玩具：各种语言实现累加器

保尔·格雷厄姆在《书呆子的复仇》一文中为了讨论不同语言编程能力的差别，使用了一个累加器的例子。这部分内容相当有趣，但这篇文章有些年头了。本文会将保尔·格雷厄姆原文的大部分内容重新展示，并结合更新的技术给出更多的例子和自己的一点看法。

玩具：累加器生成函数

我们要编写一个函数（或者对象、类，只要能实现我们的需求就行），它接受一个参数 n，然后返回另一个函数，后者接受参数 i，n 在原先的基础上增加 i，然后这个函数返回增加之后的值。

比如这个函数是 foo, 我们需要的结果是：

```
f = foo(5)
f(4) // ⇒ 9
f(10) // ⇒ 19
f(1) // ⇒ 20
```

Lisp家族

Lisp 家族对闭包的天然支持（除了最古老的 Lisp）使得用 Lisp 来写累加器生成函数非常容易。以下三个函数的写法均来自格雷厄姆的原文：

Common Lisp:

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

Scheme:

```
(define (foo n)
  (lambda (i) (set! n (+ n i)) n))
```

Arc:

```
(def foo (n) [++ n _])
```

这里我省去了原文中 Goo 语言的例子，因为在今天看来，Goo 的存在与否似乎没人关心了。这样说的话 Arc 好像也没有多少人关心，但它毕竟是保尔的孩子，为了表示尊敬，我把它留在了这里。Racket 的写法和 Scheme 基本一致，用不着浪费篇幅再讨论了。相比之下，可能接触和使

用 Clojure 的人更多。Clojure 不推荐随便改变状态，所以要写一个这样的函数有些麻烦，但不是完全不能做：

```
(defn foo [n]
  (let [acc (atom n)]
    (fn [i] (swap! acc + i))))
```

和 Clojure 实例一样，后面由我自己编写的代码都是验证正确的。用 Haskell 这样的纯函数式语言要怎样完成这个任务？想要像上面的例子中这样得到一个干净的结果是做不到的，但总可以把累加结果和累加器封装起来，再编写另一个函数将我们需要的结果数值提取。

Ruby, Perl 5, Smalltalk

Ruby:

```
def foo(n)
  lambda {|i| n += i }
end
```

相比较闭包和高阶函数，Ruby 对面向对象的关照明显更多。尽管如此，完成累加器这样的任务对于 Ruby 来说要轻而易举。

Perl 5:

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

> *这比 Lisp 和 Ruby 的版本有更多的语法元素，因为在 Perl 语言中必须手工提取参数。*

Smalltalk:

```
foo: n
  |s|
  s := n.
  ^[:i| s := s+i. ]
```

Smalltalk 无法给参数赋值，这一点和后面我们将会在 Python 和 Kotlin 中看到的情况相似。

JavaScript

JavaScript 是一门有趣的语言，它从 Scheme 那里搞来了 first-class function 且支持闭包。JavaScript 是单线程的，所以捕获的变量能放心地修改。但同时 JavaScript 却明显地区分语句和表达式。JavaScript 编写的累加器生成函数将是这样：

```
function foo(n) {
  return function(i) {
    return n += i;
  };
}
```

现在，ES2015 支持胖箭头函数了，一定会有人想这样写：

```
const foo = n => i => n += i;
```

Python

《书呆子的复仇》原文中给出了三种 Python 的实现方法：

```
def foo(n):
  s = [n]
  def bar(i):
    s[0] += i
    return s[0]
  return bar
```

```
def foo(n):
  class acc:
    def __init__(self, s):
      self.s = s
    def inc(self, i):
      self.s += i
      return self.s
  return acc(n).inc
```

```
class foo:
  def __init__(self, n):
    self.n = n
  def __call__(self, i):
    self.n += i
    return self.n
```

原文中针对这些写法，保尔给出了自己的观点。我们只关注第一种写法，因为第二种和第三种做法没有体现 Python 的能力（如果不谈及括号操作符重载），事实上要程序员亲自创建对象来保存 n 的值。相比起来，第一种写法更符合我们的要求。

如今，Python 中有了 nonlocal 关键字，我们不必要创建 list 来保存变量了：

```
def foo(n):
  def bar(i):
    nonlocal n
    n += i
    return n
  return bar
```

保尔当时预言了 Python 对 lambda 的支持，然而我们现在看到的是，Python 中加入的 lambda 仅仅是匿名函数，而且有很多限制。大家可以尝试用 lambda 编写这个累加器生成函数，不是完全不能做，但没有上面这种写法优雅。

C家族

这里提到 C 家族，我有意指那些结构化的、编译执行的、静态类型的、不支持闭包的编程语言。除了 C, C++, Java, C#（如果不考虑 Java 和 C# 的反射能力）可能还包括 VB, Fortran 等。当然，时代在进步，随着一些语言新特性的出现，不可能说不定变成了可能。下面的例子中，可能会包含一些编程“黑魔法”，你尽量不要在你的程序中使用，我写出它们只是为了表示有些事情是可行的。

考虑到它们是静态类型语言，我们放松问题的要求，只需要得到一个整数累加器就够了。

Java 和 C++ 都不能胜任这个工作。当然，你总是可以写出像上面 Python 的那种手动保存 n 值的写法，尤其是 C++ 还允许操作符重载。有趣的是，Java SE 8 和 C++ 11 都开始支持 lambda 表达式，但它们都和上面所说的 Python 中的 lambda 差不多，没有语言能力层面的增强。同理，gcc 曾经为 C 语言做的嵌套函数扩展也是一样，虽然看起来好像可以和 Python 中在函数内定义函数，同时捕获外面的变量，再把函数指针返回，然而 C 终究不会自动管理内存，如果你像我上面所说的这样尝试，就会得到混乱的结果，因为在退出函数作用域时，局部变量的栈内存也被回收了。

Clang 为 C, C++, Obj-C, Obj-C++ 做了一个名为 Blocks 的扩展，它看起来和闭包很像。事实上，它就有闭包的作用：

```
typedef int (^inttoint)(int);
inttoint foo(int n) {
    __block int _n = n;
    return Block_copy(^int(int i) {
        return _n += i;
```

```
    });
}
```

（如果你手头上有 Clang 编译器，可以尝试编译上述代码并运行，注意要添加预处理指令 `#include <Block.h>`。）

虽然我们还是需要手动捕获变量，但它已经和上面提到的第一个 Python 写法很相近了。这种手法在 Obj-C 里使用得比较多，一般应当没有程序员真正会在 C/C++ 项目中使用这样的扩展。

相比较 C++, Java 对 lambda 表达式的限制，C# 表现得宽松得多。用 C# 可以轻松完成任务：

```
public static Func<int, int> Foo(int n)
{
    return (int i) => n += i;
}
```

Go, Kotlin, Rust

这些现代语言也是静态类型的，但它们对闭包这样的经典语言特性都有很好的支持。

Go:

```
func foo(n int) func(int) int {
    return func(i int) int {
        n += i
        return n
    }
}
```

Kotlin 和 Swift 的写法差不多:

```
val foo: (Int) -> (Int) -> Int = { n -> var _n = n; { i -> _n += i; _n} }
```

早期的 Kotlin 是可以修改函数参数的，但现在参数都是 val 了，所以需要手动保存。

Rust 需要考虑到所有权问题，所以会复杂一些:

```
let foo = |mut n: i32| { move |i: i32| { n += i; n } };
```

获胜者是?

显然，这个玩具需求的目的就是为了测试不同语言的能力，选出一个获胜者是必须的。但我不想把事情说得太绝对，毕竟不同的编程语言是为不同场景而生的。就上面的比拼来看，Go 显然比 Java 要好，但我对 Go 的好感要远低于 Java，这背后的原因以后再细谈。

foo 函数的编写需要语言在两个方面的基本支持：嵌套函数捕获外部变量（词法作用域/闭包）和修改外部状态。显然我们的获胜者是 Scheme/Racket, Common Lisp 和 Arc！其次是 JavaScript 和 Ruby！接下来，Perl 5, Go, Rust 和 C# 也能很好地完成任务！Smalltalk, Python, Kotlin, Clojure 虽然需要手动捕获变量，但勉强能实现需求。C/C++ 本体感到吃力，只能依靠 Clang 提供的 Blocks 扩展。垫底选手是 Java, 什么样的黑魔法都拯救不了它不能返回函数的特性。

（如果你喜欢本文，请出门扫码献爱心。本文定价2元）