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

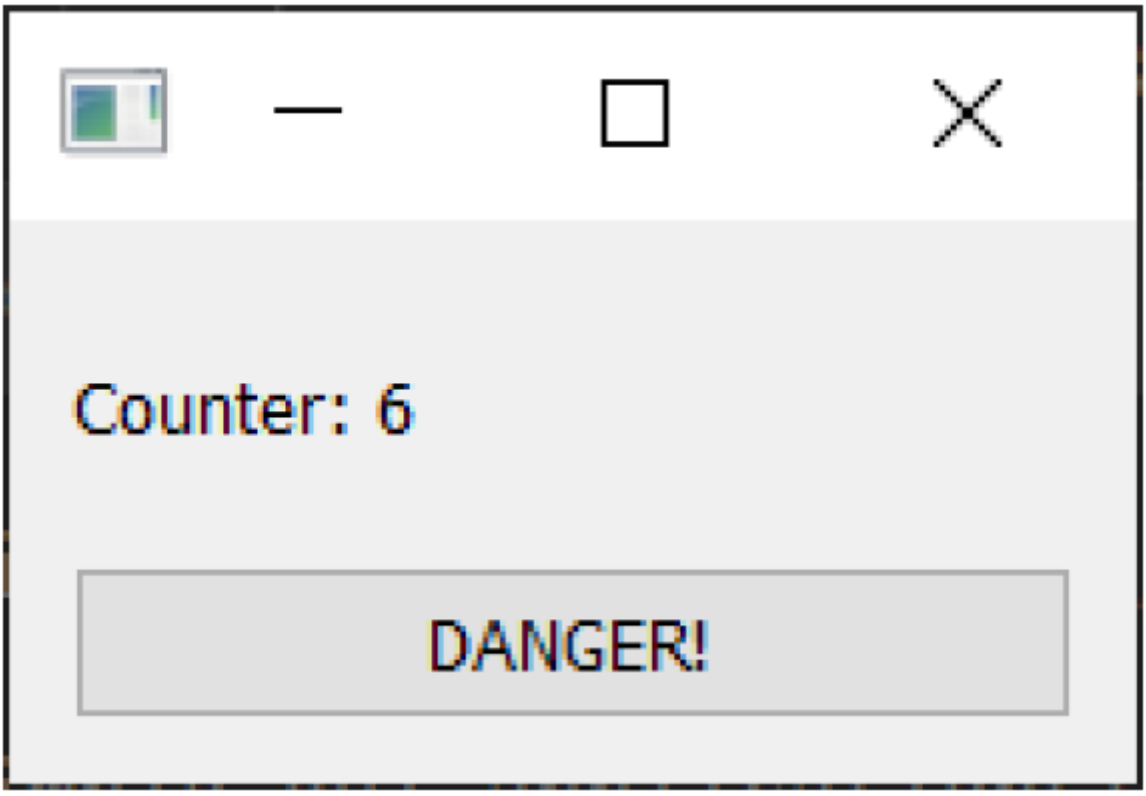
以下是一个用于 PyQt6 的最小示例应用程序，它将使我们能够演示问题并随后进行修复。您可以将此代码复制并粘贴到一个新文件中，并将其保存为适当的文件名，例如 `concurrent.py`。

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()
        )
```

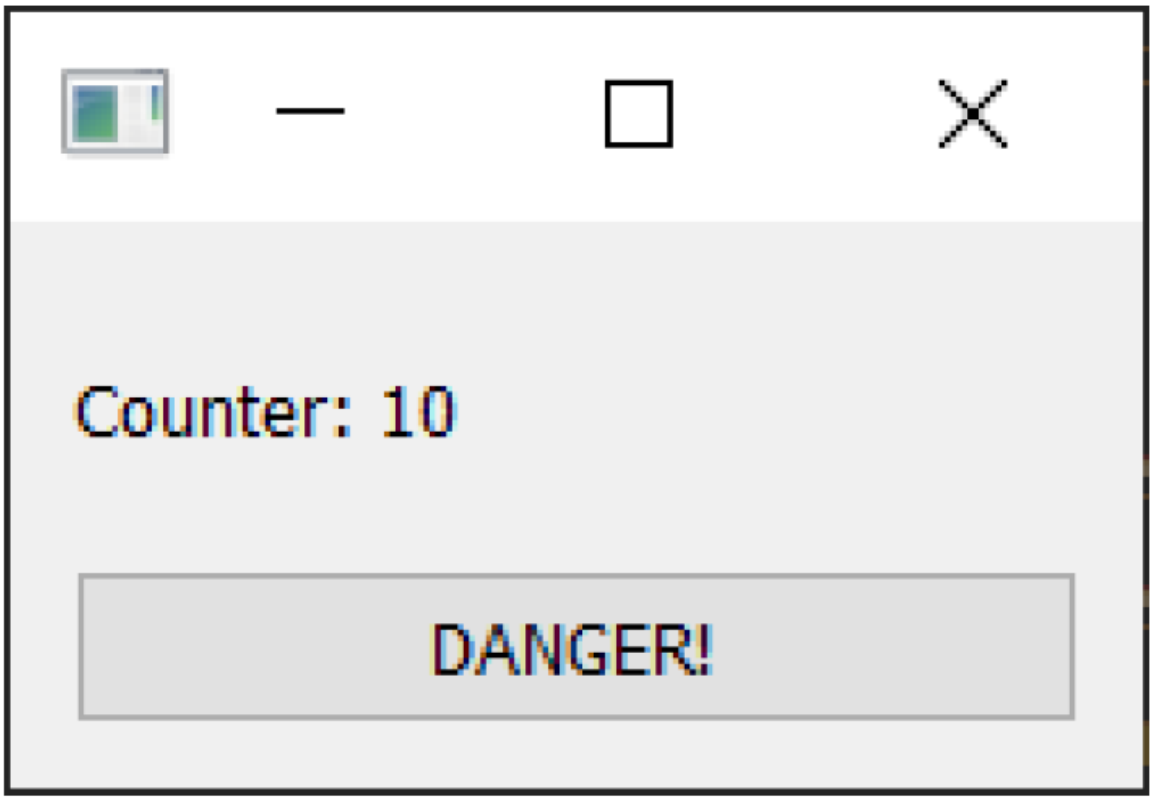
最后，用以下代码替换 `oh_no` 方法，以创建并提交

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

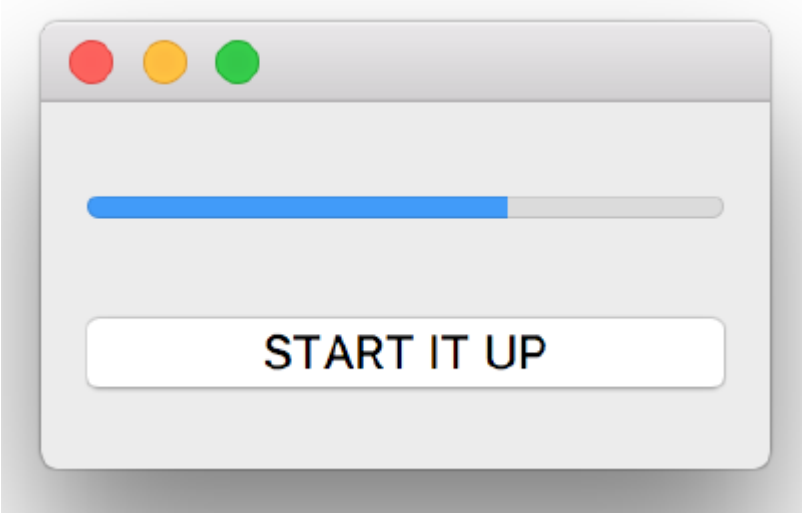
如果您正在使用线程来执行耗时较长的操作，您应该让用户了解任务的进展情况。一种常见的做法是通过显示一个进度条来实现，该进度条会以从左到右填充的方式，显示任务完成的进度。为了在任务中显示进度条，您需要从 `worker` 中发出当前的进度状态。

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。

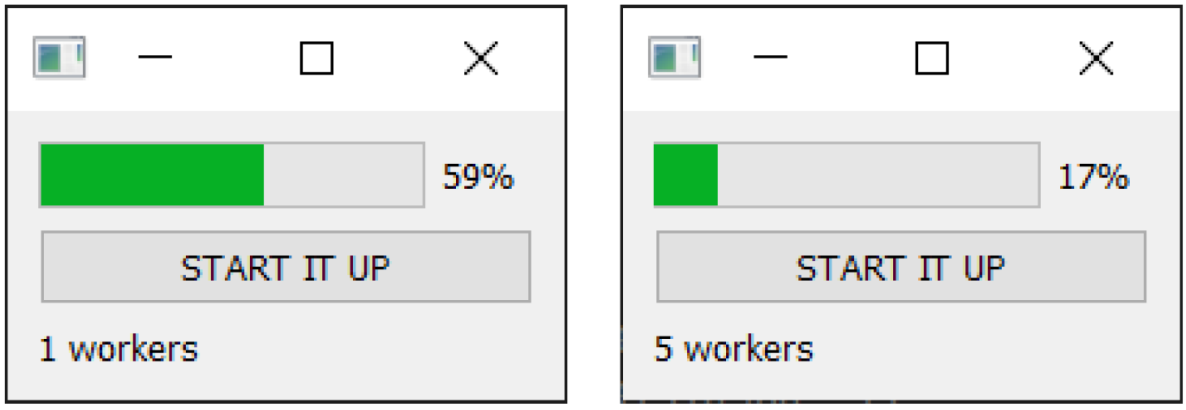


图202：显示全局进度状态的窗口，以及活跃工作者的数量。

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。

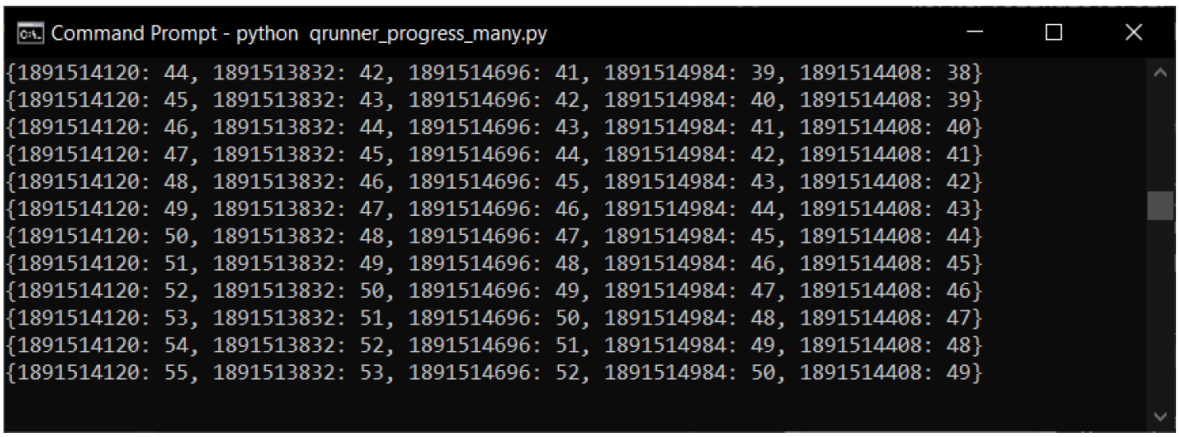


图203：查看 shell 输出以查看每个工作进程的进度。

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)
    """
```

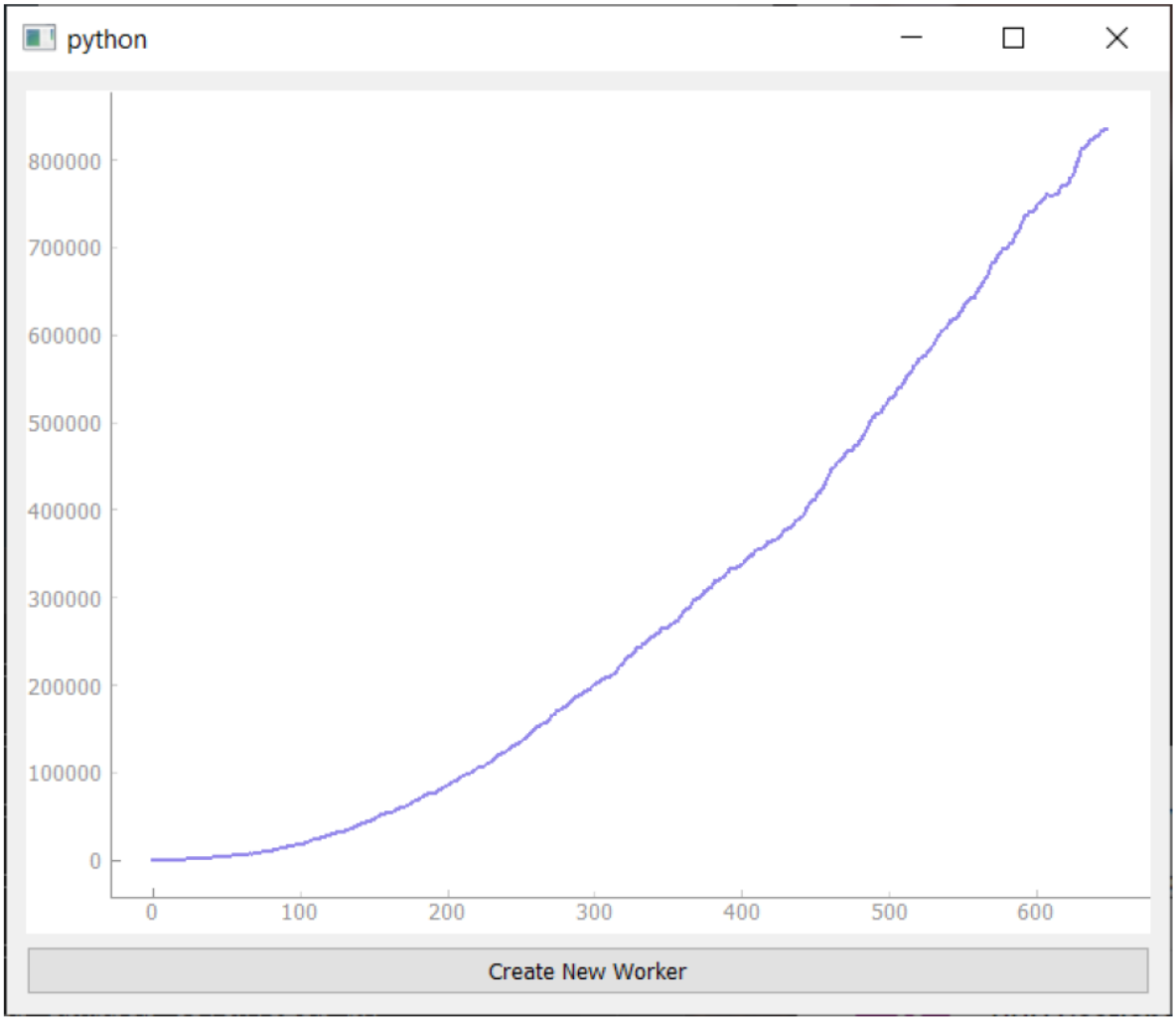
```
    data = pyqtSignal(tuple) #1


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。

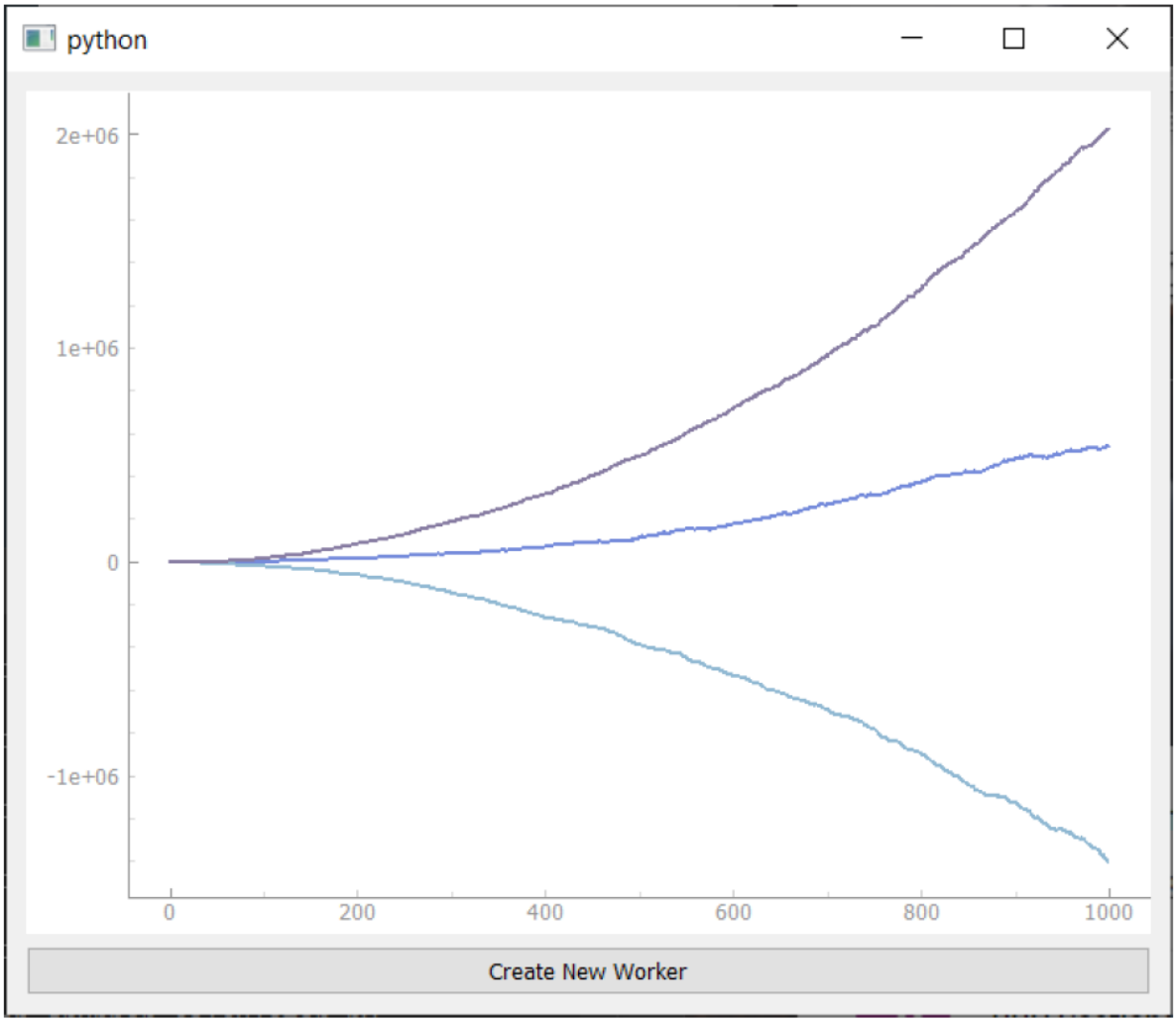


图205：来自多个运行器的数据

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

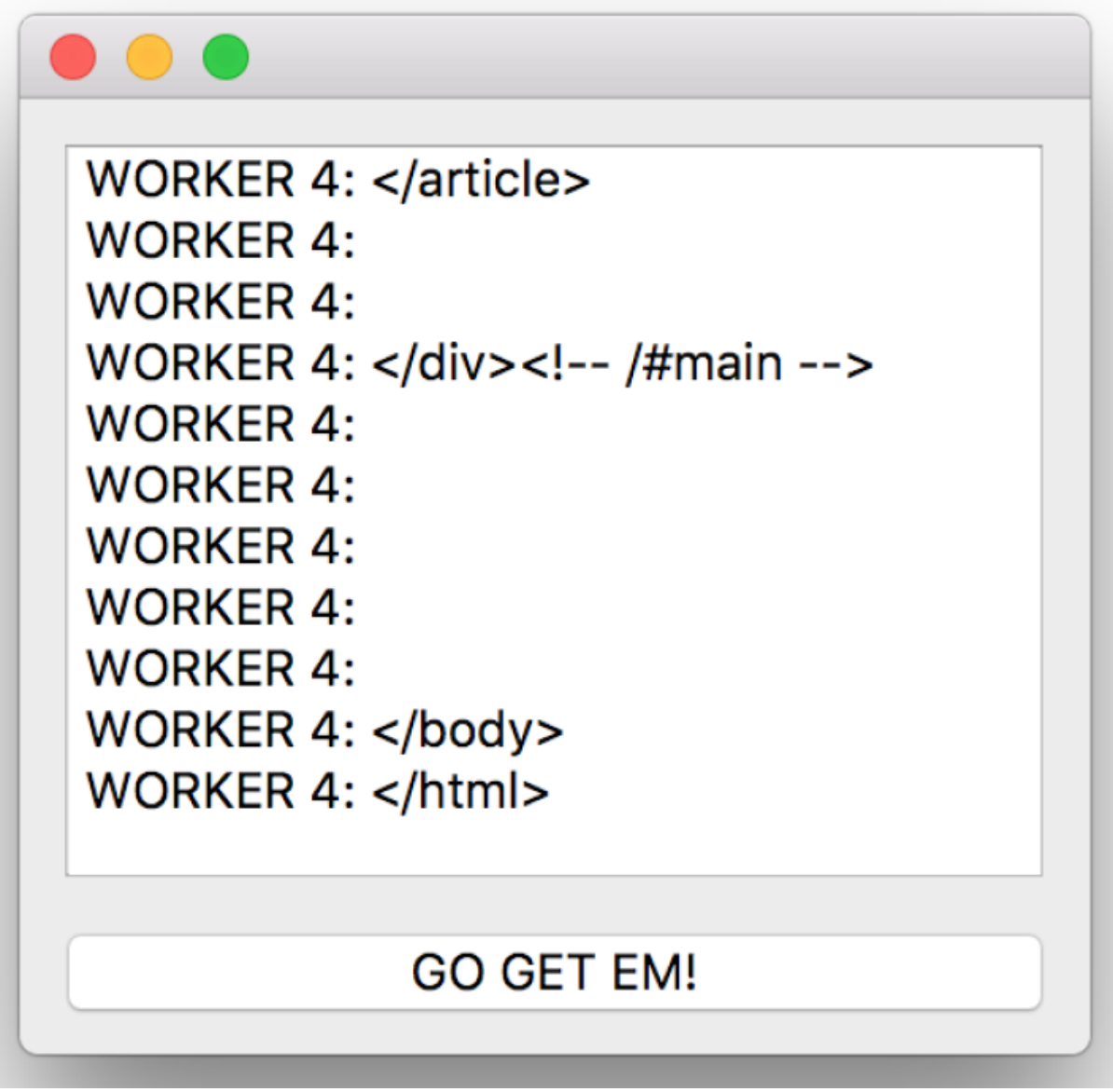
```
import sys
import requests

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。

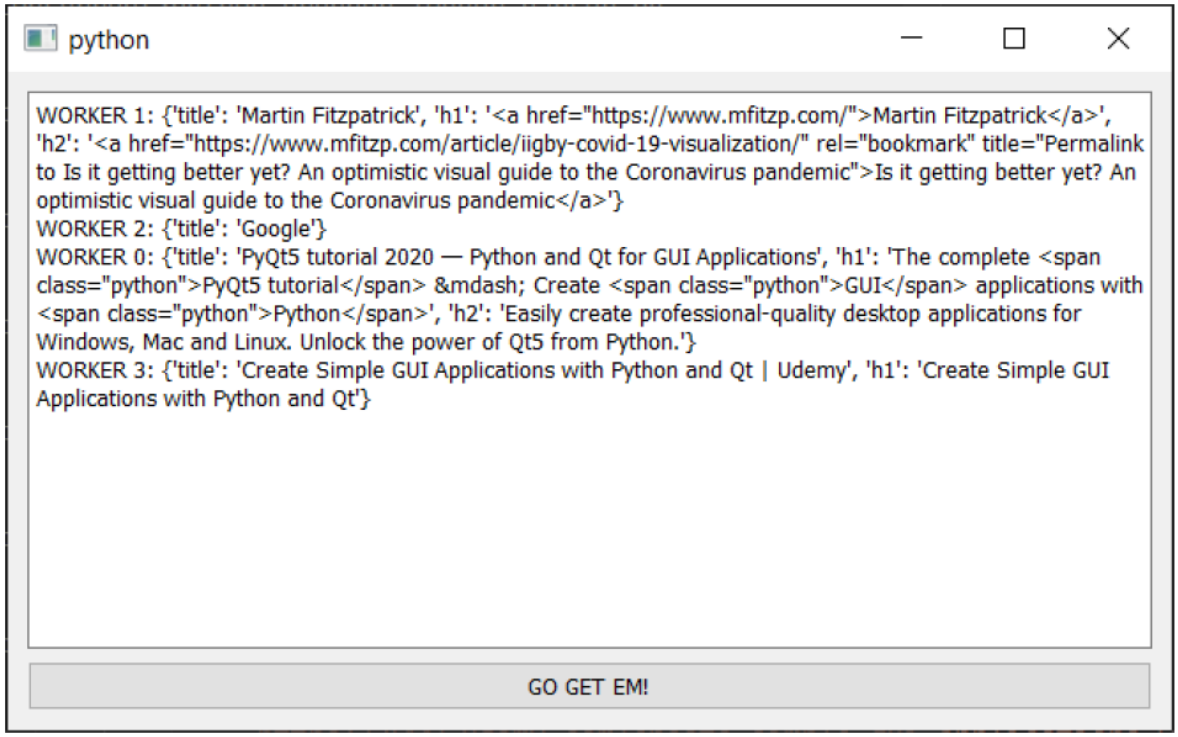


图208：显示多个工作进程解析后的输出结果

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。

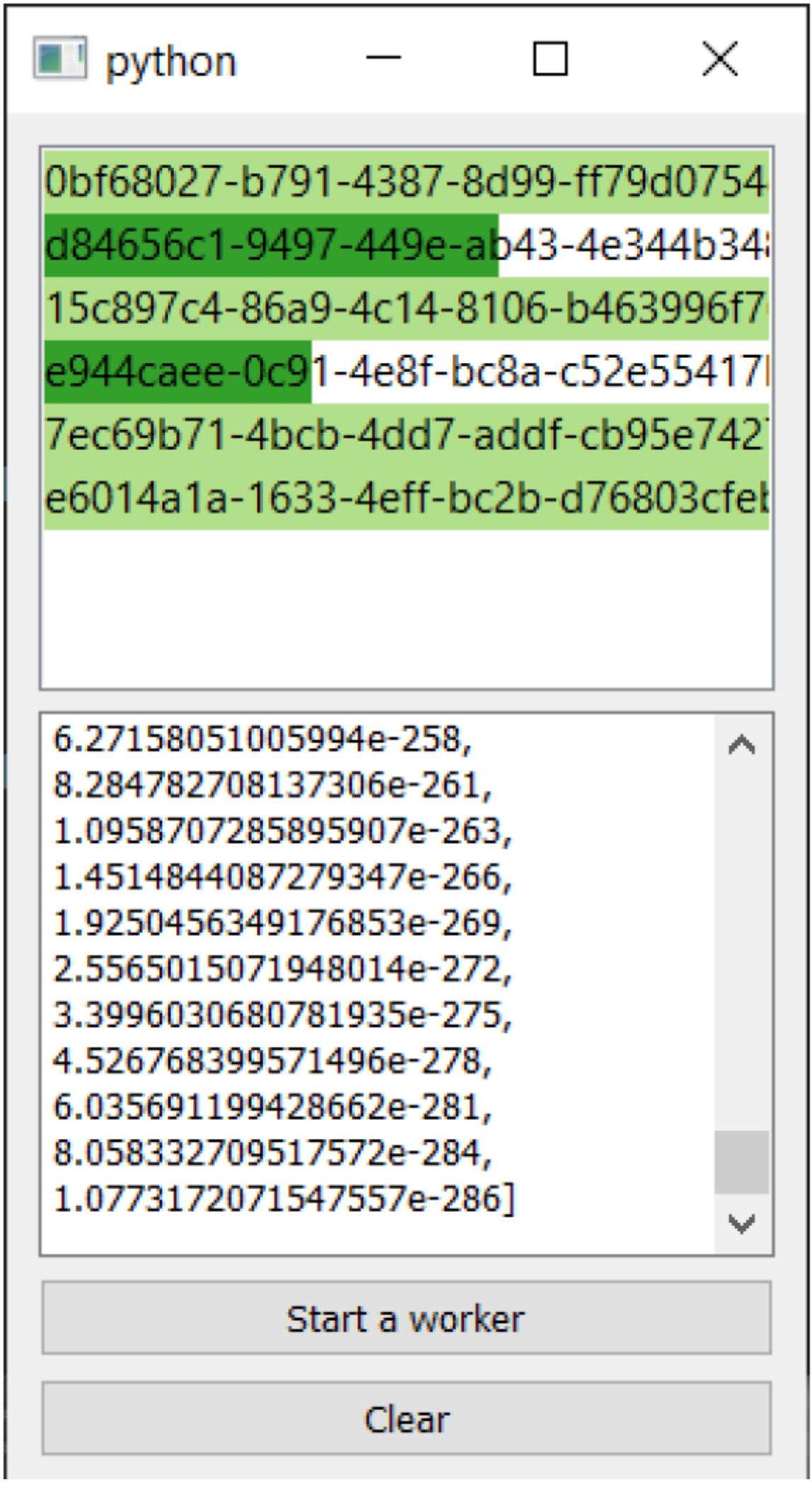
开始一个任务

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

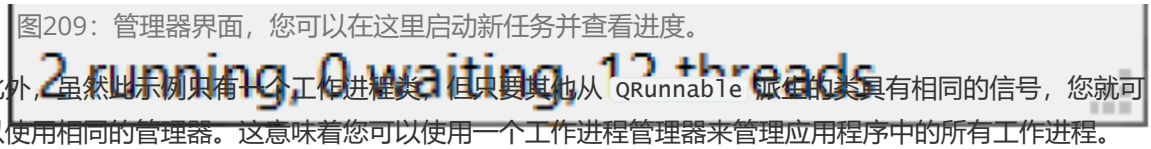


图209：管理器界面，您可以在这里启动新任务并查看进度。

此外，虽然此示例只有一个工作进程类，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。

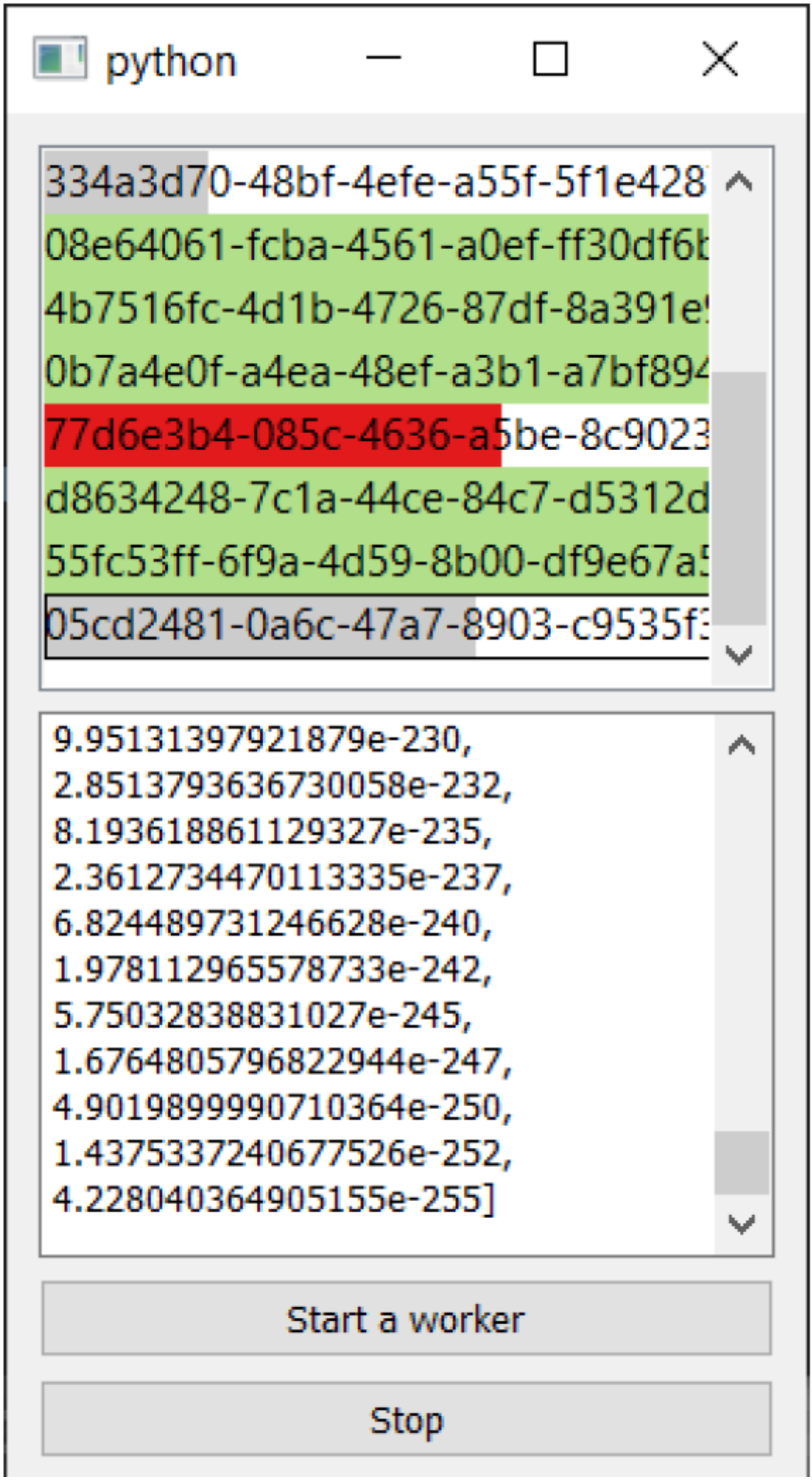
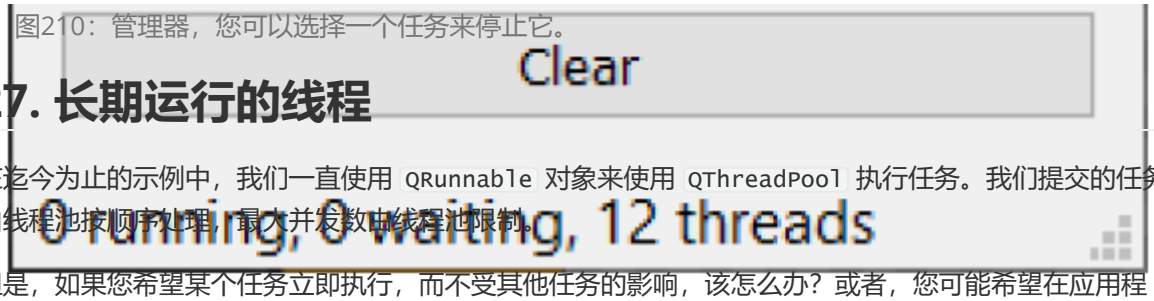

python
334a3d70-48bf-4efe-a55f-5f1e428
08e64061-fcba-4561-a0ef-ff30df6
4b7516fc-4d1b-4726-87df-8a391e
0b7a4e0f-a4ea-48ef-a3b1-a7bf894
77d6e3b4-085c-4636-a5be-8c9023
d8634248-7c1a-44ce-84c7-d5312d
55fc53ff-6f9a-4d59-8b00-df9e67a
05cd2481-0a6c-47a7-8903-c9535f
9.95131397921879e-230,
2.8513793636730058e-232,
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）。

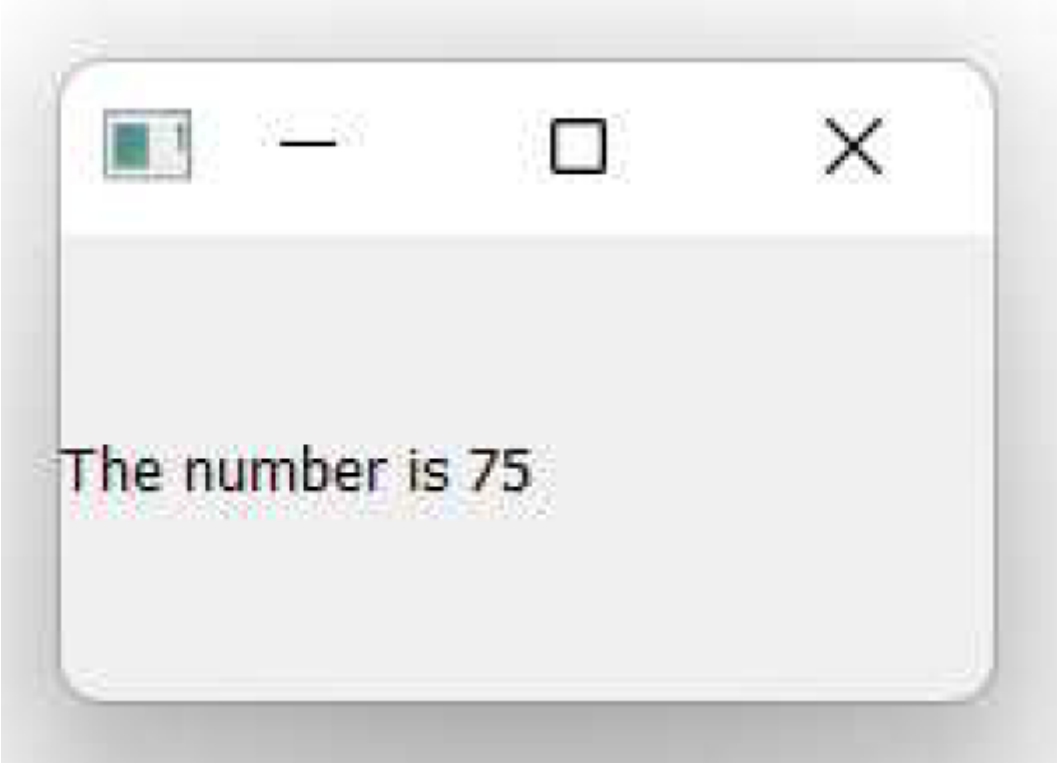


图211：通过信号显示结果的 QThread 计数器

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。

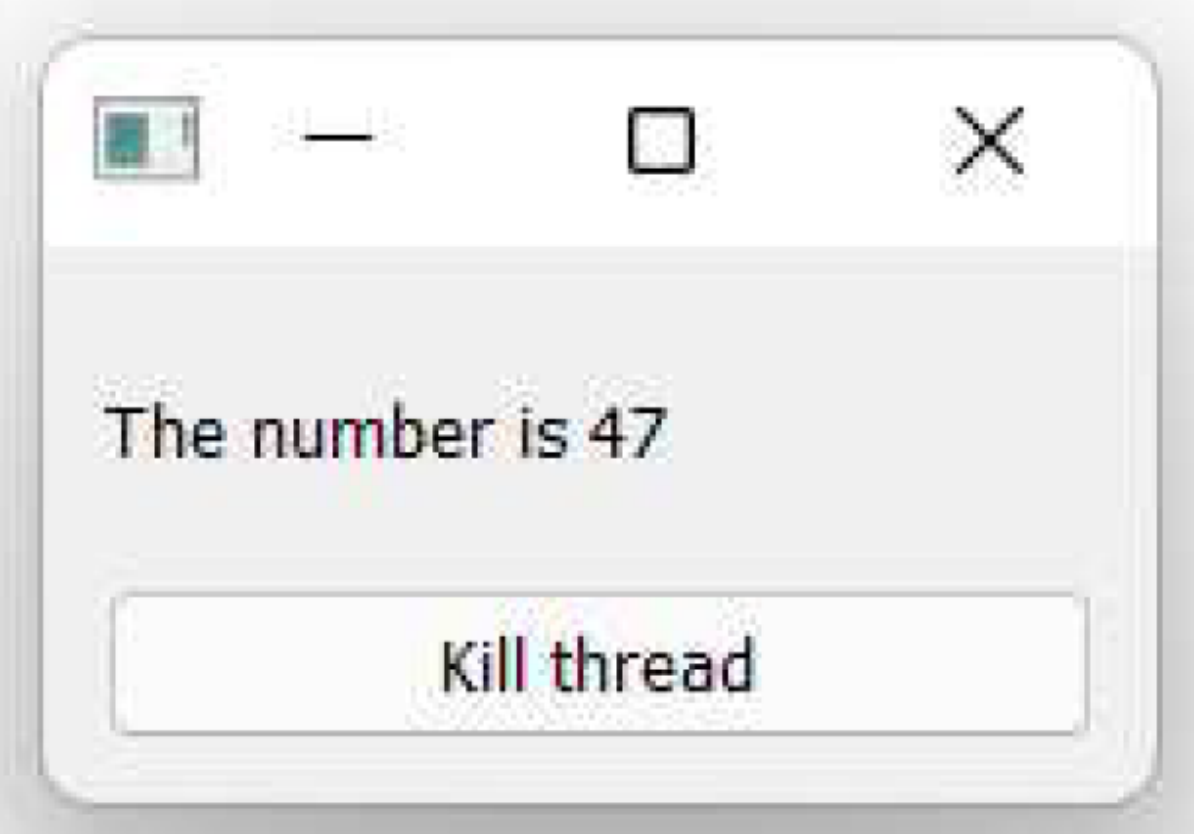


图212：可以通过按钮控制来终止该线程。

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

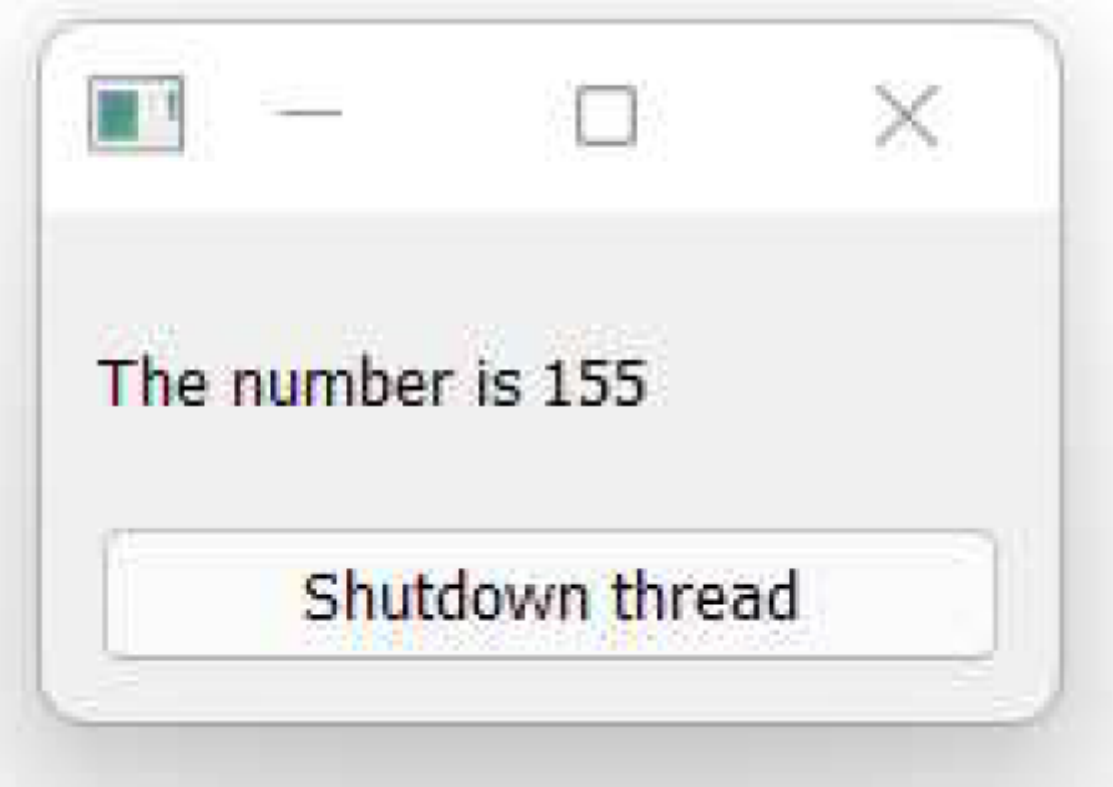
```
        button = QPushButton("Shutdown thread")
        # 优雅地关闭线程.
        button.pressed.connect(self.thread.stop)
```

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。

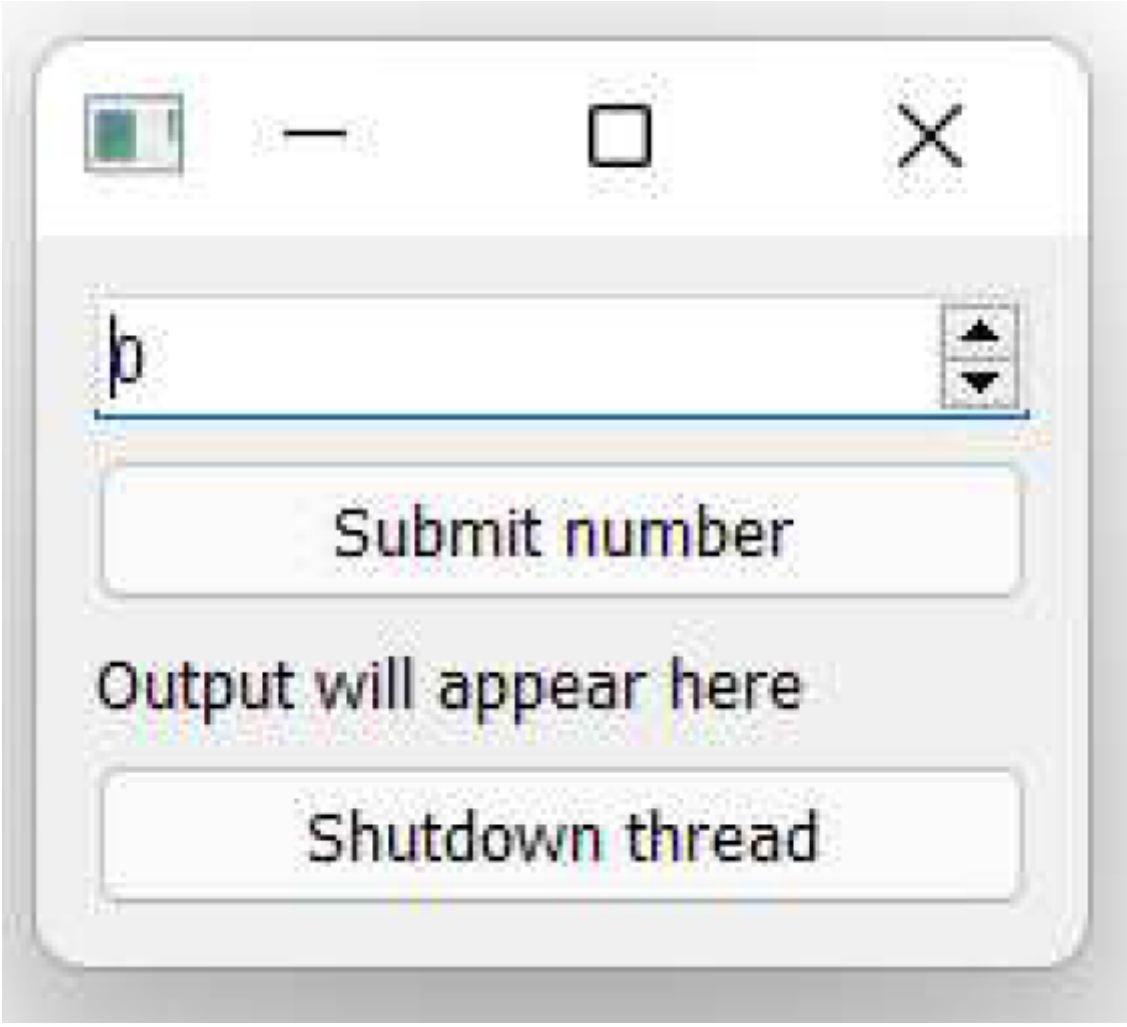
Listing 212. concurrent/qthread_4.py

```
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。

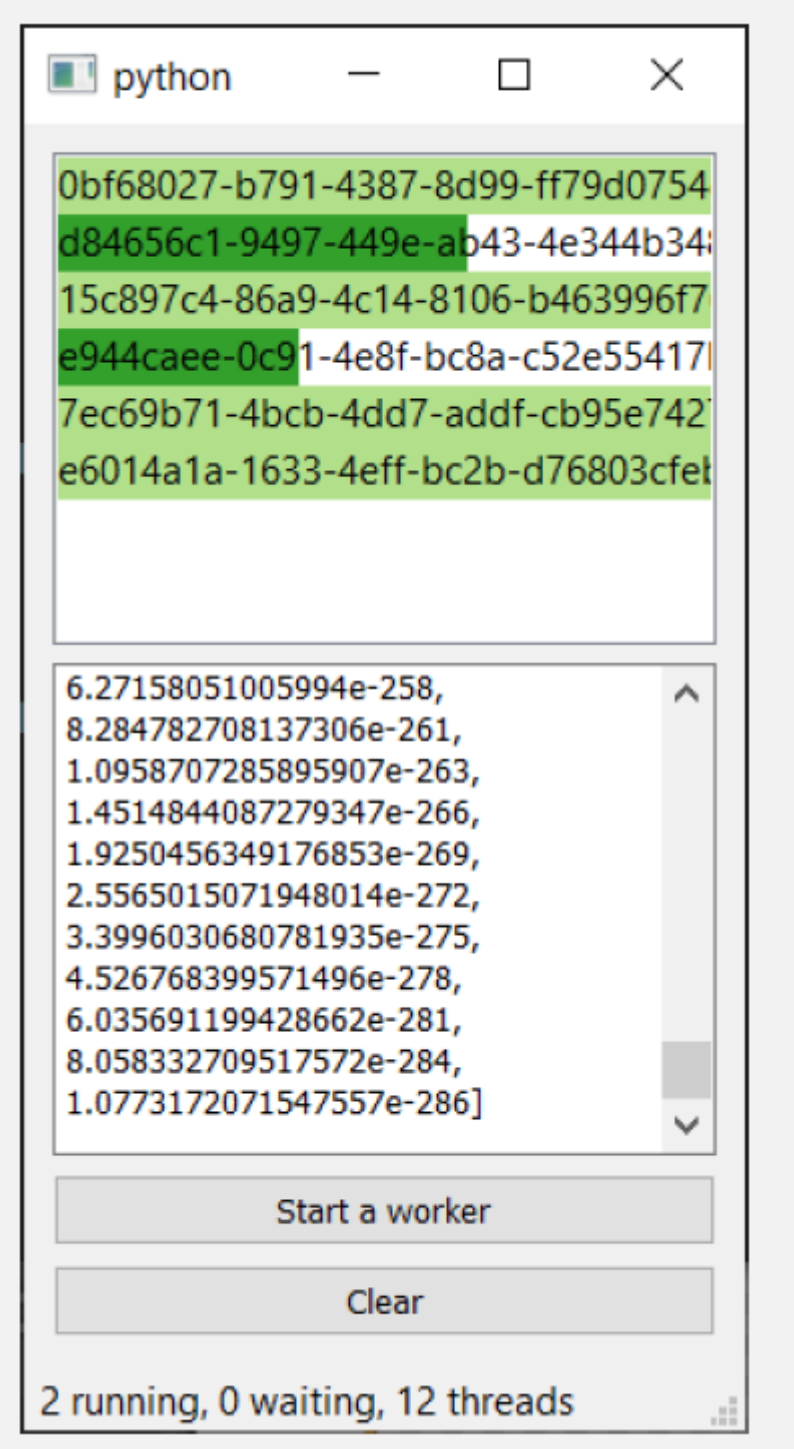
使用过程的感受

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。

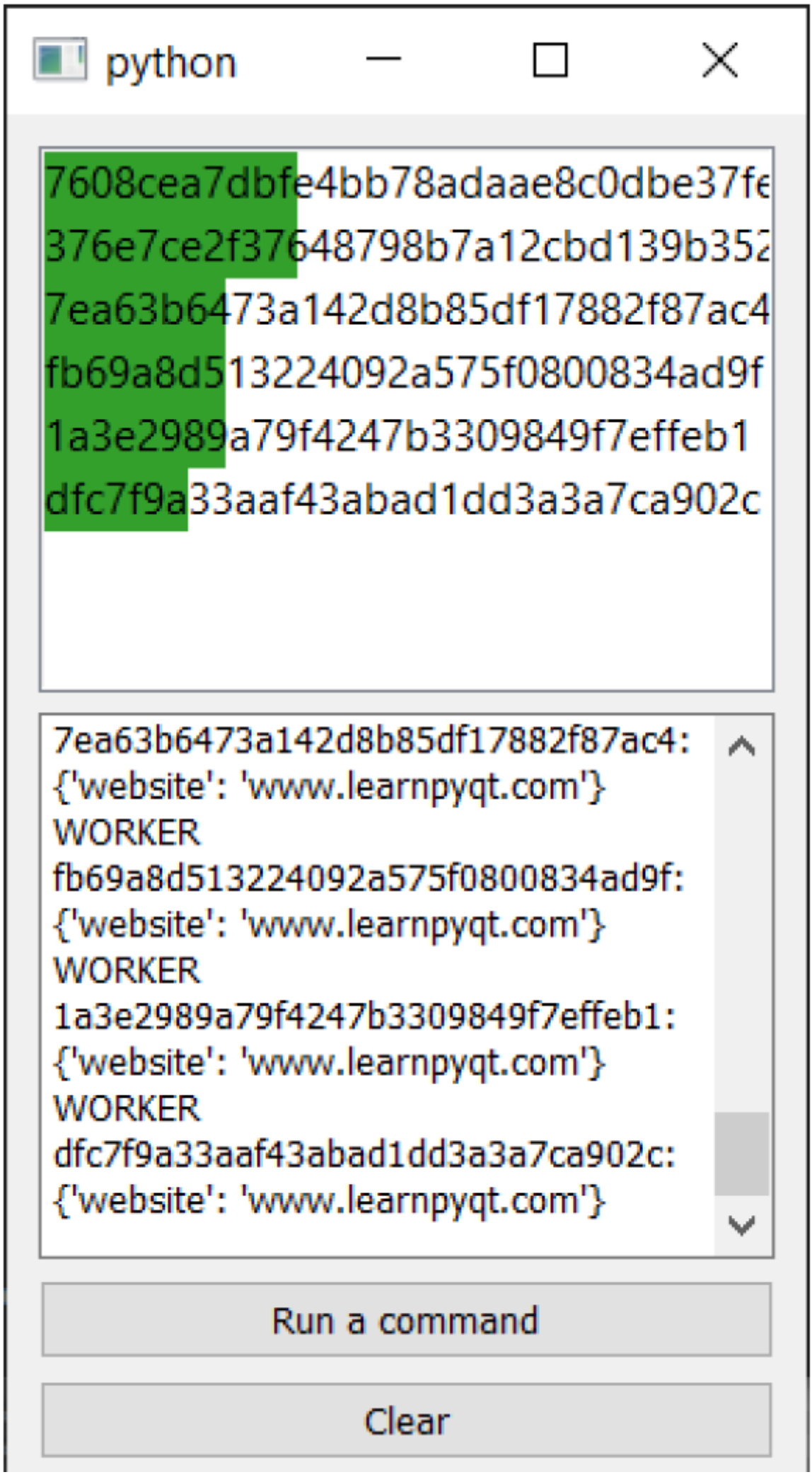

python
7608cea7dbfe4bb78adaae8c0dbe37fe
376e7ce2f37648798b7a12cbd139b352
7ea63b6473a142d8b85df17882f87ac4
fb69a8d513224092a575f0800834ad9f
1a3e2989a79f4247b3309849f7effeb1
dfc7f9a33aaf43abad1dd3a3a7ca902c
7ea63b6473a142d8b85df17882f87ac4: {'website': 'www.learnpyqt.com'}
WORKER fb69a8d513224092a575f0800834ad9f: {'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