六个编程范型将改变你对编程的看法 原 荐

编辑部的故事 发布于 2017/05/02 20:46 字数 3710 阅读 2.3W 收藏 148 点赞 10 评论 9

每时每刻我都在琢磨一种编程语言所做的一些与众不同的事情，这改变了我对编程的思考。在这篇文章中，我想分享一些我最喜欢的发现。

这不是那种“函数式编程将改变世界”的博客文章：这篇文章的内容会更加深奥。我敢打赌大多数读者都没有听过下面的编程语言和范型，所以我希望你像我一样有很大的兴趣来学习这些新概念。

注意：对于下面的大多数语言我拥有的经验很少：我只是发现它们背后的思想十分有魅力，但对于它们我没有任何专业知识，所以有任何更正和错误请指出。另外，如果你发现这里存在没有提到的任何新的范型和想法，欢迎把它们分享出来。

更新：这篇文章上了 r/programming 和 HN 的首页。感谢反馈，我已经添加了一些修正。

默认支持并发（Concurrent by default）

示例语言：ANI, Plaid

让我们先从改变思维方式开始吧：有一些编程语言默认情况下就是支持并发的。也就是说，每行代码都是并行执行的。

例如，假设你写了三行代码 A，B 和 C：

```
A;
B;
C;
```

在大多数编程语言中，A 会先执行，然后执行 B，最后执行 C。但在像 ANI 这样的语言中，A, B, 和 C 都将同时执行。

ANI 语言中代码行之间的控制流或排序只是代码行之间显式依赖的副作用。例如，如果 B 具有对 A 中定义的变量的引用，则 A 和 C 将同时执行，并且 B 将在 A 完成之后执行。

来看一下 ANI 中的一个例子。如教程中所述，ANI 程序由用于操作流和数据流的“管道”和“锁存器”组成。这种非同一般的语法很难解析，而且这门语言似乎已经死了，不过这些概念还是非常有趣的。

这是 ANI 中的“Hello World”示例：

```
"Hello, World!" ->std.out
```

在 ANI 语法中，我们将 `"Hello, World!"` 对象（一个字符串）发送到 `std.out` 流。如果我们发送另一个字符串到 `std.out` 会怎么样？

```
"Hello, World!" ->std.out
"Goodbye, World!" ->std.out
```

这两行代码并行执行，所以它们可能在控制台以任何顺序结束。现在，看看当我们某行代码中引入一个变量并在之后引用会发生什么情况：

```
s = [string\];
"Hello, World!" ->s;
\s ->std.out;
```

第一行声明了一个名为 `s` 的“锁存器”（锁存器有一点像变量），其中包含一个字符串；第二行将 `"Hello, World!"` 发送到 `s`，第三行“解锁” `s` 并将内容发送到 `std.out`。因此，可以看到 ANI 的隐形程序排序：由于每一行的运行都取决于前一行，因此这里的代码将会按照编写的顺序执行。

Plaid 语言也声称默认情况下支持并发，但使用的是这篇论文中所描述的一种权限模型来构建控制流。Plaid 还探索了其他有趣的概念，例如 Typestate-Oriented Programming（面向类型状态编程），其中状态更改成为语言的重要因素：你定义的对象不再是作为类，而是可以由编译器检查的一系列状态和转换。看起来这十分有趣，正如 Rich Hickey 的 Are we there yet 讲话中所讨论的将时间作为语言结构的首要因素。

Multicore 正处在上升期，并发性仍然比大多数语言更难。ANI 和 Plaid 对于这个可能产生惊人的性能提升的问题提供了一个新的思路；不过问题是“默认支持并行”是否让并发更容易或难以管理。

更新：上面的描述讲解了 ANI 和 Plaid 的基本本质，但我互换地使用了术语“并发”和“并行”，事实上它们具有不同的含义。有关更多信息，请参阅 Concurrency Is Not Parallelism （并发不是并行）这篇文章。

依赖类型（Dependent types）

示例语言：Idris, Agda, Coq

你可能已经习惯 C 和 Java 等语言的类型系统，编译器可以检查一个变量是整数、列表还是字符串。但如果你的编译器可以检查变量是“正整数”、“长度为 2 的列表”，还是“一个回文字符串”，那又会怎么样呢?

这是支持依赖类型的语言背后的思想：你可以在编译时指定检查变量值的类型。Scala 的 shapeless 库增加了对 Scala 依赖类型的部分实验支持（尚未正式支持），并提供了简单的方

法来查看示例。

这里是如何声明一个 `Vector` 的代码，其中使用了 shapeless 库，包含值 1，2，3：

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
```

这里创建了一个变量 `l1`，它的类型签名不仅指定它是一个包含 `Ints` 的 `Vector`，而且还指定它是一个长度为 3 的 `Vector`。编译器可以使用此信息来捕获错误。让我们使用 `Vector` 中的 `vAdd` 方法来执行两个 `Vectors` 之间的成对加法（pairwise addition）：

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
val l2 = 1 :#: 2 :#: 3 :#: VNil

val l3 = l1 vAdd l2

// Result: l3 = 2 :#: 4 :#: 6 :#: VNil
```

上面的例子正常运行，因为类型系统知道两个 `Vectors` 的长度都为 3。然而，如果我们尝试 `vAdd` 两个长度不同的 `Vectors`，我们会在编译时得到一个错误，而不必等到运行时。

```
val l1 = 1 :#: 2 :#: 3 :#: VNil
val l2 = 1 :#: 2 :#: VNil

val l3 = l1 vAdd l2

// Result: a *compile* error because you can't pairwise add vectors
// of different lengths!
```

Shapeless 是一个了不起的库，但就我所看到的，它仍然有点粗糙，只支持依赖类型的一个子集，并导致生成相当详细的代码和类型签名。另一方面，Idris 使类型成为编程语言的首要成员，因此它的依赖类型系统看起来更强大和更干净。为了比较，请查看演讲 — Scala vs Idris: Dependent Types, Now and in the Future 。

形式化验证方法已经存在很长一段时间了，但大多数情况下都十分麻烦，无法用于通用编程。依赖类型在像 Idris 这样的语言中，甚至在未来的 Scala 中，可能会提供更轻量的和更实用的替代方案，这仍然能大大提高类型系统提供捕获错误的能力。当然，没有类型系统可以捕获所有的错

误，由于终止（halting）问题的固有局限性，但如果做得好，依赖类型可能是静态类型系统下一个大的飞跃。

Concatenative 语言

示例语言：Forth, cat, joy

有没有想过在没有变量和函数应用的情况下编程是一种怎样的体验？没有？我也没试过。但显然有些人做了，他们提出了 concatenative 编程这个概念。这个概念背后的思想是语言中的所有内容都是一个函数，用于将数据推送到堆栈或从堆栈弹出数据；程序几乎完全通过功能组合来构建（concatenation is composition）。简单的说即是基于堆栈的编程语言。

这听起来很抽象，所以让我们来看一个简单的例子：

```
2 3 +
```

在这里，我们将两个数字推到堆栈上，然后调用 + 函数，它将两个数字从堆栈中弹出，并将结果添加到堆栈中：代码的输出是 5。这里有一个更有趣的例子：

```
def foo {
  10 <
  [ 0 ]
  [ 42 ]
  if
}

20
foo
```

我们来一行一行看这段代码：

1. 首先，我们声明一个函数 `foo` 。请注意，cat 中的函数不指定输入参数：所有参数从堆栈中隐式读取。
2. `foo` 调用 `<` 函数，它会弹出堆栈中的第一个选项，将其与 10 进行比较，并将 `True` 或 `False` 返回到堆栈。
3. 接下来，我们将值 0 和 42 推到堆栈上：我们将它们放在括号中，以确保它们被压入未被评估的堆栈。这是因为它们将被用作“then”和“else”分支（分别）用于调用下一行的 `if` 函数。
4. `if` 函数从堆栈中弹出 3 个选项：布尔条件，“then”分支和“else”分支。根据布尔条件的值，它会将“then”或“else”分支的结果推回堆栈。
5. 最后，我们将 20 推到堆栈并调用 `foo` 函数。
6. 当所有都完成了，我们最终会得到数字 42。

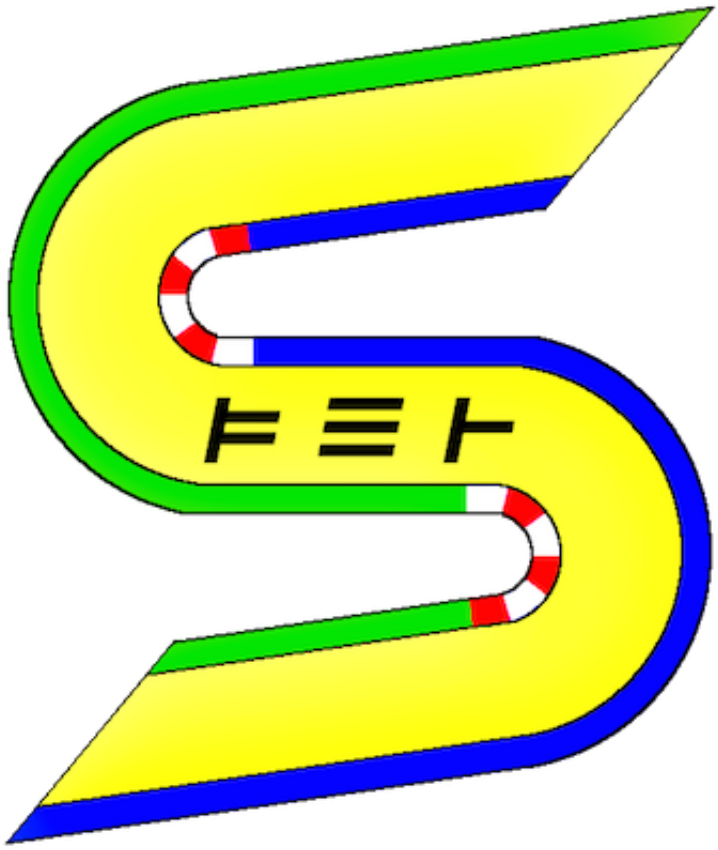
有关这种语言更详细的介绍，请参阅 The Joy of Concatenative Languages。

这种编程风格有一些有趣的属性：

- 程序可通过无数的方式来分割和连接，以创建新的程序；
- 极简的语法（甚至比 LISP 还小）产生了非常简洁的程序；
- 强大的元编程支持

我发现 concatenative 编程是一个非常开眼界的体验，但我还没实践过。似乎你必须记住或想象堆栈的当前状态，而不是能够从代码中的变量名读取它，这可能使得很难理解代码。

声明式编程（Declarative programming）



示例语言：Prolog, SQL

声明式编程已经存在了很多年，但大多数程序员仍不知道它是一个怎样的概念。简要来说：在大多数主流语言中，你是在描述如何解决特定的问题；在声明式语言中，你只需描述所需的结果，语言本身可推导出结果。

例如，如果你在 C 语言中从头开始编写排序算法，你会编写合并排序的说明，逐步描述如何递归地将数据集分为两部分并按顺序将其合并到一起：这里是一个例子。如果使用声明式语言如 Prolog 对数字进行排序，可以直接描述你需要的输出：“我想要相同的值列表，但索引 `i` 中的每个项目应小于或等于索引 `i + 1` 中的项目”。将上面 C 语言的解决方案和 Prolog 代码进行比较：

```
sort_list(Input, Output) :-
  permutation(Input, Output),
  check_order(Output).

check_order([]).
check_order([Head]).
check_order([First, Second | Tail]) :-
  First =< Second,
  check_order([Second | Tail]).
```

如果你使用过 SQL，那么你已经使用了声明式编程，可能自己没有意识到这一点：当你发出一个像 `select X from Y where Z` 这样的查询，你就是在描述你想要返回的数据集；数据库引擎的工作实际上是如何执行查询。你可以在大多数数据库中使用 explain 命令来查看执行计划并弄清楚在引擎下发生了什么。

声明式语言的优点在于它允许你在更高层次的抽象下工作：你的任务就是描述所需输出的规范。例如，在 Prolog 语言实现的一个简单数独解算器中只列出了一个数独谜题的答案的每行、列和对角线应该是怎样的：

```
sudoku(Puzzle, Solution) :-
  Solution = Puzzle,

  Puzzle = [S11, S12, S13, S14,
            S21, S22, S23, S24,
            S31, S32, S33, S34,
            S41, S42, S43, S44],

  fd_domain(Solution, 1, 4),

  Row1 = [S11, S12, S13, S14],
  Row2 = [S21, S22, S23, S24],
  Row3 = [S31, S32, S33, S34],
  Row4 = [S41, S42, S43, S44],

  Col1 = [S11, S21, S31, S41],
  Col2 = [S12, S22, S32, S42],
  Col3 = [S13, S23, S33, S43],
  Col4 = [S14, S24, S34, S44],

  Square1 = [S11, S12, S21, S22],
  Square2 = [S13, S14, S23, S24],
  Square3 = [S31, S32, S41, S42],
  Square4 = [S33, S34, S43, S44],

  valid([Row1, Row2, Row3, Row4,
         Col1, Col2, Col3, Col4,
         Square1, Square2, Square3, Square4]).

valid([]).
valid([Head | Tail]) :- fd_all_different(Head), valid(Tail).
```

下面是如何运行上面的数独解算器：

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).


S = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]
```

不幸的是，声明式编程语言的缺点是性能开销大。上面提到的排序算法时间复杂度可能是 O(n!)；数独解算器使用暴力搜索；而且大多数开发人员不得不提供数据库提示和额外的索引，以避免执行 SQL 查询时开销大且效率低的计划。

符号式编程（Symbolic programming）

示例语言：Aurora

Aurora 语言是符号式编程的一个例子：使用这些语言编写的“代码”不仅可以包含纯文本，还可以包括图像、数学方程、图和图表等。你可以以该数据的原生格式来操作和描述各种大量的数据，而不是全部以文本形式描述。Aurora 也是完全可交互的，它会立即显示每行代码的结果，像 steroids 中的 REPL。

Aurora 语言由 Chris Granger 创建，它还构建了 Light Table IDE。Chris 在它的文章 Toward a better programming（为了更好地编程）中概述了创建 Aurora 的动机：一些目标是使编程更直

观、直接和减少偶然的复杂性。要了解更多的信息，观看 Bret Victor 的演讲：Inventing on Principle, Media for Thinking the Unthinkable, 和 Learnable Programming。

更新：“符号式编程”可能不太适合用于描述 Aurora。有关更多信息，请参阅 Symbolic programming 的维基主页。

基于知识的编程（Knowledge-based programming）

示例：Wolfram Language

很像上面提到的 Aurora 语言，Wolfram 语言也是基于符号的编程。然而，符号层仅仅是为 Wolfram 语言的核心提供一致的接口，Wolfram 语言是基于知识的编程语言：内置了大量的库、算法和数据。这使得可以轻松地从图形化的 Facebook 连接，到操纵图像、查找天气、处理自然语言查询、绘制地图上的方向、求解数学方式等方面做好一切。

我猜想 Wolfram 语言有最大的“标准库”和任何现有语言的数据集。对于 Internet connectivity 是编写代码的固有功能，我也感到十分兴奋：它几乎像一个 IDE，其中的自动完成功能进行谷歌搜索。看符号式编程模型是否像 Wolfram 所说的那样灵活并真正利用所有这些数据，这将是非常有趣的。

更新：尽管 Wolfram 声称 Wolfram 语言支持“符号式编程”和“知识编程”，但这些术语的定义有所不同，有关更详细的信息请参阅 Knowledge level 和 Symbolic Programming 的维基主页。

译自：http://www.ybrikman.com

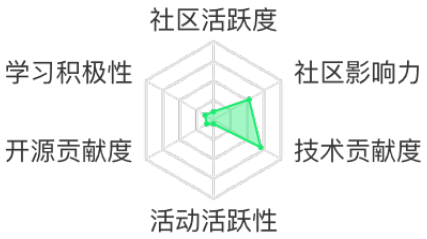
原文链接：https://my.oschina.net/editorial-story/blog/890965
责任编辑：开源中国 – 局长

上一篇：【源资讯 第26期】编程语言的纷争：P... 下一篇：【软件周刊第 26 期】2017 Percona L...

编辑部的故事

粉丝 1663 博文 289 码字总数 602499
作品 0
深圳 运营/编辑

关注 私信 提问



相关文章 最新文章

Netty那点事（三）Channel与Pipeline

Channel是理解和使用Netty的核心。Channel的涉及内容较多，这里我使用由浅入深的介绍方法。在这篇文章中，我们主要介绍Channel部分中Pipeline实现机制。为了避免枯燥，借用一下《盗梦空间》的...

黄亿华 2013/11/24 1.9W 22

Flappy Bird（安卓版）逆向分析（一）

更改每过一关的增长分数 反编译的步骤就不介绍了，我们直接来看反编译得到的文件夹 方法1：在smali目录下，我们看到org/andengine/，可以知晓游戏是由andengine引擎开发的。打开/res/raw/at...

enimey 2014/03/04 5.7K 18

代码生成器--Codgen

Codgen是一个基于数据库元数据模型，使用freemarker模板引擎来构建输出的代码生成器。freemarker的数据模型结构通常来说都是一个Map树状结构模型，codgen也不例外，它的...

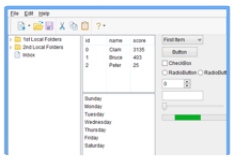
黄天政 2013/01/29 1.4W 2

数据库表单生成器--SQLScreens

SQLScreens 是一个使用 Tcl/TK 编写的简单关系型数据库表单生成工具。可让你快速创建查询界面，并指定相应的表和字段。支持多种数据库，包括：MySQL, SQLite, and INFORMI...

匿名 2013/02/17 823 0

Flash 皮肤样式--Windows8UIStyle



Windows8UIStyle 模仿 Windows 8 的桌面用户界面，使得 FlashSwing 应用程序在 Windows 8 系统中拥有与传统应用程序一致的用户界面。 Windows8UIStyle 对 FlashSwin...

Gregary 2013/02/19 1.3K 1

加载更多