0. 前言-让我们开始吧

在开始构建我们的第一个 BeeWare app 之前，确保已经安装好所有 BeeWare 运行依赖的包。

0.1. 安装 python

首先需要安装 python 编译器，运行 python3.7 或更高版本

0.1.1. MacOS

如果你使用 macos 系统，可以从 python 官网获取官方安装包。你可以使用 3.7 之后的任何稳定版本的 Python。我们建议避免使用阿尔法，贝塔和其他已经发布的候选版本除非你真的知道自己在做什么。

0.1.2. Linux

如果你使用 Linux 系统，最好使用系统包管理器安装 Python(在 Debian/Ubuntu/Mint 系统使用 apt 命令，在 Fedora 系统使用 dnf 命令, 在 Arch 系统使用 pacman 命令)。

你应该确保安装的 python 版本为 3.7 或更新，如果不是的话（例如，Ubuntu 18.04 附带 Python 3.6），可能需要使用源码安装。（在 Ubuntu 系统使用 deadsnakes PPA 安装）

此时 Ubuntu 不支持树莓派。

0.1.3. Windows

如果你使用 Windows 系统，可以从 python 官网获取官方安装包。您可以使用 3.7 之后的任何稳定版本的 Python。我们建议避免使用阿尔法，贝塔和其他已经发布的候选版本除非你真的知道自己在做什么。

0.1.4. 其他可替代的已发行 Python 版本

安装 Python 的方式有很多种。可通过 Homebrew 安装 Python。可通过 pyenv 管理一台电脑上的多个 Python。Windows 用户可以从 Windows 应用商店中安装 Python。有数据科学背景的用户可能想使用 Anaconda 或 Miniconda。

怎么安装上的 Python 不重要，重要的是能通过系统的命令提示符或终端应用运行 Python3，并且使 Python 编译器工作起来。

0.2. 安装依赖包

0.2.1. MacOS

在 MacOS 系统上构建 BeeWare 需要：

Git，一种分布式版本控制系统。可以从官网下载 Git

Xcode，苹果的 IDE，可以从 MacOS 应用商店免费获取。

0.2.2. Linux

为了支持本地开发，需要安装一些系统包。所需的包列表根据您的发行版而有所不同:

Ubuntu 16.04 / Debian 9

```
1. $ sudo apt-get update
2. $ sudo apt-get install git python3-dev python3-venv python3-gi python3-gi-cairo libgirepository1.0-dev libcairo2-dev libpango1.0-dev libwebkitgtk-3.0-0 gir1.2-webkit2-3.0
```

Ubuntu 18.04, 20.04 / Debian 10, 11

```
1. $ sudo apt-get update
2. $ sudo apt-get install git python3-dev python3-venv python3-gi python3-gi-cairo libgirepository1.0-dev libcairo2-dev libpango1.0-dev libwebkit2gtk-4.0-37 gir1.2-webkit2-4.0
```

Fedora

```
1. $ sudo dnf install git pkg-config python3-devel gobject-introspection-devel cairo-devel cairo-gobject-devel pango-devel webkitgtk4
```

Arch, Manjaro

```
$ sudo pacman -Syu git pkgconf cairo python-
cairo pango gobject-introspection gobject-introspection-
runtime python-gobject webkit2gtk
```

Briefcase 也可以使用 AppImage 构建可跨 Linux 发行版使用的二进制文件。然而，因为每个发行版上存在不一致的库版本，为 Linux 构建 AppImage 二进制文件有点复杂。Briefcase 使用 Docker 托管 AppImage 构建，并提供了一个易于控制的二进制环境。

Docker Engine 的官方安装程序可用于一系列 Unix 发行版。按照平台的说明操作。一旦安装了 Docker，你应该能够启动一个 Ubuntu 18.04 容器:

```
$ docker run -it ubuntu:18.04
```

这个命令会在 Docker 容器中显示一个 Unix 提示符(类似于 root@84444e31cff9:/#)。Ctrl+D 退出 Docker 并返回到本地 shell。

0.2.3. Windows

在 Windows 系统上构建 BeeWare 需要：

Git，一种分布式版本控制系统。可以从官网下载 Git

安装这些工具后，重新启动所有终端。Windows 只会在安装完成后显示新安装的工具终端。

0.3. 设置虚拟环境

现在将要创建一个虚拟环境-一个“沙盒”，我们可以使用它将本教程的工作与主 Python 安装隔离开来。如果我们将安装包安装到虚拟环境中，主 Python（包括我们电脑上的其他 Python 工程）的安装就不会受影响。如果把虚拟环境弄得一团糟，简单的方法是删除它并重新开始，这样不影响计算机上的任何其他 Python 项目，也不需要重新安装 Python。

0.3.1. MacOS

```
$ mkdir beeware-tutorial
$ cd beeware-tutorial
$ python3 -m venv beeware-venv
$ source beeware-venv/bin/activate
```

0.3.2. Linux

```
$ mkdir beeware-tutorial
$ cd beeware-tutorial
$ python3 -m venv beeware-venv
$ source beeware-venv/bin/activate
```

0.3.3. Windows

```
C:\...>md beeware-tutorial
C:\...>cd beeware-tutorial
C:\...>py -m venv beeware-venv
C:\...>beeware-venv\Scripts\activate.bat
```

如果上述操作生效了，你的提示符现在应该有变化-多了(beeware-venv)前缀。这使你确认现在处于 BeeWare 虚拟环境中。学习本教程时，需要确保激活虚拟环境。如果没在激活状态下，重新运行最后一条命令（激活命令）以重新激活虚拟环境。

0.3.4. 其他虚拟环境

如果你使用 Anaconda 或 miniconda，可能对 conda 环境更熟悉。你也可能听说过 virtualenv，是 Python 内置 venv 模块的前身。与 Python 安装一样——如何创建虚拟环境并不重要，只要有虚拟环境。即便如此——严格来说，使用虚拟环境也是可选的。您可以将 BeeWare 的工具直接安装到您的主 Python 环境中。但是，非常非常非常建议使用虚拟环境。

0.4. 下一步

现在，我们已经设置好了环境。接下来准备创建第一个 BeeWare 应用程序。

1. 第一章-第一个 app

我们将要创建第一个应用

1.1. 安装 BeeWare 工具

首先需要安装 Briefcase，这是一个用来把你的应用打包已发送给终端用户的 BeeWare 工具，但是也可以用来引导一个新项目。确保你在教程 0 中创建的 beeware-tutorial 目录中，激活 beeware-venv 虚拟环境，然后运行:

1.1.1. MacOS

```
(beeware-venv) $ python -m pip install briefcase
```

1.1.2. Linux

```
(beeware-venv) $ python -m pip install briefcase
```

如果安装过程中报错了，极有可能是系统所需的包没有安装。保你已经安装了系统所需的所有包。

1.1.3. Windows

```
(beeware-venv)C:\...>python -m pip install briefcase
```

1.2. 新建项目

让我们开始第一个 BeeWare 项目吧！使用 Briefcase new 命令创建一个名为 HelloWorld 的应用。在你的命令提示符中运行如下命令：

1.2.1. MacOS

```
(beeware-venv) $ briefcase new
```

1.2.2. Linux

```
(beeware-venv) $ briefcase new
```

1.2.3. Windows

```
(beeware-venv) C:\...>briefcase new
```

Briefcase 会向我们询问一些新应用的细节问题。为学习新建项目，请参照一下选项：

正式名称—接受默认值:Hello World。

应用名称—接受默认值:helloworld。

Bundle -如果你拥有自己的域名，按相反的顺序输入该域名。(例如，如果你拥有“cupcakes.com”域名，输入 com.cupcakes)。如果您没有自己的域，则接受默认包(com.example)。

项目名称-接受默认值:Hello World。

说明-接受默认值（或者，如果你想使项目更具创造性，提出自己的说明）

作者-此处输入你自己的名字

作者的邮箱-输入你自己的邮箱地址。邮箱地址会被记录在配置文件中，在帮助文档或其他地方。当你把 app 提交到应用商店的时候需要邮箱。

URL -您的应用程序的登录页面的 URL。同样，如果您拥有自己的域名，请在该域名上输入一个 URL(前缀 https://). 否则，只需接受默认 URL (https://example.com/helloworld)。这个 URL 不需要实际存在(目前);它只会在你发布应用到应用商店时使用。

许可-接受默认许可(BSD)。不过，这不会影响教程的操作——因此，如果您对许可选择特别重要，请随意选择另一个许可。

GUI 框架-接受默认选项 Toga (BeeWare 自己的 GUI 工具包)。

之后 Briefcase 会生成一个项目框架供你使用。如果你按照教程操作得到此处并且接受了那些默认选项，你的文件系统应该是这样的：

```
beeware-tutorial/
    beeware-venv/
        ...
    helloworld/
        LICENSE
        README.rst
        pyproject.toml
        src/
            helloworld/
                resources/
                    helloworld.icns
                    helloworld.ico
                    helloworld.png
                __init__.py
                __main__.py
                app.py
```

这个框架是一个具有完整功能的 app，没有添加其他东西。Src 文件夹包含这个 app 的所有代码。Pyproject.toml 文件描述了如何对应用程序进行打包以便

发行。如果在编辑器中打开 pyproject.toml，你将看到刚才提供给 Briefcase 的配置细节。

现在我们有一个可扩展 app，可以使用 Briefcase 来运行这个 app。

1.3. 在开发者模式下运行应用程序

进入 HelloWorld 工程的目录并告诉 Briefcase 在开发者模式运行项目。

1.3.1. MacOS

```
(beeware-venv) $ cd helloworld
(beeware-venv) $ briefcase dev

[hello-world] Installing dependencies...
...
[helloworld] Starting in dev mode...
```

1.3.2. Linux

```
(beeware-venv) $ cd helloworld
(beeware-venv) $ briefcase dev

[hello-world] Installing dependencies...
...
[helloworld] Starting in dev mode...
```

1.3.3. Windows

```
(beeware-venv) C:\...>cd helloworld
(beeware-venv) C:\...>briefcase dev

[hello-world] Installing dependencies...
...
[helloworld] Starting in dev mode...
```

安装过程中可能出现的错误：

如果您使用的是最新版本的 Python(3.9+)，此步骤在安装依赖项时可能会引发错误。这通常表现为:

```
Traceback (most recent call last):
      File "<string>", line 1, in <module>
```

```
    File "C:\...\Local\Temp\pip-install-ytuu_37_\pythonnet\setup.py", line 18, in <module>
      from wheel import bdist_wheel
  ModuleNotFoundError: No module named 'wheel'
  ----------------------------------------
ERROR: Command errored out with exit status 1: python setup.py egg_info Check the logs for full command output.
```

根据您环境的具体情况，它也可能表现为包括以下内容的报告:

```
Building wheel for pythonnet (setup.py) ... error
  ERROR: Command errored out with exit status 1:
  ...
    File "c:\...\Local\Programs\Python\Python38\lib\subprocess.py", line 364, in check_call
      raise CalledProcessError(retcode, cmd)
  subprocess.CalledProcessError: Command '['...\\python.exe', 'tools\\geninterop\\geninterop.py', 'src\\runtime\\interop38.cs']' returned non-zero exit status 1.
  ----------------------------------------
  ERROR: Failed building wheel for pythonnet
  Running setup.py clean for pythonnet
Failed to build pythonnet
```

导致在上申诉错误的原因可能是教程中的 app 依赖 Python for .NET 包去获取 windows 上的系统库。Python for .NET 不是一个纯 Python 包，它还包含一些需要编译的 c 模块。Python for .NET 发布了这些模块的预编译版本，但在支持最新版本的 Python 方面，它们有时会落后。

如果你遇到了这些错误，有四种解决方法：

①使用旧版本的 Python ，查看 pythonnet entry on PyPI 确定 Python for .NET 目前支持的 Python 版本。

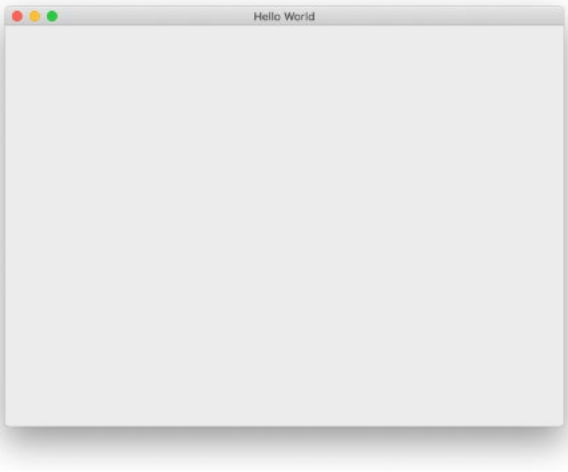
②安装 Python for .NET 的预发布版本，3.0 版，目前正在 alpha 测试中，是一个纯 Python 包，而且只需要在你的电脑上安装.NET 5 SDK。不幸的是，这种方法没有文档可供查询，详情请咨询 Python for .NET 的支持频道。

③配种子环境使之支持编译 Python for .NET。这是一个有点复杂的过程，而且你需要安装 Visual Studio。详情请看 the Python for .NET wiki

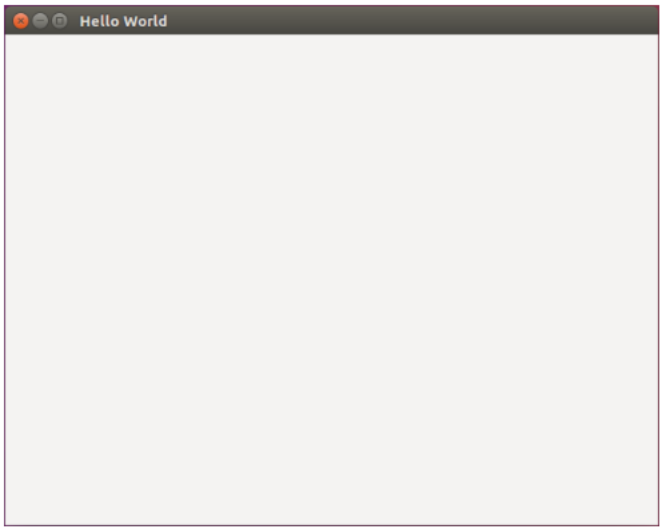
④安装非官网的已 编译 wheel 包。Python for .NET 团队建议使用这个 wheel 包的集合作为源。.whl 文件可通过 `python -m pip install somefile.whl` 命令安装；然后需要重新运行 `python -m pip install --pre beware`

此时应该会打开一个 GUI 窗口：

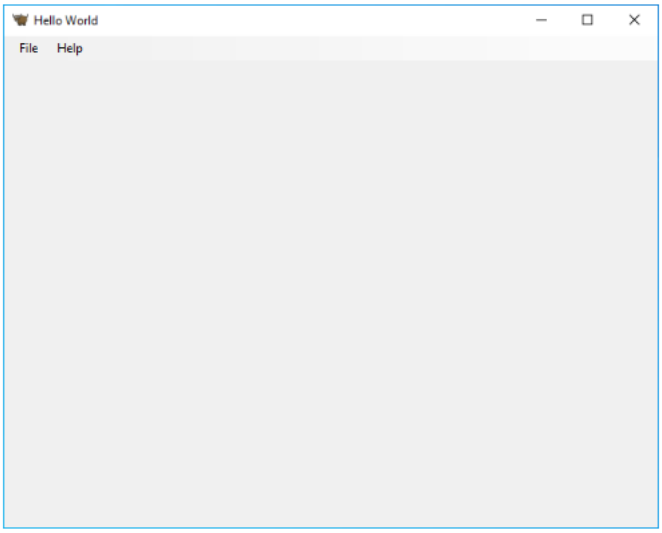
1.3.4. MacOS



1.3.5. Linux



1.3.6. Windows



按下关闭按钮或者选择 app 的菜单中的退出，然后就完成啦！祝贺你-你 使用 Python 写了一个独立的本地 app。

1.4. 下一步

现在有一个可以工作的 app，并且以开发者模式运行。现在我们可以加一些逻辑使我们的 app 做一些更有趣的事情。在第二章，我们会向 app 中添加更多有用的用户界面。

2. 第二章-使我们的项目更有趣

在第一章中，我们生成了一个可运行可扩展的项目，但里面没有添加任何自己的代码，让我们看看生成的是什么样子吧。

2.1. 生成了什么

在 src/helloworld 目录下，可以看到三个文件__init__.py, __main__.py 和 app.py。__init__.py 将 helloworld 目录标记为一个可导入的 Python 模块。这是一个空文件，它的存在事实上是为了告诉 Python 编译器 helloworld 目录定义了一个模块。

__main__.py 将 helloworld 目录标记为一个特殊的可执行模块。如果你尝试使用 `python -m helloworld` 命令运行 helloworld 模块，Python 将会从__main__.py

文件处开始执行。__main__.py 文件的目录相当简单：

```
from helloworld.app import main

if __name__ == '__main__':
    main().main_loop()
```

__main__.py 文件从 helloworld app 中导入 main 函数，如果它作为入口点执行，则调用 main 函数，并启动应用程序的主循环。主循环是 GUI 应用程序监听用户输入（如点击鼠标或摁下键盘）的方法。

最有趣的文件是 app.py，其中包含创建 app 窗口的逻辑：

```
import toga
from toga.style import Pack
from toga.style.pack import COLUMN, ROW

class HelloWorld(toga.App):
    def startup(self):
        main_box = toga.Box()

        self.main_window = toga.MainWindow(title=self.formal_name)
        self.main_window.content = main_box
        self.main_window.show()

def main():
    return HelloWorld()
```

让我们一行一行来看：

```
import toga
from toga.style import Pack
from toga.style.pack import COLUMN, ROW
```

首先，导入 toga 控件工具包和一些与样式相关的使用程序类和常量。我们的代码还没有用到这些-但很快就会用到。

然后，定义一个类：

```
class HelloWorld(toga.App):
```

每一个 Toga 应用程序都有一个单独的 toga.App 应用实例，表示应用程序正在运行的实体。App 最终会管理多个窗口，但对于简单的 app 来说，只有一个主窗口。

接下来，定义一个 startup()函数：

```
def startup(self):
    main_box = toga.Box()
```

startup 函数中首先定义了一个主盒子。Toga 的布局方案和 HTML 很类似。通过构造一组盒子(每个控件包含其他盒子或实际的小部件)来构建应用程序。然

后将不同的样式应用于这些盒子以确定它们如何使用可用的窗口空间。

在这个 app 中，我们定义了一个单独的盒子，没有放任何东西进去。

接下来定义一个窗口，可以把空的盒子放进去：

```
1.  self.main_window = toga.MainWindow(title=self.formal_name)
```

这将创建一个 `toga.MainWindow` 的实体，标题与 app 的名字一致。主窗口是 Toga 中的一种特殊窗口，与 app 的生命周期密切相关。当主窗口关闭，app 也会退出。主窗口也是 app 的菜单窗口（如果你使用的是 windows 平台，菜单栏就是窗口的一部分）

然后为主窗口中添加空的盒子，并指示 app 显示我们的窗口：

```
1.  self.main_window.content = main_box
2.  self.main_window.show()
```

最后，定义 maian 函数，创建应用程序实例：

```
1.  def main():
2.      return HelloWorld()
```

main 函数就是__main__.py 文件导入并调用的函数之一。它创建并返回我们的 HelloWorld 应用程序实例。

这是最简单的 Toga 应用程序。让我们添加向 app 中添加一些自己的内容使之做一些有趣的事。

2.2. 加入一些我们自己的内容

在 src/ HelloWorld /app.py 中修改 HelloWorld 类，修改成这样:

```
1.  class HelloWorld(toga.App):
2.      def startup(self):
3.          main_box = toga.Box(style=Pack(direction=COLUMN))
4.
5.          name_label = toga.Label(
6.              'Your name: ',
7.              style=Pack(padding=(0, 5))
8.          )
9.          self.name_input = toga.TextInput(style=Pack(flex=1))
10.
11.         name_box = toga.Box(style=Pack(direction=ROW, padding=
5))
12.         name_box.add(name_label)
13.         name_box.add(self.name_input)
14.
```

```
        button = toga.Button(
            'Say Hello!',
            on_press=self.say_hello,
            style=Pack(padding=5)
        )

        main_box.add(name_box)
        main_box.add(button)

        self.main_window = toga.MainWindow(title=self.formal_name)
        self.main_window.content = main_box
        self.main_window.show()

    def say_hello(self, widget):
        print("Hello,", self.name_input.value)
```

2.2.1. 注意

不要删除文件顶部的导入，也不要删除文件底部的 main()。你只需要更新 HelloWorld 类。

让我们详细看看修改了什么。

仍然是创建一个主盒子，但是现在应用一个样式：

```
main_box = toga.Box(style=Pack(direction=COLUMN))
```

Toga 的内置布局系统被称为“Pack”,有一点像 CSS。HTML 中以层次结构定义对象，对象是 ,和其他 DOM 元素；在 Toga 中，对象是控件和盒子。可以为每个元素分配样式。在本例中，我们指出这是一个 COLUMN 盒子，这个盒子将使用所有可用宽度，并随着内容的增加扩展高度，但要使它尽可能短。

接着，我们定义一组控件：

```
name_label = toga.Label(
    'Your name: ',
    style=Pack(padding=(0, 5))
)
self.name_input = toga.TextInput(style=Pack(flex=1))
```

这里，我们定义了一个标签和一个文本输入。每一个控件都有与之关联的样式；标签的左边和右边将有 5px 的填充，顶部和底部没有填充。文本输入被标记为灵活的，它可以使用布局轴上的所有可用空间。文本输入被赋值为该类的实例变量。使我们易于获取待会将要用到的控件实例，。

接着，定义一个盒子包含两个控件：

```
name_box = toga.Box(style=Pack(direction=ROW, padding=5))
```

```
    name_box.add(name_label)
    name_box.add(self.name_input)
```

`name_box` 就像主盒子；然而这是一个 ROW 盒子。意味着它的内容将会水平向增加，要尽可能使它窄。这个盒子每边有 5px 的填充。

现在我们定义一个按钮：

```
button = toga.Button(
    'Say Hello!',
    on_press=self.say_hello,
    style=Pack(padding=5)
)
```

这个按钮每边有 5px 的填充。我们还定义了一个处理程序，当按钮被摁下的时候调用这个函数。

然后我们为这个盒子增加名字，并把它放到主盒子中。

```
main_box.add(name_box)
main_box.add(button)
```

此时完成了布局；其余的启动函数和之前一样-定义一个主窗口，将主盒子指定为窗口的内容：

```
self.main_window = toga.MainWindow(title=self.formal_name)
self.main_window.content = main_box
self.main_window.show()
```

需要做的最后一件事就是为按钮定义一个处理函数。处理函数可以是任何函数，生成器或者异步协同例程；它接受参数为生成事件的控件，并且在按钮被摁下的任何时候被调用：

```
def say_hello(self, widget):
    print("Hello,", self.name_input.value)
```

这个函数的主体是一个简单的 print 语句-然后，它可以查询当前输入的任何值，并将该内容作为文本打印出来。

现在我们已经做了一些修改可以通过再次启动 app 来看看。像以前一样，需要使用开发者模式。

2.2.2. MacOS

```
(beeware-venv) $ briefcase dev

[helloworld] Starting in dev mode...
```

2.2.3. Linux

```
(beeware-venv) $ briefcase dev

[helloworld] Starting in dev mode...
```

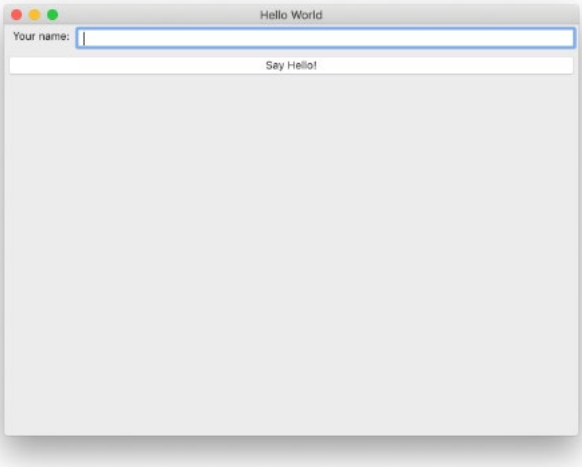
2.2.4. Windows

```
(beeware-venv) C:\...>briefcase dev

[helloworld] Starting in dev mode...
```

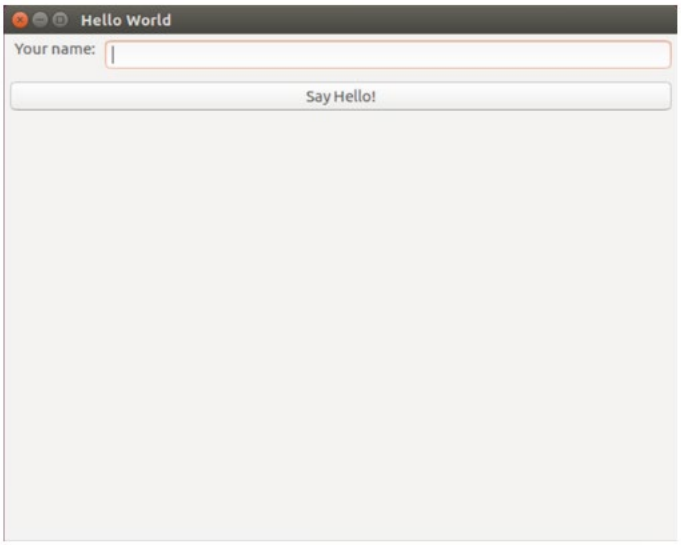
这次你会注意到不再需要安装依赖项。Briefcase 可以检测到应用程序之前已经运行过，为了节省时间，将只运行该应用程序。如果你增加了新的依赖项，可以在运行 `briefcase dev` 时通过传入-d 选项来确保它们被安装。

这将会打开一个 GUI 窗口：

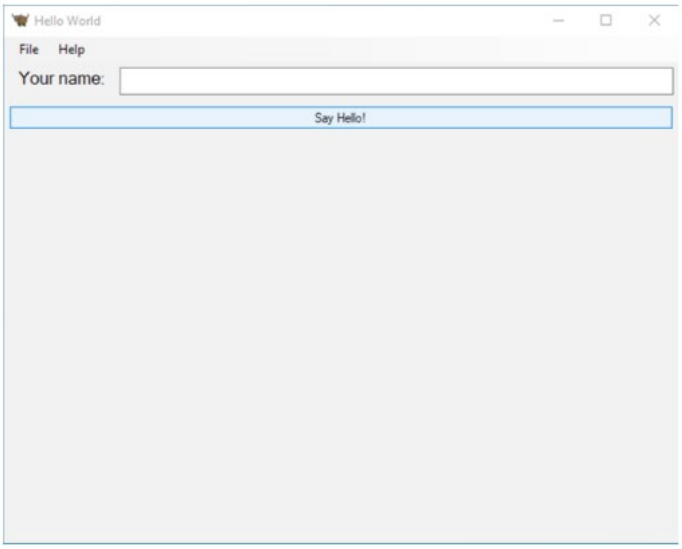
2.2.5. MacOS



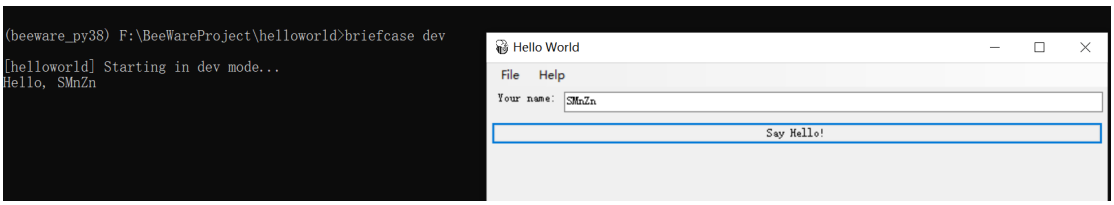
2.2.6. Linux



2.2.7. Windows



如果你在文本框中输入一个名字，摁下 GUI 按钮，你应该会在启动应用程序的控制台中看到输出。



2.3. 下一步

现在我们已经得到了一个 app 并做了一些更有意思的事情。但它还只能在我们的电脑上运行。接下来让我们打包应用程序以便发行。在第三章，我们将把应用程序打包成一个独立的可安装程序，可以发给朋友，消费者或者从应用商店中下载。

3. 第三章-打包以便发布

目前为止，我们已经以开发者模式运行了 app。这使得本地运行 app 很容易-但我们真正想做的是把应用程序给其他用户。

然而，我们并不想教使用者如何安装 Python，创建虚拟环境，克隆一个 Git 库，并且以开发者模式运行 Briefcase。我们只想给他们一个安装包，并使这个 app 工作起来。

Briefcase 可以打包应用程序并发布。

3.1. 创建应用程序脚手架

因为这是第一次打包 app，需要创建一些配置和脚手架以支持打包过程。在 helloworld 目录下，运行：

3.1.1. MacOS

```
(beeware-venv) $ briefcase create

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-macOS-app-template.git
...
[helloworld] Installing support package...
...
[helloworld] Installing dependencies...
...
    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
    ...
```

```
    [helloworld] Created macOS/Hello World
```

3.1.2. Linux

```
(beeware-venv) $ briefcase create

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-linux-appImage-template.git
...
[helloworld] Installing support package...
...
[helloworld] Installing dependencies...
[helloworld] Entering Docker context...

    [helloworld] Building Docker container image...
    ...
    [helloworld] Leaving Docker context.

    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
    ...
    [helloworld] Created linux/Hello World
```

3.1.3. 注意

第一运行这个命令肯需要一些时间，因为 Briefcase 需要准备一个能用于构建 AppImage 二进制文件的 Ubuntu 16.04 Docker 镜像。这涉及到下载大量系统包。之后运行，Docker 镜像可以被重复使用。

3.1.4. Windows

```
(beeware-venv) C:\...>briefcase create

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-windows-msi-template.git
...
[helloworld] Installing support package...
...
```

```
[helloworld] Installing dependencies...
...
    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
    ...
    [helloworld] Created windows\msi\Hello World
```

你可能在终端上看到了一页页内容滑过……所以到底发生了什么呢？Briefcase 做了如下事情：

① 生成一个应用程序模板。搭建一个本地安装包需要大量文件和配置，超出实际应用程序的代码。这个额外的搭建对同一个平台上的每个应用程序是一样的，除了正在构建的实际应用程序的名称-因此 Briefcase 为它支持的每个平台提供了一个应用程序模板。这个步骤推出模板，替换应用程序的名称，包 ID，和配置文件的其他属性，以支持正在构建的平台。

如果对 Briefcase 提供的模板不满意，你可以提供自己的模板。然而，在你已经对使用 Briefcase 的默认模板有一定经验时，你可能不想这么做。

② 下载和安装一个支持包。Briefcase 的打包方法可以说是最简单的方法-它提供一个完整的，独立的 Python 解释器作为构建应用程序的一部分。这略微降低了空间效率-如果 Briefcase 有 5 个应用程序需要打包，就需要复制 5 次 Python 解释器。然而，这种方法保证每个应用程序完全独立，使用已知可使 app 工作起来的特定 Python 版本。

同样，Briefcase 为每个平台提供一个默认的支持包，如果你想，你也可以提供自己的支持包，并把这些包作为构建过程的一部分。如果您在 Python 解释器中有需要启用的特定选项，或者如果您想从运行时不需要的标准库中剥离模块，那么您可能希望这样做。

Briefcase 维护支持包的本地缓存，因此一旦你下载了特定版本的支持包，未来构建时将使用缓存副本。

③ 安装 app 依赖项。app 可指定任何运行时所需的第三方模块。这些都提供使用 pip 安装到 app 安装包中。

④ 安装 app 代码。你的应用程序有自己的代码和资源（例如，运行时所需的镜像）这些文件都会被复制到安装包中。

⑤ 安装 app 所需资源。最后添加任何安装包所需的额外资源。包括需要附加到最终应用中的图标和启动画面图像。

当这些完成时，你将在工程目录下看到一个对应于你的平台（macOS, linux, o 或者 windows）的目录，其中包含附加文件。这是应用程序对于特定平台的打包配置。

3.2. 搭建应用程序

现在可以编译 app。这个步骤执行应用程序在目标平台上可执行所必须的任何二进制编译。

3.2.1. MacOS

```
(beeware-venv) $ briefcase build

[helloworld] Built macOS/Hello World/Hello World.app
```

在 MacOS 上，build 命令不需要做任何事。.app 文件夹是 macos 自带的布局；只要文件夹有.app 扩展，遵循一些内部布局规则，在已知位置提供一些元数据，文件夹就会作为 app 出现在操作系统中。

3.2.2. Linux

```
(beeware-venv) $ briefcase build

[helloworld] Building AppImage...
...
[helloworld] Built linux/Hello World-x86_64-0.0.1.AppImage
```

当这一步完成时，linux 文件夹下会包含一个名为 `Hello World-x86_64-0.0.1.AppImage`。这个文件是可执行的；可以从 shell 中运行，或者在文件资源管理器中双击它。你也可以把它发给其他任何 Linux 用户，只要他们有 Linux2016 年以后的发行版，就可以以同样的方式运行。

3.2.3. Windows

```
(beeware-venv) C:\...>briefcase build

[helloworld] Built windows\msi\Hello World
```

在 Windows 上，在这一步不需要做任何事。发布的二进制文件在 Windows

上是一个已知入口点的文件。安装程序(当它最终创建时)将对如何启动应用程序的细节进行编码，并安装一个开始菜单项来调用应用程序。

3.3. 运行 app

你可以使用 Briefcase 运行 app。

3.3.1. MacOS

```
 (beeware-venv) $ briefcase run

[helloworld] Starting app...

(beeware-venv) $
```

3.3.2. Linux

```
(beeware-venv) $ briefcase run

[helloworld] Starting app...

(beeware-venv) $
```

3.3.3. Windows

```
(beeware-venv) C:\...>briefcase run

[helloworld] Starting app...

(beeware-venv) C:\...>
```

使用 build 命令输出，将会开始运行你的本地 app。

你应该会注意到我们之前能在控制台看到输出，但现在看不到任何东西了。这是因为我们运行的是一个独立的、打包好的 app，没有任何可见的控制台可以输出。

你也可能注意到 app 运行时有一些小差异。例如，操作系统中展示的图标和名字与开发者模式下运行时你看到的略微不同。这也是因为你运行的打包好的 app，而不是在运行 Python 代码。从操作系统的角度来看你正在运行一个 app，而不是一个 Python 程序，这也反映了 app 时如何产生的。

如果你使用 macOS，也会注意到和我们早期在控制台看到的输出有些不同。这是因为打包好的 app 将控制台输出写入系统日志。当你运行打包好的 app，看到的是过滤版本的系统日志，而不是控制台的原原始输出，因此，可以看到更多系统日志细节（如时间戳和信息源）。当关闭 app 时，系统日志会持续运行，尽管没有展示什么日志信息。你可以摁 Ctrl+C 停止展示系统日志。

3.4. 搭建安装包

现在可以使用 package 命令打包 app 以进行分发。Package 命令做的事将搭建好的过程转成最终可以分发的产品的任何编译工作。取决于平台，可能涉及编译安装程序，执行代码签名或执行其他预发布任务。

3.4.1. MacOS

```
(beeware-venv) $ briefcase package --no-sign

[helloworld] Building DMG...
...
[helloworld] Created macOS/Hello World-0.0.1.dmg
```

MacOS 系统将会包含一个名为 `Hello World-0.0.1.dmg` 的文件。如果你查到到这个文件，双击它的图标，就可以挂载 DMG，给你一个 Hello World app 的副本，并连接到应用程序文件夹以便安装。将 app 文件拖到 Applications 中，就已经安装了应用程序。给朋友发送 DMG 文件，他们就能以同样的方式安装。

在这个例子中，我们使用了 `--no-sign` 选项，这意味着我们决定不签约 app。这么做是为了使例程简单。设置代码签名身份有点麻烦，只有在你打算将应用程序分发给其他人时才绝对需要它们。如果我们发布一个真正的 app，需要去掉 `-no-sign` 标志。

当你准备发行一个真正的 app 时，查看 Briefcase 指南。Setting up a macOS code signing identity

3.4.2. Linux

```
(beeware-venv) $ briefcase package

```

```
3. [helloworld] Building AppImage...
4. ...
5. [helloworld] Created linux/Hello World-x86_64-0.0.1.AppImage.
```

在 Linux 系统上，这一步没做什么，build 命令创建的 APPImage 可执行，不需要其他额外处理。

3.4.3. Windows

```
1. (beeware-venv) C:\...>briefcase package
2.
3. [helloworld] Building MSI...
4. ...
5. [helloworld] Packaged windows\Hello_World-0.0.1.msi
```

一旦到这步，windows 文件夹会包含一个名为 `Hello_World-0.0.1.msi` 的文件。如果双击这个安装包运行它，你应该会看到是熟悉的 windows 安装流程。一旦安装完成，就会在开始菜单看到“Hello World”。

3.5. 下一步

现在，我们已经将应用程序打包，准备在桌面平台上发布。但是，当我们需要更新应用程序中的代码时，会发生什么呢?我们如何将这些更新放到打包的应用程序中?请参阅第 4 章来找出答案……

4. 第四章-更新 app

在上一章中，我们将 app 打包为本地 app。如果您处理的是一个真实的 app，那工作到此并没有结束，你还将要测试、发现问题并做一些改变。即使你的 app 很完美，最终也可能希望发布改进后的版本 2.

因此，当你的代码发生改变时，怎么升级已经安装的 app 呢？

4.1. 更新 app 代码

目前当你摁下按钮时我们的 app 会向控制台打印输出。然后 GUI 应用程序不应该使用控制台作为输出。需要使用对话框和用户交流。

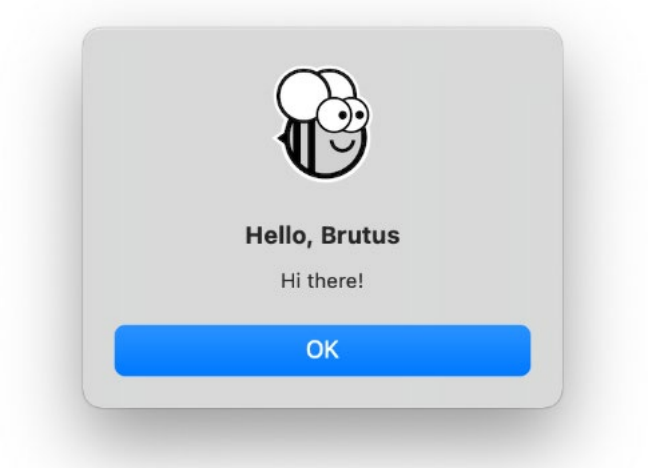
让我们添加一个 say hello 的对话盒子，而不是写入控制台。修改 say_hello 回调函数如下：

```
def say_hello(self, widget):
    self.main_window.info_dialog(
        'Hello, {}'.format(self.name_input.value),
        'Hi there!'
    )
```

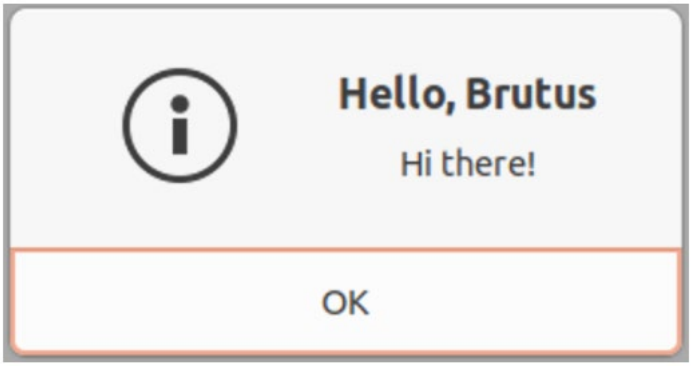
这将指示 Toga 在摁下按钮时打开一个模态对话框。

如果你执行 `briefcase dev`，输入一个名字，摁下按钮，将会看到一个新的 对话框。

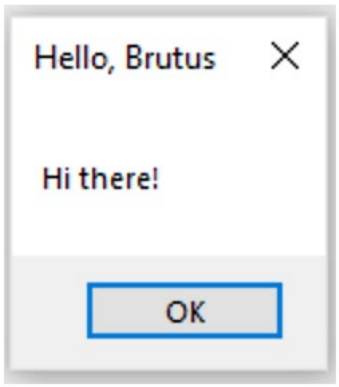
4.1.1. MacOS



4.1.2. Linux



4.1.3. Windows



然而，如果你执行 `briefcase run`，对话框不会出现。

这是为什么呢？是这样的，`briefcase run` 通过在适当的地方运行代码来进行操作—它试图为你的代码生成尽可能真实的运行环境，但它不提供或使用任何平台基础设施来将你的代码包装为应用程序。打包应用程序的过程中，有一部分涉及到将你的代码复制到应用程序包中—此时，你的应用程序仍然保留着旧的代码。

因此，我们需要告诉 Briefcase 运行新版本的代码。可以通过删除旧平台上的目录并从头开始。然而，Briefcase 提供了一种更简单的方法-可以为现存的应用程序包更新代码。

4.1.4. MacOS

```
(beeware-venv) $ briefcase update

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Application updated.
```

4.1.5. Linux

```
(beeware-venv) $ briefcase update

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Application updated.
```

4.1.6. Windows

```
(beeware-venv) C:\...>briefcase update

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Application updated.
```

如果 Briefcase 找不到脚手架模板，将会自动调用 create 命令生成新模板。

现在我们已经更新了安装包代码，可以使用 `briefcase build` 命令重新编译 app，`briefcase run` 命令运行升级后的 app，使用 `briefcase package` 命令重新打包 app 以便分发。

macOS 的用户记住在第三章中提到的使用 `briefcase package` 命令时带上 `-no-sign` 标志避免设置代码签名标识的复杂工作并使教程尽可能简单。

4.2. 一步更新和运行

如果你正在快速迭代更新代码，你可能想更改代码，更新 app，迅速重新运行你的 app。Briefcase 有一个支持这种使用模式的快捷方式-使用 run 命令时加上 -u

让我们尝试一下另一个改变。你可能注意到如果你不在文本框中输入一个名字，对话框就只会说“Hello”。让我们修改 `say_hello` 函数去处理接种极端情况。

```
def say_hello(self, widget):
    if self.name_input.value:
        name = self.name_input.value
    else:
        name = 'stranger'

    self.main_window.info_dialog(
        'Hello, {}'.format(name),
        'Hi there!'
        )
```

在开发者模式下运行 app（使用 `briefcase dev` 命令）确保新的逻辑生效；然后使用一条命令更新，搭建，运行 app

4.2.1. MacOS

```
(beeware-venv) $ briefcase run -u

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Application updated.

[helloworld] Starting app...
```

4.2.2. Linux

```
(beeware-venv) $ briefcase run -u

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Application updated.

[helloworld] Building AppImage...
...
    [helloworld] Created linux/Hello World-x86_64-
0.0.1.AppImage.

    [helloworld] Starting app...
```

4.2.3. Windows

```
(beeware-venv) C:\...>briefcase run -u

[helloworld] Updating application code...
Installing src/helloworld...

[helloworld] Application updated.

[helloworld] Starting app...
```

只有在测试应用程序如何作为本机二进制文件运行，或寻找应用程序以打包形式运行才会出现的 bug 时，才需要这样做。对大多数日常开发，`briefcase dev` 命令更便捷。

打包命令只接受-u 参数，因此如果你修改了 app 代码并想快速打包，可以使用 `briefcase package -u` 命令。

4.3. 下一步

现在，我们已经将应用程序打包以便在桌面平台上发布，并且能够更新 app 中的代码。

但是移动设备呢？在第五章，我们将把 app 转变成移动 app，并且将之部署到设备模拟器或者手机上。

5. 第五章-使之成为移动 app

到目前为止，我们已经在桌面运行并测试了 app。然而，BeeWare 也支持移动平台-因此我们的 app 也可以安装到移动设备上。

5.1. iOS

iOS 应用程序只能在 macOS 上编译。需要使用 Xcode，已经在第 0 章中安装过了。

当你安装好 Xcode，就可以为 iOS 搭建 app 了。

现在，将我们的 app 部署在 iOS 系统上。

将 app 部署到 iOS 上的过程和部署到桌面的过程很相似。首先，使用 `create` 命令-但这次，需要指定创建 iOS app。

```
(beeware-venv) $ briefcase create iOS

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-iOS-Xcode-template.git
...
[helloworld] Installing support package...
...
[helloworld] Installing dependencies...
...
    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
```

```
    ...
    [helloworld] Created iOS/Hello World
```

完成之后，工程下会有一个 ios 目录。这个目录包含一个 Hello World 文件，文件中包含一个 Xcode 项目，同时可以支持 app 所需的库和应用程序代码。

你可以使用 Briefcase 中的 build 命令编译 app。系统会提示你 选择需要编译的设备；如果你安装了多个 iOS 版本的调试器，系统也会询问你想要编译哪个版本的 iOS。你看到的选项和此输出中展示的选项可能不同；就我们的目的而言，选择哪个版本的模拟器并不重要。

```
(beeware-venv) $ briefcase build iOS

Select iOS version:

  1) 10.3
  2) 13.3

> 2

    Select simulator device:

      1) iPad (7th generation)
      2) iPad Air (3rd generation)
      3) iPad Pro (11-inch)
      4) iPad Pro (12.9-inch) (3rd generation)
      5) iPad Pro (9.7-inch)
      6) iPhone 11
      7) iPhone 11 Pro
      8) iPhone 11 Pro Max
      9) iPhone 8
      10) iPhone 8 Plus

    >  6

    Targeting an iPhone 11 running iOS 13.3 (device UDID 4768AA69-497B-4B37-BD0C-3961756C38AC)

    [hello-world] Building XCode project...
    ...
    Build succeeded.

    [hello-world] Built iOS/Hello World/build/Debug-iphonesimulator/Hello World.app
```

现在准备运行 app。通过 `briefcase run iOS` 命令运行 app。如果使用同样的方法运行 Briefcase，会被再次询问目标设备。如果你已经知道这个设备是可选的，可通过添加-d 选项告诉 Briefcase 使用这个调试器。当你搭建 app 的时候使用你选择的设备的名字，然后运行。

```
1. $ briefcase run iOS -d "iPhone 11"
```

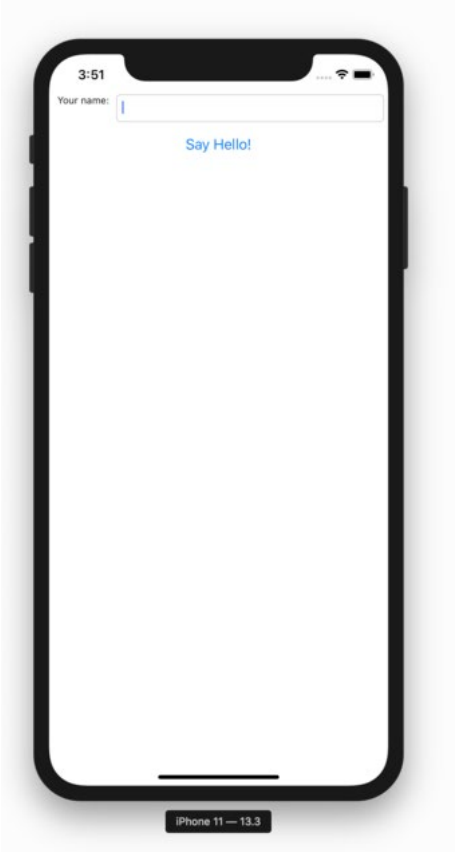
如果你有多个 iPhone 11 调试器，Briefcase 会选择最高版本的 iOS；如果你想选择特定版本的 iOS，告诉 Briefcase 使用特定版本：

```
1. $ briefcase run iOS -d "iPhone 11::iOS 13.3"
```

或者，可以使用特定设备的 UDID 名称：

```
1. $ briefcase run iOS -d 4768AA69-497B-4B37-BD0C-3961756C38AC
```

这将会启动 iOS 调试器，安装 app，然后开始运行。你会看到调试器开始工作，最终打开 iOS app。



5.2. 安卓

现在，将我们的 app 部署在安卓系统上。

将 app 部署到安卓系统上的过程和部署到桌面的过程很相似。Briefcase 处理安卓系统的依赖项，包括安卓 SDK，安卓模拟器和 java 编译器。

5.2.1. 创建安卓 app 并编译

首先运行 create 命令下载一个安卓 app 模板并添加你的 Python 代码进去。

5.2.1.1. MacOS

```
(beeware-venv) $ briefcase create android

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git
...
[helloworld] Installing support package...
...
[helloworld] Installing dependencies...
...
    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
    ...
    [helloworld] Application created.
```

5.2.1.2. Linux

```
(beeware-venv) $ briefcase create android

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git
...
[helloworld] Installing support package...
...
[helloworld] Installing dependencies...
...
    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
    ...
    [helloworld] Application created.
```

5.2.1.3. Windows

```
(beeware-venv) C:\...>briefcase create android

[helloworld] Generating application template...
Using app template: https://github.com/beeware/briefcase-android-gradle-template.git
...
[helloworld] Installing support package...
...
[helloworld] Installing dependencies...
...
    [helloworld] Installing application code...
    ...
    [helloworld] Installing application resources...
    ...
    [helloworld] Created android\gradle\Hello World.
```

当你第一次使用 `briefcase create android` 命令时，Briefcase 下载 Java JDK 和安卓 SDK。文件大小和下载时间相关联；可能要一会（10 分钟或者更长，取决于你的网速）。下载完成之后，系统会提示你接受谷歌的安卓 SDK 协议。

完成之后，我们的工程下会有一个 android 目录。这个目录包含一个 Hello World 文件，它将包含一个带有 Gradle 构建配置的 Android 项目。项目中包含 app 代码，和一个含有 Python 编译器的支持包。

我们可以使用 Briefcase 的 build 命令将这些编译成一个安卓 APK 文件。

5.2.1.4. MacOS

```
(beeware-venv) $ briefcase build android
[helloworld] Building Android APK...
Starting a Gradle Daemon
...
BUILD SUCCESSFUL in 1m 1s
28 actionable tasks: 17 executed, 11 up-to-date
[helloworld] Built android/Hello World/app/build/outputs/apk/debug/app-debug.apk
```

5.2.1.5. Linux

```
(beeware-venv) $ briefcase build android
[helloworld] Building Android APK...
```

```
Starting a Gradle Daemon
...
BUILD SUCCESSFUL in 1m 1s
28 actionable tasks: 17 executed, 11 up-to-date
[helloworld] Built android/Hello World/app/build/outputs/apk/debug/app-debug.apk
```

5.2.1.6. Windows

```
(beeware-venv) C:\...>briefcase build android
[helloworld] Building Android APK...
Starting a Gradle Daemon
...
BUILD SUCCESSFUL in 1m 1s
28 actionable tasks: 17 executed, 11 up-to-date
[helloworld] Built android\Hello World\app\build\outputs\apk\debug\app-debug.apk
```

5.2.1.7. Gradle 可能停止工作

在使用 `briefcase build android` 命令这一步，Gradle（安卓平台的搭建工具）将会打印 `CONFIGURING: 100%`，但看起来什么事也没做。别担心，它并没有停止工作-它在下载更多安卓 SDK 组件。这可能需要十分钟或更长时间都取决于网速。当你第一次使用 build 命令时才产生这种迟滞，当你下次搭建的时候，将会使用之前的缓存版本。

5.2.2. 在虚拟设备给上运行 app

现在准备运行 app。可以使用 Briefcase 的 run 命令在安卓设备上运行 app。让我们在安卓模拟器上运行一下。

使用 `briefcase run android` 命令运行 app。之后，系统会提示你可运行 app 的设备清单。最后一个选项始终是创建一个新的安卓模拟器。

5.2.2.1. MacOS

```
(beeware-venv) $ briefcase run android

Select device:
```

```
4.
5.   1) Create a new Android emulator
6.
7. >
```

5.2.2.2. Linux

```
1. (beeware-venv) $ briefcase run android
2.
3. Select device:
4.
5.   1) Create a new Android emulator
6.
7. >
8.
```

5.2.2.3. Windows

```
1. (beeware-venv) C:\...>briefcase run android
2.
3. Select device:
4.
5.   1) Create a new Android emulator
6.
7. >
```

现在可以选择想使用的设备。选择“创建一个新的安卓模拟器”选项，接受设备名称的默认选项（beePhone）

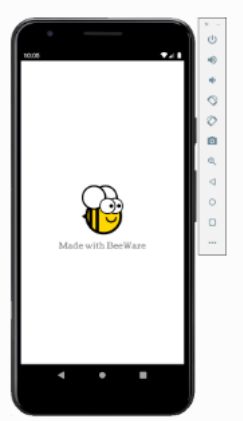
Briefcase 的 run 命令将会自动启动虚拟设备。设备启动后，你将看到安卓 logo。

虚拟安卓设备正在启动

设备启动完成之后，Briefcase 会在设备安装 app。你会看到一个启动界面。

虚拟安卓设备启动完成界面

App 启动后你将看到一个启动画面。



App 启动画面

5.2.3. 模拟器无法启动

安卓模拟器是一种相当复杂的软件，依赖大量硬件和操作系统特性-旧一点的电脑可能不具有或不支持这些特性。如果在启动安卓模拟器时遇到任何困难，咨询安卓开发者文档的要求和建议部分。

第一次启动 app 时，需要在设备上解压缩。这需要花一点时间。一旦解包完成之后，你将会看到我们 app 的安卓版本。

App 完全启动后

如果你的 app 没有启动，当你运行 `briefcase run` 命令时，你可能需要检查一下终端， 找到错误信息。

未来，如果你想不使用菜单在设备上运行 app，你可以模拟器名字告诉 Briefcase，直接使用 `briefcase run android -d @beePhone` 命令在虚拟设备上运行。

5.2.4. 在实体设备上运行 app

如果你有一部安卓手机或平板，可以通过 usb 线把它们和电脑连起来，然后使用 Briefcase 连接到实体设备。

安装运行安卓 app 需要你在将设备用于开发之前做好准备。你需要在设备上修改两个选项：

启用开发者选项

启用 USB 调试

关于如何修改的详细细节可以在安卓开发者文档中找到。

完成上述步骤之后，当你运行 `briefcase run android` 时，你的设备就会出现在可选设备清单中。

5.2.4.1. MacOS

```
(beeware-venv) $ briefcase run android

Select device:

  1) Pixel 3a (94ZZY0LNE8)
  2) @beePhone (emulator)
```

```
  3) Create a new Android emulator

>

```

5.2.4.2. Linux

```
(beeware-venv) $ briefcase run android

Select device:

  1) Pixel 3a (94ZZY0LNE8)
  2) @beePhone (emulator)
  3) Create a new Android emulator

>
```

5.2.4.3. Windows

```
(beeware-venv) C:\...>briefcase run android

Select device:

  1) Pixel 3a (94ZZY0LNE8)
  2) @beePhone (emulator)
  3) Create a new Android emulator

>
```

这里我们可以看到一个新的设备，序列号在部署清单中-在本例中是 Pixel 3a。未来，如果你不想使用菜单在这个设备上运行，可以把序列号提供给 Briefcase（在本例中，使用 `briefcase run android -d 94ZZY0LNE8` 命令）。这将不需要任何提示直接运行设备。

我的设备没有出现

如果你的设备根本没有出现在清单中，要么你没有启动 USB 调试，要么设备根本没插进去。

如果你的设备出现了，但显示为“未知设备（不是未授权开发）”，开发者模式没正确启用。重新启用开发者模式，然后重新运行 `briefcase run android` 命令。

5.3. 下一步

现在我们已经将 app 部署到手机上了！然而，这个 app 太简单了，没有引用任何第三方库。我们能不能从 Python 包中引用一些库到 app 中呢？到第六章中去发现吧……

6. 第六章-开始使用第三方库

到目前为止，搭建的 app 只使用了我们自己的代码，加上 BeeWare 提供的代码。然而，实际开发 app，可能想使用从 Python 包指引（PyPI）中下载的第三方库。

让我们修改 app 使之包含第三方库。

6.1. 访问 API

app 需要执行的一个常见任务是在 web API 上请求检索数据，并将数据显示给用户。这是一个示例 app，因此我们不需要和真的 API 打交道，使用{JSON} Placeholder API 作为数据源。

{JSON} Placeholder API 有许多可以用作测试数据的“假”API 端口。其中一个 API 是/posts/端口，该端口返回假的日志戳。如果你在浏览器中打开 `https://jsonplaceholder.typicode.com/posts/42`，你将会得到一个描述单戳的 JSON-其中包含一篇内容是一些拉丁文，ID 为 42 的日志戳。

Python 标准库包含你需要从 API 获取的所有工具。然而，内置的 API 很低效。它们可以很好的实现 HTTP 协议的—但是需要用户管理许多低级细节，比如 URL 重定向、会话、身份验证和有效负载编码。 作为一个“普通浏览器用户”，您可能习惯于将这些细节视为理所当然，因为浏览器会为您管理这些细节。

因此，人们开发了包装内置 API 的第三方库，并提供更简单的，更符合日常浏览器体验 API。我们将使用这些库中的一个来访问{JSON}占位符 API-一个名为 httpx 的库。

让我们向 app 中添加一个 httpx API 调用。在 app.py 文件的顶部导入 httpx。

```
1. import httpx
```

修改回调函数 say_hello()：

```
1. def say_hello(self, widget):
```

```
    if self.name_input.value:
        name = self.name_input.value
    else:
        name = 'stranger'

    with httpx.Client() as client:
        response = client.get("https://jsonplaceholder.typicode.com/posts/42")

        payload = response.json()

        self.main_window.info_dialog(
            "Hello, {}".format(name),
            payload["body"],
        )
```

当调用 say_hello 回调函数的时候将会：

向 JSON placeholder API 发出获取 42 戳的请求；

将响应解码为 JSON；

提取戳正文并将戳正文作为对话框文本。

让我们以开发者模式运行升级后的 app，检查修改是否生效。

6.1.1. MacOS

```
(beeware-venv) $ briefcase dev
Traceback (most recent call last):
File ".../venv/bin/briefcase", line 5, in <module>
    from briefcase.__main__ import main
File ".../venv/lib/python3.9/site-packages/briefcase/__main__.py", line 3, in <module>
    from .cmdline import parse_cmdline
File ".../venv/lib/python3.9/site-packages/briefcase/cmdline.py", line 6, in <module>
    from briefcase.commands import DevCommand, NewCommand, UpgradeCommand
File ".../venv/lib/python3.9/site-packages/briefcase/commands/__init__.py", line 1, in <module>
        from .build import BuildCommand  # noqa
    File ".../venv/lib/python3.9/site-packages/briefcase/commands/build.py", line 5, in <module>
        from .base import BaseCommand, full_options
    File ".../venv/lib/python3.9/site-packages/briefcase/commands/base.py", line 14, in <module>
```

```
14.         import httpx
15.     ModuleNotFoundError: No module named 'httpx'
```

6.1.2. Linux

```
1. (beeware-venv) $ briefcase dev
2. Traceback (most recent call last):
3. File ".../venv/bin/briefcase", line 5, in <module>
4.     from briefcase.__main__ import main
5. File ".../venv/lib/python3.9/site-
packages/briefcase/__main__.py", line 3, in <module>
6.     from .cmdline import parse_cmdline
7. File ".../venv/lib/python3.9/site-
packages/briefcase/cmdline.py", line 6, in <module>
8.     from briefcase.commands import DevCommand, NewCommand, Upgrade
Command
9. File ".../venv/lib/python3.9/site-
packages/briefcase/commands/__init__.py", line 1, in <module>
10.         from .build import BuildCommand  # noqa
11.     File ".../venv/lib/python3.9/site-
packages/briefcase/commands/build.py", line 5, in <module>
12.         from .base import BaseCommand, full_options
13.     File ".../venv/lib/python3.9/site-
packages/briefcase/commands/base.py", line 14, in <module>
14.         import httpx
15.     ModuleNotFoundError: No module named 'httpx'
```

6.1.3. Windows

```
1. (beeware-venv)C:\...>briefcase dev
2. Traceback (most recent call last):
3. File "...\venv\bin\briefcase", line 5, in <module>
4.     from briefcase.__main__ import main
5. File "...\venv\lib\python3.9\site-
packages\briefcase\__main__.py", line 3, in <module>
6.     from .cmdline import parse_cmdline
7. File "...\venv\lib\python3.9\site-
packages\briefcase\cmdline.py", line 6, in <module>
8.     from briefcase.commands import DevCommand, NewCommand, Upgrade
Command
9. File "...\venv\lib\python3.9\site-
packages\briefcase\commands\__init__.py", line 1, in <module>
10.         from .build import BuildCommand  # noqa
```

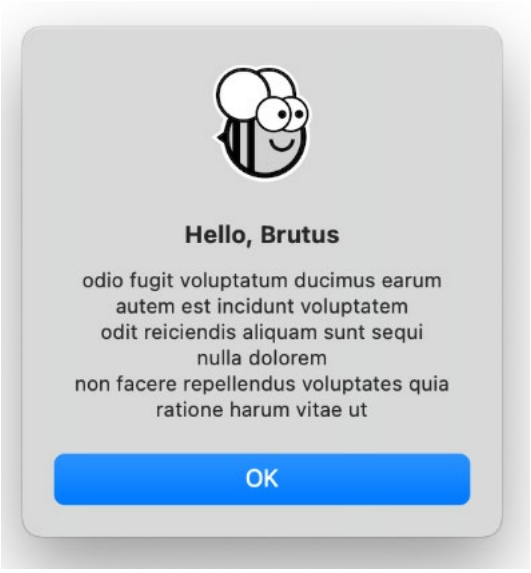
```
11.     File "...\venv\lib\python3.9\site-
packages\briefcase\commands\build.py", line 5, in <module>
12.         from .base import BaseCommand, full_options
13.     File "...\venv\lib\python3.9\site-
packages\briefcase\commands\base.py", line 14, in <module>
14.         import httpx
15.     ModuleNotFoundError: No module named 'httpx'
```

发生什么了？我们已经向代码中添加了 httpx，但是并没有把它添加到虚拟开发环境中。我们可以通过 pip 命令安装 httpx，然后重新运行 `briefcase dev`。

6.1.4. MacOS

```
1. (beeware-venv) $ python -m pip install httpx
2. (beeware-venv) $ briefcase dev
```

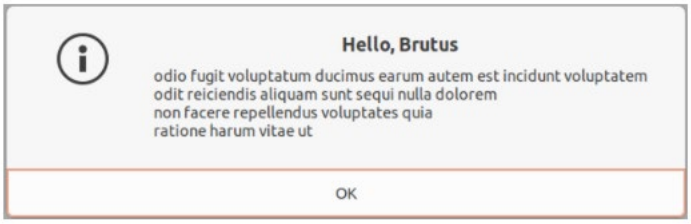
当你输入一个名字并摁下按钮时会看到这样一个对话框：



6.1.5. Linux

```
1. (beeware-venv) $ python -m pip install httpx
2. (beeware-venv) $ briefcase dev
```

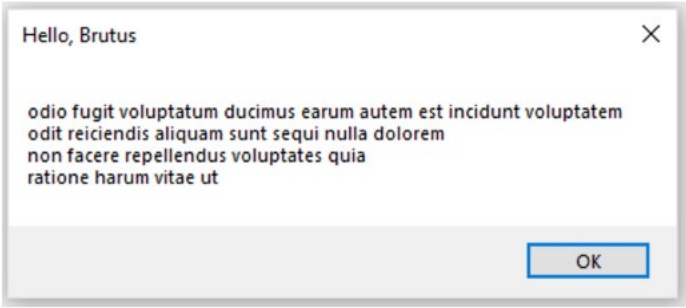
当你输入一个名字并摁下按钮时会看到这样一个对话框：



6.1.6. Windows

```
(beeware-venv)C:\...>python -m pip install httpx
(beeware-venv)C:\...>briefcase dev
```

当你输入一个名字并摁下按钮时会看到这样一个对话框：



我们现在可以以开发者模式，使用第三方库，运行 app。

6.2. 运行并更新 app

让我们把升级后的 app 代码打包成独立的 app。因为修改了代码，所以需要按照第四章中的步骤操作：

6.2.1. MacOS

升级打包完成的 app 中的代码：

```
(beeware-venv) $ briefcase update

[hello-world] Updating application code...
Installing src/hello_world...

[hello-world] Application updated.
```

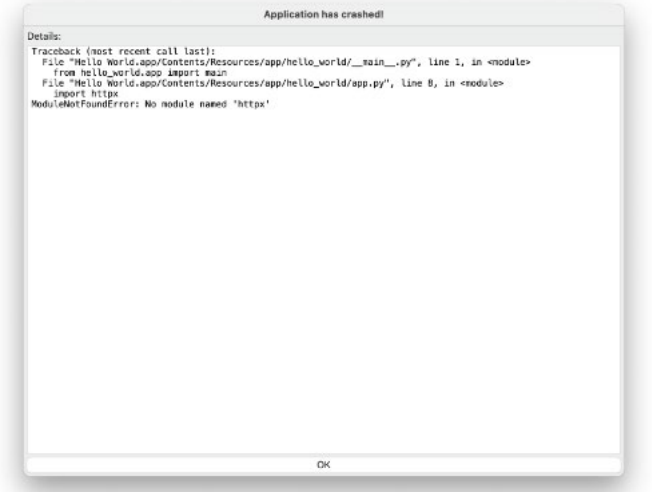
重建 app

```
(beeware-venv) $ briefcase build

[hello-world] Building AppImage...
...
[hello-world] Built linux/Hello_World-0.0.1-x86_64.AppImage
```

最后，运行 app

```
(beeware-venv) $ briefcase run

[hello-world] Starting app...
```

然而，当 app 运行时，将看到一个崩溃对话框：



6.2.2. Linux

升级打包完成的 app 中的代码：

```
(beeware-venv) $ briefcase update

[hello-world] Updating application code...
Installing src/hello_world...

[hello-world] Application updated.
```

重建 app

```
(beeware-venv) $ briefcase build

[hello-world] Building AppImage...
...
[hello-world] Built linux/Hello_World-0.0.1-x86_64.AppImage
```

最后，运行 app

```
(beeware-venv) $ briefcase run

[hello-world] Starting app...

Traceback (most recent call last):
  File "/tmp/.mount_Hello_ifthSH/usr/lib/python3.8/runpy.py", line 194, in _
run_module_as_main
    return _run_code(code, main_globals, None,
  File "/tmp/.mount_Hello_ifthSH/usr/lib/python3.8/runpy.py", line 87, in _r
un_code
```

```
    exec(code, run_globals)
  File "/tmp/.mount_Hello_ifthSH/usr/app/hello_world/__main__.py", line 1, in <module>
    from hello_world.app import main
  File "/tmp/.mount_Hello_ifthSH/usr/app/hello_world/app.py", line 8, in <module>
    import httpx
ModuleNotFoundError: No module named 'httpx'

Unable to start app hello-world.
```

6.2.3. Windows

升级打包完成的 app 中的代码：

```
(beeware-venv)C:\...>briefcase update

[hello-world] Updating application code...
Installing src/hello_world...

[hello-world] Application updated.
```

重建 app

```
(beeware-venv)C:\...>briefcase build

[hello-world] Built windows/msi/Hello World
```

最后，运行 app

```
(beeware-venv)C:\...>briefcase run

[hello-world] Starting app...

Unable to start app hello-world.
```

又一次报错，app 启动失败因为 httpx 已经安装-但是为什么呢？我们已经安装过 httpx 了嘛？

我们安装了，但仅仅是在开发者模式下安装的。开发环境完全是机器本地的环境-只有当你专门激活它的时候才能启动。尽管 Briefcase 有开发者模式，但使用 Briefcase 的主要目的是把代码打包到 app 中以便发送给其他人。

保证其他人的 Python 环境中包含所需的一切东西的唯一方法是搭建完全独立的 Python 环境。这意味着独立安装 Python，并且和其他依赖项完全隔开。这样当使用 `briefcase build` 命令时 Briefcase 就会搭建一个完全独立的 Python 环

境。这也解释了为什么 httpx 没有被安装-它只是被安装在开发环境中，并没有被打包到 app 中。

因此，需要告诉 Briefcase 我们的 app 需要额外依赖项。

6.3. 更新依赖库

在 app 的根目录中有一个名为 `pyproject.toml` 的文件。这个文件包含你第一次运行 `briefcase new` 命令时你提供的所有配置细节。

`pyproject.toml` 文件分好几个章节，其中一个章节描述了 app 的配置：

```
1. [tool.briefcase.app.hello-world]
2. formal_name = "Hello World"
3. description = "A Tutorial app"
4. icon = "src/hello_world/resources/hello-world"
5. sources = ['src/hello_world']
6. requires = []
```

requires 选项描述了 app 依赖项。它是一个字符串清单，指定了你的 app 中包含的库（以及可选的版本）。

修改 reqquire 设置：

```
1. requires = [
2.     "httpx",
3. ]
```

增加这个设置，相当于告诉 Briefcase 当你构建 app 时，运行 `pip install httpx` 命令把 httpx 包安装到 app 中。任何对 pip install 命令合理的输入都可以用在这里-所以你可以指定：

特定库的版本（例如 `"httpx==0.19.0"`）；

库版本的范围（例如 `"httpx>=0.19"`）

克隆仓库的路径（例如 `"git+https://github.com/encode/httpx"`）

或者本地文件路径（然而-警告：如果你把代码发给其他人，他们的电脑上可能不存在这个路径）

继续往下看 `pyproject.toml` 文件，其他章节是造作系统依赖项，如 `[tool.briefcase.app.hello-world.macOS]` 和 `[tool.briefcase.app.hello-`

`world.windows]` 。这些章节也有 require 设置。这些设置允许定义其他平台-指定依赖项-如果你需要一个额外的平台，指定库处理 app 某些请求，你可以在平台特定的 require 章节指定那个库，设置只在该平台有效。你将注意到 toga 库在所有平台特定的 require 章节都被指定-这是因为显示用户交互界面是平台特定的。

在我们的例子中，希望 htppx 库安装到所有平台上，因此使用 app 级的 require 设置。除了 app 级的依赖项将被安装，平台特定的依赖项也将被安装。

6.3.1. Python 只在移动端(目前!)

在桌面平台（macOS，Windows，Linux），任何 pip 能安装的都可以被添加到你的需求中，但是在移动平台，你的选项有一点受限-你只能使用纯 Python 包及包中不能包含二进制模块。

这意味着 `numpy` ， `scikit-learn` ，或者 `cryptography` 可以在桌面 app 使用，不能在移动 app 中使用。这主要是因为移动应用需要针对多个平台编译的二进制模块，这很难设置。

在 Python 中使用二进制模块搭建一个 app 是可能的，但很难-这完全超出了像本教程这样入门级教程的范围。这是我们想要解决的领域-但不是一个轻松的任务。如果你想看 BeeWare 中添加这个功能，请参考支持这个项目并加入。

现在已经告诉 Briefcase 我们所需的额外依赖项，可以再次打包 app。确保你已经将更改保存到 `pyproject.toml` 了，然后重新升级 app-这一次添加-d 标志。这告诉 Briefcase 升级已打包 app 中的依赖项。

6.3.2. MacOS

```
(beeware-venv) $ briefcase update -d

[hello-world] Updating dependencies...
Collecting httpx
  Using cached httpx-0.19.0-py3-none-any.whl (77 kB)
...
Installing collected packages: sniffio, idna, travertino, rfc3986, h11, anyio, toga-core, rubicon-objc, httpcore, charset-normalizer, certifi, toga-cocoa, httpx
Successfully installed anyio-3.3.2 certifi-2021.10.8 charset-normalizer-2.0.6 h11-0.12.0 httpcore-0.13.7 httpx-0.19.0 idna-3.2 rfc3986-1.5.0 rubicon-objc-0.4.1 sniffio-1.2.0 toga-cocoa-0.3.0.dev28 toga-core-0.3.0.dev28 travertino-0.1.3
```

```
9.
10. [hello-world] Updating application code...
11. Installing src/hello_world...
12.
13. [hello-world] Application updated.
```

6.3.3. Linux

```
1.  (beeware-venv) $ briefcase update -d
2.
3.  [hello-world] Updating dependencies...
4.  Collecting httpx
5.    Using cached httpx-0.19.0-py3-none-any.whl (77 kB)
6.  ...
7.  Installing collected packages: sniffio, idna, travertino, rfc3986, h11, anyi
o, toga-core, rubicon-objc, httpcore, charset-normalizer, certifi, toga-
cocoa, httpx
8.  Successfully installed anyio-3.3.2 certifi-2021.10.8 charset-normalizer-
2.0.6 h11-0.12.0 httpcore-0.13.7 httpx-0.19.0 idna-3.2 rfc3986-1.5.0 rubicon-objc-
0.4.1 sniffio-1.2.0 toga-cocoa-0.3.0.dev28 toga-core-0.3.0.dev28 travertino-0.1.3
9.
10. [hello-world] Updating application code...
11. Installing src/hello_world...
12.
13. [hello-world] Application updated.
```

6.3.4. Windows

```
1.  (beeware-venv)C:\...>briefcase update -d
2.
3.  [hello-world] Updating dependencies...
4.  Collecting httpx
5.    Using cached httpx-0.19.0-py3-none-any.whl (77 kB)
6.  ...
7.  Installing collected packages: sniffio, idna, travertino, rfc3986, h11, anyi
o, toga-core, rubicon-objc, httpcore, charset-normalizer, certifi, toga-
cocoa, httpx
8.  Successfully installed anyio-3.3.2 certifi-2021.10.8 charset-normalizer-
2.0.6 h11-0.12.0 httpcore-0.13.7 httpx-0.19.0 idna-3.2 rfc3986-1.5.0 rubicon-objc-
0.4.1 sniffio-1.2.0 toga-cocoa-0.3.0.dev28 toga-core-0.3.0.dev28 travertino-0.1.3
9.
10. [hello-world] Updating application code...
11. Installing src/hello_world...
```

```
12.
13. [hello-world] Application updated.
```

升级完成后，使用 `briefcase build` 和 `briefcase run` 命令，将会看到打包完成的 app，出现了新的对话框。

6.4. 下一步

现在已经完成了一个需要第三方库的 app！然而，你可能注意到，摁下按钮的时候，app 有些迟钝。我们能不能做点什么解决一下呢？到第七章看看吧……

7. 第七章-让 app 运行更流畅

除非你的网速很快，否则你会注意到当你摁下按钮的时候，app 中的 GUI 有点迟滞。这是因为我们提同时在请求网络。当我们的应用发生请求时，继续执行之前需要等带 API 返回响应值。当等待响应时，不允许 app 执行其他操作-因此，app 被锁定。

7.1. GUI 事件循环

要理解为什么会发生这种情况，需要深入挖掘 GUI app 的工作细节。具体内容因平台而异。但无论使用何种平台或 GUI 环境，其高级概念都是相同的。

GUI app 从本质上说是一个单循环：

```
1. while not app.quit_requested():
2.     app.process_events()
3.     app.redraw()
```

这个循环被称为事件循环。这些并不是实际的函数名称—它只是“伪代码”的一个说明。

当你点击按钮时，或拖动滚动条时，或按下键盘 时，都在产生一个“事件”。事件会被放进队列中，当有机会时，app 就会处理事件队列。响应用户事件时触发的代码称为事件处理函数。事件处理函数作为 `process_events()` 调用的一部分被调用。

一旦 app 处理完所有可获取事件，就会重绘 GUI。这将需要考虑事件对应用程序显示的任何改变，包括操作系统正在发生的事情-例如，其他 app 的窗口可能会隐藏或显示我们 app 的部分窗口，我们的 app 的重绘将需要反映窗口当前可

见的部分。

有个重要细节值得注意：当 app 正在处理一个事件，不能重绘也不能处理其他事件。

这意味着包含事件处理函数中的任何逻辑都需要快速完成。完成事件处理函数时的任何延迟都有可能被用户在 GUI 更新时作为迟滞或停止被观察到。如果延迟足够长，操作系统就会作为问题报告-macOS 提示“Beachball”或 Windows 提示“Hourglass”都是操作系统在告诉你，你的 app 处理事件花费太长时间了。

像更新 “label”或“重新计算总输入”这种操作都可以很快完成。然而，有很多操作不可能很快完成。如果正在执行一个复杂的数学计算，或者在一个文件系统中索引所有文件或者执行一个巨大的网络请求，都不可能迅速完成-这些操作本身就很慢。

因此，怎么才能在 GUI app 上执行时间很长的操作呢？

7.2. 异步编程

我们需要告诉 app 在长周期事件处理中期可以暂时将控制权还给事件循环，只要能找到离开的位置。由 app 决定何时将控制权交出，如果 app 周期性的向事件循环释放控制权，我们就可以有一个长周期的事件处理函数并维持 UI 的响应性。

可以通过异步编程实现。异步编程是一种允许编译器同时运行多个函数，在多个正在运行的函数间共享资源的编程方式。

异步函数（也叫多线程）清楚的声明为异步。还需要在内部声明什么时候可以将改内容更改为另一个多线程。

在 Python 中，异步编程通过使用 async 和 await 关键字实现，异步模块在标准库中。Async 关键字允许我们声明该函数是异步多线程的。await 关键字提供了一种何时可以将改内容更改为另一个多线程的方法。异步模块为异步编程提供了一些有用的工具和函数。

7.3. 制作异步教程

为实现异步编程，修改事件处理函数 say_hello():

```
1.  async def say_hello(self, widget):
2.      if self.name_input.value:
3.          name = self.name_input.value
4.      else:
5.          name = 'stranger'
```

```
    6.
    7.      async with httpx.AsyncClient() as client:
    8.          response = await client.get("https://jsonplaceholder.typicode.com/po
sts/42")
    9.
    10.     payload = response.json()
    11.
    12.     self.main_window.info_dialog(
    13.         "Hello, {}".format(name),
    14.         payload["body"],
    15.     )
```

在原始代码的基础上只做了 4 个改变：

① 函数定义为 `async def`，而不仅仅是 `def`。这告诉 Python 该函是异步多线程函数。

② 创建的客户端是异步接收的 `AsyncClient()`，而不是同步接收 `Client()`。这告诉 httpx 应该工作在异步模式而不是同步模式。

③ 用于创建客户端的内容管理器被标记为 async。这告诉 Python 应该在进入和退出内容管理器时释放控制权。

④ 调用 get 时加上 await 关键字。这指示 app 当等待网络响应时，app 可以释放控制权回到事件循环。

Toga 允许你使用周期函数或异步多线程处理事件；Toga 在幕后管理一切，以确保按需要调用或等待处理程序。

如果你保存更改，并重新运行 app（要么在开发者模式下使用 `briefcase dev` 命令要么升级之后重新运行打包好的 app），app 不会有任何明显的改变。然而，当你点击按钮触发对话框时，可能会注意到一些细微的改进：

按钮返回到未点击状态而不是停留在点击状态。

“beachball”/”hourglass”图标不会出现。

等待对话框出现时移动窗口或调整窗口大小，窗口都会重绘。

如果尝试打开 app 菜单，菜单会迅速出现。

7.4. 下一步

现在的 app 即使在等待响应很慢的 API，也可以快速响应其他请求。但他看起来还只是一个例程 app。我们可不可以做一些其他事呢？到第八章看看……

8. 第八章-打造独家 app

8.1. 添加图标

8.2. 下一步