好处不止一点点编程结构--闭包 PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/145/2021_2022__E5_A5_BD_E5_A4_84_E4_B8_8D_E6_c104_145035.htm 闭包是可以用作函数参数和方法参数的代码块。一直以来，这种编程结构都是一些语言（如 Lisp、Smalltalk 和 Haskell）的重要组成部分。尽管一些颇具竞争力的语言（如 C#）采纳了闭包，但 Java 社区至今仍抵制对它的使用。本文探讨闭包在为编程语言带来一点点便利的同时是否也带来不必要的复杂性、闭包还有无更多的益处。 10 年前，我刚刚开始山地自行车运动的时候，我更愿意选用零件尽可能少尽可能简单的自行车。稍后，我意识到一些零件（如后减震器）可以保护我的背部和我自行车的框架在德克萨斯州高低起伏的山区中免受损害。我于是可以骑得更快，出问题的次数也渐少。虽然随之带来了操作上的复杂性和维护需求的增加，但对于我来说这点代价还是值得的。 关于本系列在 跨越边界系列 文章中，作者 Bruce Tate 提出这样一种观点，即当今的 Java 程序员们通过学习其他方法和语言很好地武装了自己。自从 Java 技术明显成为所有开发项目的最佳选择以来，编程前景得以改变。其他框架影响着 Java 框架的构建方式，您从其他语言中学到的概念也可以影响 Java 编程。您编写的 Python（或 Ruby、Smalltalk 等语言）代码可以改变编写 Java 代码的方式。 本系列介绍与 Java 开发完全不同的编程概念和技术，但是这些概念和技术也可以直接应用于 Java 开发。在某些情况下，需要集成这些技术来利用它们。在其他情况下，可以直接应用这些概念。具体的工具并不那么重要，重要的是其他语言和框架可以影

响 Java 社区中的开发人员、框架，甚至是基本方式。 关于闭包这个问题，Java 爱好者们现在陷入了类似的争论中。一些人认为闭包带给编程语言的额外复杂性并不划算。他们的论点是：为了闭包带来的一点点便利而打破原有语法糖的简洁性非常不值得。其他一些人则认为闭包将引发新一轮模式设计的潮流。要得到这个问题的最佳答案，您需要跨越边界，去了解程序员在其他语言中是如何使用闭包的。 Ruby 中的闭包闭包是具有闭合作用域 的匿名函数。下面我会详细解释每个概念，但最好首先对这些概念进行一些简化。闭包可被视作一个遵循特别作用域规则且可以用作参数的代码块。我将使用 Ruby 来展示闭包的运行原理。用 irb 命令启动解释程序，然后用 load filename 命令加载每个样例。清单 1 是一个最简单的闭包： 清单 1. 最简单的闭包 3.times {puts "Inside the times method."}Results:Inside the times method.Inside the times method.Inside the times method. times 是作用在对象 3 上的一个方法。它执行三次闭包中的代码。{puts "Inside the times method."} 是闭包。它是一个匿名函数，times 方法被传递到该函数，函数的结果是打印出静态语句。这段代码比实现相同功能的 for 循环（如清单 2 所示）更加紧凑也更加简单： 清单 2: 不含闭包的循环 for i in 1..3 puts "Inside the times method."end Ruby 添加到这个简单代码块的第一个扩展是一个参数列表。方法或函数可通过传入参数与闭包通信。在 Ruby 中，使用在 || 字符之间用逗号隔开的参数列表来表示参数，例如 |argument, list|。用这种方法使用参数，可以很容易地在数据结构（如数组）中构建迭代。清单 3 显示了在 Ruby 中对数组进行迭代的一个例子： 清单 3. 使用了集合的闭包 [lions, tigers,

bears].each {|item| puts item}Results: lionstigersbears each 方法用来迭代。您通常想要用执行结果生成一个新的集合。在 Ruby 中，这种方法被称为 collect。您也许还想在数组的内容里添加一些任意字符串。清单 4 显示了这样的一个例子。这些仅仅是众多使用闭包进行迭代的方法中的两种。 清单 4. 将参数传给闭包 animals = [lions, tigers, bears].collect {|item| item.upcase}puts animals.join(" and ") " oh, my."LIONS and TIGERS and BEARS oh, my. 在清单 4 中，第一行代码提取数组中的每个元素，并在此基础上调用闭包，然后用结果构建一个集合。第二行代码将所有元素串接成一个句子，并用 " and " 加以分隔。到目前为止，介绍的还都是语法糖而已。所有这些均适用于任何语言。 到目前为止看到的例子中，匿名函数都只不过是一个没有名称的函数，它被就地求值，基于定义它的位置来决定它的上下文。但如果含闭包的语言和不含闭包的语言间惟一的区别仅仅是一点语法上的简便 即不需要声明函数 那就不会有如此多的争论了。闭包的好处远不止是节省几行代码，它的使用模式也远不止是简单的迭代。 闭包的第二部分是闭合的作用域，我可以用另一个例子来很好地说明它。给定一组价格，我想要生成一个含有价格和它相应的税金的销售-税金表。我不想将税率硬编码到闭包里。我宁愿在别处设置税率。清单 5 是可能的一个实现： 清单 5. 使用闭包构建税金表 tax = 0.08prices = [4.45, 6.34, 3.78]tax_table = prices.collect {|price| {:price => price, :tax => price * tax}}tax_table.collect {|item| puts "Price: #{item[:price]} Tax: #{item[:tax]}"}Results:Price: 4.45 Tax: 0.356Price: 6.34 Tax: 0.5072Price: 3.78 Tax: 0.3024 在讨论作用域前，我要介绍两个

Ruby 术语。首先，symbol 是前置有冒号的一个标识符。可抽象地把 symbol 视为名称。:price 和 :tax 就是两个 symbol。其次，可以轻易地替换字符串中的变量值。第 6 行代码的 puts "Price: #{item[:price]} Tax: #{item[:tax]}" 就利用了这项技术。现在，回到作用域这个问题。 请看清单 5 中第 1 行和第 4 行代码。第 1 行代码为 tax 变量赋了一个值。第 4 行代码使用该变量来计算价格表的税金一栏。但此项用法是在一个闭包里进行的，所以这段代码实际上是在 collect 方法的上下文中执行的！现在您已经洞悉了闭包 这个术语。定义代码块的环境的名称空间和使用它的函数之间的作用域本质上是一个作用域：该作用域是闭合的。这是个基本特征。这个闭合的作用域是将闭包同调用函数和定义它的代码联系起来的纽带。用闭包进行定制您已经知道如何使用现成的闭包。Ruby 让您也可以编写使用自己的闭包的方法。这种自由的形式意味着 Ruby API 的代码会更加紧凑，因为 Ruby 不需要在代码中定义每个使用模型。您可以根据需要通过闭包构建自己的抽象概念。例如，Ruby 的迭代器数量有限，但该语言没有迭代器也运行得很好，这是因为可以通过闭包在代码中构建您自己的迭代概念。 要构建一个使用闭包的函数，只需要使用 yield 关键字来调用该闭包。清单 6 是一个例子。paragraph 函数提供第一句和最后一句输出。用户可以用闭包提供额外的输出。 清单 6. 构建带有闭包的方法 def paragraph puts "A good paragraph should have a topic sentence." yield puts "This generic paragraph has a topic, body, and conclusion."endparagraph {puts "This is the body of the paragraph."}Results:A good paragraph should have a topic sentence.This is the body of the paragraph. This generic paragraph

has a topic, body, and conclusion.优点通过将参数列表附加给 yield，很容易利用定制闭包中的参数，如清单 7 中所示。 清单 7. 附加参数列表 def paragraph topic = "A good paragraph should have a topic sentence, a body, and a conclusion. " conclusion = "This generic paragraph has all three parts." puts topic yield(topic, conclusion) puts conclusionendt = ""c = ""paragraph do |topic, conclusion| puts "This is the body of the paragraph. " t = topic c = conclusionendputs "The topic sentence was: #{t}"puts "The conclusion was: #{c}" 不过，请认真操作以保证得到正确的作用域。在闭包里声明的参数的作用域是局部的。例如，清单 7 中的代码可以运行，但清单 8 中的则不行，原因是 topic 和 conclusion 变量都是局部变量：清单 8. 错误的作用域 def paragraph topic = "A good paragraph should have a topic sentence." conclusion = "This generic paragraph has a topic, body, and conclusion." puts topic yield(topic, conclusion) puts conclusionendmy_topic = ""my_conclusion = ""paragraph do |topic, conclusion| # these are local in scope puts "This is the body of the paragraph. " my_typic = topic my_conclusion = conclusionendputs "The topic sentence was: #{t}"puts "The conclusion was: #{c}" 闭包的应用下面是一些常用的闭包应用：重构 定制 遍历集合 管理资源 实施策略 当您可以用一种简单便利的方式构建自己的闭包时，您就找到了能带来更多新可能性的技术。重构能将可以运行的代码变成运行得更好的代码。大多数 Java 程序员都会从里到外 进行重构。他们常在方法或循环的上下文中寻找重复。有了闭包，您也可以从外到里 进行重构。 用闭包进行定制会有一些惊人之处。清单 9 是

Ruby on Rails 中的一个简短例子，清单中的闭包用于为一个 HTTP 请求编写响应代码。Rails 把一个传入请求传递给控制器，该控制器生成客户机想要的数据（从技术角度讲，控制器基于客户机在 HTTP accept 头上设置的内容来呈现结果）。如果您使用闭包的话，这个概念很好理解。 清单 9. 用闭包来呈现 HTTP 结果 @person = Person.find(id)respond_to do |wants| wants.html { render :action => @show } wants.xml { render :xml => @person.to_xml }end 清单 9 中的代码很容易理解，您一眼就能看出这段代码是用来做什么的。如果发出请求的代码块是在请求 HTML，这段代码会执行第一个闭包；如果发出请求的代码块在请求 XML，这段代码会执行第二个闭包。您也能很容易地想象出实现的结果。wants 是一个 HTTP 请求包装程序。该代码有两个方法，即 xml 和 html，每个都使用闭包。每个方法可以基于 accept 头的内容选择性地调用其闭包，如清单 10 所示： 清单 10. 请求的实现 def xml yield if self.accept_header == "text/xml" end def html yield if self.accept_header == "text/html" end到目前为止，迭代是闭包在 Ruby 中最常见的用法，但闭包在这方面的用法远不止使用集合内置的闭包这一种。想想您每天使用的集合的类型。XML 文档是元素集。Web 页面是特殊的 XML 集。数据库由表组成，而表又由行组成。文件是字符集或字节集，通常也是多行文本或对象的集合。Ruby 在闭包中很好地解决了这几个问题。您已经见过了几个对集合进行迭代的例子。清单 11 给出了一个对数据库表进行遍历的示例闭包： 清单 11. 对数据库的行进行遍历 require mysqldb=Mysql.new("localhost", "root", "password")db.0select_db("database")result = db.query

"0select * from words"result.each {|row| do_something_with_row}db.close 清单 11 中的代码也带出了另一种可能的应用。MySQL API 迫使用户建立数据库并使用 close 方法关闭数据库。实际上可以使用闭包代替该方法来建立和清除资源。Ruby 开发人员常用这种模式来处理文件等资源。使用这个 Ruby API，无需打开或关闭文件，也无需管理异常。File 类的方法会为您处理这一切。您可以使用闭包来替换该方法，如清单 12 所示： 清单 12. 使用闭包操作 File File.open(name) {|file| process_file(f)} 闭包还有一项重大的优势：让实施策略变得容易。例如，若要处理一项事务，采用闭包后，您就能确保事务代码总能由适当的函数调用界定。框架代码能处理策略，而在闭包中提供的用户代码能定制此策略。清单 13 是基本的使用模式： 清单 13. 实施策略 def do_transaction begin setup_transaction yield commit_transaction rescue roll_back_transaction endend Java 语言中的闭包Java 语言本身还没有正式支持闭包，但它却允许模拟闭包。可以使用匿名的内部类来实现闭包。和 Ruby 使用这项技术的原因差不多，Spring 框架也使用这项技术。为保持持久性，Spring 模板允许对结果集进行迭代，而无需关注异常管理、资源分配或清理等细节，从而为用户减轻了负担。清单 14 的例子取自于 Spring 框架的示例宠物诊所应用程序：清单 14. 使用内部类模拟闭包 JdbcTemplate template = new JdbcTemplate(dataSource).final List names = new LinkedList(). template.query("SELECT id,name FROM types ORDER BY name", new RowCallbackHandler() { public void processRow(ResultSet rs) throws SQLException { names.add(rs.getString(1)). } }). 编写清单

14 中的代码的程序员不再需要做如下这些事： 打开联接 关闭联接 处理迭代 处理异常 处理数据库-依赖性问题 程序员们不用再为这些问题烦恼，因为该框架会处理它们。但匿名内部类只是宽泛地近似于闭包，它们并没有深入到您需要的程度。请看清单 14 中多余的句子结构。这个例子中的代码至少一半是支持性代码。匿名类就像是满满一桶冰水，每次用的时候都会洒到您的腿上。多余句子结构所需的过多的额外处理阻碍了对匿名类的使用。您迟早会放弃。当语言结构既麻烦又不好用时，人们自然不会用它。缺乏能够有效使用匿名内部类的 Java 库使问题更为明显。要想使闭包在 Java 语言中实践并流行起来，它必须要敏捷干净。过去，闭包绝不是 Java 开发人员优先考虑的事情。在早期，Java 设计人员并不支持闭包，因为 Java 用户对无需显式完成 new 操作就在堆上自动分配变量心存芥蒂（参见 参考资料）。 如今，围绕是否将闭包纳入到基本语言中存在极大的争议。最近几年来，动态语言（如 Ruby、JavaScript，甚至于 Lisp ）的流行让将闭包纳入 Java 语言的支持之声日益高涨。从目前来看，Java 1.7 最终很可能会采纳闭包。只要不断跨越边界，总会好事连连。