什么是continuation

与其问什么是 continuation，不如先问什么是控制流。

控制流在各编程语言中都是相当基础的概念，它表示程序接下来应该怎么执行。典型的命令式语言程序中存在条件控制流和循环控制流，部分语言中存在 try-catch 异常捕获机制。它们的实现往往和语言本身的实现有紧密的关系。例如，一个 C 语言的编译器需要为条件表达式单独创建一个 AST，并对这个 AST 进行特有的语义分析（或者进行仅属于条件表达式规则的求值）。C 语言中 continuation 表现为当前状态下处理器里各个寄存器的值和栈内数据。

和以前一样，为了清晰性，我会选择 Scheme 这样易于解释的语言来编写程序作为辅助说明，有时可能也会选择 JavaScript 这种人人都能轻易地看懂而且表达能力强大的语言，或许也会使用 Haskell 这样拥有相对强大类型系统的语言。

作为控制流的抽象

Continuation 是控制流的抽象。在一个表达式求值的过程中，continuation 就是求值到目前为止剩下的部分。

（在经典教材 EOPL 中，这类抽象一般会使用“过程表达”和“数据结构表达”两种形式进行表达。这篇文章中，我们仅使用过程表达，下次有机会或许我可能专门写一篇文章来尝试用数据结构表达来重写一些文章里的示例。）

考虑一个阶乘函数：

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

它的某个求值过程：

```
  (fact 2)
= (* 2 (fact 1))
= (* 2 (* 1 (fact 0)))
= (* 2 (* 1 1))
= (* 2 1)
= 2
```

在求值的第 2 行处，对于`(fact 1)`这个待求值的表达式，continuation 就是：

```
(lambda (result)
  (* 2 result))
```

也就是说，`(fact 1)`求值完毕后，得到一个结果`result`，然后再让 continuation 作用到这个结果之上就得到了整个表达式的值。从求值函数`eval`的角度来看，暴露出 continuation，求值过程就是这样的：

```
  (fact 2)
= ((lambda (r) (* 2 r)) (fact 1))
= ((lambda (r) (* 2 r))
   ((lambda (r) (* 1 r))
    (fact 0)))
```

执行到这一步，`(fact 0)`外面有两个 continuation：一个在求值`(fact 1)`到过程中，`(fact 0)`求值完后要做的事情；另一个是`(fact 1)`求值完后要做的事情。事实上，它们可以看作一个 continutaion，都是`(fact 0)`求值完后要做的事情。也就是说，对参数的求值过程使得 continuation 变大了。

让我们再看看一个暴露 continuation 的条件表达式求值过程：

```
  (if (zero? 0) 1 2)
= ((lambda (r)
     (if r 1 2)
   (zero? 0)))
= ((lambda (r)
     (if r 1 2))
   #t)
= (if #t 1 2)
= 1
```

我想我已经表达清楚 continuation 的概念了。那么它到底有什么用呢？解释器直接对表达式求值和先构造其 continuation 再 apply 有什么区别呢？

call/cc

当 continuation-passing interpreter 对表达式求值时，构造了表达式的 continuation，并通过把它作为参数传递再进行求值。这样做的一个很大好处是增加解释器的灵活性，使得解释器可以面对更多其它的控制流。还有一个好处，就是可以把 continuation 暴露给程序员！

call/cc 的含义是 call with current continuation，是 Scheme 中用来操作 continuation 的函数。例如：

```
(* 2 (call/cc
      (lambda (cc)
        (cc (fact 1)))))
```

的值是 2。这个求值过程很奇妙，它不同于我们接触到的其它函数调用的求值。含 call/cc 表达式的求值规则是这样：

call/cc 接受一个函数作为参数，该函数的参数我们统一命名为 cc，这个 cc 就是 call/cc 表达式的 continuation。如果函数内部出现了 cc 调用，那么这个调用的结果就是整个表达式的结果；如果没有出现 cc 调用，那么 call/cc 的调用结果就像普通函数一样放回整个表达式继续求值。

对于上面例子里的这个表达式，求值过程是：

```
  (* 2 (call/cc
       (lambda (cc)
         (cc (fact 1)))))
= (let ([cc
         (lambda (x)
           (* 2 x))])
    (cc (fact 1)))
= ...  ;; (fact 1) = 1
= (* 2 1)
= 2
```

当 call/cc 表达式里出现了 cc 调用，那么外面的`(* 2 ...)`这个乘法就作为 continuation 变成 cc 了，永远不会执行了。这里只体现了 cc 调用的情况，下面这个例子中，call/cc 内部没有 cc 调用，call/cc 调用的结果就是整个表达式的结果：

```
  (* 2 (call/cc
       (lambda (cc)
         1)))
= (* 2 1)
= 2
```

结果仍然是 2。

到此为止，我们还是不知道 call/cc 有什么用途，似乎

```
( ... (call/cc (lambda (cc) (cc expr))) ... )
```

和

```
( ... expr ... )
```

没什么区别，除了多打了很多字和括号。

在 Scheme 中，continuation 是 first-class 的，像函数一样可以赋值给一个变量或者当作参数传递。在上面 call/cc 的例子中，我们已经看到，call/cc 表达式内的 cc 就是 continuation 的抽象，因此我们除了可以把它当作函数直接调用，还可以把它赋值给一个变量：

```
(define cont 0)

(* 2 (call/cc
      (lambda (cc)
        (set! cont cc)
        (cc (fact 1))))) ;; ⇒ 2

(cont 20) ;; ⇒ 40
```

在第 5 行，我们把 cc 赋值给了 cont，这样 cont 就成为了

```
(lambda (x) (* 2 x))
```

我们可以在任何再需要它的时候进行调用。因此第 8 行调用的结果是`(* 2 20) = 40`。

Continuation-Passing Style

以下简称 CPS。CPS 是一种编程风格，这种风格的函数将 continuation 作为参数，并在原本的结果上调用 continuation。以下是一个例子：

```
(define add
  (lambda (x y)
    (+ x y)))

(define add/k
  (lambda (x y cont)
    (cont (+ x y))))
```

这里，`add/k`就是`add`的 CPS 版本。使用 NodeJS 写过异步程序的程序设计师可能熟悉这种风格。以下是上述代码的 ECMAScript 版本：

```
function add_k(x, y, cont) {
  return cont(x + y);
```

```
}
```

如果我们想表达如下一段顺序逻辑：

```
function add(x, y) {
    return x + y;
}

function double(x) {
    return x * 2;
}

console.log(add(double(3), 10)); // ⇒ 16
```

CPS 版本就是：

```
function add_k(x, y, cont) {
    return cont(x + y);
}

function double_k(x, cont) {
    return cont(x * 2);
}

double_k(3, (x) ⇒ add_k(x, 10, console.log)); // ⇒ 16
```

这里为了简便，没有把 `console.log` 函数进行 CPS 变换，否则结果应当是这样的：

```
function id(x) {
  return x;
}

double_k(3, (x) ⇒ add_k(x, 10, (x) ⇒ console.log_k(x, id))); // ⇒ 16
```

这里用了一个 `id` 函数来表示 continuation 的终点，也就是整个程序的终点。

如果前面关于 continuation 的讲解你看懂了，这个程序理解起来应该没有什么困难。为了异步编程，NodeJS 经常会使用这样的风格，缺点就是它往往会导致回调地狱，写出的代码难读难维护。

关于 CPS 和 call/cc 暂时还不能说太多，下次谈到 Monad 和类型系统时或许还可以再谈谈这个问题。

（本文定价1元）