第8章. 防御式编程

如果一个程序存在语法错误，那么不论编译器还是解释器都会在正式执行该程序前报错。然而不存在语法错误的程序被执行后，依然可能突发一些对执行结果造成不利影响的情况，例如计算除法时发现除数为0，或者用户按下了Ctrl+C。这些情况是无法预知的，或者更准确地说，我们知道这些情况有可能发生，但却无法知道它们在何时发生。为了应对这类情况，所有现代编程语言都提供了“异常（exceptions）”这一机制，而利用该机制处理此类突发情况的技巧被称为“防御式编程（defensive programming）”。

8-1. 什么是异常

（教程：8.6、16.1.1）
（标准库：内置异常）

防御式编程的内在思想源自“防御式驾驶（defensive driving）”：在开车时必须牢记公路上可能发生任何事情，且永远不可能预判其他司机和行人将要做什么，因此必须自己承担起保护自己的责任，在其他人做出危险动作时也要避免发生交通事故。

类似的，编写计算机程序时也必须牢记程序被执行期间可能发生任何事情，且永远不可能预判用户将要做什么。当遇到某种突发情况时，计算机会通过抛出相应类型异常的方式来通知程序。在Python中，广义的异常指的是BaseException类或它的某个子类的实例。基类BaseException本身是可被实例化的，其语法为：

```
class BaseException(*args)
```

也就是说它的__init__可以接受任何基于位置对应的实际参数列表，而该实际参数列表会被生成的异常对象记录下来，此后可以通过异常对象的args属性以一个元组的形式取回，例如：

```
$ python3

>>> e1 = BaseException(1, 2, 3)
>>> e1.args
(1, 2, 3)
>>> e2 = BaseException('abc', 5.2, [True, False])
>>> e2.args
('abc', 5.2, [True, False])
>>>
```

args参数通常被传入一个代表错误消息的字符串。BaseException还提供了函数属性with_traceback()和add_note()，在本章后面会详细解释它们的用法。

Python解释器预定义了许多派生自BaseException的类，它们与BaseException在逻辑上都通过builtins模块定义，被统称为“内置异常（built-in exceptions）”。BaseException

的直接子类有4个，总结在表8-1中。从该表可以看出，SystemExit、KeyboardInterrupt和GeneratorExit是比较特殊的异常类型，严格来说并不能算突发情况，而都是在计划中的事件，很少需要改变处理它们的默认流程。本章重点讨论Exception类派生出的各种类，它们才是最典型的异常。

表8-1. BaseException的直接子类

类	说明
SystemExit	由sys.exit()抛出，其__init__只接收一个可选的参数，传入sys.exit()的arg参数，并被解读为退出状态。
KeyboardInterrupt	用户按下发出中断信号的键组合（默认为Ctrl+C）时抛出。
GeneratorExit	一个生成器或异步生成器关闭时抛出。
Exception	所有其他突发情况导致的异常的基类，具体含义由子类确定。

表8-2列出了Exception的所有直接子类。与迭代器相关的StopIteration异常会在第12章详细讨论。与异步迭代器相关的StopAsyncIteration异常会在第14章详细讨论。与弱引用相关的ReferenceError异常会在第15章详细讨论。与assert语句相关的AssertionError异常会在第16章详细讨论。其他异常会在本章详细讨论，其中Warning代表的警告将占用单独一节。

表8-2. Exception的直接子类

类	说明
StopIteration	由迭代器或生成器抛出，表示无法产生下一项。具有value实例属性。
StopAsyncIteration	由异步迭代器或异步生成器抛出，表示无法产生下一项。
ReferenceError	在通过弱引用访问某个已被垃圾回收机制销毁的对象时抛出。
AssertionError	assert语句失败时抛出。
Warning	所有代表警告的类的基类。
NameError	在名字空间中未能找到某个标识符时抛出。具有name实例属性。
AttributeError	访问一个属性失败时抛出。具有实力属性name和obj。
LookupError	容器的索引或键无效时抛出。
TypeError	当表达式中的操作数或函数调用的实际参数类型不合适时抛出。
ValueError	当表达式中的操作数或函数调用的实际参数类型合适但值不合适时抛出。
ArithmeticError	发生算术类错误时抛出。
ImportError	import语句失败时抛出。具有实力属性name和path。
SyntaxError	当通过import语句导入一个模块，或通过compile()伪编译一个字符串，或通过eval()或exec()执行一个字符串，并发现了语法错误时抛出。具有实例属性filename、lineno、offset、text、end_lineno和end_offset。
EOFError	input()直接读取到EOF，没有读取到其他有效数据时抛出。
BufferError	与缓冲区相关的操作无法执行时抛出。
MemoryError	物理内存耗尽但可通过删除一些对象来挽救时抛出。

类	说明
OSError	某个系统调用返回失败结果时抛出。具有实力属性errno、winerror、strerror、filename和filename2。
RuntimeError	监测到一个不归属于任何其他类型的错误时抛出。
SystemError	当Python解释器产生不算太严重的内部错误时抛出。

NameError异常表明某个标识符解析失败，实例化语法为：

```
class NameError(*args, name=None)
```

其仅关键字形式参数name被用于传入代表该标识符本身的字符串，这会被NameError异常的name属性引用。下面的例子创建了一个NameError异常：

```
$ python3

>>> e = NameError("name 'abc' is not defined", name='abc')
>>> e.args
("name 'abc' is not defined",)
>>> e.name
'abc'
>>>
```

特别的，UnboundLocalError是NameError的直接子类，既存在如下继承关系：

```
NameError
  └─ UnboundLocalError
```

UnboundLocalError异常特指在函数体内访问某个本地变量，但该变量并未绑定任何对象的情况。

AttributeError异常表明对某个属性的访问失败，这或者是由于指定的对象并不具有指定的属性，或者是由于试图设置一个只读属性。AttributeError的实例化语法为：

```
class AttributeError(*args, name=None, obj=None)
```

其仅关键字形式参数name和obj分别用于传入属性名和访问该属性的对象，而它们会分别被AttributeError异常的name属性和obj属性引用。

下面的例子创建了一个AttributeError异常：

```
$ python3

>>> x = 0
>>> e = AttributeError("'x' object has no attribute 'value'", name='value',
obj=x)
>>> e.args
("'x' object has no attribute 'value'",)
>>> e.name
'value'
>>> e.obj
0
>>>
```

LookupError异常表明在对容器对象进行抽取或切片操作时指定了无效的索引或键，其实例化语法不包括任何仅关键字形式参数。LookupError是IndexError和KeyError的直接基类，即存在如下继承关系：

```
LookupError
  ├─ IndexError
  └─ KeyError
```

IndexError异常特指索引无效的情况；KeyError异常特指键无效的情况。

TypeError异常和ValueError异常都与表达式中不合适的操作数，或者函数调用中不合适的实际参数有关。两者的区别在于，TypeError异常表明不合适的是操作数/实际参数的类型，而ValueError异常表明不合适的是操作数/实际参数的值。举例来说，除法运算要求两个操作数都是数值，如果其中一个操作数是字符串，则会抛出TypeError异常；而内置函数int()的功能是将一个数字或数字的字符串表示转换为一个整数，如果传入的实际参数是“abc”，那么它的类型没有问题，但它并不是某个数字的字符串表示，所以会抛出ValueError。

从ValueError派生出了下列子类：

```
ValueError
  └─ UnicodeError
       ├─ UnicodeEncodeError
       ├─ UnicodeDecodeError
       └─ UnicodeTranslateError
```

UnicodeError异常表示发生了与Unicode编解码相关的错误。从UnicodeError派生出的直接子类的含义是：

- UnicodeEncodeError：在编码过程中发生的错误。
- UnicodeDecodeError：在解码过程中发生的错误。
- UnicodeTranslateError：在翻译过程中发生的错误。

ArithmeticError异常表示发生了某种算术类错误。它派生出了下列子类：

```
ArithmeticError
  ├─ FloatingPointError
  ├─ OverflowError
  └─ ZeroDivisionError
```

FloatingPointError表示发生浮点运算错误，即导致结果为NaN的浮点运算。然而在默认情况下FloatingPointError异常并没有被启用，相关表达式会导致抛出TypeError异常或ValueError异常。OverflowError异常表示发生浮点运算溢出，即导致结果为+INF或-INF的浮点运算。而ZeroDivisionError表示发生除以零，即除法、整除或取余中第二个操作数为0。

ImportError异常表示导入模块时发生了某种错误，具有name和path参数，分别为被导入模块的模块名和指向实现该模块的文件的路径。换句话说，ImportError的实例化语法为：

```
class ImportError(*args, name=None, path=None)
```

而ModuleNotFoundError是ImportError的直接子类，即有如下继承关系：

```
ImportError
  └─ ModuleNotFoundError
```

ModuleNotFoundError异常特指找不到指定模块的情况。

SyntaxError异常代表语法错误。虽然存在语法错误的Python脚本在被执行之前就会报错，但由于可以通过import语句动态导入模块，可以通过compile()动态创建代码对象，可以通过eval()或exec()动态执行一个字符串，所以在Python脚本被执行的过程中依然可能遇到新解释的代码存在语法错误的情况，此时就会抛出SyntaxError异常。SyntaxError异常具有下列属性：

- filename：引用一个字符串，为发生语法错误的代码所在文件的文件名或路径。如果代码源自一个字符串，则为“<string>”。
- lineno：引用一个整数，为存在语法错误的代码起始字符的行号。
- offset：引用一个整数，为存在语法错误的代码起始字符的列号。
- text：引用一个字符串，为存在语法错误的代码本身。
- end_lineno：引用一个整数，为存在语法错误的代码末尾后面那个字符的行号。
- end_offset：引用一个整数，为存在语法错误的代码末尾后面那个字符的列号。

注意计算行号和列号时都从1而非0开始计数。

需要特别强调的是，SyntaxError实例化时并非通过仅关键字形式参数来设置上述属性。它接收一个可迭代对象，该对象的6个元素依次被用于设置上述属性。更具体的，SyntaxError的实例化语法为：

```
    class SyntaxError(message='', details=(None, None, None, None,
None, None))
```

下面的例子创建了一个SyntaxError异常：

```
    $ python3

    >>> e = SyntaxError('invalid syntax', ('<string>', 1, 4, 'a===0\n', 1, 5))
    >>> e.args
    ('invalid syntax', ('<string>', 1, 4, 'a===0\n', 1, 5))
    >>> e.filename
    '<string>'
    >>> e.lineno
    1
    >>> e.offset
    4
    >>> e.text
    'a===0\n'
    >>> e.end_lineno
    1
    >>> e.end_offset
    5
    >>>
```

这表明了“a===0\n”中的第3个等号是语法错误。

从SyntaxError派生出了下列子类：

```
    SyntaxError
      └─ IndentationError
           └─ TabError
```

其中IndentationError异常表示缩进相关语法错误，而TabError异常则表示因混用制表符和空格导致的缩进相关语法错误。

EOFError异常表示调用input()之后，没有输入任何文本，直接按下了Ctrl+D（以产生一个EOF）。

BufferError异常表示某种与缓冲区相关的操作（例如I/O操作和对内存视图的操作）无法执行。这可能是由于执行读取操作时缓冲区为空，也可能是由于执行写入操作时缓冲区已满，而相关操作本身不具有处理此类情况的能力（例如非阻塞式I/O操作）。

MemoryError异常表示某操作将导致物理内存耗尽，但可以通过如删除一些对象之类的方法进行挽救。这种异常常见于递归调用时。

OSError异常表示某个系统调用返回了失败的结果，典型例子包括“文件未找到”或“磁盘已满”。该异常具有下列属性：

- errno：错误码（源自C中的errno）。在Windows上为Window错误码的近似转换形式。
- strerror：错误消息（源自C中的perror()）。在Windows上为FormatMessage()的返回值。
- winerror：Windows错误码。仅在Windows上可用。
- filename：与文件系统有关的系统调用（例如open()）被传入的第一个路径参数。
- filename2：与文件系统有关的系统调用（例如rename()）被传入的第二个路径参数。

OSError被实例化时有两种语法，第一种语法是：

```
    class OSError(errno, strerror, filename=None, winerror=None, filename2=None)
```

其五个仅位置形式参数分别用于设置上述五个属性。而第二种语法是：

```
    class OSError([arg])
```

也就是只有一个可选的仅位置形式参数。上述五个属性与arg参数没有关系，都引用None。

由于系统调用非常多，所以OSError派生出了很多的子类，以将异常限定到特定的系统调用集合。下面是这些子类和OSError之间的继承关系：

```
    OSError
       ├─ BlockingIOError
       ├─ ChildProcessError
       ├─ ConnectionError
       │     ├─ BrokenPipeError
       │     ├─ ConnectionAbortedError
       │     ├─ ConnectionRefusedError
       │     └─ ConnectionResetError
       ├─ FileExistsError
       ├─ FileNotFoundError
       ├─ InterruptedError
       ├─ IsADirectoryError
       ├─ NotADirectoryError
       ├─ PermissionError
       ├─ ProcessLookupError
```

```
    └─ TimeoutError
```

关于这些子类的说明被总结在表8-3中。由于这些异常都是由操作系统自动产生的，我们很少会主动创建它们，所以关于它们的实例化语法本书从略。这里值得一提的是，从Python 3.5开始，当执行系统调用时遇到InterruptedError异常，Python解释器会自动截获它并重试该系统调用，使得该异常很少被抛出。

表8-3. OSError的直接子类

类	说明
BlockingIOError	不支持阻塞式I/O的流被阻塞时抛出。具有characters_written实例属性，引用一个整数以代表被阻塞时已经写入流的字节/字符数。对应errno中的EAGAIN、EALREADY、EWOULDBLOCK和EINPROGRESS。
ChildProcessError	当一个子进程上的操作失败时抛出。对应errno中的ECHILD。
ConnectionError	发生连接错误时抛出。具有如下直接子类： BrokenPipeError：当试图写入已关闭的套接字或另一端已关闭的管道时抛出。对应errno中的EPIPE和ESHUTDOWN。 ConnectionAbortedError：建立连接的尝试被主动终止时抛出。对应errno中的ECONNABORTED。 ConnectionRefusedError：建立连接的尝试被对方拒绝时抛出。对应errno中的ECONNREFUSED。 ConnectionResetError：连接被对方重置时抛出。对应errno中的ECONNRESET。
FileExistsError	当试图创建一个已经存在的文件或目录时抛出。对应errno中的EEXIST。
FileNotFoundError	当指定的文件或目录不存在时抛出。对应errno中的ENOENT。
InterruptedError	当系统调用被中断信号打断时抛出。对应errno中的EINTR。
IsADirectoryError	当试图对目录执行针对文件的操作时被抛出。对应errno中的EISDIR。
NotADirectoryError	当试图对文件执行针对目录的操作时被抛出。对应errno中的ENOTDIR。
PermissionError	当试图执行一个没有足够权限的操作时被抛出。对应errno中的EACCES和EPERM。
ProcessLookupError	当指定的进程不存在时被抛出。对应errno中的ESRCH。
TimeoutError	当系统调用执行超时时被抛出。对应errno中的ETIMEDOUT。

RuntimeError异常是一个兜底异常，当遇到上面没有提到的突发情况时就会抛出该异常或其派生异常。RuntimeError派生了下列子类：

```
RuntimeError
  ├─ NotImplementedError
  └─ RecursionError
```

其中NotImplementedError异常代表一个类定义没有实现其基类中的某些抽象属性，而RecursionError异常代表递归超出了函数栈允许的最大深度。

最后，SystemError异常是非常少见的，代表发生了Python解释器的一个不算太严重的内部错误。如果遇到了SystemError异常，我们应该做的是将其通报给Python解释器的开发和维护人员。

以上讨论仅涉及内置异常。我们也可以自定义异常，但这些类必须继承Exception或它的某个子类。与内置异常通常是由Python解释器抛出不同，自定义异常必须由Python脚本自身主动抛出，详见本章后面对raise语句的讨论。

8-2. 异常处理的默认流程

（教程：8.1、8.2）
（语言参考手册：3.2、3.3.1）
（标准库：types、sys）

Python解释器为处理所有异常都提供了默认流程：

➤ 处理SystemExit异常的默认流程是：如果sys.exit()的arg参数被传入整数和None之外的对象，则将str(arg)写入标准出错，然后Python解释器退出并以1作为退出状态；否则，Python解释器退出并以arg参数传入的整数（None转化为0）作为退出状态。
➤ 处理KeyboardInterrupt异常的默认流程是：中断正在执行的操作，转而与用户交互。
➤ 处理GeneratorExit异常的默认流程是：让生成器或异步生成器完成对自身的清理，以保证它们能正常关闭。
➤ 处理Exception异常及其派生异常的默认流程是：将回溯信息写入标准出错，然后终止脚本的执行。这意味着若Python解释器以交互式启动则会显示新的主提示符以等待用户下一步操作，否则会以适当参数调用sys.exit()以使自身退出。

前面已经提到，我们很少会改变处理SystemExit、KeyboardInterrupt和GeneratorExit异常的默认流程。因此本节着重讨论处理Exception异常及其派生异常的默认流程。下面先通过一个列子来体现该流程：

```
$ python3

>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

在该例子中Python解释器以交互式启动，因对表达式1/0求值抛出了ZeroDivisionError异常。由于ZeroDivisionError是Exception的子类，所以处理它的默认流程是将回溯信息写入标准出错，然后显示新的主提示符。事实上，在本书前面的不少例子中也出现了类似的回溯信息，它们都是处理某种异常的默认流程的结果。

那么我们该如何解读上述格式的回溯信息呢？回溯信息中的第一行总是“Traceback (most recent call last):”，这表明回溯信息源自“回溯（traceback）”这一机制。要想完

全清晰地阐明回溯机制，必须深入了解计算机系统的底层，所以本书从略。概括地说，导致一个异常被抛出的指令可以有多条，它们可能属于函数栈中的不同帧，并基于指令被执行的先后顺序形成一条路径，而回溯则指的是逆着这条路径依次定位到所有相关指令。这一技术对于调试非常重要。

接下来的若干行回溯信息给出了所有导致抛出该异常的指令，并将其转换为对应源代码。此类行通常是三行一组：上面那行提供相关文件的路径、代码在该文件中的行号、代码属于的函数、类或模块（尽可能精确）；中间那行是代码本身；下面那行则标记出问题代码的具体位置。然而在上面的例子中此类行只有一行，即“File "<stdin>", line 1, in <module>”，其中“<stdin>”代表标准输入，这意味着导致抛出该异常的代码是用户之前通过标准输入键入的，因此没有必要再额外重复一次，当然也就无法标记问题代码的具体位置。

回溯信息的最后一行总是“异常类型: 错误消息”的格式：冒号左侧的部分是异常所属类的类名，在该例子中为“ZeroDivisionError”；冒号右侧的部分是错误消息，在该例子中为“division by zero”。从这一行可以知道抛出了什么异常，以及抛出该异常的大概原因。

下面再给出一个异常处理默认流程的例子。请将下面的代码保存为exception1.py：

```
def f(a):
    return 'str:' + a

f(1)
```

在这个例子中我们定义了一个拼接字符串的函数f()，而在调用它时却传入了一个整数，因此会导致抛出TypeError异常。

下面先以交互式启动Python解释器来抛出这一异常：

```
$ python3

>>> import exception1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/exception1.py", line 4, in <module>
    f(1)
    ^^^^
  File "/Users/wwy/exception1.py", line 2, in f
    return 'str:' + a
           ~~~~~~~^~~
TypeError: can only concatenate str (not "int") to str
>>> ^D
```

然后以非交互式启动Python解释器来抛出这一异常：

```
$ python3 exception1.py
Traceback (most recent call last):
  File "/Users/wwy/exception1.py", line 4, in <module>
```

```
    f(1)
    ^^^^
  File "/Users/wwy/exception1.py", line 2, in f
    return 'str:' + a
           ~~~~~~~^~~
TypeError: can only concatenate str (not "int") to str
$
```

在这个例子中，导致抛出异常的代码有两处，逆着回溯路径，第一处是文件exception1.py中的函数调用，具体位置是“f(1)”；第二处是文件exception1.py中的return语句，具体位置是“+”。

需要说明的是，标记问题代码的具体位置是Python 3.11添加的功能，使用Python 3.10或更早版本时回溯信息中不会有相关行。对于上面的例子来说，在Python 3.10中输出的回溯信息是这样的：

```
Traceback (most recent call last):
  File "/Users/wwy/exception1.py", line 4, in <module>
    f(1)
  File "/Users/wwy/exception1.py", line 2, in f
    return 'str:' + a
TypeError: can only concatenate str (not "int") to str
```

显然，精确定位问题代码可以极大的提高调试的效率。然而获得该信息是有代价的，需要用到代码对象的co_positions属性引用的函数。第4章已经说明，这会略微增大.pyc文件的大小，且略微增加使用的内存。当启动Python解释器时添加了-X no_debug_ranges选项，或者设置了PYTHONNODEBUGRANGES环境变量，那么代码对象的co_positions属性将引用None，这样回溯信息中就不会包含标记问题代码具体位置的行，与Python 3.10的输出完全相同。

最后需要强调，上述回溯信息格式对除了SyntaxError之外的异常都适用。SyntaxError异常是一个较特殊的异常，代表着语法错误。导致抛出SyntaxError异常的代码不会被执行，因此也就不存在回溯信息。下面的例子说明了处理SyntaxError异常的默认流程会显示的信息：

```
>>> 0 = 1
  File "<stdin>", line 1
    0 = 1
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of
'='?
>>>
```

注意不存在“Traceback (most recent call last):”这一行，表明这些信息不是回溯信息。

以上就是处理Exception异常及其派生异常的默认流程。特别的，如果以交互式启动Python解释器，那么在看到回溯信息之后，还可以通过表8-4列出的sys属性来获取关于该异常的更详细信息。这些信息对于调试是很有用的。如果访问这些sys属性时并没有抛出异常，或抛出了异常但被Python脚本自身截获并处理，则这些sys属性都会引用None。

表8-4. 未截获异常信息相关sys属性

属性	说明
sys.last_type	引用一个类对象，为最新抛出异常的类型。
sys.last_value	引用最新抛出异常本身。
sys.last_traceback	引用一个回溯对象，以支持回溯。

在第3章已经提到过，“回溯对象（traceback objects）”是由Python解释器在内部使用的，几乎总是在抛出异常时被自动创建。每个回溯对象对应一条导致该异常被抛出的指令，因此一个异常可能关联多个回溯对象，它们之间形成一个单链表以代表回溯路径。回溯对象具有下列实例属性：

- tb_next：引用前一个回溯对象，以实现回溯路径。如果当前回溯对象是回溯路径的终点，则引用None。
- tb_frame：引用一个帧对象，为相关指令被执行时所属的函数栈帧。
- tb_lineno：引用一个整数，为相关指令在在源代码中对应行的行号。
- tb_lasti：引用一个整数，为相关指令在帧对象引用的代码对象中的索引。

易知，回溯对象中储存的信息形成了某个帧对象在异常被抛出时刻的快照，哪怕之后该帧对象因代码的执行而改变了状态，回溯对象中固化的信息也不受影响。

下面用exception1.py来验证上述sys属性的作用：

```
$ python3 -i exception1.py
Traceback (most recent call last):
  File "/Users/wwy/exception1.py", line 4, in <module>
    f(1)
    ^^^^
  File "/Users/wwy/exception1.py", line 2, in f
    return 'str:' + a
           ~~~~~~~^~~
TypeError: can only concatenate str (not "int") to str

>>> import sys
>>> sys.last_type
<class 'TypeError'>
>>> sys.last_value
```

```
      TypeError('can only concatenate str (not "int") to str')
      >>> sys.last_traceback
      <traceback object at 0x10dfa73c0>
      >>> tb1 = sys.last_traceback
      >>> tb1.tb_next
      <traceback object at 0x10dfa7200>
      >>> tb2 = tb1.tb_next
      >>> tb2.tb_next
      >>> tb1.tb_frame
      <frame at 0x10dde9ea0, file '/Users/wwy/exception1.py', line 4, code
<module>>
      >>> tb1.tb_lineno
      4
      >>> tb1.tb_lasti
      26
      >>> tb2.tb_frame
      <frame at 0x10de2ec00, file '/Users/wwy/exception1.py', line 2, code f>
      >>> tb2.tb_lineno
      2
      >>> tb2.tb_lasti
      6
      >>>
```

可以看出，当因直接执行exception1.py而抛出TypeError异常时，问题代码有两处，因此关联到该异常的回溯路径中有两个回溯对象。

我们也可以通过标准库的types模块中定义的如下类来显式创建回溯对象：

***class* types.TracebackType(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)**

手工创建回溯对象的目的主要是作为调用异常的函数属性with_traceback()的参数，会在后面讨论。

在有些情况下，一个异常可能无法被传递。例如在垃圾回收期间抛出的异常就无法进入正常的异常处理流程。处理无法传递异常的默认流程与处理可传递异常的默认流程基本相同，只存在如下区别：写入标准出错的信息的第一行是“Exception ignored in: ...”，以表明该异常无法被传递。请通过如下命令行和语句来验证这种情况：

```
      $ python3

      >>> class C:
      ...     def __del__(self):
      ...         raise Exception("I am destroyed!")
      ...
      >>> obj = C()
      >>> del obj
      Exception ignored in: <function C.__del__ at 0x103ab30a0>
      Traceback (most recent call last):
        File "<stdin>", line 3, in __del__
      Exception: I am destroyed!
      >>>
```

在该例子中我们通过raise语句主动抛出了一个自定义异常，这一技巧会在后面讨论。

我们可以通过表8-5列出的sys属性访问Python解释器实现处理Exception异常及其派生异常的默认流程的函数。

sys.excepthook用于处理可传递异常，负责将回溯信息按照前述格式写入标准出错，其语法为：

```
sys.excepthook(type, value, traceback)
```

而sys.unraisablehook则用于处理无法传递异常，负责将上述格式的字符串写入标准出错，其语法为：

```
sys.unraisablehook(unraisable)
```

通过让它们引用其他函数，可以改变Python解释器的异常处理默认流程，但一般不建议这样做（除非你是Python专家）。即便改写了sys.excepthook和sys.unraisablehook，也能分别通过sys.__excepthook__和sys.__unraisablehook__取得它们默认引用的函数。

表8-5. 异常处理默认流程相关sys属性

属性	说明
sys.excepthook	将传入的回溯信息以指定格式写入标准出错。其中type、value和traceback参数分别传入异常类型、异常和回溯对象。
sys.unraisablehook	处理一个无法传递异常。通过unraisable参数传入的对象需要具有如下属性： exc_type：异常类型。 exc_value：异常，可以引用None。 exc_traceback：回溯对象，可以引用None。 err_msg：错误消息，可以引用None。 object：导致该异常的对象，可以引用None。
sys.__excepthook__	sys.excepthook的默认值。
sys.__unraisablehook__	sys.unraisablehook的默认值。

8-3. 异常处理的自定义流程

（教程：8.3、8.7）
（语言参考手册：4.3、7.6、7.9、7.10、8.4）
（标准库：sys）

我们可以在Python脚本中截获一个被抛出的异常，自定义对它的处理方式，而这就需要用到try语句。try语句的完整语法为：

```
try:
    suite0
except expression1 [as identifier1]:
    suite1
except expression2 [as identifier2]:
    suite2
...
else:
     suiteE
finally:
     suiteF
```

该语句的执行逻辑是这样的：无条件执行suite0，然后按如下规则分类处理：

➤ 如果在执行suite0的过程中没有抛出异常，也没有执行return语句、continue语句或break语句，则执行suiteE，然后再执行suiteF。
➤ 如果在执行suite0的过程中没有抛出异常，但执行了return语句、continue语句或break语句，则跳过suiteE，直接执行suiteF。
➤ 如果在执行suite0的过程中抛出了异常，那么设抛出的异常是e，依次判断e是否与expression1、expression2、……匹配。这样又分两种情况：
a. expressionN是第一个与e匹配的表达式，执行suiteN（此时若存在as子句则在suiteN中可通过标识符identifierN来引用e），然后执行suiteF。
b. e不与通过except子句给出的任何表达式匹配，直接执行suiteF。

而判断一个异常与某个表达式是否匹配的规则是这样的：对该表达式求值，如果得到的对象满足下面的条件之一则认为匹配：

➤ 该对象是一个类对象，是异常所属类，或者是异常所属类的某个非抽象基类。
➤ 该对象是一个类对象形成的元组，其中某个类对象满足条件1。

注意异常中包含的错误消息是可以任意设置的，即便内置异常的错误消息也会因Python解释器版本的变化而变化，因此不能用于匹配异常。

可以看出，try语句的执行逻辑非常复杂。此外，except子句、else子句和finally子句并不总是需要同时在try语句中出现，而是允许如下两种情况：

➤ 至少有一条except子句，而else子句和finally子句都是可选的。
➤ except子句和else子句都不存在，只存在finally子句。

因此，try语句有非常多的变体，下面通过大量例子来说明该语句的每一种变体。

首先考虑try语句仅包含若干条except子句的变体。使用这种变体背后的逻辑是：我们知道执行一段代码可能会抛出特定的某种或某几种异常。为了避免Python程序因此而终止，我们选择截获并自行处理这些异常，而其他异常则依然使用默认流程处理。

下面的exception2.py是对exception1.py的改写，使得当传递给f()的参数不是字符串时返回一条错误消息：

```
def f(a):
    try:
        return 'str:' + a
    except TypeError:
        return 'str:Please enter a string!'
```

请通过如下命令行和语句验证：

```
$ python3 -i exception2.py

>>> f('b')
'str:b'
>>> f(1)
'str:Please enter a string!'
>>> f(None)
'str:Please enter a string!'
>>> f('b', 'c')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
>>>
```

注意在上面例子中给f()传入两个参数时，抛出的TypeError异常并没有被try语句截获。这是因为该异常是在函数将实际参数赋值给形式参数时抛出的，还没有执行到try语句。

下面的exception3.py是预计到一段代码可能抛出多种异常，且需要对这些异常进行分别处理的情况：

```
import math


def f(x, y):
    try:
        return math.sqrt(x)/y
    except TypeError:
        print('Please enter two numbers!')
```

```
            return math.nan
        except ValueError:
            print('x must be non-negative!')
            return math.nan
        except ZeroDivisionError:
            print('y cannot be 0!')
            return math.nan
```

请通过如下命令行和语句验证：

```
$ python3 -i exception3.py

>>> f(1, 2)
0.5
>>> f('a', 'b')
Please enter two numbers!
nan
>>> f(-1, 2)
x must be non-negative!
nan
>>> f(1, 0)
y cannot be 0!
nan
>>>
```

如果需要对多种异常进行同样的处理，那么与其提供多条具有相同处理代码的except子句，不如提供一条except子句，将所有异常所属类都放在一个元组内，以使它可以截获其中任意一种异常。下面的exception4.py是对exception3.py的改写，使得f()在try语句截获TypeError、ValueError和ZeroDivisionError异常的情况下都返回NaN：

```
import math


def f(x, y):
    try:
        return math.sqrt(x)/y
    except (TypeError, ValueError, ZeroDivisionError):
        return math.nan
```

请通过如下命令行和语句验证：

```
$ python3 -i exception4.py

>>> f(1, 2)
0.5
>>> f('a', 'b')
nan
>>> f(-1, 2)
nan
>>> f(1, 0)
nan
>>>
```

在通过except子句截获一个异常时，可以通过as子句将其绑定到一个标识符。下面的exception5.py是对exception4.py的改写，使得截获的异常会被记录到err.log文件中：

```
import math
import time


def f(x, y):
    try:
        return math.sqrt(x)/y
    except (TypeError, ValueError, ZeroDivisionError) as e:
        t = time.asctime()
        fd = open('err.log', 'a')
        fd.writelines((
                'time: ' + t + '\n',
                'type: ' + repr(type(e)) + '\n',
                'args: ' + repr(e.args) + '\n',
                '\n'))
        fd.close()
        return math.nan
```

请通过如下命令行和语句验证：

```
$ python3 -i exception5.py

>>> f(1, 2)
0.5
>>> f('a', 'b')
nan
>>> f(-1, 2)
nan
>>> f(1, 0)
nan
>>>
```

注意工作目录下会出现err.log文件，其内容为（时间可能不同）：

```
time: Fri Apr  1 11:36:18 2022
type: <class 'TypeError'>
args: ('must be real number, not str',)

time: Fri Apr  1 11:36:21 2022
type: <class 'ValueError'>
args: ('math domain error',)

time: Fri Apr  1 11:36:24 2022
type: <class 'ZeroDivisionError'>
args: ('float division by zero',)
```

第3章已经提到，try语句的except子句是标识符绑定操作的一种。需要强调的是，通过这种方式绑定的标识符只能在该except子句对应的代码块中使用，离开该代码块后该标识符就会被自动销毁。可以这样认为：当具有as子句的except子句对应的代码块执行结束时，会自

动额外执行一条语句“del identifier”。如果as子句中的标识符在之前已经被定义，那么不论它是局部变量还是全局变量都会被销毁。请看下面的例子exception6.py：

```
import math


c = 'c'

def f(x, y):
    global c
    a = 'a'
    try:
        b = 'b'
        return math.sqrt(x)/y
    except TypeError as a:
        print(a)
        return math.nan
    except ValueError as b:
        print(b)
        return math.nan
    except ZeroDivisionError as c:
        print(c)
        return math.nan
    finally:
        print(a)
        print(b)
        print©
```

请通过如下命令行和语句验证：

```
$ python3 -i exception6.py

>>> f(1, 2)
a
b
c
0.5
>>>
>>> f('a', 'b')
must be real number, not str
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/exception6.py", line 22, in f
    print(a)
          ^
UnboundLocalError: cannot access local variable 'a' where it is not
associated with a value
>>>
>>> f(-1, 2)
math domain error
a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/exception6.py", line 23, in f
    print(b)
          ^
UnboundLocalError: cannot access local variable 'b' where it is not
associated with a value
>>>
```

```
>>> f(1, 0)
float division by zero
a
b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/exception6.py", line 24, in f
    print(c)
          ^
NameError: name 'c' is not defined
>>>
```

注意该例子中使用了finally子句，使得即便截获并处理了异常后也会执行该子句指定的代码块，进而判断标识符a、b和c是否还存在：如果不存在，则会抛出UnboundLocalError异常或NameError异常，而这些异常是通过默认流程处理的。后面还会给出更多关于finally子句的例子。

除了通过as子句将抛出的异常绑定到一个标识符之外，在except子句中还可以通过表8-6列出的sys属性来获得关于该异常的信息。

表8-6. 截获异常信息相关sys属性

属性	说明
sys.exc_info()	返回当前正在被处理的异常的信息。
sys.exception()	返回当前正在被处理的异常。

sys.exc_info()会返回一个三元组：第一个元素为异常的类型（类似sys.last_type）；第二个元素为异常本身（类似sys.last_value）；第三个元素为关联到该异常的回溯对象（类似sys.last_traceback）。事实上，sys.exc_info()并不局限于在except子句中被使用，它会在当前线程中从函数栈的当前帧对象开始搜索尚未完成处理的异常，如果找不到则退回到上一个帧对象，依此类推。如果当前线程没有任何尚未完成处理的异常，则sys.exc_info()会返回（None, None, None）。

易知，从sys.exc_info()获得的信息是通过as子句获得的信息的超集，其返回的三元组中前两个元素包含的信息与as子句包含的信息相同，第三个元素则使我们可以手工进行回溯。下面的例子exception7.py说明了这一技巧（该例子在调用sys.exc_info()时使用了解包赋值，这会在第12章被详细讨论）：

```
import math
import time
import sys


def f(x, y):
    try:
        return math.sqrt(x)/y
    except (TypeError, ValueError, ZeroDivisionError):
        t = time.asctime()
        ty, v, tb = sys.exc_info()
```

```
            cursor = tb
            tb_info = ''
            while cursor is not None:
                tb_info = tb_info + repr(cursor.tb_frame) + '\n'
                cursor = cursor.tb_next
            fd = open('err.log', 'a')
            fd.writelines((
                    'time: ' + t + '\n',
                    'type: ' + repr(ty) + '\n',
                    'args: ' + repr(v.args) + '\n',
                    'traceback:\n',
                    tb_info,
                    '\n'))
            fd.close()
            return math.nan
```

可以看出，exception7.py是对exceptio5.py的改写，以将回溯信息也记录在err.log文件中。请通过如下命令行和语句验证：

```
$ python3 -i exception7.py

>>> f(1, 2)
0.5
>>> f('a', 'b')
nan
>>> f(-1, 2)
nan
>>> f(1, 0)
nan
>>>
```

注意err.log文件会被追加如下内容（时间可能不同）：

```
time: Fri Apr  1 14:51:46 2022
type: <class 'TypeError'>
args: ('must be real number, not str',)
traceback:
<frame at 0x7fcfcc70d720, file '/Users/wwy/exception7.py', line 15, code f>

time: Fri Apr  1 14:51:48 2022
type: <class 'ValueError'>
args: ('math domain error',)
traceback:
<frame at 0x7fcfcc596e00, file '/Users/wwy/exception7.py', line 15, code f>

time: Fri Apr  1 14:51:51 2022
type: <class 'ZeroDivisionError'>
args: ('float division by zero',)
traceback:
<frame at 0x7fcfcc72c750, file '/Users/wwy/exception7.py', line 15, code f>
```

而sys.exception()是Python 3.11引入的，其功能就等同于as子句。与sys.exc_info()类似，sys.exception()并不局限于在except子句中被使用。

异常之间存在继承关系，而根据except子句的匹配规则，可以用一个异常匹配它的所有派生异常，例如可以用ArithmeticError异常截获FloatingPointError、OverflowError和ZeroDivisionError异常。然而，如果我们想对某种派生异常进行较特殊的处理，那么就一定要单独截获该派生异常，并将相应except子句放在截获派生它的异常的except子句前面。下面的exception8.py说明了这点：

```
import math


def f(x, y):
    try:
        return math.exp(x)/y
    except ZeroDivisionError:
        print("Divisor cannot be 0.")
        return math.nan
    except ArithmeticError:
        print("Unknown arithmetic error.")
        return math.nan
```

请通过如下命令行和语句验证：

```
$ python3 -i exception8.py

>>> f(1, 0)
Divisor cannot be 0.
nan
>>> f(100000000000000, 1)
Unknown arithmetic error.
nan
>>>
```

从上面的结果可以看出，f()可以区分ZeroDivisionError和其他ArithmeticError派生异常。但如果交换两个except子句的位置，即如exception9.py：

```
import math


def f(x, y):
    try:
        return math.exp(x)/y
    except ArithmeticError:
        print("Unknown arithmetic error.")
        return math.nan
    except ZeroDivisionError:
        print("Divisor cannot be 0.")
        return math.nan
```

那么f()将无法对ZeroDivisionError异常特殊处理，因为“except ArithmeticError”就会将其截获。请通过如下命令行和语句验证：

```
$ python3 -i exception9.py

>>> f(1, 0)
Unknown arithmetic error.
nan
>>> f(100000000000000, 1)
Unknown arithmetic error.
nan
>>>
```

从上面的例子很容易想到，可以将“except Exception”作为最后一条except子句，使得suite0不论抛出什么异常都可以被截获。这一技巧在实践中是经常被使用的，但要特别强调，不能用“except BaseException”代替“except Exception”，因为前者还会截获SystemExit异常、KeyboardInterrupt异常和GeneratorExit异常，使Python解释器失去一些最基本的功能（例如退出和中断）。此外，如果省略了except子句中的表达式，则默认会以BaseException作为表达式，而这是PEP 8所不推荐的。

以上就是关于try语句的第一种变体所需掌握的知识。

下面考虑try语句的第二种变体，即包括若干条except子句，以及else子句。使用这种变体背后的逻辑是：我们知道执行一段代码可能会抛出异常，其中某些特定的异常会被截获并自行处理，而如果没有抛出任何异常时，则额外执行一段代码。

使用这种变体时，必须注意suite0中的return语句、break语句和continue语句是否被执行。前面的很多例子中都有return语句，而break语句和continue语句是当try语句被嵌入循环体使用时才有意义的，分别用于跳出循环和执行下一次循环。这三种语句都会导致else子句被跳过。

请看下面的例子exception10.py：

```
print("这是一个判断整数符号的小游戏。\n")

while True:
    try:
        s = input("请输入一个整数: ")
        if s == "quit":
            print("你已经退出了游戏。\n")
            break
        n = int(s)
    except ValueError:
        print("您输入的并非整数。")
    except Exception:
        print("发生了未知错误。")
    else:
        if n < 0:
            print("这是一个负数。")
        elif n > 0:
            print("这是一个正数。")
```

```
            else:
                print("这是零。")
```

该例子是对第3章中的number_sign_game3.py的改写，放弃了通过正则表达式来判断输入的字符串是否代表一个整数，而是直接使用int()进行强制类型转换，使得当该字符串不代表一个整数时抛出ValueError异常。注意对整数符号的判断放在了else子句中，而当输入的字符串是“quit”时则会执行break语句跳出循环，此时else子句会被跳过。该例子还通过“except Exception”子句作为兜底的异常处理。请通过如下命令行来验证上面的例子：

```
$ python3 exception10.py
这是一个判断整数符号的小游戏。

请输入一个整数: 10
这是一个正数。
请输入一个整数: 0
这是零。
请输入一个整数: -1
这是一个负数。
请输入一个整数: adf
您输入的并非整数。
请输入一个整数: ^D发生了未知错误。
请输入一个整数: quit
你已经退出了游戏。
```

上面导致输出“发生了未知错误。”的操作是直接输入Ctrl+D，使input()抛出了EOFError异常，进而被“except Exception”子句截获。

用“except Exception”子句兜底似乎很完美，但其实依然存在漏洞。except子句和else子句中的代码块也可能抛出异常，甚至如果except子句中的表达式很复杂则也有抛出异常的可能。这些异常被视为try语句本身抛出的异常，因此是不会被该try语句的except子句截获的。能够起到真正兜底作用的是finally子句。

下面讨论try语句的第三种变体：存在若干except子句，else子句可有可无，而最后有finally子句。

当finally子句存在时，suiteF在任何情况下都会被执行。如果存在未被截获的异常，或者except子句/else子句在执行过程中抛出了新的异常，那么当suiteF开始被执行时，该异常会被自动截获并暂时保存起来。

suiteF自身也可能抛出异常，如果这种情况发生，则该异常会替换掉被保存的异常。如果suiteF执行到了return语句、break语句或continue语句，则被保存的异常会被丢弃；否则，当suiteF执行完之后，被保存的异常会被重新抛出。finally子句不支持对保存的异常进行处理，也不支持通过as子句将保存的异常与某个标识符绑定，但在suiteF中依然可以通过sys.exc_info()和sys.exception()取得被保存异常的相关信息。

需要强调的是，即便在执行except子句/else子句时执行到了return语句，suiteF依然会被执行。这种情况下，如果执行suiteF时没有执行另一条return语句，则返回值将由except子句/else子句决定；否则，suiteF中的return语句将覆盖掉except子句/else子句中的return语句。而如果suiteF执行过程中抛出了新异常，那么except子句/else子句中的return语句将被跳过（exception6.py是这种情况的一个例子）。

下面的例子exception11.py说明了try语句的第三种变体的作用：

```
import math
import sys


def f(x, y):
    try:
        result = math.nan
        result = math.sqrt(x)/y
    except TypeError:
        print('Please enter two numbers!')
    except ValueError:
        print('x must be non-negative!')
    finally:
        print(sys.exc_info())
        return result
```

该例子中的try语句通过except子句截获并处理了TypeError异常和ValueError异常，其他异常都会传递给finally子句，而由于finally子句总会执行到return语句，所以这些异常最终都会被丢弃，不会传递到try语句之外。请通过下面的命令行和语句来验证：

```
$ python3 -i exception11.py

>>> f(1, 2)
(None, None, None)
0.5
>>> f('a', 'b')
Please enter two numbers!
(None, None, None)
nan
>>> f(-1, 2)
x must be non-negative!
(None, None, None)
nan
>>> f(1, 0)
(<class 'ZeroDivisionError'>, ZeroDivisionError('float division by zero'),
<traceback object at 0x11070cc80>)
nan
>>>
```

从上面的例子很容易想到，如果不想截获并处理任何异常，但想避免抛出任何异常，那就只需要finally子句，不需要except子句和else子句。而这就是try语句的第四种变体：只包含finally子句作为异常清理器。下面的例子exception12.py是对exception11.py的改写：

```
import math


def f(x, y):
    try:
        result = math.nan
        result = math.sqrt(x)/y
    finally:
        return result
```

这使得f(x, y)在传入的参数完成正确时返回一个数值，否则返回NaN，函数体本身不会抛出任何异常。

请通过如下命令行和语句验证：

```
$ python3 -i exception12.py

>>> f(1, 2)
0.5
>>> f('a', 'b')
nan
>>> f(-1, 2)
nan
>>> f(1, 0)
nan
>>>
```

在实际应用中，finally子句主要被用于执行一些任何情况下都需要执行的操作，例如关闭已经打开的文件或外围设备。PEP 8建议让suiteF尽可能短，以最大程度避免执行suiteF时抛出新的异常。

8-4. 异常链

（标准库： 内置异常）

上一节已经说明，在执行try语句中的except子句的表达式和代码块，else子句的代码块，以及finally子句的代码块时，有可能抛出新的异常。如果不存在finally子句，或者finally子句的代码块没有执行到return语句、break语句和continue语句，则该新异常将被传递到try语句之外，并被视为try语句本身抛出的异常。这是使用try语句最麻烦的地方。

为了处理try语句本身抛出的异常，必要用try语句嵌套try语句。此外，还必须建立起“异常链（exception chains）”这个概念。

事实上，任何异常都具有下面三个与异常链相关的实例属性：

➤ __context__：引用异常链中的前一个异常，表明该异常是在处理该属性引用的异常的过程中被抛出的。
➤ __cause__：引用另一个异常，表明该异常是由该属性引用的异常导致的。

➤ __suppress_context__：引用一个布尔值，用于控制处理该异常的默认流程是否先显示该异常的__context__属性引用的异常。

请执行如下命令行和语句以验证上述三个属性的存在：

```
$ python3

>>> try:
...     1/0
... except Exception as e:
...     print(e.__context__)
...     print(e.__cause__)
...     print(e.__suppress_context__)
...
None
None
False
```

异常链是通过__context__属性构建的：多个异常相互间通过__context__属性连接成一个单链表，而异常链顶点处的异常的__context__属性总是引用None。本节之前给出的所有例子中的异常都位于异常链的顶点。仅在下列情况下，一个异常的__context__属性不引用None：

➤ 执行except子句的表达式或代码块时抛出的新异常，其__context__属性将引用导致该表达式或代码块被执行的原异常。
➤ 执行finally子句的代码块时抛出的新异常，且finally子句保存了原异常。新异常的__context__属性将引用它所替换掉的原异常。

而在下列情况下，即便一个异常是由try语句本身抛出的，其__context__属性依然会引用None：

➤ 执行else子句的代码块时抛出的新异常。
➤ 执行finally子句的代码块时抛出的新异常，且finally子句没有保存原异常。

下面是一个由except子句形成异常链的例子：

```
$ python3

>>> try:
...     try:
...         1/0
...     except Exception:
...         a = a + 1
... except Exception as e:
```

```
...     print(e.__context__)
...     print(e.__cause__)
...     print(e.__suppress_context__)
...
division by zero
None
False
>>>
```

下面则是一个由finally子句形成异常链的例子：

```
$ python3

>>> try:
...     try:
...         1/0
...     finally:
...         a = a + 1
... except Exception as e:
...     print(e.__context__)
...     print(e.__cause__)
...     print(e.__suppress_context__)
...
division by zero
None
False
>>>
```

__suppress_context__属性则控制着异常链的显示。当一个异常被默认流程处理时，如果其__suppress_context__属性引用False，则会先显示异常链中该异常前面那个异常；而如果其__suppress_context__属性引用True，则仅会显示该异常本身。这意味着默认流程会逆着异常链从最后一个异常开始显示到第一个__suppress_context__属性引用True的异常，而如果异常链中的所有异常的该属性都引用False，则整个异常链都会被显示。

下面的例子显示了整个异常链：

```
$ python3

>>> try:
...     try:
...         1/0
...     finally:
...         a = a + 1
... except Exception:
...     print(e)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
NameError: name 'a' is not defined
```

```
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
NameError: name 'e' is not defined
>>>
```

注意该异常链由三个异常组成，因为内层try语句的finally子句和外层try语句的except子句都抛出了新的异常。异常链不能无限增长，CPython对其的长度限制是1000。

__cause__属性对异常链的结构不会构成影响，但对异常链的显示会造成影响。该属性默认引用None，必须通过具有from子句的raise语句（会在下一节讨论）来设置，其引用的异常可以在异常链上，也可以不在异常链上。当__cause__属性引用的不是None时，__suppress_context__属性必然引用True，在这种情况下异常处理的默认流程会先显示__cause__属性引用的异常，再显示该异常本身。

在有了异常链的概念之后，我们就可以明白sys.exc_info()和sys.exception()其实获取的是异常链中最后一个异常的相关信息。下面的例子说明了这点：

```
>>> import sys
>>> try:
...     try:
...         1/0
...     finally:
...         a = a + 1
... except Exception:
...     print(sys.exc_info())
...     print(sys.exception())
...
(<class 'NameError'>, NameError("name 'a' is not defined"), <traceback
object at 0x1061f32c0>)
name 'a' is not defined
>>>
```

两者的区别在于sys.exc_info()返回该异常被解析后的信息，而sys.exception()返回该异常本身。

8-5. 主动触发异常

（教程：8.4～8.6、8.10）
（语言参考手册：7.8）
（标准库：sys）

到目前为止，我们仅考虑了Python解释器自动抛出的异常。主动触发异常是一种进一步利用异常处理流程的技巧。概括地说，我们可以通过raise语句主动抛出异常，这些异常既可以是内置异常，也可以是自定义异常。raise语句的语法为：

```
raise [expression1 [from expression2]]
```

它有三种变体，下面分别通过例子说明。

---

raise语句的第一种变体是不带任何参数，仅有“raise”关键字自身。这种raise语句的行为是：如果当前有尚未完成处理的异常，则将其抛出；否则抛出RuntimeError异常。

下面的例子直接执行raise语句的第一种变体，抛出RuntimeError异常：

```
$ python3

>>> raise
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: No active exception to reraise
>>>
```

下面的例子raise1.py在内层try语句的except子句中使用raise语句的第一种变体。内层try语句的作用是计算表达式math.sqrt(x)/y，并通过全局变量统计三种不同异常被抛出的累计次数。外层try语句的作用是返回计算结果，并在有异常被抛出时显示统计信息：

```
import math

tn = 0
vn = 0
zn = 0

def f(x, y):
    global tn, vn, zn
    try:
        try:
            result = math.nan
            result = math.sqrt(x)/y
        except TypeError:
            tn = tn + 1
            raise
        except ValueError:
            vn = vn + 1
            raise
        except ZeroDivisionError:
            zn = zn + 1
            raise
    except Exception:
        print('tn:' + str(tn) + ', vn:' + str(vn) + ', zn:' + str(zn))
    finally:
        return result
```

请通过如下命令行和语句验证：

```
$ python3 -i raise1.py

>>> f(1, 2)
0.5
>>> f('a', 'b')
tn:1, vn:0, zn:0
nan
>>> f('a', 'b')
tn:2, vn:0, zn:0
nan
>>> f(-1, 0)
tn:2, vn:1, zn:0
nan
>>> f(1, 0)
tn:2, vn:1, zn:1
nan
>>>
```

---

raise语句的第二种变体是只有expression1，没有from子句。对expression1求值得到的对象必须是BaseException或它的某个子类，或者它们的实例（前一种情况会自动创建该类的实例）。

理论上，raise语句可以用来抛出内置异常，但由于内置异常具有特定含义，绝大多数情况下都应该由Python解释器抛出，所以raise语句的这种变体多用于抛出自定义异常。前面已经说明，自定义异常必须派生自Exception异常。

下面通过例子raise2.py同时说明raise语句的第二种变体和自定义异常的用法：

```
#自定义异常类，具有下列实例属性：
#  low：合法范围的下限。
#  high：合法范围的上限。
#  current：合法范围之外的当前值。
class RangeError(Exception):
    def __init__(self, *args, low=0, high=100, current=-1):
        self.low = low
        self.high = high
        self.current = current
        super().__init__(*args)

    #重写魔术属性__str__。
    def __str__(self):
        extra = ('The legal range is ['
                + str(self.low) + ',' + str(self.high)
                + '], but the current value is '
                + str(self.current) + '. ')
        return extra + super().__str__()


#定义代表温度计的类。
class Thermometer():
    def __init__(self, temperature=0):
        if temperature < -20 or temperature > 50:
```

```
                e = RangeError('Out of range!', low=-20, high=50,
current=temperature)
                e.add_note("Raise during initialization!")
                raise e
            self.temperature = temperature

        #改变温度计的温度。
        def change(self, delta):
            temperature = self.temperature + delta
            if temperature < -20 or temperature > 50:
                e = RangeError('Out of range!', low=-20, high=50,
current=temperature)
                e.add_note("Raise during changing!")
                raise e
            self.temperature = temperature

        #以“-”、“0”、“+”和“@”绘制温度计读数。  四个“-”代表零下温度区域，十个“+”
        # 代表零上温度区域，“0”代表零度。  “@”代表当前温度，需要替换掉一个“-”、
        #  “0”或“+”。
        def show(self):
            scale = '----0++++++++++'
            index = self.temperature//5 + 4
            marked_scale = scale[0:index] + '@' + scale[index+1:14]
            print(marked_scale)
```

这个例子略微复杂：它先定义了异常类RangeError，以表示指定的温度超出了合法范围。相应异常具有low、high和current三个实例属性，而被写入标准出错时会显示合法范围和超出范围的当前值。然后它定义了代表温度计的类Thermometer。在该类被实例化，以及调用类属性change()时，会检查设置的温度是否超出范围，如果是则通过raise语句的第二种变体抛出RangeError。类属性show()的作用是以准图形化的方式显示当前温度，而当前温度的具体值则应通过实例属性temperature查看。

请通过如下命令行和语句验证：

```
$ python3 -i raise2.py

>>> t1 = Thermometer()
>>> t1.show()
----@+++++++++
>>> t1.temperature
0
>>> t1.change(-10)
>>> t1.show()
--@-0+++++++++
>>> t1.temperature
-10
>>> t1.change(-20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise2.py", line 36, in change
    raise e
    ^^^^^^^
RangeError: The legal range is [-20,50], but the current value is -30. Out
of range!
Raise during changing!
>>> t1.change(40)
>>> t1.show()
----0+++++@+++
```

```
>>> t1.temperature
30
>>> t2 = Thermometer(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise2.py", line 27, in __init__
    raise e
    ^^^^^^^
RangeError: The legal range is [-20,50], but the current value is 100. Out
of range!
Raise during initialization!
>>>
```

上面这个例子中有两处通过raise语句的第二种变体抛出了自定义异常，第一处是在类Thermometer的__init__魔术属性中，相应代码为：

```
    e = RangeError('Out of range!', low=-20, high=50, current=temperature)
    e.add_note("Raise during initialization!")
    raise e
```

而抛出的自定义异常被默认流程写入标准输出后的结果是：

```
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/raise2.py", line 27, in __init__
        raise e
        ^^^^^^^
    RangeError: The legal range is [-20,50], but the current value is 100. Out
of range!
    Raise during initialization!
```

第二处是在类Thermometer的函数属性change()中，相应代码为：

```
    e = RangeError('Out of range!', low=-20, high=50, current=temperature)
    e.add_note("Raise during changing!")
    raise e
```

而抛出的自定义异常被默认流程写入标准输出后的结果是：

```
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/raise2.py", line 36, in change
        raise e
        ^^^^^^^
    RangeError: The legal range is [-20,50], but the current value is -30. Out
of range!
    Raise during changing!
```

从这个例子可以看出使用自定义异常的3个技巧：

➤ 可以给代表自定义异常的类的__init__魔术属性添加任意仅关键字形式参数，并可以给异常添加基于这些参数计算出的任意实例属性。但要注意，一定要通过super()调用该类的基类的__init__魔术属性。
➤ 可以通过重写其所属类的__str__魔术属性控制自定义异常被转换成字符串时的格式，且该字符串可以包含该异常的实例属性中的信息。我们还可以通过super()调用该异常所属类的基类的__str__魔术属性，以为该字符串增加额外信息。当自定义异常被默认流程处理时，该字符串会作为错误消息被写入回溯信息。
➤ 在自定义异常被创建后，可以通过调用其add_note()属性为其增加一个字符串作为“笔记（note）”。当自定义异常被默认流程处理时，该笔记会在回溯信息之后被显示。

在本章开头已经提到了，add_note()是由BaseException提供的，因此事实上任何异常都能通过调用add_note()给自己添加笔记，不局限于自定义异常。被添加的笔记会被异常的__notes__实例属性记录，因此try语句处理该异常时可通过该属性获得它。实例属性__notes__仅当调用了add_note()后才会被创建，所以可以通过内置函数hasattr()来判断一个异常是否具有笔记。add_note()必须被传入一个字符串，否则会抛出TypeError异常。最后，add_note()和__notes__是Python 3.11引入的，更早版本的Python不支持。

---

raise语句的第三种变体是同时具有expression1和from子句，表示导致expression1对应的异常被抛出的原因是expression2对应的异常。对expression2求值得到的对象必须是BaseException或它的某个子类，或者它们的实例（前一种情况同样会自动创建该类的实例）。而被抛出的expression1对应的异常，将通过__cause__属性引用expression2对应的异常，同时其__suppress_context__属性将被设置为True。

下面的例子raise3.py说明了raise语句的第三种变体的用法：

```
import math


def f(x, y):
    try:
        return math.sqrt(x)/y
    except TypeError as e:
        raise RuntimeError("Please enter two numbers!") from e
    except ValueError as e:
        raise RuntimeError("x must be non-negative!") from SyntaxError
    except ZeroDivisionError as e:
        raise RuntimeError("y cannot be 0!") from None


def exception_chain(x, y):
    try:
        f(x, y)
    except Exception as e:
        print(repr(e.__context__))
        print(repr(e.__cause__))
```

```
            print(e.__suppress_context__)
```

该例子定义了两个函数：f()通过raise语句的第三种变体抛出了三个RuntimeError异常，但from子句分别指定了一个异常、一个异常类和None；exception_chain()则用于分析被抛出的RuntimeError异常的三个异常链相关属性。

请通过如下命令行和语句验证raise语句的第三种变体的用法：

```
$ python3 -i raise3.py

>>> f('a', 'b')
Traceback (most recent call last):
  File "/Users/wwy/raise3.py", line 6, in f
    return math.sqrt(x)/y
           ^^^^^^^^^^^^
TypeError: must be real number, not str

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise3.py", line 8, in f
    raise RuntimeError("Please enter two numbers!") from e
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
RuntimeError: Please enter two numbers!
>>>
>>> exception_chain('a', 'b')
TypeError('must be real number, not str')
TypeError('must be real number, not str')
True
>>>
>>> f(-1, 2)
SyntaxError: None

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise3.py", line 10, in f
    raise RuntimeError("x must be non-negative!") from SyntaxError
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
RuntimeError: x must be non-negative!
>>>
>>> exception_chain(-1, 2)
ValueError('math domain error')
SyntaxError()
True
>>>
>>> f(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise3.py", line 12, in f
    raise RuntimeError("y cannot be 0!") from None
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
RuntimeError: y cannot be 0!
>>>
>>> exception_chain(1, 0)
ZeroDivisionError('float division by zero')
None
True
```

```
>>>
```

额外值得一题的是，从调用f(1, 0)和exception_chain(1, 0)的结果可知，语句“raise ... from None”可在不破坏异常链结构的前提下只显示最后一个异常，而忽略之前的所有异常，这是一个非常有用的技巧。

在本章前面提到过，我们可以通过types.TracebackType()显式创建一个回溯对象，然后通过异常的with_traceback()属性将该回溯对象关联到一个异常。这个技巧通常与raise语句联合使用。下面的例子raise4.py通过这个技巧修改了抛出的RuntimeError异常的回溯对象：

```
import math
import sys
import types

def f(x, y):
    try:
        return math.sqrt(x)/y
    except Exception:
        t, v, tb = sys.exc_info()
        new_tb = types.TracebackType(None, tb.tb_frame, 1, 1)
        if t == TypeError:
           E = RuntimeError('Please enter two numbers!')
            E.with_traceback(new_tb)
            raise E from None
        elif t == ValueError:
            E = RuntimeError('x must be non-negative!')
            E.with_traceback(new_tb)
            raise E from None
        else:
            E = RuntimeError('y cannot be 0!')
            E.with_traceback(new_tb)
            raise E from None
```

请通过下面的命令行和语句验证：

```
$ python3 -i raise4.py

>>> f('a', 'b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise4.py", line 14, in f
    raise E from None
    ^^^^^^^^^^^^^^^^
  File "/Users/wwy/raise4.py", line 1, in f
    import math

RuntimeError: Please enter two numbers!
>>>
>>> f(-1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/raise4.py", line 18, in f
    raise E from None
    ^^^^^^^^^^^^^^^^
  File "/Users/wwy/raise4.py", line 1, in f
```

```
        import math

    RuntimeError: x must be non-negative!
    >>>
    >>> f(1, 0)
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/raise4.py", line 22, in f
        raise E from None
        ^^^^^^^^^^^^^^^^
      File "/Users/wwy/raise4.py", line 1, in f
        import math

    RuntimeError: y cannot be 0!
    >>>
```

该结果说明，即便通过types.TracebackType创建的回溯对象的tb_next属性被设置为了None，当将该回溯对象作为with_traceback()属性的参数后，该属性也会自动被设置以形成完整的回溯路径，因此有意义的设置仅涉及tb_lasti和tb_lineno属性，即通过它们指向其他代码。

值得一提的是，在Python 3.10以及更早的版本中，raise语句和sys.exc_info()存在一个bug，即如果异常链中最后一个异常的回溯对象被修改了，该异常被重新抛出或显示相关信息时依然会使用原来的回溯对象。Python 3.11修复了这个bug。

8-6. 异常组

（教程：8.9）
（语言参考手册：8.4）
（标准库：内置异常）

当编写较复杂的Python脚本时，有些情况下需要同时抛出多个异常（例如响应某一请求导致多个线程并行抛出异常）；有些情况下对某些异常的处理方式是将其暂时记录下来，等收集齐所有异常后再一次性抛出（这有点像finally子句的加强版）。Python 3.10及之前的Python版本不具有处理这些情况的能力。Python 3.11则通过引入被称为“异常组（exception groups）”的新机制来应对这些情况。

有两个实现异常组的类：ExceptionGroup和BaseExceptionGroup。ExceptionGroup是Exception的直接子类，而BaseExceptionGroup是BaseException的直接子类，这意味着异常组从逻辑上被视为特殊的异常。

ExceptionGroup的实例化语法为：

```
class ExceptionGroup(msg, excs)
```

而BaseExceptionGroup的实例化语法为：

```
class BaseExceptionGroup(msg, excs)
```

其中msg参数是一个字符串，excs参数是一个以异常作为元素的序列，含义都是将excs参数指定的异常打包为一个异常组，并以msg参数指定的字符串作为该异常组的消息。

两者的区别在于，ExceptionGroup的excs参数只能包含Exception及其子类（包括ExceptionGroup）的实例，而BaseExceptionGroup的excs参数则可以包含BaseException及其子类（包括ExceptionGroup和BaseExceptionGroup）的实例。换句话说，BaseExceptionGroup能打包任何异常，而能用ExceptionGroup打包的异常必须属于Exception或它的某个子类。特别的，如果实例化BaseExceptionGroup时其excs参数包含的所有异常都属于Exception或其子类，则会得到一个ExceptionGroup实例。

请通过如下命令行和语句创建一些异常组：

```
$ python3

>>> eg1 = ExceptionGroup("exception group 1", [NameError(name="o"),
AttributeError(name="a", obj="o")])
>>> eg2 = BaseExceptionGroup("exception group 2", [SystemExit(0),
KeyboardInterrupt(), GeneratorExit(), Exception()])
>>> eg3 = ExceptionGroup("exception group 3", [TypeError(), eg1])
>>> eg4 = BaseExceptionGroup("exception group 4", [ValueError(), eg2, eg3])
>>>
```

注意eg1和eg2只打包了普通的异常，eg3和eg4则打包了异常组。后两者说明，异常组是可以嵌套的。

每个异常组都通过message实例属性引用通过msg参数指定的字符串，通过exceptions实例属性引用通过excs参数指定的异常序列。这两个属性都是只读的，且不论excs参数是哪种类型的序列，exceptions属性都会引用一个元组。请通过如下语句验证：

```
>>> eg1.message
'exception group 1'
>>> eg1.exceptions
(NameError(), AttributeError())
>>>
>>> eg2.message
'exception group 2'
>>> eg2.exceptions
(SystemExit(0), KeyboardInterrupt(), GeneratorExit(), Exception())
>>>
>>> eg3.message
'exception group 3'
>>> eg3.exceptions
(TypeError(), ExceptionGroup('exception group 1', [NameError(),
AttributeError()]))
>>> eg3.exceptions[1].message
'exception group 1'
>>> eg3.exceptions[1].exceptions
(NameError(), AttributeError())
```

```
      >>>
      >>> eg4.message
      'exception group 4'
      >>> eg4.exceptions
      (ValueError(), BaseExceptionGroup('exception group 2', [SystemExit(0),
KeyboardInterrupt(), GeneratorExit(), Exception()]), ExceptionGroup('exception
group 3', [TypeError(), ExceptionGroup('exception group 1', [NameError(),
AttributeError()])]))
      >>> eg4.exceptions[1].message
      'exception group 2'
      >>> eg4.exceptions[1].exceptions
      (SystemExit(0), KeyboardInterrupt(), GeneratorExit(), Exception())
      >>> eg4.exceptions[2].message
      'exception group 3'
      >>> eg4.exceptions[2].exceptions
      (TypeError(), ExceptionGroup('exception group 1', [NameError(),
AttributeError()]))
      >>> eg4.exceptions[2].exceptions[1].message
      'exception group 1'
      >>> eg4.exceptions[2].exceptions[1].exceptions
      (NameError(), AttributeError())
      >>>
```

储存在异常组中的信息是不能改变的。使用异常组的方法是调用它的类属性subgroup()以获得一个子异常组，或者调用它的类属性split()以获得一个划分。这两个类属性都在内部调用类属性derive()。下面分别讨论这三个类属性。

derive()的语法为：

```
ExceptionGroup.derive(excs)
BaseExceptionGroup.derive(excs)
```

其中excs参数的含义同上。derive()的作用是返回一个与调用它的异常组同类型的异常组，后者拷贝前者的实例属性message、__context__、__cause__和__notes__，但基于excs参数设置其实例属性exceptions。请通过如下语句验证：

```
      >>> eg5 = eg1.derive([NameError(name="oo"), AttributeError(name="b",
obj="oo"), RuntimeError()])
      >>> eg5.message
'exception group 1'
      >>> eg5.exceptions
      (NameError(), AttributeError(), RuntimeError())
      >>>
```

subgroup()的语法为：

```
ExceptionGroup.subgroup(condition)
BaseExceptionGroup.subgroup(condition)
```

其中condition参数的可被传入下列对象：

➤ 一个代表异常的类，或者一个以代表异常的类为元素的元组，以从异常组中抽取匹配的异常。
➤ 一个返回布尔值的回调函数，该函数具有一个用于接收异常的参数，以从异常组中抽取所有使该回调函数返回True的异常。

如果异常组中具有至少1个满足condition参数指定条件的异常，则subgroup()将在内部以这些异常形成的序列为参数调用derive()，进而返回一个子异常组；否则，subgroup()返回None。请通过如下语句验证：

```
>>> eg2.subgroup(NameError)
>>> eg2.subgroup(SystemExit)
BaseExceptionGroup('exception group 2', [SystemExit(0)])
>>> eg2.subgroup((Exception, KeyboardInterrupt))
BaseExceptionGroup('exception group 2', [KeyboardInterrupt(), Exception()])
>>> eg2.subgroup(lambda e: isinstance(e, GeneratorExit))
BaseExceptionGroup('exception group 2', [GeneratorExit()])
>>> eg2.subgroup(lambda e: False)
>>>
```

split()的语法为：

```
ExceptionGroup.split(condition)
BaseExceptionGroup.split(condition)
```

其中condition参数的含义与在subgroup()中相同。split()会根据condition参数指定的条件将异常组中的异常划分为两部分——满足条件的异常和不满足条件的异常，然后分别以两部分为参数依次调用derive()两次，最后返回一个“(match, rest)”格式的元组，其中match代表满足条件的异常形成的子异常组，rest代表不满足条件的异常形成的子异常组。当然，如果划分中的某部分是空集，则会用None代替相应的子异常组。请通过如下语句验证：

```
>>> eg2.split(NameError)
(None, BaseExceptionGroup('exception group 2', [SystemExit(0),
KeyboardInterrupt(), GeneratorExit(), Exception()]))
>>> eg2.split(SystemExit)
(BaseExceptionGroup('exception group 2', [SystemExit(0)]),
BaseExceptionGroup('exception group 2', [KeyboardInterrupt(), GeneratorExit(),
Exception()]))
>>> eg2.split((Exception, KeyboardInterrupt))
```

```
      (BaseExceptionGroup('exception group 2', [KeyboardInterrupt(),
Exception()]), BaseExceptionGroup('exception group 2', [SystemExit(0),
GeneratorExit()]))
      >>> eg2.split(lambda e: isinstance(e, GeneratorExit))
      (BaseExceptionGroup('exception group 2', [GeneratorExit()]),
BaseExceptionGroup('exception group 2', [SystemExit(0), KeyboardInterrupt(),
Exception()]))
      >>> eg2.split(lambda e: False)
      (None, BaseExceptionGroup('exception group 2', [SystemExit(0),
KeyboardInterrupt(), GeneratorExit(), Exception()]))
      >>>
```

当我们以BaseExceptionGroup或ExceptionGroup为基类定义自己的异常组类时，必须重写derive()以使其返回属于该自定义类的异常组。此外，BaseExceptionGroup重写了__new__，因此如果自定义类需要重写__new__，则必须通过super()调用BaseExceptionGroup的__new__。

至此，我们已经对代表异常组的类有了足够的了解。下面讨论如何通过try语句来处理异常组。由于异常组被视为特殊的异常，所以可以通过except子句直接截获ExceptionGroup或BaseExceptionGroup类的“异常”，然后在该子句的代码块中通过上面介绍的异常组特有属性来处理。

下面是一个简单例子：

```
#该函数接收一个异常组，将其自动抛出，然后以except子句对异常组进行处理。
def eg_handler1(eg):
    try:
        raise(eg)
    except ExceptionGroup as e:
        print("An ExceptionGroup!")
        print(e.message)
        n = len(e.exceptions)
        i = 0
        while i < n:
            print(repr(e.exceptions[i]))
            i = i + 1
    except BaseExceptionGroup as e:
        print("An BaseExceptionGroup!")
        print(e.message)
        n = len(e.exceptions)
        i = 0
        while i < n:
            print(repr(e.exceptions[i]))
            i = i + 1
    return None
```

该例子通过except子句截获一个异常组，显示异常组的消息，以及打包的每一个异常。请将上述代码保存为exception_group.py，然后通过如下语句验证：

```
>>> from exception_group import *
>>>
>>> eg_handler(eg1)
An ExceptionGroup!
exception group 1
NameError()
AttributeError()
>>>
>>> eg_handler(eg2)
An BaseExceptionGroup!
exception group 2
SystemExit(0)
KeyboardInterrupt()
GeneratorExit()
Exception()
>>>
>>> eg_handler(eg3)
An ExceptionGroup!
exception group 3
TypeError()
ExceptionGroup('exception group 1', [NameError(), AttributeError()])
>>>
>>> eg_handler(eg4)
An BaseExceptionGroup!
exception group 4
ValueError()
BaseExceptionGroup('exception group 2', [SystemExit(0),
KeyboardInterrupt(), GeneratorExit(), Exception()])
ExceptionGroup('exception group 3', [TypeError(), ExceptionGroup('exception
group 1', [NameError(), AttributeError()])])
>>>
```

注意截获ExceptionGroup的except子句必须在截获BaseExceptionGroup的except子句之前，以避免后者将前者屏蔽。

上述例子对异常组的处理过于简单。大多数情况下需要针对异常组内的不同异常进行不同处理，导致except子句下的代码块非常复杂。然而我们无法用标准的try语句将这些代码分散到多条except子句中去，因为每个try语句都至多有一条except子句被执行。如果截获异常组的except子句只通过split()处理异常组中的部分异常，然后将剩余未处理的异常打包成一个新异常组抛出，则新异常组将成为整个try语句抛出的异常。该异常必须通过外层try语句来处理，这是很不方便的。

为了解决这一问题，Python 3.11扩展了try语句的语法，引入了如下变体：

```
try:
    suite0
except* expression1 [as identifier1]:
    suite1
except* expression2 [as identifier2]:
    suite2
```

```
...
else:
    suiteE
finally:
    suiteF
```

也就是用except*子句代替了except子句。这样的try语句专门用来处理异常组，except*子句指定的异常将与部分异常组打包的异常匹配，而相应语句块仅处理匹配的异常。当一个异常组中的某个异常被某条except*子句处理后，就会从该异常组中被删除，而如果except*子句下的代码块内没有return语句、continue语句或break语句，则该异常组会被传递给下一条except*子句来处理，直到该异常组中的所有异常都被处理，或者最后一条except*子句被使用时，才会执行finally子句。如果你熟悉C的话，就会明白try语句的这种变体的执行流程类似于C中的switch语句，而标准的try语句的执行流程则类似于C中的if语句。

下面的例子显示了能够通过except*子句处理完异常组中的所有异常的情形：

```
>>> import sys
>>> try:
...     raise(eg1)
... except* NameError as e:
...     print("Can not find object: " + e.exceptions[0].name)
... except* TypeError as e:
...     print("Bad type!")
... except* AttributeError as e:
...     print("Can not find attribute: " + e.exceptions[0].obj + "." +
e.exceptions[0].name)
... except* ValueError as e:
...     print("Illegal value!")
... finally:
...     print(repr(sys.exception()))
...
Can not find object: o
Can not find attribute: o.a
None
>>>
```

在这个例子中被抛出的eg1的处理流程是这样的：

➤ 步骤一：“except* NameError as e”子句截获异常组，并处理了其内包含的NameError异常，使得异常组内仅包含AttributeError异常。
➤ 步骤二：“except* TypeError as e”子句没有匹配异常组中的任何异常。
➤ 步骤三：“except* AttributeError as e”子句截获异常组，并处理了其内包含的AttributeError异常。
➤ 步骤四：由于异常组变成了空的，所以“except* ValueError as e”被跳过，直接执行finally子句，而此时sys.exception()将返回None。

对于这个例子，还需要特别注意except*子句中的“as e”使标识符e引用了哪个对象。这里需要强调，e既没有引用传到该子句时的异常组，也没有引用该异常组中包含的某个异常。

事实上，当异常组被传递给一条except*子句时，会以该子句中的异常类或异常类的元组作为参数调用该异常组的split()属性，得到的“(match, rest)”格式的元组中的rest将被暂时保存起来，而match则被e引用。换句话说，e引用的是一个新生成的异常组，与原异常组的区别仅在于其exceptions实例属性仅包含符合匹配条件的异常。在该except*子句的代码块执行完成后，如果没有遇到return语句、continue语句或break语句，e引用的match将被丢弃，被保存起来的rest将传递给下一条except*子句（除非它已经是空的）。

下面的例子显示了未能通过except*子句处理完异常组中的所有异常的情形：

```
>>> import sys
>>> try:
...     raise(eg2)
... except* (SystemExit, KeyboardInterrupt, GeneratorExit):
...     print("Some non-Exception instance exceptions occurred!")
... finally:
...     print(repr(sys.exception()))
...     print("")
...
Some non-Exception instance exceptions occurred!
ExceptionGroup('exception group 2', [Exception()])

  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: exception group 2 (1 sub-exception)
  +-+---------------- 1 ----------------
    | Exception
    +------------------------------------
>>>
```

在该例子中，“except* (SystemExit, KeyboardInterrupt, GeneratorExit)”子句使得异常组eg2中包含的SystemExit、KeyboardInterrupt和GeneratorExit异常被处理，然而还剩下一个Exception异常随异常组传给了finally子句，而由于finally子句不包含return语句，所以在它被显示之后被默认流程处理。Python 3.11扩展了异常处理默认流程，使得针对异常组的回溯信息可以分层显示其内包含的每个异常。

当异常组包含异常组时，对其的处理将变得非常麻烦。原因有如下两方面：

1. except*子句不能用来从异常组中提取异常组，也就是说它用于匹配异常的表达式中不能包含ExceptionGroup和BaseExceptionGroup。这意味着我们只能用except子句来匹配异常组。

2. 在一个try语句中不能同时出现except子句和except*子句，亦即标准try语句和它的变体不能混合。这意味着我们只能通过try语句和它的变体的嵌套来处理异常组内的异常组。

下面的例子显示了嵌套异常组的处理方法：

```
>>> import sys
>>> try:
...     try:
...         raise(eg3)
...     except* TypeError as e:
...         print(repr(e))
... except ExceptionGroup as e:
...     try:
...         raise(e.exceptions[0])
...     except* NameError as e:
...         print(repr(e))
...     except* AttributeError as e:
...         print(repr(e))
... finally:
...     print(repr(sys.exception()))
...
ExceptionGroup('exception group 3', [TypeError()])
ExceptionGroup('exception group 1', [NameError()])
ExceptionGroup('exception group 1', [AttributeError()])
None
>>>
```

在该例子中，内层try语句的变体通过except*子句处理异常组中的所有普通异常，使它仅包含异常组；而外层标准try语句则通过except子句定位到这个异常组，由其exceptions属性提取出其内包含的异常组，再通过另外一个内层try语句的变体来处理。

需要强调的是，上面的例子不具有通用性，而且我们也很难设计出能够处理任意结构的嵌套异常组的代码。这是因为ExceptionGroup是Exception的子类，不能简单地用Exception来匹配异常组中的所有普通异常；同理，也不能简单地用BaseException来匹配异常组中的所有普通异常。另外，异常组相互嵌套每多一层，处理它的try语句嵌套也就需要多一层（请想象一下如何用try语句处理eg4）。因此除非万不得已，否则应保持异常组中只包含普通异常，强烈不建议异常组相互嵌套。

8-7. 上下文管理器

（教程：8.8）
（语言参考手册：3.3.9、8.5）
（标准库：内置类型）

对于那些无论怎么做都无法避免抛出异常的对象（例如文件对象），使用时需要自行捕获处理其抛出的异常，以避免程序因此退出。然而从上面的例子可以看出，try语句并不简洁，广泛使用会严重降低代码的可读性。“上下文管理器（context managers）”技术就是为了解决这一问题而被引入的。

上下文管理器是实现了魔术属性__enter__和__exit__的类的实例，它们仅在with语句中才能发挥作用。with语句的语法为：

```
with expression [as target], ... :
    suite
```

其中每个expression代表的表达式被求值后都得到一个上下文管理器，而该上下文管理器的__enter__会被自动调用，其返回值在存在as子句的情况下会被赋值给target标识符，而这些标识符可以在suite中被使用。当suite执行完成时，该上下文管理器的__exit__会被自动调用，并被传入（None, None, None），其返回值会被忽略。如果执行suite的过程中抛出了异常，则该异常会自动被__exit__截获，其回溯信息会被作为__exit__的参数传入，而__exit__的返回值决定了该异常是否会被继续抛出。

下面说明with语句本质上是try语句的语法糖。首先请注意：

```
with expression1 [as target1], expression2 [as target2]:
    suite
```

就等价于

```
with expression1 [as target1]:
    with expression2 [as target2]:
        suite
```

所以下面只考虑with语句中只有一个上下文管理器的情况，即：

```
with expression [as target]:
    suite
```

而它就等价于：

```
context_manager = expression
[target = ]context_manager.__enter__()
try:
    suite
except:
    if not context_manager.__exit__(*sys.exc_info()):
        raise
else:
    context_manager.__exit__(None, None, None)
```

上下文管理器的__enter__不需要实际参数，可以返回上下文管理器自身，也可以返回另一个对象。如果with语句中存在as子句，则该返回值会被赋值给指定的标识符，而在with语句执行完后该标识符依然存在，因此正如第3章提到的，这也是一种标识符绑定操作。当然，如果with语句中没有as子句，那么__enter__的返回值就会直接进入垃圾回收流程，但__enter__被调用依然有意义，因为它可能改变上下文管理器的状态。

上下文管理器的__exit__需要传入三个实际参数，正好与sys.exc_info()的返回值一一对应。通过这些参数，__exit__可以判断执行suite的过程中抛出了哪种异常，进而进行处理。__exit__可以只处理某些类型的异常（遇到它们时返回True），其他类型的异常则透明地抛出（遇到它们时返回False），就好像一个过滤器一样。在被传入（None, None, None）时，__exit__得知执行suite的过程中没有抛出异常，此时它应该直接返回True（虽然该返回值会被忽略）。如果执行__exit__自身的过程中抛出了新的异常，则它会覆盖原来的异常，并导致异常链增长，就好像try语句的except子句那样。

下面的例子context_manager.py定义了一个上下文管理器类MyContextManager：

```
import math


class MyContextManager():
    def __enter__(self):
        def f(x, y):
            return math.sqrt(x)/y
        return f

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type == TypeError:
            print('Please enter two numbers!')
            return True
        elif exc_type == ValueError:
            print('x must be non-negative!')
            return True
        else:
            return False
```

该类的__enter__返回一个函数f(x, y)，该函数被传入不同参数时可能抛出包括TypeError、ValueError和ZeroDivisionError在内的多种异常；而其__exit__则仅截获并处理TypeError和ValueError异常，允许其他异常被抛出。

请通过下面的命令行和语句验证上下文管理器的用法：

```
$ python3 -i context_manager.py

>>> with MyContextManager() as f:
...     f(1, 2)
...
0.5
>>> with MyContextManager() as f:
...     f('a', 'b')
...
Please enter two numbers!
>>> with MyContextManager() as f:
...     f(-1, 2)
...
x must be non-negative!
>>> with MyContextManager() as f:
...     f(1, 0)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

```
    File "/Users/wwy/context_manager.py", line 7, in f
      return math.sqrt(x)/y
             ~~~~~~~~~~~~^~
ZeroDivisionError: float division by zero
>>>
```

Python解释器本身已经定义了不少上下文管理器。第7章提到过，IOBase实现了魔术属性__enter__和__exit__，故所有文件对象都是上下文管理器。调用文件对象的__enter__将返回该文件对象本身，而调用文件对象的__exit__将导致调用其close()属性。因此，使用文件对象最安全的方法是遵循如下范式：

```
with open("somefile", ...) as fd:
    ... any code using fd ...
```

这样做的优点是即便在使用文件对象的过程中抛出了异常，也会保证调用其close()属性。

最后值得一提的是，标准库中的contextlib模块提供了很多定义上下文管理器的工具，但对该模块的详细讨论超出了本书的范围。

8-8. 警告

（标准库：内置异常、warnings）

Python解释器在执行脚本时可能会遇到一些值得提醒用户注意但又不值得为此终止程序的情况，此时就需要使用“警告（warning）”这一机制。Warning是Exception的直接子类，而从Warning派生出的各种异常类被作为警告使用，总结在表8-6中。注意由于它们从技术上讲派生自Exception，所以同样可以通过raise语句抛出；但从逻辑上讲它们不同于其他异常，因此几乎总是通过标准库中的warnings模块来处理。

表8-6. Warning的直接子类

类	说明
SyntaxWarning	语法相关警告。
BytesWarning	二进制序列相关警告。
UnicodeWarning	Unicode相关警告。
EncodingWarning	文本流使用了操作系统的本地字符编码方式。
RuntimeWarning	运行时相关警告。
FutureWarning	使用了已经弃用特性。针对最终用户。
DeprecationWarning	使用了已经弃用特性。针对开发者。除非由主模块发布，否则默认会被忽略。在Python开发模式会被显示。
PendingDeprecationWarning	使用了预计在未来弃用的特性。默认会被忽略。在Python开发模式会被显示。
ImportWarning	模块导入相关警告。默认会被忽略。在Python开发模式会被显示。

类	说明
ResourceWarning	资源使用相关警告。默认会被忽略。在Python开发模式会被显示。
UserWarning	用户自定义警告。

对于最终用户来说，可以看到的由Python解释器自动发布的警告包括SyntaxWarning警告、BytesWarning警告、UnicodeWarning警告、RuntimeWarning警告和FutureWarning警告。EncodingWarning警告仅当启动解释器时添加了-X warn_default_encoding或设置了PYTHONWARNDEFAULTENCODING环境变量时才会被发布。如果开启了开发模式（将在第16章讨论），则可以看到DeprecationWarning警告、PendingDeprecationWarning警告、ImportWarning警告和ResourceWarning警告。UserWarning警告不会由Python解释器自动发布，它是所有用户自定义警告类型的基类，其自身也能被实例化。

发布一个警告最基本的方法是使用warnings.warn_explicit()函数。

该函数的语法为：

```
warnings.warn_explicit(message, category, filename, lineno, module=None, registry=None, module_globals=None, source=None)
```

其各参数的含义是：

- message：传入一个字符串或一个Warning或其子类的实例，代表警告消息。
- category：传入Warning或它的某个子类，代表警告类型。当message参数被传入了一个警告实例时，会直接从其__class__提取该信息，category参数会被忽略（因此可以被传入None）。
- filename：传入一个字符串，为导致该警告被发布的代码所在源文件的路径。
- lineno：传入一个整数，为导致该警告被发布的代码在源文件中的行号。
- module：传入一个字符串，为源文件对应模块的模块名。如果被传入None，则自动将源文件的文件名去掉后缀名.py作为模块名。
- registry：传入一个字典，为模块的注册表。如果被传入None，则先尝试从模块的实例属性__warningregistr__获取，当模块不具有该实例属性时则保持None。
- module_globals：传入一个字典，为导致该警告被发布的代码执行时的全局名字空间。如果传入None，则会自动获取。
- source：仅当发布ResourceWarning警告时才有意义，为被销毁的对象。

下面是一个使用warnings.warn_explicit()函数的例子：

```
import warnings


#调用该函数会发布一个用户自定义警告。
def f(a, b, c):
    warnings.warn_explicit(
            'Using this function should be warned!',
            UserWarning,
            f.__code__.co_filename,
            f.__code__.co_firstlineno + 1,
    )
    return a*b-c
```

请将上述代码保存为warnings1.py，然后通过如下命令行和语句验证：

```
$ python3 -i warnings1.py

>>> f(1, 2, 3)
/Users/wwy/warnings1.py:6: UserWarning: Using this function should be
warned!
  warnings.warn_explicit(
-1
>>> f(6, 0, 9)
/Users/wwy/warnings1.py:6: UserWarning: Using this function should be
warned!
  warnings.warn_explicit(
-9
>>>
```

函数warnings.warn()是对warnings.warn_explicit()的简化。

该函数的语法为：

```
warnings.warn(message, category=None, stacklevel=1,
source=None)
```

warnings.warn()会在内部调用warnings.warn_explicit()，前者的message、category和source参数会被直接传递给后者（但当message被传入字符串而category被传入None时，默认发布UserWarning警告）；前者的stacklevel参数则用于设置后者的filename和lineno参数，其取值含义是在函数栈中的回溯深度，1代表当前帧对应的函数，2代表调用该函数的函数，……，依此类推，并将filename参数和lineno参数设置成指向该函数；而后者的module、registry和module_globals参数则总是被传入None。

下面的例子warnings2.py是对warnings1.py的改写，用warnings.warn()代替了warnings.warn_explicit()，功能则保持不变：

```
import warnings


def f(a, b, c):
    warnings.warn('Using this function should be warned!')
    return a*b-c
```

请通过如下命令行和语句验证：

```
$ python3 -i warnings2.py

>>> f(1, 2, 3)
/Users/wwy/warnings2.py:5: UserWarning: Using this function should be
warned!
  warnings.warn('Using this function should be warned!')
-1
>>> f(6, 0, 9)
-9
>>>
```

注意与前一个例子不同，第二次调用f()时没有显示警告，这是因为警告过滤器的影响，会在后面详细讨论。

从上面的例子可以看出，Python解释器对警告的默认处理方式是将一条警告信息写入标准出错，而这并不会影响后续代码的执行。这是通过调用warnings.showwarning()实现的。

该函数的语法为：

```
warnings.showwarning(message, category, filename, lineno,
file=None, line=None)
```

其前4个参数的含义与在warnings.warn_explicit()中相同；file参数必须被传入一个文件对象，以指定将警告信息写入哪个文件，如果传入None则默认使用标准出错；而line参数则被传入导致该警告被发布的代码，如果被传入None则选择filename参数和lineno参数共同指定的那行源代码。

下面的例子warnings3.py是对warnings1.py的改写，通过使用warnings.showwarning()将警告写入了warning.log文件，而非标准出错：

```
import warnings


#调用该函数会将一个用户自定义警告写入warning.log文件。
def f(a, b, c):
```

```
        with open('warning.log', 'a') as fd:
            warnings.showwarning(
                    'Using this function should be warned!',
                    UserWarning,
                    f.__code__.co_filename,
                    f.__code__.co_firstlineno + 2,
                    fd
            )
        return a*b-c
```

请通过如下命令行和语句验证：

```
    $ python3 -i warnings3.py

    >>> f(1, 2, 3)
    -1
    >>> f(6, 0, 9)
    -9
    >>>
```

注意在工作目录下会多出warning.log文件，其内容为：

```
    /Users/wwy/warnings3.py:7: UserWarning: Using this function should be
warned!
      warnings.showwarning(
    /Users/wwy/warnings3.py:7: UserWarning: Using this function should be
warned!
      warnings.showwarning(
```

函数warnings.showwarning()会在内部调用warnings.formatwarning()，以将警告转换为具有标准格式的警告信息。

从上面的例子可以看出，标准的警告信息由两行组成，第一行的格式为：

```
    filename:lineno: category: message
```

而第二行则是导致该警告被发布的代码。

warnings.formatwarning()的语法为：

```
    warnings.formatwarning(message, category, filename, lineno,
line=None)
```

其所有参数都与在warnings.showwarning()中相同。

下面的例子warnings4.py直接调用warnings.formatwarning()获得警告信息，然后通过print()将其输出到标准输出：

```
import warnings


#调用该函数会将一段用户自定义警告信息写入标准输出。
def f(a, b, c):
    msg = warnings.formatwarning(
            'Using this function should be warned!',
            UserWarning,
            f.__code__.co_filename,
            f.__code__.co_firstlineno + 1,
    )
    print(msg)
    return a*b-c
```

请通过如下命令行和语句验证：

```
$ python3 -i warnings4.py

>>> f(1, 2, 3)
/Users/wwy/warnings4.py:6: UserWarning: Using this function should be
warned!
  msg = warnings.formatwarning(

-1
>>> f(6, 0, 9)
/Users/wwy/warnings4.py:6: UserWarning: Using this function should be
warned!
  msg = warnings.formatwarning(

-9
>>>
```

需要强调的是，warnings.showwarning是可写的，可以给其赋值任何可调用对象（但形式参数列表需保持不变）。这使得我们可以改变警告信息的输出格式。请看下面的例子warnings5.py：

```
import warnings


def __myshowwarning(message, category, filename, lineno, file=None,
line=None):
    if isinstance(message, Warning):
        category = message.__class__
    msg = (str(category) + ": " + str(message) + "\n"
            + filename + ", line " + str(lineno) + "\n")
    if line is not None:
        msg = msg + "  " + line + "\n"
    print(msg)
```

```
    warnings.showwarning = __myshowwarning


    def f(a, b, c):
        warnings.warn('Using this function should be warned!')
        return a*b-c
```

该例子将自定义格式的警告信息写入了标准输出。

请通过如下命令行和语句验证：

```
    $ python3 -i warnings5.py

    >>> f(1, 2, 3)
    <class 'UserWarning'>: Using this function should be warned!
    /Users/wwy/warnings5.py, line 18

    -1
    >>>
```

除了修改warnings.showwarning()之外，我们还可以通过警告过滤器控制对被发布警告的处理。

警告过滤器就是一个如下格式的文本行：

```
    action:message:category:module:lineno
```

其中后面四个字段与warnings.warn_explicit()中的同名参数一一对应，用于匹配警告，但取值是这样的：

- message：一个正则表达式，用于匹配警告信息的开头，匹配时不区分大小写。
- category：Warning或其某个子类的类名，警告必须是该类的实例才能匹配。
- module：一个正则表达式，用于匹配导致警告被发布的代码所在模块的模块名，匹配时区分大小写。
- lineno：一个整数，用于匹配导致警告被发布的代码的行号，0可以匹配任意行号。

注意这4个字段不需要同时出现，可以任意省略，被省略的字段表示无限制，而如果4个字段都被省略则表示匹配任何警告。action则表示对匹配的警告执行的动作，可取值被总结在表8-7中。

表8-7. 警告过滤器可能的动作

动作	说明
"default"	将该警告写入标准出错，但通过warn()发布重复的警告（message+category）时为每个位置（module+lineno）至多写入一次。
"error"	将该警告当成普通异常用默认流程处理。
"always"	将该警告写入标准出错，不论它被发布了多少次。
"module"	将该警告写入标准出错，但通过warn()发布重复的警告（message+category）时为每个模块（module）至多写入一次。
"once"	将该警告写入标准出错，但通过warn()发布重复的警告（message+category）时至多写入一次。
"ignore"	忽略该警告，无任何额外操作。

Python解释器正常启动时具有下面的默认警告过滤器列表：

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning,
ignore::PendingDeprecationWarning,
ignore::ImportWarning,
ignore::ResourceWarning,
default,
```

这些过滤器的优先级从下到上逐渐增加。而当开启了开发模式时，默认警告过滤器列表则是空的。

在启动Python解释器时，可以通过-W选项或PYTHONWARNINGS环境变量来改变警告过滤器的设置，它们的取值都是用逗号分隔的警告过滤器序列，该序列中的过滤器会被依次添加到默认警告过滤器列表的顶部，因此优先级从左到右逐渐增加。此外，当同时具有多个-W选项时，后面的-W选项优先级高于前面的-W选项。特别的，-W选项还有如下快捷设置（方括号表示可以省略其内的部分）：

- -Wd[efault]：对所有警告都采用default动作。
- -We[rror]：对所有警告都采用error动作。
- -Wa[lways]：对所有警告都采用always动作。
- -Wm[odule]：对所有警告都采用module动作。
- -Wo[nce]：对所有警告都采用once动作。
- -Wi[gnore]：对所有警告都采用ignore动作。

在Python脚本的运行过程中，也可以通过三个函数来动态改变警告过滤器的设置。

首先是warnings.filterwarnings()，功能是向警告过滤器列表添加一个过滤器，语法为：

```
warnings.filterwarnings(action, message='', category=Warning,
module='', lineno=0, append=False)
```

其前5个参数都用于描述该过滤器，而append则用于指定将该过滤器添加的列表的顶部（传入False）还是底部（传入True）。

warnings.simplefilter()是warnings.filterwarnings()的简化版，语法为：

```
warnings.simplefilter(action, category=Warning, lineno=0,
append=False)
```

通过它添加的警告过滤器的message字段和module字段总是被省略。

warnings.resetwarnings()用于将警告过滤器列表恢复到默认状态，其语法为：

```
warnings.resetwarnings()
```

注意调用该函数不仅会取消之前调用warnings.filterwarning()和warnings.simplefilter()的效果，还会取消-W选项和PYTHONWARNINGS环境变量的效果。

下面的例子说明了warnings.filterwarnings()和warnings.resetwarnings()的作用：

```
$ python3 -i warnings2.py

>>> warnings.filterwarnings('ignore', category=UserWarning)
>>> f(1, 2, 3)
-1
>>> warnings.resetwarnings()
>>> f(6, 0, 9)
/Users/wwy/warnings2.py:5: UserWarning: Using this function should be
warned!
  warnings.warn('Using this function should be warned!')
-9
>>>
```

下面的例子则说明了warnings.simplefilter()的作用：

```
$ python3 -i warnings2.py

>>> warnings.simplefilter('error', UserWarning)
>>> f(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/warnings2.py", line 5, in f
```

```
        warnings.warn('Using this function should be warned!')
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    UserWarning: Using this function should be warned!
    >>> warnings.simplefilter('ignore', UserWarning)
    >>> f(6, 0, 9)
    -9
    >>>
```

最后，在有些情况下我们希望临时改变warnings.showwarning()和警告过滤器的设置，此时最便捷的方案是使用上下文管理器warnings.catch_warnings。

该函数语法为：

```
    class warnings.catch_warnings(*, record=False, module=None, action=None, category=Warning, lineno=0, append=False)
```

该上下文管理器的__enter__会将当前的warnings.showwarning()和警告过滤器设置保存起来，而__exit__则会利用这些被保存起来的对象还原到进入上下文管理器之前的状态。warnings.catch_warnings()默认返回None，但record参数是True时会返回一个当前储存的所有warnings.showwarning()对象形成的列表（对warnings. catch_warnings()的调用可以嵌套）。而module参数是为了测试warnings模块本身使用的，我们几乎不需要用到它。

action、category、lineno和append是Python 3.11给warnings.catch_warnings()添加的。当action参数不是None时，相当于给with语句的代码块顶部添加一次对warnings.simplefilter()的调用，而调用时将使用这四个参数。

下面的例子说明了warnings.showwarning()的作用：

```
    $ python3 -i warnings2.py

    >>> with warnings.catch_warnings():
    ...     def g(message, category, filename, lineno, file=None, line=None):
    ...         print("A warning occured!")
    ...     warnings.showwarning = g
    ...     warnings.simplefilter('always')
    ...     f(1, 2, 3)
    ...     f(6, 0, 9)
    ...
    A warning occured!
    -1
    A warning occured!
    -9
    >>> f(1, 2, 3)
    /Users/wwy/warnings2.py:5: UserWarning: Using this function should be warned!
      warnings.warn('Using this function should be warned!')
    -1
    >>> f(6, 0, 9)
```

```
-9
>>>
```

而当我们使用Python 3.11时，可以将上面的with语句写成：

```
>>> with warnings.catch_warnings(action='always'):
...     def g(message, category, filename, lineno, file=None, line=None):
...         print("A warning occured!")
...     warnings.showwarning = g
...     f(1, 2, 3)
...     f(6, 0, 9)
...
```

效果与上面完全相同。