第十一章 - 异常处理

张建章

阿里巴巴商学院

杭州师范大学

2024-09

异常（Exception）是指在程序执行过程中出现的错误或意外情况，导致程序无法按照预期继续运行。异常处理机制使程序能够捕获并处理这些错误，确保程序的稳健性和可靠性。

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print(result)
except ZeroDivisionError:
    print(" 错误: 除数不能为零。")
```

在上述代码中，`try`块包含可能引发异常的代码。当执行到`result = numerator / denominator`时，由于`denominator`为零，会引发`ZeroDivisionError`异常。此时，程序跳转到对应的`except`块，输出提示信息“错误：除数不能为零。”。通过这种方式，程序避免了因未处理的异常而崩溃。

异常传播机制指的是当异常在当前作用域未被捕获时，沿调用栈向上传递，直到被捕获或导致程序终止的过程。这种机制确保异常信息能够传递给适当的处理程序，以便采取相应的措施。

以下示例演示了异常的传播过程：

```
def function_a():
    function_b()

def function_b():
    function_c()

def function_c():
    raise ValueError(" 发生了一个值错误")

try:
    function_a()
except ValueError as e:
    print(f" 捕获到异常: {e}")
```

在上述代码中，`function_c`中显式引发了`ValueError`异常。由于`function_c`内部没有处理该异常，异常被传播到调用它的`function_b`。同样地，`function_b`也未处理该异常，异常继续传播到`function_a`。最终，异常传播到`function_a`的调用者，即`try`块。在此处，存在匹配的`except`子句来捕获`ValueError`异常，因此异常被成功捕获，程序输出相应的提示信息。

异常处理是确保程序稳健性和可靠性的重要机制。Python 提供了丰富的内置异常类，用于捕获和处理各种错误情况。这些异常类均继承自 `BaseException`，并形成层次化的结构，方便开发者根据具体需求进行捕获和处理。以下是一些常见的 Python 内置异常类及其使用示例：

1. ZeroDivisionError

当尝试除以零时引发。

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print(" 错误: 除数不能为零。")
```

2. ValueError

当函数接收到参数类型正确但值不合适时引发。

```
try:
    number = int("abc")
except ValueError:
    print(" 错误: 无法将字符串转换为整数。")
```

3. TypeError

当操作或函数应用于不适当类型的对象时引发。

```
try:
    result = '2' + 2
except TypeError:
    print(" 错误: 不能将字符串与整数相加。")
```

4. IndexError

当尝试访问序列中不存在的索引时引发。

```
try:
    numbers = [1, 2, 3]
    print(numbers[5])
except IndexError:
    print(" 错误: 索引超出列表范围。")
```

5. KeyError

当在字典中使用一个不存在的键时引发。

```
try:
    data = {'name': 'Alice'}
    print(data['age'])
except KeyError:
    print(" 错误: 键不存在于字典中。")
```

6. FileNotFoundError

当尝试打开一个不存在的文件时引发。

```
try:
    with open('nonexistent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print(" 错误: 文件未找到。")
```

7. AttributeError

当尝试访问对象中不存在的属性时引发。

```
try:
    t = (1,2,3)
    t.pop()
except AttributeError:
    print(" 错误: 对象没有该属性或方法。")
```

8. ModuleNotFoundError

导入的模块不存在或模块路径不在 `sys.path` 中时引发。

```
try:
    import nonexistent_module
except ModuleNotFoundError:
    print(" 错误: 模块导入失败。")
```

了解并正确处理这些内置异常，有助于编写健壮的 Python 程序，确保在各种错误情况下程序能够优雅地处理并继续运行。讲义中的表 11.1 列出了常见的 Python 内置异常类及其含义。

在 Python 编程中，异常处理机制通过 `try`、`except`、`else`、`finally` 和 `raise` 语句来管理程序运行时的错误。

- `raise` 语句允许程序员在特定条件下主动引发异常，以便在检测到错误或异常情况时中断正常的程序流程，并将控制权交给相应的异常处理器。
- `try` 块包含可能引发异常的代码；
- `except` 块用于捕获并处理这些异常；
- `else` 块在未发生异常时执行；
- `finally` 块无论是否发生异常都会执行；

1. raise 语句

`raise` 语句用于显式引发异常，以便在程序运行过程中处理特定的错误或异常情况。通过使用 `raise`，可以在代码中指定何时以及为何引发异常，从而提高程序的健壮性和可维护性。

(1) 引发通用的异常：

```
raise Exception(" 这是一个通用异常的提示信息")
```

上述代码将引发一个通用的 `Exception` 异常，并附带自定义的错误消息。

(2) 引发特定类型的异常：

```
raise ValueError(" 无效的输入")
```

此示例引发一个 `ValueError` 异常，通常用于表示传入函数的参数具有有效类型但不在期望的值范围内。

2. try/except 语句

`try/except` 语句用于捕获和处理程序运行时可能发生的异常，确保程序的稳健性和可靠性。其基本结构如下：

```
try:
    # 可能引发异常的代码
except 异常类型:
    # 处理异常的代码
```

在 `try` 块中编写可能引发异常的代码；如果发生指定类型的异常，程序将跳转到对应的 `except` 块执行处理代码。

以下示例演示了如何使用 `try/except` 语句处理除零错误：

```
def divide_numbers(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print(" 错误：除数不能为零。")
        return None


# 测试代码
num1 = 10
num2 = 0
output = divide_numbers(num1, num2)
if output is not None:
    print(f" 结果：{output}")
```

3. else 子句

`else` 子句紧跟在所有 `except` 子句之后，仅在 `try` 块未引发任何异常时执行。这对于将正常执行路径与异常处理逻辑分开非常有用。

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print(" 除数不能为零。")
else:
    print(f" 结果是: {result}")
```

在上述代码中，`try` 块成功执行除法操作，未引发任何异常，因此执行 `else` 子句，输出计算结果。

4. finally 子句

`finally` 子句无论是否发生异常，都会执行，通常用于释放资源或执行清理操作。

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print(" 文件未找到。")
else:
    print(content)
finally:
    file.close()
    print(" 文件已关闭。")
```

在上述代码中，`try` 块尝试打开并读取文件内容。如果文件存在且读取成功，执行 `else` 子句，打印文件内容。无论是否发生异常，`finally` 子句都会执行，确保文件被关闭。

`try/except` 语句可以捕获和处理程序运行时可能发生的异常。当需要捕获多个异常时，可以在单个 `except` 块中指定多个异常类型，或为每种异常类型定义独立的 `except` 块。

1. 在单个 except 块中捕获多个异常

可以在一个 `except` 块中通过元组指定多个异常类型。当 `try` 块中的代码引发这些异常之一时，程序将执行该 `except` 块。

```
try:
    # 可能引发异常的代码
except (TypeError, ValueError) as e:
    print(f" 捕获到异常: {e}")
```

以下示例演示了如何在单个 `except` 块中捕获 `TypeError` 和 `ValueError` 异常：

```python
def process_data(data):
    try:
        # 尝试将数据转换为整数
        number = int(data)
        # 执行除法操作
        result = 10 / number
    except (ValueError, ZeroDivisionError) as e:
        print(f" 处理数据时发生错误: {e}")
    else:
        print(f" 结果是: {result}")

# 测试代码
process_data("abc")   # 将引发 ValueError
process_data("0")     # 将引发 ZeroDivisionError
process_data("5")     # 正常处理
```

2. 在多个 except 块中捕获不同的异常

如果需要对不同类型的异常执行不同的处理操作，可以为每种异常类型定义独立的 except 块。

```
try:
    # 可能引发异常的代码
except TypeError as e:
    print(f" 捕获到 TypeError: {e}")
except ValueError as e:
    print(f" 捕获到 ValueError: {e}")
```

下例演示了为 `TypeError` 和 `ValueError` 定义独立的 `except` 块：

```
def add_numbers(a, b):
    try:
    # 如果参数中有非数值类型，就会在比较运算和加法运算时触发
    ↪ TypeError
    # 如果参数小于 0，这里我们人为抛出 ValueError
        if a < 0 or b < 0:
            raise ValueError(" 参数不能是负数")
        result = a + b
    except TypeError as e:
        print(f" 类型错误: {e}")
    except ValueError as e:
        print(f" 数值错误: {e}")
    else:
        print(f" 结果是: {result}")

# 测试代码
add_numbers(5, "10")    # 将引发 TypeError
add_numbers(-3, 10)     # 将引发 ValueError
add_numbers(5, 10)      # 正常处理
```

在上述代码中，`add_numbers` 函数首先检查参数值，如果发现有负数就抛出 `ValueError`；若参数类型不匹配，则在比较运算或加法运算时会触发 `TypeError`。通过分别为 `TypeError` 和 `ValueError` 定义 `except` 块，可以对不同的异常类型执行特定的处理操作。

THE END