更大的应用 - 多个文件

如果你正在开发一个应用程序或 Web API，很少会将所有的内容都放在一个文件中。

FastAPI 提供了一个方便的工具，可以在保持所有灵活性的同时构建你的应用程序。

!!! info 如果你来自 Flask，那这将相当于 Flask 的 Blueprints。

一个文件结构示例

假设你的文件结构如下：

```
.
├── app
│   ├── __init__.py
│   ├── main.py
│   ├── dependencies.py
│   └── routers
│   │   ├── __init__.py
│   │   ├── items.py
│   │   └── users.py
│   └── internal
│       ├── __init__.py
│       └── admin.py
```

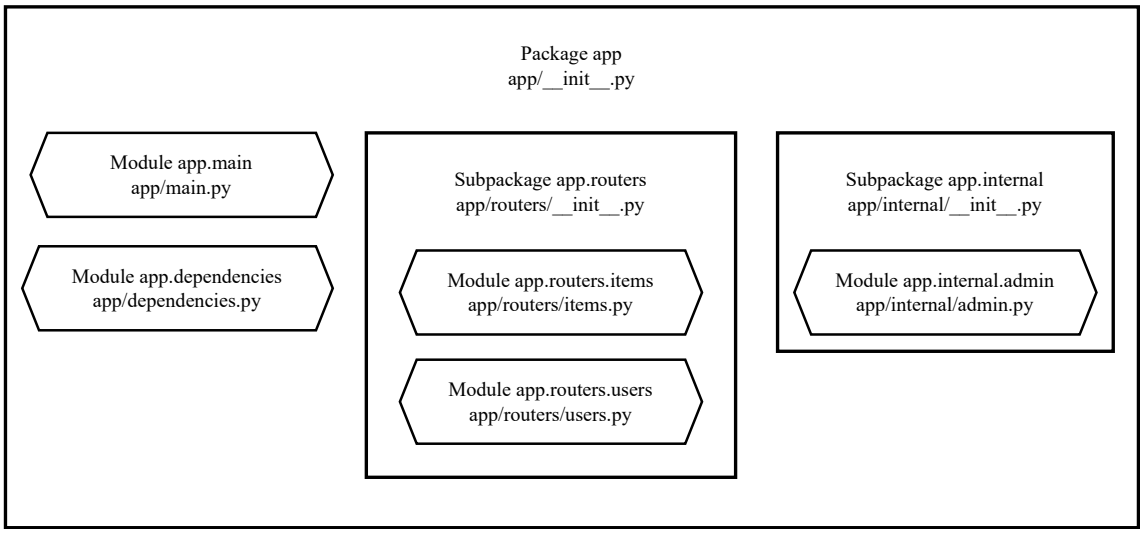
!!! tip 上面有几个 `__init__.py` 文件：每个目录或子目录中都有一个。

这就是能将代码从一个文件导入到另一个文件的原因。

例如，在 `app/main.py` 中，你可以有如下一行：

```
from app.routers import items
```

- `app` 目录包含了所有内容。并且它有一个空文件 `app/__init__.py`，因此它是一个「Python 包」（「Python 模块」的集合）：`app`。
- 它包含一个 `app/main.py` 文件。由于它位于一个 Python 包（一个包含 `__init__.py` 文件的目录）中，因此它是该包的一个「模块」：`app.main`。
- 还有一个 `app/dependencies.py` 文件，就像 `app/main.py` 一样，它是一个「模块」：`app.dependencies`。
- 有一个子目录 `app/routers/` 包含另一个 `__init__.py` 文件，因此它是一个「Python 子包」：`app.routers`。
- 文件 `app/routers/items.py` 位于 `app/routers/` 包中，因此它是一个子模块：`app.routers.items`。
- 同样适用于 `app/routers/users.py`，它是另一个子模块：`app.routers.users`。
- 还有一个子目录 `app/internal/` 包含另一个 `__init__.py` 文件，因此它是又一个「Python 子包」：`app.internal`。
- `app/internal/admin.py` 是另一个子模块：`app.internal.admin`。



带有注释的同一文件结构：

```
.
├── app                  # 「app」是一个 Python 包
│   ├── __init__.py      # 这个文件使「app」成为一个 Python 包
│   ├── main.py          # 「main」模块，例如 import app.main
│   ├── dependencies.py  # 「dependencies」模块，例如 import app.dependencies
│   └── routers          # 「routers」是一个「Python 子包」
│   │   ├── __init__.py  # 使「routers」成为一个「Python 子包」
│   │   ├── items.py     # 「items」子模块，例如 import app.routers.items
│   │   └── users.py     # 「users」子模块，例如 import app.routers.users
│   └── internal         # 「internal」是一个「Python 子包」
│       ├── __init__.py  # 使「internal」成为一个「Python 子包」
│       └── admin.py     # 「admin」子模块，例如 import app.internal.admin
```

`APIRouter`

假设专门用于处理用户逻辑的文件是位于 `/app/routers/users.py` 的子模块。

你希望将与用户相关的*路径操作*与其他代码分开，以使其井井有条。

但它仍然是同一 **FastAPI** 应用程序/web API 的一部分（它是同一「Python 包」的一部分）。

你可以使用 `APIRouter` 为该模块创建*路径操作*。

导入 `APIRouter`

你可以导入它并通过与 `FastAPI` 类相同的方式创建一个「实例」：

```
{!../../../docs_src/bigger_applications/app/routers/users.py!}
```

使用 `APIRouter` 的*路径操作*

然后你可以使用它来声明*路径操作*。

使用方式与 `FastAPI` 类相同：

```
{!../../../docs_src/bigger_applications/app/routers/users.py!}
```

你可以将 `APIRouter` 视为一个「迷你 `FastAPI` 」类。

所有相同的选项都得到支持。

所有相同的 `parameters` 、 `responses` 、 `dependencies` 、 `tags` 等等。

!!! tip 在此示例中，该变量被命名为 `router` ，但你可以根据你的想法自由命名。

我们将在主 `FastAPI` 应用中包含该 `APIRouter` ，但首先，让我们来看看依赖项和另一个 `APIRouter` 。

依赖项

我们了解到我们将需要一些在应用程序的好几个地方所使用的依赖项。

因此，我们将它们放在它们自己的 `dependencies` 模块（ `app/dependencies.py` ）中。

现在我们将使用一个简单的依赖项来读取一个自定义的 `X-Token` 请求首部：

```
{!../../../docs_src/bigger_applications/app/dependencies.py!}
```

!!! tip 我们正在使用虚构的请求首部来简化此示例。

但在实际情况下，使用集成的[安全性实用工具](./security/index.md){.internal-link target=_blank}会得到更好的效果。

其他使用 `APIRouter` 的模块

假设你在位于 `app/routers/items.py` 的模块中还有专门用于处理应用程序中「项目」的端点。

你具有以下*路径操作*：

- `/items/`
- `/items/{item_id}`

这和 `app/routers/users.py` 的结构完全相同。

但是我们想变得更聪明并简化一些代码。

我们知道此模块中的所有*路径操作*都有相同的：

- 路径 `prefix` ： `/items` 。
- `tags` ：（仅有一个 `items` 标签）。
- 额外的 `responses` 。
- `dependencies` ：它们都需要我们创建的 `X-Token` 依赖项。

因此，我们可以将其添加到 `APIRouter` 中，而不是将其添加到每个路径操作中。

```
{!../../../docs_src/bigger_applications/app/routers/items.py!}
```

由于每个*路径操作*的路径都必须以 `/` 开头，例如：

```
@router.get("/{item_id}")
async def read_item(item_id: str):
    ...
```

...前缀不能以 `/` 作为结尾。

因此，本例中的前缀为 `/items`。

我们还可以添加一个 `tags` 列表和额外的 `responses` 列表，这些参数将应用于此路由器中包含的所有*路径操作*。

我们可以添加一个 `dependencies` 列表，这些依赖项将被添加到路由器中的所有*路径操作*中，并将针对向它们发起的每个请求执行/解决。

!!! tip 请注意，和*路径操作装饰器中的依赖项*{.internal-link target=_blank}很类似，没有值会被传递给你的*路径操作函数*。

最终结果是项目相关的路径现在为：

- `/items/`
- `/items/{item_id}`

...如我们所愿。

- 它们将被标记为仅包含单个字符串 `"items"` 的标签列表。
    - 这些「标签」对于自动化交互式文档系统（使用 OpenAPI）特别有用。
- 所有的路径操作都将包含预定义的 `responses`。
- 所有的这些*路径操作*都将在自身之前计算/执行 `dependencies` 列表。
    - 如果你还在一个具体的*路径操作*中声明了依赖项，**它们也会被执行。**
    - 路由器的依赖项最先执行，然后是装饰器中的 `dependencies`{.internal-link target=_blank}，再然后是普通的参数依赖项。
    - 你还可以添加具有 `scopes` 的 `Security` 依赖项{.internal-link target=_blank}。

!!! tip 在 `APIRouter` 中具有 `dependencies` 可以用来，例如，对一整组的*路径操作*要求身份认证。即使这些依赖项并没有分别添加到每个路径操作中。

!!! check `prefix`、`tags`、`responses` 以及 `dependencies` 参数只是（和其他很多情况一样）**FastAPI** 的一个用于帮助你避免代码重复的功能。

导入依赖项

这些代码位于 `app.routers.items` 模块，`app/routers/items.py` 文件中。

我们需要从 `app.dependencies` 模块即 `app/dependencies.py` 文件中获取依赖函数。

因此，我们通过 `..` 对依赖项使用了相对导入：

```
{!../../../docs_src/bigger_applications/app/routers/items.py!}
```

相对导入如何工作

!!! tip 如果你完全了解导入的工作原理，请从下面的下一部分继续。

一个单点 `.`，例如：

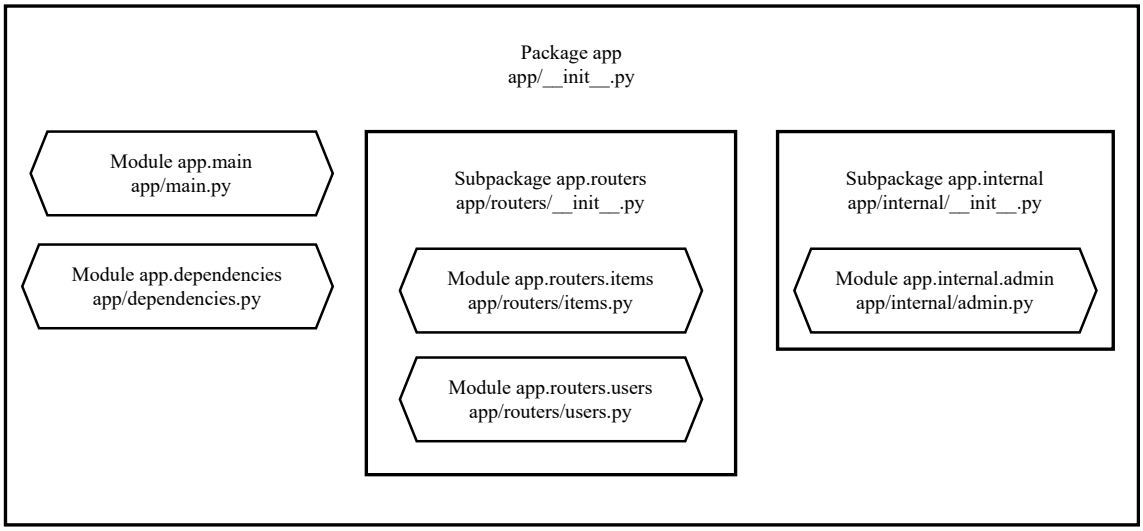
```
from .dependencies import get_token_header
```

表示：

- 从该模块（`app/routers/items.py` 文件）所在的同一个包（`app/routers/` 目录）开始...
- 找到 `dependencies` 模块（一个位于 `app/routers/dependencies.py` 的虚构文件）...
- 然后从中导入函数 `get_token_header`。

但是该文件并不存在，我们的依赖项位于 `app/dependencies.py` 文件中。

请记住我们的程序/文件结构是怎样的：



两个点 `..`，例如：

```
from ..dependencies import get_token_header
```

表示：

- 从该模块（`app/routers/items.py` 文件）所在的同一个包（`app/routers/` 目录）开始...
- 跳转到其父包（`app/` 目录）...
- 在该父包中，找到 `dependencies` 模块（位于 `app/dependencies.py` 的文件）...
- 然后从中导入函数 `get_token_header`。

正常工作了！

同样，如果我们使用了三个点 `...`，例如：

```
from ...dependencies import get_token_header
```

那将意味着：

- 从该模块（`app/routers/items.py` 文件）所在的同一个包（`app/routers/` 目录）开始...

- 跳转到其父包（`app/` 目录）...
- 然后跳转到该包的父包（该父包并不存在，`app` 已经是最顶层的包 😱）...
- 在该父包中，找到 `dependencies` 模块（位于 `app/` 更上一级目录中的 `dependencies.py` 文件）...
- 然后从中导入函数 `get_token_header`。

这将引用 `app/` 的往上一级，带有其自己的 `__init __.py` 等文件的某个包。但是我们并没有这个包。因此，这将在我们的示例中引发错误。🚨

但是现在你知道了它的工作原理，因此无论它们多么复杂，你都可以在自己的应用程序中使用相对导入。😓

添加一些自定义的 `tags`、`responses` 和 `dependencies`

我们不打算在每个*路径操作*中添加前缀 `/items` 或 `tags =["items"]`，因为我们将它们添加到了 `APIRouter` 中。

但是我们仍然可以添加*更多*将会应用于特定的*路径操作*的 `tags`，以及一些特定于该*路径操作*的额外 `responses`：

```
{!../../../docs_src/bigger_applications/app/routers/items.py!}
```

!!! tip 最后的这个路径操作将包含标签的组合：`["items", "custom"]`。

并且在文档中也会有两个响应，一个用于 `404`，一个用于 `403`。

`FastAPI` 主体

现在，让我们来看看位于 `app/main.py` 的模块。

在这里你导入并使用 `FastAPI` 类。

这将是你的应用程序中将所有内容联结在一起的主文件。

并且由于你的大部分逻辑现在都存在于其自己的特定模块中，因此主文件的内容将非常简单。

导入 `FastAPI`

你可以像平常一样导入并创建一个 `FastAPI` 类。

我们甚至可以声明全局依赖项{.internal-link target=_blank}，它会和每个 `APIRouter` 的依赖项组合在一起：

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

导入 `APIRouter`

现在，我们导入具有 `APIRouter` 的其他子模块：

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

由于文件 `app/routers/users.py` 和 `app/routers/items.py` 是同一 Python 包 `app` 一个部分的子模块，因此我们可以使用单个点 `.` 通过「相对导入」来导入它们。

导入是如何工作的

这段代码：

```
from .routers import items, users
```

表示：

- 从该模块（`app/main.py` 文件）所在的同一个包（`app/` 目录）开始...
- 寻找 `routers` 子包（位于 `app/routers/` 的目录）...
- 从该包中，导入子模块 `items`（位于 `app/routers/items.py` 的文件）以及 `users`（位于 `app/routers/users.py` 的文件）...

`items` 模块将具有一个 `router` 变量（`items.router`）。这与我们在 `app/routers/items.py` 文件中创建的变量相同，它是一个 `APIRouter` 对象。

然后我们对 `users` 模块进行相同的操作。

我们也可以像这样导入它们：

```
from app.routers import items, users
```

!!! info 第一个版本是「相对导入」：

```
```Python
from .routers import items, users
```
```

第二个版本是「绝对导入」：

```
```Python
from app.routers import items, users
```
```

要了解有关 `Python` 包和模块的更多信息，请查阅关于 `Modules` 的 `Python` 官方文档。
```

### 避免名称冲突

我们将直接导入 `items` 子模块，而不是仅导入其 `router` 变量。

这是因为我们在 `users` 子模块中也有另一个名为 `router` 的变量。

如果我们一个接一个地导入，例如：

```
from .routers.items import router
from .routers.users import router
```

来自 `users` 的 `router` 将覆盖来自 `items` 中的 `router`，我们将无法同时使用它们。

因此，为了能够在同一个文件中使用它们，我们直接导入子模块：
```

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

包含 `users` 和 `items` 的 `APIRouter`

现在，让我们来包含来自 `users` 和 `items` 子模块的 `router`。

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

!!! info `users.router` 包含了 `app/routers/users.py` 文件中的 `APIRouter`。

```
`items.router` 包含了 `app/routers/items.py` 文件中的 `APIRouter`。
```

使用 `app.include_router()`，我们可以将每个 `APIRouter` 添加到主 `FastAPI` 应用程序中。

它将包含来自该路由器的所有路由作为其一部分。

!!! note "技术细节" 实际上，它将在内部为声明在 `APIRouter` 中的每个*路径操作*创建一个*路径操作*。

```
所以，在幕后，它实际上会像所有的东西都是同一个应用程序一样工作。
```

!!! check 包含路由器时，你不必担心性能问题。

```
这将花费几微秒时间，并且只会在启动时发生。

因此，它不会影响性能。⚡
```

包含一个有自定义 `prefix`、`tags`、`responses` 和 `dependencies` 的 `APIRouter`

现在，假设你的组织为你提供了 `app/internal/admin.py` 文件。

它包含一个带有一些由你的组织在多个项目之间共享的管理员*路径操作*的 `APIRouter`。

对于此示例，它将非常简单。但是假设由于它是与组织中的其他项目所共享的，因此我们无法对其进行修改，以及直接在 `APIRouter` 中添加 `prefix`、`dependencies`、`tags` 等：

```
{!../../../docs_src/bigger_applications/app/internal/admin.py!}
```

但是我们仍然希望在包含 `APIRouter` 时设置一个自定义的 `prefix`，以便其所有*路径操作*以 `/admin` 开头，我们希望使用本项目已经有的 `dependencies` 保护它，并且我们希望它包含自定义的 `tags` 和 `responses`。

我们可以通过将这些参数传递给 `app.include_router()` 来完成所有的声明，而不必修改原始的 `APIRouter`：

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

这样，原始的 `APIRouter` 将保持不变，因此我们仍然可以与组织中的其他项目共享相同的 `app/internal/admin.py` 文件。

结果是在我们的应用程序中，来自 `admin` 模块的每个*路径操作*都将具有：

- `/admin` 前缀。

- `admin` 标签。
- `get_token_header` 依赖项。
- `418` 响应。 

但这只会影响我们应用中的 `APIRouter`，而不会影响使用它的任何其他代码。

因此，举例来说，其他项目能够以不同的身份认证方法使用相同的 `APIRouter`。

包含一个*路径操作*

我们还可以直接将*路径操作*添加到 `FastAPI` 应用中。

这里我们这样做了...只是为了表明我们可以做到🤷：

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

它将与通过 `app.include_router()` 添加的所有其他*路径操作*一起正常运行。

!!! info "特别的技术细节" **注意**：这是一个非常技术性的细节，你也许可以**直接跳过**。

```
---

`APIRouter` 没有被「挂载」，它们与应用程序的其余部分没有隔离。

这是因为我们想要在 `OpenAPI` 模式和用户界面中包含它们的*路径操作*。

由于我们不能仅仅隔离它们并独立于其余部分来「挂载」它们，因此*路径操作*是被「克隆的」（重新创建），而不是直接包含。
```

查看自动化的 API 文档

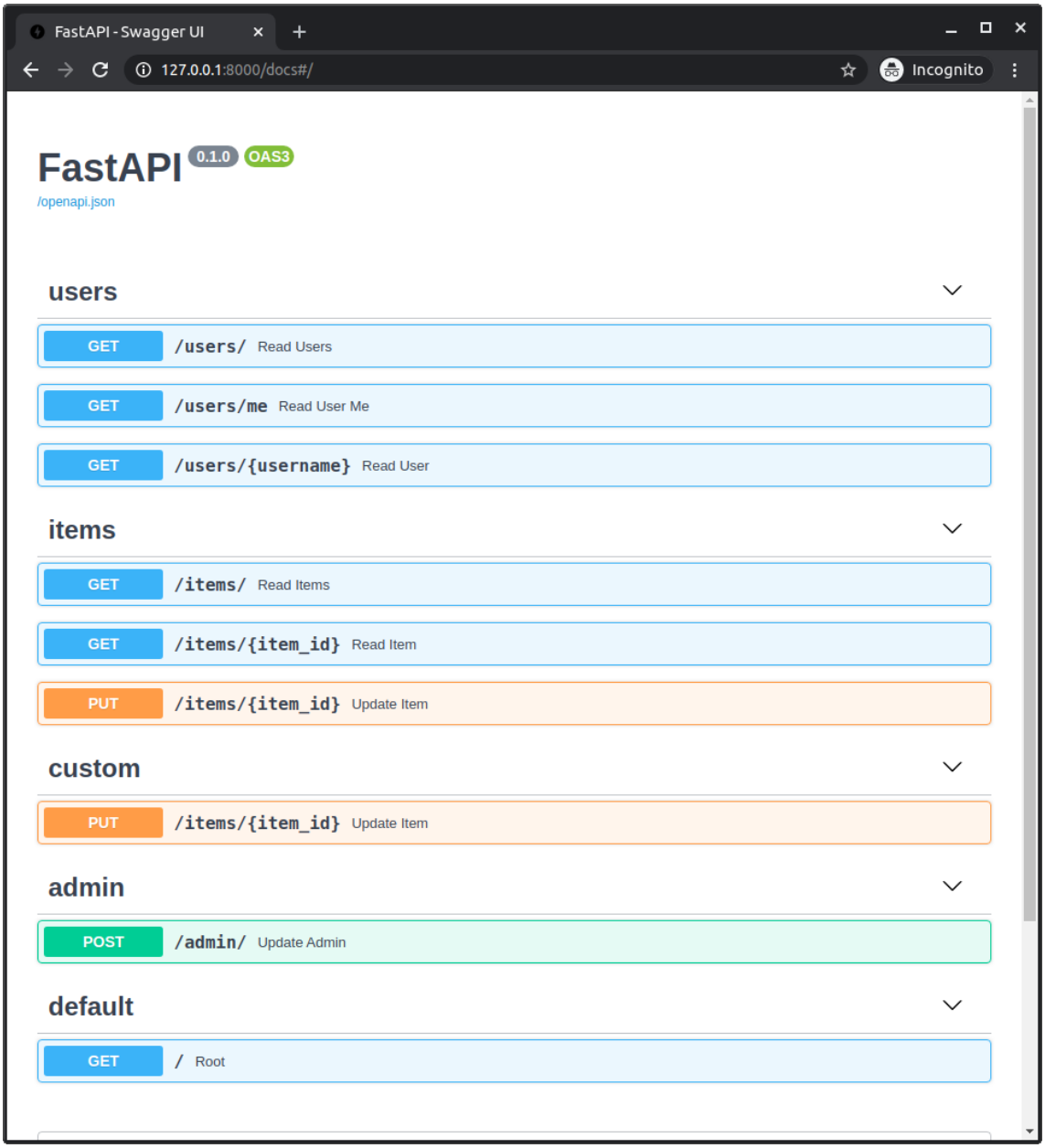
现在，使用 `app.main` 模块和 `app` 变量运行 `uvicorn`：

```
$ uvicorn app.main:app --reload

<span style="color: green;">INFO</span>:     Uvicorn running on
http://127.0.0.1:8000 (Press CTRL+C to quit)
```

然后打开位于 http://127.0.0.1:8000/docs 的文档。

你将看到使用了正确路径（和前缀）和正确标签的自动化 API 文档，包括了来自所有子模块的路径：



多次使用不同的 `prefix` 包含同一个路由器

你也可以在*同*一路由器上使用不同的前缀来多次使用 `.include_router()`。

在有些场景这可能有用，例如以不同的前缀公开同一个的 API，比方说 `/api/v1` 和 `/api/latest`。

这是一个你可能并不真正需要的高级用法，但万一你有需要了就能够用上。

在另一个 `APIRouter` 中包含一个 `APIRouter`

与在 `FastAPI` 应用程序中包含 `APIRouter` 的方式相同，你也可以在另一个 `APIRouter` 中包含 `APIRouter`，通过：

```
router.include_router(other_router)
```

请确保在你将 `router` 包含到 `FastAPI` 应用程序之前进行此操作，以便 `other_router` 中的 路径操作 也能被包含进来。