← Ruby Dir 类和方法 Ruby 面向对象 →

Ruby 异常

异常和执行总是被联系在一起。如果您打开一个不存在的文件，且没有恰当地处理这种情况，那么您的程序则被认为是低质量的。

如果异常发生，则程序停止。异常用于处理各种类型的错误，这些错误可能在程序执行期间发生，所以要采取适当的行动，而不至于让程序完全停止。

Ruby 提供了一个完美的处理异常的机制。我们可以在 *begin/end* 块中附上可能抛出异常的代码，并使用 *rescue* 子句告诉 Ruby 完美要处理的异常类型。

语法

```
begin #开始
raise.. #抛出异常
rescue [ExceptionType = StandardException] #捕获指定类型的异常 缺省值是StandardException
$! #表示异常信息
$@ #表示异常出现的代码位置
else #其余异常
..
ensure #不管有没有异常，进入该代码块
end #结束
```

从 *begin* 到 *rescue* 中的一切是受保护的。如果代码块执行期间发生了异常，控制会传到 *rescue* 和 *end* 之间的块。

对于 *begin* 块中的每个 *rescue* 子句，Ruby 把抛出的异常与每个参数进行轮流比较。如果 rescue 子句中命名的异常与当前抛出的异常类型相同，或者是该异常的父类，则匹配成功。

如果异常不匹配所有指定的错误类型，我们可以在所有的 *rescue* 子句后使用一个 *else* 子句。

实例

```
#!/usr/bin/ruby
begin
file = open("/unexistant_file")
if file
puts "File opened successfully"
end
rescue
file = STDIN
end
print file, "==", STDIN, "\n"
```

以上实例运行输出结果为。您可以看到，*STDIN* 取代了 *file* ，因为*打开*失败。

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

使用 *retry* 语句

您可以使用 *rescue* 块捕获异常，然后使用 *retry* 语句从开头开始执行 *begin* 块。

语法

```
begin
# 这段代码抛出的异常将被下面的 rescue 子句捕获
rescue
# 这个块将捕获所有类型的异常
retry # 这将把控制移到 begin 的开头
end
```

实例

```
#!/usr/bin/ruby
begin
file = open("/unexistant_file")
if file
puts "File opened successfully"
end
rescue
fname = "existant_file"
retry
end
```

以下是处理流程：

- 打开时发生异常。
- 跳到 rescue。fname 被重新赋值。
- 通过 retry 跳到 begin 的开头。
- 这次文件成功打开。
- 继续基本的过程。

注意：如果被重新命名的文件不存在，本实例代码会无限尝试。所以异常处理时，谨慎使用 *retry*。

使用 *raise* 语句

您可以使用 *raise* 语句抛出异常。下面的方法在调用时抛出异常。它的第二个消息将被输出。

语法

```
raise
或
raise "Error Message"
或
raise ExceptionType, "Error Message"
或
raise ExceptionType, "Error Message" condition
```

第一种形式简单地重新抛出当前异常（如果没有当前异常则抛出一个 RuntimeError）。这用在传入异常之前需要解释异常的异常处理程序中。

第二种形式创建一个新的 *RuntimeError* 异常，设置它的消息为给定的字符串。该异常之后抛出到调用堆栈。

第三种形式使用第一个参数创建一个异常，然后设置相关的消息为第二个参数。

第四种形式与第三种形式类似，您可以添加任何额外的条件语句（比如 *unless*）来抛出异常。

实例

```
#!/usr/bin/ruby
begin
puts 'I am before the raise.'
raise 'An error has occurred.'
puts 'I am after the raise.'
rescue
puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

以上实例运行输出结果为：

```
I am before the raise.
I am rescued.
I am after the begin block.
```

另一个演示 *raise* 用法的实例：

实例

```
#!/usr/bin/ruby
begin
raise 'A test exception.'
rescue Exception => e
puts e.message
puts e.backtrace.inspect
end
```

以上实例运行输出结果为：

```
A test exception.
["main.rb:4"]
```

使用 *ensure* 语句

有时候，无论是否抛出异常，您需要保证一些处理在代码块结束时完成。例如，您可能在进入时打开了一个文件，当您退出块时，您需要确保关闭文件。

ensure 子句做的就是这个。ensure 放在最后一个 rescue 子句后，并包含一个块终止时总是执行的代码块。它与块是否正常退出、是否抛出并处理异常、是否因一个未捕获的异常而终止，这些都没关系，*ensure* 块始终都会运行。

语法

```
begin
#.. 过程
#.. 抛出异常
```

```
rescue
#.. 处理错误
ensure
#.. 最后确保执行
#.. 这总是会执行
end
```

实例

```
begin
raise 'A test exception.'
rescue Exception => e
puts e.message
puts e.backtrace.inspect
ensure
puts "Ensuring execution"
end
```

以上实例运行输出结果为：

```
A test exception.
["main.rb:4"]
Ensuring execution
```

使用 *else* 语句

如果提供了 *else* 子句，它一般是放置在 *rescue* 子句之后，任意 *ensure* 之前。

else 子句的主体只有在代码主体没有抛出异常时执行。

语法

```
begin
#.. 过程
#.. 抛出异常
rescue
#.. 处理错误
else
#.. 如果没有异常则执行
ensure
#.. 最后确保执行
#.. 这总是会执行
end
```

实例

```
begin
# 抛出 'A test exception.'
puts "I'm not raising exception"
rescue Exception => e
puts e.message
puts e.backtrace.inspect
else
```

```
puts "Congratulations-- no errors!"
ensure
puts "Ensuring execution"
end
```

以上实例运行输出结果为：

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

使用 $! 变量可以捕获抛出的错误消息。

Catch 和 Throw

raise 和 rescue 的异常机制能在发生错误时放弃执行，有时候需要在正常处理时跳出一些深层嵌套的结构。此时 catch 和 throw 就派上用场了。

catch 定义了一个使用给定的名称（可以是 Symbol 或 String）作为标签的块。块会正常执行直到遇到一个 throw。

语法

```
throw :lablename
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end
或
throw :lablename condition
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end
```

实例

下面的实例中，如果用户键入 '!' 回应任何提示，使用一个 throw 终止与用户的交互。

实例

```
def promptAndGet(prompt)
print prompt
res = readline.chomp
throw :quitRequested if res == "!"
return res
end
catch :quitRequested do
name = promptAndGet("Name: ")
age = promptAndGet("Age: ")
sex = promptAndGet("Sex: ")
# ..
# 处理信息
end
promptAndGet("Name:")
```

上面的程序需要人工交互，您可以在您的计算机上进行尝试。以上实例运行输出结果为：

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

类 Exception

Ruby 的标准类和模块抛出异常。所有的异常类组成一个层次，包括顶部的 Exception 类在内。下一层是七种不同的类型：

- Interrupt
- NoMemoryError
- SignalException
- ScriptError
- StandardError
- SystemExit

Fatal 是该层中另一种异常，但是 Ruby 解释器只在内部使用它。

ScriptError 和 StandardError 都有一些子类，但是在这里我们不需要了解这些细节。最重要的事情是创建我们自己的异常类，它们必须是类 Exception 或其子代的子类。

让我们看一个实例：

实例

```
class FileSaveError < StandardError
attr_reader :reason
def initialize(reason)
@reason = reason
end
end
```

现在，看下面的实例，将用到上面的异常：

实例

```
File.open(path, "w") do |file|
begin
# 写出数据 ...
rescue
# 发生错误
raise FileSaveError.new($!)
end
end
```

在这里，最重要的一行是 raise *FileSaveError.new($!)*。我们调用 raise 来示意异常已经发生，把它传给 FileSaveError 的一个新的实例，由于特定的异常引起数据写入失败。

← Ruby Dir 类和方法 Ruby 面向对象 →

点我分享笔记