

    def execute(self, command, arguments, parsers=None):
        """
        通过启动一个新进程来执行命令
        """
        job_id = uuid.uuid4().hex

        # 默认情况下，信号无法访问发送它的进程的任何信息。因此，我们使用此构造函数为每个信号添加
job_id 注释。

        def fwd_signal(target):
            return lambda *args: target(job_id, *args)

        self._parsers[job_id] = parsers or []

        # 将默认状态设置为等待，进度为0。
        self._state[job_id] = DEFAULT_STATE.copy()

        p = QProcess()
        p.readyReadStandardOutput.connect(
            fwd_signal(self.handle_output)
        )
        p.readyReadStandardError.connect(fwd_signal(self.
                                                    handle_output))
        p.stateChanged.connect(fwd_signal(self.handle_state))
        p.finished.connect(fwd_signal(self.done))

        self._jobs[job_id] = p

        p.start(command, arguments)

        self.layoutChanged.emit()

    def handle_output(self, job_id):
        p = self._jobs[job_id]
        stderr = bytes(p.readAllStandardError()).decode("utf8")
        stdout = bytes(p.readAllStandardOutput()).decode("utf8")
        output = stderr + stdout

        parsers = self._parsers.get(job_id)
        for parser, signal_name in parsers:
            # 依次使用每个解析器对数据进行解析.
            result = parser(output)
            if result:
                # 按名称（使用 signal_name）查找信号，并输出解析结果.
                signal = getattr(self, signal_name)
                signal.emit(job_id, result)
```

```
    def handle_progress(self, job_id, progress):
        self._state[job_id]["progress"] = progress
        self.layoutChanged.emit()

    def handle_state(self, job_id, state):
        self._state[job_id]["status"] = state
        self.layoutChanged.emit()

    def done(self, job_id, exit_code, exit_status):
        """
        任务/工作进程已完成。将其从活动工作者字典中移除。
        我们将其保留在工作进程状态中，因为这用于显示过去/已完成的工作进程。
        """
        del self._jobs[job_id]
        self.layoutChanged.emit()

    def cleanup(self):
        """
        从 worker_state 中移除所有已完成或失败的任务。
        """
        for job_id, s in list(self._state.items()):
            if s["status"] == QProcess.ProcessState.NotRunning:
                del self._state[job_id]
        self.layoutChanged.emit()

    # 模型接口
    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            # 请参见下文的数据结构.
            job_ids = list(self._state.keys())
            job_id = job_ids[index.row()]
            return job_id, self._state[job_id]

    def rowCount(self, index):
        return len(self._state)
```

本类提供了一个模型视图接口，使其可作为 `QListView` 的基础。自定义委托 `ProgressBarDelegate` 委托为每个项绘制进度条，并显示任务标识符。进度条的颜色由进程状态决定——若处于活动状态则为深绿色，若已完成则为浅绿色。

在此设置中，解析来自工作进程的进度信息比较棘手，因为 `.readyReadStandardError` 和 `.readyReadStandardOutput` 信号不会传递数据或关于已准备就绪的工作的信息。为了解决这个问题，我们定义了自定义的 `job_id`，并拦截信号以将此数据添加到它们中。

解析器在执行命令时被传递进来并存储在 `_parsers` 中。每个任务接收的输出会通过相应的解析器处理，用于输出数据或更新任务的进度。我们定义了两个简单的解析器：一个用于提取当前进度，另一个用于获取输出数据。

Listing 219. concurrent/qprocess_manager.py

```
progress_re = re.compile("Total complete: (\d+)%", re.M)

def simple_percent_parser(output):
    """
```

```
    使用 progress_re 正则表达式匹配行，
    返回一个整数表示百分比进度。
    """
    m = progress_re.search(output)
    if m:
        pc_complete = m.group(1)
        return int(pc_complete)

def extract_vars(l):
    """
    从行中提取变量，查找包含等号的行，并拆分为键值对。
    """
    data = {}
    for s in l.splitlines():
        if "=" in s:
            name, value = s.split("=")
            data[name] = value
    return data
```

解析器作为一个简单的元组列表传递，该列表包含用作解析器的函数和要发出的信号的名称。信号通过在 `JobManager` 上使用 `getattr` 根据名称进行查找。在示例中，我们只定义了 2 个信号，一个用于数据/结果输出，另一个用于进度。但您可以根据需要添加任意数量的信号和解析器。使用这种方法，您可以根据需要选择省略某些任务的某些解析器（例如，没有进度信息的情况下）。

您可以运行示例代码，并在另一个进程中运行任务。您可以启动多个任务，并观察它们的完成情况，同时实时更新其当前进度。尝试为自己的任务添加额外的命令和解析器。

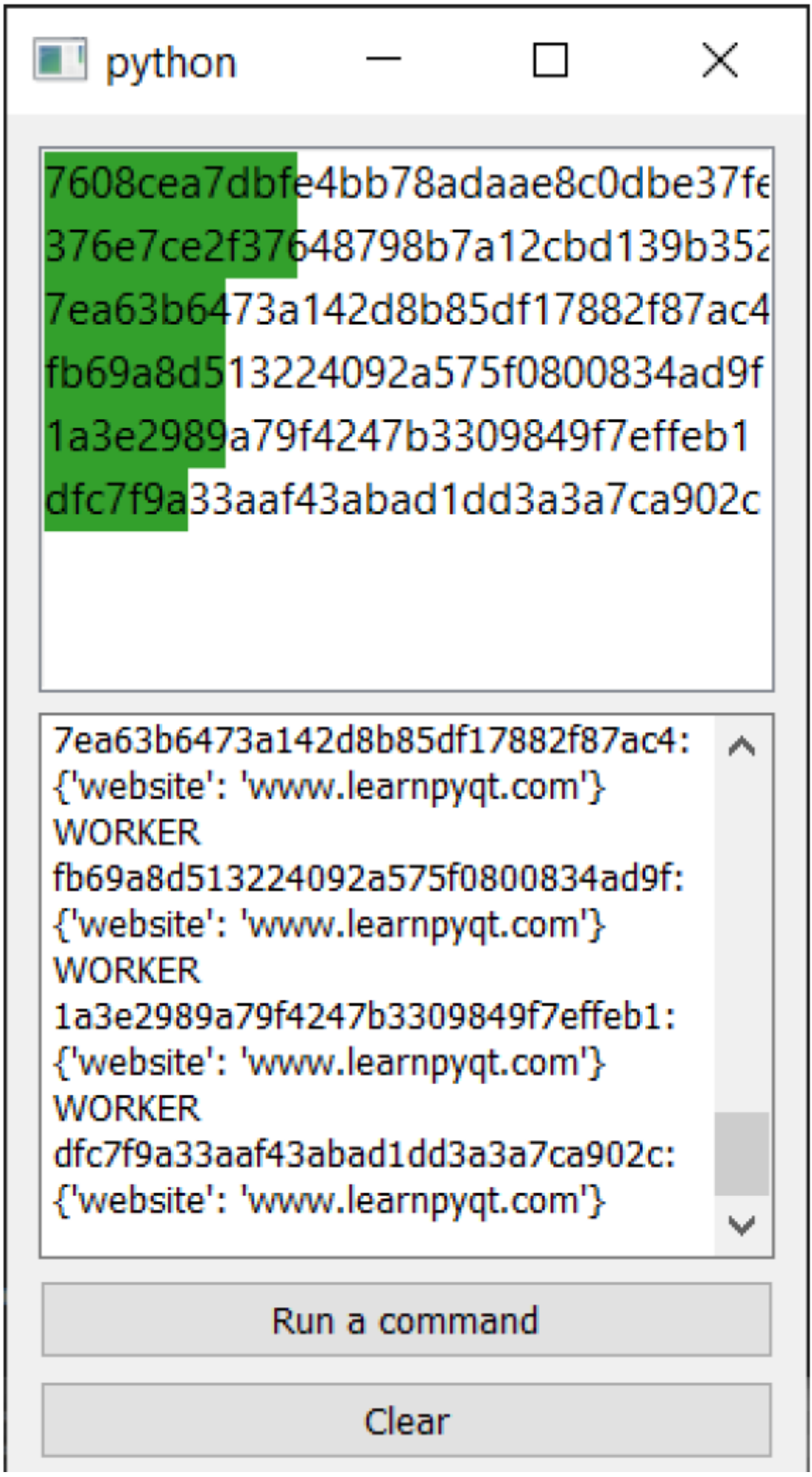

python
7608cea7dbfe4bb78adaae8c0dbe37fe
376e7ce2f37648798b7a12cbd139b352
7ea63b6473a142d8b85df17882f87ac4
fb69a8d513224092a575f0800834ad9f
1a3e2989a79f4247b3309849f7effeb1
dfc7f9a33aaf43abad1dd3a3a7ca902c
7ea63b6473a142d8b85df17882f87ac4: {'website': 'www.learnpyqt.com'}
WORKER fb69a8d513224092a575f0800834ad9f: {'website': 'www.learnpyqt.com'}
WORKER 1a3e2989a79f4247b3309849f7effeb1: {'website': 'www.learnpyqt.com'}
WORKER dfc7f9a33aaf43abad1dd3a3a7ca902c: {'website': 'www.learnpyqt.com'}
Run a command
Clear

图215：进程管理器，显示正在运行的进程和进度

数据可视化

Python 的主要优势之一在于数据科学和可视化，它使用 Pandas、numpy 和 sklearn 等工具进行数据分析。使用 PyQt6 构建图形用户界面应用程序，您可以直接从应用程序中访问所有这些 Python 工具，从而构建复杂的数据驱动型应用程序和交互式仪表板。我们已经介绍了模型视图，它允许我们以列表和表格的形式显示数据。在本章中，我们将探讨这个难题的最后一块拼图——可视化数据。

在使用 PyQt6 开发应用程序时，您有两个主要选择——matplotlib（它也提供对 Pandas 图表的访问权限）和 PyQtGraph，后者使用 Qtnative 图形创建图表。在本章中，我们将探讨如何利用这些库在您的应用程序中可视化数据。

29. 使用 PyQtGraph 进行数据可视化

虽然您可以在 PyQt6 中嵌入 `matplotlib` 图表，但使用体验并不完全原生。对于简单且高度交互的绘图，您可能需要考虑使用 PyQtGraph 代替。PyQtGraph 基于 PyQt6 本机 `QGraphicsScene` 构建，可提供更好的绘图性能，特别是对于实时数据，同时提供交互性，并能够使用 Qt 图形控件轻松自定义绘图。

在本章中，我们将介绍使用 PyQtGraph 创建绘图控件的第一步，然后演示如何使用线条颜色、线条类型、轴标签、背景颜色和绘制多条线条来定制绘图。

开始使用

要使用 PyQtGraph 与 PyQt6，您首先需要将该包安装到您的 Python 环境中。您可以使用 `pip` 进行安装。

撰写本文时，PyQt6 还非常新，因此您需要使用 PyQtGraph 的开发者安装版本。

```
pip install git+https://github.com/pyqtgraph/pyqtgraph@master
```

安装完成后，您应该能够像往常一样导入该模块。

创建 PyQtGraph 控件

在 PyQtGraph 中，所有图都是使用 `PlotWidget` 控件创建的。该控件提供了一个包含的画布，可以在上面添加和配置任何类型的图。在后台，该图控件使用 Qt 本地的 `QGraphicsScene`，这意味着它快速、高效且易于与应用程序的其他部分集成。您可以像创建其他控件一样创建 `PlotWidget`

以下是基本模板应用程序的示例，该应用程序在一个 `QMainWindow` 中包含一个 `PlotWidget`。

> 在以下示例中，我们将创建 PyQtGraph 控件。但是，您也可以从 Qt Designer 中嵌入 PyQtGraph 控件。

Listing 220. plotting/pyqtgraph_1.py

```python
import sys

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        temperature = [30, 32, 34, 32, 33, 31, 29, 32, 35, 45]

        # 绘制数据：x、y 值
        self.graphWidget.plot(hour, temperature)


app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()
```

> 在以下所有示例中，我们使用 `import pyqtgraph as pg` 导入 PyQtGraph。这是 PyQtGraph示例中常见的约定，旨在保持代码整洁并减少重复输入。如果您更喜欢，也可以使用 `import pyqtgraph` 进行导入。

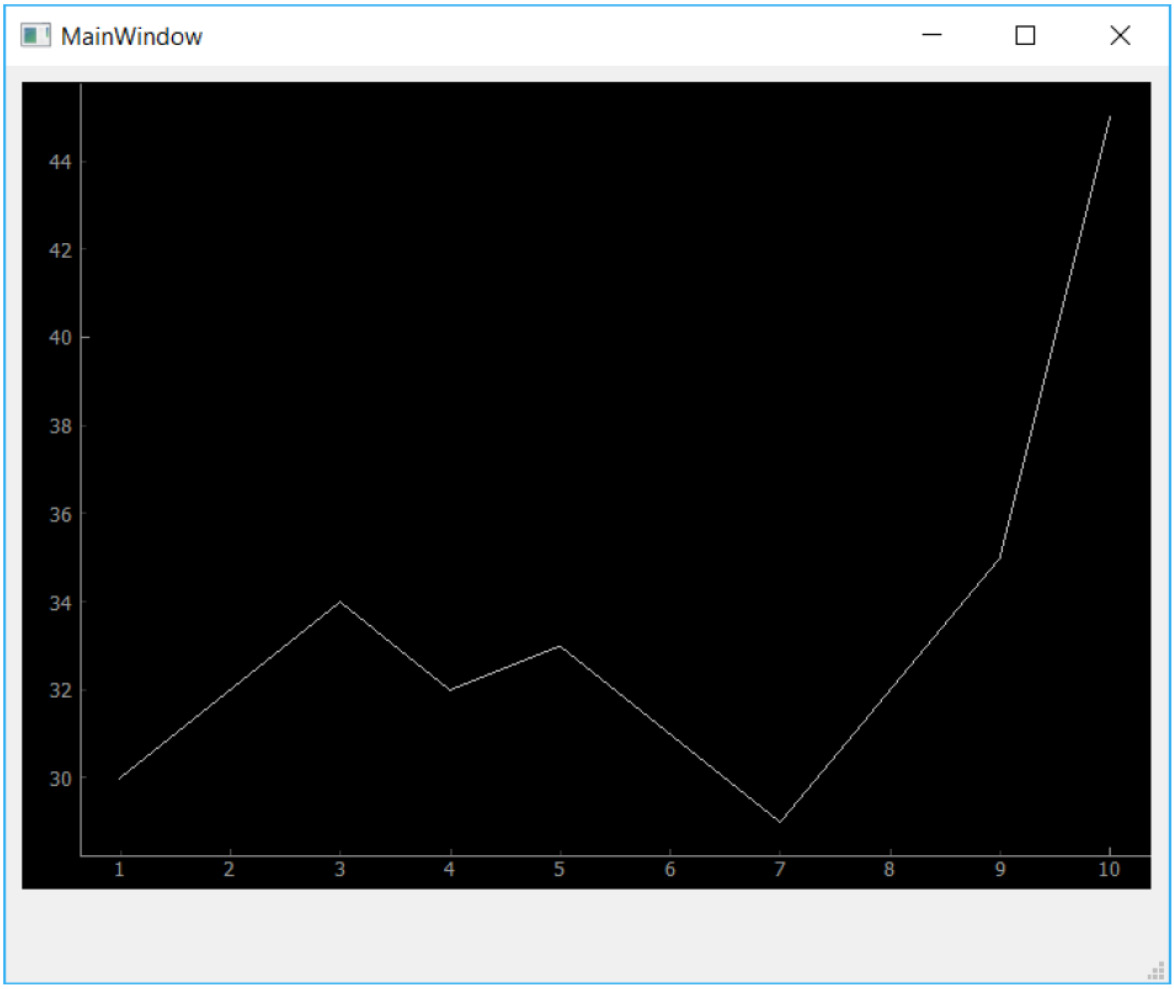


图216：显示虚拟数据的自定义 PyQtGraph 控件。

PyQtGraph 的默认绘图样式非常简单——黑色背景，一条细细的（几乎看不见的）白色线条。在下一节中，我们将看看 PyQtGraph 中有哪些可用的选项，以改善绘图的外观和可用性。

样式绘制

PyQtGraph 使用 Qt 的 `QGraphicsScene` 来绘制图形。这使我们能够访问所有标准的 Qt 线条和形状样式选项，以用于绘制图。然而，PyQtGraph 提供了一个 API，用于绘制图和管理图画布。

下面我们将介绍创建和自定义自己的图所需的最常见的样式功能。

背景颜色

从上面的应用程序骨架开始，我们可以更改背景颜色，方法是调用 `PlotWidget` 实例（在 `self.graphWidget` 中）的 `.setBackground` 方法。下面的代码将背景设置为白色，方法是传入字符串 'w'。

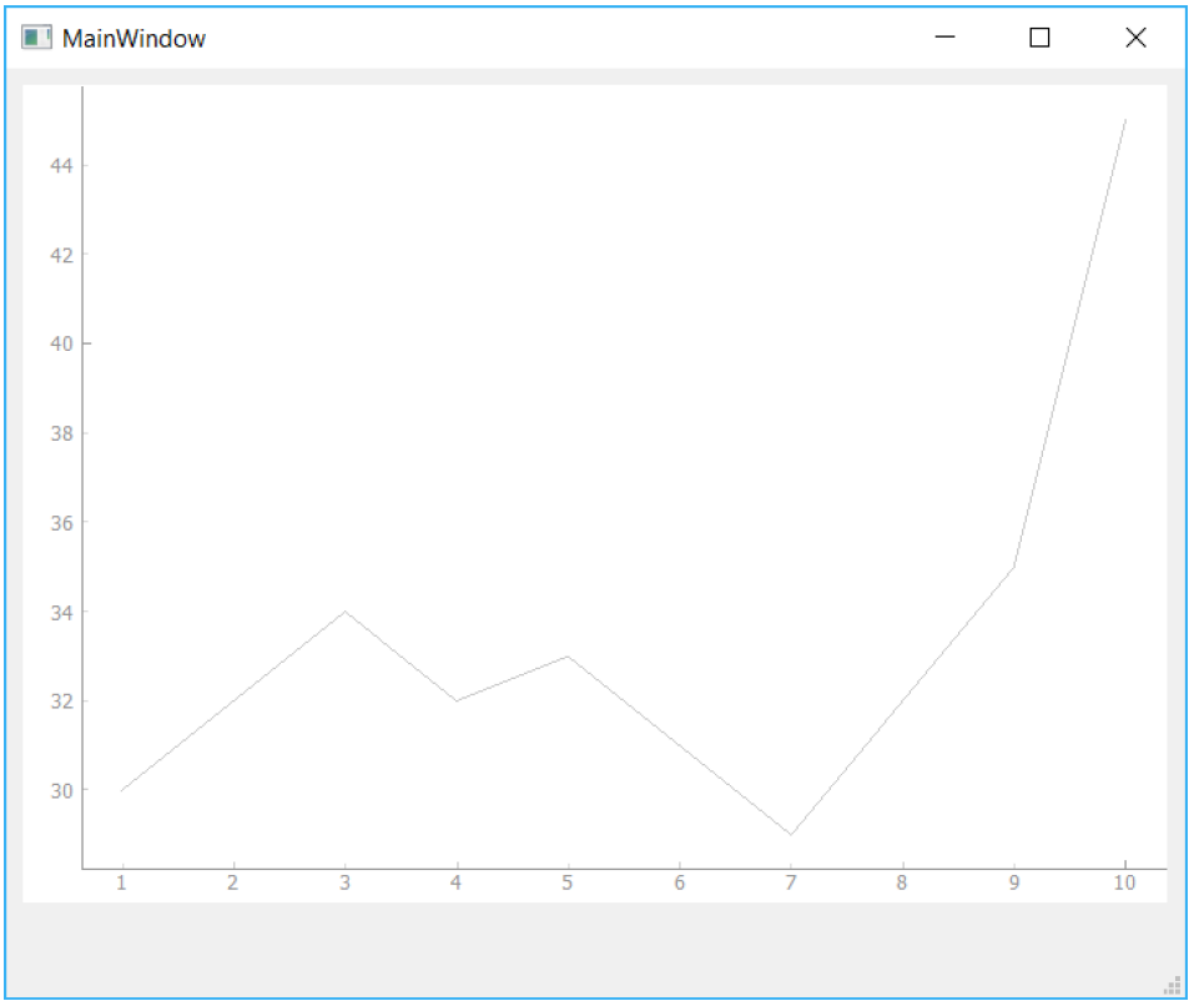
```
self.graphWidget.setBackground('w')
```

您可以随时设置（并更新）图表的背景颜色。

Listing 221. plotting/pyqtgraph_2.py

```
import sys

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph
```

```
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        temperature = [30, 32, 34, 32, 33, 31, 29, 32, 35, 45]

        self.graphWidget.setBackground("w")
        self.graphWidget.plot(hour, temperature)


app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()
```



> 图217：将PyQtGraph的绘图背景改为白色

使用单个字母可以生成多种简单颜色，这些颜色基于 `Matplotlib` 中使用的标准颜色。它们大多较为常见，唯一例外的是'k'用于表示黑色。

Table 7. Common color codes

颜色	字母代号
blue	b
green	g
red	r
cyan (明亮的蓝绿色)	c
magenta (亮粉色)	m
yellow	y
black	k
white	w

除了这些单字母代码外，您还可以使用十六进制表示法设置颜色例如，使用字符串 `#672922`。

```
self.graphWidget.setBackground('#bbccaa') # 十六进制
```

RGB 和 RGBA 值可以分别作为 3 元组或 4 元组传递，使用0-255 的值。

```
self.graphWidget.setBackground((100,50,255)) # RGB each 0-255
self.graphWidget.setBackground((100,50,255,25)) # RGBA (A = alpha opacity)
```

最后，您还可以直接使用Qt的 `QColor` 类型来指定颜色。

```
self.graphWidget.setBackground(QtGui.QColor(100,50,254,25))
```

如果您在应用程序的其他地方使用特定的 `QColor` 对象，或者将图形的背景设置为图形用户界面的默认背景颜色，此功能非常有用。

```
color = self.palette().color(QtGui.QPalette.Window) # 获取默认窗口背景，
self.graphWidget.setBackground(color)
```

线条颜色、宽度和样式

PyQtGraph 中的线条使用标准的 Qt `QPen` 类型绘制。这使您能够像在任何其他 `QGraphicsScene` 绘图中一样，对线条绘制拥有完全的控制权。要使用笔来绘制线条，您只需创建一个新的 `QPen` 实例，并将其传递给 `plot` 方法。

下面我们创建一个 `QPen` 对象，传入一个包含三个整数值的元组，指定一个RGB 值（全红）。我们也可以通过传入 'r' 或一个 `QColor` 对象来定义它。然后我们将此对象传入 `plot` 函数的 pen 参数中。

```
pen = pg.mkPen(color=(255, 0, 0))
self.graphWidget.plot(hour, temperature, pen=pen)
```

完整的代码如下所示：

```
import sys
```

```
from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        temperature = [30, 32, 34, 32, 33, 31, 29, 32, 35, 45]

        self.graphWidget.setBackground("w")

        pen = pg.mkPen(color=(255, 0, 0))
        self.graphWidget.plot(hour, temperature, pen=pen)


app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()
```

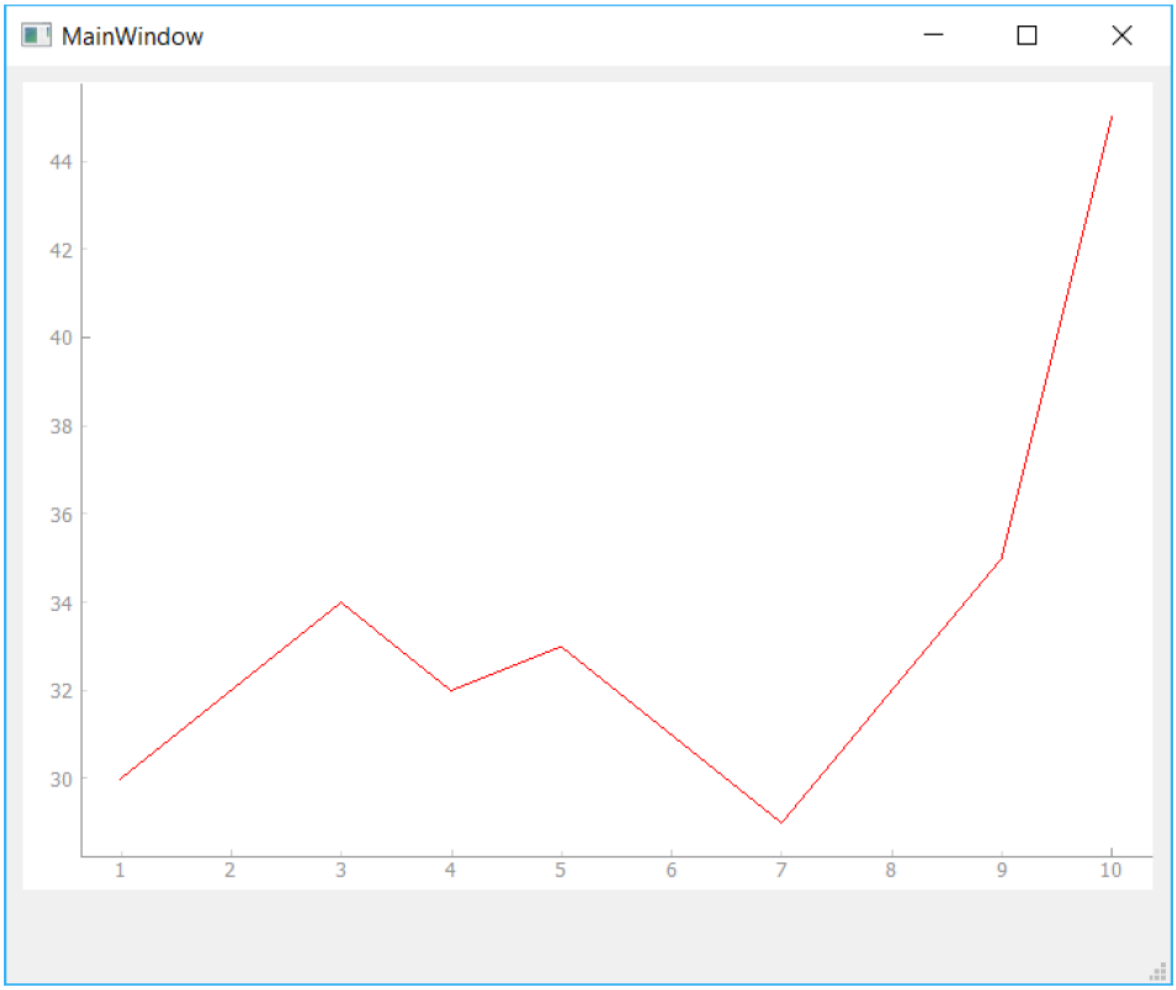
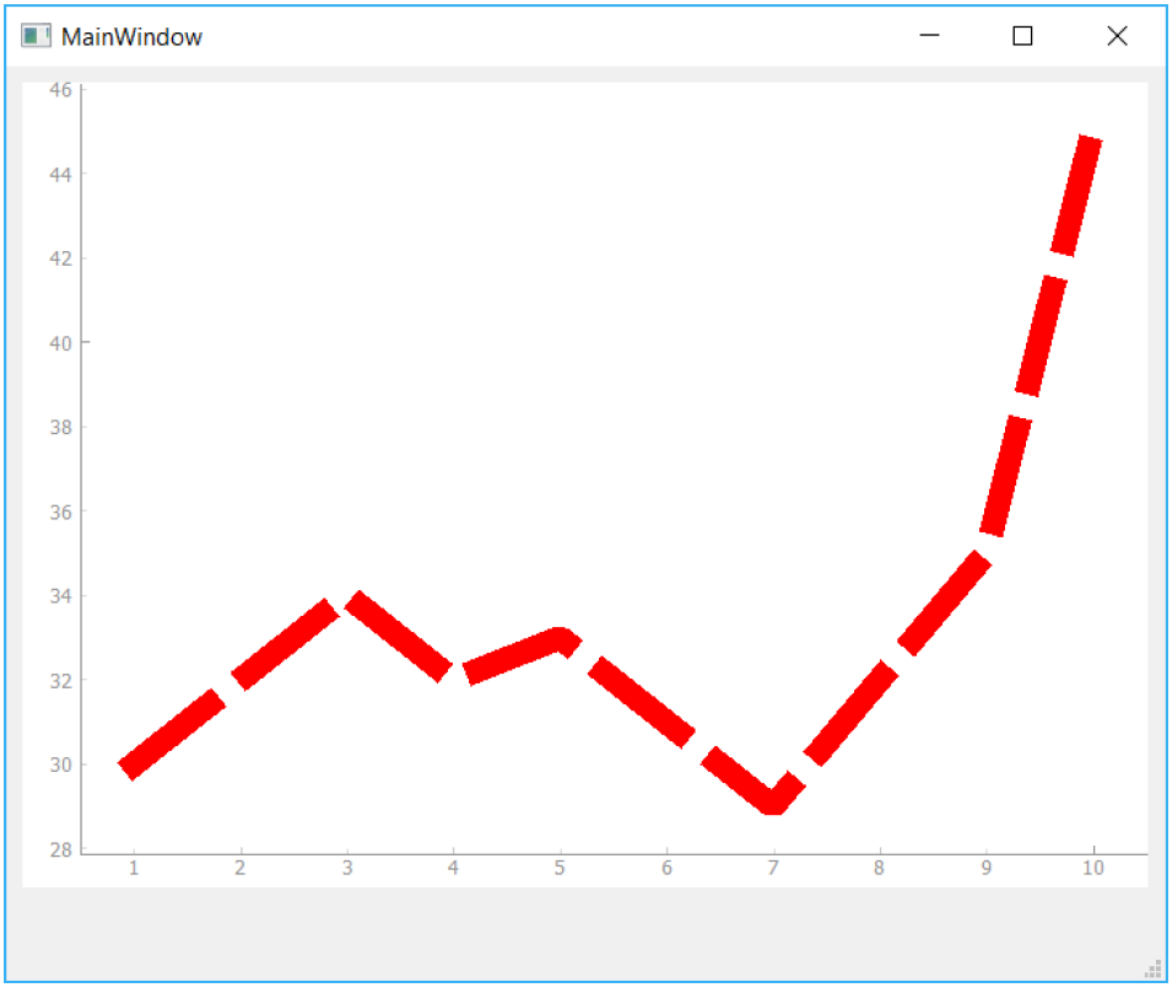


图218：更改线条颜色

通过更改 `QPen` 对象，我们可以更改线条的外观，包括使用标准 Qt 线条样式更改线条的像素宽度和样式（虚线、点线等）。例如，以下示例创建了一条 15 像素宽的红色虚线。

```
pen = pg.mkPen(color=(255, 0, 0), width=15, style=QtCore.Qt.PenStyle.DashLine)
```

结果如下所示，显示一条15像素的红色虚线。



> 图219：更改线宽和样式。

您可以使用所有标准的 Qt 线条样式，包括 `Qt.PenStyle.SolidLine`、`Qt.PenStyle.DashLine`、`Qt.PenStyle.DotLine`、`Qt.PenStyle.DashDotLine` 和 `Qt.PenStyle.DashDotDotLine`。下图显示了这些线条的示例，您可以在 Qt 文档 中阅读更多相关内容。

线条标记器

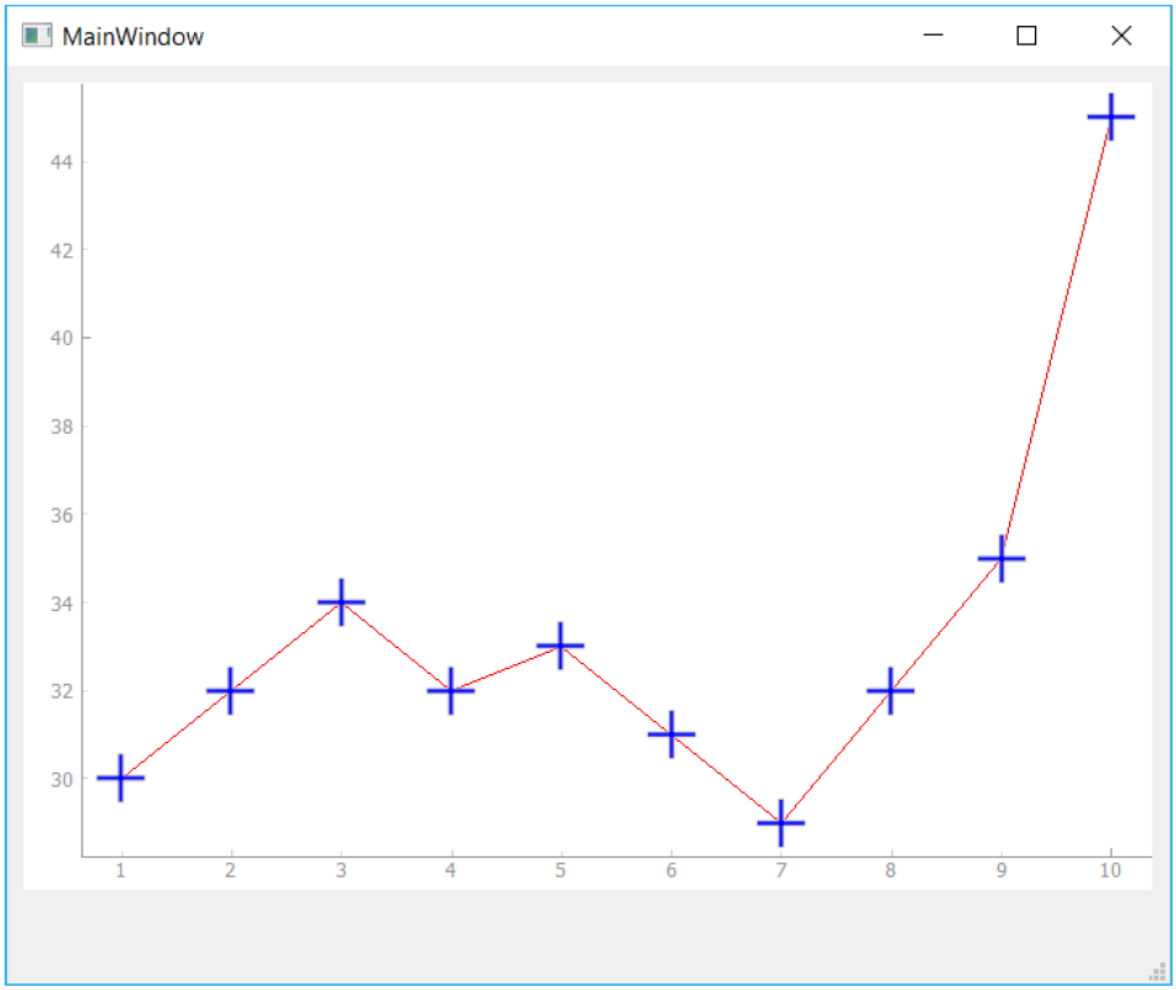
对于许多图表，在图表上添加标记（而非或替代线条）可能会有帮助。要在图表上绘制标记，可在调用 `.plot` 时传入要用作标记的符号，如下所示：

```
self.graphWidget.plot(hour, temperature, symbol='+')
```

除了 `spmbol` 外，您还可以传递符号大小（`symbolSize`）、符号画笔（`symbolBrush`）和符号钢笔（`symbolPen`）参数。传递给 `symbolBrush` 的值可以是任何颜色，或 `QBrush` 类型，而符号钢笔可以传递任何颜色或 `QPen` 实例。钢笔用于绘制形状的轮廓，而**画笔**用于填充。

例如，下面的代码将生成一个大小为30的蓝色十字标记，位于一条粗厚的红色线条上。

```
pen = pg.mkPen(color=(255, 0, 0), width=15, style=QtCore.Qt.PenStyle.DashLine)
self.graphWidget.plot(hour, temperature, pen=pen, symbol='+',symbolSize=30,
symbolBrush=('b'))
```



> 图220：每个数据点上都显示了符号。

除了 `+` 图标标记外，PyQtGraph还支持以下标准标记，如表中所示。这些标记均可按相同方式使用。

变量	标记的类型
o	圆形
s	方形
t	三角形
d	菱形
+	交叉点

> 如果您有更复杂的需求，还可以传入任何 QPainterPath 对象，从而能够绘制完全自定义的标记形状。

图表标题

图表标题对于提供图表所展示内容的上下文非常重要。在PyQtGraph中，您可以通过调用 `PlotWidget` 上的 `setTitle()` 方法，并传入标题字符串，来添加主图表标题。

```
self.graphWidget.setTitle("Your Title Here")
```

您可以通过传递附加参数来为标题（以及 PyQtGraph 中的任何其他标签）应用文本样式，包括颜色、字体大小和粗细。可用的样式参数如下所示：

样式	类型
color	(`str`) e.g. `CCFF00`
size	(`str`) e.g. `8pt`
bold	(`bool`) `True` or `False`
italic	

下面的代码将颜色设置为蓝色，字体大小为30pt。

```
self.graphWidget.setTitle("Your Title Here", color="b", size="30pt")
```

如果您愿意，还可以使用 HTML 标签语法来设置标题样式，尽管这样会降低可读性。

```
self.graphWidget.setTitle("<span style=\"color:blue;font-size:30pt\">Your Title Here</span>")
```

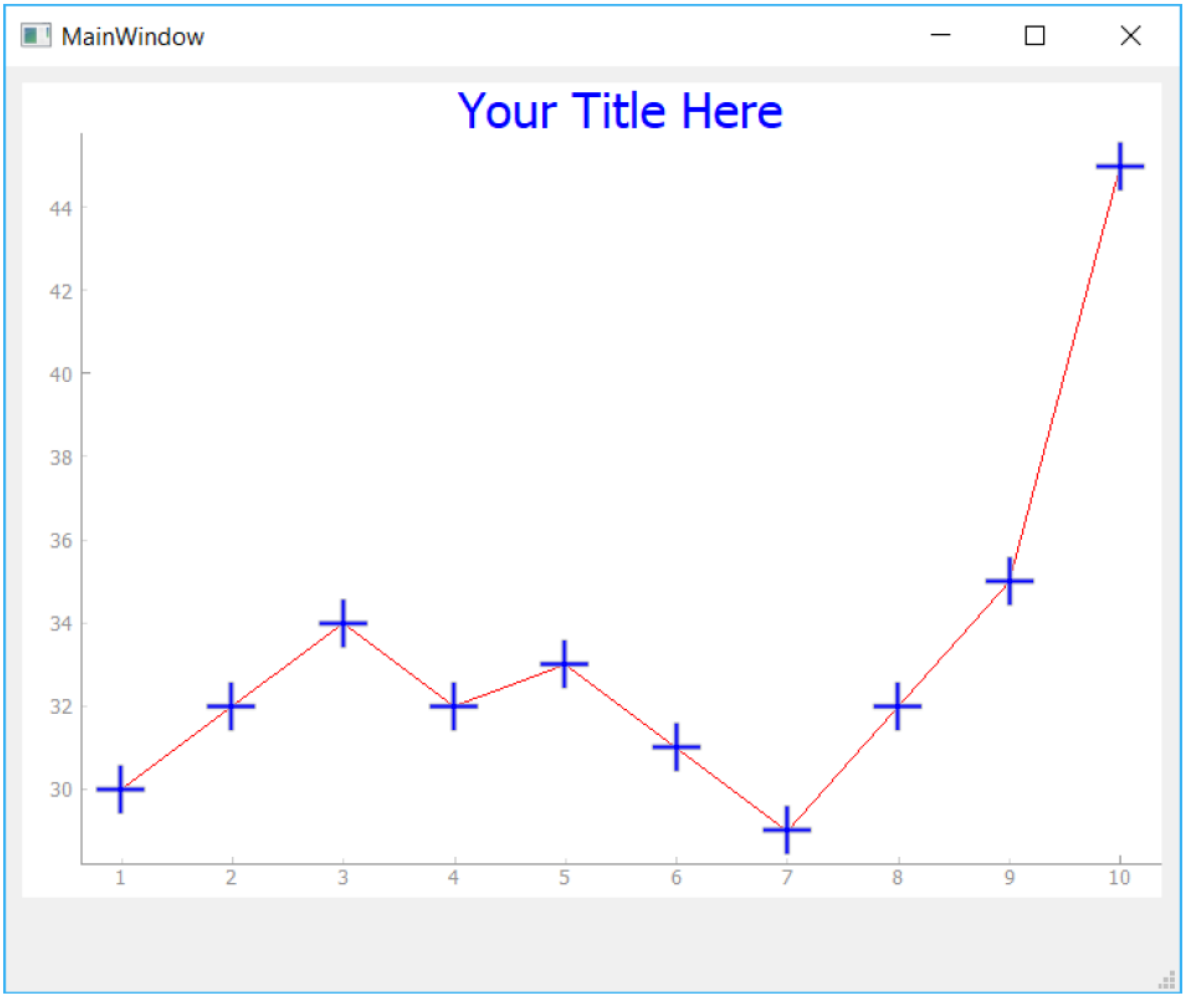


图221：带有样式标题的图例

坐标轴标题

与标题类似，我们可以使用 `setLabel()` 方法来创建轴标题。此方法需要两个参数：位置和文本。位置可以是 `left`、`right`、`top` 或 `bottom` 中的任意一个，用于描述轴上文本的位置。第二个参数文本是您希望用于标签的文本。

您可以将额外的样式参数传递给该方法。这些参数与标题的参数略有不同，因为它们需要是有效的 CSS 名称-值对。例如，大小现在是 `font-size`。由于名称 `font-size` 中包含连字符，因此您不能将其直接作为参数传递，而必须使用 `**dictionary` 方法

```
styles = {'color':'r', 'font-size':'30pt'}
self.graphWidget.setLabel('left', 'Temperature (°C)', **styles)
self.graphWidget.setLabel('bottom', 'Hour (H)', **styles)
```

这些也支持 HTML 语法，您可以自由选用。

```
self.graphWidget.setLabel('left', "<span
style=\"color:red;fontsize:30px\">Temperature (°C)</span>")
self.graphWidget.setLabel('bottom', "<span style=\"color:red;fontsize:30px\">Hour
(H)</span>")
```

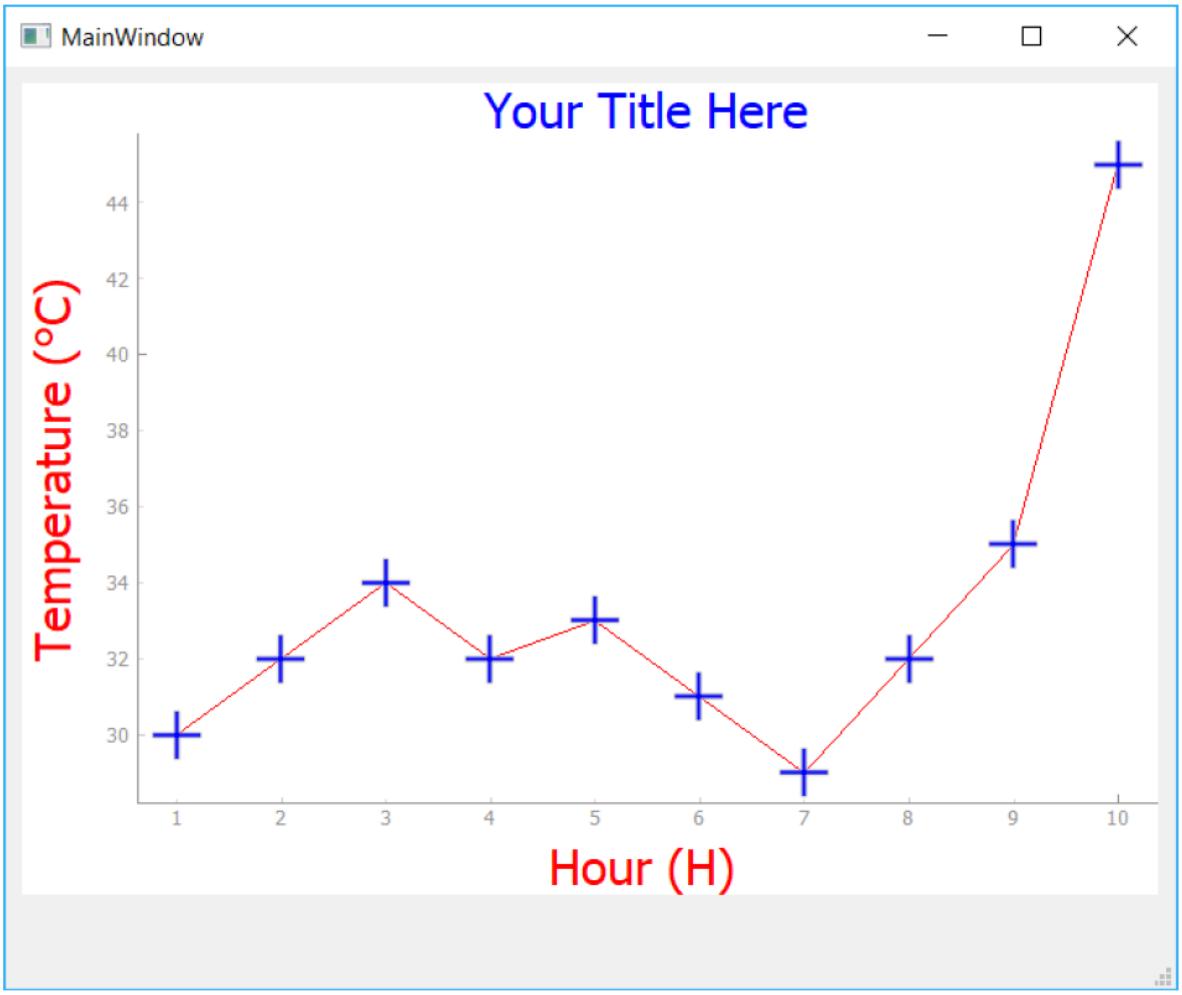


图222：自定义样式的坐标轴标签

图例

除了坐标轴和图例标题外，您通常还希望显示一个图例，用于标识特定线条所代表的内容。这在您开始向图表中添加多条线条时尤为重要。向图表添加图例可以通过调用 `PlotWidget` 的 `.addLegend` 方法实现，但在此之前，您需要在调用 `.plot()` 方法时为每条线条提供一个名称。

下面的示例将我们使用 `.plot()` 绘制的线条命名为“Sensor 1”。该名称将在图例中用于标识该线条。

```
self.graphWidget.plot(hour, temperature, name = "Sensor 1", pen = NewPen,
symbol='+', symbolSize=30, symbolBrush=('b'))
self.graphWidget.addLegend()
```

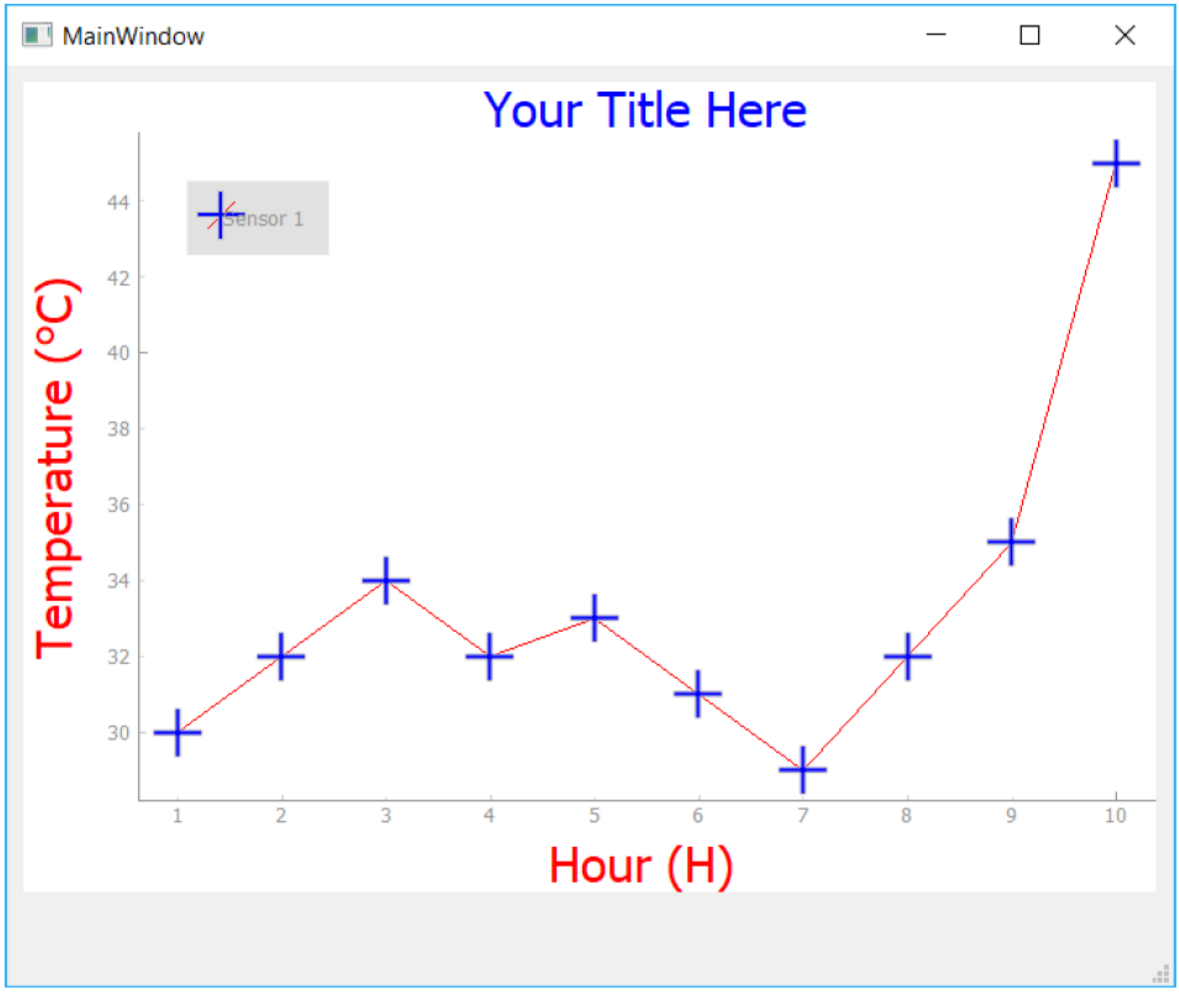


图223：带有图例的图表，显示单个项目。

图例默认显示在左上角。如果您希望移动图例，可以轻松地将图例拖动到其他位置。您还可以通过在创建图例时将一个2元组传递给 offset 参数来指定默认偏移量。

背景网格

添加背景网格可以使您的图表更易于阅读，尤其是在试图比较相对的 x 和 y 值时。您可以通过调用 `PlotWidget` 上的 `.showGrid` 方法来启用图表的背景网格。您可以独立地切换 x 和 y 网格。

以下内容将为X轴和Y轴创建网格。

```
self.graphWidget.showGrid(x=True, y=True)
```

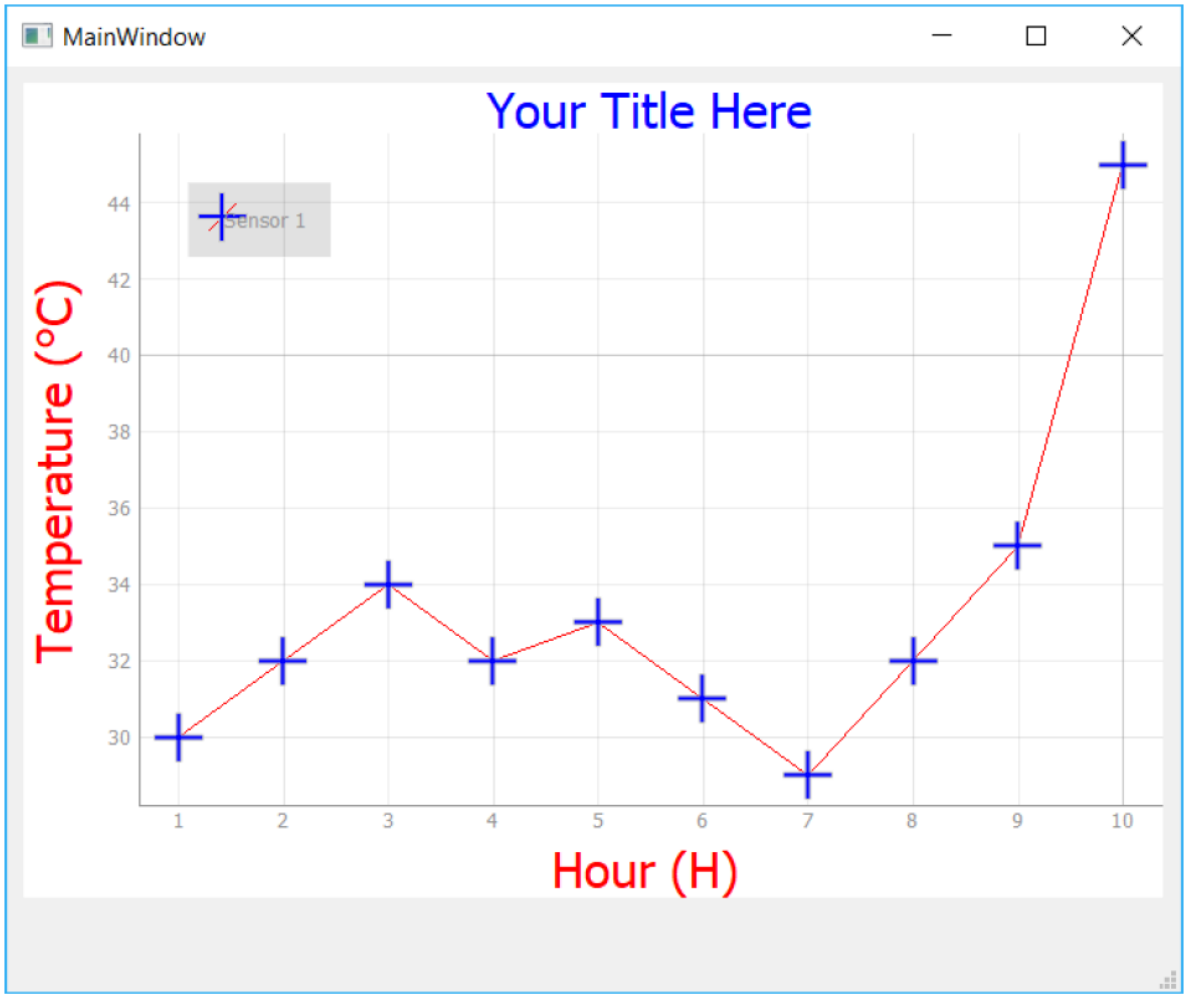


图224：背景网格

设置坐标轴的范围

有时，限制在图表上显示的数据范围，或将坐标轴锁定在固定范围（例如已知的最小值和最大值范围）可能是有用的。在 PyQtGraph 中，可以通过调用 `.setXRange()` 和 `.setYRange()` 方法实现这一点。这些方法会强制图表仅显示每个坐标轴上指定范围内的数据。

下面我们设置两个范围，每个轴一个。第一个参数是最小值，第二个参数是最大值。

```
self.graphWidget.setXRange(5, 20, padding=0)
self.graphWidget.setYRange(30, 40, padding=0)
```

可选的填充参数会使范围设置得比指定的范围更大，具体取决于指定的百分比（默认值在0.02到0.1之间，具体取决于视图框的大小）。如果您想完全移除这个填充，请传入0。

```
self.graphWidget.setXRange(5, 20, padding=0)
self.graphWidget.setYRange(30, 40, padding=0)
```

到目前为止的完整代码如下所示：

```
import sys

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph
```

```python
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        temperature = [30, 32, 34, 32, 33, 31, 29, 32, 35, 45]

        # 将背景颜色设置为白色
        self.graphWidget.setBackground("w")

        # 添加标题
        self.graphWidget.setTitle(
            "Your Title Here", color="b", size="30pt"
        )
        # 添加坐标轴标题
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphWidget.setLabel("left", "Temperature (°C)", **
                                  styles)
        self.graphWidget.setLabel("bottom", "Hour (H)", **styles)
        # 添加图例
        self.graphWidget.addLegend()
        # 添加背景网格
        self.graphWidget.showGrid(x=True, y=True)
        # 设置范围
        self.graphWidget.setXRange(0, 10, padding=0)
        self.graphWidget.setYRange(20, 55, padding=0)

        pen = pg.mkPen(color=(255, 0, 0))
        self.graphWidget.plot(
            hour,
            temperature,
            name="Sensor 1",
            pen=pen,
            symbol="+",
            symbolSize=30,
            symbolBrush=("b"),
        )


app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()
```

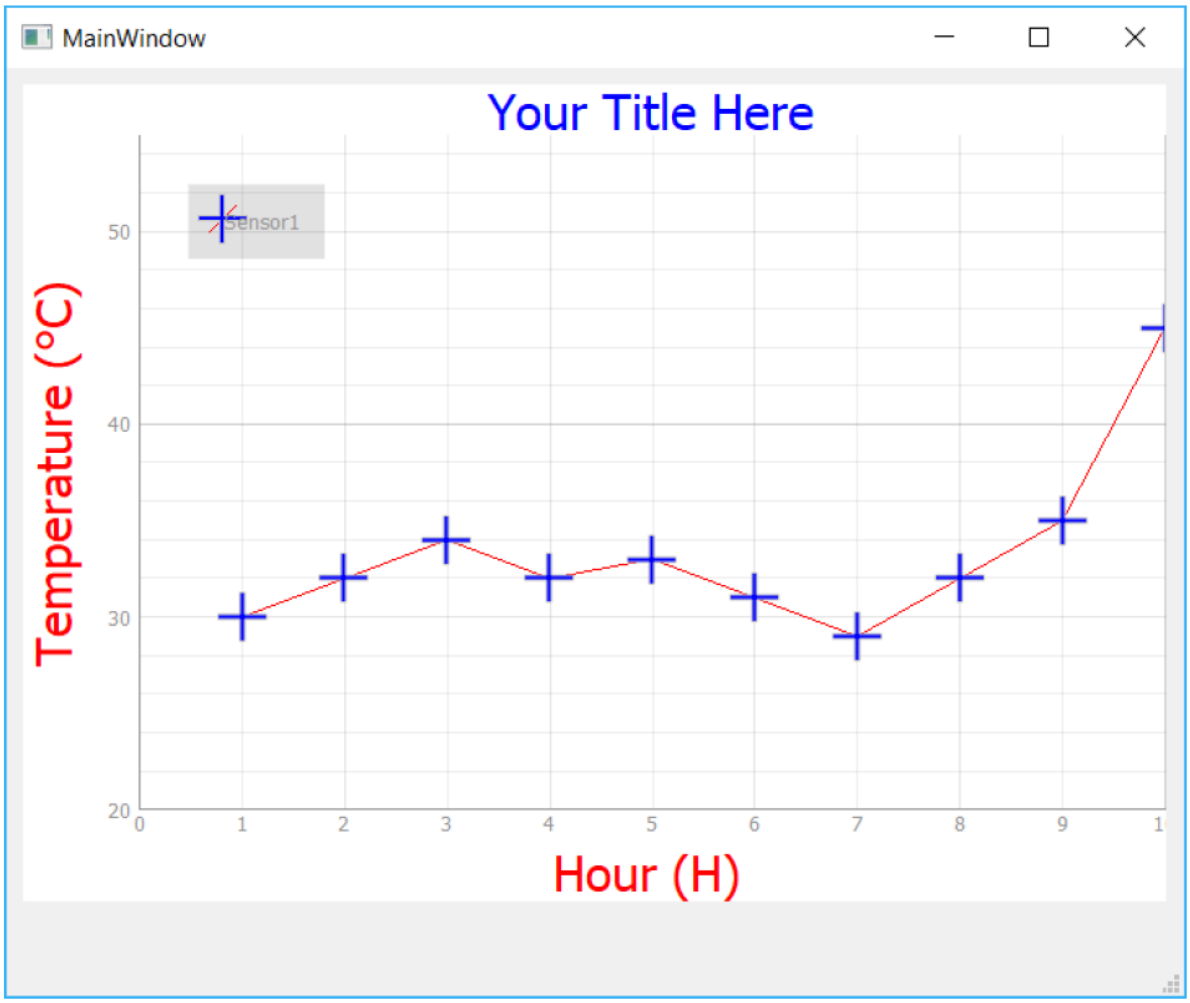


图225：限制轴的范围。

绘制多条线

绘图通常涉及多条线。在 PyQtGraph 中，这就像在同一个 `PlotWidget` 上多次调用 `.plot()` 一样简单。在以下示例中，我们将绘制两条类似的数据线，每条线使用相同的线样式、厚度等，但更改线颜色。

为了简化这一过程，我们可以在 `MainWindow` 上创建自己的自定义绘图方法。该方法接受用于绘图的 `x` 和 `y` 参数、用于图例的线条名称以及颜色参数。我们使用该颜色同时设置线条和标记的颜色。

```
    def plot(self, x, y, plotname, color):
        pen = pg.mkPen(color=color)
        self.graphWidget.plot(x, y, name=plotname, pen=pen, symbol=
                              '+', symbolSize=30, symbolBrush=(color))
```

要绘制单独的线条，我们将创建一个名为 `temperature_2` 的新数组，并用与 `temperature`（现在为 `temperature_1`）类似的随机数填充它。将这些线条并排绘制，我们可以将它们进行比较。现在，您可以调用 `plot` 函数两次，这将在图表上生成两条线。

```
self.plot(hour, temperature_1, "Sensor1", 'r')
self.plot(hour, temperature_2, "Sensor2", 'b')
```

Listing 224. plotting/pyqtgraph_5.py

```
import sys
```

```python
from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        temperature = [30, 32, 34, 32, 33, 31, 29, 32, 35, 45]

        # 将背景颜色设置为白色
        self.graphWidget.setBackground("w")

        # 添加标题
        self.graphWidget.setTitle(
            "Your Title Here", color="b", size="30pt"
        )
        # 添加坐标轴标题
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphWidget.setLabel("left", "Temperature (°C)", **
                                  styles)
        self.graphWidget.setLabel("bottom", "Hour (H)", **styles)
        # 添加图例
        self.graphWidget.addLegend()
        # 添加背景网格
        self.graphWidget.showGrid(x=True, y=True)
        # 设置范围
        self.graphWidget.setXRange(0, 10, padding=0)
        self.graphWidget.setYRange(20, 55, padding=0)

        self.plot(hour, temperature_1, "Sensor1", "r")
        self.plot(hour, temperature_2, "Sensor2", "b")

    def plot(self, x, y, plotname, color):
        pen = pg.mkPen(color=color)
        self.graphWidget.plot(
            x,
            y,
            name=plotname,
            pen=pen,
            symbol="+",
            symbolSize=30,
            symbolBrush=(color),
        )


app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()
```

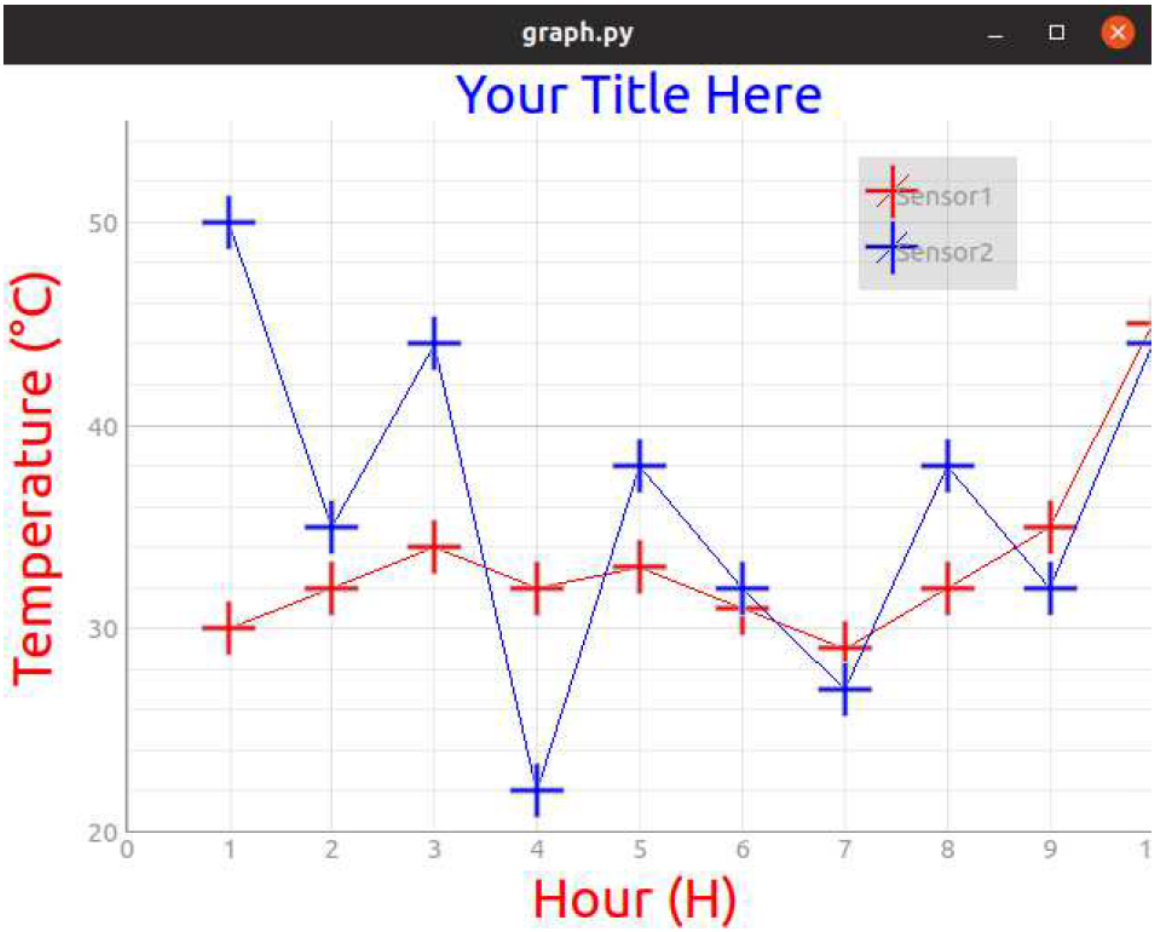


图226：包含两条线的图表

您可以尝试使用此功能，自定义您的标记、线条宽度、颜色及其他参数。

清空图表

最后，有时您可能需要定期清除并刷新绘图。您可以通过调用 `.clear()` 方法轻松实现这一点。

```
self.graphwidget.clear()
```

这将从图中删除线条，但保持所有其他属性不变。

更新图表

虽然您可以简单地清除绘图区域并重新绘制所有元素，但这意味着Qt必须销毁并重新创建所有 `QGraphicsScene` 对象。对于小型或简单的图形，这可能不会被注意到，但如果您想创建高性能的流式图形，最好直接更新数据。PyQtGraph 会获取新数据并更新绘制的线条以匹配，而不会影响图形中的其他元素。

要更新一条线，我们需要获取该线对象的引用。该引用在首次使用 `.plot` 方法创建线时返回，我们可以将其存储在变量中。请注意，这是对线对象的引用，而非对绘图的引用。

```
my_line_ref = graphWidget.plot(x, y)
```

一旦我们获得了引用，更新图表只需调用 `.setData` 方法，将新数据应用到该引用即可。

Listing 225. plotting/pyqtgraph_6.py

```
import sys
from random import randint

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        self.x = list(range(100)) # 100个时间点
        self.y = [
            randint(0, 100) for _ in range(100)
        ] # 100个数据点

        self.graphWidget.setBackground("w")

        pen = pg.mkPen(color=(255, 0, 0))
        self.data_line = self.graphWidget.plot(
            self.x, self.y, pen=pen
        ) #1

        self.timer = QtCore.QTimer()
        self.timer.setInterval(50)
        self.timer.timeout.connect(self.update_plot_data)
        self.timer.start()

    def update_plot_data(self):
        self.x = self.x[1:] # 移除第一个 y 元素.
        self.x.append(
            self.x[-1] + 1
        ) # 添加一个比上一个值大1的新值.
        self.y = self.y[1:] # 删除第一个
        self.y.append(randint(0, 100)) # 添加一个新的随机值.
        self.data_line.setData(self.x, self.y) # 更新数据.
```

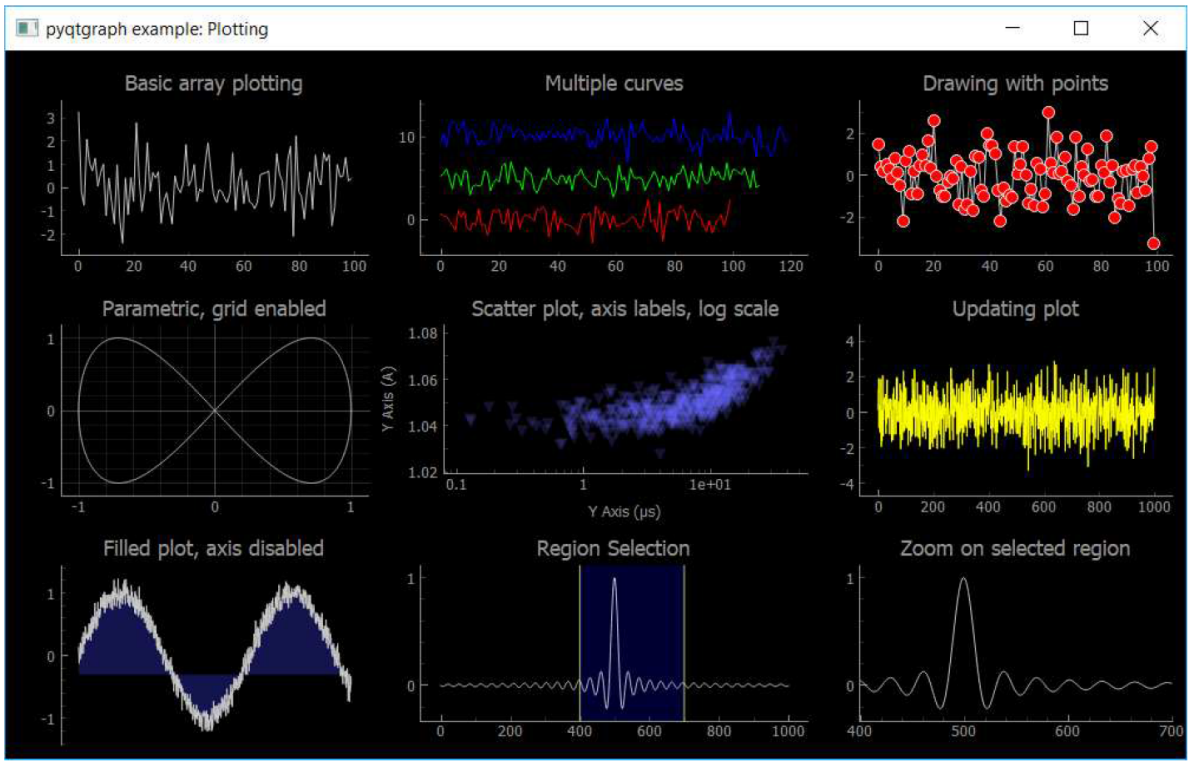
> 1. 这里我们引用了之前绘制的线，并将它存储为 `self.data_line` 。

我们使用 `QTimer` 每 50 毫秒更新一次数据，将触发器设置为调用自定义的槽方法 `update_plot_data`，在那里我们将更改数据。我们在 `__init__` 块中定义此计时器，因此它会自动启动。

如果运行该应用程序，您应该会看到一个图表，其中随机数据快速向左滚动，X 值也会同步更新并滚动，仿佛在流式传输数据。您可以用自己的真实数据替换随机数据，例如从实时传感器读数或 API 中获取。PyQtGraph 性能足够强大，可以支持使用此方法创建多个图表。

总结

在本章中，我们学习了如何使用 PyQtGraph 绘制简单的图表，并自定义线条、标记和标签。要全面了解 PyQtGraph 的方法和功能，请参阅 PyQtGraph文档及API参考。PyQtGraph在GitHub上的仓库 中还包含了一系列更复杂的示例图表，这些示例位于Plotting.py文件中（如下所示）。



> 图227：示例图表摘自PyQtGraph文档。

30. 使用 Matplotlib 进行数据可视化

在前一部分中，我们介绍了如何使用PyQtGraph在PyQt6中进行绘图。该库使用Qt基于向量的 QGraphicsScene 来绘制图表，并提供了交互式和高性能绘图的出色接口。

然而，还有另一个用于 Python 的绘图库，它被更广泛地使用，并提供了更丰富的绘图种类——Matplotlib。如果您正在将现有的数据分析工具迁移到 PyQt6 图形用户界面，或者您只是想使用 Matplotlib 提供的各种绘图功能，那么您需要了解如何将 Matplotlib 绘图纳入您的应用程序。

在本章中，我们将介绍如何在 PyQt6 应用程序中嵌入 Matplotlib 图表。

> 许多其他 Python 库（如 seaborn 和 pandas）也使用 Matplotlib 进行绘图。这些绘图可以像这里示例中一样嵌入到 PyQt6 中，并且在绘图时传递的轴引用。本章末尾有一个 pandas 的示例。

安装 Matplotlib

以下示例假设您已安装 Matplotlib。如果未安装，您可以使用 `pip` 进行安装。

撰写本文时，PyQt6 还非常新。有一个实验性分支，其中包含Qt6 支持，您可以使用它 —

```
pip install git+https://github.com/anntzer/matplotlib.git@qt6
```

一个简单的例子

以下最简单的示例设置了一个 Matplotlib 画布 `FigureCanvasQTAgg`，该画布创建了 `Figure` 并向其中添加了一组轴。该画布对象也是一个 `QWidget`，因此可以像其他 Qt 控件一样直接嵌入到应用程序中。

Listing 226. plotting/matplotlib_1.py

```
import sys

from PyQt6 import QtWidgets # 在导入matplotlib之前先导入PyQt6。

import matplotlib
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.figure import Figure

matplotlib.use("QtAgg")


class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        # 创建 maptlotlib FigureCanvasQTAgg 对象，该对象定义了一组坐标轴，即 self.axes.
        sc = MplCanvas(self, width=5, height=4, dpi=100)
        sc.axes.plot([0, 1, 2, 3, 4], [10, 1, 20, 3, 40])
        self.setCentralWidget(sc)

        self.show()


app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()
```

在这种情况下，我们使用 `.setCentralWidget()` 将 `MplCanvas` 控件添加到窗口作为中央控件。这意味着它将占据整个窗口，并随窗口一起调整大小。绘制的数据 `[0,1,2,3,4], [10,1,20,3,40]` 以两个数字列表（分别表示 x 和 y 坐标）的形式提供，符合 `.plot` 方法的要求。

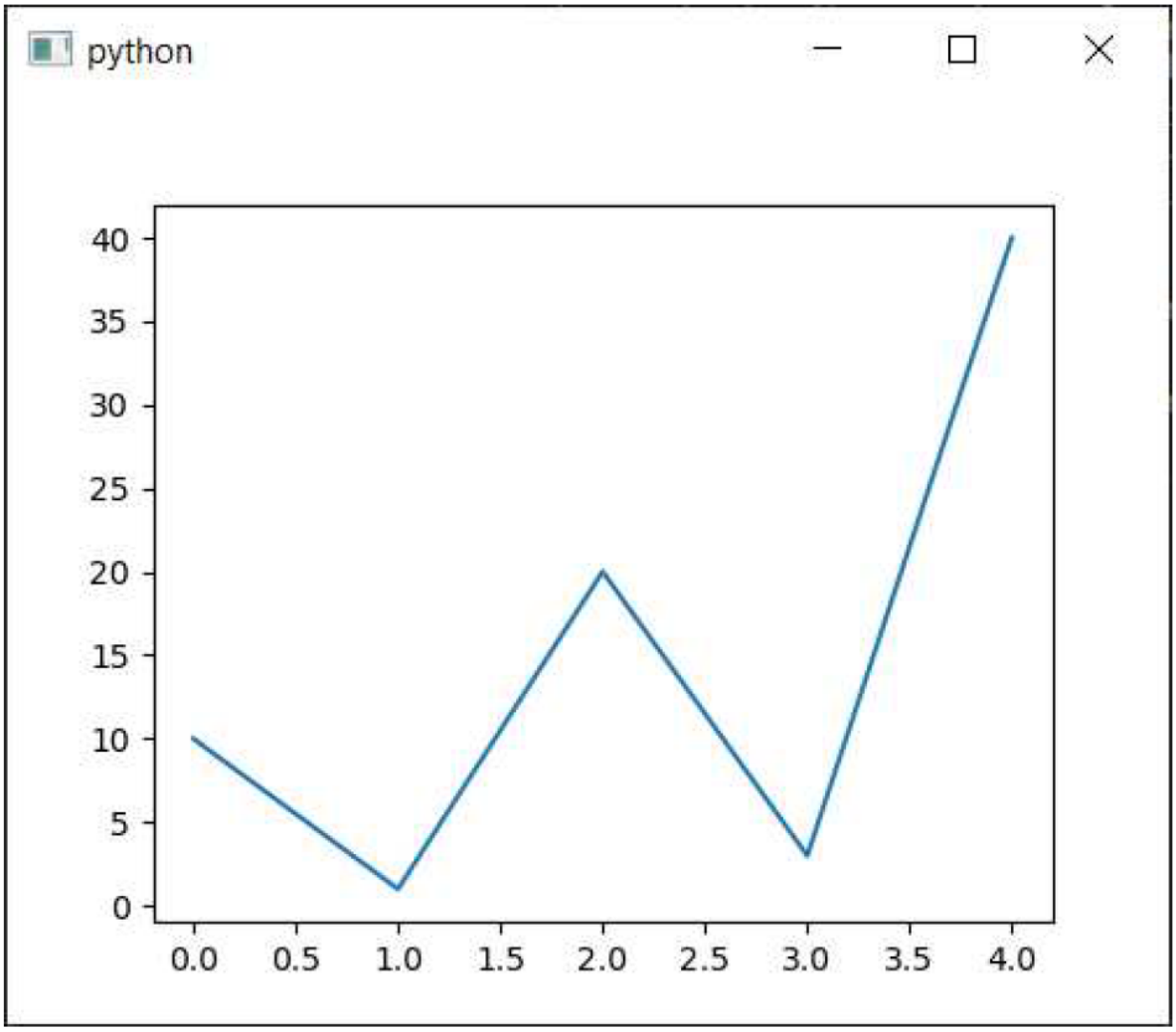


图228：一个简单的图表

图表控制

在 PyQt6 中显示的 Matplotlib 图实际上是由 Agg 后端渲染为简单的（位图）图像。`FigureCanvasQTAgg` 类封装了这个后端，并将生成的图像显示在 Qt 控件上。这种架构的效果是，Qt 不知道线和其他绘图元素的位置，只知道在控件上点击和鼠标移动的 x、y 坐标。

然而，Matplotlib 内置了对 Qt 鼠标事件的处理和将其转换为图上的交互的支持。这可以通过一个自定义工具栏来控制，该工具栏可以与图一起添加到您的应用程序中。在本节中，我们将介绍如何添加这些控件，以便我们能够缩放、平移和从嵌入的 Matplotlib 图中获取数据。

完整的代码如下所示，它导入了工具栏控件 `NavigationToolbar2QT`，并将其添加到 `QVBoxLayout` 中的界面中。

Listing 227. plotting/matplotlib_2.py

```
import sys
from PyQt6 import QtWidgets # 在导入matplotlib之前先导入PyQt6
import matplotlib
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.backends.backend_qtagg import (
NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

matplotlib.use("QtAgg")
```

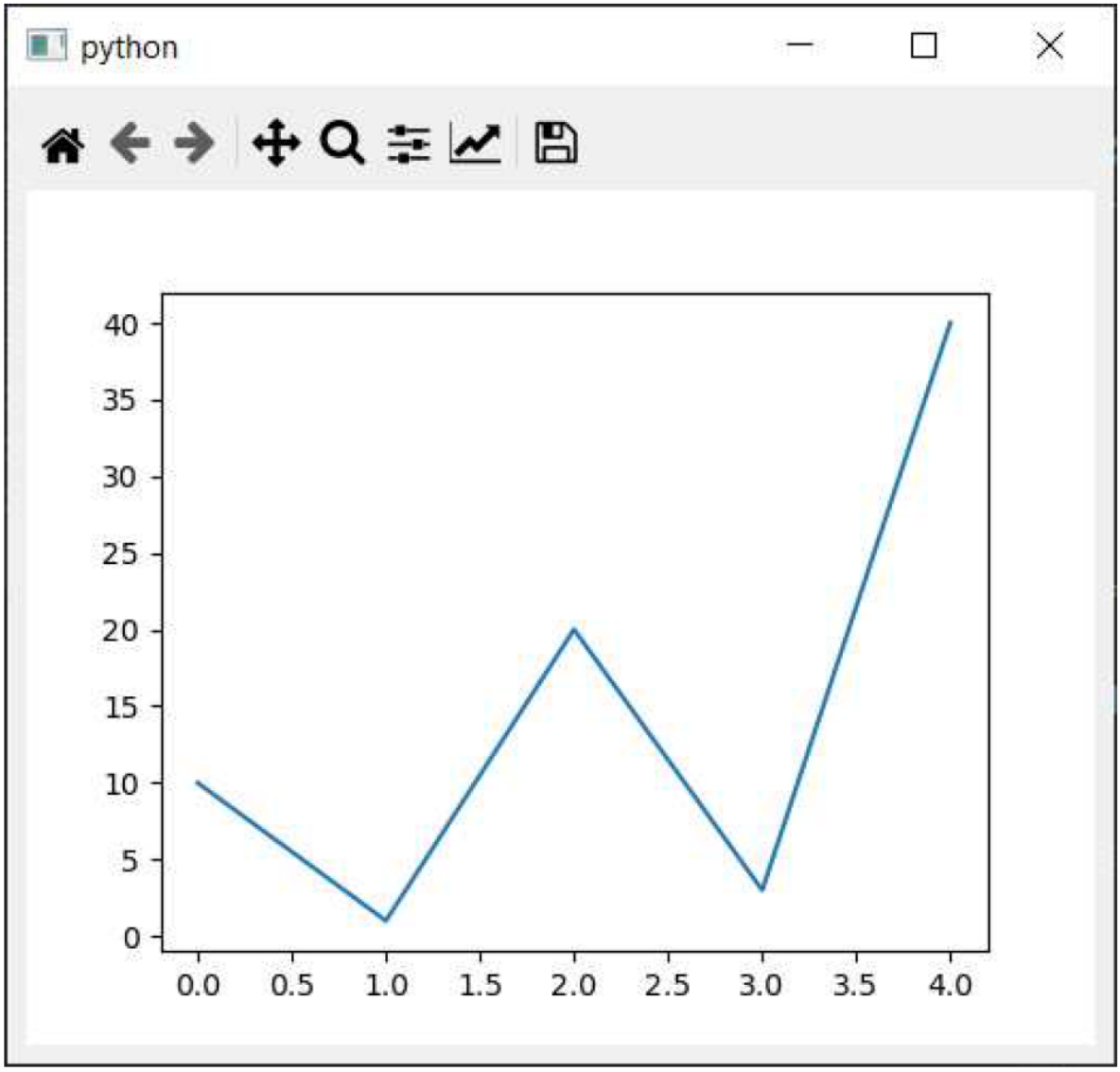
```
class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        sc = MplCanvas(self, width=5, height=4, dpi=100)
        sc.axes.plot([0, 1, 2, 3, 4], [10, 1, 20, 3, 40])

        # 创建工具栏，将画布作为第一个参数传递，父窗口(self, the MainWindow)作为第二个参
数。
        toolbar = NavigationToolbar(sc, self)

        layout = QtWidgets.QVBoxLayout()
        layout.addWidget(toolbar)
        layout.addWidget(sc)

        # 创建一个占位符控件来容纳我们的工具栏和画布.
        widget = QtWidgets.QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)

        self.show()


app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()
```

我们将逐一说明这些变更。

首先，我们从 `matplotlib.backends.backend_qt5agg.NavigationToolbar2QT` 导入工具栏控件，并将其重命名为更简单的名称 `NavigationToolbar`。我们通过调用 `NavigationToolbar` 并传入两个参数来创建工具栏实例，第一个参数是画布对象 `sc`，第二个参数是工具栏的父对象，在本例中是 `MainWindow` 对象 `self`。传入画布对象可将创建的工具栏与之关联，从而实现对其的控制。生成的工具栏对象被存储在变量 `toolbar` 中。

我们需要在窗口中添加两个控件，一个在另一个上方，因此我们使用一个 `QVBoxLayout`。首先，我们将工具栏控件 `toolbar` 和画布控件 `sc` 添加到此布局中。最后，我们将此布局设置到我们的简单控件布局容器中，该容器被设置为窗口的中央控件。

运行上述代码将生成以下窗口布局，显示在底部的绘图和顶部的工具栏作为控件。



> 图229：带工具栏的Matplotlib画布

NavigationToolbar2QT 提供的按钮可用于控制以下操作：

- 首页、后退/前进、平移与缩放，用于在图中导航。后退/前进按钮可用于在导航步骤之间前后移动，例如先放大再点击后退按钮将返回上一个缩放级别。首页按钮可返回图的初始状态。
- 绘图边距/位置配置，可调整绘图在窗口内的位置。
- 轴/曲线样式编辑器，您可以在其中修改图标题和轴刻度，以及设置图线颜色和线样式。颜色选择使用平台默认的颜色选择器，允许选择任何可用的颜色。
- 保存，将生成的图形保存为图像（支持所有Matplotlib支持的格式）。

以下列出了其中一些配置设置：

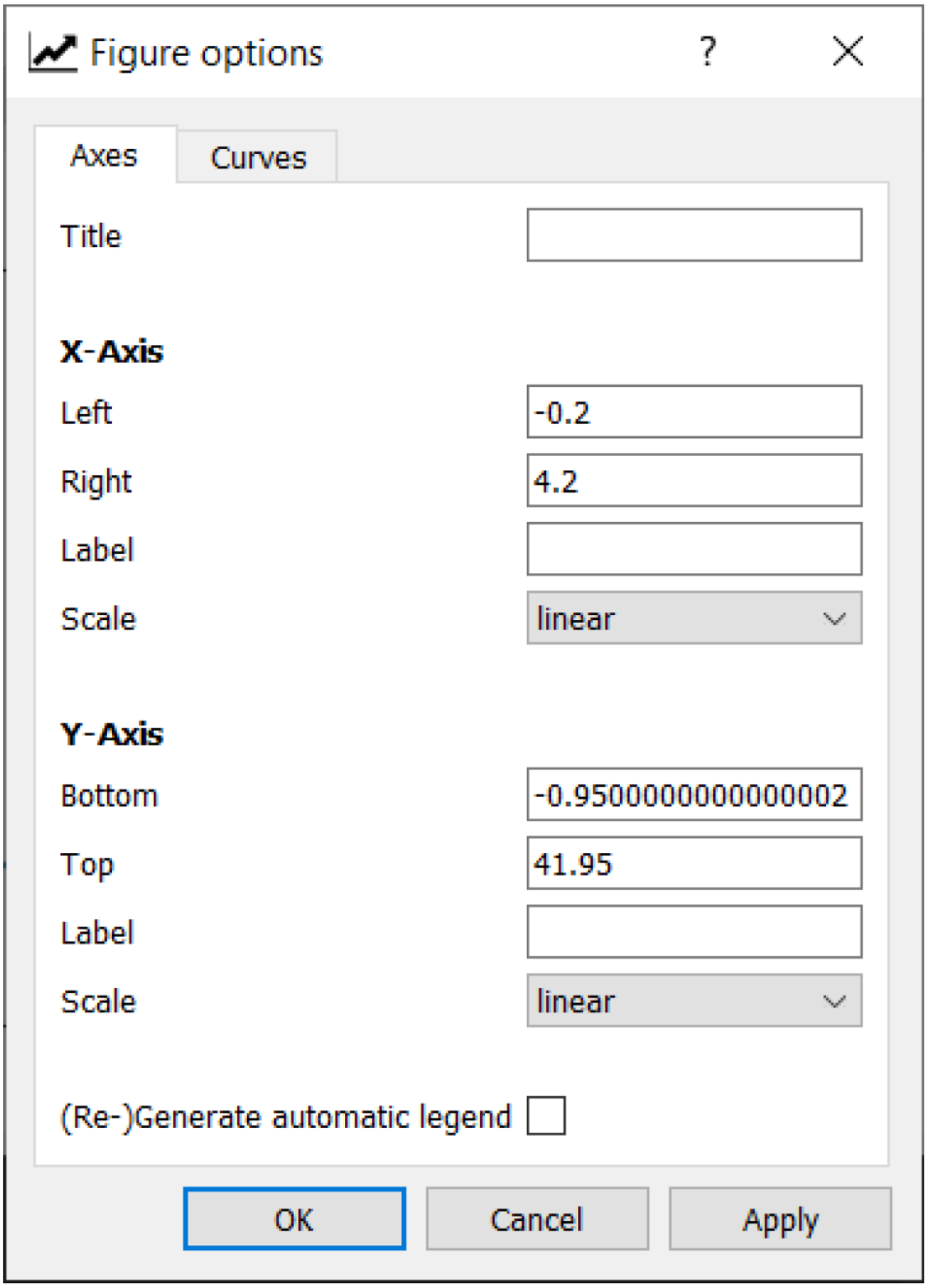


图230：Matplotlib 图表选项

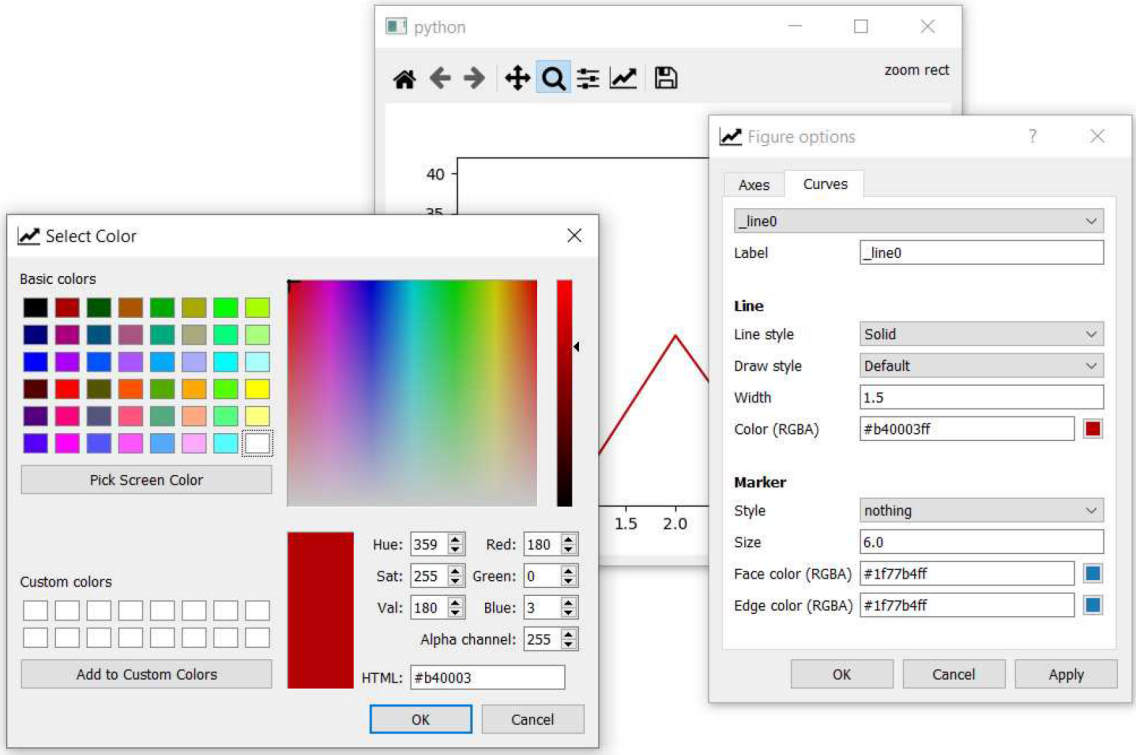


图231：Matplotlib 曲线选项

有关如何操作和配置Matplotlib图表的更多信息，请参阅官方 Matplotlib工具栏文档。

更新图表

在应用程序中，您经常需要更新图表中显示的数据，无论是响应用户的输入还是来自 API 的更新数据。在 Matplotlib 中，有两种方法可以更新图表，您可以选择：

1. 清除并重新绘制画布（更简单，但速度较慢）
2. 通过保留绘制线的引用并更新数据

如果性能对您的应用程序很重要，建议您选择后者，但前者更简单。我们先从简单的清除并重绘方法开始：

清除并重新绘制

Listing 228. plotting/matplotlib_3.py

```
import sys
from PyQt6 import QtWidgets # 在导入matplotlib之前先导入PyQt6
import matplotlib
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.backends.backend_qtagg import (
NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

matplotlib.use("QtAgg")


class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
```

```
        super().__init__(fig)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.canvas = MplCanvas(self, width=5, height=4, dpi=100)
        self.setCentralWidget(self.canvas)

        n_data = 50
        self.xdata = list(range(n_data))
        self.ydata = [random.randint(0, 10) for i in range(n_data)]
        self.update_plot()

        self.show()

        # 设置一个定时器，通过调用 update_plot 函数触发重新绘制.
        self.timer = QtCore.QTimer()
        self.timer.setInterval(100)
        self.timer.timeout.connect(self.update_plot)
        self.timer.start()

    def update_plot(self):
        # 删除第一个 y 元素，并添加一个新的 y 元素.
        self.ydata = self.ydata[1:] + [random.randint(0, 10)]
        self.canvas.axes.cla() # 清空画布.
        self.canvas.axes.plot(self.xdata, self.ydata, "r")
        # 触发画布更新并重新绘制.
        self.canvas.draw()


app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()
```

在此示例中，我们将绘图操作移至 `update_plot` 方法中，以保持其独立性。在此方法中，我们对 `ydata` 数组的第一个值进行 `[1:]` 操作移除，然后追加一个0到10之间的随机整数。此操作会将数据向左滚动。

要重新绘制，我们只需调用 `axes.cla()` 清除坐标轴（整个画布），然后调用 `axes.plot(…)` 重新绘制数据，包括更新后的值。然后，通过调用 `canvas.draw()` 将生成的画布重新绘制到控件上。

`update_plot` 方法每 100 毫秒通过 `QTimer` 调用一次。`clear-andrefresh` 方法足够快，可以以这种速度保持绘图更新，但如我们将看到的，随着速度的提高，它会出现问题。

就地重绘

更新绘制线条所需的更改相对较少，仅需添加一个变量来存储并获取绘制线条的引用。更新后的 `MainWindow` 代码如下所示：

Listing 229. plotting/matplotlib_4.py

```
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
```

```
        super().__init__()
        self.canvas = MplCanvas(self, width=5, height=4, dpi=100)
        self.setCentralWidget(self.canvas)
        n_data = 50
        self.xdata = list(range(n_data))
        self.ydata = [random.randint(0, 10) for i in range(n_data)]

        # 我们需要将绘制的线条的引用保存在某个地方，以便我们可以将新数据应用到它上.
        self._plot_ref = None
        self.update_plot()

        self.show()

        # 设置一个定时器，通过调用 update_plot 函数触发重新绘制.
        self.timer = QtCore.QTimer()
        self.timer.setInterval(100)
        self.timer.timeout.connect(self.update_plot)
        self.timer.start()

    def update_plot(self):
        # 删除第一个 y 元素，并添加一个新的元素.
        self.ydata = self.ydata[1:] + [random.randint(0, 10)]

        # 注意：我们不再需要清除轴。.
        if self._plot_ref is None:
            # 第一次没有图表参考，所以做一个正常的图表.
            # .plot 返回一个线性引用列表，由于我们只获取一个，可以直接取第一个元素.
            plot_refs = self.canvas.axes.plot(
                self.xdata, self.ydata, "r"
            )
            self._plot_ref = plot_refs[0]
        else:
            # 我们有一个引用，可以使用它来更新该行的数据.
            self._plot_ref.set_ydata(self.ydata)
        # 触发画布更新并重新绘制.
        self.canvas.draw()
```

首先，我们需要一个变量来保存要更新的绘制线条的引用，这里我们将其命名为 `_plot_ref` 。我们初始化 `self._plot_ref` 为 `None`，这样我们可以在后续代码中检查其值以确定线条是否已绘制——如果值仍然为 `None`，则表示线条尚未绘制。

> 如果您需要绘制多条线，您可能希望使用列表或字典数据结构来存储多个引用，并跟踪每条线的具体信息。

最后，我们像之前一样更新 `ydata`数据，将其向左旋转并添加一个新的随机值。然后我们可以选择：

1. 如果 `self._plot_ref` 为 `None` （即尚未绘制该线），则绘制该线，并将引用存储在 `self._plot_ref` 中。

2. 通过调用 `self._plot_ref.set_ydata(self.ydata)` 来更新当前的线条。

在调用 .plot 时，我们会获得绘制线的引用。然而，`.plot` 方法会返回一个列表（以支持单次 `.plot` 调用绘制多条线的情况）。在我们的情况下，我们只绘制一条线，因此我们只需获取该列表中的第一个元素——一个 `Line2D` 对象。为了将这个单一值赋给我们的变量，我们可以先将它赋给一个临时变量 `plot_refs`，然后将第一个元素赋给我们的 `self._plot_ref` 变量。

```
plot_refs = self.canvas.axes.plot(self.xdata, self.ydata, 'r')
self._plot_ref = plot_refs[0]
```

您还可以使用元组解包，从列表中提取第一个（也是唯一）元素：

```
self._plot_ref, = self.canvas.axes.plot(self.xdata, self.ydata, 'r')
```

如果您运行生成的代码，在当前速度下，这种方法与之前的方法在性能上不会有明显差异。然而，如果您尝试以更快的速度更新图表（例如每10毫秒一次），您会发现清除图表并重新绘制所需的时间更长，且更新速度无法跟上计时器。这种性能差异是否足以在您的应用程序中产生影响，取决于您正在构建的内容，并且您应该权衡保持和管理绘制线条的引用所带来的额外复杂性。

从 Pandas 中嵌入图表

Pandas 是一个专注于处理表格（数据框）和系列数据结构的 Python 包，对于数据分析工作流程特别有用。它内置了对 Matplotlib 绘图的支持，本文将简要介绍如何将这些绘图嵌入到 PyQt6 中。通过此方法，您可以开始构建基于 Pandas 的 PyQt6 数据分析应用程序

Pandas 的绘图函数可直接从 DataFrame 对象访问。该函数的签名较为复杂，提供了大量选项来控制绘图的具体方式。

```
DataFrame.plot(
    x=None, y=None, kind='line', ax=None, subplots=False,
    sharex=None, sharey=False, layout=None, figsize=None,
    use_index=True, title=None, grid=None, legend=True, style=None,
    logx=False, logy=False, loglog=False, xticks=None, yticks=None,
    xlim=None, ylim=None, rot=None, fontsize=None, colormap=None,
    table=False, yerr=None, xerr=None, secondary_y=False,
    sort_columns=False, **kwargs
)
```

我们最感兴趣的参数是 `ax`，它允许我们传入自己的 `matplotlib.Axes` 实例，Pandas 将在此实例上绘制 `DataFrame`。

Listing 230. plotting/matplotlib_5.py

```
import sys
from PyQt6 import (
    QtCore,
    QtWidgets,
) # 在导入matplotlib之前先导入PyQt6
import matplotlib
import pandas as pd
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.figure import Figure
```

```
matplotlib.use("QtAgg")


class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        # 创建 maptlotlib FigureCanvasQTAgg 对象，该对象定义了一组坐标轴，即 self.axes.
        sc = MplCanvas(self, width=5, height=4, dpi=100)
        # 创建一个包含简单数据和列名的 pandas DataFrame.
        df = pd.DataFrame(
            [
                [0, 10],
                [5, 15],
                [2, 20],
                [15, 25],
                [4, 10],
            ],
            columns=["A", "B"],
        )
        # p绘制 pandas DataFrame，传入 matplotlib Canvas 坐标轴.
        df.plot(ax=sc.axes)

        self.setCentralWidget(sc)
        self.show()


app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()
```

这里的关键步骤是在调用 `DataFrame` 的 `plot` 方法时传入画布坐标轴，即在行`df.plot(ax=sc.axes)` 中传入。您可以使用相同的模式来随时更新图表，不过需要注意的是，Pandas会清空并重新绘制整个画布，这意味着它并不适合高性能绘图。

通过Pandas生成的结果图如下所示：

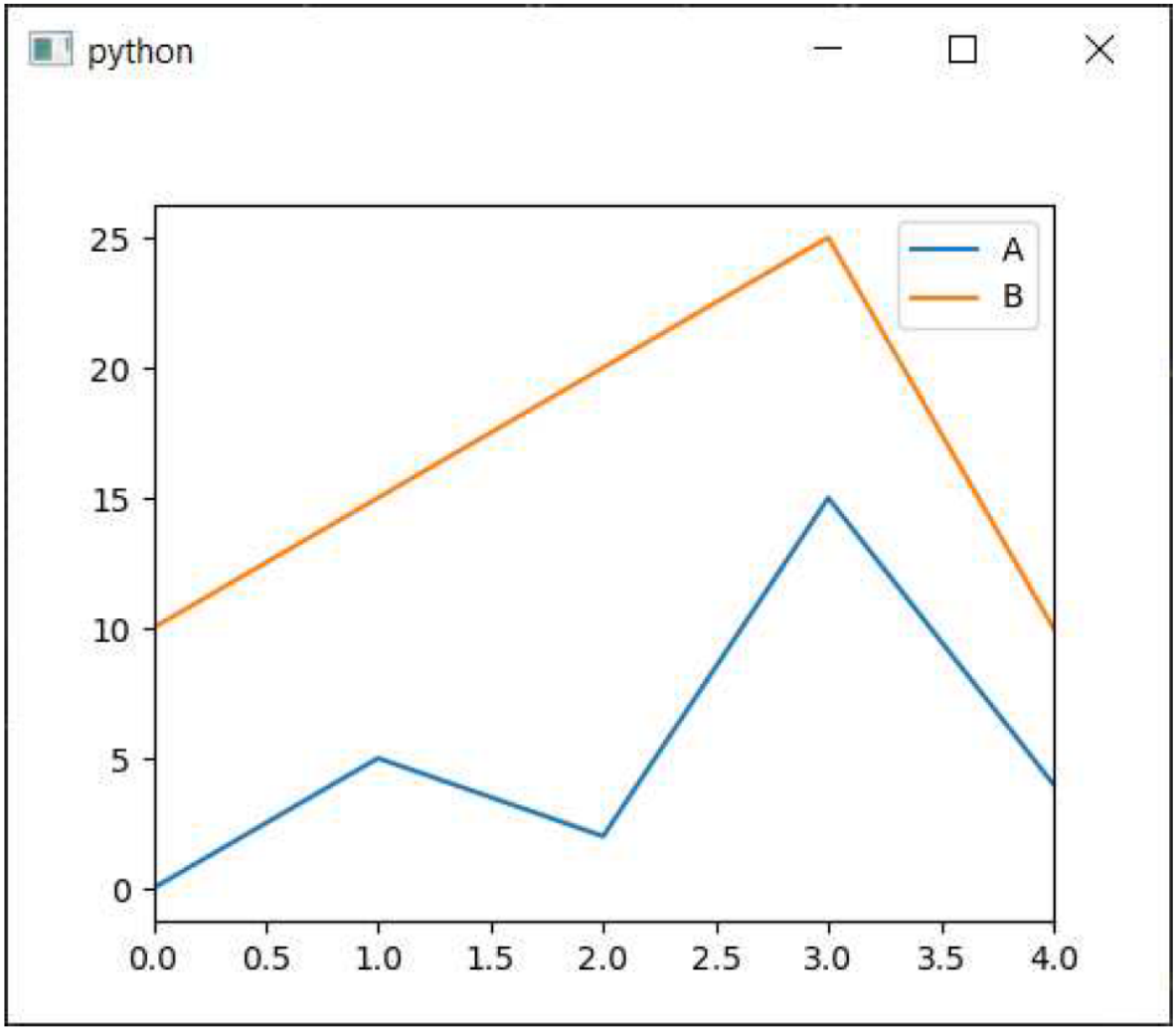


图232：使用matplotlib Canvas生成的 pandas 图

与之前一样，您可以为使用Pandas生成的图表添加Matplotlib工具栏和控制支持，允许您实时缩放/平移并修改图表。以下代码将我们之前的工具栏示例与Pandas示例相结合：

Listing 231. plotting/matplotlib_6.py

```
import sys
from PyQt6 import (
    QtCore,
    QtWidgets,
) # 在导入matplotlib之前先导入PyQt6
import matplotlib
import pandas as pd
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.backends.backend_qtagg import (
    NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

matplotlib.use("QtAgg")


class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
```

```
        super().__init__(fig)


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        # 创建 maptlotlib FigureCanvasQTAgg 对象，该对象定义了一组坐标轴，即 self.axes.
        sc = MplCanvas(self, width=5, height=4, dpi=100)
        # 创建一个包含简单数据和列名的 pandas DataFrame.
        df = pd.DataFrame(
            [
                [0, 10],
                [5, 15],
                [2, 20],
                [15, 25],
                [4, 10],
            ],
            columns=["A", "B"],
        )
        # p绘制 pandas DataFrame，传入 matplotlib Canvas 坐标轴.
        df.plot(ax=sc.axes)

        # 创建工具栏，将画布作为第一个参数传递，父窗口(self, the MainWindow)作为第二个参数.
        toolbar = NavigationToolbar(sc, self)

        layout = QtWidgets.QVBoxLayout()
        layout.addWidget(toolbar)
        layout.addWidget(sc)

        # 创建一个占位符控件来容纳我们的工具栏和画布.
        widget = QtWidgets.QWidget()
        widget.setLayout(layout)

        self.setCentralWidget(sc)
        self.show()


app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()
```

运行此代码后，您应看到以下窗口，其中显示了一个Pandas图表嵌入在PyQt6中，并配有Matplotlib工具栏

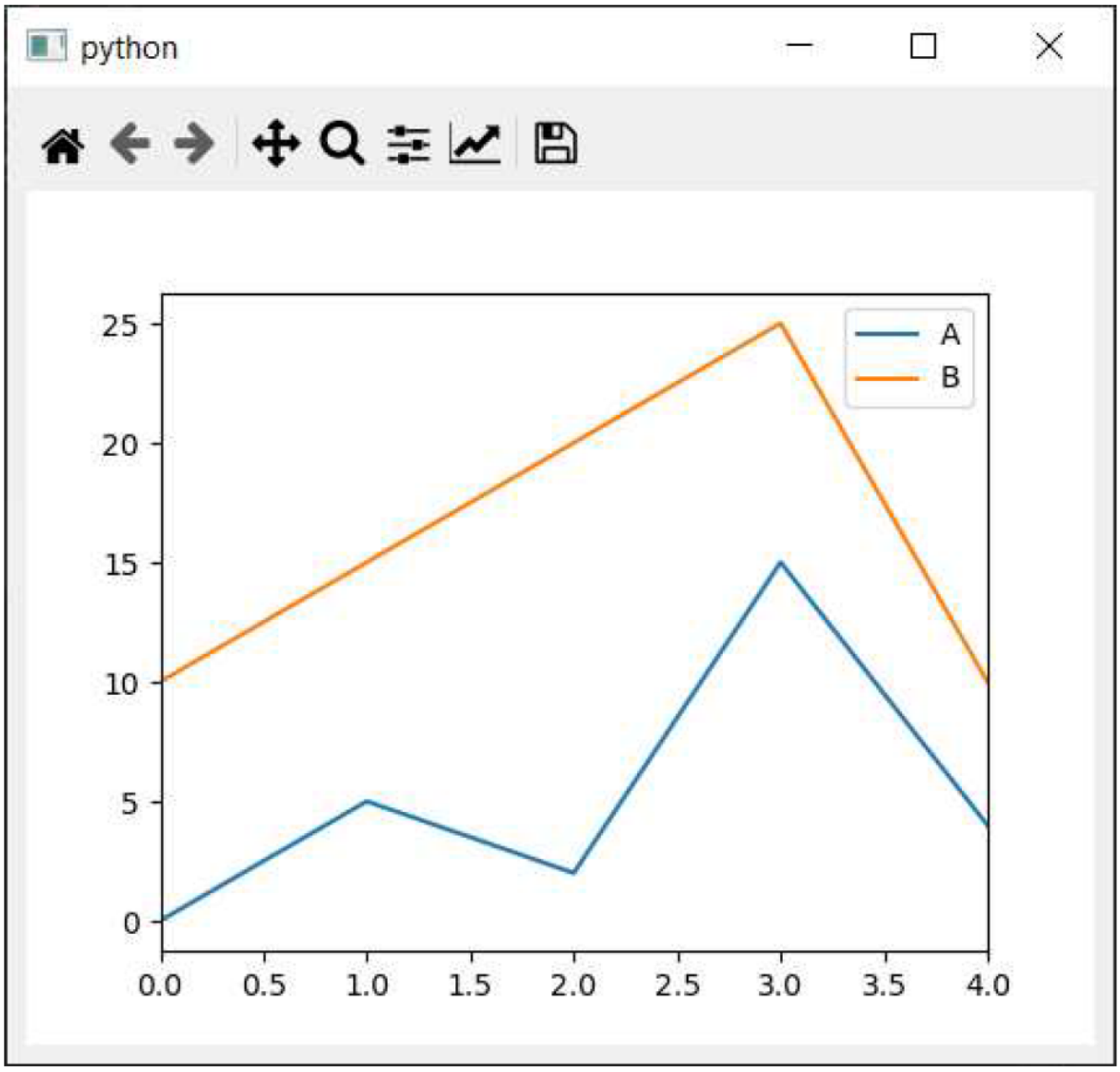


图233：使用matplotlib工具栏绘制 Pandas 图

接下来是什么?

在本章中，我们探讨了如何在 PyQt6 应用程序中嵌入 Matplotlib 图表。能够在应用程序中使用 Matplotlib 图表，使您能够从 Python 创建自定义数据分析和可视化工具。

Matplotlib 是一个庞大的库，内容过于丰富，无法在此详细展开。如果您对 Matplotlib 的绘图功能不熟悉，但想尝试使用，建议您查阅 相关文档 和 示例图表，以了解其功能范围。

PyQt6 的更多功能

到目前为止，我们所涵盖的主题已经足以使用PyQt6构建功能完善的桌面应用程序。在本章中，我们将探讨Qt框架中一些更具技术性且不太为人所知的方面，以加深对系统运作原理的理解。对于许多应用程序而言，本章所涉及的主题并非必要，但它们是工具箱中值得拥有的资源，以便在需要时随时调用！

31. 计时器

在应用程序中，您经常需要定期执行某些任务，甚至只是在未来某个时间点执行。在 PyQt6 中，这是通过使用定时器来实现的。`QTimer` 类为您提供两种不同类型的定时器——循环定时器或间隔定时器，以及单次定时器或一次性定时器。这两种定时器都可以与应用程序中的函数和方法关联，使其在需要时执行。在本章中，我们将探讨这两种定时器类型，并演示如何使用它们来自动化您的应用程序。

间隔计时器

使用 `QTimer` 类，您可以创建任何持续时间（以毫秒为单位）的间隔计时器。在每个指定的时间段，计时器都会超时。为了触发每次发生时都会发生的事情，您可以将计时器的超时信号连接到您想要执行的任何操作——就像处理其他信号一样。

在下面的示例中，我们设置了一个定时器，每 100 毫秒运行一次，该定时器旋转一个刻度盘。

Listing 232. further/timers_1.py

```
import sys

from PyQt6.QtCore import QTimer
from PyQt6.QtWidgets import QApplication, QDial, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.dial = QDial()
        self.dial.setRange(0, 100)
        self.dial.setValue(0)

        self.timer = QTimer()
        self.timer.setInterval(10)
        self.timer.timeout.connect(self.update_dial)
        self.timer.start()

        self.setCentralWidget(self.dial)

    def update_dial(self):
        value = self.dial.value()
        value += 1 # 递增
        if value > 100:
            value = 0
        self.dial.setValue(value)


app = QApplication(sys.argv)
w = MainWindow()
w.show()

app.exec()
```

这只是一个简单的例子——您可以在连接的方法中做任何您想做的事情。但是，标准事件循环规则仍然适用，触发任务应快速返回，以避免阻塞图形用户界面。如果您需要执行定期的长期任务，可以使用计时器触发一个单独的线程或进程。

> 您**必须**在计时器运行期间始终保留对创建的计时器对象的引用。如果未保留引用，计时器对象将被删除，计时器将停止运行——且不会有任何警告。如果您创建了计时器但它似乎无法正常工作，请检查是否已保留对该对象的引用。

如果计时器的精度很重要，您可以通过将一个 `Qt.QTimerType` 值传递给 `timer.setTimerType` 来调整它。

Listing 233. further/timers_1b.py

```
        self.timer.setTimerType(Qt.TimerType.PreciseTimer)
```

可用的选项如下所示。不要让计时器比实际需要的更精确，否则可能会阻塞重要的 UI 更新。

计时器类型	值	描述
`Qt.TimerType.PreciseTimer`	0	精准计时器力求保持毫秒级精度
`Qt.TimerType.CoarseTimer`	1	粗略计时器试图将精度保持在目标间隔的5%以内
`Qt.TimerType.VeryCoarseTimer`	2	非常粗糙的计时器仅能保持整秒精度

请注意，即使是最精确的计时器也只能保持毫秒级的精度。图形用户界面线程中的任何内容都可能被 UI 更新和您自己的 Python 代码阻塞。如果精度非常重要，请将工作放在另一个线程或您完全控制的进程中。

单次计时器

如果您想触发某个操作，但只希望它发生一次，可以使用单次触发定时器。这些定时器是通过 `QTimer` 对象的静态方法构建的。最简单的形式只需接受一个以毫秒为单位的时间参数，以及您希望在定时器触发时调用的可调用对象——例如，您希望运行的方法。

在下面的示例中，我们使用单次计时器在按下可切换的按钮后取消其选中状态。

Listing 234. further/timers_2.py

```
import sys

from PyQt6.QtCore import QTimer
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.button = QPushButton("Press me!")
        self.button.setCheckable(True)
        self.button.setStyleSheet(
```

```
            # 将复选框状态设置为红色，以便更容易辨识.
            "QPushButton:checked { background-color: red; }"
        )
        self.button.toggled.connect(self.button_checked)

        self.setCentralWidget(self.button)

    def button_checked(self):
        print("Button checked")
        QTimer.singleShot(1000, self.uncheck_button) #1

    def uncheck_button(self):
        print("Button unchecked")
        self.button.setChecked(False)


app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

1. `uncheck_button` 方法将在 1000 毫秒后被调用。

运行此示例并按下按钮后，您会看到按钮被选中并变为红色——这是使用了自定义样式。一秒钟后，按钮将恢复为未选中状态。

为了实现这一点，我们使用单次计时器将两个自定义方法链接在一起。首先，我们将按钮的切换信号连接到方法 `button_checked`。这会触发单次计时器。当计时器超时时，它会调用 `uncheck_button`，该方法实际上会取消选中该按钮。这使我们能够将取消选中该按钮的时间推迟到可配置的时间。

与间隔定时器不同，您无需保留创建的定时器（QTimer）的引用 —— `QTimer.singleShot()` 方法不会返回定时器引用。

通过事件队列进行延迟处理

您可以使用零延迟单次定时器通过事件队列延迟操作。当定时器触发时，定时器事件会被添加到事件队列的末尾（因为它是新事件），并且只有在所有现有事件都被处理完毕后才会被处理。

请记住，信号（和事件）只有在您将控制权从 Python 返回事件循环后才会被处理。如果您在一个方法中触发了一系列信号，并且希望在它们发生后执行某些操作，则不能直接在同一个方法中执行。该方法中的代码将在信号生效之前被执行。

```
def my_method(self):
    self.some_signal.emit()
    self.some_other_signal.emit()
    do_something_here() #1
```

1. 该函数将在两个信号生效之前执行。

通过使用单次计时器，您可以将后续操作推送到事件队列的末尾，并确保它最后执行。

```
def my_method(self):
    self.some_signal.emit()
    self.some_other_signal.emit()
    QTimer.singleShot(0, do_something_here) #1
```

1. 这将在信号的效果执行之后执行。

此技术仅保证 `do_something_here` 函数在先前的信号之后执行，而不保证这些信号的任何下游效果。不要试图通过增加 `msecs` 的值来解决这个问题，因为这会使您的应用程序依赖于系统计时。

32. 自定义信号

我们已经对信号进行了基本介绍，但这只是冰山一角。在本章中，我们将探讨如何创建自己的信号并自定义随信号发送的数据。

定制信号

到目前为止，我们只看了 Qt 本身在内置控件上提供的信号。不过，您也可以在自己的代码中使用自定义信号。这是将应用程序的模块部分解耦的好方法，这意味着应用程序的部分可以响应其他地方发生的事情，而无需了解应用程序的结构。

需要将应用程序的各个部分分离的一个明显迹象是，使用 `.parent()` 来访问其他无关控件上的数据。但它也适用于任何通过其他对象引用对象的地方，例如：`self.my_other_window.dialog.some_method`。此类代码在修改或重构应用程序时，容易在多个地方出现故障。尽可能避免使用此类代码！

将这些更新放入事件队列中，还可以帮助您保持应用程序的响应性——与使用一个大型更新方法相比，您可以将工作分成多个槽方法，并使用一个信号触发所有这些方法。

您可以使用 PyQt6 提供的 `pyqtSignal` 方法定义自己的信号。信号被定义为类属性，传递 Python 类型（或类型），这些类型将随信号一起发出。您可以为信号选择任何有效的 Python 变量名，并为信号类型选择任何 Python 类型。

Listing 235. further/signals_custom.py

```
import sys

from PyQt6.QtCore import pyqtSignal
from PyQt6.QtWidgets import QApplication, QMainWindow


class MainWindow(QMainWindow):
```

```
    message = pyqtSignal(str) #1
    value = pyqtSignal(int, str, int) #2
    another = pyqtSignal(list) #3
    onemore = pyqtSignal(dict) #4
    anything = pyqtSignal(object) #5

    def __init__(self):
        super().__init__()

        self.message.connect(self.custom_slot)
        self.value.connect(self.custom_slot)
        self.another.connect(self.custom_slot)
        self.onemore.connect(self.custom_slot)
        self.anything.connect(self.custom_slot)

        self.message.emit("my message")
        self.value.emit(23, "abc", 1)
        self.another.emit([1, 2, 3, 4, 5])
        self.onemore.emit({"a": 2, "b": 7})
        self.anything.emit(1223)

    def custom_slot(self, *args):
        print(args)


app = QApplication(sys.argv)
window = MainWindow()
window.show()

app.exec()
```

> 1. 发出字符串的信号。
> 2. 发出 3 种不同类型的信号。
> 3. 发出列表的信号。
> 4. 发出字典的信号。
> 5. 发出任何东西的信号。

如您所见，信号可以正常连接和发射。您可以发送任何 Python 类型，包括多种类型和复合类型（例如字典、列表）。

如果您将信号定义为 `pyqtSignal(object)`，它将能够传输任何绝对 Python 类型。但通常情况下，这并不是一个好主意，因为接收槽将需要处理所有类型

> 您可以对 QObject 的任何子类创建信号。这包括所有控件，包括主窗口和对话框。

修改信号数据

信号连接到槽，槽是每次信号触发时都会运行的函数（或方法）。许多信号还会传输数据，提供有关状态变化或触发信号的控件的信息。接收槽可以使用这些数据对同一信号执行不同的操作。

但是，有一个限制——信号只能发出其设计时指定的数据。以 `QPushButton.clicked` 信号为例，该信号在按钮被点击时触发。*clicked+* 信号会发出单个数据——按钮被点击后的 `_checked` 状态。

> 对于不可选中的按钮，此值始终为 `False`。

槽接收这些数据，但仅此而已。它不知道是哪一个控件触发了它，也不知道该控件的任何信息。通常情况下，这没毛病。您可以将一个特定的控件与一个独特的功能绑定，该功能可以精确地完成该控件所需的操作。但有时，您可能希望添加额外的数据，以便槽方法能够更智能一些。有一个巧妙的小技巧可以做到这一点。

您发送的附加数据可以是触发控件本身，也可以是槽执行信号预期结果所需的一些相关元数据。

拦截信号

您无需将信号直接连接到目标槽函数，而是使用一个中间函数来拦截信号，修改信号数据，然后将数据转发到目标槽。如果您在能够访问发出信号的控件的上下文中定义中间函数，则可以将该函数与信号一起传递。

此槽函数必须接受信号发送的值（此处为检查状态），然后调用实际槽，并将任何附加数据作为参数传递。

```
def fn(checked):
    self.button_clicked(checked, <additional args>)
```

与其像这样定义这个中间函数，您也可以使用 `lambda` 函数在线实现相同的功能。如上所述，它接受一个参数 `checked`，然后调用真正的槽。

```
lambda checked: self.button_clicked(checked, <additional args>)
```

在这两个示例中，`<additional args>` 可以替换为任何您想要转发到槽的内容。在下面的示例中，我们将 `QPushButton` 对象的 `action` 转发到接收槽。

```
btn = QPushButton()
btn.clicked.connect( lambda checked: self.button_clicked(checked, btn))
```

我们的 `button_clicked` 槽方法将接收原始的选中值和 `QPushButton` 对象。我们的接收槽可能看起来像这样

```
# 类方法.
def button_clicked(self, checked, btn):
    # 在此处执行操作.
```

> 您可以重新排列中间函数中的参数顺序，如果您喜欢的话

以下示例展示了实际应用情况，我们的 `button_clicked` 槽接收检查状态和控件对象。在此示例中，我们在处理程序中隐藏了按钮，因此您无法再次点击它！

Listing 236. further/signals_extra_1.py

```
import sys

from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        btn = QPushButton("Press me")
        btn.setCheckable(True)
        btn.clicked.connect(
            lambda checked: self.button_clicked(checked, btn)
        )

        self.setCentralWidget(btn)

    def button_clicked(self, checked, btn):
        print(btn, checked)
        btn.hide()


app = QApplication(sys.argv)

window = MainWindow()
window.show()
app.exec()
```

循环问题

以这种方式连接信号的一个常见原因是，当您在循环中构建一系列控件并通过编程连接信号时。遗憾的是，事情并不总是那么简单。

如果您在循环中构建了拦截信号，并希望将循环变量传递给接收槽，您会遇到一个问题。例如，在下面的示例中，我们创建了一系列按钮，并尝试将序列号与信号一起传递。点击按钮应使用按钮的值更新标签

Listing 237. further/signals_extra_2.py

```
import sys
```

```
from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLabel,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        v = QVBoxLayout()
        h = QHBoxLayout()

        for a in range(10):
            button = QPushButton(str(a))
            button.clicked.connect(
                lambda checked: self.button_clicked(a)
            ) #1
            h.addWidget(button)
        v.addLayout(h)
        self.label = QLabel("")
        v.addWidget(self.label)
        w = QWidget()
        w.setLayout(v)
        self.setCentralWidget(w)

    def button_clicked(self, n):
        self.label.setText(str(n))


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

1. 我们在 `lambda` 表达式中接受 `check` 变量，但会丢弃它。此按钮不可选中，因此它将始终为 `False`。

如果您运行这个程序，您应该会看到问题——无论您点击哪个按钮，标签上显示的都是相同的数字（9）。为什么是9？因为它是循环的最后一个值。

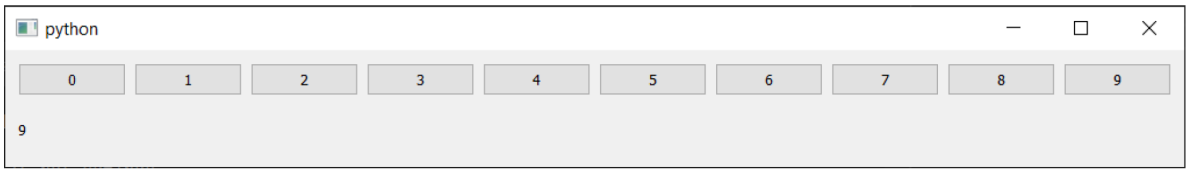


图234：无论您按下哪个按钮，标签上始终显示9。

问题就在这里——

```
for a in range(10):
    button = QPushButton(str(a))
    button.clicked.connect(
        lambda checked: self.button_clicked(a)
    )
```

问题出在行 `lambda: self.button_clicked(a)` 上，我们在这里定义了对最终槽的调用。这里我们传递了 `a`，但它仍然与循环变量绑定。当 `lambda` 被评估时（当信号触发时），`a` 的值将是它在循环结束时的值，因此点击任何一个都会导致发送相同的值（这里为 9）。

解决方案是将值作为命名参数传递。通过这种方式，值在 `lambda` 表达式创建时就被绑定，并且在循环的每次迭代中都会保持 `a` 的值。这样可以确保在每次调用时都使用正确的值。

> 如果这听起来像天书，别担心！只需记住，在使用中间函数时，始终使用命名参数。

```
lambda checked, a=a: self.button_clicked(a)
```

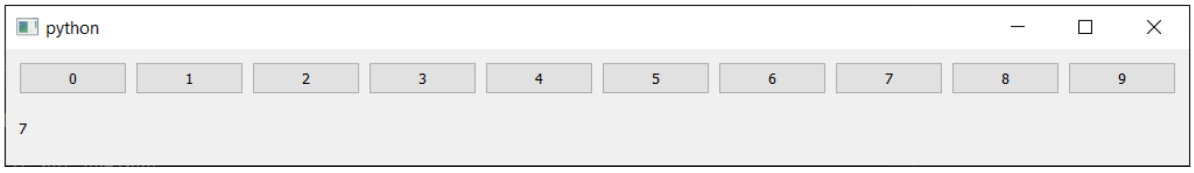
> 您不必使用相同的变量名，您可以使用 `lambda val=a: self.button_clicked(val)` 如果更喜欢这样。关键是使用命名参数。

将此内容放入我们的循环中，效果如下：

Listing 238. further/signals_extra_3.py

```
        for a in range(10):
        button = QPushButton(str(a))
        button.clicked.connect(
            lambda checked, a=a: self.button_clicked(a)
        ) #1
        h.addWidget(button)
```

如果您现在运行此操作，您将看到预期行为——单击按钮时，标签中将显示正确值。



> 图235：当您按下按钮时，所按的数字会显示在下方。

以下是使用内联 `lambda` 函数修改 `MainWindow.windowTitleChanged` 信号发送的数据的几个示例。当到达 `.setWindowTitle` 行时，它们都会触发，`my_custom_fn` 槽将输出它们接收到的内容。

Listing 239. further/signals_extra_4.py

```
import sys

from PyQt6.QtWidgets import QApplication, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        # SIGNAL: 每当窗口标题发生改变时，将调用连接的功能。新标题将传递给该函数.
        self.windowTitleChanged.connect(self.on_window_title_changed)
        # SIGNAL: 每当窗口标题发生变化时，都会调用连接的功能。新标题在lambda中被丢弃，该功能在没有参数的情况下被调用.
        self.windowTitleChanged.connect(lambda x: self.my_custom_fn())
        # SIGNAL: 当窗口标题发生变化时，将调用该连接函数。新标题将作为参数传递给该函数，并替换默认参数.
        self.windowTitleChanged.connect(lambda x: self.my_custom_fn(x))
        # SIGNAL: 当窗口标题发生变化时，将调用该连接函数。新标题将传递给该函数并替换默认参数。额外数据将从lambda内部传递.
        self.windowTitleChanged.connect(
            lambda x: self.my_custom_fn(x, 25)
        )
        # 这将设置窗口标题，该标题将触发所有上述信号，将新标题作为第一个参数发送给附加函数或lambda表达式.
        self.setWindowTitle("This will trigger all the signals.")
        # SLOT: 该函数接受一个字符串（例如窗口标题），并将其打印出来.
    def on_window_title_changed(self, s):
        print(s)
        # SLOT: 该函数具有默认参数，因此可以不传入值直接调用.
    def my_custom_fn(self, a="HELLLO!", b=5):
        print(a, b)


app = QApplication(sys.argv)

window = MainWindow()
window.show()
app.exec()
```

33. 使用相对路径

路径描述了文件在文件系统中的位置。

当我们将外部数据文件加载到应用程序中时，通常会使用路径来完成这一操作。虽然从理论上讲这很简单，但实际操作中可能会遇到一些问题。随着应用程序规模的扩大，维护这些路径可能会变得有些繁琐，因此值得退一步考虑实施一个更可靠的系统。

相对路径

路径有两种类型——绝对路径和相对路径。绝对路径描述了从文件系统根目录（底部）开始的完整路径，而相对路径则描述了从当前文件系统位置开始的路径（或相对于当前位置的路径）。

这并不明显，但当您仅提供文件名时，例如：`hello.jpg`，这实际上是一个相对路径。当文件被加载时，它是相对于当前活动文件夹加载的。令人困惑的是，当前活动文件夹并不一定就是您的脚本所在的文件夹。

在"控件"一章中，我们介绍了一种处理加载图像时出现此问题的简单方法。我们使用内置函数 `__file__` 获取当前正在运行的脚本（我们的应用程序）的路径，然后使用 `os` 函数首先获取脚本的目录，再用该目录构建完整路径。

Listing 240. basic/widgets_2b.py

```
import os
import sys

from PyQt6.QtGui import QPixmap
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow

basedir = os.path.dirname(__file__)
print("Current working folder:", os.getcwd()) #1
print("Paths are relative to:", basedir) #2


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        widget = QLabel("Hello")
        widget.setPixmap(QPixmap(os.path.join(basedir, "otje.jpg")))

        self.setCentralWidget(widget)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

对于简单应用程序而言，这种方法效果良好，尤其是当您仅有一个主脚本且需要加载的文件较少时。然而，在加载每个文件时都需要重复计算基目录，并使用 `os.path.join` 方法在各处构建路径，这很快就会变成维护噩梦。如果您需要重新组织项目中的文件结构，那将是一场噩梦。幸运的是，有一种更简单的方法！

> 为什么不直接使用绝对路径呢？因为它们只适用于您的文件系统，或者结构完全相同的文件系统。如果我在自己的主目录中开发应用程序，并使用绝对路径来引用文件，例如：`/home/martin/myapp/images/somefile.png`，那么它只会在其他也拥有名为 `martin` 的主目录并将其放置在该目录下的人的系统上生效。这会有点奇怪。

使用 Paths 类

应用程序需要加载的数据文件通常具有一定的结构——加载的文件类型通常较为常见，或者加载这些文件的目的是为了实现常见的功能。通常，您会将相关的文件存储在相关的文件夹中，以便于管理。我们可以利用这种现有的结构，建立一种常规的方法来构建文件的路径。

要实现这一点，我们可以创建一个自定义的 `Paths` 类，该类通过结合使用属性与方法来分别构建文件夹和文件路径。其核心逻辑与上述使用的 `os.path.dirname(__file__)` 和 `os.path.join()` 方法相同，但具有更高的自包含性且易于修改。

现在请您将以下代码添加到项目根目录下的一个名为 `paths.py` 的文件中。

Listing 241. further/paths.py

```
import os


class Paths:

    base = os.path.dirname(__file__)
    ui_files = os.path.join(base, "ui")
    images = os.path.join(base, "images")
    icons = os.path.join(images, "icons")
    data = os.path.join(base, "images")

    # 文件加载器.
    @classmethod
    def ui_file(cls, filename):
        return os.path.join(cls.ui_files, filename)

    @classmethod
    def icon(cls, filename):
        return os.path.join(cls.icons, filename)

    @classmethod
    def image(cls, filename):
        return os.path.join(cls.images, filename)

    @classmethod
    def data(cls, filename):
        return os.path.join(cls.data, filename)
```

> 要尝试使用 paths 模块，您可以启动一个 Python 解释器，位于您的项目根目录下，并使用 `from paths import Paths`

现在，在应用程序的任何位置，您都可以导入 Paths 类并直接使用它。属性 `base`、`ui_files`、`icons`、`images` 和 `data` 均返回其对应文件夹在 `base` 文件夹下的路径。请注意 `icons` 文件夹是如何从 `images` 路径构建的——将该文件夹嵌套在该路径下。

> 您可以自由地自定义路径的名称和结构等，以匹配您自己项目中的文件夹结构。

```
>>> from paths import Paths
>>> Paths.ui_files
'U:\\home\\martin\\books\\create-simple-gui-applications\\code\\further\\ui'
>>> Paths.icons
'U:\\home\\martin\\books\\create-simple-gui-
applications\\code\\further\\images\\icons'
```

> 我们**不会**从这个类创建对象实例——我们不会调用 `Paths()`——因为我们不需要它。路径是静态且不变的，因此无需通过创建对象来管理内部状态。注意，方法必须使用 `@classmethod` 装饰器才能在类本身访问。

方法 `ui_file`、`icon`、`image` 和 `data` 用于生成包含文件名的路径。在每种情况下，您都需要调用该方法并传入要添加到路径末尾的文件名。这些方法均依赖于上述描述的文件夹属性。例如，如果您想加载特定的图标，可以调用 `Paths.icon()` 方法，传入图标名称，以获取完整的路径。

```
>>> Paths.icon('bug.png')
'U:\\home\\martin\\books\\create-simple-gui-
applications\\code\\further\\images\\icons\\bug.png'
```

在您的应用程序代码中，您可以按照以下方式构建路径并加载图标：

```
QIcon(Paths.icon('bug.png'))
```

这使得您的代码更加整洁，有助于确保路径正确，并且如果您以后想重新组织文件的存储方式，会变得更加容易。例如，假设您想将图标移动到顶级文件夹：现在您只需修改 `paths.py` 中的定义，所有图标仍可正常工作。

```
    icons = os.path.join(images, 'icons')
    # 要提升到顶层，请让图标从基类继承。
    icons = os.path.join(base, 'icons')
```

34. 系统托盘与 macOS 菜单

系统托盘应用程序（或菜单栏应用程序）可用于通过少量点击即可访问常见功能。对于完整的桌面应用程序，它们是控制应用程序的便捷方式，无需打开整个窗口。

Qt 提供了一个简单的接口，用于构建跨平台的系统托盘（Windows）或菜单栏（macOS）应用程序。以下是一个最小的可工作示例，用于在工具栏/系统托盘中显示一个图标并附带一个菜单。菜单中的操作尚未连接，因此目前还不会执行任何操作。

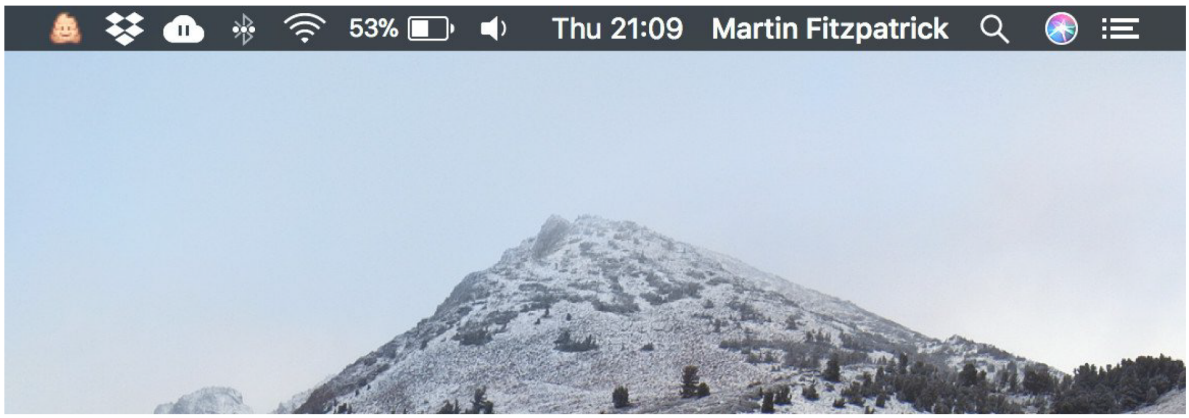
Listing 242. further/systray.py

```
import os
import sys

from PyQt6.QtGui import QIcon
from PyQt6.QtWidgets import (
    QAction,
    QApplication,
    QColorDialog,
    QMenu,
    QSystemTrayIcon,
)

basedir = os.path.dirname(__file__)

app = QApplication(sys.argv)
app.setQuitOnLastWindowClosed(False)

# 创建图标
icon = QIcon(os.path.join(basedir, "icon.png"))

# 创建托盘
tray = QSystemTrayIcon()
tray.setIcon(icon)
tray.setVisible(True)

# 创建菜单
menu = QMenu()
action = QAction("A menu item")
menu.addAction(action)

# 在菜单中加入退出选项.
quit = QAction("Quit")
quit.triggered.connect(app.quit)
menu.addAction(quit)

# 将菜单添加到托盘
tray.setContextMenu(menu)

app.exec()
```

您会发现这里没有 `QMainWindow`，因为我们根本没有窗口需要显示。您可以像往常一样创建一个窗口，而不会影响系统托盘图标的行为。

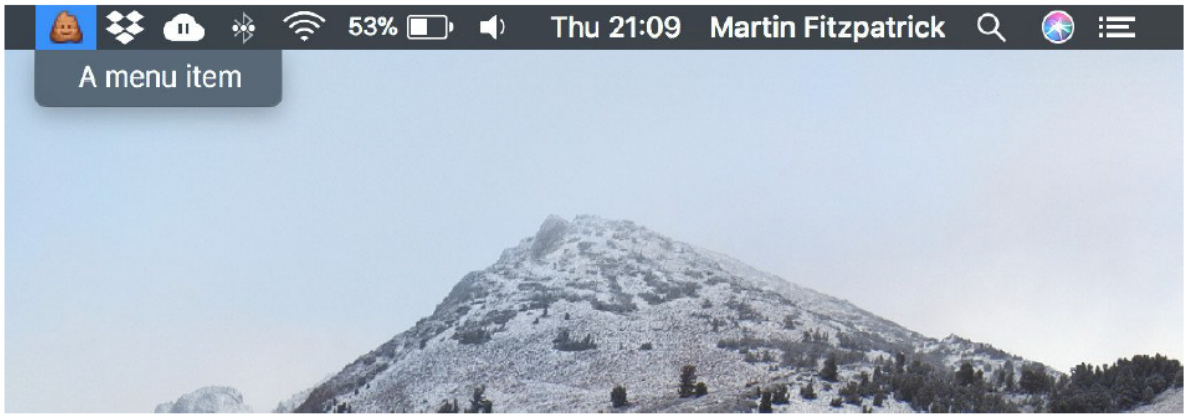
> Qt 的默认行为是在所有活动窗口关闭后关闭应用程序。这不会影响这个示例，但在创建窗口后再关闭它们的应用程序中会成为问题。您可以设置 `app.setQuitOnLastWindowClosed(False)` 可阻止此行为，并确保应用程序继续运行。

提供的图标会显示在工具栏中（您可以在系统托盘或菜单栏右侧的图标组左侧看到它）。



> 图236：菜单栏上显示的图标

单击（或在 Windows 上右键单击）图标会显示添加的菜单。



> 图237：菜单栏应用程序菜单

目前这个应用程序还什么都不做，所以在接下来的部分，我们将扩展这个示例，创建一个小型颜色选择器。

以下是一个更完整的示例，使用Qt内置的 `QColorDialog` 来提供一个可通过工具栏访问的颜色选择器。该菜单允许您选择以HTML格式 `#RRGGBB` 、 `rgb(R,G,B)` 或 `hsv(H,S,V)` 格式获取所选颜色。

ing 243. further/systray_color.py

```
import os
import sys
```

```
from PyQt6.QtGui import QIcon
from PyQt6.QtWidgets import (
    QAction,
    QApplication,
    QColorDialog,
    QMenu,
    QSystemTrayIcon,
)

basedir = os.path.dirname(__file__)


app = QApplication(sys.argv)
app.setQuitOnLastWindowClosed(False)

# 创建图标
icon = QIcon(os.path.join(basedir, "color.png"))

clipboard = QApplication.clipboard()
dialog = QColorDialog()


def copy_color_hex():
    if dialog.exec():
        color = dialog.currentColor()
        clipboard.setText(color.name())


def copy_color_rgb():
    if dialog.exec():
        color = dialog.currentColor()
        clipboard.setText(
            "rgb(%d, %d, %d)"
            % (color.red(), color.green(), color.blue())
        )


def copy_color_hsv():
    if dialog.exec():
        color = dialog.currentColor()
        clipboard.setText(
            "hsv(%d, %d, %d)"
            % (color.hue(), color.saturation(), color.value())
        )


# 创建托盘
tray = QSystemTrayIcon()
tray.setIcon(icon)
tray.setVisible(True)

# 创建菜单
menu = QMenu()
action1 = QAction("Hex")
action1.triggered.connect(copy_color_hex)
menu.addAction(action1)
```

```
action2 = QAction("RGB")
action2.triggered.connect(copy_color_rgb)
menu.addAction(action2)

action3 = QAction("HSV")
action3.triggered.connect(copy_color_hsv)
menu.addAction(action3)

quit = QAction("Quit")
quit.triggered.connect(app.quit)
menu.addAction(quit)

# 将菜单添加到托盘中
tray.setContextMenu(menu)

app.exec()
```

与前一个示例类似，本示例中也没有 `QMainWindow`。菜单的创建方式与之前相同，但新增了 3 个用于不同输出格式的操作项。每个操作项都与代表其格式的特定处理函数相连。每个处理函数会显示一个对话框，如果用户选择了颜色，则将该颜色以指定格式复制到剪贴板中。

与之前一样，该图标会出现在工具栏中。

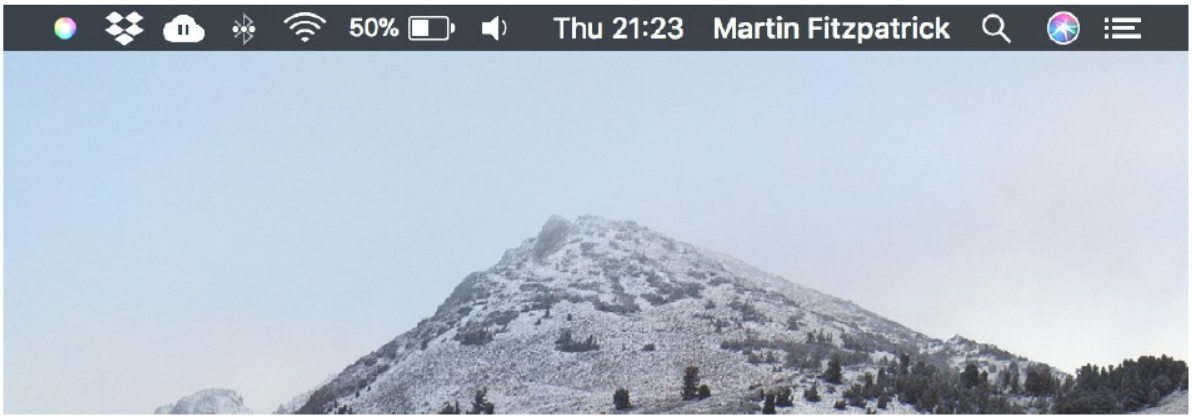


图238：工具栏上的颜色选择器

点击图标会显示一个菜单，从中您可以选择要返回的图像格式。

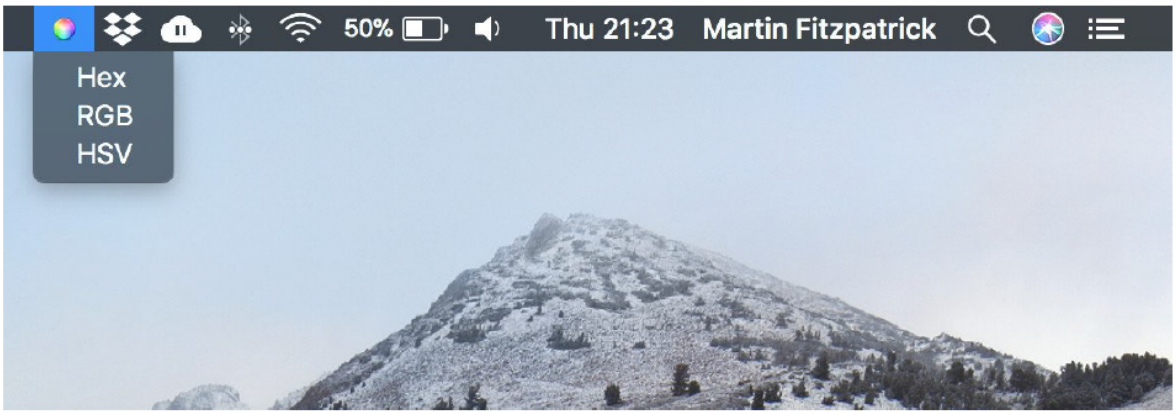


图239：颜色选择器菜单

选择格式后，您将看到标准的Qt颜色选择器窗口。

图240：系统颜色选择器窗口

选择您想要的颜色并点击确定。所选颜色将以您请求的格式复制到剪贴板。可用的格式将产生以下输出：

值	范围
`#a2b3cc`	00-FF
`rgb(25, 28, 29)`	0-255
`hsv(14, 93, 199)`	0-255

为完整应用程序添加系统托盘图标

到目前为止，我们已经展示了如何创建一个没有主窗口的独立系统托盘应用程序。然而，有时您可能希望同时拥有一个系统托盘图标和一个窗口。当这样做时，通常可以通过托盘图标打开和关闭（隐藏）主窗口，而无需关闭应用程序。在本节中，我们将探讨如何使用Qt5构建此类应用程序。

原则上来说，这非常简单——创建主窗口，并将一个动作的信号连接到窗口的 `.show()` 方法。

以下是一个名为“PenguinNotes”的小型便签应用程序。运行时，它会在系统托盘或 macOS 工具栏中显示一个小企鹅图标。

点击托盘中的小企鹅图标将显示窗口。该窗口包含一个 `QTextEdit` 编辑器，您可以在其中输入笔记。您可以像往常一样关闭该窗口，或再次点击托盘图标关闭。应用程序将持续在托盘中运行。要关闭应用程序，您可以使用“File › Close”关闭”选项，关闭时将自动保存笔记。

Listing 244. further/systray_window.py

```
import os
import sys

from PyQt6.QtGui import QIcon
from PyQt6.QtWidgets import (
    QAction,
    QApplication,
    QMainWindow,
    QMenu,
    QSystemTrayIcon,
    QTextEdit,
)

basedir = os.path.dirname(__file__)

app = QApplication(sys.argv)
app.setQuitOnLastWindowClosed(False)

# 创建图标
icon = QIcon(os.path.join(basedir, "animal-penguin.png"))

# 创建托盘
tray = QSystemTrayIcon()
tray.setIcon(icon)
tray.setVisible(True)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
```

```
        self.editor = QTextEdit()
        self.load() # Load up the text from file.

        menu = self.menuBar()
        file_menu = menu.addMenu("&File")

        self.reset = QAction("&Reset")
        self.reset.triggered.connect(self.editor.clear)
        file_menu.addAction(self.reset)

        self.quit = QAction("&Quit")
        self.quit.triggered.connect(app.quit)
        file_menu.addAction(self.quit)

        self.setCentralWidget(self.editor)

        self.setWindowTitle("PenguinNotes")

    def load(self):
        with open("notes.txt", "r") as f:
            text = f.read()
        self.editor.setPlainText(text)

    def save(self):
        text = self.editor.toPlainText()
        with open("notes.txt", "w") as f:
            f.write(text)

    def activate(self, reason):
        if (
            reason == QSystemTrayIcon.ActivationReason.Trigger
        ): # 图标被点击.
            self.show()


w = MainWindow()

tray.activated.connect(w.activate)
app.aboutToQuit.connect(w.save)

app.exec()
```

> 在 macOS 上，“退出”操作会出现在应用程序菜单中（位于最左侧，与应用程序名称并列），而非文件菜单。如果我们没有同时添加“File › Reset”操作，文件菜单将为空且隐藏（您不妨试试看！）

以下是笔记应用程序的屏幕截图，窗口处于打开状态。

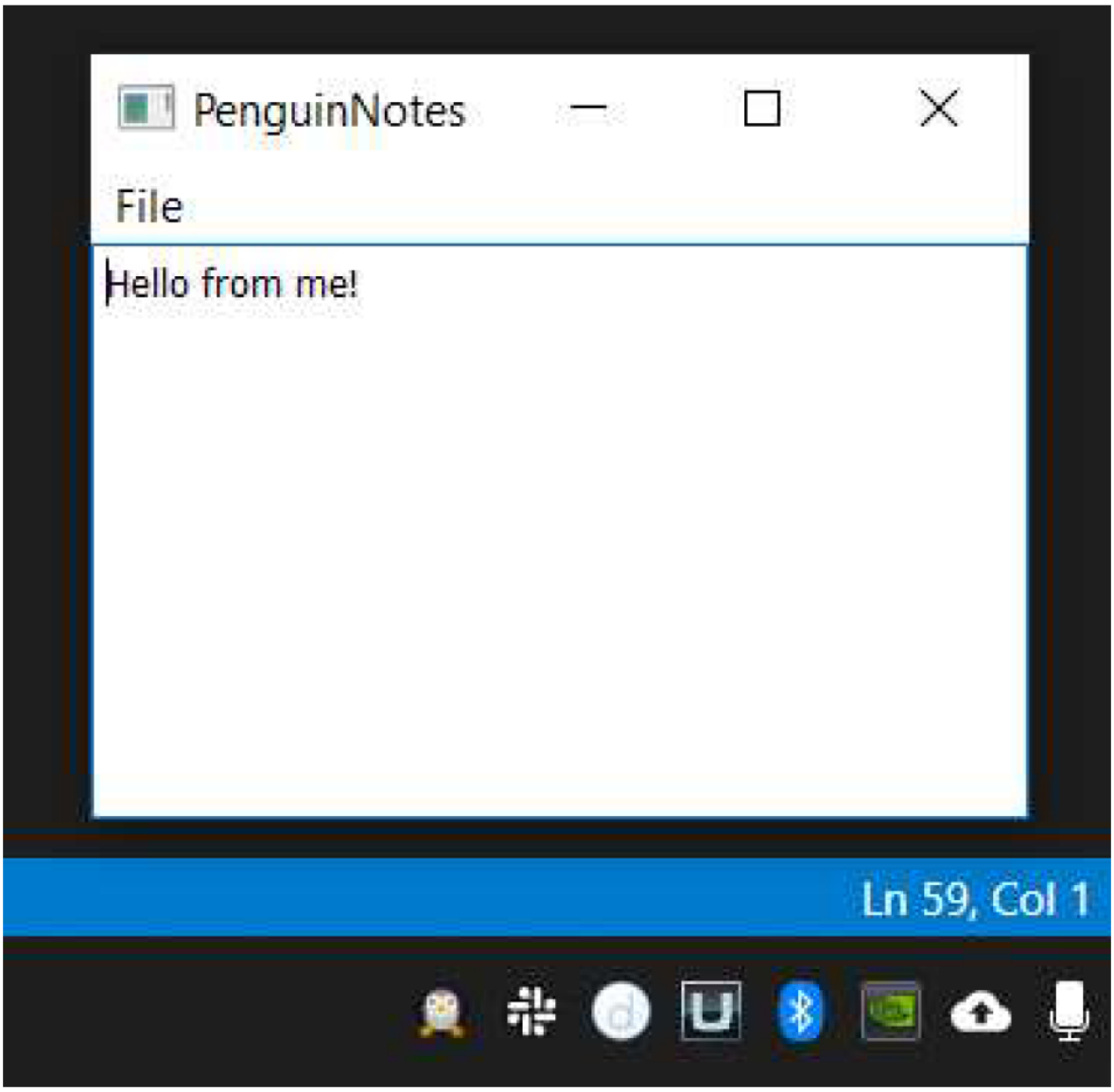


图241：笔记编辑器窗口

显示和隐藏窗口的控制通过我们的 `QMainWindow` 上的 `activate` 方法滑块来实现。这与代码底部的托盘图标 `.activated` 信号相连，使用 `tray.activated.connect(w.activate)` 实现。

```
    def activate(self, reason):
        if reason == QSystemTrayIcon.Trigger: # 图标被点击.
            if self.isVisible():
                self.hide()
            else:
                self.show()
```

该信号在许多不同情况下都会被触发，因此我们必须首先检查以确保我们只使用 `QSystemTrayIcon.Trigger`。

原因	值	描述
`QSystemTrayIcon.Unknown`	0	未知原因
`QSystemTrayIcon.Context`	1	上下文菜单请求（macOS为单击，Windows为右键单击）

原因	值	描述
`QSystemTrayIcon.DoubleClick`	2	图标被双击。在 macOS 上，双击仅在未设置上下文菜单时触发，因为菜单在单击时打开。
`QSystemTrayIcon.Trigger`	3	图标被单击一次
`QSystemTrayIcon.MiddleClick`	4	图标被鼠标中键点击

通过监听这些事件，您应该能够构建任何类型的系统托盘行为。然而，请务必在所有目标平台上测试该行为。

35. 枚举与 Qt 命名空间

当您在应用程序中看到类似以下的代码时，您可能会想知道 `Qt.ItemDataRole.DisplayRole` 或 `Qt.ItemDataRole.CheckStateRole` 对象究竟是什么。

> 在 PyQt 的早期版本中，也存在诸如 `Qt.DisplayRole` 之类的快捷名称，因此您在代码中仍可能看到这些名称。在 PyQt6 中，您必须始终使用完整名称。本书中的所有示例均使用完全限定名称。

```
    def data(self, role, index):

        if role == Qt.ItemDataRole.DisplayRole:
            # 做些什么
```

Qt 在代码中广泛使用这些类型来表示有意义的常量。其中许多在 Qt 命名空间中可用，即 `Qt.<something>`，尽管还有一些对象特定的类型，例如 `QDialogButtonBox.StandardButton.Ok`，它们的工作方式完全相同。

但它们是如何工作的呢？在本章中，我们将详细探讨这些常量是如何形成的，以及如何有效地使用它们。为此，我们需要涉及一些基础知识，如二进制数。但深入理解这些内容并非本章的必要条件——正如往常一样，我们将重点放在如何在学习过程中应用所学知识上。

这不过是一些数字而已

如果您查看一个标志的类型（`type()`），您会看到一个类的名称。这些类是该标志所属的组。例如，`Qt.ItemDataRole.DecorationRole` 的类型是 `Qt.ItemDataRole` —— 您可以在 Qt 文档中看到这些组。

> 您可以在 Python Shell 中运行以下代码，只需先使用 `from PyQt6.QtCore import Qt` 导入 Qt 命名空间即可。

```
>>> type(Qt.ItemDataRole.DecorationRole)
<enum 'ItemDataRole'>
```

这些类型是枚举类型——一种将值限制为一组预定义值的类型。在 PyQt6 中，它们被定义为 Python 枚举 (`Enum`) 类型。

这些值实际上都是简单的整数。`Qt.ItemDataRole.DisplayRole` 的值为 0，而 `Qt.ItemDataRole.EditRole` 的值为 2。这些整数值本身没有意义，但在它们被使用的特定上下文中具有意义。

```
>>> int(Qt.ItemDataRole.DecorationRole)
1
```

例如，您是否认为以下代码会评估为 True?

```
>>> Qt.ItemDataRole.DecorationRole == Qt.AlignmentFlag.AlignLeft
True
```

可能不是。但 `Qt.ItemDataRole.DecorationRole` 和 `Qt.AlignmentFlag.AlignLeft` 的整数值均为 1，因此在数值上是相等的。这些数值通常可以忽略。只要在适当的上下文中使用这些常量，它们就会始终按预期工作。

Table 8. Values given in the documentation can be in decimal or binary

标识符	值（十六进制）	值（十进制）	描述
`Qt.AlignmentFlag.AlignLeft`	`0x0001`	1	与左侧对齐
`Qt.AlignmentFlag.AlignRight`	`0x0002`	2	与右侧对齐
`Qt.AlignmentFlag.AlignHCenter`	`0x0004`	4	在可用空间中水平居中
`Qt.AlignmentFlag.AlignJustify`	`0x0008`	8	在可用空间内对齐文本。
`Qt.AlignmentFlag.AlignTop`	`0x0020`	32	与顶部对齐
`Qt.AlignmentFlag.AlignBottom`	`0x0040`	64	与底部对齐
`Qt.AlignmentFlag.AlignVCenter`	`0x0080`	128	在可用空间中垂直居中

标识符	值（十六进制）	值（十进制）	描述
`Qt.AlignmentFlag.AlignBaseline`	`0x0100`	256	与基准线对齐

如果您查看上表中的数字，可能会发现一些异常。首先，这些数字并非每次增加1，而是每次翻倍。其次，水平对齐的十六进制数字都集中在同一列，而垂直对齐的数字则分布在另一列。

这种数字模式是故意设计的，它使我们能够做一件非常巧妙的事情——将标志位组合在一起以创建复合标志位。要理解这一点，我们需要快速了解计算机如何表示整数。

二进制与十六进制

当我们进行常规计数时，使用的是十进制（基数为10）的数字系统。该系统包含10个数字，从0到9，且十进制数字中的每个数字的值是其前一个数字的10倍。例如，数字1251由1×1000、2×100、5×10和1×1组成。

1000	100	10	1
1	2	5	1

计算机以二进制形式存储数据，即一系列开和关的状态，以1和0的形式表示。二进制是一种以2为基数的数制。它有2个数字，从0到1，二进制数中的每个数字的值是前一个数字的2倍。在以下示例中，数字5由1×4和1×1组成。

8	4	2	1	十进制
0	1	0	1	5

二进制数书写起来很快就会变得繁琐——5893的二进制形式是 `1011100000101` ——但将其与十进制数来回转换也并不方便。为了更方便地处理二进制数，十六进制在计算机领域中被广泛使用。这是一种由16个数字（0-9A-F）组成的数制。每个十六进制数字的值在0-15（0-A）之间，相当于4个二进制位。这使得在两者之间进行转换变得直观。

下表显示了数字0-15，以及它们在二进制和十六进制中的对应值。一个给定的二进制数的值可以通过将每列顶部含有1的数字相加来计算。

8	4	2	1	十六进制	十进制
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7

8	4	2	1	十六进制	十进制
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	A	10
1	0	1	1	B	11
1	1	0	0	C	12
1	1	0	1	D	13
1	1	1	0	E	14
1	1	1	1	F	15

这种模式在更大的数字中继续适用。例如，下图是数字25的二进制表示，由16×1、8×1和1×1组成。

16	8	4	2	1
1	1	0	0	1

由于二进制值中的每个数字要么是1，要么是0（`True` 或 `False`），我们可以将二进制数字用作布尔标志——状态标记，这些标记要么处于开启状态，要么处于关闭状态。一个整数值可以存储多个标志，每个标志使用唯一的二进制数字。每个标志都会根据其设置为1的二进制数字的位置拥有自己的数值。

这就是 Qt 标志的工作原理。再次查看我们的对齐标志，我们现在可以理解为什么选择这些数字——每个标志都是一个唯一的、不重叠的位。标志的值来自标志设置为 1 的二进制位。

`Qt.AlignmentFlag.AlignLeft`	**1**	`00000001`
`Qt.AlignmentFlag.AlignRight`	2	`00000010`
`Qt.AlignmentFlag.AlignHCenter`	4	`00000100`
`Qt.AlignmentFlag.AlignJustify`	8	`00001000`
`Qt.AlignmentFlag.AlignTop`	32	`00100000`
`Qt.AlignmentFlag.AlignBottom`	64	`01000000`
`Qt.AlignmentFlag.AlignVCenter`	128	`10000000`

当直接使用 `==` 运算符测试这些标志时，您无需担心这些问题。但这种值的排列方式解锁了将标志组合在一起的能力，以创建复合标志，这些标志同时代表多个状态。这使您能够使用单个标志变量表示，例如，左对齐和底对齐。

位或运算（`|`）组合

任何两个二进制表示不重叠的数字都可以相加，同时保持其原有的二进制位不变。例如，下图中我们对1和2进行相加，得到3 —

Table 9. Add

`001`	**1**
`010`	+ 2
`011`	= 3

原始数字中的1位数在输出中得以保留。相比之下，如果我们将1和3相加得到4，原始数字中的1位数在结果中不存在——两者现在都是零。

`001`	**1**
`011`	+ 3
`100`	= 4

> 在十进制中也能观察到相同的效果——您可以比较将100和50相加得到150，与将161和50相加得到211的情况。

由于我们在特定二进制位置使用1值来表示某种含义，这会造成问题。例如，如果我们将对齐标志的值添加两次，我们将得到一个在数学上完全正确但意义上完全错误的结果。

Table 10. Add

`00000001`	**1**	`Qt.AlignmentFlag.AlignLeft`
`00000001`	+ 1	`+ Qt.AlignmentFlag.AlignLeft`
`00000010`	= 2	`= Qt.AlignmentFlag.AlignRight`

```
>>> Qt.AlignmentFlag.AlignLeft + Qt.AlignmentFlag.AlignLeft ==
Qt.AlignmentFlag.AlignRight
True
```

因此，在处理二进制标志时，我们使用位或运算将它们组合起来。在 Python 中，位或运算使用 `|`（管道）运算符实现。在位或运算中，我们通过在二进制级别比较两个数来将它们组合在一起。结果是一个新数，其中二进制位如果在任一输入中为 1，则设置为 1。但重要的是，位不被传递，也不影响相邻位。

> 当数字不重叠时，按位或运算等同于加法（+）。

`Qt.AlignmentFlag.AlignLeft`	`00000001`
`Qt.AlignmentFlag.AlignTop`	`00100000`

使用上述两个对齐常量，我们可以将它们的值结合起来，使用位或运算生成输出，以实现顶部左对齐。

Table 11. Bitwise OR

`00000001`	**1**	`Qt.AlignmentFlag.AlignLeft`
`00100000`	OR 32	`| Qt.AlignmentFlag.AlignTop`
`00100001`	= 33	`Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop`

```
>>> int(Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop)
33
```

因此，如果我们将 32 与 1 相加，结果是 33。这应该不会太令人惊讶。但是，如果我们不小心多次添加 `Qt.AlignmentFlag.AlignLeft`，会发生什么情况?

```
>>> int(Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignLeft |
Qt.AlignmentFlag.AlignTop)
33
```

我们得到了相同的结果！位或运算在二进制位置上输出1，只要输入中任何一个位置有1。它不会将它们相加，也不会将任何内容进位或溢出到其他位——这意味着您可以多次对同一个值进行位或运算，最终得到的只是您最初的值。

```
>>> int(Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignLeft |
Qt.AlignmentFlag.AlignLeft)
1
```

或者，用二进制表示——

Table 12. Bitwise OR

`00000001`	**1**	`Qt.AlignmentFlag.AlignLeft`
`00000001`	OR 1	`| Qt.AlignmentFlag.AlignLeft`
`00000001`	= 1	`= Qt.AlignmentFlag.AlignLeft`

最后，比较这些值。

```
>>> Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignLeft ==
Qt.AlignmentFlag.AlignLeft
True

>>> Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignLeft ==
Qt.AlignmentFlag.AlignRight
False
```

检查组合标志

我们可以直接比较标志本身来检查简单的标志，正如我们已经看到的——

```
>>> align = Qt.AlignmentFlag.AlignLeft
>>> align == Qt.AlignmentFlag.AlignLeft
True
```

对于组合标志，我们还可以检查与标志组合的相等性

```
>>> align = Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop
>>> align == Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop
True
```

但有时，您可能想知道某个变量是否包含特定标志。例如，我们可能想知道 `align` 是否设置了 `align left` 标志，而与其他对齐状态无关。

一旦与另一个元素合并后，如何判断一个元素是否应用了 `Qt.AlignmentFlag.AlignLeft` 属性？在这种情况下，使用 a `==` 比较不会生效，因为它们在数值上并不相等。

```
>> alignment = Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop
>> alignment == Qt.AlignmentFlag.AlignLeft # 33 == 1
False
```

我们需要一种方法来比较 `Qt.AlignmentFlag.AlignLeft` 标志与我们的复合标志的位。为此，我们可以使用位与运算。

位与运算（&）检查

在 Python 中，位与运算使用 & 运算符进行。

在上一步骤中，我们结合了 `Qt.AlignmentFlag.AlignLeft` (1) 和 `Qt.AlignmentFlag.AlignTop` (32) 以生成 “Top Left” (33)。现在，我们需要检查组合后的对齐标志是否设置了左对齐标志。为了进行测试，我们需要使用位与运算，该运算逐位检查两个输入值是否均为 1，如果为真，则在该位置返回 1。

Table 13. Bitwise AND

`00100001`	**33**	`Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop`
`00100001`	AND 1	`& Qt.AlignmentFlag.AlignLeft`
`00100001`	= 1	`= Qt.AlignmentFlag.AlignLeft`

这会过滤输入变量中的位，仅保留那些在目标标志 `Qt.AlignmentFlag.AlignLeft` 中设置的位。如果该位被设置，结果为非零值；如果未设置，结果为 0。

```
>>> int(alignment & Qt.AlignmentFlag.AlignLeft)
1 # 结果是标志的数值，这里是1.
```

例如，如果我们将对齐变量与 `Qt.AlignmentFlag.AlignRight` 进行测试，结果为 0

`00100001`	**33**	`Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop`
`00000010`	2	`& Qt.AlignmentFlag.AlignRight`
`00000000`	0	`= Qt.AlignmentFlag.AlignLeft`

```
>>> int(alignment & Qt.AlignmentFlag.AlignRight)
0
```

因为在 Python 中，0 等于 `False`，而其他任何值都等于 `True`。这意味着，当使用位与运算符对两个数字进行比较时，如果它们有任何位是相同的，结果将大于 0，并且为 `True`。

通过结合位运算的或运算和与运算，您应该能够利用Qt标志实现所需的所有功能。

36. 使用命令行参数

如果您创建了一个与特定文件类型配合使用的应用程序——例如一个视频编辑器，它可以打开视频文件，一个文档编辑器，它可以打开文档文件——让您的应用程序自动打开这些文件可能会很有用。在所有平台上，当您告诉操作系统使用特定的应用程序打开一个文件时，要打开的文件名会被作为命令行参数传递给该应用程序。

当您的应用程序运行时，传递给应用程序的参数始终可在 `sys.argv` 中找到。要自动打开文件，你可以在启动时检查 `sys.argv` 的值，如果其中包含文件名，则打开该文件。

以下应用程序运行时将打开一个窗口，显示所有接收到的命令行参数。

Listing 245. further/arguments.py

```
from PyQt6.QtWidgets import (
    QApplication,
    QWidget,
    QLabel,
    QVBoxLayout,
)

import sys


class Window(QWidget):
    def __init__(self):
        super().__init__()

        layout = QVBoxLayout()

        for arg in sys.argv: #1
            l = QLabel(arg)
            layout.addWidget(l)

        self.setLayout(layout)
        self.setWindowTitle("Arguments")


app = QApplication(sys.argv)
w = Window()
w.show()
```

```
app.exec()
```

> 1. `sys.argv` 是一个字符串列表。所有参数都是字符串。

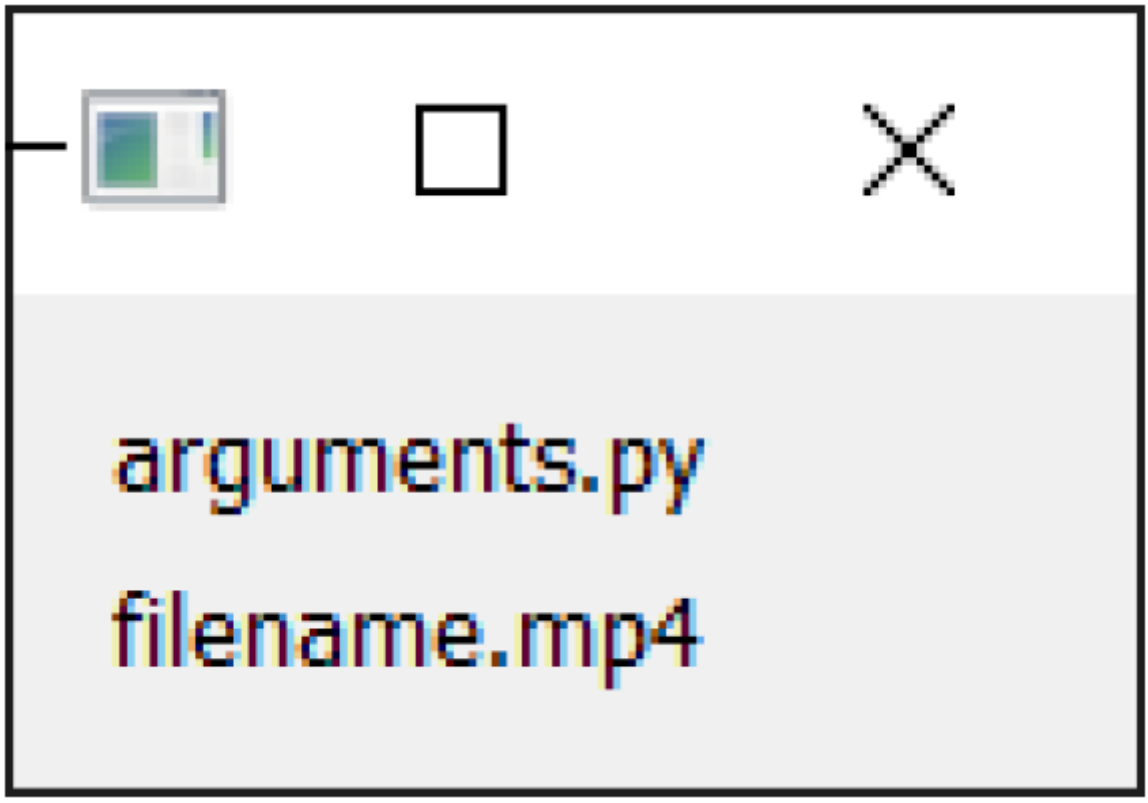
从命令行运行此应用程序，并传入一个文件名（您可以随意编造，我们不会加载它）。您可以传入任意数量的参数，多或少都行。

参数以字符串列表的形式传递给您的应用程序。所有参数都是字符串，即使是数值参数也是如此。您可以使用常规列表索引访问任何参数——例如，`sys.argv[1]` 将返回第二个参数。

您可以尝试运行上述脚本，并使用以下内容：

```
python arguments.py filename.mp4
```

这将生成下面的窗口。请注意，当使用 Python 运行时，第一个参数实际上是正在执行的 Python 文件。



> 图242：窗口打开，显示命令行参数。

如果您将应用程序打包用于分发，情况可能不再是这样——第一个参数现在可能是您正在打开的文件，因为没有作为参数传递的 Python 文件。这可能会导致问题，但一个简单的解决方法是将传递给应用程序的最后一个参数用作文件名，例如：

```
if len(sys.argv) > 0:
    filename_to_open = sys.argv[-1]
```

或者，您可以从列表中移除当前正在执行的脚本名称。当前正在执行的 Python 脚本名称始终可在 `__file__` 中获取。

```
if __file__ in sys.argv:
    sys.argv.remove(__file__)
```

> 它将始终出现在列表中，除非您已打包您的应用程序。

以下是一个进一步的示例，其中我们在命令行上接受一个文件名，然后打开该文本文件并在`QTextEdit`中显示。

Listing 246. further/arguments_open.py

```
from PyQt6.QtWidgets import QApplication, QMainWindow, QTextEdit

import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.editor = QTextEdit()

        if __file__ in sys.argv: #1
            sys.argv.remove(__file__)

        if sys.argv: #2
            filename = sys.argv[0] #3
            self.open_file(filename)

        self.setCentralWidget(self.editor)
        self.setWindowTitle("Text viewer")

    def open_file(self, fn):

        with open(fn, "r") as f:
            text = f.read()

        self.editor.setPlainText(text)


app = QApplication(sys.argv)
w = MainWindow()
w.show()

app.exec()
```

> 1. 如果脚本名称在`sys.argv`中，则将其移除。
> 2. 如果`sys.argv`中仍有内容（不为空）。
> 3. 将第一个参数作为要打开的文件名。

您可以按照以下方式运行此命令，以查看传入的文本文件。

```
python arguments_open.py notes.txt
```

打包与分发

> 设计只有在有人使用它时才算完成。
>
> ——布伦达·劳瑞尔（Brenda Laurel），博士

如果您无法与他人分享自己开发的应用程序，那么开发应用程序的乐趣就大打折扣了——无论是商业发布、在线分享，还是仅仅赠送给认识的人。分享您的应用程序，可以让其他人也能受益于您的辛勤工作！

将 Python 应用程序打包以供分发通常有点棘手，尤其是当目标平台是多个系统（Windows、macOS 和 Linux）时。这是因为需要将源代码、数据文件、Python 运行时以及所有相关库打包成一个能够在目标系统上可靠运行的包。幸运的是，有工具可以帮您解决这个问题！

在本章中，我们将逐步讲解如何打包应用程序以与他人分享。

37. 使用PyInstaller进行打包

PyInstaller 是一个跨平台的 PyQt6 打包系统，支持为 Windows、macOS 和 Linux 构建桌面应用程序。它会自动将您的 Python 应用程序以及任何相关的库和数据文件打包到一个独立的单文件可执行文件或可分发的文件夹中，您然后可以使用它来创建安装程序。

在本章中，我们将逐步演示如何使用 PyInstaller 打包一个 PyQt6 应用程序。我们即将构建的应用程序被我故意设计得非常简单，仅包含一个窗口和几个图标，但相同的流程可用于构建您自己的任何应用程序。我们将介绍如何以可重复的方式自定义应用程序名称、图标以及打包数据文件。我们还将探讨在构建自有应用程序时可能遇到的常见问题。

一旦我们将应用程序打包成可分发的可执行文件，我们将继续创建Windows安装程序、macOS磁盘映像和Linux包，这些都可以与他人分享。

> 本书的源代码下载包含适用于Windows、macOS和Ubuntu Linux的完整构建示例。

> 您始终需要在目标系统上编译您的应用程序。因此，如果您想要构建一个 Windows 可执行文件，您需要在 Windows 系统上进行此操作。

依赖

PyInstaller 默认支持 PyQt6，且截至本文撰写时，PyInstaller 的最新版本与 Python 3.6+ 兼容。无论您正在处理什么项目，都应能够打包您的应用程序。本教程假设您已安装了 Python 并配置了 `pip` 包管理器。

您可以使用 `pip` 安装 PyInstaller。

```
pip3 install PyInstaller
```

如果您在打包应用程序时遇到问题，您的第一步应该是使用以下命令将 PyInstaller 和 hooks 包更新到最新版本：

```
pip3 install --upgrade PyInstaller pyinstaller-hooks-contrib
```

挂钩模块包含针对常见 Python 包的特定打包说明和解决方法，并且比 PyInstaller 本身更新更频繁。

开始使用

从一开始就着手打包应用程序是个好主意，这样您就可以在开发过程中确认打包功能是否正常工作。这在添加额外依赖项时尤为重要。如果您只在最后才考虑打包问题，那么要准确定位问题所在可能会非常困难。

对于这个示例，我们将从一个简单的骨架应用程序开始，该应用程序目前不做任何有趣的事情。一旦我们完成了基本的打包过程，我们将开始扩展功能，并在每个步骤中确认构建过程仍然正常工作。

首先，为您的应用程序创建一个新文件夹，然后在名为 `app.py` 的文件中添加以下内容：

Listing 247. packaging/basic/app.py

```
from PyQt6.QtWidgets import QMainWindow, QApplication, QPushButton

import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")

        button = QPushButton("My simple app.")
        button.pressed.connect(self.close)

        self.setCentralWidget(button)
        self.show()

app = QApplication(sys.argv)
w = MainWindow()
app.exec()
```

这是一个基本的简易应用程序，它创建了一个自定义的 `QMainWindow` 并向其中添加了一个简单的 `QPushButton` 。点击该按钮将关闭窗口。您可以按照以下方式运行此应用程序：

```
python app.py
```

这应该会显示以下窗口

> 图243：适用于 Windows、macOS 和 Ubuntu Linux 的简单应用程序

构建基础应用程序

现在我们已经确认简单的应用程序可以正常运行，我们可以创建第一个测试构建。打开终端（shell）并导航到包含项目文件的文件夹。运行以下命令以创建 PyInstaller 构建。

```
pyinstaller --windowed app.py
```

> `--windowed` 命令行选项是构建 macOS 上的 `.app` 应用程序包以及在 Windows 上隐藏终端输出的必要条件。在 Linux 上该选项无效。

您将看到一系列输出消息，这些消息提供了关于PyInstaller正在执行什么操作的调试信息。这些信息对于排查构建过程中的问题非常有用，但如果不需要，也可以忽略。

Listing 248. Output running pyinstaller on Windows

```
> pyinstaller app.py
388 INFO: PyInstaller: 4.7
388 INFO: Python: 3.7.6
389 INFO: Platform: Windows-10-10.0.22000-SP0
392 INFO: wrote app.spec
394 INFO: UPX is not available.
405 INFO: Extending PYTHONPATH with paths
....etc.
```

构建完成后，查看您的文件夹，您会发现现在有了两个新的文件夹：`dist` 和 `build`。

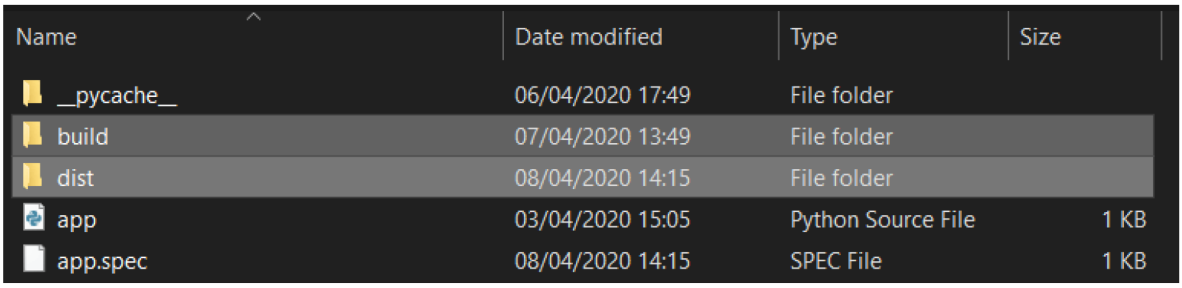


图244：PyInstaller 生成的 build 和 dist 文件夹

以下是文件夹结构的简要列表，展示了 `dist` 和 `build` 文件夹。实际文件会根据您构建的平台而有所不同，但整体结构始终保持一致。

```
.
├── app.py
├── app.spec
├── build
│   └── app
│       ├── localpycos
│       ├── Analysis-00.toc
│       ├── COLLECT-00.toc
│       ├── EXE-00.toc
│       ├── PKG-00.pkg
│       ├── PKG-00.toc
│       ├── PYZ-00.pyz
│       ├── PYZ-00.toc
│       ├── app
│       ├── app.pkg
│       ├── base_library.zip
│       ├── warn-app.txt
│       └── xref-app.html
└── dist
    └── app
        ├── lib-dynload
        ...
```

构建文件夹由 PyInstaller 用于收集和准备打包文件，其中包含分析结果和一些额外的日志。对于大多数情况，您可以忽略此文件夹的内容，除非您正在尝试调试问题。

`dist`（即“分发”）文件夹包含待分发的文件。这包括您的应用程序，以可执行文件形式打包，以及任何关联的库（例如PyQt6）。运行您的应用程序所需的一切都将包含在此文件夹中，这意味着您可以将此文件夹分发给他人以运行您的应用程序。

您可以现在尝试自行运行已构建的应用程序，只需运行 `dist` 文件夹中的可执行文件名为 `app` 的文件。稍等片刻后，您将看到应用程序的熟悉窗口弹出，如下图所示。

图245：简单应用程序，打包后即可运行。

在您的 Python 文件所在的同一文件夹中，与 `build` 和 `dist` 文件夹并列，PyInstaller 还会生成一个 `.spec` 文件。

`.spec` 文件

`.spec` 文件包含 PyInstaller 用于打包应用程序的构建配置和指令。每个 PyInstaller 项目都有一个 `.spec` 文件，该文件基于您在运行 `pyinstaller` 时传递的命令行选项生成。

当我们使用 `pyinstaller` 运行脚本时，除了传入我们的 Python 应用程序文件名外，没有传入其他任何内容。这意味着我们的 spec 文件目前仅包含默认配置。如果你打开它，您会看到类似于我们下面所示的内容。

Listing 249. packaging/basic/app.spec

```
# -*- mode: python ; coding: utf-8 -*-


block_cipher = None


a = Analysis(['app.py'],
             pathex=[],
             binaries=[],
             datas=[],
             hiddenimports=[],
             hookspath=[],
             hooksconfig={},
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)

exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='app',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=True,
          disable_windowed_traceback=False,
          target_arch=None,
          codesign_identity=None,
          entitlements_file=None )
coll = COLLECT(exe,a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
               upx=True,
```

```
               upx_exclude=[],
               name='app')
```

首先需要注意的是，这是一个 Python 文件，这意味着您可以编辑它并使用 Python 代码来计算设置的值。这在处理复杂构建时特别有用，例如当您针对不同平台进行构建时，希望根据条件定义要打包的额外库或依赖项。

如果您在 macOS 上进行构建，您还将有一个额外的 `BUNDLE` 块，用于构建 `.app` 包。该部分将大致如下所示：

```
app = BUNDLE(coll,
             name='app.app',
             icon=None,
             bundle_identifier=None)
```

如果您在其他平台上开始构建，但希望以后针对 macOS 进行构建，您可以手动将以下内容添加到 `.spec` 文件的末尾：

一旦生成了 `.spec` 文件，您可以将该文件传递给 `pyinstaller` ，而不是您的脚本，以重复之前的构建过程。现在运行此命令以重新构建您的可执行文件。

```
pyinstaller app.spec
```

生成的构建结果将与用于生成 `.spec` 文件的构建完全相同（假设您未对项目进行任何修改）。对于许多PyInstaller配置更改，您可以选择通过命令行参数传递参数，或修改现有的 `.spec` 文件。具体选择哪种方式由您决定，不过我建议对于更复杂的构建，最好直接编辑 `.spec` 文件。

调整构建过程

我们已经创建了一个非常简单的应用程序并构建了第一个可执行文件。现在我们将看看可以对构建过程进行的一些调整

为您的应用程序命名

您可以做的最简单的更改之一就是为您的应用程序提供一个合适的“名称”。默认情况下，应用程序会采用源文件的名称（不包括扩展名），例如 `main` 或 `app` 。这通常不是您希望为可执行文件命名的名称。

您可以通过编辑 `.spec` 文件并修改 `EXE` 和 `COLLECT` 块（以及 macOS 上的 `BUNDLE` ）下的 `name=` 属性，为 PyInstaller 指定一个更友好的名称用于您的可执行文件（以及 `dist` 文件夹）。

Listing 250. packaging/custom/hello-world.spec

```
exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='hello-world',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=True,
          disable_windowed_traceback=False,
          target_arch=None,
```

```
          codesign_identity=None,
          entitlements_file=None )
coll = COLLECT(exe,
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
               upx=True,
               upx_exclude=[],
               name='hello-world')
```

`EXE` 下的名称是可执行文件的名称，而 `COLLECT` 下的名称是输出文件夹的名称。

> 我建议您为可执行文件使用不包含空格的名称——请改用连字符或驼峰式命名法。

`BUNDLE` 块中指定的名称用于 macOS 应用程序包，该名称是应用程序在 Launchpad 和 Dock 中显示的用户可见名称。在我们的示例中，我们将应用程序可执行文件命名为 “hello-world”，但对于 `.app` 包，您可以使用更友好的 “Hello World.app”。

Listing 251. packaging/custom/hello-world.spec

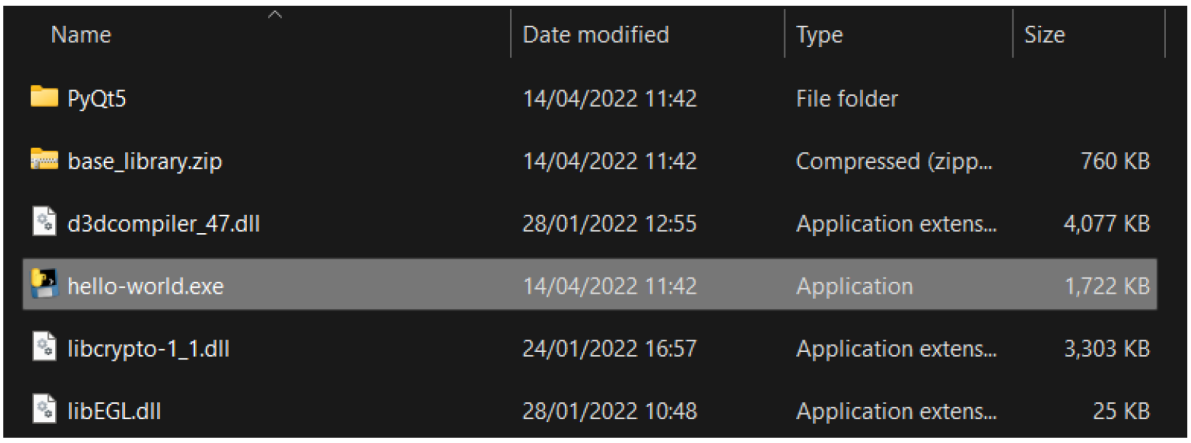
```
app = BUNDLE(coll,
             name='Hello World.app',
             icon=None,
             bundle_identifier=None)
```

或者，您可以重新运行 `pyinstaller` 命令，并传递 `-n` 或 `--name` 配置选项，同时提供您的 `app.py` 脚本。

```
pyinstaller --windowed -n "hello-world" app.py
# 或者
pyinstaller --windowed --name "hello-world" app.py
```

生成的可执行文件将命名为 `hello-world`，而展开后的构建文件将放置在 `dist\hello-world\` 文件夹中。`.spec` 文件的名称取自命令行中传递的名称，因此这也将为您创建一个新的 `spec` 文件，名为 `hello-world.spec`，位于您的根文件夹中。

> 如果您创建了一个新的 `.spec` 文件，请删除旧的文件以避免混淆！



> 图246：带有自定义名称“hello-world”的应用程序。

应用程序图标

另一个简单的改进是更改应用程序运行时显示的应用程序图标。我们可以设置应用程序窗口/任务栏的图标，这可以通过在代码中调用 `.setWindowIcon()` 实现。

Listing 252. packaging/custom/app.py

```
from PyQt6.QtWidgets import QMainWindow, QApplication, QPushButton
from PyQt6.QtGui import QIcon

import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")

        button = QPushButton("My simple app.")
        button.pressed.connect(self.close)

        self.setCentralWidget(button)
        self.show()

app = QApplication(sys.argv)
app.setWindowIcon(QIcon("icon.svg"))
w = MainWindow()
app.exec()
```

在此，我们向应用程序实例添加了 `.setWindowIcon` 调用。这定义了一个用于应用程序所有窗口的默认图标。如果您愿意，可以针对每个窗口单独覆盖此设置，通过在窗口本身调用 `.setWindowIcon` 来实现。将图标复制到与脚本相同的文件夹中。

如果您运行上述应用程序，现在应该会在Windows系统的窗口上看到图标，而在macOS或Ubuntu Linux系统中则会在Dock栏上看到图标。

图247：显示自定义图标的窗口

关于图标的说明。

在此示例中，我们设置了一个单一的图标文件，使用可缩放矢量图形（SVG）文件，该文件在任何大小下都将保持清晰。您也可以使用位图图像，在这种情况下，您需要提供多种尺寸以确保图标始终保持清晰。在 Windows 上，您可以通过创建 ICO 文件来实现，该文件是包含多个图标的特殊文件。在 Linux 上，您可以在安装过程中提供多个不同的 PNG 文件（参见 Linux 打包部分）。在 macOS 上，多个图标尺寸由包含在 `.app` 包中的 ICNS 文件提供。

是的，这确实让人困惑！但幸运的是，Qt支持所有平台上的各种图标格式。

即使您没有看到图标，请继续阅读！

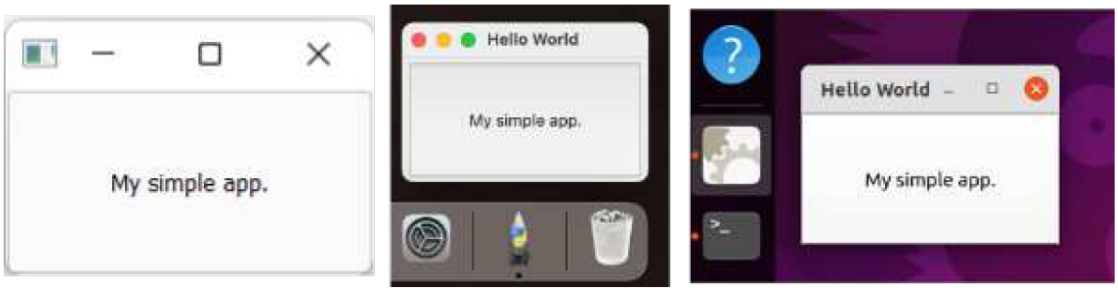
处理相对路径

这里有一个需要注意的细节，可能并不明显。请您打开终端，切换到保存脚本的文件夹。然后像往常一样运行它：

```
python3 app.py
```

如果图标位于正确的位置，您应该能够看到它们。现在切换到父文件夹，并再次运行您的脚本（将 `<folder>` 替换为脚本所在文件夹的名称）。

```
cd ..
python3 <folder>/app.py
```



> 图248：缺少图标的窗口

图标没有显示。这是怎么回事？

我们使用相对路径来引用数据文件。这些路径相对于当前工作目录——不是脚本所在的文件夹，而是您运行脚本的文件夹。如果您从其他地方运行脚本，它将无法找到这些文件。

> 图标无法显示的一个常见原因是，在使用项目根目录作为当前工作目录的IDE中运行示例。

这是在应用程序打包前的一个小问题，但一旦安装后，您将无法知道应用程序运行时当前的工作目录是什么——如果工作目录错误，您的应用程序将无法找到其数据文件。我们需要在继续之前解决这个问题，可以通过将路径设置为相对于应用程序文件夹的相对路径来实现。

在下面的更新代码中，我们定义了一个新变量 `basedir`，使用 `os.path.dirname` 获取 `__file__` 所包含的文件夹，该文件夹包含当前 Python 文件的完整路径。然后，我们使用 `os.path.join()` 构建数据文件的相对路径。

> 请参阅“使用相对路径”部分以获取更多信息，以及在应用程序中使用相对路径的更可靠方法。

由于我们的 `app.py` 文件位于文件夹的根目录下，因此所有其他路径都是相对于该目录的。

Listing 253. packaging/custom/app_relative_paths.py

```
import os
import sys

from PyQt6.QtGui import QIcon
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

basedir = os.path.dirname(__file__)


class MainWindow(QMainWindow):
```

```
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")

        button = QPushButton("My simple app.")
        button.setIcon(QIcon(os.path.join(basedir, "icon.svg")))
        button.pressed.connect(self.close)

        self.setCentralWidget(button)
        self.show()

app = QApplication(sys.argv)
app.setWindowIcon(QIcon(os.path.join(basedir, "icon.svg")))
w = MainWindow()
app.exec()
```

请尝试从父文件夹重新运行您的应用程序——您会发现图标现在会如预期般显示，无论您从何处启动应用程序。

任务栏图标（仅限Windows系统）

在 Windows 系统中，使用 `.setWindowIcon()` 方法可以正确设置窗口的图标。然而，由于 Windows 系统对窗口的跟踪和分组方式，有时图标可能不会在任务栏上显示。

> 如果对您有效，太好了！但当您分发应用程序时，它可能无法正常工作，因此请务必按照以下步骤操作！

当您运行应用程序时，Windows 会检查可执行文件并尝试确定其所属的“应用程序组”。默认情况下，所有 Python 脚本（包括您的应用程序）都会被归类到同一个“Python”组中，因此会显示 Python 图标。要阻止这种情况发生，我们需要为应用程序提供一个不同的应用程序标识符。

下面的代码通过调用 `SetCurrentProcessExplicitAppUserModelID()` 方法，并传入自定义的应用程序 ID 来实现这一点。

Listing 254. packaging/custom/app_windows_taskbar.py

```
from PyQt6.QtWidgets import QMainWindow, QApplication, QPushButton
from PyQt6.QtGui import QIcon

import sys, os

basedir = os.path.dirname(__file__)

try: #1
    from ctypes import windll # 仅存在于Windows系统中.

    myappid = "mycompany.myproduct.subproduct.version" #2
```

```
    windll.shell32.SetCurrentProcessExplicitAppUserModelID(myappid)
except ImportError:
    pass


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")

        button = QPushButton("My simple app.")
        button.setIcon(QIcon(os.path.join(basedir, "icon.svg")))
        button.pressed.connect(self.close)

        self.setCentralWidget(button)
        self.show()


app = QApplication(sys.argv)
app.setWindowIcon(QIcon(os.path.join(basedir, "icon.svg")))
w = MainWindow()
app.exec()
```

> 1. 该代码被包裹在 `try/except` 块中，因为 `windll` 模块在非Windows平台上不可用。这使得您的应用程序可以在macOS和Linux上继续运行。
> 2. 为您的应用程序自定义应用程序标识符字符串。

上述列表显示了一个通用的 `mycompany.myproduct.subproduct.version` 字符串，但您应将其修改为与实际应用程序相符。此处填写的内容并不重要，但惯例是使用反向域名表示法，即使用 `com.mycompany` 作为公司标识符。

将此代码添加到您的脚本中，您的图标一定会显示在任务栏上。

图249：自定义图标显示在任务栏上

可执行文件图标（仅限Windows系统）

现在，当应用程序运行时，图标显示正确。但是，您可能会注意到，您的应用程序可执行文件仍然使用不同的图标。在 Windows 系统中，应用程序可执行文件可以嵌入图标，以便更容易识别。默认图标是由 PyInstaller 提供的，但您可以用自己的图标替换它。

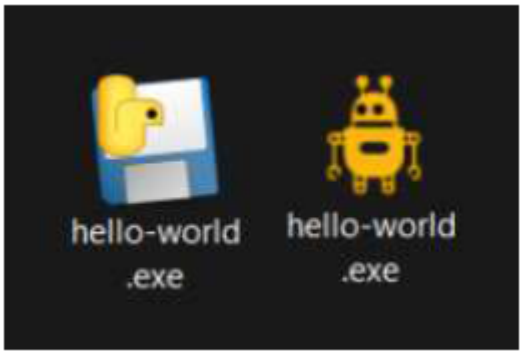
要为 Windows 可执行文件添加图标，您需要向 `EXE` 块提供一个 `.ico` 格式的文件。

Listing 255. packaging/custom/hello-world-icons.spec

```
exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='hello-world',
          icon='icon.ico',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=True,
          disable_windowed_traceback=False,
          target_arch=None,
          codesign_identity=None,
          entitlements_file=None )
```

要创建 .ico 文件，我建议您使用 Greenfish Icon Editor Pro，这是一个免费且开源的工具，也可以为 Windows 创建图标。本书的下载内容中包含一个示例 `.ico` 文件。

如果您使用修改后的 `.spec` 文件运行 `pyinstaller` 构建，您会发现可执行文件现在有了自定义图标。



> 图250：Windows 可执行文件显示默认和自定义图标

> 您还可以通过在初始构建时向 pyinstaller 传递 `--icon icon.ico` 参数来提供图标。您可以通过这种方式提供多个图标，以支持 macOS 和 Windows。

macOS .app 应用程序包图标（仅限 macOS）

在 macOS 上，应用程序以 `.app` 包的形式分发，这些包可以拥有自己的图标。包图标用于在启动台和应用启动时在 Dock 上识别应用程序。PyInstaller 可以为您添加图标到应用程序包中，您只需将 ICNS 格式文件传递到 `.spec` 文件中的 `BUNDLE` 块即可。该图标随后将显示在生成的包中，并在应用启动时显示。

Listing 256. packaging/custom/hello-world-icons.spec

```
app = BUNDLE(coll,
             name='Hello World.app',
             icon='icon.icns',
             bundle_identifier=None)
```

ICNS 是 macOS 系统中图标文件的文件格式。您可以在 macOS 系统上使用 Icon Composer 创建图标文件。您还可以在 Windows 系统上使用 Greenfish Icon Editor Pro 创建 macOS 图标。

> 图251：macOS .app 应用程序包显示默认和自定义图标

> 您还可以通过在初始构建时向 `pyinstaller` 传递 `--icon icon.icns` 参数来提供图标。您可以通过这种方式提供多个图标，以支持 macOS 和 Windows。

在我们的示例中，应用程序启动时，包中的图标集将被 `.setWindowIcon` 调用替换。然而，在 macOS 上，您可以完全跳过 `.setWindowIcon()` 调用，只需通过 `.app` 包设置图标即可。

数据文件和资源

现在我们已经有一个可以正常运行的应用程序，它拥有自定义名称、自定义应用程序图标，以及一些调整，以确保该图标在所有平台上显示，并且无论应用程序从何处启动，都能正确显示。在这些设置就位后，最后一步是确保该图标正确打包到应用程序中，并且在从 `dist` 文件夹运行时继续显示。

> 试试看，它不会如您所愿。

问题是，我们的应用程序现在依赖于一个外部数据文件（图标文件），而该文件不属于我们的源代码。为了使应用程序正常运行，我们现在需要将该数据文件与应用程序一起分发。PyInstaller 可以帮助我们实现这一点，但我们需要告诉它要包含哪些内容，以及在输出中将它们放置在哪里。

在下一节中，我们将探讨管理与应用程序相关的数据文件的可用选项。这种方法不仅适用于图标文件，还可用于任何其他数据文件，包括您的应用程序所需的Qt Designer `.ui` 文件。

使用PyInstaller打包数据文件

我们的应用程序现在依赖于一个图标文件。

Listing 257. packaging/data-file/app.py

```
from PyQt6.QtWidgets import (
    QMainWindow,
    QApplication,
    QPushButton,
    QVBoxLayout,
    QLabel,
    QWidget,
)
from PyQt6.QtGui import QIcon

import sys, os

basedir = os.path.dirname(__file__)
```

```
try:
    from ctypes import windll # 仅存在于Windows系统中.

    myappid = "mycompany.myproduct.subproduct.version"
    windll.shell32.SetCurrentProcessExplicitAppUserModelID(myappid)
except ImportError:
    pass


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")
        layout = QVBoxLayout()
        label = QLabel("My simple app.")
        label.setMargin(10)
        layout.addWidget(label)

        button = QPushButton("Push")
        button.pressed.connect(self.close)
        layout.addWidget(button)

        container = QWidget()
        container.setLayout(layout)

        self.setCentralWidget(container)

        self.show()


app = QApplication(sys.argv)
app.setWindowIcon(QIcon(os.path.join(basedir, "icon.svg")))
w = MainWindow()
app.exec()
```

将此数据文件放入 `dist` 文件夹的最简单方法是直接告诉PyInstaller将其复制过去。PyInstaller支持指定要复制的单个文件路径列表，以及相对于 `dist/<应用程序名称>` 文件夹的文件夹路径，用于将文件复制到该位置。

与其他选项一样，这可以通过命令行参数指定，`--add-data`，您可以多次提供该参数。

```
pyinstaller --add-data "icon.svg:." --name "hello-world" app.py
```

> 路径分隔符因平台而异，在 Linux 或 Mac 上使用 `:` 而在 Windows 上使用 `;`

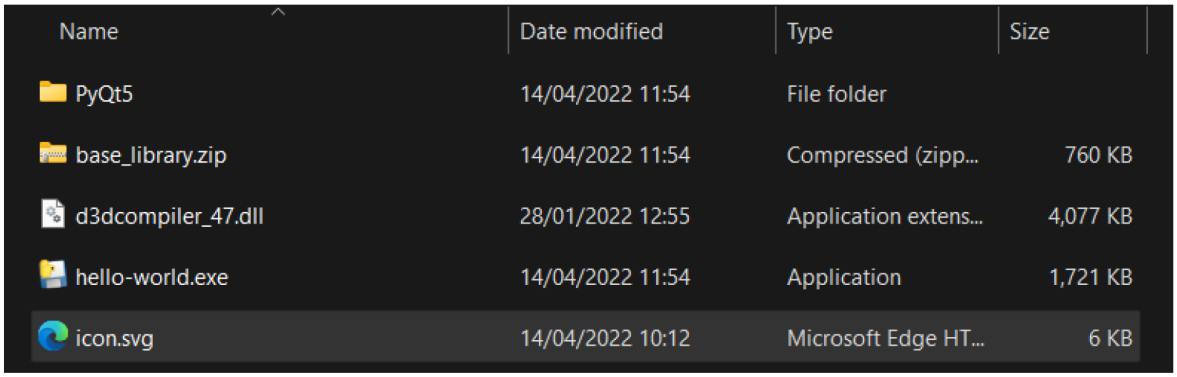
或者通过规格文件分析部分中的数据列表，以源位置和目标位置的元组形式。

```
a = Analysis(['app.py'],
             pathex=[],
             binaries=[],
             datas=[('icon.svg', '.')],
             hiddenimports=[],
             hookspath=[],
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
```

然后使用以下命令执行 `.spec` 文件：

```
pyinstaller hello-world.spec
```

在两种情况下，我们都告诉 PyInstaller 将指定的文件 `icon.svg` 复制到 `.` 位置。这意味着输出文件夹 `dist`。如果需要，我们也可以在此指定其他位置。 如果您现在运行构建，您应该可以在输出文件夹 `dist` 中看到您的 `.svg` 文件，准备与您的应用程序一起分发。



> 图252：被复制到 dist 文件夹的图标文件

如果您从 `dist` 目录运行应用程序，现在应该能看到预期的图标。

> 图253：窗口（Windows）和底栏（macOS 和 Ubuntu）上显示的图标

> 该文件必须使用相对路径在Qt中加载，并且其相对位置与EXE文件的相对位置相同，与它在 `.py` 文件中的相对位置相同，以确保其正常工作。

> 如果您在 Windows 机器上开始构建，您的 `.spec` 文件可能会包含使用双反斜杠 `\\` 的路径。这在其他平台上无法正常工作，因此您应该将这些路径替换为单正斜杠 `/`，因为单正斜杠在所有平台上都有效。

打包数据文件夹

通常，您会有多个数据文件需要包含在您的打包文件中。最新版本的PyInstaller允许您像打包文件一样打包文件夹，同时保留子文件夹结构。为了演示如何打包数据文件夹，让我们在应用程序中添加几个按钮并为它们添加图标。我们可以将这些图标放在一个名为 `icons` 的文件夹下。

Listing 258. packaging/data-folder/app.py

```
from PyQt6.QtWidgets import (
    QMainWindow,
    QApplication,
    QLabel,
    QVBoxLayout,
    QPushButton,
    QWidget,
)
from PyQt6.QtGui import QIcon
import sys, os

basedir = os.path.dirname(__file__)

try:
    from ctypes import windll # 仅存在于Windows系统中.
    myappid = "mycompany.myproduct.subproduct.version"
    windll.shell32.SetCurrentProcessExplicitAppUserModelID(myappid)
except ImportError:
    pass


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")
        layout = QVBoxLayout()
        label = QLabel("My simple app.")
        label.setMargin(10)
        layout.addWidget(label)

        button_close = QPushButton("Close")
        button_close.setIcon(
```

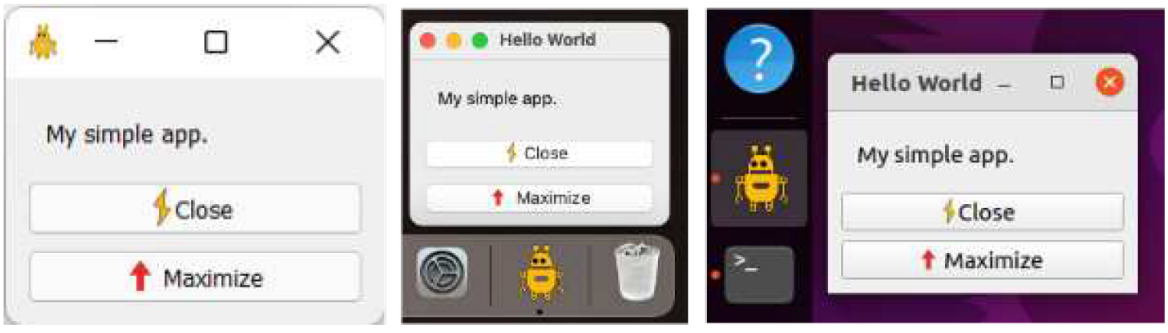
```
            QIcon(os.path.join(basedir, "icons", "lightning.svg"))
        )
        button_close.pressed.connect(self.close)
        layout.addWidget(button_close)

        button_maximize = QPushButton("Maximize")
        button_maximize.setIcon(
            QIcon(os.path.join(basedir, "icons", "uparrow.svg"))
        )
        button_maximize.pressed.connect(self.showMaximized)
        layout.addWidget(button_maximize)

        container = QWidget()
        container.setLayout(layout)

        self.setCentralWidget(container)

        self.show()


app = QApplication(sys.argv)
app.setWindowIcon(QIcon(os.path.join(basedir, "icons", "icon.svg")))
w = MainWindow()
app.exec()
```

> 此代码中包含了Windows任务栏图标的修复程序，如果您不是在为Windows构建应用程序，可以跳过它。

图标（均为SVG文件）存储在名为“icons”的子文件夹中。

```
.
├── app.py
└── icons
    └── lightning.svg
    └── uparrow.svg
    └── icon.svg
```

如果您运行此程序，将看到以下窗口，其中按钮上带有图标，窗口或 Dock 栏中也有一个图标。



> 图254：带有多个图标的窗口

要将 `icons` 文件夹复制到我们的构建应用程序中，我们只需将该文件夹添加到我们的 `.spec` 文件的 `Analysis` 块中。对于单个文件，我们将其作为元组添加，其中包含源路径（来自我们的项目文件夹）和目标文件夹，该目标文件夹位于生成的 `dist` 文件夹下。

Listing 259. packaging/data-folder/hello-world.spec

```
# -*- mode: python ; coding: utf-8 -*-


block_cipher = None


a = Analysis(['app.py'],
             pathex=[],
             binaries=[],
             datas=[('icons', 'icons')],
             hiddenimports=[],
             hookspath=[],
             hooksconfig={},
             runtime_hooks=[],
             excludes=[],
             win_no_prefer_redirects=False,
             win_private_assemblies=False,
             cipher=block_cipher,
             noarchive=False)
pyz = PYZ(a.pure, a.zipped_data,
          cipher=block_cipher)

exe = EXE(pyz,
          a.scripts,
          [],
          exclude_binaries=True,
          name='hello-world',
          icon='icons/icon.ico',
          debug=False,
          bootloader_ignore_signals=False,
          strip=False,
          upx=True,
          console=False,
          disable_windowed_traceback=False,
          target_arch=None,
          codesign_identity=None,
          entitlements_file=None )
coll = COLLECT(exe,
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=False,
               upx=True,
               upx_exclude=[],
               name='hello-world')
app = BUNDLE(coll,
             name='Hello World.app',
```

```
            icon='icons/icon.icns',
            bundle_identifier=None)
```

如果您使用这个规格文件运行构建，你会发现图标文件夹已被复制到 `dist` 文件夹中。如果您从该文件夹运行应用程序——或从任何其他位置运行——图标将如预期显示，因为相对路径在新的位置仍然正确。

总结

随着这些更改的实施，您现在可以跨所有平台可重复地构建您的应用程序。在接下来的章节中，我们将继续探讨如何将已构建的可执行文件打包成可用的安装程序。

到目前为止，我们已经逐步介绍了如何在您自己的平台上使用PyInstaller构建应用程序。通常，您可能希望为所有平台构建您的应用程序。

如前所述，您只能在特定平台上为该平台构建软件——即，如果您想构建一个 Windows 可执行文件，您需要在 Windows 系统上进行构建。然而，理想情况下，您希望能够使用同一个 `.spec` 文件来完成这一操作，以简化维护工作。如果您想支持多个平台，现在也可以在其他系统上测试您的 `.spec` 文件，以确保构建配置正确。如果出现问题，请查阅本章中各平台的特定说明。

38. 使用InstallForge创建Windows安装程序

到目前为止，我们一直使用PyInstaller来打包应用程序以供分发。打包过程的输出结果是一个名为 `dist` 的文件夹，其中包含应用程序运行所需的所有文件。虽然您可以将此文件夹作为ZIP文件分享给用户，但这并非最佳的用户体验。

Windows 桌面应用程序通常随安装程序一起分发，安装程序负责将可执行文件（以及任何其他文件）放置到正确的位置，并添加“开始”菜单快捷方式。接下来，我们将探讨如何使用我们的 `dist` 文件夹来创建一个可运行的Windows安装程序。

为了创建我们的安装程序，我们将使用一个名为 InstallForge 的工具。InstallForge是免费的，可以从 本页 下载。本书下载中的InstallForge工作配置可作为 `Hello World.ifp` 文件获取，但请注意源路径需要根据您的系统进行更新。

> 如果您急于尝试，可以先下载 示例Windows安装程序。

我们将逐步演示使用InstallForge创建安装程序的基本步骤。

通用设置

当您首次运行InstallForge时，将看到此“通用”(General)选项卡。在此，您可以输入应用程序的基本信息，包括名称、程序版本、公司和网站。

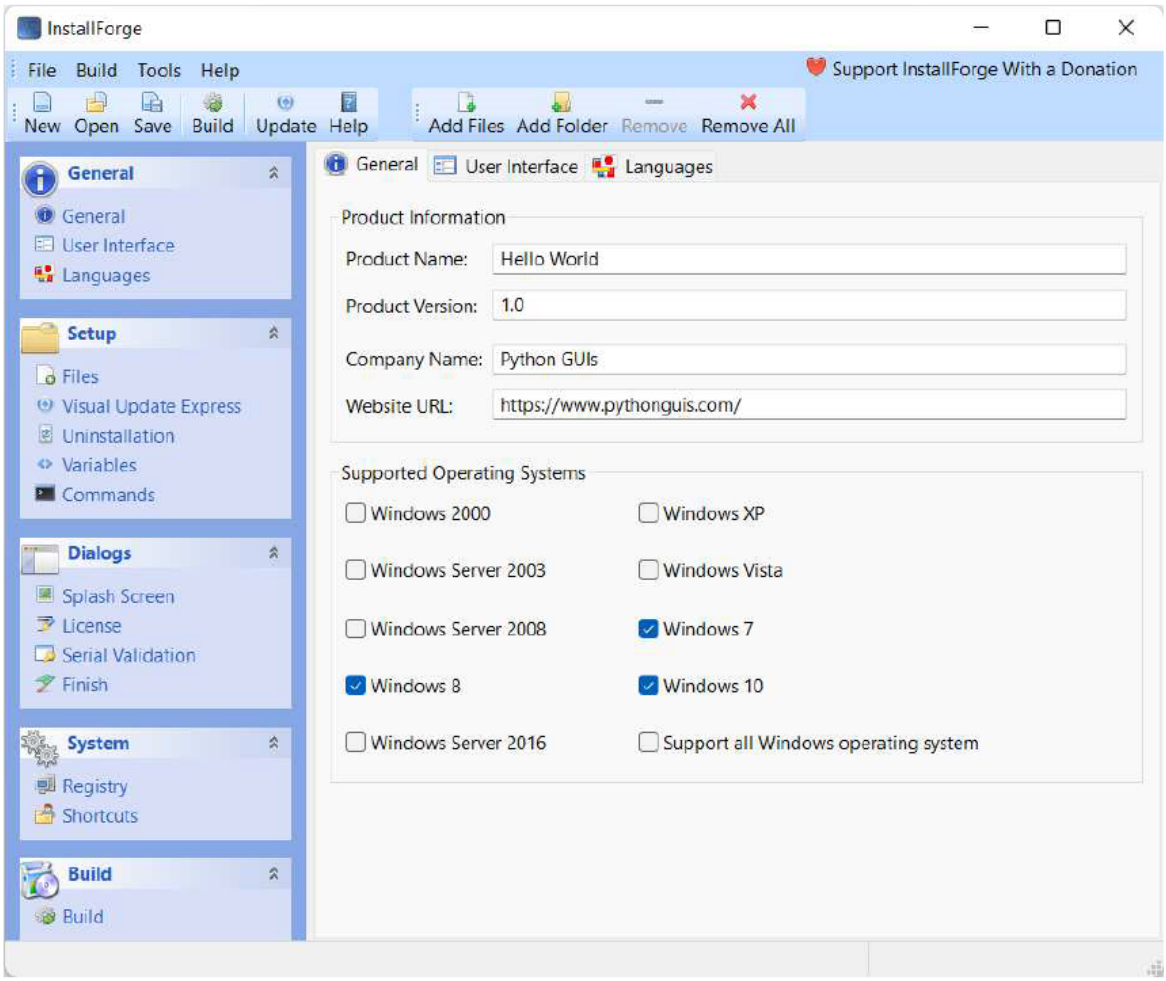


图255：InstallForge 初始视图，显示通用设置

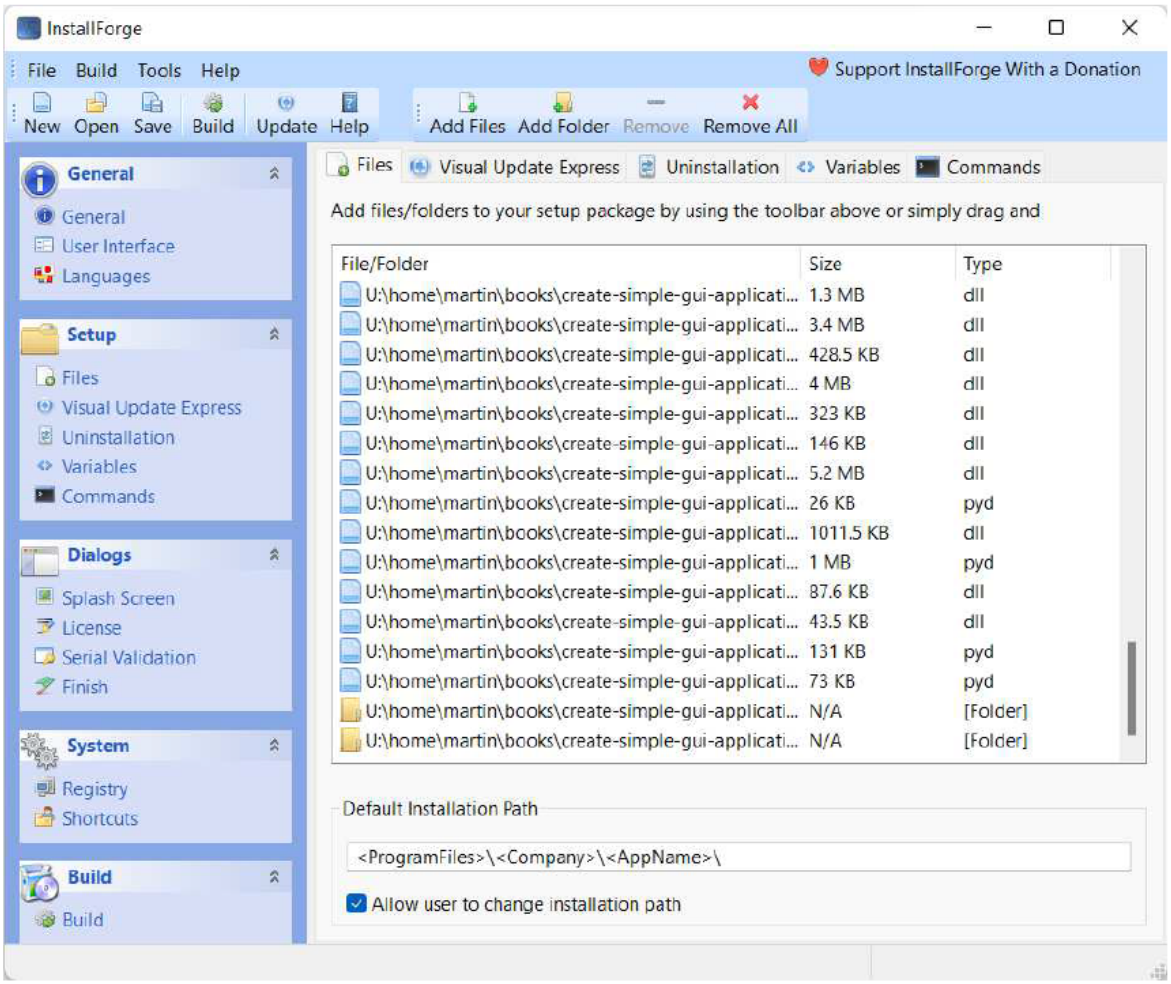
您还可以选择安装程序的目标平台，从目前可用的各种版本的Windows 中进行选择。这确保用户只能将您的应用程序安装在与之兼容的 Windows 版本上。

> 这里没有什么神奇之处，在安装程序中选择额外的平台并不会让您的应用程序在这些平台上运行！您需要在安装程序中启用这些平台之前，先确保您的应用程序可以在目标版本的Windows上运行。

选定安装文件

点击左侧边栏以打开“Setup”下的“Files”页面。在此处您可以指定要打包到安装程序中的文件。

在工具栏上点击“Add Files...”并选择PyInstaller生成的 `dist/hello-world` 目录下的所有文件。弹出的文件浏览器支持多文件选择，因此您可以一次性添加所有文件，但需要单独添加文件夹。您可以点击“添加文件夹...”并添加 `dist/hello-world` 目录下的任何文件夹，例如您的 `icons` 文件夹和其他库文件夹。

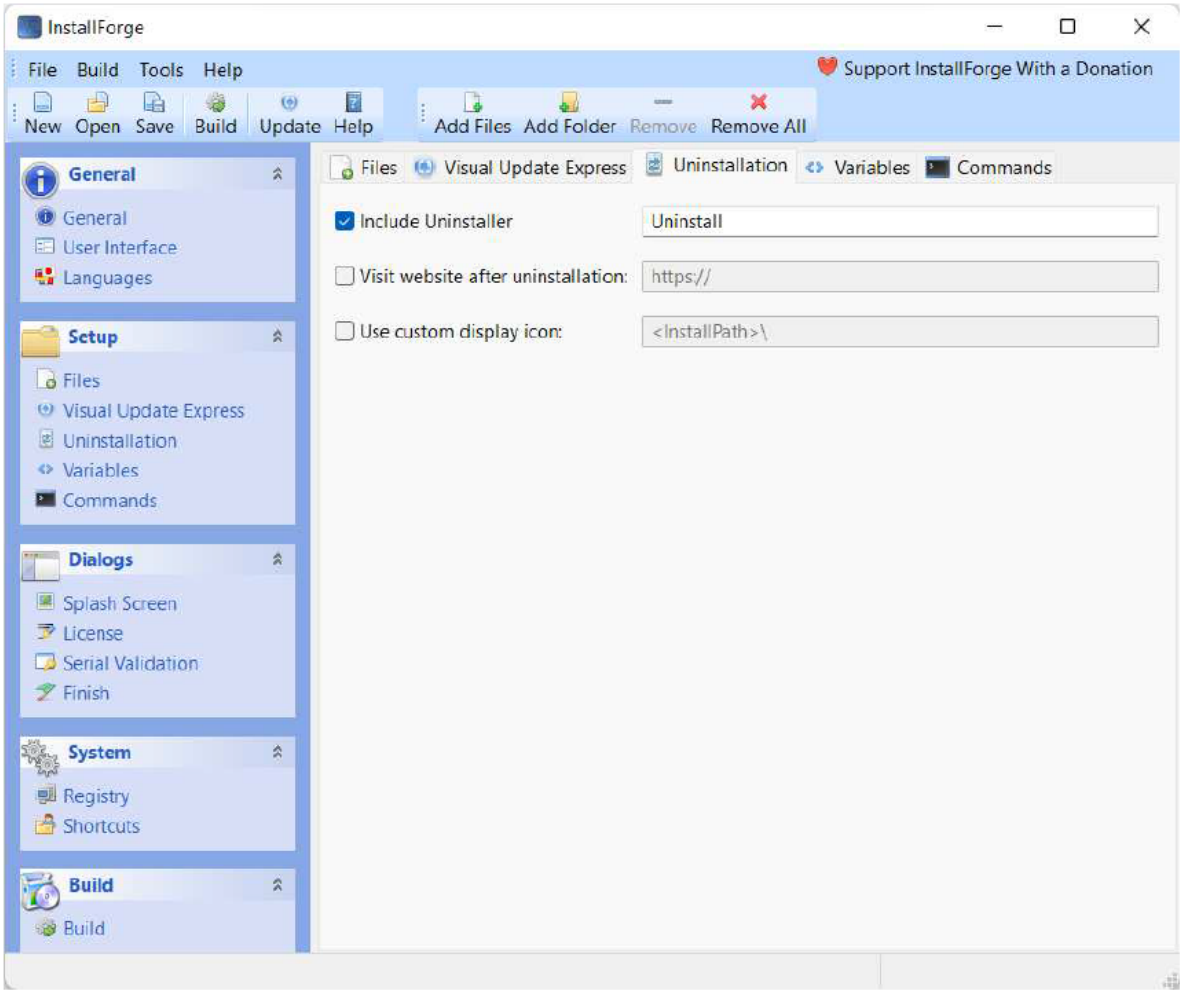


> 图256：在InstallForge文件视图中，将所有要打包的文件和文件夹添加进去。

> 所选文件夹的内容将被递归包含，您无需选择子文件夹。

完成后，您可以滚动列表到底部，确保以下文件夹被列出以包含在内：`dist/helloworld` 下的所有文件和文件夹都应存在。但 `dist/hello-world` 文件夹本身不应被列出。

默认安装路径可以保持不变。尖括号中的值，例如 `<company>` 是变量，将从配置中自动填充。接下来，建议允许用户卸载您的应用程序。尽管它无疑很棒，但用户未来可能希望卸载它。您可以在“卸载”选项卡中通过勾选复选框来实现此功能。这也将使应用程序出现在Windows“添加或删除程序”列表中。



> 图257：InstallForge 为您的应用程序添加卸载程序

对话框

“对话框”部分可用于向用户显示自定义消息、启动画面或许可证信息。 “完成”选项卡可用于控制安装程序完成后发生的事情，在此处为用户提供在安装完成后运行程序的选项会很有帮助。

要实现这一点，您需要勾选“Run program”旁边的复选框，并将您自己的应用程序 EXE 文件添加到该框中。由于 `<installpath>\` 已经指定，我们只需添加 `hello-world.exe` 即可。在首次启动时，可以使用参数向程序传递任何参数。

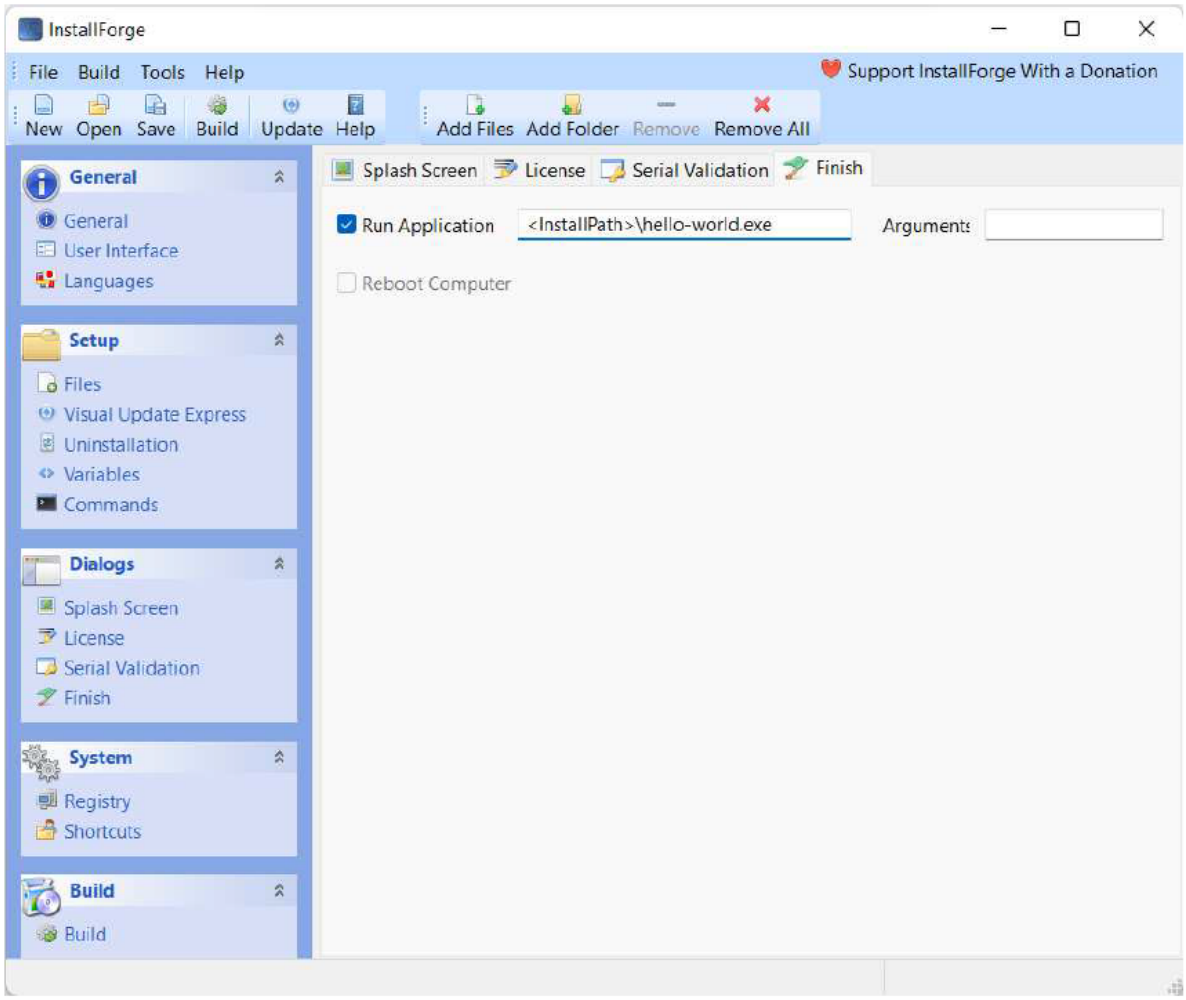
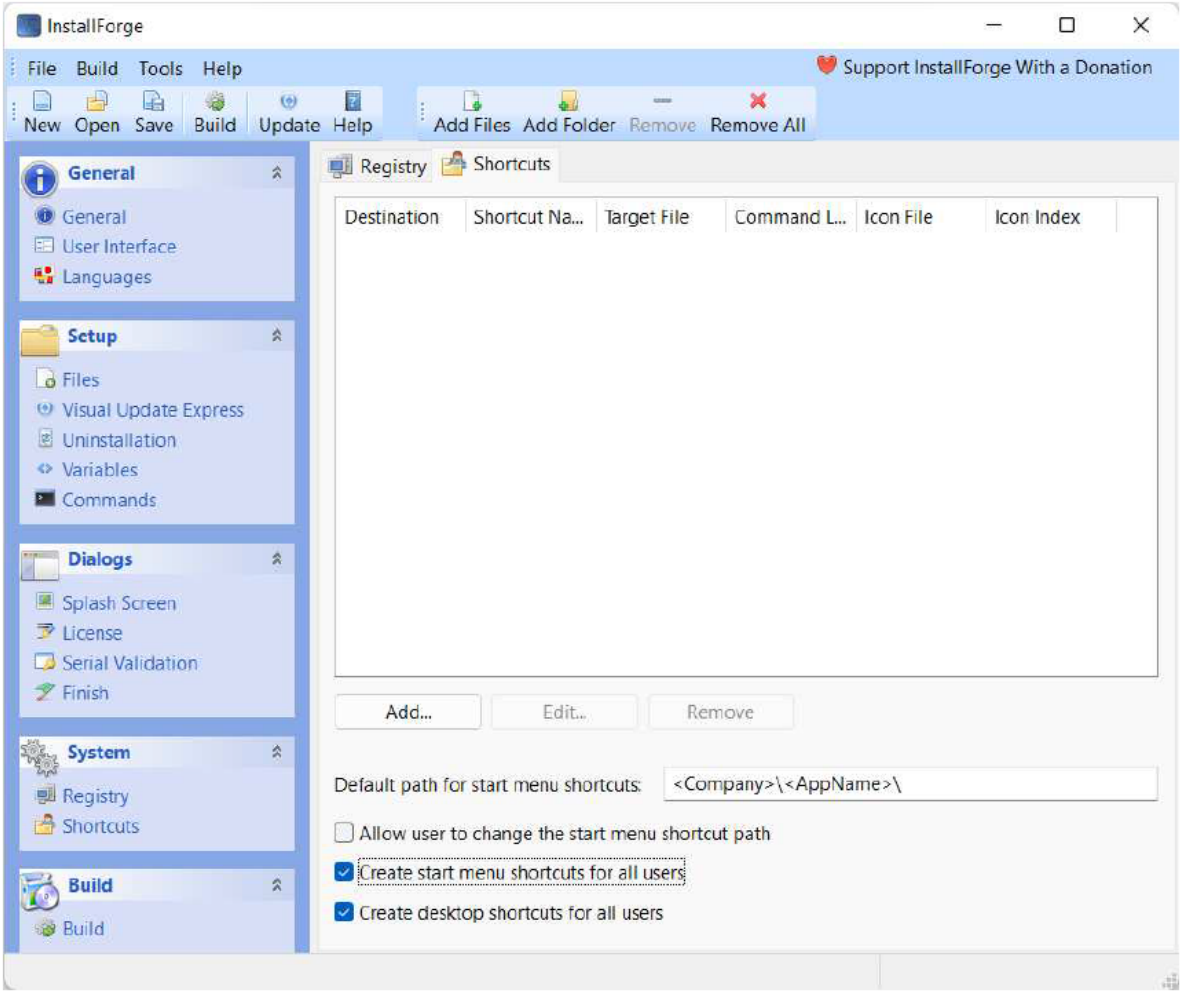


图258：InstallForge 在安装完成后配置可选的运行程序。

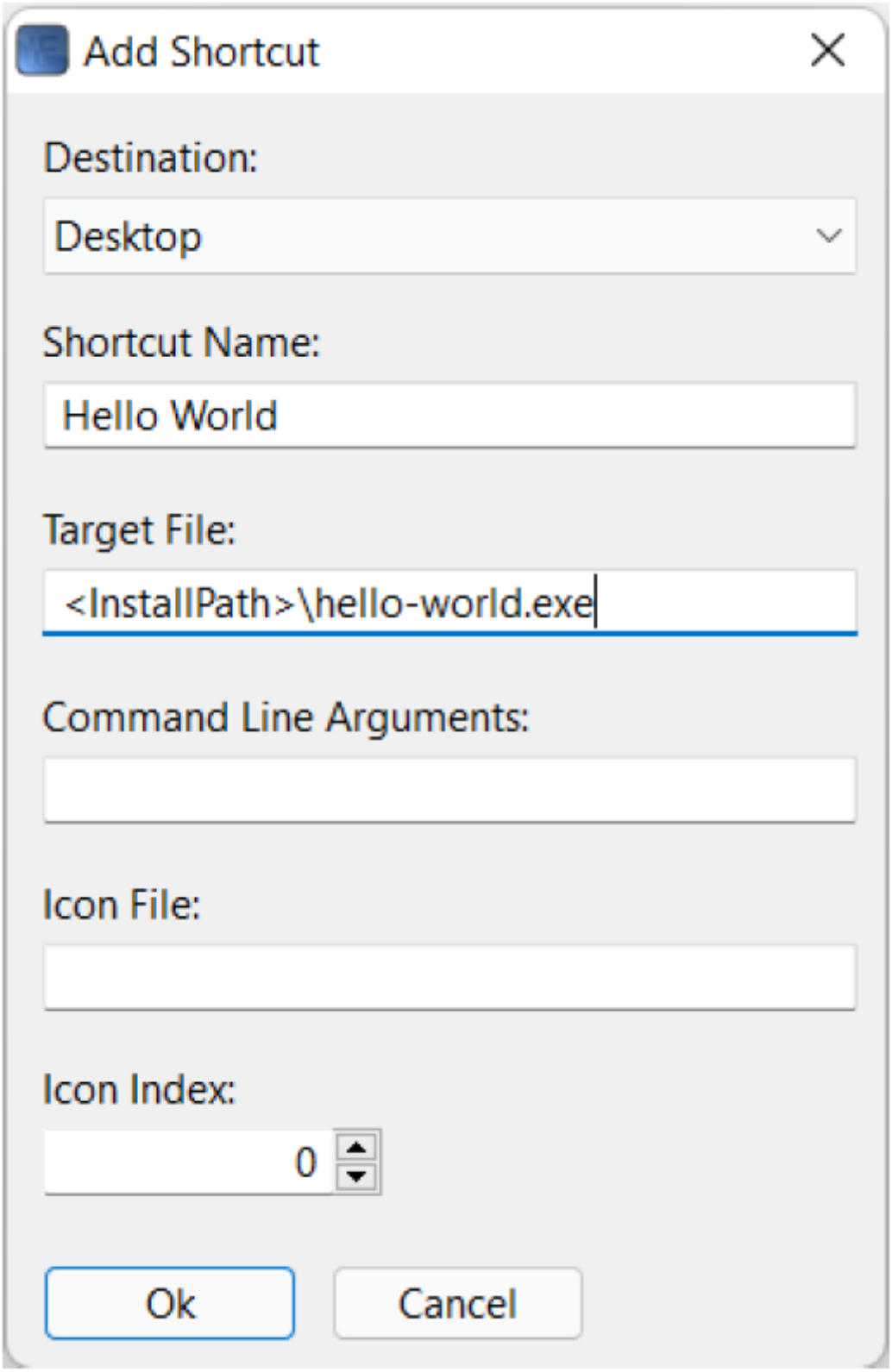
系统

在“System”下选择“System”可以打开快捷方式编辑器。在这里，您可以为开始菜单和桌面设置快捷方式。



> 图259：InstallForge 配置快捷方式，用于开始菜单和桌面

点击“Add...”可以添加应用程序的新快捷方式。选择“开始菜单” (Start menu) 或“桌面” (Desktop) 快捷方式，并填写名称和目标文件。这是应用程序安装后EXE文件的最终路径。由于 `<installpath>\` 已指定，您只需在末尾添加应用程序的EXE文件名，例如 `hello-world.exe` 。



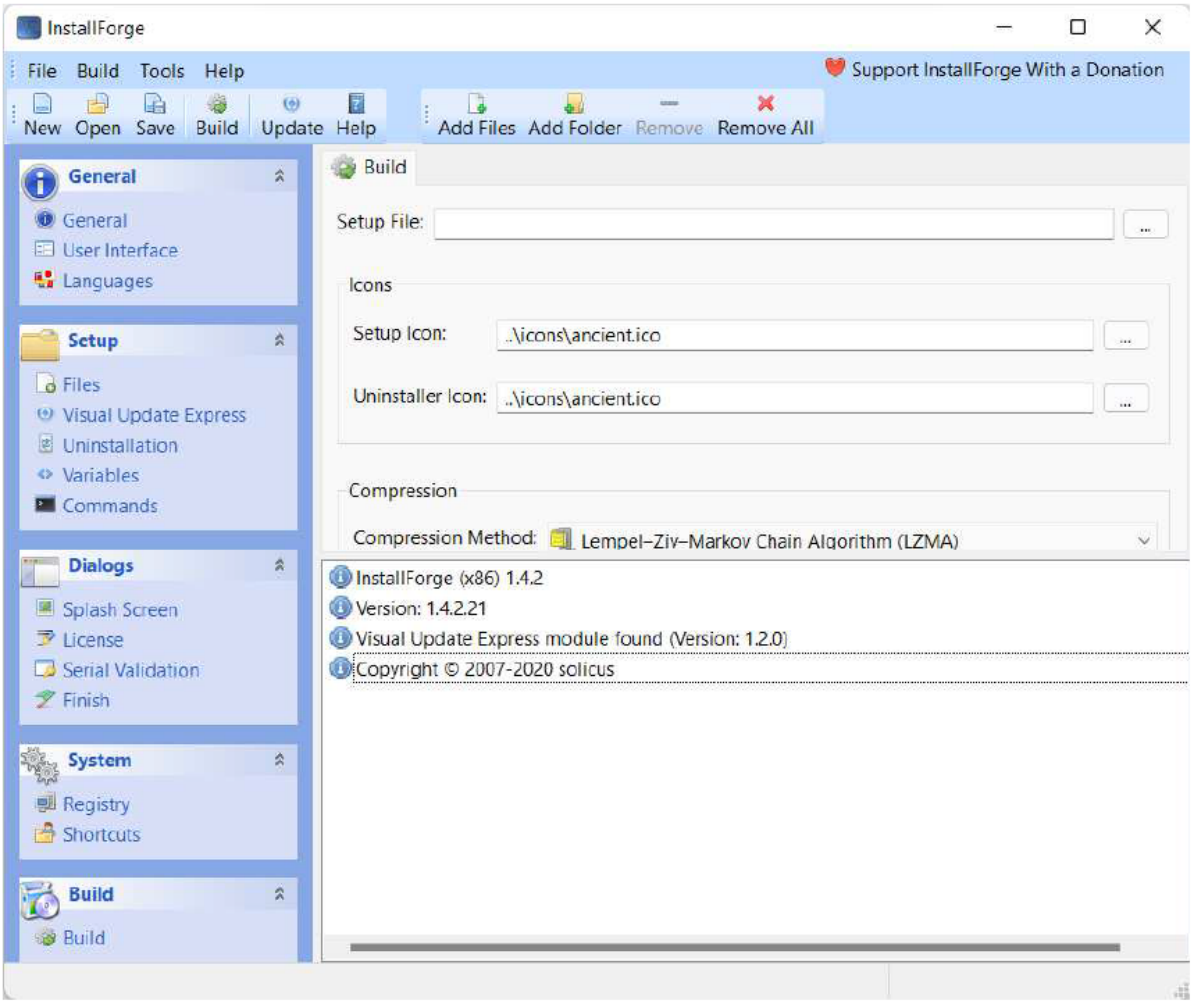
> 图260：InstallForge，添加快捷方式

构建

基本设置完成后，您现在可以开始构建安装程序了。

> 此时您可以保存您的InstallForge项目，以便将来可以从相同的设置重新构建安装程序。

点击底部“Build”部分以打开构建面板



> 图261：InstallForge，随时准备构建

点击工具栏上的“构建”图标以启动构建过程。如果您尚未指定安装程序文件的位置，系统将提示您选择一个。这是您希望保存**完成的安装程序**的位置。构建过程将开始，这会收集并压缩文件到安装程序中。

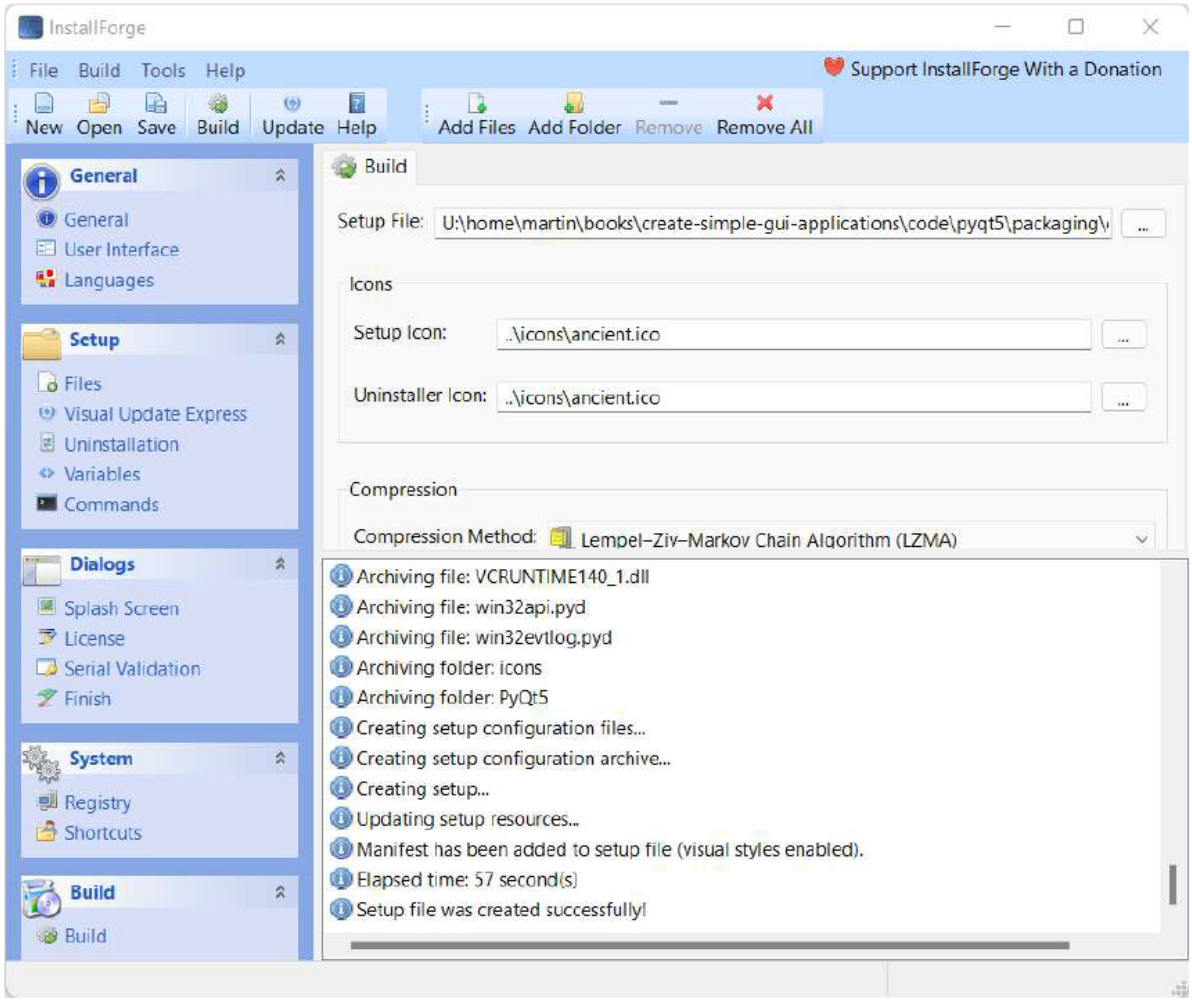


图262：InstallForge，构建完成

完成后，系统将提示您运行安装程序。这是可选项，但是相当必要，因为您可以通过这样来检验您的安装程序是否可以正常运行。

运行安装程序

安装程序本身应该不会有意外，运行正常。根据在InstallForge中选择的选项，您可能会看到额外的面板或选项。

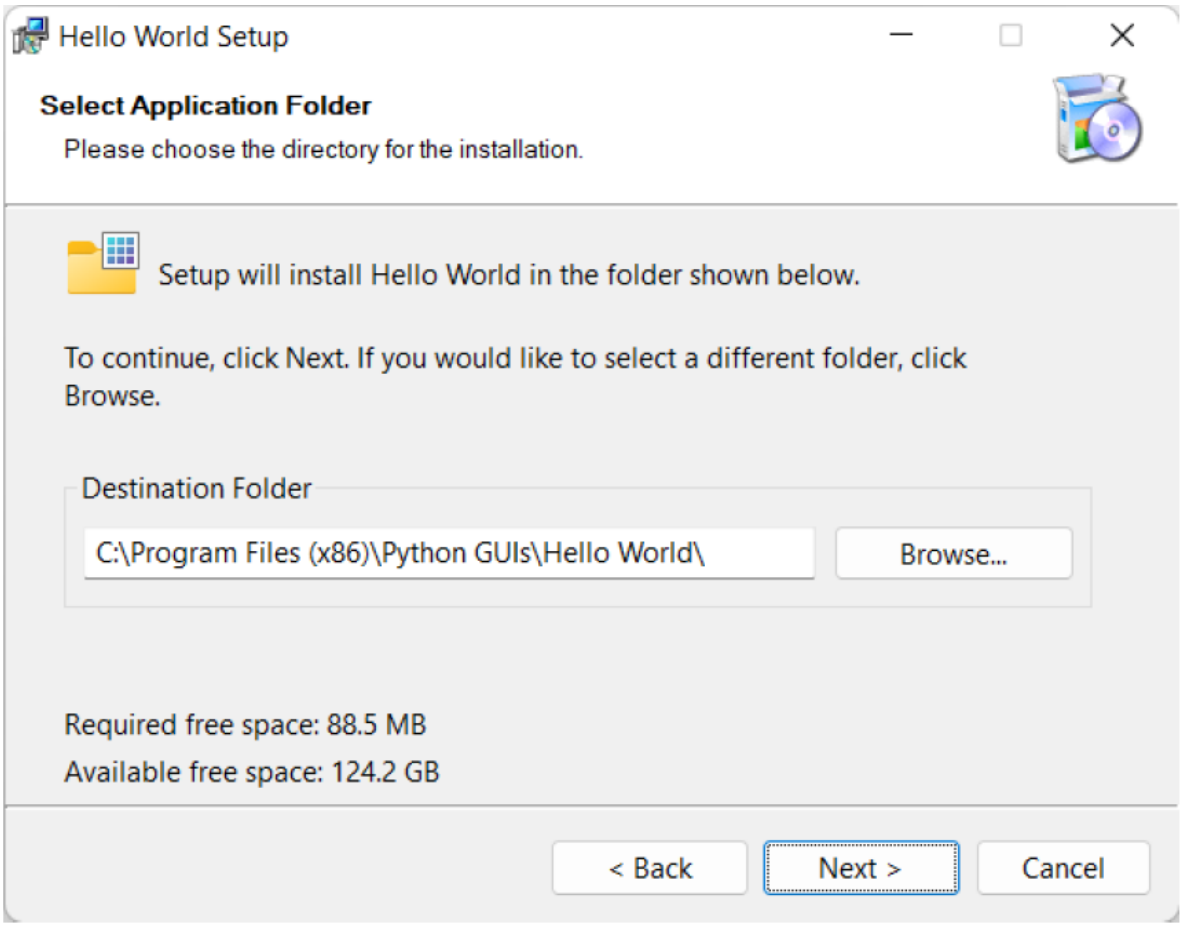
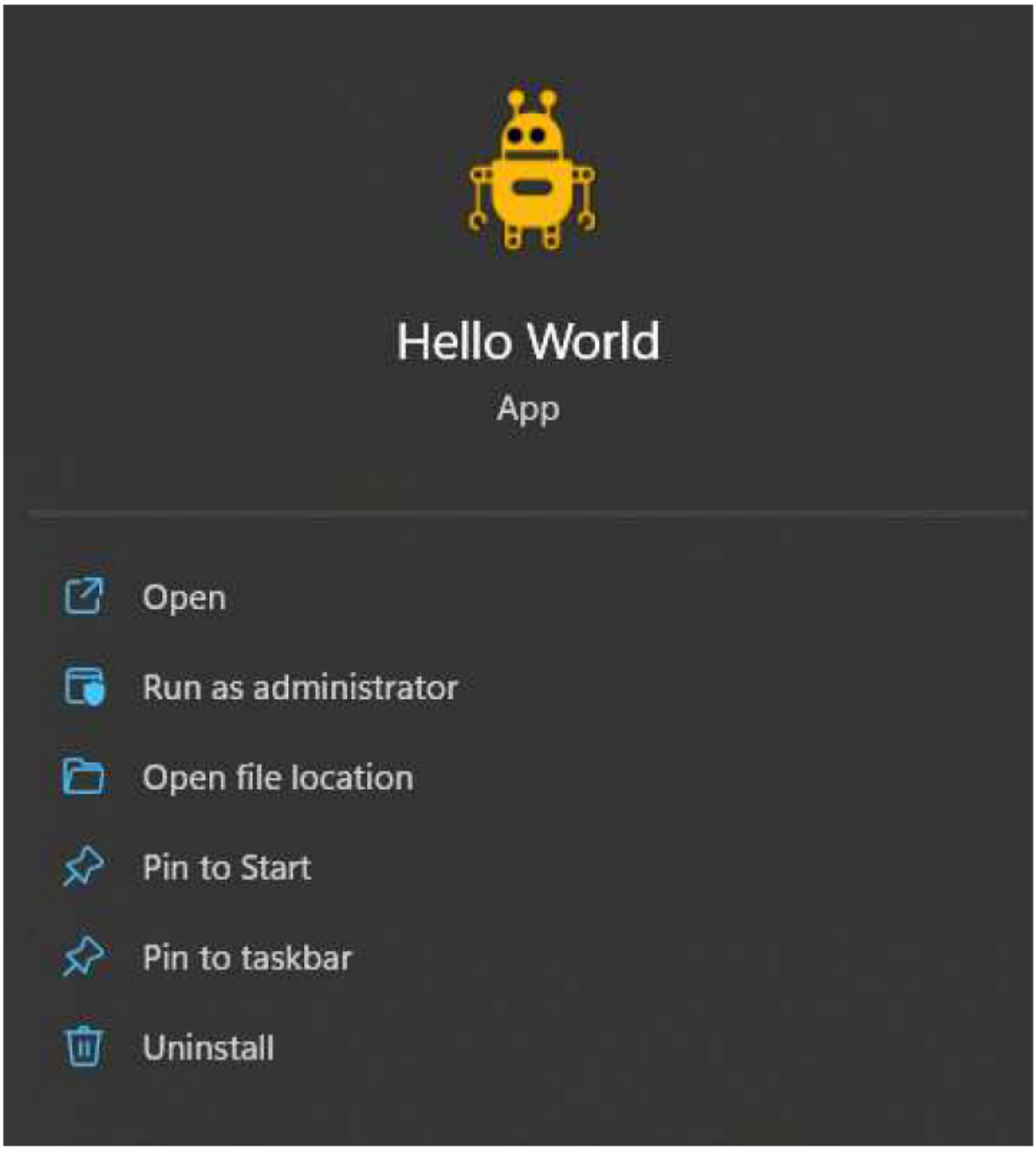


图263：InstallForge，运行生成的安装程序

请您按照安装程序的提示完成安装。您可选择在安装程序的最后一步直接运行该应用程序，或在开始菜单中找到它。



> 图264：在 Windows 11 的开始菜单中显示“Hello World”程序。

总结

在前一章中，我们介绍了如何使用 PyInstaller 将您的 PyQt6 应用程序打包成可分发的可执行文件。在本章中，我们使用已打包的 PyInstaller 应用程序，并逐步演示了如何使用 InstallForge 构建该应用程序的安装程序。按照这些步骤操作，您应该能够将自己的应用程序打包，并使其在 Windows 系统上可供他人使用。

> 另一个用于构建 Windows 安装程序的流行工具是 NSIS，它是一个可脚本化的安装程序，这意味着您可以通过编写自定义脚本来配置其行为。如果您需要频繁构建应用程序并希望自动化该过程，那么它绝对值得一试。

39. 创建 macOS 磁盘映像安装程序

在前一章中，我们使用 PyInstaller 从我们的应用程序构建了一个 macOS `.app` 文件。打开这个 `.app` 文件会运行你的应用程序，理论上您可以直接将它分发给其他人。然而，有一个问题——macOS `.app` 文件实际上只是带有特殊扩展名的文件夹。这意味着它们不适合直接分享——最终用户需要下载文件夹内的所有单独文件。

解决方案是将 `.app` 文件封装在 Zip 文件（`.zip`）或磁盘映像文件（`.dmg`）中。大多数商业软件都采用磁盘映像格式，因为这样可以包含一个快捷方式，指向用户的“应用程序”文件夹，使用户能够通过拖放操作将应用程序直接拖入该文件夹。这种做法如今已非常普遍，以至于许多用户如果遇到其他格式可能会感到困惑。我们还是遵循这一惯例吧。

> 如果您等不及了，您可以先下载 示例 macOS 磁盘映像 。

创建DMG文件

自行创建DMG文件相对简单，但我建议您首先使用可通过Homebrew安装的工具 `create-dmg` 。该工具以简单的命令行工具形式安装，您只需传入几个参数即可生成DMG安装程序。

您可以使用 Homebrew 安装 `create-dmg` 包。

```
brew install create-dmg
```

安装完成后，您即可使用 `create-dmg` Bash 脚本。以下是部分选项示例，可通过运行 `create-dmg --help` 命令查看完整列表：

```
--volname <name>: set volume name (displayed in the Finder sidebar and window
title)
--volicon <icon.icns>: set volume icon
--background <pic.png>: set folder background image (provide png, gif, jpg)
--window-pos <x> <y>: set position the folder window
--window-size <width> <height>: set size of the folder window
--text-size <text_size>: set window text size (10-16)
--icon-size <icon_size>: set window icons size (up to 128)
--icon <file_name> <x> <y>: set position of the file's icon
--hide-extension <file_name>: hide the extension of file
--app-drop-link <x> <y>: make a drop link to Applications, at location x, y
--eula <eula_file>: attach a license file to the dmg
--no-internet-enable: disable automatic mount&copy
--format: specify the final image format (default is UDZO)
--add-file <target_name> <file|folder> <x> <y>: add additional file or folder
(can be used multiple times)
--disk-image-size <x>: set the disk image size manually to x MB
--version: show tool version number
-h, --help: display the help
```

> 卷是磁盘的术语名称，因此卷名称是您希望为磁盘映像（DMG）本身命名的名称。

除了上述选项外，您还需要指定DMG文件的输出名称以及输入文件夹——即包含由PyInstaller生成的 `.app` 文件的文件夹。

下面我们将使用 `create-dmg` 工具为我们的 Hello World 应用程序创建一个安装程序 DMG 文件。我们这里只使用了部分可用选项——设置磁盘卷的名称和图标、调整窗口的位置和大小、设置应用程序的图标，以及添加 `/Applications` 目标目录链接。这是您为自己的应用程序设置的最低要求，如果您愿意，可以进一步自定义这些设置。

由于 `create-dmg` 会将指定文件夹中的所有文件复制到 DMG 中，因此您需要确保 `.app` 文件位于一个单独的文件夹中。我建议创建一个名为 `dmg` 的文件夹，并将生成的 `.app` 包复制到该文件夹中。下面我编写了一个小型脚本用于打包操作，其中包含一个测试，用于检查并删除任何之前生成的 DMG 文件。

Listing 260. packaging/installer/mac/makedmg.sh

```
#!/bin/sh
test -f "Hello World.dmg" && rm "Hello World.dmg"
test -d "dist/dmg" && rm -rf "dist/dmg"
# 创建 dmg 文件夹并复制我们的 .app 包到其中.
mkdir -p "dist/dmg"
cp -r "dist/Hello World.app" "dist/dmg"
# 创建 dmg 文件.
create-dmg \
  --volname "Hello World" \
  --volicon "icons/icon.icns" \
  --window-pos 200 120 \
  --window-size 800 400 \
  --icon-size 100 \
  --icon "Hello World.app" 200 190 \
  --hide-extension "Hello World.app" \
  --app-drop-link 600 185 \
  "Hello World.dmg" \
  "dist/dmg/"
```

请您将此内容保存到项目根目录，命名为 `build-dmg.sh`，然后将其设为可执行文件。使用以下命令：

```
$ chmod +x build-dmg.sh
```

然后执行脚本以构建包。

```
$ ./build-dmg.sh
```

`create-dmg` 进程将开始运行，并在当前目录中生成一个 DMG 文件，其名称与您为输出文件指定的名称一致（即最后一个参数，带 `.dmg` 扩展名）。现在您可以将生成的 DMG 文件分发给其他 macOS 用户了！

图265：生成的磁盘映像显示我们的 `.app` 包和Applications快捷方式。您可以将应用程序拖动到目标位置以进行安装。

有关 `create-dmg` 的更多信息，请参阅 Github 上的文档 。

40. 创建一个Linux软件包

在前一章中，我们使用 PyInstaller 将应用程序打包成一个 Linux 可执行文件，连同相关的数据文件。打包过程的输出是一个文件夹，可以与其他用户共享。然而，为了方便他们将其安装到自己的系统上，我们需要创建一个 Linux 包。

软件包是可分发的文件，允许用户在他们的 Linux 系统上安装软件。它们会自动将文件放置在正确的位置，并设置应用程序在任务栏/菜单中的条目，以便于启动应用程序。

在 Ubuntu（和 Debian）中，软件包被命名为 `.deb` 文件，在 Redhat 中是 `.rpm`，而在 Arch Linux 中是 `.pacman`。这些文件格式各不相同，但幸运的是，使用名为 fpm 的工具来构建它们的过程是相同的。fpm 是由 Jordan Issel 开发的打包系统，它可以将一个文件夹（或文件列表）组装成一个 Linux 软件包。

在本章中，我们将逐步讲解创建 Linux 软件包的步骤，以 Ubuntu 的 `.deb` 文件为例。然而，得益于 fpm 的强大功能，您也可以使用相同的方法来处理其他 Linux 系统。

> 图266：Ubuntu 软件包，用于我们的“Hello World”应用程序

> 如果您等不及了，您可以先下载 示例 Ubuntu包 。

安装 fpm

fpm 工具是用 Ruby 编写的，使用它需要安装 Ruby。安装 Ruby 可以使用系统包管理器。

```
$ sudo apt-get install ruby
```

安装 Ruby 后，您可以使用 `gem` 工具安装 `fpm` 。

```
$ gem install fpm --user-install
```

> 如果您看到一个警告，提示您没有在 PATH 中添加 `~/.local/share/gem/ruby/2.7.0/bin`，那么您就需要 将该路径添加到 您的 `.bashrc` 文件中。

安装完成后，您即可使用 `fpm`。您可以通过运行以下命令来检查它是否已安装并正常工作：

```
$ fpm --version
1.14.2
```

检查您的构建

在终端中，您可以切换到包含应用程序源文件的文件夹并运行PyInstaller 构建以生成 `dist` 文件夹。通过在文件管理器中打开 `dist` 文件夹并双击应用程序可执行文件，测试生成的构建是否按预期运行（它正常工作且图标显示正确）。

如果一切正常，您就可以开始打包应用程序了——如果出现问题，请返回并仔细检查所有内容。

> 在打包应用程序之前，始终建议先测试已构建的应用程序。这样，如果出现任何问题，您就能知道问题出在哪里！

现在让我们使用 `fpm` 打包我们的文件夹。

打包您的软件包

Linux 包用于安装各种应用程序，包括系统工具。正因如此，它们被设计成允许您将文件放置在 Linux 文件系统中的任何位置——而不同类型的文件有特定的正确存放位置。对于像我们这样的图形用户界面应用程序，我们可以将可执行文件和相关数据文件都放在同一个文件夹（`/opt`）下。但是，为了使我们的应用程序出现在菜单/搜索中，我们还需要在 `/usr/share/applications` 下安装一个 `.desktop` 文件。

确保文件最终存放在正确位置的最简单方法是，在文件夹中重建目标文件结构，然后告诉 `fpm` 使用该文件夹作为根目录进行打包。此过程也可通过脚本轻松自动化（详见后文）。

请您在项目根目录下，创建一个名为 `package` 的新文件夹及其子文件夹，这些子文件夹对应目标文件系统中的目录结构——`/opt` 将存放我们的应用程序文件夹 `helloworld`，`/usr/share/applications` 将存放我们的 `.desktop` 文件，而 `/usr/share/icons…` 将存放我们的应用程序图标。

```
$ mkdir -p package/opt
$ mkdir -p package/usr/share/applications
$ mkdir -p package/usr/share/icons/hicolor/scalable/apps
```

接下来，使用递归方式（带 `-r` 参数以包含子文件夹）将 `dist/app` 目录下的内容复制到 `package/opt/hello-world` 目录中——`/opt/hello-world` 路径是应用程序安装后的目标目录。

```
$ cp -r dist/hello-world package/opt/hello-world
```

> 我们正在复制 `dist/hello-world` 文件夹。该文件夹的名称将取决于 PyInstaller 中配置的名称。

图标

我们已经为应用程序在运行时设置了图标，使用了 `penguin.svg` 文件。然而，我们希望应用程序在 `dock/menus` 中显示其图标。要正确实现这一点，我们需要将应用程序图标复制到 `/usr/share/icons` 目录下的特定位置。

此文件夹包含系统中安装的所有图标主题，但应用程序的默认图标始终放置在备用 `hicolor` 主题中，位于 `/usr/share/icons/hicolor` 。在此文件夹内，有用于不同大小图标的多个文件夹。

```
$ ls /usr/share/icons/hicolor/
128x128/    256x256/    64x64/      scalable/
16x16/      32x32/      72x72/      symbolic/
192x192/    36x36/      96x96/
22x22/      48x48/      icon-theme.cache
24x24/      512x512/    index.theme
```

我们使用的是可缩放矢量图形（SVG）文件，因此我们的图标应放置在 `scalable` 文件夹下。如果您使用的是特定尺寸的PNG文件，请将其放置在正确的位置——并可自由添加多种不同尺寸，以确保您的应用程序图标在缩放时看起来良好。应用程序图标应放置在子文件夹 `apps` 中。

```
$ cp icons/penguin.svg
package/usr/share/icons/hicolor/scalable/apps/hello-world.svg
```

> 将图标的目标文件名命名为您的应用程序名称可以避免与其他文件冲突！这里我们将其命名为 `helloworld.svg`

`.desktop` 文件

`.desktop` 文件是一个文本配置文件，用于向 Linux 桌面系统描述一个桌面应用程序——例如，应用程序的可执行文件位置、应用程序名称以及应显示的图标。您应为应用程序包含一个 `.desktop` 文件，以使它们易于使用。以下是一个示例 `.desktop` 文件——将其添加到项目根目录中，命名为 `hello-world.desktop`，并可根据需要进行任何修改。

```
[Desktop Entry]

# 此桌面文件所指的对象类型（例如可以是链接）
Type=Application

# 应用程序名称
Name=Hello World

# 工具提示在菜单中显示
```

```
Comment=A simple Hello World application.

# 可执行文件运行的路径（文件夹）
Path=/opt/hello-world

# 可执行文件（可包含参数）
Exec=/opt/hello-world/hello-world

# 条目的图标，使用目标文件系统路径
Icon=hello-world
```

现在 `hello-world.desktop` 文件已准备就绪，我们可以将其复制到我们的安装包中。

```
$ cp hello-world.desktop package/usr/share/applications
```

权限

软件包保留了打包时已安装文件的权限，但将以 `root` 用户身份进行安装。为了让普通用户能够运行该应用程序，您需要修改所创建文件的权限。

我们可以递归地为可执行文件和文件夹应用正确的权限 755 - 所有者可读/写/执行，组/其他人可读/执行。并为所有其他库文件和图标/桌面文件应用权限 644，所有者可读/写，组/其他人可读。

```
$ find package/opt/hello-world -type f -exec chmod 644 -- {} +
$ find package/opt/hello-world -type d -exec chmod 755 -- {} +
$ find package/usr/share -type f -exec chmod 644 -- {} +
$ chmod +x package/opt/hello-world/hello-world
```

构建您的软件包

现在，我们软件包中的“文件系统”部分一切就绪，我们可以开始构建软件包本身了。

在您的终端中输入以下内容。

```
fpm -C package -s dir -t deb -n "hello-world" -v 0.1.0 -p hello-world.deb
```

按照顺序，这些参数分别是：

- `-C` 在搜索文件之前需要切换到的文件夹：我们的 `package` 文件夹
- `-s` 要打包的源类型：在我们的案例中是 `dir`，即一个文件夹。
- `-t` 要构建的包类型：Debian/Ubuntu 的 `deb` 包
- `-n` 应用程序的名称：“hello-world”
- `-v` 应用程序的版本：0.1.0
- `-p` 要输出的包名称：hello-world-deb

> 您可以通过修改 `-t` 参数来创建其他包类型（适用于其他Linux发行版）。有关更多命令行参数，请参阅 fpm文档 。

几秒钟后，您应该会看到一条消息，表明软件包已创建。

```
$ fpm -C package -s dir -t deb -n "hello-world" -v 0.1.0 -p helloworld.deb
Created package {:path=>"hello-world.deb"}
```

安装

软件包已准备就绪！让我们开始安装吧。

```
$ sudo dpkg -i hello-world.deb
```

安装完成后，您将看到一些输出内容。

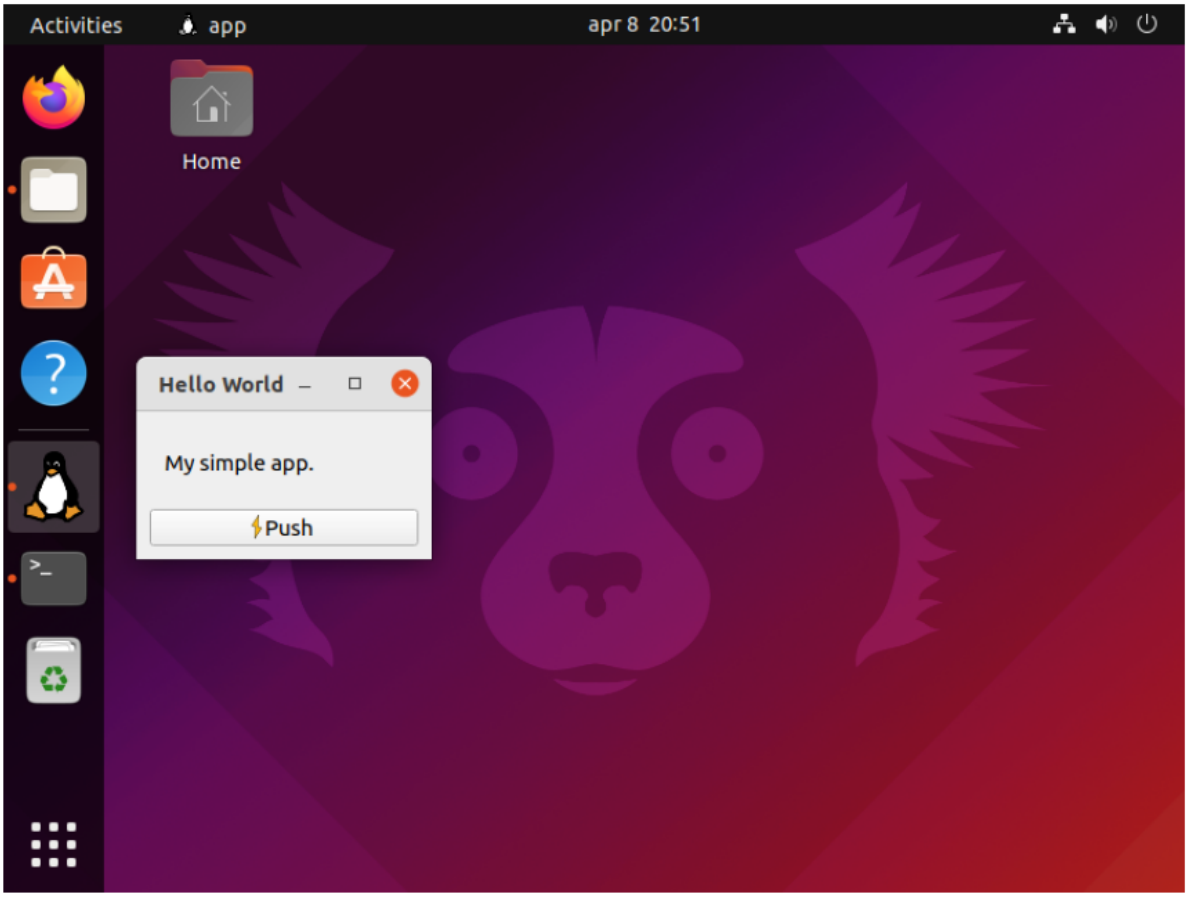
```
Selecting previously unselected package hello-world.
(Reading database ... 172208 files and directories currently installed.)
Preparing to unpack hello-world.deb ...
Unpacking hello-world (0.1.0) ...
Setting up hello-world (0.1.0) ...
```

安装完成后，您可以检查文件是否位于预期位置，位于 `/opt/hello-world` 目录下。

接下来，尝试从 `menu/Dock` 运行应用程序——您可以搜索“HelloWorld”，应用程序将被找到（得益于 `.desktop` 文件）。

> 图267：应用程序会在Ubuntu的搜索面板中显示，并且也会出现在其他环境的菜单中。

如果您运行该应用程序，图标将如预期般显示。



> 图268：应用程序正常运行，所有图标均按预期显示。

脚本化构建过程

我们已经详细介绍了如何从一个 PyQt6 应用程序构建可安装的 Ubuntu `.deb` 软件包的步骤。虽然一旦您掌握了方法，这个过程相对简单，但如果您需要经常进行这项工作，它可能会变得相当繁琐且容易出错。

为了避免问题，我建议使用简单的 Bash 脚本与 fpm 的自动化工具进行脚本化处理。

`package.sh`

请您将文件保存到项目根目录，并使用 `chmod +x` 命令使其可执行。

Listing 262. packaging/installer/linux/package.sh

```
#!/bin/sh
# 创建文件夹.
[ -e package ] && rm -r package
mkdir -p package/opt
mkdir -p package/usr/share/applications
mkdir -p package/usr/share/icons/hicolor/scalable/apps

# 复制文件（更改图标名称，为非缩放图标添加行）
cp -r dist/hello-world package/opt/hello-world
cp icons/penguin.svg
package/usr/share/icons/hicolor/scalable/apps/hello-world.svg
cp hello-world.desktop package/usr/share/applications

# 更改权限
find package/opt/hello-world -type f -exec chmod 644 -- {} +
```

```
find package/opt/hello-world -type d -exec chmod 755 -- {} +
find package/usr/share -type f -exec chmod 644 -- {} +
chmod +x package/opt/hello-world/hello-world
```

`.fpm` 文件

`fpm` 允许您将打包配置存储在配置文件中。文件名必须为 `.fpm`，且必须位于运行 fpm 工具的文件夹中。我们的配置如下。

Listing 263. packaging/installer/linux/.fpm

```
-C package
-s dir
-t deb
-n "hello-world"
-v 0.1.0
-p hello-world.deb
```

> 在执行 fpm 时，您可以通过传递命令行参数来覆盖任何您喜欢的选项，就像平时一样。

执行构建

有了这些脚本，我们的应用程序可以使用以下命令进行可重复打包：

```
pyinstaller hello-world.spec
./package.sh
fpm
```

您可以根据自己的项目需求，自由地进一步自定义这些构建脚本！

在本章中，我们详细介绍了如何从PyInstaller获取一个可运行的构建版本，并使用 `fpm` 将其打包成适用于Ubuntu的可分发Linux包。按照这些步骤操作，您应该能够将自己的应用程序打包，并使其可供他人使用。

应用程序示例

到目前为止，您应该已经掌握了如何使用PyQt6构建简单应用程序的方法。为了展示如何将所学知识付诸实践，本章中包含了几个示例应用程序。这些应用程序功能齐全、简单易用，但在某些方面可能不够完善。您可以将它们作为灵感来源，进行拆解分析，并借此机会进行改进。请您继续阅读，我们将对每个应用程序的精彩部分进行详细讲解。

这两个应用程序的完整源代码均可下载，此外，您还可以在我的GitHub上的 15分钟应用程序 仓库中找到另外13个应用程序。祝您玩得愉快！

本书中还有其他一些微型应用程序的示例——例如绘图和待办事项应用程序——我鼓励您也要扩展这些应用程序，这是学习的最佳方式。

41. Mozzarella Ashbadger

Mozzarella Ashbadger 是网页浏览领域的最新革命！返回和前进！打印！保存文件！获取帮助！（您可能会需要它）。与其他浏览器的任何相似之处纯属巧合。

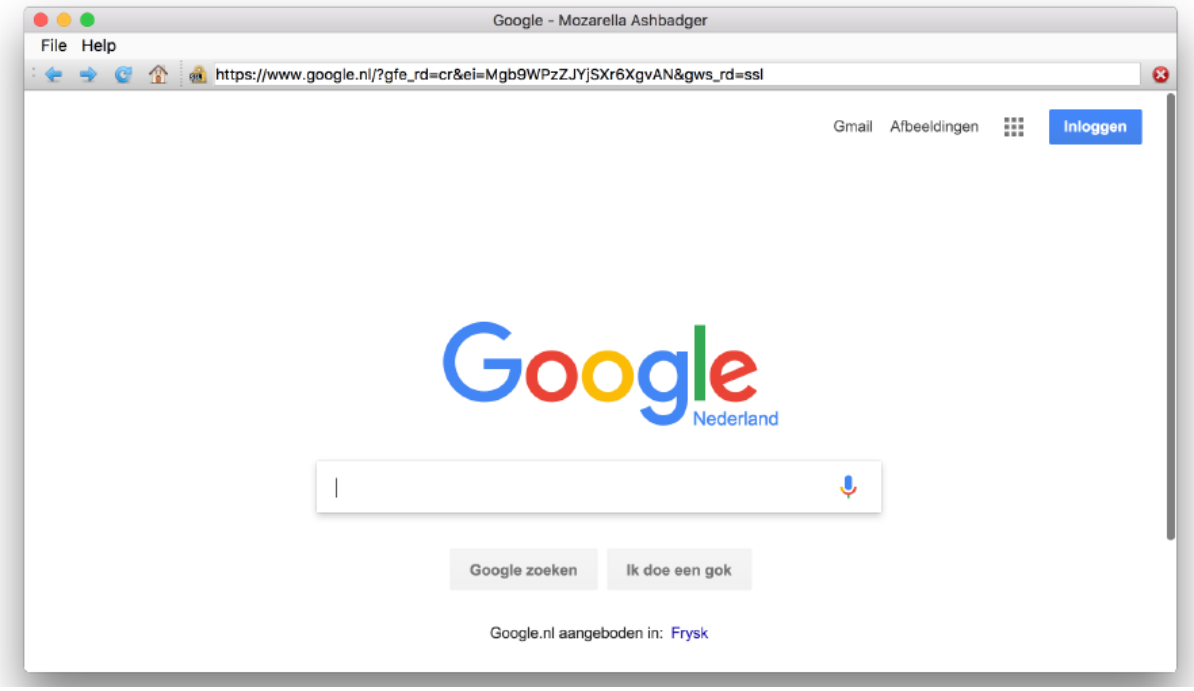


图269：Mozzarella Ashbadger

此应用程序利用了信号与槽、扩展信号和控件中介绍的功能。

Mozzarella Ashbadger 的源代码有两种形式，一种带有标签式浏览，另一种没有。添加标签会稍微增加信号处理的复杂性，因此我们首先介绍没有标签的版本。

要创建浏览器，我们需要安装一个额外的 PyQt6 组件 — PyQtWebEngine。您可以通过命令行使用 `pip` 进行安装，具体步骤如下：

```
pip3 install pyqt6-webengine
```

源代码

无标签浏览器的完整源代码包含在本书的下载内容中。浏览器代码的文件名为 `browser.py`。

```
python3 browser.py
```

🚀 **运行它吧！** 在开始编写代码之前，请先探索 Mozzarella Ashbadger 的界面和功能。

浏览器控件

浏览器的核心是 `QWebEngineView`，我们从 `QtWebEngineWidgets` 导入它。它提供了一个完整的浏览器窗口，该窗口负责滑块下载页面的渲染。以下是 PyQt6 中使用网页浏览器控件所需的最低限度代码。

Listing 264. app/browser_skeleton.py

```
import sys

from PyQt6.QtCore import QUrl
from PyQt6.QtWebEngineWidgets import QWebEngineView
from PyQt6.QtWidgets import QApplication, QMainWindow


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.browser = QWebEngineView()
        self.browser.setUrl(QUrl("https://www.google.com"))

        self.setCentralWidget(self.browser)

        self.show()


app = QApplication(sys.argv)
window = MainWindow()

app.exec()
```

如果您稍微点击一下，您会发现浏览器表现得像预期的一样——链接正常工作，您可以与页面互动。然而，您也会注意到一些你习以为常的东西缺失了——比如地址栏、控制按钮或任何类型的界面。这使得使用起来有点棘手。

让我们把这个简陋的浏览器改造成一个稍微实用一点的工具吧！

路径

为了更方便地处理界面图标，我们可以首先定义一个“使用相对路径”（参见前面的章节）。它为 `icons` 数据文件和一个 `icon` 方法定义了一个单一的文件夹位置，用于创建图标的路径。这使我们能够使用 `Paths.icon()` 来加载浏览器界面的图标。

Listing 265. app/paths.py

```
import os


class Paths:

    base = os.path.dirname(__file__)
    icons = os.path.join(base, "icons")

    # 文件加载器.
    @classmethod
    def icon(cls, filename):
        return os.path.join(cls.icons, filename)
```

请您将它与我们的浏览器保存在同一文件夹中，它可以导入为：

Listing 266. app/browser.py

```
from paths import Paths
```

导航

现在这些都已就绪，我们可以添加一些界面控件，例如在 `QToolbar` 上使用一系列的 `QActions` 。我们将这些定义添加到 `QMainWindow` 的 `__init__` 块中。我们使用我们的 `Paths.icon()` 方法使用相对路径加载文件。

Listing 267. app/browser.py

```
        navtb = QToolBar("Navigation")
        navtb.setIconSize(QSize(16, 16))
        self.addToolBar(navtb)

        back_btn = QAction(
            QIcon(Paths.icon("arrow-180.png")), "Back", self
        )
        back_btn.setStatusTip("Back to previous page")
        back_btn.triggered.connect(self.browser.back)
        navtb.addAction(back_btn)
```

`QWebEngineView` 包括用于前进、后退和重新加载导航的槽，我们可以将其直接连接到我们的动作的 `.triggered` 信号。

我们为剩余的控件使用相同的 `QAction` 结构。

Listing 268. app/browser.py

```
        next_btn = QAction(
            QIcon(Paths.icon("arrow-000.png")), "Forward", self
        )
        next_btn.setStatusTip("Forward to next page")
        next_btn.triggered.connect(self.browser.forward)
        navtb.addAction(next_btn)

        reload_btn = QAction(
            QIcon(Paths.icon("arrow-circle-315.png")),
```

```
            "Reload",
            self,
        )
        reload_btn.setStatusTip("Reload page")
        reload_btn.triggered.connect(self.browser.reload)
        navtb.addAction(reload_btn)

        home_btn = QAction(QIcon(Paths.icon("home.png")), "Home", self)
        home_btn.setStatusTip("Go home")
        home_btn.triggered.connect(self.navigate_home)
        navtb.addAction(home_btn)
```

请注意，虽然前进、后退和重新加载可以使用内置槽，但导航主页按钮需要自定义槽函数。该槽函数在我们的 `QMainWindow` 类中定义，只需将浏览器的URL设置为谷歌主页即可。请注意，URL必须作为 `QUrl` 对象传递。

Listing 269. app/browser.py

```
    def navigate_home(self):
        self.browser.setUrl(QUrl("http://www.google.com"))
```

> 挑战
>
> 尝试将主页导航位置设置为可配置。您可以创建一个带输入字段的 `QDialog` 对话框。

任何一款合格的网页浏览器都必须具备地址栏，并且需要提供一种方式来停止导航——无论是由于误操作，还是页面加载过慢。

Listing 270. app/browser.py

```
        self.httpsicon = QLabel() # 是的，就像这样！
        self.httpsicon.setPixmap(QPixmap(Paths.icon("lock-nossl.png")))
        navtb.addWidget(self.httpsicon)

        self.urlbar = QLineEdit()
        self.urlbar.returnPressed.connect(self.navigate_to_url)
        navtb.addWidget(self.urlbar)

        stop_btn = QAction(
            QIcon(Paths.icon("cross-circle.png")), "Stop", self
        )
        stop_btn.setStatusTip("Stop loading current page")
        stop_btn.triggered.connect(self.browser.stop)
        navtb.addAction(stop_btn)
```

与之前一样，`QWebEngineView` 上提供了“停止”功能，我们只需将停止按钮的 `.triggered` 信号连接到现有的槽即可。但是，URL 栏的其他功能必须单独处理。

首先，我们添加一个 `QLabel` 来保存我们的 SSL 或非 SSL 图标，以指示页面是否安全。接下来，我们添加一个 URL 栏，它只是一个 `QLineEdit`。为了触发在输入（按回车键）时在栏中加载 URL，我们连接到控件上的 `.returnPressed` 信号，以驱动一个自定义槽函数，触发导航到指定的 URL。

Listing 271. app/browser.py

```
    def navigate_to_url(self): # 未接收 URL
        q = QUrl(self.urlbar.text())
        if q.scheme() == "":
            q.setScheme("http")

        self.browser.setUrl(q)
```

我们还希望 URL 条能够根据页面变化进行更新。为此，我们可以使用 `QWebEngineView` 的 `.urlChanged` 和 `.loadFinished` 信号。我们在 `__init__` 块中按照以下方式设置了信号的连接：

Listing 272. app/browser.py

```
        self.browser.urlChanged.connect(self.update_urlbar)
        self.browser.loadFinished.connect(self.update_title)
```

然后，我们定义这些信号的目标槽函数。第一个函数用于更新 URL 栏，它接受一个 `QUrl` 对象，并确定这是 `http` 还是 `https` URL，然后使用此信息设置 SSL 图标。

> 这是一种非常糟糕的测试连接是否“安全”的方法。要正确地进行测试，我们应该执行证书验证。

`QUrl` 被转换为字符串，URL 栏也更新为该值。请注意，我们还将光标位置设置回行首，以防止 `QLineEdit` 控件滚动到行尾。

Listing 273. app/browser.py

```
    def update_urlbar(self, q):

        if q.scheme() == "https":
            # 安全挂锁图标
            self.httpsicon.setPixmap(
                QPixmap(Paths.icon("lock-ssl.png"))
            )

        else:
            # 不安全的挂锁图标
            self.httpsicon.setPixmap(
                QPixmap(Paths.icon("lock-nossl.png"))
            )

        self.urlbar.setText(q.toString())
```

```
        self.urlbar.setCursorPosition(0)
```

另外，将应用程序窗口的标题更新为当前页面的标题也是一个不错的细节。我们可以使用 `browser.page().title()` 方法获取此信息，该方法返回当前加载的网页中 `<title></title>` 标签的内容。

Listing 274. app/browser.py

```
    def update_title(self):
        title = self.browser.page().title()
        self.setWindowTitle("%s - Mozzarella Ashbadger" % title)
```

文件操作

使用 `self.menuBar().addMenu("&File")` 可以创建一个标准的"文件"菜单，将 F 键分配为 Alt 快捷键（与通常一样）。获得菜单对象后，我们可以将 `QAction` 对象分配给该对象以创建条目。我们在这里创建了两个基本条目，用于打开和保存 HTML 文件（来自本地磁盘）。这两个条目都需要自定义槽函数。

Listing 275. app/browser.py

```
        file_menu = self.menuBar().addMenu("&File")

        open_file_action = QAction(
            QIcon(Paths.icon("disk--arrow.png")),
            "Open file...",
            self,
        )
        open_file_action.setStatusTip("Open from file")
        open_file_action.triggered.connect(self.open_file)
        file_menu.addAction(open_file_action)

        save_file_action = QAction(
            QIcon(Paths.icon("disk--pencil.png")),
            "Save Page As...",
            self,
        )
        save_file_action.setStatusTip("Save current page to file")
        save_file_action.triggered.connect(self.save_file)
        file_menu.addAction(save_file_action)
```

打开文件的槽函数使用内置的 `QFileDialog.getOpenFileName()` 函数创建一个文件打开对话框并获取一个名称。我们默认将名称限制为与 `*.htm` 或 `*.html` 匹配的文件。

我们使用标准的 Python 函数将文件读取到一个名为 `html` 的变量中，然后使用 `.setHtml()` 将 HTML 内容加载到浏览器中。

Listing 276. app/browser.py

```
    def open_file(self):
        filename, _ = QFileDialog.getOpenFileName(
            self,
            "Open file",
            "",
            "Hypertext Markup Language (*.htm *.html);;"
```

```
            "All files (*.*)",
        )

        if filename:
            with open(filename, "r") as f:
                html = f.read()

            self.browser.setHtml(html)
            self.urlbar.setText(filename)
```

同样地，为了保存当前页面的 HTML，我们使用内置的 `QFileDialog.getSaveFileName()` 方法获取文件名。不过这次我们通过 `self.browser.page().toHtml()` 获取 HTML。

这是一个异步方法，这意味着我们不会立即收到 HTML。相反，我们必须传递一个回调方法，该方法将在 HTML 准备就绪后接收它。在这里，我们创建了一个简单的 `writer` 函数，它使用本地范围中的文件名来处理它。

Listing 277. app/browser.py

```
    def save_file(self):
        filename, _ = QFileDialog.getSaveFileName(
            self,
            "Save Page As",
            "",
            "Hypertext Markup Language (*.htm *html);;"
            "All files (*.*)",
        )

        if filename:
            # 定义回调方法以处理写入操作.
            def writer(html):
                with open(filename, "w") as f:
                    f.write(html)

            self.browser.page().toHtml(writer)
```

打印

我们可以使用之前的方法在“文件”菜单中添加打印选项。同样，这需要一个自定义槽函数来执行打印操作。

Listing 278. app/browser.py

```
        print_action = QAction(
            QIcon(Paths.icon("printer.png")), "Print...", self
        )
        print_action.setStatusTip("Print current page")
        print_action.triggered.connect(self.print_page)
        file_menu.addAction(print_action)

        # 创建我们的系统打印机实例.
        self.printer = QPrinter()
```

Qt 提供了一个基于 `QPrinter` 对象的完整打印框架，您可以在其上绘制要打印的页面。为了启动打印过程，我们首先为用户打开一个 `QPrintDialog`。这允许用户选择目标打印机并配置打印设置。

我们在 `__init__` 中创建了 `QPrinter` 对象，并将其存储为 `self.printer`。在我们的打印处理方法中，我们将此打印机传递给 `QPrintDialog`，以便其可以被配置。如果对话框被接受，我们将（现在已配置的）打印机对象传递给 `self.browser.page().print` 以触发打印。

Listing 279. app/browser.py

```
def print_page(self):
    page = self.browser.page()

    def callback(*args):
        pass

    dlg = QPrintDialog(self.printer)
    dlg.accepted.connect(callback)
    if dlg.exec() == QDialog.DialogCode.Accepted:
        page.print(self.printer, callback)
```

请注意，`.print` 方法还接受第二个参数——一个回调函数，该函数会接收打印操作的结果。这允许您在打印操作完成后显示一条通知，但在此示例中，我们只是静默地忽略了回调函数。

帮助

最后，为了完成标准界面，我们可以添加一个“帮助”菜单。它与之前一样，定义为两个自定义槽函数，一个用于显示“关于”对话框，另一个用于加载包含更多信息的“浏览器页面”。

Listing 280. app/browser.py

```
        help_menu = self.menuBar().addMenu("&Help")
        about_action = QAction(
            QIcon(Paths.icon("question.png")),
            "About Mozzarella Ashbadger",
            self,
        )
        about_action.setStatusTip(
            "Find out more about Mozzarella Ashbadger"
        ) # 我真饿了！
        about_action.triggered.connect(self.about)
        help_menu.addAction(about_action)
        navigate_mozzarella_action = QAction(
            QIcon(Paths.icon("lifebuoy.png")),
            "Mozzarella Ashbadger Homepage",
            self,
        )
        navigate_mozzarella_action.setStatusTip(
            "Go to Mozzarella Ashbadger Homepage"
        )
        navigate_mozzarella_action.triggered.connect(
            self.navigate_mozzarella
        )
        help_menu.addAction(navigate_mozzarella_action)
```

我们定义了两个方法，作为"帮助"菜单信号的槽。第一个 `navigate_mozzarella` 打开一个页面，提供有关浏览器（或在本书的本例中）的更多信息。第二个创建并执行一个自定义的 `QDialog` 类 `AboutDialog`，我们将在下面进行定义。

Listing 281. app/browser.py

```
    def navigate_mozzarella(self):
        self.browser.setUrl(QUrl("https://www.pythonguis.com/"))

    def about(self):
        dlg = AboutDialog()
        dlg.exec()
```

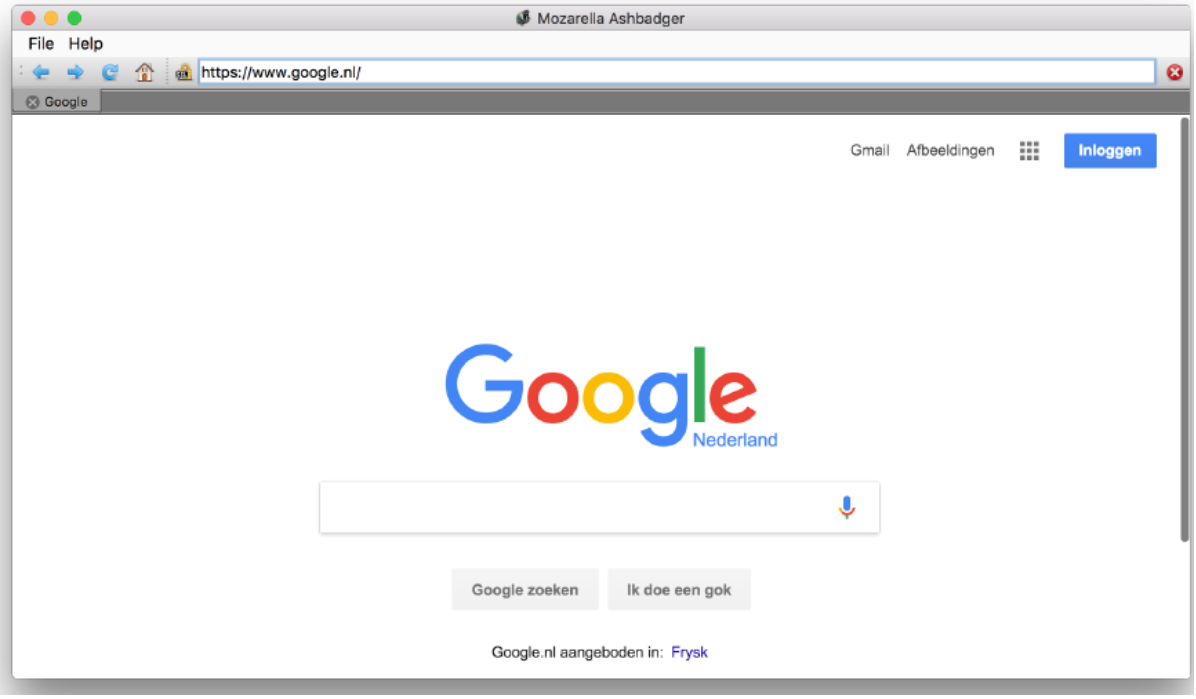
关于对话框的定义如下。该结构与本书前面介绍的结构相似，使用 `QDialogButtonBox` 和相关信号来处理用户输入，并使用一系列 `QLabels` 来显示应用程序信息和徽标。

这里唯一的技巧是将所有元素添加到布局中，然后在单个循环中遍历它们，将对齐方式设置为居中。这样可以避免在各个部分中重复设置。

Listing 282. app/browser.py

```
class AboutDialog(QDialog):
    def __init__(self):
        super().__init__()

        QBtn = QDialogButtonBox.StandardButton.Ok # 不取消
        self.buttonBox = QDialogButtonBox(QBtn)
        self.buttonBox.accepted.connect(self.accept)
        self.buttonBox.rejected.connect(self.reject)

        layout = QVBoxLayout()

        title = QLabel("Mozzarella Ashbadger")
        font = title.font()
        font.setPointSize(20)
        title.setFont(font)

        layout.addWidget(title)

        logo = QLabel()
        logo.setPixmap(QPixmap(Paths.icon("ma-icon-128.png")))
        layout.addWidget(logo)

        layout.addWidget(QLabel("Version 23.35.211.233232"))
        layout.addWidget(QLabel("Copyright 2015 Mozzarella Inc."))

        for i in range(0, layout.count()):
            layout.itemAt(i).setAlignment(Qt.AlignmentFlag.AlignHCenter)

        layout.addWidget(self.buttonBox)

        self.setLayout(layout)
```

标签式浏览



> 图270：标签化的Mozzarella Ashbadger

源代码

本书的下载内容中包含了带标签页浏览器的完整源代码。浏览器代码的文件名为 `browser_tabs.py`。

> 🚀 **运行它吧！** 在开始编写代码之前，请先探索标签化的 Mozzarella Ashbadger 的界面和功能。

创建一个 `QTabWidget`

使用 `QTabWidget` 可以轻松地为浏览器添加标签页界面。它为多个控件（在本例中为 `QWebEngineView` 控件）提供了一个简单的容器，并内置了用于在控件之间切换的标签页界面。

我们在这里使用的两个自定义设置是 `.setDocumentMode(True)`，它在 macOS 上提供一个类似 Safari 的界面，以及 `.setTabsClosable(True)`，它允许用户在应用程序中关闭标签页。

我们还将 `QTabWidget` 信号 `tabBarDoubleClicked`、`currentChanged` 和 `tabCloseRequested` 连接到自定义槽方法，以处理这些行为。

Listing 283. app/browser_tabs.py

```
        self.tabs = QTabWidget()
        self.tabs.setDocumentMode(True)
        self.tabs.tabBarDoubleClicked.connect(self.tab_open_doubleclick)
        self.tabs.currentChanged.connect(self.current_tab_changed)
        self.tabs.setTabsClosable(True)
        self.tabs.tabCloseRequested.connect(self.close_current_tab)

        self.setCentralWidget(self.tabs)
```

这三种槽方法都接受一个 `i`（索引）参数，该参数指示信号来自哪个选项卡（按顺序）。

我们双击标签栏中的空白处（由索引 `-1` 表示）来触发新标签的创建。要删除标签，我们直接使用索引来删除控件（以及标签），并进行简单的检查以确保至少有 2 个标签——关闭最后一个标签会导致您无法打开新标签。

`current_tab_changed` 处理程序使用 `self.tabs.currentWidget()` 结构来访问当前活动标签页的控件（`QWebEngineView` 浏览器），然后使用它来获取当前页面的 URL。

Listing 284. app/browser_tabs.py

```
    def tab_open_doubleclick(self, i):
        if i == -1: # N点击后没有选项卡
            self.add_new_tab()

    def current_tab_changed(self, i):
        qurl = self.tabs.currentWidget().url()
        self.update_urlbar(qurl, self.tabs.currentWidget())
        self.update_title(self.tabs.currentWidget())

    def close_current_tab(self, i):
        if self.tabs.count() < 2:
            return

        self.tabs.removeTab(i)
```

Listing 285. app/browser_tabs.py

```
    def add_new_tab(self, qurl=None, label="Blank"):

        if qurl is None:
            qurl = QUrl("")

        browser = QWebEngineView()
        browser.setUrl(qurl)
        i = self.tabs.addTab(browser, label)

        self.tabs.setCurrentIndex(i)
```

信号和槽的变化

虽然 `QTabWidget` 和相关信号的设置很简单，但在浏览器槽方法中，事情就变得稍微复杂一些了。

以前我们只有一个 `QWebEngineView`，现在有多个视图，每个视图都有自己的信号。如果处理隐藏标签的信号，事情就会变得一团糟。例如，处理 `loadCompleted` 信号的槽必须检查源视图是否在可见标签中。

我们可以使用发送附加数据信号的技巧来实现这一点。在标签式浏览器中，我们使用 `lambda` 样式语法来实现这一点。

以下是在创建新的 `QWebEngineView` 时，在 `add_new_tab` 函数中实现此功能的示例：

Listing 286. app/browser_tabs.py

```
        # 更复杂了！我们只希望在URL来自正确标签页时进行更新。
        browser.urlChanged.connect(
            lambda qurl, browser=browser: self.update_urlbar(
                qurl, browser
            )
        )

        browser.loadFinished.connect(
            lambda _, i=i, browser=browser: self.tabs.setTabText(
                i, browser.page().title()
            )
        )
```

如您所见，我们将 `lambda` 设置为 `urlChanged` 信号的槽，接受该信号发送的 `qurl` 参数。我们将最近创建的 `browser` 对象添加到 `update_urlbar` 函数中。

结果是，每当这个 `urlChanged` 信号触发时，`update_urlbar` 将同时收到新 URL 和它来自的浏览器。在槽方法中，我们可以检查以确保信号的来源与当前可见的浏览器相匹配，如果不匹配，我们只需丢弃该信号即可。

Listing 287. app/browser_tabs.py

```
    def update_urlbar(self, q, browser=None):
        if browser != self.tabs.currentWidget():
            # 如果该信号不是来自当前选项卡，则忽略它
            return

        if q.scheme() == "https":
            # 安全挂锁图标
            self.httpsicon.setPixmap(
                QPixmap(Paths.icon("lock-ssl.png"))
            )

        else:
            # 不安全的挂锁图标
            self.httpsicon.setPixmap(
                QPixmap(Paths.icon("lock-nossl.png"))
            )

        self.urlbar.setText(q.toString())
        self.urlbar.setCursorPosition(0)
```

继续深入

现在您可以探索浏览器标签页版本的其余源代码，请特别注意 `self.tabs.currentWidget()` 的使用以及通过信号传递额外数据。这是您所学知识的一个很好的实际应用案例，所以尝试一下，看看您能否以有趣的方式打破/改进它。

> 挑战
>
> 您可能想尝试添加一些额外功能——
>
> - 书签（或收藏夹）——您可以将这些存储在一个简单的文本文件中，并在菜单中显示它们
> - 网站图标（Favicons）——这些小小的网站图标，在标签页上看起来会非常棒。
> - 查看源代码 ——添加一个菜单选项以查看页面源代码。
> - 在新标签页中打开 ——添加右键点击上下文菜单，或键盘快捷键，以在新标签页中打链接

42. Moonsweeper

探索神秘的 Q'tee 卫星，但不要太靠近外星人！

Moonsweeper 是一款单人解谜游戏。游戏的目标是探索您着陆的太空火箭周围的区域，同时避免过于靠近致命的 B'ug 外星人。您可靠的计数器会告诉你附近有多少 B'ug。

> 推荐阅读
>
> 此应用程序利用了信号与槽以及事件中的功能。

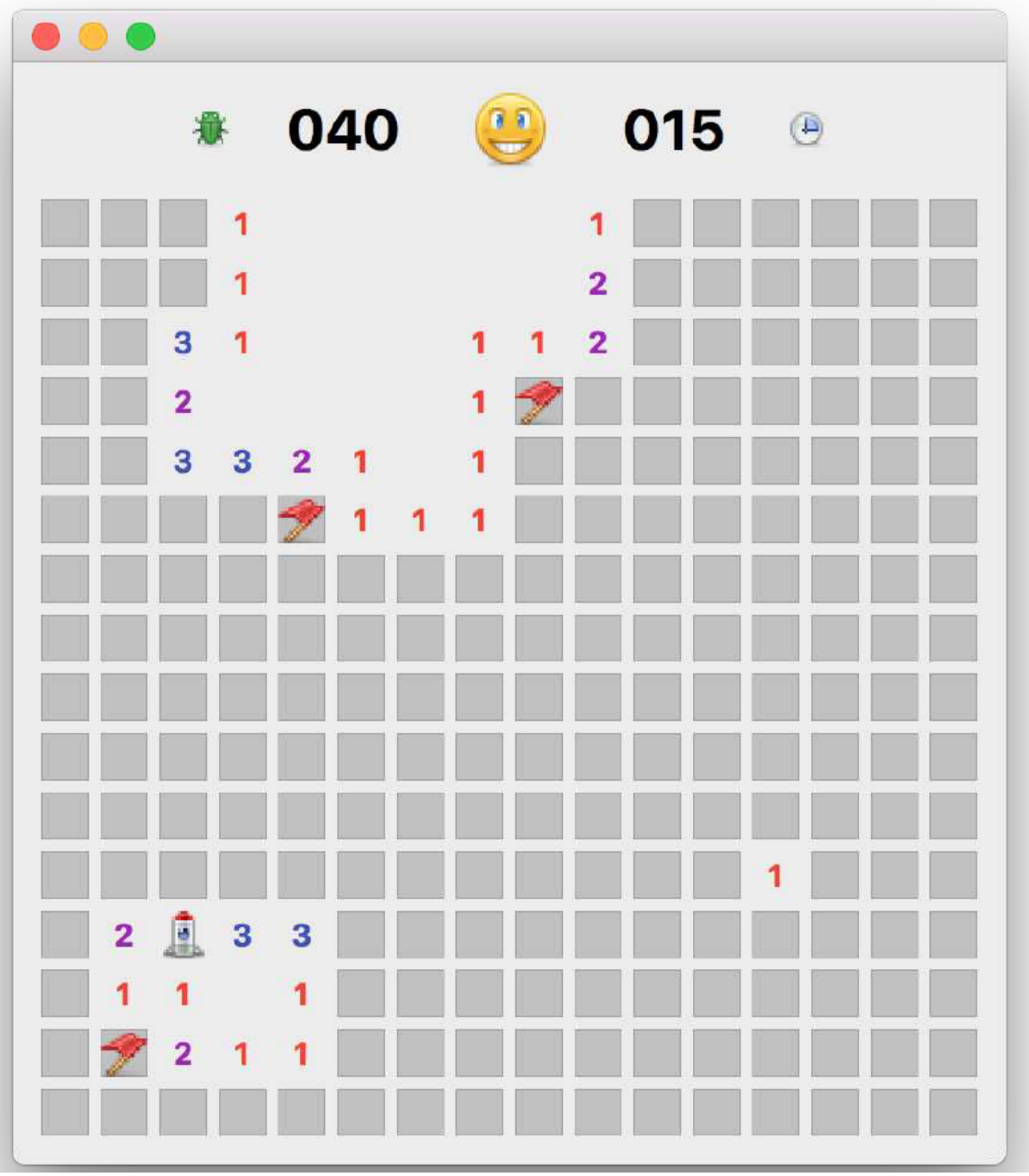


图271：Moonsweeper

这是一个以扫雷游戏为原型的简单单人探索游戏，其中您必须揭开所有方块而不触发隐藏的地雷。此实现使用自定义的 `QWidget` 对象表示方块，每个方块单独保存其状态，包括是否为地雷、当前状态以及相邻地雷的数量。在此版本中，地雷被替换为外星虫子（B'ug），但它们也可以是任何其他物体。

在许多扫雷变体中，初始回合被视为安全回合——如果您在第一次点击时触碰到地雷，它会被移动到其他地方。在这里我们稍微作弊一下，将玩家的首次操作固定在非地雷位置。这样做可以避免因首次操作错误而需要重新计算相邻格子的情况。我们可以将此解释为"火箭周围的初始探索"，使其听起来完全合理。

> 挑战！
>
> 如果您想实现这一点，可以在位置上捕获第一次点击，然后在处理点击之前生成地雷/相邻位置，但不包括您的位置。您需要让自定义控件访问父窗口对象

源代码

Moonsweeper 游戏的完整源代码包含在本书的下载内容中。游戏文件以 `minesweeper.py` 的名称保存。

```
python3 minesweeper.py
```

路径

为了更方便地处理界面图标，我们可以首先定义一个“使用相对路径”（参见前面的章节）。它为 `icons` 数据文件和一个 `icon` 方法定义了一个单一的文件夹位置，用于创建图标的路径。这使我们能够使用 `Paths.icon()` 来加载游戏界面的图标。

Listing 288. app/paths.py

```
import os


class Paths:

    base = os.path.dirname(__file__)
    icons = os.path.join(base, "icons")

    # 文件加载器.
    @classmethod
    def icon(cls, filename):
        return os.path.join(cls.icons, filename)
```

与我们的 Moonsweeper 应用程序保存在同一文件夹中，它可以被导入为：

Listing 289. app/moonsweeper.py

```
from paths import Paths
```

图标与颜色

现在路径已经定义，我们可以使用它们来加载一些图标，用于我们的游戏——一个虫子、一面旗帜、一枚火箭和一个时钟。我们还定义了一组颜色用于界面状态，以及一系列状态标志来跟踪游戏的进展——每个标志都关联一个笑脸图标。

Listing 290. app/moonsweeper.py

```
IMG_BOMB = QImage(Paths.icon("bug.png"))
IMG_FLAG = QImage(Paths.icon("flag.png"))
IMG_START = QImage(Paths.icon("rocket.png"))
IMG_CLOCK = QImage(Paths.icon("clock-select.png"))

NUM_COLORS = {
    1: QColor("#f44336"),
```

```
    2: QColor("#9C27B0"),
    3: QColor("#3F51B5"),
    4: QColor("#03A9F4"),
    5: QColor("#00BCD4"),
    6: QColor("#4CAF50"),
    7: QColor("#E91E63"),
    8: QColor("#FF9800"),
}

STATUS_READY = 0
STATUS_PLAYING = 1
STATUS_FAILED = 2
STATUS_SUCCESS = 3

STATUS_ICONS = {
    STATUS_READY: Paths.icon("plus.png"),
    STATUS_PLAYING: Paths.icon("smiley.png"),
    STATUS_FAILED: Paths.icon("cross.png"),
    STATUS_SUCCESS: Paths.icon("smiley-lol.png"),
}
```

游戏区域

Moonsweeper 的游戏区域是一个 NxN 的网格，其中包含固定数量的矿井。我们使用的网格尺寸和矿井数量来自 Windows 版本《扫雷》的默认值。所使用的值如表所示：

Table 14. Table Dimensions and mine counts

难度等级	尺寸	地雷数量
简单	8 x 8	10
中等	16 x 16	40
困难	24 x 24	99

我们将这些值存储为文件顶部定义的常量 `LEVELS`。由于所有游戏区域均为正方形，因此只需存储一次值（8、16或24）。

Listing 291. app/minesweeper.py

```
LEVELS = [("Easy", 8, 10), ("Medium", 16, 40), ("Hard", 24, 99)]
```

游戏网格可以以多种方式表示，例如使用一个二维的"列表的列表"来表示游戏位置的不同状态（地雷、已揭示、已标记）。

然而，在我们的实现中，我们将采用面向对象的方法，其中地图上的每个位置都包含自身相关的所有数据。进一步而言，我们可以让这些对象各自负责绘制自身。在Qt中，我们可以通过继承 `QWidget` 类，并实现自定义绘制函数来实现这一点。

在介绍这些自定义控件的外观之前，我们将先介绍它们的结构和行为。由于我们的瓷砖对象是 `QWidget` 的子类，因此我们可以像其他控件一样对它们进行布局。为此，我们需要设置一个 `QGridLayout`。

Listing 292. app/minesweeper.py

```
        self.grid = QGridLayout()
        self.grid.setSpacing(5)
        self.grid.setSizeConstraint(QLayout.SizeConstraint.SetFixedSize)
```

接下来，我们需要设置游戏区域，创建位置图块控件并将其添加到网格中。关卡的初始设置在自定义方法中定义，该方法从 `LEVELS` 读取数据，并将一些变量分配给窗口。窗口标题和地雷计数器更新后，网格的设置就开始了。

Listing 293. app/minesweeper.py

```
    def set_level(self, level):
        self.level_name, self.b_size, self.n_mines = LEVELS[level]

        self.setWindowTitle("Moonsweeper - %s" % (self.level_name))
        self.mines.setText("%03d" % self.n_mines)

        self.clear_map()
        self.init_map()
        self.reset_map()
```

接下来我们将介绍设置功能。

我们这里使用了一个自定义的 `Pos` 类，我们稍后会详细介绍它。目前，您只需要知道它包含地图中相关位置的所有相关信息——例如，是否为地雷、是否已被揭示、是否被标记以及附近地雷的数量。

每个 `Pos` 对象还有 3 个自定义信号 `clicked`、`revealed` 和 `expandable`，我们将其连接到自定义槽方法。最后，我们调用 `resize` 来调整窗口的大小，以适应新内容。请注意，这实际上只有在窗口缩小时才需要——窗口会自动扩大。

Listing 294. app/minesweeper.py

```
    def init_map(self):
        # 在地图上添加位置
        for x in range(0, self.b_size):
            for y in range(0, self.b_size):
                w = Pos(x, y)
                self.grid.addWidget(w, y, x)
                # 将信号连接到滑块扩展件.
                w.clicked.connect(self.trigger_start)
                w.revealed.connect(self.on_reveal)
                w.expandable.connect(self.expand_reveal)

            # 将调整大小操作放入事件队列，并在操作完成前将控制权交还给Qt.
            QTimer.singleShot(0, lambda: self.resize(1, 1)) #1
```

> 1. 单次定时器（`singleShot timer`）用于确保在Qt检测到新内容后再执行窗口大小调整。通过使用定时器，我们可以确保控制权在窗口大小调整发生前返回给Qt。

我们还需要实现 `init_map` 函数的逆函数，以从地图中移除瓷砖对象。在从较高层级移动到较低层级时，移除瓷砖将是必要的。在这里我们可以稍微聪明一点，只添加/移除那些达到正确尺寸所需的瓦片。但是，既然我们已经有了一个函数可以将所有瓦片添加到正确尺寸，我们可以稍微作弊一下。

> 挑战
>
> 更新此代码以添加/移除必要的瓷砖，以调整新关卡的尺寸。

请注意，我们使用 `self.grid.removeItem(c)` 将项目从网格中删除，并清除父级 `c.widget().setParent(None)`。第二步是必要的，因为添加项目时会将父级窗口指定为父级。仅删除它们会使它们漂浮在布局外的窗口中。

Listing 295. app/minesweeper.py

```
    def clear_map(self):
        # 从地图上移除所有位置，直至达到最大容量.
        for x in range(0, LEVELS[-1][1]): #1
            for y in range(0, LEVELS[-1][1]):
                c = self.grid.itemAtPosition(y, x)
                if c: #2
                    c.widget().close()
                    self.grid.removeItem(c)
```

> 1. 为了确保我们能够处理所有尺寸的地图，我们采用最高级别的尺寸。
> 2. 如果网格中该位置没有内容，我们可以跳过它。

现在我们已经将位置瓷砖对象的网格布局到位，可以开始创建游戏板的初始条件。这个过程相当复杂，因此被分解为多个函数。我们将其命名为 `_reset`（前缀下划线是表示私有函数的约定，不供外部使用）。主函数 `reset_map` 依次调用这些函数来进行设置。

流程如下

1. 移除所有地雷（并重置数据）并清空场地。
2. 在场地中添加新的地雷。
3. 计算每个位置相邻地雷的数量。
4. 添加起始标记（火箭）并触发初始探索。
5. 重置计时器。

Listing 296. app/minesweeper.py

```
    def reset_map(self):
        self._reset_position_data()
        self._reset_add_mines()
        self._reset_calculate_adjacency()
        self._reset_add_starting_marker()
        self.update_timer()
```

以下将依次详细描述步骤1至5，并附上各步骤的代码.

第一步是重置地图上每个位置的数据。我们遍历棋盘上的每个位置，在每个点上调用控件的 `.reset()` 方法。`.reset()` 方法的代码在我们的自定义 `Pos` 类中定义，我们稍后会详细探讨。目前只需知道它会清除地雷、旗帜，并将位置设置为未揭示状态即可。

Listing 297. app/minesweeper.py

```
    def _reset_position_data(self):
        # 清除所有地雷位置
        for x in range(0, self.b_size):
            for y in range(0, self.b_size):
                w = self.grid.itemAtPosition(y, x).widget()
                w.reset()
```

现在所有位置均为空，我们可以开始将地雷添加到地图了。地雷的最大数量 `n_mines` 由关卡设置定义，如前所述。

Listing 298. app/minesweeper.py

```
    def _reset_add_mines(self):
        # 添加地雷位置
        positions = []
        while len(positions) < self.n_mines:
            x, y = (
                random.randint(0, self.b_size - 1),
                random.randint(0, self.b_size - 1),
            )
            if (x, y) not in positions:
                w = self.grid.itemAtPosition(y, x).widget()
                w.is_mine = True
                positions.append((x, y))

            # 计算终局条件
            self.end_game_n = (self.b_size * self.b_size) - (
                self.n_mines + 1
            )
            return positions
```

地雷就位后，我们可以计算每个位置的“邻近”数——即该点周围 3x3 网格内地雷的数量。自定义函数 `get_surrounding` 仅返回给定x和y位置周围的这些位置。我们统计其中 `is_mine == True`（即为地雷）的数量并存储

> 预计算
>
> 通过这种方式预计算相邻计数，有助于简化后续的显示逻辑。

Listing 299. app/minesweeper.py

```
    def _reset_calculate_adjacency(self):
        def get_adjacency_n(x, y):
            positions = self.get_surrounding(x, y)
            return sum(1 for w in positions if w.is_mine)

        # 为位置添加相邻关系
        for x in range(0, self.b_size):
            for y in range(0, self.b_size):
                w = self.grid.itemAtPosition(y, x).widget()
                w.adjacent_n = get_adjacency_n(x, y)
```

起始标记用于确保第一步总是有效的。这通过对网格空间进行暴力搜索来实现，即随机尝试不同位置，直到找到一个不是地雷的位置。由于我们不知道需要尝试多少次，因此需要将此过程包裹在一个循环中。

一旦找到该位置，我们将它标记为起始位置，然后触发对所有周边位置的探索。我们退出循环，并重置就绪状态。

Listing 300. app/minesweeper.py

```
    def _reset_add_starting_marker(self):
        # 放置起始标记.

        # 设置初始状态（.click 功能需要此设置）
        self.update_status(STATUS_READY)

        while True:
            x, y = (
                random.randint(0, self.b_size - 1),
                random.randint(0, self.b_size - 1),
            )
            w = self.grid.itemAtPosition(y, x).widget()
            # 我们不想从地雷上开始.
            if not w.is_mine:
                w.is_start = True
                w.is_revealed = True
                w.update()

                # 如果这些位置也不是地雷，则显示所有相关位置.
                for w in self.get_surrounding(x, y):
                    if not w.is_mine:
                        w.click()

                break

        # 初始点击后将状态重置为就绪.
        self.update_status(STATUS_READY)
```

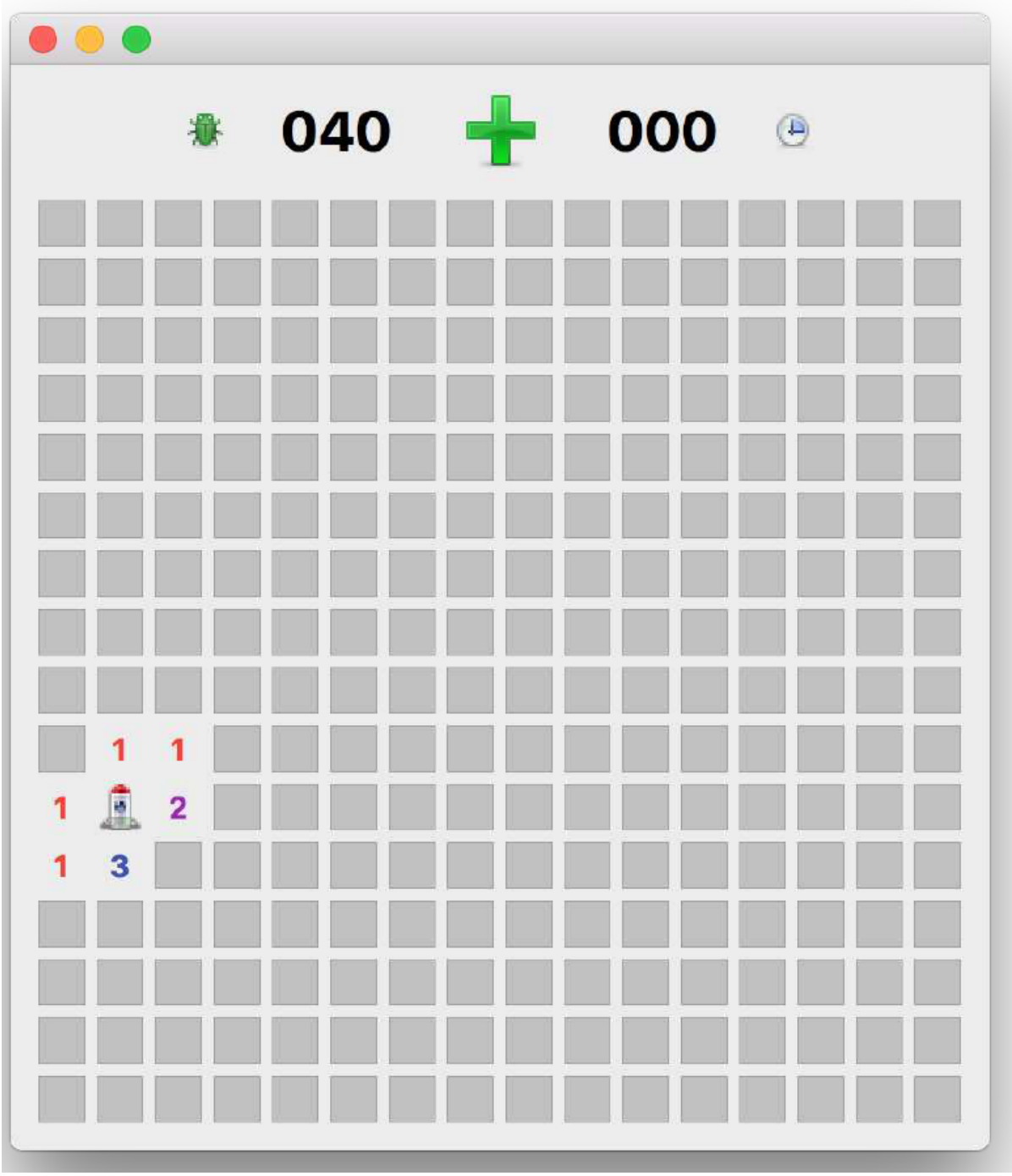


图272：火箭的初步探索

位置瓷砖

如前所述，我们设计了游戏结构，使每个方块的位置都保存自己的状态信息。这意味着 `Pos` 对象处于理想的位置，可以处理与自身状态相关的交互反应的游戏逻辑。换句话说，这就是奥妙所在之处。

由于 `Pos` 类相对复杂，这里将其分解为主要主题，并依次进行讨论。初始化设置 `__init__` 块非常简单，接受 `x` 和 `y` 坐标并将其存储在对象中。`Pos` 坐标一旦创建就不会改变。

完成设置后，我们调用 `.reset()` 函数将所有对象属性重置为默认值，即零值。这将标记该地雷为非起始位置、非地雷、未揭示且未标记。我们还重置了相邻计数。

Listing 301. app/minesweeper.py

```
class Pos(QWidget):

    expandable = pyqtSignal(int, int)
```

```
    revealed = pyqtSignal(object)
    clicked = pyqtSignal()

    def __init__(self, x, y):
        super().__init__()
        self.setFixedSize(QSize(20, 20))
        self.x = x
        self.y = y
        self.reset()

    def reset(self):
        self.is_start = False
        self.is_mine = False
        self.adjacent_n = 0
        self.is_revealed = False
        self.is_flagged = False

        self.update()
```

游戏玩法以鼠标与游戏场中的方块的交互为中心，因此检测鼠标点击并做出反应是至关重要的。在 Qt 中，我们通过检测 `mouseReleaseEvent` 来捕获鼠标点击。为了对我们的自定义 `Pos` 控件执行此操作，我们在类上定义了一个处理程序。该处理程序接收包含发生事件信息的 `QMouseEvent`。在此情况下，我们仅关注鼠标释放操作是来自左键还是右键。

对于左键点击，我们检查该方块是否已被标记或已揭开。如果它符合其中一种情况，我们将忽略该点击——使标记过的方块"安全"，无法被意外点击。如果方块未被标记，我们只需调用 `.click()` 方法（见后文）。

对于右键点击未显示的瓷砖，我们调用我们的 `.toggle_flag()` 方法来切换标志的状态。

Listing 302. app/minesweeper.py

```
    def mouseReleaseEvent(self, e):
        if (
            e.button() == Qt.MouseButton.RightButton
            and not self.is_revealed
        ):
            self.toggle_flag()

        elif e.button() == Qt.MouseButton.LeftButton:
            # 阻止点击标记的雷区.
            if not self.is_flagged and not self.is_revealed:
                self.click()
```

由鼠标释放事件处理程序调用的方法如下所示：

`.toggle_flag` 处理程序仅将 `.is_flagged` 设置为其自身的反转值（`True` 变为 `False`，`False` 变为 `True`），从而实现其开关状态的切换。注意，我们必须调用 `.update()` 方法以强制重新绘制，因为状态已发生变化。我们还发出自定义的 `.clicked` 信号，该信号用于启动计时器，因为放置标志也应算作启动，而不仅仅是显示一个方块。

`.click()` 方法处理鼠标左键点击，并触发显示正方形。如果该位置相邻的地雷数量为零，则触发 `.expandable` 信号，开始自动扩展已探索区域（见下文）。最后，我们再次发出 `.clicked` 信号，以指示游戏开始。

最后，`.reveal()` 方法检查该方块是否已经显示，如果没有，则将 `.is_revealed` 设置为 `True`。再次调用 `.update()` 以触发控件的重绘。

可选的 `.revealed` 信号仅用于游戏结束时全地图的揭示。由于每次揭示都会触发进一步的查找，以查找哪些瓷砖也可以揭示，因此揭示整个地图会产生大量冗余的回调。通过在此处抑制信号，我们可以避免这种情况。

Listing 303. app/minesweeper.py

```
    def toggle_flag(self):
        self.is_flagged = not self.is_flagged
        self.update()

        self.clicked.emit()

    def click(self):
        self.reveal()
        if self.adjacent_n == 0:
            self.expandable.emit(self.x, self.y)

        self.clicked.emit()

    def reveal(self, emit=True):
        if not self.is_revealed:
            self.is_revealed = True
            self.update()

        if emit:
                self.revealed.emit(self)
```

最后，我们为 `Pos` 控件定义了一个自定义的 `paintEvent` 方法，以处理当前位置状态的显示。如 前文所述，要在控件画布上执行自定义绘制，我们需要一个 `QPainter` 和 `event.rect()`，它提供了我们要绘制的边界——在本例中，是 `Pos` 控件的外边框。

已揭示的方块根据其类型（起始位置、炸弹或空格）以不同方式绘制。前两种类型分别由火箭和炸弹的图标表示。这些图标通过 `.drawPixmap` 方法绘制到方块的 `QRect` 中。注意：我们需要将 `QImage` 常量转换为像素图，通过将 `QPixmap` 传递给 `QImage` 的 `.toPixmap()` 方法实现。

> QPixmap 与 QImages
>
> 您可能会想：“既然我们正在使用它们，为什么不直接将这些存储为 `QPixmap` 对象？我们不能这样做并将其存储在常量中，因为在您的 `QApplication` 启动并运行之前，您无法创建 `QPixmap` 对象

对于空位（非火箭、非炸弹），我们可选地显示相邻数，如果该数大于零。为了在 QPainter 上绘制文本，我们使用 `.drawText()` 方法，传入 `QRect`、对齐标志以及要绘制的数字作为字符串。我们为每个数字定义了标准颜色（存储在 `NUM_COLORS` 中），以提高易用性。

对于未显示的瓷砖，我们绘制一个瓷砖，通过用浅灰色填充一个矩形并绘制一个1像素宽的深灰色边框。如果 `.is_flagged` 被设置，我们还会在瓷砖上方绘制一个旗帜图标使用 `drawPixmap` 和瓷砖的 `QRect`。

Listing 304. app/minesweeper.py

```
    def paintEvent(self, event):
        p = QPainter(self)
        p.setRenderHint(QPainter.RenderHint.Antialiasing)

        r = event.rect()

        if self.is_revealed:
            if self.is_start:
                p.drawPixmap(r, QPixmap(IMG_START))

            elif self.is_mine:
                p.drawPixmap(r, QPixmap(IMG_BOMB))

            elif self.adjacent_n > 0:
                pen = QPen(NUM_COLORS[self.adjacent_n])
                p.setPen(pen)
                f = p.font()
                f.setBold(True)
                p.setFont(f)
                p.drawText(
                    r,
                    Qt.AlignmentFlag.AlignHCenter
                    | Qt.AlignmentFlag.AlignVCenter,
                    str(self.adjacent_n),
                )

        else:
            p.fillRect(r, QBrush(Qt.GlobalColor.lightGray))
            pen = QPen(Qt.GlobalColor.gray)
            pen.setWidth(1)
            p.setPen(pen)
            p.drawRect(r)
            if self.is_flagged:
                p.drawPixmap(r, QPixmap(IMG_FLAG))
```

游戏过程

我们通常需要获取给定点周围的所有图块，因此我们为此定制了一个函数。它简单地遍历该点周围的 3x3 网格，并检查以确保我们不会超出网格边缘的范围（`0 ≥ x ≤ self.b_size`）。返回的列表包含每个周围位置的 `Pos` 控件。

Listing 305. app/minesweeper.py

```
    def get_surrounding(self, x, y):
        positions = [
        for xi in range(max(0, x - 1), min(x + 2, self.b_size)):
            for yi in range(max(0, y - 1), min(y + 2, self.b_size)):
                if not (xi == x and yi == y):
                    positions.append(
                        self.grid.itemAtPosition(yi, xi).widget()
                    )

        return positions
```

`expand_reveal` 方法在点击一个周围没有地雷的方块时触发。在这种情况下，我们希望将点击区域扩展到任何周围也没有地雷的区域，并揭示扩展区域边界周围的任何非地雷方块。

这可以通过查看点击方块周围的所有方块来实现，并对任何 `.n_adjacent == 0` 的方块触发 `.click()`。正常的游戏逻辑会接管并自动扩展区域。然而，这有些低效，会产生大量冗余信号（每个方块会为每个周围方块触发多达 9 个信号）。因此，我们需要一种更高效的方法来处理这些信号。

相反，我们使用一个独立的方法来确定要显示的区域，然后触发显示（使用 `.reveal()` 来避免 `.clicked` 信号）。

我们首先创建一个列表 `to_expand`，其中包含下一次迭代要检查的位置；一个列表 `to_reveal`，其中包含要显示的瓷砖控件；以及一个标志 `any_added`，用于确定何时退出循环。当 `to_reveal` 中没有添加新控件时，循环就会停止。

在循环内部，我们将 `any_added` 重置为 `False`，并清空 `to_expand` 列表，同时在 `l` 中保留一个临时存储空间用于迭代。

对于每个 `x` 和 `y` 位置，我们获取周围的 8 个控件。如果这些控件中任何一个不是地雷，并且尚未添加到 `to_reveal` 列表中，则将其添加到该列表中。这样可以确保扩展区域的边缘全部被揭示。如果该位置没有相邻的地雷，我们将坐标附加到 `to_expand`，以便在下次迭代时进行检查。

通过将任何非地雷方块添加到 `to_reveal` 中，并且仅展开那些尚未在 `to_reveal` 中存在的方块，我们可以确保不会访问同一方块超过一次。

Listing 306. app/minesweeper.py

```
    def expand_reveal(self, x, y):
        """
        从初始点开始向外迭代，将新位置添加到队列中。这使我们能够一次性展开所有内容，而不是依赖多个回调.
        """
        to_expand = [(x, y)]
        to_reveal = []
        any_added = True

        while any_added:
            any_added = False
            to_expand, l = [], to_expand

            for x, y in l:
                positions = self.get_surrounding(x, y)
                for w in positions:
                    if not w.is_mine and w not in to_reveal:
                        to_reveal.append(w)
```

```
                        if w.adjacent_n == 0:
                            to_expand.append((w.x, w.y))
                            any_added = True

        # 遍历并显示我们找到的所有位置.
        for w in to_reveal:
            w.reveal()
```

终局

终局状态在点击标题后进行的显示过程中被检测到。有两种可能的结果——

1. 地砖是地雷，游戏结束。
2. 地砖不是地雷，减少 `self.end_game_n` 。

此过程持续进行，直到 `self.end_game_n` 达到零，这将触发游戏结束流程，通过调用 `game_over` 或 `game_won` 函数来实现。成功或失败的触发条件是揭示地图并设置相关状态，在两种情况下均需执行此操作。

Listing 307. app/minesweeper.py

```
    def on_reveal(self, w):
        if w.is_mine:
            self.game_over()

        else:
            self.end_game_n -= 1 # 减少剩余空位

            if self.end_game_n == 0:
                self.game_won()

    def game_over(self):
        self.reveal_map()
        self.update_status(STATUS_FAILED)

    def game_won(self):
        self.reveal_map()
        self.update_status(STATUS_SUCCESS)
```

图273：哦不。被B'ug吃掉了。

状态

Moonsweeper 的用户界面非常简单：一个显示屏显示地雷的数量，一个显示屏显示已过去的时间，以及一个用于启动/重新启动游戏的按钮。

这两个标签均被定义为 `QLabel` 对象，且使用相同的 `QFont` 字体大小和颜色。这些标签在 `QMainWindow` 对象上进行定义，以便我们可以在后续时间访问并更新它们。另外，还定义了两个额外的图标（时钟和地雷）作为 `QLabel` 对象。

该按钮是一个带有定义图标的 `QPushButton`，该图标在 `set_status` 中根据状态变化进行更新。`.pressed` 信号连接到自定义槽方法 `button_pressed`，该方法根据游戏状态以不同的方式处理信号。

Listing 308. app/minesweeper.py

```
        self.mines = QLabel()
```

```
        self.mines.setAlignment(
            Qt.AlignmentFlag.AlignHCenter
            | Qt.AlignmentFlag.AlignVCenter
        )

        self.clock = QLabel()
        self.clock.setAlignment(
            Qt.AlignmentFlag.AlignHCenter
            | Qt.AlignmentFlag.AlignVCenter
        )

        f = self.mines.font()
        f.setPointSize(24)
        f.setWeight(QFont.Weight.Bold)
        self.mines.setFont(f)
        self.clock.setFont(f)

        self.clock.setText("000")

        self.button = QPushButton()
        self.button.setFixedSize(QSize(32, 32))
        self.button.setIconSize(QSize(32, 32))
        self.button.setIcon(QIcon(Paths.icon("smiley.png")))
        self.button.setFlat(True)

        self.button.pressed.connect(self.button_pressed)

        self.statusBar()
        l = QLabel()
        l.setPixmap(QPixmap.fromImage(IMG_BOMB))
        l.setAlignment(
            Qt.AlignmentFlag.AlignRight | Qt.AlignmentFlag
            .AlignVCenter
        )
        hb.addWidget(l)

        hb.addWidget(self.mines)
        hb.addWidget(self.button)
        hb.addWidget(self.clock)

        l = QLabel()
        l.setPixmap(QPixmap.fromImage(IMG_CLOCK))
        l.setAlignment(
            Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignVCenter
        )
        hb.addWidget(l)

        vb = QVBoxLayout()
        vb.setSizeConstraint(QLayout.SizeConstraint.SetFixedSize)
        vb.addLayout(hb)
```

如果游戏当前正在进行中，且 `self.status == STATUS_PLAYING`，则按下按钮会被解释为“我放弃”，并触发游戏结束状态。

如果当前游戏状态为获胜（`self.status == STATUS_SUCCESS`）或失败（`self.status ==STATUS_FAILED`），则按下按钮被视为“重新尝试”，游戏地图将被重置。

Listing 309. app/minesweeper.py

```
    def button_pressed(self):
        if self.status == STATUS_PLAYING:
            self.game_over()

        elif (
            self.status == STATUS_FAILED
            or self.status == STATUS_SUCCESS
        ):
            self.reset_map()
```

菜单

Moonsweeper 只有一个菜单，用于控制游戏。我们通过调用 `QMainWindow.menuBar()` 的 `.addMenu()` 方法来创建 `QMenu`，与通常操作一致。

第一个菜单项是一个标准的 `QAction`，用于“新游戏”，其 `.triggered` 属性与 `.reset_map` 函数关联，该函数负责整个地图的初始化过程。对于新游戏，我们保留现有的棋盘大小和布局，因此无需重新初始化地图。

此外，我们添加了一个子菜单 “Levels”，其中包含 `LEVELS` 中定义的每个级别的 `QAction`。级别名称取自相同的常量，自定义状态消息由存储的尺寸构建。我们将动作 `.triggered` 信号连接到 `.set_level`，使用 `lambda` 方法丢弃默认信号数据，并传递级别编号。

Listing 310. app/minesweeper.py

```
        game_menu = self.menuBar().addMenu("&Game")

        new_game_action = QAction("New game", self)
        new_game_action.setStatusTip(
            "Start a new game (your current game will be lost)"
        )
        new_game_action.triggered.connect(self.reset_map)
        game_menu.addAction(new_game_action)

        levels = game_menu.addMenu("Levels")
        for n, level in enumerate(LEVELS):
            level_action = QAction(level[0], self)
            level_action.setStatusTip(
                "{1}x{1} grid, with {2} mines".format(*level)
            )
            level_action.triggered.connect(
                lambda checked=None, n=n: self.set_level(n)
            )
            levels.addAction(level_action)
```

继续深入

请您查看我们尚未介绍的源代码的其余部分。

> 挑战
>
> 您还可以尝试进行以下修改——
>
> - 尝试修改图形，制作你自己的主题版扫雷游戏。
> - 添加对非正方形游戏区域的支持。矩形？不妨试试圆形！
> - 将计时器改为倒计时——在月球上与时间赛跑！
> - 添加道具：方块提供奖励、额外时间、无敌状态。

附录A：安装 PyQt6

在开始编码之前，您需要先在系统上安装好 PyQt6。如果尚未安装 PyQt6，以下各节将指导您在 Windows、macOS 和 Linux 系统上完成安装。

> 请注意，以下说明**仅**适用于安装**GPL 许可**版本的 PyQt。如果您需要在非 GPL 项目中使用 PyQt，您需要从 Riverbank Computing 购买替代许可以发布您的软件。

在 Windows 系统上的安装

PyQt6 对于 Windows 系统的安装方式与其他应用程序或库类似。自 Qt 5.6 起，可通过 Python 包存档（PyPi）安装 PyQt6。要从 Python 3 安装 PyQt6，只需运行以下命令：

```
pip3 install pyqt6
```

安装完成后，您应该能够运行 Python 并导入 PyQt6。

请注意，如果您想使用 Qt Designer 或 Qt Creator，您需要从 Qt 下载网站 下载这些工具。

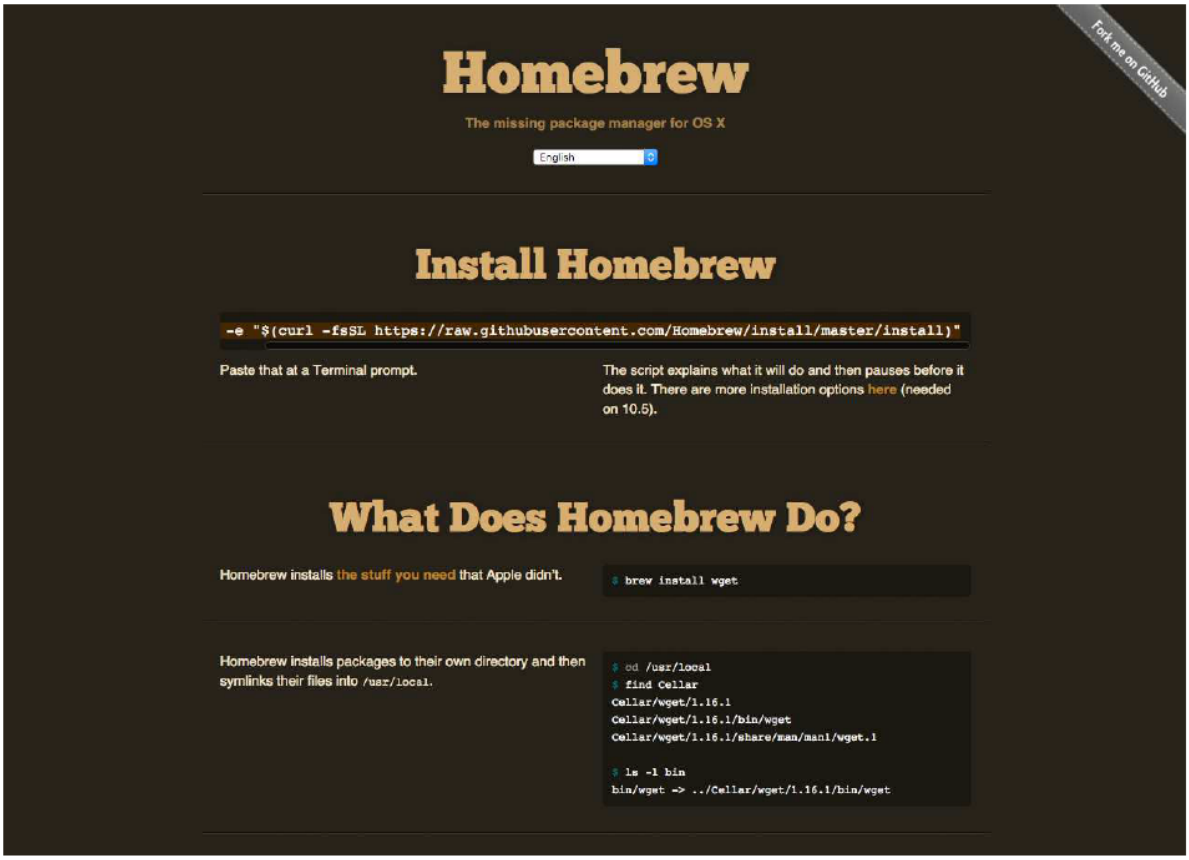
在 macOS 系统上的安装

如果您已经在 macOS 上安装了可用的 Python 3，您可以继续安装 PyQt6，就像安装其他 Python 包一样，使用以下命令：

```
pip3 install pyqt6
```

如果您尚未安装 Python 3，则需要先进行安装。您可以从 Python 官方网站 下载适用于 macOS 的 Python 3 安装程序。安装完成后，您应能够使用上述 `pip3 install` 命令安装PyQt6。

另一个选择是使用Homebrew。Homebrew是macOS上用于管理命令行软件的包管理器。Homebrew的仓库中同时提供了Python 3和PyQt6。



> 图274：Homebrew — macOS 系统中缺失的包管理器

要安装 Homebrew，请在命令行中执行以下操作：

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

> 您也可以从Homebrew的官网复制并粘贴此内容。

安装 Homebrew 后，您可以使用以下命令安装 Python：

```
brew install python3
```

安装 Python 后，您可以像往常一样安装 PyQt6，使用 `pip3 install pyqt6`，或者选择使用 Homebrew 进行安装，使用：

```
brew install pyqt6
```

在 Linux 系统上的安装

在 Linux 上安装 PyQt6 的最简单方法是使用 Python 的 pip 包管理工具，与安装其他包的方式相同。对于 Python 3 安装，我们通常称为 pip3。

```
pip3 install pyqt6
```

安装完成后，您应该能够运行 python3（或 python，具体取决于您的系统）并导入 PyQt6。

附录B：将 C++ 示例翻译为 Python

在使用PyQt6编写应用程序时，我们实际上是在使用Qt编写应用程序。

PyQt6 作为 Qt 库的包装器，将 Python 方法调用转换为 C++，处理类型转换，并透明地创建 Python 对象来代表应用程序中的 Qt 对象。所有这些巧妙的设计使您可以在编写大部分 Python 代码时使用 Qt（如果忽略驼峰式命名法的话）。

虽然网上有很多PyQt6示例代码，但Qt C++示例代码要多得多。核心文档是为C++编写的。该库是用C++编写的。这意味着，有时当您想要了解如何实现某项功能时，您找到的唯一资源可能是C++教程或一些C++代码。

您能使用它吗？当然可以！如果您没有C++（或类似C的语言）的经验，那么代码看起来可能像天书一样。但在您熟悉Python之前，Python可能也看起来有点像天书。你您需要会写C++就能阅读它。理解和解读比编写更容易。

只需稍作努力，您就能将任何 C++ 示例代码转换为功能完整的 Python 的 PyQt6 代码。在本章中，我们将选取一段 Qt5 代码，并逐步将其转换为可正常运行的 Python 代码。

示例代码

我们先从以下代码块开始，创建一个简单的窗口，其中包含一个 `QPushButton` 和一个 `QLineEdit`。按下按钮将清除行编辑内容。这看起来非常令人兴奋，但其中包含将 Qt 示例转换为 PyQt6 的几个关键部分，即控件、布局和信号。

```
#include <QtWidgets>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    QLineEdit *lineEdit = new QLineEdit();
    QPushButton *button = new QPushButton("Clear");
    QHBoxLayout *layout = new QHBoxLayout();
    layout->addWidget(lineEdit);
    layout->addWidget(button);

    QObject::connect(&button, &QPushButton::pressed,
                     &lineEdit, &QLineEdit::clear);
    window.setLayout(layout);
    window.setWindowTitle("Why?");
    window.show();
    return app.exec();
}
```

> 请记住，没有父控件的 Qt 控件总是单独的窗口。这里，我们创建了一个作为 `QWidget` 的单个窗口。

下面我们将逐步讲解如何将这段代码转换为Python代码。

导入语句

在 C++ 中，导入（import）被称为包含（include）。它们位于文件的顶部，与 Python 类似（尽管只是出于惯例），并且看起来像这样——

```
#include <QtWidgets>
```

在 C 类语言中，`#` 表示 `include` 是预处理指令，而不是注释。`<>` 之间的值是要导入的模块的名称。请注意，与 Python 不同，导入模块会将该模块的所有内容都放在全局命名空间中。这相当于在 Python 中执行以下操作：

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

像这样的全局导入在 Python 中通常是不被推荐的，您应该改用以下方式之一：

1. 仅导入所需的对象，或
2. 导入模块本身并通过它引用其子对象

```
from PyQt6.QtWidgets import QApplication, QWidget, QLineEdit, QPushButton,
QHBoxLayout
```

或者，换一种说法……

```
from PyQt6 import QtWidgets
```

……然后引用为 `QtWidgets.QApplication()`。您在自己的代码中选择哪种样式完全取决于您的喜好，但在本例中，我们将遵循第一种样式。将此应用于代码后，到目前为止，我们得到了以下结果。

```
from PyQt6.QtWidgets import (
    QApplication, QWidget, QLineEdit, QPushButton, QHBoxLayout
)

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    QLineEdit *lineEdit = new QLineEdit();
    QPushButton *button = new QPushButton("Clear");
    QHBoxLayout *layout = new QHBoxLayout();
    layout->addWidget(lineEdit);
```

```
    layout->addWidget(button);
    QObject::connect(&button, &QPushButton::pressed,
                     &lineEdit, &QLineEdit::clear);
    window.setLayout(layout);
    window.setWindowTitle("Why?");
    window.show();
    return app.exec();
}
```

> 由于我们是通过迭代方式进行修改，因此代码在最终完成前都不会正常工作。

int main(int argc, char *argv[])

每个 C++ 程序都需要一个 `main(){}` 代码块，其中包含应用程序执行时首先运行的代码。在 Python 中，模块顶层的任何代码（即未嵌套在函数、类或方法内部的代码）将在脚本执行时被运行。

```
from PyQt6.QtWidgets import (
    QApplication, QWidget, QLineEdit, QPushButton, QHBoxLayout
)

QApplication app(argc, argv);
QWidget window;
QLineEdit *lineEdit = new QLineEdit();
QPushButton *button = new QPushButton("Clear");
QHBoxLayout *layout = new QHBoxLayout();
layout->addWidget(lineEdit);
layout->addWidget(button);
QObject::connect(&button, &QPushButton::pressed,
                 &lineEdit, &QLineEdit::clear);
window.setLayout(layout);
window.setWindowTitle("Why?");
window.show();
app.exec();
```

您可能在 Python 应用程序代码中见过以下代码块，它也被称为 `__main__` 块。

```
if __name__ == '__main__':
# ...your code here...
```

然而，这种方式的工作原理略有不同。虽然当脚本被执行时，这个块会被执行，但任何未缩进的代码也会被执行。这个块的实际目的是防止在模块被导入时执行这段代码，而不是作为脚本执行。

您可以将代码嵌套在这个代码块中，尽管除非您的文件将被作为模块导入，否则这并非严格必要。

C++ 类型

Python 是一种动态类型语言，这意味着您可以在变量定义后更改其类型。例如，以下代码是完全有效的 Python。

```
a = 1
a = 'my string'
a = [1,2,3]
```

许多其他语言，包括C++在内，都是静态类型的，这意味着一旦定义了变量的类型，就不能再更改它。例如，以下代码绝对不是有效的C++代码。

```
int a = 1;
a = 'my string';
```

上述内容突显了静态类型语言的直接后果：在创建变量时定义其类型。

在 C++ 中，这是通过在定义变量时在变量声明行上显式提供类型装饰器来实现的，位于 `int` 之上。

在类似以下的语句中，第一个名称是正在被创建的类型（类）的名称，而语句的其余部分用于定义该类型。

```
QApplication app(argc, argv);
QWidget window;

QLineEdit *lineEdit = new QLineEdit();
QPushButton *button = new QPushButton("Clear");
QHBoxLayout *layout = new QHBoxLayout();
```

在 Python 中，我们不需要这些类型定义，因此可以直接删除它们。

```
lineEdit = new QLineEdit();
button = new QPushButton("Clear");
layout = new QHBoxLayout();
```

对于应用程序和窗口，原理完全相同。然而，如果您不熟悉C++，可能不会立即意识到这些代码行正在创建一个变量。

在 C++ 中，使用 `new` 创建对象与不使用 `new` 创建对象之间存在差异，但在 Python 中，您无需关心这一点，可以将它们视为等价的。

```
QWidget *window = new QWidget();
QWidget window;
QApplication *app = new QApplication(argc, argv);
QApplication app;
```

要转换为 Python，请从左侧获取类名（例如 `QApplication`），并将其放置在开括号和闭括号 `()` 之前，如果它们不存在，则添加它们。然后将变量名移到左侧，并添加一个 `=` 。对于 `window`，我们应该这样——

```
window = QWidget()
```

在 Python 中，`QApplication` 仅接受一个参数，即来自 `sys.argv`（等同于 `argv`）的参数列表。这为我们提供了以下代码

```
import sys
app = QApplication(sys.argv);
```

到目前为止，我们的完整代码块看起来如下所示。

```
from PyQt6.QtWidgets import (
    QApplication, QWidget, QLineEdit, QPushButton, QHBoxLayout
)

import sys
app = QApplication(argc, argv);
window = QWidget()
lineEdit = QLineEdit();
button = QPushButton("Clear");
layout = QHBoxLayout();
layout->addWidget(lineEdit);
layout->addWidget(button);
QObject::connect(&button, &QPushButton::pressed,
                 &lineEdit, &QLineEdit::clear);
window.setLayout(layout);
window.setWindowTitle("Why?");
window.show();
app.exec();
```

信号

信号是使示例正常运行的关键，但遗憾的是，C++ 语法对于 Qt 信号来说有些复杂。我们正在使用的示例信号如下所示。

```
QObject::connect(&button, &QPushButton::pressed,
                 &lineEdit, &QLineEdit::clear);
```

如果您不熟悉 C++，这段内容可能很难理解。但如果我们移除所有语法，就会清晰得多。

```
connect(button, QPushButton.pressed, lineEdit, QLineEdit.clear)
// or...
connect(<from object>, <from signal>, <to object>, <to slot>>)
```

从左到右依次为：我们连接的对象、我们连接的信号、我们连接的对象，最后是我们连接的对象上的槽（或函数）。这相当于在 PyQt6 中编写以下代码：

```
button.pressed.connect(lineedit.clear)
```

进行该更改后，我们在进行中的代码中将获得以下内容：

```
from PyQt6.QtWidgets import (
    QApplication, QWidget, QLineEdit, QPushButton, QHBoxLayout
)
app = QApplication(sys.argv)
```

```
window = QWidget()
lineEdit = QLineEdit()
button = QPushButton("Clear")
layout = QHBoxLayout()
layout->addWidget(lineEdit);
layout->addWidget(button);

button.pressed.connect(lineEdit.clear)

window.setLayout(layout);
window.setWindowTitle("Why?");
window.show();
app.exec();
```

语法

到目前为止，我们已经处理了所有特别棘手的部分，因此可以进行最后的语法校正。这些操作只需简单的搜索替换即可完成。

首先搜索所有 `->` 或 `::` 的实例，并替换为 `.` 。您会发现 C++ 代码在某些地方也使用了 `.` ——这与这些变量更简化的创建方式有关（`new` 与否）。同样，您可以在这里忽略这一点，并简单地使用 `.` 代替。

```
layout.addWidget(lineEdit);
layout.addWidget(button);
```

最后，删除所有行尾的分号（`;`）标记。

```
layout.addWidget(lineEdit)
layout.addWidget(button)
```

> 从技术上讲，您不需要这样做，因为 `;` 是 Python 中有效的行结束符。只是没有必要。

以下代码现在在Python上运行正常。

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLineEdit,
    QPushButton,
    QWidget,
)

app = QApplication(sys.argv)
```

```
window = QWidget()
lineEdit = QLineEdit()
button = QPushButton("Clear")
layout = QHBoxLayout()
layout.addWidget(lineEdit)
layout.addWidget(button)

button.pressed.connect(lineEdit.clear)

window.setLayout(layout)
window.setWindowTitle("Why?")
window.show()
app.exec()
```

在 Python 代码中，通常（尽管不是必需的）会继承窗口类，以便初始化代码可以包含在 `__init__` 块中。下面的代码已重新组织为这种结构，将除创建窗口对象（现在为 `MyWindow`）和 `app`，以及 `app.exec()` 调用之外的所有内容移至 `__init__` 块中。

```
import sys

from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QLineEdit,
    QPushButton,
    QWidget,
)


class MyWindow(QWidget):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        lineEdit = QLineEdit()
        button = QPushButton("Clear")
        layout = QHBoxLayout()
        layout.addWidget(lineEdit)
        layout.addWidget(button)

        button.pressed.connect(lineEdit.clear)

        self.setLayout(layout)
        self.setWindowTitle("Why?")
        self.show()


app = QApplication(sys.argv)
window = MyWindow()
app.exec()
```

将该流程应用于您自己的代码

这是一个非常简单的示例，然而如果您遵循相同的流程，您可以可靠地将任何C++ Qt代码转换为其Python等价代码。在转换您自己的代码示例时，尽量遵循这种分步方法，以最大限度地减少遗漏内容或意外破坏代码的风险。如果您最终得到一段能够运行的Python代码，但它与原代码存在细微差异，那么调试起来可能会很困难。

> 如果您有需要帮助翻译的代码示例，您可以随时与我联系，我会尽力为您提供帮助。

附录C：PyQt6 和 PySide6 两者有何不同?

如果您开始使用Qt6构建Python应用程序，您应该很快就会发现，实际上您可以使用两个包来实现这一点——PyQt6和PySide6。

在本章中，我将详细解释为什么会出现这种情况，以及您是否需要关心这一点（剧透：您真的不需要关心），同时也会介绍（少数）差异以及如何绕过这些差异。到本章结束时，您应该能够自如地复用PyQt6和PySide6教程中的代码示例来构建您的应用程序，无论您自己使用的是哪个包。

背景

为什么有两个库?

PyQt 由 Riverbank Computing Ltd. 的 Phil Thompson 开发，并已存在了很长时间，支持回溯至 2.x 版本的 Qt。在2009 年，当时拥有 Qt 工具包的诺基亚公司希望将 Qt 的 Python 绑定以更宽松的 LGPL 许可证形式提供。由于无法与 Riverbank 达成协议（Riverbank 因此会损失收入，这可以理解），他们随后发布了自己开发的绑定库 _PySide。

> 它被称为PySide，因为“side”在芬兰语中意为“粘合剂”。

这两个接口基本上是等效的，但随着时间的推移，PySide 的开发进度逐渐落后于 PyQt。这一差距在 Qt 5 发布后尤为明显——PyQt 的 Qt5 版本（PyQt5）自 2016 年中旬起便已可用，而 PySide2 的首个稳定版本则在两年后才发布。考虑到这一点，许多 Python 上的 Qt5 示例使用 PyQt5 就不足为奇了——仅仅是因为它已经可用。

然而，Qt 项目最近已将 PySide 作为官方的 Qt for Python 发布版本，这应能确保其未来的发展前景。当 Qt6 发布时，两个 Python 绑定版本均在短时间内同步推出。

	PyQt6	PySide6
首个稳定版本	2021.1	2020.12

	PyQt6	PySide6
开发者	Riverbank Computing Ltd.	Qt
许可证	GPL 许可证或商业许可证	LGPL
平台	Python 3	Python 3

您应该使用哪一个？坦白说，其实并没有太大区别。

这两个包都封装了同一个库——Qt6——因此它们的API有99.9%是相同的（见下文的少量差异）。您使用其中一个库学到的任何内容都可轻松应用于使用另一个库的项目。此外，无论您选择使用哪个库，都值得熟悉另一个库，以便您能够充分利用所有可用的在线资源——例如，使用 PyQt6 教程来构建您的 PySide6 应用程序，反之亦然。

在本短章中，我将简要介绍这两个包之间的几个显著差异，并解释如何编写能够与两者无缝兼容的代码。阅读本章后，您应该能够将任何在线的 PyQt6 示例转换为与 PySide6 兼容的代码。

许可证

两个版本之间的主要区别在于许可协议——PyQt6可通过GPL或商业许可协议获取，而PySide6则采用LGPL许可协议。

如果您计划将您的软件本身在GPL许可证下发布，或者您正在开发不会分发的软件，PyQt6的GPL要求很可能不会成为问题。然而，如果您希望分发您的软件但不分享源代码，您需要从Riverbank购买PyQt6的商业许可证或使用PySide6。

> Qt本身可通过Qt商业许可证、GPL 2.0、GPL3.0和LGPL 3.0许可证获取。

命名空间和枚举

PyQt6 引入的一项重大变更之一是需要使用完全限定名来引用枚举和标志。此前，在 PyQt5 和 PySide2 中，您可以使用快捷方式——例如 `Qt.DecorationRole`、`Qt.AlignLeft`。在 PyQt6 中，这些现在分别是 `Qt.ItemDataRole.DisplayRole` 和 `Qt.Alignment.AlignLeft`。这一变更影响了 Qt 中所有枚举类型和标志组。在 PySide6 中，长名称和短名称均继续支持。

UI 文件

这两个库之间的另一个主要区别在于它们对从 Qt Creator/Designer 导出的 `.ui` 文件的加载方式。PyQt6 提供了 uic 子模块，可用于直接加载 UI 文件以生成对象。这看起来非常符合 Python 的风格（如果忽略驼峰式命名法的话）。

```
import sys
from PyQt6 import QtWidgets, uic

app = QtWidgets.QApplication(sys.argv)

window = uic.loadUi("mainwindow.ui")
window.show()
app.exec()
```

使用 PySide6 时，代码会多出一行，因为您需要先创建一个 `QUILoader` 对象。不幸的是，这两个接口的 API 也有所不同（`.load` 与 `.loadUI`）。

```
import sys
from PySide6 import QtCore, QtGui, QtWidgets
from PySide6.QtUiTools import QUiLoader

loader = QUiLoader()

app = QtWidgets.QApplication(sys.argv)
window = loader.load("mainwindow.ui", None)
window.show()
app.exec()
```

要在 PyQt6 中将 UI 加载到现有对象上，例如在您的 `QMainWindow.init` 中，您可以调用 `uic.loadUI`，并将 `self`（现有控件）作为第二个参数传递。

```
import sys
from PyQt6 import QtCore, QtGui, QtWidgets
from PyQt6 import uic


class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        uic.loadUi("mainwindow.ui", self)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

PySide6 加载器不支持此功能——`.load` 的第二个参数是您正在创建的控件的父控件。这会阻止您将自定义代码添加到控件的 `__init__` 块中，但您可以使用一个单独的函数来解决这个问题。

```
import sys
from PySide6 import QtWidgets
from PySide6.QtUiTools import QUiLoader

loader = QUiLoader()

def mainwindow_setup(w):
```

```
    w.setWindowTitle("MainWindow Title")

app = QtWidgets.QApplication(sys.argv)

window = loader.load("mainwindow.ui", None)
mainwindow_setup(window)
window.show()
app.exec()
```

将UI文件转换为Python

两者都提供了相同的脚本，用于从 Qt Designer `.ui` 文件生成可导入 Python 的模块。对于 PyQt6，该脚本名为 `pyuic5`

```
pyuic6 mainwindow.ui -o MainWindow.py
```

然后，您可以导入 `UI_MainWindow` 对象，并通过多继承方式从您使用的基类（例如 `QMainWindow`）派生子类，然后调用 `self.setupUI(self)` 来设置用户界面。

```
import sys
from PyQt6 import QtWidgets
from MainWindow import Ui_MainWindow


class MainWindow(QtWidgets.QMainWindow, Ui_MainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.setupUi(self)


app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

对于 PySide6，其名称为 `pyside6-uic`

```
pyside6-uic mainwindow.ui -o MainWindow.py
```

后续设置完全相同。

```
import sys
from PyQt6 import QtWidgets
from MainWindow import Ui_MainWindow


class MainWindow(QtWidgets.QMainWindow, Ui_MainWindow):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.setupUi(self)
```

```
app = QtWidgets.QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

> 有关在 PyQt6 或 PySide6 中使用 Qt Designer 的更多信息，请参阅 Qt Creator 章节。

exec() 还是 exec_()

`.exec()` 方法用于 Qt 中启动 `QApplication` 或对话框的事件循环。在 Python 2.7 中，`exec` 是关键字，因此无法用于变量、函数或方法名称。PyQt4 和 PySide 中采用的解决方案是将 `.exec` 的使用改为 `.exec_()` 以避免此冲突。

Python 3 移除了 `exec` 关键字，释放了该名称以便重新使用。因此，从 Qt6 开始，所有 `.exec()` 调用都与 Qt 本身中的命名方式一致。然而，PySide6 仍然支持 `.exec_()`，因此如果您在某些代码中看到这个，不要感到惊讶。

槽与信号

在两个库中，定义自定义槽和信号时使用的语法略有不同。PySide6 在 `Signal` 和 `Slot` 名称下提供了此接口，而 PyQt6 则分别在 `pyqtSignal` 和 `pyqtSlot` 名称下提供了此接口。它们在定义槽和信号时的行为完全相同。

以下 PyQt6 和 PySide6 示例是相同的——

```
my_custom_signal = pyqtSignal() # PyQt6
my_custom_signal = Signal() # PySide6

my_other_signal = pyqtSignal(int) # PyQt6
my_other_signal = Signal(int) # PySide6
```

或者对于一个槽——

```
@pyqtslot
def my_custom_slot():
    pass

@Slot
def my_custom_slot():
    pass
```

如果您想确保 PyQt6 和 PySide6 之间的一致性，可以使用以下导入模式，以便 PyQt6 也能使用 `Signal` 和 `@Slot` 样式。

```
from PyQt6.QtCore import pyqtSignal as Signal, pyqtSlot as Slot
```

> 当然，您也可以从 `PySide6.QtCore` 导入 `Signal` 作为 `pyqtSignal`，导入 `Slot` 作为 `pyqtSlot`，尽管这会有些令人困惑。

QMouseEvent

在 PyQt6 中，`QMouseEvent` 对象不再具有用于访问事件位置的 `.pos()`、`.x()` 或 `.y()` 缩写属性方法。您必须使用 `.position()` 属性来获取 `QPoint` 对象，并访问其上的 `.x()` 或 `.y()` 方法。`.position()` 方法在 PySide6 中也可用。

PySide6 中存在但 PyQt6 中不存在的功能

从 Qt 6 开始，PySide 支持两个 Python 功能标志，以帮助代码更符合 Python 风格，使用蛇形变量名（`snake_case`），并能够直接分配和访问属性，而不是使用 getter/setter 函数。下面的示例显示了这些更改对代码的影响：

Listing 311. Standard PySide6 code

```
table = QTableWidget()
table.setColumnCount(2)

button = QPushButton("Add")
button.setEnabled(False)

layout = QVBoxLayout()
layout.addWidget(table)
layout.addWidget(button)
```

相同的代码，但启用了 `snake_case` 和 `true_property`。

Listing 312. PySide6 code with Snake case & properties.

```
from __feature__ import snake_case, true_property

table = QTableWidget()
table.column_count = 2

button = QPushButton("Add")
button.enabled = False

layout = QVBoxLayout()
layout.add_widget(table)
layout.add_widget(button)
```

这些功能标志对代码可读性有所提升，然而由于它们在 PyQt6 中不被支持，这使得在不同库之间进行迁移变得更加困难。

在两种库中都支持的特性

> 如果您正在开发独立的应用程序，则无需担心这个问题。只需使用您偏好的任何 API 即可。

如果您正在编写一个与 PyQt6 和 PySide6 兼容的库、控件或其他工具，只需添加两组导入即可轻松实现。

```
import sys

if 'PyQt6' in sys.modules:
    # PyQt6
    from PyQt6 import QtGui, QtWidgets, QtCore
    from PyQt6.QtCore import pyqtSignal as Signal, pyqtSlot as Slot

else:
    # PySide6
    from PySide6 import QtGui, QtWidgets, QtCore
    from PySide6.QtCore import Signal, Slot
```

这是我们的自定义控件库中使用的方法，我们通过导入一个库来支持 PyQt6 和 PySide6。唯一需要注意的是，在导入该库时，必须确保先导入 PyQt6（如上行或更早行），以确保它位于 `sys.modules` 中。

为了弥补PyQt6中缺少简写枚举和标志，您可以自行生成这些内容。例如，以下代码将为每个枚举对象的元素复制引用，直至其父对象，使其可像在PyQt5、PySide2和PySide6中一样访问。该代码仅需在PyQt6环境下运行。

```
enums = [
    (QtCore.Qt, 'Alignment'),
    (QtCore.Qt, 'ApplicationAttribute'),
    (QtCore.Qt, 'CheckState'),
    (QtCore.Qt, 'CursorShape'),
    (QtWidgets.QSizePolicy, 'Policy'),
]

# 使用长名称进行查找（例如 QtCore.Qt.CheckState.Checked，用于 PyQt6）并以短名称存储（例如 QtCore.Checked，用于 PyQt5、PySide2 且被 PySide6 接受）。
for module, enum_name in enums:
    for entry in getattr(module, enum_name):
        setattr(module, entry.name, entry)
```

或者，您可以定义一个自定义函数来处理命名空间查找

```
def _enum(obj, name):
    parent, child = name.split('.')
    result = getattr(obj, child, False)
    if result: # 仅使用短名称进行查找.
        return result

    obj = getattr(obj, parent) # 获取父节点，然后获取子节点.
    return getattr(obj, child)
```

当传入一个对象和一个与 PyQt6 兼容的长格式名称时，此函数将在 PyQt6 和 PySide6 上均返回正确的枚举值或标志。

```
>>> _enum(PySide6.QtCore.Qt, 'Alignment.AlignLeft')
PySide6.QtCore.Qt.AlignmentFlag.AlignLeft
>>> _enum(PyQt6.QtCore.Qt, 'Alignment.AlignLeft')
<Alignment.AlignLeft: 1>
```

如果您在多个文件中这样做，可能会有点麻烦。一个不错的解决方案是将导入逻辑和自定义适配方法移动到一个单独的文件中，例如在项目根目录下命名为 `qt.py`。该模块从两个库中的一个导入 Qt 模块（`QtCore`、`QtGui`、`QtWidgets` 等），然后您可以从那里导入到你的应用程序中。

`qt.py` 文件的内容与我们之前使用的相同 —

```
import sys
if 'PyQt6' in sys.modules:
    # PyQt6
    from PyQt6 import QtGui, QtWidgets, QtCore
    from PyQt6.QtCore import pyqtSignal as Signal, pyqtSlot as Slot
else:
    # PySide6
    from PySide6 import QtGui, QtWidgets, QtCore
    from PySide6.QtCore import Signal, Slot


def _enum(obj, name):
    parent, child = name.split('.')
    result = getattr(obj, child, False)
    if result: # 仅使用短名称进行查找.
        return result

    obj = getattr(obj, parent) # 获取父节点，然后获取子节点.
    return getattr(obj, child)
```

您必须记得在if语句的两支分支中添加您使用的其他PyQt6模块（如browser、multimedia等）。随后，您可以按照以下方式将Qt6导入到您自己的应用程序中：

```
from .qt import QtGui, QtWidgets, QtCore, _enum
```

...并且它将能够在两个库之间无缝运行。

这就是全部了

没什么好说的了——这两个库确实非常相似。不过，如果您在使用PyQt6/PySide6时遇到任何其他示例或功能，而这些内容难以直接转换，请随时与我联系。

附录D：下一步是什么?

本书涵盖了使用 Python 开始创建图形用户界面应用程序所需了解的关键知识。如果您已经阅读到这里，那么您应该已经准备好创建自己的应用程序了！

但在构建应用程序时，还有许多东西需要探索。为了帮助您实现这一目标，我会在 配套网站 上定期发布技巧、教程和代码片段。与本书一样，所有示例均获得 MIT 许可，可自由融入您自己的应用程序中。您可能还对加入我的 Python 图形用户界面学院 感兴趣，在那里，我提供了涵盖本书主题及更多内容的视频教程！

感谢阅读，如有任何反馈或建议，请随时告诉我！

获取更新内容

如果您直接从我这里购买了这本书，您将自动获得本书的更新。如果您在其他地方购买了这本书，您可以将收据发送给我以获取未来更新的访问权限。

参考文档

资源
Qt6 文档
PyQt6 库文档
PySide “Qt for Python” 库文档

版权

本书的翻译版本为 Drtxdt 发起的项目，在 Github 开源，不收取任何费用