

---

# 注解最佳实践

发行版本 3.14.0a0

Guido van Rossum and the Python development team

九月 25, 2024

Python Software Foundation  
Email: docs@python.org

## Contents

1	在 3.10 及更高版本的 Python 中访问对象的注解字典	2
2	在 3.9 及更低版本的 Python 中访问对象的注解字典	2
3	解析字符串形式的注解	3
4	任何版本 Python 中使用 <code>__annotations__</code> 的最佳实践	3
5	<code>__annotations__</code> 的一些“坑”	4
	索引	5

---

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。

## 1 在 3.10 及更高版本的 Python 中访问对象的注解字典

Python 3.10 adds a new function to the standard library: `inspect.get_annotations()`. In Python versions 3.10 through 3.13, calling this function is the best practice for accessing the annotations dict of any object that supports annotations. This function can also “un-stringize” stringized annotations for you.

In Python 3.14, there is a new `annotationlib` module with functionality for working with annotations. This includes a `annotationlib.get_annotations()` function, which supersedes `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。

## 2 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。

若待检查的对象是类（即 `isinstance(o, type)`），代码不得不另辟蹊径。这时的最佳实践依赖于 3.9 及更低版本的 Python 中的一处实现细节：若类定义了注解，则注解会被放入类的 `__dict__` 字典。类不一定有注解，故最佳实践是在类的字典上调用 `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，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：

- 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。

### 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。
- You should avoid accessing `__annotations__` directly on any object. Instead, use `annotationlib.get_annotations()` (Python 3.14+) or `inspect.get_annotations()` (Python 3.10+).
- If you do directly access the `__annotations__` member of an object, you should ensure that it's a dictionary before attempting to examine its contents.
- 应避免修改 `__annotations__` 字典。
- 应避免删除对象的 `__annotations__` 属性。

## 5 `__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。

If you use a class with a custom metaclass and access `__annotations__` on the class, you may observe unexpected behavior; see 749 for some examples. You can avoid these quirks by using `annotationlib.get_annotations()` on Python 3.14+ or `inspect.get_annotations()` on Python 3.10+. On earlier versions of Python, you can avoid these bugs by accessing the annotations from the class's `__dict__` (e.g., `cls.__dict__.get('__annotations__', None)`).

## 索引

### P

Python 增强建议; PEP 604, [3](#)

Python 增强建议; PEP  
749#pep749-metaclasses, [4](#)