Ruby 中的 Block、Proc 和 Lambda

黄京

May 21, 2025

Ruby 作为一门兼具面向对象与函数式特性的语言，其代码块（Block）、过程对象（Proc）和匿名函数（Lambda）是构建灵活代码逻辑的三大核心工具。理解它们的区别不仅有助于避免常见的编程陷阱，更能解锁诸如闭包封装、延迟执行等高阶技巧。本文将通过语法解析、行为对比和实战示例，揭示三者的本质差异与最佳实践。

1 基础概念与语法

1.1 Block（代码块）

Block 是 Ruby 中最基础的匿名代码单元，必须依附于方法调用存在。其语法表现为 `do...end` 或 `{ ... }` 包裹的代码段，隐式参数通过 `|x|` 声明。例如：

```
[1, 2, 3].each { |x| puts x * 2 }
```

这里 `{ |x| puts x * 2 }` 作为 Block 传递给 `each` 方法，通过 `yield` 关键字触发执行。Block 的局限性在于无法独立存储或复用，其生命周期严格绑定于所属方法调用。

1.2 Proc（过程对象）

Proc 将代码块对象化，允许存储和延迟执行。通过 `Proc.new` 或 `proc` 创建，使用 `call` 方法显式调用：

```
my_proc = Proc.new { |x| x + 1 }
puts my_proc.call(3) # => 4
```

Proc 的闭包特性使其能捕获定义时的上下文变量。例如，以下代码中的 `factor` 变量会被 Proc 记住：

```
def multiplier(factor)
  Proc.new { |x| x * factor }
end
double = multiplier(2)
puts double.call(5) # => 10
```

1.3 Lambda（匿名函数）

Lambda 是参数严格的 Proc 变体，通过 `lambda` 或 `->` 语法定义：

```ruby
my_lambda = -> (x) { x * 2 }
puts my_lambda.call(5) # => 10
```

与 Proc 的关键区别在于参数检查机制。调用 Lambda 时参数数量不匹配会抛出 ArgumentError，而 Proc 会静默忽略异常。

2 核心区别对比

2.1 参数处理的严格性

Lambda 要求参数数量精确匹配。例如，以下代码会因参数错误而失败：

```ruby
strict = ->(x) { x * 2 }
strict.call(1, 2) # ArgumentError: wrong number of arguments (2 for 1)
```

而 Proc 会忽略多余参数，缺失参数则赋值为 nil：

```ruby
flexible = Proc.new { |x| x || 0 }
puts flexible.call # => 0 (x 为 nil)
```

2.2 控制流行为的差异

Lambda 中的 return 仅退出自身，而 Proc 的 return 会直接结束外围方法：

```ruby
def test_flow
  lambda { return 1 }.call # 返回 1
  proc { return 2 }.call # 直接结束方法，返回 2
  return 3 # 不会执行到此
end
puts test_flow # => 2
```

此特性使得 Proc 不适合在需要局部退出的场景中使用。

2.3 对象化与复用能力

Block 作为一次性代码片段，无法直接存储复用。而 Proc 和 Lambda 作为对象，可赋值给变量或作为参数传递。例如，动态生成多个计算器：

```ruby
adders = (1..3).map { |n| ->(x) { x + n } }
puts adders[0].call(5) # => 6 (1+5)
```

2.4 闭包特性的共性

三者均具备闭包能力，能够捕获定义时的局部变量。以下示例中，counter 变量被 Proc 捕获并修改：

```
def create_counter
  count = 0
  Proc.new { count += 1 }
end
counter = create_counter
puts counter.call # => 1
puts counter.call # => 2
```

3 应用场景与实战示例

3.1 Block 的典型场景

Block 天然适用于迭代器和资源管理。例如，File.open 方法通过 Block 确保文件自动关闭：

```
File.open("data.txt") do |file|
  puts file.read
end # 自动关闭文件句柄
```

在 Rails 路由 DSL 中，Block 用于声明式配置：

```
Rails.application.routes.draw do
  resources :users do
    collection { get 'search' }
  end
end
```

3.2 Proc 的适用场景

Proc 适用于需要动态绑定上下文的场景。例如，通过 instance_eval 将 Proc 绑定到特定对象：

```
class Widget
  def initialize(&block)
    instance_eval(&block)
  end
  def title(text)
    @title = text
  end
end

widget = Widget.new { title "Demo" }
puts widget.instance_variable_get(:@title) # => "Demo"
```

3.3 Lambda 的适用场景

Lambda 的参数严格性使其适合函数式编程。例如，在 reduce 操作中进行类型安全计算：

```
numbers = [1, 2, 3]
summer = ->(acc, x) { acc + x }
puts numbers.reduce(0, &summer) # => 6
```

4 常见陷阱与最佳实践

Proc 中误用 return 是常见错误来源。例如，在回调中意外终止主流程：

```
def process_data(data, &callback)
  data.each { |item| callback.call(item) }
end

dangerous = Proc.new { |x| return if x.zero?; puts x }
process_data([0, 1, 2], &dangerous) # 抛出 LocalJumpError
```

最佳实践包括：优先选用 Lambda 以提高代码可预测性；利用 & 操作符在 Block 和 Proc 间转换：

```
def wrap_block(&block)
  block.call
end
wrap_block { puts "转换为␣Proc␣执行" }
```

Block、Proc 和 Lambda 的差异本质在于对象化程度、参数处理策略和控制流行为。在需要严格参数检查时选择 Lambda，在需要灵活闭包时使用 Proc，而在临时迭代场景中直接用 Block。理解这些特性后，开发者可以更精准地根据场景选择工具，编写出既简洁又健壮的 Ruby 代码。