注解最佳实践

作者: Larry Hastings

摘要

本文档旨在概括与注解字典打交道的最佳实践。查看 Python 对象的 `__annotations__` 的代码应遵循下面的准则。

本文档按四部分组织：在 3.10 及更高版本的 Python 中查看对象注解的最佳实践、在 3.9 及更低版本的 Python 中查看对象注解的最佳实践、其它一些适于任何版本的 Python 的 `__annotations__` 的最佳实践、`__annotations__` 的一些"坑"。

本文是 `__annotations__` 的文档，不是注解的用法。如果在寻找如何使用"类型提示"，请参阅 `typing` 模块。

在 3.10 及更高版本的 Python 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 至 3.13 版中，调用该函数是访问任何支持标注的对象的标注字典的最佳实践。 该函数还能为你"反字符串化"已被字符串化的标注。

在 Python 3.14 中，有一个新的 `annotationlib` 模块，具有处理注解的功能。 这包括 `annotationlib.get_annotations()` 函数，它取代了 `inspect.get_annotations()`。

不用 `inspect.get_annotations()` 也可以手动访问 `__annotations__` 这一数据成员。该方法的最佳实践在 Python 3.10 中也发生了变化：从 Python 3.10 开始，对于 Python 函数、类和模块，`o.__annotations__` *保证* 会正常工作。只要你确信所检查的对象是这三种之一，你便可以用 `o.__annotations__` 获取该对象的注解字典。

不过，其它类型的可调用对象可不一定定义了 `__annotations__` 属性，就比如说，`functools.partial()` 创建的可调用对象。当访问某个未知对象的 `__annotations__` 时，3.10 及更高版本的 Python 中的最佳实践是用三个参数去调用 `getattr()`，像 `getattr(o, '__annotations__', None)` 这样。

Python 3.10 之前，在一个没定义注解而其父类定义了注解的类上访问 `__annotations__` 将返回父类的 `__annotations__`。在 3.10 及更高版本的 Python 中，这样的子类的注解是个空字典。

在 3.9 及更低版本的 Python 中访问对象的注解字典

在 3.9 及更低版本的 Python 中访问对象的注解字典要比新版复杂。这是低版本 Python 的设计缺陷，特别是类的注解。

访问其它对象——函数、其它可调用对象和模块——的注解字典的最佳实践与 3.10 版本相同，如果不用 inspect.get_annotations()，就用三个参数去调用 getattr() 以访问对象的 __annotations__ 属性。

不幸的是，对类而言，这并不是最佳实践。问题在于，由于 __annotations__ 在某个类上是可有可无的，而类又可以从基类继承属性，所以访问某个类的 __annotations__ 属性可能会无意间返回 *基类* 的注解字典。如：

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

class Derived(Base):
    pass

print(Derived.__annotations__)
```

会打印出 Base 的注解字典，而非 Derived 的。

如果你所检查的对象是一个类 (isinstance(o, type)) 则你的代码将不得不使用单独的代码路径。在此情况下，最佳实践依赖于 Python 3.9 及之前版本的一个实现细节：如果一个类定义了标注，它们将存储在类的 __dict__ 字典中。 由于类可能有也可能没有定义标注，因此最佳实践是在类的 dict 字典上调用 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。

请注意某些特别的或错误的类型对象可能没有 __dict__ 属性，因此为确保绝对安全你可能会需要使用 getattr() 来访问 __dict__。

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` 为可调用对象（但不是类），则在调用 `eval()` 时可以使用 `o.__globals__` 作为 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：

- 在 Python 支持 **PEP 604** 的联合类型 `|` (Python 3.10) 之前使用它。
- 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。

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。
- 你应该避免在任何对象上直接访问 `__annotations__`。 相反，使用 `annotationlib.get_annotations()` (Python 3.14+) 或 `inspect.get_annotations()` (Python 3.10+)。
- 如果你直接访问一个对象的``__annotations__``成员，在尝试检查其内容之前，你应该确保它是一个字典。
- 应避免修改 `__annotations__` 字典。
- 应避免删除对象的 `__annotations__` 属性。

__annotations__ 的一些“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。

如果你使用一个带有自定义元类的类，并在该类上访问 `__annotations__`，你可能会观察到意外的行为；请参阅 **749** 以获取一些示例。 你可以通过在 Python 3.14+ 上使用 `annotationlib.get_annotations()` 或在 Python 3.10+ 上使用 `inspect.get_annotations()` 来避免这些问题。 在 Python 的早期版本中，你可以通过访问类的 `__dict__` 中的注解来避免这些 bug (例如 `cls.__dict__.get('__annotations__', None)`) 。

在某些版本的 Python 中，类的实例可能具有 `__annotations__` 属性。 但是，这不是支持的功能。 如果你需要一个实例的注解，你可以使用 `type()` 来访问它的类 (例如，Python 3.14+ 上的 `annotationlib.get_annotations(type(myinstance))`)。