闭包（整理自Wiki）

在计算机科学中，**闭包**（英语：Closure），又称**词法闭包**（Lexical Closure）或**函数闭包**（function closures），是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

闭包的概念出现于60年代，最早实现闭包的程序语言是Scheme。之后，闭包被广泛使用于函数式编程语言如ML语言和LISP。很多命令式程序语言也开始支持闭包。

在一些语言中，在函数中可以（嵌套）定义另一个函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。运行时，一旦外部的 函数被执行，一个闭包就形成了，闭包中包含了内部函数的代码，以及所需外部函数中的变量的引用。其中所引用的变量称作**上值**(upvalue)。

闭包一词经常和匿名函数混淆。这可能是因为两者经常同时使用，但是它们是不同的概念。

词源

Peter J. Landin 在1964年将术语 闭包 定义为一种包含 环境成分 和 控制成分的实体，用于在他的SECD 机器上对表达式求值。 Joel Moses 认为是 Landin 发明了 闭包 这一术语，用来指代某些其开放绑定（自由变量）已经由其语法环境完成闭合（或者绑定）的 lambda 表达式，从而形成了 闭合的表达式，或称闭包。 这一用法后来于 1975 年被 Sussman 和Steele 在定义 Scheme 语言的时候予以采纳。 并广为流传。

语义

闭包和状态表达

闭包可以用来在一个函数与一组“私有”变量之间创建关联关系。在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。变量的作用域仅限于包含它们的函数，因此无法从其它程序代码部分进行访问。不过，变量的生存期是可以很长，在一次函数调用期间所创建所生成的值在下次函数调用时仍然存在。正因为这一特点，闭包可以用来完成信息隐藏，并进而应用于需要状态表达的某些编程范型中。

不过，用这种方式来使用闭包时，闭包不再具有引用透明性，因此也不再是纯函数。即便如此，在某些“近似于函数式编程语言”的语言，例如Scheme中，闭包还是得到了广泛的使用。

闭包和第一类函数

典型的支持闭包的语言中，通常将函数当作第一类对象——在这些语言中，函数可以被当作参数传递、也可以作为函数返回值、绑定到变量名、就像字符串、整数等简单类型。例如以下Scheme代码：

```
; Return a list  of all books with at least THRESHOLD copies sold.
(define  (best-selling-books  threshold)
   (filter
    (lambda (book) (>= (book-sales book)  threshold))
    book-list))
```

在这个例子中，lambda表达式`(lambda (book) (>= (book-sales book) threshold))`出现在函数`best-selling-books`中。当这个lambda表达式被执行时，Scheme创造了一个包含此表达式以及对`threshold`

变量的引用的闭包，其中`threshold`变量在lambda表达式中是自由变量。

这个闭包接着被传递到`filter`函数。这个函数的功能是重复调用这个闭包以判断哪些书需要增加到列表哪些书需要丢弃。因为闭包中引用了变量`threshold`，所以它在每次被`filter`调用时都可以使用这个变量，虽然`filter`可能定义在另一个文件中。

下面是用ECMAScript (JavaScript)写的同一个例子：

```
// Return a  list of all books with at least 'threshold' copies sold.
function  bestSellingBooks(threshold) {
  return bookList.filter(
      function  (book) { return book.sales >= threshold; }
    );
}
```

这里，关键字`function`取代了`lambda`，`Array.filter`方法取代了`filter`函数，但两段代码的功能是一样的。

一个函数可以创建一个闭包并返回它，如下述JavaScript例子：

```
// Return a function that approximates the derivative of f
// using an interval of dx, which should be appropriately small.
function derivative(f,  dx) {
  return  function (x) {
    return (f(x + dx) - f(x)) / dx;
  };
}
```

因为在这个例子中闭包已经超出了创建它的函数的范围，所以变量`f`和`dx`将在函数`derivative`返回后继续存在。在没有闭包的语言中，变量的生命周期只限于创建它的环境。但在有闭包的语言中，只要有一个闭包引用了这个变量，它就会一直存在。清理不被任何函数引用的变量的工作通常由垃圾回收完成。

闭包的用途

- 因为闭包只有在被调用时才执行操作，即“惰性求值”，所以它可以被用来定义控制结构。例如：在Smalltalk语言中，所有的控制结构，包括分歧条件(if/then/else)和循环(while和for)，都是通过闭包实现的。用户也可以使用闭包定义自己的控制结构。
- 多个函数可以使用一个相同的环境，这使得它们可以通过改变那个环境相互交流。比如在Scheme中：

```
(define foo #f)
(define bar #f)

(let ((secret-message "none"))
  (set! foo (lambda (msg) (set! secret-message msg)))
  (set! bar (lambda () secret-message)))

(display (bar)) ; prints "none"
(newline)
(foo "meet me by the docks at midnight")
```

```
(display (bar)) ; prints "meet me by the docks at midnight"
```

- 闭包可以用来实现对象系统。

闭包的实现

典型实现方式是定义一个特殊的数据结构，保存了函数地址指针与闭包创建时的函数的词法环境表示（那些nonlocal变量的绑定）。使用函数调用栈的语言实现闭包比较困难，因而这也说明了为什么大多数实现闭包的语言是基于垃圾收集机制。

闭包的实现与函数对象很相似。这种技术也叫做lambda lifting。