typing —— 对类型提示的支持 — Python 3.12.1 文档

`typing` —— 对类型提示的支持¶

3.5 新版功能.

源代码： Lib/typing.py

备注

Python 运行时不强制要求函数与变量类型标注。 它们可被 类型检查器、IDE、语法检查器等第三方工具使用。

本模块提供对类型提示的运行时支持。对于类型系统的原始说明，请参阅 **PEP 484**。一个更简明的介绍是 **PEP 483**。

下面的函数接收与返回的都是字符串，注解方式如下：

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

`greeting` 函数中，参数 `name` 的类型应是 `str` ，返回类型是 `str` 。子类型也可以作为参数。

新的功能频繁地被添加到 `typing` 模块中。typing_extensions 包提供了这些新功能对旧版本 Python 的向后移植。

要获取已弃用特性及其弃用时间线的概要，请参阅 Deprecation Timeline of Major Features。

参见

"类型系统备忘单"

关于类型提示的概览（发布于 mypy 文档站点）

mypy 文档 的 "Type System Reference" 章节

Python 类型系统是通过 PEP 来标准化的，因此该参考应当广泛适用于大多数 Python 类型检查器。 （但某些部分仍然是 mypy 专属的。）

"Static Typing with Python"

由社区编写的不限定具体类型检查器的文档，详细讲解了类型系统特性，有用的类型相关工具以及类型的最佳实践。

相关的 PEP¶

自从在 **PEP 484** 和 **PEP 483** 中首次引入类型提示之来，已有多个 PEP 对 Python 的类型标注框架进行了修改和加强:

▶ The full list of PEPs

类型别名¶

类型别名是使用 `type` 语句来定义的，它将创建一个 `TypeAliasType` 的实例。 在这个示例中， `Vector` 和 `list[float]` 将被静态类型检查器等同处理:

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a
Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

类型别名适用于简化复杂的类型签名。例如：

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers:
Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type
signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
        servers: Sequence[tuple[tuple[str, int], dict[str,
str]]]) -> None:
    ...
```

`type` 语句是在 Python 3.12 中新增加的。 为了向下兼容，类型别名也可以通过简单的赋值来创建：

```
Vector = list[float]
```

或者用 `TypeAlias` 标记来显式说明这是一个类型别名，而非一般的变量赋值：

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

NewType¶

用 `NewType` 助手创建与原类型不同的类型：

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

静态类型检查器把新类型当作原始类型的子类，这种方式适用于捕捉逻辑错误：

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; an int is not a UserId
user_b = get_user_name(-1)
```

`UserId` 类型的变量可执行所有 `int` 操作，但返回结果都是 `int` 类型。这种方式允许在预期 `int` 时传入 `UserId` ，还能防止意外创建无效的

`UserId`：

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

注意，这些检查只由静态类型检查器强制执行。在运行时，语句 `Derived = NewType('Derived', Base)` 将产生一个 `Derived` 可调用对象，该对象立即返回你传递给它的任何参数。 这意味着语句 `Derived(some_value)` 不会创建一个新的类，也不会引入超出常规函数调用的很多开销。

更确切地说，在运行时， `some_value is Derived(some_value)` 表达式总为 True。

创建 `Derived` 的子类型是无效的:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

然而，我们可以在 "派生的" `NewType` 的基础上创建一个 `NewType` 。

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

同时， `ProUserId` 的类型检查也可以按预期执行。

详见 **PEP 484**。

备注

请记住使用类型别名将声明两个类型是相互 *等价* 的。 使用 `type Alias = Original` 将使静态类型检查器在任何情况下都把 `Alias` 视为与 `Original` *完全等价*。 这在你想要简化复杂的类型签名时会很有用处。

反之， `NewType` 声明把一种类型当作另一种类型的 *子类型*。 `Derived = NewType('Derived', Original)` 时，静态类型检查器把 `Derived` 当作 `Original` 的 *子类*，即， `Original` 类型的值不能用在预期 `Derived` 类型的位置。这种方式适用于以最小运行时成本防止逻辑错误。

3.5.2 新版功能.

在 3.10 版更改: `NewType` 现在是一个类而不是一个函数。 因此，当调用 `NewType` 而非常规函数时会有一些额外的运行时开销。

在 3.11 版更改: 调用 `NewType` 的性能已恢复到 Python 3.9 时的水平。

标注可调用对象¶

函数 -- 或其他 callable 对象 -- 可以使用 `collections.abc.Callable` 或 `typing.Callable` 来标注。 `Callable[[int], str]` 表示一个接受 `int` 类型的单个参数并返回 `str` 的函数。

例如:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None])
-> None:
```

```
    ...  # Body

async def on_update(value: str) -> None:
    ...  # Body

callback: Callable[[str], Awaitable[None]] = on_update
```

下标语法总是要刚好使用两个值：参数列表和返回类型。 参数列表必须是一个由类型组成的列表、 `ParamSpec` 、 `Concatenate` 或省略号。 返回类型必须是单一类型。

如果将一个省略号字面值 `...` 作为参数列表，则表示可以接受包含任意形参列表的可调用对象:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # Also OK
```

`Callable` 无法表达复杂的签名如接受可变数量参数的函数，重载的函数，或具有仅限关键字形参的函数。 但是，这些签名可通过自定义具有 `__call__()` 方法的 `Protocol` 类来表达:

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None =
None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner)
```

```
-> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) ->
list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) ->
list[bytes]:
    ...

batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has
incompatible type because of
                         # different name and kind in the
callback
```

以其他可调用对象为参数的可调用对象可以使用 `ParamSpec` 来表明其参数类型是相互依赖的。 此外，如果该可调用对象增加或删除了其他可调用对象的参数，可以使用 `Concatenate` 操作符。 它们分别采取 `Callable[ParamSpecVariable, ReturnType]` 和 `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` 的形式。

在 3.10 版更改: `Callable` 现在支持 `ParamSpec` 和 `Concatenate` 。 详情见 **PEP 612**。

参见

`ParamSpec` 和 `Concatenate` 的文档提供了在 `Callable` 中使用的例子。

泛型（Generic）¶

由于无法以通用方式静态地推断容器中保存的对象的类型信息，标准库中的许多容器类都支持下标操作来以表示容器元素的预期类型。

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the
sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values
in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None:
...
```

泛型函数和类可以通过使用 类型形参语法 来实现参数化:

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # Function is generic
over the TypeVar "T"
    return l[0]
```

或直接使用 `TypeVar` 工厂:

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # Declare type variable
"U"

def second(l: Sequence[U]) -> U:  # Function is generic over
```

```
the TypeVar "U"
    return l[1]
```

在 3.12 版更改: 对泛型的语法支持是在 Python 3.12 中新增的。

标注元组¶

对于 Python 中的大多数容器，类型系统会假定容器中的所有元素都是相同类型的。 例如:

```
from collections.abc import Mapping

# Type checker will infer that all elements in ``x`` are
meant to be ints
x: list[int] = []

# Type checker error: ``list`` only accepts a single type
argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant
to be strings,
# and that all values in ``z`` are meant to be either
strings or ints
z: Mapping[str, str | int] = {}
```

`list` 只接受一个类型参数，因此类型检查器将在上述代码中对 `y` 赋值时报告错误。同样， `Mapping` 只接受两个类型参数：第一个给出键的类型，第二个则给出值的类型。

然而，与大多数其它 Python 容器不同的是，在常见的 Python 代码中，元组中元素的类型并不相同。因此，在 Python 的类型系统中，元组是特殊情况。 `tuple` 可以接受 *任意数量* 的类型参数:

```
# OK: ``x`` is assigned to a tuple of length 1 where the
sole element is an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

要表示一个可以是 *任意* 长度的元组，并且其中的所有元素都是相同类型的 `T` ，请使用 `tuple[T, ...]` 。要表示空元组，请使用 `tuple[()]` 。只使用 `tuple` 作为注解等效于使用``tuple[Any, ...]``：

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates
x can be of any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must
be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent
to ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

类对象的类型¶

用 `C` 注解的变量可以接受类型 `C` 的值。然而，用类型 `type[C]` （或者 `typing.Type[C]` ）注解的变量则可以接受本身是类的值——准确地说，是接受 `C` 的 *类对象* 。例如：

```
a = 3         # Has type ``int``
b = int       # Has type ``type[int]``
c = type(a)   # Also has type ``type[int]``
```

注意， `type[C]` 是协变的：

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)      # OK
make_new_user(ProUser)   # Also OK: ``type[ProUser]`` is a
subtype of ``type[User]``
make_new_user(TeamUser)  # Still fine
make_new_user(User())    # Error: expected ``type[User]``
but got ``User``
make_new_user(int)       # Error: ``type[int]`` is not a
subtype of ``type[User]``
```

`type` 的合法形参只有类， `Any` ，类型变量 以及前面这些类型的并集。 例如：

```
def new_non_team_user(user_class: type[BasicUser |
ProUser]): ...
```

```
new_non_team_user(BasicUser)  # OK
new_non_team_user(ProUser)    # OK
new_non_team_user(TeamUser)   # Error: ``type[TeamUser]`` is
not a subtype
                              # of ``type[BasicUser |
ProUser]``
new_non_team_user(User)       # Also an error
```

`type[Any]` 等价于 `type` ，它是 Python 的 元类层级结构 的根对象。

用户定义的泛型类型¶

用户定义的类可以定义为泛型类。

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger)
-> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value
```

```
def log(self, message: str) -> None:
    self.logger.info('%s: %s', self.name, message)
```

这种语法表示类 `LoggedVar` 是围绕单个 类型变量 `T` 实现参数化的。 这也使得 `T` 成为类体内部有效的类型。

泛型类隐式继承自 `Generic` 。为了与 Python 3.11 及更低版本兼容，也允许显式地从 `Generic` 继承以表示泛型类：

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

泛型类具有 `__class_getitem__()` 方法，这意味着泛型类可在运行时进行参数化（例如下面的 `LoggedVar[int]` ）：

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

一个泛型可以有任何数量的类型变量。所有种类的 `TypeVar` 都可以作为泛型的参数：

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...
```

```
OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes],
covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

`Generic` 类型变量的参数应各不相同。下列代码就是无效的:

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

泛型类也可以从其他类继承:

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

从泛型类继承时，某些类型参数可被固定:

```
from collections.abc import Mapping
```

```
class MyDict[T](Mapping[str, T]):
    ...
```

在这个例子中， `MyDict` 就只有一个参数 `T` 。

未指定泛型类的类型参数时，会假定每个位置的类型都为 `Any` 。在下面的例子中， `MyIterable` 不是泛型，但却隐式继承了 `Iterable[Any]` ：

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

用户定义的泛型类型别名也同样受到支持。例如：

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: #
Same as Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

出于向后兼容性的考虑，也允许使用简单的赋值来创建泛型类型别名：

```
from collections.abc import Iterable
from typing import TypeVar
```

```
S = TypeVar("S")
Response = Iterable[S] | int
```

在 3.7 版更改: `Generic` 不再支持自定义元类。

在 3.12 版更改: 3.12 版本新增了对泛型和类型别名的语法支持。在之前的版本中，泛型类必须显式继承自 `Generic` ，或者在其基类之一中包含有类型变量。

用户定义的参数表达式的泛型也受到支持，可以采用 `[**P]` 形式的参数规格变量来表示。该行为与上面描述的类型变量一致，因为参数规格变量被 typing 模块视为专门的类型变量。这方面的一个例外是，类型的列表可用于替代 `ParamSpec` ：

```
>>>
```

```
>>> class Z[T, **P]: ...  # T is a TypeVar; P is a ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

带有 `ParamSpec` 的泛型类也可以使用从 `Generic` 显式继承的方式来创建。在这种情况下，不需要使用 `**` ：

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

`TypeVar` 与 `ParamSpec` 的另一个区别在于只有单个参数规格变量的泛型会接受形如 `X[[Type1, Type2, ...]]` 的参数列表，同时为了美观，也接受 `X[Type1, Type2, ...]` 这样的形式。 在内部，后者被转换为前者，所以下面的内容是等价的：

```
>>>
```

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

请注意：在某些情况下，具有 `ParamSpec` 的泛型在替换后可能不具有正确的 `__parameters__` ，因为参数规格主要用于静态类型检查。

在 3.10 版更改: `Generic` 现在可以通过参数表达式进行参数化。参见 `ParamSpec` 和 **PEP 612** 以了解更多细节。

用户定义的泛型类可以将 ABC 作为基类而不会导致元类冲突。 参数化泛型的输出结果会被缓存，且 typing 模块中的大多数类型都是 hashable 并且支持相等性比较。

`Any` 类型¶

`Any` 是一种特殊的类型。静态类型检查器认为所有类型均与 `Any` 兼容，同样， `Any` 也与所有类型兼容。

也就是说，可对 `Any` 类型的值执行任何操作或方法调用，并赋值给任意变量：

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

注意，`Any` 类型的值赋给更精确的类型时，不执行类型检查。例如，把 `a` 赋给 `s`，在运行时，即便 `s` 已声明为 `str` 类型，但接收 `int` 值时，静态类型检查器也不会报错。

此外，未指定返回值与参数类型的函数，都隐式地默认使用 `Any`：

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

需要混用动态与静态类型代码时，此操作把 `Any` 当作 *应急出口*。

`Any` 和 `object` 的区别。与 `Any` 相似，所有类型都是 `object` 的子类型。然而，与 `Any` 不同，object 不可逆：`object` *不是* 其它类型的子类型。

就是说，值的类型是 `object` 时，类型检查器几乎会拒绝所有对它的操作，并且，把它赋给更精确的类型变量（或返回值）属于类型错误。例如：

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic'
method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses
of object
hash_a(42)
hash_a("foo")

# Passes type checking, since Any is compatible with all
types
hash_b(42)
hash_b("foo")
```

使用 `object` ，说明值能以类型安全的方式转为任何类型。使用 `Any` ，说明值是动态类型。

名义子类型 vs 结构子类型¶

最初 **PEP 484** 将 Python 静态类型系统定义为使用 *名义子类型*。这意味着当且仅当类 `A` 是 `B` 的子类时，才满足有类 `B` 预期时使用类 `A` 。

此项要求以前也适用于抽象基类，例如， `Iterable` 。这种方式的问题在于，定义类时必须显式说明，既不 Pythonic，也不是动态类型式 Python 代码的惯用写法。例如，下列代码就遵从了 **PEP 484** 的规范：

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 允许用户在类定义时不显式说明基类，从而解决了这一问题，静态类型检查器隐式认为 `Bucket` 既是 `Sized` 的子类型，又是 `Iterable[int]` 的子类型。这就是 *结构子类型* （又称为静态鸭子类型）：

```
from collections.abc import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

此外，结构子类型的优势在于，通过继承特殊类 `Protocol` ，用户可以定义新的自定义协议（见下文中的例子）。

模块内容¶

`typing` 模块定义了下列类、函数和装饰器。

特殊类型原语¶

特殊类型¶

这些类型可用于在注解中表示类型，但不支持下标用法（`[]`）。

typing.**Any**¶

特殊类型，表示没有约束的类型。

- 所有类型都与 `Any` 兼容。
- `Any` 与所有类型都兼容。

在 3.11 版更改: `Any` 现在可以用作基类。这有助于避免类型检查器在高度动态或可通过鸭子类型使用的类上报错。

typing.**AnyStr**¶

。

定义：

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` 用于可接受 `str` 或 `bytes` 参数但不允许两者混用的函数。

例如：

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b
```

```
concat("foo", "bar")    # OK, output has type 'str'
concat(b"foo", b"bar")  # OK, output has type 'bytes'
concat("foo", b"bar")   # Error, cannot mix str and bytes
```

请注意：尽管名为 `AnyStr`，但它与 `Any` 类型毫无关系，也不是指"任何字符串"。而且， `AnyStr` 更是和 `str | bytes` 彼此互不相同，各有各的使用场景：

```
# Invalid use of AnyStr:
# The type variable is used only once in the function
signature,
# so cannot be "solved" by the type checker
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

typing.LiteralString¶

只包括字符串字面值的的特殊类型。

任何字符串字面值或其他 `LiteralString` 都与 `LiteralString` 兼容。但 `str` 类型的对象不与其兼容。组合 `LiteralString` 类型的对象产生的字符串也被认为是 `LiteralString` 。

示例:

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string:
```

```
LiteralString) -> None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name =
{arbitrary_string}"
    )
```

`LiteralString` 对于会因用户可输入任意字符串而导致问题的敏感 API 很有用。例如，上述两处导致类型检查器报错的代码可能容易被 SQL 注入攻击。

请参阅 **PEP 675** 了解详情。

3.11 新版功能.

typing.Never¶

底类型，一个没有成员的类型。

这可以用于定义一个永不应该被调用的函数，或一个永不返回的函数：

```
from typing import Never

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # type checker error
    match arg:
        case int():
            print("It's an int")
        case str():
```

```
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never
```

3.11 新版功能: 在更老的 Python 版本上， `NoReturn` 可被用于表达相同的概念。 `Never` 为了更显式地表达这个意图被加入。

typing.NoReturn¶

特殊类型，表示永不返回的函数。

例如：

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

`NoReturn` 也可以用于 底类型 的定义，这是一种没有值的类型。自从 Python 3.11 开始，应该使用 `Never` 类型代替这个概念。类型检查器应该将这两种类型视为等价。

3.5.4 新版功能.

3.6.2 新版功能.

typing.Self¶

特殊类型，表示当前闭包内的类。

例如：

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type
is "SubclassOfFoo"
```

此注解在语法上等价于以下代码，但形式更为简洁：

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

通常来说，如果某些内容返回 `self`，如上面的示例所示，您应该使用 `Self` 作为返回值注解。如果 `Foo.return_self` 被注解为返回 `"Foo"`，那么类型检查器将推断从 `SubclassOfFoo.return_self` 返回的对象是 `Foo` 类型，而不是 `SubclassOfFoo`。

其它常见用例包括：

- 被用作替代构造器的 `classmethod`，它将返回 `cls` 形参的实例。
- 标注一个返回自身的 `__enter__()` 方法。

如果不能保证在子类中方法会返回子类的实例（而非父类的实例），则不应使用 `Self` 作为返回值注解：

```
class Eggs:
    # Self would be an incorrect return annotation here,
    # as the object returned is always an instance of Eggs,
    # even in subclasses
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

更多细节请参见 **PEP 673**。

3.11 新版功能.

typing.TypeAlias¶

特殊注解，用于显式声明 类型别名.

例如：

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

在较早的 Python 版本上，`TypeAlias` 对注解使用前向引用的别名时特别有用，因为类型检查器可能很难将这些别名与正常的变量赋值区分开来：

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on
```

```
Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a
type alias declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

请参阅 **PEP 613** 了解详情。

3.10 新版功能.

3.12 版后已移除: `TypeAlias` 被弃用，请使用 `type` 语句，后者创建 `TypeAliasType` 的实例，并且天然支持正向引用。请注意，虽然 `TypeAlias` 和 `TypeAliasType` 具有相似的用途和名称，但它们是不同的，后者并不是前者的类型。目前还没有移除 `TypeAlias` 的计划，但鼓励用户迁移到 `type` 语句。

特殊形式¶

这些内容在注解中可以视为类型，且都支持下标用法（ `[]` ），但每个都有唯一的语法。

typing.Union¶

联合类型； `Union[X, Y]` 等价于 `X | Y` ，意味着满足 X 或 Y 之一。

要定义一个联合类型，可以使用类似 `Union[int, str]` 或简写 `int | str` 。建议使用这种简写。细节:

- 参数必须是某种类型，且至少有一个。

- 联合类型之联合类型会被展平，例如：

```
Union[Union[int, str], float] == Union[int, str, float]
```

- 单参数之联合类型就是该参数自身，例如：

```
Union[int] == int  # The constructor actually returns
int
```

- 冗余的参数会被跳过，例如：

```
Union[int, str, int] == Union[int, str] == int | str
```

- 比较联合类型，不涉及参数顺序，例如：

```
Union[int, str] == Union[str, int]
```

- 不可创建 `Union` 的子类或实例。

- 没有 `Union[X][Y]` 这种写法。

在 3.7 版更改: 在运行时，不要移除联合类型中的显式子类。

在 3.10 版更改: 联合类型现在可以写成 `X | Y` 。 参见 联合类型表达式。

typing.**Optional**¶

`Optional[X]` 等价于 `X | None` （或 `Union[X, None]` ） 。

注意，可选类型与含默认值的可选参数不同。含默认值的可选参数不需要在类型注解上添加 `Optional` 限定符，因为它仅是可选的。例如：

```
def foo(arg: int = 0) -> None:
    ...
```

另一方面，显式应用 `None` 值时，不管该参数是否可选， `Optional` 都适用。例如：

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

在 3.10 版更改: 可选参数现在可以写成 `X | None` 。 参见 联合类型表达式。

typing.Concatenate¶

特殊形式，用于注解高阶函数。

`Concatenate` 可用于与 Callable 和 `ParamSpec` 连用来注解高阶可调用对象，该可象可以添加、移除或转换另一个可调用对象的形参。 使用形式为 `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]` 。 `Concatenate` 目前仅可用作传给 Callable 的第一个参数。传给 `Concatenate` 的最后一个形参必须是 `ParamSpec` 或省略号（ `...` ）。

例如，为了注释一个装饰器 `with_lock` ，它为被装饰的函数提供了 `threading.Lock` ， `Concatenate` 可以用来表示 `with_lock` 期望一个可调用对象，该对象接收一个 `Lock` 作为第一个参数，并返回一个具有不同类型签名的可调用对象。 在这种情况下， `ParamSpec` 表示返回的可调用对象的参数类型取决于被传入的可调用程序的参数类型:

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing
```

```
a function
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R])
-> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) ->
float:
    '''Add a list of numbers together in a thread-safe
manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the
decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

3.10 新版功能.

参见

- **PEP 612** -- 参数规范变量（引入 `ParamSpec` 和 `Concatenate` 的 PEP）
-
-

typing.Literal¶

特殊类型注解形式，用于定义“字面值类型”。

`Literal` 可以用来向类型检查器说明被注解的对象具有与所提供的字面量之一相同的值。

例如：

```
def validate_simple(data: Any) -> Literal[True]:  # always
returns True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

`Literal[...]` 不能创建子类。在运行时，任意值均可作为 `Literal[...]` 的类型参数，但类型检查器可以对此加以限制。字面量类型详见 **PEP 586** 。

3.8 新版功能.

在 3.9.1 版更改: `Literal` 现在能去除形参的重复。 `Literal` 对象的相等性比较不再依赖顺序。 现在如果有某个参数不为 hashable， `Literal` 对象在相等性比较期间将引发 `TypeError` 。

typing.ClassVar¶

特殊类型注解构造，用于标注类变量。

如 **PEP 526** 所述，打包在 ClassVar 内的变量注解是指，给定属性应当用作类变量，而不应设置在类实例上。用法如下：

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

`ClassVar` 仅接受类型，也不能使用下标。

`ClassVar` 本身不是类，不应用于 `isinstance()` 或 `issubclass()` 。`ClassVar` 不改变 Python 运行时行为，但可以用于第三方类型检查器。例如，类型检查器会认为以下代码有错：

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on
instance
Starship.stats = {}     # This is OK
```

3.5.3 新版功能.

typing.Final¶

特殊类型注解构造，用于向类型检查器表示最终名称。

不能在任何作用域中重新分配最终名称。类作用域中声明的最终名称不能在子类中重写。

例如：

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
```

```
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

这些属性没有运行时检查。详见 **PEP 591**。

3.8 新版功能.

typing.Required¶

特殊类型注解构造，用于标记 `TypedDict` 键为必填项。

这主要用于 `total=False` 的 TypedDict。有关更多详细信息，请参阅 `TypedDict` 和 **PEP 655** 。

3.11 新版功能.

typing.NotRequired¶

特殊类型注解构造，用于标记 `TypedDict` 键为可能不存在的键。

详情参见 `TypedDict` 和 **PEP 655**。

3.11 新版功能.

typing.Annotated¶

特殊类型注解形式，用于向注解添加特定于上下文的元数据。

使用注解 `Annotated[T, x]` 将元数据 `x` 添加到给定类型 `T` 。使用 `Annotated` 添加的元数据可以被静态分析工具使用，也可以在运行时使用。在运行时使用的情况下，元数据存储在 `__metadata__` 属性中。

如果库或工具遇到注解 `Annotated[T, x]` ，并且没有针对这一元数据的特殊处理逻辑，则应该忽略该元数据，简单地将注解视为 `T` 。因此，`Annotated` 对于希望将注解用于 Python 的静态类型注解系统之外的目的的代码很有用。

使用 `Annotated[T, x]` 作为注解仍然允许对 `T` 进行静态类型检查，因为类型检查器将简单地忽略元数据 `x` 。因此， `Annotated` 不同于 `@no_type_check` 装饰器，后者虽然也可以用于在类型注解系统范围之外添加注解，但是会完全禁用对函数或类的类型检查。

具体解释元数据的方式由遇到 `Annotated` 注解的工具或库来负责。遇到 `Annotated` 类型的工具或库可以扫描元数据的各个元素以确定其是否有意处理（比如使用 `isinstance()` ）。

Annotated[<type>, <metadata>]

以下示例演示在进行区间范围分析时使用 `Annotated` 将元数据添加到类型注解的方法：

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

语法细节：

- `Annotated` 的第一个参数必须是有效的类型。
- 可提供多个元数据的元素（ `Annotated` 支持可变参数）：

```
@dataclass
class ctype:
    kind: str

Annotated[int, ValueRange(3, 10), ctype("char")]
```

由处理注解的工具决定是否允许向一个注解中添加多个元数据元素，以及如何合并这些注解。

- `Annotated` 至少要有两个参数（ `Annotated[int]` 是无效的）
- 元数据元素的顺序会被保留，且影响等价检查：

```
assert Annotated[int, ValueRange(3, 10), ctype("char")]
!= Annotated[
    int, ctype("char"), ValueRange(3, 10)
]
```

- 嵌套的 `Annotated` 类型会被展平。元数据元素从最内层的注解开始依次展开：

```
assert Annotated[Annotated[int, ValueRange(3, 10)],
ctype("char")] == Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

- 元数据中的重复元素不会被移除：

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

- `Annotated` 可以与嵌套别名和泛型别名一起使用：

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will
treat "V" the same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

- `Annotated` 不能与已解包的 `TypeVarTuple` 一起使用：

```
type Variadic[*Ts] = Annotated[*Ts, Ann1]  # NOT valid
```

这等价于：

```
Annotated[T1, T2, T3, ..., Ann1]
```

其中 `T1` 、 `T2` 等都是 `TypeVars` 。这种写法无效：应当只有一个类型被传递给 Annotated。

- 默认情况下， `get_type_hints()` 会去除注解中的元数据。传入 `include_extras=True` 可以保留元数据：

>>>

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None:
pass
...
```

```
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
{'x': typing.Annotated[int, 'metadata'], 'return':
<class 'NoneType'>}
```

- 在运行时，与特定 `Annotated` 类型相关联的元数据可通过 `__metadata__` 属性来获取：

>>>

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

参见

PEP 593 - 灵活的函数与变量标注

该 PEP 将 `Annotated` 引入到标准库中。

3.9 新版功能.

typing.**TypeGuard**¶

用于标记用户自定义类型防护函数的特殊类型构造。

`TypeGuard` 可被用于标注用户自定义类型保护函数的返回类型。 `TypeGuard` 只接受单独的类型参数。 在运行时，以这种方式标记的函数应当返回一个布尔值。

PX旨在使 *类型缩小* 受益--这是静态类型检查器用来确定程序代码流中表达式的更精确类型的一种技术。通常，类型缩小是通过分析条件性代码流并将缩小的结果应用于一个代码块来完成的。 这里的条件表达式有时被称为 "类型保护":

```
def is_str(val: str | float):
    # "isinstance" type guard
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

有时，使用一个用户定义的布尔函数作为类型保护会很方便。 这样的函数应该使用 `TypeGuard[...]` 作为其返回类型，以提醒静态类型检查器注意这一意图。

对于一个给定的函数，使用 `-> TypeGuard` 告诉静态类型检查器:

1. 返回值是一个布尔值。

2. 如果返回值是 `True` ，其参数的类型是 `TypeGuard` 里面的类型。

例如:

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are
strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
```

```
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

如果 `is_str_list` 是一个类或实例方法，那么 `TypeGuard` 中的类型映射到 `cls` 或 `self` 之后的第二个参数的类型。

简而言之，`def foo(arg: TypeA) -> TypeGuard[TypeB]: ...` 形式的意思是：如果 `foo(arg)` 返回 `True`，那么 `arg` 将把 `TypeA` 缩小为 `TypeB`。

备注

`TypeB` 无需为 `TypeA` 的缩小形式 -- 它甚至可以是扩大形式。 主要原因是允许像把 `list[object]` 缩小到 `list[str]` 这样的事情，即使后者不是前者的一个子类型，因为 `list` 是不变的。 编写类型安全的类型防护的责任留给了用户。

`TypeGuard` 也适用于类型变量。 详情参见 **PEP 647**。

3.10 新版功能.

typing.**Unpack**¶

在概念上将对象标记为已解包的类型运算符。

例如，在一个 类型变量元组 上使用解包运算符 `*` 就等价于使用 `Unpack` 来将该类型变量元组标记为已被解包:

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

实际上，`Unpack` 在 `typing.TypeVarTuple` 和 `builtins.tuple` 类型的上下文中可以和 `*` 互换使用。你可能会看到 `Unpack` 在较旧版本的 Python 中被显式地使用，这时 `*` 在特定场合则是无法使用的:

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and
backwards-compatible
```

`Unpack` 也可以与 `typing.TypedDict` 一起使用以便在函数签名中对 `**kwargs` 进行类型标注:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of
type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

请参阅 **PEP 692** 了解将 `Unpack` 用于 `**kwargs` 类型标注的更多细节。

3.11 新版功能.

构造泛型类型与类型别名¶

下列类不应被直接用作标注。 它们的设计目标是作为创建泛型类型和类型别名的构件。

这些对象可通过特殊语法 (类型形参列表 和 `type` 语句) 来创建。 为了与 Python 3.11 及更早版本的兼容性，它们也可不用专门的语法来创建，如下文所述。

***class*typing.Generic¶**

用于泛型类型的抽象基类。

泛型类型通常是通过在类名后添加一个类型形参列表来声明的:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

这样的类将隐式地继承自 `Generic` 。 对于该语法的运行语义的讨论参见 语言参考。

该类的用法如下:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X,
default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

此处函数名之后的圆括号是表示 泛型函数。

为了保持向下兼容性，泛型类也可通过显式地继承自 `Generic` 来声明。 在此情况下，类型形参必须单独声明:

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

*class*typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*)¶

类型变量。

构造类型变量的推荐方式是使用针对 泛型函数, 泛型类 和 泛型类型别名 的专门语法:

```
class Sequence[T]:  # T is a TypeVar
    ...
```

此语法也可被用于创建绑定和带约束的类型变量:

```
class StrSequence[S: str]:  # S is a TypeVar bound to str
    ...


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar
constrained to str or bytes
    ...
```

不过，如有需要，也可通过手动方式来构造可重用的类型变量，就像这样:

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

类型变量的主要用处是为静态类型检查器提供支持。 它们可作为泛型类型以及泛型函数和类型别名定义的形参。 请参阅 `Generic` 了解有关泛型类型的更多信息。 泛型函数的作用方式如下:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

请注意，类型变量可以是 *被绑定的*， *被约束的*，或者两者都不是，但不能既是被绑定的 *又是* 被约束的。

类型变量的种类是在其通过 类型形参语法 创建时或是在传入 `infer_variance=True` 时由类型检查器推断得到的。 手动创建的类型变量可通过传入 `covariant=True` 或 `contravariant=True` 被显式地标记为 covariant 或 contravariant。 在默认情况下，手动创建的类型变量为 invariant。 请参阅 **PEP 484** 和 **PEP 695** 了解更多细节。

绑定类型变量和约束类型变量在几个重要方面具有不同的主义。 使用 *绑定* 类型变量意味着 `TypeVar` 将尽可能使用最为专属的类型来解析:

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of
str
```

类型变量可以被绑定到具体类型、抽象类型（ABC 或 protocol），甚至是类型的联合:

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of
the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with
an __abs__ method
```

但是，如果使用 *约束* 类型变量，则意味着 `TypeVar` 只能被解析为恰好是给定的约束之一:

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'),
StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite
StringSubclass being passed in
```

```
c = concatenate('one', b'two')  # error: type variable 'A'
can be either str or bytes in a function call, but not both
```

在运行时， `isinstance(x, T)` 将引发 `TypeError` 。

__name__¶

类型变量的名称。

__covariant__¶

类型变量是否已被显式地标记为 covariant。

__contravariant__¶

类型变量是否已被显式地标记为 contravariant。

__infer_variance__¶

类型变量的种类是否应由类型检查器来推断。

3.12 新版功能.

__bound__¶

类型变量的绑定，如果有的话。

在 3.12 版更改: 对于通过 类型形参语法 创建的类型变量，只有在属性被访问的时候才会对绑定求值，而不是在类型变量被创建的时候 (参见 惰性求值)。

__constraints__¶

一个包含对类型变量的约束的元组，如果有的话。A tuple containing the constraints of the type variable, if any.

在 3.12 版更改: 对于通过 类型形参语法 创建的类型变量，只有在属性被访问的时候才会对约束求值，而不是在类型变量被创建的时候 (参见 惰性求值)。

在 3.12 版更改: 类型变量现在可以通过使用 **PEP 695** 引入的 类型形参 语法来声明。 增加了 `infer_variance` 形参。

***class*typing.TypeVarTuple(*name*)¶**

类型变量元组。 一种启用了 *variadic* 泛型的专属 类型变量 形式。

类型变量元组可以通过在 类型形参列表 中使用名称前的单个星号 (`*`) 来声明:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -
> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

或者通过显式地发起调用 `TypeVarTuple` 构造器:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) ->
tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

一个普通类型变量将启用单个类型的形参化。 作为对比，一个类型变量元组通过将 *任意* 数量的类型变量封包在一个元组中来允许 *任意* 数量类型的形参化。 例如:

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str,
int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type
tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

请注意解包运算符 `*` 在 `tuple[T, *Ts]` 中的使用。 在概念上，你可以将 `Ts` 当作一个由类型变量组成的元组 `(T1, T2, ...)` 。 那么 `tuple[T, *Ts]` 就将变为 `tuple[T, *(T1, T2, ...)]` ，这等价于 `tuple[T, T1, T2, ...]` 。 （请注意在旧版本 Python 中，你可能会看到改用 `Unpack` 的写法，如 `Unpack[Ts]` 。）

类型变量元组 *总是* 要被解包。 这有助于区分类型变量元组和普通类型变量:

```
x: Ts          # Not valid
x: tuple[Ts]   # Not valid
x: tuple[*Ts]  # The correct way to do it
```

类型变量元组可被用在与普通类型变量相同的上下文中。 例如，在类定义、参数和返回类型中:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

类型变量元组可以很好地与普通类型变量结合在一起:

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()     # Totally
fine
int_array_2d: Array[int, Height, Width] = Array()  # Yup,
fine too
```

但是，请注意在一个类型参数或类型形参列表中最多只能有一个类型变量元组:

```
x: tuple[*Ts, *Ts]            # Not valid
class Array[*Shape, *Shape]:  # Not valid
    pass
```

最后，一个已解包的类型变量元组可以被用作 `*args` 的类型标注:

```
def call_soon[*Ts](
        callback: Callable[[*Ts], None],
        *args: *Ts
) -> None:
```

```
    ...
    callback(*args)
```

相比非解包的 `*args` 标注 —— 例如 `*args: int` ，它将指明 *所有* 参数均为 `int` —— `*args: *Ts` 启用了对 `*args` 中 *单个* 参数的类型的引用。 在此，这允许我们确保传入 `call_soon` 的 `*args` 的类型与 `callback` 的（位置）参数的类型相匹配。

关于类型变量元组的更多细节，请参见 **PEP 646**。

__name__¶

类型变量元组的名称。

3.11 新版功能.

在 3.12 版更改: 类型变量元组现在可以使用 **PEP 695** 所引入的 类型形参 语法来声明。

***class*typing.ParamSpec(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*)**¶

形参专属变量。 类型变量 的一个专用版本。

In 类型形参列表，形参规格可以使用两个星号（ `**` ）来声明:

```
type IntFunc[**P] = Callable[P, int]
```

为了保持与 Python 3.11 及更早版本的兼容性， `ParamSpec` 对象也可以这样创建:

```
P = ParamSpec('P')
```

参数规范变量的存在主要是为了使静态类型检查器受益。 它们被用来将一个可调用对象的参数类型转发给另一个可调用对象的参数类型——这种模式通常出现在高阶函数和装饰器中。 它们只有在 `Concatenate` 中使用时才有效，或者作为 `Callable` 的第一个参数，或者作为用户定义的泛型的参数。 参见 `Generic` 以了解更多关于泛型的信息。

例如，为了给一个函数添加基本的日志记录，我们可以创建一个装饰器 `add_logging` 来记录函数调用。 参数规范变量告诉类型检查器，传入装饰器的可调用对象和由其返回的新可调用对象有相互依赖的类型参数:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P,
T]:
    '''A type-safe decorator to add logging to a
function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

如果没有 `ParamSpec` ，以前注释这个的最简单的方法是使用一个 `TypeVar` 与绑定 `Callable[..., Any]` 。

1. 类型检查器不能对 `inner` 函数进行类型检查，因为 `*args` 和 `**kwargs` 的类型必须是 `Any` 。

2. `cast()` 在返回 `inner` 函数时，可能需要在 `add_logging` 装饰器的主体中进行，或者必须告诉静态类型检查器忽略 `return inner` 。

args¶

kwargs¶

由于 `ParamSpec` 同时捕获了位置参数和关键字参数， `P.args` 和 `P.kwargs` 可以用来将 `ParamSpec` 分割成其组成部分。 `P.args` 代表给定调用中的位置参数的元组，只能用于注释 `*args` 。 `P.kwargs` 代表给定调用中的关键字参数到其值的映射，只能用于注释 `**kwargs` 。在运行时， `P.args` 和 `P.kwargs` 分别是 `ParamSpecArgs` 和 `ParamSpecKwargs` 的实例。

__name__¶

形参规格的名称。

用 `covariant=True` 或 `contravariant=True` 创建的参数规范变量可以用来声明协变或逆变泛型类型。 参数 `bound` 也被接受，类似于 `TypeVar` 。 然而这些关键字的实际语义还有待决定。

3.10 新版功能.

在 3.12 版更改: 形参说明现在可以使用 **PEP 695** 所引入的 类型形参 语法来声明。

备注

只有在全局范围内定义的参数规范变量可以被 pickle。

参见

- **PEP 612** -- 参数规范变量（引入 `ParamSpec` 和 `Concatenate` 的 PEP）

-

-

typing.**ParamSpecArgs**¶

typing.**ParamSpecKwargs**¶

`ParamSpec`的参数和关键字参数属性。``ParamSpec` 的 `P.args` 属性是 `ParamSpecArgs` 的一个实例， `P.kwargs` 是 `ParamSpecKwargs` 的一个实例。 它们的目的是用于运行时内部检查的，对静态类型检查器没有特殊意义。

在这些对象中的任何一个上调用 `get_origin()` 将返回原始的 `ParamSpec` :

>>>

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

3.10 新版功能.

*class*typing.**TypeAliasType**(*name*, *value*, *, *type_params=()*)¶

通过 `type` 语句创建的类型别名的类型。

示例:

```
>>>
```

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

3.12 新版功能.

__name__¶

类型别名的名称:

```
>>>
```

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__¶

类型别名定义所在的模块名称:

```
>>>
```

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__¶

类型别名的类型形参，或者如果别名不属于泛型则为一个空元组:

>>>

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__¶

类型别名的值。 它将被 惰性求值，因此别名定义中使用的名称将直到 `__value__` 属性被访问时才会被解析:

>>>

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

其他特殊指令¶

这些函数和类不应被直接用作标注。 它们的设计目标是作为创建和声明类型的构件。

*class*typing.**NamedTuple**¶

`collections.namedtuple()` 的类型版本。

用法：

```
class Employee(NamedTuple):
    name: str
    id: int
```

这相当于：

```
Employee = collections.namedtuple('Employee', ['name',
'id'])
```

为字段提供默认值，要在类体内赋值：

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

带默认值的字段必须在不带默认值的字段后面。

由此产生的类有一个额外的属性 `__annotations__` ，给出一个 dict ，将字段名映射到字段类型。（字段名在 `_fields` 属性中，默认值在 `_field_defaults` 属性中，这两者都是 `namedtuple()` API 的一部分。）

`NamedTuple` 子类也支持文档字符串与方法：

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` 子类也可以为泛型：

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

反向兼容用法：

```
# For creating a generic NamedTuple on Python 3.11 or lower
class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id',
int)])
```

在 3.6 版更改: 添加了对 **PEP 526** 中变量注解句法的支持。

在 3.6.1 版更改: 添加了对默认值、方法、文档字符串的支持。

在 3.8 版更改: `_field_types` 和 `__annotations__` 属性现已使用常规字典，不再使用 `OrderedDict` 实例。

在 3.9 版更改: 移除了 `_field_types` 属性， 改用具有相同信息，但更标准的 `__annotations__` 属性。

在 3.11 版更改: 添加对泛型命名元组的支持。

*class*typing.**NewType**(*name*, *tp*)¶

用于创建低开销的 独有类型 的辅助类。

`NewType` 将被类型检查器视为一个独有类型。 但是，在运行时，调用 `NewType` 将原样返回其参数。

用法:

```
UserId = NewType('UserId', int)  # Declare the NewType
"UserId"
first_user = UserId(1)  # "UserId" returns the argument
unchanged at runtime
```

__module__¶

新类型定义所在的模块。

__name__¶

新类型的名称。

__supertype__¶

新类型所基于的类型。

3.5.2 新版功能.

在 3.10 版更改: `NewType` 现在是一个类而不是函数。

***class*typing.Protocol(*Generic*)¶**

协议类的基类。

协议类是这样定义的:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

这些类主要与静态类型检查器搭配使用，用来识别结构子类型（静态鸭子类型），例如:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

请参阅 **PEP 544** 了解详情。 使用 `runtime_checkable()` 装饰的协议类（稍后将介绍）可作为只检查给定属性是否存在，而忽略其类型签名的简单的运行时协议。

Protocol 类可以是泛型，例如:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

在需要兼容 Python 3.11 或更早版本的代码中，可以这样编写泛型协议:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

3.8 新版功能.

@typing.runtime_checkable¶

用于把 Protocol 类标记为运行时协议。

该协议可以与 `isinstance()` 和 `issubclass()` 一起使用。应用于非协议的类时，会触发 `TypeError` 。该指令支持简易结构检查，与 `collections.abc` 的 `Iterable` 非常类似，只擅长做一件事。 例如:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

备注

`runtime_checkable()` 将只检查所需方法或属性是否存在，而不检查它们的类型签名或类型。 例如， `ssl.SSLObject` 是一个类，因此它通过了针对 Callable 的 `issubclass()` 检查。 但是， `ssl.SSLObject.__init__` 方法的存在只是引发 `TypeError` 并附带更具信息量的消息，因此它无法调用 (实例化) `ssl.SSLObject` 。

备注

针对运行时可检查协议的 `isinstance()` 检查相比针对非协议类的 `isinstance()` 检查可能会惊人的缓慢。 请考虑在性能敏感的代码中使用替代性写法如 `hasattr()` 调用进行结构检查。

3.8 新版功能.

在 3.12 版更改: 现在 `isinstance()` 的内部实现对于运行时可检查协议的检查会使用 `inspect.getattr_static()` 来查找属性 (在之前版本中，会使用 `hasattr()`)。 因此，在 Python 3.12+ 上一些以前被认为是运行时可检查协议的实例的对象可能不再被认为是该协议的实例，反之亦反。 大多数用户不太可能受到这一变化的影响。

在 3.12 版更改: 一旦类被创建则运行时可检查协议的成员就会被视为在运行时“已冻结”。 在运行时可检查协议上打上猴子补丁属性仍然有效，但不会影响将对象与协议进行比较的 `isinstance()` 检查。 请参阅 "Python 3.12 有什么新变化 了解更多细节。

*class*typing.**TypedDict**(*dict*)¶

把类型提示添加至字典的特殊构造器。在运行时，它是纯 `dict` 。

`TypedDict` 声明一个字典类型，该类型预期所有实例都具有一组键集，其中，每个键都与对应类型的值关联。运行时不检查此预期，而是由类型检查器强制执行。用法如下：

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type
check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2,
label='first')
```

为了在不支持 **PEP 526** 的旧版 Python 中使用此特性，`TypedDict` 支持两种额外的等价语法形式:

- 使用字面量 `dict` 作为第二个参数:

  ```
  Point2D = TypedDict('Point2D', {'x': int, 'y': int,
  'label': str})
  ```

- 使用关键字参数:

  ```
  Point2D = TypedDict('Point2D', x=int, y=int, label=str)
  ```

从 3.11 版起不建议使用，将在 3.13 版中移除: 使用关键字的语法在 3.11 中被弃用，并且会于 3.13 被移除。同时，该语法可能不被静态类型检查器支持。

当任何一个键不是有效的 标识符 时，例如因为它们是关键字或包含连字符，也应该使用函数式语法。例子:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
```

```
    x-y: int  # name with hyphens

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
```

默认情况下，所有的键都必须出现在一个 `TypedDict` 中。 可以使用 `NotRequired` 将单独的键标记为非必要的:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label':
NotRequired[str]})
```

这意味着一个 `Point2D` `TypedDict` 可以省略 `label` 键。

也可以通过全部指定 `False` 将所有键都标记为默认非必要的:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int},
total=False)
```

这意味着一个 `Point2D` `TypedDict` 可以省略任何一个键。 类型检查器只需要支持一个字面的 `False` 或 `True` 作为 `total` 参数的值。 `True` 是默认的，它使类主体中定义的所有项目都是必需的。

一个 `total=False` `TypedDict` 中单独的键可以使用 `Required` 标记为必要的:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# Alternative syntax
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

一个 `TypedDict` 类型有可能使用基于类的语法从一个或多个其他 `TypedDict` 类型继承。用法:

```
class Point3D(Point2D):
    z: int
```

`Point3D` 有三个项目: `x` , `y` 和 `z` 。 其等价于定义:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

`TypedDict` 不能从非 `TypedDict` 类继承，除了 `Generic` 。 例如:

```
class X(TypedDict):
    x: int
```

```
class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError
```

`TypedDict` 也可以为泛型的：

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

要创建与 Python 3.11 或更低版本兼容的泛型 `TypedDict`，请显式地从 `Generic` 继承：

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

`TypedDict` 可以通过注解字典（参见 对象注解属性的最佳实践 了解更多关于注解的最佳实践）、 `__total__` 、 `__required_keys__` 和 `__optional_keys__` 进行内省。

__total__¶

`Point2D.__total__` 给出了 `total` 参数的值。 例如：

```
>>>
```

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

该属性 *只是* 反映传给当前 `TypedDict` 类的``total`` 参数的值，而不反映这个类在语义上是否完整。 例如，一个 `__total__` 被设为 True 的 `TypedDict` 可能有用 `NotRequired` 标记的键，或者它可能继承自另一个设置了 `total=False` 的 `TypedDict` 。 因此，使用 `__required_keys__` 和 `__optional_keys__` 进行内省通常会更好。

__required_keys__¶

3.9 新版功能.

__optional_keys__¶

`Point2D.__required_keys__` 和 `Point2D.__optional_keys__` 返回分别包含必要的和非必要的键的 `frozenset` 对象。

标记为 `Required` 的键总是会出现在 `__required_keys__` 中而标记为 `NotRequired` 的键总是会出现在 `__optional_keys__` 中。

为了向下兼容 Python 3.10 及更老的版本，还可以使用继承机制在同一个 `TypedDict` 中同时声明必要和非必要的键。 这是通过声明一个具有 `total` 参数值的 `TypedDict` 然后在另一个 `TypedDict` 中继承它并使用不同的 `total` 值来实现的：

```
>>>
```

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

3.9 新版功能.

备注

如果使用了 `from __future__ import annotations` 或者如果以字符串形式给出标注，那么标注不会在定义 `TypedDict` 时被求值。 因此，`__required_keys__` 和 `__optional_keys__` 所依赖的运行时内省可能无法正常工作，这些属性的值也可能不正确。

更多示例与 `TypedDict` 的详细规则，详见 **PEP 589**。

3.8 新版功能.

在 3.11 版更改: 增加了对将单独的键标记为 `Required` 或 `NotRequired` 的支持。 参见 **PEP 655**。

在 3.11 版更改: 添加对泛型 `TypedDict` 的支持。

协议¶

下列协议由 typing 模块提供并已全被装饰为 `可在运行时检查的` 。

*class*typing.**SupportsAbs**¶

一个抽象基类，含一个抽象方法 `__abs__` ，该方法与其返回类型协变。

*class*typing.**SupportsBytes**¶

一个抽象基类，含一个抽象方法 `__bytes__` 。

*class*typing.**SupportsComplex**¶

一个抽象基类，含一个抽象方法 `__complex__` 。

*class*typing.**SupportsFloat**¶

一个抽象基类，含一个抽象方法 `__float__` 。

*class*typing.**SupportsIndex**¶

一个抽象基类，含一个抽象方法 `__index__` 。

3.8 新版功能.

*class*typing.**SupportsInt**¶

一个抽象基类，含一个抽象方法 `__int__` 。

*class*typing.**SupportsRound**¶

一个抽象基类，含一个抽象方法 `__round__` ，该方法与其返回类型协变。

与 IO 相关的抽象基类¶

*class*typing.**IO**¶

*class*typing.**TextIO**¶

*class*typing.**BinaryIO**¶

泛型 `IO[AnyStr]` 及其子类 `TextIO(IO[str])` 、`BinaryIO(IO[bytes])` 表示 I/O 流——例如 `open()` 返回的对象——的类型。

函数与装饰器¶

typing.**cast**(*typ*, *val*)¶

把一个值转换为指定的类型。

这会把值原样返回。对类型检查器而言这代表了返回值具有指定的类型，在运行时我们故意没有设计任何检查（我们希望让这尽量快）。

typing.**assert_type**(*val*, *typ*, */*)¶

让静态类型检查器确认 *val* 具有推断为 *typ* 的类型。

在运行时这将不做任何事：它会原样返回第一个参数而没有任何检查或附带影响，无论参数的实际类型是什么。

当静态类型检查器遇到对 `assert_type()` 的调用时，如果该值不是指定的类型则会报错:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is
`str`
    assert_type(name, int)  # type checker error
```

此函数适用于确保类型检查器对脚本的理解符合开发者的意图:

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands
our function
    assert_type(arg, int)
```

3.11 新版功能.

typing.assert_never(*arg*, */*)¶

让静态类型检查器确认一行代码是不可达的。

示例：

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

在这里，标注允许类型检查器推断最后一种情况永远不会执行，因为 `arg` 要么是 `int` 要么是 `str` ，而这两种选项都已被之前的情况覆盖了。

如果类型检查器发现对 `assert_never()` 的调用是可达的，它将报告一个错误。 举例来说，如果 `arg` 的类型标注改为 `int | str | float` ，则类型检查器将报告一个错误指出 `unreachable` 为 `float` 类型。 对于通过类型检查的 `assert_never` 调用，参数传入的推断类型必须为兜底类型 `Never` ，而不能为任何其他类型。

在运行时，如果调用此函数将抛出一个异常。

参见

Unreachable Code and Exhaustiveness Checking 有更多关于使用静态类型进行穷尽性检查的信息。

3.11 新版功能.

typing.**reveal_type**(*obj*, */*)¶

让静态类型检查器显示推测的表达式类型。

当静态类型检查器遇到一个对此函数的调用时，它将发出带有所推测参数类型的诊断信息。 例如:

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

这在你想要调试你的类型检查器如何处理一段特定代码时很有用处。

在运行时，此函数会将其参数类型打印到 `sys.stderr` 并不加修改地返回该参数 (以允许该调用在表达式中使用):

```
x = reveal_type(1)  # prints "Runtime type is int"
print(x)  # prints "1"
```

请注意在运行时类型可能不同于类型静态检查器所推测的类型（明确程度可能更高也可能更低）。

大多数类型检查器都能在任何地方支持 `reveal_type()` ，即使并未从 `typing` 导入该名称。 不过，从 `typing` 导入该名称将允许你的代码在运行时不会出现运行时错误并能更清晰地传递意图。

3.11 新版功能.

@typing.dataclass_transform(*, *eq_default=True*, *order_default=False*, *kw_only_default=False*, *frozen_default=False*, *field_specifiers=()*, ***kwargs*)¶

将一个对象标记为提供类似 `dataclass` 行为的装饰器。

`dataclass_transform` 可被用于装饰类、元类或本身为装饰器的函数。 使用 `@dataclass_transform()` 将让静态类型检查器知道被装饰的对象会执行以类似 `@dataclasses.dataclass` 的方式来转换类的运行时"魔法"。

装饰器函数使用方式的例子:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
    id: int
    name: str
```

在基类上:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

在元类上:

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

上面定义的 `CustomerModel` 类将被类型检查器视为类似于使用 `@dataclasses.dataclass` 创建的类。 例如，类型检查器将假定这些类具有接受 `id` 和 `name` 的 `__init__` 方法。

被装饰的类、元类或函数可以接受以下布尔值参数，类型检查器将假定它们具有与 `@dataclasses.dataclass` 装饰器相同的效果: `init` , `eq` , `order` , `unsafe_hash` , `frozen` , `match_args` , `kw_only` 和 `slots` 。 这些参数的值 (`True` 或 `False`) 必须可以被静态地求值。

传给 `dataclass_transform` 装饰器的参数可以被用来定制被装饰的类、元类或函数的默认行为:

参数

- **eq_default** (*bool*) -- 指明如果调用方省略了 `eq` 形参则应将其假定为 `True` 还是 `False` 。 默认为 `True` 。
- **order_default** (*bool*) -- 指明如果调用方省略了 `order` 形参则应将其假定为 `True` 还是 `False` 。 默认为 `False` 。
- **kw_only_default** (*bool*) -- 指明如果调用方省略了 `kw_only` 形参则应将其假定为 `True` 还是 `False` 。 默认为 `False` 。
- **frozen_default** (*bool*) -- 指明如果调用方省略了 `frozen` 形参则应将其假定为 `True` 还是 `False` 。 默认为 `False` 。 .. versionadded::

3.12

- **field_specifiers** (*tuple[Callable[..., Any], ...]*) -- 指定一个受支持的类或描述字段的函数的静态列表，类似于 `dataclasses.field()` 。 默认为 `()` 。

- ****kwargs** (*Any*) -- 接受任何其他关键字以便允许可能的未来扩展。

类型检查器能识别下列字段设定器的可选形参:

形参名称	描述
`init`	指明字段是否应当被包括在合成的 `__init__` 方法中。 如果未指明，则 `init` 默认为 `True` 。
`default`	为字段提供默认值。
`default_factory`	提供一个返回字段默认值的运行时回调。 如果 `default` 或 `default_factory` 均未指定，则会假定字段没有默认值而在类被实例化时必须提供一个值。
`factory`	字段说明符上 `default_factory` 形参的别名。

形参名称	描述
`kw_only`	指明字段是否应被标记为仅限关键字的。 如为 `True` ，字段将是仅限关键字的。 如为 `False` ，它将不是仅限关键字的。 如未指明，则将使用以 `dataclass_transform` 装饰的对象的 `kw_only` 形参的值，或者如果该值也未指明，则将使用 `dataclass_transform` 上 `kw_only_default` 的值。
`alias`	提供字段的替代名称。 该替代名称会被用于合成的 `__init__` 方法。

字段设定器的可识别形参¶

在运行时，该装饰器会将其参数记录到被装饰对象的 `__dataclass_transform__` 属性。 它没有其他的运行时影响。

更多细节请参见 **PEP 681**。

3.11 新版功能.

@typing.overload¶

用于创建重载函数和方法的装饰器。

`@overload` 装饰器允许描述支持多参数类型不同组合的函数和方法。 一系列以 `@overload` 装饰的定义必须带上恰好一个非 `@overload` 装饰的定义（用于同一个函数/方法）。

以 `@overload` 装饰的定义仅对类型检查器有用，因为它们将被非 `@overload` 装饰的定义覆盖。 与此同时，非 `@overload` 装饰的定义将在

运行时使用但应被类型检查器忽略。 在运行时，直接调用以 `@overload` 装饰的函数将引发 `NotImplementedError` 。

一个提供了比使用联合或类型变量更精确的类型的重载的示例：

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # actual implementation goes here
```

请参阅 **PEP 484** 了解更多细节以及与其他类型语义的比较。

在 3.11 版更改: 现在可以使用 `get_overloads()` 在运行时内省有重载的函数。

typing.**get_overloads**(*func*)¶

为 *func* 返回以 `@overload` 装饰的定义的序列。

func 是用于实现过载函数的函数对象。 例如，根据文档中为 `@overload` 给出的 `process` 定义， `get_overloads(process)` 将为所定义的三个过载函数返回由三个函数对象组成的序列。 如果在不带过载的函数上调用， `get_overloads()` 将返回一个空序列。

`get_overloads()` 可被用来在运行时内省一个过载函数。

3.11 新版功能.

typing.**clear_overloads**()¶

清空内部注册表中所有已注册的重载。

这可用于回收注册表所使用的内存。

3.11 新版功能.

@typing.**final**¶

表示最终化方法和最终化类的装饰器。

以 `@final` 装饰一个方法将向类型检查器指明该方法不可在子类中被重载。
以 `@final` 装饰一个类表示它不可被子类化。

例如:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type
checker
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

这些属性没有运行时检查。详见 **PEP 591**。

3.8 新版功能.

在 3.11 版更改: 该装饰器现在将尝试在被装饰的对象上设置 `__final__` 属性为 `True` 。 这样，可以在运行时使用 `if getattr(obj, "__final__", False)` 这样的检查来确定对象 `obj` 是否已被标记为终结。 如果被装饰的对象不支持设置属性，该装饰器将不加修改地返回对象而不会引发异常。

@typing.no_type_check¶

标明注解不是类型提示的装饰器。

此作用方式类似于类或函数的 decorator。 对于类，它将递归地应用到该类中定义的所有方法和类（但不包括在其超类或子类中定义的方法）。 类型检查器将忽略带有此装饰器的函数或类的所有标注。

`@no_type_check` 将原地改变被装饰的对象。

@typing.no_type_check_decorator¶

让其他装饰器具有 `no_type_check()` 效果的装饰器。

本装饰器用 `no_type_check()` 里的装饰函数打包其他装饰器。

@typing.override¶

该装饰器指明子类中的某个方法是重载超类中的方法或属性。

如果一个以 `@override` 装饰的方法实际未重载任何东西则类型检查器应当报告错误。 这有助于防止当基类发生修改而子类未进行相应修改而导致的问题。

例如:

```
class Base:
    def log_status(self) -> None:
```

```
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides
Base.log_status
        ...

    @override
    def done(self) -> None:  # Error reported by type
checker
        ...
```

没有对此特征属性的运行时检查。

该装饰器将尝试在被装饰的对象上设置 `__override__` 属性为 `True`。这样，可以在运行时使用 `if getattr(obj, "__override__", False)` 这样的检查来确定对象 `obj` 是否已被标记为重载。如果被装饰的对象不支持设置属性，该装饰器将不加修改地返回对象而不会引发异常。

更多细节参见 **PEP 698**。

3.12 新版功能.

@typing.type_check_only¶

将类或函数标记为在运行时不可用的装饰器。

在运行时，该装饰器本身不可用。实现返回的是私有类实例时，它主要是用于标记在类型存根文件中定义的类。

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
```

```
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

注意，建议不要返回私有类实例，最好将之设为公共类。

内省辅助器¶

typing.**get_type_hints**(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)¶

返回函数、方法、模块、类对象的类型提示的字典。

这往往与 `obj.__annotations__` 相同。 此外，编码为字符串字面值的前向引用是通过在 `globals` 与 `locals` 命名空间中执行求值来处理的。 对于一个类 `C` ，则返回一个由所有 `__annotations__` 与 `C.__mro__` 逆序合并所构建的字典。

本函数会递归地将所有 `Annotated[T, ...]` 替换为 `T` ，除非 `include_extras` 被设为 `True` (请参阅 `Annotated` 了解详情)。 例如:

```
class Student(NamedTuple):
    name: Annotated[str, 'some marker']

assert get_type_hints(Student) == {'name': str}
assert get_type_hints(Student, include_extras=False) ==
{'name': str}
assert get_type_hints(Student, include_extras=True) == {
    'name': Annotated[str, 'some marker']
}
```

备注

`get_type_hints()` 在导入的 类型别名 中不工作，包括前向引用。启用注解的延迟评估（ **PEP 563** ）可能会消除对大多数前向引用的需要。

在 3.9 版更改: 增加了作为 **PEP 593** 组成部分的 `include_extras` 形参。 请参阅 `Annotated` 文档了解详情。

在 3.11 版更改: 在之前，如果设置了等于 `None` 的默认值则会为函数和方法标注添加 `Optional[t]` 。 现在标注将被不加修改地返回。

typing.get_origin(*tp*)¶

获取一个类型的不带下标的版本：对于 `X[Y, Z, ...]` 形式的类型对象将返回 `X` 。

如果 `X` 是一个内置类型或 `collections` 类在 typing 模块中的别名，它将被正规化为原始的类。 如果 `X` 是 `ParamSpecArgs` 或 `ParamSpecKwargs` 的实例，则返回下层的 `ParamSpec` 。 对于不受支持的对象将返回 `None` 。

示例：

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

3.8 新版功能.

typing.get_args(*tp*)¶

获取已执行所有下标的类型参数：对于 `X[Y, Z, ...]` 形式的类型对象将返回 `(Y, Z, ...)` 。

如果 `X` 是一个并集或是包含在另一个泛型类型中的 `Literal`，则 `(Y, Z, ...)` 的顺序可能因类型缓存而与原始参数 `[Y, Z, ...]` 存在差异。对于不受支持的对象将返回 `()`。

示例：

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

3.8 新版功能.

typing.is_typeddict(*tp*)¶

检查一个类型是否为 `TypedDict`。

例如：

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

3.10 新版功能.

***class*typing.ForwardRef**¶

用于字符串前向引用的内部类型表示的类。

例如， `List["SomeClass"]` 会被隐式转换为 `List[ForwardRef("SomeClass")]` 。 `ForwardRef` 不应由用户来实例化，但可以由内省工具使用。

备注

PEP 585 泛型类型例如 `list["SomeClass"]` 将不会被隐式地转换为 `list[ForwardRef("SomeClass")]` 因而将不会自动解析为 `list[SomeClass]` 。

3.7.4 新版功能.

常量¶

typing.TYPE_CHECKING¶

会被第 3 方静态类型检查器假定为 `True` 的特殊常量。 在运行时将为 `False` 。

用法:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

第一个类型注解必须用引号标注，才能把它当作“前向引用”，从而在解释器运行时中隐藏 `expensive_mod` 引用。局部变量的类型注释不会被评估，因此，第二个注解不需要用引号引起来。

备注

使用 `from __future__ import` 时，函数定义时不处理注解， 而是把注解当作字符串存在 `__annotations__` 里，这样就不必为注解使用引号。（详见 **PEP 563**）。

3.5.2 新版功能.

已弃用的别名¶

本模块给标准库中已有的类定义了许多别名，这些别名现已不再建议使用。起初 typing 模块包含这些别名是为了支持用 `[]` 来参数化泛型类。然而，在 Python 3.9 中，对应的已有的类也支持了 `[]` (参见 **PEP 585**)，因此这些别名了就成了多余的了。

这些多余的类型从 Python 3.9 起被弃用。然而，虽然它们可能会在某一时刻被移除，但目前还没有移除它们的计划。因此，解释器目前不会对这些别名发出弃用警告。

如果在某一时刻决定了要移除这些别名，解释器将在比正式移除提前至少两个发行版本发出弃用警告。但至少在 Python 3.14 之前，这些别名会保证保留在 typing 模块中且不引发弃用警告。

如果被类型检查器检查的程序旨在运行于 Python 3.9 或更高版本，则鼓励类型检查器标记出这些不建议使用的类型。

内置类型的别名¶

*class*typing.**Dict**(*dict, MutableMapping[KT, VT]*)¶

`dict` 的已弃用的别名。

请注意，要注解参数，更推荐使用 `Mapping` 这样的抽象容器类型，而不是使用 `dict` 或者 `typing.Dict` 。

该类型用法如下：

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

3.9 版后已移除: `builtins.dict` 现在支持下标操作（`[]`）。 参见 **PEP 585** 和 GenericAlias 类型。

***class*typing.List(*list, MutableSequence[T]*)¶**

`list` 的已弃用的别名。

请注意，要注解参数，更推荐使用 `Sequence` 或者 `Iterable` 这样的抽象容器类型，而不是使用 `list` 或者 `typing.List` 。

该类型用法如下：

```
def vec2[T: (int, float)](x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives[T: (int, float)](vector: Sequence[T]) ->
List[T]:
    return [item for item in vector if item > 0]
```

3.9 版后已移除: `builtins.list` 现在支持下标操作（`[]`）。 参见 **PEP 585** 和 GenericAlias 类型。

***class*typing.Set(*set, MutableSet[T]*)¶**

`builtins.set` 的已弃用的别名。

请注意，要注解参数，更推荐使用 `AbstractSet` 这样的抽象容器类型，而不是使用 `set` 或者 `typing.Set` 。

3.9 版后已移除: `builtins.set` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)¶

`builtins.frozenset` 的已弃用的别名。

3.9 版后已移除: `builtins.frozenset` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

typing.**Tuple**¶

`tuple` 的已弃用的别名。

`tuple` 和 `Tuple` 是类型系统中的特例；更多详细信息请参见 标注元组。

3.9 版后已移除: `builtins.tuple` 现在支持下标操作（ `[]` ）。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Type**(*Generic[CT_co]*)¶

`type` 的已弃用的别名。

有关在类型注解中使用 `type` 或 `typing.Type` 的详细信息，请参阅 类对象的类型 。

3.5.2 新版功能.

3.9 版后已移除: `builtins.type` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

`collections` 中的类型的别名。¶

*class*typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)¶

`collections.defaultdict` 的已弃用的别名。

3.5.2 新版功能.

3.9 版后已移除: `collections.defaultdict` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)¶

`collections.OrderedDict` 的已弃用的别名。

3.7.2 新版功能.

3.9 版后已移除: `collections.OrderedDict` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)¶

`collections.ChainMap` 的已弃用的别名。

3.5.4 新版功能.

3.6.1 新版功能.

3.9 版后已移除: `collections.ChainMap` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Counter**(*collections.Counter, Dict[T, int]*)¶

`collections.Counter` 的已弃用的别名。

3.5.4 新版功能.

3.6.1 新版功能.

3.9 版后已移除: `collections.Counter` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

***class*typing.Deque(*deque, MutableSequence[T]*)¶**

`collections.deque` 的已弃用的别名。

3.5.4 新版功能.

3.6.1 新版功能.

3.9 版后已移除: `collections.deque` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

其他具体类型的别名¶

> *从 3.8 版起不建议使用，将在 3.13 版中移除:* `typing.io` 命名空间被弃用并将被删除。 这些类型应该被直接从 `typing` 导入。

***class*typing.Pattern¶**
***class*typing.Match¶**

`re.compile()` 和 `re.match()` 的返回类型的已弃用的别名。

这些类型（与对应的函数）是 `AnyStr` 上的泛型。 `Pattern` 可以被特化为 `Pattern[str]` 或 `Pattern[bytes]` ； `Match` 可以被特化为 `Match[str]` 或 `Match[bytes]` 。

从 3.8 版起不建议使用，将在 3.13 版中移除: `typing.re` 命名空间被弃用并将被删除。 这些类型应该被直接从 `typing` 导入。

3.9 版后已移除: `re` 模块中的 `Pattern` 与 `Match` 类现已支持 `[]` 。详见 **PEP 585** 与 GenericAlias 类型。

*class*typing.**Text**¶

`str` 的已弃用的别名。

`Text` 被用来为 Python 2 代码提供向上兼容的路径：在 Python 2 中，`Text` 是 `unicode` 的别名。

使用 `Text` 时，值中必须包含 unicode 字符串，以兼容 Python 2 和 Python 3：

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

3.5.2 新版功能.

3.11 版后已移除: Python 2 已不再受支持，并且大部分类型检查器也都不再支持 Python 2 代码的类型检查。 目录还没有移除该别名的计划，但建议用户使用 `str` 来代替 `Text` 。

`collections.abc` 中容器 ABC 的别名¶

*class*typing.**AbstractSet**(*Collection[T_co]*)¶

`collections.abc.Set` 的已弃用的别名。

3.9 版后已移除: `collections.abc.Set` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**ByteString**(*Sequence[int]*)¶

该类型代表了 `bytes` 、 `bytearray` 、 `memoryview` 等字节序列类型。

从 3.9 版起不建议使用，将在 3.14 版中移除: 首选 `collections.abc.Buffer`，或是 `bytes | bytearray | memoryview` 这样的并集。

*class*typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)¶

`collections.abc.Collection` 的已弃用的别名。

3.6.0 新版功能.

3.9 版后已移除: `collections.abc.Collection` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Container**(*Generic[T_co]*)¶

`collections.abc.Container` 的已弃用的别名。

3.9 版后已移除: `collections.abc.Container` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**ItemsView**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)¶

`collections.abc.ItemsView` 的已弃用的别名。

3.9 版后已移除: `collections.abc.ItemsView` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**KeysView**(*MappingView, AbstractSet[KT_co]*)¶

`collections.abc.KeysView` 的已弃用的别名。

3.9 版后已移除: `collections.abc.KeysView` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Mapping**(*Collection[KT], Generic[KT, VT_co]*)¶

`collections.abc.Mapping` 的已弃用的别名。

该类型用法如下：

```
def get_position_in_index(word_list: Mapping[str, int],
word: str) -> int:
    return word_list[word]
```

3.9 版后已移除: `collections.abc.Mapping` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**MappingView**(*Sized*)¶

`collections.abc.MappingView` 的已弃用的别名。

3.9 版后已移除: `collections.abc.MappingView` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**MutableMapping**(*Mapping[KT, VT]*)¶

`collections.abc.MutableMapping` 的已弃用的别名。

3.9 版后已移除: `collections.abc.MutableMapping` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**MutableSequence**(*Sequence[T]*)¶

`collections.abc.MutableSequence` 的已弃用的别名。

3.9 版后已移除: `collections.abc.MutableSequence` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**MutableSet**(*AbstractSet[T]*)¶

`collections.abc.MutableSet` 的已弃用的别名。

3.9 版后已移除: `collections.abc.MutableSet` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)¶

`collections.abc.Sequence` 的已弃用的别名。

3.9 版后已移除: `collections.abc.Sequence` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**ValuesView**(*MappingView, Collection[_VT_co]*)¶

`collections.abc.ValuesView` 的已弃用的别名。

3.9 版后已移除: `collections.abc.ValuesView` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

`collections.abc` 中异步 ABC 的别名¶

*class*typing.**Coroutine**(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)¶

`collections.abc.Coroutine` 的已弃用的别名。

类型变量的变化形式和顺序与 `Generator` 的相对应，例如:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined
elsewhere
x = c.send('hi')                   # Inferred type of 'x' is
```

```
list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is
int
```

3.5.3 新版功能.

3.9 版后已移除: `collections.abc.Coroutine` 现在支持下标操作（`[]`）。参见 **PEP 585** 和 GenericAlias 类型。

***class*typing.AsyncGenerator(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)¶**

`collections.abc.AsyncGenerator` 的已弃用的别名。

异步生成器可由泛型类型 `AsyncGenerator[YieldType, SendType]` 注解。例如：

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

与常规生成器不同，异步生成器不能返回值，因此没有 `ReturnType` 类型参数。与 `Generator` 类似，`SendType` 也属于逆变行为。

如果生成器只产生值，可将 `SendType` 设置为 `None`：

```
async def infinite_stream(start: int) -> AsyncGenerator[int,
None]:
    while True:
```

```
        yield start
        start = await increment(start)
```

此外，可用 `AsyncIterable[YieldType]` 或 `AsyncIterator[YieldType]` 注解生成器的返回类型：

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

3.6.1 新版功能.

3.9 版后已移除: `collections.abc.AsyncGenerator` 现在支持下标操作（`[]`）。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**AsyncIterable**(*Generic[T_co]*)¶

`collections.abc.AsyncIterable` 的已弃用的别名。

3.5.2 新版功能.

3.9 版后已移除: `collections.abc.AsyncIterable` 现在支持下标操作（`[]`）。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**AsyncIterator**(*AsyncIterable[T_co]*)¶

`collections.abc.AsyncIterator` 的已弃用的别名。

3.5.2 新版功能.

3.9 版后已移除: `collections.abc.AsyncIterator` 现在支持下标操作（`[]`）。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Awaitable**(*Generic[T_co]*)¶

`collections.abc.Awaitable` 的已弃用的别名。

3.5.2 新版功能.

3.9 版后已移除: `collections.abc.Awaitable` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

`collections.abc` 中其他 ABC 的别名¶

*class*typing.**Iterable**(*Generic[T_co]*)¶

`collections.abc.Iterable` 的已弃用的别名

3.9 版后已移除: `collections.abc.Iterable` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Iterator**(*Iterable[T_co]*)¶

`collections.abc.Iterator` 的已弃用的别名。

3.9 版后已移除: `collections.abc.Iterator` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

typing.**Callable**¶

`collections.abc.Callable` 的已弃用的别名。

有关如何在类型标注中使用 `collections.abc.Callable` 和 `typing.Callable` 的详细信息请参阅 标注可调用对象。

3.9 版后已移除: `collections.abc.Callable` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

在 3.10 版更改: `Callable` 现在支持 `ParamSpec` 和 `Concatenate` 。详情见 **PEP 612**。

*class*typing.**Generator**(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)¶

`collections.abc.Generator` 的已弃用的别名。

生成器可以由泛型类型 `Generator[YieldType, SendType, ReturnType]` 注解。例如:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

注意，与 typing 模块里的其他泛型不同， `Generator` 的 `SendType` 属于逆变行为，不是协变行为，也是不变行为。

如果生成器只产生值，可将 `SendType` 与 `ReturnType` 设为 `None` :

```
def infinite_stream(start: int) -> Generator[int, None,
None]:
    while True:
        yield start
        start += 1
```

此外，还可以把生成器的返回类型注解为 `Iterable[YieldType]` 或 `Iterator[YieldType]` :

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
```

```
        yield start
        start += 1
```

3.9 版后已移除: `collections.abc.Generator` 现在支持下标操作 (`[]`)。参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Hashable**¶

`collections.abc.Hashable` 的已弃用的别名。

3.12 版后已移除: 请改为直接使用 `collections.abc.Hashable` 。

*class*typing.**Reversible**(*Iterable[T_co]*)¶

`collections.abc.Reversible` 的已弃用的别名。

3.9 版后已移除: `collections.abc.Reversible` 现在支持下标操作 (`[]`)。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**Sized**¶

`collections.abc.Sized` 的已弃用的别名。

3.12 版后已移除: 请改为直接使用 `collections.abc.Sized` 。

`contextlib` ABC 的别名¶

*class*typing.**ContextManager**(*Generic[T_co]*)¶

`contextlib.AbstractContextManager` 的已弃用的别名。

3.5.4 新版功能.

3.6.0 新版功能.

3.9 版后已移除: `contextlib.AbstractContextManager` 现在支持下标操作（`[]`）。 参见 **PEP 585** 和 GenericAlias 类型。

*class*typing.**AsyncContextManager**(*Generic[T_co]*)¶

`contextlib.AbstractAsyncContextManager` 的已弃用的别名。

3.5.4 新版功能.

3.6.2 新版功能.

3.9 版后已移除: `contextlib.AbstractAsyncContextManager` 现在 支持下标操作（`[]`）。 参见 **PEP 585** 和 GenericAlias 类型。

主要特性的弃用时间线¶

`typing` 的某些特性被弃用，并且可能在将来的 Python 版本中被移除。下表总结了主要的弃用特性。该表可能会被更改，而且并没有列出所有的弃用特性。

特性	弃用于	计划移除	PEP/问题
`typing.io` 和 `typing.re` 子模块	3.8	3.13	bpo-38291
标准容器的 `typing` 版本	3.9	未定（请参阅 已弃用的别名 了解详情）	**PEP 585**