处理错误

某些情况下，需要向客户端返回错误提示。

这里所谓的客户端包括前端浏览器、其他应用程序、物联网设备等。

需要向客户端返回错误提示的场景主要如下：

- 客户端没有执行操作的权限
- 客户端没有访问资源的权限
- 客户端要访问的项目不存在
- 等等 ...

遇到这些情况时，通常要返回 **4XX**（400 至 499）**HTTP 状态码**。

4XX 状态码与表示请求成功的 **2XX**（200 至 299） HTTP 状态码类似。

只不过，**4XX** 状态码表示客户端发生的错误。

大家都知道「**404 Not Found**」错误，还有调侃这个错误的笑话吧?

使用 `HTTPException`

向客户端返回 HTTP 错误响应，可以使用 `HTTPException`。

导入 `HTTPException`

```
{!../../../docs_src/handling_errors/tutorial001.py!}
```

触发 `HTTPException`

`HTTPException` 是额外包含了和 API 有关数据的常规 Python 异常。

因为是 Python 异常，所以不能 `return`，只能 `raise`。

如在调用*路径操作函数*里的工具函数时，触发了 `HTTPException`，FastAPI 就不再继续执行*路径操作函数*中的后续代码，而是立即终止请求，并把 `HTTPException` 的 HTTP 错误发送至客户端。

在介绍依赖项与安全的章节中，您可以了解更多用 `raise` 异常代替 `return` 值的优势。

本例中，客户端用 `ID` 请求的 `item` 不存在时，触发状态码为 `404` 的异常：

```
{!../../../docs_src/handling_errors/tutorial001.py!}
```

响应结果

请求为 `http://example.com/items/foo`（`item_id` 为 「`foo`」）时，客户端会接收到 HTTP 状态码 - 200 及如下 JSON 响应结果：

```
{
  "item": "The Foo Wrestlers"
}
```

但如果客户端请求 `http://example.com/items/bar`（`item_id` 「bar」 不存在时），则会接收到 HTTP 状态码 - 404（「未找到」错误）及如下 JSON 响应结果：

```
{
  "detail": "Item not found"
}
```

!!! tip "提示"

触发 `HTTPException` 时，可以用参数 `detail` 传递任何能转换为 `JSON` 的值，不仅限于 `str`。

还支持传递 `dict`、`list` 等数据结构。

`**FastAPI**` 能自动处理这些数据，并将之转换为 `JSON`。

添加自定义响应头

有些场景下要为 HTTP 错误添加自定义响应头。例如，出于某些方面的安全需要。

一般情况下可能不会需要在代码中直接使用响应头。

但对于某些高级应用场景，还是需要添加自定义响应头：

```
{!../../../docs_src/handling_errors/tutorial002.py!}
```

安装自定义异常处理器

添加自定义处理器，要使用 Starlette 的异常工具。

假设要触发的自定义异常叫作 `UnicornException`。

且需要 FastAPI 实现全局处理该异常。

此时，可以用 `@app.exception_handler()` 添加自定义异常控制器：

```
{!../../../docs_src/handling_errors/tutorial003.py!}
```

请求 `/unicorns/yolo` 时，路径操作会触发 `UnicornException`。

但该异常将会被 `unicorn_exception_handler` 处理。

接收到的错误信息清晰明了，HTTP 状态码为 `418`，JSON 内容如下：

```
{"message": "Oops! yolo did something. There goes a rainbow..."}
```

!!! note "技术细节"

`from starlette.requests import Request` 和 `from starlette.responses import JSONResponse` 也可以用于导入 `Request` 和 `JSONResponse`。

FastAPI 提供了与 `starlette.responses` 相同的 `fastapi.responses` 作为快捷方式，但大部分响应操作都可以直接从 `Starlette` 导入。同理，`Request` 也是如此。

覆盖默认异常处理器

FastAPI 自带了一些默认异常处理器。

触发 `HTTPException` 或请求无效数据时，这些处理器返回默认的 JSON 响应结果。

不过，也可以使用自定义处理器覆盖默认异常处理器。

覆盖请求验证异常

请求中包含无效数据时，**FastAPI** 内部会触发 `RequestValidationError`。

该异常也内置了默认异常处理器。

覆盖默认异常处理器时需要导入 `RequestValidationError`，并用 `@app.exception_handler(RequestValidationError)` 装饰异常处理器。

这样，异常处理器就可以接收 `Request` 与异常。

```
{!../../../docs_src/handling_errors/tutorial004.py!}
```

访问 `/items/foo`，可以看到以下内容替换了默认 JSON 错误信息：

```
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

以下是文本格式的错误信息：

```
1 validation error
path -> item_id
  value is not a valid integer (type=type_error.integer)
```

`RequestValidationError` vs `ValidationError`

!!! warning "警告"

如果您觉得现在还用不到以下技术细节，可以先跳过下面的内容。

`RequestValidationError` 是 Pydantic 的 `ValidationError` 的子类。

FastAPI 调用的就是 `RequestValidationError` 类，因此，如果在 `response_model` 中使用 Pydantic 模型，且数据有错误时，在日志中就会看到这个错误。

但客户端或用户看不到这个错误。反之，客户端接收到的是 HTTP 状态码为 `500` 的「内部服务器错误」。

这是因为在*响应*或代码（不是在客户端的请求里）中出现的 Pydantic `ValidationError` 是代码的 bug。

修复错误时，客户端或用户不能访问错误的内部信息，否则会造成安全隐患。

覆盖 `HTTPException` 错误处理器

同理，也可以覆盖 `HTTPException` 处理器。

例如，只为错误返回纯文本响应，而不是返回 JSON 格式的内容：

```
{!../../../docs_src/handling_errors/tutorial004.py!}
```

!!! note "技术细节"

```
还可以使用 `from starlette.responses import PlainTextResponse`。

**FastAPI** 提供了与 `starlette.responses` 相同的 `fastapi.responses` 作为快捷方式，但大部分响应都可以直接从 Starlette 导入。
```

使用 `RequestValidationError` 的请求体

`RequestValidationError` 包含其接收到的无效数据请求的 `body` 。

开发时，可以用这个请求体生成日志、调试错误，并返回给用户。

```
{!../../../docs_src/handling_errors/tutorial005.py!}
```

现在试着发送一个无效的 `item`，例如：

```
{
  "title": "towel",
  "size": "XL"
}
```

收到的响应包含 `body` 信息，并说明数据是无效的：

```
{
  "detail": [
    {
      "loc": [
        "body",
        "size"
      ],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
```

```
  ],
  "body": {
    "title": "towel",
    "size": "XL"
  }
}
```

FastAPI `HTTPException` vs Starlette `HTTPException`

FastAPI 也提供了自有的 `HTTPException`。

FastAPI 的 `HTTPException` 继承自 Starlette 的 `HTTPException` 错误类。

它们之间的唯一区别是，**FastAPI** 的 `HTTPException` 可以在响应中添加响应头。

OAuth 2.0 等安全工具需要在内部调用这些响应头。

因此你可以继续像平常一样在代码中触发 **FastAPI** 的 `HTTPException`。

但注册异常处理器时，应该注册到来自 Starlette 的 `HTTPException`。

这样做是为了，当 Starlette 的内部代码、扩展或插件触发 Starlette `HTTPException` 时，处理程序能够捕获、并处理此异常。

注意，本例代码中同时使用了这两个 `HTTPException`，此时，要把 Starlette 的 `HTTPException` 命名为 `StarletteHTTPException`：

```
from starlette.exceptions import HTTPException as StarletteHTTPException
```

复用 FastAPI 异常处理器

FastAPI 支持先对异常进行某些处理，然后再使用 **FastAPI** 中处理该异常的默认异常处理器。

从 `fastapi.exception_handlers` 中导入要复用的默认异常处理器：

```
{!../../../docs_src/handling_errors/tutorial006.py!}
```

虽然，本例只是输出了夸大其词的错误信息。

但也足以说明，可以在处理异常之后再复用默认的异常处理器。