4. 执行模型

4.1. 程序的结构

Python 程序是由代码块构成的。 *代码块* 是被作为一个单元来执行的一段 Python 程序文本。 以下几个都属于代码块：模块、函数体和类定义。 交互式输入的每条命令都是代码块。 一个脚本文件（作为标准输入发送给解释器或是作为命令行参数发送给解释器的文件）也是代码块。 一条脚本命令（通过 `-c` 选项在解释器命令行中指定的命令）也是代码块。 通过在命令行中使用 `-m` 参数作为最高层级脚本（即 `__main__` 模块）运行的模块也是代码块。 传递给内置函数 `eval()` 和 `exec()` 的字符串参数也是代码块。

代码块在 *执行帧* 中被执行。 一个帧会包含某些管理信息（用于调试）并决定代码块执行完成后应前往何处以及如何继续执行。

4.2. 命名与绑定

4.2.1. 名称的绑定

名称 用于指代对象。 名称是通过名称绑定操作来引入的。

下面的结构将名字绑定:

- 函数的正式参数，
- 类定义，
- 函数定义，
- 赋值表达式，
- 如果在一个赋值中出现，则为标识符的 目标 :
  - `for` 循环头，
  - 在 `with` 语句, `except` 子句, `except*` 子句，或格式化模式匹配的 as 模式的 `as` 之后，
  - 在结构模式匹配中的捕获模式
- `import` 语句。
- `type` 语句。
- 类型形参列表。

形式为 `from ... import *` 的 `import` 语句绑定所有在导入的模块中定义的名字，除了那些以下划线开头的名字。这种形式只能在模块级别上使用。

`del` 语句的目标也被视作一种绑定（虽然其实际语义为解除名称绑定）。

每条赋值或导入语句均发生于类或函数内部定义的代码块中，或是发生于模块层级（即最高层级的代码块）。

如果某个名称绑定在一个代码块中，则它就是该代码块的局部变量，除非声明为 `nonlocal` 或 `global`。 如果某个名称绑定在模块层级，则它就是全局变量。 （模块代码块的变量既是局部变量

又是全局变量。） 如果某个变量在一个代码块中被使用但不是在其中定义的，则它是 free variable。

每个在程序文本中出现的名称是指由以下名称解析规则所建立的对该名称的 *绑定*。

4.2.2. 名称的解析

作用域 定义了一个代码块中名称的可见性。 如果代码块中定义了一个局部变量，则其作用域包含该代码块。 如果定义发生于函数代码块中，则其作用域会扩展到该函数所包含的任何代码块，除非有某个被包含代码块引入了对该名称的不同绑定。

当一个名称在代码块中被使用时，会由包含它的最近作用域来解析。 对一个代码块可见的所有这种作用域的集合称为该代码块的 *环境*。

当一个名称完全找不到时，将会引发 `NameError` 异常。 如果当前作用域为函数作用域，且该名称指向一个局部变量，而此变量在该名称被使用的时候尚未绑定到特定值，将会引发 `UnboundLocalError` 异常。 `UnboundLocalError` 为 `NameError` 的一个子类。

如果一个代码块内的任何位置发生名称绑定操作，则代码块内所有对该名称的使用都会被视为对当前代码块的引用。 当一个名称在其被绑定前就在代码块内被使用时将会导致错误。 这个规则是很微妙的。 Python 缺少声明语法并且允许名称绑定操作发生于代码块内的任何位置。 一个代码块的局部变量可通过在整个代码块文本中扫描名称绑定操作来确定。 请参阅 UnboundLocalError 的 FAQ 条目 来获取示例。

如果 `global` 语句出现在一个代码块中，则所有对该语句所指定名称的使用都是在最高层级命名空间内对该名称绑定的引用。 名称在最高层级命名空间内的解析是通过搜索全局命名空间，也就是包含该代码块的模块的命名空间，以及内置命名空间即 `builtins` 模块的命名空间。 全局命名空间会先被搜索。 如果未在其中找到相应名称，将再搜索内置命名空间。 如果未在内置命名空间中找到相应名称，将在全局命名空间中创建新变量。 global 语句必须位于所有对其所列名称的使用之前。

`global` 语句与同一代码块中名称绑定具有相同的作用域。 如果一个自由变量的最近包含作用域中有一条 global 语句，则该自由变量也会被当作是全局变量。

`nonlocal` 语句会使得相应的名称指向之前在最近包含函数作用域中绑定的变量。 如果指定的名称不存在于任何包含函数作用域中则将在编译时引发 `SyntaxError`。 类型形参 不能使用 `nonlocal` 语句来重新绑定。

模块的作用域会在模块第一次被导入时自动创建。 一个脚本的主模块总是被命名为 `__main__`。

类定义代码块以及传给 `exec()` 和 `eval()` 的参数是名称解析的上下文中的特殊情况。 类定义是可能使用并定义名称的可执行语句。 这些引用遵循正常的名称解析规则，例外之处在于未绑定的局部变量会在全局命名空间中查找。 类定义的命名空间会成为该类的属性字典。 在类代码块中定义的名称的作用域会被限制在类代码块中；它不会扩展到方法的代码块中。 这包括推导式和生成器表达式，但不包括 标注作用域，因为它可以访问所包含的类作用域。 这意味着以下代码将会失败:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

但是，下面的代码将会成功:

```
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)  # <type 'A.Nested'>
```

4.2.3. 标注作用域

标注, 类型形参列表 和 type 语句引入了 *标注作用域*，其行为基本类似于函数作用域，但具有下文讨论的一些差异。

标注作用域将在下列情况中使用:

- 函数标注。
- 变量标注。
- 针对 泛型类型别名 的类型形参列表。
- 针对 泛型函数 的类型形参列表。 泛型函数的标注会在标注作用域内执行，但其默认值和装饰器则不会。
- 针对 泛型类 的类型形参列表。 泛型类的基类和关键字参数会在标注作用域内执行，但其装饰器则不会。
- 针对类型形参的绑定、约束和默认值 (惰性求值)。
- 类型别名的值 (惰性求值)。

标注作用域在以下几个方面不同于函数作用域:

- 标注作用域能够访问其所包含的类命名空间。 如果某个标注作用域紧接在一个类作用域之内，或是位于紧接一个类作用域的另一个标注作用域之内，则该标注作用域中的代码将能使用在该类作用域中定义的名称，就像它是在该类内部直接执行一样。 这不同于在类中定义的常规函数，后者无法访问在类作用域中定义的名称。
- 标注作用域中的表达式不能包含 yield, yield from, await 或 := 表达式。 (这些表达式在包含于标注作用域之内的其他作用域中则是允许的。)
- 在标注作用域中定义的名称不能在内部作用域中通过 nonlocal 语句来重新绑定。 这只包括类型形参，因为没有其他可以在标注作用域内部出现的语法元素能够引入新的名称。
- 虽然标注作用域具有一个内部名称，但该名称不会反映在作用域内定义的对象的 qualified name 中。 相反，这些对象的 __qualname__ 就像它们是定义在包含作用域中的对象一样。

> *Added in version 3.12:* 标注作用域是在 Python 3.12 中作为 **PEP 695** 的一部分引入的。

> *在 3.13 版本发生变更:* 标注作用域也被用于类型形参默认值，这是由 **PEP 696** 引入的。

> *在 3.14 版本发生变更:* 标注作用域现在也被用于标注，如 **PEP 649** 和 **PEP 749** 所说明的。

4.2.4. 惰性求值

大多数标注作用域采用 *惰性求值*。 这包括标注、通过 type 语句创建的类型别名的值，以及通过 类型形参语法 创建的类型变量的绑定、约束和默认值。 这意味着它们在类型别名或类型变量被创建

时、或带有标注的对象被创建时不会被求值。 作为替代行为，它们只有在需要时才会被求值，例如在一个类型别名上的 `__value__` 属性被访问的时候。

示例:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

此处的异常只有在类型别名的 `__value__` 属性或类型变量的 `__bound__` 属性被访问时才会被引发。

此行为主要适用于当创建类型别名或类型变量时对尚未被定义的类型进行引用。 例如，惰性求值将允许创建相互递归的类型别名:

```
from typing import Literal

type SimpleExpr = int | Parenthesized
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

被惰性求值的值是在 标记作用域 内进行求值的，这意味着出现在被惰性求值的值内部的名称的查找范围就相当于它们是在紧邻的作用域中被使用。

> *Added in version 3.12.*

4.2.5. 内置命名空间和受限的执行

用户不应该接触 `__builtins__`，严格说来它属于实现细节。 用户如果要重载内置命名空间中的值则应该 `import` `builtins` 并相应地修改该模块中的属性。

与一个代码块的执行相关联的内置命名空间实际上是通过在其全局命名空间中搜索名称 `__builtins__` 来找到的；这应该是一个字典或一个模块（在后一种情况下会使用该模块的字典）。默认情况下，当在 `__main__` 模块中时，`__builtins__` 就是内置模块 `builtins`；当在任何其他模块中时，`__builtins__` 则是 `builtins` 模块自身的字典的一个别名。

4.2.6. 与动态特性的交互

自由变量的名称解析发生于运行时而不是编译时。 这意味着以下代码将打印出 42:

```
i = 10
def f():
    print(i)
i = 42
f()
```

`eval()` 和 `exec()` 函数没有对完整环境的访问权限来解析名称。 名称可以在调用者的局部和全局命名空间中被解析。 自由变量的解析不是在最近包含命名空间中，而是在全局命名空间中。 [1] `exec()` 和 `eval()` 函数有可选参数用来重载全局和局部命名空间。 如果只指定一个命名空间，则它会同时作用于两者。

4.3. 异常

异常是中断代码块的正常控制流程以便处理错误或其他异常条件的一种方式。 异常会在错误被检测到的位置 *引发*，它可以被当前包围代码块或是任何直接或间接唤起发生错误的代码块的其他代码块所 *处理*。

Python 解析器会在检测到运行时错误（例如零作为被除数）的时候引发异常。 Python 程序也可以通过 `raise` 语句显式地引发异常。 异常处理是通过 `try` ... `except` 语句来指定的。 该语句的 `finally` 子句可被用来指定清理代码，它并不处理异常，而是无论之前的代码是否发生异常都会被执行。

Python 的错误处理采用的是"终止"模型：异常处理器可以找出发生了什么问题，并在外层继续执行，但它不能修复错误的根源并重试失败的操作（除非通过从顶层重新进入出错的代码片段）。

当一个异常完全未被处理时，解释器会终止程序的执行，或者返回交互模式的主循环。 无论是哪种情况，它都会打印栈回溯信息，除非是当异常为 `SystemExit` 的时候。

异常是通过类实例来标识的。 `except` 子句会依据实例的类来选择：它必须引用实例的类或是其所属的 非虚基类 。 实例可通过处理器被接收，并可携带有关异常条件的附加信息。

> **备注：** 异常消息不是 Python API 的组成部分。 其内容可能在 Python 升级到新版本时不经警告地发生改变，不应该被需要在多版本解释器中运行的代码所依赖。

另请参看 try 语句 小节中对 `try` 语句的描述以及 raise 语句 小节中对 `raise` 语句的描述。

4.4. 运行时组件

4.4.1. 通用计算模型

Python 的执行模型并非是在真空中运作的。 它存在于一台主机并通过该主机的运行时环境运行，包括其中的操作系统 (OS)，如果有具体操作系统的话。 当一个程序运行时，决定它如何在主机上运行的各个概念层大致是这样的：

主机
  进程 (全局资源)
    线程 (运行机器码)

每个进程代表一个在主机上运行的程序。 可以将每个进程本身视作其程序的数据部分。 可以将进程的各个线程视为程序的执行部分。 这一区别对于理解 Python 运行时的概念是很重要的。

进程作为数据部分，是程序运行所在的执行上下文。 它主要由主机分配给程序的资源集合组成，包括内存、信号、文件句柄、套接字以及环境变量等。

Processes are isolated and independent from one another. (The same is true for hosts.) The host manages the process' access to its assigned resources, in addition to coordinating between processes.

Each thread represents the actual execution of the program's machine code, running relative to the resources assigned to the program's process. It's strictly up to the host how and when that execution takes place.

From the point of view of Python, a program always starts with exactly one thread. However, the program may grow to run in multiple simultaneous threads. Not all hosts support multiple threads per process, but most do. Unlike processes, threads in a process are not isolated and independent from one another. Specifically, all threads in a process share all of the process' resources.

The fundamental point of threads is that each one does *run* independently, at the same time as the others. That may be only conceptually at the same time ("concurrently") or physically ("in parallel"). Either way, the threads effectively run at a non-synchronized rate.

> **备注:** That non-synchronized rate means none of the process' memory is guaranteed to stay consistent for the code running in any given thread. Thus multi-threaded programs must take care to coordinate access to intentionally shared resources. Likewise, they must take care to be absolutely diligent about not accessing any *other* resources in multiple threads; otherwise two threads running at the same time might accidentally interfere with each other's use of some shared data. All this is true for both Python programs and the Python runtime.
>
> The cost of this broad, unstructured requirement is the tradeoff for the kind of raw concurrency that threads provide. The alternative to the required discipline generally means dealing with non-deterministic bugs and data corruption.

4.4.2. Python Runtime Model

The same conceptual layers apply to each Python program, with some extra data layers specific to Python:

主机
  进程 (全局资源)
    Python global runtime (*state*)
      Python interpreter (*state*)
        thread (runs Python bytecode and "C-API")
          Python thread *state*

At the conceptual level: when a Python program starts, it looks exactly like that diagram, with one of each. The runtime may grow to include multiple interpreters, and each interpreter may grow to include multiple thread states.

> **备注:** A Python implementation won't necessarily implement the runtime layers distinctly or even concretely. The only exception is places where distinct layers are directly specified or

exposed to users, like through the `threading` module.

备注: The initial interpreter is typically called the "main" interpreter. Some Python implementations, like CPython, assign special roles to the main interpreter.

Likewise, the host thread where the runtime was initialized is known as the "main" thread. It may be different from the process' initial thread, though they are often the same. In some cases "main thread" may be even more specific and refer to the initial thread state. A Python runtime might assign specific responsibilities to the main thread, such as handling signals.

As a whole, the Python runtime consists of the global runtime state, interpreters, and thread states. The runtime ensures all that state stays consistent over its lifetime, particularly when used with multiple host threads.

The global runtime, at the conceptual level, is just a set of interpreters. While those interpreters are otherwise isolated and independent from one another, they may share some data or other resources. The runtime is responsible for managing these global resources safely. The actual nature and management of these resources is implementation-specific. Ultimately, the external utility of the global runtime is limited to managing interpreters.

In contrast, an "interpreter" is conceptually what we would normally think of as the (full-featured) "Python runtime". When machine code executing in a host thread interacts with the Python runtime, it calls into Python in the context of a specific interpreter.

备注: The term "interpreter" here is not the same as the "bytecode interpreter", which is what regularly runs in threads, executing compiled Python code.

In an ideal world, "Python runtime" would refer to what we currently call "interpreter". However, it's been called "interpreter" at least since introduced in 1997 (CPython:a027efa5b).

Each interpreter completely encapsulates all of the non-process-global, non-thread-specific state needed for the Python runtime to work. Notably, the interpreter's state persists between uses. It includes fundamental data like `sys.modules`. The runtime ensures multiple threads using the same interpreter will safely share it between them.

A Python implementation may support using multiple interpreters at the same time in the same process. They are independent and isolated from one another. For example, each interpreter has its own `sys.modules`.

For thread-specific runtime state, each interpreter has a set of thread states, which it manages, in the same way the global runtime contains a set of interpreters. It can have thread states for as many host threads as it needs. It may even have multiple thread states for the same host thread, though that isn't as common.

Each thread state, conceptually, has all the thread-specific runtime data an interpreter needs to operate in one host thread. The thread state includes the current raised exception and the thread's

Python call stack. It may include other thread-specific resources.

> **备注:** The term "Python thread" can sometimes refer to a thread state, but normally it means a thread created using the `threading` module.

Each thread state, over its lifetime, is always tied to exactly one interpreter and exactly one host thread. It will only ever be used in that thread and with that interpreter.

Multiple thread states may be tied to the same host thread, whether for different interpreters or even the same interpreter. However, for any given host thread, only one of the thread states tied to it can be used by the thread at a time.

Thread states are isolated and independent from one another and don't share any data, except for possibly sharing an interpreter and objects or other resources belonging to that interpreter.

Once a program is running, new Python threads can be created using the `threading` module (on platforms and Python implementations that support threads). Additional processes can be created using the `os`, `subprocess`, and `multiprocessing` modules. Interpreters can be created and used with the `interpreters` module. Coroutines (async) can be run using `asyncio` in each interpreter, typically only in a single thread (often the main thread).

备注

[1] 出现这样的限制是由于通过这些操作执行的代码在模块被编译的时候并不可用。