对象注解属性的最佳实践

发布 *3.10.4*

Guido van Rossum
and the Python development team

四月 10, 2022

Python Software Foundation
Email: docs@python.org

Contents

作者 Larry Hastings

摘要

本文意在汇聚对象的注解字典用法的最佳实践。如果 Python 代码会去查看 Python 对象的 `__annotations__` 属性，建议遵循以下准则。

本文分为四个部分：在 Python 3.10 以上版本中访问对象注解的最佳实践、在 Python 3.9 以上版本中访问对象注解的最佳实践、适用于任何 Python 版本的其他 `__annotations__` 最佳实践、`__annotations__` 的特别之处。

请注意，本文是专门介绍 `__annotations__` 的，而不是介绍注解的用法。若要了解“类型提示”的使用信息，请参阅 `typing` 模块。

1 在 Python 3.10 以上版本中访问对象的注解字典

Python 3.10 在标准库中加入了一个新函数：`inspect.get_annotations()`。在 Python 3.10 以上的版本中，调用该函数就是访问对象注解字典的最佳做法。该函数还可以“解析”字符串形式的注解。

有时会因为某些原因看不到 `inspect.get_annotations()`，也可以直接访问 `__annotations__` 数据成员。这方面的最佳实践在 Python 3.10 中也发生了变化：从 Python 3.10 开始，Python 函数、类和模块的 `o.__annotations__` 保证可用。如果确定是要查看这三种对象，只要利用 `o.__annotations__` 读取对象的注释字典即可。

不过其他类型的可调用对象可能就没有定义 `__annotations__` 属性，比如由 `functools.partial()` 创建的可调用对象。当访问某个未知对象的“`__annotations__`“时，Python 3.10 以上版本的最佳做法是带三个参数去调用 `getattr()`，比如 `getattr(o, '__annotations__', None)`。

2 在 Python 3.9 及更早的版本中访问对象的注解字典

在 Python 3.9 及之前的版本中，访问对象的注解字典要比新版本中复杂得多。这个是 Python 低版本的一个设计缺陷，特别是访问类的注解时。

要访问其他对象——函数、可调用对象和模块——的注释字典，最佳做法与 3.10 版本相同，假定不想调用 `inspect.get_annotations()`：你应该用三个参数调用 `getattr()`，以访问对象的 `__annotations__` 属性。

不幸的是，对于类而言，这并不是最佳做法。因为 `__annotations__ 是类的可选属性，并且类可以从基类继承属性，访问某个类的 `__annotations__` 属性可能会无意间返回 基类的注解数据。例如：

```
class Base:
    a: int = 3
    b: str = 'abc'

class Derived(Base):
    pass

print(Derived.__annotations__)
```

如此会打印出 `Base` 的注解字典，而非 `Derived` 的。

若要查看的对象是个类（`isinstance(o, type)`），代码不得不另辟蹊径。这时的最佳做法依赖于 Python 3.9 及之前版本的一处细节：若某个类定义了注解，则会存放于字典 `__dict__` 中。由于类不一定会定义注解，最好的做法是在类的 dict 上调用 `get` 方法。

综上所述，下面给出一些示例代码，可以在 Python 3.9 及之前版本安全地访问任意对象的 `__annotations__` 属性：

```
if isinstance(o, type):
    ann = o.__dict__.get('__annotations__', None)
else:
    ann = getattr(o, '__annotations__', None)
```

运行之后，`ann` 应为一个字典对象或 `None`。建议在继续之前，先用 `isinstance()` 再次检查 `ann` 的类型。

请注意，有些特殊的或畸形的类型对象可能没有 `__dict__` 属性，为了以防万一，可能还需要用 `getattr()` 来访问 `__dict__`。

3 解析字符串形式的注解

有时注释可能会被“字符串化”，解析这些字符串可以求得其所代表的 Python 值，最好是调用 `inspect.get_annotations()` 来完成这项工作。

如果是 Python 3.9 及之前的版本，或者由于某种原因无法使用 `inspect.get_annotations()`，那就需要重现其代码逻辑。建议查看一下当前 Python 版本中 `inspect.get_annotations()` 的实现代码，并遵照实现。

简而言之，假设要对任一对象解析其字符串化的注释 `o`：

- 如果 `o` 是个模块，在调用 `eval()` 时，`o.__dict__` 可视为 `globals`。
- 如果 `o` 是一个类，在调用 `eval()` 时，`sys.modules[o.__module__].__dict__` 视作 `globals`，`dict(vars(o))` 视作 `locals`。
- 如果 `o` 是一个用 `functools.update_wrapper()`、`functools.wraps()` 或 `functools.partial()` 封装的可调用对象，可酌情访问 `o.__wrapped__` 或 `o.func` 进行反复解包，直到你找到未经封装的根函数。
- 如果 `o` 是个可调用对象（但不是一个类），在调用 `eval()` 时，`o.__dict__` 可视为 `globals`。

但并不是所有注解字符串都可以通过 `eval()` 成功地转化为 Python 值。理论上，注解字符串中可以包含任何合法字符串，确实有一些类型提示的场合，需要用到特殊的 *无法被解析的字符串*来作注解。比如：

- **PEP 604** union types using `|`, before support for this was added to Python 3.10.
- 运行时用不到的定义，只在 `typing.TYPE_CHECKING` 为 True 时才会导入。

如果 `eval()` 试图求值，将会失败并触发异常。因此，当要设计一个可采用注解的库 API，建议只在调用方显式请求的时才对字符串求值。

4 任何版本 Python 中使用 `__annotations__` 的最佳实践

- 应避免直接给对象的 `__annotations__` 成员赋值。请让 Python 来管理“__annotations__“。
- 如果直接给某对象的 `__annotations__` 成员赋值，应该确保设成一个“dict“对象。
- 如果直接访问某个对象的 `__annotations__` 成员，在解析其值之前，应先确认其为字典类型。
- 应避免修改 `__annotations__` 字典。
- 应避免删除对象的 `__annotations__` 属性。

5 `__annotations__` 的坑

在 Python 3 的所有版本中，如果对象没有定义注解，函数对象就会直接创建一个注解字典对象。用 `del fn.__annotations__` 可删除 `__annotations__` 属性，但如果后续再访问 `fn.__annotations__`，该对象将新建一个空的字典对象，用于存放并返回注解。在函数直接创建注解字典前，删除注解操作会抛出 `AttributeError` 异常；连续两次调用 `del fn.__annotations__` 一定会抛出一次 `AttributeError` 异常。

以上同样适用于 Python 3.10 以上版本中的类和模块对象。

所有版本的 Python 3 中，均可将函数对象的 `__annotations__` 设为 `None`。但后续用 `fn.__annotations__` 访问该对象的注解时，会像本节第一段所述那样，直接创建一个空字典。但在任何 Python 版本中，模块和类均非如此，他们允许将 `__annotations__` 设为任意 Python 值，并且会留存所设值。

如果 Python 会对注解作字符串化处理（用 `from __future__ import annotations`），并且注解本身就是一个字符串，那么将会为其加上引号。实际效果就是，注解加了 *两次* 引号。例如：

```
from __future__ import annotations
def foo(a: "str"): pass

print(foo.__annotations__)
```

这会打印出 `{'a': "'str'"}`。这不应算是个“坑”；只是因为可能会让人吃惊，所以才提一下。

索引

P