14.8 创建自定义异常¶

问题¶

在你构建的应用程序中，你想将底层异常包装成自定义的异常。

解决方案¶

创建新的异常很简单——定义新的类，让它继承自 `Exception` （或者是任何一个已存在的异常类型）。 例如，如果你编写网络相关的程序，你可能会定义一些类似如下的异常：

```
class NetworkError(Exception):
    pass

class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

然后用户就可以像通常那样使用这些异常了，例如：

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

讨论¶

自定义异常类应该总是继承自内置的 `Exception` 类， 或者是继承自那些本身就是从 `Exception` 继承而来的类。 尽管所有类同时也继承自 `BaseException` ，但你不应该使用这个基类来定义新的异常。 `BaseException` 是为系统退出异常而保留的，比如 `KeyboardInterrupt` 或 `SystemExit` 以及其他那些会给应用发送信号而退出的异常。 因此，捕获这些异常本身没什么意义。 这样的话，假如你继承 `BaseException` 可能会导致你的自定义异常不会被捕获而直接发送信号退出程序运行。

在程序中引入自定义异常可以使得你的代码更具可读性，能清晰显示谁应该阅读这个代码。 还有一种设计是将自定义异常通过继承组合起来。在复杂应用程序中， 使用基类来分组各种异常类也是很有用的。它可以让用户捕获一个范围很窄的特定异常，比如下面这样的：

```
try:
    s.send(msg)
except ProtocolError:
    ...
```

你还能捕获更大范围的异常，就像下面这样：

```
try:
    s.send(msg)
except NetworkError:
    ...
```

如果你想定义的新异常重写了 `__init__()` 方法， 确保你使用所有参数调用 `Exception.__init__()` ，例如：

```
class CustomError(Exception):
    def __init__(self, message, status):
        super().__init__(message, status)
        self.message = message
        self.status = status
```

看上去有点奇怪，不过Exception的默认行为是接受所有传递的参数并将它们以元组形式存储在 `.args` 属性中. 很多其他函数库和部分Python库默认所有异常都必须有 `.args` 属性， 因此如果你忽略了这一步，你会发现有些时候你定义的新异常不会按照期望运行。 为了演示 `.args` 的使用，考虑下下面这个使用内置的 *RuntimeError* `异常的交互会话， 注意看raise语句中使用的参数个数是怎样的：

```
>>> try:
...     raise RuntimeError('It failed')
... except RuntimeError as e:
...     print(e.args)
...
('It failed',)
>>> try:
...     raise RuntimeError('It failed', 42, 'spam')
... except RuntimeError as e:

...     print(e.args)
...
('It failed', 42, 'spam')
>>>
```

关于创建自定义异常的更多信息，请参考`Python官方文档 <https://docs.python.org/3/tutorial/errors.html>`_